The RISC-V Compressed Instruction Set Manual

Version 1.9

Warning! This draft specification may change before being accepted as standard, so implementations made to this draft specification might not conform to the future standard.

Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanović
CS Division, EECS Department, University of California, Berkeley
{waterman|yunsup|pattrsn|krste}@eecs.berkeley.edu

November 5, 2015

This document is also available as Technical Report UCB/EECS-2015-209.

1.1 Introduction

This excerpt from the RISC-V User-Level ISA Specification describes the current draft proposal for the RISC-V standard compressed instruction set extension, named "C", which reduces static and dynamic code size by adding short 16-bit instruction encodings for common operations. The C extension can be added to any of the base ISAs (RV32, RV64, RV128), and we use the generic term "RVC" to cover any of these. Typically, 50%–60% of the RISC-V instructions in a program can be replaced with RVC instructions, resulting in a 25%–30% code-size reduction.

We believe this draft represents the close to final design for RV32C and RV64C (it seems premature to freeze R128C), though we are requesting one more round of comments, hence the 1.9 revision number. Please send your comments to the `isa-dev` mailing list at `isa-dev@lists.riscv.org`.

1.2 Overview

RVC uses a simple compression scheme that offers shorter 16-bit versions of common 32-bit RISC-V instructions when:

- the immediate or address offset is small, or
- one of the registers is the zero register (`x0`), the ABI link register (`x1`), or the ABI stack pointer (`x2`), or
- the destination register and the first source register are identical, or
- the registers used are the 8 most popular ones.

The C extension is compatible with all other standard instruction extensions. The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions, with the latter now able to start on any 16-bit boundary.

Removing the 32-bit alignment constraint on the original 32-bit instructions allows significantly greater code density.

The compressed instruction encodings are mostly common across RV32C, RV64C, and RV128C, but as shown in Table 1.3, a few opcodes are used for different purposes depending on base ISA width. For example, the wider address-space RV64C and RV128C variants require additional opcodes to compress loads and stores of 64-bit integer values, while RV32C uses the same opcodes to compress loads and stores of single-precision floating-point values. Similarly, RV128C requires additional opcodes to capture loads and stores of 128-bit integer values, while these same opcodes are used for loads and stores of double-precision floating-point values in RV32C and RV64C. If the C extension is implemented, the appropriate compressed floating-point load and store instructions must be provided whenever the relevant standard floating-point extension (F and/or D) is also implemented. In addition, RV32C includes a compressed jump and link instruction to compress short-range subroutine calls, where the same opcode is used to compress ADDIW for RV64C and RV128C.

Double-precision loads and stores are a significant fraction of static and dynamic instructions, hence the motivation to include them in the RV32C and RV64C encoding.

Although single-precision loads and stores are not a significant source of static or dynamic compression for benchmarks compiled for the currently supported ABIs, for microcontrollers that only provide hardware single-precision floating-point units and have an ABI that only supports single-precision floating-point numbers, the single-precision loads and stores will be used at least as frequently as double-precision loads and stores in the measured benchmarks. Hence, the motivation to provide compressed support for these in RV32C.

Short-range subroutine calls are more likely in small binaries for microcontrollers, hence the motivation to include these in RV32C.

Although reusing opcodes for different purposes for different base register widths adds some complexity to documentation, the impact on implementation complexity is small even for designs that support multiple base ISA register widths. The compressed floating-point load and store variants use the same instruction format with the same register specifiers as the wider integer loads and stores.

RVC was designed under the constraint that each RVC instruction expands into a single 32-bit instruction in either the base ISA (RV32I/E, RV64I, or RV128I) or the F and D standard extensions where present. Adopting this constraint has two main benefits:

- Hardware designs can simply expand RVC instructions during decode, simplifying verification and minimizing modifications to existing microarchitectures.
- Compilers can be unaware of the RVC extension and leave code compression to the assembler and linker, although a compression-aware compiler will generally be able to produce better results.

We felt the multiple complexity reductions of a simple one-one mapping between C and base IFD instructions far outweighed the potential gains of a slightly denser encoding that added additional instructions only supported in the C extension, or that allowed encoding of multiple IFD instructions in one C instruction.

It is important to note that the C extension is not designed to be a stand-alone ISA, and is meant to be used alongside a base ISA.

Variable-length instruction sets have long been used to improve code density. For example, the IBM Stretch [2], developed in the late 1950s, had an ISA with 32-bit and 64-bit instructions, where some of the 32-bit instructions were compressed versions of the full 64-bit instructions. Stretch also employed the concept of limiting the set of registers that were addressable in some of the shorter instruction formats, with short branch instructions that could only refer to one of the index registers. The later IBM 360 architecture [1] supported a simple variable-length instruction encoding with 16-bit, 32-bit, or 48-bit instruction formats.

In 1963, CDC introduced the Cray-designed CDC 6600 [3], a precursor to RISC architectures, that introduced a register-rich load-store architecture with instructions of two lengths, 15-bits and 30-bits. The later Cray-1 design used a very similar instruction format, with 16-bit and 32-bit instruction lengths.

The initial RISC ISAs from the 1980s all picked performance over code size, which was reasonable for a workstation environment, but not for embedded systems. Hence, both ARM and MIPS subsequently made versions of the ISAs that offered smaller code size by offering an alternative 16-bit wide instruction set instead of the standard 32-bit wide instructions. The compressed RISC ISAs reduced code size relative to their starting points by about 25–30%, yielding code that was significantly smaller *than 80x86. This result surprised some, as their intuition was that the variable-length CISC ISA should be smaller than RISC ISAs that offered only 16-bit and 32-bit formats.*

Since the original RISC ISAs did not leave sufficient opcode space free to include these unplanned compressed instructions, they were instead developed as complete new ISAs. This meant compilers needed different code generators for the separate compressed ISAs. The first compressed RISC ISA extensions (e.g., ARM Thumb and MIPS16) used only a fixed 16-bit instruction size, which gave good reductions in static code size but caused an increase in dynamic instruction count, which led to lower performance compared to the original fixed-width 32-bit instruction size. This led to the development of a second generation of compressed RISC ISA designs with mixed 16-bit and 32-bit instruction lengths (e.g., ARM Thumb2, microMIPS, PowerPC VLE), so that performance was similar to pure 32-bit instructions but with significant code size savings. Unfortunately, these different generations of compressed ISAs are incompatible with each other and with the original uncompressed ISA, leading to significant complexity in documentation, implementations, and software tools support.

