# Verified Compilation on a Verified Processor

Andreas Lööw
Chalmers University
Gothenburg, Sweden

Ramana Kumar
DeepMind
London, UK

Yong Kiam Tan
CMU
Pittsburgh, PA, USA

Magnus O. Myreen
Chalmers University
Gothenburg, Sweden

Michael Norrish
Data61, CSIRO and ANU
Canberra, Australia

Oskar Abrahamsson
Chalmers University
Gothenburg, Sweden

Anthony Fox
Arm Limited
Cambridge, UK

## Abstract

Developing technology for building verified stacks, i.e., computer systems with comprehensive proofs of correctness, is one way the science of programming languages furthers the computing discipline. While there have been successful projects verifying complex, realistic system components, including compilers (software) and processors (hardware), to date these verification efforts have not been compatible to the point of enabling *a single end-to-end correctness theorem* about running a verified compiler on a verified processor.

In this paper we show how to extend the trustworthy development methodology of the CakeML project, including its verified compiler, with a connection to verified hardware. Our hardware target is Silver, a verified proof-of-concept processor that we introduce here. The result is an approach to producing verified stacks that scales to proving correctness, at the hardware level, of the execution of realistic software including compilers and proof checkers. Alongside our hardware-level theorems, we demonstrate feasibility by hosting and running our verified artefacts on an FPGA board.

***CCS Concepts*** • **Hardware → Hardware description languages and compilation**; **Theorem proving and SAT solving**; • **Software and its engineering → Software verification**; **Compilers**.

***Keywords*** verified stack, program verification, hardware verification, compiler verification

**ACM Reference Format:**
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 (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3314221.3314622

*PLDI '19, June 22–26, 2019, Phoenix, AZ, USA*
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314622

## 1 Introduction

A *verified stack* is a computer system that is demonstrably correct. Specifically, it is a system with a formal proof of correctness that covers all layers of the implementation, from the hardware through to the application code. Enabling the construction of verified stacks is a guiding light for the field of formal verification; several projects have made progress towards its achievement [2, 3, 33]. In this paper, we report on a milestone in this tradition: a verified stack consisting of a verified processor that we have synthesized for an FPGA board on which we can run a realistic and verified compiler.

To reach this milestone, we have developed a new verified processor called Silver[1] that is simple but general-purpose, and have extended the trustworthy chain in the CakeML project with a link to Silver. The Silver processor was verified with ease thanks to a new proof-producing hardware generator that is grounded in a semantics for the hardware description language (HDL) Verilog. To produce machine code for Silver, we use the CakeML translator [16, 29] and compiler [14, 34]. We obtain end-to-end correctness theorems by composing the CakeML compiler's correctness theorem with the Silver processor's correctness theorem.

Our combination of Silver with CakeML yields a general method for verification down to the hardware level. Given a high-level executable specification of behaviour, our method produces machine code for Silver plus an end-to-end correctness theorem stating that the verified Silver hardware will have the observable behaviour of the original high-level specification, provided the generated Silver machine code is initially present in memory.

We demonstrate our method on several applications taken from CakeML's library, including word-count, sort, a proof-checker for OpenTheory proofs [18], and the CakeML compiler itself. To our knowledge, this is the first verified stack development that scales to the point of executing a realistic compiler on top of verified hardware, in a setting with

[1] As silverware may be used in consuming cakes and other food, Silver is hardware that can run CakeML as well as other programs.

*a single correctness theorem* that covers the full end-to-end composition.

Previously, the CakeML compiler targeted only architectures without verified implementations [14], such as x86 and ARM. When the target architecture has no correctness proof, the hardware and runtime environment must be modelled as assumptions on the compiler's correctness theorem. In this paper, by targeting the verified Silver processor, we address the question "How can the CakeML compiler (and other verified compilers) be extended to reduce assumptions about the hardware and environment?" In explaining this, we make the following contributions:

- We exhibit sufficient properties that, if proved about a compiler and a processor, enable them to be used together for constructing verified stacks. The shape of the correctness proofs in our method (§2) should be informative for other verified stacks.
- We show how we constructed and verified (§3) the Silver processor (§4) down to its implementation in Verilog, a mainstream low-level hardware description language. (The software side of our method has been described elsewhere [29, 34].)
- We address claims [21, 34] that the assumptions on the CakeML compiler's correctness theorem (§5) are reasonable, by showing how they can be satisfied (§6), and highlight minor changes that were required (§6.1).
- Finally, we contribute the Silver processor and extensions to CakeML to support Silver as re-usable artefacts for constructing verified stacks.

The whole development, including the CakeML compiler and the Silver processor, has been built using the HOL theorem prover [32]; the source code and proofs are available at https://github.com/CakeML/cakeml and https://github.com/CakeML/hardware.

## 2 Approach

Our approach to building verified stacks divides concerns, just as in the traditional approach to building (unverified) systems, with steps including:

1. Write functional specifications for the application.
2. Implement the specifications as source code in a high-level programming language.
3. Compile the source code to machine code.
4. Link the application machine code with code implementing any required system calls.
5. Run the resulting machine code on a processor (connected to memory and I/O devices).

The main omission is interaction with an operating system: at present, we focus on applications that run on "bare metal". In order to produce *verified* stacks, we have a verification story for each of the steps above, and we produce a single end-to-end theorem that composes the correctness theorems associated with each step.

Now, let us turn to the specific components we use to instantiate the template above, and their associated verification story. To make things concrete, we consider an example application, namely, wc, a program that counts the words it receives as input.

### 2.1 Specification

We write formal specifications in higher-order logic, specifically by defining functions in the HOL theorem prover. For wc, these include functions used in logical expressions such as: $|\mathsf{tokens}\ \mathsf{is\_space}\ \mathit{input}|$, which is the length $|\ldots|$ of the tokens function applied to the is_space function and *input*. We summarise the specification for wc as a relation, wc_spec *input output*, between input and output strings.

### 2.2 High-Level Implementation

We implement the application in CakeML, generating code from the specification whenever possible. To do this, we use the CakeML translator [16, 29], which, given specifications that are pure functions or monadic functions representing impure computations, produces both an implementation in CakeML code and a certificate theorem.[2] Ultimately, we obtain a proof that a CakeML program, wc_prog in our example, successfully terminates[3] with output conforming to the specification[4]:

$$\begin{aligned}
&\vdash \exists\, \mathit{io}.\\
&\quad \mathsf{cakeml\_sem}\ ([\text{“wc”}],\mathsf{fs}_{\mathsf{in}}\ \mathit{input})\ \mathsf{wc\_prog} =\\
&\qquad \{\ \mathsf{Terminate}\ \mathsf{Success}\ \mathit{io}\ \}\ \wedge\\
&\quad \mathsf{wc\_spec}\ \mathit{input}\ (\mathsf{get\_stdout}\ \mathit{io})
\end{aligned} \tag{1}$$

Here ["wc"] represents the command line and $\mathsf{fs}_{\mathsf{in}}\ \mathit{input}$ represents the filesystem state when the program starts, in this case with no files but with the string *input* to be read on standard input.

### 2.3 Compilation to Machine Code

We compile the CakeML code to Silver machine code using the optimising CakeML compiler, selecting the new target, Silver (ag32), that we have added to the compiler backend. The compiler is proved correct once and for all, with a correctness theorem of this form:

$$\begin{aligned}
&\vdash \mathsf{cakeml\_sem}\ (\mathit{cl},\mathit{fs})\ \mathit{prog} = \mathit{behaviours}\ \wedge\\
&\quad \mathsf{Fail} \notin \mathit{behaviours}\ \wedge\\
&\quad \mathsf{compile}\ \mathsf{conf}_{\mathsf{Ag}}\ \mathit{prog} = \mathsf{Some}\ \mathit{compiled\_prog}\ \wedge\\
&\quad \mathsf{installed}_{\mathsf{Ag}}\ \mathit{compiled\_prog}\ (\mathsf{basis\_ffi}\ \mathit{cl}\ \mathit{fs})\ \mathit{ms} \Rightarrow\\
&\qquad \mathsf{machine\_sem}\ (\mathsf{basis\_ffi}\ \mathit{cl}\ \mathit{fs})\ \mathit{ms} \subseteq\\
&\qquad\quad \mathsf{extend\_with\_oom}\ \mathit{behaviours}
\end{aligned} \tag{2}$$

[2] If parts of the desired implementation cannot be translated from the specification, we write CakeML code by hand and verify it using Characteristic Formulae [15] in the tradition of Hoare logic.

[3] CakeML supports correctness proofs for non-terminating programs, but we do not cover them in this paper.

[4] Equality (=) binds tighter than conjunction (∧); termination and conformance are represented by the two conjuncts.

Here, basis_ffi *cl fs* models the behaviour of system calls that access the command line *cl* and file system *fs* as expected by CakeML's basis library; machine_sem produces a set of behaviours by repeatedly stepping the machine state *ms*, applying external interference and executing system calls according to the basis_ffi model [14]; and extend_with_oom adds additional behaviours to the compiled program, namely, allowing it to terminate prematurely with an out-of-memory error, having done only a prefix of the correct I/O events.

