31 25	24 20	19 15	14 12	11 7	6 0	
00010 aq rl	00000	rs1	010	rd	0101111	R lr.w
00011 aq rl	rs2	rs1	010	rd	0101111	R sc.w
00001 aq rl	rs2	rs1	010	rd	0101111	R amoswap.w
00000 aq rl	rs2	rs1	010	rd	0101111	R amoadd.w
00100 aq rl	rs2	rs1	010	rd	0101111	R amoxor.w
01100 aq rl	rs2	rs1	010	rd	0101111	R amoand.w
01000 aq rl	rs2	rs1	010	rd	0101111	R amoor.w
10000 aq rl	rs2	rs1	010	rd	0101111	R amomin.w
10100 aq rl	rs2	rs1	010	rd	0101111	R amomax.w
11000 aq rl	rs2	rs1	010	rd	0101111	R amominu.w
11100 aq rl	rs2	rs1	010	rd	0101111	R amomaxu.w

Figure 6.2: RV32A opcode map has instruction layout, opcodes, format type, and names. (Table 19.2 of [Waterman and Asanović 2017] is the basis of this figure.

The AMO instructions atomically perform an operation on an operand in memory and set the destination register to the original memory value. Atomic means there can be no interrupt between the read and the write of memory, nor could other processors modify the memory value between the memory read and write of the AMO instruction.

AMOs and LR/SC require naturally aligned memory addresses because it is onerous for hardware to guarantee atomicity across cache-line boundaries.

Load reserved and store conditional provide an atomic operation across two instructions. Load reserved reads a word from memory, writes it to the destination register, and records a reservation on that word in memory. Store conditional stores a word at the address in a source register *provided there exists a load reservation on that memory address*. It writes zero to the destination register if the store succeeded, or a nonzero error code otherwise.

An obvious question is: Why does RV32A have two ways to perform atomic operations? The answer is that there are two quite distinct use cases.

Programming language developers assume the underlying architecture can perform an atomic compare-and-swap operation: Compare a register value to a value in memory addressed by another register, and if they are equal, then swap a third register value with the one in memory. They make that assumption because it is a universal synchronization primitive, in that any other single-word synchronization operation can be synthesized from compare-and-swap [Herlihy 1991].

While that is powerful argument for adding such an instruction to an ISA, it requires three source registers in one instruction. Alas, going from two to three source operands would complicate the integer datapath, control, and the instruction format. (The three source operands of RV32FD's multiply-add instructions affect the floating-point datapath, not the integer datapath.) Fortunately, load reserved and store conditional have only two source registers and can implement atomic compare and swap (see top half of Figure 6.3).

The rationale for also having AMO instructions is that they scale better to large multiprocessor systems than load reserved and store conditional. They can also be used to implement reduction operations efficiently. AMOs are useful as well for communicating with I/O devices, because they perform a read and a write in a single atomic bus transaction. This atomicity can both simplify device drivers and improve I/O performance. The bottom half of Figure 6.3 shows how to write a critical section using atomic swap.

```
# Compare-and-swap (CAS) memory word M[a0] using lr/sc.
# Expected old value in a1; desired new value in a2.
   0: 100526af    lr.w  a3,(a0)     # Load old value
   4: 06b69e63    bne   a3,a1,80    # Old value equals a1?
   8: 18c526af    sc.w  a3,a2,(a0)  # Swap in new value if so
   c: fe069ae3    bnez  a3,0        # Retry if store failed
        ... code following successful CAS goes here ...
  80:                               # Unsuccessful CAS.


# Critical section guarded by test-and-set spinlock using an AMO.
   0: 00100293    li           t0,1        # Initialize lock value
   4: 0c55232f    amoswap.w.aq t1,t0,(a0)  # Attempt to acquire lock
   8: fe031ee3    bnez         t1,4        # Retry if unsuccessful
        ... critical section goes here ...
  20: 0a05202f    amoswap.w.rl x0,x0,(a0)  # Release lock.
```

Figure 6.3: Two examples of synchronization. The first uses load reserved/store conditional `lr.w,sc.w` to implement compare-and-swap, and the second uses an atomic swap `amoswap.w` to implement a mutex.

■ ***Elaboration: Memory consistency models***

RISC-V has a relaxed memory consistency model, so other threads may view some memory accesses out of order. Figure 6.2 shows that all RV32A instructions have an *acquire bit* (`aq`) and a *release bit* (`rl`). An atomic operation with the `aq` bit set guarantees that other threads will see the AMO in-order with *subsequent* memory accesses. If the `rl` bit is set, other threads will see the atomic operation in-order with *previous* memory accesses. To learn more, [Adve and Gharachorloo 1996] is an excellent tutorial on the topic.

What's Different? The original MIPS-32 had no mechanism for synchronization, but architects added load reserved / store conditional instructions to a later MIPS ISA.

6.2 Concluding Remarks

RV32A is optional, and a RISC-V processor is simpler without it. However, as Einstein said, everything should be as simple *as possible*, but no simpler. Many situations require RV32A.

6.3 To Learn More

S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 1991.

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

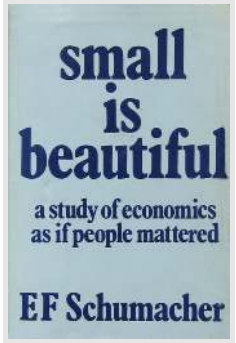
A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

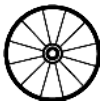
Notes

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

7 RV32C: Compressed Instructions

E. F. Schumacher (1911-1977) wrote this economics book that advocated human-scale, decentralized, and appropriate technologies. Translated into numerous languages, it was named one of the 100 most influential books since World War II.





Small is Beautiful.

—E. F. Schumacher, 1973

7.1 Introduction

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. ARM and MIPS invented whole ISAs twice to shrink code: ARM Thumb and Thumb-2 plus MIPS16 and microMIPS. These new ISAs hampered the processor and the compiler and increased the cognitive load on the assembly language programmer.

RV32C takes a novel approach: *every* short instruction *must* map to *one* single standard 32-bit RISC-V instruction. Moreover, 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 can be blissfully oblivious of the RV32C instructions and their formats, except ending up with programs that are smaller than most. Figure 7.1 is a graphical representation of the RV32C extension instruction set.

The RISC-V architects chose the instructions in the RVC extension to obtain good code compression across a range of programs, using three observations to fit them into 16 bits. First, ten popular registers (`a0`–`a5`, `s0`–`s1`, `sp`, and `ra`) are accessed far more than the rest. Second, many instructions overwrite one of their source operands. Third, immediate operands tend to be small, and some instructions favor certain immediates. So, many RV32C instructions can access only the popular registers; some instructions implicitly overwrite a source operand; and almost all immediates are reduced in size, with loads and stores using only unsigned offsets in multiples of the operand size.

Figures 7.3 and 7.4 list the RV32C code for Insertion Sort and DAXPY. We show the RV32C instructions to demonstrate the impact of compression explicitly, but normally these instructions are invisible in the assembly language program. The comments show the equivalent 32-bit instructions to the RV32C instructions parenthetically. Appendix A includes the 32-bit RISC-V instruction that corresponds to each 16-bit RV32C instruction.

For example, at address 4 in Insertion Sort in Figure 7.3, the assembler replaced the following 32-bit RV32I instruction:

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

with this 16-bit RV32C instruction: