RISC-V 指令集手册

卷 1：用户级指令集体系结构（User-Level ISA）

2.1 版

Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanović
CS Division, EECS Department, University of California, Berkeley
{waterman|yunsup|pattrsn|krste}@eecs.berkeley.edu

2016 年 5 月 31 日

该文档同时也是 UCB/EECS-2016-118 技术报告

序言

这是描述 RISC-V 用户级体系结构文档的 2.1 版。注意已经冻结的基本用户级 ISA 和 2.0 版的 IMAFDQ 扩展从该文档的上一个版本[30]以来并没有发生变化，但是填充了一些规范的“空洞”以及改善了文档。对软件约定做了一些改变。

- 对注释部分做了大量地添加和改进。
- 每一章有单独的版本号。
- 修改了＞64 位的长指令编码，以避免在非常长的指令格式里移动 ***rd*** 区分符。
- 现在使用基本整数格式来描述 CSR 指令，引入了计数器寄存器，不同于（以前版本）仅在后面的浮点部分（和特权体系结构手册中）引入。
- SCALL 指令和 SBREAK 指令被分别重命名为 ECALL 指令和 EBREAK 指令。它们的编码和功能并没有改变。
- 澄清了浮点 NaN 的处理，以及一个新的规定的 NaN 值。
- 澄清了浮点到整数转换溢出时的返回值。
- 澄清了 LR/SC 允许的成功和要求的失败，包括在序列中使用压缩指令。
- 一个新的 RV32E 基本 ISA 提案，可减少整数寄存器数量。
- 修订了调用约定。
- 放松了软浮点调用约定的栈对齐，描述了 RV32E 调用约定。
- 一个修订的 C 压缩扩展提案，版本 1.9。

2.0 版的序言

用户指令集体系结构规范的第二个发布版本，我们试图保持这个基本的用户 ISA 加上通用扩展（就是 IMAFD），在未来版本中保持固定不变。从这个 ISA 的 1.0 版本[29]以来，有如下变化：

- ISA 被划分为一个整数基本内核和几个标准扩展。
- 重新组织了指令格式，使得立即数编码更加高效。
- 基本的 ISA 被定义为拥有一个小端（little-endian）的存储器系统，而大端、双端作为非标准的变种。
- Load-Reserved/Store-Conditional（LR/SC）指令被添加进原子指令集扩展。
- AMO 和 LR/SC 指令可以支持释放一致性模型（release consistency model）。
- FENCE 指令提供了细粒度的存储器和 I/O 序列化（orderings）。
- 加入了 fetch-and-XOR 的 AMO（AMOXOR），对 AMOSWAP 的编码进行了修改，以便留出空间。
- 将 20 位立即数加到 PC 上的 AUIPC 指令，替换了 RDNPC 指令，AUIPC 指令只读取当前的 PC 值。这导致对位置无关代码（position-independent code）的大量简化。
- JAL 指令现在被移动到 U 类型格式，具有一个显示的（explicit）目标寄存器，而 J 指令被 ***rd=x0*** 的 JAL 指令所代替。这个改变，消除了唯一一条需要隐式（implicit）目标寄存器的指令，并且从标准 ISA 中去掉了 J 类型指令格式。这虽然是 JAL 指令的一个附加效果，但是却极大地减少了基本 ISA 的复杂性。
- 去掉了 JALR 指令的静态提示（static hints）。对于使用标准调用约定编译的代码来说，这些提示和 ***rd***、***rs1*** 寄存器是冗余的。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

- JALR 指令现在清除了计算出来的目标地址的最低位，以简化硬件并允许在函数指针中存储附加信息。
- MFTX.S、MFTX.D 指令被分别命名为 FMV.X.S、FMV.X.D 指令，MXTF.S、MXTF.D 指令被分别命名为 FMV.S.X、FMV.D.X 指令。
- MFFSR、MTFSR 指令被分别命名为 FRCSR、FSCSR 指令，新增了 FRRM、FSRM、FRFLAGS 和 FSFLAGS 指令，用于独立地访问 **fcsr** 寄存器的舍入模式和异常标志。
- FMV.X.S、FMV.X.D 指令现在使用 ***rs1*** 作为源操作数，而不是 ***rs2***。这样可以简化数据通路设计。
- 新增了 FCLASS.S、FCLASS.D 浮点指令。
- 采用了一种更简单的 NaN 生成和传播模式。
- 对于 RV32I，系统性能计数器被扩展成 64 位宽度，可以单独读取高 32 位和低 32 位。
- 定义了规定的（Canonical）NOP 和 MV 指令编码。
- 对于 48 位、64 位和大于 64 位指令，定义了标准指令长度编码。
- 新增了一个 128 位地址空间变种 RV128 的描述。
- 32 位基本指令格式中的大部分操作码分配给用户自定义的定制扩展。
- 一个印刷错误被纠正：store 的源操作数来源于 ***rd***，其实应该来源于 ***rs2***。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

目录

第 1 章 介绍 1
1.1 RISC-V ISA 概述 3
1.2 指令长度编码 5
1.3 异常、自陷和中断 6
第 2 章 RV32I 基本整数指令集，2.0 版 8
2.1 基本整数子集的程序员模型 8
2.2 基本指令格式 9
2.3 立即数编码变种 10
2.4 整数计算指令 11
整数寄存器-立即数指令 12
整数寄存器-寄存器操作 13
NOP 指令 14
2.5 控制转移指令 14
无条件跳转 14
条件分支 15
2.6 Load 和 store 指令 17
2.7 存储器模型 18
2.8 控制和状态寄存器指令 20
CSR 指令 20
定时器和计数器 22
环境调用和断点 23
第 3 章 RV32E 基本整数指令集，版本 1.9 24
3.1 RV32E 程序员模型 24
3.2 RV32E 指令集 24
3.3 RV32E 扩展 25
第 4 章 RV64I 基本整数指令集，版本 2.0 26
4.1 寄存器状态 26
4.2 整数计算指令 26
整数寄存器-立即数指令 26
整数寄存器-寄存器操作 28
4.3 Load 和 Store 指令 28
4.4 系统指令 29
第 5 章 整数乘法除法的“M”标准扩展，版本 2.0 30
5.1 乘法操作 30
5.2 除法操作 30
第 6 章 原子性指令的“A”标准扩展，版本 2.0 32
6.1 原子性操作的指定顺序 32
6.2 Load-reserved/store-conditional 指令 33
6.3 原子性存储器操作 36
第 7 章 单精度浮点的“F”标准扩展，版本 2.0 38
7.1 F 寄存器状态 38
7.2 浮点控制和状态寄存器 39

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

7.3 NaN 生成和传递......40
7.4 非规格化数算术......41
7.5 单精度 load 和 store 指令......41
7.6 单精度浮点计算指令......41
7.7 单精度浮点转换和传输指令......42
7.8 单精度浮点比较指令......44
7.9 单精度浮点分类指令......44
第 8 章 双精度浮点的“D”标准扩展，版本 2.0......46
8.1 D 寄存器状态......46
8.2 双精度 load 和 store 指令......46
8.3 双精度浮点计算指令......47
8.4 双精度浮点转换和传输指令......47
8.5 双精度浮点比较指令......49
8.6 双精度浮点分类指令......49
第 9 章 RV32/64G 指令集列表......50
第 10 章 扩展 RISC-V......55
10.1 扩展术语......55
标准 vs. 非标准扩展......55
指令编码空间和前缀......55
绿地扩展 vs. 棕地扩展......56
标准兼容的全局编码......57
保证的非标准编码空间......57
10.2 RISC-V 扩展设计理念......57
10.3 在固定 32 位指令格式内的扩展......58
可用的 30 位指令编码空间......58
可用的 25 位指令编码空间......58
可用的 22 位指令编码空间......58
其他空间......58
10.4 增加对齐的 64 位指令扩展......59
10.5 支持 VLIW 编码......59
定长指令组......59
编码长度指令组......59
固定大小指令束......59
前缀中“组结束位”......60
第 11 章 ISA 子集命名约定......61
11.1 大小写敏感......61
11.2 基本整数 ISA......61
11.3 指令扩展名字......61
11.4 版本号......61
11.5 非标准扩展命名......62
11.6 管理员级指令子集......62
11.7 管理员级扩展......62
11.8 子集命名约定......62
第 12 章 四精度浮点的“Q”标准扩展，版本 2.0......64
Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

12.1 四精度 load 和 store 指令 ...64
12.2 四精度计算指令 ...64
12.3 四精度转换和传输指令 ...65
12.4 四精度浮点比较指令 ...66
12.5 四精度浮点分类指令 ...67
第 13 章 十进制浮点的“L”标准扩展，版本 0.0 ...68
13.1 十进制浮点寄存器 ...68
第 14 章 压缩指令的“C”标准扩展，版本 1.9 ...69
14.1 概述 ...69
14.2 压缩指令格式 ...71
14.3 Load 和 store 指令 ...72
基于栈指针的 load 和 store ...73
基于寄存器的 Load 和 store ...74
14.4 控制转移指令 ...75
14.5 整数计算指令 ...76
整数常数-生成指令 ...76
整数寄存器-立即数指令 ...77
整数寄存器-寄存器指令 ...79
预定义非法指令 ...80
NOP 指令 ...80
断点指令 ...81
14.6 在 LR/SC 序列中使用 C 指令 ...81
14.7 RVC 指令集列表 ...81
14.8 指令压缩统计 ...85
14.9 优化寄存器保存/恢复代码大小 ...88
第 15 章 向量操作的“V”标准扩展，版本 0.0 ...90
第 16 章 位操作的“B”标准扩展，版本 0.0 ...91
第 17 章 事务存储器的“T”标准扩展，版本 0.0 ...92
第 18 章 打包 SIMD 指令的“P”标准扩展，版本 0.1 ...93
第 19 章 RV128I 基本整数指令集，版本 1.7 ...95
第 20 章 调用约定 ...96
20.1 C 数据类型和对齐 ...96
20.2 RVG 调用约定 ...96
20.3 软浮点调用约定 ...99
20.4 RV32E 调用约定 ...99
第 21 章 RISC-V 汇编程序员手册 ...100
第 22 章 历史和致谢 ...102
22.1 到 ISA 手册 1.0 版本以前的历史 ...102
22.2 从 ISA 手册 2.0 版本以来的历史 ...102
致谢 ...104
22.3 版本 2.1 的历史 ...104
致谢 ...104
22.4 资助 ...104
参考文献 ...106
Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第1章 介绍

RISC-V（读音“risk-five”）是一个新的指令集体系结构（ISA），它最初用于支持计算机体系结构研究和教学，但现在我们希望它也成为一个对于工业实现来说标准、免费、开放的体系结构。我们定义 RISC-V 的目的包括：

- 一个完全***开放的*** ISA，能够自由地提供给学术界和工业界使用。
- 一个***真正的*** ISA，能够适合直接在硬件上实现，而不仅仅是适用于模拟或者二进制翻译。
- 一个避免对某一种微体系结构风格（例如微编码、按序、去耦合、乱序等）或者实现技术（例如全定制、ASIC、FPGA）“过度体系结构化（over-architecting）”的 ISA，但是也能够非常高效地利用任何一种技术实现。
- 包含一个***小的***基本整数 ISA（可以作为一个定制的加速器的基础或者作为教学用途）和多个可选的标准扩展的 ISA，可以支持通用的软件开发。
- 支持修订的 2008 IEEE-754 浮点标准[10]。
- ISA 支持丰富的用户级 ISA 扩展和各种特殊的变种。
- 对应用程序、操作系统内核、硬件实现的 32 位、64 位地址空间变种。
- ISA 支持高度并行的多核、众核实现，包括异构多处理器等。
- 可选的***变长指令***，以支持扩展可用的指令编码空间、支持一个可选的***密集指令编码***，以提高性能、静态代码大小和能耗效率。
- 一个可完全虚拟化的 ISA，以简化虚拟机监督管理器（Hypervisor）的开发。
- ISA 支持新的管理员级（supervisor-level）和虚拟机监督管理级（hypervisor-level）ISA 设计。

我们的设计考虑，将出现在类似的文本段落内，如果读者只关心规范，则可以跳过这些段落。

RISC-V 这个名字，代表了 UC Berkeley 大学设计的第五代主要的 RISC ISA（前四个是 RISC-I[18]、RISC-II[11]、SOAR[27]和 SPUR[14]）。罗马数字“V”也暗示了“变种（Variations）”和“向量（Vectors）”，以支持各种体系结构研究，包括各种数据并行加速器，也是这个 ISA 设计的明确目标。

我们研发 RISC-V 以满足我们自己的科研和教学需求，我们对如何在真实硬件上实现一些研究思想特别感兴趣（自从这个规范的第一个版本发布之后，我们已经完成了 11 块不同的 RISC-V 硅片的制造），在课堂上提供给学生真实的实现（在 Berkeley，RISC-V 处理器的 RTL 设计代码已经用于多个本科生、研究生的课程）。在我们当前的研究中，由于传统晶体管不断变小带来的能耗约束，我们对特殊、异构的加速器特别感兴趣。我们需要一个高度灵活、高度可扩展的基本 ISA，在此基础上可以构建我们自己的研究。

我们总被问及这样一个问题“为什么要开发一个新的 ISA？”。使用一个已有的商业化的 ISA，其显而易见最大的优势在于其已经具备了丰富和广泛支持

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

的软件生态系统，包括开发工具和可移植的应用程序，而在研究和教学中，这些都是可以利用的。其他的好处包括拥有大量的文档和教程示例。然而，我们的经验证明，在科研和教学中使用商业的指令集，在实际中获得的好处很小，而且掩盖不了它的缺点：

- **商业 ISA 都是私有的。**除了 SPARC V8（它是一个开放的 IEEE 标准[1]），绝大多数 ISA 的拥有者非常小心地保护他们的知识产权，并且并不欢迎自由实现的竞争实现。对于仅仅使用软件模拟器来进行学术研究和教学来说，这并不是一个问题，但是对于那些希望分享真实硬件实现的科研小组来说，这就是一个大问题。对于那些被强迫信任仅有的几个商业 ISA 实现，而不允许创建自己的全新实现（clean room implementation）的企业来说，这也是一个大问题。我们并不能确保所有的 RISC-V 实现没有侵犯第三方专利，但是我们确保我们绝不会起诉一个 RISC-V 的实现者。
- **商业 ISA 仅仅在某个市场领域比较流行。**当书写此文档时，最显而易见的例子就是 ARM 体系结构在服务器领域并没有得到很好的支持，而 Intel x86 体系结构（或者几乎任何一种其他的体系结构）在移动领域并没有得到很好的支持，虽然 Intel 和 ARM 正在试图进入对方的市场领域。另外一个例子是 ARC 和 Tensilica，它们提供了可扩展的内核，但是只关注嵌入式市场。这种市场的划分，使得支持某种特定商业 ISA 获得的好处大大削弱，因为事实上软件生态系统只存在于某个领域，到了别的领域，必须重新构建。
- **商业 ISA 此起彼伏。**以前基于商业 ISA 构建的研究基础设施，并不流行（SPARC、MIPS），甚至不再生产（Alpha）。这对于一个活跃的软件生态系统来说是一个大损失，一些围绕 ISA 和支持工具的知识产权问题，也使得感兴趣的第三方难以继续支持这个 ISA。一个开放的 ISA 也可能失去流行性，但是任何感兴趣的人，都可以继续使用它并研发相应的生态系统。
- **流行的商业 ISA 是复杂的。**占统治地位的 ISA（x86 和 ARM）若要支持常用软件栈和操作系统，那么其硬件实现都非常复杂。更糟糕的是，几乎所有的复杂性都来自于糟糕的、或者至少是过时的ISA 设计考虑，而不是那些真正提高效率的特性。
- **仅靠商业 ISA 并不足以运行应用程序。**即使我们努力实现了一个商业 ISA，对于运行一个现有的应用程序来说，仍然是不够的。绝大多数应用程序需要一个完整的 ABI（application binary interface）才能运行，而不仅仅是用户级 ISA。绝大多数 ABI 依赖于库（libraries），而库又依赖于操作系统支持。为了运行一个已有的操作系统，需要实现管理员级 ISA、OS 需要的设备接口。这些通常并没有很好的规范，而在实现上比用户级 ISA 具有更大的复杂性。
- **流行的商业 ISA 不是为可扩展性设计的。**占统治地位的商业 ISA 并没有为可扩展性而进行特殊的设计，结果就是，随着后续指令集不断地增长，指令编码的复杂度大幅度增加。而类似 Tensilica（被 Cadence 公司收购）、ARC（被 Synopsys 公司收购）这样的公司，它们围绕可扩展性构建了 ISA 和工具链（toolchain），但是它们瞄准的是嵌入式应用而不是通用计算系统。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

- ***一个修改过的商业 ISA 实际上是一个新的 ISA。****我们的一个主要目标是支持体系结构研究，包括主要的 ISA 扩展。即使是很小的扩展，也减弱了使用标准 ISA 而带来的好处，因为必须修改编译器，而应用程序必须从源代码进行重新编译，以利用这些扩展。引入了新的体系结构状态的大一些的扩展，也需要对操作系统进行修改。最终使得一个修改的商业 ISA 变成一个新的 ISA，但是不得不肩负着所有基本 ISA 遗留下来的包袱。*

我们坚信 ISA 是整个计算系统中最重要的接口，没有理由把这么重要的接口变成私有的。占统治地位的商业 ISA 都是基于超过 30 年历史的指令集。软件开发者应当能够定位到一个开放标准的硬件目标机，商业处理器设计者应当在实现质量上进行竞争。

我们并不是第一个为了适合硬件实现而提出开放 ISA 设计的。我们也考虑了其他现有的开放 ISA 设计，其中 OpenRISC 体系结构[17]与我们的目标最为接近。我们由于几个技术原因，并不采用 OpenRISC ISA：

- *OpenRISC 有条件码（condition code）和分支延迟槽（branch delay slot），这对于更高性能的实现来说，变得更为复杂。*
- *OpenRISC 使用了 32 位定长指令编码和 16 位立即数，阻碍了更密集的指令编码，并对后续 ISA 扩展限制了空间。*
- *OpenRISC 并不支持 2008 修订的 IEEE-754 浮点标准。*
- *在我们开始的时候，64 位 OpenRISC 设计并没有完成。*

从零开始，我们可以设计一个符合我们所有需求的 ISA，当然，这花了比我们在开始时预期多得多的努力。现在我们在构建 RISC-V ISA 基础设施上投入了大量的精力，包括文档、编译器工具链、操作系统移植、参考 ISA 模拟器、FPGA 实现、高效的 ASIC 实现、体系结构测试套件、教学材料等。自本文档的上一个版本以来，在学术界和工业界对此 RISC-V ISA 都有大量的吸收（uptake），我们也创建了非盈利的 RISC-V 基金会来保护和推进这个标准。RISC-V 基金会的网址在 http://riscv.org，包含了基金会成员最新的信息和各种各样使用 RISC-V 的开源项目。

RISC-V 手册被分为两卷。本卷涵盖了用户级 ISA 设计，包括可选的 ISA 扩展。第二卷给出了特权体系结构。

在这个用户级手册中，我们的目标是移除所有与特定微体系结构特性或者管特权体系结构相关的细节。这样做，主要是为了清晰，并最大程度地允许其他实现的灵活性。

1.1 RISC-V ISA 概述

RISC-V ISA 被定义为一个基本的整数 ISA，必须在任何实现中存在，另外可以包含基于基本 ISA 的其他扩展。这个基本的整数 ISA 与早期的 RISC 处理器非常相似，除了没有分支延迟槽（delay slot），另外支持可选的变长指令编码。这个基本核心被小心地限制具有最少的指令，足够支持一个合理的目标机，以便编译器、汇编器、链接器、操作系统（包含额外的管

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

理员级操作）可以在之上运行，这样就可以提供一个方便的 ISA 和软件工具链“骨架”，围绕它可以构建更为定制化的处理器 ISA。

每一个基本整数指令集，被整数寄存器宽度和相应的用户地址空间大小进行分类。有两种主要的基本整数变种，RV32I 和 RV64I，在第 2 章和第 4 章中进行描述，分别提供了 32 位和 64 位用户级地址空间。硬件实现和操作系统可以提供给用户程序使用 RV32I 或者 RV64I 中的一种或者两种。第 3 章描述了 RV32I 基本指令集的子集变种 RV32E，它被加入以支持小的微控制器。第 19 章描述了未来支持 128 位用户地址空间的 RV128I 变种基本整数指令集。

虽然对于更大的系统来说，64 位地址空间是必须的，但我们相信在未来的数十年中，对于许多嵌入式应用和客户端设备来说，32 位地址空间是足够的，而且适合更低的存储器传输和能耗消耗。另外，32 位地址空间对于教育来说足够了。更大的 128 位地址空间也许最终也是必须的，于是我们确保在 RISC-V ISA 框架中也包含它。

一个硬件实现可以只实现基本整数 ISA 的一个子集，但是在更高特权层必须实现操作码自陷（trap）和软件仿真，用于实现那些硬件没有提供的功能。

基本整数 ISA 的一个子集对于教学目的来说，也许是有用的。但是基本核心被定义得非常简单，使得真实硬件实现没理由只实现它的一个子集，除了省略对非对齐存储器访问支持、以及将所有 SYSTEM 指令作为自陷实现之外。

RISC-V 被设计成可以支持丰富的定制化和特殊化。基本整数 ISA 可被一个或者多个可选指令集扩展进行增强，但是基本整数指令集不能被重新定义。我们将 RISC-V 指令集扩展分为***标准扩展***和***非标准扩展***。标准扩展一般都是有用的，并且与其它的标准扩展并不冲突。非标准扩展是高度特殊化的，并可能与其它的标准扩展或者非标准扩展冲突。指令集扩展根据基本整数指令集宽度不同，可能有轻微的功能差异。第 10 章描述了各种用来扩展 RISC-V ISA 不同的方法。我们为 RISC-V 基本指令和指令集扩展开发了一个命名规则，在第 11 章里有详细的描述。

为了支持更一般的软件开发，定义了一组标准扩展，提供乘法/除法、原子操作以及单精度、双精度浮点算术。基本整数 ISA 被命名为“I”（依据整数寄存器宽度不同，前缀 RV32 或者 RV64），其中包含了整数计算指令、整数 load、整数 store 和控制流指令，并且在所有 RISC-V 实现中，都是必须的。标准整数乘法和除法扩展被命名为“M”，其中增加了对保存在整数寄存器中的值进行乘法和除法的指令。标准原子指令扩展被命名为“A”，其中增加了对存储器进行原子的读、修改和写操作的指令，以支持处理器间的同步。标准单精度浮点扩展，被命名为“F”，增加了浮点寄存器、单精度计算指令、单精度 load 和 store 指令。标准双精度浮点扩展，被命名为“D”，扩展了浮点寄存器，并增加了双精度计算指令、load 和 store 指令。一个基本整数内核加上这四个标准扩展（“IMAFD”），被缩写为“G”，它提供了一个通用的标量指令集。RV32G 和 RV64G 现在是我们编译器工具链的缺省目标机器。后续章节描述了这些扩展以及其他计划中的标准 RISC-V 扩展。

除了基本整数 ISA 和标准扩展之外，很少有一条新指令对所有应用程序巨大的好处，虽然它可能在某些领域中非常有用。由于能耗效率要求更为特殊化，我们相信对于一个 ISA 规范中的必须部分的简化是很重要的。鉴于其他的体系结构通常将它们的 ISA 作为一个单一的整体，它们会随着时间推移，当加入新指令的时候，就变化到一个新的版本。然而 RISC-V 尝试随着时间的推移，保持基本内核和每一个标准扩展不变，相反的，将新指令作为可选的

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

扩展。例如，基本整数 ISA 将成为一个被持续支持的独立 ISA，而不管任何随后而来的扩展。

随着用户 ISA 规范的 2.0 版本发布，我们尝试在未来的开发中，保持“RV32IMAFD”、“RV64IMAFD”基本内核和标准扩展（也就是“RV32G”和“RV64G”）保持不变。

1.2 指令长度编码

基本 RISC-V ISA 具有 32 位固定长度指令，并且必须在 32 位边界对齐。然而，标准 RISC-V 编码模式被设计成支持变长指令的扩展，在这个扩展中，每条指令长度可以是 16 位指令包裹（parcel）长度的整数倍，并且这些指令包裹必须在 16 位边界对齐。第 14 章中描述的标准压缩 ISA 扩展，通过提供压缩的 16 位指令，减少了代码大小，并放松了对齐要求，允许所有指令（16 位和 32 位）对齐到任意 16 位边界，以提高代码密度。

图 1.1 展示了标准 RISC-V 指令长度编码约定。所有基本 ISA 中的 32 位指令的最低 2 位被设置为 11。可选的压缩 16 位指令集扩展中的指令，最低 2 位被设置为 00、01 或者 10。超过 32 位的标准指令集扩展，在低位有额外的位被设置为 1，48 位、64 位长度约定如图 1.1 所示。指令长度在 80 位到 176 位之间的长度信息，被编码到一个 3 位的字段[14:12]中，给出了 16 位字的数量，加上最开始的 5×16 位字。位[14:12]编码为 111，保留给未来更长的指令编码。

	基址+4	基址+2	基址	
			xxxxxxxxxxxxxxaa	16 位（aa≠11）
••• xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxxxxbbb11		32 位（bbb≠111）
••• xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxxx011111		48 位
••• xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxx0111111		64 位
••• xxxx	xxxxxxxxxxxxxxxx	xnnnxxxxx1111111		（80+16*nnn）位，nnnn≠1111
••• xxxx	xxxxxxxxxxxxxxxx	x111xxxxx1111111		保留给≥192 位

字节地址：基址+4　基址+2　基址

图 1.1：RISC-V 指令长度编码

由于压缩格式的代码大小和能耗节约，我们想将对压缩格式内建支持加入到 ISA 编码模式中，而不是事后再来添加。但是为了简化实现，我们也不希望压缩格式支持是必须的。我们也希望可选地支持更长的指令，用于实验和更大的指令集扩展。虽然我们的编码约定需要一个对核心 RISC-V ISA 更为紧凑的编码，但是这样做有几个好处。

一个支持标准 G 的 ISA 实现，只需要在指令缓存（instruction cache）保存指令的最高 30 位（带来 6.25%的节约）。当重新填充指令缓存时，任何最低两位有一位为零的指令，应当在被保存到缓存之前，被重新编码为 30 位非法指令，以确保出现非法指令异常行为。

可能更为重要的是，通过浓缩我们的基本 ISA 为一个 32 位指令字的子集，我们留给定制扩展更多的空间。特别地，基本 RV32I ISA 在 32 位指令字中，使用了少于 1/8 的编码空间。如同在第 10 章所述，一个不需要支持标准压缩指

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

令扩展的实现，可以将3 个额外的30 位指令空间映射到32 位固定长度格式中，同时确保对大于32 位指令集扩展的支持。更进一步，如果该实现并不需要支持超过32 位长度的指令，它还可以恢复4 个主要的操作码区域。

我们认为这是一个特性，即任何长度的指令，如果它的所有位都是零，那么这是一条非法的指令，因此这将快速地自陷错误跳转到填满零的存储器区域（as this quickly traps erroneous jumps into zeroed memory regions）。类似的，我们也将指令编码全是1 保留为一条非法指令，以捕获另外一种当断开存储器总线或损坏的存储器设备时常见的情形。

基本RISC-V ISA具有一个小端存储器系统，但是非标准变种可以提供大端或者双端存储器系统。指令被保存在存储器中，每个16位包裹以实现的端字节顺序，被保存到一个存储器半字中。包含一条指令的包裹，被保存到递增的半字地址，其中最低寻址的包裹保存着指令规范中最低位的二进制值，也就是说，指令总是按照一系列包裹的小端顺序保存的，而不管存储器的端字节顺序。图 1.2中的代码序列，将把一条32位指令正确地保存到存储器中，而不管存储器的端字节顺序。

```
// 将x2中的32位指令，保存到x3指向的存储器
sh          x2, 0(x3)      // 将指令的低半部分保存到第一个包裹中
srli        x2, x2, 16     // 将高位移动到低位，覆盖x2
sh          x2, 2(x3)      // 将高位保存到第二个包裹中
```

图 1.2：将 32 位指令从寄存器保存到存储器的推荐代码序列。在大端、小端存储器系统中都能正确工作，当使用变长指令集扩展时，可避免出现非对齐访问。

我们为 RISC-V 存储器系统选择小端字节顺序，是因为当前小端系统占据商业应用的统治地位（所有的x86 系统；iOS，Android，Windows for ARM）。一个小问题就是，我们发现小端存储器系统对于硬件设计者来说，更为自然。然而，某些应用领域，例如 IP 网络，在大端数据结构上进行操作，因此我们留给非标准大端或者双端系统一些机会。

我们必须将指令包裹在存储器中保存的顺序固定下来，而与存储器系统的端字节顺序无关，来确保指令的长度编码位总是出现在半字地址的最前面。这就允许取指单元通过一次取指，读取第一个16 位指令包裹的最低几位，就可以确定变长指令的长度。一旦我们确定了小端存储器系统和指令包裹顺序，自然导致我们将指令长度编码放到指令格式的LSB 位置，以避免破坏操作码字段。

1.3 异常、自陷和中断

我们将术语***异常（exception）***认为是在运行时出现了一个与当前RISC-V线程中的一条指令相关的非正常的情况。我们将术语***自陷（trap）***认为是在一个RISC-V线程中出现了一个异常的情况，导致将控制同步传输到自陷处理函数。自陷处理函数通常是在一个更高特权环境中执行的。

我们将术语***中断（interrupt）***认为是在当前RISC-V线程外异步出现了一个事件。如果出

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

现了一个必须处理的中断，将会选择某条指令来接收中断异常，然后顺序地产生一个自陷。

在后续章节中的指令描述了在执行时产生异常的条件。这些异常是否和如何转变为自陷的，依赖于执行环境，虽然预期是绝大多数环境在一个异常被触发时（signaled），采取一个***精确的***自陷（除了浮点异常，在标准浮点扩展中，并不会产生自陷）。

