Expressing Intent

An evaluation of the Arm Machine Readable Specification

Expressing Intent

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jeroen Kloppenburg
born in Haarlem, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Expressing Intent

Author: Jeroen Kloppenburg
Student id: 4477960

Abstract

The behaviour of software is intrinsically linked to the hardware it runs on. As hardware innovations continuously change the landscape of computing, software has to adapt to these changes. Running legacy software on new hardware requires either the old hardware to be emulated, or a very time-consuming and error-prone process of reverse engineering the software to determine its behaviour, and then writing new software that has the same behaviour, but runs on the new hardware. Binary lifting and translation tools aim to automate this process, but are often limited by the lack of accurate and complete instruction semantics.

This project aims to evaluate the feasibility of using the Arm Machine Readable Specification to aid in this process. The specification is a machine-readable description of the Arm architecture, including self-proclaimed "executable" instruction semantics, written in a specification language called ASL. This project has created an emulator that can run Arm instructions on a non-Arm architecture, using the specification to determine the behaviour of the instructions. The emulator is able to run simple programs with little context, but is not yet able to run more complex programs, due to the lack of support for behaviour that depends on the specific hardware implementation, and context dependencies outside of the instruction set, such as operating system interactions.

This emulation based solely on the specification has demonstrated that it is a promising approach to reason about the behaviour of Arm programs, but it is not complete enough to be used for binary lifting and translation. The context dependency of executables, and the lack of implementation specific behaviour in the specification are the main reasons for this.

Thesis Committee:

Chair:	Prof. dr. K.G. Langendoen, Faculty EEMCS, TU Delft
University Supervisor:	Dr. S.S. Chakraborty, Faculty EEMCS, TU Delft

Preface

This thesis serves as the final project for my master's degree in Computer Science at the Delft University of Technology.

I have always loved working with the lowest level of abstraction, so while talking with the late Eelco Visser about possible topics for my thesis, he introduced me to my daily supervisor, Soham Chakraborty, who had a project in mind that would have me working on adding support for preserving concurrency semantics during binary lifting of Arm executables to an existing project called llvm-mctoll. Partway through the project, it became clear however that some of the assumptions we had made about llvm-mctoll were incorrect, in that it was incapable of lifting even the most basic Arm executables, and that the project was not in a state where it could be used for the project. Several months were spent trying to get llvm-mctoll to work, but in the end, after basically rewriting the entire project, it became clear that it would not be possible to get any meaningful results out of it. At this point, we decided to change the focus of the project to something that would be more feasible to accomplish in the remaining time. What followed was a period of small, tangentially related projects, until we eventually settled on the current topic of working with the Arm specification.

Overall, this project has been a stressful and frustrating experience, with many setbacks and unexpected problems.

Jeroen Kloppenburg
Delft, the Netherlands
July 5, 2023

Contents

Chapter 1

Introduction

1.1 Motivation

Computing technology is ever-changing, with new innovations in software and hardware driving the field forward. Where in the past, programmers had to use clever low-level hacks to squeeze the last bit of performance out of a system, now the focus is on very high-level concepts such as what kind of machine learning model would perform best on a dataset, what kind of lighting makes a 3D render look more realistic, or how to structure the back-end of a web application to make it more scalable. This is a good thing, as it allows users to focus on the task at hand, rather than having to worry about the details of the system they are using, but it also means that the systems they are using are becoming increasingly complex and opaque. However, despite this progress, many outdated systems continue to be used due to their heavy reliance on legacy software and hardware. Additionally, as computing systems become increasingly complex and pervasive, security vulnerabilities and bugs are major concerns, particularly in systems that have a significant impact on society, such as medical devices and financial systems. In order to address these issues, it is important to be able to update these legacy systems. The main challenge in doing so is that the software is so closely tied to the hardware it is running on, that it is often not possible to move the software to a new system without significant changes.

Reverse engineering is the process of recreating higher-level representations of programs from compiled machine code. This can be very time-consuming and challenging when done manually, leading to a growing interest in automating various parts of the process. By creating higher-level representations of programs, it becomes easier to detect bugs and security vulnerabilities, as well as modify and recompile programs to target different architectures. For legacy software, one of the goals of this process can be to reach a full binary translation, where the original code can be completely transformed into a new program that runs on a different architecture, preferably with as little manual

intervention as possible. The best way to accomplish this is to lift or raise the program to a level of abstraction high enough that it can be compiled later.

Key to this process is having a way to represent the behaviour of a program. Instruction semantics refer to the meaning and behavior of individual machine instructions. They play a critical role in this lifting process, as this involves mapping the original instructions to constructs in the higher level representation. Many existing tools that do binary analysis or translation have their own system to represent and work with these instruction semantics. Writing these instruction semantics and making sure they are correct with respect to the actual behaviour of the instruction on hardware is again time-consuming and error-prone.

A recent project by ARM has created a machine-readable specification for large parts of their architecture, including "executable" instruction semantics written in their own specification language called ASL. If this specification is comprehensive enough, it could be used to replace the existing instruction semantics in many tools. This would not only reduce the amount of time and effort spend by developers of individual tools to write their own instruction semantics, but it would also allow for more accurate and complete tools, as the semantics are not only written by the authority on the architecture, but any and all improvements to the specification would automatically be reflected in the tools.

1.2 Problem Statement

This thesis aims to discuss the possibility of using formal specification to manipulate binaries by evaluating the accuracy and limitations of the ARM machine-readable specification. To do so, an emulator is created that uses the specification to run ARM instructions on an x86-64 machine. The development of this emulator brings to light the limitations of the specification, and the more general challenges that arise when trying to run programs on hardware that it was not designed for.

1.3 Overview

The background chapter will provide essential background information on binary lifting, semantics, ARM architecture, and the current landscape of tools. The methodology chapter will describe the approach taken in working with the ARM Machine-Readable Specification and the development of an emulator that uses this specification. The evaluation chapter will present the roadblocks encountered and the results of trying to run programs on the emulator, including its performance and current limitations. The discussion chapter will explore future research directions, including the need for more detailed specifications and the potential for OS-specific specifications. Finally, the conclusion

will summarize the limitations of the ARM machine-readable specification and outline areas for future work.

Chapter 2

Background

Since the aim of this thesis is to explore the use of the ARM Machine-Readable Specification for binary analysis and lifting, it is important to understand these concepts in more detail. This chapter will provide a brief overview of the ARM instruction set architecture, and other concepts that are important to understand the rest of this thesis. An Instruction Set Architecture, or ISA, describes an abstract model of a computer. It is used as a simple layer of abstraction over the specific hardware implementations. An ISA is usually described in enormous documents, describing behaviour in natural languages and occasionally pseudocode. A program written in a high-level language is compiled into sequences of instructions, and combined with data to form an executable binary file.

2.1 Binaries

In order to execute a program, it has to be loaded by a different program called the loader. The loader is responsible for setting up the context of the executable, such as setting up all the relevant memory regions, loading the executable into memory, setting up the initial stack and registers, and loading any shared libraries that the executable depends on. File formats differ between operating systems, but they all contain roughly the same information. The main executable formats are ELF for Unix systems, PE for Windows, and Mach-O for macOS. For the sake of this thesis, we will limit ourselves to the ELF format, but the concepts are similar for other formats.

ELF

The Executable and Linkable Format (ELF) is the file format for executables and shared libraries for most Unix systems like Linux. In order to deal with dynamically linked code, the system first has to load the dynamic linker, which reads the dependencies and performs relocations on the program by replacing

placeholder values with the resolved addresses of the dependencies. Because very large programs may require an incredible amount of relocations, the Procedure Linkage Table (PLT) and the Global Offset Table (GOT) are used as a level of indirection in order to lazily resolve the dependencies. Instead of resolving all the relocations at the start, the first time a symbol is used, the dynamic linker will resolve it, after which it will be cached.

The segments within an ELF file contain data or instructions. The instructions are written under the assumption that it will be executed by a specific architecture, so the instructions do not explicitely describe their meaning. For that, we have to look at the specification of the architecture.

2.2 Semantics

Semantics is the study of meaning. Formally talking about logic and theoretical meaning of programs or parts of programs has natural parallels to the study of meaning in natural language.

Meaning

The meaning of a word can be literal or figurative. A literal meaning is the meaning of a word as it is defined in a dictionary. A figurative meaning is the meaning of a word as it is used in a sentence. The literal meaning of a word is the same for everyone, but the figurative meaning is dependent on the context of the sentence. Likewise, the meaning of a part of the program can depend on the level of abstraction you are talking about, so I will use the terms operational semantics and natural semantics to differentiate between the literal and more abstract meanings.

