Trustworthy Specifications of ARM® v8-A and v8-M System Level Architecture

Alastair Reid
Research, ARM Ltd.
first.last@arm.com

***Abstract*—Processor specifications are of critical importance for verifying programs, compilers, operating systems/hypervisors, and, of course, for verifying microprocessors themselves. But to be useful, the *scope* of these specifications must be sufficient for the task, the specification must be *applicable* to processors of interest and the specification must be *trustworthy*.**

This paper describes a 5 year project to change ARM's existing architecture specification process so that machine-readable, executable specifications can be automatically generated from the same materials used to generate ARM's conventional architecture documentation. We have developed executable specifications of both ARM's A-class and M-class processor architectures that are complete enough and trustworthy enough that we have used them to formally verify ARM processors using bounded model checking. In particular, our specifications include the semantics of the most security sensitive parts of the processor: the memory and register protection mechanisms and the exception mechanisms that trigger transitions between different modes. Most importantly, we have applied a diverse set of methods including ARM's internal processor test suites to improve our trust in the specification using many other expressions of the architectural specification such as ARM's simulators, testsuites and processors to defend against common-mode failure. In the process, we have also found bugs in all those artifacts: testing specifications is very much a two-way street.

While there have been previous specifications of ARM processors, their *scope* has excluded the system architecture, their *applicability* has excluded newer processors and M-class, and their *trustworthiness* has not been established as thoroughly.

Our focus has been on enabling the formal verification of ARM processors but, recognising the value of this specification for verifying software, we are currently preparing a public release of the machine-readable specification.

I. Introduction

Recent years have seen an increasing focus on verification of machine-code programs [1], compilers [2], operating system kernels [3], hypervisors [4] and processors [5]. These activities rely on having correct specifications of the meaning of machine-code and one of the first steps in such verification efforts is creating a specification of the computer architecture of interest.

Three key properties of a processor specification are its *scope*, its *applicability* and its *trustworthiness*.

The *scope* of a specification is the set of features that one can reason about. For example, a certified compiler such as CompCert [2] only requires a specification of those instructions that the compiler could generate. But in order to reason about arbitrary user-mode binaries, one would need a specification of the entire instruction set. And to reason about Operating System code, the scope of the specification is dramatically increased and includes a specification of instructions for changing execution mode (e.g., entering/leaving supervisor mode), interrupt handling mechanisms, page faults, mechanisms for changing memory protection, etc. To date, all formal specifications of the ARM architecture have been targetted at reasoning about user-mode programs and have not included a specification of these system-level features.

The *applicability* of a processor specification is whether the specification applies to the target processor. Most changes to architecture specifications are backward compatible extensions and so most proofs about code for one architecture version are valid when executing that code on a processor implementing a later architecture version. But architecture revisions also remove instructions, add restrictions or change functionality so proofs based on the ARMv6 specification (1996) or the ARMv7-A specification (2007) are not necessarily sound for ARMv8-A (2013). This is especially true for ARM's Microcontroller architecture which has a completely different exception model from ARM's mainstream architecture.

The *trustworthiness* of a processor specification is whether the specification can be trusted to reflect the behaviour of all processors implementing the specification. The ARMv7 HOL specification of Fox and Myreen [1] is noteworthy for the degree of testing performed: systematically testing all user-mode, integer instructions against three actual processors. This is a critical step and must be repeated against as many expressions of the architecture as possible (processors, implementations, testsuites, etc.) and must be used to test the full scope of the specification.

The effort required to create a specification increases with the desired scope, applicability and trustworthiness of the specification. Worse, since ARM regularly releases extensions and corrections to the architecture, the challenge of retaining applicability to current processors is more of a continuous process rather than a one-off sprint. Our solution to this problem has been to change ARM's existing architecture specification process so that machine-readable, executable specifications can be automatically generated from the same materials used to generate conventional documentation.

This paper describes our work over the last 5 years on transforming the ARM processor specifications from documents intended for human consumption into trustworthy machine-readable specifications.

Creating this specification required understanding and cod-

Accepted for publication at Formal Methods in Computer-Aided Design (FMCAD), Mountain View, USA, October 3–6, 2016.

ifying the precise meaning of various notations used in the documentation; inferring the lexical, syntax, type rules and semantics from examples in the documentation; making the specification conform to these rules; filling gaps in the original specification; and creating a frontend and several backends to allow the specification to be executed.

