



College of Business, Technology and Engineering

Department of Computing

Project (Technical Computing)

[55-604708]

2021/22

Author:	Barnaby Wilks
Student ID:	28017695
Year Submitted:	2022
Supervisor:	Adrian Oram
Second Marker:	Andy Hamilton
Degree Course:	BSc Computer Science
Title of Project:	Compiler Optimization Techniques

Confidentiality Required?

NO

I give permission to make my project report, video and deliverable accessible to staff and students on the Project (Technical Computing) module at Sheffield Hallam University.

YES

Acknowledgements

Firstly, I would like to thank my supervisor Dr Adrian Oram. I would also like to thank my friend Robert Graham, whose hardware was used for testing.

Abstract

The aim of this project was to investigate and implement techniques and systems utilised by optimising compilers. Over the course of the project a compiler was developed for a subset of the C language, targeting the ARM platform. Given a functioning compiler, benchmarks were then written, profiled and optimisation techniques applied to improve the runtime performance of the code.

The conclusion is that, whilst this project was successful in its goal, this only touched the surface of what goes into modern optimising compilers, and there is much scope for expansion.

Glossary

AOT – Ahead of Time (Compilation)

AAPCS/AAPCS32 – Arm Architecture Procedure Calling Standard (32 bit)

GCC – GNU Compiler Collection

HLIR – High Level Intermediate Representation, platform independent IR used in the Helix Compiler.

IR – Intermediate Representation

JIT – Just In Time (Compilation)

JVM – Java Virtual Machine

LLIR – Low Level Intermediate Representation, ARM specific IR in the Helix Compiler.

SSA – Single Static Assignment

Contents

Acknowledgements	2
Abstract	3
Glossary	4
1 Introduction	7
1.1 Goals	7
1.2 Naming	7
2 Research.....	8
2.1 Basic Compiler Structure	8
2.1.1 IR	8
2.1.2 The backend/code generation.....	8
2.2 Existing Compilers.....	9
2.2.1 Clang/LLVM.....	9
2.2.2 GCC.....	9
2.2.1 Conclusion	9
2.3 Choosing a suitable language frontend.	10
2.3.1 Clang.....	10
2.3.2 GCC.....	10
2.3.3 .NET IL/JVM Bytecode	10
2.3.4 Custom Frontend	10
2.3.5 Conclusion	11
2.4 Optimisation Techniques	11
2.4.1 Peephole Optimisations	11
2.4.2 Linear Scan Register Allocation.....	11
2.4.3 Simple Constant Propagation	12
3 Development.....	14
3.1 Design - Intermediate Representation.....	14
3.1.1 Modules.....	14
3.1.2 Functions	14
3.1.3 Basic Blocks.....	14
3.1.4 Instructions	14
3.1.5 Values	15
3.1.6 Types	15
3.2 Implementing a C language frontend with Clang	16

3.2.1 Building LLVM & Clang Libraries	16
3.2.2 LibClang vs LibTooling	16
3.2.3 Writing the frontend	17
3.3 Testing Methodology	18
3.3.1 Performance Benchmarks	18
3.3.2 Integration Tests	19
3.3.3 Unit Tests.....	20
3.3.4 External Testsuites	20
3.3.5 Testing Infrastructure & Tools	20
3.4 Register Allocation	21
3.4.1 Spill Only Register Allocation	21
3.4.2 Linear Scan Register Allocation.....	21
3.5 Optimising String Length Benchmark	23
3.5.1 Introduction	23
3.5.2 Analysis & Optimisations.....	24
3.5.3 Peephole Optimisations	25
3.5.4 Optimising (out) load and stores.	26
3.5.5 Conclusion	27
3.6 Optimising “Addition” Benchmark (Constant Propagation)	28
3.6.1 Introduction	28
3.6.2 Further Work	31
3.6.3 Conclusion	32
3.7 Final Performance & Testing Results	33
3.7.1 Performance Results	33
3.7.2 Testsuite Results	34
4 Evaluation	35
4.1 Retrospective on the structure and design of the compiler	35
4.1.1 IR	35
4.2 Testing and Correctness	36
4.3 Limitations and Future Developments.....	36
4.4 Conclusion	37
4.4.1 Personal Development	37
References	38
Appendix 1 – High Level Intermediate Language (HLIR)	40
Appendix 2 – String Length Disassembly	41
Appendix A – Project Specification	42

Module Deadlines.....	44
Appendix B – Ethics Form	45

1 Introduction

In the early days of computing, programmers had no choice but to write their software in assembly language (sometimes, as machine code) – a necessity due to the lack of alternative methods or motivated by the need to maximally exploit the potential of the frequently limited hardware.

As computer architectures became more complex and the scale of software systems grew, writing assembly programs by hand became increasingly difficult. Instead, it was much easier to write software in a higher-level language and then employ a compiler to it to the native machine code for a particular hardware platform.

These compilers began as simple tools that would naively generate code, often line by line (since the compilers, being software themselves, were restrained by the limited hardware). Over time these compilers became increasingly complex, capable of producing highly optimised binaries - to the point where compilers can now take even poorly written high-level code and convert it into machine code that executes efficiently on a variety of hardware.

1.1 Goals

This report will investigate how these compilers are structured and some of the possible optimisation techniques used to increase runtime performance. The aim is to produce a simple compiler capable of demonstrating some of these methods.

1.2 Naming

During development the compiler was given the codenamed *Helix*. As a result, any reference to *Helix* (or *hxc*, the executable name) should be taken to refer to this projects compiler.

2 Research

2.1 Basic Compiler Structure

Modern compilers generally consist of three stages – a “frontend”, a “middle end”, and a “backend”:

- The frontend parses source code into an “intermediate representation” (IR) that is used for the later stages.
- The middle end is responsible for platform independent optimisations and analysis. For example, dead code elimination or constant propagation.
- The backend takes the IR and can apply platform specific optimisations. It then is responsible for generating target specific code.

Splitting compilers into these stages is useful for maximum reusability – allowing the use of different source language and target architectures. A compiler that supports multiple backends is known as a “retargetable” compiler.

2.1.1 IR

At the heart of any compiler is its IR, with most passes in the compiler reading or manipulating the IR in some form.

There are broadly three classes of IR:

- Graphical IRs – the program is represented as a graph, using nodes edges and trees.
- Linear IRs – the program is represented as linear sequences, much like assembly for some abstract machine.
- Hybrid IRs – combines graphical and linear IRs. A common hybrid IR format is to use a linear IR to represent linearly executing sequences of code and using a graph to represent the flow of control between these.

2.1.2 The backend/code generation

Code generation encompasses several different tasks, most importantly – Instruction Selection, Instruction Scheduling and Register Allocation.

Instruction Selection is the process of choosing which machine instructions to emit. Instruction Scheduling then decides on the order to put those instructions in. This is more important when generating code for pipelined CPUs and can provide significant performance benefits. Register allocation then has the process of assigning high level variables to registers.

2.2 Existing Compilers

This section serves as research into two existing compilers/compiler toolchains, GCC and LLVM. GCC and LLVM are two of the most popular compilers in use today for AOT compilation, especially for C like languages.

2.2.1 Clang/LLVM

LLVM is a collection of reusable compiler technologies and toolchains and not a compiler in of itself. Clang is a C/C++ compiler utilising those libraries. Rust (a language focused on memory safety and concurrency) also uses LLVM (LLVM Project, 2022).

LLVM IR is a Single Static Assignment (SSA) based IR and is used throughout all phases of the compiler. Due to LLVMs wide variety of uses, its IR needs to be capable of cleanly representing high level constructs, as well as providing access to low level operations.

LLVM is natively a cross compiler, meaning the compiler itself doesn't have to be recompiled to target different architectures. This is appealing, although has its disadvantages – by relying on toolchains/libraries already being built and present for the target architecture.

2.2.2 GCC

GCC is a compiler developed by the GNU Project and supports many different languages (including C/C++) and machine architectures. GCC was originally released in 1987 by Richard Stallman, to enable bootstrapping the GNU operating system using entirely free software¹, and has been under almost continuous development since.

GCCs core has three different IRs – GENERIC, GIMPLE and RTL. Each language frontend produces GENERIC, which is designed to represent functions as trees in a language agnostic manner. The GENERIC is then converted to GIMPLE (by the “*gimplifier*”) which is a subset of GENERIC, specifically for use in optimization (Merrill, 2003). The final IR form used is RTL (Register Transfer Language) – this is very close to the final assembly and was the original IR the compiler used (GENERIC and GIMPLE were added later).

Unlike LLVM, SSA form is not always enforced for the IR – GIMPLE does have a SSA form used for some optimisations (called SSA GIMPLE), but RTL, GENERIC and Low/High forms of GIMPLE are not in SSA form (GCC Wiki, 2008).

Also, unlike LLVM, GCC does need to be recompiled to change the target architecture. This can present a problem, since GCC is not always trivial to compile. Because each target architecture gets its own set of binaries, the GCC package includes the required toolchains/libraries.

2.2.1 Conclusion

A large influence on the designs of both LLVM and GCC is that they are retargetable compilers. For this project it is known that the target architecture will be ARM, so many of those design decisions do not apply here.

Whilst SSA IRs do seem to be the common form, it does bring with it additional complexity in terms of maintaining the SSA structure regardless of modifications. Transforming code into and out of SSA form efficiently is also a non-trivial problem (Cytron, Ferrante, Barry, Wegman, & Zadeck, 1991).

¹ Freedom defined by the Free Software Foundation as users having the “*freedom to run, edit, contribute to, and share the software*” (Free Software Foundation, 2022) and does not refer to price.

2.3 Choosing a suitable language frontend.

All compilers that are of any real-world use require a frontend – the phase of compilation that transforms textual source code into an internal format that is easier to understand and manipulate during the later phases of compilation.

This section evaluates the choice of 4 possible frontends.

2.3.1 Clang

Clang is C/C++ compiler (written in C++) and a part of the LLVM project. Apple originally created Clang to serve as a more permissively licensed replacement to GCC. Clang is designed to have a modular library-based architecture, making it a common choice for companies who want a custom compiler without building one from the ground up, for example the Intel C Compiler is now based off Clang & LLVM (Intel Corporation, 2021). Clang also natively supports Windows and can be built using Visual Studio.

2.3.2 GCC

GCC has a monolithic structure (by design²) and because of its long lifespan it's codebase is complex and can be hard to work in. For example, GCC was originally written in C, but moved to C++ in 2012 (GCC, 2012). This has resulted in an odd mix of legacy C and modern C++ in the codebase. Unlike Clang, GCC development & usage is focused on Unix style platforms – it is possible to build GCC on Windows but requires a Unix-like environment such as Cygwin (GCC, 2022).

2.3.3 .NET IL/JVM Bytecode

.NET IL (also known as CIL – Common Intermediate Language) is the abstract instruction set, executed by the .NET runtime. The advantage to targeting this platform is that it would allow writing code in any language that compiles to .NET IL (such as C#, F#, Visual Basic etc...). These compilers are all very mature and sit completely detached from the backend. The .NET IL instruction set is also significantly simpler than an Abstract Syntax Tree that would be directly produced by language frontends. However, .NET IL is designed to execute in a JIT (Just in Time) execution environment and not be compiled AOT (Ahead of Time). For example, runtime code introspection (reflection) is difficult to implement in an AOT compiler.

JVM (Java Virtual Machine) bytecode is very similar to .NET IL but exists within the Java ecosystem. Using this as a frontend shares the same advantages and disadvantages as targeting .NET IL.

2.3.4 Custom Frontend

Instead of utilising existing frontends, another option is to write a new frontend from scratch, either for a new language or an existing language. Writing a new frontend would eliminate the mental overhead of having to integrate and understand an existing implementation. Writing a new language could also result in a simple language that only includes features supported by the compiler, instead of having a frontend for a full language but the backend only supports a subset of those features.

² Richard Stallman (a leading figure in the Free Software movement & the original developer of GCC) has resisted attempts to make GCC more modular, since it “would make it easy to extend GCC with proprietary programs” (Stallman, 2014) – transforming GCC from a free compiler to a platform for non-free compilers.

However, writing a language frontend would not be trivial, and whilst there are tools to simplify some of the process (such as parser generators, like ANTLR or Yacc/Bison) it would still be a considerable time investment.

2.3.5 Conclusion

After evaluating the different options, the choice for this project will be to use the Clang C frontend. The C language has the advantage not having many high-level features, as well as being a language that is almost exclusively AOT compiled. With Clang being modular and library based, it will be easier to integrate compared to GCC.

