

Novel Paradigm in Formally Verifying Complex Algorithms

M, Achutha KiranKumar V, Intel Technology India Pvt Ltd, Bangalore, India (achutha.kirankumar.v.m@intel.com)

Disha Puri, Intel Technology India Pvt Ltd, Bangalore, India (disha.puri@intel.com)

Mohit Choradia, Intel Technology India Pvt Ltd, Bangalore, India (mohit.choradia@intel.com)

Paras Gupta, Intel Technology India Pvt Ltd, Bangalore, India (paras.gupta@intel.com)

Abstract: Modern Graphics designs are equipped with efficient hardware implementations of complex arithmetic algorithms to cater to different market segments. While these state-of-the-art features enhance the performance and marketability of the design, they also substantially add to the verification challenges. Due to computational complexity in these algorithms, there may exist corner case bugs that would be hard to find via simulation. Many of these complex algorithms are in fact, hard to completely verify through formal as well. Availability of consumable C++ model is another factor that hinders formal verification. In this paper, we discuss the challenges faced in verifying such complex algorithms and elaborate via case studies on our approach. We mention how we have used extensive divide and conquer approach in designs to find corner case bugs. We also describe how we have used property verification in datapath tools to find bugs in a design for which we did not have a C++ model.

I. INTRODUCTION

Semiconductor Industry is a very competitive market, with ever increasing focus on performance and quality. In order to cater to the different requirements for different consumer segments, a lot of variants of the same base product enter the market. Over the product generations, companies keep on adding features and performance optimizations. A lot of such design changes are algorithmic in nature with complex arithmetic. Owing to intricate data transformations, such designs cannot be verified fully using dynamic simulations.

Verifying a SLM (System Level Model) written in C, C++ or System C against the RTL is an established methodology to tackle such designs. However, not every C++ model can be readily consumed by a formal tool. In one of our previous papers [1], we have mentioned how we have collaborated with vendors and SLM designers to allow the tools to consume C++ designs with lot of modern dynamic constructs. Our efforts maximized the C++ golden specs compliance to verification. On the other hand, due to the inherent limitation of formal methodologies that several non-synthesizable constructs such as unbounded loops, pointer arithmetic etc. cannot be verified, we worked with the spec designers to provide them with crisp guidelines on what is and is not formal friendly. This effort helped us to handle multiple algorithms and showcase the value of formal by finding corner case bugs quickly. The overwhelming response and success of this initiative presented us with an opportunity to take up more advanced designs. Owing to large corner cases and huge impact of these designs, management mandated that FV must be done for a new feature before being added into the taped in RTL.

For many complex designs with multiple algorithmic paths/corner cases etc., relying on simulation alone does not guarantee a bug free environment. Due to success in SLM to RTL approach, we decided to try all such complex algorithms such as decompression, dot product accumulation etc. through formal. However, we not only need to consume the golden C++ models, we also need to handle complexity and achieve full convergence for complete confidence in the design. In this paper, we showcase how we were able to find corner case bugs and also get convergence on complex designs by correctly translating the Verification problem into the right formal assertions and helping the formal tool in handling convergence complexity, via case studies.

II. CHALLENGES AND SOLUTIONS

Ideally, one would want to compare the golden specs directly with RTL to achieve complete data space coverage[2]. However, there are various challenges associated with using these specifications directly. The high level code, agnostic to the formal verification requirements, tends to have a lot of features that are not formal friendly. Some examples could be extensive use of pointers, use of *new* and *delete* constructs (in context of dynamic memory

allocation), STL (Standard Template Library), string functions (mostly used in test generation code) etc. In most of the cases, the problem may be due to inherent style and comfort level of the designer writing the code and it may be entirely possible to avoid such constructs by rewriting this in a formal friendly way. However, the legacy code involved may have many such deep rooted constructs which would be impossible/impractical to rectify and rewrite. The coding standards for these kind of simulation codes would aim in maximizing the efficiency and speed, which aren't the correct fit for the FV compatibility. Also, every company has a set of optimized and customized libraries which are essential in writing the golden C++ models. It would be challenging if the constructs used in these libraries are not supported by formal verification tool.

