Risc-v 开发

Risc-v 基础知识

北京飞利信科技股份有限公司

2018 年 5 月

飞利信 MCU 芯片遵循的开源指令集 Rv32IMC 标准特点中文介绍

1 开源指令集 Rv32IMC 概述

标准 ISA 扩展是由单个字母构成的名字。例如，基本整数核心的最前面 4 个标准扩展是：“M”表示整数乘法和除法，“A”表示原子性存储器指令，“F”表示单精度浮点指令，“D”表示双精度浮点指令。任何 RISC-V 指令集变种，都可以简洁地通过将基本整数前缀和所包含的扩展连接起来描述。例如，“RV64IMAFD”。RISC-V ISA 标准扩展使用了其他的保留字母，例如“Q”表示四精度浮点，“C”表示 16 位压缩指令格式。

Rv32IMC 是 Risc-v ISA 的重要变体之一，BottleRocket 在 GitHub 进行了 RV32IMC 微处理器的开源实现。

子集	名字
标准通用ISA	
整数	I
整数乘法和除法	M
原子性	A
单精度浮点	F
双精度浮点	D
通用	G=IMAFD
标准用户级扩展	
四精度浮点	Q
十进制浮点	L
16位压缩指令	C
位操作	B
事务存储器	T
打包的SIMD扩展	P
向量扩展	V

该模型的所有变体
RV32IM
RV32IMC
RV32IMAC
RV32G
RV32GC
RV32GCN
RV32E
RV32EC
RV64I
RV64IM
RV64IMC
RV64IMAC
RV64G
RV64GC
RV64GCN

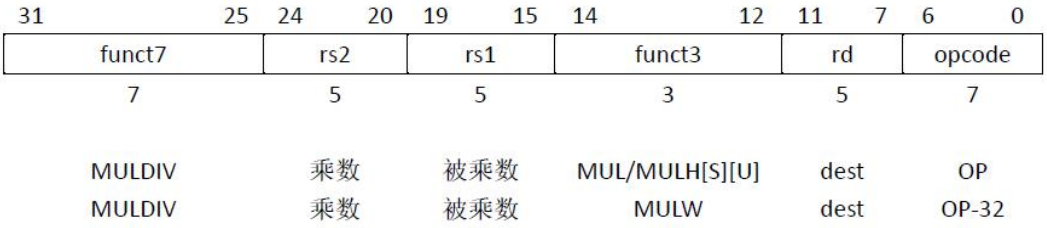
2 基本整数指令“I”

详见第 2 章第一部分。

3 整数乘法除法指令“M”

标准整数乘法和除法的指令扩展，被命名为“M”，并包含针对两个整数寄存器中的数值进行乘法或者除法的指令。

3.1 乘法操作



MUL 指令执行一个 XLEN 位×XLEN 位乘法，并将结果的低 XLEN 位放置到目标寄存器中。MULH、MULHU、MULHSU 执行相同的乘法，分别针对有符号×有符号、无符号×无符号、有符号×无符号乘法，只是将运算结果 2×XLEN 位的高 XLEN 位返回。如果同时需要乘法结果的高位和低位，那么建议的代码顺序为：MULH[[S]U] rdh,rs1,rs2; MUL rdl,rs1,rs2（源寄存器区分符必须按照同样的顺序，并且 rdh 不能是 rs1 或者 rs2）。因此微体系结构可以将这些融合为一个单一的乘法操作，而不是执行两次分开的乘法。

3.2 除法操作

DIV 和 DIVU 指令分别执行有符号、无符号的 XLEN 位整数除以 XLEN 位整数除法操作。REM、REMU 给出了相应除法的余数。如果同时需要商和余数，那么建议的代码顺序为：DIV[U] rdq,rs1,rs2; REM[U] rdr,rs1,rs2（rdq 不能是 rs1 或者 rs2）。因此微体系结构可以将这些融合为一个单一的除法操作，而不是执行两次分开的除法。

除以零和除法溢出的语义如下图所示。除以零，结果商的所有位被置为 1，也即是说，对于无符号除法来说，商是 2XLEN-1，对于有符号除法来说，商是-1。除以零，结果的余数等于被除数。有符号除法溢出仅当用最小的负整数，-2XLEN-1，除以-1 时，才会出现。有符号除法溢出的商等于被除数，余数等于零。无符号除法不会产生溢出。

条件	被除数	除数		DIVU	REMU	DIV	REM
除以0	x	0		2^{XLEN-1}	x	-1	x
溢出（仅有符号）	-2^{XLEN-1}	-1		—	—	-2^{XLEN-1}	0

图 1.8 除以 0 和除法溢出的语义

4 压缩指令“C”

