A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture

Anthony Fox and Magnus O. Myreen

Computer Laboratory, University of Cambridge, UK

Abstract. This paper presents a new HOL4 formalization of the current ARM instruction set architecture, ARMv7. This is a modern RISC architecture with many advanced features. The formalization is detailed and extensive. Considerable tool support has been developed, with the goal of making the model accessible and easy to work with. The model and supporting tools are publicly available – we wish to encourage others to make use of this resource. This paper explains our monadic specification approach and gives some details of the endeavours that have been made to ensure that the sizeable model is valid and trustworthy. A novel and efficient testing approach has been developed, based on automated forward proof and communication with ARM development boards.

1 Introduction

Instruction set architectures (ISAs) provide a precise interface between hardware (microprocessors) and software (high-level code). Formal models of instruction sets are pivotal when verifying computer micro-architectures and compilers. There are also areas where it is appropriate to reason directly about low-level machine code, e.g. in verifying operating systems, embedded code and device drivers. Detailed architecture models have also been used in formalizing multiprocessor memory models, see [15].

In the past, academic work has frequently examined the pedagogical DLX architecture, see [6]. When compared to commercial architectures, DLX is relatively uncomplicated – being a simplified version of the MIPS RISC architecture. Consequently, it is rarely cumbersome to work with the entire DLX instruction set. More recently there has been a move to modelling, and working with, commercial instruction sets (possibly in a reduced form). This has been motivated by a desire to carry out demonstrably realistic case studies, showing that various techniques scale and are not purely "academic" in nature. Common commercial architectures include: IA-32, x86-64, PowerPC, Sparc and ARM. The ARM architecture is ubiquitous in low-powered (mobile and embedded) computing devices – the latest version of the architecture, dubbed ARMv7, is implemented by, for example, the Cortex-A8 processor.

There are many challenges when working with full-blown ISAs, these include: (*i*) official reference manuals are large, stretching to many hundreds of pages – one can easily overlook subtle details or become bogged down with "uninteresting" background information; (*ii*) official descriptions are semi-formal (ambiguous); (*iii*) many details are *implementation dependent* or *unpredictable*; (*iv*)

M. Kaufmann and L. Paulson (Eds.): ITP 2010, LNCS 6172, pp. 243–258, 2010.

architectures frequently have multiple generations, versions and optional extensions; and (*v*) large formalizations can stretch the capabilities of interactive theorem provers. Most importantly: how can one be sure that the formalization does not contain bugs? The scale and complexity is such that it is not possible to eradicate all errors by simply eyeballing the specification or examining a few instructions. This paper discusses our experiences with constructing and validating a complete model of the ARMv7 architecture using the HOL4 proof system [16].

2 Approach

Some key aspects of the work presented here are:

- The ARM instruction set architecture has been modelled in HOL using a monadic style. This approach has a number of advantages, which are discussed in Section 3.
- The model is extensive and detailed – it covers all architecture versions currently supported by ARM, including full support for Thumb-2 instructions.[1]
- A collection of tools have been built around the model, making it accessible and easy to work with. This includes an assembler and disassembler, both of which are implemented in Standard ML. There is also a tool for automatically extracting the semantics of a single instruction from the model: this is implemented through evaluation (forward proof) and is discussed in Section 4.
- A distinction is made between entities that are defined or derived inside of the HOL logic and those that reside outside – this is illustrated in Figure 1. It is important that everything defined inside of the logic is valid. On the other hand, although it is advantageous that the other tools (e.g. the parser and encoder) are bug free, these are not fundamentally relied upon in formal verification work.
- The model operates at the *machine code* level, as opposed to a more abstract assembly code level. In particular, the model does not provide assembly level “pseudo instructions” and instruction decoding is explicitly modelled inside the logic. This means that the the model can be directly validated by comparing results against the behaviour of hardware employing ARM processors – this is discussed in Section 5.

Through the use of extensive validation, trust in the model is progressively established. An efficient testing framework has been developed, which is based on a HOL session communicating with an ARM development board (via a serial port). This set-up is required because most PCs are based on x86 processors and cannot natively run ARM code. The results from this testing are discussed in Section 5.3. Due to space constraints, precise details of the ARM architecture are not provided here, readers are instead referred to [4] and [11].

[1] It does not cover the ARMv7-M profile (which has a slightly different programmer’s model) or the ThumbEE, Jazelle, VFP and Advanced SIMD extensions.

