Reduce Static Code Size and Improve RISC-V Compression

Peijie Li

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-46
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-46.html

May 16, 2019

Reduce Static Code Size and Improve RISC-V Compression

Abstract

Reduce Static Code Size and Improve RISC-V Compression

by

Peijie Li

Master of Sciences in Electrical Engineering and Computer Sciences

University of California, Berkeley

,

Due to the evolution of technologies, embedded systems and IoT applications have become an essential part of our lives - mobile phones, smart watches, video game consoles, digital cameras, GPS - and at the same time they become more and more complex. Unlike traditional systems that are equipped with large size of memory, such applications must fit themselves inside ever-shrinking envelopes, limiting the amount of memory available to the developers. As the static code size affects the spaced used and contributes greatly to the development of complex applications and features, the study of code compression techniques is at increasing premium.

RISC-V ISA is an open-source instruction set architecture designed to be useful in a wide range of devices, and its Compression extension, named RVC, is designed to reduce instruction bandwidth of frequently occurring instructions. Aiming to evaluate the efficiency of RVC in reducing static code size of applications compiled by RISC-V Embedded GCC and potentially further improve compression rate, this report proposes several changes to the existing RVC extension. The new RVC extension is expected to compress 10% more program instructions and fetch 5% fewer instruction bits than RVC programs.

Contents

Contents **i**

List of Figures **ii**

List of Tables **iii**

1 Introduction **1**

2 Background and Related Work **2**

3 The RISC-V Approach **4**
3.1 RVC: A Variable Length RISC-V ISA 4
3.2 Optimizing Register Save/Restore Code Size 4
3.3 RVC Optimization Design Methodology 7

4 Optimizing RISC-V C Extension **8**
4.1 Performance of RVC 8
4.2 Add-Immediate Encoding 13
4.3 Control Transfer Instruction Encoding 17
4.4 Branch Control Transfer Instruction Encoding 18
4.5 Load and Store Instruction Encoding 21
4.6 Implementation Consideration 23

5 Evaluation **24**

6 Conclusion **27**

Bibliography **28**

List of Figures

3.1 Compiler-generated function calls that handle register save/restore 5

4.1 Performance Result of RVC compiling benchmarks from various benchmark suites. . . 8
4.2 Uncompressed RISC-V Instruction, statically, seen in the set of benchmark suite. . . . 10
4.3 Cumulative distribution of immediate operand (size and whether in multiple of half-words, words, etc) in a subset of the combined benchmark suite. 11
4.4 Distribution of instruction operators and immediate operand sizes in a subset of the combined benchmark suite. 12
4.5 RVC instruction encodings for ADDI. rd specifies a destination register, while rs1 specifies source registers. The funct fields are additional opcodes. 13
4.6 Distribution of immediate operands seen across uncompressed Group A/B/C ADDI instructions 15
4.7 Register Usage Among ADDI operations where rs1 == rd 16
4.8 Proposed 16-bit instruction encoding for ADDI operations. 16
4.9 RVC instruction encoding for unconditional jump operations. 17
4.10 Immediate Operands Among Uncompressed J/JAL operations. 17
4.11 Proposed 16-bit instruction encoding for Jump operations. 18
4.12 RVC instruction encoding for branch operations. 18
4.13 Immediate Operands seen among Uncompressed branch control transfer operations. . . 20
4.14 Proposed 16-bit instruction encoding for branch operations. 20
4.15 Percentage of operator usage among uncompressed data-transfer operations. Majority of data transfer instructions are register-based. 21
4.16 Registers Among Uncompressed (a) Word-size and (b) Byte-size Load and Store operations. 22
4.17 Immediate Operands Among Uncompressed Register-based word/byte loads and stores. 22
4.18 Proposed 16-bit instruction encodings for LD/ST operations. 23

5.1 Instruction Compression Rate With and Without proposed changes to RVC. 25
5.2 Compression Rate With and Without proposed changes to RVC. 26

List of Tables

3.1 Benchmark used for experiments 6

4.1 Most frequent RVC instructions (as expressed in percentage of total instruction count in benchmark program), statically, in a subset of the ZephyrRTOS, FreeRTOS, MiBench, MediaBench benchmarks. 9
4.2 Uncompressed ADDI operations based on whether its register references satisfy one of the RVC encoding. 14
4.3 Frequency of register usage of uncompressed ADDI operations. Approximately 50% of uncompressed ADDI instructions reference only rvc-registers. 14
4.4 Operator Distribution and Register Usage seen among Uncompressed branch control transfer operations. 19

Chapter 1

Introduction

Embedded systems are ubiquitous and indispensable from our daily life today. In contrast to traditional programs that are given large size of memory, embedded applications - such as smart watches and GPS - only have very limited memory where they can store the object code. Hence the static code size plays a crucial role in the development of modern complex embedded applications: The smaller the generated code size, the more features a software can implement. It is imperative to improve existing code compression technique or to design new compression algorithms in order to further reduce code size for such applications.

Similar to data compression techniques, code compression algorithms aim to reduce redundancy and increase the amount of content in a given block of information. However, code compression is particularly more challenging because the data being compressed are a series of instructions that together compose an executable program. Hence code compression techniques must satisfy additional restrictions, retrieve certain information efficiently as well as make the code size smaller. In particular, code compression algorithms must provide a decompression technique that can retrieve branch targets and function entry points correctly and quickly.

There has been extensive work looking for an efficient code compression technique for program code in general. Researchers developed reduced instruction set architecture to reduce code size in trade of the number of instructions in compiled programs. Starting in late 1900s, several dictionary-based code-compression techniques as well as statistical methods have been proposed to compress compiler-generated static program code size. For example, MIPS and RISC-V, two instruction set architecture based on this reduced instruction set computer principles, provide compression extensions to further improve program code size. The RISC-V C compression extension, named RVC, is designed to replace commonly reoccurring operations with a shorter 16-bit instruction encoding. Unlike standard dictionary-based approaches that require additional memory space to store information necessary to decompress instruction code-word, RVC contains sufficient information in its 16-bit encoding to retrieve original operations at run time without storing data in memory.

In this paper, we examine and evaluate the efficiency of RVC in compiling embedded and IoT program applications. We then propose a few modifications to the RVC encoding set. Lastly, we evaluate the performance of RVC with our proposed modifications.

Chapter 2

Background and Related Work

There has been extensive study on code-compression technique for embedded software program applications. In 1992, Wolfe and Chanin [25] proposed the very first code-compression technique that use huffman coding algorithm to compress instruction bits. The decompressor references a so-called Line Address Table (LAT) to retrieve original instruction block addresses using compressed codeword.

Wolfe and Chanin's algorithm inspired the study and implementation on dictionary-based code compression techniques that take advantages of reoccuring instruction sequences observed across program applications [9, 16, 17]. IBM used CodePack in its embedded PowerPC Systems [5]. Lefurgy proposed to use a post-compilation compressor that replaces common sequences of instructions with a single instruction codeword [13, 12]. His technique stores the encoded instruction codewords in a dictionary which, in turn, is used at execution time to expand the codewords back into the original sequence of instructions. Since then, many researchers worked on improving dictionary-based code-compression approaches. Prakash and Ros extends the technique to use Hamming distance to compress instruction sequence that differ in a few bit positions [18, 19, 20]. Seong and Mishra uses bitmask patterns to aggressively create more matching sequences before applying the dictionary-based technique in order to improve the compression ratio [21]. Collin proposed a 2-level dictionary-based approach to further encode compressed instructions into compressed sequences [4].

The technique discussed so far targeted program code either at bit level or instruction level. There also has been significant amount of research on separate compression of instruction operator and operands. Araujo introduced operand factorization, the separation of program expression trees into sequences of tree-patterns (operators) and operand patterns (registers and immediate operands) [1]. Lekatsas and Wolf furthers uses arithmetic coding and Markov model to compress tree-patterns and operand patterns separately [14] [15]. Recently Bonny introduced Combined Compression Technique that splits individual instruction into portions of varying size before Huffman coding is applied [2].

The efficiency of dictionary-based code compression techniques are limited [22] because they store additional information in memory at compile time so that at execution time the de-compressor can use it to retrieve the original program code. In order to reduce and further eliminate the use

of extra memory space, a new variable-length instruction encoding technique was proposed and widely implemented in reduced instruction set architectures. MIPS implements a compressed MIPS16 ISA [8, 7]; the ARM processor has an additional condensed "*Thumb*" instructor set [26]; Waterman proposed a similar compression extension for RISC-V ISA, named RVC. Their technique adds short 16-bit instruction encoding for commonly reoccurring operations [23, 24]. The RVC extension is reported to cover 50% - 60% of the RISC-V instructions in a program, resulting in a 25% - 30% code-size reduction.

Additionally, procedural abstraction has been used to take advantage of common sequences of instructions that appear in different sections across program code. Lao's "echo-instruction" is designed to compress these repeating sequences of instructions by replacing them with an "echoing" instruction that executes existing sub-sequences of code sequences from other locations in the program [10]. RISC-V and MIPS also implements a save-restore routine to compress the register save/restore code at function entry/exit, which represents a significant portion of static code size [24, 3].

In the following sections, we first provide an evaluation of RVC, RISC-V's current compressed instruction set extension (Chapter 3). Then we propose several modifications to RVC (Chapter 4) and lastly present a performance evaluation for the modified RVC (Chapter 5).

Chapter 3

The RISC-V Approach

3.1 RVC: A Variable Length RISC-V ISA

RISC-V is a free and open-source instruction set architecture designed to be useful in a wide range of devices, including but not limited to embedded systems. The RISC-V ISA contains an extension for compressed instruction, named RVC, designed to reduce static and dynamic code size by substituting common instructions with shorter 16-bit instruction encoding[1]. For example, a 32-bit stack-pointer-based load (or store) with an immediate representable in 8 bits are replaced by a 16-bit RVC instruction, reducing the size in half. At run-time the de-compressor will expand a RVC instruction into a single 32-bit instruction in the base ISA. RVC compresses 32-bit RISC-V instructions that satisfy one or more of the conditions:

- the immediate or address offset is small,
- one of the registers is the zero register, the ABI link register, or the ABI stack pointer,
- the destination register and the first source register are identical,
- the registers used are the 8 most popular ones (”denoted as rvc-registers”).

