

The unforeseen evolution of an ARM verification project

Mike Gordon

June 19, 2015

Abstract

The story of how a project to formally verify an ARM processor evolved and changed focus over fifteen years is told here. I have tried to make the story accessible to a general audience: no detailed knowledge of formal verification or theorem proving is assumed. I hope to illustrate by example how long it can take for research to have any impact and the difficulty of predicting what the impact will be.

The Leeds-Cambridge ARM6 verification project took place between 2000 and 2004. Leeds' goal was to specify the ARM instruction set and the ARM6 microprocessor implementation. Cambridge's goal was to use automated theorem proving to verify that the processor correctly implemented ARM instructions. The experience gained from this project resulted in changed priorities for subsequent research, which was possible as a result of continuing funding from the USA. New methods for coupling processor specifications to theorem proving tools emerged and the goals expanded to include software verification. Eventually the work started to find applications at Cambridge and elsewhere. These applications range from proving properties of operating system machine code to exploring new security enhanced processor designs.

This document was prepared partly as background for a talk given at Swansea University and partly as a report for some of the funders of the research. It is not intended for publication. The scientific contributions are due to Anthony Fox, Magnus Myreen and others who devised and worked on the applications. My role has mainly been administrative. I retire on September 30, 2015, but the research continues as part of the EPSRC REMS project. If I mention your work (e.g. by including material from online papers or web pages) and if you are unhappy and would like me to change, add or delete anything, then please accept my apologies and email me with the changes that you'd like me to make.

Contents

1	Background	3
2	The first EPSRC project	5
2.1	What does it mean to formally verify a processor	6
2.1.1	Modelling the execution of machine instructions	6
2.1.2	Modelling a processor implementation	7
2.1.3	Proving instructions are correctly executed	8
2.2	Results of the ARM6 verification project	11
3	The second EPSRC project	12
3.1	Formal verification of machine code	12
3.2	Myreen decompilation	13
3.3	Accurate model of ARM ISA version ARMv4T	16
3.4	Achievements of the second EPSRC project	17
4	From 2007 to the present	17
4.1	New ARM ISA models	18
4.2	First impact of ARM models emerge	19
4.2.1	GammaTech’s static analyser	19
4.2.2	NICTA’s seL4 microkernel verification	20
4.2.3	KTH PROSPER project	21
4.3	The ISA specification language L3	22
4.4	The D-RisQ project	25
4.5	CakeML	26
4.6	MIPS-based processors: BERI and CHERI	28
4.7	Using HOL and L3 ARM models for testing	30
5	Reflections	30
6	Acknowledgements	32
	References	32

1 Background

Since the 1980s I've had an incredibly productive collaboration on hardware modelling and verification with Graham Birtwistle. We exchanged ideas, PhD students and postdocs, and jointly developed theorem proving software (early versions of the HOL system [1, 2]). At the start of our collaboration Graham was a professor at the University of Calgary in Canada. In the 1990s Graham moved to Leeds University and became interested in modelling ARM¹ processors. He invited me to join in research that he was discussing with ARM. This resulted in an EPSRC project proposal with Leeds, Cambridge and ARM as partners. The box below contains the project summary from the application form.

We propose to help develop a methodology for the production of executable specifications of microprocessors cores at both the instruction set and pipelined architectural levels and to use formal verification to establish equivalence or refinement relationships between these specifications. The key technical challenge is to show that academic verification techniques developed in the 1980's for microprocessor subsystems scale to realistic complete architectures. The project is a partnership between specifiers and verifiers at Leeds and Cambridge, and designers and engineers from ARM who will actively participate by providing understanding and insights, and reading and critiquing specifications as they emerge. We concentrate upon the ARM6 microprocessor to give the project a concrete focus, but the results and methods will be generally applicable.

The ARM6 microprocessor was chosen because ARM determined that it did not to contain sensitive ARM IP.² The initial EPSRC proposal was turned down on the grounds that verifying an ARM6 was uninteresting research because the processor was obsolete and we were invited to resubmit a similar project based around a current ARM processor.

¹Here and elsewhere “ARM” refers to the company ARM Holdings plc. Terms like “ARM6” and “ARMv3”, which refer to particular ARM products, will be explained as needed.

²The ARM610 implementation (a member of the ARM6 family) was used in the Apple Newton “personal digital assistant” (PDA). The phrase “personal digital assistant” was coined during the Newton’s \$100M development by Apple’s then CEO John Sculley [3]. A recent article in Wired Magazine [4] describes the Apple Newton as a “prophetic failure” .

For the resubmission ARM provided us with a letter explaining why the designs of their current processors were too sensitive to pass to us. I don't have the original source text, but here is an image of an extract from the letter.

However, there are IP issues to consider in all of this, and it seems that these issues are at the heart of EPSRC's recent rejection of Graham and Mikes first proposal. As a company, our IP is our only stream of business, and as such we may appear overly paranoid about leaking this IP into the public domain. ARM7, although several years old now, and licensed to all of the ARM partners (approx. 35 to date), is still considered to be our cash-cow in terms of licensing. It has found its way into many consumer products such as Mobile Phones and hand-held computers and is still being shipped in huge volumes by partners such as TI. ARM9 is expected to be the successor to ARM7, finding its way into similar applications with increasing volumes. Putting detailed specifications of these products into the public domain would pose a serious threat to ARM's business in our opinion. Although protected by copyright law, there is often little that can be practically done when a competitor clones the ARM architecture, especially when done overseas. I doubt that ARM will change its position on this issue regarding the ARM7 and ARM9 architectures. However, we are very confident that the ARM6 architecture will prove to be a challenging test-case for the proposed work.

This refers to various implementations of the ARM instruction set architecture (ISA). ARM has a somewhat confusing naming convention for the various versions of ISA and the various hardware implementations. The ISA executed by the ARM6 implementation is called ARMv3. Later ARM processors were, for example ARM7 and ARM8 (as mentioned in the letter extract above). Later instruction sets are, for example, ARMv5, ARMv6, ARMv7. Implementations like ARM6 are subdivided into cores, which are intended for different applications. The ARM6 implementation family was implemented by three cores: ARM60, ARM600, ARM610 which all execute instructions specified by ARMv3, but have some different hardware details (e.g. ARM60 has no cache). A Wikipedia article gives the full list of ARM instruction architectures, implementations families and cores [5].

ARM's letter did the trick. The proposal was resubmitted and became funded in the year 2000 [6].

I what follows §2 to §4 tell what happened in the next fifteen years, starting with the EPSRC project; §5 then reviews and reflects on how the project evolved.

2 The first EPSRC project

The first EPSRC project had the title “Formal Specification and Verification of ARM6” and was a collaboration between ARM and the universities of Leeds and Cambridge. ARM’s role was advisory: they would help in understanding the available documentation and provide details that might not be documented. Leeds had the goal of developing executable functional specifications of both the instruction set architecture (ISA) – i.e. the semantics of machine instructions – and the ARM6 processor implementation which executed instructions. Cambridge’s task was to formalise the specifications being developed at Leeds and then to use the HOL4 proof assistant to prove theorems establishing the correctness of the ARM6 implementation.

The research at Leeds was undertaken by two PhD talented students supervised by Graham Birtwistle: Dominic Pajak and Daniel Schostak. Dominic worked on creating a model of the ARMv3 ISA and Daniel on a model of an ARM6 implementation. These models consisted of function definitions in the Standard ML programming language [7, 8] and were thus executable so both could be validated on test data. Both Leeds students spent summers working at ARM in Cambridge where they could do ‘field work’ to discover information that wasn’t available in public documents. This was particularly important for building a model of the ARM6 processor as few details of this were in the public domain (though ARM had agreed for its design to be used in the project). Although the meaning of machine instructions was available in natural language documentation for the ARMv3 ISA [9] there were details and ambiguities that needed to be clarified by talking to experts.

The EPSRC grant provided funds for a research associate to assist with the work at Cambridge. Finding a suitable person might have been a challenge, but fortunately I’d recently been the external PhD examiner for Anthony Fox who had written an outstanding thesis on processor verification and I managed to hire him. Fox’s processor proofs were formulated in an algebraic framework that had been developed by John Tucker and Neil Harman over many years [10] (Harman was Fox’s supervisor). Here is an extract from a

paper by Harman and Tucker [11] outlining their approach.

