Improved Tool Support for Machine-Code Decompilation in HOL4

Anthony Fox

University of Cambridge, UK

Abstract. The HOL4 interactive theorem prover provides a sound logical environment for reasoning about machine-code programs. The rigour of HOL's LCF-style kernel naturally guarantees very high levels of assurance, but it does present challenges when it comes implementing efficient proof tools. This paper presents improvements that have been made to our methodology for soundly *decompiling* machine-code programs to functions expressed in HOL logic. These advancements have been facilitated by the development of a domain specific language, called L3, for the specification of Instruction Set Architectures (ISAs). As a result of these improvements, decompilation is faster (on average by one to two orders of magnitude), the instruction set specifications are easier to write, and the proof tools are easier to maintain.

Traditional formal software verification has primarily focussed on developing and using formalisations of high-level programming languages, with formal reasoning occurring at the level of the programmer. However, in some high-assurance applications, such language formalisations could be too abstract or unrealistic, and the trustworthiness of compilers may come into play. These issues can be addressed by using a verified compiler, see [9] and [8]. However, an alternative approach is to work directly with machine-code, which could be generated by any compiler for a particular platform. This has the advantage that one does not have to formalise the semantics of high-level source languages, and formal reasoning relates more directly to the code that is actually being run. The success of this approach hinges upon the ability to accurately formalise a processor's instruction set and on the ability to overcome the challenges of working with low-level code, which is less structured and replete with platform specific details. To this end, Magnus Myreen has developed an approach for soundly *decompiling* machine-code using the HOL4 interactive theorem prover, see [12].

Commercial instruction set architectures are large and complex, with reference manuals running to thousands of pages in length.[1] We use the L3 domain specific language to formally specify ISAs, see [4]. Details of our current ISA formalisations can be found in Sections 2 and 8. Each architecture has its own idiosyncrasies, which must be accommodated when writing proof automation for a theorem prover. In this paper our main working instruction set is ARMv7-A

[1] For example, 2736 pages for ARMv7-A, 5242 pages for ARM-v8 (which contains a full description of the legacy AArch32 mode) and 3020 pages for x86.

but we also support other architectures.[2] In particular, we have recently added support for the new 64-bit ARMv8 architecture.

Papers [12] and [13] of Myreen et al. present proof tools for decompiling machine-code programs into logic using the HOL4 interactive theorem prover. The second paper (from 2012) presented a new version of this proof-producing decompiler, which provided significant improvements in the speed of decompilation. Benchmark figures showed that the overall run time had become dominated by model evaluation, i.e. the time taken to generate Hoare triples that capture the semantics of each individual machine-code instruction. These Hoare triples take the form of *spec* theorems, which provide an interface between ISA models and the decompiler, see Section 1. This paper presents substantial improvements in our treatment of ISA specification and model evaluation. The techniques presented here supersede those of [5]. For comparison purposes, updated benchmark figures are provided in Section 7 for the examples listed in [13]. The HOL4 tools for generating *spec* theorems have been enhanced to be roughly one hundred times faster than previous versions (those presented in [5]). This speedup comes from utilising dynamic databases of generic *spec* theorems, which capture the semantics of multiple instruction instances (for multiple operating modes). These theorems are derived by partially evaluating ISA models, see Sections 3 and 4.

The improvements described in this paper have made it more tractable to work with larger machine-code programs. The largest decompilation undertaken to date has been for the seL4 microkernel (see [15]), where the code size is roughly 12,000 ARM instructions. Our models have also been used by other organisations and research groups, including at the KTH Royal Institute of Technology, see [2].

The approach described here has matured to the point where specifying a new ISA, and linking the generated HOL model to the decompiler, is mostly routine. Experience indicates that the effort of supporting a new ISA is split almost evenly between model development (in L3) and implementing tool support (in HOL4). For simple RISC architectures, such as MIPS, preliminary support (without model validation) has been provided within a few weeks. Although our tools have been implemented in HOL4, the overall approach is applicable to other LCF-style provers, such as HOL Light, Isabelle/HOL and ProofPower.

1 Decompilation of Machine-code to HOL Logic

In [12] Myreen describes methods for soundly decompiling machine-code into HOL functions. The decompiler outputs a collection of definitions, as well as a *certificate* theorem, which proves that the definitions correctly capture the semantics of the supplied machine-code. For example, the ARM code

```
e0010291   (*  loop: mul  r1, r1, r2   *)
e2500001   (*        subs r0, r0, #1   *)
1affffc    (*        bne  loop         *)
```

[2] L3 specifications are available at `www.cl.cam.ac.uk/~acjf3/l3/isa-models.tar.bz2` and the HOL4 developments can be viewed at the Github repository `github.com/HOL-Theorem-Prover/HOL` under the directory `examples/l3-machine-code`.

is decompiled into the following HOL function:[3]

```
power (r0,r1,r2) =
   let r1 = r1 * r2 in
   let r0' = r0 - 1w
   in
     if r0 = 1w then (r0',r1,r2) else power (r0',r1,r2)
```