Of the commonly used 64-bit ISAs, only PowerPC and microMIPS currently supports a compressed instruction format. It is surprising that the most popular 64-bit ISA for mobile platforms (ARM v8) does not include a compressed instruction format given that static code size and dynamic instruction fetch bandwidth are important metrics. Although static code size is not a major concern in larger systems, instruction fetch bandwidth can be a major bottleneck in servers running commercial workloads, which often have a large instruction working set.

Benefiting from 25 years of hindsight, RISC-V was designed to support compressed instructions from the outset, leaving enough opcode space for RVC to be added as a simple extension on top of the base ISA (along with many other extensions). The philosophy of RVC is to reduce code size for embedded applications and *to improve performance and energy-efficiency for all applications due to fewer misses in the instruction cache. Waterman shows that RVC fetches 25%-30% fewer instruction bits, which reduces instruction cache misses by 20%-25%, or roughly the same performance impact as doubling the instruction cache size [4].*

1.3 Compressed Instruction Formats

Table 1.1 shows the eight compressed instruction formats. CR, CI, and CSS can use any of the 32 RVI registers, but CIW, CL, CS, and CB are limited to just 8 of them. Table 1.2 lists these popular registers, which correspond to registers `x8` to `x15`. Note that there is a separate version of load and store instructions that use the stack pointer as the base address register, since saving to and restoring from the stack are so prevalent, and that they use the CI and CSS formats to allow access to all 32 data registers. CIW supplies an 8-bit immediate for the ADDI4SPN instruction.

The RISC-V ABI was changed to make the frequently used registers map to registers `x8`–`x15`*. This simplifies the decompression decoder by having a contiguous naturally aligned set of register numbers, and is also compatible with the RV32E subset base specification, which only has 16 integer registers.*

Compressed register-based floating-point loads and stores also use the CL and CS formats respectively, with the eight registers mapping to `f8` to `f15`.

The standard RISC-V calling convention maps the most frequently used floating-point registers to registers `f8` *to* `f15`*, which allows the same register decompression decoding as for integer register numbers.*

The formats were designed to keep bits for the two register source specifiers in the same place in all instructions, while the destination register field can move. When the full 5-bit destination register specifier is present, it is in the same place as in the 32-bit RISC-V encoding. Where immediates are sign-extended, the sign-extension is always from bit 12. Immediate fields have been scrambled, as in the base specification, to reduce the number of immediate muxes required.

The immediate fields are scrambled in the instruction formats instead of in sequential order so that as many bits as possible are in the same position in every instruction, thereby simplifying implementations. For example, immediate bits 17—10 are always sourced from the same instruction bit positions. Five other immediate bits (5, 4, 3, 1, and 0) have just two source instruction bits, while four (9, 7, 6, and 2) have three sources and one (8) has four sources.

For many RVC instructions, zero-valued immediates are disallowed and `x0` is not a valid 5-bit register specifier. These restrictions free up encoding space for other instructions requiring fewer operand bits.

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

Table 1.1: Compressed 16-bit RVC instruction formats.

RVC Register Number	000	001	010	011	100	101	110	111
Integer Register Number	`x8`	`x9`	`x10`	`x11`	`x12`	`x13`	`x14`	`x15`
Integer Register ABI Name	`s0`	`s1`	`a0`	`a1`	`a2`	`a3`	`a4`	`a5`
Floating-Point Register Number	`f8`	`f9`	`f10`	`f11`	`f12`	`f13`	`f14`	`f15`
Floating-Point Register ABI Name	`fs0`	`fs1`	`fa0`	`fa1`	`fa2`	`fa3`	`fa4`	`fa5`

Table 1.2: Registers specified by the three-bit rs1', rs2', and rd' fields of the CIW, CL, CS, and CB formats.

1.4 Load and Store Instructions

To increase the reach of 16-bit instructions, data-transfer instructions use zero-extended immediates that are scaled by the size of the data in bytes: ×4 for words, ×8 for double words, and ×16 for quad words.

RVC provides two variants of loads and stores. One uses the ABI stack pointer, `x2`, as the base address and can target any data register. The other can reference one of 8 base address registers and one of 8 data registers.

Stack-Pointer-Based Loads and Stores

15 13	12	11 7	6 2	1 0
funct3	imm	rd	imm	op
3	1	5	5	2
C.LWSP	offset[5]	dest≠0	offset[4:2|7:6]	C2
C.LDSP	offset[5]	dest≠0	offset[4:3|8:6]	C2
C.LQSP	offset[5]	dest≠0	offset[4|9:6]	C2
C.FLWSP	offset[5]	dest	offset[4:2|7:6]	C2
C.FLDSP	offset[5]	dest	offset[4:3|8:6]	C2

These instructions use the CI format.

C.LWSP loads a 32-bit value from memory into register *rd*. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the stack pointer, `x2`. It expands to `lw rd, offset[7:2](x2)`.

C.LDSP is an RV64C/RV128C-only instruction that loads a 64-bit value from memory into register *rd*. It computes its effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, `x2`. It expands to `ld rd, offset[8:3](x2)`.

C.LQSP is an RV128C-only instruction that loads a 128-bit value from memory into register *rd*. It computes its effective address by adding the zero-extended offset, scaled by 16, to the stack pointer, `x2`. It expands to `lq rd, offset[9:4](x2)`.

C.FLWSP is an RV32FC-only instruction that loads a single-precision floating-point value from memory into floating-point register *rd*. It computes its effective address by adding the *zero*-extended offset, scaled by 4, to the stack pointer, `x2`. It expands to `flw rd, offset[7:2](x2)`.

C.FLDSP is an RV32DC/RV64DC-only instruction that loads a double-precision floating-point value from memory into floating-point register *rd*. It computes its effective address by adding the *zero*-extended offset, scaled by 8, to the stack pointer, `x2`. It expands to `fld rd, offset[8:3](x2)`.

15 13	12 7	6 2	1 0
funct3	imm	rs2	op
3	6	5	2
C.SWSP	offset[5:2|7:6]	src	C2
C.SDSP	offset[5:3|8:6]	src	C2
C.SQSP	offset[5:4|9:6]	src	C2
C.FSWSP	offset[5:2|7:6]	src	C2
C.FSDSP	offset[5:3|8:6]	src	C2

These instructions use the CSS format.

C.SWSP stores a 32-bit value in register *rs2* to memory. It computes an effective address by

adding the *zero*-extended offset, scaled by 4, to the stack pointer, `x2`. It expands to `sw rs2, offset[7:2](x2)`.

C.SDSP is an RV64C/RV128C-only instruction that stores a 64-bit value in register *rs2* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the stack pointer, `x2`. It expands to `sd rs2, offset[8:3](x2)`.

