Achieving End-to-End Formal Verification of Large Floating-Point Dot Product Accumulate Systolic Units

Emiliano Morini, Bill Zorn, Disha Puri, Madhurima Eranki, Shravya Jampana
Intel Corporation
Emiliano.morini@intel.com, Bill.zorn@intel.com, Disha.puri@intel.com,
Madhurima.eranki@intel.com, Shravya.jampana@intel.com

***Abstract* - Dot Product Accumulate Systolic (DPAS) units are developed to implement matrix multiply-add operations, which are fundamental in the most popular ML and AI algorithms. These units are subject to frequent and extensive performance optimizations; hence their fast and complete verification is extremely important. We propose an end-to-end flow to verify these components via Formal Equivalence Checking, explaining how we create trusted reference models, how we achieve full convergence of our proofs and how we check the soundness of our Formal Verification setup. With this flow we have identified very hard bugs and, once the code was fixed, we have proven that the units under test are bug-free, completing all our proofs in just a few hours. The end-to-end methodology proposed can be reused in future projects and is expected to enable a shift left for every future generation of RTL.**

I. Introduction

As technology continues to progress, Machine Learning (ML) and Artificial Intelligence (AI) find new applications and are deployed to produce results never achieved before. One key component of these achievements is the evolution of hardware, which enables the execution of mathematical operations at high speed, keeping the power consumption under control.

One of the most common operations is floating-point matrix multiply-add, which requires a dot product for each entry of the resulting matrix, as shown in figure 1. Dot Product Accumulate Systolic (DPAS) units are developed to implement matrix multiply-add operations, as systolic arrays are considered a primary part in the ML accelerator architectures. Such designs often go through frequent/massive performance optimizations to meet market demands, thus requiring quick and extensive verification.

$$\underset{R^{m,p}}{\begin{pmatrix} r_{11} & r_{12} & \dots & r_{1p} \\ r_{21} & r_{22} & \dots & r_{2p} \\ \vdots & \vdots & \dots & \vdots \\ r_{m1} & r_{m2} & \dots & r_{mp} \end{pmatrix}} = \underset{A^{m,n}}{\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}} \cdot \underset{B^{n,p}}{\begin{pmatrix} b_{11} & b_{12} & \dots & b_{1p} \\ b_{21} & b_{22} & \dots & b_{2p} \\ \vdots & \vdots & \dots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{np} \end{pmatrix}} + \underset{C^{m,p}}{\begin{pmatrix} c_{11} & c_{12} & \dots & c_{1p} \\ c_{21} & c_{22} & \dots & c_{2p} \\ \vdots & \vdots & \dots & \vdots \\ c_{m1} & c_{m2} & \dots & c_{mp} \end{pmatrix}}$$

$$r_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} + c_{ij}$$

Figure 1. Matrix multiply-add operation, where each entry of the result matrix is a dot product (and an addition) involving 2n+1 floating-point numbers.

The RTL implementation of a matrix multiply-add computes all the entries of R. Each of the r_{ij} is a dot product of dimension n, plus the addition of an extra term, which can be seen as another "half dimension", considering that c_{ij} is multiplied by an implicit 1. We usually refer to this operation as DPn.5, a dot product of dimension n and a half. A DPn.5 is computed by accumulating every multiplication $a_{ik} \cdot b_{kj}$ and then adding c_{ij}. The exact details on how the systolic array is implemented depend on the specific design and on the power, speed, and area requirements that it needs to achieve. For example, in one of the architectures under test, the result was obtained by computing a DP2n as the addition of two DPn, as shown in figure 2.

The DPAS units have multiple channels, each computing a row of the result matrix R. The systolic nature of the implementation requires the input values to be provided in the correct cycles and the output results are ready after a

predefined latency and potentially not all at the same time. As a result, the setup to test a DPAS unit is error prone and requires some extra care, as described in the following sections.

