

## 第九章 RV64：64 位地址指令

### C. Gordon Bell

(1934-) 是当时最受欢迎的两种小型机架构的首席架构师之一：1970 年宣布的数字设备公司 PDP-11 (16 位地址) 及其七年后的继任者, 数字设备公司 32 位地址 VAX-11 (虚拟地址扩展)。



在计算机设计中只能出现一个错误是难以恢复的——没有足够的地址位用于存储器寻址和存储器管理。

——C. Gordon Bell, 1976

### 9.1 引言

图 9.1 至 9.4 是 RV32G 指令集的 64 位版本 RV64G 指令集的图示。由图可见, 到切换到 64 位 ISA, ISA 只添加了若干指令。切换到 64 位, 指令集只添加了 32 位指令对应的字, 双字和长字版本的指令, 并将所有寄存器 (包括 PC) 扩展为 64 位。因此, RV64I 中的 sub 减的是两个 64 位数字而不是 RV32I 中的减两个 32 位数字。RV64 很接近 RV32 但实际上又有所不同; 它添加了少量指令同时基础指令做的事情与 RV32 中稍有不同。

例如, 图 9.8 中 RV64I 版本的插入排序与第 2 章第 27 页的图 2.8 中 RV32I 版本的插入排序非常相似。它们指令数量和大小都相同。唯一的变化是加载和存储字指令变为加载并存储双字, 地址增量从 4 (4 字节) 变为 8 双字 (8 字节)。图 9.5 列出了图 9.1 到 9.4 中的 RV64GC 指令的操作码。

尽管 RV64I 有 64 位地址且默认数据大小为 64 位, 32 位字仍然是程序中的有效数据类型。因此, RV64I 需要支持字, 就像 RV32I 需要支持字节和半字一样。更具体地说, 由于寄存器现在是 64 位宽, RV64I 添加字版本的加法和减法指令: `addw`, `addiw`, `subw`。这些指令将计算结果截断为 32 位, 结果符号扩展后再写入目标寄存器。RV64I 也包括字版本的移位指令 (`sllw`, `slliw`, `srlw`, `srliw`, `sraw`, `sraiw`) 的字, 以获得 32 位移位结果而不是 64 位移位结果。要进行 64 位数据传输, RV64 提供了加载和存储双字指令: `ld`, `sd`。最后, 就像 RV32I 中有无符号版本的加载单字节和加载半字的指令, RV64I 也有一个无符号版本的加载字: `lwu`。

出于类似的原因, RV64 需要添加字版本的乘法, 除法和取余指令: `mulw`, `divw`, `divuw`, `remw`, `remuw`。为了支持对单字及双字的同步操作, RV64A 为其所有的 11 条指令都添加了双字版本。

## RV64I

### Integer Computation

add { immEDIATE } { word }  
subtract { word }  
{ and } { immEDIATE }  
{ or } { immEDIATE }  
{ exclusive or } { immEDIATE }  
{ shift left logical } { immEDIATE } { word }  
{ shift right arithmetic } { immEDIATE } { word }  
{ shift right logical } { immEDIATE } { word }  
load upper immediate  
add upper immediate to pc  
set less than { immEDIATE } { unsigned }

### Control transfer

branch { equal }  
branch { not equal }  
branch { greater than or equal } { unsigned }  
branch { less than } { unsigned }  
jump and link { register }

### Loads and Stores

{ load } { byte }  
{ sore } { halfword }  
{ sore } { word }  
{ sore } { doubleword }  
load { byte } { unsigned }  
load { halfword } { word }

### Miscellaneous instructions

fence loads & stores  
fence instruction & data  
environment { break }  
environment { call }  
control status register { read & clear bit } { immEDIATE }  
control status register { read & set bit } { immEDIATE }  
control status register { read & write } { immEDIATE }

图 9.1: RV64I 指令图示。带下划线的字母从左到右连起来构成 RV64I 指令。灰色部分是扩展到 64 位寄存器的旧 RV64I 指令，而暗（红色）部分是 RV64I 的新指令。

