School of Informatics

Informatics Project Proposal
Generating an ARM Instruction Decoder for Captive

[redacted]

Abstract

Architecture specifications outline the precise behaviour of processors, but do so with a mixture of prose and pseudocode; which makes the process of transcribing instruction decode logic time consuming and error prone. ARM publishes a machine readable form of their architecture specification to facilitate the automation of emulation and analysis. This paper outlines how Captive, a research hypervisor, would benefit substantially from a mechanically-derived instruction decoder, how such a system could be developed, and a full outline of the project to do so.

Date: Tuesday 21st April, 2020

[redacted]

1 Motivation

Architecture specifications are fundamental communication tools in facilitating the development of software that can rely on the behaviour that processors will conform to. It is vital that the specifications are correct, accurate, and clearly communicate the behaviour of the hardware. Despite these fundamental requirements, these specifications have traditionally been published as a mixture of prose and pseudocode[1]. Academics, researchers, and engineers have then transcribed these definitions into a myriad of verification, simulation, and code generation tooling, which is incredibly time consuming, prone to errors, and is a full time job to keep up with changes[1]. With the publishing of machine-readable forms of architecture specifications[1], it is possible for this transcription work to be mechanised; reducing the necessary human labour and preventing the introduction of bugs into the resulting system.

Captive[2] is a re-targetable cross-architecture hypervisor (a tool for emulating a processor on a machine of the same or different architecture). As such, it relies on a transcribed ARM specification to decode the ARM-targeted binaries; a manual process that has been identified as both time consuming and regularly prone to errors. Given a machine readable form of the ARM specification is published by Arm[3], it seems reasonable to suggest that this process can reasonably be automated.

1.1 Problem Statement and Objectives

To elaborate further, the objective of this project will be to develop a tool that can parse the ARM specification documents (published as XML), produce a decode tree, and then generate code that can parse ARM instructions – automating the currently labour-intensive process.

The aims of this tool is to enhance the reliability of Captive's ARM decode functionality (in that it will not contain errors introduced through manual transcription of the decode logic), add support for all ARM instructions, and ease the maintenance as the architecture continues to evolve.

1.2 Timeliness and Novelty

The number of embedded and autonomous devices (the Internet of Things) have exploded in recent years; devices that are becoming an increasing security and safety concern as they are deployed into environments that make their software difficult to patch and applied to use cases – such as medical, military, and transportation – where failure comes with an increasingly greater cost. Due to a variety of factors, the primary being power consumption, the vast majority of small processing devices are based on the ARM architecture and as such it is the most valuable to focus research efforts on. This work is possible now that ARM publishes their specification in a machine readable format, and has proven their commitment to doing so through a number of revisions.

1.3 Significance

The manual transcription of instructions into decode tools for the purposes of simulation, verification, etc is a slow process that is prone to errors. It involves fine details and an understanding of the meaning expressed in the largely prose descriptions and accompanying pseudo-code. The cutting-edge approach is to transcribe these definition mechanically from machine-readable definitions facilitating a smooth process that is void of errors. This dependence on verifiable documentation delivers a greater level of trust in the downstream systems and ensures that as the underlying architecture continues to evolve the maintenance costs of ensuring compatibility are negligible.

Captive is one such tool that would benefit enormously from have a complete set of definitions for all ARM instructions, as well as ensuring that those definitions are exactly as the architecture specifies. This, in turn, provides a great level of trust in all software that is executed as part of the Captive system.

1.4 Feasibility

In the development of this project we are applying existing algorithms[4] to implement the software solution, following the existing, manual process of transcribing instruction definitions closely. Furthermore, we will be solely utilising readily available documentation and development resources that are not subject to constrained access or restrictive intellectual property. As a result of the clarity in the definition of its goals, its sole dependence on open technologies, and the application of known techniques, we have a strong confidence that there is nothing to suggest that this project will prove infeasible.

1.5 Beneficiaries

The beneficiaries of this work can be reasonably divided into four groups based on their use of the Captive tool that this research will enhance, or the indirect benefits that they gain from the development of systems that relies on emulation tooling.

Researchers Captive is first and foremost a research endeavour and any improvements to its capabilities will primarily be felt by those in the research community. Furthermore, the application of the techniques that will be used in this project will be of interest to other similar projects, such as the REMS project[5].

