RISC-V 压缩指令集手册

版本 1.9

警告！这个规范的初稿在成为标准之前，可能会被修改，因此基于此规范初稿的实现，可能与未来的标准规范并不相符。

（翻译：要你命 3000@EETOP 翻译版本 1.0）

Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanović
CS Division, EECS Department, University of California, Berkeley
{waterman|yunsup|pattrsn|krste}@eecs.berkeley.edu
2015 年 11 月 5 日

该文档同时也是 UCB/EECS-2015-209 技术报告

1.1 介绍

本文档是从 RISC-V 用户级 ISA 规范节录出来的，用于描述 RISC-V 标准压缩指令集扩展的当前初稿，标准压缩指令集扩展，被命名为“C”，通过对常用操作加入短的 16 位指令编码，减少了静态和动态代码大小。这个“C”扩展可以添加到任何基本的 ISA 上（RV32、RV64、RV128），我们使用术语 RVC 来指明这种情形。典型的，程序中大约 50%~60%的 RISC-V 指令可以被 RVC 指令代替，导致大约 25%~30%代码大小的减少。

我们相信这个初稿将和最终的 RV32C 和 RV64C 设计相接近（看起来现在形成 RV128C 并不成熟），但我们仍然需要一轮或者多轮的评价，因此定名为 1.9 版本。请将您的评价发送到 isa-dev@lists.riscv.org 的邮件列表的 isa-dev。

1.2 概述

RVC 使用一种简单的压缩方案，以便在下列情形时，提供更短的 16 位版本的 32 位 RISC-V 指令：

- 立即数或者地址偏移量较小时
- 其中一个寄存器是零寄存器（x0）、ABI 链接寄存器（x1）或者 ABI 栈寄存器（x2）
- 目标寄存器和第一个源寄存器相同
- 最常见情况下使用了 8 个寄存器

C 扩展与其它所有标准扩展兼容。C 扩展允许 16 位指令可以自由地和 32 位指令混合执行，并运行 32 位指令可以在任何 16 位边界开始。（***译者注：一般情况下，32 位指令必须“天然地”对齐到 32 位存储器地址边界上，否则会导致非对齐存储器访问异常。***）

在原来的 32 位指令上面去掉 32 位对齐约束要求，可以大幅度提高代码密度。

压缩指令编码在绝大多数情况下，在 RV32C、RV64C 和 RV128C 下都是一样的，但如表 0.3 所示，少数操作码依据基本 ISA 的宽度有不同的用途。例如，更宽地址空间的 RV64C 和 RV128C 变种，需要额外的操作码来完成压缩 load 和 store 64 位整数值，而 RV32C 使用与单精度浮点值一样的操作码来进行压缩 load 和 store。类似的，RV128C 需要额外的操作码来完成压缩 load 和 store 128 位整数值，而在 RV32C 和 RV64C 中，它们使用与双精度浮点值一样的操作码来进行压缩 load 和 store。如果要实现标准 C 扩展，必须提供相应的压缩浮点 load 和 store 指令，而不管相关的标准浮点扩展（F 和/或 D 扩展）是否实现。另外，RV32C 包含一条压缩跳转和链接指令，以压缩短范围的子过程调用，同样的操作码被用于 RV64C 和 RV128C 的压缩 ADDIW 指令。

双精度 load 和 store 是静态和动态指令的重要部分，因此想到要把它们加入到 RV32C 和 RV64C 编码中。

虽然对于当前支持的 ABI 编译出来的基准测试程序（benchmark）来说，单精度 load 和 store 并不十分重要，但是对于那些在硬件上仅支持单精度浮点

单元、ABI 仅支持单精度浮点数的微控制器来说，在基准程序上，单精度 load 和 store 的使用频度至少和双精度 load 和 store 相同。因此，想到在 RV32C 中提供这些指令的压缩支持。

对于微控制器来说，短范围子过程调用在小的二进制代码中非常常见，因此想到在 RV32C 中加入它们。

虽然对不同的基本寄存器宽度、不同的用途，重用操作码会增加一定的文档复杂度，但是它对实现的影响非常小，即使是对同时支持多个基本 ISA 寄存器宽度也是如此。压缩浮点 load 和 store 指令使用了与更宽整数 load 和 store 相同的指令格式、相同的寄存器区分符。

RVC 是在这样的约束下设计的，即每一条 RVC 指令被扩展成在基本 ISA（RV32I/E、RV64I 或者 RV128I）或者 F、D 标准扩展中的一条 32 位指令。采用这条约束，有如下一些好处：

- 硬件设计可以在译码时简单地扩展 RVC 指令，简化了验证，并使得对现有微体系结构的改动最小化。
- 编译器可以不处理 RVC 扩展部分，留到汇编器和链接器来进行代码压缩，虽然一个压缩敏感的编译器通常可以生成更好代码。

我们感到通过在 C 和 IFD 指令之间进行简单的一对一映射，得到的多种复杂度减少，其远远超过通过增加一些仅仅支持 C 扩展的指令以获得稍微高一些的编码密度而带来的收益，也远高于允许将多条 IFD 指令编码入一条 C 指令而带来的收益。

值得重视的是，C 扩展并不是作为一个单独的 ISA 而被设计的，意味着它需要与一个基本 ISA 一块使用。

变长指令集已经被用来提高代码密度使用很长时间了。例如，在 1950 后期研发的 IBM Stretch[2]使用了一个具有 32 位和 64 位指令的 ISA，其中有些 32 位指令是 64 位指令的压缩版本。Stretch 也使用了这样一个概念，就是在一些较短的指令格式中，限制了可寻址的寄存器集合；具有短分支指令仅能引用索引寄存器中的一个。后来的 IBM 360 体系结构[1]支持一种简单的变长指令编码，支持 16 位、32 位或者 48 位指令格式。

