Verified Compilation of CakeML to Multiple Machine-Code Targets

Anthony Fox
University of Cambridge, UK
anthony.fox@cl.cam.ac.uk

Magnus O. Myreen
Chalmers University of Technology, Sweden
myreen@chalmers.se

Yong Kiam Tan
Carnegie Mellon University, USA
yongkiat@cs.cmu.edu

Ramana Kumar
Data61, CSIRO / UNSW, Australia
ramana.kumar@data61.csiro.au

Abstract

This paper describes how the latest CakeML compiler supports verified compilation down to multiple realistically modelled target architectures. In particular, we describe how the compiler definition, the various language semantics, and the correctness proofs were organised to minimize target-specific overhead. With our setup we have incorporated compilation to four 64-bit architectures, ARMv8, x86-64, MIPS-64, RISC-V, and one 32-bit architecture, ARMv6. Our correctness theorem places particular emphasis on possible interference from the external environment: the top-level correctness statement takes into account execution of foreign code and per-instruction interference from external processes, such as that of interrupt handlers in operating systems. The entire CakeML development is formalised in the HOL4 theorem prover.

1. Introduction

The CakeML compiler (Tan et al. 2016) has an end-to-end correctness theorem. The correctness theorem states that the concrete machine code produced by the compiler will only behave in ways that are consistent with the source level semantics for the CakeML language. We call this an end-to-end result because the theorem relates source-level behaviour with the behaviour of the concrete machine code as executed by the target architectures.

Following the success of CompCert there have been other ambitious compiler verification projects. Most compiler verifications stop at assembly code (Section 9), and thus fall short of making the proofs relate to concrete machine code. As a result, such compilers may inadvertently generate code that has silent encoding errors, e.g. offsets that overflow or fields that accidentally take on values with special meanings. Writing instruction encoders is non-trivial, as we will illustrate in this paper.

To the best of our knowledge, no previous compiler verification project have delivered a verified end-to-end compiler that (a) targets concrete machine code for several commercially available architectures, and (b) has proofs relating the behaviour of the generated machine code with the semantics of detailed and carefully validated models of real instruction set architectures.

This paper describes how (a) and (b) have been achieved with the latest CakeML compiler. In particular:

- This paper explains how target-specific details of the compiler have been factored out as far as possible to facilitate support for several target architectures. Section 2 describes where and how target-specific details affect the CakeML compiler's operations.
- We explain, in Section 3, how the compiler's top-level correctness theorem takes the target architecture into account and how we model the low-level interference that the surrounding execution environment can cause at the level of machine code.
- Sections 4 through 7 explain how the target-specific proofs were separated from the rest of the proofs, how much effort it is to add a new target, how we overcame various quirks of each supported architecture, and finally how proof automation reduced manual work.

CakeML is formalised in the HOL4 theorem prover and is available at `https://code.cakeml.org`. Our previous publication provides a general overview of the new compiler, while this paper focuses on the multi-target aspect.

2. Target-specific details in the compiler

This section gives an overview of how the compiler implementation is organised to target multiple ISAs. A description of the proofs will be covered in subsequent sections.

Configuration All target-specific details are transported around the compiler as part of the compiler's configuration record. The configuration contains various options, e.g. switches for the optional optimisations, and some compiler state. The target-specific details are kept in a field called asm_conf which has type α `asm_config`. Here α is a type variable for the architecture size, which is in practice instantiated to type 32 or 64. The definition of α `asm_config` is shown in Figure 1. Each instance c of asm_conf carries the following information.

- The name of the ISA: c.ISA.
- An encoding function, c.encode, which takes a well-formed unlabelled assembly instruction and returns the bytes that simulate it in ISA machine code. An instruction is well-formed if it satisfies the criteria outlined informally in the bullet-points below and these criteria are formally checked by a predicate called asm_ok. The datatype for assembly instructions is shown in Figure 2.
- Whether the ISA is big endian: c.big_endian.[1]
- An alignment: machine instructions on this architecture are always a multiple of $2^{c.\mathsf{code\_alignment}}$ bytes.
- Information about available registers: n is available as a register name if $n < c$.reg_count and does not appear in the list c.avoid_regs. We use natural numbers (type `num`) for the register names here. If c.two_reg_arith is set then certain register name combinations are required to make the code fit the register requirements of x86-64 instructions.
- Limits on various offsets, e.g. the minimum and maximum offsets of an unconditional jump are represented by the pair c.jump_offset.
- A predicate c.valid_imm which checks whether constant immediate values are encodable for a particular `binop` or `cmp`. It would be disadvantageous to use simple min/max boundaries since some architectures have complicated criteria that allow large immediate values of certain shapes.
- Finally, a c.link_reg field specifying whether the ISA supports a conventional call instruction: None means that it is not supported and Some v means that v is the name of the register where the call instruction saves the return address.

Example compilation To illustrate how the target-specific details affect compilation, we show how compilation of a simple piece of source code touches on the α `asm_config` record during compilation. We will use part of the code generated for the following source expression as our running example. The parentheses are not necessary here, but might make the nesting clearer.

```
(fn n => (n = 5 000 000))
```

The first place where target-specific details affect compilation is when the functional data abstraction is removed.

[1] Some architectures allow switching endianness at runtime, but we forbid such switching and keep to the conventional endianness for each ISA.

```
α asm_config =
  ⦃ ISA : architecture;
    encode : (α asm → 8 word list);
    big_endian : bool;
    code_alignment : num;
    link_reg : (num option);
    avoid_regs : (num list);
    reg_count : num;
    two_reg_arith : bool;
    valid_imm : (binop + cmp → α word → bool);
    addr_offset : (α word × α word);
    jump_offset : (α word × α word);
    cjump_offset : (α word × α word);
    loc_offset : (α word × α word) ⦄
architecture = ARMv6 | ARMv8 | MIPS | RISC_V | x86_64
```

Figure 1. The compiler's target-specific configuration.

```
α asm =
    Inst (α inst)
  | Jump (α word)
  | JumpCmp cmp num (α reg_imm) (α word)
  | Call (α word)
  | JumpReg num
  | Loc num (α word)
α inst =
    Skip
  | Const num (α word)
  | Arith (α arith)
  | Mem memop num (α addr)
α addr = Addr num (α word)
α arith =
    Binop binop num num (α reg_imm)
  | Shift shift num num num
  | LongMul num num num num
  | LongDiv num num num num num
  | AddCarry num num num num
α reg_imm = Reg num | Imm (α word)
memop = Load | Load8 | Store | Store8
binop = Add | Sub | And | Or | Xor
cmp = Equal | Lower | Less | Test | NotEqual
    | NotLower | NotLess | NotTest
shift = UnsignedLeft | UnsignedRight | SignedRight
```

Figure 2. Target-neutral assembly instruction datatype.

