



# A Dynamically Recompiling ARM Emulator

3<sup>rd</sup> year project report 2000-2001  
University of Warwick

Written by  
David Sharp

Supervised by  
Graham Martin



## Abstract

Dynamically recompiling from one machine code to another can be used to emulate modern microprocessors at realistic speeds. This report is a discussion of the techniques used in implementing a dynamically recompiling emulator of an ARM processor for use in an emulation of a complete computer system.

## Keywords

Emulation, Dynamic Recompilation, Binary Translation, Just-In-Time compilation, Virtual Machine, Microprocessor Simulation, Intermediate Code, ARM.

# Contents

<b>1. Introduction</b>	<b>7</b>
1.1. What is emulation?	7
1.2. Applications of emulation	7
1.3. Processor emulation techniques	9
1.4. This Project	13
<b>2. Analysis</b>	<b>16</b>
2.1. Introduction to the ARM	16
2.2. Identifying the problem	18
2.3. Getting started	20
2.4. The System Design	21
<b>3. Disassembler</b>	<b>23</b>
3.1. Purpose	23
3.2. ARM decoding	23
3.3. Design	24
<b>4. Interpreter</b>	<b>26</b>
4.1. Purpose	26
4.2. The problem with JIT	26
4.3. The HotSpot™ alternative	26
4.4. Quantifying the JIT problem	27
4.5. Faster decoding	28
4.6. Interfaces	29
4.7. The emulation loop	30
4.8. Implementation	31
4.9. Debugging	32
4.10. Compatibility	33
<b>5. Recompilation</b>	<b>35</b>
5.1. Overview	35
5.2. Methods of generating native code	35
5.3. The use of intermediate code	36
<b>6. Armlets – An Intermediate Code</b>	<b>38</b>
6.1. Purpose	38
6.2. The ‘explicit-implicit problem’	38
6.3. Options	39
6.4. Characteristics of Armlets	40
6.5. The Program Counter	42
<b>7. Profiler</b>	<b>43</b>
7.1. Purpose	43
7.2. Characteristics of a chunk	43
7.3. Chunk generation	45
7.4. Testing	47
7.5. Unrecompilable code	48

<b>8. Code Optimisation</b>	<b>53</b>
8.1. The optimiser	53
8.2. The nature of the source code	53
8.3. The requirements of optimisation	54
8.4. Traditional compiler optimisations	54
8.5. Java run-time optimisation techniques	56
8.6. Conditional Blocks	56
8.7. Redundant condition flag calculation elimination	58
<b>9. Native Code Generator</b>	<b>60</b>
9.1. Overview	60
9.2. Instruction selection	60
9.3. Register allocation options	62
9.4. Dynamic register allocation	63
9.5. Condition flag calculations	64
9.6. Control Flow	66
9.7. Constants	67
9.8. Emitting machine code	69
9.9. Invoking native code	70
9.10. Debugging	70
<b>10. Conclusion</b>	<b>71</b>
10.1. Evaluation	71
10.2. Future extensions	75
10.3. Skills learned	75
10.4. Final conclusions	76
References	77
Bibliography	83
Acknowledgements	88
Appendix A – Overview of ARM assembly	89
Appendix B – Armlet definition	93
Appendix C – Example code generation	95
Appendix D – Source code	98
Appendix E – Glossary of Terms	99
Appendix F – Advice on attempting similar projects	102

# Index of Figures

Figure 1- Screenshot of the emulator of the Manchester Baby, originally built in 1948	8
Figure 2 - The Fetch-Decode-Execute loop of a processor or interpreting emulator	10
Figure 3 - The dynamic recompilation decision process	11
Figure 4- The execution of threaded code	12
Figure 5 - ARM's semiconductor partners who manufacture processors	16
Figure 6 - The Nintendo Gameboy Advance, one of many portable ARM-based devices.	17
Figure 7 - R15 in 26-bit-PC ARM processors, containing both the PC and PSR	19
Figure 8 - System Object Model	22
Figure 9 - ARM instruction set encoding	23
Figure 10 - The relative average cost per emulation of different emulation methods	27
Figure 11 - The format of the interpreter's instruction decode table for ADD instructions	28
Figure 12 - Example of the interface used to separate ARM and MEMC emulations	29
Figure 13 - The interface between ARM and system emulation	30
Figure 14 - The ARM interpreter loop	31
Figure 15 - Direct Translation of individual instructions	35
Figure 16 - The recompilation steps using an intermediate code	36
Figure 17 - Recompilation without an intermediate representation	37
Figure 18 - Deciding whether a branch is inside or outside the current chunk	44
Figure 19 - The control flow of an if...then statement, showing forward branching	45
Figure 20 - Outline of the decision process concerning whether to end the current chunk	47
Figure 21 - The ARM processor modes and their register banks.	48
Figure 22 - The three stages of dynamic recompilation	53
Figure 23 - Observed execution statistics for conditionally executed instructions.	55
Figure 24 - Permutations of the add armlet and appropriate x86 instructions	60
Figure 25 - Decision tree for x86 instruction selection	61
Figure 26 - Register usage in RISC OS	62
Figure 27 - The ARM and x86 flag similarities	65
Figure 28 - Identifying the x86 address of a given armlet to avoid backpatching	66
Figure 29 - The relationship between armlets in constant evaluation	68
Figure 30 - The RISC OS initialising screen	71
Figure 31 - Screenshot of the !Draw program, written in C, that comes with RISC OS	71
Figure 32- Screenshot showing the RISC OS task manager and about box	72
Figure 33 - Screenshot showing the command line running ARM BASIC, written in ARM assembly	72

# 1. Introduction

## 1.1. What is emulation?

Emulation allows software written for one computer system to be run on another. It does this by simulating the low level behaviour of the emulated hardware. The software's instructions are then followed and the emulator updates its state in the same way as the original machine would have done to its own registers and memory. As a result, the software can be made to run on another system identically to the way it would run on the system it was written for, despite the fact that the two may have different underlying hardware.

## 1.2. Applications of emulation

Emulation is an incredibly useful technique that has many different applications to problems across the field of computer science and the computing industry. These applications range from issues concerning the oldest computers to the very latest technologies. The following sections discuss some of these applications.

### 1.2.1. Backwards Compatibility

Backwards compatibility with older systems (especially when application source code is no longer available) is often only possible through emulation. In business, emulation allows old hardware to be replaced without losing the use of the software that ran on it. In this way, emulation provides a cheap alternative to rewriting software from scratch for the new hardware.

In consumer computer systems, providing emulators for previous models has proved a successful way of migrating a customer base to the new generation of computers. For example, Acorn were able to lure customers to their 32 bit models by providing an emulator of their established BBC Micro range. Additionally Acorn was able to convert customers from other platforms by providing an emulator of an IBM-compatible PC.

### 1.2.2. Portability

Emulation techniques can provide the capability of creating software that can run on any computer system without alteration. This is done by implementing an emulation of an abstract processor or 'virtual machine' (i.e. one that does not already exist) on various platforms. Compilers that convert programs into code for that virtual machine are then developed so that any program created using them can run on any of the systems that have an implementation of the virtual machine.

This concept is at least partially credited for the success for the Pascal language with the introduction of P-code in the late 1970s [1]. More recently, virtual machines have become prominent with the growth of the inherently multi-platform Internet. This has caused great interest in Sun Microsystems' Java language which compiles to a bytecode that runs on a Java Virtual Machine (JVM).

### 1.2.3. Software Development

For some platforms, developing and testing software on the system itself is not always feasible. This might be because the hardware is still being developed, or in the case of embedded or portable systems, it may be expensive or time-consuming to install the software for testing. In conjunction with cross-platform development tools, emulators are commonly used to bridge the gap so that software can be developed quickly and cheaply before being implemented on the real hardware for final testing. Emulators such as SPIM [2], which emulates a MIPS processor, can provide a test environment for software that has features not available on the actual machine, for example with real breakpoints [3].

### 1.2.4. Hardware Development

As the complexity of microprocessors has grown, emulation of new processors has often been used during the design phase. In the development of the ARM processor, emulations of the design were used from an early stage to verify the processor's logic.

In some processors, emulation is built into the hardware for backwards compatibility, for example the 16 bit 65816 processor (used in the Super Nintendo game console) can emulate in hardware the 8 bit 6502 processor (used in the older Nintendo Entertainment System) [4]. Other processors, such as the Crusoe range produced by Transmeta, have made emulation a central part of their design. By using dynamic binary translation techniques at a low level inside the processor, Crusoe processors are able to emulate an x86 processor at a comparable speed to the real thing but using less power [5].

### 1.2.5. Historical preservation

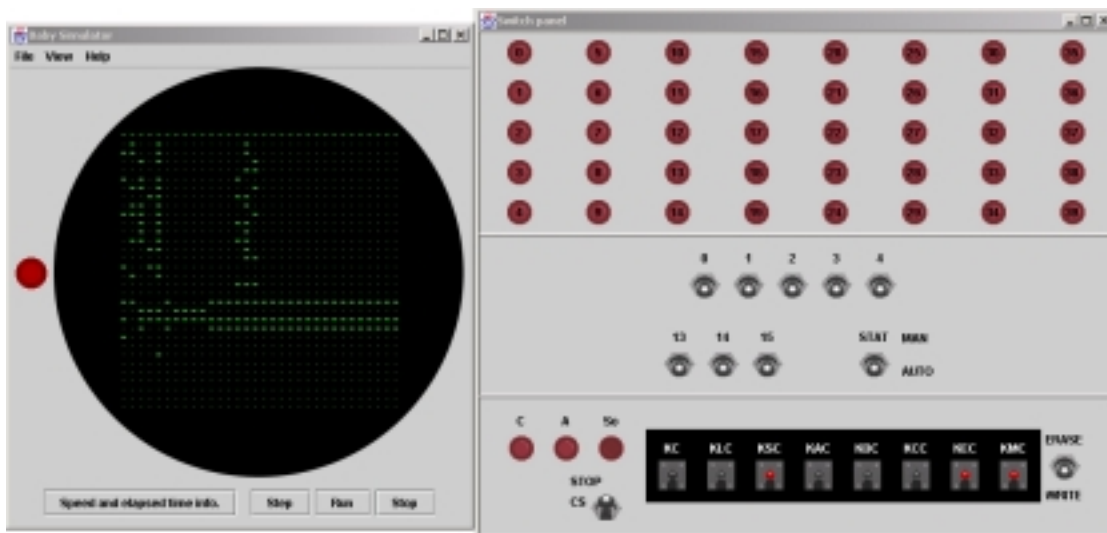


Figure 1- Screenshot of the Manchester Baby emulator, the original was built in 1948

Building replica machines of the first computers from the early 1940s is a time consuming and costly business. In order to preserve the 'feel' of the first computers, historians have developed emulators of these early machines, such as Martin



Campbell-Kelly's Warwick EDSAC Simulator [6] and my own Manchester Baby Simulator [7], shown in Figure 1. Since the original machines have often been destroyed over the years, these emulators continue to provide cheap and widespread access to them.

In recent years, the first home computers of the early 1980s have been superseded by systems more than powerful enough to emulate them. Sentimental attachment to the older computers has motivated many programmers to develop emulators for them on their modern home computers. This has gathered momentum resulting in a wealth of emulators covering almost every consumer computer ever made. Increasingly the emulator authors are striving to emulate the latest technology, resulting in the situation where systems that are still being sold commercially are already available under emulation for free.

## 1.3. Processor emulation techniques

### 1.3.1. Importance of CPU emulation

The emulation of the central processing unit (CPU) is one of the core parts of any emulator in the same way the real CPU is one of the core parts of any computer. To emulate a processor running at 8 MHz, assuming a conservative average of 2 cycles per instruction (which is fair for an ARM2 processor) requires the emulator to emulate around 4 million instructions per second. From this it is clear that the CPU emulation can consume a large fraction of the processing time in any emulator, and therefore that the CPU emulation is extremely important for the emulator's overall performance.

There are several types of CPU emulator: the type chosen depends on the requirements placed on the emulator. The terminology and general concepts of these classifications are derived from concepts in compiler design, with the emulated machine's code being treated as the source language to the interpreter or compiler.

### 1.3.2. Interpreter

Interpreting emulators work in much the same way as the emulated processor's fetch-decode-execute loop, shown in Figure 2. The emulated environment is first initialised to a known start state, which tends to involve setting the emulated registers, memory and interrupt timers to their initial values (as they would be on the real machine). The emulator then proceeds fetching the next instruction from emulated memory. The fetched instruction is then decoded in order to select the emulator's corresponding subroutine. The selected subroutine is then called which updates the emulated environment's registers and memory in the same way as that instruction's execution would have updated the machine's registers and memory. The whole process then repeats.

This method has several advantages:

- It is relatively simple to design and implement.
- It has relatively low memory requirements, since there is no data stored other than the emulated environment.
- There are no problems handling self-modifying code.

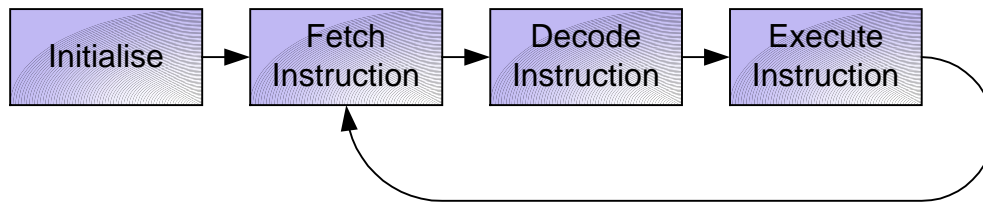


Figure 2 - The Fetch-Decode-Execute loop of a processor or interpreting emulator

Because of these advantages, particularly the simplicity, most emulators are implemented using this method. In emulators for embedded and portable devices, memory constraints can make interpreting the only feasible solution (for example the Java Virtual Machine for the Psion Series 5mx [8]). However, the interpreting method suffers the significant disadvantage of being the slowest approach.

The reason for this is that interpreting emulation is a fairly naïve algorithm that ignores the principle of locality (as utilised in processor caches) [9]. This is an empirical observation that an "[emulated] program spends 90% of its time in 10% of its code". There are three aspects to this principle:

- Temporal locality* – once an instruction is executed it is likely to be executed again soon.
- Spatial locality* – the next instruction is likely to be near the current instruction.
- Sequential locality* – the next instruction is likely to be immediately after the current instruction.

An interpreting emulator only takes into account the current instruction being emulated. It is unable to utilise any previous times the instruction may have been emulated or the instructions emulated before and after it. As a result, the same sequence of instructions in a loop has to be fetched and decoded unnecessarily by the emulator many thousands of times. The problem is summarised by the philosopher George Santayana's comment that, "those who cannot remember the past are condemned to repeat it" [10]. Fortunately other more complex emulation methods exist that are able to take advantage of the principle of locality.

### 1.3.3. Dynamic Recompiler

Dynamic recompilation is one method that attempts to avoid the shortcomings of interpreting emulators. Rather than only looking at the current instruction being executed, dynamic recompilers act on chunks of sequentially emulated instructions. The idea is that by storing information about that chunk the first time it is emulated, using this information will make all of its subsequent emulations faster. This is the same idea used by processor caches in transferring executed code to a faster storage area to benefit subsequent executions.

A dynamic recompiler identifies a section of code to be emulated and checks to see if it has been emulated before. If it has not then the dynamic recompiler generates a

chunk of native machine code (i.e. machine code for the computer that the emulator is running on). This code updates the emulated environment as if those instructions had been emulated. Every successive time that this section of code needs to be emulated, the native machine code chunk is executed again, removing the need for repeated fetching and decoding of emulated instructions. By repeatedly identifying sections of code and executing machine code for them as shown in Figure 3, the emulated environment's state is updated correctly.

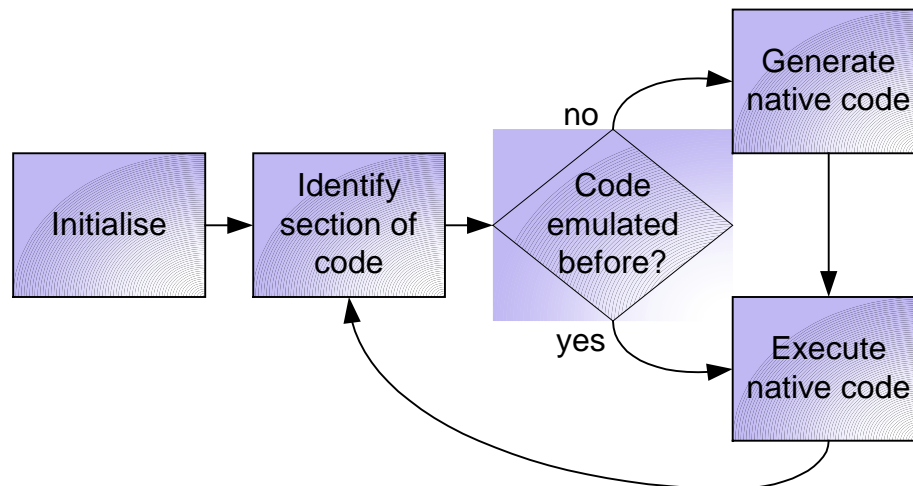


Figure 3 - The dynamic recompilation decision process

Dynamic recompilers have the advantage that they are significantly faster than interpreting emulators. However, they can have large memory overheads as a result of storing many chunks of native code for long periods of time, making them unsuitable for some embedded applications.

#### 1.3.4. Threaded Code

Threaded code [11] works in a similar way to dynamic recompilation. However, rather than generating native machine code, a list of addresses of handwritten subroutines is generated, that performs the emulation for each instruction. The list is stepped through one by one and each subroutine called to emulate the instructions, as shown in Figure 4. Since each handwritten subroutine is fixed during development, they are not as flexible and therefore not as fast as code generated by a dynamic recompiler. However, threaded interpreters do have the advantage that they are easier to implement and can be completely portable to a new platform with no modification as they need not use any machine code [12]. For languages such as Java that execute on a virtual machine which does not lend itself to having its code dynamically generated in this way (because of security requirements), threaded code is a good alternative to an interpreter to improve the performance of an emulator.

Both dynamic recompilers and threaded code can incur problems as a result of caching the information they generate. If the program code at an address in emulated memory changes, whether because a new program is loaded, the memory map changes, or the program is self-modifying, the previously generated code that is

associated with that address will be incorrect. How often this occurs on the emulated system can affect the benefits gained from caching this information.

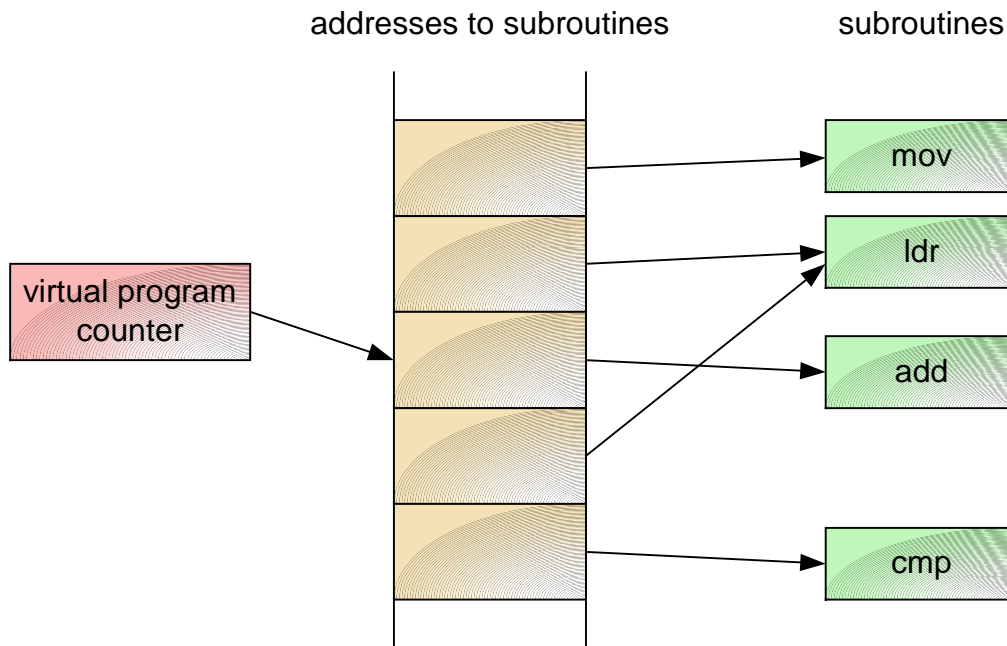


Figure 4- The execution of threaded code

### 1.3.5. Static Recompiler

Rather than recompiling code at run time (during emulation) as is done in a dynamic recompiler, static recompilers generate all the code before hand in the same way as a traditional programming language compiler. This has the benefit that that there is no time-consuming recompilation process when the emulation is running, making the emulation faster. The generated code can also be more optimal than that generated by a dynamic recompiler since better optimisation techniques can be applied that are too time-consuming to use dynamically.

Translating machine code before run-time can have complex problems. An inability to differentiate program code from data is one such problem. This is complicated by calculations that manipulate the value of the program counter. For example, the following section of ARM code shows a jump table that might be compiled from a C switch statement. This code could set the `pc` to a very large range of values depending on the value of `r0` which is difficult to determine at compile time.

```
add    pc,pc,r0,ls1 #2    ; pc = pc + (r0 * 4)
mov    r0,r0              ; nop for pipeline effect
b      case0              ; go to case0 if r0 is 0
b      case1              ; go to case1 if r0 is 1
b      case2              ; go to case2 if r0 is 2
.
.
.
```

Static recompilers alone cannot cope with code modification at run time as they have no way of updating the code that they execute. The only way that they can support this is to additionally implement one of the other three emulation methods to fall back on. As a result, static recompilers are not particularly suited to emulating an entire computer system, where code is frequently changed by loading new programs.

Static recompilation is better suited to the field of binary translation where attempting to convert a single executable program from one platform to another as such code modification is not normally an issue. Multi-platform operating systems such as Windows NT, Solaris and various flavours of Unix, then provide the environment that the program is executed in, rather than fully emulating a computer system as with normal emulation [13].

### 1.3.6. Summary

In summarising this review of the different CPU emulation techniques it is important to emphasise that dynamic recompilation techniques allow the fastest execution of emulated instructions, without the limitations of static recompilation.

## 1.4. This Project

### 1.4.1. Motivations

Most of the hundreds of emulators developed in the past few years emulate systems that are outdated and from the previous generation of computing to the systems being used to emulate them. As a result, the power of the host system dwarfs that of the one being emulated and performance is not a problem.

However, any emulator that attempts to model a more recent system has a significant challenge, as the complexity of modern processors is an order of magnitude greater than earlier systems. As well as being more complex, these emulated processors have clock speeds that are a reasonable proportion of the host system's processor clock speed. In order to emulate such modern systems at a reasonable speed, difficult approaches to emulation such as dynamic recompilation have to be employed.

Several dynamic recompilers have been developed in recent years to meet such demands. Unfortunately almost all have significant limitations. Commercial emulators such as VirtualPC [14], Ardi's Executor [15] and Apple's DR emulator [16], while impressive, tend to limit the information released to brief overviews in white papers (for commercial reasons). Many of the ground-breaking, freely available emulators such as UltraHLE [17], Corn [18] and PSEmu Pro [19] remain closed source and little information is released about their techniques. Other free emulators, that are open source, tend to lack documentation and use primitive techniques (such as direct translation and pre-assembled templates, see section 5.2). Additionally many of these emulators are for video game consoles that do not suffer the same complexities as emulators for computer systems (such as a full operating system, and an ability to load multiple programs).

Developments in Java Virtual Machine technology have been a good source of information on dynamic recompiling methods. However, due to the nature of stack-

based java bytecode relative to register-based machine code, many significant problems have been ignored (such as condition flags, interrupts and exceptions). Academic research on the subject tends towards the problems of binary translation [20], erring towards the complexities of static recompilation and retargetability and away from emulating a full computer system.

As a result of these limitations there is a significant omission in the computing literature concerning emulating real computer systems using dynamic recompilation, something I hope to rectify with this report.

#### 1.4.2. Aim

The aim of this project is to emulate as accurately as possible a modern microprocessor, the ARM, using dynamic recompilation techniques.

#### 1.4.3. Previous work

There are several ARM emulators already in existence that have been developed for various purposes:-

<b>ARMulator</b>	ARM Ltd's own very accurate emulator of most models of ARM processor, designed to be flexible enough to debug developing software or alternatively completely model every aspect of the inner workings of the processor. [21]
<b>ArcEm</b>	The first Acorn Archimedes emulator. This makes use of a modified version of the ARMulator for the ARM emulation. [22]
<b>ARM2</b>	Only able to run simple programs in BBC BASIC as an alternative to the Unix shell. [23]
<b>Archie</b>	The first Acorn Archimedes emulator for MS DOS. [24]
<b>SWARM</b>	Used to model the internal ARM data path to facilitate research into possible modifications. [25]
<b>Red Squirrel</b>	The first Acorn Archimedes emulator for Windows.[26]
<b>ARMphetamine</b>	A dynamically recompiling emulator for a subset of the ARM processor. Unfortunately support for exceptions, interrupts and processor modes was omitted. [27]
<b>Sleeve</b>	User mode emulator for use with Riscose [28], an implementation of the RISC OS API for Linux. Unfortunately it does not support exceptions or other processor modes. [29]

**Leeds Model**

An executable formal specification of an ARM6 processor written in SML for use in a formal verification. [30, 31]

Unfortunately, many of these existing emulators such as ARMulator, SWARM and the Leeds Model are used for purposes other than emulating a real machine. Others such as ARM2, ARMphetamine and Sleeve are designed to only run a limited amount of application-level code and are not suitable for emulating a complete machine. The remaining three, ArcEm, Archie and RedSquirrel all emulate an ARM processor, though suffer from the inherent speed problems associated with interpreted emulation.

#### 1.4.4. Approach

The accuracy of a CPU emulator is extremely hard to measure. This can be done empirically by compatibility (what software can run), or quantitatively by completeness (how much of the processor's facilities it emulates). Different levels of completeness might include support for:

- small stand-alone programs in a test environment.
- interrupts
- exceptions
- processor modes
- coprocessors

To develop the dynamic recompiler to support as many of these as possible, it is helpful to emulate all the other hardware (MMU, graphics etc.) required by a system. This allows scope for using real applications and operating systems to be emulated as tests of accuracy.

Given the anticipated time scale and the size of the problem of implementing a dynamic recompiler, efforts had to be taken to reduce the workload. Rather than attempt to implement the dynamic recompiler as well as the emulation of the rest of the hardware, the approach taken is to make use of an existing (interpreting) emulator for a complete system. By removing the CPU emulation implemented by a third party and adding the dynamic recompilation in its place, the remainder of the system emulation can be utilised as required without distracting from the development of the recompiler.

## 2. Analysis

### 2.1. Introduction to the ARM

#### 2.1.1. Background

The ARM processor was the first commercial 32 bit RISC processor. It was developed at Acorn (ARM originally standing for Acorn RISC Machine), in the mid-1980s in an effort to develop a processor for use in their new range of computers to succeed the 8 bit 6502-based BBC Micro.

