Analysing the Impact of RISC-V Compressed Instruction Extensions

Elijah Yang

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

The embedded processor market is larger than ever before. This growth necessitates innovation in task-specific processor architectures. RISC-V is an open source instruction-set architecture that has gained traction in recent years, now even competing with ARM, which has long reigned as the common choice for an embedded ISA. RISC-V's recent success is underpinned by its ethos of "bespoke" configurability for any system. This manifests itself as a wide range of possible extensions, each of which implement some new functionality. One of these functionalities is code compression for reduced memory and power consumption. For years, there was only one extension which implemented code compression, the standard RISC-V C extension, commonly referred to as RVC. In 2023, a new collection of code compression extensions, all prefixed with Zc, were ratified. These provide further code size optimisations and increased configurability. In this project, I attempt to analyse the compression impact of each of these extensions, and provide insight on which one is best suited to various application types. I also propose additions to RISC-V for improved compression of floating point intensive applications.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Elijah Yang*)

Acknowledgements

I dedicate this paper to my parents Michelle and Justin; my brothers Joshua, Noah and Isaac; and my flatmates Álvaro, Carlos, Harry, Natalia, and Julius. Thank you for getting me through this project with your continued love and support. I would also like to thank Dr Nigel Topham. Nigel guided this project with invaluable supervision, as well as inspiring my interest in computer architecture in the first place with his teaching.

Table of Contents

Chapter 1

Introduction

Embedded processors are task specific processors which are integrated into larger systems. They're usually smaller, cheaper and less powerful than their general purpose counterparts but are more varied in architecture due to the range of possible applications. Examples of embedded systems include consumer devices, medical devices, security devices, network devices, and control systems (automotive, aerospace, industrial etc). The global embedded processor market was estimated to be worth US$14.3 billion at the end of 2022 and is estimated to grow to over US$42 billion by 2033 with a compound annual growth rate of 10.4% [7]. The bounds of memory requirements are typically more known for embedded systems than general purpose computers because they run preinstalled firmware rather than unknown software. Due to this, one effective technique for reducing the production cost of an embedded system is to store programs and program data in a compressed format. This allows the devices to be produced with smaller and cheaper memory and storage components. It also reduces the operational energy consumption by increasing the efficiency of read/write operations [2]. Motivated by this, methods to compress stored programs and program data have been a busy area of research since the early 1990's.

Reduced Instruction Set Computer (RISC) Is the name given to a strategy for computer architecture design which started gaining popularity in the early 1980's [18]. Due to a wide range of designs labelled as RISC, the exact definition of RISC is somewhat unclear but the most consistent characteristics across RISC machines are:

- Simple instruction operations
- A fast, hierarchical memory architecture
- Instruction pipelining
- Fixed instruction length
- High data throughput

RISC-V is an Instruction Set Architecture (ISA) which, as the name suggests is intended for a RISC machine. RISC-V provides a base ISA implementing what is deemed as the bare minimum functionality. The ISA can be configured to be suit-

able for a wide variety of applications with many standard ISA extensions. This aspect of the ISA makes it a good choice for highly specific embedded processors which need to be optimised for their task. There exists a group of standard ISA extensions for RISC-V which aim to reduce the size of the stored code by providing compressed instruction encodings, and in some cases reducing the number of required instructions [22]. The primary goals of this project lie around this group of extensions. For a few years, when designing a RISC-V architecture with a stringent memory requirement, there was only a single compressed instruction format ISA extension (RVC) to chose. Due to recent updates to RISC-V, the correct ISA choice for a system with this requirement has become slightly more complicated. The newly ratified Zc* extension comprises seven different options for code size reduction, most of which are combinable with eachother [11]. Moreover, there are many more proposals for different code size reduction extensions in development [5]. The computer architect is left to figure out which option is best suited to their system.

With this project, I intend to conduct experiments and analyses which will shed light on the individual effectiveness of each Zc* option for code compression. The results of the experiments and analyses should include:

- The average amount of compression to be expected from each extension
- How the compression rates achieved by the extensions change depending on the embedded application it implements.
- How well various functional groups of instructions are compressed by each extension

Along with this, I will provide a discussion of other important factors, unrelated to ISA choice, that can have a significant impact on compiled code size. I will also examine the hardware performance and price trade-offs one might have to make by implementing a compressed instruction ISA extension. In doing all of this, hopefully I will present information that could positively influence design choices for RISC-V embedded systems where memory and storage capacity are a limiting factor.

My final objective is to conduct an analysis of the instruction content of the compressed RISC-V executables. Using this, I am hoping to identify some further changes one could make to the RISC-V ISA that would reduce stored code size.

In Chapter 2, I will give a summary of important background knowledge and project context. In Chapter 3, I aim to give enough detail of the experiment's methodology, the challenges I encountered and it's limitations, so that other researchers may repeat this experiment and clarify any ambiguities left by my experimental shortcomings. In Chapter 4, I will present all the data gathered throughout the experiment and I will discuss it's implications. Finally, I will present the new RISC-V compressed instruction extensions Zcfa and Zcda and assess their potential compression impact.

Chapter 2

Literature Review

2.1 RISC-V Overview

RISC-V is an open source instruction set architecture first conceptualised in the University of California, Berkley in 2010. Krste Asanović, the author of the original paper proposing RISC-V, saw great merit in an open and flexible architecture that is suited to both academic and industrial purposes. Since 2015, RISC-V has been an international non-profit organisation. A large community of developers collaborating openly, has continuously driven innovations in the architecture. One of the main reasons for the success of RISC-V is it's extensibility. The architecture consists of a core instruction set with 32 and 64 bit formats as well as a multitude of extensions which allow it to be tailor-fit to many applications [19] [21]. This premise can be seen in the Introduction of the RISC-V instruction set manual, where the authors state that they have tried to avoid "over-architecting" the ISA for any one purpose, but to make it extensible enough that it can provide an efficient implementation of most applications [22].

The RISC-V user-level 32 bit ISA (RV32I) provides 32 registers x0-x31. Some registers are used to store important values regarding the execution state. These ABI registers have assembly aliases for improved readability. ZERO maps to x0 and always holds the value 0. ra maps to x1 and holds function return addresses. sp maps to x2 and points to the top of the stack. a0-a7 map to x10-17 and hold function arguments. s0-s11 map to x8,x9 and x18-27 and represent saved stack variables. The ISA provides four base instruction formats: R-type, I-type, S-type and U-type. R-type instructions consist of 2 source registers (rs1, rs2) and one destination register (rd). The I-type instruction format is similar to the R-type except, rs2 is substituted for a 12 bit immediate. An S-type instruction consists of 2 source registers and a 12 bit immediate. A U-type instruction consists of a 20-bit immediate and a destination register. The ISA also includes SB-type and UJ-type instructions which have the same format as S-type and U-type instructions respectively except, the encoded immediates are treated differently to form target address offsets. These instruction formats are shown in Figure 2.1. Integer computation instructions are typically encoded with the R-type format for register-register operations, the I-type format for register-immediate operations, and the U-type format for constant generation operations. Integer compu-

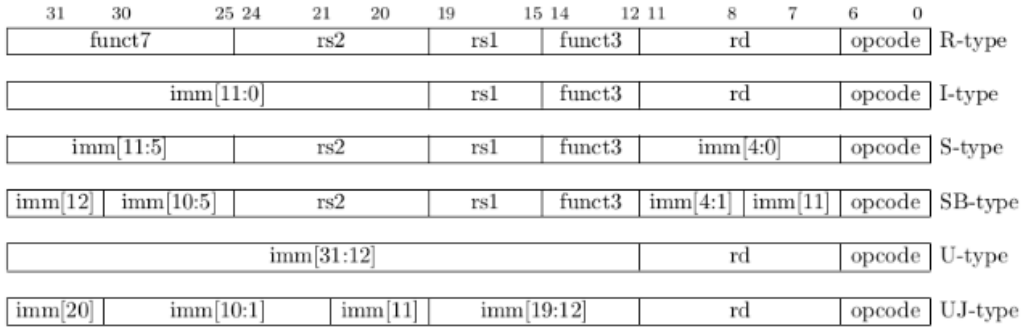

Figure 2.1: RV32I Base Instruction Formats [22]

tation instructions consist of variations of add, subtract, set less than, and, or, xor, and shift. Control transfer instructions provided include: a jump an link operation encoded with the UJ-type format, a jump and link register operation encoded with the I-type format and various conditional branch operations encoded with the SB-type format. The ISA provides memory load instructions using the I-type format and a memory store instructions using the S-type format. The RISC-V user-level 64 bit ISA (RV64I) adapts this foundation to provide an analogous instruction set for 64 bit wide registers and a 64 bit address space. RV64I still provides support for instructions which operate on 32 bit values, the names of which are suffixed with W. These ignore the upper 32 bits of the operands and output a 32 bit value which is sign extended to fit in a 64 bit register. The most commonly used extensions for RISC-V are M, A, F, and D. RVM implements multiplication and division instructions. RVA implements atomic loads and stores for safe concurrent and parallel memory access. RVF implements support for floating point operations. The RVF extension comes with an additional 32 FP registers f0-f31. It also defines FP loads/stores with analogous encodings and behaviours to their integer counterparts. All FP arithmetic operations have two register operands and use the R-type format. Arithmetic operations include ADD, SUB, MUL, DIV, MIN, MAX and SQRT. RVF provides CVT instructions which convert the content of a FP register into an integer, storing the result in an integer register or vice-versa. RVF MV instructions move a bit pattern from a FP register to an integer register or vice-versa, without alteration. The RVF sign injection instructions change the sign bit of rs1 to either match that of rs2 or be the negation rs2's. Floating point registers in RVF can take on special values including NaN, ∞ and subnormal values. Subnormal means smaller than is representable in the normal floating point encoding. Due to this, RVF also has a classify instruction which determines both the type and sign of it's floating point operand. RISC-V also implements compressed instruction format extensions RVC and Zc* which are the main focus of this paper and will be explained in depth in the next section [22].

There are currently 111 processor designs on the market which implement a RISC-V ISA, as well as many SoC designs [16]. Due to the customisability of RISC-V, these processors are extremely varied in design and cover a wide range of application domains. Nonetheless, it will be helpful for us to have an understanding of what a basic RISC-V processor architecture might look like. For this purpose we will examine the

features of the PicoRV32 processor design by Claire Wolf. PicoRV32 is a lightweight auxiliary CPU that supports compressed instruction encodings, designed to be implemented on a FPGA or ASIC. It is therefore well suited to memory sensitive embedded systems. This design implements many configurable hardware units including:

- A register file with either 16 or 32 registers and one or two read ports
- Three choices of memory interface (native, axi, wb) to allow easy integration into systems with different memory standards. All of these support variable length instruction fetching in their logic implementations.
- A decoder with extra logic and control signals to account for compressed instructions
- An integer ALU which can be configured to span one or two clock cycles
- A co-processor interface to facilitate communication with other cores
- Optional dedicated multiply, fast-multiply and divide cores

Most of these configurations give the user the choice to either optimise performance speed by adding more hardware or make the CPU as lightweight as possible by sacrificing performance [23].

2.2 RISC-V Compressed Instruction Encoding Extensions

In the RISC-V Instruction Set Manual they present the C standard extension, commonly referred to as RVC. This provides compressed 16 bit encodings for common instructions. Instructions with one or more of the following qualities were translated into compressed encodings:

- The immediate/address-offset is small
- One of the register fields reference ZERO, ra or sp
- the first operand and destination register are identical
- the registers used are one of the 8 most popular registers, referred to as the RVC registers (x8-x15)

The RVC instruction formats are shown in Figure 2.2. CR, CI and CS have 5 bit register fields and can reference any of the 32 registers. CIW, CL, CB and CS can only reference the RVC registers due to their 3 bit register fields [22].

