University of Dublin

TRINITY COLLEGE

Binary Translation using LLVM

Laurence Reading
B.A.(Mod.) Computer Science
Final Year Project  April 2011
Supervisor: Dr. David Gregg

School of Computer Science and Statistics
O'Reilly Institute, Trinity College, Dublin 2, Ireland
Binary translation using LLVM

by

Laurence Reading

B.A. (Mod.) Computer Science
Final Year Project 2011
Supervisor: David Gregg
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

Laurence Reading
April 11, 2011
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

Laurence Reading
April 11, 2011
Acknowledgments

I would like to thank my supervisor David Gregg for his advice and encouragement throughout the project.

I would also like to thank my friends and family for their support, with special thanks to my parents and to Katya Semler.
## Contents

1 Introduction ............... 1
   1.1 Background ...................... 1
       1.1.1 Binary Translation .......... 1
       1.1.2 ARM introduced ............ 1
       1.1.3 LLVM introduced .......... 2
   1.2 Project Goals ............... 3
   1.3 Motivation .................... 3
   1.4 Readers Guide ................... 3
   1.5 Summary ....................... 4

2 Design and Important Features ....... 5
   2.1 Features of ARM ............... 5
       2.1.1 THUMB ....................... 5
       2.1.2 Registers .................... 5
       2.1.3 The Current Program Status Register ......... 6
       2.1.4 Conditional ................. 6
       2.1.5 Qualifiers ................... 7
       2.1.6 Shifter Operand ............. 7
       2.1.7 Call and Return .............. 8
   2.2 Features of LLVM .............. 8
       2.2.1 Basics ....................... 8
       2.2.2 SSA ......................... 8
       2.2.3 Call and Return ............. 9
       2.2.4 Comparisons .................. 9
   2.3 Design decisions .............. 9
       2.3.1 THUMB ....................... 9
       2.3.2 Error Checking .............. 10
       2.3.3 Status Flags ............... 10
       2.3.4 Call and return .............. 10
       2.3.5 Recognizing instructions ....... 11
   2.4 Summary ....................... 11

3 Implementation ............... 12
   3.1 Overview ....................... 12
   3.2 Finite State Recogniser ...... 12
Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3 Parser</td>
<td>13</td>
</tr>
<tr>
<td>3.3.1 Labels</td>
<td>14</td>
</tr>
<tr>
<td>3.3.2 Instructions</td>
<td>14</td>
</tr>
<tr>
<td>3.3.3 Conditionals</td>
<td>15</td>
</tr>
<tr>
<td>3.3.4 Qualifiers</td>
<td>15</td>
</tr>
<tr>
<td>3.3.5 Operands</td>
<td>15</td>
</tr>
<tr>
<td>3.4 Generator</td>
<td>17</td>
</tr>
<tr>
<td>3.4.1 The Basics</td>
<td>18</td>
</tr>
<tr>
<td>3.4.2 Labels</td>
<td>19</td>
</tr>
<tr>
<td>3.4.3 Conditional</td>
<td>19</td>
</tr>
<tr>
<td>3.4.4 Shifting</td>
<td>19</td>
</tr>
<tr>
<td>3.4.5 Instructions</td>
<td>19</td>
</tr>
<tr>
<td>3.4.6 Branching</td>
<td>19</td>
</tr>
<tr>
<td>3.4.7 Virtualisation</td>
<td>21</td>
</tr>
<tr>
<td>3.5 Summary</td>
<td>24</td>
</tr>
<tr>
<td>4 Analysis</td>
<td>25</td>
</tr>
<tr>
<td>4.1 Obstacles to Translation and Possible Solutions</td>
<td>25</td>
</tr>
<tr>
<td>4.1.1 Program Counter</td>
<td>25</td>
</tr>
<tr>
<td>4.1.2 Calling to External Libraries</td>
<td>26</td>
</tr>
<tr>
<td>4.1.3 Virtualising the Current Program Status Register</td>
<td>27</td>
</tr>
<tr>
<td>4.2 Translated Program Performance using LLVM Optimizations</td>
<td>28</td>
</tr>
<tr>
<td>4.2.1 Bubble Sort</td>
<td>29</td>
</tr>
<tr>
<td>4.2.2 Insertion Sort</td>
<td>30</td>
</tr>
<tr>
<td>4.2.3 Binary Search</td>
<td>31</td>
</tr>
<tr>
<td>4.2.4 Fibonacci</td>
<td>32</td>
</tr>
<tr>
<td>4.2.5 Fanfich-redux</td>
<td>33</td>
</tr>
<tr>
<td>4.2.6 Code Size</td>
<td>34</td>
</tr>
<tr>
<td>4.2.7 Evaluation</td>
<td>34</td>
</tr>
<tr>
<td>4.3 Summary</td>
<td>35</td>
</tr>
<tr>
<td>5 Conclusions</td>
<td>36</td>
</tr>
<tr>
<td>5.1 Learning Outcomes</td>
<td>36</td>
</tr>
<tr>
<td>5.2 Future Work</td>
<td>36</td>
</tr>
<tr>
<td>5.3 Contributions</td>
<td>36</td>
</tr>
<tr>
<td>Bibliography</td>
<td>38</td>
</tr>
<tr>
<td>6 Attached CD</td>
<td>39</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Diagram of the Parser</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Simple diagram of the Generator</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Run times of three versions of the bubble sort program</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Run times of three versions of the insertion sort program</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>Run times of three versions of the binary search program</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>Run times of two versions of the fibonacci program</td>
<td>32</td>
</tr>
<tr>
<td>4.5</td>
<td>Run times of three versions of the fannkuch program</td>
<td>33</td>
</tr>
<tr>
<td>4.6</td>
<td>Code size for various example programs, showing optimized, unoptimized and native code size.</td>
<td>34</td>
</tr>
</tbody>
</table>
1 Introduction

Binary translation is the emulation of one instruction set by another through translation of code. Binary translation is a valuable tool in the development of new hardware [AKS00]. Unfortunately binary translation can introduce inefficiencies resulting in poor performance.

The LLVM project is a suite of tools and technologies used for compiling high level languages. It includes optimizers and code generators for many target architectures. It is also an assembly language which is used as an internal representation during compilation within the project.

This project aims to explore the difficulties and benefits, of using LLVM in the process of static binary translation from ARM assembly to X86 assembly. This is done by using LLVM assembly as an intermediary form between the ARM assembly code and the X86 code.

1.1 Background

The LLVM project, the ARM architecture and the need for efficient binary translation, form the background for this project.

