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
yongkia@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 allows interference from the environment: the top-level correctness statement takes into account execution of foreign code and per-instruction interference from external processes, such as interrupt handlers in operating systems. The entire CakeML development is formalised in the HOL4 theorem prover.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification—Correctness proofs, Formal methods; F.3.1 [Logics and meanings of programs]: Specifying and Verifying and Reasoning about Programs—Mechanical verification, Specification techniques, Invariants

Keywords Compiler verification; ML; verified assembly.

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 has 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:

• We explain how target-specific details of the compiler have been factored out as far as possible to facilitate support for several targets. Section 2 describes where and how target-specific details affect the compiler’s operations.
• We explain, in Section 3, how our 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](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 optional optimisations, and some compiler state. The target-specific details are kept in a field called `t.config` with type `α asm_config`. Here `α` is a type variable for the architecture size, which is in practice instantiated to type 32 or 64. Figure 1 shows the definition of `α asm_config`. Each instance `c` carries the following:

- 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 implement it in ISA machine code. An instruction is well-formed if it satisfies the criteria outlined informally in the bullet-points below; 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`.
- An alignment: machine instructions on this architecture are always a multiple of `2^c.code_alignment` bytes.
- Information about available registers: `n` is available as a register name if `n < c.reg_count` and `n` does not appear in the list `c.avoid_regs`. We use natural numbers (type `num`) for register names. If `c.two_reg_arith` is set then certain combinations of names must be used, to fit the requirements of two-register architectures (e.g. x86-64).
- 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 it is not supported and Some `v` means there is a call instruction and it saves the return address in `v`.

**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:

\[ α asm_config = \]

\[
\begin{array}{l}
| ISA : \text{architecture};
| encode : (\alpha \text{ asm} \to \text{8 word list});
| big_endian : \text{bool};
| code_alignment : \text{num};
| link_reg : (\text{num option});
| avoid_regs : (\text{num list});
| reg_count : \text{num};
| two_reg_arith : \text{bool};
| valid_imm : (\text{binop + cmp} \to \text{word} \to \text{bool});
| addr_offset : (\text{word} \times \text{word});
| jump_offset : (\text{word} \times \text{word});
| cjump_offset : (\text{word} \times \text{word});
| loc_offset : (\text{word} \times \text{word}) \}
\end{array}
\]

architecture = ARv6 | ARv8 | MIPS | RISC_V | x86_64

---

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

\[ α asm = \]

\[
\begin{array}{l}
| \text{Inst} (α \text{ inst})
| \text{Jump} (α \text{ word})
| \text{JumpCmp} \text{ cmp num} (α \text{ reg_imm}) (α \text{ word})
| \text{Call} (α \text{ word})
| \text{JumpReg} \text{ num}
| \text{Loc} \text{ num} (α \text{ word})
| \text{inst} = \]

\[
\begin{array}{l}
\text{Skip}
| \text{Const num} (α \text{ word})
| \text{Arith} (α \text{ arith})
| \text{Mem} \text{ memop num} (α \text{ addr})
| \text{addr} = \text{Addr} \text{ num} (α \text{ word})
| \text{arith} = \]

\[
\begin{array}{l}
\text{Binop \text{ binop num} num (α \text{ reg_imm})}
| \text{Shift \text{ shift num num}} \text{ num}
| \text{Div \text{ num num num}}
| \text{LongMul \text{ num num num num}}
| \text{LongDiv \text{ num num num num num}}
| \text{AddCarry \text{ num num num num}}
| \text{reg_imm} = \text{Reg} \text{ num} | \text{Imm} (α \text{ word})
| \text{memop} = \text{Load} | \text{Load8} | \text{Store} | \text{Store8}
| \text{binop} = \text{Add} | \text{Sub} | \text{And} | \text{Or} | \text{Xor}
| \text{cmp} = \text{Equal} | \text{Lower} | \text{Less} | \text{Test} | \text{NotEqual} | \text{NotLower}
| \text{NotLess} | \text{NotTest}
| \text{shift} = \text{UnsignedLeft} | \text{UnsignedRight} | \text{SignedRight}
\end{array}
\]

---

Figure 2. Target-neutral assembly instruction datatype.