We execute the compiler inside the theorem prover (essentially by rewriting with its definition) in order to obtain a compiled program (i.e., bytes of machine code), here wc_ag32, and a compilation theorem:

$$\vdash \text{compile conf}_{\text{Ag}}\ \text{wc\_prog} = \text{Some wc\_ag32} \tag{3}$$

Instantiating the compiler correctness theorem (2) with the compilation theorem (3) and the application semantics theorem (1), we obtain a correctness theorem about the application machine code. For `wc`, the theorem is:

$$\begin{array}{l}
\vdash \text{installed}_{\text{Ag}}\ \text{wc\_ag32}\ (\text{basis\_ffi}\ [\text{“wc”}]\ (\text{fs}_{\text{in}}\ \mathit{input}))\ \mathit{ms} \Rightarrow \\
\quad \exists\, \mathit{io}. \\
\quad\quad \text{machine\_sem}\ (\text{basis\_ffi}\ [\text{“wc”}]\ (\text{fs}_{\text{in}}\ \mathit{input}))\ \mathit{ms} \subseteq \\
\quad\quad\quad \text{extend\_with\_oom}\ \{\ \text{Terminate Success}\ \mathit{io}\ \}\ \land \\
\quad\quad \text{wc\_spec}\ \mathit{input}\ (\text{get\_stdout}\ \mathit{io})
\end{array} \tag{4}$$

This theorem, relating the semantics of a machine code program, wc_ag32, to the high-level specification, wc_spec, is what the CakeML project provided prior to this paper. Crucially, this theorem has an assumption constraining the initial machine state, namely $\text{installed}_{\text{Ag}}$ wc_ag32 . . ., which states that the compiled program is loaded correctly and that the external environment (via system calls and interference) behaves as modelled (§5). By targeting a verified processor and verifying the system library, we reduce this assumption, replacing substantial parts of it by proofs.

### 2.4 Verified System Calls

CakeML programs interact with their environment via system calls for reading command-line arguments and reading/writing to standard streams and files. For Silver, we have realised the standard streams `std{in,out,err}`, and the command line, as in-memory devices accessed by Silver machine code that we have verified to implement the system calls required by CakeML. The input devices are pre-filled before execution, and the output devices are connected to a text terminal. As we are developing bare-metal applications, the verified system calls code is included as part of the memory image loaded at startup.

Our theorems about the system calls (§6) enable us to replace the $\text{installed}_{\text{Ag}}$ assumption in theorem (4) with a simpler assumption, $\text{init}_{\text{Ag}}$ . . ., merely stating that the compiled code, system calls code, and input data is in memory. The resulting theorem has this form:

$$\begin{array}{l}
\vdash |\mathit{input}| \le \text{stdin\_size} \land \\
\quad \text{init}_{\text{Ag}}\ \text{wc\_ag32}\ ([\text{“wc”}], \mathit{input})\ \mathit{ms} \Rightarrow \\
\quad\quad \exists\, \mathit{io}\ k. \\
\quad\quad\quad \text{machine\_sem}\ (\text{basis\_ffi}\ [\text{“wc”}]\ (\text{fs}_{\text{in}}\ \mathit{input})) \\
\quad\quad\quad\quad (\text{Next}^k\ \mathit{ms}) \subseteq \\
\quad\quad\quad\quad \text{extend\_with\_oom}\ \{\ \text{Terminate Success}\ \mathit{io}\ \}\ \land \\
\quad\quad\quad \text{wc\_spec}\ \mathit{input}\ (\text{get\_stdout}\ \mathit{io})
\end{array} \tag{5}$$

Here, $\text{Next}^k\ \mathit{ms}$ is the result of $k$ steps of execution from the initial machine state *ms*, corresponding to execution of startup code that sets a few registers to satisfy the initialisation assumptions of CakeML (§5); and stdin_size is a constant representing the maximum amount of pre-filled input we support (about 5 MB).

Working through the definition of machine_sem (essentially, repeated application of Next, using basis_ffi to handle system calls), and using our verified system calls code, we obtain the following version of this theorem phrased entirely in terms of the Silver ISA and its next-state function Next:

$$\begin{array}{l}
\vdash |\mathit{input}| \le \text{stdin\_size} \land \\
\quad \text{init}_{\text{Ag}}\ \text{wc\_ag32}\ ([\text{“wc”}], \mathit{input})\ \mathit{ms} \Rightarrow \\
\quad\quad \exists\, \mathit{io}.\ \text{FG}\ k. \\
\quad\quad\quad \text{wc\_spec}\ \mathit{input}\ (\text{get\_stdout}\ \mathit{io}) \land \\
\quad\quad\quad \text{is\_halted}_{\text{Ag}}\ (\text{Next}^k\ \mathit{ms}) \land \\
\quad\quad\quad \text{stdout}_{\text{Ag}}\ (\text{Next}^k\ \mathit{ms}).\text{io\_events} \preccurlyeq \text{get\_stdout}\ \mathit{io} \land \\
\quad\quad\quad (\text{exit\_code\_0}_{\text{Ag}}\ (\text{Next}^k\ \mathit{ms}) \Rightarrow \\
\quad\quad\quad\quad \text{stdout}_{\text{Ag}}\ (\text{Next}^k\ \mathit{ms}).\text{io\_events} = \text{get\_stdout}\ \mathit{io})
\end{array} \tag{6}$$

We use the FG operator here to capture the notion that a predicate becomes true at some unspecified point in the future, and then remains true thereafter.[5] Thus we see that the Terminate Success *io* behaviour of machine_sem corresponds to execution for some number of steps, at which point the machine reaches a halting state ($\text{is\_halted}_{\text{Ag}}$ . . .), which is a program-specific location where the machine remains for any further steps. Furthermore, at this point, the trace of writes to `stdout` ($\text{stdout}_{\text{Ag}}$ . . .) will be a prefix ($\preccurlyeq$) of the specified output, attaining equality if the machine exited successfully ($\text{exit\_code\_0}_{\text{Ag}}$ . . .) without running out of memory.

The assumptions on theorem (4) that we have proved to reach theorem (6) cannot be discharged when developing applications for unverified operating systems. In particular, to reach theorems (5) and (6), we must prove (see §6) that the system calls run correctly under the same ISA semantics, Next, that runs the application code.

### 2.5 Execution on a Verified Processor

We have proved that the Silver processor, as a hardware circuit, silver_cpu_verilog, implemented in the HDL Verilog, implements the Silver ISA (with next-state function Next). To prove this, we used our new proof-producing Verilog code

[5] The FG operator, based on temporal logic, is defined as: $(\text{FG}\ t.\ P\ t) \overset{\text{def}}{=} \exists\, t_0.\ \forall\, t.\ t_0 \le t \Rightarrow P\ t$

generator (§3). The processor correctness theorem for Silver is as follows:

$$\begin{array}{l} \vdash \text{let } \mathit{vstep} \;=\; \text{verilog\_sem } \mathit{env} \text{ silver\_cpu\_verilog } \mathit{init} \text{ in} \\ \quad \text{is\_lab\_env acc\_env\_verilog } \mathit{vstep}\ \mathit{env} \wedge \\ \quad \text{ag32\_eq\_init\_isa\_verilog } (\mathit{env}\ 0)\ \mathit{ms}\ \mathit{init} \Rightarrow \\ \quad\;\; \forall\, k.\ \exists\, m\ \mathit{fin}. \\ \quad\quad \mathit{vstep}\ m = \text{Ok } \mathit{fin} \wedge \\ \quad\quad \text{ag32\_eq\_isa\_verilog } (\mathit{env}\ m)\ (\text{Next}^k\ \mathit{ms})\ \mathit{fin} \end{array} \tag{7}$$

Here, *vstep m* is the state of the Silver processor implementation after *m* clock cycles. The *env* function is used to represent processor-external entities (is_lab_env ...), such as memory. The two relations ag32_eq_init_isa_verilog and ag32_eq_isa_verilog belong to a family of relations used to express equality between processor states at different abstraction levels and specify the values of various implementation-level registers (see §4). The theorem thus states that any number of steps $k$ taken by the ISA can be simulated by a number of steps $m$ by the implementation.

Combining theorem (7) with theorem (6), and working through some details about the processor state, we obtain:[6]

$$\begin{array}{l} \vdash \text{let } \mathit{vstep} \;=\; \text{verilog\_sem } \mathit{env} \text{ silver\_cpu\_verilog } \mathit{init} \text{ in} \\ \quad |\mathit{input}| \leq \text{stdin\_size} \wedge \\ \quad \text{is\_lab\_env acc\_env\_verilog } \mathit{vstep}\ \mathit{env} \wedge \\ \quad \text{verilog\_init}_{\text{Ag}} \text{ wc\_ag32 } ([\text{``wc''}], \mathit{input})\ \mathit{init}\ \mathit{env} \Rightarrow \\ \quad\;\; \exists\, \mathit{output}.\ \text{FG } m.\ \exists\, \mathit{fin}. \\ \quad\quad \text{wc\_spec } \mathit{input}\ \mathit{output} \wedge \mathit{vstep}\ m = \text{Ok } \mathit{fin} \wedge \\ \quad\quad \text{verilog\_is\_halted}_{\text{Ag}}\ \mathit{fin} \wedge \\ \quad\quad \text{stdout}_{\text{Ag}}\ (\mathit{env}\ m).\text{io\_events} \preccurlyeq \mathit{output} \wedge \\ \quad\quad (\text{verilog\_exit\_code\_0}_{\text{Ag}}\ \mathit{fin} \Rightarrow \\ \quad\quad\quad \text{stdout}_{\text{Ag}}\ (\mathit{env}\ m).\text{io\_events} = \mathit{output}) \end{array} \tag{8}$$

Theorems of this form are our milestone: working from a high-level specification (wc_spec), theorem (8) states that a piece of *hardware* (described in Verilog) implements that specification. Importantly, the creative verification work of the programmer is done at the high level of a CakeML program, not at the hardware level of Verilog.

