Enhancing the RISC-V Instruction Set Architecture

Plamen Petrov

MInf Project (Part 1) Report
Master of Informatics
School of Informatics
University of Edinburgh
2019
Abstract

Since its emergence, RISC-V has offered a license-free alternative to proprietary instruction sets designed by ARM and Intel by providing a stable base to researchers and small companies. This makes it possible for small companies and researchers to design new cores without incurring exorbitant costs. In our two-part project we explore RISC-V’s performance compared to the commercial ARM and X86 architectures. Most of the first part focuses on static code size, where we find that RISC-V performs very reasonably, but is behind Thumb2. For this reason, we propose improvements to RISC-V which constitute of new prologue and epilogue instructions that almost fully bridge this gap. We give ideas for other possible enhancements and plan to extend the project to cover dynamic measures in the second part. Overall, we conclude that RISC-V performs very well given the resources invested in it and open instruction sets, in general, deserve to be developed further for the better of the computer architecture field as a whole.
Acknowledgements

I would like to express my gratitude to my supervisor Professor Nigel Topham for supporting me with the project throughout the year.
# Table of Contents

1 Introduction ................................................................. 7
   1.1 Overview .............................................................. 7
   1.2 Aims ........................................................................ 9
   1.3 Contributions .......................................................... 10
   1.4 Outline ..................................................................... 10

2 Background ...................................................................... 11
   2.1 Abbreviations .......................................................... 11
   2.2 Binary File ............................................................... 11
   2.3 SPEC2006 .................................................................. 12
   2.4 Register Usage .......................................................... 13
   2.5 Stack Frame ............................................................... 13
   2.6 Function Prologue and Epilogue .................................. 14

3 Methodology .................................................................... 17

4 Benchmark Results ......................................................... 19
   4.1 Pure and Stripped Binary Size ...................................... 19
   4.2 Loadable Section Size .................................................. 20
   4.3 Text Section Size ....................................................... 23
   4.4 Compressed vs. Non-Compressed ................................. 25
   4.5 CSiBE ....................................................................... 28
   4.6 Instruction Frequencies .............................................. 28
   4.7 Average Instruction Encoding Length ......................... 30
   4.8 Instruction Count ....................................................... 31
   4.9 Summary and Evaluation ............................................ 32

5 Prologue and Epilogue Improvement .................................... 33
   5.1 Overview ................................................................. 33
   5.2 Motivation ............................................................... 34
   5.3 Expected Code Size Reduction ..................................... 34
      5.3.1 Considerations and Assumptions ......................... 35
      5.3.2 Results .............................................................. 36
   5.4 Design and Implementation ........................................ 39
      5.4.1 Compressed Instruction Format ......................... 40
      5.4.2 Implementation .................................................. 44
TABLE OF CONTENTS

5.4.3 Implementation Pipeline ...................................... 45
5.5 Verification .......................................................... 46
5.6 Evaluation ............................................................. 48

6 Other Improvements and Future Plan ................................ 51
6.1 Other Improvements ................................................... 51
6.1.1 Caller-saved Registers ............................................. 51
6.1.2 Pointer Arithmetic .................................................. 52
6.2 Future Plan .............................................................. 53
6.2.1 Dynamic Analysis ................................................... 54
6.2.2 Implement in GCC ................................................... 55

7 Conclusion ................................................................. 57

Bibliography ................................................................. 59

A Graphs ...................................................................... 61
Chapter 1

Introduction

1.1 Overview

Nowadays, computer systems are immensely complex and it is not uncommon to have more than 10 cores using different instruction sets on the same chip. Thus, the instruction set is the most important interface in computer systems and it is where the software meets the hardware. Despite the rapid development in the computer systems field in the last 60 years, arguably less has evolved in the computer architecture industry. For example, contrary to other computing fields such as networking (TCP/IP, Ethernet), databases (SQL), and operating systems (Posix), where open standards have become the norm, there has not been significant mutual effort put into designing an open universal instruction set standard [1]. In fact, the most widely-used instruction sets have serious IP restrictions, like patents over some of their main features, which makes it hard to impossible for small companies or researchers to acquire access to them [2]. To give concreteness, only around large 15 companies have licences to design new ARM cores [3]. There is another drawback to proprietary instruction sets where if the company dies, the instruction set and all of the software support dies with it, which was the case for DEC and their Alpha and VAX ISA’s. On the other hand, an open standard would promote competition and drive research in the area by allowing more contributors with different innovative ideas and make testing and prototypes cheaper and easier to implement. Before RISC-V there were a couple of attempts to popularise open instruction sets. SPARC V8 was made an IEEE standard [4], but it only had a 32-bit version and is now owned by Oracle and is proprietary. What is more, Asanovic and Patterson argue that its register windows are more of a deficiency than an enabling. Another attempt was OpenRISC [5], which was used in a couple of SoC’s [6,7], but, because of being written in Verilog, the implementation and design of the instruction set were very tightly coupled which made the ISA inflexible. These architectures are still used, but did not manage to make the open instructions sets prevalent and the standard in the instruction set field.

Following these unsuccessful attempts to create a widely-used open instruction set, it is natural to ask what features one needs to have in order to last, and be competitive with commercial instruction sets. Krste Asanovic tells a story of receiving an angry
email from a third party after the RISC-V base instruction set was modified, because the changes were incompatible with that party’s work [1]. Therefore, the first requirement for a universal ISA is to freeze its base, so that it does not change over time, and is a stable target for software tools. The base has to be complemented by standard optional extensions and leave encoding space for custom domain and application relevant instructions. This design has the benefit that the base implementation is simple, small and cheap to implement and can be supplemented only by the necessary extensions. For example, tiny embedded and IoT devices might only implement the base while mainframes and machines which do scientific computations may have all standard extensions. This concept is known as the fixed plus variable structure computer [8]. Finally, the instruction set should also be suitable for all implementation technologies like FPGA, ASIC, etc. in order to make the architecture flexible in different domains.

RISC-V is an open instruction set architecture (ISA) that promises to fulfil all of the above requirements and to break the staleness in the instruction set industry. It is under the BSD standard and royalty free, which makes it suitable for use by researchers and small companies that want to design their own core [9]. It was developed by the Computer Science Division of the EECS Department at UC Berkeley and is being maintained by the RISC-V Foundation [10]. It is a RISC instruction set that has 32, 64 and even 128 bit address space variants. It has a defined base which will not change as well as standard extensions, and it also allows the users to define their own extensions as relevant to their application [11] [12]. As a result the base is clean and can be implemented cheaply in almost half as much area as ARM or x86. RISC-V provides the programmer with 31 general-purpose registers (GPR), x1-x31, with register x0 being hardwired to 0, a load-store memory model and a register+offset addressing mode. It defines instruction formats which place source and destination operands in the same place for the different instructions, which accelerates decoding, but incurs some complexity costs such as having to split immediates across the instruction. The standard extensions and their functionality are summarised in Figure 1.1.

<table>
<thead>
<tr>
<th>Extension</th>
<th>Instructions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>8</td>
<td>Integer multiply and divide</td>
</tr>
<tr>
<td>A</td>
<td>11</td>
<td>Atomic memory operations, load-reserve/store conditional</td>
</tr>
<tr>
<td>F</td>
<td>26</td>
<td>Single-precision (32 bit) floating point</td>
</tr>
<tr>
<td>D</td>
<td>26</td>
<td>Double-precision (64 bit) floating point; requires F extension</td>
</tr>
<tr>
<td>Q</td>
<td>26</td>
<td>Quad-precision (128 bit) floating point; requires F and D extensions</td>
</tr>
<tr>
<td>C</td>
<td>46</td>
<td>Compressed integer instructions; reduces size to 16 bits</td>
</tr>
</tbody>
</table>

Figure 1.1: Standard RISC-V Extensions

RISC-V has fixed-length instructions - 4 Bytes, but the 'C' extension shortens the most common instructions to 2 Bytes. By design, all compressed instructions can be expanded to a full instruction, which means that the compressed and non-compressed versions can coexist without the compiler’s awareness [13]. More details about the
1.2. Aims

So far, we have made RISC-V seem like the solution to all instruction set issues, as indeed, with a lot hindsight, it avoids the pitfalls of previous open ISA’s and provides an alternative to the expensive proprietary architectures. But in order to transition from a prototype, research instruction set to a commercial one, RISC-V needs to be able to compete with the widely-used alternatives, ARM and X86, both in terms of performance and software/customer support. As far as performance is concerned, because ARM and X86 work in different domains, competing with each of them means different things. For example, ARM is most prominent in the embedded world where power efficiency, chip area and code size are crucial, which means RISC-V must be competitive in these fields. X86 is the most common server and desktop architecture, where parallelism and execution time is more important. Being competitive on all of these measures is indeed one of the challenges of designing a universal instruction set and potentially the reason why there has not been a successful open instruction set.

Our goal in this project is to explore the performance of RISC-V and improve it. The first objective is to run benchmarks on RISC-V and compare it with ARM and X86 in order to learn where it is faring well or lacking. We compare RISC-V32GC with Thumb2 and IA32 as well as RISC-V64GC with ARMv8-A and X64. The second objective is to propose improvements to RISC-V in the form of new instructions and to evaluate their effect on real programs and implementation feasibility. Due to the fact that this is an MInf project, this first part of the project focuses on static statistics like code size, while next year we are going to focus on dynamic measures like execution time. During the course of the project, we also elaborate on our methodology to make sure our results are reproducible.
1.3 Contributions

This section lists the main contributions that I have made to the project.

1. Compared the relative performance of RISC-V, ARM, and X86 for 32-bit and 64-bit on static code size, average instruction length and instruction count. Explored and reported on instruction frequencies and binary size.

2. Proposed, designed, implemented and evaluated new prologue and epilogue instructions for RISC-V that would reduce its code size.

3. Proposed other possible code density improvements to RISC-V and identified cases where they might be utilised.

4. Implemented a verification script for the RISC-V simulator Spike that is used to verify the behaviour of the new prologue and epilogue instructions, but can also be used more generally to validate any behaviour.

5. Designed the code in a modular way which would make it possible to perform the same study on another benchmark without major changes.

6. Designed the Gem5 configuration that would be used to dynamically compare the instruction sets. This is going to be useful in the second part of the project.

1.4 Outline

Chapter 1 introduces RISC-V as an open instruction set and motivates the motivation for improving it and making it competitive with other commercial instruction sets. It outlines the aims, achievements and structure of the project.

Chapter 2 is the background which gives an overview of important topics needed to understand the project such as the stack frame, instruction prologues and epilogues, etc.

Chapter 3 elaborates on our methodology to make sure the results are reproducible and valuable for what they are.

Chapter 4 presents the results obtained from comparing RISC-V with ARM and X86 statically. It discusses binary size, loadable section size, code size, average instruction length, instruction count, and instruction frequencies.

Chapter 5 introduces the main improvement we propose to RISC-V - dedicated prologue and epilogue instructions. It elaborates on why they are needed, their design, implementation, verification and effect.