在 1963 年，CDC 发布了 Cray 设计的 CDC 6600[3]，它是 RISC 的前辈，引入了一个大量寄存器的 load-store 体系结构，其指令长度 15 位和 30 位两种。后来的 Cray-1 设计使用了一种非常相似的指令格式，它采用了 16 位和 32 位指令长度。

自 1980 以来的早期 RISC ISA，都是选择性能而不是代码大小，这在工作站环境中是情理之中的，但对嵌入式系统并不合理。因此，ARM 和 MIPS 在后续的 ISA 版本中，都通过在标准 32 位指令集之外提供 16 位指令集，来达到更小的代码大小。压缩过后的 RISC ISA 相对于它们的完全版本，可以减少大约 25%~30%的代码大小，生成的代码远远小于 80x86 生成的代码。这个结果令一些人感到震惊，因为他们认为一个变长 CISC ISA 产生的代码应当比仅提供 16 位和 32 位格式的 RISC ISA 产生的代码要小。

由于原来的RISC ISA 并没有为这些计划外的压缩指令留有足够的操作码空间，因此它们被当做一个新的 ISA 进行开发。这意味着编译器需要为单独的压

缩 ISA 使用不同的代码生成器。第一个压缩 RISC ISA 扩展（就是 ARM Thumb 和 MIPS16）仅仅使用了一种固定 16 位指令长度，这在静态代码大小上得到了很好的减少，但是导致动态指令数目的增加，这也导致了与原始固定 32 位长度指令相比，性能的下降。这导致了 16 位和 32 位指令长度混合的第二代压缩 RISC ISA（就是 ARM Thumb2、microMIPS、PowerPC VLE）的研发，于是可以获得与纯 32 位指令相似的性能，同时有巨大的代码大小减少。不幸的是，这些不同代次的压缩 ISA 彼此之间、与原始为压缩 ISA 之间，并不兼容，导致了文档、实现和软件工具上的巨大复杂性。

在通用 64 位 ISA 中，当前只有 PowerPC 和 microMIPS 支持压缩指令格式。令人感到惊讶的是，考虑到静态代码大小和动态取指带宽是重要的指标，当前移动平台上最为流行的 64 位 ISA（ARM v8）并没有包含一个压缩指令格式。虽然对于较大的系统来说，静态代码大小并不是一个主要关心的因素，但是在服务器运行商业负载（通常具有一个大的指令工作集）的时候，取指带宽可能会成为一个主要的瓶颈。

*得益于 25 年以来的经验，RISC-V 被设计成从外部支持压缩指令，为 RVC 保留了足够的操作码空间，使得它可以在基本 ISA 之上（与其它的标准扩展一起）作为一个简单的扩展加入。RVC 的理念在于为嵌入式应用程序减小代码大小，为所有应用程序提高性能和能耗效率，因为可获得更少的指令 cache 缺失（**译者注：指令长度缩小，可使得一个 Cache line 保存更多的指令，或者意味着在同样大小的指令 Cache 中容纳更多的指令**）。Waterman 指出 RVC 少取了 25%~30%的指令位，减少了 20~25%的指令 Cache 缺失，或者等效于指令 Cache 大小翻倍同样的性能[4]。*

1.3 压缩指令格式

表 0.1：压缩 16 位 RVC 指令格式给出了 8 种压缩指令格式。CR、CI 和 CSS 格式可以使用任何的 32 个 RVI 寄存器，但是 CIW、CL、CS 和 CB 被限制只能使用所有 32 个寄存器中的 8 个。表 0.2：CIW、CL、CS 和 CB 格式中 rs1’、rs2’和 rd’字段三位指向的寄存器给出了这些常用的寄存器，对应于 x8 到 x15。注意到有一个单独版本的 load 和 store 指令，将栈指针作为基地址寄存器使用，这是因为保存到栈和从栈恢复太常见了，并且它们使用 CI 和 CSS 格式，以便能够访问到所有 32 个数据寄存器。CIW 格式为 ADDI4SPN 指令提供了一个 8 位的立即数。

RISC-V 的 ABI 已经做了修改，将常用寄存器映射到寄存器 x8-x15。这通过提供一个连续的、自然对齐的寄存器编号，将简化解压缩的译码器，并且与 RV32E 子集基本规范相兼容，在那里只有 16 个整数寄存器。

压缩的、基于寄存器的浮点 load 和 store 指令也分别使用了 CL 和 CS 格式，将 8 个寄存器映射到 f8 到 f15。

标准 RISC-V 调用规范，将最常用的浮点寄存器映射到 f8-f15，这将允许与整数寄存器编号相同的寄存器解压缩译码。

这些格式被设计成将两个源寄存器区分符放在所有指令中的同一个地方，因此可以去掉目的寄存器字段。如果存在完整的 5 位目的寄存器区分符，它与 32 位 RISC-V 编码中的位置是一样的。当立即数是符号扩展的时候，符号扩展总是从第 12 位开始。立即数字段被打乱了，就如同在基本规范中一样，以便减少用于立即数的多路选择器数量（immediate mux）。

指令格式中的立即数字段是被打乱的而不是按顺序存放的，这是为了保证在每一条指令中尽可能让尽量多的位处在相同的位置，这将简化实现。例如，立即数的位 17-10，总是来自指令的相同位位置。5 个其他的立即数位（5、4、3、2、1）只有两个指令位的来源，而 4 个其他的立即数位（9、7、6、2）有 3 个指令位的来源，而 1 个（第 8 位）有四个来源。