The majority of Acorn's processor research team in collaboration with Apple were later spun-off as a separate company, ARM (Advanced RISC Machines) Ltd. Today the ARM design is licensed to over 50 semiconductor companies who manufacture the processors, including many of the largest names in computer hardware, as shown in Figure 5 [32].



Figure 5 - ARM's semiconductor partners who manufacture the processors

As a result of the low price and low power consumption of the ARM, it has been widely adopted for embedded systems and portable devices such as mobile phones. It has also been used in a wide range of consumer computers from desktop machines such as Acorn's RISC OS series (for which it was originally designed) to home consoles such as the 3DO and the Sega Saturn console (where the ARM is used for sound processing), as well as arcade machines such as the Sega NAOMI system. The ARM has really become known for use in portable devices such as the Apple Newton, the Psion Series 5 and Nintendo's Gameboy Advance handheld games console, shown in Figure 6. With over 400 million ARM-based systems being produced in the year 2000, up from 182 million the previous year, the rate of growth of ARM products has been phenomenal.





*Figure 6 - The Nintendo Gameboy Advance, one of many portable ARM-based devices.*

### 2.1.2. Technical introduction

Parts of the ARM design were based on the concepts of the Berkeley RISC chip [33], developed by postgraduates at the University of California at Berkeley in the early 1980s. The ARM adopted some of the RISC concepts, such as a load-store architecture (memory access is only possible through explicit load and store instructions), fixed length instructions (all instructions are encoded in 32 bits) and 3-address instructions (i.e. a destination and two operands). Other RISC-specific concepts, such as delayed branches and every instruction executing in a single cycle, were rejected. The designers at Acorn were forced to keep the architecture very simple as they had insufficient processor design experience to attempt anything more complex [34]. As a result, Steve Furber, one of the lead designers of the ARM, noted that it is, “less radical than many subsequent RISC designs” [35].

The ARM has 15 general-purpose registers; the 16<sup>th</sup> being dedicated to the PC, fewer than for many RISC processors (most having at least 32) but significantly more than most CISC processors. In true RISC style, only simple operations can be performed in an ARM instruction, leading to the divide operation and other more complex operations being omitted. The instruction set is extremely orthogonal to the extent that even the PC can be used as the operand or destination of most instructions.

One of the major strengths of the ARM design is its flexible attitude to condition flags. Every single ARM instruction is conditionally executed depending on the status of the condition flags. In other processors, conditional execution is only available on branch instructions (i.e. to jump to a subroutine if a condition is satisfied). After the influence of the RISC concept, conditional execution has often been extended to move (on x86) and test instructions (on MIPS) to reduce the number of conditional branches needed. The caveat to this flexibility on the ARM is that additionally all instructions have to be able to adjust or not the condition flags.

The other major strength is the ability to perform logical, arithmetic and rotational shifts on instruction operands, while still only taking up a single clock cycle. This combined with the 3-address structure allows incredible flexibility in a logic or arithmetic instruction. Additionally the ARM has extremely flexible block memory access instructions. These allow any combination of the 16 registers to be stored to or loaded from memory in a single instruction. Although not comprehensive (i.e. unable

to even perform a divide), the flexibility of individual ARM instructions makes them extremely powerful.

The uninitiated reader is directed to Appendix A for an overview of the ARM assembly language. This is useful both to get a feel for the subject of this project and as a brief tutorial to ARM instructions, knowledge of which is necessary for understanding the examples in this report.

## 2.2. Identifying the problem

### 2.2.1. Other source processors considered

In the very early stages of planning, the ARM was not the only processor considered as the subject for the dynamic recompiler.

Many of the open-source hobby emulators available are for systems that use early 8 bit processors such as the MOS 6502 and Zilog Z80. These would be relatively simple candidates to implement and running at clock speeds of less than 4 MHz are able to be emulated at many times their real speed using interpreting emulation, making a dynamic recompiler interesting but unnecessary. Motorola's 16 bit 68000 processor was also a candidate, though because of an existing project to create a dynamic recompiler for it (as well as being competently emulated using interpreting methods [36]) it too was rejected.

Emulating a 32-bit Intel x86 processor around the level of an 80386 was briefly considered but rejected because the myriad of different instructions and idiosyncrasies could make it incredibly difficult to implement accurately. Additionally, despite the x86 being so pervasive in home computing, having its origins in the earliest processors of the 1970s means that it is not a typical modern architecture. Any successful emulation while commercially valuable would be of limited use for emulating other architectures.

The MIPS (Microprocessor without Interlocking Pipeline Spurs) processor, another 32 bit RISC architecture developed around the same time as the ARM, was a strong candidate. With MIPS processors having been used in both the Sony Playstation and Nintendo Ultra 64 games consoles, there are several open source emulators available for both platforms (some of which implement a crude form of dynamic recompilation) that could have been used. The MIPS is a more radical RISC design than that of the ARM, doing away with condition flags completely (instead conditional branches test register values [37]), which would simplify the implementation of a dynamic recompiler. The MIPS design goes to extraordinary lengths to optimise use of pipelining, with delayed branches and "bizarre" [38] effects such as multiply and divide operating outside the main pipeline. As a result, techniques developed for a MIPS dynamic recompiler would be unique to that project and might have less relevance to dynamic recompilers for other architectures.

The ARM was settled on for various reasons. The apparent simplicity of individual instructions and limited number of different operations meant that there are few complications (such as floating point or division) on top of the dynamic recompilation issues, or so it seemed initially. Other factors in the ARM's favour were the

availability of well-structured source code to emulators for Acorn RISC OS ARM-based machines and the difficulties of emulating an ARM processor at real world speed, a strong motivating factor. Additionally the relatively un-radical design of the ARM means that a dynamic recompiler for it still has relevance to dynamic recompilers for many other architectures both old and new, RISC and CISC.

### 2.2.2. Which ARM architecture?

As a reflection of the multitude of different applications of the ARM processor, there are many configurations of ARM core currently marketed by ARM Ltd. and still more that are outdated. The question of which one to focus on for this project was carefully considered.

The original Acorn design for the ARM processor stored the PSR (Processor Status Register) in r15 along with the PC, only allowing space for a 26-bit-PC, as shown in Figure 7. After ARM was spun off as a separate company, their later designs separated the PSR and PC, allowing a full 32-bit-PC giving increased address space (4 Gb as opposed to 64 Mb). Most of these new processors retained the ability to execute 26-bit-PC ARM code for backwards compatibility, almost solely to support Acorn's RISC OS operating system which was written largely in 26-bit-ARM assembly.

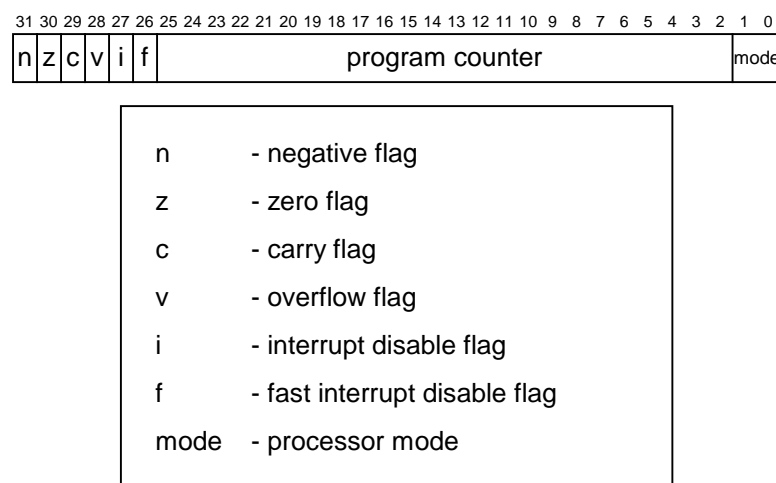


Figure 7 - R15 in 26-bit-PC ARM processors, containing both the PC and PSR

Other later additions to the ARM design have included the Thumb instruction set (a 16 bit encoding of a subset of the ARM instruction set), a 5-stage rather than 3-stage pipeline, 64 bit-result multiplication and Java bytecode execution [39]. These later additions do not have particularly wide ranging consequences for the core of the ARM emulation and could all be added to an ARM dynamic recompiler relatively easily later on.

It was decided that supporting both 32-bit-PC and 26-bit-PC modes in the initial development would be too much effort with no benefit for the recompilation techniques. Choosing 32-bit-PC emulation would be easier since the complexities of the combined PC and PSR would not arise. Additionally it would be similar to all recent ARM developments. In contrast choosing 26-bit-PC emulation would be harder, would not be as relevant to recent ARM developments but had the significant

advantage that it could be used to run RISC OS, allowing use of the large library of RISC OS programs for testing. 26-bit-PC mode was chosen for the available software and the challenge of the combined PC and PSR, with the issues of the 32-bit-PC mode being seen as a subset of those of the 26-bit-PC mode.

The architecture decided upon was the ARM architecture version 2a as implemented by the ARM3 processor. This was the final ARM processor without 32-bit-PC support, as used in the Acorn A5000 computer. The functionality of this architecture is a subset of that in architecture versions 3 and 4 (ARM6xx, ARM7xx, ARM8xx and StrongARM processors) for backwards compatibility [40].

### 2.2.3. System emulator

Red Squirrel was chosen as the emulator the dynamic recompiler was to use for development and access was granted to the source code. Red Squirrel was favoured as it was still under active development by the author, the source was well structured with clearly separate CPU emulation and it is built using a good interactive development environment (Visual C++). It is important to remember that although Red Squirrel has been used during development, the dynamic recompiler is designed to have a clean generic interface so that it can be used with any emulator for an ARM-based system.

### 2.2.4. Target processor

The principal target processor chosen was the 32-bit Intel x86 processor, as used in modern IBM-compatible PCs. This was selected largely because Red Squirrel was available for the platform. Additionally the x86's gargantuan CISC instruction set is practically the antithesis of the ARM's clean RISC design. As a result, converting from ARM to x86 (RISC to CISC) provides a challenging problem.

## 2.3. Getting started

### 2.3.1. Preparation

Before approaching a problem of this scale, considerable time was spent researching methods used in other emulators and relevant technologies. Many different emulators and dynamic recompilers for various platforms were investigated. Work in similar fields such as JVM technology, binary translation and optimising compilers was also found to be relevant although sometimes not practical. Additionally a lot of learning about the ARM and x86 architectures was necessary, studying every detail from various reference manuals.

### 2.3.2. Design overview

The dynamic recompiler is only a part of this project. In attempting to accurately convert fragments of arbitrary programs from one machine code to another, thousands of times a second, there is significant groundwork to prepare. There are two main parts to this work, an ARM disassembler necessary for debugging the rest of the development and an interpreting ARM emulator. This interpreter was to prove invaluable for investigating the problem, debugging and for making design decisions

that could not be made any other way, it also forms a fundamental part of the complete system.

The early stages of the Fusion method for object-oriented design were employed to identify the relationships between various parts of the system. Unfortunately, this high level approach was inappropriate to be continued into the intricate details of the emulation that make up the majority of the project.

### 2.3.3. Implementation

The implementation details of the project were restricted, based on the choice of Red Squirrel as the system emulator to be used. The Microsoft Visual C++ development environment was used to develop the dynamic recompiler. C++ was deemed a suitable language because of its low level features such as shifting and structure unions, its speed relative to many other higher-level languages and its object-oriented nature promoting good program structure. C++ was also the obvious choice given that it is the language used to implement Red Squirrel. As already mentioned Red Squirrel had been implemented for Windows and I saw no need to change that, though the project implementation has been kept as platform-independent as possible.

### 2.3.4. Title

The final decision to be made in preparation was that of a title for the project. In attempting to follow in a long line of bad puns for ARM projects, initially there was little inspiration. As the project progressed an apt working title sprang to mind and has stuck: ‘**Tarmac**’ – it’s very hard.

## 2.4. The System Design

The overall system design comprises of several independent systems, as shown on Figure 8. The system object model is introduced at this stage to provide an explanation as to how the different components of the system fit together; however, the operation of the complete system is complex and will only be understood once the individual components have been explained.

The ARM interpreter communicates with the emulation of IOC and MEMC in order to drive the computer’s emulation. The dispatcher is able to invoke the interpreter, the profiler or native code as required. When the profiler is invoked, it reads from memory the relevant ARM instructions and generates a chunk of intermediate code. The chunk of intermediate code is then passed on to the optimiser and finally to the x86 generator which creates x86 machine code. The machine code is then stored in the code cache so that the dispatcher can invoke it. The ARM disassembler, Armlet disassembler and x86 disassembler are all debugging tools that interact closely with the system during development.

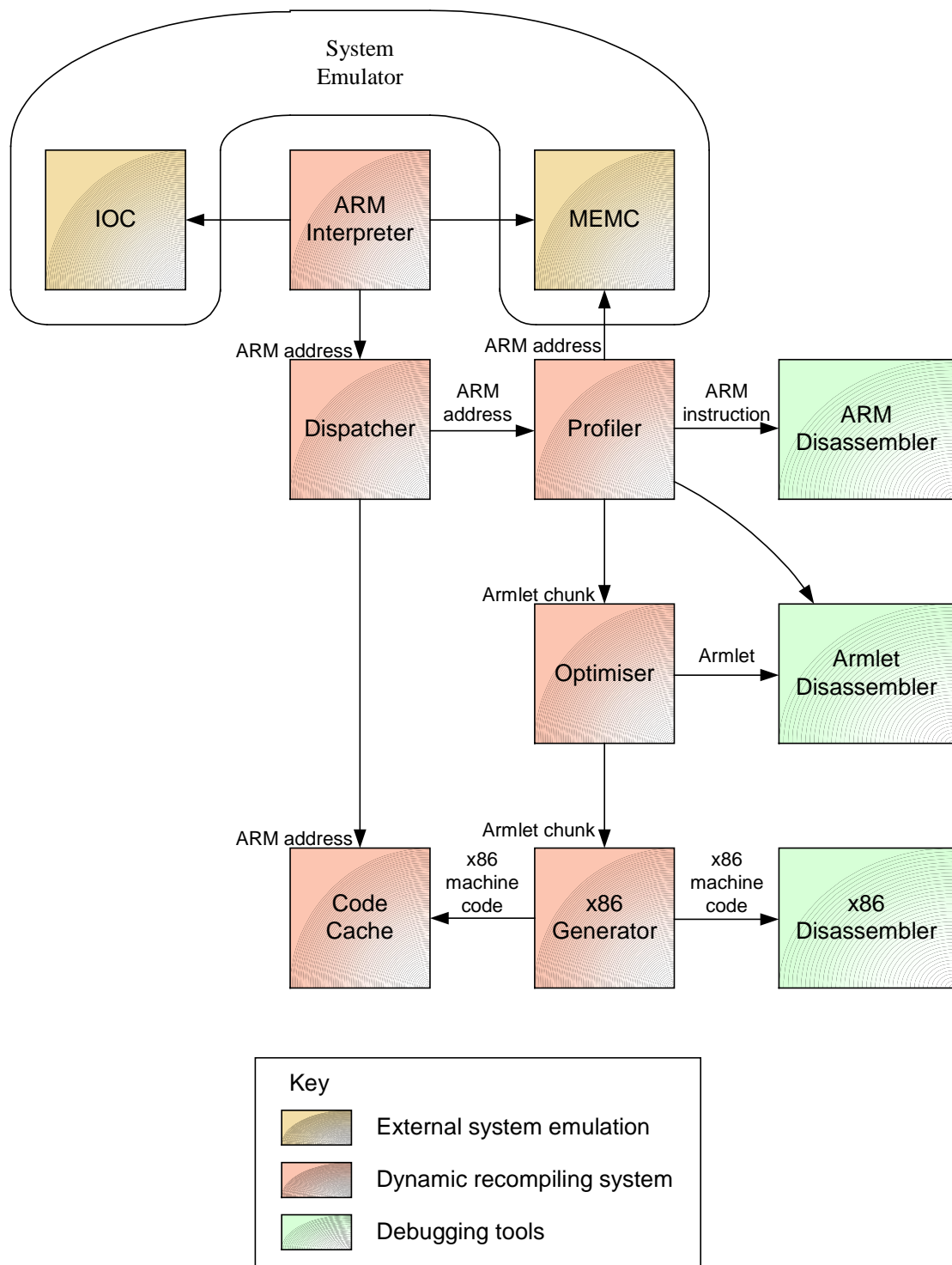


Figure 8 - System Object Model

## 3. Disassembler

### 3.1. Purpose

Although not contributing to the emulation, the ARM disassembler developed in the early stages was an essential tool used in creating the dynamic recompiler. Throughout the debugging of the rest of the project, various sequences of ARM instructions had to be examined in order to determine where the program was going wrong. Using an existing disassembler that takes a binary file and outputs the disassembled code in a text file was of no use since the facility for disassembled code to be displayed with other information from the emulator was needed. Creating an ARM disassembler specifically for the project was also a chance to investigate the problems of decoding ARM instructions before attempting the interpreter.

### 3.2. ARM decoding

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	cond	0 0	i	opcode	s	Rn	Rd	operand 2									
																	Data Processing
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	cond	0 0 0 0 0 0	a	s		Rd	Rn	Rs	1 0 0 1	Rm							
																	Multiply
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	cond	0 0 0 1 0	b	0 0		Rn	Rd	0 0 0 0	1 0 0 1	Rm							
																	Single Data Swap
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	cond	0 1	i	p	u	b	w	l		Rn	Rd	offset					
																	Single Data Transfer
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	cond	0 1 1												1			
																	Undefined
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	cond	1 0 0	p	u	S	w	l		Rn	register list							
																	Block Data Transfer
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	cond	1 0 1	L	offset													
																	Branch
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	cond	1 1 0	p	u	n	w	l		Rn	CRd	Cp#	offset					
																	Coprocessor Data Transfer
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	cond	1 1 1 0	Cp	Opc		CRn	CRd	Cp#	C p	0	CRm						
																	Coprocessor Data Operation
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	cond	1 1 1 0	Cp	Opc	l	CRn	Rd	Cp#	C p	1	CRm						
																	Coprocessor Register Transfer
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	cond	1 1 1 1	comment field (ignored by processor)														
																	Software Interrupt
																	See [41] for key

See [41] for key

Figure 9 - ARM instruction set encoding

In emulators for older processors it is feasible to have a single look up table with the necessary number of entries (256 or 65536) to directly call a function to handle every

possible instruction encoding [42]. For a 32 bit processor like the ARM, with over 4 billion encodings possible this is no longer possible so a more complex decision algorithm for decoding is needed. Fortunately, with relatively few operations on the ARM, much of the 32 bit instruction encoding is taken up encoding operand registers or immediate values (which limits the permutations of different instructions).

The ARM instruction set, as shown in Figure 9, divides neatly into 11 different classes, largely distinguishable by the values in bits 24 to 27 of the instruction. It is on these bits that the disassembler performs initial decoding with further decisions being made as necessary specific to each case for the various other fields.

### 3.3. Design

As an example, take the case of a single data transfer instruction, such as:

```
ldr r0,[r1,r2,ls1 #2]!
```

When executed this takes the address from r2, logically shifts it left by 2 (effectively multiplying by 4), and that result added to r1 gives the address of the word to be loaded from memory into r0. The calculated address is then written back into r1.

As already stated, the single data transfer class of instruction is identified by performing a ‘look up’ on bits 24-27 of the instruction. Parts of that range of bits, such as bit 25, denotes that the modifier to r1 is a register rather than an immediate value and bit 24 denotes pre-index i.e. that `r2 ls1 #2` should be added to r1 *before* the memory access. As a result these two variations are decoded for free by virtue of being included in the look up table on bits 24-27, hence 4 variations of single data transfer disassembly for pre/post index and immediate/register are implemented.

Additional decoding then has to be used on bits 20,21,22 and 23 to decide whether the instruction is load or store, writeback or not, byte or word, increment or decrement respectively. The remainder of the instruction space is used to encode the registers involved, with r0 and r1 being encoded in bits 12-15 and 16-19 respectively. The logical shift left is denoted by bits 5 and 6, the use of r2 by bits 0-3 with the amount to shift by (i.e. 2) encoded in bits 7-11. Each of these separate sections affects the final string of text output by the disassembler.

While apologies must be made for the rather dry explanation, the length of the description highlights the critical problem with emulating an ARM processor - the complexity of decoding. Performing this kind of decoding in a debugging tool such as the disassembler is not a problem, however attempting to perform this millions of times a second in an interpreting emulator is.

#### 3.3.1. Testing

Being certain that the disassembler was accurate was of great importance, as bugs left undetected at this stage could compound the difficulty of debugging the rest of the project. In order to guard against this, large samples of disassembled text produced by disassembling real RISC OS programs, were systematically compared to the results from StrongEd [43]. StrongEd is a disassembler on the RISC OS platform and has



been thoroughly tested during use by hundreds of programmers, this makes it a good source for comparison to the disassembler in order to verify it.

The use of the disassembler was so successful that the envisaged debugging graphical user interface (mentioned in the progress report) was unnecessary. In a previous emulation project [44], a debugging interface to execute single instructions one at a time was implemented; however, in this case, having a flexible disassembler proved sufficient. The disassembler was invaluable for testing all other parts of the project but particularly the interpreting emulator.

## 4. Interpreter

### 4.1. Purpose

The purpose of the ARM interpreter is to emulate any instructions that are not dynamically recompiled. It is also needed to provide a fully working model of the ARM processor to aid in the development of the dynamic recompiler.

### 4.2. The problem with JIT

In attempts to speed up Java Virtual Machines, *just-in-time* (JIT) compilers have often been employed. In a JIT JVM when a Java method is called for the first time it is recompiled into native code and stored to be used every subsequent time the method is called [45]. As a result of always executing recompiled code, a JIT JVM need have no interpreting emulator.

The JIT JVM gets most of its performance boost from the second (and subsequent) times that a method is called, as the first time it is called, expensive recompilation has to take place. Indeed recompiling and executing a method once can take longer than simply interpreting it and as a result, overall speed is only improved if a method is called multiple times. The converse observation of the principle of locality is that an '[emulated] program executes 90% of its code only 10% of the time'. Having to recompile this 90% of code that is only rarely reused can lead to an undesirable performance reduction in the JIT JVM.

This problem is particularly true of program initialisation where most of the code executed will only be executed once while the program loads. Interpreting these sections of code can often give better performance than recompiling them. The lesson is that there is no point in recompiling code that is never going to be executed again.

### 4.3. The HotSpot™ alternative

Rather than recompile every method when it is called, Sun Microsystems' HotSpot™ JVM is more sophisticated and only recompiles selected methods. The first few times a piece of code is executed, the Java bytecode is interpreted and some analysis is done on the code to identify performance 'hot spots'. Performance 'hot spots' are sections of code that take a lot of processing time and would benefit from optimisation by recompiling to native code [46]. It is only these sections of code that are actually recompiled since other areas that are not executed enough to recoup the cost of recompilation, would slow the emulation. This demonstrates the necessity of an interpreting emulator as a critical part of a dynamic recompiler and is the approach taken in this project.

To avoid confusion the reader should note that JIT is sometimes used to refer to any JVM which recompiles Java bytecode. In this report, the naming convention used by Sun Microsystems has been followed where a JIT JVM is one that recompiles every piece of code that is emulated (and therefore does not require an interpreting emulator).

## 4.4. Quantifying the JIT problem

The difference between the JIT, HotSpot and interpreting emulators is best demonstrated in some quantifiable way. This can be done by calculating the relative total cost of emulating a chunk of code for each algorithm, using the following formulae.

Interpreter	$n c_i$	
JIT	$c_r + n c_c$	
Hotspot-style recompiler	<i>if</i> $n > t$ <i>then</i>	$t c_i + c_r + (n - t)c_c$
	<i>else</i>	$n c_i$

Where:

- $c_i$  – cost of a single interpreting emulation of the code.
- $c_r$  – cost of recompiling the code.
- $c_c$  – cost of calling the recompiled code.
- $n$  – number of times the code is to be emulated.
- $t$  – number of times the code is interpreted before being recompiled.

Dividing these results by  $n$ , gives the ‘average cost per emulation’, i.e. the average amount of processor time used to emulate a chunk of code just once. By assigning typical costs to these values in the ratio 1 : 5 : 25 for  $c_c : c_i : c_r$  with  $t$  as 1 (i.e. the hotspot-style algorithm interprets just once before recompiling) and varying the values of  $n$  used, the graph generated is as shown in Figure 10.

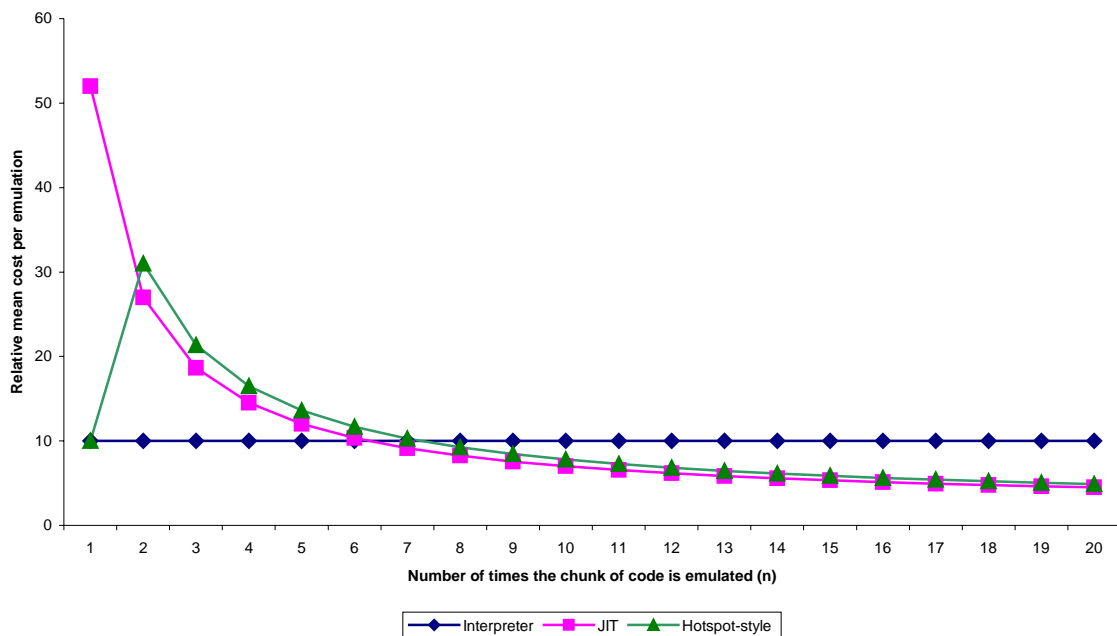


Figure 10 - The relative average cost per emulation of different emulation methods

From the graph it is clear that the JIT method is the cheapest approach for code that is executed many times (by a small margin). Unfortunately, this is coupled with an intolerable cost for code that is executed only once. Interpreting emulation, as

expected, has the same cost for each emulation no matter how many times emulated. The hotspot-style algorithm has the best of both methods, combining a progressively cheaper emulation for high values of  $n$  relative to interpreting, with a large cost saving over JIT for the smallest values of  $n$ . It is this property that makes it the best approach to emulation.