C.SQSP is an RV128C-only instruction that stores a 128-bit value in register *rs2* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 16, to the stack pointer, `x2`. It expands to `sq rs2, offset[9:4](x2)`.

C.FSWSP is an RV32FC-only instruction that stores a single-precision floating-point value in floating-point register *rs2* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the stack pointer, `x2`. It expands to `fsw rs2, offset[7:2](x2)`.

C.FSDSP is an RV32DC/RV64DC-only instruction that stores a double-precision floating-point value in floating-point register *rs2* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the stack pointer, `x2`. It expands to `fsd rs2, offset[8:3](x2)`.

Register-Based Loads and Stores

15 13	12 10	9 7	6 5	4 2	1 0
funct3	imm	rs1′	imm	rd′	op
3	3	3	2	3	2
C.LW	offset[5:3]	base	offset[2|6]	dest	C0
C.LD	offset[5:3]	base	offset[7:6]	dest	C0
C.LQ	offset[5|4|8]	base	offset[7:6]	dest	C0
C.FLW	offset[5:3]	base	offset[2|6]	dest	C0
C.FLD	offset[5:3]	base	offset[7:6]	dest	C0

These instructions use the CL format.

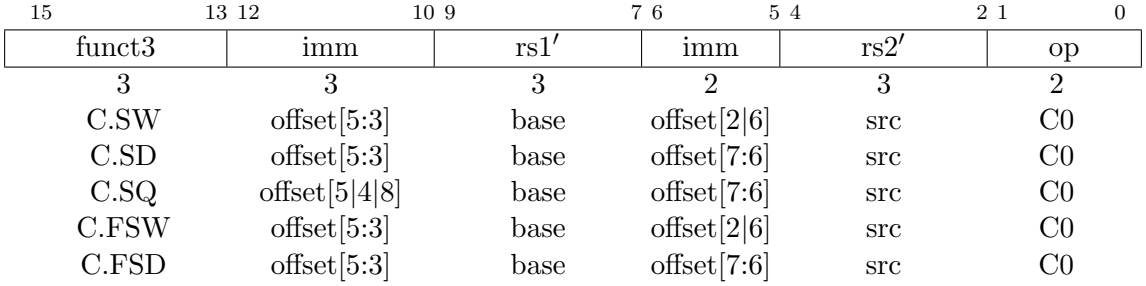
C.LW loads a 32-bit value from memory into register *rd′*. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register *rs1′*. It expands to `lw rd′, offset[6:2](rs1′)`.

C.LD is an RV64C/RV128C-only instruction that loads a 64-bit value from memory into register *rd′*. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register *rs1′*. It expands to `ld rd′, offset[7:3](rs1′)`.

C.LQ is an RV128C-only instruction that loads a 128-bit value from memory into register *rd′*. It computes an effective address by adding the *zero*-extended offset, scaled by 16, to the base address in register *rs1′*. It expands to `lq rd′, offset[8:4](rs1′)`.

C.FLW is an RV32FC-only instruction that loads a single-precision floating-point value from memory into floating-point register *rd′*. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register *rs1′*. It expands to `flw rd′, offset[6:2](rs1′)`.

C.FLD is an RV32DC/RV64DC-only instruction that loads a double-precision floating-point value from memory into floating-point register *rd′*. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register *rs1′*. It expands to `fld rd′, offset[7:3](rs1′)`.



15 13	12 10	9 7	6 5	4 2	1 0
funct3	imm	rs1′	imm	rs2′	op
3	3	3	2	3	2
C.SW	offset[5:3]	base	offset[2|6]	src	C0
C.SD	offset[5:3]	base	offset[7:6]	src	C0
C.SQ	offset[5|4|8]	base	offset[7:6]	src	C0
C.FSW	offset[5:3]	base	offset[2|6]	src	C0
C.FSD	offset[5:3]	base	offset[7:6]	src	C0

These instructions use the CS format.

C.SW stores a 32-bit value in register *rs2′* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register *rs1′*. It expands to `sw rs2′, offset[6:2](rs1′)`.

C.SD is an RV64C/RV128C-only instruction that stores a 64-bit value in register *rs2′* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register *rs1′*. It expands to `sd rs2′, offset[7:3](rs1′)`.

C.SQ is an RV128C-only instruction that stores a 128-bit value in register *rs2′* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 16, to the base address in register *rs1′*. It expands to `sq rs2′, offset[8:4](rs1′)`.

C.FSW is an RV32FC-only instruction that stores a single-precision floating-point value in floating-point register *rs2′* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register *rs1′*. It expands to `fsw rs2′, offset[6:2](rs1′)`.

C.FSD is an RV32DC/RV64DC-only instruction that stores a double-precision floating-point value in floating-point register *rs2′* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register *rs1′*. It expands to `fsd rs2′, offset[7:3](rs1′)`.

1.5 Control Transfer Instructions

RVC provides unconditional jump instructions and conditional branch instructions. As with base RVI instructions, the offsets of all RVC control transfer instruction are in multiples of 2 bytes.

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

These instructions use the CJ format.

C.J performs an unconditional control transfer. The offset is sign-extended and added to the `pc` to form the jump target address. C.J can therefore target a ±2 KiB range. C.J expands to `jal x0, offset[11:1]`.

C.JAL is an RV32C-only instruction that performs the same operation as C.J, but additionally writes the address of the instruction following the jump (`pc`+2) to the link register, `x1`. C.JAL expands to `jal x1, offset[11:1]`.

15 12	11 7	6 2	1 0
funct4	rs1	rs2	op
4	5	5	2
C.JR	src≠0	0	C2
C.JALR	src≠0	0	C2

These instructions use the CR format.

C.JR (jump register) performs an unconditional control transfer to the address in register *rs1*. C.JR expands to `jalr x0, rs1, 0`.

C.JALR (jump and link register) performs the same operation as C.JR, but additionally writes the address of the instruction following the jump (`pc`+2) to the link register, `x1`. C.JALR expands to `jalr x1, rs1, 0`.

Strictly speaking, C.JALR does not expand exactly to a base RVI instruction as the value added to the PC to form the link address is 2 rather than 4 as in the base ISA, but supporting both offsets of 2 and 4 bytes is only a very minor change to the base microarchitecture.

15 13	12 10	9 7	6 2	1 0
funct3	imm	rs1′	imm	op
3	3	3	5	2
C.BEQZ	offset[8|4:3]	src	offset[7:6|2:1|5]	C1
C.BNEZ	offset[8|4:3]	src	offset[7:6|2:1|5]	C1

These instructions use the CB format.

