Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Petter Duus Berven

Porting Inferno OS to ARMv7-M and Cortex-M7

Master's thesis in Computer Science
Supervisor: Michael Engel
January 2022

NTNU
Norwegian University of
Science and Technology

Petter Duus Berven

Porting Inferno OS to ARMv7-M and Cortex-M7

Master's thesis in Computer Science
Supervisor: Michael Engel
January 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Contents

List of Figures

Abstract

Abstract and thesis continues in English below

Norwegian:
Mengden av sammenkoblede enheter øker stadig. Mange av disse enhetene krever ikke tidsgarantier og er bedre tjent med administrerte operativsystemer som muliggjør rask utvikling med reduserte sikkerhetsproblemer og kompleksiteten som følger med utvikling på lavt nivå. For å muliggjøre dette forsøker denne oppgaven å portere Inferno-operativsystemet til ARMv7-M-arkitekturen som er bredt distribuert i edge-enheter over hele verden. Inferno OS er et distribuert operativsystem med innebygd støtte for kommunikasjon mellom enheter. En ny backend for den medølgende vertktøkjeden rettot mot ARMv7-M arkitekturen som produserer ARM Thumb instruksjoner er utviklet. Denne backenden resulterer i en 18% reduksjon i binær størrelse sammenlignet med ARM32 varianten. Det komplette operativsystemet er ennå ikke brukbart på grunn av den virtuelle maskinen som brukes til brukerromsapplikasjoner. Oppgaven diskuterer alternative WebAssembly som ett alternativ til den eksisterende virtuelle maskinen.

English:
The amount of interconnected devices is ever-increasing. Many of these devices do not require timing constraints and are better served by managed operating systems that enable rapid development with reduced security concerns and the intricacy of low-level development. In order to enable this, this thesis attempts to port the Inferno operating system to the ARMv7-M architecture which is broadly deployed in edge devices throughout the world. Inferno is a distributed operating system with native support for inter-device communication. A new toolchain backend for the ARMv7-M architecture that produces ARM Thumb instructions is produced. This backend results in a 18% reduction in binary size compared to the ARM32 variant. The complete operating system is not yet usable due to the virtual machine used for userspace applications. The thesis discusses WebAssembly as an alternative to the current virtual machine.

Chapter 1

Introduction

Small embedded devices and their interconnection is a topic of interest as the number of deployed embedded devices increase. The devices are deployed in a wide array of different products and can be separated into two broad categories; Real-Time and Non-Real-Time. Non-Real-Time systems are not bound by strict timing constraints and can function without timing guarantees. Real-Time systems, however, require timing guarantees to ensure correct operation. There are many available operating systems for developing Real-Time applications, however, there are few directed at Non-Real-Time applications. One key difference between these two types of operating systems is the facilities they provide the developer. Real-Time systems present the developer with a framework where tasks can be given priorities and guarantees, depending on the hardware, can be provided. These systems require the developer to use low-level systems languages, typically C/C++. Non-Real-Time operating systems however allow the developer to use higher-level languages that enable faster development and allow the developer to ignore some of the finer details of how the hardware functions. This thesis attempts to port Inferno OS to the ARM Cortex-M series CPUs. The ARM Cortex-M is ARMs series of CPUs designed for embedded systems, the one used in this thesis is the Cortex-M7. Cortex-M devices utilize the ARMv7-M architecture, which differs from the better-known ARM-A/R architecture. The differences that matter for this thesis will be explained more in-depth in the Background chapter.

Inferno OS is a small distributed operating system, officially only requiring 1MiB of RAM [1]. Inferno does not require hardware memory management functionality and is, therefore, a good contender for a Non-Real-Time operating system running on Cortex-M devices, as this memory management hardware is an optional feature for these devices.

The need for embedded intelligence in endpoint devices has facilitated the growth of small embedded devices. In the fourth quarter of 2019 ARMs partners sold a total of 6.4 billion ARM-based chips and shipped 4.2 billion Cortex-M-based devices [2]. In the fourth quarter of 2020 the numbers are 6.6 billion and 4.4 billion respectively [3], showing that Cortex-M devices constitute a large part

of ARMs' total chip sales. With a total of 225 billion devices sold in 2021 [4] and assuming the Cortex-M chips constitute a fair percentage of this sale, it is apparent that Cortex-M-based devices are flourishing throughout the world.

Many of these endpoint devices are accumulating data through attached hardware, perform various degrees of processing and then relay the data to a central system. The interaction between the endpoint devices and the central systems is facilitated by public infrastructure, be it wireless or wired. Consequently, they are possible targets for ill-intentioned actors who wish to abuse the systems for some nefarious purpose. Common problems with these systems include the lack of updates and improper security implementation in the networking stack. While Inferno does not mitigate the issues arising from outdated software it does assist with securing the connection between devices. Inferno achieves this by providing a uniform interface for accessing remote resources [5], removing the need for application developers to concern themselves with the security aspects of their application.

Microcontrollers that utilize the Cortex-M series CPUs are memory constrained. There are multiple providers of these devices. The STM32 family offers boards with memory ranging from 32KiB to 512KiB, with some sporting up to 1184KiB, although these are intended for devices that provide a graphical user interface, which is possible with Inferno although not intended for the applications envisioned here. Nordic semiconductors offer microcontrollers with networking capabilities that come with 256KiB of memory. There are also Cortex-M devices sporting 8MiB of RAM. Regardless, the target is to enable deployment to as many devices as possible, and preferably devices that many companies and organizations use to enable adaptation. It is unlikely that Inferno will be able to run on systems with less than 512KiB of memory, at least without heavy modification to how and when the system allocates memory.

This thesis looks at porting the Inferno operating system to ARM Cortex-M series CPUs, specifically the ARMv7-M architecture using the Thumb instruction set. A functional Thumb backend for the compiler toolchain is produced, although with some limitations. A primitive port for the Teensy41 has been used to evaluate the ported backend.

The OS is ported to the Teensy41 [6] board. The board uses the i.MX RT1060 [7] Micro Controller Unit (MCU) which uses a Cortex-M7 and has 1MiB of RAM divided into two 512KiB chunks.

Chapter 2

Background

As costs and size of devices have decreased they are becoming an ever more prominent part of new products. Additionally, as we seek to improve and expand how we interact with technology, the need for decentralized processing is becoming more important. While central processing is still necessary, where warehouse-scale computing is utilized, running thousands of devices with full-blown commercial operating systems running x86 systems with terabytes of memory, the devices at the end of the stream are in many cases running smaller Reduced Instruction Set Computer (RISC) CPUs and have limited memory availability.

ARM [8] has for many years provided CPU designs intended for these smaller devices. Today ARM has three profiles for their 32bit architecture, where the profile specifies the intended use case for the device. The profiles are *A* for Application, *R* for Real-time and *M* for Microcontroller. The *A* and *R* profiles support the same Instruction Set Architecture (ISA), with support for both the ARM32 and Thumb instruction set. The difference between *A* and *R* is that *A* requires support for virtual addresses in the memory management model while *R* does not. The *M* profile does not support the ARM32 instruction set, instead opting only for Thumb. The *M* profile is intended for situations where the overall size and deterministic operation is more important than absolute performance.

The focus of this work is to add support for the ARM *M* profile to the existing *5* [9] toolchain that supports the ARM *A* profile with a subset of the Thumb instruction set. The Thumb instruction set consists of 16bit and 32bit instruction encodings. The 16bit versions were added to the ARM32 instruction set and offer an opportunity for developers to use the instructions to reduce the binary size. However, for the CPU to execute Thumb instructions it has to change execution mode. Since the original Thumb extension does not support all commonly used instructions, there was a need for extending the instruction set so that it could be used without changing execution mode back to ARM32. To resolve this Thumb2 was introduced. Thumb2 is an extension for the Thumb instruction set, adding support for all the instructions required to compile a program and to control the CPU. With these features, the collective Thumb instruction set can represent any program previously represented by the ARM32 version. ARM even revised

their assembler syntax to produce the Unified Assembler Language (UAL) [10] that enables developers to produce ARM32 and Thumb compliant binaries from the same assembler source code.

While ARM has defined the UAL the two instruction sets are not equal. ARM32 generally has support for more features than Thumb, both because ARM32 can leverage Thumb instructions, assuming a modern CPU and because some features of ARM32 have not been incorporated into Thumb. The differences between the two instruction sets will be discussed in more detail when porting the loader is presented in Section 3.1.1.

2.1 Inferno

Inferno [11] is based on Plan9 [12] created by Ken Thompson, Rob Pike, Dave Presotto, and Phil Winterbottom at Bell Labs in the 1980s. Inferno is a compact operating system designed for building distributed and networked systems on various devices and platforms. Some of the intended use cases are hand-held devices, TV set-top boxes, and inexpensive networked computers.

Inferno retains the core aspects of Plan9, an expansion of the UNIX philosophy of *"everything is a file."*. In Inferno, process state, IO, and hardware devices are all remotely accessible by other devices. The kernel provides this for all devices by requiring that devices implement a standard interface. One of the exciting differences between Inferno and other operating systems is the ability to share all resources natively without involving the application developer.

Accessing the resources of remote devices requires a connection to the remote device, and the remote must have marked the resource as shared. Once these criteria are fulfilled, the resource can be mounted into the local filesystem namespace. The resource is then accessible by applications as any other resource in the system. There are no further considerations the application developer must make.

Under the hood, Inferno's filesystem uses the 9P [13] protocol. Because all devices also implement the interface to the filesystem, Inferno can leverage this protocol between devices. In Inferno, using 9P for the local filesystem and remote communication is referred to as the Styx [14] protocol. Styx defines operations for resources such as; open, read, stat and close.

Because the kernel handles remote resource access, it also handles the security aspect of this interaction. This frees the application developer from having to concern themselves with the security aspect of the application and provides a centralized implementation of protocols, making them easier to maintain.

One common concern when working with small embedded devices, such as the Cortex-M series CPUs, is that they are not guaranteed to have memory management or protection hardware, which some modern operating systems require.

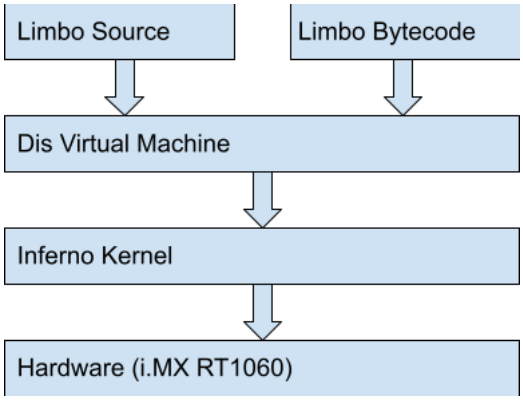


Figure 2.1: Inferno Layer Overview

Inferno does not require any such hardware in order to function. If a Memory Management Unit or Memory Protection Unit is available, Inferno can use it, although the lack of one does not present an issue for the operating system.

The market envisioned in this work is the usage of Inferno in networked devices without strict real-time requirements, such as; cameras, set-top boxes, monitoring systems, and applications commonly featured in smart-home solutions. In other words, networked devices that see broad deployment. There are other projects targeted at these devices, however, most of them are Real-Time Operating System (RTOS). Inferno being a Non-RTOS system, leads it to have certain differences from RTOS: Firstly, Inferno applications are developed using Limbo [15] a higher-level language instead of C. Limbo executes on the Dis virtual machine [16]. By developing applications in a higher-level language, developers can disregard some of the details of low-level programming. Additionally, since applications are executed on a virtual machine, they can be shared between devices as the virtual machine abstracts any hardware specific concerns. Figure 2.1 shows the basic stack for Limbo applications, since Limbo languages can either be distributed as raw code or in bytecode format there are two alternate sources. Secondly, Inferno is designed around resource sharing. All mounted resources, including hardware devices and process states, can be shared between devices without implementing protocols or being aware that the resource is remote. These benefits provide a framework to create endpoint devices as a central server commonly controls these. It also follows that since the remote connection is handled by the kernel all security mechanisms are also handled by the kernel. This improves security since there is a common implementation of the security protocols and again, these security protocols do not need to be implemented by the application developer.

