2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Pioneering Software Formal Verification Methodology for Firmware

Disha Puri, Intel, disha.puri@intel.com
Sparsa Roychowdhury, Intel, sparsa.roychowdhury@intel.com
Sudipa, Mandal, Intel, sudipa.mandal@intel.com

***Abstract*- The efficiency and built-in power management capabilities of a CPU are some of the biggest factors in overall data center efficiency. Saving power inside of the CPU have multiplicative savings at large scales. Currently, CPU's have added internal microcontrollers that have special firmware called PCU (power control unit). The PCU is responsible for the bulk of the power and thermal management capabilities. The PCU is connected to almost every major block of logic on the CPU die and is constantly monitoring and controlling their activity. These firmware needs to be tested well to ensure correct power and thermal management of the CPU. However, we have seen that firmware has bugs which when found later in design cycles are leading to huge waste of debug time. Even pinpointing the correct cause of failure in a flow which encompasses multiple blocks across the firmware is cumbersome. Most firmware has moved to high level language (C/C++), industry standard infrastructure and object-oriented designs which are tested using conventional simulation-based techniques. Contrary to conventional verification methodologies, formal verification can provide a much higher assurance that a design is bug free [1]. Due to this guarantee, formal verification for hardware is widely adopted and standardized in the industry. In this paper we showcase a methodology to enable formal verification of firmware written in high level languages like C/C++.**

I. Introduction

As an overall dependence on software systems has grown in last few decades, the price of failure has also increased exponentially. To cope with this, Formal Verification (FV) of software has also been an active area of research. But even though the possible impact on the industry, software FV has been done so far in pockets using multiple tools such as CBMC [2], Spark [3], Spin [4] etc. However, most of the industry uses are limited to very critical systems. Majority of the software industry still relies on traditional testing. One possible reason can be attributed to the fact that software projects have very complicated dependencies with wide-spread dynamic constructs which are very difficult for a formal tool to consume and model. To cope around this problem, tools usually translate the existing code to an intermediate FV friendly version. This can be achieved by either manual effort or by using third-party compilers. One clear disadvantage of this approach is to establish correctness of the intermediate code itself. One example could be presence of language specific run time exceptions, which may get lost in translation to another intermediate language leading to bug escapes.

C2RTL equivalence checking is an established methodology in hardware verification [5]. It has now become an industry standard to verify algorithms of encryption-decryption, compression-decompression, error correcting codes, etc., using C2RTL equivalence checking. We have also used C2RTL equivalence checking to prove correctness of arithmetic operations of CPU/GPU. Even though C2RTL meant for verification of the implementation, we have also found issues with the C/C++ specification used in C2RTL. This gave us the idea to leverage these EDA tools to verify firmware written in C/C++.

Application: Applying software FV on firmware would require multiple hurdles to be resolved.

1. How to consume the source code directly into the FV tool?
2. What is to be tested to ensure more confidence on the design?
3. Is the approach user-friendly and scalable?
4. Can we enable regressions and leverage the benefits in next gen?
5. Can we justify Returns on Investment (ROI) of the effort to all stakeholders?

Consuming the source code directly into the FV tool is a very important step. We tried many tools that claim to support C/C++, but they cannot consume the latest iteration of the programming language (e.g., C++17, C++20). In addition to that, software developers often incorporate modules that are not FV friendly. So, if we want to verify a piece of code, we need to take care of these nuances so that the FV tool can consume it. We closely collaborated with the EDA vendors so that the source code can be consumed by the tool without manual interventions.

Firmware is big and complicated. There are many dependencies, interconnects, and global variables. To prevent state space explosion as is common with big designs in FV, we have currently employed this at unit level. We created our framework based on the existing simulation unit-test framework to draw a direct parallel between the features intended to be tested by the designer and showcased the value add of formal by finding bugs missed before. It reduces the complexity of the verification process and helps us understand the design in a more modular way. This also helps the designers to formulate the properties they want to test for the module.