Format	Name	Description	Base ISA Expansion
CL	C.LW / C.LD	Loads a 32/64 bit integer value from memory into one of the RVC registers, limited to a 5 bit offset	lw/ld rd' offset(rs1')
	C.FLW / C.FLD	Loads a 32/64 bit FP value from memory into one of the RVC registers, limited to a 5 bit offset	flw/fld rd' offset(rs1')
CS	C.SW / C.SD	Stores a 32/64 bit integer value from a RVC register into memory, limited to a 5 bit offset	sw/sd rs2' offset(rs1')
	C.FSW / C.FSD	Stores a 32/64 bit FP value from a RVC register into memory, limited to a 5 bit offset	fsw/fsd rs2' offset(rs1')
CI	C.LWSP / C.LDSP	Loads a 32/64 bit integer value from any register to sp + 6 bit offset	lw/ld rd offset(sp)
	C.FLWSP / C.FLDSP	Loads a 32/64 bit FP value from any register to sp + 6 bit offset	flw/fld rd offset(sp)
CSS	C.SWSP / C.SDSP	Stores a 32/64 bit integer value from any register to sp + 6 bit offset	sw/sd rs2 offset(sp)
	C.FSWSP / C.FSDSP	Stores a 32/64 bit FP value from any register to sp + 6 bit offset	fsw/fsd rs2 offset(sp)
CJ	C.J	Unconditional control transfer to pc + 11 bit offset	jal ZERO offset
	C.JAL	Unconditional control transfer to pc + 11 bit offset, then moves pc + 2 into ra	jal ra offset
CR	C.JR	Unconditional control transfer to address in rs1	jalr ZERO rs1 0
	C.JALR	Unconditional control transfer to address in rs1, then moves pc + 2 into ra	jalr ra rs1 0
CB	C.BEQZ / C.BNEZ	Control transfer to pc + 8 bit offset, if rs1' $= / \neq 0$, $rs1' \in RVC$	beq/bne rs1' ZERO offset
CI	C.LI	Loads a 6 bit sign extended immediate into rd	addi rd ZERO imm
	C.LUI	Loads a 6 bit non-zero sign extended unsig ned immediate into rd	lui rd uimm
	C.ADDI / C.ADDIW	Adds the 6 bit sign extended immediate to the 64/32 bit value in rd, stores result in rd	addi/addiw rd rd imm
	C.ADDI16SP	Adds 6 bit immediate scaled by 16 to sp, stores result in sp	addi sp sp (imm«4)

CIW	C.ADDI4SPN	Add zero-extended u-immediate scaled by 4 to sp, stores result in $rs1' \in RVC$	addi rs1' sp (uimm«2)
CI	C.SLLI	Left logical shift rd, stores result in rd	slli rd rd shamt
CB	C.SRLI / C.SRAI	Right logical/arithmetic shift rd', stores result in rd', $rd' \in RVC$	srli/srai rd' rd' shamt
	C.ANDI	Bit-wise and of 6 bit sign extended immediate and rd', stores result in rd'	andi rd' rd' imm
CR	C.MV	Copies value in rs2 into rd	add rd ZERO rs2
	C.ADD	Adds values in rd and rs2, stores the result in rd	add rd rd rs2
CS	C.AND	Bit-wise and of values in rd' and rs2', stores result in rd', $rd', rs2' \in RVC$	and rd' rd' rs2
	C.OR	Bit-wise or of values in rd' and rs2', stores result in rd', $rd', rs2' \in RVC$	or rd' rd' rs2
	C.XOR	Bit-wise xor of values in rd' and rs2', stores result in rd', $rd', rs2' \in RVC$	xor rd' rd' rs2
	C.SUB	subtracts the value in rs2' from rd', stores the result in rd', $rd', rs2' \in RVC$	sub rd' rd' rs2
	C.ADDW	Adds the 32 bit value in rd' to the one in rs2', stores the result in rd', $rd', rs2' \in RVC$	addw rd' rd' rs2
	C.SUBW	subtracts the 32 bit value in rs2' from the one in rd', stores the result in rd', $rd', rs2' \in RVC$	subw rd' rd' rs2

Format	Meaning	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	Register	funct4		rd/rs1		rs2		op
CI	Immediate	funct3	imm	rd/rs1		imm		op
CSS	Stack-relative Store	funct3	imm			rs2		op
CIW	Wide Immediate	funct3	imm				rd′	op
CL	Load	funct3	imm		rs1′	imm	rd′	op
CS	Store	funct3	imm		rs1′	imm	rs2′	op
CB	Branch	funct3	offset		rs1′	offset		op
CJ	Jump	funct3	jump target					op

Figure 2.2: RVC compressed instruction formats [11]

Since the first publication, there has been a series of additions and refinements to the compressed instruction support implemented by RISC-V. These are broadly categorised under the Zc* extensions which include Zca, Zcb, Zcd, Zce, Zcf, Zcmp and Zcmt. Zca encompasses all the instructions defined in the original RVC extension other than floating point loads and stores. Zcb implements new compressed integer computation instructions including sign/zero extension, NOT and MUL. It also implements compressed byte and half bytes loads and stores. Zcf and Zcd encompass the floating point loads and stores which are already defined for RVC. They contain single and double precision operations respectively. Zcmp provides instructions which greatly reduce function prologue and epilogue code sections. It defines cm.push and cm.pop. cm.push adjusts the stack pointer to create the stack frame before storing a range of registers to the stack. cm.pop loads a range of registers from the stack before adjusting the stack pointer to destroy the stack frame. The range of registers is specified as a reg_list or xreg_list variable (Figure 2.3). cm.popret behaves the same as pop and additionally jumps back to ra. cm.popretz does all of this and sets the argument register a0 to 0. cm.mvsa01 moves a0 and a1 into two selected stack registers s0-s7 and cm.mva01s does the reverse. Zcmt defines table jump instructions cm.jt and cm.jalt.

```
switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
```

Figure 2.3: A switch enumerating the possible reg_list and xreg_list values [11]

It also implements the additional infrastructure required for these instructions i.e. a jump vector table (JVT) and a control status register which stores the base address of the JVT. These table jump instructions index into the JVT to retrieve the stored jump target address. Zce is the recommended set of compressed instruction formats for embedded micro-controllers, this includes Zca, Zcb, Zcmp, Zcmt for 64 bit architectures.

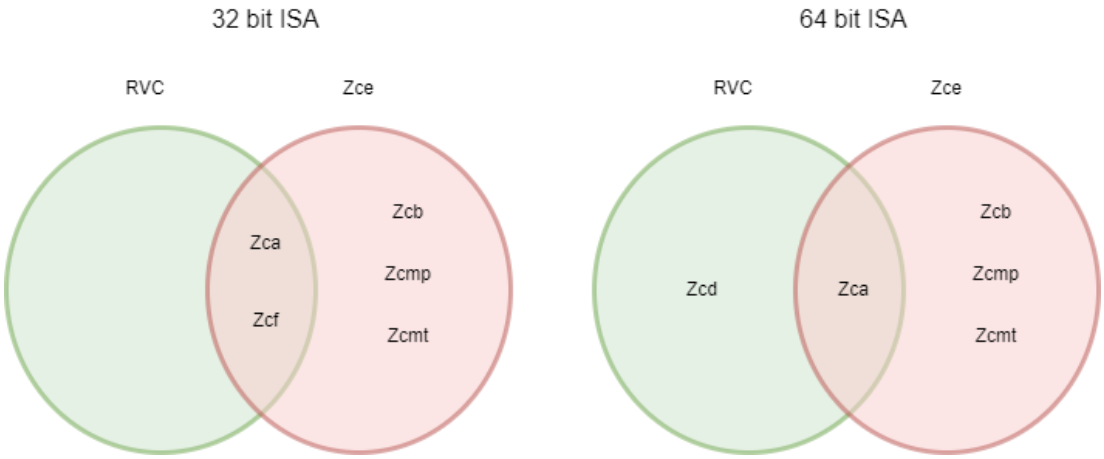

Figure 2.4: A Venn diagram showing the make up of RVC and Zce in the 32 and 64 bit variants of RISC-V

In 32 bit architectures with the RVF extension enabled Zcf is also included. Zcf is not supported in RV64 [11].

2.3 Similar Work

Many aspects of the experiment I wish to conduct in this project were inspired by a paper by Peije Li entitled Reduce Static Code Size and Improve RISC-V Compression [14]. In this section I will give a summary of his experiment's methodology and results. I will also discuss how the paper might have influenced more recent changes to the RISC-V ISA.

Li attempts to evaluate the effectiveness of the RISC-V C extension at compressing embedded IoT program applications. I would like to note that the benchmark programs which Li compiles are not exclusively IoT applications, programs such as Dhrystone, Whetstone, Basicmath, Stringsearch and Bitcount are equally applicable to offline embedded systems. Li then proposes some optimisations to the RVC extension based on observations from his experimental analyses. Finally he evaluates the performance implications of his proposed optimisations.

Li compiles the selected benchmark programs to riscv32ima targets with and without the RVC extension enabled. He compiles each program with the -O3, -Os, -msave-restore, and -flto flags enabled. Li does not disclose whether the the executables are statically or dynamically linked. He does not use the -static linker flag but, depending on the default configuration of his compilation environment, the files may still be statically linked. We will see in Section 3.1, that this can be a major contributing factor to compression ratio. Li presents his findings from this experiment which include the average compression ratio achieved from RVC, the average percentage of compressed instructions and the percentage of the benchmark programs constituted by each instruction.

From his collected data, Li makes the following observations:

- A few instruction formats make up most of the code body
- A notable portion of instructions with small immediates remain uncompressed by RVC because the register operand does not reference an RVC register
- Most of the uncompressed immediates are multiples of four
- Data transfer instructions account for one third of the uncompressed instruction body
- A negligible portion of the compressed instructions are floating point operations

Using this final observation Li claims that the opcodes for the compressed floating-point could be recycled for more effective compressed instruction formats. Li proceeds to propose seven new instruction encodings summarised in the table below.

Observations	Instruction Name	Description	Base ISA Expansion
The majority of uncompressed ADDI instructions have $rd \in RVC$. A quarter have rs1=x0. Of the instructions with both properties, 80% of their immediates are representable in eight bits and 40% are odd	C.LI*	Loads a zero-extended eight bit immediate into one of the RVC registers	addi rd' x0 imm[7:0]
In the $\approx$ 50% of addi instructions where both rs1 and rd reference an RVC register: 96% are multiples of two, 50% occupy more twelve or more bits and 50% reference a5	C.ADDI*	Adds a 12 bit, sign-extended immediate scaled by 2 to the value in register a5 before storing it in a5	addi a5 a5 nzimm[10:0]
Most uncompressed jump instructions have an immediate offset that requires at least fifteen bits. Almost all control transfer instructions target a function label.	C.J*	Unconditionally jumps to an entry in the symbol table, indexed by a ten bit immediate. The link register is specified in a three bit rd field	jal rd* symbols[offset[9:0]]

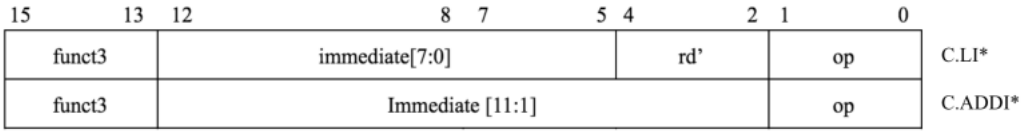

Figure 2.5: Li's proposed C.LI* and C.ADDI* instruction encodings [14]