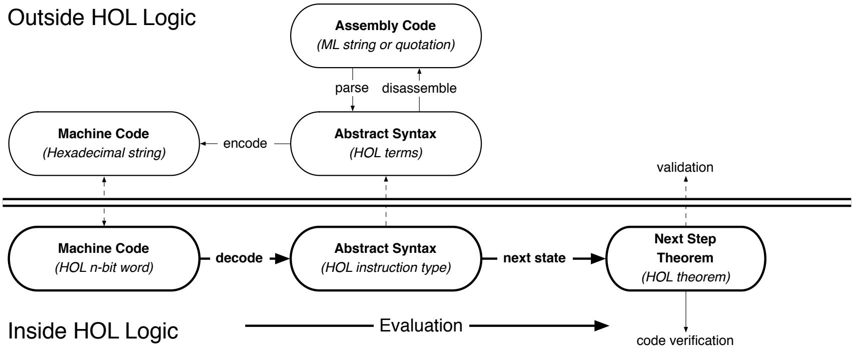

Fig. 1. Overall structure of the formalization

3 Monadic Specification

The HOL4 system provides built-in support for defining recursive, total functions (see [17]). Consequently, formal specifications can be written in a functional programming style using syntax roughly similar to that of ML. For example, in the HOL4 model of the ARMv4 architecture (see [3]), a typical definition is of the following form:

$$\begin{aligned} f(v_1,\ldots,v_m) = \; & \text{let } x_1 = g_1(\ldots) \text{ in} \\ & \ldots \\ & \text{let } x_n = g_n(\ldots) \text{ in} \\ & \quad (w_1,\ldots,w_m) \end{aligned}$$

If f defines the semantics of a machine code instruction then the vector v would represent the components of the *programmer's model* state space (for example, machine registers) and w specifies the next state. The variables x_i are intermediate computations; typically the result of accessing, updating and manipulating state components. There are a few problems with this particular style of specification:

- Explicitly naming state component (splitting the state into a vector) can make it harder to make global changes to sizeable specifications, e.g. adding, removing or changing the type of state components.
- The semantics is rigidly fixed to that of a state transformer. In particular, it is not possible to reason about the order of intermediate computations, observing whether or not they were performed sequentially or in parallel.
- For those more familiar with imperative code, the specification is not especially readable.
- It is not obvious how to handle memory I/O, non-determinism or "error states".

All of these factors motivate the use of a monadic programming style (see [18]), where computations themselves are represented with an abstract data type.

Let **M** represent a monad type constructor. The two fundamental monad operations are *return* and *bind*, represented by:

$$\text{return} : \alpha \to \alpha\,\mathbf{M} \quad \text{and} \quad \gg\!= \; : \alpha\,\mathbf{M} \to (\alpha \to \beta\,\mathbf{M}) \to \beta\,\mathbf{M}$$

respectively. The return operation gives a value to a computation. The bind operation composes computations: it takes the result of one computation and passes it on to another. The type variables α and β represent the types for working values in a computation: these roughly correspond with the variables and arguments of procedures and functions in an imperative language. The monad type constructor **M**, and associated primitive operations, can be defined in any number of ways, implementing various underlying computational models – this can loosely correspond with defining a semantics (and run-time environment) for a given programming language.

In addition to the two primitive monad operations, our specifications also makes use of a parallel operation:

$$||| : \alpha\,\mathbf{M} \to \beta\,\mathbf{M} \to (\alpha \times \beta)\,\mathbf{M}\ \ .$$

This performs two operations, but without imposing an order of evaluation.

3.1 Sequential Monad

It is possible to define monads in HOL without considering concrete implementations: one could, for example, provide an axiomatic formalization. However, there are advantages to being able to actually carry out computations with the model (see Section 4). This section presents a *sequential* monad – this has formed the primary basis for working with the ARM specification. The monad provides a simple *operational semantics* in a *shallow embedding* style – this is suited to evaluation and code verification with a Hoare style logic. The overall HOL specification is split into two parts: the monad specification and the instruction set specification. This means that the instruction set specification part can be interpreted by any other monad with the same interface.

The type constructor for the sequential monad is as follows:

$$\alpha\,\mathbf{M_{seq}} \equiv \text{state} \to (\alpha \times \text{state})\ \text{MaybeError}$$

where state is the programmer's model state space and

$$\beta\ \text{MaybeError} = \text{Okay of}\ \beta \mid \text{Error of string}\ \ .$$

This is essentially a state-transformer monad with error states. The monad type can be viewed as a partial map, representing a state transition with a return value. When the map is "undefined", the result is a tagged string – this can be used to identify where an error occurred.

The basic monad operations are defined as follows:

$$\text{return}\ (v)\ =\ \lambda s.\ \text{Okay}\ (v, s)\ ,$$

$$(f\ \gg=\ g) = \lambda s.\ \begin{bmatrix} \text{case}\ f(s)\ \text{of Error}\ e \rightarrow \text{Error}\ e \\ \qquad\qquad\qquad \mid \text{Okay}\ (v, s') \rightarrow g(v)(s') \end{bmatrix}$$

and

$$(f\ \mid\mid\mid\ g) = (f\ \gg=\ (\lambda v_1.\ g\ \gg=\ (\lambda v_2.\ \text{return}\ (v_1, v_2))))\ .$$

Note that errors are terminal (no further next state computation is performed) and the parallel operation simply performs computations in a left to right order.