The limbo language used for Inferno can either be interpreted or compiled to bytecode. Since the dis VM is by default present in Inferno these applications can be executed on any device without modification. Assuming the device has the necessary optional features enabled as is discussed in the next section.

2.1.1 Configuration

When working with Inferno, it is possible to select the features that should be built into the kernel. These features are selected from the pool of available features using a configuration file. In addition to selecting the features wanted in the kernel, the configuration file can also be used to supply constants that will be made available during the compilation stage of the kernel.

The configuration file is split into ten sections:

- `dev` - device drivers compiled into the kernel
- `root` - filesystem paths and raw files
- `ip` - Connectivity protocols
- `link` - hardware-specific complements for device drivers
- `misc` - architecture-specific complements for device drivers.
- `lib` - Libraries to include in the kernel
- `mod` - Dis modules to include
- `port` - Portable components to include in the kernel
- `code` - C parameters provided during kernel build
- `init` - Specify the userspace initialization program

In the first section `dev`, the devices that should be part of the kernel are specified. These devices can be hardware-specific device drivers such as `uart` or general kernel devices such as the OS `root` serving as the filesystem root and the `prog` device which handles the scheduling of processes in the kernel.

In the second section `root`, a list of files and directories that should be included in the kernel is specified. There are often a few required files as they are needed by features enabled in the other sections, for instance, the `chan` port requires that `/chan/` is mounted in the `root` section.

In the third section `ip`, the desired network protocols are specified. A list of the currently available protocols is shown in the Appendix C.

In the fourth section `link`, hardware-specific parts of the device drivers can be defined. For instance, serial and LED interfaces.

In the fifth section `misc`, architecture-specific interfaces can be supplied. Examples are Direct Memory Access(DMA), co-processor, and MultiMediaCard(MMC).

The sixth section is the `lib` section. Here the libraries that should be linked into the kernel are specified. A list of the currently available libraries shipped with the Inferno source code is shown in the Appendix D

The seventh section is the `mod` section. Here the builtin Dis modules one wishes to use are specified. The `sys` module is required for interaction with the kernel from the VM.

The eight section is the `port` section. Here portable components, excluding those that are defined as drivers, can be specified.

The ninth section is the `code` section. Here C code can be specified which will be included as-is in a generated configuration file which is then available to the rest of the source. Common variables are the size of the different memory pools and if the console should be enabled.

The tenth section is the `init` section. This section specifies the name of the Dis script that should be launched on startup. The script is then responsible for attaching the required devices and starting any user interface that is necessary. The `init` script configured in this thesis attaches common devices and then initiates a shell session with input from the UART interface.

The configuration file used through this project in order to produce a bootable Inferno kernel is show in the Appendix B.

2.1.2 Ports

Inferno uses the concept of ports. Ports contain the specific configuration, device drivers, and non-portable features used to run Inferno on a specific platform. Commonly ports contain:

- A configuration file as presented above.
- Files that define constant values.
- Implementation of hardware-specific procedures.
- The main function that enables and configures features of the OS.
- Device drivers needed by the system.
- Tooling to produce a bootable binary file.

Some of the constant values required by the system are: Where the kernel's memory starts, the address of the core *machine* structure, and the start of the text segment.

Hardware-specific procedures relate to interrupts/exception handling, exit, and reboot.

The main procedure enables desired features, configures memory pools, and initializes the kernel. Once the kernel is configured, the main procedure is responsible for initializing userspace by initializing the Dis VM and loading the configured initialization script defined in the `init` section.

Adding device drivers is done by implementing a `Dev` interface for the hardware structure. The implementation is responsible for abstracting away any hardware-specific operations so that the device can be accessed in the same way as any other. Once the interface is created, the device can be hooked into the kernel and will be available for other applications to use.

Since different platforms have different requirements for the binary format, the port implementation should also provide a method to produce the final binary in the correct format.

2.1.3 Compiler Toolchain

Inferno is distributed with a custom C toolchain. The toolchain consists of three components: The assembler, compiler, and loader. All three of these consist of architecture-independent and architecture-dependent parts. Generally speaking, the assembler and compiler produce intermediate representations from the assembler and C input data, respectively, and the linker converts this representation to the binary format of the target architecture.

Inferno already supports producing binary instructions for the ARMv6-A architecture with partial support for Thumb instructions. The added ARMv7-M backend discussed in this thesis is based on this ARMv6-A backend.

Assembler

The Inferno toolchain's assembler, just like any other assembler, enables the programmer to define low-level procedures. One big difference between the Inferno assembler and the GCC/LLVM counterparts is that Inferno employs an internal assembler syntax while the others enable the usage of ARM's UAL [10]. The UAL allows programmers to write assembler code that can be compiled to both ARM32 and Thumb instructions. While the Inferno assembler for Thumb and ARM32 are very similar, there is currently no practical way of implementing on-demand predication without inserting `nop` instructions in Thumb.

An important feature of the Inferno assembler is the ability to specify instructions that the compiler is not able to produce due to limitations in the C syntax. Since the Inferno compiler does not support inline assembler in the C source, the assembler must enable developers to use hardware features that cannot be expressed in C.

Compiler

The Inferno compiler implements ANSI C with some restrictions and extensions. The compiler uses a YACC-based parser to produce a block-structured symbols table. The compiler performs nine passes on this structure, resulting in an object file that the linker can process. Of the nine passes performed by the compiler, four are machine-dependent. The other passes are part of the shared compiler infrastructure, and consequently, any changes to them will impact the compilers for other targets as well. Therefore, only the four machine-dependent passes are relevant to review when adding a new compiler. These four passes are addressability, code generation, registerization, and machine-code optimization.

The functionality of the compiler is hardly modified in this work, as the base compiler produces object files that conform to the format of the loader. Therefore only a slight description of the machine-dependent passes is presented to give an overview of the procedures that might be modified for future optimizations of the compiler.

Addressability concerns the legal address field expressions of a machine language instruction. Both ARM32 and Thumb are load-store architectures and employ identical rules regarding addressability, with slight variation in the immediate values used. In some cases, Thumb1 instructions limit the number of valid registers to the first seven. However, there are extended Thumb2 variants that support the complete set of registers. While performing the addressability pass the compiler will label each node with a Sethi-Ullmancomplexity, which is roughly equal to the number of registers required to compile the tree.

The next pass is code generation. Code generation uses recursive descent through the structure mentioned above. It converts the structure from a syntax tree to machine code in an internal format. This is then used in the following steps.

Using the machine code in internal format produced in the previous pass the compiler performs registerization. The purpose of this pass is to reintroduce registers for heavily used variables.

The final machine dependent pass is machine code optimization. This pass is a catch all for opportunistic optimizations that fit the target architecture.

Loader

The final part of the toolchain is the loader. Both the assembler and compiler produce object files that are passed to the loader which converts the internal format used in the object file to a binary file with instructions for the targeted architecture.

The loader is the most relevant part for this thesis. Since all parts of the toolchain are based on a Thumb1 implementation they already respect most of the differences between Thumb and ARM32 instructions. Therefore, in most cases the work is related extending the loader to support new instruction formats.

2.2 ARMv7-M

The ARMv7 architecture is available in three profiles; *A*, *R* and *M*. Both *A* and *R* support mixing ARM32 and Thumb instructions, while *M* only supports Thumb. The profiles have different suggested areas of usage, the one of concern here, the *M* profile, is targeted at microcontrollers where overall size and deterministic operation for an implementation is more important than absolute performance.

The Thumb instruction set has been released in two instances, first it was a supplement to the ARM32 instruction set, providing the ability to use 16bit instructions alongside the 32bit ARM instructions. Later, in 2003 ARM announced an extension to the Thumb instruction set with 32bit instructions as well, with this the number of features were extended enabling applications to be developed solely with Thumb. The reasoning behind extending Thumb was that, while ARM32 and Thumb instructions can be mixed, it requires that the CPU's execution mode is changed between the two. The execution mode can be changed by using the BranchX instructions. While this only incurs a slight degradation in performance it complicates the hardware design and with limited support for operations in the first iteration of Thumb, the possible applications are limited.

Thumb instructions consists of halfwords. The value of the most significant bits specify whether the next halfword is also part of the same instruction. The separate widths are referred to as *narrow* for one halfword and *wide* for two, with many instructions offering both narrow and wide variants. The first iteration of Thumb consists of mostly narrow instructions, although there are some wide instructions such as branches.

When instructions are wide the bits [15:11] are set to any of the following values: `0b11101`, `0b11110` or `0b11111`. Figure 2.2 shows the 16bit T1 encoding of *add register* and equivalent 32bit T3 encoding. One difference of note between the encoding is that the 16bit version does not support the full set of 16 registers, since only three bits are reserved for register selection. This distinction is present in many of the 16bit variations, presenting a new challenge when optimizing for space during the register allocation phase of the compiler.

2.2.1 ARMv7-M vs ARMv6-A

The Inferno ARM toolchain already present in mainline Inferno supports the first iteration of Thumb instructions interleaved with ARM32 instructions. However, there is only support for Thumb1 instructions. In other words, it's a ARMv6-A

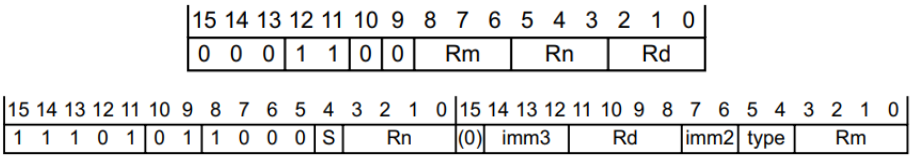


Figure 2.2: Add register 16 vs 32bit

compiler. Since this instruction set only allows code size reduction for ARM32 by using Thumb1 instructions where possible, it is not able to produce working ARMv7-M binaries. Additionally, the ARMv6-A architecture differs in usage of exception and special registers. Therefore the compiler needs to be modified so that the ARM32 cases are altered to produce valid Thumb2 instructions instead in addition to replacing the exception handler structure and instructions used to access special registers.

2.2.2 Predication

ARM Thumb instructions do not support the predication commonly used in ARM32. Predication is the ability to specify the condition under which the instruction should execute without other control structures. This can be written in assembler as `addeq`, where the *add* instruction should only execute if the state of the CPU has the zero flag set to true, which was set by some preceding instruction.

Instead of this per-instruction predication, Thumb supports the usage of *If-Then*(`IT`) blocks. `IT` blocks consist of one `IT` instruction that sets the predication for up to four subsequent instructions.

2.2.3 LDM/STM

The LDM and STM instructions enable loading multiple registers from a memory region or storing multiple registers to a memory region with a single instruction. Both LDM and STM have variants that define how the provided memory address should be changed for each register. One type is Increment After(`IA`) in which the address is incremented after each register. The other is Decrement Before(`DB`) where the address is decremented before each register. The ARM32 instruction set supports two additional variants that are not supported in Thumb, Increment Before(`IB`) and Decrement After(`DA`). If these are required they can be implemented with a combination of instructions. By offsetting the base register by four prior to the instruction so that `LDMIB rx, {}` becomes `ADD rx, #4; LDMDB rx, {}; SUB rx, #4`. The restoration of `rx` is only necessary if subsequent instructions depend on the value of `rx` or if writeback is enabled.

Common usecases for LDM and STM in Inferno is in saving or restoring registers

when calling and returning from procedures, as well as copying fields from one struct to another. These instructions are commonly used in the 64bit library of Inferno which is presented in 3.1.5.