Handling these complexities requires a crisp understanding of genuine requirements of the designers and the limitations of the formal verification tool. Some solutions that are applied to solve this problem are:

- Many constructs are brought into the gamut of formal friendly specifications, with the support of the tool vendors. Support for C++11 constructs, bit manipulation vectors and many static constructs used in an SLM code have been enhanced. Additionally, support for several legacy customized libraries at interface level have been augmented that one can use the same functions while calling these libraries but internally they link to tool specific formal friendly functions.
- Crisp guidelines have been provided to the designers on what can/cannot be supported by formal verification tool. For example, only simple use of pointers can be supported.
- Standard static assertions which flag the program crashes, lead to undefined behavior (such as Out of bound array accesses, max iteration checks, division by zero checks, check whether value of second operator in a shift is within the bit width of the first operand and so on) are used. Further, addition of user defined assertions for the SLM model help goldenizing the C++ specs.

Supporting Golden specs in C-RTL solves the problem of building an initial trust in the verification process. However, this effort is futile if we cannot provide complete data space coverage and ensure 100% completeness. One such hindrance in reaching the aim comes in the form of complexity. Specification-implementation verification debug is simpler when written in such a way that the two designs have some similarities. However, we have the Golden specs on one side which are written by software experts who specialize in writing codes efficiently and use templates, library functions such as boost libraries and other techniques to write reusable codes. The software writing style of an expert by default involves some element of STL's and dynamic constructs even when simple arrays can be used due to habit. On the other hand, we have hardware engineers who are writing RTL's close to the hardware elements. There is a huge gap between the two designs and finding internal mappings is generally not feasible. In such a scenario, we only have to opt for end-to-end verification but with complex designs, this leads to data space explosion which needs to be handled gracefully.

In Section III., we showcase how we have solved this problem in one of a complex design using combination of various convergence techniques such as divide and conquer, use of cutpoints, assume guarantee etc.

Another interesting problem we faced was expectation of proving a complex design using Formal without the presence of C++ model. We showcase in Section IV how we have successfully used property verification to find several corner case bugs without a reference model.

III. CASE STUDY I: DOT PRODUCT ACCUMULATE

This was a newly added execution engine for handling large matrix multiplications catering to machine learning applications. The basic operation being done repetitively was MAC (multiply-accumulate). It allows different input and output data formats, including integers and floats of varying precisions, like 2,4,8 or 16-bit integer multiplicands for integers, and floats of half precision, full precision and some custom precisions. It had 8 pipeline stages of multiply accumulate that were functionally the same. The final accumulated result was the design output to be compared against the C++ SLM model.

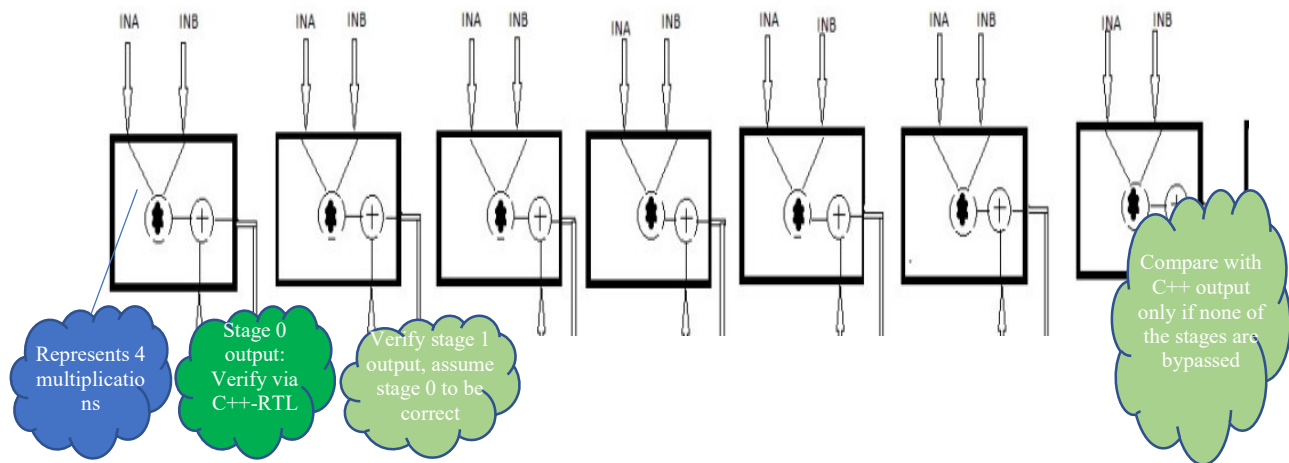


Figure 1: 8 Stage Multiply Accumulate Pipeline

The C++ had a lot of pre/post processing code. A lot of effort was spent to chalk out the actual logic from convoluted C++ code and to map the correct inputs between the C++ and RTL. This design also contained Float library functions unsupported by formal tool. By understanding the design, we were able to replace small functions, like leading zero detection, mantissa addition, normalization etc. with pre-verified code. One common cause of concern whenever spec code is to be changed to make it viable for a formal tool is to ensure the validity of the replaced code.