Operational semantics refers to the denotation or literal meaning of a construct. In case of a machine instruction, this means the actual observable behaviour as a transformation of the system within the context it is executed in. Natural semantics refers to the connotation or intention of a construct. This is the higher level meaning that is implemented by the operational semantics, and as such are usually emergent properties of the context in which some operational semantics are placed.

An instruction that adds two numbers has very clear operational semantics. It takes the two numbers and produces their sum. The natural semantics of this instruction is dependent on the context. It could be just the addition of two numbers for the sake of their sum, but it could also be the calculation of an address to load a value from, or the address of a function to call.

Programming

With a perfect understanding of how a computer interprets instructions, programming becomes the art of encoding intent. The intent is usually encoded

into a set of plain texts in a rather constrained language. These texts are parsed, analysed, and lowered to a more compact representation that is optimized for the specific computer that runs the program. This transformation breaks down abstractions into their implementations, which depend on the specific context of the computer; the operating system it runs, the hardware available to it, even the presence of other programs.

The resulting representation is usually optimized based on some desired properties such as overall runtime or memory usage. This might change the exact operational semantics of the program in a way that preserves the natural semantics of the program.

While lowering abstractions, the program accumulates dependencies on other pieces of code, usually code that interacts with the operating system. These pieces of code are usually not directly inserted into the program, but rather linked into it. Like how the meaning of a text can depend on circumstances not contained within the text, the meaning of a program can depend on context not contained within the program, so figuring out the meaning of a program may require more than just the program itself.

2.3 Binary Analysis

Binary analysis is the process of analyzing executable code or binary files to gain an understanding of their functionality, behavior, and potential vulnerabilities. Binary analysis is often used by security researchers, malware analysts, and reverse engineers to identify security issues and potential threats in software. In addition to security analysis, binary analysis can also be used for software optimization and performance tuning. By analyzing the binary code and identifying areas of inefficiency or bottlenecks, developers can make improvements to the software to optimize its performance and reduce its resource usage.

Static Analysis

Static binary analysis involves analyzing the binary code at the assembly level without executing it. This can be done using disassemblers and decompilers that improve the readability of the code by looking for patterns in the code and replacing them with higher-level constructs.

Dynamic Analysis

Dynamic binary analysis involves running the binary in a controlled environment in order to observe its behaviour. Dynamic analysis is usually done using a debugger, which allows the user to step through the program and observe the state of the program at each step. This is usually more effective than static analysis, as programs often have complex behaviour that is difficult to

predict from the code alone. One obvious example is obfuscation, where the code is deliberately made difficult to understand in order to hide its true functionality, something that is very prevalent in malware. Another example is the use of code generation, where the code being executed is generated during execution, which used to be common on older systems to work around the lack of memory. The obvious downside of dynamic analysis is that only executed code can be analyzed, so it is possible to miss important parts of the program. One way to mitigate this is to use a type of symbolic execution, but this is computationally expensive for non-trivial programs.

2.4 Existing Tools

Working with instruction semantics is not limited to a single, narrow field of research. The entire discipline of reverse engineering is concerned with the semantics of binary code, and as such, there are many tools that are relevant to this work, with goals ranging from security analysis to software optimization. The rough categories of these tools are **analysis**[12, 11, 36, 10], **disassembly**[37, 3], **emulation**[19], **instrumentation**[22, 26], **lifting**[23, 31, 30, 16, 15], **rewriting**[2, 7, 35], **symbolic execution**[6], **and virtualization**[25]. Because of the overlap in these fields, there are also many "framework" projects that do a bit of everything[9, 27, 28, 17, 18, 33, 34, 13, 14]. Many tools in these fields are either very minimalistic and start and end their lives as academic research projects, or are heavily sponsored or commercialized projects developed by many people over many years. Special mention goes to QEMU[8], a full system emulator, and IDA[20], the "de-facto tool" for binary analysis. At different stages of the project, we have looked at many of these tools, specifically how they represent instruction semantics, and how they use these semantics to perform their analysis.

2.5 ARM Architecture

The ARM architecture is a family of microprocessor architectures that are used in a wide range of electronic devices, from smartphones and tablets to servers and supercomputers.

The ARM architecture is based on a reduced instruction set computing (RISC) design, which means that the instructions that the processor can execute are simplified and streamlined, resulting in faster and more power-efficient performance. RISC architectures like ARM are typically used in devices that have limited power and memory resources, such as mobile phones and tablets.

ARM defines three architecture profiles, A, R, and M; the Application profile, the Real-time profile, and the Microcontroller profile. The Application profile is the most general profile, and is used in most devices. The Real-time profile is used in devices which require real-time guarantees, such as medical

devices. The Microcontroller profile is used in devices which have very limited resources, such as microcontrollers. From this point on, we will only talk about the Application profile.

Instruction Sets

As with most architectures that are multiple decades old, the ARM architecture has evolved a lot over time. One facet of this evolution is the creation of specialized instruction sets and extensions that improve performance and efficiency in certain situations.

The main instruction sets are A32, the 32-bit ARM instruction set; T32, the mixed 16- and 32-bit Thumb-2 instruction set; and A64, the 64-bit ARM instruction set. In addition to these instruction sets, there are many extensions, including but not limited to; SVE (Scalable Vector Extension), VFP (Vector Floating Point), NEON (Advanced SIMD), Helium/MVE (M-Profile Vector Extension), RME (Realm Management Extension), and MTE (Memory Tagging Extension).

Memory Model

The ARM architecture uses a simple addressing load/store model with weak memory ordering. This means that the processor can execute instructions in any order, and that the processor does not guarantee that memory operations are performed in any particular order, unless explicit memory barriers are used.

Because of this simple addressing mode, instructions can only perform operations on registers, and all load/store addresses are purely determined from register contents and instruction fields. This is in contrast to other architectures, such as x86, which have a complex addressing mode.

Conditional Execution

Most instruction sets only allow conditional execution for branches. ARM has conditional execution encoded in almost all instructions in its A32 and A64 instruction sets by means of the first 4 bits of each opcode. Because of the small size of the T32 instructions, a special instruction is used which sets conditional execution for the next couple of instructions. This starkly contrasts with other architectures, such as x86, which only allow conditional execution for branches. This allows for more compact code and reduces the amount of branching required, but it also makes it harder to statically analyze the code.

Chapter 3

Methodology

In this chapter, we describe the structure of the Arm Machine-Readable Specification, and the overall process of working with it. We also describe the overall emulator design choices made in preparation for the implementation of the emulator in the following chapter.

3.1 The ARM Machine-Readable Specification

When hardware is designed, it is of the utmost important that the behaviour of the hardware is well-defined, such that software can be written that interacts with this hardware. As processors have evolved over the years, their specifications have not only become more precise, but also significantly larger. As these specifications usually contain enormous amounts of explanations written in a mixture of natural language and pseudocode, this makes it difficult to check the correctness of the specification, or to verify the implementation of the processors against the specification. Because of this, there are many instances of specifications being changed or amended after the related processor has been released to improve on accuracy.

Arm Architecture Reference Manual

The current version of the Arm Architecture Reference Manual for A-profile architecture[5], version DDI 0487I.a, stands at an ungodly 11,952 pages. For many years, these specifications were created after the implementation had been built and served as a reference. Anyone wanting to use the specifications for large-scale verification or reasoning about the correctness of the specification had to manually read and transcribe the relevant parts of the specifications to whatever language they were using. This was both time consuming and error prone. In an effort to run formal verification on ARM Processors, a 5-year internal project spearheaded by Alastair Reid changed the existing process that generates the specification in order to produce a machine-readable,

executable specification[29]. The pseudocode originally used as an informal description of the processor's behaviour has been formalized into a new specification language called ASL, for Arm Specification Language. After recognizing the other potential uses of such a specification, this machine-readable specification was made (partially) publicly available under the term "Exploration Tools"[1], and is part of the Arm Architecture Reference Manual.

Machine-Readable Specification

With access to the Machine-Readable Specification, the first step was to get a better understanding of the specification and how to extract specific information from it. Since there is no overarching documentation for the specification, this was done by reading parts of the specification and trying to extract information from it.

The Exploration Tools consist of three parts;

- Arm Architecture System Registers
- Arm A64 Instruction Set Architecture
- Arm A32/T32 Instruction Set Architecture

All three of these specifications are available in the form of a PDF document, a set of machine-readable XML files, and a browsable HTML version. All versions hold on to the same information, but the HTML version is much easier to navigate manually and the XML version is much easier to work with programmatically. For the sake of this project, we extract information from the XML version of the specification and only use the HTML version for manual verification.