3.2 Optimizing Register Save/Restore Code Size

The RISC-V compiler supports procedural abstraction to optimize register save/restore code size at function entry/exit. This optimization is designed to reduce the redundancy of memory access instructions seen in function prologues and epilogues. With the *-msave-restore* flag, the compiler replaces prologues and epilogues with a jal instruction, executing compiler generated procedures that automatically handles grouped register saves and restores. Figure 3.1 shows part of .text section obtained by compiling the qsort benchmark with *-msave-restore* flag. At function entry point one can simply jump to procedures such as *riscv_save_0* to save register $s0$ - $s2$, along with return-address register ra, onto the stack. One could also call *riscv_save_4* to save registers $s0$ - $s6$

and ra onto the stack, or *riscv_save_10* to save additional registers $s7$ - $s10$ (it jumps to *riscv_save_4* to store register $s0$ - $s6$), or *riscv_save_12* (which reuses instruction blocks from *riscv_save_10* and *riscv_save_4*) to save all 12 $s*$ registers. Similarly, *riscv_restore_** handles automatic restore of saved registers.

```
0001342a <__riscv_save_12>:
   1342a:   7139                    addi    sp,sp,-64
   1342c:   4301                    li  t1,0
   1342e:   c66e                    sw  s11,12(sp)
   13430:   a019                    j   13436 <__riscv_save_10+0x4>

00013432 <__riscv_save_10>:
   13432:   7139                    addi    sp,sp,-64
   13434:   5341                    li  t1,-16
   13436:   c86a                    sw  s10,16(sp)
   13438:   ca66                    sw  s9,20(sp)
   1343a:   cc62                    sw  s8,24(sp)
   1343c:   ce5e                    sw  s7,28(sp)
   1343e:   a019                    j   13444 <__riscv_save_4+0x4>

00013440 <__riscv_save_4>:
   13440:   7139                    addi    sp,sp,-64
   13442:   5301                    li  t1,-32
   13444:   d05a                    sw  s6,32(sp)
   13446:   d256                    sw  s5,36(sp)
   13448:   d452                    sw  s4,40(sp)
   1344a:   d64e                    sw  s3,44(sp)
   1344c:   d84a                    sw  s2,48(sp)
   1344e:   da26                    sw  s1,52(sp)
   13450:   dc22                    sw  s0,56(sp)
   13452:   de06                    sw  ra,60(sp)
   13454:   40610133            sub sp,sp,t1
   13458:   8282                    jr  t0

0001345a <__riscv_save_0>:
   1345a:   1141                    addi    sp,sp,-16
   1345c:   c04a                    sw  s2,0(sp)
   1345e:   c226                    sw  s1,4(sp)
   13460:   c422                    sw  s0,8(sp)
   13462:   c606                    sw  ra,12(sp)
   13464:   8282                    jr  t0

00013466 <__riscv_restore_12>:
   13466:   4db2                    lw  s11,12(sp)
   13468:   0141                    addi    sp,sp,16

0001346a <__riscv_restore_10>:
   1346a:   4d02                    lw  s10,0(sp)
   1346c:   4c92                    lw  s9,4(sp)
   1346e:   4c22                    lw  s8,8(sp)
   13470:   4bb2                    lw  s7,12(sp)
   13472:   0141                    addi    sp,sp,16

00013474 <__riscv_restore_4>:
   13474:   4b02                    lw  s6,0(sp)
   13476:   4a92                    lw  s5,4(sp)
   13478:   4a22                    lw  s4,8(sp)
   1347a:   49b2                    lw  s3,12(sp)
   1347c:   0141                    addi    sp,sp,16

0001347e <__riscv_restore_0>:
   1347e:   4902                    lw  s2,0(sp)
   13480:   4492                    lw  s1,4(sp)
   13482:   4422                    lw  s0,8(sp)
   13484:   40b2                    lw  ra,12(sp)
   13486:   0141                    addi    sp,sp,16
   13488:   8082                    ret
```

Figure 3.1: Compiler-generated function calls that handle register save/restore

Benchmark	Static Program Instructions	Compression Rate
epic	28545	70.62%
jpeg	57973	69.98%
adpcm	9732	66.84%
mpeg	38351	71.89%
g721	8951	67.70%
basicmath	15918	70.83%
dijkstra	5059	67.60%
fft	8752	70.30%
crc	3920	66.75%
linpack	6838	71.02%
qsort	7616	70.84%
stringsearch	3167	65.90%
dhrystone	6478	69.61%
sha	3609	66.13%
bitcnts	4848	67.14%
ghostscript	6947	69.65%
rijndael	7586	71.16%
whetstone	7643	70.28%
susan	16037	71.79%
pi_css5	31881	66.85%
pegwit	11236	74.03%
freertos	9441	64.57%
cppsynchronization	2430	68.02%
philosophers	2847	72.00%
mpu	1938	68.55%
synchronization	2122	68.10%
grove	2531	69.79%
sensor	29239	70.68%
driver	19462	72.35%

Table 3.1: Benchmark used for experiments

3.3 RVC Optimization Design Methodology

To evaluate RVCs effectiveness on reducing static embedded code size, we collected static measurements from a set of embedded benchmarks [11, 6]. Table 3.1 lists the benchmarks, their static instruction counts and compression rate when compiled for RISC-V *-march=rv32imac -mabi=ilp32*.

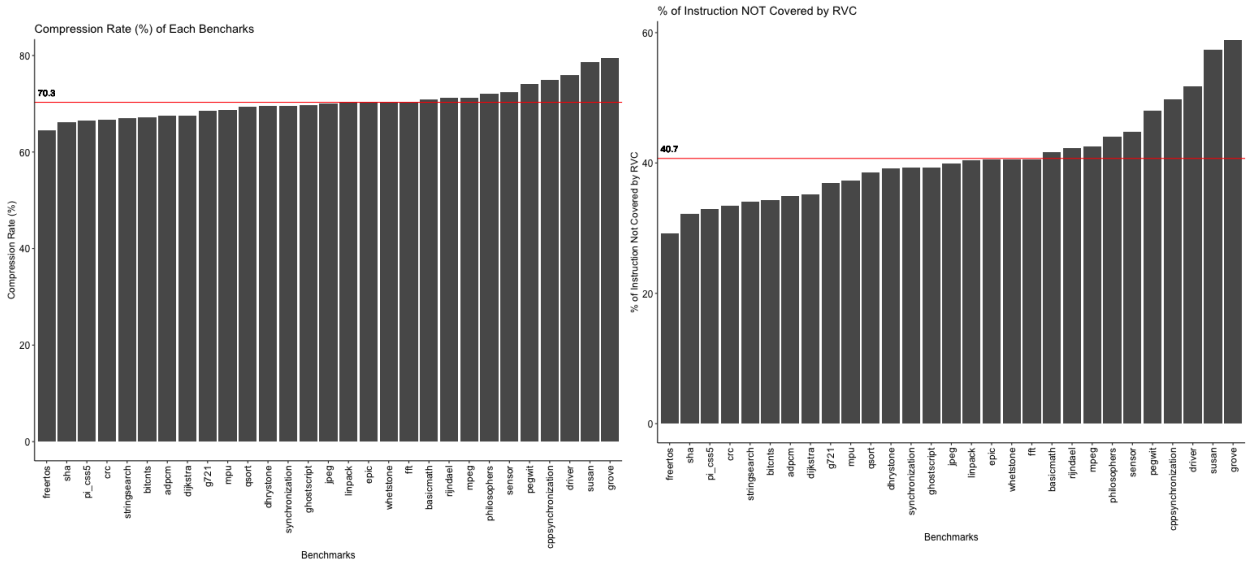
All benchmarks were compiled with a GNU MCU Eclipse RISC-V Embedded GCC, 32-bit 8.1.0 compiler, optimizing for size and more and linker-time optimization (*-O3 -Os -msave-restore -flto*). Static measurements were obtained directly from the resulting executables.

Chapter 4

Optimizing RISC-V C Extension

4.1 Performance of RVC

We compiled the embedded benchmarks with 32-bit RISC-V Architecture *rv32imac* tool-chain, optimization flags *-O3 -Os*, *-msave-restore* together with link-time optimizer (-flto) to evaluate the performance of current RVC proposal. Figure 4.1a shows that RVC reduces static code-size by 20%-35%, giving a 65%-80% compression rate. On average, 60% of the RISC-V instructions in an embedded program can be replaced with RVC instructions, resulting in a 30% static code-size reduction.



(a) Compression Rate of each Benchmarks Suite

(b) Percentage of Uncompressed Instructions for each Benchmarks

Figure 4.1: Performance Result of RVC compiling benchmarks from various benchmark suites.

We further observe the following interesting patterns about the compressed and uncompressed instruction composition of embedded programs:

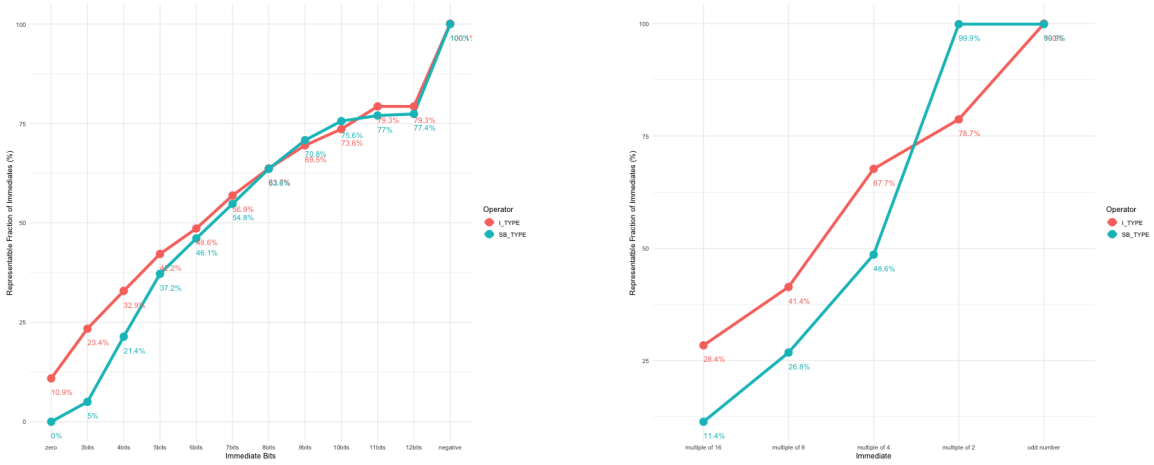
RVC encoding	Static Frequency	Cumulative
C.MV	9.89%	9.89%
C.LI	6.22%	16.11%
C.LWSP	5.95%	22.06%
C.LW	5.84%	27.90%
C.SWSP	5.80%	33.71%
C.J	3.85%	37.56%
C.ADDI	3.55%	41.10%
C.ADD	3.06%	44.16%
C.SW	2.57%	46.73%
C.BEQZ	1.86%	48.59%
C.JR	1.63%	50.22%
C.SLLI	1.45%	51.67%
C.BNEZ	1.44%	53.11%
C.JAL	1.36%	54.47%
C.ADDI16SP	0.81%	55.28%
C.JALR	0.77%	56.04%
C.ADDI4SPN	0.73%	56.78%
C.LUI	0.48%	57.26%
C.OR	0.46%	57.72%
C.ANDI	0.45%	58.17%
C.SRLI	0.43%	58.60%
C.SUB	0.42%	59.02%
C.SRAI	0.23%	59.25%
C.XOR	0.23%	59.48%
C.AND	0.21%	59.68%
C.FLDSP	0.05%	59.74%
C.FLD	0.00%	59.74%

Table 4.1: Most frequent RVC instructions (as expressed in percentage of total instruction count in benchmark program), statically, in a subset of the ZephyrRTOS, FreeRTOS, MiBench, MediaBench benchmarks.