## 4.5. Faster decoding

Since the interpreter affects the overall performance of the emulation, great pains have been taken to make the interpreter as fast as possible. The problem of decoding ARM instructions quickly, forced the design of the interpreter to take a more aggressive approach than the disassembler. Rather than decoding on the obvious instruction bits of 24-27, which would lead to an extensive (and expensive) decoding tree, decoding is performed on the byte from bits 20-27.

This leads to a 256-entry-table of the code to deal with all instructions and far more extensive specialisation (and therefore less decisions) for each entry. As a result, where in the disassembler a single entry covered the entire class of data processing instructions (logic and arithmetic operations), the interpreter table has 4 table entries just for variations of the add instruction that use immediate or register values and do or do not adjust the condition flags. The form of the resulting structure is shown in Figure 11.

```
switch( getField(currentInstruction, 20, 27) )
{
    . . .
    // add rd, rn, rm
    case 0x08:
    . . .
    // addS rd, rn, rm
    case 0x09:
    . . .
    // add rd, rn, imm
    case 0x28:
    . . .
    // addS rd, rn, imm
    case 0x29:
    . . .
}
```

*Figure 11 - The format of the interpreter's instruction decode table showing add instructions*

This has the clear advantage of algorithmic efficiency by combining many decisions across the different classes of instruction into one fast look up. The disadvantage is that since there are large pieces of intricate code that may be identical or very similar, the complexity of the program increases dramatically leaving it more vulnerable than most to programmer error. To combat this, extensive templates are used throughout for common operations such as for getting the operands for data processing instructions.

## 4.6. Interfaces

The ARM emulation cannot emulate a complete system on its own. In Red Squirrel it is linked to an emulation of the IOC (Input/Output Controller) which manages interrupts, and MEMC (Memory Controller) which is an interface between the ARM and the system memory. Both the IOC and MEMC chips were custom designed by Acorn Computers Ltd. for their RISC OS machines but almost all ARM-based systems have some equivalent interrupt controller and memory management unit. The functionality provided by these subsystems is so closely linked to the CPU that in some ARM processors (such as the ARM7500) they are actually on the same chip [47].

As a matter of good design and in order that the results of the project could be used with any ARM system emulator, great care was taken to ensure that no dependency on Red Squirrel was established unnecessarily. As a result, the interface between the interrupt and memory controllers is through an extra indirect interface used throughout the ARM interpreter. The ARM emulation accesses memory through this interface indicating a logical address. This is passed to the MEMC emulation, which accesses the appropriate page in memory and updates any memory-mapped systems without the ARM emulation knowing any details about MEMC, as shown in Figure 12. The only other ARM to MMU action is where the ARM can set the 'trans' flag. The 'trans' flag is used for certain memory access instructions and forces the MMU to treat the memory access as if from a non-privileged mode irrespective of the current processor mode.

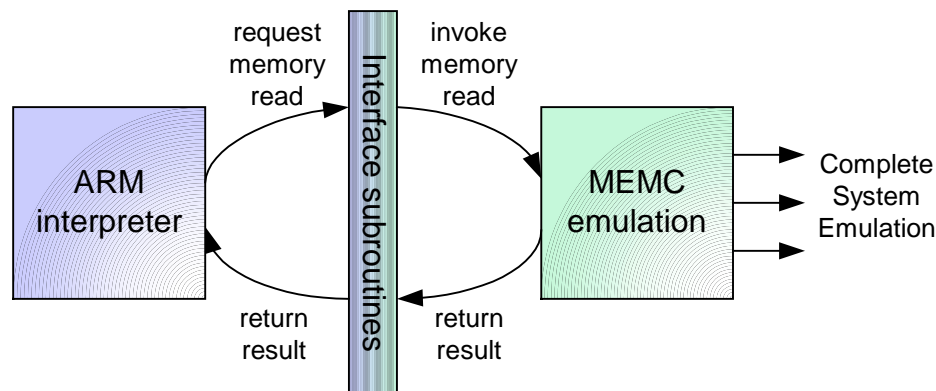


Figure 12 - Example of the interface used to separate ARM and MEMC emulations

Interrupts are likewise implemented transparently to the ARM emulation, with the ARM emulation signalling the IOC emulation at the start of each instruction's execution and IOC returning information as to whether an interrupt has occurred. This leaves the CPU emulation unburdened with the complexities of calculations and timings for the IOC emulation, having just to handle the effects of an interrupt that the software expects. This divorcing of the emulation of external systems is a strength as the finished project can then be used for a completely different machine's ARM emulation.

The third interface between the ARM emulation and the system emulation is that of coprocessors. From the earliest designs, the ARM has included a generic coprocessor

interface; used for floating point, signal processing and system control. While many types of coprocessor are relatively under-utilised on the ARM, the system control coprocessor is commonly included ‘on-chip’ for adjustments to the MMU and cache, and is emulated by Red Squirrel. The ARM interpreter uses Red Squirrel’s emulation for the system control coprocessor, though the interface is generic enough and limited in its impact on the interpreter to be adjusted for any other ARM system emulator or coprocessor type.

The complete interface between the system emulation (at present, Red Squirrel) and the ARM emulation is summarised by the actions shown in Figure 13.

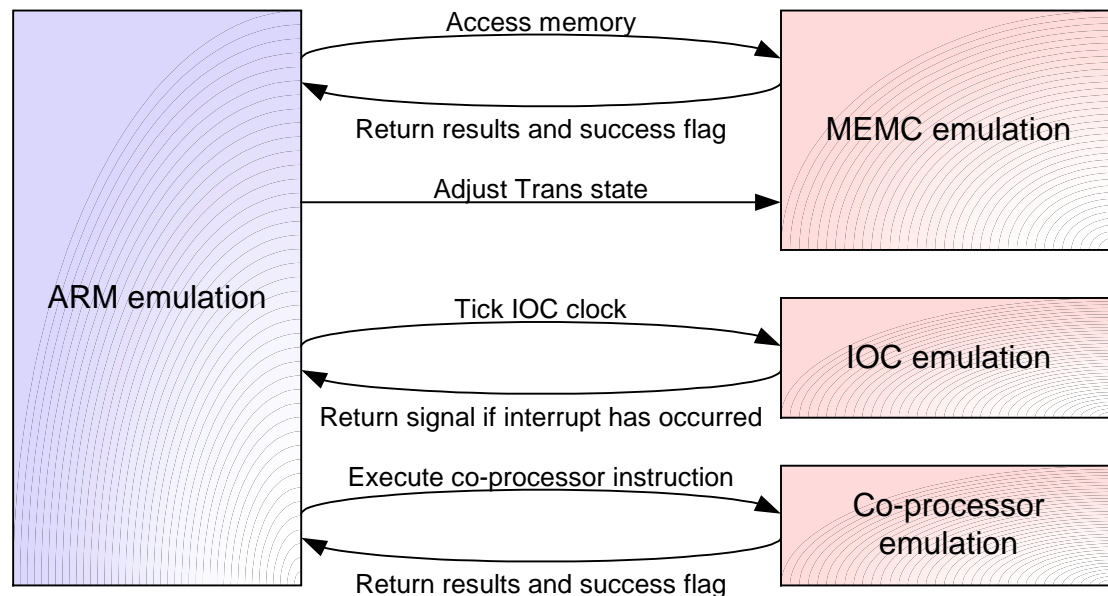


Figure 13 – The interface between the ARM and system emulation

The good design and minimal scope of these interfaces keeps the ARM emulation sufficiently distant from that of the rest of the system even for something like memory access which affects large parts of the ARM interpreter.

Further interfaces to remove the way in which the emulated state is stored from the way in which it is implemented were used internally to the ARM emulation. This is often frowned upon in emulators, where the extra layer of indirection in function calls is costly (when executed millions of times a second). However, it is necessary so that the way emulated registers and flags are stored can be adjusted for the dynamic recompiler without repercussions for the interpreter.

## 4.7. The emulation loop

As a result of pipelining, the looping used to emulate an ARM processor is slightly more complex than the simple fetch-decode-execute design described in section 1.3.2. In a real ARM processor, one instruction is being executed, the one after it is being decoded and the one after that is being fetched from memory at any one time. A 5-stage pipeline was implemented after the ARM7, adding buffer and write-back stages, though this does not affect this project [48].

As a consequence of the pipeline, the PC is always 2 or 3 instructions ahead of the current instruction being executed, depending on the circumstances. This means that in the rare event that an instruction changes the next instruction, the original version will already have been prefetched and will be executed unchanged (some games software is known to actually do this [49]). The solution is to emulate the prefetch by always fetching the next instruction in the iteration of the loop before it is to be executed. On a branch or adjustment of the PC, the pipeline is flushed and the prefetched instruction invalidated causing the prefetched instruction to have to be reloaded.

As described in the previous section, interrupts are triggered by ‘ticking’ the IOC clocks. The prefetched instruction value is then decoded and emulated and after the instruction is executed, any interrupts signalled by the IOC emulation are dealt with. The loop then returns to tick the IOC again, as shown in Figure 14.

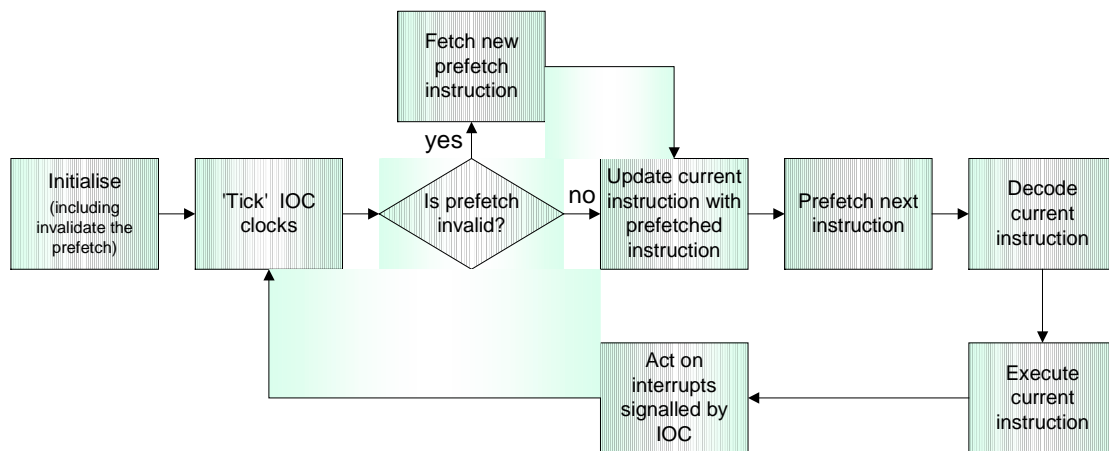


Figure 14 - The ARM interpreter loop

## 4.8. Implementation

In any software, using assembly language allows faster programs to be produced than using a high level language. This is even more so in emulation where similarities between the way the emulated and emulating processors adjust condition flags can be utilised in a way not possible in higher level languages. However, due to the scale of the problem of writing an ARM interpreter (over 5000 lines of C++), as well as the interface to Red Squirrel being entirely in C++, it was not suitable to implement the interpreter in assembly. Additionally, despite selecting the x86 architecture as the target for the dynamic recompiler, the project is not necessarily tied to that platform so a multi-platform language such as C++ is desirable.

During the development it became apparent that algorithmic considerations are not the only factors affecting program performance. In order to escape the overheads of extra function calls in implementing interfaces as described previously, it is important to leverage the optimisation facilities of the implementation language. For example, in C++ the use of the `inline` directive can significantly improve performance by inlining methods [50].

## 4.9. Debugging

Bugs in the ARM interpreter are extremely difficult to identify. The main problem is that the effects of a slight anomaly in the emulation, such as incorrectly setting a single bit on a single iteration of a loop, may not become apparent until millions of emulated instructions later. This problem was appreciated well in advance and precautions were taken in comparing descriptions from several sources at every stage of the implementation.

It is not feasible to exhaustively test every possible ARM instruction looking for problems. However, many of the earliest bugs identified were found by writing small test programs on an Acorn Risc PC, to test subsets of the instruction set. These programs are then loaded into a ‘dummy’ memory and emulated until a `mov pc,r14` instruction is reached (normally used to end a program). The results visible in the emulated registers are then compared to the results from the real hardware to find errors.

Following the success of using test programs, a line by line inspection was made of the code, searching for problems, at each stage verifying operation with the manuals. Often the manuals are vague, conflicting or simply omit boundary cases leaving only experimentation with the real hardware and Red Squirrel’s existing interpreting emulator. For example, one source declares that for the `mul` instruction, “r15 [the PC] may be used as one or more of the operands” [51], while another states that doing so has, “unpredictable results” [52].

Other sources and emulators highlighted features of the ARM’s operation that were not at all obvious [53, 54]. A good example of this is where the instruction

```
movs r0,#256
```

will clear the Carry flag for no obvious reason whereas in most cases the `movs` instruction only adjusts the Negative and Zero flags. Investigations showed that because of the way the immediate value 256 is encoded in the instruction, as 1 rotated right by 24 places, the value of the last bit to be rotated across the end of the register is 0 and the Carry flag is therefore set to 0.

Having attempted to accurately translate every aspect of the ARM processor from the manuals to a fully working model and tested it thoroughly, the final test was to attempt to boot Acorn’s operating system, RISC OS. This is seen as an extremely intensive test of the compatibility and completeness of the emulator as the operating system performs a lengthy system test on start up and will refuse to boot at the slightest error.

A copy of RISC OS v3.11 was extracted from an Acorn A5000 where it is stored on ROM and set up to be loaded into Red Squirrel’s emulated memory map. The loading code and dummy memory previously used for test programs were deactivated and the interfaces adjusted to access Red Squirrel’s IOC and MEMC emulation. Despite all the testing that had gone before, a different approach was required to get the ARM interpreter to a fully working state before moving on to the dynamic recompiler.



Miniscule errors either in the interpretation of the sources, or in the sources themselves initially prevented the ARM interpreter from running RISC OS. To locate these errors, a certain number of instructions (typically 50,000 at a time) were emulated on Red Squirrel's interpreter, dumping the disassembled instruction and all register and flag values to a text file after every instruction. Performing the same test on my ARM interpreter and then running file comparisons on the two dumps was a good way to find the smallest discrepancy between the two interpreters.

Even with such a mechanical bug detection system, the linear search for differences was still a slow one. It was found that in the event of a single bug, the execution of every successive instruction was affected. As a result a manual binary search could be performed across the execution space at intervals of several hundred thousand instruction emulations in order to find discrepancies. Once discrepancies were found, time-consuming investigation into the cause of the problem ensued, an example of such an investigation follows.

The 6,531,120<sup>th</sup> instruction executed in loading RISC OS is `teqp pc,#3`. The instruction executed immediately after it is at a completely different address in memory (0x1C) and is `b 0x381134C`. The only difference between the two emulations is that after the first instruction, r14 in Red Squirrel's interpreter had been incremented by 4 to 0x3811757, while r14 in my interpreter had been set to 0.

The `teqp` instruction performs an exclusive-or on the two operands, in this case the PC and the value 3, updating the PSR with the result. As shown in Figure 7, this adjusts the processor mode flags, changing to supervisor mode. Immediately after that instruction an IRQ (Interrupt Request) exception occurred, deducible from the PC changing to 0x1C (the IRQ vector 0x18 incremented by 4 because of pipelining effects).

When an IRQ exception occurs, the ARM processor changes the register bank used for r13 and r14, copies the PC from r15 into r14 and jumps to the IRQ vector. From examining the emulation of the IRQ exception, it was found that incorrectly r14 was being updated with the value of r15 *before* the register bank changed, instead of afterwards (and therefore r14\_irq was unmodified and held the value 0).

While a complete understanding of the investigation may escape the reader, an appreciation of the difficulty of discovering the cause of the problem should not. Many such problems had to be tracked down, which took considerable time. Although most parts of Red Squirrel are deterministic, unfortunately the keyboard interrupts are asynchronous resulting in non-deterministic behaviour. This manifests itself by the ability to run the interpreter on the same test data twice and get different results, making automatic comparisons between the Red Squirrel interpreter and my own impossible. Fortunately, this did not affect RISC OS until the later stages of loading and slower manual examination of test dumps allowed sufficient testing.

## 4.10. Compatibility

The level of compatibility achieved by the ARM interpreter is extremely good. RISC OS boots completely to the desktop and is able to run high-level software, including

the very complex ARM BASIC interpreter. This is quite an achievement and allowed development of the dynamic recompiler to continue with confidence, having a working model of the complete ARM processor for comparison.

## 5. Recompilation

### 5.1. Overview

The recompilation system has a simple purpose: to generate native x86 machine code to emulate a given sequence of ARM instructions. This generation must be done quickly so that it does not slow down the emulation but should also make the generated machine code as fast as possible.

An algorithm programmed for two different architectures could be implemented optimally for each processor. However, despite the algorithm being identical, the resulting machine code would have little in common. Ideally a dynamically recompiling emulator would be able to reverse engineer the machine code to the original algorithm and then optimally recompile it for the target processor. Unfortunately automatically inferring the high level semantics of an arbitrary piece of machine code is a difficult problem. Under the limited processing conditions of a dynamic recompiler, attempting to do so is simply not feasible.

The alternative is to emulate the ARM instructions in the same way as the ARM interpreter does. This means generating native machine code that adjusts the registers and flags of the emulated ARM processor, in the same way as executing the instructions would on the real machine. In this way, we avoid the problems of trying to deduce the high level operation of a program.

### 5.2. Methods of generating native code

Direct Translation is the simplest approach of generating native machine code, where instructions are translated one by one. Once a source instruction has been identified, a predefined section of native code is generated for that instruction. The next instruction is then decoded and translated and appended to the machine code generated for the first. All this method does is to take pre-assembled 'covers' of target machine code and put them in the same order as their corresponding source instructions, as shown in Figure 15. This method removes the cost of fetching and decoding emulated instructions relative to an interpreting emulator though does little else. Although relatively straightforward to implement, this method is inflexible as it is unable to take account of instructions either side of the current one being translated and is therefore fairly crude in its optimisations.

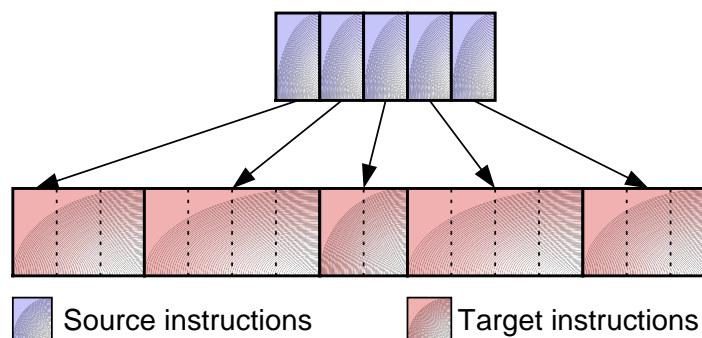


Figure 15 - Direct Translation of individual instructions

Better forms of recompilation are performed on entire sequences of source instructions at once. In this way, optimisations over several instructions can be used to generate much faster code than is possible using direct translation.

### 5.3. The use of intermediate code

There are two main ways to perform optimising dynamic recompilation: with or without an intermediate code. An intermediate code is a third representation (other than source and target machine codes) of the instructions to be translated. The source code is converted into intermediate code and then the intermediate code translated into the target machine code, as shown in Figure 16. This method has been used in recompiling JVM's [55] and other dynamic recompilers [56].

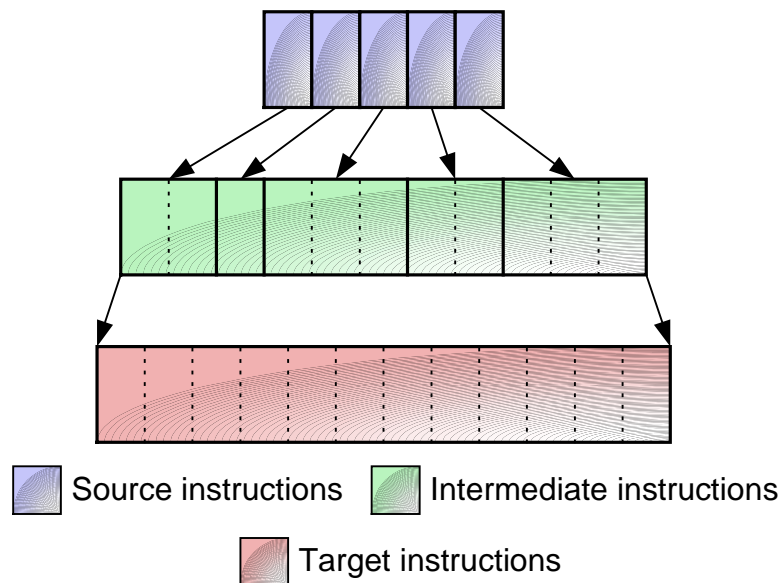


Figure 16 – Recompilation using an intermediate code

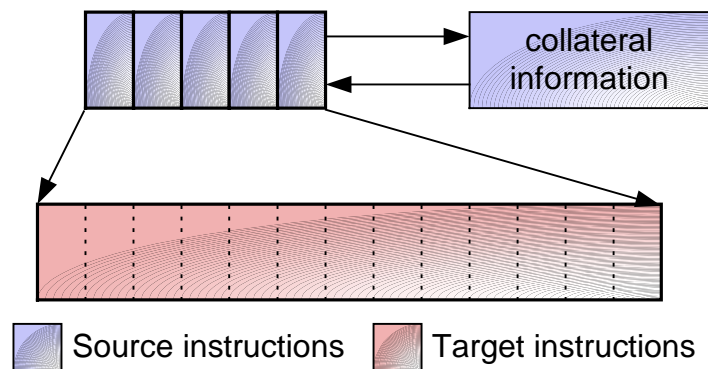
#### Advantages:

- The native code generator can be changed to 'retarget' the dynamic recompiler so that it translates from ARM to a different target platform, without affecting the rest of the system.
- Recompiling ARM instructions that have many stages (such as Block Data Transfer) is complex and benefits from being decomposed to smaller simpler intermediate instructions.
- It is easier to debug than going direct from ARM to x86 as the intermediate code maps more closely to x86 than complex RISC instructions.
- It is easier and possibly faster to implement optimisations of the generated code on an intermediate code rather than only working with machine codes.

#### Disadvantages:

- The recompilation is slower as a result of having to generate intermediate code as well as target code.

The alternative to employing an intermediate code is to do a translation straight from source to target code but avoiding the problems of Direct Translation. Collateral information, such as whether the condition flags need adjusting, is collected by looking at the rest of the sequence pre-translation. The target code generator can then make decisions about what instruction to generate to best emulate parts of the sequence rather than just individual instructions, as shown in Figure 17. The main advantage of this method is that the target code generator has explicit knowledge of the source instructions it is representing. Unfortunately, the cost of repeatedly decoding instructions is high for a RISC architecture, making optimisations less practical. This approach is a good one when the source architecture is relatively simple, such as for 8 bit processors or java bytecode, where to attempt to decompose instructions any further is unnecessary.



*Figure 17 - Recompilation without an intermediate representation*

As a result of the inherent complexity possible with a single ARM instruction, decomposing to an intermediate code aids conversion to the relatively cumbersome target x86 instructions. Therefore, although both approaches have their merits, the intermediate representation is preferred for emulating the ARM architecture.

## 6. Armlets – An Intermediate Code

### 6.1. Purpose

The purpose of the intermediate code is to decompose ARM instructions into several sub-instructions that can be used to aid optimisation and x86 generation. So for example the ARM instruction:

```
add r0,r1,r2,ls1 #3
```

might be decomposed into sub-instructions that describe its operation, such as:

```
t = r2 << 3
r0 = r1 + r2 + t
```

This could then be translated relatively easily into the x86 instructions (written with the latter register being the destination):

```
movl r2,t
shll $3,t
movl r1,r0
addl r0,r2
addl r0,t
```

### 6.2. The ‘explicit-implicit problem’

With the previous example, it is straightforward to represent the internal operation of the ARM instruction in terms of the logical-shift-left and addition operators. Unfortunately, other ARM instructions have more complex side effects that to explicitly define in an intermediate code would be very lengthy. For example, at first appearance the ARM instruction

```
adds r0,r0,r1
```

seems more simple than the previous example. However, the ‘s’ appended to the add mnemonic specifies that the instruction should update the condition flags with the result of the calculation. The operation would be explicitly defined in intermediate code as

```
r0 = r0 + r1
N flag = r0 >> 31
Z flag = if (r0 == 0) then 1 else 0
C flag = CarryFrom(r0 + r1)
V flag = OverflowFrom(r0 + r1)
```

This would translate to a complex and lengthy stream of x86 instructions. However, both the ARM and x86 add instructions (like add instructions for many other architectures) set their condition flags in exactly the same way. If the x86 flags could

be made to represent the ARM condition flags, the entire ARM instruction can be translated to one x86 instruction:

```
addl r1,r0
```

However, this one-to-one conversion would not be possible from such an explicit representation in the intermediate code and would require some kind of representation that was implicit of the instruction's complexities.

There are other ARM instructions that have such complex behaviour but unfortunately are not similar to any x86 instruction. For such instructions there is no option other than to explicitly define their operation in both the intermediate and x86 code.

This leads to what I term the 'explicit-implicit problem' – that in order to generate the best possible code, an intermediate code for a binary translator must be able to represent the both explicit behaviour of some instructions and the implicit behaviour of others. This can be more complex than the simple statement suggests.

### 6.3. Options

The designs of various existing intermediate codes were considered for the purposes of this project. The obvious intermediate codes to explore were those used in programming language compilers. These are typically used to describe mathematical expressions and data flow after parsing, to aid optimisation and code generation. Unfortunately, language compilers generate intermediate code to represent relatively high-level structures such as if-then-else and arithmetic operations without the complexities of emulated condition flags.

The code of virtual machines such as Java bytecode and Pascal P-code were considered but dismissed as their stack-based design and lack of condition flags would hinder a useful representation of the ARM register-based system. Hardware description languages such as VHDL [57] were briefly considered; however, such languages are used to define processors and electronic components and as such are completely explicit.

The approach of other dynamic recompilers that use an intermediate code [58] has been to use a stylised register transfer language and apparently waive any attempts at exploiting the similarities between processors. This is done in the hope of allowing complete retargetability by changing the source or target processor.

The intermediate code developed for ARMphetamine [59] is a close match to the requirements of this project. This is to be expected given the similar theme of the projects. However, ARMphetamine's intermediate code does not address many of the issues inherent with a real processor. As a result, exceptions, interrupts and the coprocessor, all fundamental for a real system emulation, are practically unsupported.

Unfortunately because of the extremely specific requirements of the intermediate code, there are no designs that can be used directly. Instead I have designed my own intermediate code, influenced by the existing work.

