

## 第七章 压缩指令

小即是美。——E. F. Schumacher, 1973

### 7.1 导言

以前的 ISA 为了缩短代码长度而显著扩展了指令和指令格式的数量，比如添加了一些只有两个（而不是三个）操作数的指令，减小立即数域，等等。ARM 和 MIPS 两次发明了整个 ISA 以缩减代码：ARM Thumb 和 Thumb 2，加上 MIPS16 和 microMIPS。这些新的 ISA 成为了处理器和编译器的阻碍，同时也增加了汇编语言程序员的认知负担。

RV32C 采用了一种新颖的方法：每条短指令必须和一条标准的 32 位 RISC-V 指令一一对应。此外，16 位指令只对汇编器和链接器可见，并且是否以短指令取代对应的宽指令由它们决定。编译器编写者和汇编语言程序员可以幸福地忽略 RV32C 指令及其格式，他们能感知到的则是最后的程序大小小于大多数其它 ISA 的程序。图 7.1 是 RV32C 扩展指令集的图形化表示。

RISC-V 架构师在一系列程序中选择使用 RVC 扩展中的指令来达到良好的代码压缩效果。把它们缩减为 16 位指令的可行性来源于观察到的以下三个现象。第一，对十个常用寄存器（a0-a5, s0-s1, sp 以及 ra）的访问频率远超过其他寄存器；第二，许多指令的写入目标是它的源操作数之一；第三，立即数往往很小，而且有些指令比较喜欢某些特定的立即数。因此，许多 RV32C 指令只能访问那些常用寄存器；一些指令隐式写入源操作数的位置；几乎所有的立即数都被缩短了，load 和 store 操作只使用操作数整数倍尺寸的无符号数偏移量。

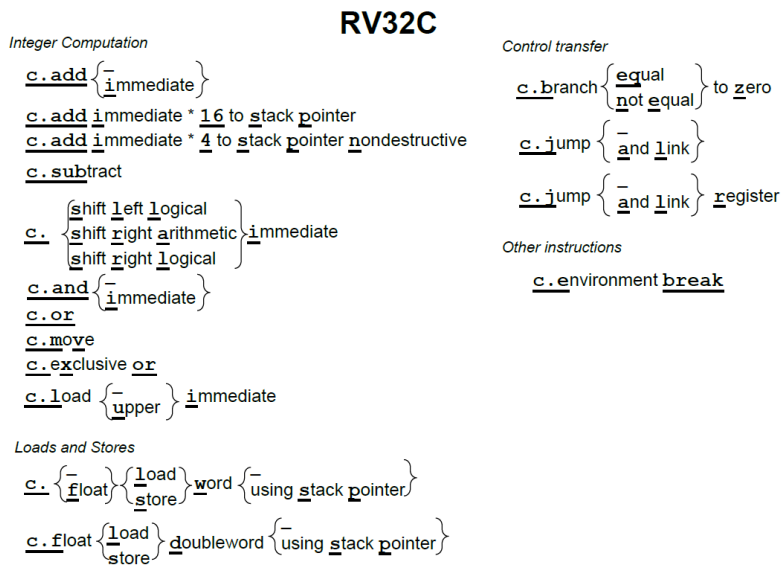


图 7.1: RV32C 的指令图示。移位指令的立即数域和 c.addi4spn 是零扩展的，其它指令采用符号位扩展。

Benchmark	ISA	ARM Thumb-2	microMIPS	x86-32	RV32I+RVC
Insertion Sort	Instructions	18	24	20	19
	Bytes	46	56	45	52
DAXPY	Instructions	10	12	16	11
	Bytes	28	32	50	28

图 7.2：用压缩指令集写成的插入排序和 DAXPY 程序的指令数和代码长度。

图 7.3 和 7.4 列出了插入排序程序和 DAXPY 的 RV32C 代码。我们展示了这些 RV32C 指令，从而清楚地显示了这些压缩操作的效果，但是通常这些指令在汇编程序中是不可见的。注释中在括号内标出了与 RV32C 指令对应的等效 32 位指令。附录 A 中完整列出了 16 位 RV32C 指令和 32 位 RISC-V 指令的对应关系。

