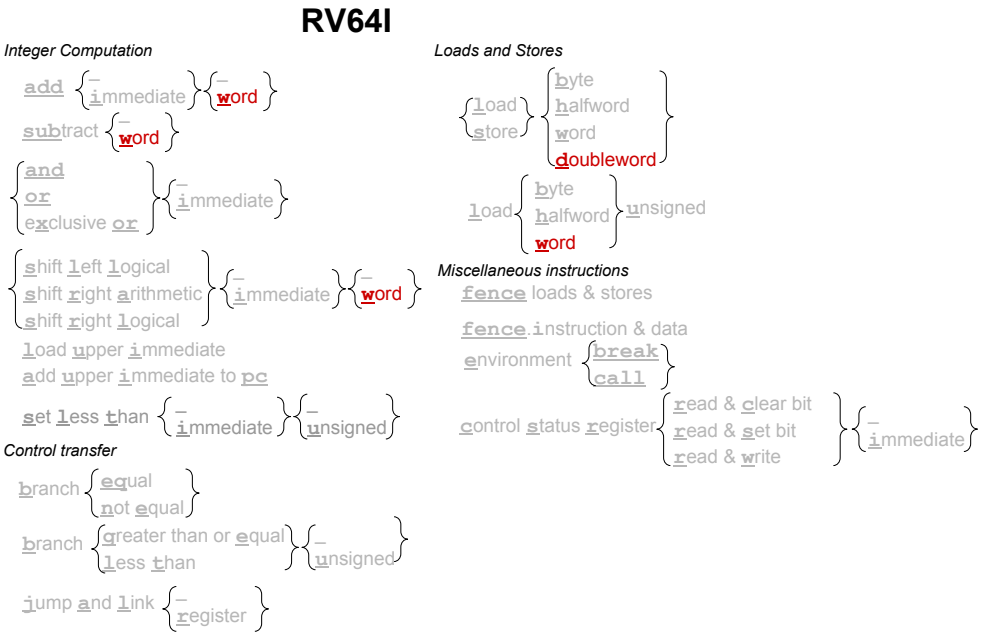


Figure 9.1: Diagram of the RV64I instructions. The underlined letters are concatenated from left to right to form RV64I instructions. The dimmed portion are the old RV64I instructions extended to 64-bit registers and the dark (red) portion are the new instructions for RV64I.

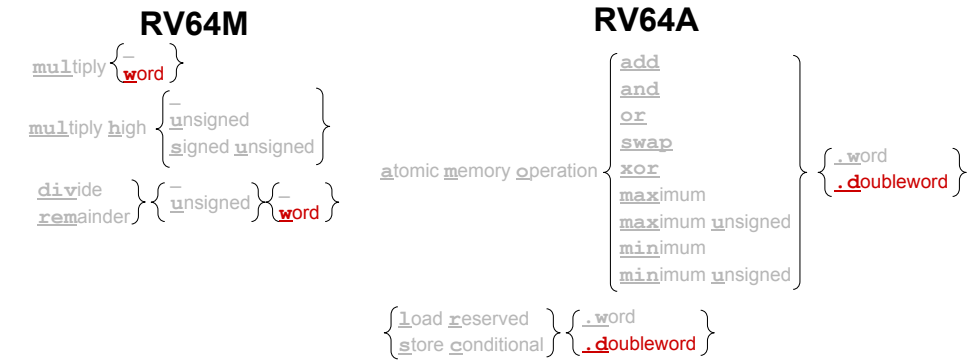


Figure 9.2: Diagrams of the RV64M and RV64A instructions.