## 6.4. Characteristics of Armlets

In attempting to confer the reduced nature of the instructions relative to a full ARM instruction I have called the intermediate code *armlets*. One or more armlets describe completely the accurate emulation of a single ARM instruction. This section serves as an overview and introduction to armlets. A complete definition is provided in appendix B.

Armlets are a form of ‘3-address code’ [60], as used in many programming language compilers. This means that for each armlet there is a single operation and three arguments, normally one destination and two operands. This is the same approach used in ARM instructions and armlets are written in the same format for simplicity:

`<operation>   <destination>, <operand1>, <operand2>`

The 3-address code implementation is using quadruples [61], where the operands to be used for an operation refer to variables, as opposed to triples where the operand values are references to the intermediate code instruction that created them [62].

Unfortunately, the x86 is not as flexible as the ARM and must write the result of a calculation back over one of the source registers. Therefore, while an ARM instruction can execute 3-address statements of the form

`add x, y, z`                      meaning                       $x = y + z$

leaving *x* and *y* unchanged. An x86 instruction is only capable of

`add y, x`                      meaning                       $x = x + y$

This overwrites the original value of *x* with the result. As a result, at some stage in the recompilation all 3-address instructions have to be split into two or more 2-address instructions. In the early stages of designing the intermediate code, careful consideration was given as to whether this should occur during translation from ARM to armlets or armlets to x86. Most modern processors use 3-address instructions and so would suffer if the intermediate code used 2-address. Additionally most compiler intermediate codes used for optimisations act on 3-address instructions. As a result, the conversion to 2-address instructions is left to the x86 generator.

As a reflection of the fact that an ARM instruction is ‘decomposed’ into a set of armlets, the operations that armlets perform are (mostly) constituent parts of their parent instruction and operations that are available on the x86. Standard logic and arithmetic operations such as `eor`, `not`, `add` and `mul` as well as shifting operations such as `lsl`, `lsr`, `asr` and `ror`, available on the ARM are all implemented.



Armlets operate on a set of *variables*, mostly representing the state of the emulated ARM processor, as listed below:

r0 - r14	The ARM registers.
pc	Program Counter.
nflag	Negative flag.
zflag	Zero flag.
cflag	Carry flag.
vflag	Overflow flag.
iflag	Interrupt-request disable flag.
fflag	Fast-interrupt-request disable flag.
mode	Current processor mode.
t0 - t29	Temporaries.

The ARM condition flags can be adjusted explicitly by making one of the `flag` variables the destination for the result of an armlet. Additionally each armlet has a set of *outflags* associated with it that denote which of the condition flags it updates. The detail of exactly how the flags are updated is implicit in the armlet and is identical to the corresponding ARM instruction. In this way, the target code generator is able to implement the armlet in the best way possible for the target platform. Whether this is by making use of a near identical instruction on the target machine, or by explicitly calculating all the flags, is not a concern of the armlet generator.

Each armlet also has a set of *inflags* associated with it that denotes which condition flags it makes use of in its operation. Like the outflags these are identical to the armlet's corresponding ARM instruction by default. This allows the code generator to make decisions about which flags it needs to operate on. As a result, armlets are formally written as follows:

```
[xxxx->xxxx] movc    t0,0x1
[xxCx->NZCV] adc     r1,r1,t0
```

which is equivalent to the ARM instruction:

```
adcs    r1,r1,#0x1
```

The first sequence of 4 characters in the square brackets are the *inflags*, so the `adc` armlet makes use of the `cflag` as input. The second sequence of 4 characters in the square brackets are the *outflags*, so the `adcs` armlet will adjust the `nflag`, `zflag`, `cflag` and `vflag`. The other feature to note is that the `movc` armlet operates completely independently of the flag variables, neither requiring them as input nor affecting them as output. All armlets can be specified like this in order to be used for armlet control flow, without affecting the emulated state.

Although armlets are primarily a 3-address structure, there are five main classes of armlet with different numbers of arguments, as follows:

Implied	– has no operands, e.g. <code>settrans</code>
Transfer	– a variable and 32 bit immediate, e.g. <code>movc r0,#0x12345678</code>
2-variable	– two operand variables, e.g. <code>cmp r0,r1</code>
3-variable	– three operand variables, e.g. <code>ror r0,r1,r2</code>
Immediate	– just one 32 bit immediate operand, e.g. <code>goto #0x87654321</code>

In order to allow for easier optimisations and to keep the armlet encoding compact, all data processing armlets operate exclusively on armlet variables. In order for an immediate value to be used, it must be loaded into a variable using the `movc` (for move constant) armlet.

## 6.5. The Program Counter

In an interpreting emulator, one of the overheads of every instruction emulation is updating the PC to point to the next instruction. One of the benefits in recompiled code is that the PC does not need updating after every instruction (since the next instruction does not need fetching and decoding) and can just be set to the correct value before returning from the recompiled code. For this reason the `pc` armlet variable is unused most of the time and is only set to the correct value, calculated at compile-time, immediately before leaving the recompiled code. The point is that because of this, the `pc` variable will rarely contain the actual PC value. Instead, it is treated as a way of returning a value to the interpreter so that the interpreter knows which instruction to continue emulating at.

As a result of the extreme orthogonality of the ARM, the program can treat the PC almost like any other register operand. These cases are the only other situation where the `pc` variable will be updated to contain the correct value.

## 7. Profiler

### 7.1. Purpose

The purpose of the profiler is to take a sequence of ARM instructions passed to it by the dispatcher and generate a sequence of armlets to represent them. This sequence of armlets is then passed on to the optimiser.

### 7.2. Characteristics of a chunk

How much code should be recompiled at any one time? It is obvious that if only one instruction is recompiled at a time, the system will have to keep returning to the dispatcher after every instruction emulation, a large and unnecessary overhead. The system therefore deals with sequences of source instructions, but what denotes where these sequences, known as ‘translation units’, start and end?

In traditional compilers, one of the crucial quantities used for optimisation and code generation is the *basic block*. This is a sequence of consecutive instructions where execution starts with the first instruction and ends at the last, with no possibility of interruption or leaving part way through [63]. This characteristic makes them very useful for optimisation as they interact with the rest of the program in only a very simple way. Basic blocks were considered for use as the translation unit but they are typically quite short and would also mean returning to the dispatcher at the end of every iteration of a short loop.

The translation unit decided on shares some of the properties of the basic block. I have called this a *chunk*, (in keeping with ARMphetamine [64] though the exact definition of that project’s chunk is different) to differentiate it from the basic block. A chunk has only one entry point at the first instruction, though it can have several exit points. Each instruction in the chunk is evaluated and identified as belonging to one of four categories according to its effect on control flow:

- **Unaffected** – an instruction that has no effect on control flow.
- **Branch-inside-chunk** – a branch to an instruction that has already been identified as being in the chunk.
- **Branch-outside-chunk** – a branch outside the chunk.
- **PC-adjusting** – a non-branch instruction which affects the PC.

Instructions in the first category are simply added to the chunk, moving on to the next instruction. A branch-inside-chunk instruction is handled internally to the chunk by generating a `goto` back to the appropriate armlet, which allows small loops to be handled within a single chunk.

A branch-outside-chunk is a branch instruction that points to a location either before the first instruction in the chunk or after the current one, as shown in Figure 18.

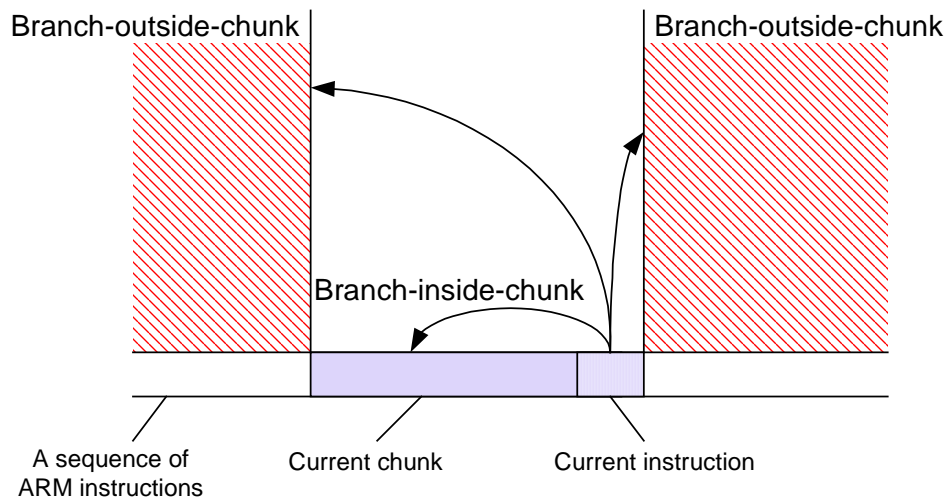


Figure 18 - Deciding whether a branch is inside or outside the current chunk

A branch-outside-chunk can have the interesting property of branching to an address that is already recompiled. It is possible to generate a native code branch to the start of this existing chunk so that execution of recompiled code can continue uninterrupted. However, complicated interdependencies between recompiled chunks would inevitably develop. This would mean that in the event that code was modified and a chunk had to be deleted, many other chunks would need either modifying or deleting also (so that they did not just branch into a void). In this way, a chain reaction across dependent chunks could end up affecting large sections of recompiled code. The approach taken is to drop back to the dispatcher, which then checks for an existing chunk to invoke.

Adjustments to the PC value can be coped with in very different ways. Mapping the new arbitrary value of the PC to a possible chunk of recompiled code for that address, without leaving the current chunk is a significant problem. The approach taken is to drop back to the dispatcher so that it can handle it. It has been suggested that alternatively a hash table (from emulated PC to appropriate chunk) could be implemented in the recompiled code for all the values of the PC that occur [65]. This could then be updated post-recompilation when a new PC value occurs. However, as well as the complexities of the code involved, the cost-benefit of updating the hash table seems prohibitive and there are no known implementations of such a system.

Unfortunately, since any forward branch ends the chunk, a relatively common ‘if-then’ statement would also end the chunk. The reason is that if the condition is not satisfied then the execution has to jump (forward) to after the ‘then’ code to continue the program, as shown in Figure 19. On the ARM however this is not such a problem since conditional execution of several instructions in a row is often used to avoid the expense of a branch instruction. As a result, this case will not significantly harm the performance as might be expected. If emulating other systems without this benefit, some heuristic could be used to attempt to determine whether a piece of code fitted the style of an if-then statement. Features such as branching only a short distance might allow the forward branch to be included in the chunk.

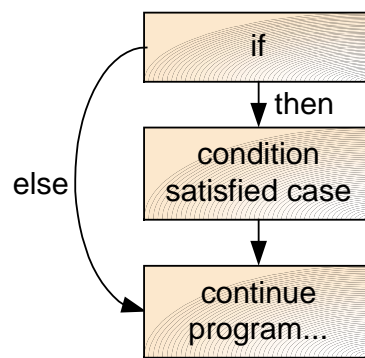


Figure 19 - The control flow of an if...then statement, showing forward branching

It is worth emphasizing that unlike the basic block defined earlier which ends on a branch whether conditionally executed or not, the chunk does not. When a conditional branch-outside-chunk occurs, if the condition is satisfied and the branch is to be taken, only then does the chunk end. Unconditional branches-outside-chunk, as for basic blocks, still end the chunk of recompiled code.

### 7.3. Chunk generation

In the early stages of the design, a separate profiler and analyser were planned. The profiler would decide the start and end of the chunk and the analyser would then separately translate the specified ARM instructions to armlets. However, it was found that much of the work done in decoding ARM instructions for translation was duplicated in identifying the chunk. As a result, the instructions are translated during the process of identifying the chunk, in a single pass over the code.

The dispatcher specifies the address of the first instruction to be recompiled and the profiler starts processing instructions from there. ARM instructions are dealt with one by one and decoded in a similar way to the disassembler, the interpreter's more intensive decoding approach being unnecessary. Decisions about which armlets to generate for a particular instruction are performed in much the same structure as the disassembler, except generating armlets rather than strings of disassembled text. As they are emitted, the armlets are simply appended to a linked list.

A hash table mapping from the address of each translated ARM instruction to the first armlet generated for that instruction is maintained. When translating any ARM instructions that branch backwards but within the chunk, the armlet that is the destination of the branch is located using this hash table. This destination armlet might normally be flagged and then patched up in a second pass over the generated armlets, so using this approach saves the expense of the extra pass.

Although a forward branch in an ARM instruction forces the chunk to end, forward-branches in armlets can still occur. The ARM's conditional execution of every instruction means that every instruction is wrapped in an implicit if-then statement, i.e.

```
if(condition true)
{
    execute ARM instruction
}
```

The consequence of the condition evaluating to false is that execution must jump (forward) over the armlets for this ARM instruction, introducing forward branches. This is a classic compiler problem and ordinarily would force a second pass over the generated armlets to 'backpatch' the forward-references once their destinations are known. For example, when generating the `goto` armlet in the fragment below, the recompiler can not know the location of the subsequent `mul` armlet to which it needs to jump (as it has not been generated yet). This detail needs to be 'filled in' after the armlets for the ARM `subeq` instruction have been generated.

Armlets	Generated from these ARM instructions
0: <code>movc t0,#2</code>	<code>add r0,r0,#2</code>
1: <code>add r0,r0,t0</code>	
2: <code>goto ?</code>	<code>subeq r2,r2,r0</code>
3: <code>sub r2,r2,r0</code>	
4: <code>mul r2,r3,r4</code>	<code>mul r2,r3,r4</code>

The solution devised is that when a conditionally executed instruction occurs a reference is stored to the `goto` that needs to be updated. When all the armlets for that instruction have been generated, the `goto` is then 'backpatched' with the location of the next armlet to be generated. Since these forward branches cannot be nested, this is an efficient and effective way of avoiding a naïve and expensive second pass over the generated armlets.

During the translation, trivial compile-time calculations are performed. For example, immediate values are expanded to their full 32-bit representation from the restricted encoding used in ARM instructions. Whenever the PC is used in an instruction the actual value of the PC is calculated based on the address of the instruction in memory and how it is accessed in the ARM instruction (which affects the pipelining influence on the value). The resulting value is then put into the `pc` armlet variable ready for use.

Temporary variables are liberally allocated from `t0` upwards as required and are often needed to hold an intermediate value in a calculation, for example the ARM instruction

```
mula r0,r1,r2,r3 ; meaning r0 = (r1 * r2) + r3
```

is converted to

```
mul t0,r1,r2
add r0,t0,r3
```

Note the temporary variable highlighted in red. Alternatively, temporaries are used to hold constant values for use in other armlets (recall the restriction that armlets use variables almost exclusively, described in section 6.4). At the end of the ARM instruction, the temporary variables are automatically deallocated and will be reused for the next ARM instruction.

Once an ARM instruction is translated, the routine that generated its armlets returns a value as to whether to keep recompiling or not. This is so that each ARM instruction's armlet-generation routine can decide whether emulating that instruction necessitates ending the chunk. This might be because the armlets modify the PC, the ARM instruction is a branch, or some other instruction that prevents the chunk continuing. The overall recompiling routine then uses this decision to decide whether to continue the chunk. If the instruction is executed unconditionally and the armlet generation decides against continuing then the recompiler ends the chunk. If however the instruction is conditionally executed or the armlet generator decided in favour of continuing, recompilation continues with the next ARM instruction. This decision process is outlined in the flowchart shown on Figure 20.

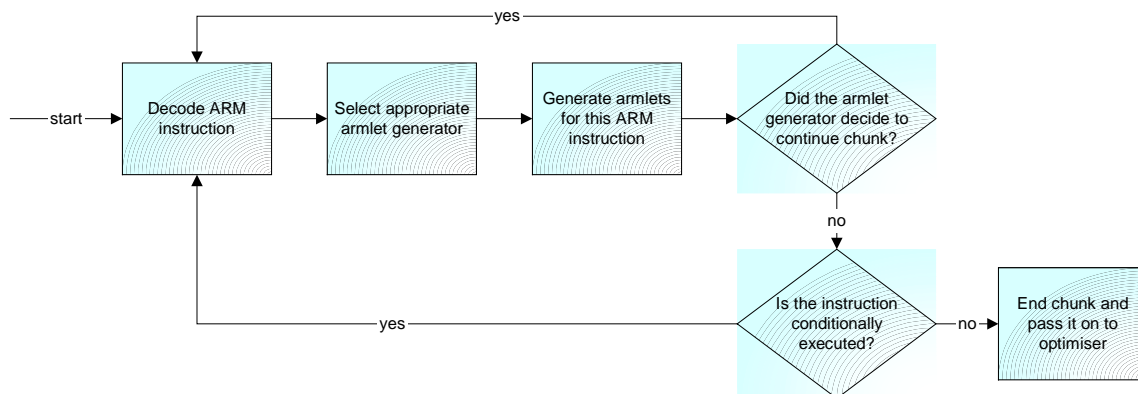


Figure 20 - Outline of the decision process concerning whether to end the current chunk

## 7.4. Testing

Verifying that the profiler generates the correct armlets is quite difficult. The first step was to develop an armlet disassembler so that the encoded armlets (which, for performance reasons, are not in a human-readable form) could be displayed in testing. The design of the armlet disassembler is similar to the ARM disassembler and the simpler format of armlets made it possible to debug this manually. Using the armlet disassembler, comparison between the ARM interpreter's implementation and samples of generated armlets was possible to look for obvious problems.

Probably the best way to debug the profiler would be to create an armlet interpreter that emulates armlets. Emulating the armlets using the armlet interpreter and comparing the results to those of the ARM interpreter would highlight any bugs in the generated armlets. Unfortunately, time constraints and the sheer scale of development needed to develop such a system prevented its implementation.

## 7.5. Unrecompilable code

Although all ARM instructions can be converted to armlets and x86 code, there are several cases where it is not practical to continue to emulate in recompiled code and execution must return to the dispatcher and interpreter. In these cases, a `leave` armlet is executed and a reason code passed back to the dispatcher so that it knows the reason for leaving the chunk and can act appropriately.

### 7.5.1. Processor mode change

The changing of the current processor mode is one such reason for leaving the recompiled code. The ARM3 processor has 4 different modes of operation: user, fast interrupt, interrupt and supervisor. Each of these modes has its own register bank for `r13` (the stack pointer) and `r14` (the link register) so that a change of mode can happen without affecting normal program operation. Additionally Fast Interrupt mode also has `r8` to `r12` in its own bank in order to minimise the save and restore costs associated with this kind of interrupt [66]. Only one set of registers 0 to 15 is accessible to the program at one time, depending on which mode the processor is currently in. The banking system and all physical registers along with which modes they are available from are shown on Figure 21.

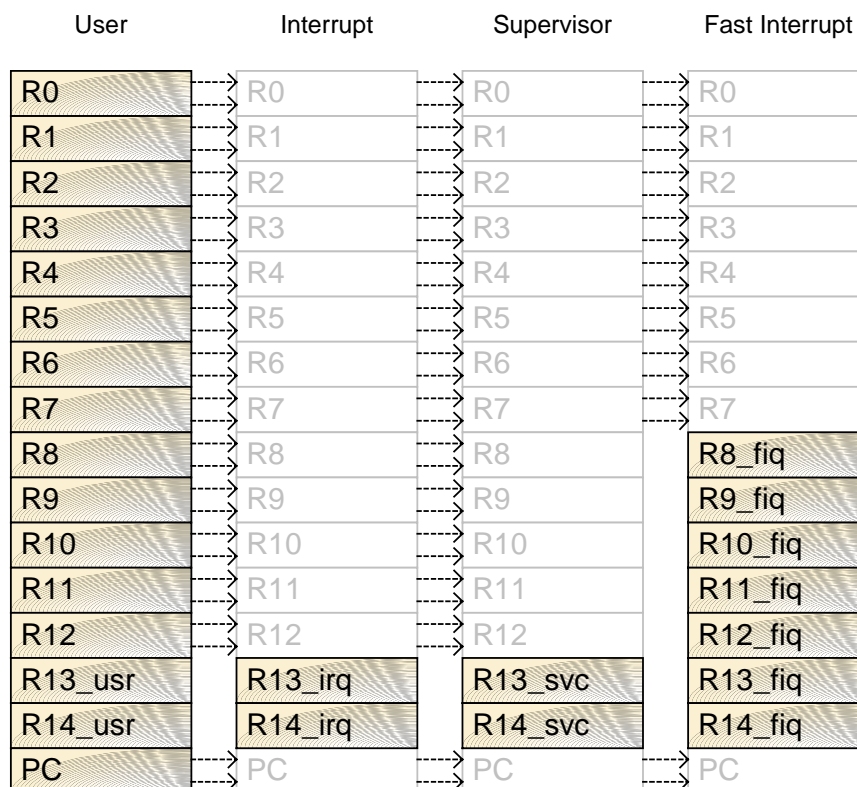


Figure 21 - The ARM processor modes and their register banks.

In the initial design, all the physical registers from all the different banks were to be valid armlet variables and therefore accessible to any armlet irrespective of the current processor mode. This would mean that a chunk would be 'hardcoded' to access the actual physical register for the processor mode that the emulated ARM was in at



compile-time. Unfortunately, there are sections of ARM code that can be executed in several processor modes. As a result the only way that registers could be ‘hardcoded’ as before was if a different section of recompiled code were compiled for every processor mode, and only the appropriate section for the current mode executed each time. Apart from taking four times the memory space and increasing the compile time for some sections of code, there is the pathological case that the mode could change inside the chunk causing this approach to break.

One possible solution was to still allow access to all the physical registers as armlet variables but to have a ‘current register’ bank used for all normal operations. On a mode change, the values of the new mode’s registers are transferred to the current bank and then swapped back out when the mode changes again (as is done in the interpreter). However, the amount of code required to handle all possible mode changes, swapping up to seven registers, would result in enormous sections of code (as can be seen from the source code in method `CARM::setProcessorMode()`).

The other possible solution was to have a level of indirection on every register access to point to the current register bank. This would result in the changing of the register bank being extremely fast (just changing the index for the indirection). Unfortunately, the added cost incurred by every single register access would significantly harm performance.

The chosen method was to make all generated code mode-neutral so that it could execute no matter what the current processor mode was. This is done by having a current register bank and having to drop back to the dispatcher on a mode change so that it can swap different modes registers in and out of the current bank, leaving the recompiled code unencumbered with this. Although it seems a shame not to recompile the mode-switching code, the number of times that it is possible that a mode change could occur would drastically increase code size.

Consideration was given to having a single ‘hand-coded’ subroutine to handle mode changes that could be called from within generated code whenever necessary, to avoid dropping back to the dispatcher. However, since mode changes largely occur on exceptions which cause a branch to another location (such as a handling vector) this leaves the chunk anyway. As a result, the overhead of dropping back to the dispatcher is inevitable and makes externally handling the mode change acceptable.

### 7.5.2. Exceptions

When an exception occurs in an ARM processor, the processor mode is changed, often to supervisor mode though alternatively to one of the interrupt-specific modes. The PC is adjusted to point to the exception’s vector address that normally contains a branch to the operating system’s routine to handle the exception.

Either a processor mode change or a branch-outside-chunk is sufficient to cause execution to leave the recompiled chunk for the reasons already described. As a result, exceptions will cause a `leave armlet` to be generated and the emulation of the exception is then handled by the dispatcher.

### 7.5.3. Memory access

Memory access is another part of the emulation that is not inlined into the generated code. The MMU emulation is quite lengthy and to handle this in inlined code would result in unacceptably long sections of generated code. Additionally if the ARM emulation is to be used for other ARM-based systems in the future, it makes no sense to inline MEMC emulation into the generated code.

The alternative is to leave the MMU emulation external to the ARM emulation as is done with the ARM interpreter. Memory access is performed via one of several armlets to load or store a word or byte quantity. Before the memory is accessed, an address-exception-check is performed in the generated code. This verifies that the address is inside the 26-bit address space and if it is not, leaves the chunk with a reason code signalling that an address exception has occurred and needs to be emulated.

Each memory-accessing armlet specifies a variable containing the logical address to be accessed, a variable for the data to be written to or read from memory and a temporary variable to receive a success flag. The success flag is a value returned from the MMU emulation denoting whether the memory access was a success. The checks for data abort exceptions (when an address attempts to access a non-existent page) are all handled by the MMU emulation. If the success flag denotes failure then the chunk leaves, signalling that a data abort exception has occurred and needs to be emulated by the dispatcher. If the success flag shows that the memory access was successful then the recompiled code continues unaffected. The armlets generated for the ARM instruction

```
ldr r0,[r1,r2]
```

are as follows

```
add    t0,r1,r2                ; add r1, r2 to get address
movc   t1,#0xfc000000          ; test address exception
and    t2,t0,t1
movc   t1,#0
cmp    t2,t1
gotoeq addressOk               ; if ok then skip leave
movc   pc,<address+8>           ; else leave code
interrupt check code          ; check for interrupts
leave  leaveAddressException

.addressOk
ldw    r0,t0,t3                ; load word from memory
movc   t4,#0                   ; test data abort exception
cmp    t3,t4
gotoeq noDataAbort            ; if ok then skip leave
movc   pc,<address+8>           ; else leave code
interrupt check code          ; check for interrupts
leave  leaveDataAbortException

.noDataAbort
```

The complexity and variety of block data transfer instructions is an order of magnitude greater than for single data transfer instructions. Many ARM data manuals are vague about the consequences of an exception occurring part way through the

instruction execution and various other boundary cases. Breaking down the execution into several stages of individual memory accesses and exception checking is one of the strengths of the intermediate code approach.

#### 7.5.4. Interrupts

The approach taken in the ARM interpreter of updating the IOC clocks and checking for interrupts before every instruction is overly expensive. Experiments with the ARM interpreter demonstrated that this was unnecessary and showed that intervals of tens of instructions caused no noticeable difficulties, though this does vary according to the ‘behind the scenes’ timing code in Red Squirrel.

Although it is desirable for generated code to only check for interrupts every few instructions, it is important that code is not executed for long periods without performing this check. Indeed some sections of the RISC OS ‘Power On Self Test’ are known to infinitely cycle in small loops waiting for interrupts to occur in order to test the hardware’s timing. If interrupts were not checked within this loop, it is conceivable that the emulation could enter an infinite loop. The solution is for an `intcheck` armlet to be generated immediately before any `leave` armlet or `branch-inside-chunk`, to prevent such infinite loops.

