Formal Verification of High-Level Synthesis

YANN HERKLOTZ, Imperial College London, UK
JAMES D. POLLARD, Imperial College London, UK
NADESH RAMANATHAN, Imperial College London, UK
JOHN WICKERSON, Imperial College London, UK

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 such as 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. Furthermore, 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 most C constructs, including all integer operations, function calls, local arrays, structs, unions, and general control-flow statements. An evaluation on the PolyBench/C benchmark suite indicates that Vericert generates hardware that is around an order of magnitude slower (only around 2× slower in the absence of division) and about the same size as hardware generated by an existing, optimising (but unverified) HLS tool.

CCS Concepts: • **Hardware → High-level and register-transfer level synthesis**; • **Software and its engineering → Formal software verification**; • **Theory of computation → Program verification**.

Additional Key Words and Phrases: CompCert, Coq, high-level synthesis, C, Verilog

ACM Reference Format:
Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal Verification of High-Level Synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 117 (October 2021), 30 pages. https://doi.org/10.1145/3485494

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 [Canis et al. 2011], Vivado HLS [Xilinx 2020], Intel i++ [Intel 2020a], and Bambu HLS [Pilato and Ferrandi 2013] promise designs with comparable performance and energy-efficiency to those hand-written in an HDL [Gauthier and Wadood 2020;

Authors' addresses: Yann Herklotz, Imperial College London, UK, yann.herklotz15@imperial.ac.uk; James D. Pollard, Imperial College London, UK, jamespollard@acm.org; Nadesh Ramanathan, Imperial College London, UK, n.ramanathan@ieee.org; John Wickerson, Imperial College London, UK, j.wickerson@imperial.ac.uk.

2475-1421/2021/10-ART117
https://doi.org/10.1145/3485494

Homsirikamol and Gaj 2014; Pelcat et al. 2016], 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. [2015] 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. [2021a] fuzz-tested three commercial HLS tools using Csmith [Yang et al. 2011], 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 were compiled to designs that behaved incorrectly.

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 generated 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.

One alternative is to use *translation validation* [Pnueli et al. 1998] to prove equivalence between the input program and the output design. Translation validation has been successfully applied to several HLS optimisations [Banerjee et al. 2014; Chouksey and Karfa 2020; Chouksey et al. 2019; Karfa et al. 2006; Youngsik Kim et al. 2004]. Nevertheless, it 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 [Mentor 2020] may require several rounds of expert 'adjustments' [Chauhan 2020, 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 [e.g. Tristan and Leroy 2008], which has not been the case in HLS tools to date.

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 design it produces always has the same behaviour as its input program. 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 [Leroy 2009] with a new hardware-specific intermediate language and a Verilog back end. It supports most C constructs, including integer operations, function calls (which are all inlined), local arrays, structs, unions, and general control-flow statements, but currently excludes support for case statements, function pointers, recursive function calls, non-32-bit integers, 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, including certain optimisations related to memory accesses and division.
- We state the correctness theorem of Vericert with respect to an existing semantics for Verilog due to Lööw and Myreen [2019]. In Section 3, we describe how we extended this semantics to

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

make it suitable as an HLS target. We also describe how the Verilog semantics is integrated into CompCert's language execution model and its framework for performing simulation proofs. A mapping of CompCert's infinite memory model onto a finite Verilog array is also described.

- In Section 4, 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 the byte-addressed memory assumed by the input software and the word-addressed memory that we implement in the output hardware, different handling of unsigned comparisons between C and Verilog, and carefully implementing memory reads and writes so that these behave properly as a RAM in hardware.
- In Section 5, we evaluate Vericert on the PolyBench/C benchmark suite [Pouchet 2020], and compare the performance of our generated hardware against an existing, unverified HLS tool called LegUp [Canis et al. 2011]. We show that Vericert generates hardware that is 27× slower (2× slower in the absence of division) and 1.1× larger than that generated by LegUp. This performance gap can be largely attributed to Vericert's current lack of support for instruction-level parallelism and the absence of an efficient, pipelined division operator. We intend to close this gap in the future by introducing (and verifying) HLS optimisations of our own, such as scheduling and memory analysis. This section also reports on our campaign to fuzz-test Vericert using over a hundred thousand random C programs generated by Csmith [Yang et al. 2011] in order to confirm that its correctness theorem is watertight.

Companion material. Vericert is fully open source and available on GitHub at https://github.com/ymherklotz/vericert. A snapshot of the Vericert development is also available in a Zenodo repository [Herklotz et al. 2021b].

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 also covers an example of converting a simple C program into hardware, expressed in the Verilog language.

2.1 Main Design Decisions

Choice of source language. C was chosen as the source language as it remains the most common source language amongst production-quality HLS tools [Canis et al. 2011; Intel 2020a; Pilato and Ferrandi 2013; Xilinx 2020]. This, in turn, may be because it is "[t]he starting point for the vast majority of algorithms to be implemented in hardware" [Gajski et al. 2010], lending a degree of practicality. The availability of CompCert [Leroy 2009] also provides a solid basis for formally verified C compilation. We considered Bluespec [Nikhil 2004], but decided that although it "can be classed as a high-level language" [Greaves 2019], 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 [Page and Luk 1991], Spatial [Koeplinger et al. 2018] or Scala [Bachrach et al. 2012].

Choice of target language. Verilog [IEEE Std 1364 2006] 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 [Lööw et al. 2019; Meredith et al. 2010]. Bluespec, previously

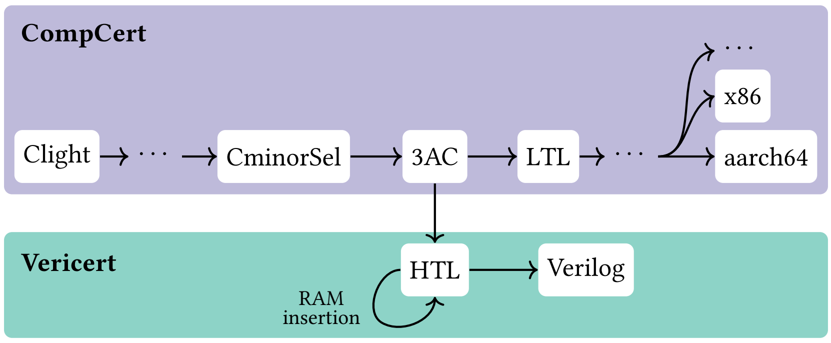


Fig. 1. Vericert as a Verilog back end to CompCert.

ruled out as a source language, is another possible target and there exists a formally verified translation to circuits using Kôika [Bourgeat et al. 2020].

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 [Ellis 2008]. CompCert [Leroy 2009] was chosen as the front end because it has a well established framework for simulation proofs about intermediate languages, and it already provides a validated C parser [Jourdan et al. 2012]. The Vellvm framework [Zhao et al. 2012] 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 [Greaves and Singh 2008], and LLHD [Schuiki et al. 2020] 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 Fig. 1, which shows those parts of the translation that are performed in CompCert, and those that have been added. This includes translations to two new intermediate languages added in Vericert, HTL and Verilog, as well as an additional optimisation pass labelled as "RAM insertion".

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, despite being designed for software compilers, have been found useful in HLS tools as well [Cong et al. 2011]. And 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

[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.

```
module main(input rst, input y, input clk,
            output reg z);
  reg tmp, state;
  always @(posedge clk)
    case (state)
      1'b0: tmp <= y;
      1'b1: begin tmp <= 1'b0; z <= tmp; end
    endcase
  always @(posedge clk)
    if (rst) state <= 1'b0;
    else case (state)
      1'b0: if (y) state <= 1'b1;
            else state <= 1'b0;
      1'b1: state <= 1'b0;
    endcase
endmodule
```

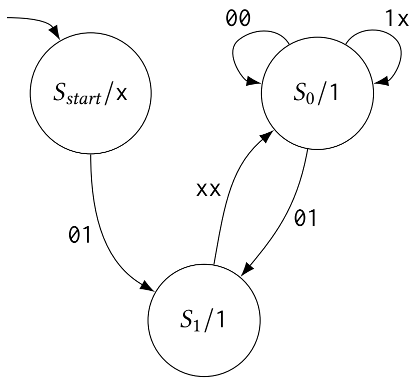


Fig. 2. A simple state machine implemented in Verilog, with its diagrammatic representation on the right. The x stands for "don't care" and each transition is labelled with the values of the inputs `rst` and `y` that trigger the transition. The output that will be produced is shown in each state.

