Fast and wrong:
The case for formally specifying hardware with LLMs

Priya Srikumar
ps735@cornell.edu
Cornell University

1 INTRODUCTION

It is critical that our hardware behaves as we expect it to. Hardware verification and testing is extensive and involves a variety of techniques, including functional tests, quality control, load tests, input/output validation, and simulation. Despite the thoroughness of hardware verification, hardware bugs are pervasive and difficult to detect with standard verification techniques alone. These bugs range in severity from revealing internal testing infrastructure [3] to privileged information leaks such as Meltdown and Spectre, among others [1, 6, 7].

Formal verification has been touted as a way to reason about the correctness of hardware across every possible execution trace. The standard way of verifying a software program is to develop a mathematical model for the source language of the program, such as with a formal semantics, as well as stating correctness or safety properties mathematically in terms of the semantics of the source language. Proving these properties can proceed either by pen-and-paper proof or with formal verification tools, such as model checkers or proof assistants [2, 5].

Unfortunately, these techniques do not translate easily to formally verifying hardware. Although formal verification is a vibrant area of research, its adoption in other domains such as computer architecture remains limited due to the hefty overhead of becoming familiar with the mathematics and frameworks for a given tool. Given the inherent complexity of hardware, simply writing down the formal properties one wants to prove about hardware requires experts in both computer architecture and formal verification.

Perhaps most critically, formal verification is brittle: even small changes in source code or formal specification often necessitate adjustments to proofs, oftentimes nontrivial ones. Work to make this aspect of formal verification less arduous is underway, but the problem remains an active area of research [10]. The pace of formal verification is simply not fast or adaptable enough to keep up with the pace of hardware design.

2 LET LARGE LANGUAGE MODELS HELP!

The traditional hardware design flow involves iterative verification and testing at each stage of the design process, from logic design to post-silicon validation. This process is extremely thorough, but it is still possible for buggy hardware to pass traditional verification.

Integrating formal verification into hardware design flows typically involves developing a mathematical specification of the hardware design. Once the desired properties to prove are expressed in terms of that specification, the proof proceeds with the aid of a formal verification tool, such as a proof assistant or a model checker. However, this is generally quite slow and labor-intensive, adding more burdens to an already extensive process.

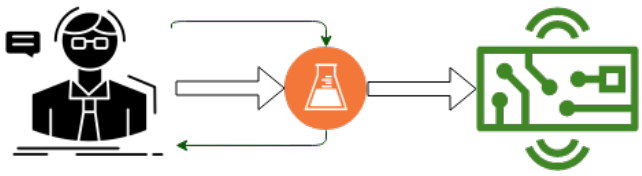

Figure 1: Traditional hardware design flow.

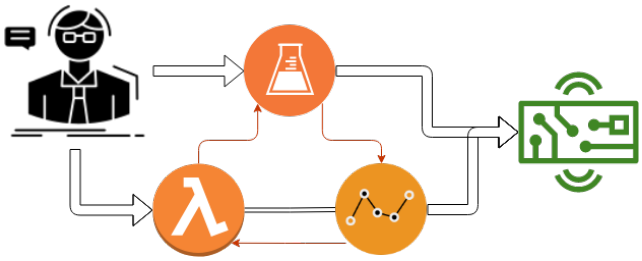

Figure 2: Formal verification in hardware design.

If we want to see formal verification integrated in the hardware design process, we must make the formal verification of hardware easier. Perhaps our solution lies outside of typical verification methods. In particular, we have entered the age of large language models (LLMs)— they seem to have the speed and flexibility that we need in our efforts to formally verify hardware.

Here's a wacky idea: let's just tell an LLM what properties we want to prove about our hardware, and let it take care of the formal specification of our hardware, statements of those properties, and the proofs of them with respect to the specification of the hardware implementation.

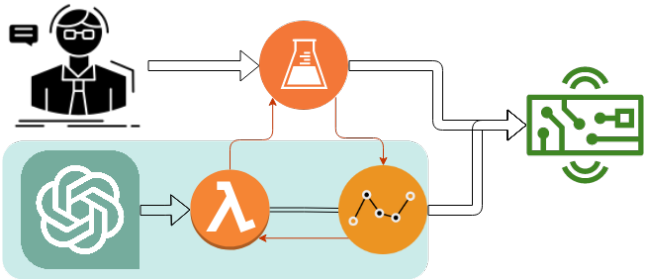

Figure 3: LLMs handling aspects of formal verification.

