Formal Verification of High-Level Synthesis

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

James Pollard
Imperial College London, UK
james.pollard16@imperial.ac.uk

Nadesh Ramanathan
Imperial College London, UK
n.ramanathan14@imperial.ac.uk

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

Abstract

High-level synthesis (HLS), which refers to the automatic compilation of software into hardware, is rapidly gaining popularity. In a world increasingly reliant on application-specific hardware accelerators, HLS promises hardware designs of comparable performance and energy efficiency to those coded by hand in a hardware description language like Verilog, while maintaining the convenience and the rich ecosystem of software development. However, current HLS tools cannot always guarantee that the hardware designs they produce are equivalent to the software they were given, thus undermining any reasoning conducted at the software level. Worse, there is mounting evidence that existing HLS tools are quite unreliable, sometimes generating wrong hardware or crashing when given valid inputs.

To address this problem, we present the first HLS tool that is mechanically verified to preserve the behaviour of its input software. Our tool, called Vericert, extends the CompCert verified C compiler with a new hardware-oriented intermediate language and a Verilog back end, and has been proven correct in Coq. Vericert supports all C constructs except for case statements, function pointers, recursive function calls, integers larger than 32 bits, floats, and global variables. An evaluation on the PolyBench/C benchmark suite indicates that Vericert generates hardware that is around an order of magnitude slower and larger than hardware generated by an existing, optimising (but unverified) HLS tool.

Keywords: CompCert, Coq, high-level synthesis, C, Verilog

1 Introduction

Can you trust your high-level synthesis tool? 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 [7], Vivado HLS [51], Intel i++ [25], and Bambu HLS [44] promise designs with comparable performance and energy-efficiency to those hand-written in HDL [18, 21, 41], 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. [33] 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. And more recently, Du et al. [15] fuzz-tested three commercial HLS tools using Csmith [52], 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* to prove the input and output equivalent [45]. Translation validation has been successfully applied to several HLS optimisations [3, 11, 12, 29, 53]. But translation validation must be repeated

Authors' addresses: Yann Herklotz, Imperial College London, UK, yann.herklotz15@imperial.ac.uk; James Pollard, Imperial College London, UK, james.pollard16@imperial.ac.uk; Nadesh Ramanathan, Imperial College London, UK, n.ramanathan14@imperial.ac.uk; John Wickerson, Imperial College London, UK, j.wickerson@imperial.ac.uk.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

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

every time the compiler is invoked, and it is an expensive task, especially for large designs. For example, the translation validation for Catapult C [36] may require several rounds of expert 'adjustments' [9, 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 [4] 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 [32] with a new hardware-specific intermediate language and a Verilog back end. It supports all C constructs except for case statements, function pointers, recursive function calls, integers larger than 32 bits, floats, and global variables.

Contributions and Outline. The contributions of this paper are as follows:

- We present Vericert, the first mechanically verified HLS tool that compiles C to Verilog. In Section 2, we describe the design of Vericert.
- We state the correctness theorem of Vericert with respect to an existing semantics for Verilog due to Lööw and Myreen [35]. In Section 3, we describe how we lightly extended this semantics to make it suitable as an HLS target.
- In Section 4, we describe how we proved this 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, and correctly mapping CompCert's memory model onto a finite Verilog array.
- In Section 5, we evaluate Vericert on the Polybench/C benchmark suite [46], and compare the performance of our generated hardware against an existing, unverified HLS tool called LegUp [7]. We show that Vericert generates hardware that is 56× slower (10× slower in the absence of division) and 21× 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.

Vericert is fully open source and available online.

https://github.com/ymherklotz/vericert

2 Designing a verified HLS tool

This section describes the main architecture of the HLS tool, and the way in which the Verilog back end was added to CompCert. This section will also cover an example of converting a simple C program into hardware, expressed in the Verilog language.

Choice of source language. C was chosen as the source language as it remains the most common source language amongst production-quality HLS tools [7, 25, 44, 51]. This, in turn, may be because it is "[t]he starting point for the vast majority of algorithms to be implemented in hardware" [17], lending a degree of practicality. We considered Bluespec [39], but decided that although it "can be classed as a high-level language" [19], it is too hardware-oriented to be suitable for traditional HLS. We also considered using a language with built-in parallel constructs that map well to parallel hardware, such as occam [40], Spatial [30] or Scala [2], but found these languages too niche.

Choice of target language. Verilog [23] is an HDL that can be synthesised into logic cells which can either be placed onto a field-programmable gate array (FPGA) or turned into an application-specific integrated circuit (ASIC). Verilog was chosen as the output language for Vericert because it is one of the most popular HDLs and there already exist a few formal semantics for it that could be used as a target [34, 37]. Bluespec, previously ruled out as a source language, is another possible target and there exists a formally verified translation to circuits using Kôika [6].

Choice of implementation language. We chose Coq as the implementation language because of its mature support for code extraction; that is, its ability to generate OCaml programs directly from the definitions used in the theorems. We note that other authors have had some success reasoning about the HLS process using other theorem provers such as Isabelle [16]. CompCert [32] was chosen as the front end framework, as it is a mature framework for simulation proofs about intermediate languages, and it already provides a validated C parser [28]. The Vellvm framework [55] was also considered because several existing HLS tools are already LLVM-based, but additional work would be required to support a high-level language like C as input. The .NET framework has been used as a basis for other HLS tools, such as Kiwi [20], and LLHD [48] has been recently proposed as an intermediate language for hardware design, but neither are suitable for us because they lack formal semantics.

Architecture of Vericert. The main work flow of Vericert is given in Figure 1, which shows those parts of the

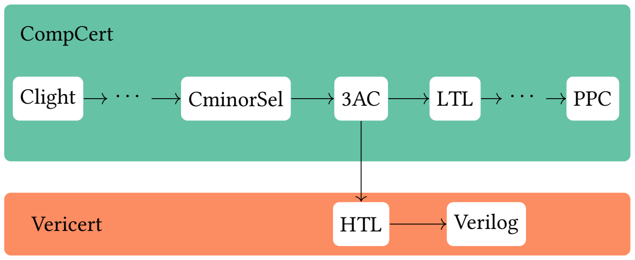

Figure 1. Verilog back end to CompCert, branching off at the three address code (3AC), at which point the three address code is transformed into a state machine. Finally, it is transformed to a hardware description of the state machine in Verilog.

translation that are performed in CompCert, and those that have been added.

CompCert translates Clight[3] input into assembly output via a sequence of intermediate languages; we must decide which of these ten languages is the most suitable starting point for the HLS-specific translation stages.

We select CompCert's three-address code (3AC)[4] as the starting point. Branching off before this point (at CminorSel or earlier) denies CompCert the opportunity to perform optimisations such as constant propagation and dead code elimination, which have been found to be useful in HLS tools as well as software compilers [14]. Instead, if we branch off after this point (at LTL or later) then CompCert has already performed register allocation to reduce the number of registers and spill some variables to the stack; this transformation is not required in HLS because there are many more registers available, and these should be used instead of RAM whenever possible.

3AC is also attractive because it is the closest intermediate language to LLVM IR, which is used by several existing HLS compilers. It has an unlimited number of pseudo-registers, and is represented as a control flow graph (CFG) where each instruction is a node with links to the instructions that can follow it. One difference between LLVM IR and 3AC is that 3AC includes operations that are specific to the chosen target architecture; we chose to target the x86_32 backend, because it generally produces relatively dense 3AC thanks to the availability of complex addressing modes.