Clang is also written completely in C++, which matches the choice of language for this project. On the other hand, tooling for .NET IL and JVM bytecode is almost entirely based in their native languages (C# and Java), which would either force the rest of the compiler to be written in that language (undesirable) or require complex interop code.

Writing a custom frontend was disregarded early on, due to the significant time it would take to develop. The focus of this project is on the backend and optimisations and spending too much time on the frontend has the potential to detract from that.

2.4 Optimisation Techniques

There are many optimisation techniques, and as a result only a small number can be covered in this project.

This section provides some examples of these techniques, and for two (Linear Register Allocation and Simple Constant propagation), attempts to give an overview of the algorithms.

2.4.1 Peephole Optimisations

Peephole optimisation is not a specific technique, but a way of grouping various optimisations. Peephole optimisations are all done on small windows of code, be that physical windows or logical windows (Torczon & Cooper, 2012). A physical window is looking at code that is physically closer together, whereas a logical window is looking at code connected by the flow of values between them instead of proximity. Peephole optimisations are, in of themselves, not complicated so can be explained in further depth later in the report.

2.4.2 Linear Scan Register Allocation³

Linear scan is a global⁴ register allocation approach, designed to be simple and quick whilst at the same time producing relatively good code. This is incredibly useful for dynamic systems, such as Just in Time (JIT) compilers or interactive environments, for example V8 and the Hot Spot Java Compiler both use Linear Scan - or slight variants thereof (Eisl, Grimmer, Simon, Würthinger, & Mössenböck, 2016).

Linear Scan works by computing the live intervals for each virtual register in the program. Live intervals are a very coarse way of looking at variable lifetimes –roughly the duration of the program between the variable first being used and last being used.

³ In this context, this register allocation technique is considered an optimisation, since it is possible to do register allocation in an incredibly inefficient manner, and this is an improvement upon that.

⁴ “Global” in this context means to operate on the function instead of “local” – which is to operate on individual basic blocks or “intraprocedural” which is to operate across different functions.

Live intervals can be constructed by doing liveness analysis on the function – liveness analysis (also known as live variable analysis) computes a list of variables that are live on entry a block and variables that are live upon leaving a block. Algorithms for doing live variable analysis are well documented (Compilers: Principles, Techniques, & Tools, 2007).

Given this live variable information, the starting and ending positions for each variable can be determined by iterating over the blocks and variables in a function applying the following rough heuristics:

- Is a variable live exiting this block, but not live on entry? If so, then start its live interval here.
- Is a variable live on entry to this block but not live on exit? If so, then we can terminate this variable's live interval here.
- Is a variable only used within this block (e.g., not live on either entry or exit)? If so, can compute it's interval by linearly iterating over this block and recording the first use and last use.

Then given a list of live intervals, the linear register allocation algorithm works by iterating over those intervals and trying to assign a free register to each. If there are no free registers the variable is spilled to the stack. If an interval has expired (e.g., the variable is no longer in use) then the register occupied by that interval is "freed" - e.g., make available again for use by other intervals.

This is only a summary overview of the algorithm, and a full explanation is out of the scope for this report. For a full explanation consult the original paper (SARKAR & POLETTI, 1999).

2.4.3 Simple Constant Propagation

There are 4 popular constant propagation algorithms – *Simple Constant* (Kildall, 1973), *Conditional Constant*, *Sparse Simple Constant* and *Sparse Conditional Constant* (Wegman & Zadeck, Constant Propagation with Conditional Branches, 1991). LLVM uses a variation of Sparse Conditional Constant propagation.

This will primarily consider the *Simple Constant* algorithm since *Sparse Simple Constant* and *Sparse Conditional Constant* both require the input IR to be in SSA form, and it has already been established that SSA will not be used for this project.

Conditional Constant is very similar to *Simple Constant* but takes into considering conditional expressions and the effect of constants on them. As a result, it can discover more constants and do unreachable code elimination at the same time.

Kildall's *Simple Constant* algorithm works giving each node in the program (a node being an individual instruction in this case), two sets – one set that contains the state of every variable in the program upon entry to the node and one set containing the state of every variable upon exit of the node

The algorithm is rooted in lattice theory –the variables stored at each node are lattice cells, and the lattice cell upon entry to a node is the *meet* of the values at the previous nodes exit.

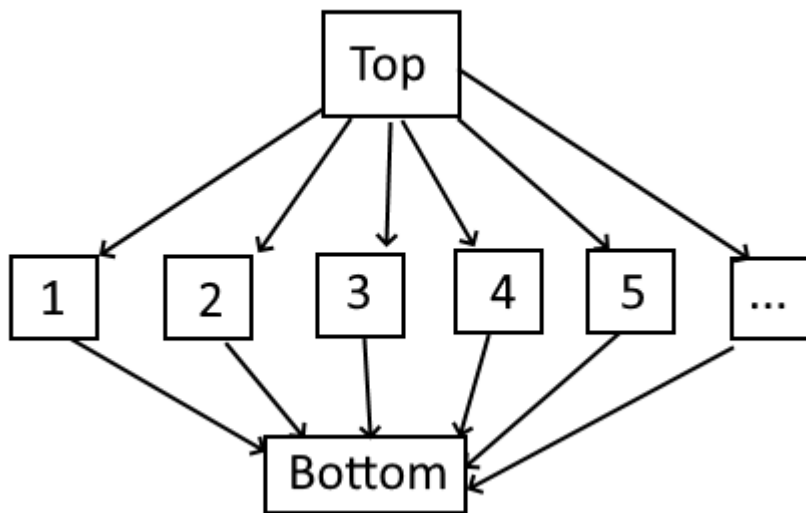


Figure 2.1 Lattice representing each value in the program.

Figure 2.1 shows the 3 different types that a variable can be at any point in the program as a lattice. *TOP* means that the at some point the variable may be a (yet undetermined) constant. *BOTTOM* means that the variable is not a constant. If a value is neither *TOP* nor *BOTTOM*, then it is a constant (the middle values in the graph).

The algorithm starts by setting each variable to *TOP* (meaning that it could be a constant), except for the start node where each variable is assigned to *BOTTOM*. Then as the program processes each node, the new type of the variable is determined by applying the following rules to the variables leaving the previous node, and the variables entering the current node

$$\begin{aligned}
 any \cap TOP &= any \\
 any \cap BOTTOM &= BOTTOM \\
 \langle constant \rangle_i \cap \langle constant \rangle_j &= \langle constant \rangle_i \text{ (if } i == j) \\
 \langle constant \rangle_i \cap \langle constant \rangle_j &= BOTTOM \text{ (if } i \neq j)
 \end{aligned}$$

Figure 2.2 Rules for the meet operator.

This serves to lower values from the top of the graph to the bottom. The type of a value never moves upwards. This works iteratively until a whole pass has been done over the program where nothing is changed (then the algorithm exits).

The theory and backing behind this algorithm can be quite complicated, so this only attempts to serve as a brief overview. For a full explanation consult the original paper, *A unified approach to global program optimization* (Gary A. Kildall, 1973).

3 Development

3.1 Design - Intermediate Representation

Central to the compiler is its intermediate representation (IR) format. The final IR is a hybrid between LLVM IR and the ILOC IR described in *Engineering a Compiler, Second Edition* (Torczon & Cooper, 2012). Unlike LLVM, this IR is not in Single Static Assignment (SSA) format – this is by design, with the intention to avoid the complexity that SSA brings.

Over the course of the project, two forms of the IR developed High Level IR (HLIR) and Low Level IR (LLIR). The original design was just for HLIR – LLIR was introduced to allow for register allocation after high level constructs had been converted to their ARM representation. LLIR is identical in format to HLIR, the only difference being that the opcodes no longer refer to abstract instructions, but real instructions.

The IR is hierarchical in nature, and this section attempts to give an overview of each level.

3.1.1 Modules

A program in the IR is known as a “module”, and a module is composed of one or more functions. A module contains any number of global variables declared in the program and keeps track of all user-defined structure types in the program.

3.1.2 Functions

Functions follow the traditional form – they specify a return type as well as any number of parameters that the function accepts. Variadic argument lists are not supported by this compiler. Each function is composed of one or more basic blocks.

3.1.3 Basic Blocks

A basic block is a sequence of linearly executing instructions – they have one entry point (which is into the start of the block) and one exit point (out of the end of the block). This means that only the last instruction in a block is allowed to be a branching instruction, and this is known as the “terminator” instruction. A basic block is ill formed if the last instruction is not a terminator.

Note that in this context, a call is not considered a branching instruction, since once the called function has finished executing control flow is returned to the calling point.

3.1.4 Instructions

Instructions follow a three-address-code format, meaning that (generally) each instruction has at most 3 operand values as well as an operator (also known as an opcode). There are two exceptions here: function calls, which are allowed to have more than three operands and LLIR instructions that represent ARM instructions with more than three operands. In practice however, no ARM instructions like this have been encountered.

An overview of the HLIR instruction set can be found in Appendix 1.

3.1.5 Values

Values are the backbone of the IR and represent the objects/entities in the program. Every value has a type (see section below) and a list of users. The use list is a way to keep track of which instructions refer to the given value in the program. This is useful for analysing and transforming the IR (e.g., replacing all uses of a given value with another). The most important value types are given in table 3.1, below

Value	Description
Virtual Register	An abstract register. There is no fixed limit on the number of virtual registers in a program, unlike physical registers.
Physical Register	A physical hardware register, number is constrained by the target architecture. Virtual Registers are converted to Physical Registers by the register allocation pass.
Constant Integer	Represents an immediate integer constant, e.g., as an argument to an instruction.
Undefined	Represents an unspecified value, e.g., the state of declaring a variable in C but not immediately assigning it a value.

Table 3.1 Different types of value

There are more value types in the compiler, but they exist either as an implementation detail (e.g., functions and basic blocks are technically called as values) or to make frontend IR generation easier (e.g., value types for C array and struct literals).

3.1.6 Types

These represent the type of each value. There are 4 built-in integer types - for 8, 16, 32 and 64 bit integers. Note that 64-bit integers & arithmetic are not actually supported in the code generator, so are only in the IR for completeness.

Pointers (addresses) are also their own data type, but unlike C, pointers are not typed. This means that the C concepts of *int** or *float** don't have direct representations in the IR, instead there are just pointers to data (called opaque pointers). This is inspired by the project in LLVM to move from their typed pointers to a new opaque pointer type (LLVM Project, 2022). The advantage to opaque pointers is that they can reduce complexity later in the pipeline and can avoid unnecessary casts. The disadvantage is that the language frontend instead needs to keep track of the type of data pointed to by each pointer, but since this project uses Clang this work has already been done.

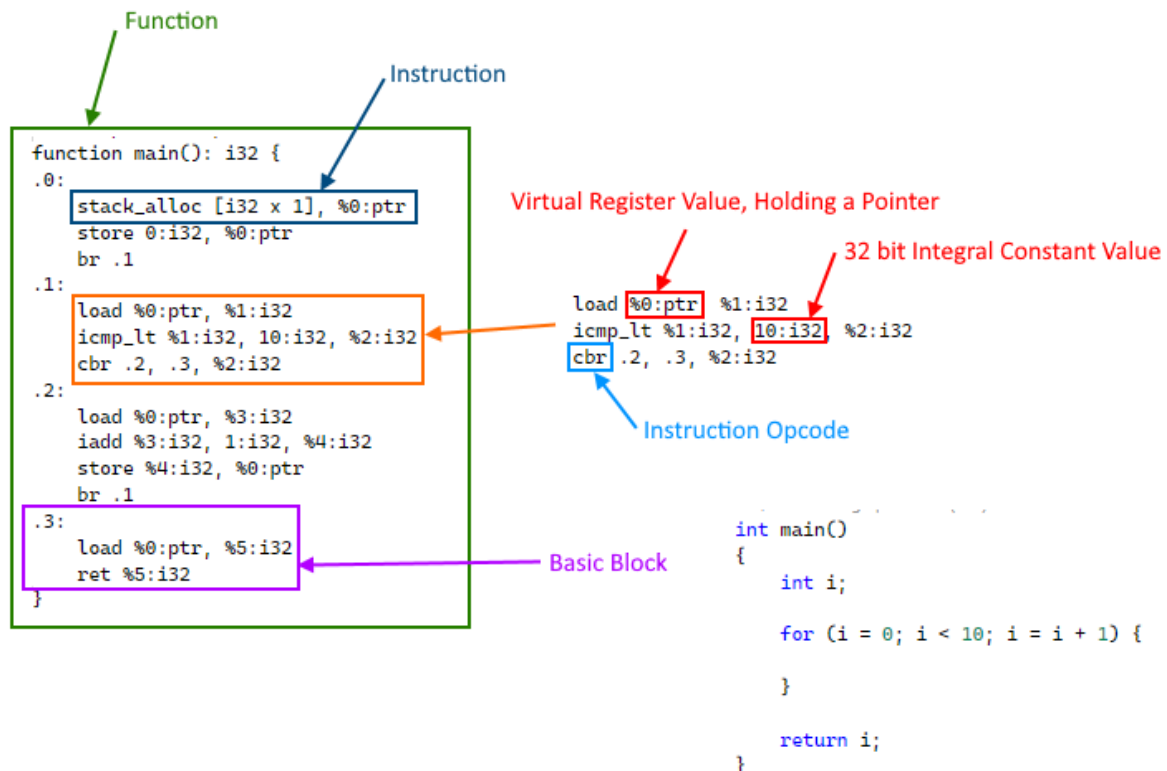


Figure 3.1 IR breakdown

Figure 3.1 gives a breakdown of a C “for loop” and its HLLR representation.