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 back end because it generally produces relatively dense 3AC thanks to the availability of complex addressing modes.

2.2 An Introduction to Verilog

This section will introduce Verilog for readers who may not be familiar with the language, concentrating on the features that are used in the output of Vericert. Verilog is a hardware description language (HDL) and is used to design hardware ranging from complete CPUs that are eventually produced as integrated circuits, to small application-specific accelerators that are placed on FPGAs. Verilog is a popular language because it allows for fine-grained control over the hardware, and also provides high-level constructs to simplify development.

Verilog behaves quite differently to standard software programming languages due to it having to express the parallel nature of hardware. The basic construct to achieve this is the always-block, which is a collection of assignments that are executed every time some event occurs. In the case of Vericert, this event is either a positive (rising) or a negative (falling) clock edge. All always-blocks triggering on the same event are executed in parallel. Always-blocks can also express control-flow using if-statements and case-statements.

A simple state machine can be implemented as shown in Fig. 2. At every positive edge of the clock (`clk`), both of the always-blocks will trigger simultaneously. The first always-block controls the values in the register `x` and the output `z`, while the second always-block controls the next state the state machine should go to. When the `state` is 0, `tmp` will be assigned to the input `y` using nonblocking assignment, denoted by <=. Nonblocking assignment assigns registers in parallel at the end of the clock cycle, rather than sequentially throughout the always-block. In the second always-block, the input `y` will be checked, and if it's high it will move on to the next state, otherwise it will stay in the current state. When `state` is 1, the first always-block will reset the value of `tmp` and then set `z` to the original value of `tmp`, since nonblocking assignment does not change its value until the end of the clock cycle. Finally, the last always-block will set the state to 0 again.

```c
int main() {
    int x[2] = {3, 6};
    int i = 1;
    return x[i];
}
```

(a) Example C code passed to Vericert.

```
main() {
    x5 = 3
    int32[stack(0)] = x5
    x4 = 6
    int32[stack(4)] = x4
    x1 = 1
    x3 = stack(0) (int)
    x2 = int32[x3 + x1
                  * 4 + 0]
    return x2
}
```

(b) 3AC produced by the CompCert front-end without any optimisations.

```verilog
1   module main(reset, clk, finish, return_val);
2     input [0:0] reset, clk;
3     output reg [0:0] finish = 0;
4     output reg [31:0] return_val = 0;
5     reg [31:0] reg_3 = 0, addr = 0, d_in = 0, reg_5 = 0, wr_en = 0;
6     reg [0:0] en = 0, u_en = 0;
7     reg [31:0] state = 0, reg_2 = 0, reg_4 = 0, d_out = 0, reg_1 = 0;
8     reg [31:0] stack [1:0];
9     // RAM interface
10    always @(negedge clk)
11      if ({u_en != en}) begin
12        if (wr_en) stack[addr] <= d_in;
13        else d_out <= stack[addr];
14        en <= u_en;
15      end
16    // Data-path
17    always @(posedge clk)
18      case (state)
19        32'd11: reg_2 <= d_out;
20        32'd8: reg_5 <= 32'd3;
21        32'd7: begin u_en <= ( ~ u_en); wr_en <= 32'd1;
22                     d_in <= reg_5; addr <= 32'd0; end
23        32'd6: reg_4 <= 32'd6;
24        32'd5: begin u_en <= ( ~ u_en); wr_en <= 32'd1;
25                     d_in <= reg_4; addr <= 32'd1; end
26        32'd4: reg_1 <= 32'd1;
27        32'd3: reg_3 <= 32'd0;
28        32'd2: begin u_en <= ( ~ u_en); wr_en <= 32'd0;
29                     addr <= {{{reg_3 + 32'd0} + {reg_1 * 32'd4}} / 32'd4}; end
30        32'd1: begin finish = 32'd1; return_val = reg_2; end
31        default: ;
32      endcase
33    // Control logic
34    always @(posedge clk)
35      if ({reset == 32'd1}) state <= 32'd8;
36      else case (state)
37             32'd11: state <= 32'd1;         32'd4: state <= 32'd3;
38             32'd8: state <= 32'd7;          32'd3: state <= 32'd2;
39             32'd7: state <= 32'd6;          32'd2: state <= 32'd11;
40             32'd6: state <= 32'd5;          32'd1: ;
41             32'd5: state <= 32'd4;          default: ;
42           endcase
43  endmodule
```

(c) Verilog produced by Vericert. It demonstrates the instantiation of the RAM (lines 9–15), the data-path (lines 16–32) and the control logic (lines 33–42).

Fig. 3. Translating a simple program from C to Verilog.

2.3 Translating C to Verilog by Example

Fig. 3 illustrates the translation of a simple program that stores and retrieves values from an array. In this section, we describe the stages of the Vericert translation, referring to this program as an example.

2.3.1 Translating C to 3AC. The first stage of the translation uses unmodified CompCert to transform the C input, shown in Fig. 3a, into a 3AC intermediate representation, shown in Fig. 3b. As part of this translation, function inlining is 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 and is therefore a common HLS optimisation [Noronha et al. 2017]. Inlining precludes support for recursive function calls, but this feature is not supported in most HLS tools anyway [Thomas 2016].

2.3.2 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 [Hwang et al. 1999]. The core idea of the FSMD

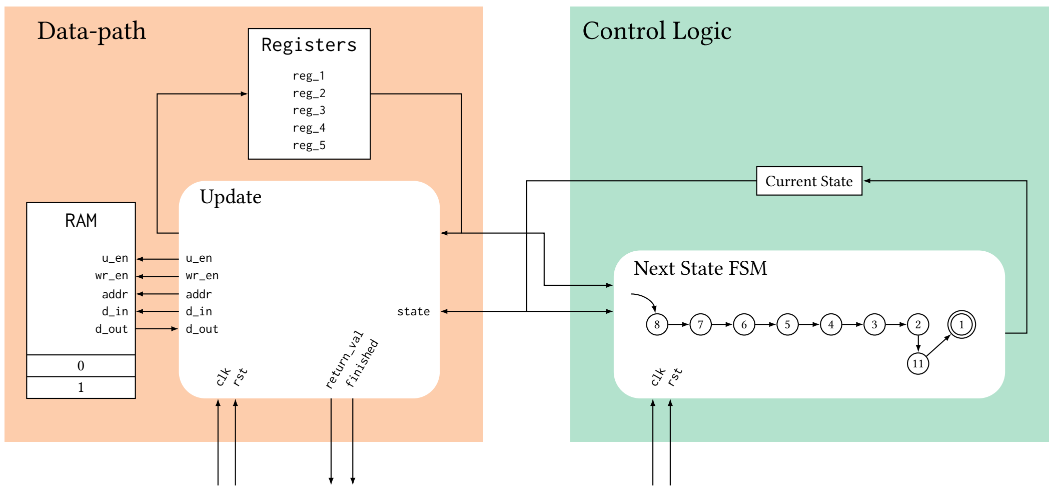


Fig. 4. The FSMD for the example shown in Fig. 3, split into a data-path and control logic for the next state calculation. The Update block takes the current state, current values of all registers and at most one value stored in the RAM, and calculates a new value that can either be stored back in the RAM or in a register.

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: the *control logic* map, which expresses state transitions, and the *data-path* map, which expresses computations. Fig. 4 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.

The HTL language was mainly introduced to simplify the proof of translation from 3AC to Verilog, as these languages have very different semantics. It serves as an intermediate language with similar semantics to 3AC at the top level, using maps to represents what to execute at every state, and similar semantics to Verilog at the lower level by already using Verilog statements instead of more abstract instructions. Compared to plain Verilog, HTL is simpler to manipulate and analyse, thereby making it easier to prove optimisations like proper RAM insertion.

