The RISC-V Compressed Instruction Set Manual, Version 1.7

Andrew Waterman
Yunsup Lee
David A. Patterson
Krste Asanović

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-157
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-157.html

May 28, 2015

The RISC-V Compressed Instruction Set Manual

Version 1.7

Warning! This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely 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

May 28, 2015

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 integer operations. The C extension can be added to any of the base ISAs (RV32I, RV64I, RV128I), and we use the generic term "RVC" to cover any of these. Typically, over half of the RISC-V instructions in a program can be replaced with RVC instructions, resulting in greater than a 25% code-size reduction. Section 1.7 describes a possible extended set of instructions for RVC, for which we would like your opinion.

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`) 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 compressed instruction encodings are mostly common across RV32I, RV64I, and RV128I, with a few differences caused by the different register widths. The C extension is compatible with all other standard instruction extensions. The C extension allows 16-bit instructions to be freely intermixed with the 32-bit base 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.

RVC was designed under the constraint that each RVC instruction expands into one of the base RISC-V instructions: RV32I, RV64I, or RV128I. 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 I 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 I instructions into one C instruction.

One example that we carefully considered before rejecting was load-multiple and store-multiple instructions, which have been used in some other ISAs to reduce register save/restore code size at function entry and exit. These instructions complicate microarchitecture design and verification, and can complicate compiler instruction scheduling around register saves and restores. For a longer discussion, see the rationale at the end of Section 1.7.

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

Variable-length instruction sets have long been used to improve code density. For example, the IBM Stretch, 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 format. The later IBM 360 architecture 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, 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 designs. 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. ARM Thumb and MIPS16 have only 16-bit instructions, while later Thumb2 and MIPS16e/microMIPS variants offer both 16-bit and 32-bit instructions.

Compressed 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 there was not enough opcode space left in the original ISAs to include these unplanned compressed instructions, they instead became new full ISAs. One downside is that to make enough opcode space for the 16-bit instructions, they discarded many 32-bit instructions from their predecessor ISAs. Thus, the dynamic instruction count increases for these ISAs, even though their static code size is smaller. This split meant compilers needed different code generators for the compressed ISAs, and that there was a performance penalty to get smaller programs.

Perhaps more importantly, some compressed ISAs were orphaned as their parent ISAs evolved, as new instructions could not be added to the compact ISAs. For example, when ARM increased address size to 64 bits in ARM v8, Thumb and Thumb2 were left as 32-bit-address-only ISAs.

Leveraging 25 years of hindsight, RISC-V was designed to support compressed instructions from the start, leaving enough opcode space for RVC and many other extensions. As all compressed instructions are variations of RVI instructions—which must be included in every RISC-V implementation—the optional compressed instructions are compatible with all current and future RISC-V extensions. Thus, RV32C, RV64C, and RV128C are all valid, as would be RV32GQLCBTP.

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.[1]*

1.3 Compressed Instruction Formats

Table 1.1 shows the 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 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.

For many RVC instructions, `x0` is not a valid register specifier and zero-valued immediates are disallowed. These restrictions increase the encoding space available to instructions that require 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
RISC-V Register Name	`x8`	`x9`	`x10`	`x11`	`x12`	`x13`	`x14`	`x15`
ABI Register Name	`s0`	`s1`	`a0`	`a1`	`a2`	`a3`	`a4`	`a5`

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 opportunity for 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]	C1
C.LDSP	offset[5]	dest≠0	offset[4:3|8:6]	C1
C.LQSP	offset[5]	dest≠0	offset[4|9:6]	C1

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)`.

15 13	12 7	6 2	1 0
funct3	imm	rs2	op
3	6	5	2
C.SWSP	offset[5:2|7:6]	src	C1
C.SDSP	offset[5:3|8:6]	src	C1
C.SQSP	offset[5:4|9:6]	src	C1

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)`.

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

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′)`.

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

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′)`.