## RV64M

multiply { word }  
multiply high { unsigned }  
multiply high { signed unsigned }  
divide { unsigned } { word }  
remainder { unsigned } { word }

## RV64A

atomic memory operation { add } { word }  
atomic memory operation { and } { word }  
atomic memory operation { or } { word }  
atomic memory operation { swap } { word }  
atomic memory operation { xor } { word }  
atomic memory operation { maximum } { word }  
atomic memory operation { maximum unsigned } { word }  
atomic memory operation { minimum } { word }  
atomic memory operation { minimum unsigned } { word }  
{ load reserved } { word }  
{ sore conditional } { word }

图 9.2: RV64M 和 RV64A 指令图示

## RV64F and RV64D

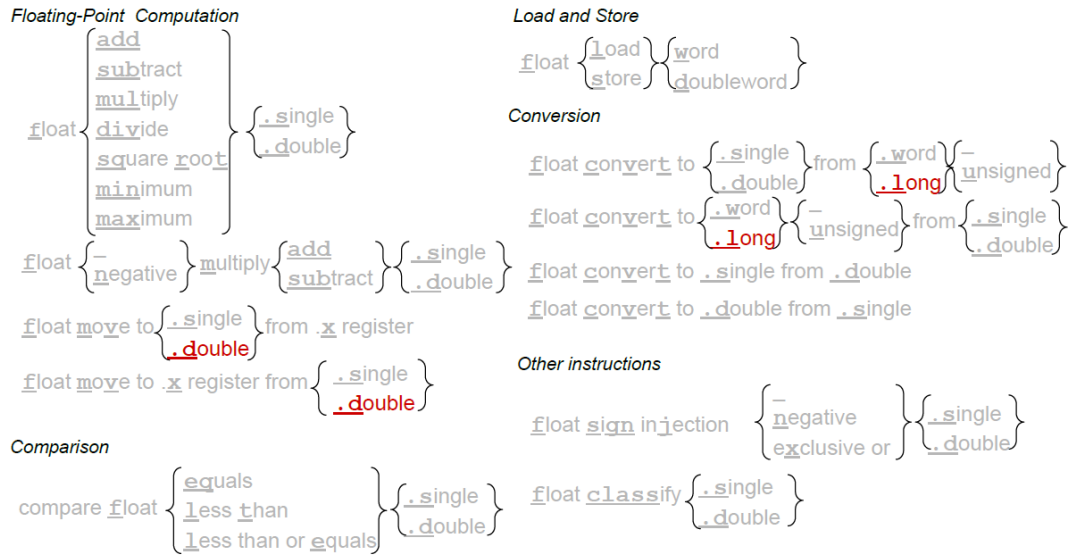


图 9.3: RV64F 和 RV64D 指令图

## RV64C

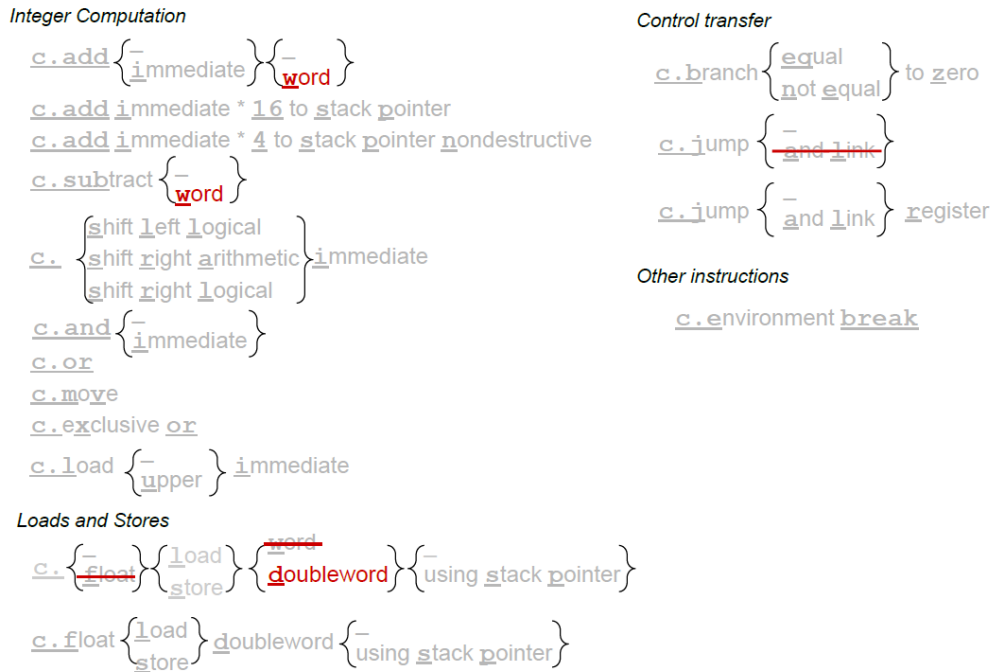


图 9.4L: RV64C 指令图

31	27	26	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		011		imm[4:0]		0100011			S sd
0000000					shamt		001		rd		0010011			I slli
0000000					shamt		101		rd		0010011			I srli
0100000					shamt		101		rd		0010011			I srai
imm[11:0]					rs1		000		rd		0011011			I addiw
0000000					shamt		001		rd		0011011			I slliw
0000000					shamt		101		rd		0011011			I srliw
0100000					shamt		101		rd		0011011			I sraiw
0000000					rs2		000		rd		0111011			R addw
0100000					rs2		000		rd		0111011			R subw