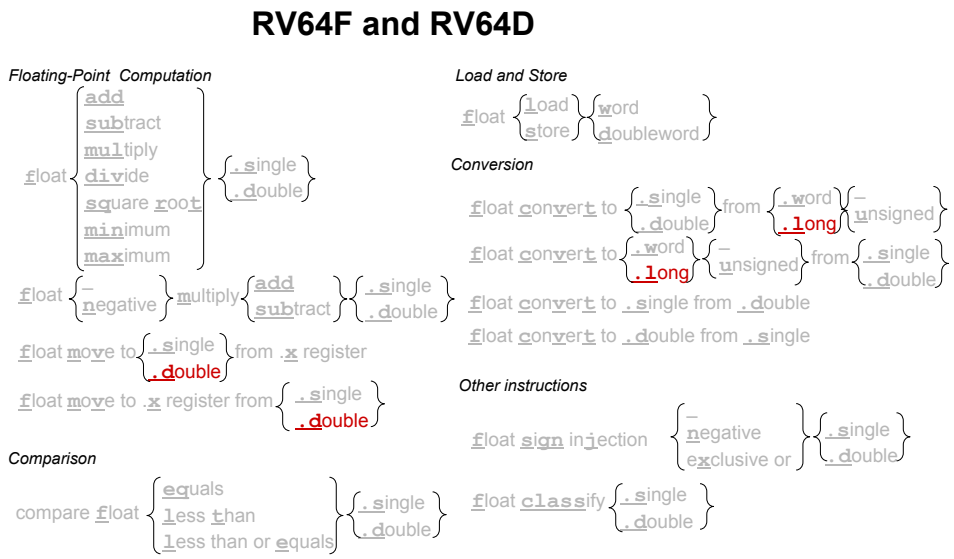


Figure 9.3: Diagram of the RV64F and RV64D instructions.

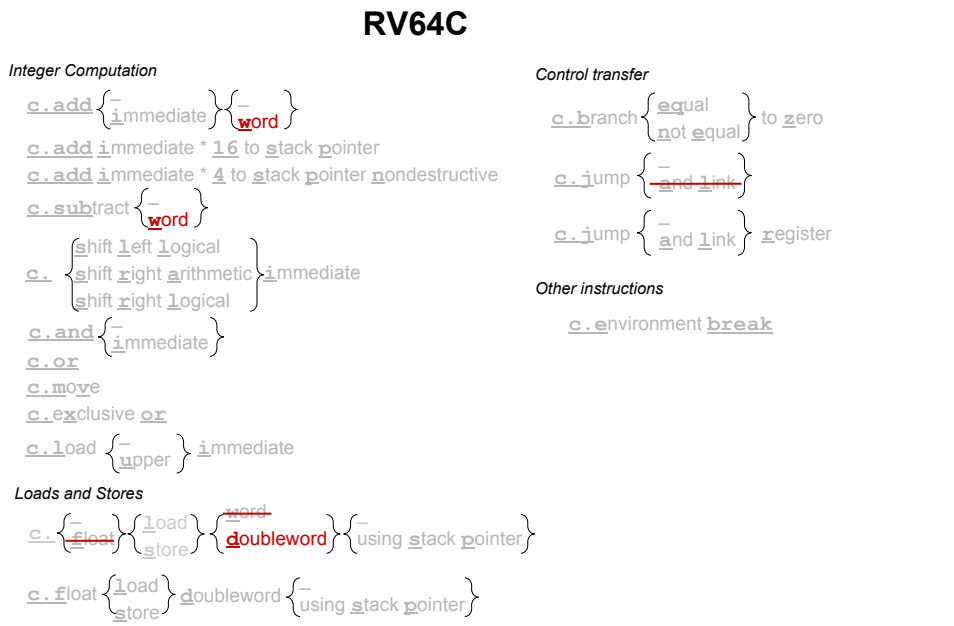


Figure 9.4: Diagram of the RV64C instructions.

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	110	rd	0000011	I lwu
imm[11:0]		rs1	011	rd	0000011	I ld
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	S sd
000000	shamt	rs1	001	rd	0010011	I slli
000000	shamt	rs1	101	rd	0010011	I srli
010000	shamt	rs1	101	rd	0010011	I srai
imm[11:0]		rs1	000	rd	0011011	I addiw
0000000	shamt	rs1	001	rd	0011011	I slliw
0000000	shamt	rs1	101	rd	0011011	I srliw
0100000	shamt	rs1	101	rd	0011011	I sraiw
0000000	rs2	rs1	000	rd	0111011	R addw
0100000	rs2	rs1	000	rd	0111011	R subw
0000000	rs2	rs1	001	rd	0111011	R sllw
0000000	rs2	rs1	101	rd	0111011	R srlw
0100000	rs2	rs1	101	rd	0111011	R sraw

RV64M Standard Extension (in addition to RV32M)

0000001	rs2	rs1	000	rd	0111011	R mulw
0000001	rs2	rs1	100	rd	0111011	R divw
0000001	rs2	rs1	101	rd	0111011	R divuw
0000001	rs2	rs1	110	rd	0111011	R remw
0000001	rs2	rs1	111	rd	0111011	R remuw

RV64A Standard Extension (in addition to RV32A)