Chapter 6 discusses the plan for the second part of the project and mentions other possible enhancements that could be implemented in RISC-V.

Chapter 7 concludes this part of the project by summarising the results. It gives my personal opinion on the results and their implications for RISC-V.
Chapter 2

Background

This chapter describes topics which are essential to understanding the later parts of the project.

2.1 Abbreviations

In this section we list the abbreviations that we use throughout the project for the various instruction sets that we refer to:

- RISC-V can be seen as RV. It is used to refer to the 32-bit and the 64-bit RISC-V ISA collectively.
- RISC-V32GC refers to the 32-bit RISC-V ISA with all standard extensions and the compressed extension. This can also be seen as RV32GC. Analogously we define RISC-V64GC and RV64GC.
- RISC-V32G refers to the 32-bit RISC-V ISA with all standard extensions but without the compressed extension. This can also be seen as RV32G. Analogously we define RISC-V64G and RV64G.
- Thumb2 refers to ARMv7-A compiled with the Thumb2 compression format. This can also be referred to as ARMv7-A(Thumb2).
- ARMv7-A refers to ARMv7-A compiled without the Thumb2 compression format.

2.2 Binary File

Within this study we use binary (file) or executable (file) to refer to files that contain compiled programs, which can be run on a specific machine. Thus, there are different binary formats for the various machines. Some examples are EXE for Windows systems and ELF for UNIX systems. Moreover, executables differ depending on the
instruction set of the computer. There is an Application Binary Interface (ABI) document for each ISA that specifies how arguments are to be passed to functions, what the recognised types are and their width, etc.

Binaries are divided into sections. The sections of most interest to us are present in most binary files, regardless of machine. Among them are `.text`, `.bss`, `.rodata`, and `.data`. The `text` section contains the instructions or the code of the program. The data and bss sections contain initialised or uninitialised statically-allocated data, respectively. Read-only data can be found in rodata. As we will see in Section 4.3, these distinctions are not strict, but the common thing about these sections is that they are loaded into the memory, when the programs are run.

There are other sections, which do not have to have the same name across architectures and they are compiler and ISA-dependent. These can contain debugging data or information about how the binary was generated. These sections are not typically loaded in main memory during execution. The structure of an ELF file (Executable and Linkable Format) is shown in Figure 2.1. The program header contains information that is needed for run time execution of the file, while the section header keeps important data for linking and relocation.

![Figure 2.1: ELF file structure](image)

2.3 SPEC2006

In order to carry out our study to compare RISC-V with ARM and X86 we use the SPEC2006 benchmark suite. Although dated, it is well-established and a lot of relevant documentation and troubleshooting techniques can be found online. The suite contains 12 integer benchmarks, collectively called "SPECint 2006", and 19 floating-point benchmarks - "SPECfp 2006". Since we do not use another SPEC suite, we
will shorten to SPECint and SPECfp, respectively. Due to unresolved build issues, we do not consider some of the benchmarks:

- **32 bit**: 403.gcc, 483.xalancbmk, 447.dealII, 450.soplex, 481.wrf
- **64 bit**: 483.xalancbmk, 447.dealII, 450.soplex, 481.wrf

This means that we utilise 26 32-bit (10 INT and 16 FP) benchmarks and 27 64-bit benchmarks (11 INT and 16 FP).

### 2.4 Register Usage

In this section we elaborate on the difference between caller-saved and callee-saved register. RISC-V has 31 general-purpose registers (x1-x31), while x0 is hardwired to 0. Their purposes, names and descriptions are shown in Table [2.1]. Although some of them have special uses, such as the stack pointer or the return address, registers x5-x31 are indistinguishable. However, assembly language programmers have defined conventions to differentiate registers in order to allow the code written by different people to be compatible. One if the most prominent conventions is which registers are caller-saved and which are callee-saved. Caller-saved registers are the ones whose value is not guaranteed to persist after the callee terminates. Hence, if they are needed after the callee terminates, the caller needs to save them on the stack before the function call to the callee and restore them after. Similarly, callee-saved registers are the GPR’s whose value must stay the same after the callee terminates. Thus, if the callee wishes to utilise any of them, it needs to store them on the stack and restore them before returning. In RISC-V, registers s0-s11 are callee-saved (also known as ”s-registers”) and registers t0-t6 and a0-a7 are caller-saved (”t-registers” and ”a-registers”).

### 2.5 Stack Frame

The stack frame is central to the idea of function prologue and epilogue which we discuss in Section [2.6]. The stack frame of a function is a region of memory on the stack that contains the local variables and the parameters of the subroutine. Its structure is defined quite strictly by the ABI of the instruction set. From inspecting the RISC-V port of GCC we find that the RISC-V stack frame has the structure shown in Figure [2.2]. Let us cover in detail each of the different components:

- **Frame pointer** The frame pointer is the value of the stack pointer before the stack frame is allocated. Since the stack frame exists only over the lifetime of the subroutine, the stack pointer needs to be restored to the value of the frame pointer before returning control to the caller. The frame pointer register is most often used to offset the function parameters and local variables. If these do not exist, the frame pointer register might not even be used.

- **Subroutine parameters** The parameters passed to the function.
Table 2.1: RISC-V Register Usage  

Source: [23]

- **GP callee-saved registers** The value of general purpose callee-saved registers, whose value the subroutine is going to modify, is saved and restored from here before termination. Not all callee-saved registers need to be on the stack if they are not going to be modified.

- **FP callee-saved registers** The same as for GPR, but for FP registers if they are available.

- **Other local variables** Space for local variables of the function.

Finally, not all of these sections need to occupy any space. For example, if we are not using the ‘F’ or ‘D’ extensions of RISC-V, there would not be any FP registers saved on the stack.

### 2.6 Function Prologue and Epilogue

After emphasising the importance of adhering to the convention of which registers are callee-saved and which are caller-saved, we introduce a closely-related notion of function prologues and epilogues. The prologue appears at the start of a function and sets up the stack frame. This includes allocating space on the stack for local variables, as well as storing the value of callee-saved registers whose value the procedure is going to modify, setting up the frame pointer if it is necessary, etc. A typical RISC-V32G prologue is shown in Figure 2.3. Line 1 allocates 64 bytes on the stack, and line 3 stores the value of the return address at offset 60 from the stack pointer. Then, all 12
calleesaved registers are stored on the stack at offsets 12-60. The space from 0 to 12 is for other local variables. Line 16 sets the frame pointer to the value of register a3. The epilogue that corresponds to this prologue is shown in Figure 2.4, and it just restores the value of the calleesaved registers and deallocates the stack space. Sometimes, not all calleesaved registers are used by a function and so it is pointless to store (and later restore, in the epilogue) their value because it never changes. For 64-bit RISC-V the prologue and epilogue are almost the same, except they would use `ld` and `sd`, instead of `lw` and `sw` and each register would occupy 8 bytes on the stack, which would be reflected in double the space needed to store all calleesaved registers on the stack.
Figure 2.4: A typical RISC-V32GC function epilogue

```
# restore callee-saved regs
lw $ra, 60($sp)
lw $s0, 56($sp)
lw $s1, 52($sp)
lw $s2, 48($sp)
lw $s3, 44($sp)
lw $s4, 40($sp)
lw $s5, 36($sp)
lw $s6, 32($sp)
lw $s7, 28($sp)
lw $s8, 24($sp)
lw $s9, 20($sp)
lw $s10, 16($sp)
lw $s11, 12($sp)
addi $sp, $sp, 64
```
Chapter 3

Methodology

In this chapter, we explain in detail and justify the way in which we reach the results that we present later. This is essential in order to allow the experiments and outcomes to be reproducible and trustful for what they are. First of all, let us list the toolchains that we use to build the benchmarks. For RISC-V, we use the official GNU toolchain found from RISC-V’s GitHub. An alternative is to use the LLVM compiler but it is not as developed [16]. We build the Linux cross-compiler both for 32-bit and 64-bit (using multilib). The other option is the Newlib cross-compiler, which emits a binary to run on bare metal, but performs static linking, which results in a lot bigger code and binary size. This is undesirable as we are not comparing the instruction sets on bare metal but full systems, which would typically provide the necessary libraries dynamically. For 64-bit ARM we use the aarch64-gnu-linux toolchain for Ubuntu to cross-compile for ARMv8-A, while for 32-bit ARM we use the arm-gnueabihf toolchain which compiles code to be run on a Linux system, and it can emit code in thumb or arm mode, depending on whether we want to compress the code. Other possible toolchains to use are arm-eabi(hf), which produces a bare metal binary if one is needed to compare bare metal binaries across the instruction sets. The ’hf’ option just stands for implicit hard floating point calling convention which allows the generation of floating-point instructions. This is something we want as we would like to use the full power of the ARM ISA. For IA32 and x64 we use the x86_64-linux-gnu toolchain, which can cross-compile both for 32- and 64-bit.

We provide the commands with which we compile the SPEC and CSiBE benchmarks. We show how to compile programs in C, but C++ and Fortran are analogous, except g++ and gfortran are used instead of gcc. In all cases we use the flag -Os for the most code size optimisation, as this is going to be the main metric we will be interested in.

- **RISC-V32GC:**
  riscv64-unknown-linux-gnu-gcc -march=rv32gc -mabi=ilp32d -Os, where
  -march specifies the target ISA and -mabi is the calling convention. We use the hard calling convention which allows single and double precision floating-point values to be passed in registers.

- **RISC-V64GC:**
Chapter 3. Methodology

riscv64-unknown-linux-gnu-gcc -march=rv64gc -mabi=lp64d -Os

- **IA32:**
  x86_64-linux-gnu-gcc -m32 -march=i386, where -m32 specifies that we are targeting a 32-bit machine

- **X64:**
  x86_64-linux-gnu-gcc -march=x86-64

- **Thumb2**
  arm-linux-gnueabihf-gcc -march=armv7-a -mthumb, where -mthumb specifies that Thumb instructions are enabled.

- **ARMv8-A**
  aarch64-linux-gnu-gcc -march=armv8-a

When comparing Thumb2 and RVC, we also need to build the benchmarks with the uncompressed instruction set extensions. In these cases, we make slight changes to the commands. To compile for `RISC-V32G` and `RISC-V64G`, we have to use -march=rv32g or -march=rv64g, respectively. It is important to mention that we need to rebuild the compiler to target the original ISA variant (without the 'C' extension). To compile for ARMv7-A, we use -marm instead of -mthumb.
This chapter presents and discusses the work carried out to evaluate how RISC-V performs compared to ARM and X86. We evaluate the instruction sets statically, meaning that we examine the compiled files. The plan is to focus on dynamic measures next year (see Section 6.2).