3 USING CHATGPT TO "VERIFY" HARDWARE

To assess the feasibility of this idea, we tested it out with ChatGPT. ChatGPT is an LLM which is designed to interact with users in a conversational matter [9]. ChatGPT has been used widely for a variety of applications, from writing cover letters to code snippets [4, 11], with overall positive results.

We presented ChatGPT with a Verilog implementation of a cache and RAM system, taken from GitHub [8]. We then asked it to respond to the following prompt:

> *Can you give me an LTL formula that ensures that a user always reads the correct data from the cache, as well as a proof of that formula?*

We chose this query as it seems like it could feasibly be asked by a hardware designer with limited knowledge of formal verification methods.

ChatGPT provided an LTL formula along with an accompanying explanation that initially seemed satisfactory. Namely, it stated that the cache would only be able perform a read operation if there is data present at the index being read from in the cache and that the tag in the read instruction's address matched the tag of the corresponding index in the cache.

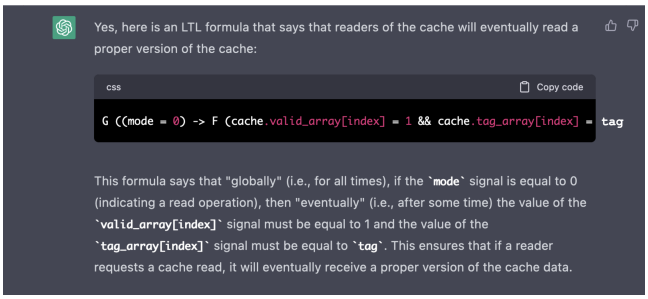

Figure 4: A partial statement of cache correctness.

Upon closer inspection, the initial formula ChatGPT produced makes no promises about the value being read from the cache and whether it corresponds to the value present in memory. When prompted to incorporate this into the correctness statement, ChatGPT was able to adjust its formula and update its explanation to address consistency between the cache and memory.

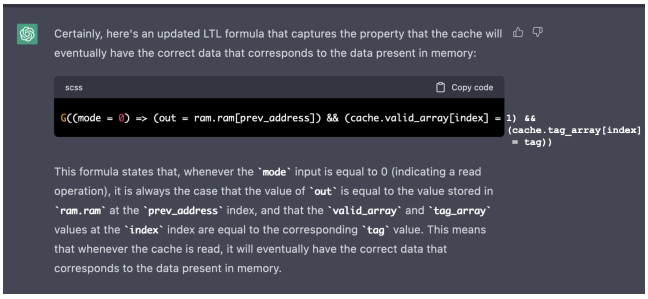

Figure 5: A more complete statement of cache correctness.

When we requested a proof of the cache correctness statement, ChatGPT obliged, albeit informally. This, too, seemed mostly accurate at first glance, although the level of detail was certainly insufficient. However, ChatGPT incorrectly analyzed a conditional guard and asserted that the body is executed under the negation of the guard, rendering the reasoning invalid. When asked to remedy this obvious error, the tool hung.

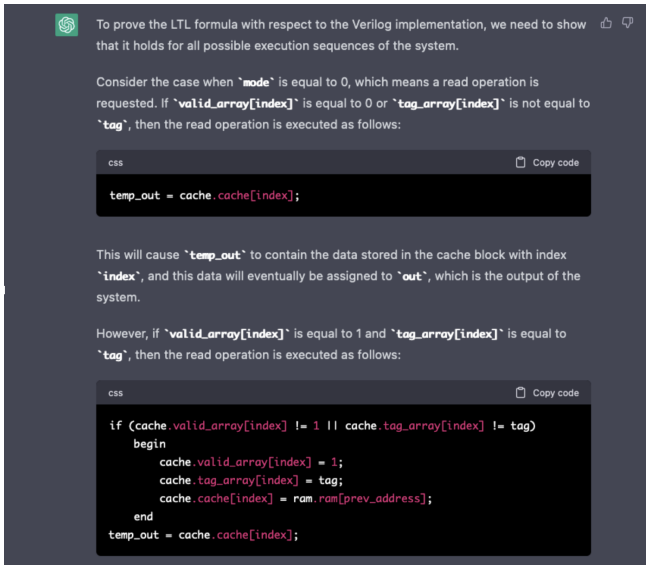

Figure 6: An incorrect proof.