Within uncompressed branches, rs1 usually references an RVC register and rs2 usually references x0 or a5. A notable portion of branch offsets are representable in a four or five bit, unsigned immediates	C.BEQBNE	Chooses to execute a BEQ or BNE operation with a new funct bit. $rs2* \in \{xo, a5\}$	beq/bne rs1 rs2* offset[4:1]
	C.BRANCH	Has a sb_funct field which selects the BEQ, BNE, BLT or BGE operation. $rs2* \in \{xo, a5\}$ $rs1 \in RVC$	BranchOP rs1' rs2* offset[5:1]
The most common uncompressed load/store transfer size is word followed by byte. The uncompressed transfers mainly have $rs1 \in RVC$. 50% of the immediate offsets of uncompressed byte transfers are zero	C.LDSTBYTE0	New funct bit chooses the load or store operation. Either store an unsigned byte from rs2/rd to the address specified in rs1, or load a byte from the address in rs1 into rs2/rd (offset = 0)	lbu rd 0(rs1) or sb rs2 0(rs1)
	C.LDSTWORD0	Either store an unsigned word from rs2/rd to the address specified in rs1, or load a word from the address in rs1 into rs2/rd (offset = 0)	lwu rd 0(rs1) or sw rs2 0(rs1)

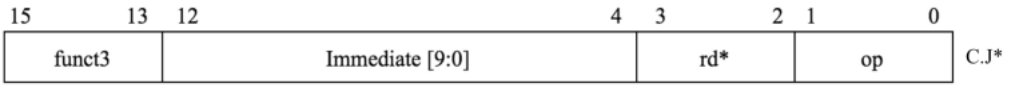

Figure 2.6: Li's proposed C.J* instruction encoding [14]

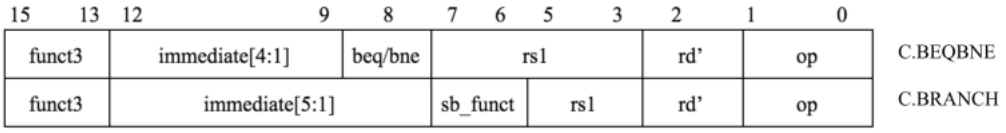

Figure 2.7: Li's proposed C.BEQBNE and C.BRANCH instruction encodings. *Typo: rd' should be rs2** [14]

To assess the potential impact of these proposed encodings, Li calculates the compression ratios they could theoretically achieve. He does this by going through the code and compressing instructions that meet the translation criteria into his proposed encodings, as well as decompressing the floating point instructions from which he borrows the opcodes. He finds that that the code size drops by a further 10% on average and that the overall file size is reduced by an average of 4.97%.

The publication of this paper in 2019 predates the release of the Zc* extensions which are the first update to RISC-V's code compression support since RVC. We can therefore look at how Li's paper might have influenced some of the additions in Zc*. The proposed C.J* is very similar to the instructions cm.jt and cm.jalt in the Zcmt extension, the functionality of which is explained in Section 2.2. The Zcmt instructions also tackle the problem that Li observes where most uncompressed jump instructions have large offsets. While the instructions are slightly different in the accessible link registers and the type of target-address table they index into, it is not a reach of the imagination to think that Zcmt could have taken inspiration from Li's findings. Zcmt also reuses a compressed floating point opcode making it incompatible with the Zcd extension. This decision could have been influenced by Li's finding which indicates that compressed floating point instructions are not as effective as the others, making them the natural choice for a necessary opcode sacrifice. Zcb implements the compressed data transfer instructions c.lbu, and c.sb. These instructions are similar to C.LDSTBYTE0 in having very small offset immediates and a byte trasfer-size. These could also be motivated by a similar observations regarding a large proportion of the uncompressed code consisting byte loads/stores with small or zero offsets [14].

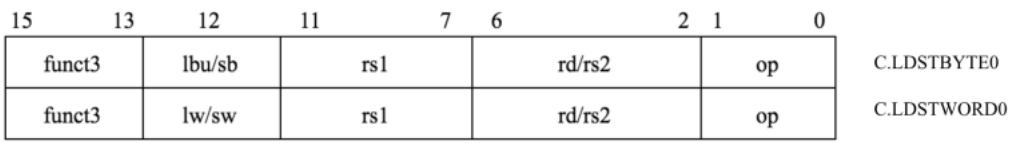

Figure 2.8: Li's proposed C.LDSTBYTE0 and C.LDSTWORD0 instruction encodings [14]

Chapter 3

Methodologies

3.1 Compiling Embedded Benchmarks to Compressed RISC-V Targets

As stated in the introduction, the primary goal of this project is to provide insights into the effectiveness of the different RVC/Zc* extensions at reducing compiled code size. The secondary goal is to conduct an in depth analysis of code compiled to RVC/Zc* targets in order to identify further code compression optimisations that could be made to the RISC-V ISA. Both of these goals require a sufficiently large and representative sample of embedded programs, compiled to RISC-V targets with the RVC/Zc* ISA extensions enabled. In this section, I will give an overview of the chosen benchmark programs. I will explain the compilation process and the challenges I faced along the way. Finally, I will identify the flaws in my acquired data for experimental transparency.

MiBench is an open source set of embedded benchmark programs originally proposed in 2001 [10]. The authors notice that there are many existing embedded benchmark programs, but these span a broad range of applications and share few unifying characteristics. In an attempt to provide a general purpose embedded benchmark set, MiBench contains 35 embedded applications divided into six categories which are deemed to be the most common embedded application types. These categories are Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications. Even though embedded device applications are becoming more diverse as the market grows, these categories still cover the most common embedded applications today. I have chosen all but one of the benchmark programs for this experiment from MiBench (at least one from each category). I originally set out to compile all of MiBench, but some of the programs had large and intricate build files which automatically configure to target your host architecture. Unfortunately, reconfiguring the most complex of these to target RISC-V was beyond my capabilities. There were also a few benchmarks that included libraries, which the compiling environment I set up did not support. While many of the chosen MiBench programs do implement floating point calculations, I wanted to ensure that my experiment had at least one program designed to be FP operation intensive, so I can assess Peije Li's

claim regarding a low percentage of compressed FP instructions in embedded applications (discussed in Section 2.3) with confidence [14]. I chose Whetstone for this purpose.

The benchmark programs used in this experiment are:

- Automotive and Industrial Control
  - Basicmath
  - Bitcount
  - Qsort
  - Susan
- Consumer Devices
  - Cjpeg
  - Djpeg
  - Lame
- Office Automation
  - Stringsearch
- Networking
  - Dijkstra
- Security
  - Blowfish
  - Rijndael
  - SHA
- Telecommunications
  - CRC32
  - FFT
- Floating Point Intensive
  - Whetstone

Basicmath implements a series of mathematical operations that typically don't have dedicated hardware support in embedded processors such as square root and solving cubic equations. Bitcount tests bit manipulation capabilities, by counting the bits in an array of integers, using a few different methods to do so. Qsort sorts a large array of strings into ascending order using the Quick Sort algorithm. All three of these implement the type of operations which occupy most of the workload in control systems. Susan is an image recognition program developed to identify corners and edges in MRI scans. It is representative of computer vision applications. Dijkstra computes the shortest path between every node of a graph which is stored as an adjacency matrix, this is typical of workloads for routers and other networking devices. Blowfish and Rijndael apply a block cipher to text inputs using variable length encryption keys. Rijndael was the selected Advanced Encryption Standard (AES) algorithm, when MiBench was published. SHA is a hashing algorithm commonly used for secure key exchange. Encryption and hashing are some of the most common tasks for data security devices. Cjpeg and Djpeg implement lossy image compression and decompression algorithms for jpeg files. Lame is a MP3 encoder with configurable bit-rates, it accepts WAV inputs. These programs are representative of those in multimedia consumer devices. Stringsearch searches for given words in phrases. This kind of string manipulation program is often found in printers and other office devices. FFT implements the Fast Fourier Transform algorithm and it's inverse function. This algorithm is crucial for digital signal processing in telecomm devices. CRC32 implements a 32 bit cyclic redundancy check. This type of check is usually carried out on transmission data to identify errors [10]. Whetstone is a benchmark which is designed to test floating point computation efficiency [20].

```
qemu-system-riscv64 -machine virt
-m 4G -smp 8 -bios default
-kernel /usr/lib/u-boot/qemu-riscv64_smode/uboot.elf
-device virtio-net-device,netdev=eth0 -netdev user,id=eth0
-device virtio-rng-pci
-drive file=ubuntu-23.10-preinstalled-server-riscv64.img,
format=raw,if=virtio
-nographic
```

Figure 3.1: The QEMU command used to generate the RISC-V virtual machine

Ideally, I would draw my analyses from a significantly larger dataset however, I believe my selection of benchmarks programs is varied enough such that my results are generally applicable to embedded systems, and as large as the time constraints of this project allowed me to make it. I encourage other researchers to replicate this experiment on a larger dataset.

The next task was to attain a method of compiling the benchmarks to RISC-V targets with the RVC and Zc* extensions enabled. Due to how recently the Zc* extensions were released, this proved to be far from trivial. As I had already been working with the RISC-V GNU tool-chain (Appendix A), this seemed like the natural choice as a compilation tool. With it's implemented version of gcc (riscv64-unknown-elf-gcc), I was able to compile programs to RVC extended ISA targets but found that this compiler doesn't currently support the Zc* extensions. In GCC's documentation I found that the Zc* extensions are listed as possible -march compiler flag options. -march specifies the target ISA for RISC-V compilations [9]. This led me to believe that the RISC-V GNU tool-chain repository implements an outdated version of GCC. So, I attempted to compile the programs from a RISC-V virtual machine running a disk image with the latest versions of Ubuntu and GCC installed. To do this, I familiarised myself with QEMU. QEMU is a system emulation and virtualization tool. QEMU uses one of it's supported virtual machine models to run a guest OS managed by a choice of hypervisors. I configured QEMU to run Ubuntu 23.10 on a virtual RISC-V machine (Figure 3.1). I soon discovered that my virtual machine efforts were in vain, as the GCC documentation claiming Zc* support was for the unreleased GCC-14. GCC-13 does not support the Zc* ISA extensions. In looking into other compiler options, I discovered that LLVM's Clang was a significantly better option for this task. Clang is a cross-compiler by design, meaning it can target many supported target ISAs regardless of the host architecture. Clang also implements complete support for Zc* targets in it's latest release (18). The only caveat was that my OS's software package handler only supports Clang distributions up to Clang-16. Thankfully, LLVM is open-source and I was able to do a manual download and install of Clang-18. This involves cloning the repository, checking out to the 18 release branch, configuring the build files for the project with a cmake command, and installing the tools. My first install was successful. I was able to access the full utility and ISA support implemented by Clang-18. Despite this, I was still unable to compile any of the benchmark programs due to linking errors, which arose from including C standard library headers. This was due to LLVM not contain-

```
cmake -S llvm -B build -G Ninja
-DLLVM_ENABLE_PROJECTS="clang;lld"
-DCMAKE_INSTALL_PREFIX="~/llvm_install"
-DCMAKE_BUILD_TYPE=Release
-DLLVM_TARGETS_TO_BUILD="RISCV"
-DCMAKE_C_COMPILER="clang"
-DCMAKE_CXX_COMPILER="clang++"
-DDEFAULT_SYSROOT="~/rv64_gnu_install/riscv64-unknown-elf"
-DGCC_INSTALL_PREFIX="~/rv64_gnu_install"
```

Figure 3.2: The cmake command used to configure the LLVM install

ing implementations of the C standard libraries for cross-compilation targets. It instead requires that you use compiler flags to link separately installed C standard library files. Eventually, I found a forum post where they resolved this issue by setting the LLVM build configuration flags *-DDEFAULT_SYSROOT* and *-DGCC_INSTALL_PREFIX*, as a filepath to an install of the RISC-V GNU Tool-chain. The exact LLVM install configuration I used is shown in Figure 3.2. This install of Clang-18 was able to compile the benchmark programs, along with their linked libraries to RISC-V Zc* ISA targets.