In this chapter we measure and discuss statistics such as binary size, text section size, instruction frequencies, average instruction encoding length and instruction count. One might argue that these are not as important as dynamic measures, because we usually care about how fast a program executes. This is quite a superficial way of looking at things, but is often employed even by compiler writers who study performance deeply, and results in neglecting the side effects of dynamic optimisations such as loop unrolling. In embedded systems, where memory and storage is very limited, keeping programs small is critical to minimise swapping and secondary storage usage, which reduces the power consumption as well. Not to mention that the tasks performed by embedded systems are not computationally expensive, so sometimes the execution time is not first priority. What is more, dynamic measures are very often impacted by static measures. For instance, given shorter instructions, we would fetch more instructions in one cache block, thus reducing the cold misses in the cache on average. Cold misses are one type of cache miss, which occur when we fetch the data for the first time (always a miss), and there is usually very little to do about them without causing the number of other misses to soar. Smaller instructions make the cache seem bigger in effect, which also reduces the capacity and conflict misses. The examples given above have nothing to do with the implementation and the microarchitecture of the processor. They are solely ISA-dependent but alleviate the effect of the bottleneck in virtually all modern systems - memory.

4.1 Pure and Stripped Binary Size

Let us define pure binaries to refer to the binary files produced by the compiler without any special flags, as shown in Chapter 3. In a similar spirit, let stripped binaries be the binary files which result from running the GNU binary utilities command strip on...
the pure binaries.
Comparing the size of the pure binaries for the different ISA's was a naive initial attempt to get an understanding of how the ISA's compare with one another in terms of code size. It is not representative, because the pure binaries contain symbols, debugging information and other data which are not relevant to the code size. More importantly, there are benchmarks for which the .bss section is larger than the pure binary file itself. This is possible because .bss keeps data about statically-allocated variables, which are not initialised to an explicit value (Section 2.2), so it is more space-efficient to just store the boundaries of .bss in the binary file.

Instead of throwing away these data, we tried to find some merit for them by comparing the size of the pure binaries to the stripped binaries. We ran `strip --strip-unneeded` on the pure binary files, which removes all symbols except the ones necessary for relocation purposes as well as debugging sections, like the .dwo DW ARF sections. Stripping is common when binary files are released, as not only does it reduce the disk space required, but it also makes the files more resistant to reverse engineering. We did not expect the size difference between the pure and stripped binaries to vary across the instruction sets, because this debugging information should be of similar size for all ISA's. This was indeed the case except for a couple of benchmarks where RISC-V had more debugging information, but this is tolerable given it is newer and used for research and prototyping.

### 4.2 Loadable Section Size

Let us define the loadable sections of a binary file as the sum of the .text, .data, and .bss sections, as output by the GNU command `size -B`, where `-B` stands for Berkeley formatting of the size of the different sections. This means the size reported for .bss contains the .sbss section. Similarly, the results for .text and .data contain other sections such as .sdata and .rodata, etc. The bottom line is that the sum of these loadable sections represents the memory footprint of the program (the amount of main memory that a it might use or reference while running), since it contains just the instructions and the data the program operates on. The results for the 32-bit instruction sets are presented in Figure 4.1.

For the integer benchmarks (Figure 4.1a), IA32 has 50% bigger loadable sections than the other two ISA's for libquantum. Thumb2 has substantially smaller loadable sections for omnetpp and astar, but these seem to be extreme cases, as the average suggests all the instruction sets are very even.

The floating-point case (Figure 4.1b) is similar, RISC-V and ARM are again in tight competition. But IA32 seems to have even bigger loadable sections than in the integer case if we look at the individual benchmarks. We would expect that, if for most benchmarks IA32 had larger loadable sections, that would hold on average as well. However, the average again suggests that IA32 is extremely close to the other two, even though the sizes are only very close for gamess and zeusmp. This is no longer so unusual when we see the loadable sections for these two programs. Indeed, they are
4.2. Loadable Section Size

Figure 4.1: SPEC2006 loadable sections sizes for the 32-bit ISA’s, normalised with respect to RV32GC.

**many** orders of magnitude larger than the loadable sections of the other benchmarks, thus dominating the average (Figure 4.2).

If we look into how each of the .data, .bss, and .text sections contribute to the enormous loadable size of these benchmarks (Figure 4.3), we find that the .bss section accounts for 99.2% and 99.9% of the size of the loadable sections for 416.gamess and 434.zeusmp, respectively. What is more, the size of the .bss section is practically the same across the three instruction sets. The reason for this is that .bss is not really ISA-
dependent, but program dependent, and these two benchmarks just happen to have a huge amount of uninitialised data.

As a result, the averages we obtain for the loadable sections (Figure 4.1) are almost exclusively based on the size of the .bss section of the two largest benchmarks. Therefore, the individual benchmark results are fairly representative of the code size for benchmarks that have a relatively small .bss and .data section, but the averages are insignificant.

We encounter the same issue when we examine the loadable section sizes for the 64-bit instruction sets (Figures A.1 and A.2). Again, the total FP average is almost the same, although there is quite a bit of difference on an individual level and the reason, again, is the size of .bss.
4.3 Text Section Size

The obstacle we faced above leads us to discard the .bss section from consideration when calculating the code sizes per ISA. The .data section is similar to .bss in that it is much more program-dependent than ISA-dependent, as it also contains statically-allocated variables. Therefore, we have to treat it in the same way, because once we remove .bss, then .data can cause the same problem of dominating the loadable size results. For example, if the .data section for one benchmark is huge, compared to the .text and .data sections of the other benchmarks. Indeed, we see in Figure 4.4 that this is the case for 445.gobmk. Although 465.tonto has a .text section which is about double the size than that of 445.gobmk, collectively the latter’s text and data sections would have a bigger weight.

![Figure 4.4: Loadable sections of gobmk and tonto.](image)

It seems that if we were to get an accurate idea of the code size, we would need to only consider the .text section. However, compilers sometimes store statically-allocated constants in the .text section, as opposed to .data for efficiency. This can happen in different cases, but, if a global constant is only used once in the program, it is better to fetch it to the cache with the instructions in which it is used, as opposed to having to access a different region of the memory, which might miss in the cache (or pollute it) and cause the processor to stall. When we work with the total loadable section size in Section 4.2, it is not important if the data and code sections overlap, because we take all of them into consideration. But now it is possible that if for one instruction set there are data in the .text section and for another there are not, then the former ISA would look like it is generating more code than it actually is.

To this end, the next step is to subtract the space occupied by statically-allocated variables in the .text section for all instruction sets. We can insert data in the text section with different assembler directives. For example, in order to allocate a byte or a word (4 bytes), we could use `.byte <value>` or `.word <value>`, respectively. We find that the only ISAs that store data in the .text section are Thumb2 in the 32-bit case and ARMv8-A in the 64-bit case. The percentages of the .text section occupied by data per benchmark are shown in Figure 4.5. Thumb2 keeps more data in the .text section - 6.54% on average, as opposed to 1.37% for ARMv8-A. It is essential to note that although we remove this statically-allocated data in our discussion here, this is
an optimisation that is going to make a difference dynamically when we execute the programs.

Figure 4.5: Percentage of text section that contains data for Thumb2 and ARMv8-A.

With that in mind, we are ready to show the results for the size of the .text section for the different instruction sets. The total results are shown in Table 4.1. The 32-bit results per individual benchmark are presented in Figure 4.6. We see that on average Thumb2 generates 12% less code than RISC-V32GC, while IA32 generates about 25% more. For most of the individual benchmarks, the tendency seems to be that RISC-V has a bigger code size. This provides some reassurance that the average was not just dominated by one large benchmark for which ARM just happens to have smaller code size. For the integer benchmarks (Figure 4.6a), Thumb2 emits even less code, 18% less than RISC-V. The results for the 64-bit benchmarks are shown in Figure 4.7, where the situation is very different, but not unexpected. RISC-V64GC produces smaller code for every single benchmark, with X64 again lagging behind by generating about 35% more code, and ARMv8-A emitting 24% more code on average. X64’s performance does not surprise us because it was lacking in the 32-bit variant as well, and there does not exist a fundamental difference between IA32 and X64 which would improve the code density significantly. However, for ARMv8-A it could be argued that if there was a compressed instruction set format, like Thumb2 for the 32-bit ARM instruction sets, then 64-bit ARM would generate code of competitive size to RISC-V64GC (We explore this possibility in Section 4.4.) Another interesting observation from the graphs is that X64 has better code density than ARMv8-A for every single integer benchmark (Figure 4.7a). This is almost reversed in the floating point case (Figure 4.7b), where for most benchmarks ARM has smaller code size than X64. This phenomenon deserves attention, but it does not directly relate to RISC-V, and as such will not be explored further.
4.4 Compressed vs. Non-Compressed

Naturally, the next thing to ask is what the 64-bit results for the code size would look like if a compressed format like Thumb2 existed for 64-bit ARM instruction sets. To find out, we first build the benchmarks to target ARMv7-A without Thumb2 instructions (as explained in Chapter 3) to calculate how much Thumb2 manages to shrink the code size. We also need to be careful to subtract the space in the text section which is occupied by data for ARMv7-A, like we did in Section 4.3, in order to compare the code sizes.

Now is also a good opportunity to compare the RISC-V ‘C’ extension to the Thumb2 format to see which one accomplishes better code savings with respect to the base ISA. We summarise the results in Table 4.2 and observe that Thumb2 manages to pull down the code size by a remarkable 31%. On the other hand, the RISC-V ‘C’ extension has almost the same effect in the 32-bit and 64-bit case, at 26.4% and 25.6%, respectively, which is noticeably less performant than Thumb2. Moreover, the ‘C’ extension reduces the size of integer and floating-point benchmarks by almost the same amount (within 1.4%, while Thumb2 reduces the size of the integer benchmarks by 5% more than it does for floating-point, which leads us to believe that Thumb2 is more focused on its integer subset. Based on the small difference in code savings that the RISC-V ‘C’ extension achieves between 32-bit and 64-bit, it would be reasonable to assume that if a compressed format for ARMv8-A was to be implemented, then it could result in code savings close to the ones of Thumb2. In Table 4.3, we show how RV64C and ARMv8-A would compare in terms of code size if the latter had a compressed variant. We observe that ARM would produce 14.4% smaller code, which is even better than ARM’s performance in the 32-bit case, where it had 12% less code. However, these results are not very believable, as if we look closer at the table, the total code size for ARMv8-A is bigger than the code size for the integer and floating-point benchmarks individually. Far from a calculation error, this demonstrates that the two ARM architectures work differently and we could not just carry over Thumb2 to 64-bit and expect the same code compression. For this reason, it is impossible for us to tell whether ARMv8-A would have better code density than RV64C if a compressed format like Thumb2 existed for it. All in all, RISC-V really has the upper hand in this situation, because it was designed with the ‘C’ extension in mind and has already implemented

<table>
<thead>
<tr>
<th>ISA</th>
<th>Normalised Text Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>RV32GC</td>
<td>INT 1</td>
</tr>
<tr>
<td>Thumb2</td>
<td>0.82</td>
</tr>
<tr>
<td>IA32</td>
<td>1.184</td>
</tr>
<tr>
<td>RV64GC</td>
<td>1</td>
</tr>
<tr>
<td>ARMv8-A</td>
<td>1.295</td>
</tr>
<tr>
<td>X64</td>
<td>1.158</td>
</tr>
</tbody>
</table>

Table 4.1: Text section sizes, normalised with respect to RISC-V.
Chapter 4. Benchmark Results

(a) Integer benchmarks

(b) Floating-point benchmarks

Figure 4.6: Text section sizes per individual benchmark for the 32-bit architectures, normalised with respect to RISC-V32GC.

it, so that it is compatible with the non-compressed version and manages to shrink the code size by more than 25%. On the other hand, designing and implementing an extension like this for 64-bit ARM might prove to be infeasible, because of the greater complexity and irregularity of the ARM architecture and its design which perhaps lacked foresight that such an addition might have to be introduced. At any rate, it would be a huge investment, and ARM do not look like this is their main focus at the moment.
### 4.4. Compressed vs. Non-Compressed

#### (a) Integer benchmarks

![Sizes of Text Sections for SPECint](image)

<table>
<thead>
<tr>
<th>Code Size Reduction (%)</th>
<th>ISA</th>
<th>INT</th>
<th>FP</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>RV32GC</td>
<td>27.4</td>
<td>26</td>
<td>26</td>
<td>26.4</td>
</tr>
<tr>
<td>Thumb2</td>
<td>35.2</td>
<td>30.2</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>RV64GC</td>
<td>26.5</td>
<td>25.2</td>
<td>25.6</td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.2**: Code size reduction of compressed ARM and RISC-V formats with respect to the base ISA.