To ensure that there is no hole in verification, two approaches were adopted in parallel.

- 1) Wherever possible, the non formal friendly C++ code was replaced with standard libraries such as softfloat (eg: 2 input float added etc) or commonly widely used implementations such as for lzd
- 2) For other changes such as replacing non-formal friendly shift library functions, we carefully vetted each change with the C++ designer for sanctity of code.

Not only that, there was a huge bit vector of 256 bits used to store intermediate mantissa results which the formal tool could not handle. By our understanding of the allowed precisions, we were able to replace it with a 64-bit vector which also reduced the complexity. For such big designs, achieving convergence is a huge challenge. We used divide and conquer reasoning to complete the proof. Wherever possible, intermediate points were identified to match C++ with RTL and cutpoints were added to reduce the complexity of the proof. For example, we mapped the inputs of one of the stages in C++ and RTL and identified and proved the output mappings of that stage. Then we used that proof as a base to complete the entire proof. For the next generations, we were able to reap the benefits of our efforts by quickly setting up the verification environment.

The following table illustrates the different cases that were handled individually to prove the design in completeness. It is to be noted that the specification restricts the inputs formats of the multiplicands to have the same value. Also the input formats of multiplicands A and B should be the same across the 8 stages for one single instruction. Only for integers, formats of A and B can differ in sign, for example input A can have format Unsigned INT-4, while input B can be Signed INT-4. The format of the accumulator input C can only be Signed INT-32 when A and B are integers, and Float-32 when A and B are floats.

INPUT FORMAT	OUTPUT FORMAT	Multiplications per Pipeline Stage	No. of internal properties per stage	No of output properties per stage	Properties written for final output
INT-4 Unsigned	INT-32	8	8 (1 per multiplier)	2 (1 general case + 1 zero bypass)	1
INT-8 Unsigned	INT-32	4	4 (1 per multiplier)	2 (1 general case + 1 zero bypass)	1
INT-4 Signed	INT-32	8	8 (1 per multiplier)	2 (1 general case + 1 zero bypass)	1
INT-8 Signed	INT-32	4	4 (1 per multiplier)	2 (1 general case + 1 zero bypass)	1
BFLOAT-16(7 bit mantissa, 8 bit exponent)	FLOAT-32	2	4 (2 for mantissa multiplication, 2 for multiplication exponents)	4 (1 general case + 1 zero bypass + 1 INF bypass + 1 NAN bypass)	1
HFLOAT-16(10 bit mantissa, 5 bit exponent)	FLOAT-32	2	4 (2 for mantissa multiplication, 2 for multiplication exponents)	4 (1 general case + 1 zero bypass + 1 INF bypass + 1 NAN bypass)	1

Two Corner Case Issues were found by Formal which simulation had not been able to detect

- Issue 1: Design is expected to Flush the denormals on input. There was an issue related to setting of an internal flag (src0_denorm here) for denormal number on the accumulator input (src0 here) that comes at stage 0. This was propagating to the LSB of the output of stage 0.

RTL equation : $\text{src0_denorm} = \text{src0_exp_eq0} \ \& \ \sim(\&\text{src0_mant})$ Correct equation : $\text{src0_denorm} = \text{src0_exp_eq0} \ \& \ (\text{src0_mant})$

