Challenges of Formal Verification on Deep learning Hardware accelerator

Satish, Yellinidi Dasarathanaidu, Computer Vision Engineer, Intel Corp, Bangalore, India
dasarathanaidu.satish.yellinidi@intel.com

***Abstract*—this paper describes the challenges of adapting Formal verification on complex and fully configurable hardware accelerator, solutions to the challenges, and benefits of Formal after adaptation over dynamic simulation**

Keywords—Machine Learning, Deep Learning, Hardware accelerator, Formal Verification, Formal property Verification, Transaction equivalence verification, Sequential Equivalence checks, System Verilog Properties, Assertions, Assertion Based Verification. Intellectual property, Turnaround time, Bus Functional Model, Jasper Gold, Hector

I. Introduction

In this paper, we will discuss about formal verification challenges and its solution in verifying Deep learning Hardware accelerator, where complex set of computational modes and its configuration leads to several complex scenarios, which are very hard to verify exhaustively with traditional constrained random verification. The formal solution includes divide-and-conquer on sub-units for better formal convergence, black boxing verified blocks and memories, verifying data path and control path independently, and sequence equivalence checks. Each of these solution are implemented independently and sometime results are overlapped with each other, but all results converge and gives maximum formal coverage numbers in addition to the existing constrained random verification.

Formal Verification also has positive impact on designers and verification engineers debug interaction TAT, relieving work load on machine resources and human resources, better quality of results, and better utilization of time and resources on development activities.

II. Formal Challenges on DELHA

The formal verification is implemented on Deep learning Hardware accelerator (DELHA), which is a highly programmable soft IP in a compute cluster of Inference computer engine, which provides a SIMD Grid compute and highly configurable Engine that provides fixed function accelerator for Machine Learning/ Deep Learning algorithms, which is heart of deep learning workload computation and hidden layers formulation.

As shown in the figure 1, the design units includes compute Tile clusters block where all the computational elements process the input and generate the output from several sub-units, such as control state machine, Tile blocks, Processing elements(PE), and memory blocks, such as Input SRAM, Bias and Kernel SRAM, Output SRAM.

The design also includes Load store Control unit, which manages data fetch and save from/to internal memory or system memory, through its independent channels for each of the buffers. The external interface of DELHA is through AXI and APB standard interfaces, which are connected to other IP's and fabrics respectively. The verification of this IP needs to be exhaustive as some of its application requires safety as highest priority with no room for an error.

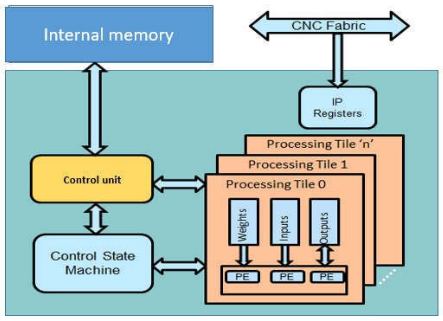

Figure 1: High level block diagram of DELHA

The formal verification challenges being faced in verifying the complex IP such as DELHA are addressed here,

- Formal tool limitations: Formal tool are having difficulty in verifying both data path and control path effectively in same run, as algorithm required for these are different to verify the complete design, run-times and effort needed for such convergence are huge.
- Result convergence: The formal result convergence in proving the given properties is directly proportional to complexity of the design. Complex design take more time to converge the result ranging from few hours to several weeks.
- Initial setup time: Formal tool need mapping of design specification into properties, writing checkers to automate result comparison, setting up bus functional models for standard interface, which may need tremendous one-time effort from both design and verification team to keep the verification inputs in sync between FV and simulation.
- Formal tool-flow expertise: Integrating the formal tool into existing verification flow needs formal domain and tool knowledge, formal expertise are scarce in the industry as FV is not always primary verification flow for the projects.

III. APPROACH TO THE FORMAL CHALLENGES

The formal verification strategy is formulated to counter the first two challenges as mentioned in above section, by dividing the verification into 3 sets with tools that works best exclusively with one of these sets.