Using ARM's specifications directly addresses the issues of *scope* and *applicability* but the resulting formal part of the specification is just one part of the whole specification and, like any large specification, may contain bugs wrt the informal parts of the specification or with the architects' informal intent. To address the issue of *trust*, we have used a diverse set of testing methodologies to compare against as many different expressions of the specification as possible: testsuites, simulators and processors. We have simulated billions of instructions and used bounded model checking to compare the RTL of five ARM processors currently in development against the specification [6]. Bugs found in the process have been fixed in the master copy of the specification from which ARM's architecture specification documents are generated. This process has the effect of distilling more of the architectural intent into the formal part of ARM's official specification.

The structure of this paper is summarized in Figure 1 which gives an overview of the specifications, tools, verification IP, and testing we created or used in the process of this project. Section II gives a brief overview of the structure and content of the different ARM Architectures. Sections III and IV describe the steps we took to convert ARM's existing informal documentation into machine-readable, executable, trustworthy specifications of the ARM-v8A and ARM-v8-M architectures; Section V discusses related work; and Section VI concludes.

This paper deals with the Instruction Set Architecture (ISA), Exceptions, Memory Protection/Translation and Security. It does not deal with multiprocessor features and, in particular, the Memory Ordering Model [3], [7], [8]. And it does not deal with debug or performance monitoring features.

II. ARM Specifications

ARM Architecture specifications have two main sections: Application Level Architecture and System Level Architecture.

The Application Level Architecture (aka the Instruction Set Architecture or ISA) consists of all instructions and all user-mode registers (the integer and floating point register files, condition flags, stack pointer and program counter). ISA specifications consist of instruction encodings, matching rules to match encodings to opcodes and the semantics of instruction execution.

The System Level Architecture defines Memory Translation and Protection, Synchronous Exceptions (e.g., page faults and system traps), Asynchronous Exceptions (e.g., interrupts), Security (e.g., register banking and access protection of registers), and System Registers and System Operations (which are used to control and read the status of all the system-level features), In other words, the facilities needed to support Operating Systems, Hypervisors and Secure Monitors.

The ARM architecture comprises three main processor classes: "A-class" processors support Applications (characterized by having an operating system that uses address translation to provide virtual memory); "R-class" processors support Real-Time systems that cannot handle the timing variability associated with virtual memory and use memory protection instead; and "M-class" microcontrollers are optimized for programming interrupt-driven systems in the C language. The A-class specification consists of two parts: *AArch32* supports 32-bit programs and is generally backward compatible with ARM's traditional architecture; and *AArch64* which supports 64-bit programs.

The A- and R-class architecture [9] share the same ISA and exception model but have different memory protection/translation models. The M-class architecture [10] has a subset of the A-class ISA but has significant differences from A-class at both the Application Level and System Level.

A. ISA Differences between A/R- and M-class

The M-class architecture only supports the Thumb® (aka "T32") variable-length instruction encodings whereas the A/R-class architecture also supports the A32 and A64 encodings.

Much more significantly though, the specifications identify certain instruction encodings as UNPREDICTABLE for which a processor is free to do anything that can be achieved at the current or a lower level of privilege using instructions that are not UNPREDICTABLE and that does not halt or hang the processor or parts of the system.

In the M-class architecture, many of the instruction encodings which access the stack pointer (R13) or the program counter (R15) are UNPREDICTABLE but the same encodings are well defined in the A/R-class architecture. This is a significant difference — it would be unsound to use the A-class specification to reason about Thumb machine code intended for an M-class processor.

More broadly, when performing formal verification, it is essential to ensure that the specification version being used matches the architecture version supported on the target processor because later specifications are *almost* but *not entirely* backward compatible. This is obvious but easily overlooked.

B. System Differences between A-, R- and M-class

The R/M-class architectures support memory protection based on setting attributes and protection for a small number of contiguous memory regions whereas the A-class architecture supports both address translation and memory protection for a large number of memory pages.

M-class processors automatically save the callee-save registers on the stack on taking an exception whereas A/R processors require registers to be saved in software. This allows M-class processors to respond more quickly to interrupts and also allows exception handlers to be written in plain C with no assembly language or special calling conventions. This has a large impact on the architecture specification since it introduces many corner cases associated with the effect of triggering memory faults while saving or restoring registers.