... a mathematical theory for specifications based on *abstract data types, streams, clocks and retimings*, and *recursive functions* is developed. A specification is a function that transforms infinite streams of data. The mathematical theory supports formal methods and software tools.

Although the Harman-Tucker theory “supports formal methods and software tools”, it had not been adapted for use with the HOL4 theorem prover that was to be used for the ARM6 proofs in the EPSRC project. The micro-processor verifications that Fox carried out during his PhD research were manually generated pencil-and-paper proofs, but they provided an excellent foundation for mechanisation.

Fox’s first task after taking up the research associate position at Cambridge was to learn how to mechanise proofs similar to those in his PhD. This resulted in a tutorial technical report [12] which established the method that would be used to prove theorems about the ARM processors using the HOL4 system. Whilst mechanising his hand proofs, Fox found a few minor errors. This provided reassuring motivation for mechanization. However, the main motivation is the impracticability of scaling up pencil-and-paper proof to real-world examples, even for relatively simple processors like ARM6.

2.1 What does it mean to formally verify a processor

Proving a processor correct consists in three steps:

1. making a mathematical model of the execution of machine instructions,
2. making a mathematical model of the implementation,
3. proving the implementation correctly executes instructions.

These steps are sketched in the following three subsections. For technical details see Fox’s report on the ARM6 verification [13].

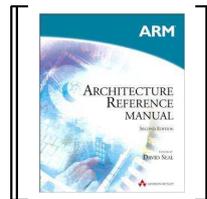
2.1.1 Modelling the execution of machine instructions

The machine instructions for ARM processors are described in a document entitled *ARM Architecture Reference Manual* (AARM) which is available

both as a printed book and online [9]. This uses a mixture of English and pseudo-code to describe the effect of executing machine instructions. Here is a picture of the AARM book:



An early task in the project was to create a representation of the meaning of instructions – their semantics – in a formal logical notation. Fox created a such a representation in the logic supported by the HOL4 theorem-proving system, which will be abbreviated to just “HOL”. He collaborated with Pajak who was building an executable model of ARMv3 instructions in Standard ML, a functional programming language whose programs were similar to HOL terms. Semantic brackets $\llbracket \dots \rrbracket$ around a picture of the AARM will denote Fox’s formalisation of machine instruction semantics in HOL:

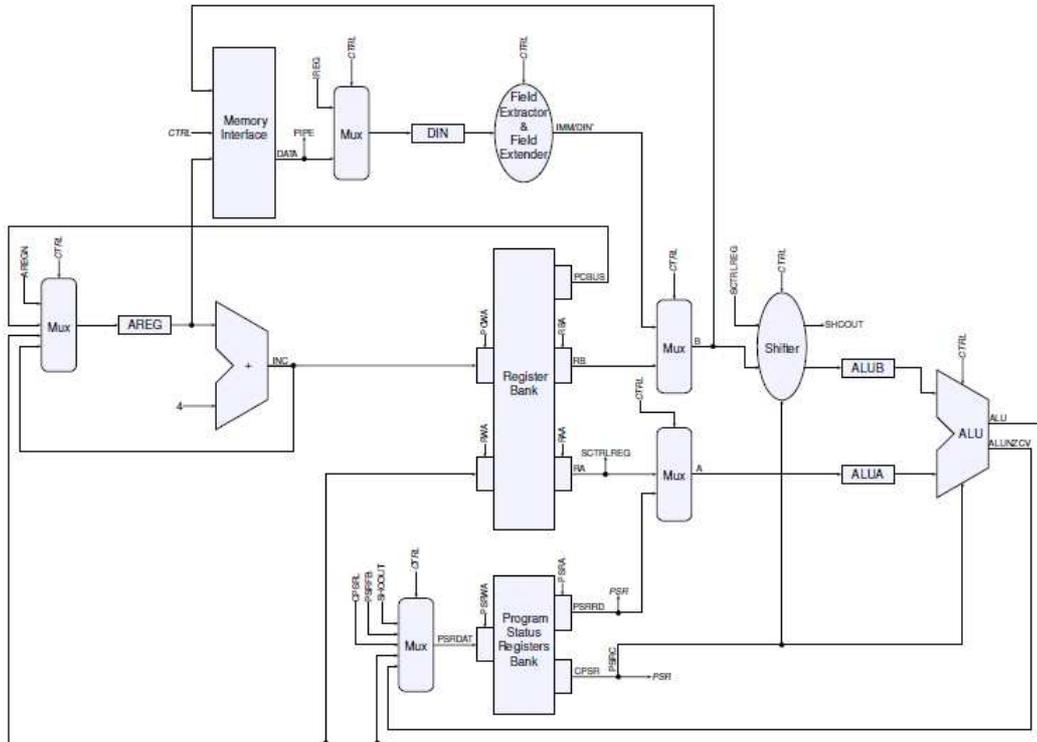


2.1.2 Modelling a processor implementation

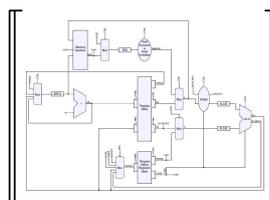
There was no published description of the ARM6 processor. Daniel Schostak put together a specification from various documents he could find and input from current and past ARM engineers. His specification was the basis for the implementation formalised by Fox.

The implementation combines hardware components including memory, registers and an arithmetic logic unit (ALU). The diagram below is from Fox’s ARM6 report [13]. The details are unimportant here, but it is given to

provide an impression of the sort of thing the ARM6 implementation is.



The behaviour of this implementation was represented in HOL using methods Fox developed in his PhD. This representation is denoted by putting semantic brackets around a shrunk image of the illustrative implementation diagram:



2.1.3 Proving instructions are correctly executed

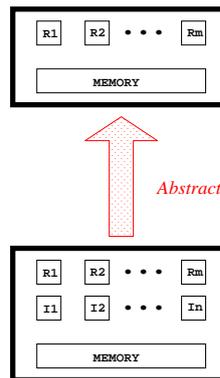
Before proving the correctness of an implementation one must first formulate what correctness means. This can be quite tricky and it is critical to have a good formulation: proving a theorem that does not accurately represent

what correctness means in practice could yield a misleading ‘false positive’ result.

Both the ARMv3 instruction semantics and the ARM6 implementation are represented in HOL by state machines. Each such machine consists of a set of states and a next-state function to represent how states change when instructions are executed.

The kind of states needed to model the effect of executing machine instructions, as specified by the ISA, is different from the kind of states needed to model the actual hardware implementation. There will be many registers in the implementation that are not visible in the ISA which may hold intermediate results, control state etc.

A common way to relate ISA and implementation states is via a function that abstracts the latter to the former. An over-simplified example is where the implementation just has extra registers, e.g. I_1, \dots, I_n in the diagram below. The abstraction function might just discard these registers.



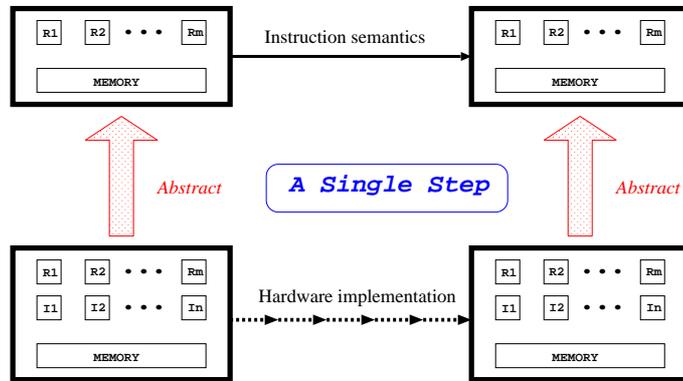
The abstraction from ARM6 states to the ARMv3 ISA states is hugely more complex than this. In addition to registers in the implementation that are not visible by the ISA, there are ISA registers that are split into separate hardware components of the implementation in ways that are not apparent at the ISA level.

As well as the implementation and ISA having different states, they also have different next-state functions. A single step at the ISA level typically consists of executing a single machine instruction. A single step in the implementation will typically be a micro-step, several of which are needed for implementing each machine instruction. Proving the implementation correct consists in

showing that the result of the sequence of micro-steps corresponding to each machine instruction has the correct effect on the ISA state.