Certificate theorems are instances of a total correctness Hoare triple assertion SPEC *model* p *code* q, see [12]. The certificate theorem for our example is

```
⊢ SPEC ARM_MODEL
   (~aS * arm_OK m * arm_PC p * arm_REG (R_mode m 0w) r0 *
    arm_REG (R_mode m 1w) r1 * arm_REG (R_mode m 2w) r2 *
    cond (power_pre (r0,r1,r2)))
   {(p,0xE0010291w); (p + 4w,0xE2500001w); (p + 8w,0x1AFFFFFCw)}
   (let (r0,r1,r2) = power (r0,r1,r2)
    in
      ~aS * arm_OK m * arm_PC (p + 12w) * arm_REG (R_mode m 0w) r0 *
      arm_REG (R_mode m 1w) r1 * arm_REG (R_mode m 2w) r2)
```

Here `ARM_MODEL` is a 5-tuple that incorporates an ARM semantics relation. A set (the *code* pool) associates memory addresses with machine-code values. The pre- and post-conditions (p and q) are split into assertions, combined using Myreen's machine-code logic *separating conjunction operator* (`*`), see [12]. For example,

```
arm_OK m * arm_PC (p + 12w) * arm_REG (R_mode m 0w) r0
```

asserts that the program-counter has value `p + 12w` and the zeroth general-purpose register (`R_mode m 0w`) has value `r0`. The processor mode is constrained to be valid with the assertion `arm_OK m`, e.g. `m = 16w` (user mode) is valid.

The decompiler works by deriving and composing SPEC theorems for single machine-code instructions. The semantics of each instruction is determined by symbolically evaluating the next-state function of an ISA. This is implemented in HOL with *step* and *spec* tools, which are described in Sections 3 and 4. This paper focusses on improvements made to the ISA models and associated model evaluation tools. A list of these improvements is contained in Table 1.

The implementation of ISA specific *step* and *spec* tools – which effectively link ISA models to the decompiler – is aided through the use of special purpose HOL libraries. The primitive assertion predicates (such as `arm_REG`) and associated lemmas are now automatically generated by a tool which examines the state type of the ISA. This automation makes it much easier to accommodate changes to specifications and to support new ISAs. However, the process of developing ISA specific tools is not fully automatic. An understanding of each ISA is required, and there are places where cognisant choices are made, i.e. which parts of the ISA model are pertinent. Naïve symbolic evaluation of sizeable ISA formalisations typically leads to the generation of unwieldy terms (perhaps taking minutes to compute in HOL). Also, it is frequently necessary to manually prove simplification rewrites, as steadfastly following ISA reference manuals can result in unwanted expressions arising within assertions and decompiler output.

[3] This computes `r1 := r1 * pow(r2, r0)`. Note that `1w` denotes a machine word (bit-vector) with value one.

Table 1. Comparison of the old and new approaches.

	Old	New
Specification and Formal Model	Native HOL using a state-transformer monad with exceptions.	Specified in an imperative style using the L3 domain specific language. The exported HOL models employ let-expressions and have an 'exception status' state component. [See Section 2]
Step Tool	Requires a full machine-code value (opcode). Evaluates the instruction set model directly. The ARM *step* tool employs a call-by-value based conversion.	Mainly based on machine-code bit-patterns (named instructions). Uses a database of pre-computed 'instruction semantics' theorems, with the *step* theorem being derived using HOL's `MATCH_MP` rule. [See Section 3]
Spec Tool	State assertions and associated theorems are all defined and proved manually. The theorems are based on concrete opcode values. The derivation tactic is hard-coded for each ISA.	Assertions are automatically defined and various theorems are automatically derived. Supports *generic* Hoare triples, based on instruction bit-patterns. The derivation tactic is customisable, so as to support multiple ISAs. [See Section 4]
Improved Decompiler Support [13]	The Hoare *triple* theorems are derived direct from *step* theorems. This derivation is hard-coded for each ISA.	The *triple* theorem derivation uses *spec* theorems. This method is generic (readily customisable), so it is easy to support new ISAs. [See Section 5]
Assembly Code Support	Written mostly in Standard ML, using a parser generator or parser combinators.	Mostly written in L3, with a small amount of Standard ML. [See Section 6]
ISAs	ARMv4 through to ARMv7, PowerPC, x86-32, x86-64.	ARMv4 through to ARMv7, ARMv6-M, ARMv8, x86-64, MIPS, CHERI. [See Section 8]

2 ISA Specification using L3

The domain specific language L3 [4] provides an effective means to specify and maintain a diverse selection of instruction set models. The language has been designed to be simple and intuitive, with specifications being compact and easy to comprehend. To date, L3 models have been exported to HOL4, Standard ML, Isabelle/HOL and TSL [10].

The primary component of a formal ISA specification is a *next-state* function, which defines the architecture's operational semantics at the *programmer's model* level. One machine-code instruction is run for every application of the next-state function. A HOL4 formalisation of the ARMv7 architecture was presented in [5]. This model was based on defining a state-transformer monad (with exceptions) directly in HOL. Our new L3 source specifications now look much more like the pseudo-code found in architecture reference manuals. The exported HOL4 version is treated as a reference formal specification (trusted model), whereas exported SML code is used to implement emulators and assemblers. There is no formal or verified connection between each of these models. We primarily carry out model validation with respect to the trusted HOL model, typically using *step* tools to determine the behaviour of machine-code instructions (see Section 3). Additional validation and development work may be performed using the much faster ML-based emulators. In particular, our MIPS model is now capable of booting FreeBSD, see Section 8. This emulation work has also demonstrated that L3 models are well suited to rapidly prototyping new architectural designs.