0000000					rs2		001		rd		0111011			R sllw
0000000					rs2		101		rd		0111011			R srlw
0100000					rs2		101		rd		0111011			R sraw

#### RV64M Satndard 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 Satndard 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 Satndard Extension (in addition to RV32F)

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

#### RV64D Satndard Extension (in addition to RV32D)

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

图 9.5: RV64 基本指令和可选扩展指令的操作码表。这张图包含了指令布局, 操作码, 格式类型和名称(基于[Waterman and Asanovic 2017]的表 19.2)。

RV64F 和 RV64D 添加了整数双字转换指令，并称它们为长整数，以避免与双精度浮点数据混淆：fcvt.l.s, fcvt.l.d, fcvt.lu.s, fcvt.lu.d, fcvt.s.l, fcvt.s.lu, fcvt.d.l, fcvt.d.lu。由于整数 x 寄存器现在是 64 位宽，它们现在可以保存双精度浮点数据，因此 RV64D 增加了两个浮点指令：fmv.x.w 和 fmv.w.x。

RV64 和 RV32 之间基本是超集关系，但是有一个例外是压缩指令。RV64C 取代了一些 RV32C 指令，因为其他一些指令对于 64 位地址可以取得更好的代码压缩效果。RV64C 放弃了压缩跳转并链接（c.jal）和整数和浮点加载和存储字指令（c.lw, c.sw, c.lwsp, c.swsp, c.flw, c.fsw, c.flwsp 和 c.fswsp）。在他们的位置，RV64C 添加了更受欢迎的字加减指令（c.addw, c.addiw, c.subw）以及加载和存储双字指令（c.ld, c.sd, c.ldsp, c.sdsp）。

---

补充说明：RV64 ABI 包括 lp64, lp64f 和 lp64d

lp64 表示 C 语言中的长整型以及指针类型为 64 位；整型仍然是 32 位。与 RV32（见第 3 章）相同，后缀 f 和 d 表示如何传递浮点参数。

---

补充说明：RV64V 没有指令图示

是因为有动态寄存器类型的存在，它与 RV32V 的完全一致。唯一的变化是第 75 页的图 8.2 中的 X64 和 X64U 动态寄存器类型仅在 RV64V 中出现，RV32V 并不支持。

