

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.

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

1 INTRODUCTION

Can you trust your high-level synthesis tool? As latency, throughput, and energy efficiency become increasingly important, custom hardware accelerators are being designed for numerous applications. Alas, designing these accelerators can be a tedious and error-prone process using a hardware description language (HDL) such as Verilog. An attractive alternative is *high-level synthesis* (HLS), in which hardware designs are automatically compiled from software written in a high-level language like C. Modern HLS tools such as LegUp [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; 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 incorrectly¹ 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)

¹<https://bit.ly/vivado-hls-pipeline-bug>

²<https://bit.ly/vivado-hls-pointer-bug>

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

2018. 2475-1421/2018/1-ART1 \$15.00

<https://doi.org/>

50 OpenCL compiler since it “either crashed or emitted an internal compiler error” on so many of
51 their test inputs. More recently, Herklotz et al. [2021] fuzz-tested three commercial HLS tools
52 using Csmith [Yang et al. 2011], and despite restricting the generated programs to the C fragment
53 explicitly supported by all the tools, they still found that on average 2.5% of test cases were compiled
54 to designs that behaved incorrectly.

55
56 *Existing workarounds.* Aware of the reliability shortcomings of HLS tools, hardware designers
57 routinely check the generated hardware for functional correctness. This is commonly done by
58 simulating the generated design against a large test-bench. But unless the test-bench covers all
59 inputs exhaustively – which is often infeasible – there is a risk that bugs remain.

60 One alternative is to use *translation validation* [Pnueli et al. 1998] to prove equivalence between
61 the input program and the output design. Translation validation has been successfully applied
62 to several HLS optimisations [Banerjee et al. 2014; Chouksey and Karfa 2020; Chouksey et al.
63 2019; Karfa et al. 2006; Youngsik Kim et al. 2004]. Nevertheless, it is an expensive task, especially
64 for large designs, and it must be repeated every time the compiler is invoked. For example, the
65 translation validation for Catapult C [Mentor 2020] may require several rounds of expert ‘adjust-
66 ments’ [Chauhan 2020, p. 3] to the input C program before validation succeeds. And even when it
67 succeeds, translation validation does not provide watertight guarantees unless the validator itself
68 has been mechanically proven correct [e.g. Tristan and Leroy 2008], which has not been the case in
69 HLS tools to date.

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

72
73 *Our solution.* We have designed a new HLS tool in the Coq theorem prover and proved that any
74 output design it produces always has the same behaviour as its input program. Our tool, called
75 Vericert, is automatically extracted to an OCaml program from Coq, which ensures that the object of
76 the proof is the same as the implementation of the tool. Vericert is built by extending the CompCert
77 verified C compiler [Leroy 2009] with a new hardware-specific intermediate language and a Verilog
78 back end. It supports most C constructs, including integer operations, function calls (which are all
79 inlined), local arrays, structs, unions, and general control-flow statements, but currently excludes
80 support for case statements, function pointers, recursive function calls, non-32-bit integers, floats,
81 and global variables.

82
83 *Contributions and Outline.* The contributions of this paper are as follows:

- 84 • We present Vericert, the first mechanically verified HLS tool that compiles C to Verilog. In
85 Section 2, we describe the design of Vericert, including a few optimisations related to memory
86 accesses and division.
- 87 • We state the correctness theorem of Vericert with respect to an existing semantics for Verilog
88 due to Löw and Myreen [2019]. In Section 3, we describe how we extended this semantics to
89 make it suitable as an HLS target. We also describe how the Verilog semantics is integrated
90 into CompCert’s language execution model and its framework for performing simulation
91 proofs. A mapping of CompCert’s infinite memory model onto a finite Verilog array is also
92 described.
- 93 • In Section 4, we describe how we proved the correctness theorem. The proof follows standard
94 CompCert techniques – forward simulations, intermediate specifications, and determinism
95 results – but we encountered several challenges peculiar to our hardware-oriented setting.
96 These include handling discrepancies between the byte-addressed memory assumed by the
97

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 $27\times$ slower ($2\times$ slower in the absence of division) and $1.1\times$ 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.

Vericert is fully open source and available online.

<https://github.com/ymherklotz/vericert>

2 DESIGNING A VERIFIED HLS TOOL

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

Choice of source language. C was chosen as the source language as it remains the most common source language amongst production-quality HLS tools [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 [Löw et al. 2019; Meredith et al. 2010]. Bluespec, previously ruled out as a source language, is another possible target and there exists a formally verified translation to circuits using Kōika [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.

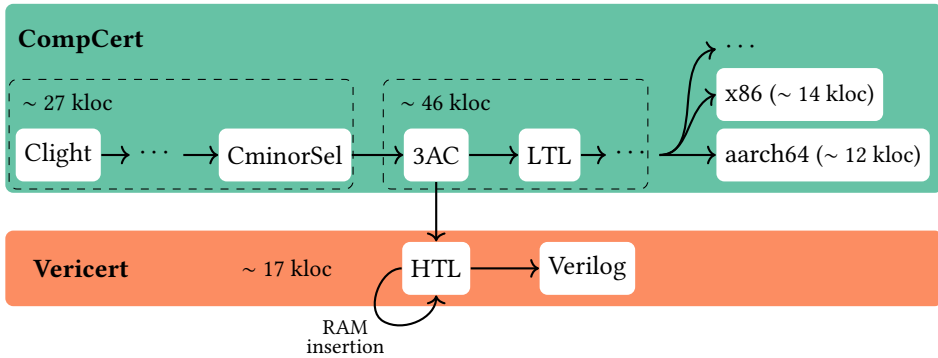


Fig. 1. Vericert as a Verilog back end to CompCert. For scale, the approximate lines of code (kloc) are shown for Vericert, as well as for the front end and back end of CompCert, including any comments and whitespace.

Architecture of Vericert. The main work flow of Vericert is given in Figure 1, which shows those parts of the translation that are performed in CompCert, and those that have been added. 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 Clight³ 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 as a control flow graph (CFG) where each instruction is a node with links to the instructions that can follow it. One difference between LLVM IR and 3AC is that 3AC includes operations that are specific to the chosen target architecture; we chose to target the x86_32 backend because it generally produces relatively dense 3AC thanks to the availability of complex addressing modes.

2.1 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 an integrated circuit, to small application-specific accelerators that are placed on an FPGA. Verilog is a popular language because it allows for fine-grained control over the hardware, and also provides high-level constructs to simplify the development.

³A deterministic subset of C with pure expressions.

⁴This 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.

```

197 1  module main(input rst, input y, input clk,
198 2             output reg z);
199 3  reg x, state;
200 4  always @(posedge clk)
201 5      case (state)
202 6          1'b0: x <= y;
203 7          1'b1: begin x <= 1'b0; z <= x; end
204 8      endcase
205 9  always @(posedge clk)
206 10     if (rst) state <= 1'b0;
207 11     else case (state)
208 12         1'b0: if (y) state <= 1'b1;
209 13              else state <= 1'b0;
210 14         1'b1: state <= 1'b0;
211 15     endcase
212 16 endmodule

```

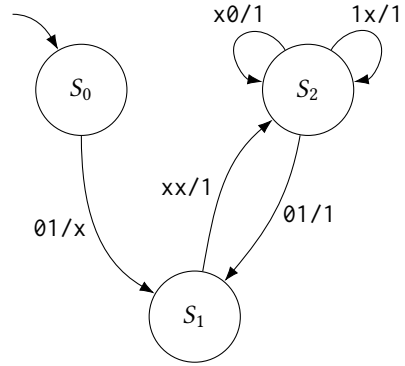


Fig. 2. A simple state machine implemented in Verilog, with its diagrammatic representation on the right, where the x's stand for don't cares and each transition is labelled with the values for the rst, y and output z.

