Chapter 1
Specification and Verification of ARM Hardware and Software

Anthony C. J. Fox, Michael J. C. Gordon, and Magnus O. Myreen

Abstract The ARM verification project started in 2000 with the aim of seeing whether existing mechanised formal specification and verification methods could be applied to a commercial off-the-shelf processor. After succeeding in formally verifying that a model of the ARM6 microarchitecture correctly implemented a model of the ARMv3 instruction set architecture (ISA), the project gradually moved away from processor verification to software verification. Models of relatively recent ISAs were specified and a code verification methodology is being developed, where the semantics of code execution is given by the processor ISA model. The long term goal, similar to that of the pioneering CLI stack project, is to create systems on bare metal with as much as possible formally modelled and verified. The current case study is implementing a simple Lisp machine in ARM machine code. This chapter is an overview of the Cambridge ARM project together with some technical highlights that have emerged from the research.

1.1 Introduction and overview

This introductory section provides a high level summary of the history and evolving goals of the ARM verification project. Section 1.2, by Anthony Fox, is a more detailed look into the modelling and verification of ARM processors. Section 1.3, by Magnus Myreen, is more detailed than the others and introduces a new method for creating trustworthy software implementations directly on bare metal. This approach uses the Fox processor model for the semantics of a machine code programming logic that borrows some ideas from separation logic.

In the late 1990s Graham Birtwistle, at the University of Leeds, was investigating the use of the Standard ML (SML) functional programming language for modelling

Anthony C. J. Fox · Michael J. C. Gordon · Magnus O. Myreen
University of Cambridge, Cambridge, United Kingdom,
e-mail: {anthony.fox,mjcg,Magnus.Myreen}@cl.cam.ac.uk

ARM processors. He approached Mike Gordon, a longtime collaborator, about the possibility of a joint project to extend the Leeds modelling work to formal verification. Birtwistle and Gordon, together with contacts at ARM Ltd in Cambridge, submitted a research proposal to the UK Engineering and Physical Sciences Research Council (EPSRC) entitled "Formal Specification and Verification of ARM6". This application was initially turned down on the grounds that the ARM6 processor was obsolete. However, following a strong letter from ARM pointing out that they could not place more modern designs in the public domain, the project was funded on resubmission.

The EPSRC project supported two PhD students at Leeds: Dominic Pajak and Daniel Schostak, and a postdoctoral researcher at Cambridge: Anthony Fox. Pajak and Schostak developed SML models of the ARMv3 ISA and ARM6 microarchitecture, respectively. They both had summer internships at ARM in Cambridge, and this enabled them to talk to ARM engineers to find out details, especially concerning the ARM6 microarchitecture, that were not easily available. Fox took details from Pajack and Schostak's models, and public ARM documentation, and developed formal specifications in higher order logic (HOL) suitable for formal verification. An overview of this work is in Section 1.2. Pajak and Schostak subsequently completed their PhDs at Leeds and took jobs at ARM in Cambridge.

The formal verification that a model of the ARM6 microarchitecture corresponded to the ARMv3 ISA was completed by Fox within a couple of years (much of which were spent developing general proof infrastructure for the HOL4 proof assistant, which was used for the verification). This demonstrated proof-of-concept for the verification of a simple commercial off-the-shelf (COTS) processor. More complex processors, like those implementing the x86 ISA, are widely considered to be too complex for complete formal verification, although very impressive work has been done by Intel, AMD and VIA (Centaur) on the formal verification of parts of implementations x86 processors and by Rockwell Collins on the AAMP7G specialised processor [15]. Many critical systems use simple processors comparable to ARM, and the Leeds-Cambridge project showed that the complete formal verification of these is within the current state-of-the-art.

Following the successful first project, Gordon and Fox applied for continued support and eventually got a new EPSRC grant entitled "Formal Specification and Verification of ARM-based Systems". The aim here was to go beyond the processor to surrounding system components and accurately model things like input/output, coprocessors, bus protocols etc. with the goal of conducting case studies involving these. We also proposed to upgrade our formal ISA models to match more recent versions of ARM. It was decided not to upgrade the microarchitecture verification for two reasons: (i) we would be unable to get access to more recent designs (processor implementations are confidential ARM IP, but ISA specifications are largely in the public domain) and (ii) we felt that re-verifying new implementations would be a lot of detailed work without much research value. Current ARM ISAs are a lot more complex than ARM6, having, for example, instructions for floating point, vector processing, virtual addressing etc. Current ARM microarchitecture implementations have complex pipelines that are much more complicated than that used

in ARM6; this makes the relationship between microarchitecture and ISA computations harder to relate formally. The concepts needed for verifying complex (e.g. superscalar) implementations are reasonably well understood (it was the topic of Fox's PhD and several academic projects [3, 27]) but the actual verifications are significantly more work than those needed for the ARM6 three-stage pipeline. Despite this increased complexity, our feeling is that with modern theorem proving infrastructure (including that developed for the ARM6 verification), the complete formal verification of a modern ARM implementation would be similar in kind to the ARM6 proof. The ARM9 microarchitecture, which is still widely used in mobile devices, would be relatively straightforward, but the latest Cortex designs would be very much more effort (e.g. at least $10\times$ more). The academic research benefits of doing such microarchitecture verifications (e.g. the potential for publication) would not be commensurate with the effort required.

The second EPSRC project was significantly more challenging and the work is still continuing even though the end date of the project has passed. This is possible because the initial ARM research attracted some positive attention and we were offered additional funding from a US Government agency to continue the work and to extend it to explore high assurance cryptographic implementations. We were joined in this work by Konrad Slind and students at Utah, who concentrated on formal compilation of higher order logic specifications directly to a code representation close to ARM assembler. This is described in Slind's chapter in this book, so we will not say more here. At Cambridge, Joe Hurd joined the project to work on formalising the mathematics underlying elliptic curve cryptography (ECC). Our goal was to ensure, by machine checked formal proof, that ARM machine code, with a semantics provided by a high-fidelity processor model, correctly implemented ECC algorithms that were specified using the mathematical concepts of elliptic curves. To this end Hurd developed HOL formalisations of the textbook level mathematical theory underlying ECC in the version of higher order logic supported by the HOL4 system [18, 19]. This lead to difficult proof challenges, such as mechanically proving the associativity of addition on elliptic curves [30].

In parallel with Hurd's investigation of elliptic curve mathematics, Magnus Myreen, then a PhD student at Cambridge, was developing a method of directly verifying ARM assembler. He verified example ARM code implementing some of the operations needed for ECC (e.g. Montgomery multiplication). The overall flow we envisaged was:

1. start with the textbook level mathematical specifications of ECC applications in HOL (Hurd);
2. use a proof-producing compiler to translate the HOL specifications to ARM assembler (Slind);
3. link compiled ARM code to verified runtime code (Myreen).

Although significant progress has been made on all three of these steps we have still (2009) to join everything up into a seamless flow.

One issue that arose as we upgraded to current ARM ISA specifications was the challenge of accurately modelling the communication between an ARM CPU and

its environment, which might include a variety of memories, coprocessors etc. This impacts especially on systems code (see Point 3 above). As described in Section 1.2, the current ARM model separates memory and coprocessors from the main CPU, reflecting how systems are configured. We thus represent an ARM system as a structure containing separately modelled CPUs, various memories and other hardware. An executable model – i.e. a next-state function – is derived by deduction from such a system structure, and it is this that provides the semantics of code. Since the ARM ISA model is complex, the first step in deriving verification infrastructure is to derive higher level rules for reasoning about code that hides the details of the derived next-state function from the verifier. The abstraction methods used for this are described in Myreen's Section 1.3 below. It turns out that these methods can also be used to validate synthesis from low-level HOL, which provides a way of linking the output of Slind's compiler to the Fox processor model. This is also outlined in Section 1.3.

1.2 Specification and verification of ARM architectures

1.2.1 The Swansea methodology

Before coming to work at the University of Cambridge Computer Lab, Anthony Fox completed his PhD at the computer science department of the University of Wales, Swansea. His supervisor, Neal Harman, and the then head of the department, John Tucker, had written a series of papers examining algebraic correctness models for formally verifying computer hardware. For example, this included examining Mike Gordon's micro-programmed case study, see [12, 16]. Fox took Harman and Tucker's work further, adapting their approach to cover pipelined and superscalar micro-architectures. The key features of the Swansea approach are:

- Modelling systems at identified *levels of abstraction*, with particular attention given to formally defining precise classes of *data* and *temporal* abstraction.[1] Correctness is expressed as a commutativity statement that formally relates two abstraction levels.
- When establishing the correctness of microprocessor designs, two key levels are considered: the *programmer's model* (PM) level and the *abstract circuit* (AC) level.
- Formal modelling is based on the use of equational specification, defining the *operational semantics* for a given system at established levels of abstraction. In particular, *primitive recursion* is the principle definition mechanism. This means that the formal specifications can be run or *symbolically evaluated*. Systems with and without I/O were considered.

[1] For example, the class of temporal abstraction maps required for superscalar designs is necessarily more general than that needed for conventional pipelined processors.