Hardware Engineers Hardware engineers prototyping new embedded systems make extensive use of emulation tooling; with dependability being key to ensuring the development of safe and reliable devices.

Software Engineers Emulation is used by software engineers when actual devices are difficult to test on or are not yet available. One such domain is mobile app development where a myriad of ARM-architectured devices are being targeted on typically x86-architectured development machines. Reliable emulation is critical in ensuring only minimal changes are required when software is finally tested on actual hardware.

End Users End users also benefit, albeit indirectly, from reliable, efficient emulation technologies through the rapid development of new systems and continued support of legacy ones.

2 Background and Related Work

2.1 Emulation

Captive[2] is a research-grade cross-architecture hypervisor designed to utilise the host architecture's specialised emulation instructions to improve efficiency, much as is commonplace for same-architecture hypervisors. For example, the guest's (e.g. ARM's) memory model is mapped to the host's (x86) - which is far more efficient than software virtualisation. This work aims to improve the instruction decoder logic of the hypervisor; the development of which currently involves manual transcription of the ISA specification into the architecture description language supported by Captive's generation tool – a slow and error-prone process.

ARM itself provides some tooling – what they refer to as Fast Models[6] – that allows some degree of user-configured virtualisation. However, due to scalability issues it has limited applicability and thus does not solve the emulation problem in a generic sense[2]. Given this is developed by ARM themselves it is not clear how the instruction decoder was generated.

The most common cross-architecture emulator is QEMU[7] which is used in a myriad domains, including forming the core of the Android emulator. Its instruction decoder for each guest architecture is manually coded; a translation of instructions into QEMU's intermediate model.

2.2 Machine-Readable Specification

In 2017, Arm began publishing a machine-readable form of their ISA specification detailing the particulars of everything from the memory model, to the registers, to - crucial for our purposes - a full description of the machine instructions and their behaviour. This work was spearheaded by Alastair Reid who has detailed a great deal of this process and how to consume these documents on his website[8]. This includes explanations and code snippets of how to dissect the instruction definitions[9].

2.3 Derived Instruction Decoders

The development of the SAIL language[10], by the REMS project[5], for describing ISA semantics utilises the machine-readable ARM specification. Whilst this was less concerned with the particulars of the instruction definitions, the work highlights the possibility of using the machine-readable specification documents to rapidly support the ARM ISA. This work also highlights some other noteworthy points that have been taken on board in the planning of this project. Firstly, their evaluative measure of executing Linux to demonstrate coverage is a technique that we will seek to replicate, see section 4, as a strong measure of completeness. Also, in defining their domain-specific language they made design decisions to optimise for the accessibility of the code to a wider variety of developers so as to ensure that there was a general consensus that the system was correct. We have taken this advice into account when selecting the development tool-set with which this project shall be implemented – see section 3.1.

There have been other efforts to generate an instruction decoder from a machine readable definition of the shape of machine instructions - in particular, how this can be done efficiently by applying optimisation algorithms and experimenting with heuristics. A regular-expression-like domain-specific language for defining the structure of machine instruction op-codes, from which a decoder tree can be automatically derived and decoder logic generated is defined by [4]. The particular algorithms that will be utilised from this research are outlined in section

3.2; the code itself, if published, no longer appears to be available online.

3 Programme and Methodology

3.1 Tooling

Given the constraints that the input to the system must be the XML files that constitute the ARM ISA specification and the output must be a C++ implementation of an instruction decoder compatible with the Captive hypervisor, the choice of language and tooling reflects the nature of the problem and desired properties of the resulting software. It is desirable that the system be built atop a mature and cross-platform set of technologies, with a variety of libraries for performing key tasks like XML parsing, generating code, and rigorous unit testing. The source code should also be easy to understand for all interested parties to ensure that the resulting instruction decoder is trustworthy. As such, the Python programming language seems most applicable, applied within a test-driven development methodology to ensure the resulting system is correct and reliable.

3.2 Components

There are four main development activities, as outlined below, contributing to three software components - with the fourth activity being an optimisation effort of the second, and the fifth a process of evaluating those programs.

3.2.1 Parsing

The first of the three main components, parsing of the XML files that constitute the ARM specification into a form that can be further manipulated is the easiest of the four development activities and should be the shortest in required time. Utilising existing libraries for parsing XML, it should be fairly straightforward to extract the required information. Alastair Reid's explanatory blog post[9] will act as a guide in this. Some additional manipulation of the data may be required to simplify the further work such as marking alias instructions and simplifying the initial set of considered instruction to aid debugging.