2.1 Translating C to Verilog, by example

Figure 2 illustrates the translation of a simple program that sums the elements of an array. In this section, we describe the stages of the Vericert translation, referring to this program as an example.

[3]A deterministic subset of C with pure expressions.

[4]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.

```c
int main() {
  int x[3] = {1, 2, 3};
  int sum = 0;
  for (int i = 0;
       i < 3;
       i++)
    sum += x[i];
  return sum;
}
```

(a) Input C code.

```
main() {
 15:  x8 = 1
 14:  int32[stack(0)] = x8
 13:  x7 = 2
 12:  int32[stack(4)] = x7
 11:  x6 = 3
 10:  int32[stack(8)] = x6
  9:  x2 = 0
  8:  x1 = 0
  7:  x5 = stack(0) (int)
  6:  x4 = int32[x5 + x1 * 4 + 0]
  5:  x2 = x2 + x4 + 0 (int)
  4:  x1 = x1 + 1 (int)
  3:  if (x1 <s 3) goto 7 else goto 2
  2:  x3 = x2
  1:  return x3
}
```

(b) 3AC produced by CompCert.

```verilog
module main(reset, clk, finish, return_val);
  reg [31:0] stack [2:0];
  input [0:0] clk, reset;
  output reg [31:0] return_val;
  output reg [0:0] finish;
  reg [31:0] reg_8, reg_4, state,
             reg_6, reg_1, reg_7,
             reg_5, reg_3;
  always @(posedge clk)
    case (state)
      32'd15: reg_8 <= 32'd1;
      32'd14: stack[32'd0] <= reg_8;
      32'd13: reg_7 <= 32'd2;
      32'd12: stack[32'd1] <= reg_7;
      32'd11: reg_6 <= 32'd3;
      32'd10: stack[32'd2] <= reg_7;
      32'd9: reg_2 <= 32'd0;
      32'd8: reg_1 <= 32'd0;
      32'd7: reg_5 <= 32'd0;
      32'd6: reg_4 <= stack[{{{reg_5 + 32'd0}
        + {reg_1 * 32'd4}} / 32'd4}];
      32'd5: reg_2 <= {reg_2 + {reg_4 + 32'd0}};
      32'd4: reg_1 <= {reg_1 + 32'd1};
      32'd3: ;
      32'd2: reg_3 <= reg_2;
      32'd1: begin
        finish = 1'd1;
        return_val = reg_3;
      end
      default:;
    endcase
```

```verilog
  always @(posedge clk)
    if ({reset == 1'd1})
      state <= 32'd16;
    else
      case (state)
        32'd15: state <= 32'd14;
        32'd14: state <= 32'd13;
        32'd13: state <= 32'd12;
        32'd12: state <= 32'd11;
        32'd11: state <= 32'd10;
        32'd10: state <= 32'd9;
        32'd9: state <= 32'd8;
        32'd8: state <= 32'd7;
        32'd7: state <= 32'd6;
        32'd6: state <= 32'd5;
        32'd5: state <= 32'd4;
        32'd4: state <= 32'd3;
        32'd3: state <=
          ({$signed(reg_1)
            < $signed(32'd3)}
             ? 32'd7 : 32'd2);
        32'd2: state <= 32'd1;
        32'd1: ;
        default:;
      endcase
endmodule
```

(c) Verilog produced by Vericert. The left column contains the datapath and the right column contains the control logic.

Figure 2. Translating a simple program from C to Verilog.

2.1.1 Translating C to 3AC.

The first stage of the translation uses unmodified CompCert to transform the C input, shown in Figure 2a, into a 3AC intermediate representation, shown in Figure 2b. As part of this translation, function inlining is also performed on all functions, which allows us to support function calls without having to support the `Icall` 3AC instruction. Although the duplication of the function bodies caused by inlining can increase the area of the hardware, it can have a positive effect on latency. Inlining precludes support for recursive function calls, but this feature isn't supported in most other HLS tools either [50].

2.1.2 Translating 3AC to HTL.

The next translation is from 3AC to a new hardware translation language (HTL). This involves going from a CFG representation of the computation to a finite state machine with data-path (FSMD) representation [22]. The core idea of the FSMD representation is that it separates the control flow from the operations on the memory and registers. Hence, an HTL program consists of two maps from states to Verilog statements: control- and data-path maps that express state transitions and computations respectively. Figure 3 shows the resulting FSMD architecture. The right-hand block is the control logic that computes the next state, while the left-hand block updates all the registers and RAM based on the current program state.

Translating memory. Typically, HLS-generated hardware consists of a sea of registers and RAM memories. This memory view is very different to the C memory model, so we perform the following translation. Variables that do not have their address taken are kept in registers, which correspond to the registers in 3AC. All address-taken variables, arrays, and structs are kept in RAM. The stack of the main function becomes an unpacked array of 32-bit integers, which may be translated to a RAM when the hardware description is passed through a synthesis tool. Finally, global variables are not translated in Vericert at the moment. A high-level overview of the architecture can be seen in Figure 3.

Translating instructions. Each 3AC instruction either corresponds to a hardware construct, or does not have to be handled by the translation, such as function calls (because of inlining). For example, state 15 in Figure 2b shows a 32-bit register x8 being initialised to 1, after which the control flow moves to state 14. This initialisation is also encoded in HTL at state 15 in both the control- and data-path always-blocks, as shown in Figure 2c. Simple operator instructions are translated in a similar way. For example, in state 5, the value of the array element is added to the current sum value, which is simply translated to an addition of the equivalent registers in the HTL code.

Note that the comparison in state 3 is signed. C and Verilog handle signedness quite differently; by default, all operators and registers in Verilog (and HTL) are unsigned, so to force an operation to handle the bits as signed, both operators have to be forced to be signed. In addition to that, Verilog implicitly resizes expressions to the largest needed size by default, which can affect the result of the computation. This feature is not supported by the Verilog semantics we adopted, so to match the semantics to the behaviour of the simulator and synthesis tool, braces are placed around all expressions as this inhibits implicit resizing. Instead, explicit resizing is used in the semantics and operations can only be performed on two registers that have the same size.

In addition to that, equality between *unsigned* variables are actually not supported, because this requires supporting the comparison of pointers, which should only be performed between pointers with the same provenance. In Vericert there is currently no way to determine the provenance of a pointer, and it therefore cannot model the semantics of unsigned comparison in CompCert. As dynamic allocation is not supported either, comparison of pointers is rarely needed, and for the comparison of integers, these can be cast to signed integers during the comparison for the translation to succeed.

2.1.3 Translating HTL to Verilog.

Finally, we have to translate the HTL code into proper Verilog. The challenge here is to translate our FSMD representation into a Verilog AST. However, as all the instructions in HTL are already expressed as Verilog statements, only the top level data-path and control logic maps need to be translated to valid Verilog. We also require declarations for all the variables in the program, as well as declarations of the inputs and outputs to the module, so that the module can be used inside a larger hardware design. Figure 2c shows the final Verilog output that is generated for our example.

Although this translation seems quite straightforward, proving that this translation is correct is complex. All the implicit assumptions that were made in HTL need to be translated explicitly to Verilog statements and it needs to be shown that these explicit behaviours are equivalent to the assumptions made in the HTL semantics. We discuss these proofs in upcoming sections.

