High-Level Synthesis Tools should be Proven Correct

Yann Herklotz
Imperial College London, UK
yann.herklotz15@imperial.ac.uk

John Wickerson
Imperial College London, UK
j.wickerson@imperial.ac.uk

ABSTRACT

With hardware designs becoming ever more complex, and demand for custom accelerators ever growing, high-level synthesis (HLS) is increasingly being relied upon. However, HLS is known to be quite flaky, with each tool supporting subtly different fragments of the input language and sometimes even generating incorrect designs. We argue that a formally verified HLS tool could solve this issue by dramatically reducing the amount of trusted code, and providing a formal description of the input language that is supported. To this end, we are developing Vericert, a formally verified HLS tool, based on CompCert, a formally verified C compiler.

1 INTRODUCTION

High-level synthesis research and development is inherently prone to introducing bugs or regressions in the final circuit functionality.

— Andrew Canis [7]
Co-founder of LegUp Computing

Research in high-level synthesis (HLS) often concentrates on performance: trying to achieve the lowest area with the shortest run-time. What is often overlooked is ensuring that the HLS tool is correct, which means that it outputs hardware designs that are equivalent to the behavioural input.

When working with HLS tools, it is often assumed that they transform the behavioural input into a semantically equivalent design [17]. However, this is not the case, and as with all complex pieces of software there are bugs in HLS tools as well. For example, Vivado HLS was found to incorrectly apply pipelining optimisations[1] or generate incorrect designs silently when straying outside the supported fragment of C.[2] These types of bugs are difficult to identify, and exist because it is not quite clear firstly what input these tools support, and secondly whether the output design actually behaves the same as the input.

Our position: We believe that a formally verified HLS tool could be the solution to these problems. It not only guarantees that the output is correct, but also brings a formal specification of the input and output language semantics. These are the only parts of the compiler that need to be trusted, and if these are well-specified, then the behaviour of the resulting design can be fully trusted. In addition to that, if the semantics of the input language are taken from a tool that is widely trusted already, then there should not be any strange behaviour; the resultant design will either behave exactly like the input specification, or the translation will fail early at compile time. To this end, we are building a formally verified HLS tool called Vericert [15].

In what follows, we will argue our position by presenting several possible *objections* to our position, and then responding to each in turn.

2 ARGUMENTS AGAINST FORMALISED HLS

Objection 1: People should not be designing hardware in C to begin with. *Formally verifying HLS of C is the wrong approach. C should not be used to design hardware, let alone hardware where reliability is crucial. First of all, HLS tools are not unreliable in terms of correctness, but more so in terms of the quality of the hardware, which can be quite unpredictable [21]. For example, when writing C for an HLS tool, small changes in the input could result in large differences in area and performance of the resulting hardware. On the other hand, there have been many efforts to formally verify the translation of high-level hardware description languages like Bluespec with Kôika [5], to formalise the synthesis of Verilog into technology-mapped net-lists with Lutsig [19], or to formalise circuit design in Coq itself to ease design verification [8, 23].*

Our response: Verifying HLS is also important. First, C is often the starting point for hardware designs, as initial models are written in C to produce a quick prototype [13], so it is only natural to continue using C when designing the hardware. Not only is HLS from C becoming more popular, but much of that convenience comes from the easy behavioural testing that HLS allows to ensure correct functionality of the design [17]. This assumes that HLS tools are correct. Finally, even though unpredictability of the output of HLS tools might seem like the largest issue, working on a functionally correct HLS tool provides a good baseline to work on improving the predictability of the output as well. Reasoning about correctness could maybe be extended with proofs about the variability of the generated output hardware, thereby also improving the quality of the output. In this vein, CompCert [18], an existing formally verified C compiler, has recently been extended with a proof of preservation of constant-time [3], and a similar approach could be taken to to prove properties about the preservation of the hardware area or performance.

Objection 2: HLS tools are already commonly used in industry, so they are clearly already reliable enough.

Our response: They are widely used, but they are also widely acknowledged to be quite flaky. In prior work [14], we have shown that on average 2.5% of randomly generated C programs, tailored to the specific HLS tool, end up with incorrect designs. These bugs

[1] https://bit.ly/vivado-hls-pipeline-bug
[2] https://bit.ly/vivado-hls-pointer-bug

were reported and confirmed to be new bugs in the tools, demonstrating that existing internal tests did not catch them.