The `intcheck` specifies the number of instructions that have just been emulated and a temporary variable for the result of the check. This armlet causes the IOC clock to be ‘ticked’ the appropriate number of times for the number of instructions that have been emulated. If an interrupt is generated it is signalled in the result variable. The result variable is then examined by the generated code and if an interrupt has occurred, the chunk leaves, signalling the interrupt. An interrupt is just a special case of an exception and follows the same approach as other exceptions, dropping back to external code to handle it. The armlets generated to perform an interrupt check are as follows

```
intcheck t0,#X          ; update IOC to check interrupts
movc     t1,#0           ; test success flag
cmp      t0,t1
gotoeq   noInterrupt
leave    leaveIntCheck   ; leave if interrupt occurred
.noInterrupt
```

#### 7.5.5. Software Interrupts

Software Interrupts, or SWIs, are a method for programs to call operating system routines. These routines can be anything from writing a character to the screen or resetting the computer to resizing a window or accessing a floppy disk drive. When executed, a SWI instruction causes a Software Interrupt exception to be thrown and execution jumps to the SWI vector. Control then goes to the Operating System’s SWI handler, which decodes the SWI and calls the appropriate routine for the command. The result of this is that an exception occurs (with the resulting mode change and `branch-outside-chunk`) which forces the chunk to leave.

This approach allows the dispatcher to ‘properly’ emulate the exception, SWI handler and SWI code, or alternatively to jump to a routine that emulates just the high-level

behaviour of the SWI. In this way, the actions that the SWI instruction implies, such as opening a window, can be mapped to a completely different operating system's equivalent code. This allows much software written on one operating system to run on another. This approach has been used successfully for the WINE [67] system to run Windows software on Unix and attempted by Riscose [68] to support RISC OS software on Unix.

#### 7.5.6. Coprocessors

An ARM coprocessor emulation can range from the almost trivial cache control systems to full IEEE specifications of floating point mathematics. When an ARM coprocessor instruction is executed on the real hardware there are two possible scenarios. If the coprocessor is present the instruction is executed by the coprocessor and the ARM updated as necessary, otherwise an 'undefined instruction exception' is thrown which causes a jump to an operating system routine to handle it. Normally the OS routine emulates the coprocessor hardware, leaving the program containing the instruction unaware that the coprocessor does not actually exist (other than the instruction taking substantially longer to complete).

The ARM interpreter was used to investigate the occurrence of coprocessor instructions in RISC OS. It was found that the system control coprocessor is of little consequence as it is used only seldom in the early stages of loading RISC OS. The other coprocessor, for floating point arithmetic, is used more frequently and has several instructions executed at regular intervals (every few thousand instructions) throughout RISC OS. Since the floating point coprocessor is regularly emulated in software on the real system, an undefined instruction exception occurs for each of these instructions. It is feasible that emulations for this and other unknown coprocessors will be implemented in the future and so some generic interface is required. By far the best way of handling this is again to leave the chunk with the `leave armlet` specifying a coprocessor instruction as the reason. In this way, external code can invoke a coprocessor emulation if it exists, or as normally occurs, generate the undefined instruction exception which necessitates leaving the chunk anyway.

Although several cases have been described which force the chunk to leave, in practice these normally occur seldom enough to allow an adequate size of chunk. Generally the benefit of handling these situations like this is that the size of the code generated is minimised and the flexibility of the recompiler improved.

## 8. Code Optimisation

### 8.1. The optimiser

There are three stages to the dynamic recompilation. The initial stage is the conversion from ARM instruction to armlets, performed by the profiler. The final stage is the x86 generator where equivalent x86 code for the armlets is generated. There is another stage in between these two that has the sole purpose of optimising the armlets. This structure is summarised in Figure 22.

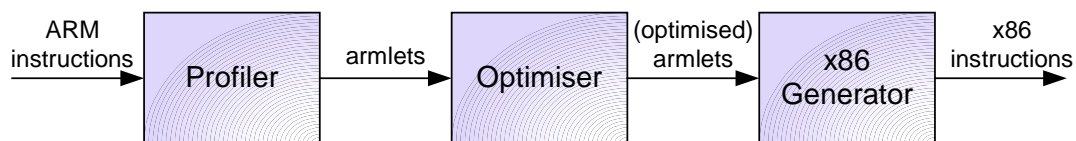


Figure 22 - The three stages of dynamic recompilation

Despite the initial appearance, code generation and optimisation are not completely disjoint and it is often more efficient to perform optimisations during code generation than separately. As a result both the profiler and the x86 generator implement appropriate optimisations.

### 8.2. The nature of the source code

Optimisations performed by a dynamic recompiler are generally very different to those performed in a traditional compiler. The main reason for this is the obvious difference in the source code being compiled. In a high level language, programs are rarely written with the underlying architecture in mind. For example, a calculation in a loop might repeatedly calculate the same value or an exponential function might be used where a simple multiplication will suffice. Traditional compilers are largely concerned with optimising these issues to improve the quality of the original program.

However, in recompiling machine code, the program being recompiled has (one would hope) already been optimised by a compiler (or manually by an assembly programmer), removing all of these high level inefficiencies. If such optimisations were performed in the recompiler, the benefit gained by the small amount of code that was not previously optimised would be dwarfed by the cost of the recompilation for code that had been. Rather, the problem for a recompiler is that the original compiler is likely to have gone further than just improving the program's algorithmic strengths and will have taken advantage of features of the source architecture which may be inefficient to perform directly on the target platform. An obvious example in the ARM to x86 case is the ARM's conditional execution of every instruction, which the x86 does not support.

This use of architecture features is even truer of the ARM platform than most. Uncharacteristically for a RISC processor, where the assembly language is often a maze of pipeline hazards [69], the orthogonal ARM assembly language is almost a joy to program in. This combined with a BASIC interpreter and assembler built into RISC

OS means that many programs for the platform are written in assembly language. This has been taken to unusual extremes with many complex GUI-based programs being written entirely in ARM assembly. On top of this, the recent market for ARM processors in embedded systems has encouraged the development of programs written in assembly in order to minimise the memory taken up by the program (and therefore the costs of the ROM to store it). As a result of the extensive use of assembly language, idioms of the architecture tend to be utilised more often than might be the case for a largely high level language platform. It is optimisations of these features that tend to be the concern of recompilers.

### 8.3. The requirements of optimisation

Since the recompilation affects the performance of the overall system, these optimisations must be performed as quickly as possible. This is in stark contrast to traditional programming language compilers, where the quality of the code generated is the overriding concern and an increased cost in compilation is acceptable for a gain in the quality of generated code. In dynamic recompilation, there is a trade-off between improving the quality of the generated code and keeping recompilation time as brief as possible, in order to get the best overall performance.

As a result of the speed requirement of optimisations, any techniques that require much memory or complex data structures to represent them tend to be ruled out. The remaining optimisations tend to be what are known as ‘peephole’ optimisations [70]. These are techniques where only a short sequence of instructions is examined and replaced with a faster sequence where possible.

The other major contrast to traditional compilers is that optimisations have to be performed only on local code. In a normal compiler, optimisations over large sections of the program are possible. However, in a dynamic recompiler, only the chunk being currently recompiled is optimised and no account is taken of other chunks. The reason for this is mainly speed, as scanning larger amounts of code takes longer. Additionally, as only one section of code is being recompiled at any one time the compilation of different chunks is inherently quite separate.

### 8.4. Traditional compiler optimisations

Many code optimisations in the compiler literature were investigated to try and identify any that might be used in the system. The following common techniques were rejected on the grounds that they are likely to have already been performed on the ARM code and so would have little benefit.

#### **Algebraic simplification**

For example,  $x=x*1$  can be removed. [71]

#### **Strength reduction**

For example  $x=x*9$  is equivalent to  $x=x+(x<<3)$ , this optimisation is particularly common on the ARM with its barrelshifter which was originally significantly faster than the `mul` instruction. [72]

#### **Common subexpression elimination**

Reusing results from identical calculations rather than recalculating them. [73]

**Induction variables**

Simplifying variables that vary by a constant amount in each loop iteration. [74]

**Loop unrolling**

Replace each iteration of a loop with inline code for that iteration. [75]

**Function inlining/Procedure integration**

Remove the overhead of a procedure call. [76]

**Trap elimination/Array bounds checking**

Remove unnecessary safety checks on array element access. [77]

**Straightening**

Join unconditionally sequential basic blocks into a single basic block to remove branch overheads. [78]

There are several other traditional optimisations that initially seem relevant but have not been implemented for various reasons. One such case is that of machine idioms [79] which can be made use of to convert a fairly generic instruction or series of instructions into a single target instruction. For example with multiplication being relatively expensive, cheaper combinations of adds and shifts were regularly used on the ARM. For example, multiplying a register by 9 would be performed by the following ARM instruction

```
add x,x,x,lsl #3
```

Translated ‘as is’ this instruction would be relatively expensive on the x86 and should ideally be recompiled to a single x86 `mul` instruction. Unfortunately, checking for these special cases would slow down the generation of other more typical cases and so cannot be used.

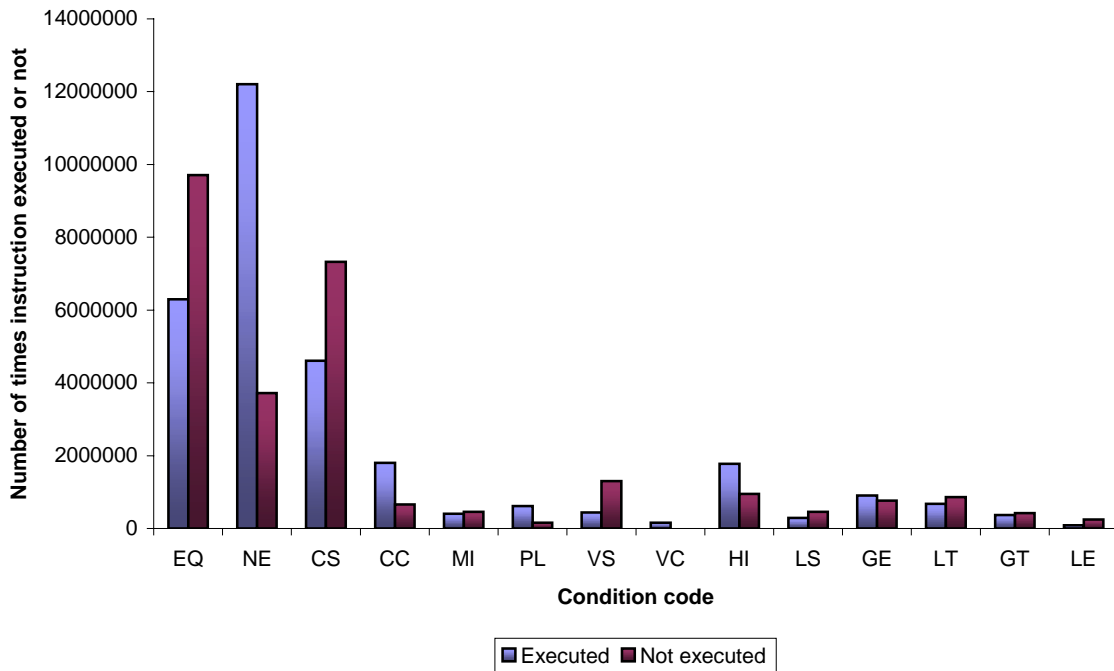


Figure 23 – Observed execution statistics for conditionally executed instructions.

Another similar case is the possible use of the conditional `move` instruction on the x86 [80] to mimic the ARM's conditional `move` instruction. These have not been utilised in the target code because as well as being detrimental to generating code for the more general case, it would harm other more powerful optimisations of conditional ARM instructions that have been employed (see section 8.6).

Although the x86 platform does not support static branch prediction, other architectures such as SPARC-V9 and the RIOS version of POWER do [81]. The ARM interpreter was used to analyse the proportion of conditionally executed ARM instructions that are executed or not in RISC OS use. The findings, as shown on Figure 23, were that some condition codes have a clear trend as to whether they are executed or not. Therefore the recompiler may benefit from supporting static branch prediction code if targetting architectures that make use of it. However, since the x86 would not benefit, the implementation of such a system at present would be likely to slow the recompilation.

## 8.5. Java run-time optimisation techniques

Java Virtual Machines that employ dynamic recompilation techniques also suffer the problem that optimisations must be as fast as possible. Predictably, JVMs also use many traditional compiler techniques, such as constant folding [82] and common subexpression elimination [83], although generally avoid the more expensive optimisations.

One optimisation that is very popular with JVMs is method inlining. The Java documentation [84] encourages the extensive use of accessor methods to object member variables (i.e. 'get' and 'set' methods), resulting in a large number of methods containing relatively little code. Java employs dynamic class loading and it is common for subclasses to be loaded, which may create an alternative version of a method [85] (for the subclass). As a result, methods cannot be inlined when the program is compiled to bytecode, causing inlining methods at run-time to significantly improve performance. Real machine code does not suffer such issues and many resulting dynamic JVM optimisations (such as dead code elimination [86]) are not relevant to dynamically recompiling real machine code.

## 8.6. Conditional Blocks

In attempts to avoid relatively expensive branches, it is common for several consecutive ARM instructions to be conditionally executed. So for example, the following C fragment

```
if(x != 0)
{
    primaryCounter++;
    secondaryCounter+=2;
}
```

might compile to these ARM instructions

```
cmp r0,#0
addne r1,r1,#1
```



```
addne r2,r2,#2
```

Since the x86 processor does not support conditionally executed instructions to the extent that the ARM does, a conditional branch has to be inserted before each armlet (shown in blue below) to check the condition and skip the instruction if it is not met. A naïve dynamic recompiler would then generate the following armlets

```
[xxxx->xxxx] movc t0,#0
[xxxx->NZCV] cmp r0,t0
[xZxx->xxxx] gotoeq afterFirstAdd
[xxxx->xxxx] movc t0,#1
[xxxx->xxxx] add r1,r1,t0
.afterFirstAdd
[xZxx->xxxx] gotoeq afterSecondAdd
[xxxx->xxxx] movc t0,#2
[xxxx->xxxx] add r1,r1,t0
.afterSecondAdd
```

It should be obvious that checking the condition for every ARM instruction in the conditional block is unnecessary since the results of all subsequent condition checks will be the same as the first. This is provided the instructions in the block do not affect the condition flags, which in this and many cases they do not. As a result, a conditional block can be used so that the following armlets are generated where the condition is only checked at the start of the block

```
[xxxx->xxxx] movc t0,#0
[xxxx->NZCV] cmp r0,t0
[xZxx->xxxx] gotoeq afterConditionalBlock
[xxxx->xxxx] movc t0,#1
[xxxx->xxxx] add r1,r1,t0
[xxxx->xxxx] movc t0,#2
[xxxx->xxxx] add r1,r1,t0
.afterConditionalBlock
```

This optimisation is performed by the profiler during the generation of armlets from ARM instructions. For each ARM instruction, the condition code is examined and if it is a different condition code to the previous instruction then a `goto` armlet with the inverse condition code is generated before the instruction's armlets. The destination of that `goto` is not known at the time it is generated, so a reference to it is held for backpatching. All subsequent ARM instructions with the same condition code are then generated without the conditional check. When the condition code changes, the initial `goto` is backpatched to point to after the conditional block. Other circumstances such as an instruction that adjusts the condition flags, or a branch inside the chunk, also cause the conditional block to end and a new one to start with the an additional condition check necessary. Conditional branches inside the chunk have to be handled as a special case since they can be recompiled to a single conditional branch on the x86.

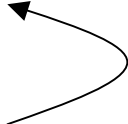
One complication was that a later branch-inside-chunk could branch into the middle of a conditional block, avoiding the initial condition check. Only performing conditional blocks on basic blocks was considered but this would limit the benefits gained. Instead, details of the start and end of each conditional block are kept and any later branch-inside-chunk instructions that are generated are checked to see if their

destination falls inside a conditional block. If it does, then an additional condition check is inserted at the destination so that the emulation is correct. For example, the following sequence of ARM instructions (with the branch destinations shown by the arrows and the conditionally executed instructions in red)

```

0x0:  cmp      r0,r1
0x4:  addgt   r2,r2,#0x1
0x8:  addgt   r3,r3,#0x2
0xc:  addgt   r4,r4,#0x4
0x10: sub     r1,r1,#0x1
0x14: cmp     r1,r2
0x18: bne     0x8

```

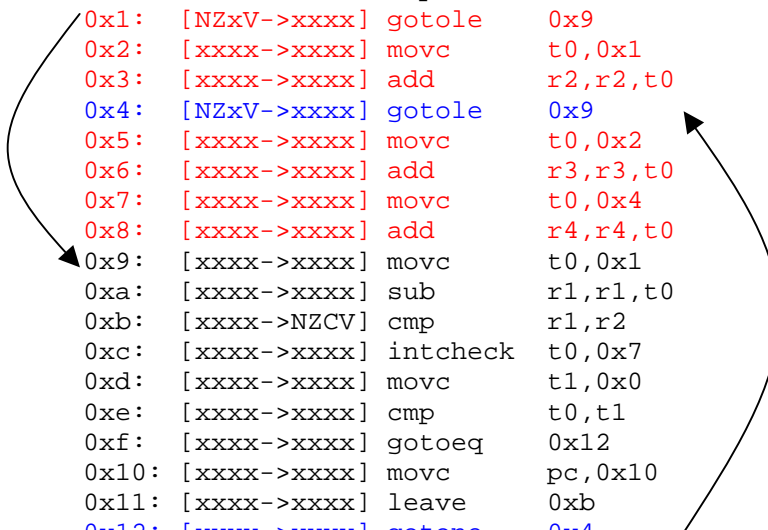


are translated into the following armlets, with the additional `gotole` armlet inserted inside the conditional block as a result of the later branch.

```

0x0:  [xxxx->NZCV] cmp      r0,r1
0x1:  [NZxV->xxxx] gotole   0x9
0x2:  [xxxx->xxxx] movc     t0,0x1
0x3:  [xxxx->xxxx] add      r2,r2,t0
0x4:  [NZxV->xxxx] gotole   0x9
0x5:  [xxxx->xxxx] movc     t0,0x2
0x6:  [xxxx->xxxx] add      r3,r3,t0
0x7:  [xxxx->xxxx] movc     t0,0x4
0x8:  [xxxx->xxxx] add      r4,r4,t0
0x9:  [xxxx->xxxx] movc     t0,0x1
0xa:  [xxxx->xxxx] sub      r1,r1,t0
0xb:  [xxxx->NZCV] cmp      r1,r2
0xc:  [xxxx->xxxx] intcheck  t0,0x7
0xd:  [xxxx->xxxx] movc     t1,0x0
0xe:  [xxxx->xxxx] cmp      t0,t1
0xf:  [xxxx->xxxx] gotoeq   0x12
0x10: [xxxx->xxxx] movc     pc,0x10
0x11: [xxxx->xxxx] leave    0xb
0x12: [xxxx->xxxx] gotone    0x4

```



Notice how the ARM `bne` translates to a single conditional `gotone` armlet, not a conditional block. This optimisation is very effective as it is performed extremely cheaply in the same pass over the source code that is used to translate from ARM to armlets. The resulting code performs very well in handling a feature of the ARM architecture that could have become quite costly in translation to the x86.

## 8.7. Redundant condition flag calculation elimination

In emulating older processors, the fact that every logic and arithmetic instruction adjusts the condition flags meant that redundant calculations of condition flags are often performed, even when their results are not utilised [87]. In contrast to these older processors, the ARM's instructions can either set the condition flags or not. As a result, less time is wasted on emulating redundant condition flag calculations than is the case for these older processors. However, despite this, it is still common for redundant condition flags to be calculated unless removed.

This optimisation is applied by the optimiser to a chunk of armlets, after the translation from ARM instructions. It is only applied internally to basic blocks in order to keep the optimisation cost low. As a result, the leaders of the basic blocks

have to be identified by the optimiser, before attempting to remove redundant flags. Basic block identification is necessary for the code generation phase and so is not a significant cost increase. Once the leaders have been identified, the chunk is scanned from start to end, examining the inflags and outflags of each armlet.

If an armlet adjusts a given condition flag, that flag is recorded as having been changed unnecessarily. If a subsequent armlet then requires that flag as input, the record of it having been changed unnecessarily is erased. If however, an armlet is encountered that adjusts that flag while it is still recorded as having been changed unnecessarily, it is the case that the previous flag adjustment is redundant. Since that first changing of the flag is never used before it is changed again, the outflag for that previous flag-changing instruction can be cleared, removing that flag from the calculation without affecting the emulation.

For example, the following sequence of ARM instructions, that performs a 64 bit addition

```
0x10: addS    r0,r0,r2    ; add low half of value
0x14: adc     r1,r1,r3    ; add high half of value
0x18: cmp     r0,#0x5
```

is initially translated to the following sequence of armlets

```
0x8:  [xxxx->NZCV] add     r0,r0,r2
0x9:  [xxCx->xxxx] adc     r1,r1,r3
0xa:  [xxxx->xxxx] movc    t0,0x5
0xb:  [xxxx->NZCV] cmp     r0,t0
```

However, of the four outflags from the `add` armlet, only the carry flag is used (by the `adc` armlet) before they are changed again by the `cmp` armlet. Therefore, the optimiser eliminates the redundant N, Z and V flags set by the `add` armlet, as shown in the following section:

```
0x8:  [xxxx->xxCx] add     r0,r0,r2
0x9:  [xxCx->xxxx] adc     r1,r1,r3
0xa:  [xxxx->xxxx] movc    t0,0x5
0xb:  [xxxx->NZCV] cmp     r0,t0
```

Depending on the final implementation of the flag calculation in native code, the savings made by reducing the number of flags that have to be calculated can amount to a considerable saving.

The x86 generator performs the remaining optimisations and it is most appropriate to describe them having examined its operation.

## 9. Native Code Generator

### 9.1. Overview

The purpose of the native code generator is to take a chunk of armlets and generate a chunk of x86 machine code that when executed emulates the actions of the armlets. In this respect, the native code generator has much in common with the backend of a compiler, converting its intermediate code to executable machine code. However, the native code generator is much more than a compiler backend as the armlets contain implicit information about condition flags.

### 9.2. Instruction selection

Appropriate x86 instructions are selected for each armlet. This can vary from a single x86 instruction to a long sequence depending on the armlet. In order to be able to generate any appropriate instructions with complete control, each armlet has a completely independent section of generating code to be tailored to its needs.

Although initially it might seem fairly straightforward to select the x86 instructions for an armlet, variations of a single armlet can be fairly complex. Initially a look up table decodes the armlet, invoking the appropriate routine. The contents of this routine varies greatly from emitting a relatively fixed sequence of x86 instructions for inflexible armlets (e.g. the `intcheck` armlet) to a large decision tree for armlets that have various combinations of operands (e.g. logic and arithmetic armlets). The main reason for this is the limitations of the x86's 2-address format instructions relative to armlets, as described in section 6.4. It was stated before that the conversion from 3 to 2 address code would need to happen in the native code generator and it is during the instruction selection that this occurs. For example, there are many different permutations of x86 instructions for the `add` armlet, depending on its operands, as shown in Figure 24.

Armlet	x86
op destination, operand1, operand2	op operand1, destination
add a,#,#	movl #,a
add a,#,a	addl #,a
add a,#,b	movl b,a addl #,a
add a,a,#	addl #,a
add a,a,a	addl a,a
add a,a,b	addl b,a
add a,b,#	movl b,a addl #,a
add a,b,a	addl b,a
add a,b,b	movl b,a addl b,a
add a,b,c	movl a,b addl a,c

Note, the labels, a,b and c denote different emulated registers and the # symbol denotes a known constant.

Figure 24 - Permutations of the `add` armlet and appropriate x86 instructions

The decision tree breaks down into a series of comparisons and when a match is found the routine to generate the appropriate x86 instructions is called. Figure 25 shows how matching based on initially the operation type, then the type of the operands for the `add` armlet, selects the appropriate code to be generated. The colours of the leaves of the decision tree group the code generation to show how similarities between the x86 instructions generated allows the same code generation to be used for certain permutations. This form of decision tree is used for many of the armlets, some such as `sub` are more complex than `add` because they are not commutative and so the generation routines selected are not duplicated as much.

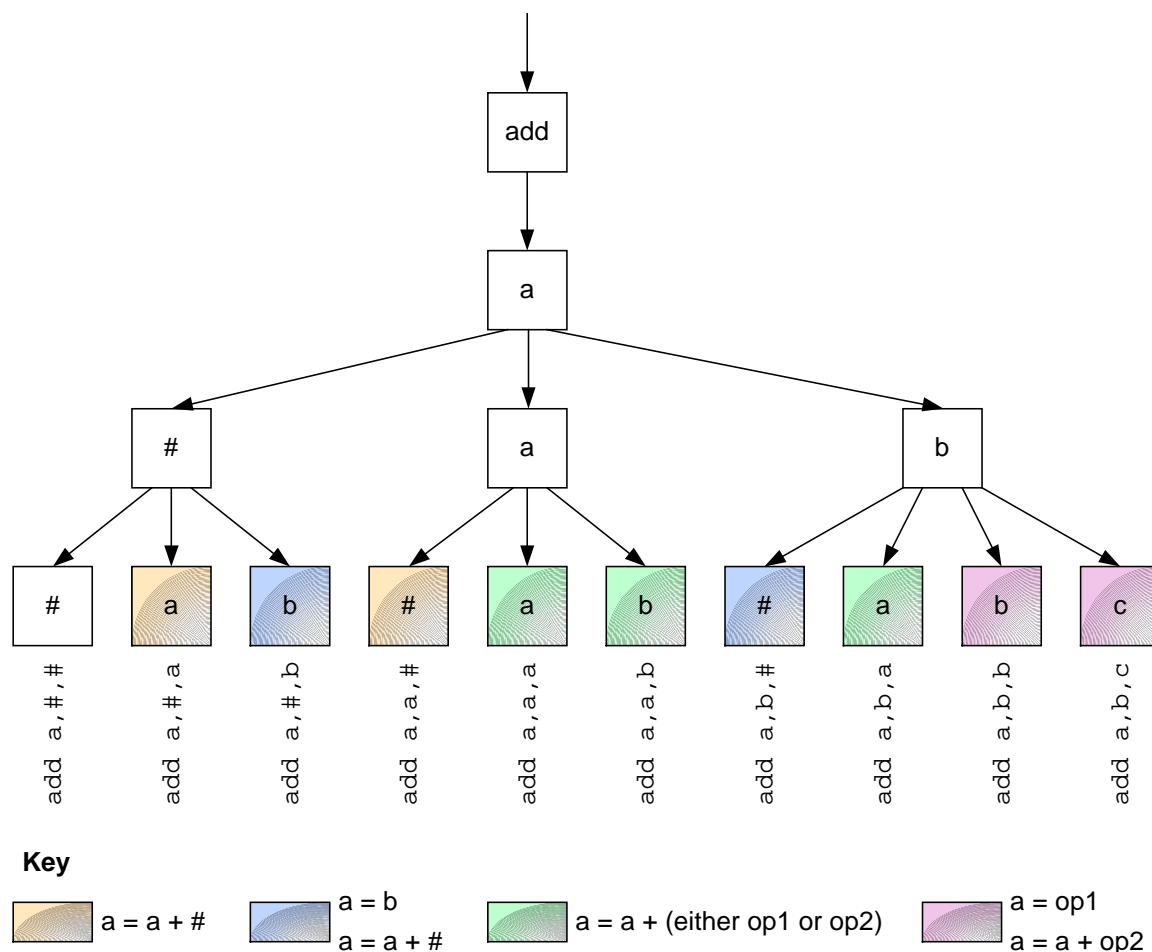


Figure 25 - Decision tree for x86 instruction selection

It would have been possible to have more generic templates, so that they could be applied to more permutations: however, this would have resulted in poorer quality code and only negligible improvements in recompile time. It would also have been possible to have increasingly specialised templates in order to produce more CISC-specific code for each case. For example, the x86 has an `inc` instruction which could have been generated in the case of an `add r0, r0, #1` ARM instruction. However, this would have slowed down instruction selection for any `add` armlet with a constant operand. Additional research showed that `inc` and `add` both execute in 1 cycle on modern x86 processors which would mean no speed optimisation (although the

instruction encoding would take 1 byte not 3). Additionally the `inc` instruction does not adjust the carry flag like the `add` instruction does which would actually slow emulation significantly when flags had to be adjusted.