2.2.4 ARMv7-M Exceptions

In ARMv7-M exceptions are normal procedures that do not require any special handling by the compiler or developer. The CPU will always save the caller-saved registers (R0-R3, LR, PC, and PSR) prior to calling the handler. Because of this, implementing exception handling with ARMv7-M only requires that exceptions are enabled and that the handler is registered. Registering the handler can be done by setting the address of the handler in the vector table or pointing the Vector Table Offset Register (VTOR) to an array at runtime.

2.2.5 Size comparison

One of the selling points of the Thumb instruction set is that it can reduce the binary size. The major constraint when using ARMv7-M is that devices that use this architecture are usually severely limited in the amount of memory available. Therefore, one important aspect of porting Inferno to this architecture is that the resulting binary is smaller, so that it may be used on a general selection of devices, rather than a subset of devices that have high memory availability.

ARMv7-A/R also support using Thumb instructions and can therefore optimize for size by opting for using only Thumb instructions, however in this case the ARM32 pipeline is unnecessary and adds costs to the device. Also, this is only an option if the toolchain supports the full Thumb instruction set, which Inferno currently does not.

Because size is a primary concern for adopting Inferno in the ARMv7-M space the following section will look at the binary size of simple C programs. A comparison between the existing ARMv6-A compiler, the produced ARMv7-M compiler and the ARMv7-M variant of GCC is shown in Figure 2.3. The source for the programs is found in the Appendix A.

As can be seen from the Figure, there is a fair amount of space to be saved by using the Thumb instruction set, additionally it is apparent that there is room for adding size optimizations to the Inferno compiler as the results from GCC show.

The `call_deep` program tests the calling convention used by the toolchain. As can be seen from the graph, this is one area where the ARM32 version outperforms Thumb in size. The reason for this is that the linker has not been made aware that it can use store instructions that access the link register. It assumes it only has access to Thumb1 instructions, where storing to registers with number higher than seven is not possible. Therefore, it copies the link register to register

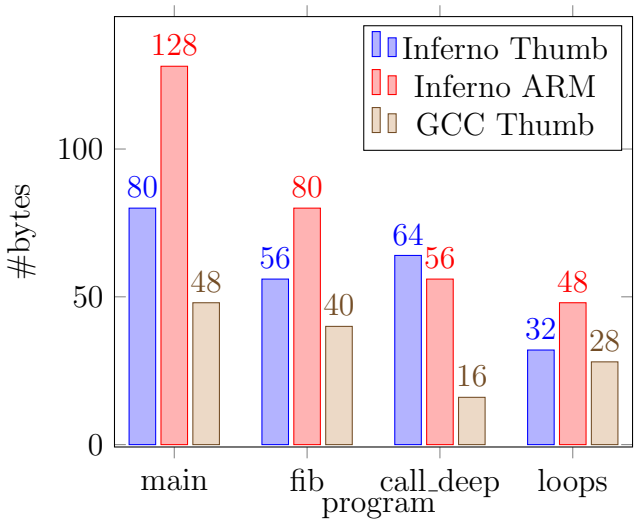


Figure 2.3: Size of binary with different compilers

six and then stores that register onto the stack. Additionally it modifies the stack pointer with a subtract instruction, whereas the ARM32 version uses the writeback features of stores to update the stack pointer at the same time as it saves the return address onto the stack. This will be further discussed in 3.1.3

2.3 Teensy41

The board used to evaluate the new backend for Inferno is the Teensy41 [6]. The Teensy41 utilizes the i.MX RT1060 [7] microcontroller which provides a ARM Cortex-M7 CPU running at 600MHz. It has 1MiB of RAM divided into two separate 512KiB blocks, and 7636KiB of Flash memory. It supports many external interfaces, although in this work only the UART interface is of interest.

The Teensy41 has multiple memory regions: Instruction Tightly Coupled Memory(ITCM), Data Tightly Coupled Memory(DTCM), On-Chip RAM(OCRAM) and OCRAM2. ITCM, DTCM and OCRAM share the same 512KiB block of memory which can be assigned to the different memory regions in 32KiB blocks. While all three share the same physical memory unit, the memory addresses are different for all. ITCM starts at address 0x0, DTCM starts at 0x20000000 and OCRAM starts at 0x20280000. OCRAM2 has a full 512KiB block of memory starting at 0x20200000. ITCM being intended for instructions has a single 64bit interface while DTCM, intended for data, has two 32bit interfaces.

2.3.1 Boot sequence

The i.MX RT1060 supports various boot procedures, an overview of the process is shown in Figure 2.4. The one used in this thesis is SPI NOR. Specific details of the boot sequence is not important in order to understand the contribution of this

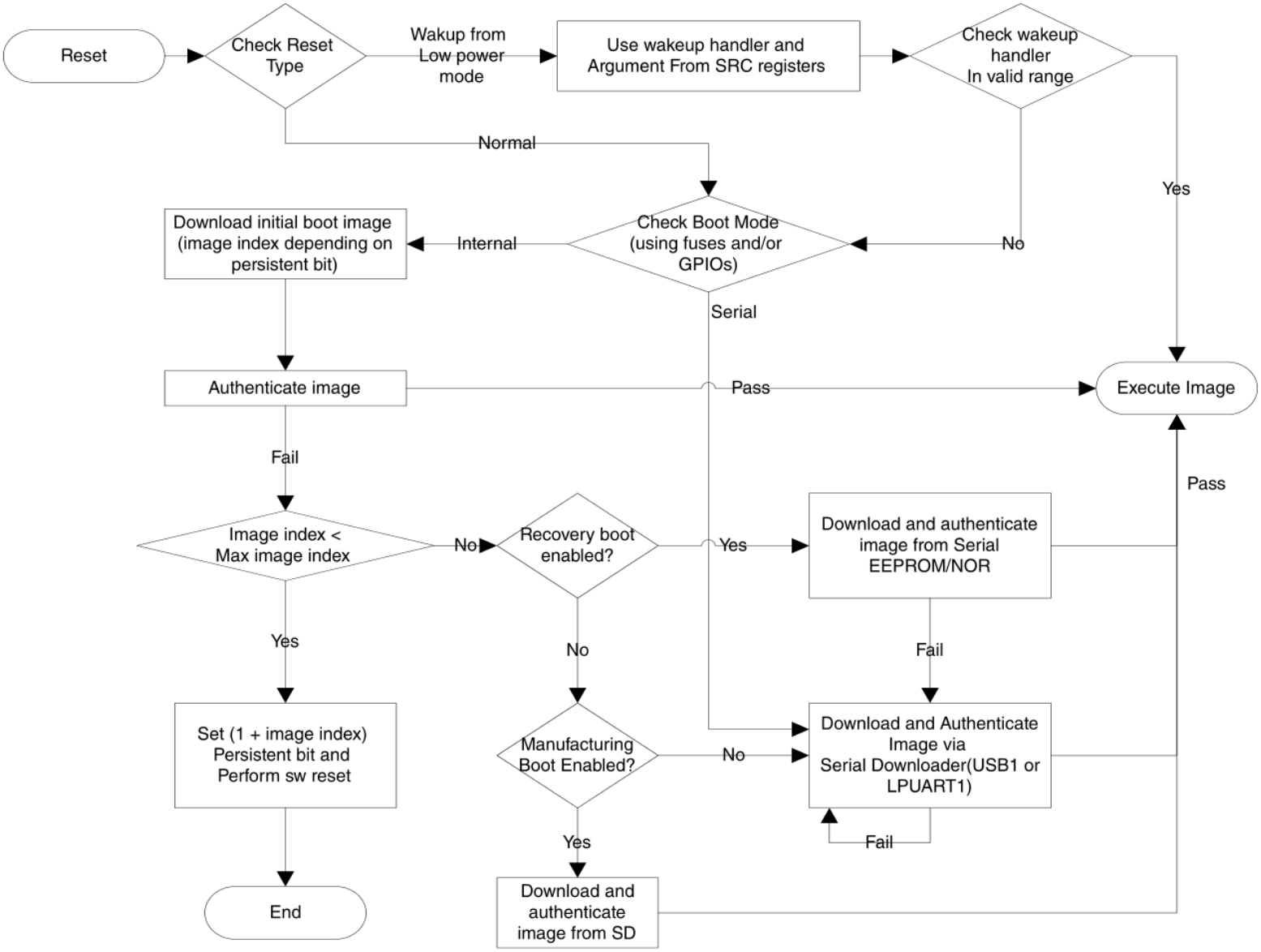


Figure 2.4: i.MX RT1060 boot sequence([17])

paper, and interested readers are referred to the reference manual [17]. The i.MX RT1060 requires that the beginning of the binary contains a specific structure, this structure is used to specify the type of boot sequence and provide some variables used during boot. It also requires the implementation of a setup procedure that configures hardware devices on the board, loads the main application into RAM and finally branches into the main procedure of application.

Chapter 3

Method

This thesis is concerned with adding a new target architecture to the Inferno toolchain and porting Inferno to the Teensy41 board running the i.MX RT1060 MCU. This chapter presents how the toolchain is extended, what extensions have been made, and why. The process of porting Inferno to a new target device is also presented, along with the hardware-specific adaptations made to the operating system. The chapter starts with the toolchain, followed by Inferno, then the specifics of loading the kernel onto the Teensy41, and finally presents some general debugging methods used throughout the process.

3.1 Porting the Toolchain

Porting the toolchain is the core contribution of this thesis as it enables others to develop their own ports of Inferno for the ARMv7-M architecture. The parts of the toolchain that have been modified are the compiler, assembler and loader. Both the compiler and assembler are concerned with parsing a source file and producing an object file. This object file can then be consumed by the loader in order to produce an executable binary for the target architecture. Consequently, both the compiler and assembler are constrained by the capabilities of the loader. As such, this section will first present the loader so that the operations of the others are better understood.

3.1.1 Loader

The method used for porting the loader consists of three broad parts: discovery, extension, and testing. The discovery involves mapping out the combinations of internal instructions and parameters the compiler and assembler produce. The second part, extension, is concerned with implementing the missing combinations. The third and final part, testing, is concerned with verifying the added encodings and adding checks to the loader, so it provides errors for unsupported parameters or conditions.

Discovery

Since the loader is based on the existing ARM32 loader with Thumb1 capabilities, and the ARM32 instruction set is similar to the Thumb instruction set, the discovery part was performed by pushing the loader to produce Thumb instructions rather than ARM32. In order to push the loader, two trivial modifications were made to the main loop of the loader. First, when the loader begins processing a new `TEXT` block, which is equivalent to a procedure, it marks it as Thumb by setting a flag in the Intermediate Representation (IR) structure. When the loader begins emitting binary instructions for the operations contained within the `TEXT` block it uses this flag to set a global state `thumb` to 1. When `thumb` is set to 1 the loader will use the Thumb structures to lookup operations and produce binary instructions.

This method works for all the instructions that are intended to be supported. However, one condition will cause the loader to revert to ARM32, and that is when it encounters floating-point instructions. In this case, it will unset the thumb flag for the `TEXT` block and produce a warning with the name of the procedure and the instruction it encountered that made it revert to ARM32. Again, this is because floating-point support is beyond the scope of this thesis.

While floating-point support is not required to run the kernel with the configuration used in this thesis, some procedures contain branches that utilize floating-point. Since the entire procedure is promoted to ARM32 this will raise an exception when branching to the procedure. To prevent the exception from crashing the runtime, procedures that contain these branches have had these branches replaced with an error so that the procedure remains thumb compliant and when floating-point support is added these branches can be reverted to operate correctly.

With the addition of the modifications described above, the loader will produce an error when it encounters unsupported operations previously supported by ARM32. This provides the feedback necessary to move to the extension stage.

Extension

The process of extending the loader is, in most cases, simple. Only two parts of the loader source need modifications. First, a new entry is added to a structure called `optab`. The `optab` is used to lookup supported combinations of operations and parameters. After extending the `optab` an encoder is added to the `asmout` procedure, which processes the operation and supplied parameters to produce the appropriate binary representation.