The compilation ISA targets for this experiment are:

- rv64ifd (control)
- rv64ifdc
- rv64ifd_zca
- rv64ifd_zcd
- rv64ifd_zcb
- rv64ifd_zcmp
- rv64ifd_zcmt
- rv64ifd_zce

These comprise a base ISA control and a result for each available compressed instruction extension. I used the -Os flag for every compilation. This enables compiler code size optimisations. This way, my found compression ratios are representative of the size reduction achievable after the compiler has done everything it can. With similar justification I used the -msave-restore flag. This flag replaces code sequences which store registers before function calls, and load registers after function calls, with a jal instructions to subroutines which perform the same load/store sequences. This removes repeated function prologues and epilogues. This optimisation is similar to some of the functionality of Zcmp and may reduce it's effectiveness. I included it anyways, because it's redundant to implement ISA and hardware support for something that the compiler can already do. I also compiled each target with and without the -static linker flag. This flag ensures that all library code needed for the program is statically linked in the executable, as opposed to a dynamically linked where the library object files are found externally and linked during execution. The choice between these two linking options should have a large impact on the program size. Figure 3.3 illustrates the experimental set of compilations in the case of a benchmark program that consists of a single C file. Many of the programs had more complicated project structures and Make files but the compilation and linking configurations remained the same.

The resulting data was imperfect in a few ways. By listing the issues which caused

```
clang -Os (-static)? -msave-restore
-march=rv64ifd{c,_zca,_zcd,_zcb,_zcmp,_zcmt,_zce}?
-mabi=lp64d example.c -o example LIBFLAGS
```

Figure 3.3: An example of the compilation commands used for an individual benchmark encapsulated by a regex expression

these imperfections, I hope to give others a means of repeating the experiment while avoiding them. You may have noticed that the compilations target RISC-V's 64 bit variant. This isn't ideal because 32 bit embedded processors are more common than 64 bit. Particularly in the context of this project where memory is a critical resource, a 32 bit address space is more probable. I originally tried to target rv32ifd but, the compilation environment I set up failed to link the object files, reporting that they were in the wrong format. Despite this, the effect on the results should be very small. All compressed instruction extensions implement equivalent versions of compressed encodings for 32 and 64 bit ISAs, except for Zcf which is incompatible with 64 bit [11]. For this reason the Zcf extension has been omitted from the experiment. I believe I could have avoided this outcome by installing the 32 bit version of the RISC-V GNU Tool-chain however, I have not had the chance to verify this. The other issue with the compiled programs presented itself when I originally calculated compression ratios as shown in Equation 3.5.

$$\forall \textit{compressed ISA targets } t \in \tag{3.1}$$
$$\{rv64ifdc,\ rv64ifd\_zca,\ rv64ifd\_zcd,\ rv64ifd\_zcb, \tag{3.2}$$
$$rv64ifd\_zcmp,\ rv64ifd\_zcmt,\ rv64ifd\_zce\} : \tag{3.3}$$
$$t \sim \textit{compression ratio} = \tag{3.4}$$
$$\frac{t \sim \textit{compressed executable file size}}{\textit{uncompressed executable file size}} \tag{3.5}$$

These were much larger than expected, in the range (0.87, 1.0001). For comparison, the compression ratios Peije Li found for RVC targets in his experiment were tightly distributed around 0.7 [14]. On top of this, the compilations that were meant to be statically linked were the exact same size as those that were meant to be dynamically linked. I investigated the object-dump assembly programs and discovered the reason for these issues. All the executables were statically linked with their included libraries. These libraries were stored as object files in the RISC-V GNU Tool-chain, pre-compiled to a target which had the RVC extension enabled. They also made up a significant majority of the compiled code. Therefore, regardless of the specified ISA target of my compilations, the majority of the file consisted of RVC compiled library code. This reduced the size of the "uncompressed" executables, increasing the compression ratios. As for the files with $\textit{compression ratio} > 1$, my best hypothesis is that each ISA extension adds a small overhead to the executable file size. Without enough time to find a solution that obtains the originally planned compilations, I decided to

strip the object-dump files of all library code and calculate the compression ratios as shown in Equation 3.10

$$\forall compressed\ ISA\ targets\ t \in \tag{3.6}$$
$$\{rv64ifdc,\ rv64ifd\_zca,\ rv64ifd\_zcd,\ rv64ifd\_zcb, \tag{3.7}$$
$$rv64ifd\_zcmp,\ rv64ifd\_zcmt,\ rv64ifd\_zce\} : \tag{3.8}$$
$$t \sim compression\ ratio = \tag{3.9}$$
$$\frac{\#(instruction\ bytes\ in\ t \sim compressed\ file)}{\#(instruction\ bytes\ in\ uncompressed\ file)} \tag{3.10}$$

This should be effectively the same as calculating the compression ratio for dynamically linked versions of the compilations and excluding the linking table and any other program data from the calculation.

3.2 Compression Analysis Metrics

Having acquired something resembling the necessary data for my planned experiments. I set out to extract information regarding the effectiveness of the Zc* extensions and conduct analyses that may lead to the identification of new code size optimisations for RISC-V. In this section, I will explain the motivation for my chosen statistics and the information we can extract from them.

As mentioned in the previous section, for each benchmark, I calculated the compression ratio achieved by each compressed ISA target using the formula defined in Equation 3.10. This crucial metric tells us the percentage by which the code size has reduced in one of the compressed ISA targets, compared to the control. The average compression ratio achieved by an ISA target over all the benchmarks should give us an idea of how effective it is for compressing embedded code. At this point, it is important to clarify that Zcd, Zcb, Zcmp, and Zcmt are dependent on Zca. This means that although Zca is not specified in the compilation target along with these extensions, it is automatically included. It would be useful to see how much compression is contributed by each extension explicitly and implicitly included in the ISA target. For this, I calculated the a new metric: compression contribution. This is defined as the fraction by which the uncompressed code size is reduced, by the instructions defined for one extension (Equation 3.14).

$$\forall\ extensions\ e \tag{3.11}$$
$$s.t.\ e\ is\ included\ in\ the\ ISA\ target : \tag{3.12}$$
$$e \sim compression\ contribution = \tag{3.13}$$
$$\frac{16\ \times\ \#(e \sim instructions)}{\#(instruction\ bytes\ in\ uncompressed file)} \tag{3.14}$$

By appending a compression contribution breakdown to each compression ratio, we gain understanding of how each extension's instructions have contributed to that number. RVC and Zce were not counted as contributors because they are defined as groupings of other extensions however, the compression ratios which RVC and Zce achieved have been broken down in the same way.

The next insight I want to provide is how one can expect the effectiveness of the Zc* extensions to change depending on the content of the embedded program. I have measured this in two ways. The simpler of these is the average compression ratio achieved by each ISA target, over each MiBench embedded application category. This should should tell us about how the compression achieved by a target changes depending on the application it's used for. This isn't the best metric because of the the small sample size for each application type and the dependence on MiBench to accurately summarise application workloads. For this reason, I also decided to calculate the effectiveness of each ISA target at compressing the RISC-V instruction categories listed below.

- Integer Computation
- Conditional Control-Transfer
- Unconditional Control-Transfer
- Integer Load/Store
- Floating Point Load/Store

Each of these categories represents a compressible portion of any RISC-V program. Knowing how well an extension will perform at compressing these instruction categories, could inform the ISA choice for an embedded system where, the prevalence of these instruction categories is either known or can be predicted. To assess the the compression that can be achieved by an ISA target over each of these categories, I defined category compression ratio (Equation 3.21).

$$\forall instruction\ categories\ c \in \tag{3.15}$$
$$\{IntComp,\ CondCT,\ UncondCT, IntLdSt,\ FPLdSt\} : \tag{3.16}$$
$$\forall compressed\ ISA\ targets\ t \in \tag{3.17}$$
$$\{rv64ifdc,\ rv64ifd\_zca,\ rv64ifd\_zcd,\ rv64ifd\_zcb, \tag{3.18}$$
$$rv64ifd\_zcmp,\ rv64ifd\_zcmt,\ rv64ifd\_zce\} : \tag{3.19}$$
$$c \sim t \sim compression\ ratio = \tag{3.20}$$
$$\frac{\#(c \sim instruction\ bytes\ in\ t \sim compressed\ file)}{\#(c \sim instruction\ bytes\ in\ uncompressed\ file)} \tag{3.21}$$

Chapter 4

Results and Discussions

4.1 Compression Comparison of RISC-V C/Zc* Extended ISA Targets

In this section, I intend to offer some insight on how to answer the question 'Which RISC-V code compression extension should I use for my embedded system?'. Below, I will present the data collected and processed as described in Sections 3.1 & 3.2, and discuss how this data could inform such a decision. All the data compares the performance of the RISC-V compilation targets rv64ifdc, rv64ifd_zca, rv64ifd_zcd, rv64ifd_zcb, rv64ifd_zcmp, rv64ifd_zmt and rv64ifd_zce with respect to different compression metrics.

Figure 4.1 shows us every compression ratio achieved across the experiment. Here we see values in the range (0.64 , 0.84). Telling us we can expect to see a 16-36% decrease in code size by applying any one of these extensions. We can compare the compression ratios in Figure 4.1 to those found by Peije Li. None of my rv64ifdc compression ratios match those found by Li using the same compilation target for the same benchmark programs. Furthermore, on average Li's compression ratios appear to be lower than not only my rv64ifdc results but, all extension targets bar rv64ifd_zce. rv64ifd_zce only achieves slightly lower compression ratios. I can't deduce the exact reason for this without more information about Li's methodology but, there are two potential causal factors. Li's programs could be statically linked and contain a large volume of library code that wasn't considered in my calculations. The difference in our compiler flags also could have impacted the results. Li used the additional flags -O3 and -flto flags, which can impact the assembly code content[14].

Directing our attention to Figure 4.2 we see some of our most important results. The figure shows the mean compression ratio achieved by each extended ISA target over all the benchmarks. The large green segments of the pie charts display the compression ratio as a percentage, and each of the other segments represent a contribution to the compression from one of the extensions. I will remind you for clarity that the targets rv64ifd_zcd, rv64ifd_zcb, rv64ifd_zcmp, and rv64ifd_zmt are dependent on the Zca extension, and the targets rv64ifdc and rv64ifd_zce are both groupings of the other

Figure 4.1: The compression ratios achieved by each extension ISA target on each benchmark

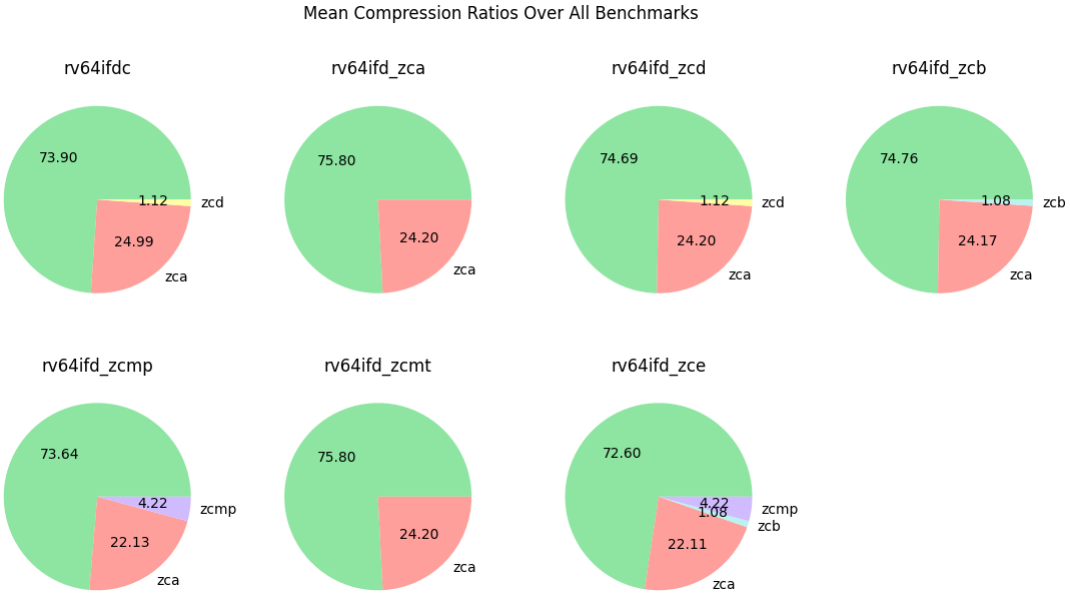

Figure 4.2: The mean compression ratio on achieved by each extension ISA target over all benchmarks, broken down into compression contributions from each extension

Zc* extensions. Thus, there is multiple extension contributors to every compilation target bar rv64ifd_zca. We can see, the extension targets that achieved the best average compression ratios in order are:

1. rv64ifd_zce
2. rv64ifd_zcmp
3. rv64ifdc
4. rv64ifd_zcd
5. rv64ifd_zcb
6. rv64ifd_zca & rv64ifd_zcmt

It is expected but still an important to verification that rv64ifd_zce performs the best as, Zce is the compressed ISA extension that's recommended by RISC-V for embedded systems. We can see that the difference in performance between the extensions is small with only a 3.2% difference between rv64ifd_zce and rv64ifd_zca. This is explained when we look at the extension contributions. Zca makes up the large majority of the compression contribution in every compilation target, reducing the code size by an average of 22.1% or 24.1% depending on the extensions it's paired with. This makes sense because it defines the most compressed instruction encodings. Zcmp achieves an average compression contribution of 4.2%. However, there is evidently some overlap in Zcmp's and Zca's contribution because Zca's average contribution reduces by 2% when paired with Zcmp. This reduces the pragmatic average compression contribution of Zcmp to 2.2%. Zcd and Zcb both contribute an average of about 1% compression.