### 9.3. Register allocation options

There were three main approaches considered for how to allocate armlet variables to native registers in order to increase the code execution speed by reducing accesses to relatively slow memory. The first of these is to dynamically allocate registers so that at every stage of a chunk any variable can be allocated to any native register. This is the approach taken by traditional compilers in order to get close to the best possible register allocation. Unfortunately as a side effect of dynamic register allocation, it would become significantly harder for a chunk to be re-entrant if required (i.e. enterable at different armlets), without extensive details of the register allocation at every instruction boundary being stored. Dynamic register allocation results in the best quality code but at the expense of extra work during recompilation.

Universal static register allocation was considered, whereby a few armlet variables are allocated to native registers for the duration of every chunk, with all other variables being stored in memory. In any traditional compiler this would be unthinkable; however, the savings in recompilation time for a dynamic recompiler are significant. Additionally when emulating architectures with few registers (e.g. older processors) on those that have many (e.g. modern RISC processors), it is possible to allocate all emulated registers to native registers making this the obvious choice.

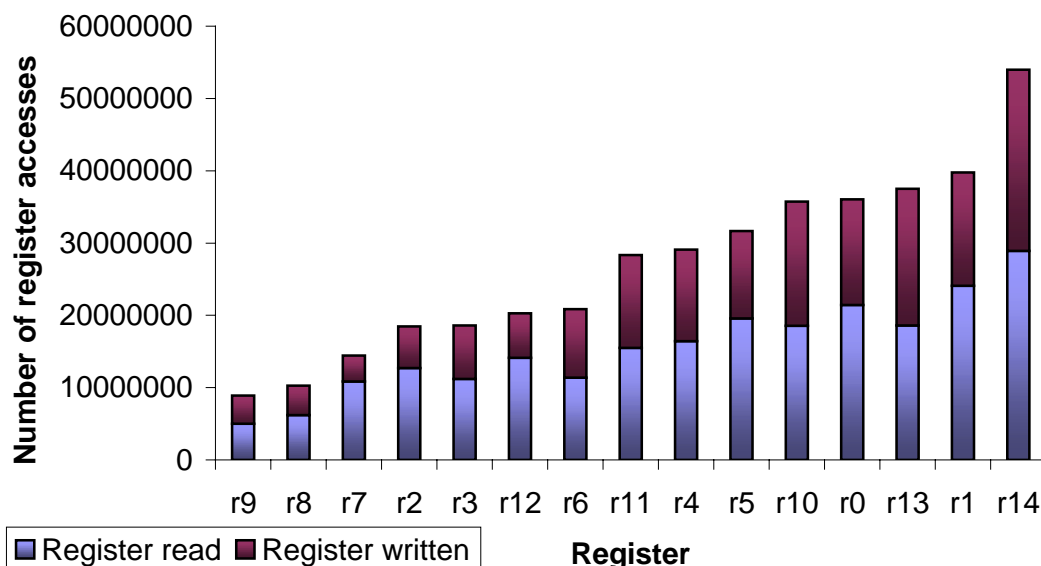


Figure 26 - Register usage in RISC OS

Although the ARM has more registers than the x86, in the event that certain registers were accessed significantly more than others, universal static allocation might be practical. Using the ARM interpreter to record the number of times each register is

accessed in emulating RISC OS, the results (excluding r15, the program counter) shown in Figure 26 were observed.

The most used registers are not surprising since the ARM Procedure Call Standard [88] (the convention for calling subroutines) defines specific uses for some of the registers, such as:

r14	- the link address register (for holding return addresses).
r0, r1	- scratch registers for subroutines.
r13	- lower end of current stack frame.
r10	- stack limit

This causes these registers to be used more often than others. Unfortunately to utilise universal static register allocation; the graph would have to be more sigmoid-shaped, with only a small number of registers being used significantly more than others. This is unlikely to occur for a processor with so many general-purpose registers.

The third option is to statically allocate registers within each chunk. This would require a pre-scan of the chunk before compilation, with the number of times each variable is used being recorded. The most used variables can then be statically allocated for the duration of the chunk. In this way each chunk would have its most used variables allocated to native registers throughout. Unfortunately this is likely to suffer the same problem as universal static allocation as some chunks would have no variables that are used a lot more than others, while also incurring extra costs in the recompilation from analysing the variable usage. When there are a relatively large number of general-purpose variables to be allocated static register allocation has a number of weaknesses so dynamic register allocation is used.

## 9.4. Dynamic register allocation

A dynamic register allocator allocates variables to registers until all the registers are allocated. When another variable needs to be allocated to a register, the problem is in deciding which variable should be displaced to memory. The aim is to choose the variable to be ‘spilled’ to memory to minimise the number of memory accesses. Perfect register allocation is an NP-complete problem [89] and so traditional compilers use heuristic algorithms to give approximate results. Unfortunately, these graph-colouring techniques [90] are still too expensive for use in a dynamic recompiler.

Consideration was given to naïve allocation algorithms that are very fast, keeping compilation time down. Choosing a variable to be spilled at random, although inevitably resulting in some very poor decisions, would be very fast to compile. Using a first-in-first-out approach to spilling would take slightly longer than a random decision during recompilation but seems less likely to make very poor decisions. Making a decision to spill certain variables based on the observed register usage in Figure 26 would be sensible. Given the choice between spilling r14 and r9 the register allocator might be predisposed to spill the least used. However, this might result in r14 being permanently allocated to a register, which may be undesirable. A least-recently-used algorithm, as implemented in processor caches, might have more

relevance to which register is likely to be used next, though still does no attempt to investigate future usage.

There are more complex techniques which attempt to give good approximations to the results possible through graph colouring. This is done using local (rather than global) register allocation based on the live ranges of variables (i.e. what parts of the code a variable is used in). Algorithms such as second-chance binpacking [91] and linear scan [92] are two such algorithms. These methods although promising for JVMs, are still relatively slow for use in a complete system emulator and so a different method has been implemented that is limited in its analysis (and so relatively fast) but makes fairly sensible decisions.

Operating only within basic blocks, the register allocation algorithm allocates a variable to the first register it finds that is not allocated. If all the registers are allocated, a simple lookahead system is used to assign a score to each x86 register. All the x86 registers are initially allocated scores of 0. If the variable held in a register is used in the next armlet the score is incremented by 2. If the variable held in a register is also used in the armlet after that the score is incremented by 1. Variables that are the destination of one of these two armlets get their score incremented again. The algorithm then scans through the x86 registers looking for a register with a score of 0 (i.e. a variable not used in either of the next two armlets), then 1, then 2 etc. In this way, the register chosen to be spilled is unlikely to be used in both the successive armlets and is certain to not be the destination of one of the next two armlets. This is done since the destination value for an armlet is likely to be used again soon. For example, the following section of armlets shows the scores associated with a subset of the armlet variables when evaluating each armlet.

sub r2,r2,r1	r0=5 r1=3 r2=1 t0=0
mov r1,r0	r0=5 r1=0 r2=2 t0=2
add r0,r0,r2	r0=2 r1=0 r2=0 t0=4
movc t0,#5	r0=3 r1=0 r2=0 t0=2
cmp r0,t0	r0=0 r1=0 r2=0 t0=0

With a register-starved architecture such as the x86, which has just 6 registers able to be used for storing armlet variables, frequent register spilling is almost inevitable and this algorithm makes a good attempt at minimising spillage in the short term.

## 9.5. Condition flag calculations

The x86 processor has condition flags which are adjusted in very similar ways to the ARM condition flags that need to be emulated. In fact the ARM's negative, zero, carry and overflow flags map onto the x86's sign, zero, carry and overflow flags as shown in Figure 27 in their positions in the respective flags registers.



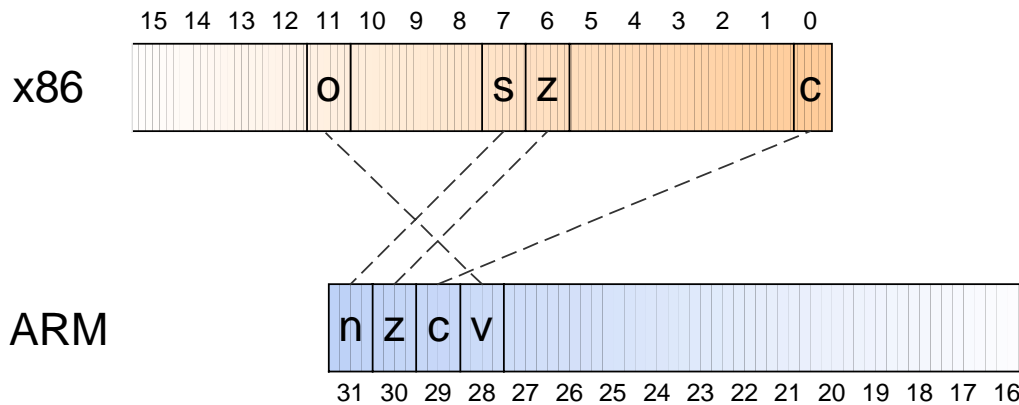


Figure 27 - The ARM and x86 flag similarities

By mapping the emulated flags onto the x86 flags, the complex explicit calculations highlighted in section 6.2 can be avoided for many armlets. The recompiler keeps track of where each of the emulated flags is currently being stored, either in memory or in the appropriate native flag. Inside each armlet's code generating routine the outflags are compared to the native flags that the x86 instructions will affect and any that should not be adjusted are spilled to the register file held in memory. Likewise, any inflags that are required can be put into the appropriate native flag if the x86 instructions are able to make use of it, or be held in memory or a register if it needs to be explicitly calculated. For example, the `gotone` armlet is recomplied to a `jnz` x86 instruction that requires the emulated zero flag to be the same as the native zero flag in order to work correctly.

Unfortunately, the x86 flags cannot be accessed as easily as the condition flags can be on the ARM. In order to get the values of the condition flags into the register file, the following sequence of instructions are required

```
sets    0x3d(%ebp)
setz    0x3e(%ebp)
setc    0x3f(%ebp)
seto    0x40(%ebp)
```

each of which sets or clears the appropriate byte in the register file (a memory block pointed to by the `EBP` register) according to the value of its condition flag.

Although the way that similar instructions adjust the condition flags is identical for many instructions, the subtract operations on the two processors set the flags in the opposite way. On the ARM the carry flag's effect on an `SBC` instruction is

$$\text{destination} = \text{operand1} - \text{operand2} - (1 - \text{carry})$$

while on the the x86, for the equivalent `SBB` instruction it is

$$\text{destination} = \text{destination} - \text{operand} - \text{carry}$$

This is solved fairly simply by inserting a `CMC` instruction in generated code before and after every subtract instruction, which inverts the carry flag. This is so that except for temporarily during the subtract instruction, the carry flag contains the value it would on an ARM.

## 9.6. Control Flow

The machine code generated rarely executes in a straight line despite the restrictions of the chunk definition. References are kept in a hash table for each armlet generated, mapping it to the first x86 instruction generated for it. Whenever a backwards-referencing `goto` armlet occurs, the address of the x86 code generated for the `goto` destination's armlet is looked up in the hash table, as shown in Figure 28. In this way the `goto` can be translated directly to the appropriate `jump` instruction, without needing a second pass over the code.

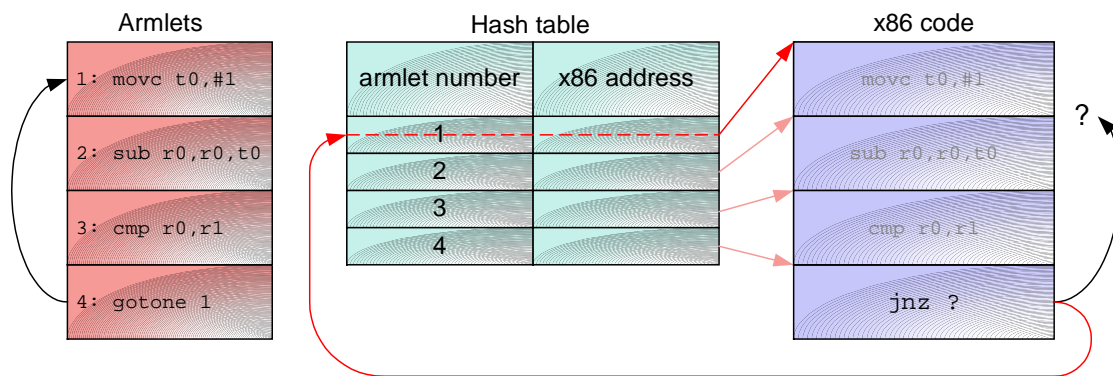


Figure 28 - Identifying the x86 address of a given armlet to avoid backpatching

Forward-referencing `goto` armlets have a record added to a linked list holding a reference to them and details of their destination. When recompilation of the chunk is complete, each entry in the linked list is backpatched with the correct destination address using the same hash table as for a backward-referencing `goto`. In handling `gotos` in this way there is no need for any second pass over the code to backpatch unknown values, making the recompilation more efficient.

There are further complications caused by changes in the control flow. Using dynamic register allocation means that the variables allocated to registers before a `goto` are unlikely to be the same as the variables allocated to registers at the destination of the `goto`. The solution to this is that whenever a forward-referencing `goto` occurs in a chunk, a copy of the details of the current register allocation is linked to the destination armlet of the `goto` (which will be by definition the leader of a basic block). When each armlet that is a leader of a basic block is about to be recompiled, the recompiler checks for an existing register allocation (from a previous forward-referencing `goto`) and if found, uses this register allocation, otherwise it continues with the previous armlet's allocation. Changing the register allocation requires that the old and new allocations are reconciled by spilling and then loading from memory the variables that change location between the two allocation descriptions. The

condition flags are included in this comparison and may also need to be spilled to memory.

The other occasion when the control flow is changed is when an armlet occurs that is handled externally to the recompiled code. For example, the `intcheck` armlet requires an external call to the IOC emulation, performed by calling a C function from the generated code, using the following x86 instructions

```
movl    $0x402b30,%edx    ; put function address in EDX
call    *%edx             ; call function at address in EDX
```

Any arguments that need to be passed to the C function (such as the number of ARM instructions emulated) and any results that need to be returned (such as whether an interrupt occurred) are stored in the register file in special locations, as the register file is accessible from both the C function and the generated code. In this way, generating the full code to handle function parameters, compatible with the code generated by the C compiler, is avoided. Fortunately the register allocation is not affected by this function call, and upon returning from the function, emulation continues unaffected. The chunk ends when a `leave` armlet is executed and this causes all register and flags allocated to the native facilities to be spilled back to the memory bank for returning to the main program.

## 9.7. Constants

The final optimisations, alluded to in section 8.7, involve the handling of constants in the x86 code generator. It is faster to evaluate a calculation only once at compile time, so that just the result is used at run time. This can be done if the result of the calculation is constant. This optimisation is known as constant expression evaluation or constant folding [93]. The other related optimisation is known as constant propagation [94] which is when a constant is assigned to a variable and then from there until the variable is changed again, all occurrences of that variable are replaced with that constant. A combination of these techniques is employed in the code generator within each basic block.

The `movc` armlet highlights all constants in the chunk. When a `movc` armlet is identified during recompilation, the variable and its associated constant value are added to a ‘constant pool’, replacing any previous entries for that variable. Any later assignments to that variable then invalidate its entry in the constant pool. When generating subsequent armlets, the instruction selection has a special case for when all the operand variables are in the constant pool, as shown for the `add` armlet on Figure 25. When this occurs, the recompiler performs the calculation and the destination variable of the armlet is added to the constant pool with its newly associated value. In this way, no code need be generated for constant expressions and the constant values propagate throughout the basic block. For example, the following sequence of ARM instructions:

```
mov r1,#2
add r0,r1,#123
sub r1,r0,#1
```

is translated by the profiler to this chunk of armlets:

```

movc r1,#2
movc t0,#123
add r0,r1,t0
movc t0,#1
sub r1,r0,t0

```

The relationship between these armlets is shown in Figure 29. As you can see the value of the expression is evaluated to 124 and no code need be generated at all.

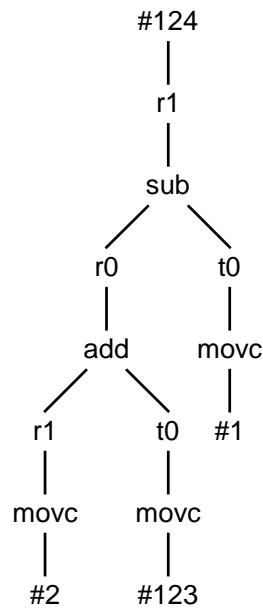


Figure 29 - The relationship between armlets in constant evaluation

In order to keep constant evaluation fast and simple, this optimisation is only performed within basic blocks. However, at the stage where the end of a basic block is identified by the code generator, it is too late to backtrack and insert code that actually put the constants into their variables. The solution is to effectively spill the constant pool to the code at the end of the basic block. Constant spilling is performed by generated code that moves constants into the appropriate armlet variables, whether they are held in native registers or memory. The other interesting scenario is where armlets that evaluate to a constant expression additionally adjust the flags. The solution is that code may be generated to adjust the flags but not actually calculate the result of the calculation, which is in the constant pool.

Ordinarily all possible constant evaluation will be performed when the program was originally written, however there are some interesting features of the ARM that prevent this from being the case. The problems of immediate value encoding on the ARM were mentioned in section 4.9. As a result of the limited number of bits in a fixed-length 32 bit instruction, immediates are encoded as a 4 bit rotate value and an 8 bit immediate, the actual value used being calculated by the following equation

$$\text{result} = \text{immediate ROR} (\text{rotate value} * 2)$$

This allows most of the commonly used values over a wide range to be encoded as immediates. However, some values, for example 0x12345678, cannot be encoded in this way and must either be loaded from memory or be formed from a sequence of ARM instructions as follows

```
mov r0,#0x12000000
orr r0,r0,#0x340000
orr r0,r0,#0x5600
orr r0,r0,#0x78
```

This entire sequence has no code generated for it at all by the recompiler as the constants are handled in the constant pool.

Another case generating a need for constant evaluation occurs as a result of the visibility of the PC. This was commonly used to perform PC-relative addressing to access values in a specific memory location. For example, the ARM instruction

```
ldr r0,[pc,#32]
```

loads the word from memory at address `pc+32` into `r0`. Since the PC can be determined at compile-time, the offset calculation can be too (when it is an immediate rather than a register) resulting in further savings. It is unclear whether constant evaluation and propagation would have such a significant impact when dynamically recompiling from other architectures, though it is certainly very useful for the ARM.

## 9.8. Emitting machine code

Emitting raw machine code quickly and in a programmer-friendly way is not simple. The fastest ways of generating machine code involve sprinkling the code generator with hexadecimal values of machine code, or at best C macros that evaluate to hexadecimal values, leaving the program code practically illegible and very hard to debug. The alternative of using a traditional assembler, parsing strings of assembly language, although very easy to read is unfortunately far too slow for a dynamic recompiler. There are third-party systems such as GNU Lightning [95] and the New Jersey Machine-Code Toolkit [96] that claim to generate machine code. However, in the time available, rather than learning a relatively complex system, something simpler was required.

Even once the complex decisions have been made about what code is to be generated, creating the software to emit machine code is laborious and time-consuming. Rather than implementing the machine code emitter myself and re-inventing the wheel, I have, with permission, used the run-time assembler from Julian Brown's ARMphetamine. This is a list of macros that define the encoding of around 300 different x86 instructions. So for example, the x86 instruction

```
cmp $0x1,%edx
```

has its encoding defined by the C macro

```
#define CMPlri 8,0x81,3,rm,3,0x7,2,0x3,32,imm
```

The macro name, `CMPlri` denotes that the macro is a `CMP` instruction that operates on longs (32 bit values), and takes a register and an immediate as operands. The subsequent numbers are pairs, the first being the number of bits and the second being the value to put into those bits, with `rm` and `imm` being variables for the register and immediate values that are specified by the code generation. Macros such as these, with their associated variables defined are then passed to an `assemble()` method that takes a variable number of arguments (in the same way as the C function `printf()`). This method then generates the correct values as defined by the macro into a stream of bits to make up the x86 code chunk.

## 9.9. Invoking native code

In order to invoke the generated machine code, the C++ program has to be able to call that code and then have it return back to the C++ program. This is made more complex by the fact that any registers used in the C++ program must be restored by the recompiled code before returning so as to not corrupt the program. Rather than write assembly code to manipulate the registers, it is better to make the C++ program treat the recompiled code as a C function. This is done by creating a type definition for a function, then casting the block of machine code to a function and finally calling that function in the same way as any other, as shown in the following code.

```
typedef uint32 (*codeInvoker)(Context* aContext);
codeInvoker callNativeCode = (codeInvoker)nativeChunk->area;
uint32 returnVal = callNativeCode(context);
```

A `Context` structure containing the emulated registers as held in memory is passed to the native code and the value in the `EAX` register, containing the reason code from the `leave armlet`, is returned when the chunk ends.

## 9.10. Debugging

Finding problems in the generated code is a slow and arduous process. An external x86 disassembler [97] is invoked by the program, which disassembles native code chunks directly and dumps the results to a text file. Actually studying the generated code looking for problems is extremely hard since the implicit workings of the condition flags are very difficult to follow. All that can be practically done is to take small sections of ARM code, emulate them on the interpreter then recompile them and search for any differences between the emulation by the interpreter and the generated code.

## 10. Conclusion

### 10.1. Evaluation

#### 10.1.1. ARM Interpreter

The ARM interpreter has been very successful and has achieved a near-perfect emulation of an ARM3 processor. There are still a few small glitches in the emulation that cause occasional high level issues. However, all of the measures of compatibility that were laid down at the outset have been achieved. The following screenshots show various examples of the ARM interpreter in action.



Figure 30 - The RISC OS initialising screen



Figure 31 - Screenshot of the !Draw program, written in C, that comes with RISC OS



Figure 32- Screenshot showing the RISC OS task manager and about box



Figure 33 - Screenshot showing the command line running ARM BASIC, written in ARM assembly

### 10.1.2. Profiler

The profiler is able to translate any sequence of ARM instructions into chunks of armlets as required. Even sections of code that deal with the trickier side of the ARM architecture such as coprocessors, SWIs and interrupts are all represented in the armlet instruction set. The conditional block optimisation that is performed during armlet generation is one of the strengths of the project and reduces one of the significant overheads in generated x86 code. The other strength of the profiler is in managing to translate ARM instructions to armlets using only a single pass over the code. This was something that was not originally thought feasible, hence the separate profiler and ARM analyser described in the progress report.



### 10.1.3. Optimiser

Optimising out any redundant flag calculations is a fast way to remove large amounts of wasted calculation from an explicitly calculating emulator. Although this technique has been employed successfully in the past with threaded interpreters [98] the benefits that it has for code that makes use of native flag calculation requires further investigation. The basic block identification performed by the optimiser is used both in the redundant flag elimination and in the code generation and is able to be identified quickly and easily and the data reused thanks to the internal representation used for armlets.

### 10.1.4. Native Code Generator

The native code generation performs complex tasks such as managing dynamic register allocation very well. Instruction selection is fast and flexible so that it can be manipulated to generate specific code for any variety of armlet. As a result, a single RISC ARM instruction is occasionally able to be translated to a single CISC x86 instruction. The techniques used are designed to be fairly generic and most deal largely with armlets. This means that were the system to be retargeted to another architecture most of the framework of even the code generator could be reused.

In the same way as the conditional block optimisation, the constant folding and propagation are very successful at eliminating some of the obscure traits of the ARM processor that would not normally recompile at all well to the x86. Examples of all the different classes of armlet are supported by the generator, including those that are performed by calls back to C functions. However, there are still several armlet code generation routines that need to be implemented. The dynamic register allocation sometimes makes bad decisions about when to spill temporary values to memory, only for them never to be used again. Currently all armlet variables are treated in the same way by the register allocator. However, it seems that temporaries should be handled differently, possibly with the profiler explicitly stating when the value a temporary is holding is dead.

### 10.1.5. Debugging tools

Developing tools to debug the actual system was a significant part of this project. The ARM disassembler worked very well and was able to be easily used in many parts of the system. It alone is a fairly complex program and is very competent at disassembling ARM code. The disassembler is fully functional with the only limitation being that coprocessor instruction disassembly was not implemented in the need to move on to developing the interpreter. This was not a problem as coprocessor instructions are used relatively infrequently and are handled externally to the ARM emulator.

A lot of effort went in to dynamic profiling of the ARM instructions being executed in RISC OS. Details about the use of exceptions, coprocessor instructions, SWIs, condition codes and register usage were invaluable in making implementation decisions. The significance of this information was relatively understated until publicly discussing such results lead to third parties requesting additional profiling for information to be used in their own ARM-based projects.

The armlet disassembler was used in a similar way to the ARM disassembler and was extremely important for comprehending the armlets that were generated. Despite having already implemented two disassemblers, the decision to use an external disassembler for x86 code due to the variety of the x86 instruction set was a wise one.

#### 10.1.6. Design

The design of the system has been very successful and has banished any concerns I had about the lengthy reading period prior to starting the ‘real work’. Despite initial doubts over whether to use an intermediate code at all, I do not believe it would be feasible to develop an optimising dynamic recompiler for an architecture of the ARM’s complexity without it. The intermediate code has allowed testing throughout the translation process, breaking down the problem into manageable sections and will ultimately allow for around 90% of the code, including all optimising routines, to be reused unchanged if the system is retargeted.

The decision to use an existing system emulator, Red Squirrel, was a very good one. It allowed concentration purely on the CPU emulation and provided a strong test bed able to support a full operating system for testing, also providing the best demonstration of the system working to support a full microprocessor emulation.

#### 10.1.7. Software engineering

The object-oriented nature of C++ made it easy to remain strictly disciplined in keeping the system modular and the different sections completely separate. Additionally the low level nature of C++ made the implementation of complex sections of ‘bit-bashing’ code relatively simple. With around 13000 lines of C++ written, using the Microsoft Visual C++ Development Environment made moving around the code during development easy as well as providing a good debugging environment that could interrupt program execution and inspect variables. This made the difficult problem of debugging the program slightly less painful than it might have been.