From the verified circuit silver_cpu_verilog, we have synthesised the Silver processor for an FPGA board. (This is possible because our code generator produces synthesisable Verilog.) If we load a memory image (§6) containing the machine code wc_ag32, the input text, and the system calls code onto the board with the synthesised processor and set it running, the board outputs the number of words in the input (i.e., it runs wc).

The software half of the approach (the production of verified machine code from functional specifications) is explained in detail in previous work [14, 29, 34] that we do not repeat. Our focus is on the hardware side and the software-hardware connection. We start by explaining how we developed and verified Silver (§3, §4), then describe the connection to CakeML (§5) including verification of the system calls (§6), and finally present the hardware-level correctness theorem for the CakeML compiler itself (§7) and discuss what remains in the trusted computing base (§8).

## 3 Producing Verified Hardware

We have developed a new proof-producing Verilog code generator that translates HOL functions modelling circuits to deeply embedded Verilog programs. The Verilog programs are animated by a new operational semantics for a subset of Verilog that we have developed in parallel with the code generator. The code generator enables relating circuit verification results to a deeply embedded semantics for a mainstream low-level HDL (here, Verilog), which is novel (see §9). The output from the Verilog code generator can be pretty-printed and fed into synthesis toolchains, such as Xilinx's Vivado Design Suite which we have used, to produce FPGA artefacts.

***Example.*** The code generator takes as input a *circuit function* in HOL. A circuit function takes a world-state function *env*, a circuit-state *s*, and a clock *n*, and returns the circuit state after *n* cycles. Circuit functions are expressed in terms of next-state functions representing Verilog processes. Our example circuit AB consists of two processes A and B that count the number of pulses (*env*.pulse) and set done to true (T) after 10 pulses. Here $1w$ and $10w$ are word literals (with lengths inferred from context); $<_+$ denotes unsigned less-than; with updates a record; and ⦇ ... ⦈ constructs a record:

$$\begin{array}{l} \text{A } \mathit{env}\ s \overset{\text{def}}{=} \\ \quad \text{if } \mathit{env}.\text{pulse then } s \text{ with count } := s.\text{count} + 1w \text{ else } s \end{array}$$

$$\begin{array}{l} \text{B } s \overset{\text{def}}{=} \\ \quad \text{if } 10w <_+ s.\text{count then } s \text{ with done } := \text{ T else } s \end{array}$$

$$\begin{array}{l} \text{AB } \mathit{env}\ s\ 0 \overset{\text{def}}{=} s \\ \text{AB } \mathit{env}\ s\ (\text{Suc } n) \overset{\text{def}}{=} \\ \quad \text{let } s' \;=\; \text{AB } \mathit{env}\ s\ n \text{ in} \\ \quad\;\; ⦇\text{count } := (\text{A } (\mathit{env}\ n)\ s').\text{count}; \\ \quad\quad \text{done } := (\text{B } s').\text{done}⦈ \end{array}$$

If, as in AB, the input processes do not interfere with each other—i.e., all writes to variables used for communication between processes can be handled by Verilog's non-blocking assignment construct—then the code generator produces a (deeply embedded) Verilog process for each HOL next-state function and then composes them into a complete Verilog module. For the AB example, the code generator produces a Verilog module ABv containing the following code:

```
always_ff @ (posedge clk) // A
 if (pulse) count <= count + 8'd1;

always_ff @ (posedge clk) // B
 if (8'd10 < count) done = 1;
```

The code generator is proof-producing: for each run it produces a correspondence theorem stating that the generated Verilog program has the same behaviour as the input HOL circuit function. This correspondence theorem enables us to transfer properties proved about HOL-level hardware

[6] The predicates with a `verilog_` prefix are analogues of those in theorem (6) defined at the Verilog rather than the ISA level.

descriptions to the Verilog level. To illustrate, if we assume that the input pulse to AB is high infinitely often,

$$\mathsf{pulse\_spec}\ env \overset{\text{def}}{=} \forall n.\ \exists m.\ (env\ (n+m)).\mathsf{pulse},$$

then we can easily prove

$$\vdash \mathsf{pulse\_spec}\ env \Rightarrow \exists n.\ (\mathsf{AB}\ env\ init\ n).\mathsf{done},$$

which in turn can be transported to the level of the Verilog semantics using the generated correspondence theorem:

$$\begin{array}{l} \vdash \mathsf{pulse\_spec}\ env \wedge \mathsf{vars\_has\_type}\ s\ \mathsf{ABtypes} \Rightarrow \\ \quad \exists n\ s'. \\ \qquad \mathsf{verilog\_sem}\ env\ \mathsf{ABv}\ s\ n = \mathsf{Ok}\ s' \wedge \\ \qquad \mathsf{verilog\_get\_var}\ s'\ \text{“done”} = \mathsf{Ok}\ (\mathsf{VBool}\ \mathsf{T}) \end{array}$$

***Tool implementation.*** Our Verilog code generator is inspired by the CakeML translator [16, 29], and the Verilog semantics that goes with it is based on the official Verilog standard [1]. We aimed at soundly capturing the standard for a restricted subset of Verilog that we found to be sufficient for describing simple synthesisable synchronous hardware.

In our semantics, we consider a flattened module hierarchy, where all processes correspond to `always_ff` procedural blocks waiting on a program-common clock's `posedge`. We limit the amount of concurrency we need to model by only considering non-interfering processes, where all non-blocking writes are saved in a queue during cycle execution. The contents of this queue is merged into the program state at the end of every clock cycle.

The code generator translates HOL Booleans to Verilog Booleans, and HOL words to Verilog arrays. The Verilog Booleans in our semantics only take on the standard Boolean values true (`1`) and false (`0`), as we do not consider wires driven by multiple drivers (`Z`) in our formalisation, and unknown values (`X`) are modelled using quantification inside the logic. The details of the code generator and the semantics are described in more detail in Lööw and Myreen [25].

## 4 The Silver CPU

In this section, we present the Silver ISA, the Silver FPGA implementation and the environment it executes in, and the correctness theorem relating the ISA and its implementation. Figure 1 outlines the relationships between the different layers of the implementation of the Silver ISA.

### 4.1 The Silver ISA

The Silver ISA (instruction set architecture) is the target of the CakeML compiler's Silver backend. As shown in the topmost layer of Figure 1, the ISA is written in the L3 language [13], a domain-specific language for ISAs that is also used for the other CakeML compiler targets, and can be transformed into HOL definitions by the L3-to-HOL compiler. Neither the L3 ISA nor the L3-to-HOL compiler are part of the trusted base of the Silver processor: L3 is merely a convenient way to generate HOL definitions of the Silver ISA that can be independently inspected. The generated ISA—the second layer in Figure 1—is what is used in our proofs.

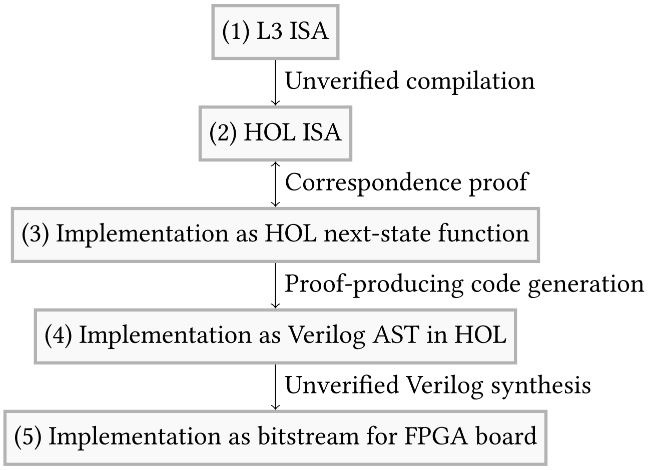

(1) L3 ISA
Unverified compilation
(2) HOL ISA
Correspondence proof
(3) Implementation as HOL next-state function
Proof-producing code generation
(4) Implementation as Verilog AST in HOL
Unverified Verilog synthesis
(5) Implementation as bitstream for FPGA board

**Figure 1.** The layers involved in the construction and verification of the Silver processor.

The Silver ISA in HOL is an operational semantics over a machine state represented as a HOL record ⟨|...|⟩. The machine state contains memory (a function from addresses to bytes), registers (a function from register indices to words), the current program counter (PC), some flags, and a trace of I/O events. The semantics, Next, is a fetch-execute function that retrieves the bytes from memory pointed to by the program counter, decodes them into an instruction, then executes the instruction by updating the machine state. For example, the execute part of Silver's LoadConstant instruction is described as follows:

$$\begin{array}{l} \mathsf{LoadConstant}\ (reg,negate,imm)\ s \overset{\text{def}}{=} \\ \quad \mathsf{let}\ v\ =\ \mathsf{w2w}\ imm\ \mathsf{in} \\ \quad\ s\ \mathsf{with} \\ \quad\ \langle|\mathsf{R}\ :=\ s.\mathsf{R}(|reg \leftarrow \mathsf{if}\ negate\ \mathsf{then}\ -v\ \mathsf{else}\ v|); \\ \quad\quad \mathsf{PC}\ :=\ s.\mathsf{PC} + 4w|\rangle \end{array}$$

Here w2w is an unsigned resizing of words, with updates a record, $f(|k \leftarrow v|)$ denotes a function identical to $f$ except that $f\ k = v$, and $4w$ is a word literal. The *reg*, *negate*, and *imm* parameters are information from the instruction decoder specifying which register to update with what content. We see that the function LoadConstant updates the state record fields R (registers) and PC (program counter). There are similar semantics functions for each instruction in the Silver ISA.