对于许多 RVC 指令来说，不允许 0 立即数，而且 x0 不是一个有效的 5 位寄存器区分符。这个限制，为其他需要较少操作数位的指令，释放了编码空间。

格式	含义	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	寄存器	funct4		rd/rs1		rs2		op
CI	立即数	funct3	imm	rd/rs1		imm		op
CSS	栈相关 store	funct3	imm			rs2		op
CIW	宽立即数	funct3	imm				rd’	op
CL	Load	funct3	imm		rs1’	imm	rd’	op
CS	Store	funct3	imm		rs1’	imm	rd’	op
CB	分支	funct3	offset		rs1’	offset		op
CJ	跳转	funct3	jump target					op

表 0.1：压缩 16 位 RVC 指令格式

RVC 寄存器编号	000	001	010	011	100	101	110	111
整数寄存器编号	x8	x9	x10	x11	x12	x13	x14	x15
整数寄存器 ABI 名字	s0	s1	a0	a1	a2	a3	a4	a5
浮点寄存器编号	f8	f9	f10	f11	f12	f13	f14	f15
浮点寄存器 ABI 名字	fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

表 0.2：CIW、CL、CS 和 CB 格式中 rs1’、rs2’和 rd’字段三位指向的寄存器

1.4 Load 和 store 指令

为了增加 16 位指令能够访问的范围，使用零扩展立即数的数据传输指令，其数据值的大小被放大了多倍：对字×4、对双字×8、对四字×16。

RVC 提供了两种类型的 load 和 store。一种使用 ABI 栈指针 x2 作为基址寄存器，并可定位到任何数据寄存器。另外一种可以引用 8 个基址寄存器之一，并引用 8 个数据寄存器之一。

基于栈指针的 load 和 store

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

这些指令使用 CI 格式。

C.LWSP 指令将一个 32 位数值从存储器读入寄存器 rd 中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上栈指针 x2 形成的。它被扩展为 lw rd, offset[7:2](x2)指令。

C.LDSP 是一条 RV64C/RV128C 仅有指令，它将一个 64 位数值从存储器读入寄存器 rd 中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上栈指针 x2 形成的。它被扩展为 ld rd, offset[8:3](x2)指令。

C.LQSP 指令是一条 RV128C 仅有指令，它将一个 128 位数值从存储器读入寄存器 rd 中。其有效地址的计算是通过将零扩展的偏移量×16，然后加上栈指针 x2 形成的。它被扩展为 lq rd, offset[9:4](x2)指令。

C.FLWSP 是一条 RV32FC 仅有指令，它将一个单精度浮点数值从存储器读入浮点寄存器 rd 中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上栈指针 x2 形成的。它被扩展为 flw rd, offset[7:2](x2)指令。

C.FLDSP 是一条 RV32DC/RV64DC 仅有指令，它将一个双精度浮点数值从存储器读入浮点寄存器 rd 中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上栈指针 x2 形成的。它被扩展为 fld rd, offset[8:3](x2)指令。

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

这些指令使用 CSS 格式。

C.SWSP 指令将寄存器 rs2 中的 32 位值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上栈指针 x2 形成的。它被扩展为 sw rs2, offset[7:2](x2)指令。

C.SDSP 是一条 RV64C/RV128C 仅有指令，它将寄存器 rs2 中的 64 位值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上栈指针 x2 形成的。它被扩展为 sd rs2, offset[8:3](x2)指令。

C.SQSP 是一条 RV128C 仅有指令，它将寄存器 rs2 中的 128 位值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×16，然后加上栈指针 x2 形成的。它被扩展为 sq rs2, offset[9:4](x2)指令。

C.FSWSP 是一条 RV32FC 仅有指令，它将浮点寄存器 rs2 中的单精度浮点数值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上栈指针 x2 形成的。它被扩展为 fsw rs2, offset[7:2](x2)指令。

C.FSDSP 是一条 RV32DC/RV64DC 仅有指令，它将浮点寄存器 rs2 中的双精度浮点数值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上栈指针 x2 形成的。它被扩展为 fsd rs2, offset[8:3](x2)指令。

基于寄存器的 Load 和 store

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	偏移量[5:3]	基址	偏移量[2|6]	dest	C0
C.LD	偏移量[5:3]	基址	偏移量[7:6]	dest	C0
C.LQ	偏移量[5|4|8]	基址	偏移量[7:6]	dest	C0
C.FLW	偏移量[5:3]	基址	偏移量[2|6]	dest	C0
C.FLD	偏移量[5:3]	基址	偏移量[7:6]	dest	C0

这些指令使用 CL 格式。

C.LW 指令将一个 32 位数值从存储器读入寄存器 rd′中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上寄存器 rs1′中的基址形成的。它被扩展为 lw rd′, offset[6:2](rs1′)指令。

C.LD 是一条 RV64C/RV128C 仅有指令，它将一个 64 位数值从存储器读入寄存器 rd′中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上寄存器 rs1′中的基址形成的。它被扩展为 ld rd′, offset[7:3](rs1′)指令。

C.LQ 是一条 RV128C 仅有指令，它将一个 128 位数值从存储器读入寄存器 rd′中。其有效地址的计算是通过将零扩展的偏移量×16，然后加上寄存器 rs1′中的基址形成的。它被扩展为 lq rd′, offset[8:4](rs1′)指令。

C.FLW 是一条 RV32FC 仅有指令，它将一个单精度浮点数值从存储器读入浮点寄存器 rd′中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上寄存器 rs1′中的基址形成的。它被扩展为 flw rd′, offset[6:2](rs1′)指令。