This happens in the transition from DATALANG into WORDLANG (Tan et al. 2016). When the data abstraction is removed, values become machine words and memory becomes explicit in the state of the program. Machine values get their size based on the type variable, α, in the compiler configuration record. When the body of the lambda (`fn`) above enters WORDLANG, it looks roughly as follows. Here we take the liberty to make up a concrete syntax for WORDLANG. The strange switch of value from 5 000 000 to 20 000 000 is due to a shift that inserts marker bits into values. Similarly, the value for true is 24 and false is 2 at this level of abstraction.

```
procedure generated_name_46 (arg1, clos_ptr):
  if arg1 == 20 000 000 then
    temp1 := 24; return temp1
  else
    temp1 := 2; return temp1
```

This program then passes through instruction selection, register allocation and stack concretisation — all of which rely on the compiler configuration to produce code with acceptable register names and immediate constants. If we assume that this compilation targets ARMv6, then the code above turns into the following by these transformations. Note that the large constant has been moved to a separate const assignment, since it does not pass c.valid_imm, but a separate constant assignment can be done for a constant of any size. In the code below, CakeML's internal calling convention has been enforced: register 0 has the return address, register 1 has the first argument etc.

```
procedure generated_name_46:
  reg2 := 20 000 000;
  if reg1 == reg2 then
    reg1 := 24; return_to_addr reg0
  else
    reg1 := 2; return_to_addr reg0
```

A little further down the compiler, the CakeML register names get mapped to target-specific names to fit the target ISA's naming conventions. Note that the renaming is not part of α `asm_config`, but still target specific. For ARMv6, the naming maps register 0 to register 14, and the other register names are shifted down. The resulting code expects the first argument in register 0 and the return address in register 14:

```
procedure generated_name_46:
  reg1 := 20 000 000;
  if reg0 == reg1 then
    reg0 := 24; return_to_addr reg14
  else
    reg0 := 2; return_to_addr reg14
```

At the tail-end of the compiler, our running example gets translated into the following labelled assembly:

```
Label lab46:
    Const 1 20 000 000
    JumpCmp NotEqual 0 (Reg 1) lab46_1
    Const 0 24
    JumpReg 14
Label lab46_1:
    Const 0 2
    JumpReg 14
```

An assembler then runs over this code and replaces the labels with concrete values, which makes each line in the program map into the α `asm` datatype from Figure 2. Note that the compiler can at this point fail to encode the JumpCmp in case the label happens to be too far (outside c.cjump_offset) from the jump instructions. In such cases, the compiler exits with an encoding error.

The generated concrete machine code for ARMv6 is shown below, with assembler mnemonics in comments on the right hand side.

```
00010430 <lab46>:
 10430: e59f1000 ldr r1, [pc];10438 <lab46+0x8>
 10434: ea000000 b 1043c <lab46+0xc>
 10438: 01312d00 ---
 1043c: e1500001 cmp r0, r1
 10440: 1a000001 bne 1044c <lab46+0x1c>
 10444: e3a00018 mov r0, #24
 10448: e12fff1e bx lr
 1044c: e3a00002 mov r0, #2
 10450: e12fff1e bx lr
```

Readers familiar with ARM assembly will immediately note that this is not idiomatic ARM code for two reasons: the large constant is usually stored at the end of the code segment, and the compilation of the short `bne`-jump can be avoided if the `mov` and `b` had been conditionally executed code. We made a decision early on to keep the machine code closely connected to ASM for each target, as opposed to producing completely idiomatic code. This was a decision motivated by the need to limit the complexity of the compiler.

Same example compiled to other targets ARMv6 was the target for the compilation above. Compiling the same example to x86-64 would not required a separate Const instruction to get the large constant, but would otherwise have produced similar looking code using other register names. For MIPS-64 and RISC-V, the constant load would have to be expanded into multiple instructions that build the constant in a register. MIPS-64 has a branch-delay slot which we fill with a no-op — again, we do not attempt to produce strictly idiomatic code here.

3. Correctness theorem and semantics

The top-level correctness theorem for the CakeML compiler is shown in Figure 3. This section explains what the top-level correctness statement means, including the target semantics.

The correctness theorem can be read as follows: if the compiler configuration cc is valid and consistent with a machine configuration mc (which is explained further down), then the list of bytes ($bytes$) produced by a successful execution of the compiler function (compile) will only have behaviours that are consistent with the $behaviours$ of the

```
⊢ config_ok cc mc ⇒
    case compile cc prelude input of
     Success (bytes,ffi_limit) ⇒
      ∃ behaviours.
       cakeml_semantics ffi prelude input =
        Execute behaviours ∧
       ∀ ms.
        code_installed (bytes,cc,ffi,ffi_limit,mc,ms) ⇒
         machine_sem mc ffi ms ⊆
         extend_with_resource_limit behaviours
    | Failure ParseError ⇒
       cakeml_semantics ffi prelude input = CannotParse
    | Failure TypeError ⇒
       cakeml_semantics ffi prelude input = IllTyped
    | Failure CompileError ⇒ true
```

Figure 3. Top-level compiler correctness theorem

source level semantics for the input CakeML program (a combination of *prelude* and *input*).

The machine-level behaviours are consistent with the source-level *behaviours*, if the machine-level semantics (machine_sem, explained further down) only produces behaviours that are a subset of source-level *behaviours* or behaviours that match source-level *behaviours* up to an *out-of-memory error*. The theorem allows for *out-of-memory errors* because the source semantics has no resource limits while the target has hard limits, e.g. finite memory.

We allow the machine-level semantics (machine_sem) to start from any machine state *ms* that satisfies code_installed. This is an property which requires that (1) the *bytes* are present in memory and ready to execute, (2) the machine state is consistent with the compiler and machine configurations, and (3) any call to the foreign function interface (FFI) will behave as specified by the *ffi* parameter.

Parametrised machine semantics The machine-level semantics (machine_sem) used in the top-level correctness theorem is not tied to any particular target.

We parametrised the details of the target using a machine configuration, shown in Figure 4. The machine configuration collects various functions that we need for each target. The most significant function is the next function which specifies how the target machine will execute an instruction. There are also functions for reading the value of registers (get_reg) or memory (get_byte), and information about the calling convention (e.g. caller_saved_regs). Other significant parts include the interference oracles, ffi_interfer and next_interfer, which specify how the environment can interfere with the execution of the next function.