The `optab` is an array of type `Optab` seen in Figure 3.1. The `as` property is the operation type, for example if the `as` is set to `AADD` then it is an *add* operation. The first A is used to namespace the enumerations. The instruction types are

```
struct Optab {
    char as;
    char a1;
    char a2;
    char a3;
    char type;
    char size;
    char param;
    char flag;
}
```

Figure 3.1: Loader optab structure

shared by the compiler, assembler, and loader so that the value is equal in all of them. The next three properties `a1`, `a2`, `a3` specify the type of data the instruction will be working on, these are commonly either register or constant. The `type` property specifies how the binary output should be processed in `asmout`. The `size` property defines the size of the encoded instruction in bytes. When there are one-to-one mappings between internal operation and the ARMv7-M instruction set the size is either two or four. Some special combinations of operations and parameters result in more than one binary instruction. An example is when an immediate value exceeds the supported size of the operation so the loader will insert data operations to produce the immediate. It can produce these sequences without modifying the internal representation or special handling beyond the `asmout` handler producing all the necessary operations.

Adding a new supported combination of an operation and parameters is done by adding an entry to the `optab` with the operation and the parameter types. To reduce the number of entries in the `optab` it is possible to alias one operation to another. This is preferable because many instructions are identical except for a few bits specifying the sub-operation. For instance, ADD, SUB, MOV, MVN, and CMP with parameters of a register and an immediate are all identical except for bits [8:5] of the first halfword. Selecting the specific bits is therefore left to `asmout`. This results in fewer entries in the `optab` improving readability and fewer entries in `asmout`. One downside to this approach stems from the fact that aliasing is done globally. Therefore, if there are operations that are equal in some cases, yet differ in others, then they will still resolve to the same `optab` entry, so it is necessary to check what types alias to the newly added entry and either add special handling for edge cases in `asmout` or emit an error informing that the combination is not yet supported.

By adding a new entry to the `optab` the loader will be able to recognize the pattern and pass the arguments on to the `asmout` procedure. The `asmout` procedure is primary a large switch statement. It selects the appropriate case from the `type` in

the `optab` entry it is passed. Each entry in the switch statement stores the binary representation of the instruction in local variables, these are then, depending on the `size`, written to the binary file. Common checks performed by the cases in `asmout` are: check that the arguments are valid, for instance, validate that the register number is within the bounds of the requested encoding. If any input variables are immediate values the handler should validate that the value is within the acceptable range and transform the immediate to the format expected by the instruction.

In the case of immediate encoding, one addition made by this work is the addition of a `ThumbExpandImm` encoder. The `ThumbExpandImm` encoding is optionally used by some of the data processing instructions that accept immediate values. The format enables the representation of 32bits of value given 12bits. The format is not able to present the full scope of the 32bit values, instead, the first four bits are used to specify the pattern of the latter eight bits. The encoding table can be seen in Table 3.1. The implemented encoding does not support all patterns, specifically the repetitive patterns. Still, it is featureful enough for the Inferno configuration used.

Table 3.1: Thumb Expand Immediate encoding

i:imm3:a	const[a]
0000x	00000000 00000000 00000000 abcdefgh
0001x	00000000 abcdefgh 00000000 abcdefgh[b]
0010x	abcdefgh 00000000 abcdefgh 00000000[b]
0011x	abcdefgh abcdefgh abcdefgh abcdefgh[b]
01000	1bcdefgh 00000000 00000000 00000000
01001	01bcdefg h0000000 00000000 00000000
01010	001bcdef gh000000 00000000 00000000
01011	0001bcde fgh00000 00000000 00000000
⋮	8-bit vaules shifted to other positions
11101	00000000 00000000 000001bc defgh000
11110	00000000 00000000 0000001b cdefgh00
11111	00000000 00000000 00000001 bcdefgh0

[a] In this table, the immediate constant value is shown in binary form, to relate abcdefgh to the encoding diagram . In assembly syntax, the immediate value is specified in the usual way(a decimal number by default)

[b] UNPREDICTABLE if abcdefgh == 00000000

In addition to the operation and parameters `asmout` can produce alternate encoding based on a value `scond`. The `scond` is passed by either the compiler or

the assembler and provides additional conditions for the operation. These conditional values were used for predication in ARM32, and are still used for supplying parameters to load/store operations. These operations have optional parameters for updating the register used as the base address, whether it should use post or pre-increment, and if ut should add or subtract the immediate value from the base register.

Testing

When evaluating the produced encoding it was compared to the ARM32 equivalent, which is known to be correct. In most cases the instructions are identical. This makes it simple to detect if the loader encodes instructions correctly. In some other cases, such as conditional execution, more effort is needed to compare the two results. The comparison was made for already existing logic to spot immediate differences, then very simple tests were written which could be produced using the assembler or compiler and then validated manually. Manual validation, while slow, is in many cases quite simple to perform if the source is known and the number of instructions is limited.

Since many features were missing from the loader used as the base for the produced Thumb loader, starting with the entire OS would result in there being many erroneous encodings and it would be hard to discern issues. Therefore, in addition to manually verifying the produced output and before using the loader to build the Inferno kernel, a simple qemu [18] instance running on the Cortex-M4 was used to test the basic features of the loader. The goal for the qemu application was to define the vector table used to startup the Cortex-M core and test the features the loader was missing but were required in order to build the kernel itself. There are some benefits of using qemu over a physical device that made it a good candidate: Launching a new build on the qemu device is faster than flashing onto a physical device, and exceptions are easier to spot in qemu since it will immediately crash with a basic error condition, and qemu requires very little setup and no physical connections. The qemu application produced can boot on the Cortex-M4, enable the UART interface and read/write messages through this interface.

3.1.2 Compiler

The first step in porting the compiler was to choose the base compiler on which the port would be based. For this, the existing ARMv6-A compiler was chosen. The reason for using the ARMv6-A backed as the base was in part because it already had support for Thumb1 instructions which resolves the compiler side differences between Thumb and ARM32 instructions. Furthermore, there already exists a raspberry PI port for the ARM32 version which made it possible to run the kernel and inspect the kernel on a fully functional system.

Following the base selection it is important to recognize which optimizations/concepts are incompatible with the new target. Since the existing compiler already supports Thumb1 it already recognizes the key differences between the ARM32 and Thumb instruction set such as predicates ad limited register usage. Beyond the differences already remedied by using the Thumb1 compiler there are no architecture dependant alterations required by the compiler.

One unexpected issue with using the ARM32 toolchain as the base for a Thumb port was that it had explicitly removed forwarding of conditional parameters for Thumb. In the ARM32 compiler, a field called `scond` is used to set conditional parameters that can alter the resulting encoded instruction. In the Thumb compiler, these parameters were explicitly removed and replaced by a constant value. This constant is fine for most instructions in the Thumb instruction set, however, it is necessary in the case of LDM and STM instructions. Since these instructions support alternate modes of operation it is necessary to know which mode is expected. Re-implementing the `scond` parameters was done by backporting it from the ARM33 compiler. This involved updating the value of the `scond` value in the same manner as the ARM32 compiler and passing it through to the object file output. Because the interpretation of the `scond` parameters change depending on the type of instruction being encoded the parameters are only forwarded the `MOVM` operation which is encoded as either STM or LDM instructions.

3.1.3 Assembler

As with the rest of the toolchain, the assembler is based on the ARMv6-A assembler. As with the compiler, the assembler uses a YACC-based parser to parse the source assembler. Alongside the parser, a lookup table enables conversion from convenient string-based symbols to types and constant values.

The structure of the entries in the assembler lookup table is:

```
char *name;
ushort type;
ushort value;
```

The name is the symbol that can be used in the assembler file, the type is used internally so that operations are only performed on the relevant types, the type is also used to select the relevant encoding in the loader. Finally, value is the value assigned in the intermediate representation. For instance, the symbol `R1` is mapped to a type of `LREG` with a value of 1.

The process of extending the assembler depends on what type of extension is required. Since the assembler already supports the required features of ARM32 there is no need for extending the syntax of the assembler. The only addition to the assembler made for the purpose of porting it to ARMv7-M is to add a few new keywords used to access special registers in the architecture. This extension

relates to updating and reading the PRIMASK register. In order to add support for this in the assemble all that is required is to add a new entry to the lookup table on the form:

```
"PRIMASK",      LPSR,   4,
```

The ARMv6-A assembler uses the LPSR type to modify the Application Program Status Register(APSR) register. While the PRIMASK is not part of the ARMv7-M special-purpose Program Status Registers xPSR the value type LPSR is used because it falls under the special-purpose mask registers, which are not that far removed from the xPSR registers. Additionally, by using the existing value type, no further modifications are required in the assembler.

Improving stack save and restore

The Inferno loader saves the LR register when a procedure is called, updates the stack pointer, and performs the inverse operation before returning. The only exception to this is if the procedure is a leaf node, meaning it does not call any other procedures. Because the Thumb1 ISA does not support load and store operations with registers higher than 7, this operation, therefore, requires three separate instructions. With Thumb2 these three operations can be performed with one instruction. Figure 3.2 shows the old and new setup for saving the LR and updating the stack pointer.

```
sub sp, #4
mov r6, lr
str r6, [sp, #0]
```

(a) Old update SP and save LR

```
str.w lr, [sp, #-4]!
```

(b) New update SP and save LR

Figure 3.2: New VS old update SP and save LR

The existing loader already had support for this if it was running in ARM32 mode. Therefore, this feature could be added by adding support for the new store and load instruction. However, for ARM32 the load and store instructions accept 12bit immediate with writeback, while Thumb only supports 8bit immediate if writeback is required. In cases where the stack pointer needs to be moved by an immediate larger than 8bit, the loader uses the old approach.

3.1.4 Added instructions

Since the original loader only implements Thumb1 instructions, some additional instructions have been added so that the loader is capable of producing all the instructions required to compile the kernel.

The following list shows the instructions that have been added to the loader. They are shown with the UAL syntax since that is the syntax most will be familiar with. In Appendix E the same list is shown with the encoding formats used and the `optab` entries added for each instruction. Some items in the list below resolve to two instructions, these are shown with two instructions on one line separated with a semicolon(;).

- Usecase: Store the PC or SP to a memory address. The reason it consists of both a move and a store is that using the PC as the source register causes unpredictable behaviour.

  UAL syntax:

  ```
  mov PC/SP, rtmp; str PC/SP, [rtmp, offset]
  ```

- Usecase: Load the PC or SP from a memory address. Same reasoning here for using two separate instructions as the case above

  UAL syntax:

  ```
  ldr rtmp, [rx, offset]; mov rtmp, PC/SP
  ```

- Usecase: Read the current PRIMASK value. Currently there is only support for PRIMASK as it was the only one required. Extending it would be simple, although it also requires that the assembler is extended because the compiler would not generate this sequence from the C source code.

  UAL syntax:

  ```
  mrs rd, PRIMASK
  ```

- Usecase: Set the value of PRIMASK. Same reasoning as the one above.

  UAL syntax:

  ```
  msr PRIMASK, rd
  ```

- Usecase: 32bit dataprocessing instructions, used for larger immediates and registers higher than seven. These are added because they support larger immediate encoding than the Thumb1/narrow equivalent and they support 4bit registers.

  UAL syntax:

  ```
  and rd, rn, #const
  add rd, rn, #const
  orr rd, rn, #const
  ```

- Usecase: Store multiple registers at once. Used when copying structs or when assigning many fields to a struct at once.

  UAL syntax:

  ```
  STMIA rn, <registers>
  STMDB rn, <registers>
  ```

- Usecase: Store and load multiple registers at once. Used when copying structs or when assigning many fields to a struct at once.

  UAL syntax:

  ```
  LDMIA rn, <registers>
  LDMDB rn, <registers>
  ```

- Usecase: Not used yet, will be used by userspace to trap into kernel procedures.

  UAL syntax:

  ```
  svc #imm8
  ```

- usecase: Used by the manual div and modulo implementation.