1.1.1 Binary Translation

Static binary translation is the process of converting the code of an executable file into code that runs on the target architecture without running the code first. This is difficult to do perfectly, as some code may only be discovered at run-time. For example the executable may use indirect branches, whose value is only calculated at run-time, to reach parts of the code. The result of a static binary translator should be an executable that can run on the target architecture with no special support.

1.1.2 ARM introduced

ARM is a 32-bit RISC architecture developed by ARM Holdings. Due to its simplicity it is well suited to low-power applications like phones and other portable devices.
As ARM assembly uses a reduced instruction set it is quite a simple design where complex behaviour is created by the combination of the small set of available instructions. This is ideal for my project because the more basic an individual instruction is the easier it is to examine and translate. The book “ARM Assembly Language” \cite{Hoh09} along with the website “Whirlwind Tour of ARM Assembly” \cite{Vij08} were the main sources of technical information on the ARM assembly language used during this project. The ARM information center \cite{Liu} was also used to confirm an error in the book.

1.1.3 LLVM introduced

LLVM assembly is the target language of my project

*The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.* -llvm.org

The important part for my project is the LLVM assembly language. LLVM assembly is a hardware independent language for cleanly describing programs written in high level languages. LLVM assembly is used as follows; during the compilation of a high level language for example C, using the LLVM tools, the first thing that would happen is the translation of the C code to LLVM assembly. This results in the C program being expressed solely in LLVM assembly. This form of the program is a standardized form and can be dealt with consistently throughout the rest of the compilation. The LLVM assembly is used as the internal representation (IR) of the C program within the LLVM compiler. After this various optimizers and code generators can run to create a executable file for a target architecture. The benefits of this are that once you have a program that converts your source language to LLVM assembly you can make use of the rest of the tools without any modification to them or your code. All the optimizers and target architecture support can be utilized for any source language once the front-end translator has been written.

LLVM assembly is exclusively designed to represent high level languages, this leads to some key differences between it and other traditional assembly languages, which are designed to control real hardware.

LLVM is well supported and used in industry and academic research \cite{LLv09}. A particularly prominent user is Apple Inc. who use LLVM in some of their products. Apple Inc. are also active in the development of the LLVM project.

The LLVM project provides the reference manual for the LLVM assembly language on their website \cite{Lat11}. That was the main source of technical information on LLVM. The project mailing list and IRC channel were also used to gather extra information.
1 Introduction

1.2 Project Goals

The project’s aim was to investigate the potential for using LLVM as part of the binary translation process. LLVM can be used in binary translation by writing a translator from the source language (ARM assembly language in this project) to LLVM assembly. The LLVM compiler tools, including optimizers and code generators, can then be used on the produced LLVM assembly to create an executable file in the target architecture (X86 in the case of this project). Once this translator is written, statistics about the advantages of using this approach can be gathered. Additionally, the development of the translator will highlight the inefficient and difficult portions of this kind of binary translation. Therefore, the goals of this project are:

- Develop a binary translator that translates ARM assembly to LLVM assembly.
- Evaluate the benefits of using LLVM in the binary translation process.
- Analyze the difficulties involved in this approach.

1.3 Motivation

Using LLVM in binary translation will mean that a simple binary translator which does not attempt to optimize its produced code (thereby simplifying its development), will be able to make use of the powerful optimizers available in LLVM. Another important outcome will be that once the code is translated to LLVM assembly the LLVM tools can compile it to many target architectures. This project will be able to provide data on the performance of the LLVM optimizers on code translated from ARM. The development of a translator will expose the difficulties involved in this process, providing valuable information about the potential for future work in this area.

1.4 Readers Guide

<table>
<thead>
<tr>
<th>Chapter 1: Introduction</th>
<th>Introduce the project, its background, aims and motivation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 2: Design and Important Features</td>
<td>Outline the important parts of the technologies used, and the early design decisions.</td>
</tr>
<tr>
<td>Chapter 3: Implementation</td>
<td>The implementation details of the binary translator developed in this project.</td>
</tr>
<tr>
<td>Chapter 4: Analysis</td>
<td>The evaluation and findings of this project.</td>
</tr>
<tr>
<td>Chapter 5: Conclusion</td>
<td>A description of the outcomes of this project.</td>
</tr>
</tbody>
</table>
1.5 summary

This chapter has introduced the project and described its background, goals and motivation. The next chapter outlines some features of the languages used in the project and the high level design decisions taken at the start of the project.
2 Design and Important Features

This chapter discusses the technical aspects of the two languages (ARM and LLVM) that are key to the project and the design decisions those aspects influenced.

2.1 Features of ARM

Some parts of the ARM architecture played important roles in the project design and either created obstacles or aided in the conversion to LLVM. These parts include hardware in the ARM architecture and the nature of the assembly language. This section describes those features.

2.1.1 THUMB

Some versions of the ARM processor also support a compressed instruction set called THUMB. THUMB is used to keep code size small as this is important on embedded devices.

2.1.2 Registers

The register is the most fundamental storage area on the chip. As with most architectures these are the locations that the majority of instructions use as destination and source operands, other instructions may deal with loading from or storing to memory. At any one time, 15 general-purpose registers (R0 to R14) and the program counter (pc or R15) are visible.

R15: This is the program counter, writing to it causes a jump and reading from it gives the current position in the executable code. This register is furthest from a 'general purpose register', however it can still be used as a source or destination register in any instruction.

R14: This is the link register it is used by the BL (Branch and Link) instruction to store the current contents of the PC before branching to the target specified by the program. Otherwise this instruction behaves exactly as a normal register.

R13: This is used as the stack pointer, this is by convention used to point to next available byte of memory in the stack.
The other registers are general-purpose and are used for storing the values which the currently executing program is working on.

2.1.3 The Current Program Status Register.

The CPSR can be seen as the state of the machine. It contains status flags and other information indicating the current state of the machine, it can be used to evaluate conditions and is modified indirectly by many instructions.

The important flags in the context of this project are the four status flags, N (Negative), Z (Zero), C (Carry) and V (oV erflow). Only some instructions are capable of affecting the flags. Most arithmetic instructions may modify them based on the result of the arithmetic expression. Crucially these instructions only affect the status flags when a specific qualifier is used in the code. However, four instructions always affect the status flags CMN (compare negative), CMP (compare), TST (Test) and TEQ (test equality), these instructions only purpose is to set the status flags based on their operands, therefore the qualifier would be superfluous.

These flags are used to determine the results of conditional statements in subsequent instructions. They are also used in a few other instructions. For example, ADC is 'add with carry', it adds two operands but also adds the C flag to the result.