Two styles of HOL specification can be exported from L3: one uses a generic state-transformer monad; and the other uses let-expressions, which directly manipulate (pass through) the state of the architecture. The monadic style is suited to security oriented proofs, such as [2]. This paper is primarily concerned with the let-expression style, since this is the version that is used by the decompiler. The following two sections describe this new style of specification.

Model Exceptions Model exceptions[4] are normally used to handle various instances of *unpredictable* (under-specified) behaviour.[5] With the use of L3, our treatment of model exceptions has changed so as to make it easier to write efficient automated proof tools.

In our old monadic approach, the type of each (impure or state transforming) operation is roughly:

$$[args \rightarrow] state \rightarrow (value \times state) + exception$$

where + denotes the disjoint union type operator and square parenthesis are used to indicate an optional type component. Such operations either return a

[4] This should not be confused with ISA exceptions (e.g. software interrupts), which are typically modelled explicitly, following the ISA reference manual semantics.

[5] The ARM manual describes the *unpredictable* as meaning "the behaviour cannot be relied upon". For example, the shift instruction `ASR r1,r2,pc` is unpredictable.

value together with an updated state, or they return an exception value. A monadic *bind* operation (`>>=`) is used to compose sequential computations. Note that performing symbolic evaluation in the context of `>>=` frequently leads to the generation of unwieldy terms, since the bind definition includes a case split over the *value-state* (exception free) and *exception* cases.

The following operation type is used in L3 generated models:

$$[args \rightarrow]state' \rightarrow value \times state'$$

where $state' = state[\times exception\text{-}status]$. The range now consists of a value paired with a state, which may incorporate an exception status component. The *value* and *state* parts of the result should be regarded as meaningless when an exception occurs, which is flagged using the exception status component of $state'$. Although this representation is less principled – since junk values and states are returned when an exception has occurred – it does have the advantage that we can express operational semantics using a standard state-transformer monad or using ordinary let-expressions. This makes the generated models easier to work with.

Example Specification Consider the following pedagogical L3 specification:

```
exception UNPREDICTABLE {- declare a new exception type -}

declare { A :: bits(8), B :: bits(8) } {- global variables -}

bits(8) example (c :: bits(8)) =
{
   A <- B;                        {- assignment -}
   when A < 4 do #UNPREDICTABLE; {- raise an exception -}
   return (A * c + 1)
}
```

This L3 specification declares a new model exception `UNPREDICTABLE`, two global state components `A` and `B`, and a unary function `example`. The generated HOL script for this specification defines the following function:

```
example c =
(λstate.
   (let s = state with A := state.B in
    let s = if s.A < 4w then SND (raise'exception UNPREDICTABLE s)
            else s
    in
      (s.A * c + 1w,s)))
```

Note that the state is explicitly modified using let-expressions. A record type is used, with state components `A` and `B` being sub-fields of the global state. The helper function `raise'exception` is used to tag the state when an exception occurs (it ensures earlier exceptions are not overridden).

3 Model Evaluation: *Step* Theorems

A *step* tool uses forward proof to derive a theorem that characterises the next-state behaviour of a particular instruction. In [5], *step* theorems are of the form:

$$\vdash \forall s.\ P(s) \Rightarrow (\text{next}(s) = \text{SOME}\ (f_0(f_1(\ldots f_n(s)))))$$

where s is a state and predicate P consists of various conjuncts, usually including assertions of the from $mem(pc + i) = b_i$ for $i \in 0, \ldots, 3$. Each byte b_i forms part of the machine-code opcode of the instruction being run, which is located in memory *mem* at the program-counter address *pc*. Other clauses determine the operating mode, e.g. big- or little-endian byte ordering and so forth. Additional clauses are also required to avoid unpredictable or undesired behaviour; for example, we assert that the program-counter address is word aligned (divisible by four). The functions f_i denote state updates for particular components; for example, one function might write a value to a register. The next-state function returns an *option* type, which is a standard HOL datatype.[6]

Generic *step* theorems. With the move to L3 specifications, we now use *step* theorems of the form:

$$P_0, \ldots, P_n \vdash \text{next}(s) = \text{SOME}\ (s\ \text{with}\ \ldots)\ .$$

The hypotheses P_i correspond with clauses of the old *spec* theorem predicate P. The 'with' syntax denotes updates to a record, e.g. the next-state might be

$$s\ \text{with}\ pc := s.pc + 4\ .$$

Here the program-counter is updated so as to point to the next instruction in memory. A key improvement is that we now generate *step* theorems that are not restricted to concrete (ground) machine-code opcodes. Instead, each byte of the fetched instruction can be represented by a list of Booleans, which form part of the instruction's opcode pattern. For example, the four bytes for the 'Multiply Accumulate' instruction `MLA` (with little-endian ordering) are

```
i=0  v2w [T; F; F; T; x13; x14; x15; x16]
i=1  v2w [x5; x6; x7; x8; x9; x10; x11; x12]
i=2  v2w [F; F; T; F; x1; x2; x3; x4]
i=3  v2w [T; T; T; F; F; F; F; F]
```

Here `T` represents true, `F` is false and the HOL function `v2w` : `bool list` → `word8` constructs a bit-vector from a list. Variables are used to encode register and immediate argument values, as well as various instruction configuration options. A particular instance of this ARM instruction is `MLA r1, r2, r3, r4`, which has opcode `0xE0214392`. This instance corresponds with the substitution:

$$x_1, x_2, x_3, x_4 \mapsto \texttt{F, F, F, T},\quad (\texttt{r1}) \qquad x_5, x_6, x_7, x_8 \mapsto \texttt{F, T, F, F},\quad (\texttt{r4})$$
$$x_9, x_{10}, x_{11}, x_{12} \mapsto \texttt{F, F, T, T},\quad (\texttt{r3}) \qquad x_{13}, x_{14}, x_{15}, x_{16} \mapsto \texttt{F, F, T, F}\quad (\texttt{r2})\ .$$

[6] A state option value is either NONE (when there is an exception) or otherwise it is SOME s for some state s. We are only interested in exception free cases here.