C.BEQZ performs conditional control transfers. The offset is sign-extended and added to the `pc` to form the branch target address. It can therefore target a ±256 B range. C.BEQZ takes the branch if the value in register *rs1′* is zero. It expands to `beq rs1′, x0, offset[8:1]`.

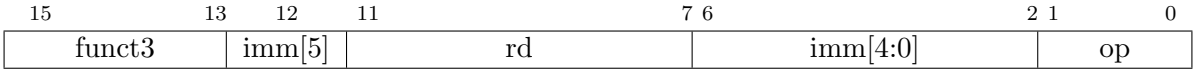
C.BNEZ is defined analogously, but it takes the branch if *rs1′* contains a nonzero value. It expands to `bne rs1′, x0, offset[8:1]`.

1.6 Integer Computational Instructions

RVC provides several instructions for integer arithmetic and constant generation.

Integer Constant-Generation Instructions

The two constant-generation instructions both use the CI instruction format and can target any integer register.



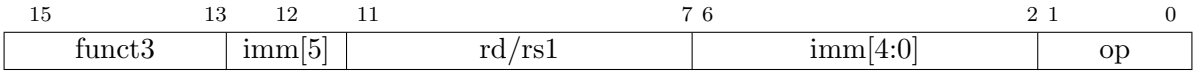
3	1	5	5	2
C.LI	imm[5]	dest≠0	imm[4:0]	C1
C.LUI	nzimm[17]	dest≠{0, 2}	nzimm[16:12]	C1

C.LI loads the sign-extended 6-bit immediate, *imm*, into register *rd*. C.LI is only valid when *rd*≠`x0`. C.LI expands into `addi rd, x0, imm[5:0]`.

C.LUI loads the non-zero 6-bit immediate field into bits 17–12 of the destination register, clears the bottom 12 bits, and sign-extends bit 17 into all higher bits of the destination. C.LUI is only valid when *rd*≠{`x0`, `x2`}, and when the immediate is not equal to zero. C.LUI expands into `lui rd, nzimm[17:12]`.

Integer Register-Immediate Operations

These integer register-immediate operations are encoded in the CI format and perform operations on any non-`x0` integer register and a 6-bit immediate. The immediate cannot be zero.



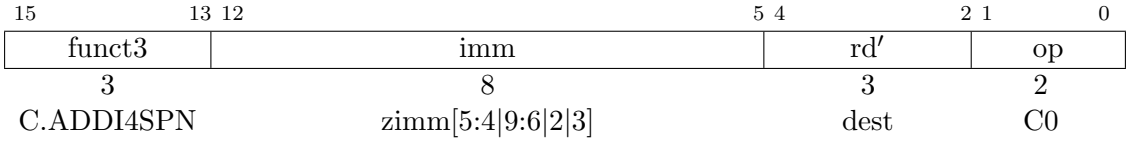
3	1	5	5	2
C.ADDI	nzimm[5]	dest	nzimm[4:0]	C1
C.ADDIW	imm[5]	dest≠0	imm[4:0]	C1
C.ADDI16SP	nzimm[9]	2	nzimm[4|6|8:7|5]	C1

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]`.

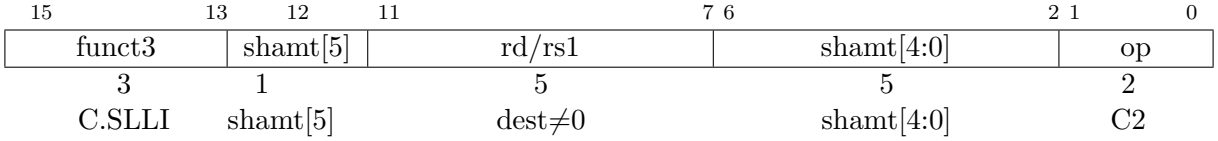
C.ADDIW is an RV64C/RV128C-only instruction that performs the same computation but produces a 32-bit result, then sign-extends result to 64 bits. C.ADDIW expands into `addiw rd, rd, imm[5:0]`. The immediate can be zero for C.ADDIW, where this corresponds to `sext.w rd`.

C.ADDI16SP shares the opcode with C.LUI, but has a destination field of `x2`. 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). C.ADDI16SP is used to adjust the stack pointer in procedure prologues and epilogues. It expands into `addi x2, x2, nzimm[9:4]`.

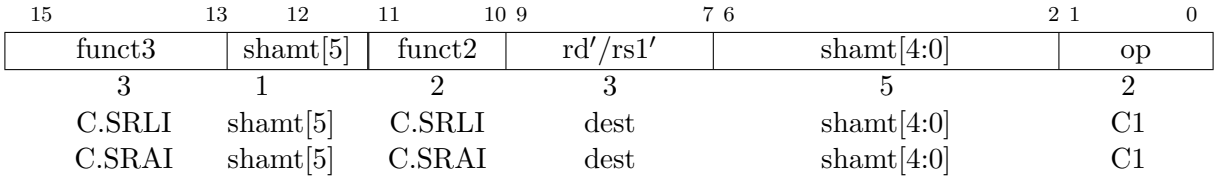
In the standard RISC-V calling convention, the stack pointer `sp` *is always 16-byte aligned.*



C.ADDI4SPN is a CIW-format RV32C/RV64C-only instruction that adds a *zero*-extended non-zero immediate, scaled by 4, to the stack pointer, `x2`, and writes the result to `rd′`. This instruction is used to generate pointers to stack-allocated variables, and expands to `addi rd′, x2, zimm[9:2]`.



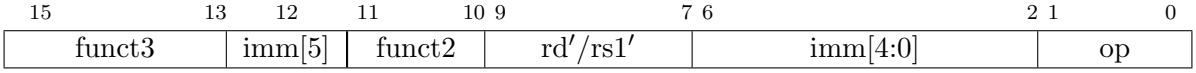
C.SLLI is a CI-format instruction that performs a logical left shift of the value in register *rd* then writes the result to *rd*. The shift amount is encoded in the *shamt* field, where *shamt[5]* must be zero for RV32C. For RV32C and RV64C, the shift amount must be non-zero. For RV128C, a shift amount of zero is used to encode a shift of 64. C.SLLI expands into `slli rd, rd, shamt[5:0]`, except for RV128C with `shamt=0`, which expands to `slli rd, rd, 64`.



C.SRLI is a CB-format instruction that performs a logical right shift of the value in register *rd′* then writes the result to *rd′*. The shift amount is encoded in the *shamt* field, where *shamt[5]* must be zero for RV32C. For RV32C and RV64C, the shift amount must be non-zero. For RV128C, a shift amount of zero is used to encode a shift of 64. Furthermore, the shift amount is sign-extended for RV128C, and so the legal shift amounts are 1–31, 64, and 96–127. C.SRLI expands into `srli rd′, rd′, shamt[5:0]`, except for RV128C with `shamt=0`, which expands to `srli rd′, rd′, 64`.