  UAL syntax:

  ```
  RSB.W rd, rn, #const
  ```

- Usecase: load from memory using a register as an offset. The versions with support for writeback are used in the improved stack save/restore implementation.

  UAL syntax:

  ```
  ldr rt, [rn, #+/-imm8]
  ldr rt, [rn, #+/-imm8]!
  ldr rt, [rn] #+/-imm8
  ldr rt, [rn, imm12]
  ```

- Usecase: Perform logical AND operation on a shifted register to update conditional flags.

  UAL syntax:

```
TST.w rn, rm, shift
```

- Usecase: Shift register by register value, added to enable shifting on registers higher than 7.

  UAL syntax:

```
LSR.W rd, rn, rm
LSL.W rd, rn, rm
```

- Usecase: Enable usage of high registers for AND, ORR, and EOR.

  UAL syntax:

```
ORR.W rd, rn, rm
AND.W rd, rn, rm
EOR.W rd, rn, rm
```

- Usecase: store with writeback and post/pre-increment.

  UAL syntax:

```
STR rt [rn, #-imm8]
STR rt [rn], #+/-imm8]
STR rt [rn, #-imm8]!
```

3.1.5 Special Handling

The Inferno toolchain does not expect that the target architecture natively supports all features the OS requires. When the compiler encounters requirements that it knows the architecture does not support, it will insert predefined symbols that handle these operations. The loader then expects these symbols to be available during linking. There are a few instances where this is necessary for the Inferno kernel.

The first instance is the handling of 64bit integers. While Inferno is targeted at 32bit architectures, some OS structures rely on 64bit integers. In order to provide 64bit support, the loader inserts predefined procedures when it encounters 64bit operations. These procedures must therefore be provided to the loader when building a program that requires 64bit support. In the case of the kernel, these procedures are provided by *libkern*. In *libkern* there are structures for holding the 64bit values along with procedures for data processing such as compare, add, and sub. These procedures are implemented in C. However, they are also defined

as architecture-dependent, therefore, a copy of the ARM32 implementation is provided for Thumb.

The second instance is for division and modulo operations. While ARMv7-M supports integer division, it does not support modulo operations. There are two available methods for adding the missing features. Either the compiler will replace division and modulo operations with predefined symbols that must be linked, much like the 64bit procedures discussed above, or the loader can be extended to provide support for these operations. Since porting the existing division and modulo procedures from ARM32 to Thumb was a fast non-intrusive method of providing support for the missing procedures, that option was selected.

3.2 Inferno

Porting Inferno to a support ISA is concerned with implementing drivers for hardware devices, producing a bootable binary and configuring the memory layout. Most of the setup code can be sourced from existing ports and only minor modifications are needed to achieve a running system. In the case of this thesis there are some additional concerns since the kernel uses certain features that are architecture dependant that are not provided by the port. The reason these procedures require manual porting is that they are implemented in assembler. By basing the ports on the existing ARM32 implementation the primary concern is to replace predication with branches.

Since the kernel configuration used for building the kernel in this thesis uses a minimal amount of libraries only a couple of libraries have received the necessary modifications to support the ARMv7-M architecture. The first library is the `libkern` which implements many kernel specific procedures. In `libkern` the procedures `memmove`, `memcpy`, `memset`, `strchr`, `div`, `_divu`, and `_modu` have been modified and tested.

In the case of `memmove`, `memset` and `strchr` the existing ARM32 implementations rely heavily on load and store operations with writeback and post-indexing. While this is supported by the Thumb instruction set, it was decided that adding the 32bit implementiation to the compiler would require modifications beyond extending the `optab` structure, as the 32bit version is only required in the case that writeback is necessary(there is no option to have post index without writeback). Therefore the pre/post incrementing was added manually in the assembler files using ADD/SUB on the appropriate register before or after the instruction. Figure 3.3 compares the ARM UAL syntax, Inferno ARM32 assembler with post increment and Inferno Thumb assembler with manual post increment.

```
strb r4, [r2], #1
```

(a) ARM Unified Assembler Syntax

```
MOVBU.P
R(4), 1(R(2))
```

(b) Inferno Assembler with Post Index

```
MOVBU R(4), (R(2))
ADD $1, R(2)
```

(c) Inferno Assembler without Post Index

Figure 3.3: Comparing STRB with post incrementation variants

3.3 The Interpreter

The Dis interpreter is implemented in *libinterp*. It uses a large number of macros to define how to produce the architecture specific instructions. As of now this has not been ported, therefore the userspace aspect of the OS is unusable since this is implemented with Limbo running on the Dis VM. It is however possible to start other binaries, for instance another virtual machine, and leverage the Inferno API from this binary.

3.4 Building & Booting

On ARMv7-M systems booting the device is handled by providing the vector table at the start of the binary, with the reset handler and the stack pointer, along with optional additional exception handlers. The Teensy41, however, employs a separate boot path. There are multiple boot schemes available for Teensy41. The SPI NOR sequence is the one used and is the one discussed here. First, the boot configuration is read from the start of flash, address 0x60000000. This configuration defines the type of boot sequence and provides other values required to configure the system. One essential value in the configuration is the address of the setup handler. This address is sourced and jumped to immediately after parsing the configuration. The setup handler is responsible for configuring the system's hardware components, loading the actual binary from flash into memory, and branching into the reset handler of the actual binary.

The SPI NOR implementation used was sourced from the Teensy41 Arduino SDK [19] and modified so that it would work with the Inferno toolchain. Since there is no linker file available, the configuration structure is defined in assembly and provided as the first parameter to the loader, ensuring that it is the first part of the resulting binary.

Following the initial boot structure, the setup handler is loaded. Next, the handler configures the entire first 512KiB of memory as ITCM memory. This is done to provide as much contiguous space as possible for the main binary in an effort to avoid needless memory issues during the development process. Next, the setup handler loads the main binary into the ITCM memory, configures the clock for the UART interfaces, enables the FPU, and then branches into the reset handler in ITCM. After this the main binary has control of the system.

Since the setup handler uses address 0x60000000 as the base address for the text section it is unable to be compiled alongside the main binary which expects 0x0 as the start of the text section. To accommodate for this a script is used to merge the two binaries together after they have been compiled. This script also updates values in the boot structure providing information about where the main binary is positioned in flash and how large it is, both values are used by the setup handler when copying the main binary into RAM.

3.5 Debugging

Debugging is an core part of any type of software development. Since this thesis is concerned with a compiler toolchain and an operating system, the amount of tooling available is very limited. One common way of debugging applications is to use GDB. GDB enables stepping through code, inspecting variables and the state of the CPU. However, in order to use GDB a debugger must be attached to the target device. This is not immediately possible for the Teensy without some workarounds. Additionally, without debugging symbols emitted by the toolchain, the features of GDB is very limited. In this case, the Inferno toolchain does not provide support for any of the debug symbols formats supported by GDB. Because of this, the majority of the debugging done throughout this project is done through a combination of assembler inspection, print statements, and verbose exception handlers.

Assembler inspecting is performed by using arm-none-eabi-objdump from GNU binutils [20] to dump the binary in UAL assembler format. The objdump tool highlights obvious errors as binary sequences that are invalid will be output as unknown instructions. Beyond this the tool requires a lot of manual inspection of the assembler in order to spot errors. Combining objdump with small test programs intended to hit the specific parts of the loader that are being tested will in most cases highlight errors. In some cases it is also useful to define the assembler file as well if there are very specific patterns being tested.

The usage of objdump is not limited to only the loader. By implementing detailed logging in the Inferno source the path the program takes becomes apparent. When the kernel fails, either because of incorrect values or because it performed an illegal action, printing the address of the offending procedure can be used alongside objdump to inspect the specific procedure. In many cases, there are apparent discrepancies between the C source and the produced assembler. In these cases, copying the procedure to a standalone file with verbose logging from the loader provides the necessary into where the loader fails to remedy the issue. In some rare cases, the issue is caused by incorrect assignment in earlier procedures cause the issue. For these types of issues inspection of the error clause, for instance a branch that was or was not taken is used as the beginning and the

variable is tracked to every mutation and reviewed to see if the mutation was performed incorrectly. Once the incorrect mutation is found, objdump is used again to inspect the invalid assignment.

Some incorrect instructions or bad values will result in the processor attempting to perform an illegal instruction. In these cases it is very convenient to review the exception that was thrown as they provide a good base understanding for what is likely causing the problem. For this reason, all non-hardware dependent exceptions have registered handlers that log the type of exception and dump the values in the registers related to the exception. These are especially practical when there is a branch to a ARM32 instruction.

Chapter 4

Results

This thesis is concerned with multiple goals, the first being to produce a new backend for the Inferno toolchain to produce ARMv7-M compliant binaries with the Thumb instruction set. The second is to produce a port of Inferno running on the Teensy41. The third is to get the Inferno OS to a state where it can be interacted with through an UART interface. Alongside these the evaluation of the size requirements of the system is central due to the memory constraints of microcontrollers that use the ARMv7-M architecture.

4.1 State of the toolchain

The new toolchain is able to compile a functional version of the kernel. It has support for essential data handling and memory operations, with some alternate variants that enable the loader to produce fewer instructions overall. It has support for arithmetic and logic operations although not all variants are supported, and floating-point is not supported to any degree. It supports control flow operations with improved call sequences over the original Thumb1 implementation.

The lack of support for floating-point operations is conveyed by the loader by emitting a warning when building the binary. The program will also produce a `USAGEFAULT` exception with `INVSTATE` set should the running binary attempt to call into a procedure that uses floating-point operations.

Additionally, a trivial optimization for updating the stack pointer on procedure enter and return has been implemented highlighting that there are low hanging optimizations that can be implemented to improve the overall binary size.

4.2 Binary size

Figure 4.1 shows the binary size of the kernel compiled with the ARM32, Thumb, and the improved Thumb toolchains. All were built using the same configuration file, seen in the Appendix B, and the same source. From the figure, it is apparent

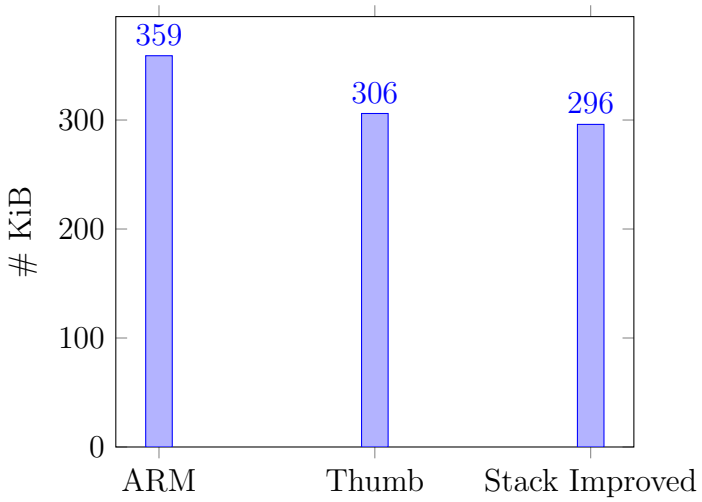


Figure 4.1: Binary sizes

Feature	**Thumb Size**	**ARM32 size**	**Saved(bytes)**	**Diff(%)**
devdup	1256	1560	304	19.5
devpipe	3536	4112	576	14.0
libmath	11808	12352	544	4.4
modmath	66560	63192	-3368	-5.3

Table 4.1: Size comparison of different features

that the Thumb toolchain produces smaller binaries. For the base case shown here, the improved Thumb compiler reduces the binary size by 18% over ARM32. While all the included modules are compiled without errors, it is possible that some of the included source code is incorrectly compiled and the size might change slightly in order to remedy any issues found. However, it is unlikely that this will result in any significant size change.

The improved version references the improved save and restore stack sequence used at procedure entry and exit. This simple improvement reduced the binary size by 10KiB or 4% from the base Thumb output.