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 Figure 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, x 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 x and then set z to the original value of x, 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 be 0 again.

2.2 Translating C to Verilog, by example

Figure 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.

2.2.1 Translating C to 3AC. The first stage of the translation uses unmodified CompCert to transform the C input, shown in Figure 3a, into a 3AC intermediate representation, shown in Figure 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 other HLS tools either [Thomas 2016].

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

274

275

276

277

278

279

280

281

282

283

284

285

286

287

288

289

290

291

292

293

294

(a) Example C code passed to Vericert.

```

1  main() {
2    x5 = 3
3    int32[stack(0)] = x5
4    x4 = 6
5    int32[stack(4)] = x4
6    x1 = 1
7    x3 = stack(0) (int)
8    x2 = int32[x3 + x1
9          * 4 + 0]
10   return x2
11 }

```

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

```

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.

2.2.2 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 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. Figure 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 make it easier to prove the 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 represent what to execute at every state, and similar semantics to Verilog at the low 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.

295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343

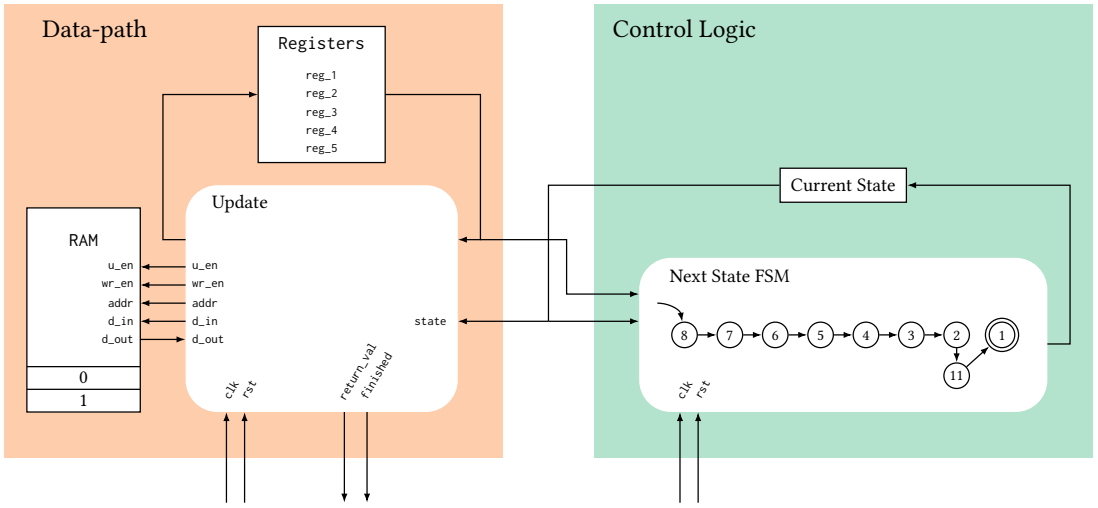


Fig. 4. The FSMD for the example shown in Figure 3, split into a data-path and control logic for the next state calculation. The Update block takes the current state, current values of all registers and at most one value stored in the RAM, and calculates a new value that can either be stored back in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

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 Figure 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 Figure 3c. A high-level overview of the architecture and of the RAM interface can be seen in Figure 4.

Translating instructions. Most 3AC instructions correspond to hardware constructs. For example, line 2 in Figure 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 Figure 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 is the Oshrximm instruction, which is discussed further in Section 2.3.3. 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 those related to function calls (Icall, Ibuiltin, and Itailcall), because we enable inlining by default.

344 2.2.3 *Translating HTL to Verilog.* Finally, we have to translate the HTL code into proper Verilog.
345 The challenge here is to translate our FSMMD representation into a Verilog AST. However, as all
346 the instructions in HTL are already expressed as Verilog statements, only the top-level data-path
347 and control logic maps need to be translated to valid Verilog case-statements. We also require
348 declarations for all the variables in the program, as well as declarations of the inputs and outputs
349 to the module, so that the module can be used inside a larger hardware design. In addition to
350 translating the maps of Verilog statements, an always-block that will behave like the RAM also has
351 to be created, which is only modelled abstractly at the HTL level. Figure 3c shows the final Verilog
352 output that is generated for our example.

353 Although this translation seems quite straightforward, proving that this translation is correct is
354 complex. All the implicit assumptions that were made in HTL need to be translated explicitly to
355 Verilog statements and it needs to be shown that these explicit behaviours are equivalent to the
356 assumptions made in the HTL semantics. One main example of this is proving that specification
357 of the RAM in HTL does indeed behave the same as its Verilog implementation. We discuss these
358 proofs in upcoming sections.

359

360

361 2.3 Optimisations

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

365

366

367 2.3.1 *Byte- and word-addressable memories.* One big difference between C and Verilog is how
368 memory is represented. Although Verilog arrays use similar syntax to C arrays, they must be
369 treated quite differently. To make loads and stores as efficient as possible, the RAM needs to be
370 word-addressable, which means that an entire integer can be loaded or stored in one clock cycle.
371 However, the memory model that CompCert uses for its intermediate languages is byte-address-
372 sable [Blazy and Leroy 2005]. If a byte-addressable memory was used in the target hardware,
373 which is closer to CompCert’s memory model, then a load and store would instead take four clock
374 cycles, because a RAM can only perform one read and write per clock cycle. It therefore has to
375 be proven that the byte-addressable memory behaves in the same way as the word-addressable
376 memory in hardware. Any modifications of the bytes in the CompCert memory model also have to
377 be shown to modify the word-addressable memory in the same way. Since only integer loads and
378 stores are currently supported in Vericert, it follows that the addresses given to the loads and stores
379 will be multiples of four. If that is the case, then the translation from byte-addressed memory to
380 word-addressed memory can be done by dividing the address by four.

381

382

383 2.3.2 *Implementation of RAM interface.* The simplest way to implement loads and stores in Vericert
384 would be to access the Verilog array directly from within the data-path (i.e., inside the always-block
385 on lines 16–32 of Figure 3c). This would be correct, but when a Verilog array is accessed at several
386 program points, the synthesis tool is unlikely to detect that it can be implemented as a RAM block,
387 and will resort to using lots of registers instead, ruining the circuit’s area and performance. To avert
388 this, we arrange that the data-path does not access memory directly, but simply sets the address it
389 wishes to access and then toggles the `u_en` flag. This activates the RAM interface (lines 9–15 of