We compared the RVC immediates to more conventional schemes for SPEC2006 programs. The *current scheme is able to use RVC instructions in place of about 50% of static RISC-V instructions. time (see Section 1.8 for more details). Not scaling the load/store immediates by the word size in bytes would reduce that fraction to 43%. If we used two's complement for the load/store immediates instead of zero-extending them, it would further fall to 39%.*

1.5 Control Transfer Instructions

RVC provides unconditional jump instructions and conditional branch instructions. To accommodate 16-bit instructions, the offsets of all RVC control transfer instruction are multiples of 2 bytes, which is also true for RVI instructions.

15 13	12 2	1 0
funct3	imm	op
3	11	2
C.J	offset[11:6|4:1|5]	C2
C.JAL	offset[11:6|4:1|5]	C2

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 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`.

15 13	12 10	9 7	6 2	1 0
funct3	imm	rs1′	imm	op
3	3	3	5	2
C.BEQZ	offset[8:6]	src	offset[4:1|5]	C2
C.BNEZ	offset[8:6]	src	offset[4:1|5]	C2

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]`.

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 implementation. For example, immediate bits 17—10 always come from the same bit positions. Four other immediate bits (4, 3, 1, and 0) have just two choices, while three (9, 8 and 5) have three choices and three (7, 6, and 2) have four choices.

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.

15 13	12	11 7	6 2	1 0
funct3	imm[5]	rd	imm[4:0]	op
3	1	5	5	2
C.LI	nzimm[5]	dest≠0	nzimm[4:0]	C2
C.LUI	nzimm[17]	dest≠0	nzimm[16:12]	C2

C.LI loads the non-zero sign-extended 6-bit immediate, *nzimm*, into register *rd*. C.LI is only valid when *rd*≠`x0`, and when the immediate is not equal to zero. C.LI expands into `addi rd, x0, nzimm[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`, 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.

15 13	12	11 7	6 2	1 0
funct3	imm[5]	rd/rs1	imm[4:0]	op
3	1	5	5	2
C.ADDI	nzimm[5]	dest≠0	nzimm[4:0]	C2
C.ADDIW	imm[5]	dest≠0	imm[4:0]	C2
C.ADDI16SP	nzimm[9]	0	nzimm[4|5|8:6]	C2

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.ADDI, but has a destination field of zero. 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 move the stack pointer in function call/return code, and expands into `addi x2, x2, nzimm[9:4]`.

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

15 13	12 5	4 2	1 0
funct3	imm	rd′	op
3	8	3	2
C.ADDI4SPN	zimm[5:4|9:6|2|3]	dest	C1

C.ADDI4SPN is a CIW-format RV32C/RV64C-only instruction that adds a *zero*-extended 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]`.

15 13	12	11 7	6 2	1 0
funct3	shamt[5]	rd/rs1	shamt[4:0]	op
3	1	5	5	2
C.SLLI	shamt[5]	dest≠0	shamt[4:0]	C1

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`.

Left shifts are considerably more common than right shifts due to their use in address calculations, hence the better support for left shifts versus right shifts in the (standard) compressed extension.

Integer Register-Register Operations

15 12	11 7	6 2	1 0
funct4	rd/rs1	rs2	op
4	5	5	2
C.MV	dest≠0	src	C0
C.ADD	dest≠0	src≠0	C0
C.ADDW	dest≠0	src≠0	C0
C.SUB	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.MV may be used to initialize a register to 0 by setting *rs2* to `x0`.

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`. C.ADDW is an RV64C/RV128C-only instruction that performs the same computation, but produces a 32-bit result then sign-extends it to 64 bits. C.ADDW expands into `addw rd, rd, rs2`.

C.SUB subtracts the value in register *rs2* from the value in register *rd* and writes the result to register *rd*. It expands to `sub rd, rd, rs2`.