Table 4.1 compares the size of different Inferno features built for ARM32 and Thumb. The table shows that, as expected, the size of features will likely reduce when built for the Thumb instructions set. Additionally, it also shows that using Thumb is not guaranteed to improve the size. Exactly what causes `modmath` to become larger for Thumb than in ARM32 is unclear. The most likely reason is that math operations often have many local variables. Since only a subset of the registers are allocated by the compiler it has to store variables on the stack resulting in added instructions to store and load these variables. The reason only four features are shown in the table above is because it is impractical to remove and add features due to dependencies resulting in errors if core features are removed, and libraries that require hardware-specific implementations.

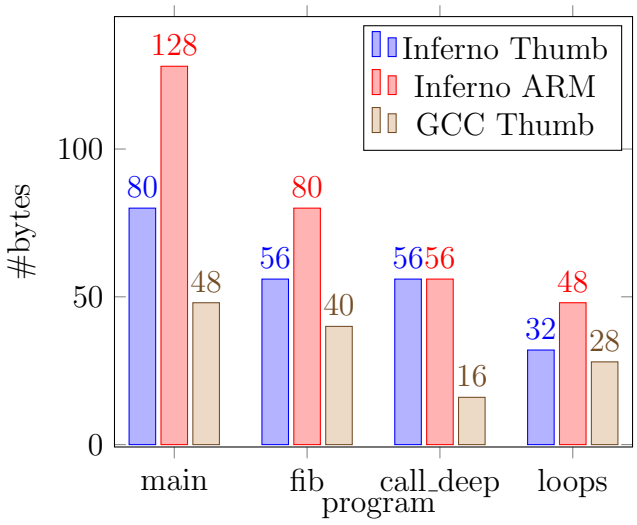


Figure 4.2: Size of binary with different compilers

Figure 4.2 shows the same test programs as in the background built with Inferno Thumb, Inferno ARM32 and GCC Thumb. There is little change in the size of these programs, the only thing of note is that `call_deep` is now equal in size for ARM32 and Thumb after the save/restore stack improvement.

4.3 State of the OS

Currently Inferno OS is not usable with the Dis VM. While the OS boots successfully it expects that userspace is launched through the Dis VM. Since the dis VM is not yet ported to ARMv7-M userspace is also not operable. It is possible, instead of running the Virtual Machine, to go into applications that run natively on the hardware. However, this removes the envisioned benefits of using a managed operating system and removes the ability to use most of the libraries and tools that ship with Inferno. Alternatively another virtual machine can be used.

Additionally, a minimal port of the i.MX RT1060 boot sequence has been ported to the Inferno toolchain. Along with some external tooling, it is capable of dynamically loading the main binary into memory, and branching to the main procedure.

Chapter 5

Discussion

5.1 Compiler Choice

Currently Inferno employs its own toolchain, while this works just fine it introduces some complications when porting between architectures as demonstrated in this work. Additionally since the Inferno toolchain is far from being a mainstream compiler it is not afforded the benefits of constant updates to add improved features. Toolchains such as GCC and LLVM have huge communities involved in their development and as a result they provide more features. Additionally, since these toolchains are pretty much the standard across the industry it is easier to attract new developers without having to teach them the quirks of Inferno's toolchain and the features it lacks.

The GCC and LLVM toolchains have many features not present in the Inferno toolchain. Consequently, they are quite complex systems that are harder to get into than that of Inferno, the respective binary sizes of the individual tools are 2-3 times larger for the LLVM and GCC toolchains running an x86 targeted at ARM. While this further inhibits deploying the compilers onto a embedded board to enable compilation on the board, the Inferno toolchain alone exceeds 1MiB. Therefore, without alternate approaches such as execute in place, it is not possible to use either for on-device compilation of C. Since the intended application of the ARMv7-M Inferno port is in IoT applications it is also unlikely that the ability to compile C code on the device itself is necessary. Therefore the port would benefit more from the advanced features of GCC and LLVM than it gains from having a slightly smaller, less complex toolchain.

If the ability to compile C code on the device is strictly necessary then toolchains such as TCC [21] might be a decent middle ground solution. Offering a more adopted compiler without drastically increasing the memory required in order to host the OS.

If any other toolchain is to be used for building the Inferno kernel, it is not as simple as changing the toolchain and everything will work. First off, all features

that are implemented in assembler will have to be ported to the appropriate assembler language of the new compiler. Secondly, it is very likely that some concepts used by the Inferno toolchain will not translate to other toolchains. Therefore it will be necessary to modify the source as well. This has already been achieved by Harvey OS [22], a port of Plan9 that can be built using C11 compliant toolchains. They achieved this using semantic patching [23].

Another benefit, beyond reducing the size of the binary, is that these compilers support the usage of a linker script. With this it would be possible to use Execute In Place for static libraries, for instance the compiler. A lot of RAM could then be freed up for use by the OS with static resources being stored in flash.

One downside of using the Inferno toolchain is that due to it not supporting features that are commonplace in more commonly used toolchains is that using external libraries might require that they are modified to make them compatible.

5.2 Addressability

While the addressability was not modified in this work, it holds potential for improved binary size and performance since instructions that access registers eight to fifteen are better served by thumb2 instructions. Therefore, if the compiler is made aware of the limitiations of certain instructions it can attempt to store variables that are used a lot in registers 0-7 if they do not require Thumb2 instructions. Additionally, if it knows that a variable requires Thumb2 instructions and there are not enough free registers to store all variables in registers 0-7 it can instead store this specific variable in registers 8-12. This however requires a bit more insight since if the value that is stored in registers 8-12 is accessed often it would be better to store it in lower registers. It becomes a cost analysis of storing and loading from the stack or heap versus using larger instructions.

5.3 Binary Size

The Thumb binary is quite a bit smaller than the ARM32 equivalent. However, it is still larger than that of the GCC thumb output, as was shown in the Results 4.2. In the test cases shown, for all except `call_deep`, GCC outperforms the Inferno Thumb compiler. The main reason is that the Thumb compiler always saves the return address to the stack, even if it does not call into another procedure. For `call_deep` GCC optimizes out all the procedures, though it still keeps the `bx lr` and a few `nop` instructions for the ones that were optimized out.

5.4 Floating Point Handling

As mentioned in Method 3, the loader will change to ARM32 mode if it encounters floating-point operations. This was done since the current configuration of the

kernel does not require floating-point operations to function. Regardless, some alterations were made so that the loader did not promote entire procedures simply because floating-point operations were encountered. While it would have been possible to extend the loader so that instead of producing ARM32 instructions in the case of floating-point usage it could instead produce `noop` or some other type of instruction. This was implemented, however it required that the loader be extended with new accepted immediate formats. While not a problem on its own, this would result in the loader containing untested and very likely incorrect encodings which would make the process of adding these features in the future becoming a cumbersome bug hunt to fix these issues. One of the benefits of the current solution is that since ARMv7-M CPUs do not support changing to ARM32 mode a `USAGFAULT` exception with `INVSTATE` being set in the Usage Fault Status Register. This way it is very apparent when the CPU is calling into a routine with unsupported operations. Along with the log from the loader informing of loaded procedures that have been emitted as ARM32 it is relatively easy to find the offending procedure and modify it as needed.

Floating-point support was not added because it went beyond the scope of this thesis. ARMv7-M however, does provide optional an floating-point extension. Indeed, the i.MX RT1060 supports floating-point operations, therefore adding support for this at a later date should not pose a problem.

5.5 Boot Handler

As mentioned in 3.4, the Teensy41 uses a configuration structure and a setup handler when booting. The implemented boot setup was ported from the Arduino SDK for the Teensy41 [19] and a custom script was used to merge the two binaries into one before flashing the device. Since the boot binary is not linked with the main binary it could technically be compiled using GCC which would enable the utilization of the existing boot handler in the SDK which is more feature full than the one ported to Inferno. This way, it would be possible to use the linker script functionality of GCC to embed the main binary and therefore remove the need for the merge script since the relevant values can be extracted from the main binary since it includes the standard ARMv7-M vector table.

5.6 Virtual Machine

At the moment the Dis VM has not been ported to Thumb, and therefore the OS is not usable. Porting the VM is a non-trivial task. Instead of porting Dis VM it might instead be more beneficial to implement WebAssembly (WASM) [24] support for Inferno. WASM is a standard for portable bytecode which can be produced from several source languages. It has risen in popularity in resent years, with support for WASM in all major webbrowsers. WASM would expand upon the amount of libraries and tools available since the ecosystem is much larger

than that of Dis/Limbo. WASM also has a lower barrier of entry since different high level languages can be compiled to WASM, this makes it possible for many contributors from different fields to participate in growing the ecosystem.

One large issue with swapping the virtual machine used for the OS is that all the libraries currently written in Limbo would no longer be compatible. A lot of the functionality provided by these is already available for WASM, however wrapper libraries that map with the system calls of Inferno would will need to be created in order to fully leverage a new virtual machine in Inferno. One possibility however, is to implement a WASM backend for the Limbo compiler.

5.7 Space Saving

Other methods of reducing the overall memory requirements of the system can be applied. One of those is eXecute In Place(XIP) which allows running code from ROM. This reduces the amount of RAM required as most of the application code of the kernel does not require modification. XIP however, introduces some requirements to the hardware which might not always be available. In the case of the i.MX RT1060, the MCU used on the teensy41 board, the interface to flash is handled by FlexSPI. This interface always supports XiP, however it requires that the device is configured properly with the variables that can be provided in the boot structure. These parameters have yet to be explored as they appear quite involved when reading through the reference manual for the MCU, and since the MCU already has 1MiB of RAM available, this feature is more interesting on devices that are even more memory constrained.

For other devices that do not support the use of XiP, another possibility could be to dynamically load and unload Dis binaries as they are needed. As of now, the binaries are embedded directly into the main binary along with the filesystem, and consequently they are also loaded into RAM along with the kernel. If instead they were loaded into RAM when used and freed afterwards the maximum RAM usage at any one point would be reduced.

5.8 Edge computing

The envisioned market for a distributed operating system is running on small embedded devices which typically have a lot of idle time. Because interacting with other devices is very simple in Inferno, one possible benefit of it is to use this free time in order to work on some other task. The specifics of the task are not of concern, although it would have to be tasks that can be easily separated into smaller chunks of work.

Chapter 6

Conclusion & Future Work

The size of the kernel, with the used configuration, has seen a size reduction of approximately 18% from switching from ARM32 to the Thumb instruction set. It is now possible to build and run the kernel on microcontroller-class devices, instead of only larger CPUs. While the size is still too large for many devices running Cortex-M CPUs there are size optimizations that can be added to the Inferno toolchain to further reduce the binary size. Without the usage of execute-in-place or other size optimizations, few Cortex-M devices will be able to run the current version of Inferno.

From this work the Inferno compiler toolchain has been successfully ported to ARMv7-M with some restrictions. The benefit of this is that it opens a new market for deploying Inferno applications although the Dis VM will also have to be ported. While the port has been produced, it is noted that Inferno would benefit from an upgrade in the toolchain. There is no explicit reason for Inferno needing a custom C toolchain when there are already many better and well maintained toolchains that can be used free of charge. This sentiment is also reflected in the virtual machine used by Inferno, the Dis VM programmed with Limbo is a niche VM and by replacing it with a more modern alternative for instance WASM would open for greater adaptation of the OS, not only for Cortex-M devices but also for all other platforms supported by Inferno.

6.1 Future work

As has been mentioned several times through this thesis, the Dis VM is not yet ported to the Thumb instruction set. It has also been mentioned that alternative Virtual Machines should be considered. In my view the most interesting alternative is WASM as this enables many different types of developers to contribute with libraries since many different languages can be used to develop WASM applications. There are also different WASM runtimes available such as wasm3 [25] and wasmtime [26], a list of WASM runtimes can be found here [27]. The different runtimes present different features and have different requirements. Developers

are therefore able to choose the runtime that best matches their needs. In the case of this project an interpreter(wasm3) or ahead-of-time runtime would reduce size of the runtime. For processors with fewer restrictions runtimes with Just-In-Time optimizations such as wasmtime enable better performance for applications. Any future additions to this work should consider if compliance with the current Inferno standard is more important than the possible benefits of replacing the Dis VM and Limbo. It would also be possible to reap the benefit of the existing Limbo modules by implementing a new WASM backend for the Limbo compiler.

Regarding the toolchain used to build the kernel, either the current one should be expanded to add more space saving optimizations or the source should be ported to be compatible with another toolchain. Since there are many features missing in the toolchain that are convenient when developing for embedded devices and the fact that there are few people invested in maintaining the current toolchain, it would most likely be better to invest into porting the kernel to be compatible with a better maintained toolchain.

If the Inferno toolchain is used in future works it should be extended to support floating-point instructions. Once this has been implemented the loader should be restructured, removing the ARM32 part and properly separating the Thumb implementation into the file structure used by the other loaders in Inferno. At the moment almost all Thumb related features are defined in a single file which makes it somewhat unorganized.

Bibliography

[1] Vita Nuova, "Inferno OS Overview," 2022. https://www.inferno-os.org/inferno/.

[2] ARM, "ARM sales 4th quarter 2019," 2020. https://www.arm.com/company/news/2020/02/record-shipments-of-Arm-based-chips-in-previous-quarter.

[3] P. Hughes, "ARM sales 4th quarter 2020," 2021. https://www.arm.com/company/news/2021/02/arm-ecosystem-ships-record-6-billion-arm-based-chips-in-a-single-quarter.

[4] P. Hughes, "Arm sales 2021," 2022. https://www.arm.com/company/news/2022/05/arm-delivers-record-revenues-and-record-profits-in-fy21.

[5] "Introduction to the inferno file protocol, styx," 2022. http://man.cat-v.org/inferno/5/intro.

[6] PJRC, "Teensy41," 2022. https://www.pjrc.com/store/teensy41.html.

[7] NXP semiconductors, "i.mx rt1060 product overview," 2022. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/i-mx-rt-crossover-mcus/i-mx-rt1060-crossover-mcu-with-arm-cortex-m7-core:i.MX-RT1060.

[8] "ARM website," 2022. https://www.arm.com/.

[9] K. Thompson, "Plan 9 C compilers," 1990. Accessed on 15.06.2022.

[10] ARM, "Unified assembler language," 2011.

[11] Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard Trickey, Phil Winterbottom, "The Inferno Operating System," 1997.

[12] Plan 9 Foundation, "Plan9," 2022. https://plan9.io/plan9/about.html.

[13] "9p protocol manual," 2022. http://man.cat-v.org/plan_9/5/intro.

[14] R. Pike and D. M. Ritchie, "The styx architecture for distributed systems," *Bell Labs Technical Journal, Vol. 4, No. 2*, 1999.

[15] D. M. Ritchie and Vita Nuova, "Limbo language," 2005.

[16] Lucent Technologies Inc and Vita Nuova Limited, "Dis Virtual Machine Specification," 2000.

[17] NXP semiconductors, *i.MX RT1060 Processor Reference Manual Rev. 3*, 2021. https://www.pjrc.com/teensy/IMXRT1060RM_rev3.pdf.

[18] "qemu," 2022. https://www.qemu.org/.

[19] "Teensyduino," 2022. https://www.pjrc.com/teensy/td_download.html.

[20] "GNU binutils," 2022. https://www.gnu.org/software/binutils/.

[21] F. Bellard, "Tiny C Compiler," 2022. https://bellard.org/tcc/.

[22] "Harvey OS," 2022. https://github.com/Harvey-OS/harvey/tree/GPL-C11.

[23] J. Lawall, "Program Manipulation of C Code: From Partial Evaluation to Semantic Patches for the Linux Kernel," 2021. https://popl21.sigplan.org/details/pepm-2021-papers/11/Program-Manipulation-of-C-Code-From-Partial-Evaluation-to-Semantic-Patches-for-

[24] "Webassembly," 2022. https://webassembly.org/.

[25] "Wasm3," 2022. https://github.com/wasm3/wasm3.

[26] "wasmtime," 2022. https://wasmtime.dev/.

[27] S. Akinyemi, "WASM runtimes," 2022. https://github.com/appcypher/awesome-wasm-runtimes.

Appendix A

Sample programs

I Call Deep

A simple program highlighting the cost of calling procedures.

```
void func3(void) {
    int a = 4;
}

void func2(void) {
    func3();
}

void func1(void) {
    func2();
}

int main(void) {
  func1();
  return 1;
}
```

II Fib

A simple Fibonacci implementation.

```
int fib(int n) {
  int f = 1;

  int prev = 0;
  int tmp;
  for (int i = 1; i < n; i++) {
    tmp = f;
```

```
    f = prev + f;
    prev = tmp;
  }
  return f;
}

int main() { return fib(4); }
```

III Loops

A simple program highlighting the cost of loops. Note volatile is used to prevent GCC from optimizing out the loop.

```
int main(void) {
  volatile int b = 0;
  for (int i = 0; i < 1000; i++) {
    b += i;
  }

  return 0;
}
```

IV Simple program

A simple program with some common operations.

```
int fib(int n);

int main(void) {
  int a = 2 + 2;

  if (a == 4) {
    a = 6;
  }

  int b = 6;

  fib(4);

  return a;
}

int max(int a, int b) {
  if (a > b) {
```

```
    return a;
  } else {
    return b;
  }
}

int fib(int n) {
  int f = 1;

  int prev = 0;
  int tmp;
  for (int i = 1; i < n; i++) {
    tmp = f;
    f = prev + f;
    prev = tmp;
  }
  return f;
}
```

Appendix B

Configuration

```
dev
        root
        cons archteensy noscreen not
        env
        mnt
        pipe
        prog
        srv
        uart
        dup

ip

lib
        interp
        math
        kern
        sec

mod
        sys

port
        alarm
        alloc
        allocb
        chan
        dev
        dial
        dis
        discall
```

```
        exception
        exportfs
        inferno
        latin1
        nocache
        nodynld
        parse
        pgrp
        print
        proc
        qio
        qlock
        random
        sysfile
        taslock
        xalloc

link

code
        int main_pool_pcnt = 40;
        int heap_pool_pcnt = 40;
        int image_pool_pcnt = 0;
        int cflag = 0;

        int consoleprint = 1;
        int redirectconsole = 1;
        char debug_keys = 1;
        int panicreset = 0;
        Type *Trdchan;
        Type *Twrchan;

init
        evalinit

root
        /chan /
        /dev /
        /dis /
        /dis/sh.dis
        /net /
        /prog    /
        /osinit.dis
        /n/remote /
        /lib     /
```

```
/env      /
/fd                 /
/n      /
/tmp      /
/dis/lib
/dis/disk
```

Appendix C

Configuration IP protocols

- arp
- bootp
- dhcp
- esp
- ethermedium
- gre
- icmp
- icmp6
- igmp
- il
- ip
- ipv6
- ppp
- rudp
- tcp
- udp

Appendix D

Configuration Libraries

- libbio
- libdraw
- libdynld
- libfreetype
- libinterp
- libkern
- libkeyring
- liblogfs
- libmath
- libmemdraw
- libmemlayer
- libmp
- libnandfs
- libprefab
- libsec
- libtk

Appendix E

Added Instructions

- Usecase: Store the PC or SP to a memory address
  UAL syntax:

  ```
  mov PC/SP, rtmp
  str PC/SP, [rtmp, offset]
  ```

  Encodings:

  – MOV(register) T1 + STR(immediate) T1

  Optab entries:

  ```
  {AMOVW, C_SP, C_NONE, C_SOREG, 60, 4, 0},
  {AMOVW, C_PC, C_NONE, C_SOREG, 60, 4, 0},
  ```

  The reason it consists of both a move and a store is that using the PC as the source register causes unpredictable behaviour.

- Usecase: Load the PC or SP from a memory address
  UAL syntax:

  ```
  ldr rtmp, [rx, offset]
  mov rtmp, PC/SP
  ```

  Optab entries:

  ```
  {AMOVW, C_SOREG, C_NONE, C_PC, 61, 4, 0},
  {AMOVW, C_SOREG, C_NONE, C_SP, 61, 4, 0},
  ```

  Encodings:

  – LDR(immediate) T1 + MOV(register) T1

  Same reasoning here for using two separate instructions as the case above

is there really no ideal way of storing the PC in one go?

- Usecase: Read the current PRIMASK value
  UAL syntax:

  ```
  mrs rd, PRIMASK
  ```

  Optab entry:

  ```
  {AMOVW, C_PSR, C_NONE, C_REG, 62, 4, 0},
  ```

  Encodings:

  – MRS T1

  Currently there is only support for PRIMASK as it was the only one required. Extending it would be simple, although it also requires that the assembler is extended because the compiler would not generate this sequence from the C source code.

- Usecase: Set the value of PRIMASK
  UAL syntax:

  ```
  msr PRIMASK, rd
  ```

  Optab entry:

  ```
  {AMOVW, C_REG, C_NONE, C_PSR, 63, 4, 0},
  ```

  Encodings:

  – MSR T1

  Same reasoning as the one above.

- Usecase: 32bit dataprocessing instructions, used for larger immediates and registers higher than seven.
  These are added because they support larger immediate encoding than the Thumb1/narrow equivalent and they support 4bit registers.
  UAL syntax:

  ```
  and rd, rn, #const
  add rd, rn, #const
  orr rd, rn, #const
  ```

  Optab entries:

```
{AMVN, C_LCON, C_NONE, C_REG, 64, 4, 0}
{AMVN, C_LCON, C_REG, C_REG, 64, 4, 0}
```

Encodings:

- AND (immediate) T1
- ADD (immediate) T3
- ORR (immediate) T1

- Usecase: Store multiple registers at once. Used when copying structs or when assigning many fields to a struct at once.

  UAL syntax:

  ```
  STMIA rn, <registers>
  STMDB rn, <registers>
  ```

  Optab entries:

  ```
  {AMOVM, C_LCON, C_NONE, C_REG, 66, 4, 0},
  {AMOVM, C_REG, C_NONE, C_LCON, 76, 4, 0},
  ```

  Encodings:

  - STMIA T2
  - STMDB T1

- Usecase: Store and load multiple registers at once. Used when copying structs or when assigning many fields to a struct at once.

  UAL syntax:

  ```
  LDMIA rn, <registers>
  LDMDB rn, <registers>
  ```

  Optab entries:

  ```
  {AMOVM, C_REG, C_NONE, C_LCON, 76, 4, 0},
  ```

Encodings:

- LDMIA T2
- LDMDB T1

- Usecase: Not used yet, will be used by userspace to trap into kernel procedures.
  UAL syntax:

```
svc #imm8
```

  Optab entries:

```
{ASWI, C_NONE, C_NONE, C_NONE, 68, 2, 0},
{ASWI, C_NONE, C_NONE, C_LOREG, 68, 2, 0},
```

  Encodings:

  - SVC T1

- usecase: Used by the manual div and modulo implementation.
  UAL syntax:

```
RSB.W rd, rn, #const
```

  Optab entries:

```
{ARSB, C_LCON, C_REG, C_HREG, 69, 4, 0},
{ARSB, C_LCON, C_REG, C_REG, 69, 4, 0},

{ARSB, C_SAUTO, C_REG, C_HREG, 69, 4, 0},
{ARSB, C_SAUTO, C_REG, C_REG, 69, 4, 0},
```

  Encoding:

  - RSB (immediate) T2

- Usecase: used to load from memory using a register as an offset. The versions with support for writeback are used in the improved stack save/restore implementation.

