Formal Verification Framework for Hardware Accelerator Designs

Kevin Bhensdadiya, Anmol Patel, Anshul Jain
{kevin.bhensdadiya, anmol.patel, anshul.jain}@intel.com
Intel Corporation, India

***Abstract*- This paper presents a novel framework for verifying hardware accelerators using formal verification techniques, addressing the unique challenges posed by these specialized computing engines. Hardware accelerators have emerged as a solution to improve performance and efficiency in various compute-intensive tasks, outperforming general-purpose programmable processors. However, their lack of precise specifications and the wide variety of configurations pose significant challenges in pre-silicon verification. Our proposed framework tackles these issues and streamlines the verification process. A case study demonstrates the successful application of this framework to a deflate algorithm accelerator, showcasing its effectiveness and highlighting noteworthy results.**

I. Introduction

Traditionally, most of the computing tasks run on general-purpose programmable processors called CPUs. CPUs have been the main workhorse because they are easy to program – programmability of CPUs enables their ability to execute sequences of simple instructions such as ADD, BRANCH, etc. However, the energy required to fetch and interpret an instruction is 10x-400x more than that required to perform a simple operation. Such high overhead compelled SoC architects to explore alternate architectures such as domain-specific hardware accelerators to sustain the scale of performance and efficiency.

A hardware accelerator is a computing engine that is specialized for a particular domain of application. Today, accelerators are being designed for graphics, machine learning, bioinformatics, block-chain, image processing, encryption, decryption, and many other computation intensive tasks. Accelerators offer order-of-magnitude improvements in performance/cost and performance/watt compared to general-purpose programmable processors. Such improvements are attributed to a combination of specialized operations and parallelism. Some accelerators are up to 15,000x faster than a reasonably fast CPU in executing the same computation task.

As hardware accelerators are becoming more pervasive as one of the ways to extract more performance and efficiency from architectural innovation, new challenges have emerged in verifying them in pre-silicon phase. Hardware accelerators often lack precise and detailed specification (such as ISA for processors), therefore the verification teams require to tread a steep learning curve before producing meaningful results. Different SoCs instantiate different configurations of accelerators (for various energy and execution timing targets) with a wide variety of functions and algorithms, therefore the verification methods need to be configurable and capable of singing off all use-case models of the accelerator. Moreover, since accelerators are newer types of designs, we are yet to acquire experience in verifying them accurately and efficiently.

In this paper, we introduce an effective framework for verifying hardware accelerators using formal verification, addressing the distinct verification challenges of verifying hardware accelerators, and outlining our successful strategies to overcome them. We will share a case study of successful application of proposed formal verification framework applied to a hardware accelerator design for decompression of files compressed with deflate algorithm used in server class SoCs to highlight the effectiveness of the framework and share the implementation details and noteworthy results it has produced.

II. Problem Statement

Though hardware accelerators are specialized to accelerate specific computational tasks, it comes with its own challenges. In our experience with deflate decompression accelerator, the following key challenges stood out especially in the context of verifying hardware accelerators.

1. Design Complexity: As accelerators become more sophisticated, they integrate more transistors and intricate designs. Verifying such complex designs becomes an immense task.
2. Concurrent Operations: Hardware accelerators often work concurrently or in parallel to boost computation/performance. Verifying correct operation when multiple tasks run simultaneously is challenging.
3. Error Handling: As decompression accelerators handle encrypted files from numerous implementations of encryption engines, detection of erroneous files becomes essential for robust and reliable system behavior.

Implementing and verifying effective error detection mechanisms, especially for subtle or rare errors, can be challenging.

4. Real-world scenarios and corner cases: Simulating and testing the accelerator for every possible real-world scenario (encrypted file in our case) is challenging. Often, corner cases or uncommon scenarios are where unexpected behaviors might manifest.

While hardware accelerators offer significant benefits in terms of performance, their verification poses multifaceted challenges. A holistic approach encompassing design, testing, and real-world deployment is essential for effective verification. In this paper, we plan to go into more details of the following decompression accelerator architecture shown in Figure 1.