390 Figure 3c) on the next falling clock edge, which performs the requested load or store. By factoring
391 all the memory accesses out into a separate interface like this, we ensure that the underlying array

392

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 ($en \neq u_en$) 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 to not have to reason 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.

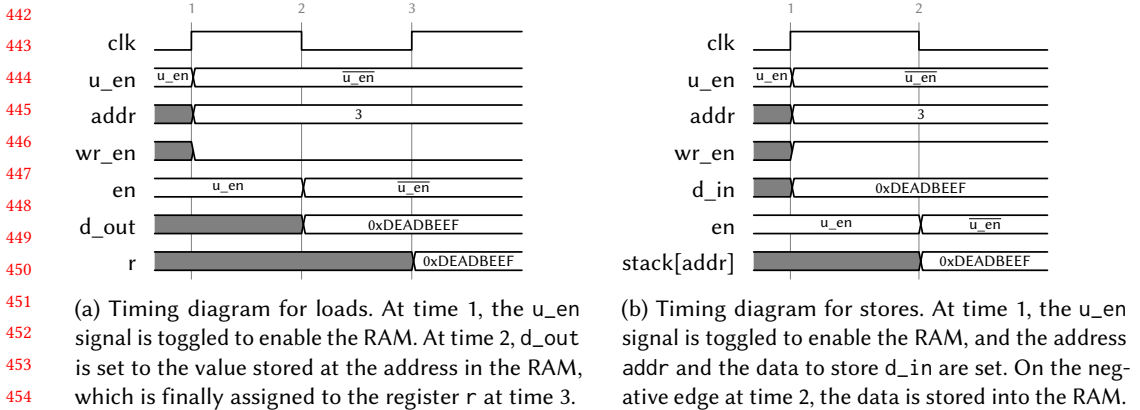
Figure 5 gives an example of how the RAM interface behaves when values are loaded and stored.

2.3.3 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 naively. The problem is that in CompCert it is specified as a signed division:

$$\text{Oshrximm } x \ y = \text{round_towards_zero} \left(\frac{x}{2^y} \right)$$

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

⁵Interestingly, 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.



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

457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490

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 Löow and Myreen [2019] and used to prove the translation from HOL4 to Verilog [Löow 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:

$$\frac{\text{ALWAYS} \quad (\Sigma, s) \downarrow_{\text{stmt}} \Sigma'}{(\Sigma, \text{always } @(posedge \text{clk}) \ s) \downarrow_{\text{always}^+} \Sigma'}$$

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:

$$\frac{\text{NONBLOCKING REG} \quad \text{name } d = \text{OK } n \quad (\Gamma, e) \downarrow_{\text{expr}} v}{((\Gamma, \Delta), d \leftarrow e) \downarrow_{\text{stmt}} (\Gamma, \Delta[n \mapsto v])}$$

where assuming that \downarrow_{expr} evaluates an expression e to a value v , the nonblocking assignment $d \leftarrow e$ 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:

$$\frac{\text{MODULE} \quad (\Gamma, \epsilon, \vec{m}) \downarrow_{\text{module}} (\Gamma', \Delta')}{(\Gamma, \text{module main}(\dots); \vec{m} \text{ endmodule}) \downarrow_{\text{program}} (\Gamma' // \Delta')}$$

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

3.1 Changes to the Semantics

Five changes were made to the semantics proposed by Löw and Myreen [2019] to make it suitable as a HLS target.

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

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

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

$$\begin{array}{c}
\text{540} \\
\text{541} \\
\text{542} \\
\text{543} \\
\text{544} \\
\text{545} \\
\text{546} \\
\text{547} \\
\text{548} \\
\text{549} \\
\text{550} \\
\text{551} \\
\text{552} \\
\text{553} \\
\text{554} \\
\text{555} \\
\text{556} \\
\text{557} \\
\text{558} \\
\text{559} \\
\text{560} \\
\text{561} \\
\text{562} \\
\text{563} \\
\text{564} \\
\text{565} \\
\text{566} \\
\text{567} \\
\text{568} \\
\text{569} \\
\text{570} \\
\text{571} \\
\text{572} \\
\text{573} \\
\text{574} \\
\text{575} \\
\text{576} \\
\text{577} \\
\text{578} \\
\text{579} \\
\text{580} \\
\text{581} \\
\text{582} \\
\text{583} \\
\text{584} \\
\text{585} \\
\text{586} \\
\text{587} \\
\text{588}
\end{array}$$

$$\begin{array}{c}
\text{STEP} \\
\frac{\Gamma_r[\text{rst}] = 0 \quad \Gamma_r[\text{fin}] = 0 \quad (m, (\Gamma_r, \Gamma_a)) \downarrow_{\text{program}} (\Gamma'_r, \Gamma'_a)}{\text{State } sf \ m \ \Gamma_r[\sigma] \ \Gamma_r \ \Gamma_a \longrightarrow \text{State } sf \ m \ \Gamma'_r[\sigma] \ \Gamma'_r \ \Gamma'_a} \\
\text{FINISH} \\
\frac{\Gamma_r[\text{fin}] = 1}{\text{State } sf \ m \ \sigma \ \Gamma_r \ \Gamma_a \longrightarrow \text{Returnstate } sf \ \Gamma_r[\text{ret}]} \\
\text{CALL} \\
\frac{\text{Callstate } sf \ m \ \vec{r} \longrightarrow \text{State } sf \ m \ n \ ((\text{init_params } \vec{r} \ a)[\sigma \mapsto n, \text{fin} \mapsto 0, \text{rst} \mapsto 0]) \ \epsilon}{\text{Returnstate } (\text{Stackframe } r \ m \ pc \ \Gamma_r \ \Gamma_a :: sf) \ v \longrightarrow \text{State } sf \ m \ pc \ (\Gamma_r[\sigma \mapsto pc, r \mapsto v]) \ \Gamma_a} \\
\text{RETURN}
\end{array}$$

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

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 ??), support for negative-edge-triggered always-blocks was added to the semantics. This is shown in the modifications of the MODULE rule shown below:

$$\begin{array}{c}
\text{MODULE} \\
\frac{(\Gamma, \epsilon, \vec{m}) \downarrow_{\text{module}^+} (\Gamma', \Delta') \quad (\Gamma' // \Delta', \epsilon, \vec{m}) \downarrow_{\text{module}^-} (\Gamma'', \Delta'')}{(\Gamma, \text{module } \text{main}(\dots); \vec{m} \text{ endmodule}) \downarrow_{\text{program}} (\Gamma'' // \Delta'')}
\end{array}$$

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} .

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 .

Figure 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 Figure 6, an initial state and final state need to be defined:

$$\frac{\text{INITIAL} \quad \text{is_internal}(P.\text{main})}{\text{initial_state}(\text{Callstate } [] (P.\text{main}) [])} \quad \frac{\text{FINAL}}{\text{final_state}(\text{Returnstate } [] n)}$$

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

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

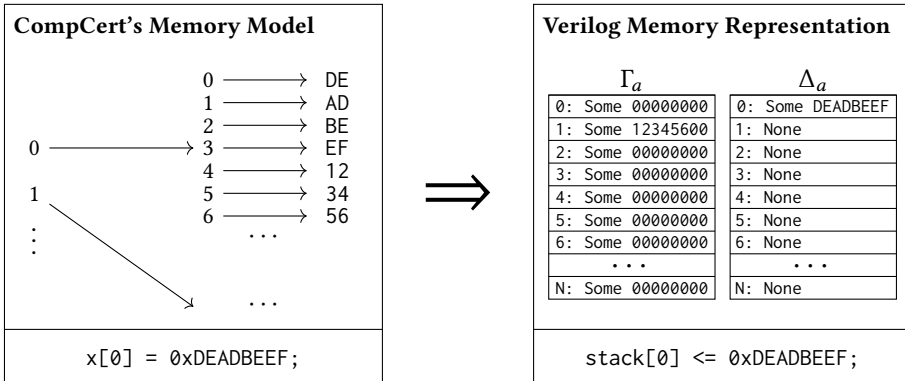


Fig. 7. Change in the memory model during the translation of 3AC into HTL. The state of the memories in each case is right after the execution of the store to memory.

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 compiler passes in CompCert, such as CompCertS [Besson et al. 2018], or to translate to a more concrete finite memory model such as in CompCertELF [Wang et al. 2020] and CompCertTSO [Ševčík et al. 2013], however, we define the translation from CompCert’s standard infinite memory model to finite arrays that can be represented in Verilog, leaving the compiler passes intact. There is therefore no more memory model in Verilog, and all the interactions to memory are encoded in the hardware itself.

This translation is represented in Figure 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 PROOF

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

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 Verilog semantics compared to the standard CompCert intermediate languages and because of the translation of the memory model.