C.FLD 是一条 RV32DC/RV64DC 仅有指令，它将一个双精度浮点数值从存储器读入浮点寄存器 rd′中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上寄存器 rs1′中的基址形成的。它被扩展为 fld rd′, offset[7:3](rs1′)指令。

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	偏移量[5:3]	基址	偏移量[2|6]	src	C0
C.SD	偏移量[5:3]	基址	偏移量[7:6]	src	C0
C.SQ	偏移量[5|4|8]	基址	偏移量[7:6]	src	C0
C.FSW	偏移量[5:3]	基址	偏移量[2|6]	src	C0
C.FSD	偏移量[5:3]	基址	偏移量[7:6]	src	C0

这些指令使用 CS 格式。

C.SW 指令将寄存器 rs2′中的 32 位值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上寄存器 rs1′中的基址形成的。它被扩展为 sw rs2′, offset[6:2](rs1′)指令。

C.SD 是一条 RV64C/RV128C 仅有指令，它将寄存器 rs2′中的 64 位值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上寄存器 rs1′中的基址形成的。它被扩展为 sd rs2′, offset[7:3](rs1′)指令。

C.SQ 是一条 RV128C 仅有指令，它将寄存器 rs2′中的 128 位值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×16，然后加上寄存器 rs1′中的基址形成的。它被扩展为 sq rs2′, offset[8:4](rs1′)指令。

C.FSW 是一条 RV32FC 仅有指令，它将浮点寄存器 rs2′中的单精度浮点数值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上寄存器 rs1′中的基址形成的。它被扩展为 fsw rs2′, offset[6:2](rs1′)指令。

C.FSD 是一条 RV32DC/RV64DC 仅有指令，它将浮点寄存器 rs2′中的双精度浮点数值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上寄存器 rs1′中的基址形成的。它被扩展为 fsd rs2′, offset[7:3](rs1′)指令。

1.5 控制转移指令

RVC 提供了无条件跳转指令和条件分支指令。如同基本 RVI 指令一样，所有 RVC 控制转移指令的偏移量都是 2 字节的倍数。

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

这些指令使用 CJ 格式。

C.J 指令执行一个无条件控制转移。偏移量被符号扩展后，与 pc 相加形成跳转目标地址。C.J 指令因此可以在±2KB 范围内进行跳转。C.J 指令被扩展为 jal x0, offset[11:1]。

C.JAL 指令是一条 RV32C 仅有指令，它执行与 C.J 指令相同的操作，但是它还将在跳转指令后的指令地址（pc+2）写入到链接寄存器 x1 中。C.JAL 指令被扩展为 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

这些指令使用 CR 格式。

C.JR（jump register）指令执行一个无条件控制转移到寄存器 rs1 的地址。C.JR 指令被扩展为 jalr x0, rs1, 0。

C.JALR（jump and link register）指令执行与 C.JR 指令相同的操作，但是它还将在跳转指令后的指令地址（pc+2）写入到链接寄存器 x1 中。C.JALR 指令被扩展为 jalr x1, rs1, 0。

严格来说，C.JALR 指令并没有被精确地扩展为基本 RVI 指令，因为那个被加到 pc 上以形成链接地址的值是 2，而不是基本 ISA 中的 4，但是同时支持偏移量 2 个字节和 4 个字节，只对微体系结构产生微小的影响。

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

这些指令使用 CB 格式。

C.BEQZ 指令执行条件控制转移。偏移量被符号扩展后，与 pc 相加形成跳转目标地址。C.BEQZ 指令因此可以在±256B 范围内进行跳转。如果寄存器 rs1’的值是 0，则 C.BEQZ 指令产生控制转移（take the branch）。这条指令被扩展为 beq rs1’, x0, offset[8:1]。

C.BNEZ 指令定义相似，只是当寄存器 rs1’的值是非 0 值，则指令产生控制转移（take the branch）。这条指令被扩展为 bne rs1’, x0, offset[8:1]。

1.6 整数计算指令

RVC 提供了一些用于整数算术和常数生成的指令。

整数常数-生成指令

两条常数-生成指令都使用 CI 格式，并且可以以任何整数寄存器为目标。

15 13	12	11 7	6 2	1 0
funct3	imm	rd	imm[4:0]	op
3	1	5	5	2
C.LI	立即数[5]	dest≠0	立即数[4:0]	C1
C.LUI	非零立即数[17]	dest≠{0,2}	非零立即数[16:12]	C1

C.LI 指令将符号扩展的 6 位立即数 ***imm***，写入寄存器 rd 中。C.LI 指令仅在 rd≠x0 时才是有效的。C.LI 指令被扩展为 addi rd, x0, imm[5:0]。

C.LUI 指令将非零的 6 位立即数写入到目标寄存器的 17-12 位，并将目标寄存器的低 12 位清零，然后将第 17 位符号扩展到整个目标寄存器的高位部分。C.LUI 寄存器仅在 rd≠{x0, x2} 且立即数不等于 0 时才是有效的。C.LUI 指令被扩展为 lui rd, nzimm[17:12]。

整数寄存器-立即数指令

这些整数寄存器-立即数指令都使用 CI 格式，并在认为非 x0 整数寄存器和一个 6 位立即数之间进行操作。立即数不能为 0。

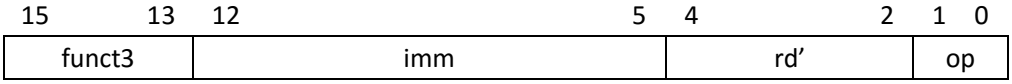
15 13	12	11 7	6 2	1 0
funct3	imm[5]	rd/rs1	imm[4:0]	op
3	1	5	5	2
C.ADDI	非零立即数[5]	dest	非零立即数[4:0]	C1
C.ADDIW	立即数[17]	dest≠0	立即数[4:0]	C1
C.ADDI16SP	非零立即数[9]	2	非零立即数[4|6|8:7|5]	C1