00010	aq	rl	00000	rs1	011	rd	0101111	R lr.d
00011	aq	rl	rs2	rs1	011	rd	0101111	R sc.d
00001	aq	rl	rs2	rs1	011	rd	0101111	R amoswap.d
00000	aq	rl	rs2	rs1	011	rd	0101111	R amoadd.d
00100	aq	rl	rs2	rs1	011	rd	0101111	R amoxor.d
01100	aq	rl	rs2	rs1	011	rd	0101111	R amoand.d
01000	aq	rl	rs2	rs1	011	rd	0101111	R amoor.d
10000	aq	rl	rs2	rs1	011	rd	0101111	R amomin.d
10100	aq	rl	rs2	rs1	011	rd	0101111	R amomax.d
11000	aq	rl	rs2	rs1	011	rd	0101111	R amominu.d
11100	aq	rl	rs2	rs1	011	rd	0101111	R amomaxu.d

RV64F Standard Extension (in addition to RV32F)

1100000	00010	rs1	rm	rd	1010011	R fcvt.l.s
1100000	00011	rs1	rm	rd	1010011	R fcvt.lu.s
1101000	00010	rs1	rm	rd	1010011	R fcvt.s.l
1101000	00011	rs1	rm	rd	1010011	R fcvt.s.lu

RV64D Standard Extension (in addition to RV32D)

1100001	00010	rs1	rm	rd	1010011	R fcvt.l.d
1100001	00011	rs1	rm	rd	1010011	R fcvt.lu.d
1110001	00000	rs1	000	rd	1010011	R fmv.x.d
1101001	00010	rs1	rm	rd	1010011	R fcvt.d.l
1101001	00011	rs1	rm	rd	1010011	R fcvt.d.lu
1111001	00000	rs1	000	rd	1010011	R fmv.d.x

Figure 9.5: RV64 opcode map of the base instructions and optional extensions. It shows instruction layout, opcodes, format type, and name. (Table 19.2 of [Waterman and Asanović 2017] is the basis of this figure.)

RV64F and RV64D adds integer doublewords to the convert instructions, calling them *longs* so to prevent confusion with double precision floating-point data: `fcvt.l.s`, `fcvt.l.d`, `fcvt.lu.s`, `fcvt.lu.d`, `fcvt.s.l`, `fcvt.s.lu`, `fcvt.d.l`, `fcvt.d.lu`. As the integer `x` registers are now 64 bits wide, they can now hold double precision floating-point data, so RV64D adds two floating-point moves: `fmv.x.w` and `fmv.w.x`.

The one exception to the superset relationship between RV64 and RV32 is the compressed instructions. RV64C replaced a few RV32C instructions, since other instructions shrank code more for 64-bit addresses. RV64C drops the compressed jump and link (`c.jal`) and the integer and floating-point load and store word instructions (`c.lw`, `c.sw`, `c.lwsp`, `c.swsp`, `c.flw`, `c.fsw`, `c.flwsp`, and `c.fswsp`). In their place, RV64C adds the more popular add and subtract word instructions (`c.addw`, `c.addiw`, `c.subw`) and load and store doubleword instructions (`c.ld`, `c.sd`, `c.ldsp`, `c.sdsp`).

■ ***Elaboration: The RV64 ABIs are lp64, lp64f, and lp64d.***

lp64 means that the C language data types long and pointer are 64 bits; int is still 32 bits. The suffixes f and d indicate how floating-point arguments are passed, which is the same as for RV32 (see Chapter 3).

■ ***Elaboration: There is no instruction diagram for RV64V***

because it exactly matches RV32V due to dynamic register typing. The only change is that the X64 and X64U dynamic register types in Figure 8.2 on page 75 are available in RV64V but not RV32V.

9.2 Comparison to Other 64-bit ISAs using Insertion Sort

As Gordon Bell said at the opening of this chapter, the one fatal architecture flaw is running out of address bits. As programs pushed the limits of a 32-bit address space, architects began to make 64-bit address versions of their ISAs [Mashey 2009].

The earliest was MIPS in 1991. It extended all registers and the program counter from 32 to 64 bits and added new 64-bit versions of the MIPS-32 instructions. The MIPS-64 assembly language instructions all begin with the letter "d", such as `daddu` or `dsll` (see Figure 9.10). Programmers can mix MIPS-32 and MIPS-64 instructions in the same program. MIPS-64 dropped the load delay slot from MIPS-32 (the pipeline stalls on a read-after-write dependence).

A decade later, it was time for a successor to x86-32. When architects increased the addressing size, they took the opportunity to make a few more improvements in x86-64:

- Increased the number of integer registers from 8 to 16 (`r8`–`r15`);
- Increased the number of SIMD registers from 8 to 16 (`xmm8`–`xmm15`); and
- Added PC-relative data addressing to better support position-independent code.

