Binary Code Translation from Register to Stack based code

Charles Pigott
Supervisor: Dr. Christopher Crispin-Bailey

3rd May 2017

Abstract

This project aims to show that by converting code written for a simple register-based architecture or processor to a stack-based architecture, performance improvements can be found. A review is made of the history of register and stack-based computing, and previous work on stack optimisation techniques. A combined translator and interpreter for running register code on a stack machine is designed and implemented (`https://github.com/LordAro/reg2stack`). Several different optimisation models implemented by the software are then tested, the results of which are evaluated.

The main body of the report (excluding the appendices) is 10388 words, counted by `texcount`

Contents

Chapter 1

Introduction

1.1 Background

Transmeta were a US based company, who in 2000 surprised the computer architecture industry by developing a chip which dynamically translates Intel binary code into machine language instructions for another (highly optimised RISC) CPU core. In doing so it allows Intel code to execute in real-time without recompilation, but using a (claimed) much more power-efficient processor architecture.

1.2 Aims

The objective of this project is to show that by converting code written for a simple register-based architecture or processor to a stack-based architecture, performance improvements can be found. This includes an implementation that can translate code intended for a register-based architecture to run on a stack-based architecture. Testing the output of this implementation is intended to gain some performance related results and enable conclusions to be drawn from them.

1.3 Limitations

Sourcing physical processors and the toolchains capable of compiling and running code on both register-based and stack-based architectures is difficult, as existing systems tend to be either very out of date, with only partial support, or not exist on both platforms. Because of this, the project seeks to emulate the architectures instead.

Since the architectures will be emulated, there are very few advantages to continuing to use the architecture's native binary code, so this project will instead use the respective architecture's assembly language, rather than the binary code.

1.4 Statement of Ethics

This project has very few ethical concerns. This project only intends to implement already published work and show that their results can be replicated. In addition, Transmeta published results for their similar project over a decade ago with their Crusoe processor and have since closed down any microprocessor production operations and been bought out, therefore any implications of the results of this project will be unlikely to have any further affect on any commercial businesses. There is the case of transferring the ideas from this project to different architectures, where the source architectures in this project are fully evaluated for side effects, but changing to a different architecture may result in different results if not examined fully, depending on the architecture. This could cause issues for high-integrity software where correctness is crucial. The software produced by this project has been released under a permissive open-source licence now that it is completed, should anyone else want to take the project any further.

1.5 Problem approach

Initially, relevant literature is reviewed, with a focus on the work done by Transmeta, but also covers computer architectures, code translation and stack optimisation (Chapter 2). The knowledge from this is used to make a list of requirements for the implentation of the translation program (Chapter 3). The program is then designed and implemented, documenting the problems and solutions found along the way (Chapter 4). The completed implementation is then tested appropriately and results recorded and evaluated (Chapter 5). Finally, conclusions are drawn and options for any further work set out and discussed (Chapter 6).

Chapter 2

Literature Review

2.1 Computer architectures

With the exception of Babbage's Analytical Engine as described in 1837, probably the earliest computer architecture to be formally described is that of the EDVAC (Electronic Discrete Variable Automatic Computer) in 1945 by John von Neumann in his incomplete 'First Draft of a Report on the EDVAC' [1]. In it, von Neumann described a fully programmable computer which is subdivided into six separate parts — Central Arithmetic (CA); Central Control (CC); Memory; Input; Output and External Memory (at the time, this was intended to be something like punched tape). These components were described using the human nervous sytem as an analogy, with the CA, CC & Memory acting as associative (linking) neurons, and the Input & Output acting as sensory & motor neurons respectively. Von Neumann also wanted to keep the computer as simple as possible, so suggested that arithmetic operations (such as add and multiply) should not be overlapped, and performed only one digit (this being before bits and bytes were named as such) at a time. He also noted that external input/output should first be placed into memory, rather than directly in/out of the CC or CA. This approach to laying out the components of a computer prevailed, and the approximate idea is still used in modern processors today.

2.1.1 Register machines

In the 1950s, logic gates and switches were largely implemented using the quite large vacuum tubes and discrete transistors. With technology improving, integrated circuits were invented, constructing logic components using layers of metal and oxides on polished silicon. Initially, these were only used to implement individual logic components for computers, replacing the diodes and resistors used before, until in 1971, Intel released the 4004 microprocessor (Figure 2.1). It was the first commercially available fully self-contained microprocessor, with its circuitry fabricated using the new 'silicon gate' technology which is why it was able to be designed and fabricated as one chip. However, Intel as a company did not have much faith in this product, opting to instead focus on its

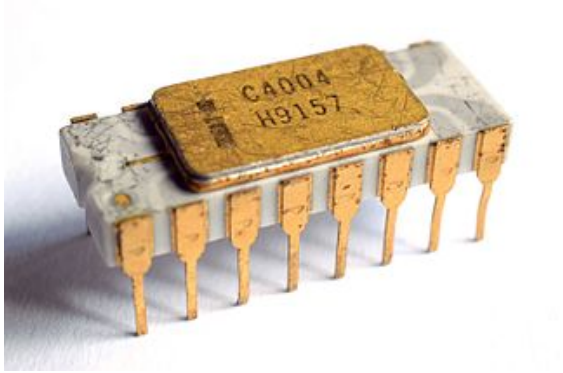


Figure 2.1: Intel's 4004 microprocessor

line of memory chips and partnering with a Japanese electronic calculator firm, Busicom, to help finance the project. Nonetheless, the 4004 was a huge commercial success being produced for 10 years and helping take Intel to the giant in the computing world that it is today [2].

In their simplest form, register architectures work by having several named locations (sometimes called the register file) within the CPU (Central Processing Unit) providing the fastest way to access data in a system, often in a single CPU cycle, and are used to store and retrieve intermediate results of computations. There are a number of different types of register, with data registers storing numerical values, address registers storing memory addresses for accessing system memory, general purpose registers store both data and addresses, flag registers for storing states of previous Boolean (true or false) test operations, and various others. Often there is also an instruction register that is not accessible to the program running on the CPU, but the CPU uses to refer to the current instruction that is being executed [3][4].

In register-based computing, there are two different competing architecture types, Reduced Instruction Set Computers (RISC) and CISC (Complex Instruction Set Computers). The RISC design uses the idea that a simple instruction set enables a simpler microprocessor design, which can then run more instructions per instruction cycle as a result. There is no formal definition of what consitutes a RISC processor, but the general trend is that they have a single instruction per type of operation, e.g. only having '`load`' and '`store`' for memory accesses, rather than another instruction being able to load or store to memory. Examples of RISC architectures include ARM, PowerPC and SPARC. In recent years, the most powerful supercomputers have also taken to using RISC processors, such as the 'K Computer' which made use of a processor from the SPARC family [5].

CISC designed processors are opposite in this regard, providing instructions that can execute many operations within its execution cycle, sometimes taking several cycles of the CPU to complete. For example, an '`add`' instruction may be able to read from memory, perform the addition, and write back to memory automatically. There are often several instructions provided that implement the same operation, but take different

types of operands (i.e. registers, constants or memory addresses). Examples of CISC architectures include IBM's System/360 and related mainframe architectures, the Zilog Z80 and Intel's now ubiquitous x86 series of processors [6].

2.1.2 Stack machines

The use of stacks as part of computation has been around almost as long as computing itself, with Zuse's Z4 making use of them for subroutines in 1945 [7], and the Burroughs B5000 and its later models were widely used in the 1960s as well [8]. At the time, stack machines were considered to be the fastest computer architecture with the simple instruction sets afforded by the stack greatly simplifying compiler design. However, by the 1970s it had fallen out of favour [9], with improvements in compiler design making memory-to-memory architectures (or register based) more appealing. A common criticism was that stack architectures necessitated the use of memory locations as imitations of registers, which slowed the architectures down.

Instead of having named registers to store values as part of computations, stack machines use a stack data structure to store the computationally active variables. A stack is a so-called 'last in, first out' (LI-FO) data structure, meaning that the data that was last put into the structure is the first to be pulled out again, usually with push and pop operations, which place data onto the stack and take it off again, respectively. This is compared to a data structure such as a queue, which is a 'first in, first out' structure. Stack machines generally still have a couple of registers, although not often anything much more than a program counter (for storing current position in the program) and a flags register (for storing state about the last instruction for branching).

In the present day, the use of stacks in computer architectures and instruction sets is nearly exclusively limited to stack frames, to give the ability to do context 'saving' and 'loading' with function calls, in favour of register-based computation. However, there is a notable exception — the Java Virtual Machine (JVM) uses a stack-based bytecode as its underlying assembly language and virtual machine design [10].

2.2 Stack optimisations

Because of the popularity of register architectures, higher-level languages tend to get designed with them particularly in mind. This means that the majority of optimisation research for these languages gets done only for register architectures. This in turn makes stack architectures less appealing in somewhat of a spiral, making stack machines less and less appealing for additional usage.

Despite this, there has been significant work in the area of compiling these 'conventional' languages to a stack executation model. One of the main areas of research has been focused on fixing the repeated loading and storing to the same memory location that the naïve compilation model tends to produce.

Peephole optimisations are a method that can be applied to register and stack architectures alike, with the effect of removing redundant instructions from compiled code.

McKeeman [11] gave this name to this optimisation technique as it involves looking at small sections of the generated object code, identifying inefficient instruction segments and modifying them accordingly to generate more efficient or shorter code. McKeeman used ALGOL to give an example where an assignment is followed by an addition.

```
                    Compiled code:

                    LDA Y ; load the accumulator from Y
X := Y;             STA X ; store the accumulator in X
Z := X + Z;         LDA X ; load the accumulator from X
                    ADD Z ; add the contents of Z
                    STA Z ; store the accumulator in Z
```

As long as the store (`STA X`) operation is nondestructive, the subsequent `LDA X` is unnecessary and can be removed. Peephole optimisations such as this one generally make assumptions based on the specific architecture about which instructions are expensive and should be replaced. For example, for stack machines a '`DUP`' on the stack is generally much faster than a '`LOAD`' from memory. Another common peephole optimisation is 'constant collapsing', where the compiler can compute the result of an arithmetic operation between two or more constants, saving time calculating it at runtime. However, peephole optimisers have to be careful not to optimise away behaviour that can be reached elsewhere, for example with a label statement, which other code elsewhere may be relying on.

One major area of interest of stack optimisations is the concept of stack scheduling, i.e. separating the code into blocks and optimising them by removing extra memory accesses so that they can be executed concurrently. Baker [12] applied the concept to real-time systems, which have deadlines with which they have to have completed their computation by, which has potentially disastrous results if they do not.

Koopman [13] describes a method of optimisation of (generated) stack code that is able to remove 91% to 100% of redundant local variable accesses. This is achieved in four optimisation steps, in addition to the input processing, basic register to stack conversion that begin the process and the code generation that ends it. The code conversion step uses local stack variables to replace registers, and also takes note of the parts of the code which are set and then never reused, marking them as 'DEAD'. Next is a 'code cleanup', which makes more use of the stack, converting from the local variables. This is then followed by a preliminary peephole optimisation. Then stack scheduling is done, which uses an 'intra-block' algorithm on each block (defined as code sequences that contain no labels or branches) of stack code which tries to remove memory loads and stores from the program by maintaining a copy on the stack, similar to 'register scheduling' for register architectures. This is finished with another peephole optimisation pass to clean up some redundancy produced by the stack scheduling. The end result is a quite significant optimisation of generated stack code.

Following on from this, Bailey [14] went further and described an algorithm that can optimise memory accesses across block boundaries. This algorithm works out which memory locations are used in adjacent 'parent-child' (i.e. loops) blocks and duplicating

them on the stack, rather than doing the expensive store and load operations. In particular, this algorithm is able to optimise loop variables out of memory and onto the stack, greatly reducing the number of memory accesses for highly used loops.

