Integration Verification across Software and Hardware for a Simple Embedded System

Andres Erbsen*
Samuel Gruetter*
Joonwon Choi
Clark Wood
Adam Chlipala
MIT CSAIL
USA

Abstract

The interfaces between layers of a system are susceptible to bugs if developers of adjacent layers proceed under subtly different assumptions. Formal verification of two layers against the same formal model of the interface between them can be used to shake out these bugs. Doing so for every interface in the system can, in principle, yield unparalleled assurance of the correctness and security of the system as a whole. However, there have been remarkably few efforts that carry out this exercise, and all of them have simplified the task by restricting interactivity of the application, inventing new simplified instruction sets, and using unrealistic input and output mechanisms. We report on the first verification of a realistic embedded system, with its application software, device drivers, compiler, and RISC-V processor represented inside the Coq proof assistant as one mathematical object, with a machine-checked proof of functional correctness. A key challenge is structuring the proof modularly, so that further refinement of the components or expansion of the system can proceed without revisiting the rest of the system.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Hardware → Theorem proving and SAT solving**.

Keywords: Formal Verification, Hardware-Software Interface, Proof Assistants, Embedded Systems, RISC-V Instruction-Set Family

*equal contribution

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454065

ACM Reference Format:
Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification across Software and Hardware for a Simple Embedded System. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454065

1 Introduction

We present the comprehensive and modular verification of functional correctness of a newly realistic but still very simple embedded system, highlighting important challenges that remain in scaling up the scope and realism of verification and reducing the effort required. Our development of an Ethernet-connected IoT lightbulb controller culminates in a single Coq proof relating the network packets entering our integrated system through memory-mapped I/O (MMIO) to the action the controller takes by emitting MMIO writes, ruling out any bugs or vulnerabilities that could be exploited over the network. In particular, the proof spans a pipelined processor implementation, the RISC-V instruction set, a compiler, a software-verification system, drivers, and application code. We choose rather simple designs for the components and mainly focus on *integration verification*, i.e. on ruling out integration bugs that arise when two components (e.g., compiler and application code, or compiler and processor) interpret the interface between them in independently reasonable but subtly different ways.

We believe that verifying intercomponent interaction is crucial for preventing the nastiest bugs across the stack, e.g.:

- A network interface card receiving a large frame overrunning a statically allocated buffer in the driver (our initial prototype had this bug)
- A C compiler deleting reasonable-looking code that calls `memcpy` after determining that a little-known and unnecessary precondition of the function is violated [26]
- Code compiled for a userland environment failing inexplicably in a kernel due to differences in stack discipline [14]
- Code written for AMD processors allowing for privilege escalation attacks on Intel's "compatible" processors due

to a subtle difference in the behavior of an instruction originally defined by AMD, even though each implementation technically matched its published specification [13]

We further insist on *modular verification*: it should be possible to modify and optimize each component individually (within the flexibility allowed by its interfaces) and reprove it without having to recheck and potentially update the proofs of other components. For example, the compatibility between a new processor and established instruction set could be proven without access to all code for that instruction set, or more ambitiously, optimizations added to a compiler could be verified against the same spec as the unoptimized version.

The main challenge in this work was to come up with appropriately precise interface specifications. Each of them needs to delineate the responsibilities of the components on the two sides of the interface sufficiently, so that all components can be verified individually against their interfaces and yet compose into an *end-to-end theorem stated without referencing any of the intermediate specifications*. We believe this rigor about interfaces is necessary and useful even though we could have built a full-stack proof for our particular application with less effort – and for reasons discussed in section 7.3.1, we chose to build the smallest application that crosses the interfaces we wanted to study.

Concretely, the I/O of our IoT lightbulb demo is described by a simple regex-like expression (see section 3.1) that becomes the top-level specification for the system of software and hardware together:

```
BootSeq +++ ((EX b: bool, Recv b +++ LightbulbCmd b)
             ||| RecvInvalid ||| PollNone) ^*
```

Contributions. To our knowledge, no prior verification-integration project spanning software and hardware has involved *unbounded reactive execution*, *interactive behaviors*, or *realistic I/O* interfaces like MMIO. Treatment of I/O in a software/hardware-integration proof is especially challenging to incorporate into our modular design, because hardware optimizations like pipelining and instruction caching require additional validity conditions from the software, but these conditions cannot be stated in terms of the concepts of any one layer and need to span the entire stack. We describe intricate specifications of intercomponent interfaces that together guarantee predictable execution without exposing undue detail and which are parameterized over the underlying I/O mechanism (MMIO in the demo).

Semantics of our system's internal layers from source code to assembly are written in a novel manner that we call CPS semantics. We found it particularly convenient for control-flow-directed "forward" reasoning even in the presence of undefined behavior, external input (or any nondeterminism), and potentially divergent computations.

Additionally, ours is the first applications-to-hardware integration-verification project to use an ISA supported by commercial off-the-shelf processors (concretely, with the RISC-V ISA). This choice allowed us to build our first prototype by buying a commercial microcontroller and implementing its software stack with off-the-shelf tools; then we experimented separately with replacing each of the hardware and software parts with a verified version, testing it against the mainstream version of the other side. We also felt this baseline of realism was important to keep us from cutting corners.

Availability. All code and mechanized proofs in this project are available under a permissive open-source license at

https://github.com/mit-plv/bedrock2/

Structure of the Paper. The remainder of this paper is structured as follows: section 2 reviews prior work, also taking the opportunity to define a few important concepts. Section 3 gives a brief overview of our system. Section 4 introduces our application programming language and the style of semantics used throughout the system. In section 5, we describe the layers of our stack, the interfaces between them, and our verification that they adhere to these interfaces, culminating in the presentation of our end-to-end theorem describing the behavior of the overall system concisely and with formal accuracy. While section 5 focuses on the *vertical* modular decomposition of the system into layers, section 6 discusses the *horizontal* modular decomposition achieved by parameterization throughout layers. Section 7 discusses the engineering effort and other evaluation criteria, and section 8 concludes.

2 Related Work and Concepts

2.1 Integration Verification

When integrating two components, we need to ensure that they make the same assumptions about the interface between them. Our strategy is to write down the interface specification in a format that is both human- and machine-readable and to verify that both components adhere to it.

Successful examples of such work include the Verified Software Toolchain (VST) [3], which verifies C programs against logical specifications in Coq, compiles these programs using CompCert [27], and achieves verified integration of the C programs and the compiler because both of them use the exact same specification of the C language written in Coq.

Similarly (but with a different, abstraction-layer-based proof approach), the CertiKOS [19] verified operating system implemented in C integrates in a verified way with a fork [18] of the CompCert compiler, and recent encouraging work [24, 30] is trying to integrate it with VST.

2.2 Automated Symbolic Execution

A promising approach for integration verification that is more automated than the projects just surveyed and ours

Table 1. Our evaluation criteria for verified stacks

Key:
✓ met
~ partially met
✗ not met
– not applicable

	seL4 [23]	VST+CertiKOS [30]	CompCertMC [41]	Everest [6]	Serval [33]	Vigor [43]	CLI stack [5]	Verisoft [2]	CakeML [25, 29]	This paper
Applications										
OS and/or drivers										
Source language										
Assembly										
Machine code										
HDL										
Integration verification	~	~	✓	✗	✓	✓	✓	✓	✓	✓
One proof assistant	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓
Modularity	~	✓	✓	~	✗	✓	✓	✓	✓	✓
Standardized ISA	✓	✓	✓	✓	✓	✓	~	✗	✗	✓
HW optimizations	–	–	–	–	–	–	~	✓	✗	✓
Realistic I/O	✓	~	✗	✗	~	✓	✗	~	✗	✓

is to use SMT solvers and symbolic execution. Solvers are used to test reachability of unexplored control-flow paths, and this symbolic analysis can eventually certify that all relevant paths have been explored in full generality. Examples of verification tools built in this style include in the Hyperkernel project [34], the Nickel information-flow-checking tool [38], Serval [33], and Vigor [43]. Several of these tools apply directly to assembly or machine languages, solving the integration-verification problem between the source language and the compiler.

While these approaches offer a high degree of automation, they rely on knowing simultaneously the *implementations* of all components being verified as compatible with each other. We are not aware of past work of this kind that crosses the software-hardware boundary, e.g. using symbolic execution to realize that a tricky conditional in a processor went a certain way, so that we should find a software-level test vector exercising the other case. Perhaps more importantly, the inherently unmodular nature of analysis fails to reap the benefits associated with classic techniques in Hoare logic and elsewhere, where we can modify one component of a system without needing to adjust the proofs of others. There are examples of mixing the two approaches, like how Vigor [43] uses Hoare-logic-style proofs of important library routines to summarize them soundly in symbolic execution, though such hybrids suffer from larger verification trusted code bases, typically with no proof of the interface between the symbolic executor and the Hoare-logic-style tool.