我们的“异常”和“自陷”术语的使用，与 IEEE-754 浮点标准是一致的。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第2章 RV32I 基本整数指令集，2.0 版

本章描述了2.0版的RV32I基本整数指令集。多数注解也应用于RV64I变种。

RV32I 被设计成足以构建一个编译器目标机，并支持现代操作系统环境。这个 ISA 也被设计成在最小实现时减少所需的硬件。RV32I 包括了 47 条单独的指令，虽然某个简单的实现可以使用一条 SYSTEM 硬件指令将 8 条 SCALL/SBREAK/CSRR（**译者注：应当是 ECALL/EBREAK/CSRR***）指令全部用自陷实现，并将 FENCE 指令和 FENCE.I 指令都作为 NOP 指令实现，这将把硬件指令数减少到总共 38 条。RV32I 可以仿真几乎所有其他的 ISA 扩展(除了 A 扩展，它需要额外的硬件以支持原子性)。*

2.1 基本整数子集的程序员模型

图 2.1给出了基本整数子集的用户可见状态。有31个通用寄存器**x1~x31**，它们保存了整数数值。寄存器**x0**是硬件连线的常数0。没有硬件连线的子程序返回地址连接寄存器，但是在一个过程调用中，标准软件调用约定使用寄存器**x1**来保存返回地址。对于RV32，其**x**寄存器是32位宽度的，对于RV64，它们是64位宽度的。本文档使用术语XLEN来指明当前**x**寄存器的宽度（不是32就是64）。

还有一个额外的用户可见寄存器：程序计数器pc保存了当前指令的地址。

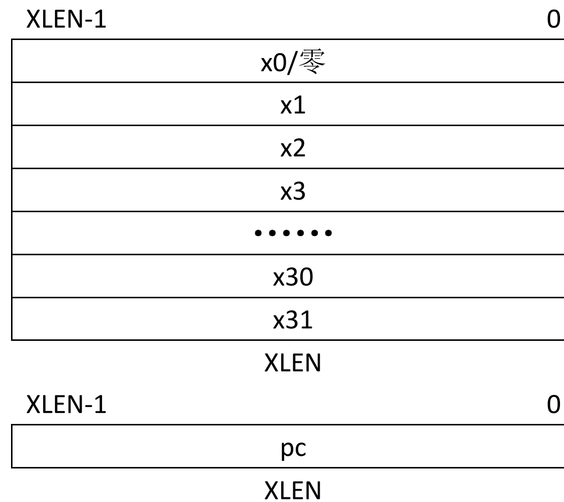


图 2.1：RISC-V 用户级基本整数寄存器状态

---

*可用的体系结构寄存器数量，对代码大小、性能和能耗有巨大的影响。虽然有争论说 16 个寄存器对于一个运行编译代码的整数 ISA 来说足够了，但是在使用 3 地址格式的 16 位指令中编码 16 个寄存器，从而实现一个完整的 ISA 几乎是不可能的（**译者注：16 个寄存器，需要 4 位来区别。3 地址格式，就需要使用 12 位来编码，对于 16 位指令来说，留给操作码的只有 4 位了，几***

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

乎不可能）。虽然一个 2 地址格式是可行的，它将增加指令数目并降低效率。我们想避免出现中间的指令长度（例如Xtensa 的24 位指令）来简化基本硬件实现，并且一旦采用32 位指令长度，支持32 个寄存器就是直截了当的事情。更大的整数寄存器数量也有助于提高高性能代码的性能，可以大量使用循环展开（loop uunrolling）、软件流水线（software pipelining）和cache 分块（cache tiling）。

基于这些原因，我们在基本ISA 上选择了传统大小32 个整数寄存器。动态寄存器使用趋向于被几个频繁访问的寄存器所左右，并且寄存器文件实现可以针对频繁访问的寄存器进行优化，减少访问能耗[26]。可选的压缩16 位指令格式大部分时间仅仅访问8 个寄存器，并且因此可以提供密集的指令编码，同时额外的指令集扩展如果需要的话，可以支持大得多的寄存器空间（平坦的或者层次的）。

对于资源约束的嵌入式应用，我们定义了RV32E 子集，它只有16 个寄存器（第3 章）。

2.2 基本指令格式

在基本ISA中，有四种核心指令格式（R/I/S/U），如图 2.2所示。所有的指令都是固定32位长度的，并且在存储器中必须在4字节边界对齐。当发生一个条件分支或者无条件转移而且目标地址不是对齐到4字节时，将会产生一个指令地址不对齐的异常。如果条件分支没有发生（not taken），那么将不会产生一个取指不对齐异常。

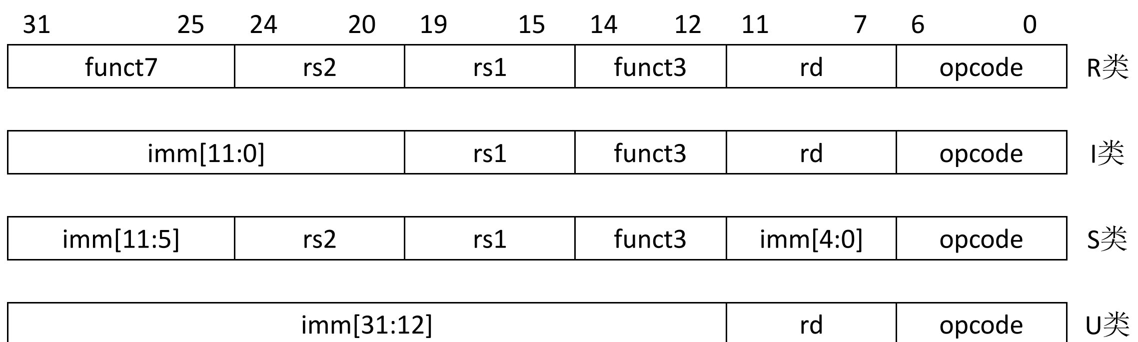


图 2.2：RISC-V 基本指令格式

在所有格式中，RISC-V ISA将源寄存器（***rs1***和***rs2***）和目标寄存器（***rd***）固定在同样的位置，以简化指令译码。在指令中，立即数被打包，朝着最左边可用位的方向，并且是分配好的，以减少硬件复杂度。特别地，所有立即数的符号位总是在指令的第31位，以加速符号扩展电路。

解码寄存器区分符通常处于实现的关键路径上，因此指令格式选择将所有寄存器区分符，在所有格式中，都固定在相同的位置。这是有代价的，不得不把立即数的一些位分散到格式中（这是与 RISC-IV，也就是 SPUR[14]，相同的特性）。

事实上，绝大多数立即数要么很小，要么需要所有的XLEN 位。我们选择

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

*了一种非对称的立即数切分方法（在常规的指令中的低 12 位，加上一条特殊的 load 立即数指令提供高 20 位）来增加常规指令的可用操作码空间。（**译者注：为了加载一个 32 位立即数，需要两步：load 指令提供该立即数的高 20 位[31:12]，常规指令提供该立即数的低 12 位[11:0]，最后拼接成一个 32 位立即数**）。另外，这些立即数都是符号扩展的（**译者注：有符号数**）。我们并没有观察到使用零扩展（**译者注：无符号数**）带来的好处，并且我们想把 ISA 做得尽可能简单。*

2.3 立即数编码变种

基于立即数处理，还有额外两种指令格式变种（SB/UJ），如图 2.3所示。

在图 2.3中，每个立即数字段被所生成的立即数值中的位的位置（imm[x]）标签，而不是在指令的立即数字段中的通常位的位置。图 2.4给出了每一种基本指令格式生成的立即数，并被标签，以显示哪个指令位（inst[y]）生成了立即数值中的哪个位。

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0

funct7	rs2	rs1	funct3	rd	opcode	R类

imm[11:0]	rs1	funct3	rd	opcode	I类

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S类

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	SB类

imm[31::12]	rd	opcode	U类

imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode	UJ类

图 2.3：RISC-V 显示了立即数的基本指令格式

31 30 20 19 12 11 10 5 4 1 0

—inst[31]—	inst[30:25]	inst[24:21]	inst[20]	I立即数

—inst[31]—	inst[30:25]	inst[11:8]	inst[7]	S立即数

—inst[31]—	inst[7]	inst[30:25]	inst[11:8]	0	B立即数

inst[31]	inst[30:20]	inst[19:12]	—0—	U立即数

—inst[31]—	inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J立即数

图 2.4：RISC-V 指令生成的立即数。用指令的位标注了用于构成立即数的字段。符号扩展总是使用 inst[31]。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

S和SB格式唯一的区别在于，在SB格式中，12位立即数字段用于编码2的倍数的分支偏移量。与通常在硬件中将编码在指令中的立即数所有位向左移动1位不同，此处中间位（imm[10:1]）和符号位保持在固定的位置，而S格式中的最低位（inst[7]）编码为SB格式中的高位（imm[11]）。

类似的，U和UJ格式唯一的区别在于，20位立即数被左移12位以生成U立即数，而被左移1位以生成J立即数。在U和UJ格式立即数，其在指令中的位置的选择，以最大化与其它指令的相互覆盖，以及最大化U和UJ格式立即数的相互覆盖。

立即数的符号扩展是最关键的操作之一（特别是在 RV64I 当中），而在 RISC-V 中，所有立即数的符号位总是在指令的 31 位，这允许符号扩展操作可以和指令译码并行。

虽然更为复杂的实现可能对分支和跳转计算有独立的加法器，因此也不会受益于将所有指令中的立即数位置保持固定不变，但是我们想降低简单实现的硬件代价。通过在指令中旋转位，来对 B 和 J 立即数进行编码，而不是使用动态硬件多路选择器（mux）来将立即数乘 2，我们减少了大约 2 倍的指令信号的扇出（fanout）和立即数多路选择器。混乱的立即数编码将给静态或者预先编译带来可忽略的时间开销。对于动态生成指令，则有一些小的额外开销，但是最常见的向前短分支具有直截了当的立即数编码。

2.4 整数计算指令

绝大多数整数计算指令对保存在整数寄存器中的XLEN位值进行操作。整数计算指令要么使用I类格式编码为寄存器-立即数操作，要么使用R类格式编码为寄存器-寄存器操作。对于寄存器-立即数指令和寄存器-寄存器指令，其目标都是寄存器***rd***。没有整数计算指令产生算术异常。

我们并没有包含特殊的指令集支持整数算术操作的溢出检测，因为许多溢出检测可以使用 RISC-V 分支指令以较低的代价实现。无符号数加法的溢出检测，只需要在加法后执行一条额外的分支指令。类似的，有符号数组边界检测，也只需要一条分支指令。有符号数加法溢出检测，需要几条指令，这与加数是一个立即数还是一个变量有关。我们考虑过添加分支指令，用于检测它们的有符号数寄存器操作数的和是否会溢出，但是最终选择了将这些指令从基本 ISA 中去掉。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

整数寄存器-立即数指令

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	
I立即数[11:0]	src	ADD/SLTI[U]	dest	OP-IMM
I立即数[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

ADDI将符号扩展的12位立即数加到寄存器***rs1***上。算术溢出被忽略，而结果就是运算结果的低XLEN位。ADDI ***rd,rs1,0***用于实现MV ***rd,rs1***汇编语言伪指令。

SLTI（set less than immediate）将数值1放到寄存器***rd***中，如果寄存器***rs1***小于符号扩展的立即数（比较时，两者都作为有符号数），否则将0写入***rd***。SLTIU与之相似，但是将两者作为无符号数进行比较（也就是说，立即数被首先符号扩展为XLEN位，然后被作为一个无符号数）。注意，SLTIU ***rd,rs1,1***将设置***rd***为1，如果***rs1***等于0，否则将***rd***设置为0（汇编语言伪指令SEQZ ***rd,rs***）。

ANDI、ORI、XORI是逻辑操作，在寄存器***rs1***和符号扩展的12位立即数上执行按位AND、OR、XOR操作，并把结果写入***rd***。注意，XORI ***rd,rs1,-1***在***rs1***上执行一个按位取反操作（汇编语言伪指令NOT ***rd,rs***）。

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	移位次数[4:0]	src	SLLI	dest	OP-IMM
0000000	移位次数[4:0]	src	SRLI	dest	OP-IMM
0100000	移位次数[4:0]	src	SRAI	dest	OP-IMM

被移位常数次，被编码为I类格式的特例。被移位的操作数放在***rs1***中，移位的次数被编码到I立即数字段的低5位。右移类型被编码到I立即数的一位高位。SLLI是逻辑左移（0被移入低位）；SRLI是逻辑右移（0被移入高位）；SRAI是算术右移（原来的符号位被复制到空出的高位中）。

31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U立即数[31:12]	dest	LUI
U立即数[31:12]	dest	AUIPC

LUI（load upper immediate）用于构建32位常数，并使用U类格式。LUI将U立即数放到目

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

标寄存器rd的高20位，将rd的低12位填0。

AUIPC（add upper immediate to pc）用于构建**pc**相对地址，并使用U类格式。AUIPC从20位U立即数构建一个32位偏移量，将其低12位填0，然后将这个偏移量加到**pc**上，最后将结果写入寄存器***rd***。

对于控制流转移和数据访问，AUIPC 指令支持双指令序列，以从当前 pc 访问任意偏移地址。通过一条AUIPC 指令和一条12 位立即数JALR 指令的组合，可以将控制转移到任意32 位pc 相对地址；而一条AUIPC 指令加上一条12 位立即数偏移的常规load 或者store 指令，可以访问任意32 位pc 相对数据的地址。

*当前pc 的值，可以通过将U 立即数设置为0 来读取。虽然一条JAL+4 指令也可以获得pc 值，但是它在简单的实现中可能会导致流水线停顿，或者在更复杂的微体系结构中，导致BTB 结构被污染。（**译者注：JAL+4 实际上要执行一个分支操作。这个分支会对流水线造成负面影响。而AUIPC 则不会，它只是运算指令**）*

整数寄存器-寄存器操作

RV32I定义了几种算术R类操作。所有操作都是读取***rs1***和***rs2***寄存器作为源操作数，并把结果写入到寄存器***rd***中。***funct7***和***funct3***字段选择了操作的类型。

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

ADD和SUB分别执行加法和减法。溢出被忽略，并且结果的低XLEN位被写入目标寄存器***rd***。SLT和SLTU分别执行符号数和无符号数的比较，如果***rs1<rs2***，则将1写入***rd***，否则写入0。注意，SLTU ***rd,x0,rs2***，如果***rs2***不等于0（**译者注：在RISC-V中，x0寄存器永远是0**），则把1写入***rd***，否则将0写入***rd***（汇编语言伪指令SNEZ ***rd,rs***）。AND、OR、XOR执行按位逻辑操作。SLL、SRL、SRA分别执行逻辑左移、逻辑右移、算术右移，被移位的操作数是寄存器***rs1***，移位次数是寄存器***rs2***的低5位。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

NOP 指令

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	
0	0	ADDI	0	OP-IMM

NOP指令并不改变任何用户可见的状态，除了使得**pc**向前推进。NOP被编码为ADDI ***x0,x0,0***。

NOP 可用于将代码段对齐到对微体系结构有重要作用的地址边界上，或者给内联（inline）代码修改保留空间。虽然有很多种编码可以成为 NOP，我们定义了一个正规的 NOP 编码，允许微体系结构对此进行优化，同时也使得反汇编输出更具可读性。

2.5 控制转移指令

RV32I提供了两类控制转移指令：无条件跳转和条件分支。RV32I中的控制转移指令，并没有体系结构可见的分支延迟槽。

无条件跳转

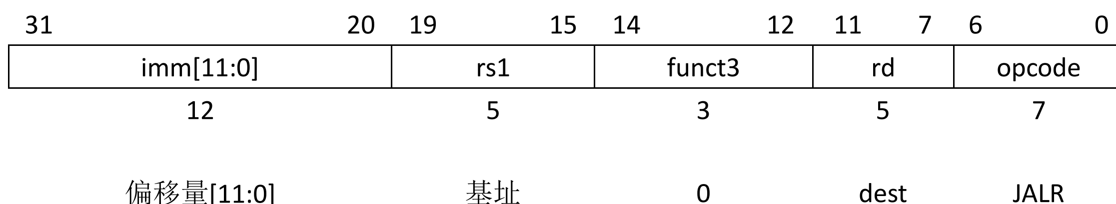
跳转并连接（JAL）指令使用了UJ类格式，此处J立即数编码了一个2的倍数的有符号偏移量。这个偏移量被符号扩展，加到**pc**上，形成跳转目标地址，跳转范围因此达到±1MB。JAL将跳转指令后面指令的地址（**pc**+4）保存到寄存器***rd***中。标准软件调用约定使用**x1**来作为返回地址寄存器。

普通的无条件跳转指令（汇编语言伪指令J）被编码为***rd=x0***的JAL指令。（**译者注：x0是只读寄存器，无法写入**）

31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
	偏移量[20:1]			dest	JAL

间接跳转指令JALR（jump and link register）使用I类编码。通过将12位有符号I类立即数加上***rs1***，然后将结果的最低位设置为0，作为目标地址。跳转指令后面指令的地址（**pc**+4）保存到寄存器***rd***中。如果不需要结果，则可以把**x0**作为目标寄存器。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.



31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
偏移量[11:0]	基址	0	dest	JALR

JAL指令和JALR指令会产生一个非对齐指令取指异常，如果目标地址没有对齐到4字节边界。

所有的无条件跳转指令都是用pc相对寻址，这有助于支持位置无关代码。JALR指令被定义为可以使用双指令序列来跳转到32位绝对地址空间的任何地方。首先一条LUI指令将目标地址的高20位加载到rs1中，然后JALR指令可以加上低12位。类似的，AUIPC指令，然后JALR指令就可以跳转到32位绝对地址空间的任何地方。

注意到JALR指令并没有把12位立即数作为2字节的倍数看待，这与条件分支指令不同。这避免了在硬件上多出一种立即数格式。事实上，绝大多数JALR指令的使用要么是一个立即数0，要么与LUI或者AUIPC成对使用，因此在范围上的稍微减小，影响并不显著。

JALR指令忽略了计算出来的目标地址的最低位。这不但稍微简化了硬件，同时也允许函数指针的最低位可以用于存放额外的信息。虽然此种情形下，可能会有潜在轻微的误差损失，实际上跳转到一个不正确的指令地址，通常将很快会引起一个异常。

在支持16位对齐指令扩展的机器上，例如压缩指令集扩展C，不可能产生指令取指非对齐异常。

返回地址预测栈，是高性能指令取指单元的一种常见特性。我们注意到rd和rs1可用于指导一个实现的指令取指预测逻辑，指示JALR指令是否应当push（rd=x1）、pop（rd=x0,rs1=x1）还是不操作（其余情况）一个返回地址栈。类似的，一条JAL指令只有在rd=x1的时候，才能将返回地址push到返回地址栈中。

当rs1=x0时，JALR可完成一个单一指令的过程调用，实现从任意地址空间对最低的2KB或者最高的2KB地址区域进行调用，这可用于实现对小的运行时库的快速调用。

条件分支

所有分支指令使用SB类指令格式。12位B立即数编码了以2字节倍数的有符号偏移量，并被加到当前**pc**上，生成目标地址。条件分支范围是±4KB。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
偏移量[12,10:5]		src2	src1	BEQ/BNE	偏移量[11,4:1]		BRANCH
偏移量[12,10:5]		src2	src1	BLT[U]	偏移量[11,4:1]		BRANCH
偏移量[12,10:5]		src2	src1	BGE[U]	偏移量[11,4:1]		BRANCH

分支指令比较两个寄存器。BEQ和BNE将跳转，如果***rs1***和***rs2***相等或者不相等。BLT和BLTU将跳转，如果***rs1***小于***rs2***，分别使用有符号数和无符号数进行比较。BGE和BGEU将跳转，如果***rs1***大于等于***rs2***，分别使用有符号数和无符号数进行比较。注意，BGT、BGTU、BLE和BLEU可以通过将BLT、BLTU、BGE、BGEU的操作数对调来实现。

软件应当优化，使得顺序代码路径是最常见执行路径，而频率较少的跳转执行代码则放到直线路径之外。软件同时也应当假设向回（向后）跳转总是被预测跳转的，而向前（向下）跳转总是被预测不跳转的，至少第一次碰到分支指令的时候，是这样的。动态分支预测器将很快学会任何可以预测的分支行为。

与其它某些体系结构不同，无条件跳转应当总是使用RISC-V的跳转（***rd=x0***的JAL）指令，而不是一条条件永远为真的条件分支指令。RISC-V跳转总是**pc**相对寻址的，并且比分支指令支持大得多的偏移量范围，而且还不会对条件分支预测表造成压力。（**译者注：现代处理器都有条件分支预测器，对每一条碰到的条件分支指令，都会记录其结果，以便后面对其进行预测**）。

---

*条件分支指令被设计为在两个寄存器之间进行算术比较操作（如同PA-RISC 和 Xtensa ISA 做的那样），而不是使用条件码（x86、ARM、SPARC、PowerPC），也不是只将寄存器与零进行比较（Alpha、MIPS），也不是比较两个寄存器相等（MIPS）。这样的设计灵感来自我们观察到，一条组合了比较和分支的指令，可以很好地适应常规的流水线，避免了使用额外的条件码状态或者使用临时寄存器，减少了静态代码大小、降低了动态指令取指通信量。另外一点是，将寄存器与零比较需要不可忽视的电路延迟（non-trivial circuit delay）（特别是在先进工艺中使用静态逻辑的时候），因此几乎和算术比较是一样昂贵的操作。将比较和分支融合成一条指令的另外一个好处是，分支指令会在指令流的前端被检测到（**译者注：分支预测功能，需要在取指段即判断是否是条件分支指令，而不会像通常指令一样等到译码段**），以便可以提前进行预测。采用条件码的设计，也许在当基于相同条件码而执行多个分支时，有一些好处，但是我们相信这种情形相对来说比较少见。*

*我们考虑过但是并没有在指令编码中加入静态分支提示（static branch hints）（**译者注：某些处理器允许在指令中加入提示位，告知此条分支指令是否会执行分支**）。静态分支提示可以降低对动态分支预测器的压力，但是需要更多的指令编码空间以及软件 profiling 才能取得较好的效果，而且一旦程序的运行与 profiling 运行不同时，性能会大幅度下降。*

*我们考虑过但并没有加入条件传输或者条件执行指令（**译者注：原文为 predicted instruction，为方便理解，翻译为条件执行指令。类似 ARM 中，每***

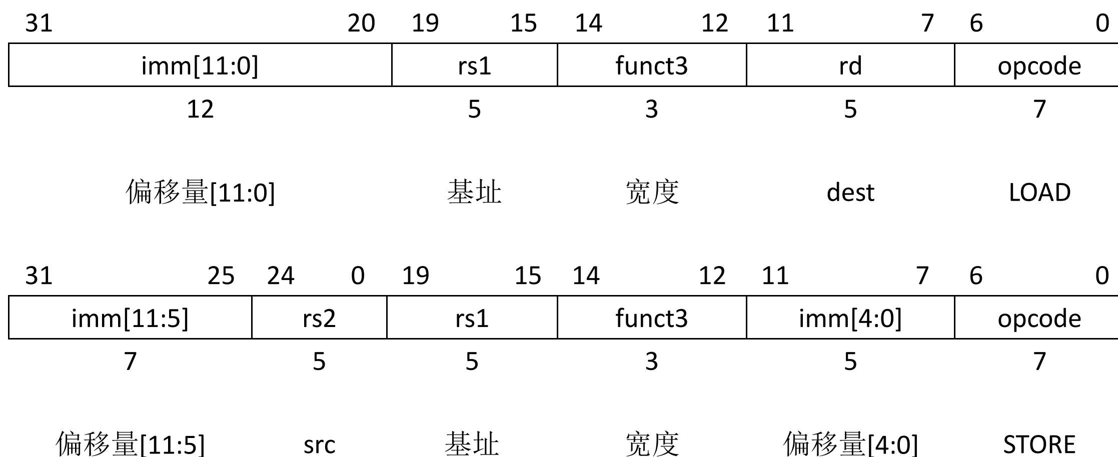
Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

__条指令都是条件执行的，在每条指令的头部，都有 4 位条件码，只有符合条件码，这条指令才能被正常执行，否则就是空操作__)，它们可以有效地替代不可预测的向前短分支指令。条件传输指令在两者中较为简单，但是当条件码可能产生异常时（存储器访问和浮点运算），变得难以使用。条件执行指令在系统中加入了额外的标志状态，需要额外的指令来设置和清除这些标志，需要对每条指令有额外的编码开销。条件传输指令和条件执行指令对于乱序执行微体系结构来说，都增加了复杂性，当出现预测错误时，增加了隐式的第三个源操作数，因为需要把目标体系结构寄存器的原始值，复制到重命名物理寄存器中（__译者注：在乱序执行结构中，需要解决寄存器相关，因此引入了重命名物理寄存器，程序员可见的称为体系结构寄存器。当出现分支预测错误时，需要把寄存器恢复到分支预测前的状态__）。另外，使用静态编译时的决定来预测，而不是使用分支来预测，在使用没有包含在编译时的输入训练集的时候，将会得到较低的性能，特别是实际上不可预测的分支非常稀少，而且随着分支预测技术的进步，不可预测的分支变得更为稀少。

我们注意到，现存各种微体系结构技术可以将不可预测的向前短分支指令，在内部动态地转换为条件执行的代码，以避免当出现预测错误时清空流水线[9][13][12]，这在一些商用处理器中得到了实现[22]。最简单的技术就是通过仅仅清空受到分支指令影响的其他指令，而不是清空整个流水线，或者通过使用更宽的指令取指同时从分支的两边取指，或者暂停取指，这样可以减少预测错误时带来的恢复开销。乱序执行内核采用更为复杂的技术，在受到分支影响的其他指令上，附加上内部的预测信息，这些预测信息是分支指令写入的，这样就允许分支指令以及其后的指令可以推测地执行，并可以和其他代码一样乱序执行[22]。

2.6 Load 和 store 指令

RV32I是一个load-store体系结构，也就是说，只有load和store指令可以访问存储器，而算术指令只在CPU寄存器上进行操作运算。RV32I提供了一个32位用户地址空间，它是字节寻址并且是小端的。执行环境将定义这个地址空间的哪些部分是可以合法访问的（**译者注：这涉及到存储保护等**）。



31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
偏移量[11:0]	基址	宽度	dest	LOAD

31 25	24 0	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
偏移量[11:5]	src	基址	宽度	偏移量[4:0]	STORE

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

Load和store指令在寄存器和存储器之间传输数值。Load指令编码为I类格式，而store指令编码为S类格式。有效字节地址是通过将寄存器***rs1***与符号扩展的12位偏移量相加而获得的。Load指令将存储器中的一个值复制到寄存器***rd***中。Store指令将寄存器***rs2***中的值复制到存储器中。

LW指令将一个32位数值从存储器复制到***rd***中。LH指令从存储器中读取一个16位数值，然后将其进行符号扩展到32位，再保存到***rd***中。LHU指令存储器中读取一个16位数值，然后将其进行零扩展到32位，再保存到***rd***中。对于8位数值，LB和LBU指令的定义与前面类似。SW、SH、SB指令分别将从***rs2***低位开始的32位、16位、8位数值保存到存储器中。

为了获得最高的性能，所有load和store指令的有效地址，应该与该指令对应的数据类型相对齐（也就是说，32位访问应该在4字节边界对齐，16位访问应该在2字节边界对齐）。基本ISA支持非对齐的访问，但是根据实现的不同，这可能会运行得非常慢。更进一步的，对齐的load和store访问执行时，可以确保是原子性的，而非对齐的load和store可能不能原子性的完成，因此需要额外的同步来确保原子性（**译者注：具体实现非对齐访问时，可能一次访问会被分解为两次存储器访问，这就不是不可分割的原子性操作，有潜在的危险**）。

非对齐访问在移植遗留代码时有时是需要的，而且在那些使用任何形式packed-SIMD扩展的应用程序上，这是取得高性能的基本要求。我们通过常规的load和store指令支持非对齐访问的根据在于，这样做，可以简化需要额外非对齐硬件的支持。一种选项是在基本ISA中禁止非对齐访问，然后通过某些单独的ISA来提供对非对齐访问的支持，要么是通过某些特殊指令帮助软件处理非对齐访问，要么是一个新的非对齐硬件寻址模式。特殊指令是难以使用的，导致ISA复杂化，并且通常会加入新的处理器状态（例如SPARC VIS 对齐地址偏移寄存器）或者访问当前处理器状态复杂化（例如MIPS LWL/LWR部分寄存器写）。另外，对于面向循环的packed-SIMD代码，当操作数不对齐时导致的性能开销，促使软件根据操作数的对齐方式提供多种形式的循环样式，这将导致软件代码产生复杂化，并且导致循环启动时的开销。新的非对齐硬件寻址模式，在指令编码中占据大量空间或者需要非常简化的寻址模式（例如只有寄存器间接寻址）。

我们并没有强制非对齐访问的原子性，因此简化了实现，可以通过使用一个机器自陷和软件处理函数来处理非对齐访问。如果硬件支持非对齐访问，软件则可以通过使用常规load和store指令来简化。硬件则可以依据运行时地址是否是对齐的，来自动优化访问。

2.7 存储器模型

基本RISC-V ISA在一个单一的用户地址空间内支持多个同时线程的执行。每个RISC-V线程拥有它自己的寄存器和程序计数器，并执行一段不相关的顺序指令流。执行环境将定义RISC-V线程是如何创建和管理的。RISC-V线程可以通过调用执行环境或者直接通过共享存储器系统来在相互之间进行通信和同步，执行环境将在规范的另外文档中描述。RISC-V线程也可以与I/O设备交互，并可通过对指派给I/O的地址空间部分进行load和store，间接地在I/O设备间通信。

在基本RISC-V ISA中，每个RISC-V线程看到它自己的存储器操作，如同它们就是按照程序中的顺序执行一样。RISC-V在线程间有一个放松的存储器模型（relaxed memory model），在