C.ADDI 指令将非零的、符号扩展的 6 位立即数加到寄存器 rd 的值上，将结果写入 rd。C.ADDI 指令被扩展为 addi rd, rd, nzimm[5:0]。

C.ADDIW 指令是一条 RV64C/RV128C 仅有的指令，它执行相同的计算，但是生成一个 32 位的结果，然后符号扩展结果到 64 位。C.ADDIW 指令被扩展为 addiw rd, rd, imm[5:0]。对 C.ADDIW 指令而言，立即数可以是 0，这对应于 sext.w rd。

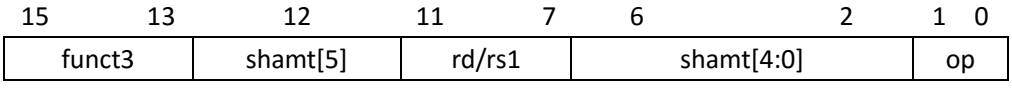
C.ADDI16SP 指令的操作码与 C.LUI 指令相同，但是使用 x2 作为目标寄存器。C.ADDI16SP 指令将一个非零的、符号扩展的 6 位立即数加到栈指针寄存器（sp=x2）上，此处立即数被放大 16 倍，其范围为（-512,496）。C.ADDI16SP 指令用于在过程的头部和尾部对栈指针进行调整。它被扩展为 addi x2, x2, nzimm[9:4]。

在标准 RISC-V 调用约定中，栈指针 sp 总是 16 字节对齐的。



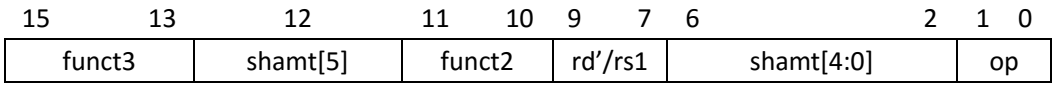
C.ADDI4SPN	非零立即数[5:4|9:6|2|3]	dest	C0

C.ADDI4SPN 指令是一条 CIW 格式的、RV32C/RV64C 仅有的指令，它将一个零扩展的、非零立即数，乘以 4，然后加到栈指针 x2 上，并将结果写入 rd’。这条指令用于产生指向分配在栈中的变量的指针，它被扩展为 addi rd’, x2, zimm[9:2]。



C.SLLI	移位次数[5]	dest≠0	移位次数[4:0]	C2

C.SLLI 指令是一条 CI 格式的指令，它对寄存器 rd 中的数值进行逻辑左移操作，并将结果写入 rd。移位次数被编码到 ***shamt*** 字段，此处对 RV32C，***shamt[5]***必须为 0。对于 RV32C/RV64C，移位次数必须为非零值。对于 RV128C，一个 shamt 为 0，编码为移位 64 次。C.SLLI 指令被扩展为 slli rd, rd, shamt[5:0]，除了对于 RV128C 且 shamt=0，则被扩展为 slli rd, rd, 64。



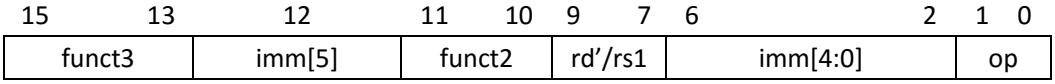
C.SRLI	移位次数[5]	C.SRLI	dest	移位次数[4:0]	C1
C.SRAI	移位次数[5]	C.SRAI	dest	移位次数[4:0]	C1

C.SRLI 指令是一条 CB 格式的指令，它对寄存器 rd’中的数值进行逻辑右移操作，并将结果写入 rd’。移位次数被编码到 ***shamt*** 字段，此处对 RV32C，***shamt[5]***必须为 0。对于 RV32C/RV64C，移位次数必须为非零值。对于 RV128C，一个 shamt 为 0，编码为移位 64 次。而且对于 RV128C，移位次数是符号扩展的，因此合法的移位次数是 1-31、64、96-127。C.SRLI 指令被扩展为 srli rd’, rd’, shamt[5:0]，除了对于 RV128C 且 shamt=0，则被扩展为 srli rd’, rd’, 64。

C.SRAI 指令与 C.SRLI 指令相似，不过它执行一个算术右移操作。C.SRAI 指令被扩展为 srai rd’, rd’, shamt[5:0]。

左移通常比右移更为常用，因为左移被频繁地用于对地址值进行放大操作。因此右移被分配了较小的编码空间，并且处于一个其他立即数都是符号扩展的编码区域中。对于 RV128，我们决定使得 6 位移位次数值也是符号扩展的。除了减少硬件译码复杂度之外，我们相信右移 96-127 次要比 64-95 更为有用，这可以用于从 128 位地址指针的高部分提取标签（tag）。我们注意到 RV128C

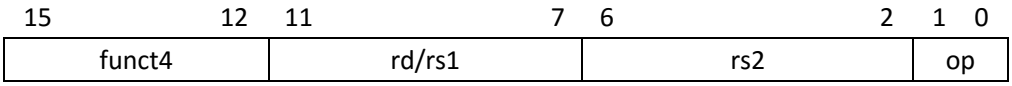
并不像 RV32C 和 RV64C 那样已经确定下来，以允许对使用 128 位地址空间的典型代码进行评估。