C.SRAI is defined analogously to C.SRLI, but instead performs an arithmetic right shift. C.SRAI expands to `srai rd′, rd′, shamt[5:0]`.

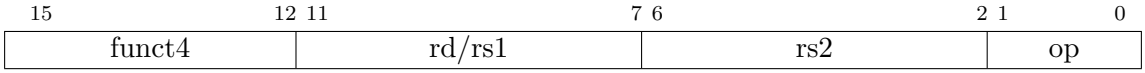
Left shifts are usually more frequent than right shifts, as left shifts are frequently used to scale address values. Right shifts have therefore been granted less encoding space and are placed in an encoding quadrant where all other immediates are sign-extended. For RV128, the decision was made to have the 6-bit shift-amount immediate also be sign-extended. Apart from reducing the decode complexity, we believe right-shift amounts of 96–127 will be more useful than 64–95, to allow extraction of tags located in the high portions of 128-bit address pointers. We note that RV128C will not be frozen at the same point as RV32C and RV64C, to allow evaluation of typical usage of 128-bit address-space codes.



3	1	2	3	5	2
C.ANDI	imm[5]	C.ANDI	dest	imm[4:0]	C1

C.ANDI is a CB-format instruction that computes the bitwise AND of of the value in register *rd′* and the sign-extended 6-bit immediate, then writes the result to *rd′*. C.ANDI expands to `andi rd′, rd′, imm[5:0]`.

Integer Register-Register Operations

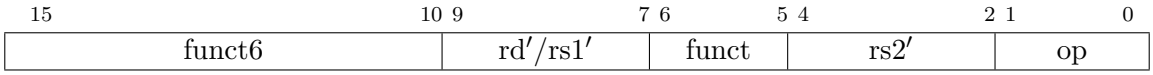


4	5	5	2
C.MV	dest≠0	src≠0	C0
C.ADD	dest≠0	src≠0	C0

These instructions use the CR format.

C.MV copies the value in register *rs2* into register *rd*. C.MV expands into `add rd, x0, rs2`.

C.ADD adds the values in registers *rd* and *rs2* and writes the result to register *rd*. C.ADD expands into `add rd, rd, rs2`.



6	3	2	3	2
C.AND	dest	C.AND	src	C1
C.OR	dest	C.OR	src	C1
C.XOR	dest	C.XOR	src	C1
C.SUB	dest	C.SUB	src	C1
C.ADDW	dest	C.ADDW	src	C1
C.SUBW	dest	C.SUBW	src	C1

These instructions use the CS format.

C.AND computes the bitwise AND of the values in registers *rd′* and *rs2′*, then writes the result to register *rd′*. C.AND expands into `and rd′, rd′, rs2′`.

C.OR computes the bitwise OR of the values in registers *rd′* and *rs2′*, then writes the result to register *rd′*. C.OR expands into `or rd′, rd′, rs2′`.

C.XOR computes the bitwise XOR of the values in registers *rd′* and *rs2′*, then writes the result to register *rd′*. C.XOR expands into `xor rd′, rd′, rs2′`.

C.SUB subtracts the value in register *rs2′* from the value in register *rd′*, then writes the result to register *rd′*. C.SUB expands into `sub rd′, rd′, rs2′`.

C.ADDW is an RV64C/RV128C-only instruction that adds the values in registers *rd′* and *rs2′*, then sign-extends the lower 32 bits of the sum before writing the result to register *rd′*. C.ADDW expands into `addw rd′, rd′, rs2′`.

C.SUBW is an RV64C/RV128C-only instruction that subtracts the value in register *rs2′* from the value in register *rd′*, then sign-extends the lower 32 bits of the difference before writing the result to register *rd′*. C.SUBW expands into `subw rd′, rd′, rs2′`.

This group of six instructions do not provide large savings individually, but do not occupy much encoding space and are straightforward to implement, and as a group provide a worthwhile improvement in static and dynamic compression.

Defined Illegal Instruction

15–13	12	11–7	6–2	1–0
0	0	0	0	0
3	1	5	5	2
0	0	0	0	0

A 16-bit instruction with all bits zero is permanently reserved as an illegal instruction.

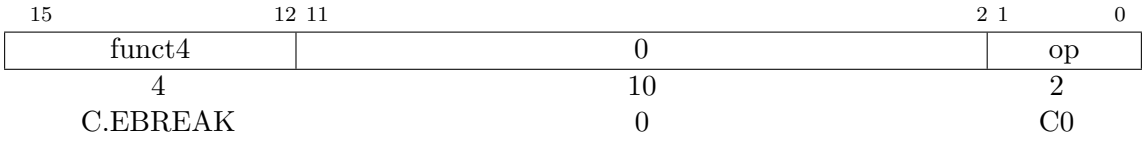
We reserve all-zero instructions to be illegal instructions to help trap attempts to execute zero-ed or non-existent portions of the memory space. The all-zero value should not be redefined in any non-standard extension. Similarly, we reserve instructions with all bits set to 1 (corresponding to very long instructions in the RISC-V variable-length encoding scheme) as illegal to capture another common value seen in non-existent memory regions.

NOP Instruction

15–13	12	11–7	6–2	1–0
funct3	imm[5]	rd/rs1	imm[4:0]	op
3	1	5	5	2
C.NOP	0	0	0	C1

C.NOP is a CI-format instruction that does not change any user-visible state, except for advancing the `pc`. C.NOP is encoded as `c.addi x0, 0` and so expands to `addi x0, x0, 0`.

Breakpoint Instruction



Debuggers can use the C.EBREAK instruction, which expands to `ebreak`, to cause control to be transferred back to the debugging environment. C.EBREAK shares the opcode with the C.ADD instruction, but with *rd* and *rs2* both zero, thus can also use the CR format.

1.7 Optimizing Register Save/Restore Code Size

Register save/restore code at function entry/exit represents a significant portion of static code size. The stack-pointer-based compressed loads and stores in RVC are effective at reducing the save/restore static code size by a factor of 2 while improving performance by reducing dynamic instruction bandwidth.

The standard RISC-V toolchain provides an alternative approach to reduce save/restore static code size even further in exchange for reduced performance. Instead of inlining the register save/restore code in each function, register save code is replaced with a jump-and-link instruction to call a subroutine to copy registers to the stack then return to the function. Register restore code is replaced with a jump to a routine that restores registers from the stack then jumps to the restored return address.