The definition of machine_sem is shown in Figure 5 and follows the functional big-step semantics style described in (Owens et al. 2016). Here machine_sem is defined using a clocked evaluate function, a function that returns Halt, TimeOut or Error. The machine, i.e. machine_sem, has

```
(α, β, γ) target =
 ⦇ config : (α asm_config);
   next : (β → β);
   get_pc : (β → α word);
   get_reg : (β → num → α word);
   get_byte : (β → α word → 8 word);
   state_ok : (β → bool);
   proj : ((α word → bool) → β → γ) ⦈
(α, β, γ) machine_config =
 ⦇ prog_addresses : (α word → bool);
   ffi_entry_pcs : (α word list);
   ptr_reg : num;
   len_reg : num;
   ffi_interfer : (num → num → 8 word list → β → β);
   callee_saved_regs : (num list);
   next_interfer : (num → β → β);
   halt_pc : (α word);
   target : ((α, β, γ) target) ⦈
```

Figure 4. Machine and target configurations

terminating behaviour if there is some clock value such that evaluate reaches Halt; machine_sem has Diverge behaviour if there does not exist such a clock; and machine_sem has Error behaviour if evaluate can get stuck at an Error. Our top-level correctness theorem rules out Error behaviour.

For each step, evaluate checks whether the clock has run out. If the clock is non-zero, then evaluate checks whether the program counter is an address within the code generated from the CakeML compiler. If it is, then next is executed followed by one application of the next_interfer oracle and execution continues from the top. If the program counter is not part of the generated code, then evaluate checks whether execution has reached the halt_pc address. If so, the process Halts. Otherwise, execution must be about to enter an FFI call: the name, i.e. index, of the FFI entry point is looked up and a byte array is read from memory and passed to ffi_-interfer which updates the state according to the FFI call.

4. General compiler proofs

Each major phase of the CakeML compiler is composed at the level of observable semantics as described in (Tan et al. 2016). The final phase which transforms labelled assembly, LabLang programs, to machine code is no different in this respect. However, for the verification of this final phase, we had to know that repeated applications of the next-state function in machine_sem implement the asm instructions of Figure 2. For this reason, the correctness proof of the final phase relies on an assumption about the correctness of the encode function from the compiler configuration record, asm_config of Figure 1. This assumption is called asm_to_target_correct and is shown in Figure 6. The relation asm_step captures the next state semantics of ASM, the predicate target_state_rel relates ASM and target states and the predicate asserts establishes the correctness of all

```
apply_oracle $oracle\ x = (oracle\ 0\ x, (\lambda\, n.\ oracle\ (n + 1)))$

evaluate $config\ ffi\ k\ ms$ =
  if $k = 0$ then (TimeOut,$ms$,$ffi$)
  else if $config$.target.get_pc $ms \in config$.prog_addresses then
    let $ms_1$ = $config$.target.next $ms$; ($ms_2$,$new\_oracle$) = apply_oracle $config$.next_interfer $ms_1$
    in
      if $config$.target.state_ok $ms \wedge config$.target.state_ok $ms_1 \wedge config$.target.state_ok $ms_2$ then
        evaluate ($config$ with next_interfer := $new\_oracle$) $ffi$ $(k - 1)$ $ms_2$
      else (Error,$ms$,$ffi$)
  else if $config$.target.get_pc $ms = config$.halt_pc then
    (if $config$.target.get_reg $ms$ $config$.ptr_reg = $0w$ then Halt Success else Halt Resource_limit_hit,$ms$,$ffi$)
  else
    case find_index ($config$.target.get_pc $ms$) $config$.ffi_entry_pcs $0$ of
      None ⇒ (Error,$ms$,$ffi$)
    | Some $ffi\_index$ ⇒
        case read_byte_array $config$ $ms$ of
          None ⇒ (Error,$ms$,$ffi$)
        | Some $bytes$ ⇒
            let ($do\_ffi$,$new\_oracle$) = apply_oracle $config$.ffi_interfer $ffi\_index$;
                ($new\_ffi$,$new\_bytes$) = call_FFI $ffi$ $ffi\_index$ $bytes$
            in
              case $new\_ffi$.final_event of
                None ⇒ evaluate ($config$ with ffi_interfer := $new\_oracle$) $new\_ffi$ $(k - 1)$ ($do\_ffi$ $new\_bytes$ $ms$)
              | Some $final\_event$ ⇒ (Halt (FFI_outcome $final\_event$),$ms$,$new\_ffi$)

machine_sem $config\ st\ ms$ (Terminate $t\ io\_list$) ⟺
  $\exists\, k\ ms'\ st'$. evaluate $config\ st\ k\ ms$ = (Halt $t$,$ms'$,$st'$) $\wedge$ $st'$.io_events = $io\_list$
machine_sem $config\ st\ ms$ (Diverge $io\_trace$) ⟺
  ($\forall\, k.\ \exists\, ms'\ st'$. evaluate $config\ st\ k\ ms$ = (TimeOut,$ms'$,$st'$) $\wedge$ $st'$.final_event = None) $\wedge$
  $io\_trace = \bigvee k$. fromList (snd (snd (evaluate $config\ st\ k\ ms$))).io_events
machine_sem $config\ st\ ms$ Fail ⟺ $\exists\, k$. fst (evaluate $config\ st\ k\ ms$) = Error
```

Figure 5. A functional big-step semantics for machine-code evaluation (evaluate) and the top-level observable semantics (machine_sem) of the target platform.

the machine states that are reached in the course of running an ASM instruction — this is illustrated graphically in Figure 7. A predicate interference_ok is used to assume that environment env preserves the parts of the machine state that CakeML depends upon, as identified by the projection t.proj from the target configuration. This is illustrated below:

$$\begin{tikzcd} t.\mathsf{next}(ms_i) \arrow[r, "env_i"] \arrow[dr, "t.\mathsf{proj}"'] & ms_{i+1} \arrow[d, "t.\mathsf{proj}"] \\ & cake\text{-}ms_{i+1} \end{tikzcd}$$

The target-neutral part of the compiler assumes the simulation of Figure 7, i.e. asm_to_target_correct. For each target, we then prove this simulation as described in Section 7.

5. Target specifications

CakeML currently targets five different instruction set architectures: ARMv6, ARMv8, MIPS-64, RISC-V and x86-64. These architectures have been specified using the domain specific language L3 (Fox 2015), which provides export to HOL4 and Isabelle/HOL.[2] These specifications have been validated against hardware and/or test suites[3] and they have also been used extensively in other projects, e.g. the ARMv6 model has been used in the seL4 micro-kernel verification (Sewell et al. 2013). The RISC-V model was developed independently by Prashanth Mundkur at SRI International. The specifications were not developed with CakeML in mind and have not been influenced by the design of the compiler. The instruction set coverage for each of these models goes well beyond the present needs of CakeML. Although there are significant differences in the features and details of these architectures (which is reflected in their associated L3 specifications), the abstraction methods described in this paper have helped ensure that these models could be used without the need for adaptations.[4]

Specification style. The L3 target specifications define *state* and *instruction* data types. The state type is a record

[2] The ISA models used in CakeML can be found under the HOL4 directory `examples/l3-machine-code`.

[3] At present, the ARMv8 model has not been fully validated.