3.2 Implementing a C language frontend with Clang

3.2.1 Building LLVM & Clang Libraries

Building LLVM/Clang turned out to be a trivial enough process – both this project and LLVM use CMake⁵. The primary irritations were down to LLVM’s size, both in the sense it took a long time to compile (up to an hour on a good PC), and the binaries were large on disk.

The long compile times could be mitigated, however, by precompiling the binaries and sticking to the same version of LLVM. This did cause one issue during development – a newer version of the LLVM headers were fetched, but the libraries not recompiled, leading to some confusing link errors.

In debug mode the libraries came to 2.5GB on disk and the release binaries came to 330MB. This is only a minor irritation however and didn’t cause any problems.

3.2.2 LibClang vs LibTooling

Clang has two support libraries designed to make the process of using the Clang C/C++ frontend easier – LibClang and LibTooling.

LibClang is a stable C wrapper over the Clang libraries. LibClang focuses more on high level abstractions (e.g., walking over the AST with a cursor or code completion) and is written to be simpler, with backwards compatibility in mind.

⁵ CMake is a system for cross platform build configuration. Scripts are written in a custom language to define the project, and CMake generates build files for Visual Studio, Make etc...

LibTooling sits much closer to the core Clang AST and as a result, is not stable (if the Clang AST changes, code written with LibTooling that interfaces with that AST will also need to be rewritten). It does also not have the same high level abstractions as LibClang.

Considering these factors, a better choice for this project is LibTooling – providing a full control over the AST will make IR generation easier.

It's worth noting that whilst the initial “setup” documentation is good for both these libraries, it is lacking when attempting more complex operations – resulting in having to dig around the Clang source code & headers themselves to figure out what's what. This is something that became more of a problem later in the development of the frontend.

3.2.3 Writing the frontend

Writing the frontend itself mostly went completely smoothly and was the focus of the first month of development (after that it didn't really need to be touched again). It became easier as time went on, owing to the recursive and reusable nature of the code (since language constructs often build on each other). The best example of this are expressions – expressions are a fundamental building block for C and are used everywhere e.g., as the right-hand side of assignments, the conditions of if/loop constructs, as function arguments et cetera. This then meant that all that was needed was to add support for a new expression in one place, and then all the places that used expressions would have that feature available to them. The fact that C was designed as a low-level language helps in making the frontend simple.

Expressions: lvalue and rvalue semantics.

One major problem that did arise during development of the frontend related to how expressions are handled – specifically the different types of value that C has. C specifies two types of value, *lvalues* and *rvalues*. An *lvalue* is “an expression (other than **void**) that potentially designates an object” (ISO/IEC 9899, 2018), whereas an *rvalue* denotes all other expressions – or more simply *lvalues* can appear on the left hand side of an assignment (hence the name) and have memory addresses.

```
int main()
{
    int a;
    lvalue  rvalue
    a = 10 + 10;
}
```

The distinction is significant, since whilst they are both expressions, there are places where it is only valid for one to appear. This presented a problem since originally the frontend was written with the idea that there was only one type of expression - *rvalues* (assignment was coded as a special case of evaluating the right-hand side and storing it into the variable on the left).

Figure 3.2 shows a simple example.

Figure 3.2 lvalues & rvalues

This worked fine until it came to implementing support for pointers (e.g., the address of operator “&” and the dereference operator “*”), here the distinction between *lvalues* and *rvalues* becomes important since you can only take the address of a *lvalue*. This also meant that coding assignment as a special case that assumes a variable is on the left is no longer valid - “*a = 20;” is a valid assignment but the left-hand side is not a variable, instead is an *lvalue* expression.

Refitting the frontend to understand the difference between *lvalues* and *rvalues* was a significant amount of work, and in hindsight could have been avoided if a bit more planning went into the

frontend before starting (instead of starting with the intention of getting it done as quickly as possible to get onto the more interesting code generation).

3.3 Testing Methodology

Testing was a continuous and critical part of the development process – fundamentally a compiler is useless if it cannot be relied upon to generate functional code. There were 3 different types of testing used to validate correctness: integration tests, unit tests, external testsuites. There are also performance benchmarks used to compare runtime performance against other compilers.

3.3.1 Performance Benchmarks

Once the compiler had reached a stage where it could produce executable binaries, it was possible to write benchmarks which compared the runtime performance of code emitted by this compiler, to that of other compilers. The chosen compilers are TCC (0.9.27), Clang (11.0.1) and GCC (10.2.1). Clang and GCC exist to provide a reference for what a production optimising compiler can do. TCC on the other hand should provide results more like ours. Whilst it does have some optimisations (Bellard, 2018), such as constant folding or promoting multiplications & divisions to shifts, it instead is designed to be small with very fast compilation times. All benchmarks are compiled at -O3 (maximum optimisations) for GCC and Clang. TCC and Helix do not offer any optimisation flags (all on by default).

Benchmarks were written using Google Benchmark – a microbenchmarking support library. The same code was compiled using each compiler producing 4 object files, and then linked together with a harness program which was used Google Benchmark to run the compiled functions produced by each compiler. This process turned out to be troublesome given that Helix is a cross compiler and as a result the tooling is awkward and error prone. Better planning could have mitigated these problems.

Addition (“add”)

Adds 3 numbers together, designed to be one of the simplest possible benchmarks. Tests the level of constant propagation done by each compiler.

Binary Search (“binary_search”)

Performs a binary search on a sorted list of 17 integers.

Fibonacci (“fib”)

Calculates the 13th Fibonacci number.

Sieve of Eratosthenes (“sieve”)

A simple implementation of the Sieve of Eratosthenes algorithm that calculates the first 25 prime numbers. This is a relatively computationally expensive algorithm, so is commonly used as a compiler benchmark.

String Length (“strlen”)

A naïve implementation of C’s “strlen” function – calculates the length of a string by scanning for a null terminator.

Byte Sieve ("byte_sieve")

Another implementation of the Sieve of Eratosthenes, but this one was originally published in 1981 as a programming language benchmark in the BYTE magazine (Gilbreath, 1981). This implementation and variations of have been used widely since, and whilst aging is still complex enough to be used for this project.

3.3.2 Integration Tests

The integration tests were the primary method used for testing correctness. Each integration test was comprised of two components: an XML test "definition" and an input C source file. The basic process was as follows:

1. The provided C source file would be run through the compiler (the test definition file could specify any command line arguments to pass)
2. If specified by the test definition, the stdout of the compiler would be compared to an expected value (set in the test definition).
3. If the compilation succeeded & produced an executable file, the executable would then be run (via QEMU userspace ARM emulation).
4. The exit value of the executable would then be compared to an expected value set in the test definition.

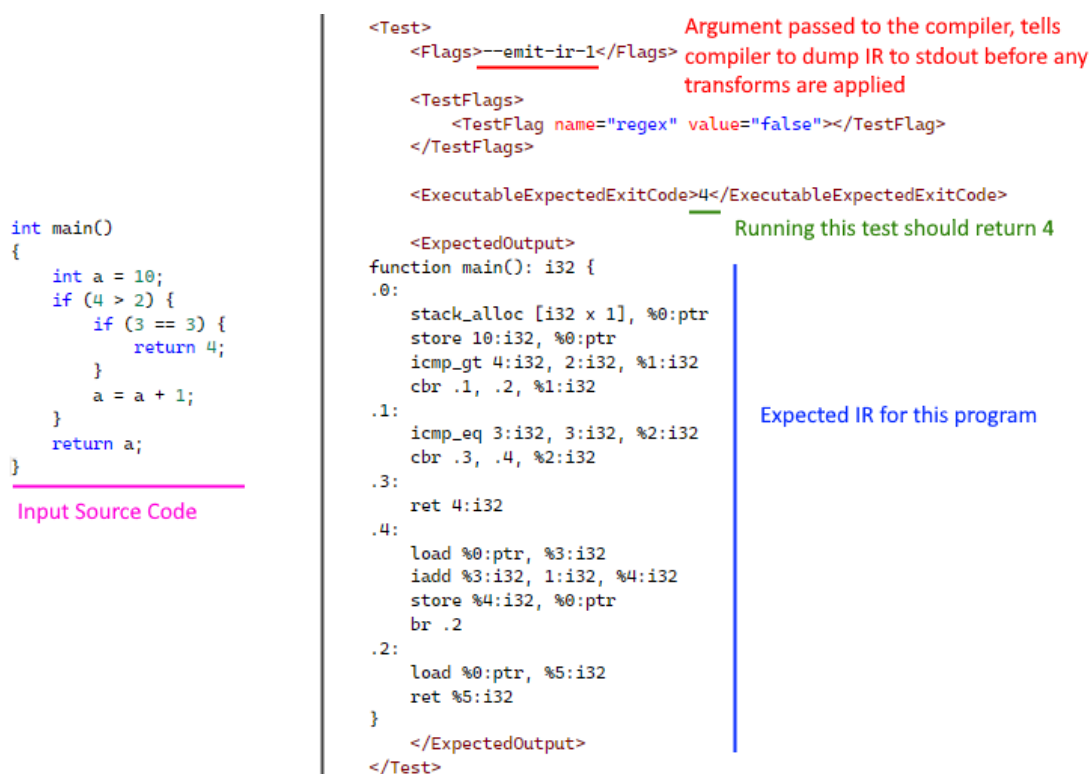


Figure 3.3 An example of an integration test, input C left, test definition right

Something that is worth noting, is that this style of integration test was born at the very start of the project when the compiler frontend was in development. At this point in time, the compiler didn't produce executable binaries, so the only thing to test was the correctness of the IR coming out the frontend. This explains the odd "testing compiler stdout" aspect of the tests.

As the compiler grew and more optimisations/IR transforms were added it became more and more difficult to test IR correctness by just comparing it 1:1 to an expected output (for example register allocation output can be changed massively by tiny changes in the IR, but the program can still be correct). Originally to counter this, a “regex” text comparison option was added, with the idea that regular expressions could be used to factor out commonly changing aspects of the IR and just compare the bits that matter. However, this did not turn out to be useful at all and was disabled for all tests. As a result, integration tests added later during the compilers development tended to rely more on running the compiled binary and checking exit codes.

3.3.3 Unit Tests

There is some unit test coverage for various parts of the compiler, but unit tests were generally only used to test internal “infrastructures” – data structures, boilerplate/glue code and some aspects that were harder to test with the integration tests (e.g., the linear register allocator had some unit tests, since this was harder to test externally via the integration tests).

In the end, there were more unit tests than integration tests, however in practice the integration tests proved to be more useful in catching real world bugs. The code tested by the unit tests didn’t tend to change that much, whereas the integration tests tested the overall end to end compilation process.

Unit testing was done using the Catch framework.

3.3.4 External Testsuites

In comparison to the other forms of testing mentioned, external test suites didn’t play a massive part in ensuring the correctness of the compiler and were not used regularly – if anything, just used as a comparison to other C compilers.

The problem with using an external test suite is that they are designed for full C compilers and are often aimed at minute and obscure parts of the language. As a result, they tended to throw up a lot of errors due to unsupported features, instead of discovering actual bugs.