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），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。

4.1 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。（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。）

压缩指令编码在绝大多数情况下都是一样的，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。

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。

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。

4.2 压缩指令格式

图 1.9 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。图 1.10 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。

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。

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）。

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

图 1.9 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

图 1.10 CIW、CL、CS 和 CB 格式中 rs1'、rs2'和 rd'字段三位指向的寄存器

4.3 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。

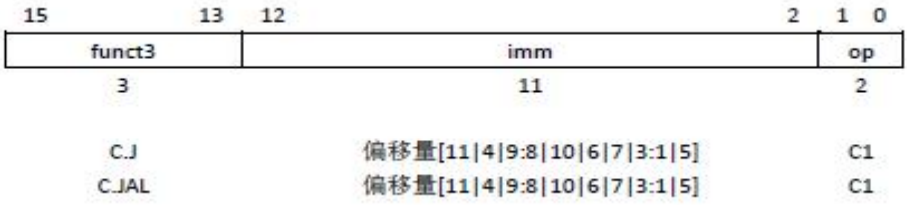
C.SQ 是一条 RV128C 仅有指令，它将寄存器 rs2’中的 128 位值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×16，然后加上寄存器 rs1’中的基址形成的。它被扩展为 sq rs2’, offset[8:4](rs1’)指令。

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。

4.4 控制转移指令

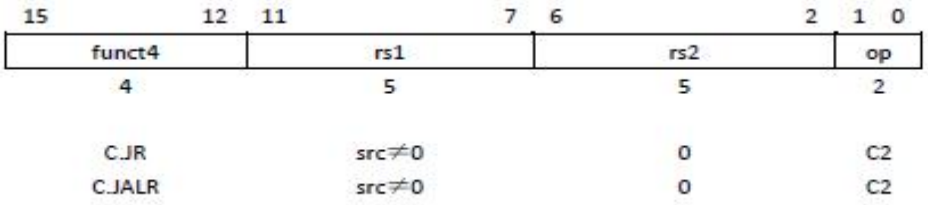
RVC 提供了无条件跳转指令和条件分支指令。如同基本 RVI 指令一样，所有 RVC 控制转移指令的偏移量都是 2 字节的倍数。



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]***。

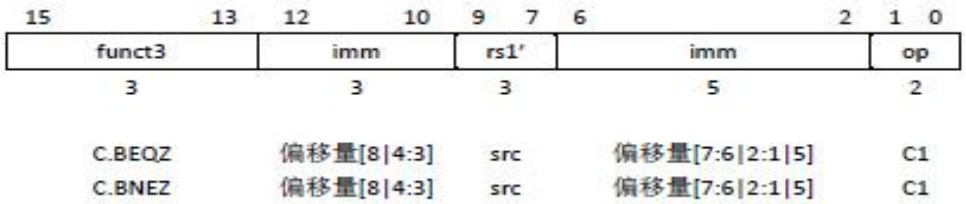
C.JAL 指令是一条 RV32C 仅有指令，它执行与 C.J 指令相同的操作，但是它还将在跳转指令后的指令地址（**pc**+2）写入到链接寄存器 **x1** 中。C.JAL 指令被扩展为 jal ***x1, offset[11:1]***。



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。



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]。

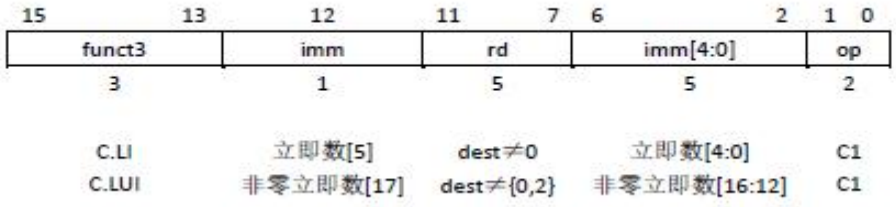
C.BNEZ 指令定义相似，只是当寄存器 rs1'的值是非 0 值，则指令产生控制转移（take the branch）。这条指令被扩展为 bne rs1', x0, offset[8:1]。

4.5 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。



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

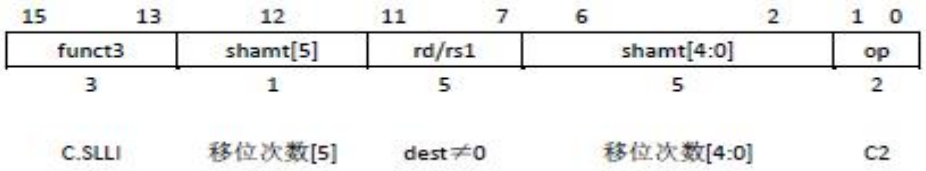
这些整数寄存器-立即数指令都使用 CI 格式，并在认为非 x0 整数寄存器和一个 6 位立即数之间进行操作。立即数不能为 0。