Translating memory. Typically, HLS-generated hardware consists of a sea of registers and RAMs. This memory view is very different from the C memory model, so we perform the following translation from CompCert's abstract memory model to a concrete RAM. 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 representing the RAM block. Any loads and stores are temporarily translated to direct accesses to this array, where each address has its offset removed and is divided by four. In a separate HTL-to-HTL conversion, these direct accesses are then translated to proper loads and stores that use a RAM interface to communicate with the RAM, shown on lines 21, 24 and 28 of Fig. 3c. This pass inserts a RAM block with the interface around the unpacked array. Without this interface and without the RAM block, the synthesis tool processing the Verilog hardware description would not identify the array as a RAM, and would instead implement it using many registers. This interface is shown on lines 9–15 in the Verilog code in Fig. 3c. A high-level overview of the architecture and of the RAM interface can be seen in Fig. 4.

Translating instructions. Most 3AC instructions correspond to hardware constructs. For example, line 2 in Fig. 3b shows a 32-bit register `x5` being initialised to 3, after which the control flow moves execution to line 3. This initialisation is also encoded in the Verilog generated from HTL at state 8 in both the control logic and data-path always-blocks, shown at lines 33 and 16 respectively in Fig. 3c. Simple operator instructions are translated in a similar way. For example, the add instruction is just translated to the built-in add operator, similarly for the multiply operator. All 32-bit instructions can be translated in this way, but some special instructions require extra care. One such instruction is the `Oshrximm` instruction, which is discussed further in Section 2.4.3. Another is the `Oshldimm` instruction, which is a left rotate instruction that has no Verilog equivalent and therefore has to be implemented in terms of other operations and proven to be equivalent. The only 32-bit instructions that we do not translate are case-statements (`Ijumptable`) and those instructions related to function calls (`Icall`, `Ibuiltin`, and `Itailcall`), because we enable inlining by default.

2.3.3 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 case-statements. 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. In addition to translating the maps of Verilog statements, an always-block that will behave like the RAM also has to be created, which is only modelled abstractly at the HTL level. Fig. 3c 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. One main example of this is proving that the specification of the RAM in HTL does indeed behave in the same as its Verilog implementation. We discuss these proofs in upcoming sections.

2.4 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.

2.4.1 Byte- and Word-Addressable Memories. One big difference between C and Verilog is how memory is represented. Although Verilog arrays use similar syntax to C arrays, they must be treated quite differently. 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 [Blazy and Leroy 2005]. If a byte-addressable memory was used in the target hardware, which is closer to CompCert's memory model, then a load and store would instead take four clock cycles, because a RAM can only perform one read and write per clock cycle. 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 only integer loads and stores are currently supported in Vericert, it follows that the addresses given to the loads and stores will be multiples of four. Translating from byte-addressed memory to word-addressed memory can then be done by dividing the address by four.

2.4.2 Implementation of RAM Interface. The simplest way to implement loads and stores in Vericert would be to access the Verilog array directly from within the data-path (i.e., inside the always-block on lines 16–32 of Fig. 3c). This would be correct, but when a Verilog array is accessed at several program points, the synthesis tool is unlikely to detect that it can be implemented as a RAM block, and will resort to using lots of registers instead, ruining the circuit's area and performance. To avert this, we arrange that the data-path does not access memory directly, but simply sets the address it wishes to access and then toggles the `u_en` flag. This activates the RAM interface (lines 9–15 of Fig. 3c) on the next falling clock edge, which performs the requested load or store. By factoring all the memory accesses out into a separate interface, we ensure that the underlying array is only accessed from a single program point in the Verilog code, and thus ensure that the synthesis tool will correctly infer a RAM block.[5]

Therefore, an extra compiler pass is added from HTL to HTL to extract all the direct accesses to the Verilog array and replace them by signals that access the RAM interface in a separate always-block. The translation is performed by going through all the instructions and replacing each load and store expression in turn. Stores can simply be replaced by the necessary wires directly. Loads are a little more subtle: loads that use the RAM interface take two clock cycles where a direct load from an array takes only one, so this pass inserts an extra state after each load.

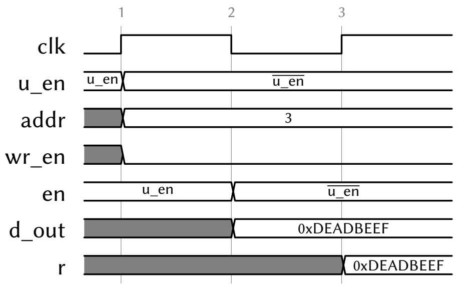
There are two interesting parts to the inserted RAM interface. Firstly, the memory updates are triggered on the negative (falling) edge of the clock, out of phase with the rest of the design which is triggered on the positive (rising) edge of the clock. The advantage of this is that instead of loads and stores taking three clock cycles and two clock cycles respectively, they only take two clock cycles and one clock cycle instead, greatly improving their performance. Using the negative edge of the clock is widely supported by synthesis tools, and does not affect the maximum frequency of the final design.

Secondly, the logic in the enable signal of the RAM (`en != u_en`) is also atypical in hardware designs. Enable signals are normally manually controlled and inserted into the appropriate states, by using a check like the following in the RAM: `en == 1`. This means that the RAM only turns on when the enable signal is set. However, to make the proof simpler and avoid reasoning about possible side effects introduced by the RAM being enabled but not used, a RAM which disables itself after every use would be ideal. One method for implementing this would be to insert an extra state after each load or store that disables the RAM, but this extra state would eliminate the speed advantage of the negative-edge-triggered RAM. Another method would be to determine the next state after each load or store and disable the RAM in that state, but this could quickly become complicated, especially in the case where the next state also contains a memory operation, and hence the disable signal should not be added. The method we ultimately chose was to have the RAM become enabled not when the enable signal is high, but when it *toggles* its value. This can be arranged by keeping track of the old value of the enable signal in `en` and comparing it to the current value `u_en` set by the data-path. When the values are different, the RAM gets enabled, and then `en` is set to the value of `u_en`. This ensures that the RAM will always be disabled straight after it was used, without having to insert or modify any other states.

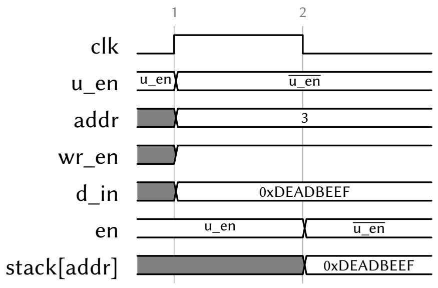
Fig. 5 gives an example of how the RAM interface behaves when values are loaded and stored.

2.4.3 Implementing the `Oshrximm` Instruction. Many of the CompCert instructions map well to hardware, but `Oshrximm` (efficient signed division by a power of two using a logical shift) is expensive if implemented naïvely. The problem is that in CompCert it is specified as a signed

[5]Interestingly, the Verilog code shown for the RAM interface must not be modified, because the synthesis tool will only generate a RAM when the code matches a small set of specific patterns.



(a) Timing diagram for loads. At time 1, the u_en signal is toggled to enable the RAM. At time 2, d_out is set to the value stored at the address in the RAM, which is finally assigned to the register r at time 3.



(b) Timing diagram for stores. At time 1, the u_en signal is toggled to enable the RAM, and the address addr and the data to store d_in are set. On the negative edge at time 2, the data is stored into the RAM.

Fig. 5. Timing diagrams showing the execution of loads and stores over multiple clock cycles.

division:

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

(where $x, y \in \mathbb{Z}$, $0 \le y < 31$, and $-2^{31} \le x < 2^{31}$) 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. It should therefore be implemented using a sequence of shifts.

CompCert eventually performs a translation from this representation into assembly code which uses shifts to implement the division, however, the specification of the instruction in 3AC itself still uses division instead of shifts, meaning this proof of the translation cannot be reused. In Vericert, the equivalence of the representation in terms of divisions and shifts is proven over the integers and the specification, thereby making it simpler to prove the correctness of the Verilog implementation in terms of shifts.

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 [IEEE Std 1364 2006; IEEE Std 1364.1 2005], 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.

The Verilog semantics we use is ported to Coq from a semantics written in HOL4 by Lööw and Myreen [2019] and used to prove the translation from HOL4 to Verilog [Lööw et al. 2019]. 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. [2010].