These improvements smoothed some rough edges of x86-32.

You can see the benefits by comparing the x86-32 version of Insertion Sort in Figure 2.11 on page 30 in Chapter 2 to the x86-64 version in Figure 9.11. The newer ISA keeps all the variables in registers rather than having several in memory, which reduces the instruction

ISA	ARM-64	MIPS-64	x86-64	RV64I	RV64I+RV64C
Instructions	16	24	15	19	19
Bytes	64	96	46	76	52

Figure 9.6: Number of instructions and code size for Insertion Sort for four ISAs. ARM Thumb-2 and microMIPS are 32-bit address ISAs, so are unavailable for ARM-64 and MIPS-64.

count from 20 to 15 instructions. The code size is actually larger by one byte with the newer ISA despite having fewer instructions: 46 versus 45. The reason is that to squeeze in the new opcodes to enable more registers, x86-64 added a prefix byte to identify the new instructions. The average instruction length increases in x86-64 over x86-32.

ARM faced the same address problem another decade later. Rather than evolve the old ISA to have 64-bit addresses as did x86-64, they used the opportunity to invent a brand new ISA. Given a fresh start, they changed many of the awkward ARM-32 traits to give them a modern ISA:

- Increase the number of integer registers from 15 to 31;
- Remove the PC from the set of registers;
- Provide a register that's hardwired to zero for most instructions (`r31`);
- Unlike ARM-32, all ARM-64 data addressing modes work with all data sizes and types;
- ARM-64 dropped the load and store multiple instructions of ARM-32; and
- ARM-64 omitted the conditional execution option of ARM-32 instructions.

It still shares some weaknesses of ARM-32: condition codes for branch, source and destination register fields move in the instruction format, conditional move instructions, complex addressing modes, inconsistent performance counters, and only 32-bit length instructions. ARM-64 can't switch to the Thumb-2 ISA, as Thumb-2 only works with 32-bit addresses.

Unlike RISC-V, ARM decided to take a maximalist approach to ISA design. While certainly a better ISA than ARM-32, it is also bigger. For example, it has more than 1000 instructions and the ARM-64 manual is 3185 pages long [ARM 2015]. Moreover, it is still growing. There have been three expansions of ARM-64 since its announcement a few years ago.

The ARM-64 code for Insertion Sort in Figure 9.9 looks closer to the RV64I code or x86-64 code than to the ARM-32 code. For example, with 31 registers, there is no need to save and restore registers from the stack. And since the PC is no longer one of the registers, ARM-64 uses a separate return instruction.

Figure 9.6 is a table that summarizes the number of instructions and number of bytes in Insertion Sort for the ISAs. Figures 9.8 to 9.11 show the compiled code for RV64I, ARM-64, MIPS-64, and x86-64. Parenthetical phrases in the comments of these four programs identify the differences between the RV32I versions in Chapter 2 and these RV64I versions.

MIPS-64 needs the most instructions, primarily because of the `nop` instructions of the unfilled delayed branch slots. RV64I needs fewer because of the compare-and-branch instructions and no delayed branch. While ARM-64 and x86-64 need two compare instructions

Intel didn't invent the x86-64 ISA. When switching to 64-bit addresses, Intel invented a new ISA called Itanium that was incompatible with x86-32. Its competitor for x86-32 processors was locked out of Itanium, so AMD invented a 64-bit version of x86-32 called AMD64. Itanium eventually failed, so Intel was forced to adopt the AMD64 ISA as the 64-bit address successor of x86-32, which we call x86-64 [Kerner and Padgett 2007].