SysReg

The "Arm Architecture System Registers" specification, referred to as "SysReg" from now on, contains XML files that describe the behaviour of the system registers in the architecture. Each register is described in a separate XML file, containing information like register fields, reset values, and access permissions. A lot of this information is still written in natural language. While this is useful for a reference document, this is very difficult to use in an automated way.

Information regarding registers is implicitly used in ASL snippets. An ASL snippet can directly reference a register by name and use its fields. For example, the file for AArch32 register SCTLR (System Control Register), defines a field called TE. This is the T32 Exception Enable bit that determines whether an exception should run in the A32 or T32 state. This field is directly referenced as SCTLR.TE, and there are no additional tags around it to help narrow down where this is defined. Since the length of a field might be crucial

to the interpretation of an ASL snippet, the knowledge of the existence of every single register and their fields is required to correctly interpret the ASL snippets.

Register files also do not exclusively define a single register. Some registers have additional Secure and Non-Secure versions, denoted with a _S or _NS postfix. These mapped registers are usually, but not always, defined as a register mapping. For example, the PRRR register (Primary Region Remap Register) defines PRRR_S and PRRR_NS, while SCTLR does not define SCTLR_S and SCTLR_NS, while they are used in ASL snippets.

These cases where ASL presumably references a register and/or a register field that is not described in the specification do not evoke confidence in the correctness of the registers that are described. While any unknown register and/or field can be abstracted away as a globally available variable, this is not a good solution for a formal verification tool. The tool should be able to reason about the correctness of the specification, and not just blindly accept it.

Index files

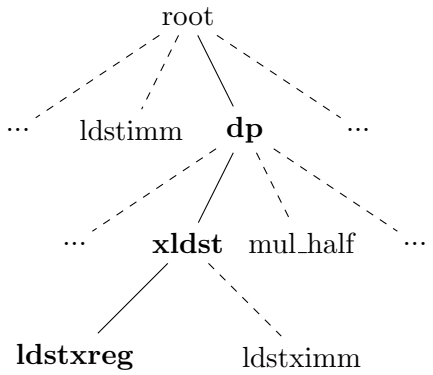
The most important parts of the "Arm A64 Instruction Set Architecture" and "Arm A32/T32 Instruction Set Architecture" specifications, referred to as "A64" and "AArch32" from now on, are their index files. These special files describe how to map the bits of an instruction to the actual instruction it encodes. The index files contain a tree of nodes called the hierarchy, which represent the way instructions are decoded. Non-leaf nodes represent increasingly more specific groups of instructions, while leaf nodes represent instruction classes, from now on called iclasses. The T32, A32, and A64 instruction sets consist of 1327, 1201, and 4321 nodes, respectively.

The hierarchy tree contains many structures called regdiagrams that specify bitpatterns with named ranges. These patterns are mutually exclusive between nodes on the same level and can be used to decode an instruction. The regdiagram is described in more detail in Section 4.1. By matching the bitpattern of an instruction with the regdiagrams of the hierarchy, a path from the root of the tree to an iclass can be found, as shown in Figure 3.1.

iclass sections

After the hierarchy, every iclass has a section that contains information about the instruction class. An instruction table further decodes the instruction to a specific instance within the class, identified by its unique encoding name, or encname, and the name of the file that describes the instruction, called its iform file. The instruction table is described in more detail in Section 4.1.

Figure 3.1: Example path in the a32 hierarchy tree



iform files

The specification contains XML files, called iform files, for each instruction. The AArch64 profile contains 1607 iform files, and the AArch32 profile contains 566 iform files. These files contain a lot of information, from assembly syntax to operational notes, but we are mostly interested in the ps_section tags. These ps_sections express the semantics of the operation using ASL, with XML tags throughout that links specific objects like functions to where they are defined. This is mostly used to directly link to them in the HTML version and sadly these tags are not always available, so this is not something that can be relied on. Other parts of the iform files are used to improve the overall quality of the projects, like the assembly syntax. This contains templates for instruction mnemonics and operands, which is added to the decoders to give us a disassembler for free.

Every unique instruction encoding corresponds to multiple ps_sections; a Decode section, an Execute section, and occasionally a Postdecode section. These sections all have name attributes that suggest the existence of some file hierarchy that is not included in the specification itself. While extracting these snippets to their own file, we attempt to reconstruct this hierarchy, though it became obvious later that this hierarchy is neither consistent, nor needed for the tool to function.

ASL

Besides the index files, there are also special shared_pseudocode.xml files, which contain an enormous amount of pseudocode snippets which describe functions and other objects that are used by the instructions. The AArch64 profile for example contains 1697 of these sections.

ASL grew organically by expanding on the pseudocode from earlier versions of the specification. This was done in order to improve the transition from the original specification, since moving over to a completely new speci-

fication gets rid of existing trust in the correctness of the specification. The design of ASL was therefore also based on non-functional criteria such as the space it occupies when printed.

All ASL is written for the context of the entire specification, and many snippets have additional context, such as instruction snippets depending on variables defined as fields in a regdiagram. In order to make sense of a function, it might be required to go through large parts of the specification. This is made significantly more difficult by the many different types of ambiguity in the language that can only be resolved by knowledge from other parts of the specification, such as the usage of "accessor" functions, which are functions written like array accesses. Other more low-level ambiguities require the use of a parsing with unbounded lookahead. ASL also uses bitvectors of arbitrary length, and because it is strongly-typed, this means that type interference has to use polynomials to represent the bitvector lengths.

At the time of writing, there is no formal language definition of the ASL pseudocode. The PDF version of the specification contains an appendix which provides a minimal definition useful for understanding the pseudocode, but it glosses over many important details. Working with ASL was therefore a process of trial and error, and not having a formal definition for the very language that is used to describe the specification add another point of possible failure to the verification process, since now you also have to reason about the correctness of the parser.

3.2 Emulation

Emulation is the process of creating a virtual version of a system or machine on another system or machine. This virtual version, or emulator, is designed to behave like the original system, allowing software designed for the original system to run on the emulator. Emulation is used in many fields, including gaming, software development, and system testing. After going over the basics of emulation, we will discuss the general design we used for our emulator, and how we used the Arm Machine-Readable Specification to implement it.

High-Level Emulation

High-Level emulation aims to emulate the functionality of a program running on specific hardware. The specifics of how the hardware implements certain instructions are not important, and this allows the emulator to make full use of the capabilities of the host hardware. This is the most common form of emulation for professional emulators that try to fairly faithfully emulate a system, but primarily focus on the experience of running the system. High-level emulators have been developed for almost all major consoles, usually using commonly-used disassembly frameworks or using their own set of tools.

Low-Level Emulation

Low-Level emulation aims to emulate the system itself, focusing on how individual instructions are implemented. This is the most common form of emulation for small hobbyist emulators that target smaller, simpler systems. Most of the work comes from transcribing the semantics of the instructions into whatever language the emulator is written in. When these hobby projects want to run as much of the original programs as possible, the systems have to get more complex however, as there is a need for the emulation to be bug-compatible. This means that the edge-cases and normally incorrect behaviour are also preserved, as programs are occasionally reliant on specific incorrect behaviour. Many older systems have published hardware errata that give insight into how things are implemented in hardware. This allows for the tracking down of the causes of unexpected behaviour. Some programs unknowingly make use of this behaviour, so a less faithful emulator would behave differently. Because of the precision potentially required to preserve the original behaviour of the programs, many emulators aim to be cycle-accurate, or even subcycle-accurate.

Emulation of the Arm Architecture

Emulation of the Arm architecture involves creating a virtual environment that replicates the behaviour of an Arm-based system. Given that the goal is not to emulate a specific system, but rather the abstract Arm architecture itself, it is not possible to speak of higher-level emulation. There are hand-written low-level emulators for Arm, but these are usually written for specific systems, usually the ones available. The Arm Machine-Readable Specification gives us the opportunity to go one step lower and emulate what is effectively an abstraction over the microarchitecture. This allows us to emulate the Arm architecture to an incredible level of precision, as the specification is designed to be used by hardware designers to implement the Arm architecture in hardware. This means that the specification is very precise, and includes many details that are normally not relevant to software developers, that might be overlooked when talking about the semantics of instructions.

You could also say that the emulator is running an abstract machine with its own instruction set, the ASL instructions, and an execution loop that uses the specification as a description of how to translate Arm instructions into instructions for this abstract machine. In this way, the specification has shifted the manual work of writing an emulator from the Arm semantics to the ASL semantics. This is much less work, as the ASL semantics are much smaller, but the lack of documentation still presents a hurdle.

Chapter 4

Implementation

With an endgoal in mind, but no clear path to get there, development was largely driven by exploration and encountered limitations. In hindsight, many of the design decisions made were not optimal, but were made to keep the project moving forward.