15 13	12	11 10	9 7	6 2	1 0
funct3	imm[5]	funct2	rd’/rs1	imm[4:0]	op
3	1	2	3	5	2
C.ANDI	立即数[5]	C.ANDI	dest	立即数[4:0]	C1

C.ANDI 指令是一条 CB 格式的指令，它在寄存器 rd’的值和一个符号扩展的 6 位立即数之间进行按位 AND 运算，并将结果写入到 rd’中。C.ANDI 指令被扩展为 andi rd’, rd’, imm[5:0]。

整数寄存器-寄存器指令

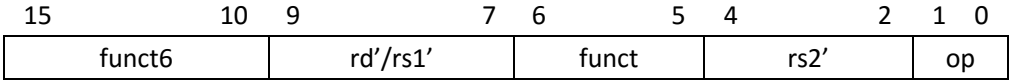


15 12	11 7	6 2	1 0
funct4	rd/rs1	rs2	op
4	5	5	2
C.MV	dest≠0	src≠0	C0
C.ADD	dest≠0	src≠0	C0

这些指令使用 CB 格式。

C.MV 指令将寄存器 rs2 的值复制到寄存器 rd 中。C.MV 指令被扩展为 add rd, x0, rs2

C.ADD 指令将寄存器 rd 的值与寄存器 rs2 的值相加，并将结果写入到寄存器 rd 中。C.ADD 指令被扩展为 add rd, rd, rs2。



15 10	9 7	6 5	4 2	1 0
funct6	rd’/rs1’	funct	rs2’	op
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

这些指令使用 CS 格式。

C.AND 指令在寄存器 rd’和 rs2’之间执行按位 AND 操作，并将结果写入寄存器 rd’。C.AND 指令被扩展为 and rd’, rd’ rs2’。

C.OR 指令在寄存器 rd’和 rs2’之间执行按位 OR 操作，并将结果写入寄存器 rd’。C.OR 指

令被扩展为 or rd′, rd′ rs2′。

C.XOR 指令在寄存器 rd′和 rs2′之间执行按位 XOR 操作，并将结果写入寄存器 rd′。C.XOR 指令被扩展为 xor rd′, rd′ rs2′。

C.SUB 指令将寄存器 rd′的值减去 rs2′的值，并将结果写入寄存器 rd′。C.SUB 指令被扩展为 sub rd′, rd′ rs2′。

C.ADDW 是一条 RV64C/RV128C 仅有的指令，它将寄存器 rd′的值加上 rs2′的值，将结果的低 32 位进行符号扩展，再写入 rd′中。C.ADDW 指令被扩展为 addw rd′, rd′, rs2′。

C.SUBW 是一条 RV64C/RV128C 仅有的指令，它将寄存器 rd′的值减去 rs2′的值，将结果的低 32 位进行符号扩展，再写入 rd′中。C.SUBW 指令被扩展为 subw rd′, rd′, rs2′。

这组的 6 条指令，每条指令并没有提供太多的好处，但是也没有占用很多的编码空间，并且实现起来也直截了当，作为一组指令是在静态和动态压缩中提供了一定的提高。

预定义非法指令

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

一条所有位都是 0 的 16 位指令，被永久的保留为一条非法指令。

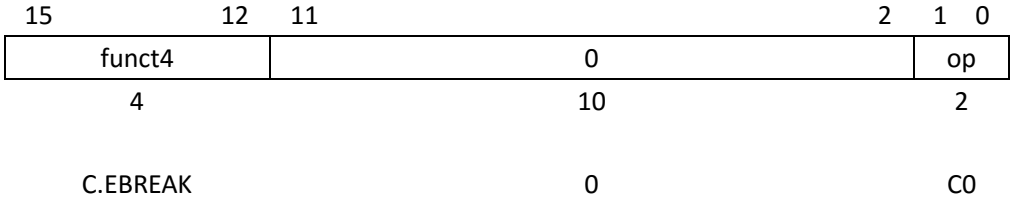
*我们将全零指令保留为非法指令，以帮助捕获试图执行被零填充的或者不存在的存储器空间，产生自陷。全零值不应当被任何非标准扩展重新定义。类似的，我们保留全 1 指令（在 RISC-V 变长编码方式中，对应于非常长的指令）为非法指令，以捕获在不存在存储器区域的另外一种常见值。（**译者注：如果处理器试图访问不存在的存储器空间地址，外部硬件常规做法是总是返回全 0，或者返回全 1**）*

NOP 指令

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 指令是一条 CI 格式指令，它不改变任何用户可见状态，除了推进 pc 之外。C.NOP 指令被编码为 c.addi x0, 0 并且被扩展为 addi x0, x0, 0。

断点指令



调试器可以使用 C.EBREAK 指令，它将被扩展为 ebreak 指令，并导致控制被转移回到调试环境。C.EBREAK 指令的操作码与 C.ADD 指令的操作码相同，但是其 rd 和 rs2 都是 0，因此也可以使用 CR 格式。

1.7 优化寄存器保存/恢复代码大小

在函数入口、出口处的寄存器保存/恢复代码占据了静态代码大小的很重要组成部分。RVC 中的基于栈指针的 load 和 store 指令可以有效地减少一半的保存恢复静态代码大小，同时通过减少动态指令带宽，提高了执行性能。

标准 RISC-V 软件工具链提供了另外一种更进一步减少保存/恢复静态代码大小的方法，这是以降低性能来交换的。与将寄存器保存/恢复代码嵌入到每个函数中不同，寄存器保存代码被一条跳转并链接指令代替，它将调用一个子过程将寄存器复制到栈中，然后再返回函数。寄存器恢复代码被一条跳转指令代替，它将跳转到一个子过程从栈中恢复寄存器，然后再跳转到恢复的返回地址。