The next challenge is to scale the current verification technique, so that anyone with no prior knowledge of formal can run the tests created by the formal expert. Our framework closely resembles the unit-test framework. There are many advantages of this approach i) anyone who have prior experience with unit-test framework will be able to use formal for their design, ii) FV can be easily put in regression along with the unit-tests. This allows the designers to incorporate optimizations without the fear of introducing new bugs.

Section 2 describes the motivation and the challenges we faced in pioneering this methodology and possible solutions being implemented. Section 3 discussed the ROI of our approach and explains the kinds of interesting bugs found in the process. We conclude and showcase future work in Section 4.

II. DESCRIPTION

Formal Verification (FV) aims to exhaustively test a design. To generate a formal model and exhaustively test it, it is imperative that we define the boundaries under which we are claiming the verification and absence of bugs. This gels well with hardware verification since hardware also is bounded in terms of actual area and hardware available. Software, however, does not have such limitations giving designers leeway in using un-bounded, dynamic constructs to enhance speed and efficiency, which makes using FV a challenge.

To pioneer this methodology in first-of-its-kind FV attempt on firmware C++ models, we went through a barrage of challenges which we have tried to articulate below along with the proposed solutions which worked for us.

Picking the right Formal Verification tool

We surveyed multiple open-source tools such as CBMC [2], Spark [3], Seahorn [5], Isabelle [6], z3 [7] etc., but no tool could consume C++ firmware code directly without any modification. But due to recent advancements the vendor tools give us the following advantages.

1. Consume complex C++ constructs till C++17 and to some extent C++21.
2. GUI based debug support would enable ease-of-use and render scalability in future.
3. There is vendor R&D support for quick resolution of tool issues.

Compiling the Design Under Test (DUT) in the vendor tools

For the DUT to be consumed in the vendor tools, it needs to be GCC/Clang compliant. Also, for formal model to be generated, the entire dependency source code needs to be available. It is not possible to consume binaries in the tools.

We faced lots of challenges as there are lot of proprietary dependencies in the Firmware. Initially, we worked to stub these proprietary functions and replace them with GCC/Clang compliant functions. We worked with validation team to leverage their framework for our purposes to remove proprietary dependencies and replace them with GCC/Clang based abstractions.

For source dependencies, Mocklib creates stubs of the dependency functions which returns a fixed value based on the test vector upon direct calls to the function. That does not help formal, nor can we consume the entire source code as it is because it will put pressure on the tool and leads to scalability issues. To get around this issue, we built an in-house rule-based model reduction tool (filed as patent). This tool can extract only the required source dependencies from the entire list of dependency files to create a GCC/Clang complaint small compilation suite. This is then ready to be consumed by formal tools.

Auto-Formal Checks Enhancements:

As part of current methodology, firmware validation teams run static auto checks at cluster level and dumps information on possible division by zero, type conversions etc. This is very useful; however, it also contains several spurious issues and a huge dump, making it difficult for designer to find the real bugs. Formal, on the other hand, works on unit test and only provide real counterexamples, if correct constraints are provided. Not only that, but it can also provide the shortest counterexample resulting in that bug. Modern formal tools provide us with numerous such automatic properties which we call *auto-formal checks* which can run on their own as soon as the model compiles and correct constraints are set. Such checks include *division by zero*, *type conversion*, *array index out-of-bound*, *memory checks*, *pointer dereference errors*, *signed overflow*, *pointer overflow*, *NaN (*not a number*) checks*, *undefined shift* etc.

Execution Framework: Collaboration with designers/architects/validators

To enable formal verification for a DUT, we need to work closely with designers and architects. To ease the process, we have created an execution framework that lays the foundation for the collaborative effort for successful formal verification. As mentioned earlier, the firmware code has lots of proprietary dependencies which are not supported by the formal tools. With the help of the designers and validation team we were able to replace these proprietary dependencies with equivalent code that can be consumed by the FV tool.

