Taming an Authoritative Armv8 ISA Specification: L3 Validation and CakeML Compiler Verification

Hrutvik Kanabar
University of Kent, UK

Anthony C. J. Fox
Arm Limited, UK

Magnus O. Myreen
Chalmers University of Technology, Sweden

Abstract

Machine-readable specifications for the Armv8 instruction set architecture have become publicly available as part of Arm's release processes, providing an official and unambiguous source of truth for the semantics of Arm instructions. To date, compiler and machine code verification efforts have made use of unofficial *theorem-proving-friendly* specifications of Armv8, e.g. CakeML uses an L3-based specification. The validity of these verification efforts hinges upon their unofficial ISA specifications being *valid* with respect to the official Arm specification.

Leveraging the Sail language ecosystem, we bridge this validation gap by formally verifying that an L3-based specification simulates the official Arm specification using the HOL4 interactive theorem prover. We exercise this simulation by proving a novel compiler correctness result for CakeML with respect to Arm's official specification of the Armv8.6 A-class instruction set.

2012 ACM Subject Classification Software and its engineering → Software verification; Hardware → Theorem proving and SAT solving; Software and its engineering → Domain specific languages

Keywords and phrases Compiler verification, ISA specification, HOL4, interactive theorem proving

Supplementary Material Specifications and proofs in the HOL4 interactive theorem prover.
Software (ISA specifications): `https://github.com/HOL-Theorem-Prover/armv8.6-asl-snapshot`
Software (proofs, HOL4 repository): `https://github.com/HOL-Theorem-Prover/HOL/pull/981`
Software (proofs, CakeML repository): `https://github.com/CakeML/cakeml/pull/858`

Funding *Hrutvik Kanabar*: UK Research Institute in Verified Trustworthy Software Systems (VeTSS)
Magnus O. Myreen: Swedish Foundation for Strategic Research

1 Introduction

Rigorously verifying the behaviour of software requires faithful modelling of the hardware on which it runs. Instruction set architectures (ISAs) provide a convenient abstraction of hardware behaviour: if hardware manufacturers ensure a processor adheres to the ISA it implements, it can be viewed by software as a machine that executes exactly the specification of that ISA.

Instruction set specifications have been used in a wide range of theorem proving projects [3, 9, 11, 18, 25, 29, 32, 40, 52, 54, 55], and much work has gone into using DSLs for the rigorous engineering of accurate ISA specifications [1, 15, 45]. Specifications of the Armv8 architecture are available in three such DSLs: L3 [15], ASL [44], and Sail [2]. ASL emerged in 2011, building on the pseudocode language used in Arm documentation since the late 1990s. L3 and Sail were conceived in academia concurrently, with differing objectives; L3 was implemented in 2011 ahead of Sail.

Though superficially similar, these three languages have distinct design goals:

- L3 supports interactive theorem proving, so all L3 specifications can be reliably extracted to valid HOL4 and Isabelle/HOL, producing tractable and useable (idiomatic) definitions.

- ASL is developed within Arm, with broad design objectives (Section 2). Fully-automated verification and testing are prioritised (model-checking, SMT verification, and simulation). Arm-internal processes produce ASL specifications for Arm ISAs and subject them to rigorous internal testing to ensure full architecture compliance [43, 44, 53]. They are then released publicly, allowing users to examine and even run them.
- Sail was initially designed for use in concurrency tools, but it is now one-size-fits-all: it supports many ISAs, and diverse extraction targets such as theorem proving (interactive and automated), symbolic evaluation, simulation, and documentation. It is therefore more ambitious and featureful than both ASL and L3, with a significant ecosystem of tools. Conversely, it is less directly connected to theorem proving backends than L3.

Sail's ecosystem includes an ASL frontend to translate official ASL specifications into Sail, and a HOL4 backend to translate Sail specifications into HOL4. Put together, these provide a pathway from official ASL specifications to HOL4.

The HOL4 specification of Armv8 derived from this process is one of the largest (488 kLoC) and most complicated known ISA specifications in a theorem prover, matched only by other specifications produced from official Arm ASL via Sail.

However, the extensive Arm-internal validation of ASL specifications provides the closest approximation to ground truth for semantics of the Arm ISA. To the best of our knowledge, there is currently no way to obtain a more faithful Armv8 specification in an interactive theorem prover.

Therefore, we prove once and for all that the L3 Armv8 specification simulates the ASL-derived one, allowing verification efforts to enjoy the ease-of-use of the former while retaining the faithful modelling of the latter. Future work can avoid navigating the complexity of the ASL-derived specification, and other users of the L3 specification benefit from its validation against official ASL. We demonstrate our approach by using this simulation result to prove a novel compiler correctness theorem for the CakeML compiler.

Contributions

We make the following contributions:

- We demonstrate that official Armv8 ISA specifications can be used as reference semantics in interactive proof, through translation to HOL4 via Sail (Section 3) and interactive abstraction to a more theorem-prover-friendly specification.
- We validate the L3 specification for Armv8 by proving it simulates the ASL-derived specification (Section 4.4). Previously this specification had not been rigorously validated.
- We leverage these two results to prove the *first compiler correctness result with respect to an official Arm ISA specification* (Section 5).

Our work is open-source and has been integrated into the HOL4 and CakeML public repositories for future re-use.

This paper also addresses a number of questions about the HOL4 specification of Armv8 derived from ASL via Sail:

- How does it differ from the L3-derived specification? (Section 4.2)
- How useable are its semantics of instructions within the theorem prover? (Section 4.3)
- What can be said about its trustworthiness, particularly in comparison to the L3 specification? (Section 6)
- How could we better adapt it for use in the theorem prover? (Section 6)

2 Background

This paper draws together prior research and industrial work on *architecture specification languages*: DSLs used to define behaviours of ISAs.

ISAs are extremely complex, so their documentation is sizeable — running into thousands of pages and covering topics such as: processor state (operating mode, registers, and memory); instruction encoding, semantics, and assembly syntax; memory models, memory protection (virtualisation), memory attributes, and caching; synchronisation and semaphores; debug, trace, and monitoring; interrupts, timers, and exceptions.

Although machine-readable ISA specifications cannot cover *all* aspects of ISAs comprehensively, they can provide a working reference semantics for machine code. There are many benefits to using machine-readable specifications, including:

- Avoidance of ambiguities and errors detected by parsing and type-checking;
- Robust validation paths through simulation-based testing;
- Formal verification of processor designs through model-checking [21, 53, 55];
- Support for other formal modelling and verification activities, such as work on memory and concurrency models [12,13,22,42], architecture security [5,8,40,54], OS and hypervisor verification [6, 23, 24, 28, 30], compiler and runtime verification [29, 34, 36, 37], and machine code verification [3, 16, 19, 31].

The domain of architecture specification lends itself to first-order, imperative languages with some common features. Scattered functions define each of their clauses separately: encoding, decoding, and execution behaviour of a particular instruction can be grouped together, enhancing readability and modularity. Strong typing enables early error-detection, and type inference reduces the need for type annotations.

ASL. ASL is the Arm-internal architecture specification language. Its initial objective was to reduce errors in the informal pseudocode found in Arm documentation by providing parsing and type-checking [45, 50]. The language has since evolved and is now in extensive use within Arm. Its type system supports lightweight dependent typing: bit vector widths can depend on values, but bounds-safe bit vector accesses are not enforced. A built-in "undefined" value models architecturally unknown values, and an exception-handling mechanism streamlines specification of error cases.

Sail. The Sail language [2] and ecosystem is actively developed by the Rigorous Engineering of Mainstream Systems (REMS) project. The language effectively supports a superset of ASL to permit automatic translation from ASL [1]. A type-and-effect system ensures memory and register accesses are visible at the type level, and the lightweight dependent types of Sail are more expressive than those of ASL. In particular, bit vector accesses are statically bounds-checked, necessitating a flow-sensitive type system: bit vector width constraints are propagated according to program flow and discharged by the Z3 SMT solver.