Thanks to Michael Norrish, it is possible to use Haskell's *do-notation* when parsing and printing monadic terms in HOL4. For example, the parallel operation above is more readable when written as follows:

$$(f\ \mid\mid\mid\ g) = \textbf{do}\ v_1 \leftarrow f;\ v_2 \leftarrow g;\ \text{return}\ (v_1, v_2)\ \textbf{od}\ .$$

In addition to these operations, there is a collection of operations for accessing (reading an writing) state components. For example, registers are accessed with:

$$\text{read\_reg} : \text{iid} \rightarrow \text{bool}[4] \rightarrow \text{bool}[32]\ \mathbf{M}\ \text{and}$$
$$\text{write\_reg} : \text{iid} \rightarrow \text{bool}[4] \rightarrow \text{bool}[32] \rightarrow \text{unit}\ \mathbf{M}$$

where bool[4] is a 4-bit register index (for registers `r0`–`r15`); and bool[32] is a 32-bit register value. The type iid is used to identify threads and is of little interest here.[2] The definitions for these operations derive from pseudo-code contained within the official ARM programmer's model description (see [11]).

3.2 Instruction Set Specification

Having defined the underlying monad, one can then define the semantics of instructions. The following operation runs one instruction:

$$\text{arm\_instr} : \text{iid} \rightarrow \text{encoding} \times \text{bool}[4] \times \text{instruction} \rightarrow \text{unit}\ \mathbf{M}\ .$$

This operation takes a triple (*enc*, *cond*, *ast*), which represents the result of fetching and decoding an instruction. Instructions are conditionally run: for example, the instruction `addcs r1,r2` will have a *cond* value of 2 and it will be a no-op when the carry flag is not set. The *enc* field indicates the instruction encoding, e.g. 16-bit Thumb, 32-bit Thumb or 32-bit ARM. The behaviour of instructions is specified with various sub-operations, these are selected by pattern matching over the abstract syntax term (*ast*).

[2] It allows register and memory accesses to be tagged with the identity of the thread that made them.

As an example, consider the bit-field-insert instruction (`bfi`).[3] This is specified on page A8-49 of the ARM reference [11] with the following pseudo code:

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = R[n]<(msbit-lsbit):0>;
        // Other bits of R[d] are unchanged
    else
        UNPREDICTABLE;
```

The "encoding specific operations" part assigns values to components based on the instruction encoding. For example, with a 32-bit Thumb encoding the following applies from page A8-48:

Encoding T1 ARMv6T2, ARMv7

```
BFI<c> <Rd>,<Rn>,#<lsb>,#<width>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	(0)	1	1	0	1	1	0	Rn				0	imm3			Rd				imm2		(0)	msb				

```
if Rn == '1111' then SEE BFC;
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if BadReg(d) || n == 13 then UNPREDICTABLE;
```

The description above additionally provides the concrete syntax and encoding for this instruction, together with a list of architecture versions over which the instruction is defined. The corresponding HOL4 specification for `bfi` is:

```
⊢ bit_field_clear_insert_instr ii enc (Bit_Field_Clear_Insert msb d lsb n) =
  instruction ii "bit_field_clear_insert"
    (thumb2_support CROSS U(:ARMextensions → bool))
    (λ v.
       (if enc = Encoding_Thumb2 then
          ((d = 13w) ∨ (d = 15w)) ∨ (n = 13w)
        else
          d = 15w) ∨ w2n msb < w2n lsb)
    do
      rd ← read_reg ii d |||
      rn ← if n = 15w then return 0w else read_reg ii n;
      increment_pc ii enc |||
      write_reg ii d (bit_field_insert (w2n msb) (w2n lsb) rn rd);
      return ()
    od
```

This code simultaneously specifies the closely related bit-field-clear (`bfc`) instruction. Grouping related instructions together greatly reduces the size specification, which in turn limits the scope for introducing errors. The helper function `instruction` takes: the thread identifier; a string naming the *instruction class* (this is used to tag error states); a set representing the architectures and extensions over which the instruction is defined; a predicate that determines whether the instruction is unpredictable for a particular instruction set version; and an operation that specifies the behaviour when the instruction is defined and predictable. Together with the decoding function, this covers all aspects of the official ARM pseudo code specification.

[3] This replaces a bit range in a destination register with bits from a source register, which is implemented in HOL4 with the bit vector operation `bit_field_insert`.

Although the HOL4 specification is far from being aesthetically perfect, it is at least fairly compact and reasonably readable. More importantly, it is precise and formal. In fact, in order to be of use, the HOL4 specification is in some ways over-precise, since it specifies the order of resource accesses, as well as specifying when the program counter is updated. The ARM reference explicitly states that their pseudo code does not cover such low-level aspects of the behaviour (see page 4 of Appendix I in [11]). However, cases in which such design choices would become visible are typically designated (by ARM) as being unpredictable, so this over-specification should not be a problem at this level of abstraction.