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 positive or negative 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 that is triggered at the positive edge of the clock 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 arrays, and Δ, which contains the values that will be assigned at the end of the clock cycle. Σ can therefore be defined as follows: $\Sigma = (\Gamma, \Delta)$. 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:

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

where Γ is the initial state of all the variables, ϵ is the empty map because the Δ map is assumed to be empty at the start of the clock cycle, 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

Five changes were made to the semantics proposed by Lööw and Myreen [2019] to make it suitable as an 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 further split up into Γ_r for instantaneous updates to variables, and Γ_a for instantaneous updates to arrays ($\Gamma = (\Gamma_r, \Gamma_a)$); Δ is split similarly ($\Delta = (\Delta_r, \Delta_a)$). 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 negative edge support. To reason about circuits that execute on the negative edge of the clock (such as our RAM interface described in Section 2.4.2), support for negative-edge-triggered always-blocks was added to the semantics. This is shown in the modifications of the Module rule shown below:

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

The main execution of the module $\downarrow_{\text{module}}$ is split into $\downarrow_{\text{module}^+}$ and $\downarrow_{\text{module}^-}$, which are rules that only execute always-blocks triggered at the positive and at the negative edge respectively. The positive-edge-triggered always-blocks are processed in the same way as in the original Module rule. The output maps Γ' and Δ' are then merged and passed as the blocking assignments map into the negative edge execution, so that all the blocking and nonblocking assignments are present. Finally, all the negative-edge-triggered always-blocks are processed and merged to give the final state.

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.

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 four special-purpose registers in order to integrate it into CompCert.

CompCert executions pass through three main states:

`State` $sf\ m\ v\ \Gamma_r\ \Gamma_a$ 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.

`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 $\vec{r}$.

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

$$\textsc{Finish}\quad \frac{\Gamma_r[fin] = 1}{\texttt{State}\ sf\ m\ \sigma\ \Gamma_r\ \Gamma_a \longrightarrow \texttt{Returnstate}\ sf\ \Gamma_r[ret]}$$

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

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

Fig. 6. Top-level small-step semantics for Verilog modules in CompCert's computational framework.

`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 four 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 *ret* 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*.

Fig. 6 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 ever to 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 Fig. 6, an initial state and final state need to be defined:

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

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

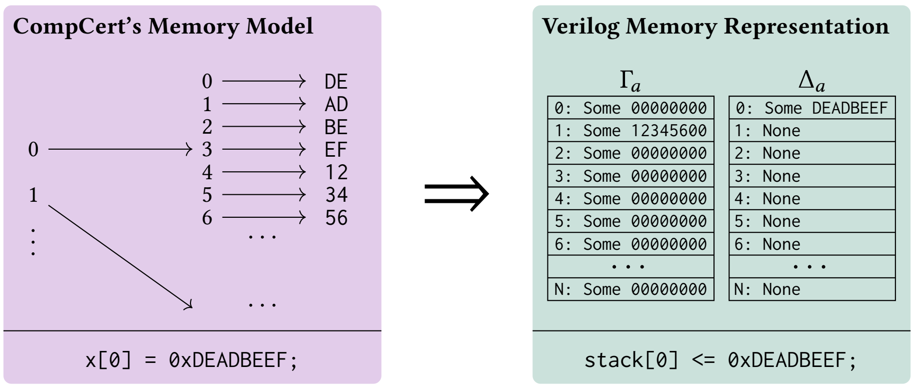


Fig. 7. Change in the memory model during the translation of 3AC into HTL. The state of the memories in each case is right after the execution of the store to memory.

final state results in the program output of value n when reaching a `Returnstate` with an empty stack frame.

3.3 Memory Model

The Verilog semantics do not define a memory model for Verilog, as this is not needed for a hardware description language. There is no preexisting architecture that Verilog will produce; it can describe any memory layout that is needed. Instead of having specific semantics for memory, the semantics only needs to support the language features that can produce these different memory layouts, these being Verilog arrays. We therefore define semantics for updating Verilog arrays using blocking and nonblocking assignment. We then have to prove that the C memory model that CompCert uses matches with the interpretation of arrays used in Verilog. The CompCert memory model is infinite, whereas our representation of arrays in Verilog is inherently finite. There have already been efforts to define a general finite memory model for all intermediate languages in CompCert, such as CompCertS [Besson et al. 2018] or CompCert-TSO [Ševčík et al. 2013], or keeping the intermediate languages intact and translate to a more concrete finite memory model in the back end, such as in CompCertELF [Wang et al. 2020]. We also define such a translation from CompCert's standard infinite memory model to finite arrays that can be represented in Verilog. There is therefore no more notion of an abstract memory model and all the interactions to memory are encoded in the hardware itself.

This translation is represented in Fig. 7. CompCert defines a map from blocks to maps from memory addresses to memory contents. Each block represents an area in memory; for example, a block can represent a global variable or a stack for a function. As there are no global variables, the main stack can be assumed to be block 0, and this is the only block we translate. Meanwhile, our Verilog semantics defines two finite arrays of optional values, one for the blocking assignments map Γ_a and one for the nonblocking assignments map Δ_a. The optional values are present to ensure correct merging of the two association maps at the end of the clock cycle. The invariant used in the proofs is that block 0 should be equivalent to the merged representation of the Γ_a and Δ_a maps.

4 CORRECTNESS 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 key ideas in the proof. The full Coq proof is available online [Herklotz et al. 2021b].

4.1 Main Challenges in the Proof

The proof of correctness of the Verilog back end is quite different from the usual proofs performed in CompCert, mainly because of the difference in the memory model and semantic differences between Verilog and CompCert's existing intermediate languages.

- As already mentioned in Section 3.3, because the memory model in our Verilog semantics is finite and concrete, but the CompCert memory model is more abstract and infinite with additional bounds, the equivalence of these models needs to be proven. Moreover, our memory is word-addressed for efficiency reasons, whereas CompCert's memory is byte-addressed.
- Second, the Verilog semantics operates quite differently to the usual intermediate languages in CompCert. All the CompCert intermediate languages use a map from control-flow nodes to instructions. An instruction can therefore be selected using an abstract program pointer. Meanwhile, in the Verilog semantics the whole design is executed at every clock cycle, because hardware is inherently parallel. The program pointer is part of the design as well, not just part of an abstract state. This makes the semantics of Verilog simpler, but comparing it to the semantics of 3AC becomes more challenging, as one has to map the abstract notion of the state to concrete values in registers.

Together, these differences mean that translating 3AC directly to Verilog is infeasible, as the differences in the semantics are too large. Instead, HTL, which was introduced in Section 2, bridges the gap in the semantics between the two languages. HTL still consists of maps, like many of the other CompCert languages, but each state corresponds to a Verilog statement.

4.2 Formulating the Correctness Theorem

The main correctness theorem is analogous to that stated in CompCert [Leroy 2009]: for all Clight source programs C, if the translation to the target Verilog code succeeds, $\mathit{Safe}(C)$ holds and the target Verilog has behaviour B when simulated, then C will have the same behaviour B. $\mathit{Safe}(C)$ means all observable behaviours of C are safe, which can be defined as $\forall B,\ C \Downarrow B \implies B \in \texttt{Safe}$. A behaviour is in `Safe` if it is either a final state (in the case of convergence) or divergent, but it cannot 'go wrong'. (This means that the source program must not contain undefined behaviour.) 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.

Theorem 1. *Whenever the translation from C succeeds and produces Verilog V, and all observable behaviours of C are safe, then V has behaviour B only if C has behaviour B.*

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

Why is this correctness theorem also the right one for HLS? It could be argued that hardware inherently runs forever and therefore does not produce a definitive final result. This would mean that the CompCert correctness theorem would probably be unhelpful with proving hardware correctness, as the behaviour would always be divergent. However, in practice, HLS does not normally produce the top-level of the design that needs to connect to other components, therefore needing to run forever. Rather, HLS often produces smaller components that take an input, execute, and then terminate with an answer. To start the execution of the hardware and to signal to the HLS component that the inputs are ready, the *rst* signal is set and unset. Then, once the result is ready, the *fin* signal is set and the result value is placed in *ret*. These signals are also present in the semantics of execution shown in Fig. 6. The correctness theorem therefore also uses these signals, and the proof shows that once the *fin* flag is set, the value in *ret* is correct according to the semantics of Verilog and Clight. Note that the compiler is allowed to fail and not produce any output; the correctness theorem only applies when the translation succeeds.