图 0.1 给出了当直接应用到 SPEC CPU 2006 基准测试程序的所有函数上时，这些子过程对静态代码和动态指令数目的影响。平均来说，代码大小减少了 4%，而动态指令数目增大了 3%。

当将-Os（减少代码大小）标志传递给 gcc 时，嵌入函数的保存/恢复代码被替换成调用保存/恢复子过程。

在其他 ISA 中，另外一种减少保存/恢复代码大小的机制是 load-multiple、store-multiple 指令。我们考虑过将它们加入到 RISC-V 中，但是注意到这些指令由下面这些不足：

- *这些指令导致复杂的处理器实现。*
- *对于虚拟存储器系统来说，一些数据访问可能处在物理存储器中，而另外一些数据访问不在，这就需要一种新的机制，以重启已经部分执*

*行了的指令。（**译者注：比如 LDM r2,r3 指令，假设需要读取到 r2 数据在存储器中，而 r3 不在。那么在处理异常时，需要重新启动 LDM 指令的一部分，因为 r2 上次已经读取了***）

- *与其他的 RVC 指令不同，Load multiple 和 Store multiple 并没有对等的 IFD。*
- *与其他的 RVC 指令不同，编译器必须在生成指令和分配寄存器时，特别注意这些指令，以便最大化它们被按序保存和恢复，因为它们将来会被按序保存和恢复的。*
- *简单的微体系结构实现，将会限制在 load 和 store multiple 指令周围，如何调度其他指令，导致潜在的性能损失。*
- *理想的按顺序寄存器分配可能与为 CIW、CL、CS 和 CB 格式选择的特别寄存器冲突。*

虽然一些体系结构设计师可能会得出不同的结论，但是我们决定去掉 load 和 store multiple 支持，而是使用调用保存/恢复子过程的软件方法，来获得最大限度地代码大小减少。

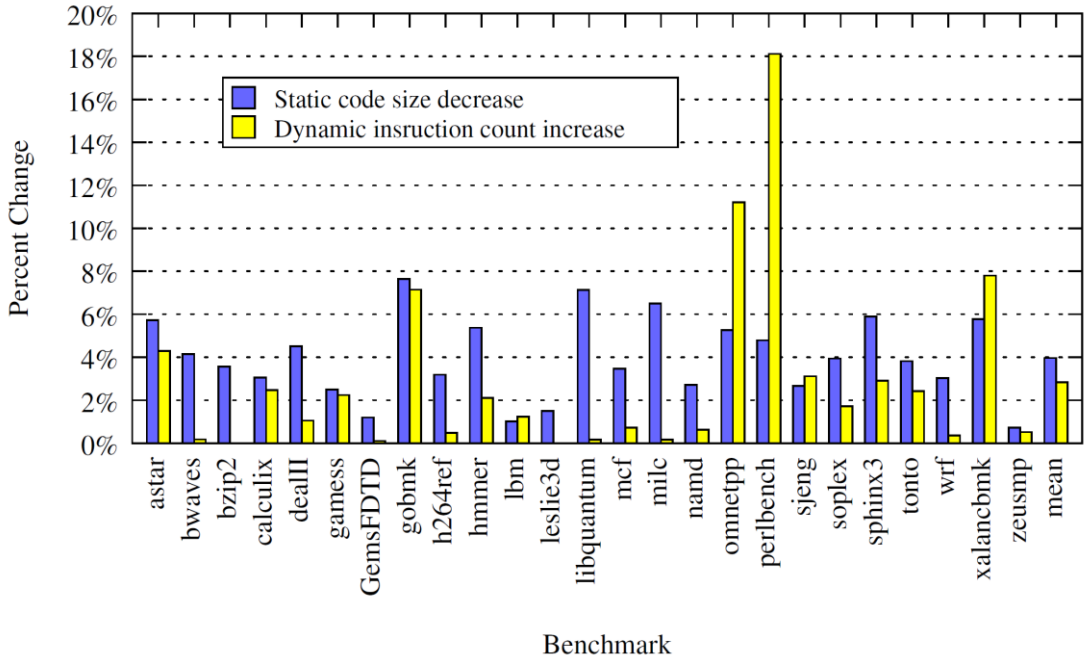


图 0.1：压缩的函数入口、出口处的子过程，对静态代码大小和动态指令数目的影响。

1.8 RVC 指令集列表

表 0.3 给出了 RVC 主要操作码的映射表。对于指令长度超过 16 位的指令，其最低两位都是 1，包括那些处于基本 ISA 中的指令。一些指令仅在某些操作数时是有效的；当无效时，它们要么被标记为 ***RES***，意味着这个操作码被保留给未来的标准扩展；要么被标记为 ***NSE***，意味着这个操作码被保留给未来的非标准扩展；或者被标记为 ***HINT***，意味着这个操作码被保留给未来的微体系结构提示（hint）。在提示没有效果的实现上，标记为 ***HINT*** 的指令必须作为空操作指令执行。

HINT 指令被设计成支持未来增加微体系结构提示，这些提示可能影响性能，但是不能影响体系结构状态。HINT 编码已经被选定，因此简单的实现可以忽略 HINT 编码，并将 HINT 指令作为常规指令执行，不改变体系结构状态。例如，C.ADD 指令的目标寄存器如果是 x0，那么它是一条 HINT 指令，此处 5 位的 rs2 字段编码了 HINT 的细节。然而，一个简单的实现可以简单地把 HINT 当作一条目标是 x0 的加法指令执行，这时将被忽略（即 NOP 指令）。