- A verification approach based on the use of a series of one-step theorems was developed (not to be confused with the single-step theorems in this chapter). This provides a way to verify systems by principally using case-splitting and equational term-rewriting. That is to say, without the need to carry out an explicit temporal induction or to define top-level invariant predicates. Instead, initialisation functions are used to specify the reachable state-space.
- The approach was designed to be tool neutral, enabling it to be implemented by a wide variety of proof assistants.

In Fox's thesis, a toy architecture with a pipelined implementation was defined, and a pen-and-paper proof of correctness was also presented, see [4]. A superscalar implementation was also defined, together with a formal statement of correctness.[2]

1.2.2 Starting at Cambridge

In the latter half of 2000, Fox moved to Cambridge to start working on the ARM6 project. Work had already begun at Leeds, however their ARM6 model had not been completed yet. This meant that Fox, who had no previous experience in using theorem provers, could gradually start learning HOL4 (then at version Taupo-4). Getting to know HOL can be challenging but fortunately he shared an office with Michael Norrish and there were various other HOL gurus around, including Konrad Slind. As an initial project, the Swansea approach was formalized in HOL. This involved: defining predicates that characterised the various classes of state systems and abstraction maps (for example, state-dependent immersions); formalizing the definition of correctness (one general enough to cover conventional pipelined processors); and proving the 1-step theorems. Then the framework was given a test run with the formal verification of a tiny microprogrammed CPU, see [5]. This was followed by the formal verification of the pipelined design from Fox's thesis.[3]

1.2.3 Modelling the ARM instruction set architecture

Work on specifying the ARM instruction set architecture (ISA) in HOL began in 2001, see [6]. In this context, the ISA is taken to correspond with the assembly programmer's view of an architecture. In general, programmers have access to a fixed set of registers (contained in a CPU), and to a much larger main memory – this is usually connected to the processor via a memory bus.[4] To write code, the assembly

[2] At the time, a pen-and-paper proof of correctness was not feasible/attempted for the superscalar design. Since then some bugs have been identified.

[3] A minor bug was found in the pen-and-paper proof.

[4] In practice, memory may be implemented with a series of caches, firmware, RAM and sometimes with virtual memory e.g. a hard disk or a solid-state drive (SSD). However, memory details are

programmer has at their disposal a set of low-level instructions – these all update the registers and memory in various precisely defined ways. For example, typically there will be a set of *data processing* instructions, which use an arithmetic logic unit (ALU) to perform primitive operations – such as addition, multiplication or bitwise logic – on registers.[5] There will also be a set of *memory access* instructions for loading data from memory to registers and for storing registers to memory. The overall set of instruction is often extended with the introduction of new architecture generations. The number and variety of registers and instructions can vary considerably across platforms but there will normally be *at least* a handful of registers and few dozen or so instructions. Instructions are encoded as a sequence of bits (machine code) to be stored in the main memory. In the x86 architecture instructions have a variable length[6] but with the ARM architecture all instructions are 32 bits long.

The official descriptions of ISAs need to be relatively precise. This is invariably achieved through the use of pseudo-code and in some cases the descriptions are semi-formal. To define an operational semantics for the ARM architecture in HOL, Fox used the specifications produced by Birtwistle's group at Leeds, in conjunction with Steve Furber's book [10] and the official ARM610 data sheet. The objective was to accurately declare a type S, corresponding with the programmer's model state space (registers and memory), and to define a next state function $next : S \rightarrow S$ that specifies the operational semantics of the ARM instructions i.e. the effect of the instructions on the registers and memory. Fortunately, HOL provides excellent support for modelling systems in a functional style, thanks to its "type base" tools,[7] and by virtue of Slind's TFL environment, see [29]. The specification was structured according to instruction *classes* i.e. groups of similar instructions were specified as a whole. To begin with I/O was not considered, in particular, hardware interrupts were not modelled.

1.2.3.1 The state space.

The ARM architecture provides sixteen user-accessible registers and a program status register, each 32-bit words – some of these are then shadowed with versions that are accessible only in *privileged* or *system* modes. These modes are used when running operating system and exception handling code. The main memory is effectively an array of bytes with a 32-bit address space. Thus, the overall state space is:

$$S = (RName \rightarrow word32) \times (word32 \rightarrow word8)$$

invariably implementation dependent and are *mostly* hidden from the programmer. In some cases the actual behaviour can be somewhat counter-intuitive, see [1].

[5] In CISC architectures these instructions may address the memory as well as just registers.

[6] This is mainly because the x86 architectures has its origins in 8-bit and 16-bit computing. Although this variable instruction length can greatly complicate the hardware needed to decode instructions, it can give excellent code density. ARM added a set of 16-bit (Thumb) instructions in order to improve code density.

[7] It is possible to define and work with types in HOL that correspond with algebraic data types.

where *RName* represents the complete set of register names e.g. r8_usr and CPSR. Although HOL has good support for working with algebraic types, there was a slight problem with regards to modelling machine words. At the time, HOL had a theory of words developed by Wai Wong, see [31]; however, this theory was list based and made heavy use of *restricted quantifiers*, with predicates used to restrict the scope of a universal quantifiers.[8] It was decided that this theory would be too cumbersome to use in the context of the ARM6 verification effort, particularly with regards to symbolic evaluation. This started the winding road to developing HOL's current theory of *n*-bit words, with the latest version using an idea from John Harrison (see [17]) in order to get around the perceived need for restricted quantifiers.

1.2.4 Modelling the ARM6 micro-architecture

In 2002 the ARM6 micro-architecture was modelled in HOL, see [7], and in 2003 a formal verification was completed, see [8]. The ARM6 microprocessor dates from around 1994 and was widely deployed in a number of low-powered devices, such as the Apple Newton PDA. The processor's micro-architecture is relatively simple, employing a 3-stage pipeline with *fetch*, *decode* and *execute* stages. As with other commercial designs, details of the processor's implementation are not in the public domain. It was only through collaboration with ARM Ltd., and Daniel Schostak's internship there, that is was possible to develop the formal model. The ARM6 processor is no longer in production, which was a factor in us gaining permission to carry out this research. However, it is worth noting that the ARM9 (circa 2004, and used in the Nokia N-Gage) is *not* a superscalar design and has a 5-stage pipeline. It can be argued therefore that the verification of the ARM6 is still pertinent with respect to some more modern designs.

Daniel Schostak produced a very detailed model of the ARM6 for his thesis, see [28]. He only introduced a limited amount of abstraction, modelling the RTL (register transfer level) with a two-phase clock model. A limited amount of data abstraction was applied when producing the cycle accurate HOL model. One of the most useful resources in achieving this was Schostak's tabular style paper specification.[9] For example, his tabular description of the DIN latch (which stores input from the data bus) is shown below.

[8] HOL is based on simple type theory and does not directly support predicate sub-typing.

[9] Schostak produced extensive paper specifications of the ARM6 using various styles. He also produced a high-fidelity implementation in ML and now works full time at ARM Ltd.

DIN

IC *	*IS* *	
data_proc	t_2	*IREG*
mrs_msr	t_2	*IREG*
ldr	t_2	*IREG*
ldr	t_4	***DATA***
str	t_2	*IREG*
ldm	t_4	***DATA***
ldm	t_n	***DATA***
swp	t_4	***DATA***
br	t_2	*IREG*
mrc	t_4	***DATA***
ldc	t_2	*IREG*
stc	t_2	*IREG*
x	x	x

This was translated into the following HOL definition:

```
⊢ ∀ic is ireg data.
    DIN ic is ireg data =
    if
      ((ic = ldr) ∨ (ic = ldm) ∨ (ic = swp) ∨ (ic = mrc)) ∧
      (is = t4) ∨ (ic = ldm) ∧ (is = tn)
    then
      data
    else
      ireg
```

Here `ic` represents the instruction class and `is` is the instruction *step*, for example, `t3` is the first cycle of the pipeline execute stage and `tn` represents an iterated phase. By defining the next-state behaviour of all of the processor's latches and buses, it was possible to define a next-state function for the entire ARM6 core. Formal verification proceeded by case-splitting over the instruction class – the final version had seventeen such classes. Inevitably a small number of bugs were found in all of the specifications. Ultimately the ARM6 can be regarded as a reference implementation and so the formal verification can be seen an exercise in developing an ISA model that is a verified abstraction of the processor.

1.2.4.1 Coverage.

Somewhat confusingly, the ARM6 processor implements version *three* of the ARM architecture, written as ARMv3. To begin with, all of the ARMv3 instructions were modelled at the ISA level but some "hard" features were not included in the first ARM6 model – accordingly they were dropped from the ISA model prior to carrying out the initial verification attempt. The omissions included the `mul`, `ldm` and `stm` instructions, which all have relatively complex low-level behaviour (an iterated

phase).[10] To complete the formal proof invariants were constructed for these cases. The coprocessor instructions and hardware interrupts required models with input and output and this is discussed below. A feature complete formal verification was finished in 2005, see [9].