#### (b) Floating-point benchmarks

![Sizes of Text Sections for SPECfp](image)
Table 4.3: Code size comparison between RV64GC and ARMv8-A if a compressed version of ARM existed for 64-bit that managed to achieve the same code density enhancement as Thumb2.

### 4.5 CSiBE

CSiBE (short for GCC Code Size BEnchmark) is a benchmark suite developed for measuring code size. It is different from the SPEC2006 suite, which is all-around and does not focus on code size or execution time alone. We build and obtain results for CSiBE-v2.1.1, which we summarise in Figure 4.8 and Table 4.4. It is encouraging to see that the data for CSiBE kind of match our SPEC2006 results in Section 4.3 given we built the benchmarks in the same way. The ranking for the instruction set remains the same, although some of the numbers differ. The difference between RV32GC and Thumb2 is largely the same, but IA32 comes close to the other two. The most visible difference between the two suites is that for CSiBE, ARMv8-A is a much closer match for RV64GC than in SPEC2006. Overall, it is acceptable to have differences like this among various suites, so this difference is not an indication of a mistake.

![Total size of the CSiBE suite](image)

Figure 4.8: CSiBE raw results.

### 4.6 Instruction Frequencies

We calculate the instruction frequencies in the whole SPEC2006 benchmarks suite for the 6 instruction sets by parsing the disassembly files. The RISC-V results are presented in Figures 4.9 and 4.10, while the results for the other ISA's are left are in the appendix for completeness (Figures A.3 - A.6). They show the 20 most common
instructions as well as their "coverage", which represents the proportion of all instructions that the 20 most common ones take up. We include directives to allocate data such as .word, which are not technically instructions but are part of the text section and it is interesting to look at how their frequencies compare to the other instructions. These results do not distinguish between compressed and non-compressed instructions, i.e. the RV32GC instructions lw and c.lw (which is the compressed variant of lw and takes only 2 bytes) both contribute to the lw instruction count. The first reason is that this is how the objdump command reports it, and, although it would be possible to tell if the compressed version was used from the space that the instruction takes (objdump has this information), it would involve substantially more work to split the compressed and non-compressed instructions. This is not the focus of this section, and compressed instructions are discussed in Section 4.4. The second more important reason is that in presenting the instruction frequencies, we aim to present the instructions with the same functional behaviour together. For example, ideally we would consider lw (load word) and ld (load double) together. In the same way, we would have groups for conditional branches, ALU immediate and ALU register-register operations, etc. We do not do this here, as it would require fine-grained treatment for each instruction set, but might consider doing it in the second part of the project if it might lead to a significant result.

If we focus on RISC-V32GC (Figure 4.9), we notice that the 20 most common instructions account for 91% of all the instructions. Loads are the most common single instruction but add and addi together are more common. They are followed by stores and we notice there are moves, conditional and unconditional branches and some other arithmetic operations. These simple operations are the most common across all instruction sets [24] so the results are not surprising. RISC-V64GC has the same most frequent instruction classes (Figure 4.10), but uses instructions that take 32-bit and 64-bit operands, such as lw and ld, which means that the loads are spread between two instructions. On average, this reduces the frequency of any single instruction and results in a smaller coverage. Still, if we were to aggregate the most common instruction classes together, we would expect RV32 and RV64 to have almost the same instruction frequencies.

The IA32 and X64 ISA’s (Figures A.3 A.4) utilise a lot of move instructions because they have fewer registers than RISC-V and ARM and they overwrite the source
operand, so its value has to be copied if it is to be used later. These are special features of the Intel ISA, but other than that, the most common instructions are similar. As far as ARM is concerned (Figures A.5, A.6), loads and stores are most common, followed by the same instruction classes as in RISC-V, which is to be expected given ARM is somewhat RISC as well. An interesting difference is that the .word assembler directive is the 5th most common ARM “instruction”, but this is more pertinent to the compiler than the instruction set.

### 4.7 Average Instruction Encoding Length

From our discussion about RVC and Thumb2 in Section 4.4, we found that Thumb2 manages to shrink the code more, so we expect that the average instruction length of Thumb2 is smaller than RV32GC. We summarise the average instruction lengths per instruction set in Table 4.5 and show the results per the individual benchmarks in Fig-
If we focus on RISC-V, the average instruction length for 32-bit and 64-bit is just under 3 bytes which suggests that slightly more than half of the instruction used the compressed format. The 64-bit comparison is not very interesting, as ARM has fixed-length instructions, and X64’s CISC nature means that its instructions are typically longer, so RISC-V wins easily. In the 32-bit domain, our expectations are confirmed that Thumb2 manages to compress more instructions than RISC-V and brings down the average instruction length to 2.74 bytes, compared to RISC-V’s 2.95 bytes. Considering only the efficacy, Thumb2 is better all-round than RVC as it has better results both for the integer and the floating-point benchmarks, but there are other factors that need to be considered such as ease of implementation, complexity, compatibility, extensibility and cost where RISC-V’s ’C’ extension is claiming to be better than ARM’s Thumb2.

<table>
<thead>
<tr>
<th>ISA</th>
<th>INT</th>
<th>FP</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>RV32GC</td>
<td>2.87</td>
<td>2.97</td>
<td>2.95</td>
</tr>
<tr>
<td>Thumb2</td>
<td>2.56</td>
<td>2.77</td>
<td>2.74</td>
</tr>
<tr>
<td>IA32</td>
<td>3.18</td>
<td>3.78</td>
<td>3.68</td>
</tr>
<tr>
<td>RV64GC</td>
<td>2.94</td>
<td>2.98</td>
<td>2.97</td>
</tr>
<tr>
<td>ARMv8-A</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>X64</td>
<td>3.86</td>
<td>4.85</td>
<td>4.57</td>
</tr>
</tbody>
</table>

Table 4.5: Average instruction length.

4.8 Instruction Count

Instruction count is an important metric because it can show the expressive power of an instruction set. Typically, an instruction set that uses up fewer instruction for the same task would be preferred but we need to make sure that the instruction sets are comparable. CISC architectures have instructions which accomplish more tasks but are long. Thus, typically X64 would have fewer instructions than RV64, but that does not mean that X64 has better code density, because, as we saw, its instructions are longer to compensate. In our case, it is interesting to compare the instruction counts of ARM and RISC-V. This allows us to compare the two instruction sets without taking into account the compressed variants - RVC and Thumb2.

The summarised results can be seen in Table 4.6 and the individual benchmark results are available in the appendix (Figures A.9 - A.12). In calculating the instruction counts, we exclude the directives like .word, because they are for data. It is good to see that RV32GC has almost the same instruction count as IA32, since the latter is a CISC ISA for which it is common to have fewer more complex instructions. Still, it falls behind Thumb2, which generates 4.8% fewer instructions. In the 64-bit case, RV64GC is 12.5% behind X64 which is not terrible or unexpected given X64’s CISC nature. What is more important is that it is worse than ARMv8-A by 8.4%.
## 4.9 Summary and Evaluation

We compared RISC-V32GC with Thumb2 and IA32 as well as RISC-V64GC with ARMv8-A and X86 in terms of code size, instruction frequencies, average instruction encoding length, instruction count. We also evaluated the quality of the RISC-V 'C' extension versus the Thumb2 compression variant. We found that in the 32-bit case Thumb2 is slightly ahead of RV32GC on all of these statistics while IA32 is behind its counterparts on most of them. Overall, the magnitude of the discrepancy between RV32GC and Thumb2 in code density is not negligible but could, for the most part, be overcome by enhancing the instruction count or the instruction length of RISC-V. We propose ways to do this in Chapters 5 and 6. When considering 64-bit architectures, RV64GC generates less code and had shorter instructions than the other ISA’s, but ARMv8-A emits fewer instructions than RV64GC. One of the reasons for this is that ARMv8-A does not compress its instructions. Although we tried, it was not possible to predict how a compressed format would affect ARMv8-A.

All things taken into account, RISC-V’s relative performance is quite impressive, although it has the benefit of hindsight. Given how much shorter than the other architectures it has been around and how much less money has been invested in it, it would not be surprising if it could surpass ARM and X86, especially since more companies and researchers are taking part in developing it. Nevertheless, static measures are only one side of the coin and we could not draw definitive conclusions until we have looked at dynamic performance.

<table>
<thead>
<tr>
<th>ISA</th>
<th>Normalised Instruction Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INT</td>
</tr>
<tr>
<td>RV32GC</td>
<td>1</td>
</tr>
<tr>
<td>Thumb2</td>
<td>0.917</td>
</tr>
<tr>
<td>IA32</td>
<td>1.069</td>
</tr>
<tr>
<td>RV64GC</td>
<td>1</td>
</tr>
<tr>
<td>ARMv8-A</td>
<td>0.943</td>
</tr>
<tr>
<td>X64</td>
<td>0.882</td>
</tr>
</tbody>
</table>

Table 4.6: Normalised instruction counts.
Chapter 5

Prologue and Epilogue Improvement

5.1 Overview

The goal of our proposed improvements is to enhance the code density of RISC-V. There are two factors that make up the code size of a program: instruction count and average instruction length. In other words:

\[ \text{CodeSize} = \text{InstructionCount} \times \text{AverageInstructionLength} \]  \hfill (5.1)

Consequently, we need to somehow reduce at least one of the factors. Let us turn our attention to RISC-V32GC and Thumb2, and their instruction count (Figure [A.9]) and instruction length (Figure [A.7]). We observe that RISC-V32GC is behind on both statistics, which then, when multiplied, contribute to the larger difference in code size than the individual difference for both of them (Figure 4.6).

