# PLDI: G: Formal Verification of High-Level Synthesis

Yann Herklotz

## Problem and Motivation

As latency, throughput and energy efficiency become increasingly important, custom hardware accelerators are being designed for numerous applications. Alas, designing these accelerators can be a tedious and error-prone process using a hardware description language (HDL) such as Verilog. An attractive alternative is *high-level synthesis* (HLS), in which hardware designs are automatically compiled from software written in a high-level language like C. Modern HLS tools such as LegUp [1], Vivado HLS [2], Intel i++ [3], and Bambu HLS [4] promise designs with comparable performance and energy-efficiency to those hand-written in an HDL [5], while offering the convenient abstractions and rich ecosystems of software development. But existing HLS tools cannot always guarantee that the hardware designs they produce are equivalent to the software they were given, and this undermines any reasoning conducted at the software level.

Indeed, there are reasons to doubt that HLS tools actually *do* always preserve equivalence. For instance, Vivado HLS has been shown to apply pipelining optimisations incorrectly[1] or to silently generate wrong code should the programmer stray outside the fragment of C that it supports.[2] Meanwhile, Lidbury et al. [6] had to abandon their attempt to fuzz-test Altera's (now Intel's) OpenCL compiler since it "either crashed or emitted an internal compiler error" on so many of their test inputs. More recently, Herklotz et al. [7] fuzz-tested three commercial HLS tools using Csmith [8], and despite restricting the generated programs to the C fragment explicitly supported by all the tools, they still found that on average 2.5% of test cases generated a design that did not match the behaviour of the input.

### Existing workarounds

Aware of the reliability shortcomings of HLS tools, hardware designers routinely check the generated hardware for functional correctness. This is commonly done by simulating the design against a large test-bench. But unless the test-bench covers all inputs exhaustively, which is often infeasible, there is a risk that bugs remain.

An alternative is to use *translation validation* [9] to prove the input and output equivalent. Translation validation has been successfully applied to several HLS optimisations [10]. But translation validation is an expensive task, especially for large designs, and it must be repeated every time the compiler is invoked. For example, the translation validation for Catapult C [11] may require several rounds of expert 'adjustments' [12, p. 3] to the input C program before validation succeeds. And even when it succeeds, translation validation does not provide watertight guarantees unless the validator itself has been mechanically proven correct, which is seldom the case.

Our position is that none of the above workarounds are necessary if the HLS tool can simply be trusted to work correctly.

### Our solution

We have designed a new HLS tool in the Coq theorem prover and proved that any output it produces always has the same behaviour as its input. Our tool, called Vericert, is automatically extracted to an OCaml program from Coq, which ensures that the object of the proof is the same as the implementation of the tool. Vericert is built by extending the CompCert verified C compiler [13] with a new hardware-specific intermediate language and a Verilog back end. It supports most C constructs, including integer operations, function calls, local arrays, structs, unions, and general control-flow statements, but currently excludes support for case statements, function pointers, recursive function calls, integers larger than 32 bits, floats, and global variables.

## Background and Related Work

Most practical HLS tools [1, 2, 14] fit into the category of usable tools that take high-level inputs. On the other spectrum, there are tools such as BEDROC [15] for which there is no practical tool, and even though it is described as high-level synthesis, it more closely resembles today's hardware synthesis tools.

Ongoing work in translation validation [9] seeks to prove equivalence between the hardware generated by an HLS tool and the original behavioural description in C. An example of a tool that implements this is Mentor's Catapult [11], which tries to match the states in the 3AC description to states in the original C code after an unverified translation. Using translation validation is quite effective for verifying

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

complex optimisations such as scheduling [10] or code motion [16], but the validation has to be run every time the HLS is performed. In addition to that, the proofs are often not mechanised or directly related to the actual implementation, meaning the verifying algorithm might be wrong and hence could give false positives or false negatives.

Finally, there are a few relevant mechanically verified tools. First, Kôika is a formally verified translation from a core fragment of BlueSpec into a circuit representation which can then be printed as a Verilog design. This is a translation from a high-level hardware description language into an equivalent circuit representation, so is a different approach to HLS. Lööw and Myreen [17] used a verified translation from HOL4 code describing state transitions into Verilog to design a verified processor, which is described in Lööw et al. [18]. In addition to that, there is also work on formally verifying a synthesis tool to transform, which can transform hardware descriptions into low-level netlists [19]. Their approach translated a shallow embedding in HOL4 into a deep embedding of Verilog. Perna and Woodcock [20] designed a formally verified translation from a deep embedding of Handel-C [21], which is translated to a deep embedding of a circuit. Finally, Ellis [22] used Isabelle to implement and reason about intermediate languages for software/hardware compilation, where parts could be implemented in hardware and the correctness could still be shown.