This substitution effectively specialises the instruction pattern for our instruction instance (choice of registers), i.e. `r2 * r3 + r4` is written to register `r1`.

By developing tools that generate generic *step* theorems (which represent partial evaluations of a next-state function with respect to opcode bit-patterns), we are then able to derive generic *spec* theorems, i.e. Hoare triples for a class of instructions. This generalisation provides a means to greatly speed-up the model evaluation phase of decompilation. Rather than generate Hoare triple theorems from scratch for every machine-code opcode (which is expensive), we can instead dynamically build up a database of generic triples that can be quickly specialised when needed.

Derivation. Generic *step* theorems are derived using HOL's `MATCH_MP` rule.[7] For ARM, we first derive antecedent theorems (roughly) of the form:

$$F_0, \ldots, F_f \vdash \texttt{Fetch}(s) = (v, s_0) \tag{1}$$

$$D_0, \ldots, D_d \vdash \texttt{DecodeARM}(v, s_0) = (ast, s_1) \tag{2}$$

$$\vdash \forall s.\ \texttt{Run}(ast)(s) = defn(x)(s) \tag{3}$$

$$I_0, \ldots, I_i \vdash defn(x)(s_1) = s \text{ with } \ldots \tag{4}$$

which together imply the following *step* theorem

$$F_0, \ldots, F_f, D_0, \ldots, D_d, I_0, \ldots, I_i \vdash \texttt{NextARM}(s) = \text{SOME } (s \text{ with } \ldots)\ . \tag{5}$$

In the above, v represents a machine-code bit-pattern; *ast* is an instruction datatype value; x is an instruction's arguments (e.g. register indices and immediate values); and *defn* represents an instruction semantics function. The various hypotheses relate to different stages of execution, e.g. the F hypotheses assert that the bytes of the opcode v are located in main memory, starting at the program-counter address.

The functions `Fetch`, `DecodeARM`, `Run` and `NextARM` all come from the current L3 specification of ARM. Eqs. (1) to (4) deconstruct the next-state function `NextARM`, and the implication above is proved in HOL with a one-off theorem. Similar theorems are proved for each ISA that we support. It is hard to fully automate the process of decomposing next-state specifications, so as to construct and verify suitable `MATCH_MP` theorems. These theorems are very architecture specific, for example, the MIPS theorems have to take the branch-delay slot into consideration and the x86 model has to accommodate variable width instruction opcodes and an instruction cache.

In deriving a *step* theorem for a particular instruction class (bit-pattern), various parts of Eqs. (1) to (4) are specialised, prior to applying the `MATCH_MP` rule. For example, for a particular instruction instance the semantics function *defn* might be `dfn'StoreByte` or `dfn'MultiplyLong` and Eq. (4) will give the

[7] This is the Modus Ponens inference rule with automatic matching. For example, given a theorem $A_0 \vdash t_1 \wedge \cdots \wedge t_n \Rightarrow t_0$ and a list of theorems $A_1 \vdash t_1, \ldots, A_n \vdash t_n$, we can use this rule to derive $A_0, \ldots, A_n \vdash t_0$.

next-state semantics for that type of instruction. Eqs. (1) and (3) can be readily derived on the fly for any particular opcode bit-pattern. Eqs. (2) and (4) are precomputed for a fixed set of *supported* instruction bit-patterns. These theorems are stored in databases that are based on Michael Norrish's `LVTermNet` structure, which implements *local variable term nets*.[8] Using this method, the resulting *step* tool is extremely efficient. The four antecedent theorems can be deduced very quickly (since database lookup does not require any *additional* logical inference) and an application of `MATCH_MP` rule is fast as well.

The practicability of this new approach hinges upon the ability to precompute all of the required instruction semantics theorems (instances of Eq. (4)) in an "acceptable" amount of time. The ARM model is complex and, at the time of writing, there are 318 of these theorems. They are produced by expanding definitions to a canonical form that consists of primitive state (record field) updates. This symbolic evaluation involves: considering instruction sub-cases; eliminating let-expressions, avoiding unpredictable cases (by adding new conditions to the set of hypotheses); and applying simplifications. A custom tool has been developed to aid this process. Where appropriate, simplification rules are manually identified and verified, e.g. they may involve reasoning about machine arithmetic and bit-vector manipulations. Here, HOL4's bit-blasting procedure for deciding bit-vector problems is of great use, see [3]. The HOL library `arm_stepLib` implements the ARM *step* tool; it consists of 4014 lines of hand-written code and takes just under two minutes to build. This library uses theorems from `arm_stepTheory`, which is built using 1498 lines of hand-written proof script.