[4] Minor changes were made to the RISC-V specification before exporting it to HOL4. This made easier to treat the memory as a byte addressable map.

```
asm_to_target_correct $t$ $\iff$
  target_ok $t$ $\wedge$
  $\forall\, s_1\ i\ s_2\ ms$.
    asm_step $t$.config $s_1$ $i$ $s_2$ $\wedge$ target_state_rel $t$ $s_1$ $ms$ $\Rightarrow$
      $\exists\, n$.
        $\forall\, env$.
          interference_ok $env$ ($t$.proj $s_1$.mem_domain) $\Rightarrow$
            let $pcs$ = all_pcs (length ($t$.config.encode $i$)) $s_1$.pc
            in
              asserts $n$ ($\lambda\, k\ s.\ env\ (n-k)$ ($t$.next $s$)) $ms$
                ($\lambda\, ms'$. $t$.state_ok $ms'$ $\wedge$ $t$.get_pc $ms' \in pcs$)
                ($\lambda\, ms'$. target_state_rel $t$ $s_2$ $ms'$)
target_state_rel $t$ $s$ $ms$ $\iff$
  $t$.state_ok $ms$ $\wedge$ $t$.get_pc $ms = s$.pc $\wedge$
  ($\forall\, a$. $a \in s$.mem_domain $\Rightarrow$ $t$.get_byte $ms$ $a = s$.mem $a$) $\wedge$
  $\forall\, i$.
    $i <$ $t$.config.reg_count $\wedge$ $\neg$mem $i$ $t$.config.avoid_regs $\Rightarrow$
      $t$.get_reg $ms$ $i = s$.regs $i$
```

Figure 6. Target correctness definition.

consisting of numerous ISA components — this includes general purpose registers and main memory. The instruction type provides an abstract syntax representation of machine code instructions. The following functions are defined for each target:

$$\begin{aligned} next &: state \to state \\ fetch &: state \to code \\ decode &: code \to instruction \\ run &: instruction \to state \to state \\ encode &: instruction \to code \end{aligned}$$

The $code$ type is a byte list for x86-64 and a 32-bit word for all of the other architectures. The next state function $next$ captures the operational semantics of the ISA and is typically defined using the functions run, $decode$ and $fetch$. The precise details vary according to architecture, e.g. the MIPS-64 model needs to accommodate the behaviour of branch delays. The function $encode$ is used when developing assemblers (Section 6.1).

Under specification. Each architecture state record contains a *model exception* component, which is used to indicate whether or not the state is valid. For example, some ARMv6 instruction op-codes give rise to *unpredictable* behaviour and this is specified by setting the exception state component to a suitable error value. Model exceptions should not be confused with architectural exceptions and interrupts, which are normally modelled explicitly.

6. Target configuration

The first task in supporting a new target architecture is constructing a target record (t : (α, β, γ) `target`). These target records are defined in separate HOL4 scripts, one for each target. Currently each of the next state functions t.next come directly from L3 specifications of the architectures (Section 5). The remaining target record elements are discussed in the following paragraphs.

Registers and memory The functions t.get_pc, t.get_reg and t.get_byte provide an interface to each ISA specification. These definitions are quite straightforward; for example, for ARMv6 they are defined such that:

```
arm6.get_pc $s$ = $s$.REG RName_PC
arm6.get_reg $s$ $n$ = $s$.REG (R_mode $s$.CPSR.M (n2w $n$))
arm6.get_byte $s$ = $s$.mem
```

Here s represents an ARM state, s.REG is a map from register names to 32-bit values, and s.mem is a map from 32-bit addresses to bytes. The function R_mode gives the register name corresponding with the current processor mode and a 4-bit register index.[5] The program counter is called RName_PC and this always corresponds with register fifteen. With the other target architectures the set of visible registers is independent of the operating mode (such as user mode or a system mode), so register access is simpler. However, the RISC-V model supports multiple cores, so register access is parametrised by the core number.

State predicate It is important that the property t.state_ok holds when running CakeML code, otherwise compiled programs are not guaranteed to behave correctly. Verifying asm_to_target_correct proves t.state_ok will not switch from true to false when running machine code generated by CakeML. However, programs must be started from a valid state, and all "foreign" code (including exception handlers) must be well behaved (the state must be valid when control is handed back to CakeML code). For ARMv6, the following predicate is used:

```
arm6.state_ok $s$ $\iff$
  GoodMode $s$.CPSR.M $\wedge$ $\neg s$.CPSR.E $\wedge$ $\neg s$.CPSR.J $\wedge$
  $\neg s$.CPSR.T $\wedge$ $s$.Architecture = ARMv6 $\wedge$
  $\neg s$.Extensions Extension_Security $\wedge$
  $s$.exception = NoException $\wedge$ aligned 2 ($s$.REG RName_PC)
```

This asserts that: the processor mode is valid (there are only seven valid operating modes); the processor is running in little-endian configuration and is not in Jazelle or Thumb mode; the machine implements ARMv6 without security extensions;[6] a model exception has not occurred (Section 5); and the program counter is aligned to four bytes (the lower two bits are clear).

State projections The function t.proj identifies state components that are needed for the interference_ok property.

[5] The 5-bit component s.CPSR.M encodes the current operating mode, e.g. it will have value sixteen when running in user mode. The function (n2w : `num` $\to$ α `word`) maps natural numbers to words (modulo 2^α).

[6] The s.Architecture and s.Extensions components never change value. Specialising the model to ARMv6 establishes the availability of instructions, e.g. CakeML makes use of BLX and BX instruction, which are not available under ARMv4.

$$\begin{tikzcd}
s_1 \arrow[rrrrrrr, "\mathsf{asm\_step}"] \arrow[d, leftrightarrow, "\mathsf{target\_state\_rel}\ t"] & & & & & & & s_2 \arrow[d, leftrightarrow, "\mathsf{target\_state\_rel}\ t"'] \\
ms \arrow[r, "t.\mathsf{next}"] & \cdot \arrow[r, "env_0"] & ms_1^\star \arrow[r, "t.\mathsf{next}"] & \cdot \arrow[r, "env_1"] & ms_2^\star \arrow[r, "t.\mathsf{next}"] & \cdot \arrow[r, "env_2"] & ms_3^\star \arrow[r, dashed] & ms_{n+1}
\end{tikzcd}$$

Figure 7. Commuting diagram for target correctness. Machine-states marked with $\star$ must satisfy the property t.state_ok and the program counter for these states must be valid (the program counter must not go beyond the machine code produced by the instruction encoding function).