Objection 3: Existing approaches for testing or formally verifying hardware designs are sufficient for ensuring reliability. *Besides the use of test-benches to test designs produced by HLS, there has been research on performing equivalence checks between the output design and the behavioural input, focusing on creating translation validators [22] to prove equivalence between the design and the input code, while supporting various optimisations such as scheduling [9, 16, 25] or code motion [2, 10].*

Our response: Existing verification techniques for checking the output of HLS tools may not be enough to catch these bugs reliably. Checking the final design against the original model using a test-bench may miss edge cases that produce bugs. In addition to that, these test-benches need to be kept up to date when new features are introduced. Translation validation seems like a good solution, as the tool can be automatically checked as it generates hardware, and any potential issues in the hardware are reported at the time the hardware is generated. However, this is not a perfect solution either, as there is no guarantee that the translation validation proofs really compose with each other. In response, equivalence checkers are often designed to check the translation from start to finish, but this is computationally expensive, as well as possibly being highly incomplete. In addition to that, these translation validation algorithms have not been validated mechanically by a theorem prover. Therefore, there is no guarantee that the validator itself is correct, which would require more testing.

The radical solution to this problem is to formally verify the whole tool. This proved successful for CompCert [18], for example, which is a formally verified C compiler written in Coq [12]. The reliability of this formally verified compiler was demonstrated by Csmith [24], a random, valid C generator, finding more than 300 bugs in GCC and Clang, but no bugs in the verified parts of CompCert.

Objection 4: HLS applications don't require the levels of reliability that a formally verified compiler affords. *One might argue that developing a formally verified tool in a theorem prover and proving correctness theorems about it might take too long, and that HLS tools specifically do not need that kind of reliability. Indeed, in our experience developing a verified HLS tool called Vericert [15] based on CompCert, we found that it normally takes* $5\times$ *or* $10\times$ *longer to prove a compiler pass correct compared to writing the algorithm.*

Our response: However, proving the correctness of the HLS tool proves the absence of any bugs according to the language semantics, meaning much less time has to be spent on fixing bugs. In addition to that, verification also forces the algorithm to deal with many different edge cases that may be hard to identify normally.

Objection 5: Any HLS tool that is simple enough for formal verification to be feasible won't produce sufficiently optimised designs to be useful. *If that is the case, then the verification effort could be seen as useless, as it could not be used.*

Our response: We think that even a verified HLS tool can be comparable in performance to a state-of-the-art unverified HLS tool. Taking Vericert as an example, which does not currently include many optimisations, we found that performing comparisons between Vericert and LegUp [6], we found that the speed and area were comparable ($1\times$ - $1.5\times$) to that of LegUp without LLVM optimisations and without operation chaining. With those optimisations fully turned on, Vericert is around $4.5\times$ slower than LegUp, with half the speed up being due to LLVM.

There are many optimisations that need to be added to Vericert to turn it into a viable and competitive HLS tool. First of all, a good scheduling implementation that supports operation chaining and pipelined operators is critical. Our main focus is implementing scheduling based on systems of difference constraints [11], which is the same algorithm LegUp uses. With this optimisation turned on, Vericert is only $2\times$ to $3\times$ slower than fully optimised LegUp, with a slightly larger area. The scheduling step is implemented using verified translation validation, which means that the scheduling algorithm can be tweaked and optimised without ever having to touch the correctness proof.

Objection 6: Even a formally verified HLS tool can't give absolute guarantees about the hardware it produces.

Our response: It is true that a verified tool is still allowed to fail at compile time, which means that no output is produced despite the valid input. However, this is mostly a matter of putting more engineering work into the tool to make it more complete. Bugs are easier to identify as they will induce tool failures at compile time.

In addition to that, specifically for an HLS tool taking C as input, undefined behaviour will allow the HLS tool to behave any way it wishes. This becomes even more important when passing the C to a verified HLS tool, because if it is not free of undefined behaviour, then none of the proofs will hold. Extra steps therefore need to be performed to ensure that the input is free of any undefined behaviour, by using a tool like VST [1] for example.

Finally, the input and output language semantics need to be trusted, as the proofs only hold as long as the semantics are a faithful representation of the languages. In Vericert this comes down to trusting the C semantics developed by CompCert [4] and the Verilog semantics that we adapted from Lööw and Myreen [20].

3 CONCLUSION

In conclusion, we have argued that HLS tools should be formally verified, and through our Vericert prototype, we have demonstrated that doing so is feasible. Even though the performance does not yet match state-of-the-art HLS tools, as more and more optimisations are implemented and formally verified, similar performance should be achievable. We believe that Vericert has the potential to raise the standard of reliability across the HLS field. It also has the potential to bring HLS to a new domain: designers of security- or safety-critical hardware, who are currently forced to design at very low levels of abstraction in order to minimise their trusted computing base.