			Benchmark Suites																										
Instruction	epic	jpeg	adpcm	mpeg	9721	basicmath	dijkstra	fft	crc	linpack	qsort	ingsearch	dhrystone	sha	bitcnts	hostscript	rijndael	whetstone	susan	pi_css5	pegwit	freertos	ronization	ilosophers	mpu	ronization	grove	sensor	driver	subsys
addi	6.95%	7.22%	8.87%	7.15%	7.72%	5.45%	7.75%	7.14%	8.39%	4.26%	7.11%	8.30%	8.52%	6.62%	7.22%	8.72%	8.87%	7.20%	15.26%	1.58%	6.60%	8.33%	8.54%	13.03%	9.34%	7.78%	9.87%	9.79%	11.78%	9.93%
jal	4.11%	4.16%	1.06%	4.90%	2.40%	4.54%	1.82%	3.56%	1.76%	10.58%	1.98%	1.18%	2.61%	1.83%	1.38%	2.27%	1.37%	2.84%	12.63%	13.86%	4.55%	4.05%	3.20%	4.36%	1.93%	2.88%	1.89%	1.75%	5.70%	2.88%
lw	1.80%	5.50%	1.31%	5.12%	1.59%	2.32%	2.15%	1.60%	1.15%	8.13%	1.17%	1.52%	1.95%	1.61%	1.16%	1.89%	5.77%	1.35%	4.21%	8.55%	3.77%	0.57%	10.32%	2.56%	3.27%	4.91%	5.25%	3.61%	2.79%	5.38%
sw	1.81%	3.97%	2.39%	2.51%	2.52%	1.57%	3.14%	1.92%	2.40%	4.77%	2.63%	2.60%	2.49%	2.85%	2%	2.69%	1.37%	1.56%	0%	3.91%	1.89%	1.05%	7.47%	1.93%	2.39%	3.55%	3.57%	2.61%	1.78%	3.62%
bne	2.27%	2.11%	2.28%	2.59%	2.37%	2.14%	3.08%	2.24%	2.65%	0.77%	2.18%	2.75%	2.49%	2.49%	2.78%	2.94%	1.71%	2.25%	3.68%	0.13%	2.07%	1.30%	0.36%	1.51%	1.58%	1.02%	0.42%	1.41%	1.01%	1.31%
beq	2.48%	1.72%	1.40%	1.47%	1.69%	2.44%	1.76%	2.25%	1.43%	1.29%	2.56%	1.72%	1.93%	1.55%	2.08%	1.74%	1.05%	2.67%	5.26%	0.10%	1.23%	0.56%	1.42%	1.76%	1.69%	0.85%	0.84%	2.07%	1.78%	2.34%
auipc	1.33%	0.74%	5.22%	1.15%	2.07%	1.56%	3.50%	1.78%	5.20%	0.26%	1.88%	3.69%	2.21%	2.63%	2.21%	2.58%	1.36%	1.96%	1.05%	0.23%	1.08%	5.97%	3.91%	0.42%	0.70%	1.86%	2.31%	1.05%	0.75%	1.76%
lui	1.42%	1.17%	0.31%	1.14%	0.58%	1.48%	0.28%	1.25%	0.13%	2.71%	1.10%	0.25%	1.05%	0.42%	0.64%	0.48%	0.24%	1.16%	7.89%	1.46%	1.01%	0.42%	2.85%	8.47%	5.66%	5.08%	2.31%	4.10%	7.67%	3.78%
lbu	0.89%	1.76%	0.74%	2.18%	0.61%	0.17%	1.52%	0.49%	0.74%	0.13%	0.16%	1.52%	1.31%	0.78%	1.30%	1.70%	0.78%	0.59%	1.05%	0.15%	1.28%	0.39%	0.36%	0.98%	0.88%	0.68%	0.84%	1.07%	5.13%	1.54%
andi	1.44%	0.86%	1.38%	0.79%	1.37%	0.95%	1.42%	1.51%	1.38%	0.13%	0.81%	1.47%	1.73%	1.44%	1.63%	1.19%	5.63%	1.54%	1.05%	0%	2.40%	0.10%	0%	0.56%	0.64%	0.68%	0%	0.78%	2.13%	0.80%
slli	1.77%	0.95%	0.71%	0.91%	1.46%	1.88%	0.42%	1.97%	0.41%	1.81%	1.92%	0.54%	1.10%	0.67%	0.68%	0.58%	0.25%	1.84%	0.53%	0.31%	2.30%	0.23%	0%	0.49%	0.53%	0.51%	1.26%	0.84%	0.42%	0.45%
sub	1.60%	0.85%	0.89%	1.20%	1.38%	2.31%	0.65%	1.54%	0.64%	0.77%	1.92%	0.93%	0.93%	0.78%	0.89%	1.05%	0.44%	1.66%	0%	0.23%	0.43%	0.41%	0.36%	0.42%	0.58%	0.85%	1.26%	1.14%	0.34%	0.74%
srli	1.17%	0.29%	0.04%	0.42%	0.17%	1.65%	0.08%	1.67%	0.05%	0%	1.43%	0.15%	1.05%	0.36%	0.97%	0.78%	4.39%	1.84%	0%	0.01%	3.71%	0.33%	0%	0.04%	0.06%	0.17%	2.73%	1.04%	0.93%	0.22%
sb	0.65%	0.95%	1.47%	0.60%	1.03%	0.26%	1.13%	0.49%	1.05%	0%	0.58%	1.87%	0.94%	1.22%	0.89%	0.98%	0.70%	0.58%	0%	0.12%	0.85%	0.18%	0%	0.84%	0.64%	0.68%	0%	1.17%	3.14%	0.93%
bge	1.31%	1.05%	0.81%	1.36%	1.25%	1.07%	0.69%	1.23%	0.66%	0.65%	1.07%	0.59%	0.91%	0.69%	0.76%	0.73%	0.38%	1.10%	0.53%	0.57%	0.36%	0.15%	0%	0.63%	0.29%	0.17%	0%	0.50%	0.33%	0.35%
blt	1.50%	1.06%	0.71%	1.12%	0.90%	1.01%	0.67%	1.06%	0.56%	1.55%	0.91%	0.54%	0.66%	0.58%	0.62%	0.65%	0.28%	0.98%	2.11%	0.48%	0.36%	0.14%	0%	0.56%	0.64%	0.34%	0%	0.77%	0.32%	0.26%
or	1.23%	0.31%	0.18%	0.54%	0.12%	2.51%	0.32%	2%	0.15%	0%	2.40%	0.69%	0.83%	0.36%	0.80%	0.43%	0.21%	2.29%	0%	0%	2.79%	0.17%	0%	0.04%	0.06%	0%	1.89%	0.95%	0.12%	0%
add	0.63%	0.87%	0.42%	1.73%	0.36%	1.38%	0.34%	0.78%	0.31%	1.42%	0.91%	0.34%	0.34%	0.39%	0.25%	0.30%	0.21%	0.67%	0%	0.22%	1.21%	0.06%	0%	0.14%	0.18%	0.17%	0%	0.28%	0.24%	0.26%
mul	0.69%	0.93%	0.22%	1.73%	0.22%	0.70%	0.18%	0.47%	0.13%	0.77%	0.19%	0.15%	0.65%	0.19%	0.45%	0.76%	0.12%	0.51%	0.53%	0.16%	0.14%	0.29%	0%	0.21%	0.18%	0.17%	2.10%	0.64%	0.22%	0.35%
bgeu	0.52%	0.54%	0.72%	0.42%	0.52%	0.62%	0.63%	0.59%	0.64%	0%	0.45%	0.64%	0.88%	0.72%	0.95%	1.07%	0.36%	0.68%	0%	0%	0.55%	1.13%	0.71%	0.25%	0.23%	0.51%	4.83%	0.93%	0.45%	0.51%
lhu	0.40%	0.42%	0.86%	0.27%	1.05%	0.20%	0.83%	0.35%	1.02%	0%	0.62%	0.44%	0.57%	1.05%	0.83%	0.81%	0.57%	0.42%	0%	0.01%	1.80%	0.05%	0.36%	0.11%	0.18%	0.17%	0.21%	0.21%	0.26%	0.16%
bltu	0.57%	0.40%	0.38%	0.20%	0.33%	0.43%	0.49%	0.41%	0.33%	0%	0.36%	0.44%	0.65%	0.36%	0.70%	1.11%	0.26%	0.54%	1.58%	0%	0.41%	0.66%	0.36%	0.53%	0.76%	0.51%	2.94%	0.98%	0.44%	0.83%
sh	0.37%	0.39%	0.68%	0.64%	1.41%	0.16%	0.69%	0.25%	0.79%	0%	0.52%	0.29%	0.48%	0.86%	0.68%	0.75%	0.41%	0.26%	0%	0.22%	1.09%	0.04%	0%	0.04%	0.06%	0%	0%	0.11%	0.13%	0%
xor	0.14%	0.02%	0.03%	0.03%	0.16%	0.04%	0.02%	0.07%	0.03%	0%	0.03%	0.05%	0.08%	0.17%	0.04%	0.01%	4.64%	0.05%	0%	0.04%	4.51%	0%	0%	0%	0%	0%	0%	0.03%	0.95%	0%
sltu	0.64%	0.15%	0.04%	0.20%	0.01%	1.83%	0.08%	1.02%	0.05%	0%	1.04%	0.15%	0.43%	0.03%	0.23%	0.22%	0.05%	1.13%	0%	0%	0.03%	0.11%	0.36%	0.21%	0.29%	0.51%	0.84%	1.07%	0.26%	0.38%
sll	0.72%	0.28%	0.13%	0.23%	0.25%	0.72%	0.10%	0.91%	0.10%	0%	0.81%	0.20%	0.52%	0.11%	0.31%	0.37%	0.05%	0.92%	0%	0.03%	0.06%	0.24%	0.71%	0.21%	0.35%	0.34%	2.31%	0.95%	0.17%	0.32%
and	0.43%	0.12%	0%	0.09%	0.04%	0.64%	0.14%	0.41%	0%	0%	0.13%	0.25%	0.32%	0.03%	0.25%	0.20%	0.09%	0.39%	0%	0.01%	0.87%	0.08%	1.42%	0.14%	0.23%	0.68%	1.26%	0.58%	0.27%	0.64%
lh	0.16%	0.28%	0.47%	0.42%	1.32%	0.09%	0.36%	0.16%	0.46%	0%	0.39%	0.34%	0.28%	0.50%	0.37%	0.27%	0.24%	0.18%	0%	0.18%	0.15%	0%	0%	0%	0%	0%	0%	0%	0%	0%
srl	0.47%	0.09%	0.03%	0.10%	0.04%	0.60%	0.02%	0.62%	0.03%	0%	0.65%	0%	0.25%	0.03%	0.14%	0.24%	0.01%	0.67%	0%	0%	0.04%	0.22%	0.36%	0.07%	0.12%	0.17%	2.10%	0.70%	0.08%	0.16%
ori	0.28%	0.20%	0.42%	0.14%	0.35%	0.18%	0.51%	0.27%	0.48%	0%	0.32%	0.15%	0.49%	0.53%	0.50%	0.42%	0.25%	0.27%	0%	0%	0.20%	0.07%	0%	0.07%	0%	0%	0%	0.03%	0.03%	0%
srai	0.25%	0.13%	0.19%	0.77%	0.87%	0.22%	0%	0.07%	0%	0%	0.10%	0%	0%	0%	0%	0.01%	0.03%	0.04%	0%	0.12%	0.04%	0%	0%	0.18%	0.29%	0.34%	0%	0.15%	0.11%	0.13%
jalr	0.07%	0.04%	0.19%	0.04%	0.13%	0.03%	0.12%	0.06%	0.13%	0%	0.13%	0.25%	0.09%	0.14%	0.10%	0.09%	0.07%	0.07%	0%	0.03%	0.04%	0%	2.49%	0.25%	0.41%	1.18%	1.47%	0.66%	0.47%	1.12%
divu	0.07%	0.06%	0.03%	0.02%	0.06%	0.04%	0.06%	0.07%	0.03%	0%	0.06%	0.05%	0.15%	0.06%	0.12%	0.43%	0.04%	0.08%	0%	0%	0.03%	0.19%	0%	0.25%	0.18%	0.34%	1.89%	0.41%	0.14%	0.26%
xori	0.13%	0.08%	0.17%	0.08%	0.19%	0.13%	0.12%	0.16%	0.13%	0.13%	0.03%	0%	0.11%	0.11%	0.12%	0.12%	0.05%	0.14%	0%	0%	0.07%	0.11%	0%	0.04%	0.06%	0%	0.21%	0.09%	0.08%	0%
csrr	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.05%	2.49%	0.32%	0.53%	1.18%	1.47%	0.70%	0.49%	1.12%
div	0.16%	0.14%	0.08%	0.18%	0.03%	0.02%	0.04%	0.03%	0.05%	0.13%	0%	0%	0.05%	0.06%	0.04%	0.14%	0.03%	0.03%	0%	0.12%	0.02%	0%	0%	0%	0%	0%	0%	0.03%	0.02%	0%
remu	0.05%	0.06%	0.03%	0.01%	0.01%	0.04%	0.06%	0.07%	0.03%	0%	0%	0.05%	0.14%	0.03%	0.12%	0.30%	0.01%	0.08%	0%	0%	0.01%	0.17%	0%	0.04%	0.06%	0.17%	1.68%	0.26%	0.05%	0.10%
csrci	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	1.13%	0%	0%	0%	0%	0%	0%	0%	0%
csrrc	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.36%	0.70%	0.76%	0.17%	0.21%	0.43%	0.21%	0.22%
sra	0.03%	0.09%	0%	0.05%	0.23%	0.07%	0%	0.05%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.04%	0%	0%	0%	0%	0%	0%	0%	0%	0%
sltiu	0.02%	0.02%	0.03%	0.07%	0.03%	0.19%	0.02%	0.03%	0.03%	0%	0.03%	0%	0.05%	0.03%	0.04%	0.09%	0.04%	0.03%	0%	0%	0.04%	0.04%	0%	0.04%	0.06%	0%	0%	0.03%	0.01%	0%
csrrs	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.36%	0.63%	0.76%	0.17%	0.21%	0.38%	0.19%	0.22%
csrw	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.07%	1.07%	0.11%	0.18%	0.51%	0.63%	0.28%	0.20%	0.48%
lb	0%	0%	0.03%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.04%	0%	0%	0.77%	0.35%	0%	0%	0.14%	0.11%	0%
rem	0.04%	0.01%	0%	0.02%	0%	0%	0.02%	0.02%	0%	0.13%	0%	0%	0%	0%	0%	0.17%	0%	0%	0%	0.03%	0%	0%	0%	0%	0%	0%	0%	0.02%	0.02%	0%
slt	0.01%	0.01%	0%	0.05%	0.04%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.01%	0%	0%	0%	0.04%	0.06%	0%	0%	0.02%	0.01%	0%
csrwi	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.07%	0.12%	0%	0%	0.04%	0.02%	0%
mulhu	0%	0%	0.01%	0%	0%	0%	0%	0.01%	0%	0%	0%	0%	0%	0%	0.02%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.03%	0.03%	0.03%
slti	0%	0.01%	0%	0%	0.03%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.01%	0%	0%	0%	0.01%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
csrs	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.04%	0%	0%	0%	0%	0%	0%	0%	0%
csrsi	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0.01%	0%	0%	0%	0%	0%	0%	0%	0%