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

不同的RISC-V线程之间的存储器操作，需要一条明确的FENCE指令来确保任何特定地顺序。第6章介绍了可选的原子性存储器指令扩展“A”，它可以在共享存储器空间中提供额外的同步操作。

31 28	27	26	25	24	23	22	21	20	19 15	14 12	11 7	6 0
0	PI	PO	PR	PW	SI	SO	SR	SW	rs1	funct3	rd	opcode
4	1	1	1	1	1	1	1	1	5	3	5	7
0	前续（predecessor）				后续（successor）				0	FENCE	0	MISC-MEM

FENCE指令用于顺序化其他RISC-V线程、外部设备或者协处理器看到的设备I/O和存储器访问。任何设备输入（I）、设备输出（O）、存储器读（R）、存储器写（W）的组合，相对于其他一样的组合，可能需要按序的。通俗的说，在所有***前续集合（predecessor set）***执行到FENCE指令前的任何操作之前，处在FENCE指令后的***后续集合（successor set）***中的任何操作，都不能被任何其他RISC-V线程或者外部设备看到（**译者注：FENCE就像一个栅栏，FENCE之前所有的存储器操作、I/O操作必须完成后，在FENCE之后的指令才能看到结果**）。执行环境将定义什么I/O操作是可能的，特别地，哪些load或者store指令被处理并且作为设备输入或设备输出操作顺序化，而不是被作为存储器读和写来处理。例如，内存映射I/O（memory-mapped I/O）设备通常被非缓存（uncached）的load和store指令来访问，并使用（FENCE指令中的）I和O位，而不是R和W位。

FENCE指令中未使用的字段***imm[11:8]***、***rs1***和***rd***被保留给未来扩展中更细粒度的栅栏。为了保持前向兼容性，基本实现应当忽略这些字段，而标准软件应当对这些字段写0。

我们选择一个放松的存储器模型，以允许对一个简单的机器实现可以得到较高的性能。放松的存储器模型也非常可能兼容未来协处理器或者加速器扩展。我们将I/O顺序化与存储器R/W顺序化区分开来，以避免在设备驱动程序线程中不必要的串行化，同时也支持控制附加的协处理器或者I/O设备的可选的非存储器通路。简单的实现可以进一步忽略前续（predecessor）和后续（successor）字段，而总是对所有操作执行一个保守的栅栏动作。

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	FENCE.I	0	MISC-MEM

FENCE.I指令用于同步指令和数据流。RISC-V并不能确保在同一个RISC-V线程中，取指看得到前面对指令存储器的store，直到执行一条FENCE.I指令。一条FENCE.I指令只是保证在一个RISC-V线程中，该指令之后的取指操作，可以看得到这条指令之前的任何数据store。在多处理器系统中，FENCE.I指令并***不能***确保其他RISC-V线程的取指看得到本地线程的store。为了使得一条对指令存储器的store对所有RISC-V线程可见，写数据的线程必须在要求所有远程RISC-V线程执行FENCE.I指令之前，执行一条数据FENCE指令。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

FENCE.I指令中未使用的字段***imm[11:0]***、***rs1***和***rd***被保留给未来扩展中更细粒度的栅栏。为了保持前向兼容性，基本实现应当忽略这些字段，而标准软件应当对这些字段写0。

---

*FENCE.I 指令被设计为支持各种实现方式。一种简单的实现就是当执行一条 FENCE.I 指令时，清空本地指令 Cache 和指令流水线。更复杂一些的实现，可以在指令（数据）Cache 上监听每一次数据（指令）Cache 失效，或者使用一个包含的统一私有 L2 Cache（inclusive unified private L2 Cache），当 L2 Cache 的 Cache line 被一条本地 store 指令写时，作废掉主指令 Cache（**译者注：就是 L1 指令 Cache**）中对应的 Cache line（**译者注：这是为了实现运行时修改指令的目的。运行时修改指令时，首先使用 store 指令写存储器，然后数据会被写入 L1 数据 Cache，然后写入 L2 统一 Cache。在此，检测到是写入到指令段，所以 L2 统一 Cache 将反向作废掉 L1 指令 Cache 中的对应行，以保持一致性。L1 指令 Cache 从程序员角度，是不能写的**）。如果指令和数据可以以这种方式保持一致性，那么在执行 FENCE.I 指令时，仅仅需要清空流水线。*

我们考虑过但没有加入一条“保存指令字”指令（如同 MAJC 中一样[25]）。JIT 编译器可能在一条 FENCE.I 指令之前，生成一个较大的指令 trace，并通过写翻译后的指令到那些已知不会缓存到指令 Cache 的存储器区域中，来分散任何指令 cache 监听/作废的开销。

2.8 控制和状态寄存器指令

系统指令用于访问那些可能需要特权访问的系统功能，以I类指令格式编码。这可以分为两类：一类是原子性读-修改-写控制和状态寄存器（CSR）的指令，另一类是其他特权指令。CSR指令在本节描述，另外两条其他的用户级SYSTEM指令将在后面一节描述。

系统指令被定义为，允许在简单的实现中，总是自陷到一个单一的软件自陷处理函数（software trap handler）。更高级的实现，可以在硬件上执行一条或者多条系统指令。

CSR 指令

我们在此定义所有的CSR指令，虽然在用户级基本ISA中，只能访问少数几个只读计数器。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	zimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	zimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	zimm[4:0]	CSRRCI	dest	SYSTEM

CSRRW（Atomic Read/Write CSR）指令原子性的交换 CSR 和整数寄存器中的值。CSRRW 指令读取在 CSR 中的旧值，将其零扩展到 XLEN 位，然后写入整数寄存器 ***rd*** 中。***rs1*** 寄存器中的值将被写入 CSR 中。如果 ***rd*＝*x0***，那么这条指令将不会读该 CSR，且不会导致任何因为 CSR 读而出现的副作用。

CSRRS（Atomic Read and Set Bits in CSR）指令读取 CSR 的值，将其零扩展到 XLEN 位，然后写入整数寄存器 ***rd*** 中。整数寄存器 ***rs1*** 中的初始值被当做按位掩码指明了哪些 CSR 中的位被置为 1。***rs1*** 中的任何为 1 的位，将导致 CSR 中对应位被置为 1，如果 CSR 中该位是可以写的话。CSR 中的其他位不受影响（虽然当 CSR 被写入时可能有些副作用）。

CSRRC（Atomic Read and Clear Bits in CSR）指令读取 CSR 的值，将其零扩展到 XLEN 位，然后写入整数寄存器 ***rd*** 中。整数寄存器 ***rs1*** 中的初始值被当做按位掩码指明了哪些 CSR 中的位被置为 0。***rs1*** 中的任何为 1 的位，将导致 CSR 中对应位被置为 0，如果 CSR 中该位是可以写的话。CSR 中的其他位不受影响。

对于 CSRRS 指令和 CSRRC 指令，如果 ***rs1*＝*x0***，那么指令将根本不会去写 CSR，因此应该不会产生任何由于写 CSR 产生的副作用（**译者注：某些特殊 CSR 检测是否有人尝试写入，一旦有写入，则执行某些动作。这和写入什么值没什么关系**），例如试图访问一个只读 CSR 时产生一个非法指令异常。注意如果 ***rs1*** 寄存器包含的值是 0，而不是 ***rs1*＝*x0***，那么将会把一个不修改的值写回 CSR（**译者注：这时会有一个写 CSR 的操作，但是写入的值就是旧值，因此可能会产生副作用**）。

CSRRWI 指令、CSRRSI 指令、CSRRCI 指令分别于 CSRRW 指令、CSRRS 指令、CSRRC 指令相似，除了它们是使用一个处于 ***rs1*** 字段的、零扩展到 XLEN 位的 5 位立即数（***zimm[4:0]***）而不是使用 ***rs1*** 整数寄存器的值。对于 CSRRSI 指令和 CSRRCI 指令，如果 ***zimm[4:0]***字段是零，那么这些指令将不会写 CSR，因此应该不会产生任何由于写 CSR 产生的副作用。对于 CSRRWI 指令，如果 ***rd=x0***，则这条指令将不会读 CSR，且不会导致任何因为 CSR 读而出现的副作用。

某些 CSR，例如指令退休计数器，instret，可能由于指令的执行副作用而被修改。在这种情形下，如果一条 CSR 访问指令读取了一个 CSR，它读取到的是该指令执行之前的值。如果一条 CSR 指令写了一个 CSR，这个写的值更新是在该指令执行完之后才发生的。特别地，一条指令写入 instret 一个值，将是该指令后面一条指令读取到的值（也就是说，由于第一条指令退休导致的 instret 的增长，是在写入新值之前发生的）。

用于读取 CSR 的汇编语言伪指令 CSRR ***rd, csr*** 被编码为 CSRRS ***rd, csr, x0***。用于写 CSR 的汇编语言伪指令 CSRW ***csr, rs1*** 被编码为 CSRRW ***x0, csr, rs1***，而伪指令 CSRWI ***csr, zimm*** 被编码为 CSRRWI ***x0, csr, zimm***。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

还有汇编语言伪指令被定义在不需要 CSR 旧值时，用来设置和清除 CSR 中的位：CSRS/CSRC ***csr, rs1***；CSRSI/CSRCI ***csr, zimm***。

定时器和计数器

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
RDCYCLE[H]	0	CSRRS	0	SYSTEM
RDTIME[H]	0	CSRRS	0	SYSTEM
RDINSTRET[H]	0	CSRRS	0	SYSTEM

RV32I提供了多个用户级只读的64位计数器，它们被映射到一个12位的CSR地址空间中，它们可以使用CSRRS指令以32位片段的形式进行访问。

RDCYCLE伪指令读取cycle CSR的低XLEN位，这个计数值是从硬件线程从过去的任意时刻开始执行以来的时钟周期计数值。RDCYCLEH指令是一条RV32I仅有的指令，它读取同样的计数值的63-32位。底层的64位计数器在实际使用中应当永远不会溢出。这个周期计数器推进的速率，与实现和操作系统有关。执行环境应当提供一种手段来判定当前的速率（每秒钟多少个时钟周期），周期计数器就是按这个时钟周期速率递增的。

RDTIME伪指令读取time CSR的低XLEN位，这个计数值是从过去的任意时刻开始以来的墙钟实时时间计数值。RDTIMEH指令是一条RV32I仅有的指令，它读取同样的实时时钟计数值的63-32位。底层的64位计数器在实际使用中应当永远不会溢出。执行环境应当提供一种手段来判定实时时钟的周期（每tick多少秒），这个周期应当是一个常量。在一个单用户应用程序中，所有硬件线程的实时时钟必须是同步的，而且误差不能超过实时时钟的一个tick。环境应当提供一种手段来判定时钟的精度。

RDINSTRET伪指令读取instret CSR的低XLEN位，这个计数值是从硬件线程从过去的任意时刻开始执行以来的本硬件线程退休（retire）指令的计数值。RDINSTRETH指令是一条RV32I仅有的指令，它读取同样的指令计数值的63-32位。底层的64位计数器在实际使用中应当永远不会溢出。

下面的代码序列可以将一个有效的64位周期计数器值写入到**x3:x2**中，即使这个计数器在读取它的高低两部分之间产生溢出。

```
again:
        rdcycleh    x3
        rdcycle     x2
        rdcycleh    x4
        bne         x3, x4, again
```

图 2.5：在 RV32 中读取 64 位周期计数器的示例代码

我们强制这些基本计数器在所有实现中都必须提供，因为它们对于基本的性能分析、自适应和动态优化来说是必须的，同时允许应用程序在实时流中工

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

作。可能应当提供额外的计数器，以帮助诊断性能问题，而且这些计数器应当是以较小的代价从用户级应用程序代码中进行访问。

我们要求这些计数器是 64 位宽度的，即使是在 RV32I 上也是如此。否则的话，软件就非常难以判定这些值是否已经溢出。对于一个低端的实现，每个计数器的高 32 位可以实现为使用软件计数器来递增，这个软件计数器的递增，是由一个自陷处理函数来实现的，每当发生低 32 位溢出时，就触发一次自陷。上面给出的示例代码显示了如何使用各条 32 位指令，安全地读取完整的 64 位宽度的数值。

在一些应用程序中，同时读取多个计数器的值是很重要的。当运行在一个多任务下时，一个用户线程在读取这些计数器值时，可能遭受到上下文切换。用户线程的一个解决方案就是，在读取其他计数器之前和之后，分别读取实时计数器，用它来判断在这些操作序列过程中是否发生了上下文切换，如果发生了上下文切换，那么可以重新执行这些读操作。我们考虑过添加一个输出锁存器，允许用户线程对各个计数器值同时进行一个原子性的快照，但是这会增大用户上下文的大小，特别是实现了一大堆计数器的时候，更是如此。

环境调用和断点

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

ECALL指令用于向支持的运行环境发出一个请求，这个运行环境通常是一个操作系统。系统的ABI将定义环境请求的参数是如何传递的，但通常这些参数应当是保存在整数寄存器中确定的位置。

EBREAK指令被调试器所使用，用来将控制权传送回给调试环境。

ECALL 指令和 EBREAK 指令以前被命名为 SCALL 指令和 SBREAK 指令。这些指令具有相同的功能和编码，但是被重新命名了，以反映它们可以比调用管理员级操作系统或者调试器更通用。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第3章 RV32E 基本整数指令集，版本 1.9

本章描述RV32E基本整数指令集，它是RV32I为嵌入式系统而设计的简化版本。主要的变动在于将整数寄存器的数目减少到16个，去掉了RV32I里面强制必须的计数器。本章将仅仅介绍RV32E与RV32I不同的地方，因此必须在第2章之后阅读。

---

*RV32E 被设计成为嵌入式微控制器提供一个更小的基本内核。虽然我们在本文档的 2.0 版本的时候提到过这个可能性，但我们开始的时候是反对定义这个子集的。然而，考虑到最小的可能的 32 位微控制器的需求，并对占领这个空间感兴趣，我们现在将 RV32E 作为除了 RV32I、RV64I、RV128I 之外的第四个标准基本 ISA 进行了定义。这个 E 变种仅对 32 位地址空间宽度是标准化的（**译者注：意思是不会有 RV64E、RV128E**）。*

3.1 RV32E 程序员模型

RV32E 将整数寄存器的数目减少到 16 个通用寄存器，（**x0-x15**），此处 **x0** 是专门的零寄存器。

我们发现在小的 RV32I 核心设计中，上部的 16 个寄存器消耗了大约四分之一除了存储器之外的总面积，因此去掉它们将节约 25%的核心面积，并且降低了相应的核心功耗。

这个改动需要一个不同的调用约定和 ABI。特别地，RV32E 仅使用一个软浮点调用约定。拥有硬件浮点的系统必须使用 I 基本内核。

3.2 RV32E 指令集

RV32E 使用与 RV32I 相同的指令集编码，除了在一条指令中使用 **x16-x31** 寄存器区分符将导致产生一个非法指令异常。

未来任何标准扩展都不会使用减少的寄存器区分符释放的位空间，因此这些位空间可被非标准扩展使用。

更进一步的简化就是计数器指令（rdcycle[h]、rdtime[h]、rdinstret[h]）不再是强制必须的了。

强制必须的计数器需要额外的寄存器和逻辑，可以被更为应用专用的功能所替代。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

3.3 RV32E 扩展

RV32E 可以使用 M 和 C 用户级标准扩展进行扩展。

我们并不想在 RV32E 子集中支持硬件浮点。减少寄存器数量节约出来的空间，在硬件浮点单元面前不值一提，并且我们希望减少 ABI 的增生（proliferation）。

RV32E 系统的特权体系结构可以包含用户模式和机器模式，以及 Mbare、Mbb 和 Mbbid 存储器管理方案，它们在卷 II 中描述。

我们并不想在 RV32E 子集中支持完整的 Unix 类操作系统。减少寄存器数量节约出来的空间，在操作系统能力特性内核面前不值一提，并且我们希望避免操作系统碎片化。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第4章 RV64I 基本整数指令集，版本 2.0

本章介绍RV64I基本整数指令集，它构建于第2章所介绍的RV32I变种之上。本章将仅仅介绍与RV32I不同的地方，因此必须与前面的章节结合起来阅读。

4.1 寄存器状态

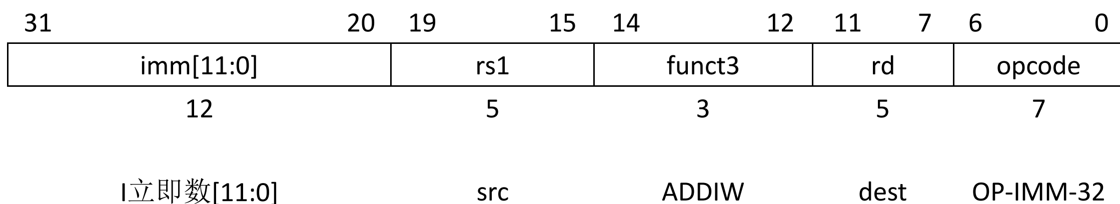
RV64I加宽了整数寄存器并支持64位用户地址空间（图 2.1中，XLEN=64）。

4.2 整数计算指令

在RV64I中，提供了额外的指令变种来操作32位数值，这通过在操作码后面加上“W”后缀来识别。这些“*W”指令忽略了它们输入的高32位，并且总是生成32位有符号数值，也就是说，XLEN-1位到31位都是相同的。这在RV32I中将会产生一个非法指令异常。

编译器和调用约定总是认为所有的32 位数值总是以符号扩展的格式保存在64 位寄存器中的，即使是32 位无符号整数，它都会把它的31 位扩展到63 位~32 位。因此，在无符号32 位整数和有符号32 位整数之间进行相互转换，并没有实质的操作，就像从有符号32 位整数转成有符号64 位整数一样。现有的64 位宽度SLTU 指令和无符号分支比较指令，在这种不改变的约定下，仍然能够针对无符号32 位整数正确工作。相似的，在32 位符号扩展整数上，执行现有的64 位宽度逻辑操作，仍然保持了符号扩展的正确性。需要一些新的指令（ADD[I]W/SUBW/SxxW），来执行针对 32 位数值的额外的加法、移位，以确保适当的性能。

整数寄存器-立即数指令



31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I立即数[11:0]	src	ADDIW	dest	OP-IMM-32

ADDIW是一条RV64I仅有的指令，它将符号扩展的12位立即数和寄存器***rs1***相加，生成正确符号扩展的32位结果，保存到***rd***中。溢出被忽略，并且结果的低32位符号扩展成64位。注意，ADDIW ***rd,rs1,0***将寄存器***rs1***的低32位符号扩展，结果写入寄存器***rd***中（汇编语言伪指令SEXT.W）。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

31　　26	30　　25	24　　20	19　15	14　12	11　7	6　　0
imm[11:6]	imm[5]	Imm[4:0]	rs1	funct3	rd	opcode
6	1	5	5	3	5	7
000000	移位次数[5]	移位次数[4:0]	src	SLLI	dest	OP-IMM
000000	移位次数[5]	移位次数[4:0]	src	SRLI	dest	OP-IMM
010000	移位次数[5]	移位次数[4:0]	src	SRAI	dest	OP-IMM
000000	0	移位次数[4:0]	src	SLLIW	dest	OP-IMM-32
000000	0	移位次数[4:0]	src	SRLIW	dest	OP-IMM-32
010000	0	移位次数[4:0]	src	SRAIW	dest	OP-IMM-32

移位常数次被编码为I类指令格式的特例，使用与RV32I相同的指令操作码。被移位的操作数存放在寄存器***rs1***中，RV64I中移位次数被编码到I立即数字段的低6位。右移类型被编码到30位。SLLI是逻辑左移操作（0被移入低位）；SRLI是逻辑右移操作（0被移入高位）；SRAI是算术右移操作（原来的符号位被复制到空出的高位中）。对于RV32I，如果***imm[5]≠0***，则SLLI、SRLI、SRAI指令将会产生一个异常。

SLLIW、SRLIW、SRAIW是RV64I仅有的指令，与其定义相类似，但是它们对32位数值进行操作，并产生有符号的32位结果。如果***imm[5]≠0***，SLLIW、SRLIW、SRAIW指令将会产生一个异常。

31　　12	11　7	6　　0
imm[31:12]	rd	opcode
20	5	7
U立即数[31:12]	dest	LUI
U立即数[31:12]	dest	AUIPC

LUI（load upper immediate）使用了与RV32I一样的指令操作码，LUI将20位的U立即数放到目标寄存器***rd***的31-12位，将***rd***的低12位填0。32位结果将会符号扩展到64位。

AUIPC（add upper immediate to pc）使用了与RV32I一样的指令操作码，AUIPC指令用于构建**pc**相对地址，并使用U类格式。AUIPC将在20位U立即数低部添加了12个0位，符号扩展成64位，然后将它加到**pc**上，最后将结果写入寄存器***rd***。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

整数寄存器-寄存器操作

31　　25	24　　20	19　　15	14　　12	11　　7	6　　0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SRA	dest	OP
0000000	src2	src1	ADDW	dest	OP-32
0000000	src2	src1	SLLW/SRLW	dest	OP-32
0100000	src2	src1	SUBW/SRAW	dest	OP-32

ADDW、SUBW是RV64I仅有的指令，它们的定义和ADD、SUB指令相类似，但是它们作用在32位数值上，并产生有符号的32位结果。溢出被忽略，结果的低32位被符号扩展成64位，再写入目标寄存器。

SLL、SRL、SRA分别执行逻辑左移、逻辑右移、算术右移，被移位的操作数是寄存器***rs1***，移位次数是寄存器***rs2***。对于RV64I，只有***rs2***的低6位被认为是移位次数。

SLLW、SRLW、SRAW是RV64I仅有的指令，它们具有类似的指令定义，但是它们作用在32位数值上，并产生有符号的32位结果。移位的次数由***rs2***[4:0]给出。

4.3 Load 和 Store 指令

RV64I将地址空间扩展到64位。执行环境将定义地址空间的哪些部分是可以合法访问的。

31　　20	19　　15	14　　12	11　　7	6　　0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
偏移量[11:0]	基址	宽度	dest	LOAD

31　　25	24　　0	19　　15	14　　12	11　　7	6　　0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
偏移量[11:5]	src	基址	宽度	偏移量[4:0]	STORE

在RV64I中，LD指令将从存储器中把64位数值写入到寄存器***rd***中。

在RV64I中，LW指令将从存储器中把32位数值符号扩展成64位，然后写入到寄存器***rd***中。LWU执行，刚好相反，把存储器中的32位数值零扩展成64位，然后写入到寄存器***rd***中。LH和LHU指令定义类似，但是是针对16位数值。LB和LBU是针对8位数值。SD、SW、SH、SB分别把寄存器***rs2***的低64位、32位、16位、8位写入存储器。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

4.4 系统指令

在RV64I中，CSR指令可以操作64位CSR。特别地，RDCYCLE、RDTIME、RDINSTRET伪指令读取了完整的64位的cycle、time、instret计数器。因此，在RV64I中，不需要RDCYCLEH、RDTIMEH、RDINSTRETH伪指令，并且它们是非法的。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第5章 整数乘法除法的“M”标准扩展，版本 2.0

本章介绍标准整数乘法和除法的指令扩展，它被命名为“M”，并包含针对两个整数寄存器中的数值进行乘法或者除法的指令。

我们将整数乘法和除法从基本内核中分离出来，以简化低端的实现，或者对于某些应用来说，整数乘法和除法操作要么是不常用，要么是最好由附加的加速器来处理。

5.1 乘法操作

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
MULDIV	乘数	被乘数	MUL/MULH[S][U]	dest	OP
MULDIV	乘数	被乘数	MULW	dest	OP-32

MUL指令执行一个XLEN位×XLEN位乘法，并将结果的低XLEN位放置到目标寄存器中。MULH、MULHU、MULHSU执行相同的乘法，分别针对有符号×有符号、无符号×无符号、有符号×无符号乘法，只是将运算结果2×XLEN位的高XLEN位返回。如果同时需要乘法结果的高位和低位，那么建议的代码顺序为：MULH[[S]U] *rdh, rs1, rs2*; MUL *rdl, rs1, rs2*（源寄存器区分符必须按照同样的顺序，并且*rdh*不能是*rs1*或者*rs2*)。因此微体系结构可以将这些融合为一个单一的乘法操作，而不是执行两次分开的乘法。

MULW是RV64I仅有的指令，它将源寄存器的低32位进行相乘，将结果的低32位进行符号扩展，结果放入目标寄存器中。MUL指令可以用于获取64位乘积的高32位，但是有符号的参数，必须使用正确的32位有符号数值，反之，无符号的参数，必须使得它们的高32位清零。

5.2 除法操作

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
MULDIV	除数	被除数	DIV[U]/REM[U]	dest	OP
MULDIV	除数	被除数	DIV[U]W/REM[U]W	dest	OP-32

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

DIV和DIVU指令分别执行有符号、无符号的XLEN位整数除以XLEN位整数除法操作。REM、REMU给出了相应除法的余数。如果同时需要商和余数，那么建议的代码顺序为：DIV[U] ***rdq,rs1,rs2***; REM[U] ***rdr,rs1,rs2***（***rdq***不能是***rs1***或者***rs2***）。因此微体系结构可以将这些融合为一个单一的除法操作，而不是执行两次分开的除法。

DIVW、DIVUW是RV64I仅有的指令，它将寄存器***rs1***的低32位除以寄存器***rs2***的低32位，操作数分别被当做有符号整数和无符号整数，将32位商进行符号扩展，放入目标寄存器中。REMW和REMUW是RV64I仅有指令，它们分别提供了对应除法的有符号或者无符号余数。REMW和REMUW指令都会将它们的32位结果进行符号扩展到64位。

除以零和除法溢出的语义如表 5.1所示。除以零，结果的商的所有位被置为1，也即是说，对于无符号除法来说，商是2^{XLEN}-1，对于有符号除法来说，商是-1。除以零，结果的余数等于被除数。有符号除法溢出仅当用最小的负整数，-2^{XLEN}-1，除以-1时，才会出现。有符号除法溢出的商等于被除数，余数等于零。无符号除法不会产生溢出。

条件	被除数	除数		DIVU	REMU	DIV	REM
除以0	x	0		2^{XLEN-1}	x	-1	x
溢出（仅有符号）	-2^{XLEN-1}	-1		—	—	-2^{XLEN-1}	0

表 5.1：除以 0 和除法溢出的语义

我们考虑过在整数除以零的时候，触发异常，这个异常在绝大多数执行环境里，会导致一个自陷。然而，这将成为标准 ISA 中唯一的算术自陷（浮点异常将设置标志，并写入缺省值，但不会引起自陷），并在此情形下，需要语言实现者与执行环境的自陷处理函数交互。更进一步，当语言标准要求一个除以零异常必须导致一个立即的控制流改变时，只需要在每一个除法操作时增加一条分支指令即可，并且这条分支指令可被放到除法之后，通常情形下它基本上被预测为不跳转的，这将增加一点点性能开销。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第6章 原子性指令的“A”标准扩展，版本2.0

标准原子性指令扩展被称为“A”扩展，包含了对存储器执行原子性读-写-修改的指令，以支持运行在同一个存储器空间中的多个RISC-V线程之间的同步操作。有两种原子性指令，一种是load-reserved/store-conditional指令，另一种是原子性fetch-and-op存储器指令。两种类型的原子性指令都支持各种存储器一致性排序，包括乱序（unordered）、获取（acquire）、释放（release）和顺序一致性语义。这些指令使得RISC-V可以支持RCsc存储器一致性模型[6]。

经过大量的争论，语言社区和体系结构社区看来最后都认同释放一致性（release consistency）作为标准的存储器一致性模型，因此RISC-V原子性操作正是围绕这个模型构建的。

6.1 原子性操作的指定顺序

基本的RISC-V ISA有一个放松的存储器模型（relaxed memory model），使用FENCE指令来强加额外的顺序性约束。地址空间被执行环境划分为两类：存储器域和I/O域，FENCE指令提供了其中一个地址域或者两个地址域的按序访问选项。

为了更高效地支持释放一致性（release consistency）[6]，每一条原子性指令有两位，***aq***位和***rl***位，用于指定其他RISC-V线程看到的额外的存储器顺序性。这两位排序了两个地址域（存储器域和I/O域）中一个的访问，这依赖于原子性指令正在访问哪个域。对另外一个域的访问并没有隐式的顺序约束，需要使用一条FENCE指令来在两个域实现按序访问（**译者注：应该是FENCE指令只作用在其中的一个域**）。

如果两位都被清零，则没有额外的顺序性约束强加于原子性存储器操作。如果只有***aq***位被置为1，则原子性存储器操作被认为是***获取访问***的（***acquire access***），也就是说，在获取存储器操作之前，此RISC-V线程后续的存储器操作是看不到的（no following memory operations on this RISC-V thread can be observed to take place before the acquire memory operation.）。如果只有***rl***位被置为1，则原子性存储器操作被认为是***释放访问***的（***release access***），也就是说，在此RISC-V线程的任何前面的存储器操作完成之前，释放存储器操作是看不到的（the release memory operation can not be observed to take place before any earlier memory operations on this RISC-V thread.）。如果***aq***位和***rl***位都被置为1，则原子性存储器操作是***顺序一致性***的（***sequentially consistent***），在同样的RISC-V线程中，在任何前面的存储器操作完成之前，或者在任何后续的存储器操作完成之后，它们都是不可见的（cannot be observed to happen before any earlier memory operations or after any later memory operations in the same RISC-V thread），只能被任何其他按照同样全局顺序的线程看到，它们也全部使用了顺序一致性原子性存储器操作，对同一个地址域（can only be observed by any other thread in the same global order of all sequentially consistent atomic memory operations to the same address domain.）。

---

*理论上说，**aq** 位和 **rl** 位的定义允许没有全局原子性 store 的实现。然而当*

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

__aq__ 位和 __rl__ 位都被置为 1 时，我们需要原子性操作的完全顺序一致性，也意味着全局原子性 store，还加上获取和释放语义。实际上，硬件实现通常实现了全局原子性 store，包括在本地处理器的顺序化规则，以及 single-writer cache 一致性协议。