#### 4.1.1 Instruction Listing

The Silver ISA is a general-purpose RISC ISA designed to support CakeML. Each instruction is 32-bits long and operates over 32-bit words. The Silver ISA has its roots in Thacker's Tiny 3 computer [35], but has since evolved significantly.

***Loading constants into registers.*** The CakeML compiler frequently wants to load large constants into registers. The

Silver ISA supports loading a 23-bit immediate (or its negation) into the lower bits of a register. With another instruction, Silver supports loading a 9-bit immediate value into the upper bits of a register.

***ALU operations.*** The Silver ISA provides instructions for two-argument ALU operations. The ALU supports integer addition, integer add with carry, integer subtraction, increment by one, decrement by one, multiplication (with 64-bit output), logical and, logical or, logical xor, equality, unsigned less-than, signed less-than, retrieving the current carry flag, retrieving the current overflow flag, and simply returning the second operand. The add and subtraction operations update the carry and overflow flags.

***Shifts and rotations.*** Separately from the ALU instructions, there are bit-shift and bit-rotation instructions, in both signed and unsigned variants where necessary.

***Memory.*** Memory can be stored or loaded either as words or individual bytes.

***Jumps.*** The Silver ISA supports conditional and unconditional PC-relative jumps, as well as unconditional jumps to absolute addresses. Jump offsets (or addresses) can be computed (i.e., obtained from a register), which is important when tail-calling a closure or returning from a function (moving the value of the link register into the PC).

***Interrupt.*** The Silver ISA includes an Interrupt instruction, which is used for notifying external hardware of an observable event. In the implementation, Interrupt notifies external hardware and waits for a response before continuing execution. In the semantics of the ISA, Interrupt silently records the current state of memory by pushing it onto the trace of I/O events.

### 4.2 The Silver Implementation

We have constructed a Silver processor, implementing the Silver ISA, that is designed for the PYNQ-Z1 FPGA SoC board. The target board is relevant in that any implementation must be adapted to the I/O and memory devices available. The centrepiece of our implementation is an environment-independent processor core, which is connected to the runtime environment by a layer of environment-dependent glue.

The PYNQ board hosts an FPGA chip, an ARM core running Linux which is accessible over SSH, and a DRAM module that is shared between the FPGA chip and the ARM core. In our "lab setup", when we execute a program compiled by the CakeML compiler, we use the ARM core first to load the synthesised Silver processor, as an FPGA bitstream, onto the FPGA chip, and then to preload the shared DRAM module with the appropriate memory image (the image (§6) contains machine code produced by the compiler, our system calls code, and data for the command line and standard input).

Formally, we represent the external environment the processor interacts with as a function *env* from timesteps to the state of the world. The environment is assumed to include a memory interface (is_mem, the DRAM module), an initialisation interface (is_mem_start_interface, notifying when memory is correctly pre-filled), and an interrupt handling interface (is_interrupt_interface, invoked when an Interrupt instruction is executed):

$$\begin{array}{l}\mathsf{is\_lab\_env}\ \mathit{accessors}\ \mathit{step}\ \mathit{env} \overset{\text{def}}{=} \\ \quad \mathsf{is\_mem}\ \mathit{accessors}\ \mathit{step}\ \mathit{env} \wedge \mathsf{is\_mem\_start\_interface}\ \mathit{env} \wedge \\ \quad \mathsf{is\_interrupt\_interface}\ \mathit{accessors}\ \mathit{step}\ \mathit{env}\end{array}$$

The *accessors* argument is an implementation detail and makes the definitions usable at multiple abstraction levels. As an example, the initialisation interface is formalised as:

$$\begin{array}{l}\mathsf{is\_mem\_start\_interface}\ \mathit{env} \overset{\text{def}}{=} \\ \quad \exists\, n. \\ \quad\quad (\forall\, m.\ m < n \Rightarrow \neg(\mathit{env}\ m).\mathsf{mem\_start\_ready}) \wedge \\ \quad\quad (\mathit{env}\ n).\mathsf{mem\_start\_ready}\end{array}$$

In our lab setup, the interrupt interface is connected to the ARM core, and is used to notify the core of, e.g., system calls it must react to, such as text output calls.

To produce a hardware description of the Silver processor inside HOL—that is, layer 3 in Figure 1—we refined the HOL ISA step by step into a hardware description. The implementation is not pipelined, and executes instructions in-order, and is consequently similar, at a high-level, to the ISA.

The main difference between the implementation and the ISA is that the implementation must interact with the external interfaces defined above, e.g., instead of updating an abstract memory map as in the ISA, the implementation must access external memory (using the interface defined by is_mem). As a result, the implementation has additional wait states that do not correspond to any state in the ISA, as the processor sometimes has to wait for external interfaces, such as memory, to respond to requests. Because of these additional states, there are two different notions of time. In the ISA, a "step" corresponds to an *instruction cycle*, whereas an implementation-level "step" corresponds to a *clock cycle*; an instruction cycle takes multiple clock cycles to realise in hardware.

Another important step in the refinement process was de-duplication of some elements of the ISA. For illustration, consider the definition of the LoadConstant instruction given above. The computation of the next PC is computed directly in the definition of the instruction's semantics. This pattern is repeated for every instruction (except, e.g., jump instructions, where the next PC is computed by more complicated means). One does not want to translate descriptions of instructions such as these to hardware naively, because then the hardware component computing the next PC would be duplicated one time per instruction, wasting hardware resources. Computing the next PC should instead be carried out by a single, shared, hardware component. So, part of the manual refinement process was to identify which parts of the ISA should be implemented by shared hardware components, and which could be implemented in a more direct way, similar to the structure suggested by the ISA.

### 4.3 Algorithmic Correctness of Silver

To show that the implementation correctly implements the ISA, we have proved a simulation correspondence between the two levels, saying that for any $n$ instruction cycles the ISA can take, these steps can be simulated by running the implementation $m$ clock cycles:

$$\begin{aligned}&\vdash \mathsf{let}\ \mathit{cstep}\ =\ \mathsf{silver\_cpu}\ \mathit{init}\ \mathit{env}\ \mathsf{in}\\ &\quad \mathsf{ag32\_eq\_init\_hol\_isa}\ (\mathit{env}\ 0)\ \mathit{init}\ s\ \wedge\\ &\quad \mathsf{is\_lab\_env}\ \mathsf{acc\_env}\ \mathit{cstep}\ \mathit{env} \Rightarrow\\ &\qquad \forall n.\ \exists m.\\ &\qquad\quad \mathsf{ag32\_eq\_hol\_isa}\ (\mathit{env}\ m)\ (\mathit{cstep}\ m)\ (\mathsf{Next}^n\ s)\end{aligned} \tag{9}$$

The relation ag32_eq_init_hol_isa belongs to the family of state relations mentioned in §2, and says that all ISA-visible state components at the two levels must be equal, e.g., that the memories have the same content and the registers are element-wise equal. The relation also says that some implementation registers must be in their start-up states. The relation ag32_eq_hol_isa is similar to ag32_eq_init_hol_isa in that it also says that all ISA-visible state components (again including memory) must be equal, but differs by stating that some implementation registers must now be in their in-execution states. The two different state-equality relations are needed as the implementation initially needs to wait for memory to respond with a first instruction before it can proceed in in-execution mode. Lastly, silver_cpu is the HOL hardware description of the processor, in the form of a next-state function expressed such that it is accepted as input by our Verilog code generator.

The simulation correspondence is quite weak, as it does not provide any information about what happens during the implementation's execution of an instruction. In particular, it does not tell us anything about the wait states mentioned in the previous section. For example, in terms of the `wc` example in §2, to prove theorem (8), beyond theorem (7) and theorem (6), we needed a separate lemma saying that the processor "does nothing" after a CakeML program has terminated; or, in other words, a lemma stating that the ISA-visible state is unchanged at any clock cycle after program termination, not just at any instruction cycle.

### 4.4 Correctness of the Verilog Implementation

The correspondence between the HOL processor implementation and the Verilog processor implementation is more direct than the correspondence between the HOL processor implementation and the processor ISA. More precisely, with the help of the Verilog code generator invoked on the HOL processor, we can prove the following theorem:

$$\begin{aligned}&\vdash \mathsf{ag32\_eq\_hol\_verilog}\ \mathit{init}\ \mathit{vs} \Rightarrow\\ &\quad \exists \mathit{vs}'.\\ &\qquad \mathsf{verilog\_sem}\ \mathit{env}\ \mathsf{silver\_cpu\_verilog}\ \mathit{vs}\ n = \mathsf{Ok}\ \mathit{vs}'\ \wedge\\ &\qquad \mathsf{ag32\_eq\_hol\_verilog}\ (\mathsf{silver\_cpu}\ \mathit{init}\ \mathit{env}\ n)\ \mathit{vs}'\end{aligned} \tag{10}$$

Here ag32_eq_hol_verilog is another relation from the state-equality family, again requiring that its two parameters represent the same machine state at two different abstraction levels. As seen previously, the verilog_sem function runs Verilog programs in our Verilog semantics, and silver_cpu_verilog refers to the Verilog program the translator built out of the HOL processor implementation silver_cpu.

The derivation of theorem (10) is mostly automated by the code generator; the main obligation to discharge as a user of the code generator is to express the input circuit as a hardware description at the same level as the example from §3. To derive the ISA and Verilog implementation correspondence theorem (7) from §2, we simply compose theorem (10) with the implementation correctness theorem (9).

For synthesis for our FPGA board, we have used the Verilog code generated by the process described in this section in combination with some Verilog glue to connect the processor to its environment (see the discussion in §8).