Further work still was done by Shannon [15], who implemented a C compiler for an abstract stack machine. He implemented both Koopman and Bailey's algorithms, and came up with his own 'global register' optimisations. These are an iteration on top of Bailey's algorithm in that it analyses the data-flow of the program (rather than just a region) and works out which registers are used where, pushing that data to the stack where possible. Shannon's results also showed how different memory models can affect the performance of (optimised) stack code by varying the costs of different instructions.

2.3 Binary Translation

Gschwind et. al. [16] described the design issues they experienced when implementing a binary translator for IBM's ESA/390 architecture, for their mainframes, to a VLIW processor to improve performance. The primary difficulties experienced were due to the source CISC architecture's nature and what it allowed, notably with self-modifying code, and preserving behaviour with the atomic nature of instructions in interrupts and with memory access reordering (for performance purposes). These issues were solved with a variety of methods, for example solving the interrupt preciseness by testing early for the exception conditions before modifying and translating the code. They were also able to do optimisations as part of the translation by identifying spurious conditions with a compilation method based on 'deferred materialisation'. From there, they were able to use their work to begin designing a common VLIW architecture for various other architectures, including PowerPC and even the Java Virtual Machine. They suggest that this work could eventually result in a common base VLIW architecture that all (large) computers use where traditional computer architectures are just software layers on top of it.

The Transputer [17] is a RISC style computer that was developed by INMOS to provide a (commercial) computer architecture that was easy to program and engineer in, while still giving 'maximum' performance and giving a high degree of scalability. The Occam programming language, which was based on the CSP process calculus, was designed for this architecture. The idea is that with the Transputer's architecture and the Occam language, it would be very easy to write programs that would scale to systems with potentially thousands of processors and still make good use of all of them through use of concurrency by using the Occam language. While the Transputer was not a success overall, it did find use in satellites, as the highly parallel nature of the processors was good for redundancy, which is important for long-lived satellites with no possibility of repair, and also for use in error correction [18].

2.3.1 Transmeta Crusoe

In 2000, Transmeta[19] published a paper detailing their new Crusoe microprocessor. This was a VLIW (Very Long Instruction Word) processor — an architecture type designed to be used for instruction-level parallelism, where instructions can be executed concurrently without the extra complexity required by other processor designs. Using this processor, Transmeta was able to completely implement the x86 instruction set and its registers in software running on the processor. Because this was implemented in software, acting as a layer between the hardware and the programs running on it, it made it possible to completely change the internal workings of the translation software and the underlying hardware, all invisibly to the x86 code running on the processor.

Transmeta's translation software ('Code Morphing Software' or CMS) was structured as a dynamic translator in that it first did sequential interpretation of the x86 instructions, being careful to replicate all side-effects such as interrupts and memory access ordering. While doing this, it collected information on the instructions running on it, for example how often particular instructions were executed, what IO operations are performed, and which direction branches took. This monitoring allowed the CMS to selectively take often-run sections of x86 code and translate them into native VLIW code, which was then stored in a 'translation cache' for future lookup and use, as seen in the flow chart in Figure 2.2.

The novel approach here was the Translation Threshold — code sections were only translated into native code when they hit a certain execution threshold. This prevented too much time being spent on translating, which was an expensive operation, instead of executing the code. Programs often only have a few bottlenecks in its code; by identifying the commonly run code, the Transmeta CMS could remove these bottlenecks.

Transmeta also built several error handling routines into their processor, which a combination of hardware and software being used to detect and handle various forms of failing assumptions, e.g. assuming that two code sections reference non-overlapping memory. In the case of infrequent failures the CMS switched to the x86 interpreter rather than the translated code which, while slower, can be guaranteed to be correct.

However, Transmeta quickly ran into issues with the Crusoe processor, with slow performance in real world tests and even accusations of cheating benchmarks to improve their perceived performance [20]. While the processor was able to emulate enough of the x86 instruction set to run Microsoft Windows and several applications including web browsers, it often crashed and weaknesses in the processor itself and how its components were connected (notably a complete lack of an Advanced Graphics Port for 3D performance). Investigations indicated that the underlying processor was actually being very underused by the CMS in actual applications with the CMS's component 'molecules' (a 128-bit instruction word) only hampering its performance. Even in synthetic benchmark programs the processor's performance only rarely approached similar speeds of processors contemporary to it such as the Intel Pentium III or the AMD Athlon XP. These performance issues, among others, were largely blamed on the very large (for the time) 128-byte L2 CPU cache, meaning that it both negatively affected performance on code high in branching, as it had to pull in a full 128-bytes even if it only required a

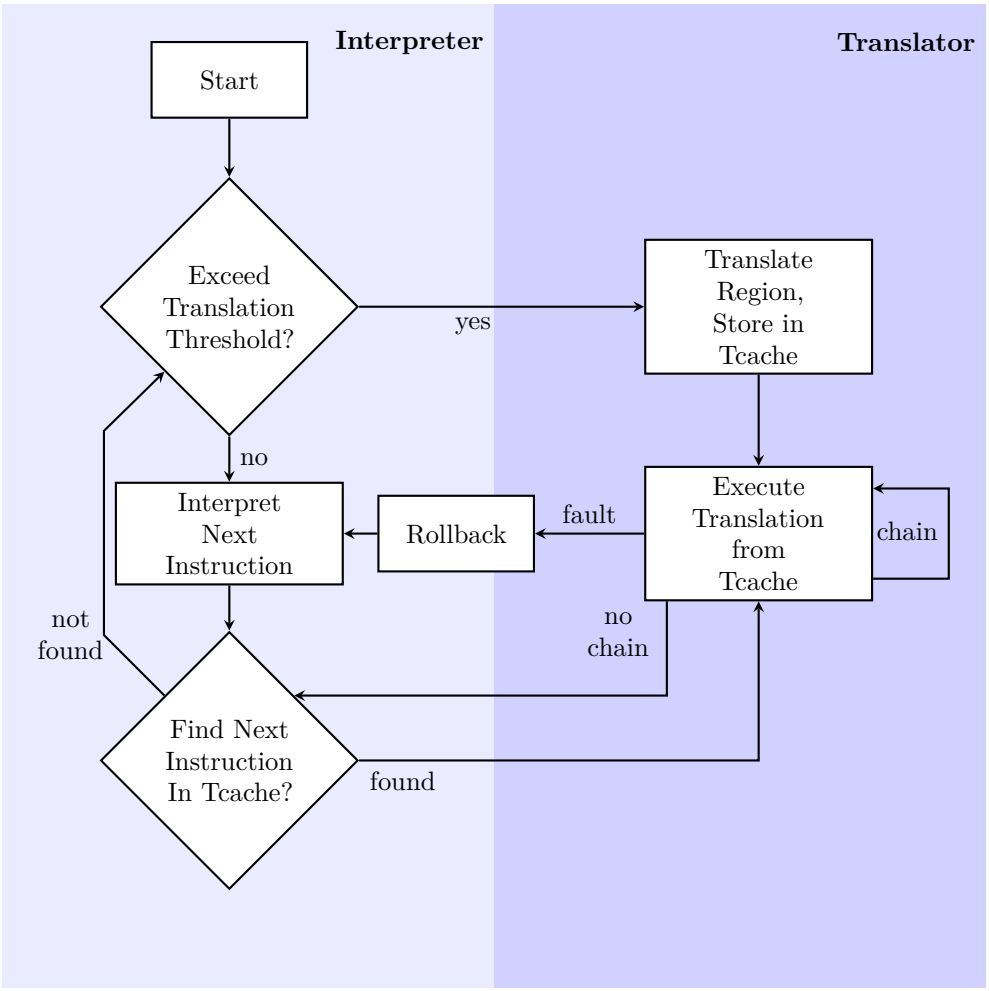


Figure 2.2: Typical CMS Control Flow [19]

Figure 2.3: The Transmeta TM5600 CPU from a Fujitsu laptop

single byte, and also was not enough cache to hold both the CMS and the translated code, causing excessive memory thrashing.

Attempting to address these concerns, Transmeta developed a new processor, the Efficeon, that addressed many of the concerned raised about the Crusoe, but even with its 256-bit wide instruction word, die size twice as large as its predecessor and doubled clock speed, it still had performance issues compared to contemporary processors and even often thermal throttling itself in benchmarking, and die size significantly increasing the cost of the processor, so failed to make any significant impact in the computing world [21].

Chapter 3

Problem Analysis

3.1 Summary of approach

The objective of this project is to create a program that can run register-based assembly instructions on an emulated stack machine and implement optimisation techniques that show improved execution times over the basic implementation.

The program implementation is broken down into a number of steps. The first part of the implementation is to take the chosen architectures and implement emulators for them. Some conversion routines for register to stack architectures are then implemented. After this, optimisation techniques are implemented and applied to the generated stack. Finally, suitable test programs are written.

3.2 Requirements

The following is a collected list of key features for the implementation of the project that are required for the project to properly succeed:

R1 Fully functioning emulator for a register machine architecture

- R1.1 Working parser
- R1.2 Implement basic interpreter
- R1.3 Implement all instructions

R2 Fully functioning emulator for a stack machine architecture

- R2.1 Working parser
- R2.2 Implement basic interpreter
- R2.3 Implement all instructions

R3 Ability to generate stack code from register code

- R3.1 Implement conversions for all register instructions
- R3.2 Transpiler — translate entire program and output the result
- R3.3 Translate code in blocks at a time

R4 Apply optimisations to the generated stack code

- R4.1 Peephole optimisations [11]
- R4.2 Koopman-style stack scheduling optimisations [13]

R5 Series of results that show significance of optimisation of generated stack code

- R5.1 Compare number of register instructions with the number of generated stack instructions
- R5.2 Show reduction in instruction count of optimised stack code
- R5.3 Show reduction in memory accesses of generated stack code after optimisations

There are also a certain number of desirable features that are optional for the project.

R6 Additional stack optimisations on the generated stack code

- R6.1 Bailey's optimisation algorithm [14]
- R6.2 Shannon's optisation algorithm [15]

R7 Implement translation block caching [19]

- R7.1 Cache converted blocks
- R7.2 Implement a translation threshold — like the Transmeta implementation, only translate regions of code if they are executed more than a certain number of times

R8 Ability to more easily visualise differences between handwritten and generated code, e.g. graphs or running side-by-side

The degree to which these objectives are successful is discussed in Section 5.3 and further work according to which points are not completed is discussed in Section 6.1.

3.3 Architectural choices

Prior to any programming, the programming language used has to be decided on, as do the source (register) and target (stack) architectures.

The decision made to write emulators for the register and stack machines chosen. Toolchains for 'obscure' architectures, such as the ones likely to be chosen, tend to be rather limited in nature and either out-of-date, broken or both. Writing emulators for the source and target architectures creates extra programming work, but means that the choices of programming language and style are much greater.

It was decided that to simplify the design, the program would interpret assembly source code of the register and stack architectures instead of running the compiled binary. Doing it this way is only an abstraction over actually reading the binaries, and removes the step of having to compile the source code and then decode it again. It would not be difficult to make a program that does this but was deemed unnecessary for the core part of this project, to convert a register-based instruction set to a stack-based one.

3.3.1 Programming language

The advantages and disadvantages of several programming languages were evaluated for the project.

C

C is one of the oldest programming languages in the world and is only just a step above the 'raw' assembly level. Despite the age of the language, its low-level nature makes it extremely fast, often being the fastest of all languages, as it has over 40 years of compiler theory and optimisations backing it up. With the emulators written in C there would almost no concern for their speed. However, it does mean that the programmer has to do everything themselves, such as handle memory management, and it has very little in the way of type system which makes it easy to make mistakes in and slows down development of features.