- Issue 2: When accumulator input src0 was zero, the exponent of stage 0 output was incorrect for cases where the unbiased exponent resulting from both of the multiplications per stage is lesser than decimal - 127. The issue is corner case in the sense that it is seen only when src0 is absolute zero, and not a denormal number. Also, the multiplicands for both of the multiplications at the stage 0 must be very small numbers. The design is expected to deal with the fact that src0 is zero, and internally clamp its exponents to a lowest possible value. The bug that is seen is in the logic to select the maximum exponent of the three terms at stage 0, which will be used in further logic for shifting and addition. This issue is caught by the assertion written in order to compare the RTL stagewise output against C++. Below snippet has the equivalence checks written to compare RTL against C++ in bypass and non-bypass cases. This issue was caught by the non-bypass assertion. SLM refers to C++ or the spec whereas RTL is the implementation. The multiplicands are named src1, and src2, and each stage has two multiplications being performed for floating point formats.

```

1. check_stage_n_bypass      : (SLM.Src_1_Nan[0] | SLM.Src_2_Nan[0] | SLM.Src_1_Nan[1] | SLM.Src_2_Nan[1] | SLM.Src_1_Inf[0] |
SLM.Src_2_Inf[0] | SLM.Src_1_Inf[1] | SLM.Src_2_Inf[1]) |> (SLM.Result_stage_n[31:0] == RTL.STAGE[n].renormalized_res_out[31:0])

2. check_stage_n_non_bypass_multiplication_1 : !(SLM.Src_1_Nan[0] | SLM.Src_2_Nan[0] | SLM.Src_1_Nan[1] | SLM.Src_2_Nan[1] |
SLM.Src_1_Inf[0] | SLM.Src_2_Inf[0] | SLM.Src_1_Inf[1] | SLM.Src_2_Inf[1]) |> (SLM.Multiplication_0_Exponent[8:0] ==
RTL.STAGE[n].Multiplication_0_Exponent[8:0])

3. check_stage_n_non_bypass_multiplication_1 : !(SLM.Src_1_Nan[0] | SLM.Src_2_Nan[0] | SLM.Src_1_Nan[1] | SLM.Src_2_Nan[1] |
SLM.Src_1_Inf[0] | SLM.Src_2_Inf[0] | SLM.Src_1_Inf[1] | SLM.Src_2_Inf[1]) |> (SLM.Multiplication_0_Mantissa[15:0] ==
RTL.STAGE[n].Multiplication_0_Mantissa[15:0])

4. check_stage_n_non_bypass_multiplication_2 : !(SLM.Src_1_Nan[0] | SLM.Src_2_Nan[0] | SLM.Src_1_Nan[1] | SLM.Src_2_Nan[1] |
SLM.Src_1_Inf[0] | SLM.Src_2_Inf[0] | SLM.Src_1_Inf[1] | SLM.Src_2_Inf[1]) |> (SLM.Multiplication_1_Exponent[8:0] ==
RTL.STAGE[n].Multiplication_1_Exponent[8:0])

5. check_stage_n_non_bypass_multiplication_2 : !(SLM.Src_1_Nan[0] | SLM.Src_2_Nan[0] | SLM.Src_1_Nan[1] | SLM.Src_2_Nan[1] |
SLM.Src_1_Inf[0] | SLM.Src_2_Inf[0] | SLM.Src_1_Inf[1] | SLM.Src_2_Inf[1]) |> (SLM.Multiplication_1_Mantissa[15:0] ==
RTL.STAGE[n].Multiplication_1_Mantissa[15:0])

6. check_stage_n_non_bypass_output      : !(SLM.Src_1_Nan[0] | SLM.Src_2_Nan[0] | SLM.Src_1_Nan[1] | SLM.Src_2_Nan[1] |
SLM.Src_1_Inf[0] | SLM.Src_2_Inf[0] | SLM.Src_1_Inf[1] | SLM.Src_2_Inf[1]) |> (SLM.Result_stage_n[31:0] ==
RTL.STAGE[n].renormalized_res_out[31:0]) (checks '2', '3', '4' and '5' used as assumes in proof of '6')

```

At each stage, separate properties were written to handle the different cases on inputs A, B and C. This technique of case-splitting the output assertion to be proved helps in two ways. One, it reduces the per property complexity, as each property exercises only a subset of the design needed to prove/disprove it, thereby creating a smaller fan-in cone. Secondly, it helps accelerate fixing of FV setup by giving easy to debug counter-examples per property which generally boil down to constraint issues in the initial phase of verification. For integers, there was a zero bypass flag as an output at each stage. For Floats, there were bypass flags for Zero, NAN and Infinity as output of every stage. Separate properties were written for bypass/non-bypass cases based on understanding of input conditions for when the flag would be set. The bypass properties compared the flags in the RTL against their C++ counter-parts, which are also signals internal to the C++.

At each stage, the non-bypass case stage output check did not converge directly. We first proved each of the multipliers in the stage to be correct, and used assume guarantee reasoning to prove the final output. For INT-4 formats, there were 8 multiplication per stage, as the packed 32 bit stage inputs for multiplicands A and B can be split into 8 4-bit integers. Similarly, for INT-8 and Float-16 formats, there were 4 and 2 multiplications per stage respectively.