These include, but are not limited to, all of the components that are referenced by t.state_ok, t.get_pc, t.get_reg and t.get_byte. The projection for ARMv6 is:

```
arm6.proj d s =
  (s.CPSR,s.Architecture,s.Extensions,s.exception,
   s.REG ∘ R_mode s.CPSR.M,fun2set (s.mem,d))
```

Here d is the set of memory addresses that CakeML is using and fun2set $(f,d) = \{(a,f\ a) \mid a \in d\}$ gives us the graph of a function with respect to a domain. The term s.REG ∘ R_mode s.CPSR.M represents the general purpose registers that are visible in the current processor mode. The Current Program Status Register (s.CPSR) is projected out in full. In particular, this ensures that the arithmetic/logic flags are preserved (e.g. the carry flag s.CPSR.C), as these are used by some ASM instructions.

ASM configuration The components of record t.config were described in Section 2. The basic configuration values for each architecture are show in Table 1. The offset limits and a description of the immediate values for binary operations are shown in Table 2. The precise values for the offsets and immediate values are influenced by the definition of the encode function t.config.encode, which is discussed below.

6.1 ASM instruction encoders

Functions t.config.encode are defined by pattern matching on ASM syntax, with calls made to target specific encoders (specified in L3). This is achieved by using the abstract syntax type for the target machine code (Section 5). In some cases there may be conditional expressions that choose instructions based on the parameters to the instruction (register indices and/or immediate values).

Example The ASM instruction

```
JumpCmp cmp r (Imm i) a
```

compares register r with immediate value i using cmp (e.g. cmp could be Equal or Lower); if the result is true then a jump is made by adding offset a to the program counter, otherwise the program counter moves to the next instruction. The ARMv6 encoding for this instruction is the following.

```
arm6_enc (JumpCmp cmp r (Imm i) a) =
  (let (opc,c) = arm6_cmp cmp and
       imm12 = THE (EncodeARMImmediate i)
   in
     arm6_encode 14w
       (Data (TestCompareImmediate (opc,n2w r,imm12))) @
     arm6_encode c (Branch (BranchTarget (a − 12w))))
```

The function arm6_encode takes a condition code together with ARM abstract syntax and it returns a list of bytes (machine code). The bytes for each ARM instruction are concatenated together. The function EncodeARMImmediate (defined in L3) attempts to encode a 32-bit immediate value using a 12-bit representation and the function arm6_cmp picks an appropriate condition code, as well as a test or comparison operation.[7]

The function arm6_enc can be evaluated within the HOL4 logic, e.g. one can prove

```
⊢ arm6_enc (JumpCmp Equal 1 (Imm 175w) 2048w) =
    [175w; 0w; 81w; 227w; 253w; 1w; 0w; 10w].
```

which corresponds with the ARM code

```
cmp r1, #0xAF   ; e35100af
beq +#0x7FC     ; 0a0001fd
```

Note that the function EncodeARMImmediate could fail to produce a 12-bit encoding imm_{12} for the 32-bit immediate i. CakeML uses the component arm6.config.valid_imm to check if this aspect of the encoding is successful. Similarly, the encoding will only be valid when the 32-bit address $a - 12w$ can be encoded as a 24-bit value. This is checked using the bounds arm6.config.cjump_offset.

Observe that the offset supplied to the ARM branch is $a - 12w$, whereas the ASM instruction will branch to $pc + a$. The reason for this is that when the branch destination address is computed the ARM program counter will be eight bytes ahead of the address used to fetch the branch (this is a legacy of early 3-stage pipeline designs); furthermore, the branch must be relative to the *first* instruction of the encoding, which is four bytes before the branch. This

[7] The condition codes are: 14 (always run); 11 (`LT`); 10 (`GE`); 3 (`CC`); 2 (`CS`); 1 (`NE`); and 0 (`EQ`). The op-codes are: 2 (`SUB`) and 0 (`AND`).

	ARMv6	ARMv8	MIPS-64	RISC-V	x86-64
Word size	32	64	64	64	64
Endianness	Little	Little	Big	Little	Little
Code alignment (bytes)	4	4	4	4	1
Number of registers	16	32	32	32	16
Link register	14	30	31	1	-
"Avoid" registers	13, 15	31	0, 1, 25 – 29	0, 2, 3, 31	4, 5
Two register binary-ops	No	No	No	No	Yes

Table 1. Basic ISA configuration. The targets are mostly RISC designs with 32-bit instructions that align on four byte boundaries, however x86-64 is a CISC design with variable width instructions, e.g. `90` and `48 BB F0 DE BC 9A 78 56 34 12` are both valid x86-64 codes (for `nop` and `mov rbx, 0x123456789abcdef0` respectively).

	Location	**Memory**	**Jump**	**Conditional Jump**
ARMv6	`-0xFFF7 - 0x10007`	`-0xFFF - 0xFFF`	`-0x1FFFFF8 - 0x2000007`	`-0x1FFFFF4 - 0x200000B`
ARMv8	`-0x80000 - 0x7FFFF`	`-0x100 - 0xFF`	`-0x8000000 - 0x7FFFFFF`	`-0xFFFFC - 0x100003`
MIPS-64	`-0x7FF4 - 0x8007`	`-0x8000 - 0x7FFF`	`-0x8000 - 0x7FFF`	`-0x8000 - 0x7FFF`
RISC-V	`-0x80000000 - 0x7FFFF7FF`	`-0x800 - 0x7FF`	`-0x10000 - 0xFFFFF`	`-0xFFFF8 - 0x100003`
x86-64	`-0x7FFFFFF9 - 0x80000006`	`-0x80000000 - 0x7FFFFFFF`	`-0x7FFFFFF3 - 0x80000004`	`-0x7FFFFFF3 - 0x80000004`

		Immediate values (for binary operations)
ARMv6		12-bit encoding (an 8-bit value is rotated right by a 4-bit value)
ARMv8	Arithmetic	12-bit immediate, optionally shifted left 12 places
	Logical	Bit run-length encoding (specified algorithmically)
MIPS-64	Subtract	`-0x7FFF - 0x7FFF` (encoded using `DADDIU`)
	Arithmetic	`-0x8000 - 0x7FFF`
	Logical	`0 - 0xFFFF`
RISC-V	Subtract	`-0x7FF - 0x7FF` (encoded using `ADDI`)
	Otherwise	`-0x800 - 0x7FF`
x86-64		`-0x80000000 - 0x7FFFFFFF`

Table 2. Configuration of offset ranges and valid immediate values. [All of the number ranges are inclusive.] The ARM architectures have complex immediate value encoding schemes, whereas the other architectures have simple ranges that depend on the binary operation type. MIPS-64 and RISC-V do not have dedicated 'subtract immediate' instructions, so addition instructions (with a negated immediate) are used instead.

explains why many of the offset ranges in Table 2 look a bit peculiar. Subtleties like this provide a strong motivating case for the use of formal verification. It is extremely easy to get encodings and/or target configuration values slightly wrong.

6.2 Instruction selection

The target encoder definitions have been manually developed and they give rise to many possible instruction sequences for each architecture. Table 3 shows the set of instruction sequences for four ASM instruction patterns. Some issues regarding instruction selection are discussed below.