1.2.4.2 Input and output.

To accommodate input and output (I/O) features, the HOL formalization of the Swansea approach was extended. It was also necessary to make significant changes to the ISA and micro-architecture models, and the formal verification required a fair amount of reworking. More sophisticated reasoning is required when verifying the correctness of coprocessor instructions and hardware interrupts. For example, the communication between the ARM core and coprocessor happens through a busy-wait loop, which has to be assumed to terminate after some indeterminate interval. It is also necessary to reason about the priority and timing of interrupts and, added to this mix, a reset signal can abort instructions at any cycle.

In the process of adding I/O, the memory was removed from the state-space of the ISA and micro-architecture specifications. At the ISA level this meant that the state-space consisted of just the programmer's model registers, together with an instruction register (the op-code of the instruction to be run) and a exception status field, that is:

$$S = (\mathit{RName} \rightarrow \mathit{word32}) \times \mathit{word32} \times \mathit{Exception}$$

The next-state function is then of the form $\mathit{next} : S \times I \rightarrow S$, where I represents a set of input values i.e. data from memory and coprocessors, together with hardware interrupts. There is also an output function $\mathit{out} : S \rightarrow O$ that models data being passed from the processor to the memory and coprocessors. One consequence of these changes is that the resulting next-state functions no longer provide a *direct* means to run programs i.e. there is no longer a prescriptive model of memory, just an interface.

1.2.5 Beyond the ARM6

Following the formal verification of 2005, it was decided to extend the ISA model and focus on machine code verification, forgoing the considerable overhead associated with further extending and re-verifying the ARM6 model. It was at about this time that Magnus Myreen started his PhD at Cambridge. To begin with ARMv3M was supported (with the inclusion of long [64-bit] multiplies) and then ARMv4 was covered (through the addition of half-word and signed load and store instructions). At the time of writing this article, the ARMv4 architecture is still very much in use

[10] The ARM6 ALU does not contain a multiplier, so instead the processor's adder and shifter are used to implement Booth's algorithm over a number of clock cycles.

– it is implemented by a selection of processors in the ARM7, ARM8 and ARM9 families (as used in the Nintendo DS and Apple iPod).

After making these extensions, the next step was to provide support for reasoning about assembly code. In particular, it is not especially practical to work directly with 32-bit machine-code values. To this end, a HOL type was added to represent decoded ARM instructions, a parser/assembler was written,[11] and there was also support for pretty-printing instructions i.e. providing disassembly of machine code.

A new top-level next-state function was defined (using the existing definitions as sub-functions) and this reintroduced the main memory as part of the state-space. Consequently it was again possible to run code using the model and one could also start reasoning about the semantics of programs. A pure memory model was assumed, that is to say, the memory was treated as a simple array with read and write accesses that never fail. A fast method for running code (useful in testing the model) was provided through the use of Konrad Slind's EmitML tool – this converted the HOL definitions into Standard ML. This ML code was compiled with MLton, resulting in an instruction throughput performance of approximately ten thousand instructions-per-second (10 kips).

It was then necessary to address the problem that the formal model somewhat obfuscates the behaviour of particular instruction instances. For example, one cannot read the specification and immediately see the effect of the instruction `add r1,r2,r3`. The reasons for this are: the underlying model is based on machine code; the specification is structuring according to instructions classes (not instruction instances); and the overall semantics is expressed through one monolithic, top-level next-state function. To address this, a collection of *single-step* theorems of the following form are generated:

$$P(s) \Rightarrow (next(s) = s') \ .$$

Here the antecedent predicate P represents the context (showing exactly which instruction instance is to be run) and s' is the result of symbolically evaluating the model in this context. These theorems are generated using *forward-proof* (as opposed to goal-directed proof) and simplifications are applied to make the results as user-friendly as possible. The resulting theorems make the specification more accessible and usable. The term representing s' can be examined to see which registers and memory locations have been read and/or updated and this is pertinent to Magnus Myreen's code verification work.

1.2.5.1 Further refinements.

With the addition of the 16-bit Thumb instructions, the ARMv4 model was later extended to ARMv4T. A more advanced mechanism for constructing a complete system was also examined i.e. building a system composed of ISA, memory and

[11] This was originally done using `mosmllex` and `mosmlyac` and later ported to `mllex` and `mlyacc`, so as to generate Standard ML.

coprocessors models. A compositional, circuit-based style was adopted, wherein the output from one unit is connected to the input of another. This means that one can more easily consider different system configurations, for example, "plugging-in" different memory models. This contrasts with the previous approach wherein the memory was more hard-wired into the ISA specification.

1.2.6 Going Monadic

In addition to his work with the ARM architecture, Myreen has also worked with formal models of the x86 and PowerPC architectures. The x86 model initially came from collaborating with Susmit Sarkar, who has been working with Peter Sewell and others in the field of relaxed memory models, see [1]. This group made enquiries as to the suitability of the ARM model with respect to their research. However, a single-step operational semantics was not what they after – they needed to know the precise order of all memory and register accesses. In collaboration with Myreen, they had developed a monadic approach to ISA specification and, inspired by this, Fox agreed to completely re-specify the entire ARM ISA using this approach. This would provide an event-based semantics for work on relaxed memory models and an operational semantics for work on code verification.

In their monadic approach three principle operators are used: sequencing (`seqT` or `>>=`), parallel composition (`parT` or `|||`), and returning a constant value (`constT` or `return`). For example, in

```
(f ||| g) >>= (λ(x,y). return (x + y + 1))
```

the operations `f` and `g` are performed in parallel and the results are then combined in a summation and returned. The overall type of this term is `num M` and the precise details of this type are hidden underneath a HOL type abbreviation on `M`. For example, in the standard ARM operational semantics we have:

```
'a M = arm_state → ('a, arm_state) error_option
```

Here `arm_state` is the state-space and `error_option` is just like the standard functional *option* type, except that the "none" case is tagged with a string, which provides a useful mechanism for reporting erroneous behaviour. In this *sequential* operational semantics, the `parT` operator is evaluated sequentially with a left-to-right ordering e.g. `f` is applied before `g` in the example above.

There are many advantages to working in this monadic style, these include:

- The ability to modify the underlying semantics by simply changing the monad's type and the definitions of the monadic operators.
- The ability to avoid excessive parameter passing and to hide details of the state-space. In some cases there might not even be a state-space.
- It provides a clean way to handle erroneous cases. In particular, it is easy to model behaviour that the ARM architecture classifies as UNPREDICTABLE.

- With some pretty-printing support, the definitions look more like imperative code. This makes the specifications more readable to those unfamiliar with functional programming and it also provides a more visible link with pseudo-code from reference manuals.

For example, consider the following pseudo-code from the ARM architectural reference manual:

```
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_ARM then
        if ArchVersion() < 6 && address<1:0> != '00' then UNPREDICTABLE;
        BranchTo(address<31:2>:'00');
    else
        BranchTo(address<31:1>:'0');
```

With pretty-printing turned on this corresponds with the following HOL code:

```
⊢ ∀ii address.
    branch_write_pc ii address =
    do
      iset ← current_instr_set ii;
      if iset = InstrSet_ARM then
        do
          version ← arch_version ii;
          if version < 6 ∧ (1 >< 0) address ≠ 0w then
            errorT "branch_write_pc: unpredictable"
          else
            branch_to ii ((31 '' 2) address)
        od
      else
        branch_to ii ((31 '' 1) address)
    od
```

Although the translation is not literal, there is clearly a connection between the two specifications. The function `branch_write_pc` has return type `unit M`, that is to say, it is similar to a procedure (or `void` function in C). The HOL model introduces a variable `ii`, which is used to uniquely identify the source of all read and write operations – this becomes significant in multi-core systems with shared memory. The operator `errorT` is used to handle the unpredictable case. The word *extract* and *slice* operations (`><` and `''`) are used to implement the bit operations shown in the ARM reference. Inequality is overloaded to be `<>`, which corresponds with `!=` in the pseudo-code. Observe that the HOL specification does not explicitly refer to state components; such details are hidden by the monad, and the operations `arch_version` and `current_instr_set` automatically have access to all the data that they need. In the sequential model, the state actually contains a component that identifies the specific version of the architecture being modelled e.g. `ARMv4` or `ARMv4T`, both of which give a version number of four. This makes it possible to simultaneously support multiple architecture versions. Further refinement has also

been made in the process of producing the new specification, especially with regard to instruction decoding and the representation of instructions.

1.2.6.1 Coverage.

The monadic specification covers nearly all of the currently supported ARM architecture versions, that is to say: ARMv4, ARMv4T, ARMv5T, ARMv5TE, ARMv6, ARMv6K, ARMv6T2, ARMv7-A and ARMv7-R. A significant number of new instruction were introduced with ARMv6, which was introduced with the ARM11 family of processors. The latest generation (ARMv7) has only a small number of extra ARM instructions but these versions do all support Thumb2 – this provides a large number of double-width Thumb instructions, which cover nearly all of the functionality of the standard ARM instructions. The Cortex-A8 processor (as found in the Apple iPhone 3GS and Palm Pre) implements ARMv7-A.

The HOL model also covers the Security and Multiprocessor extensions. It does not support Jazelle, which provides hardware support for running Java bytecode. Technical details about Jazelle and its implementations are restricted to ARM licensees only, see [2]. Consequently, a HOL specification of Jazelle is very unlikely. Documentation is available for the ThumbEE, VFP (vector floating-point) and Advanced SIMD extensions but they have not been specified yet – the SIMD extensions were introduced with ARMv7 and the associated infrastructure is referred to as NEON™ technology.