表 0.4-表 0.6 列出了 RVC 指令。

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	>16 位								

表 0.3：RVC 操作码映射表

15 14 13	12 11 10	9 8	7 6 5	4 3 2	1 0	
000	0			0	00	非法指令
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)

表 0.4：RVC 指令列表，00 部分

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	rs2′	01	Reserved
100	1	11	——	11	rs2′	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

表 0.5：RVC 指令列表，01 部分

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)

表 0.6：RVC 指令列表，10 部分

1.9 指令压缩统计

下面一些表格给出了一些数据，我们使用这些数据来指导选择将什么指令包含到 RVC 中。

表 0.7 列出了标准 RVC 指令，按照使用频度从高到低排序，给出了对静态代码大下，单条指令的贡献，然后运行了总共 3 个实验。对 RV32，RVC 在 Dhrystone 减少了静态代码 24.5%，在 CoreMark 减少了 30.9%。对 RV64，RVC 在 SPECint 减少了静态代码 26.3%，在 SPECfp 减少了 25.8%，在 Linux kernel 减少了 31.1%。

表 0.8 根据典型动态频率对 RVC 指令进行了排序。对 RV32，RVC 在 Dhrystone 减少了取指的动态字节 29.2%，在 CoreMark 减少了 29.3%。对 RV64，RVC 在 SPECint 减少了取指的动态字节 26.9%，在 SPECfp 减少了 22.4%，在启动 Linux kernel 时减少了 26.11%。

指令	RV32GC			RV64GC		MAX
	Dhry-stone	Core-Mark	SPEC 2006	SPEC 2006	Linux Kernel	
C.MV	1.78	5.03	4.06	3.62	5	5.03
C.LWSP	4.51	2.8	2.89	0.49	0.14	4.51
C.LDSP	—	—	—	3.2	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	2.1	1.68	2	0.74	0.62	2.1
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	0.81	0.05	0.12	0.31	0.81
C.BNEZ	0.19	0.53	0.53	0.32	0.8	0.8
C.SD	—	—	—	0.25	0.79	0.79
C.ADDIW	—	—	—	0.77	0.5	0.77
C.JAL	0.38	0.59	0.05	—	—	0.59
C.ADDI4SPN	0.57	0.37	0.45	0.5	0.3	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	0.42	0.2	0.07	0.35	0.42
C.FLD	0	0	0.16	0.39	0	0.39
C.FLDSP	0	0.02	0.2	0.31	0	0.31
C.FSDSP	0.13	0.09	0.15	0.26	0	0.26
C.SUB	0.25	0.09	0.13	0.06	0.11	0.25
C.AND	0	0	0.07	0.03	0.21	0.21
C.FSD	0	0	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	0.1	0.14	0.17
C.ADDW	—	—	—	0.16	0.12	0.16
C.EBREAK	0	0.02	0	0	0.08	0.08
C.FLW	0	0	0.05	—	—	0.05
C.XOR	0	0.04	0.01	0.01	0.03	0.04
C.SUBW	—	—	—	0.04	0.03	0.04
C.FLWSP	0	0	0.03	—	—	0.03
C.FSW	0	0	0.02	—	—	0.02
C.FSWSP	0	0	0.02	—	—	0.02
总计	24.46	30.92	25.78	25.98	25.98	—

表 0.7：按典型静态频率排序的 RVC 指令。表中的数据给出了每条指令在静态代码大小中节约的比例。这个列表是由通过一个压缩汇编器产生的，它对 RISC-V GCC 编译器的输出进行处理。对 RV32GC 使用了 Dhrystone、CoreMark 和 SPEC CPU2006，对 RV64GC 使用了 SPEC CPU2006 和 Linux kernel 3.14.29 版本。表中的横线表示该指令没有在这个地址大小下面的定义。

指令	RV32GC		RV64GC		MAX
	Dhry-stone	Core-Mark	SPEC 2006	Linux Kernel	
C.ADDI	3.7	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	1.7	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	0.2	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	2.48	0.2	0.38	2.48
C.JR	2.07	0.34	0.46	0.42	2.07
C.FLD	0	0	1.63	0	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	1.3	0.1	0.23	1.3
C.ADDIW	—	—	1.26	1.03	1.26
C.SLLI	0.15	1.1	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	0	0.21	0.75	0.75
C.SRAI	0	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	0	0.4	0	0.4
C.ADDI16SP	0.13	0.18	0.28	0.38	0.38
C.FSD	0	0	0.29	0	0.29
C.FSDSP	0	0	0.25	0	0.25
C.ADDW	—	—	0.19	0.04	0.19
C.XOR	0	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	0.1	0.1
C.JALR	0	0.05	0.05	0.03	0.05
C.SUBW	—	—	0.04	0.02	0.04
C.EBREAK	0	0	0	0	0
C.FLW	0	0	—	—	—
C.FLWSP	0	0	—	—	—
C.FSW	0	0	—	—	—
C.FSWSP	0	0	—	—	—
总计	29.18	29.29	24.03	26.11	—

表 0.8：按典型动态频率排序的 RVC 指令。表中的数据给出了每条指令在动态代码大小中节约的比例。这个列表是通过执行来获得的。对 RV32GC 执行了 Dhrystone、CoreMark，对 RV64GC 执行了 SPEC CPU2006，对于 SPEC，我们使用了参考输入集。Linux 启动时间包括引导内核、执行 init 进程、执行 shell 以及 poweroff 命令。

参考文献

[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.