Once proprietary codes are resolved, the next step is to resolve the dependencies of a DUT. For any DUT, the dependencies are the functions in the cone of influence (COI) of it. To apply formal verification on a DUT it is important to figure out the exact COI of it. Anything more will add complexity in the verification, and anything less will cause spurious failures during verification. We resolve these dependencies using our in-house tool.

After that, we discuss the properties of the DUT along with the constraints on the inputs with the designers. Once we resolved these, we can start our verification process. During verification, if we find any bug in the DUT, we try to resolve it with the help of the designers, and then run the checks again on the corrected source till we have no more failures.

Finally, when we fixed all bugs, and all our checks are passing we create a regression script that will automatically run our checks when the source is changed by the designer without any help from the FV engineers. This will enable the designers to experiment with their code for optimization without the worry of introducing new bugs in the design. The framework is represented as a flow in Figure 1.

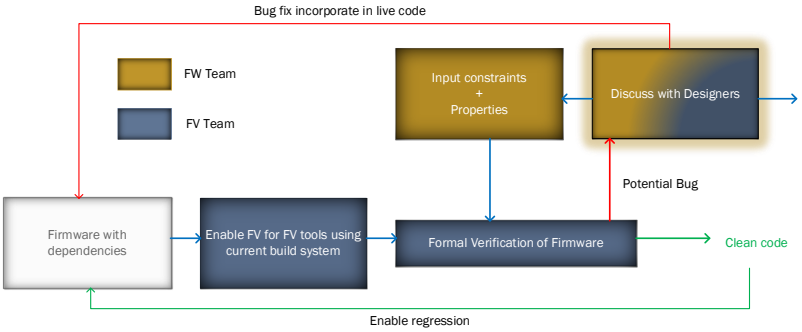

Figure 1: Flow of the execution framework.

We worked on 2 IP's and one flow as part of the proof-of-concept process. We picked these based on bug count; this was considered a high ROI option where we can make an impact by reproducing/finding bugs.

As is the norm in any validation, we referred to FAS (Firmware Architecture Spec)/HAS (High level Architecture Spec) to understand each of the designs and come up with checks and properties to prove the intent of the design. We also discussed with the designers/architects to understand their intent to enhance the validation framework with more user-based assertions.

For example, one of the IP's has two functions *get()* and *set()*. The function *set()* writes an Enum value to an Array, and *get()* reads the Array to get back the values as shown in Figure 2. We wrote properties that check the conversion done by the *set()* function from a float number to a fixed number with a specific width and precision. We also checked if we are getting the same float number back using *get()* under some constraints. These simple properties helped us to find a corner case bug in math library.

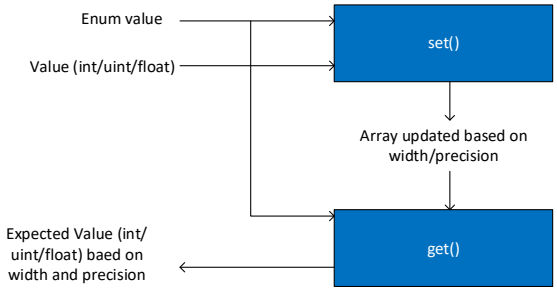

Figure 2: Getters and setters' behavior.

For another IP, we were able to write a property to find under what conditions an exponential function gives an overflow and help the designer resolve a crash. We also got several *division-by-zero* errors and *type conversion* errors. For the flow, we did *equivalence checking* against the pseudocode provided in HAS for power management and were able to find multiple unique bugs.

Integrating FV under unit test framework: Ongoing Collaboration with Validation team

This is an ongoing effort where we are working on showcasing salability of our approach. We are working to automate our flow and make it part of already existing unit test framework. To do that, we have taken help of already existing framework and remodified it for our use. Formal verification tools need to consume the real functions to create appropriate models of the code. This entire process is being automated using our in-house tool and other automation scripts being written in-house. We believe enabling this regression setup would help in wide-spread adoption of this methodology due to its ease of use and benefits across multiple generations.