III. Methodology

A. Formal Verification Framework – Slice (Sequentials), Dice (Parallels) & Stitch

Though formal verification as a technology offers exhaustive breadth-first search analysis to find design issue, it suffers from exponential computational complexity which primarily stems from three main sources – sequential depth of the design, spatial footprint of the design and input space of the design.

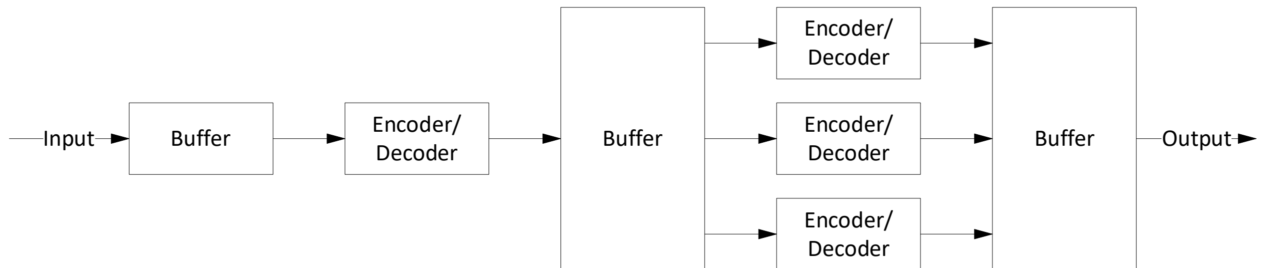

Figure 1. General Hardware Accelerator Architecture

Here shown is a general structure of an accelerator, where we see several encoders or decoders with buffers in between to match throughput. The data flow through accelerators is mostly in a pipelined manner with little possibility of complex data flow loops. This can allow us to conveniently disintegrate the design into smaller and easy-to-verify blocks. Based upon this fact, we propose a formal verification framework that addresses the verification complexity challenges with the following technical solutions.

1. Slice: Modular approach for verifying different components or blocks of a hardware design separately. This approach is often used to simplify the verification process and break down a complex design and long sequential depths into more manageable parts.
2. Dice: Identify parallel streams of computation, split them, create custom formal properties for one of the dice and apply them to all parallel streams.
3. Stitching: Cross-prove formal properties of neighboring steams and pipe-stages to achieve an exhaustive analysis of all the functions of the accelerator. Assumption of neighboring block becomes the assertions of current block.

Design partitioning and cross-proofs are some techniques that have been well-researched in applied formal verification. However, our framework stands out from already published work because it outlines effective strategies for slicing, dicing, and stitching, in the context of widely used accelerator architectures. Also, it provides details about implementation of formal properties and reference model for commonly used micro-architectures such as history buffers, Huffman decoders, merge buffers, byte serializers, etc

IV. Case Study on Deflate Algorithm Accelerator

Our work is on a decompression design which uses Deflate algorithm to convert a compressed file into clear text. The Deflate algorithm is one of the many lossless compression algorithms being extensively used in .gzip compressed files and .png files. This compression algorithm can be described as a two-step encoding process wherein LZ77 encoding is done on the clear text which is followed by Huffman encoding [1]. So, to decompress a given compressed data through Deflate algorithm, first Huffman decoding is to be done followed by LZ77 decoding to finally obtain the clear text. To perform this task, the decompression IP has a serializer to create input bit stream for Huffman decoder, CAMs (Content Addressable Memory) for Huffman code look-up by the Huffman decoder, a memory to read previously written data from during LZ77 decoding and several buffers to match throughput wherever required.

As it stands, decoders usually deal with data transformation. For the same, these blocks tend to have complex FSM implementations with multitude of state transitions possible. When formal tool drives all possible combinations at the inputs, the number of state transitions goes up exponentially with each successive bound of the checker, leading to high sequential complexity. Being tasked with performing FV (Formal Verification) on such a design, our first thought was to try to slice-up the design so that the decoders become stand-alone DUTs. Thus, DUTs were cut at the boundaries shown by dotted lines in Figure 2, so that they have manageable complexity. The fact that the data flow is mostly one-directional, the depicted slicing gave us convenient interfaces at which constraints were generally easy to model. So, in the attempt to reduce the design complexity, the increase in implementation complexity was in low proportion.