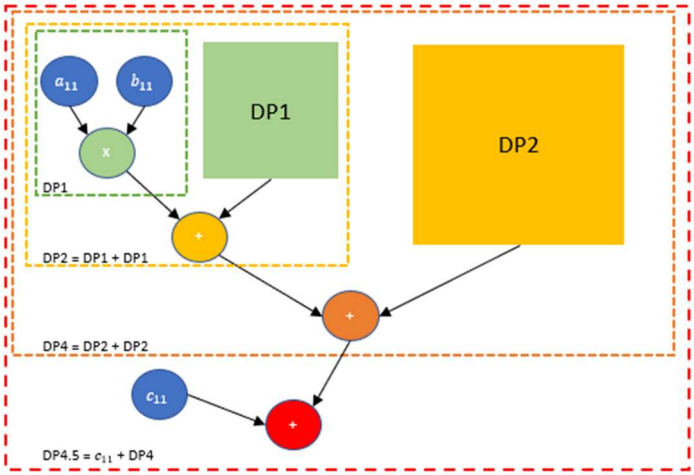

Figure 2. Example of implementation of a DP4.5: starting from the initial multiplication (DP1), two DPn are added to compute a DP2n. To compute the final DP4.5, the term c_{11} is added to a DP4.

These units have a huge input space, as shown in table 1, and exhaustive simulation is infeasible, leaving formal methods as the only option to check the overall correctness. However, the size of the input space and the arithmetic involved raise interesting challenges also for the formal verification approach, requiring the usage of several sophisticated procedures.

In this paper, we present a comprehensive end-to-end Formal Verification sign-off approach for DPAS units. The proposed workflow begins with the creation of a new validated and formal friendly C++ model, written using the internal iFP library, developed starting from the FPCore [1] functional programming language, as explained in section II. In section III, we describe the verification approach we developed, based on Formal Equivalence Checking [2], [3], focusing on a double precision DPAS, which is complex and requires advanced and innovative techniques, as presented in section IV. The checks done to confirm the correctness of our formal setup are described in section V. We show the results achieved in section VI and we present our conclusion in section VII.

DPn.5	DP4.5	DP4.5	DP4.5	DP8.5	DP8.5
Precision	16	32	64	32	64
Input Space	2^{144}	2^{288}	2^{576}	2^{544}	2^{1088}

Table 1. Examples of the input space size for a DPn.5, depending on the value of n and on the floating-point precision considered.

II. Developing Golden Reference Models with iFP

To simplify the process of building a golden reference model and ensure its correctness, we use standard operations from the internally developed iFP floating-point reference library. The goal of the library is to provide fully specified, IEEE 754 compliant implementations of common floating-point operations that are "formal friendly", allowing them to be used for equivalence checking, and sufficiently performant to be included without modification in high performance simulation or emulation environments. iFP is implemented in C++, and it takes advantage of template metaprogramming to provide a comprehensive suite of multiprecision operations with a single core codebase.

Most standard programming languages represent numerical precision with storage types. For example, C and C++ have the `float` and `double` types for single and double-precision IEEE 754 floating-point data. Numerical operations are defined statically over these types, typically using the same type for inputs and outputs. This is limiting, as there are not very many numeric types, it is not convenient to add more types, and every supported combination of types and operation requires an independent interface definition.

Rather than depending on storage types, iFP determines the precision of operations using a rounding context, an approach adapted from the open-source FPCore language [1]. iFP *has* storage types, but they are calculated automatically via template expansion, and users can simply declare their variables `auto` in C++ to avoid interacting

with them explicitly. The rounding behavior of an operation, and hence its output type, is determined by the rounding context object passed to the operation. Implicit conversion between storage types is completely forbidden in iFP, to prevent accidental double rounding.