Sail's ecosystem provides toolchains to translate from ASL to Sail, and from Sail to specifications in Coq and Lem, executable simulators in OCaml and C, and SMT formats. Lem in turn is a lightweight language for engineering reusable semantic models, inspired by both functional programming languages and proof assistants [33, 41]. Its ecosystem provides translations to HOL4 and Isabelle/HOL, amongst others.

L3. L3 [15] is designed to manage the complexity of writing ISA specifications for use in theorem provers: previous work constructed verbose specifications directly in HOL4 [14, 17].

L3 targets HOL4 and Isabelle/HOL directly (rather than via Lem), and has a simpler type system than ASL and Sail, supporting bit vector width restrictions on declared function arguments only. This tight coupling with its backends produces streamlined definitions.

L3's extensive automation minimises the effort of specification-writing and maximises usability within its prover backends. Key features include: prioritisation of abstract syntax for instructions; simultaneous declaration of instruction syntax, encoding, and semantics; symmetric syntax for register reads/writes; and extensibly declarable architectural state.

Unofficial L3 specifications have been produced for many different architectures: Armv4 through to Armv8 (the latter AArch64 mode only), MIPS, x86 (core 64-bit mode instructions only), and RISC-V. The Armv7 specification in particular has been extensively validated against Arm hardware. To keep generated prover specifications idiomatic, L3 Arm specifications re-factor official ASL ones in certain key areas, e.g. preferring bit vectors to integers, sharing common logic where possible, and manually defining efficient instruction decoders.

3 Generating a HOL4 specification for Armv8

Our first step in proving against a HOL4 specification of Armv8 is to generate the specification from official ASL. We rely on the Sail ecosystem, but its design choices have important implications for the resulting specification. In this section we review the extraction process in more depth to aid reproducibility and better inform the rest of the paper. Our generated HOL4 specification is available in a public repository.

The Sail ecosystem is under active development, and we believe we are the first to use a Sail-generated ISA specification for interactive proof in HOL4. The diagram below depicts the pipeline for translation from ASL to HOL4. At present, the process requires some manual intervention, and we are indebted to the Sail developers for helping navigate this space. We examine each stage in turn in the upcoming subsections.

$$\text{ASL} \xrightarrow[\text{Section 3.1}]{\texttt{asl\_to\_sail}} \text{Sail} \xrightarrow[\text{Section 3.2}]{\texttt{sail -lem}} \text{Lem} \xrightarrow[\text{Section 3.3}]{\texttt{lem -hol}} \text{HOL4}$$

3.1 ASL to Sail

The tool `asl_to_sail`[1] translates ASL specifications into Sail. It relies on ASLi ("ASL interpreter") [46] for parsing and type-checking of ASL. The MRA tools collection [47] can be used to extract the input ASL from public, XML-format specifications released by Arm. However, we simply use an ASL specification provided by Arm.

As Sail can be considered a superset of ASL, the translation is relatively naïve. Some optimisations are made, e.g. mutable assignments are turned into immutable let-bindings where possible. However, Sail's richer type system does not accept all ASL programs, so a degree of interactive patching is required. In these cases, `asl_to_sail` halts to request a patch, displaying the original ASL, generated Sail, and the failed typing derivation. The changes required are often straightforward restrictions to permit inference of tighter typing constraints. For example, lifting subexpressions to immutable let-bindings, or specialising type signatures with effect annotations or bit vector width restrictions. However, Sail's type derivation output can be difficult to understand without in-depth knowledge of the typechecker.

[1] `https://github.com/rems-project/asl_to_sail`

We are grateful to Arm Limited and the Sail developers for providing an Armv8.6 A-class ASL specification and the necessary patches to translate it.

3.2 Sail to Lem

Translation from Sail to Lem is more involved: Lem mirrors its HOL4 and Isabelle/HOL backends, so imperative must become functional, and lightweight dependent types must become simple types. We highlight some key aspects of the translation here.

State/exception/non-determinism monad. Sail ships with hand-written Lem libraries encapsulating its built-in types and operations, building on the libraries that ship with Lem itself. These include the implementation of a monad designed to represent imperative, effectful Sail code. During translation, Sail specifications are converted to A-normal form to make explicit the calculation of intermediate values, and embedded into a state/exception/non-determinism monad of the following type (where σ is the state type, α the return type, and ϵ the exception type), with standard return, bind, and exception-handling operators:

$$\mathsf{M}\ \alpha\ \epsilon\ \sigma \stackrel{\mathsf{def}}{=} \sigma \longrightarrow \mathcal{P}\,(\mathsf{Result}\ \alpha\ \epsilon \times \sigma) \qquad\qquad \mathsf{Result}\ \alpha\ \epsilon ::= \mathsf{Value}\ \alpha \ \mid\ \mathsf{Ex}\ \epsilon\,.$$

Non-determinism is modelled by using a set of possible outputs, rather than a single one. Imperative early-return statements are modelled by extending the exception type to a sum, and throwing the early-return value as an exception. The sum type distinguishes between early-return "exceptions" and actual exceptions. All early-return functions are then wrapped with a monadic operator which embeds them back into the original monad.

Data representation. Users can choose to represent bit vectors as lists of three-value logic "trits" or machine-words from default Lem libraries. The Lem libraries for Sail define a typeclass for bit vectors, with trit-list and machine-word instantiations allowing easy switching between the two. There is a key trade-off here: trit-lists are a simpler extraction target, but require extra reasoning about bit vector widths (i.e. list lengths); machine-words have type-backed widths but require further processing (see below) to target Lem's simply-typed setting. We use the machine-word representation: this has stronger library support and more efficient call-by-value evaluation procedures within HOL4.

Sail can refer to registers by reference, so its Lem libraries define a typeclass for register references. Each register is defined by an instance of this typeclass, containing a canonical name and functions to read from/write to the register in the processor state.

Monomorphisation. Sail functions over bit vectors can be both dependently-typed and polymorphic, which is incompatible with Lem's simple types. Arm specifications make heavy use of assertions and case splits on bit widths, producing many dependently-typed functions.

Sail attempts to partially monomorphise such functions: each call-site of the function could in theory produce a width-specialised definition in Lem. The approach is inspired partly by prior work translating ASL to Verilog [48]: case splits are introduced until bit vector widths can remain constant throughout a function, and constant propagation determines these widths. Full monomorphisation is not required as Lem permits polymorphism over bit vector widths (functions over bit vectors can be width-agnostic). If a function can be case-split to maintain consistent bit widths over each case, the various cases can be recombined into a single polymorphic definition. Bit vector extension operations are inserted as necessary to cast types without changing values, and type signatures are simplified until

Lem-compatible. Bit vector slicing operations, which rely heavily on dependent typing, are converted into masking operations wherever possible.

For example, consider a Sail function which accepts any bit vector of width divisible by 8, and returns its length in bytes (of type `forall 'n. bits('n * 8) -> int`). Monomorphisation could produce multiple versions of this function, each with a type specialised to its call-site argument (e.g. of types `bits({8,16,32}) -> int`). However since each specialised version treats the input bit vector uniformly, a single polymorphic definition (of type α word $\rightarrow$ int) suffices. Note that the resulting function accepts bit vectors of widths not divisible by 8, where the original Sail function does not. However, Sail's typechecking guarantees it will never receive such inputs.

This monomorphisation pass can fail, or produce invalid Lem. While Sail remains under active development, the class of specifications which translate error-free is a moving target.

Other code transformations. Scattered functions are collected into single, monolithic functions, with clauses ordered by appearance. Each function clause may be guarded in Sail, so the guards are converted to if-statements.