2.3 Height of the Verified Stack

As every layer of a software stack could contain bugs, it is desirable that the verification effort spans a stack height as large as possible (visualized in Table 1).

When it comes to starting the verification as high-up as possible, a notable project is Everest [6], which develops a TLS stack and an appropriate set of cryptographic primitives using a number of SMT-based tools. And when it comes to ending the verification as low as possible in the stack, another project worth mentioning is CompCertMC [41], which extends CompCert [27] to compile to machine code running on a realistic machine model rather than compiling just to an assembly language with pseudo-instructions and a machine model with an unbounded stack. However, we have not yet seen this work being integrated with projects building on top of CompCert, such as VST and CertiKOS.

When thinking about extending the verified stack at the bottom, the interface between software and hardware is both important and subtle. The question is not just "what if the hardware contains bugs?" but crucially also "what if the software and the hardware make different assumptions about how the instructions should behave?"

2.4 Verified Software-Hardware Integration

We are aware of three prior projects that achieve *integration verification* across the software-hardware boundary. They all do so by connecting all components within one proof assistant, thus reducing the trusted audit-worthy code base to just their top-most and bottom-most specifications and the proof assistant.

In the late 1980s, the CLI stack [5] connected a Pascal-like language to a 32-bit microprocessor design described in minimalistic register-transfer language. The purpose-built languages were modeled using interpreters and omitted input or output facilities. The processor implementation is described as a loop that executes one instruction per iteration and includes, for example, waiting for responses to memory requests [21]. The verified software for this stack included arithmetic on large integers and a solver for the mathematical game Nim, and a successor of the processor was fabricated using gate-array technology.

The Verisoft project [2], begun in the early 2000s, connects a correctness framework for programs written in a language they call C0 to a compiler targeting their purpose-built VAMP processor architecture. To our knowledge, no complete physical demonstration system including input and output was ever built with this stack, and we also are not aware of any full-system proof against a concise application specification in terms of input and output. The closest we are aware of related a correctness proof of a small automotive-control C0 application to the correctness proof of an operating system [12], plugging into a proved stack including compiler and processor, but there is no discussion of a short full-system

theorem, even though each interface individually seems to have been crossed for non-I/O code [40].

In work begun roughly 15 years after the Verisoft project started, the CakeML optimizing compiler [25] was extended with a backend to a new, purpose-built instruction set called Silver [29]. This time the software stack did support input and output, but the complete stack still did not. Instead, external calls for file-system access and standard input/output were compiled into reads and writes of a memory buffer. The stack was run on an FPGA, with a commodity microprocessor connected to the same memory to initialize input and collect output (in contrast to our experiments using a freestanding system). With this setup, several nontrivial programs were executed: word count, sorting, and even compiling a "hello word" program using a cross-compiled copy of CakeML itself. Application software was written in an ML dialect, while our project involves low-level software written in a C-like language and proved functionally correct, hewing more closely to typical practices in embedded systems.

Unfortunately, none of the three above projects performs what we call *realistic I/O*: CLI and Silver only provide end-to-end proofs about values written into the main memory, and Verisoft I/O proofs do not cross the hardware-software interface [1]. Moreover, each of them uses its custom-built ISA instead of a *standardized ISA*, raising the question of whether integration techniques are up to the challenge of realistic architectures.

3 Overview

This project provides a simple embedded-systems stack including software as well as hardware for building single-threaded applications that do not require an operating system and communicate with the external world over Ethernet. Each of its components is developed in the Coq proof assistant and comes with a specification that is both human-readable and machine-readable. As a result, we can write machine-checked end-to-end proofs about the I/O behavior of a system, where all the intermediate specifications cancel out, resulting in a concise description of the system's behavior stated in terms of just the highest-level application logic and lowest-level hardware model.

Figure 1 gives an overview of our system. The software is written in a minimal C-like language of our devising, Bedrock2. The compiler can be fed a source program and be executed inside Coq to create a RISC-V binary, and the processor written in the Kami framework [10] can be exported to a design in the Bluespec HDL[1]. (This automatic translation to a language outside Coq is where our trust structure bottoms out, handing over control to tools that are not verified.) Using the Bluespec compiler, our design is compiled to Verilog and then synthesized onto an FPGA, whose BRAM memory is initialized with the RISC-V binary.

[1] https://github.com/B-Lang-org/bsc

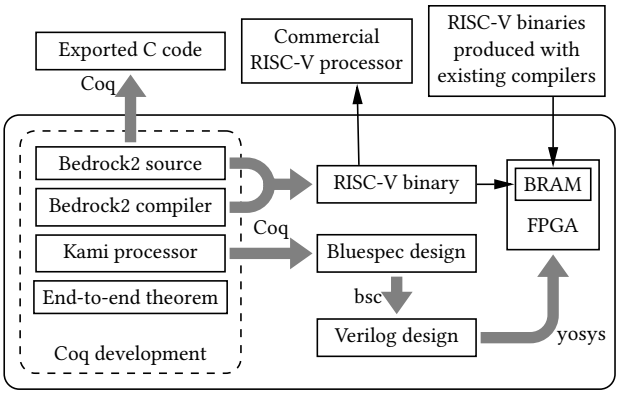

Figure 1. System overview. The top row highlights compatibility with existing interfaces and tools.

While this paper focuses on the components inside the large box in Figure 1, it is worth emphasizing that the system adheres to existing interfaces where it makes sense, as illustrated by the arrows crossing the boundary of the large box: RISC-V binaries compiled with other compilers can be run on the Kami-generated processor, RISC-V binaries compiled with the Bedrock2 compiler can be run on commercial RISC-V processors, and Bedrock2 source programs can be exported to C code. However, there is an existing interface, the C language, that we take inspiration from but do not adhere to strictly, similarly to other verification projects [17, 20, 34]. Using Bedrock2 instead of C was expedient both in that it allowed us to skip implementing unneeded features and to avoid choosing and defending an interpretation of contentious points in C semantics [16, 31]. Importing C code into Bedrock2 is thus not supported in general, but manual translation of our embedded-systems code proved straightforward.

While this system could be used for any simple application, this paper focuses on one specific example we call *the verified IoT lightbulb*. In this example, the FPGA running the verified system is connected to a network interface card and to a power switch controlling a lightbulb, as shown in Figure 2. The only functionality of the application running on the FPGA is to read UDP packets from the network interface card and turn the lightbulb on or off depending on the first byte of the received packet. Any unexpected packet, no matter how maliciously malformed at any layer, is ignored, and the application does not send any packets to the network card. This guarantee is important despite the simplicity of the application: confusing a word count for a byte count led to an unprovable Coq goal during the development of our Ethernet driver. The nature of the issue was quickly confirmed by exploiting the bad check to grow the heap into the stack and gain remote code execution on the development system.

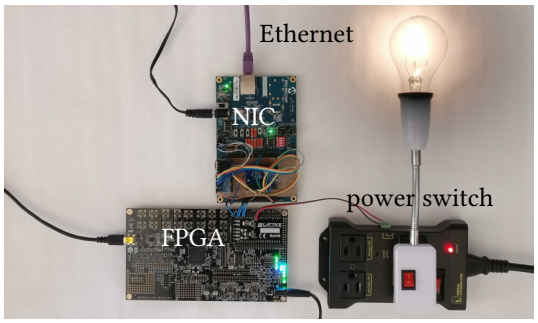

Figure 2. System demo

It is important that this property can not only be written down and checked against our implementation (which we have done) but also that the statement of this property is easy to audit, because it is expressed concisely, as we show in the following subsection.

3.1 The Trace Predicate

We state application-level specifications as predicates over traces of the MMIO reads and writes issued by the processor. An MMIO trace is a list of triples, where `("ld", addr, value)` means that the system issued an MMIO-load request with address `addr` on the memory bus and got `value` as the reply, and `("st", addr, value)` means that the system issued an MMIO-store request with address `addr` and value `value`.

Our specifications stand for sets of legal I/O traces.