2.2 Optimisations

Although we would not claim that Vericert is a proper 'optimising' HLS compiler yet, we have nonetheless made several design choices that aim to improve the quality of the hardware designs it produces.

2.2.1 Byte- and word-addressable memories.

One big difference between C and Verilog is how memory is represented. Although Verilog arrays might seem to mirror their C counterparts directly, they must be treated quite differently. To reduce the design area and avoid timing issues, it is beneficial if Verilog arrays can be synthesised as RAMs, but this imposes various constraints on how Verilog arrays are used; for instance, RAMs often only allow one read and one write operation per clock cycle. To make loads and stores as efficient as possible, the RAM needs to be word-addressable, which means that an entire integer can be loaded or stored in one clock cycle. However, the memory model that CompCert uses for its intermediate languages is byte-addressable [5]. It therefore has to be proven that the byte-addressable memory behaves in the same way as the word-addressable memory in hardware. Any modifications of the bytes in the CompCert memory model also have to be shown to modify the word-addressable memory in the same way. Since

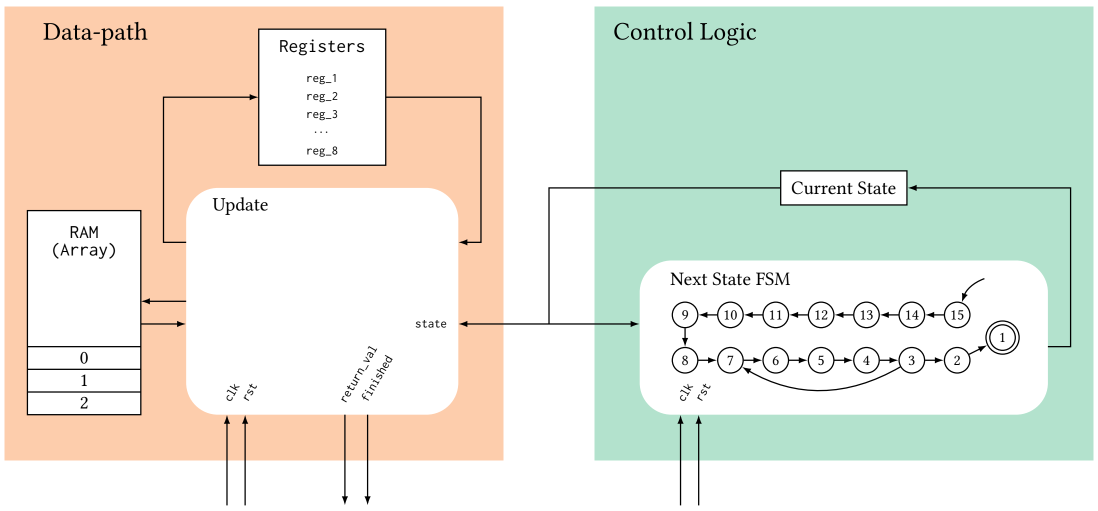

Figure 3. The FSMD for the example shown in Figure 2, split into a data-path and control logic for the next state calculation. The Update block takes the current state, current values of all registers and at most one value stored in the array, and calculates a new value that can either be stored back in the array or in a register.

only integer loads and stores are currently supported in Vericert, it follows that the addresses given to the loads and stores should be multiples of four. If that is the case, then the translation from byte-addressed memory to word-addressed memory can be done by dividing the address by four.

2.2.2 Implementing the `Oshrximm` instruction.

Many of the CompCert instructions map well to hardware, but `Oshrximm` is expensive if implemented naïvely. The problem is that in CompCert it is specified as a signed division:

$$\texttt{Oshrximm}\ x\ y = \text{round\_towards\_zero}\left(\frac{x}{2^y}\right)$$

(where $x, y \in \mathbb{Z}$, $0 \leq y < 31$, and $-2^{31} \leq x < 2^{31}$) and instantiating divider circuits in hardware is well-known to cripple performance. Moreover, since Vericert requires the result of a divide operation to be ready within a single clock cycle, the divide circuit needs to be entirely combinational. This is inefficient in terms of area, but also in terms of latency, because it means that the maximum frequency of the hardware must be reduced dramatically so that the divide circuit has enough time to finish.

One might hope that the synthesis tool consuming our generated Verilog would convert the division to an efficient shift operation, but this is unlikely to happen with signed division which requires more than a single shift. However, the observation can be made that signed division can be implemented using shifts:

$$\text{round\_towards\_zero}\left(\frac{x}{2^y}\right) = \begin{cases} x \gg y & \text{if } x \geq 0 \\ -(-x \gg y) & \text{otherwise.} \end{cases}$$

where $\gg$ stands for a logical right shift. When proving this equivalence, we actually found a bug in our original implementation that was due to the fact that a naïve shift rounds towards $-\infty$.

3 A Formal Semantics for Verilog

This section describes the Verilog semantics that was chosen for the target language, including the changes that were made to the semantics to make it a suitable HLS target. 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.

The Verilog semantics we use is ported to Coq from a semantics written in HOL4 by Lööw and Myreen [35] and used to prove the translation from HOL4 to Verilog [34]. 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; these are modelled in other semantics such as that by Meredith et al. [37].

The semantics of Verilog differs from regular programming languages, as it is used to describe hardware directly, which is inherently parallel, rather than an algorithm, which is usually sequential. The main construct in Verilog is the always-block. A module can contain multiple always-blocks, all of which run in parallel. These always-blocks further contain statements such as if-statements or assignments to variables. We support only *synchronous* logic, which means that the always-block is triggered on (and only on) the rising edge of a clock signal.

The semantics combines the big-step and small-step styles. The overall execution of the hardware is described using a small-step semantics, with one small step per clock cycle; this is appropriate because hardware is routinely designed to run for an unlimited number of clock cycles and the big-step style is ill-suited to describing infinite executions. Then, within each clock cycle, a big-step semantics is used to execute all the statements. An example of a rule for executing an always-block is shown below, where Σ is the state of the registers in the module and s is the statement inside the always-block:

$$\textsc{Always}\quad \frac{(\Sigma, s) \downarrow_{\text{stmnt}} \Sigma'}{(\Sigma, \texttt{always @(posedge clk)}\ s) \downarrow_{\text{always}} \Sigma'}$$

This rule says that assuming the statement s in the always-block runs with state Σ and produces the new state Σ', the always-block will result in the same final state.

Two types of assignments are supported in always-blocks: nonblocking and blocking assignment. Nonblocking assignments all take effect simultaneously at the end of the clock cycle, while blocking assignments happen instantly so that later assignments in the clock cycle can pick them up. To model both of these assignments, the state Σ has to be split into two maps: Γ, which contains the current values of all variables, and Δ, which contains the values that will be assigned at the end of the clock cycle. Nonblocking assignment can therefore be expressed as follows:

$$\textsc{Nonblocking Reg}\quad \frac{\texttt{name}\ d = \texttt{OK}\ n \qquad (\Gamma, e) \downarrow_{\text{expr}} v}{((\Gamma, \Delta), d\ \texttt{<=}\ e) \downarrow_{\text{stmnt}} (\Gamma, \Delta[n \mapsto v])}$$

where assuming that $\downarrow_{\text{expr}}$ evaluates an expression e to a value v, the nonblocking assignment d `<=` e updates the future state of the variable d with value v.