4.1 Working with the ARM Machine-Readable Specification

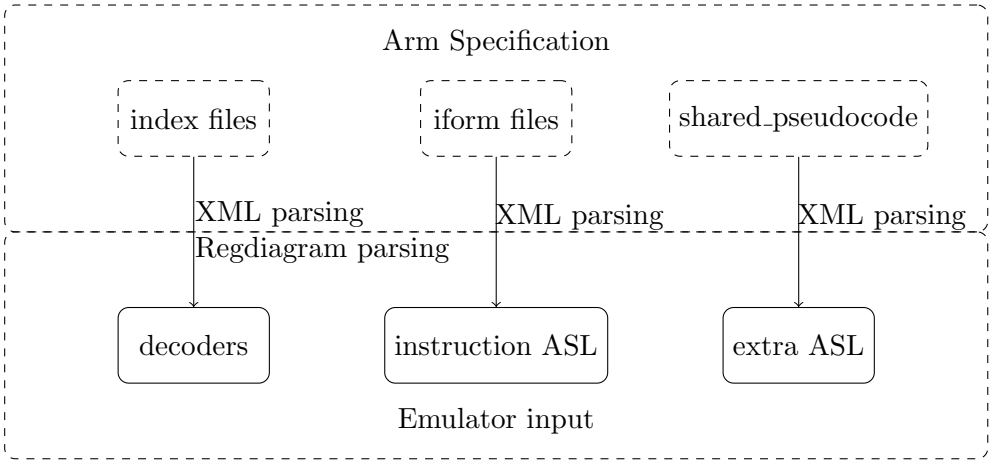
There is no specification for the specification, so the layout and structure had to be experimented with to find out how it works. As mentioned in Section 3.1, the specification can be viewed as consisting of two parts; the instruction decoding by going through the encodingindex's hierarchy tree, and the instruction specific iformfiles. In order to simplify the process of working with the specification, the files are preprocessed into forms that more easily allow for integration into the emulator, as shown in Figure 4.1.

Because walking the hierarchy tree in xml for every instruction encountered is a slow process, the regdiagrams are extracted, and a decoder is generated for each instruction set that has cascading functions corresponding to nodes in the tree. These functions end up with generating an instruction state which has variables for the different fields in the regdiagram, initialized based on the specific instruction. This state is directly used by the ASL interpreter, as identifiers can directly refer to these fields by name.

Regdiagrams

All throughout the specification, regdiagrams are used to describe bitpatterns. These regdiagrams are lists of boxes that describe ranges in those bitpatterns. For example, the decoding regdiagram for the instruction group mortlach_32bit_prod (Scalable Matrix Extension 32-bit Outer Product) includes the boxes shown in Listing 4.1.

Figure 4.1: Extracting parts of the specification



Listing 4.1: Two regdiagram boxes

```
1 <box hibit="29" width="1" name="op0" usename="1">
2   <c colspan="1" />
3 </box>
4 [...]
5 <box hibit="23" width="2">
6   <c colspan="2">10</c>
7 </box>
```

The first box defines a field called "op0", starting at bit 29 counting from the right, with a width of 1. A single pattern runs over the whole range of 31-0, or 15-0 for the shorter Thumb instructions, so bit 29 is the third bit from the left. The box does not contain a value for it, so it matches with anything. The second box does not name its field, but it does contain the value "10". In order to condense this information, any further patterns will be written as a bitstring with "x" for unknown named bits and "-" for unknown unnamed bits. The regdiagram for the two boxes above would be written as;

-	-	x	-	-	-	-	-	1	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

If we look at the A32 instruction "ands x1, x2, x3, lsl #4", encoded as

1	1	1	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1

,
we can follow the hierarchy tree to see how the instruction is decoded. The root node of the hierarchy has ten child nodes that describe different groups or classes. Our instruction matches with a node with the groupname "dpreg" with decode regdiagram

x	-	-	x	1	0	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

.
The "dpreg" group has twelve child nodes, of which our instruction matches with the node with the iclass "log_shift" with decode regdiagram

-	x	-	0	-	-	-	0	x	x	x	-	-	-	-	-	x	-	-	-	-	-	-	-	-	-	-	-	-	-

.

This is a leaf node, so it has no further children.

Instruction Table

The instruction table is a table that maps the values of the fields in the regdiagram to the instruction that matches with those values. The regdiagram for the "log_shift" iclass defines 8 fields, but 4 are relevant for the instruction table;
sf: bit 31
op: bits 30-29
N: bit 21
imm6: bits 15-10

The instruction table, partially shown in Listint 4.2, has a header that defines the order of the fields, and rows that correspond to specific instances of the instruction class. Matching on these fields gives us an encoding name, iformid, and iformfile.

Listing 4.2: log_shift's iclass_sect

```
<instructiontable iclass="log_shift" cols="6">
  <thead class="instructiontable">
    [...]
    <tr id="heading2">
      <th class="bitfields">sf</th>
      <th class="bitfields">opc</th>
      <th class="bitfields">N</th>
      <th class="bitfields">imm6</th>
    </tr>
  </thead>
  <tbody>
    [...]
    <tr class="instructiontable" encname="ANDS_64_log_shift"
        ↪ iformfile="ands_log_shift.xml" label="64-bit"
        ↪ oneofthismnem="2" first="t" last="t">
      <td bitwidth="1" class="bitfield">1</td>
      <td bitwidth="2" class="bitfield">11</td>
      <td bitwidth="1" class="bitfield">0</td>
      <td bitwidth="6" class="bitfield"></td>
      <td class="iformname" iformid="ANDS_log_shift">ANDS (
          ↪ shifted register)</td>
      <td class="enctags">64-bit</td>
    </tr>
    [...]
  </tbody>
```

```
23 </instructiontable>
```

There are instances in which the attributes do not make sense and the overall design starts to break down. For example, these bitfields can consist of multiple named fields together with colons between them. The actual meaning of this is not explained within the specification, and no matter what you choose, it breaks something. In some cases, it seems to mean the entire interval between the fields, which corresponds to the bitwidth given to the field. The contents of these fields however mostly follow the idea of simply concatenating the fields, leading to cases where the bitwidth and the actual length differ.

Listing 4.3: log_shift's iclass_sect

```
1 <td bitwidth="7" class="bitfield">00000</td>
```

It is rather problematic that something as seemingly simple as expressing a bitpattern is convoluted and inconsistent. This is a problem that is not limited to the instruction table, but is present throughout the specification.

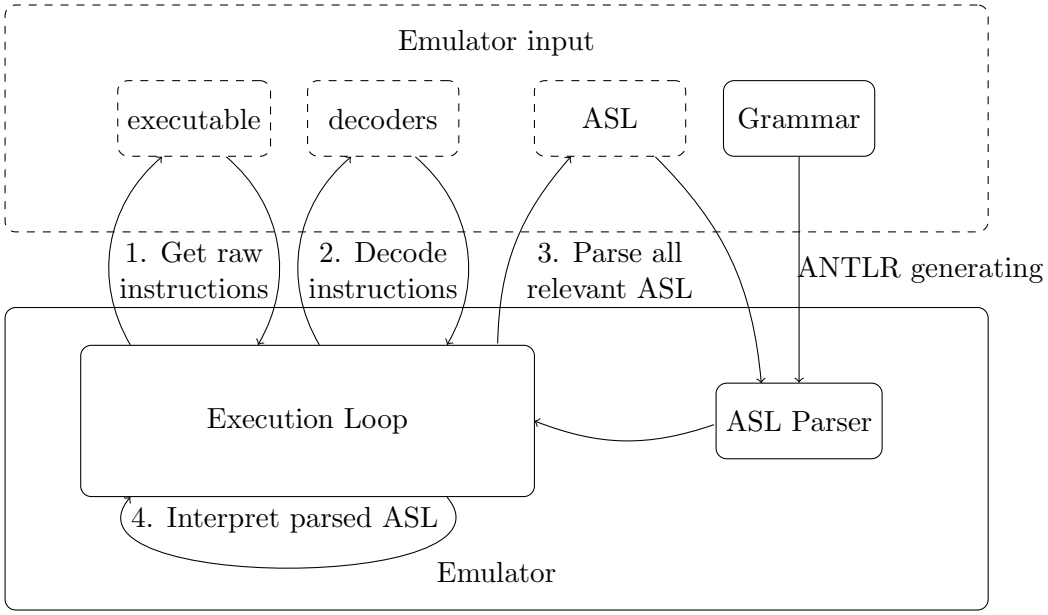
Parsing ASL

There is no formal grammar for ASL, so the development of a parser involved a lot of trial and error. I chose to use the ANTLR[4] parser generator because of its great python runtime. The ambiguities of ASL and the inherent problem of generating useful error messages made it difficult to write a complete, correct grammar in one go, so it was developed in parallel with the rest of the prototype; adding new rules as they were needed by the specific instructions or functions encountered. There are a few weird edge cases in the specification that heavily influenced the grammar. This eventual grammar used to generate the parser is overly permissive to deal with these edge cases and generally assumes that the input is valid. The entire specification was used as a test-suite for the parser, so while the generated parser can completely parse the existing specifications, this does not guarantee that it can parse all valid ASL code. In most cases, it would be preferable to change the edge cases in the specification instead of changing the grammar to accommodate them, but the specification is meant to be input to the project. The unbounded lookahead required for parsing and the large amount of ambiguities in the specification make it a relatively slow process, but preferably all parsing would be done separately from the execution of the model, so this is not a major concern.