Using a direct comparison of results, Peije Li's seven proposed encodings achieve a much larger performance improvement on rv64ifdc result than rv64ifd_zce. Li's instructions theoretically reduce RVC compiled code size by an average of 10% while my results show rv64ifd_zce only achieves a 1.48% code size reduction on rv64ifdc. This however, should not be taken as evidence the Li's proposal is better than Zce. As we have already seen, there is obviously some crucial differences in Li's experimental method and mine, leading to inconsistent results. Also, there is no guarantee that Li's proposal would perform the same in practical implementation as it does in his theoretical calculations.

Unexpectedly, we find that rv64ifd_zcmt performs identically to rv64ifd_zca. This can be explained by the fact that across all benchmarks, not a single Zcmt instruction was used, so all of rv64ifd_zcmt's compression was contributed by Zca. On first look, that seems wrong, as if there was some problem with the compilations however, I can assure you this is a genuine result. We can find an explanation if we examine the optimisations which Zcmt implements and the way I compiled the benchmarks. Zcmt provides table jump instructions which allow unconditional control transfer to code anywhere in the address space with a single 16 bit instruction. In the 32 bit ISA, individual jump (and link ra) instructions are translated to cm.jt and cm.jalt. In the 64 bit ISA, only sequences of auipc+j/jal ra are translated into cm.jt/cm.jalt i.e. control transfers outside of the PC +/- 1MB range. Recall from Section 3.1 that despite my best efforts, all my compilations were statically linked. This means that all the needed program code and library functions are stored in a contiguous section of the address space. As all the attained executables are less than 1MB in size, there are no control transfers which demand a Zcmt translation. This is not the case for dynamically linked

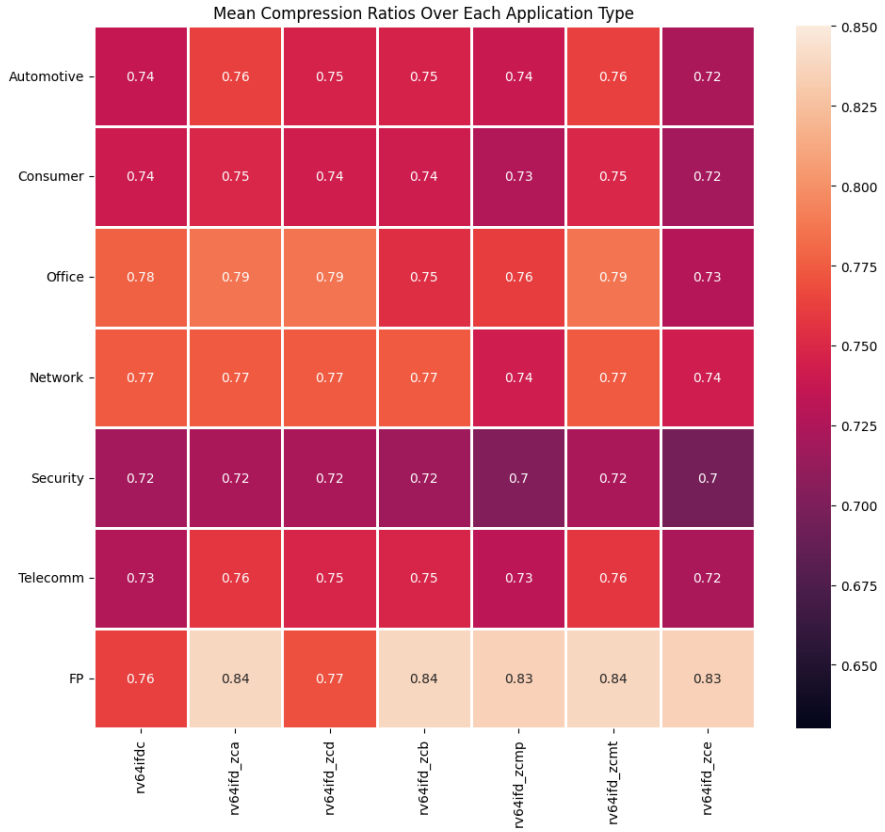

Figure 4.3: The mean compression ratios achieved by each extension ISA target type over each MiBench application type

programs and we might have found a very different Zcmt result had I managed to attain such compilations. The reasoning for the restricted translation criteria in the 64 bit ISA is not clear in the documentation. It seems to me that Zcmt translation of 64 bit j/jal instructions with offsets insinde the +/-1MB range would still reduce code size by changing 32 bit instructions to 16 bit instructions, even though it is not necessary to reach the desired address in a single instruction. It could be due to the expected number of required 64 bit JVT entries outweighing the expected code size reduction but requires further investigation [11].

Now that we have an understanding of the general performance of the extensions, we can examine their effectiveness for individual applications. Figure 4.3 shows the mean compression ratios achieved by each ISA target over each MiBench application type. Figure 4.4 displays the same information using the green bars. Additionally, it shows the individual extension contributions with the other stacked bars. Note that the ISA target names rv64ifdc, rv64ifd_zca etc have been written as their extension names for space efficiency in Figure 4.4. We can see that for most of the applications, the extension rankings remain similar to the results we've already looked at. The only application type which behaves notably different from the general results is floating point intensive. Here we see rv64ifdc performs the best, closely followed by rv64ifd_zcd with the rest trailing behind by a long way. If we look at the contribution make up of the rv64ifdc and rv64ifd_zcd compilations for floating point applications,

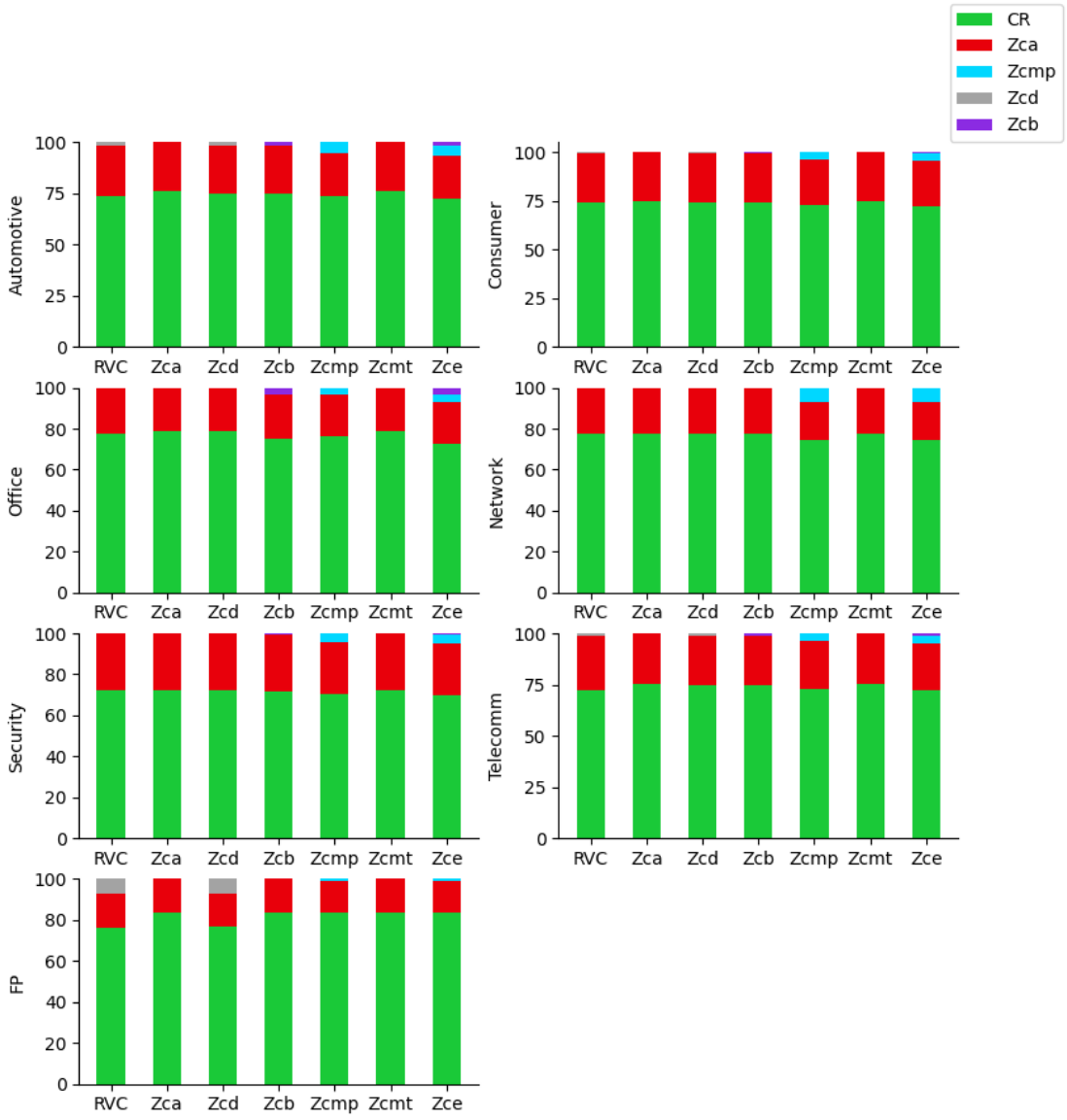

Figure 4.4: The mean compression ratios achieved by each extension ISA target over each MiBench application type and their contributions

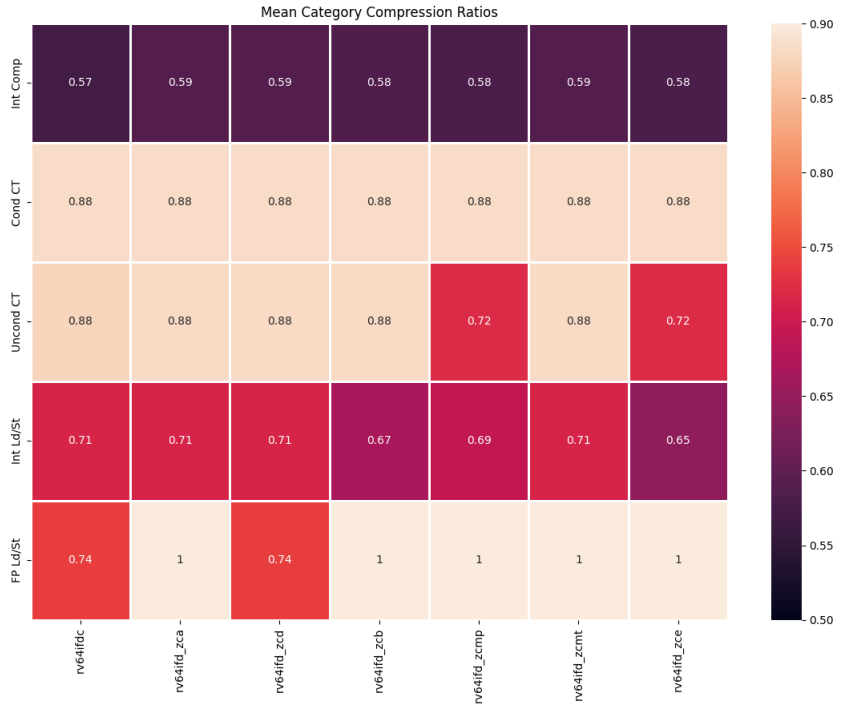

Figure 4.5: The mean of each category compression ratio achieved by each extension ISA target over all benchmarks

we see that Zcd's contribution is significantly larger than it is for any other application type, growing from 1% to 7%; Zca's contribution is much smaller, dropping from 24% to 16%, and the other contributions shrink to irrelevance. This drastically different compression behaviour, suggests that embedded systems with floating point intensive applications need different compression treatment. This also brings a new perspective to the result Li found which indicates floating point instructions in RVC make up a negligible amount of the compressed instruction body [14]. While this is technically true, most of the embedded benchmarks he studies are lightweight in terms of floating point operations. Furthermore, the reduction in contribution from Zca might indicate that floating point intensive applications could greatly benefit from compressed FP computation instructions. This prediction is based on the assumption that FP computation instructions have "replaced" integer computation instructions which are usually compressed by Zca.