4 Model Evaluation: Machine-Code Hoare Triples

A *spec* tool derives *spec* theorems (Hoare triples) for machine-code instructions. The performance of these tools has been greatly enhanced through the use of generic *spec* theorems, which can be instantiated to obtain triples for concrete opcodes (where instruction arguments become fixed). This new approach is illustrated in Fig. 1. A feature of this algorithm is that it incorporates a form of memoization. The performance of the tool improves with use, since the costly "no" branch in Fig. 1 only occurs when new instruction types are encountered.

Multiple generic *spec* theorems (up to sixteen for ARM) are derived for each generic *step* theorem. These theorems cover various instruction forms, e.g. `MOV Rm, Rn` (with `Rm` $\neq$ `Rn`) and `MOV Rm, Rm` are distinct forms. The pre- and post-conditions are determined by syntactically examining the supplied *step* theorem. The *spec* theorem derivation is based on using a carefully crafted tactic; see `HOL/examples/l3-machine-code/common/stateScript.sml` for an overview of the approach and for proofs of key lemmas.

The 'reject vacuous' stage in Fig. 1 is used to select the appropriate specialised *spec* theorems. In practice, we also apply post-processing stages, which support different state assertions, e.g. viewing registers and/or memory as maps.

[8] These are a form of *discrimination net*. A similar structure `Net` (credited to Larry Paulson) has been used for many years in HOL's term-rewriting conversions.

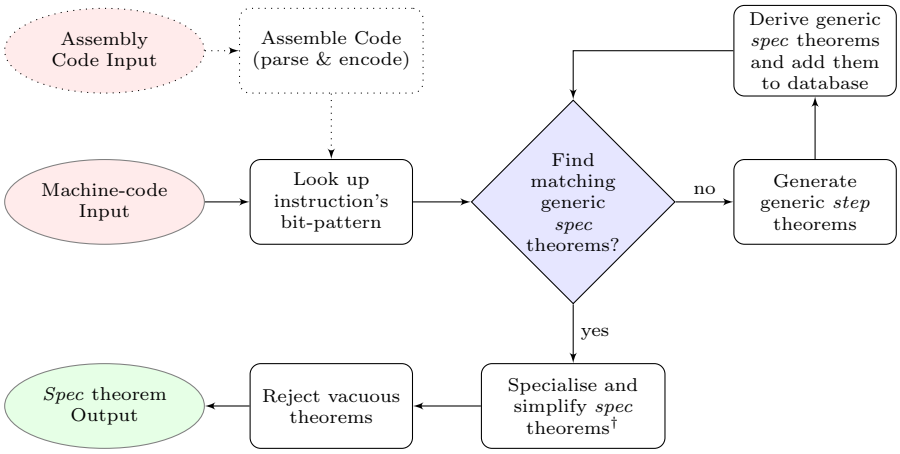


Fig. 1. Generation of *spec* theorems. † Note that theorem specialisation is fast and the simplification phase has been fine-tuned for performance.

To illustrate the improvements in performance, consider the 'Store Register Dual' instruction `STRD r0, r1, [r2, r3]!`, which has opcode `0xE1A200F3`. The timings for this instruction (on a 3.4 GHz Core i7, 32 GB) are as follows:

Old *step* tool:	0.35 s
Old *spec* tool:	2.54 s
New *step* tool (`"STRD (+reg,pre,wb)"` instruction class):	0.0017 s
New *spec* tool (first call):	0.91 s
New *spec* tool (subsequent calls within class):	0.0027 s

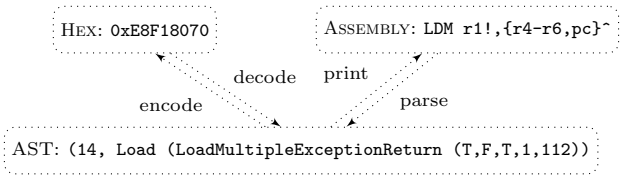
5 Supporting the Improved Decompiler

The improved decompiler [13] uses a new Hoare triple predicate, wherein the pre- and post-condition assertions are more hardwired, i.e. fixed for a particular (manually determined) processor configuration and state 'view'. The uniformity of these *triple* theorems helps in speeding up the decompilation process. A tool has been developed to derive *triple* theorems from *spec* theorems. There is a small overhead (typically a few thousandths of a second) for this conversion. As such, it is relatively easy to support both versions of the decompiler.

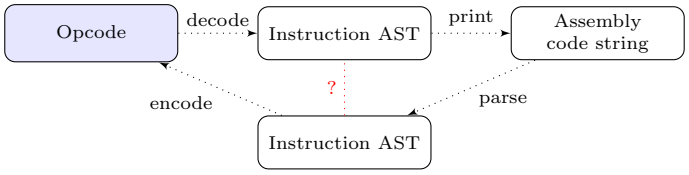
6 Assembly Code Support

When working with an instruction set model, it is helpful to provide support for some standard assembly code representation of instructions (which humans can more readily comprehend). In particular, it is useful to be able to input assembly

code programs (or single instructions) and then output machine-code opcodes. We achieve this by implementing light-weight assemblers, which consist of two parts: a *parser*, which maps assembly code syntax (strings) into an instruction datatype (AST); and an *encoder*, which maps instruction datatype values into machine-code opcodes. We also define *pretty-printers*, which map instruction datatype values back into assembly code syntax (strings). An example of these mappings is shown below for an ARM load-multiple instruction:



Round-trip validation is used to test the consistency of the decode, encode, parse and print specifications; this is illustrated below.