3.2.2 Construct Decode Tree

The process for generating a decode tree (and optimising it, see section 3.2.4), will be a process of implementing published algorithms[4]. This starts with the naive approach of branching for each bit in the op-code (ARM instructions are not variable-width), optimising by grouping bits in such a way to reduce the size of the decode tree, and then further optimising as described in the optimising section below (section 3.2.4). Initially, this should be a straightforward process and a naive prototype should be quick to complete. However, optimising this process so that the generated decode tree is practical (and not exponential on the number of bits) will take some time longer. Therefore, once an initial prototype is complete, the next section of generated C++ code will be started as it is of higher priority.

3.2.3 Generate C++ Decoder

The final mandatory implementation activity to prove the potential of the projects hypothesis is that of generating the code that when executed decodes a given ARM instruction. This will be generated from a decode tree, generating switch-case statements in a manner similar to the SAIL project[10]. The Python ecosystem provides a number of libraries for generating templated code, from the standard format function (the named variables make it particularly powerful) to domain-specific libraries like Jinja[11] and Cheetah[12]. Early prototyping will be used to evaluate the options based on their applicability to the use-case.

3.2.4 Optimising the Decode Tree

Up to this point the development unfortunately follows a waterfall model, with the resulting risks that are identified in section 3.3. Optimising the decode tree is highly desirable due to the expected computational costs of executing the more naive implementations, however it is not vital in proving the hypothesis. As such, it has a lower priority than the evaluative steps of completeness and correctness and will be dropped if for unforeseen reasons the prior implementation tasks take longer than expected.

The algorithm, based on the principle of compacting common sub-trees, is detailed at length in "Efficient Software Decoder Design"[4].

3.2.5 Evaluation

The evaluative procedures outlined in section 4 will then be conducted. The different activities have dependencies on different implementation stages and will be conducted as soon as possible, ordered by priority. For a complete breakdown, see section 6.1, and figure 1.

3.3 Risk Assessment

The following table highlights the most significant risks to the completion of this project – including development and evaluation of the software.

Description	Impact	Likelihood	Mitigation
Unforeseeable development issues that cause delays to the laid out timeline. This is by far the greatest risk.	**HIGH** – Speaking from experience, accurate estimates of software development are hard and projects often overrun and an incomplete system would not meet the goals.	**MODERATE** – Effort has been taken to produce accurate estimations but there are numerous unknown variables	A minimum prototype will be the priority. Optional features and optimisations have been noted as such where they have been described elsewhere in this document. Tooling has been chosen to ease rapid prototyping.
Necessary waterfall development model prevents evaluation until after most development	**MODERATE** – Most of the evaluative measures are dependent on a complete system which may be delayed (see above). In which case, there may be insufficient time for rigorous evaluation of the complete system.	**MODERATE** – Same as above	Many evaluative measures are considered optional, see section 4. Required evaluation procedures will be completed prior to optional development activities.

3.4 Ethics

There are no ethical concerns with this work as there is neither data concerning any persons nor any wider implications of the existence of this software tool given it simply automates an existing (albeit manual) process.

4 Evaluation

The success of this project will be evaluated on three key metrics, of decreasing importance: correctness, completeness, and performance. A procedure to evaluate the deliverables by these metrics is outlined here, as well as mention of an additional procedure that will highlight the value of this endeavour.

Correctness We will take some simple compiled C program that is known to be executed correctly when emulated by the Captive hypervisor with its current, manual-coded instruction decoder. We will compare the execution of that program when then emulated by Captive with the new, ISA-derived instruction decoder, and assert the same behaviour. This behaviour may then also be compared to execution on actual ARM hardware, time permitting (see section 3.3). Ensuring the correctness of the generated decode code is vital in ensuring that this project adds value to the existing transcription process.

Completeness Ensuring that the decode logic is complete - in that it decodes all of the instructions in the ARM instruction set - follows from the fact that the formal ISA specification includes all of the instructions and from that we will derive the decode logic. Executing linux in the emulated environment is a common procedure in the literature to demonstrate that all of the standard functionality of the architecture is supported. We will perform the same, by providing Captive with the generated instruction decoder and demonstrating that a simple linux distribution targeted at the ARM architecture can be emulated on an x86 machine using the generated instruction decoder.