6.2 Load-reserved/store-conditional 指令

31　27	26	25	24　20	19　15	14　12	11　7	6　0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
LR	顺序		0	地址	宽度	dest	AMO
SC	顺序		src	地址	宽度	dest	AMO

针对单个存储器字的复杂原子性操作，是由load-reserved（LR）指令和store-conditional（SC）指令来完成的。LR将***rs1***存储器地址的字读出，符号扩展后，放入***rd***，同时在存储器地址上注册一个预约（registers a reservation on the memory address）。SC将***rs2***中的字保存到***rs1***存储器地址中，提供给还存在这个地址上一个有效的预约（provided a valid reservation still exists on that address）。如果成功，则SC将0写入***rd***，失败则将一个非零的代码写入***rd***。（**译者注：简单地说，LR指令从存储器读一个数值，同时处理器会监视这个存储器地址，看它是否会被其他处理器修改；SC指令发现在此期间没有其他处理器修改这个值，则将新值写入该地址。因此一个原子的LR/SC指令对，就是LR读取值，进行一些计算，然后试图保存新值。如果保存失败，那么需要重新开始整个序列。**）

比较和交换（CAS）指令和 LR/SC 指令都可以用来构建 lock-free 的数据结构。经过大量的讨论，我们选择了 LR/SC，由于下列几个原因：1）CAS 遭受到 ABA 问题，而 LR/SC 避免了这个问题，因为它监视了对此地址的所有访问，而不仅仅是检查数据值是否发生改变；2）CAS 指令需要一种新的整数指令格式，以支持 3 个源操作数（地址、比较值、交换值），还有一个不同的存储器系统消息格式，它们将会导致微体系结构复杂化；3）更进一步，为了避免 ABA 问题，其他系统提供了一个双宽的 CAS（DW-CAS）以允许检测一个计数器并递增一个数据字。这就要在一条指令中读取 5 个寄存器、写 2 个寄存器，同时还需要一个新的、更大的存储器系统消息类型，进一步导致微体系结构复杂化；4）LR/SC 提供了一种实现许多原语（primitive）更加有效的实现，它只需要一次 load，而 CAS 需要 2 次 load（在 CAS 指令之前一次 load，以获得数值进行推测性计算，然后作为 CAS 指令一部分的第二次 load，以检查在更新前，该数值是否发生了变化）。

LR/SR 相对于 CAS 的一个主要缺点是活锁（livelock），我们通过如下所述的一种最终前向推进的体系结构保证（an architected guarantee of eventual forward progress）来加以避免。另外一个关心的问题是，这是否会影响到当前 x86 体系结构（它是 DW-CAS 的），使得移植同步库、其他假设 DW-CAS 是基本的机器原语的软件复杂化。一个可能减轻任务的因素是，x86 最近加入了事务

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

存储器指令（transactional memory instructions），这将导致从 DW-CAS 上进行转移。

错误代码1被保留作为未指定的错误。其他错误代码此时还是保留的，可移植性软件应当仅仅假设错误代码就是非零值。LR、SC指令工作在自然对齐的64位（仅RV64）或者32位字存储器边界。非对齐寻址将产生非对齐地址异常。

我们保留了错误代码值 1 表示“未指定”，这样简单的实现就可以利用 SLT/SLTU 指令已有的 mux 来返回值。更特定的错误代码可能在未来的版本中定义或者由 ISA 的扩展定义。

在标准A扩展中，某些限定的LR/SC序列必须确保以便最后能够成功。LR/SC序列的静态代码加上当失败时重试这个序列的代码，必须由不超过16条整数指令的代码顺序存放在存储器中构成。为了确保这个序列最终能够成功，在LR指令和SC指令之间执行的动态代码，只能来自于基本“I”的子集，不能包括load指令、store指令、向后跳转指令或者向后的分支指令、FENCE指令、SYSTEM指令。当失败时重试LR/SC序列的代码可以包含向后跳转和/或分支指令，以便重新执行LR/SC序列，其他约束相同。SC指令的地址必须和最近一次执行的LR指令地址相同。那些不符合这些要求的LR/SC例程，在某些实现上，经过一定次数的尝试，也许会成功，但是没办法确保最终会成功。

CAS 的一个优点在于它总能确保某些线程最终会向前推进，然而在某些系统上，LR/SC 原子性序列可能陷入无限的活锁。为了避免这种情况，我们为 LR/SC 序列加入了一种最终前向推进的体系结构保证。对 LR/SC 序列内容的限制，允许一种实现，这种实现在 LR 指令时捕捉一个 cache line，并通过在较短的时间内拖延一下远程的 cache 来完成 LR/SC 序列。中断和 TLB 缺失可能会导致这个保留预约丢失，但是最终这个原子性序列是可以完成的。我们将 LR/SC 序列的长度限定在基本 ISA 中 64 个连续指令字节中，以避免过度限制指令 cache 和 TLB 的大小和关联性。相类似的，我们在序列中不允许出现其他的 load 和 store 指令，以避免限制数据 cache 的关联性。分支和跳转指令的限制，限制了序列花费的时间。禁止使用浮点操作和整数乘法/除法，使得在缺少相应硬件支持的实现上，简化了操作系统仿真这些指令。

一个实现可以保留任意的存储器空间子集，对单个硬件线程而言，可以在这个存储器空间子集上同时有多个LR指令的预约。在此线程的一条SC指令会成功，如果在此SC指令和最近一次LR指令预约这个地址之间，没有来自其他硬件线程的访问这个地址。注意这条LR可能有一个不同的地址参数，但是将SC的地址作为存储器子集的一部分预约。按照这个模型，在拥有地址翻译的系统中，一条SC可以允许成功，如果前面的LR使用了一个别名（不同的虚拟地址）预约了相同的位置，但是也可以允许失败，如果虚拟地址不同。如果有一个来自其他硬件线程对此地址可观察的存储器访问，或者在此硬件线程上有一个上下文切换，或者在此硬件线程执行一条特权异常返回指令，那么SC指令必须失败。

本规范明确允许实现支持更强大的、更宽保证的实现，前提是它们不会破坏受限序列的原子性保证。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

LR/SC可以用来构建lock-free的数据结构。一个用LR/SC实现一个比较并交换的函数如图6.1所示。如果采用内联方式实现，比较和交换功能仅仅需要3条指令。

```
        # a0保存存储器地址
        # a1保存期望的值
        # a2保存理想的值
        # a0返回值，成功返回0，其他情况返回非0
cas:
        lr.w    t0, (a0)        # 读取原始值
        bne     t0, a1, fail    # 不匹配，失败
        sc.w    a0, a2, (a0)    # 试图更新
        jr      ra              # 返回
fail:
        li      a0, 1           # 设置返回值为失败
        jr      ra              # 返回
```

图 6.1：使用 LR/SC 指令实现比较并交换的示例代码

在立即处理的LR指令之前，另一个RISC-V线程是永远看不到SC指令的。由于LR/SC序列的原子性特性，在LR指令和一个成功的SC指令之间，任何线程的存储器操作都看不到。这个LR/SC序列可以通过设置SC指令的***aq***位而被赋予获取（acquire）语义。这个LR/SC序列可以通过设置LR指令的***rl***位而被赋予释放（release）语义。设置LR指令的***aq***位和***rl***位都为1，设置SC指令的***aq***位为1，使得LR/SC指令序列对于其他顺序一致性原子性操作来说，是顺序一致性的。

如果LR和SC的所有***aq***、***rl***位都被清零，那么在来自同一个RISC-V线程的附近存储器操作之前或者之后，可以看到LR/SC指令序列（LR/SC sequence can be observed to occur before or after surrounding memory operations from the same RISC-V thread）。这很适合，当使用LR/SC序列来实现一个并行归约操作（a parallel reduction operation）时。

通常来说，一个多字的原子性原语（a multi-word atomic primitive）是理想的，但是就如何构造这个原语，并确保向前推进增加的系统复杂性，还存在大量的争论。我们当前的想法是，引入一个小的、有限容量的事务存储器缓冲区，以及原来的事务存储器的一些行，这将作为可选的标准扩展"T"。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

6.3 原子性存储器操作

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
AMOSWAP.W/D	顺序		src	地址	宽度	dest	AMO
AMOADD.W/D	顺序		src	地址	宽度	dest	AMO
AMOAND.W/D	顺序		src	地址	宽度	dest	AMO
AMOOR.W/D	顺序		src	地址	宽度	dest	AMO
AMOXOR.W/D	顺序		src	地址	宽度	dest	AMO
AMOMAX[U].W/D	顺序		src	地址	宽度	dest	AMO
AMOMIN[U].W/D	顺序		src	地址	宽度	dest	AMO

原子性存储器操作（AMO）为多处理器同步执行读-修改-写操作，并编码为R类指令格式。这些AMO指令原子性地从***rs1***地址读取数据值、将这个值写入寄存器***rd***、在这个值和***rs2***的原始值上施加一个二进制操作、然后把结果保存到***rs1***地址的存储器中。AMO指令可以对存储器中的64位（仅RV64）或者32位字进行操作。对于RV64，32位的AMO指令总是将值进行符号扩展保存到***rd***中。保存在***rs1***中的地址必须与操作数的大小对齐（也即，对于64位字是8字节边界，对于32位字是4字节边界）。如果地址没有自然对齐，将会产生一个非对齐地址异常。

支持的操作包括交换、整数加、逻辑AND、逻辑OR、逻辑XOR和有符号、无符号整数最大值和最小值。没有顺序性约束，这些AMO指令可用于实现并行归约操作（parallel reduction operation），此时通常返回值通过写入**x0**而被丢弃。

我们提供了fetch-and-op风格的原子性原语，因为相比起LR/SC或者CAS指令而言，它能更好地扩展到高度并行的系统。一个简单的实现，可以使用LR/SC原语来实现AMO指令。更复杂的实现，则可以在存储器控制器上实现AMO，并在目标寄存器是x0时，优化读取原始数据的操作。

为了帮助实现多处理器同步，AMO指令可选地提供释放一致性语义。如果***aq***位被置为1，则在本线程AMO指令或者存储器操作执行的AMO之前，看不到在当前RISC-V线程中的后续存储器操作（no later memory operations in this RISC-V thread can be observed to take place before the AMO or memory operations preceding the AMO in this thread）。相反的，如果***rl***位被置为1，则在这个RISC-V线程存储器访问执行AMO之前，其他RISC-V线程看不到AMO指令（other RISC-V threads will not observe the AMO before memory accesses preceding the AMO in this RISC-V thread）。

AMO指令被设计成能够有效地支持C11和C++11的存储器模型。虽然FENCE R、RW指令足以实现获取（acquire）操作，而FENCE RW、W指令足以实现释放（release）操作，但是两者相对于设置了相应aq或者rl位的AMO

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

指令而言，都暗含了额外的不必要的顺序化。

AMO指令也可用于提供顺序化一致性load和store。一个顺序化一致性load可以实现为一条AMOADD ***x0***指令，其***aq***位和***rl***位都设置为1。一个顺序化一致性store可以实现为一条AMOSWP指令，它将旧值写入**x0**，其***aq***位和***rl***位都设置为1。

一个使用test-and-set spinlock实现关键段保护的示例代码序列如图 6.2所示。注意到在关键段前面，第一个AMO指令设置了***aq***位，以便按序获取锁，而在释放锁之前，第二个AMO指令设置了***rl***位，以便按序进入关键段。

```
        li              t0, 1           # 初始化交换值
again:
        amoswap.w.aq    t0, t0, (a0)    # 试图获取锁
        bnez            t0, again       # 如果获取失败，重试
        # •••
        # 关键段
        # •••
        amoswap.w.rl    x0, x0, (a0)    # 通过保存0来释放锁
```

图 6.2：互斥访问示例代码。a0 是锁的地址。

我们推荐在获取锁和释放锁时，都使用上面的AMO 交换风格，以简化推测锁省略的实现（simplify the implementation of speculative lock elision）[20]。

存在导致原子操作实现复杂化的风险，如果锁的值和交换的值相同，微体系结构可以省略掉获取交换中的 store，以避免污染保存在一个共享或者exclusive 干净状态的cache line。其效果类似于“测试-测试-设置锁”，但具有更短的代码路径。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第7章 单精度浮点的“F”标准扩展，版本 2.0

本章描述单精度浮点的标准指令集扩展，被命名为“F”，加入了符合IEEE-754 2008算术标准[10]的单精度浮点计算指令。

7.1 F 寄存器状态

F扩展加入了32个浮点寄存器，**f0-f31**，每个是32位宽度，一个浮点控制和状态寄存器**fcsr**，它包含了操作模式和浮点单元的异常状态。这个额外的状态如图 7.1所示。在RISC-V ISA中，我们使用术语FLEN来描述浮点寄存器的宽度，对于F单精度浮点扩展来说，FLEN=32。绝大多数浮点指令对浮点寄存器文件中的值进行操作。浮点load和store指令在寄存器和存储器之间传输浮点值。同时也提供了从整数寄存器文件读写数值的指令。

我们考虑过对整数和浮点值采用一个统一的寄存器文件，因为这将简化软件寄存器分配和调用约定，减少总的用户状态。然而，分开的组织方式，在一个给定的指令宽度条件下，增加了可以访问的寄存器总数，为宽的超标量发射提供足够的寄存器文件端口数，支持去耦合的浮点单元体系结构，简化内部浮点编码技术。对分开的寄存器文件体系结构，编译器支持和调用约定已经被人熟知，在浮点寄存器文件状态上使用脏位，可以减少上下文切换的开销。

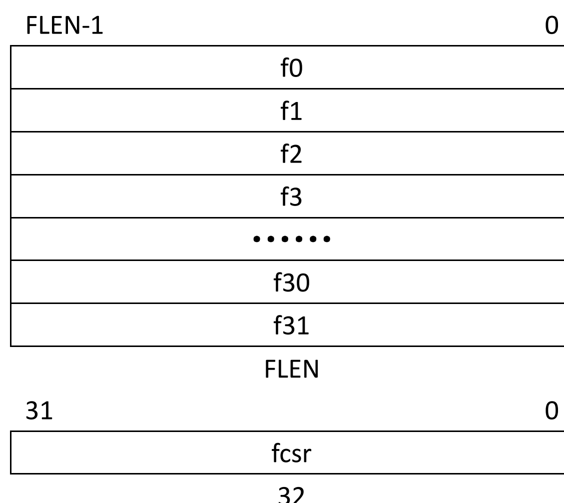


图 7.1：RISC-V 标准 F 扩展单精度浮点状态

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

7.2 浮点控制和状态寄存器

浮点控制和状态寄存器**fcsr**是一个RISC-V控制和状态寄存器（CSR）。它是一个32位的读/写寄存器，用于为浮点算术操作选择动态舍入模式，并保存产生的异常标志，如图 7.2所示。

图 7.2：浮点控制和状态寄存器

fcsr寄存器可以被FRCSR和FSCSR指令进行读写，它们是由底层的CSR访问指令实现的。FRCSR指令通过将**fcsr**寄存器复制到整数寄存器***rd***实现读操作。FSCSR通过将**fcsr**原来的值复制到***rd***，然后将整数寄存器***rs1***中的值写入**fcsr**来实现写操作。

fcsr中的字段也可以通过不同的CSR寻址方式来单独访问，并为这些访问，在汇编语言上定义了伪指令。FRRM指令读取了舍入模式字段***frm***，并将它复制到整数寄存器***rd***的最低3位中，其他位被置为0。FSRM通过将***frm***的旧值复制到整数寄存器***rd***中、然后将整数寄存器***rs1***最低3位中的新值写入***frm***，来实现交换***frm***的值。FRFLAGS、FSFLAGS定义类似，但针对的是产生的异常标志字段***fflags***。其他的伪指令FSRMI和FSFLAGSI使用立即数值而不是***rs1***来进行交换。

浮点操作要么使用编码在指令中的舍入模式，要么使用保存在***frm***中的动态舍入模式。舍入模式的编码如表 7.1所示。指令编码***rm***字段中的编码如果为111，则选择保存在***frm***中的动态舍入模式。如果***frm***被设置为一个非法值（101-111），那么后续任何试图执行使用动态舍入模式的浮点操作的指令，都会引起一个非法指令自陷。一些指令尽管有***rm***字段，但不受舍入模式的影响，它们应当把***rm***字段设置为RNE（000）。

舍入模式	助记符	含义
000	RNZ	向最接近的值舍入，首选“偶数”值
001	RTZ	向零舍入
010	RDN	向下舍入（向－∞）
011	RUP	向上舍入（向 + ∞）
100	RMM	向最接近的值舍入，首选最大值
101		非法。保留给未来使用
110		非法。保留给未来使用
111		在指令的rm字段，选择动态舍入模式； 在舍入模式寄存器，非法。

表 7.1：舍入模式编码

C99 语言标准实际上强制规定了动态舍入模式寄存器。

产生的异常标准指示了自从上次被软件重置以来，已经由浮点指令产生的异常情况，如

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

表 7.2所示。

标志助记符	标志含义
NV	非法操作
DZ	除以0
OF	上溢
UF	下溢
NX	不精确

表 7.2：产生的异常标志编码

如同标准允许的那样，我们在基本ISA 中并不支持在浮点异常上产生自陷，而是需要软件明确的对标志进行检查。我们考虑过加入一些分支指令，它们直接由浮点已发生的异常内容来控制，但是最终我们选择去掉这些指令以保持ISA 简单化。

7.3 NaN 生成和传递

除非另有说明，如果一个浮点操作的结果是NaN（Not a Number），则结果就是规定的（canonical）NaN。这个规定的NaN具有一个正的符号位，并且除了MSB（也称为quiet位）之外，其有效数的位都是0。对于单精度浮点，这对应于模式0x7fc00000。

对于FMIX和FMAX指令，如果其中至少一个输入是SNaN（signaling NaN）（**译者注：SNaN，其MSB位为0，在参与运算的时候，就会引起异常**），或者两个输入都是QNaN（quiet NaN），那么运算的结果就是规定的NaN。如果一个操作数是QNaN，另一个操作数不是一个NaN，则结果是非NaN的那个操作数。

符号注入指令（FSGNJ、 FSGNJN、 FSGNJX）不会产生规定的NaN；它们直接操作底层的位模式。

我们考虑过传递 NaN 有效载荷（payload），如同标准推荐的那样，但是这个决定将增加硬件的代价。而且，由于这个特性在标准中是可选的，因此它不能用于可移植的代码。

实现者可以自由地以非标准扩展的形式提供一种 NaN 传递有效载荷方案，用一种非标准的操作模式来使能。然而，上述的规定的 NaN 方案必须总是被支持的，并且应该是缺省的模式。

我们要求当出现异常情况时，各种实现应当返回标准强制的缺省值，而不需要用户级软件的进一步干涉（这与Alpha ISA 浮点自陷栅栏不同）。我们相信完全由硬件处理异常情况正在变得越来越普遍，所以希望避免用户级 ISA 优化其他方法，变得复杂化。实现也可以总是（碰到异常时）产生自陷，在机器模式软件处理函数里提供异常的缺省值。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

7.4 非规格化数算术

在非规格化数上的操作处理遵照 IEEE 754-2008 标准。

用 IEEE 标准的说法，极小（tininess）是在舍入之后检测的——也就是说，只有在舍入后结果是非规格化数时，才会产生下溢异常，即使未舍入的结果是一个非规格化数。

在舍入后检测极小导致更少的虚假的下溢信号。

7.5 单精度 load 和 store 指令

浮点load和store指令使用与整数ISA相同的“基址+偏移量”的寻址方式，寄存器***rs1***保存着基址，字节偏移量是一个12位有符号数。FLW指令从存储器中将一个单精度浮点值读入到浮点寄存器***rd***中。FSW指令将浮点寄存器***rs2***中的单精度浮点值写入到存储器中。

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	width	rd	opcode
12	5	3	5	7
偏移量[11:0]	基址	W	dest	LOAD-FP

31 — 25	24 — 0	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:5]	rs2	rs1	width	imm[4:0]	opcode
7	5	5	3	5	7
偏移量[11:5]	src	基址	W	偏移量[4:0]	STORE-FP

FLW指令和FSW指令只有在有效地址是自然对齐时，才能保证执行的原子性。

7.6 单精度浮点计算指令

具有1个或者2个操作数的浮点算术指令，使用R类指令格式，其主操作码是OP-FP。FADD.S、FSUB.S、FMUL.S、FDIV.S指令在***rs1***和***rs2***之间分别执行单精度浮点的加、减、乘、除运算，并将结果写入***rd***。FMIN.S、FMAX.S指令分别将***rs1***、***rs2***中的较小者或者较大者写入到***rd***中。FSQRT.S指令计算***rs1***的平方根，并把结果写入***rd***。

2位的浮点格式字段***fmt***，其编码如表 7.3所示。对于F扩展中的所有指令，它被设置为S（00）。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

fmt字段	助记符	含义
00	S	32位单精度
01	D	64位双精度
10	—	保留
11	Q	128位四精度

表 7.3：格式字段编码

所有执行舍入的浮点操作指令，可以通过***rm***字段选择其舍入模式，舍入模式编码如表7.1所示。

31　　27	26　25	24　　20	19　　15	14　　12	11　　7	6　　0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	S	src2	src1	RM	dest	OP-FP
FMUL/FDIV	S	src2	src1	RM	dest	OP-FP
FMIN-FMAX	S	src2	src1	MIN/MAX	dest	OP-FP
FSQRT	S	0	src	RM	dest	OP-FP

浮点融合乘加指令需要一种新的标准指令格式。R4类指令指明了3个源寄存器（***rs1***、***rs2***、***rs3***）和一个目的寄存器（***rd***）。这种格式仅被浮点融合乘加指令使用。浮点融合乘加指令将***rs1***、***rs2***中的值相乘，然后用未舍入的乘积加上或者减去***rs3***中的值，可选地取负舍入的结果。FMADD.S指令计算***rs1×rs2＋rs3***；FMSUB.S指令计算***rs1×rs2－rs3***；FNMSUB.S指令计算－***(rs1×rs2－rs3)***；FNMADD.S指令计算－***(rs1×rs2＋rs3)***。

31　　27	26　25	24　　20	19　　15	14　　12	11　　7	6　　0
rs3	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
src3	S	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

7.7 单精度浮点转换和传输指令

浮点-整数和整数-浮点转换指令被编码在OP-FP主操作码空间。FCVT.W.S、FCVT.L.S指令将浮点寄存器***rs1***中的一个浮点数分别转换为一个有符号的32位或者64位整数，保存到整数寄存器***rd***中。FCVT.S.W、FCVT.S.L指令分别将整数寄存器***rs1***中的一个有符号的32位或者64位整数，转换成一个浮点数，保存到浮点寄存器***rd***中。FCVT.WU.S、FCVT.LU.S、FCVT.S.WU、FCVT.S.LU指令转换到无符号整数值，或者从无符号整数值转换。FCVT.L[U].S和FCVT.S.L[U]指令在RV32中是非法的。如果舍入的结果在目标格式中是不能被表示的，它将被裁剪为最接近的值，并

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

且invalid标志被设置。表 7.4给出了FCVT.int.S的有效输入范围和无效输入时的行为。

	FCVT.W.S	**FCVT.WU.S**	**FCVT.L.S**	**FCVT.LU.S**
最小有效输入（舍入后）	-2^{31}	0	-2^{63}	0
最大有效输入（舍入后）	2^{31}-1	2^{32}-1	2^{63}-1	2^{64}-1
超出范围的负输入时的输出	-2^{31}	0	-2^{63}	0
－∞时的输出	-2^{31}	0	-2^{63}	0
超出范围的正输入时的输出	2^{31}-1	2^{32}-1	2^{63}-1	2^{64}-1
＋∞或者NaN时的输出	2^{31}-1	2^{32}-1	2^{63}-1	2^{64}-1

表 7.4：浮点到整数转换的范围和无效输入时的行为

所有的浮点到整数转换指令、整数到浮点转换指令，都按照***rm***字段进行舍入。一个浮点寄存器可以使用FCVT.S.W ***rd,x0***指令初始化为浮点的正0，这永远不会产生异常。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.int.fmt	S	W[U]/L[U]	src	RM	dest	OP-FP
FCVT.fmt.int	S	W[U]/L[U]	src	RM	dest	OP-FP

浮点到浮点符号注入指令（sign-injection instruction），FSGNJ.S、FSGNJN.S、FSGNJX.S指令从***rs1***中读取除了符号位以外的所有位。对于FSGNJ指令，结果的符号位是***rs2***的符号位；对于FSGNJN指令，结果的符号位是***rs2***的符号位取反；对于FSGNJX指令，结果的符号位是***rs1***的符号位和***rs2***的符号位进行XOR运算。符号注入指令并不设置浮点异常标志。注意到，FSGNJ.S ***rx,ry,ry***将***ry***传输到***rx***（汇编语言伪指令FMV.S ***rx,ry***）；FSGNJN.S ***rx,ry,ry***将***ry***取负后传输到***rx***（汇编语言伪指令FNEG.S ***rx,ry***）；FSGNJX.S ***rx,ry,ry***将***ry***的绝对值传输到***rx***（汇编语言伪指令FABS.S ***rx,ry***）；

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	S	src2	src1	J[N]/JX	dest	OP-FP

符号注入指令是 John Hauser 提出来的。这些指令提供了浮点 MV、ABS 和 NEG，并支持其他一些操作，包括 IEEE 复制符号（copySign）操作和抽象数学函数库里的符号操作。虽然 MV、ABS 和 NEG 只需要一个寄存器操作数，然而 FSGNJ 指令需要两个，看起来大多数微体系结构不会对这些不常使用的指令加入减少寄存器读数目的优化。即使如此，一个微体系结构可以简单地检测到 FSGNJ 指令的两个源寄存器是一样的时候，此时只需要读取一个寄存器拷贝。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

提供了在浮点寄存器和整数寄存器直接进行位传输的指令。FMV.X.S将浮点寄存器***rs1***中以IEEE 754-2008编码的单精度浮点值，传输到***rd***寄存器的低32位。对RV64，目标寄存器***rd***的高32位，被浮点数的符号位填充。FMV.S.X将整数寄存器***rs1***的低32位（以IEEE 754-2008编码的单精度浮点值），传输到浮点寄存器***rd***中。在传输过程中，位不会被修改，特别地，非规定NaN有效载荷保持不变。

31　　27	26　25	24　　20	19　　15	14　　12	11　　7	6　　0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FMV.X.fmt	S	0	src	000	dest	OP-FP
FMV.fmt.X	S	0	src	000	dest	OP-FP

基本浮点ISA被定义为允许各种实现在寄存器中使用内部编码的浮点格式，以简化非规格化数处理，并可能减少功能单元的延迟。为此，基本ISA通过定义直接读写整数寄存器文件的转换和比较指令，避免了在浮点寄存器中表示整数值。这也消除了许多明确需要在整数和浮点寄存器之间传输的常见情形，对于常见的混合格式代码序列来说，减少了指令数，缩短了关键路径。

7.8 单精度浮点比较指令

浮点比较指令在浮点寄存器***rs1***和***rs2***之间执行指定的比较（等于、小于、小于等于），并将结果的布尔值写入整数寄存器***rd***中。

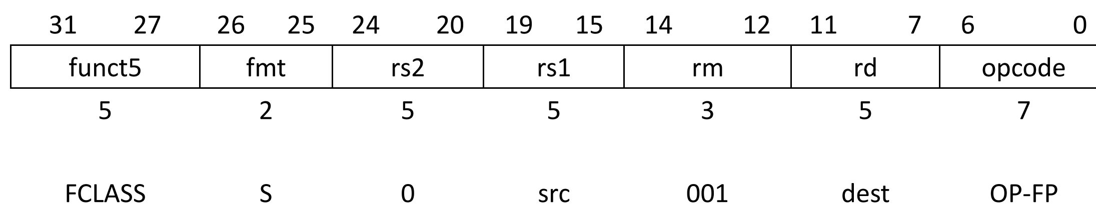
FLT.S、FLE.S执行IEEE 754-2008标准称为signaling的比较：也就是说，如果任一输入是NaN，则产生一个无效操作（Invalid Operation）异常。FEQ.S执行安静（quiet）的比较：只有输入是signaling NaN时，才产生一个无效操作（Invalid Operation）异常。对于所有3条指令，如果任一操作数是NaN，则结果是0。

31　　27	26　25	24　　20	19　　15	14　　12	11　　7	6　　0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCMP	S	src2	src1	EQ/LT/LE	dest	OP-FP

7.9 单精度浮点分类指令

FCLASS.S指令检测浮点寄存器***rs1***中的值，并将一个指示这个浮点数类别的10位掩码，写入整数寄存器***rd***中。这个掩码的格式如表 7.5所示。***rd***中的对应位被置为1，如果具有该属性，否则被清零。***rd***中的其他所有位都被清零。注意到***rd***中仅有1位被置位。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.