How can we prove this theorem? First, note that the theorem is a 'backwards simulation' result (every target behaviour must also be a source behaviour), following the terminology used in the CompCert literature [Leroy 2009]. The reverse direction (every source behaviour must also be a target behaviour) is not demanded because compilers are permitted to resolve any non-determinism present in their source programs. However, since Clight programs are all deterministic, as are the Verilog programs in the fragment we consider, we can actually reformulate the correctness theorem above as a forwards simulation result (following standard CompCert practice), which makes it easier to prove. To prove this forward simulation, it suffices to prove forward simulations between each pair of consecutive intermediate languages, 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 4.3), the forward simulation for the RAM insertion is shown in Lemma 4 (Section 4.4), then the forward simulation between HTL and Verilog is shown in Lemma 5 (Section 4.5), and finally, the proof that Verilog is deterministic is given in Lemma 6 (Section 4.6).

4.3 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. Instead of defining small-step semantics for each construct in Verilog, the semantics are 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). *Writing* `tr_htl` *for the translation from 3AC to HTL, we have:*

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

Proof sketch. 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. □

4.3.1 *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:

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

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:

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

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 logic.

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.*

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

4.3.2 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 $R \leq \Gamma_r$). 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: $M \leq \Gamma_a$.
- 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* `spec_htl` *holds for the translation.*

$$\begin{tikzcd}
R, M, pc \arrow[r, dash, "\mathcal{I} \land (R \leq \Gamma_r) \land (M \leq \Gamma_a) \land (pc = \Gamma_r[\sigma])"] \arrow[d] & \Gamma_r, \Gamma_a \arrow[d, dashed, "+"] \\
R', M', pc' \arrow[r, dash, dashed, "\mathcal{I} \land (R' \leq \Gamma_r') \land (M' \leq \Gamma_a') \land (pc' = \Gamma_r'[\sigma])"] & \Gamma_r', \Gamma_a'
\end{tikzcd}$$

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. □

$$\textsc{Idle}\quad \frac{\Gamma_{\mathrm{r}}[r.en] = \Gamma_{\mathrm{r}}[r.u_{en}]}{((\Gamma_{\mathrm{r}}, \Gamma_{\mathrm{a}}), \Delta, r) \downarrow_{\mathrm{ram}} \Delta}$$

$$\textsc{Load}\quad \frac{\Gamma_{\mathrm{r}}[r.en] \neq \Gamma_{\mathrm{r}}[r.u_{en}] \qquad \Gamma_{\mathrm{r}}[r.wr_{en}] = 0}{((\Gamma_{\mathrm{r}}, \Gamma_{\mathrm{a}}), (\Delta_{\mathrm{r}}, \Delta_{\mathrm{a}}), r) \downarrow_{\mathrm{ram}} (\Delta_{\mathrm{r}}[r.en \mapsto r.u_{en}, r.d_{out} \mapsto (\Gamma_{\mathrm{a}}[r.mem])[r.addr]], \Delta_{\mathrm{a}})}$$

$$\textsc{Store}\quad \frac{\Gamma_{\mathrm{r}}[r.en] \neq \Gamma_{\mathrm{r}}[r.u_{en}] \qquad \Gamma_{\mathrm{r}}[r.wr_{en}] = 1}{((\Gamma_{\mathrm{r}}, \Gamma_{\mathrm{a}}), (\Delta_{\mathrm{r}}, \Delta_{\mathrm{a}}), r) \downarrow_{\mathrm{ram}} (\Delta_{\mathrm{r}}[r.en \mapsto r.u\_en], \Delta_{\mathrm{a}}[r.mem \mapsto (\Gamma_{\mathrm{a}}[r.mem])[r.addr \mapsto r.d_{in}]])}$$

Fig. 8. Specification for the memory implementation in HTL, where r is the RAM, which is then implemented by equivalent Verilog code.

4.4 Forward Simulation of RAM Insertion

HTL can only represent a single state machine, so we must model the RAM abstractly to reason about the correctness of replacing the direct read and writes to the array by loads and stores to a RAM. The specification for the RAM is shown in Fig. 8, which defines how the RAM r will behave for all the possible combinations of the input signals.

4.4.1 From Implementation to Specification. The first step in proving the simulation correct is to build a specification of the translation algorithm. There are three possibilities for the transformation of an instruction. For each Verilog statement in the map at location i, the statement is either a load, a store, or neither. The load or store is translated to the equivalent representation using the RAM specification and all other instructions are left intact. An example of the specification for the translation of the store instruction is shown below, where σ is state register, r is the RAM, d and c are the input data-path and control logic maps, and i is the current state. (n is the newly inserted state, which only applies to the translation of loads.)

$$\textsc{Store Spec}\quad \frac{d[i] = (r.mem\texttt{[}e_1\texttt{]}\ \texttt{<=}\ e_2) \qquad t = (r.u\_en\ \texttt{<=}\ \neg r.u\_en; r.wr\_en\ \texttt{<=}\ 1; r.d\_in\ \texttt{<=}\ e_2; r.addr\ \texttt{<=}\ e_1)}{\texttt{spec\_ram\_tr}\ \sigma\ r\ d\ c\ d[i \mapsto t]\ c\ i\ n}$$

A similar specification is created for the load. We then also prove that the implementation of the translation proves that the specification holds.

4.4.2 From Specification to Simulation. Another simulation proof is performed to prove that the insertion of the RAM is semantics preserving. As in Lemma 3, we require some invariants that always hold at the start and end of the simulation. The invariants needed for the simulation of the RAM insertion are quite different to the previous ones, so we can define these invariants $\mathcal{I}_r$ to be the following:

- The association map for arrays Γ_a always needs to have the same arrays present, and these arrays should never change in size.
- The RAM should always be disabled at the start and the end of each simulation step. (This is why self-disabling RAM is needed.)

The other invariants and assumptions for defining two matching states in HTL are quite similar to the simulation performed in Lemma 3, such as ensuring that the states have the same value, and that the values in the registers are less defined. In particular, the less defined relation matches up all the registers, except for the new registers introduced by the RAM.

Lemma 4 (Forward simulation from HTL to HTL after inserting the RAM). *Given an HTL program, the forward-simulation relation should hold after inserting the RAM and wiring the load, store, and control signals.*

$$\forall h, h', B \in \texttt{Safe}, \quad \texttt{tr\_ram\_ins}(h) = h' \land h \Downarrow B \implies h' \Downarrow B.$$

4.5 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 is quite different to that of the case-statement to which it is converted.

Lemma 5 (Forward simulation from HTL to Verilog). *In the following, 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.)*

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

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

4.6 Deterministic Verilog Semantics

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

Lemma 6. *If a Verilog program V admits behaviours B_1 and B_2, then B_1 and B_2 must be the same.*

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

Proof sketch. The Verilog semantics is deterministic because the order of operation of all the constructs is defined, so there is only one way to evaluate the module, and hence only one possible behaviour. This was proven for the small-step semantics shown in Fig. 6. □

4.7 Coq Mechanisation

The lines of code for the implementation and proof of Vericert can be found in Table 1. Overall, it took about 1.5 person-years to build Vericert – about three person-months on implementation and 15 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 3069 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, instead of as a separate 'pointer' type like in the CompCert semantics, it was painful to reason about them. Many new theorems had to be proven about them in Vericert to prove the conversion from pointer to integer. Moreover, the second-largest proof of the correct RAM generation includes many proofs about the extensional equality of array operations, such as merging arrays with different

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

	Coq code	OCaml code	Spec	Theorems & Proofs	Total
Data structures and libraries	280	—	—	500	780
Integers and values	98	—	15	968	1081
HTL semantics	50	—	181	65	296
HTL generation	590	—	66	3069	3725
RAM generation	253	—	—	2793	3046
Verilog semantics	78	—	431	235	744
Verilog generation	104	—	—	486	590
Top-level driver, pretty printers	318	775	—	189	1282
Total	1721	775	693	8355	11544

assignments. As the negative edge implies that two merges take place every clock cycle, the proofs about the equality of the contents in memory and in the merged arrays become more tedious too.

Looking at the trusted computing base of Vericert, the Verilog semantics is 431 lines of code. This and the Clight semantics from CompCert are the only parts of the compiler that need to be trusted. The Verilog semantics specification is therefore much smaller compared to the 1721 lines of the implementation that are written in Coq, which are the verified parts of the HLS tool, even when the Clight semantics are added. In addition to that, understanding the semantics specification is simpler than trying to understand the translation algorithm. We conclude that the trusted base has been successfully reduced.