For readability, we write them in the style of regular expressions, with notation `|||` for union, `+++` for concatenation, and `^*` for zero or more repetitions. However, our trace predicates are general logical functions over traces, so we retain the full expressive power of higher-order logic. For instance, we can define a notation (where `a` may occur free in `P`)

```
EX a:T, P := (λ t:trace ⇒ ∃ a:T, P(t))
```

which means that there exists an `a` of type `T` such that the trace satisfies `P`. Our top-level spec is named `goodHlTrace` ("good high-level trace") and defined as

```
Definition goodHlTrace :=
 BootSeq +++ ((EX b: bool, Recv b +++ LightbulbCmd b)
              ||| RecvInvalid ||| PollNone) ^*.
```

Every trace accepted by `goodHlTrace` starts with a series of incantations `BootSeq` mandated by the Ethernet controller. After that, `goodHlTrace` requires that the trace only consists of three kinds of interactions: receiving a valid UDP packet containing a Boolean `b` (`Recv b`) followed by turning the lightbulb on or off depending on the value of `b` (`LightbulbCmd b`); or silently ignoring an invalid packet (`RecvInvalid`); or polling the Ethernet card for a new packet but getting the response that there is none (`PollNone`).

The subspecifications (`BootSeq`, etc.) are defined similarly along with a simple (and lax) specification of byte strings accepted as Ethernet and UDP packets. All the above take up less than a page of code and form our application-level promise to the user, which we prove all the way down to the level of particular hardware designs with memory initialized with concrete bytes.

4 CPS Semantics

The Bedrock2 compiler uses a particular style of semantics that we call *CPS semantics*. CPS semantics are compatible with the weakest-precondition semantics of our program logic as well as with the traditional small-step operational semantics of Kami, but they offer the advantage that they enable forward-style compiler-correctness proofs even in the presence of external and internal nondeterminism.[2]

Forward-style compiler-correctness proofs [28] associate to each successful source-language execution a successful target-language execution. With traditional small-step or big-step operational semantics, forward-style proofs only work for external nondeterminism and become (almost) useless once internal nondeterminism is added to the semantics, because they cannot exclude that the target program has unwanted behaviors that differ from those chosen by the compiler-correctness proof, and one has to use backward-style proofs instead, which prove that for each target-language execution, a corresponding source-language execution exists. However, backward-style proofs are much more tedious, and experts avoid them whenever possible [28, 36], because they require a more-detailed simulation relation that considers each intermediate target-language state for the case where one source-language instruction is translated to several target-language instructions.

On the other hand, derivations in CPS semantics, as we will see, talk about all possible executions at once and therefore do not suffer from this problem. A forward proof between CPS-semantics derivations says that if all source-program executions are successful then all executions of the compiled code are successful.

We start by presenting the weakest-precondition definition of the program logic, so that we can contrast CPS semantics to it in the next subsection.

4.1 The Bedrock2 Program Logic

Bedrock2 programs, such as the application and driver code of the lightbulb, are proven against a *verification-condition generator*, which serves as the top-level specification for this language. It takes as arguments the program c, the trace of past external calls t, the Bedrock2-owned memory m, the values of the local variables ℓ, and a claimed postcondition Q

[2] We say that a labeled state transition system has *internal* nondeterminism if it has states that can step to several possible next states without reading any input to decide which state to pick, and we speak of *external* nondeterminism if different inputs cause a given state to step to different next states. We believe this terminology is consistent with CompCert [28].

which itself is a predicate on trace, memory, and locals:

$$\mathsf{vcgen}(c, t, m, \ell, Q)$$

When applied to a program c, vcgen answers the question "what needs to be proven to know that executing statement c from state (t, m, ℓ) always terminates in states satisfying Q?", i.e. it returns the *weakest precondition* that must hold before executing c if we want Q to hold after executing c. At the beginning of verification of each function, vcgen is invoked on universally quantified inputs, so for each function with body c, precondition P, and postcondition Q, we prove

$$\forall\ t\ m\ \ell.\ P(t, m, \ell) \Rightarrow \mathsf{vcgen}(c, t, m, \ell, Q)$$

The definition of vcgen is structurally recursive on the program and handles most cases very similarly to a hypothetical continuation-passing-style (CPS) interpreter for Bedrock2 programs, except in the loop case it asks for a loop invariant and a decreasing measure instead of unrolling the loop. For example, the sequence case $\mathsf{vcgen}(c_1; c_2, t, m, \ell, Q)$ returns

$$\mathsf{vcgen}\ (c_1, t, m, \ell,\ (\lambda t' m' \ell'.\ \mathsf{vcgen}(c_2, t', m', \ell', Q)))$$

4.2 Induction on all Executions

For the compiler-correctness proof, we need semantics that allow us to write proofs by induction on the execution (rather than the structure) of a program, so we cannot directly use the above vcgen semantics. *The CPS semantics is obtained by translating the weakest-precondition generator vcgen into an inductively defined relation $(c, t, m, \ell) \Downarrow Q$ between starting states and postconditions while maintaining the CPS form.* In order to make sure our top-level theorem does not depend on this semantics that is not (yet) well-established in the community, we prove in section 5.8 that it agrees with traditional small-step semantics. A small tweak is needed to make the sequence and loop cases pass the strict-positivity requirement of inductive definitions: we "bake in" the weakening rule of Hoare logic to avoid invoking the inductive under binders in its own postcondition position (Q_1 below). We see that, having removed the structural-recursion requirement, the loop case can be specified in a step-by-step fashion:

$$\frac{\mathsf{expr\_evaluates}(e, m, \ell, 0) \quad Q(t, m, \ell)}{(\texttt{while}(e)c, t, m, \ell) \Downarrow Q}$$

$$\frac{\begin{array}{c}\mathsf{expr\_evaluates}(e, m, \ell, v) \quad v \neq 0 \quad (c, t, m, \ell) \Downarrow Q_1 \\ \forall\ t'\ m'\ \ell'.\ Q_1(t', m', \ell') \Rightarrow (\texttt{while}(e)c, t', m', \ell') \Downarrow Q_2\end{array}}{(\texttt{while}(e)c, t, m, \ell) \Downarrow Q_2}$$

A (potentially infinitary) derivation tree of this relation is a step-by-step explanation of how *all* possible executions of this program terminate in states satisfying the postcondition. Following this intuition, both failure and nondeterminism can be modeled straightforwardly: if any of the possible nondeterministic execution branches starting in state (c, t, m, ℓ) fails, $(c, t, m, \ell) \Downarrow Q$ cannot be proven, no matter what Q is. On the other hand, traditional operational semantics need to talk about failures explicitly or model their absence separately, to make sure failing execution branches are not discarded silently. For instance, CompCert's definition of backwards simulation needs to reference two separate judgments about program execution, called `safe` and `Step`, and CompCert cannot use the more convenient [28, 36] forward simulations in the presence of internal nondeterminism, whereas CPS semantics allow us to deal only with one judgment about program execution and to do all proofs in forward style. Inductively defined Hoare triples can be used as specifications for both source and target language during compiler verification, in which case induction over the semantics corresponds (modulo weakening) to induction over syntax trees (with nested induction over termination measures for the loops). We believe this style might have worked just as well above the assembly-language level. Hoare logics for machine code have also been crafted, but their design has been the main subject of entire papers [9, 32], whereas we got away with instantiating an existing monadic interpreter to get a CPS-semantics definition (section 5.4).

4.3 CPS Semantics for RISC-V

Contrary to Bedrock2 programs, RISC-V programs stored in the memory of a RISC-V machine do not really have a notion of termination: the processor keeps executing the instruction pointed to by the program counter forever. This means that a structurally recursive weakest-precondition generator can only be written for individual RISC-V instructions, not entire programs. We write $s \rightarrow Q$ to say that executing one instruction on a RISC-V machine in state s (which includes data memory as well as instruction memory, registers, program counter, etc.) successfully results in a state that satisfies Q.

To lift this predicate from a single step to multiple steps, we use an operator $^\Diamond$ we call the *eventually operator* that serves a similar purpose as the transitive-closure operator:

$$\frac{Q(x)}{x \rightarrow^\Diamond Q} \qquad \frac{x \rightarrow Q_1 \quad \forall y.\ Q_1(y) \Rightarrow y \rightarrow^\Diamond Q_2(y)}{x \rightarrow^\Diamond Q_2}$$

Note that it allows each nondeterministic branch to use a different number of steps, so the number of execution steps can depend on an input value modeled as nondeterministic.

5 Layers of the Stack

Figure 3 presents a detailed view of the system. This section consists of a top-to-bottom tour through the figure, explaining how each component (white rectangle) is implemented and verified against the interfaces surrounding it (gray rectangles), and a presentation of how to compose the hardware and software proofs into a single end-to-end theorem.