Const The instruction Const $r\ i$ transfers an immediate value i to a register r, i.e. *any* 32-bit value for ARMv6 and *any* 64-bit value for the other architectures. Only x86-64 is capable of achieving this with just one instruction. Finding the optimal (shortest and fastest) sequence of instructions for the other targets is non-trivial, since there are many target specific tricks that could be used. To avoid proof complexity, the CakeML encoders have been kept reasonably simple. The MIPS-64 and RISC-V architectures do not provide a *direct* means of transferring an immediate value to the top thirty-two bits of a register, which is why sequences of six instruction (employing left shifts) are used. The sign-extension used in RISC-V was problematic and the encoder resorts to using tricks with `XORI`, as well as using an extra register (register thirty-one is added to riscv.config.avoid_regs list). An alternative approach would be to use a load instruction instead; this method is used for ARMv6, which has a more complex immediate encoding scheme.[8] Fortunately ARMv8 does make it relatively easy to progressively build up immediate values, with at most four instructions needed.

[8] Having data (an immediate) within a code segment is not ideal with modern Harvard architectures, so this encoding choice may be changed.

	Constant: Inst (Const _ _)
ARMv6	`[ MOV|MVN ] [ LDR; B; <const> ]`
ARMv8	`[ MOVZ|MOVW|ORR ] [ MOVK; MOVK; MOVK; MOVK ]`
MIPS-64	`[ ORI|ADDIU ] [ LUI; XORI ] [ LUI; ORI; DSLL; ORI; DSLL; ORI ]`
RISC-V	`[ ORI ] [ LUI; ADDI|XORI ] [ LUI; ADDI|XORI; LUI; ADDI|XORI; SLLI; XOR|OR ]`
x86-64	`[ MOV ]`
	Add with carry: Inst (Arith (AddCarry _ _ _ _))
ARMv6	`[ CMP; CMNEQ; ADCS; MOVCC; MOVCS ]`
ARMv8	`[ CMP; CCMN; ADCS; MOV; ADC ]`
MIPS-64	`[ SLTU; DADDU; SLTU; DADDU; SLTU; OR ]`
RISC-V	`[ SLTU; ADD; SLTU; ADD; SLTU; OR ]`
x86-64	`[ CMP; CMC; ADC; MOV; ADC ]`
	Conditional Jump (immediate): JumpCmp _ _ (Imm _) _
ARMv6	`[ CMP; BEQ|BCC|BLT|BNE|BCS|BGE ] [ TST; BEQ|BNE ]`
ARMv8	`[ CMP; B.EQ|B.CC|B.LT|B.NE|B.CS|B.GE ] [ ANDS; B.EQ|B.NE ]`
MIPS-64	`[ DADDIU|SLTI|SLTIU|ANDI; BEQ|BNE; NOP ]`
RISC-V	`[ ORI; BEQ|BLTU|BLT|BNE|BGEU|BGE ] [ ANDI; BEQ|BNE ]`
	`[ ORI; BEQ|BLTU|BLT|BNE|BGEU|BGE; JAL ] [ ANDI; BEQ|BNE; JAL ]`
x86-64	`[ CMP; JE|JB|JL|JNE|JNB|JNL ] [ TEST; JE|JNE ]`
	Location with offset: Loc _ _
ARMv6	`[ ADD|SUB ] [ ADD; ADD ] [ SUB; SUB ]`
ARMv8	`[ ADR ]`
MIPS-64	`[ BLTZAL; DADDIU ] [ ORI; BLTZAL; DADDIU; ORI ]`
RISC-V	`[ AUIPC; ADDI ]`
x86-64	`[ LEA ]`

Table 3. Machine code instruction sequences for some ASM instructions. Instruction sequences are represented by bracketed blocks with semi-colons indicating the order of instructions. Instruction variants within a sequence are marked with a | infix. Thus, with conditional jumps there are 16 possible instruction sequences for RISC-V and eight possible sequences for the other targets (one for each jump condition). Some mnemonics may have complex encodings (giving rise to further cases), e.g. the precise encoding of MOV on x86-64 will depend on the choice of registers and on the size of any immediate value.

AddCarry The ARMv6, ARMv8 and x86-64 architectures all support an ADC instruction, which sums together registers using a carry-in flag. In these architectures arithmetic/logic flags form part of the programmer's model state. The carry flag can be updated with the carry output of a sum — this happens by default on x86-64 and with ADCS on ARMv6 and ARMv8. However, MIPS-64 and RISC-V do not provide arithmetic/logic flags and for uniformity it was decided not to have flags within ASM as well. Instead, the instruction AddCarry r_1 r_2 r_3 r_4 treats register r_4 as a carry flag. This instruction first computes

$$r = R_2 + R_3 + \text{if } R_4 = 0 \text{ then } 0 \text{ else } 1$$

(where R_i is the natural number value of register r_i) and updates register r_1 with r mapped to a 32-bit (ARMv6) or 64-bit value, and updates r_4 with 1 or 0, according to whether or not there was a carry-out. This instruction can be implemented on all of the target architectures, but it is not idiomatic for any of them. For ARMv6, ARMv8 and x86-64, two instructions are required either side of the add-with-carry in order to transfer the carry flag to and from a register. For MIPS-64, two DADDU instructions are needed, with tests for carry out occuring after each of them. This is illustrated below:

```
-- Inst (Arith (AddCarry 2 3 4 5)) --
sltu  $1, $0, $5  ; r1 := is 0 < r5 (1 or 0)
daddu $2, $3, $4  ; r2 := r3 + r4
sltu  $5, $2, $4  ; r5 := is r2 < r4 (carry out 1)
daddu $2, $2, $1  ; r2 := r2 + r1 (result)
sltu  $1, $2, $1  ; r1 := is r2 < r1 (carry out 2)
or    $5, $5, $1  ; r5 := r1 or r5 (carry out)
```

This code uses register one to facilitate the computation and this register is included in the mips.config.avoid_regs list. Similar code is generated for RISC-V.

JumpCmp On ARMv6, ARMv8 and x86-64 the instruction JumpCmp is implemented cleanly by virtue of their arithmetic/logic flags. With RISC-V the immediate is first moved to register thirty-one and then a register-register compare and branch is used. Unfortunately, this gives a fairly short offset range, so a three instruction form is used when the offset is larger, e.g.

```
--  JumpCmp Less 5 (Imm 4w) 16w --
ori t6, $0, 0x4 ; r31 := 4 (t6 is r31, t0 is r5)
blt t0, t6, 0xc ; if r5 < r31 then pc := pc + 12
```

```
-- JumpCmp Less 5 (Imm 4w) 0x8000w --
ori t6, $0, 0x4 ; r31 := 4
bge t0, t6, 0x8 ; if r5 < r31 then pc := pc + 8
jal $0, 0x7ff8  ; pc := pc + 0x7ff8
```

With MIPS-64 register one either stores the result of the comparison (using the instruction SLTI, SLTIU or ANDI) or it stores the immediate value itself (transferred using DADDIU). The branch is then taken via either BEQ or BNE, and the branch delay slot is filled with a NOP.