The waterfall model of system development, proved a good approach both to the overall project and to the implementation phase. Starting at the beginning of the system and developing through to the end, returning to previous stages as required was ideal for this one-man project where the different components were very separate systems in their own right.

The time available was managed well despite underestimating the difficulty of the project and the intrinsic complexity of practically every line of code. A detailed development diary was kept and is included with the source code for indepth information about how the project progressed. All six milestones that were decided upon at the start of the project [99] have been successfully met: something that part way through the development seemed unlikely.

## 10.2. Future extensions

Many features that I want to add to Tarmac have not been possible to implement in the time available. One such feature is the development of an armlet interpreter that would be used to verify that the armlets being generated are completely correct by emulating software using them. It could also be used to test the principles of the dynamic recompiler in a full system emulation before attempting the same thing with machine code. I would also like to experiment with developing a threaded interpreter based on armlets, which would result in a fast and completely portable ARM emulator.

More questions have arisen concerning optimisation than have been answered. It is still not clear how aggressive optimisations can be before the benefit in code quality is undermined by the expense of recompilation. The same can be said of the register allocation method. Though dynamic allocation generates the best code, it is difficult to know the possible benefits of alternative methods without implementing them. For these experiments to be possible, the dynamic recompiler has to be emulating real program code, something that it currently struggles with because of remaining glitches. It would certainly be an interesting and worthwhile project to investigate such dynamic optimisations.

Over the duration of this project, there have been many developments in ARM emulation. While writing this report, emulators for the Nintendo Gameboy Advance started to appear. With its simple code (a single ROM image), relatively simple address space, and 32-bit PC ARM processor, it is a far more attractive platform for experimenting with dynamic recompilation than a full RISC OS computer. Red Squirrel has advanced too and is now able to emulate later ARM processors that use a 32-bit PC. With this in mind, the simpler 32-bit-PC mode and the Thumb instruction set (used in the Gameboy Advance) are priorities to be added to Tarmac so that it can be used to fully emulate these systems.

With the completion of the x86 generator and dispatching framework I would like to retarget the ARM emulator, probably to another RISC platform. MIPS would be a good choice, as the lack of condition flags on that processor would highlight the benefits of the redundant flag elimination optimisation. Additionally it would be interesting to attempt to recompile from the 26-bit-PC mode ARM code to 32-bit-PC mode ARM code. This would be useful in supporting RISC OS on the newer range of ARM processors that no longer support 26-bit-PC mode.

Development of Tarmac is continuing, moving towards integrating it into Red Squirrel as that emulator's primary CPU emulation. This will provide a level of performance never before seen in ARM emulation and prove that running ARM software on other platforms at realistic speeds is possible.

## 10.3. Skills learned

The amount of learning required for this project was considerable. Quite apart from studying all available compiler and JVM techniques, an extremely intimate knowledge of the ARM processor had to be developed.

As well as the knowledge of techniques that was needed, programming in C++ and using the Visual C++ environment were both skills that I initially lacked. I have also learnt x86 assembly language from scratch.

## 10.4. Final conclusions

The release of several Nintendo Gameboy Advance emulators as this report is being concluded have reinforced the existence of the problem that projects such as this set out to solve. Despite having to emulate an ARM processor running at just 16MHz, these emulators run significantly slower than the real hardware on fast modern home computers.

This project has been successful in clearly demonstrating the techniques that can be used for a dynamically recompiling emulation of a modern microprocessor. I'm particularly proud of the innovative optimisations and their implementation. This has moved away from the traditional optimisations used in recompiling JVMs that are inappropriate for real machine code emulation. The actual method of translation, using a single pass over the source machine code, although going to great lengths to avoid a second pass, works very well. Many of these techniques have had to be developed from scratch or modified from traditional compiler algorithms.

The project has been an extremely large and difficult one by any standards though has also been very rewarding, in terms of both the material produced and the recognition it has received. Indeed, I am very proud that one of the designers of the original ARM processor came across this project which lead to an offer of a position developing a recompiling emulator for their new architecture.

The problems of emulating a real microprocessor using dynamic recompilation are far from solved, though I believe this project has made a contribution towards this goal.

## References

- [1] Nori, Ammann, Jenson, Nageli and Jacobi, *Pascal – The Language and Its Implementation*, New York: John Wiley & Sons, 1981.
- [2] Larus, *SPIM: A MIPS R2000/R3000 Simulator*, <http://www.cs.wisc.edu/~larus/spim.html>
- [3] Hennessy and Patterson, *Computer Organization & Design*, 2<sup>nd</sup> edition, Morgan Kaufmann, 1997, Appendix A.9.
- [4] ]SiMkIn[, *A 65816 Primer*, <http://www.zophar.net/tech/files/65816info.txt>
- [5] Klaiber, Transmeta Corporation, *The Technology Behind Crusoe<sup>TM</sup> processors*, <http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf>
- [6] Campbell-Kelly, *The EDSAC Simulator*, <http://www.edsac.net>
- [7] Sharp, *Manchester Baby Simulator*, <http://www.backlitgames.com/baby.zip>
- [8] Tasker, et. al, *Professional Symbian Programming: Mobile Solutions on the EPOC Platform*, Wrox Press Inc., 2000, p.743.
- [9] Wilkinson, *Computer Architecture: design and performance*, 2<sup>nd</sup> edition, Prentice Hall, 1996, p. 82.
- [10] Santayana, *The Life of Reason*, 1905, vol. 1, ch. 12.
- [11] Bell, “Threaded Code”, *Communications of the ACM*, 16(6):370–372, 1973.
- [12] Grune, Bal, Jacobs and Langendoen, *Modern Compiler Design*, John Wiley & Sons Ltd., 2000, pp.297-300.
- [13] Compaq, *How DIGITAL FX!32 works*, <http://www.support.compaq.com/amt/fx32/fx-white.html>
- [14] Connectix, *Virtual PC 4*, [http://www.connectix.com/downloadcenter/pdf/vpc4\\_faq\\_jan01.pdf](http://www.connectix.com/downloadcenter/pdf/vpc4_faq_jan01.pdf)
- [15] Ardi Ltd., *Executor Internals: How to Efficiently Run Mac Programs on PCs*, [http://www.ardi.com/ardi/executor\\_paper.html](http://www.ardi.com/ardi/executor_paper.html)
- [16] Apple Computer, Inc., *Technical Note PT39: The DR Emulator*, [http://developer.apple.com/technotes/pt/pt\\_39.html](http://developer.apple.com/technotes/pt/pt_39.html)
- [17] EmuUnlimited, *UltraHLE*, <http://www.emuunlim.com/UltraHLE/>
- [18] PSXEmu, *PSEmu Pro*, <http://www.psxemu.com/psemu.shtml>

- [19] EmuHQ, *Corn: n64 emulator*, <http://www.emuhq.com/corn/>
- [20] Cifuentes, Emmerik, Ramsey, *UQBT – A Resourceable and Retargetable Binary Translator*, <http://www.csee.uq.edu.au/~csmweb/uqbt.html>
- [21] Furber, *ARM system-on-chip architecture, 2<sup>nd</sup> edition*, Addison-Wesley, 2000, pp. 45-46.
- [22] Gilbert, *ArcEm*, <ftp://ftp.arm.uk.linux.org/pub/linux/arcem/>
- [23] Dorr, *ARM2*, <http://www.nvg.unit.no/bbc/emul/Arm2.zip>
- [24] Lloyd, *Archie*, <http://www.geocities.com/SiliconValley/Campus/5427/index.html>
- [25] Dales, *SWARM – Software ARM*, <http://www.dcs.gla.ac.uk/~michael/phd/swarm.html>
- [26] Barnes, *Red Squirrel*, <http://www.red-squirrel.org>
- [27] Brown, *ARMphetamine: A Dynamically Recompiling ARM Emulator*, <http://www.dynarec.com/~jules/>
- [28] Bloch, *Riscose*, <http://riscose.sourceforge.net/>
- [29] Rutter, *Sleeve*, <http://www.armlinux.org/projects/sleeve/>
- [30] Pajak, *ARM6 Model*, <http://www.comp.leeds.ac.uk/pajak/demo6.html>
- [31] Gordon, Pratt, Birtwistle, Hobley, *Formal Specification and Verification of ARM6*, 1999, <http://www.comp.leeds.ac.uk/graham/research/research.html>
- [32] ARM Ltd., *Semiconductor Partners*, [http://www.arm.com/arm/semicon\\_partners?OpenDocument](http://www.arm.com/arm/semicon_partners?OpenDocument)
- [33] Wilkinson, op cit., p.38.
- [34] Furber, op cit. p.37.
- [35] Furber, op cit. p. 38
- [36] Corlett, *Starscream 680x0 emulation library*, <http://www4.ncsu.edu/~nscorlet/star/>
- [37] Sweetman, *See MIPS Run*, Morgan Kaufmann Publishers, Inc., 1999, p.13.
- [38] Ibid., p.38.
- [39] ARM Ltd., *Jazelle<sup>TM</sup> – ARM Architecture Extensions for Java Applications*, <http://www.arm.com/armtech/jazelle?OpenDocument>

- [40] Furber, op cit., p.148
- [41] ARM Ltd., *ARM610 Data Sheet*, 1993, p.19.
- [42] Ponder, *Generator: A Sega Genesis emulator*, 1998, p. 21.  
<http://www.squish.net/generator/>
- [43] Vik, *StrongED*, <http://home.eunet.no/~guttorvi/strong.html>.
- [44] Sharp, *Goby: The Gameboy emulator for EPOC*,  
<http://www.backlitgames.com>
- [45] Sun Microsystems, Inc., *Writing Advanced Applications*, 2001,  
<http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/perf2.html>
- [46] Meloan, The Java HotSpot™ Performance Engine: An In-Depth Look,  
<http://developer.java.sun.com/developer/technicalArticles/Networking/HotSpot/>
- [47] Furber, op cit., p.360.
- [48] Furber, op cit., p. 79.
- [49] Bell, *Archimedes Elite*,  
<http://www.iancgbell.clara.net/clara.net/i/a/n/iancgbell/webpace/elite/arc/index.htm>
- [50] Stroustrup, *The C++ Programming Language*, 2<sup>nd</sup> edition, Addison-Wesley, 1991, p. 53.
- [51] Advanced RISC Machines, *ARM3 Datasheet*, issue 1.2, 1991, section 6.4.4, p. 26.
- [52] Jaggar, *ARM Architecture Reference Manual*, Prentice Hall, 1996, section 3-58.
- [53] Barnes, *Source code to Red Squirrel*, not publicly available.
- [54] Brown, op, cit.
- [55] Krall, “Efficient JavaVM Just-in-Time Compilation”, *Proceedings of PACT’98*, IEEE, 1998,  
<http://www.complang.tuwien.ac.at/java/cacao/index.html>
- [56] Cifuentes and Malhotra, “Binary Translation: Static, Dynamic, Retargetable?”, *Proceedings International Conference on Software Maintenance*. Monterey, CA, Nov 4-8 1996. IEEE-CS Press. pp 340-349.

- [57] VHDL International, *VHDL*, <http://www.vhdl.org>
- [58] Cifuentes and Mahotra, op cit.
- [59] Brown, op cit.
- [60] Aho, Sethi and Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley, 1986, p. 466.
- [61] Appel, *Modern Compiler Implementation in C*, Cambridge University Press, 1997, p. 336.
- [62] Fischer and LeBlanc, *Crafting a Compiler*, The Benjamin/Cummings Publishing Company, Inc., 1988, p. 247.
- [63] Aho, Sethi and Ullman, op cit., p. 528.
- [64] Brown, op cit.
- [65] Brown, *A neat hack for speeding up arbitrary flow control in dynamically-recompiled code*, <http://dynarec.com/~jules/livehash.html>
- [66] Furber, op cit., p. 313.
- [67] Julliard et.al, *Wine Development HQ*, <http://www.winehq.com>
- [68] Bloch, op cit.
- [69] Sweetman, op cit., pp. 29-31.
- [70] Aho, Sethi and Ullman, op cit., p. 554.
- [71] Fischer and LeBlanc, op cit., p. 569.
- [72] Muchnik, *Advanced Compiler Design & Implementation*, Morgan Kaufmann, 1997, p. 435.
- [73] Mycroft and Norman, *Optimising compilation – Part I: classical imperative languages*, SOFSEM '92 Czechoslovakia, 22.11.-4.12.1992.
- [74] Aho, Sethi and Ullman, op cit., p. 596.
- [75] Muchnik, op cit., p. 559.
- [76] Muchnik, op cit., p. 465.
- [77] Appel, op cit., p. 374.
- [78] Auslander and Hopkins, *An Overview of the PL.8 Compiler*, ACM, 1982, p. 29.



- [79] Aho, Sethi and Ullman, op cit., p. 557.
- [80] Muchnik, op cit., p. 591.
- [81] Muchnik, op cit., p. 598.
- [82] Arnold, Fink, Grove, Hind and Sweeney, *Adaptive Optimization in the Jalapeno JVM*, ACM SIGPLAN, Conference on Object Oriented Programming, Systems, Languages and Applications, October 15-19, 2000.
- [83] Krall, op cit.
- [84] Gosling and McGilton, *The Java Language Environment*, Sun Microsystems, 1995.
- [85] Sun Microsystems, *Method Inlining Example*,  
<http://developer.java.sun.com/developer/technicalArticles/Networking/HotSpot/inlining.html>
- [86] Muchnik, op cit., p. 582.
- [87] Ponder, op cit., p. 35.
- [88] Furber, op cit., p. 177.
- [89] Muchnik, op cit., p. 482.
- [90] Grune, Bal, Jacobs and Langendoen, op cit., p. 360.
- [91] Traub, *Quality and Speed in Linear-Scan Register Allocation*, Harvard College, 1998. <http://www.eecs.harvard.edu/~hube/publications/otraub-thesis.pdf>
- [92] Poletto and Sarkar, *Linear Scan Register Allocation*, ACM, 1997.  
<http://www.research.ibm.com/jalapeno/papers/toplas99.pdf>
- [93] Fischer and LeBlanc, op cit., p. 569.
- [94] Tremblay and Sorenson, *The Theory and Practice of Compiler Writing*, McGraw-Hill, 1985, p. 612.
- [95] GNU, *lightning*, <http://www.gnu.org/software/lightning/>
- [96] Fernandez and Ramsey, *The New Jersey Machine-Code Toolkit*,  
<http://www.eecs.harvard.edu/~nr/toolkit/>
- [97] Delorie and GNU, *objdump*, DJGPP, <http://www.djgpp.com>
- [98] Ponder, op cit.

[99] Sharp, *Advanced Emulation Techniques:Progress Report*, 2000.

[100] Furber, op cit., p. 38.

All URLs that have been referenced are confirmed correct as of 1<sup>st</sup> May 2001.

# Bibliography

Included here are all the texts referenced in this report and several others that have been used in the development of the project.

## Books

Acorn Computers Ltd., *RISC OS Programmers Reference Manual*, 1989.

Aho, Sethi and Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley, 1986.

Appel, *Modern Compiler Implementation in C*, Cambridge University Press, 1997.

Brey, *The Intel Microprocessors, 5<sup>th</sup> edition*, Prentice Hall, 2000.

Brown, *Writing interactive compilers and interpreters*, Wiley, 1979.

Cormen, Leiserson and Rivest, *Introduction to Algorithms*, The MIT Press, 1990.

Dandamudi, *Introduction to Assembly Language Programming: From 8086 to Pentium Processors*, Springer-Verlag, 1998.

Fischer and LeBlanc, *Crafting a Compiler*, The Benjamin/Cummings Publishing Company, Inc., 1988.

Furber, *ARM system-on-chip architecture, 2<sup>nd</sup> edition*, Addison-Wesley, 2000.

Ginns, *Archimedes Assembly Language, 2<sup>nd</sup> edition*, Dabs press, 1988.

Grune, Bal, Jacobs and Langendoen, *Modern Compiler Design*, John Wiley & Sons Ltd., 2000.

Hennessy and Patterson, *Computer Organization & Design, 2<sup>nd</sup> edition*, Morgan Kaufmann, 1997.

Horton, *Introduction to Microsoft Visual C++ 6.0 Standard Edition*, Wrox, 1998.

Irvine, *Assembly Language for Intel-based Computers, 3<sup>rd</sup> edition*, Prentice Hall, 1999.

Jaggar, *ARM Architecture Reference Manual*, Prentice Hall, 1996.

Muchnik, *Advanced Compiler Design & Implementation*, Morgan Kaufmann, 1997.

Nori, Ammann, Jenson, Nageli and Jacobi, *Pascal – The Language and Its Implementation*, New York: John Wiley & Sons, 1981.

Santayana, *The Life of Reason*, 1905.

Schildt, *Teach yourself C++*, 3<sup>rd</sup> edition, Osborne, 1998.

Stroustrup, *The C++ Programming Language*, 2<sup>nd</sup> edition, Addison-Wesley, 1991.

Sweetman, *See MIPS Run*, Morgan Kaufmann Publishers, Inc., 1999.

Tasker, et. al, *Professional Symbian Programming: Mobile Solutions on the EPOC Platform*, Wrox Press Inc., 2000.

Tremblay and Sorenson, *The Theory and Practice of Compiler Writing*, McGraw-Hill, 1985, p. 612.

Van Someren and Van Someren, *Archimedes Operating System*, Dabs press, 1991.

Venners, *Inside the Java Virtual Machine*, McGraw-Hill, 1998.

Wilkinson, *Computer Architecture: design and performance*, 2<sup>nd</sup> edition, Prentice Hall, 1996.

## Papers and Reports

[SiMkIn], *A 65816 Primer*, <http://www.zophar.net/tech/files/65816info.txt>

Advanced RISC Machines, *ARM3 Datasheet*, issue 1.2, 1991.

Apple Computer, Inc., *Technical Note PT39: The DR Emulator*,  
[http://developer.apple.com/technotes/pt/pt\\_39.html](http://developer.apple.com/technotes/pt/pt_39.html)

Ardi Ltd., *Executor Internals: How to Efficiently Run Mac Programs on PCs*,  
[http://www.ardi.com/ardi/executor\\_paper.html](http://www.ardi.com/ardi/executor_paper.html)

ARM Ltd., *ARM610 Data Sheet*, 1993.

ARM Ltd., *Jazelle<sup>TM</sup> – ARM Architecture Extensions for Java Applications*,  
<http://www.arm.com/armtech/jazelle?OpenDocument>

Arnold, Fink, Grove, Hind and Sweeney, *Adaptive Optimization in the Jalapeno JVM*, ACM SIGPLAN, Conference on Object Oriented Programming, Systems, Languages and Applications, October 15-19, 2000.

Auslander and Hopkins, *An Overview of the PL.8 Compiler*, ACM, 1982.

Bell, “Threaded Code”, *Communications of the ACM*, 16(6):370–372, 1973.

Brown, *A neat hack for speeding up arbitrary flow control in dynamically-recompiled code*, <http://dynarec.com/~jules/livehash.html>

Brown, *ARMphetamine: A Dynamically Recompiling ARM Emulator*,  
<http://www.dynarec.com/~jules/>

Cifuentes and Malhotra, “Binary Translation: Static, Dynamic, Retargetable?”, *Proceedings International Conference on Software Maintenance*. Monterey, CA, Nov 4-8 1996. IEEE-CS Press. pp 340-349.

Compaq, *How DIGITAL FX!32 works*, <http://www.support.compaq.com/amt/fx32/fx-white.html>

Connectix, *Virtual PC 4*,  
[http://www.connectix.com/downloadcenter/pdf/vpc4\\_faq\\_jan01.pdf](http://www.connectix.com/downloadcenter/pdf/vpc4_faq_jan01.pdf)

Gordon, Pratt, Birtwistle, Hobley, *Formal Specification and Verification of ARM6*, 1999, <http://www.comp.leeds.ac.uk/graham/research/research.html>

Gosling and McGilton, *The Java Language Environment*, Sun Microsystems, 1995.

Klaiber, Transmeta Corporation, *The Technology Behind Crusoe<sup>TM</sup> processors*,  
<http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf>

König, *ARM - Architectural Family Bonds*, <http://www.linguistik.uni-erlangen.de/~mlkoenig/ArchARM.html>

König, *Dynamic Recompilation - Frequently Asked Questions*,  
<http://dynarec.com/~mike/drfaq.html>

Krall, “Efficient JavaVM Just-in-Time Compilation”, *Proceedings of PACT’98*, IEEE, 1998, <http://www.complang.tuwien.ac.at/java/cacao/index.html>

Meloan, *The Java HotSpot<sup>TM</sup> Performance Engine: An In-Depth Look*,  
<http://developer.java.sun.com/developer/technicalArticles/Networking/HotSpot/>

Mycroft and Norman, *Optimising compilation – Part I: classical imperative languages*, SOFSEM ’92 Czechoslovakia, 22.11.-4.12.1992.

Pajak, *ARM6 Model*, <http://www.comp.leeds.ac.uk/pajak/demo6.html>

Poletto and Sarkar, *Linear Scan Register Allocation*, ACM, 1997.  
<http://www.research.ibm.com/jalapeno/papers/toplas99.pdf>

Ponder, *Generator: A Sega Genesis emulator*, 1998. <http://www.squish.net/generator/>

Sharp, *Advanced Emulation Techniques: Progress Report*, 2000.

Sun Microsystems, Inc., *Writing Advanced Applications*, 2001,  
<http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/perf2.html>

Sun Microsystems, *Method Inlining Example*,  
<http://developer.java.sun.com/developer/technicalArticles/Networking/HotSpot/inlining.html>

Traub, *Quality and Speed in Linear-Scan Register Allocation*, Harvard College, 1998.  
<http://www.eecs.harvard.edu/~hube/publications/otraub-thesis.pdf>

## Software

Barnes, *Red Squirrel*, <http://www.red-squirrel.org>

Bell, *Archimedes Elite*,  
<http://www.iancgbell.clara.net/clara.net/i/a/n/iancgbell/webpace/elite/arc/index.htm>

Bloch, *Riscose*, <http://riscose.sourceforge.net/>

Campbell-Kelly, *The EDSAC Simulator*, <http://www.edsac.net>

Corlett, *Starscream 680x0 emulation library*, <http://www4.ncsu.edu/~nscorlet/star/>

Dales, *SWARM – Software ARM*, <http://www.dcs.gla.ac.uk/~michael/phd/swarm.html>

Delorie and GNU, *objdump*, DJGPP, <http://www.djgpp.com>

Dorr, *ARM2*, <http://www.nvg.unit.no/bbc/emul/Arm2.zip>

EmuHQ, *Corn: n64 emulator*, <http://www.emuhq.com/corn/>

EmuUnlimited, *UltraHLE*, <http://www.emuunlim.com/UltraHLE/>

Fernandez and Ramsey, *The New Jersey Machine-Code Toolkit*,  
<http://www.eecs.harvard.edu/~nr/toolkit/>

Gilbert, *ArcEm*, <ftp://ftp.arm.uk.linux.org/pub/linux/arcem/>

GNU, *lightning*, <http://www.gnu.org/software/lightning/>

Julliard et.al, *Wine Development HQ*, <http://www.winehq.com>

Larus, *SPIM: A MIPS R2000/R3000 Simulator*,  
<http://www.cs.wisc.edu/~larus/spim.html>

Lloyd, *Archie*, <http://www.geocities.com/SiliconValley/Campus/5427/index.html>

PSXEmu, *PSEmu Pro*, <http://www.psxemu.com/psemu.shtml>

Rutter, *Sleeve*, <http://www.armlinux.org/projects/sleeve/>

Sharp, *Goby: The Gameboy emulator for EPOC*, <http://www.backlitgames.com>

Sharp, *Manchester Baby Simulator*, <http://www.backlitgames.com/baby.zip>

Vik, *StrongED*, <http://home.eunet.no/~gutturvi/strong.html>

## Websites

Cifuentes, Emmerik, Ramsey, *UQBT – A Resourceable and Retargetable Binary Translator*, <http://www.csee.uq.edu.au/~csmweb/uqbt.html>

Hyde, *Art of Assembly Language Programming*, <http://webster.cs.ucr.edu/>

## Acknowledgements

I am very grateful to the the following people:

Graeme Barnes for allowing me the use of Red Squirrel and for being so supportive.

Julian Brown for allowing me the use of the run-time assembler and for answering so many of my questions.

Everyone on the dynarec.com mailing list and particularly Michael König and Neil Bradley.

Dr Graham Martin for allowing me to persuade him to supervise the project.

Dr Sara Kalvala for all the good advice.

The helpful people on the `comp.sys.arm` and `comp.lang.asm.x86` newsgroups.

The people at ARM, particularly Dr Mark Burton and John Callan.

Dr Graham Birtwistle and Dominic Pajak for the information.

CJE Micros for running their RISC OS Programmers Initiative to supply a cut price Risc PC which greatly helped with the development of this project.

Chris Tucker for proof reading this report.



## Appendix A – Overview of ARM assembly

This appendix contains brief examples of typical ARM instructions, their syntax and their operation, sufficient to understand the ARM assembly examples in this report.

### Data Processing

Data processing instructions are the logic and arithmetic operations (with the exception of multiply). Most of the instructions have a destination register specified and set its value to the appropriate result, as described:

AND	operand1 AND operand2
EOR	operand1 EXCLUSIVE-OR operand2
ORR	operand1 OR operand2
MOV	operand2 (simply moves the value)
BIC	operand1 AND (NOT operand2)
MVN	NOT operand2 (inverts and moves the value)
SUB	operand1 – operand2
RSB	operand2 – operand1
ADD	operand1 + operand2
ADC	operand1 + operand2 + carry flag
SBC	operand1 – operand2 – (1 – carry flag)
RSC	operand2 – operand1 – (1 – carry flag)

Other instructions perform a logical or arithmetic operation but don't place the result in a register and just update the condition flags. These instructions and their operation are:

TST	operand1 AND operand2
TEQ	operand1 EOR operand2
CMP	operand1 – operand2
CMN	operand1 + operand2

All these data processing instructions have one operand which must be a register and another which can either be a register, a register shifted in some way or an immediate value. The five methods of shifting available are:

LSL	logical shift left by n positions
LSR	logical shift right by n positions
ASR	arithmetic shift right by n positions
ROR	rotate right by n positions
RRX	rotate right with extend (through carry flag) by 1 position

Examples of the syntax of these data processing instructions are:

```
add r0,r1,r2,ls1 #1      ; r0 = r1 + (r2 << 1)
cmp r0,#10               ; compare r0 to 10
mov r0,r1,lsr r2         ; r0 = r1 >> r2
sbc r0,r1,#1             ; r0 = r1 - 1 - (1 - carry flag)
```

## Multiply

There are two types of multiply instruction, normal multiply and multiply-with-accumulate. They only take straightforward registers as operands. For example

```
mul r0,r1,r2          ; r0 = r1 * r2
mla r0,r1,r2,r3        ; r0 = (r1 * r2) + r3
```

## Single Data Swap

This instruction is used to perform an atomic swap between registers and memory, specifically for implementing semaphores. The instruction takes three register operands and has the syntax

```
swp r0,r1,[r2]
```

This instruction takes the location specified by an address in r2, loads the value from that location in memory and places it in r0. The value in r1 is then stored in memory at that location. The instruction can also operate on byte quantities by suffixing a 'B' to the instruction mnemonic.