To illustrate, consider a simple floating-point operation to multiply a single precision value by a double precision value, producing a single precision result. Using native C++ datatypes, this can be expressed:

```
float x; double y;
float result = x * y;
```

iFP expresses things slightly differently:

```
using FP32 = RoundingContext<8, 32>;
using FP64 = RoundingContext<11, 64>;
FP32::Unpacked x; FP64::Unpacked y;
auto result = fp_mul<FP32>(x, y);
```

Note that the native C++ implementation is not correctly rounded; C++ lacks an operation that takes one single precision and one double precision input and produces a single precision output. The native C++ code above will compute at double precision, then implicitly cast to single precision to put the output into a `float` storage type. iFP, by contrast, can provide a correctly rounded multiprecision operation based on the inputs and the output context passed to the template. The iFP code is also easily extensible to other precisions than just single and double, by changing the parameters used to create the rounding contexts.

Every numerical operation in iFP has a common core implementation which is specialized to different input and output precisions symbolically by template parameters. This simplifies implementing the library, as there is no need to implement multiple versions of each operation at different precisions, and also makes it easier to validate its correctness, as there is only a single implementation that needs to be checked. Once expanded, the template parameters become static, and the compiled code is completely well defined and finite, allowing it to be loaded into a formal environment. The large degree of static specialization also allows for aggressive optimization by the C++ compiler.

Using iFP, even with custom, non-IEEE 754 internal precision, the behavior of a DPAS operation can be specified succinctly in only a few dozen lines of C++. All datatypes are managed by the rounding context, so the code simply resembles a high-level algorithmic description of the DPAS linear algebra.

Even if the iFP library is proprietary to Intel and not accessible from outside the company, its foundation, the FPCore language, is open-source and accessible to everyone interested in developing a reusable, formal-friendly numerical library with multiprecision semantic.

A. iFP library validation - Simulation

The iFP library is validated with multiple independent approaches to ensure correctness. As a first pass, the static C++ library is compared exhaustively to another dynamic reference library based on MPFR [4]. Exhaustive testing is possible because of the parameterizable precision; operations are tested comprehensively for all number formats with up to 10-20 bits, depending on the number of inputs. This ensures that most combinations of edge cases will be covered, and as there is only one core algorithm, it provides good confidence for operations with larger number formats as well, even if they are not tested directly.

B. iFP library validation - FV

In addition to exhaustive testing, Formal Equivalence Checking (described in the next section) is used to check that the iFP model agrees with other established reference models, in particular Berkeley SoftFloat [5]. This is only possible at specific bitwidths, but it gives additional confidence for commonly used datatypes. Formal regression testing is used to ensure that the behavior of the iFP library does not change across versions.

C. iFP library validation – Asserts

The iFP C++ model contains assertions to check that the code operates as expected. These assertions are checked during simulation but are also parsed by formal tools and converted into properties which are formally verified.

III. Formal Equivalence Checking

A very successful way to verify datapath intense circuits is Formal Equivalence Checking (FEC), where the design under test (DUT) is formally compared against a golden reference model, usually untimed and written in C/C++. EDA

companies have developed tools which allow this comparison, solving many equivalence checking problems out of the box. Unfortunately, for the units we are considering in this paper, out of the box convergence is not possible and advanced techniques are required.

One crucial aspect of this FEC methodology is the quality of the golden model, which must be developed independently from the RTL, to minimize the risk of having the same bug replicated in both implementation and specification, and go through a thorough validation process, in addition to be "formal friendly", i.e., written using constructs supported by formal tools. The iFP reference models introduced in the previous section are perfect for this and we have used them for our verification.

Depending on the language used to write specification and implementation, the equivalence checking problem is often referred to as C2RTL, RTL2RTL or C2C.

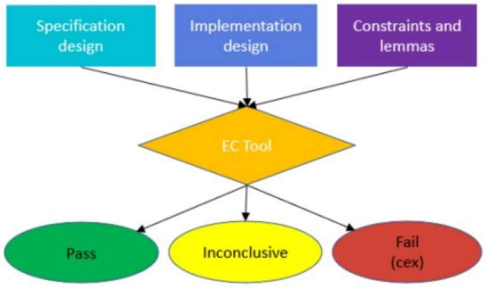

Figure 3. The inputs of an EC tool are two designs, specification and implementation, a set of constraints to drive the possible values to tests and a set of lemmas to prove. Each lemma can pass, fail or be inconclusive. A counterexample (cex) is provided for each failing lemma.