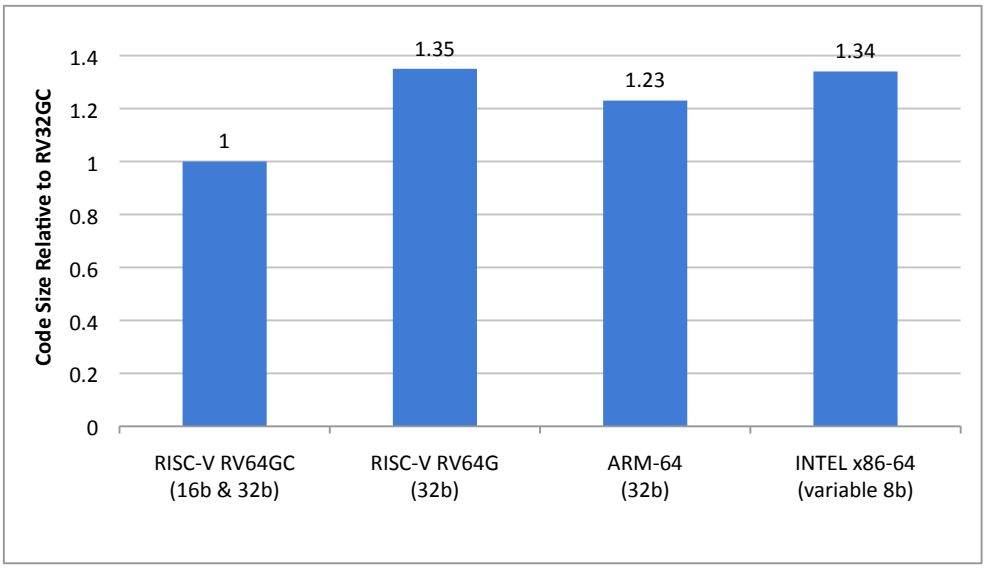


Figure 9.7: Relative program sizes for RV64G, ARM-64, and x86-64 versus RV64GC. This comparison measures much bigger programs than in Figure 9.6. This graph is the 64-bit address equivalent to the graph of 32-bit ISAs in Figure 1.5 on page 9 in Chapter 2. RV32C code size almost matches to RV64C; it is 1% smaller. There is no Thumb-2 option for ARM-64, so the core of other 64-bit ISAs significantly exceeds the size of RV64GC code. The programs measured were the SPEC CPU2006 benchmarks using the GCC compilers [Waterman 2016].

that are unnecessary for RV64I, their scaling addressing modes avoid address arithmetic instructions needed in RV64I, giving them the fewest instructions. However, RV64I+RV64C has much smaller code size, as the next section explains.

■ ***Elaboration: ARM-64, MIPS-64, and x86-64 aren't the official names.***

The official names are: ARMv8 is what we call ARM-64, MIPS-IV is MIPS-64, and AMD64 is x86-64 (see the sidebar on the previous page for the history of x86-64).

9.3 Program size

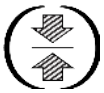


Figure 9.7 compares average relative code sizes for RV64, ARM-64, and x86-64. Compare this figure to Figure 1.5 on page 9 in Chapter 1. First, RV32GC code is almost identical in size to RV64GC; it is only 1% smaller. This closeness is also true for RV32I and RV64I. While ARM-64 code is 8% smaller than ARM-32 code, there is no 64-bit address version of Thumb-2, so all instructions remain 32-bits long. Hence, ARM-64 code is 25% *larger* than ARM Thumb-2 code. Code for x86-64 is 7% larger than x86-32 code due to adding prefix opcodes to x86-64 instructions to accommodate new operations and the expanded set of registers. RV64GC wins as ARM-64 code is 23% bigger than RV64GC and x86-64 code is 34% bigger than RV64GC. That difference is large enough that either it will improve performance due to lower instruction cache miss rates, or reduce cost by allowing a smaller instruction cache that still provides satisfactory miss rates.

9.4 Concluding Remarks

One of the problems of being a pioneer is you always make mistakes, and I never, never want to be a pioneer. It's always best to come second when you can look at the mistakes the pioneers made.

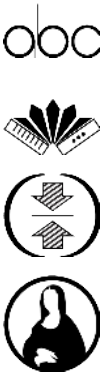
—Seymour Cray, architect of the first supercomputer, 1976

MIPS is for sale. Imagination Technologies, which bought the MIPS ISA in 2012 for $100M, recently announced that it is for sale; no buyers yet.

Running out of address bits is the Achilles heel of computer architecture. Many an architecture has died from a wound there. ARM-32 and Thumb-2 remain 32-bit architectures, so they're no help for big programs. Some ISAs like MIPS-64 and x86-64 survived the transition, but x86-64 is not a paragon of ISA design and the future of MIPS-64 is unclear at the time of this writing. ARM-64 is a new large ISA, and time will tell how successful it will be.

RISC-V benefited from designing both the 32-bit and the 64-bit architectures together, whereas older ISAs had to architect them sequentially. Unsurprisingly, the transition between 32-bit and 64-bit is easiest for RISC-V programmers and compiler writers; the RV64I ISA has virtually all RV32I instructions. Indeed, that is why we can list both RV32GCV and RV64GCV in only two pages of the Reference Card. More important, the simultaneous design meant the 64-bit architecture did not have to be squeezed into a cramped 32-bit opcode space. RV64I has plenty of room for optional instruction extensions, particularly RV64C, which makes it the leader in code size.