1.2.6.2 Single-step theorems.

Recently a tool for generating single-step theorems for the monadic model has been developed. These theorems are now generated entirely *on-the-fly* for specific opcodes.[12] This contrasts with the previous approach whereby a collection of pregenerated theorems (effectively templates) are stored and then specialised prior to use. The old approach is not practical in the context of the much larger number of instructions and range of contexts. The single-step theorems are generated entirely through forward proof and so the process is not especially fast. Consequently, it may prove necessary to store some of the resulting theorems in order to improve runtimes further down the line.

The function call

```
arm_stepLib.arm_step "v6T2,be,thumb,sys" "FB02F103";
```

produces the following theorem

[12] This tool makes heavy use of the HOL conversion `EVAL`.

```
⊢ ∀state.
    (ARM_ARCH state = ARMv6T2) ∧ (ARM_EXTENSIONS state = {}) ∧...∧
    (ARM_MODE state = 31w) ∧ aligned (ARM_READ_REG 15w state,2) ∧
    (ARM_READ_MEM (ARM_READ_REG 15w state + 3w) state = 3w) ∧
    (ARM_READ_MEM (ARM_READ_REG 15w state + 2w) state = 241w) ∧
    (ARM_READ_MEM (ARM_READ_REG 15w state + 1w) state = 2w) ∧
    (ARM_READ_MEM (ARM_READ_REG 15w state) state = 251w) ⇒
    (ARM_NEXT state =
     SOME
       (ARM_WRITE_REG 1w
          (ARM_READ_REG 2w state * ARM_READ_REG 3w state)
          (ARM_WRITE_REG 15w (ARM_READ_REG 15w state + 4w) state)))
```

For brevity/clarity, some parts of the antecedent have been omitted. The first argument to `arm_step` is a string containing configuration options e.g. the architecture version and the byte ordering. The second string is the instruction op-code. In the example above, 0xFB02F103 is the machine code for the Thumb2 instruction `mul r1,r2,r3`.[13] The four instruction bytes are read from memory using the program-counter, which is register fifteen.

The next-step theorem shown above bears little or no *visible* resemblance to the underlying monadic specification. The functions in uppercase are defined in post-hoc manner, so as to present a more conventional state-oriented semantics. The top level next-state function `ARM_NEXT` returns an option type – if an error occurred (e.g. with an unpredictable case) then the result would be `NONE` but in practice the tool raises an ML exception for such cases.

1.2.6.3 Active and future work.

Recent work includes updating the parser, assembler and pretty-printing support. The instruction parser has been completely rewritten in ML, abandoning the use of `mllex` and `mlyacc`. It would have been possible to avoid writing an assembler and instead interface with GNU's `binutils` but this would require users to specifically install these tools, configuring them as an ARM cross-compiler. Also, only the very latest version of the GNU assembler supports Thumb2 and it has had a small number of teething problems (bugs) in that area.

Active work includes validating the HOL ARM models by comparing the results of executing instructions on the formal models in HOL with the results of executing the same instructions on ARM processors. The test bench for this uses off-the-shelf boards and runs continuously using automatically generated instructions (e.g. one such board that is used is the Beagle Board: `beagleboard.org`). This testing has revealed some subtle formalisation errors that would have been very hard to identify by simply reading the ARM instruction set documentation.

One future area of work will be in handling I/O. It should be relatively straightforward to add hardware interrupts. Readers may have observed that the sequential

[13] At the moment op-codes are being generated using GNU's `binutils` tools. FB02F103 breaks up into 251w, 2w, 241w and 3w, which are used in the theorem.

version of the monadic model again includes the memory as part of the state-space. This could be considered to be a regressive step in comparison with the approach discussed at the end of Section 1.2.5. However, the monadic approach does make it easier modify the underlying memory model.[14] It is expected that memory-mapped I/O (MMIO) can be supported by interleaving calls to next-state functions i.e. the ISA next-state function would be interleaved with a MMIO next-state function.

1.3 High-assurance software engineering

In 2005, Myreen started his PhD which came to focus on theories and tools for proving ARM machine code correct on top of Fox's formal specification of the ARM ISA. This section presents the current state of the resulting framework which has come to support both formal verification of existing ARM machine code and synthesis of new ARM code from functional specifications. The framework consists of three layers:

1. **Hoare logic for machine code** is used for making concise and composable formal specifications about ARM code (Section 1.3.1).
2. **A decompiler** aids verification by automatically extracting functional descriptions of ARM code from Fox's detailed ISA specification (Section 1.3.2).
3. **A compiler** is used for synthesis of new ARM code from, possibly partial, functional specifications (Section 1.3.3).

Our to-date largest case study, synthesis of a formally verified LISP interpreter, is outlined in Section 1.3.4.

1.3.1 Machine-code Hoare logic

Machine-code programs operate over a heterogeneous state consisting of register, memory locations and various status bits. As a result keeping track of which resources might have been altered by some ARM code can easily become tedious. In order to avoid always explicitly stating "...and nothing else changed" (a frame property), we write our theorems in terms of a machine-code Hoare triple $\{p\}\ c\ \{q\}$ which implicitly formalise a frame property (from separation logic [26]):

> $\{p\}\ c\ \{q\}$ is true if any state of an ARM processor s which satisfies precondition p, can through execution of code c on the ARM ISA reach a state s' which satisfies postcondition q; and, furthermore, all resources not mentioned in p will remain unchanged in the transition from s to s'.

[14] Although at the moment the `arm_step` tool does make some assumptions about the memory model.

A formal definition of this Hoare triple will be given later, also see [22].

The frame property manifests itself in practice as a proof rule called *the frame rule* (again borrowed from separation logic). The frame rules allows an arbitrary assertion r to be added to any Hoare triple $\{p\}\ c\ \{q\}$ using the separating conjunction $*$ (defined later):

$$\{p\}\ c\ \{q\} \implies \forall r.\ \{p * r\}\ c\ \{q * r\}$$

The frame property of our Hoare triples allows us to only mention locally relevant resources, e.g. a theorem describing the ARM instruction `add r4,r3,r4` (encoded as `0xE0834004`), need only mention resources register 3, 4 and 15 (the program counter). For example the following Hoare-triple theorem states, if register 3 has value a, register 4 has value b and the program counter is p, then the code `E0834004` at location p will reach a state where register 3 has value a, register 4 holds $a+b$ and the program counter is set to $p+4$:

$$\begin{array}{l}\{\,\mathsf{r3}\ a * \mathsf{r4}\ b * \mathsf{pc}\ p\,\}\\ \quad p : \texttt{E0834004}\\ \{\,\mathsf{r3}\ a * \mathsf{r4}\ (a+b) * \mathsf{pc}\ (p+4)\,\}\end{array}$$

The frame rule allows us to infer that the value of register 5 is left unchanged by the above ARM instruction, since we can instantiate r in the frame rule above with an assertion saying that register 5 holds value c, i.e. $\mathsf{r5}\ c$.

$$\begin{array}{l}\{\,\mathsf{r3}\ a * \mathsf{r4}\ b * \mathsf{pc}\ p * \mathsf{r5}\ c\,\}\\ \quad p : \texttt{E0834004}\\ \{\,\mathsf{r3}\ a * \mathsf{r4}\ (a+b) * \mathsf{pc}\ (p+4) * \mathsf{r5}\ c\,\}\end{array}$$

All user-level ARM instructions satisfy specification in this style. Memory reads and writes are not much different, e.g. Hoare-triple theorem describing the instruction `swp r4,r4,[r3]` (`E1034094`) for swapping the content of memory location a given in register 3 with that of register 4 is given as follows. Here $\mathsf{m}\ m$ states that a function m, a partial mapping from addresses (32-bit words) to values (32-bit words) correctly represents a portion of memory (addresses $\mathsf{domain}\ m$), address a must be in the memory portion covered by m and for tidiness needs to be word-aligned, i.e. $a\ \&\ 3 = 0$; we write $m[a \mapsto b]$ for m updated to map a to b.

$$\begin{array}{l}a\ \&\ 3 = 0 \wedge a \in \mathsf{domain}\ m \implies\\ \{\,\mathsf{r3}\ a * \mathsf{r4}\ b * \mathsf{m}\ m * \mathsf{pc}\ p\,\}\\ \quad p : \texttt{E1034094}\\ \{\,\mathsf{r3}\ a * \mathsf{r4}\ (m(a)) * \mathsf{m}\ (m[a \mapsto b]) * \mathsf{pc}\ (p+4)\,\}\end{array}$$

The following subsections will present the definition of our machine-code Hoare triple and some proof rules (HOL theorems) that have been derived from the definition of the Hoare triple, and are hence sound.