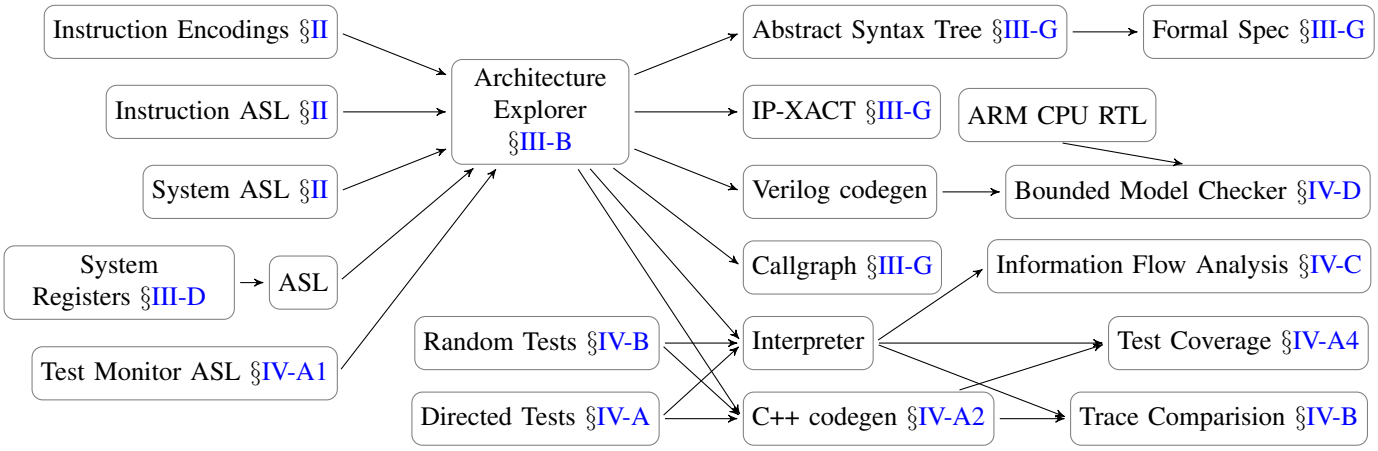


Fig. 1: Overview of specifications, tools, verification IP and testing. This flow was applied separately to the v8-A specification and to the v8-M specification. Section numbers indicate which section primarily discusses each aspect.

M-class processors have an orthogonal set of 8 execution states composed of combinations of three properties: privileged/unprivileged, secure/non-secure and handler/thread. A/R-class processors have a more traditional set of nested execution states EL0, EL1 (supervisor), EL2 (virtualization) and EL3 (secure monitor) with increasing levels of privilege at each level.

A consequence of these differences is that the M-class system specification is completely different from the A/R-class system specification.

III. Executable Specifications

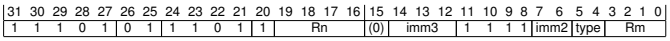
We faced five major challenges in turning ARM's documentation-based specification into an executable specification: (1) Scale: ARM specifications are very large; (2) Informality: ARM specifications are written in "pseudocode"; (3) Gaps: key parts of the specification only existed in natural language specification; (4) System Register Specifications; and (5) Implementation Defined Behaviour.

A. ARM Specifications Are Large

One of the main challenges in creating machine-readable specifications of the ARM Architecture is the scale of the problem. The A and M-class architectures together consist of over 6,000 pages of documentation, 1,570 instruction encodings, over 50,000 lines of pseudocode, over 4,500 system register fields grouped into 772 system register, and 112 system operations. To this specification that ARM publishes, we added an additional 8,190 lines of support pseudocode which were required to make the execution executable. (A more detailed breakdown of the size of the specification is given in table 2a and table 2b.)

B. Pseudocode

A secondary challenge in creating a machine readable specification was that the bulk of the specification is written in what the ARM documentation refers to as "pseudocode". For example, the T32 CMP instruction is specified with the following encoding diagram and pseudocode in the v8-A architecture. (The same instruction is UNPREDICTABLE in v8-M if "m == 13".)




```
CONDITIONAL
    n = UInt(Rn); m = UInt(Rm);
    (shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
    if n == 15 || m == 15 then UNPREDICTABLE;
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcv) = AddWithCarry(R[n], NOT(shifted), '1');
    PSTATE.<N,Z,C,V> = nzcv;
```

Fortunately for us, this "pseudocode" was fairly complete and it appeared possible to implement a conventional parser, typechecker and interpreter for pseudocode (a tool we call "Architecture Explorer"). Through a process of experimentation, discussion and negotiation with the architecture designers, we were able to infer consistent indentation rules, precedence rules, a type-system and semantics and to clean up the specifications to use the resulting simpler, more consistent language that is now internally referred to as ARM Specification Language (ASL).

At a high level, ASL is an indentation-sensitive, imperative, strongly typed, first-order language with dependent types (to reason about length of bit vectors), type inference, exceptions, enumerations, arrays, records, no pointers. Unusually for an otherwise simple language, ASL allows overloading of array syntax for function calls: the use of "R[m]" and "R[n]" on lines 4 and 5 of the example above are both function calls. This syntactic sugar provides an initial impression that registers (and memory) are simple arrays, while allowing one to dig deeper and understand register banking, virtual memory, etc. We refer readers to Fox and Myreen [1] or to ARM's specification [9, Appendix G] for a more detailed description of ASL.