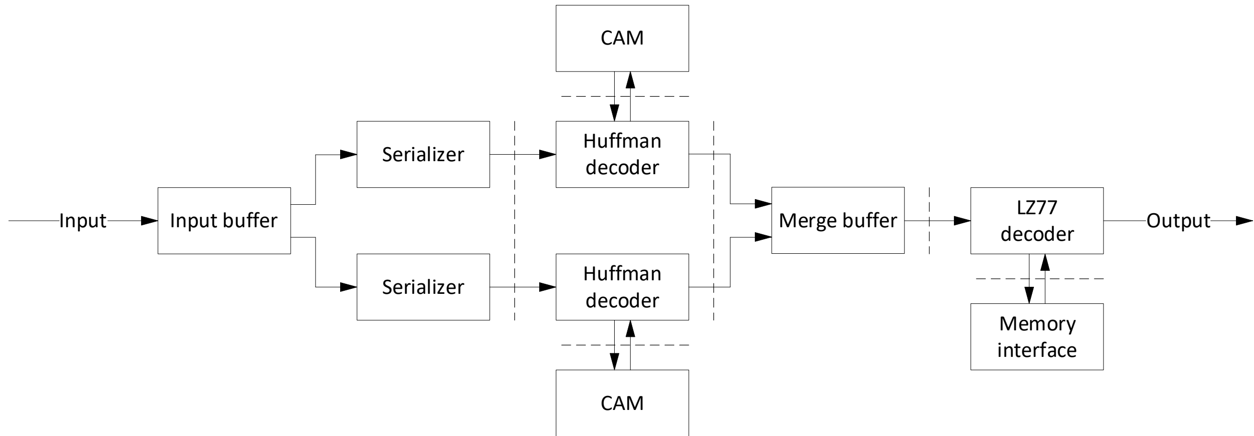

Figure 2. Simplified block diagram of Deflate decompression design with slicing boundaries

A. Addressing verification challenges of concurrent operations

Hardware accelerators often work concurrently or in parallel to boost computation/performance. One such example is multi-threading, which is quite a common feature to increase the throughput of design.

In designs with a decoder block, an input buffer is often placed at the beginning because inputs to the design are faster compared to the decoding speed of the decoder block. To enhance data readout, two instances, a leading thread, and a lagging thread, are created. They simultaneously pass data from two different memory locations. The leading pointer starts decoding from (start_address + OFFSET), while the lagging pointer starts from start_address. When the lagging pointer reaches (start_address + OFFSET), it takes the leading pointer's current value, and the leading pointer jumps ahead by OFFSET amount. It can be challenging to verify data integrity on such design due to the jumping behavior of the pointers which can be a potential cause of data dropping, duplication or re-ordering issues.