4 Single Step Theorems: Evaluation

For the purposes of code verification and model validation, we require theorems of the form:

$$\vdash \forall s.\ \mathrm{P}(s) \Rightarrow (\mathrm{NEXT}(s) = s') \tag{1}$$

where the predicate P specifies a context (e.g. instruction to be run); and the function NEXT : state → state option defines the next state behaviour for the architecture with respect to the sequential monad. Such *single step* theorems are needed, for example, to generate Hoare triples for every op-code encountered in the machine code of a program being verified. How can such theorems be derived "on-demand" from the monadic specification? First it is necessary to define a monad operation next : unit **M**, which calls `arm_instr` using the result of fetching and decoding an instruction.[4] The following definition is then possible:

$$\mathrm{NEXT}(s) = \left[\begin{array}{l}\text{case next}(s) \text{ of Error } \_ \rightarrow \text{NONE} \\ \qquad\qquad\qquad \mid \text{Okay } ((), s') \rightarrow \text{SOME } s')\end{array}\right] .$$

It is possible to derive Equation 1 by directly expanding function definitions using the HOL4 simplifier, which supports contextual rewriting.[5] Unfortunately, the simplifier is fairly slow and this is a significant problem when working with such a large model. There is a *much* faster call-by-value rewrite engine (`EVAL`), but this does not directly support contextual rewriting. To get around this limitation, the following theorem (which is proved once and for all) is used:

$$\begin{aligned}
&\vdash \forall s\, x\, h\, g\, P. \\
&\quad (\forall i.\ P(i) \Rightarrow (g(i) = i)) \wedge && (2)\\
&\quad (\mathrm{next}(g(s)) = \mathrm{Okay}\ ((), x)) \wedge && (3)\\
&\quad (P(s) \Rightarrow (h(g(s)) = x)) && (4)\\
&\quad \Rightarrow \\
&\quad (P(s) \Rightarrow (\mathrm{NEXT}(s) = \mathrm{SOME}\ (h(s))))\ . && (5)
\end{aligned}$$

A tool for deriving single step theorems "on-the-fly" works in stages as follows:

[4] For simplicity sake, non-pipelined operation is assumed here.

[5] Some infrastructure is needed to intelligently expand the context, i.e. generate the predicate P.

- The user supplies an instruction op-code, together with some other context, e.g. the current architecture version, processor mode and instruction set (Thumb or ARM).
- The tool examines the op-code (decodes it) and constructs a custom context predicate P and corresponding function g. Equation 2 is proved to hold. The context predicate and function must ensure that the supplied instruction runs successfully. Amongst other things, this includes avoiding error states, e.g. ensuring that memory addresses are suitably aligned and things like division by zero do not occur. This stage can be completed fairly quickly.
- The next state operation is evaluated for the initial "context" state $g(s)$, giving Equation 3. The rewriter (`EVAL`) is provided with many lemmas to ensure that the evaluation proceeds properly, i.e. making sure that error states are actually avoided. It is also necessary to restrict evaluation from proceeding too far, e.g. expanding with the definition of bit vector operations.
- The terms representing the states $g(s)$ and x are compared and a function h is constructed. Equation 4 is proved to hold.
- The consequent, Equation 5, is derived by *modus ponens* using the general theorem (above) and the three generated theorems (Equations 2–3). This is a simple application of the `MATCH_MP` rule in HOL4. Finally, simplifications are applied and the resulting single step theorem is returned to the user.

Most of the effort with this approach went into automating the construction of P and g, and proving appropriate evaluation lemmas. For example, consider the instruction `eor pc,r1,r2,asr #2`.[6] To be predictable in ARMv7, we require a context P containing $(r_1 \oplus (r_2 \gg 2))[1:0] \neq 2$. This is achieved by defining g such that

$$r_1 \mapsto \textbf{if } (r_1 \oplus (r_2 \gg 2))[1:0] \neq 2 \textbf{ then } r_1 \textbf{ else } r_2 \gg 2$$

and, during the next state evaluation, using the *general* lemma

$$\vdash \forall x\, y.\ ((\textbf{if } (x \oplus y)[1:0] \neq 2 \textbf{ then } x \textbf{ else } y) \oplus y)[1:0] \neq 2$$

which holds because $y \oplus y = 0$ and $0[1:0] \neq 2$. Thus, by applying function g, we successfully satisfy P and avoid the error case – the evaluation proceeds automatically as required. Covering all such cases, over all instructions and architecture versions, was an arduous undertaking. However, the resulting instruction evaluator is relatively fast (see below) and the internal complexities are invisible to the user.