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	C2

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

Since C.NOP shares most of its encoding with C.ADDI16SP, low-end implementations may expand C.NOP into `addi sp, sp, 0`*, rather than* `addi x0, x0, 0`*, to slightly simplify the control logic. This optimization may not be desirable for superscalar implementations, for which the additional RAW hazards might induce pipeline breaks.*

Breakpoint Instruction

15 12	11 2	1 0
funct4	0	op
4	10	2
C.EBREAK	0	C0

Debuggers 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 Extended RVC Instructions

The 31 RVC instructions described thus far provide virtually all of the compression savings possible for C programs compiled with the current GCC compiler, which is not aware of the compression scheme. In this section, we describe 24 potential RVC instructions that can be used by assembly-language programmers or possibly generated by other compilers for C and compilers for other languages. Using current GCC code generation, these 24 instructions and 4 new instruction formats only reduce code size by about another 1%. Several of these instructions are only available in RV32, or only RV32 and RV64, as they make use of encoding space occupied by other opcodes for wider base integer registers.

Section 1.8 lists the pros and cons of the extended RVC instructions, and we welcome your advice on whether to include them. Please send your comments to the isa-dev@lists.riscv.org mailing list.

Table 1.3 shows the 4 new compressed instruction formats. CDS, CSD, and CR3 operate on the 8 registers of RVC, with CDS having the destination first of two operands, and CSD having it second of two operands, and CR3 having three operands. CRI adds an immediate to the two restricted register fields.

Format	Meaning	15 14 13	12 11 10	9 8 7	6 5	4 3 2	1 0
CDS	2 Registers, Destination then Source	funct6		rd′	funct2	rs2′	op
CSD	2 Registers, Source then Destination	funct6		rs1′	funct2	rd′	op
CRI	2 Registers and Imm	funct3	imm	rs1′	funct2	rd′	op
CR3	3 Registers	funct3	rd′	rs1′	funct2	rs2′	op

Table 1.3: Compressed 16-bit instruction formats for Extended RVC.

1.7.1 Extended Control Transfer Instructions

Extended RVC adds two conditional branch instructions.

15 13	12 10	9 7	6 2	1 0
funct3	imm	rs1′	imm	op
3	3	3	5	2
C.BLTZ	offset[8:6]	src	offset[4:1|5]	C2
C.BGEZ	offset[8:6]	src	offset[4:1|5]	C2

These instructions use the CB format.

C.BLTZ is an RV32C-only instruction defined analogously to C.BEQZ, but it takes the branch if *rs1′* contains a value less than zero. It expands to `blt rs1′, x0, offset[8:1]`.

C.BGEZ is an RV32C-only instruction defined analogously to C.BEQZ, but it takes the branch if *rs1′* contains a value greater than or equal to zero. It expands to `bge rs1′, x0, offset[8:1]`.

1.7.2 Extended Integer Register-Immediate Operations

Arithmetic-Logical Immediate Instructions

15 13	12	11 7	6 2	1 0
funct3	imm[5]	rd/rs1	imm[4:0]	op
3	1	5	5	2
C.ANDI	nzimm[5]	dest≠0	nzimm[4:0]	C2

C.ANDI is an RV32C-only instruction that performs a bitwise-AND of the non-zero sign-extended 6-bit immediate and the value in register *rd*, then writes the result to rd. C.ANDI expands to `andi rd, rd, nzimm[5:0]`. This instruction uses the CI format and operates on any non-`x0` integer register and on an immediate that cannot be zero.

15 13	12 10	9 7	6 5	4 2	1 0
funct3	imm	rs1′	funct2	rd′	op
3	3	3	2	3	2
OPN	nzimm[2:0]	src1	C.ADDIN	dest	C0
OPN	nzimm[2:0]	src1	C.ANDIN	dest	C0
OPN	nzimm[2:0]	src1	C.ORIN	dest	C0
OPN	nzimm[2:0]	src1	C.XORIN	dest	C0

These instructions use the CRI format.

