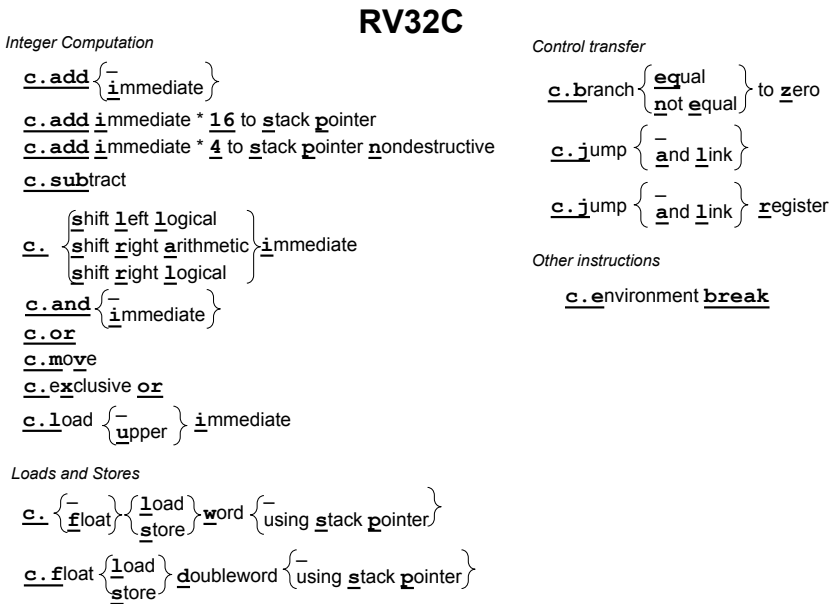


Figure 7.1: Diagram of the RV32C instructions. The immediate fields of the shift instructions and `c.addi4spn` are zero extended and sign extended for the other instructions.

```
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. The `c.li` machine code is only four hexadecimal digits in Figure 7.3, showing that the `c.li` instruction is indeed 2 bytes long.

Another example is at address 10 in Figure 7.3, where the assembler replaced:

```
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.

While the processor designer can't ignore RV32C, an implementation trick makes them inexpensive: a decoder translates all 16-bit instructions into their equivalent 32-bit version *before* they execute. Figures 7.6 to 7.8 list the RV32C instruction formats and opcodes that the decoder translates. It is equivalent to only 400 gates when the tiniest 32-bit processor—without any RISC-V extensions—is 8000 gates. If it's 5% of such a tiny design, the decoder nearly disappears inside a moderate processor that with caches is order 100,000 gates.

What's Different? There are no byte or halfword instructions in RV32C because other instructions have a bigger influence on code size. The small size advantage of Thumb-2 over RV32C in Figure 1.5 on page 9 is due to the code size savings of Load and Store Multiple on procedure entry and exit. RV32C excludes them to maintain the one-to-one mapping to RV32G instructions, which omits them to reduce implementation complexity for high-end processors. Since Thumb-2 is a separate ISA from ARM-32, but a processor can switch between them, the hardware must have two instruction decoders: one for ARM-32 and one for Thumb-2. RV32GC is a single ISA, so RISC-V processors need only a single decoder.

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

Figure 7.2: Instructions and code size for Insertion Sort and DAXPY for compressed ISAs.

■ ***Elaboration: Why would architects ever skip RV32C?***

Instruction decode can be a bottleneck for superscalar processors that try to fetch several instructions per clock cycle. Another example is *macrofusion*, whereby the instruction decoder combines RISC-V instructions together to form more powerful instructions for execution (see Chapter 1). A mix of 16-bit RV32C and 32-bit RV32I instructions can make sophisticated decoding more difficult to complete within the clock cycle of a high-performance implementation.

7.2 Comparing RV32GC, Thumb-2, microMIPS, and x86-32

Figure 7.2 summarizes the size of Insertion Sort and DAXPY for these four ISAs.

Of the 19 original RV32I instructions in Insertion Sort, 12 become RV32C, so the code shrinks from $19 \times 4 = 76$ bytes to $12 \times 2 + 7 \times 4 = 52$ bytes, saving $24/76 = 32\%$. DAXPY shrinks from $11 \times 4 = 44$ bytes to $8 \times 2 + 3 \times 4 = 28$ bytes, or $16/44 = 36\%$.

The results for these two small examples are surprisingly in line with Figure 1.5 on page 9 in Chapter 2, which shows that RV32G code is about 37% larger than RV32GC code, for a larger set of much bigger programs. To achieve that level of savings, over half of the instructions in the programs had to be RV32C instructions.

■ ***Elaboration: Is RV32C really unique?***

RV32I instructions are indistinguishable in RV32IC. Thumb-2 is actually a separate ISA with 16-bit instructions plus most but not all of ARMv7. For example, *Compare and Branch on Zero* is in Thumb-2 but not ARMv7, and vice versa for *Reverse Subtract with Carry*. Nor is microMIPS32 a superset of MIPS32. For example, microMIPS multiplies branch displacements by two but it's four in MIPS32. RISC-V *always* multiplies by two.

7.3 Concluding Remarks

I would have written a shorter letter, but I did not have the time.

—Blaise Pascal, 1656.

He was a mathematician who built one of the first mechanical calculators, which led Turing Award laureate Niklaus Wirth to name a programming language after him.