Finally, the following rule dictates how the whole module runs in one clock cycle:

$$\textsc{Module}\quad \frac{(\Gamma, \epsilon, \vec{m})\ \downarrow_{\text{module}} (\Gamma', \Delta')}{(\Gamma, \texttt{module main(...);}\ \vec{m}\ \texttt{endmodule}) \downarrow_{\text{program}} (\Gamma' /\!/ \Delta')}$$

where Γ is the initial state of all the variables, and $\vec{m}$ is a list of variable declarations and always-blocks that $\downarrow_{\text{module}}$ evaluates sequentially to obtain (Γ', Δ'). The final state is obtained by merging these maps using the $/\!/$ operator, which gives priority to the right-hand operand in a conflict. This rule ensures that the nonblocking assignments overwrite at the end of the clock cycle any blocking assignments made during the cycle.

3.1 Changes to the Semantics

Four changes were made to the semantics proposed by Lööw and Myreen [35] to make it suitable as a 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. Consider the following Verilog code:

```
reg [31:0] x[1:0];
always @(posedge clk) begin
  x[0] = 1;
  x[1] <= 1;
end
```

which modifies one array element using blocking assignment and then a second using nonblocking assignment. If the existing semantics were used to update the array, then during the merge, the entire array `x` from the nonblocking association map would replace the entire array from the blocking association map. This would replace `x[0]` with its original value and therefore behave incorrectly. Accordingly, we modified the maps so they record updates on a per-element basis. Our state Γ is therefore split up into Γ_r for instantaneous updates to variables, and Γ_a for instantaneous updates to arrays; Δ is split similarly. The merge function then ensures that only the modified indices get updated when Γ_a is merged with the nonblocking map equivalent Δ_a.

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 shouldn't 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 of arrays of Booleans. This is because Vericert (currently) only supports types represented by 32 bits.

3.2 Integrating the Verilog Semantics into CompCert's Model

The CompCert computation model defines a set of states through which execution passes. In this subsection, we explain how we extend our Verilog semantics with five special-purpose registers in order to integrate it into CompCert.

CompCert executions pass through three main states:

State $\mathit{sf}\ m\ v\ \Gamma_r\ \Gamma_a$ The main state when executing a function, with stack frame sf, current module m, current state v and variable states Γ_r and Γ_a.

$$\textsc{Step}\quad \frac{\Gamma_r\ !\ \mathit{rst} = \texttt{Some}\ \ 0 \qquad \Gamma_r\ !\ \mathit{fin} = \texttt{Some}\ \ 0 \qquad \Gamma_r\ !\ \sigma = \texttt{Some}\ \ v \qquad (m, (\Gamma_r, \Gamma_a))\ \downarrow_{\text{program}}\ (\Gamma_r', \Gamma_a') \qquad \Gamma_r'\ !\ \sigma = \texttt{Some}\ \ v'}{\texttt{State}\ \ \mathit{sf}\ \ m\ v\ \Gamma_r\ \Gamma_a \longrightarrow \texttt{State}\ \ \mathit{sf}\ \ m\ v'\ \Gamma_r'\ \Gamma_a'}$$

$$\textsc{Finish}\quad \frac{\Gamma_r\ !\ \mathit{fin} = \texttt{Some}\ \ 1 \qquad \Gamma_r\ !\ \mathit{ret} = \texttt{Some}\ \ r}{\texttt{State}\ \ \mathit{sf}\ \ m\ \sigma\ \Gamma_r\ \Gamma_a \longrightarrow \texttt{Returnstate}\ \ \mathit{sf}\ \ r}$$

$$\textsc{Call}\quad \frac{}{\texttt{Callstate}\ \ \mathit{sf}\ \ m\ \vec{r} \longrightarrow \texttt{State}\ \ \mathit{sf}\ \ m\ n\ ((\texttt{init\_params}\ \vec{r}\ a)[\sigma \mapsto n, \mathit{fin} \mapsto 0, \mathit{rst} \mapsto 0])\ \epsilon}$$

$$\textsc{Return}\quad \frac{}{\texttt{Returnstate}\ \ (\texttt{Stackframe}\ \ r\ m\ \mathit{pc}\ \ \Gamma_r\ \Gamma_a :: \mathit{sf})\ v \longrightarrow \texttt{State}\ \ \mathit{sf}\ \ m\ \mathit{pc}\ \ (\Gamma_r[\mathit{st} \mapsto \mathit{pc}, r \mapsto v])\ \Gamma_a}$$

Figure 4. Top-level small-step semantics for Verilog modules in CompCert's computational framework.

`Callstate` *sf* $m\ \vec{r}$ The state that is reached when a function is called, with the current stack frame *sf*, current module m and arguments $\vec{r}$.

`Returnstate` *sf* v The state that is reached when a function returns back to the caller, with stack frame *sf* and return value v.

To support this computational model, we extend the Verilog module we generate with the following five registers and a RAM block:

program counter The program counter can be modelled using a register that keeps track of the state, denoted as σ.

function entry point When a function is called, the entry point denotes the first instruction that will be executed. This can be modelled using a reset signal that sets the state accordingly, denoted as *rst*.

return value The return value can be modelled by setting a finished flag to 1 when the result is ready, and putting the result into a 32-bit output register. These are denoted as *fin* and *rtrn* respectively.

stack The function stack can be modelled as a RAM block, which is implemented using an array in the module, and denoted as *stk*.

Figure 4 shows the inference rules for moving between the computational states. The first, Step, is the normal rule of execution. It defines one step in the `State` state, assuming that the module is not being reset, that the finish state has not been reached yet, that the current and next state are v and v', and that the module runs from state Γ to Γ' using the Step rule. The Finish rule returns the final value of running the module and is applied when the *fin* register is set; the return value is then taken from the *ret* register.

Note that there is no step from `State` to `Callstate`; this is because function calls are not supported, and it is therefore impossible in our semantics to ever reach a `Callstate` except for the initial call to main. So the Call rule is only used at the very beginning of execution; likewise, the Return rule is only matched for the final return value from the main function. Therefore, in addition to the rules shown in Figure 4, an initial state and final state need to be defined:

$$\textsc{Initial}\quad \frac{\texttt{is\_internal}\ (P.\texttt{main})}{\texttt{initial\_state}\ (\texttt{Callstate}\ \ []\ (P.\texttt{main})\ [])}$$

$$\textsc{Final}\quad \frac{}{\texttt{final\_state}\ (\texttt{Returnstate}\ \ []\ n)\ n}$$

where the initial state is the `Callstate` with an empty stack frame and no arguments for the `main` function of program P, where this `main` function needs to be in the current translation unit. The final state results in the program output of value n when reaching a `Returnstate` with an empty stack frame.

4 Proof

Now that the Verilog semantics have been adapted to the CompCert model, we are in a position to formally prove the correctness of our C-to-Verilog compilation. This section describes the main correctness theorem that was proven and the main ideas behind the proof. The full Coq proof is available in auxiliary material.

The main correctness theorem is analogous to that stated in CompCert [32]: 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.*

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

The theorem is a 'backwards simulation' result (from target to source). 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.

The second observation that needs to be made is that to prove this 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. The forward simulation from 3AC to HTL is stated in Lemma 1 (Section 4.1), then the forward simulation between HTL and Verilog is shown in Lemma 4 (Section 4.2) and finally, the proof that Verilog is deterministic is given in Lemma 5 (Section 4.3).