The external test suite used for this project was c-testsuite (Chambers, 2020). Finding a good external test suite is difficult, on account of them either being too big or proprietary. c-testsuite was a good middle ground as it is permissively licensed (under the MIT license), not too big but exercises enough functionality to be a good judge of if the compiler works.

3.3.5 Testing Infrastructure & Tools

To facilitate the various forms of testing, and to provide great flexibility a custom test runner tool was written – known as *Testify*. This tool was written in C# and was the primary way to run the integration tests and external test suite.

In the very beginning the integration tests were run by a combination of batch and small python scripts, however this became difficult to work with as the test suite grew and more reporting functionality was required. To counter this, the *Testify* tool was written. Initially all this tool did was run the compiler on each test and compare the stdout of the compiler against the expected text. Over time greater reporting functionality was added, as well as the ability to run the compiled tests.

The implementation of this tool is functional and uninteresting – simply serving as an implementation detail in the scope of the compiler. As a result, it will not be covered in depth here.

3.4 Register Allocation

Register allocation is a critical part of any code generator – the IR consistent of an infinite set of abstract “virtual registers”, but this is not the reality of the target hardware. Instead, the hardware consists of a finite set of registers and so register allocation is required to map the virtual registers to a physical register or (if all registers are in use) spilling the variable to the stack.

3.4.1 Spill Only Register Allocation

The first register allocation scheme implemented in the compiler was a “spill only” register allocator. The basic idea is that every virtual register in a function is allocated to its own stack slot and before any value can be read in an instruction it has to be loaded from the stack and any registers written to during the instruction must be saved back to the stack again.

Consider the example in Figure 1, the IR adds together two virtual registers %1 and %2 and stores the result in %3. The spill only register allocator would then allocate a stack slot to each virtual register, and must “restore” (i.e., load from the stack) %1 and %2 into registers *r0* and *r1* before they can be used in the addition, then storing the result of the addition (*r2*) into the stack slot for %3.

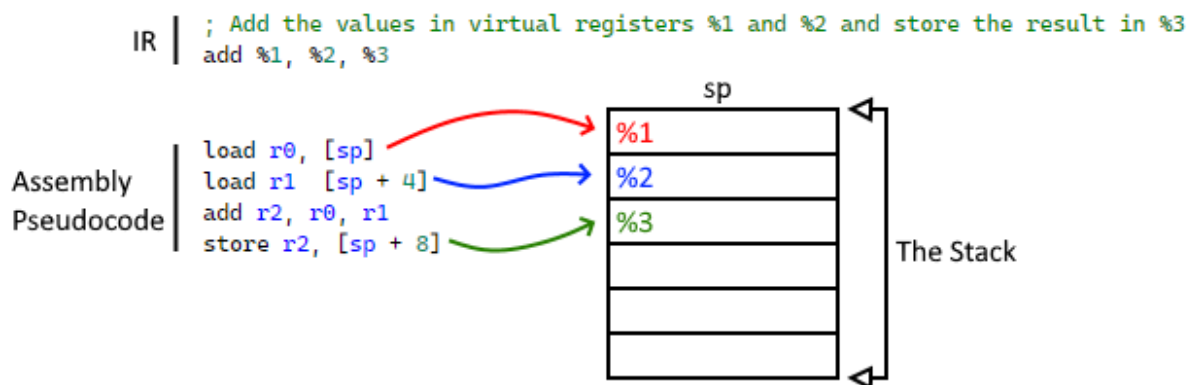


Figure 3.4 Spill Only Register Allocation

The allocator makes no assumptions or does any analysis on the lifetimes of the virtual registers and treats each instruction in isolation. This leads to poor generated assembly, which is dominated by often redundant loads and stores.

Despite all its flaws it did work and was quick to implement. This meant time could instead be spent on getting the rest of the code generation systems to work.

3.4.2 Linear Scan Register Allocation

The next attempt at register allocation was implementing the “linear scan” register allocation algorithm.

For this project, Linear Scan is attractive due to its simplicity (or apparent simplicity) – full blown graph colouring-based register allocation can be incredibly complicated and could be a project in of itself.

Whilst Linear Scan is not an overly complex algorithm, the compiler had grown to the point where it was awkward to quickly prototype in – C++ is infamous for its slow compilation. For these reasons, a prototype was written in Python to get a feel for how the algorithm is implemented. This proved to be immensely useful and massively sped up the initial development phase. Once this had been done

implementing the algorithm in the compiler proved to be trivial, as a design for how it should work had already been established.

The linear scan register allocator was implemented as a new pass *regalloc2* (instead of *regalloc* which was the spill only allocator). In contrast to the spill only allocator, this pass ran after the transformation from HIR to LLIR, which enabled the HIR to LLIR conversion pass to create new virtual registers for whatever purpose they were needed. This contrasts with the previous solution to this, which was to have the register allocator reserve a set of registers as scratch registers, meaning they could not be used for register allocation.

Live variable analysis was run as the first step in the register allocation pass – a further improvement would be to instead run live variable analysis automatically after any pass that modifies the IR enabling the analysis data to elsewhere in the compiler.

The paper that originally published this algorithm states that the process of transforming live variable information into live intervals can be done “*easily with one pass through the intermediate representation*”. This didn’t turn out to be the case for this implementation, and a lot of time was spent trying to get this process right. It would probably be worth revisiting this in the future, to try and improve the implementation.

3.5 Optimising String Length Benchmark

3.5.1 Introduction

This follows the task of analysing the `strlen` benchmark, and then applying targeted optimisations to improve performance.

The C implementation for this benchmark is given in Figure 3.5 (note: despite this being a common implementation, it is not the most efficient).

```
int benchmark_main() {
    char* c = "dKxLIImNLlMSKucIlaprXsQUpSWdyIZwtDCNSoICZDrCcQsFXNwTXemQobCGnWk"
              "mIYgEUrqgUZoOdiHauENbyAPuvOyeZKRTvTbNOYFbSqUwoerMuOwCDwFMjZCwBm"
              "uBnuQkKmDnqOAhjizxyBVwgasyWAMSiXSqQBjwdwRoSfGFhUhIXDVBZZnvprXlJ"
              "eVMTLBHXMESDRPbDoCHaqByuZLFszUciRzOwzsswyDhSTHVRtnzvBevTXNreKJCaip";

    int l = 0;
    while (c[l] != '\0')
        l++;

    return l;
}
```

Figure 3.5 'strlen' implementation in C

Compiler configuration for benchmarks is described in section 3.3.1 *Performance Benchmarks*.

The benchmarking results are given in Table 3.2 below.

Compiler	Time ⁶	CPU ⁷	Iterations ⁸
<i>Helix</i>	<i>1751ns</i>	<i>1750ns</i>	<i>399783</i>
TCC	1435ns	1434ns	487950
Clang	4.00ns	4.00ns	174896358
GCC	358ns	358ns	1957812

Table 3.2 Benchmarks before optimisation

These results are mostly as expected – this projects compiler (Helix) is the slowest, followed by TCC. What is slightly unexpected is the large disparity between Clang and GCC (I would have expected them to be very similar).

After some investigation it is evident that Clang can recognise the string length algorithm at compile time and can thus optimise it out - moving the known string length (in this case, 255) immediately into the return value register and returning (Figure 3.6). GCC instead does the expected and generates an - admittedly very optimised – loop⁹ (Figure 3.7).

⁶ Wall clock time per iteration

⁷ Average CPU time per iteration. This may slightly differ from the wall clock time, e.g., when the process sleeps it will no longer accumulate wall clock time. These benchmarks should be very similar to wall time but are included for reference.

⁸ Google Benchmark tries to run all benchmarks for a similar same amount of time, this is the number of times this specific benchmark was run. A slower benchmark will thus run for fewer iterations than a faster benchmark.

⁹ The latest development version of GCC now implements this optimisation and indeed removes the loop, but this is yet to be released.

```
benchmark_main:
    mov     r0, #255
    bx      lr
```

Figure 3.6 Clang 'strlen' generated code

```
benchmark_main:
    movw    r3, #::lower16::LC0
    movt    r3, #::upper16::LC0
    movs    r0, #0

.L2:
    ldrb     r2, [r3, #1]!    @ zero_extendqisi2
    adds    r0, r0, #1
    cmp     r2, #0
    bne     .L2
    bx      lr
```

Figure 3.7 GCC 'strlen' generated code

3.5.2 Analysis & Optimisations

For context, the generated code consisted of essentially 4 sections (basic blocks) – the scalar code executed before the loop to setup the string and counter variable, the loop “head” that evaluates the condition and whether to run another loop or break, the loop body that increments the counter and the final exit block that returns the counter value (a full readout of the assembly is not worth covering in detail, but is included in Appendix 2 for reference).

		00010c28 <.bb1>:
		.bb1():
17.42	10c28:	ldr r6, [r4]
0.10	10c2c:	ldr r0, [sp, #4]
18.68	10c30:	ldr r7, [r0]
	10c34:	mov r5, r6
	10c38:	movw r6, #1
	10c3c:	movt r6, #0
	10c40:	mul r8, r7, r6
	10c44:	add r7, r5, r8
	10c48:	mov r6, r7
62.84	10c4c:	ldr r5, [r6]
	10c50:	movw r7, #0
	10c54:	movt r7, #0
	10c58:	cmp r5, r7
	10c5c:	mov r7, #0
	10c60:	movwne r7, #1
	10c64:	cmp r7, #1
	10c68:	↓ bge 10c70 <.bb2>
0.96	10c6c:	→ b 10c90 <.bb3>

Figure 3.8 Profiling Data for the "head" block

Profiling revealed that, as expected, the loop head (identified as *.bb1*) was the slowest section, followed by the loop body (identified as *.bb2*).

This might seem odd, but here all the body does is increment a value, whereas the head section must access memory to get the character, compare it to the null terminator constant and then branch, which amounts to much more work.

Profiling the head block wasn't immediately useful (Figure 3.8). Except for the branch at the end of the block, the only significantly measured instructions are memory operations, which are expected to be slow.

There is some nuance here, however, due to the nature of *perf* being a sampling profiler. At regular intervals *perf* records where the instruction pointer is and uses that to construct the percentages given – with the idea being that if

more samples land in a particular place, then that is probably slower. This works well at a macro level, for larger programs (e.g., which functions/lines appear the most) but can fail when profiling small bodies of assembly (as we are here).

This is primarily down to two reasons:

- a) the timings recorded become very small, often single figure nanoseconds, and at that resolution accurate timings are generally difficult for a couple of reasons: profiling and timing has its own overheads, and at small resolutions it can be hard to differentiate between profiling overhead and code performance. Also, as code gets smaller, there are more, hard to predict, factors in play. For example, whether the right data/code is in cache memory can have a large impact on performance.
- b) out of order, pipelined, CPUs do not execute code in a linear fashion, and a delay observed on a particular instruction might not be directly related to that instruction but instead the result of having to wait for earlier pipelined instructions to retire.

It is assumed that this is cause of the load instruction (in Figure 3.8, at 10c4c) taking up most of the timing profile – and is most likely related to the previous multiplication, which is known to be a typically slow instruction.

3.5.3 Peephole Optimisations

It is worth looking at this multiplication (isolated in Figure 3.9) in more detail, since the astute reader might notice that it is redundant – it multiplies the left-hand side value by 1, but arithmetic identities tell us that $a * 1 = a$.

```
movw r6, #1
movw r6, #0
mul r8, r7, r6
```

Whilst this is odd, it occurs for a good reason – assembly doesn't understand typed arrays, instead everything is just a series of bytes. So, to access an element in an array, we must multiply the index by the size of the array type, and then add that to the array base address to get a pointer to the element.

Figure 3.9 Multiplication

In this case, we're dealing with a *char* array, the size of *char* being 1.

This could be mitigated by recognising if we're indexing into a char array in the frontend, and if so, omitting the multiplication.

But that adds a special case, something that is good to avoid if possible. Instead, we can solve this in a more generic way, by running a "peephole optimisation" pass over the IR, trying to detect small blocks (windows) of code that can be transformed to something similar. This is a common concept and isn't limited to arithmetic simplifications – another common example is that, under X86, it is faster to execute *xor <reg>, <reg>* instead of *mov <reg>, 0*. This transform doesn't require context about the larger body of code, and as such could be implemented as a peephole transform.

This multiplication optimisation provided significant performance benefits – roughly 22.9% decrease (Table 3.3). Original numbers are superimposed in red alongside the updated timings.