---

## 9.2 使用插入排序来比较 RV64 与其他 64 位 ISA

正如 Gordon Bell 在本章开头所说，一个致命的架构缺陷是用光了地址位。随着程序使用的内存大小逐渐逼近 32 位地址空间的极限，不同指令集的架构师开始了设计他们指令集的 64 位地址版本[Mashey 2009]。

最早的是 MIPS，在 1991 年，它将所有寄存器以及程序计数器从 32 扩展至 64 位并添加了新的 64 位版本的 MIPS-32 指令。MIPS-64 汇编语言指令都以字母“d”开头，例如 daddu 或 dsll（参见图 9.10）。程序员可以在同一个程序中混合使用 MIPS-32 和 MIPS-64 指令。MIPS-64 删除了 MIPS-32 中的加载延迟槽（流水线在侦测到写后读依赖时会停止）。

十年之后，是 x86-32 指令集也迎来了 64 位。架构师们在拓展地址空间的同时，也借机在 x86-64 中进行了一系列改进：

- 整数寄存器的数量从 8 增加到 16（r8-r15）；
- 将 SIMD 寄存器的数量从 8 增加到 16（xmm8-xmm15）并且添加了 PC 相关数据寻址，以更好地支持与位置无关的代码。
- 添加了 PC 相关数据寻址，以更好地支持与位置无关的代码。

这些改进部分缓和了 x86-32 指令集长久以来的一些弊端。通过比较插入排序的 x86-32 版本（第 2 章第 30 页上图 2.11 中）和 x86-64 版本（图 9.11 中）的指令，我们可以发现 x86-64 指令集的优势。新的 64 位 ISA 将所有变量分配在寄存器中，而不是像 x86-32 一样，要将多个变量分配到内存中，这将指令的数量从 20 条减少到了 15 条。尽管 64 位代码指令数量比 32 位少，但是代码大小实际上要大一个字节，从 45 变成了 46 字节。原因是为了挤进新的操作码以便使用更多寄存器，x86-64 添加了一个前缀字节来识别新指令。从 x86-32 到

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

图 9.6: 四个 ISA 的插入排序的指令数和代码大小。ARM Thumb-2 和 microMIPS 是 32 位地址 ISA，因此不适用于 ARM-64 和 MIPS-64。

x86-64 平均指令长度增长了。

又过了十年，ARM 也遇到了同样的地址问题。但是他们没有像 x86-64 那样，把旧的 ISA 扩展到支持 64 位地址。他们利用这个机会发明了一个全新的 ISA。从头设计一个新 ISA，使得他们不必继承 ARM-32 的许多尴尬特性，他们重新设计了一个现代 ISA：

- 将整数寄存器的数量从 15 增加到 31；
- 从寄存器组中删除 PC；
- 为大多数指令提供硬连线到零的寄存器（r31）；
- 与 ARM-32 不同，ARM-64 的所有数据寻址模式都适用于所有数据大小和类型；
- ARM-64 去除了 ARM-32 的加载存储多个数据的指令
- ARM-64 去除了 ARM-32 指令的条件执行选项。

ARM-32 的一些弱点依然存在于 ARM-64 指令集中：分支指令使用的条件码，指令中源和目标寄存器字段并不固定，条件移动指令，复杂寻址模式，不一致的性能计数器，以及只支持 32 位长度的指令。另外 ARM-64 无法切换到 Thumb-2 ISA，因为 Thumb-2 仅适用于 32 位地址。

与 RISC-V 不同，ARM 决定采用最大限度的方法来设计 ISA。虽然 ISA 比 ARM-32 更好，但它也更大。例如，它有超过 1000 条指令并且 ARM-64 手册长 3185 页[ARM 2015]。而且，它的指令数仍然在增长。自公布几年以来，ARM-64 已经经历了三次扩展。