C.ADDIN is an RV32C-only instruction that adds the non-zero sign-extended 3-bit immediate to the value in register *rs1′* then writes the result to *rd′*. C.ADDIN expands into `addi rd′, rs1′, nzimm[2:0]`.

C.ANDIN, C.ORIN, and C.XORIN are RV32C-only instructions that perform bitwise AND, OR, and XOR, respectively, on the non-zero sign-extended 3-bit immediate and the value in register *rs1′*, then write the result to register *rd′*. They expand to `andi rd′, rs1′, nzimm[2:0]`; `ori rd′, rs1′, nzimm[2:0]`; and `xori rd′, rs1′, nzimm[2:0]`, respectively.

Shift-By-Immediate Instructions

15 13	12	11 7	6 2	1 0
funct3	shamt[5]	rd/rs1	shamt[4:0]	op
3	1	5	5	2
C.SLLIW	0	dest≠0	shamt[4:0]	C1
C.SRLI	shamt[5]	dest≠0	shamt[4:0]	C1
C.SRAI	shamt[5]	dest≠0	shamt[4:0]	C0

These instructions use the CI format.

C.SLLIW is an RV64C/RV128C-only instruction that performs the same computation as C.SLLI but produces a 32-bit result, then sign-extends the result to 64 bits. The shift amount is encoded in the *shamt[4:0]* field; *shamt[5]* must be zero. C.SLLIW expands into `slliw rd, rd, shamt[4:0]`.

C.SRLI is an RV32C/RV64C-only 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. The shift amount must be non-zero. C.SRLI expands into `srli rd, rd, shamt[5:0]`.

C.SRAI is an RV32C/RV64C-only instruction that performs an arithmetic 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. The shift amount must be non-zero. C.SRAI expands into `srai rd, rd, shamt[5:0]`.

1.7.3 Integer Register-Register Operations

Arithmetic-Logical Three-Register Instructions

15 13	12 10	9 7	6 5	4 2	1 0
funct3	rd′	rs1′	funct2	rs2′	op
3	3	3	2	3	2
OP3	dest	src1	C.ADD3	src2	C0
OP3	dest	src1	C.SUB3	src2	C0
OP3	dest	src1	C.AND3	src2	C0
OP3	dest	src1	C.OR3	src2	C0

These instructions use the CR3 format.

C.ADD3 and C.SUB3 are RV32C/RV64C-only instructions that perform addition and subtraction, respectively, on the values in registers *rs1′* and *rs2′*, then write the result to register *rd′*. They expand to `add rd′, rs1′, rs2′` and `sub rd′, rs1′, rs2′`, respectively.

C.AND3 and C.OR3 are RV32C/RV64C-only instructions that perform bitwise AND and OR, respectively, on the values in registers *rs1′* and *rs2′*, then write the result to register *rd′*. They expand to `and rd′, rs1′, rs2′` and `or rd′, rs1′, rs2′`, respectively.

Logical, Shift, and Compare Two-Register Instructions

15 10	9 7	6 5	4 2	1 0
funct6	rd′	funct2	rs2′	op
6	3	2	3	2
C.XOR	dest	C.XOR	src	C0
C.SLL	dest	C.SLL	src	C0
C.SRL	dest	C.SRL	src	C0
C.SRA	dest	C.SRA	src	C0
C.SLT	dest	C.SLT	src	C0
C.SLTU	dest	C.SLTU	src	C0

These instructions use the CDS format.

C.XOR is an RV32C-only instruction that computes the bitwise XOR of the values in registers *rd′* and *rs2′* and writes the result to register *rd′*. It expands to `xor rd′, rd′, rs2′`.

C.SLL, C.SRL, and C.SRA are RV32C-only instructions that perform logical left, logical right, and arithmetic right shifts of the value in register *rd′* by the shift amount held in the lower 5 bits of *rs2′*, then write the result to register *rd′*. They expand to `sll rd′, rd′, rs2′`, `srl rd′, rd′, rs2′`, and `sra rd′, rd′, rs2′`, respectively.