III. Results & Discussion

We classify FV checks in three different categories as shown in Table 1. We map our findings to any one of these three categories.

1. **Auto formal checks:** We have been working with Cadence to provide an array of formal checks which can be run automatically by the tool once the setup is established without any user effort. These checks are generated by analyzing the code under the correct constraints on inputs. These checks can catch division-by-zero, array index out of bound, arithmetic overflows, type-conversion issues, memory-leaks, pointer overflow, dereference errors etc., to name a few.

TABLE 1
TYPES OF SOFTWARE FORMAL CHECKS

Automated Formal Checks	User Assertion Checks	Arithmetic/Logic Checks (Intent check) Equivalence check?
Array index out-of-bound, **Division by zero,** **Overflows,** **Undefined shift,** **Memory Leaks,** **Type-conversion errors, etc.**	Properties on bounds of signals, Invalidate certain scenarios, Prove bypass signals, Register accesses, etc.	Compare against the specification, Property/ Formula on algorithmic intent, etc.

In each of the POC, we were able to find *auto-formal bugs* showcasing the value add these simple checks bring to the table.

As mentioned earlier, the flow is responsible for distributing the available power to the different clusters of cores. Based on the available budget, priority, and base frequencies, the algorithm computes what should be correct frequency limits of the clusters. We found that there were many errors like *type conversion*, wrong *sorting algorithm*, *division by zero*, etc., in the code. Among them the most interesting bug was an overflow due to subtraction of two unsigned integer as shown in Figure 3.

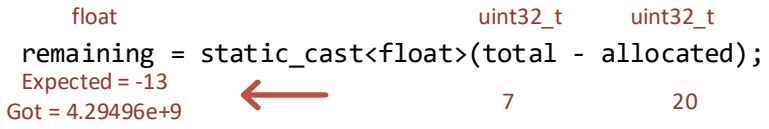

Figure 3: Overflow due to wrong type casting.

Let us discuss that issue here in more detail. Assume you have two unsinged integer variables ***total*** and ***allocated***. The ***remaining*** is the difference between the ***total*** and the ***allocated***. But, if the ***allocated*** is more than the ***total*** then, the result of subtraction is expected to be a negative number. But these variables are declared as unsigned integer thus, these variables will not be able to capture the sign of the result. This will result in an overflow. The ***remaining*** will become a value which can meet all the energy requirements of the clusters. Thus, the algorithm will allow the clusters to achieve highest possible frequency even though there is no energy budget available. Note that, this issue was also found in the post-silicon just after a few weeks of our finding.

We found a division-by-zero issue in an algorithm using the automated checks as shown in

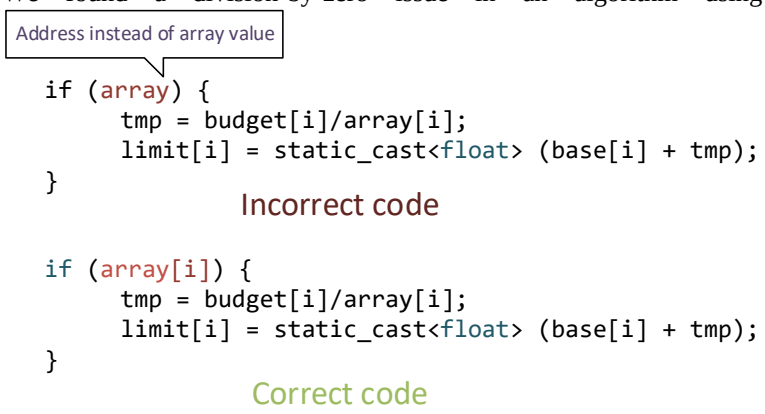

Figure 4. The issue was caused by a division by an element of the array. There was a check to ensure the denominator is not zero prior to the division. But, instead of the element, wrongly, the address of the array is checked. The address is always a positive number, causing the if statement to get executed every time even if the element is zero. Formal tool quickly found that issue using the automated checks and we were able to get the fix in no time.