- 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 both 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, however, 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, and C has safe observable behaviour B when executed, then the target Verilog code will have the same behaviour B . Here, a 'safe' execution is one that either converges or diverges, but does not 'go wrong'. If the program does admit some wrong behaviour (such as undefined behaviour in C), the correctness theorem does not apply. A behaviour, then, is either a final state (in the case of convergence) or divergence. In CompCert, a behaviour is also associated with a trace of I/O events, but since external function calls are not supported in Vericert, this trace will always be empty.

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

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

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 likely not help with proving that the hardware is actually working correctly, 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 and would therefore need 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 Figure 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, as instead of defining small-step semantics for each construct in Verilog, the semantics are instead defined over one clock cycle and mirror the semantics defined for Verilog. Lemma 1 shows the result that needs to be proven in this subsection.

LEMMA 1 (FORWARD SIMULATION FROM 3AC TO HTL). *We write tr_htl for the translation from 3AC to HTL.*

$$\forall c, h, B \in \text{Safe}, \text{tr_htl}(c) = \text{OK}(h) \wedge c \Downarrow B \implies h \Downarrow B.$$

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

4.3.1 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:

$$\text{spec_instr } \text{fin } \text{ret } \sigma \text{ stk } i \text{ data } \text{control}$$

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:

$$\frac{\text{IOP} \quad \text{tr_op } \text{op } \vec{a} = \text{OK } e}{\text{spec_instr } \text{fin } \text{ret } \sigma \text{ stk } (\text{Iop } \text{op } \vec{a} \text{ } d \text{ } n) (d \leftarrow e) (\sigma \leftarrow n)}$$

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.*

$$\forall c \text{ } h, \text{tr_htl}(c) = \text{OK}(h) \implies \text{spec_htl } c \text{ } h.$$

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 fn are not modified and therefore stay at a constant 0 throughout execution, and
- that the stack frames match.

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

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

$$\begin{array}{ccc}
 R, M, pc & \xrightarrow{\mathcal{I} \wedge (R \leq \Gamma_r) \wedge (M \leq \Gamma_a) \wedge (pc = \Gamma_r[\sigma])} & \Gamma_r, \Gamma_a \\
 \downarrow & & \downarrow + \\
 R', M', pc' & \xrightarrow{\text{-----} \mathcal{I} \wedge (R' \leq \Gamma'_r) \wedge (M' \leq \Gamma'_a) \wedge (pc' = \Gamma'_r[\sigma]) \text{-----}} & \Gamma'_r, \Gamma'_a
 \end{array}$$

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. \square

4.4 Forward simulation of RAM insertion

HTL can only represent a single state machine, it is therefore necessary to 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 used to model the RAM is shown in Figure 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 algorithm. The three possibilities of the transformation is that for each

$$\begin{array}{c}
\text{IDLE} \\
\frac{\Gamma_r[r.en] = \Gamma_r[r.u_en]}{((\Gamma_r, \Gamma_a), \Delta, r) \downarrow_{\text{ram}} \Delta} \\
\text{LOAD} \\
\frac{\Gamma_r[r.en] \neq \Gamma_r[r.u_en] \quad \Gamma_r[r.wr_en] = 0}{((\Gamma_r, \Gamma_a), (\Delta_r, \Delta_a), r) \downarrow_{\text{ram}} (\Delta_r[r.en \mapsto r.u_en, r.d_out \mapsto (\Gamma_a[r.mem])][r.addr]], \Delta_a)} \\
\text{STORE} \\
\frac{\Gamma_r[r.en] \neq \Gamma_r[r.u_en] \quad \Gamma_r[r.wr_en] = 1}{((\Gamma_r, \Gamma_a), (\Delta_r, \Delta_a), r) \downarrow_{\text{ram}} (\Delta_r[r.en \mapsto r.u_en], \Delta_a[r.mem \mapsto (\Gamma_a[r.mem])][r.addr \mapsto r.d_in])]}
\end{array}$$

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

Verilog statement in the map at location i , either the statement is a load or a store, in which case it is translated to the equivalent signal representation, otherwise it is not changed. An example of the specification for 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.

$$\begin{array}{c}
\text{STORE SPEC} \\
\frac{d[i] = (r.mem[e_1] \leq e_2) \\
t = (r.u_en \leq \neg r.u_en; r.wr_en \leq 1; r.d_in \leq e_2; r.addr \leq e_1)}{\text{spec_ram } \sigma \ r \ d \ c \ d[i \mapsto t] \ c \ i \ n}
\end{array}$$

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 the simulation proof shown in Lemma 3, some invariants need to be defined, which always hold at the start of the simulation and at the end of the simulation. The invariants needed for the simulation of the RAM insertion are quite different to the previous invariants, 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 of the simulation, and at the end of the simulation step. This is the reason for why the self-disabling RAM was 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.*

$$\forall h, B \in \text{Safe}, \text{tr_ram}(h) = h' \wedge h \Downarrow B \implies h' \Downarrow B.$$

	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

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

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 are quite different to the semantics of the case-statement they are converted to.

LEMMA 5 (FORWARD SIMULATION FROM HTL TO VERILOG). *We write tr_verilog for the translation from HTL to Verilog. (Note that this translation cannot fail, so we do not need the `OK` constructor here.)*

$$\forall h, V, B \in \text{Safe}, \text{tr_verilog}(h) = V \wedge h \Downarrow B \implies V \Downarrow B.$$

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

4.6 Deterministic Semantics

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

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

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