Finally we will examine Figure 4.5. This shows the mean category compression ratios (defined in Section 3.2) achieved by each extension ISA target. From this, I originally planned to gain some insight into which extension target to choose, given some knowledge of the instruction make up of the application i.e. column-wise comparison. In the end, I found that the more interesting part of this data is the variation in the rows as well as an absent row which we can infer. Starting with the columns, the key observations I have made are:

- rv64ifdc performs slightly better than the rest at compressing integer computation instructions

- rv64ifd_zcmp and rv64ifd_zce outperform the other extensions at compressing unconditional control transfer instructions by a significant margin
- rv64ifd_zcb and rv64ifd_zce work the best for compressing integer loads and stores
- rv64ifdc and rv64ifd_zcd are the only extension targets which achieve compression on floating point loads and stores

While the above information is useful, it could be easily predicted by carefully reading the specification. What this data does highlight is where the currently provided extensions thrive, and where there still might be room for improvement. We can see that all of the extension targets perform excellently for integer computation. Unconditional control transfers and all loads/stores can be compressed well, with the correct target choice. Conditional control transfer instructions aren't compressed to the same degree as the others. We can also imagine an entire row of beige 1s at the bottom of the table for FP computation instructions. There is no compression support for this category of instructions. As we saw from Figure 4.4, there is some indication that floating point intensive applications could achieve much better compression if there was support for this instruction category.

4.2 Considerations and Trade-offs

The results in the previous section detail what we can expect to achieve from the RVC and Zc* extensions in terms of code size reduction. It is however, important to remember the original results we saw in Section 3.1, when we looked at file size compression. Recall that these were significantly larger, ranging from 0.87 to 1.0001. While these are far from the optimal compression ratios achievable, this does emphasise the importance of other compression factors, unrelated to the ISA choice. The object dump of lame, the largest benchmark program we compiled, went from 47592 to 23667 lines in removing the standard library code. This means standard library functions, most of which weren't called, accounted for about half of the code size. Based on this observation my advice for embedded system development is:

- Define all the necessary functionality for the application yourself instead of including standard libraries, so that unnecessary code isn't present in the executables
- Ensure all executables are dynamically linked, so library code isn't repeated across multiple programs
- After compiling, check the object-dumps to make sure everything has been compiled to the right target and linked as expected

Another important factor in reducing program size is compiler and linker optimisations. Compile-time flags for RISC-V that can reduce code size include -Os and -msave-restore. These flags are supported by GCC and LLVM. I explained what -msave-restore does in Section 3.1. -Os implements many code size optimisations including function inlining to reduce the number of jal instructions, consolidating re-

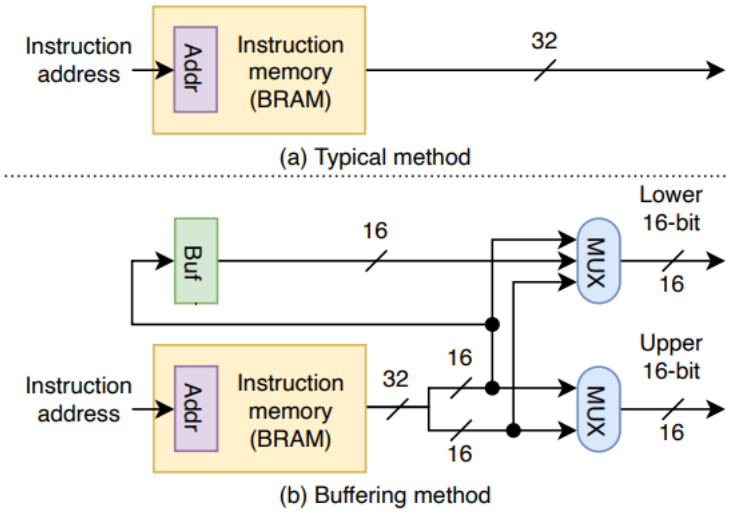

Figure 4.6: A typical instruction fetching mechanism for riscv32i (a) and for riscv32ic (b) [12]

peated code into shared functions and ensuring there is no loop expansion which is sometimes implemented for performance speed optimisation. GCC's LD provides the additional link-time optimisation -flto. This unifies compile-time code size optimisations across all linked object files, instead of treating them individually. The garbage collection flag -gc-sections can also improve code size by eliminating any unreferenced functions from the object files [15].

When designing an embedded system to comply with requirements, it is important to understand any other requirements an ISA decision might impact. We will therefore examine the functionalities provided by the RISC-V code compression extensions and identify the extra hardware costs they impose. We have already seen that support for Zcmt specifically, adds the extra hardware requirements of a (256 x XLEN) JVT and a CSR which points to the base of it. Therefore, it is important to identify whether the memory reduction achievable by Zcmt on your application is worth the adding the required infrastructure. Moreover, supporting any mixture of 16 and 32 bit instructions demands extra hardware components and comes at a potential performance cost. We will exemplify this by examining an instruction fetching mechanism for a RISCV32IC processor. Figure 4.7 shows instruction fetch units for a RISC-V architecture that supports compressed instruction encodings and one that doesn't. The architecture without compressed instruction support indexes into into BRAM instruction memory with the (address in pc)+4. Once loaded, the instruction is sent to a pipeline register to be decoded in the next cycle. The complexity of instruction fetching with a compressed ISA comes from the fact that 32 bit instructions can straddle word boundaries. In the variable length instruction fetch unit, pc+4 indexes into instruction memory to retrieve the next word. If the the word is a 32 bit instruction, it behaves the same. If the first 16 bits are a compressed instruction, they are sent to be decoded. The other 16 bits are often the lower half of a 32 bit instruction. In this case they're stored in a buffer until the next cycle when they're combined with the upper 16 bits before moving along

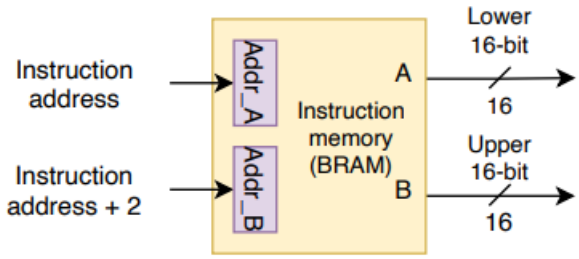

Figure 4.7: Kanamori and Kise's proposed TIF unit [12]

the pipeline. This reduces the average executed instructions per cycle (IPC) due to an incurred branch penalty. When a branch is taken, the buffer is not usable. Therefore, if the next instruction is not word aligned, it will not be decoded until the third cycle after the branch. Additionally, the data path is extended by extra logic operations which identify the length of the next instruction. This can decrease the operating frequency as instruction fetching is often the critical path in a lightweight CPU, due to having complex indexing logic. In 2021 Kanamori and Kise proposed the RVCoreP-32IC. This processor utilises a two port instruction fetch (TIF) to access the next two 16 bit values in memory, in parallel. The TIF unit dedicates port A to compressed instructions or the lower 16 bits uncompressed instructions, and port B to the upper 16 bits of an uncompressed instruction. This removes the branch penalty because the next instruction can always be fetched in one cycle. While the logic complexity of the TIF unit is greater than that of a uniform length instruction fetching unit, it has a shorter data path than preceding variable length instruction fetch units without a branch penalty, like the banked instruction cache fetch unit that's implemented for some CISC architectures. The RVCoreP-32IC achieved a 10.8% FPGA performance speed improvement on an existing RISCV32IC processor design VexRiscv which implements a buffered fetch mechanism like the one in Figure 4.7 [12]. Even with an architecture like RVCoreP-32IC, there is potentially a small performance penalty compared to an uncompressed architecture. Furthermore, it requires an extra read port in instruction memory and more complex implementations of fetching, decoding and branch prediction units, all of which could increase the price of the processor.

4.3 Proposing RISC-V ISA Improvements for Code Compression

In Section 4.1 we made two important observations regarding the effect of RISC-V's compressed ISA extensions on floating point intensive systems. The first of these being that C/Zc* compression produces very different results on floating point intensive systems than any other embedded application type we looked at. Differences included reduced average compression rates, a large increase in the compression contribution from Zcd and a substantial decrease in the compression contribution from Zca. We also observed that there is no compression support for floating point computation operations.

We noted that it's possible that the decrease in overall compression achieved and the decrease in Zca's contribution, is due to a decrease in integer computation instructions and an increase in floating point computation instructions. The collected data confirms this hypothesis showing that in FP intensive applications, integer computation instructions make up 25.4% of the code and FP computations make up 22.8% of the code. Across all other application types, integer computation instructions make up 48.2% of the code and FP computation instructions make up 4.5% of the code.

I counted the number of instances of each floating point instruction across all the compiled benchmarks. The results show:

- The most frequently occurring floating point instruction is fld followed by fsd, accounting for 3.4% and 1.4% of the uncompressed code body respectively
- The next most frequent are fmul.d and fadd.d accounting for about 1% each
- Fifth, sixth, seventh, and eighth place are taken by flw, fmadd.d, fsgnj.d, and fsub.d Each of which make up 0.5% to 0.6% of the uncompressed instructions.

It's important to remember that most of the benchmarks are light on floating point operations, so these percentages would be larger if exclusively floating point intensive applications were examined. Moreover, the benchmarks were compiled to a 64 bit ISA thus, the prevalence of double precision operations over single precision. Double precision FP values aren't supported in a 32 bit ISA, so we would likely see single precision instructions replacing their double precision counterparts in the rankings with a 32 bit target.