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]。

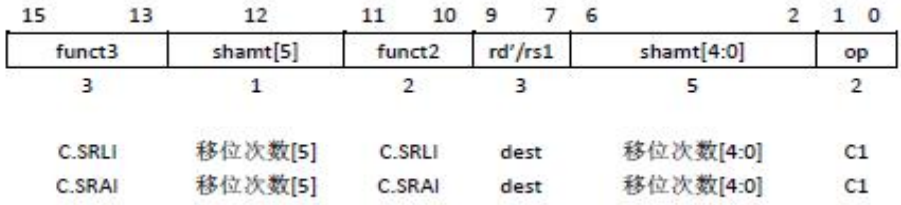
C.ADDIW 指令是一条 RV64C/RV128C 仅有的指令，它执行相同的计算，但是生成一个 32 位的结果，然后符号扩展结果到 64 位。C.ADDIW 指令被扩展为 addiw rd, rd, imm[5:0]。对 C.ADDIW 指令而言，立即数可以是 0，这对应于 sext.w rd。

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]。

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]。



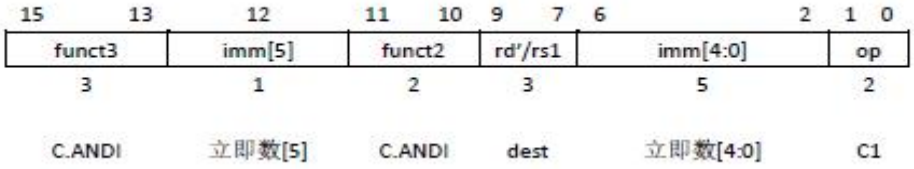
C.SLLI 指令是一条 CI 格式的指令，它对寄存器 rd 中的数值进行逻辑左移操作，并将结果写入 rd。移位次数被编码到 shamt 字段，此处对 RV32C, shamt[5] 必须为 0。对于 RV32C/RV64C，移位次数必须为非零值。对于 RV128C，一个 shamt 为 0，编码为移位 64 次。C.SLLI 指令被扩展为 slli rd, rd, shamt[5:0]，除了对于 RV128C 且 shamt=0，则被扩展为 slli rd, rd, 64。



15–13	12	11–10	9–7	6–2	1–0
funct3	shamt[5]	funct2	rd'/rs1	shamt[4:0]	op
3	1	2	3	5	2
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。

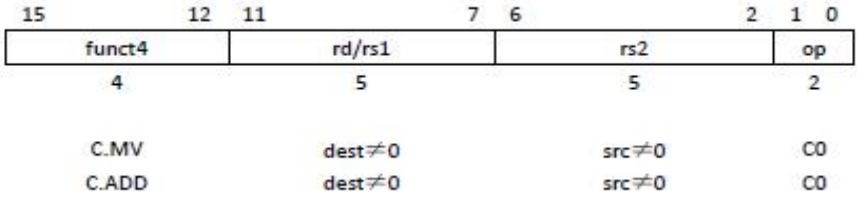
C.SRAI 指令与 C.SRLI 指令相似，不过它执行一个算术右移操作。C.SRAI 指令被扩展为 srai rd', rd', shamt[5:0]。



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

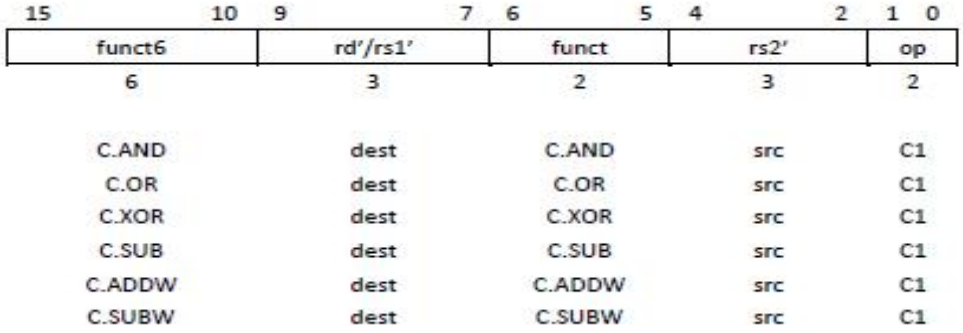


15–12	11–7	6–2	1–0
funct4	rd/rs1	rs2	op
4	5	5	2
C.MV	dest≠0	src≠0	C0
C.ADD	dest≠0	src≠0	C0