## 5 CakeML's Assumptions

The CakeML compiler's correctness theorem makes a long list of assumptions regarding the execution environment of the generated code. This section presents what the assumptions are, while the next section explains how we have met these assumptions with the verified Silver processor and verified implementation of system calls.

The compiler's correctness theorem, an instance of which is theorem (2) in §2.3, includes the following assumption:

$$\mathsf{installed}_{\mathsf{Ag}}\ \mathit{compiled\_prog}\ (\mathsf{basis\_ffi}\ \mathit{cl}\ \mathit{fs})\ \mathit{ms},$$

which encapsulates the assumptions about the execution environment of the generated code. It relates the generated code *compiled_prog*, the command-line arguments *cl*, the state of the file system *fs*, and the Silver ISA machine state *ms*. Informally, it requires that *ms* is set up correctly for execution of *compiled_prog* to begin.

The formal definition of $\mathsf{installed}_{\mathsf{Ag}}$ is too long to reproduce in full here, but the following list outlines its contents:

(i) Registers 1–4 provide accurate information on where the part of memory usable by *compiled_prog* is located in machine state *ms*.
(ii) The read-only data of *compiled_prog* is stored in memory, where it is expected to be (based on registers 1–4).
(iii) The machine code of *compiled_prog* is stored in memory, and the program counter of *ms* points at the first address of this machine code.
(iv) The code and data sections do not overlap and various pointers are aligned to word boundaries.

(v) Calls to external functions (i.e., system calls) behave according to the modelled behaviour of the filesystem *fs* and command line *cl*.

The last point (v) above is by far the most complicated assumption. It requires that each time the CakeML-generated code jumps to external code (e.g., code for reading external input), the external code will execute and safely return to the CakeML code according CakeML's calling convention for external calls. Furthermore, each execution of external code must adhere to the CakeML basis library's assumption basis_ffi *cl fs*, explained in detail below.

The formal definition (omitted) of (v) is slightly unintuitive because the property is defined in terms of restricting the freedom of an oracle function. This oracle function, which we call the *interference oracle* of the foreign-function interface (FFI), is an argument to the operational semantics machine_sem that is used in the correctness theorem for the CakeML compiler. For the most part, machine_sem executes the next instruction using the Silver ISA's next-state function Next. However, when machine_sem encounters an entry point to external code (an FFI call), the semantics consults the interference oracle to determine what the resulting Silver ISA state should be.

The interference oracle is restricted to leave unchanged the part of the machine state that is private to CakeML code; the oracle is obliged to write the correct return value (according to basis_ffi *cl fs*, see below) to the shared array that is used for communicating between CakeML code and the external code; and the interference oracle is forced to set the program counter to the correct return address (in order to continue execution of the CakeML code).

So, what is basis_ffi *cl fs*? It encapsulates the assumptions that the CakeML standard basis library makes of its foreign-function interface. In the context of bare-metal systems, this is the interface to system calls that support the I/O functions in the basis library. Concretely, basis_ffi is defined as a record that defines (1) an oracle function basis_ffi_oracle, which specifies the behaviour of each call, and (2) the current state of the external world consisting of a command line (*cl*) and a filesystem (*fs*). The basis_ffi_oracle recognises calls to: "read", "write", "get_arg_count", "get_arg_length", "get_arg", "open_in", "open_out", "close", and "exit". An excerpt of its definition is shown below:

```
basis_ffi_oracle $name$ $(cl,fs)$ $conf$ $bytes$ $\stackrel{\text{def}}{=}$
  if $name$ = "read" then
    case ffi_read $conf$ $bytes$ $fs$ of
      FFIfail $\Rightarrow$ Oracle_final FFI_failed
    | FFIreturn $bytes$ $fs$ $\Rightarrow$ Oracle_return $(cl,fs)$ $bytes$
    | FFIdiverge $\Rightarrow$ Oracle_final FFI_failed
  else ...
```

When the FFI with name "read" is called, basis_ffi_oracle delegates the task to ffi_read, which receives configuration *conf*, input *bytes* and the current state of the filesystem *fs* as arguments. The *conf* and *bytes* are arguments that the CakeML programmer passed to the call at the source level. From the programmer's perspective, *bytes* is a byte array that they have made the FFI call with, in this case, expecting it to be filled with characters from reading a file. The function ffi_read is defined as:

```
ffi_read $conf$ $(b_0 :: b_1 :: b_2 :: b_3 :: bytes)$ $fs$ $\stackrel{\text{def}}{=}$
  do
    assert $(|bytes| \geq$ w22n $[b_0;\ b_1] \wedge |conf| = 8)$;
    $(l,fs') \leftarrow$ read (w82n $conf$) $fs$ (w22n $[b_0;\ b_1]$);
    FFIreturn
      ($[0w]$ ++ n2w2 $|l|$ ++ $[b_3]$ ++ map c2w $l$ ++
       drop $|l|$ $bytes$) $fs'$
  od otherwise (FFIreturn $(1w :: b_1 :: b_2 :: b_3 :: bytes)$ $fs$)
```

When ffi_read receives a *bytes* argument of sufficient length, it calls a read function from the filesystem model. This read function (definition omitted) is given a file handler, a filesystem state and the maximum length it is allowed to read. The read function returns $(l,fs')$, where $l$ is the number of bytes that were actually read and $fs'$ is the new filesystem state. The complicated list expression passed to FFIreturn specifies how the length of list $l$ and its content is communicated in the shared array on return. On failure, ffi_read returns $1w$ in the first element of the array. All of the functions involved here are defined in a monadic style (do . . . od, etc.) since there can be assertion failures at many different points. The w22n, w82n, n2w2, and n2w functions are conversions between bytes and natural numbers.

## 6 Setting Up Silver for CakeML

In this section, we explain how we transition from theorem (4) to theorem (6) in §2, i.e., how we prove the $\text{installed}_{\text{Ag}}$ assumption in order to move our correctness theorem from a property about CakeML's machine_sem down to Silver's Next. We make this transition by fixing a memory layout that includes code implementing the required system calls, and verifying that code. The memory layout that we use is shown in Figure 2.

Recall from §5 that machine_sem either takes an ordinary step of executing a CakeML-generated Silver instruction or takes an *interference oracle* step representing a call to a foreign function. To move from the machine_sem level to the Silver ISA level, we define a predicate, interference_impltd, which states that the effect of the interference oracle step can be obtained by normal execution of machine code located somewhere in memory (separate from CakeML-generated code). This predicate bridges the gap between machine_sem and our verification of the system call code at the ISA level.

The first theorem we prove about interference_impltd connects it to machine_sem. It states that interference_impltd $R_{\mathit{ffi}}$, for an arbitrary relation $R_{\mathit{ffi}}$ between the Silver machine state *ms* and FFI oracle state *ffi*, implies that a terminating machine_sem can be replaced by a sequence of Next steps that preserve $R_{\mathit{ffi}}$. The FFI oracle states $\mathit{ffi}$, $\mathit{ffi}'$ in this theorem are records of the same type as $\mathsf{basis\_ffi}\ \mathit{cl}\ \mathit{fs}$. Recall that $\mathsf{basis\_ffi}\ \mathit{cl}\ \mathit{fs}$ is the initial FFI oracle state from which a CakeML program is started (featuring, e.g., in theorem (6)); $\mathit{ffi}$ and $\mathit{ffi}'$ are a pair of FFI oracle states reached by the CakeML program at runtime. The $\mathit{md}$ argument gives the memory domain in which the system call code is expected to reside:

$$\begin{array}{l}\vdash \mathsf{interference\_impltd}\ R_{\mathit{ffi}}\ \mathit{md}\ \mathit{ms} \wedge R_{\mathit{ffi}}\ \mathit{ms}\ \mathit{ffi} \wedge \\ \quad \mathsf{machine\_sem}\ \mathit{ffi}\ \mathit{ms} \subseteq \\ \qquad \mathsf{extend\_with\_oom}\ \{\ \mathsf{Terminate}\ \mathsf{Success}\ \mathit{io}\ \}\ \Rightarrow \\ \qquad \exists\, k\ \mathit{ffi}'. \\ \qquad\quad R_{\mathit{ffi}}\ (\mathsf{Next}^k\ \mathit{ms})\ \mathit{ffi}' \wedge \mathsf{is\_halted}_{\mathsf{Ag}}\ (\mathsf{Next}^k\ \mathit{ms}) \wedge \\ \qquad\quad \mathit{ffi}'.\mathsf{io\_events} \preccurlyeq \mathit{io} \wedge \\ \qquad\quad (\mathsf{exit\_code\_0}_{\mathsf{Ag}}\ (\mathsf{Next}^k\ \mathit{ms}) \Rightarrow \mathit{ffi}'.\mathsf{io\_events} = \mathit{io})\end{array} \tag{11}$$

The second theorem provides a concrete relation, $\mathsf{ffi\_rel}_{\mathsf{Ag}}$, and memory domain, $\mathsf{ffi\_mem\_domain}_{\mathsf{Ag}}$, and proves that they satisfy $\mathsf{interference\_impltd}$:

$$\vdash \ldots \Rightarrow \mathsf{interference\_impltd}\ \mathsf{ffi\_rel}_{\mathsf{Ag}}\ \mathsf{ffi\_mem\_domain}_{\mathsf{Ag}}\ \mathit{ms} \tag{12}$$

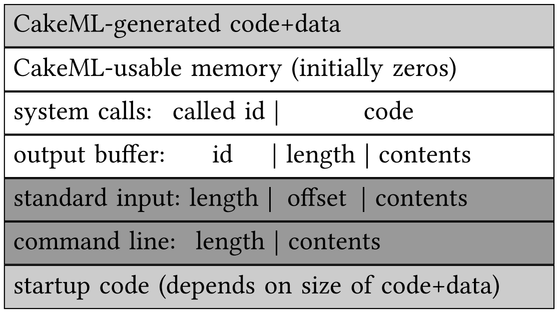
The omitted assumptions (. . .) are routine: e.g., that the FFI oracle state's command line and file system are well-formed, the code is correctly placed in memory (e.g., within the domain), and so on. We omit these routine assumptions here and below for brevity.