4.1 Forward simulation from 3AC to HTL

As HTL is quite far removed from 3AC, this first translation is the most involved and therefore requires a larger proof, because the translation from 3AC instructions to Verilog statements needs to be proven correct in this step. In addition to that, the semantics of HTL are also quite different to the 3AC semantics, as instead of defining small-step semantics for each construct in Verilog, the semantics are instead defined over one clock cycle and mirror the semantics defined for Verilog. Lemma 1 shows the result that needs to be proven in this subsection.

Lemma 1 (Forward simulation from 3AC to HTL). *We write* `tr_htl` *for the translation from 3AC to HTL.*

$$\forall c, h, B \in \texttt{Safe}, \texttt{tr\_htl}(c) = \texttt{OK}(h) \land c \Downarrow B \implies h \Downarrow B.$$

We prove this lemma by first establishing a specification of the translation function `tr_htl` that captures its important properties, and then splitting the proof into two parts: one to show that the translation function does indeed meet its specification, and one to show that the specification implies the desired simulation result. This strategy is in keeping with standard CompCert practice.

4.1.1 From Implementation to Specification.

The specification for the translation of 3AC instructions into HTL data-path and control logic can be defined by the following predicate:

$$\texttt{spec\_instr}\ \mathit{fin}\ \mathit{rtrn}\ \sigma\ \mathit{stk}\ i\ \mathit{data}\ \ \mathit{control}$$

Here, the *control* and *data* parameters are the statements that the current 3AC instruction i should translate to. The other parameters are the special registers defined in Section 3.2. An example of a rule describing the translation of an arithmetic/logical operation from 3AC is the following:

Iop

$$\frac{\texttt{tr\_op}\ \mathit{op}\ \vec{a} = \texttt{OK}\ e}{\texttt{spec\_instr}\ \mathit{fin}\ \mathit{rtrn}\ \sigma\ \mathit{stk}\ (\texttt{Iop}\ \ \mathit{op}\ \vec{a}\ d\ n)\ (d \texttt{ <= } e)\ (\sigma \texttt{ <= } n)}$$

Assuming that the translation of the operator op with operands $\vec{a}$ is successful and results in expression e, the rule describes how the destination register d is updated to e via a non-blocking assignment in the data path, and how the program counter σ is updated to point to the next CFG node n via another non-blocking assignment in the control path.

In the following lemma, `spec_htl` is the top-level specification predicate, which is built using `spec_instr` at the level of instructions.

Lemma 2. *If a 3AC program c is translated correctly to an HTL program h, then the specification of the translation holds.*

$$\forall\ c\ h,\ \texttt{tr\_htl}(c) = \texttt{OK}(h) \implies \texttt{spec\_htl}\ c\ h.$$

4.1.2 From Specification to Simulation.

To prove that the specification predicate implies the desired forward simulation, we must first define a relation that matches each 3AC state to an equivalent HTL state. This relation also captures the assumptions made about the 3AC code that we receive from CompCert.These assumptions then have to be proven to always hold assuming the HTL code was created by the translation algorithm. Some of the assumptions that need to be made about the 3AC and HTL code for a pair of states to match are:

- The 3AC register file R needs to be 'less defined' than the HTL register map Γ_r (written $R \le \Gamma_r$). This means that all entries should be equal to each other, unless a value in R is undefined, in which case any value can match it.
- The RAM values represented by each Verilog array in Γ_a need to match the 3AC function's stack contents, which are part of the memory M; that is: $M \le \Gamma_a$.
- The state is well formed, which means that the value of the state register matches the current value of the program counter; that is: $pc = \Gamma_r!\sigma$.

We also define the following set $\mathcal{I}$ of invariants that must hold for the current state to be valid:

- that all pointers in the program use the stack as a base pointer,

- that any loads or stores to locations outside of the bounds of the stack result in undefined behaviour (and hence we do not need to handle them),
- that *rst* and *fin* are not modified and therefore stay at a constant 0 throughout execution, and
- that the stack frames match.

We can now define the simulation diagram for the translation. The 3AC state can be represented by the tuple (R, M, pc), which captures the register file, memory, and program counter. The HTL state can be represented by the pair (Γ_r, Γ_a), which captures the states of all the registers and arrays in the module. Finally, $\mathcal{I}$ stands for the other invariants that need to hold for the states to match.

Lemma 3. *Given the 3AC state (R, M, pc) and the matching HTL state (Γ_r, Γ_a), assuming one step in the 3AC semantics produces state (R', M', pc'), there exist one or more steps in the HTL semantics that result in matching states (Γ_r', Γ_a'). This is all under the assumption that the specification* `tr_htl` *holds for the translation.*

$$\begin{tikzcd}
R, M, pc \arrow[r, no head, "\mathcal{I} \land (R \le \Gamma_r) \land (M \le \Gamma_a) \land (pc = \Gamma_r!\sigma)"] \arrow[d] & \Gamma_r, \Gamma_a \arrow[d, dashed, "+"] \\
R', M', pc' \arrow[r, dashed, no head, "\mathcal{I} \land (R' \le \Gamma_r') \land (M' \le \Gamma_a') \land (pc' = \Gamma_r'!\sigma)"] & \Gamma_r', \Gamma_a'
\end{tikzcd}$$

Proof sketch. This simulation diagram is proven by induction over the operational semantics of 3AC, which allows us to find one or more steps in the HTL semantics that will produce the same final matching state. □

4.2 Forward simulation from HTL to Verilog

The HTL-to-Verilog simulation is conceptually simple, as the only transformation is from the map representation of the code to the case-statement representation. The proof is more involved, as the semantics of a map structure are quite different to the semantics of the case-statement they are converted to.

Lemma 4 (Forward simulation from HTL to Verilog)**.** *We write* `tr_verilog` *for the translation from HTL to Verilog. (Note that this translation cannot fail, so we do not need the* `OK` *constructor here.)*

$$\forall h, V, B \in \texttt{Safe}, \texttt{tr\_verilog}(h) = V \land h \Downarrow B \implies V \Downarrow B.$$

Proof sketch. The translation from maps to case-statements is done by turning each node of the tree into a case-expression with the statements in each being the same. The main difficulty for the proof is that a random-access structure is transformed into an inductive structure where a certain number of constructors need to be called to get to the correct case. □

One problem with our representation of the state as an actual register is that we have to make sure that the state does not overflow. Currently, the state register always has 32 bits, meaning the maximum number of states supported is 2^{32}. Vericert will error out if there are more than this many nodes in the 3AC, thus satisfying the correctness theorem vacuously.

4.3 Deterministic Semantics

The final lemma we need is that the Verilog we generate is deterministic. This result allows us to replace the forwards simulation we have proved with the backwards simulation we desire.

Lemma 5. *If a Verilog program V admits both behaviours B_1 and B_2, then B_1 and B_2 must be the same.*

$$\forall V, B_1, B_2, V \Downarrow B_1 \land V \Downarrow B_2 \implies B_1 = B_2.$$

Proof sketch. The Verilog semantics is deterministic because the order of operation of all the constructs is defined, and there is therefore only one way to evaluate the module and hence only one possible behaviour. This was proven for the small-step semantics shown in Figure 4. □

4.4 Coq Mechanisation

The lines of code for the implementation and proof of Vericert can be found in Table 1. Overall, it took about 1 person-year to build Vericert – about two person-months on implementation and ten person-months on proofs. The largest proof is the correctness proof for the HTL generation, which required equivalence proofs between all integer operations supported by CompCert and those supported in hardware. From the 3349 lines of proof code in the HTL generation, 1189 are for the correctness proof of just the load and store instructions. These were tedious to prove correct because of the substantial difference between the memory models used, and the need to prove properties such as stores outside of the allocated memory being undefined, so that a finite array could be used. In addition to that, since pointers in HTL and Verilog are represented as integers, whereas there is a separate pointer value in the CompCert semantics, it was painful to reason about them and many new theorems had to be proven about integers and pointers in Vericert.