The final simplification stage provides a canonical form for state accesses and updates. For example, registers are read and written using the following functions:

$$\begin{aligned}\text{ARM\_READ\_REG} &: \text{bool}[4] \rightarrow \text{state} \rightarrow \text{bool}[32] \text{ and}\\ \text{ARM\_WRITE\_REG} &: \text{bool}[4] \rightarrow \text{bool}[32] \rightarrow \text{state} \rightarrow \text{state}\ .\end{aligned}$$

[6] In ARMv7 this instruction performs a *branch with exchange* to the target address $r_1 \oplus (r_2 \gg 2)$, where $\oplus$ is exclusive-or and $\gg$ is signed right shift. The "exchange" part relates to switching between Thumb and ARM code. The behaviour is different for ARMv6 and different again for all earlier versions – the tool is aware of this.

The theorem below is derived from the single step theorem for 32-bit Thumb op-code `F362 01C7` (`bfi r1,r2,#3,#5`):

```
⊢ Abbrev (pc = ARM_READ_REG 15w state) ∧ Abbrev (rd = ARM_READ_REG 1w state) ∧
  Abbrev (rn = ARM_READ_REG 2w state) ⇒
  (ARM_ARCH state = ARMv7_A) ∧ ... ∧ aligned (pc,2) ∧
  (ARM_READ_MEM (pc + 3w) state = 1w) ∧ (ARM_READ_MEM (pc + 2w) state = 199w) ∧
  (ARM_READ_MEM (pc + 1w) state = 243w) ∧ (ARM_READ_MEM pc state = 98w) ⇒
  (ARM_NEXT state = SOME
     (ARM_WRITE_MEM_READ (pc + 3w) (ARM_WRITE_MEM_READ (pc + 2w)
           (ARM_WRITE_MEM_READ (pc + 1w) (ARM_WRITE_MEM_READ pc
                 (ARM_WRITE_REG 1w (bit_field_insert 7 3 rn rd)
```

The first two lines show some *abbreviations* – these have been added here to aid readability. Note that these theorems are not really designed for human consumption – instead, they provide raw input to other automated tools. Observe that memory accesses (from fetching the instruction) have been recorded with `ARM_WRITE_MEM_READ`. This example took around 0.9 s to run,[7] which is approximately the same time that it takes to perform full ground-term evaluation.

5 Validation

Our ARM model formalizes a substantial part of the 2000-page ARM reference manual. As a result, the specification is very large and detailed. The ARM model is sufficiently complex that mistakes are very hard to avoid and very hard to discover. How do we know that our model correctly describes the execution of ARM instructions on ARM processors? Furthermore, if there are mistakes in the model, how do we find them?

Our solution is a validation infrastructure that allows us to compare the execution of ARM instructions in our model with their execution on real ARM hardware. This infrastructure consists of a mixture of ML, C and custom assembly programs, together with the hardware used to run machine code on ARM processors. The following ARM development boards have been used:

Olimex LPC-2129P board, with a comparatively old ARM7TDMI-S core;
Atmel SAM3U-EK board, with a "lightweight" ARM Cortex-M3 core; and
Texas Instruments BeagleBoard, with a "heavyweight" ARM Cortex-A8 core.

The majority of the testing has been performed on the BeagleBoard, which supports the latest architecture version, namely ARMv7-A.

5.1 Random Testing

Our principle validation approach is based on generating large test suites of randomly generated instructions. The generator is designed to provide broad coverage over the ARM and Thumb instruction spaces. The number of instruction instances is sufficiently large that it is not feasible to manually achieve such

[7] For HOL4 (experimental kernel) under Poly/ML with a Pentium 4, 3.0 GHz and 2 GB of RAM.

wide coverage in a reliable fashion. However, in some cases a custom test suit is used, which may include manually selected op-codes. This helps speed up the testing process when examining op-codes that are currently deemed “of interest” or that require “special treatment”.

Although substantial software engineering effort went into writing this validation infrastructure, its top-level functionality is conceptually simple:

Step 1: Instruction selection and evaluation.

Instructions are generated by randomly choosing *valid* abstract syntax terms, representing instructions of a given kind, e.g. data-processing, branch, load, store or other. These terms are then encoded into 16-bit or 32-bit instruction op-codes and the ARM model is evaluated for each concrete instruction encoding, i.e. we calculate the step theorem described in Section 4. We ignore instructions for which the model returns “unpredictable”.

Step 2: Installing test code onto an ARM board.

With some boards, installation of new programs requires physically removing and inserting jumpers on the boards. (The reason for this is that the boards are implemented as a Harvard architecture, i.e. programs cannot alter themselves or install new code.) Consequently, human interaction is sometimes required, and instructions are generated and tested in batches.

Step 3: Random input generation.

Once the boards have the correct batch of tests installed, test cases can be sent across the serial cable. We generate random inputs for all registers that are, according to the model, relevant for this instruction. In order to increase the chances of hitting corner cases such as “result equals zero”, each input is chosen, by a fixed probability, to be one of the following constants:

```
0  1  2  01010101  00FF00FF  FF00FF00  FFFFFFFE  FFFFFFFF
```

otherwise input is chosen uniformly from the set of 32-bit numbers.

Step 4: Sending input via serial cable, waiting for reply.