The standard proof strategy is to show that for any machine instruction and pair of states s_{ISA} , s_{imp} where s_{ISA} is a state of the ISA, s_{imp} a state of the implementation and $\text{Abstract}(s_{\text{imp}}) = s_{\text{ISA}}$, then if the ISA model predicts that executing the instruction results in state s'_{ISA} and the processor model predicts that executing the instruction (which might take several micro-steps) results in state s'_{imp} , then $\text{Abstract}(s'_{\text{imp}}) = s'_{\text{ISA}}$.

Such arguments are often illustrated with a ‘commuting diagram’, such as:

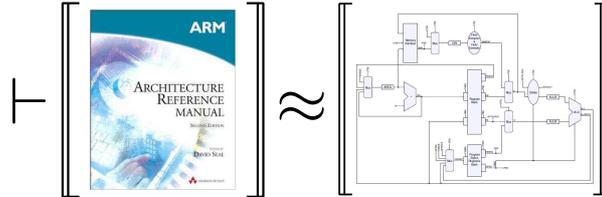


This diagram ‘commutes’ if going from the bottom left to the top right via the two possible paths – i.e. abstract-then-execute-instruction-semantics or execute-hardware-steps-then-abstract – results in the same state.

The proof that such a diagram commutes is by symbolic simulation along each path for each instruction and then verifying the results agree. Symbolic simulation consists of ‘executing’ each instruction by mechanically simplifying a logic term representing the instruction applied to an arbitrary state in which the values of memory and registers are represented symbolically by variables. For example, a term representing state s_{ISA} may contain $\text{R1} = x$ and $\text{R2} = y$ specifying symbolic values x, y for registers **R1**, **R2**, respectively. One step of symbolic simulation of a instruction to add registers **R1** and **R2** and put the result in register **R3** might result in a logical term representing a symbolic state s'_{ISA} containing $\text{R3} = x + y$.

A proof by symbolic simulation verifies the execution of all instructions in all states. It is sometimes called static verification in contrast with dynamic verification which runs the model on particular instructions in particular

states to find bugs. The proportion of instructions executed to all instructions is the coverage. Static verification can be thought of as giving 100% coverage. If \approx is a HOL relation that formalises the property that the ISA and implementation are related by the diagram above commuting, then the ARM6 correctness theorem can be visualised as below ($\vdash t$ means t is provable).



This section oversimplifies almost every detail. It merely aims to give a first impression of what the ARM6 verification consists of. The full story is in Fox’s report [13].

2.2 Results of the ARM6 verification project

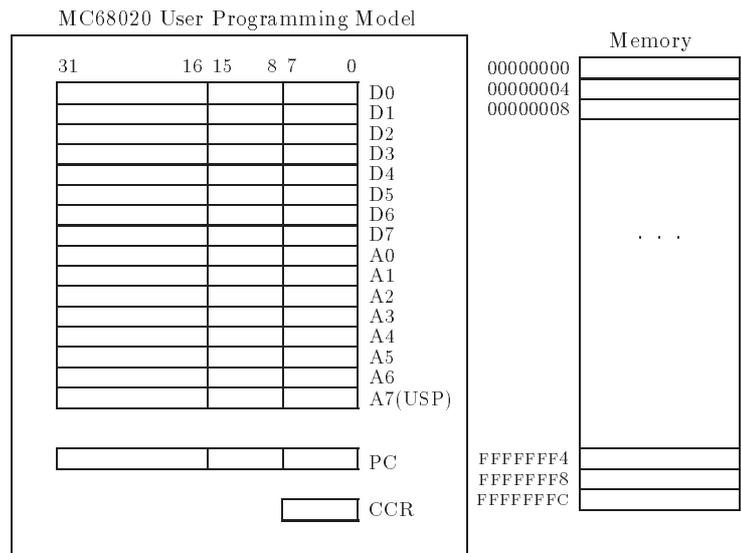
The project was a success. It established the feasibility of verifying the correctness of ARM-sized processors by interactive theorem proving. Fox took about a year to prove that a model of the ARM6 processor correctly implemented the ARMv3 ISA, though this was spread over a longer period as the work first needed a mechanisation of the Fox-Harman-Tucker algebraic method to be developed, and then lots of supporting theorem-proving infrastructure to be implemented to. Verifying another similar processor should be much quicker with this methodology and infrastructure in place. A natural next step might have been to test this prediction by actually verifying another ARM processor, say ARM6, but this wasn’t possible since for commercial reasons ARM would not be willing to provide details of the implementation. Even if ARM had been willing to provide details of more recent processor implementations, the effort of repeating a similar verification would have been hard to justify as novel research. It was therefore decided to cease hardware verification and focus on proving machine code programs correct by building on the ISA modelling methods and infrastructure that had been created.

3 The second EPSRC project

The EPSRC supported a follow-on project entitled “Formal Specification and Verification of ARM-Based Systems” (EPSRC project, 2004-2007). The goal was to use the verified ARM ISA model as foundation for verifying software.

3.1 Formal verification of machine code

The initial idea was to build on earlier pioneering research by Boyer and Yu [14] for verifying machine code programs for the MC68020 processor. This worked by first proving an abstract specification of an algorithm corresponds to a concrete machine code implementation on an ISA model of the processor, and then proving the abstract specification has desired properties. The proofs were by symbolic simulation and induction (for loops). Here is a diagram of MC68020 taken from the Boyer-Yu paper.



Very impressive small examples were done (“Euclid’s GCD, Hoare’s Quick Sort, binary search, and other well-known algorithms”) but the approach was hard to scale. This is because for each example the proof that the abstract specification corresponds to the execution of concrete machine code on the processor involves the entire processor model, including those parts not relevant to the code being verified. One is faced by an instance of the ‘Frame Problem’ – e.g. specifying and then proving which values of registers,

memory locations etc. are not changed by the computation.

Fortunately, by combining ideas from Boyer and Yu with concepts from Separation Logic [15], Magnus Myreen – then a new PhD student at Cambridge – came up with a new method based on automatic decompilation that has turned out to scale well.

3.2 Myreen decompilation

Myreen’s idea is to precompute verified specifications of the semantics of each machine instruction in the form of ‘machine code Hoare triples’ $\{P\}C\{Q\}$ [16] in which C is a machine instruction, the precondition P specifies only the resources (registers, memory etc.) in the fragment of the state relevant to executing C and the postcondition Q specifies the state of these resources after executing C . Myreen’s Hoare triples are inspired by Separation Logic [15] in that they are ‘small’: the precondition P determines the ‘footprint’ of C and doesn’t constrain the state outside this. Additional invariants are added using the Frame Rule of Separation Logic. The idea is to split a large monolithic ISA model into small separate behavioural specifications of each instruction. The verification of machine code programs then only needs to use the specifications of the instructions it uses.

To illustrate Myreen’s approach, the specification of some simplified ARM-like machine instructions are given below. The notation

$$\left[\begin{array}{c} \text{ARM} \\ \text{ARCHITECTURE} \\ \text{REFERENCE} \\ \text{MANUAL} \end{array} \right] \vdash \begin{array}{c} \{P\} \\ C \\ \{Q\} \end{array}$$

means that $\{P\} C \{Q\}$ is proved from the ARM ISA model.

The first example is the derived specification of `MOV b,a`. This instruction moves the contents of register a to register b . It doesn’t change the contents of a . The operator “*” is separating conjunction from Separation Logic (more on this later).

$$\left[\begin{array}{c} \text{ARM} \\ \text{ARCHITECTURE} \\ \text{REFERENCE} \\ \text{MANUAL} \end{array} \right] \vdash \begin{array}{c} \{R\ a\ x * R\ b\ _ * R\ pc\ p\} \\ \text{MOV } b,a \\ \{R\ a\ x * R\ b\ x * R\ pc\ (p+1)\} \end{array}$$

In this example, the precondition $\{R a x * R b _ * R pc p\}$ specifies that register a holds the value x , register b holds some value³ and the program counter, which in ARM is held in a register pc , has the value p . The postcondition $\{R a x * R b x * R pc (p+1)\}$ specifies the state after executing the instruction `MOV b,a`, namely register b holds x (i.e. the value in a before the instruction was executed), register a is unchanged and the program counter is incremented to point to the next instruction.⁴

The instructions `MUL b,a1,a2` and `SUBS b,a1,a2` in the box below perform multiplication and subtraction, respectively, on the contents of registers a_1 and a_2 and put the result in register b . The `SUBS` instruction also sets a status flag (the Z flag) if the result of the subtraction is zero. The statement `S b` asserts that the status is b (a Boolean). The instruction `BNE #-n` jumps back n instructions if the status flag is not false, otherwise it does nothing.