Based on these observations, I propose the new RISC-V ISA extensions Zcfa and Zcda. These extensions are designed to be new modules of the Zc* extensions to be used in floating point intensive systems. Zcfa and Zcda each consist of four analogous instructions, encoded with the CR format, for single and double precision floating point computations respectively:

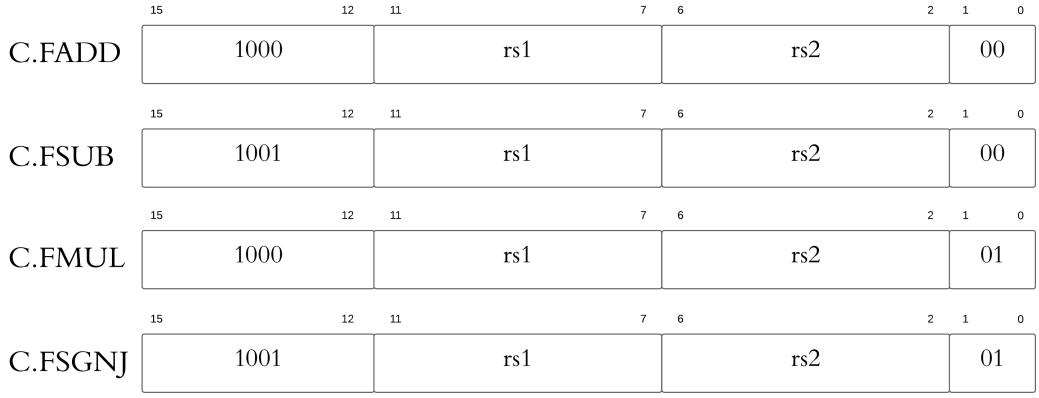

Figure 4.8: The proposed instruction encodings for Zcfa and Zcda

Instruction Name	Description	Base ISA Expansion
C.FADD	Add the floating point values in rs1 and rs2, store the result in rs1	fadd.(s/d) rs1 rs1 rs2
C.FSUB	Subtract the floating point value in rs2 from that in rs1, store the result in rs1	fsub.(s/d) rs1 rs1 rs2
C.FMUL	Multiply the floating point values in rs1 and rs2, store the result in rs1	fmul.(s/d) rs1 rs1 rs2
C.FSGNJ	Put a value consisting of the sign bit from rs2 and the rest of the bits from rs1 in rs1	fsgnj.(s/d) rs1 rs1 rs2

The four encodings are identical in Zcfa and Zcda, meaning the extensions are not compatible with eachother. This was due to a lack of available opcodes and to remove the fmt field which usually specifies the FP precision level. The encodings also omit the rm field which is present in the uncompressed equivalent to specify the rounding mode. The rounding mode is instead discovered dynamically with the floating point control status register (FCSR). This behaviour is equivalent to if the rm field contained the bits 11 in the 32 bit expansion. I defined these instructions with opcodes reused from the Zcb extension, making them incompatible. This trade-off is justified by the observation that Zcb instruction formats contribute the least to compression out of the Zc* extensions other than Zcmt (1.08% on average). We didn't manage to get concrete data for Zcmt and it borrows opcodes from the Zcd extension, so it isn't a viable choice for opcode sacrifice.

In order to assess the potential compression improvement Zcfa and Zcda can achieve on FP intensive applications, I calculated the theoretical code size reduction that would

be achieved by substituting Zcb for Zcda in the rv64ifd_zce executable of Whetstone. To do this, I went through the code and subtracted 16 from the total code size for every fadd.d, fsub.d, fmul.d or fsgnj.d instruction where rd is equal to one of the register operands. I also added 16 to the total code size for every Zcb instruction. In doing this I found that the code size reduces from 1262 bytes to 718 bytes, with 34 compression candidates and 0 decompression candidates. This is a 43.1% further reduction in code size. While promising, this result is not sufficient to confidently assert that these extensions will cause a large compression enhancement for any FP intensive application. Whetstone, the program under test, is very small without it's linked library functions and doesn't receive any Zcb code size optimisation. To more accurately analyse the impact of Zcfa and Zcda, this test would need to be done on numerous large FP intensive applications which benefit from Zcb.

Chapter 5

Conclusions

In this project report, I have measured and compared the compression capabilities of the RISC-V C and Zc* ISA extensions, on benchmarks representing common embedded applications. I used collected data to predict the variance in the compression achieved by these extensions, depending on the compiled program's content. Using these analyses, I identified a potential compression optimisation, for RISC-V implementation of systems with a heavy floating point workload. Based on this finding, I proposed the new RISC-V ISA extensions Zcfa and Zcda, with the intention of improving RISC-V's compression capability for floating point intensive programs.

In a direct compression comparison of RISC-V targets which specify one of the RVC/Zc* extension options, I found that the Zce target performs the best, followed by Zcmp, RVC, Zcd, then Zca. A flaw in my experimental method hindered me from placing Zcmt in the rankings. When I examined how these rankings might change depending on the implemented application type, I found that the targets perform similarly for automotive/control-systems, consumer, office, networking, security, and telecommunications applications. The only outlier was floating point intensive applications. The results indicated that this application type receives less compression than the rest. The targets' rankings also changed to, RVC and Zcd in the first and second spots with the rest trailing behind by roughly 7%, for floating point intensive applications. Next, I assessed the ISA targets' compression performance on different instruction categories. This illustrated the RVC/Zc* extensions' strengths and weaknesses. The data showed that the most compression is achieved on integer computation instructions, and the least compression is achieved on floating point computation instructions i.e. none. The latter of the observations is due to the fact that, the RVC/Zc* extensions provide no compression support for floating point operations beyond loads and stores.

I proceed to propose the Zcfa and Zcda extensions. These implement 16 bit encodings for floating point multiply, add, subtract and sign-injection operations. I found that these were the most frequently occurring floating point computation operations in my collected data. I calculated Zcda's theoretical effect on a Zce compilation of Whetstone, finding that it could decrease the code size by a further 43%. This result is very promising, but a much more thorough test is needed.

While collecting the experimental data, I identified a few important software considerations for minimising compiled program size. The most impactful of these were library implementation and linker configuration. I found that negligence of these factors can more than double the executable size of a large software project. The code size can also be significantly reduced by implementing compiler and linker optimisations with command-line flags.

By examining RISC-V processor designs which support compressed instruction formats, I discovered that using the RVC/Zc* ISA extensions can come at a performance cost. This must be taken into account to make the best ISA choice for your system's set of requirements.

This experiment was not without limitations. Due to the complicated cross-compilation process and limited time, the integrity of the gathered data isn't as strong as it could have been. My hope is that continued research in this field can verify my results and further justify my extension proposal.

This paper only discusses one small area in the vast, ever growing and changing landscape that is RISC-V. It is inevitable that the ISA will become more diverse and configurable, to keep up with the increased demand for specialised processors. I sincerely hope Zcfa and Zcda, or some improved version of them can assist in this movement.

Appendix A

Implementing Support for Znew Into the RISC-V GNU Tool-chain

In October 2023, Qualcomm Technologies proposed an idea for a RISC-V code size reduction extension labelled as Znew. The motivation of the Znew extension is to reduce the static code size of RISC-V compiled programs by combining common instruction sequences into single 32 bit instructions. One notable advantage of this approach compared to using 16 bit instruction encodings, is that the ISA can maintain a uniform instruction size. This can simplify architecture's instruction fetching and decoding mechanisms by removing extra logic and control signals such as those implemented in RVCoreP-32IC [12]. All instructions in the specification are implemented with 'brownfield' opcodes. This means that rather than occupy unused opcode values, the instructions reuse existing opcodes. The decoder distinguishes the instruction variant with funct fields. Znew presents two novel variants of load/store instructions. The load/store update instructions execute the desired load or store and then add the offset to the base-address register. This reduces the number of instructions needed for common programming tasks such as sequentially accessing each element in an array. They present pre and post update variants, depending on the desired order of operations. Due to using 'brownfield' opcodes, these instructions require a funct7 field which loads and stores in the base ISA don't. This means that the offset is 5 bits wide instead of 12, severely limiting the offset range in the register-immediate addressing mode. The pro-

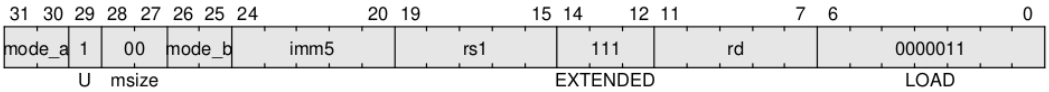

Figure A.1: The encoding for a lb instruction with Register-Immediate Addressing. mode_a:mode_b specifies the update variant (no-update,pre-update or post-update) [5]

posed register-register addressed load/store instructions use a register field to specify the offset instead of an immediate. This provides a XLEN bit offset, and only occupies 5 bits in the instruction. The one downside is that an extra instruction is required to move the offset into a register but, as long as the offset is used for more than one

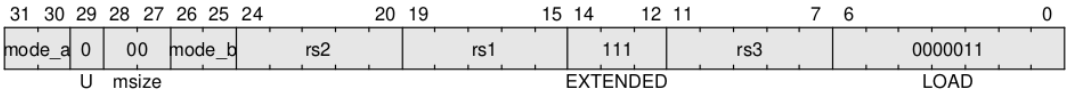

Figure A.2: The encoding for a sb instruction with Register-Register Addressing. [5]

load/store update, there is still a net instruction decrease. This addressing mode is also useful for when the desired offset is not known during runtime and needs to be retrieved from memory. The register offsets can be optionally scaled to the transfer size of the instruction in order to elide additional shift instructions needed for offset alignment in the update phase. The load/store pair instructions atomically execute two loads/two stores to sequential memory addresses. The addresses in these instructions are SP relative. They specify two destination registers for load pairs and two source registers for store pairs. The offset is again limited to a 5 bit immediate and there is no register-register addressed alternative. Similar to the Zcmp push/pop instructions, the motivation for such instructions is to reduce the the number of required instructions for function prologues and epilogues, where many local variables are usually pushed to/popped from the stack. The proposal also facilitates update variants of load/store pairs. Next, the document presents move pair instructions. These instructions move

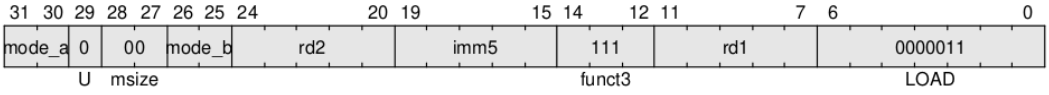

Figure A.3: The encoding for a lbp instruction (load byte pair) [5]

two specified source registers into a0 and a1 or into a2 and a3. In other words, a move pair instruction loads two arguments in preparation for a function call. This is similar in motivation to Zcmp's cm.mvsa01 but rather than storing arguments to the stack, it executes the preceding step of moving values into the ABI argument registers. Finally,

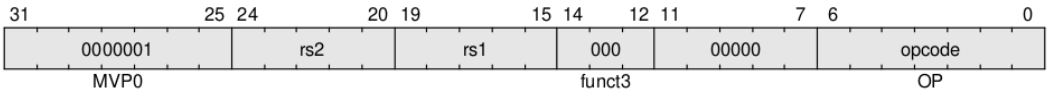

Figure A.4: The encoding for a mvp0 instruction (The a0, a1 variant of the move pair instruction) [5]

the document points out another common instruction sequence where a small constant is assigned to a register with an addi instruction, followed by a conditional branch with respect to this constant. In response to this, they propose conditional branch instructions beqi and bneqi. These have a second immediate field where a constant to be conditioned on is specified [5].

In the early exploratory phase of this project, I had a different vision for what I hoped to achieve. I was inspired by the discovery of the Znew extension and decided to implement support for this extension into GCC, in hopes to make a valuable contribution to the RISC-V community. Ultimately, I discarded this project idea for the project laid

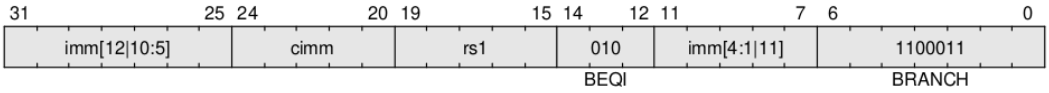

Figure A.5: The encoding for a beqi instruction [5]

out in the main body of this paper. I realised that it was naive to attempt this for such an early proposal. I found many valid criticisms of the specification in the forum where it was posted, and came to the conclusion that it will either be discarded or undergo major changes before ratification is even discussed. Therefore, to add compiler support at this point in development wouldn't be productive. Furthermore, Such an implementation is a tall task involving a steep learning curve for someone of my skill-set and I was not entirely confident in my ability to produce something close to a completed project. Nonetheless, there are many valuable takeaways from this attempt. In this appendix, I aim to give an overview of the task and summarise my development activities.