图 9.9 中插入排序的 ARM-64 代码看起来更接近 RV64I 代码或 x86-64 代码，而不太像 ARM-32 代码。例如，因为有 31 个寄存器可用，就没有必要从堆栈中保存和恢复寄存器。而且由于 PC 不再存放于通用寄存器中，ARM-64 单独增加了一条返回指令。

图 9.6 总结了插入排序在不同 ISA 下的指令数和字节数。图 9.8 到 9.11 显示了 RV64I，ARM-64，MIPS-64 和 x86-64 的代码。这四段代码注释中括号内的短语阐明了第 2 章中的 RV32I 版本与这些 RV64I 版本之间的差异。？ ？ ？

MIPS-64 需要最多的指令，主要是因为它需要使用 nop 指令来填充无法有效利用起来的分支延迟槽。由于比较和分支用一条指令完成，而且分支指令没有延迟槽，RV64I 需要的指令更少。虽然相较于 RV64I，对于每个分支，ARM-64 和 x86-64 需要多使用两条指令，但它们的缩放寻址模式避免了 RV64I 中所需的地址算术指令，可以让它们少使用一些指令。但是总的而言，RV64I + RV64C 代码大小要小得多，具体原因会在下一节阐述。

英特尔没有发明 x86-64 ISA。当向 64 位地址转换时，英特尔发明了一种名为 Itanium 的新 ISA，它与 x86-32 不兼容。x86-32 处理器的竞争对手被拦在了 Itanium 的门外，因此 AMD 发明了一款名为 AMD64 的 64 位版 x86-32。Itanium 最终失败了，因此英特尔被迫采用 AMD64 ISA 作为 x86-32 的 64 位地址继承者，我们称之为 x86-64 [Kerner & Padgett 2007]。

补充说明：ARM-64，MIPS-64 和 x86-64 不是官方名称

他们的官方名称是：我们所说的 ARM-64 实则是 ARMv8，MIPS-64 是 MIPS-IV 和 x86-64 是 AMD64（有关 x86-64 的历史记录，请参见上一頁的侧栏）。