2.1.4 Conditional

In ARM any instruction can be conditionally executed. Any instruction can have a condition code appended to it. A condition code indicates that the instruction should only be executed if the CPSR coupled with that condition code (e.g. NE for not equal) indicate it should. Otherwise the execution skips over the instruction. This conditional execution of individual instructions is very useful for avoiding branches. For example:

<table>
<thead>
<tr>
<th>ARM code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP R0, #10</td>
</tr>
<tr>
<td>ADDNE R2, R1, #2</td>
</tr>
<tr>
<td>ADDEQ R1, R1, #1</td>
</tr>
</tbody>
</table>

- First instruction: Compares the contents of R0 with the value 10.
- Second instruction: Note the NE after the ADD. Based on the CMP instruction will add 2 to the value in R1 and store it in R2 only if the value in R0 is Not Equal(NE) to 10.
- Third instruction: Based on the same CMP instruction this will increment R1 if the value in R0 is Equal(EQ) to 10.

In this example, instead of conditionally branching over the first instruction if the two values were equal, and branching over the second instruction if they were not, both in-
Instructions are conditionally executed so no branches are necessary. Removing conditional branches brings big speedups and is the focus, along with predicting branches, of a lot of research.

2.1.5 Qualifiers

Qualifiers are extra letters that can be appended to specific instructions to change how they work. They are appended after the conditional (if there is one). A simple example is: \[ \text{LDMFD R13, \{R0, R1\}} \]. In this example the FD is a qualifier to the LDM (Load Multiple) instruction, it changes the way the LDM instruction behaves. Qualifiers for the LDM instruction indicate how it should load multiple values into registers starting from the memory address stored in the first operand (R13 in this example). For instance is R13 the lowest, or highest, memory address of the values being loaded. FD in this example means full descending and says that a value should be loaded into R0 from the address stored in R13. Subsequently a value should be loaded into R1, from the address obtained by adding 4 (4 bytes, 32 bits, the size of a register) to the value stored in R13. Other qualifiers here would indicate subtracting or adding, before or after, loading the first register.

2.1.6 Shifter Operand

ARM instructions may take variety of types of operand, an important type is the shifter operand. The shifter operand, as its name suggest, allows the value of the operand to be shifted. It has many forms (11 in fact) but there are three basic formats:

<table>
<thead>
<tr>
<th>Immediate operand value</th>
<th>The value is hard coded into the assembly.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register operand value</td>
<td>Simply a register, the value stored in the register is used as the operand.</td>
</tr>
<tr>
<td>Shifted register operand value</td>
<td>The value of a register shifted (or rotated) before it is used.</td>
</tr>
</tbody>
</table>

There are five shift types:
- LSL – Logical Shift Left
- LSR – Logical Shift Right
- ASR – Arithmetic Shift Right (sign maintained)
- ROR – ROtate Right
- RRX – Rotate Right with eXtend

The number of bits to shift by is specified either as an immediate value or as the value in a register. An example instruction would be \[ \text{MOV R2, R0, LSL \#4} \]. This would result in the value of R0 being shifted left by four bits then being stored in R0. \[ R0, LSL \#4 \] is the shifter operand in that example. The shifter operand has many purposes it is used for supplying immediate values to instructions and is the only way to shift values.
2 Design and Important Features

in ARM. A common use of the shifter operand would be to index into an array using the following command: 
\[ \text{ADD R1, R0, LSL #2} \] In our example we assume R1 is the start address of the array. This instruction would take the value of R0 (our index into the array), shift it left by two (the equivalent of multiplying by four) and add it to R1. As each element of the array is 4 bytes long, shifting R0 left by two bits means converts it from the index of an element in the array, to the memory offset from the start of the array to the element at that index.

2.1.7 Call and Return

The usual control flow during a call and return is that when a program 'calls' a subroutine or function, the code in that subroutine starts being executed, when the subroutine is finished it will 'return' to where it was originally called from. To accomplish this in ARM the BL (branch and link) instruction is used. The BL instruction branches to its target and it also stores the current address, which will be the return address, into the link register(R14). It is up to the programmer to deal with the contents of the link register appropriately (i.e. store it to the stack if another BL is to be called). To return, the program simply branches to the address stored in the link register. This returns control to the point at which the original BL instruction was executed.

2.2 Features of LLVM

This section introduces some technical aspects of LLVM which are important to the design of the project.

2.2.1 Basics

Local variables in LLVM always start with a '%'. The variables are typed, types usually look something like 'i32' or 'i8*', meaning a 32 bit integer and a pointer to an 8 bit integer respectively. A typical instruction looks like this: 
\[ \%\text{result} = \text{add} \ i32 \ \%\text{variable1}, \ \%\text{variable2} \] i.e. add the 32 bit integer variables, %var1 and %var2, and store them in the variable %result.

2.2.2 SSA

LLVM is Static Single Assignment which means each variable is assigned exactly once. LLVM expects increasing numbers to be used as temporary variable names. These temporary variables are used to store partial results in complicated expressions. For example the C code \(x=2+5*8\) would translate to:
SSA form is very useful for the automated optimization of code as it is very clear to a compiler the structure of the assignments in a program.

If a variable is to be stored to multiple times it must be placed in memory i.e. the store instruction above stores to a memory address pointed to by the pointer variable %x, the actual memory address that x points to can only be assigned once. However, the contents of memory at that location can be changed.

### 2.2.3 Call and Return

In LLVM the call and return mechanism is very similar to a C style function call. A typical call in LLVM looks like this: \[
\%\text{retval} = \text{call} \ i32 \ @\text{test}(i32 \ \%\text{argc})
\]
This would call the function test, which has the return type i32, passing the variable %argc as an argument. To return in LLVM the `ret' instruction e.g. \[
\text{ret} \ i32 \ 0
\]
would return the 32 bit value 0.

### 2.2.4 Comparisons

Comparisons in LLVM are done with the icmp instruction, it takes 4 operands; what comparison is to be done (e.g. less than or equal), the type of the values to be compared (e.g. 32 bit integer) and the two values to be compared. The instruction results in a boolean value which can be used in a conditional branch instruction. For example the LLVM code \[
\%\text{booleanresult} = \text{icmp} \ \text{ge} \ i32 \ \%\text{var}, 5
\]
would compare the value stored in the %var and 5, if %var was greater than 5 the result(a 1 bit integer) would be 1.

### 2.3 Design decisions

Some high level decisions made about the development of the program and the project.

#### 2.3.1 THUMB

It was decided THUMB would not be supported by this project. The translator could be extended to support it using the same methods used in this project.
2.3.2 Error Checking

It was decided that all the input ARM assembly would be assumed to be syntactically correct. Although there is some error checking in the parser it is primarily to aid development and is far from exhaustive. This decision was made to simplify the parsing process and because verifying the input ARM assembly does not aid in examining the benefits of LLVM in binary translation.