Figure 4.2: Uncompressed RISC-V Instruction, statically, seen in the set of benchmark suite.

- A small number of opcodes account for most instructions in an embedded program. Table 4.1 and Figure 4.2 show the static frequencies of most commonly occurring compressed and base RISC-V instructions seen across the combined benchmark suite. Statically, 10 out of 17 RVC opcodes account for over 80% of compressed instructions and 10 RISC-V opcodes account for 70% of uncompressed instructions. ADDI alone accounts for one-third of total program instructions and, on average, 20% of uncompressed instructions seen across compiled benchmark programs. The benchmarks do not have floating-point intensive programs: The use of floating-point arithmetic/load/store instructions is very light (¡ 0.1%).



(a) Size of Immediate Operands seen among uncompressed I-type and SB-type operations

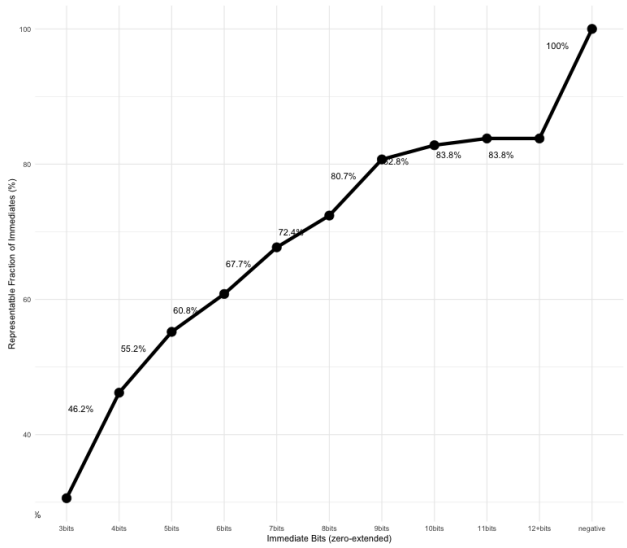
(b) Immediate Operands seen among uncompressed I-type and SB-type operations

Figure 4.3: Cumulative distribution of immediate operand (size and whether in multiple of half-words, words, etc) in a subset of the combined benchmark suite.

- A notable portion of immediate operands, seen among uncompressed instructions, are positive and small enough to fit in a shorter 16-bit encoding. Since current RVC already takes advantages of most small immediate operands, many of those instructions are not compressed by RVC due to their register references to registers other than the eight rvc-registers. Figure 4.3 shows the size of immediate operands seen in the uncompressed I-TYPE and SB-type instructions across combined benchmark suite. Branch displacement are usually larger but shorter branches are quite common.
- Most of immediate operands seen in uncompressed operations are multiples of 4. Many are additionally multiple of 8 or 16. Figure 4.3 shows the size of immediate operands seen in the uncompressed I-Type and SB-type instructions across combined benchmark suite.
- Data-transfer instructions account for one-third of uncompressed instructions in compiled programs across combined benchmark suite. Among the uncompressed data-transfer operations, three instructions (lw, sw, lbu) account for over 78% (Figure 4.4a). Similarly, immediate operands of these uncompressed loads and stores are usually small (Figure 4.4b).

instruction	staticFrequency
lw	36.70%
sw	27.40%
lbu	14.00%
sb	9.20%
lhu	5.10%
sh	4.60%
lh	2.70%
lb	0.30%

(a) Operators



(b) Immediate Operands

Figure 4.4: Distribution of instruction operators and immediate operand sizes in a subset of the combined benchmark suite.

From observations of static frequencies of compressed and uncompressed instructions, we notice that our embedded benchmarks are not floating-point intensive: floating-point arithmetic operations are rarely seen and floating-point loads and stores only account for a trivial percentage of total program code. Therefore, we could potentially reuse those RVC instruction encoding formats that are currently reserved for floating-point related operations to cover and compress other frequently occurring uncompressed instruction formats.

In the following sections we take a closer look at most frequent uncompressed RISC-V instructions: add-immediate (ADDI), jump control transfers (J/JAL), branch control transfers and data-transfer instructions. Since current proposal of RVC already includes shorter encoding of such instructions, we focus on the patterns of such instructions that are not compressed by RVC.

4.2 Add-Immediate Encoding

As displayed in Figure 4.2, uncompressed add-Immediate instructions account for over 7% among total program instructions. In the base ISA, ADDI instructions takes a 5-bit registers (rs1), add to it a 12-bit immediate and write the result to another 5-bit register (rd). Current RVC Extension (Figure 4.5) provides several 16-bit encoding for ADDI operations where signed immediate operand is small enough to fit in 6 bits, or the operation specifies the same source and destination register or can only reference the eight rvc-registers: two return-value registers and four argument registers, and two callee-saved registers, in addition to x0 and the stack pointer:

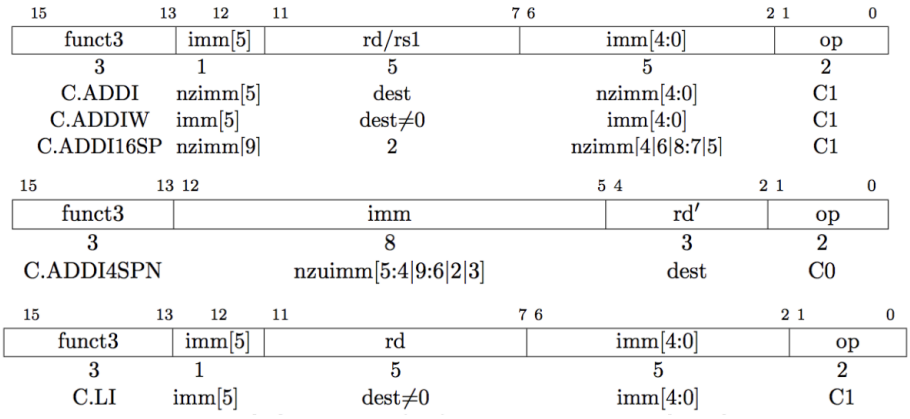


Figure 4.5: RVC instruction encodings for ADDI. rd specifies a destination register, while rs1 specifies source registers. The funct fields are additional opcodes.