1.3.1.1 Set-based separating conjunction.

The definition of our machine-code Hoare triple uses the separating conjunction, which we define unconventionally to split sets rather than partial functions. Our definition of the set-based separating conjunction states that $(p * q)\ s$ whenever s can be split into two disjoint sets u and v such that p holds for u and q holds for v:

$$(p * q)\ s\ =\ \exists u\ v.\ p\ u \wedge q\ v \wedge (u \cup v = s) \wedge (u \cap v = \{\})$$

In order to make use of this set-based separating conjunction we need to translate ARM states into sets of state components. We define the type of an ARM state elements as a data-type with constructors:

$$\begin{array}{ll} \mathsf{Reg} & : \mathsf{word}_4 \to \mathsf{word}_{32} \to \mathsf{arm\_state\_element} \\ \mathsf{Status} & : \mathsf{status\_names} \to \mathsf{boolean} \to \mathsf{arm\_state\_element} \\ \mathsf{Memory} & : \mathsf{word}_{30} \to \mathsf{word}_{32} \to \mathsf{arm\_state\_element} \\ \mathsf{Undef} & : \mathsf{bool} \to \mathsf{arm\_state\_element} \end{array}$$

We define a function arm2set for translating states representation used in the ARM ISA specification into sets of ARM state elements, using read functions arm_read_reg, arm_read_mem arm_read_status, arm_read_undefined which, respectively, read a register, memory location, status bit and undefined flag. Here range $f = \{\ y \mid \exists x.\ f\ x = y\ \}$.

$$\begin{array}{l} \mathsf{arm2set}\ \mathit{state} = \\ \quad \mathsf{range}\ (\lambda r.\ \mathsf{Reg}\ r\ (\mathsf{arm\_read\_reg}\ r\ \mathit{state})) \cup \\ \quad \mathsf{range}\ (\lambda a.\ \mathsf{Mem}\ a\ (\mathsf{arm\_read\_mem}\ a\ \mathit{state})) \cup \\ \quad \mathsf{range}\ (\lambda s.\ \mathsf{Status}\ s\ (\mathsf{arm\_read\_status}\ s\ \mathit{state})) \cup \\ \quad \{\ \mathsf{Undef}\ (\mathsf{arm\_read\_undefined}\ \mathit{state})\ \} \end{array}$$

Some basic assertions are defined over sets of ARM state elements as follows. We often write r1 a, r2 b, etc. as abbreviations for reg 1 a, reg 2 b, etc.

$$\begin{array}{c} (\mathsf{reg}\ i\ a)\ s = (s = \{\ \mathsf{Reg}\ i\ a\ \}) \\ (\mathsf{mem}\ a\ w)\ s = (s = \{\ \mathsf{Mem}\ a\ w\ \}) \end{array}$$

These assertions have their intended meaning when used with arm2set:

$$\begin{array}{l} \forall p\ s.\ (\mathsf{mem}\ a\ w * p)\ (\mathsf{arm2set}\ s)\ \Longrightarrow\ (\mathsf{arm\_read\_mem}\ a\ s = w) \\ \forall p\ s.\ (\mathsf{reg}\ i\ v * p)\ (\mathsf{arm2set}\ s)\ \Longrightarrow\ (\mathsf{arm\_read\_reg}\ i\ s = v) \end{array}$$

The separating conjunction separates assertions:

$$\forall p\ s.\ (\mathsf{mem}\ a\ x * \mathsf{mem}\ b\ y * \mathsf{reg}\ i\ u * \mathsf{reg}\ j\ v * p)\ (\mathsf{arm2set}\ s)\ \Longrightarrow\ a \neq b \wedge i \neq j$$

Other assertions used in this text are:

$$
\begin{aligned}
\mathsf{aligned}\ a &= (a \,\&\, 3 = 0) \\
\mathsf{emp}\ s &= (s = \{\}) \\
\langle b \rangle\ s &= (s = \{\}) \wedge b \\
(\mathsf{code}\ c)\ s &= (s = \{\, \mathsf{Mem}\ (a[31\text{—}2])\ i \mid (a,i) \in c \,\}) \\
(\mathsf{m}\ m)\ s &= (s = \{\, \mathsf{Mem}\ (a[31\text{—}2])\ (m\ a) \mid a \in \mathsf{domain}\ m \wedge \mathsf{aligned}\ a \,\}) \\
(\mathsf{pc}\ p)\ s &= (s = \{\mathsf{Reg}\ 15\ p, \mathsf{Undef}\ \mathsf{F}\}) \wedge \mathsf{aligned}\ p \\
(\mathsf{s}\ (n,z,c,v))\ s &= (s = \{\, \mathsf{Status}\ \mathsf{N}\ n, \mathsf{Status}\ \mathsf{Z}\ z, \mathsf{Status}\ \mathsf{C}\ c, \mathsf{Status}\ \mathsf{V}\ v \,\})
\end{aligned}
$$

1.3.1.2 Definition of Hoare triple.

Let $\mathsf{run}(n,s)$ be a function which applies the next-state function from our ARM ISA specification n times to ARM state s.

$$
\begin{aligned}
\mathsf{run}(0,s) &= s \\
\mathsf{run}(n{+}1,s) &= \mathsf{run}(n, \mathsf{arm\_next\_state}(s))
\end{aligned}
$$

Our machine-code Hoare triple has the following definition: any state s which satisfies p separately from code c and some frame r (written $p * \mathsf{code}\ c * r$), will reach (after some k applications of the next-state function) a state which satisfies q separately from the code c and frame r (written $q * \mathsf{code}\ c * r$).

$$
\begin{aligned}
\{p\}\ c\ \{q\} = \forall s\ r.\ (p * \mathsf{code}\ c * r)\ (\mathsf{arm2set}(s)) \implies \\
\exists k.\ (q * \mathsf{code}\ c * r)\ (\mathsf{arm2set}(\mathsf{run}(k,s)))
\end{aligned}
$$

As a convention, we write concrete code sets $c = \{(p,i),(q,j),\ldots\}$ as comma separated lists without brackets "$p : i, q : j, \ldots$" in order to avoid confusion with the curly brackets used when writing Hoare triples.

An example of what a machine-code Hoare triple means in terms of the basic read functions is shown in Figure 1.1. The last line which relates arm2set *state* to arm2set $state'$ states that nothing (observable through the read functions) changed other than registers 7, 8 and 15. This fact that nothing outside of the foot-print of the specification was affected, comes from the universally quantified frame r in the definition of the machine-code Hoare triple.

1.3.1.3 Proof rules.

Below we list some theorems proved from the definition of our Hoare triple. These theorems are cumbersome to use in manual proofs, but easy to use in building proof automation, which is the topic of the next two sections.

Frame: $\{p\}\ c\ \{q\} \implies \forall r.\ \{p * r\}\ c\ \{q * r\}$

The frame rule allows any assertions to be added to the pre- and postconditions of a Hoare triple, often applied before composition.

$$
\begin{array}{l}
\{\,\mathsf{r7}\ x * \mathsf{r8}\ y * \mathsf{pc}\ p\,\}\ \ p : \mathtt{E2878001}\ \{\,\mathsf{r7}\ x * \mathsf{r8}\ (x{+}1) * \mathsf{pc}\ (p{+}4)\,\} \\
= \\
\forall \mathit{state}.\ (\mathsf{arm\_read\_reg}\ 7\ \mathit{state} = x) \wedge \\
\qquad (\mathsf{arm\_read\_reg}\ 8\ \mathit{state} = y) \wedge \\
\qquad (\mathsf{arm\_read\_reg}\ 15\ \mathit{state} = p) \wedge \mathsf{aligned}\ p \wedge \\
\qquad (\mathsf{arm\_read\_undefined}\ 15\ \mathit{state} = \mathsf{F}) \wedge \\
\qquad (\mathsf{arm\_read\_mem}\ p\ \mathit{state} = \mathtt{E2878001}) \implies \\
\qquad \exists n\ \mathit{state}'.\ (\mathit{state}' = \mathit{run}(n, \mathit{state})) \wedge \\
\qquad\qquad (\mathsf{arm\_read\_reg}\ 7\ \mathit{state}' = x) \wedge \\
\qquad\qquad (\mathsf{arm\_read\_reg}\ 8\ \mathit{state}' = x{+}1) \wedge \\
\qquad\qquad (\mathsf{arm\_read\_reg}\ 15\ \mathit{state}' = p{+}4) \wedge \mathsf{aligned}\ (p{+}4) \wedge \\
\qquad\qquad (\mathsf{arm\_read\_undefined}\ 15\ \mathit{state}' = \mathsf{F}) \wedge \\
\qquad\qquad (\mathsf{arm\_read\_mem}\ p\ \mathit{state}' = \mathtt{E2878001}) \wedge \\
\qquad\qquad (\mathsf{arm2set}\ \mathit{state} - \mathit{Frame} = \mathsf{arm2set}\ \mathit{state}' - \mathit{Frame})
\end{array}
$$

where $\mathit{Frame} = \mathsf{range}\ (\lambda w.\ \mathsf{Reg}\ 7\ w) \cup \mathsf{range}\ (\lambda w.\ \mathsf{Reg}\ 8\ w) \cup \mathsf{range}\ (\lambda w.\ \mathsf{Reg}\ 15\ w)$

Fig. 1.1 A machine-code Hoare triple expanded.

