Raising the Bar: Achieving Formal Verification Sign-Off for Complex Algorithmic Designs, with a Dot Product Accumulate Case Study

Disha Puri, Madhurima Eranki, Shravya Jampana
Intel Corporation
Disha.puri@intel.com , Madhurima.eranki@intel.com , Shravya.jampana@intel.com

***Abstract*- Matrix multiplication is a crucial operation in Machine Learning (ML) applications. Dot Product Accumulate Systolic (DPAS) units include interconnected multipliers and adders to perform the matrix multiplication (Figure 1). Given the need for frequent and extensive performance optimizations to meet market requirements, thorough verification becomes indispensable. While traditional validation primarily focuses on data validation by verifying the Register Transfer Language (RTL) against a golden specification, there remains a potential gap when the specification model itself is not entirely dependable. Additionally, the novelty of these designs necessitates multiple iterations to ensure the validity of constraints. In this paper, we present a comprehensive End-to-End (E2E) Formal Signoff approach for DPAS unit and aim to bridge the potential reliability gap. The proposed workflow starts with converting a new C++ model (provided by architect) to a golden model, followed by C2RTL equivalence checking to establish correctness of RTL and continues with various methods employed to enhance validation process. The workflow and methodologies outlined in this paper establish a solid foundation for future endeavors in achieving reliable and accelerated formal signoff.**

I. INTRODUCTION

DPAS has wide and quite complex design space that supports multiple floating-point formats with order of matrix multiplications (Figure 2) often customized. In addition to the design complexity, systolic architectures go through several optimizations to support high-speed computation and to reduce area/power consumption. The huge design space and multiple versions emphasizes the need for formal methods that can address the complexity and perform equivalence checking with lower turn-around time.