Rd寄存器位	含义
0	rs1是－∞
1	rs1是一个负的规格化数
2	rs1是一个负的非规格化数
3	rs1是－0
4	rs1是＋0
5	rs1是一个正的非规格化数
6	rs1是一个正的规格化数
7	rs1是＋∞
8	rs1是一个signaling NaN
9	rs1是一个quiet NaN

表 7.5：FCLASS 指令结果的格式

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第8章 双精度浮点的“D”标准扩展，版本2.0

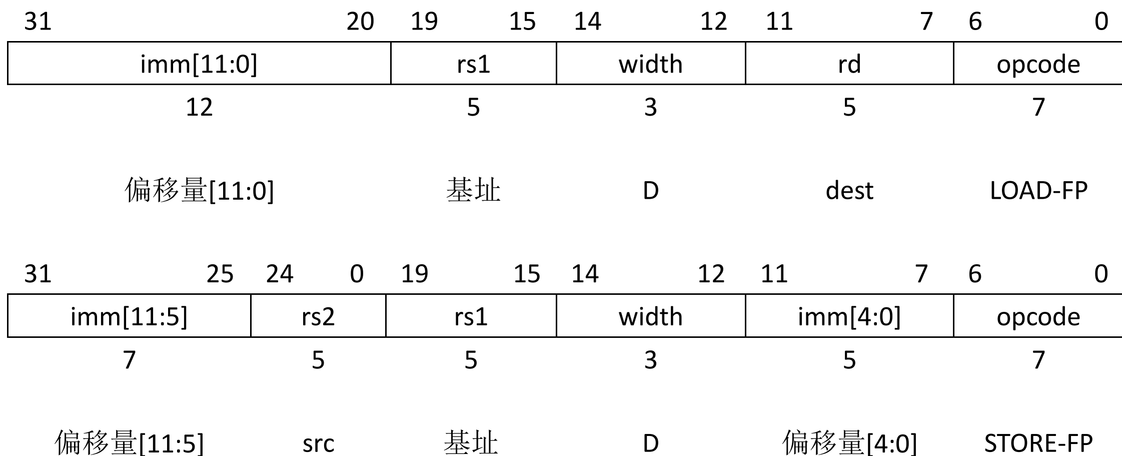
本章描述标准双精度浮点指令集扩展，被命名为“D”，加入了符合IEEE-754 2008算术标准的双精度浮点计算指令。D扩展依赖于基本的单精度指令子集F。

8.1 D 寄存器状态

D扩展加宽了32个浮点寄存器，**f0-f31**，到64位（图 7.1中FLEN=64）。

8.2 双精度 load 和 store 指令

FLD指令从存储器读取一个双精度浮点值，写入到浮点寄存器***rd***中。FSD指令从浮点寄存器读取双精度浮点值，写入到存储器中。



如果一个浮点寄存器保存了一个单精度浮点值，必须确保使用FSD指令将寄存器的值保存到存储器之后，再使用FLD指令重新读取该值，那么将在寄存器中重新生成原始的单精度浮点值。除了这个特性之外，保存在存储器中的数据格式并没有明确定义。

用户级代码可能并不知道当前保存在浮点寄存器中数值的类型，但是可以保存和恢复寄存器值。一种常见的情形就是调用者保存寄存器，但这对于实现可变参数和用户级线程库也是必须的。

FLD指令和FSD指令只有在有效地址是自然对齐且XLEN≥64时，才能保证执行的原子性。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

8.3 双精度浮点计算指令

双精度浮点计算指令的定义与它们的单精度版本相类似，但是它们作用在双精度操作数上，并产生双精度结果。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	D	src2	src1	RM	dest	OP-FP
FMUL/FDIV	D	src2	src1	RM	dest	OP-FP
FMIN-FMAX	D	src2	src1	MIN/MAX	dest	OP-FP
FSQRT	D	0	src	RM	dest	OP-FP

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
src3	D	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

8.4 双精度浮点转换和传输指令

浮点-整数和整数-浮点转换指令被编码在OP-FP主操作码空间。FCVT.W.D、FCVT.L.D指令将浮点寄存器***rs1***中的一个双精度浮点数分别转换为一个有符号的32位或者64位整数，保存到整数寄存器***rd***中。FCVT.D.W、FCVT.D.L指令分别将整数寄存器***rs1***中的一个有符号的32位或者64位整数，转换成一个双精度浮点数，保存到浮点寄存器***rd***中。FCVT.WU.D、FCVT.LU.D、FCVT.D.WU、FCVT.D.LU指令转换到无符号整数值，或者从无符号整数值转换。FCVT.L[U].D和FCVT.D.L[U]指令在RV32中是非法的。FCVT.int.D指令的有效输入范围和非法输入时的行为，和FCVT.int.S指令一样。

所有的浮点到整数转换指令、整数到浮点转换指令，都按照***rm***字段进行舍入。注意FCVT.D.W[U]指令总是产生精确的结果，并不受到舍入模式的影响。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.int.fmt	D	W[U]/L[U]	src	RM	dest	OP-FP
FCVT.fmt.int	D	W[U]/L[U]	src	RM	dest	OP-FP

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

双精度浮点到单精度浮点和单精度浮点到双精度浮点转换指令被编码在OP-FP主操作码空间，它们的源寄存器和目的寄存器都是浮点寄存器。指令中的***rs2***字段编码了源数据类型，***fmt***字段编码了目的操作数类型。FCVT.S.D指令根据***rm***字段进行舍入，FCVT.D.S永远不会舍入。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.fmt.fmt	S	D	src	RM	dest	OP-FP
FCVT.fmt.fmt	D	S	src	RM	dest	OP-FP

浮点到浮点符号注入指令，FSGNJ.D、FSGNJN.D、FSGNJX.D，它们的定义与单精度符号注入指令相类似。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	D	src2	src1	J[N]/JX	dest	OP-FP

仅对于RV64，提供了在浮点寄存器和整数寄存器之间进行位传输的指令。FMV.X.D将浮点寄存器***rs1***中的双精度浮点值，以IEEE 754-2008标准编码，传输到***rd***整数寄存器中。如果最后一次写入源浮点寄存器的值是一个单精度浮点值的话，则FMV.X.D返回的值是未定义的，除了将这个值重新移动回浮点寄存器将重新生成原始的单精度浮点值这个特性。FMV.D.X将以IEEE 754-2008标准编码的双精度浮点值，从整数寄存器***rs1***传输到浮点寄存器***rd***中。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FMV.X.fmt	D	0	src	000	dest	OP-FP
FMV.fmt.X	D	0	src	000	dest	OP-FP

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

8.5 双精度浮点比较指令

双精度浮点比较指令的定义与它们的单精度版本相类似，但是它们作用在双精度操作数上。

31　　27	26　25	24　　20	19　　15	14　　12	11　　7	6　　　0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCMP	D	src2	src1	EQ/LT/LE	dest	OP-FP

8.6 双精度浮点分类指令

双精度浮点分类指令FCLASS.D的定义与它的单精度版本相类似，但是它们作用在双精度操作数上。

31　　27	26　25	24　　20	19　　15	14　　12	11　　7	6　　　0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCLASS	D	0	src	001	dest	OP-FP

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第9章 RV32/64G 指令集列表

RISC-V项目的目标之一，就是可被用作一个稳定的软件开发目标机。为了这个目的，我们定义了一个基本ISA（RV32I或RV64I）加上一些标准扩展（IMAFD）作为“通用”ISA，我们对这个IMAFD的指令集扩展组合，缩写为G。本章给出了RV32G和RV64G的操作码映射表和指令集列表。

Inst[4:2] Inst[6:5]	000	001	010	011	100	101	110	111 (>32b)
00	LOAD	LOAD-FP	***custom-0***	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	***custom-1***	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	***reserved***	***custom-2/rv128***	48b
11	BRANCH	JALR	***reserved***	JAL	SYSTEM	***reserved***	***custom-3/rv128***	≥80b

表 9.1：RISC-V 基本操作码映射表，inst[1:0]=11

表 9.1给出了RVG主要操作码的映射表。具有3个或者更多个最低位被置为1的主要操作码，保留给长度超过32位的指令。标记为***reserved***的操作码应当避免在定制的指令集扩展中使用，因为它们可能被未来的标准扩展使用。在标准32位指令格式中，标记为***custom-0***和***custom-1***的主要操作码会避免被未来的标准扩展使用，因此推荐给定制指令集扩展使用。标记为***custom-2/rv128***和***custom-3/rv128***的操作码，保留给未来的RV128使用，但是也会避免被标准扩展使用，因此也可以用于RV32和RV64的定制指令集扩展。

我们相信RV32G和RV64G为广阔的、各种各样的通用计算，提供了简洁但完整的指令集。第14章描述的可选压缩指令集可以加进来（构成RV32GC和RV64GC），以提高性能、减小代码大小、提高能耗效率，虽然也增加了一些硬件复杂性。

随着我们在未来的指令集扩展中超过了IMAFDC，新增的指令趋向于更加领域专用，并仅对某些类型的应用程序有益，例如对多媒体或者安全。与绝大多数商业ISA不同，RISC-V ISA设计清晰地分离了基本ISA、适用面很广的标准扩展、更为专用的增强。第10章将就向RISC-V ISA添加扩展的方法，有更加详细的讨论。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

31 25	24	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R类
imm[11:0]		rs1	funct3	rd	opcode	I类
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S类
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	SB类
imm[31:12]				rd	opcode	U类
imm[20|10:1|11|19:12]				rd	opcode	UJ类

RV32I 基本指令集

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 Pred	Succ	00000	000	00000	0001111	FENCE
0000 0000	0000	00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREA
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRW
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

31 25	24	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R类
imm[11:0]		rs1	funct3	rd	opcode	I类
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S类

RV64I基本指令集（除了RV32I之外）

imm[11:0]		rs1	110	rd	0000011	LWU
imm[11:0]		rs1	011	rd	0000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD
000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SRLI
010000	shamt	rs1	101	rd	0010011	SRAI
imm[11:0]		rs1	000	rd	0011011	ADDIW
0000000	shamt	rs1	001	rd	0011011	SLLIW
0000000	shamt	rs1	101	rd	0011011	SRLIW
0100000	shamt	rs1	101	rd	0011011	SRAIW
0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

RV32M标准扩展

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

RV64M标准扩展（除了RV32M之外）

0000001	rs2	rs1	000	rd	0111011	MULW
0000001	rs2	rs1	100	rd	0111011	DIVW
0000001	rs2	rs1	101	rd	0111011	DIVUW
0000001	rs2	rs1	110	rd	0111011	REMW
0000001	rs2	rs1	111	rd	0111011	REMUW

RV32A标准扩展

00010	a	rl	00000	rs1	010	rd	0101111	LR.W
00011	a	rl	rs2	rs1	010	rd	0101111	SC.W
00001	a	rl	rs2	rs1	010	rd	0101111	AMOSWAP.W
00000	a	rl	rs2	rs1	010	rd	0101111	AMOADD.W
00100	a	rl	rs2	rs1	010	rd	0101111	AMOXOR.W
01100	a	rl	rs2	rs1	010	rd	0101111	AMOAND.W
01000	a	rl	rs2	rs1	010	rd	0101111	AMOOR.W
10000	a	rl	rs2	rs1	010	rd	0101111	AMOMIN.W
10100	a	rl	rs2	rs1	010	rd	0101111	AMOMAX.W
11000	a	rl	rs2	rs1	010	rd	0101111	AMOMINU.W
11100	a	rl	rs2	rs1	010	rd	0101111	AMOMAXU.W

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

31	25	24	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R类
rs3	funct2	rs2	rs1	funct3	rd	opcode	R4类
imm[11:0]			rs1	funct3	rd	opcode	I类
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S类

RV64A标准扩展（除了RV32A之外）

00010	a	rl	00000	rs1	011	rd	0101111	LR.D
00011	a	rl	rs2	rs1	011	rd	0101111	SC.D
00001	a	rl	rs2	rs1	011	rd	0101111	AMOSWAP.D
00000	a	rl	rs2	rs1	011	rd	0101111	AMOADD.D
00100	a	rl	rs2	rs1	011	rd	0101111	AMOXOR.D
01100	a	rl	rs2	rs1	011	rd	0101111	AMOAND.D
01000	a	rl	rs2	rs1	011	rd	0101111	AMOOR.D
10000	a	rl	rs2	rs1	011	rd	0101111	AMOMIN.D
10100	a	rl	rs2	rs1	011	rd	0101111	AMOMAX.D
11000	a	rl	rs2	rs1	011	rd	0101111	AMOMINU.D
11100	a	rl	rs2	rs1	011	rd	0101111	AMOMAXU.D

RV32F标准扩展

imm[11:0]			rs1	010	rd	0000111	FLW
imm[11:5]		rs2	rs1	010	imm[4:0]	0100111	FSW
rs3	00	rs2	rs1	rm	rd	1000011	FMADD.S
rs3	00	rs2	rs1	rm	rd	1000111	FMSUB.S
rs3	00	rs2	rs1	rm	rd	1001011	FNMSUB.S
rs3	00	rs2	rs1	rm	rd	1001111	FNMADD.S
0000000		rs2	rs1	rm	rd	1010011	FADD.S
0000100		rs2	rs1	rm	rd	1010011	FSUB.S
0001000		rs2	rs1	rm	rd	1010011	FMUL.S
0001100		rs2	rs1	rm	rd	1010011	FDIV.S
0101100		00000	rs1	rm	rd	1010011	FSQRT.S
0010000		rs2	rs1	000	rd	1010011	FSGNJ.S
0010000		rs2	rs1	001	rd	1010011	FSGNJN.S
0010000		rs2	rs1	010	rd	1010011	FSGNJX.S
0010100		rs2	rs1	000	rd	1010011	FMIN.S
0010100		rs2	rs1	001	rd	1010011	FMAX.S
1100000		00000	rs1	rm	rd	1010011	FCVT.W.S
1100000		00001	rs1	rm	rd	1010011	FCVT.WU.S
1110000		00000	rs1	000	rd	1010011	FMV.X.S
1010000		rs2	rs1	010	rd	1010011	FEQ.S
1010000		rs2	rs1	001	rd	1010011	FLT.S
1010000		rs2	rs1	000	rd	1010011	FLE.S
1110000		00000	rs1	001	rd	1010011	FCLASS.S
1101000		00000	rs1	rm	rd	1010011	FCVT.S.W
1101000		00001	rs1	rm	rd	1010011	FCVT.S.WU
1111000		00000	rs1	000	rd	1010011	FMV.S.X

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

31	25	24	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R类
rs3	funct2	rs2	rs1	funct3	rd	opcode	R4类
imm[11:0]			rs1	funct3	rd	opcode	I类
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S类

RV64F 标准扩展（除了 RV32F 之外）

1100000	00010	rs1	rm	rd	1010011	FCVT.L.S
1100000	00011	rs1	rm	rd	1010011	FCVT.LU.S
1101000	00010	rs1	rm	rd	1010011	FCVT.S.L
1101000	00011	rs1	rm	rd	1010011	FCVT.S.LU

RV32D 标准扩展

imm[11:0]			rs1	011	rd	0000111	FLD
imm[11:5]		rs2	rs1	011	imm[4:0]	0100111	FSD
rs3	00	rs2	rs1	rm	rd	1000011	FMADD.D
rs3	00	rs2	rs1	rm	rd	1000111	FMSUB.D
rs3	00	rs2	rs1	rm	rd	1001011	FNMSUB.D
rs3	00	rs2	rs1	rm	rd	1001111	FNMADD.D
0000001		rs2	rs1	rm	rd	1010011	FADD.D
0000101		rs2	rs1	rm	rd	1010011	FSUB.D
0001001		rs2	rs1	rm	rd	1010011	FMUL.D
0001101		rs2	rs1	rm	rd	1010011	FDIV.D
0101101		00000	rs1	rm	rd	1010011	FSQRT.D
0010001		rs2	rs1	000	rd	1010011	FSGNJ.D
0010001		rs2	rs1	001	rd	1010011	FSGNJN.D
0010001		rs2	rs1	010	rd	1010011	FSGNJX.D
0010101		rs2	rs1	000	rd	1010011	FMIN.D
0010101		rs2	rs1	001	rd	1010011	FMAX.D
0100000		00001	rs1	rm	rd	1010011	FCVT.S.D
0100001		00000	rs1	rm	rd	1010011	FCVT.D.S
1010001		rs2	rs1	010	rd	1010011	FEQ.D
1010001		rs2	rs1	001	rd	1010011	FLT.D
1010001		rs2	rs1	000	rd	1010011	FLE.D
1110001		00000	rs1	001	rd	1010011	FCLASS.D
1100001		00000	rs1	rm	rd	1010011	FCVT.W.D
1100001		00001	rs1	rm	rd	1010011	FCVT.WU.D
1101001		00000	rs1	rm	rd	1010011	FCVT.D.W
1101001		00001	rs1	rm	rd	1010011	FCVT.D.WU

RV64D 标准扩展（除了 RV32D 之外）

1100001	00010	rs1	rm	rd	1010011	FCVT.L.D
1100001	00011	rs1	rm	rd	1010011	FCVT.LU.D
1110001	00000	rs1	000	rd	1010011	FMV.X.D
1101001	00010	rs1	rm	rd	1010011	FCVT.D.L
1101001	00011	rs1	rm	rd	1010011	FCVT.D.LU
1111001	00000	rs1	000	rd	1010011	FMV.D.X

表 9.2：RISC-V 指令列表

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第10章　扩展 RISC-V

除了支持标准通用软件开发之外，RISC-V的另外一个目标就是，给更专门的指令集扩展或者更定制化的加速器，提供一个基础。指令编码空间和可选的变长指令集编码被设计成使得当构建更加定制化的处理器时，标准ISA工具链更容易改进。例如，专注于为只使用标准I基本内核的实现提供完整持续的软件支持，可能包含许多非标准的指令集扩展。

本章描述了扩展RISC-V ISA的多种方法，以及由不同小组研发指令集扩展的管理模式。本卷内容仅仅处理用户级ISA，虽然在第二卷管理员级扩展中也使用相同的方法和术语。

10.1 扩展术语

这一部分描述了描述RISC-V扩展的一些标准术语。

标准 vs. 非标准扩展

任何RISC-V处理器实现都必须支持一个基本整数ISA（RV32I或者RV64I）。另外，一个实现可以支持一个或者多个扩展。我们将扩展分为两大类：***标准的***、***非标准的***。

- 一个标准扩展是一个通用的扩展，并不与其它任何标准扩展冲突。当前，在本手册其它章节描述的“MAFDQLCBTPV”扩展，是已经完成或者正在计划的标准扩展。
- 一个非标准扩展是一个高度特殊化的扩展，可能与其它标准或者非标准扩展冲突。我们预测随着时间的推移，将研发出大量非标准扩展，可能最后有些非标准扩展将发展成为标准扩展。

指令编码空间和前缀

一个指令编码空间是一些指令的位，基本ISA或者ISA扩展就编码在这些位中。RISC-V支持各种指令长度，但是即使对于一个单一的指令长度，也有各种大小的编码空间可供使用。例如，基本ISA被定义为一个30位的编码空间（32位指令的31-2位），而原子性扩展“A”落在一个25位编码空间中（31-7位）。

我们使用术语***前缀（prefix）***指明一个指令编码空间的***右边***的位（由于RISC-V是小端的，右边的位保存在存储器地址较小的地方，因此在按顺序取指时，构成了前缀）。标准基本ISA编码的前缀是2位“11”，它处在32位字的1-0位。而标准原子扩展“A”的前缀是7位“0101111”，它处在32位字的6-0位，代表了AMO主要操作码（major opcode）。编码格式有一个奇怪的事情是，在32位指令格式中，用于编码次要操作码（minor opcode）的3位funct3字段，与主要操作码并不连续，但是仍被认为是22位指令空间的前缀的一部分。

虽然一个指令编码空间可以是任意大小，但是采用一少部分通用的大小，可以简化将各种独立研发的扩展融入一个整体的编码中。表 10.1给出了RISC-V推荐的指令编码空间大小。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

大小	使用	标准指令长度中可用数目			
		16位	32位	48位	64位
14位	压缩16位编码的四分之一	3			
22位	基本32位编码中的次要操作码		2^8	2^{20}	2^{35}
25位	基本32位编码中的主要操作码		32	2^{17}	2^{32}
30位	基本32位编码的四分之一		1	2^{12}	2^{27}
32位	48位编码中的次要操作码			2^{10}	2^{25}
37位	48位编码中的主要操作码			32	2^{20}
40位	48位编码的四分之一			4	2^{17}
45位	64位编码中的亚次要操作码				2^{12}
48位	64位编码中的次要操作码				2^9
52位	64位编码中的主要操作码				32

表 10.1：建议的标准 RISC-V 指令编码空间大小

绿地扩展 vs. 棕地扩展

我们使用术语“***绿地扩展（greenfield extension）***”来描述一个扩展开始使用一个新的指令编码空间，因此仅可能在前缀级别产生编码冲突。我们使用术语“***棕地扩展（brownfield extension）***”来描述一个扩展嵌入到一个前面已经定义指令编码空间。一个棕地扩展必须挂靠在某个特定的绿地“父母”编码下，而同一个绿地“父母”编码下，可能有多个棕地扩展（**译者注：就像一个儿子，必须有一对父母；而一对父母，可能有多个孩子一样**）。例如，基本ISA是一个30位指令空间的绿地编码，而FDQ浮点扩展，都是基本ISA 30位编码空间下的棕地扩展。

注意到我们认为标准A扩展是一个绿地扩展，因为它在整个32位基本指令空间中，定义了一个新的、以前空白的、最左边25位的编码空间，即使它的标准前缀落在基本ISA的30位编码空间中。只改变它的单个7位前缀，就可以将A扩展移动到一个不同的30位编码空间，而仅仅需要考虑在前缀级别的冲突，而不需要考虑编码空间内部的冲突。

	增加状态	没有新状态
绿地	RV32I(30), RV64I(30)	A(25)
棕地	F(I), D(F), Q(D)	M(I)

表 10.2：标准指令集扩展的二维特性

表 10.2以简单的二维表格方式给出了基本和标准扩展。一个坐标是表明扩展是绿地扩展还是棕地扩展，另一个坐标表明扩展是否增加了体系结构状态。对于绿地扩展，指令编码空间的大小在括号内给出。对于棕地扩展，括号内给出了这个棕地扩展基于的（绿地的或者棕地的）扩展名字。额外的用户级体系结构状态通常意味着管理员级系统的改变或者标准调用约定的改变。

注意到RV64I不被认为是RV32I的一个扩展，而是一个完全不同的基本编码。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

标准兼容的全局编码

一个真正的RISC-V实现的ISA完整或者*全局*编码（complete or ***global*** encoding），必须对其包含的每一个指令编码空间分配一个唯一的、不冲突的前缀。基本内核和每一个标准扩展都有一个已经分配好的标准前缀，可以确保它们可以在一个全局编码中共存。

一个***标准兼容的***全局编码，是一个其基本内核和每一个包括在内的标准扩展都是标准前缀的全局编码。一个标准兼容的全局编码可以包含非标准扩展，它们和所包含的标准扩展不冲突。一个标准兼容的全局编码可以为非标准扩展使用标准前缀，如果相对应的标准扩展并没有包括在这个全局编码内的话。换句话说，在一个标准兼容的全局编码中，一个标准扩展必须使用它的标准前缀，但如果没有包含某个标准扩展，那么这个标准扩展的标准前缀，是可以挪作他用的。这些约束，可以使得一个通用工具链可以以任何RISC-V标准兼容全局编码的子集的实现作为目标机。

保证的非标准编码空间

为了支持私有定制扩展的研发，部分编码空间被确保永远不会被标准扩展使用。

10.2 RISC-V 扩展设计理念

我们试图支持大量的、不相关的、研发出来的扩展，通过鼓励扩展的设计者们，在指令编码空间内工作，提供工具通过分配唯一的前缀，将这些工作打包进一个标准兼容的全局编码中。某些扩展可更自然地实现为现有扩展的棕地扩展，可共享已分配给它们“父母”绿地扩展的任何前缀。标准扩展前缀避免在编码核心功能时出现兼容问题，而允许定制更多私有的扩展。

这个将RISC-V扩展重新打包进不同的标准兼容全局编码的能力，有几个作用。

一种应用情形是研发了高度特殊的定制加速器，设计用于运行来自重要领域的核心应用。这时可能想把除了基本整数ISA之外的其他东西都去掉，而只加入任务所需要的扩展。基本ISA被设计为最小的硬件需求，并被编码为只使用很少一部分32位指令编码空间。

另一种应用情形是为一种新的指令集扩展构建一个研究用的原型。研究者可能并不想花时间去实现一个变长指令取指单元，于是他们就可以使用一个简单的32位固定长度指令编码来构建原型的扩展。然而，这个新的扩展可能太大了，不能在32位空间里与标准扩展共存。如果研究实验并不需要所有的标准扩展，那么一个标准兼容的全局编码可以丢弃无用的标准扩展，在非标准的位置上，将这些标准扩展的前缀用于所设计的扩展上，从而减轻研究原型的工程量。标准工具仍然可以工作在基本内核和现存的任何标准扩展上，减少了开发时间。一旦对指令集扩展的评估和调优完成，它就可以打包进一个更大的变长编码空间，以避免和任何标准扩展产生冲突。

下面的章节将描述逐步复杂的、开发新指令扩展的策略。这些大部分试图用于高度定制、教学性质、或者实验性体系结构，而不是RISC-V ISA研发的主流。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

10.3 在固定 32 位指令格式内的扩展

在这一节里，我们将讨论在只支持基本固定32位指令格式的实现内增加扩展。

我们预测这个固定 32 位编码将在许多受限的加速器和研究原型中广为流行。

可用的 30 位指令编码空间

在标准编码中，3个可用的30位指令编码空间（以00、01、10为前缀）可用于可选的压缩指令扩展。然而，如果不需要压缩指令集扩展，那么这3个30位编码空间就可以自由使用。这将在32位格式中扩大了4倍的可用编码空间。

可用的 25 位指令编码空间

在基本和标准扩展编码中，一个25位指令编码空间对应于一个主要操作码。有4个主要操作码特意保留给定制扩展（表 9.1），每一个代表了一个25位编码空间。其中两个保留给RV128基本编码的最终使用（将是OP-IMM-64和OP-64），但是可被用于RV32和RV64的标准或者非标准扩展。

保留给RV64的两个操作码（OP-IMM-32和OP-32）也可仅用于RV32的标准或者非标准扩展。

如果一个实现不需要浮点，则保留给标准浮点扩展的7个主要操作码（LOAD-FP、STORE-FP、MADD、MSUB、NMSUB、NMADD、OP-FP）可以被非标准扩展重用。类似的，AMO主要操作码在不需要标准原子性扩展时，也可以被重用。

如果一个实现不需要超过32位长度的指令，则有额外的4个主要操作码可供使用（表 9.1 中灰色部分）。

基本RV32I只使用了11个主要操作码和3个保留操作码，给扩展留下18个可用的操作码。基本RV64I只使用了13个主要操作码和3个保留操作码，给扩展留下16个可用的操作码。

可用的 22 位指令编码空间

在基本和标准扩展编码中，一个22位指令编码空间对应于一个funct3次要操作码。好几个主要操作码拥有没被完全占用的funct3次要操作码，留下了几个可用的22位编码空间。通常一个主要操作码使用这种格式在指令剩余的位中编码操作数，理想情况下，一个扩展应当遵守这种主要操作码的操作数格式，以简化硬件译码。

其他空间

在某些主要操作码下还有更小的空间可用，并且并不是所有的次要操作码都被完全使用了的。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

10.4 增加对齐的 64 位指令扩展

当扩展太大，不能在32位固定长度指令格式中容纳时，最简单的方法就是加入自然对齐的64位指令。实现仍然必须支持32位基本指令格式，但可以要求64位指令都在64位边界对齐，以简化指令取指，可在需要时，使用32位NOP指令来作为对齐填充。

为了简化标准工具的使用，64位指令应当按照图 1.1中描述那样进行编码。然而，实现可以为64位指令选择一个非标准的指令长度编码，同时保持为32位指令的标准编码。例如，如果不需要压缩指令，那么一条64位指令可编码为在它最前面2位设置1位或者更多位0。

我们预测出现处理器生成器，能够产生自动处理任何支持的变长指令编码组合的取指单元。

10.5 支持 VLIW 编码

虽然RISC-V并不是设计为一个纯的VLIW机器（**译者注：Very Long Instruction Word，超长指令字**）的基础，但是通过使用几种方法，可以将VLIW编码作为扩展加入。在任何情况下，必须支持基本32位编码，以便能够使用任何标准的软件工具。

定长指令组

最简单的方法就是定义一个单一而巨大的自然对齐指令格式（例如128位），VLIW操作可以编码在里面。在传统的VLIW中，这种方法可能导致浪费指令空间来保存NOP（**译者注：VLIW的一个主要问题是，编译器可能无法同时调度大量运算来填充这么宽的指令，导致指令中可能会出现大量的“空隙”**），但是一个RISC-V兼容的实现可以支持基本的32位指令，将VLIW代码大小扩展限制在VLIW加速函数上。

编码长度指令组

另外一种方法就是使用图 1.1中的标准长度编码来编码并行指令组，允许将NOP压缩到VLIW指令之外。例如，一条64位指令可以包含两个28位操作，而一条96位指令可以包含三个28位操作，以此类推。另外，一条48位指令可以包含一个42位操作，而一条96位指令可以包含两个48位操作，以此类推。

此方法的优点在于对包含单一操作的指令，保持了基本ISA编码。缺点在于在VLIW指令中的操作需要新的28位或者42位编码，对更大指令组的非对齐取指。一种简化就是不允许VLIW指令跨过某些微体系结构重要的边界（例如cache line或者虚拟存储器页）。

固定大小指令束

另外一种方法，与Itanium类似，使用较大的、自然对齐的、固定大小指令束（例如128位），并行操作分组编码其中。这简化了指令取指，但是把复杂性转交给了分组执行引擎。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

为了保证RSIC-V兼容，还必须支持基本32位指令。

前缀中“组结束位”

上述所有方法，都不能对在一条VLIW指令中的单独操作使用RISC-V编码。然而有另外一种方法对固定32位编码中的2位前缀赋予新的用途。一位前缀当置位1时，用于表示“组结束（end-of-group）”，而第2位如果被清零则表明在谓词下执行。由工具生成的标准RISC-V 32位指令由于两位前缀都被置为（11），因此不会被VLIW扩展执行，因此有正确的语义，每一条在一个组最后的指令，并且不是谓词。

这种方法的主要缺点是，基本的ISA缺少一个激进VLIW系统通常所需要的复杂谓词，而且在标准30位编码空间中加入更多的谓词寄存器很困难。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第11章　ISA 子集命名约定

本章描述RISC-V ISA子集命名体系，用于简明地描述在硬件实现中提供的指令集合，或者由应用程序二进制接口（ABI）使用的指令集合。

RISC-V ISA 被设计成能够支持各种实验性指令集扩展的各种实现。我们发现一个有组织的命名体系，可以简化软件工具和文档。

11.1 大小写敏感

ISA命名字符串是大小写不敏感的。

11.2 基本整数 ISA

RISC-V ISA字符串开始于RV32I、RV32E、RV64I或者RV128I，指明了基本整数ISA所支持的地址空间大小。

11.3 指令扩展名字

标准ISA扩展是由单个字母构成的名字。例如，基本整数核心的最前面4个标准扩展是："M"表示整数乘法和除法，"A"表示原子性存储器指令，"F"表示单精度浮点指令，"D"表示双精度浮点指令。任何RISC-V指令集变种，都可以简洁地通过将基本整数前缀和所包含的扩展连接起来描述。例如，"RV64IMAFD"。