We found another division-by-zero issue in the code where there is a subtraction happening in loop. And the result of the subtraction is used as a denominator. Due to the loop, there was an instance when the subtraction will eventually result in a zero, causing division-by-zero error in the code. We found this issue using the auto formal checks and debugging it by unrolling loop using the formal tool.

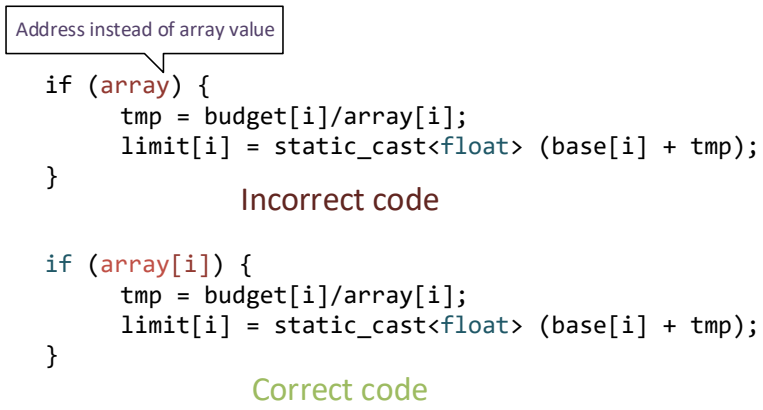

Figure 4: Division-by-zero error.

2. **User assertion checks**: These checkers are written by the formal verification engineer based on the specification or the discussions with the designer. These property checks if the DUT is obeying the specifications or not, especially in the control points. Some of these checkers are used to check if the bound on the variables are properly respected or not. In one of the designs, the designer requested us to debug a crash found in simulation. We figured out that the issues were coming from the exponential function. We did a bound analysis and provided range of inputs on which the code would give crashes as shown in Figure 5.

We also found an interesting bug in a sorting algorithm. The correctness property of a sorting algorithm is very simple. We check if the resultant array is coming in ascending (or descending) order. We wrote a property at the end of the algorithm to check if the output of the sorting algorithm is coming in ascending order or not.

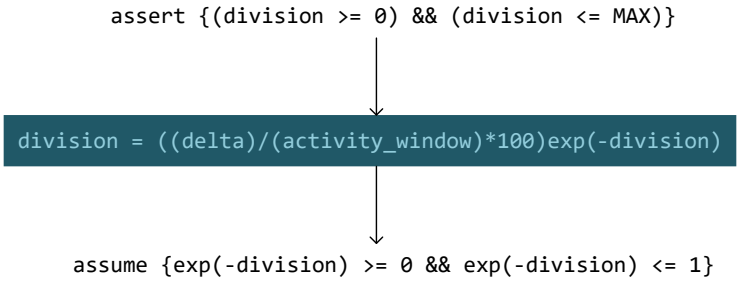

Figure 5: Bounds of exponential function.

3. **Equivalence checks**: The last type of checkers is known as equivalence checkers. To write equivalence checkers one must implement equivalent model as per the specification. These checkers mostly check the correctness of algorithmic computation.

We found an error which led to precision loss in the computation. This issue can be explained with a very easy example. Assume you are dividing 7 by 2. The correct answer should be 3.5. But to get this result we need to ensure that the numbers are represented as floating point number. But, if we represent the numbers as regular integers, the division will behave as an integer division i.e., it will not be able to represent the fractional part of the computation. So, in our example, the result will be 3, instead of 3.5. If we are expecting a floating-point result, we need to cast the variables to float before the computation. But, as shown in Figure 6, the casting happened after the computation, which should have been before the division.

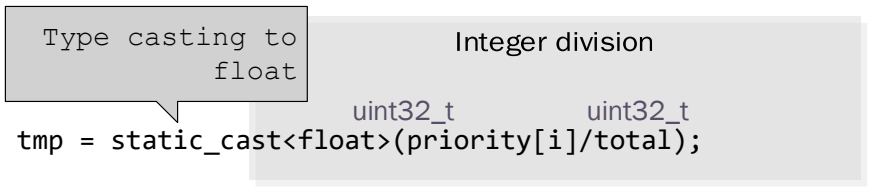

Figure 6: Precision loss due to integer division.