4.2 Design of the Emulator

Under the original goal of trying to emulate an Arm executable on a different architecture, the emulator would have to describe the abstract behaviour of each individual instruction. Given the size of the Arm instruction sets, this

Figure 4.2: Emulator design



would be a very large task. The specification provides the operational semantics of the individual instructions, so using this specification shifts the level of abstraction on which emulation is done. Instead of emulating the individual instructions, the emulator can emulate the operational semantics, available in ASL. The biggest advantage of this setup is that changes to the specifation are automatically reflected in the emulator.

Skipping over the setup of an executable for a bit, Figure 4.2 shows the design of the emulator. The emulator takes an executable, and the preprocessed specification and runs in a loop;

1. Fetch the next instruction from the executable.
2. Decode the instruction to determine the operational semantics.
3. Parse the relevant ASL snippets.
4. Interpret the parsed ASL.
5. Execute meta operations.

These meta operations are operations separate from the operational semantics of the instructions, but are still necessary for the emulator to function. Examples of these are incrementing the program counter after an instruction is executed, and handling interrupts. In actuality, the emulator does not separate the parsing and execution of the ASL into separate steps, since the interpretation of the ASL leads to new ASL snippets that need to be parsed

and interpreted. If performance was a concern, it would be better to fully process the ASL snippets and partially evaluate instruction semantics, but this is not the case for this emulator.

This is a very high level description of the execution loop, but it is enough to start implementing the emulator. The first step is to fetch the next instruction from the executable. We are limiting ourselves to ELF executables, as this is the most common format for Arm executables.

Reading ELF files

ELF files are normally executed by the operating system, which loads the executable into memory, sets up the stack and registers, and then jumps to the entry point of the executable. Often, this entry point is also in a special section that invokes a dynamic linker, which loads all the shared libraries that the executable depends on. The emulator does not have an operating system it can depend on to do all this, so it has to do it itself.

The first step is to read the initial part of the executable, the elf header. This header contains a lot of information about the executable, but we are mostly interested in e_entry, the entry point, and e_phoff, the offset of the program header table. The program header table holds headers for each individual program header, which describe segments of the executable and where in memory they should be loaded. Reading these program headers, we can set up the memory for the executable, and we now know where to start executing the executable.

Emulating memory

This introduces the first problem; the emulator needs to know how to deal with memory. The obvious solution is to use a large continuous block of memory for this, but this can be expensive and glosses over important memory behaviour like memory mapped I/O, in which the same address space is used for both memory and I/O devices. The specification is rather unclear about how memory is supposed to be handled, choosing to declare but not define the functions related to actually interacting with the memory. The way we addressed this is by using a memory map, which is a mapping from address ranges to their concrete values.

Executing ASL

After the context for the executable has been set up and it is possible to speak of a current instruction, the emulator can start executing. After fetching the instruction, decoding it, determining what ASL snippet expresses the operational semantics, and parsing said ASL, we come to the point where we have to execute the ASL. Since large parts of the code only depend on static information, such as what features are enabled, it would be possible to precompile

the ASL to a more efficient format or do partial evaluation on it. This was originally planned, but due to limitations in the specification, this was not expanded upon.

The ASL is executed by an interpreter that directly uses the parsed abstract syntax tree and evaluates it via in-order traversal. This is definitively not the most efficient way to execute the ASL and heavily ties the emulator to the parser, but it is a simple way to execute the ASL. The interpreter is also responsible for handling operations that are not part of the specification, such as the aforementioned memory interactions.

Dealing with unknowns

The specification is filled with ASL using partial values and unknowns. In order to work with these, the most fundamental datatype in the specification, the 'bits', is expressed as a tritstring, a string of 1s, 0s, and xs. These xs are both used as wildcards during matching, and as unknown values, which means that normal equality is not symmetric. These unknows are also propagated through binary operations.

Meta operations

While the specification is very detailed when it comes to the operational semantics of the instructions, it completely ignores the overarching process. Something as simple as incrementing the program counter after an instruction is executed that does not change the program counter is not part of the specification. This means that we had to use the standard pdf specifications to determine what to do about more complicated meta operations, such as handling interrupts and timers.

4.3 Missing Specifications

While the specification covers important parts of the architecture, including a detailed specification of every single instruction, there are still many parts to running an executable that are not covered by the specification.

Implementation Defined behaviour

Many important parts of the overall specification are underdefined; most noticeably the main execution loop mentioned earlier, speculative execution and multi-processing. The most important limiting factors of the specification are the behaviours that differs per implementation, such as memory access and caching. This leads to a lot of important parts of the specification simply stating it is IMPLEMENTATION_DEFINED, and often not specifying what the implementation should do.

Listing 4.4: ASL function only declared

```
1 // Returns the value read from memory, and a status.
2 // Returned value is UNKNOWN if an External abort occurred while
    ↪  reading the
3 // memory.
4 // Otherwise the PhysMemRetStatus statuscode is Fault_None.
5 (PhysMemRetStatus, bits(8*size)) PhysMemRead(AddressDescriptor
    ↪ desc, integer size, AccessDescriptor accdesc);
```

If a function is not defined, only declared, then any model that wants to use the specification has to provide a definition.

Listing 4.5: Implementation Defined behaviour

```
1 // HaveRME()
2 // =========
3 // Returns TRUE if the Realm Management Extension is implemented
    ↪ , and FALSE
4 // otherwise.
5
6 boolean HaveRME()
7 return boolean IMPLEMENTATION_DEFINED "Has RME extension";
```

Because of the modular nature of Arm processors, large parts of the specification hit functions that check whether a specific extension is supported. While building a model on top of the specification, these IMPLEMENTATION_DEFINED values can be given a value, but some of these implementation defined values require a very deep understanding of the entire architecture. Some examples of such IMPLEMENTATION_DEFINED values are:

- "JOSCR UNDEFINED at EL0"
- "ID_AA64ISAR2_EL1 trapped by HCR_EL2.TID3"
- "EL3 trap priority when SDD == '1'"
- "Has increased Reciprocal Estimate and Square Root Estimate precision support"

Together with the overall specification that outsources implementation defined behaviour, it would be very helpful if there were also actual implementations so you can reason about the behaviour of the specification without having to figure out these implementation defined behaviour. It is of course near impossible to provide the implementation behaviour of every single possible combination of extensions, but it would be helpful if there were at least some implementations of the most common combinations of extensions, such as the most common processor configurations.

The way the emulator handles these implementation defined values and functions is by using a separate set of files that contain handwritten ASL snippets that define some of these. These snippets were all created in the order in which they were needed by benchmark programs ran on the emulator.

Limits of ASL

Listing 4.6: Obscured Execution

```
1 if !UsingAArch32() then
2 ExecuteA64(0xD5330501<31:0>); // A64 "MRS X1,DBGDTRRX_EL0"
3 ExecuteA64(0xB8004401<31:0>); // A64 "STR W1,[X0],#4"
4 X[1] = bits(64) UNKNOWN;
5 else
6 ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/); // T32 "
    ↪ MRS R1,DBGDTRRXint"
7 ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/); // T32 "
    ↪ STR R1,[R0],#4"
8 R[1] = bits(32) UNKNOWN;
```

Because of the limitations of ASL itself, some operations like invoking a specific instruction are unintentionally obfuscated, with the important information left in comments. In this case, ExecuteA64 and ExecuteT32 are also only declared and not defined. These limitations are not a problem for using the specification, but it does show a remarkable property of the specification; the instructions all exist in isolation, with some overlapping functionality split off into the shared_pseudocode snippets. Inputs to these instructions are left as fields in the encoding XML, and it is impossible to directly refer to other instructions.