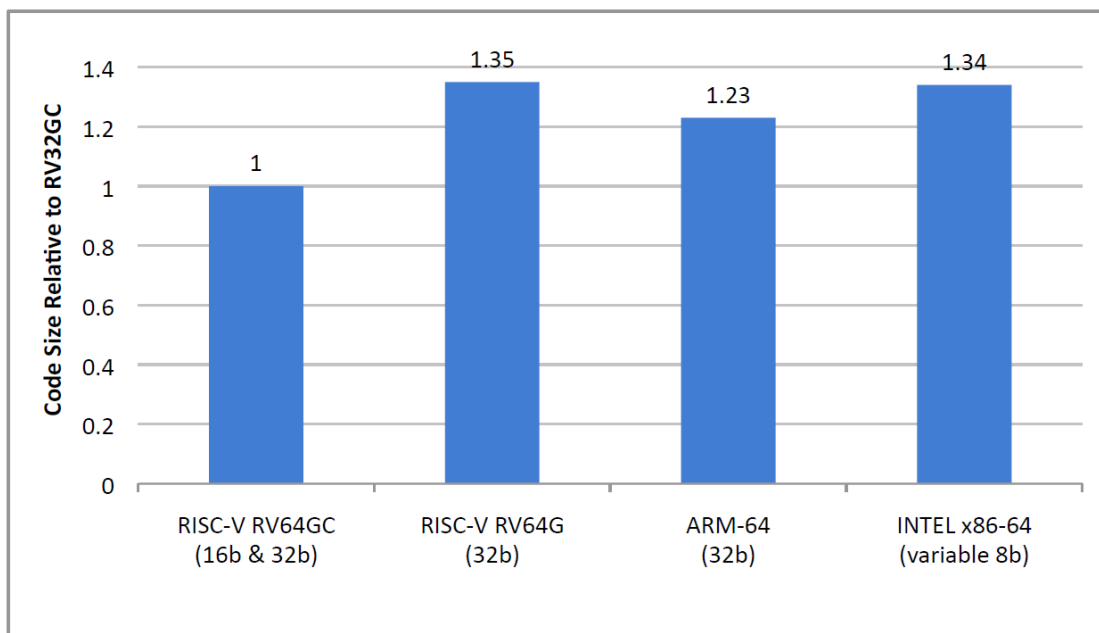


图 9.7: RV64G, ARM-64 和 x86-64 与 RV64GC 的相对程序大小比较。我们使用了比图 9.6 中更大的程序来进行对比。该图第 2 章中第 9 页的图 1.5 中的 32 位 ISA 比较的对应的 64 位 ISA 比较。RV32C 代码大小与 RV64C 几乎一致;仅比 RV64C 小 1%。ARM-64 没有 Thumb-2 选项,因此其他 64 位 ISA 的代码大小明显大于 RV64GC 代码。测量的程序是 SPEC CPU2006 基准测试,使用的 GCC 编译器[Waterman 2016]。

### 9.3 程序大小

图 9.7 比较了 RV64, ARM-64 和 x86-64 的平均相对代码大小。将这个图见第 1 章第 9 页的图 1.5 比较。首先, RV32GC 代码的大小与 RV64GC 几乎相同;它只比 RV64GC 小 1%。RV32I 和 RV64I 的代码大小也很接近。而 ARM-64 代码比 ARM-32 代码小 8%, 由于没有 64 位地址版本的 Thumb-2, 所以所有指令都保持 32 位长。因此, ARM-64 代码比 ARMThumb-2 代码大 25%。由于添加了前缀操作码以装下新的指令以及扩展的寄存器, x86-64 的代码比 x86-32 代码大 7%。因此, 就程序大小而言, RV64GC 更优秀, 因为 ARM-64 代码比 RV64GC 大 23%, x86-64 代码比 RV64GC 大 34%。程序大小的差异如此得大, 以至于 RV64 可以较低的指令高速缓存缺失率来提供更高的性能, 或者可以使用更小的指令缓存来降低成本, 但依然能提供令人满意的缺失率。

### 9.4 结束语

成为先驱者的一个问题是你总是犯错误, 而我永远不会想成为先驱者。最好是在看到先驱者所犯的错误后, 赶紧来做这件事情, 成为第二个做这件事情的人。

——Seymour Cray, 第一台超级计算机的架构师, 1976 年

MIPS 正在出售。Imagination Technologies 于 2012 年以 1 亿美元收购了 MIPS ISA, 最近宣布将其出售; 还没有买家。

耗尽地址位是计算机体系结构的致命弱点。许多架构因为这个缺点而小王。ARM-32 和 Thumb-2 仍然是 32 位架构, 所以他们对大型程序没有帮助。像 MIPS-64 和 x86-64 这样的一些 ISA 在转型中幸存下来, 但 x86-64 并不是 ISA 设计的典范, 而写这篇文章的时候, MIPS-64 的前路依然迷茫。ARM-64 是一个新的大型 ISA, 时间会告诉我们它会有多成功。



RISC-V 受益于同时设计 32 位和 64 位架构，而较老的 ISA 必须按照顺序设计它们。不出所料，对于 RISC-V 程序员和编译器编写者来说，32 位到 64 位之间的过渡是最简单的；RV64I ISA 几乎包含了所有 RV32I 指令。这也就是为什么我们只用两页参考卡片，就可以列出 RV32GCV 和 RV64GCV 指令集。更重要的是，同步设计意味着 64 位架构指令集不必被狭窄的 32 位操作码空间限制。RV64I 有足够的空间用于可选的指令扩展，特别是 RV64C，这使它成为代码大小比其他所有 64 位 ISA 都要小。

我们认为 64 位架构更能体现 RISC-V 设计上的优越性，毕竟我们设计 64 位 ISA 比先行者们晚了 20 年，这样我们可以学习先行者们的好的设计并从他们的错误中吸取教训。