	$\{R a_1 x_1 * R a_2 x_2 * R b _ * R pc p\}$ \vdash <code>MUL b,a1,a2</code> $\{R a_1 x_1 * R a_2 x_2 * R b (x_1 \times x_2) * R pc (p+1)\}$
	$\{R a_1 x_1 * R a_2 x_2 * R b _ * S _ * R pc p\}$ \vdash <code>SUBS b,a1,a2</code> $\{R a_1 x_1 * R a_2 x_2 * R b (x_1 - x_2) * S (x_1 = x_2) * R pc (p+1)\}$
	$\{S b * R pc p\}$ \vdash <code>BNE #-n</code> $\{R pc \text{ (if } b \text{ then } p-n \text{ else } p+1)\}$

Using the Hoare triples for the machine instructions occurring in a program, Myreen's decompiler automatically extracts a mathematical function to represent the data transformations on the state and then proves a Hoare triple giving a formal symbolic representation of the overall state change that happens when the program is executed.

³ $R b _$ is a shorthand for an existential quantification $\exists y. R b y$.

⁴The next instruction is shown as $p+1$ here, but actually it would be $p+4$ due to the way ARM addresses memory. The simplified view that adding one to the program counter gives the next instruction is also used in subsequent examples.

Here's an example: a simple program to compute the factorial function. The effect of the code is decompiled into the definition of the mathematical function computed by the program: $\text{fact}(n)$ (shown in red in the example below). The decompiler also proves a Hoare triple verifying that the program executed on the ARM model computes this function applied to the contents of register a and puts the result in register b . This is indicated by the big blue \vdash below.

	\vdash <pre> MOV b,#1 MUL b,a,b SUBS a,a,#1 BNE #-4 </pre>	$\vdash \text{fact}(n) =$ if $n = 0$ then 1 else $n \times \text{fact}(n-1)$
	$\{R a n * R b _ * S _ * R pc p * n \neq 0\}$ $\{R a 0 * R b (\text{fact}(n)) * S _ * R pc (p+4)\}$	+

The Hoare triple describing the effect of a machine code program is deduced from the Hoare triples for each machine instruction in the program using Myreen's machine code Hoare Logic. For example, Hoare triples for sequences of instructions are obtained using the Sequencing Rule:

$$\frac{\vdash \{P\} C_1 \{Q\} \quad \vdash \{Q\} C_2 \{R\}}{\vdash \{P\} C_1; C_2 \{R\}}$$

To apply this rule, the postcondition of the triple for C_1 must equal the precondition of the triple for C_2 (i.e. both be Q). The Frame Rule of Separation Logic is used to add invariants R to the 'small' footprints of preconditions and postconditions when performing logical deductions to make them match.

$$\frac{\vdash \{P\} C \{Q\}}{\vdash \{P * R\} C \{Q * R\}}$$

A special mechanism generates recursive functions to represent the semantics of code with loops (e.g. the definition of fact in the example). The details will not be described here – they can be found in Myreen's award winning⁵ PhD thesis [17].

The Hoare triple representation of the machine code behaviour that is automatically generated by the decompiler is a theorem mechanically proved in HOL4 from the formal specification of the ISA processor model. It certifies

⁵<http://academy.bcs.org/content/distinguished-dissertations-2010>

how the function decompiled from the code (e.g. fact above) corresponds to computations on the processor.

The decompiler can handle thousands of lines of code (e.g. the ARM binary for the seL4 microkernel [18]). This is because it works bottom incrementally combining the precomputed Hoare triples for the instructions in the code: decompiling binaries does not use the full ISA model.

3.3 Accurate model of ARM ISA version ARMv4T

In the second EPSRC project Fox upgraded the ISA ARMv3 executed by the ARM6 implementation to ARMv4, which was then still in use. He also added features to the model that had been ignored in the ARM6 verification work such as input/output, exceptions (e.g. interrupts) and coprocessor support. A PhD student at the time, James Reynolds, built a detailed model of an ARM floating point coprocessor in HOL, though alas this never got integrated with the ARM processor model (much later Fox added floating support models to some of his ARM models).

ARM processors are physically separate from main memory. This was not accurately reflected in the ARM6 work, which treated the registers and memory as part of a single processor state. For ARMv4 Fox decoupled processor and memory so that different memory models could be used with his processor model. To execute programs one needs both a processor and memory; Fox's approach provided a way to combine a processor model with a memory model to get an executable machine on which machine code programs could be symbolically executed. Such execution is a key part of generating the Hoare triples used for decompilation.

Recently Peter Sewell's group at Cambridge have been exploring current multi-processor ARM memory models. Current ongoing research [19] aims to combine this work with Fox's processor modelling.

3.4 Achievements of the second EPSRC project

The project summary for the second EPSRC project included the following statement.

If successful, this project will result in possibly the first machine checked formal verification of software running on a formally verified commercial off the-shelf (COTS) processor.

This had not really been achieved by the end of the project, but infrastructure had been created that supported Myreen’s decompilation method which later used this infrastructure for impressive examples like seL4 [18]. However, Fox’s later ISA models (ARMv4 onwards) did not run on a formally verified processor, since an ARM processor that ran this ISA was never verified. Thus a summary of what was achieved would be something like the following.

The project resulted in ideas and infrastructure that were subsequently used in the machine checked formal verification of software running on formally specified (but not verified) commercial off the-shelf (COTS) processors.

4 From 2007 to the present

During the second EPSRC project we were offered additional funding for ARM verification research by the US DoD via a contract with GCHQ. After the second EPSRC project ended, this new funding became the primary support for subsequent ARM work.

The new funding explicitly supported the development of software infrastructure in a way that would be hard to justify as research to EPSRC. It supported improvements to the HOL4 system by Fox and others. Things added to HOL4 included: bit-blasting tools, bignum and floating-point arithmetic theories, documentation, and improving general theorem-proving efficiency to support large examples.

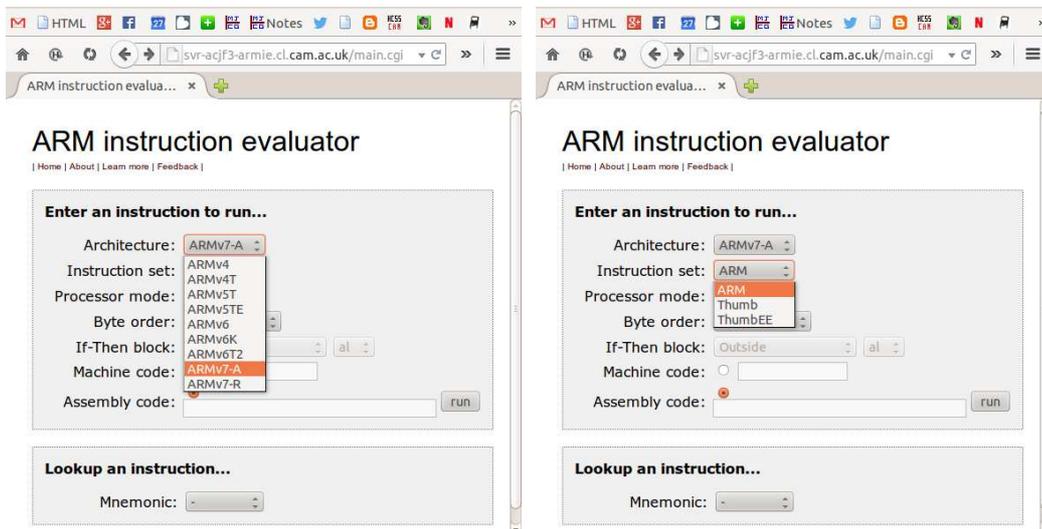
4.1 New ARM ISA models

The new funding enabled Fox to upgrade the existing ARM model to ARMv7, the ISA supported by the latest processors at the time. It was a significantly bigger model than ARMv4 as lots of new instructions arrived with ARMv6.

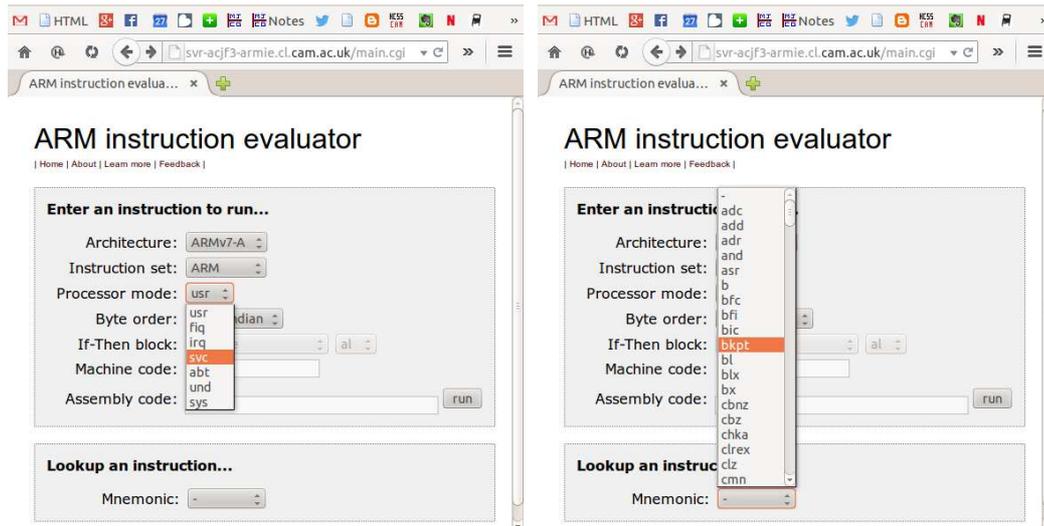
The formal specification of ARMv7 was not verified against a processor but was tested against ARM hardware by comparing executions of thousands of random instructions on the formal model using HOL4 theorem proving and on ARM processor hardware. Minor bugs were found in the formal model. Formal verification against an implementation model would have been better, but was impractical (see the discussion in §2.2).

Fox also created ARM ISA models for many of the current ISAs in use, including: ARMv4, ARMv4T, ARMv5T, ARMv5TE, ARMv6, ARMv6K, ARMv6T2, ARMv7-A, ARMv7-R, ARMv8, and ARMv6/ARM-M0. ARMv8 was created for the KTH PROSPER project (see §4.2.3); ARMv6/ARM-M0 is a version of ARMv6 with additional cycle counts corresponding to ARM-M0 implementations and was created for research at Edinburgh (see §4.7).

Fox also created a web page enabling the execution of particular instructions on many of his models. Here are some screenshots of the web interface. A menu allows one first to select an architecture (e.g. ARMv7-A, left screenshot) and then an instruction set can be chosen (e.g. 32-bit ARM rather than 16-bit Thumb, right screenshot).



Next a processor mode (e.g. svc, left screenshot) can be selected. Finally an instruction to execute is chosen from the selected architecture, instruction set and processor mode (e.g. bkpt, right screenshot). The semantics of the selected instruction is proved for the selected ISA using the HOL4 theorem prover and displayed on the web page.



The ARM models and supporting tools were initially just used at Cambridge by Myreen for his PhD research. It took a long time for the work to have any impact elsewhere, but eventually it did.

4.2 First impact of ARM models emerge

After about seven or eight years, the first applications of the ARM models outside Cambridge appeared. These were at a US company GrammaTech, the Australian Research Centre NICTA, and the PROSPER research project at the Royal Institute of Technology, in Sweden (KTH).

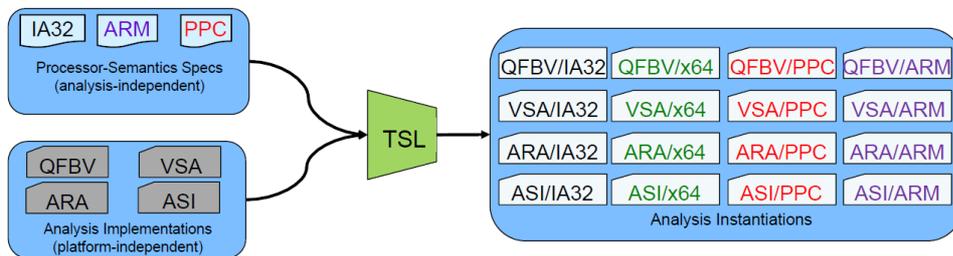
4.2.1 GrammaTech’s static analyser

GrammaTech is “a software-development tools vendor based in Ithaca, New York. The company was founded in 1988 as a technology spin-off of Cornell University. They now develop CodeSonar, a static analysis tool for source code and binaries, and perform cyber-security research” [20].

GrammaTech learnt about Fox’s ARM models at a Government scientific meeting and wanted to explore using them with their static analysis tools. Keen to get users of his ARM models, Fox translated his model into a subset of their proprietary ISA description language TSL. Although GrammaTech ended up not using this ARM model in TSL, they later made their own model based on a later model written in Fox’s ISA description language L3 (see §4.3 below) which they ported. Here is a diagram scraped from a talk at a recent workshop [21].

TSL/ISAL: Automatic Analysis Retargeting

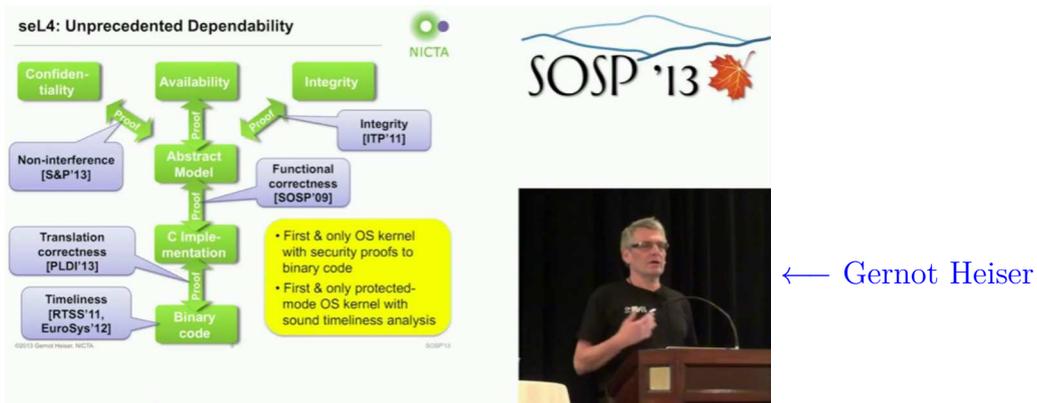
- Separate processor semantics specs from analysis implementations
- Automatically generate platform-specific analysis instantiations
- Benefits:
 - › **Independence** of semantics and analyses
 - Validation of each ISA semantics is separate from static analyses
 - Validation of each static analysis is separate from ISA definitions
 - › **Consistency**. All analyses for given ISA driven off of same definition
 - › **Completeness**. Full analysis generated for all instructions.



4.2.2 NICTA’s seL4 microkernel verification

The seL4 is “the world’s first operating-system kernel with an end-to-end proof of implementation correctness and security enforcement. It is still the world’s most highly-assured OS” [22]. The main part of the seL4 verification shows that a specification of the desired behaviour of the OS in the form of functional model is correctly implemented by C language like code. Both the functional model and C-like code were represented in Isabelle/HOL.

The seL4 OS is intended to be run on ARM. Experiments with the CompCert “a high-assurance compiler for almost all of the ISO C90 / ANSI C language, generating efficient code for the PowerPC, ARM and x86 processors” [23] did not work out so, following a visit to NICTA by Myreen, a ‘translation correctness’ approach based on decompiling and verifying an ARM binary obtained using the unverified GCC compiler was adopted. This was successfully carried out by Myreen during a series of winter visits to NICTA. Here is an image scraped from a YouTube video of a talk by Gernot Heiser [24].



4.2.3 KTH PROSPER project

The box below contains an online description of the PROSPER project [25].

PROSPER (Provably Secure Execution Platforms for Embedded Systems) is a research project that aims to build the next generation framework for fully verified, secure hypervisors for embedded systems. The following components constitute the core of the project:

1. A provably secure execution platform for embedded devices such as mobile phones based on a virtualization core. Our hypervisor is available as open source.
2. A prototype toolset for formal specification and verification of different versions of the hypervisor. Within this work, ISA isolation lemmas for user mode execution on ARM have been verified. Those proofs are available from the HOL4 GitHub site.

PROSPER is a collaboration between the Group for Theoretical Computer Science at KTH and SICS Swedish ICT, a research institute for applied information and communication technology in areas strategic for Swedish industry.