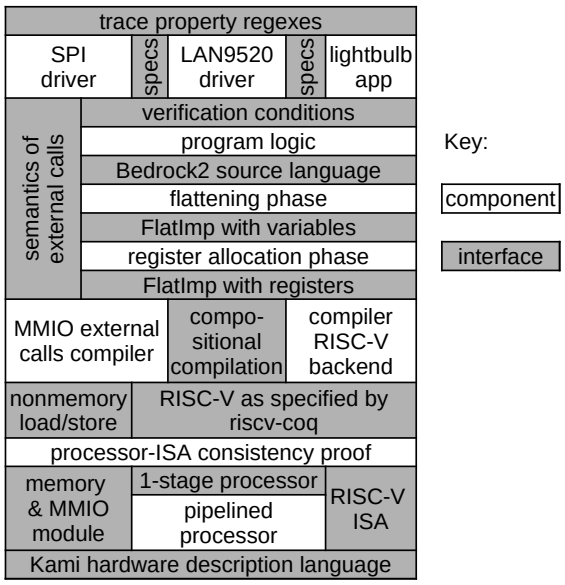

Figure 3. Components and interfaces of our system

5.1 The Application Layer

At the very top of Figure 3, a trace property we call `goodHlTrace` (sketched for the lightbulb in section 3.1) defines which I/O traces are acceptable behaviors of the system. In general, this property should ignore all internal implementation details and only describe what transpired on the hardware-software system's inputs and outputs. As we are only dealing with digital systems, it is natural for these traces to apply discretely, with at most one entry per hardware cycle.

Our prototype consists of three Bedrock2 source files: `SPI`, the driver used to communicate with the network interface card; `LAN9250`, the Ethernet device driver; and `lightbulb`, an infinite loop that polls the network card for packets, processes them, and turns the lightbulb on or off depending on their content. We replicated the SPI and GPIO interfaces from the commercial FE310 RISC-V microcontroller [22] based on the Rocket [4] RV32IMAC core, which allowed us to do separate testing of our hardware and software (on the FE310 chip itself). The SPI interface exposes send and receive queues via MMIO, relying on polling to detect peripheral-initiated flag changes. The LAN9250 Ethernet controller's API is exposed as a range of SPI-accessible address space where reads and writes to different addresses correspond to different operations.

5.2 The Bedrock2 Source Language

We write our application code in a syntactic subset of C that we call Bedrock2, with semantics that include most but not all opportunities for undefined behavior from C. For instance, like in C, accessing out-of-bounds memory is undefined behavior, but unlike C, division by zero is allowed[3], and comparisons between any two pointers are allowed, whereas C assigns undefined behavior to less-than comparisons of two pointers if they do not belong to the same object. The language statements include memory write, if-then-else, while loops, and function calls with support for returning tuples of values. Syntactically, we distinguish calls to Bedrock2-defined procedures and calls to external procedures. The semantics records the latter in an interaction trace (which is only used in specifications, not maintained at runtime). External procedures can update the memory (and such updates are recorded in the trace), but we do not make use of this feature, because we have not yet modeled it on the RISC-V level. The memory is modeled as a global (not necessarily contiguous) address space of bytes without any artificial limitations (effective type, provenance, alignment, etc.) on how they can be accessed. All function arguments and local variables in Bedrock2 have the same type, `word`, whose bitwidth depends on the bitwidth of the target machine.

The Bedrock2 source language is very simple, to the point where one might wonder whether it would scale to bigger applications. However, since the development of Bedrock2 programs happens *inside* Coq, we already have the full power of all of Coq's abstraction mechanisms at our disposal, and we have used them for both data-representation specifications and syntactic-sugar macros. In addition, there is ongoing work on compiling higher-level languages to Bedrock2.

We outright omit higher-order features such as function pointers and mutually dependent compilation units, which lets us avoid the semantic intricacies studied in the work on compositional compilation [39]. While implementations of external calls are still required to preserve the same invariants as the compiler itself when called in accordance to their specifications, this requirement can be stated straightforwardly and proven without reference to the calling code (section 6.1).

The Event-Loop Invariant. We only model behavior of terminating programs in the Bedrock2 source language, implicitly identifying nontermination with undefined behavior. Totality is an important correctness property and a slight simplification, but it also forced us to verify the customary top-level `init(); while(1) loop()` idiom of simple embedded programming directly against the RISC-V semantics. We first state an invariant `inv` on a RISC-V machine that holds at the beginning of each loop iteration. As loop-iteration boundaries are not observable from outside the system, we then use the *eventually operator* (section 4.3) to construct an

[3]Division is modeled as an axiomatically specified total deterministic function, and the source-language semantics do not specify what it returns for divisions by zero, whereas the compiler assumes that it returns the concrete values specified by RISC-V.

instruction-by-instruction invariant saying that the execution is a finite number of steps away from a state satisfying `inv`, and we use the *always operator* (5.8) to assert that for all correctly initialized RISC-V states `s`, this new invariant applies as well: `swalways s (fun s' ⇒ s' →◊ inv)`.

5.3 The Bedrock2 Compiler

Below the box for the source language in Figure 3, we show the intermediate languages of the 3-phase compiler, which are fairly standard for verified compilers today, bottoming out in lists of position-independent RISC-V instructions, which are then encoded to bytes as specified by RISC-V.

The semantics for Bedrock2 and internal intermediate languages of the compiler are all in CPS (section 4), which enables us to write all compiler-correctness proofs as forward proofs, even though the languages have external nondeterminism (from axiomatically specified external calls) as well as internal nondeterminism (the address at which stack allocation allocates memory is unspecified).

To describe the application memory as well as the memory introduced by the compiler such as the stack and the instruction memory, we use separation logic throughout all compiler-correctness proofs. This, together with disallowing recursive functions and statically tracking the stack-space requirements of each function, enables us to prove that the application memory, stack memory, and instruction memory all fit into the same memory. We also prove that the application will never run out of memory, a guarantee that other compilers such as CakeML or CompCert do not provide but which is essential to obtain a meaningful end-to-end theorem.

5.4 The RISC-V ISA and its Formal Semantics

RISC-V [42] is the only instruction set allowing us to create and distribute our own implementations without running afoul of patents, while also featuring realistic silicon-fabricated processors and production-quality software toolchains. We used a formal model of RISC-V written in Haskell [7] that was translated to Coq using hs-to-coq [8].

For flexibility reasons, the semantics in Haskell deliberately only specifies how each RISC-V instruction is defined in terms of a small number of primitives such as reading and writing registers or memory, *without* giving semantics to these primitives or specifying any data type representing the state of a RISC-V machine. For instance, the store-word instruction `Sw` is turned into a sequence of the four primitives `getRegister`, `translate` (performing virtual-to-physical address translation), `getRegister`, and `storeWord`:

```
| Sw rs1 rs2 simm12 ⇒
    a ← getRegister rs1;
    addr ← translate Store 4 (add a simm12);
    x ← getRegister rs2;
    storeWord Execute addr (regToInt32 x)
```

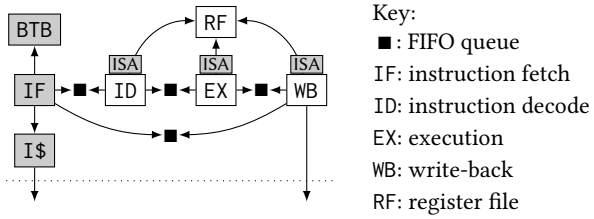

Figure 4. The Kami processor and its two asynchronous memory interfaces. Our additions are highlighted in gray.

To prove a compiler correct against these semantics, we need to define a state type and define how the primitives operate on it. We have both a deterministic implementation, allowing us to run small RISC-V programs in Coq, and a nondeterministic CPS semantics that includes MMIO. The compiler proof is parameterized to apply to either (section 6.3). Our models describe a single-core environment with no address translation, but they expose limitations of naive instruction caching (section 5.6).

5.5 The Kami Processor

We build on top of the Coq-verified processor from the Kami hardware-verification framework [10], already equipped with a four-stage in-order pipeline, and we made several improvements and additions (shown in gray in Figure 4). The baseline processor required the program to be specified in the processor design; we added logic to fetch instructions eagerly from main memory into an interface-compatible instruction cache residing in FPGA block RAM upon reset. We also reconciled the instruction set with RV32I, by adding missing instructions, including the *fine-grained memory operations* load-byte (`lb`) and store-byte (`sb`), which required adding `byte-enable` signals to the memory interface, and by fixing some bugs. The bugs were of two kinds: liveness, which is not covered by Kami's specification and was found through testing our application; and specification bugs that had not been found by Kami's specification-validation efforts but showed up while trying to prove Kami's RISC-V specification equivalent to the one used by the compiler. We also added a branch predictor `BTB` [35] and fixed an issue that kept pipeline stages from executing concurrently.