## Single Data Transfer

These instructions allow a single value to be loaded from memory into a register or stored from a register to memory, operating either on byte or word (32 bit) quantities. These instructions come in a myriad of different addressing modes but all have a register that contains a value to be stored, or that the value to be loaded should be placed in. They also all have a register that contains the base address in memory for the operation to take place. Finally they all have an offset from that base address which when combined with the base gives the actual location to transfer data to/from. This offset can be a register, a shifted register or an immediate.

Additionally the offset can be added to the base address before or after the data transfer takes place (i.e. pre or post addressing) depending on whether the offset is inside the square braces or not. The address resulting from combining the base and the offset can optionally be written-back into the base register or not (denoted by a '!'). Typical instructions are as follows:

```
str  r0,[r1,#4]        ; memory[r1 + 4] ← r0
ldr  r0,[r1],#1        ; r0 ← memory[r1]; r1 +=1
ldrb r0,[r1]           ; r0 ← memory[r1] & 0xff
strb r0,[r1,r2,ls1 #2] ; memory[r1 + (r2 << 2)] ← (r0 & 0xff)
ldr  r0,[r1,r2]!       ; r0 ← memory[r1 + r2]; r1 += r2
```

## Block Data Transfer

These instructions allow any combination of the registers to be either stored to memory or loaded from memory. Each instruction has a base register, containing the address at which the transfer is to start, and a list of registers to be transferred. The instruction is also configured to increment or decrement the address being stored at, either after or before each register is transferred. Again, the final address when all

registers have been transferred can be written back to the base register (denoted by a '!'). Typical instructions are:

```
ldmia      r0,{r1-r14}      ; load r1 - r14 from memory
                                incrementing after each
stmdb      r0,{r1,r3,r5}    ; store r1,r3,r5 to memory
                                decrementing before each
ldmib      r1!,{r1,r12}     ; load r1 and r12 from memory
                                incrementing before each and
                                writing the final address back
                                into r1.
```

## Branch

There are two forms of branch instruction which allow a branch +/- 32 Mbytes from the current instruction. They are a normal branch instruction and a branch-with-link, the latter placing the address of the instruction after the branch into r14 before branching. For example

```
b  0x12345678      ; normal branch
bl 0x87654321      ; branch with link
```

Note that the PC register, r15, can be used as the destination of any data processing instruction which is another way to adjust the PC.

## SWI

A software interrupt instruction triggers a software interrupt exception which allows the operating system to access a specific routine. This routine is specified in the instruction's encoding as a 24 bit comment field. SWI instructions are typically referred to by name, although encoded as a number, some example are as follows:

```
swi OS_WriteC      ; write the char in r0 to the display
swi OS_BinaryToDecimal ; convert an integer to a string
swi 0x400c6        ; equivalent to Wimp_CloseWindow
```

## The S flag

Most ARM instructions that might adjust the flags have a so called 'S' flag. If the S flag is set in an instruction's encoding then the result of that instruction is used to update the processor condition flags. This is denoted by an 's' being suffixed to the instruction mnemonic, for example:

```
adds r0,r1,r2
muls r0,r1,r2
```

## Conditional execution

The one thing all ARM instructions have in common is that they are conditionally executed. This means that prior to each instruction, its condition field is compared to the current processor condition flags (Negative, Zero, Carry and Overflow) and only if the condition is satisfied is the instruction executed. The conditions available are as

follows with their meaning and condition flags for the condition check to be successful:

- EQ – Equal (Z set)
- NE – Not Equal (Z clear)
- CS – Carry Set (C set)
- CC – Carry Clear (C clear)
- MI – Negative (N set)
- PL – Positive or Zero (N clear)
- VS – Overflow Set (V set)
- VC – Overflow Clear (V clear)
- HI – Unsigned Higher (C set and Z clear)
- LS – Unsigned Lower or Same (C clear or Z set)
- GE – Greater or Equal (N set and V set, or N clear and V clear)
- LT – Less Than (N set and V clear, or N clear and V set)
- GT – Greater Than (Z clear, and either N set and V set, or N clear and V clear)
- LE – Less than or Equal (Z set, or N set and V clear, or N clear and V set)
- AL – Always (whatever the flags)
- NV – Never (never executed)

### **Further information**

There has only been space here to briefly describe most of the ARM instructions. For more information you should refer to any of the ARM data sheets.

## Appendix B – Armlet definition

This is the definition of all armlet instructions used by Tarmac. The inflags and outflags are the defaults for a given armlet related to its ARM instruction origins. Each armlet can be generated with a different set of inflags/outflags adjusted according to the circumstances in which the armlet is to be executed. The use of each armlet is defined in terms of the instruction mnemonic followed by a series of letters (denoting armlet variables) and # symbols (denoting immediate constants).

armlet	use	inflags	outflags	operation
adc	adc a,b,c	xxCx	NZCV	a = b + c + carryflag
add	add a,b,c	xxxx	NZCV	a = b + c
and	and a,b,c	xxxx	NZxx	a = b & c
asr	asr a,b,c	xxxx	xxCx	a = b arithmetic-shift-right c
cleartrans	cleartrans	xxxx	xxxx	Tell the MMU to clear the Trans flag
cmn	cmn a,b	xxxx	NZCV	Set flags according to the result of a - (-b)
cmp	cmp a,b	xxxx	NZCV	Set flags according to the result of a - b
eor	eor a,b,c	xxxx	NZxx	a = b ^ c
getpc	getpc v,#	NZCV	xxxx	Place the PC and PSR into a variable (the # is the PC)
goto	goto #	xxxx	xxxx	goto armlet number # unconditionally
gotocc	gotocc #	xxCx	xxxx	goto armlet number # if flags denote carry clear
gotocs	gotocs #	xxCx	xxxx	goto armlet number # if flags denote carry set
gotoeq	gotoeq #	xZxx	xxxx	goto armlet number # if flags denote equal
gotoge	gotoge #	NxxV	xxxx	goto armlet number # if flags denote >=
gotogt	gotogt #	NZxV	xxxx	goto armlet number # if flags denote >
gotohi	gotohi #	xZCx	xxxx	goto armlet number # if flags denote unsigned >
gotole	gotole #	NZxV	xxxx	goto armlet number # if flags denote <=
gotols	gotols #	xZCx	xxxx	goto armlet number # if flags denote unsigned <=
gotolt	gotolt #	NxxV	xxxx	goto armlet number # if flags denote <
gotomi	gotomi #	Nxxx	xxxx	goto armlet number # if flags denote negative
gotone	gotone #	xZxx	xxxx	goto armlet number # if flags denote not equal
gotopl	gotopl #	Nxxx	xxxx	goto armlet number # if flags denote positive
gotovc	gotovc #	xxxV	xxxx	goto armlet number # if flags denote overflow clear
gotovs	gotovs #	xxxV	xxxx	goto armlet number # if flags denote overflow set
intcheck	intcheck v,#	xxxx	xxxx	Check for interrupts after # ARM instructions
ldb	ldb a,b,c	xxxx	xxxx	load byte at address b into a, success flag in c
ldw	ldw a,b,c	xxxx	xxxx	load word at address b into a, success flag in c
leave	leave #	NZCV	xxxx	leave the chunk, # is the reason code
lsl	lsl a,b,c	xxxx	xxCx	a = b << c
lsr	lsr a,b,c	xxxx	xxCx	a = b >> c
mov	mov a,b	xxxx	NZxx	Set a to the value of b
movc	movc v,#	xxxx	xxxx	Move a constant into a variable
mul	mul a,b,c	xxxx	NZxx	a = b * c
mvn	mvn a,b	xxxx	NZxx	Set a to the value of negative b
orr	orr a,b,c	xxxx	NZxx	a = b   c
ror	ror a,b,c	xxxx	xxCx	a = b rotate-right c

rrx	rrx v,v	xxCx	xxCx	Perform rotate-right 1 location through carry flag
sbc	sbc a,b,c	xxCx	NZCV	$a = b - c - (1 - \text{carryflag})$
setpc	setpc	NZCV	xxxx	Place the program counter value in the pc variable
settrans	settrans	xxxx	xxxx	Tell the MMU to set the Trans flag
stb	stb a,b,c	xxxx	xxxx	store byte in a at address b, success flag in c
stw	stw a,b,c	xxxx	xxxx	store word in a at address b, success flag in c
sub	sub a,b,c	xxxx	NZCV	$a = b - c$
teq	teq a,b	xxxx	NZxx	Set flags according to the result of $a \wedge b$
tst	tst a,b	xxxx	NZxx	Set flags according to the result of $a \& b$

## Appendix C – Example code generation

The greatest common divisor algorithm is represented in C as follows

```
while( x != y)
{
    if(x > y)
        x = x - y;
    else
        y = y - x;
}
```

This can be written in ARM assembly as the following sequence of ARM instructions, with the initial values of 24 and 18.

```
0x0:  mov        r0,#0x18
0x4:  mov        r1,#0x12
0x8:  cmp        r0,r1
0xc:  subgt      r0,r0,r1
0x10: sublt      r1,r1,r0
0x14: bne        0x0
0x18: mov        r15,r14
```

If the profiler is directed to recompile the entire sequence, the following armlets are generated. The arrows denote the leaders of the basic blocks, as identified by the optimiser.

```
0x0:  [xxxx->xxxx] movc      t0,0x18  <--
0x1:  [xxxx->xxxx] mov       r0,t0
0x2:  [xxxx->xxxx] movc      t0,0x12
0x3:  [xxxx->xxxx] mov       r1,t0
0x4:  [xxxx->NZCV] cmp       r0,r1    <--
0x5:  [NZxV->xxxx] gotole    0x7
0x6:  [xxxx->xxxx] sub       r0,r0,r1 <--
0x7:  [NxxV->xxxx] gotoge    0x9      <--
0x8:  [xxxx->xxxx] sub       r1,r1,r0 <--
0x9:  [xxxx->xxxx] intcheck   t0,0x6   <--
0xa:  [xxxx->xxxx] movc      t1,0x0
0xb:  [xxxx->xxxx] cmp       t0,t1
0xc:  [xxxx->xxxx] gotoeq    0xf
0xd:  [xxxx->xxxx] movc      pc,0x10  <--
0xe:  [xxxx->xxxx] leave     0xb
0xf:  [xZxx->xxxx] gotone    0x4      <--
0x10: [xxxx->xxxx] mov       pc,r14   <--
0x11: [xxxx->xxxx] intcheck   t0,0x1
0x12: [xxxx->xxxx] movc      t1,0x0
0x13: [xxxx->xxxx] cmp       t0,t1
0x14: [xxxx->xxxx] gotoeq    0x16
0x15: [xxxx->xxxx] leave     0xb      <--
0x16: [NZCV->xxxx] leave     0x2      <--
```

The interrupt checking armlets are in grey, to show just how many of the generated armlets are devoted to this task. This sequence of armlets is then converted into the following x86 instruction sequence.

```

0:  c7 c0 18 00 00      movl    $0x18,%eax          ; movc t0,0x18 ; mov r0,t0
5:  00
6:  c7 c1 12 00 00      movl    $0x12,%ecx          ; movc t0,0x12 ; mov r1,t0
b:  00
c:  39 c8              cmpl    %ecx,%eax          ; cmp r0,r1
e:  f5                cmc
f:  0f 98 45 3d        sets   0x3d(%ebp)
13: 0f 94 45 3e        sete  0x3e(%ebp)
17: 0f 92 45 3f        setb  0x3f(%ebp)
1b: 0f 90 45 40        seto  0x40(%ebp)
1f: 0f 8e 12 00 00      jle    37                  ; gotole 0x7
24: 00
25: 0f 98 45 3d        sets   0x3d(%ebp)
29: 0f 94 45 3e        sete  0x3e(%ebp)
2d: 0f 92 45 3f        setb  0x3f(%ebp)
31: 0f 90 45 40        seto  0x40(%ebp)
35: 29 c8              subl   %ecx,%eax          ; sub r0,r0,r1
37: 0f 98 45 3d        sets   0x3d(%ebp)
3b: 0f 94 45 3e        sete  0x3e(%ebp)
3f: 0f 92 45 3f        setb  0x3f(%ebp)
43: 0f 90 45 40        seto  0x40(%ebp)
47: 0f 8d 12 00 00      jge    5f                  ; gotoge 0x9
4c: 00
4d: 0f 98 45 3d        sets   0x3d(%ebp)
51: 0f 94 45 3e        sete  0x3e(%ebp)
55: 0f 92 45 3f        setb  0x3f(%ebp)
59: 0f 90 45 40        seto  0x40(%ebp)
5d: 29 c1              subl   %eax,%ecx          ; sub r1,r1,r0
5f: 0f 98 45 3d        sets   0x3d(%ebp)
63: 0f 94 45 3e        sete  0x3e(%ebp)
67: 0f 92 45 3f        setb  0x3f(%ebp)
6b: 0f 90 45 40        seto  0x40(%ebp)
6f: c7 85 c6 00 00      movl    $0x6,0xc6(%ebp)    ; intcheck t0,0x6
74: 00 06 00 00 00
79: c7 c2 30 2b 40      movl    $0x402b30,%edx
7e: 00
7f: ff d2              call    *%edx
81: 8b 95 c2 00 00      movl    0xc2(%ebp),%edx
86: 00
87: 0f 98 45 3d        sets   0x3d(%ebp)
8b: 0f 94 45 3e        sete  0x3e(%ebp)
8f: 0f 92 45 3f        setb  0x3f(%ebp)
93: 0f 90 45 40        seto  0x40(%ebp)
97: 0f 98 45 3d        sets   0x3d(%ebp)
9b: 0f 94 45 3e        sete  0x3e(%ebp)
9f: 0f 92 45 3f        setb  0x3f(%ebp)
a3: 0f 90 45 40        seto  0x40(%ebp)
a7: 81 fa 00 00 00      cmpl    $0x0,%edx          ; movc t1,0x0 ; cmp t0,t1
ac: 00
ad: 0f 98 45 3d        sets   0x3d(%ebp)
b1: 0f 94 45 3e        sete  0x3e(%ebp)
b5: 0f 92 45 3f        setb  0x3f(%ebp)
b9: 0f 90 45 40        seto  0x40(%ebp)
bd: 0f 84 0d 00 00      je     d0                  ; gotoeq 0xf
c2: 00
c3: c7 c3 10 00 00      movl    $0x10,%ebx          ; movc pc,0x10
c8: 00
c9: c7 c0 0b 00 00      movl    $0xb,%eax          ; leave 0xb
ce: 00
cf: c3                ret
d0: 0f 98 45 3d        sets   0x3d(%ebp)
d4: 0f 94 45 3e        sete  0x3e(%ebp)
d8: 0f 92 45 3f        setb  0x3f(%ebp)
dc: 0f 90 45 40        seto  0x40(%ebp)
e0: 0f 85 26 ff ff      jne    c                   ; gotone 0x4
e5: ff
e6: 8b 5d 38           movl    0x38(%ebp),%ebx      ; mov pc,r14
e9: 0f 98 45 3d        sets   0x3d(%ebp)
ed: 0f 94 45 3e        sete  0x3e(%ebp)
f1: 0f 92 45 3f        setb  0x3f(%ebp)

```



```

f5: 0f 90 45 40      seto    0x40(%ebp)
f9: c7 85 c6 00 00  movl    $0x1,0xc6(%ebp)      ; intcheck t0,0x1
fe: 00 01 00 00 00
103: c7 c2 30 2b 40   movl    $0x402b30,%edx
108: 00
109: ff d2           call    *%edx
10b: 8b 95 c2 00 00   movl    0xc2(%ebp),%edx
110: 00
111: 0f 98 45 3d      sets    0x3d(%ebp)
115: 0f 94 45 3e      sete    0x3e(%ebp)
119: 0f 92 45 3f      setb    0x3f(%ebp)
11d: 0f 90 45 40      seto    0x40(%ebp)
121: 0f 98 45 3d      sets    0x3d(%ebp)
125: 0f 94 45 3e      sete    0x3e(%ebp)
129: 0f 92 45 3f      setb    0x3f(%ebp)
12d: 0f 90 45 40      seto    0x40(%ebp)
131: 81 fa 00 00 00   cmpl    $0x0,%edx      ; movc t1,0x0 ; cmp t0,t1
136: 00
137: 0f 98 45 3d      sets    0x3d(%ebp)
13b: 0f 94 45 3e      sete    0x3e(%ebp)
13f: 0f 92 45 3f      setb    0x3f(%ebp)
143: 0f 90 45 40      seto    0x40(%ebp)
147: 0f 84 07 00 00   je      154             ; gotoeq 0x16
14c: 00
14d: c7 c0 0b 00 00   movl    $0xb,%eax      ; leave 0xb
152: 00
153: c3             ret
154: c7 c0 02 00 00   movl    $0x2,%eax      ; leave 0x2
159: 00
15a: c3             ret

```

The major feature of the code generation is the way that 7 ARM instructions expand to 23 armlets which in turn expand to nearly 100 x86 instructions.

The insertion of the CMC instruction after the CMP (subtraction-based) instruction at address 0xe is to maintain the ARM-style carry flag, as described in section 9.5. The jne instruction at 0xe0 is notable for having been directly translated from the original bne instruction to a single gotone armlet and finally to a single x86 instruction. The constant folding has worked effectively, demonstrated by the way that the movc and cmp armlets have coalesced back into a single cmpl instruction at address 0xa7.

The instructions from 0xf9 to 0x10b demonstrate the way in which the intcheck armlet has code generated to call a C function, passing arguments to and from the C program without leaving the generated code. The last two instructions place the number 2 in the EAX register (signifying a leave because of dynamic PC) and then execute a ret instruction to return from the native code in the same way as a C function would.

With around 48% of the armlets being leaders of basic blocks, the over-zealousness of the flag spilling is highlighted by the repeated spilling of the condition flags to memory, despite them only ever being adjusted by a single armlet. This results in the repeated sequence of setcc x86 instructions in generated code.

## Appendix D – Source code

The source code is divided into several C++ files and header files for each class. The following table specifies the files with the data structures and classes that they define. The naming convention used by Visual C++ of prefixing class names with a 'C' has been followed, other types are implemented as C structures.

File name	Type	System component
Arm.cpp	CArm	ARM Interpreter
ArmDisassembler.cpp	CArmDisassembler	ARM Disassembler
ArmletDisassembler.cpp	CArmletDisassembler	Armlet Disassembler
CodeCache.cpp	CCodeCache	Code Cache
Dispatcher.cpp	CDispatcher	Dispatcher
Generator.cpp	CGenerator	x86 Generator
LinkedList.cpp	CLinkedList	
LinkedList.h	LinkedListElement	
TarmacGlobals.h	Context	
	NativeChunk	
Optimiser.cpp	COptimiser	Optimiser
Profiler.cpp	CProfiler	Profiler
TestMemory.cpp	CTestMemory	
Profiler.h	Armlet	
	ConditionalBlockInfo	
Generator.h	GotoBackpatch	
	RegisterAllocation	
	VariableLocation	

To compile the program 'as is', a copy of the source code to Red Squirrel is required to supply the IOC, MEMC and coprocessor emulations. However, all attempts to access these components could be 'commented out' relatively easily in order to compile and test the code generation section of the project.

The program has only been compiled using Microsoft Visual Studio 6 though it should compile with any other C++ compiler. If used with other compilers the source code may require simple modifications where the Visual Studio MFC CString class has been used for the various disassemblers.

A daily diary was kept throughout the project development detailing the progress made and the problems faced. This has been included with the source code for completeness.

## Appendix E – Glossary of Terms

This is a brief glossary of some of the terms used in the report. It is included in an attempt to allow more readers to understand the report without patronizing experienced readers.

**2-address code** – An instruction set that has an operation and only 2 operands (values used in that operation). Normally this results in the operation overwriting one of the operands with its result. Commonly found in the x86 architecture and older CISC processors.

**3-address code** – An instruction set that has an operation and 3 operands. This allows an operation to take place without overwriting one of the operands. Commonly found in RISC architectures.

**Accessor method** – A method which gives access to the state of an object, commonly used to ‘get’ or ‘set’ the value of a private variable.

**Address space** – The space in memory that can be addressed by the processor. Typically for 16 bit processors this is  $2^{16}$  bytes or 64 Kbytes, or for a 32 bit processor this is  $2^{32}$  or 4 Gbytes.

**API** – Application Programming Interface. This is a library of routines that allow an underlying system to be used by other software without the other software needing to know the details about how it works.

**Backpatching** – A technique used in compilers when generating instructions. Any information that isn’t known when an instruction is generated is left blank and backpatched or ‘filled in’ later.

**Basic block** – A sequence of instructions that executes from start to end without any possibility of executing another instruction in between.

**Breakpoint** – Points in a program that are set by the programmer so that when that section is reached in execution, the program pauses and a debugging interface appears. This allows the programmer to interrogate the state of the program.

**Bytecode** – The term used for the code that runs on a Java Virtual Machine.

**Condition flags** – When a calculation is performed by an instruction, it sets certain condition flags, typically Negative, Zero, Carry and Overflow, describing the result of the instruction. Later instructions can then use the state of these flags to find out the condition of that calculation.

**Conditional execution** – Where an instruction is only executed if the condition associated with it is successful. Often this means that the condition code matches the state of the condition flags.

**Control flow** – Control flow is a term used to describe the flow of execution from one instruction to another and how it might be modified by an instruction which branches to another part of the program.

**Coprocessor** – A specialised chip to execute certain calculations faster than the main processor could do them. These are commonly used for floating point arithmetic, digital signal processing and graphics processing.

**Delayed branches** – When a branch instruction occurs, the instruction immediately after it is executed before the branch is taken. This allows better pipelining as conditional branches no longer cause the pipeline to empty.

**Dynamic profiling** – Rather than statically examining a program as a list of instructions, if information can be stored about exactly which instructions the processor executes when running the program, far better results about the program's execution can be gained.

**Host system** – See Native System.

**Immediate value** – operands can either be encoded in instructions as registers or as numbers. An immediate is simply a constant number encoded into an instruction.

**JIT** - Just-In-Time is the term used to describe a Java Virtual Machine that recompiles every method to native code before it is executed.

**Leader** – The leader of a basic block is an instruction that marks the start of a basic block. This tends to be instructions that are branched to and instructions that occur directly after branches.

**Memory-mapped IO** – Components of the computer need some way of being accessed by programs. This is done by mapping sections of the address space to them so that programs can access the device as if accessing memory.

**Method** – Is very similar to a function in imperative programming languages. The difference is that a method 'belongs' to a certain object in object oriented programming.

**Naïve algorithm** – An algorithm that takes the obvious and inefficient approach to solving a problem.

**Native system** – The system that is running the emulator and which has machine code for it generated by the recompiler.

**Operand** – A value that is passed to an instruction for it to perform its operation on.

**Orthogonal** – a feature of a microprocessor that can be used in the same way by all other features. For example, an add instruction that can accept any one of the registers, including the PC as an operand.

**PC** – The Program Counter. A special register that holds an address in memory pointing to the instruction to be executed.

**Pipelined** – A processor that performs several operations at one to increase speed, for example fetching one instruction while decoding the previous instruction while executing the one before that.

**Portable** – A program that can be transported to another architecture and made to run there with only minor changes.

**Register bank** – A register bank is a set of registers. It is common for modern processors to have more than one bank and to be able to quickly switch between which one is to be used.

**Register Transfer Language** - A simple language that describes how values change between registers. Commonly used to describe the operation of an instruction.

**RISC** – Reduced Instruction Set Chip. A processor that has relatively simple instructions (often discounting even division as being too complex) that can be executed quickly. The actual meaning of the term has become blurred and is generally used to refer to any processor of the late 1980s or later.

**Source system** – The system being emulated and the system that has its instructions recompiled into native code.

**Static branch prediction** – Normally a conditional branch is known to be executed or not most of the time. By the instruction reflecting whether the branch is likely to be taken or not, the pipelining can continue down the path that is most probable, improving performance.

**Status flags** – These are flags that hold details about the current state of the processor. These generally include details about whether interrupts are enabled and the current processor mode.

**Target system** – See Native System.

**Translation unit** – A section of source code that is recompiled to a section of native code.

**Vector** – A location in memory that is branched to in specific circumstances such as when an interrupt occurs. The code at the vector address then deals appropriately with the event.

**VHDL** – Very High Speed Integrated Circuit Hardware Description Language. A language used to explicitly define the operation of integrated circuits.

**x86** – The name given to generically refer to any processor compatible with Intel's range of 8086, 80186, 80286, 80386, 80486, Pentium and Pentium 2-4 processors. In this report, it is the 32 bit versions, the 80386 and above that are used.

## Appendix F – Advice on attempting similar projects

CPU emulation is far from the only important part to emulating a computer. Unless the graphics emulation is well written too, it will cripple the emulator's performance. That said, using techniques such as dynamic recompilation will minimise the processing time required by the CPU emulation, leaving even fairly naïve graphics emulations enough time to appear to perform well.

I believe it is more important to know the target processor than the source processor when starting a dynamic recompilation project. The reason is that the level of knowledge required to emulate a processor surpasses any general knowledge someone may have about that processor. The only way to get the knowledge required is by referring to very low level documentation such as processor data sheets. In contrast, the level of knowledge required of the target processor is little more than that of any good assembly programmer.

It is very important to build a competent interpreting emulator before attempting a dynamic recompiler. Without it there is little to compare the recompiler's operation to and no guarantee that bad assumptions about the source processor have not been made. If an existing third party's interpreting emulator is suitably well written, it may be possible to use this to refer to and test the recompiler against. This is recommended in order to allow the maximum possible time for developing the recompiler.

Debugging any emulator requires good detective skills, debugging a dynamic recompiler is a job for Sherlock Holmes - it is hard to underestimate the problems of debugging a dynamic recompiler. All that can be recommended is that every effort be made to cross-reference and test every level of the program. I cannot emphasize enough how much faster it is to perform exhaustive testing during development than to try to find the bugs afterwards. It is particularly tempting with an emulation project to press ahead to attempt to get early results. This temptation must be ignored.

There are no early results with an emulation project. Hundreds of man-hours are required to make any progress at all and hundreds more to make an emulator perform at an acceptable level of compatibility and speed.

As a result of the difficulty of getting real code to emulate correctly, a dynamic recompiler for a real computer system pushes the boundaries of what is feasible in the time available for a university project. For those that attempt this, I would recommend every opportunity be taken to simplify the project. Whether this means choosing an 8 bit source processor or using static register allocation, there is little need to attempt to add extra complexity to the project. The words of Steve Furber, when talking about designing the original ARM processor, are particularly apt:

“When venturing into unknown territory it is advisable to minimize those risks which are under your control, since this still leaves significant risks from those factors which are not well understood or are fundamentally not controllable.” [100]