Both ASL and Sail support an "undefined" built-in, used extensively in Arm specifications to model architecturally unknown values. Lem must explicitly implement this built-in: for each declared type in a specification, Sail automatically generates a function to produce the corresponding undefined value.

Sail unrolls recursive functions whose recursion depth can be determined, removing the need for some termination proofs in theorem prover backends. Users can also manually tweak parts of a specification by providing alternative function implementations to splice in. We make use of this splicing feature to modify our specification (Section 4.1).

3.3 Lem to HOL4

Lem straightforwardly translates to HOL4, though the raw specifications are not human-friendly until parsed/pretty-printed by HOL4: all expressions are type-annotated and fully-bracketed. HOL4 libraries for Sail are automatically translated from the corresponding Lem libraries, and Lem's simple typeclasses are represented as record types.

Some manual intervention is required for well-foundedness checking: Lem automatically generates simple well-foundedness proofs, but some Sail library functions are well-founded for non-trivial reasons. However, naïve definitions for pure and monadic while-looping constructs are not well-founded. We have redefined these functions correctly, deriving theorems showing their behaviour is as originally intended.

We also corrected a minor HOL4 usability issue when working with this substantial specification. We introduced a syntax for let-declarations within monads to enable clear printing of definitions which mix monadic and pure assignments. This permitted interactive inspection of the translated specification.

4 Using the specification for semantics of machine code

We now consider our verification work with respect to the detailed specification we have generated from official ASL. Several previous efforts have used such specifications to prove architectural properties [8, 40, 54, 55]. Instead, we use our HOL4 specification as a semantics for Arm machine code. In particular, we are interested in proofs of *semantics preservation*, equating the semantics of Arm machine code programs with those of other programs (likely

in another language). This is exactly the class of proofs for which L3 and its unofficial specifications were designed.

In this section we modify our specification to adapt it to this goal (Section 4.1), before examining the result more closely (Sections 4.2 and 4.3). Finally, we use an L3 specification to simulate our ASL-derived one, allowing re-use of L3 machinery to reason about ASL-derived semantics of Arm machine code (Section 4.4).

4.1 Modifications to the specification

During translation from ASL to HOL4, we make a number of modifications to our specification. Both original and modified HOL4 models are available in a public repository. We justify our changes here, omitting one minor modification irrelevant to our presentation, that is, disabling a testing harness.

Monad types. We remove the set-based non-determinism from the state/exception/non-determinism monad (Section 3.2), preferring to use HOL4's Hilbert choice operator to express unknown values. This is more idiomatic, and streamlines interactive proof.

More precisely, set-based non-determinism is not compatible with symbolic evaluation. For example, an undefined 64-bit bit vector in Sail is translated to a set of all 2^{64} possible 64-bit bit vectors in HOL4, which is intractable to evaluate. To generate undefined values for enumerated types, Sail libraries create an undefined bit-list of appropriate length and cast it to a natural number, which indexes into a list of all possible elements of the type. This too is more cleanly expressed as a Hilbert choice.

We must be careful with manual changes in a high-fidelity specification. However, our modifications are conservative: we change only the monad implementation in the hand-written Lem libraries for Sail. This gives us confidence in their validity.

Address translation. Address translation is a detail orthogonal to many proofs of semantics preservation: if we assume it is correctly implemented, we can view it as a simple map which abstracts physical memory to virtual memory. This is in keeping with L3 specifications and other specifications modelling the semantics of machine code [11, 56].

We therefore stub out address translation functions to express an identity mapping between virtual and physical memory. However, in Armv8 AArch64 virtual addresses are 64-bit and physical addresses up to 52-bit. Sail's user-splicing feature (Section 3.2) cannot modify types, so we manually modify the specification to convert the type of physical addresses. Again, we keep changes conservative to maintain trust in the specification: we are guided entirely by the Sail typechecker.

4.2 Examining the specification

Figure 1 shows metrics taken throughout the extraction processes of L3 and ASL specifications to HOL4. The difference here is clear — to the best of our knowledge, the ASL-derived HOL4 specification is one of the most complex, unwieldy specifications to be used in interactive proof. Other points of comparison include the 0.59×10^6 character Sail specification of RISC-V [2] (3.6×10^6 characters in raw HOL4), and the 1.7×10^6 character ACL2 specification of x86 [20]. We examine these stark differences in more depth.

Why is the ASL specification so much larger than its peers?

It covers more of its intended ISA, modelling most modes/instruction sets, where other

Original specification	No. non-whitespace characters / 10^6					Size / kLoC		Total time to extract	HOL4 build time
	Source	*Sail*	*Lem*	*Raw HOL4*	*HOL4*	*Source*	*Raw HOL4*		
L3	0.053	-	-	0.20	0.070	2.4	8.5	1 s	$<$ 30 s
ASL	4.2	7.4	19.9	26.7	12.2	168	488	$\sim$ 2 hrs	$\sim$ 3 hrs

Figure 1 Metrics for the extraction processes of L3 and ASL specifications via the L3 and Sail ecosystems respectively. Character counts in HOL4 are shown both as extracted (raw) and after pretty-printing to 80 columns. Timings taken using Intel® Xeon® E-2186G and 64 GB RAM.

specifications tend to formalise a particular mode of operation. For example: the L3 specification covers only AArch64 mode, and also omits vector (SIMD) and floating-point instructions; until recently [10] the ACL2 specification of x86 only covered 64-bit mode. L3 specifications model mostly user-level code, omitting most system registers/instructions.

ASL specifications have differing goals to their peers: primarily they provide documentation, focussing on thoroughness rather than simplicity. Other efforts focus on verification: conciseness and usability are primary goals. For example, the ASL specification defines IEEE floating-point semantics from scratch to ensure faithful behaviour, whereas other efforts outsource to libraries.

Why does extraction to HOL4 bloat the ASL specification?

Representing sequential, imperative code monadically adds considerable bloat due to conversion to A-normal form and instrumentation with explicit monadic operations. Instead, L3 keeps definitions pure wherever possible (looping constructs remain monadic).

Various translation artefacts are verbose, for example: succinct bit vector slicing in Sail is often converted to masking; Sail's undefined built-in primitive must be implemented for each declared type; registers passed by reference in Sail require explicit definitions in HOL4.

L3 couples tightly with prover backends, using relatively simple types and HOL-flavoured constructs to provide an extremely sophisticated syntactic sugar for higher-order logic. Sail is more general-purpose, and not well-optimised for use with HOL4: bit-vector built-ins must be realised in libraries, and these often re-implement HOL4 operations unnecessarily and non-idiomatically (for example, see Example 1).

Why are extraction and build times so long for the ASL pipeline?

Naturally, a significantly larger specification takes longer to extract and build. However, the increase is not proportional to size alone.

The ASL specification must be type-checked multiple times during extraction: in ASL, in Sail, and in Lem. Monomorphisation is a complex process, and lightweight dependent types in ASL/Sail necessitate heavy use of Z3 to discharge constraints.

Sail produces concrete HOL4 syntax via Lem, which must be parsed and type-checked in HOL4. Instead, L3 produces ML constructors which stitch together its definitions. Larger definitions built from Sail's scattered functions are particularly slow, including the decoder. By contrast, L3's decoder is manually defined to minimise its footprint; it tests opcodes in an order which avoids ambiguities from overlapping opcode spaces. The ASL-derived decoder is forced to implement additional machinery to disambiguate opcodes instead.

4.3 Working with the specification

We can now evaluate the practicality of our ASL-derived specification for interactive proofs of semantics preservation. Unfortunately, there are some significant obstacles.