5.6 Dealing with Stale Instructions

The instruction-set specification does not require memory accesses for instruction fetching to be consistent with those of load and store instructions, allowing implementations to fetch and start executing instructions early (in this case, through an `I$` memory interface) without synchronization circuitry to detect when an already-fetched instruction was written to (through the memory interface of `WB`). While more-sophisticated CPU designs can detect and handle such hazards, it is overwhelmingly common for embedded systems to implement the synchronization in software instead.

We encode this discipline in the RISC-V model used by the compiler by tracking a set of executable addresses `XAddrs` throughout the execution. Whenever an instruction is fetched, undefined behavior is triggered if the fetch address is not in `XAddrs`, and each written address is removed from the set of executable addresses. At boot, `XAddrs` covers the entire memory. The correctness proof of our compiler includes showing that the program addresses remain executable throughout program execution.

The preservation of this invariant relies on external calls not modifying the set of executable addresses, which in turn only holds if the program is calling the external calls in accordance with their specifications, which of course depends on the correctness of execution of compiled code so far. We have yet another example of how correctness specifications of interfaces between embedded systems' components are intertwined in nonobvious ways that have important engineering benefits but are easily lost in academic simplifications.

5.7 Correctness of the Processor

A hardware design in Kami consists of *modules* with encapsulated private state (registers), public methods, and *rules* that make atomic state changes. Kami exports to Bluespec, and the Bluespec compiler discovers opportunities for parallel execution of rules, but with the convenient semantic guarantee that the execution can be interpreted as if the rules were executed one-by-one, called *one-rule-at-a-time execution*. Behavior of Kami designs is modeled using traditional small-step semantics where each step corresponds to a state transition by a rule in a module and records the I/O events (external method calls) that happened. The step definition has the form `kstep m s1 s2` (which means that the Kami module `m` steps from state `s1` to state `s2`) and can be lifted to characterize multiple steps (`kstep`*) from the initial state of a module (`initial m`) to define the set of *traces* of a module:

```
Definition Trace (m: KModule) (tr: list Label) :=
  ∃ s, kstep* m (initial m) s ∧ tr = trace s.
```

The pipelined processor is proven to implement a single-cycle processor model in the sense of *refinement*, showing that the set of possible traces of the implementation is contained in the trace set of the spec. A key property of the Kami module system is *modular refinement*: in a system composed of multiple modules, replacing any module with a spec that it refines does not lose any behaviors. Thus, we were able to prove our new instruction-cache logic without considering the rest of the processor: it simply refines the original fetch stage in the baseline processor. The combinational-logic functions for decoding and executing instructions are shared between baseline single-cycle processor spec and the pipelined implementation, so we were able to extend the ISA and fix bugs in it without needing to touch a line of proof. Unlike in the original Kami case study, we also need to prove that all this logic matches the software's assumptions.

5.8 Interfacing Hardware and Software

The Bedrock2 compiler and the Kami processor were developed independently and are proven correct against very different RISC-V specifications: a software-oriented assembly-like semantics (section 5.4) and a single-cycle Kami model processor (section 5.7). There are two challenges for proving these specifications compatible: reconciling the encoding styles in the two developments (a concise but tricky proof) and proving equality of alternative ways in which the two wrote down "the same" bitvector expressions (a simple but strenuous proof). We will elaborate on the former.

The software-oriented specification is in CPS, that is, it relates each starting state to a set of possible next states if well-defined: `swstep s S` means that all states reachable from `s` in one step belong to the set `S`, and crucially *no execution path can trigger undefined behavior*. A state `s` where a step may trigger undefined behavior satisfies `∀ S, ¬ (swstep s S)`, which allows us to reject undefined scenarios *before* considering the possibilities for the next state. For non-MMIO instructions, S is a singleton set.

The Kami semantics does not have a notion of undefined behavior. Scenarios that result in undefined behavior according to the software-oriented `swstep` just proceed in some arbitrary way according to Kami's `kstep` relation. For example, memory accesses at too-large addresses just wrap around, ignoring the more-significant address bits. Input nondeterminism is encoded in `kstep` by relating the state before the input to multiple possible subsequent states, each with a different label added to the I/O trace.

We want to show that `swstep` is a conservative model of the possible Kami-processor-model executions. More precisely, we want to consider an arbitrary Kami step from a state `s1` that does not trigger undefined behavior according to `swstep` but instead steps to a state in `S`, to show that the resulting Kami state `s2` is in `S`. As the two semantics use different state types, we need to use a simulation relation (`related`) to derive a type-correct version of the last sentence:

```
Theorem kstep1_sound : ∀ ks1 ks2 rs1 S,
  swstep rs1 S ∧ kstep ks1 ks2 ∧ related ks1 rs1 →
  related ks2 rs1 ∨ ∃ rs2, rs2 ∈ S ∧ related ks2 rs2
```

Note that *both* an invocation of the software-oriented RISC-V semantics and a Kami execution appear as hypotheses. Further, `related` includes invariants that arise from well-defined `swsteps`, most importantly that the instruction cache is consistent with main memory at the executable addresses in the sense of section 5.6.

To derive correctness of multi-step executions, we prove that if a predicate is an invariant (i.e., always holds on all states reachable from a given initial state) according to the

compiler's semantics, it is also an invariant according to Kami's semantics:

```
Let swalways s P := P s ∧ ∀ s', P s' → swstep s' P.
Let kalways s P := ∀ s', kstep* s s' → P s'.
Theorem kstep_star_sound: ∀ inv ks1 rs1,
  related ks1 rs1 ∧ swalways rs1 inv →
  kalways ks1 (fun ks2 ⇒
      ∃ rs2, related ks2 rs2 ∧ inv rs2).
```

To obtain the end-to-end theorem, the invariant `inv` is instantiated with the event-loop invariant from section 5.2.

5.9 The End-to-End Theorem

We finally assert application-level correctness of Kami I/O:

```
Theorem end2end_lightbulb: ∀ mem0 t,
  bytes_at (instrencode lightbulb_insts) 0 mem0 ∧
  Trace (p4mm mem0) t →
  ∃ t': list (string * word * word),
    KamiRiscv.KamiLabelSeqR t t' ∧
    prefix_of t' goodHlTrace.
```

In words, running our pipelined processor `p4mm` with any memory `mem0` that contains the lightbulb-program machine code at address `0` only produces I/O traces that are related to (prefixes of) traces allowed by the application specification. The prefix closure is important because this theorem holds at *any point* during the execution, without reference to any notion of the software having "completed" a loop iteration. The relation `KamiRiscv.KamiLabelSeqR` simply maps Kami MMIO traces to triples with `"ld"` and `"st"` of section 3.1.

Another way to read this theorem is as system-bring-up recipe: compute `instrencode lightbulb_insts` in Coq, place it at address `0` in a memory, and arrange for this memory to be connected to a correctly synthesized copy of `p4mm`. Then, behavior described by `goodHlTrace` is to follow based on our proofs. We would like to emphasize that, compared to other verification projects, only requiring the three items described above to be understood and trusted is very minimal. No semantics of instruction sets nor software programming languages need to be trusted in order to trust this theorem, and there is no unverified "host" device in our case study.

Moreover, there is also no bootloader. All one has to do in order to create a physical system that satisfies the preconditions of our theorem is to program an FPGA with the design of the Kami processor and to put the Coq-generated RISC-V binary code into the FPGA's memory at address 0. On FPGA reset, the Kami processor directly starts executing at hard-coded address 0, so every instruction executed by the system is taken into account by our theorem.

6 Parameterization across Layers

So far we have mostly emphasized *vertical* modularity. For instance, we could swap the implementation of a layer such as the compiler or the processor for a different implementation, and the specifications at the layer boundaries guarantee that we need not revisit the other layers of the system. However, some dimensions of variation across systems are orthogonal to that decomposition. One natural example is which peripheral devices are available and how to interact with them. Every layer of our stack is parameterized by its relevant choices there, and we think of this parameterization as *horizontal* modularity.

Table 2. Parameterization throughout the stack

Parameter	Used in
external-call semantics	program logic and compiler
external-calls compiler	compiler and its proof
event-loop invariant	compiler-processor lemma
bitwidth	Bedrock2, ISA, processor
I/O mechanisms	compiler and its proof
I/O load/store semantics	instruction-set specification
external invariant	ISA, compiler and its proof
ISA	processor and its proof

For our lightbulb case study, the processor communicates with the network card and the lightbulb power switch through MMIO. In this section, we particularly focus on the parameterization for I/O devices, while Table 2 summarizes other examples of parameterization in our stack.