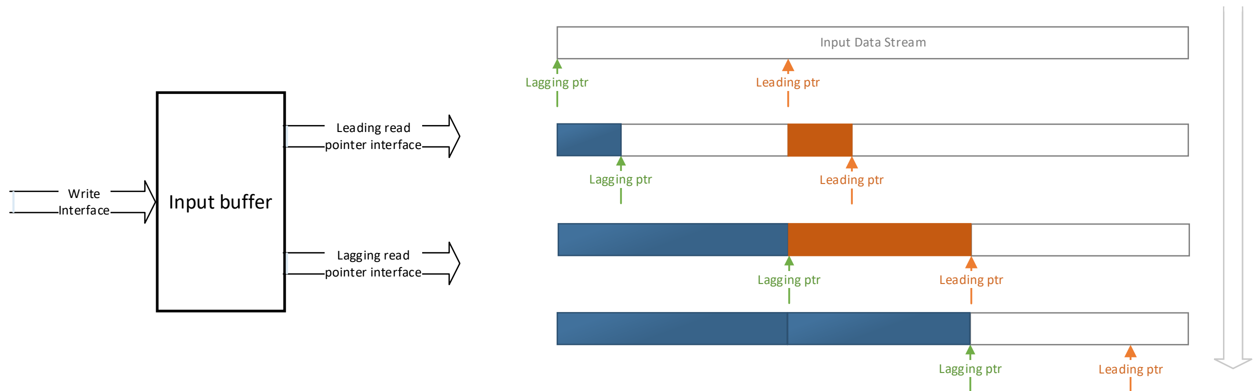

Figure 3: Leading-lagging pointers behavior

Formal scoreboards supported by different FV EDA tools will not be able to verify the data correctness in multi-threading merging. So, we decided to use the data coloring technique[3]. It can verify all the 4 aspects of data

integrity (correctness, ordering, duplication and dropping). First, we created a floating pulse, which has the properties as listed below:

1) A variable should assert only once throughout the trace of a waveform.
2) Being used to randomly tag a input request, and associated behavior of this tagged request is verified.

Then we used the task specific over-constraints to color the input data. Data before the pulse as “RED==XX” and data after the pulse as “BLUE==YY”, and pulse data as “GREEN==ZZ”.

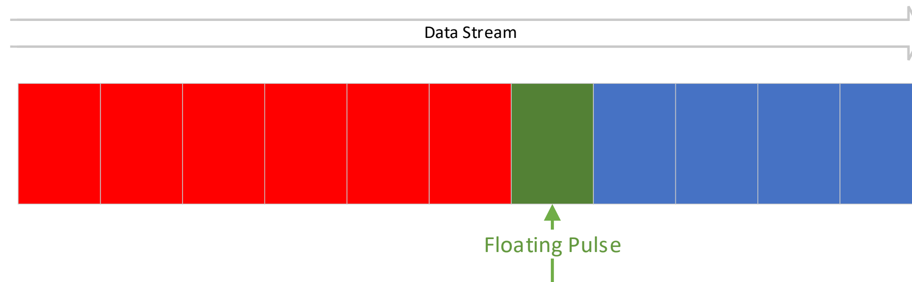

Figure 4: Data stream coloring demonstration

```
reg pulse_reg;
wire pulse;
always @(posedge clk) begin
   if (rst) pulse_reg <= 1'b0;
   else if (pulse)
      pulse_reg <= 1'b1;
end

pulse_model: assume property (
   @(posedge clk) disable iff(rst)
   pulse_reg |-> !pulse
);
```

Figure 5: Implementation of pulse model

Using this technique, it is possible to completely verify the data integrity as follows-

1) *Data Correctness:* When read address becomes equal to pulse address(write address at the time of pulse), readout data should be equal to pulse data(write data at pulse).
2) *Data Ordering:* Data coloring pattern forms 3 checkers to verify data ordering. We created 3 flags, updates with one cycle delay. Based on that following properties has been formed-
   a. If current output data is RED, then GREEN and BLUE data should not be seen.
   b. If current output data is GREEN, then GREEN and BLUE data should not be seen.
   c. If current output data is BLUE, then RED and GREEN should be seen.
3) *Data duplication & dropping:* Using the data coloring, we implemented a counter to count the number of occurrences of pulse data at output. Counter used as follows in properties:
   a. Dropping check - Counter should eventually become 1.
   b. Duplication check - Counter should not go beyond 1.

B. Data transformation verification

Decoders are quite common blocks inside the high computing processors. Decoder transforms the data, and verifying the correctness of decoded values always become tricky for verification engineer. There are always so many one-to-one mapping between encoded and decoded values, that to check for all values is almost impossible. However, using the formal verification tools and the method we used, will make it a completely holistic approach to deal with any decoders. A similar mechanism can be used to verify encoders.

A decoder first decodes the header of the file and then the payload (encoded data). In payload decoding, any decoder follows 3 properties as listed below:

1) For any random input value, it should always decode the consistently same value.
2) If consistency breaks, it should flag an error.
3) For 2 different inputs it should never produce the same output.