RV32C gives RISC-V one of the smallest code sizes today. You can almost think of them as hardware-assisted pseudoinstructions. However, now the assembler is hiding them from the assembly language programmer and compiler writer rather than, as in Chapter 3,

expanding the real instruction set with popular operations that make RISC-V code easier to use and to read. Both approaches aid programmer productivity.

We consider RV32C as one of RISC-V's best examples of a simple, powerful mechanism that improves its cost-performance.

7.4 To Learn More

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

Notes

[1] `http://parlab.eecs.berkeley.edu`

```
# 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
```

Figure 7.3: RV32C code for Insertion Sort. The twelve 16-bit instructions make the code 32% smaller. The width of each instruction is evident by the number of hexadecimal characters in the second column. The RV32C instructions (starting with `c.`) are shown explicitly in this example, but normally assembly language programmers and compilers cannot see them.

```
# 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
```

Figure 7.4: RV32DC code for DAXPY. The eight 16-bit instructions shrink the code by 36%. The width of each instruction is evident by the number of hexadecimal characters in the second column. The RV32C instructions (starting with `c.`) are shown explicitly in this example, but normally they are invisible to the assembly language programmer and compiler.

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.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

Figure 7.5: RV32C opcode map (bits$[1:0]$ = 01) lists layout, opcodes, format, and names. `rd'`, `rs1'`, and `rs2'` refer to the 10 popular registers `a0`–`a5`, `s0`–`s1`, `sp`, and `ra`. (Table 12.5 of Waterman and Asanović 2017] is the basis of this figure.)

15–13	12–11	10–9	8–7	6–5	4–3	2–0	
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

Figure 7.6: RV32C opcode map (bits$[1:0]$ = 00) lists layout, opcodes, format, and names. `rd'`, `rs1'`, and `rs2'` refer to the 10 popular registers `a0`–`a5`, `s0`–`s1`, `sp`, and `ra`. (Table 12.4 of Waterman and Asanović 2017] is the basis of this figure.)

15–13	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

Figure 7.7: RV32C opcode map (bits$[1:0]$ = 10) lists layout, opcodes, format, and names. (Table 12.6 of Waterman and Asanović 2017] is the basis of this figure.)

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

Figure 7.8: Compressed 16-bit RVC instruction formats. `rd'`, `rs1'`, and `rs2'` refer to the 10 popular registers `a0`–`a5`, `s0`–`s1`, `sp`, and `ra`. (Table 12.1 of Waterman and Asanović 2017] is the basis of this figure.)

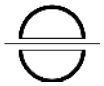
8 RV32V: Vector

I'm all for simplicity. If it's very complicated I can't understand it.

—Seymour Cray

Seymour Cray (1925-1996) was architect of the Cray-1 in 1976, the first commercially successful supercomputer using a vector architecture. The Cray-1 was a gem; it was the world's fastest computer even *without* using the vector instructions.

The Intel Multimedia Extensions (MMX) in 1997 made SIMD popular. They were embraced and expanded via Streaming SIMD Extensions (SSE) in 1999 and Advanced Vector Extensions (AVX) in 2010. MMX fame was fueled by an Intel ad campaign showing disco-dancing workers of a semiconductor line clad in technicolor clean suits (https://www.youtube.com/watch?v=paU16B-bZEA).



8.1 Introduction

This chapter focuses on *data-level parallelism*, where there is plenty of data that the desired application can compute on concurrently. Arrays are a popular example. While fundamental to scientific applications, multimedia programs use arrays as well. The former uses single- and double-precision float-point data and the latter often uses 8- and 16-bit integer data.

The best known architecture for data-level parallelism is *Single Instruction Multiple Data* (*SIMD*). SIMD first became popular by partitioning 64-bit registers into many 8-, 16-, or 32-bit pieces and then computing on them in parallel. The opcode supplied the data width and the operation. Data transfers are simply loads and stores of a single (wide) SIMD register.

The first step of partitioning existing 64-bit registers is tempting because it is straightforward. To make SIMD faster, architects subsequently widen the registers to compute more partitions concurrently. Because the SIMD ISAs belong to the incremental school of design, and the opcode specifies the data width, expanding the SIMD registers also expands the SIMD instruction set. Each subsequent step of doubling the width of SIMD registers and the number of SIMD instructions leads ISAs down the path of escalating complexity, which is borne by processor designers, compiler writers, and assembly language programmers.

An older and, in our opinion, more elegant alternative to exploit data-level parallelism is the *vector* architecture. This chapter provides our rationale for using vectors instead of SIMD in RISC-V.

Vector computers gather objects from main memory and put them into long, sequential vector registers. Pipelined execution units compute very efficiently on these vector registers. Vector architectures then scatter the results back from the vector registers to main memory. The size of the vector registers is determined by the implementation, rather than baked into the opcode, as with SIMD. As we shall see, *separating the vector length and maximum operations per clock cycle from the instruction encoding is the crux of the vector architecture:* the vector microarchitect can flexibly design the data-parallel hardware without affecting the programmer, and the programmer can take advantage of longer vectors without rewriting the code. In addition, vector architectures have many fewer instructions than SIMD architectures. Moreover, vector architectures have well-established compiler technology, unlike SIMD.