Such composition of different pieces of compiled programs has been studied before, e.g. in CompCompCert [39] using a block-based infinite memory managed by an allocator behind the curtains (even after compilation), whereas our composition has to work in a setting where the different pieces of machine code access the same flat finite address space that is used not only as data memory but also as instruction memory and for memory-mapped I/O.

6.1 I/O in Bedrock2

In Bedrock2 source code, we use a syntactically distinct construct for MMIO. To keep the language more general, we do not introduce a specific construct just for MMIO but rather a more-general construct we call *external calls*, which appear as special functions callable like any others. The semantics of the source language are parameterized over the behavior of these external calls. The concept of external calls is a strict generalization of MMIO, not a relaxation of semantics: the source-code-level verification condition for an MMIO external call still needs to restrict the address to be within MMIO range.

Recall the discussion in section 4.1 of the verification-condition generation. Here is the case for external calls (simplified assuming one argument and one return value):

$$\begin{aligned}&\mathsf{vcgen}((x = f_{\mathsf{ext}}(e)), t, m, \ell, Q) :=\\ &\quad \exists v.\ \mathsf{expr\_evaluates}(m, \ell, e, v) \wedge\\ &\qquad \mathsf{vcextern}\big(f_{\mathsf{ext}}, t, [v],\\ &\qquad\quad \lambda r.\ Q((f,[v],[r])::t, m, \ell[x := r])\big)\end{aligned}$$

The predicate vcextern is a parameter of the semantics – for the lightbulb, we instantiate it with a characterization of MMIO load and store operations and allowed address ranges in our platform. Like vcgen, vcextern computes a precondition that is sufficient to guarantee that the postcondition Q received as input to vcextern holds after the call. An important difference between vcgen and vcextern is that vcgen models deterministic steps, whereas vcextern needs to account for unknown runtime inputs, which are represented using a universal quantifier in the definition of vcextern. For example, an external call called `"arbitrary"` that requires exactly one nonzero argument b and can return any number less than b would have the specification

$$\begin{aligned}&\mathsf{vcextern}(\texttt{"arbitrary"}, t, args, Q) :=\\ &\quad \exists b.\ args = [b] \wedge 0 < b \wedge (\forall r.\ r < b \Rightarrow Q(r))\end{aligned}$$

Note that when proving the proof obligation returned by vcextern, the programmer has to prove Q (i.e., verify the remainder of the program) for all possible r.

6.2 I/O in the ISA Semantics

Our RISC-V specification is also parameterized over external interactions, implemented by giving special treatment to loads and stores that fall outside the memory owned by the code running on this processor. This special treatment records nonmemory loads and stores in the I/O trace of all externally visible behavior of the system, for which the end-to-end theorem will assert that it satisfies the `goodHlTrace` property. In our instantiation of the ISA specification, the memory footprint remains unchanged throughout execution.

The parameter modeling external interactions caused by an `n`-byte nonmemory load, `nonmem_load`, takes an address `a`, a machine state `s`, and (in the same style as `vcgen` and `vcextern`) a postcondition `Q`, returning the proof obligation the compiler has to prove to make sure that `Q` holds after executing the load instruction. Here is the instance for MMIO:

```
nonmem_load n a s Q :=
  isMMIOAddr a ∧ isMMIOAligned n a ∧
  ∀ v, Q v (withLogItem (@mmioLoadEvent a n v) s).
```

It requires the compiler to prove that `a` is in the MMIO range, it is `n`-byte aligned, and the desired postcondition holds for a machine state where the address and the unknown read value `v` have been added to the I/O log. The same interface is also powerful enough to model direct memory access (DMA), by recording memory-ownership changes in the I/O trace, but we do not make use of this feature in the lightbulb application.

6.3 I/O in the Bedrock2 Compiler

Our compiler pipeline is parameterized over an *external-calls compiler*, which defines how to implement each call with machine code. In the lightbulb example, it simply translates `MMIOREAD` and `MMIOWRITE` calls to `lw` and `sw` instructions.

We prove our compiler correct for all possible implementations of external calls in a compositional manner, requiring the same `correctness` of the external-calls compiler as we are proving about the whole compiler:

```
Lemma compiler_correct: ∀ compile_ext,
  (∀ x f_ext a, correct compile_ext (x = f_ext(a))) →
  (∀ program, correct (compile compile_ext) program).
```

Condition `correct comp p` says that feeding program `p` into the compilation function `comp` produces position-independent code that takes any machine state satisfying the *compiler invariant* to some machine state satisfying the compiler invariant and the postcondition of `p` (assuming no execution of `p` from the given starting state can trigger undefined behavior).

Note that the postcondition has to be translated as well, because a source-level postcondition P takes an I/O trace t and a source-level state as arguments, whereas a target-level postcondition takes a target-level state instead of a source-level state. We do so using a state-representation relation R between source and target states, translating the source-level postcondition P into the target-level postcondition $\lambda t\ s_{tgt}.\ \exists s_{src}.\ R(s_{src}, s_{tgt}) \wedge P(t, s_{src})$. If we wanted to support different trace formats for the source and target languages, we could simply include the trace in the representation relation and translate the source-level postcondition P into $\lambda t_{tgt}\ s_{tgt}.\ \exists t_{src}\ s_{src}.\ R(t_{src}, s_{src}, t_{tgt}, s_{tgt}) \wedge P(t_{src}, s_{src})$.

Verifying the External-Calls Compiler. The correctness proof of the external-calls compiler (i.e., the proof of the main hypothesis of the last lemma) relies both on the compiler invariant and on the source-level verification condition of external calls (`vcextern`). However, the main compiler as well as `correct` are too general to know anything about the concept of MMIO, but they still need to empower the correctness proof of the external-calls compiler to show that the loads and stores emitted for MMIO do not modify application data or code. Therefore, the compiler invariant includes not only administrative conditions regarding the stack and registers but also an *external invariant* that the code emitted by the external-calls compiler can rely on and which the proof of the main compiler takes as an abstract parameter. In our case study with MMIO only, it is sufficient to use an external invariant that requires MMIO addresses not to overlap with the physical memory, and `vcextern` requires the application

programmer to show that the addresses are indeed within the MMIO range (see section 6.1).

We must also provide a means to the main compiler to prove that it preserves the abstract external invariant, and we do so by imposing the condition on the abstract external invariant that it is preserved by all ordinary RISC-V instructions the main compiler uses (that is, in particular, excluding `lw` and `sw` outside the physical memory).

We found these details to be a particularly tricky exercise in parameterization and "threading" of invariants through a development. The solution we describe here relies on quantifying over predicates (`vcextern` and the external invariant) and their properties, an example of how use of higher-order logic enables modularity.

6.4 I/O in Hardware

I/O is encoded in Kami as invoking methods on an unspecified external module, which the semantics tracks in a behavior trace. The processor itself does not distinguish ordinary memory operations from MMIO. When the memory module is attached, it handles the loads and stores to memory addresses but makes designated external method calls for the rest. This factoring appears both in the pipelined processor and in the spec processor, making for an easy correctness proof by modular refinement.

6.5 Parameterization: Coq Pragmatics

By instantiating parameters appropriately, our main theorem depends only on standard Coq axioms[4]. We parameterize proof modules on records of assumptions, with many assumptions shared across different record types, which introduced more proof-automation complexity than we had anticipated (see section 7.3.1).

7 Evaluation

7.1 Remaining Opportunities for Bugs

7.1.1 Trusted Code Base. The trusted code base of any deployment of our system is dominated by external tools; the key specifications of the verified part are minuscule in comparison. This comparison holds even though the specifications of intermediate layers of our system are rather intricate – only the top (application) and bottom (HDL) specifications are critical for correctness, and they are simple and short. Table 3 provides a component-by-component breakdown of the specification size on the left, with a summary of the tools we rely on to translate the HDL code to an FPGA bitstream on the right. We would like to emphasize that all tools we used are open-source and actively maintained.

7.1.2 Specification Fidelity and Security. We believe that single-Qed integration verification against an application-level specification provides unparalleled assurance against many known and unknown attacks. Any behavior-changing attack through the MMIO interface is ruled out by an end-to-end correctness theorem, even though our specification contains no description of potential attacks. For example, the theorem we proved implies that the attacker cannot gain remote-code execution by sending specially crafted network packets: executing code on our system could easily be used to turn the lightbulb on when not allowed by the specification.