- Formal Property Verification
- Transaction equivalence verification
- Sequential Equivalence Checks

The latter two challenges are countered as below,

The initial one-time tool-flow setup time is huge, as this requires some basic knowledge of tools and flow for initial setup, it took several weeks to understand the tools and flows and thereafter Jasper Gold and Hector tool were picked based on their efficient Formal algorithms, faster run-time, ease of integration and successful signoff of other Internal projects and recommendations from other project teams, thereafter formal verification framework/infrastructure was put in place to implement the verification.

Once the initial setup was done, consequent derivatives and setup are far quicker, and even faster than simulation based verification for all subsequent verification efforts.

The setup time includes

1. Writing system Verilog based design constraints
2. Assertion based verification Bus functional model (BFM) for standard protocol in the design
3. Automating the result checks by writing Assertion based verification properties
4. Flow integration into the existing environment and generic flow

The design constraints is extracted from existing OVM/UVM based verification infrastructure, some constraints are easily converted into assume properties and some needs imitating the logic in system Verilog and deriving properties based on that, however the constraints range has to begin with over constrained and slowly stretched to actual constrains and in some cases pushed to under- constraint just to check the stretch goals, performance and negative scenarios.

The Bus Functional Model(BFM) for standard protocol such as ABP and AXI, for configuration registers and data path transactions, are provided by tool vendor and easily integrated into the flow, if there are any non-standard protocols exist, BFM has to be written to imitate the inputs and response behavior.

The result checking are automated by writing properties which are extracted from OVM/UVM scoreboard and design intent, the properties are partially hand written and partially generated from RTL by the vendor tool Jasper SuperLint app, which has property extraction feature from given RTL. The flow integration is accomplished by collaborating with other project teams, where formal verification is used, and referring to the flow manual provided by local CAD team and working with tool vendor for any tool related issues during integration.

The tool-flow expertise is gained by attending various trainings, participating and contributing to the related forums, working with tool vendors, collaborating with projects where formal is used and hands-on with examples and demos provided by vendors and local CAD team.
The Formal verification complements the OVM/UVM based verification, as latter is implemented to verify basic and critical scenarios, and drive sequences with directed and some random tests, but with reduced number of seeds and fewer run iterations, as rest are covered by Formal verification, which is exhaustive and covers entire state space with full proof verification and complete design coverage.

A. *Formal Property Verification (FPV)*

The verification is often executed on control path of the design with Jasper Gold FPV tool. The analysis, it does is mathematically exhaustive, and covers all corner cases and possible input combinations based on the proof environment. In order to unravel the complexity of control path and its interfaces across several units and subunits, the formal property verification is executed on each of the subunits in design hierarchy, with `Divide-and-conquer` to make the verification less complex and faster state space convergence.

The sub-units which are targeted are main controllers for SRAM inside control unit, Control state machine for coordinating various controllers, Multiplication and Accumulation (MAC) blocks inside Processing Element (PE), local FSM for PE. Here the sub-units are chosen based on its complexity to trigger several parallel operation, and several loops of interaction and iteration with other components, and fully configurability nature of the units, which may lead to more cross scenario across the units, and exhaustive approach is needed to cover all possible scenarios. All these blocks are parameterizable, and hence enabling reuse of the assumptions/checks at their boundaries.

The sub-units are presented to the tool with set of local design constraints and property checkers to automate the result checks. The constraints used in OVM/UVM based verification is extracted into assume property statements at the sub-unit level and presented to the tool. The initial set of constraint are over-constrained to eliminate setup related failures, the properties are executed with formal bug hunting proof run, and many assertion failures will pop-up during initial run.
The failures are visualized with help from formal waveform and utilities. The false failures are fixed by tweaking local constraints and/or properties and re-run the iterative process, to eliminate the failures related to constraints and properties. After completely eliminating false failures, the constraints are stretched to meet the design requirements and sometimes under-constraint also help to wiggle out the design behavior on negative and/or corner case scenarios, with formal full proof run of all properties to cover the complete state space of the design.

After gaining confidence on sub-unit level formal verification, the results are converged by proving properties at the unit-level, to gain confidence on overall connectivity on control path, and design behavior on several modes combines the sub-units, and memories units are black boxed to save the properties proof running time, and reduced the state space for faster convergence. The constraints and properties are written at top-level to prove the properties behavior meeting the expectation when all sub-units/blocks combined.