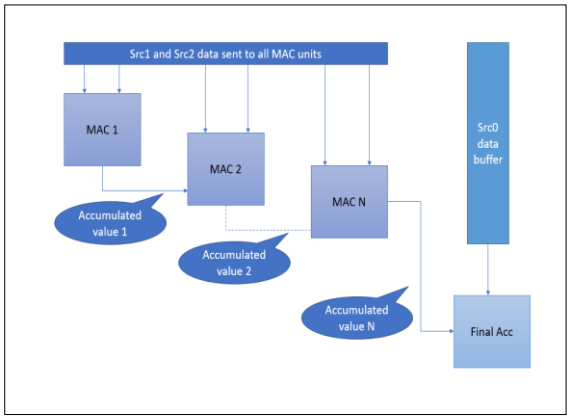

Figure 1: Dot Product Accumulate Systolic

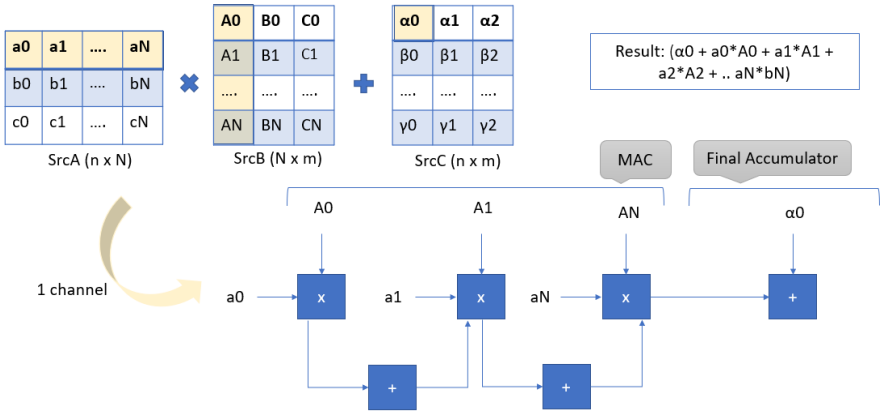

Figure 2: Matrix multiplication in a single systolic channel

The verification of DPAS unit comes with a list of challenges. An untimed architectural C-Model is a fundamental reference needed to perform C2RTL equivalence checking. Prior to performing C2RTL verification, the absence of a golden reference model posed a greater challenge. Since verifying a single transaction of data could lead to bug escapes, an effective verification flow demands to allow multiple transactions and catch interaction bugs which happen under uncommon scenarios due to multiple floating-point formats. Arriving at a complete set of constraints which allow only legal scenarios for an entirely new design is challenging. An over-constrained environment can unintentionally mask an underlying bug and thus needs an effective method to validate constraints. The proposed flow surpasses the limitations of traditional verification methods by providing a more streamlined and efficient verification flow. The significance of the achieved left shift through this flow is considerable, as the workflow enables early bug detection and resolution, minimizes rework, and improves the efficiency of the verification and development process for DPAS units in future generations. Subsequent sections of this study demonstrate how each challenge was addressed while identifying crucial design bugs along the way.

II. E2E Datapath Formal Sign-Off Methodology

A. C++ Property Verification

The process of making the C++ model golden involves refining or improving the reference model to make it more accurate and reliable. To refine the model, we have converted design restrictions to C++ assertions and included them in the model to ensure that the algorithm itself operates accurately. In formal verification, assertions play a crucial role in specifying and checking the desired properties or behaviour of a design. The C++ "assert" macro, allows validation of properties at runtime to ensure that the model aligns with the expected behaviour.

After industry survey of projects done in software formal domain, we realized that minimal work is done on checking C++ model. The proposed work uses the current industrial C2RTL tools and enhanced them with help from vendors to perform C++ property verification. Figure 3 depicts the detailed flow that begins with the wrapping constraints with architectural C++ model to generate a constrained model. Assertions failures seen while performing C-FPV would need refining of architectural model. The golden model can then be used for further verification.

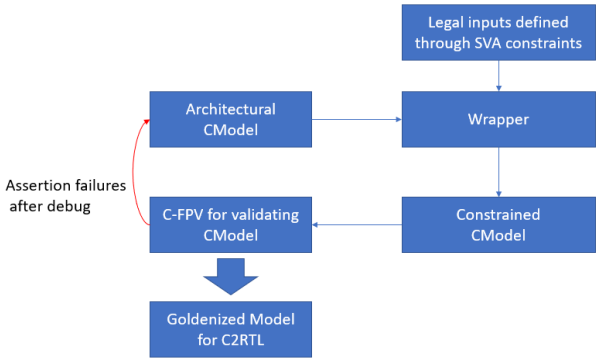

Figure 3: Process of making C++ model golden

B. C2RTL Equivalence Checking

Design complexity is addressed by splitting the design into two Designs Under Test (DUT): MAC (multiply-accumulate) and Final accumulator (adder). Further division was made to handle the time complexity as shown in Figure 4:

a) Single transaction model:

A single instruction is allowed in the pipeline, enable signal is high only for one cycle

In conventional single transaction data path verification approach, we perform the equivalence checking by allowing only a single instruction into the pipeline. Single transaction verification can lead to missing critical errors that only manifest during complex interactions. For designs like DPAS that allows multiple formats and has dedicated pipelines for each format, formal tools can struggle to give full proof results due to the scale and complexity of the design. Since the objective of the proposed work is to provide full proof for single transaction models of both MAC and adder, we have employed various convergence techniques such as case split, assume lemma guarantee [4], etc.

Assume-Lemma-guarantee technique is used to simplify the verification process by abstracting the system's behaviour into high-level properties or lemmas. These lemmas which capture the essential aspects of the system's functionality are then used as building blocks for proving more complex properties. By breaking down the verification process into smaller, manageable lemmas we have achieved significant reduction in convergence time for the MAC unit. Initially, we have proved the C2RTL equivalence of partial products which are the internal results of multiplier. These proven helper lemmas are then provided as assumptions to reduce the cone of influence and ease the convergence of final multiplier result lemmas for the formal tool engines.