C++

C++ is a systems programming language that is well known for its hugely powerful templates for generic programming, its ability to use multiparadigm styles of programming and its high speed, with significant work being put into optimising compilers for the language. Using object-orientated programming, the conversion routines can be built on top of one of the emulators, with little duplication of code.

The language's power also comes with some disadvantage however. It makes it easy to make errors in programming and logic, and its compiler errors for templates are infamous for taking up many pages. In fact, C++'s own creator, Bjarne Stroustrup, has been quoted as saying "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off" when comparing C++ to its parent language C. That said, modern C++ has been making big improvements to the language since 2011, with smart pointers, lambdas & type deduction fixing many of the language's major criticisms and making it almost unnecessary to ever use manage the program's memory manually.

Rust

Rust is a relatively new systems language that takes inspirations from C++ and Haskell, with its main selling point being guaranteed memory safety and 'zero-cost abstractions' which makes the language very fast to use and very difficult to break in the standard ways with segmentation faults and similar errors. This safety comes at a cost as it has a very steep learning curve and the compiler's 'borrow checker' (how it tracks where values in memory are used) is already gaining a reputation for being hard to deal with and make programs correct to the point that they will compile.

Python

Python is a higher-level language that has a syntax that allows concepts to be expressed quickly and easily, with resource management being handled by the language. Unlike the other languages discussed here, its type system is dynamic, meaning that variables can be reassigned to different types, which helps with the ease of programming in it. Also unlike the other languages here, it is an interpreted language (instead of compiled). While this has its advantages, mostly with the speed of prototyping and development, it has a big impact on the speed of the language by comparison.

Go

Go (often called Golang) is also a new systems programming language. It is strongly inspired by C, but adds several features that make it easier to use. The main feature is that the language is garbage collected, meaning no manual memory management. Go's garbage collector is highly sophisticated too, boasting often getting microsecond times for its 'Stop the World' phase. The language also includes several concurrency features that make it a lot easier to program parallel code, in particular for web services. Go's garbage collector is also a common criticism as systems programming often require manual memory management due to system restraints, which the language offers no ability to do so. Go's designers have also taken the decision to leave certain features out of the language such as generics and operator overloading, which leads to duplicated code.

Conclusion

Given that Go and Rust are still relatively immature languages, that Python would likely run into some program speed issues and that C requires manual memory management, it was therefore decided to use C++ as this would be a reasonable compromise between its speed and multi-paradigm programming ability, over the disadvantages of the risk of reduced development speed from fixing programming errors and the language's verbosity over a high-level language.

3.3.2 Architecture choices

Register architectures

A shortlist of register architectures was drawn up, of the Zilog Z80, the Xilinx Picoblaze, and the DCPU-16. The Z80 is a microprocessor originally introduced in 1976. It is 8-bit based with the capability to address 64KB of memory by means of combining its registers. An extremely popular processor during the 1980s, it is still used in embedded systems. It has its own assembly language that has both register and stack elements. The Picoblaze is a 'soft processor core', meaning that is not manufactured specially, rather is fabricated on an FPGA (field-programmable gate array) by the user. It uses an

8-bit RISC architecture, which makes it very simple to fabricate and run. The DCPU-16 is not a real processor, and has never been produced. It is the invention of Markus Persson in 2011 to be emulated as part of a game he was creating but never completed, however there was significant interest in it, which means that there was many programs and emulators produced for the DCPU-16 processor. It is a 16-bit processor that has modularity in mind in the architecture — the computer would 'connect' to peripherals to do its IO instead of a more common console output interface. Listings 3.1, 3.2 and 3.3 show examples of a "Hello World" program in the respective assembly languages of the Z80, Picoblaze and DCPU-16.

```
      LD a, 0
      LD (CURCOL), a
      LD (CURROW), a
      LD hl, text
      B_CALL(_PutS)
      RET
text:
      .db "Hello, world!", 0
```

Listing 3.1: Z80 ASM

```
module hello_world;

initial begin
  $display ("Hello, world!");
  #10 $finish;
end

endmodule
```

Listing 3.2: Picoblaze ASM

```
; Attach screen
SET A, 0
SET B, vram
HWI 0

SET J, 0

:loop
SET I, vram
ADD I, J
ADD [I], 0x2000
ADD J, 1
IFN J, 12
    SET PC, loop

; loop forever to end
:crash
SET PC, crash

:vram
DAT "Hello, world!", 0
```

Listing 3.3: DCPU-16 ASM

Stack architectures

In terms of stack architectures, Forth, Jasmin, and J5 were shortlisted. Forth is a very early programming language that dates from 1970, when stack machines were still

common in everyday computing. It is a very simple compiled language that still finds uses today in embedded systems due to how easy it is to implement in new systems, and its low memory overhead. Jasmin is a variant of JVM bytecode, meaning it is a text format rather than the binary files of raw JVM bytecode. J5 is a teaching stack language to introduce people to stack machines and their respective languages and it is very simple in that regard, drawing on ideas from various different actual stack languages. Examples of 'Hello World' programs in Forth, J5 and Jasmin can be seen in Listings 3.4, 3.5 and 3.6 respectively.

```
CR .( Hello, world!)
```

Listing 3.4: Forth ASM

```
OUT "Hello, world!"
```

Listing 3.5: J5 ASM

```
.class public HelloWorld
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
  .limit stack 3
  .limit locals 1
  getstatic      java/lang/System/out Ljava/io/PrintStream;
  ldc            "Hello World."
  invokevirtual  java/io/PrintStream/println(Ljava/lang/String;)V
  return
.end method
```

Listing 3.6: Jasmin ASM

Conclusions

As the Picoblaze is a soft-processor, it is separated into its ASM and Verilog components and the processor's separate components have to be connected together manually as part of the build process. The Picoblaze was therefore discounted as it was decided that this would be too much boilerplate. After some initial emulator implementation for the Z80 processor, its ability to combine registers proved difficult to implement in a good way in C++, so after some thought it was decided to continue with the DCPU-16 assembly language (specification included as Appendix A), after cutting it down to get rid of the peripherals part of the language which is not necessary for this project.

For the stack-based architectures, given Jasmin's complexity to write and Forth's complexity to implement, it was decided to target the J5 stack language (specification included as Appendix B) due to its simplicity compared to the other two languages.

Chapter 4

Design and Implementation

4.1 Emulator implementation

Both the register and stack emulators (plus the conversion 'machine') are written to be part of the same program, with the different functionality controlled with commandline flags. This method enables the emulators to share functionality where appropriate, especially for the conversion 'machine' which is built directly on top of the stack machine using object orientated programming techniques.

Early in the implementation, it was decided to cut down on the instructions that the emulators, as they were either esoteric or difficult to implement while being irrelevant to this project. In particular, none of the special opcodes for the DCPU-16 (see Appendix A) are implemented, and neither are the `PUSH` or `POP` instructions, as they are used to implement a 'reverse stack' starting at the top of the memory, 0xffff. Since this project tests converting register code to stack code, it does not make much sense to have most of a stack in the register architecture. This does exclude the possibility of having a call stack, with procedures or functions, but this is an acceptable compromise as they can be emulated by branching instructions instead.

Similarly, the J5 (see Appendix B) includes `CALL nn` & `RETURN` instructions, so for feature parity these were excluded as well. The J5 also has a `SSET n` function, which is intended as a fast version of `SET nn` for small numbers. Since this is an emulator, this is replaced with `SET` quite trivially. However, if it were included, it would be a good target for optimisation, as the operand is constant, it would be trivial to replace the `SET` instruction with a `SSET` in the appropriate places.

4.1.1 Algorithms used

The general structure for both emulators is fairly similar. First the input file is read into memory. It is then iterated over, line-by-line, and tokenised into an intermediary representation, which serves to validate the input syntax. This can be approximated with the following pseudocode for the register architecture:

```
function tokenise(filepath: string): instruction
  load file from filepath
  For line in file do
    instruction;
    split line into words # words separated by whitespace
    word = words[0]
    If word is a label # is prefixed by a ':'
      instruction.label = first word
      word++
    If word is not an opcode
      return error "not a valid instruction"
    instruction.code = word
    word++

    # opcodes take a variable number of operands
    If number of operands for instruction.code != remaining_words
      return error "invalid instruction" + instruction.opcode
    instruction.operands = remaining words
    return instruction
  end
```

Listing 4.1: Tokenising algorithm for the DCPU-16

Determining the type of the operands from the source proved difficult. C++ in particular made it difficult due to its type system. The eventual solution uses a variant (typesafe union) type to store integers (for literals), register references, and strings for everything else. This necessitated the use of a third-party library, Boost, as C++ does not have such a type in its standard library. Although it does gain one in C++17, the latest standard that as of 2017 is not yet fully included in a stable release of any of the common compilers. Boost is a large and complicated library, but if usage is limited to only required features it does not increase maintenance costs significantly. Using strings for the 'everything else' part of the operand proves quite complicated as it includes not only labels and string literals, but memory address lookups as well (e.g. `SET [2000], 42`). For executing purposes it is possible to check if the string starts and ends with `[` and `]` respectively, and getting the value of the operand inside. This is further complicated as the DCPU-16 has the ability to use compound expressions inside the memory address brackets, e.g. `[2000+I]` where the contents of the 'I' register is added to 2000 to get the memory address referred to. A similar solution is made for the stack emulator.

For running the emulators themselves, their class provides a `run` function, which is passed the tokenised program to run (along with any flags such as verbosity of output). It then initialises the machine's state and runs through each instruction in the program until terminated or it reaches the end.

```
procedure run(program: Program, flags..)
  set registers to 0
  program_counter = 0
  while program_counter < sizeof(program)
    run_instruction(program[program_counter])
    program_counter++
  end
```

Listing 4.2: Running a DCPU-16 program

Listing 4.2 is a simple view of the actual function though, which in reality also has the ability to run at a certain clock speed (to mimic the speed the actual processor would process instructions) or just as fast as the emulator can run, which is more useful for testing purposes. There is also a verbosity flag which dumps the state of the machine, for debugging purposes.

For each parsed instruction, a member function for the class of the relevant emulator has been written. To link each of these methods to their respective instruction, it is best to use a `std::map` as provided by the standard library. In many places, the functions are very short, only consisting of a single line of code, so the code makes use of C++'s lambdas. This means, for example, that the implementation for the J5's `INC` instruction only consists of the lambda `[](machine *m){m->stack.top()++;}` instead of referring to a function pointer such as `&machine::inc_func` in the class' function map, which avoids all the extra code overhead of having to fully declare an extra method. Where the instruction implementation functions were beyond a line, the usual function pointer style has been kept to help keep the code readable. Taking this, many of the instructions follow a similar pattern (for example, `ADD` and `SUB` from the J5 both operate on the top two items on the stack). As the code for these implementations would have otherwise been duplicated several times (where only the exact operation differs), a helper function has been created to deal with these. This results in the lambda `[](machine *m){m->binop_func([](int a, int b){return a+b;});}` which even though works as expected, it is decidedly unreadable for anyone looking through the code for themselves.

The function loop also has to deal with the effects of the conditional expressions. For the DCPU-16 this is the `IFx` statements where they only perform the next expression if the condition is true, whereas the J5 has some test instructions (e.g. `TEQ`) and `BRZERO` instruction which conditionally branches on whether the test instruction set the appropriate flag. The notable thing here is that the test instruction can be executed long before the branch is actually taken, which allows for more freedom over branching. It also has `BRANCH` for unconditionally branching.