Performance Performance of the decoder generating program itself is not important, as it will only be executed once with each version of the ARM architecture, and will therefore not be measured: we will simply ensure that it executes in a reasonable period of time. The generated decoder itself will, time permitting, be optimised to be as performant as the current, manually-transcribed decoder. This will be demonstrated through timing the completeness evaluation procedure and other similar examples to be determined. This is not the greatest priority and will also form part of the risk mitigation strategy, see section 3.3.

Demonstrate Value Given the generated decoder will be shown to be correct, it would be desirable to identify the advantage of this clearly be attempting to identify bugs in the existing manually-transcribed decoder. By iterating over all instructions supported by the manual decoder, an attempt will be made to identify if there are any inconsistencies between the decoders, thereby demonstrating the value of this project not just in terms of improving the maintainability of Captive but also its reliability.

5 Expected Outcomes

The primary expected outcome of this project is that the Captive hypervisor will be enhanced through the replacement of its manually transcribed instruction decoder with a decoder mechanically derived from the published machine-readable ARM architecture specification. As such, it will be more reliable in that it will be free from bugs inadvertently added during the manual transcription and will now support all instructions; where at the current time it only supports those that have been identified as required in the conducted research.

Furthermore, this research will also add to the body of literature demonstrating the value of machine-readable ISA specifications that together will continue to convince processor manufacturers to publish these documents.

Ultimately, this work will contribute towards improving the productivity of engineers across computing domains and aid the development of safe, reliable systems.

5.1 Further Research

The outcome of this research will be extendable in two dimensions.

The more immediate contribution would be extend the program to support the parsing of ASL, the language with which the instruction behaviour is defined, to automatically generate the execution of instructions within Captive. We estimate that this would take several months to accomplish and it is for that reason that it is out of scope of this project.

Moreover, an interesting piece of work would be to formally verify the output decoder to ensure that it is correct according the specification. This would be a significant step in ensuring the reliability and safety of the Captive's instruction processing functionality, but would by no means be trivial to accomplish.

6 Plan, Milestones, and Deliverables

6.1 Plan

The project can be quite evenly split into implementation and evaluation (though it is expected that implementation will take significantly longer – see Figure 1). Outlined here are the steps and their dependencies, shown graphically – again – in Figure 1.

Implementation As outlined in section 3, there are four steps to the implementation.

Name	Required?	Priority	Dependencies
Parsing	Yes	HIGH	None
Build Decode Tree	Yes	HIGH	Parsing
Generate C++ Decoder	Yes	HIGH	Build Decode Tree
Optimise Decode Tree	No	MODERATE	Build Decode Tree

Evaluation As outlined in section 4, there are four steps to the evaluation of the project.

Name	Highly Desirable	Priority	Dependencies
Correctness	Yes	HIGH	Generate C++ Decoder
Completeness	Yes	HIGH	Generate C++ Decoder
Performance	No	MODERATE	Optimise Decode Tree
Demonstrate Value	No	LOW	Generate C++ Decoder

Dissertation Whilst most dissertation writing will be completed throughout the development and evaluation stages where possible, some time has been allocated in figure 1 for the completion of parts that have particular dependencies.

6.2 Milestones

The following key milestones outline the absolute latest that the key components must be completed. Ideally, they will be completed earlier and there will be sufficient time for the additional components outlined in other sections. Figure 1 shows the dependence and ideal timeline of all components, with the milestones marking when their dependence must be completed. Note the unavoidable waterfall nature of the implementation activities up to Milestones 3. The activities identified as Required or Highly Desirable in the tables above are shown in the figure 1 in red.

Milestone	Week	Description
M_1	3	"Parsing" completed
M_2	4	"Decode Tree" prototype completed
M_3	7	"Generate C++" so all "Required" implementation completed
M_4	9	"Highly desirable" evaluation completed
M_5	11	Submission of dissertation

6.3 Deliverables

The sole deliverable will be a software system that given a version of the machine-readable ARM ISA specification, will produce an instruction decoder for the Captive hypervisor.