C.SLT is an RV32C-only instruction that writes 1 to register *rd′* if the value in register *rd′* is less than the value in register *rs2′*, using a signed comparison, and writes 0 to *rd′* otherwise. It expands to `slt rd′, rd′, rs2′`. C.SLTU is an RV32C-only instruction that performs the same operation using an unsigned comparison. It expands to `sltu rd′, rd′, rs2′`.

Reversed Shift and Compare Two-Register Instructions

15 10	9 7	6 5	4 2	1 0
funct6	rs1′	funct2	rd′	op
6	3	2	3	2
C.SLLR	src	C.SLLR	dest	C0
C.SRLR	src	C.SRLR	dest	C0
C.SLTR	src	C.SLTR	dest	C0
C.SLTUR	src	C.SLTUR	dest	C0

These instructions use the CSD format.

C.SLLR and C.SRLR are RV32C-only instructions that perform logical left and logical right shifts of the value in register *rs1′* by the shift amount held in the lower 5 bits of *rd′*, then write the result to register *rd′*. They expand to `sll rd′, rs1′, rd′` and `srl rd′, rs1′, rd′`, respectively.

C.SLTR is an RV32C-only instruction that writes 1 to register *rd′* if the value in register *rs1′* is less than the value in register *rd′*, using a signed comparison, and writes 0 to *rd′* otherwise. It expands to `slt rd′, rs1′, rd′`. C.SLTUR is an RV32C-only instruction that performs the same operation using an unsigned comparison. It expands to `sltu rd′, rs1′, rd′`.

There are optimizations that would further shrink code not included in either the standard or extended RVC. As mentioned above, the biggest omission is Load Multiple and Store Multiple, which would be useful at the entry and exit points of procedures. These two instructions, which are found in Thumb and Thumb2, would shrink RVC static code size by perhaps another 5% at most. However, there are 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 RVI 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.*

- *The desire for sequential register allocation might conflict with the featured registers selected for the CR3, CL, CS, CB, CDS, CSD, and CRI formats.*

While reasonable architects might come to different conclusions, we decided to omit load and store multiple.

	RV32C SPEC2006		RV64C SPEC2006		RV64C Linux kernel	
Instruction	%	Running Total	%	Running Total	%	Running Total
C.MV	4.92	4.92	5.05	5.05	6.03	6.03
C.LDSP	-	4.92	2.93	7.98	4.36	10.39
C.SDSP	-	4.92	2.40	10.38	3.73	14.12
C.LI	1.62	6.54	1.56	11.94	1.85	15.97
C.ADD	2.21	8.75	1.93	13.87	0.73	16.70
C.J	1.29	10.04	1.23	15.10	1.67	18.37
C.LWSP	3.38	13.42	0.52	15.62	0.13	18.50
C.ADDI	1.89	15.31	1.02	16.64	1.02	19.52
C.LW	2.22	17.53	0.93	17.57	0.67	20.19
C.LD	-	17.53	1.24	18.81	2.19	22.38
C.SWSP	2.80	20.33	0.37	19.18	0.18	22.56
C.BEQZ	0.65	20.98	0.63	19.81	1.35	23.91
C.JR	0.56	21.54	0.59	20.40	1.15	25.06
C.SLLI	0.62	22.16	0.80	21.20	0.52	25.58
C.ADDI16SP	0.37	22.53	0.55	21.75	0.99	26.57
C.BNEZ	0.34	22.87	0.34	22.09	0.85	27.42
C.ADDIW	-	22.87	0.83	22.92	0.52	27.94
C.LUI	0.41	23.28	0.43	23.35	0.50	28.44
C.SW	0.68	23.96	0.35	23.70	0.30	28.74
C.SD	-	23.96	0.31	24.01	0.88	29.62
C.JAL	0.30	24.26	0.31	24.32	0.44	30.06
C.ADDI4SPN	0.36	24.62	0.33	24.65	0.31	30.37
C.ADDW	-	24.62	0.35	25.00	0.23	30.60
C.SUB	0.19	24.81	0.09	25.09	0.16	30.76
C.JALR	0.14	24.95	0.14	25.23	0.15	30.91
C.EBREAK	0	24.95	0	25.23	0.09	31.00