[4]Functional and propositional extensionality, Axiom K, and `JMeq_eq`

Table 3. Summary of our trusted code base

Coq spec, total lines of code	
Lightbulb application	27
LAN9250 Ethernet driver	77
SPI driver	30
Driving digital outputs	10
Trace predicate notations	25
Semantics of Kami HDL	~ 400

Other TCB

Verilog wrapper (~200 LOC), Kami→Bluespec extraction, Bluespec compiler, Yosys & Nextpnr [37], Coq proof checker & dependencies

One security-relevant limitation of our setup is that the top-level specification does not specify the timing of inputs and outputs. Even though the software is proven to terminate, and the compiler is proven to preserve termination, the Kami processor (according to the proofs alone) is not guaranteed ever to execute an instruction – we just tested that it does. For the same reason, our framework does not give a way to guarantee that execution time does not depend on secret values, but the application we chose for our case study does not handle secrets.

Finally, we emphasize that our theorem applies to the interface *between* the FPGA and the network interface card shown in Figure 2, i.e. the network interface card is excluded from the verification.

7.2 Performance

7.2.1 Runtime Performance. Our system is fast enough to control a lightbulb and many other mechanical systems. We did not track or attempt to optimize the performance of our system during development. Our goal to exercise seriously the intramodule flexibility provided by our interfaces sometimes led us to implement common optimizations (e.g., register allocation), but most design decisions were made in favor of simplicity over speed.

Running our processor with a 12MHz clock on a Lattice ECP5-85k FPGA, we measured that it takes 5.5 ms from the moment when the Ethernet device starts handing a packet over to the processor to the actuation of the control output. The corresponding figure for our initial unverified prototype code with `gcc -O3` and FE310 is 10x faster, just above 0.5ms. We will now explain this ratio as a combination of two I/O differences, a compiler weakness, and performance issues of the Kami processor: $10x \approx (1.4x \times 1.2x) \times 2.1x \times 2.7x$.

The vast majority of the running time is spent transferring incoming packet data from the Ethernet controller to the

processor over SPI using MMIO reads and writes to drive the SPI peripheral. Both the verified and unverified versions of the code use naive 4-byte transactions, but the unverified version makes use of the FE310 SPI pipelining feature within each transaction: even though SPI communication is inherently synchronous and bidirectional, the code first writes the outgoing command and address into the transmit FIFO and then reads the entire response out of the receive FIFO. The trace specification applies to the MMIO interface between the processor and the SPI peripheral, so we would have needed to include this optimization in the *specification* of the system behavior to support it. Our verified system instead interleaves one-byte writes and reads, as captured in the simplest specification we could come up with (and our Verilog implementation of SPI does not support pipelining). Changing the original prototype to do the same slows it down by 1.4x.

Another place where the final version of the code differs from our initial prototype is that it maintains timeout counters for polling at both LAN9250 and SPI levels, exiting with an error if the device does not respond within a reasonable amount of time. The unverified prototype would happily poll forever, which may be acceptable in some applications and not others. We added the timeout logic when setting up to prove total correctness for each iteration of the top-level event loop. Measuring the verified code with `gcc -O3` and FE310, the timeout logic increases the response time by 1.2x.

Our compiler does not do constant propagation, function inlining, or exploit caller-saved registers, whereas `gcc -O3` inlines the SPI driver function call in the innermost loop and compiles it to two instructions. Compiling the same verified code with our compiler instead of `gcc -O3` increases the response time by 2.1x.

Using the Kami processor instead of FE310 is responsible for the largest slowdown factor in our system, just above 2.7x. This system-level clock-frequency-relative slowdown we observed is actually *smaller* than the 4.8x reported in [10, Fig. 15.] (approximating the Rocket core as executing 1 instruction per cycle). However, our code is I/O-heavy, and the FE310 SPI peripheral is connected to the CPU through two layers of TileLink buses, which we expect to add considerable latency compared to our SPI code in the same clock domain and Verilog compilation unit as the Kami processor. Thus it is not clear how much of the change from 4.8x to 2.7x to attribute to peripheral differences or our fixes to the Kami processor to get it to run our code at all.

7.2.2 Verification Performance.

The main Coq development is built and verified automatically after every change by continuous integration, requiring less than 7.5GB of RAM and 80 minutes per build. Additionally, checking the Kami refinement proofs takes around 2 hours.

Table 4. Lines of code

Excluded:		implementation m	interface n	interesting proof p	low-insight proof q	proof overhead $(m+n+p+q)/m$	imagined overhead $(m+n+p)/m$
unrelated	10044						
library	7301						
imports	1907						
doc	354						
Kami	48294						
lightbulb app		176	130	33	1443	10.1	1.9
program logic		0	208	552	1785	–	–
compiler		931	1114	1325	6654	10.8	3.6
SW/HW interface		0	2053	991	3804	–	–
end-to-end		0	254	74	539	–	–

7.3 Effort

This project was completed over two-and-a-half years, starting from a preliminary specification of RISC-V in Haskell and (independently) the Kami framework and baseline processor. Three people did the vast majority of the engineering, joining respectively 6 months and 13 months into the two-and-a-half-year project. While this project was largely the main task for the people involved, it was hardly ever the only responsibility, and we estimate an average level of effort of 66%, adding up to around four person-years of work.

In the "proof overhead" column of Table 4, the overhead of formal verification (measured as the factor by which the number of lines of code increases due to verification) is calculated for different layers of the system (and omitted for those layers that consist purely of proof).

7.3.1 Coq Wishlist.

We caution against trying to use these ratios to quantify the fundamental difficulty of systems verification. Contrary to the conceptual questions about interface specifications discussed throughout this paper, we see the vast majority of *proof* work in this project as using Coq to emulate, semimanually, domain-specific verification strategies we would expect to find in layer-specific tools.

1. We experienced performance bottlenecks and inexplicable tactic failures connected both to logical theories with associated standard Coq tactics (e.g., linear arithmetic) and those without (e.g., bitvectors, finite maps, sequences). Tools like SMT-solver bridges to Coq [11, 15] crashed or ran too slowly on too many of our goals (even months after reporting issues to their authors) to be viable.
2. This code base's level of parameterization (as sketched in section 6) posed challenges for integrating with automation. We defined hierarchies of record types for collecting parameters, often leading to multiple ways of writing a lookup of the same parameter, though Coq's automation failed to take seamless advantage of those equalities.
3. Sometimes we coded workarounds in Coq's Ltac language, revealing some apparently fundamental bottlenecks in

core parts of Coq today: time scaling in number of hypotheses (leading us to invest in heuristics to prune hypotheses unlikely to help), unpredictable and slow unification (we found ourselves often wanting to trigger the most basic but fast heuristics), and rechecking of proof terms at the end of a proof (many tactics do poor jobs of recording term-reduction strategies in proof terms).

One notable bright side was Coq's evolving notation mechanism, improving in the course of this project as we sent feature requests to the Coq developers, to the point where we can now write fairly natural-looking C-like code directly within Coq (conveniently located alongside its proof).

7.3.2 What if our Wishlist Were Addressed Fully? But what if all these problems were solved, and Coq also had access to layer-specific tools such as performant symbolic-execution engines and theory solvers that are already successfully used in many unintegrated verification projects? We tried to answer this question with Table 4, by classifying the source lines of proofs manually into "interesting proof" and "low-insight proof." The hypothetical improved proof assistant would only need the former. The large differences in the factors in the two last columns suggest that with the current state of the art, the complexity of formal-systems verification is mostly *accidental* complexity caused by tooling issues and only a small amount of *inherent* complexity, so that improving proof assistants would be a key enabler for larger projects in the direction explored in this paper. In a sense, it is a chicken-and-egg problem: to motivate improvement of proof assistants, we need convincing system demos, preferably much larger than the one presented in this paper, but in order to create such demos, we already need better proof assistants. We hope that this paper can make a contribution in breaking this cyclic dependency by highlighting the potential benefits of improving proof assistants and their libraries and performance, motivating further development of these tools.

8 Conclusion

We presented another step toward more complete end-to-end mechanized proof of systems combining software and hardware, with small trusted code bases. Since our top-level theorem does not reference any of the intermediate specifications, we can rule out a large class of potential integration bugs.

In order to focus on this kind of integration verification, we chose rather simple designs for the individual components, so the work we presented here cannot yet fully answer the question how our technique would scale if we replaced the individual components by more complex designs. More complex designs would likely lead to more complex intermediate specifications, and since we have not yet encountered any friction points in our specification style, we believe that it is ready for use with more complex designs that need features such as direct memory access or, more generally, external calls that acquire and release logical ownership of memory. On the other hand, concurrent software execution (on multiple cores or in interrupt handlers) would require considerable changes to our current approach. As far as the proofs (rather than specifications) are concerned, we already did encounter scalability issues (section 7.3.1), but as we described, we believe that these are not fundamental.