Figure 1.1 shows the impact on static code size and dynamic instruction count of these routines when naïvely applied to all functions in the SPEC CPU2006 benchmarks. On average, code size is reduced by 4% in exchange for a 3% increase in dynamic instruction count.

The inline save/restore code is replaced with calls to the save/restore subroutines when the `-Os` *flag (reduce code size) is passed to* `gcc`.

A common alternative mechanism used in other ISAs to reduce save/restore code size is load-multiple and store-multiple instructions. We considered adopting these for RISC-V but noted the following drawbacks to these instructions:

- *These instructions complicate processor implementations.*
- *For virtual memory systems, some data accesses could be resident in physical memory and some could not, which requires a new restart mechanism for partially executed instructions.*
- *Unlike the rest of the RVC instructions, there is no IFD equivalent to Load Multiple and Store Multiple.*
- *Unlike the rest of the RVC instructions, the compiler would have to be aware of these instructions to both generate the instructions and to allocate registers in an order to maximize the chances of the them being saved and stored, since they would be saved and restored in sequential order.*
- *Simple microarchitectural implementations will constrain how other instructions can be scheduled around the load and store multiple instructions, leading to a potential performance loss.*

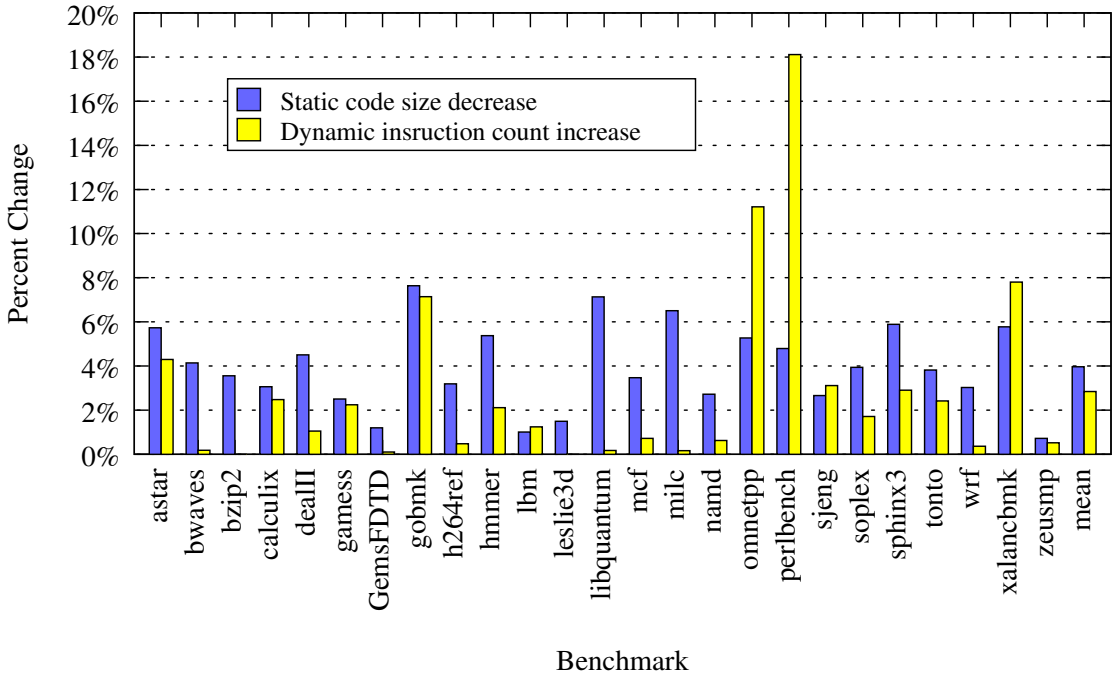


Figure 1.1: Impact on static code size and dynamic instruction count of compressed function prologue and epilogue subroutines.

- *The desire for sequential register allocation might conflict with the featured registers selected for the CIW, CL, CS, and CB formats.*

While reasonable architects might come to different conclusions, we decided to omit load and store multiple and instead use the software-only approach of calling save/restore millicode routines to attain the greatest code size reduction.

1.8 RVC Instruction Set Listings

Table 1.3 shows a map of the major opcodes for RVC. Opcodes with the lower two bits set correspond to instructions wider than 16 bits, including those in the base ISAs. Several instructions are only valid for certain operands; when invalid, they are marked either *RES* to indicate that the opcode is reserved for future standard extensions; *NSE* to indicate that the opcode is reserved for non-standard extensions; or *HINT* to indicate that the opcode is reserved for future standard microarchitectural hints. Instructions marked *HINT* must execute as no-ops on implementations for which the hint has no effect.

The HINT instructions are designed to support future addition of microarchitectural hints that might affect performance but cannot affect architectural state. The HINT encodings have been chosen so that simple implementations can ignore the HINT encoding and execute the HINT as regular operation that does not change architectural state. For example, C.ADD is a HINT if the destination register is `x0`*, where the five-bit rs2 field encodes details of the HINT. However, a simple implementation can simply execute the HINT as an add to register* `x0`*, which will be ignored.*

Tables 1.4–1.6 list the RVC instructions.

inst[15:13] / inst[1:0]	000	001	010	011	100	101	110	111	
00	ADDI4SPN	FLD FLD LQ	LW	FLW LD LD	*Reserved*	FSD FSD SQ	SW	FSW SD SD	RV32 RV64 RV128
01	ADDI	JAL ADDIW ADDIW	LI	LUI/ADDI16SP	MISC-ALU	J	BEQZ	BNEZ	RV32 RV64 RV128
10	SLLI	FLDSP FLDSP LQ	LWSP	FLWSP LDSP LDSP	J[AL]R/MV/ADD	FSDSP FSDSP SQ	SWSP	FSWSP SDSP SDSP	RV32 RV64 RV128
11	>16b								

Table 1.3: RVC opcode map

15 14 13	12 11 10	9 8 7	6 5	4 3 2	1 0	
000	0			0	00	*Illegal instruction*
000	nzimm[5:4|9:6|2|3]			rd′	00	C.ADDI4SPN *(RES, nzimm=0)*
001	imm[5:3]	rs1′	imm[7:6]	rd′	00	C.FLD *(RV32/64)*
001	imm[5:4|8]	rs1′	imm[7:6]	rd′	00	C.LQ *(RV128)*
010	imm[5:3]	rs1′	imm[2|6]	rd′	00	C.LW
011	imm[5:3]	rs1′	imm[2|6]	rd′	00	C.FLW *(RV32)*
011	imm[5:3]	rs1′	imm[7:6]	rd′	00	C.LD *(RV64/128)*
100	—				00	*Reserved*
101	imm[5:3]	rs1′	imm[7:6]	rs2′	00	C.FSD *(RV32/64)*
101	imm[5:4|8]	rs1′	imm[7:6]	rs2′	00	C.SQ *(RV128)*
110	imm[5:3]	rs1′	imm[2|6]	rs2′	00	C.SW
111	imm[5:3]	rs1′	imm[2|6]	rs2′	00	C.FSW *(RV32)*
111	imm[5:3]	rs1′	imm[7:6]	rs2′	00	C.SD *(RV64/128)*

Table 1.4: Instruction listing for RVC, Quadrant 0.