Here is an image scraped from a YouTube video of a talk by Mads Dam, leader of the PROSPER project entitled “Formal verification of information flow security for a simple ARM-based separation kernel” [26].

Mads Dam →

<i>ARMv7 properties</i>	<i>Handler code</i>
User Lemma Switch Lemma	Handler Lemmas Boot Lemma
Property of ARMv7 instruction set architecture	Code property Frequently updated
HOL4 + Cambridge ARMv7 model + MMU	BAP + STP
Inference lemmas	Contract verification
"Quarto"-automatic	"Semi"-automatic

PROSPER has now moved on to ARMv8 from ARMv7 shown in this picture. Mads Dam recommends the recent CCS'13 paper [27] for further reading.

4.3 The ISA specification language L3

The creation of all the various ARM models resulted in a problem of validating and maintaining them. To solve this Fox created a model authoring languages which he called L3 (abbreviating “Low Level Language”).

L3 is a language specifically designed for writing ISA models in HOL. However, as described later, it has found other uses. Fox designed L3 with a syntax partly inspired by the pseudo code from ARM documentation. His emphasis was ease of capturing the content of informal documentation, not theoretical innovation. The original goal was to make it easier to write and maintain ISA models for use in formal verification.

Here is a slide from a recent talk by Fox showing the various models currently available in L3.

L3 Instruction Set Models

Model	Supported Modes	Coverage	Lines of L3
ARMv4 through to ARMv7-A	ARM and Thumb	Partial VFP, no Advanced SIMD & coprocessor	9238 + 7687
ARMv6-M	Thumb only	No coprocessor	1996 + 2095
ARMv8	AArch64 only	No VFP, Advanced SIMD and only partial system	2434 + 4097
x86-64	64-bit only	Only 40 instructions	1357 + 1579
MIPS (in HOL)		No floating-point, partial coprocessor & system.	2080 + 700
CHERI		Enough to boot FreeBSD	5203

Here's another slide from the same talk illustrating decompilation.

Decompilation to Logic

- The original use for the L3 instruction set models was the decompilation of machine-code to HOL logic, as devised by Magnus Myreen.

Machine code

```
e3b01000 (* movs r1, #0 *)
e1a03000 (* mov r3, r0 *)
e2933028 (* adds r3, r3, #40 *)
e5902004 (* ldr r2, [r0, #4] *)
e2900004 (* adds r0, r0, #4 *)
e0811002 (* add r1, r1, r2 *)
e1500003 (* cmp r0, r3 *)
baffffff (* blt -#16 *)
```



```
test (r0, dmem, mem) =
  let r1 = 0w in
  let r3 = r0 in
  let r3 = r3 + 40w in
  let (r0, r1, r2, r3, dmem, mem) = test1 (r0, r1, r3, dmem, mem)
  in
  (r0, r1, r2, r3, dmem, mem)

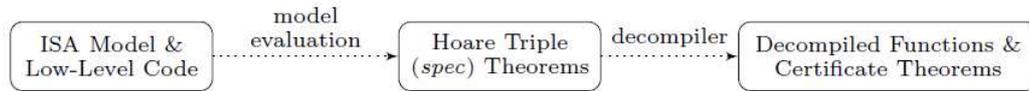
test1 (r0, r1, r3, dmem, mem) =
  let r2 =
    mem (r0 + 4w + 3w) @@ mem (r0 + 4w + 2w) @@
    mem (r0 + 4w + 1w) @@ mem (r0 + 4w)
  in
  let r0 = r0 + 4w in
  let r1 = r1 + r2
  in
  if word_bit 31 (r0 - r3) => OVERFLOW r0 (-r3) T then
    (r0, r1, r2, r3, dmem, mem)
  else test1 (r0, r1, r3, dmem, mem)
```

HOL functions

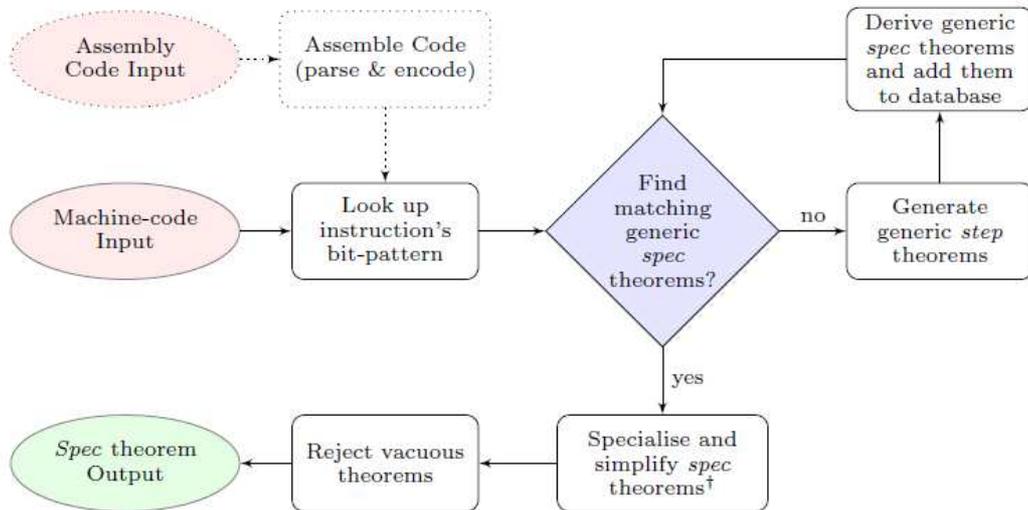
```
⊢ SPEC_ARM_MODEL
  (~aS * arm_OK mode * arm_PC p * arm_MEMORY dmem mem *
   arm_REG (R_mode mode 0w) r0 * ~arm_REG (R_mode mode 2w) *
   ~arm_REG (R_mode mode 1w) * ~arm_REG (R_mode mode 3w) *
   cond (test_pre (r0, dmem, mem)))
  {(p, 0xE3B01000w); (p + 4w, 0xE1A03000w); (p + 8w, 0xE2933028w);
   (p + 12w, 0xE5902004w); (p + 16w, 0xE2900004w);
   (p + 20w, 0xE0811002w); (p + 24w, 0xE1500003w);
   (p + 28w, 0xBAFFFFFFAw)}
  (let (r0, r1, r2, r3, dmem, mem) = test (r0, dmem, mem)
  in
  ~aS * arm_OK mode * arm_PC (p + 32w) * arm_MEMORY dmem mem *
  arm_REG (R_mode mode 0w) r0 * arm_REG (R_mode mode 1w) r1 *
  arm_REG (R_mode mode 2w) r2 * arm_REG (R_mode mode 3w) r3)
```

Certificate theorem

The formal semantics of a model written in L3 is given by the HOL that is generated from it; this is accomplished by tools Fox implemented to support the language. Other tools are used to decompile machine code to generate Hoare triples. The following diagram is from a currently unpublished paper by Fox.



Here is a diagram from the same paper illustrating how the tools work.



All the current projects at Cambridge now use HOL models generated from L3 specifications. The ISA model used in the seL4 binary verification at NICTA has also moved to using L3. The PROSPER project originally used a model written directly in HOL, but for future work Fox has collaborated with the KTH researchers to create an L3-generated model of the recently released ARMv8.

Although L3 was initially motivated by the need to specify and maintain multiple models of ARM ISAs, it is suitable for specifying other architectures. Projects at Cambridge are using a partial model of x86 (see §4.5) and a fairly complete model of MIPS (see §4.6). Some other projects using L3 are outlined below.

4.5 CakeML

CakeML (logo: ) is an international collaborative project [30] to develop and implement a dialect of Standard ML designed to support program verification. Here is part of the abstract of a recent paper [31].

We have developed and mechanically verified an ML system called CakeML, which supports a substantial subset of Standard ML. CakeML is implemented as an interactive read-eval-print loop (REPL) in x86-64 machine code. Our correctness theorem ensures that this REPL implementation prints only those results permitted by the semantics of CakeML. Our verification effort touches on a breadth of topics including lexing, parsing, type checking, incremental and dynamic compilation, garbage collection, arbitrary-precision arithmetic, and compiler bootstrapping.

Our contributions are twofold. The first is simply in building a system that is end-to-end verified, demonstrating that each piece of such a verification effort can in practice be composed with the others, and ensuring that none of the pieces rely on any over-simplifying assumptions. The second is developing novel approaches to some of the more challenging aspects of the verification.