The initial cleanup of syntax and type errors resulted in changes to approximately 12% of the lines of code but,

	ARMv8-A				ARMv8-M	
	AArch32	AArch64	Shared	Support	Spec	Support
Instrs.	18318	5757			4998	
Integer	23		352		246	
Float Point			1179		953	76
Exceptions	1474	1611	235		781	
Registers	310	446	398		2011	461
Memory	1584	1169	393		369	481
Debug	675	537	1103			
Instr. Fetch				199	367	128
Test Monitor	-	-	-	1323	-	1893
Misc.	1647	1137	2984	1678	415	1434
Total	24315	10657	5489	3200	9898	4990

(a) Size of ASL specification (lines of code)

	v8-A	v8-M
Registers	586	186
Fields	3951	622
Constant	985	177
Reserved	940	208
Impl. Defined	70	10
Passive	1888	165
Active	68	62
Operations	112	10

(b) Size of System Register specification

Fig. 2: Size of ARM Specifications

since ARM specifications are extensively reviewed before release, these were all fairly low-grade errors: they confused automatic tools but few were likely to confuse a human reader. The process of cleaning up the specification also uncovered a number of instances of "implement by comment" where comments were used instead of pseudocode: these parts had to be rewritten before the code could be executed. These simple comments often turned out to be surprisingly complicated and the process of writing code would identify corner cases or the need to modify other parts of the specification.

C. Gaps in the specification

Some parts of the architecture were only defined in English and the information to implement them was typically scattered throughout the documentation. An example is the specification of the "top-level" step of fetching an instruction, decoding and executing it, and incrementing the program counter was not written in ASL and the description was scattered across the specification document. The exact specification of this step took some time to develop as it includes details like dealing with page faults that occur during instruction fetch, not incrementing the PC after a branch instruction or exception, conditional execution of instructions and its interaction with UNDEFINED encodings, and testing for pending interrupts.

D. System Register Specification

The major negative surprise of this project was how hard it was to specify something as apparently simple as a register.

The A-class architecture specification comprises 586 system registers which are used to read the status of and to control the behaviour of the processor (such as whether the MMU or cache is turned on) and to perform operations such as flushing the cache or invalidating the TLB. The main properties of these registers are captured in the architecture specification by tables specifying the opcode to access each register, its name, size (32/64-bits) whether it is read-only and the reset value of the register. For each register, there is a description consisting of a register diagram which identifies the name and extent of any used bits in the register. And each such field of contiguous bits has a natural language specification.

The challenge in creating a machine-readable specification for system registers is that different fields within the register can behave in several different ways. After some experimentation we settled on identifying five major types of field.

i) *Constant fields* have an architecture defined value and cannot be changed.

ii) *Reserved fields* are not used in the current version of the architecture but could be assigned a meaning in future versions of the architecture. These are like constant fields but, to maintain forward compatibility, software should not assume that the field is constant and should avoid changing the value of that field.

iii) *Implementation Defined fields* have an implementation defined value that programs may read to determine whether the processor has some ISA or system level feature.

iv) *Passive fields* behave like a global variable and simply store the value last written to the field. The value written often has a significant effect such as enabling address translation but this effect is completely captured by the ASL functions implementing the affected behaviour.

v) *Active fields* do not behave like a global variable: reading the field may not see the last value written to the field; writing to the field may be disabled by the value of some other register; etc. These are used for everything from system timer registers (which decrement every cycle) to allowing a hypervisor to intercept interrupts targetted at the guest operating system.

Fields that are Constant, Reserved, Implementation Defined or Passive are easy to describe completely and are described in a simple table-based format but 68 of the fields of system registers are Active fields whose behaviour can only be captured by writing ASL getter and setter functions to implement the natural language specification. The process of implementing registers with active fields proved to be quite error prone as the behaviour of the fields was rather subtle.

It was also hard to find the correct design point. We chose to identify just 5 classes of field but we could have identified further common patterns within the Active class. For example,

there are some pairs of registers that have complementary effects such as enabling and disabling exceptions. If this pattern is a one-off, it is probably best described as an Active register but if the pattern occurs in several pairs of registers, then the argument for recognizing it as a new class of field becomes stronger. As the number of tools using the system register specification grows, we expect that we will identify a number of patterns that are useful to recognise explicitly because that enables tools to make more use of the specification without having to embed the ASL parser/interpreter.

One significant aspect of system registers not yet captured in the executable specification is what Lustig et al. [8] call a *memory transistency model* which captures places where the specification allows reordering of writes to system registers with respect to other instructions and requires insertion of instruction barrier instructions (ISB) to restrict.