ACKNOWLEDGMENTS

We acknowledge financial support from the Research Institute on Verified Trustworthy Software Systems (VeTSS), which is funded by the UK National Cyber Security Centre (NCSC).

REFERENCES

[1] Andrew W. Appel. 2011. Verified Software Toolchain. In *Programming Languages and Systems*, Gilles Barthe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17.

[2] K. Banerjee, C. Karfa, D. Sarkar, and C. Mandal. 2014. Verification of Code Motion Techniques Using Value Propagation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 8 (Aug 2014), 1180–1193. https://doi.org/10.1109/TCAD.2014.2314392

[3] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal verification of a constant-time preserving C compiler. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–30. https://doi.org/10.1145/3371075

[4] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning* 43, 3 (01 Oct 2009), 263–288. https://doi.org/10.1007/s10817-009-9148-3

[5] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965

[6] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. Association for Computing Machinery, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

[7] Andrew Christopher Canis. 2015. *Legup: open-source high-level synthesis research framework*. Ph.D. Dissertation.

[8] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: a Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (Aug. 2017), 30 pages. https://doi.org/10.1145/3110268

[9] R. Chouksey and C. Karfa. 2020. Verification of Scheduling of Conditional Behaviors in High-Level Synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2020), 1–14. https://doi.org/10.1109/TVLSI.2020.2978242

[10] R. Chouksey, C. Karfa, and P. Bhaduri. 2019. Translation Validation of Code Motion Transformations Involving Loops. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 7 (July 2019), 1378–1382. https://doi.org/10.1109/TCAD.2018.2846654

[11] J. Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *43rd ACM/IEEE Design Automation Conference*. 433–438. https://doi.org/10.1145/1146909.1147025

[12] Thierry Coquand and Gérard Huet. 1986. *The calculus of constructions*. Ph.D. Dissertation. INRIA.

[13] Dan Gajski, Todd Austin, and Steve Svoboda. 2010. What input-language is the best choice for high level synthesis (HLS)?. In *Design Automation Conference*. 857–858. https://doi.org/10.1145/1837274.1837489

[14] Yann Herklotz, Zewei Du, Nadesh Ramanathan, and John Wickerson. 2021. An Empirical Study of the Reliability of High-Level Synthesis Tools. In *29th IEEE International Symposium on Field-Programmable Custom Computing Machines*. https://yannherklotz.com/docs/drafts/fuzzing_hls.pdf (to appear).

[15] Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal Verification of High-Level Synthesis. (2021). https://yannherklotz.com/docs/drafts/formal_hls.pdf (under review).

[16] C Karfa, C Mandal, D Sarkar, S R. Pentakota, and Chris Reade. 2006. A Formal Verification Method of Scheduling in High-level Synthesis. In *Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED '06)*. IEEE Computer Society, Washington, DC, USA, 71–78. https://doi.org/10.1109/ISQED.2006.10

[17] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen. 2019. Are We There Yet? a Study on the State of High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (May 2019), 898–911. https://doi.org/10.1109/TCAD.2018.2834439

[18] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[19] Andreas Lööw. 2021. Lutsig: A Verified Verilog Compiler for Verified Circuit Development. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Virtual, Denmark) *(CPP 2021)*. Association for Computing Machinery, New York, NY, USA, 46–60. https://doi.org/10.1145/3437992.3439916

[20] Andreas Lööw and Magnus O. Myreen. 2019. A Proof-producing Translator for Verilog Development in HOL. In *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering* (Montreal, Quebec, Canada) *(FormaliSE '19)*. IEEE Press, Piscataway, NJ, USA, 99–108. https://doi.org/10.1109/FormaliSE.2019.00020

[21] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable Accelerator Design with Time-Sensitive Affine Types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 393–407. https://doi.org/10.1145/3385412.3385974

[22] A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernhard Steffen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166.

[23] Satnam Singh. [n.d.]. Silver Oak. https://github.com/project-oak/silveroak

[24] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[25] Youngsik Kim, S. Kopuri, and N. Mansouri. 2004. Automated formal verification of scheduling process using finite state machines with datapath (FSMD). In *International Symposium on Signals, Circuits and Systems. Proceedings, SCS 2003. (Cat. No.03EX720)*. 110–115. https://doi.org/10.1109/ISQED.2004.1283659