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'。

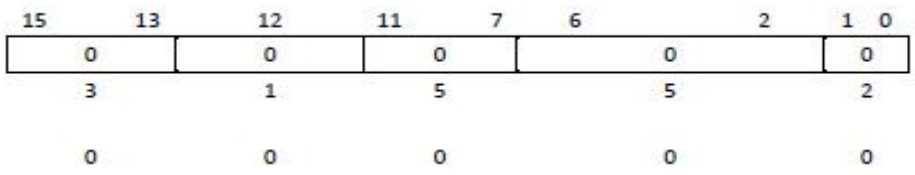
C.XOR 指令在寄存器 rd'和 rs2'之间执行按位 XOR 操作，并将结果写入寄存器 rd'。C.XOR 指令被扩展为 xor rd', rd'rs2'。

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'。

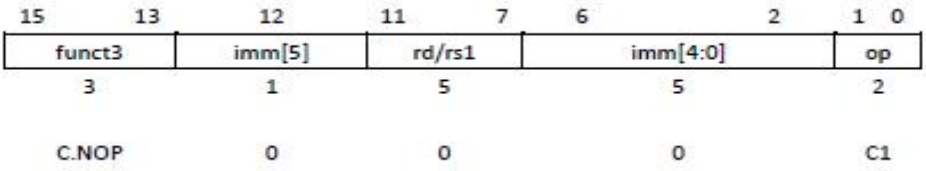
预定义非法指令



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

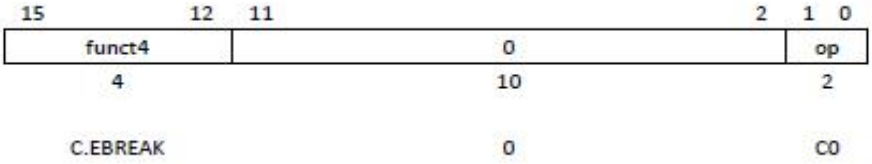
一条所有位都是 0 的 16 位指令，被永久的保留为一条非法指令。

NOP 指令



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。

4.6 在 LR/SC 序列中使用 C 指令

在支持 C 扩展的实现上，当要确保最终成功时，可以在 LR/SC 序列中使用 I 类压缩格式指令。

4.7 RVC 指令集列表

图 1.11 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。

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								

图 1.11 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)

图 1.12 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 (HINT, nzimm=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

图 1.13 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 (HINT, rd=0; RV32 NSE, nzimm[5]=1)
000	0	rd≠0	0	10	C.SLLI64 (RV128; RV32/64 HINT; HINT, rd=0)
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)

图 1.14 RVC 指令列表，10 部分

4.8 指令压缩统计

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

图 1.15 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%。

图 1.16 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.BBREAK	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	—

图 1.15 按典型静态频率排序的 RVC 指令。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	—

图 1.16 按典型动态频率排序的 RVC 指令。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。

4.9 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 1.17 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%。

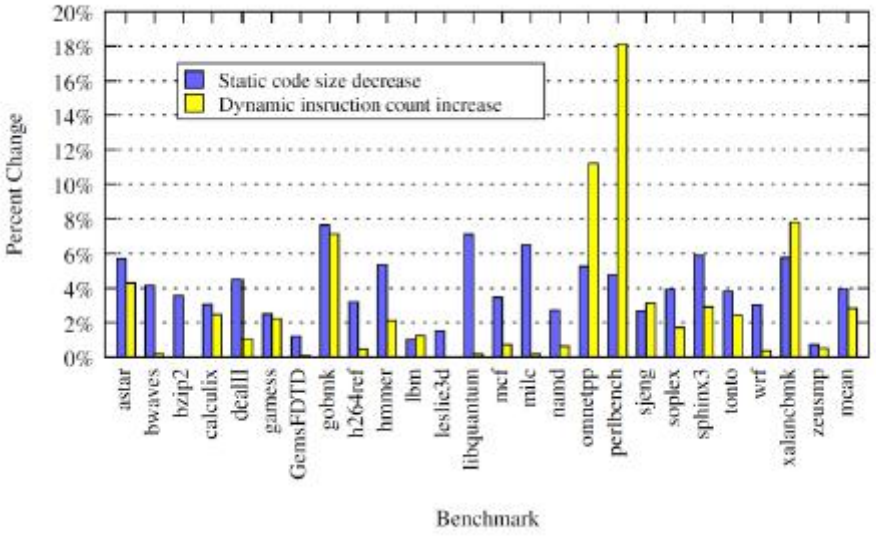


图 1.17 压缩的函数入口、出口处的子过程，对静态代码大小和动态指令数目的影响。

参考文献： riscv-spec-v2.2