4.2 Conversion routines

Initially, implementation of the conversion was to do it all at once — the program would take in a register program and output a stack program that could be run by the stack interpreter. This served well as an initial step, but to get to the point of caching blocks

```
cache: map positions to stack code;
function convert_instructions(start_pos: instruction)
  If start_pos not in cache
    last_pos = find next label
    cache[start_pos] = convert_instructions(start_pos, last_pos)
  return cache[start_pos]
```

Listing 4.3: Converted instruction caching

of code, the implementation was to be changed first to converting one instruction at a time and then executing it as the stack machine all at once, to converting blocks of instructions and executing them. The process for this can be seen in Listing 4.3

Table 4.1 is a table that shows some examples of the stack code produced for some register instructions. By way of explanation, for the `ADD A, B` row the code snippet first loads register A (represented by 8191) onto the stack, loading register B (8190) onto the stack, adding them together, and storing the result back into A. The DCPU-16's `IFx` statements are more complex to convert as the J5 does not have a direct equivalent. The `IFx` instruction only executes the next instruction if the condition (e.g. `B != A` for `IFN B, A`) evaluates to true. For the J5, the only conditional branching mechanism is `BRZERO` and a series of 'test' instructions, that have to be executed before hand and set a flag to signify whether its test is true or not. So, the conversion routine needs to load the operands for the statement onto the stack, test the operands on the stack, the drop the operands (so not to leave them 'dangling' on the stack regardless of the condition result), then `BRZERO` with an operand that is equal to the generated length of the next instruction. With regard to implementation, it was necessary to leave a temporary value for the operand initially, then go back and change it once the next instruction had been converted.

The implementation of this is particularly nice as the functions which load the instruction operands onto the stack (as in, e.g., `ADD B, A`) were able to use one another, with minimal code duplication. For example, loading a register value onto the stack is just the same as loading an address onto the stack but with an extra `LOAD` instruction following it. These segment functions allow operands to be loaded onto the stack with no additional side effects. A complicated example of this can be seen with the `SET A, [3000+I]` in Table 4.1 that uses the DCPU-16's ability to have arithmetic operations inside memory addresses.

Register instruction	Converted stack code

`SET A, 5000`	`SET 5000`
	`SET 8191`
	`STORE`

`ADD A, B`	`SET 8191`
	`LOAD`
	`SET 8190`
	`LOAD`
	`ADD`
	`SET 8191`
	`STORE`

`SET PC, LOOP`	`BRANCH LOOP`

`IFN A, 0`
`SET PC, LOOP`
	`SET 8191`
	`LOAD`
	`SET 0`
	`TEQ`
	`DROP`
	`DROP`
	`BRZERO 2`
	`BRANCH LOOP`

`SET A, [3000+I]`	`SET 3000`
	`SET 8185`
	`LOAD`
	`ADD`
	`LOAD`
	`SET 8191`
	`STORE`

Table 4.1: Instruction conversion examples

Original stack code	Optimised stack code	Notes
`SET 1` `ADD`	`INC`	Take advantage of the J5's `INC` instruction
`SET 1` `SUB`	`DEC`	Same as above, but for subtraction

`SET 0`
`TEQ`
`DROP`
	`TSZ`	Generated with an `IFN, x, 0` statement. Instead take advantage of the `TSZ` instruction of the J5

`SET x`
`STORE`
`SET x`
`LOAD`
	`DUP`
	`SET x`
	`STORE`
		A store followed by an immediate load of the same value. Instead, duplicate the stored value and just store (since it might be used elsewhere)

`DUP` `SWAP`	`DUP`	Removes redundant swap
`SWAP` `SWAP`	`<NOP>`	Removes redundant swaps
`SET x` `DROP`	`<NOP>`	Removes unused stack usage

Table 4.2: Peephole optimisation examples

4.3 Optimisation

Optimisation is done on the stack code generated by blocks of register instructions.

4.3.1 Peephole optimisation

A number of peephole optimisations are implemented as part of the optimisation process. A list of them can be seen in Table 4.2. While it has some effect on its own, typically around the 'edges' of generated stack code, it tends to have greater effect following a stack scheduling pass.

The optimiser iterates over the program (which is again separated into regions by label) in blocks of a certain size that is determined by the individual optimisation. When it finds a block that matches the pattern that it is looking for, it replaces that block as defined by the optimisation.

Original register code	Initial stack code	Stack scheduled code
	`SET 23`	
	`SET 8191`	`SET 23`
	`STORE`	`DUP`
	`SET 7`	`SET 8191`
	`SET 8189`	`STORE`
	`STORE`	`SET 7`
	`SET 8188`	`TUCK2`
	`LOAD`	`SET 8189`
	`SET 8191`	`STORE`
`; b = a + c`	`LOAD`	`SET 8188`
`SET A, 23`	`ADD`	`LOAD`
`SET C, 7`	`SET 8188`	`SWAP`
`ADD X, A`	`STORE`	`ADD`
`ADD X, C`	`SET 8188`	`DUP`
`SET B, X`	`LOAD`	`SET 8188`
	`SET 8189`	`STORE`
	`LOAD`	`ADD`
	`ADD`	`DUP`
	`SET 8188`	`SET 8188`
	`STORE`	`STORE`
	`SET 8188`	`SET 8190`
	`LOAD`	`STORE`
	`SET 8190`	
	`STORE`	

Table 4.3: Stack scheduling example

4.3.2 Stack scheduling

Stack scheduling is simple to implement and is described in pseudocode by Listing 4.4.

The key to which instruction needs to be inserted at the 'reuse' point to bring the inserted instruction to the top of the stack depends on how many values have been added by instructions between its store point and its reuse point. This can be determined statically as all instructions have a known behaviour (whether they remove or add values to the stack).

An example of what this algorithm produces is listed in Table 4.3. It is worth noting that even though the stack code produced by the stack scheduling is only three instructions shorter, it only contains six memory read/writes, compared to ten for the initial unoptimised stack code which is a 40% reduction.

Due to the naïvity of how the blocks are determined (splitting on labels), if a loop does not have any labels that segment the code that follows the end of the loop, it will also be included in the block. This causes issues with the stack-scheduling optimiser as it may find pairs of `STORE`/`LOAD` which are actually outside the block. For the purposes of this project, an (unused) label at the actual logical end of the block is added to solve the problem.

```
pairs = []

function get_stack_depth(pos)
  return number of pairs such that p.i < pos and pos < p.j

For ins = program.begin to program.end
  if ins == SET and ins + 1 == STORE
    For ins2 = ins + 2 to program.end
      if ins2 == SET and ins2 + 1 == LOAD
        # found a match
        pairs.append({ins, ins2})
        break ; don't include the same values more than once
    end
end
sort pairs by distance (j-i) # so we apply tightest loops first
For (i, j) in pairs
  stack_depth = get_stack_depth(i)
  if stack_depth > 2 or get_stack_depth(j) > 2
    continue ; too deep in the stack to bring back to the top

  if stack_depth == 0
    ins = DUP
  else if stack_depth == 1
    ins = TUCK2
  else if stack_depth == 2
    ins = TUCK3

  remove instructions at positions j and j+1
  insert ins before position i

  stack difference = net stack depth change between i and j
  if stack difference == 0
    # no op
  else if stack difference == 1
    insert SWAP after position j
  else if stack difference == 2
    insert RSD3 after position j

  For p in remaining pairs
    adjust i, j according to insertion/deletion change
  end
end
```

Listing 4.4: Stack scheduling implementation

4.4 Overall structure

The overall structure of how the translator performs is shown in Figure 4.1. The program is similar to Transmeta's CMS structure (as seen in Figure 2.2) with its translator and associated cache system along with how it iterates and refers to the translated code, but lacks the translation threshold and the ability to rollback and interpret the instruction in the register language. While this implementation can interpret the register code, it cannot switch between that and translating in a single run. This diagram also shows the control flow for the optimisation patterns which Transmeta's diagram lacks. Similarly to the Transmeta diagram, it also does not show the end condition of the Interpreter/Translator combination, when the program counter reaches the end of the program, or an unrecoverable fault occurs.

4.5 Test implementations

For getting results out, the programs have to be converted several times with various different commandline flags for the register vs stack conversion and for the different optimisation levels. For this a simple 'test harness' has been written in Python, which loops through all the programs, generating the correct flags for running them, and runs them, capturing their outputs. First check is to check the equality of the converted code output with the output of the standard register emulator. This check would not suffice if both emulators were incorrect, but in practice checking 'by eye' was satisfactory to cover that eventuality. Then the register code is converted with each level of optimisation and the output of which is captured and parsed. The emulator has been written in such a way that at a certain debug level it includes every instruction that has been run, or information about instruction caching. After some pattern matching on this output, numerical results are obtained.

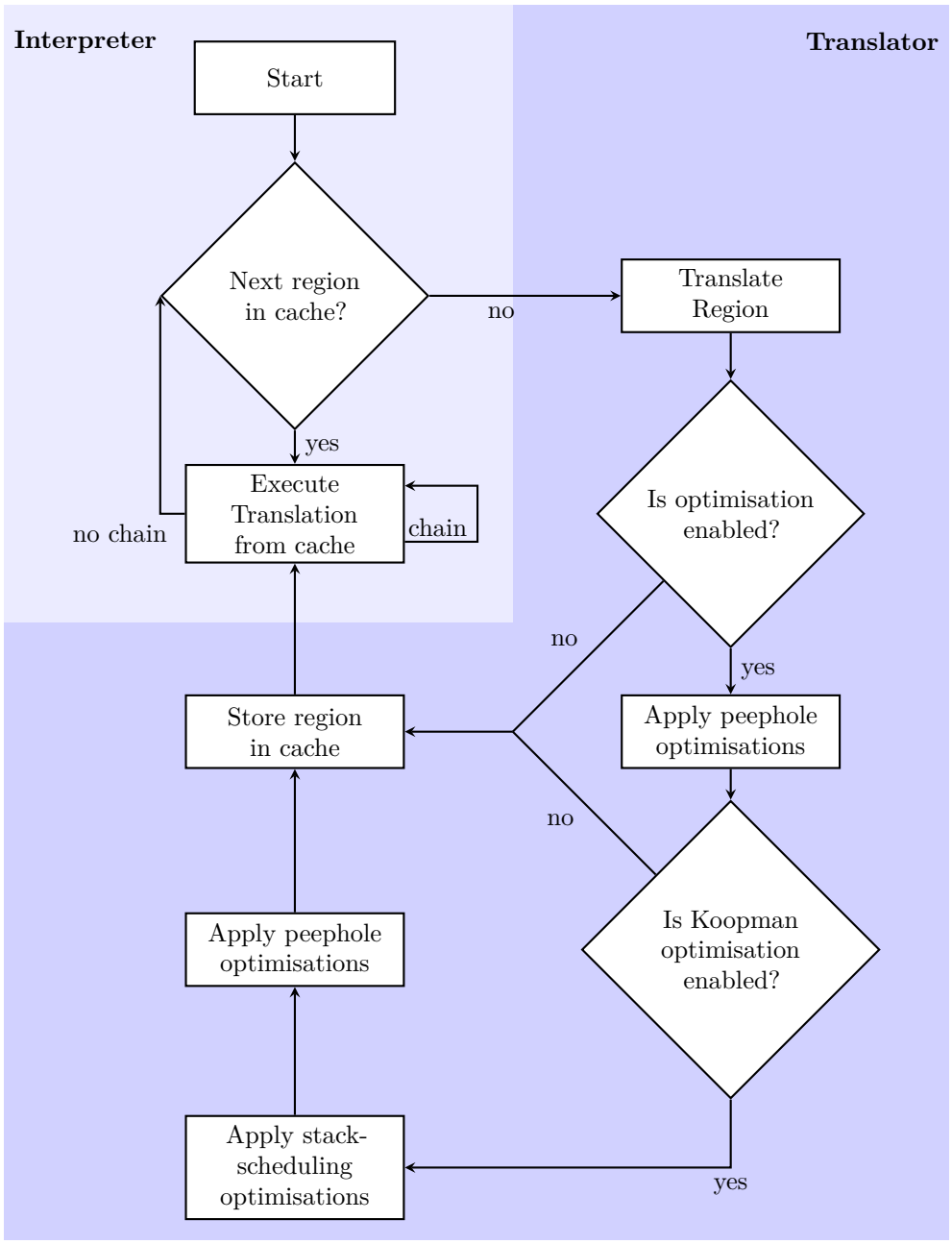


Figure 4.1: Implementation structure

Chapter 5

Testing, Results and Evaluation

5.1 Testing

Due to the emulated nature of the implementation of this project, testing focuses on the instructions of the original register-based programs and stack programs that they produce, at all levels of optimisation — no optimisations, peephole optimisations, and stack-scheduling.