▶ **Example 1.**

```
l3_HighestSetBit x =def=
  if x = 0w then -1
  else w2i (word_log2 x)
```

```
asl_HighestSetBit x =def=
  catch_early_returnS
   (let
     loop_i_lower = 0;
     loop_i_upper = n2i (word_len x) - 1
    in
     do
      foreachS (index_list loop_i_upper loop_i_lower (-1)) ()
       (λ i unit_var.
          if vec_of_bits [access_vec_dec x i] = 1w then
           early_returnS i
          else returnS ());
      returnS (-1)
     od)
```

Opaqueness to inspection and interaction. The ASL-derived specification does not implement an AST for Arm instructions, in contrast to the L3 specification. Users must work directly with opcodes or manually define an AST.

Non-idiomatic data manipulations and explicit monadic operations obfuscate high-level semantics. These are awkward and tedious in interactive proofs, which must step over each monadic operation. L3 instead prefers HOL4's let-declarations, mostly removing monadic sequencing.

Monolithic HOL4 functions coalesced from ASL's scattered functions are unwieldy, e.g. the 23 kLoC decoding/execution function DecodeA64 (for which each instruction implements a clause). Examining the semantics of a particular instruction is therefore challenging.

ASL specifications and Sail libraries use many auxiliary functions: many steps of definition expansion are required to examine or use intended semantics.

Opaqueness to automated evaluation. Non-idiomatic bit vector operations have poor evaluation support in HOL4, often re-implementing functionality in libraries shipped with HOL4. Some operations even convert machine-word operands to bit-lists, perform operations on bit-lists, and convert back. In these cases, permeative book-keeping of list lengths complicates reasoning. Integers are used throughout even when known to be positive, despite the comparative ease-of-use of natural numbers.

Consider the L3-derived (left) and ASL-derived (right) HOL4 definitions in Example 1, which determine the index of the highest set bit of input bit vector x. The L3-derived definition uses HOL4 library definitions to convert the base-2 logarithm of the input word to an integer (w2i). The ASL-derived definition is more computational: it examines each bit from high to low, returning early if any is set. However, this intended definition is obfuscated: the early return necessitates embedding within the Sail monad and use of early-return monadic operations (catch_early_returnS, early_returnS); explicit monadic operations are used for looping and sequencing (made more palatable here by do notation); integer loop variables are used over natural numbers (n2i casts from natural to integer); custom bit vector operations from Sail libraries are used (vec_of_bits and access_vec_dec). The definitions of these last

▶ **Definition 2.**

$$\begin{aligned}
&\mathsf{l3_models_asl}\ \mathit{opcode} \stackrel{\mathsf{def}}{=} \\
&\quad \mathsf{Decode}\ \mathit{opcode} \neq \mathsf{Unallocated} \wedge \\
&\quad \forall\, \mathit{l3}\ \mathit{asl}\ \mathit{l3}'. \\
&\qquad \mathsf{state_rel}\ \mathit{l3}\ \mathit{asl} \wedge \mathsf{asl_sys_regs_ok}\ \mathit{asl} \wedge \mathsf{Run}\ (\mathsf{Decode}\ \mathit{opcode})\ \mathit{l3} = \mathit{l3}' \wedge \\
&\qquad \mathit{l3}'.\mathsf{exception} = \mathsf{NoException} \Rightarrow \\
&\qquad \exists\, v\ \mathit{asl}'. \\
&\qquad\quad \mathsf{ExecA64}\ \mathit{opcode}\ \mathit{asl} = (\mathsf{Value}\ v, \mathit{asl}') \wedge \mathsf{state_rel}\ \mathit{l3}'\ \mathit{asl}' \wedge \\
&\qquad\quad \mathsf{asl_sys_regs_ok}\ \mathit{asl}'
\end{aligned}$$

$$\begin{aligned}
&\mathsf{l3_models_asl_instr}\ \mathit{instr} \stackrel{\mathsf{def}}{=} \\
&\quad \exists\, \mathit{opcode}.\ \mathsf{Encode}\ \mathit{instr} = \mathsf{ARM8}\ \mathit{opcode} \wedge \mathsf{l3_models_asl}\ \mathit{opcode}
\end{aligned}$$

operations involve a large number of auxiliary functions: they convert x to a list of booleans, access the boolean at index i, and use it to create a 1-bit bit vector. This re-implements HOL4's well-supported bit vector access operation. This small example exemplifies the increased complexity which pervades the ASL-derived specification.

4.4 Simulation between L3 and ASL-derived specifications

Definition 2 expresses our simulation relation, l3_models_asl, a predicate on the AST for instructions defined by the L3 specification. The instruction should successfully encode (using the L3-specified Encode) to produce a 32-bit *opcode*. Given L3 and ASL-derived machine states related by state_rel, if the L3 specification can decode and run (Run (Decode *opcode*)) the opcode without failure, the ASL-derived specification should also run (ExecA64) it successfully, both producing resultant states that remain related by state_rel. In addition, the predicate asl_sys_regs_ok should hold of the ASL-derived state throughout.

Note that we rely on L3 machinery: its AST for instructions and its encoder. These are orthogonal to the semantics of instructions, but use of an AST is well-suited to interactive proof and makes it simpler to carve out classes of opcodes. We have already noted that the ASL-derived specification does not provide such an AST or encoder.

The state equality relation state_rel is effectively a simple inclusion: the ASL-derived specification models strictly more registers and processor state than the L3 one, so state_rel asserts that the specifications agree on the parts modelled by both. We omit the full definition, which is verbose due to differences between the two specifications. In particular, L3 machine states (*l3*) are concise records, with clean access to registers, e.g. *l3*.PC is the program counter. Instead, ASL-derived states (*asl*) group registers together by their types, e.g. *asl*.regstate.bitvector_64_dec_reg of type string → word64 models all simple 64-bit system registers. The sheer number of registers forces this unusual approach: HOL4 struggles to cope with very large records in which each register is declared separately, so they must be grouped. Register references (*reg*) are used to index into these groups: these records contain a canonical name (*reg*.name), and read-from/write-to functions (*reg*.read_from, *reg*.write_to). For example, PC_ref.read_from *asl*.regstate reads the program counter. This is equivalent to *asl*.regstate.bitvector_64_dec_reg "_PC".

We also explore a few subtleties that arise.

The L3 specification models Armv8.0, whereas the ASL-derived specification models Armv8.6. Though this is a "minor" version difference, there are observable effects, e.g. certain system control registers are 32-bit in L3 but 64-bit in ASL. In Armv8.0, these registers were 64-bit with their upper 32 bits reserved, so only their lower 32 bits required modelling. However, ASL faithfully models all 64 bits regardless.

The L3 specification represents memory cleanly as a total function from addresses to bytes (word64 $\rightarrow$ word8). However, the ASL-derived specification models memory as a finite mapping from natural numbers to trit-lists (num $\mapsto$ trit list), each intended to represent a byte. It also models per-address validity tags (num $\mapsto$ trit). We impose restrictions on ASL-derived memory to ensure we can equate it to L3-derived memory: its domain must be exactly the natural numbers representable by 64-bit words; its range must contain only *bit*-lists (i.e. no "unknown" trits) of length 8; all addresses must have a valid tag.

To model Arm's 31 general-purpose registers and zero register, the L3 specification uses a total mapping from 5-bit words (word5 $\rightarrow$ word64). The ASL specification instead uses a list of registers (word64 list). To access the register n, it takes the nth index of the list. We must require the list to have length of exactly 32.

The predicate asl_sys_regs_ok is necessary as L3 specifications model mostly user-level operation, omitting most system registers. We fix certain bits of these registers in the ASL-derived specification to ensure it models a processor in a similar mode of operation. We omit its definition here, but refer readers to our proofs and the Arm Architecture Reference Manual for a full account. Overall, we fix 11 bits of various system registers, and clear one memory-mapped register. This disables optional features not modelled in L3, such as secure modes for low exception levels and hypervisors.