1 Some architectures allow switching endianness at runtime, but we forbid such switching and keep to the conventional endianness for each ISA.
example. The parentheses are not necessary here, but might make the nesting clearer.

(fn n => (n = 5000000))

The first place where target-specific details affect compilation is when the functional data abstraction is removed. 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, \( \alpha \), 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 \( 5000000 \) to \( 20000000 \) 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.

```plaintext
procedure generated_name_46 (arg1, clos_ptr):
  if arg1 == 20000000 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 constant assignment, since it does not pass as an immediate. Similarly, the value for true is 24 and false is 2 at this level of abstraction.

```plaintext
procedure generated_name_46:
  reg2 := 20000000;
  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 the \( \alpha \) 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:

```plaintext
procedure generated_name_46:
  reg1 := 20000000;
  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:

```plaintext
Label lab46:
  Const 1 20000000
  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 \( \alpha \) 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.

```plaintext
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: e12ff1e bx lr
1044c: e3a00002 mov r0, #2
10450: e12ff1e 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 short bne-jump could be avoided if the mov and bx were 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.
Then the list of bytes (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 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 a 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 operate on a byte array passed from CakeML as specified by the ffi parameter (see Tan et al. [2016] for details).

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

We parameterised 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 semantics. 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 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 Fail behaviour if evaluate can get stuck at an Error. We assume Error is not encountered; type soundness and sound and complete type inference ensure we can still prove our top-level correctness theorem.

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 by 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 outside the generated code, then evaluate checks whether it 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 entry point is found and bytes are read from memory and passed to call__FFI, which simulates the foreign call, then ffi_interfer updates the machine state.

### 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 must reason about the target-parameterised machine_sem.
apply_oracle oracle \( x = (\text{oracle } 0 \ x, (\lambda n.\, \text{oracle } (n + 1))) \)

evaluate config ffi k ms =
if \( k = 0 \) then (TimeOut, ms, ffi)
else if \( \text{config.target.get\_pc} \) ms \( \in \) \( \text{config.prog\_addresses} \) then
let \( ms_{\text{1}} = \text{config.target.next} \) ms; \( (ms_{\text{2}}, \text{new_oracle}) = \) apply_oracle config.next_interfer ms
in
if \( \text{config.target.state\_ok} \) ms \&\& \( \text{config.target.state\_ok} \) ms_{\text{1}} \&\& \( \text{config.target.state\_ok} \) ms_{\text{2}} then
evaluate (config with next_interfer := new_oracle) ffi (k - 1) ms_{\text{2}}
else (Error, ms, ffi)
else if \( \text{config.target.get\_pc} \) ms = \( \text{config.halt\_pc} \) 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 \( \Rightarrow (\text{Error, ms, ffi}) \)
| Some ffi_index \( \Rightarrow \)
case read_byte_array config ms of
None \( \Rightarrow (\text{Error, ms, ffi}) \)
| Some bytes \( \Rightarrow \)
let \( (\text{new ffi}, \text{new bytes}) = \) call_FFI ffi ffi_index bytes;
\( (ms_{\text{1}}, \text{new_oracle}) = \) apply_oracle config ffi_interfer (ffi_index, new_bytes, ms)
in
case new ffi.final_event of