### Verilog Semantics

The Verilog standard is quite large [23, 24], but the syntax and semantics can be reduced to a small subset that Vericert needs to target. This section also describes how Vericert's representation of memory differs from CompCert's memory model [25].

The Verilog semantics we use is ported to Coq from a semantics written in HOL4 by Lööw and Myreen [17] and used to prove the translation from HOL4 to Verilog [18]. This semantics is quite practical as it is restricted to a small subset of Verilog, which can nonetheless be used to model the hardware constructs required for HLS. The main features that are excluded are continuous assignment and combinational always-blocks.

## Uniqueness of the Approach

### Extension to CompCert

The main work flow of Vericert is given in Figure 1, which shows those parts of the translation that are performed in CompCert, and those that have been added. First, the intermediate language had to be chosen at which to branch off. The three-address code (3AC)[3] was chosen as the branching point. Any earlier, such as CminorSel, and the language is not yet in an instruction format that can easily be converted into hardware. Any later, such as LTL, and CompCert will have already performed optimisations that are not optimal for HLS, such as register allocation with extra registers spilling onto the stack. Hardware normally has an abundant number of registers, and interaction with memory is often quite slow and is often sequential, it is therefore beneficial to have as many registers as possible to exploit as much instruction parallelism as possible. In addition to that, 3AC is a good intermediate language to branch off of, because it is quite similar to LLVM IR, which is already used by many HLS tools as their intermediate language.

From 3AC, three passes were added to translate the instruction level code into hardware:

**HTL generation** The first step is to generate HTL, which is a new intermediate language that models a finite state machine with data-path (FSMD). This language consists of two maps from states to Verilog statements that correspond to that state, describing the control logic and data-path of the FSMD. The translation from 3AC to HTL is quite straightforward, as it translates each 3AC instruction into its Verilog equivalent, and creates the necessary state changes in the control logic to follow the same control flow as 3AC. The main complexity of this translation was the translation of memory operations on the stack. As there is no concept of stack in HTL and in hardware in general, a word-addressed Verilog array is used instead. Any addresses into the stack therefore need to be divided by four to obtain the correct address for the corresponding array.

**RAM insertion** The second step is to insert a proper RAM interface instead, which is used instead of accessing a Verilog array directly:

```
always @(negedge clk)
  if ({u_en != en}) begin
    if (wr_en) stack[addr] <= d_in;
    else d_out <= stack[addr];
    en <= u_en;
  end
```

Instead of directly accessing the `stack` array in the data-path, a request is made to the RAM interface shown above to load or store a value to memory. Firstly, always-block of the RAM interface is triggered at the negative edge of the clock, whereas the control logic and the data-path are triggered at the positive edge of the clock. This means that the loads from memory only take two clock cycles instead of three, and stores take one clock cycle instead of two. In addition to that, the first if-statement compares the equality of the user enable `u_en`, which is toggled by the data-path when it wants to use the RAM,

[3] This is known as register transfer language (RTL) in the CompCert literature. '3AC' is used in this paper instead to avoid confusion with register-transfer level (RTL), which is another name for the final hardware target of the HLS tool.

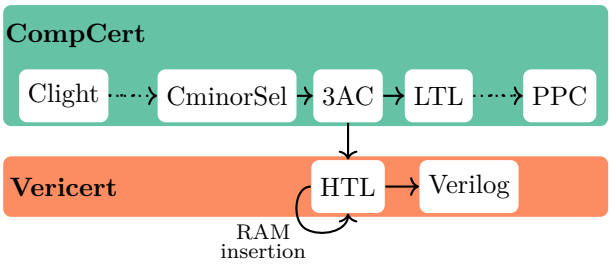


Figure 1: Vericert as a Verilog back end to CompCert

and the internal enable `en`, which is controlled by the RAM itself. This feature allows the RAM to disable itself, therefore simplifying the proof of this compiler pass dramatically.

**Verilog generation** The final step is to generate the Verilog from HTL. This pass is conceptually simple, as the data-path and control logic maps are translated to case statements inside a positive edge triggered always-block. In addition to that, the RAM interface which was only modelled abstractly in HTL is actually implemented in Verilog, and proven to behave equivalently for all possible inputs.

## Changes to the Semantics

Five changes were made to the semantics proposed by Lööw and Myreen [17] to make it a suitable HLS target.

**Adding array support**: The main change is the addition of support for arrays, which are needed to model RAM in Verilog. RAM is needed to model the stack in C efficiently, without having to declare a variable for each possible stack location.