Input is sent to the boards as strings, e.g. the following echo command will tell the board to test instruction `1917F303` on inputs `20000000 FF00FF00 9E466F33`, if the board is listening to serial port `ttyS0`:

```
echo "1917F303 20000000 FF00FF00 9E466F33" > /dev/ttyS0
```

(To save space, this example omits showing all other register values.) The tester program on the board: reads this input; finds the right instruction to execute; sets up the state; executes the instruction; saves the state and sends back the following output, which can be read from ML as a normal text file at location `/dev/ttyS0`:

```
instruction: 1917F303
input: 20000000 FF00FF00 9E466F33
output: 28000000 FF00FF00 FF800000
```

Programming the board software was by far the hardest part of the validation effort.

Step 5: Validating results against the model.

Once the board has responded to the input, the instruction's step theorem is instantiated and evaluated using the concrete values for the input line (as shown above). If the test results disagree with the model then a failure is reported in a log file, e.g. the following log entry records a genuine error in the first revision of our ARM model. Here the values of the flag register `cpsr` and register `r9` differ from their expected values.

```
FAIL:  1917F303  ARCH ARMv7-M THUMB ssat r9,#24,r3,lsl #4
 resource:   cpsr     r3       r9
 input:      20000000 FF00FF00 9E466F33
 board.out:  28000000 FF00FF00 FF800000
 model.out:  20000000 FF00FF00 00800000
 diff:       ^^^^^^^^          ^^^^^^^^
```

The cause of this error was a minor misinterpretation of the ARM manual.

Step 6: Repeat from Step 3.

By looping through Steps 3–6, we get through five tests per second on average. This speed is achieved by only once evaluating the full ARM model symbolically for each instruction (Step 1) and then in the test loop (Steps 3–6) evaluating only fully instantiated terms, which is relatively fast in HOL. The overall performance is limited by communication speeds.

5.2 Other Means of Gaining Assurance

There are other means of gaining assurance that the model is correct. For example, we gain some assurance that the model cannot be completely wrong from:

- Observing that code verified against this model (see [13]) seems to behave as expected when executed on real hardware.
- Running the model over ARM code that calculates a non-trivial known function, e.g. MD5. For example, a reference C implementation of MD5 (see [14]) was cross-compiled to ARM machine code using GCC. This was then run on an SML version of the HOL model, which was generated using Konrad Slind's EmitML tool. This approach sacrifices trust (using the LCF approach) for performance – running a few thousand ARM instructions per second.

However, both of these approaches are inferior to the testing described above, since these approaches have smaller coverage of the instruction space and make finding the source of erroneous output very complicated.

5.3 Test Results

Comparing the execution of instructions on hardware to evaluations of the ARM model has been a successful method for both quickly finding bugs in the model and as a means of gaining evidence that the HOL definitions are, if not completely accurate, very close to exactly right. At the time of writing the testing coverage is good but not yet complete. Progress and tests are recorded at `www.cl.cam.ac.uk/~mom22/arm-tests`. This should allow others to benefit from, and independently assess, this work.

The following bugs were found using the approach described in Section 5.1.

1. Bit-field insert (`BFI`): the following update should occur
   `Rd<msb:lsb>` ← `Rn<(msb-lsb):0>`
   but instead the following was occurring
   `Rd<msb:lsb>` ← `Rn<msb:lsb>` .
2. Signed saturates (`SSAT` and `SSAT16`): there was a missing application of a sign-extension function.
3. 16-bit signed saturate (again) and an assortment of signed multiples (`SSAT16`, `SMLA<x><y>`, `SMUL<x><y>`, `SMLAW<x><y>`, `SMULW<x><y>` and `SMLAL<x><y>`): sign extension was not working properly because the bit vector operation `word_bits` was being used instead of `word_extract`.
4. The 32-bit Thumb versions of load signed half-word and load signed byte (`LDRSH` and `LDRSB`): these were incorrectly decoded (flag values were being extracted from the wrong bit position).

In each of the cases listed above there was a clear discrepancy between the "real" register output values and those obtained through evaluating the model. In addition to these bugs it also became clear that the 32-bit Thumb register shift instructions (`LSR`, `ASR`, `LSL` and `ROR`) were not being tested. This was because the model was incorrectly identifying them as being unpredictable. It later transpired that there was also a bug in decoding these instruction.[8]

Finally, a bug was found through the MD5 example mentioned in Section 5.2. The condition test was wrong for the greater-than (`GT`) and less-than or equal (`LE`) conditions: the carry flag (`C`) was being used instead of the zero flag (`Z`).[9]

As an unforeseen consequence of this project, it has has been possible for us to identify and report bugs in the GNU assembler (`gas`). These mostly concern Thumb-2 support in versions 2.19 and 2.20. That is to say, there were errors in the binary encoding of `SEV.W`, `PKHTB`, `QADD`, `QDADD`, `QSUB` and `QDSUB`. The reported bugs are documented at `sourceware.org/bugzilla/` under bug numbers 10168, 10169, 10186 and 11013.