- C.ADDI adds the non-zero sign-extended 6-bit immediate to the value in register rd then writes the result to rd. C.ADDI expands into addi rd, rd, nzimm[5:0].
- C.ADDI16SP adds the non-zero sign-extended 6-bit immediate to the value in the stack pointer (sp=x2), where the immediate is scaled to represent multiples of 16 in the range (-512,496). It expands into addi x2, x2, nzimm[9:4].
- C.ADDI4SPN adds a zero-extended non-zero immediate, scaled by 4, to the stack pointer, x2, and writes the result to rd'. It expands to addi rd', x2, nzuimm[9:2].
- C.LI loads the sign-extended 6-bit immediate, imm, into register rd. C.LI is only valid when rd is not x0. C.LI expands into addi rd, x0, imm[5:0].

Register References	Static Frequency Among Uncompressed ADDI	Static Frequency Among Overall Program
Partially Satisfies C.ADDI	41.66%	2.98%
Partially Satisfies C.LI	26.04%	1.86%
Partially Satisfies C.ADDI16SP	0.41%	0.03%
Partially Satisfies C.ADDI4SPN	0.30%	0.02%
Other	31.60%	2.26%

Table 4.2: Uncompressed ADDI operations based on whether its register references satisfy one of the RVC encoding.

Current RVC encoding for ADDI already covers most operations where immediate operand is small, restricting the potential registers an operation can reference. Table 4.2 shows that two-thirds of ADDI instructions are not compressed because their immediate offsets are too large to fit in an RVC encoding. About one-third of ADDI operations are not handled by RVC because their register usage does not satisfy the constraints of any RVC encoding.

Register rd	Register rs1	Static Freq. (32-bit ADDI)	Static Freq. (Overall)
rd $\in$ RVC Registers	rs1 $\in$ RVC Registers	52.52%	3.30%
rd $\in$ RVC Registers	rs1 == x0	22.83%	1.43%
rd $\in$ RVC Registers	rs1 $\notin$ x0, RVC Registers	9.80%	0.62%
rd $\notin$ RVC Registers	rs1 $\in$ RVC Registers	3.90%	0.25%
rd $\notin$ RVC Registers	rs1 == x0	3.22%	0.20%
rd $\notin$ RVC Registers	rs1 $\notin$ x0, RVC Registers	7.72%	0.49%

Table 4.3: Frequency of register usage of uncompressed ADDI operations. Approximately 50% of uncompressed ADDI instructions reference only rvc-registers.

In Table 4.3 we present a further analysis of the usage of rd and rs1 registers across uncompressed ADDI instructions. Majority of rd and rs1 references are to the eight rvc-registers. One-quarter of rs1 registers reference x0.

We then take a closer look at two groups of uncompressed ADDI operations based on their register usage:

- (A) rd register references one of the eight rvc-registers. rs1 references x0,
- (B) rd == rs1, and references one of the eight rvc-registers.
- (C) rd != rs2 and both reference the eight rvc-registers.

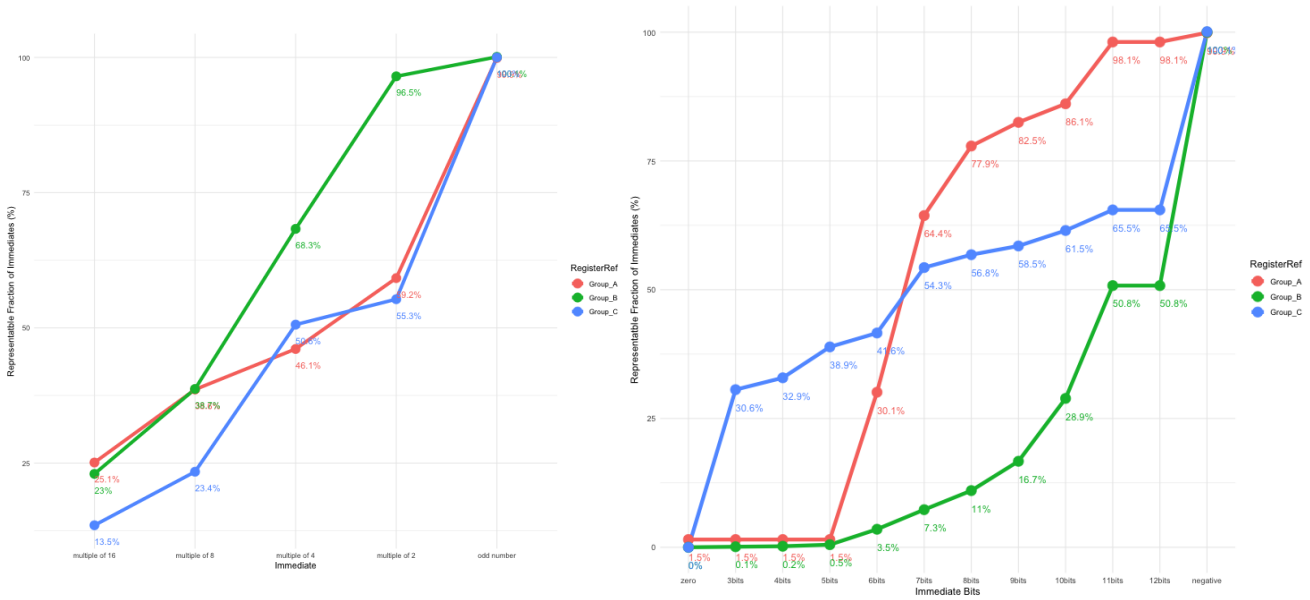


Figure 4.6: Distribution of immediate operands seen across uncompressed Group A/B/C ADDI instructions

Figure 4.6 shows that majority of Group A has odd immediate, and about 60% of those immediate operands can be represented using 7 bits. Group B has an extensive use of immediate operands that are in multiple of 2 but those immediate typically requires more than 10 bits to represent. Half of Group C has an even immediate operand and over 40% of them is small enough to be represented in 6 bits. Furthermore, we notice that about half of Group B operations has an extensive reference to register a5 as both rd and rs1 registers. (Figure 4.7).

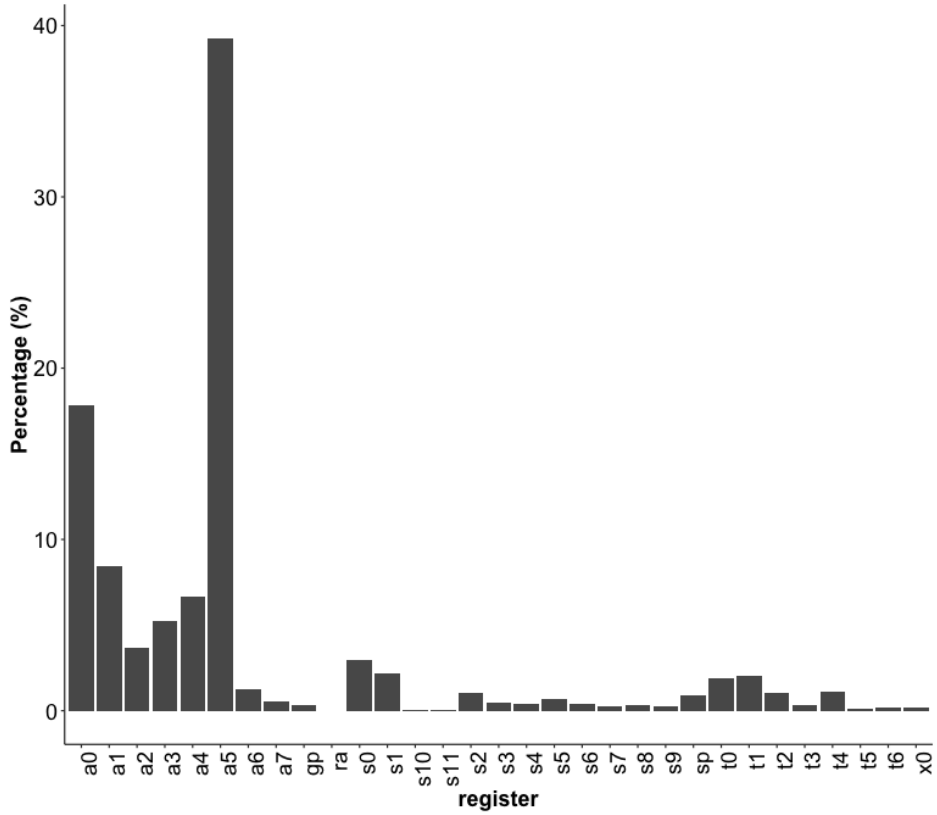


Figure 4.7: Register Usage Among ADDI operations where rs1 == rd

Hence, we propose to add the following 16-bit encoding to RVC in order to further compress ADDI operations:

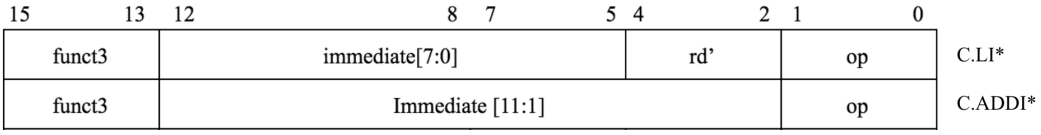


Figure 4.8: Proposed 16-bit instruction encoding for ADDI operations.

- C.LI* loads the zero-extended 8-bit immediate, imm, into register rd'. C.LI* is only valid when $rd \neq x0$ and $rd \in$ rvc-registers. C.LI* expands into addi rd', x0, imm[7:0].
- C.ADDI* adds the non-zero sign-extended 11-bit immediate, scaled by 2, to the value in register a5 then writes the result to a5. C.ADDI* expands into addi a5, a5, nzimm[10:0].

4.3 Control Transfer Instruction Encoding

15 13	12 2	1 0
funct3	imm	op
3	11	2
C.J	offset[11|4|9:8|10|6|7|3:1|5]	C1
C.JAL	offset[11|4|9:8|10|6|7|3:1|5]	C1

Figure 4.9: RVC instruction encoding for unconditional jump operations.

Unconditional jump is the second most frequent operations among uncompressed instructions, accounting for 3.85% of total program instructions on average (Table 4.2). In the base ISA, unconditional jumps take in one destination register and a 20-bit immediate used to calculate target jump address. RVC provides two 16-bit encoding for jumps that reference registers $x0$ or ra as destination register and an immediate operand that is small enough to be represented using 11-bits (Figure 4.9):

- C.J expands to jal x0, offset[11:1]. Offset is scaled by 2 and then added to the pc to form the jump target address.
- C.JAL, similarly, expands to jal x1, offset[11:1].

We made the following observations about jump instructions across compiled programs:

- Unconditional jump instructions reference possibly only three registers: ra, x0, t0. Register t0 only accounts for a small percentage of total register usage among jump operations.