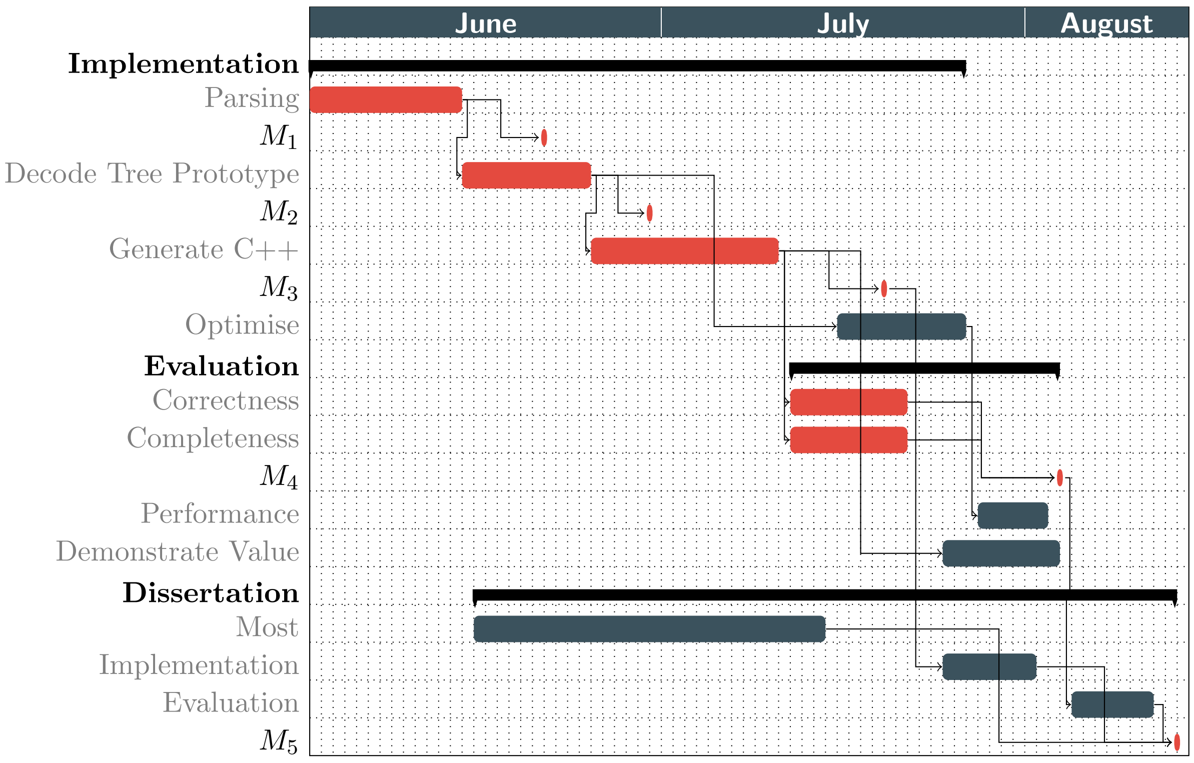

Figure 1: Gantt Chart.

References

[1] Alastair Reid. Arm releases machine readable architecture specification. "https://alastairreid.github.io/ARM-v8a-xml-release/". Accessed: "2020-03-17".

[2] Tom Spink. Efficient cross-architecture hardware virtualisation. 2017.

[3] Arm Holdings. A-profile architectures — exploration tools – arm developer. "https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools". Accessed: "2020-03-26".

[4] Todd Austin Rajeev Krishna. Efficient software decoder design. 2001.

[5] Rems. "https://www.cl.cam.ac.uk/~pes20/rems/". Accessed: "2020-04-09".

[6] Arm Holdings. Fast models – arm developer. "https://developer.arm.com/tools-and-software/simulation-models/fast-models". Accessed: "2020-03-26".

[7] Fabrice Bellard. Qemu: a fast and portable dynamic translator. *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[8] Alastair Reid. Alastair reid – researcher at google. "https://alastairreid.github.io". Accessed: "2020-03-26".

[9] Alastair Reid. Dissecting arm's machine readable specification. "https://alastairreid.github.io/dissecting-ARM-MRA/". Accessed: "2020-03-17".

[10] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. Isa semantics for armv8-a, risc-v, and cheri-mips. *Proc. ACM Program. Lang.*, 2019.

[11] Pallets Projects. Jinja — the pallets projects. "https://palletsprojects.com/p/jinja/". Accessed: "2020-03-26".

[12] Cheetah3, the python-powered template engine — cheetah3 - the python-powered template engine. "https://cheetahtemplate.org". Accessed: "2020-03-26".