We used the following symbolic variables in this method for initial randomization:

1) Symbolic cycle: Same as using a floating pulse.
2) Symbolic thread: This variable, varying between 0 to (NUMBER_OF_THREADS-1). Represent which thread to select.
3) Symbolic index- This variable is used to select a single index from multi-bit data signals.

Let's deep dive into the first property of decoder "For any random input value, it should always decode the consistently same value". This method stores a random input chunk sent to the decoder. To randomize the chunk stored, we use the method of the floating pulse (discussed in previous section). The floating pulse can go high in any cycle and will remain set only for one cycle. The chunk is stored when the pulse goes high, as well as the corresponding output. If it receives the exact same chunk afterwards at the input, the decompressed output should be the same as the previously stored output. To reduce the complexity of the checker, we further case-split the checker to do this comparison for a symbolic index of the decompressed data output.

```
//Decoded byte count logic
always@(posedge clk) begin
   if(rst || soft_rst) begin
      fv_input <= '0;
      fv_output <= '0;
   end
   else if(floating_pulse && input_valid && (decoding_type==deflate) &&
payload_dec_in_progress) begin
      fv_input <= main_input_data;
      fv_output <= {main_output1, main_output2, main_output3};
      pulse_seen <= 1'b1;
   end
end
//Functional consistency check
Decoder_should_always_be_fuctional_consistent : assert property (
   @(posedge clk) disable iff(rst)
   (input_valid) &&
   (fv_input==main_input_data) &&
   (decoding_type==deflate) &&
   (payload_dec_in_progress && pulse_seen)
   |->
       fv_output == {main_output1, main_output2, main_output3}
);
```

Figure 6: Functional consistency assertion implementation

The 2nd property is straight forward to implement. However, the 3rd one can utilize the same mechanism as the 1st one with 2 floating pulse signals.

By employing this approach, we identified a bug caused by an undriven bit in a vector signal, leading to a disruption in consistency. While it was benign and could be detected in the subsequent stages of the FPV flow through coverage. Also, the following observations are made:

1) All action signals(inputs) which are indirectly controlling the input data transformation, need to behave in the same manner for two chunks which are being compared otherwise assertion should not trigger.
2) The end-to-end nature of the method enables exhaustive verification to find functional inconsistency.
3) It could dramatically increase the productivity for compression/decompression IPs (Intellectual Property) for two reasons:
    a. As these IPs consist of content address memory (CAM) and it should be exhaustive and exclusive in terms of one-on-one mapping between input and output. In simple terms, it should be consistent and gives the same output for the same input irrespective of running time. The property of CAM can be exhaustively verified by this check.
    b. It has significantly increased bug coverage as compared to conventional verification flow.
4) Though it was observed that this method increases bug coverage, still depending on the type of modules and algorithms for which it is being used, may increase the complexity. For example blocks with a greater number of action signals and flop count could take much more time to reach meaningful coverage bounds.

C. LZ77 decoder verification - handling a tricky situation

LZ77 encoded data has two components namely literals and tokens, where a token constitutes of length and distance values written as, [<distance>:<length>]. LZ77 decoding includes converting the tokens into clear text as literals are themselves clear text. During a token decode, from a position which is distance number of literals behind,

one literal is copied, and this process is length number of times [2]. An example of a sample LZ77 encoded data and its clear text conversion is shown next. In the following LZ77 encoded data, two tokens are present:

AA1BB2[6:3]CC3[6:9]DD4

We get the clear text as (literals inserted by tokens are underlined):

AA1BB2AA1CC3AA1CC3AA1DD4

A complex FSM implementation in the block clearly indicated that it needs to be verified as a stand-alone DUT. But, as shown in Figure 2, LZ77 decoder interacts with memory interface to fetch previously written clear text to decode tokens. So, to verify that the LZ77 decoder is correctly decoding tokens, we must somehow keep track of the previously written clear text. This is solved by using the concept of symbolic variables. For our purpose, we store the data being sent out (sym_byte_data in Figure 7) for a symbolic memory address. Then we track a token whose distance and length values are so that it will fetch data from the symbolic memory address. Now, when we expect the output corresponding to this token, we compare it against the data stored earlier. Thus, completing data integrity check for tokens.