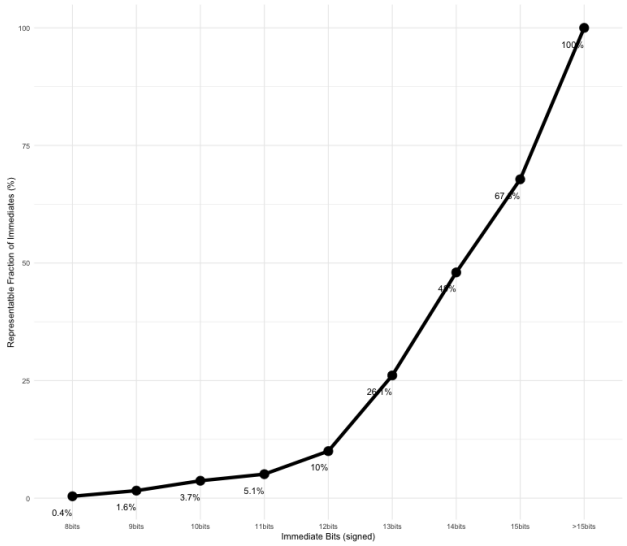


Figure 4.10: Immediate Operands Among Uncompressed J/JAL operations.

- Immediate operands across uncompressed jump instructions typically requires 15 or more bits to represent. (Figure 4.10). Current RVC extension Jump instructions where immediate operand can be represented in 11-bits are mostly handled by RVC already, except those that reference t0.
- Uncompressed unconditional jumps mostly jump to another function label. Those target addresses are stored in linker symbol table and resolved by linker during compilation.

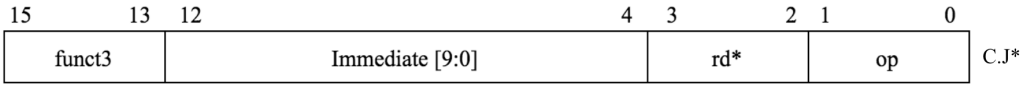


Figure 4.11: Proposed 16-bit instruction encoding for Jump operations.

Hence, we propose to have unconditional jump instructions to reference the linker symbol table to calculate target jump address. A Jump operation can then take an index (or pointer) to index into address table and retrieve addresses (Figure 4.11):

- C.J* performs an unconditional control transfer. The index is used to retrieve target address from linker symbol table. C.J* expands to jal rd*, symbols[offset[9:0]].

4.4 Branch Control Transfer Instruction Encoding

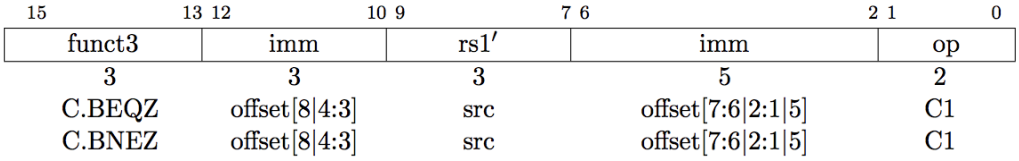


Figure 4.12: RVC instruction encoding for branch operations.

Branch (SB-type) instructions account for 17.32% among uncompressed program instructions and 6.85% of total program instructions. In base ISA, a branch operator compares values from two registers rs1 and rs2 and jump to target branch address, computed by adding the 12-bit immediate to pc, if the branch condition holds. RVC extension provides encoding branch operators where the condition is an equality check with register x0 (Fig 4.12).

- C.BEQZ takes the branch if the value in register rs1' is zero. It expands to beq rs', x0, offset[8:1].
- C.BNEZ expands to bne rs1', x0, offset[8:1].

Instruction	Register rs1	Register rs2	Static Freq. (Among Uncompressed ADDI)
beq	rs1 $\in$ RVC Register	rs2 $\in \{$x0, a5$\}$	8.74%
		rs2 $\notin \{$x0, a5$\}$	8.52%
	rs1 $\notin$ RVC Register	rs2 $\in \{$x0, a5$\}$	7.22%
		rs2 $\notin \{$x0, a5$\}$	1.98%
bne	rs1 $\in$ RVC Register	rs2 $\in \{$x0, a5$\}$	12.97%
		rs2 $\notin \{$x0, a5$\}$	10.18%
	rs1 $\notin$ RVC Register	rs2 $\in \{$x0, a5$\}$	5.19%
		rs2 $\notin \{$x0, a5$\}$	2.99%
bge	rs1 $\in$ RVC Register	rs2 $\in \{$x0, a5$\}$	5.16%
		rs2 $\notin \{$x0, a5$\}$	3.85%
	rs1 $\notin$ RVC Register	rs2 $\in \{$x0, a5$\}$	1.55%
		rs2 $\notin \{$x0, a5$\}$	2.78%
bgeu	rs1 $\in$ RVC Register	rs2 $\in \{$x0, a5$\}$	1.61%
		rs2 $\notin \{$x0, a5$\}$	4.81%
	rs1 $\notin$ RVC Register	rs2 $\in \{$x0, a5$\}$	0.67%
		rs2 $\notin \{$x0, a5$\}$	1.98%
blt	rs1 $\in$ RVC Register	rs2 $\in \{$x0, a5$\}$	3.06%
		rs2 $\notin \{$x0, a5$\}$	4.80%
	rs1 $\notin$ RVC Register	rs2 $\in \{$x0, a5$\}$	1.74%
		rs2 $\notin \{$x0, a5$\}$	3.14%
bltu	rs1 $\in$ RVC Register	rs2 $\in \{$x0, a5$\}$	0.41%
		rs2 $\notin \{$x0, a5$\}$	4.40%
	rs1 $\notin$ RVC Register	rs2 $\in \{$x0, a5$\}$	0.12%
		rs2 $\notin \{$x0, a5$\}$	1.53%

Table 4.4: Operator Distribution and Register Usage seen among Uncompressed branch control transfer operations.

We make the following observation about uncompressed branch control transfers:

- The top frequent operators among uncompressed branch control transfer instructions are beq, bne, bge and blt.
- RS1 register reference are usually to the eight rvc-registers.
- RS2 register reference are usually to x0 or a5.

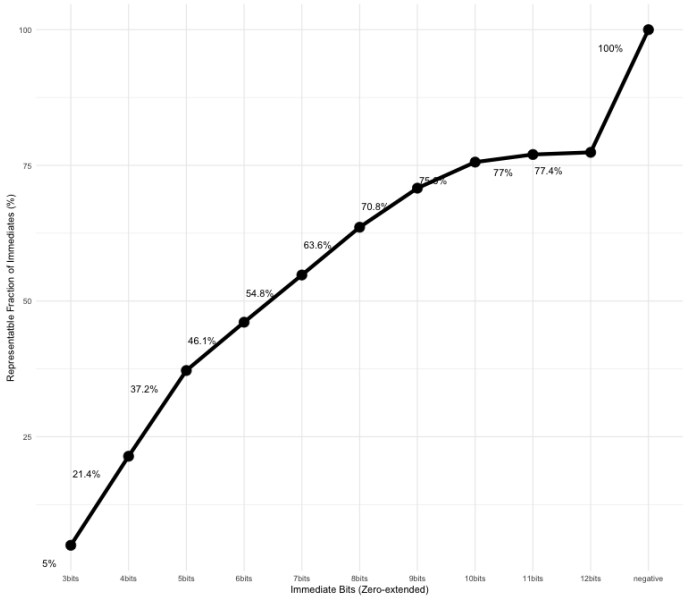


Figure 4.13: Immediate Operands seen among Uncompressed branch control transfer operations.

- A notable percentage of branch offsets are positive and representable using only 4 or 5 bits. Due to the usage of branch instruction. branch offsets are always in multiple of 2 and about 75% are positive. Hence over 30% of offsets are positive and representable using 4 bits if treated as unsigned numbers, dropping the least significat bit. Almost 50% of them are positive and representable in 5 bits.

Hence, we propose to add the following 16-bit encodings to RVC to further compress branch instructions:

15 13	12 9	8	7 6	5 3	2 1	0	
funct3	immediate[4:1]	beq/bne	rs1		rd'	op	C.BEQBNE
funct3	immediate[5:1]		sb_funct	rs1	rd'	op	C.BRANCH

Figure 4.14: Proposed 16-bit instruction encoding for branch operations.

- C.BEQBNE performs conditional control transfers bne or beq depending on a select bit (bit 8). The offset is zero-extended, scaled by 2, and then added to the pc to form the branch target address. It requires rs2 reference to only x0 or a5. It expands to beq/bne rs1, rs2*, offset[4:1].
- C.BRANCH performs conditional control transfers specified by sb_funct (bne, beq, blt or bge). The offset is zero-extended, scaled by 2, and then added to the pc to form the branch target address. It is only valid when $rs1 \in$ rvc-registers and $rs2 \in$ x0, a5. It expands to branchOP rs1', rs2*, offset[5:1].

4.5 Load and Store Instruction Encoding

Data-transfer instructions account for 23.74% of uncompressed instructions and 9.55% of total program instructions (Figure 4.2). Currently RVC includes several word and double-word loads and stores where register references are to rvc-registers or the stack pointer and immediate operands are small enough ([0, 255]) and in multiple of 4. Floating-point loads and stores are also included in RVC as they are dynamically common in nearly all floating-point programs.