5 EVALUATION

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

RQ1 How fast is the hardware generated by Vericert?
RQ2 How area-efficient is the hardware generated by Vericert?
RQ3 How quickly does Vericert translate the C into Verilog?
RQ4 How effective is the correctness theorem in Vericert?

5.1 Experimental Setup

Choice of HLS tool for comparison. We compare Vericert against LegUp 4.0, because it is open-source and hence easily accessible, but still produces hardware "of comparable quality to a commercial high-level synthesis tool" [Canis et al. 2011]. We also compare against LegUp with different optimisation levels in an effort to understand which optimisations have the biggest impact on the performance discrepancies between LegUp and Vericert. The baseline LegUp version has all the default automatic optimisations turned on. First, we only turn off the LLVM optimisations in LegUp, to eliminate all the optimisations that are common to standard software compilers, referred to as 'LegUp no-opt'. Secondly, we also compare against LegUp with LLVM optimisations and operation chaining turned off, referred to as 'LegUp no-opt no-chaining'. Operation chaining [Paulin and Knight 1989; Venkataramani and Goldstein 2007] is an HLS-specific optimisation that combines data-dependent operations into one clock cycle, and therefore dramatically reduces the number of cycles, without necessarily decreasing the clock speed.

Choice and preparation of benchmarks. We evaluate Vericert using the PolyBench/C benchmark suite (version 4.2.1) [Pouchet 2020], which is a collection of 30 numerical kernels. PolyBench/C is popular in the HLS context [Choi and Cong 2018; Pouchet et al. 2013; Zhao et al. 2017; Zuo et al. 2013], 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/C's parameters so that only integer types are used. We use PolyBench/C'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. We have not modified the benchmarks to make them run through LegUp optimally, e.g. by adding pragmas that trigger more advanced optimisations.

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. Implementing pipelined operators such as the divide and modulus operator can be solved by scheduling the instructions so that these can execute in parallel, which is the main optimisation that needs to be added to Vericert. In the meantime, we have prepared an alternative version of the benchmarks in which each division/modulo operation is replaced with our own implementation that uses repeated division and multiplications by 2. Fig. 9 shows the results of comparing Vericert with optimised LegUp 4.0 on the PolyBench/C benchmarks, where divisions have been left intact. Fig. 10 performs the comparison where the division/modulo operations have been replaced by the iterative algorithm.

Synthesis setup. The Verilog that is generated by Vericert or LegUp is provided to Xilinx Vivado v2017.1 [Xilinx 2019], which synthesises it to a netlist, before placing-and-routing this netlist onto a Xilinx XC7Z020 FPGA device that contains approximately 85000 LUTs.

5.2 RQ1: How Fast is Vericert-Generated Hardware?

Firstly, before comparing any performance metrics, it is worth highlighting that any Verilog produced by Vericert is guaranteed to be *correct*, whilst no such guarantee can be provided by LegUp. This guarantee in itself provides a significant leap in terms of HLS reliability, compared to any other HLS tools available.

The top graphs of Fig. 9 and Fig. 10 compare the execution time of the 27 programs executed by Vericert and the different optimisation levels of LegUp. Each graph uses optimised LegUp as the baseline. 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. The benchmarks with division/modulo replaced show that Vericert actually achieves the same execution speed as LegUp without LLVM optimisations and without operation chaining, 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

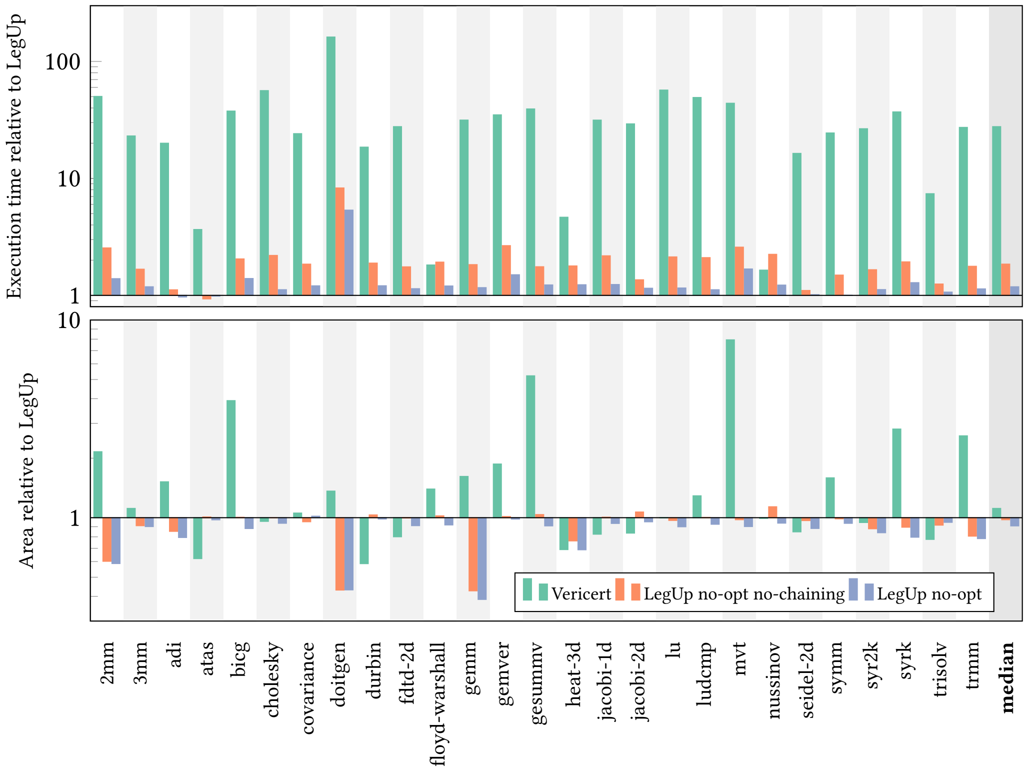


Fig. 9. Performance of Vericert compared to LegUp, with division and modulo operations enabled. The top graph compares the execution times and the bottom graph compares the area of the generated designs. In both cases, the performance of Vericert, LegUp without LLVM optimisations and without operation chaining, and LegUp without LLVM optimisations is compared against default LegUp.

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 an 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 explained in Section 5.1. 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.

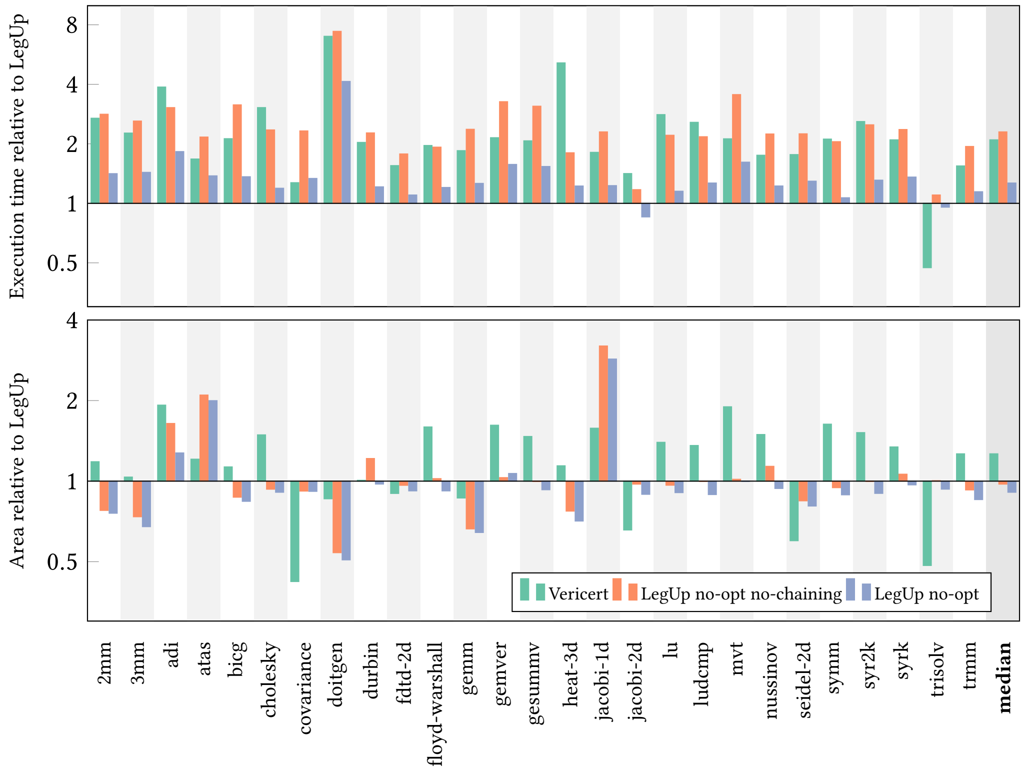


Fig. 10. Performance of Vericert compared to LegUp, with division and modulo operations replaced by an iterative algorithm in software. The top graph compares the execution times and the bottom graph compares the area of the generated designs. In both cases, the performance of Vericert, LegUp without LLVM optimisations and without operation chaining, and LegUp without LLVM optimisations is compared against default LegUp.