2.3.3 Status Flags

The Current Program Status Register in ARM, which is described in 2.1.3, contains the status flags. The status flags are used to compute the results of comparisons and by some special instructions. As such they are an important part of the architecture. The binary translator in this project has to output LLVM code that replicates the behaviour of the status flags and the instructions that interact with them. The method that was chosen to do that was to maintain variables representing each of the status flags. These virtual status flags would be kept up to date throughout the program, by outputting LLVM instructions that would calculate what their values should be and store them. The LLVM code for updating the virtual status flags would only need to be created when translating an ARM instruction that would update them. Alternatives to this approach are discussed in 3.4.7.

2.3.4 Call and return

The call and return mechanisms for LLVM (2.2.3) and ARM (2.1.7) are quite different so a decision had to be made about how the ARM version would be translated to LLVM. There were two apparent possibilities:

- Use the LLVM call and return mechanism. Make the ARM BL instruction translate to an LLVM ‘call’ instruction and then recognise a branch to the return address in the ARM code and replace it with a LLVM ‘ret’ instruction.

- Replicate the behaviour of the ARM BL instruction without using the LLVM ‘call’ instruction. Instead generate equivalent code out of LLVM branches and other instructions.

The second option was chosen; to try and replicate the ARM behaviour without using the LLVM ‘call’ instruction. This was done because of the many incompatibilities between BL and ‘call’. Consider the following ARM code snippet.
2 Design and Important Features

In this code the program branch and links to the subroutine. Within that subroutine a comparison is made and a conditional branch to the errorhandler is made. That conditional branch is not a call but a direct branch. If the branch and link was translated to LLVM as a call instruction then the errorhandler label would be out of scope within the subroutine once it was translated to LLVM.

2.3.5 Recognizing instructions

The basic design for recognizing instructions in the parser is to use a Finite State Machine (FSM) which if graphed would look like a tree structure. The FSM is traversed by taking the transition which matches the next character in the input. If that is not a valid transition or the next character is not a letter then the contents of the current node is considered to contain the relevant information about the instruction being parsed. If the character string was not a valid instruction then the node that the traversal ends up on will contain a value indicating that fact (The nodes value will be -1), and the parser will output an error. If the character string was valid the node will contain some values that represent the instruction (primarily an integer which arbitrarily represents the instruction within the program).

2.4 Summary

This chapter has outlined the most important features of ARM and LLVM assembly Language and some key design decisions about how to deal with these features. The next chapter describes the implementation of an ARM assembly language to LLVM translator.
3 Implementation

This chapter deals with the technical implementation of the binary translator. It details the implementation of some of the design decisions discussed in chapter 2 and the major components of the translator.

3.1 Overview

The translator was split into two programs. The Parser which read the ARM assembly file and generated an internal representation of the ARM assembly code.

The Generator which generated LLVM code from the IR created by the Parser.

A finite state recogniser is used to recognise different types of input; instructions and condition codes.

3.2 Finite State Recogniser

This is a data structure which is used to recognise valid sequences of letters and store some basic information about those sequences (e.g. an integer representing an instruction). This data structure is made of nodes containing an array of 26 pointers, each of these pointers represents a letter. If a pointer is NULL then that letter is not part of a valid sequence. The nodes also contain 3 values which are used to store information about the sequence that has led to this node. This data structure is used for recognizing two different types of sequence, namely instructions and condition codes, but if the first value is -1 then that node does not represent a valid sequence for whatever is being recognized. This tree-like data structure is initialized from hard coded information when the parser starts.

To use the data structure the program passes the next character in the sequence to the traverse (char) function on the current node (starting with a root node). If this returns a NULL pointer then that letter does not represent a valid addition to the sequence (this does not necessarily indicate an error, the sequence might be followed immediately by some other valid letters). If it returns a pointer to another node then that is switched with the current node and the process continues until either there are no letters left or a NULL is returned. The following is a simplified use of this data structure:
3 Implementation

\texttt{C++ code}
\begin{verbatim}
fir->insert(string("add"),30,1,0); temp=fir->traverse('a'); temp=temp->traverse('d');
tmp=temp->traverse('d'); result=temp->value;
\end{verbatim}

In this example result end up being equal to 30. The first line adds the string “add” to be recognised. The traverse function returns the node pointed to by the character passed.

3.3 Parser

The parser breaks down the various parts of a line of ARM assembly and creates an object which contains all the information necessary to generate an LLVM version of that line. An ASMLine object is declared and initialized with the string from the ARM assembly, the interpret (IR*) function is then called on that ASMLine object passing a pointer to an empty IR object. During the interpret functions execution the ASMLine object is populated with data about the string that it was initialized with. At the end of the function the memory at the IR pointer, which is passed into the function, is initialized with the relevant information for generating the LLVM. The main function then writes that IR object to file and moves on to the next line.
The following sections describe how each of the parts of an ARM assembly line are parsed.

### 3.3.1 Labels

Labels are the first assembly element to be checked for; the `interpret()` function looks to see if the line contains the `:` character, if it does the line is assumed to be a label. If the line is a label a boolean is set to indicate that and the characters before the `:` are stored. When the IR is created this boolean and the stored label are copied to it.

### 3.3.2 Instructions

The `consumeInstruction()` function is used to parse the first chunk of an ARM assembly line. The first thing that the function parses is the actual instruction this is done using the FSR described in 3.2. The three values stored in the node represent respectively; an identifier of the instruction, whether the instruction is capable of having the S flag appended to it and whether the instruction needs to be checked for special qualifiers.
3 Implementation

The value of the instruction is stored and copied to the IR when it is created, the other two values are used to simplify the parsing.

3.3.3 Conditionals

The conditionals are also detected in the `consumeInstruction()` function as they are lumped together in ARM assembly. Another FSR is used to detect which conditional (if any) is being appended to the instruction. This one has been populated with the various condition codes and a single value representing that condition code. This value is stored (-1 indicates the instruction is not conditional) and that value is copied to the IR when it is created.

3.3.4 Qualifiers

Some instructions may take additional qualifiers, the most common qualifier is an 'S' at the end indicating the instruction should set the CPSR. Qualifiers are detected using an if statement based on what instruction was recognised as discussed in 3.3.2 and what values were stored in FSM node for that instruction. An example is the LDM instruction, it must take one of four qualifiers, this is noted by the third value in its FSR node being 1. After the potential conditional has been parsed the program checks if each one of these four qualifiers follows, if it does an appropriate value is set in the ASMLine object which is copied to the IR object when it is created.