The most common understanding of a compiler is a program that translates a high-level programming language into assembly language although, the target of a compiler can be any other language. GCC (GNU Compiler Collection) implements compilers which translate C, C++, Objective-C, Objective-C++, and others into assembly language for a wide range of architectures. When GCC is invoked, a preprocessor is called, which includes header files, removes any comments, tokenises the program and expands any header macros. It is is now ready to be compiled. Compiling involves creating an annotated AST (Abstract Syntax Tree) from the program and turning this into assembly code which is passed on to an assembler. The task of an assembler is to translate compiled assembly language programs into binary object files. GAS (GNU Assembler) provides a family of assemblers, one for each of the targets supported by GNU. Once invoked, GAS selects one of these assemblers based on it's configuration. It then assembles one or more specified source files into a single object file output, given that the sources are in the correct assembly language format [6]. Once assembled, the desired object files and any included external libraries are linked together using LD (GNU Linker) into a single executable file [8]. With this understanding, we can begin to identify which parts of the GNU tools need to be altered to add support for a new ISA extension. It is clear that everything up to and including generation of the AST is not impacted by the ISA target, this is also true for the linking process. This narrows down the necessary alteration areas to the RISC-V assembler in GAS and the RISC-V code generation unit in GCC.

The first step of my implementation was to download the open source RISC-V GNU tool-chain from GitHub [17]. This repository contains architecture cross-compatible implementations of GCC, Binutils, GDB, GLIBC and LLVM which target RISC-V. It also contains RISC-V execution tools such as Spike and QEMU. I began development by extending GAS. I found an article which outlines in detail how to add new RISC-V instructions into GAS [1]. It explains that in order to accomplish this, first you need to add entries for each new instruction, to the RISC-V encodings table located in binutils/opcodes/riscv-opc.c. Such an entry consists of:

```
{"znew.lb",    0, INSN_CLASS_ZNEW, "Nd, No(+Ns)", MATCH_ZNEW_PRE_LB_RI,MASK_ZNEW_UPDT_LOAD, match_opcode, 0},
{"znew.lb",    0, INSN_CLASS_ZNEW, "Nd, No(Ns+)", MATCH_ZNEW_POST_LB_RI,MASK_ZNEW_UPDT_LOAD, match_opcode, 0},
```

Figure A.6: Encoding table entries for the pre update and post update variants of Znew.lb

```
#define MASK_ZNEW_UPDT_LOAD 0xfe00707f
#define MATCH_ZNEW_PRE_LB_RI 0x2007003
#define MATCH_ZNEW_POST_LB_RI 0x4007003
```

Figure A.7: The match and mask value definitions used in the the table entries shown in Figure A.6

- The instruction name
- Whether its supported for the 32 bit ISA, the 64 bit ISA or both
- The instruction class e.g. Znew
- The instruction operands
- The match and mask
- The match function
- Pinfo

The match and mask are bit values used to identify a binary encoding as the listed instruction. The match function defines the bitwise operation which takes the instruction, mask and match as inputs and returns an instruction identity. Usually, the AND of the binary encoding and the mask are computed and if the result equals the match, then it is recognised as the listed instruction. Pinfo is a bit value which describes any relevant hazard information. The default value when there is no hazard concerns is 0. To complete these table entries, I needed to add Znew to the instruction class enum definition in binutils/include/opcode/riscv.h. I also had to write definitions for the matches and masks in binutils/include/opcode/riscv-opc.h. To do this, I identified the relevant bits for decoding e.g. funct fields and the opcode. The mask should be defined with 1s in the positions of these identifying bits and 0s elsewhere. The match should consist of the expected identifying bits, also with 0s elsewhere.

The final step to complete the Znew implementation in GAS, was to add support for each instruction into the print_insn_args function, located in binutils/opcodes/riscv-dis.c. This function disassembles a bitstream of machine code back into assembly language. Unfortunately, I changed project idea before I was able to make any notable progress in this task.

Appendix B

Historical Overview of Code Compression Strategies

In this appendix I intend to give a select overview of developments in the code compression research over the past thirty years or so. I would like to preface by stating that, although I have cited the original papers, a large portion of the information comes from a very comprehensive and well laid out paper surveying the history of code compression research in the 90's and early 2000's [3].

Code compression is a specialised field of data compression. Often, code compression can achieve a better compression ratio than generally applicable data compression because, known structural details of code-data can be exploited to further optimise size reduction. Compression of code through generally applicable data compression techniques is referred to as ISA-independent. Code compression which relies on know structural details of code instructions is called ISA-dependent.

In 1992 Wolfe and Chanin tackled the problem of the large machine-code size generated by Reduced Instruction Set Computers in order to make RISC architectures more compatible with memory critical embedded systems. They proposed the idea of a CCRP (Compressed Code RISC Processor). In such a processor, the embedded code is compressed with some good statistical compression method before being stored in memory. A hardware decompression mechanism translates the instructions back to their original form on a cache-miss. Architectures that decompress code in this way are now referred to as pre-cache architectures. During compression, a LAT (Line Address Table) is created. This is used to map memory addresses to their original value and is stored in memory alongside the compressed code. They also propose the use of a CLB (cache look-aside buffer) to cache recently accessed LAT entries [24].

In 1999 Benini et al puplished a paper called, Selective Instruction Compression for Memory Energy Reduction in Embedded Systems. In this paper they propose a means to save computation energy costs in embedded systems, by reducing the stored size of firmware instructions and the width of the bus carrying these instructions to the CPU. This method involves storing the 255 most frequently executed instructions in a compressed 8-bit representation and inserting an instruction decoding unit between

memory and the CPU. The decoding unit translates the instructions back into their original form before delivering them to be executed. This contrasted some of the existing industry code compaction methods at the the time, such as ARM Thumb and MIPS16. These involved defining compressed instruction encodings and implementing support for them in the processors decoding unit. Unlike these methods, this one is invisible to the CPU and compatible with a normal decoder.[2].

In 2005 Lekatsas et al presented a computer architecture customised to store compressed program executable binaries. Previous works to this primarily focused on compressing the instruction segment of such files. This paper identifies that in many cases programs consist of large proportions of data that are not program instructions. It aims to improve on previous work by compressing both the program and program data. They present an algorithm which parses input byte sequences and replaces them with smaller translation table indices. They also propose a memory framework to support this architecture which includes a buffer used to store recent index translations [13].

In 2010, Bonny and Henkel combined two of their previously proposed code compression techniques into one, which achieves an improved average compression ratio on both. The previous techniques include the instruction splitting technique which is ISA-independent and the instruction re-encoding technique which is ISA-dependent. Both of these techniques utilised Huffman coding in ways especially designed to reduce the size of the decoding table. Huffman coding is a statistical data compression technique which involves replacing individual sections of the data with shortened codewords, frequently repeating sections get assigned short codewords and less frequently occurring sections get longer codewords. A decoding table is generated during compression to translate the codewords back into their original form. Since these tables can be very large, reducing their size can have a significant impact on the data compression ratio. The aptly named combined compression technique they propose involves identifying bits of instructions that can be re-encoded in a more compact format and replacing the remaining unused bits with don't care symbols (borrowed from the instruction re-encoding technique). It then splits the instructions into variable length sections. This is done with an algorithm designed by them, that attempts to optimize the frequency that each split section occurs (borrowed from the instruction splitting technique). The don't care symbols in these sections are replaced with 1s and 0s so that they becomes identical to other sections if possible. This further reduces the number of unique sections. The remaining unique sections are compressed with Huffman coding [4].

Bibliography

[1] "Adding an Instruction to the GNU Assembler". In: (Oct. 2020). URL: https://ossg.bcs.org/blog/2020/10/29/adding-an-instruction-to-the-gnu-assembler/.

[2] Luca Benini et al. "Selective Instruction Compression for Memory Energy Reduction in Embedded Systems". In: *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*. ISLPED '99. San Diego, California, USA: Association for Computing Machinery, 1999, pp. 206–211. ISBN: 158113133X. DOI: 10.1145/313817.313927. URL: https://doi.org/10.1145/313817.313927.

[3] Árpád Beszédes et al. "Survey of Code-Size Reduction Methods". In: *ACM Comput. Surv.* 35.3 (Sept. 2003), pp. 223–267. ISSN: 0360-0300. DOI: 10.1145/937503.937504. URL: https://doi.org/10.1145/937503.937504.

[4] Talal Bonny and Jörg Henkel. "Huffman-Based Code Compression Techniques for Embedded Processors". In: *ACM Trans. Des. Autom. Electron. Syst.* 15.4 (Oct. 2010). ISSN: 1084-4309. DOI: 10.1145/1835420.1835424. URL: https://doi.org/10.1145/1835420.1835424.

[5] James Ball Derek Hower and Albert Yosher. *Code Size Reduction Instructions*. 0.1. Qualcomm Technologies, Inc. Oct. 2023.

[6] Dean Elsner and Jay Fenlason. *Using as*. 2.42. Available at https://sourceware.org/binutils/docs/as.pdf. Free Software Foundation, Inc.

[7] "Embedded Microprocessors Market". In: *Persistence Market Research* (2023).

[8] "GCC". In: (). URL: https:https://www.incredibuild.com/integrations/gcc.

[9] *GNU Manual*. 14.0.1. Available at https://gcc.gnu.org/onlinedocs/gcc/RISC-V-Options.html. Free Software Foundation, Inc.

[10] Matthew R Guthaus et al. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE. 2001, pp. 3–14.

[11] RISC-V International. *ZC* Specification*. 1.0.4. Available at https://github.com/riscv/riscv-code-size-reduction/tree/main/Zc-specification. RISC-V International. 2022–2023.

[12] Takuto Kanamori and Kenji Kise. "RVCoreP-32IC: An optimized RISC- V soft processor supporting the compressed instructions". In: *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. 2021, pp. 38–45. DOI: 10.1109/MCSoC51149.2021.00014.

[13] H. Lekatsas et al. “A unified architecture for adaptive compression of data and code on embedded systems”. In: *18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*. 2005, pp. 117–123. DOI: 10.1109/ICVD.2005.36.

[14] Peijie Li. “Reduce static code size and improve RISC-V compression”. In: (2019).

[15] Matteo Perotti et al. “HW/SW approaches for RISC-V code size reduction”. In: *Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*. 2020.

[16] *RISC-V Cores and SoC Overview*. https://github.com/riscvarchive/riscv-cores-list. 2020.

[17] *riscv-gnu-toolchain*. https://github.com/riscv-collab/riscv-gnu-toolchain. RISCV Collaboration.

[18] Manfred Schlett. “Embedded microprocessors: Evolution, trends, and challenges”. In: *Fortieth Anniversary Volume: Advancing into the 21st Century*. Ed. by Marvin V. Zelkowitz. Vol. 52. Advances in Computers. Elsevier, 2000, pp. 329–379. DOI: https://doi.org/10.1016/S0065-2458(00)80021-2. URL: https://www.sciencedirect.com/science/article/pii/S0065245800800212.

[19] inc Synopsys. “What is RISC-V”. In: *Glossary of Application Security, EDA, Semiconductor IP & Optics* (2023).

[20] *The Whetstone Benchmark*. https://www.keil.com/benchmarks/whetstone.asp. arm KEIL.

[21] Roddy Urquhart. “What does RISC-V stand for?” In: *Semiconductor Engineering* (2021).

[22] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual*. 2.2. Available at https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf. SiFive Inc. May 2017.

[23] Claire Xenia Wolf. *PicoRV32*. https://github.com/YosysHQ/picorv32/blob/main/picorv32.v. 2015.

[24] Andrew Wolfe and Alex Chanin. “Executing Compressed Programs on an Embedded RISC Architecture”. In: *SIGMICRO Newsl.* 23.1–2 (Dec. 1992), pp. 81–91. ISSN: 1050-916X. DOI: 10.1145/144965.145003. URL: https://doi.org/10.1145/144965.145003.