Other specification scopes

A program is much more than just the instructions that are executed. There are many other parts of a program that are not covered by the specification, like its interactions with the operating system. For an emulator that wants to run a program, it needs to know how to load the program into memory, how to start the program, what system calls do, etc. These are all parts of the operating system specification, which is obviously not available in the same way as the architecture specification. In order to fully emulate a program, other specifications are needed, and it would obviously be very helpful if different machine-readable specifications were available and expressed in the same way.

Chapter 5

Evalutation

The goal of this project is to evaluate the feasibility of using the Arm Machine-Readable Specification for reasoning about the behaviour of Arm programs. To this end, we have constructed an emulator that could in theory run binaries compiled for Arm on a non-Arm architecture.

5.1 Emulation In Steps

Since there are many moving parts involved with emulating a program, we decided to evaluate the capabilities of the emulator in steps, starting with the simplest possible program, and gradually increasing the complexity of the programs. This allowed us to test the emulator in isolation, and to ensure that the emulator was able to execute the instructions correctly before moving on to more complex programs. While the stereotypical "Hello World" program is usually considered the simplest possible program, on a machine code level, it is actually quite complex, as it uses a runtime with a standard library and uses system calls to interact with the operating system. Instead, we first try to run individual instructions.

Prerequisites

Instructions make use of structures and functions defined in the shared pseudocode, and are intended to be run in the context of an initialized processor. In order to run any instruction, it is therefore required to load the shared pseudocode into the emulator, and initialize the processor.

The shared pseudocode spans 1697 individual ASL files, defining or declaring the structures shown in Figure 5.1. These files have to be parsed, interpreted, and potentially partially evaluated before any instruction can be executed.

The SysReg part of the specification spans 1627 individual xml files, defining 273 AArch32 registers, 548 AArch64 registers, and 535 shared registers.

Figure 5.1: Shared Pseudocode Structures

	AArcher32	**AArch64**	**Shared**
Function Definitions	192	517	596
Function Declarations	16	58	93
Constants	4	20	60
Enumerations	5	27	45
Accessors	16	56	10
Variables	2	33	15
Arrays	1	4	3
Types	0	13	28

These registers will also have to be loaded into the emulator before any instruction can be executed, as it would otherwise be impossible to know whether an identifier is a register or not.

Loading and parsing all these files takes well over a minute, and has to be done every time the emulator is started. This could be improved by preparsing and precompiling the files, and an earlier version of the emulator did this, but it was not reintroduced after a major refactor of the emulator due to the minor benefit it provided compared to the increased complexity of the emulator.

Registers

In order for a program to achieve anything, it needs to be able to interact with registers. The simplest possible program that allows us to test this is a program that sets a register, and then copies that value to another register. This not only tests the ability to set and read registers, but also all the surrounding functionality.

```
1       E3A00001 - mov r0, #1
2       E1A01000 - mov r1, r0
```

In order to execute these instructions, we need to set the conditional flags, since those are relied upon by all instructions. To observe the behaviour of the instructions, we display the values of the registers after every instruction is executed.

```
1  PC         : 0x0000000000000000
2  N Z C V
3  0 0 0 0
4
5  R[ 0]      : 00000000000000000000000000000001 = 00000001
6  R[ 1]      : 00000000000000000000000000000001 = 00000001
7  R[ 2]      : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
8  R[ 3]      : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
9  R[ 4]      : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
R[ 5]     : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
R[ 6]     : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
R[ 7]     : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
R[ 8]     : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
R[ 9]     : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
R[10]     : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
R[11] (fp): xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
R[12] (ip): xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
R[13] (sp): xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
R[14] (lr): xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Memory Access

If a program needs to hold on to more values than there are registers, it needs to be able to access memory. Normally, the operating system handles memory management, so programs have to request memory regions and are only allowed to access the memory regions they have been granted access to. In the emulator, this is glossed over, and the program is allowed to access any memory address. This is obviously not a realistic assumption, but it is good enough for testing the emulator. In order to test memory access, we need to be able to write to memory, and then read from that memory location. This can be done by writing to register r1, storing that register value into memory at address #400, and then loading that memory into register r2.

```
E3A00E19 - mov r0, #400
E3A0104D - mov r1, #77
E5801000 - str r1, [r0]
E5902000 - ldr r2, [r0]
```

Trying to run these instructions surfaces the next kind of problem. The ASL code uses the term SCTLR_NS, which is not defined anywhere in the specification. Context clues lead us to infer that this is a Non-Secure version of the group of registers like SCTLR_EL1, but this is yet another assumption someone has to make when trying to use the specification.

Cold Boot

In order to initialize the processor, the emulator has to run a cold boot sequence. This is defined in the specification as the function AArch32.TakeReset. This function checks security permissions, resets register values, sets up the PSTATE, and then jumps to the reset vector. This reset vector also has to be set by initializing the Reset Vector Base Address Register.

The entire boot sequence requires setting thirteen feature flags, uses five declared but undefined functions, needs two implementation defined values, and makes 2009 function calls. A total of 57 ASL files are used to execute this

function with the features that we have enabled. The vast majority of these functions calls are for checking exception levels and security permissions which depend on static information, so it would be possible to either precompile the ASL to a more efficient format or do partial evaluation on it. This was originally planned, but due to limitations in the specification, this was not expanded upon.

Graceful Termination

Up until this point, we have been disregarding the Program Counter and the entire execution loop around it, instead directly feeding the emulator what instructions to execute. In order to emulate a program, we need to be able to have the emulator decide what instruction to execute next. To construct this execution loop, we start at the end.

Program execution usually starts at a predefined entry point, and keeps going until something prevents further execution. In the context of a complete computer, the execution of a program is handled by the operating system, and when the program runs out of instructions to execute without gracefully telling the operating system it wants to stop, it will keep going and try to execute instructions from memory that it is not allowed to access. This will trigger a memory access violation, which will prompt the operating system to terminate the program. In the emulator, there is no operating system to handle this, so the execution will run into unknown memory, which will crash the emulator if it is not handled. In order to prevent this, we need to be able to detect when the program has finished executing. The two ways we can do this is by either triggering an exception, such as a breakpoint, or by making a system call to the operating system to terminate the program.

```
1       E1200070 - bkpt #0
```

Trying to trigger a breakpoint in the emulator surfaces the next problem. The specification defines the BKPT instruction as a call to AArch32.SoftwareBreakpoint(imm16). This function intends to go through the normal exception process, but it runs into a situation where it tries to access a value that has not been set, in a structure it defines itself. It is of course possible to alter the AArch32.SoftwareBreakpoint function to set this value, but if we have to fix every single problem we run into by creating our own implementation, it defeats the purpose of having this specification. The whole point of this instruction was to add a way to inspect the state of the emulator to observe the behaviour of the program. Running into an exception before we even have a program to inspect kills any confidence we might have had in the possibility of fully emulating a program. At the end of the function cascade, AArch32.EnterModeInDebugState sets specific flags in the most important registers, including the ERR flag on the External Debug Status and Control Register EDSCR, to indicate that normal control flow is

aborted. The status code is set to indicate that the exception was caused by a breakpoint, so we can use this to detect the breakpoint and inspect the state of the emulator, followed by restoring the state of the processor and resuming execution by means of a special ExitDebugState function.

To eventually support full programs, we also look at the system call instruction. Since operating system interactions are obviously context dependent, we cannot directly ask it to stop the program for us. We cannot do this in the emulator without making some assumptions about the operating system, so we chose to assume a Linux environment, in which system calls are made by setting the value of the r7 register to the system call number, and then triggering an exception with the SuperVisor Call instruction.

```
1       E3A00000 - mov r0, #0
2       E3A07001 - mov r7, #1
3       EF000000 - svc #0
```

Register r0 is an argument supplied to the system call, which in this case is the return code of the program, where 0 indicates success.

The execution of the SVC instruction reaches the same part of the code responsible for handling breakpoints, where execution is moved to an address that normally holds a handler function defined by the operating system. Since we are not running in an operating system, we would have to define this handler function ourselves. We instead chose to detect that control flow was transferred to the handler, and use the r0 and r7 registers to determine what system call was made. In this case, we can detect that the system call was exit, and terminate the program.

Function Calls

Function calls in Arm are implemented as branches "with link", which means that the address of the next instruction to return to after the function is saved into general register r14, also known as the link register lr. For nested function calls, the link register is usually pushed onto the stack. This behaviour can be tested by running a simple function call;

```
1       EB000000 - bl #8
```

This will branch and link one instruction forward, meaning the address after this instruction is saved into the link register. The function can then return using the bx instruction, which branches to the address in a register.