Table 1.4: Standard RVC instructions in order of typical frequency. The numbers in the table show the percentage savings in static code size that each instruction generated. This list was generated using a compacting assembler for the output of the RISC-V gcc compiler for the SPEC2006 benchmarks with the compiler directed to produce RV32C and RV64GC code and for the Linux kernel for RV64GC code. A dash means that instruction is not defined for this address size. The five omitted RVC instructions are C.NOP and the four quadword loads and stores.

1.8 Pros and Cons of the Extended RVC Instructions

Tables 1.4 and 1.5 list the standard and extended RVC instructions with the most frequent first, showing the individual contributions of those instructions and then the running total for three experiments: the SPEC benchmarks for both RV32C and RV64C for the Linux kernel. The 31 standard RVC instructions shrink programs by 24.95% to 31.00%. The 24 extended RVC instructions shrink programs by an additional 0.55% to 1.06%.

	RV32C SPEC2006		RV64C SPEC2006		RV64C Linux kernel	
Instruction	%	Running Total	%	Running Total	%	Running Total
C.SRLI	0.05	0.05	0.18	0.18	0.38	0.38
C.AND3	0.05	0.10	0.05	0.23	0.23	0.61
C.OR3	0.08	0.18	0.05	0.28	0.18	0.79
C.SUB3	0.10	0.28	0.05	0.33	0.09	0.88
C.ADD3	0.11	0.39	0.07	0.40	0.05	0.93
C.SRAI	0.12	0.51	0.05	0.45	0.04	0.97
C.SLLIW	-	0.51	0.10	0.55	0.09	1.06
C.ANDI	0.11	0.62	-	0.55	-	1.06
C.ADDIN	0.09	0.71	-	0.55	-	1.06
C.BLTZ	0.07	0.78	-	0.55	-	1.06
C.BGEZ	0.03	0.81	-	0.55	-	1.06
C.SLL	0.03	0.84	-	0.55	-	1.06
C.ORIN	0.02	0.86	-	0.55	-	1.06
C.SLLR	0.02	0.88	-	0.55	-	1.06
C.XOR	0.02	0.90	-	0.55	-	1.06
C.ANDIN	0.01	0.91	-	0.55	-	1.06
C.SLTUR	0.01	0.92	-	0.55	-	1.06
C.SRL	0.01	0.93	-	0.55	-	1.06
C.SRLR	0.01	0.94	-	0.55	-	1.06
C.XORIN	0.01	0.95	-	0.55	-	1.06
C.SLT	0	0.95	-	0.55	-	1.06
C.SLTR	0	0.95	-	0.55	-	1.06
C.SLTU	0	0.95	-	0.55	-	1.06
C.SRA	0	0.95	-	0.55	-	1.06

Table 1.5: Extended RVC instructions in order of typical frequency. This list was generated using a compacting assembler for the output of the RISC-V gcc compiler for the SPEC2006 benchmarks with the compiler directed to produce RV32C and RV64GC code and for the Linux kernel for RV64GC code. A dash means that instruction is not defined for this address size.

Here are the arguments for including these extended instructions in RVC:

- Since the assembler can do compression, the compiler code generator does not have to be more complex even though there are more instructions to choose from.
- It is better to put the extra instructions in RVC now, since it will be much harder to add them later after the RVC extension is adopted.
- The RVC instruction decoder probably isn't that much bigger for the extra instructions.
- Our experiments with current compression-oblivious C compilers see swiftly diminishing returns. Perhaps different compression-aware C compilers will actually see greater benefit from

the extended RVC instructions.

