RISC-V
RV32C: Compressed Instructions

Sparsh Mittal

Courtesy: "The RISC-V Reader: An Open Architecture Atlas" by David Patterson and Andrew Waterman

Why add compressed instructions

The smaller the program, the smaller the area on a chip needed for the program memory, which can be a significant cost for embedded devices.

Indeed, that issue inspired ARM architects to retroactively add shorter instructions in the Thumb and Thumb-2 ISAs.

Smaller programs also lead to fewer misses in instruction caches, which saves power since off-chip DRAM accesses use much more energy than on-chip SRAM accesses, and improves performance.

RV32C

- **Prior ISAs** significantly expanded the number of instructions and instruction formats to shrink code size: adding short instructions with two operands instead of three, small immediate fields, and so on.
- **RV32C** takes a novel approach: every short instruction must map to one single standard 32-bit RISC-V instruction.
- Only the assembler and linker are aware of the 16- bit instructions, and it is up to them to replace a wide instruction with its narrow cousin.

- The compiler writer and assembly language programmer need not worry about RV32C instructions.
  - RV32I instructions are indistinguishable in RV32IC.
- A decoder translates all 16-bit instructions into their equivalent 32-bit version **before** they execute.
- This makes compression instructions inexpensive.

RV32C

Integer Computation

c.add { – / immediate }

c.add immediate * 16 to stack pointer

c.add immediate * 4 to stack pointer nondestructive

c.subtract

c. { shift left logical / shift right arithmetic / shift right logical } immediate

c.and { – / immediate }

c.or

c.move

c.exclusive or

c.load { – / upper } immediate

Loads and Stores

c. { – / float } { load / store } word { – / using stack pointer }

c.float { load / store } doubleword { – / using stack pointer }

Control transfer

c.branch { equal / not equal } to zero

c.jump { – / and link }

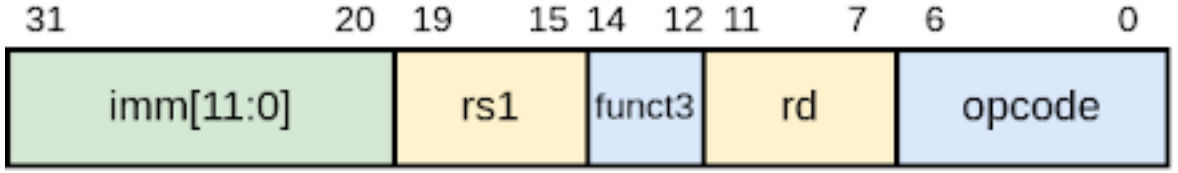
c.jump { – / and link } register

Other instructions

c.environment break

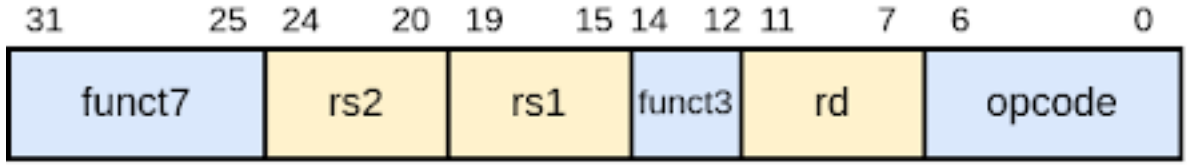
The immediate fields of the shift instructions and c.addi4spn are zero extended and sign extended for the other instructions.

Comparison of original and compressed instruction

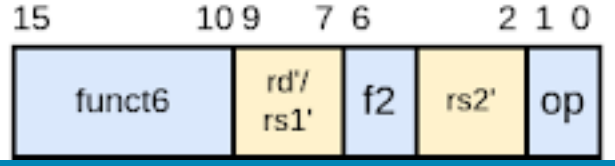


I-type

CI-type



R-type



CA-type

Which instructions to compress?

- Three key observations

1. Ten popular registers (a0–a5, s0–s1, sp, and ra) are accessed far more than the rest.
2. Many instructions overwrite one of their source operands.
3. Immediate operands tend to be small, and some instructions favor certain immediates.
   - E.g., loads and stores use only unsigned offsets in multiples of the operand size.

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

RV32C code for Insertion Sort and DAXPY

Instructions and code size for Insertion Sort and DAXPY for compressed ISAs

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

RV32C code for Insertion Sort

```
# 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
```

Notice Width of instructions

The twelve 16-bit instructions make the code 32% smaller.

RV32C instructions (starting with c.) are shown explicitly in this example, but normally assembly language programmers and compilers cannot see them.

RV32DC code for DAXPY

```
# 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
```

The eight 16-bit instructions shrink the code by 36%

Example 1 of compressed instruction

The assembler replaced the following 32-bit RV32I instruction:

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

with this 16-bit RV32C instruction:

```
c.li a4,1              # (expands to addi a4,x0,1) i = 1
```

The RV32C load immediate instruction is narrower because it must specify only one register and a small immediate.

Example 2 of compressed instruction

The assembler replaced the following 32-bit RV32I instruction:

```
add a2,x0,a3                # a2 is pointer to a[j]
```

with this 16-bit RV32C instruction:

```
c.mv a2,a3          # (expands to add a2,x0,a3) a2 is pointer to a[j]
```

The RV32C move instruction is merely 16 bits long because it specifies only two registers

Compressed 16-bit RVC instruction formats.

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

RV32C opcode map (bits[1 : 0] = 01)

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]					01	CJ c.jal
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.addi16s
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

rd', rs1', and rs2' refer to the 10 popular registers a0–a5, s0–s1, sp, and ra.

RV32C opcode map (bits[1 : 0] = 00)

15 14 13	12 11 10	9 8 7	6 5	4 3 2	1 0	
000	0			0	00	CIW *Illegal instruction*
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

RV32C opcode map (bits[1 : 0] = 10)

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.fswsp

References

https://fprox.substack.com/p/riscv-c-extension