Looking at a few benchmarks in particular in Fig. 10 for some interesting cases. For the `trmm` benchmark, Vericert produces hardware that executes with the same cycle count as LegUp, and manages to create hardware that achieves twice the frequency compared to LegUp, thereby actually producing a design that executes twice as fast as LegUp. Another interesting benchmark is `doitgen`, where Vericert is comparable to LegUp without LLVM optimisations, however, LLVM optimisations seem to have a large effect on the cycle count.

5.3 RQ2: How Area-Efficient is Vericert-Generated Hardware?

The bottom graphs in both Fig. 9 and Fig. 10 compare the resource utilisation of the PolyBench/C programs generated by Vericert and LegUp at various optimisation levels. By looking at the median, when division/modulo operations are enabled, we see that Vericert produces hardware that is about the same size as optimised LegUp, whereas the unoptimised versions of LegUp actually produce slightly smaller hardware. This is because optimisations can often increase the size of the hardware to make it faster. Especially in Fig. 9, there are a few benchmarks where the size of the LegUp design is much smaller than that produced by Vericert. This can mostly be explained because of resource sharing in LegUp. Division/modulo operations need large circuits, and it is therefore usual

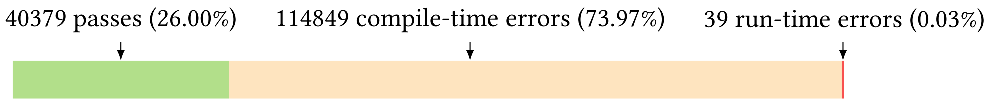


Fig. 11. Results of fuzzing Vericert using 155267 random C programs generated by Csmith.

to only have one circuit per design. As Vericert uses the naïve implementation of division/modulo, there will be multiple circuits present in the design, which blows up the size. Looking at Fig. 10, one can see that without division, the size of Vericert designs are almost always around the same size as LegUp designs, never being more than 2× larger, and sometimes even being smaller. The similarity in area also shows that RAM is correctly being inferred by the synthesis tool, and is therefore not implemented using registers.

5.4 RQ3: How Quickly does Vericert Translate the C into Verilog?

LegUp takes around 10× as long as Vericert to perform the translation from C into Verilog, showing at least that verification does not have a substantial effect on the run-time of the HLS tool. However, this is a meaningless victory, as a lot of the extra time that LegUp uses is on performing computationally heavy optimisations such as loop pipelining and scheduling.

5.5 RQ4: How Effective is the Correctness Theorem in Vericert?

"Beware of bugs in the above code; I have only proved it correct, not tried it."
– D. E. Knuth (1977)

To gain further confidence that the Verilog designs generated by Vericert are actually correct, and that the correctness theorem is indeed effective, we fuzzed Vericert using Csmith [Yang et al. 2011]. Yang et al. previously used Csmith in an extensive fuzzing campaign on CompCert and found a handful of bugs in the unverified parts of that compiler, so it is natural to explore whether it can find bugs in Vericert too. Herklotz et al. [2021a] have recently used Csmith to fuzz other HLS tools including LegUp, so we configured Csmith in a similar way. In addition to the features turned off by Herklotz et al., we turned off the generation of global variables and non-32-bit operations. The generated designs were tested by simulating them and comparing the output value to the results of compiling the test-cases with GCC 10.3.0.

The results of the fuzzing run are shown in Fig. 11. Out of 155267 test-cases generated by Csmith, 26% of them passed, meaning they compiled without error and resulted in the same final value as GCC. Most of the test-cases, 73.97%, failed at compile time. The most common reasons for this were unsigned comparisons between integers (Vericert requires them to be signed), and the presence of 8-bit operations (which Vericert does not support, and which we could not turn off due to a limitation in Csmith). Because the test-cases generated by Csmith could not be tailored exactly to the C fragment that Vericert supports, such a high compile-time failure rate is expected. Finally, and most interestingly, there were a total of 39 run-time failures, which the correctness theorem should be proving impossible. However, all 39 of these failures are due to a bug in the pretty-printing of the final Verilog code, where a logical negation (`!`) was accidentally used instead of a bitwise negation (`~`). Once this bug was fixed, all test-cases passed.

6 RELATED WORK

A summary of the related works can be found in Fig. 12, which is represented as an Euler diagram. The categories chosen for the Euler diagram are: whether the tool is usable, whether it takes a

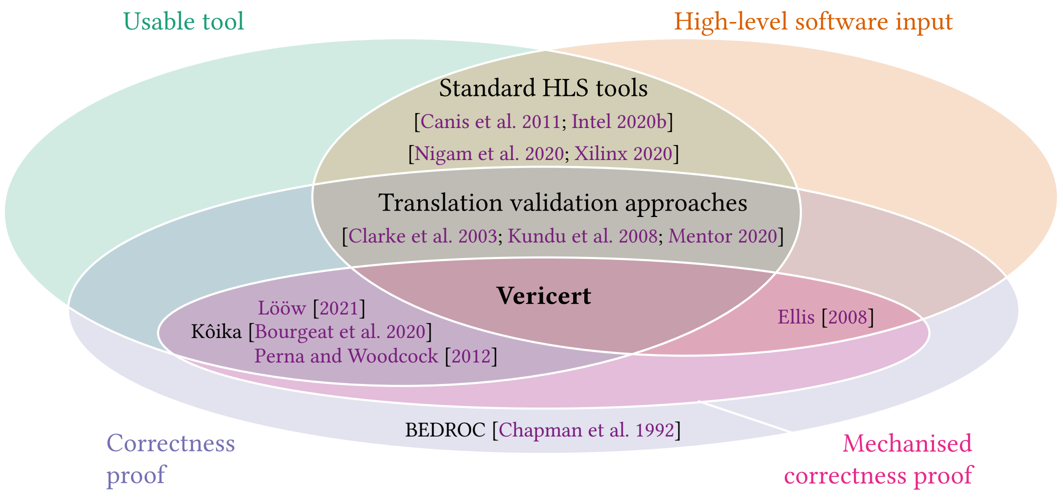

Usable tool
High-level software input
Standard HLS tools
[Canis et al. 2011; Intel 2020b]
[Nigam et al. 2020; Xilinx 2020]
Translation validation approaches
[Clarke et al. 2003; Kundu et al. 2008; Mentor 2020]
Vericert
Lööw [2021]
Kôika [Bourgeat et al. 2020]
Perna and Woodcock [2012]
Ellis [2008]
BEDROC [Chapman et al. 1992]
Correctness proof
Mechanised correctness proof

Fig. 12. Summary of related work

high-level software language as input, whether it has a correctness proof, and finally whether that proof is mechanised. The goal of Vericert is to cover all of these categories.

Most practical HLS tools [Canis et al. 2011; Intel 2020b; Nigam et al. 2020; Xilinx 2020] fit into the category of usable tools that take high-level inputs. On the other end of the spectrum, there are tools such as BEDROC [Chapman et al. 1992] for which there is no practical tool, and even though it is described as high-level synthesis, it more closely resembles today's logic synthesis tools.