Another such convergence technique applied for this design is case splitting. The case split convergence strategy is used to separately handle different floating-point formats supported by the DPAS architecture. We have applied case split on both MAC and final accumulator based on different floating-point formats. Every formal testbench performs C2RTL equivalence by allowing input data of only single format and disabling rest of the formats, this collectively provides proof for all formats.

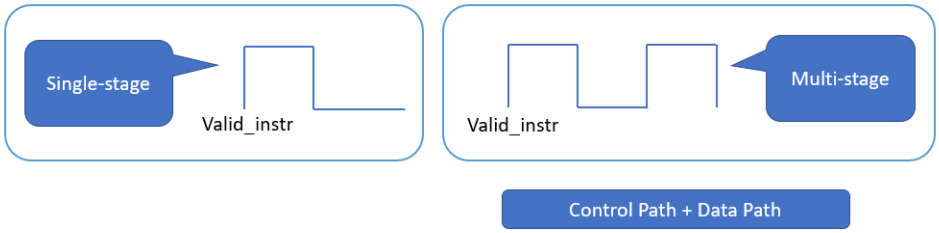

Figure 4: Single/Multiple transaction models

b) Multi transaction model:

Multiple instructions are allowed in the pipeline, enable signal is unconstrained for all cycles. Single transaction verification can lead to missing critical errors that only manifest during complex interactions. Creating a comprehensive set of test cases that cover a wide range of transaction scenarios can be time-consuming and may still fail to capture all possible interactions. So, we need a strong and exhaustive data path verification strategy to formally validate the pipeline. Multiple stages and concurrent transactions in a pipeline introduce additional complexity and makes it challenging to ensure the correctness of data flow, control signals, and synchronization between stages. By formally specifying the properties and requirements of the data path, the verification process can check if these properties hold for all possible transactions and pipeline stages. With the collaborative efforts between designers and verification, we were able to verify the multi-stage pipeline architecture of DPAS.

C. Ensuring correctness of Formal Environment

a) Integrating FV constraints in DV Environment:

To arrive at the set legal constraints, an initiative was taken to implement them in the form of System Verilog Assertions (SVA) and were then integrated in Dynamic Verification (DV) environment. The constraints are carefully crafted to ensure that the pipeline runs smoothly with no scope of over constraints being unintentionally added in FV environment. SVA helped to specify properties, check correctness, detect bugs, achieve coverage closure, and aided in debugging. This initiative was highly appreciated by designers and is made part of future methodology

b) Covers Methodology:

Additional functional covers are included in the verification flow to detect over-constraints or any unreachable logic. Such covers also increase the confidence factor for any bounded proof. For example, maximum positive/negative values and each combination of exception flags are included as covers to ensure that the testbench is allowing all possible legal values.

D. Regression Methodology

The formal testbench used for E2E proof is automated and integrated to the main repository. We achieved a huge left shift as this automated process generates detailed results and quickly finds all bugs of every release, before the simulation runs.

III. RESULTS

The design statistics of DPAS unit are shown in Table 1. Control elements like counters, FSMs or FIFOs are not present in the design and do not contribute to the design complexity.

# Lines of Code	5 thousand
# Gates	120 thousand

Table 1: Design statistics of MAC and Adder

We applied the proposed methodology on 2 units of DPAS: MAC and Final Accumulator and found over 15 corner case bugs as shown in Figure 5 of both RTL design as well as the architectural C++ Model. Majority of the design issues were found while performing C++ property verification and C2RTL equivalence checking.