The following points are extracted from `cakeml.org`, the CakeML web page.

- We have written, verified, and bootstrapped a compiler (including lexing, parsing, type inference, and code generation) from CakeML to CakeML Bytecode. The correctness theorem covers both terminating and diverging programs, and says that the generated code behaves according to the semantics in either case. The compiler is written in HOL; we use the translator, mentioned below, to generate a verified CakeML implementation, and then evaluate the compiler on this implementation (bootstrap) to generate verified bytecode.
- One of our initial target case studies is to construct a verified CakeML version of the HOL light theorem prover. For this case study, we extended the translation tool, mentioned below, to be able to translate into stateful CakeML code.
- We have developed a tool which translates functions from higher-order logic into CakeML. This tool is proof producing: for each translation it proves a theorem which states that the generated CakeML code correctly implements the original HOL function, according to the operational semantics of CakeML.

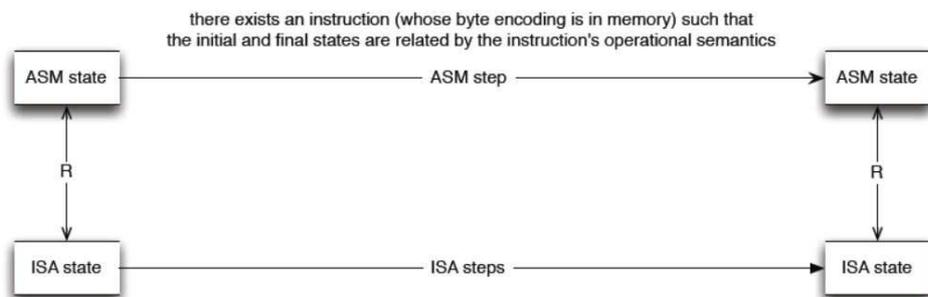
The text in the box above has been slightly condensed (e.g. sentences deleted). See the web page for further details, including links to papers.

The translation to machine code of CakeML bytecode, called ASM, which the compiler produces, is verified against L3 models of ARM, x86 and MIPS using decompilation.

Here is a diagram from one of Fox's recent talks illustrating this:

CakeML: ASM to Machine-Code Correctness

- The following is shown to hold for each ARM state. (+ Side-conditions.)



- The choice of ASM instruction is required to result in deterministic behaviour. In practice, this means that the encoding is required to be “decodable”.
- Multiple ISA instructions can be used to implement each ASM instruction.

Rockwell Collins in the USA are evaluating CakeML via a research project. Full details are not public, but the work is funded by NASA with the goal of demonstrating that CakeML can be used to get verified executable binaries of interest to them from logical specifications. The CakeML compiler is also being evaluated in relation to the possibility of using verified compilers in certification processes.

4.6 MIPS-based processors: BERI and CHERI

The BERI and CHERI processors are being designed and implemented (on FPGAs) at Cambridge as part of a project entitled *Clean Slate Trustworthy Secure Research and Development* (CTSRD), which is part of the DARPA CRASH programme.

Here's a picture of a device containing a BERI processor running FreeBSD (from <http://www.cl.cam.ac.uk/research/security/ctsr/cheri/>).



Here's an extract from the CTSRD web page [32].

The project is revisiting the hardware-software security interface for general-purpose CPUs to fundamentally improve security; to this end, we are integrating a hybrid capability model and continuous hardware-assisted validation of security design principles with a commodity CPU ISA and open source operating systems. We are pursuing several new software/hardware features as part of this research:

- BERI: a open-source hardware-software research and teaching platform: a 64-bit RISC processor implemented in the high-level Bluespec hardware description language (HDL), along with compiler, operating system, and applications;
- CHERI: capability hardware enhanced RISC instructions : hardware-accelerated in-process memory protection and sandboxing model based on a hybrid capability model;

The golden reference models of BERI and CHERI are represented in L3.

Fox’s L3 simulator is being used to generate execution traces for comparison with the hardware and also for design space explorations of both the processor and cache. The L3 model supports several processors with a shared memory. The simulator is able to boot FreeBSD on BERI and CHERI.

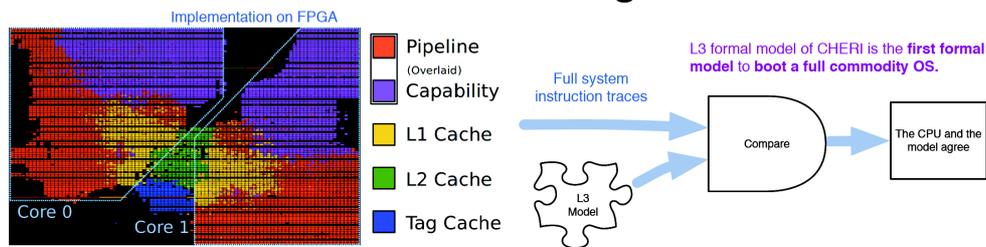
Here’s an extract from a recent technical report [33].

We have also developed a more complete ISA model incorporating both MIPS and CHERI instructions using Cambridge’s L3 instruction-set description language. Although we have not yet used this for automated theorem proving, we increasingly use the L3 description as a “gold model” of the instruction set against which our test suite is validated, software implementations can be tested in order to generate traces of correct processor execution, and so on. We have used the L3 model to identify a number of bugs in multiple hardware implementations of CHERI, as well as to discover software dependences on undefined instruction-set behavior.

This application is remarkable in using a specification method developed explicitly to support theorem proving, but using it for something different. It is hoped in the future to exploit decompilation to verify properties of CHERI formally.

The diagram below is a fragment cut out of a larger project presentation poster that contains material on other parts of the CTSRD research. [34]:

CHERI Processor and ISA Testing and Verification



An unexpected benefit of modelling BERI and CHERI in L3 is that formal methods for generating test data for ARM models being developed at Edinburgh turns out to be useful for generating tests for models of these MIPS-based machines. This is briefly described in §4.7 below.

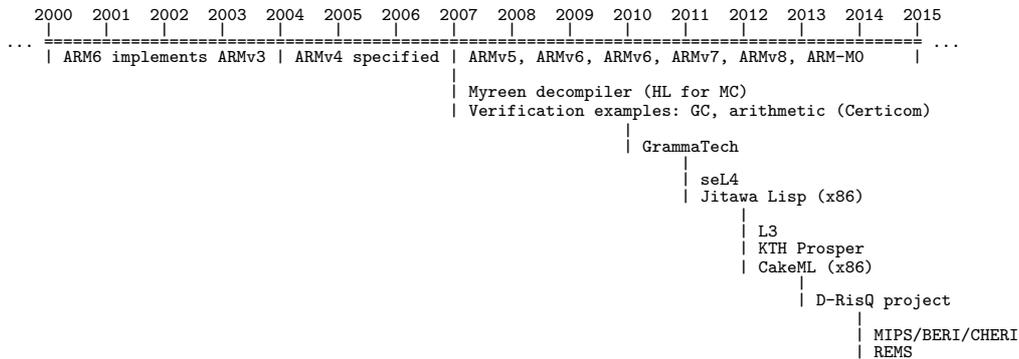
4.7 Using HOL and L3 ARM models for testing

Brian Campbell and Ian Stark from Edinburgh University have devised a semantics-driven random testing method based on HOL and L3 models. Using a model of ARMv6 equipped with accurate cycle counts for several ARM-M0 implementations, they discovered timing anomalies between stores and fetches that affects multiple implementations. Their methods are being applied to BERI and CHERI. Here's an abstract from an unpublished paper.

We validate a HOL4 model of the ARM Cortex-M0 microcontroller core by testing the model's behaviour on randomly chosen instructions against real chips from several manufacturers. The model and our intended application involve precise timing information about instruction execution, but the implementations are pipelined, so checking the behaviour of single instructions would not give us sufficient confidence in the model. Thus we test the model using sequences of randomly chosen instructions. The main challenge is to meet the constraints on the initial and intermediate execution states: we must ensure that memory accesses are in range and that we respect restrictions on the instructions. By careful transformation of these constraints an off-the-shelf SMT solver can be used to find suitable states for executing test sequences. We also use additional constraints to test our hypotheses about the timing anomalies encountered. The randomised test case generation has been generalised to make it easier to adapt to new L3 models, starting with the simple MIPS model. This detected issues with the modelling of branch delay slots and a potentially security relevant bug in the BERI/CHERI processor implementation of `movz` instructions. Ongoing work is adapting this to the CHERI model. The library for MIPS that provides an equational presentation of the model's behaviour for each instruction is not available for the CHERI model (and was not likely to be complete because it was intended for verification), so a more general library performing symbolic evaluation of the model for a given instruction was constructed, taking advantage of the monadic structure of L3 generated HOL terms.