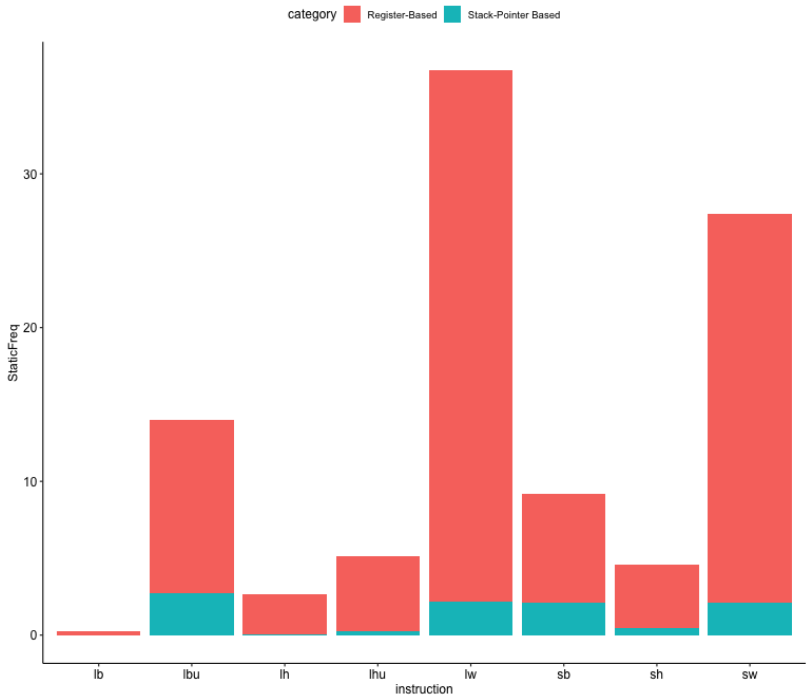
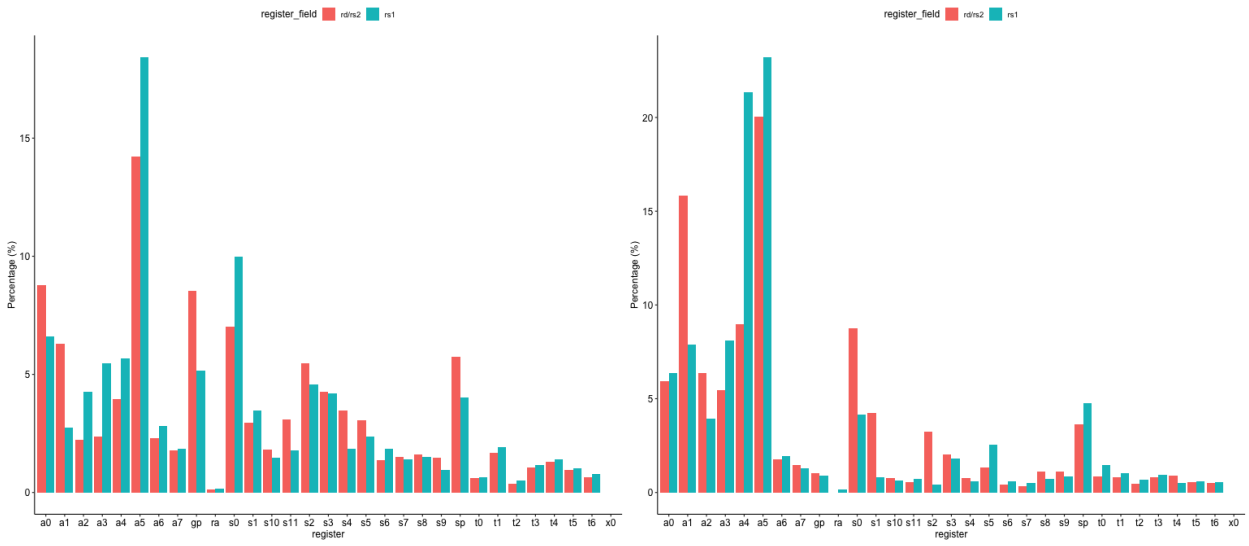


Figure 4.15: Percentage of operator usage among uncompressed data-transfer operations. Majority of data transfer instructions are register-based.

We make the following observations about uncompressed load and store instructions:

- Most uncompressed data-transfer instructions are word loads and stores (37% and 27%, respectively). Byte loads and stores (14% LBU and 9% SB) also account for a notable portion. (Figure 4.15)
- Majority of uncompressed data-transfer operations are register-based ($rs1$!= sp). Majority of stack-based data-transfer instructions ($rs1$ == sp) are handled by current RVC encodings already. (Figure 4.15)
- Byte-size data-transfer instructions exhibit substantial locality of register reference. Figure 4.16 shows the static frequency of register usage across uncompressed byte-size loads and stores distributed mainly among rvc-registers. None of the remaining references are to the



(a) (b)

Figure 4.16: Registers Among Uncompressed (a) Word-size and (b) Byte-size Load and Store operations.

stack pointer or to the zero register. In contrast, uncompressed word-size data-tranfer operations reference both rvc-registers and other registers.

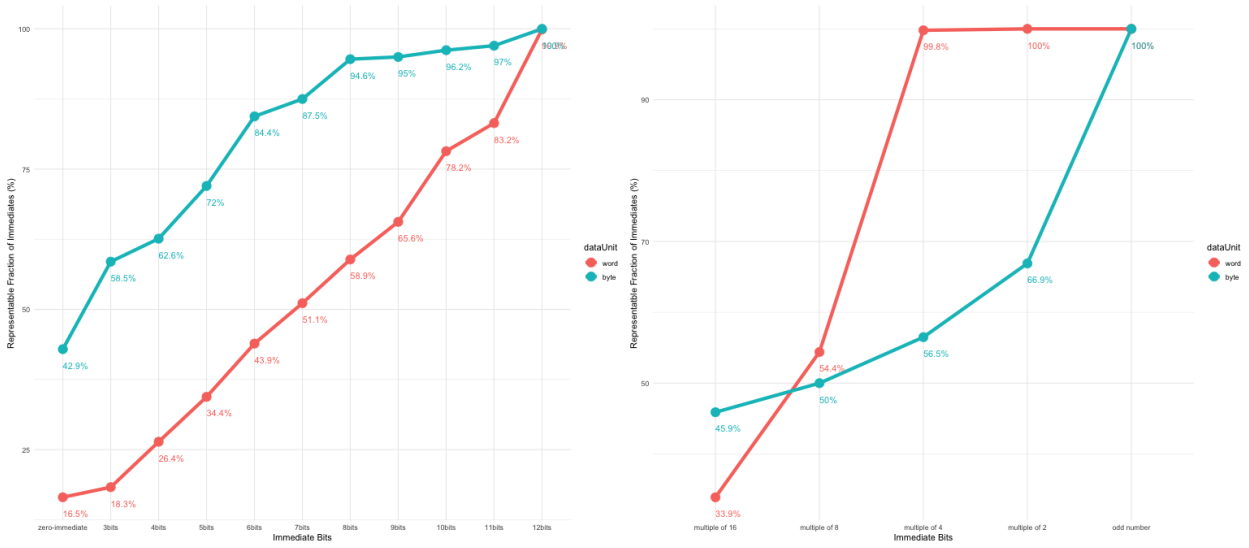


Figure 4.17: Immediate Operands Among Uncompressed Register-based word/byte loads and stores.

- Immediate operand of byte-size data transfer instructions are mostly small. Over 50% of immediate operands are zero and about 70% can be represented using 3 bits. Those of word-size data-transfer instructions are larger and over 50% require 7 bits. However, many of those immediate operands are mostly multiple of 8.

Hence, we propose to add the following 16-bit encodings to RVC to further compress branch instructions:

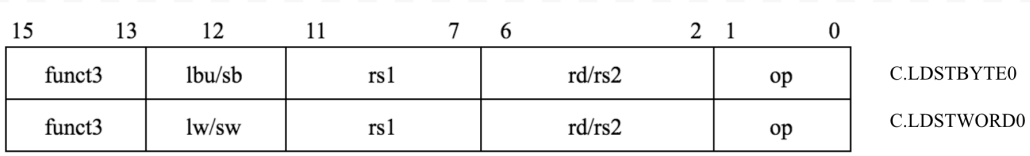


Figure 4.18: Proposed 16-bit instruction encodings for LD/ST operations.

- C.LDSTBYTE0 loads a 8-bit (unsigned) value from memory into register rd or stores a 32-bit value in register rs2 to memory, depending on the select bit. The effective memory address is retrieved by reading the base address in register rs1. It expands to lbu rd, 0(rs1) or sb rs2, 0(rs1).
- C.LDSTWORD0 loads a 32-bit (unsigned) value from memory into register rd or stores a 32-bit value in register rs2 to memory, depending on the select bit. The effective memory address is retrieved by reading the base address in register rs1. It expands to lw rd, 0(rs1) or sw rs2, 0(rs1).

4.6 Implementation Consideration

As mentioned in section 4.1, the proposed instruction encoding will replace those existing ones that are rarely used to compress static program code. Based on the statistics seen in Table 4.1, floating-point loads and stores (C.FLD, C.FLDSP, C.FLW, C.FLWSP, C.FSW, C.FSWSP, C.FSD, C.FSDSP) are seldom seen and individual instruction format accounts for a trivial percentage of instructions in benchmark programs. Hence, we propose to redesign the RVC ISA, remove the floating-point data transfer instructions and allocate those opcode and funct3 bits for the 7 proposed instruction formats (C.LI*, CLADDI*, C,J*, C.BEQBNE, C.BRANCH, C.LDSTBYTE0, C.LDSTWORD0).

Chapter 5

Evaluation

In this section, we present our evaluation results. The benchmarks are collected from ZephyrRTOS, Apache Mynewt, MiBench and MediaBench benchmark suites. We compiled the benchmarks using GNU MCU Eclipse RISC-V Embbed GCC, 32-bit 8.1.0 compiler with *ISA = rv32imac* and *ABI = ilp32*, optimizing for size (*-O3 -Os*), register save/restore code (*-msave-restore*) and linker-time optimization (-flto). Proposed compressed instruction set is evaluated as a pass at the last step. The procedure is straight forward: instructions that satisfy the proposed instruction formats are compressed using corresponding 16-bit formats, whereas floating-point loads and stores are ”decompressed” into 32-bit instructions because modified RVC no longer has a 16-bit encoding for them any more.

Figure 5.1 compares the instruction compression rate before and after adding proposed changes to RVC. On average the instruction compression rate is reduced by 10%. Figure 5.2 shows that compression rate is improved by 4.97%.