The proof of theorem (12) involves showing that each piece of system call code correctly implements the call as specified by $\mathsf{basis\_ffi\_oracle}$, which we saw in §5. We prove a theorem of the following form (shown here for "read") for each system call:

$$\begin{array}{l}\vdash \ldots \wedge \mathit{md} = \mathsf{prog\_mem\_domain}_{\mathsf{Ag}} \quad \ldots \wedge \mathsf{ffi\_rel}_{\mathsf{Ag}}\ \mathit{ms}\ \mathit{ffi} \wedge \\ \quad \mathsf{index\_of}\ \text{"read"}\ \mathit{ffi\_names} = \mathsf{Some}\ \mathit{index} \wedge \\ \quad \mathsf{call\_FFI}\ \mathit{ffi}\ \text{"read"}\ \mathit{conf}\ \mathit{bytes} = \mathsf{FFI\_return}\ \mathit{ffi}'\ \mathit{bytes}' \Rightarrow \\ \qquad \exists\, k. \\ \qquad\quad \mathsf{ffi\_interfer}_{\mathsf{Ag}}\ \mathit{md}\ (\mathit{index},\mathit{bytes}',\mathit{ms}) = \mathsf{Next}^k\ \mathit{ms} \wedge \\ \qquad\quad \mathsf{ffi\_rel}_{\mathsf{Ag}}\ (\mathsf{Next}^k\ \mathit{ms})\ \mathit{ffi}'\end{array} \tag{13}$$

Here, $\mathsf{call\_FFI}$ is a wrapper around $\mathsf{basis\_ffi\_oracle}$ that takes an initial FFI oracle state $\mathit{ffi}$ and returns a new FFI oracle state $\mathit{ffi}'$ along with the $\mathit{bytes}$ returned by $\mathsf{basis\_ffi\_oracle}$. The conclusion of this theorem has two parts. First, it shows that the FFI call ($\mathsf{ffi\_interfer}_{\mathsf{Ag}}$, described below) is identical to stepping $\mathsf{Next}$ $k$ times. Second, it shows that the $\mathsf{ffi\_rel}_{\mathsf{Ag}}$ relation is preserved across these $k$ steps.

So what is the $\mathsf{ffi\_interfer}_{\mathsf{Ag}}$ function? It is a concrete interference oracle instance for CakeML's FFI semantics, that specifies the effect on a machine state $\mathit{ms}$ of running a system call that returns $\mathit{bytes}'$. The $\mathit{md}$ argument indicates the memory domain ($\mathsf{prog\_mem\_domain}_{\mathsf{Ag}}$) that CakeML uses, i.e., the parts with a CakeML prefix in Figure 2. The $\mathit{index}$ argument indicates which FFI call is made (in this case "read"). The $\mathsf{ffi\_interfer}_{\mathsf{Ag}}$ function updates the machine state by writing $\mathit{bytes}'$ to the part of $\mathit{md}$ used for communicating with the external call, updating registers and the PC according to the calling convention, and, based on $\mathit{index}$, updating memory (outside of $\mathit{md}$) used for book-keeping by the external FFI call. Thus to verify each piece of system call code, we must show that executing the code has the effect of $\mathsf{ffi\_interfer}_{\mathsf{Ag}}$.



| | | | |
|---|---|---|---|
| CakeML-generated code+data | | | |
| CakeML-usable memory (initially zeros) | | | |
| system calls: | called id | code | |
| output buffer: | id | length | contents |
| standard input: | length | offset | contents |
| command line: | length | contents | |
| startup code (depends on size of code+data) | | | |

**Figure 2.** The memory layout for running CakeML programs bare-metal on Silver. When preparing the initial memory, parts with a white background are application-independent, parts with an intermediate background are application-dependent, and parts with the darkest background are input for each execution.

Each system call is verified in two refinement steps. The first step abstracts from the machine code implementing a system call to a logical specification of its effect. For example, the logical specification for the code implementing "read" is a theorem of the form:

$$\vdash \ldots \Rightarrow \exists\, k.\ \mathsf{Next}^k\ \mathit{ms} = \mathsf{ffi\_read}_{\mathsf{Ag}}\ \mathit{ms}$$

The omitted assumptions (. . .) ensure, e.g., that the relevant code and data are placed in memory correctly and that the program counter is currently pointing at the start of the code. The theorem's conclusion shows that stepping by $k$ steps yields the machine state given by $\mathsf{ffi\_read}_{\mathsf{Ag}}\ \mathit{ms}$. This logical specification ($\mathsf{ffi\_read}_{\mathsf{Ag}}$) is the glue to the second refinement step.

The second step connects $\mathsf{ffi\_read}_{\mathsf{Ag}}$ to $\mathsf{ffi\_interfer}_{\mathsf{Ag}}$ assuming the $\mathsf{ffi\_read}$ specification (§5) from CakeML's basis library:

$$\begin{array}{l}\vdash \ldots \wedge \mathsf{ffi\_read}\ \mathit{conf}\ \mathit{bytes}\ \mathit{fs} = \mathsf{FFIreturn}\ \mathit{bytes}'\ \mathit{fs}' \Rightarrow \\ \quad \mathsf{ffi\_read}_{\mathsf{Ag}}\ \mathit{ms} = \mathsf{ffi\_interfer}_{\mathsf{Ag}}\ \mathit{md}\ (\mathit{index},\mathit{bytes}',\mathit{ms})\end{array}$$

As before, the omitted assumptions (. . .) are routine ones about how the initial machine state $\mathit{ms}$ is set up. At the point of writing, both refinement steps were verified manually with the help of some specially written automation. We are confident, however, that the first step can be fully automated with decompilation tools [28].

As discussed in §5, the most complicated part of the existing CakeML assumption are the ones asserting that the system calls are correctly implemented according to CakeML's basis library assumptions. This assumption is concretely discharged in a few steps, but mainly using the composition of theorems (11) and (12) discussed in this section. Discharging this assumption is what allows us to go from theorem (4) to theorem (6) in §2. It is also in this step where the routine (omitted) assumptions from earlier are discharged. This discharging step is done automatically for concrete compiled programs such as wc_ag32. Crucially, the only remaining assumptions are the ones shown in theorem (6).

The remaining assumptions inside $\mathsf{installed}_{\mathsf{Ag}}$ are straightforward compared to the FFI ones. They concern putting the machine in an appropriate initial state for CakeML code to run. Their verification did, however, lead to some minor surprises as we detail next.

### 6.1 Changes to the Assumptions

In proving $\mathsf{installed}_{\mathsf{Ag}}$ to move from theorem (4) to theorem (6), we found that some parts of $\mathsf{installed}_{\mathsf{Ag}}$ were inconsistent, specifically point (iv) in §5 about pointers being aligned (which is independent of the ag32 target). Although the inconsistency was easy to fix, it was not caught previously and had appeared in the final top-level theorem for the CakeML compiler. This highlights the value of reducing the assumptions, ideally by proving them away, in any large formal development.

More substantially, we made some changes to CakeML's target machine semantics, machine_sem. On the one hand, the design of machine_sem that allows arbitrary interference to non-CakeML parts of the machine state whenever an external call is made, is vindicated by our instantiation of the interference oracle with a concrete implementation of system calls. On the other hand, the invariants about both CakeML steps and interference steps needed to be strengthened: we needed to know, in both cases, that memory not used by the currently running code does not change, and that the PC stays within the correct part of memory. The invariants that are now present in our definitions are sufficient for proving correctness of the sequence of calls from CakeML to external code and back on the same machine.

Finally, the extend_with_oom feature of the CakeML compiler's correctness theorem was previously (ab)used to allow compiler-generated startup checks to fail at runtime. They failed unexpectedly when we first tried running programs on Silver, because we had the memory layout and startup code slightly wrong. These dynamic checks have now all been replaced by checks that resort to a valid default configuration instead of causing runtime failures; in other words, CakeML's new startup code will never cause an out-of-memory error.

## 7 Results

The process described so far does not just work for the word-counting (wc) application. We can establish the same sorts of connection between other pieces of verified software and the verified Silver platform, creating verified software-to-hardware stacks for a variety of tools. These applications have all been verified previously: the hard intellectual work has already been performed (at the level of HOL and/or CakeML functions). Here, we confirm that the same applications can be compiled for and executed on the Silver platform.

First, we have successfully run all of the programs mentioned in the introduction (§1) on our FPGA board. Silver is not a high-performance processor, but small programs such as `sort` complete almost instantaneously when run on small inputs. Running `sort` on a 1000-line file takes a few seconds. Silver's low performance is more noticeable for larger programs, such as the compiler itself. For example, compiling a one-line hello world program on a modern Intel processor takes around two to three seconds, whereas compiling the same program on Silver takes around four hours.