---

#### 补充说明：RV128

RV128 最初是作为 RISC-V 架构师内部的一个玩笑话，只是为了显示 128 位地址的 ISA 是可能的。但是，仓库规模的计算机可能很快就会拥有超过  $2^{64}$  字节存储容量的半导体存储器（DRAM 和闪存），同时程序员可能会想要用访问内存的方式来访问这些存储。同时还有人[伍德拉夫等人 2014]提议使用 128 位地址来提高安全性。RISC-V 手册确实指定了一个完整的 128 位 ISA 叫做 RV128G [Waterman and Asanovic 2017]。如图 9.1 至 9.4 所示，新增的指令基本上是与从 RV32 切换到 RV64 新增的指令类似。所有的寄存器也增长到 128 位，新的 RV128 指令要么指定了 128 位版本的指令（指令名称中使用 Q，意为四字（quadword））或其他指令的 64 位版本（指令名称中使用 D，意为双字（double word）的）。

---

## 9.5 了解更多

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/o6au/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.



```

# 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 of 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

```

图 9.8: 图 2.5 中插入排序的 RV64I 代码。RV64I 汇编语言程序与第 2 章中第 27 页的图 2.8 中的 RV32I 汇编语言程序类似。我们在注释中的括号内列出了差异。数据的大小现在是 8 个字节而不是 4 个，所以三条指令中的常数从 4 变到了 8。由于数据宽度的变化，两个加载字（lw）相应变成了加载双字（ld）和两个存储字（sw）相应变成了存储双字（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: 17ffffff2  b    4                       # jump to Outer Loop

```

图 9.9：图 2.5 所示的插入排序的 ARM-64 代码。ARM-64 汇编语言程序是第 2 章中第 30 页图 2.11 中的 ARM-32 汇编语言不同，它是一套全新的指令系统。寄存器以 x 开始而不是以 a 开头。数据寻址模式支持将寄存器移位 3 位用于将索引缩放为字节地址。使用 31 个寄存器，所以无需保存和恢复寄存器堆栈。由于 PC 不是寄存器之一，因此它使用了单独的返回指令。事实上，代码看起来更接近 RV64I 代码或 x86-64 代码而不是 ARM-32 代码。

```

# 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) decre 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 of 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 # Unnecessary(?)

```

图 9.10: 图 2.5 中所示的插入排序的 MIPS-64 代码。MIPS-64 汇编语言程序与第 2 章第 29 页图 2.10 中的

MIPS-32 汇编语言有一些不同之处。首先，对于 64 位数据的大多数操作都在其名称前加上“d”：

daddiu, daddu, dsll。如图 9.8，由于数据大小从 4 字节增加到 8 字节，因此有三条指令将常量从 4 更改为 8。再次与 RV64I 类似，增加的数据宽度，使得两个加载字 (lw) 变成了加载双字 (ld)，两个存储字

(sw) 变成了存储双字 (sd)。最后，MIPS-64 没有了 MIPS-32 中的加载延迟槽；当出现写后读依赖时，

流水线会阻塞。

```

# 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

```

图 9.11: 图 2.5 中插入排序的 x86-64 代码。 x86-64 汇编语言程序与第 2 章中第 30 页图 2.11 中的 x86-32 汇编语言非常不同。首先, 与 RV64I 不同, 较宽的寄存器有不同的名称 rax, rcx, rdx, rsi, rdi, r8。第二, 因为 x86-64 增加了 8 个寄存器, 现在可以将所有变量保存在寄存器而不是内存中。第三, x86-64 指令比 x86-32 更长, 因为许多指令需要预先添加 8 位或 16 位前缀码, 才能使得操作码空间中放得下这些新指令。例如, 递增或递减寄存器 (inc, dec) 在 x86-32 中只需要 1 字节, 但 x86-64 中需要 3 个字节。因此, 对于 Insertion Sort, 虽然 x86-64 指令数比 x86-32 少, 但代码大小为几乎与 x86-32 相同: 45 个字节对 46 个字节。