Compiler	Time	CPU	Iterations
<i>Helix</i>	1366ns (1751ns)	1366ns (1750ns)	512201 (399783)
TCC	1432ns	1432ns	488647
Clang	4.00ns	4.00ns	174892475
GCC	358ns	358ns	1957745

Table 2.3 Benchmark results after implementing multiplication transform

There are many more possible peephole optimisations (a file implementing peephole optimisations in GCC – *match.pd* – is ~7700 lines long). As a result, only a few are implemented for this compiler. A second transform is one that recognises a common pattern in code that compares values and jumps. Previously this would generate code that first compared the values and moved either 0 or 1 into a result register and another section that compared that result to 1 and jumped accordingly (Figure 3.10).

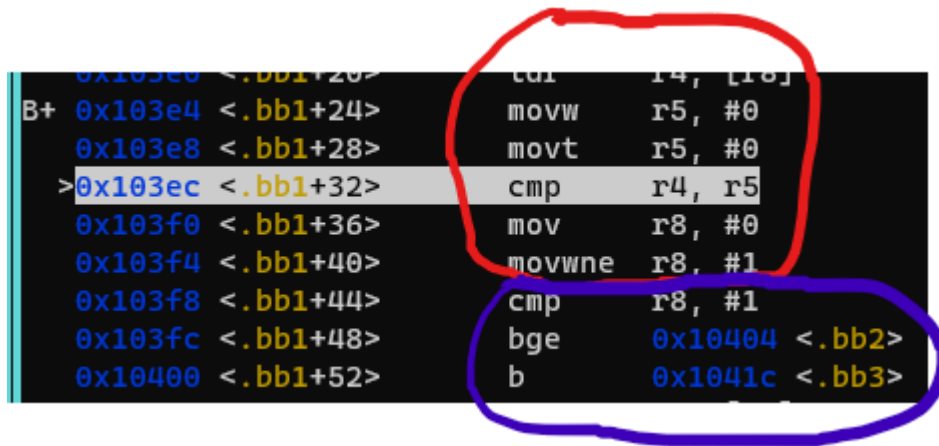


Figure 3.10 Inefficient Comparisons & Jumps

Instead of this, we can merge those two operations and use the fact that *cmp* sets the flags registers directly for the jump (generating code such as in Figure 3.11).

```

movw    r5, #0
movt    r5, #0
cmp     r4, r5
bne     64 <.bb2>
b       7c <.bb3>
  
```

For the *strlen* benchmark this transform didn't provide any noticeable performance increase. Now the primary bottleneck is indeed memory IO.

Figure 3.11 New Jump & Comparison Code

3.5.4 Optimising (out) load and stores.

The frontend does not generate code that attempts to smartly assign values to registers – instead it emits simple code that more directly corresponds to the C abstract machine (e.g., local variables in C exist on an abstract “stack”, we implement this by literally always storing them on the stack).

This leads to poor code, since often the stack can be omitted entirely and a variable can be stored in a register for its whole lifetime (for example if the variable has short lifetime, and never has its address taken).

To address this, a pass called *mem2reg* was developed (inspired by LLVM's own *mem2reg* pass which has a similar effect but exists for a different reason¹⁰). This performs a pass over all the stack

¹⁰ LLVM's *mem2reg* exists for transforming non SSA form IR emitted by Clang into the SSA form that is used for the rest of the compiler. It is a convenient side effect that it also transforms values from using the stack to registers.

variables in the IR and applies some simple heuristics to determine if it is safe to move the value into a register, as follows:

- Is this an array? If so, retain on the stack (a more complex implementation could possibly do alias analysis and attempt to split the array into individual registers, but it is pragmatic to exclude this variation here¹¹).
- Is this a struct? If so, retain on the stack (for the same reason as above⁶).
- Is the address of the variable (commonly known as a pointer) ever used? If so, retain on the stack.
- If none of the above apply, assign this value to a register and replace all loads and stores with the appropriate register/register IR constructs.

Implementing this transform provided a significant performance increase (~31.4% compared to the performance after the peephole optimisations, with the overall increase compared to the original timings being approximately 46%).

Compiler	Time	CPU	Iterations
<i>Helix</i>	<i>937ns</i>	<i>937ns</i>	<i>745278</i>
TCC	1447	1434ns	488615
Clang	4.05ns	4.01ns	174875749
GCC	362ns	358ns	1957394

Table 3.4 Benchmark results after mem2reg implementation

3.5.5 Conclusion

Overall, this process has been a success – whilst the performance is not on par with that of GCC or Clang, it is a significant improvement on what it was.

Additional optimisations could also be usefully considered, for example, due to the simplicity of this benchmark there is limited scope for constant propagation to improve performance but in larger blocks of code it would have a larger impact (this is addressed in a later section).

It is also worth noting that there were some optimisations implemented that didn't provide the expected improvements and as such are not covered here – for example, a change to how integers are loaded into registers (previously these would be stored in memory and then loaded into a register but was simplified to instead split the integer in half and load via a *movw/movt* pair¹²).

Whilst these additional optimisations didn't result in any significant performance increase, they overall increased the quality of the generated code thus provided a net benefit.

¹¹ A simpler improvement would be to identify arrays or structs that are word size or smaller and consider them as candidates for registers.

¹² *movw* loads a 16-bit immediate value into the lower 16 bits of the target register without clearing the top 32 bits, *movt* load into the top 16 bits, the equivalent of (`<upper_half> << 16> | <lower_half>`)

3.6 Optimising “Addition” Benchmark (Constant Propagation)

3.6.1 Introduction

The next optimisation to implement is constant propagation, specifically the *Simple Constant* propagation algorithm previously described.

Constant propagation is especially useful for optimising code generated because of other optimisations. Programmers themselves don’t tend to normally write code that is trivially foldable (e.g., `int n = 10 * 20`), but the use of constant variables and macros tends to result in code with a lot of opportunities for optimization.

The “addition” benchmark (given in figure 3.12), shows a good example of code that is a good target for constant propagation.

```
int main()
{
    int a = 10;
    int b = 20;
    return a + b + 30;
}
```

Figure 3.12 “addition” benchmark.

Table 3.5 presents the benchmarking timings before constant propagation was added. The conditions used to take these benchmarks are described in the *Testing Methodology* section.

Compiler	Time	CPU	Iterations
<i>Helix</i>	<i>9.34ns</i>	<i>9.34ns</i>	<i>74912588</i>
TCC	6.04ns	6.03ns	116575241
Clang	4.00ns	4.00ns	174890613
GCC	4.00ns	4.00ns	174892984

Table 3.5 Benchmark results before constant propagation

To improve this benchmark, we can implement the *Simple Constant* propagation algorithm.

```

.section .data
.text
.globl main
main:
    push {r4, r5, r6, r7, r8, r10, r11, lr}
    mov r11, sp
.bb0:
    sub r13, r13, #0
    movw r5, #10
    movt r5, #0
    mov r4, r5
    movw r6, #20
    movt r6, #0
    mov r8, r6
    add r7, r4, r8
    movw r4, #30
    movt r4, #0
    add r5, r7, r4
    mov r0, r5
    b .bb1
.bb1:
    add r13, r13, #0
    mov sp, r11
    pop {r4, r5, r6, r7, r8, r10, r11, lr}
    bx lr

```

Prologue/Epilogue

Program Code

Consider the figure 3.13, which shows the assembly generated for the “addition” benchmark before constant propagation.

This is doing a lot more work than is necessary. We know from the source code that the variables containing 10 and 20 never change, and that the addition should always evaluate to 60 (10 + 20 + 30).

Compare this to figure 3.14 (below), which shows the assembly generated by Clang for the same program.

```

main:
    mov     r0, #60
    bx     lr

```

Figure 3.14 Optimised “addition” benchmark

Figure 3.13 “addition” benchmark assembly, before constant propagation

The constant propagation algorithm was implemented as a new pass, *SCP*. This pass was executed as a platform independent IR pass, before the translation from HLIR to LLIR.

There were some problems that arose during development, mainly around the data representation. The simple constant propagation algorithm requires the whole program to be represented as a graph, since each individual instruction is required to know the state of any constants coming into it and coming out of it.

This presented a problem, since this compilers IR has a hybrid structure, with a graph that represents flow between blocks and not between individual instructions.

To combat this, a new (temporary) IR was constructed at the start of the constant propagation pass. This new IR was much simpler than the HLIR used by the rest of the compiler, containing only information required for constant propagation. This new IR represented the whole program as a graph, with control flow being represented between instructions, and not blocks.

This however does take a relatively significant amount of time at the start of the pass, as it involves doing a full pass over the whole programs IR. Further work could revisit this and investigate ways for the existing data structures to be utilised instead.

The entire algorithm is too verbose to be completely detailed here, but it roughly follows the outline given in section 2.4.3 *Simple Constant Propagation*.

```

.section .data
.text
.globl main
    push {r4, r5, r6, r7, r8, r10, r11, lr}
    mov r11, sp
.bb0:
    sub r13, r13, #0
    b .bb1
.bb1:
    movw r4, #60
    movt r4, #0
    mov r0, r4
    add r13, r13, #0
    mov sp, r11
    pop {r4, r5, r6, r7, r8, r10, r11, lr}
    bx lr

```

■ Prologue/Epilogue
■ Program Code

Figure 3.15 shows the assembly for the same “addition” benchmark after constant propagation was added.

Note how the actual program code has shrunk significantly, with it now essentially just loading 60 into the return value register.

Figure 3.15 “addition” benchmark assembly, after constant propagation

Table 3.6 shows the new benchmarking results (with the original values noted in red alongside).

Compiler	Time	CPU	Iterations
<i>Helix</i>	7.34ns (9.34ns)	7.34ns (9.34ns)	95317634 (74912588)
TCC	6.12ns	6.12ns	116575961
Clang	4.00ns	4.00ns	174890856
GCC	4.00ns	4.00ns	174895226

Table 3.6 Benchmark results after constant propagation

Whilst the implementation of constant propagation has provided some performance improvement, we’re still slow ever than the other tested compilers. This can be attributed to the large amount of prologue & epilogue code inserted unconditionally by the code generator.

```

.section .data
.text
.globl main
    push {r4, r5, r6, r7, r8, r10, r11, lr}
    mov r11, sp
.bb0:
    sub r13, r13, #0
    b .bb1
.bb1:
    movw r4, #60
    movt r4, #0
    mov r0, r4
    add r13, r13, #0
    mov sp, r11
    pop {r4, r5, r6, r7, r8, r10, r11, lr}
    bx lr

```

Note lines 7 and 13 in Figure 3.16, which subtract 0 from r13/add 0 to r13 respectively.

These are emitted by the register allocator – in a program that uses the stack this would move SP to reflect the size of the stack.

In this program all data is stored in registers, so the stack is not in use.

Figure 3.16 Highlighting unnecessary stack management code

The fix for this is to adjust the register allocator to check if the stack is used before emitting this code. This results in these unnecessary additions/subtractions being removed & provides further performance improvement.

Overall, implementing this fix gives a roughly 28% improvement on the original benchmark, and roughly a 9% improvement compared to the benchmark after constant propagation was added.

Compiler	Time	CPU	Iterations
<i>Helix</i>	<i>6.67ns</i>	<i>6.67ns</i>	<i>104836863</i>
TCC	6.06ns	6.06ns	116583080
Clang	4.00ns	4.00ns	174893510
GCC	4.00ns	4.00ns	174891042

Table 3.7 Benchmarking results after removing the unnecessary additions and subtractions.

3.6.2 Further Work

There is further potential for improvements, for example, note the circled pushes & pops in Figure 3.17.

```

.section .data
.text
.globl main
    push {r4, r5, r6, r7, r8, r10, r11, lr}
    mov r11, sp

.bb0:
    sub r13, r13, #0
    b .bb1

.bb1:
    movw r4, #60
    movt r4, #0
    mov r0, r4
    add r13, r13, #0
    mov sp, r11
    pop {r4, r5, r6, r7, r8, r10, r11, lr}
    bx lr

```

Figure 3.17 Highlighting unnecessary pushing and popping to the stack

These pushes and pops are unconditionally inserted for every function. This comes from the *Procedure Call Standard for the ARM Architecture* (ARM Software, 2022), which declares registers r4-r8 and r10-r11¹³ as *Variable Registers*, stating that:

A subroutine must preserve the contents of the registers r4-r8, r10, r11 and SP

¹³ For this target platform (Linux), r11 is not a variable register, but instead is the frame pointer.

```

.section .data
.text
.globl main
main:
    push {r4, r11, lr}
    mov r11, sp
.bb0:
    b .bb1
.bb1:
    movw r4, #60
    movt r4, #0
    mov r0, r4
    mov sp, r11
    pop {r4, r11, lr}
    bx lr

```

The code generator then takes a conservative approach, pushing all requires that must be preserved to the stack¹⁴. A smarter approach would be to instead only push to the stack variable registers that have been modified by the program. In this case, that would just be *r4*, allowing the omission of *r5-r8* and *r10*.