None \( \Rightarrow \) evaluate (config with ffi_interfer := new_oracle) new ffi (k - 1) ms_{\text{1}}
| Some final_event \( \Rightarrow (\) Halt (FFI\_outcome final\_event), ms, new ffi \()

machine_sem config st ms (Terminate t io_list) \( \iff \exists k\) ms' st'. evaluate config st k ms = (Halt t, ms', st') \land st'.io\_events = io_list
machine_sem config st ms (Diverge io_trace) \( \iff (\forall k.\) ms' st'. evaluate config st k ms = (TimeOut, ms', st') \land st'.final_event = None \) \land
io_trace = \( \forall k. \) fromList (snd (snd (evaluate config st k ms))).io_events
machine_sem config st ms Fail \( \iff \exists k.\) 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.

function. In order to reason about machine_sem in a target-neutral manner, we package up all target-specific correctness criteria into a property called \( \text{asm\_to\_target\_correct} \). We prove \( \text{asm\_to\_target\_correct} \) for each supported target architecture, as described in Section 7.

The \( \text{asm\_to\_target\_correct} \) property is used in the target-neutral compiler proofs to establish that the target-specific encode functions only produce machine code such that repeated applications of the target’s next-state function implement the target-neutral ASM language, even in the presence of interference from the next_oracle.

The formal definition of \( \text{asm\_to\_target\_correct} \) is shown in Figure 6 and an informal illustration is shown in Figure 7. In both, \( \text{asm\_step} \) captures the next-state semantics of ASM, and \( \text{target\_state\_rel} \) relates ASM states to machine states. The definition of \( \text{asm\_to\_target\_correct} \) assumes that the interference oracle is well-behaved with respect to CakeML’s part of the state (interference_ok); in particular, each application of an interfering function \( \text{env} \) 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:

Here \( \text{env}_i \) represents the intra-instruction side effects of an exception handler. Handlers for hardware interrupts save process state upon invocation, which is restored before returning control back to a process. We aim to minimise the range of \( t.\text{proj} \), so as to permit as many side effects (\( ms_{i+1} \) states) as is realistic. Section 6 shows the definition of \( t.\text{proj} \) for ARMv6.

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. These specifications have been

2 The ISA models used in CakeML can be found under the HOL4 directory examples/l3-machine-code
At present, the ARMv8 model has not been fully validated. The instructions were not developed with CakeML in mind and have not been influenced by the design of the compiler. 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.

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

- `next : state → state`
- `fetch : state → code`
- `decode : code → instruction`
- `run : instruction → state → state`
- `encode : instruction → code`

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).

**Underspecification** 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 \((t : (\alpha, \beta, \gamma) \text{ target})\). These target records are defined in separate HOL4 scripts, one for each target. Currently each of the next state functions \(t.\text{next}\) 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 \(t.\text{get}_\text{pc}\), \(t.\text{get}_\text{reg}\) and \(t.\text{get}_\text{byte}\) 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.\text{REG RName}_\text{PC}`
- `arm6.get_reg s n = s.\text{REG (R_mode s.CPSR.M (n2w n))}`
- `arm6.get_byte s = s.\text{mem}`

---

\(s_1\) \(\xrightarrow{t.\text{next}}\) \(ms\) \(\xrightarrow{\text{env}_0}\) \(ms_{s_1}\) \(\xrightarrow{t.\text{next}}\) \(ms\) \(\xrightarrow{\text{env}_1}\) \(ms_{s_1}\) \(\xrightarrow{t.\text{next}}\) \(ms\) \(\xrightarrow{\text{env}_2}\) \(ms_{s_1}\) \(\xrightarrow{\text{env}_3}\) \(ms_{n+1}\)