3.3.5 Operands

Once the instruction, its conditional and its qualifiers have been parsed the Parser moves on to the operands. Each instruction has its own set of operands. For example, some instructions only take one register while others may take two registers and a shifter operand. Each instruction has a short piece of code that directs the parsing of its operands, an example of this is the following piece of code:

```c++
numOps=2;
consumeWhite();
er(consumeReg(operands[0]));
consumeOpSpace();
er(consumeReg(operands[1]));
consumeOpSpace();
er(consumeShifter());
```

This example is for the ADD instruction. The numOps variable stores the number of operands to be copied to the IR (does not include the shifter operand). The `er()` function, is for dealing with errors in the mandatory parts of the instruction. The `consumeWhite()`
3 Implementation

and `consumeOspce()` functions just parse white space and commas etc. The other functions are dealt with in their specific subsections below. A version of this code for every instruction is in a switch statement and the Parser switches to the correct version based on the instruction parsed earlier.

Registers

Registers can take the form R0, R1 or a name can be used for the last three registers 'sp', 'lr' or 'pc'. The `consumeReg(int&)` function is used to parse these, it stores the register number in the variable passed to it, this function will produce an error if it is not possible to parse a register from the part of the string pointed to by the pointer in the ASMLine.

Shifter Operand

The shifter operand is used primarily in arithmetic instructions and is discussed in 2.1.6. The `consumeShifter()` function is used to parse the shifter operand. It can parse all the possible variants of the shifter operand and stores the resulting data in in the ASMLine object to be copied to the IR later.

Address

Another form of operand is the memory address operand, it is used to store and load, to and from, memory. The `consumeAddress()` function is used to parse it. There are 9 possible forms of this operand:

- Immediate offset: `[Rn, #+/\-<offset>]`
  Rn is the base register. The offset is added to Rn and the result is the address that is used.

- Register offset: `[Rn, +/-Rm]`
  Rn is the base register. Rm is added(or subtracted) to Rn and the result is the address that is used.

- Scaled register offsets: `[Rn, +/-Rm, <shift> #<shift immediate>]`
  Rn is the base register. Rm is shifted in the fashion indicated by <shift> by <shift immediate> number of bits. That is added(or subtracted) to Rn and the result is the address that is used.

- Immediate pre-indexed: `[Rn, #+/\-<offset>]!`
  Rn is the base register. The offset is added to Rn and the result is the address that is used. The result is also written back to Rn.
3 Implementation

- Register pre-indexed: \([R_n, +/-R_m]\)
  Rn is the base register. Rm is added(or subtracted) to Rn and the result is the address that is used. The result is also written back to Rn.

- Scaled register pre-indexed: \([R_n, +/-R_m, \langle shift \rangle \#\langle shift \text{ immediate} \rangle]\)
  Rn is the base register. Rm is shifted in the fashion indicated by \(<\text{shift}>\) by \(<\text{shift \ immediate}>\) number of bits. That is added(or subtracted) to Rn and the result is the address that is used. The result is also written back to Rn.

- Immediate post-indexed: \([R_n], \#\langle offset \rangle\)
  Rn is the base register and is used as the address. The offset is added to Rn. The result is written back to Rn afterwards.

- Register post-indexed: \([R_n], +/-R_m\)
  Rn is the base register and is used as the address. Rm is added(or subtracted) to Rn and the result is the address that is used. The result is written back to Rn afterwards.

- Scaled register post-indexed: \([R_n], +/-R_m, \langle shift \rangle \#\langle shift \text{ immediate} \rangle\)
  Rn is the base register and is used as the address. Rm is shifted in the fashion indicated by \(<\text{shift}>\) by \(<\text{shift \ immediate}>\) number of bits. That is added(or subtracted) to Rn and the result is the address that is used. The result is written back to Rn afterwards.

Target

The target operand is the operand for branch instructions, it represents the target address that the instruction will branch to. This operand is part of one of the major issues encountered during this project and its role is discussed further in 3.4.6. The \texttt{consumeTarget()} function is used to parse this, it can take the form of a register or label.

3.4 Generator

The code generator is the more complex of the two programs in this project. It takes the internal representation (IR) created by the parser and turns it into valid LLVM code. Each piece of IR is put into an array. Then the array is iterated over producing the required LLVM for each piece of IR. Each piece of IR must either represent an ARM instruction or a label.
3 Implementation

The various parts of the generator and the potential tasks the LLVM it generates must perform are discussed in this chapter.

3.4.1 The Basics

The basic processes for generating the LLVM code is to output a string made up of some hard coded strings and the contents of some variables. As mentioned earlier LLVM is static single assignment based. This means that all the partial results involved a calculation must be stored in unique variables. In LLVM these temporary variables are sequentially numbered. In this code generator the current lowest unused temporary variable number is stored in a global variable called `tempnum'. The tempnum value is incremented every time it is used. An example piece of C++ code that outputs an LLVM instruction follows:

```c++
file"%"tempnum++" = add i32 %"first", "%ir->shifterValue"endl;
```

The % signs are always at the beginning of an LLVM variable. The tempnum at the beginning is the destination temporary variable for the result of this instruction, it is post incremented. The contents of that variable will be stored to memory in the next instruction. The value of 'first' is the number of the temporary variable associated with the first number to be added. The ir->shifterValue is the immediate value to be added to the first. Partial results are often accessed by subtracting from the tempnum variable to get the result of a previous instruction.
3 Implementation

3.4.2 Labels

The first thing that is done by the generator is that it checks whether the IR it is working on is a label. If it is, the name of the label which is stored in the IR is outputted to the LLVM file in the LLVM label format. As labels denote a code block in LLVM and every code block must end in a 'terminating instruction' a branch to the new label is also outputted directly before the label itself. At this point the generator moves on to the next IR so the remainder of this chapter assumes the IR is an instruction.

3.4.3 Conditional

The next step is to deal with the conditional execution. If the IR indicates the instruction is conditional then LLVM code must be generated that does the necessary comparison as discussed in 3.4.7. Then a conditional branch based on the result is outputted. That branch jumps to one of two labels, the first label is if the comparison is true, that label is outputted directly after the branch. The second label is outputted at the end of the code generation for the current IR, branching to this second label will ignore the instruction in the current IR completely.

3.4.4 Shifting

ARM does not have instructions for shifting values. Instead shifter operands are used, a shifter operand can take additional values and keywords that indicate how the value should be shifted before it is used in the instruction. The LLVM code to prepare the shifter operand, if necessary, is the same for any instruction. The Generator outputs the code to do the relevant shifting and stores the number of the temporary variable that the result is stored in. That stored result can be used in the instruction that will actually use the the shifter operand.