5 Reflections

Below is a timeline showing what was done in the last fifteen years, starting from the first EPSRC grant. This is followed by a list of phases of activity into which the work can be factored.



Formal proof that ARM6 implements ARMv3

The first EPSRC project showed the Harman-Tucker algebraic modelling and proof method fitted well with traditional 1980s-style mechanised HOL hardware verification and scaled to at least ARM6.

Creation of many formally specified but unverified ARM models

Building on the ARMv3 specification Fox created many ARM models up to the still-used ARMv7. Researchers needing ARM models took an interest (e.g. GrammaTech in the USA and KTH in Sweden).

Hoare Logic for hardware and machine code decompilation

Concurrently with the creation of multiple ARM models by Fox, Myreen invents his method of Hoare logic based decompilation and demonstrates its surprising scalability via an increasingly impressive sequence of examples including: machine code garbage collection, multiple precision arithmetic, a JIT-ed Lisp runtime [35], the seL4 OS. Some of these used non-ARM processor ISAs.

L3 for authoring and maintaining HOL models

Fox designs L3, initially to organise his existing ARM models and for writing new ones more easily (e.g. ARMv8). He then uses it to model other ISAs including MIPS and x86. Existing users of the HOL ARM models switch to L3-generated models (e.g. seL4 and KTH PROSPER). New projects emerge (e.g. D-RisQ and Edinburgh/REMS).

Applications of L3 not using HOL or theorem proving

L3 ISA models start to be used without any involvement of HOL or theorem proving. GrammaTech port an L3 ARM directly into their TSL language. Cambridge researchers use the L3 simulator for testing MIPS implementations against hardware traces. They adopt L3 for their ‘golden ISA models’ and begin to use L3 specifications and tools for design space explorations for new architecture extensions.

The timeline shows that it took about seven years before the ARM verification work started to have any impact. The successfully achieved original goal of verifying an ARM6 processor implementation had virtually no impact at the time. When the impact came it was in places unforeseen at the start of the project.

Nobody foresaw that the research would be a part of perhaps the first formal verification of an operating system binary or provide a formal specification simulator capable of booting a commodity OS and being used for architecture research exploring designs of a security enhanced MIPS processor.

The following text is edited from a response to a draft policy document about future science funding in the UK.

It seems that immediate impact, which is likely to be incremental in nature, is overshadowing the longer-term perspective and may lead to short-term gains but a dearth of major advances for the future. EPSRC, for example, is concentrating funds into strategic areas and so squeezing the money for responsive-mode funding, and hence the space available for curiosity-driven research. Since this approach is meant to be all about achieving “impact” and plenty of impact has arisen serendipitously from high quality curiosity-driven research in the past, this could be counter-productive: even having the opposite effect to that intended.

The story of the ARM project told here is an example of research that had little “immediate impact” ... but eventually after many years, and thanks to far-sighted funders, is having “plenty of impact” which indeed has “arisen serendipitously”.

6 Acknowledgements

The work described here was done by Anthony Fox, Magnus Myreen and many others who devised and implemented the applications.

The research would not have been possible without the support of our funders: EPSRC in the early years, then the US Department of Defence and GCHQ, and now EPSRC again thanks to the REMS project [19].

I would particularly like to thank Graham Birtwistle for inviting Cambridge to participate in his collaboration with ARM and Brad Martin, Tim Thimmesch and Andy Jackson who had the vision to realise that research like this is a ‘long game’ that may not yield immediate results.

References

- [1] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
- [2] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008.
- [3] Wikipedia article on the Apple Newton: [http://en.wikipedia.org/wiki/Newton_\(platform\)](http://en.wikipedia.org/wiki/Newton_(platform)).
- [4] Mat Honan. Remembering the Apple Newton’s prophetic failure and lasting impact. *Wired Magazine*, August 2013.
<http://www.wired.com/2013/08/remembering-the-apple-newtons-prophetic-failure-and-lasting-ideals/>.
- [5] Wikipedia article giving a list of ARM architectures:
http://en.wikipedia.org/wiki/List_of_ARM_microarchitectures.
- [6] EPSRC website for grant:
<http://gow.epsrc.ac.uk/NGBOViewGrant.aspx?GrantRef=GR/N13135/01>.
- [7] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.
- [8] Lawrence C. Paulson. *ML for the Working Programmer (2nd Ed.)*. Cambridge University Press, New York, NY, USA, 1996.
- [9] ARM Architecture Reference Manual. Old versions available online, e.g. http://simplemachines.it/doc/ddi0100e_arm_arm.pdf.
- [10] N. A. Harman and J. V. Tucker. Algebraic models of microprocessors: The correctness and verification of a simple computer., 1995.
- [11] NA Harman and John V Tucker. The formal specification of a digital correlators. In *Theoretical foundations of VLSI design*, volume 10. Cambridge University Press, 2003.
- [12] Anthony Fox. An algebraic framework for modelling and verifying microprocessors using HOL. Technical Report UCAM-CL-TR-512, University of Cambridge, Computer Laboratory, March 2001.
- [13] Anthony Fox. Formal verification of the ARM6 micro-architecture. Technical Report UCAM-CL-TR-548, University of Cambridge, Computer Laboratory, November 2002.
- [14] RobertS. Boyer and Yuan Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. In Deepak Kapur, editor, *Automated Deduction - CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 416–430. Springer Berlin Heidelberg, 1992.

- [15] Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic, CSL ’01*, pages 1–19, London, UK, UK, 2001. Springer-Verlag.
- [16] Magnus O Myreen and Michael JC Gordon. Hoare logic for realistically modelled machine code. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 568–582. Springer, 2007.
- [17] Magnus O. Myreen. Formal verification of machine-code programs. Technical Report UCAM-CL-TR-765, University of Cambridge, Computer Laboratory, December 2009.
- [18] Thomas A. L. Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13*, pages 471–482, New York, NY, USA, 2013. ACM.
- [19] Web page for the REMS project:
<http://www.cl.cam.ac.uk/~pes20/rems/>.
- [20] Wikipedia article on GrammaTech:
<http://en.wikipedia.org/wiki/GrammaTech>.
- [21] Alexey Loginoff, Machine-code analysis and transformation at GrammaTech. Talk presented at TAPAS 2014, The Fifth Workshop on *Tools for Automatic Program Analysis*, September 10, 2014, Munich, Germany. <http://cs.au.dk/tapas2014/>.
- [22] General Dynamics C4 Systems and NICTA web page:
<https://sel4.systems/>.
- [23] CompCert compiler home page: <http://compcert.inria.fr/>.
- [24] YouTube video *From L3 to seL4 what have we learnt in 20 years of L4 microkernels?*:
<https://www.youtube.com/watch?v=RdoaFc5-1Rk>.
- [25] Web page for KTH PROSPER project:
<http://www.csc.kth.se/~oschwarz/prosper/>.
- [26] YouTube video *Formal verification of information flow security for a simple ARM-based separation kernel*:
<https://www.youtube.com/watch?v=RdoaFc5-1Rk>.
- [27] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, pages 223–234. ACM, 2013.
- [28] D-RisQ’s web page: <http://www.drisq.com/>.
- [29] YouTube video from D-RisQ’s web page:
https://www.youtube.com/watch?v=ftMvnR_rIF0.
- [30] Web page for the CakeML project: <http://www.cakeml.org>.

- [31] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, pages 179–191, New York, NY, USA, 2014. ACM.
- [32] Cambridge CTSRD web page:
<http://www.cl.cam.ac.uk/research/security/ctsr/>.
- [33] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Simon W. Moore, Steven J. Murdoch, and Michael Roe. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-set architecture. Technical Report UCAM-CL-TR-864, University of Cambridge, Computer Laboratory, December 2014.
- [34] Poster from CTSRD project:
<http://www.csl.sri.com/users/neumann/20140915-ctsr-jacksonville-poster.pdf>.
- [35] Magnus O. Myreen and Jared Davis. The Reflective Milawa Theorem Prover Is Sound - (Down to the Machine Code That Runs It). In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving (ITP)*, pages 421–436. Springer, 2014.