```
1       E12FFF1E - bx lr
```

Running in memory

In order to handle actual branching in the assembly code, we need to use the actual Program Counter. This is done by implementing a small loop that

fetches the instruction at the address pointed to by the Program Counter, and executes this. A random comment on the Mem_with_type function suggests that instruction fetches are done by directly calling AArch32.MemSingle. Since the Program Counter is only incremented if the instruction did not cause a branch, we add a globally available state variable that indicates whether a branch was triggered.

Because the instruction decoding is done outside of ASL, the execution loop is not represented by an ASL snippet, but rather as python function that calls upon the emulator to execute specific statements.

Since control flow is now handled within memory, we can no longer rely on feeding the emulator instructions, and instead write the instructions to memory, point the Program Counter to the first instruction, and then start the execution loop. For smaller programs, we dump the emulator state after every instruction, and for larger programs, we only dump the state when the program terminates.

Loading ELF files

The final step in running an actual executable is to load a compiled program into memory and set up its context. We go over the segments of the ELF file, and write them to the addresses specified in the headers. We then set the Program Counter to the entry point in the main header. It would be nice to be able to just start the execution loop at this point, but the program still has some context that needs to be recreated. If we pick a statically compiled C program with just a main function, there is still a lot of code surrounding it that is relevant. Instead of the main function being called, the program first sets up the C environment and the standard library, which involves setting up static data and a lot of stack manipulations. Likewise, when the main function returns, the program does not immediately terminate, but instead cleans up the C environment and the standard library first.

When we try to run a C program on the emulator, it runs into several instances where registers and memory addresses are assumed to have specific values. Setting these values allows us to run the program for a bit, but at some point, the program tries to access memory that is supposed to be initiated by the loader, and without building an entire loader, it is almost impossible to figure out what values the program is expecting.

It is possible to try to run programs by skipping the C runtime initialization by starting at the address pointed to by the "main" symbol instead of starting at the original entry point, but any non-trivial program relies on this runtime, so this severely limits the programs that can be run.

5.2 Level of Detail

The aforementioned underspecification of certain parts of the Arm architecture presents a problem for the emulation of Arm binaries. There are large parts of the specification that are left as implementation defined, which means that the emulator has to make assumptions about the behaviour of the processor. This is especially problematic for the emulation of binaries, as the compiler can make use of these implementation defined behaviours to optimise the code. This means that the compiler can generate code that is correct under the assumption that the executable will be run on a specific processor, which might behave wildly different from the processor that the emulator is pretending to be.

Memory and Cache

Going into this project, the first potential problem that came to mind was how the specification would express the caching behaviour of the processor. Over time, the performance of memory has become relatively worse compared to the processor, and so most modern processors have multiple levels of cache, including caches that are shared between clusters. Because of this setup, we expected to find some form of specification regarding the inner workings of the cache controller that controls all aspects of the caches in tandem with the core. Sadly all aspects of memory handling and multiprocessing, such as the cache, are left as implementation defined. This is not necessarily a problem for the emulator, as it can be abstracted over, but it does mean that all the interesting details of memory behaviour is lost.

5.3 Emulation Performance

There are many criteria that can be used to evaluate the performance of an emulator. Traditionally, the most important metrics for an emulator are its speed and its accuracy. However, in the context of this project, we are not as interested in the speed of the emulator, and more interested in the accuracy and functionality of the emulator.

The accuracy of the emulator refers to how faithfully it reproduces the behavior of the actual hardware. In the case of an Arm emulator, it involves ensuring that the instructions are executed correctly and produce the expected results.

For this purpose, Arm has their own private test suite, which is used to test implementations against their specification. Since this test suite is not public, we have not been able to use it to test our emulator, instead we have had to rely on creating our own test suite. The initial goal of the test suite was to gather a set of binaries that we could use to benchmark the emulator, however, since we were unable to get the emulator to run any binaries due to missing

context, we had to settle for manually running the emulator interactively and using handwritten assembly-level programs to test the emulator. The test suite is not comprehensive at all, since it was created while still under the assumption that the emulator would be able to run binaries.

Instruction Times

On an actual processor, different instructions have their execution times expressed in how many cycles it takes. This is mostly influenced by whether or not the instruction uses the barrel shifter or the program counter.

For the emulator, the execution time of a single instruction depends on the amount of specification required to express its semantics. The biggest influence on this is the security involved with the instruction; instructions that touch upon security sensitive parts of the system, such as memory management, have checks in place to see whether the current execution level of the processor is allowed to interact with that part of the system.

Encoding	**Instruction**	**Files**	**Time (ms)**
E3A00001	mov r0, #1	19	2.4
E1A01000	mov r1, r0	20	2.4
E5801000	str r1, [r0]	82	10.1
E5901000	ldr r1, [r0]	78	12.4
E0811280	add r1, r1, r0, lsl #3	28	4.1
E0010091	mul r1, r1, r0	20	4.5
E1510000	cmp r1, r0	25	4.0
EA000017	b #100	19	0.8

Table 5.1: Instruction execution times

Table 5.1 shows the execution statistics of the basic set of instructions. The emulator caches instruction decoding, so these times only reflect the actual execution of the instruction semantics. The times are the average of 1000 executions of the instruction, and the number of files is the number of ASL files interacted with during the emulation of the instruction. The absolute times are not very important, since that heavily depends on the hardware the emulator is running on, but the relative times are interesting. The memory instructions are by far the slowest, since they require a lot of checks to ensure that the memory access is allowed.

Given the successful emulation of register, memory, alu, and branching instructions, a lot of different types of programs can be emulated. One of the biggest limiting factors is the lack of support for dynamically linked library functions, since even something as simple as a modulo operation requires a call to a library function.

Chapter 6

Discussion

While working with the Arm Machine-Readable Specification, we kept running into issues related to two underspecified aspects; implementation defined behaviour; and behaviour that escapes the specification, such as operating system interactions. These issues are not necessarily problems with the specification itself, as it is not meant to be a complete specification of the Arm ISA, but it does mean that the specification is not suitable for modelling entire programs.

6.1 Requirements for a Specification

A specification is as good as its use cases. As such, we want a specification to be as complete as possible, and entirely correct.

The ASL specification covers large parts of the Arm ISA, and can be used to reason about the behaviour of sequences of simple instructions. For more complex operations however, such as memory caching and concurrency, the specification is severely lacking in detail. It instead leaves these parts as implementation defined, outsourcing the specification of these parts to the original Arm Reference Manual, and the implementation of these parts to the user of the specification. While this means that the specification can be used for comparing the behaviour of different implementations and verifying that these implementations follow the specification, it can only do this for the parts of the specification that are actually specified. Research into verifying the behaviour of Arm programs focusses on these more complex, underspecified parts of the specification, exactly because they are more complex.

While the specification is not complete, it has a very high level of detail, and is created by the authority on the Arm architecture, Arm themselves. This gives the specification a high level of trustworthiness. Of course, just like with any large enough project, the specification is not without its flaws. So far, Arm has released several versions of the specification, each accompanied by a list of changes that fixed old bugs, and a list of known issues that are to be fixed

in a future release. One of the things bringing down the trustworthiness of the specification is its obscurity. While the specification is publicly available, it is not very well known, and there is not a lot of documentation available on how to use it. This means that there is a high barrier to entry for anyone who wants to use the specification, and it is not very likely that the specification will be used by many others outside of Arm. Usage of the specification can bring to light issues, so the lack of usage means that there is a higher chance that issues will go unnoticed.

The biggest gripe we have with the specification is that there is this idea for an objective, correct interpretation of the specification, yet all executable code is written in a domain specific language that is not described within the specification, while they could have simply embedded the actual parsed abstract syntax tree. This would not only make more sense in the context of the specification, but it would also make it a lot easier to work with, as now any user of the specification has to write their own parser for the ASL language, and deal with the inconsistencies and ambiguities of the language that would be cleared up by directly embedding the AST.

6.2 Implementation Specific Specifications