我们也定义了一个缩写"G"来表示"IMAFD"基本内核和扩展，这代表了我们标准通用ISA。

RISC-V ISA标准扩展使用了其他的保留字母，例如"Q"表示四精度浮点，"C"表示16位压缩指令格式。

11.4 版本号

认识到随着时间推移，指令集可能扩展或者更改，我们在子集名字之后编码了子集版本号。版本号分为主要版本号和次要版本号，中间以"p"分割。如果次要版本号是"0"，则"p0"可以从版本字符串中省略。主要版本号的变化，暗示着向后兼容性的丢失，然而次要版本号的变化则必须保证向后兼容性。例如，本手册1.0版本定义的原来的64位标准ISA，可以写成"RV64I1p0M1p0A1p0F1p0D1p0"，简洁的是"RV64I1M1A1F1D1"，更简洁的是"RV64G1"。G ISA子集可被写成"RV64I2p0M2p0A2p0F2p0D2p0"，更简洁的是"RV64G2"。

我们在本手册的第二版中引入了版本号体系，我们想把它作为一个永久的标准。因此，我们定义标准子集的缺省版本为，"RV32G2"，这等价于"RV32I2M2A2F2D2"。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

11.5 非标准扩展命名

非标准子集被命名为使用一个“X”，后面跟在一个以字母开始的字符串和可选的版本号。例如，“Xhwacha”命名了Hwacha向量取指ISA扩展；“Xhwacha2”和“Xhwacha2p0”命名了同样机器的2.0版。

非标准扩展必须使用一个下划线与其它多字母扩展相分隔。例如，一个具有非标准扩展Argle和Bargle的ISA，可以命名为“RV64GXargle_Xbargle”。

11.6 管理员级指令子集

标准管理员指令子集在第二卷中定义，但使用“S”作为前缀，后面跟着管理员子集名字和一个可选的版本号。

管理员扩展必须使用一个下划线与其它多字母扩展相分隔。

11.7 管理员级扩展

管理员级ISA的非标准扩展，使用“SX”前缀。

11.8 子集命名约定

表 11.1总结了标准化的子集命名。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

子集	名字
标准通用ISA	
整数 整数乘法和除法 原子性 单精度浮点 双精度浮点	I M A F D
通用	G=IMAFD
标准用户级扩展	
四精度浮点 十进制浮点 16位压缩指令 位操作 事务存储器 打包的SIMD扩展 向量扩展	Q L C B T P V
非标准用户级扩展	
非标准扩展“abc”	Xabc
标准管理员级ISA	
管理员扩展“def”	Sdef
非标准管理员级扩展	
管理员扩展“ghi”	SXghi

表 11.1：标准 ISA 子集命名。该表也定义了在名字字符串中，子集名字出现的顺序，在表格中，自上而下，那么在名字中，就是自开始到最后，例如 RV32IMAFDQC 是合法的，而 RV32IMAFDCQ 是非法的。

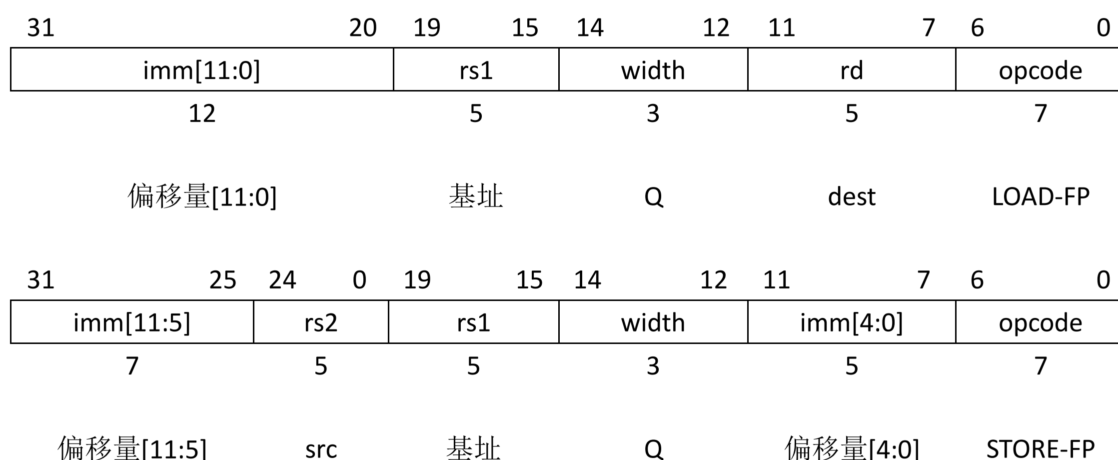
Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第12章　四精度浮点的“Q”标准扩展，版本 2.0

本章描述符合IEEE 754-2008算术标准的128位二进制浮点指令的Q标准扩展。128位或者四精度二进制浮点指令子集被命名为“Q”，需要RV64IFD。现在浮点寄存器被扩展了，可以保存一个单精度或者一个双精度或者一个四精度的浮点值（FLEN=128）。

12.1　四精度 load 和 store 指令

增加了LOAD-FP指令和STORE-FP指令的128位新变种，由一个funct3字段（**译者注：应是图中的*width*字段**）的新值进行编码。



如果一个浮点寄存器保存了一个单精度或者双精度浮点值，必须确保使用FSQ指令将寄存器的值保存到存储器之后，再使用FLQ指令重新读取该值，那么将在寄存器中重新生成原始的单精度或者双精度浮点值。除了这个特性之外，保存在存储器中的数据格式并没有明确定义。

FLQ指令和FSQ指令只有在有效地址是自然对齐且XLEN＝128时，才能保证执行的原子性。

12.2　四精度计算指令

对于大多数指令，一个新的支持格式被加入到格式字段，如表 12.1所示。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

fmt字段	助记符	含义
00	S	32位单精度
01	D	64位双精度
10	—	保留
11	Q	128位四精度

表 12.1：格式字段编码

四精度浮点计算指令的定义与它们的双精度版本相类似，但是它们作用在四精度操作数上，并产生四精度结果。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	Q	src2	src1	RM	dest	OP-FP
FMUL/FDIV	Q	src2	src1	RM	dest	OP-FP
FMIN-FMAX	Q	src2	src1	MIN/MAX	dest	OP-FP
FSQRT	Q	0	src	RM	dest	OP-FP

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
src3	Q	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

12.3 四精度转换和传输指令

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.int.fmt	Q	W[U]/L[U]	src	RM	dest	OP-FP
FCVT.fmt.int	Q	W[U]/L[U]	src	RM	dest	OP-FP

加入了新的浮点到浮点转换指令FCVT.S.Q、FCVT.Q.S、FCVT.D.Q、FCVT.Q.D。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.fmt.fmt	S	Q	src	RM	dest	OP-FP
FCVT.fmt.fmt	Q	S	src	RM	dest	OP-FP
FCVT.fmt.fmt	D	Q	src	RM	dest	OP-FP
FCVT.fmt.fmt	Q	D	src	RM	dest	OP-FP

浮点到浮点符号注入指令FSGNJ.Q、FSGNJN.Q、FSGNJX.Q与双精度符号注入指令相类似。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	Q	src2	src1	J[N]/JX	dest	OP-FP

FMV.X.Q和FMV.Q.X指令并没有提供，因此四精度位传输到整数寄存器，必须经过存储器。

RV128 在 Q 扩展中支持 FMV.X.Q 和 FMV.Q.X 指令。

12.4 四精度浮点比较指令

浮点比较指令在浮点寄存器***rs1***和***rs2***之间执行指定的比较（等于、小于、小于等于），并将结果的布尔值写入整数寄存器***rd***中。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCMP	Q	src2	src1	EQ/LT/LE	dest	OP-FP

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

12.5 四精度浮点分类指令

四精度浮点分类指令FCLASS.Q的定义与它们的双精度版本相类似，但是它作用在四精度操作数上。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCLASS	Q	0	src	001	dest	OP-FP

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第13章 十进制浮点的“L”标准扩展，版本 0.0

本章是标准扩展“L”规范的预留位置，这个扩展支持IEEE 754-2008标准中定义的十进制浮点算术。

13.1 十进制浮点寄存器

现有的浮点寄存器被用来保存64位和128位十进制浮点值，现有浮点load和store指令用来在寄存器和存储器之间传输数值。

由于融合乘加指令需要大的操作码空间，因此十进制浮点指令扩展在 30 位编码空间中将需要 5 个 25 位主要操作码。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第14章　压缩指令的“C”标准扩展，版本 1.9

本章描述描述RISC-V标准压缩指令集扩展的当前初稿，标准压缩指令集扩展，被命名为“C”，通过对常用操作加入短的16位指令编码，减少了静态和动态代码大小。这个“C”扩展可以添加到任何基本的ISA上（RV32、RV64、RV128），我们使用术语RVC来指明这种情形。典型的，程序中大约50%~60%的RISC-V指令可以被RVC指令代替，导致大约25%~30%代码大小的减少。

我们相信这个初稿将和最终的 RV32C 和 RV64C 设计相接近（看起来现在形成 RV128C 并不成熟），但我们仍然需要一轮或者多轮的评价，因此定名为 1.9 版本。请将您的评价发送到 isa-dev@lists.riscv.org 的邮件列表的 isa-dev。

14.1 概述

RVC 使用一种简单的压缩方案，以便在下列情形时，提供更短的 16 位版本的 32 位 RISC-V 指令：

- 立即数或者地址偏移量较小时
- 其中一个寄存器是零寄存器（**x0**）、ABI 链接寄存器（**x1**）或者 ABI 栈寄存器（**x2**）
- 目标寄存器和第一个源寄存器相同
- 最常见情况下使用了 8 个寄存器

C 扩展与其它所有标准扩展兼容。C 扩展允许 16 位指令可以自由地和 32 位指令混合执行，并运行 32 位指令可以在任何 16 位边界开始。（***译者注：一般情况下，32 位指令必须“天然地”对齐到 32 位存储器地址边界上，否则会导致非对齐存储器访问异常。***）

在原来的 32 位指令上面去掉 32 位对齐约束要求，可以大幅度提高代码密度。

压缩指令编码在绝大多数情况下，在 RV32C、RV64C 和 RV128C 下都是一样的，但如表 14.3 所示，少数操作码依据基本 ISA 的宽度有不同的用途。例如，更宽地址空间的 RV64C 和 RV128C 变种，需要额外的操作码来完成压缩 load 和 store 64 位整数值，而 RV32C 使用与单精度浮点值一样的操作码来进行压缩 load 和 store。类似的，RV128C 需要额外的操作码来完成压缩 load 和 store 128 位整数值，而在 RV32C 和 RV64C 中，它们使用与双精度浮点值一样的操作码来进行压缩 load 和 store。如果要实现标准 C 扩展，必须提供相应的压缩浮点 load 和 store 指令，而不管相关的标准浮点扩展（F 和/或 D 扩展）是否实现。另外，RV32C 包含一条压缩跳转和链接指令，以压缩短范围的子过程调用，同样的操作码被用于 RV64C 和 RV128C 的压缩 ADDIW 指令。

双精度 load 和 store 是静态和动态指令的重要部分，因此想到要把它们加

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

入到 RV32C 和 RV64C 编码中。

虽然对于当前支持的 ABI 编译出来的基准测试程序（benchmark）来说，单精度 load 和 store 并不十分重要，但是对于那些在硬件上仅支持单精度浮点单元、ABI 仅支持单精度浮点数的微控制器来说，在基准程序上，单精度 load 和 store 的使用频度至少和双精度 load 和 store 相同。因此，想到在 RV32C 中提供这些指令的压缩支持。

对于微控制器来说，短范围子过程调用在小的二进制代码中非常常见，因此想到在 RV32C 中加入它们。

虽然对不同的基本寄存器宽度、不同的用途，重用操作码会增加一定的文档复杂度，但是它对实现的影响非常小，即使是对同时支持多个基本 ISA 寄存器宽度也是如此。压缩浮点 load 和 store 指令使用了与更宽整数 load 和 store 相同的指令格式、相同的寄存器区分符。

RVC 是在这样的约束下设计的，即每一条 RVC 指令被扩展成在基本 ISA（RV32I/E、RV64I 或者 RV128I）或者 F、D 标准扩展中的一条 32 位指令。采用这条约束，有如下一些好处：

- 硬件设计可以在译码时简单地扩展 RVC 指令，简化了验证，并使得对现有微体系结构的改动最小化。
- 编译器可以不处理 RVC 扩展部分，留到汇编器和链接器来进行代码压缩，虽然一个压缩敏感的编译器通常可以生成更好代码。

我们感到通过在 C 和 IFD 指令之间进行简单的一对一映射，得到的多种复杂度减少，其远远超过通过增加一些仅仅支持 C 扩展的指令以获得稍微高一些的编码密度而带来的收益，也远高于允许将多条 IFD 指令编码入一条 C 指令而带来的收益。

值得重视的是，C 扩展并不是作为一个单独的 ISA 而被设计的，意味着它需要与一个基本 ISA 一块使用。

变长指令集已经被用来提高代码密度使用很长时间了。例如，在 1950 后期研发的 IBM Stretch[3]使用了一个具有 32 位和 64 位指令的 ISA，其中有些 32 位指令是 64 位指令的压缩版本。Stretch 也使用了这样一个概念，就是在一些较短的指令格式中，限制了可寻址的寄存器集合；具有短分支指令仅能引用索引寄存器中的一个。后来的 IBM 360 体系结构[2]支持一种简单的变长指令编码，支持 16 位、32 位或者 48 位指令格式。

在 1963 年，CDC 发布了 Cray 设计的 CDC 6600[23]，它是 RISC 的前辈，引入了一个大量寄存器的 load-store 体系结构，其指令长度 15 位和 30 位两种。后来的 Cray-1 设计使用了一种非常相似的指令格式，它采用了 16 位和 32 位指令长度。

自 1980 以来的早期 RISC ISA，都是选择性能而不是代码大小，这在工作站环境中是情理之中的，但对嵌入式系统并不合理。因此，ARM 和 MIPS 在后续的 ISA 版本中，都通过在标准 32 位指令集之外提供 16 位指令集，来达到更小的代码大小。压缩过后的 RISC ISA 相对于它们的完全版本，可以减少大约 25%~30%的代码大小，生成的代码远远小于 80x86 生成的代码。这个结果令一些人感到震惊，因为他们认为一个变长 CISC ISA 产生的代码应当比仅提供 16

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

位和 32 位格式的 RISC ISA 产生的代码要小。

由于原来的RISC ISA 并没有为这些计划外的压缩指令留有足够的操作码空间，因此它们被当做一个新的 ISA 进行开发。这意味着编译器需要为单独的压缩 ISA 使用不同的代码生成器。第一个压缩 RISC ISA 扩展（就是 ARM Thumb 和 MIPS16）仅仅使用了一种固定 16 位指令长度，这在静态代码大小上得到了很好的减少，但是导致动态指令数目的增加，这也导致了与原始固定 32 位长度指令相比，性能的下降。这导致了 16 位和 32 位指令长度混合的第二代压缩 RISC ISA（就是 ARM Thumb2、microMIPS、PowerPC VLE）的研发，于是可以获得与纯 32 位指令相似的性能，同时有巨大的代码大小减少。不幸的是，这些不同代次的压缩 ISA 彼此之间、与原始为压缩 ISA 之间，并不兼容，导致了文档、实现和软件工具上的巨大复杂性。

在通用 64 位 ISA 中，当前只有 PowerPC 和 microMIPS 支持压缩指令格式。令人感到惊讶的是，考虑到静态代码大小和动态取指带宽是重要的指标，当前移动平台上最为流行的 64 位 ISA（ARM v8）并没有包含一个压缩指令格式。虽然对于较大的系统来说，静态代码大小并不是一个主要关心的因素，但是在服务器运行商业负载（通常具有一个大的指令工作集）的时候，取指带宽可能会成为一个主要的瓶颈。

*得益于 25 年以来的经验，RISC-V 被设计成从外部支持压缩指令，为 RVC 保留了足够的操作码空间，使得它可以在基本 ISA 之上（与其它的标准扩展一起）作为一个简单的扩展加入。RVC 的理念在于为嵌入式应用程序减小代码大小，为所有应用程序提高性能和能耗效率，因为可获得更少的指令 cache 缺失（**译者注：指令长度缩小，可使得一个 Cache line 保存更多的指令，或者意味着在同样大小的指令 Cache 中容纳更多的指令**）。Waterman 指出 RVC 少取了 25%~30%的指令位，减少了 20~25%的指令 Cache 缺失，或者等效于指令 Cache 大小翻倍同样的性能[28]。*

14.2 压缩指令格式

表 14.1 给出了 8 种压缩指令格式。CR、CI 和 CSS 格式可以使用任何的 32 个 RVI 寄存器，但是 CIW、CL、CS 和 CB 被限制只能使用所有 32 个寄存器中的 8 个。表 14.2 给出了这些常用的寄存器，对应于 **x8** 到 **x15**。注意到有一个单独版本的 load 和 store 指令，将栈指针作为基地址寄存器使用，这是因为保存到栈和从栈恢复太常见了，并且它们使用 CI 和 CSS 格式，以便能够访问到所有 32 个数据寄存器。CIW 格式为 ADDI4SPN 指令提供了一个 8 位的立即数。

RISC-V 的 ABI 已经做了修改，将常用寄存器映射到寄存器 x8-x15。这通过提供一个连续的、自然对齐的寄存器编号，将简化解压缩的译码器，并且与 RV32E 子集基本规范相兼容，在那里只有 16 个整数寄存器。

压缩的、基于寄存器的浮点 load 和 store 指令也分别使用了 CL 和 CS 格式，将 8 个寄存器映射到 f8 到 f15。

标准 RISC-V 调用规范，将最常用的浮点寄存器映射到 f8-f15，这将允许与

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

整数寄存器编号相同的寄存器解压缩译码。

这些格式被设计成将两个源寄存器区分符放在所有指令中的同一个地方，因此可以去掉目的寄存器字段。如果存在完整的 5 位目的寄存器区分符，它与 32 位 RISC-V 编码中的位置是一样的。当立即数是符号扩展的时候，符号扩展总是从第 12 位开始。立即数字段被打乱了，就如同在基本规范中一样，以便减少用于立即数的多路选择器数量（immediate mux）。

指令格式中的立即数字段是被打乱的而不是按顺序存放的，这是为了保证在每一条指令中尽可能让尽量多的位处在相同的位置，这将简化实现。例如，立即数的位 17-10，总是来自指令的相同位位置。5 个其他的立即数位（5、4、3、2、1）只有两个指令位的来源，而 4 个其他的立即数位（9、7、6、2）有 3 个指令位的来源，而 1 个（第 8 位）有四个来源。

对于许多 RVC 指令来说，不允许 0 立即数，而且 **x0** 不是一个有效的 5 位寄存器区分符。这个限制，为其他需要较少操作数位的指令，释放了编码空间。

<table>
<tr><th>格式</th><th>含义</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>CR</td><td>寄存器</td><td colspan="4">funct4</td><td colspan="5">rd/rs1</td><td colspan="5">rs2</td><td colspan="2">op</td></tr>
<tr><td>CI</td><td>立即数</td><td colspan="3">funct3</td><td>imm</td><td colspan="5">rd/rs1</td><td colspan="5">imm</td><td colspan="2">op</td></tr>
<tr><td>CSS</td><td>栈相关 store</td><td colspan="3">funct3</td><td colspan="6">imm</td><td colspan="5">rs2</td><td colspan="2">op</td></tr>
<tr><td>CIW</td><td>宽立即数</td><td colspan="3">funct3</td><td colspan="8">imm</td><td colspan="3">rd’</td><td colspan="2">op</td></tr>
<tr><td>CL</td><td>Load</td><td colspan="3">funct3</td><td colspan="3">imm</td><td colspan="3">rs1’</td><td colspan="2">imm</td><td colspan="3">rd’</td><td colspan="2">op</td></tr>
<tr><td>CS</td><td>Store</td><td colspan="3">funct3</td><td colspan="3">imm</td><td colspan="3">rs1’</td><td colspan="2">imm</td><td colspan="3">rd’</td><td colspan="2">op</td></tr>
<tr><td>CB</td><td>分支</td><td colspan="3">funct3</td><td colspan="3">offset</td><td colspan="3">rs1’</td><td colspan="5">offset</td><td colspan="2">op</td></tr>
<tr><td>CJ</td><td>跳转</td><td colspan="3">funct3</td><td colspan="11">jump target</td><td colspan="2">op</td></tr>
</table>

表 14.1：压缩 16 位 RVC 指令格式

RVC 寄存器编号	000	001	010	011	100	101	110	111
整数寄存器编号	x8	x9	x10	x11	x12	x13	x14	x15
整数寄存器 ABI 名字	s0	s1	a0	a1	a2	a3	a4	a5
浮点寄存器编号	f8	f9	f10	f11	f12	f13	f14	f15
浮点寄存器 ABI 名字	fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

表 14.2：CIW、CL、CS 和 CB 格式中 rs1’、rs2’和 rd’字段三位指向的寄存器

14.3 Load 和 store 指令

为了增加 16 位指令能够访问的范围，使用零扩展立即数的数据传输指令，其数据值的大小被放大了多倍：对字×4、对双字×8、对四字×16。

RVC 提供了两种类型的 load 和 store。一种使用 ABI 栈指针 **x2** 作为基址寄存器，并可定位到任何数据寄存器。另外一种可以引用 8 个基址寄存器之一，并引用 8 个数据寄存器之一。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

基于栈指针的 load 和 store

15–13	12	11–7	6–2	1–0
funct3	imm	rd	imm	op
3	1	5	5	2
C.LWSP	偏移量[5]	dest≠0	偏移量[4:2|7:6]	C2
C.LDSP	偏移量[5]	dest≠0	偏移量[4:3|8:6]	C2
C.LQSP	偏移量[5]	dest≠0	偏移量[4|9:6]	C2
C.FLWSP	偏移量[5]	dest	偏移量[4:2|7:6]	C2
C.FLDSP	偏移量[5]	dest	偏移量[4:3|8:6]	C2

这些指令使用 CI 格式。

C.LWSP 指令将一个 32 位数值从存储器读入寄存器 ***rd*** 中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上栈指针 **x2** 形成的。它被扩展为 lw ***rd, offset[7:2](x2)***指令。

C.LDSP 是一条 RV64C/RV128C 仅有指令，它将一个 64 位数值从存储器读入寄存器 ***rd*** 中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上栈指针 **x2** 形成的。它被扩展为 ld ***rd, offset[8:3](x2)***指令。

C.LQSP 指令是一条 RV128C 仅有指令，它将一个 128 位数值从存储器读入寄存器 ***rd*** 中。其有效地址的计算是通过将零扩展的偏移量×16，然后加上栈指针 **x2** 形成的。它被扩展为 lq ***rd, offset[9:4](x2)***指令。

C.FLWSP 是一条 RV32FC 仅有指令，它将一个单精度浮点数值从存储器读入浮点寄存器 ***rd*** 中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上栈指针 **x2** 形成的。它被扩展为 flw ***rd, offset[7:2](x2)***指令。

C.FLDSP 是一条 RV32DC/RV64DC 仅有指令，它将一个双精度浮点数值从存储器读入浮点寄存器 ***rd*** 中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上栈指针 **x2** 形成的。它被扩展为 fld ***rd, offset[8:3](x2)***指令。

15–13	12–7	6–2	1–0
funct3	imm	rs2	op
3	6	5	2
C.SWSP	偏移量[5:2|7:6]	src	C2
C.SDSP	偏移量[5:3|8:6]	src	C2
C.SQSP	偏移量[5:4|9:6]	src	C2
C.FSWSP	偏移量[5:2|7:6]	src	C2
C.FSDSP	偏移量[5:3|8:6]	src	C2

这些指令使用 CSS 格式。

C.SWSP 指令将寄存器 ***rs2*** 中的 32 位值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上栈指针 **x2** 形成的。它被扩展为 sw ***rs2, offset[7:2](x2)***指令。

C.SDSP 是一条 RV64C/RV128C 仅有指令，它将寄存器 ***rs2*** 中的 64 位值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上栈指针 **x2** 形成的。它被扩展为

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

sd ***rs2, offset[8:3](x2)***指令。

C.SQSP 是一条 RV128C 仅有指令，它将寄存器 ***rs2*** 中的 128 位值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×16，然后加上栈指针 **x2** 形成的。它被扩展为 sq ***rs2, offset[9:4](x2)***指令。

C.FSWSP 是一条 RV32FC 仅有指令，它将浮点寄存器 ***rs2*** 中的单精度浮点数值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上栈指针 **x2** 形成的。它被扩展为 fsw ***rs2, offset[7:2](x2)***指令。

C.FSDSP 是一条 RV32DC/RV64DC 仅有指令，它将浮点寄存器 ***rs2*** 中的双精度浮点数值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上栈指针 **x2** 形成的。它被扩展为 fsd ***rs2, offset[8:3](x2)***指令。

基于寄存器的 Load 和 store

15 13	12 10	9 7	6 5	4 2	1 0
funct3	imm	rs1′	imm	rd′	op
3	3	3	2	3	2
C.LW	偏移量[5:3]	基址	偏移量[2|6]	dest	C0
C.LD	偏移量[5:3]	基址	偏移量[7:6]	dest	C0
C.LQ	偏移量[5|4|8]	基址	偏移量[7:6]	dest	C0
C.FLW	偏移量[5:3]	基址	偏移量[2|6]	dest	C0
C.FLD	偏移量[5:3]	基址	偏移量[7:6]	dest	C0

这些指令使用 CL 格式。

C.LW 指令将一个 32 位数值从存储器读入寄存器 ***rd’***中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上寄存器 ***rs1’***中的基址形成的。它被扩展为 lw ***rd’, offset[6:2](rs1’)***指令。

C.LD 是一条 RV64C/RV128C 仅有指令，它将一个 64 位数值从存储器读入寄存器 ***rd’***中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上寄存器 ***rs1’***中的基址形成的。它被扩展为 ld ***rd’, offset[7:3](rs1’)***指令。

C.LQ 是一条 RV128C 仅有指令，它将一个 128 位数值从存储器读入寄存器 ***rd’***中。其有效地址的计算是通过将零扩展的偏移量×16，然后加上寄存器 ***rs1’***中的基址形成的。它被扩展为 lq ***rd’, offset[8:4](rs1’)***指令。

C.FLW 是一条 RV32FC 仅有指令，它将一个单精度浮点数值从存储器读入浮点寄存器 ***rd’***中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上寄存器 ***rs1’***中的基址形成的。它被扩展为 flw ***rd’, offset[6:2](rs1’)***指令。

C.FLD 是一条 RV32DC/RV64DC 仅有指令，它将一个双精度浮点数值从存储器读入浮点寄存器 ***rd’***中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上寄存器 ***rs1’***中的基址形成的。它被扩展为 fld ***rd’, offset[7:3](rs1’)***指令。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

15	13	12	10	9	7	6	5	4	2	1	0
funct3		imm		rs1′		imm		rs2′		op	
3		3		3		2		3		2	

C.SW	偏移量[5:3]	基址	偏移量[2|6]	src	C0
C.SD	偏移量[5:3]	基址	偏移量[7:6]	src	C0
C.SQ	偏移量[5|4|8]	基址	偏移量[7:6]	src	C0
C.FSW	偏移量[5:3]	基址	偏移量[2|6]	src	C0
C.FSD	偏移量[5:3]	基址	偏移量[7:6]	src	C0

这些指令使用 CS 格式。

C.SW 指令将寄存器 ***rs2'***中的 32 位值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上寄存器 ***rs1'***中的基址形成的。它被扩展为 sw ***rs2', offset[6:2](rs1')*** 指令。

C.SD 是一条 RV64C/RV128C 仅有指令，它将寄存器 ***rs2'***中的 64 位值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上寄存器 ***rs1'***中的基址形成的。它被扩展为 sd ***rs2', offset[7:3](rs1')***指令。

C.SQ 是一条 RV128C 仅有指令，它将寄存器 ***rs2'***中的 128 位值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×16，然后加上寄存器 ***rs1'***中的基址形成的。它被扩展为 sq ***rs2', offset[8:4](rs1')***指令。

C.FSW 是一条 RV32FC 仅有指令，它将浮点寄存器 ***rs2'***中的单精度浮点数值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×4，然后加上寄存器 ***rs1'***中的基址形成的。它被扩展为 fsw ***rs2', offset[6:2](rs1')***指令。

C.FSD 是一条 RV32DC/RV64DC 仅有指令，它将浮点寄存器 ***rs2'***中的双精度浮点数值保存到存储器中。其有效地址的计算是通过将零扩展的偏移量×8，然后加上寄存器 ***rs1'***中的基址形成的。它被扩展为 fsd ***rs2', offset[7:3](rs1')***指令。

14.4 控制转移指令

RVC 提供了无条件跳转指令和条件分支指令。如同基本 RVI 指令一样，所有 RVC 控制转移指令的偏移量都是 2 字节的倍数。

15	13	12	2	1	0
funct3		imm		op	
3		11		2	

C.J	偏移量[11|4|9:8|10|6|7|3:1|5]	C1
C.JAL	偏移量[11|4|9:8|10|6|7|3:1|5]	C1

这些指令使用 CJ 格式。

C.J 指令执行一个无条件控制转移。偏移量被符号扩展后，与 **pc** 相加形成跳转目标地址。C.J 指令因此可以在±2KB 范围内进行跳转。C.J 指令被扩展为 jal ***x0, offset[11:1]***。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

C.JAL 指令是一条 RV32C 仅有指令，它执行与 C.J 指令相同的操作，但是它还将在跳转指令后的指令地址（**pc**+2）写入到链接寄存器 **x1** 中。C.JAL 指令被扩展为 jal ***x1, offset[11:1]***。