While proving the next stage, cut-points were needed on the outputs of previous stage outputs from both the C++ and RTL, while were assumed to be equal. This involved the cut-points on the bypass flags/signal also, and appropriate constraining. The constraints put to correlate the C++ and RTL were verified using assertions at previous stage. This also required putting constraints like mutex between the bypass conditions, and the range of exponent values from previous stage.

The design was not expected to drive a denormal value on the output. This was proved stage-wise and also for the final output.

IV. CASE STUDY II: ERROR DETECTION VERIFICATION WITHOUT C MODEL

The integer and floating point execution unit handles the arithmetic operations in our graphics pipeline. The operations range from simple operations such as logic and, logic or, shift operations etc to complex operations such as double precision floating point multiplication. Every generation, new precisions are added to support complex arithmetic such as for ML(machine learning) and AI(Artificial intelligence) algorithms. Hence, ensuring complete confidence in execution unit is extremely important. As an additional check, architects/designers decided to add a residue block into the design to double confirm the logic in execution unit [refer Figure 2] . This residue block is added for most of the integer and floating point operations. The function of this block is to detect functional errors in

an arithmetic operation. Using the design inputs and the opcode, the block calculates the expected residue value of the output. It compares the residue of the actual output from the operation with the expected value. In case of a mismatch, the error output is set.

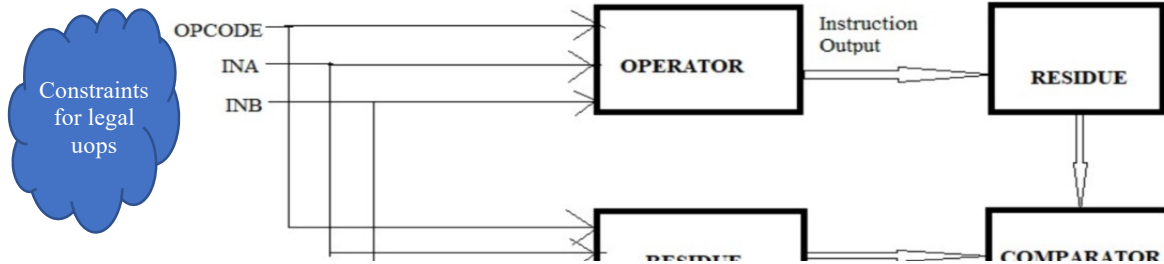


Figure 2: ALU Instruction Error Detection

The assertion that was written was simply that under the design constraints used for the pre-verified operations on the execution unit, the error flag should be zero whenever the output is valid. This check would catch issues with the residue calculation, as the actual output is already assumed to be correct. Because the residue calculation is different per opcode, there was a lot of multiplexing logic inside the residue block and this complexity led to the introduction of several bugs in residue logic.

The design was brought up as a quick addition to an existing project and a C++ model for the residue block was not ready. Getting complete coverage on this through simulation is not possible owing to the huge state space.

When the problem was posed to Formal, we decided to employ a property verification-based approach using a datapath tool. Since the execution unit was pre-verified using Formal, we could assume that the outputs from the operations were correct. Using property verification, we checked that the error output from the residue block will always be zero. There were about 30 integer opcodes and about 20 floating point opcodes. Case splitting was used to run separate proofs for each opcode. We were able to find multiple corner case scenarios in a very small timeframe. The work was well received by the design management, and Formal turned out to be a savior for the feature. We then enabled regressions and with no additional effort, were able to find new issues when RTL was changed by the designer owing to optimization/timing requirements. Formal has now become a must item in the verification checklist for this feature.

V. CONCLUDING REMARKS

The need of the hour is to make sure that the benefits of Formal proliferate to as many datapath designs as possible to ensure a shift-left in validation with enhanced confidence in design. In this paper, we have tried to present our approach to handle practical challenges in datapath Formal Verification of industrial problems. The case studies presented here exemplify our step-by-step approach in dealing with recurring problems of tool limitations, designer support, and formal convergence given the tight schedule running around these projects. Finally, we thank the Intel management and our staff for their thorough support in carrying out this work and also giving us the opportunity to present our work.

REFERENCES

- [1] Embracing Formal Verification for Datapath Designs Using Golden Specs, DVCON India 2017, M, Achutha KiranKumar V, Disha Puri, Bindumadhava SS
- [2] Am I right or Am I right: Formal Golden Specs lead the way to Embracing Datapath FV: M, Achutha KiranKumar V, Disha Puri, Bindumadhava SS, SNUG 2017
- [3] Formal Datapath Verification against SLM, Achutha KiranKumar V, Disha Puri, Bindumadhava S.S (SNUG 2016)