It is obvious that a specification for the Arm architecture cannot go into detail about the implementation of every single piece of hardware that implements the Arm architecture. This is why the specification is written in a way that allows for abstracting over the parts that differ between implementations. It does limit the specification however that there is neither a simplified abstract implementation, or at least some concrete implementations available, for example for the most common Arm processors. There is also no easy way to construct a possible implementation from the specification, as the specification is not written in a way that lends itself to implementation. You would need to provide an implementation for each declared but undefined function in the entire specification, which requires domain knowledge about nearly every extension and feature of the Arm architecture, and requires you to express those features in a way that is compatible with the rest of the specification. Since this is a monumental task, it is not surprising that there are no implementations available created by people outside of Arm.

6.3 OS Specifications

While having an instruction set specification goes a long way towards automated reasoning about the behaviour of a program, too much of a program's behaviour depends on the operating system. Operating systems used to be complex but compact pieces of software, but as they have grown in complexity and size, they have become almost as complex as the hardware they run on.

Added to this is the fact that operating systems are not completely standardised, and that many operating systems are closed-source, and it becomes clear that it is not feasible for a small team to create a specification for an operating system that is as complete as the Arm Machine-Readable Specification. While it is possible to create a small specification for specific parts of the operating system, like system calls, we would prefer to have a specification that covers the more complex parts of the operating system too, like the scheduler and memory management.

Chapter 7

Conclusion

The goal of this project was to evaluate the Arm Machine-Readable Specification as an authoritative source of information about the Arm architecture, and for its possible application in the fields of binary analysis and binary translation. For this purpose, we have constructed an emulator that uses the specification as its only source of instruction semantics. We have found that the specification is a very useful resource for understanding the inner workings of the execution of individual instructions, and that it is a good starting point for creating a binary analysis tool. However, we have also found that the specification is not complete enough to be used as a source of truth, as it is missing crucial information about overarching concepts and implementation specific details.

7.1 Summary of the limitations of the ARM Machine-Readable Specification

The Arm Machine-Readable Specification has a couple of good things going for it, but it is being held back by a couple of big limitations. Not just the gaps in the specification, but also the overall lack of attention to the needs of the user of the specification and the level of polish in general.

During our time working with the specification, we have encountered issues including but not limited to:

- References to a layout of the specification that is much better organised
- References to non-existent sections
- Inconsistent usage of XML attributes, such as the attributes related to the instruction hierarchy
- Contradictory information, such as length attributes that do not match the length of the data

- Important details left in comments

Besides the issues within the specification, the biggest stumbling block for working with the specification is the lack of documentation. Figuring out the layout of the specification, and how to use it, took a lot of time and effort spent on trial and error experimentation.

The limited scope of the specification, namely only covering the semantics shared by all implementations, yet barely providing any information about the implementation-specific semantics, means that the use-cases for the specification are rather limited.

7.2 Future Work

The Arm Machine-Readable Specification is a step in the right direction for expressing the operational semantics of a program, but there are a lot of problems with it that are not easily solved without significant resources. REMS[32], the Rigorous Engineering of Mainstream Systems group, aims to develop a mathematically rigourous approach to working with more robust and secure systems. They have worked on many projects regarding models and semantics on not just instruction sets, but also memory behaviour, concurrency, ELF file linking, and more. Their ISA specification language, Sail, is a superset of ASL, and their Arm instruction set model is generated from a closed-source, more complete version of the Arm Machine-Readable Specification. This is made possible by the extensive amount of funding and industry contact available to the group.

With Arm mostly neglecting their own Machine-Readable Specification, we would not be surprised if the Sail model would be adopted to become the official specification, just like has happened to their RISC-V model. In effect, Sail is what this project would become given enough time and resources. Most of the possible extensions or other avenues of future work we have considered are already being worked on by the REMS group; including linksem[21], formal model for ELF linking, and Cerberus[24], an executable formal model for a subset of C.

If there was a need to pick a single, specific future direction for the project, it would be to pick a specific implementation and create an addition to the specification that provides all the ASL required to never run into unknown, implementation defined behaviour. Could be done by developing a system that runs instructions on actual hardware and on the emulator, both to validate the correctness of the ASL, and as a way to figure out the implementation specific details. This would be a very large undertaking, and would require a lot of time and resources, but it would be a very valuable addition to the specification.

Bibliography

[1] URL: https://developer.arm.com/downloads/-/exploration-tools.

[2] Kapil Anand et al. "A Compiler-Level Intermediate Representation Based Binary Analysis and Rewriting System". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: Association for Computing Machinery, 2013, pp. 295–308. ISBN: 9781450319942. DOI: 10.1145/2465351.2465380. URL: https://doi.org/10.1145/2465351.2465380.

[3] Dennis Andriesse et al. "An In-Depth Analysis of Disassembly on Full-Scale X86/X64 Binaries". In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC'16. Austin, TX, USA: USENIX Association, 2016, pp. 583–600. ISBN: 9781931971324.

[4] *ANTLR*. URL: https://www.antlr.org/.

[5] *Arm Architecture Reference Manual for A-profile architecture*. URL: https://developer.arm.com/documentation/ddi0487/latest.

[6] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (2018).

[7] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. "Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics". In: *NDSS*. 2018.

[8] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 41.

[9] *Binary Ninja*. URL: https://binary.ninja/.

[10] David Brumley et al. "BAP: A Binary Analysis Platform". In: *CAV*. 2011.

[11] Vitaly Chipounov, Volodymyr Kuznetsov, and G. Candea. "S2E: a platform for in-vivo multi-path analysis of software systems". In: (2011). DOI: 10.1145/1950365.1950396. URL: https://www.semanticscholar.org/paper/2fb716c1119e2da8f896222f48dbac11209e2486.

[12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "The S2E Platform: Design, Implementation, and Applications". In: *ACM Trans. Comput. Syst.* 30.1 (Feb. 2012). ISSN: 0734-2071. DOI: 10.1145/2110356.2110358. URL: https://doi.org/10.1145/2110356.2110358.

[13] *Cutter*. URL: https://cutter.re/.

[14] *Cutter repository*. URL: https://github.com/rizinorg/cutter.

[15] Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. "rev.ng: A Multi-Architecture Framework for Reverse Engineering and Vulnerability Discovery". In: *2018 International Carnahan Conference on Security Technology (ICCST)*. 2018, pp. 1–5. DOI: 10.1109/CCST.2018.8585654.

[16] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. "Rev.Ng: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries". In: *Proceedings of the 26th International Conference on Compiler Construction*. CC 2017. Austin, TX, USA: Association for Computing Machinery, 2017, pp. 131–141. ISBN: 9781450352338. DOI: 10.1145/3033019.3033028. URL: https://doi.org/10.1145/3033019.3033028.

[17] *Ghidra*. URL: https://ghidra-sre.org/.

[18] *Ghidra repository*. URL: https://github.com/NationalSecurityAgency/ghidra.

[19] Xuan Guo and Robert Mullins. *Accelerate Cycle-Level Full-System Simulation of Multi-Core RISC-V Systems with Binary Translation*. 2020. DOI: 10.48550/ARXIV.2005.11357. URL: https://arxiv.org/abs/2005.11357.

[20] *IDA*. URL: https://hex-rays.com/ida-pro/.

[21] Stephen Kell, Dominic P. Mulligan, and Peter Sewell. "The Missing Link: Explaining ELF Static Linking, Semantically". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 607–623. ISBN: 9781450344449. DOI: 10.1145/2983990.2983996. URL: https://doi.org/10.1145/2983990.2983996.

[22] Chi-Keung Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 190–200. ISBN: 1595930566. DOI: 10.1145/1065010.1065034. URL: https://doi.org/10.1145/1065010.1065034.

[23] *McSema repository*. URL: https://github.com/lifting-bits/mcsema.

[24] Kayvan Memarian et al. "Exploring C Semantics and Pointer Provenance". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290380. URL: https://doi.org/10.1145/3290380.

[25] Susanta Nanda and tzi-cker Chiueh. "A Survey on Virtualization Technologies". In: (Jan. 2005).

[26] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 89–100. ISBN: 9781595936332. DOI: 10.1145/1250734.1250746. URL: https://doi.org/10.1145/1250734.1250746.

[27] *Radare*. URL: https://rada.re/n/.

[28] *Radare2 repository*. URL: https://github.com/radareorg/radare2.

[29] Alastair Reid. "Trustworthy Specifications of ARM R v8-A and v8-M System Level Architecture". In: Oct. 2016. DOI: 10.1109/FMCAD.2016.7886675.

[30] *rellic repository*. URL: https://github.com/lifting-bits/rellic.

[31] *remill repository*. URL: https://github.com/lifting-bits/remill.

[32] *Rigorous Engineering of Mainstream Systems*. URL: https://www.cl.cam.ac.uk/~pes20/rems/index-intro.html.

[33] *Rizin*. URL: https://rizin.re/.

[34] *Rizin repository*. URL: https://github.com/rizinorg/rizin.

[35] Eric Schulte, Michael D. Brown, and Vlad Folts. "A Broad Comparative Evaluation of x86-64 Binary Rewriters". In: *Cyber Security Experimentation and Test Workshop*. ACM, Aug. 2022. DOI: 10.1145/3546096.3546112. URL: https://doi.org/10.1145%2F3546096.3546112.

[36] Yan Shoshitaishvili et al. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 138–157. DOI: 10.1109/SP.2016.17.

[37] Shuai Wang, Pei Wang, and Dinghao Wu. "Reassembleable Disassembling". In: *USENIX Security Symposium*. 2015.