**Figure 7.** Commuting diagram for target correctness. Machine-states marked with \(*\) must satisfy the property \(t.\text{state}_{\text{ok}}\) 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).
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. The program counter is called $\text{RName}_\text{PC}$ 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 parameterised 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 $\text{asm\_to\_target\_correct}$ 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:

\[
\begin{align*}
\text{arm6\_state\_ok} (s) & \iff \\
& \quad \text{GoodMode} \land s.\text{CPSR.M} \land \neg s.\text{CPSR.E} \land \neg s.\text{CPSR.J} \land \\
& \quad \neg s.\text{CPSR.T} \land s.\text{Architecture} = \text{ARMv6} \land \\
& \quad \neg s.\text{Extensions Extension\_Security} \land \\
& \quad s.\text{exception} = \text{NoException} \land \text{aligned} 2 \land (s.\text{REG RName}_\text{PC}) \\
\end{align*}
\]

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; 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. 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:

\[
\begin{align*}
\text{arm6\_proj} (d) s & = \\
& (s.\text{CPSR.M}, s.\text{Architecture}, s.\text{Extensions}, s.\text{exception}, \\
& s.\text{REG} \circ R\_mode, s.\text{CPSR.M}, \text{fun2set} (s.\text{mem}, d)) \\
\end{align*}
\]

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 $s.\text{REG} \circ R\_mode$ and $s.\text{CPSR.M}$ represent the general purpose registers that are visible in the current processor mode.

The current Program Status Register ($s.\text{CPSR}$) is projected out in full. In particular, this ensures that the arithmetic/logic flags are preserved (e.g. the carry flag $s.\text{CPSR.C}$, as these are used by some ARM instructions.

**ASM Configuration** The components of record $t$.config were described in Section 2. The basic configuration values for each architecture are shown 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

\[
\begin{align*}
& \text{JumpCmp } cmp \ r \ (\text{Imm} \ i) \ a \\
\end{align*}
\]

compares register $r$ with immediate value $i$ using $\text{cmp}$ (e.g. $\text{cmp}$ 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.

\[
\begin{align*}
& \text{arm6\_enc (JumpCmp } cmp \ r1 \ (\text{Reg} \ r2) \ a) = \\
& \text{let } (opc,c) = \text{arm6\_cmp } cmp \\
& \text{in } \\
& \text{arm6\_encode } 14w \\
& \quad | (\text{Data} \\
& \quad \quad \text{(TestCompareRegister} \\
& \quad \quad \quad (opc, n2w \ r1, n2w r2, SRType\_LSL(0))))) @ \\
& \quad \text{arm6\_encode } c \ (\text{Branch} (\text{BranchTarget} (a - 12w))) \\
\end{align*}
\]

The function $\text{arm6\_encode}$ 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 $\text{EncodeARMImmediate}$ (defined in L3) attempts to encode a 32-bit immediate value using a 12-bit representation and the function $\text{arm6\_cmp}$ picks an appropriate condition code, as well as a test or comparison operation.

The function $\text{arm6\_enc}$ can be evaluated within the HOL4 logic, e.g. one can prove

\[
\vdash \text{arm6\_enc (JumpCmp Equal 1 (Imm 175w) 2048w)} = \\
[175w; 0w; 81w; 227w; 253w; 1w; 0w; 10w],
\]

which corresponds with the ARM code.

---

5 The 5-bit component $s.\text{CPSR.M}$ encodes the current operating mode, e.g. it will have value sixteen when running in user mode. The function $(n2w : \text{num} \to \text{a word})$ maps natural numbers to words (modulo $2^8$).

6 The $s.\text{Architecture}$ and $s.\text{Extensions}$ 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.
<table>
<thead>
<tr>
<th>Word size</th>
<th>ARMv6</th>
<th>ARMv8</th>
<th>MIPS-64</th>
<th>RISC-V</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Endianness</th>
<th>Little</th>
<th>Little</th>
<th>Big</th>
<th>Little</th>
<th>Little</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code alignment (bytes)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Number of registers</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>Link register</td>
<td>14</td>
<td>30</td>
<td>31</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>“Avoid” registers</td>
<td>13, 15</td>
<td>31</td>
<td>0, 1, 25 – 29</td>
<td>0, 2, 3, 31</td>