E. Implementation Defined Behaviour

The specification allows for some implementation defined behaviour such as whether a particular feature is implemented or the number of memory protection regions supported. This behaviour is often specified by "stub functions" returning booleans or an enumerated value and with a natural language definition. We had to implement these stub functions before we could execute the specification. In most cases, these feature test functions could be implemented by testing a corresponding implementation defined field.

F. Executable Specification

After creating all the tooling, bugfixes, etc. described above, there were some further steps required to make the specification executable so that it could be tested. We had to add additional infrastructure such as generating decode trees for a set of encodings to identify which instruction to execute; ELF readers to load test programs into memory; a physical memory implementation which allocates pages of memory on demand. and breakpoint and trace facilities to use when debugging.

We also introduced a continuous integration flow where every specification change runs regression tests. This was critical for confining new code to the ASL subset of pseudocode.

G. Machine Readable Specifications

Our primary goal in doing the above was not to make the specification executable but, rather, to improve its quality so that the specification is useful to many potential users. To support these uses, we generate a variety of machine-readable outputs.

i) *IP-XACT* is a standard XML-based format for describing registers in a chip [11]. It is used by debuggers needing to view or change the value of a register.

ii) *Callgraph summaries* are convenient summaries of the function calls and variable accesses performed by each instruction and function in the specification. One use of these summaries is in generating a summary of the list of exceptions that an instruction can raise — for inclusion in documentation.

iii) *Abstract Syntax Trees* are a complete dump of Architecture Explorer's internal representation after typechecking. We have provided these to the University of Cambridge REMS group who are in the process of transforming them into a form suitable for formal verification of machine-code programs.

IV. Trustworthy Specifications

ARM spends considerable effort on reviewing specifications. It also benefits from feedback from users of the specifications: processor designers, verification engineers, implementers of simulators, compiler writers, etc. Nevertheless, the sheer size of the specification made it unlikely that the specifications are bug-free. This was especially true of the relatively fresh v8-M specification since it had not yet had the benefit of feedback from users of the specification.

This Section describes the steps we have taken to test the v8-A and v8-M specifications using testsuites, random instruction sequences, information flow analysis and using bounded model checking to compare against the Verilog implementation of processors. One of the recurring themes of this project was that this testing process improves the specification and our trust in the specification — but it also improves the tools, verification IP, etc. that is being used to test the specification which creates a virtuous cycle of improving any other uses of those tools and artifacts.

A. Using ARM Processor testsuites

ARM performs extensive testing of its processors and simulators (it is estimated that more than 80% of the engineering effort of designing a new processor is spent on testing the processor). One part of this testing process is use of ARM's Architecture Validition Suite (AVS) which consists of programs that test the architectural conformance of individual instructions, memory protection, exception handling and all other aspects of the architecture. Excluding multiprocessor and debug tests, the AArch64 AVS consists of over 11,000 test programs with a combined runtime of over 2.5 billion instructions; the M-class AVS consists of over 3,500 test programs with a combined runtime of over 250 million instructions. Almost all of these tests were considered to be free of assumptions about instruction timing or implementation defined behaviour. (ARM has a large number of other tests which were less appropriate to run because they are aimed at testing micro-architectural performance optimizations in particular processors.)

Using ARM's official Architecture Validition Suite has some significant advantages: the suite is very thorough, checks many corner cases, and has good control and data coverage of the architecture; the suite is self-checking: each test prints "PASSED" or "FAILED" when it runs; and, since the purpose of the tests is to test processors, it was possible to compare the behaviour against actual processors for additional confidence. The primary disadvantage of using the AVS was that the tests are "bare metal" tests that exercise the System Level Architecture and require a large test harness to run.

As we started using Architecture Explorer to develop new architecture extensions (such as the new security features of v8-M), we encountered a chicken-and-egg problem: the AVS

is extended with new tests only once the architecture specification is available but we were still writing the specification. Worse, v8-M is not entirely backward compatible with the previous architecture version so we could not even run the old tests. This led us to use a hybrid approach: we temporarily created a modified specification supporting the old memory protection design so that we could use the old tests; and we created a temporary test suite to test the new security features of v8-M (see Section IV-C) before the official test suite was developed. Once updated AVS tests became available, we switched to using the official test suite.

1) Programmable Monitor and Stimulus Generator: Part of the development of every ARM processor is creating a test harness which allows the AVS to be run. This test harness consists of a programmable monitor and stimulus generator that allows programs to monitor their own behaviour at a very low-level. The test monitor design dates back to the earliest days of ARM and each successive architecture extension typically adds new test features.