Figure 2 below, shows hierarchical level verification, where each sub-units verified individually with FPV, and each of these sub-units combined into top-level, where end-to-end properties written to verify the overall functionality of the design, while black boxing the sub-units.

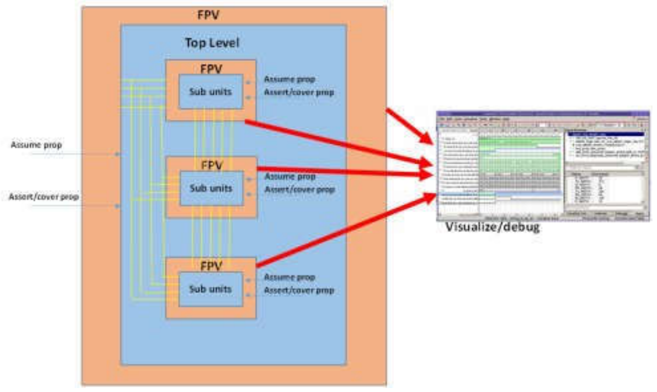

Figure 2: FPV hierarchical level verification

B. Transaction Equivalence Verification(TEV)

The data path verification is executed between RTL and C++ model with Hector tool which uses transaction based equivalence formal algorithm techniques. As shown in figure 3, the inputs to the RTL block are monitored and transferred to the reference model. The outputs computed by the reference model are then compared against the outputs of the RTL block and discrepancies are flagged. This method has found RTL bugs, but with wide data path operand values, it is often difficult to achieve complete coverage due to the

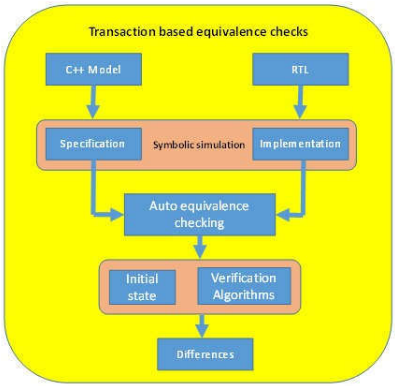

Figure 3: TEV flow chart

convergence time and effort in MAC(Multiplication and Accumulation) based operations involving int and float data-types.

An operation with two 16-bit inputs requires 4 billion stimulus patterns for exhaustive coverage. This formal approach may achieve exhaustive coverage far beyond what is practical in constrained random simulation. In corner-case discrepancies, this method is very helpful to highlight them. The design verification setup has SystemC as golden reference model, which is converted into C++ to overcome the tool limitation, the flow is initiated at sub-unit level blocks, comparing between RTL and model.

The biggest challenge here is, correlating the data flow and mapping the boundary pins between the RTL and model, to overcome this challenge, the golden model is edited to match with RTL, and checks are performed with bottom-up approach with sub-units are verified first, followed by higher hierarchical blocks. Another challenge is visualization and debugging the failures as some failures are related to bit-level transaction issues, which are sometimes hard to root-cause between the designs.

Wider data path needs to be trimmed selectively to include few bits for faster run- time and better result convergence and coverage and avoid bit level mapping issues. With proper input constraints and co-relation between the model and RTL put in place, the tool gives the best performance in identifying the functional differences between the input models.

C. *Sequential Equivalence Checks (SEC)*

This checks whether sequential releases of the designs have the same functional behavior at all external output ports. They work on both formal state-matching (1- to-1 correspondence for every register) and non-state-matching logic, and also checks internal sequential differences, this helps to avoid re-doing the verification for whole design during each release and saves both engineers time and tools/resources usage, which helps to pull the scheduled deliverables.

In this checks, the modified RTL and stable/verified RTL from previous release are given to the Jasper Gold SEC tool, the tool do the mathematical analysis of the both the RTL and finally output the differences if any, the designers debug the mismatches and understand the difference are due to newly added logic in latest RTL, apart from the known differences due to added feature or incremental changes, there shouldn't be any other mismatches which may cause latest RTL to break the previous verified RTL regions.

This checks helps to gain confidence on current RTL release, as no other part of RTL is broken due to current updates, and verifying exhaustively only the updated or new feature, which saves lots of engineers time, tools licenses, machine resources, avoid last minute surprises and faster releases.