Loc The ASM instruction Loc r a transfers the address $pc + a$ to register r. This instruction is easily implemented on ARMv8 and x86-64, via ADR and LEA. On ARMv6, the program counter is accessible as a normal register, so one can use ADD r, pc, #a. However, this would result in a very limited single byte offset range, since the 12-bit immediate encoding is not continuous. The offset range is extended by using either two additions or two subtractions.

The RISC-V instruction AUIPC r, i transfers the address $pc + \mathsf{SignExtend}\ (i \ll 12)$ to register r. This offers a large offset range and an ADDI instruction can be used to set the lower twelve bits. The address i for AUIPC must take sign-extension into consideration, i.e. $i = a - \mathsf{SignExtend}\ a'$ where a' is the lower twelve bits of a.

MIPS-64 does not have a specialist instruction for reading the program counter, so a conditional branch and link instruction (BLTZAL) is used. This instruction stores the link address ($pc + 8w$) in register thirty-one, regardless of whether or not the branch is taken. A DADDIU instruction can then be used to compute the required offset. If r is not (conveniently) register thirty-one then additional instructions are needed to save and restore register thirty-one. This is illustrated below:

```
--  Loc 31 0xF00w ----
bltzal $0, 0           ; r31 := pc + 8
daddiu $31, $31, 0xef8 ; r31 := r31 + 0xef8
--  Loc 2 0xF00w ----
ori    $1, $31, 0      ; r1 := r31
bltzal $0, 0           ; r31 := pc + 8
daddiu $2, $31, 0xef4  ; r2 := r31 + 0xef4
ori    $31, $1, 0      ; r31 := r1
```

The branches are never taken because a 'less-than-zero' test is made on register zero, which is hard-wired to value zero. As such, the branch address (zero above) is irrelevant and could be set to anything. The immediate value arguments for the DADDIU instructions differ by four because the BLTZAL instruction occurs one instruction (four bytes) later in the second code snippet.

7. Target proofs

The property asm_to_target_correct has been verified for five target architecture records. This section gives an overview of these proofs. Naturally, the majority of the proof effort centres on checking the commuting property show in Figure 7. It is also necessary to check the condition target_ok t but this part of the verification is comparatively short and uninteresting, so this is not discussed further here.

Proof outline Below is a sketch of the main verification:

1. The proof starts by considering two ASM states s_1, s_2, an ASM instruction i and a target machine state ms, with:

   $$\mathsf{asm\_step}\ t.\mathsf{config}\ s_1\ i\ s_2 \quad \text{and} \quad \mathsf{target\_state\_rel}\ t\ s_1\ ms$$

   By the definition of asm_step it is known that the bytes t.config.encode i are in ASM memory at the program counter address, and also i must satisfy the predicate asm_ok. This means that instruction i must conform to the limitations expressed within the target record t. In particular, the register indices are valid, offsets are within range, and immediate values are encodable. The state s_2 is related to s_1 by a single application of the ASM next state function. The state ms satisfied t.state_ok by the definition of target_state_rel.
2. The proof proceeds by case splitting on the structure of i. Further case splitting is considered in accordance with the definition of t.config.encode. All instruction sequences for the target architecture must be considered. Each case is checked as follows.
3. The expression t.config.encode i is simplified to give a symbolic representation of the machine code bytes in ASM memory. Also, simplification is applied for the ASM next state function, so that s_2 becomes expressed in terms of register and memory updates to s_1.
4. A witness is supplied for cycle count n. This must be one less than the exact number of ISA cycles that are needed to run the machine code under consideration. At this stage the interference_ok property can be assumed to hold for an environment function env.
5. The asserts property is expanded with respect to n. At this stage the proof goal consists of n instances of t.state_ok and t.get_pc $ms \in pcs$, and an instance of target_state_rel t s_2 ms', where ms' is the machine state after running $n + 1$ instructions with interleaved interference from env.
6. The machine code bytes in memory are split into single instruction blocks and it is inferred that these bytes must be in the target memory at incremental locations (based on target_state_rel holding for the initial states).
7. The next state function for the target architecture is symbolically evaluated and simplification occurs using rewrites derived from interference_ok for env. This happens incrementally, spanning $n + 1$ applications of the next state function.
8. Finally, the properties from stage 5 are all discharged, using simplification and bit-blasting. The simplifications

may incorporate pre-proven lemmas that are specific to the target verification.

Proof implementation Laboriously writing a purely conventional HOL4 tactic proof that follows the outline above presents a number of difficulties — the resulting proof would invariably be very long, complex and potentially slow to run. One reason for this is that there is inherently a lot repetition, since stages 3 – 8 have to be repeated (with modifications) for each case of interest (of which there are many). Furthermore, a significant challenge is stage 7, where the ISA model must be evaluated with respect to a list of machine code bytes. These bytes are generated by an encoder function and they may contain bit-vector literals, concatenations, sub-field extractions, casts and sign-extensions. The ISA models themselves are very complex and naive evaluation (direct simplification using definitions) is intractable.

To overcome these challenges, custom tactics have been developed for each architecture, which help in partially automating stages 3 – 8. These tactics are similar in style for each architecture and they make use of pre-existing HOL4 proof tools (Fox 2015). Thus, the target proofs mostly take the form of case splitting over ASM instructions, followed by a single application of a custom tactic, often called `next_tac`. These tactics achieve the following:

- They may automatically determine the cycle count witness n. With the four RISC architectures n can be deduced from the number of bytes in the encoding, possibly with checks for special cases, e.g. examining instruction i to see if code will be skipped due to a branch. For x86-64 the tactic `next_tac` takes an extra argument, which is a list of instruction lengths. This is used to determine n and to split the bytes into chunks of the right size (stage 6).
- They orchestrate calls to target specific *step tools*, which help evaluate the instruction semantics (Fox 2015). It is also necessary to accommodate interference from the environment env using assumptions derived from interference_ok.
- The automation also takes care of stage 8, where ASM states are compared with states of the target architecture.

The interface to the step tools is a list of terms that represent the bit values of the machine code instruction. These bit values will come from the encoder and they will either be concrete (T or F) or an expression, e.g. a ' 3 would denote bit three of an ASM address a. The output of the step tool is a *step theorem* of the form

$$A_0, \ldots, A_n \vdash next(s_0) = s_1$$

where state s_1 is expressed in terms of updates to s_0 (using values constructed from bits of the input op-code). The hypotheses A_i cover: absence of ISA exceptions (e.g. checks on address alignments); the processor mode and ISA configuration (e.g. endianness and architecture version); and assertions that the bytes of the evaluated instruction are in memory at the current program counter location. These hypotheses are discharged using assumptions coming from target_state_rel, t.state_ok, asm_ok and asm_step (it is assumed that there are no dynamic ASM failures).