The monitor consists of 177 memory mapped registers of which 45 are Active. The main features of the test monitor are

(i) Console FIFO for writing ASCII text to log file.

(ii) Memory attribute monitors which record the attributes of memory accesses in a given range of addresses. This allows test programs to verify that the MMU/MPU is correctly associating attributes such as cacheability of an access with each address. These checkers are repeated for each bus interface.

(iii) Memory abort generators to trigger a bus fault response if the processor accesses a specified range of addresses.

(iv) Interrupt generators to test triggering, prioritization and nesting of interrupts.

(v) Reset generators to schedule resets.

2) Optimizing the simulator: During this testing process, we slowly built our capability from being able to execute one instruction to being able to execute most usermode instructions, to being able to execute entire tests and then entire testsuites. As we did so, we were increasingly limited by the performance of our interpreter which initially ran at a few hundred instructions per second. Over time, we have optimized this in a variety of ways increasing performance to 5kHz (v8-A) and 50kHz (v8-M). The main optimizations applied are: (i) Memoizing a few critical functions associated with the current configuration or execution state (this has not been yet been applied to v8-A); (ii) Implementing a few critical arithmetic functions as builtin primitives even if they can be defined in ASL; (iii) Creating a C++ code generator and runtime (including ELF reader, etc.).

3) Testing the specification: One of the issues found while testing the specification initially manifested as a failing AVS test. On closer inspection, we found a mismatch between the English text and the pseudocode and that the test had originally followed the pseudocode and ARM's reference simulator followed the English text. This mismatch had been "fixed" by changing the test to match the simulator. Consulting the architects, we learned that the pseudocode was correct and the English text was wrong and so the English text, the test and the simulator were fixed to match the architects' intent.

The pass rate of our specifications on the AVS is summarized in Table I. We have achieved a 100% pass rate for the v8-A and v8-M ISA tests and for the v8-M System tests. For the v8-A System tests, there remain some failing tests in areas related to interprocessing (switching between 32-bit and 64-bit modes) and prioritization of multiple exceptions within the same instruction. These results omit debug and multiprocessor tests which are just under 50% of the total number of tests.

	ARMv8-A	ARMv8-M
ISA		
Integer	100%	100%
Floating Point	100%	100%
SIMD	100%	100%
System		
Exceptions	100%	100%
Memory	99%	100%
Interprocessing	98%	-

TABLE I: Pass rate for AVS testsuite

4) Testing the testsuite: Testing the specification with a testsuite has the side-effect of testing the testsuite. We found two classes of problems in the process of diagnosing test failures. The first is that a test may depend on some property not guaranteed by the architecture but which had been true in every tested processor. For example, a test might check that a reserved field of a register is always zero and will then fail on later versions of the architecture. Secondly, many of the M-class AVS tests depended on UNPREDICTABLE behaviour but this had not been observed before because, in practice, UNPREDICTABLE behaviour can depend on the particular pipeline state when an instruction runs.

To improve testing of the AVS, we extended the interpreter to collect line coverage information as it executes. A rare example of a coverage hole we found was in a floating point test which tested with inputs that produced the result $+0.0$ but did not test with inputs that produced the result -0.0 — with the result that one of the branches associated with rounding was not being exercised. The AVS development team now routinely measure the architectural coverage of testsuites.

B. Random Instruction Sequence Testing

Random Instruction Sequence (RIS) testing is a complementary technique to the directed testing of using hand-written tests based on generating random sequences of instructions. ARM's RIS tool [12] uses templates that specify the desired distribution of instructions, the likelihood of reuse of a given register, etc. Automatically generating random tests is different from hand-writing tests because it requires an accurate simulator to define the correct behaviour of a test. Also, because RIS generates random sequences of instructions, it is necessary to run the same test on multiple systems (processors, simulators or the specification) and compare execution traces. So at least two models are needed to develop RIS tests.

We were able to use the executable specification as part of the process for testing new RIS tests by extending the simulator to generate a trace and extending the existing trace comparision script to accept those traces. This process was especially useful for the v8-M specification because the v8-M support in ARM's reference simulator was new and had not been fully debugged. Using RIS to test the simulator against the executable specification was an effective way of testing the RIS tests, the simulator and the specification.

This process was able to uncover subtle errors in the specification. For example, v8-M's new security features splits some of the system registers into two banked registers –a non-secure register and a secure register– and the appropriate register is automatically accessed depending on the current security mode. But instructions that switch between secure and non-secure registers start in one mode and end in a different mode and the normally convenient automatic banking mechanism obscures exactly which of the two registers is being accessed. RIS testing found an error in the specification of the Test Target (TT) instruction which queries the security state and access permissions of a memory location.