**Adding negative edge support**: To support memory inference efficiently and create and reason about a circuit that executes at the negative edge of the clock, support for the negative edge triggered always-blocks was added to the semantics. The main execution of the module is split into a positive edge execution and a negative edge execution.

**Adding declarations**: Explicit support for declaring inputs, outputs and internal variables was added to the semantics to make sure that the generated Verilog also contains the correct declarations. This adds some guarantees to the generated Verilog and ensures that it synthesises and simulates correctly.

**Removing support for external inputs to modules** Support for receiving external inputs was removed from the semantics for simplicity, as these are not needed for an HLS target. The main module in Verilog models the main function in C, and since the inputs to a C function should not change during its execution, there is no need for external inputs for Verilog modules.

**Simplifying representation of bitvectors**: Finally, we use 32-bit integers to represent bitvectors rather than arrays of Booleans. This is because Vericert (currently) only supports types represented by 32 bits.

## Proof of equivalence of the passes

The main correctness theorem is analogous to that stated in CompCert [13]: for all Clight source programs $C$, if the translation to the target Verilog code succeeds, and $C$ has safe observable behaviour $B$ when executed, then the target Verilog code will have the same behaviour $B$. Here, a 'safe' execution is one that either converges or diverges, but does not "go wrong". If the program does admit some wrong behaviour (like undefined behaviour in C), the correctness theorem does not apply. A behaviour, then, is either a final state (in the case of convergence) or divergence. In CompCert, a behaviour is also associated with a trace of I/O events, but since external function calls are not supported in Vericert, this trace will always be empty for us. Note that the compiler is allowed to fail and not produce any output; the correctness theorem only applies when the translation succeeds.

**Theorem 1** *For any safe behaviour $B$, whenever the translation from $C$ succeeds and produces Verilog $V$, then $V$ has behaviour $B$ only if $C$ has behaviour $B$.*

$$\forall C, V, B \in \mathit{Safe}, \mathit{HLS}(C) = \mathit{OK}(V) \wedge V \Downarrow B \implies C \Downarrow B.$$

The theorem is a 'backwards simulation' result (from target to source), following the terminology used in the CompCert literature. The theorem does not demand the 'if' direction too, because compilers are permitted to resolve any non-determinism present in their source programs. In practice, Clight programs are all deterministic, as are the Verilog programs in the fragment we consider. This means that we can prove the correctness theorem above by first inverting it to become a forwards simulation result, following standard CompCert practice.

Furthermore, to prove the forward simulation, it suffices to prove forward simulations between each intermediate language, as these results can be composed to prove the correctness of the whole HLS tool.

# Results and Contributions

Figure 2 compares the cycle counts of 27 PolyBench/C programs executed by Vericert and different optimisation levels of LegUp, an existing, unverified and optimising HLS tool. First, LLVM optimisations are turned off in LegUp, which correspond to general compiler optimisations, referred to as LegUp w/o opt. Next, we additionally turned off operation chaining, a common HLS optimisation that places data-dependent instructions into the same cycle, and therefore reduces the cycle count of the final hardware. This version of LegUp is referred to as LegUp w/o opt+chain.