The results look to find the relationship between the number of register instructions to generated instructions, so that the effect of each optimising level can be assessed. Further attention is paid to the number of memory accesses made by the stack program, as stack-scheduling is supposed to be able to significantly reduce redundant memory accesses produced by emulating registers as memory.

Using a simple memory model, a basic figure for 'program cost' can be produced. While there is no equivalent available for the source register-based programs, it is beneficial to see the difference between each optimisation level. The memory model is also used to measure the effect of the translation cache, by giving the conversion and caching of the register code an (estimated) high cost, to simulate the effects of not having to convert heavily-used loops many times over.

Testing is done using a series of programs handwritten for the DCPU-16 emulator. For testing purposes, simple test programs (called `simple`, `loop` and `redundant`) were created only to test compiler correctness and were not used for benchmarking.

Several simple benchmark programs have been written in the source DCPU-16 architecture for the purpose of showing the effectiveness of the optimisation algorithms. For testing purposes, simple test programs (called `simple`, `loop` and `redundant`) were created only to test compiler correctness and were not used for benchmarking.

For the actual benchmark programs, there are several desirable features to consider. Since stack-scheduling is an intra-block algorithm, the benchmark programs should vary between types of loops that are used, and their frequency of use. Highly used loops do well to test the translation cache system. The programs should also use a wide variety of register instructions to achieve better coverage of the translator and emulators.

A common set of tests used for these type of the programs is the Stanford Benchmark

	Base	Peephole	Koopman
`bsort`	24024	22343	22233
`fib20`	665	608	606
`primes`	6717	6619	6619
`tri100`	111381	106332	106332

Table 5.1: No. of instructions executed

Suite [22]. It is a set of programs written in C that are fairly short in code and execution time, meaning that development time is sped up by a quick test run. It also has the advantage of not requiring any input, which both the source DCPU-16 and J5 instruction sets lack. Examples of programs in it include bubble sort, Tower of Hanoi and a matrix multiplication.

However, since the implementation in this project lacks any sort of subroutine or function support, many of the programs would be quite difficult to feasibly implement. With that in mind, a different set of benchmarks have been created. Given more time, further benchmarks would have been created so as to produce more raw data, for better results.

- `bsort` — Bubble sort benchmark. Sorts 32 integers and prints the result
- `fib20` — Calculates and prints the first 20 fibonacci numbers
- `primes` — Uses a prime sieve to get all the prime numbers up to 100
- `tri100` — Outputs the first 100 triangle numbers, using the addition formula

The bubble sort program, `bsort`, is an implementation of bubble sort that sorts 32 16-bit numbers. It is one of the more complex programs with it having nested loops. A prime calculator, `primes`, uses a prime sieve to find all the prime numbers up to 100. Whilst it has similarly nested loops to `bsort`, it is implemented in a way such that the loops are logically separated out in the program, so that it branches a lot more than it would have done otherwise. A Fibonacci program, `fib20`, outputs the first 20 numbers in the Fibonacci sequence. This is the simplest benchmark program, only branching within a single loop. Similarly there is a triangle number calculator, `tri100`, that calculates the first 100 triangle numbers ($T_n = n(n+1)/2$). The contents of these programs can be seen in Appendix C.

5.2 Results

Tables 5.1 and 5.2 represent the number of instructions executed for a complete run of the benchmark programs, and the number of LOAD/STORE operations for each of them for the same run, respectively. Figures 5.1 and 5.2 graph the relative decrease in the number of instructions executed for each optimisation level, where no optimisations is the base level at one.

	Base	Peephole	Koopman
`bsort`	7792	6703	6065
`fib20`	234	196	175
`primes`	1912	1912	1643
`tri100`	35345	35345	30296

Table 5.2: No. of LOAD/STORE instructions executed

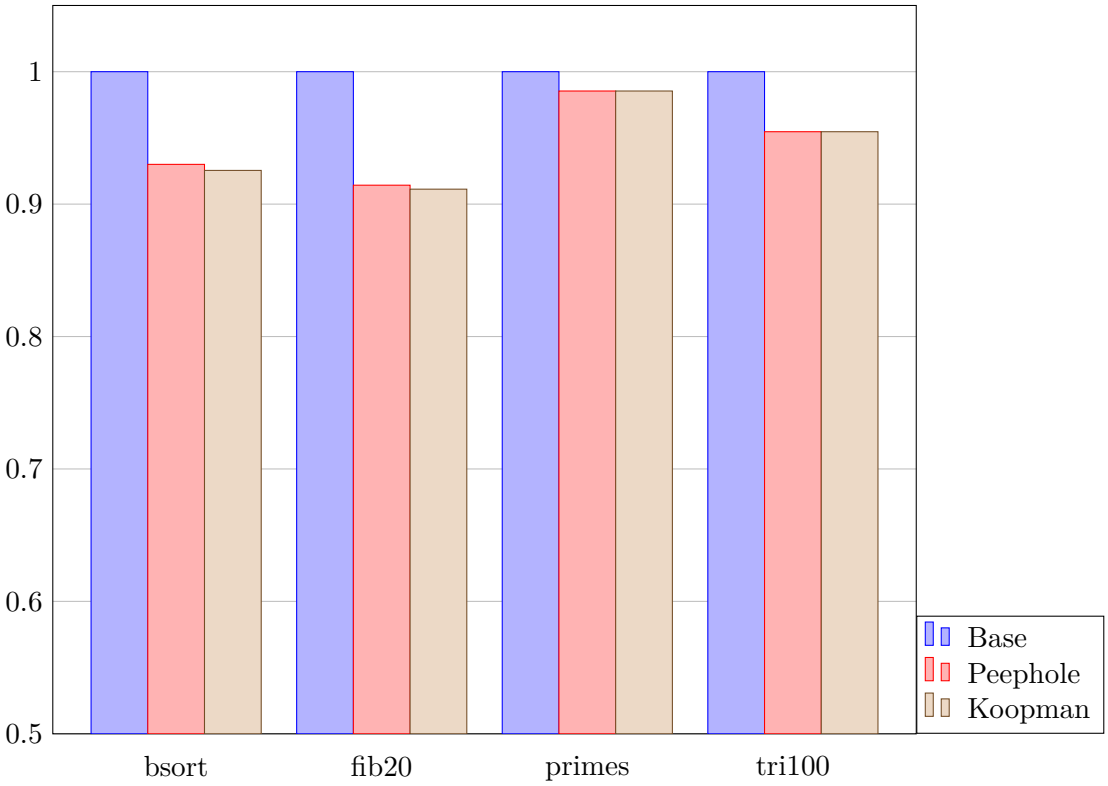


Figure 5.1: Relative no. of instructions executed

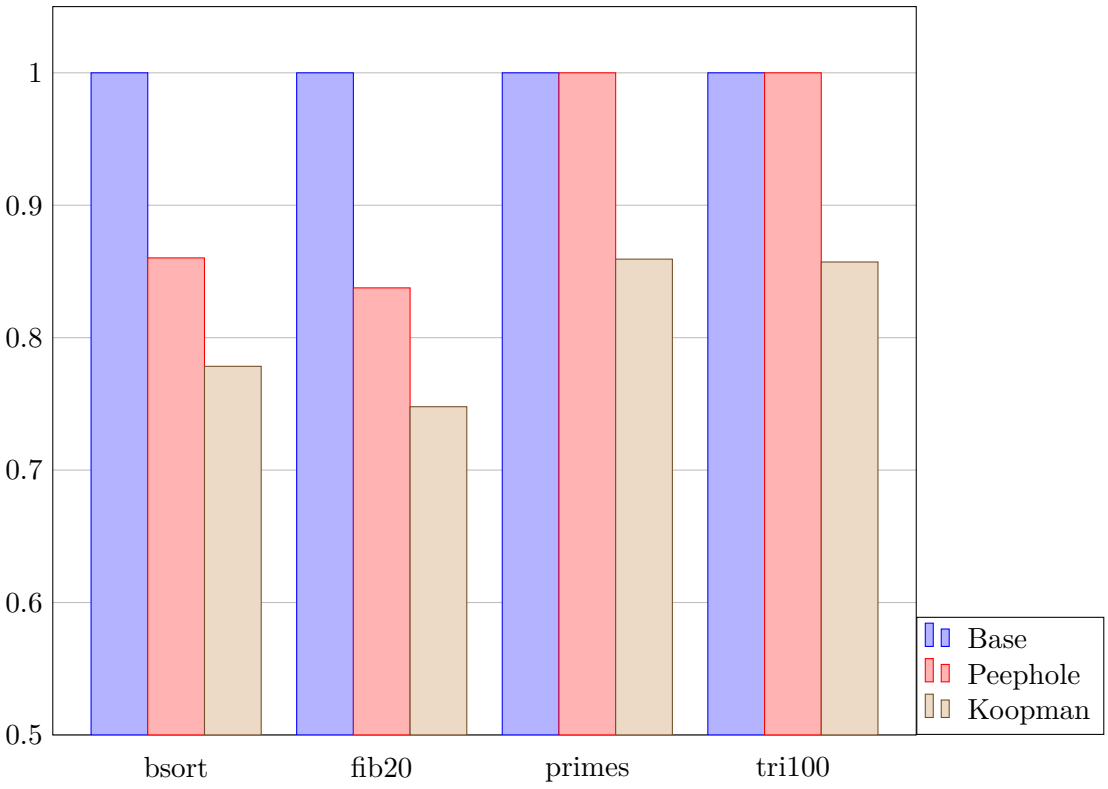


Figure 5.2: Relative no. of LOAD/STORE instructions executed

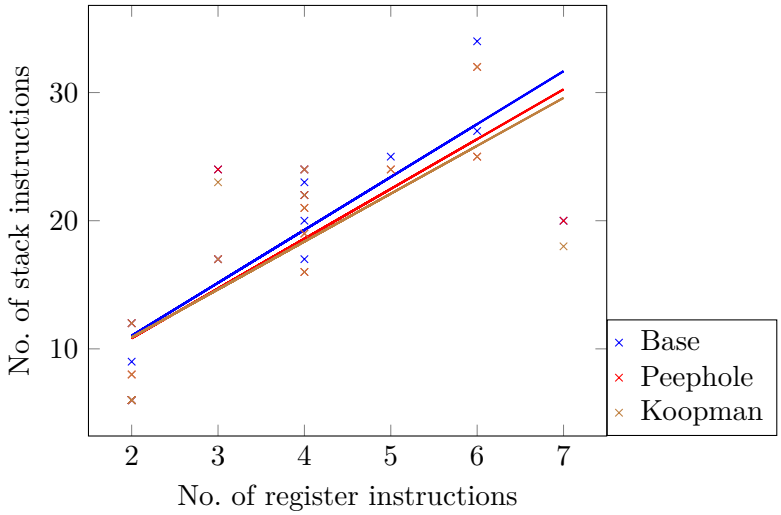


Figure 5.3: Graph of register blocks to stack equivalents

Program	Block	Register	Stack	Peephole	Koopman
bsort		34	103	102	102
	LOOP	2	9	8	8
	LOOP2	5	25	24	24
	SWAP	3	24	24	23
	POST	6	27	25	25
	RESL	4	22	21	21
fib20		6	34	32	32
	LOOP	7	20	20	18
primes		2	6	6	6
	LOOP	2	12	12	12
	SCANNED	4	17	16	16
	SCAN	3	17	17	17
	LOOP2	4	24	24	24
tri100		2	6	6	6
	LOOP	2	6	6	6
	LOOP2	4	20	19	19
	BREAK	4	23	22	22

Table 5.3: Instruction counts per block

Table 5.3 lists the number of stack instructions produced per block of register code, for different optimisation levels. The 'empty' block row represents the initial block of the program, with no label. It is graphed in Figure 5.3. Notably for the graph, the initialisation block for `bsort` is omitted as it is considerably larger than anything else and considered anomalous.