15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0	
000	0	0		0		01	C.NOP
000	nzimm[5]	rs1/rd≠0		nzimm[4:0]		01	C.ADDI *(RES, nzimm=0; HINT, rd=0)*
001	offset[11|4|9:8|10|6|7|3:1|5]					01	C.JAL *(RV32)*
001	imm[5]	rs1/rd≠0		imm[4:0]		01	C.ADDIW *(RV64/128; RES, rd=0)*
010	imm[5]	rs1/rd≠0		imm[4:0]		01	C.LI *(HINT, rd=0)*
011	nzimm[9]	2		nzimm[4|6|8:7|5]		01	C.ADDI16SP *(RES, nzimm=0)*
011	nzimm[17]	rs1/rd≠{0, 2}		nzimm[16:12]		01	C.LUI *(RES, nzimm=0; HINT, rd=0)*
100	nzimm[5]	00	rs1′/rd′	nzimm[4:0]		01	C.SRLI *(RV32 NSE, nzimm[5]=1)*
100	0	00	rs1′/rd′	0		01	C.SRLI64 *(RV128; RV32/64 HINT)*
100	nzimm[5]	01	rs1′/rd′	nzimm[4:0]		01	C.SRAI *(RV32 NSE, nzimm[5]=1)*
100	0	01	rs1′/rd′	0		01	C.SRAI64 *(RV128; RV32/64 HINT)*
100	imm[5]	10	rs1′/rd′	imm[4:0]		01	C.ANDI
100	0	11	rs1′/rd′	00	rs2′	01	C.SUB
100	0	11	rs1′/rd′	01	rs2′	01	C.XOR
100	0	11	rs1′/rd′	10	rs2′	01	C.OR
100	0	11	rs1′/rd′	11	rs2′	01	C.AND
100	1	11	rs1′/rd′	00	rs2′	01	C.SUBW *(RV64/128; RV32 RES)*
100	1	11	rs1′/rd′	01	rs2′	01	C.ADDW *(RV64/128; RV32 RES)*
100	1	11	—	10	—	01	*Reserved*
100	1	11	—	11	—	01	*Reserved*
101	offset[11|4|9:8|10|6|7|3:1|5]					01	C.J
110	offset[8:4|3]		rs1′	offset[7:6|2:1|5]		01	C.BEQZ
111	offset[8:4|3]		rs1′	offset[7:6|2:1|5]		01	C.BNEZ

Table 1.5: Instruction listing for RVC, Quadrant 1.

15 14 13	12	11 10 9 8 7	6 5 4 3 2	1 0	
000	nzimm[5]	rd≠0	nzimm[4:0]	10	C.SLLI *(RV32 NSE, nzimm[5]=1)*
000	0	rd≠0	0	10	C.SLLI64 *(RV128; RV32/64 HINT)*
001	imm[5]	rd	imm[4:3|8:6]	10	C.FLDSP *(RV32/64)*
001	imm[5]	rd≠0	imm[4|9:6]	10	C.LQSP *(RV128; RES, rd=0)*
010	imm[5]	rd≠0	imm[4:2|7:6]	10	C.LWSP *(RES, rd=0)*
011	imm[5]	rd	imm[4:2|7:6]	10	C.FLWSP *(RV32)*
011	imm[5]	rd≠0	imm[4:3|8:6]	10	C.LDSP *(RV64/128; RES, rd=0)*
100	0	rs1≠0	0	10	C.JR *(RES, rs1=0)*
100	0	rd≠0	rs2≠0	10	C.MV *(HINT, rd=0)*
100	1	0	0	10	C.EBREAK
100	1	rs1≠0	0	10	C.JALR
100	1	rd≠0	rs2≠0	10	C.ADD *(HINT, rd=0)*
101	imm[5:3|8:6]		rs2	10	C.FSDSP *(RV32/64)*
101	imm[5:4|9:6]		rs2	10	C.SQSP *(RV128)*
110	imm[5:2|7:6]		rs2	10	C.SWSP
111	imm[5:2|7:6]		rs2	10	C.FSWSP *(RV32)*
111	imm[5:3|8:6]		rs2	10	C.SDSP *(RV64/128)*

Table 1.6: Instruction listing for RVC, Quadrant 2.

1.9 Instruction Compression Statistics

The following tables provide some data we used to guide the selection of instructions to include in RVC.

Table 1.7 lists the standard RVC instructions with the most frequent first, showing the individual contributions of those instructions to static code size and then the running total for three experiments: the SPEC benchmarks for both RV32C and RV64C for the Linux kernel. For RV32, RVC reduces static code size by 24.5% on Dhrystone and 30.9% on CoreMark. For RV64, it reduces static code size by 26.3% on SPECint, 25.8% on SPECfp, and 31.1% on the Linux kernel.

Table 1.8 ranks the RVC instructions by order of typical dynamic frequency. For RV32, RVC reduces dynamic bytes fetched by 29.2% on Dhrystone and 29.3% on CoreMark. For RV64, it reduces dynamic bytes fetched by 26.9% on SPECint, 22.4% on SPECfp, and 26.11% booting the Linux kernel.