A successful round-trip occurs when the opcode produced by the encoder is the same as an initial opcode. Note that the original opcode may not be 'canonical' (e.g. ARM immediate values do not have unique encodings) — in such cases the round-trip will fail and a check is made on whether the abstract syntax for the two instructions (from decoding and parsing) are equivalent. This approach has been highly effective in terms of detecting inconsistencies and bugs in instruction representation logic. The parsing and encoding logic has also been checked laboriously with test vectors, e.g. to ensure that syntax and bounds errors are handled correctly.

We have found little need for assembly code parsers and pretty-printers to be formalised within a theorem prover, since we normally work directly with machine-code opcodes when considering the semantics of low-level programs. As such, these components are implemented at the meta-level using Standard ML. There are some use cases for formalised instruction encoders, as these can be used in compiler backends; for example, in the CakeML project [8].

Previously, encoders, parsers and printers have been written in Standard ML (HOL's meta-language), see [5]. In particular, parsing has been implemented using parser generators and later with parsing combinators. We now specify encoding, parsing and printing using L3, which helps in keeping these components consistent and synchronised with the core model. L3 is well-suited to specifying instruction encoders, since the language provides excellent support for working with bit-vectors. The complete specification is exported to ML and this is then used to write simple assemblers, using a relatively small piece of hand coded ML. The parser and printer specifications are not exported to HOL.

7 Performance

In presenting a faster decompilation algorithm, Myreen et al. provided some benchmarks figures, see Table 2. Following the changes presented in this paper, the latest performance figures are shown in Table 3. Column one shows that the performance of the decompiler itself, which excludes the model evaluation phase, has improved since 2012.[9] This has been achieved by implementing simple coding improvements within the decompiler and associated library code. The underlying algorithms, as presented by Myreen et el., have not been modified. Of most interest is the model evaluation figures (column group two). Three timings are presented: the first column contains updated figures for the old model and tools; the second corresponds with generating *spec* theorems from a 'cold' state (where no instructions have been encountered before) using the new L3 model and tools; and the third is from a 'warm' state, where all of the instructions have been encountered before.[10] It is clear that significant performance improvements have been achieved. The old tools (see [5]) have again improved due to tweaks within library code. For the new tools, the performance is at worst just a bit slower than before (i.e. in the 'sum of array' example), however this corresponds with an outlying case where just four instruction are processed from a cold state. From a 'warm' state, model evaluation is now faster than the main decompilation phase itself. As such, this new technology has successfully overcome a bottleneck in scalability. The faster and more general *step* tools also help in areas such as model validation and compiler verification.

Table 2. Performance figures found in [13]. Examples: (1) sum of array; (2) copying garbage collector; (3) 1024-bit multiword addition; and (4) 256-bit Skein hash function.

example	code size	decompilation time (and inferences)		model evaluation
	(instructions)	original version	2012 version	time (and inferences)
1	4	2.5 s (73,039 i)	0.3 s (4019 i)	7.8 s (1.5 Mi)
2	89	50 s (1,526,281 i)	6.0 s (53,301 i)	173 s (40 Mi)
3	224	70 s (1,029,685 i)	1.2 s (10,802 i)	37 s (8.9 Mi)
4	1352	5366 s (21,432,926 i)	56 s (1,842,642 i)	500 s (105 Mi)

Table 3. Latest performance figures for the same ARM examples.

decompilation time (and inferences)		model evaluation time (and inferences)		
original version	2012 version	old	cold	warm
1.47 s (78,688 i)	0.12 s (16,481 i)	3.2 s (0.74 Mi)	9.8 s (3.9 Mi)	0.02 s (15,521 i)
32.5 s (1,598,271 i)	2.0 s (349,740 i)	51.3 s (16.2 Mi)	19.8 s (13.5 Mi)	0.35 s (273,725 i)
50.0 s (1,085,104 i)	0.3 s (45,949 i)	20.3 s (3.3 Mi)	9.9 s (9 Mi)	0.03 s (8435 i)
11,786 s (19,921,648 i)	8.0 s (4,756,617 i)	350 s (44.6 Mi)	23.7 s (28 Mi)	4.0 s (2.8 Mi)

[9] The Skein example under the 'original' decompiler is an anomaly here.

[10] Caching on hexadecimal opcode values has not been enabled, so these run times correspond with specialising generic *spec* theorems.

Table 4. L3 instruction set models. The 'Lines of L3' figure is split into *core model* and *additional logic* (for instruction encoding and assembly code support).

ISA	Operating Modes	System Levels	Instruction Width	General-Purpose Registers	Flags	Endianness	Coverage	Lines of L3
ARMv4 to ARMv7	ARM, Thumb	User, System, (Hypervisor), Abort, Undefined, (Monitor), IRQ, FIQ	32-bit,16-bit	32-bit 16† (banked)	N, Z, C, V, Q, GE	Big, Little	Partial VFP No Adv. SIMD No CP	9238+7687
ARMv6-M	Thumb	Main, Process	16-bit	32-bit 16† (SP banked)	N, Z, C, V	Big, Little	No CP	1996+2095
ARMv8	AArch64 only	EL0, EL1, EL2, EL3	32-bit	64-bit 32 (reg. 31 is 0)	N, Z, C, V	Big, Little	No VFP No Adv. SIMD Partial System	2434+4097
x86-64	64-bit only	-	Variable (bytes)	64-bit, 16	CF, PF, AF, ZF, SF, OF	Little	40 basic instructions	1357+1579
MIPS (RS4000)	-	User, Supervisor, Kernel	32-bit	64-bit 32 (reg. 0 is 0)	-	Big, Little	Partial CP Partial System	2080+700
CHERI	-	User, Supervisor, Kernel	32-bit	64-bit 32 (reg. 0 is 0)	-	Big	Partial CP	5299

† The program-counter is a general-purpose register.

8 Instructions Sets

A summary of our ISA formalisations in L3 is shown in Table 4. The following sections give a brief overview of these architectures.

ARMv4 through to ARMv7. Although nominally a RISC architecture, ARM is challenging from a specification and verification perspective. Areas of complexity are: the number of system levels and the use of banked general-purpose registers; the program-counter is a general-purpose register,[11] which leads to *unpredictable* behaviour (see Section 2); and the `LDM` and `STM` block data transfer instructions are remarkably elaborate. As part of previous work, described in [5], a fairly large set of single instruction test vectors were developed for the purposes of ARMv7 validation. These tests have helped identify a handful of bugs in the new L3 specification, which were all trivial to fix. The new model is now considered to be as trustworthy as the previous version, which was specified directly in HOL. We have no plans to formally verify a correspondence between the two models.

A notable change to the new specification is with regard to the specification of *unpredictable* and *undefined*[12] instruction instances. This logic has moved from instruction semantics functions to decoders. As such, the decoders can be used to check the validity of instruction encodings, which is useful when writing an assembler, see Section 6. The instruction semantics functions have also become easier to work with.

[11] When an instruction explicitly reads the PC this *normally* gives the value of that instruction's address plus eight (in ARM mode) or plus four (in Thumb mode). This is because early ARM processors employed a 3-stage pipeline.

[12] An instruction opcode is *undefined* when it is not supported by an architecture version or configuration. For example, `CLZ` opcodes on ARMv4 will raise an undefined exception trap.

ARMv6-M. This architecture is implemented by the Cortex-M0 micro-controller and our model includes processor timing information (a cycle counter). Extensive model validation has been carried out by Brian Campbell [1].

ARMv8. This is a completely new 64-bit architecture. Although compatibility with ARMv7 is provided with an AArch32 operating mode, our L3 formalisation omits this functionality and just supports the new AArch64 mode. The instruction set is completely new; as is the underlying programmer's model. Our formalisation is currently awaiting validation. Due to some rationalisations in the ARMv8 architecture (in AArch64 mode), this ISA is actually cleaner and easier to work with when compared to ARMv7. There are still some complexities, e.g. the various encoding schemes for immediate values are somewhat elaborate.

x86-64. Being an older CISC architecture, the x86 family is renowned for its size and complexity. However, we only provide a comparatively simple model of x86-64 in L3, which covers just 64-bit operating mode for a core set of about forty instructions (providing adequate coverage for case studies). This was ported from a native HOL4 specifications. Some limited model validation with respect to hardware has been carried out with the assistance of Magnus Myreen.

MIPS64 and CHERI. MIPS64 is a relatively clean RISC architecture. The CHERI research architecture extends MIPS with capabilities for implementing security management, see [17]. One source of complexity for MIPS is the presence of a branch-delay slot, which affects the semantics of jump instructions. The CHERI model permits multi-core operation and also adds support for: interrupts; UART I/O; a translation lookaside buffer (TLB); and more coprocessor instructions. This advanced, high-fidelity model is primarily used for emulation, validation and rapid prototyping. The model is mature enough that it can boot an unmodified development version of FreeBSD (which has a 7.2 MB image size) in about ten minutes on a modern machine. Booting the OS in multi-core mode is supported as well. Alexandre Joannou, Matthew Taylor and Mike Roe have worked on the extended MIPS and CHERI models, and this development can be found on Github at `github.com/acjf3/l3mips`.

9 Related Work

Other recent work on reasoning about machine-code programs has mainly been undertaken using the ACL2 and Coq theorem provers. Most of this work has focussed on the x86 architecture. Related work is carried out in the field of binary-analysis (using flow-based approaches), where instruction set models are developed and used in less formal settings, i.e. where machine-code programs are not formally verified against specifications.

Warren Hunt's group have developed a high quality model of x86-64 using ACL2, see [6]. Shilpi et al. report that their ACL2 model covers 121 user-level instructions (much more than our L3 specification) and they note that there is work in progress (headed by J S. Moore) on an ACL2 based automated tool,

called Code Walker, that is comparable with Myreen's decompiler. By design, the ACL2 programming language naturally supports fast model evaluation. Their x86 model can be tested in an *execution mode* and proofs can be constructed in a *logical mode*. With the former evaluation mode, they achieve a performance of nearly a million instructions per second, with a two-level page table enabled. By contrast, HOL4 is an LCF-style theorem prover that is not designed for high performance model evaluation. As such, when carrying out emulation work we generate Standard ML versions of our models. When reasoning in the HOL logic, the techniques presented in this paper provide sufficient performance for formal verification work. We consider the main advantages of our approach to be that our ISA models are compact and accessible (through the use of a domain specific language); and also that our infrastructure for supporting automated decompilation to logic (for multiple ISAs) is now relatively mature.