When considering longer sequences of instructions, the expression representing the final machine state are invariably large and complex. However, discharging the proof obligations in stage 8 has not been overly problematic, mainly due to the availability of a bit-blasting decision procedure.

The simplest ASM instruction to verify is Skip, which corresponds with the machine-code `e1a00000` (ARMv6), `d503201f` (ARMv8), `00000000` (MIPS-64), `00000013` (RISC-V), `90` (x86-64). A costly instruction is JumpCmp because there are two forms (register and immediate), two outcomes (taken and not taken), eight conditions and at least two instructions for each encoding.

The proof automation described above has resulted in reasonably short and efficient proofs, with 1,091 lines of configuration scripts, 3,813 lines of proof (including code for custom tactics) and a total proof run time of around thirty-eight minutes (on a a Core i7-4771 machine). However, replaying the proofs step-by-step is not straightforward. Maintaining the proofs requires a good understanding of the automation. This contrasts with conventional HOL4 proofs, which are relatively easy for HOL4 users to step through. This situation could be improved by adapting the automation so that it generates lots of diagnostic tracing information.

Verification benefits Verifying asm_to_target_correct helped to identify many errors in the candidate target encoders. A common error would be for address or immediate value calculations to be subtly incorrect. For example, the Const encoding for RISC-V required quite a few iterations to get right. Longer sequences of machine code can be prone to accidental side-effects, i.e. registers being modified when they should not be. In some cases, the property asm_ok has been updated to reflect the ability to encode instructions for a target. For example, for MIPS-64 and RISC-V the necessary register inequality assertion $r_1 \neq r_3 \wedge r_1 \neq r_4$ for AddCarry was established during the course of the verification.

8. Benchmarks

We ran a suite of CakeML compiler benchmarks on all five targets (an emulator was used for RISC-V and MIPS-64). On each target, we tested the compiler with all optimisations on and all optimisations off. The optimisations always improved execution times, and we observed average reduction in execution times ranging from 63.4% on MIPS-64 to 39.2% on ARMv6.

Interestingly, some of the unoptimised MIPS-64 benchmarks were rejected by the compiler because the assembler produced jump address offsets that were too large. At the time of writing, we are looking into ways for the compiler to rewrite these jumps to use register indirection.

9. Related work

The first version of the CakeML compiler (Kumar et al. 2014) supports compilation down to x86-64 assembly via a bytecode IL. Each bytecode instruction is mapped to a block of x86-64 instructions, and the translation was manually verified using decompilation tools. Our approach here goes further by compiling down to an assembly IL. This exposes more target-specific detail to the compiler, and allowed us to perform target-specific optimisations. At the same time, it hides the low-level implementation details, allowing the compiler to target multiple architectures with less overhead.

The CompCert C compiler (Leroy 2009) compiles down to the Mach intermediate language before further compilation to different assembly languages for each of its targets. Importantly, these assembly languages are modelled at the AST level i.e. assembling and encoding is done outside the verified part of the compiler. In contrast, our compiler assembles and generates byte-level outputs for each ISA; this is made possible by our low-level machine models. Our compiler also differs in compiling down to a shared assembly language for all the targets. CompCert's approach opens up more opportunities for target-specific optimisation at the last compilation step down to specific target assembly. For example, peephole optimisations were provided for the x86 target (Mullen et al. 2016). We believe similar optimisations (parameterised by ISA) could be performed for CakeML at the shared assembly level although we have yet to explore that possibility.

Other verified compilers typically target assembly languages that are idealised in some way. The Lambda Tamer project (Chlipala 2010) presented a verified compiler down to idealised assembly with infinite memory and registers storing natural numbers. The original CompCert memory model was also idealised in a similar way, with infinite memory, and pointers represented using integers. Subsequent work (Besson et al. 2015; Leroy et al. 2012) concretises the memory model's pointer representation. We cannot rely on such idealised pointers in the CakeML compiler because (1) they are not present in our low-level models, and (2) we needed word-level control over pointers for garbage collection and pattern matching optimisations.

The verification of the Piton compiler (Moore 1989) goes even further than other verified compilers (including ours). The FM8502 machine on which Piton is to be executed is verified down to the gate level. However, this processor design lacks a physical implementation.

Another line of research provides assembly languages that are suited for verification. Bedrock (Chlipala 2011, 2013) is an assembly-like language designed to support highly automated reasoning over low-level code. The CAP framework (Ni and Shao 2006) allows verifiers to apply Hoare logic-style reasoning over an assembly language. Both Bedrock and CAP languages are modelled after real assembly languages, but their compilation down to real assembly is unverified. Direct assembly-level reasoning is not the main purpose of our IL, but it is certainly possible to reason over its semantics (as we do in our compiler).

10. Summary

This paper has explained how the latest verified CakeML compiler is able to compile to several machine-code targets. Our verification proofs reach all the way down to the low-level details of machine code execution for each target.

References

F. Besson, S. Blazy, and P. Wilke. A concrete memory model for compcert. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving (ITP)*, LNCS. Springer, 2015.

A. Chlipala. A verified compiler for an impure functional language. In M. V. Hermenegildo and J. Palsberg, editors, *Principles of Programming Languages (POPL)*. ACM, Jan. 2010.

A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Programming Language Design and Implementation (PLDI)*. ACM, 2011.

A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *International Conference on Functional Programming (ICFP)*. ACM, 2013.

A. C. J. Fox. Improved tool support for machine-code decompilation in HOL4. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving (ITP)*, LNCS, 2015.

R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In S. Jagannathan and P. Sewell, editors, *Principles of Programming Languages (POPL)*. ACM, 2014.

X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7), 2009. doi:10.1145/1538788.1538814.

X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert memory model, version 2. Research report RR-7987, INRIA, June 2012. URL `http://hal.inria.fr/hal-00703441`.

J. S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman. Verified peephole optimizations for CompCert. In C. Krintz and E. Berger, editors, *Programming Language Design and Implementation (PLDI)*. ACM, 2016.

Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. *SIGPLAN Not.*, 41(1), Jan. 2006. doi:10.1145/1111320.1111066.

S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan. Functional big-step semantics. In P. Thiemann, editor, *European Symposium on Programming (ESOP)*, LNCS. Springer, 2016.

T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Programming Language Design and Implementation (PLDI)*. ACM, 2013.

Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish. A new verified compiler backend for CakeML. In J. Garrigue, G. Keller, and E. Sumii, editors, *International Conference on Functional Programming (ICFP)*. ACM, 2016.