If we decide to focus on the average instruction length, there are a few things we could explore in order to decrease it. Firstly, we could try to improve the 'C' extension by looking at all instruction frequencies and identifying if there was an instruction A whose compressed variant was used less often than another instruction B for which there does not exist a compressed version. If this was the case, we could just remove A's compressed encoding and utilise the free opcode for B. However, the concept of a compressed variant has been studied quite well and RISC-V's 'C' extension was designed with the most common instruction and operands in mind [13]. Indeed, there already exist compressed formats for loads, stores and most ALU operations which make up the majority of the RISC-V instructions (Figures [4.9] and [4.10]). Therefore, it is quite hard to imagine reaping huge code savings by redesigning the compressed extension of RISC-V. That being said, we perform some similar analysis later when we introduce potential new instructions in order to find out if they would be used more often than a compressed instruction. There is another unthinkable approach to tackle instruction length and this is introducing variable-length instructions. It would be unreasonable to take on this ordeal as it goes against the design principles of RISC-V or the RISC philosophy in general. Although it would result in better code density, it would make the decoding logic a lot more complicated and thus more expensive. For
these reasons, our proposed improvements to the RISC-V instruction set aim to reduce the instruction count, rather than shorten the average instruction.

5.2 Motivation

Going back to instruction frequencies again, loads are the most common single RISC-V instruction (Figures 4.9 and 4.10). One of the most pervasive principles in instruction set and computer design, outlined in Chapter 1 of “Computer Architecture: A Quantitative Approach” [24], is to focus on the common case. The authors of the book use this fundamental to explain that optimising the fetching and decoding stage of a processor would be preferable to making, for example, the floating-point multiplier better because the former is used tremendously more often and is also much simpler. Although ours is a different context, refining the common case applies here as well. In particular, there are many more scenarios in which load instructions are used than another niche instructions, so it is natural to start off our attempt to enhance RISC-V by researching ways to emit fewer loads. One case in which load instructions are used repeatedly is the function epilogue. In Figure 2.4, we show what a typical function epilogue might look like for RISC-V32GC and see that there could be up to 12 load instructions per epilogue if all callee-saved registers need to be restored. As pointed out in Section 2.5 the structure of the stack frame is normally defined quite rigidly in the ABI, which allows us to describe each epilogue with a few parameters such as how many registers need to be restored, how much space on the stack needs to be deallocated, etc. Similarly, the same could be done for the function prologue and other commercial sets like ARC provide functionality to set up or destroy the stack with 1 instruction. Therefore, the purpose of designing new prologue and epilogue instructions for RISC-V is to replace instruction sequences like the ones in Section 2.6 with just 1 instruction each. Although we have mentioned just general-purpose registers, analogous prologue and epilogue instructions could be designed for FP registers as well. Initially it might seem surprising that RISC-V does not already define such instructions but the RISC-V designers give good of reasons for why this is the case [20]. In this literature, such instructions are known as store-multiple and load-multiple. For the time being, we ignore these arguments, but we go back to them in the evaluation part (Section 5.6).

5.3 Expected Code Size Reduction

Before proceeding to implement the prologue and epilogue instructions, we estimate how much code density enhancement they would yield. This is an important step, because introducing new instructions entails adding additional decoding logic, possibly more functional units which might raise the cost of the processor more than the improvement could justify.
5.3. Expected Code Size Reduction

5.3.1 Considerations and Assumptions

In order to obtain the expected code size reduction, we first need to define our new instructions and determine what they are going to do. We take a greedy approach and assume that the new instructions can encode as much information as we want, effectively disregarding the fact that the instructions need to be either 2 or 4 bytes long. Later on, we explain why we can carry on with our assumptions in the real implementation.

From our discussion so far, we are able to identify 4 distinct scenarios in which we could use our new instructions:

1. Function prologue that saves general-purpose s-registers.
2. Function epilogue that restores general-purpose s-registers.
3. Function prologue that saves floating-point s-registers.
4. Function epilogue that restores floating-point s-registers.

The simplest thing to do would be to introduce 4 new instructions each corresponding to one of the above. While this is efficient for saving general-purpose registers, we would need 2 instructions in the function prologue and another 2 in the epilogue to store/restore both the GP and the FP registers. To help this, we could have 1 prologue instruction and 1 epilogue instruction, which store/restore both GP and FP registers, but we want to have an instruction for cases when we have no FP registers, such as when we use the base ISA. Therefore, we have to settle for a middle ground and define the following instructions:

1. Integer prologue instruction, we call this enterint.
2. Integer epilogue instruction, we call this leaveint.
3. Integer and FP prologue instruction, we call this enterintfp.
4. Integer and FP epilogue instruction, we call this leaveintfp.

Our next consideration is what exactly these new instructions need to do. So far, we have been concerned mostly with saving and restoring s-registers, but as we saw in Section 2.6, the stack pointer needs to be modified and sometimes the return address register needs to be stored as well. If we include this functionality in the new instructions, we save another 2 instructions per prologue and epilogue. All things considered, this is what we assume that the new integer prologue instruction would do:

- Allocate space on the stack for the stack frame - this is done by subtracting the size of the stack frame from the stack pointer
- Store the necessary GP s-registers
- Store the return address on the stack

In addition to this, the floating-point prologue instruction stores the FP s-registers as well. The epilogue instructions reverse these actions by restoring the values of the registers and moving the stack pointer up.
5.3.2 Results

We are ready to present the shrinkage caused by the new instructions.

5.3.2.1 Instruction Count Shrinkage

The new instructions improve the code density by reducing the number of instructions generated. Although ultimately we care about the code size improvement in bytes, we also show how the prologue and epilogue instructions affect the instruction count, because, given equal instruction length, instruction count would be the determining factor which instruction set has better code density. It is especially relevant in the comparison between RISC-V32GC and Thumb2. For example, if our improvement manages to equalise the instruction counts between the two, then the difference in the code density would be a result of the quality of RISC-V’s ‘C’ extension versus ARM’s Thumb2.

Before we show the results, we elaborate on how they are obtained. Let us put down some definitions.

- **P** (for Prologues) is the number of prologues that have at least 1 instruction and is equal to the number of new enterint/enterintfp instructions that will be introduced.
- **INTPS** (for Integer Prologue Stores) is the number of store instructions that are part of instruction prologues and store a callee-saved GPR. They will thus be removed.
- **FPPS** is the same as INTPS but for callee-saved FP registers.
- **FPFlag** is a binary flag. 1 if we are calculating the shrinkage of enterintfp and leaventfp. 0 if we are calculating the improvement of enterint and leaveint.
- **ASP** (for Adjust Stack Pointer) is the number of instructions that allocate space on the stack for the stack frame. This could be an add or a subtract instruction. Because all prologues that save registers will have exactly 1 such instruction to subtract from the stack pointer, it follows that $ASP \equiv P$.
- **RS** (for RA Saved) is the fraction of prologues that save the return address register on the stack as well.

The formula is then as shown below and the factor of 2 accounts for the epilogues as there will be the same number of epilogues as prologues and they will have the same number of instructions.

$$\text{ImprovedInstructionCount} = \text{OriginalInstructionCount} + 2 \times P - 2 \times \text{INTPS} - F\text{FPFlag} \times 2 \times \text{FPPS} - 2 \times \text{ASP} - 2 \times \text{RS} \times P \quad (5.2)$$
5.3. Expected Code Size Reduction

The results summarised in Table 5.1 suggest that we can expect to emit around 6\% fewer instructions if we implement the `enterintfp` and `leaveintfp` instructions. This is an encouraging result and in Table 5.2 we present how this improvement is going to change the instruction count disparities between the instruction sets. (See Table 4.6 for the results we reported before.) The most noticeable achievement is that the new instructions cause RV32GC to have a lower instruction count than Thumb2 and bring RISC-V64GC and ARMv8-A within 2.4\% of each other from 8.4\% previously. This result has huge implications because it means that if we were comparing RV32G (no ’C’) with ARMv7-A (no Thumb2), RISC-V32G would have better code density with the new prologue and epilogue instructions.

<table>
<thead>
<tr>
<th>RISC-V32GC</th>
<th>Suite</th>
<th>INT</th>
<th>INT+FP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SPEC2006int</td>
<td>8.41%</td>
<td>8.66%</td>
</tr>
<tr>
<td></td>
<td>SPEC2006fp</td>
<td>4.89%</td>
<td>5.55%</td>
</tr>
<tr>
<td></td>
<td>SPEC2006</td>
<td>5.50%</td>
<td>6.08%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RISC-V64GC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Instruction count reduction as a result of adding prologue and epilogue instructions.

<table>
<thead>
<tr>
<th>ISA</th>
<th>INT</th>
<th>FP</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>RV32GC</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Thumb2</td>
<td>1.004</td>
<td>1.016</td>
<td>1.014</td>
</tr>
<tr>
<td>IA32</td>
<td>1.170</td>
<td>1.098</td>
<td>1.109</td>
</tr>
<tr>
<td>RV64GC</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ARMv8-A</td>
<td>1.021</td>
<td>0.959</td>
<td>0.976</td>
</tr>
<tr>
<td>X64</td>
<td>0.955</td>
<td>0.923</td>
<td>0.932</td>
</tr>
</tbody>
</table>

Table 5.2: Estimated instruction counts per instruction set with the new enterintfp and leaveintfp instructions, normalised with respect to RISC-V.

5.3.2.2 Text Section Size Shrinkage

The way we calculate the reduction in the code size is very similar to the method of calculating the instruction count decrease. The only difference is that we need to take into account the length of the instruction we are adding and subtracting. The formula
is as follows:

$$\text{ImprovedCodeSize} = \text{OriginalCodeSize} + 2 \times 2 \times P$$

$$- 2 \times SL \times \text{INT PS}$$

$$- FPFlag \times 2 \times FPSL \times FPPS$$

$$- 2 \times 2 \times \text{AddP}$$

$$- 2 \times RS \times 2 \times P$$

(5.3)

\(SL\) (for Store Length) and \(FPSL\) (for FP Store Length) represent the average length of stores that are part of a prologue, which operate on GPR or FP registers, respectively. The length depends on the size of the offset and, because loads and stores have the same format, we expect loads to have the same length. As stated in our assumptions in the above section, our prologue and epilogue instructions we assume to be 2 bytes long. The last thing to mention is that the instruction used to modified the stack pointer is 2 bytes long i.e. \(c.\text{addi}\) or \(c.\text{subi}\), which is reasonable as the offset range of these instructions is sufficient for most practical purposes.

Before we comment on the results, let us put forward that we expect the improvement in code size to be smaller than the one we showed for instruction count. This is due to the fact that the instructions that are part of prologues and epilogues (adds, stores, and loads) have a compressed version, which makes them on average shorter than the average instruction length. For the instruction count, each instruction has the same weight, but here removing a 2-byte instruction has a smaller effect than removing a 4 byte instruction.