- Perhaps other environments, e.g., Java JIT JVM, would give different results.
- Perhaps hand coders would actually use the extended RVC instructions frequently, even if compilers could not.
- While there is additional complexity verification for the extended RVC instructions, it is only in the decode stage as instructions expand to regular instructions that must already be supported. With 16b instructions, it is very feasible to exhaustively test the translation to 32b instructions since there are only 64K test cases.
- Our preference would obviously be to not have two extensions to reduce ISA fragmentation, but maybe the community would prefer a RVC1 and a RVC2, where the long tail is separated out into a separate optional RVC2 extension.

32-bit only	32- or 64-bit	64- or 128-bit
C.BLTZ	C.SRLI	C.SLLIW
C.BGEZ	C.SRAI	
C.ANDI	C.ADD3	
C.ADDIN	C.SUB3	
C.ANDIN	C.AND3	
C.ORIN	C.OR3	
C.XORIN		
C.XOR		
C.SLL		
C.SRL		
C.SRA		
C.SLT		
C.SLTU		
C.SLLR		
C.SRLR		
C.SLTR		
C.SLTUR		

Table 1.6: Extended RVC instructions categorized by which address sizes they are valid. Only one of the 24 extended RVC instructions is valid for all addresses larger than that for which it was originally defined.

Here are the arguments against including the Extended RVC instructions:

- Extended nearly doubles the number of RVC instructions, which in turn doubles the documentation of RVC and doubles what must be verified during implementation of RVC.
- Note that we currently can explain RISC-V as follows:

$$RV32G \subset RV64G \subset RV128G$$

That is, by using wider registers, every RV32G instruction is also defined in RV64G and

RV128G and similarly every RV64G instruction is defined in RV128G.[1] The complete RV64G and RV128G extensions add a few more instructions to operate on data types narrower than 64 or 128 bits. Because extended RVC is trying to fit many instructions into a limited opcode space, it breaks the subset relationship. Table 1.6 shows that 23 of the 24 extended RVC instructions are available in smaller address sizes but not in larger ones. The loss of this relationship makes it more difficult to explain RISC-V, and reduces some of its elegance.

- The standard RVC was data driven by measuring the code size impact of the many alternatives for compressed instructions, and only keeping the ones that proved to make a significant difference using the gcc compiler and the RISC-V assembler. The extended RVC set is intuition driven. We won't know for years if assembly language programmers or alternative compilers will produce extended RVC instructions in sufficient frequency to be worthwhile. The only data we have today suggests they aren't.
- Since we are establishing the standard extensions for RISC-V in 2015, and they are expected to change slowly, it will probably be harder for standards committees to delete existing instructions than it would be to add instructions based on solid numbers from convincing experiments. Thus, we should err on the side of leaving out possibly useful instructions versus including some that will prove to be unworthy but must be implemented by all computers needing compact code. And it is extremely unlikely that 100% of the 24 proposed extended RVC instructions will prove to be worthwhile.
- We are not discussing whether or not the proposed operation is supported in RISC-V; we are debating whether there should be a 16-bit version in addition to the 32-bit version. Thus, the downside of omitting a worthy extended RVC instruction is small.
- The main purpose of RISC-V Foundation is to have a path to evolve the RISC-V ISA once we have evidence that it needs to change. Why not rely on the evidence-based process to propose "RVC2" rather than guessing in 2015 what the right choice should be? As Donald Knuth warned in his 1974 Turing Award lecture:
  "...premature optimization is the root of all evil."

We welcome your comments on the advisability of including the extended RVC instructions, so please send them to the `isa-dev@lists.riscv.org` mailing list.

[1]The only exceptions are C.ADDI4SPN and the 3 RV32I instructions that read the upper 32 bits of a 64-bit counter: Read Cycle Upper Half (RDCYCLEH), Read Time Upper Half (RDTIMEH), Read Instructions Retired Upper Half(RDINSTRETH). In RV64I or RV128I, the destination register is wide enough to read the whole counter, so read high instructions are superfluous. These three instructions are important but rarely used.

Bibliography

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