Ongoing work in translation validation [Pnueli et al. 1998] 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 [Mentor 2020], which tries to match the states in the hardware 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 [Chouksey and Karfa 2020; Karfa et al. 2006; Youngsik Kim et al. 2004] or code motion [Banerjee et al. 2014; Chouksey et al. 2019], 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 translator 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 [2019] used a proof-producing translator from HOL4 code describing state transitions into Verilog to design a verified processor, which is described further by Lööw et al. [2019]. Lööw [2021] has also worked on formally verifying a logic synthesis tool that can transform hardware descriptions into low-level netlists. This synthesis back end can seamlessly integrate with the proof-producing HOL4 to Verilog translator as it is based on the same Verilog semantics, and therefore creates verified translation from HOL4 circuit descriptions to synthesised Verilog netlists. Perna et al. designed a formally verified translator from a deep embedding of Handel-C [Aubury et al. 1996] into a deep embedding of a circuit [Perna and Woodcock 2012; Perna et al. 2011]. Finally, Ellis [2008] 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 LIMITATIONS AND FUTURE WORK

There are various limitations in Vericert compared to other HLS tools due to the fact that our main focus was on formally verifying the translation from 3AC to Verilog. In this section, we outline the current limitations of our tool and suggest how they can be overcome in future work, first describing limitations to the generated hardware, and then describing the limitations on the software input that Vericert accepts.

7.1 Limitations to the Generated Hardware

Lack of instruction-level parallelism. The main limitation of Vericert is that it does not perform instruction scheduling, which means that instructions cannot be gathered into the same state and executed in parallel. However, each language in Vericert was designed with scheduling in mind, so that these should not have to change fundamentally when that is implemented in the future. For instance, our HTL language already allows arbitrary Verilog to appear in each state of the FSMD; currently, each state just contains a single Verilog assignment, but when scheduling is added, it will contain a list of assignments that can all be executed in parallel. We expect to follow the lead of Tristan and Leroy [2008] and Six et al. [2020], who have previously added scheduling support to CompCert in a VLIW context, by invoking an external (unverified) scheduling tool and then using translation validation to verify that each generated schedule is correct (as opposed to verifying the scheduling tool itself).

Lack of pipelined division. Pipelined operators can execute different stages of an operation in parallel, and thus perform several long-running operations simultaneously while sharing the same hardware. The introduction of pipelined operators to Vericert, especially for division, would alleviate the slow clock speed observed in the PolyBench/C benchmarks when divisions were included (Fig. 9). In HTL, pipelined operations could be represented in a similar fashion to load and store instructions, by using wires to communicate with an abstract computation block modelled in HTL and later replaced by a hardware implementation.

7.2 Limitations on the Software Input

Limitations with I/O. Vericert is currently limited in terms of I/O because the main function cannot accept any arguments if the Clight program is to be well-formed.[6] Moreover, external function calls that produce traces have not been implemented yet, but these could enable the C program to read and write values on a bus that is shared with various other components in the hardware design.

Lack of support for global variables. In CompCert, each global variable is stored in its own memory. A generalisation of our translation of the stack into a RAM block could therefore translate global variables in the same manner. This would require a slight generalisation of pointers so that they store provenance information to ensure that each pointer accesses the right RAM. It would also be necessary to generalise the RAM interface so that it decodes the provenance information and indexes the correct array.

Other language restrictions. 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

[6]Technically, Vericert (and indeed, CompCert) can compile main functions that have arbitrary arguments and will handle those inputs appropriately, but the correctness theorem offers no guarantees about such programs.

bits as signed, both operators have to be forced to be signed. Moreover, 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 to inhibit implicit resizing. Instead, explicit resizing is used in the semantics, and operations can only be performed on two registers that have the same size.

Furthermore, equality checks 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. This is not a severe restriction in practice however, because in the absence of dynamic allocation, equality comparison of pointers is rarely needed, and equality comparison of integers can still be performed by casting them both to signed integers.

Finally, the `mulhs` and `mulhu` instructions, which fetch the upper bits of a 32-bit multiplication, are not translated by Vericert, because 64-bit numbers are not supported. These instructions are only generated to optimise divisions by a constant that is not a power of two, so turning off constant propagation will allow these programs to pass without error.

8 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 [2019]. We evaluated Vericert against the existing LegUp HLS tool on the PolyBench/C benchmark suite. Currently, our hardware is 27× slower and 1.1× larger compared to LegUp, though it is only 2× 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 8.2×. Beyond this, we plan to implement scheduling and loop pipelining, since this allows more operations to be packed into fewer clock cycles. Other optimisations include resource sharing to reduce the circuit area, and using tailored hardware operators that exploit the hard IP blocks of the 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 has been both proven and fuzz-tested; the semantics for the source and target languages are both well established; and Coq is a mature and thoroughly tested system. Nonetheless, Vericert cannot guarantee to provide an output for every valid input – for instance, as remarked in Section 4.5, 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. That concern may be allayed in the future thanks to recent work by Lööw [2021] to produce a verified logic synthesis tool.

ACKNOWLEDGMENTS

We would like to thank Sandrine Blazy, Jianyi Cheng, Alastair Donaldson, Andreas Lööw, and the anonymous reviewers for their helpful feedback. This work was financially supported by the EPSRC via the Research Institute for Verified Trustworthy Software Systems (VeTSS) and the IRIS Programme Grant (EP/R006865/1).

REFERENCES

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

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

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

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2018. CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics. *Journal of Automated Reasoning* 63, 2 (Nov. 2018), 369–392. https://doi.org/10.1007/s10817-018-9496-y

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

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)*. ACM, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965

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

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

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

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

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

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

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

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

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

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

Stephane Gauthier and Zubair Wadood. 2020. High-Level Synthesis: Can it outperform hand-coded HDL? https://info.silexica.com/high-level-synthesis/1 White paper.

David J. Greaves. 2019. Research Note: An Open Source Bluespec Compiler. arXiv:1905.03746 [cs.PL]

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

Yann Herklotz, Zewei Du, Nadesh Ramanathan, and John Wickerson. 2021a. An Empirical Study of the Reliability of High-Level Synthesis Tools. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 219–223. https://doi.org/10.1109/FCCM51124.2021.00034

Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. 2021b. *ymherklotz/vericert: Vericert v1.2.1*. https://doi.org/10.5281/zenodo.5093839

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

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

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

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

Intel. 2020a. High-level Synthesis Compiler. https://intel.ly/2UDiWr5

Intel. 2020b. SDK for OpenCL Applications. https://intel.ly/30sYHz0

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

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

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

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

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

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)*. ACM, New York, NY, USA, 65–76. https://doi.org/10.1145/2737924.2737986

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

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

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

Mentor. 2020. Catapult High-Level Synthesis. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls

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

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)*. ACM, New York, NY, USA, 393–407. https://doi.org/10.1145/3385412.3385974

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

D. H. Noronha, J. P. Pinilla, and S. J. E. Wilton. 2017. Rapid circuit-specific inlining tuning for FPGA high-level synthesis. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–6. https://doi.org/10.1109/RECONFIG.2017.8279807

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.

P. G. Paulin and J. P. Knight. 1989. Scheduling and Binding Algorithms for High-Level Synthesis. In *Proceedings of the 26th ACM/IEEE Design Automation Conference* (Las Vegas, Nevada, USA) *(DAC '89)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/74382.74383

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

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

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

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

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

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

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

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)*. ACM, New York, NY, USA, 258–271. https://doi.org/10.1145/3385412.3386024

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

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

Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. ACM, New York, NY, USA, 17–27. https://doi.org/10.1145/1328438.1328444

Girish Venkataramani and Seth C. Goldstein. 2007. Operation chaining asynchronous pipelined circuits. In *2007 IEEE/ACM International Conference on Computer-Aided Design*. 442–449. https://doi.org/10.1109/ICCAD.2007.4397305

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3, Article 22 (June 2013), 50 pages. https://doi.org/10.1145/2487241.2487248

Yuting Wang, Xiangzhe Xu, Pierre Wilke, and Zhong Shao. 2020. CompCertELF: Verified Separate Compilation of C Programs into ELF Object Files. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 197 (Nov. 2020), 28 pages. https://doi.org/10.1145/3428265

Xilinx. 2019. Vivado Design Suite. https://bit.ly/2wZAmld

Xilinx. 2020. Vivado High-level Synthesis. https://bit.ly/39ereMx

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)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

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

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)*. 430–437. https://doi.org/10.1109/ICCAD.2017.8203809

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 427–440. https://doi.org/10.1145/2103656.2103709

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