IV. VERIFICATION COMPLETENESS

After the Formal verification flow is complete, the results are converged by mapping formal test-plan with OVM/UVM based verification test-plan, which merges the test scenarios and properties data points for better co-relation and consistency of mapped properties and assertions with design constraints, test scenarios and checkers/scoreboard.

This will envelope the complete verification with no room for bugs in the specification mapping, and properties mapping for Formal verification. The formal coverage results was mapped into coverage report of the simulation based verification to close the gaps in the area of concern where Formal and dynamic verification merge. For tests, which has convergence issues, bounded proof results with sufficient iterations covered are considered for coverage merging. Putting the above mentioned verification strategy into a graph as follows

Figure 4, captures the complete verification graph with both Formal and simulation methods used to verify the IP module

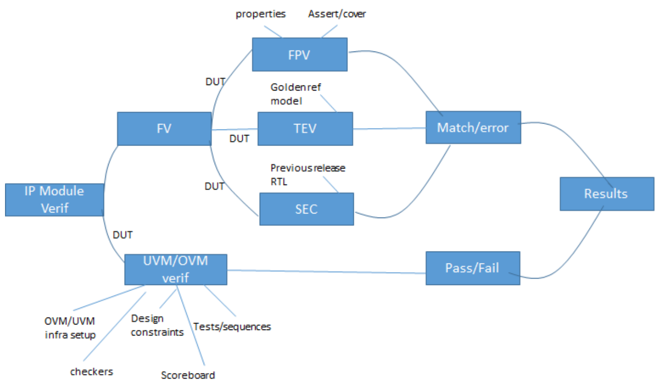

Figure 4: Complete verification graph

V. RESULTS

After resolving the challenges with Formal verification flow, we were able to accomplish following target goals with Formal verification

- Verification QoR greatly improved with FV as complementary to OVM/UVM based verification, many corner case bug were uncovered in no-time.
- Functional coverage greatly improved with few iterations.
- Saved weeks of Verification TAT by avoiding, writing test-scenarios and running with long simulation period with multiple seeds and iterations.
- Relieved workload (due to right verification strategy) on machine resources and the machines were utilized to run other tasks.
- Faster debug as results are verified with assert properties at sub-unit level, which directly points to RTL error code, which fails the property.

- Helped designers to directly verify their logic exhaustively, without test bench or verification infrastructure, this helped to avoid the dependencies and TAT between design team and verification team.

Table 1, Brief overview of the Verification target modules and effort involved for setting up the flow, bugs found with result bound for convergence.

Block	Setup time in min	Run-time to find bug in min	Bugs	Result
Control Unit	20	5	7	Full proof
Control FSM	10	1	5	Full proof
MAC	10	1	11	Bounded
data path	20	5	8	Bounded
Interface	10	1	7	Full proof

Results of Formal Verification

VI. SUMMARY

There are several challenges at every stage of Formal verification, as the IP is complex and setup is made from scratch with little knowledge about FV to start with. The verification strategy that is followed, to onion peel the challenges and succeeding the difficulties, despite several limitations helped to unravel benefits of Formal verification.

Apart from standard FV features, we also explored other advanced features provided by the vendor tool. The results and benefits of Formal verification has outplayed the challenges faced in the process, helped to extend the verification on other in-house IP's. The formal along with constrained random verification helped to unravel most unexpected corner cases bugs and helped to improve the IP verification and high confidence on quality of IP delivered to the customers.
What started as evaluation has turned into product-on-record and mandatory verification process for our IP verification due to its innumerable benefits.

ACKNOWLEDGMENT

I acknowledge and thanks our team management to approve Formal verification trails and later adapting the same in regular verification flow, also I am thankful to tool vendors, internal CAD teams for their support.

REFERENCES

[1] Formal Verification: An Essential Toolkit for Modern VLSI Design
Book by E. Thomas Schubert, Erik Seligman, and M V Achutha Kiran Kumar
[2] Vigyan Singhal, "Deploying Formal in a Simulation World," International Conference on Computer- Aided Verification, 2011.
[3] Internal Trainings on Formal verification by Support team at Intel

Tools used: Jasper Gold from Cadence, Hector from Synopsys