6 Restrictions

There are some limitations to our approach. We have not found a *clean* way to simultaneously consider multiple monadic interpretations of the specification in HOL4. This has not been a problem for our work, where we focus on the sequential semantics, but we speculate that some kind of module system (such as Locales in Isabelle or Sections in Coq) could be helpful here. The testing framework has proved to be very successful, however, note that:

[8] A bit vector extract was off by one position.

[9] This bug was fixed before the random testing covered conditional instructions.

- Store instructions require special treatment.
- Care must be taken with instructions that access or update the program counter or stack pointer (registers fifteen or thirteen). The random instruction generator normally avoids these instructions, since *most* instances are unpredictable. The predictable cases must be tested separately but it is necessary to address the problem of providing a mechanism for safely branching when running tests.
- If something does go wrong (e.g. an op-code is unexpectedly undefined or is a branch) then it can be tricky to recover and work out what has happened. A "hang" must be treated as a possible fail case.
- It is hard to be confident that the coverage is exactly right for each supported architecture version. That is to say, one cannot be totally sure that unpredictable and undefined instruction instances have been properly identified. Testing has not been carried out on ARMv5 or ARMv6 boards. Furthermore, the testing automatically filters out all instructions that the model says are unpredictable and some of these cases are not easy to spot in the ARM reference [11]. Omissions can be spotted by examining the table of results, but this process is not foolproof.
- It is not possible to test instruction instances that need to be run in privileged modes (e.g. supervisor mode) or that change the current processor mode. This affects the testing of `mrs`, `msr`, `cps`, `bkpt`, `rfe`, `svc` and `smc` instructions. This also covers hardware exceptions – interrupts, aborts and resets.
- One cannot fully test implementation dependent or system features. This includes semaphore instructions, such as `ldrex` and `clrex` (clear-exclusive), and hint instructions, such as `wfe`, `wfi` (wait for interrupt), `pld` (pre-load data) and `dmb` (data memory barrier). In some cases it is possible to simply observe whether or not these instruction behave like no-op instructions.

It is possible that many of these shortcomings could be overcome by using the JTAG interface on the development boards, instead of using the serial port. The JTAG interface is specifically designed for carrying out debugging with embedded processors. However, this would require more specialist equipment and know-how. We believe that the testing that has been completed to date provides an excellent basis for establishing trust in the model.

7 Related Work

This section discusses related work in formalizing various *commercial* instruction set architectures using interactive theorem provers, i.e. in ACL2, Coq, Isabelle and HOL4. There is much work that is indirectly related, but here we exclude non-commercial architectures (e.g. DLX) and informal or semi-formal ISA models (e.g. in C, System Verilog, Haskell and so forth). It is worth noting that there have been significant efforts made in testing large formal models in other areas, e.g. network protocols, see [2]. Work in the area of commercial ISAs includes the following.

ARM. The ARM specification presented here has its origins in work on verifying the ARM6 processor to the RTL level, see [3]. The specification of the architecture (then version 3) has been almost completely rewritten in the process of upgrading to a monadic specification for architecture versions 4–7. Nevertheless, the experience gained from that project was invaluable and it provided an excellent point of reference.

Processor implementations of the modern architecture versions are proprietary and so we are unable to prove our specification correct with respect to RTL level models. Instead we have validated the model through extensive testing against modern ARM hardware.

ARM/C. The L4 verified project [8] has produced a formally verified microkernel running on ARMv6. However, the model stays at and above the C level and only describes how ARM specific details are seen through C code (e.g. details of interrupts). They assume correctness of C compilers and assume the correctness of in-lined ARM assembly, which constitutes approximately 7% of the microkernel's implementation. Their low-level functional specification of the C code uses monads to make it look similar to the original C.

x86. Our work on testing the model against real hardware was inspired by similar work by Susmit et al. [15] on validation of an operational semantics for x86 machine code. We achieved higher throughput of tests by structuring our test framework differently: we evaluated the ARM model once for each concrete instruction instance and reuse the resulting theorem for multiple test of the same instruction, while the x86 work re-evaluated the x86 model for each test and that work did not make use of development boards.

An extensive formal model of the x86 instruction set is being developed by Hunt in conjunction with work on specifying and verifying the media unit, i.e. a unit which performs floating point arithmetic, of a Centaur Technology's x86 processor [7]. As part of this work, Hunt developed the E hardware specification language which has some monad-like features – in so far as allowing the model to support multiple interpretations. Unfortunately this high-fidelity model of the x86 instruction set architecture is not publicly available.

AAMP7G. Another commercial formal specification has been developed by Rockwell Collins. They have an executable ACL2 model of the Rockwell Collins AAMP7G microprocessor at the instruction-set level [5]. Unfortunately, as with Hunt's x86 model, this model is also not in the public domain.