We can see how the code density is affected by the new prologue and epilogue instructions in Table 5.3. On average, we conclude that integer benchmarks benefit more than FP benchmarks. An interesting observation is that although the new instructions have a slightly bigger effect for 64-bit RISC-V in terms of instruction count (See Table 5.1), they achieve better code size reduction on the 32-bit ISA. The biggest reason for this is that the load and store instructions, which are part of epilogues and prologues, are shorter on average for 64-bit RISC-V than the 32-bit ISA. In turn, this is a bit surprising because \(c.\text{lDSP}\) and \(c.\text{lwSP}\) have the same offset range in terms of doubles or words, respectively. One possible reason might be the data types in the benchmarks. For example, if local variables in the 64-bit instruction set are 32-bits wide, then we will be able to use the compressed load instructions more than for RISC-V32GC as the offset needed would be smaller in terms of doubles.

Table 5.4 shows how the text section size of RISC-V would compare with ARM and X86 if the prologue and epilogue instructions existed. (See Table 4.3 for the results with the original RISC-V ISA.) RISC-V32GC is noticeably closer in code density to Thumb2 and only falls behind by 6.7%, down from 11.7%. For 64-bits, RISC-V pulls even further ahead of the other two instruction sets.

Overall, our expected code density improvement resembles the results that Waterman reports [20]. For example, he reports a reduction of 8% in the size of the Linux kernel, while we observe 7.79% decrease for 32-bit integer benchmarks. In Figures 5.1 and 5.2 we also attach the code density improvement for each individual benchmark. The first observation we make is that the relative reduction between the 32-bit and 64-bit case is very similar per individual benchmark. This should be the case, as we do not...
5.4 Design and Implementation

We have found that our new proposed instructions could yield a code size improvement of more than 5%, which is considered quite significant. This gives us enough basis to begin designing and implementing the prologue and epilogue instructions in the actual instruction set. So far, we have outlined the logical behaviour of the instructions where we make a lot of assumptions about how much they can do. In reality, this is limited by the width of the instruction, and in this section we transition from the functional description of the new instructions to an actual design and implementation.
Chapter 5. Prologue and Epilogue Improvement

5.4.1 Compressed Instruction Format

In Section 5.3, we presumed that our instructions could be encoded in 2 Bytes. Unfortunately, there are no free opcodes in the ‘C’ extension, which makes it impossible to implement the new instructions as 2 Bytes. The only seemingly free opcode is 11 but it is reserved for 32-bit instructions [14]. Despite this obstacle, the improvement in code size is still very appreciable even if we assume all new instructions to be 4 Bytes. For reference, the improvement to RISC-V32GC drops from 5.32% (Table 5.3 to 4.8%, which is a fall of less than 10%. For RISC-V64GC, the decrease is from 4.6% to 4.06%.

But now there are new, more interesting implications of encoding the new instructions in 32-bits. First, we run the risk of making the code size bigger if we are not careful. Let us imagine a prologue that does not save any callee-saved registers and the only thing it does is allocate some space for local variables by changing the stack pointer. Such a prologue would have one add/subtract instruction, which in most cases would be compressed to look like c.addi4spn rd, sp, offset. Thus, this prologue takes 2 Bytes, whereas our new instruction would take 4 Bytes. This can be dealt with in the compiler by adding some extra logic which checks if the new prologue instruction...
5.4. Design and Implementation

(a) Integer benchmarks

(b) Floating-point benchmarks

Figure 5.2: Effect on code density of enterintfp and leaveintfp for RISC-V64GC.

should be emitted or just the add instruction. The same is relevant for epilogues.
Next, it is valuable to check if there are any instructions in the compressed extension
that are utilised less than the new prologue and epilogue instructions would be used.
If this was the case, then we could exclude such instructions from the 'C' extension
for the benefit of adding the new instructions to it. The obvious potential candidates
are the load and store instructions whose number we are going to reduce with the new
dedicated prologue and epilogue instructions. These are:

- For RISC-V32GC: C.LWSP, C.SWSP, C.FLDSP, and C.FSDSP. However, we also
  consider C.FLWSP and C.FSWSP, because they are load/store instruction although
  their count would not change.

- For RISC-V64GC: C.LDSP, C.SDSP, C.FLDSP and C.FSDSP. Similarly, we also
  consider C.FLWSP and C.FSWSP.

Figures 5.3 and 5.4 show, for 32-bit and 64-bit RISC-V respectively, how the number
of prologue and epilogue instructions relate to the frequency of the above instructions.
Let us analyse Figure 5.3 first. The blue bars represent the number of instructions that
remain after eliminating the instructions that would be part of a prologue or epilogue.
For example, the C.LWSP bar excludes the loads that would be part of an epilogue. Of course, C.FLWSP and C.FSWSP do not appear in prologue or epilogues because FP registers are 64-bit wide, so we show their original frequencies. As a result, the way to read the graph is that if the green bar is bigger than a blue bar, then we would achieve better code density by implementing the new prologue or epilogue instructions in the place of the blue instruction. In this case, we see that there are fewer C.FLDSP, C.FSDSP, C.FLWSP and FSWSP instruction than prologues. It would make the most sense to replace C.FLWSP and C.FSWSP in the compressed instruction extension because they have the smallest frequency.

The 64-bit graph is constructed in an analogous way. Indeed, the only difference is that in prologues and epilogues we now eliminate C.LDSP and C.SDSP. Therefore, we conclude that it would be better from a code size point of view to replace C.FLDSP and C.FSDSP in the 64-bit ‘C’ extension.

![Figure 5.3: Compressed load/store frequencies vs. prologue/epilogue frequencies for RISC-V32GC.](image)

![Figure 5.4: Compressed load/store frequencies vs. prologue/epilogue frequencies for RISC-V64GC.](image)

We need to mention that these frequencies we present are highly dependent on the extensions that we use. In particular, if we use just the ‘F’ extension without the ‘D’
extension, then we would use C.FLWSP and C.FSWSP to restore and save registers, respectively, because FP registers would be 32-bits wide. In that case we would expect to have more FP loads and stores that operate on words rather than doubles and it might no longer be efficient to replace C.FLWSP and C.FSWSP in the 32-bit case, but C.FLDSP and C.FSDSP. We could explore this further by building the benchmarks without the 'D' extension, but with the available time we do not. If we want to be thorough, we would have to compare the number of prologues and epilogues with the frequencies of all compressed instructions. This would allow us to make the best code size optimisation, but we are not going to go there with the amount of time and instead decide to focus on what a possible encoding of the enterintfp and leaveintfp instructions might look like. Figures 5.5 and 5.6 show a possible encoding of the instructions. We decide to replace C.FLWSP for enterintfp and C.FSWSP for leaveintfp. The instructions have the following properties:

- Inst[1:0] is the opcode. Inst[15:13] is an addition to the opcode to decode the instructions.
- Inst[5:2] represents how many callee-saved FP registers need to be saved/restored. There are 12 such registers so we need 4 bits.
- Inst[6:9] represents how many of the 12 general purpose s-registers need to be saved/restored.
- Inst[11:10] is a zero-extended offset, scaled by 4 or 8 depending on whether we are using 32-bit RISC-V or 64-bit, respectively. This offset is additional space for local variables on top of the space necessary for saving the gp/fp registers on the stack. The range is then 0-3 words or doubles.
- Inst[12] is a binary flag that specifies whether the return address register needs to be saved on the stack.

![Figure 5.5: Compressed enterintfp format](image)

![Figure 5.6: Compressed leaveintfp format.](image)

These instructions do not use any of the defined formats for the 'C' extension [14], because they are different in behaviour from the other instructions. Hence, they would require extra decoding logic. Additionally, these instructions seem to have a very small range for the offset. This is indeed true and 0-3 words or doubles might not be enough. In that case the instructions would have to be complemented with another add instruction that would allocate more space on the stack and use 2 Bytes. Because the bits for the offset, for the floating-point registers and for the general-purpose registers
need to sum up to a constant, there can be trade-offs to make the one of them bigger on behalf of another one. Still, it is the most efficient to have a smaller range for local variable space than to have fewer bits for floating point registers, or general-purpose registers. Supposing we remove one bit from \textit{num\_fp\_regs}, we would only be able to save/restore at most 7 FP registers. Thus, a prologue that needs to save all FP registers would need 5 more store instruction in addition to the prologue instruction.

But there is room for yet another optimisation, because there are only 12 callee-saved registers and 4 bits allocated to them. So values from 13-15 would be invalid. We could make use of this when we want to save all 12 callee-saved registers by making value 13 mean that we need to save all 12 callee-saved registers and allocate 1 more word for other variables. Analogously, we can do that for 14, 15 and allow for 6 more words or doubles, resulting in a total offset range of 9 words/doubles.

If we are not using the 'F' or 'D' registers, then it might be worth it to implement \textit{enterint} and \textit{leaveint} in the compressed extension, because there would be no floating-point registers. This would free up another 4 bits for the offset and raise the range to $2^6 - 1$ words/doubles, which should suffice for most functions.

### 5.4.2 Implementation

Now that we have established that it is not possible to implement the new instructions in 2 Bytes, let us look at how we could implement them in 4 Bytes. In Figures 5.7 and 5.8 we show the format of \textit{enterint} and \textit{leaveint}. They follow the U-instruction type and so do not require defining an additional format. They have the following properties:

- \textit{Inst}[6:0] is the opcode. U-type instructions are only distinguished by the opcode, so we need 2 free opcodes.
- \textit{Inst}[10:7] is the number of general purpose s-registers to save/restore.
- \textit{Inst}[11] is a binary flag that indicates whether the return address register has to be saved.
- \textit{Inst}[31:12] is a zero-extended immediate that represents the additional space for local variables that allocate/deallocate on the stack. It is scaled by 4 or 8 depending on our register width so the effective range is then $2^{20}$ words or doubles, which is excessive and enough for any kind of realistic application.

![Figure 5.7: Enterint format.](image)

The huge offset range is indeed not going to be utilised in most cases probably. This is why it might be reasonable to use the I-type instruction format for these two new instructions. Although it would shrink the offset range to $2^{12}$ words/doubles, this would still be enough for all applications and, more importantly, it would allow us to save
5.4. Design and Implementation

some opcode space, as we would be able to use the same opcode for the two instructions and distinguish them by the Inst[14-12] field.

Figures 5.9 and 5.10 show the encodings that we use for the enterintfp and leaveintfp instructions. They follow the I-type instruction format and the fields are interpreted in the same way as above, except we have a new field - Inst[19:15] specifies how many FP s-registers are to be saved/restore. What is more, the size of the offset changes to $2^{12}$.