Table 5.4 lists the number of LOAD/STORE instructions produced per block of register code, for different optimisation levels. Similarly to previous tables, the 'empty' block row represents the initial block at the start of the program, given it has no label. The table is graphed in Figure 5.4 and again excludes the initialisation block for `bsort`. Note that the Tables and Figures compare against the total register instruction count, rather than the number of memory usages the register blocks have. This is because as registers are being emulated as memory positions, it would be difficult to reliably determine which register instructions either use or cause memory usages.

Tables 5.6 and 5.7 describe the 'costs' of running the benchmarking programs with the memory model described in Table 5.5. The two tables are represented in Figure 5.5.

Program	Block	Register	Stack	Peephole	Koopman
bsort		34	35	34	34
	LOOP	2	3	3	3
	LOOP2	5	8	7	6
	SWAP	3	10	10	9
	POST	6	6	5	5
	RESL	4	6	6	5
fib20		6	6	6	4
	LOOP	7	12	11	9
primes		2	2	6	2
	LOOP	2	2	2	2
	SCANNED	4	4	5	3
	SCAN	3	5	4	5
	LOOP2	4	7	7	6
tri100		2	2	2	2
	LOOP	2	2	2	2
	LOOP2	4	7	7	6
	BREAK	4	5	5	4

Table 5.4: LOAD/STORE instruction counts per block

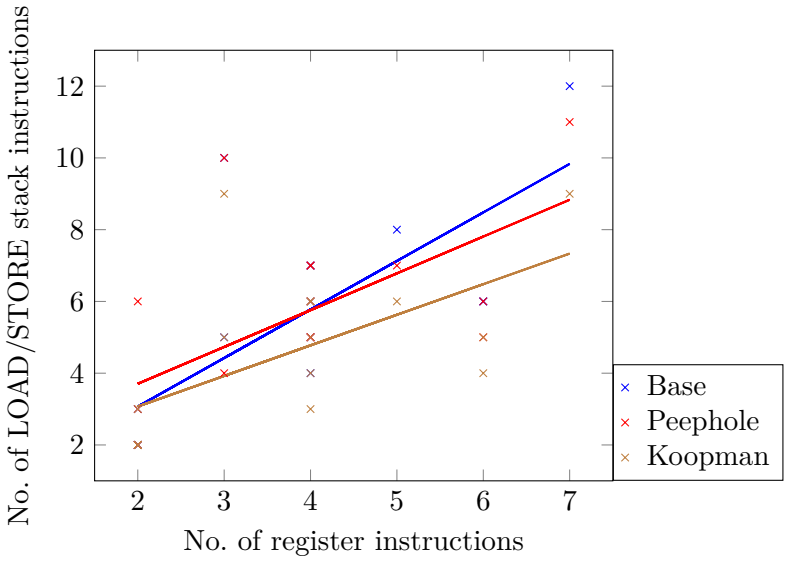


Figure 5.4: Graph of register blocks to stack memory read/writes

Caching a block	10 × number of register instructions
Retrieving a block	1
Memory accesses	3
Branches	2
Other	1

Table 5.5: Memory model used for estimating program cost

	Base	Peephole	Koopman
`bsort`	97292	93746	92454
`fib20`	1300	1224	1142
`primes`	19615	19516	18978
`tri100`	202089	197040	186942

Table 5.6: Program cost with caching disabled

	Base	Peephole	Koopman
`bsort`	40902	37356	36064
`fib20`	1300	1224	1142
`primes`	11981	11882	11344
`tri100`	192583	187534	177436

Table 5.7: Program cost with caching

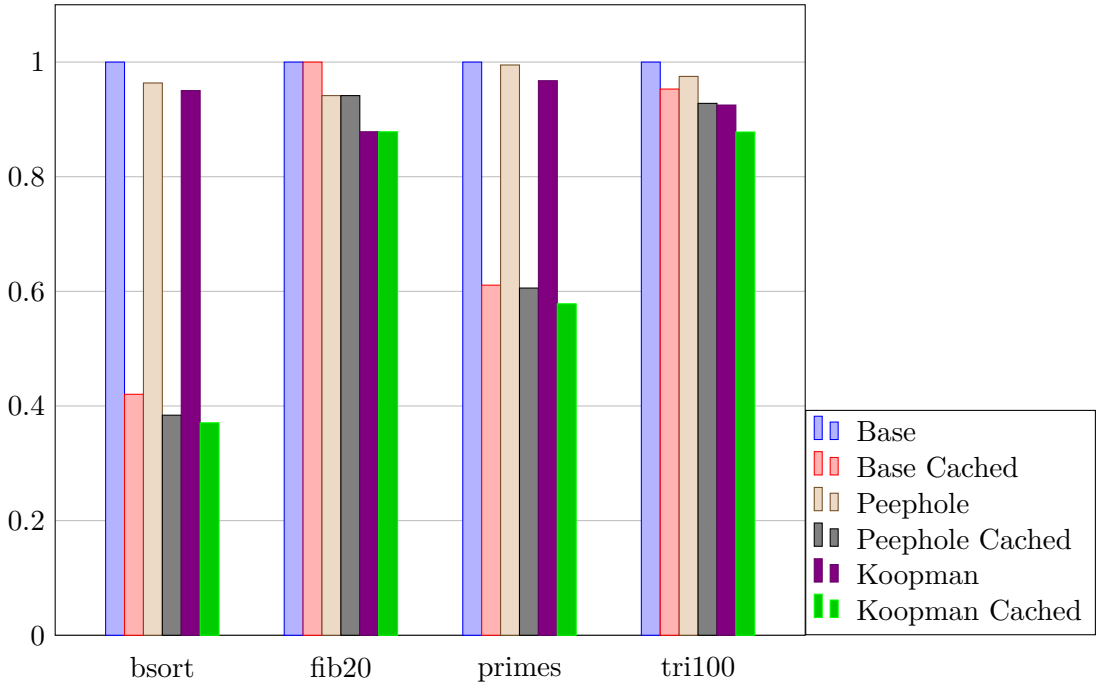


Figure 5.5: Relative program cost with and without caching

5.3 Evaluation

The basic results (seen in Table 5.1 and Figure 5.1) are somewhat counter-intuitive, showing that the peephole optimisation stage does the bulk of the the overall executed instruction count reduction and that Koopman-style stack scheduling has very little effect.However, looking at the number of memory reads and writes (with the '`LOAD`' and '`STORE`' instructions respectively) in Table 5.2 (and visualised in Figure 5.2) shows a different result, with stack scheduling doing as much reduction as peephole optimisation if not more, particularly in the case of the `primes` and `tri100` programs which have no instruction decrease with stack scheduling, but a significant decrease in memory usage.

While the optimisation results are also good (around 30% reduction in memory usage is quite substantial), making it appear as though they are not the same as what Koopman [13] achieved in his original paper, being able to remove 90–100% of redundant local variable accesses. The results here are not exactly the same measurement, but it seems reasonable to assume them to be comparable. The actual number of redundant memory accesses is quite hard to programatically calculate, and Koopman did it by studying the program's output by hand. A reasonable explanation is that the limited set of benchmark programs prevents a statistically significant result and other benchmark programs would have more opportunities for stack scheduling than the programs presented here. The programs used here have fairly short loops or regions (in number of instructions) naturally making lots of reuse of variables less likely than otherwise.

Looking at Table 5.3 and its linked Figure 5.3, it can be seen (per block) that while stack-scheduling does have an effect on instruction count, it is the preliminary peephole optimisations that have the majority of the effect. Despite this, looking at Table 5.4 and Figure 5.4 again shows where stack-scheduling becomes relevant and useful with the reduction in memory usage, where the linear regression line shows significantly lower average memory usage than peephole and no optimisations. The linear regression line for peephole optimisations starts above the base stack output, this is likely due to the small size of the data set — overall the line's gradient is shallower than the base output, implying that the results would continue to improve for larger block sizes.

Figures 5.3 and 5.4 also show the relationship of register instructions to stack instructions quite well, with one register instruction generating between 4 and 5 stack instructions depending on the optimisation level. The memory accesses are even more interesting, with Koopman's stack scheduling getting the ratio of register instructions to memory accesses down to just over one. It is reasonable to conclude then that the stack-scheduling does in fact remove many of the unnecessary memory accesses. Contrary to the conclusions of Figures 5.1 and 5.2 as presented above, this seems to go much further to validate Koopman's claim of near total removal of redundant memory accesses as it shows that register machines are not inherently any better than stack machines in terms of the overhead of fetching data from memory or registers compared to memory or the stack.

The effects of the instruction snippet cache are more impressive. As seen Figure 5.5 (with raw data in Tables 5.6 and 5.7), each optimisation level decreases cost in a sim-

ilar fashion to the raw instruction counts discussed earlier. However, caching these instruction blocks leads to some significant decreases in program cost for highly looping programs, like `bsort` and `primes`. Caching notably has no effect on `fib20` and hardly any on `tri100` — this is down to the simplicity of the programs involved as they make very little use of branching to change to a different block, only branching within the block is used. While the costs of each instruction are only estimates (Table 5.5), it shows the potential effectiveness of the instruction caching well. The estimated cost of converting register instructions is high compared to reality, given the model's unconditional cost of instruction translation (at 10x number of instructions) which would vary more in practice, so further improvements may be available depending on its actual efficiency.

Overall, this is a very positive result, as while the generated stack programs a lot larger in instruction count, the memory usages are approximately equivalent. Which when taking into account how a stack machine would be simpler than a similar register machine and so would be easier to construct and potentially run at a higher clock speed, making the machines even more equivalent in program execution cost. A reasonable continuation of this would be to alter the stack assembly language to help remove some common repeated instruction groups. For example, for the J5 assembly language, it would be useful to have a `LOAD` instruction that actually took a memory address, and similarly for the `STORE`, to save having to hardcode a separate `SET` instruction to put a value onto the stack for each instruction. A further advanced step would be to vary the produced stack code depending on its contents, i.e. if a particular group of register instructions could be replaced with something simpler than the 'standard' generated code. A simple example of that in this project is the peephole optimisation done on converting `SET 1` and `ADD` instructions to an `INC` instruction. It is not done at the initial translation and the register instruction set lacks a similar instruction.

5.3.1 Completion of requirements

All key features of the project have been completed, but only one of the desirable features has been able to be partially implemented, largely due to time constraints.

Requirements R1 and R2 (implementing the emulators) have both been fully implemented, with working parsers for their respective assembly languages and able to interpret and execute any program in that language, with a fully implemented instruction set. Code generation is also fully implemented (Requirement R3) that was free of side effects. The two key optimisation methods (Requirement R4) have also been implemented, including commandline flags to enable/disable them as required (similarly to optimise flags in C++ compilers) These flags have been used to get several sets of results (Requirement R5) focusing on length of generated code with each optimisation level and the amount of memory accesses.