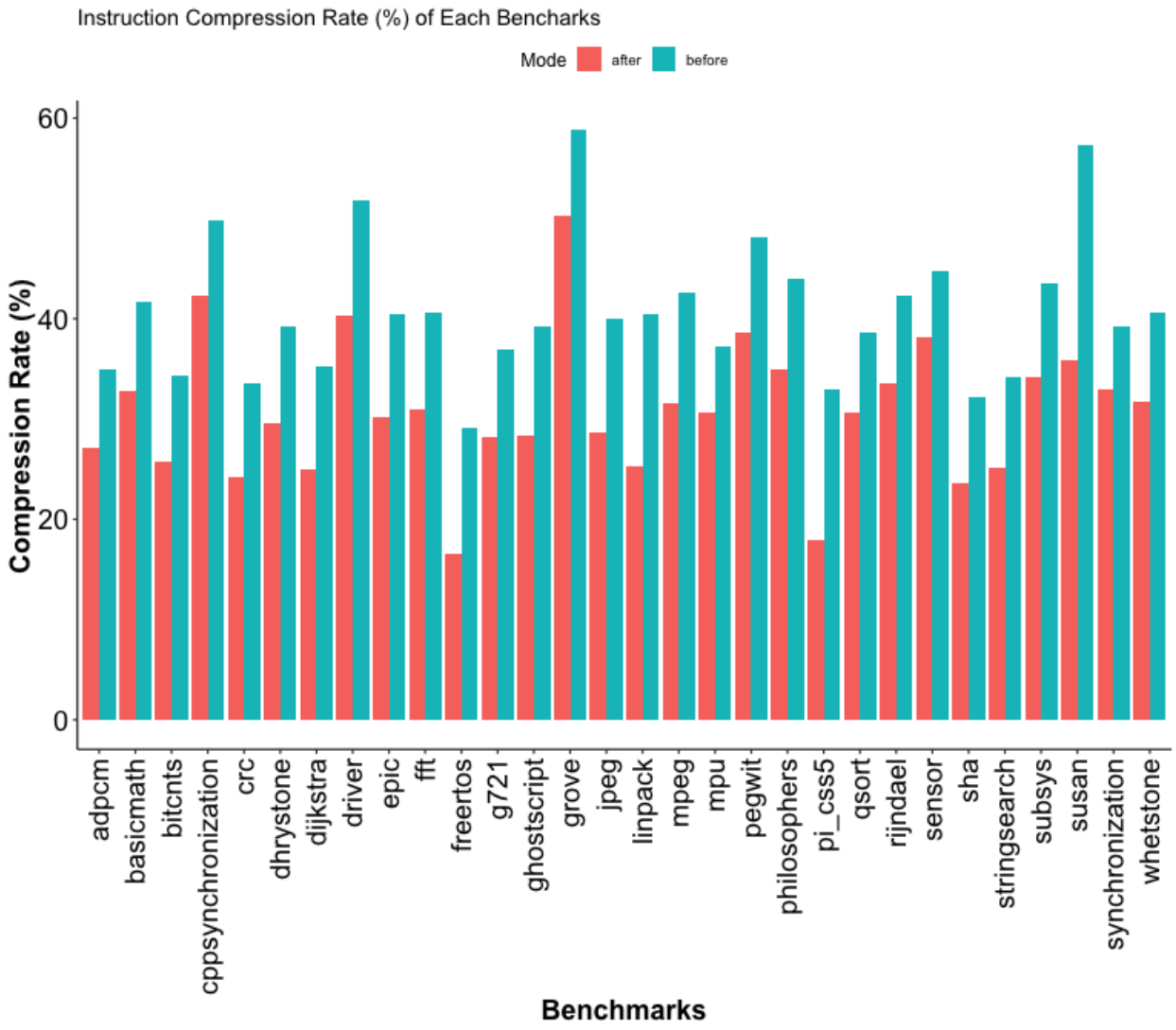


Figure 5.1: Instruction Compression Rate With and Without proposed changes to RVC.

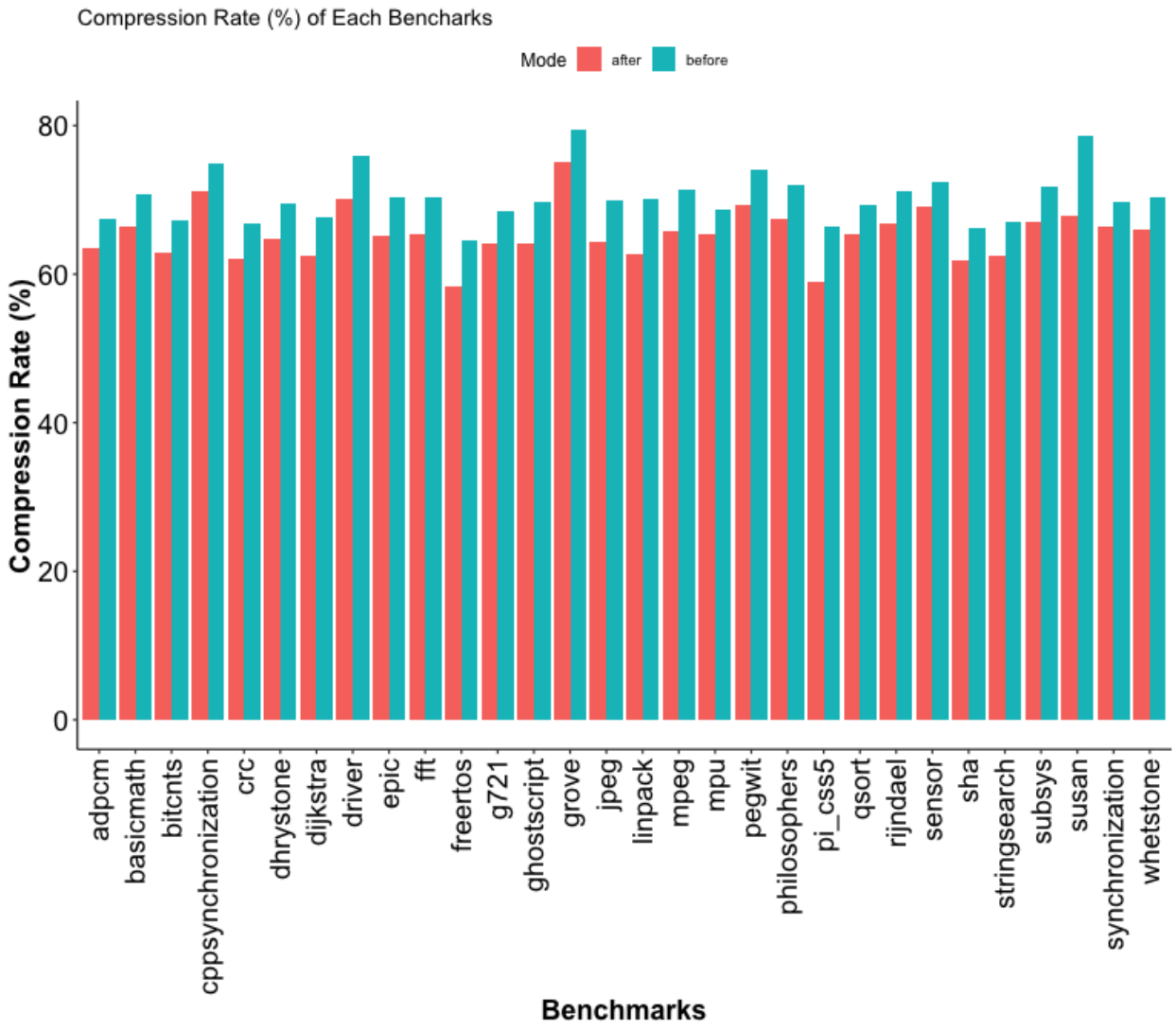


Figure 5.2: Compression Rate With and Without proposed changes to RVC.

Chapter 6

Conclusion

In this thesis report, we took a closed look at embedded benchmarks from various sources including ZephyrRTOS, MediaBench, MiBench and FreeRTOS. From obtained statistics those benchmarks are not floating-point intensive: floating-point arithmetic and data-transfer operations only account for a very small portion of total program size. Hence, for the purpose of reducing static code size for embedded systems and IoT applications, we could potentially drop those instruction format encodings reserved for floating-point operations, and instead reuse them for other commonly occurring but uncompressed instructions. Results show that the proposed encoding, for programs that are not floating-point intensive, covers an additional 10% of program instructions and as a result, improve the compression ratio by 5%. However, for floating-point intensive programs, our proposed modifications may or may not achieve comparable improvement.

Additionally, implementation of the proposed modifications to the RISC-V Compressed ISA are relatively straightforward. The change of instruction format encoding does not complicate existing compressor and de-compressor, nor does it affects the compilation and execution time of program applications.

Bibliography

[1] Guido Araujo et al. “Code Compression Based on Operand Factorization”. In: *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture* (1998), pp. 194–201.

[2] Talal Bonny and Jorg Henkel. “Huffman-based Code Compression Techniques for Embedded Processors”. In: *ACM Trans. Des. Autom. Electron. Syst.* (2010), 31:1–31:37.

[3] Robert Britton. *MIPS Assembly Language Programming*. Pearson Education, 2003. ISBN: 0131420445.

[4] Mikael Collin and Mats Brorsson. “Two-Level Dictionary Code Compression: A New Scheme to Improve Instruction Code Density of Embedded Applications”. In: *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2009), pp. 231–242.

[5] IBM (International Business Machines (IBM) Corporation. “CodePack PowerPC Code Compression Utility User’s Manual Version 3.0”. In: *IBM Technical Report* (1998).

[6] M. R. Guthaus et al. “MiBench: A free, commercially representative embedded benchmark suite”. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization* (2001).

[7] A. Halambi et al. “An Efficient Compiler Technique for Code Size Reduction Using Reduced Bit-Width ISAs”. In: *Proceedings of the Conference on Design, Automation and Test in Europe* (2002).

[8] Kissell K. “MIPS16: High-density MIPS for the Embedded Market”. In: *Silicon Graphics MIPS Group* (1997).

[9] Michael Kozuch and Andrew Wolfe. “Compression of Embedded System Programs”. In: *Proceedings of the1994 IEEE International Conference on Computer Design: VLSI in Computer Amp; Processors* (1994), pp. 270–277.

[10] Jeremy Lau et al. “Reducing Code Size with Echo Instructions”. In: *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (2003), pp. 84–94.

[11] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems”. In: *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (1997), pp. 330–335.

[12] Charles Lefurgy, Eva Piccininni, and Trevor Mudge. “Evaluation of a High Performance Code Compression Method”. In: *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture* (1999), pp. 93–102.

[13] Charles Lefurgy et al. “Improving Code Density Using Compression Techniques”. In: *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (1997), pp. 194–203.

[14] Haris Lekatsas and Wayne Wolf. “Code Compression for Embedded Systems”. In: *Proceedings of the 35th Annual Design Automation Conference* (1998), pp. 516–521.

[15] Haris Lekatsas and Wayne Wolf. “SAMC: A code compression algorithm for embedded processors”. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 18 (Jan. 2000), pp. 1689–1701.

[16] Stan Liao, Srinivas Devadas, and Kurt Keutzer. “A Text-compression-based Method for Code Size Minimization in Embedded Systems”. In: *ACM Trans. Des. Autom. Electron. Syst.* (1999), pp. 12–38.

[17] E. Wanderley Netto et al. “Multi-profile Based Code Compression”. In: 2004, pp. 244–249.

[18] J. Prakash et al. “A simple and fast scheme for code compression for VLIW processors”. In: *Data Compression Conference, 2003. Proceedings. DCC 2003* (2003), pp. 444–.

[19] Montserrat Ros and Peter Sutton. “A Hamming Distance Based VLIW/EPIC Code Compression Technique”. In: *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (2004), pp. 132–139.

[20] Montserrat Ros and Peter Sutton. “A Post-compilation Register Reassignment Technique for Improving Hamming Distance Code Compression”. In: *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (2005), pp. 97–104.

[21] S. Seong and P. Mishra. “Bitmask-Based Code Compression for Embedded Systems”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2008), pp. 673–685.

[22] KRISHNAN SUNDARESAN and NIHAR R. MAHAPATRA. “Code Compression Techniques for Embedded Systems and Their Effectiveness”. In: *Proceedings of the IEEE Computer Society Annual Symposium on VLSI* (2003).

[23] Andrew Waterman. “Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed”. In: (May 2011).

[24] Andrew Waterman et al. “The RISC-V Instruction Set Manual”. In: (2014).

[25] Andrew Wolfe and Alex Chanin. “Executing Compressed Programs on an Embedded RISC Architecture”. In: *SIGMICRO Newsl.* 23.1-2 (1992), pp. 81–91.

[26] X. H. Xu, S. R. Jones, and C. T. Clarke. “ARM/THUMB code compression for embedded systems”. In: *Proceedings of the 12th IEEE International Conference on Fuzzy Systems* (2003), pp. 32–35.