5 Evaluation

Our evaluation is designed to answer the following three research questions.

RQ1 How fast is the hardware generated by Vericert?
RQ2 How area-efficient is the hardware generated by Vericert?
RQ3 How long does Vericert take to produce hardware?

	Coq code	OCaml code	Specifications	Theorems & Proofs	Total
Data structures and libraries	274	—	—	654	928
Integers and values	98	—	15	744	857
HTL semantics	—	—	174	—	174
HTL generation	655	—	79	3349	4083
Verilog semantics	—	—	739	174	913
Verilog generation	68	—	—	396	464
Top-level driver, pretty printers	89	747	—	209	1045
Total	1184	747	1007	5526	8464

Table 1. Statistics about the numbers of lines of code in the proof and implementation of Vericert.

5.1 Experimental Setup

Choice of HLS tool for comparison. We compare Vericert against LegUp 5.1 because it is open-source and hence easily accessible, but still produces hardware "of comparable quality to a commercial high-level synthesis tool" [7].

Choice and preparation of benchmarks. We evaluate Vericert using the PolyBench/C benchmark suite (version 4.2.1) [46], which consists of a collection of 30 numerical kernels. PolyBench/C is popular in the HLS context [10, 47, 54, 56], since it has affine loop bounds, making it attractive for streaming computation on FPGA architectures. We were able to use 27 of the 30 programs; three had to be discarded (`correlation`, `gramschmidt` and `deriche`) because they involve square roots, requiring floats, which we do not support. We configured Polybench's parameters so that only integer types are used, since we do not support floats. We use Polybench's smallest datasets for each program to ensure that data can reside within on-chip memories of the FPGA, avoiding any need for off-chip memory accesses.

Vericert implements divisions and modulo operations in C using the corresponding built-in Verilog operators. These built-in operators are designed to complete within a single clock cycle, and this causes substantial penalties in clock frequency. Other HLS tools, including LegUp, supply their own multi-cycle division/modulo implementations, and we plan to do the same in future versions of Vericert. In the meantime, we have prepared an alternative version of the benchmarks in which each division/modulo operation is overridden with our own implementation that uses repeated division and multiplications by 2. Where this change makes an appreciable difference to the performance results, we give the results for both benchmark sets.

Synthesis setup. The Verilog that is generated by Vericert or LegUp is provided to Intel Quartus v16.0 [26], which synthesises it to a netlist, before placing-and-routing this netlist onto an Intel Arria 10 FPGA device that contains approximately 430000 LUTs.

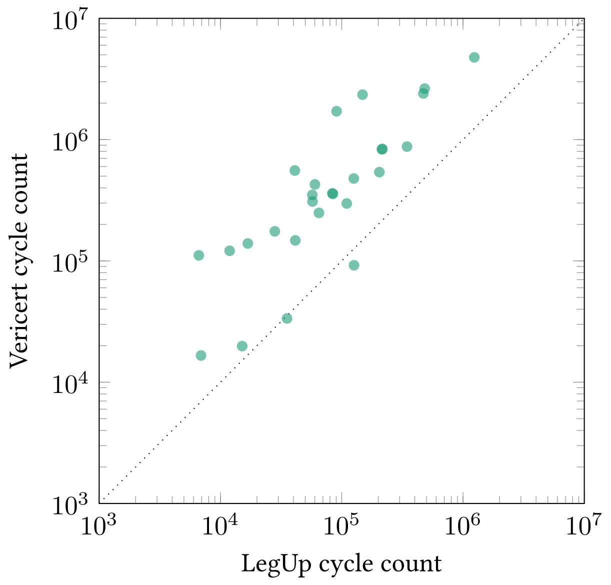

Figure 5. A comparison of the cycle count of hardware designs generated by Vericert and by LegUp.

5.2 RQ1: How fast is Vericert-generated hardware?

Figure 5 compares the cycle counts of our 27 programs executed by Vericert and LegUp respectively. In most cases, we see that the data points are above the diagonal, which demonstrates that the LegUp-generated hardware is faster than Vericert-generated Verilog.

On average, LegUp designs are 4.5× faster than Vericert designs. This performance gap is mostly due to LegUp optimisations such as scheduling and memory analysis, which are designed to extract parallelism from input programs. It is notable that even without Vericert performing many optimisations, a few data points are close to the diagonal and even below it. As we improve Vericert by incorporating further optimisations, this gap should reduce whilst preserving our correctness guarantees.

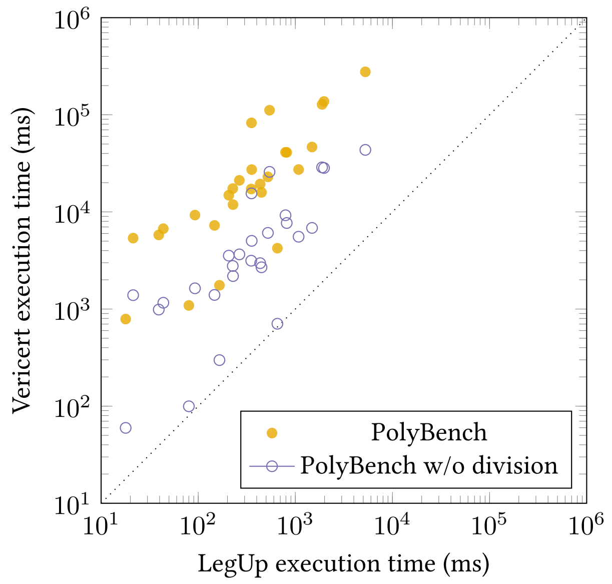

Figure 6. A comparison of the execution time of hardware designs generated by Vericert and by LegUp.

Cycle count is one factor in calculating execution times; the other is the clock frequency, which determines the duration of each of these cycles. Figure 6 compares the execution times of Vericert and LegUp. Across the original Polybench/C benchmarks, we see that Vericert designs are about 56× slower than LegUp designs. This dramatic discrepancy in performance can be largely attributed to Vericert's naïve implementations of division and modulo operations, as explained in Section 5.1. Indeed, Vericert achieved an average clock frequency of just 21MHz, while LegUp managed about 247MHz. After replacing the division/modulo operations with our own C-based implementations, Vericert's average clock frequency becomes about 112MHz. This is better, but still substantially below LegUp, which uses various additional optimisations and Intel-specific IP blocks. Across the modified Polybench/C benchmarks, we see that Vericert designs are about 10× slower than LegUp designs.

5.3 RQ2: How area-efficient is Vericert-generated hardware?

Figure 7 compares the resource utilisation of the Polybench programs generated by Vericert and LegUp. On average, we see that Vericert produces hardware that is about 21× larger than LegUp. Vericert designs are filling up to 30% of a (large) FPGA chip, while LegUp uses no more than 1% of the chip. The main reason for this is that RAM is not inferred automatically for the Verilog that is generated by Vericert; instead, large arrays of registers are synthesised. Synthesis tools such as Quartus generally require array accesses to be in a specific form in order for RAM inference to activate.