5.4.3 Implementation Pipeline

We have developed the model for our instructions in terms of their format and behaviour and in order to start executing them and testing them we need to complete a couple of steps:

1. Add the new instructions to the GNU assembler so that they can be recognised.
2. Define the behaviour of the new instructions in a simulator. In this case we use Spike, the RISC-V functional simulator.
3. Modify the compiler, in our case the RISC-V port of GCC, to emit the new instructions in the appropriate case.

The first step formally defines the format of the instructions and it is essential to translate from the mnemonic of the instruction to its binary encoding. The end result is that we are able to use the new instructions in extended assembly in C to compile programs that use them. We have completed this step fully, and Figure 5.11 shows the way in which the mnemonics are constructed. Given the type of the instructions (discussed in Section 5.4.2), the assembler expects the first argument of enterint and leaveint as well as the first two arguments of enterintfp and leaveintfp to be a register number. Because the bit that indicates whether we have to store/restore the return address register is the
MSB of the first argument of the instructions, we have to add 16 to the first argument. This slight inconvenience could be alleviated by defining new instruction types in the assembler, but we do not do that because the U- and I-types work well enough for our purposes. To summarise Figure 5.11, enterint and leaveint save/restore $28 - 16 = 12$ s-registers, save the return address and allocate 30 words of space for on the stack. Enterintfp and leaveintfp save 11 GP s-registers, but not the return address register, 12 floating-point s-registers and allocate 50 words on the stack.

```
enterint x28, 30
leaveint x28, 30
enterintfp x11, x12, 50
leaveintfp x11, x12, 50
```

Figure 5.11: Usage of the new prologue and epilogue instructions.

To complete the second step, we follow the guidelines in the Spike repository to add a new instruction. As an example, Figure 5.12 displays how the enterint instruction is integrated within Spike. After accomplishing the first two steps, the end result is that we are able to execute and test programs which use the prologue and epilogue instructions.

The third step is needed to test the real impact on code density of the new prologue and epilogue instructions. However, it is not strictly necessary to use and verify the instructions. As we mentioned, we could use extended assembly for this. Given the available time, complexity of the GCC software and the learning curve of its architecture, we decided not to fully complete this step. So far we have managed to identify that we would need to modify the `riscv_expand_prologue()` and `riscv_expand_epilogue()` functions in `riscv-gcc/gcc/config/riscv`. The biggest challenge is that these procedures make calls to functions defined in other files and this causes a chain reaction. The high learning curve of the GCC architecture likely would have caused dedicating disproportionate time on becoming familiar with GCC to prevent us from thoroughly testing the new instructions, considering other improvements and describing them well in this report. Nevertheless, some GCC expertise will be crucial for the second part of the project (see Section 6.2) and acquiring it is definitely on the agenda.

### 5.5 Verification

The verification is the last step we perform to conclude the implementation of the prologue and epilogue instructions. In order to test their behaviour we develop a script that could be used more generally to test any functionality in the Spike simulator. The Spike simulator permits us to inspect the contents of the registers at any given time, but does not have any way of assigning "expected" values that the registers need to have after the program terminates. The Python script we wrote allows us to compare
5.5. Verification

Figure 5.12: Enterint implementation.

```c
uint64_t ra_saved = insn.x[11, 1];
uint64_t nr_regs_saved = insn.x[7, 4];
uint64_t offset = insn.x[12,29];
uint64_t mult;

// Allocate space on stack
if (xlen == 32) {
    mult = 4;
} else {
    mult = 8;
}
offset *= mult;
WRITE_REG(X_SP, sext_xlen(READ_REG(X_SP) - zext_xlen(offset)));

// Save return address at top if needed
if (ra_saved) {
    if (xlen == 32) MMU.store_int32(READ_REG(X_SP) + offset, READ_REG(X_RA));
    else MMU.store_uint64(READ_REG(X_SP) + offset, READ_REG(X_RA));
    offset -= mult;
}

// Save callee-saved regs
if (xlen == 32) {
    for (uint64_t reg = 8; reg < 10 && reg < 8 + nr_regs_saved; reg++) {
        MMU.store_int32(READ_REG(X_SP) + offset, READ_REG(reg));
        offset -= mult;
    }
    for (uint64_t reg = 18; reg < 18 + nr_regs_saved - 2; reg++) {
        MMU.store_int32(READ_REG(X_SP) + offset, READ_REG(reg));
        offset -= mult;
    }
    else {
        for (uint64_t reg = 8; reg < 10 && reg < 8 + nr_regs_saved; reg++) {
            MMU.store_int32(READ_REG(X_SP) + offset, READ_REG(reg));
            offset -= mult;
        }
        for (uint64_t reg = 18; reg < 18 + nr_regs_saved - 2; reg++) {
            MMU.store_int32(READ_REG(X_SP) + offset, READ_REG(reg));
            offset -= mult;
        }
    }
}
```

the state of the register file at the end of the execution with a file that contains these expected values. We leave out the implementation details, but the only manual work that we have to carry out, in addition to running the simulator, is to set the expected values of the registers. It is hard to imagine how the testing process could be automated much further, because each test would have different expected register values at the end of execution. These would always need to be specified by the tester. An advantage of this method of verification is that it is not limited to the prologue and epilogue instructions and could be extended to any new functionality that we want to verify, which makes it suitable for use by anyone who desires to implement new or alter existing instructions.

Figure 5.13: Output of running the verification script.
The output from running two tests is shown in Figure 5.13. The test name and result is shown and when a test does not pass, the framework reports which registers did not match their expected value.

Next, we describe the specific tests that we designed for the prologue and epilogue instructions. The test that we use for enterint and leaveint is shown in Figure 5.14. Its structure is as follows:

1. Put the value 1 in all s-t-a registers. Lines 10-37.
2. Save the value of all s-registers and the return address on the stack. Line 39.
3. Put the value 0 in all s-t-a registers. This step is important to make sure that we are changing the value of the registers in between the prologue and the epilogue. Otherwise even if our instruction did not do anything the test would pass. Lines 41-68.
4. Restore the value of all s-register and the return address from the stack. Line 70.

With this specification, we expect that at the end of the program, s-registers would have value 1, while the t-a-registers would have value 0. We carry out an analogous test for enterintfp and leaveintfp, with the difference being that we include the FP registers as well.

5.6 Evaluation

We saw that the new prologue and epilogue instructions that we implement reduce the code size quite substantially. For RISC-V32GC, the shrinkage is about 8% on average for the SPECint, and about 5.3% across the whole suite. These results provide the same functionality and achieve better code savings than the millicode approach employed by RISC-V. Millicode is a software technique that can be utilised to reach similar behaviour, which is explained in further detail in Section 5.6 in Waterman’s paper [20]. Unfortunately, it has the drawback that it can lead to a drastic increase in dynamic instruction count. Waterman reports an average increase of 3% in dynamic instruction count, with only a 4% reduction in code size. However, there is a lot of variance. For benchmarks which have a lot of function calls like omnetpp and perlbench, the rise in dynamic instruction count can go up to 18%. This is not ideal, because we expect to have the greatest impact on programs that do make a lot of function calls because then we would be able to utilise the optimisations a lot more often. What is more, ideally we would want our static improvement not to be a detriment to dynamic performance. Another drawback to the millicode approach is that the new instructions occupy space in the instruction cache, which might decrease its hit rate and further decrease dynamic performance. This could be helped by hard coding the millicode instructions in the cache and their address in a table that each prologue or epilogue could then inspect. However, this introduces additional coupling between the instruction set and the microarchitecture, which it is desirable to avoid, as Asanovic [1] points it out as one of the big weaknesses of OpenRISC. It is reasonable to believe that our proposed improvement and the way in which we implement the instruction will not
5.6. Evaluation

Figure 5.14: Assembly code to test the behaviour of enterint and leaveint.

hurt the number of dynamic instructions so badly, because the new instructions could just be expanded into the old ones. However, there are a couple of intricacies that Waterman discusses which lead us to believe that the transition to a real implementation might not be as smooth. In fact, store-multiple and load-multiple instructions are well-understood and it is unlikely that the RISC-V designers decided not to have them in the instruction set. In particular, Waterman says that the most difficult choice in designing RVC was whether to include store-multiple and load-multiple instructions, while the manual and the paper provide the reasons for this decision \cite{20}. We cover the reasons they provide with how much expertise we have and relate back to our implementation. The italicised text is taken and adapted from Section 5.6 in \cite{20}.
- **Store-multiple and load-multiple instructions would violate the requirement that all RVC instructions need to have a 32-bit encoding.**

  This is relevant to implementing the instructions only as part of the compressed extension. Then, indeed it would be the case that the compiler and the decoder would have to be RVC aware. However, we show how to implement the instructions in 4 Bytes, so this constraint should only be an issue if we are running out of opcodes. As it stands, RISC-V has quite a wide "green" area of opcodes that allows us to occupy a few more without too much to worry about. "Green" area is used to refer to the free encodings in the instruction set, while the occupied encodings constitute the "brown" area.

- **The compiler would be unable to weave in code from the function body in the prologue and epilogue, reducing its ability to avoid dependencies.**

  This is caused by the fact that the prologues and epilogues, when implemented with store- and load-multiple instructions, would be a single instruction. Nevertheless, we could design a stage after the fetching stage that would expand the prologue and epilogue instructions to how prologue and epilogues with ordinary store and loads instructions, similarly to how RVC instructions are expanded and the decoder does not need to be aware of them. This would add additional cost to the cores and would not solve the software rescheduling problem, but at least some hardware rescheduling would be feasible.

- **Store- and load-multiple instructions would have low performance in superscalar approaches, prohibiting the issue of other instructions.**

  It is true that if we implement the new instructions naively we might reach a point where the processor would not be able to issue other instructions before the completion of the store-load-multiple instruction, but this problem could be mitigated with the same approach as above. What is more, there are usually no dependencies between the instructions in the prologue as they are mostly loads or stores operating on different registers and memory areas, so the only concern might be the memory bandwidth.

Another reason that is mentioned is that in virtual memory systems (essentially all modern machines), a page fault during the execution of these instructions might make precise exceptions harder and require a new restart mechanism. We do not have much expertise to comment. Finally, the millicode option is described as the 'final nail in the coffin' against the implementation of store- and load-multiple instructions. We have already discussed this issue above and underlined its disadvantages pertaining to the execution time.

Overall, introducing dedicated prologue and epilogue instructions yields a very reasonable code density enhancement and bring RISC-V very close to ARM in static code size. Some of the reason mentioned against having them are more related to the one-time cost of implementing them more than their infeasibility. Proof of this is that such functionality exists ARC. My humble opinion is that, in an attempt to catch up in code size, RISC-V might have to go in the direction of store- and load-multiple instructions eventually.
Chapter 6

Other Improvements and Future Plan

6.1 Other Improvements

In this section we list a couple of other ideas that would improve the code size of the RISC-V instruction set by reducing the number of instructions the compiler generates. However, they are not explored at the same level of detail as the prologue and epilogue instructions and are not implemented and tested as thoroughly, because of time constraints and their smaller expected effect.