<td>4, 5</td>
</tr>
<tr>
<td>Two register binary-ops</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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 4B BB F0 DE BC 9A 78 56 34 12 are both valid x86-64 codes (for `nop` and `mov rdx, 0x123456789abcdef0` respectively).

<table>
<thead>
<tr>
<th>Location</th>
<th>Memory</th>
<th>Jump</th>
<th>Conditional Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv6</td>
<td>-0xFFF7 - 0x10007</td>
<td>-0xFFF - 0xFFF</td>
<td>-0x1FFFFFFF8 - 0x2000007</td>
</tr>
<tr>
<td>ARMv8</td>
<td>-0x80000 - 0x7FFFFF</td>
<td>-0x100 - 0xFF</td>
<td>-0x80000000 - 0x7FFFFFFF</td>
</tr>
<tr>
<td>MIPS-64</td>
<td>-0x7FFFFF - 0x80000000</td>
<td>-0x8000 - 0x7FFFFF</td>
<td>-0x80000 - 0x7FFFF</td>
</tr>
<tr>
<td>RISC-V</td>
<td>-0x80000000 - 0x7FFFFF</td>
<td>-0x8000 - 0x7FFFFF</td>
<td>-0x80000 - 0x7FFFFF</td>
</tr>
<tr>
<td>x86-64</td>
<td>-0x7FFFFFFF - 0x800000000000</td>
<td>-0x80000000 - 0x7FFFFFFF</td>
<td>-0x800000000004 - 0x7FFFFFFF</td>
</tr>
</tbody>
</table>

Immediate values (for binary operations)

<table>
<thead>
<tr>
<th>Location</th>
<th>Memory</th>
<th>Jump</th>
<th>Conditional Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv6</td>
<td>12-bit encoding (an 8-bit value is rotated right by a 4-bit value)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARMv8</td>
<td>Arithmetic 12-bit immediate, optionally shifted left 12 places</td>
<td>Logical Bit run-length encoding (specified algorithmically)</td>
<td></td>
</tr>
<tr>
<td>MIPS-64</td>
<td>Subtract -0x7FFFFF - 0x7FFFFF (encoded using DADDIU)</td>
<td>Arithmetic -0x8000 - 0x7FFFFF</td>
<td></td>
</tr>
<tr>
<td>RISC-V</td>
<td>Subtract -0x7FF - 0x7FF (encoded using ADDI)</td>
<td>Logical 0 - 0x7FFFF</td>
<td></td>
</tr>
<tr>
<td>x86-64</td>
<td>-0x8000 - 0x7FFFFF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

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

Note that the function EncodeARMImmediate could fail to produce a 12-bit encoding imm12 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 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
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; DSSL; ORI; DSSL; 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; BEQ | 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 ]  
ARMv6  [ ORI; BEQ | BLTU | BLT | BNE | BGEU | BGE ] [ ANDI; BEQ | BNE ]  
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>
<thead>
<tr>
<th>Machine instruction</th>
<th>ARMv6</th>
<th>ARMv8</th>
<th>MIPS-64</th>
<th>RISC-V</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVZ</td>
<td>MOVW</td>
<td>ORR</td>
<td>MOVK</td>
<td>MOVK</td>
<td>MOVK</td>
</tr>
<tr>
<td>ADDIU</td>
<td>LUI</td>
<td>XORI</td>
<td>LUI</td>
<td>ORI</td>
<td>DSSL</td>
</tr>
<tr>
<td>MOV</td>
<td>MOVZ</td>
<td>MOVW</td>
<td>ORR</td>
<td>MOVK</td>
<td>MOVK</td>
</tr>
<tr>
<td>SLTU</td>
<td>DADDU</td>
<td>SLTU</td>
<td>DADDU</td>
<td>SLTU</td>
<td>OR</td>
</tr>
<tr>
<td>SLTU</td>
<td>ADD</td>
<td>SLTU</td>
<td>ADD</td>
<td>SLTU</td>
<td>OR</td>
</tr>
<tr>
<td>CMP</td>
<td>CMNEQ</td>
<td>ADCS</td>
<td>MOVCC</td>
<td>MOVCS</td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>CCMN</td>
<td>ADCS</td>
<td>MOV</td>
<td>ADC</td>
<td></td>
</tr>
<tr>
<td>SLTU</td>
<td>SLTU</td>
<td>SLTU</td>
<td>ADD</td>
<td>SLTU</td>
<td>OR</td>
</tr>
<tr>
<td>SLTU</td>
<td>SLTU</td>
<td>SLTU</td>
<td>SLTU</td>
<td>SLTU</td>
<td>SLTU</td>
</tr>
<tr>
<td>CMP</td>
<td>JE</td>
<td>JB</td>
<td>JL</td>
<td>JNE</td>
<td>JN</td>
</tr>
</tbody>
</table>

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.

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 instructions (employing left shifts) are used. The sign-extension 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 r1 r2 r3 r4 treats register r1 as a carry flag. This instruction first computes

\[ r = R_2 + R_3 + \text{if} \; R_4 = 0 \; \text{then} \; 0 \; \text{else} \; 1 \]

\( (R_i \in \mathbb{N} \) is the 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 occurring 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)
```

