Formal Verification Framework for Hardware Accelerator Designs

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

2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES
SAN JOSE, CA, USA
MARCH 4-7, 2024

INTRODUCTION (PROBLEM STATEMENT)

Hardware accelerators (HAs) significantly improve performance in specific computational tasks and often lack detailed specification, posing major challenges in pre-silicon verification phase. The following challenges stand out.

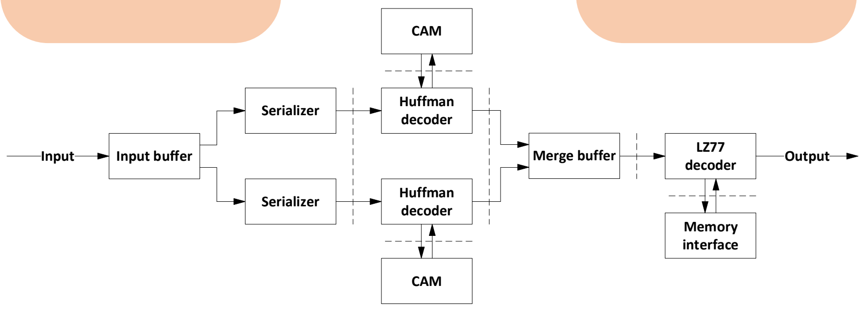
Design complexity	HAs have intricate designs to enhance performance, thus verifying them is a complicated task.
Concurrent operations	HAs often parallelize tasks which makes it challenging to verify correct behavior.
Error handling	Detection of erroneous inputs is essential trait of a reliable system. Verifying error detection logic for corner cases can be challenging.

METHODOLOGY

Slice
Verify different blocks separately to break down long sequential depths into manageable parts.

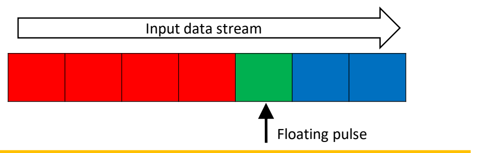
Dice
Identify parallel computations, verify one stream separately and use it for all streams.

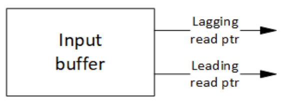
Stitch
Cross-prove all the properties in adjacent partitions to have exhaustive verification setup.



CASE STUDY 1 – Concurrent Operations

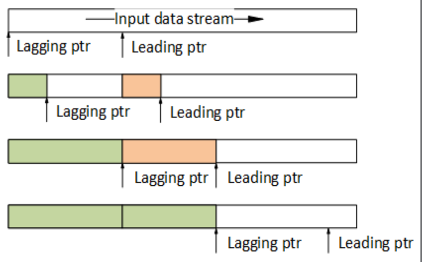
Multiple threads increase the read throughput from the input buffer.



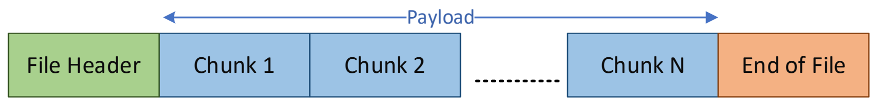


4 Data Checks

1. **Correctness** - Data read from symbolic address should be Green data
2. **Ordering** -
   - (out_data == R) then G and B data should not have been seen
   - (out_data == G) then B data should not have been seen
   - (out_data == B) then R and G data should have been seen
3. **Dropping** - Green data should come eventually
4. **Duplication** - Green data should come only once

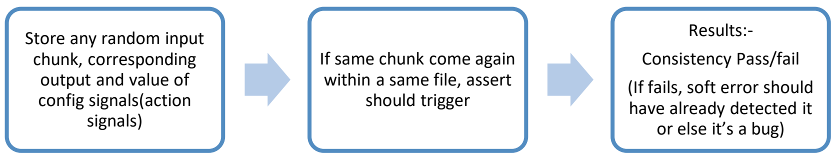


CASE STUDY 2 – Data Transformation Verification



Payload verification follows 3 properties as listed below:

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


Store any random input chunk, corresponding output and value of config signals(action signals)
If same chunk come again within a same file, assert should trigger
Results:-
Consistency Pass/fail
(If fails, soft error should have already detected it or else it's a bug)

Symbolic variables are used for deep case-splitting

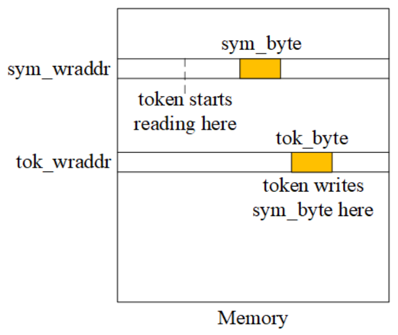
Symbolic thread: To select a thread	Symbolic cycle: To select a random chunk

CASE STUDY 3 – Tricky situation with decoder

As depicted earlier, LZ77 decoder fetches previously written clear text from memory to decode tokens and has a complex FSM. Verifying data integrity using the decoder as a stand-alone DUT was challenging.

Solution – Symbolic variables

1. Store the data output for a symbolic address (sym_wraddr).
2. Track a token whose length and distance require it to fetch data from sym_wraddr.
3. Compare the output for this token, against the data stored earlier.



Slicing can sometimes lead to increased implementation complexity. In fact, there exists a trade-off between design and implementation complexity.

RESULTS

- Slicing, dicing, and stitching offer a potential solution to complexity challenges. However, it introduces implementation complexity, necessitating a careful trade-off.
- The FV setup identified 26+ bugs and achieved 12+ performance enhancements.

26 Bugs + 12 Performance = 38

- This setup enabled the development of robust hardware accelerators for the next generation.
- Each case study in the paper details an exhaustive method tailored to a specific feature.
- The formal tool's ability to provide concise counterexamples simplified the debugging task.

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