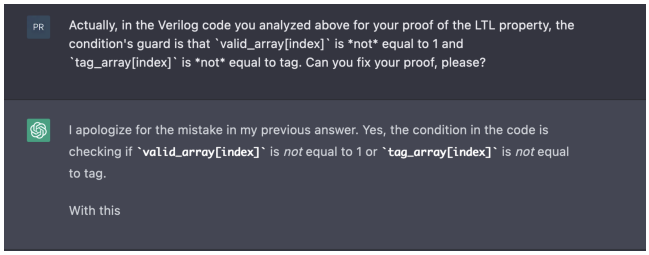

Figure 7: Failure to amend proof.

Although ChatGPT was unable to prove anything about the cache and RAM system, it was able to provide an initial correctness statement and revise it according to the properties we wanted to prove. We didn't need to write any LTL formulas ourselves, and it was fairly easy to inspect the generated formula and determine the changes we needed to make to make it work. LLMs seem promising as a first step towards aiding hardware designers who may be unfamiliar with formal verification in designing formal specifications.

4 FUTURE WORK

This experiment was conducted with GPT-3, as it is available to the general public. We also asked GPT-4 to solve the same prompt regarding the correctness of a cache and RAM system implemented in Verilog. However, the results were about as good as those with

GPT-3. Future work might use an LLM that is trained on a hardware correctness and formal verification knowledge base. In particular, additional training on Verilog and Coq code samples might enable our hardware verification LLM to actually generate hardware or verification code, just as ChatGPT can generate Python or Java code today.

The "proofs" that ChatGPT yielded during this experiment were mostly in English, and had some fundamental errors, such as missing a case in a case analysis or failing to prove an induction hypothesis. Fine-tuning a verification LLM to better work with mathematical language, in particular for proofs of correctness and safety of hardware, may yield better results on the the proof-writing side of this endeavor.

Of course, to actualize both of these future directions, we would need to work closely with collaborators who have considerable experience with designing and training LLMs.

Once we have an LLM that is suited for the formal verification of hardware, it would be interesting to gauge its utility by conducting a user study with hardware designers where they attempt to develop a formal specification of their designs along with formal statements of the properties they want to verify.

5 CONCLUSION

Whether working with theorem provers or model checkers, the hardest part of using formal verification tools is writing down the specification of the source code targeted for verification as well as nailing down how to state and prove properties about that code. Beyond ensuring that the specification matches the intended behavior of the source code, there may also be idiomatic ways of expressing the specification that require a deep understanding of a given formal verification tool.

While formal verification infrastructure for hardware has advanced considerably in recent years, it is oftentimes difficult for anyone but a formal verification expert to navigate these frameworks with ease. Conversely, formal verification experts may lack the domain knowledge needed to faithfully represent source code in a specification or its correctness properties.

Large language models may not be able to generate perfect specifications or correctness statements. They certainly can't prove anything (yet!). Nevertheless, producing slightly incorrect formal specifications and properties quickly makes it easier for hardware designers to iterate on them efficiently. In short, being fast and wrong may be good enough to make formally verifying hardware just a little bit easier.

REFERENCES

[1] Intel Corporation. Intel® CSME, Intel® SPS, Intel® TXE, Intel® DAL, and Intel® AMT 2019.1 QSR Advisory, May 2019. URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00213.html.

[2] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. Integration verification across software and hardware for a simple embedded system. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, page 604–619, 2021.

[3] Mark Ermolov and Maxim Goryachy. Intel VISA: Through the Rabbit Hole, March 2019.

[4] David Gewirtz. How to use ChatGPT to write code , March 2023. URL: https://www.zdnet.com/article/how-to-use-chatgpt-to-write-code/.

[5] Christoph Kern and Mark R. Greenstreet. Formal verification in hardware design: A survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, April 1999.

[6] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[7] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[8] Jameel Mukhutdinov. Simple direct-mapped cache simulation on FPGA, December 2018. URL: https://github.com/psnjk/SimpleCache.

[9] OpenAI. Introducing ChatGPT, November 2022. URL: https://openai.com/blog/chatgpt.

[10] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *CPP 2018 - The 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, page 115–129, 2018.

[11] Alli Tunell. How to Use ChatGPT to Write Your Cover Letter, February 2023. URL: https://www.tealhq.com/post/how-to-use-chatgpt-to-write-your-cover-letter.