C. Information Flow Analysis for v8-M

The most significant new feature of the v8-M microcontroller specification is a set of security extensions to enable secure Internet of Things applications.

To improve confidence in both the extensions and in the way they were expressed in the ASL specification, we modified the interpreter to generate dynamic dataflow graphs on which we could perform information flow analyses. Most of the analyses performed can be characterized as a non-interference property: ensuring that non-secure modes cannot see secure data and that non-secure data can only influence secure code in safe ways.

An example scenario tested in this process involved information leaks via interrupts. Interrupts automatically save integer registers on the stack of the interrupted code and zero the integer registers but, in order to keep interrupt latency low, floating point registers are lazily saved on the stack only when/if the interrupt handler uses a floating point instruction. We wanted to ensure that lazy FP state preservation did not introduce security holes. We wrote tests that iterated over all combinations of initial mode, final mode, whether FP registers had been modified and scanned the dynamic dataflow graph for information leaks.

This form of testing caught two classes of bugs. First, it caught bugs in how the architecture specification implemented the architectural intent — resulting in fixes to how the specification was written. Second, and more importantly, it caught bugs in the architectural intent by identifying potential security attacks that had not been considered before.

D. Bounded Model Checking of Processors

We have been using both the v8-A and the v8-M architecture specifications to perform bounded model checking of pipelines for processors currently under development at ARM [6]. This has primarily focused on verifying the ISA-implementation parts of the processor, not the memory system, security mechanisms or exception support. This process has been very effective at detecting bugs in various stages of processor development. But, besides verifying processors, it has another important side-effect of performing a very thorough check that the architecture specification and our tooling agrees with how the processor implementors interpret the specification. We found no errors in the published part of the specification in this process but we did find a rather subtle bug in our understanding of conditional UNDEFINED encodings and UNPREDICTABLE encodings.

The M-class specification requires that conditional execution of an UNDEFINED instruction behaves as a no-op if the condition does not hold and we had assumed that the same was true for UNPREDICTABLE instructions. During verification of a processor, the model checker detected an apparent bug that involved a conditional UNPREDICTABLE encoding but, through discussion between the processor designers and the architects, we learned that there had been a recent clarification of the architecture which said that conditional UNPREDICTABLE encodings are UNPREDICTABLE even if the condition does not hold.

This error in our interpretation of the specification had not been detected by testing because it is very, very hard to construct useful tests of the UNPREDICTABLE instructions because they are almost entirely unconstrained and can branch, change registers, trigger exceptions, etc.

E. Summary

Large specifications are as likely to contain errors as large programs so we have used many different approaches to test the specifications. In the process, we realized that although ARM publishes an official specification, the full requirements are really distributed around many different places in the company: the AVS suite, the reference simulator ARM uses for processor verification, and the processor implementations. The act of testing all these different instantiations of the specification against each other has the effect of centralizing this specification in a single location.

V. Related Work

The most closely related work is that of Goel et al. [13] who have created an executable specification of many key parts of the x86-64 ISA and system architecture including paging, segmentation and both user/supervisor levels. Their model has been verified against real processors using the Pin binary instrumentation tool and they have added a syscall emulation layer to let them run real programs including (amusingly) a SAT solver. This is a monumental piece of work that sets the standard against which other architecture specifications should be judged. Despite the similarities, our different project priorities have led to many differences: (1) They have a specification of user and supervisor levels, we also have a specification of hypervisor and secure monitor levels. (2) They have used their specification to formally verify *software* using theorem proving, we have used our

specification to formally verify *hardware* using bounded model checking. (3) They have implemented syscall emulation to let them use user-level programs as tests, we have implemented a test monitor and debugged the EL2/EL3 levels to allow us to run ARM's Architecture Conformance Suite which explores the dark corners of the architecture by running bare-metal programs. (4) They have focussed on modelling the x86-64 64-bit ISA, we have modelled the A64, A32 and T32 ISAs. (5) They have consulted processor designers to understand Intel's architecture specification document, we have had all our bugfixes and clarifications reviewed by ARM's architects and incorporated into ARM's official architecture specification document.

The most closely related ARM specifications are the Fox/Myreen ARM v7-A ISA specification in HOL [1] and Flur et al.'s ISA and concurrency specification in Sail [3] both of which were tested against actual processors using random and directed tests (8400 tests in Flur et al., 281,307 tests in Fox/Myreen). In addition to user-mode instructions, our specification covers both the ARMv8-M architecture and the larger ARMv8-A architecture, includes floating point, Advanced-SIMD and the System Level Architecture. We have tested the entire specification in multiple ways and with a larger range of values and simulated more than 2.5 billion instructions in the process. And we have used a model checker to compare the ISA specification against actual implementations for all instructions, all execution modes, all integer inputs and a subset of floating point inputs [6].