Due to a versioning difference, four of these bits are in the TCR_EL{1,2,3} registers which are also modelled in L3. A feature implementing pointer authentication codes (PACs) was made compulsory in Armv8.3 onward, supporting authentication of addresses stored in registers before targeting them for a branch or load. Fixing these bits in the Armv8.6 modelled by ASL aligns behaviour more to the Armv8.0 modelled by L3.

Proving the simulation

Establishing simulation for a particular instruction effectively requires execution of the instruction on both specifications. We leverage pre-existing automation to execute the L3 specification effectively. However, the difficulties detailed in Section 4.3 complicate execution of the ASL-derived specification.

We adopt a partial, symbolic evaluation strategy to bypass decoding, which examines only known bits of opcodes (i.e. those that distinguish it from other classes of opcode). More precisely, we use HOL4's customisable call-by-value computation library to bypass the large, monolithic ExecA64 (a wrapper around the overall instruction decoding/execution function, DecodeA64). Once this has advanced the proof to the instruction-specific execution functions to which ExecA64 calls, we proceed by interactive proof.

We use the L3 specification heavily to provide abstract representations of auxiliary functions, avoiding unwrapping definitions and making case splits. For example, for the definitions in Example 1, we prove the following theorem for any 7-bit bit vector w:

$\vdash$ asl_HighestSetBit w = returnS (l3_HighestSetBit w).

A notable sub-proof relates the L3 and ASL-derived implementations of DecodeBitMasks, used in decoding to resolve immediate fields. Its 90 LoC ASL implementation is obfuscated by its optimised implementation, but a comment block asserts an equivalent 7 LoC definition.

The L3 specification uses the shorter definition, so we must prove the asserted equivalence: we split up the large function into more manageable chunks, proving more presentable, manually-defined specifications for each by brute force enumeration of inputs.

In total, we prove l3_models_asl for the following instruction classes, requiring 7.5 kLoC of proof and 0.5 kLoC of simple automation (∼40 mins to build, limited by symbolic evaluation).

Instruction class description	Assembly shorthands
move wide operations	MOVK, MOVN, MOVZ
bit field moves	BFM, SBFM, UBFM
logical operations*†	AND[S], BIC[S], EON, EOR, ORN, ORR
addition/subtraction*†	ADD[S], SUB[S]
addition/subtraction with carry	ADC[S], SBC[S]
division	SDIV, UDIV
multiply with addition/subtraction	MADD, MSUB
multiply high	SMULH, UMULH
conditional compare*	CCMN, CCMP
conditional select	CSEL, CSINC, CSINV, CSNEG
branch immediate (call/jump)	B, BL
conditional branches	B.COND
branch register (jump)	BR
register extract	EXTR
address calculation	ADR, ADRP
byte/register loads/stores*‡	LD[U]R, LD[U]R[S]B, ST[U]R, ST[U]RB

Our reliance on the existing L3 specification keeps the proofs tractable, and no unexpected discrepancies were found in its semantics or encoder (we do encounter a known issue, see Section 8). All of our definitions and proofs thus far are self-contained, and not tied to any particular usage of our ASL-derived specification. To aid re-use in future work, they have been integrated into the HOL4 public repository.

5 Case study: compiler correctness in CakeML

Verification efforts such as CakeML and seL4 build correctness guarantees down to hardware by using unofficial L3 specifications of ISAs as a reference semantics. Equipped with our ASL-derived HOL4 specification (Section 3) and simulation proofs (Section 4.4), we can strengthen assurances in these results. We demonstrate our approach by proving a new compiler correctness result for CakeML (Section 5.2), targeting our ASL-derived specification.

What is CakeML? CakeML [27, 35] is a formally-specified language, end-to-end verified compiler, and proof ecosystem built using HOL4. The compiler is proved to preserve semantics: any valid input program is compiled to machine code with the same behaviour. The core of the project is a proof-producing translation from HOL4 to CakeML: input HOL4 functions are translated to output CakeML abstract syntax, and an accompanying proof that the output semantics models the input HOL4 function [38]. The compiler is bootstrapped by first translating the HOL4 compiler function to CakeML, then evaluating the HOL4 compiler on the output CakeML within the HOL4 logic. This produces a binary which is verified to implement the original HOL4 compiler algorithm.

* For immediate operands.
† For shifted register operands.
‡ Scaled 12-bit unsigned immediate offset and unscaled 9-bit signed immediate offset addressing modes.

5.1 Target correctness proofs in CakeML

CakeML targets x86-64, Armv7, Armv8 (AArch64), RISC-V, MIPS, and Silver (a custom ISA for a verified processor [32]), proving compiler correctness with respect to specifications for each. These are all L3-derived, but we are concerned with Armv8 AArch64.

Both the compiler and its proofs are carefully structured to remain as target-agnostic as possible, reducing the implementation burden and proof obligation of supporting a new target. The following design decisions are made [18].

- The compiler produces generic assembly instructions known as Asm, which are embedded in its final intermediate language (LabLang). Defining an encoding to a new target requires a simple translation from this generic assembly.
- The compiler implementation is parametric over a target-specific configuration record, known as a "compiler configuration". For example, this record defines: an encoder from Asm, the registers useable in register allocation, classes of supported operations, and restrictions on valid immediate values. Compiling to a new target requires definition of an appropriate compiler configuration.
- Compiler correctness proofs use a generic form of target semantics (machine_semantics), parametrised by another kind of target-specific record known as a "machine configuration". All target-specific requirements of the proof are factored out into a precondition (target_configs_ok). Proving correctness for a new target requires definition of its machine configuration and discharging of the precondition.

We re-use the L3 encoder from the existing Armv8 backend. Therefore, to instantiate the generic compiler correctness theorem we must define a *compiler_config* and *machine_config*, and establish the precondition target_configs_ok *compiler_config machine_config*.

Machine configurations (*mc*) define exactly the features of a target necessary to define machine_semantics. This generic semantics models interference from the surrounding execution environment by allowing the environment to change a subset of target state arbitrarily between instructions and on FFI calls. Important fields include: a function to execute a single step (*mc*.target.next); accessors of registers, the program counter, and memory (*mc*.target.get_{reg,pc,byte}); information about the calling convention (*mc*.callee_saved_regs); a well-formedness predicate on target state (*mc*.target.state_ok); and a projection out of target state (*mc*.target.proj). This projection is the subset of target state which must *not* be modified by the surrounding execution environment, i.e. *mc*.target.proj is always the same both before and after external interference.

A proof of target_configs_ok establishes some simple initialisation conditions, and the key property encoder_correct *mc*.target. We omit a full definition, but intuitively: given an Asm state, an Asm instruction which will successfully retire, and an equivalent target state with the encoded Asm instruction in memory (using *mc*.config.encode), the target should be able to execute some number of steps successfully (according to *mc*.target.next) and end up in a state equivalent to the resulting Asm state. Note that a single Asm instruction may be encoded to a sequence of target opcodes, so multiple execution steps may be necessary. The property accounts for the surrounding environment by permitting interference by a function satisfying interference_ok (i.e. the interference must preserve *mc*.target.proj). Target states must further satisfy *mc*.target.state_ok throughout. We omit details concerning program counters of the encoded instructions in memory, which must not be overwritten during execution.

Proving encoder_correct effectively requires running Asm and target state machines side-by-side. Therefore, proofs rely heavily on symbolic evaluation within HOL4, using L3-enabled automation to repeatedly apply *mc*.target.next and re-establish *mc*.target.state_ok after each step and its associated interference.