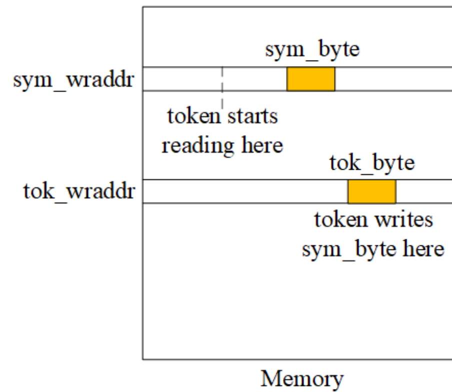

Figure 6: Schematic for tracked token in memory

```
reg tok_check_vld;
always @(posedge clk) begin
   if (~rst_b) begin
      tok_check_vld <= 1'd0;
   end
   else if ( tok_vld && (tok_dist > dist_to_sym_byte) &&
             ((tok_dist-tok_len) <= dist_to_sym_byte) ) begin
      tok_check_vld <= 1'd1;
   end
   else if ( tok_check_vld && out_wren &&
             (out_wraddr == tok_wraddr) ) begin
      tok_check_vld <= 1'd0;
   end
end

// data integrity checker
token_data_integrity_check: assert property (
   @(posedge clk) disable iff (~rst_b)
   tok_check_vld && out_wren && (out_wraddr == tok_wraddr)
   |->
   (out_wrdata[tok_byte] == sym_byte_data)
);
```

Figure 7: Checker implementation for token data integrity

In Figure 7, tok_vld, tok_dist and tok_len are token valid, distance and length inputs, respectively, to the design. The signals out_wren, out_wraddr and out_wrdata are outputs of the design which send clear text out to the memory. The signal dist_to_sym_byte tells how far back is the sym_byte that we are storing.

In this case, we establish that slicing the design can sometimes lead to implementation complexity, and in fact it is always about finding the right balance between design complexity and implementation complexity. Here we overcome implementation complexity with a workaround. Due to the sequential complexity of the complex FSM

design, a passing checker is not observed but the checker bound goes beyond what is suggested by RPD (required proof depth) analysis, thus increasing our confidence in the checker. All the highest bound cover items in the COI (cone of influence) of the checker were hitting giving us near 100% code and functional coverage on the design.

V. RESULTS

Despite late deployment of FPV, it found critical bugs and was able to stabilize the verification strategy for decompressor. This led to finding 26+ bugs and 12+ performance enhancements, and enabled robust hardware accelerators for next generation.

Figure 8: Total issues count

During the verification phase, the formal tool uncovered a corner case bug that had been overlooked in simulation verification. This scenario was rare, requiring over 32 thousand cycles of waveform data to manifest in the simulation tool. However, thanks to the formal tool's ability to provide concise counterexamples (CEX), we pinpointed the issue within just 10 cycles. We employed counter abstraction and black-boxing techniques to abstract out potential decoder components, utilizing the property that "Distance token value should be less than the number of decoded bytes within a job" to identify the problem. The implementation is illustrated in figure 9.

```
//Decoded byte count logic
always@(posedge clk) begin
   if(rst || soft_rst) fv_dec_byte_cnt <= '0;
   else if(deflate_mode) begin
      fv_dec_byte_cnt <= ({LIT_CODE_WIDTH{lit_valid}} & lit_code)+
                         ({LENGTH_CODE_WIDTH{tok_valid}} & length_tok)+
                         fv_dec_byte_cnt;
   end
end
//Token invalid soft error correctness check
token_invalid_should_assert_when_distance_value_becomes_more_than_decoded_bytes :
assert property (
   @(posedge clk) disable iff(rst)
   tok_valid & (distance_tok > fv_dec_byte_cnt)
   |->
      soft_err
);
```