As part of the Rocksalt project (for a software-based fault isolation checker), Morrisett et al. have developed an x86 model in Coq, see [11]. Other x86 models have been developed in Coq as part of research into devising logical frameworks for reasoning about low-level code, see [14] and [7]. In addition, simple assembly-level Coq models of x86, PowerPC and ARM have also been used within the CompCert verified compiler, see [9].

Within the area of binary-analysis, the work of Reps et al. is of note, see [10]. They use a domain specific language TSL to specify ISAs, including PowerPC and x86. Recently, they have customised L3, so as to generate TSL code from our ARMv7 model. These TSL specifications are used to generate a range of binary-analysis tools. Related work includes the GDSL toolkit of Simon et al., see [16]. They have a used a domain specific language to specify (fast) decoders, as well as semantics translators, e.g. for x86 and Atmel AVR. At present it is unclear how easy it would be to adapt their work for the purposes of formal verification using an interactive theorem prover.

10 Summary

This paper describes the current status of our models, tools and methodology for the formal verification of machine-code programs. Our approach is based on using three programming environments: L3 for developing ISA specifications; Standard ML (compiled using Poly/ML or MLton) for efficient emulation; and HOL4 for formal reasoning. L3 has eased the task of developing instruction set specifications, and it has also helped ensure that generated HOL models are of a known and manageable form. Improved techniques for model evaluation are presented and proof tools have been implemented. The performance of machine-code decompilation has been greatly enhanced, see Section 7. These gains have been achieved by maintaining various databases of pre-proved theorems, see Sections 3 and 4. In particular, the *spec* tool now maintains a database of generic *spec* theorems. These methods are applicable to other LCF-style theorem provers.

Thanks to Mike Gordon, Magnus Myreen and the reviewers for providing helpful comments on drafts of this paper.

References

1. Campbell, B., Stark, I.: Randomised testing of a microprocessor model using SMT-solver state generation. In: Lang, F., Flammini, F. (eds.) FMICS 2014. Lecture Notes in Computer Science, vol. 8718, pp. 185–199. Springer (2014)
2. Dam, M., Guanciale, R., Nemati, H.: Machine code verification of a tiny ARM hypervisor. In: Sadeghi, A., Armknecht, F., Seifert, J. (eds.) TrustED'13. pp. 3–12. ACM (2013)
3. Fox, A.C.J.: LCF-style bit-blasting in HOL4. In: van Eekelen, M.C.J.D., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. Lecture Notes in Computer Science, vol. 6898, pp. 357–362. Springer (2011)
4. Fox, A.C.J.: Directions in ISA specification. In: Beringer, L., Felty, A.P. (eds.) ITP 2012. Lecture Notes in Computer Science, vol. 7406, pp. 338–344. Springer (2012)
5. Fox, A.C.J., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. Lecture Notes in Computer Science, vol. 6172, pp. 243–258. Springer (2010)
6. Goel, S., Hunt, Jr., W.A., Kaufmann, M., Ghosh, S.: Simulation and formal verification of x86 machine-code programs that make system calls. In: FMCAD 2014. pp. 91–98. IEEE (2014)
7. Jensen, J.B., Benton, N., Kennedy, A.: High-level separation logic for low-level code. In: Giacobazzi, R., Cousot, R. (eds.) POPL 2013. pp. 301–314. ACM (2013)
8. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Jagannathan, S., Sewell, P. (eds.) POPL 2014. pp. 179–192. ACM (2014)
9. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
10. Lim, J., Reps, T.W.: TSL: A system for generating abstract interpreters and its application to machine-code analysis. ACM Trans. Program. Lang. Syst. 35(1), 4 (2013)
11. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J., Gan, E.: Rocksalt: better, faster, stronger SFI for the x86. In: Vitek, J., Lin, H., Tip, F. (eds.) PLDI 2012. pp. 395–404. ACM (2012)
12. Myreen, M.O., Gordon, M.J.C.: Hoare logic for realistically modelled machine code. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. Lecture Notes in Computer Science, vol. 4424, pp. 568–582. Springer (2007)
13. Myreen, M.O., Gordon, M.J.C., Slind, K.: Decompilation into logic — improved. In: Cabodi, G., Singh, S. (eds.) FMCAD. pp. 78–81. IEEE (2012)
14. Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. Lecture Notes in Computer Science, vol. 4732, pp. 189–206. Springer (2007)
15. Sewell, T.A.L., Myreen, M.O., Klein, G.: Translation validation for a verified OS kernel. In: Boehm, H.J., Flanagan, C. (eds.) PLDI. pp. 471–482. ACM (2013)
16. Simon, A., Kranz, J.: The GDSL toolkit: Generating frontends for the analysis of machine code. In: Jagannathan, S., Sewell, P. (eds.) PPREW 2014. p. 7. ACM (2014)
17. Woodruff, J., Watson, R.N.M., Chisnall, D., Moore, S.W., Anderson, J., Davis, B., Laurie, B., Neumann, P.G., Norton, R., Roe, M.: The CHERI capability model: Revisiting RISC in an age of risk. In: ISCA 2014. pp. 457–468. IEEE Computer Society (2014)