3.4.5 Instructions

The main part of code generation is the creation of the LLVM code that is equivalent to the actual ARM instruction stored in the IR. A large switch statement is used to select the correct process to turn the IR into LLVM, this switch statement selects based on the what instruction is stored in the IR.

3.4.6 Branching

Branching is a problematic area in this project. The reason this is difficult is that ARM can jump to absolute addresses using the BX instruction. The BX instruction is used by ARM to do an indirect branch, this instruction is not fully implemented in this project.
as it is also used to switch to a different instruction set architecture which is described in 2.1.1 however its functionality for the purpose of returning from a branch and link (BL) instruction is translated to LLVM.

If an ARM program uses the BL instruction, the intended functionality is that the current address will be stored in R14 and then the execution flow will jump to the BL instructions target. This instruction is used as the Call part of the call and return mechanism. To return, the program uses the BX instruction to jump to the address that was stored in the link register (this value may have been stored and retrieved from memory for a subroutine that also uses the BL instruction). That behaviour is translated in the following way: When a BL instruction is translated a specific numbered label is placed after it as a return target. LLVM code to store the number of this label into the link register is outputted, then a normal branch to the BL’s target is outputted. Once a BX to a register is encountered, the translator outputs code to branch to a return mechanism which matches the number in the target register to a return label and branches there.

The following ARM code:

<table>
<thead>
<tr>
<th>ARM code</th>
</tr>
</thead>
<tbody>
<tr>
<td>subroutine:</td>
</tr>
<tr>
<td>ADD R0,R0,R0</td>
</tr>
<tr>
<td>BX lr</td>
</tr>
<tr>
<td>main:</td>
</tr>
<tr>
<td>MOV R0,#4</td>
</tr>
<tr>
<td>BL subroutine</td>
</tr>
<tr>
<td>add R0,R0,#0</td>
</tr>
</tbody>
</table>

is translated into the following LLVM code:
Before the branch to subroutine, the value 1 (the return address) is stored into register 14 (the link register). In the subroutine some work is done then the program branches to the return mechanism which selects the correct return address to branch to.

This method allows the number to be moved around and stored in memory before being returned to a register to be used by a BX instruction. This means the return address can be treated exactly like it is in the source ARM program without any further interference from the translator. It also allows for practices like tail call elimination to work. The primary problem is that if the number is modified it will behave unexpectedly. Also numbers that are calculated in some other way than a BL instruction cannot be used. This problem is discussed further in 4.1.1.

3.4.7 Virtualisation

Although some of the translation is quite simple, for example translating the add instruction, there are some significant parts where necessary virtualisation of the ARM hardware makes things interesting. The following sections deal with the prominent examples of this.
3 Implementation

Registers

The registers are simply memory locations allocated at the start of the LLVM and pointed
to by pointers in the variables names \texttt{%R0 - %R15}. This means they are easily indexed
with numbers that are stored in the IR.

Program Counter

The program counter (PC) is one of the largest problems with the binary translation
from ARM to LLVM. This section outlines the way this problem is dealt with in this
project, the problem and possible solutions are discussed further in 4.1.1. The approach
to branching is discussed in 3.4.6. In ARM the PC can be accessed as \texttt{R15} and can be
written to (causing a jump) or read from (retrieving the current position in the code).
That behaviour is essentially ignored by this project, branches must be to labels or to
the return address after a branch and link. The PC does not contain the current address
and its contents should be considered undefined. Writing to it is the same as writing to
a general register and causes no jumps.

Current Program Status Register

The CPSR which is described in 2.1.3 is important in the ARM architecture. It is used in
comparisons and some other instructions use it to allow greater than 32 bit calculations.
LLVM has no analog of the CPSR, its comparisons result in a boolean value (a single bit
integer) and are treated like normal values.

It was decided that the CPSR should be virtualised in LLVM. This decision is discussed
in 2.3.3. Unfortunately it is difficult to generate the flags contained within the CPSR
without access to the hardware.

The status flags in the CPSR are as follows:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C flag</td>
<td>Carry flag</td>
</tr>
<tr>
<td>V flag</td>
<td>Overflow flag</td>
</tr>
<tr>
<td>Z flag</td>
<td>Zero flag</td>
</tr>
<tr>
<td>N flag</td>
<td>Negative flag</td>
</tr>
</tbody>
</table>

The process for obtaining each of these flags after an ADD is as follows:

First the carry out of the MSB and the carry in to the MSB are needed. To calculate
the carry in the easiest thing to do is clear the MSB of the two addends and then add
them. The carry in is the MSB of the result. The carry out can be calculated by logically
shifting both addends right by 31 bits, adding them, then adding the carry in. The result
is shifted right by one bit and the result of that is the carry out.
3 Implementation

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overflow</td>
<td>The carry in and carry out are exclusive ORed.</td>
</tr>
<tr>
<td>Carry</td>
<td>This is simply the carry out.</td>
</tr>
<tr>
<td>Zero</td>
<td>The equality comparison between the result of the operation and zero.</td>
</tr>
<tr>
<td>Negative</td>
<td>The result of the operation logically shifted right by 31 bits.</td>
</tr>
</tbody>
</table>

This process is slightly modified for the SUB command (the second operand is negated first) but this is the basics of how it is done for all the instructions which require it. Some instructions do not set the carry or overflow flag, they are unchanged.

Comparisons

Comparisons in ARM are used only for conditionals. As the conditional flags are virtualised it is a simple matter to work out the result of a comparison from the flags. The following table shows how the comparisons use the flags:

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>Z=1</td>
<td>Zero (Equal to 0)</td>
</tr>
<tr>
<td>ne</td>
<td>Z=0</td>
<td>Not zero (Not Equal to 0)</td>
</tr>
<tr>
<td>cs/hs</td>
<td>C=1</td>
<td>Carry Set / unsigned Higher or Same</td>
</tr>
<tr>
<td>cc/lo</td>
<td>C=0</td>
<td>Carry Clear / unsigned Lower</td>
</tr>
<tr>
<td>mi</td>
<td>N=1</td>
<td>Negative (Minus)</td>
</tr>
<tr>
<td>pl</td>
<td>N=0</td>
<td>Positive or zero (Plus)</td>
</tr>
<tr>
<td>vs</td>
<td>V=1</td>
<td>Sign overflow (Overflow Set)</td>
</tr>
<tr>
<td>vc</td>
<td>V=0</td>
<td>No sign overflow (Overflow Clear)</td>
</tr>
<tr>
<td>hi</td>
<td>C=1 &amp; Z=0</td>
<td>Unsigned Higher</td>
</tr>
<tr>
<td>ls</td>
<td>C=0</td>
<td>Z=1</td>
</tr>
<tr>
<td>ge</td>
<td>N=V</td>
<td>Signed Greater or Equal</td>
</tr>
<tr>
<td>lt</td>
<td>N != V</td>
<td>Signed Less Than</td>
</tr>
<tr>
<td>gt</td>
<td>Z=0 &amp; N=V</td>
<td>Signed Greater Than</td>
</tr>
<tr>
<td>le</td>
<td>Z=1</td>
<td>N != V</td>
</tr>
</tbody>
</table>

LLVM code to do the necessary checks on the flags is outputted which stores the boolean result to a temporary variable.