Compared to past work, we emphasize building a free-standing digital system that uses realistic I/O, an instruction set that is already widely used, and low-level coding patterns representative of embedded systems. New challenges were raised for modularity of both the vertical (layering) and horizontal (parameterization) kinds. We also found that tooling challenges with performant proof automation in Coq dominated our development time, feeding a wishlist of mundane-sounding Coq improvements. It seems that these limitations must first be overcome to attain feasibility for any integration-verification case study large enough to benefit genuinely from a modular architecture. Still, we were able to complete the last conceptual ingredients in an end-to-end functional-correctness theorem that directly captures the I/O behavior of a very simple untethered embedded system.

Acknowledgments

This work was supported in part by the Internet Policy Research Initiative at MIT and by National Science Foundation grant CCF-1521584, for the Expedition on the Science of Deep Specification. We thank Frédéric Besson and Hugo Herbelin for their work on Coq arithmetic proof automation and parsing mechanisms, including implementing several improvements and fixes in response to our experience in this project. We thank John Grosen and Adam Suhl for helping us with buffer-overflow exploitation and initial LAN9250 bring-up respectively. For comments on drafts of the paper, we thank Eric Atkinson, Lennart Beringer, Tej Chajed, Wolfgang Paul, Jade Philipoom, Jaagup Repän, Peter Schmidt-Nielsen, Gordon Stewart, and Daniel Ziegler.

References

[1] Eyad Alkassar, Mark A. Hillebrand, Steffen Knapp, Rostislav Rusev, and Sergey Tverdyshev. 2007. Formal Device and Programming Model for a Serial Interface. In *Proceedings of 4th International Verification Workshop in Connection with CADE-21, Bremen, Germany, July 15-16, 2007 (CEUR Workshop Proceedings, Vol. 259)*, Bernhard Beckert (Ed.). CEUR-WS.org. http://ceur-ws.org/Vol-259/paper04.pdf.

[2] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. 2008. The Verisoft Approach to Systems Verification. In *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08) (LNCS, Vol. 5295)*, Natarajan Shankar and Jim Woodcock (Eds.). Springer, 209–224.

[3] Andrew W. Appel. 2014. *Program Logics - for Certified Compilers*. Cambridge University Press. https://www.cs.princeton.edu/~appel/papers/plcc.pdf

[4] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[5] William R. Bevier, Warren A. Hunt, Jr., J. Strother Moore, and William D. Young. 1989. An approach to systems verification. *J. Autom. Reasoning* (1989), 411–428. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.6467&rep=rep1&type=pdf

[6] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA (LIPIcs, Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:12. https://oadoi.org/10.4230/LIPIcs.SNAPL.2017.1

[7] Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Andrew Wright, and Adam Chlipala. 2021. A Multipurpose Formal RISC-V Specification. arXiv:2104.00762 [cs.LO]

[8] Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, Set, Verify! Applying Hs-to-Coq to Real-World Haskell Code (Experience Report). *Proc. ACM Program. Lang.* 2, ICFP, Article 89 (July 2018), 16 pages. https://oadoi.org/10.1145/3236784

[9] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. 2007. Certified Self-Modifying Code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 66–77. https://oadoi.org/10.1145/1250734.1250743

[10] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (Aug. 2017), 30 pages. https://oadoi.org/10.1145/3110268

[11] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning* 61, 1-4 (June 2018), 423–453. https://oadoi.org/10.1007/s10817-018-9458-4

[12] Matthias Daum, Norbert W. Schirmer, and Mareike Schmidt. 2010. From operating-system correctness to pervasively verified applications. In *Proc. IFM*. Springer-Verlag, 105–120. https://hal.inria.fr/inria-00524575/document

[13] George Dunlap. 2012. The Intel SYSRET privilege escalation. https://xenproject.org/2012/06/13/the-intel-sysret-privilege-escalation/.

[14] Alex Dzyoba. 2014. A tale about data corruption, stack and red zone. https://alex.dzyoba.com/blog/redzone/.

[15] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Vol. 10427. Springer International Publishing, 126–133. https://homepage.divms.uiowa.edu/~tinelli/papers/EkiEtAl-CAV-17.pdf

[16] M. Anton Ertl. 2015. What every compiler writer should know about programmers or "Optimization" based on undefined behaviour hurts performance. http://www.complang.tuwien.ac.at/kps2015/proceedings/KPS_2015_submission_29.pdf

[17] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 287–305. https://www.microsoft.com/en-us/research/wp-content/uploads/2017/10/komodo.pdf 10.1145/3132747.3132782.

[18] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, Mumbai, India, 595–608. https://flint.cs.yale.edu/flint/publications/dscal.pdf 10.1145/2676726.2676975.

[19] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[20] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 165–181. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel

[21] Warren A. Hunt. 1989. Microprocessor Design Verification. http://www.cs.utexas.edu/users/boyer/ftp/cli-reports/048.pdf

[22] SiFive Inc. 2019. *SiFive FE310-G000 Manual (v3p1)*. Available under "Freedom E310-G000" at https://www.sifive.com/documentation as of 2020-05-16.

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://ts.data61.csiro.au/publications/nicta_full_text/3783.pdf 10.1145/1629575.1629596.

[24] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 234–248. https://oadoi.org/10.1145/3293880.3294106

[25] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://ts.data61.csiro.au/publications/nicta_full_text/7494.pdf 10.1145/2535838.2535841.

[26] Adam Langley. 2016. memcpy (and friends) with NULL pointers. https://www.imperialviolet.org/2016/06/26/nonnull.html.

[27] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. http://xavierleroy.org/publi/compcert-CACM.pdf

[28] Xavier Leroy. 2009. A Formally Verified Compiler Back-End. *Journal of Automated Reasoning* 43, 4 (Dec. 2009), 363–446. https://oadoi.org/10.1007/s10817-009-9155-4

[29] 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*. 1041–1053. https://cakeml.org/pldi19.pdf

[30] William Mansky, Wolf Honoré, and Andrew W. Appel. 2020. Connecting Higher-Order Separation Logic to a First-Order Outside World. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 428–455. https://oadoi.org/10.1007/978-3-030-44914-8_16

[31] Kayvan Memarian and Peter Sewell. 2016. What is C in practice? (Cerberus survey v2): Analysis of Responses – with Comments. ISO SC22 WG14 N2015. http://www.cl.cam.ac.uk/~pes20/cerberus/analysis-2016-02-05-anon.txt

[32] Magnus O. Myreen and Michael J. C. Gordon. 2007. Hoare logic for realistically modelled machine code. In *In Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007), LNCS*. Springer-Verlag, 568–582. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.96.2793

[33] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 225–242. https://oadoi.org/10.1145/3341301.3359641

[34] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 252–269. https://oadoi.org/10.1145/3132747.3132748

[35] C. H. Perleberg and A. J. Smith. 1993. Branch Target Buffer Design and Optimization. *IEEE Trans. Comput.* 42, 4 (April 1993), 396–412. https://oadoi.org/10.1109/12.214687

[36] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (June 2013), 1–50. https://www.cl.cam.ac.uk/~pes20/CompCertTSO/doc/paper-long.pdf 10.1145/2487241.2487248.

[37] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanović. 2019. Yosys+nextpnr: an Open Source Framework from Verilog to Bitstream for Commercial FPGAs. arXiv:1903.10407 [cs.DC] 4 page short paper at IEEE FCCM 2019.

[38] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 287–305. https://unsat.cs.washington.edu/papers/sigurbjarnarson-nickel.pdf

[39] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, Mumbai, India, 275–287. https://www.cs.princeton.edu/~appel/papers/compcomp.pdf 10.1145/2676726.2676985.

[40] Sergey Tverdyshev. 2009. *Formal Verification of Gate-Level Computer Systems*. Ph.D. Dissertation. Saarland University. http://www-wjp.cs.uni-saarland.de/publikationen/Tv09.pdf

[41] Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proc. ACM Program. Lang.* 3, POPL, Article 62 (Jan. 2019), 30 pages. https://oadoi.org/10.1145/3290375

[42] Andrew Waterman and Krste Asanovic. 2019. The RISC-V Instruction Set Manual. https://riscv.org/specifications/.

[43] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying Software Network Functions with No Verification Expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 275–290. https://oadoi.org/10.1145/3341301.3359647