例如，在图 7.3 的插入排序程序中地址为 4 的地方，汇编器将如下的 32 位 RV32I 指令：

```
addi a4,x0,1 # i = 1
```

替换为了这条 16 位 RV32C 指令：

```
c.li a4,1 # (可扩展为 addi a4,x0,1) i = 1
```

RV32C 的 load 立即数指令比较短，是因为它只能指定一个寄存器和一个小的立即数。

c.li 的机器码在图 7.3 中只有 4 个十六进制数，这表明 c.li 指令确实只有 2 字节长。

另一个例子在图 7.3 中地址为 10 的地方，汇编器将：

```
add a2,x0,a3 # a2 是 a[j] 的指针
```

换成了这条 16 位 RV32C 指令：

```
c.mv a2,a3 # (可扩展为 add a2,x0,a3) a2 是 a[j] 的指针
```

RV32C 的 move 指令只有 16 位长，因为它只指定两个寄存器。

尽管处理器的设计者们不能忽略 RV32C 的存在，但是有一个技巧可以让实现的代价变小：在执行之前用一个解码器将所有的 16 位指令转换为等价的 32 位指令。图 7.6 到 7.8 列了解码器可以转换的 RV32C 指令的格式和操作码。最小的不支持任何扩展的 32 位 RISC-V 处理器要用到 8000 个门电路，而解码器只要 400 个门。如果它在这么小的设计中都只占 5% 的体量，那么它在约有 100000 个门的中等大小带有 cache 的处理器中几乎等于没有。

**有什么不同之处？** RV32C 中没有字节或半字指令，因为其他指令对代码长度的影响更大。

第 9 页图 1.5 中 Thumb-2 相对于 RV32C 有代码长度更小的优势，是由于 Load and Store Multiple 在过程进入和退出时导致的节省。RV32C 中没有包括它们，从而保证和 RV32G 中指令的一一映射。RV32G 中为了降低高端处理器的实现复杂性而省略了这些指令。由于 Thumb-2 是独立于 ARM-32 的 ISA，所以硬件必须有两个解码器，一个用于 ARM-32，一个用于 Thumb-2。RV32GC 是一个单独的 ISA，因此 RISC-V 处理器只需要一个解码器。

批注 [GL1]: 类似的地方需进一步修正

**补充说明：为什么有些架构师不考虑 RV32C？**

超标量处理器在一个时钟周期内同时获取几条指令，因此译码阶段可能成为超标量处理器的瓶颈。macrofusion 是另一个例子，其中指令解码器把 RISC-V 指令组合为更加强大的指令来执行（参见第一章）。在这种情况下，16 位 RV32C 指令和 32 位 RV32I 指令混杂在一起增加了解码的复杂度，从而使得高性能处理器中在一个时钟周期内完成解码变得更难。

## 7.2 RV32GC, Thumb-2, microMIPS 和 x86-32 的比较

图 7.2 是这四个 ISA 写成的插入排序和 DAXPY 程序的大小的总览。

在插入排序的原始 19 条 RV32I 指令中, 12 条被替换成了 RV32C 指令, 所以代码长度从  $19 \times 4 = 76$  个字节变成了  $12 \times 2 + 7 \times 4 = 52$  个字节, 节省了  $24/76 = 32\%$ 。DAXPY 程序从  $11 \times 4 = 44$  个字节缩减到了  $8 \times 2 + 3 \times 4 = 28$  个字节, 节省了  $16/44 = 36\%$ 。

这两个小例子的结果令人惊讶地符合第二章第 9 页的图 1.5, 其中提到, 对于更多更复杂的程序, RV32G 代码比 RV32GC 代码长 37%。为了达到这种程度的长度缩减, 程序中必须有一半的指令可以替换成 RV32C 指令。

---

### 补充说明: RV32C 真的是独一无二的吗?

RV32 指令在 RV32IC 中无法区分。Thumb-2 实际上是一个单独的 ISA, 包含 16 位指令和 ARMv7 中大多数的指令。例如, 在 Thumb-2 中有 Compare and Branch on Zero, 而 ARMv7 中没有, 对于 Reverse Subtarct with Carry 正好相反。microMIPS 并不是 MIPS32 的超集。例如, microMIPS 计算分支偏移量的时候乘以 2, 但在 MIPS32 中则为 4。RISC-V 中总是乘以 2。