▶ **Theorem 3.**

$\vdash$ mem *instr* (asm_to_arm8 *prog*) $\wedge$
($\forall$ *s*. Encode *instr* $\neq$ BadCode *s*) $\Rightarrow$
l3_models_asl_instr *instr*

▶ **Definition 4.**

```
NextASL ≝
 do
  write_regS BranchTaken_ref F;
  pc ← PC_read ();
  instr ← Mem_read0 pc 4 AccType_IFETCH;
  ExecA64 instr;
  branch_taken ← read_regS BranchTaken_ref;
  if branch_taken then returnS ()
  else
   do pc ← PC_read (); PC_set (pc + 4w) od
 od
```

5.2 Lifting simulation to compiler correctness

We first prove Theorem 3: l3_models_asl (Definition 2) holds for any encodeable instruction produced by the CakeML compilation to Armv8 via Asm.

We then define a machine configuration (Section 5.1) for the ASL-derived specification, which largely mirrors the existing one for the L3 specification. It additionally enforces asl_sys_regs_ok (extending the state projection *mc*.target.proj to be sure that interference cannot break it), and ensures that memory and registers are well-formed (Section 4.4).

We must also define a next-step function (*mc*.target.next). Though the ASL specification provides a function TopLevel, this covers unwanted extraneous details, such as processor interrupts and memory-mapped devices. Instead we define NextASL (Definition 4), essentially TopLevel with the complexity stripped away. NextASL clears the branch-taken flag, reads the program counter, fetches the next opcode, and executes the opcode. The program counter is then updated only if no branch has been taken.

The bulk of the proof-effort is establishing encoder_correct. Interference from the surrounding execution environment prevents us from re-using the theorem proved for the L3 specification. We would require once and for all a transformation from interference with ASL-derived target states (satisfying interference_ok) to interference with L3 target states, such that the transformation preserves state_rel (to allow use of our simulation proofs). We cannot express this transformation: interference on ASL-derived specifications has a larger input space than that on L3 specifications, due to processor state not modelled in L3.

Instead, we run *three* state machines side-by-side: Asm, L3, and ASL-derived. We re-use pre-existing automation for the L3 specification to symbolically evaluate each opcode, giving a resultant L3 state, and then use Theorem 3 to produce a resultant ASL-derived state. This undergoes interference, breaking state_rel. We then interfere with the L3 state in a way which satisfies interference_ok and re-establishes state_rel between the resultant states. We repeat the process for as many opcodes as necessary for each Asm instruction encoding.

Overall, we produce Theorem 5, our top-level compiler correctness proof (mostly mechanical, 3.2 kLoC, ∼15 mins to build). A program with non-failing source semantics is compiled and installed in the memory of an appropriate *asl_state*. The resulting machine will have a machine_semantics which is a subset of the original source_semantics. We omit irrelevant details concerning CakeML's FFI model and extend_with_resource_limit. The latter extends source behaviours with out-of-memory errors: unlike source semantics, machine semantics is bounded by finite memory. This proof has been integrated into the CakeML public repository.

▶ **Theorem 5.**

⊢ Fail ∉ source_semantics *ffi program* ∧
compile arm8_compiler_config *program* = Some *compiled* ∧
asl_machine_config_ok *machine_config* ∧
program_in_memory arm8_compiler_config *machine_config compiled asl_state* ⇒
machine_semantics *machine_config ffi asl_state* ⊆
extend_with_resource_limit (source_semantics *ffi program*)

6 Discussion

We now consider our findings: what can we learn from our verification?

Trustworthiness of our specifications. The ASL specification cannot cover all aspects of its ISA, instead providing a working reference. When considering more complex features, such as concurrency and interrupts, it remains an abstraction of the authoritative detail of the Arm Reference Manual. However, even if a complete, machine-readable specification existed, proofs of semantics preservation with respect to it would be intractable without the abstractions. The ASL specifications and our work are a best-effort, modelling as much detail as currently feasible. Other proof goals (such as architecture security properties) may require different levels of detail.

Our root of trust is the extensive Arm-internal evaluation of the ASL specification, but extraction via Sail could introduce unintended semantic changes. Validating the generated HOL4 against Arm test suites or real hardware could improve trust (Sail's C backend has been tested in this way), but proving that the extraction preserves semantics is better still. This would be a significant undertaking, and require formal models of (at least) ASL and Sail. There has been some work into such models for both ASL (from personal communication) and Sail [4]. Note that both Sail and Lem have been validated through heavy usage. For example, Sail-extracted ASL models have successfully been used to simulate a Linux boot, and the Lem ecosystem is well-exercised. However, Sail's HOL4 backend has received limited prior usage, so our work better validates this pathway too.

Our simulation proofs (Section 4.4) allow each specification to improve trust in the other. The L3 and ASL-derived specifications differ in their derivations considerably, yet are formally connected by our work. Therefore, any bugs found in one must be found in the other, and the low likelihood of this strengthens assurances in both. Previously, the L3 Armv8 specification had not been rigorously validated due to the scarcity of Armv8 hardware when it was written. By contrast, the L3 Armv7 specification was tested extensively against real hardware.

Even so, we strengthen trust in a single L3 specification, and a single extraction pathway via Sail. There are many such specifications that can be generated, via different extraction options, versioning differences, choices in manual modification, and so on.

Absence of bugs. We discovered no new bugs in the L3 specification semantics or encoder (we encountered one known issue, see Section 8), despite the differing provenances of the L3 and ASL-derived specifications. This validates the approach of formalising a specification by using a DSL (L3) which can closely mirror it. However, the absence of bugs is surprising given that the ASL-derived specification covers implementation-defined behaviour and architecturally unknown values.

A partial explanation is our restricted domain of proofs of semantics preservation: we verify general-purpose instructions targeted by compilers, which avoid ambiguity to ensure portability. As a verified compiler, CakeML targets an even smaller subset of instructions.

Removal of address translation and interrupts (Sections 4.1 and 4.4 respectively) further reduces ambiguity significantly. We also do not tackle exception-handling, assuming that instructions retire without failure in the L3 specification (Definition 2, Section 4.4).

Furthermore, Arm intentionally reduced underspecification in Armv8 (compared to Armv7). For example: in Armv7 the program counter is a general-purpose register (R15) which can be modified unexpectedly by programmers; in Armv8 there is no direct access to the program counter. Architecturally unknown values are also mostly used as placeholders for variables which are declared and only later initialised.

The need for L3. We build our results on existing, unmodified tools: an ASL specification and the Sail ecosystem. Neither is designed for interactive theorem proving, so we use a purpose-built L3 specification as a stepping stone in proof. Instead, could we obviate the need for this indirection by adapting the tools to our domain?

One approach is to change the official ASL specification, though this would require support from Arm. Stylistic refactoring could reduce overly imperative code (e.g. Example 1). Logical refactoring has recently streamlined address translation, and could be applied to other parts of the specification. Forbidding early-return statements could reduce embedding of otherwise pure functions into the monad, but this is a significant language change. Alternatively ASL-to-ASL transformations before translation to Sail could achieve similar effects, without compromising our root of trust: the resultant ASL can be subjected to the Arm-internal test suite. Semantics-preserving transformations are already used in Arm to produce model-checking-friendly Verilog (from personal communication).

Another approach is to strengthen Sail's extraction to HOL4, taking inspiration from L3. Streamlining of Sail libraries for HOL4 is a first step. Extraction would also need to produce an AST for instructions. The challenge here is reproducing L3's ease of use: its AST is handcrafted for theorem proving (i.e. split into instruction classes for convenience and to avoid scaling issues with HOL4 types). However, Sail's extraction must be automatic and target-agnostic. Design choices made here will suit different domains, for example the AST could mirror assembly syntax or reference manual structure. A direct translation from ASL or Sail to HOL4 (cutting out Lem) could also streamline the extraction process and more closely target HOL4, but this is a significant undertaking.