Stack

At the start of the outputted LLVM assembly code, stack space is allocated. The address of the highest location of that memory is loaded into the stack pointer register. This means the stack can grow and shrink naturally within the allocated memory. However it is possible for the program created by the result of this translator, being compiled, to overflow this stack.
3 Implementation

3.5 Summary

This chapter showed the implementation of the translator, which was developed and used in this project. The next chapter presents the findings of this project.
4 Analysis

This project set out to investigate using LLVM as part of the binary translation process. To do that a binary translator from ARM to LLVM was developed. The development process highlighted the difficulties in such a translation and led to the research of possible solutions. That translator then went on to be used to gather statistics about the performance of the LLVM optimizers. This chapter describes the results of the project.

4.1 Obstacles to Translation and Possible Solutions

The development of the translator helped uncover incompatibilities between ARM and LLVM. These incompatibilities made it difficult to translate some instructions effectively and efficiently. These issues and the possible solutions that were developed are key parts of the project. The following sections detail the issues.

4.1.1 Program Counter

As discussed earlier in the report the program counter and branching through the executable code are difficult areas to translate. This is primarily due to the fact that these instructions are heavily involved with the ARM hardware and can dynamically change the control flow. The value in the Program Counter (PC) refers directly to a memory address where the code is stored. Every instruction has exactly one memory address associated with it. However in the translated LLVM version a single ARM instruction may break down into many LLVM instructions which then subsequently will go through optimizations.

Part of the problem is that LLVM is designed to represent high level languages which wouldn’t have direct access to the PC, therefore it doesn’t have the instructions to manipulate it in a dynamic way. It is left to the LLVM target architecture compiler to deal with those details much like in a high level language.

A value can be written to the PC to cause an indirect branch. Similar behaviour can be obtained by using the BX instruction to branch to the location stored in a register. Writing to the PC does not supply any unique behaviour however it is sometimes done by the LDM instruction. This allows registers to be restored at the end of a subroutine while simultaneously returning from the instruction. For example:
In this example the registers r4 to r7 are stored to memory at the start of the subroutine to preserve their contents. The contents of the link register (lr) is also stored to memory. The link register will contain the return address. At the end of the subroutine the contents of the registers that were stored at the start are restored. However, the contents of the link register that was stored to memory is loaded into the PC causing a branch to the return address.

A possible solution to the difficulty of translating this behaviour of the ARM architecture is an extension of the approach discussed in 3.4.6. Before generating the code for each ARM instruction a label could be generated with a number representing that instruction's address, the number would increment with each instruction and therefore be a reasonable approximation of the ARM PC. Any reads from the PC would be replaced with the value of the next instruction to be outputted. Writes to the PC or the BX instruction would be replaced by a jump to a special section of LLVM code that would use the LLVM switch statement to jump to the appropriate ‘program counter label’. Modifying stored target addresses would also produce the correct results as there would be a 1:1 ratio between these ‘program counter labels’ the original ARM instructions. Although this approach would solve a lot of the problems it would introduce its own primarily to do with the further compilation process. For example the LLVM compiler uses labels to indicate code blocks, putting so many into the program would likely hinder the optimization process. Another problem is that the optimizations reorganize the code significantly, potentially breaking the code. It would also be rather crude and introduce a huge amount of redundant work virtualising the ARM PC when there is sure to be a PC in the target architecture.

4.1.2 Calling to External Libraries

A problem that hasn’t been touched on before in this report is the difficulty of calling functions outside of the code being translated, in common libraries etc. The problem here lies in the calling convention of the ARM versions of these libraries. A brief example would be: the C function putchar (char) takes a character and prints it to standard out. Converting a call to this function into LLVM is simple, declare the function (i.e. its arguments and return type) and then simply: `%result = call @putchar(i32 65)` This would print the character ‘A’ to standard out. The same instruction converted to ARM would look like this:


4 Analysis

<table>
<thead>
<tr>
<th>ARM code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0, #65</td>
</tr>
<tr>
<td>BL putchar</td>
</tr>
</tbody>
</table>

This is also quite simple, however the problem is converting the ARM version to the LLVM version, crucially knowing which registers need to be used as arguments in the LLVM ‘call’ instruction. In ARM what arguments go where is understood by the programmer or the compiler is aware of the calling convention. However, the translator does not know what the target architecture will be and does not know the details of every possible ARM library function call. Two possible solutions are as follows:

- Translate the necessary libraries along with the program, thus the program would not have to worry about the LLVM call instruction at all and could continue to use its BL instruction. Unfortunately this would most likely lead to having to translate many libraries with every program and would be quite impractical and inefficient.

- The better way is to make the translator aware of the libraries. For example, if the translator knows a call is meant to take X arguments the translation of that call can use the contents of the first X registers as arguments. This would essentially require supplying the libraries header file. However, again this quickly could get out of hand as libraries from all the different programming languages would have to be parse-able by the translator. A middle ground would be supplying support for a common set of libraries. Again this problem is born out of the fact that this translator is trying to go from a very low level language, where the `calls’ to functions are just placing the values in the correct registers and branching to a label, to a language with a C-like “call x(y z)” syntax.

4.1.3 Virtualising the Current Program Status Register

The virtualisation of the CPSR in this project is discussed in 3.4.7. The solution used in this project is fully functional but not particularly efficient. The current solution adds a lot of additional instructions to perform a task that the target architecture probably has hardware to do.

The status flags in the CPSR are used for computing comparisons as well as some instructions such as ADC(add with carry) making direct use of them. Some alternative methods to translate instructions which interact with the CPSR are:

- After any instruction which sets the status flags in ARM, the LLVM version computes the results of each of the ARM conditional comparisons (EQ, NE etc.) based on that instruction. If a conditional was used later in the ARM code it would merely have to lookup the result of the precomputed comparison. This solution means all the effort is concentrated on the instruction which sets the CPSR flags, these instructions will generally be less common than conditionals\footnote{This is because in sensibly written code no instructions will set the CPSR unnecessarily, i.e. without a conditional that uses its result, however many conditionals may rely on a single instruction that} However it

27
could do a lot of unnecessary work as some of the conditionals computed may not be used. The LLVM version would use the icmp instruction to compute the results of the comparisons. Using the icmp instruction means that the final program could use the target's architecture hardware for doing the comparisons. Unfortunately, this solution is only valid for comparisons and not instructions which use the status flags directly.