---

## 7.3 结束语

我本可以把信写得更短, 但我没有时间。——Blaise Pascal, 1656

他是一位数学家, 建造了第一台机械计算器, 因此图灵奖得主 Niklaus Wirth 用他的名字命名了一门编程语言。

RV32C 让 RISC-V 程序拥有了当今几乎最小的代码尺寸。你几乎可以将它们视为硬件协助的伪指令。但是, 现在汇编程序将它们在汇编语言程序员和编译器编写者面前隐藏起来, 而不是像第三章所说的那样用常用操作扩展真实的指令集, 从而使得 RISC-V 代码更容易使用和阅读。这两种方法都有助于提供程序员的工作效率。

我们认为 RV32C 是 RISC-V 的简洁、有效的机制对于性价比的提升的最好的例子之一。

## 7.4 更多请见

.....

```

# RV32C (19 instructions, 52 bytes)
# a1 is n, a3 points to a[0], a4 is i, a5 is j, a6 is x
0: 00450693 addi a3,a0,4 # a3 is pointer to a[i]
4: 4705 c.li a4,1 # (expands to addi a4,x0,1) i = 1
Outer Loop:
6: 00b76363 bltu a4,a1,c # if i < n, jump to Continue Outer loop
a: 8082 c.ret # (expands to jalr x0,ra,0) return from function
Continue Outer Loop:
c: 0006a803 lw a6,0(a3) # x = a[i]
10: 8636 c.mv a2,a3 # (expands to add a2,x0,a3) a2 is pointer to a[j]
12: 87ba c.mv a5,a4 # (expands to add a5,x0,a4) j = i
InnerLoop:
14: ffc62883 lw a7,-4(a2) # a7 = a[j-1]
18: 01185763 ble a7,a6,26 # if a[j-1] <= a[i], jump to Exit InnerLoop
1c: 01162023 sw a7,0(a2) # a[j] = a[j-1]
20: 17fd c.addi a5,-1 # (expands to addi a5,a5,-1) j--
22: 1671 c.addi a2,-4 # (expands to addi a2,a2,-4) decr a2 to point to a[j]
24: fbe5 c.bnez a5,14 # (expands to bne a5,x0,14) if j!=0, jump to InnerLoop
Exit InnerLoop:
26: 078a c.slli a5,0x2 # (expands to slli a5,a5,0x2) multiply a5 by 4
28: 97aa c.add a5,a0 # (expands to add a5,a5,a0) a5 = byte address of a[j]
2a: 0107a023 sw a6,0(a5) # a[j] = x
2e: 0705 c.addi a4,1 # (expands to addi a4,a4,1) i++
30: 0691 c.addi a3,4 # (expands to addi a3,a3,4) incr a3 to point to a[i]
32: bfd1 c.j 6 # (expands to jal x0,6) jump to Outer Loop

```

图 7.3: 插入排序的 RV32C 代码。12 条 16 位指令使得代码长度缩减了 32%。每条指令的宽度可以很容易地得知。RV32C 指令（以 c. 开头）在这个例子中显式出现，但通常汇编语言程序员和编译器无法看到它们。

```

# RV32DC (11 instructions, 28 bytes)
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], fa0 is a
0: cd09 c.beqz a0,1a # (expands to beq a0,x0,1a) if n==0, jump to Exit
2: 050e c.slli a0,a0,0x3 # (expands to slli a0,a0,0x3) a0 = n*8
4: 9532 c.add a0,a2 # (expands to add a0,a0,a2) a0 = address of x[n]
Loop:
6: 2218 c.fld fa4,0(a2) # (expands to fld fa4,0(a2) ) fa5 = x[]
8: 219c c.fld fa5,0(a1) # (expands to fld fa5,0(a1) ) fa4 = y[]
a: 0621 c.addi a2,8 # (expands to addi a2,a2,8) a2++ (incr. ptr to y)
c: 05a1 c.addi a1,8 # (expands to addi a1,a1,8) a1++ (incr. ptr to x)
e: 72a7f7c3 fmadd.d fa5,fa5,fa0,fa4 # fa5 = a*x[i] + y[i]
12: fef63c27 fsd fa5,-8(a2) # y[i] = a*x[i] + y[i]
16: fea618e3 bne a2,a0,6 # if i != n, jump to Loop
Exit:
1a: 8082 ret # (expands to jalr x0,ra,0) return from function