15–12	11–7	6–2	1–0
funct4	rs1	rs2	op
4	5	5	2
C.JR	src≠0	0	C2
C.JALR	src≠0	0	C2

这些指令使用 CR 格式。

C.JR（jump register）指令执行一个无条件控制转移到寄存器 ***rs1*** 的地址。C.JR 指令被扩展为 jalr ***x0, rs1, 0***。

C.JALR（jump and link register）指令执行与 C.JR 指令相同的操作，但是它还将在跳转指令后的指令地址（**pc**+2）写入到链接寄存器 **x1** 中。C.JALR 指令被扩展为 jalr ***x1, rs1, 0***。

严格来说，C.JALR 指令并没有被精确地扩展为基本 RVI 指令，因为那个被加到 ***pc*** *上以形成链接地址的值是 2，而不是基本 ISA 中的 4，但是同时支持偏移量 2 个字节和 4 个字节，只对微体系结构产生微小的影响。*

15–13	12–10	9–7	6–2	1–0
funct3	imm	rs1′	imm	op
3	3	3	5	2
C.BEQZ	偏移量[8|4:3]	src	偏移量[7:6|2:1|5]	C1
C.BNEZ	偏移量[8|4:3]	src	偏移量[7:6|2:1|5]	C1

这些指令使用 CB 格式。

C.BEQZ 指令执行条件控制转移。偏移量被符号扩展后，与 **pc** 相加形成跳转目标地址。C.BEQZ 指令因此可以在±256B 范围内进行跳转。如果寄存器 ***rs1'***的值是 0，则 C.BEQZ 指令产生控制转移（take the branch）。这条指令被扩展为 beq ***rs1', x0, offset[8:1]***。

C.BNEZ 指令定义相似，只是当寄存器 ***rs1'***的值是非 0 值，则指令产生控制转移（take the branch）。这条指令被扩展为 bne ***rs1', x0, offset[8:1]***。

14.5 整数计算指令

RVC 提供了一些用于整数算术和常数生成的指令。

整数常数-生成指令

两条常数-生成指令都使用 CI 格式，并且可以以任何整数寄存器为目标。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

15	13	12	11	7	6	2	1	0

funct3	imm	rd	imm[4:0]	op
3	1	5	5	2
C.LI	立即数[5]	dest≠0	立即数[4:0]	C1
C.LUI	非零立即数[17]	dest≠{0,2}	非零立即数[16:12]	C1

C.LI 指令将符号扩展的 6 位立即数 ***imm***，写入寄存器 ***rd*** 中。C.LI 指令仅在 ***rd≠x0*** 时才是有效的。C.LI 指令被扩展为 addi ***rd, x0, imm[5:0]***。

C.LUI 指令将非零的 6 位立即数写入到目标寄存器的 17-12 位，并将目标寄存器的低 12 位清零，然后将第 17 位符号扩展到整个目标寄存器的高位部分。C.LUI 寄存器仅在 ***rd≠{x0, x2}*** 且立即数不等于 0 时才是有效的。C.LUI 指令被扩展为 lui ***rd, nzimm[17:12]***。

整数寄存器-立即数指令

这些整数寄存器-立即数指令都使用 CI 格式，并在认为非 x0 整数寄存器和一个 6 位立即数之间进行操作。立即数不能为 0。

15	13	12	11	7	6	2	1	0

funct3	imm[5]	rd/rs1	imm[4:0]	op
3	1	5	5	2
C.ADDI	非零立即数[5]	dest	非零立即数[4:0]	C1
C.ADDIW	立即数[17]	dest≠0	立即数[4:0]	C1
C.ADDI16SP	非零立即数[9]	2	非零立即数[4|6|8:7|5]	C1

C.ADDI 指令将非零的、符号扩展的 6 位立即数加到寄存器 ***rd*** 的值上，将结果写入 ***rd***。C.ADDI 指令被扩展为 addi ***rd, rd, nzimm[5:0]***。

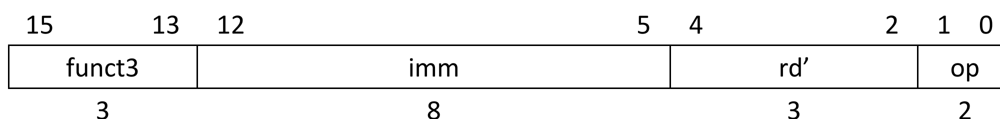
C.ADDIW 指令是一条 RV64C/RV128C 仅有的指令，它执行相同的计算，但是生成一个 32 位的结果，然后符号扩展结果到 64 位。C.ADDIW 指令被扩展为 addiw ***rd, rd, imm[5:0]***。对 C.ADDIW 指令而言，立即数可以是 0，这对应于 sext.w rd。

C.ADDI16SP 指令的操作码与 C.LUI 指令相同，但是使用 **x2** 作为目标寄存器。C.ADDI16SP 指令将一个非零的、符号扩展的 6 位立即数加到栈指针寄存器（**sp=x2**）上，此处立即数被放大 16 倍，其范围为（-512,496）。C.ADDI16SP 指令用于在过程的头部和尾部对栈指针进行调整。它被扩展为 addi ***x2, x2, nzimm[9:4]***。

---

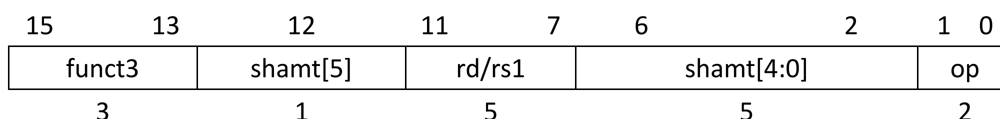
*在标准 RISC-V 调用约定中，栈指针 sp 总是 16 字节对齐的（**译者注：16 位对齐？**）。*

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.



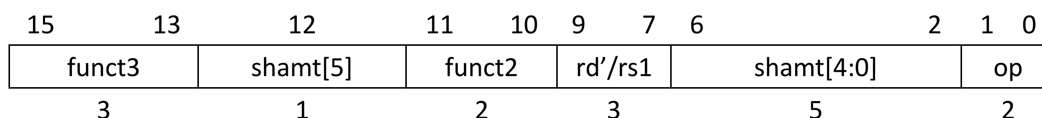
C.ADDI4SPN	非零立即数[5:4|9:6|2|3]	dest	C0

C.ADDI4SPN 指令是一条 CIW 格式的、RV32C/RV64C 仅有的指令，它将一个零扩展的、非零立即数，乘以 4，然后加到栈指针 **x2** 上，并将结果写入 ***rd’***。这条指令用于产生指向分配在栈中的变量的指针，它被扩展为 addi ***rd’, x2, zimm[9:2]***。



C.SLLI	移位次数[5]	dest≠0	移位次数[4:0]	C2

C.SLLI 指令是一条 CI 格式的指令，它对寄存器 rd 中的数值进行逻辑左移操作，并将结果写入 ***rd***。移位次数被编码到 ***shamt*** 字段，此处对 RV32C，***shamt[5]***必须为 0。对于 RV32C/RV64C，移位次数必须为非零值。对于 RV128C，一个 ***shamt*** 为 0，编码为移位 64 次。C.SLLI 指令被扩展为 slli ***rd, rd, shamt[5:0]***，除了对于 RV128C 且 ***shamt=0***，则被扩展为 slli ***rd, rd, 64***。



C.SRLI	移位次数[5]	C.SRLI	dest	移位次数[4:0]	C1
C.SRAI	移位次数[5]	C.SRAI	dest	移位次数[4:0]	C1

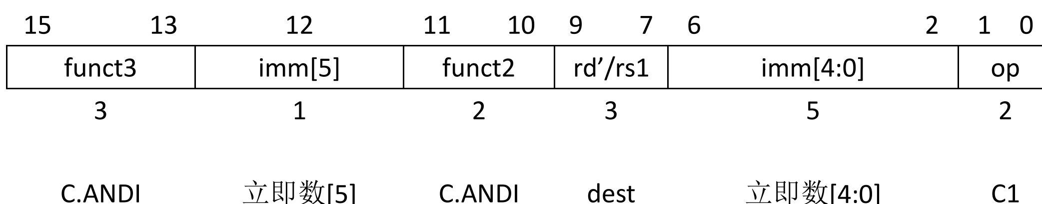
C.SRLI 指令是一条 CB 格式的指令，它对寄存器 ***rd’***中的数值进行逻辑右移操作，并将结果写入 ***rd’***。移位次数被编码到 ***shamt*** 字段，此处对 RV32C，***shamt[5]***必须为 0。对于 RV32C/RV64C，移位次数必须为非零值。对于 RV128C，一个 ***shamt*** 为 0，编码为移位 64 次。而且对于 RV128C，移位次数是符号扩展的，因此合法的移位次数是 1-31、64、96-127。C.SRLI 指令被扩展为 srli ***rd’, rd’, shamt[5:0]***，除了对于 RV128C 且 ***shamt=0***，则被扩展为 srli ***rd’, rd’, 64***。

C.SRAI 指令与 C.SRLI 指令相似，不过它执行一个算术右移操作。C.SRAI 指令被扩展为 srai ***rd’, rd’, shamt[5:0]***。

左移通常比右移更为常用，因为左移被频繁地用于对地址值进行放大操作。因此右移被分配了较小的编码空间，并且处于一个其他立即数都是符号扩展的编码区域中。对于 RV128，我们决定使得 6 位移位次数值也是符号扩展的。除了减少硬件译码复杂度之外，我们相信右移 96-127 次要比 64-95 更为有用，这可以用于从 128 位地址指针的高部分提取标签（tag）。我们注意到 RV128C

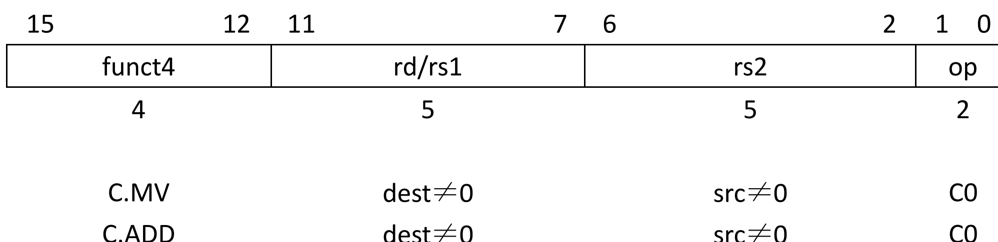
Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

并不像 RV32C 和 RV64C 那样已经确定下来，以允许对使用 128 位地址空间的典型代码进行评估。



C.ANDI 指令是一条 CB 格式的指令，它在寄存器 ***rd'***的值和一个符号扩展的 6 位立即数之间进行按位 AND 运算，并将结果写入到 ***rd'***中。C.ANDI 指令被扩展为 andi ***rd', rd', imm[5:0]***。

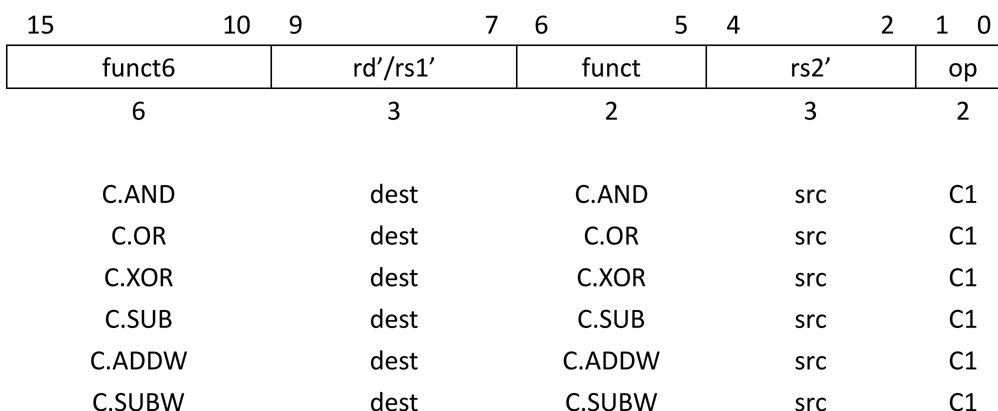
整数寄存器-寄存器指令



这些指令使用 CB 格式。

C.MV 指令将寄存器 ***rs2*** 的值复制到寄存器 ***rd*** 中。C.MV 指令被扩展为 add ***rd, x0, rs2***。

C.ADD 指令将寄存器 ***rd*** 的值与寄存器 ***rs2*** 的值相加，并将结果写入到寄存器 ***rd*** 中。C.ADD 指令被扩展为 add ***rd, rd, rs2***。



这些指令使用 CS 格式。

C.AND 指令在寄存器 ***rd'***和 ***rs2'***之间执行按位 AND 操作，并将结果写入寄存器 ***rd'***。C.AND 指令被扩展为 and ***rd', rd' rs2'***。

C.OR 指令在寄存器 ***rd'***和 ***rs2'***之间执行按位 OR 操作，并将结果写入寄存器 ***rd'***。C.OR 指

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

令被扩展为 or ***rd’, rd’ rs2’***。

C.XOR 指令在寄存器 ***rd’***和 ***rs2’***之间执行按位 XOR 操作，并将结果写入寄存器 ***rd’***。C.XOR 指令被扩展为 xor ***rd’, rd’ rs2’***。

C.SUB 指令将寄存器 ***rd’***的值减去 ***rs2’***的值，并将结果写入寄存器 ***rd’***。C.SUB 指令被扩展为 sub ***rd’, rd’ rs2’***。

C.ADDW 是一条 RV64C/RV128C 仅有的指令，它将寄存器 ***rd’***的值加上 ***rs2’***的值，将结果的低 32 位进行符号扩展，再写入 ***rd’***中。C.ADDW 指令被扩展为 addw ***rd’, rd’, rs2’***。

C.SUBW 是一条 RV64C/RV128C 仅有的指令，它将寄存器 ***rd’***的值减去 ***rs2’***的值，将结果的低 32 位进行符号扩展，再写入 ***rd’***中。C.SUBW 指令被扩展为 subw ***rd’, rd’, rs2’***。

这组的 6 条指令，每条指令并没有提供太多的好处，但是也没有占用很多的编码空间，并且实现起来也直截了当，作为一组指令是在静态和动态压缩中提供了一定的提高。

预定义非法指令

15–13	12	11–7	6–2	1–0
0	0	0	0	0
3	1	5	5	2
0	0	0	0	0

一条所有位都是 0 的 16 位指令，被永久的保留为一条非法指令。

*我们将全零指令保留为非法指令，以帮助捕获试图执行被零填充的或者不存在的存储器空间，产生自陷。全零值不应当被任何非标准扩展重新定义。类似的，我们保留全 1 指令（在 RISC-V 变长编码方式中，对应于非常长的指令）为非法指令，以捕获在不存在存储器区域的另外一种常见值。（**译者注：如果处理器试图访问不存在的存储器空间地址，外部硬件常规做法是总是返回全 0，或者返回全 1**）*

NOP 指令

15–13	12	11–7	6–2	1–0
funct3	imm[5]	rd/rs1	imm[4:0]	op
3	1	5	5	2
C.NOP	0	0	0	C1

C.NOP 指令是一条 CI 格式指令，它不改变任何用户可见状态，除了推进 **pc** 之外。C.NOP 指令被编码为 c.addi ***x0, 0*** 并且被扩展为 addi ***x0, x0, 0***。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

断点指令

15–12	11–2	1–0
funct4	0	op
4	10	2
C.EBREAK	0	C0

调试器可以使用 C.EBREAK 指令，它将被扩展为 ebreak 指令，并导致控制被转移回到调试环境。C.EBREAK 指令的操作码与 C.ADD 指令的操作码相同，但是其 ***rd*** 和 ***rs2*** 都是 0，因此也可以使用 CR 格式。

14.6 在 LR/SC 序列中使用 C 指令

在支持 C 扩展的实现上，当要确保最终成功时，可以在 LR/SC 序列中使用 I 类压缩格式指令，如 6.2 节所述。

这个含义就是任何声称同时支持 A 和 C 扩展的实现，必须确保包含有效 C 指令的 LR/SC 序列最终会成功完成。

14.7 RVC 指令集列表

表 14.3 给出了 RVC 主要操作码的映射表。对于指令长度超过 16 位的指令，其最低两位都是 1，包括那些处于基本 ISA 中的指令。一些指令仅在某些操作数时是有效的；当无效时，它们要么被标记为 ***RES***，意味着这个操作码被保留给未来的标准扩展；要么被标记为 ***NSE***，意味着这个操作码被保留给未来的非标准扩展；或者被标记为 ***HINT***，意味着这个操作码被保留给未来的微体系结构提示（hint）。在提示没有效果的实现上，标记为 ***HINT*** 的指令必须作为空操作指令执行。

HINT 指令被设计成支持未来增加微体系结构提示，这些提示可能影响性能，但是不能影响体系结构状态。HINT 编码已经被选定，因此简单的实现可以忽略 HINT 编码，并将 HINT 指令作为常规指令执行，不改变体系结构状态。例如，C.ADD 指令的目标寄存器如果是 ***x0****，那么它是一条 HINT 指令，此处 5 位的* ***rs2*** *字段编码了 HINT 的细节。然而，一个简单的实现可以简单地把 HINT 当作一条目标是* ***x0*** *的加法指令执行，这时将没有效果（即 NOP 指令）。*

表 14.4-表 14.6 列出了 RVC 指令。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

inst[15:13] inst[1:0]	000	001	010	011	100	101	110	111	
00	ADDI4SPN	FLD FLD LQ	LW	FLW LD LD	Reserved	FSD FSD SQ	SW	FSW SD SD	RV32 RV64 RV128
01	ADDI	JAL ADDIW ADDIW	LI	LUI/ADDI16SP	MISC-ALU	J	BEQZ	BNEZ	RV32 RV64 RV128
10	SLLI	FLDSP FLDSP LQ	LWSP	FLWSP LDSP LDSP	J[AL]R/MV/ADD	FSDSP FSDSP SQ	SWSP	FSWSP SDSP SDSP	RV32 RV64 RV128
11	>16 位								

表 14.3：RVC 操作码映射表

15 14 13	12 11 10	9 8 7	6 5	4 3 2	1 0	
000	0			0	00	非法指令
000	nzimm[5:4|9:6|2|3]			rd’	00	C.ADDI4SPN (RES, nzimm=0)
001	imm[5:3]	rs1’	imm[7:6]	rd’	00	C.FLD (RV32/64)
001	imm[5:4|8]	rs1’	imm[7:6]	rd’	00	C.LQ (RV128)
010	imm[5:3]	rs1’	imm[2|6]	rd’	00	C.LW
011	imm[5:3]	rs1’	imm[2|6]	rd’	00	C.FLW (RV32)
011	imm[5:3]	rs1’	imm[7:6]	rd’	00	C.LD (RV64/128)
100					00	Reserved
101	imm[5:3]	rs1’	imm[7:6]	rs2’	00	C.FSD (RV32/64)
101	imm[5:4|8]	rs1’	imm[7:6]	rs2’	00	C.SQ (RV128)
110	imm[5:3]	rs1’	imm[2|6]	rs2’	00	C.SW
111	imm[5:3]	rs1’	imm[2|6]	rs2’	00	C.FSW (RV32)
111	imm[5:3]	rs1’	imm[7:6]	rs2’	00	C.SD (RV64/128)

表 14.4：RVC 指令列表，00 部分

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0	
000	0	0		0		01	C.NOP
000	nzimm[5]	rs1/rd≠0		nzimm[4:0]		01	C.ADDI (HINT, nzimm=0)
001	offset[11|4|9:8|10|6|7|3:1|5]					01	C.JAL (RV32)
001	imm[5]	rs1/rd≠0		imm[4:0]		01	C.ADDIW (RV64/128; RES, rd=0)
010	imm[5]	rs1/rd≠0		imm[4:0]		01	C.LI (HINT, rd=0)
011	nzimm[9]	2		nzimm[4|6|8:7|5]		01	C.ADDI16SP (RES, nzimm=0)
011	nzimm[17]	rs1/rd≠{0,2}		nzimm[16:12]		01	C.LUI (RES, nzimm=0; HINT, rd=0)
100	nzimm[5]	00	rs1′/rd′	nzimm[4:0]		01	C.SRLI (RV32 NSE, nzimm[5]=1)
100	0	00	rs1′/rd′	0		01	C.SRLI64 (RV128; RV32/64 HINT)
100	nzimm[5]	01	rs1′/rd′	nzimm[4:0]		01	C.SRAI (RV32 NSE, nzimm[5]=1)
100	0	01	rs1′/rd′	0		01	C.SRAI64 (RV128; RV32/64 HINT)
100	imm[5]	10	rs1′/rd′	imm[4:0]		01	C.ANDI
100	0	11	rs1′/rd′	00	rs2′	01	C.SUB
100	0	11	rs1′/rd′	01	rs2′	01	C.XOR
100	0	11	rs1′/rd′	10	rs2′	01	C.OR
100	0	11	rs1′/rd′	11	rs2′	01	C.AND
100	1	11	rs1′/rd′	00	rs2′	01	C.SUBW (RV64/128; RV32 RES)
100	1	11	rs1′/rd′	01	rs2′	01	C.ADDW (RV64/128; RV32 RES)
100	1	11	——	10	rs2′	01	Reserved
100	1	11	——	11	rs2′	01	Reserved
101	offset[11|4|9:8|10|6|7|3:1|5]					01	C.J
110	offset[8|4:3]		rs1′	offset[7:6|2:1|5]		01	C.BEQZ
111	offset[8|4:3]		rs1′	offset[7:6|2:1|5]		01	C.BNEZ

表 14.5：RVC 指令列表，01 部分

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

15 14 13	12	11 10 9 8 7	6 5 4 3 2	1 0	
000	nzimm[5]	rd≠0	nzimm[4:0]	10	C.SLLI (HINT, rd=0; RV32 NSE, nzimm[5]=1)
000	0	rd≠0	0	10	C.SLLI64 (RV128; RV32/64 HINT; HINT, rd=0)
001	imm[5]	rd	imm[4:3|8:6]	10	C.FLDSP (RV32/64)
001	imm[5]	rd≠0	imm[4|9:6]	10	C.LQSP (RV128; RES, rd=0)
010	imm[5]	rd≠0	imm[4:2|7:6]	10	C.LWSP (RES, rd=0)
011	imm[5]	rd	imm[4:2|7:6]	10	C.FLWSP (RV32)
011	imm[5]	rd≠0	imm[4:3|8:6]	10	C.LDSP (RV64/128; RES, rd=0)
100	0	rs1≠0	0	10	C.JR (RES, rs1=0)
100	0	rd≠0	rs2≠0	10	C.MV (HINT, rd=0)
100	1	0	0	10	C.EBREAK
100	1	rs1≠0	0	10	C.JALR
100	1	rd≠0	rs2≠0	10	C.ADD (HINT, rd=0)
101	imm[5:3|8:6]		rs2	10	C.FSDSP (RV32/64)
101	imm[5:4|9:6]		rs2	10	C.SQSP (RV128)
110	imm[5:2|7:6]		rs2	10	C.SWSP
111	imm[5:2|7:6]		rs2	10	C.FSWSP (RV32)
111	imm[5:3|8:6]		rs2	10	C.SDSP (RV64/128)

表 14.6：RVC 指令列表，10 部分

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

14.8 指令压缩统计

下面一些表格给出了一些数据，我们使用这些数据来指导选择将什么指令包含到 RVC 中。

表 14.7 列出了标准 RVC 指令，按照使用频度从高到低排序，给出了对静态代码大下，单条指令的贡献，然后运行了总共 3 个实验。对 RV32，RVC 在 Dhrystone 减少了静态代码 24.5%，在 CoreMark 减少了 30.9%。对 RV64，RVC 在 SPECint 减少了静态代码 26.3%，在 SPECfp 减少了 25.8%，在 Linux kernel 减少了 31.1%。

表 14.8 根据典型动态频率对 RVC 指令进行了排序。对 RV32，RVC 在 Dhrystone 减少了取指的动态字节 29.2%，在 CoreMark 减少了 29.3%。对 RV64，RVC 在 SPECint 减少了取指的动态字节 26.9%，在 SPECfp 减少了 22.4%，在启动 Linux kernel 时减少了 26.11%。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

指令	RV32GC			RV64GC		MAX
	Dhry-stone	Core-Mark	SPEC 2006	SPEC 2006	Linux Kernel	
C.MV	1.78	5.03	4.06	3.62	5	5.03
C.LWSP	4.51	2.8	2.89	0.49	0.14	4.51
C.LDSP	—	—	—	3.2	4.44	4.44
C.SWSP	4.19	2.45	2.76	0.45	0.18	4.19
C.SDSP	—	—	—	2.75	3.79	3.79
C.LI	2.99	3.74	2.81	2.35	2.86	3.74
C.ADDI	2.16	3.28	1.87	1.19	0.95	3.28
C.ADD	0.51	1.64	1.94	2.28	0.91	2.28
C.LW	2.1	1.68	2	0.74	0.62	2.1
C.LD	—	—	—	1.14	2.09	2.09
C.J	0.32	1.71	1.63	0.97	1.53	1.71
C.SW	1.59	0.85	0.73	0.27	0.26	1.59
C.JR	1.52	1.16	0.49	0.44	1.05	1.52
C.BEQZ	0.38	1.14	0.76	0.55	1.24	1.24
C.SLLI	0.06	1.09	0.57	0.93	0.57	1.09
C.ADDI16SP	0.19	0.26	0.32	0.42	1.01	1.01
C.SRLI	0	0.81	0.05	0.12	0.31	0.81
C.BNEZ	0.19	0.53	0.53	0.32	0.8	0.8
C.SD	—	—	—	0.25	0.79	0.79
C.ADDIW	—	—	—	0.77	0.5	0.77
C.JAL	0.38	0.59	0.05	—	—	0.59
C.ADDI4SPN	0.57	0.37	0.45	0.5	0.3	0.57
C.LUI	0.32	0.37	0.44	0.56	0.52	0.56
C.SRAI	0.13	0.48	0.07	0.03	0.03	0.48
C.ANDI	0	0.42	0.2	0.07	0.35	0.42
C.FLD	0	0	0.16	0.39	0	0.39
C.FLDSP	0	0.02	0.2	0.31	0	0.31
C.FSDSP	0.13	0.09	0.15	0.26	0	0.26
C.SUB	0.25	0.09	0.13	0.06	0.11	0.25
C.AND	0	0	0.07	0.03	0.21	0.21
C.FSD	0	0	0.08	0.18	—	0.18
C.OR	0.06	0.18	0.09	0.04	0.14	0.18
C.JALR	0.13	0.07	0.17	0.1	0.14	0.17
C.ADDW	—	—	—	0.16	0.12	0.16
C.EBREAK	0	0.02	0	0	0.08	0.08
C.FLW	0	0	0.05	—	—	0.05
C.XOR	0	0.04	0.01	0.01	0.03	0.04
C.SUBW	—	—	—	0.04	0.03	0.04
C.FLWSP	0	0	0.03	—	—	0.03
C.FSW	0	0	0.02	—	—	0.02
C.FSWSP	0	0	0.02	—	—	0.02
总计	24.46	30.92	25.78	25.98	25.98	—

表 14.7：按典型静态频率排序的 RVC 指令。表中的数据给出了每条指令在静态代码大小中节约的比例。这个列表是由通过一个压缩汇编器产生的，它对 RISC-V GCC 编译器的输出进行处理。对 RV32GC 使用了 Dhrystone、CoreMark 和 SPEC CPU2006，对 RV64GC 使用了 SPEC CPU2006 和 Linux kernel 3.14.29 版本。表中的横线表示该指令没有在这个地址大小下面的定义。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

指令	RV32GC		RV64GC		MAX
	Dhry-stone	Core-Mark	SPEC 2006	Linux Kernel	
C.ADDI	3.7	3.91	4.36	1.26	4.36
C.LW	4.15	3.89	1.09	0.87	4.15
C.MV	1.93	4.01	1.7	1.37	4.01
C.BNEZ	0.44	2.57	0.47	3.62	3.62
C.SW	3.55	1.62	0.32	0.68	3.55
C.LD	—	—	1.43	3.29	3.29
C.SWSP	3.26	0.32	0.2	0.03	3.26
C.LWSP	2.96	0.48	0.14	0.02	2.96
C.LI	2.22	1.47	0.81	2.73	2.73
C.ADD	2.07	2.69	2.64	1.84	2.69
C.SRLI	0	2.48	0.2	0.38	2.48
C.JR	2.07	0.34	0.46	0.42	2.07
C.FLD	0	0	1.63	0	1.63
C.SDSP	—	—	1.14	1.38	1.38
C.J	0.44	0.46	0.33	1.35	1.35
C.LDSP	—	—	1.34	1.31	1.34
C.ANDI	0.15	1.3	0.1	0.23	1.3
C.ADDIW	—	—	1.26	1.03	1.26
C.SLLI	0.15	1.1	1.24	0.89	1.24
C.SD	—	—	0.39	1.13	1.13
C.BEQZ	0.59	0.95	0.74	0.76	0.95
C.AND	0	0	0.21	0.75	0.75
C.SRAI	0	0.72	0.02	0.01	0.72
C.JAL	0.59	0.26	—	—	0.59
C.ADDI4SPN	0.44	0.16	0.07	0.05	0.44
C.FLDSP	0	0	0.4	0	0.4
C.ADDI16SP	0.13	0.18	0.28	0.38	0.38
C.FSD	0	0	0.29	0	0.29
C.FSDSP	0	0	0.25	0	0.25
C.ADDW	—	—	0.19	0.04	0.19
C.XOR	0	0.19	0.06	0.02	0.19
C.OR	0.15	0.08	0.05	0.04	0.15
C.SUB	0.15	0.03	0.05	0.04	0.15
C.LUI	0.02	0.06	0.09	0.1	0.1
C.JALR	0	0.05	0.05	0.03	0.05
C.SUBW	—	—	0.04	0.02	0.04
C.EBREAK	0	0	0	0	0
C.FLW	0	0	—	—	—
C.FLWSP	0	0	—	—	—
C.FSW	0	0	—	—	—
C.FSWSP	0	0	—	—	—
总计	29.18	29.29	24.03	26.11	—