Unfortunately, the additional requirements (notably Requirements R6 and R8) have not been able to be implemented, largely due to time constraints on the project. While the tool does have various debugging levels with which to get output out of, the interface is rather crude and the data output needs some transformations (using a Python testing interface and `grep`) to get meaningful results out of, so it would be preferrable to have

a user interface for collecting data from. However, there has been partial completion of Requirement R7, in that each converted block is cached, but there is no concept of a 'threshold' at which to convert them. This would have been useful as the expense of converting and caching the initalisation routines (that set all the input data) is likely too much compared to just running them, bringing up the 'program cost' of the program as discussed in the results above. It would be preferable to only convert and cache loops that are executed lots of times.

5.3.2 Program correctness

No significant automated testing of the implementation has been done. Instead the program's output is only verified 'by eye' and for the test run the multiple executions are compared against each other, to ensure consistent results.

5.3.3 Code quality

The code itself should be extensible for future work. It is published as open-source software at `github.com/LordAro/reg2stack` enabling any future work to be able to be built on top of this software. The peephole optimisation implementation allows for easy additions — just add the relevant function and add the function name to the existing list of functions.

Documentation of the code could be improved. While the complicated sections of code generally have comments explaining exactly why it does what it does, it is by no means complete and more code could benefit from comments. Many functions are also missing 'doc-comments', which generally take the form of comments above the function definition which can then be parsed by some external tool to generate more formal documentation. If this were expanded, it would be really helpful to future work on the program.

Chapter 6

Conclusions

This project set out to show that it was possible to get performance improvements out of stack machine code generated from code intended for a register-based architecture. The results for this project show a reduction in memory accesses from the base translation after applying optimisations and decreases in program cost using conversion block caching. The implementation is also designed in such a way that allows for further optimisations and improvements to be made to the project and the software has been released on GitHub to facilitate this.

All the project's requirements as defined in Problem Analysis (Chapter 3) have been evaluated (Section 5.3) and it has been shown that all key requirements were completed with partial completion of one of the optional requirements. Both register architectures have "feature complete" emulators that are able to parse and run their respective assembly code. Stack code is able to be generated from each register instruction, with the ability to translate whole programs, one block at a time. Through the use of command-line flags, two different optimisation levels can be used — peephole & stack-scheduling — which reduce both instruction count and the number of memory accesses compared to the unoptimised output. Finally each converted block is able to be cached, saving effort on converting it multiple times when looped over.

The results of the tests vary depending on the program, but they generally result approximately in a 10% decrease in raw stack instruction count and up to 25% decrease in number of memory accesses, importantly showing that stack machines are not inherently less efficient at dealing with data accesses than register machines. More significant and varied results are seen with conversion block caching, with results varying from no improvement at all to up to 60% decrease in relative program cost based on the estimated memory model.

In conclusion, this project is able to show that there were several improvements to be found in stack code generated from code intended for a register machine.

6.1 Further work

There are several further steps that could be carried out to extend the work of this project.

First and foremost would be to implement further optimisations, such as the inter-block algorithm by Bailey [14] and the refinements done by Shannon [15] which show potential for further improvements especially with the tight loops of the programs tested in this project. These optimisations would only improve the generated stack code even further and its relative performance to register machines. Beyond that, it is unlikely that all types of optimisation for this type of stack code have been found and further algorithmic improvements are possible. As such, it is possible that further improvements, with the right stack machine, may show stack machines as the more efficient type of processor.

There is also further scope for improving the instruction block caching system. Other than improving how blocks are divided up, making use of Transmeta's 'translation threshold' method could prove more effective for reducing the relative cost of running a program. Further work could also be put into using a more accurate memory model, as the results of the program cost (Figure 5.5) from the one used in this project are only drawn from an approximation taken from a flat memory model and an estimated cost for translation. Transmeta ran into difficulties with their VLIW RISC-style processor, with their CMS being under utilised in all but specific benchmarks, but perhaps with a stack machine as a base processor, it could achieve better results.

Other options would be to more formally prove that the implementation of the emulators and the conversion routines are correct, using mathematical methods. This could be taken further to automatically generate test programs that try to break the conversion or the emulators, generally referred to as 'fuzz testing'.

An ambitious goal would be to implement the conversion and optimisation routines in the target stack architecture. This would mean that the the conversion could be implemented natively on a processor capable of running the stack program. Such a program would be very complicated so a stack language capable of high-level features such as functions would be necessary to make this goal feasible.

Bibliography

[1] J. von Neumann, 'First Draft of a Report on the EDVAC', University of Pennsylvania, Tech. Rep., Jun. 1945.

[2] W. Aspray, 'The Intel 4004 microprocessor: What constituted invention?', *IEEE Annals of the History of Computing*, vol. 19, no. 3, pp. 4–15, 1997.

[3] S. Mittal, 'A survey of techniques for designing and managing CPU register file', *Concurrency and Computation: Practice and Experience*, vol. 29, no. 4, Jul. 2016.

[4] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[5] M. Yokokawa, F. Shoji, A. Uno, M. Kurokawa and T. Watanabe, 'The K Computer: Japanese Next-generation Supercomputer Development Project', in *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ser. ISLPED '11, Fukuoka, Japan: IEEE Press, 2011, pp. 371–372, ISBN: 978-1-61284-660-6.

[6] D. A. Patterson and D. R. Ditzel, 'The Case for the Reduced Instruction Set Computer', *SIGARCH Comput. Archit. News*, vol. 8, no. 6, pp. 25–33, Oct. 1980, ISSN: 0163-5964.

[7] A. P. Speiser, 'The First Computers', in, R. Rojas and U. Hashagen, Eds., MIT Press, 2000, ch. Konrad Zuse's Z4: Architecture, Programming, and Modifications at the ETH Zurich, pp. 263–276, ISBN: 0-262-18197-5.

[8] E. I. Organick, *Computer system organization: the B5700/B6700 series*. Academic Press, 2014.

[9] G. J. Myers, 'The case against stack-oriented instruction sets', *SIGARCH Comput. Archit. News*, vol. 6, no. 3, pp. 7–10, Aug. 1977, ISSN: 0163-5964.

[10] M. Schoeberl, 'Design and implementation of an efficient stack machine', in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, IEEE, 2005.

[11] W. M. McKeeman, 'Peephole Optimization', *Commun. ACM*, vol. 8, no. 7, pp. 443–444, Jul. 1965.

[12] T. P. Baker, 'Stack-based scheduling of realtime processes', *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[13] P. Koopman, 'A preliminary exploration of optimized stack code generation', 1995.

[14] C. Bailey, 'Inter-boundary scheduling of stack operands: A preliminary study', *Procedings of EuroForth 2000*, pp. 3–11, 2000.

[15] M. Shannon, 'A C Compiler for Stack Machines', *Master's thesis, University of York, United Kingdom*, 2006.

[16] M. Gschwind, K. Ebcioğlu, E. Altman and S. Sathaye, 'Binary Translation and Architecture Convergence Issues for IBM System/390', in *Proceedings of the 14th International Conference on Supercomputing*, ser. ICS '00, ACM, 2000, pp. 336–347.

[17] C. Whitby-Strevens, 'The transputer', in *ACM SIGARCH Computer Architecture News*, IEEE Computer Society Press, vol. 13, 1985, pp. 292–300.

[18] P. Mattos, 'The transputer in satellite signal processing', in *Transputer and Signal Processing, IEE Colloquium on*, IET, 1990, pp. 6–1.

[19] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber and J. Mattson, 'The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges', in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '03, IEEE Computer Society, 2003, pp. 15–24, ISBN: 0-7695-1913-X.

[20] V. Smith. (15th Jul. 2003). VHJ: Tracking Transmeta, [Online]. Available: `http://www.vanshardware.com/articles/2003/07/030715_Transmeta/030715_Transmeta.htm` (visited on 15/02/2017).

[21] ——, (4th Apr. 2004). VHJ: Benchmarking Transmeta's efficeon, [Online]. Available: `http://www.vanshardware.com/reviews/2004/04/040405_efficeon/040405_efficeon.htm` (visited on 16/02/2017).

[22] J. Hennessy, *Stanford Benchmark Suite*, version 4.2, 24th Aug. 2005. [Online]. Available: `http://classes.engineering.wustl.edu/cse465/docs/BCCExamples/stanford.c` (visited on 20/04/2017).

Appendix A

DCPU-16 Specification

```
DCPU-16 Specification
Copyright 1985 Mojang
Version 1.7


=== SUMMARY ========================================================================

* 16 bit words
* 0x10000 words of ram
* 8 registers (A, B, C, X, Y, Z, I, J)
* program counter (PC)
* stack pointer (SP)
* extra/excess (EX)
* interrupt address (IA)

In this document, anything within [brackets] is shorthand for "the value of the
RAM at the location of the value inside the brackets". For example, SP means
stack pointer, but [SP] means the value of the RAM at the location the stack
pointer is pointing at.

Whenever the CPU needs to read a word, it reads [PC], then increases PC by one.
Shorthand for this is [PC++]. In some cases, the CPU will modify a value before
reading it, in this case the shorthand is [++PC].

For stability and to reduce bugs, it's strongly suggested all multi-word
operations use little endian in all DCPU-16 programs, wherever possible.


=== INSTRUCTIONS ===================================================================

Instructions are 1-3 words long and are fully defined by the first word.
In a basic instruction, the lower five bits of the first word of the instruction
are the opcode, and the remaining eleven bits are split into a five bit value b
and a six bit value a.
b is always handled by the processor after a, and is the lower five bits.
```

In bits (in LSB-0 format), a basic instruction has the format: aaaaaabbbbbooooo

In the tables below, C is the time required in cycles to look up the value, or perform the opcode, VALUE is the numerical value, NAME is the mnemonic, and DESCRIPTION is a short text that describes the opcode or value.

--- Values: (5/6 bits) ----------

C	VALUE	DESCRIPTION
0	0x00-0x07	register (A, B, C, X, Y, Z, I or J, in that order)
0	0x08-0x0f	[register]
1	0x10-0x17	[register + next word]
0	0x18	(PUSH / [--SP]) if in b, or (POP / [SP++]) if in a
0	0x19	[SP] / PEEK
1	0x1a	[SP + next word] / PICK n
0	0x1b	SP
0	0x1c	PC
0	0x1d	EX
1	0x1e	[next word]
1	0x1f	next word (literal)
0	0x20-0x3f	literal value 0xffff-0x1e (-1..30) (literal) (only for a)

* "next word" means "[PC++]". Increases the word length of the instruction by 1.
* By using 0x18, 0x19, 0x1a as PEEK, POP/PUSH, and PICK there's a reverse stack starting at memory location 0xffff. Example: "SET PUSH, 10", "SET X, POP"
* Attempting to write to a literal value fails silently