```

图 7.4: DAXPY 的 RV32DC 代码。8 条十六位指令将代码长度缩减了 36%。每条指令的宽度见第二列的十六进制字符个数。RV32C 指令（以 c. 开头）在这个例子中显式出现，但通常汇编语言程序员和编译器无法看到它们。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
000	nzimm[5]			0				nzimm[4:0]				01		CI c.nop					
000	nzimm[5]			rs1/rd≠0				nzimm[4:0]				01		CI c.addi					
001	imm[11 4 9:8 10 6 7 3:1 5]																		
010	imm[5]			rd≠0				imm[4:0]				01		CI c.li					
011	nzimm[9]			2				nzimm[4 6 8:7 5]				01		CI c.addi16sp					
011	nzimm[17]			rd≠{0, 2}				nzimm[16:12]				01		CI c.lui					
100	nzuimm[5]			00	rs1'/rd'				nzuimm[4:0]				01		CI c.srli				
100	nzuimm[5]			01	rs1'/rd'				nzuimm[4:0]				01		CI c.srai				
100	imm[5]			10	rs1'/rd'				imm[4:0]				01		CI c.andi				
100	0			11	rs1'/rd'				00	rs2'				01		CR c.sub			
100	0			11	rs1'/rd'				01	rs2'				01		CR c.xor			
100	0			11	rs1'/rd'				10	rs2'				01		CR c.or			
100	0			11	rs1'/rd'				11	rs2'				01		CR c.and			
101				imm[11 4 9:8 10 6 7 3:1 5]												01		CJ c.j	
110				imm[8 4:3]				rs1'				imm[7:6 2:1 5]				01		CB c.beqz	
111				imm[8 4:3]				rs1'				imm[7:6 2:1 5]				01		CB c.bnez	

图7.5: RV32C操作码映射 (bits[1:0] = 01) 列出了指令布局, 操作码, 指令格式和指令名称。  
rd', rs1' 和rs2' 指的是10个常用的寄存器a0-aa5, s0-s1, sp和ra。(本图来源于[Waterman and Asanović 2017]的表12.5。)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
000		0										0		00		CIW <i>Illegal instruction</i>
000		nzuimm[5:4 9:6 2 3]										rd'		00		CIW c.addi4spn
001		uimm[5:3]			rs1'			uimm[7:6]			rd'		00		CL c.fld	
010		uimm[5:3]			rs1'			uimm[2:6]			rd'		00		CL c.lw	
011		uimm[5:3]			rs1'			uimm[2:6]			rd'		00		CL c.flw	
101		uimm[5:3]			rs1'			uimm[7:6]			rs2'		00		CL c.fsd	
110		uimm[5:3]			rs1'			uimm[2:6]			rs2'		00		CL c.sw	
111		uimm[5:3]			rs1'			uimm[2:6]			rs2'		00		CL c.fsw	

图7.6: RV32C操作码映射 (bits[1:0] = 00) 列出了指令布局, 操作码, 指令格式和指令名称。  
rd', rs1' 和rs2' 指的是10个常用的寄存器a0-aa5, s0-s1, sp和ra。(本图来源于[Waterman and Asanović 2017]的表12.4。)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
000			nzuimm[5]					rs1/rd≠0				nzuimm[4:0]			10	CI c.slli
000			0					rs1/rd≠0				0			10	CI c.slli64
001			uimm[5]					rd				uimm[4:3 8:6]			10	CSS c.fldsp
010			uimm[5]					rd≠0				uimm[4:2 7:6]			10	CSS c.lwsp
011			uimm[5]					rd				uimm[4:2 7:6]			10	CSS c.flwsp
100			0					rs1≠0				0			10	CJ c.jr
100			0					rd≠0				rs2≠0			10	CR c.mv
100			1					0				0			10	CI c.ebreak
100			1					rs1≠0				0			10	CJ c.jalr
100			1					rs1/rd≠0				rs2≠0			10	CR c.add
101								uimm[5:3 8:6]				rs2			10	CSS c.fsdsp
110								uimm[5:2 7:6]				rs2			10	CSS c.swsp
111								uimm[5:2 7:6]				rs2			10	CSS c.fwsp

图7.7：RV32C操作码映射（bits[1:0] = 10）列出了指令布局，操作码，指令格式和指令名称。（本图来源于[Waterman and Asanović 2017]的表12.6。）

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							

图7.8：16位RVC压缩指令的格式。rd'，rs1' 和rs2' 指的是10个常用的寄存器a0-aa5，s0-s1，sp和ra。（本图来源于[Waterman and Asanović 2017]的表12.1。）