PROOF SKETCH. The Verilog semantics is deterministic because the order of operation of all the constructs is defined, and there is therefore only one way to evaluate the module and hence only one possible behaviour. This was proven for the small-step semantics shown in Figure 6. \square

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, whereas there is a separate ‘pointer’ value in the CompCert semantics, it was painful to reason about them and many new theorems had to be proven about integers and pointers in Vericert. 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 assignments. As the negative edge implies two merges take place every clock cycle, the proofs about the equality of the arrays becomes more tedious as well.

Looking at the trusted computing base of Vericert, the Verilog semantics are 431 lines of code. This, together with the Clight semantics from CompCert, are the only parts of the compiler that need to be trusted. Compared to the 1721 lines of the implementation that are written in Coq, which are the verified parts of the HLS tool, this is larger than the 431 lines of Verilog semantics specification, even if the Clight semantics are added. In addition to that, reading semantics specifications is simpler than trying to understand algorithms, meaning 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, grammschmidt 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

981 benchmarks to make them run through LegUp optimally, e.g. by adding pragmas that trigger more
982 advanced optimisations.

983 Vericert implements divisions and modulo operations in C using the corresponding built-in
984 Verilog operators. These built-in operators are designed to complete within a single clock cycle,
985 and this causes substantial penalties in clock frequency. Other HLS tools, including LegUp, supply
986 their own multi-cycle division/modulo implementations, and we plan to do the same in future
987 versions of Vericert. Implementing pipelined operators such as the divide and modulus operator
988 can be solved by scheduling the instructions so that these can execute in parallel, which is the main
989 optimisation that needs to be added to Vericert. In the meantime, we have prepared an alternative
990 version of the benchmarks in which each division/modulo operation is replaced with our own
991 implementation that uses repeated division and multiplications by 2. Figure 9 shows the results of
992 comparing Vericert with optimised LegUp 4.0 on the PolyBench/C benchmarks, where divisions
993 have been left intact. Figure 10 performs the comparison where the division/modulo operations
994 have been replaced by the iterative algorithm.

995
996 *Synthesis setup.* The Verilog that is generated by Vericert or LegUp is provided to Xilinx Vivado
997 v2017.1 [Xilinx 2019], which synthesises it to a netlist, before placing-and-routing this netlist onto
998 a Xilinx XC7Z020 FPGA device that contains approximately 85000 LUTs.

1000 5.2 RQ1: How fast is Vericert-generated hardware?

1001 Firstly, before comparing any performance metrics, it is worth highlighting that any Verilog
1002 produced by Vericert is guaranteed to be *correct*, whilst no such guarantee can be provided by
1003 LegUp. This guarantee in itself provides a significant leap in terms of HLS reliability, compared to
1004 any other HLS tools available.

1005 The top graphs of Figure 9 and Figure 10 compare the execution time of the 27 programs executed
1006 by Vericert and the different optimisation levels of LegUp. Each graph uses optimised LegUp as the
1007 baseline. When division/modulo operations are present LegUp designs execute around 27× faster
1008 than Vericert designs. However, when division/modulo operations are replaced by the iterative
1009 algorithm, LegUp designs are only 2× faster than Vericert designs. However, the benchmarks with
1010 division/modulo replaced show that Vericert actually achieves the same execution speed as LegUp
1011 without LLVM optimisations and without operation chaining, which is encouraging, and shows
1012 that the hardware generation is following the right steps. The execution time is calculated by
1013 multiplying the maximum frequency that the FPGA can run at with this design, by the number of
1014 clock cycles that are needed to complete the execution. We can therefore analyse each separately.

1015 First, looking at the difference in clock cycles, Vericert produces designs that have around
1016 4.5× as many clock cycles as LegUp designs in both cases, when division/modulo operations are
1017 enabled as well as when they are replaced. This performance gap can be explained in part by
1018 LLVM optimisations, which seem to account for a 2× decrease in clock cycles, as well as operation
1019 chaining, which decreases the clock cycles by another 2×. The rest of the speed-up is mostly due
1020 to LegUp optimisations such as scheduling and memory analysis, which are designed to extract
1021 parallelism from input programs. This gap does not represent the performance cost that comes with
1022 formally proving a HLS tool. Instead, it is simply a gap between an unoptimised Vericert versus an
1023 optimised LegUp. As we improve Vericert by incorporating further optimisations, this gap should
1024 reduce whilst preserving the correctness guarantees.

1025 Secondly, looking at the maximum clock frequency that each design can achieve, LegUp designs
1026 achieve 8.2× the maximum clock frequency of Vericert when division/modulo operations are
1027 present. This is in great contrast to the maximum clock frequency that Vericert can achieve when
1028 no divide/modulo operations are present, where Vericert generates designs that are actually 2×

1029

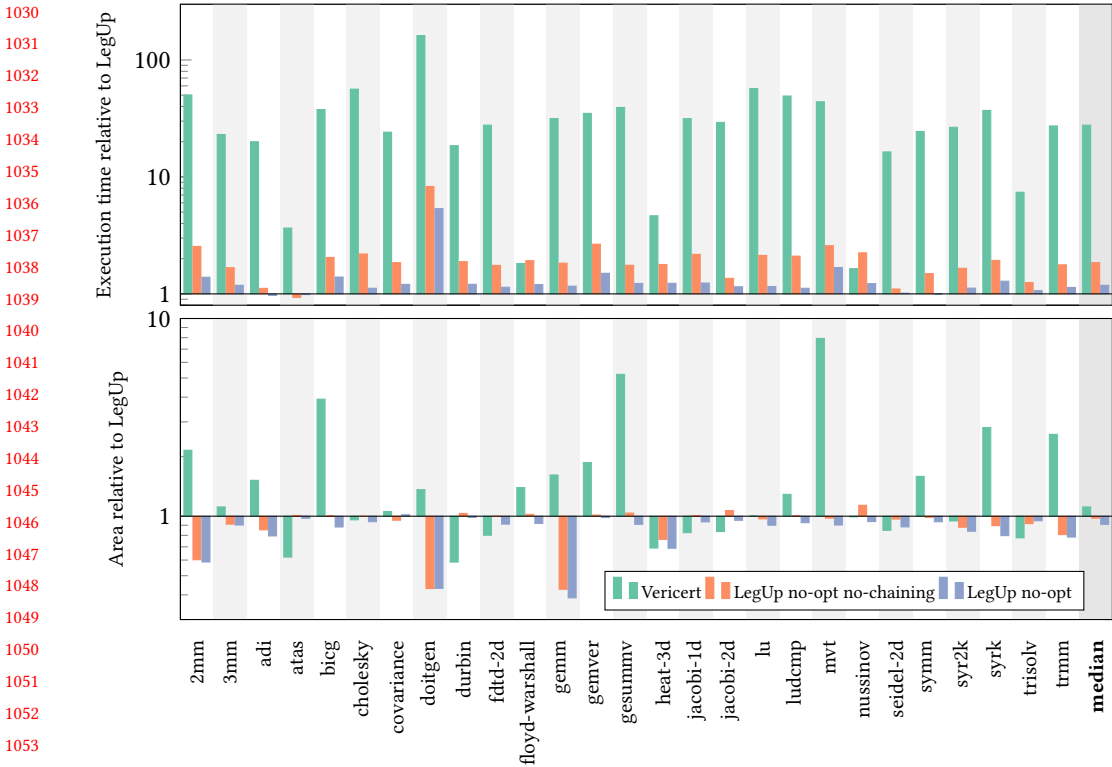


Fig. 9. Performance of Vericert compared to LegUp, with division and modulo operations enabled. The top graph compares the execution times and the bottom graph compares the area of the generated designs. In both cases, the performance of Vericert, LegUp without LLVM optimisations and without operation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

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.

Looking at a few benchmarks in particular in Figure 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 `doigen`, where Vericert is comparable to LegUp without LLVM optimisations, however, LLVM optimisations seem to have a large effect on the cycle count.