Overall, in 2 IP's and one flow, we were able to find 13 unique bugs in each of the categories covering all 3 categories as shown in Figure 7.

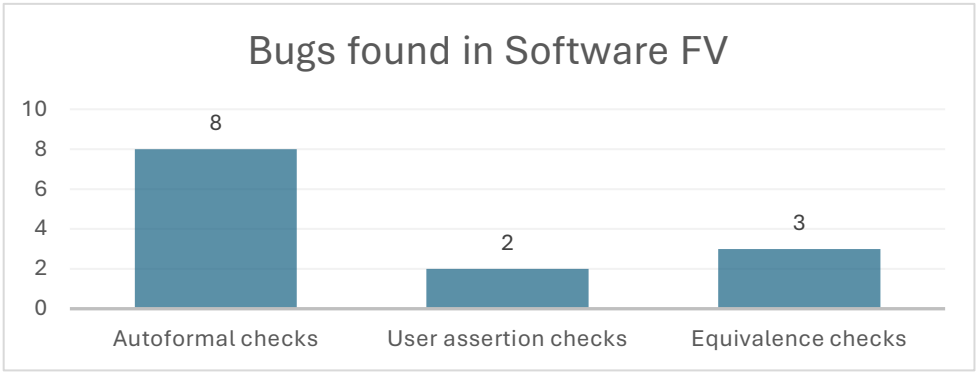

Figure 7: Bugs found in Software Formal.

IV. Conclusion and Future Plans / Use

Firmware is important for CPU to work correctly. Firmware occasionally have bugs and we need a solution that not only finds issues in the firmware, but also provides better exhaustive coverage. We have showcased how we have employed Formal Verification methodology on firmware and found many interesting bugs. Not only that, but we have also showcased a path to scalability and enabling regressions in unit test framework. We now have plans to take this initiative further and work on more IP's and Flows.

Success hinges not only on finding new bugs, but also on creating awareness on the ROI, working with vendors on ease-of-use of tool, working with architects. Once, we have acquired enough experience and expertise, we plan to leverage the same and extend to multiple areas as well. Also, we are working closely with vendors to enhance the ROI by providing coverage numbers which can be plugged back into simulation setup and provide a better coverage picture.

V. Acknowledgments

This POC project is a huge collaborative work between FVCTO, NXNE team (Sasi P Majety, Suresh Babu Juluri, Arjun Surendranath, Geethabai Biradar) and server Pcode design/val/arch teams (Ananth Krishna R). We would also like to acknowledge our management – Vinamra Goyalia and KiranKumar V M Achutha for their faith and vision in exploring a new technology and their constant guidance.

VI. References

[1] Erik Seligman, Tom Schubert, M.V. Achutha Kiran Kumar, Formal verification : an essential toolkit for modern VLSI design [second edition], Cambridge: Media, O'Reilly, 2023.

[2] "CBMC: The C Bounded Model Checker," 5 Feb 2023.

[3] C. B. M. J. F. a. J. S. P. Lourenço, "A bounded model checker for SPARK programs," in *Automated Technology for Verification and Analysis: 12th International Symposium, ATVA*, Sydney, NSW, Australia, 2014.

[4] "Spin - Formal Verification," https://spinroot.com/spin/whatispin.html.

[5] A. K. V. D. P. S. D. M, "Bringing DataPath Formal to Designers' Footsteps," in *DVCON India*, 2019.

[6] "GitHub - seahorn/seahorn: SeaHorn Verification Framework," https://github.com/seahorn/seahorn.

[7] "Isabelle," https://isabelle.in.tum.de/, 29 Feb. 2024.

[8] "GitHub - Z3Prover/z3: The Z3 Theorem Prover," https://github.com/Z3Prover/z3.