6.1.1 Caller-saved Registers

In the same way that callee-saved registers need to be saved on the stack in the function prologue if it is going to modify them, caller-saved registers need to be stored on the stack before a function call if the caller is going to need them. Caller-saved registers also need to be loaded from the stack after the callee terminates. Figure 6.2 illustrates this concept. The function has to save a2, a3, t1 in lines 2-4 because it uses them in lines 12-13. It is important to note that the sub function only modifies t1, but not a2 and a3. In fact, it does not have to alter any of them, but main has no way of knowing that, so it has to act preemptively. Nevertheless, there is a fundamental difference between this scenario and the function prologue and epilogue case. Although there are only 7 general purpose t-registers, we could not use 3 bits to say how many registers we want to save like we would if we had 7 callee-saved registers in the prologue/epilogue situation. This is because we might need to store t1, but not t0. Hence, we would need to have a bit for each a-register and each t-register. Given 5 a-registers and 7 t-registers, we would need 12 bits and this would mean that such an instruction could not be encoded in 2 Bytes, as we would need a 5-bit opcode. Of course, there is no harm to instruction count if we say we want to save all caller-saved registers, but this is very detrimental to the execution time, because we increase the dynamic work we have to do.

We do not have a sure way of identifying which instructions would be eliminated if such new instruction to save a-t registers existed, but we see that in the 32-bit and
Chapter 6. Other Improvements and Future Plan

Figure 6.1: Example where saving caller-saved registers is necessary.

64-bit architectures, 5.3% and 3.5% of the stores save a-t-registers on the stack, respectively. From here, we need to make a couple of assumptions. Let us suppose that 20% of these stores make sure that the value of the caller-saved registers is not modified by a function call. If our improvement is able to reduce the count of these by half, then the total improvement on the code size would be 0.53% and 0.35% in the 32-bit and 64-bit ISA, respectively, but that would double because we would eliminate half of the corresponding loads as well, yielding a total enhancement of 1.06% and 0.7% to code density. Overall, this is a solid improvement given our assumption. If we wanted to achieve the same reduction of code size as the prologue and epilogue improvement, discussed in Chapter 5, we would have to get rid of 49% of these stores and loads for the 32-bit version, and 72% in the 64-bit case. Achieving such a result is very, very unlikely. Furthermore, implementing these new instructions would be more complicated than the implementation of the prologue and epilogue instructions. The conclusion we draw is that this improvement is worth investigating further, but would not be the first one to delve into.

6.1.2 Pointer Arithmetic

The next improvement that we propose could be utilised in cases where we need to access consecutive addresses in memory, such as when traversing an array.

```java
int[] arr = new int[10];
for (int i = 0; i < arr.length; i++) {
    arr[i]++;
}
```

Figure 6.2: Traversing an array in Java.

Figure 6.2 shows what this operation might look like in Java code, and in Figure 6.3 we present the corresponding assembly for RISC-V32GC. We assume that register
s0 holds the initial address of the array, while register s1 holds the address of the last element of the array. The idea of this improvement is to combine the load and add instructions in lines 2 and 5, respectively, in one instruction. To achieve this, we could introduce a new **ladd** I-type instruction which has the same mnemonic (except for the name) and behaviour as **lw** or **ld**, but also adds 4 or 8 to the base address, depending on whether we are using the 32-bit ISA or 64-bit ISA. The improved version of the program is in Figure 6.4a. The ladd instruction in line 2 increments s0 and brings about a change in the offset of **sw** in line 5. So far we have been concerned with complementing loads, but there is nothing stopping us to define a similar store instruction - **sadd**. This would make the enhancement applicable in more scenarios and would also make it unnecessary to change the offset in the store instruction. For completeness, Figure 6.4b shows the program using the sadd instruction. It is quite hard to gauge the reduction in code size that this new instruction would yield without changing the compiler, but we suspect that it would be less than the previous two improvements.

![Figure 6.3: Traversing an array in RISC-V assembly.](image)

![Figure 6.4: Traversing an array with the new ladd and sadd instructions.](image)

### 6.2 Future Plan

This section describes our plan to continue the project in the next academic year. There are two focal points. The first is running benchmarks to compare RISC-V with other
commercial instruction sets *dynamically*, which will likely give ideas for dynamic improvements. The second objective will be implementing all improvements in GCC to test their effect.

### 6.2.1 Dynamic Analysis

The focus of this part of the project has been on static measurements and, most of all, code density and how other metrics like instruction count and average instruction length affect it. But, as we discussed in Chapter [4] and the evaluation part of the prologue and epilogue instructions (Section [5.6]), static and dynamic statistics are correlated, positively or negatively. In order to conduct a conclusive study of RISC-V, we need to consider its dynamic performance as well. For this reason, next year we are going to carry out similar research to the one in Chapter [4] where we compare RISC-V to ARM and X86, but instead examine the architectures during execution, rather than statically. The plan is to do this in Gem5 as it offers support for all three instruction sets, while Spike can run only RISC-V and is just a functional simulator. A reason to use the same simulator is to try and isolate the instruction set as much as possible from the implementation and the microarchitecture.

Some dynamic statistics we intend to look at are number of cycles to terminate, cycles per instruction (CPI), dynamic instruction count (DIC), dynamic instruction frequencies, dynamic average instruction length, the minimum (data or instruction) cache bandwidth required, the cache hit/miss ratio and the average memory access time (AMAT). It is hard to predict what the results are going to be and therefore what improvements we will be able to propose and this is indeed the reason we are doing the research. A few potential issues we might be faced with are:

- The CPI is of RISC-V is higher than the other ISA's. This might be as a result of poor instruction scheduler failing to recognise functional, data or control dependencies between the instructions, or a lack of enough functional units in the processor implementation.

- The cache bandwidth that RISC-V requires might be lower than the other ISA’s, because RISC-V has more registers and does not have to store or load data from memory as often.

- RISC-V might have a higher miss rate because the compiler does not put any data in the text section, which causes data that is only accessed together with some code to miss in the cache.

As it will be impossible to evaluate the effect of our proposed dynamic improvements just by looking at the produced code, we will have to implement them in GCC and Gem5 to see their impact, which brings us to our next main goal for next year.
6.2.2 Implement in GCC

We intend to implement the proposed improvements from this part of the project in GCC, in order to test their real impact on code size and also make sure that they do not have any dynamic disadvantage like the millicode approach used in RISC-V at the moment to compactly implement prologues and epilogues. Furthermore, the improvements tailored to enhance the dynamic performance could only be tested dynamically so learning GCC and how to modify it is going to be key to carry on with the project. There are a couple of ways we could approach the implementation. We could totally modify what GCC does at the moment and implement our improvements, but a better solution would be to create new flags in the compiler so that it could be run with or without our propositions. This has the advantage that one build of the compiler could be used to see which version is better. More importantly, it allows us to compile different programs with different options, which is often necessary depending on the program nature. For example, as we saw, the millicode approach might cause excessive increase in the dynamic instruction count for programs which make a lot of function calls, but it is a lot less complicated to implement than our dedicated prologue and epilogue instructions. Hence, computers where performance is not crucial but cost is would have to use the millicode approach and compile for it as a result. As explained earlier, GCC is complicated so this is going to take some time but it will be a major focus of the project next year.
Chapter 7

Conclusion

In this part of the project we looked at RISC-V as an open instruction set, its performance compared to other commercial ISAs and how it can be improved. In Chapter 1 we discussed the issue of proprietary instruction sets dominating the market and how this can hinder the progress in the computer architecture field. Then, we introduced RISC-V as the solution to this problem by meeting the requirements of an open instruction set standard and by learning from previous such attempts.

Still, in order to become a more commercial ISA that can match ARM or X86, being good in theory is not sufficient. For this reason, in our project we used the SPEC2006 benchmark suite to examine RISC-V’s performance on static measures compared to ARM and X86. In Chapter 4 we looked at statistics such as binary size, debugging information, instruction count, and instruction frequency. We spent most of our time on code size. We found that 64-bit RISC-V emits much smaller code than its counterparts, which was expected as RISC-V’s ‘C’ extension drastically reduces the average instruction length while ARM does not have a variant of Thumb2 for 64-bit. What is more, in the 32-bit field RISC-V has better code density than IA32, but falls about 11-12% short of Thumb2. To mend this, we proposed new dedicated load- and store-multiple instructions, implemented them in the GNU assembler and simulator and also tested their correct behaviour in the Spike simulator. The prologue and epilogue instructions improved the code density by about 5.3% for RV32 and brought the difference between it and Thumb2 to less than 7%. This approach achieved better code size reduction than the millicode approach that RISC-V currently uses and does not have the drawbacks of the latter, but might be more complicated to implement. We also gave ideas where the instruction set might be further improved, such as new instructions to store and restore caller-saved registers.

Overall, we achieved significant code size enhancement but there is still more to be done in the second part of the project. We need to change the GCC port of RISC-V to emit the new instruction in order to be able to run benchmarks with the them and verify that there are no significant bad dynamic consequences. Furthermore, we will do more dynamic analysis of RISC-V which might reveal other strengths or weaknesses of RISC-V.
In my opinion, the results that we obtained favour RISC-V more than they do ARM and X86. RISC-V clearly has better code density in the 64-bit field, but even though it is behind Thumb2 in code size, we also need to take into account how much more time and money has been invested in the ARM architecture. Not to mention that the ARM ISA is much larger, more complicated and more expensive to implement than the RISC-V base. This should not be an excuse for RISC-V in the future for being ever so slightly behind commercial instruction sets, though. Of course, RISC-V’s goal is to be better than its counterparts and this is to be done by uniting the effort of people in the field and researchers to contribute to RISC-V. Indeed, with major companies like Google investing in RISC-V, the future definitely is going to see the rise of RISC-V and other open instruction sets.
Bibliography

[1] K. Asanovic, “The Case for Open Instruction Sets. 40 Years of Patterson Symposium. Saturday, May 7, 2016.” [https://youtu.be/Qq1nMNVPcRg](https://youtu.be/Qq1nMNVPcRg)


Appendix A

Graphs

Figure A.1: Sizes of SPECint loadable sections for the 64-bit ISA’s, normalised with respect to RV64GC.
Figure A.2: Sizes of SPECfp loadable sections for the 64-bit ISA’s, normalised with respect to RV64GC.

Figure A.3: IA32 instruction frequencies
Figure A.4: X64 instruction frequencies.

Figure A.5: Thumb2 instruction frequencies.

Figure A.6: ARMv8-A instruction frequencies.
Appendix A. Graphs

Figure A.7: Instruction lengths for SPECint for the 32-bit architectures.

Figure A.8: Instruction lengths for SPECfp for the 32-bit architectures.
Figure A.9: Instruction counts for SPECint for the 32-bit architectures.

Figure A.10: Instruction counts for SPECfp for the 32-bit architectures.
Appendix A. Graphs

Figure A.11: Instruction counts for SPECint for the 64-bit architectures.

Figure A.12: Instruction counts for SPECfp for the 64-bit architectures.