

Detailed Models of Instruction Set Architectures: From Pseudocode to Formal Semantics

Alasdair Armstrong¹ Thomas Bauereiss¹ Brian Campbell³ Shaked Flur¹
Kathryn E. Gray² Prashanth Mundkur⁵ Robert M. Norton¹ Christopher Pulte¹
Alastair Reid³ Peter Sewell¹ Ian Stark³ Mark Wassell¹

¹ University of Cambridge `firstname.lastname@cl.cam.ac.uk`

² University of Cambridge (while this work was done) `kathy_gray@gmx.com`

³ University of Edinburgh `firstname.lastname@ed.ac.uk`

⁴ ARM Ltd, Cambridge, `alastair.reid@arm.com`

⁵ SRI International, Menlo Park, `prashanth.mundkur@sri.com`

Abstract: Processor instruction set architectures (ISAs) are typically specified using a mixture of prose and pseudocode. We present ongoing work on expressing such specifications rigorously and automatically translating them to interactive theorem prover definitions, making them amenable to mechanised proof. Our ISA descriptions are written in Sail—a custom ISA specification language designed to support idioms from various processor vendor’s pseudocode, with lightweight dependent typing for bitvectors, targeting a variety of use cases including sequential and concurrent ISA semantics. From Sail we aim to portably generate usable theorem prover definitions for multiple provers, including Isabelle, HOL4, and Coq. We are focusing on the full ARMv8.3-A specification, CHERI-MIPS, and RISC-V, together with fragments of IBM POWER and x86.

1 Introduction

Instruction Set Architectures are extremely complex, with specifications in manuals containing thousands of pages. In the last decade, there has been significant progress in making ISA specifications amenable to formal reasoning, including a model of a substantial fragment of the ARM ISA, hand-written by Fox in his L3 language [2] and used for formal verification of seL4 [5] and CakeML [9], and the x86 model of Goel et al. [3].

A notable recent industry effort is ARM’s public release of its full ARM v8-A specification in machine-readable form, in their internal ASL language [8]. This vendor-provided ISA specification is attractive because it is significantly more detailed, complete, and authoritative than existing models.

To enable theorem proving using this model, ASL has to be translated to the prover of choice. We present a translation to multiple provers, currently Isabelle/HOL and HOL4, via our Sail ISA specification language [4]. Sail aims to support many different uses, including connecting ISA semantics to analysis and exploration tools for relaxed memory models [7]. In ongoing work, we have recently improved several aspects of Sail such as the type system, the generation of efficient emulator code, and the generation of portable theorem prover definitions. We are focusing on the full ARMv8.3-A specification generated from ASL, and are also using Sail for MIPS, CHERI-MIPS, RISC-V, parts of IBM POWER and x86, and a simplified ARM fragment.

2 Structure of an ISA specification in Sail

Sail aims to provide a engineer-friendly, vendor-pseudocode-like language for describing instruction

semantics. It is a straightforward imperative language with dependent typing for numeric types and bitvector lengths, checked using Z3, so that ubiquitous bitvector manipulations in ISA specifications can be checked for length and bounds errors. These lengths can be dynamically computed, as in the following example from ARMv8-A:

```
val FPZero : forall 'n, 'n in {16, 32, 64}.  
  bits(1) → bits('n)  
  
function FPZero sign = {  
  let exponent as 'e =  
    (if 'n == 16 then 5 else if 'n == 32 then 8  
     else 11) : {5, 8, 11};  
  F = 'n - exponent - 1;  
  exp = Zeros(exponent);  
  frac = Zeros(F);  
  return sign @ exp @ frac  
}
```

This returns either a 16, 32, or 64-bit floating point value, depending on the calling context. The exponent length is dynamically derived from the length of the return bitvector, and given a type variable 'e for its size. We then create bitvectors involving 'e and the return length 'n, and the type-checker can check that they all are of the required length.

A key aim of this typing information is to generate prover code that does not force the user to constantly prove side conditions involving bitvector indexing: for non-dependently-typed provers, we can use our type information to monomorphise definitions as needed.

3 ARM v8.3-A in Sail

We have a complete translation of all the 64-bit instructions in ARM’s publicly available v8.3-A specification [8]. ARM’s specification is written in their own ASL specifica-

tion language, and we have developed a tool for converting ASL specifications into Sail automatically. Hand-written specifications tend to focus on small subsets of the architecture, while the ASL-derived Sail specification includes many aspects which are often omitted, such as floating-point support, vector extensions, and system and hypervisor modes. ASL has been used extensively for testing within ARM, giving us confidence that we are accurately modelling the full behaviour allowed by the architecture. Work on validating our translation remains ongoing.

The Sail ARM v8.3-A specification is about 30 000 lines. Despite the ASL specification itself being public, much of the tooling required to easily make use of it is not. By converting it into Sail, we aim to provide open-source tooling for working with the actual v8.3-A specification. A single instruction can often call several hundred auxiliary helper functions, so reasoning about this specification in an interactive theorem prover will be challenging, and a great deal of automation will be needed.

4 Generating Theorem Prover Definitions

We generate theorem prover definitions by first translating Sail specifications to Lem [6], which then provides translations to Isabelle/HOL and HOL4. In principle, Lem also supports translation to Coq, but a direct translation from Sail is likely to produce more idiomatic Coq definitions, allowing us to preserve Sail’s dependent types for bitvector lengths. For the translation to Lem, turning these dependent types into the simpler constant-or-parameter form allowed by Lem and theorem provers such as Isabelle/HOL is one of the more intensive transformations we perform. In Lem our example becomes:

```

val FPZero : forall 'N . Size 'N =>
  integer → mword ty1 → M (mword 'N)

let FPZero (N : integer) sign =
  if (eq N 16) then
    let (F : integer) = 16 - 5 - 1 in
    let (exp : bits ty5) =
      Zeros (mk_itself 5 : itself ty5) in
    let (frac : bits ty10) =
      Zeros (mk_itself F : itself ty10) in
    return (bitvector_cast (concat
      (concat sign exp : mword ty6)
      frac) : mword 'N)
  else if (eq N 32) ...
  else fail "FPZero:_constraints_unsatisfied"

```

An extra argument *N* has been added corresponding to *'n*, and an automated dependency analysis has detected that it needs to be a concrete value, generating a case split. Constant propagation fills in concrete values for lengths. Type-level information about lengths has been passed to `Zeros` by changing integer arguments into the singleton `itself` type, giving a function compatible with Lem’s type system. While this transformation of dependent types would not be necessary for Coq, a Coq backend would share many of the other parts of the translation pipeline with Lem, such as the translation of imperative code into monadic expressions.

In addition to targeting different provers, we aim to support different use cases. For reasoning in a purely sequential setting, a state monad can be used. In a concurrent setting, we need to be more fine-grained. Modern processors typically execute many instructions simultaneously, re-ordering their memory and register accesses for increased performance. We support this by using a free monad of an effect datatype. A monadic expression evaluates either to a pure value or to an effect and a continuation (or an exception without continuation). This gives us the fine-grained effect information needed to reason about multiple instructions concurrently; the monad is suitable as an interface to connect the ISA semantics with a relaxed memory model. We recover a purely sequential model using a lifting to the state monad. Isabelle automation for this lifting is provided by simplification rules relating the primitive operations of the monads, allowing us to seamlessly reason about the sequential behaviour of instructions, e.g. using a Hoare logic.

5 Conclusion

We plan to continue improving both Sail, e.g. by adding a Coq backend, and the ISA models. The tool and models are available online [1] under an open-source license. We plan to put the models to actual use in theorem provers, and invite other projects to consider using them as well.

Acknowledgements This work was partially supported by EP-SRC grant EP/K008528/1 (REMS), an ARM iCASE award, EP-SRC IAA KTF funding, and the CIFV project supported by the United States Air Force under contract FA8750-18-C-7809.

References

- [1] The Sail ISA semantics specification language, 2018. <http://www.cl.cam.ac.uk/~pes20/sail/>.
- [2] A. C. J. Fox. Directions in ISA specification. In *ITP*, 2012.
- [3] S. Goel. The x86isa books: Features, usage, and future plans. In *Proc. 14th ACL2 Workshop*, 2017.
- [4] K. E. Gray, G. Kerneis, D. P. Mulligan, C. Pulte, S. Sarkar, and P. Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *MICRO*, pages 635–646, Dec. 2015.
- [5] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, 2014.
- [6] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: Reusable engineering of real-world semantics. In *ICFP*, pages 175–188. ACM, Sept. 2014.
- [7] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In *POPL*, Jan. 2018.
- [8] A. Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *FMCAD 2016*, pages 161–168, October 2016.
- [9] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish. A new verified compiler backend for CakeML. In *ICFP*, pages 60–73, 2016.