Composition: $\{p\}\ c_1\ \{q\} \wedge \{q\}\ c_2\ \{r\} \implies \{p\}\ c_1 \cup c_2\ \{r\}$

The composition rule composes two specifications and takes the union of the two code sets, which may overlap (happens for loops).

Precondition strengthening: $\{p\}\ c\ \{q\} \wedge (\forall s.\ r\,s \implies p\,s) \implies \{r\}\ c\ \{q\}$
Postcondition weakening: $\{p\}\ c\ \{q\} \wedge (\forall s.\ q\,s \implies r\,s) \implies \{p\}\ c\ \{r\}$

Preconditions can be strengthened, postconditions can be weakened.

Precondition exists: $\{\exists x.\ p\,x\}\ c\ \{q\} \iff \forall x.\ \{p\,x\}\ c\ \{q\}$

Existential quantifiers in the precondition are equivalent to universal quantifiers outside of the Hoare triple specification.

Move pure condition: $\{p * \langle b \rangle\}\ c\ \{q\} \iff (b \implies \{p\}\ c\ \{q\})$

Pure stateless assertions, $\langle b \rangle$, can be pulled out of the precondition. Here b has type bool.

Code extension: $\{p\}\ c\ \{q\} \implies \forall e.\ \{p\}\ c \cup e\ \{q\}$

The code can be extended arbitrarily. This rule highlights that $\{p\}\ c\ \{q\}$ means that c is *sufficient* to transform any state satisfying p into a state satisfying q. Thus any larger set $c \cup e$ is also sufficient.

1.3.2 Decompilation of ARM code

To aid verification of machine code, we have developed a novel verification technique [24] which is based on decompiling machine code into functions in the logic of a theorem prover, in this case the HOL4 theorem prover.

1.3.2.1 Example.

Given some ARM code, which calculates the length of a linked list,

```
 0: E3A00000      mov r0, #0         ; set reg 0 to 0
 4: E3510000  L:  cmp r1, #0         ; compare reg 1 with 0
 8: 12800001      addne r0, r0, #1   ; if not equal: add 1 to reg 1
12: 15911000      ldrne r1, [r1]     ;           load mem[reg 1] into reg 1
16: 1AFFFFFB      bne L              ;           jump to compare
```

the decompiler reads the hexadecimal numbers, extracts a function f and a safety condition $\mathsf{f}_{\mathsf{pre}}$ which describe the data-update performed by the ARM code:

$$\begin{array}{ll}
\mathsf{f}(r_0,r_1,m) = & \mathsf{f}_{\mathsf{pre}}(r_0,r_1,m) = \\
\quad \mathsf{let}\ r_0 = 0\ \mathsf{in}\ \mathsf{g}(r_0,r_1,m) & \quad \mathsf{let}\ r_0 = 0\ \mathsf{in}\ \mathsf{g}_{\mathsf{pre}}(r_0,r_1,m) \\
\mathsf{g}(r_0,r_1,m) = & \mathsf{g}_{\mathsf{pre}}(r_0,r_1,m) = \\
\quad \mathsf{if}\ r_1 = 0\ \mathsf{then}\ (r_0,r_1,m)\ \mathsf{else} & \quad \mathsf{if}\ r_1 = 0\ \mathsf{then}\ \mathsf{T}\ \mathsf{else} \\
\quad\quad \mathsf{let}\ r_0 = r_0{+}1\ \mathsf{in} & \quad\quad \mathsf{let}\ r_0 = r_0{+}1\ \mathsf{in} \\
\quad\quad \mathsf{let}\ r_1 = m(r_1)\ \mathsf{in} & \quad\quad \mathsf{let}\ \mathit{cond} = r_1 \in \mathsf{domain}\ m \wedge \mathsf{aligned}\ r_1\ \mathsf{in} \\
\quad\quad\quad \mathsf{g}(r_0,r_1,m) & \quad\quad \mathsf{let}\ r_1 = m(r_1)\ \mathsf{in} \\
 & \quad\quad\quad \mathsf{g}_{\mathsf{pre}}(r_0,r_1,m) \wedge \mathit{cond}
\end{array}$$

The decompiler also proves the following theorem which state that f is accurate with respect to the ARM model, for input values that satisfy $\mathsf{f}_{\mathsf{pre}}$. Here notation $(k_1,k_2,...,k_n)$ is $(x_1,x_2,...,x_n)$ abbreviates $k_1\ x_1 * k_2\ x_2 * ... * k_n\ x_n$, i.e. expression $(\mathsf{r0},\mathsf{r1},\mathsf{m})$ is (r_0,r_1,m) states that register 0 has value r_0, register 1 is r_1 and part of memory is described by m.

$$\begin{array}{l}
\{\,(\mathsf{r0},\mathsf{r1},\mathsf{m})\ \mathsf{is}\ (r_0,r_1,m) * \mathsf{s} * \mathsf{pc}\ p * \langle \mathsf{f}_{\mathsf{pre}}(r_0,r_1,m)\rangle\,\} \\
p : \texttt{E3A00000},\texttt{E3510000},\texttt{12800001},\texttt{15911000},\texttt{1AFFFFFB} \\
\{\,(\mathsf{r0},\mathsf{r1},\mathsf{m})\ \mathsf{is}\ (\mathsf{f}\ (r_0,r_1,m)) * \mathsf{s} * \mathsf{pc}\ (p+20)\,\}
\end{array} \tag{1.1}$$

The user can then prove that the original machine code indeed calculates the length of a linked-list by simply proving that the extracted function f does that. Let list state that abstract list l is stored in memory m from address a onwards.

$$\begin{array}{rcl}
\mathsf{list}\ (\mathsf{nil},a,m) & = & a = 0 \\
\mathsf{list}\ (x{::}l,a,m) & = & \exists a'.\ m(a) = a' \wedge m(a{+}4) = x \wedge a \neq 0 \wedge \\
 & & \qquad \mathsf{list}\ (l,a',m) \wedge \mathsf{aligned}\ a
\end{array}$$

Let length l be the length of an abstract list l, e.g. length $(4{::}5{::}\mathsf{nil}) = 2$. It is easy (15 lines of HOL4) to prove, by induction on the abstract list l, that the function f, from above, calculates the length of a linked list and also that list implies the precondition $\mathsf{f}_{\mathsf{pre}}$.

$$\forall x\, l\, a\, m.\quad \mathsf{list}\ (l,a,m) \implies \mathsf{f}(x,a,m) = (\mathsf{length}\ l, 0, m) \tag{1.2}$$

$$\forall x\, l\, a\, m.\quad \mathsf{list}\ (l,a,m) \implies \mathsf{f}_{\mathsf{pre}}(x,a,m) \tag{1.3}$$

Given (1.2) and (1.3), it follows immediately from (1.1) that the ARM code calculates the length of a linked-list correctly:

$$\{\,(\mathsf{r0},\mathsf{r1},\mathsf{m})\ \mathsf{is}\ (r_0,r_1,m) * \mathsf{s} * \mathsf{pc}\ p * \langle \mathsf{list}\ (l,r_1,m)\rangle\,\}$$
$$p : \texttt{E3A00000},\texttt{E3510000},\texttt{12800001},\texttt{15911000},\texttt{1AFFFFFB}$$
$$\{\,(\mathsf{r0},\mathsf{r1},\mathsf{m})\ \mathsf{is}\ (\mathsf{length}\ l,0,m) * \mathsf{s} * \mathsf{pc}\ (p+20)\,\}$$

1.3.2.2 Implementation.

The following loop-introduction rule is the key idea behind our decompiler implementation. This rule can introduce any tail-recursive function tailrec, with safety condition tailrec_pre, of the form:

$$\begin{aligned} \mathsf{tailrec}\ x &= \mathsf{if}\ G\ x\ \mathsf{then}\ \mathsf{tailrec}\ (F\ x)\ \mathsf{else}\ (D\ x) \\ \mathsf{tailrec\_pre}\ x &= Q\ x \wedge (G\ x \implies \mathsf{tailrec\_pre}\ (F\ x)) \end{aligned}$$

Given a theorem for the step case, $\{r(x)\}\ c\ \{r(F\ x)\}$, and one for the base case, $\{r(x)\}\ c\ \{r'(D\ x)\}$, the loop rule can introduce tailrec:

$$\begin{aligned} \forall r\ r'\ c.\ &(\forall x.\ Q\ x \wedge G\ x \implies \{r(x)\}\ c\ \{r(F\ x)\}) \wedge \\ &(\forall x.\ Q\ x \wedge \neg G\ x \implies \{r(x)\}\ c\ \{r'(D\ x)\}) \\ &\implies (\forall x.\ \mathsf{tailrec\_pre}\ x \implies \{r(x)\}\ c\ \{r'(\mathsf{tailrec}\ x)\}) \end{aligned}$$

Parameters F, D, G, Q, r, r' were instantiated as follows for introduction of g in our example above.

$$\begin{aligned} F = D &= \lambda(r_0,r_1,m).\ \mathsf{if}\ r_1 = 0\ \mathsf{then}\ (r_0,r_1,m)\ \mathsf{else}\ (r_0{+}1, m(r_1), m) \\ G &= \lambda(r_0,r_1,m).\ \mathsf{if}\ r_1 = 0\ \mathsf{then}\ \mathsf{F}\ \mathsf{else}\ \mathsf{T} \\ Q &= \lambda(r_0,r_1,m).\ \mathsf{if}\ r_1 = 0\ \mathsf{then}\ \mathsf{T}\ \mathsf{else}\ (r_1 \in \mathsf{domain}\ m \wedge \mathsf{aligned}\ r_1) \\ r &= \lambda(r_0,r_1,m).\ (\mathsf{r0},\mathsf{r1},\mathsf{m})\ \mathsf{is}\ (r_0,r_1,m) * \mathsf{s} * \mathsf{pc}\ p \\ r' &= \lambda(r_0,r_1,m).\ (\mathsf{r0},\mathsf{r1},\mathsf{m})\ \mathsf{is}\ (r_0,r_1,m) * \mathsf{s} * \mathsf{pc}\ (p+20) \end{aligned}$$

The loop rule can be derived from the rule for composition of Hoare triples given in the previous section. For details of decompilation, see [22, 24].