Second, the verification story (establishing HOL theorems of correctness) for these applications follows the pattern already described in the paper to this point. For example, the correctness statement for the CakeML compiler on Silver (14) has much the same assumptions as for the wc example (8); that is, the machine has been correctly initialised ($\mathsf{verilog\_init}_{\mathsf{Ag}}$), and the input is not too large, among others:

$$
\begin{aligned}
&\vdash \mathsf{let}\ \mathit{vstep}\ =\ \mathsf{verilog\_sem}\ \mathit{env}\ \mathsf{silver\_cpu\_verilog}\ \mathit{init}\ \mathsf{in}\\
&\quad \mathsf{cl\_ok}\ \mathit{cl} \wedge |\mathit{input}| \le \mathsf{stdin\_size} \wedge\\
&\quad \mathsf{is\_lab\_env}\ \mathsf{acc\_env\_verilog}\ \mathit{vstep}\ \mathit{env} \wedge\\
&\quad \mathsf{verilog\_init}_{\mathsf{Ag}}\ \mathsf{compiler\_ag32}\ (\mathit{cl},\mathit{input})\ \mathit{init}\ \mathit{env} \Rightarrow\\
&\qquad \exists\, \mathit{stdout}\ \mathit{stderr}.\ \mathsf{FG}\ k.\ \exists\, \mathit{fin}.\\
&\qquad\ \ \mathsf{compiler\_spec}\ \mathit{input}\ \mathit{cl}\ \mathit{stdout}\ \mathit{stderr} \wedge\\
&\qquad\ \ \mathit{vstep}\ k = \mathsf{Ok}\ \mathit{fin} \wedge \mathsf{is\_halted}\ \mathsf{compiler\_ag32}\ \mathit{fin} \wedge\\
&\qquad\ \ \mathsf{stdout}_{\mathsf{Ag}}\ (\mathit{env}\ k).\mathsf{io\_events} \preccurlyeq \mathit{stdout} \wedge\\
&\qquad\ \ \mathsf{stderr}_{\mathsf{Ag}}\ (\mathit{env}\ k).\mathsf{io\_events} \preccurlyeq \mathit{stderr} \wedge\\
&\qquad\ \ (\mathsf{verilog\_exit\_code\_0}_{\mathsf{Ag}}\ \mathit{fin} \Rightarrow\\
&\qquad\qquad \mathsf{stdout}_{\mathsf{Ag}}\ (\mathit{env}\ k).\mathsf{io\_events} = \mathit{stdout} \wedge\\
&\qquad\qquad \mathsf{stderr}_{\mathsf{Ag}}\ (\mathit{env}\ k).\mathsf{io\_events} = \mathit{stderr})
\end{aligned}
\tag{14}
$$

In addition, because the compiler takes the name of its input file as its command-line argument, we have a predicate cl_ok asserting that the command-line is well-formed (essentially, that it is not too large). As before, the conclusion states that execution will eventually result in a final state ($\mathit{vstep}\ k = \mathsf{Ok}\ \mathit{fin}$) satisfying the user-level specification of the compiler behaviour. That specification (compiler_spec) describes how standard output contains a textual representation of the machine code for the input program.

The definition of compiler_spec makes this clear:

$$
\begin{aligned}
&\mathsf{compiler\_spec}\ \mathit{input}\ \mathit{cl}\ \mathit{stdout}\ \mathit{stderr} \overset{\text{def}}{=}\\
&\quad (\mathit{stdout},\mathit{stderr}) =\\
&\qquad \mathsf{if}\ \mathsf{has\_version\_flag}\ (\mathsf{tail}\ \mathit{cl})\ \mathsf{then}\\
&\qquad\quad (\mathsf{explode}\ \mathsf{current\_build\_info\_str},\text{“”})\\
&\qquad \mathsf{else}\\
&\qquad\quad \mathsf{let}\ (\mathit{cout},\mathit{cerr})\ =\ \mathsf{compile\_32}\ (\mathsf{tail}\ \mathit{cl})\ \mathit{input}\ \mathsf{in}\\
&\qquad\qquad (\mathsf{explode}\ (\mathsf{concat}\ (\mathsf{append}\ \mathit{cout})),\mathsf{explode}\ \mathit{cerr})
\end{aligned}
$$

The compile_32 function mentioned here is a (somewhat complicated) wrapper around a call to the compile function of our initial correctness result for the compiler (2). In this way, our theorem asserts the correctness of the bootstrap of CakeML on Silver.

## 8 Discussion

The promise of a verified stack is the ability to construct systems that have formal evidence for their correct implementation. Such evidence, in the form of mechanically checked proofs, is always subject to implicit and explicit assumptions, which collectively represent the *trusted computing base* (TCB) of the verified stack. The TCB is all the things that need to be trusted if we are to believe the stack operates correctly. In stack constructions, alongside the proof checker itself, only the top and bottom layers contribute to the TCB since there are proofs in between.

In this section, we describe the TCB of the bottom layer of stacks constructed using our methodology and discuss the (necessarily non-formal) ways in which we can justify the trust we put into these assumptions. We also show where we have reduced the TCB, compared to previous CakeML work, by replacing assumptions with proofs.

***Verilog semantics.*** We assume that our formal model of Verilog is accurate with respect to the Vivado toolchain that takes Verilog input and produces our hardware. Relatedly, we assume that:

- the printing of Verilog abstract syntax trees from HOL is faithful; and
- the Vivado toolchain taking Verilog to FPGA bitstreams is bug-free.

We address these assumptions by using a simple subset of Verilog, one where the semantics is uncontroversial, and where we can be relatively confident that the implementation will be straightforward. Code implementing the pretty-printing of ASTs is not complicated, so informal code inspection is helpful with respect to this assumption. Though we have not done this, we could gain assurance by implementing this code in CakeML, developing a parser for the printed syntax, and proving that the composition of parser and printer is the identity. The second item could be further addressed by standard industrial tools such as formal equivalence checkers, but such tools would not produce proofs composable with our formal development.

***Hardware.*** In our lab setup (§4.2), we have aimed for convenience rather than a minimal TCB. Consequently, some of the assumptions required by the current lab setup could be significantly reduced with little effort. Concretely, we are dependent on both the correct operation and the correct initialisation of the various hardware components that realise our final system. For example, we assume that:

- the FPGA chip works correctly;
- the shared DRAM module (and other board modules) works correctly;
- the Python script, running on the ARM core, that we use to pre-load memory and handle interrupts (such as text output requests) sent to the core is operating correctly; and
- the Verilog glue code used to connect the Silver processor to its environment works correctly. E.g., some interfaces, such as to the DRAM module, are exposed as AXI3 interfaces [24]—but as we are not interested in the details of AXI3 in particular, we expose simplified interfaces to the processor.

The dependence on the ARM core (and the Linux operating system it is running) for pre-loading memory and interrupt handling is clearly tangential, and could be improved by pre-loading memory by more primitive means and using, e.g., seven-segment displays for text output.

***Comparison to previous work.*** The bottom-layer TCB described above is different to, and a clear improvement on, the bottom-layer TCB accompanying verified software developed with CakeML previously. The assumptions about Verilog and the hardware have replaced enormous, unverified components. In particular, previous work [14, 21] had to assume:

- the correctness of the underlying operating system and its tools to link and run our executables (loading it into memory, connecting it to I/O streams, etc.);
- the correctness of our hardware semantics for targets such as ARM and x86; and
- the correctness of those hardware semantics' realisation in the silicon on which the software was being executed.

## 9 Related Work

**The CLI stack.** An early attempt at constructing a verified stack was made in the late 1980s and early 1990s in the CLI stack project [4, 33], which was built using the Nqthm theorem prover, a precursor to ACL2. The stack included, among other components, a verified processor and two verified compilers, for Pascal-like and Lisp-based languages, targeting the verified processor. A version of the stack was built for the verified FM9001 processor. FM9001 was described in a custom HDL called DUAL-EVAL, which was translated to LSI Logic's Netlist Description Language for fabrication by LSI Logic on a gate-array [8].

Moore [33], one of the stack's principal architects, describes the compilers' languages as "too simple to be of practical use", lacking e.g., I/O mechanisms. Furthermore, it was not possible to run the verified compilers on top of their stack.

**The Verisoft stack.** A later attempt at a verified stack was made in the 2000s in the Isabelle/HOL-based Verisoft stack project [2]. The processor used in the Verisoft stack is called VAMP, first developed in PVS [6], and later ported to Isabelle/HOL [36]. Beyer et al. [6] call the CLI stack's FM9001 processor "very simple" and note that the VAMP is much more complex as it is both pipelined and capable of out-of-order instruction execution. In comparison with the VAMP processor, our Silver processor must also be described as "very simple". On the other hand, the VAMP processor was also synthesised for FPGAs, but was not verified down to the Verilog code used for synthesis. Instead, for the PVS version, a tool operating outside the formal development called pvs2hdl [5] was used to produce the Verilog code out of a gate-level PVS description. Similarly, i.e., also without proof, the Isabelle/HOL VAMP version used an unverified tool called IHaVeIt [36] to translate Isabelle/HOL hardware descriptions to Verilog.

The Verisoft stack also included a verified compiler for C0 [22], a language similar to a subset of C plus garbage collection. The C0 compiler provides similar FFI functionality as the CakeML compiler, called XCalls, allowing programmers to embed VAMP assembly code inside C0 programs. Leinenbach and Petrova [22] describe the compiler as "simple". In contrast with CakeML, it does not include any optimisation passes. The C0 compiler consists of a verified compilation algorithm accompanied by a partly verified C0 implementation. Unlike the CakeML compiler, the implementation is not automatically derived from the compilation algorithm. Instead, a Hoare-logic based C0 verification environment was used to prove the implementation correct. This means that manual work is needed to keep the compilation algorithm and implementation in sync when new features are added to the compiler. Moreover, only the code generation implementation (approximately 1500 lines of C0) was proved correct; parsing and I/O were left unverified. A VAMP machine code implementation is needed to run the compiler on top of the VAMP processor, but they do not provide a way to compile, with proofs, the C0 implementation to a machine code implementation (such as running the C0 compiler in-logic, as the CakeML compiler does when compiling itself to machine code). In other words, not all pieces for running the compiler on top of the VAMP stack are present.