Shi [14] extracted the ISA pseudocode from ARM's v6 Architecture Reference Manual, automatically translated the code to Coq and used that to verify that the ARM model in the SimSoC simulator written in C faithfully implemented the Coq specification. This is an impressive piece of work, and it would be interesting to repeat their work using our new, more trustworthy specification or to extend their proof to cover the system level architecture.

The other major ARM ISA specification that we are aware of is embedded in the CompCert compiler and is used in the proof that the compiler faithfully translates the input C program to ARM assembly code. This specification is limited to a subset of the user-mode ARMv6 specification and there is no published statement of how it was validated.

Hunt created a specification of the FM8501 processor [5] and used it to formally verify the processor. The process of formal verification greatly increases the trust we can place in the corresponding parts of the specification because it ensures that all the corner cases in both the processor and the specification have been explored.

More broadly, anyone wrestling with a large specification is obligated to find ways to verify that the formal specification captures the (informal) requirements.

VI. Conclusions

Historically, ARM's specification efforts have focused on a single set of products: the ARM Architecture Reference Manuals [9], [10]. However, there are many more potential uses of the specification if the specification is delivered in a flexible, machine-readable format – for example, formal verification of hardware and software, tools that manipulate instruction encodings, debug tools, creating hardware verification tests. Traditionally, all these other users manually transcribe parts of the specification into some other notation: HOL, C, Verilog, spreadsheets, etc. This process is laborious and error-prone but, worse, it is fragmented: bugfixes or clarification found by one group are not necessarily propagated to other groups or to the master specification. Our primary goal in this project was to enable formal verification of ARM processors against the specification. But, by supporting as many of these uses as possible, we created a virtuous cycle where bugfixes or improvements were incorporated into the central specification so that all users benefit from bugfixes as well as to amortize the development effort across many uses.

This paper describes the steps required to create trustworthy specifications of the full v8-M and v8-A architectures including the instruction set architecture, memory protection and translation, exceptions and system registers. While checking that a formal specification captures the architects' informal intent is an unending process, we believe that our specification is the most trustworthy and complete system specification of any mainstream processor architecture.

We are currently working with Cambridge University on a public release of our specification suited to verification of machine code programs.

References

[1] A. C. J. Fox and M. O. Myreen, "A trustworthy monadic formalization of the ARMv7 instruction set architecture," in *Proc. Interactive Theorem Proving ITP 2010*, ser. LNCS, vol. 6172. Springer, 2010, pp. 243–258.

[2] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[3] S. Flur *et al.*, "Modelling the ARMv8 architecture, operationally: concurrency and ISA," in *Proc. Principles of Programming Languages, POPL 2016*, 2016, pp. 608–621.

[4] M. Dam, R. Guanciale, and H. Nemati, "Machine code verification of a tiny ARM hypervisor," in *Proc. Workshop on Trustworthy Embedded Devices*, ser. TrustED '13. ACM, 2013, pp. 3–12.

[5] W. A. Hunt, "FM8501: A verified microprocessor," ser. LNCS, vol. 795. Springer, 1994.

[6] A. Reid *et al.*, "End-to-end verification of ARM® processors with ISA-Formal," in *Proc. Computer Aided Verification (CAV)*, ser. LNCS, vol. 9780. Springer-Verlag, 2016, pp. 42–58.

[7] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, pp. 7:1–7:74, 2014.

[8] D. Lustig *et al.*, "Coatcheck: Verifying memory ordering at the hardware-OS interface," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS*. ACM, 2016, pp. 233–247.

[9] ARM Ltd, *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. ARM Ltd, 2013.

[10] ——, *ARM v7-M Architecture Reference Manual*. ARM Ltd, 2006.

[11] IEEE, "IP-XACT, standard structure for packaging, integrating, and reusing IP within tool flows," *IEEE Standard 1685-2014*, 2014.

[12] B. Greene and M. McDaniel, *The Cortex-A15 Verification Story*. http://www.testandverification.com/downloads/DVClub-Jan-2012/Cortex-A15-Verification-Story-DVclub-final.pdf, 2011.

[13] S. Goel *et al.*, "Simulation and formal verification of x86 machine-code programs that make system calls," in *Formal Methods in Computer-Aided Design, FMCAD*, 2014, pp. 91–98.

[14] X. Shi, "Certification of an instruction set simulator," Ph.D. dissertation, University of Grenoble, July 2013.