Figure 3.18 shows how the assembly would look given this improvement.

Figure 3.18 Highlighting further improvements to the assembly

3.6.3 Conclusion

In conclusion, adding constant propagation to the compiler has been incredibly useful, and is something that has been long needed. There were undoubtedly problems that arose during development and debugging constant propagation bugs turned out to be particularly difficult at times (owing to how it interacts with the flow of control of the whole program).

And whilst the benchmarks haven't been improved to beat any of the other compilers, the work required to do this (or to at least get on par with) would be relatively simple and just come down to detecting situations in which parts of the prologue and epilogue can be omitted.

¹⁴ Note that AAPCS also states that the stack pointer *SP*, must be preserved. This is not done by pushing *SP* to the stack directly, but instead is managed by storing *SP* in the frame pointer at the start & end of each function.

3.7 Final Performance & Testing Results

This section gives testsuite and benchmark results, as of the end of development.

3.7.1 Performance Results

Benchmark Name	Helix	TCC	Clang	GCC
Add	7.34ns	6ns	4ns	4ns
Binary Search	84.7ns	96.1ns	17.3ns	43ns
Fibonacci	55.4ns	75.9ns	4ns	4ns
Sieve	3510ns	3750ns	793ns	817ns
String Length	947ns	1448ns	4ns	358ns
Byte Sieve	2684735ns	3434801ns	765788ns	729939ns

Table 3.5 Benchmark performance per compiler

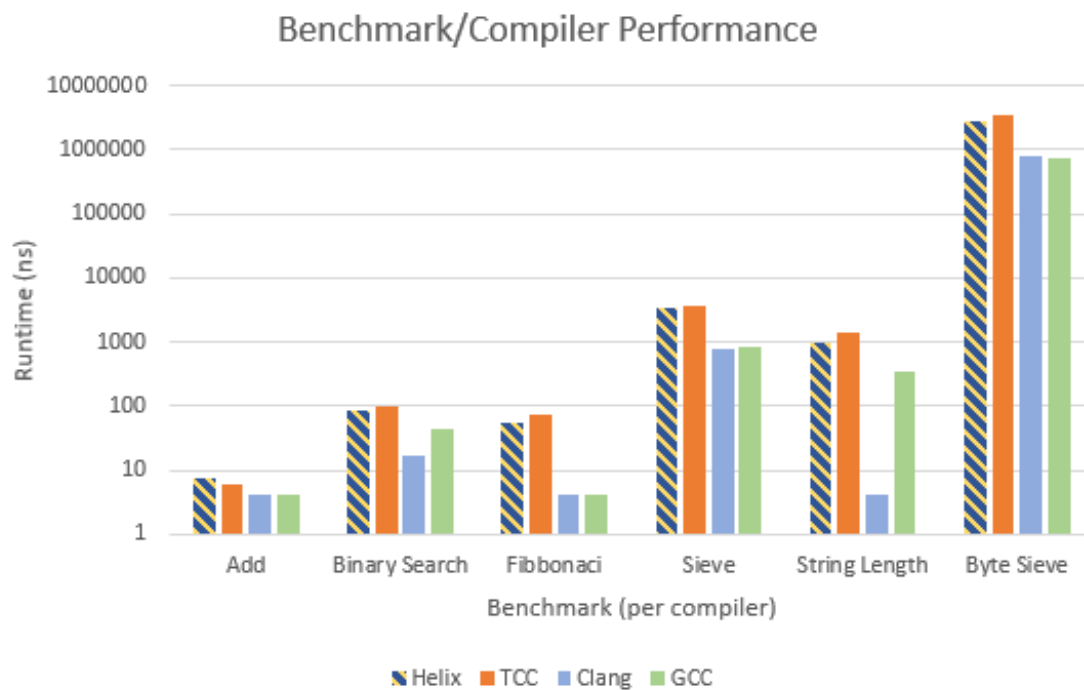


Figure 3.12 Benchmark performance per compiler (note log (10) vertical axis scale)

3.7.2 Testsuite Results

Test Type	Passes	Fails	Skipped	Total	Notes
Integration Tests	66	0	0	66	
Unit Tests	90	0	0	90	Across 276 individual assertions.
c-testsuite (External)	91	66	63	220	Tests are skipped if they use unsupported features, e.g., any tests that use LibC are skipped.

Table 3.6

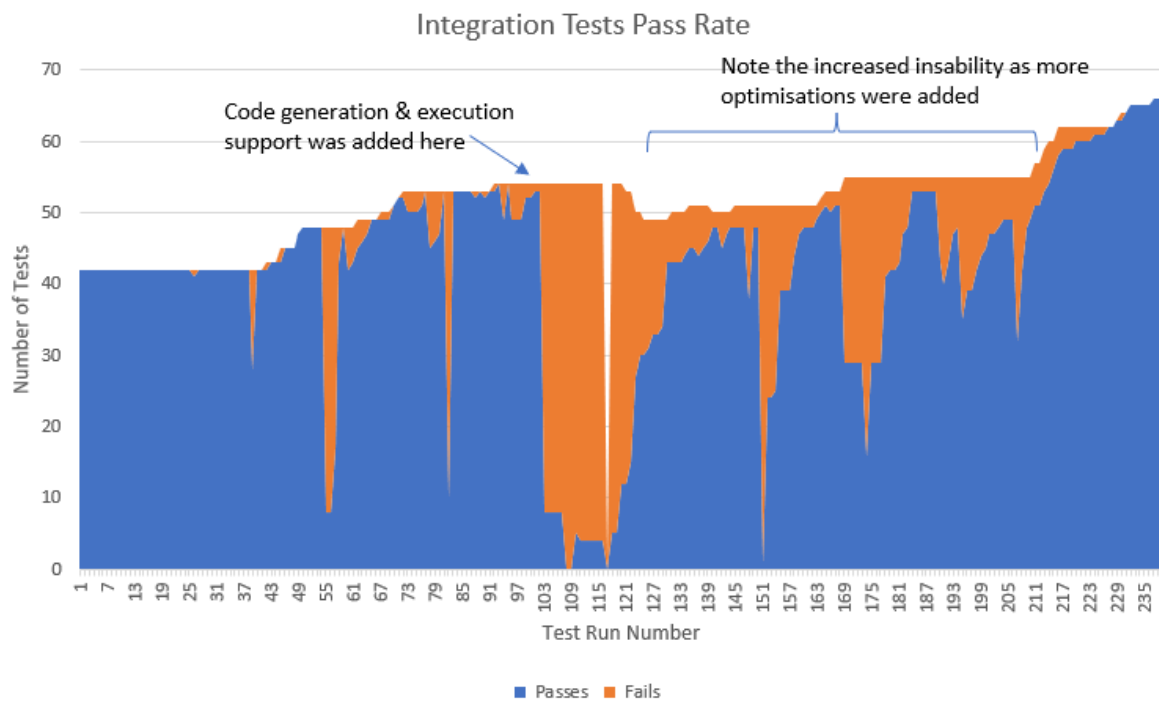


Figure 3.13 Test pass rate for the integration tests over time

Note: In the graph in Figure 3.13, the drop in the middle where both passes and fails fall to zero. This was unintentional, and the result of a bug in the test runner. It's included here for completeness.

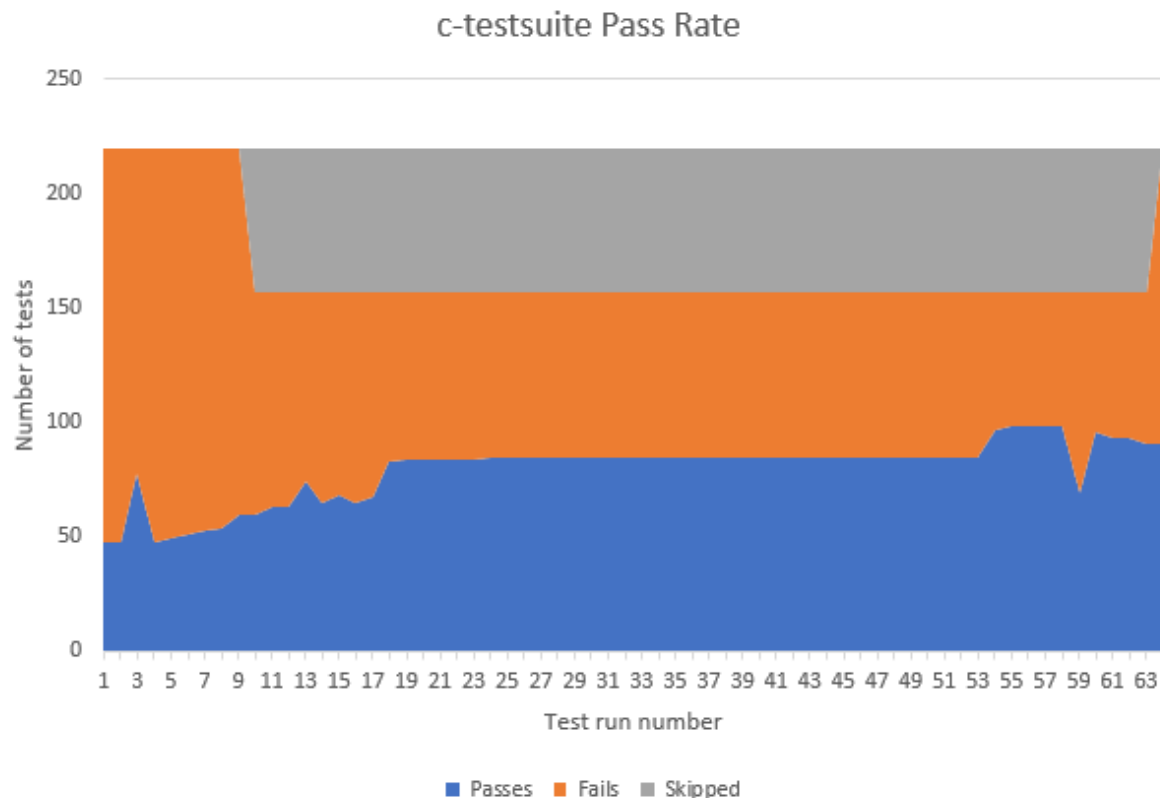


Figure 3.14 c-testsuite pass rate over time

4 Evaluation

4.1 Retrospective on the structure and design of the compiler

The overall structure and design of the compiler is solid, and I believe could stand up to more work, however there are a couple of areas that could be improved.

4.1.1 IR

The 3-address linear form of the IR itself is good, however the data structures and handling of instructions has some problems, namely the instructions being allocated on the heap, but support for then deleting those instructions is poor, and memory is often left dangling. This is OK for small programs, where the overall memory footprint isn't huge however when compiling large programs, this could cause problems. Another route for investigation would be adding reference counting to the instructions – currently it is down to the programmers to decide if it is safe to destroy an instruction at any point, and hope that no one is holding any references to that instruction, this led to a number of use after free bugs which were non obvious to debug.

The IR does not use SSA form, and this was a deliberate choice because during planning and research it was believed that using SSA would add unnecessary complexity to the project. In hindsight, it might have been worth designing the IR around SSA, since whilst there would be some extra complexity in generating code from the frontend in SSA form (there are workarounds for that), having an SSA form IR simplifies many more optimisations down the line, since it removes the need for quite a lot of data flow analysis – simple constant propagation becomes trivial since you can replace all uses of a constant with the constant itself. There is a constant propagation algorithm called Sparse Simple Constant Propagation (Wegman & Zadeck, Constant Propagation with Conditional, 1991) that is both faster than conventional constant propagation, finds more constants

and simultaneously can do some forms of dead code removal. This is just one example, there are many more (register allocation being another).

Bar those changes (and a bunch of other smaller ones, not worth noting) the IR was remarkably flexible, and the core system barely had to be changed from the original implementation. I believe this is down to good planning, and basing it (roughly) off existing compilers and using well researched concepts

4.2 Testing and Correctness

Testing a compiler could be a project in of itself, and there are still papers coming out today on this topic (CHEN & SUO, 2021). As a result, the testing done for this compiler can by no means be considered comprehensive. Ensuring good test coverage became significantly more complex as the project grew, due to limitations of the testing systems and the complexity of the compiler. For example, early on during development it was easy to write tests that compiled C to IR, dumped that IR to file and compared against expected output. However as more passes were added to the compiler, it became more difficult to test specific transforms since the IR would be affected by previous passes.