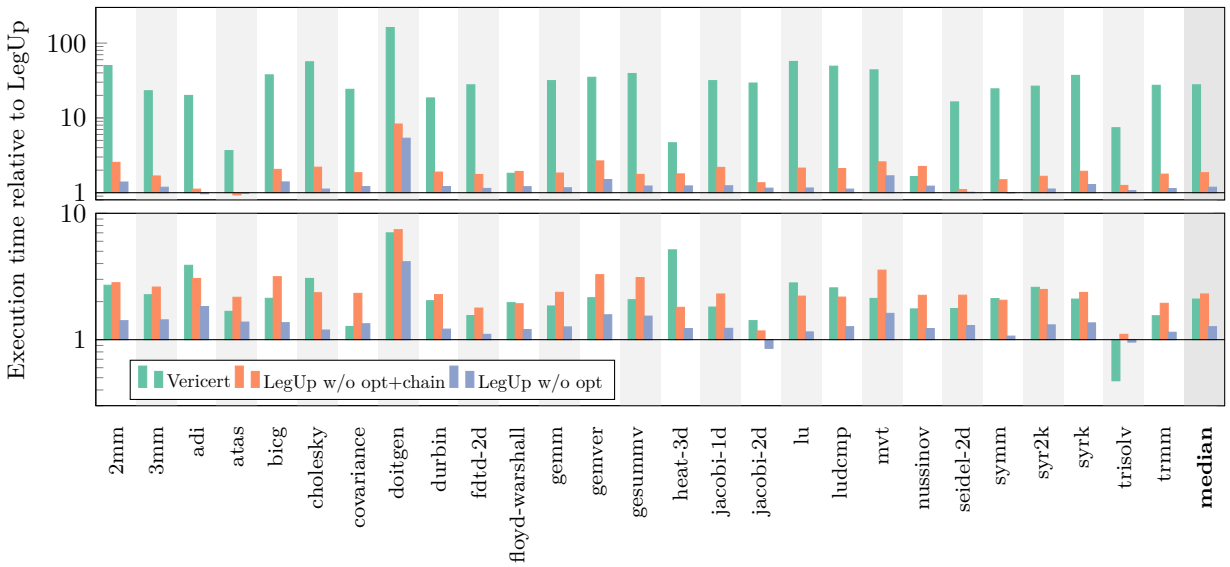


Figure 2: The top graph shows the execution time of Vericert on PolyBench/C with division/modulo operations enabled, comparing against LegUp without LLVM optimisations and without operation chaining and LegUp without front end LLVM optimisations relative to optimised LegUp. The bottom graph shows the same comparison with the division/modulo operations replaced by an iterative algorithm.

Each graph uses optimised LegUp as the baseline. The top graph in Figure 2 contains relative execution time of the standard PolyBench/C benchmarks, whereas the graph below it shows the relative execution time the benchmarks where division/modulo operations were replaced by an iterative algorithm using shifts and adds. When division/modulo operations are present LegUp designs execute around 27× faster than Vericert designs. However, when division/modulo operations are replaced by the iterative algorithm, LegUp designs are only 2× faster than Vericert designs. However, the benchmarks with division/modulo replaced show that Vericert actually achieves the same execution speed as LegUp w/o opt+chain, which is encouraging, and shows that the hardware generation is following the right steps. The execution time is calculated by multiplying the maximum frequency that the FPGA can run at with this design, by the number of clock cycles that are needed to complete the execution. We can therefore analyse each separately.

First, looking at the difference in clock cycles, Vericert produces designs that have around 4.5× as many clock cycles as LegUp designs in both cases, when division/modulo operations are enabled as well as when they are replaced. This performance gap can be explained in part by LLVM optimisations, which seem to account for a 2× decrease in clock cycles, as well as operation chaining, which decreases the clock cycles by another 2×. The rest of the speed-up is mostly due to LegUp optimisations such as scheduling and memory analysis, which are designed to extract parallelism from input programs. This gap does not represent the performance cost that comes with formally proving a HLS tool. Instead, it is simply a gap between an unoptimised Vericert versus an optimised LegUp. As we improve Vericert by incorporating further optimisations, this gap should reduce whilst preserving the correctness guarantees.

Secondly, looking at the maximum clock frequency that each design can achieve, LegUp designs achieve 8.2× the maximum clock frequency of Vericert when division/modulo operations are present. This is in great contrast to the maximum clock frequency that Vericert can achieve when no divide/modulo operations are present, where Vericert generates designs that are actually 2× better than the frequency achieved by LegUp designs. The dramatic discrepancy in performance for the former case can be largely attributed to Vericert's naïve implementations of division and modulo operations, as Vericert currently uses the built-in Verilog division and modulo operators. These create large circuits that need to complete within one clock cycle, therefore reducing the overall clock speed that the design can run at. Indeed, Vericert achieved an average clock frequency of just 13MHz, while LegUp managed about 111MHz. After replacing the division/modulo operations with our own C-based implementations, Vericert's average clock frequency becomes about 220MHz. This improvement in frequency can be explained by the fact that LegUp uses a memory controller to manage multiple RAMs using one interface, which is not needed in Vericert as a single RAM is used for the memory.

## Contributions

- We presented Vericert, the first mechanically verified HLS tool that compiles C to Verilog.

- We stated the correctness theorem of Vericert with respect to an existing semantics for Verilog due to Lööw and Myreen [17]. We described how we extended this semantics to make it suitable as an HLS target.

- We describe how we proved the correctness theorem. The proof follows standard CompCert techniques – forward simulations, intermediate specifications, and determinism results – but we encountered several challenges peculiar to our hardware-oriented setting. These include handling discrepancies between byte- and word-addressable memories, different handling of unsigned comparisons between C and Verilog, correctly mapping CompCert's memory model onto a finite Verilog array and finally correctly rearranging memory reads and writes so that these behave properly as a RAM in hardware.