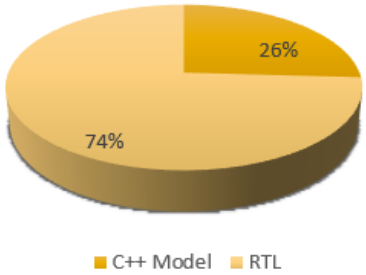

Figure 5: Bug count for EuDPAS Unit

A. Example of a C++ bug found through C++ Property Verification

Assertions included in C++ Model, also known as user asserts, assist in constraining the design to allow only legal scenarios. Additionally, verifying such user assertions will also strengthen the model, since there can be cases where the user assertion failures persist even with the required set of constraints. One such case was encountered when a user assertion that checks if the exponent is within the legal range, was failing despite constraining the design and allowing only legal exponent values. The failure was due to the incorrect values given to minimum and maximum parameters in C++ model. The parameters were later modified with the expected values by the architect.

B. Example of a C++ bug found through C2RTL equivalence checking

The proposed methodology includes implementation of C2RTL equivalence checking assertions that check if the output of RTL design is equal to the C++ model result. Several mismatch issues found by utilizing these C2RTL lemmas were missed in simulation level testing. One such significant corner case bug as shown in Figure 6 (C++ result is available in the same cycle where valid instruction is high since C++ is an untimed model, whereas RTL result can be seen a few cycles later due to the design latency), was found by formal testbench when negative input data was being falsely rounded up. The carry-add logic in the final accumulator is expected to perform addition and represent result by using 2's complement. On the contrary, C++ carry-add logic differs from the expected behaviour, since the negative result is represented in 1's complement instead. This was a critical bug and could only be found by formal testbench since such a corner case needs input to be negative with extremely small exponent.

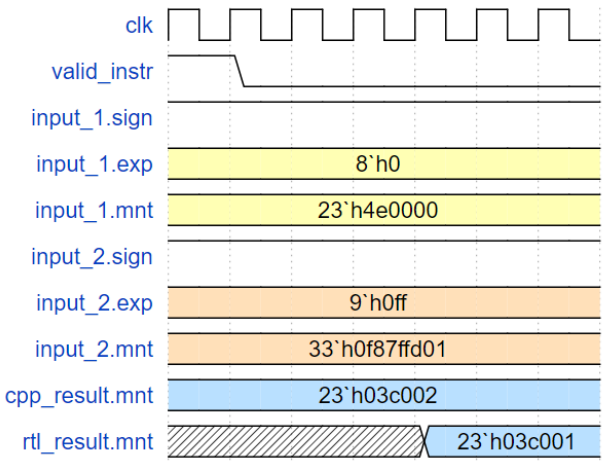

Figure 6: C++ Model bug found through C2RTL equivalence checking

C. *Example of an RTL design bug found through C2RTL equivalence checking*

Another corner case bug as shown in Figure 7, was found by formal testbench when one of the inputs provided to adder was too small. The corner case has one of the 2 inputs to the adder as zero and the other input has exponent value of -126 that is less than the minimum value for a 32bit floating point number. Based on these details, RTL design logic clamps the result to zero. But there is an exception that needs to be considered by the design when the exponent is too small. In addition to checking the exponent range, RTL design must also check if the mantissa can be renormalized to fit within a denormal 32bit float.

In floating-point arithmetic, a denormal (or subnormal) number is a special representation used to handle very small values that are below the normal range of the floating-point format. The corner case bug is seen since RTL doesn't support denormal handling and flushes such small values to zero. The expected output should be a denormal instead, since the mantissa can be renormalized to fit within a denormal even though the exponent is below the legal range. The denormal handling for such cases was later included by the designers in the logic such that the design can give rise to a precise output instead of erroneously flushing the adder result to zero.