\(^3\)Having data (an immediate) within a code segment is not ideal with modern Harvard architectures, so this encoding choice may be changed.
daddiu $2, $1, 0 ; 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, 0x0c ; if r5 < r31 then pc := pc + 12
-- JumpCmp Less 5 (Imm 4w) 0x8000w --
ori t6, $0, 0x4 ; r31 := 4
bge t0, t6, 0x08 ; 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 r1 transfers the address pc + SignExtend (i < 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 − 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 DADDI 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 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 in Figure[7] below 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 s1, s2, an ASM instruction i and a target machine state m, with:

   `asm_step t.config s1 i s2 and target_state_rel t s1 m`

   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 s2 is related to s1 by a single application of the ASM next state function. The state m satisfies 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 s2 becomes expressed in terms of register and memory updates to s1.

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 ∈ pcs, and an instance of target_state_rel t s₂ ms', where ms' is the machine state after running \( n + 1 \) instructions with interleaved interference from \( \text{env} \).

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 \( \text{env} \). 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 na"ve 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 \( \text{env} \) 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. \( a \cdot 3 \) would denote bit three of an ASM address \( a \). The output of the step tool is a step theorem of the form

\[ A_0, \ldots, A_n \vdash \text{next}(s_0) = s_1 \]

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 \( \text{e1a00000} \) (ARMv6), \( \text{d503201f} \) (ARMv8), \( \text{00000000} \) (MIPS-64), \( \text{00000013} \) (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 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 $r_1 \neq r_3 \land r_1 \neq r_4$ for AddCarry was established during the course of the verification.

8. Compiler Validation

The final step in using the CakeML compiler is to set up an environment for the compiled code to run. We use a small (unverified) assembly program specific to each target for this purpose. Briefly, this program:

1. Provides CakeML with heap and stack space
2. Sets up appropriate jump targets for external CakeML calls (FFI and system calls) assumed by the compiler
3. Initialises four registers to point to: the entry address for CakeML code, the first address of the heap, the first address of the stack, and the first address past the end of the stack
4. Jumps to the CakeML entry address

Although CakeML uses its own stack internally, it still needs to obey target-specific calling conventions for external calls. Most importantly, we ensure that the return address register which CakeML uses internally maps to the target’s link register. We also ensure that CakeML code never clobbers important global state, e.g. the stack pointer on all architectures, the base pointer on x86-64, and the global pointer on RISC-V. Both of these are achieved by tweaking the compiler’s register name mapping described in Section 2 and (Tan et al. 2016). For the latter requirement, we also restrict CakeML from using certain registers by adding to the “avoid” registers for each target (see Table 1).

To validate these assembly wrappers, we ran a suite of CakeML compiler benchmarks on all five targets (an emulator was used for RISC-V and MIPS-64). All benchmarks compiled and executed successfully, except a few compilations to MIPS-64. The compiler failed to compile some programs to MIPS-64 because the CakeML assembler produced jump offsets that were too large. At the time of writing, we are looking to improve the MIPS-64 encoder.

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.

Acknowledgments The first author was partially supported by EPSRC Programme Grant EP/K008528/1, UK. The second author was partially supported by the Swedish Research Council, Sweden.
References