PowerPC. The Compcert project [10] has produced, and proved the correctness of, an optimising C compiler that targets PowerPC. As part of this work they formalized a subset of PowerPC assembly. Their model is smaller in scope than our ARM model (but sufficient for a compiler) and does not include an instruction decoder, thus their model is an assembly level model. They also have a more abstract view of memory which is expressed in terms of memory blocks, in contrast to our very concrete mapping from 32-bit addresses to 8-bit data.

JVM. A succession of increasingly sophisticated models of the JVM bytecode have been developed in ACL2 [12], the most complicated of which includes threaded behaviour and untyped execution. Models of JVM have also been developed in Isabelle/HOL [9] and Coq [1].

8 Summary

The ARMv7 architecture reference [11] is a sizeable document (stretching to over two thousand pages in length) and it covers all aspects of the architecture. This ARM reference has been used to construct a formal instruction set model in HOL using a monadic specification style. In total the specification comes to around 6500 lines of HOL4 script. The model covers many thousands of instruction instances, which perform non-trivial arithmetic and logical bit vector operations. Instruction decoding is modelled explicitly – mapping ARM and Thumb machine code to an AST data type.

Two important questions arise. How to make the model accessible and easy to use in formal verification projects. How to ensure that the model is trustworthy and as free from bugs as possible. To address these points significant tool support has been developed. In fact, this endeavour requires more code than the model itself, accounting for approximately 15000 lines of code/script. The most important tool is an instructions evaluator – this takes an instruction op-code and outputs a theorem giving that instruction's operational semantics. This *single step* theorem can be used in code verification and in validating the model. A novel technique is used to ensure that the evaluator works efficiently and automatically. The formalization is made more accessible through tight integration with a custom written ARM assembler and disassembler. This saves users having to build and rely upon `gas` as a cross-compiler.

The model has been systematically tested through comparison against the behaviour of ARM hardware. Batches of instructions are randomly generated and loaded onto development boards. The single step theorems are used to evaluate the instructions for multiple data inputs (register assignments) and the results are compared against the output from the boards. This technique has enabled us to run many thousands of tests, identifying and fixing a number of bugs in the model. We encourage others to examine and use the model, tools and test data/results, which are publicly available at `www.cl.cam.ac.uk/~acjf3/arm`.

Acknowledgements. Many thanks to all those who have provided valuable support and feedback for this work. In particular, we would like to thank Mike Gordon, Peter Sewell, Scott Owens and Michael Norrish.

References

1. Atkey, R.: CoqJVM: An executable specification of the Java virtual machine using dependent types. In: Miculan, M., Scagnetto, I., Honsell, F. (eds.) TYPES 2007. LNCS, vol. 4941, pp. 18–32. Springer, Heidelberg (2008)
2. Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 265–276. ACM, New York (2005)
3. Fox, A.: Formal specification and verification of ARM6. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 25–40. Springer, Heidelberg (2003)

4. Furber, S.: ARM: system-on-chip architecture, 2nd edn. Addison-Wesley, Reading (2000)
5. Hardin, D.S., Smith, E.W., Young, W.D.: A robust machine code proof framework for highly secure applications. In: The ACL2 theorem prover and its applications (ACL2 '06), pp. 11–20. ACM, New York (2006)
6. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 3rd edn. Morgan Kaufmann, San Francisco (2002)
7. Hunt Jr., W.A., Swords, S.: Centaur technology media unit verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 353–367. Springer, Heidelberg (2009)
8. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Symposium on Operating Systems Principles (SOSP), pp. 207–220. ACM, New York (2009)
9. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Transactions on Programming Languages and Systems 28(4), 619–695 (2006)
10. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: Principles of Programming Languages (POPL), pp. 42–54. ACM Press, New York (2006)
11. ARM Limited. ARM architecture reference manual: ARMv7-A and ARMv7-R edition. Technical Report ARM DDI 0406B, ARM Limited (2008)
12. Liu, H., Strother Moore, J.: Executable JVM model for analytical reasoning: a study. In: Interpreters, virtual machines and emulators (IVME'03), pp. 15–23. ACM, New York (2003)
13. Myreen, M.O., Gordon, M.J.C.: Verified LISP implementations on ARM, x86 and PowerPC. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 359–374. Springer, Heidelberg (2009)
14. Rivest, R.: The MD5 message-digest algorithm, `http://www.ietf.org/rfc/rfc1321.txt` (accessed, January 2010)
15. Sarkar, S., Sewell, P., Nardelli, F.Z., Owens, S., Ridge, T., Braibant, T., Myreen, M.O., Alglave, J.: The semantics of x86-CC multiprocessor machine code. In: Principles of Programming Languages (POPL), ACM, New York (2009)
16. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer, Heidelberg (2008)
17. Slind, K.X.: TFL: An environment for terminating functional programs, `http://www.cl.cam.ac.uk/~ks121/tfl.html` (accessed, January 2010)
18. Wadler, P.: Comprehending monads. In: Mathematical Structures in Computer Science, pp. 61–78 (1992)