We see the 64-bit architecture as more evidence of RISC-V's sound design, admittedly easier to achieve if you start 20 years later so that you can borrow the pioneers' good ideas as well as learn from their mistakes.



■ ***Elaboration: RV128***

RV128 began as an inside joke with the RISC-V architects, simply to show that a 128-bit address ISA was possible. However, warehouse scale computers may soon have more than 2^{64} bytes of semiconductor storage (DRAM and Flash memory), which programmers might want to access as a memory address. There are also proposals to use a 128-bit address to improve security [Woodruff et al. 2014]. The RISC-V manual does specify a full 128-bit ISA called RV128G [Waterman and Asanović 2017]. The additional instructions are basically the same as needed to go from RV32 to RV64, which Figures 9.1 to 9.4 illustrate. All the registers also grow to 128 bits, and the new RV128 instructions specify either 128-bit versions of some instructions (using Q in the name for `quadword`) or 64-bit versions of others (using D for in the name `doubleword`).

9.5 To Learn More

I. ARM. Armv8-a architecture reference manual. 2015.

M. Kerner and N. Padgett. A history of modern 64-bit computing. Technical report, CS Department, University of Washington, Feb 2007. URL `http://courses.cs.washington.edu/courses/csep590/06au/projects/history-64-bit.pdf`.

J. Mashey. The long road to 64 bits. *Communications of the ACM*, 52(1):45–53, 2009.

A. Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html`.

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/`.

J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 457–468. IEEE, 2014.

Notes

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

```
# RV64I (19 instructions, 76 bytes, or 52 bytes with RV64C)
# a1 is n, a3 points to a[0], a4 is i, a5 is j, a6 is x
   0: 00850693  addi  a3,a0,8    # (8 vs 4) a3 is pointer to a[i]
   4: 00100713  li    a4,1       # i = 1
Outer Loop:
   8: 00b76463  bltu  a4,a1,10   # if i < n, jump to Continue Outer loop
Exit Outer Loop:
   c: 00008067  ret              # return from function
Continue Outer Loop:
  10: 0006b803  ld    a6,0(a3)   # (ld vs lw) x = a[i]
  14: 00068613  mv    a2,a3      # a2 is pointer to a[j]
  18: 00070793  mv    a5,a4      # j = i
Inner Loop:
  1c: ff863883  ld    a7,-8(a2)  # (ld vs lw, 8 vs 4) a7 = a[j-1]
  20: 01185a63  ble   a7,a6,34   # if a[j-1] <= a[i], jump to Exit Inner Loop
  24: 01163023  sd    a7,0(a2)   # (sd vs sw) a[j] = a[j-1]
  28: fff78793  addi  a5,a5,-1   # j--
  2c: ff860613  addi  a2,a2,-8   # (8 vs 4) decrement a2 to point to a[j]
  30: fe0796e3  bnez  a5,1c      # if j != 0, jump to Inner Loop
Exit Inner Loop:
  34: 00379793  slli  a5,a5,0x3  # (8 vs 4) multiply a5 by 8
  38: 00f507b3  add   a5,a0,a5   # a5 is now byte address oi a[j]
  3c: 0107b023  sd    a6,0(a5)   # (sd vs sw) a[j] = x
  40: 00170713  addi  a4,a4,1    # i++
  44: 00868693  addi  a3,a3,8    # increment a3 to point to a[i]
  48: fc1ff06f  j     8          # jump to Outer Loop  # continue outer loop
```