Instruction	RV32GC			RV64GC		Max
	Dhry-stone	Core-Mark	SPEC 2006	SPEC 2006	Linux Kernel	
C.MV	1.78	5.03	4.06	3.62	5.00	5.03
C.LWSP	4.51	2.80	2.89	0.49	0.14	4.51
C.LDSP	—	—	—	3.20	4.44	4.44
C.SWSP	4.19	2.45	2.76	0.45	0.18	4.19
C.SDSP	—	—	—	2.75	3.79	3.79
C.LI	2.99	3.74	2.81	2.35	2.86	3.74
C.ADDI	2.16	3.28	1.87	1.19	0.95	3.28
C.ADD	0.51	1.64	1.94	2.28	0.91	2.28
C.LW	2.10	1.68	2.00	0.74	0.62	2.10
C.LD	—	—	—	1.14	2.09	2.09
C.J	0.32	1.71	1.63	0.97	1.53	1.71
C.SW	1.59	0.85	0.73	0.27	0.26	1.59
C.JR	1.52	1.16	0.49	0.44	1.05	1.52
C.BEQZ	0.38	1.14	0.76	0.55	1.24	1.24
C.SLLI	0.06	1.09	0.57	0.93	0.57	1.09
C.ADDI16SP	0.19	0.26	0.32	0.42	1.01	1.01
C.SRLI	0.00	0.81	0.05	0.12	0.31	0.81
C.BNEZ	0.19	0.53	0.53	0.32	0.80	0.80
C.SD	—	—	—	0.25	0.79	0.79
C.ADDIW	—	—	—	0.77	0.50	0.77
C.JAL	0.38	0.59	0.05	—	—	0.59
C.ADDI4SPN	0.57	0.37	0.45	0.50	0.30	0.57
C.LUI	0.32	0.37	0.44	0.56	0.52	0.56
C.SRAI	0.13	0.48	0.07	0.03	0.03	0.48
C.ANDI	0.00	0.42	0.20	0.07	0.35	0.42
C.FLD	0.00	0.00	0.16	0.39	0.00	0.39
C.FLDSP	0.00	0.02	0.20	0.31	0.00	0.31
C.FSDSP	0.13	0.09	0.15	0.26	0.00	0.26
C.SUB	0.25	0.09	0.13	0.06	0.11	0.25
C.AND	0.00	0.00	0.07	0.03	0.21	0.21
C.FSD	0.00	0.00	0.08	0.18	—	0.18
C.OR	0.06	0.18	0.09	0.04	0.14	0.18
C.JALR	0.13	0.07	0.17	0.10	0.14	0.17
C.ADDW	—	—	—	0.16	0.12	0.16
C.EBREAK	0.00	0.02	0.00	0.00	0.08	0.08
C.FLW	0.00	0.00	0.05	—	—	0.05
C.XOR	0.00	0.04	0.01	0.01	0.03	0.04
C.SUBW	—	—	—	0.04	0.03	0.04
C.FLWSP	0.00	0.00	0.03	—	—	0.03
C.FSW	0.00	0.00	0.02	—	—	0.02
C.FSWSP	0.00	0.00	0.02	—	—	0.02
Total	24.46	30.92	25.78	25.98	31.11	—

Table 1.7: RVC instructions in order of typical static frequency. The numbers in the table show the percentage savings in static code size attributable to each instruction. This list was generated using a compacting assembler for the output of the RISC-V GCC compiler, directed to use RV32GC for Dhrystone, CoreMark, and SPEC CPU2006, and RV64GC for SPEC CPU2006 and the Linux kernel, version 3.14.29. A dash means that instruction is not defined for this address size.

Instruction	RV32GC		RV64GC		Max
	Dhry-stone	Core-Mark	SPEC 2006	Linux Kernel	
C.ADDI	3.70	3.91	4.36	1.26	4.36
C.LW	4.15	3.89	1.09	0.87	4.15
C.MV	1.93	4.01	1.70	1.37	4.01
C.BNEZ	0.44	2.57	0.47	3.62	3.62
C.SW	3.55	1.62	0.32	0.68	3.55
C.LD	—	—	1.43	3.29	3.29
C.SWSP	3.26	0.32	0.20	0.03	3.26
C.LWSP	2.96	0.48	0.14	0.02	2.96
C.LI	2.22	1.47	0.81	2.73	2.73
C.ADD	2.07	2.69	2.64	1.84	2.69
C.SRLI	0.00	2.48	0.20	0.38	2.48
C.JR	2.07	0.34	0.46	0.42	2.07
C.FLD	0.00	0.00	1.63	0.00	1.63
C.SDSP	—	—	1.14	1.38	1.38
C.J	0.44	0.46	0.33	1.35	1.35
C.LDSP	—	—	1.34	1.31	1.34
C.ANDI	0.15	1.30	0.10	0.23	1.30
C.ADDIW	—	—	1.26	1.03	1.26
C.SLLI	0.15	1.10	1.24	0.89	1.24
C.SD	—	—	0.39	1.13	1.13
C.BEQZ	0.59	0.95	0.74	0.76	0.95
C.AND	0.00	0.00	0.21	0.75	0.75
C.SRAI	0.00	0.72	0.02	0.01	0.72
C.JAL	0.59	0.26	—	—	0.59
C.ADDI4SPN	0.44	0.16	0.07	0.05	0.44
C.FLDSP	0.00	0.00	0.40	0.00	0.40
C.ADDI16SP	0.13	0.18	0.28	0.38	0.38
C.FSD	0.00	0.00	0.29	0.00	0.29
C.FSDSP	0.00	0.00	0.25	0.00	0.25
C.ADDW	—	—	0.19	0.04	0.19
C.XOR	0.00	0.19	0.06	0.02	0.19
C.OR	0.15	0.08	0.05	0.04	0.15
C.SUB	0.15	0.03	0.05	0.04	0.15
C.LUI	0.02	0.06	0.09	0.10	0.10
C.JALR	0.00	0.05	0.05	0.03	0.05
C.SUBW	—	—	0.04	0.02	0.04
C.EBREAK	0.00	0.00	0.00	0.00	0.00
C.FLW	0.00	0.00	—	—	—
C.FLWSP	0.00	0.00	—	—	—
C.FSW	0.00	0.00	—	—	—
C.FSWSP	0.00	0.00	—	—	—
Total	29.18	29.29	24.03	26.11	—

Table 1.8: RVC instructions in order of typical dynamic frequency. The numbers in the table show the percentage savings in dynamic code size attributable to each instruction. This list was generated by executing CoreMark and Dhrystone compiled for RV32GC and SPEC CPU2006 compiled for RV64GC. For SPEC, we used the reference input set. The Linux boot includes the time to boot the kernel, then execute the init process, the shell, and the poweroff command.

Bibliography

[1] G. M. Amdahl, G. A. Blaauw, and Jr. F. P. Brooks. Architecture of the IBM System/360. *IBM Journal of R. & D.*, 8(2), 1964.

[2] Werner Buchholz, editor. *Planning a computer system: Project Stretch.* McGraw-Hill Book Company, 1962.

[3] James E. Thornton. Parallel operation in the Control Data 6600. In *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*, AFIPS '64 (Fall, part II), pages 33–40, 1965.

[4] Andrew Waterman. Improving energy efficiency and reducing code size with RISC-V compressed. Master's thesis, University of California, Berkeley, 2011.