1.3.3 Extensible proof-producing compilation

It is often the case that we prefer to synthesise ARM code from specifications rather than apply post hoc verification to existing ARM code. For this purpose we have developed a proof-producing compiler [25] which maps tail-recursive functions in HOL4, i.e. functional specifications, to ARM machine code and proves that the ARM code is a valid implementation of the original HOL4 functions.

1.3.3.1 Example 1.

Given a function f,

$$\mathsf{f}(r_1) = \mathsf{if}\ \ r_1 < 10\ \ \mathsf{then}\ \ r_1\ \ \mathsf{else}\ \ \mathsf{let}\ r_1 = r_1 - 10\ \mathsf{in}\ \mathsf{f}(r_1)$$

the compiler produces machine code,

```
0: E351000A   L: cmp r1,#10        ; compare reg 1 with 10
4: 2241100A      subcs r1,r1,#10   ; if less: subtract 10 from reg 1
8: 2AFFFFFC      bcs L             ;       jump to compare
```

and proves that the generated code calculates f:

$$\begin{array}{l}\{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\} \\ p : \texttt{E351000A}, \texttt{2241100A}, \texttt{2AFFFFFC} \\ \{\mathsf{r1}\ \mathsf{f}(r_1) * \mathsf{pc}\ (p{+}12) * \mathsf{s}\}\end{array}$$

In case we have manually proved that f calculates unsigned-word modulus of 10, i.e. $\forall x.\ \mathsf{f}(x) = x \bmod 10$, then we immediately know that the ARM code calculates modulus of 10:

$$\begin{array}{l}\{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\} \\ p : \texttt{E351000A}, \texttt{2241100A}, \texttt{2AFFFFFC} \\ \{\mathsf{r1}\ (r_1 \bmod 10) * \mathsf{pc}\ (p{+}12) * \mathsf{s}\}\end{array}$$

1.3.3.2 Example 2.

An important feature of this compiler is its support for extensions. If the compiler is supplied with the above theorem which states that the ARM instructions `E351000A 2241100A 2AFFFFFC` together assign $r_1 \bmod 10$ to r_1, then subsequent compilations can make use of this verified code. For example,

$$\begin{array}{rl}\mathsf{f}(r_1, r_2, r_3) = & \mathsf{let}\ r_1 = r_1 + r_2\ \mathsf{in} \\ & \mathsf{let}\ r_1 = r_1 + r_3\ \mathsf{in} \\ & \mathsf{let}\ r_1 = r_1 \bmod 10\ \mathsf{in} \\ & \quad r_1\end{array}$$

will compile successfully into a theorem which makes use of the previously verified code (the last three instructions in the code below):

$$\begin{array}{l}\{\mathsf{r1}\ r_1 * \mathsf{r2}\ r_2 * \mathsf{r3}\ r_3 * \mathsf{pc}\ p * \mathsf{s}\} \\ p : \texttt{E0811002}, \texttt{E0811003}, \texttt{E351000A}, \texttt{2241100A}, \texttt{2AFFFFFC} \\ \{\mathsf{r1}\ (\mathsf{f}(r_1, r_2, r_3)) * \mathsf{r2}\ r_2 * \mathsf{r3}\ r_3 * \mathsf{pc}\ (p{+}20) * \mathsf{s}\}\end{array}$$

1.3.3.3 Implementation.

The compiler is implemented using translation validation based on the decompiler from above; for each function f, the compiler will:

1. generate machine code for input function f with an unverified algorithm;
2. decompile the generated code into a function f';
3. automatically prove $\mathsf{f} = \mathsf{f}'$.

The compiler returns to the user, the theorem certificate produced in step 2, but with f' replaced by f using the rewrite theorem produced in step 3.

The compiler's separation between code generation (step 1) and certification (steps 2 and 3) has two useful consequences: code generation need not be proof-producing, and multiple lightweight optimisations can be made in step 1 with practically no added proof burden for steps 2 and 3. Step 1 is allowed to produce any code for step 3 will be able to prove $\mathsf{f} = \mathsf{f}'$. For example, just doing expansion of let expressions in step 3 immediately makes optimisation such as register renaming, some instruction reordering and dead-code removal unobservable.

Extensions are implemented by making the decompiler use the theorems provided when constructing the step- and base-theorems for instantiating its loop rule, as explained in the previous section.

1.3.4 Case study: verified LISP interpreter

The construction of a verified LISP interpreter [23] is the, to date, largest case study conducted on top of the ARM model. This case study included producing and verifying implementations for:

- a copying garbage collector
- an implementation of basic LISP evaluation
- a parser and printer for s-expressions

These components were combined to produce an end-to-end implementation of a LISP-like language, similar to the core of the original LISP 1.5 by McCarthy [21].

1.3.4.1 Example.

For a flavour of what we have implemented and proved consider an example: if our implementation is supplied with the following call to function `pascal-triangle`,

```
(pascal-triangle '((1)) '6)
```

it parses the string, evaluates the expression and prints a string,

```
((1 6 15 20 15 6 1)
 (1 5 10 10 5 1)
```

```
(1 4 6 4 1)
(1 3 3 1)
(1 2 1)
(1 1)
(1))
```

where `pascal-triangle` had been supplied to it as

```
(label pascal-triangle
  (lambda (rest n)
    (cond ((equal n '0) rest)
          ('t (pascal-triangle
                (cons (pascal-next '0 (car rest)) rest) (- n '1))))))
```

with auxiliary function:

```
(label pascal-next
  (lambda (p xs)
    (cond ((atomp xs) (cons p 'nil))
          ('t (cons (+ p (car xs)) (pascal-next (car xs) (cdr xs)))))))
```

The theorem we have proved about our LISP implementation can be used to show e.g. that running `pascal-triangle` will terminate and print the first $n+1$ rows of Pascal's triangle, without a premature exit due to lack of heap space. One can use our theorem to derive sufficient conditions on the inputs to guarantee that there will be enough heap space.

1.3.4.2 LISP evaluation.

The most interesting part of this case study is possibly the construction of verified code for LISP evaluation. For this we used our extensible compiler, described above.

First, the compiler's input language was extended with theorems that provide ARM code that performs LISP primitives, car, cdr, cons, equal, etc. These theorems make use of an assertion lisp, which states that a heap of s-expressions $v_1 \ldots v_6$ is present in memory. For car of s-expressions v_1, we have the theorem:

$$
\begin{array}{l}
\mathsf{is\_pair}\ v_1 \implies \\
\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p\ \} \\
\quad p : \texttt{E5933000} \\
\{\ \mathsf{lisp}\ (\mathsf{car}\ v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p+4)\ \}
\end{array}
$$

The cons primitive was the hardest one to construct and prove correct, since the implementation of cons contains the garbage collector: cons is guaranteed to succeed whenever the size of all live s-expressions is less than the heap limit l.

$$
\begin{array}{l}
\mathsf{size}\ v_1 + \mathsf{size}\ v_2 + \mathsf{size}\ v_3 + \mathsf{size}\ v_4 + \mathsf{size}\ v_5 + \mathsf{size}\ v_6 < l \implies \\
\{\ \mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p\ \} \\
\quad p : \texttt{E50A3018 E50A4014 E50A5010 ... E51A8004 E51A7008} \\
\{\ \mathsf{lisp}\ (\mathsf{cons}\ v_1\ v_2, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p+332)\ \}
\end{array}
$$

The above mentioned theorems extend the compiler input language with:

$$\mathsf{let}\ v_1 = \mathsf{car}\ v_1\ \mathsf{in} \qquad \text{and} \qquad \mathsf{let}\ v_1 = \mathsf{cons}\ v_1\ v_2\ \mathsf{in}$$

Once the compiler understood enough LISP primitives, we defined lisp_eval as a lengthy tail-recursive function, and used the compiler to synthesise ARM code for implementing lisp_eval.