Figure 9.8: RV64I code for Insertion Sort in Figure 2.5. The RV64I assembly language program is very similar to the RV32I assembly language in Figure 2.8 on page 27 in Chapter 2. We list the differences in parentheses in the comments. The size of the data is now 8 bytes instead of 4, so three instructions change the constant 4 to 8. This extra width also stretches two load words (`lw`) to load doublewords (`ld`) and two store words (`sw`) to store doublewords (`sd`).

```
# ARM-64 (16 instructions, 64 bytes)
# x0 points to a[0], x1 is n, x2 is j, x3 is i, x4 is x
  0: d2800023  mov  x3, #0x1             # i = 1
Outer Loop:
  4: eb01007f  cmp  x3, x1               # compare i vs n
  8: 54000043  b.cc 10                   # if i < n, jump to Continue Outer loop
Exit Outer Loop:
  c: d65f03c0  ret                       # return from function
Continue Outer Loop:
 10: f8637804  ldr  x4, [x0, x3, lsl #3] # (x4 ca r4) vs x = a[i]
 14: aa0303e2  mov  x2, x3               # (x2 vs r2) j = i
Inner Loop:
 18: 8b020c05  add  x5, x0, x2, lsl #3   # x5 is pointer to a[j]
 1c: f85f80a5  ldur x5, [x5, #-8]        # x5 = a[j]
 20: eb0400bf  cmp  x5, x4               # compare a[j-1] vs. x
 24: 5400008d  b.le 34                   # if a[j-1]<=a[i], jump to Exit Inner Loop

 28: f8227805  str  x5, [x0, x2, lsl #3] # a[j] = a[j-1]
 2c: f1000442  subs x2, x2, #0x1         # j--
 30: 54ffff41  b.ne 18                   # if j != 0, jump to Inner Loop
Exit Inner Loop:
 34: f8227804  str  x4, [x0, x2, lsl #3] # a[j] = x
 38: 91000463  add  x3, x3, #0x1         # i++
 3c: 17fffff2  b    4                    # jump to Outer Loop
```

Figure 9.9: ARM-64 code for Insertion Sort in Figure 2.5. The ARM-64 assembly language program is different from to the ARM-32 assembly language in Figure 2.11 on page 30 in Chapter 2 since it is a new instruction set. The registers start with x instead of a. The data addressing modes can shift a register by 3 bits to scale the index to a byte address. With 31 registers, there is no need to save and restore registers from the stack. Since PC is not one of the registers, it uses is a separate return instruction. In fact, the code looks closer to the RV64I code or x86-64 code than to the ARM-32 code.

```
# MIPS-64 (24 instructions, 96 bytes)
# a1 is n, a3 is pointer to a[0], v0 is j, v1 is i, t0 is x
  0: 64860008 daddiu a2,a0,8   # (daddiu vs addiu, 8 vs 4) a2 is pointer to a[i]
  4: 24030001 li     v1,1      # i = 1
Outer Loop:
  8: 0065102b sltu   v0,v1,a1  # set on i < n
  c: 14400003 bnez   v0,1c     # if i < n, jump to Continue Outer Loop
 10: 00c03825 move   a3,a2     # a3 is pointer to a[j] (slot filled)
 14: 03e00008 jr     ra        # return from function
 18: 00000000 nop              # branch delay slot unfilled
Continue Outer Loop:
 1c: dcc80000 ld     a4,0(a2)  # (ld vs lw) x = a[i]
 20: 00601025 move   v0,v1     # j = i
Inner Loop:
 24: dce9fff8 ld     a5,-8(a3) # (ld vs lw, 8 vs. 4, a5 vs t1) a5 = a[j-1]
 28: 0109502a slt    a6,a4,a5  # (no load delay slot) set a[i] < a[j-1]
 2c: 11400005 beqz   a6,44     # if a[j-1] <= a[i], jump to Exit Inner Loop
 30: 00000000 nop              # branch delay slot unfilled
 34: 6442ffff daddiu v0,v0,-1  # (daddiu vs addiu) j--
 38: fce90000 sd     a5,0(a3)  # (sd vs sw, a5 vs t1) a[j] = a[j-1]
 3c: 1440fff9 bnez   v0,24     # if j != 0, jump to Inner Loop (next slot filled)
 40: 64e7fff8 daddiu a3,a3,-8  # (daddiu vs addiu, 8 vs 4) decr a2 pointer to a[j]
Exit Inner Loop:
 44: 000210f8 dsll   v0,v0,0x3 # (dsll vs sll)
 48: 0082102d daddu  v0,a0,v0  # (daddu vs addu) v0 now byte address oi a[j]
 4c: fc480000 sd     a4,0(v0)  # (sd vs sw) a[j] = x
 50: 64630001 daddiu v1,v1,1   # (daddiu vs addiu) i++
 54: 1000ffec b      8         # jump to Outer Loop (next delay slot filled)
 58: 64c60008 daddiu a2,a2,8   # (daddiu vs addiu, 8 vs 4) incr a2 pointer to a[i]
 5c: 00000000 nop              # Unncessary(?)
```