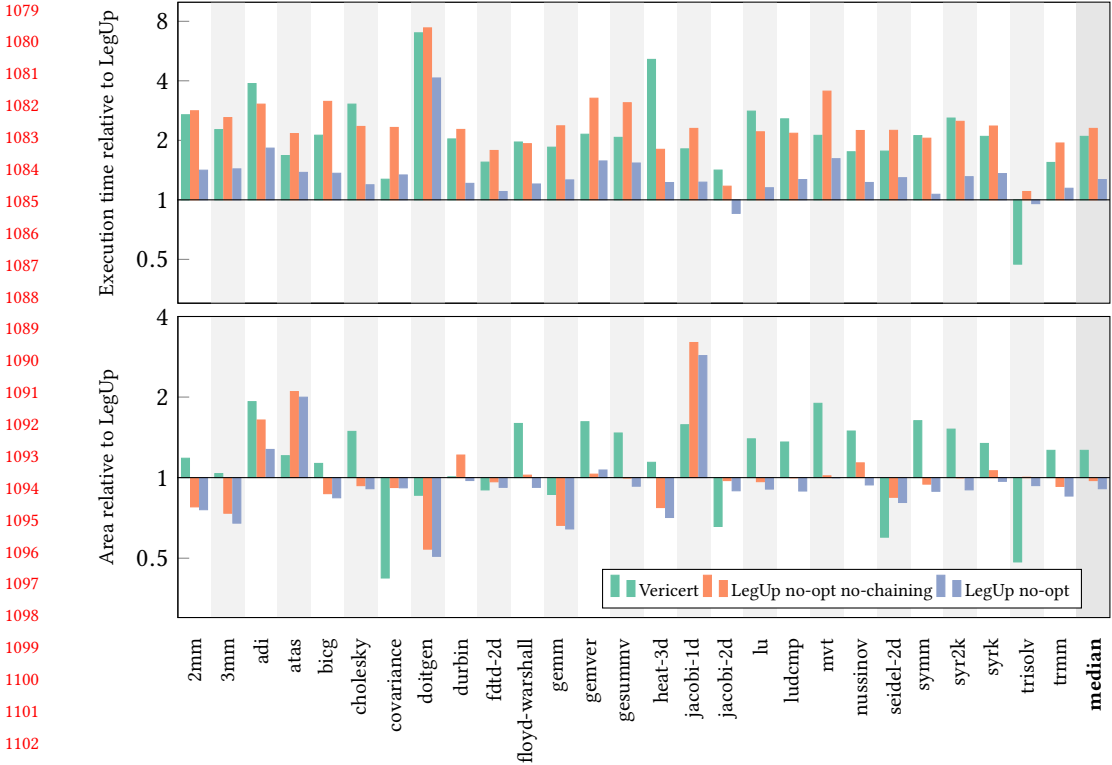


Fig. 10. Performance of Vericert compared to LegUp, with division and modulo operations replaced by an iterative algorithm in software. The top graph compares the execution times and the bottom graph compares the area of the generated designs. In both cases, the performance of Vericert, LegUp without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

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

The bottom graphs in both Figure 9 and Figure 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 Figure 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 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 Figure 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 as registers.

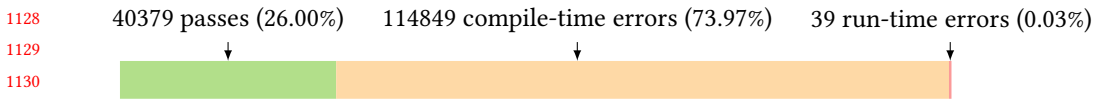


Fig. 11. Results of fuzzing Vericert using 155267 random C programs generated by Csmith.

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. [2021] 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 Figure 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 not unexpected. 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 Figure 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 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

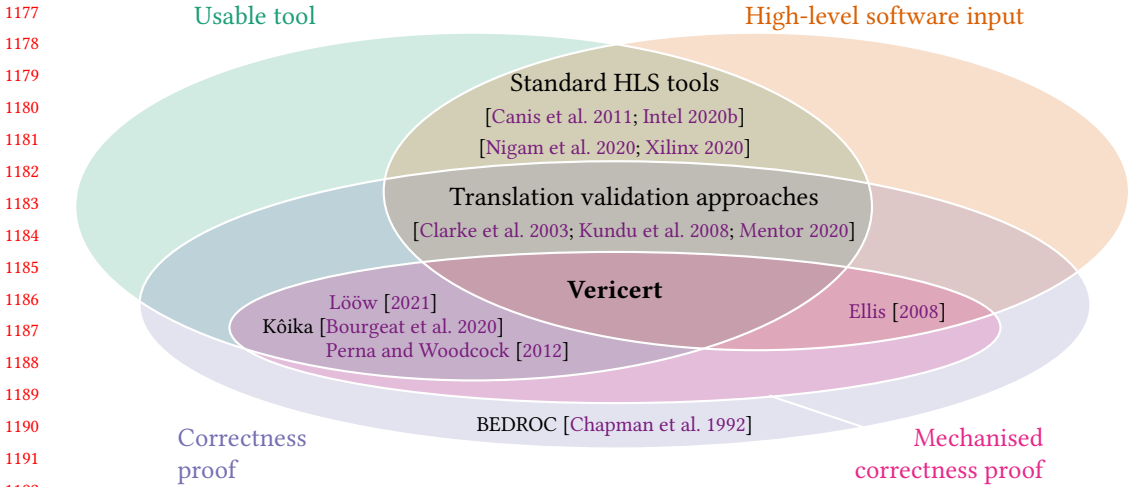


Fig. 12. Summary of related work

of a tool that implements this is Mentor’s Catapult [Mentor 2020], which tries to match the states in the 3AC description to states in the original C code after an unverified translation. Using translation validation is quite effective for verifying complex optimisations such as scheduling [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 translation from a core fragment of BlueSpec into a circuit representation which can then be printed as a Verilog design. This is a translation from a high-level hardware description language into an equivalent circuit representation, so is a different approach to HLS. Löw and Myreen [2019] used a verified translation from HOL4 code describing state transitions into Verilog to design a verified processor, which is described by Löw et al. [2019]. Löw [2021] has also worked on formally verifying a logic synthesis tool that can transform hardware descriptions into low-level netlists. His approach translates a shallow embedding in HOL4 into a deep embedding of Verilog. Perna et al. designed a formally verified translation 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 of the software input that Vericert accepts.

7.1 Limitations and improvements to the hardware

This section describes the current limitations and possible improvements that could be done 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 it is implemented in the future. For instance, our HTL language already allows arbitrary Verilog to appear in each state of the FSM; 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 with divisions 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 and improvements to the software input

This section describes the limitations and possible improvements to the software input accepted by Vericert.

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. However, just like CompCert, Vericert can actually compile main functions that have arbitrary arguments and will handle the inputs appropriately. Still, the main correctness theorem in CompCert assumes that the main function does not have any arguments, so it may be possible that unexpected behaviour is introduced. Moreover, external function calls that produce traces have not been implemented yet, but once they have, they 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 the stack translation into a RAM block could therefore be performed to 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 right 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 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

1275 implicit resizing. Instead, explicit resizing is used in the semantics, and operations can only be
1276 performed on two registers that have the same size.

1277 Furthermore, equality checks between *unsigned* variables are actually not supported, because this
1278 requires supporting the comparison of pointers, which should only be performed between pointers
1279 with the same provenance. In Vericert there is currently no way to determine the provenance of a
1280 pointer, and it therefore cannot model the semantics of unsigned comparison in CompCert. This is
1281 not a severe restriction in practice however, because, since dynamic allocation is not supported
1282 either, equality comparison of pointers is rarely needed, and equality comparison of integers can
1283 still be performed by casting them both to signed integers.

1284 Finally, the `mulhs` and `mulhu` instructions, which fetch the upper bits of a 32-bit multiplication,
1285 are not translated by Vericert, because 64-bit numbers are not supported. These instructions are
1286 only generated to optimise divisions by a constant that is not a power of two, so turning off constant
1287 propagation will allow these programs to pass without error.

1288

1289 8 CONCLUSION

1290 We have presented Vericert, the first mechanically verified HLS tool for translating software in C
1291 into hardware in Verilog. We built Vericert by extending CompCert with a new hardware-specific
1292 intermediate language and a Verilog back end, and we verified it with respect to a semantics for
1293 Verilog due to Löw and Myreen [2019]. We evaluated Vericert against the existing LegUp HLS
1294 tool on the PolyBench/C benchmark suite. Currently, our hardware is 27× slower and 1.1× larger
1295 compared to LegUp, though it is only 2× slower if inefficient divisions are removed.

1296 There are abundant opportunities for improving Vericert’s performance. For instance, as discussed
1297 in Section 5, simply replacing the naïve single-cycle division and modulo operations with C
1298 implementations increases clock frequency by 8.2×. Beyond this, we plan to implement scheduling
1299 and loop pipelining, since this allows more operations to be packed into fewer clock cycles. Other
1300 optimisations include resource sharing to reduce the circuit area, and using tailored hardware
1301 operators that use hard IP blocks on chip and can be pipelined.