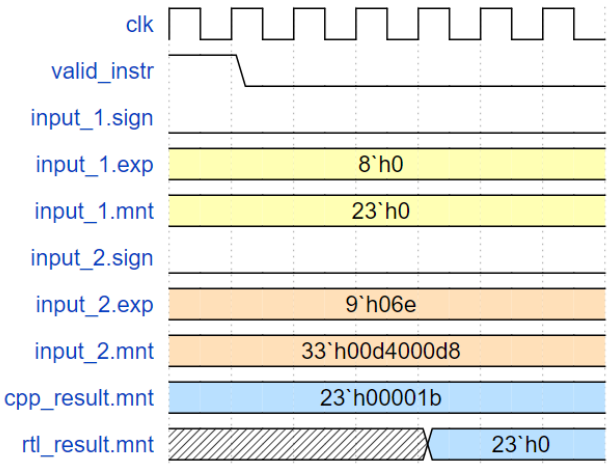

Figure 7: RTL design bug found through C2RTL equivalence checking

Furthermore, adding functional covers to the testbench has also proved to be a vital initiative that was taken for this design.

A. Example of a functional cover included in FV testbench for final accumulator

One such case that illustrates the need of adding functional covers was seen when the design was unstable and various features were being included. To reduce hardware, the size of an overflow buffer being used by the final accumulator was reduced. A functional cover was included in the formal testbench that checks if the overflow buffer was too small. Based on the status of the cover, we were able to promptly verify that the overflow buffer was shrunk to a point where the sticky bits can no longer go back into the result data and hence led the designers to modify the design and make sure that the buffer is sized adequately.

B. Example of a functional cover included in FV testbench for MAC

Floating-point formats are used to represent numbers with varying precision in computer systems. DPAS design supports multiple floating-point formats has dedicated pipelines for each of such formats. To reuse hardware, DPAS unit that includes multiple pipelines can be merged into a single format. This gives rise to cases where a single input data can be used to represent multiple formats. For example, a single 16-bit input data can be used to represent both 16-bit and 8-bit format data. Functional covers are advantageous in such cases since they strengthen the formal testbench and prove that all possible values for every format can be covered and verified.

The complete set of properties, constraints and checkers in-place found corner cases and reduced the FV turn-around time to less than a week. Multiple versions of DPAS unit have been formally verified with the testbench that allows single transaction as well as multiple transactions of data.

III. CONCLUSION

Through this activity we have showcased that it is possible to sign-off on complex arithmetic designs with complete confidence. The proposed workflow was able to create a golden C-Model, perform C2RTL equivalence checking, integrate FV properties in DV and has led to numerous bug fixes in the design. The proposed work employed various convergence techniques to deliver full proof for single transaction setup and advances to verify the design when multiple transactions of data are allowed. The well-defined and stable flow has made it possible for quick bring-up of multiple optimized designs with ensuring quality. This activity was a collaborative effort of designers, formal verification engineers and tool vendors.

Looking towards future generations, there are potential areas for further improvement. Incorporating improvements, such as integrating machine learning techniques into the verification flow, employing various formal verification methods, and addressing scalability challenges to handle larger designs can further advance the verification of such designs for future generations. These enhancements will contribute to the development of dependable and efficient ML applications that rely on matrix multiplication operations.

ACKNOWLEDGMENT

We would like to express our sincere gratitude to the RTL and C++ designers, Elliot, Alex, and Emi, for their invaluable contributions in the design discussions to deep dive into the design for resolving the convergence issues. We would also like to acknowledge the support of Vichal from the Graphics management team for understanding the algorithm's complexity and bandwidth requirements and thus helped in successfully carrying out the activity. Furthermore, we extend our thanks to Kiran, the FVCTO Manager, for providing continuous guidance throughout the project ensuring the project's success.

REFERENCES

[1] P. McLellan, “Datapath Formal Verification 101: Technology and Technique”, JUG 2021
[2] M Achutha KiranKumar V, Disha Puri, Mohit Choradia, Paras Gupta, “Novel Paradigm in Formally Verifying Complex Algorithms”, DVCON US 2021.
[3] Nathan Chong, Byron Cook, Konstantinos Kallas, “Code-level model checking in software development workflow”, ICSE-SEIP
[4] S. Roy, “Formal verification based on assume and guarantee approach: a case study”, IEEE 2002d