**Other verified stack work.** In the ongoing Coq-based DeepSpec stack project [3], the Kami project [11] enables Bluespec development and verification inside Coq. A pipelined in-order multicore processor has been developed inside the Kami project as a case study, but is not yet part of a larger stack. The Coq world's analogue to the CakeML compiler, the CompCert compiler [23], does not have a implementation verified down to machine code, so obtaining a correctness guarantee about running CompCert on top of a Coq-verified processor is non-trivial (as doing this requires having access to a verified machine code implementation).

There have also been processors developed and (sometimes partly) verified without being part of full-stack projects. Though such components might fit into a verified stack, without actually carrying out the necessary integrations, this remains a "might" rather than a demonstrated "can". Beyer et al. [6] enumerate a few verified processors published before their PVS VAMP paper, and note that of the processor papers they cite, only papers about the FM9001 processor (i.e., the processor from the CLI stack) state that the processor has been synthesized. By their account, the remaining processor papers rely on "several simplifications and abstractions". Given the controversies around the Viper processor [12, 26], it is clear that when claiming a processor "verified", one must be precise about what has actually been proved, and down to what abstraction level the proofs reach.

**Correct hardware.** Neither the stack work cited in this section, nor other ITP hardware verification work [7, 11, 17, 19, 31], has combined the verification with a formal semantics for a mainstream low-level HDL such as Verilog or VHDL (instead relying on, e.g., unverified extraction). Previous formal semantics work exists for Verilog [20, 27], but those projects do not seriously consider ITP verification.

**Verified low-level systems code.** Unlike high-level application code which can be compiled by the CakeML compiler down to Silver machine code, we implemented and verified the system calls for our stack by hand. This was manageable for the CakeML basis library but verifying more complicated system calls would require (or at least, be significantly aided by) low-level programming and verification frameworks [9, 10, 30] and automated decompilation tools [28].

## 10 Conclusion

This paper has reported on a novel workflow for producing verified stacks that connect verified hardware to verified programs that run on top of it. Our approach connects the CakeML compiler to a new verified Silver processor. We have a unique and novel contribution, which enables the proof of *single end-to-end correctness theorems* for realistic user-level programs, such as the CakeML compiler itself. In other words, not only does the CakeML compiler have a new target, the verified Silver hardware design, but it can itself be run on that hardware. This work is a relief: we now know that the assumptions made at the bottom of the CakeML compiler proof can be met by underlying verified hardware.

At certain points, we have taken the shortest route to our final end-to-end results, which means that there is room for improvement in individual parts of the project. Improvements of one part can be carried out independently of other parts as long as the interfaces between all parts stay the same.

We intend to improve our hardware implementation of Silver. The processor ought to be pipelined and otherwise optimised to support higher clock frequencies in order to produce faster applications. We will do this without fundamentally changing the Silver ISA because we want to keep the ISA at an abstraction level that does not expose implementation techniques in the hardware implementation.

We also want to make it less labour-intensive to develop and set up verified systems code that interfaces with the CakeML generated code. The set up work required for this paper was significantly more labour-intensive than expected.

***Acknowledgements.*** This work was partly supported by the Swedish Foundation for Strategic Research.

## References

[1] 2018. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018). https://doi.org/10.1109/IEEESTD.2018.8299595

[2] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. 2008. The Verisoft Approach to Systems Verification. In *Verified Software: Theories, Tools, Experiments (VSTTE)*. https://doi.org/10.1007/978-3-540-87873-5_18

[3] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017). https://doi.org/10.1098/rsta.2016.0331

[4] William R. Bevier, Warren A. Hunt, J Strother Moore, and William D. Young. 1989. An approach to systems verification. *Journal of Automated Reasoning* 5, 4 (1989). https://doi.org/10.1007/BF00243131

[5] Sven Beyer, Christian Jacobi, Daniel Kroening, and Dirk Leinenbach. 2002. *Correct Hardware by Synthesis from PVS*. Technical Report. http://www-wjp.cs.uni-sb.de/publikationen/BJKL02.pdf

[6] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. 2006. Putting it all together – Formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer (STTT)* 8, 4 (2006). https://doi.org/10.1007/s10009-006-0204-6

[7] Thomas Braibant and Adam Chlipala. 2013. Formal Verification of Hardware Synthesis. In *Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-642-39799-8_14

[8] Bishop C. Brock and Warren A. Hunt. 1997. The DUAL-EVAL Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor. *Formal Methods in System Design* 11, 1 (1997). https://doi.org/10.1023/A:1008685826293

[9] Adam Chlipala. 2011. Mostly-automated Verification of Low-level Programs in Computational Separation Logic. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1993498.1993526

[10] Adam Chlipala. 2013. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2500365.2500592

[11] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-level Parametric Hardware Specification and Its Modular Verification. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017). https://doi.org/10.1145/3110268

[12] Avra Cohn. 1989. The notion of proof in hardware verification. *Journal of Automated Reasoning* 5, 2 (1989). https://doi.org/10.1007/bf00243000

[13] Anthony Fox. 2012. Directions in ISA Specification. In *Interactive Theorem Proving (ITP)*. https://doi.org/10.1007/978-3-642-32347-8_23

[14] Anthony Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. 2017. Verified Compilation of CakeML to Multiple Machine-code Targets. In *Certified Programs and Proofs (CPP)*. https://doi.org/10.1145/3018610.3018621

[15] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. 2017. Verified Characteristic Formulae for CakeML. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-662-54434-1_22

[16] Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. 2018. Proof-Producing Synthesis of CakeML with I/O and Local State from Monadic HOL Functions. In *International Joint Conference on Automated Reasoning (IJCAR)*. https://doi.org/10.1007/978-3-319-94205-6_42

[17] Warren A. Hunt, Matt Kaufmann, J Strother Moore, and Anna Slobodova. 2017. Industrial hardware and software verification with ACL2. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017). https://doi.org/10.1098/rsta.2015.0399

[18] Joe Hurd. 2011. The OpenTheory Standard Theory Library. In *NASA Formal Methods (NFM)*. https://doi.org/10.1007/978-3-642-20398-5_14

[19] Juliano Iyoda. 2007. *Translating HOL functions to hardware*. Technical Report UCAM-CL-TR-682. University of Cambridge, Computer Laboratory.

[20] Wilayat Khan, Alwen Tiu, and David Sanán. 2017. VeriFormal: An Executable Formal Model of a Hardware Description Language. In *Singapore Cyber-Security RandD Conference (SG-CRC)*. https://doi.org/10.3233/978-1-61499-744-3-19

[21] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. 2018. Software Verification with ITPs Should Use Binary Code Extraction to Reduce the TCB (Short Paper). In *Interactive Theorem Proving (ITP)*. https://doi.org/10.1007/978-3-319-94821-8_21

[22] Dirk Leinenbach and Elena Petrova. 2008. Pervasive Compiler Verification – From Verified Programs to Verified Systems. *Electronic Notes in Theoretical Computer Science* 217 (2008). https://doi.org/10.1016/j.entcs.2008.06.040

[23] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Communications of the ACM* 52, 7 (2009). https://doi.org/10.1145/1538788.1538814

[24] Arm Limited. 2017. *AMBA AXI and ACE Protocol Specification*. Technical Report ARM IHI 0022F.b.

[25] Andreas Lööw and Magnus O. Myreen. 2019. A Proof-Producing Translator for Verilog Development in HOL. In *Formal Methods in Software Engineering (FormaliSE)*. To appear.

[26] Donald MacKenzie. 1991. The fangs of the VIPER. *Nature* 352, 6335 (1991). https://doi.org/10.1038/352467a0

[27] Patrick Meredith, Michael Katelman, José Meseguer, and Grigore Roşu. 2010. A formal executable semantics of Verilog. In *Formal Methods and Models for Codesign (MEMOCODE)*. https://doi.org/10.1109/MEMCOD.2010.5558634

[28] Magnus O. Myreen. 2009. *Formal verification of machine-code programs*. Ph.D. Dissertation. University of Cambridge.

[29] Magnus O. Myreen and Scott Owens. 2014. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming* 24, 2-3 (2014). https://doi.org/10.1017/S0956796813000282

[30] Zhaozhong Ni and Zhong Shao. 2006. Certified Assembly Programming with Embedded Code Pointers. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1111037.1111066

[31] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. 2018. Pi-Ware: Hardware Description and Verification in Agda. In *Types for Proofs and Programs (TYPES 2015)*. https://doi.org/10.4230/LIPIcs.TYPES.2015.9

[32] Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs)*. https://doi.org/10.1007/978-3-540-71067-7_6

[33] J Strother Moore. 2003. A Grand Challenge Proposal for Formal Methods: A Verified Stack. In *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST*. https://doi.org/10.1007/978-3-540-40007-3_11

[34] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. 2016. A new verified compiler backend for CakeML. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2951913.2951924

[35] Chuck Thacker. 2007. A Tiny Computer. (2007). Unpublished memo, available online.

[36] Sergey Tverdyshev. 2009. *Formal Verification of Gate-Level Computer Systems*. Ph.D. Dissertation. Saarland University.