Direct proof without L3 would also require new proof automation. This is because verification with respect to the L3 Armv8 specification relies heavily on its *step library*: concise Hoare triples which provide tractable, rule-based instruction semantics in interactive proof by explicitly stating conditions for well-definedness. No such library exists for the ASL-derived specification, but several have been painstakingly handcrafted for various other specifications. Automatic generation of certified step libraries using symbolic evaluation and symbolic execution [7] is a promising approach. A key challenge will be navigating the large, monolithic decoding/execution functions.

Though we have identified some promising avenues to verification without L3, none provides a magic bullet and taken together they represent a significant body of work. As industrial specifications of the scale of Armv8 become more prevalent, such engineering issues will be critical. We note that despite HOL4's veteran status amongst interactive theorem provers, its customisable simplifier, call-by-value evaluation, and highly accessible Standard ML metaprogramming make it an ideal candidate for work on such substantial specifications.

7 Future work

Removing the precondition to Theorem 3 is a clear next step, perhaps complicated by further incompatibilities between the L3 specification's Armv8.0 and the ASL-derived specification's Armv8.6. Upgrading the L3 to Armv8.6 and modelling a greater subset of the ISA could help. Notably, AArch32 and floating point instructions are not modelled and so remain unsupported by CakeML on Armv8. Connecting ASL floating point definitions with HOL4 libraries would be a significant challenge here.

The seL4 [25, 26] verified operating system microkernel uses a translation validation [57] phase to extend its guarantees to the binary on Armv7. An L3 specification provides the semantics for Armv7, but using an ASL-derived specification instead would strengthen the result. However, seL4 would need to be updated to target Armv8 first.

A more faithful account of address translation would improve trust in our work. We could take inspiration from the Sail developers, who prove in Isabelle/HOL that address translation adheres to a hand-written specification under certain conditions [2]. Recent simplifications to Arm's ASL specification of address translation may help here.

8 Related work

Applications of the Sail toolchain. Isla [3] is an SMT solver-based symbolic execution engine for Sail, producing simplified, per-opcode instruction execution traces derived from user-supplied constraints on machine state. Islaris [56] builds on this by formalising the traces in Coq and building a separation logic for reasoning about their semantics, using automation to simplify proofs considerably. Our clear common aim is to derive simplified machine code semantics from the complex ASL-derived specifications, using the desired domain to constrain semantics appropriately. Islaris' domain is machine code verification, so it relies on an external SMT solver on a per-opcode basis to provide sufficient automation. Our domain of semantics preservation instead allows us to abstract once and for all within the prover.

Sail is integral in key security proofs with respect to Morello, the Arm implementation of the Capability Hardware Enhanced RISC Instructions (CHERI) ISA [5, 40]. CHERI extends conventional ISAs with new features enabling memory protection, defending against many memory-based security exploits [58, 59]. The Isabelle/HOL proof (with some SMT solver oracles for bit vector operations) is complex, requiring hours to build with considerable computing power. For such security properties, monadic representation of specifications may be useful. In our case it obfuscates the high-level semantics of the instruction.

Validation of ISA specifications. The Provably Secure Execution Platforms for Embedded Systems (PROSPER) project has created Scam-V [39], an automatic validator for ISA specifications. It searches for pairs of executions which behave identically according to the specification, but are distinguishable on hardware via some observation. If such a pair exists, the specification has failed to model an observable side-channel correctly.

Scam-V has uncovered a bug in the L3 specification of Armv8: the compare and branch on non-zero instruction (CBNZ) incorrectly behaves like the compare and branch on zero instruction (CBZ). We have replicated this finding in a failed attempt to prove l3_models_asl for CBNZ. Furthermore, we have validated a corrected specification of CBNZ by successfully establishing l3_models_asl for it. Note that the CakeML compiler does not produce CBNZ instructions, so our CakeML proofs are unaffected.

Verified compilation to hardware. Erbsen et al. [11] have verified an application on a realistic embedded stack which includes a C-based language, verified compiler, and verified processor (the Kami [9] verified implementation of RISC-V). Their goals contrast with ours: they focus on a complete result down to a specific processor, prioritising clean verification across interfaces and realistic I/O. However, we must remain processor-agnostic and so are forced to verify against an ISA specification only.

Verification with respect to the Arm architecture. Official Arm ISA specifications have been used within Arm to establish architectural properties below the ISA abstraction boundary. These efforts use automated techniques (SMT solving and bounded model-checking) to suit their problem domain and their setting in industry.

ISA-Formal [49,55] uses bounded model-checking to verify that Arm ISA implementations adhere to their intended specification. The project translates ASL specifications to reference Verilog implementations [51], comparing these to the actual implementation using an off-the-shelf bounded model-checker. Failures are output as counterexamples.

Secure-M [54] produces key security properties for the Arm M-class specification, using an automated SMT solver to verify they hold. It provides a more rigorous alternative to code review and testing for architecture modifications, focussing on whole-specification properties for continuous integration testing. There is also some support for developer-stated assertion-checking. A future goal is to prove that successive versions of specifications preserve necessary backwards-compatibility.

9 Conclusion

To the best of our knowledge, we have produced the first compiler correctness proof which is backed by an official specification of an Arm ISA. Our proof is made tractable by the use of an existing L3 specification to abstract away the complexity of the official ASL. This approach validates the L3 specification more rigorously, and we report no new bugs uncovered. Our work remains decoupled from CakeML for re-use in future verification efforts, and has been incorporated in the HOL4 and CakeML public repositories.

Throughout we have kept in mind the issue of trustworthiness. Our work strengthens assurances not only in CakeML, but also in other projects relying on the L3 specification of Armv8. We have reproduced results from the Sail ecosystem and further validated its extraction process. Furthermore, we have demonstrated a novel application of the Sail ecosystem to proofs of semantic preservation.

References

1 Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Shaked Flur, Kathryn E. Gray, Prashanth Mundkur, Robert M. Norton, Christopher Pulte, Alastair Reid, Peter Sewell, Ian Stark, and Mark Wassell. Detailed models of instruction set architectures: From pseudocode to formal semantics. In *Proceedings of the Automated Reasoning Workshop*, 2018. Two-page abstract. URL: `http://www.cl.cam.ac.uk/~pes20/sail/2018-04-12-arw-paper.pdf`.

2 Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages (PACMPL)*, 3(POPL), 2019. `doi:10.1145/3290384`.

3 Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *Computer Aided*

Verification (CAV), volume 12759 of *Lecture Notes in Computer Science*. Springer, 2021. doi:10.1007/978-3-030-81685-8_14.

4 Alasdair Armstrong, Neel Krishnaswami, Peter Sewell, and Mark Wassell. Formalisation of MiniSail in the Isabelle theorem prover. In *Proceedings of the Automated Reasoning Workshop*, 2018. Two-page abstract. URL: https://www.cl.cam.ac.uk/~pes20/sail/arw18_mpew2.pdf.

5 Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. Verified security for the Morello capability-enhanced prototype Arm architecture. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer, 2022. To appear. URL: http://www.cl.cam.ac.uk/~pes20/morello-proofs-esop2022.pdf.

6 Christoph Baumann, Mats Näslund, Christian Gehrmann, Oliver Schwarz, and Hans Thorsen. A high assurance virtualization platform for ARMv8. In *European Conference on Networks and Communications (EuCNC)*. IEEE, 2016. doi:10.1109/EuCNC.2016.7561034.

7 Brian Campbell and Ian Stark. Extracting behaviour from an executable instruction set model. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2016. doi:10.1109/FMCAD.2016.7886658.

8 Adam Chlipala. Algorithmic checking of security arguments for microprocessors. In *GOMAC-Tech Conference*, 2019. URL: https://apps.dtic.mil/sti/citations/AD1075652.