Figure 9.10: MIPS-64 code for Insertion Sort in Figure 2.5. The MIPS-64 assembly language program has several differences from to the MIPS-32 assembly language in Figure 2.10 on page 29 in Chapter 2. First, most operations for 64-bit data prepend a "d" to their names: `daddiu, daddu, dsll`. Like Figure 9.8, three instructions change the constant from 4 to 8 since size of the data grew from 4 to 8 bytes. Again like RV64I, the extra width also stretches two load words (`lw`) to load doublewords (`ld`) and two store words (`sw`) to store doublewords (`sd`). Finally, MIPS-64 does not have the load delay slot from MIPS-32; the pipeline stalls on a read-after-write dependence.

```
# x86-64 (15 instructions, 46 bytes)
# rax is j, rcx is x, rdx is i, rsi is n, rdi is pointer to a[0]
  0: ba 01 00 00 00 mov edx,0x1
Outer Loop:
  5: 48 39 f2       cmp rdx,rsi             # compare i vs. n
  8: 73 23          jae 2d <Exit Loop>      # if i >= n, jump to Exit Outer Loop
  a: 48 8b 0c d7    mov rcx,[rdi+rdx*8]     # x = a[i]
  e: 48 89 d0       mov rax,rdx             # j = i
Inner Loop:
 11: 4c 8b 44 c7 f8 mov r8,[rdi+rax*8-0x8] # r8 = a[j-1]
 16: 49 39 c8       cmp r8,rcx              # compare a[j-1] vs. x
 19: 7e 09          jle 24 <Exit Loop>      # if a[j-1]<=a[i],jump to Exit InnerLoop
 1b: 4c 89 04 c7    mov [rdi+rax*8],r8      # a[j] = a[j-1]
 1f: 48 ff c8       dec rax                 # j--
 22: 75 ed          jne 11 <Inner Loop>     # if j != 0, jump to Inner Loop
Exit InnerLoop:
 24: 48 89 0c c7    mov [rdi+rax*8],rcx     # a[j] = x
 28: 48 ff c2       inc rdx                 # i++
 2b: eb d8          jmp 5 <Outer Loop>      # jump to Outer Loop
Exit Outer Loop:
 2d: c3             ret                     # return from function
```

Figure 9.11: x86-64 code for Insertion Sort in Figure 2.5. The x86-64 assembly language program is quite different from to the x86-32 assembly language in Figure 2.11 on page 30 in Chapter 2. First, unlike RV64I, the wider registers have different names `rax, rcx, rdx, rsi, rdi, r8`. Second, because x86-64 added 8 more registers, there are now enough to keep all the variables in registers instead of in memory. Third, the x86-64 instructions are longer than for x86-32 since many need to prepend 8-bits or 16-bits to fit the new instructions in the opcode space. For example, incrementing or decrementing a register (`inc, dec`) takes 1 byte in x86-32 but 3 bytes in x86-64. Hence, while many fewer instructions, x86-64 code size of Insertion Sort is almost identical to x86-32: 45 bytes vs. 46 bytes.

10 RV32/64 Privileged Architecture

Edsger W. Dijkstra (1930–2002) received the 1972 Turing Award for fundamental contributions to developing programming languages.

Simplicity is prerequisite for reliability.

—Edsger W. Dijkstra

10.1 Introduction

The book so far has focused on RISC-V support for general-purpose computation: all of the instructions we've introduced are available in *user mode*, where application code usually runs. This chapter introduces two new *privilege* modes: *machine mode*, which runs the most trusted code, and *supervisor mode*, which provides support for operating systems like Linux, FreeBSD, and Windows. Both new modes are more privileged than user mode, hence the title of the chapter. More-privileged modes generally have access to all of the features of less-privileged modes, and they add additional functionality not available to less-privileged modes, such as the ability to handle interrupts and perform I/O. Processors typically spend most of their execution time in their least-privileged mode; interrupts and exceptions transfer control to more-privileged modes.

Embedded-system runtimes and operating systems use the features of these new modes to respond to external events, like the arrival of network packets; to support multitasking and protection between tasks; and to abstract and virtualize hardware features. Given the breadth of these topics, a thorough programmer's guide would be an entire additional book; instead, this chapter aims to hit the high notes of the RISC-V features. Programmers disinterested in embedded system runtimes and operating systems can either skip or skim this chapter.

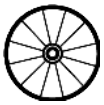


Figure 10.1 is a graphical representation of the RISC-V privileged instructions, and Figure 10.2 lists these instructions' opcodes. As you can see, the privileged architecture adds

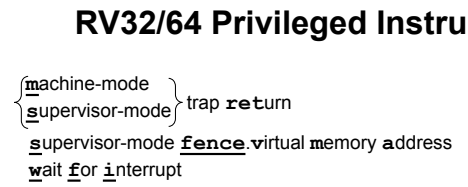


Figure 10.1: Diagram of the RISC-V privileged instructions instructions.