To address the verification complexity, our approach is structured in the following way:

A. Initial single transaction verification

In a conventional single transaction datapath verification approach, the equivalence checking is performed by allowing only a single instruction into the pipeline. Single transaction verification can lead to missing critical errors that only manifest during complex interactions, and this is clearly an over constraint. However, we find this approach useful to create a correct formal setup and to identify most of the complexities that are present also in the general equivalence checking problem.

Converging for a single transaction is obviously a necessary (but not sufficient) condition to achieve convergence in the general case [6].

B. Multiple transactions verification

Starting from the single transaction setup, we remove the over constraint on the enable signal and we allow back-to-back transactions. If the equivalence checking tool produces a counterexample, we need to fix the RTL, as it is not correctly handling multiple transactions. On the other hand, if the tool successfully proves the lemmas, the verification is complete. However, this is often not the case for DPAS units, due to their complexity, and the tool returns inconclusive results. In this case, it is required to apply more advanced FV techniques, as shown in the next section.

IV. DOUBLE PRECISION DPAS FV

When dealing with double precision numbers, the DPAS unit verification is even harder than in the other cases: not only the input space grows drastically, but also each multiplication and addition is more complicated to verify. Usually, RTL designers implement clever tricks to achieve the best PPA, while the reference C++ models implement the operations more in a direct way, as the priority is correctness, not performance. This produces differences between the two models, and it is very hard to find internal equivalent signals to use as helper properties. It is more likely to find internal "equivalence relations", which are more complex to describe than equivalent signals, but can be still used to simplify the verification. An example of equivalence relation is shown below, where the signal `prod` is present in both implementation and specification, but the computation is done differently and so we need to consider other signals to find an equivalence.

```
impl.prod == (spec.prod << spec.renorm_val)
```

We verify a double precision DPAS using the following approach:

- Verify the first element of r_{11} of R, which is a DPn.5 (C2RTL equivalence checking against the iFP model),
- Use symmetry techniques to show that a generic element r_{ij} of R is computed as r_{11}, so if r_{11} matches the reference model, also r_{ij} matches it (RTL2RTL equivalence checking).

We have implemented this novel approach because it allows multiple engineers to make progress in parallel, as the steps above can be executed completely independently. This parallelism would not be achievable if we repeated the C2RTL proof for each entry of R, because in this case no advancement is possible until the r_{11} proof is completed. Moreover, using the RTL2RTL symmetry approach, we introduce another level of risk diversification, as we do not rely on a single type of proof.

A. Verification of r_{11}

The implementation of the DPAS shown in figure 2 suggests a natural way to split the problem: verify two DPn, proving equivalence relations against the C++ model, and then assume those relations to prove a DP2n, in an assume-guarantee fashion.

To implement this approach, we need to start proving a DP1, which is just a multiplication of two double precision floating-point numbers. This is usually a relatively easy task, but for the double precision DPAS it is more complex than expected; the RTL designers implemented this multiplication as a radix-8 booth encoded multiplication, with values precomputed and then registered for a specific number of cycles, and then used these values for the actual computation of the partial products in the multiplication array. Some of these registers can be disabled due to power optimizations (without impacting the functionality of the design), and these optimizations drastically increase the verification complexity: the specialized formal solvers which normally recognize a multiplication implementation do not work anymore, returning inconclusive results. To solve this problem, we use the equivalence checking tool as an automatic theorem prover in an innovative way, proving the following steps to show the correctness of the multiplication:

- Correctness of the encoding,
- Correctness of the optimized partial product array created,
- Correctness of the array reduction, done via specialized compressor circuits,
- Correctness of the result in carry-save form.

This approach, implemented as an assume-guarantee and using case-splits to handle special cases, achieves full convergence, proving the full double precision multiplication in a few hours, using our computing farm. The big advantage of this approach is that it removes the dependency on the specialized solvers available in the commercial tools which try to recognize the multiplication at a gate level; they converge very quickly when they recognize the implementation, but they do not allow significant user intervention when they are inconclusive.

B. Columns and rows symmetry to verify R

After successfully proving that r_{11} is equal to the result of our golden model, we prove that all the elements r_{1j} in the first row of R are computed in the same way as r_{11}. By transitivity, this means that each r_{1j} is calculated exactly as the result of our golden model as well.

Given that the DPAS implements a matrix multiply-add, the nature of this operation is such that if we swap two columns of B and we do the same in C, without altering A, then the resulting matrix R must have the same columns swapped too. We check this property using the formal equivalence tool, instantiating two copies of the DUT and performing an RTL2RTL comparison, as shown in figure 4. We do such a proof for each element r_{1j} in the first row of R.

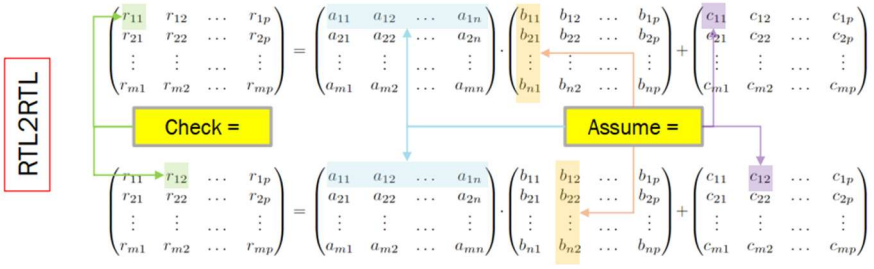

Figure 4. Representation of the column symmetry approach for r_{12}. A successful proof means that r_{12} is computed as r_{11}, which has already been proven to be equal to the reference DPn.5.

The rows symmetry idea is similar to the one used for the columns: knowing that the first row is correct, we do an RTL2RTL to show that the second row is computed exactly in the same way as the first, hence it is correct too. We repeat the procedure for all the other rows of R.

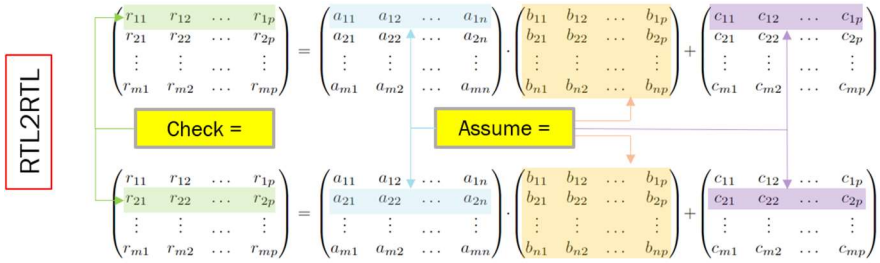

Figure 5. Representation of the row symmetry approach for the second row R_2 of R. A successful proof means that R_2 is computed as R_1, which has already been proven to be correct.

V. Checking the correctness of the C2RTL environment

To properly constrain the equivalence checking setup we write a set of System Verilog Assertions (SVA). These properties are verified with a Formal Property Verification tool and then converted to assumptions in the equivalence checking tool. The assertions are required to properly model the input sequence that the DUT expects, avoiding invalid input combinations, or to express relations between the registers used to describe the internal pipelines. For example, we write assertions of the following types (omitting clock and reset for simplicity) :

```
assert property(valid_in |-> !(ieee_compliant & flush_denormal))
assert property(valid_in |=> !valid_in)
assert property(!(normal_pipeline_reg & special_pipeline_reg))
```

The properties about the registers are required because FEC tools often overapproximate the problem, abstracting the state holding variables (like registers) and treating them as free inputs, introducing states that are not possible in the real design and can lead to spurious failures (notice that if the tool passes with the over approximation there are no issues, as this proof is valid also in all the cases that are reachable in the original design). We work with the designers to incorporate the SVA properties directly in the RTL, because we want to guarantee that every verification environment for the DUT has these properties available; even if these assertions are proven in FV, they are also checked in Dynamic Verification (DV). This might seem an unnecessary repetition, but we think it is very useful to have these properties proven both in FV and DV because, in case of discording results, we can identify issues with the property or with the verification setup. In the past, the authors have observed cases where the properties added for FV highlighted an in issue in the DV setup, but also vice versa. This initiative is highly appreciated by designers and is part of future methodology.