One example of this is when constant propagation was added, it ended up completely optimising out tests that had been written much earlier in the project for integer arithmetic, causing those tests to fail when they shouldn't. There are a couple of ways to combat this (e.g., being able to run passes in isolation and having a textual IR form as input to minimise the amount of code that could be unknowingly changed), however this project was ambitious enough, and these would take a significant amount of time to get right.

Most of the bugs that arose were miscompilation bugs – where the compiler would complete supposedly successfully but there would be problems in the generated code (for example a register being assigned a new value before the lifetime of its current value has ended). These were often confusing to debug and as a result soaked up a lot of time during development - since often simple test programs would work as expected, and the bug would only expose itself as part of a larger block of code. In one instance, there was a bug in the register allocator that went unnoticed during development and testing and was only noticed when investigating an unexpectedly slow benchmark – the bug was causing too many iterations of a loop, but the result of the program remained the same. These types of bugs are common (just look at the bug trackers for any open-source compiler) and are notoriously difficult to avoid ¹⁵.

These problems could have been given more consideration during the planning phase, however – as with any programming – most testing problems could be worked around well enough. Something to note is that testing was an ongoing process during development and wasn't done in some large effort towards the end of the project. This then meant that there wasn't a ton of bug fixing and problems with the final deliverable – that is with some limitations covered in the next section.

4.3 Limitations and Future Developments

Whilst in general I believe this project to be a success, it still has many limitations – the most significant of which is the non-complete implementation of the C language. For example, functions can only have 4 or less word size parameters, and only return word sized values (or void). Support

¹⁵ There have been recent attempts to try and formally prove the correctness of various optimisations, with aims to eliminate this type of miscompilation bug, but again trying to do this would be well outside the scope of this project (Lopes, Menendez, Nagarakatte, & Regehr, 2015)

for casting between types is limited, and function pointers are not supported at all. All these limitations exist simply due to the constraints of the project and being realistic about what is achievable, and since the focus of this presentation is on optimisation techniques, these features were pushed back. One route for future development is fleshing this out and adding support for more C features.

Future work would also need to focus on adding more optimisations to the middle and backends than were implemented for this project. Whilst this project provided a good proof of concept, any good optimising compiler would need many more optimisations that have not even been touched here, for example inter-procedural optimisations or (increasingly today) auto vectorisation and optimisations for parallel architectures.

As mentioned previously, there is a lot more work that would be worth doing around testing – adding a textual form of the IR that the compiler could accept, and the option of being able to isolate passes would be first on the list, as these would facilitate much more comprehensive and easier testing.

For the optimisations that are present, the code generation does not take full advantage of the capabilities of the target hardware, so this would be another avenue for improvement.

4.4 Conclusion

In conclusion, I believe that this project has been a success and has achieved the goal of the original specification - to have a compiler capable applying a select number of optimisations which improve the performance of generated code. The compiler itself is not a fully compliant C compiler, nor does it generate the fastest code but despite all of this it provides a good proof of concept and backing for further investigation and research.

4.4.1 Personal Development

Developing this compiler has given me critical insight into intimidating world of compilers and optimisations and provided critical insight: I believe that in programming you can read all you like about the whys, what's and how's of a topic, but you can never truly understand it until you do it yourself. This compiler has provided that practical experience for me. Whilst there have been mistakes, wasted time and things that in hindsight I would do differently, having that experience allows me to more deeply understand why things are done the way they are done, and back up any opinions I have.

In contrast to other projects in the past, I believe for this project my time keeping has generally been good – the objectives given in the specification have been (roughly) stuck to, and the project has not been done all at the last minute but stretched out nicely since October. It's not always been perfect, and there have been times of falling into old habits, especially around getting stuck in rabbit holes, fixated on one idea and wasting time when more important things could be being worked on (spending too much time on changes to the machine description tool come to mind – I added a load of features that just weren't needed for this project, and ended up being scrapped anyway).

However overall, working on this project has been an enriching and insightful process, giving me chance to work with an area that I had a little experience before, but was massively interested in. Whilst personally I would have liked to walk out with a fully optimising and complete C compiler, that obviously wasn't realistic – instead I came out with a deliverable that met the original aims and am all the better person for it.

References

- Bellard, F. (2018). *Tiny C Compiler Reference Documentation*. Retrieved from [bellard.org/tcc:](https://bellard.org/tcc/)
<https://bellard.org/tcc/tcc-doc.html>
- Chambers, A. (2020). *c-testsuite*. Retrieved from [c-testsuite/c-testsuite: https://github.com/c-testsuite/c-testsuite](https://github.com/c-testsuite/c-testsuite)
- CHEN, J., & SUO, C. (2021). Boosting Compiler Testing via Compiler Optimization Exploration. *ACM Transactions on Software Engineering and Methodology*.
- Compilers: Principles, Techniques, & Tools. (2007). In A. V. Aho, M. S. Lam, R. Sethi, & J. D. Ullman.
- Cytron, R., Ferrante, J., Barry, R. K., Wegman, M. N., & Zadeck, F. K. (1991). Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, VO1 13, NO 4.
- Eisl, J., Grimmer, M., Simon, D., Würthinger, T., & Mössenböck, H. (2016). Trace-based Register Allocation in a JIT Compiler. *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*.
- Free Software Foundation. (2022). Retrieved from [fsf.org: https://www.fsf.org/](https://www.fsf.org/)
- GCC. (2012). *C++ Conversion*. Retrieved from [gcc.gnu.org: https://gcc.gnu.org/wiki/cxx-conversion](https://gcc.gnu.org/wiki/cxx-conversion)
- GCC. (2022). *Building GCC on Windows*. Retrieved from [gcc.gnu.org: https://gcc.gnu.org/wiki/WindowsBuilding](https://gcc.gnu.org/wiki/WindowsBuilding)
- GCC Wiki. (2008). *GIMPLE*. Retrieved from [gcc.gnu.org: https://gcc.gnu.org/wiki/GIMPLE](https://gcc.gnu.org/wiki/GIMPLE)
- Gilbreath, J. (1981, 9). A High-Level Language Benchmark. *BYTE Magazine*.
- Intel Corporation. (2021, 09 08). *Intel C/C++ compilers complete adoption of LLVM*. Retrieved from [intel.com: https://www.intel.com/content/www/us/en/developer/articles/technical/adoption-of-llvm-complete-icx.html](https://www.intel.com/content/www/us/en/developer/articles/technical/adoption-of-llvm-complete-icx.html)
- ISO/IEC 9899. (2018). *Programming languages — C*. ISO.
- LLVM Project. (2022). *Opaque Pointers*. Retrieved from [llvm.org: https://llvm.org/docs/OpaquePointers.html](https://llvm.org/docs/OpaquePointers.html)
- LLVM Project. (2022). *Projects built with LLVM*. Retrieved from [llvm.org: https://llvm.org/Projects/WithLLVM/#rust](https://llvm.org/Projects/WithLLVM/#rust)
- Lopes, N. P., Menendez, D., Nagarakatte, S., & Regehr, J. (2015). Provably correct peephole optimizations with alive. *ACM SIGPLAN Notices*.
- Merrill, J. (2003). GENERIC and GIMPLE: A New Tree. *Proceedings of the GCC Developers Summit*, (pp. 171-179).
- SARKAR, V., & POLETO, M. (1999). Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*.
- Stallman, R. (2014, 1 25). *Re: clang vs free software*. Retrieved from [lists.gnu.org/: https://lists.gnu.org/archive/html/emacs-devel/2014-01/msg02071.html](https://lists.gnu.org/archive/html/emacs-devel/2014-01/msg02071.html)

Torczon, L., & Cooper, K. D. (2012). *Engineering a Compiler (2nd edition)*. Morgan Kaufmann.

Wegman, M. N., & Zadeck, F. K. (1991). Constant Propagation with Conditional. *ACM Transactions on Programming Languages and Systems*.

Wegman, M. N., & Zadeck, F. K. (1991). Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems, Vol 13, No 2*.

Appendix 1 – High Level Intermediate Language (HLIR)

Opcode	Syntax	Description
iadd	<i>iadd</i> <lhs>, <rhs>, <output>	Integer addition
isub	<i>isub</i> <lhs>, <rhs>, <output>	Integer subtraction
imul	<i>imul</i> <lhs>, <rhs>, <output>	Integer multiplication
iudiv	<i>iudiv</i> <lhs>, <rhs>, <output>	Unsigned integer division
isdiv	<i>isdiv</i> <lhs>, <rhs>, <output>	Signed integer division
isrem	<i>isrem</i> <lhs>, <rhs>, <output>	Signed integer modulo
iurem	<i>iurem</i> <lhs>, <rhs>, <output>	Unsigned integer modulo
and	<i>and</i> <lhs>, <rhs>, <output>	Bitwise and
or	<i>or</i> <lhs>, <rhs>, <output>	Bitwise or
shl	<i>shl</i> <lhs>, <rhs>, <output>	Logical shift left
shr	<i>shr</i> <lhs>, <rhs>, <output>	Logical shift right
xor	<i>xor</i> <lhs>, <rhs>, <output>	Bitwise exclusive or
load	<i>load</i> <mem>, <reg>	Load value from memory
store	<i>store</i> <reg>, <mem>	Store value to memory
stack_alloc	<i>stack_alloc</i> [<type>], <reg>	Allocate space on stack
lea	<i>lea</i> [<type>], <ptr>, <idx>, <output>	Calculate array element address
lfa	<i>lfa</i> [<type>:idx], <ptr>, <output>	Calculate struct field address
set	<i>set</i> <dst>, <src>	Set a register to a value
cbr	<i>cbr</i> <tgt_true>, <tgt_false>, <cond>	Conditional branch
br	<i>br</i> <target>	Unconditional branch
ret	<i>ret</i>	Return
call	<i>call</i> <result>, <function>, <args>...	Call a function
icmp_neq	<i>icmp_neq</i> <lhs>, <rhs>, <output>	Compare, values not equal
icmp_eq	<i>icmp_eq</i> <lhs>, <rhs>, <output>	Compare, values equal
icmp_lt	<i>icmp_lt</i> <lhs>, <rhs>, <output>	Compare, less than
icmp_gt	<i>icmp_gt</i> <lhs>, <rhs>, <output>	Compare, greater than
icmp_le	<i>icmp_le</i> <lhs>, <rhs>, <output>	Compare, less than or equal
icmp_ge	<i>icmp_ge</i> <lhs>, <rhs>, <output>	Compare, greater than or equal
ptrtoint	<i>ptrtoint</i> <input>, <output>	Convert pointer to integer
inttoptr	<i>inttoptr</i> <input>, <output>	Convert integer to pointer
sext	<i>sext</i> <input>, <output>	Sign Extend
zext	<i>zext</i> <input>, <output>	Zero Extend

Appendix 2 – String Length Disassembly

Taken before any optimisations had been applied.