9 Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP), 2017. doi:10.1145/3110268.

10 Alessandro Coglio and Shilpi Goel. Adding 32-bit mode to the ACL2 model of the x86 ISA. In *Workshop on the ACL2 Theorem Prover and Its Applications*, volume 280 of *EPTCS*, 2018. doi:10.4204/EPTCS.280.6.

11 Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. Integration verification across software and hardware for a simple embedded system. In *Programming Language Design and Implementation (PLDI)*. ACM, 2021. doi:10.1145/3453483.3454065.

12 Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Principles of Programming Languages (POPL)*. ACM, 2016. doi:10.1145/2837614.2837615.

13 Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *Principles of Programming Languages (POPL)*. ACM, 2017. doi:10.1145/3009837.3009839.

14 Anthony C. J. Fox. Formal specification and verification of ARM6. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2758 of *Lecture Notes in Computer Science*. Springer, 2003. doi:10.1007/10930755_2.

15 Anthony C. J. Fox. Directions in ISA specification. In *Interactive Theorem Proving (ITP)*, volume 7406 of *Lecture Notes in Computer Science*. Springer, 2012. doi:10.1007/978-3-642-32347-8_23.

16 Anthony C. J. Fox. Improved tool support for machine-code decompilation in HOL4. In *Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*. Springer, 2015. doi:10.1007/978-3-319-22102-1_12.

17 Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Interactive Theorem Proving (ITP)*, volume 6172 of *Lecture Notes in Computer Science*. Springer, 2010. doi:10.1007/978-3-642-14052-5_18.

18 Anthony C. J. Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. Verified compilation of CakeML to multiple machine-code targets. In *Certified Programs and Proofs (CPP)*. ACM, 2017. doi:10.1145/3018610.3018621.

19 Shilpi Goel and Warren A. Hunt Jr. Automated code proofs on a formal model of the X86. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 8164 of *Lecture Notes in Computer Science*. Springer, 2013. doi:10.1007/978-3-642-54108-7_12.

20 Shilpi Goel, Warren A. Hunt Jr., and Matt Kaufmann. Engineering a formal, executable x86 ISA simulator for software verification. In *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering. Springer, 2017. doi:10.1007/978-3-319-48628-4_8.

21 Shilpi Goel, Anna Slobodová, Rob Sumners, and Sol Swords. Verifying x86 instruction implementations. In *Certified Programs and Proofs (CPP)*. ACM, 2020. doi:10.1145/3372885.3373811.

22 Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Microarchitecture (MICRO)*. ACM, 2015. doi:10.1145/2830772.2830775.

23 Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2016. doi:10.5555/3026877.3026928.

24 Roberto Guanciale, Hamed Nemati, Mads Dam, and Christoph Baumann. Provably secure memory isolation for linux on ARM. *Journal of Computer Security*, 24(6), 2016. doi:10.3233/JCS-160558.

25 Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Operating Systems Principles (SOSP)*. ACM, 2009. doi:10.1145/1629575.1629596.

26 Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1), 2014. doi:10.1145/2560537.

27 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *Principles of Programming Languages (POPL)*. ACM, 2014. doi:10.1145/2535838.2535841.

28 Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Formal Methods (FM)*, volume 5850 of *Lecture Notes in Computer Science*. Springer, 2009. doi:10.1007/978-3-642-05089-3_51.

29 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 2009. doi:10.1145/1538788.1538814.

30 Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A secure and formally verified linux KVM hypervisor. In *Security and Privacy (SP)*. IEEE, 2021. doi:10.1109/SP40001.2021.00049.

31 Andreas Lindner, Roberto Guanciale, and Roberto Metere. TrABin: Trustworthy analyses of binaries. *Science of Computer Programming*, 174, 2019. doi:10.1016/j.scico.2019.01.001.

32 Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony C. J. Fox. Verified compilation on a verified processor. In *Programming Language Design and Implementation (PLDI)*. ACM, 2019. doi:10.1145/3314221.3314622.

33 Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In *International Conference on Functional Programming (ICFP)*. ACM, 2014. doi:10.1145/2628136.2628143.

34 Magnus O. Myreen. Verified just-in-time compiler on x86. In *Principles of Programming Languages (POPL)*. ACM, 2010. doi:10.1145/1706299.1706313.

35 Magnus O. Myreen. The CakeML project's quest for ever stronger correctness theorems (invited paper). In *Interactive Theorem Proving (ITP)*, volume 193 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.1.

36 Magnus O. Myreen and Jared Davis. A verified runtime for a verified theorem prover. In *Interactive Theorem Proving (ITP)*, volume 6898 of *Lecture Notes in Computer Science*. Springer, 2011. doi:10.1007/978-3-642-22863-6_20.

37 Magnus O. Myreen and Michael J. C. Gordon. Verified LISP implementations on ARM, x86 and PowerPC. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009. doi:10.1007/978-3-642-03359-9_25.

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

39 Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. Validation of abstract side-channel models for computer architectures. In *Computer Aided Verification (CAV)*, volume 12224 of *Lecture Notes in Computer Science*. Springer, 2020. doi:10.1007/978-3-030-53288-8_12.

40 Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony C. J. Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *Security and Privacy (SP)*. IEEE, 2020. doi:10.1109/SP40000.2020.00055.

41 Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell. Lem: A lightweight tool for heavyweight semantics. In *Interactive Theorem Proving (ITP)*, volume 6898 of *Lecture Notes in Computer Science*. Springer, 2011. doi:10.1007/978-3-642-22863-6_27.

42 Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018. doi:10.1145/3158107.

43 Alastair Reid. Arm releases machine readable architecture specification. URL: https://alastairreid.github.io/ARM-v8a-xml-release/.

44 Alastair Reid. Arm v8.3 machine readable specification released. URL: https://alastairreid.github.io/arm-v8_3/.

45 Alastair Reid. Arm's architecture specification language. URL: https://alastairreid.github.io/specification_languages/.

46 Alastair Reid. ASL lexical syntax. URL: https://alastairreid.github.io/using-asli/.

47 Alastair Reid. Dissecting Arm's machine readable specification. URL: https://alastairreid.github.io/dissecting-Arm-MRA/.

48 Alastair Reid. Formal validation of the Arm v8-M specification. URL: https://alastairreid.github.io/validating-specs/.

49 Alastair Reid. Limitations of ISA-Formal. URL: https://alastairreid.github.io/isa-formal-limitations/.

50 Alastair Reid. Using ASLi with Arm's v8.6-A ISA specification. URL: https://alastairreid.github.io/asl-lexical-syntax/.

51 Alastair Reid. Verifying against the official Arm specification. URL: https://alastairreid.github.io/using-armarm/.

52 Alastair Reid. What can you do with an ISA specification? URL: https://alastairreid.github.io/uses-for-isa-specs/.

53 Alastair Reid. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2016. doi:10.1109/FMCAD.2016.7886675.

54 Alastair Reid. Who guards the guards? formal validation of the Arm v8-M architecture specification. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(OOPSLA), 2017. doi:10.1145/3133912.

55 Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of

processors with ISA-Formal. In *Computer Aided Verification (CAV)*, volume 9780 of *Lecture Notes in Computer Science*. Springer, 2016. doi:10.1007/978-3-319-41540-6_3.

56 Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: Verification of machine code against authoritative ISA semantics. In *Programming Language Design and Implementation (PLDI)*. ACM, 2022. To appear.

57 Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Programming Language Design and Implementation (PLDI)*. ACM, 2013. doi:10.1145/2491956.2462183.

58 Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter Neumann. An introduction to CHERI. Technical Report UCAM-CL-TR-941, University of Cambridge, Computer Laboratory, September 2019. doi:10.48456/tr-941.

59 Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, October 2020. doi:10.48456/tr-951.