In addition to the assertions, functional covers are also included in the verification flow to detect over constraints or any unreachable logic, as recommended in [7].

VI. Results

We have applied the proposed methodology on several units of DPAS, including a double precision DPAS, and we have found over 20 corner case bugs. Most of the bugs have been found in the RTL doing the C2RTL verification against the iFP model, but some have also been found with the symmetry approach and with the assertions added to the code. Some of the corner cases found with our methodology occur with a probability of less than 2^{-50}, which means that they are practically impossible to find via random simulation.

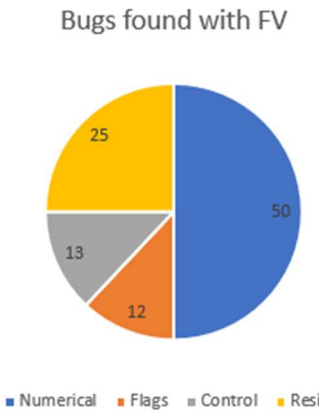

Figure 6. Distribution of the type of bugs found with the proposed FV methodology.

Examples of bugs detected include:

- Signals not sampled properly.
- Arithmetic with special floating-point input (infinity and Not a Number).
- Computation of the IEEE 754 flags.
- Implementation of the rounding.
- Flushing denormal numbers in input and output.
- Computation of the redundancy residue value.

Once the bugs have been fixed, the approach implemented allowed to achieve full convergence, **proving the absence of bugs in the final code,** and working smoothly also for the double precision case. Additional experiments show that this methodology scales up also to units where each entry depends on more than 2^{1000} input values! The largest runtime for those units is below 24 hours, and most of the time is required by verification of the double precision multiplications. This runtime allows us to quickly rerun the verification in case of internal design updates, for example to meet new PPA targets, as we can prove the full correctness in less than one day.

We have included these tests in our regression suites, using different runtime limits according to the regression type: for smoke testing, we allow these tests to run only for a few minutes, aiming for bug-hunting, while for weekly (or less frequent) regressions we allow the tests to run for several hours, expecting full convergence.

VII. Conclusion

In this paper we have presented an end-to-end methodology to sign-off on complex arithmetic designs with complete confidence. In the proposed workflow we create a trustworthy C++ reference model based on iFP, perform Formal Equivalence Checking to verify the RTL against the reference model and integrate FV properties in the RTL, so they are checked both in FV and DV. The proposed work uses several advanced and innovative convergence techniques to deliver a full proof, confirming that the DUT is bug-free. This activity was a collaborative effort of designers and verification engineers; we are also working with EDA vendors to identify which steps of our flow can be incorporated into their tools and how their solvers can be enhanced, especially considering the challenges solved for the double precision DPAS unit.

The code written to implement the formal verification strategy can be easily adapted also to verify other DPAS units, providing a solid foundation for future projects and drastically improving the time required to completely sign-off the RTL.

References

[1] https://fpbench.org/spec/fpcore-1.0.html
[2] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, *"Solver technology for system-level to RTL equivalence checking"*, in Proceedings -Design, Automation and Test in Europe, DATE, 2009
[3] P. McLellan, *"Datapath Formal Verification 101: Technology and Technique"*, JUG 2021.
[4] https://www.mpfr.org/
[5] http://www.jhauser.us/arithmetic/SoftFloat.html
[6] M. V. Achutha Kiran Kumar, Disha Puri, Mohit Choradia, Paras Gupta, "*Novel Paradigm in Formally Verifying Complex Algorithms*", DVCON US 2021.
[7] E. Seligman, T. Schubert, M. V. Achutha Kiran Kumar, "Formal Verification: An Essential Toolkit for Modern VLSI Design", 2nd Edition, Elsevier, 2023.