0:	e92d0df0	push	{r4, r5, r6, r7, r8, sl, fp}	Entry Block & Setup
4:	e1a0b00d	mov	fp, sp	
8:	e24dd018	sub	sp, sp, #24	
c:	e28d0014	add	r0, sp, #20	
10:	e58d0008	str	r0, [sp, #8]	
14:	e28d4010	add	r4, sp, #16	
18:	e3007000	movw	r7, #0	
1c:	e3407000	movt	r7, #0	
20:	e5847000	str	r7, [r4]	
24:	e28d000c	add	r0, sp, #12	
28:	e58d0004	str	r0, [sp, #4]	
2c:	e3006000	movw	r6, #0	
30:	e3406000	movt	r6, #0	
34:	e5968000	ldr	r8, [r6]	
38:	e59d0004	ldr	r0, [sp, #4]	
3c:	e5808000	str	r8, [r0]	
40:	ea000000	b	44 <.bb1>	
00000044 <.bb1>:				
44:	e5946000	ldr	r6, [r4]	Loop "Head"
48:	e59d0004	ldr	r0, [sp, #4]	
4c:	e5908000	ldr	r8, [r0]	
50:	e1a05006	mov	r5, r6	
54:	e3006000	movw	r6, #0	
58:	e3406000	movt	r6, #0	
5c:	e5967000	ldr	r7, [r6]	
60:	e0060798	mul	r6, r8, r7	
64:	e0858006	add	r8, r5, r6	
68:	e1a06008	mov	r6, r8	
6c:	e5967000	ldr	r7, [r6]	
70:	e3005000	movw	r5, #0	
74:	e3405000	movt	r5, #0	
78:	e5958000	ldr	r8, [[r5]]	
7c:	e1570008	cmp	r7, r8	
80:	e3a07000	mov	r7, #0	
84:	13007001	movwne	r7, #1	
88:	e3570001	cmp	r7, #1	
8c:	aa000000	bge	94 <.bb2>	
90:	ea000008	b	b8 <.bb3>	
00000094 <.bb2>:				
94:	e59d0004	ldr	r0, [sp, #4]	Loop Body
98:	e5907000	ldr	r7, [r0]	
9c:	e3006000	movw	r6, #0	
a0:	e3406000	movt	r6, #0	
a4:	e5965000	ldr	r5, [r6]	
a8:	e0878005	add	r8, r7, r5	
ac:	e59d0004	ldr	r0, [sp, #4]	
b0:	e5808000	str	r8, [r0]	
b4:	ea0000e2	b	44 <.bb1>	
000000b8 <.bb3>:				
b8:	e59d0004	ldr	r0, [sp, #4]	Exit Block
bc:	e5907000	ldr	r7, [r0]	
c0:	e59d0008	ldr	r0, [sp, #8]	
c4:	e5807000	str	r7, [r0]	
cc:	e59d0008	ldr	r0, [sp, #8]	
d0:	e5900000	ldr	r0, [r0]	
d4:	e28dd018	add	sp, sp, #24	
d8:	e8bd0df0	pop	{r4, r5, r6, r7, r8, sl, fp}	
dc:	e12fff1e	bx	lr	

Appendix A – Project Specification

PROJECT SPECIFICATION - Project (Technical Computing) 2021/22

Student:	Barnaby Wilks
Date:	8th October 2021
Supervisor:	Adrian Oram
Degree Course:	BSc (Hons) Computer Science
Title of Project:	Compiler Optimization Techniques

Elaboration

It is becoming increasingly rare in the world of programming to have to write low level assembly to get the most out performance out of a piece of hardware. In fact, the details of the hardware are getting more and more irrelevant as more and more software moves to running in web browsers, a land of abstractions built upon more abstractions.

Even when high performance software is required, the code still gets written in a “high level” language and the compiler is relied upon to make it run fast (e.g either by generating machine code ahead of time - like most C/C++ compilers, or at runtime by using state of the art just in time compilation techniques - e.g. the .NET or Java virtual machines).

These modern optimising compilers apply many optimisation techniques in order to improve some metrics of the generated code (commonly either speed or size optimisations, however energy aware are increasingly a concern, especially for small embedded chips). These techniques exist both at a target agnostic level (e.g. it doesn't depend on the CPU the code will run on) and on a target specific level (e.g. optimisations specifically for the machine the code will run on).

The goal of this project is to investigate and implement a compiler that makes use of some of these techniques in order to improve the performance of code for an Arm CPU. The chosen optimisations will be selected to be both target agnostic and target specific, e.g. dead code elimination, constant propagation at a high level and register allocation or peephole optimisations at a low level.

As a note, it is not the goal of this project to implement a full programming language from scratch, the language frontend itself is irrelevant and the primary concerns will be the machine code generation and optimisation.

Also as a note, the optimisations and techniques investigated and implemented as part of this project will most likely not be comprehensive, novel or able to compete with existing production optimising compilers. This project will be building upon common literature and research that has been refined over many years, as have the existing compilers that also use them. This project will serve more as an introduction and base for a small and specific set of optimisations for specific hardware (this compiler will not be retargetable as modern compilers often are - e.g. it will only produce Arm code and not also x86 etc..)

Project Aims

- A basic intermediate representation, capable of being optimised and generating machine code.
- A targeted and specific chosen set of optimisations - to improve the performance of the generated code.
- Performance analysis and benchmarking of the generated code.

Project deliverable(s)

The deliverable for this project will be some kind of compiler capable of running on the host machine (e.g. the machine of whoever is compiling the code, in this case Windows) and generating optimised code for the target machine (a system with an Arm CPU, probably some embedded device).

Input to this compiler will be a chosen language frontend, but that is not the focus of the project.

Action plan

Objective Deadlines	
Action	Rough Completion Date
<i>Phase One (Research and setup)</i> <ul style="list-style-type: none"> • Initial research and investigations: <ul style="list-style-type: none"> ○ Identifying a suitable language frontend ○ Identifying a suitable set of optimisations to be implemented in phase two. ○ Identifying and evaluating suitable target hardware. • Design an initial intermediate language representation, integrated with the chosen frontend. 	<i>Overall</i> November 29th 2021 (~1 month) <ul style="list-style-type: none"> • 15th November 2021 • November 29th 2021
<i>Phase Two (Initial optimisations and code generation)</i> <ul style="list-style-type: none"> • Initial & naive code generation. This code will not be optimised, but should be executable on the chosen hardware. • Implementation of the starting set of optimisations chosen in phase one. 	<i>Overall</i> January 24th 2022 (~2 months) <ul style="list-style-type: none"> • December 20th 2021 • January 24th 2022
<i>Phase Three (Performance analysis and targeted optimisations)</i> <ul style="list-style-type: none"> • Finding or implementing a set of benchmarks. • Profiling generated code using those benchmarks and performance analysis. 	<i>Overall</i> March 14th (~2 months) <p><i>Tasks are all mixed and interweaved, so distinct deadlines don't make sense.</i></p>

- Investigation of optimisations that could improve those benchmarks (with tentative implementations, time and complexity permitting).

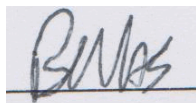
Module Deadlines

Action	Completion Date
Find project supervisor	8th October 2021
Submission of the Project Specification and ethics form	22nd October 2021
The information review	3rd December 2021
The provisional contents page	18th February 2022
Critical evaluation and sections of your draft report	18th March 2022
Upload the project report and deliverable	7th April 2022
Demonstration	5th May 2022

BCS Code of Conduct

I confirm that I have successfully completed the BCS code of conduct on-line test with a mark of 70% or above. This is a condition of completing the Project (Technical Computing) module.

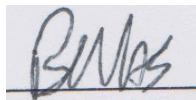
Signature: Barnaby Wilks



Publication of Work

I confirm that I understand the "Guidance on Publication Procedures" as described on the Bb site for the module.

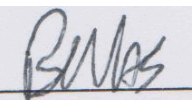
Signature: Barnaby Wilks



GDPR

I confirm that I will use the "Participant Information Sheet" as a basis for any survey, questionnaire, or participant testing materials. The participant information sheet form is available on the Bb site for the module and as an appendix in the handbook.

Signature: Barnaby Wilks



Appendix B – Ethics Form



UREC 1 RESEARCH ETHICS REVIEW FOR STUDENT RESEARCH WITH NO HUMAN PARTICIPANTS OR DIRECT COLLECTION OF HUMAN TISSUES, OR BODILY FLUIDS.

All University research is required to undergo ethical scrutiny to comply with UK law. The SHU [Research Ethics Policy](#) should be consulted before completing the form. Answering the questions below will confirm that the study fits this category and that any necessary approvals or safety risk assessments are in place. The supervisor will approve the study, but it may also be reviewed by the College Teaching Programme Research Ethics Committee (CTPREC) as part of the quality assurance process.

The final responsibility for ensuring that ethical research practices are followed rests with the supervisor for student research.

Note that students and staff are responsible for making suitable arrangements to ensure compliance with the General Data Protection Regulations (GDPR), for keeping data secure and if relevant, for keeping the identity of participants anonymous. They are also responsible for following SHU guidelines about data encryption and research data management. Information on the [ethics website](#).

The form also enables the University and College to keep a record confirming that research conducted has been subjected to ethical scrutiny.

The form may be completed by the student and the supervisor and/or module leader (as applicable). In all cases, it should be counter-signed by the supervisor and/or module leader, and kept as a record showing that ethical scrutiny has occurred. Students should retain a copy for inclusion in the appendices of their research projects, and a copy should be uploaded to the module Blackboard site for checking.

Please note if it may be necessary to conduct a health and safety risk assessment for the proposed research. Further information can be obtained from the Safety Co-ordinator.

1. General Details

Name of student	Barnaby Wilks
SHU email address	b8017695@my.shu.ac.uk
Course or qualification (student)	BSc Computer Science
Name of supervisor	Adrian Oram
email address	a.oram@shu.ac.uk
Title of proposed research	Compiler Optimization Techniques
Proposed start date	25 th October 2021
Proposed end date	7 th April 2022
Brief outline of research to include, rationale & aims (250-500 words).	<p>Research involves the investigation and implementation of basic compiler optimisation techniques, with the aim of generating performant code to execute on an Arm based system.</p> <p>A small set of optimisations will be identified and implemented as a base line.</p> <p>This generated code then will be benchmarked - further investigation and performance analysis of these benchmarks will be performed, prompting research into which/how compiler optimisation techniques could improve their performance.</p> <p>Intent is to investigate how modern compilers optimise code, and its significance.</p>

I confirm that this study does not involve collecting data from human participants
☒ ☐ (please tick)

2. Research in Organisations

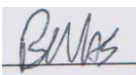
Question	Yes/No
1. Will the research involve working with/within an organisation (e.g. school, business, charity, museum, government department, international agency, etc.)?	No
1. If you answered YES to question 1, do you have granted access to conduct the research? <i>If YES, students please show evidence to your supervisor. PI should retain safely.</i>	N/A

<p>1. If you answered NO to question 2, is it because:</p> <ul style="list-style-type: none"> 0. you have not yet asked 1. you have asked and not yet received an answer 2. you have asked and been refused access. <p><i>Note: You will only be able to start the research when you have been granted access.</i></p>	N/A
---	-----

3. Research with Products and Artefacts

Question	Yes/No
1. Will the research involve working with copyrighted documents, films, broadcasts, photographs, artworks, designs, products, programmes, databases, networks, processes, existing datasets or secure data?	No
<p>2. If you answered YES to question 1, are the materials you intend to use in the public domain?</p> <p><i>Notes: 'In the public domain' does not mean the same thing as 'publicly accessible'.</i></p> <ul style="list-style-type: none"> • Information which is 'in the public domain' is no longer protected by copyright (i.e. copyright has either expired or been waived) and can be used without permission. • Information which is 'publicly accessible' (e.g. TV broadcasts, websites, artworks, newspapers) is available for anyone to consult/view. It is still protected by copyright even if there is no copyright notice. In UK law, copyright protection is automatic and does not require a copyright statement, although it is always good practice to provide one. It is necessary to check the terms and conditions of use to find out exactly how the material may be reused etc. <p><i>If you answered YES to question 1, be aware that you may need to consider other ethics codes. For example, when conducting Internet research, consult the code of the Association of Internet Researchers; for educational research, consult the Code of Ethics of the British Educational Research Association.</i></p>	N/A
<p>3. If you answered NO to question 2, do you have explicit permission to use these materials as data?</p> <p><i>If YES, please show evidence to your supervisor.</i></p>	N/A
<p>4. If you answered NO to question 3, is it because:</p> <ul style="list-style-type: none"> A. you have not yet asked permission B. you have asked and not yet received an answer C. you have asked and been refused access. <p><i>Note You will only be able to start the research when you have been granted permission to use the specified material.</i></p>	<p>A/B/C N/A</p>

Adherence to SHU policy and procedures

Personal statement	
I can confirm that: <ul style="list-style-type: none"> • I have read the Sheffield Hallam University Research Ethics Policy and Procedures • I agree to abide by its principles. 	
Student	
Name: Barnaby Wilks	Date: Monday 18 th October 2021
Signature: 	
Supervisor or other person giving ethical sign-off	
I can confirm that completion of this form has confirmed that this research does not involve human participants. The research will not commence until any approvals required under Sections 3 & 4 have been received and any health and safety measures are in place.	
Name: Dr Adrian Oram	Date: 22nd October 2021
Signature: A Oram	
Additional Signature if required:	
Name:	Date:
Signature:	

Please ensure the following are included with this form if applicable, tick box to indicate:

	Yes	No	N/A
Research proposal if prepared previously	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Any associated materials (e.g. posters, letters, etc.)	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Health and Safety Project Safety Plan for Procedures	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