--- Basic opcodes (5 bits) ----------

C	VAL	NAME	DESCRIPTION
-	0x00	n/a	special instruction - see below
1	0x01	SET b, a	sets b to a
2	0x02	ADD b, a	sets b to b+a, sets EX to 0x0001 if there's an overflow, 0x0 otherwise
2	0x03	SUB b, a	sets b to b-a, sets EX to 0xffff if there's an underflow, 0x0 otherwise
2	0x04	MUL b, a	sets b to b*a, sets EX to ((b*a)>>16)&0xffff (treats b, a as unsigned)
2	0x05	MLI b, a	like MUL, but treat b, a as signed
3	0x06	DIV b, a	sets b to b/a, sets EX to ((b<<16)/a)&0xffff. if a==0, sets b and EX to 0 instead. (treats b, a as unsigned)
3	0x07	DVI b, a	like DIV, but treat b, a as signed. Rounds towards 0
3	0x08	MOD b, a	sets b to b%a. if a==0, sets b to 0 instead.
3	0x09	MDI b, a	like MOD, but treat b, a as signed. (MDI -7, 16 == -7)
1	0x0a	AND b, a	sets b to b&a
1	0x0b	BOR b, a	sets b to b|a
1	0x0c	XOR b, a	sets b to b^a
1	0x0d	SHR b, a	sets b to b>>>a, sets EX to ((b<<16)>>a)&0xffff

```
   |      |          | (logical shift)
 1 | 0x0e | ASR b, a | sets b to b>>a, sets EX to ((b<<16)>>>a)&0xffff
   |      |          | (arithmetic shift) (treats b as signed)
 1 | 0x0f | SHL b, a | sets b to b<<a, sets EX to ((b<<a)>>16)&0xffff

 2+| 0x10 | IFB b, a | performs next instruction only if (b&a)!=0
 2+| 0x11 | IFC b, a | performs next instruction only if (b&a)==0
 2+| 0x12 | IFE b, a | performs next instruction only if b==a
 2+| 0x13 | IFN b, a | performs next instruction only if b!=a
 2+| 0x14 | IFG b, a | performs next instruction only if b>a
 2+| 0x15 | IFA b, a | performs next instruction only if b>a (signed)
 2+| 0x16 | IFL b, a | performs next instruction only if b<a
 2+| 0x17 | IFU b, a | performs next instruction only if b<a (signed)
 - | 0x18 | -        |
 - | 0x19 | -        |
 3 | 0x1a | ADX b, a | sets b to b+a+EX, sets EX to 0x0001 if there is an over-
   |      |          | flow, 0x0 otherwise
 3 | 0x1b | SBX b, a | sets b to b-a+EX, sets EX to 0xFFFF if there is an under-
   |      |          | flow, 0x0 otherwise
 - | 0x1c | -        |
 - | 0x1d | -        |
 2 | 0x1e | STI b, a | sets b to a, then increases I and J by 1
 2 | 0x1f | STD b, a | sets b to a, then decreases I and J by 1
---+------+----------+-------------------------------------------------------

* The branching opcodes take one cycle longer to perform if the test fails
  When they skip an if instruction, they will skip an additional instruction
  at the cost of one extra cycle. This lets you easily chain conditionals.
* Signed numbers are represented using two's complement.


Special opcodes always have their lower five bits unset, have one value and a
five bit opcode. In binary, they have the format: aaaaaaooooo00000
The value (a) is in the same six bit format as defined earlier.

--- Special opcodes: (5 bits) ------------------------------------------------
 C | VAL  | NAME  | DESCRIPTION
---+------+-------+-------------------------------------------------------------
 - | 0x00 | n/a   | reserved for future expansion
 3 | 0x01 | JSR a | pushes the address of the next instruction to the stack,
   |      |       | then sets PC to a
 - | 0x02 | -     |
 - | 0x03 | -     |
 - | 0x04 | -     |
 - | 0x05 | -     |
 - | 0x06 | -     |
 - | 0x07 | -     |
 4 | 0x08 | INT a | triggers a software interrupt with message a
 1 | 0x09 | IAG a | sets a to IA
 1 | 0x0a | IAS a | sets IA to a
 3 | 0x0b | RFI a | disables interrupt queueing, pops A from the stack, then
   |      |       | pops PC from the stack
 2 | 0x0c | IAQ a | if a is nonzero, interrupts will be added to the queue
```

```
   |      |       | instead of triggered. if a is zero, interrupts will be
   |      |       | triggered as normal again
 - | 0x0d | -     |
 - | 0x0e | -     |
 - | 0x0f | -     |
 2 | 0x10 | HWN a | sets a to number of connected hardware devices
 4 | 0x11 | HWQ a | sets A, B, C, X, Y registers to information about hardware a
   |      |       | A+(B<<16) is a 32 bit word identifying the hardware id
   |      |       | C is the hardware version
   |      |       | X+(Y<<16) is a 32 bit word identifying the manufacturer
 4+| 0x12 | HWI a | sends an interrupt to hardware a
 - | 0x13 | -     |
 - | 0x14 | -     |
 - | 0x15 | -     |
 - | 0x16 | -     |
 - | 0x17 | -     |
 - | 0x18 | -     |
 - | 0x19 | -     |
 - | 0x1a | -     |
 - | 0x1b | -     |
 - | 0x1c | -     |
 - | 0x1d | -     |
 - | 0x1e | -     |
 - | 0x1f | -     |
---+------+-------+-------------------------------------------------------------
```

=== INTERRUPTS ==

The DCPU-16 will perform at most one interrupt between each instruction. If multiple interrupts are triggered at the same time, they are added to a queue. If the queue grows longer than 256 interrupts, the DCPU-16 will catch fire.

When IA is set to something other than 0, interrupts triggered on the DCPU-16 will turn on interrupt queueing, push PC to the stack, followed by pushing A to the stack, then set the PC to IA, and A to the interrupt message.

If IA is set to 0, a triggered interrupt does nothing. Software interrupts still take up four clock cycles, but immediately return, incoming hardware interrupts are ignored. Note that a queued interrupt is considered triggered when it leaves the queue, not when it enters it.

Interrupt handlers should end with RFI, which will disable interrupt queueing and pop A and PC from the stack as a single atomic instruction.
IAQ is normally not needed within an interrupt handler, but is useful for time critical code.

=== HARDWARE ==

The DCPU-16 supports up to 65535 connected hardware devices. These devices can be anything from additional storage, sensors, monitors or speakers.
How to control the hardware is specified per hardware device, but the DCPU-16 supports a standard enumeration method for detecting connected hardware via the HWN, HWQ and HWI instructions.

Interrupts sent to hardware can't contain messages, can take additional cycles, and can read or modify any registers or memory adresses on the DCPU-16. This behavior changes per hardware device and is described in the hardware's documentation.

Hardware must NOT start modifying registers or ram on the DCPU-16 before at least one HWI call has been made to the hardware.

The DPCU-16 does not support hot swapping hardware. The behavior of connecting or disconnecting hardware while the DCPU-16 is running is undefined.

Appendix B

Simplified J5 Instruction Sheet

Transcribed from 2013 DACS Exam paper

Arithmetic & Logical

ADD	Add Top & Next on stack together (3,2) -> (5)
SUB	Subtract Top from Next on stack (2,8) -> (6)
INC	Increment Top of stack
DEC	Decrement Top of stack
AND/OR/NOT/XOR	Combine Top & Next in in logical operation
SHR	Shift Top of stack 1 bit to right
SHL	Shift Top of stack 1 bit to left

Comparative

TGT	Test Top greater than Next, set Zero flag to true or false*
TLT	Test Top less than Next, set Zero flag to true or false*
TEQ	Test Top equal to Next, set Zero flag to true or false*
TSZ	Test Top set to zero, set Zero flag to true or false*

*Note these do not destroy Top and Next when they are operated.

Other instructions

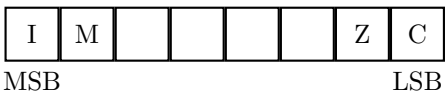
SSET n	Place an operand n on the stack
SET nn	Place an operand nn on the stack
LOAD	Take address from Top of stack, access location, put result on stack
STORE	Take Top as address, and Next as data, then write data to the address
BRANCH nn	Unconditional relative branch of +/- nn bits
BRZERO nn	Conditional branch (taken if Zero Flag set)
IBRANCH	Indirect branch PC <= M[TOS]
CALL nn	Call to address nn
RETURN	Return from call
STOP	Causes Execution to halt

Stack manipulations

```
DROP      Drop an item from the Top of the stack
DUP       Duplicate Top of stack                          (X,Y) -> (X,X,Y)
SWAP      Swap Top and Next                                 (X,Y) -> (Y,X)
RSD3      Rotate stack down three                         (X,Y,Z) -> (Z,X,Y)
RSU3      Rotate stack up three                           (X,Y,Z) -> (Y,Z,X)
TUCK2     Tuck copy of Top under 2nd item                (X,Y,Z) -> (X,Y,X,Z)
TUCK3     Tuck copy of Top under 3rd item                (X,Y,Z) -> (X,Y,Z,X)
COPY3     Copy 3rd stack item to Top of stack
PUSH rr   Push register (F,LBR,GBR,VBA) onto stack
POP rr    Pop register (F,LBR,GBR,VBA) from stack
```

Registers

The J5 has the usual stack registers, Top, Next, 3rd, etc, and also has Local Base Register (**LBR**), Global Base Register (**GBR**), Vector Base Address (**VBA**), and an F register containing the flags shown below.



C-Carry Flag **Z**-Zero Flag
I-Interrupt (1=Enable, 0=Disable)
M-Interrupt Mode (1=Vector, 0=Opcode)

Appendix C

Benchmark programs

```
; initial "input"
SET [0x3000], 6221
SET [0x3001], 16181
SET [0x3002], 29598
SET [0x3003], 10115
SET [0x3004], 18953
SET [0x3005], 44337
SET [0x3006], 12629
SET [0x3007], 7527
SET [0x3008], 56169
SET [0x3009], 44334
SET [0x300a], 2290
SET [0x300b], 38260
SET [0x300c], 62627
SET [0x300d], 9740
SET [0x300e], 40652
SET [0x300f], 58444
SET [0x3010], 21421
SET [0x3011], 45306
SET [0x3012], 38396
SET [0x3013], 25410
SET [0x3014], 31580
SET [0x3015], 17208
SET [0x3016], 27939
SET [0x3017], 59297
SET [0x3018], 55094
SET [0x3019], 45095
SET [0x301a], 55388
SET [0x301b], 30943
SET [0x301c], 33757
SET [0x301d], 48729
SET [0x301e], 42
SET [0x301f], 29832
```

```
SET [0x2fff], 32 ; length

; sort
SET J, [0x2fff]
:LOOP SUB J, 1
        SET I, 0
:LOOP2 SET A, [0x3000+I]
        SET B, [0x3000+J]
        IFG A, B ; compare
                SET PC, SWAP
        SET PC, POST
:SWAP SET C, [0x3000+I]
        SET [0x3000+I], [0x3000+J]
        SET [0x3000+J], C
:POST ADD I, 1
        IFG J, I
                SET PC, LOOP2
        IFN J, 0
                SET PC, LOOP

; output result
SET I, 0
:RESL OUT [0x3000+I]
        ADD I, 1
        IFN I, [0x2fff]
                SET PC, RESL
```

Listing C.1: bsort.reg

```
SET [0x2000], 20 ; limit

SET A, 0 ; F(0)
SET B, 1 ; F(1)

OUT A
OUT B

SET I, 1
:LOOP SET C, A
        SET A, B
        ADD B, C
        OUT B
        ADD I, 1
        IFN I, [0x2000]
                SET PC, LOOP
```

Listing C.2: fib20.reg

```
SET I, 100 ; limit
SET A, 2
:LOOP IFE [A+3000], 0
        SET PC, SCAN
:SCANNED ADD A, 1
IFG A, I
    SET PC, PC ; we're done
SET PC, LOOP

:SCAN IFL A, I
        OUT A
SET B, A
:LOOP2 SET [B+3000], 1
ADD B, A
IFL B, I
        SET PC, LOOP2
SET PC, SCANNED
```

Listing C.3: primes.reg

```
SET C, 100 ; limit
SET I, 1
:LOOP SET J, 0
SET B, 0
:LOOP2 ADD J, 1
ADD B, J
IFL J, I
  SET PC, LOOP2
:BREAK OUT B
ADD I, 1
IFL I, C
  SET PC, LOOP
```

Listing C.4: tri100.reg