1302 Finally, it’s worth considering how trustworthy Vericert is compared to other HLS tools. The
1303 guarantee of full functional equivalence between input and output that Vericert provides is a strong
1304 one, the semantics for the source and target languages are both well-established, and Coq is a
1305 mature and thoroughly tested system. However, Vericert cannot guarantee to provide an output
1306 for every valid input – for instance, as remarked in Section 4.5, Vericert will error out if given a
1307 program with more than about four million instructions! – but our evaluation indicates that it
1308 does not seem to error out too frequently. And of course, Vericert cannot guarantee that the final
1309 hardware produced will be correct, because the Verilog it generates must pass through a series of
1310 unverified tools along the way. This concern may be allayed in the future thanks to recent work
1311 by Löw [2021] to produce a verified logic synthesis tool.

1312

1313 REFERENCES

- 1314 Matthew Aubury, Ian Page, Geoff Randall, Jonathan Saul, and Robin Watts. 1996. Handel-C language reference guide.
1315 *Computing Laboratory. Oxford University, UK* (1996). 25
- 1316 Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste
1317 Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference*
1318 *2012*. IEEE, 1212–1221. <https://doi.org/10.1145/2228360.2228584> 3
- 1319 K. Banerjee, C. Karfa, D. Sarkar, and C. Mandal. 2014. Verification of Code Motion Techniques Using Value Propagation.
1320 *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> 2, 25
- 1321 Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2018. CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as
1322 Integer Semantics. *Journal of Automated Reasoning* 63, 2 (Nov. 2018), 369–392. <https://doi.org/10.1007/s10817-018-9496-y>

1323

1324 14

- 1325 Sandrine Blazy and Xavier Leroy. 2005. Formal Verification of a Memory Model for C-Like Imperative Languages. In
 1326 *Formal Methods and Software Engineering*, Kung-Kiu Lau and Richard Banach (Eds.). Springer Berlin Heidelberg, Berlin,
 1327 Heidelberg, 280–299. https://doi.org/0.1007/11576280_20 8
- 1328 Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for
 1329 Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and*
 1330 *Implementation* (London, UK) (*PLDI 2020*). ACM, New York, NY, USA, 243–257. <https://doi.org/10.1145/3385412.3385965>
 1331 3, 25
- 1332 Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Helge Anderson, Stephen Dean
 1333 Brown, and Tomasz S. Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In
 1334 *FPGA*. ACM, 33–36. <https://doi.org/10.1145/1950413.1950423> 1, 3, 20, 24, 25
- 1335 R. Chapman, G. Brown, and M. Leeser. 1992. Verified high-level synthesis in BEDROC. In [1992] *Proceedings The European*
 1336 *Conference on Design Automation*. IEEE Computer Society, 59–63. <https://doi.org/10.1109/EDAC.1992.205894> 24, 25
- 1337 Pankaj Chauhan. 2020. Formally Ensuring Equivalence between C++ and RTL designs. <https://bit.ly/2KbT0ki> 2
- 1338 Y. Choi and J. Cong. 2018. HLS-Based Optimization and Design Space Exploration for Applications with Variable Loop
 1339 Bounds. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. <https://doi.org/10.1145/3240765.3240815> 20
- 1340 R. Chouksey and C. Karfa. 2020. Verification of Scheduling of Conditional Behaviors in High-Level Synthesis. *IEEE*
 1341 *Transactions on Very Large Scale Integration (VLSI) Systems* (2020), 1–14. <https://doi.org/10.1109/TVLSI.2020.2978242> 2,
 1342 25
- 1343 R. Chouksey, C. Karfa, and P. Bhaduri. 2019. Translation Validation of Code Motion Transformations Involving Loops.
 1344 *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> 2, 25
- 1345 E. Clarke, D. Kroening, and K. Yorav. 2003. Behavioral consistency of C and Verilog programs using bounded model checking.
 1346 In *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*. 368–371. <https://doi.org/10.1145/775832.775928> 25
- 1347 Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees A. Vissers, and Zhiru Zhang. 2011. High-Level Synthesis
 1348 for FPGAs: From Prototyping to Deployment. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 30, 4 (2011), 473–491.
 1349 <https://doi.org/10.1109/TCAD.2011.2110592> 4
- 1350 Martin Ellis. 2008. *Correct synthesis and integration of compiler-generated function units*. Ph.D. Dissertation. Newcastle
 1351 University. <https://theses.ncl.ac.uk/jspui/handle/10443/828> 3, 25
- 1352 Dan Gajski, Todd Austin, and Steve Svoboda. 2010. What input-language is the best choice for high level synthesis (HLS)?
 1353 In *Design Automation Conference*. 857–858. <https://doi.org/10.1145/1837274.1837489> 3
- 1354 Stephane Gauthier and Zubair Wadood. 2020. High-Level Synthesis: Can it outperform hand-coded HDL? <https://bit.ly/2IDhKBR> White paper. 1
- 1355 David J. Greaves. 2019. Research Note: An Open Source Bluespec Compiler. *CoRR* abs/1905.03746 (2019). 3
- 1356 David J. Greaves and Satnam Singh. 2008. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *FCCM*. IEEE Computer
 1357 Society, 3–12. <https://doi.org/10.1109/FCCM.2008.46> 3
- 1358 Yann Herklotz, Zewei Du, Nadesh Ramanathan, and John Wickerson. 2021. An Empirical Study of the Reliability of
 1359 High-Level Synthesis Tools. In *29th IEEE International Symposium on Field-Programmable Custom Computing Machines*.
 1360 https://yannherklotz.com/docs/drafts/fuzzing_hls.pdf (to appear). 2, 24
- 1361 Ekawat Homsirikamol and Kris Gaj. 2014. Can high-level synthesis compete against a hand-written code in the cryptographic
 1362 domain? A case study. In *ReConFig*. IEEE, 1–8. <https://doi.org/10.1109/ReConFig.2014.7032504> 1
- 1363 Enoch Hwang, Frank Vahid, and Yu-Chin Hsu. 1999. FSM functional partitioning for low power. In *Proceedings of the*
 1364 *conference on Design, automation and test in Europe*. 7–es. <https://doi.org/10.1109/DATE.1999.761092> 6
- 1365 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (April
 1366 2006), 1–590. <https://doi.org/10.1109/IEEESTD.2006.99495> 3, 10
- 1367 2005. IEEE Standard for Verilog Register Transfer Level Synthesis. *IEC 62142-2005 First edition 2005-06 IEEE Std 1364.1* (2005),
 1368 1–116. <https://doi.org/10.1109/IEEESTD.2005.339572> 10