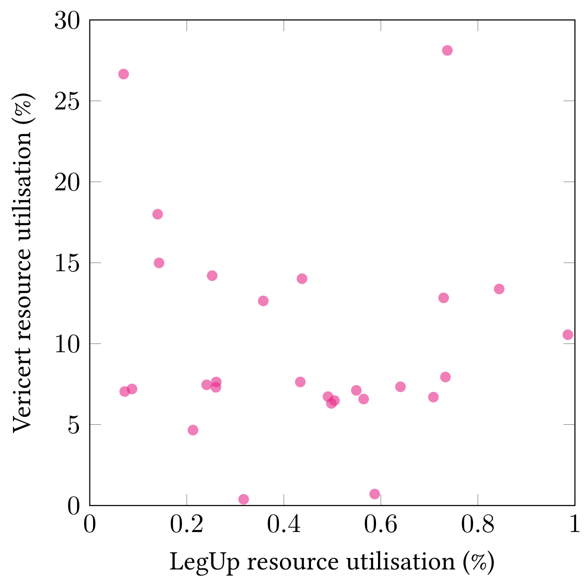

Figure 7. A comparison of the resource utilisation of designs generated by Vericert and by LegUp.

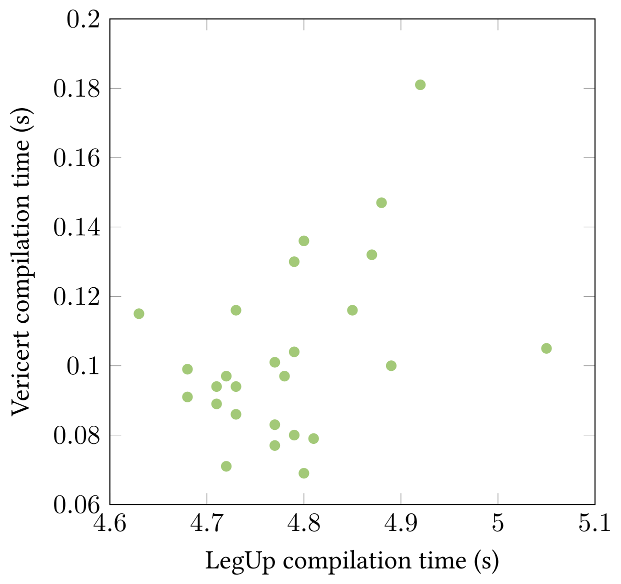

Figure 8. A comparison of compilation time for Vericert and for LegUp

LegUp's Verilog generation is tailored to enable RAM inference by Quartus, while Vericert generates more generic array accesses. This may make Vericert more portable across different FPGA synthesis tools and vendors. Enabling RAM inference is part of our future plans.

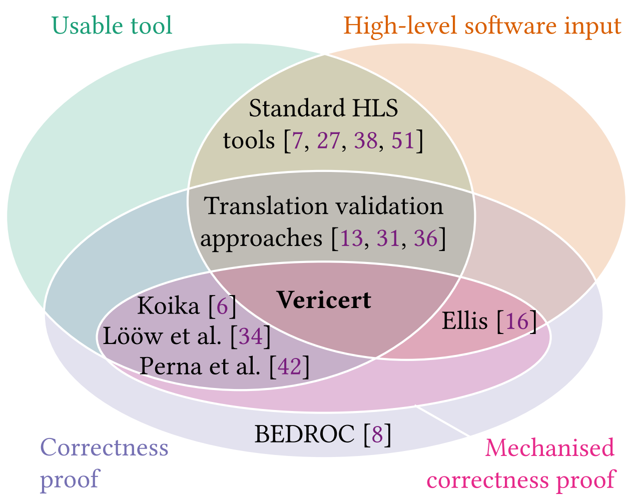
Usable tool
High-level software input
Standard HLS tools [7, 27, 38, 51]
Translation validation approaches [13, 31, 36]
Koika [6]
Vericert
Lööw et al. [34]
Perna et al. [42]
Ellis [16]
BEDROC [8]
Correctness proof
Mechanised correctness proof

Figure 9. Summary of related work

5.4 RQ3: How long does Vericert take to produce hardware?

Figure 8 compares the compilation times of Vericert and of LegUp, with each data point corresponding to one of the PolyBench/C benchmarks. On average, Vericert compilation is about 47× faster than LegUp compilation. Vericert is much faster because it omits many of the time-consuming HLS optimisations performed by LegUp, such as scheduling and memory analysis. This comparison also demonstrates that our fully verified approach does not add substantial overheads in compilation time, since we do not invoke verification for every compilation instance, unlike the approaches based on translation validation that we mentioned in Section 1.

6 Related Work

A summary of the related works can be found in Figure 9, which is represented as a Venn diagram. The categories that were chosen for the Venn diagram are: if the tool is usable and available, if it takes a high-level software language as input, if it has a correctness proof and finally if that proof is mechanised. The goal of Vericert is to cover all of these categories.

Most practical HLS tools [7, 27, 38, 51] fit into the category of usable tools that take high-level inputs. On the other spectrum, there are tools such as BEDROC [8] 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 [45] 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 [36], 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 complex optimisations such as scheduling [11, 29, 53] or code motion [3, 12], 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 [35] used a verified translation from HOL4 code describing state transitions into Verilog to design a verified processor [34]. Their approach translated a shallow embedding in HOL4 into a deep embedding of Verilog. Perna et al. [42, 43] designed a formally verified translation from a deep embedding of Handel-C [1], which is translated to a deep embedding of a circuit. Finally, Ellis [16] 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.

7 Conclusion

We have presented Vericert, the first mechanically verified HLS tool for translating software in C into hardware in Verilog. We built Vericert by extending CompCert with a new hardware-specific intermediate language and a Verilog back end, and we verified it with respect to a semantics for Verilog due to Lööw and Myreen [35]. We evaluated Vericert against the existing LegUp HLS tool on the Polybench/C benchmark suite. Currently, our hardware is 56× slower and 21× larger compared to LegUp, though it is only 10× slower if inefficient divisions are removed.

There are abundant opportunities for improving Vericert's performance. For instance, as discussed in Section 5, simply replacing the naïve single-cycle division and modulo operations with C implementations increases clock frequency by 5.6×. Beyond this, we plan to implement scheduling and loop pipelining, since this allows more operations to be packed into fewer clock cycles; recent work by Six et al. [49] indicates how these scheduling algorithms can be implemented in CompCert. Other optimisations include resource sharing to reduce the circuit area, and using tailored hardware operators that use hard IP blocks on chip and can be pipelined.

Finally, it's worth considering how trustworthy Vericert is compared to other HLS tools. The guarantee of full functional equivalence between input and output that *Vericert* provides is a strong one, the semantics for the source and target languages are both well-established, and Coq is a mature and thoroughly tested system. However, Vericert cannot guarantee to provide an output for every valid input – for instance, as remarked in Section 4.2, Vericert will error out if given a program with more than about four million instructions! – but our evaluation indicates that it does not seem to error out too frequently. And of course, Vericert cannot guarantee that the final hardware produced will be correct, because the Verilog it generates must pass through a series of unverified tools along the way. This concern may be allayed in the future by ongoing work we are aware of to produce a verified hardware synthesis tool.