In order to verify the correctness of lisp_eval, we proved that lisp_eval will always evaluate s to r in environment ρ whenever a clean relation semantics for LISP evaluation, which had been developed in unrelated previous work [11], evaluates s to r in environment ρ, written $(s,\rho) \rightarrow_{eval} r$. Here s-expression nil initialises variables v_2, v_3, v_4 and v_6; functions t and u are translation functions from one form of s-expression into another.

$$\forall s\ \rho\ r.\ \ (s,\rho) \rightarrow_{eval} r \implies \mathsf{fst}\ (\mathsf{lisp\_eval}\ (t\ s, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, u\ \rho, \mathsf{nil}, l)) = t\ r$$

1.3.4.3 Parsing and printing.

The heap of s-expressions defined within the lisp assertion used above is non-trivial to set up. Therefore we constructed verified code for setting up and tearing down a heap of s-expressions. The set-up code also parses s-expressions stored as a string in memory, and sets up a heap containing that s-expression. The tear down code prints into a buffer in memory, the string representation of an s-expression from the heap. The code for set-up/tear-down, parsing/printing, was again synthesised from functions in the HOL4 logic.

1.3.4.4 Final correctness theorem.

By composing theorems for parsing, evaluation and printing we get the final correctness theorem: if $\rightarrow_{eval}$ relates s with r under the empty environment (i.e. $(s,[]) \rightarrow_{eval} r$), no illegal symbols are used (i.e. sexp_ok $(t\ s)$), running lisp_eval on $t\ s$ will not run out of memory (i.e. lisp_eval_pre$(t\ s, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, l)$), the string representation of $t\ s$ is in memory (i.e. string a (sexp2string $(t\ s)$)), and there is enough space to parse $t\ s$ and set up a heap of size l (i.e. enough_space $(t\ s)\ l$), then the code will execute successfully and terminate with the string representation of $t\ r$ stored in memory (i.e. string a (sexp2string $(t\ r)$)). The ARM code expects the address of the input string to be in register 3, i.e. r3 a.

$$\begin{array}{l}
\forall s\ r\ l\ p.\\
\quad (s,[]) \rightarrow_{eval} r \wedge \mathsf{sexp_ok}\ (t\ s) \wedge \mathsf{lisp_eval_pre}(t\ s, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, \mathsf{nil}, l) \implies\\
\quad \{\ \exists a.\ \mathsf{r3}\ a * \mathsf{string}\ a\ (\mathsf{sexp2string}\ (t\ s)) * \mathsf{enough_space}\ (t\ s)\ l * \mathsf{pc}\ p\ \}\\
\quad p : \ldots \text{ code not shown } \ldots\\
\quad \{\ \exists a.\ \mathsf{r3}\ a * \mathsf{string}\ a\ (\mathsf{sexp2string}\ (t\ r)) * \mathsf{enough_space'}\ (t\ s)\ l * \mathsf{pc}\ (p{+}10404)\ \}
\end{array}$$

We have also proved this result for similar x86 and PowerPC code. Our verified LISP implementations run can be run on ARM, x86 and PowerPC hardware.

1.4 Conclusions and future research

The ARM verification project has been a fairly modest scale effort: one person full-time specifying and verifying the hardware (Fox) and one to two part time researchers looking at software and the background mathematical theories (Hurd, Myreen). In addition, some students have spent time assisting the research, namely Scott Owens, Guodong Li and Thomas Tuerk.

The project aims to verify systems built out of commercial-off-the-shelf components where everything – microarchitecture up to abstract mathematics – is formalised within a single framework. The research is still in progress and, unlike the celebrated CLI Stack [20], we have not yet completely joined up the various levels of modelling, but this remains our goal. Unlike most other work, we have used a COTS processor and have tried (and are still trying) to formally specify as much as possible, including difficult features like input/output and interrupts. The closest work we know of is the verification of security properties of the Rockwell Collins AAMP7G processor [14, 13]. More on AAMP7G can be found in other chapters of this book.

Even though the ARM ISA is relatively simple, the low-level details can overwhelm verification attempts. During the project we have found that it is important to abstract as much as possible so that proofs are not cluttered with such details. A key tool for this has been the derivation of a next-state function for CPU-memory combinations which then can be used to derive clean semantic specifications for instruction uses-cases and then support a further abstraction to Hoare-like rules for machine code segments, with the frame problem managed via a separating conjunction. Some of the technical details pertaining to this abstraction methodology are sketched in the preceding two sections.

Although our formal specifications include input/output, interrupts and facilities for modelling complex memory models, we have yet (2009) to demonstrate significant verification case studies that use these. Our current work aims to create a complete functional programming platform on bare metal, with high-fidelity modelling of system level timing and communication with the environment. We expect that achieving this will take several more years of research at the current level of effort.

References

1. Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In David Basin and Burkhart Wolff, editors, *Proceedings of the 4th ACM SIGPLAN workshop on Declarative Aspects of Multicore Programming*, pages 13–24. Association for Computing Machinery, 2009.
2. ARM Ltd. Jazelle technology. `http://www.arm.com/products/multimedia/java/jazelle.html`, (accessed in July 2009).
3. Jerry Burch and David Dill. Automatic verification of pipelined microprocessor control. pages 68–80. Springer-Verlag, 1994.
4. Anthony C. J. Fox. *Algebraic Models for Advanced Microprocessors*. PhD thesis, University of Wales, Swansea, 1998.
5. Anthony C. J. Fox. An algebraic framework for modelling and verifying microprocessors using HOL. Technical Report 512, University of Cambridge Computer Laboratory, April 2001.
6. Anthony C. J. Fox. A HOL specification of the ARM instruction set architecture. Technical Report 545, University of Cambridge Computer Laboratory, June 2001.
7. Anthony C. J. Fox. Formal verification of the ARM6 micro-architecture. Technical Report 548, University of Cambridge, Computer Laboratory, 2002.
8. Anthony C. J. Fox. Formal specification and verification of ARM6. In David Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 25–40. Springer-Verlag, 2003.
9. Anthony C. J. Fox. An algebraic framework for verifying the correctness of hardware with input and output: a formalization in HOL. In José Luiz Fiadeiro, Neil Harman, Markus Roggenbach, and Jan J. M. M. Rutten, editors, *CALCO 2005*, volume 3629 of *Lecture Notes in Computer Science*, pages 157–174. Springer-Verlag, 2005.
10. Steve Furber. *ARM: system-on-chip architecture*. Addison-Wesley, second edition, 2000.
11. Mike Gordon. Defining a LISP interpreter in a logic of total functions. In *the ACL2 Theorem Prover and Its Applications (ACL2)*, 2007.
12. Mike J. C. Gordon. Proving a computer correct with the LCF-LSM hardware verification system. Technical Report 42, University of Cambridge Computer Laboratory, 1983.
13. David Greve, Raymond Richards, and Matthew Wilding. A Summary of Intrinsic Partitioning Verification. In *ACL2 Workshop 2004*, November 2004.
14. David Greve, Matthew Wilding, and W. Mark Vanfleet. A Separation Kernel Formal Security Policy. In *ACL2 Workshop 2003*, June 2003.
15. David Hardin. Invited Tutorial: Considerations in the Design and Verification of Microprocessors for Safety-Critical and Security-Critical Applications. In *Proceedings of FMCAD 2008*, November 2008.
16. Neal A. Harman and John V. Tucker. Algebraic models of microprocessors: The verification of a simple computer. In Vicky Stavridou, editor, *Mathematics of Dependable Systems II*, pages 135–170. Oxford University Press, 1997.
17. John R. Harrison. A HOL theory of Euclidean space. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 114–129, Oxford, UK, 2005. Springer-Verlag.
18. Joe Hurd. Formalizing elliptic curve cryptography in higher order logic. Available from the author's website, October 2005.
19. Joe Hurd, Mike Gordon, and Anthony Fox. Formalized elliptic curve cryptography. In *High Confidence Software and Systems: HCSS 2006*, April 2006.
20. J Strother Moore (foreward). *Special issue on Systems Verification*, volume 5. 1989.
21. John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. The MIT Press, 1966.

22. Magnus O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2009.
23. Magnus O. Myreen. Verified implementation of LISP on ARM, x86 and PowerPC. In *Theorem Proving in Higher-Order Logics (TPHOLs)*. Springer, 2009.
24. Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Machine-code verification for multiple architectures – An application of decompilation into logic. In *Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2008.
25. Magnus O. Myreen, Konrad Slind, and Michael J.C. Gordon. Extensible proof-producing compilation. In *Compiler Construction (CC)*. Springer, 2009.
26. John Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of Logic in Computer Science (LICS)*. IEEE Computer Society, 2002.
27. Jun Sawada and Warren A. Hunt, Jr. Verification of fm9801: An out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability. *Form. Methods Syst. Des.*, 20(2):187–222, 2002.
28. Daniel Schostak. *Methodology for the Formal Specification of RTL RISC Processor Designs (with particular reference to the ARM6)*. PhD thesis, University of Leeds, 2003.
29. Konrad Slind. TFL: An environment for terminating functional programs. `http://www.cl.cam.ac.uk/~ks121/tfl.html`, (accessed in July 2009).
30. Laurent Thery. Proving the group law for elliptic curves formally. Technical Report RT-0330, INRIA, 2007.
31. Wai Wong. Formal verification of VIPER's ALU. Technical Report 300, University of Cambridge Computer Laboratory, April 1983.

Index