- When any instruction which sets the status flags in ARM, is translated, the LLVM version remembers the instruction and its operands. When the contents of the CPSR is needed the necessary flags are computed from the saved instruction and operands. This is termed Lazy Flag Updating\cite{CWZ11}. This method has the advantage that, as the translator knows what flags are required only those flags have to be generated.

- A third way to translate these instructions is to combine the first two. Using the concept of Lazy Flag Updating, when an instruction sets the status flags that instruction is simply saved along with its operands. If the translator translates an ARM instruction that requires a comparison, the LLVM comparison instruction is used appropriately on the stored operands. However, if a instruction like ADC is translated the relevant flag itself is generated and used (in the case of ADC the Carry flag). This has the advantage that if a comparison is being done it can be done by the native hardware of the target architecture as the LLVM compiler will compile the LLVM icmp instruction to use it.

The third solution should give the best performance with a minimum of unnecessary computation.

### 4.2 Translated Program Performance using LLVM Optimizations

One of the motivations for this project was to be able to make use of the LLVM optimizations during the binary translation process. To demonstrate the effect LLVM optimizations have on the speed of the translated code some sample programs were translated then compiled, both with and without the LLVM optimizations. In all the graphs in this section native refers to a program compiled directly from a C file to an X86 executable using an LLVM compiler (Clang). Unoptimized and Optimized refer to a programs which have been run through the binary translator and then compiled without and with optimizations.
4 Analysis

4.2.1 Bubble Sort

This program is a simple bubble sort program running on a large array which is already inversely sorted. The program was compiled from C to ARM using Clang, it was also compiled directly to an executable using Clang. The ARM version was translated using the binary translator to produce an LLVM file. That LLVM file was then compiled, using the LLVM tools, to an X86 executable, this was done both with and without optimizations. Figure 4.1 shows the speeds of the native (i.e. compiled straight from the original C file to X86) version and both the optimized and unoptimized binary translated versions. The graph shows that although the binary translated versions are much slower than the native version, the optimization roughly halves the run time.

![Bubble Sort Run Time](image)

Figure 4.1: Run times of three versions of the bubble sort program
4 Analysis

4.2.2 Insertion Sort

This program is similar to the one above but is an insertion sort. The same method was used to obtain three versions of the program. A similar result can be seen in Figure 4.2 as in the previous program. The native version is the fastest but the LLVM optimization roughly halves the run time from unoptimized.

![Insertion Sort Run Time](image)

Figure 4.2: Run times of three versions of the insertion sort program
4.2.3 Binary Search

This program runs multiple binary searches over a large array. The three versions were obtained and the results in Figure 4.3 are consistent with the previous sample programs.

![Binary Search Run Time](chart)

Figure 4.3: Run times of three versions of the binary search program
4.2.4 Fibonacci

For this program a slightly different approach was taken. A Fibonacci calculator which calculates a range of Fibonacci numbers repeatedly was written by hand in ARM assembly. This process provides a program with a very small code size. The ARM file was translated then compiled as before. Figure 4.4 shows that the optimizations did better than halve the time as they had for the previous programs.

![Fibonacci Run Time Chart](chart.png)

Figure 4.4: Run times of two versions of the fibonacci program
4 Analysis

4.2.5 Fannkuch-redux

This program was taken from the “computer language benchmark game” website and is called fannkuch-redux. The original code was in C on the website. That C code was compiled to ARM as well as directly to X86 using Clang. Figure 4.5 shows the results.

![Fannkuch Run Time](image)

Figure 4.5: Run times of three versions of the fannkuch program
4 Analysis

4.2.6 Code Size

Another metric that was gathered was code size. The code size for the optimized, unoptimized and native versions of the above programs are shown in Figure 4.6. The results are similar to the run time results; the native is the smallest but the LLVM optimizations can significantly decrease the code length. The fibonacci program has no native compile size as it was handwritten in ARM whereas the others were compiled from C.

![Code Size]

Figure 4.6: Code size for various example programs, showing optimized, unoptimized and native code size.

4.2.7 Evaluation

These optimizations are clearly not part of this project but these graphs do suggest that using the LLVM optimizers could be helpful in binary translation. The assumption that could be made is that a naive binary translator that translated from ARM directly into X86 would produce code as bad as the unoptimized versions of the sample programs. This is a reasonable assumption to make, without any optimizations built into it, the translator is going to produce a very naive translation to X86, by it’s definition it wont
4 Analysis

reuse registers or rearrange code or any other optimizations. With the use of the LLVM optimizers, a naive binary translator which is much easier to write, can convert a source assembly language into LLVM and still make use of a lot of the possible optimizations available when its compiled to any number of target architectures.

4.3 Summary

This chapter laid out the benefits and difficulties of this approach. The next chapter wraps up the project.
5 Conclusions

5.1 Learning Outcomes

Coming to the project with no knowledge of ARM assembly means the most obvious area I have learned about is the ARM assembly language and underlying architecture. I had never seen ARM assembly code before starting this project and although the project didn’t require me to write much ARM code I did have to gain a very good technical understanding of its syntax and semantics to be able to translate it. I have also become familiar with the LLVM assembly language as I had to generate the LLVM equivalent of the ARM code that I parsed. During the debugging process I was often working with 3 assembly languages ARM, LLVM and X86 (which I have also never dealt with in detail before) at the same time. I had to track the translation by the translator from this project, as well as the LLVM compiler, through all three versions of a program. Beyond that which are really incidental areas to the project, I learned a lot about areas of compiler design and code generation which is harder to quantify.

5.2 Future Work

The issues mentioned in Chapter 4.1 are obvious candidates for additional work. The possible solutions could be explored and evaluated.

Currently the binary translator is not aware of any kind of assembler directive, adding that would aid the translation process significantly.

There are various concepts which apply to binary translation in general which could be integrated into this approach.

5.3 Contributions

This project has successfully created a working ARM to LLVM binary translator. That translator has allowed the project to generate statistics on the effect that LLVM optimizations have on the output of such a translator.

The development of the translator has highlighted the difficulties of using LLVM in this manner and the primary considerations that have to be taken into account when
considering this approach to binary translation. This research has been documented within this report in the Analysis chapter.
Bibliography


6 Attached CD

The source code for the translator can be found on the CD attached to the hardcopy version of this report.