- Finally, we evaluated Vericert on the PolyBench/C benchmark suite [26], and compared the performance of our generated hardware against an existing, unverified HLS tool called LegUp [1]. We showed that Vericert generates hardware that is 27× slower (2× slower in the absence of division) and 1.1× larger than that generated by LegUp. We intend to bridge this performance gap in the future by introducing (and verifying) HLS optimisations of our own, such as scheduling and memory analysis.

# References


[1] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Helge Anderson, Stephen Dean Brown, and Tomasz S. Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *FPGA*, pages 33–36. ACM, 2011. doi: 10.1145/1950413.1950423.

[2] Xilinx. Vivado high-level synthesis, 2020. URL `https://bit.ly/39ereMx`.

[3] Intel. High-level synthesis compiler, 2020. URL `https://intel.ly/2UDiWr5`.

[4] Christian Pilato and Fabrizio Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *FPL*, pages 1–4. IEEE, 2013. doi: 10.1109/FPL.2013.6645550.

[5] Ekawat Homsirikamol and Kris Gaj. Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study. In *ReConFig*, pages 1–8. IEEE, 2014. doi: 10.1109/ReConFig.2014.7032504.

[6] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 65–76. Association for Computing Machinery, 2015. doi: 10.1145/2737924.2737986.

[7] Yann Herklotz, Zewei Du, Nadesh Ramanathan, and John Wickerson. An empirical study of the reliability of high-level synthesis tools. In *29th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2021. URL `https://yannherklotz.com/docs/drafts/fuzzing_hls.pdf`. (to appear).

[8] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294. Association for Computing Machinery, 2011. doi: 10.1145/1993498.1993532.

[9] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166. Springer, 1998. doi: 10.1007/BFb0054170.

[10] R. Chouksey and C. Karfa. Verification of scheduling of conditional behaviors in high-level synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–14, 2020. ISSN 1557-9999. doi: 10.1109/TVLSI.2020.2978242.

[11] Mentor. Catapult high-level synthesis, 2020.

[12] Pankaj Chauhan. Formally ensuring equivalence between c++ and rtl designs, 2020. URL `https://bit.ly/2KbT0ki`.

[13] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814.

[14] Intel. SDK for OpenCL applications, 2020. URL `https://intel.ly/30sYHz0`.

[15] R. Chapman, G. Brown, and M. Leeser. Verified high-level synthesis in bedroc. In *[1992] Proceedings The European Conference on Design Automation*, pages 59–63. IEEE Computer Society, March 1992. doi: 10.1109/EDAC.1992.205894.

[16] K. Banerjee, C. Karfa, D. Sarkar, and C. Mandal. Verification of code motion techniques using value propagation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(8):1180–1193, Aug 2014. ISSN 1937-4151. doi: 10.1109/TCAD.2014.2314392.

[17] Andreas Lööw and Magnus O. Myreen. A proof-producing translator for verilog development in hol. In *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering*, FormaliSE '19, pages 99–108. IEEE Press, 2019. doi: 10.1109/FormaliSE.2019.00020.

[18] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. Verified compilation on a verified processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019. ACM, 2019. doi: 10.1145/3314221.3314622.

[19] Andreas Lööw. Lutsig: A verified verilog compiler for verified circuit development. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, page 46–60. Association for Computing Machinery, 2021. doi: 10.1145/3437992.3439916.

[20] Juan Perna and Jim Woodcock. Mechanised wire-wise verification of Handel-C synthesis. *Science of Computer Programming*, 77(4):424 – 443, 2012. ISSN 0167-6423. doi: 10.1016/j.scico.2010.02.007.

[21] Matthew Aubury, Ian Page, Geoff Randall, Jonathan Saul, and Robin Watts. Handel-c language reference guide. *Computing Laboratory. Oxford University, UK*, 1996.

[22] Martin Ellis. *Correct synthesis and integration of compiler-generated function units*. PhD thesis, Newcastle University, 2008. URL `https://theses.ncl.ac.uk/jspui/handle/10443/828`.

[23] IEEE Std 1364. IEEE standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, April 2006. doi: 10.1109/IEEESTD.2006.99495.

[24] IEEE Std 1364.1. IEEE standard for Verilog register transfer level synthesis. *IEC 62142-2005 First edition 2005-06 IEEE Std 1364.1*, pages 1–116, 2005. doi: 10.1109/IEEESTD.2005.339572.

[25] Sandrine Blazy and Xavier Leroy. Formal verification of a memory model for C-like imperative languages. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering*, pages 280–299. Springer Berlin Heidelberg, 2005. doi: 0.1007/11576280_20.

[26] Louis-Noël Pouchet. Polybench/c: the polyhedral benchmark suite, 2020. URL `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`.