  UAL syntax:

```
ldr rt, [rn, #+/-imm8]
ldr rt, [rn, #+/-imm8]!
ldr rt, [rn] #+/-imm8
ldr rt, [rn, imm12]
```

Optab entries

```
{AMOVW, C_SAUTO, C_NONE, C_HREG, 70, 4, 0},
```

Encodings:

- ldr (immediate) T3
- ldr (immediate) T4

- usecase: Perform logical AND operation on a shifted register to update conditional flags.

  UAL syntax:

  ```
  TST.w rn, rm, shift
  ```

  Optab entries:

  ```
  {AMOVW, C_SAUTO, C_NONE, C_HREG, 70, 4, 0}
  ```

  Encodings:

  - TST (register) T2

- Usecase: Shift register by register value, added to enable shifting on registers higher than 7
  UAL syntax:

  ```
  LSR.W rd, rn, rm
  LSL.W rd, rn, rm
  ```

  Encodings:

  - LSR (register) T2
  - LSL (register) T2

Optab Entries:

```
{AMOVW, C_SHIFT, C_NONE, C_HREG, 74, 4, 0},
{AMOVW, C_SHIFT, C_NONE, C_REG, 74, 4, 0},
```

- Usecase: Enable usage of high registers for AND, ORR, and EOR
  UAL syntax:

```
ORR.W rd, rn, rm
AND.W rd, rn, rm
EOR.W rd, rn, rm
```

  Encodings:

  – ORR (register) T2
  – AND (register) T2
  – EOR (register) T2

  Optab entries:

```
{AMVN, C_REG, C_NONE, C_HREG, 75, 4, 0},
```

- Usecase: store with writeback and post/pre-increment
  UAL syntax:

```
STR rt [rn, #-imm8]
STR rt [rn], #+/-imm8]
STR rt [rn, #-imm8]!
```

  Encodings:

  – STR (immediate) T4

  Optab entries:

```
{AMOVW, C_HREG, C_NONE, C_LAUTO, 78, 4, 0},
```

- Usecase: Store the PC or SP to a memory address
  UAL syntax:

```
mov PC/SP, rtmp
str PC/SP, [rtmp, offset]
```

Encodings:

- MOV(register) T1 + STR(immediate) T1

Optab entries:

```
{AMOVW, C_SP, C_NONE, C_SOREG, 60, 4, 0},
{AMOVW, C_PC, C_NONE, C_SOREG, 60, 4, 0},
```

The reason it consists of both a move and a store is that using the PC as the source register causes unpredictable behaviour.

is there really no ideal way of storing the PC in one go?

- Usecase: Load the PC or SP from a memory address
  UAL syntax:

```
ldr rtmp, [rx, offset]
mov rtmp, PC/SP
```

Optab entries:

```
{AMOVW, C_SOREG, C_NONE, C_PC, 61, 4, 0},
{AMOVW, C_SOREG, C_NONE, C_SP, 61, 4, 0},
```

Encodings:

- LDR(immediate) T1 + MOV(register) T1

Same reasoning here for using two separate instructions as the case above

- Usecase: Read the current PRIMASK value
  UAL syntax:

```
mrs rd, PRIMASK
```

Optab entry:

```
{AMOVW, C_PSR, C_NONE, C_REG, 62, 4, 0},
```

Encodings:

- MRS T1

Currently there is only support for PRIMASK as it was the only one required. Extending it would be simple, although it also requires that the assembler is extended because the compiler would not generate this sequence from the C source code.

- Usecase: Set the value of PRIMASK
  UAL syntax:

  ```
  msr PRIMASK, rd
  ```

  Optab entry:

  ```
  {AMOVW, C_REG, C_NONE, C_PSR, 63, 4, 0},
  ```

  Encodings:

  – MSR T1

  Same reasoning as the one above.

- Usecase: 32bit dataprocessing instructions, used for larger immediates and registers higher than seven.
  These are added because they support larger immediate encoding than the Thumb1/narrow equivalent and they support 4bit registers.
  UAL syntax:

  ```
  and rd, rn, #const
  add rd, rn, #const
  orr rd, rn, #const
  ```

  Optab entries:

  ```
  {AMVN, C_LCON, C_NONE, C_REG, 64, 4, 0}
  {AMVN, C_LCON, C_REG, C_REG, 64, 4, 0}
  ```

  Encodings:

  – AND (immediate) T1
  – ADD (immediate) T3
  – ORR (immediate) T1

- Usecase: Store multiple registers at once. Used when copying structs or when assigning many fields to a struct at once.

  UAL syntax:

  ```
  STMIA rn, <registers>
  STMDB rn, <registers>
  ```

Optab entries:

```
{AMOVM, C_LCON, C_NONE, C_REG, 66, 4, 0},
{AMOVM, C_REG, C_NONE, C_LCON, 76, 4, 0},
```

Encodings:

- STMIA T2
- STMDB T1

- Usecase: Store and load multiple registers at once. Used when copying structs or when assigning many fields to a struct at once.

  UAL syntax:

  ```
  LDMIA rn, <registers>
  LDMDB rn, <registers>
  ```

  Optab entries:

  ```
  {AMOVM, C_REG, C_NONE, C_LCON, 76, 4, 0},
  ```

  Encodings:

  - LDMIA T2
  - LDMDB T1

- Usecase: Not used yet, will be used by userspace to trap into kernel procedures.
  UAL syntax:

  ```
  svc #imm8
  ```

  Optab entries:

  ```
  {ASWI, C_NONE, C_NONE, C_NONE, 68, 2, 0},
  {ASWI, C_NONE, C_NONE, C_LOREG, 68, 2, 0},
  ```

  Encodings:

  - SVC T1

- usecase: Used by the manual div and modulo implementation.
  UAL syntax:

```
RSB.W rd, rn, #const
```

  Optab entries:

```
{ARSB, C_LCON, C_REG, C_HREG, 69, 4, 0},
{ARSB, C_LCON, C_REG, C_REG, 69, 4, 0},

{ARSB, C_SAUTO, C_REG, C_HREG, 69, 4, 0},
{ARSB, C_SAUTO, C_REG, C_REG, 69, 4, 0},
```

  Encoding:

  – RSB (immediate) T2

- Usecase: used to load from memory using a register as an offset. The versions with support for writeback are used in the improved stack save/restore implementation.

  UAL syntax:

```
ldr rt, [rn, #+/-imm8]
ldr rt, [rn, #+/-imm8]!
ldr rt, [rn] #+/-imm8
ldr rt, [rn, imm12]
```

  Optab entries

```
{AMOVW, C_SAUTO, C_NONE, C_HREG, 70, 4, 0},
```

  Encodings:

  – ldr (immediate) T3
  – ldr (immediate) T4

- usecase: Perform logical AND operation on a shifted register to update conditional flags.

  UAL syntax:

```
TST.w rn, rm, shift
```

Optab entries:

```
{AMOVW, C_SAUTO, C_NONE, C_HREG, 70, 4, 0}
```

Encodings:

- TST (register) T2

- Usecase: Shift register by register value, added to enable shifting on registers higher than 7
  UAL syntax:

```
LSR.W rd, rn, rm
LSL.W rd, rn, rm
```

  Encodings:

  - LSR (register) T2
  - LSL (register) T2

  Optab Entries:

```
{AMOVW, C_SHIFT, C_NONE, C_HREG, 74, 4, 0},
{AMOVW, C_SHIFT, C_NONE, C_REG, 74, 4, 0},
```

- Usecase: Enable usage of high registers for AND, ORR, and EOR
  UAL syntax:

```
ORR.W rd, rn, rm
AND.W rd, rn, rm
EOR.W rd, rn, rm
```

  Encodings:

  - ORR (register) T2
  - AND (register) T2
  - EOR (register) T2

  Optab entries:

```
{AMVN, C_REG, C_NONE, C_HREG, 75, 4, 0},
```

- Usecase: store with writeback and post/pre-increment
  UAL syntax:

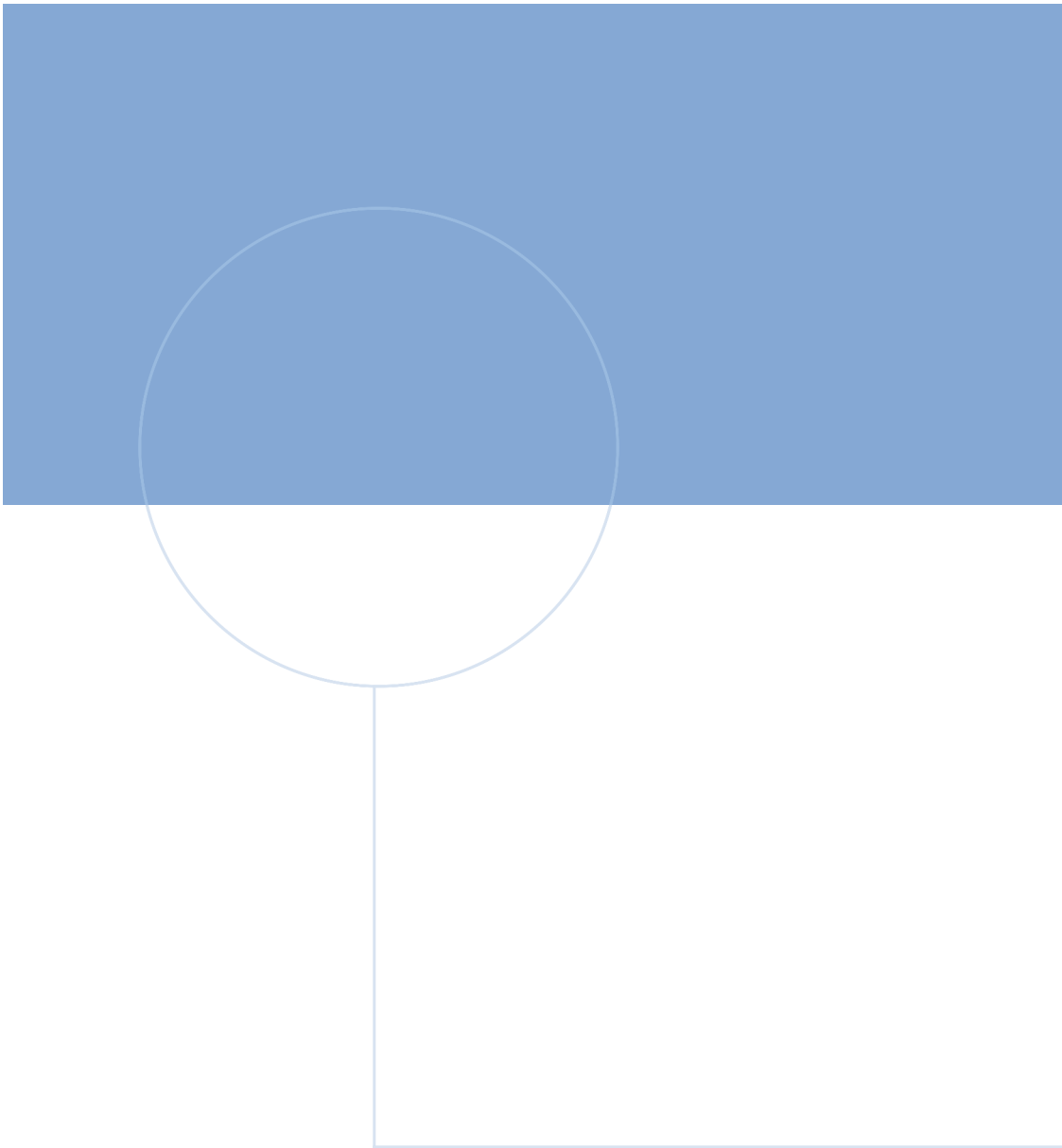
  ```
  STR rt [rn, #-imm8]
  STR rt [rn], #+/-imm8]
  STR rt [rn, #-imm8]!
  ```

  Encodings:

  – STR (immediate) T4

  Optab entries:

  ```
  {AMOVW, C_HREG, C_NONE, C_LAUTO, 78, 4, 0},
  ```



NTNU

Norwegian University of
Science and Technology