References

[1] Matthew Aubury, Ian Page, Geoff Randall, Jonathan Saul, and Robin Watts. 1996. Handel-C language reference guide. *Computing Laboratory. Oxford University, UK* (1996). 12

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. IEEE, 1212–1221. https://doi.org/10.1145/2228360.2228584 2

[3] 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 1, 12

[4] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-07964-5 2

[5] Sandrine Blazy and Xavier Leroy. 2005. Formal Verification of a Memory Model for C-Like Imperative Languages. In *Formal Methods and Software Engineering*, Kung-Kiu Lau and Richard Banach (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 280–299. https://doi.org/0.1007/11576280_20 4

[6] 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 2, 12

[7] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Helge Anderson, Stephen Dean Brown, and Tomasz S. Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *FPGA*. ACM, 33–36. https://doi.org/10.1145/1950413.1950423 1, 2, 10, 12

[8] R. Chapman, G. Brown, and M. Leeser. 1992. Verified high-level synthesis in BEDROC. In *[1992] Proceedings The European Conference on Design Automation*. IEEE Computer Society, 59–63. https://doi.org/10.1109/EDAC.1992.205894 12

[9] Pankaj Chauhan. 2020. Formally Ensuring Equivalence between C++ and RTL designs. https://bit.ly/2KbT0ki 2

[10] Y. Choi and J. Cong. 2018. HLS-Based Optimization and Design Space Exploration for Applications with Variable Loop Bounds. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/3240765.3240815 10

[11] 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 1, 12

[12] 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 1, 12

[13] E. Clarke, D. Kroening, and K. Yorav. 2003. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*. 368–371. https://doi.org/10.1145/775832.775928 12

[14] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees A. Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 30, 4 (2011), 473–491. https://doi.org/10.1109/TCAD.2011.2110592 3

[15] Zewei Du, Yann Herklotz, Nadesh Ramanathan, and John Wickerson. [n.d.]. Fuzzing High-Level Synthesis Tools. ([n. d.]). https://yannherklotz.com/docs/drafts/fuzzing_hls.pdf Unpublished. 1

[16] Martin Ellis. 2008. *Correct synthesis and integration of compiler-generated function units*. Ph.D. Dissertation. Newcastle University. https://theses.ncl.ac.uk/jspui/handle/10443/828 2, 12

[17] 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 2

[18] Stephane Gauthier and Zubair Wadood. 2020. High-Level Synthesis: Can it outperform hand-coded HDL? https://bit.ly/2IDhKBR 1

[19] David J. Greaves. 2019. Research Note: An Open Source Bluespec Compiler. *CoRR* abs/1905.03746 (2019). 2

[20] David J. Greaves and Satnam Singh. 2008. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *FCCM*. IEEE Computer Society, 3–12. https://doi.org/10.1109/FCCM.2008.46 2

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

[22] Enoch Hwang, Frank Vahid, and Yu-Chin Hsu. 1999. FSMD functional partitioning for low power. In *Proceedings of the conference on Design, automation and test in Europe*. 7–es. https://doi.org/10.1109/DATE.1999.761092 4

[23] 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (April 2006), 1–590. https://doi.org/10.1109/IEEESTD.2006.99495 2, 5

[24] 2005. IEEE Standard for Verilog Register Transfer Level Synthesis. *IEC 62142-2005 First edition 2005-06 IEEE Std 1364.1* (2005), 1–116. https://doi.org/10.1109/IEEESTD.2005.339572 5

[25] Intel. 2020. High-level Synthesis Compiler. https://intel.ly/2UDiWr5 1, 2

[26] Intel. 2020. Intel® Quartus® Prime Software Suite. https://intel.ly/3fpUNhv 10

[27] Intel. 2020. SDK for OpenCL Applications. https://intel.ly/30sYHz0 12

[28] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *Programming Languages and Systems*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 397–416. https://doi.org/10.1007/978-3-642-28869-2_20 2

[29] 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 1, 12

[30] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *PLDI*. ACM, 296–311. https://doi.org/10.1145/3192366.3192379 2

[31] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. 2008. Validating High-Level Synthesis. In *Computer Aided Verification*, Aarti Gupta and Sharad Malik (Eds.). Springer, Berlin, Heidelberg, 459–472. https://doi.org/10.1007/978-3-540-70545-1_44 12

[32] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814 2, 7

[33] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/2737924.2737986 1

[34] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. 2019. Verified Compilation on a Verified Processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 1041–1053. https://doi.org/10.1145/3314221.3314622 2, 5, 12

[35] 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 2, 5, 6, 12

[36] Mentor. 2020. Catapult High-Level Synthesis. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls 2, 12

[37] P. Meredith, M. Katelman, J. Meseguer, and G. Roşu. 2010. A formal executable semantics of Verilog. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*. 179–188. https://doi.org/10.1109/MEMCOD.2010.5558634 2, 5

[38] 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 12

[39] R. Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 69–70. https://doi.org/10.1109/MEMCOD.2004.1459818 2

[40] Ian Page and Wayne Luk. 1991. Compiling Occam into field-programmable gate arrays. In *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, Vol. 15. 271–283. 2

[41] Maxime Pelcat, Cédric Bourrasset, Luca Maggiani, and François Berry. 2016. Design productivity of a high level synthesis compiler versus HDL. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. 140–147. https://doi.org/10.1109/SAMOS.2016.7818341 1

[42] Juan Perna and Jim Woodcock. 2012. Mechanised Wire-Wise Verification of Handel-C Synthesis. *Science of Computer Programming* 77, 4 (2012), 424 – 443. https://doi.org/10.1016/j.scico.2010.02.007 12

[43] Juan Perna, Jim Woodcock, Augusto Sampaio, and Juliano Iyoda. 2011. Correct Hardware Synthesis. *Acta Informatica* 48, 7 (01 Dec 2011), 363–396. https://doi.org/10.1007/s00236-011-0142-y 12

[44] Christian Pilato and Fabrizio Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *FPL*. IEEE, 1–4. https://doi.org/10.1109/FPL.2013.6645550 1, 2

[45] 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, 151–166. https://doi.org/10.1007/BFb0054170 1, 12

[46] Louis-Noël Pouchet. 2020. PolyBench/C: the Polyhedral Benchmark suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/ 2, 10

[47] Louis-Noel Pouchet, Peng Zhang, Ponnuswamy Sadayappan, and Jason Cong. 2013. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. 29–38. https://doi.org/10.1145/2435264.2435273 10

[48] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: A Multi-Level Intermediate Representation for Hardware Description Languages. 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, 258–271. https://doi.org/10.1145/3385412.3386024 2

[49] Cyril Six, Sylvain Boulmé, and David Monniaux. 2020. Certified and efficient instruction scheduling: Application to interlocked VLIW processors. *Proc. ACM Program. Lang.* OOPSLA (2020). 12

[50] David B. Thomas. 2016. Synthesisable recursion for C++ HLS tools. In *ASAP*. IEEE Computer Society, 91–98. https://doi.org/10.1109/ASAP.2016.7760777 3

[51] Xilinx. 2020. Vivado High-level Synthesis. https://bit.ly/39ereMx 1, 2, 12

[52] 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 1

[53] 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 1, 12

[54] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 430–437. 10

[55] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. *SIGPLAN Not.* 47, 1 (Jan. 2012), 427–440. https://doi.org/10.1145/2103621.2103709 2

[56] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. 2013. Improving polyhedral code generation for high-level synthesis. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 1–10. https://doi.org/10.1109/CODES-ISSS.2013.6659002 10