Figure 9: Assertion implementation

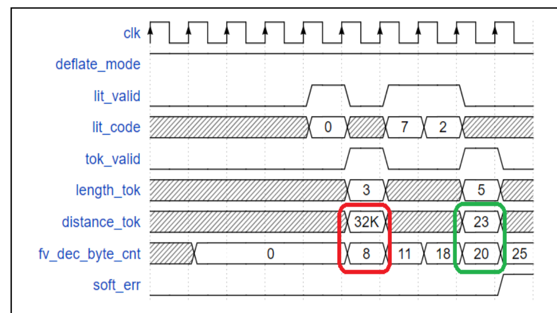

Figure 10: CEX waveform

Tokens, as previously discussed, comprise both length and distance components. While length values can vary without restrictions, there are two key limitations imposed on distance codes:

1) The distance value must be less than the memory depth.
2) The distance should not refer to data stored in the previous job.

A violation of either the first or second restriction renders the file corrupted, triggering a soft error assertion. In this case study, the second violation was identified, but it went undetected by the soft error check among 32k potential states. As illustrated in the waveform, the red box was the first occurrence of the violation, and it went undetected by soft error. But after 2 cycles again the violation occurred for different invalid distance value and that time it was detected. This led to the conclusion that the soft error mechanism was unable to identify a specific violation when the distance value reached 32k. Additionally, it was overlooked during simulation verification, revealing a security vulnerability, as malicious actors could potentially access memories from previous jobs by intentionally introducing corrupted files.

At the outset, our strategy involved validating the entire sub-system based on end-to-end properties. However, as we encountered escalating sequential complexities, we transitioned to a new methodology. Employing the slice, dice, and stitch methodology, we partitioned the design complexity of a ~15k flops, 3M gates DUT with 131 I/O signals into 6 DUTs with flop count in 600-6k range, gate count in 10k-2.5M range and the total number of I/O signals increasing to 387. Verifying the design by creating 6 separate DUTs and stitching them together requires some extra effort like gaining knowledge for internal signals and new interfaces created by partitioning, and at the end verifying that the interactions between these DUTs are working properly at the top level. As can be seen, the number of I/O signals increased to three times which means that the number of properties will also roughly increase to three times. But it can still be argued that partitioning made the work easier, as using Formal Property Verification with the whole design as DUT would mean creating a reference model for Deflate decompression which would be a huge task. Thus, effectively we reduced the design and implementation complexity.

Despite the late deployment of FPV after simulation efforts, we had a good Return on Investment (ROI) using this methodology. Our analysis of bug complexity using FPV revealed that the majority of the identified bugs were intricate and demanded the alignment of multiple events, a challenge that would have been hard to overcome with the simulation methodology. In summary, four engineers dedicated 50% of their daily time over an average period of 8 months. Consequently, the overall effort required for FPV was equivalent to 16 months for a single engineer working on the task.

VI. Conclusion and Future Work

In this paper, we discussed how the increasing use of hardware accelerators in SoCs has also proved a challenge in the pre-silicon verification phase due to newer architecture and lack of detailed specifications. Besides design understanding, there are several challenges like overcoming design complexity, tackling issues related to parallelism, finding out security vulnerabilities, etc. during verification as well. To overcome these issues and efficiently verify, we proposed our methodology of slice, dice, and stitch and discussed a case study showing how it was leveraged in Formal Property Verification of a Deflate decompression design. The effectiveness of the methodology in verifying a hardware accelerator was established through the case study showing how the slicing helped manage DUT complexity and uncover security vulnerabilities which would rather have been exceedingly difficult. We further discussed the trade-off between design complexity and implementation complexity concluding that our methodology resulted in effectively reducing both.

Our future efforts will be about how well we can use abstraction techniques with our methodology and how will it affect overall implementation complexity compared to gain in proof depth and coverage statistics.

References

[1] P. Deutsch, “DEFLATE Compressed Data Format Specification version 1.3”, https://tools.ietf.org/html/rfc1951, 1996.

[2] Ziv J., Lempel A., "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343.

[3] P. Wolper, “Expressing interesting properties of programs in propositional temporal logic”, POPL '86 Proc. of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages”, pp. 184-193