- 1369 Intel. 2020a. High-level Synthesis Compiler. <https://intel.ly/2UDiWr5> 1, 3
- 1370 Intel. 2020b. SDK for OpenCL Applications. <https://intel.ly/30sYH0> 24, 25
- 1371 Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *Programming Languages and*
 1372 *Systems*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 397–416. https://doi.org/10.1007/978-3-642-28869-2_20 3
- 1373 C Karfa, C Mandal, D Sarkar, S R. Pentakota, and Chris Reade. 2006. A Formal Verification Method of Scheduling in
 1374 High-level Synthesis. In *Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED '06)*. IEEE
 1375 Computer Society, Washington, DC, USA, 71–78. <https://doi.org/10.1109/ISQED.2006.10> 2, 25

1372

- David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszal, 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> 3
- 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 25
- Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. <https://doi.org/10.1145/1538788.1538814> 2, 3, 15
- 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> 1
- 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> 25, 27
- 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> 3, 10, 25
- Andreas Lööw and Magnus O. Myreen. 2019. A Proof-producing Translator for Verilog Development in HOL. In *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering* (Montreal, Quebec, Canada) (*FormalISE '19*). IEEE Press, Piscataway, NJ, USA, 99–108. <https://doi.org/10.1109/FormalISE.2019.00020> 2, 10, 11, 25, 27
- Mentor. 2020. Catapult High-Level Synthesis. <https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls> 2, 25
- 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 Codeign* (*MEMOCODE 2010*). 179–188. <https://doi.org/10.1109/MEMCOD.2010.5558634> 3, 10
- 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> 24, 25
- 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> 3
- 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> 5
- 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. 3
- 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> 20
- Maxime Pelcat, Cédric Bourrasset, Luca Maggiani, and François Berry. 2016. Design productivity of a high level synthesis compiler versus HDL. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. 140–147. <https://doi.org/10.1109/SAMOS.2016.7818341> 1
- 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> 25
- 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> 25
- Christian Pilato and Fabrizio Ferrandi. 2013. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *FPL*. IEEE, 1–4. <https://doi.org/10.1109/FPL.2013.6645550> 1, 3
- 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> 2, 24
- Louis-Noël Pouchet. 2020. PolyBench/C: the Polyhedral Benchmark suite. <http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/> 3, 20
- Louis-Noël 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> 20

- 1422 Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: A Multi-Level Intermediate Representation
1423 for Hardware Description Languages. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language
1424 Design and Implementation* (London, UK) (*PLDI 2020*). ACM, New York, NY, USA, 258–271. <https://doi.org/10.1145/3385412.3386024> 3
- 1425 Cyril Six, Sylvain Boulmé, and David Monniaux. 2020. Certified and efficient instruction scheduling: Application to
1426 interlocked VLIW processors. *Proc. ACM Program. Lang.* OOPSLA (2020). 26
- 1427 David B. Thomas. 2016. Synthesizable recursion for C++ HLS tools. In *ASAP*. IEEE Computer Society, 91–98. <https://doi.org/10.1109/ASAP.2016.7760777> 5
- 1428 Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: A Case Study on In-
1429 struction Scheduling Optimizations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Prin-
1430 ciples of Programming Languages* (San Francisco, California, USA) (*POPL '08*). ACM, New York, NY, USA, 17–27.
1431 <https://doi.org/10.1145/1328438.1328444> 2, 26
- 1432 Girish Venkataramani and Seth C. Goldstein. 2007. Operation chaining asynchronous pipelined circuits. In *2007 IEEE/ACM
1433 International Conference on Computer-Aided Design*. 442–449. <https://doi.org/10.1109/ICCAD.2007.4397305> 20
- 1434 Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A
1435 Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3, Article 22 (June 2013), 50 pages. <https://doi.org/10.1145/2487241.2487248> 14
- 1436 Yuting Wang, Xiangzhe Xu, Pierre Wilke, and Zhong Shao. 2020. CompCertELF: Verified Separate Compilation of C
1437 Programs into ELF Object Files. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 197 (Nov. 2020), 28 pages. <https://doi.org/10.1145/3428265> 14
- 1438 Xilinx. 2019. Vivado Design Suite. <https://bit.ly/2wZAmld> 21
- 1439 Xilinx. 2020. Vivado High-level Synthesis. <https://bit.ly/39ereMx> 1, 3, 24, 25
- 1440 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings
1441 of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA)
1442 (*PLDI '11*). ACM, New York, NY, USA, 283–294. <https://doi.org/10.1145/1993498.1993532> 2, 24
- 1443 Youngsik Kim, S. Kopuri, and N. Mansouri. 2004. Automated formal verification of scheduling process using finite state
1444 machines with datapath (FSMD). In *International Symposium on Signals, Circuits and Systems. Proceedings, SCS 2003. (Cat.
1445 No.03EX720)*. 110–115. <https://doi.org/10.1109/ISQED.2004.1283659> 2, 25
- 1446 Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. 2017. COMBA: A comprehensive model-
1447 based analysis framework for high level synthesis of real applications. In *2017 IEEE/ACM International Conference on
1448 Computer-Aided Design (ICCAD)*. IEEE, 430–437. 20
- 1449 Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate
1450 Representation for Verified Program Transformations. *SIGPLAN Not.* 47, 1 (Jan. 2012), 427–440. <https://doi.org/10.1145/2103621.2103709> 3
- 1451 Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. 2013. Improving polyhedral code
1452 generation for high-level synthesis. In *2013 International Conference on Hardware/Software Codesign and System Synthesis
1453 (CODES+ ISSS)*. IEEE, 1–10. <https://doi.org/10.1109/CODES-ISSS.2013.6659002> 20
- 1454
- 1455
- 1456
- 1457
- 1458
- 1459
- 1460
- 1461
- 1462
- 1463
- 1464
- 1465
- 1466
- 1467
- 1468
- 1469
- 1470