表 14.8：按典型动态频率排序的RVC指令。表中的数据给出了每条指令在动态代码大小中节约的比例。这个列表是通过执行来获得的。对RV32GC执行了Dhrystone、CoreMark，对RV64GC执行了SPEC CPU2006，对于SPEC，我们使用了参考输入集。Linux启动时间包括引导内核、执行init进程、执行shell以及poweroff命令。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

14.9 优化寄存器保存/恢复代码大小

在函数入口、出口处的寄存器保存/恢复代码占据了静态代码大小的很重要组成部分。RVC 中的基于栈指针的 load 和 store 指令可以有效地减少一半的保存恢复静态代码大小，同时通过减少动态指令带宽，提高了执行性能。

标准 RISC-V 软件工具链提供了另外一种更进一步减少保存/恢复静态代码大小的方法，这是以降低性能来交换的。与将寄存器保存/恢复代码嵌入到每个函数中不同，寄存器保存代码被一条跳转并链接指令代替，它将调用一个子过程将寄存器复制到栈中，然后再返回函数。寄存器恢复代码被一条跳转指令代替，它将跳转到一个子过程从栈中恢复寄存器，然后再跳转到恢复的返回地址。

图 14.1 给出了当直接应用到 SPEC CPU 2006 基准测试程序的所有函数上时，这些子过程对静态代码和动态指令数目的影响。平均来说，代码大小减少了 4%，而动态指令数目增大了 3%。

当将-Os（减少代码大小）标志传递给 gcc 时，嵌入函数的保存/恢复代码被替换成调用保存/恢复子过程。

在其他 ISA 中，另外一种减少保存/恢复代码大小的机制是 load-multiple、store-multiple 指令。我们考虑过将它们加入到 RISC-V 中，但是注意到这些指令有下面这些不足：

- *这些指令导致复杂的处理器实现。*
- *对于虚拟存储器系统来说，一些数据访问可能处在物理存储器中，而另外一些数据访问不在，这就需要一种新的机制，以重启已经部分执行了的指令。（**译者注：比如 LDM r2,r3 指令，假设需要读取到 r2 数据在存储器中，而 r3 不在。那么在处理异常时，需要重新启动 LDM 指令的一部分，因为 r2 上次已经读取了**）*
- *与其他的 RVC 指令不同，Load multiple 和 Store multiple 并没有对等的 IFD。*
- *与其他的 RVC 指令不同，编译器必须在生成指令和分配寄存器时，特别注意这些指令，以便最大化它们被按序保存和恢复，因为它们将来会被按序保存和恢复的。*
- *简单的微体系结构实现，将会限制在 load 和 store multiple 指令周围，如何调度其他指令，导致潜在的性能损失。*
- *理想的按顺序寄存器分配可能与为 CIW、CL、CS 和 CB 格式选择的特别寄存器冲突。*

虽然一些体系结构设计师可能会得出不同的结论，但是我们决定去掉 load 和 store multiple 支持，而是使用调用保存/恢复子过程的软件方法，来获得最大限度地代码大小减少。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

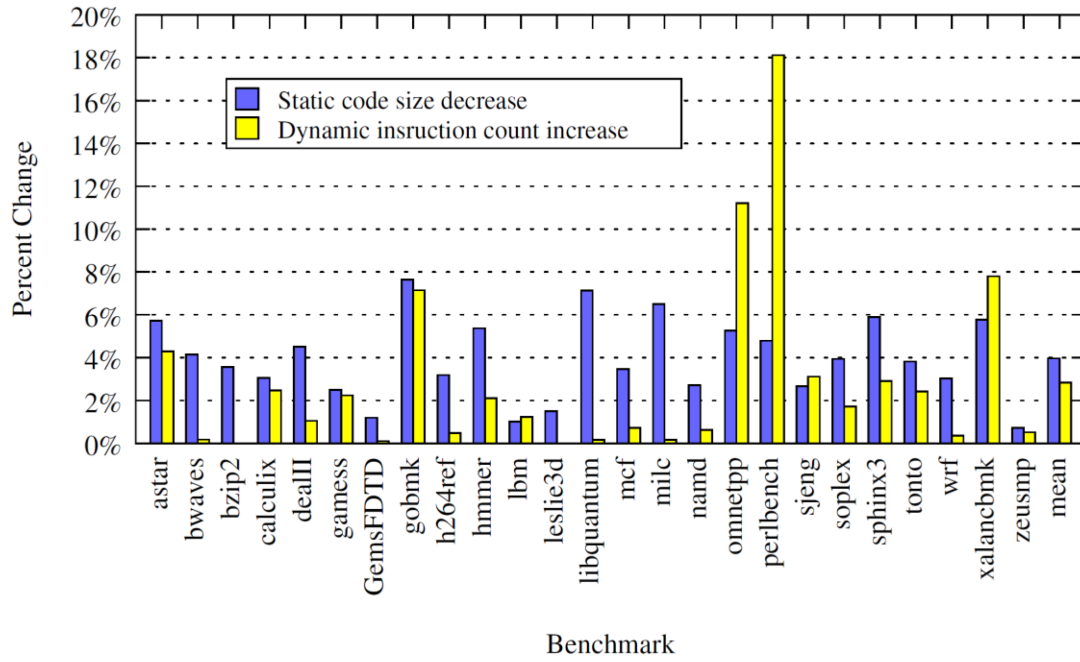


图 14.1：压缩的函数入口、出口处的子过程，对静态代码大小和动态指令数目的影响。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第15章　向量操作的“V”标准扩展，版本 0.0

本章是未来向量指令标准扩展的预留位置。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第16章　位操作的“B”标准扩展，版本 0.0

本章是未来位操作指令标准扩展的预留位置，包括插入、抽取、测试位字段、旋转、漏斗移位（funnel shift）、位和字节排序。

虽然位操作指令在某些应用领域非常高效，特别是当处理在外部打包的数据结构时，但是我们仍然将它们从基本 ISA 中去除，因为它们并不是在所有领域都有用，而且为了支持所需要的操作数，会增加额外的复杂性或者指令格式。

我们预测 B 扩展将会是基本 30 位指令空间的棕地编码。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第17章　事务存储器的“T”标准扩展，版本 0.0

本章是未来提供事务存储器操作标准扩展的预留位置。

尽管在过去二十年进行了大量的研究，以及初步的商业实现，仍然存在对涉及到多地址时，如何最好地支持原子性操作大量的争议。

我们当前的想法是，引入一个小的、有限容量的事务存储器缓冲区，以及原来的事务存储器的一些行。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第18章　打包 SIMD 指令的“P”标准扩展，版本 0.1

本章中，我们粗略地描述RISC-V的一个标准打包SIMD（packed-SIMD）扩展。我们为未来的打包SIMD扩展标准集保留了指令子集名字“P”。可以在一个打包SIMD扩展上构建许多其他的扩展，利用与整数单元相分离的、宽的数据寄存器和数据通路。

打包 SIMD 扩展，是由 Lincoln Labs TX-2[5]首先引入的，已经变成在数据并行代码上提供更高吞吐率的流行方法。早期商业微处理器实现，包括 Intel i860、HP PA-RISC MAX[15]、SPARC VIS[24]、MIPS MDMX[8]、PowerPC AltiVec[4]、Intel x86 MMX/SSE[19][21]，以及最近的设计，包括 Intel x86 AVX[16]、ARM Neon[7]。我们在本章描述了打包 SIMD 的标准框架，但是目前并没有上面做一些事情。我们认为，当重用已有的宽数据通路资源时，打包的 SIMD 才是一个合理的设计点，但是如果大量的额外的资源专门用于数据并行执行，那么基于传统的向量结构将是一个更好的选择，应当使用 V 扩展。

RISC-V打包SIMD扩展重用了浮点寄存器（**f0-f31**）。这些寄存器的宽度可以是FLEN=32到FLEN=1024。标准的浮点指令子集需要的寄存器宽度是32位（“F”）、64位（“D”）或者128位（“Q”）。

将浮点寄存器而不是整数寄存器（PA-RISC 和 Alpha 打包 SIMD 扩展）用来保存打包的SIMD 值，这很自然，因为这将释放整数寄存器用于控制和地址值，重用标量浮点单元为 SIMD 浮点执行，并得到自然的去耦合整数/浮点硬件设计。浮点的load 和store 指令编码也有空间用于处理更宽的打包SIMD 寄存器。然而，重用浮点寄存器保存打包的SIMD 值，其难度并不亚于为浮点值使用一种重新编码的内部格式。

现有的浮点load和store指令被用于将各种大小的字写入到**f**寄存器中。基本ISA支持32位和64位load和store，但是LOAD-FP和STORE-FP指令编码允许编码8种不同的宽度，如表 18.1所示。当用于打包SIMD扩展时，在硬件上最好支持非自然对齐的load和store。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

width字段	代码	位宽
000	B	8
001	H	16
010	W	32
011	D	64
100	Q	128
101	Q2	256
110	Q4	512
111	Q8	1024

表 18.1：LOAD-FP 和 STORE-FP width 编码

打包SIMD计算指令对保存在f寄存器里面的打包值进行操作。每个值可以是8位、16位、32位、64位或者128位，同时可以支持整数和浮点表示。例如，一个64位打包SIMD扩展可以把每个寄存器当做1×64位、2×32位、4×16位或者8×8位打包值。

简单的打包 SIMD 扩展可能可以嵌入到未使用的 32 位指令操作码中，但是更复杂的打包 SIMD 扩展很可能需要一个专门的 30 位指令空间。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第19章　RV128I 基本整数指令集，版本 1.7

“在设计计算机时，只有一个错误是难以修复的——没有足够的地址位来寻址存储器和进行存储器管理” Bell 和 Strecker，ISCA-3，1976

本章描述RV128I，它是一个支持平坦128位地址空间的RISC-V ISA变种。这个变种是现有RV32I和RV64I直截了当的延伸。

扩展整数寄存器宽度的主要原因在于支持更大的地址空间。现在还不清楚需要比 64 位地址空间更大的平坦地址空间什么时候会变得是必须的。在书写此文档时，TOP500 基准测试程序测量的全球最快超级计算机拥有 1PB 的 DRAM，如果将所有的 DRAM 都置于同一个地址空间内，将需要超过 50 位的地址空间。一些数据仓库级别的计算机（warehouse-scale computers）已经包含了更多的 DRAM，新的、密集的固态非易失存储器和快速的互连技术可能需要更大的存储器空间。Exascale 系统研究瞄准了 100PB 存储器系统，它将占用 57 位地址空间。按照历史的发展速度，可能在 2030 年前，将需要超过 64 位的地址空间。

历史说明，一旦清晰认识到需要超过 64 位地址空间，体系结构设计师将就其他扩展地址空间的方法展开激烈大辩论，包括分段、96 位地址空间、以及软件变通方法，直到最后将 128 位地址空间作为最简单、最好的方法予以采纳。

我们此时并没有冻结 RV128 规范，因为可能需要发展出基于真实使用 128 位地址空间的设计。

RV128I在RV64I基础上构建，如果RV64I在RV32I基础上构建一样，是将整数寄存器扩展到128位（就是XLEN=128）。绝大多数整数计算指令并没有发生改变，而是操作到XLEN位上。RV64I对寄存器低32位操作的“*W”整数指令保持不变，加入了一个对保存在128位寄存器的低64位数值进行操作的“*D”整数指令集合。在标准32位编码中，“*D”指令消耗了2个主要操作码（OP-IMM-64和OP-64）。

使用立即数进行移位的指令（SLLI/SRLI/SRAI），现在使用了I立即数的低7位进行编码，而可变移位指令（SLL/SRL/SRA）使用了移位次数源寄存器的低7位。

加入了一条LDU（load double unsigned）指令，使用现有的LOAD主要操作码，还有新的LQ和SQ指令用于load和store四字数值（**译者注：四字=128位**）。SQ指令被加入到STORE主要操作码，而LQ指令被加入到MISC-MEM主要操作码。

浮点指令集并没有发生改变，但现在128位Q浮点扩展可以支持FMV.X.Q和FMV.Q.X指令了，而且还有新增的FCVT指令用于转换T（128位）整数格式。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第20章　调用约定

本章描述RV32和RV64程序的C编译器标准，以及两个调用约定：基本ISA加上标准通用扩展（RV32G和RV64G）的调用约定，在缺少浮点单元的实现（即RV32I和RV64I）上软浮点的约定。

具有 ISA 扩展的实现，可能需要扩展调用约定。

20.1　C 数据类型和对齐

表 20.1总结了RISC-V C程序天然支持的数据类型。在RV32和RV64 C编译器中，C的***int***类型是32位宽度，而***long***类型、指针类型的宽度和对应的整数寄存器宽度一样，因此在RV32中都是32位，而在RV64中都是64位。RV32使用了ILP32整数模型，而RV64使用了ILP64整数模型。在RV32和RV64中，C类型***long long***都是64位整数、***float***都是32位IEEE 754-2008浮点数、***double***都是64位IEEE 754-2008浮点数、***long double***都是128位IEEE浮点数。

当保存到RISC-V整数寄存器中时，C类型***char***和***unsigned char***都是8位无符号整数并且都被零扩展。当保存到RISC-V整数寄存器中时，***unsigned short***是16位无符号整数并且被零扩展。当保存到RISC-V整数寄存器中时，***signed char***是8位无符号整数并且被符号扩展，即（XLEN-1）..7位相同。当保存到RISC-V整数寄存器中时，***short***是16位有符号整数并且被符号扩展。

在RV64中，32位类型，例如***int***，保存到整数寄存器的时进行了正确的符号扩展，即63..31位相同。即使对无符号32位类型都是如此。

RV32和RV64 C编译器和兼容软件在把上述数据类型保存到存储器中时，保证自然对齐。

C类型	描述	RV32中字节数	RV64中字节数
char	字符值/字节	1	1
short	短整型	2	2
int	整型	4	4
long	长整型	4	8
long long	长长整型	8	8
void*	指针	4	8
float	单精度浮点	4	4
double	双精度浮点	8	8
long double	四精度浮点	16	16

表 20.1：基本 RISC-V ISA 的 C 编译器数据类型

20.2　RVG 调用约定

RISC-V的调用约定，尽可能在寄存器中传递参数。多达8个整数寄存器，**a0-a7**，和多达8

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

个浮点寄存器，**fa0-fa7**，用于这个目的。

如果传递给一个函数的参数是C ***struct***定义好的字段时，每一个字段按指针对齐，参数寄存器就是***struct***最前面8个指针字（pointer-word）的影子（shadow）。如果参数<8是一个浮点类型，它将通过浮点寄存器***fai***传递；否则，它将通过整数寄存器***ai***传递。然而，作为***union***的一部分或者***struct***数组的浮点参数，将通过整数寄存器传递。另外，变长函数（variadic fuction）（除了那些在参数列表中明确命名的）浮点参数也通过整数寄存器传递。

比一个指针字小的参数，通过参数寄存器的LSB传递。对应的，通过栈传递的小于指针字的参数，出现在指针字的较低地址上，因为RISC-V是一个小端的存储器系统。

当通过栈传递的原始参数是指针字的两倍的时候，它们是自然对齐的。当它们通过整数寄存器传递时，它们被放置在对齐的偶-奇（even-odd）寄存器对中（**译者注：比如a0和a1**），其中的偶数编号寄存器保存了LSB。例如在RV32中，函数***void foo(int, long long)***通过**a0**传递它的第一个参数，通过**a2**、**a3**传递第二个参数。寄存器**a1**中没有东西。

大小大于两倍指针字的参数，通过引用（***reference***）传递。

定义好的***struct***中不能通过参数寄存器传递的部分，将通过栈传递。栈指针***sp***指向了第一个没有在寄存器中的参数。

函数的返回值存放在整数寄存器**a0**和**a1**、浮点寄存器**fa0**和**fa1**中。只有当它们是原始参数或者是只包含1个或者2个浮点值的***struct***的时候，浮点值才通过浮点寄存器返回。其他返回值能放入两个指针字大小的，返回值放入**a0**和**a1**。更大的返回值都全部通过存储器返回；调用者分配这些存储器区域，并将它作为第一个隐藏参数传递给被调用者。

在标准RISC-V调用约定中，栈是向下增长并且栈指针总是对齐到16字节的（**译者注：应该是对齐到16位？**）。

除了参数寄存器和返回值寄存器之外，7个整数寄存器**t0-t6**和12个浮点寄存器**ft0-ft11**是临时寄存器，它们在调用过程中被破坏，如果后面还有使用的话，在调用者中必须先保存。12个整数寄存器**s0-s11**和12个浮点寄存器**fs0-fs11**在调用过程后被保持不变，如果需要使用的话，在被调用者中必须保存。表 20.2指明了在调用约定中每个整数寄存器和浮点寄存器扮演的角色。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

寄存器	ABI名字	描述	保存者
x0	zero（零）	硬件连线0	—
x1	ra	返回地址	调用者
x2	sp	栈指针	被调用者
x3	gp	全局指针	—
x4	tp	线程指针	—
x5-7	t0-2	临时变量	调用者
x8	s0/fp	保存的寄存器/帧指针	被调用者
x9	s1	保存的寄存器	被调用者
x10-11	a0-1	函数参数/返回值	调用者
x12-17	a2-7	函数参数	调用者
x18-27	s2-11	保存的寄存器	被调用者
x28-31	t3-6	临时变量	调用者
f0-7	ft0-7	FP临时变量	调用者
f8-9	fs0-1	FP保存的寄存器	被调用者
f10-11	fa0-1	FP参数/返回值	调用者
f12-17	fa2-7	FP参数	调用者
f18-27	fs2-11	FP保存的寄存器	被调用者
f28-31	ft8-11	FP临时变量	调用者

表 20.2：RISC-V 调用约定寄存器使用情况

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

20.3 软浮点调用约定

软浮点调用约定用于缺少浮点硬件的RV32和RV64实现中。它避免使用所有的F、D、Q标准扩展中的指令，以及**f**寄存器。

整型参数如同RVG那样被传递和返回，并且栈的规则也相同，除了栈只需要对齐到XLEN/8字节边界（例如，对RV32I是4字节对齐）。

因为在栈上的浮点数据是通过整数 load 和 store 访问的，所以在软浮点调用约定中不需要在更粗的粒度上维护栈对齐。减小栈对齐、节约空间的存储器紧张系统，通常使用软浮点。

浮点参数通过整数寄存器传递和返回，使用相同大小整数参数相同的规则。例如在RV32中，函数***double foo(int, double, long double)***通过**a0**传递第一个参数，通过**a2**和**a3**传递第二个参数，通过**a4**传递第三个参数的引用；它的结果在**a0**和**a1**中返回。在RV64中，参数通过**a0**、**a1**、**a2-a3**对，来传递，结果从**a0**返回。

动态舍入模式和已发生异常标志，是通过C99头文件**fenv.h**提供的例程来访问的。

20.4 RV32E 调用约定

RV32E使用软浮点调用约定的子集。由于只有16个整数寄存器**x0-x15**，因此只有6个参数寄存器（**x10-x15**）、2个保存的寄存器（**x8-x9**）和3个临时寄存器（**x5-x7**）。栈对齐到4字节边界。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第21章　RISC-V 汇编程序员手册

本章是汇编程序员手册的预留位置。

表21.1给出了标准RISC-V伪指令的列表。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

<table>
<tr><th>伪指令</th><th>基本指令</th><th>含义</th></tr>
<tr><td>la rd, symbol</td><td>auipc rd, symbol[31:12]
addi rd, rd, symbol[11:0]</td><td>加载地址</td></tr>
<tr><td>l{b|h|w|d} rd, symbol</td><td>auipc rd, symbol[31:12]
l{b|h|w|d} rd, symbol[11:0](rd)</td><td>加载全局</td></tr>
<tr><td>s{b|h|w|d} rd, symbol, rt</td><td>auipc rt, symbol[31:12]
s{b|h|w|d} rd, symbol[11:0](rt)</td><td>保存全局</td></tr>
<tr><td>fl{w|d} rd, symbol, rt</td><td>auipc rt, symbol[31:12]
fl{w|d} rd, symbol[11:0](rt)</td><td>浮点加载全局</td></tr>
<tr><td>fs{w|d} rd, symbol, rt</td><td>auipc rt, symbol[31:12]
fs{w|d} rd, symbol[11:0](rt)</td><td>浮点保存全局</td></tr>
<tr><td>nop</td><td>addi x0, x0, 0</td><td>空操作</td></tr>
<tr><td>li rd, immediate</td><td>无数的序列</td><td>加载立即数</td></tr>
<tr><td>mv rd, rs</td><td>addi rd, rs, 0</td><td>复制寄存器</td></tr>
<tr><td>not rd, rs</td><td>xori rd, rs, -1</td><td>1的补码</td></tr>
<tr><td>neg rd, rs</td><td>sub rd, x0, rs</td><td>2的补码</td></tr>
<tr><td>negw rd, rs</td><td>subw rd, x0, rs</td><td>2的补码字</td></tr>
<tr><td>sext.w rd, rs</td><td>addiw rd, rs, x0</td><td>符号扩展字</td></tr>
<tr><td>seqz rd, rs</td><td>sltiu rd, rs, 1</td><td>等于0则置位</td></tr>
<tr><td>snez rd, rs</td><td>sltu rd, x0, rs</td><td>不等于0则置位</td></tr>
<tr><td>sltz rd, rs</td><td>slt rd, rs, x0</td><td>小于0则置位</td></tr>
<tr><td>sgtz rd, rs</td><td>slt rd, x0, rs</td><td>大于0则置位</td></tr>
<tr><td>fmv.s rd, rs</td><td>fsgnj.s rd, rs, rs</td><td>复制单精度寄存器</td></tr>
<tr><td>fabs.s rd, rs</td><td>fsgnjx.s rd, rs, rs</td><td>单精度绝对值</td></tr>
<tr><td>fneg.s rd, rs</td><td>fsgnjn.s rd, rs, rs</td><td>单精度负值</td></tr>
<tr><td>fmv.d rd, rs</td><td>fsgnj.d rd, rs, rs</td><td>复制双精度寄存器</td></tr>
<tr><td>fabs.d rd, rs</td><td>fsgnjx.d rd, rs, rs</td><td>双精度绝对值</td></tr>
<tr><td>fneg.d rd, rs</td><td>fsgnjn.d rd, rs, rs</td><td>双精度负值</td></tr>
<tr><td>beqz rs, offset</td><td>beq rs, x0, offset</td><td>不等于0则分支</td></tr>
<tr><td>bnez rs, offset</td><td>bne rs, x0, offset</td><td>等于0则分支</td></tr>
<tr><td>blez rs, offset</td><td>bge x0, rs, offset</td><td>小于等于0则分支</td></tr>
<tr><td>bgez rs, offset</td><td>bge rs, x0, offset</td><td>大于等于0则分支</td></tr>
<tr><td>bltz rs, offset</td><td>blt rs, x0, offset</td><td>小于0则分支</td></tr>
<tr><td>bgtz rs, offset</td><td>blt x0, rs, offset</td><td>大于0则分支</td></tr>
<tr><td>j offset</td><td>jal x0, offset</td><td>转移</td></tr>
<tr><td>jal offset</td><td>jal x1, offset</td><td>转移并连接</td></tr>
<tr><td>jr rs</td><td>jalr x0, rs, 0</td><td>转移到寄存器</td></tr>
<tr><td>jalr rs</td><td>jalr x1, rs, 0</td><td>转移并连接寄存器</td></tr>
<tr><td>ret</td><td>jalr x0, x1, 0</td><td>从子过程返回</td></tr>
<tr><td>call offset</td><td>auipc x6, offset[31:12]
jalr x1, x6, offset[11:0]</td><td>调用远处的子过程</td></tr>
<tr><td>tail offset</td><td>auipc x6, offset[31:12]
jalr x0, x6, offset[11:0]</td><td>跟踪调用远处的子过程（Tail call far-away subroutine）</td></tr>
</table>

表 21.1：RISC-V 伪指令

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

第22章　历史和致谢

22.1 到 ISA 手册 1.0 版本以前的历史

RISC-V ISA和指令集手册构建于几个早期的项目。管理员级机器的几个方面和数据手册的整个格式，可以追溯到始于1992年的UC Berkeley和ICSI的T0（Torrent-0）向量微处理器项目。T0是一款基于MIPS-II ISA的向量处理器，其主要体系结构设计师、RTL设计师是Krste Asanović，主要的VLSI实现者是Brian Kingsbury、Bertrand Irrisou。ICSI的David Johnson是T0 ISA设计、部分管理员模式、手册文本的主要贡献者。John Hauser为T0 ISA设计提供了大量的反馈。

在MIT始于2000的Scale（Software-Controlled Architecture for Low Energy），是在T0项目基础上构建的，修订了管理员级接口，并通过丢弃分支延迟槽而脱离了MIPS标量ISA。MIT的Scale Vector-Thread处理器主要体系结构设计师是Ronny Krashinsky和Christopher Batten，而Mark Hampton负责把基于GCC的编译器和工具移植到了Scale。

在2002年秋季学期，一个T0 MIPS标量处理器规范的修订版本（MIPS-6371）被用于新版本的MIT 6.371“VLSI概述”课程教学，教师是Chris Terman和Krste Asanović。Chris Terman完成了绝大部分课程实验的材料（课程并没有TA！）。这个6.371课程在2005年春季，发展成为MIT的6.884“复杂数字设计”试行课程，教师是Arvind和Krste Asanović，后来发展成6.375春季正式课程。一个称为SMIPS的简化版Scale基于MIPS的标量ISA，被用于6.884/6.375课程。Christopher Batten作为最早的TA，围绕SMIPS ISA开发了大量的文档和实验材料。这些SMIPS实验材料被TA Yunsup Lee采用并增强，在2009年秋季在UC Berkeley的CS250“VLSI系统设计”课程中使用，教师是John Wawrzynek、Krste Asanović和John Lazzaro。

Marven（Malleable Array of Vector-thread ENgines）是第二代向量线程体系结构。这个设计是由Christopher Batten领导，当时他还是于2007年夏开始的UC Berkeley访问学者。一名来自Hitachi的访问专家Hidetaka Aoki，为早期的Marven ISA和微体系结构提供了大量的反馈。Marven结构是基于Scale结构的，但是Marven ISA更加远离了Scale定义的MIPS ISA变种，它有一个统一的浮点和整数寄存器文件。Marven被设计用于支持备选的数据并行加速器实验。Yunsup Lee是各种Marven向量单元的主要实现者，Rimas Avižienis是各种Marven标量单元的实现者。Yunsup Lee和Christopher Batten将GCC移植到新的Marven ISA上。Christopher Celio提供了Marven的一个传统向量指令集（“Flood”）变种的初始定义。

基于所有前面这些项目，RISC-V ISA定义始于2010年夏。RISC-V 32位指令子集的一个初始版本被用于UC Berkeley 2010年秋的CS250“VLSI系统设计”课程，Yunsup Lee是课程的TA。RISC-V与早期受MIPS启发的设计完全不同。John Hauser为浮点ISA定义做了贡献。

22.2 从 ISA 手册 2.0 版本以来的历史

完成了多款RISC-V处理器实现，包括几款硅片制造，如表 22.1所示。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

名字	流片日期	工艺	ISA
Raven-1	2011-05-29	ST 28nm FDSOI	RV64G1 Xhwacha1
EOS14	2012-04-01	IBM 45nm SOI	RV64G1p1 Xhwacha2
EOS16	2012-08-17	IBM 45nm SOI	RV64G1p1 Xhwacha2
Raven-2	2012-08-22	ST 28nm FDSOI	RV64G1p1 Xhwacha2
EOS18	2013-01-06	IBM 45nm SOI	RV64G1p1 Xhwacha2
EOS20	2013-07-03	IBM 45nm SOI	RV64G1p99 Xhwacha2
Raven-3	2013-09-26	ST 28nm SOI	RV64G1p99 Xhwacha2
EOS22	2014-03-07	IBM 45nm SOI	RV64G1p9999 Xhwacha3

表 22.1：已经制造的 RISC-V 测试芯片

首款被制造的RISC-V处理器使用Verilog书写的，并作为Raven-1测试芯片，于2011年在ST的预先量产28nm FDSOI工艺上制造。在Krste Asanović的指导下，Yunsup Lee和Andrew Waterman研发了两个内核，并一同被制造：1）一个具有错误检测触发器的RV64标量内核，2）一个具有64位浮点向量单元的RV64内核。最开始的微体系结构被称为“火车失事（TrainWreck）”，主要是因为使用不成熟的设计库在短时间内完成了设计。

接着，一个全新的、按序、去耦合RV64内核微体系结构，在Krste Asanović指导下，由Andrew Waterman、Rimas Avižienis和Yunsup Lee研发出来，继续铁轨模式，代号“火箭（Rocket）”，按照George Stephenson成功的蒸汽机车设计而命名。Rocket是使用Chisel语言书写的，Chisel是由UC Berkeley开发的一种新的硬件设计语言。Rocket中的IEEE浮点单元是由John Hauser、Andrew Waterman和Brian Richards开发的。由此，Rocket被进一步修正和改进，并2次在28nm FDSOI工艺上制造（Raven-2、Raven-3），为一个光子学项目而5次在IBM 45nm SOI工艺上制造（EOS14、EOS16、EOS18、EOS20、EOS22）。现在正在做的工作就是将Rocket设计做成一个可参数化的RISC-V处理器生成器。

EOS14-EOS22芯片包含了一个早期版本的Hwacha，它是一个64位IEEE 浮点向量单元，由Yunsup Lee、Andrew Waterman、Huy Vo、Albert Ou、Quan Nguyen、Stephen Twigg在Krste Asanović的指导下研发的。EOS16-EOS22包含了具有cache一致性协议的双核，在Krste Asanović的指导下由Henry Cook、Andrew Waterman研发。EOS14芯片成功地运行在1.25GHz，EOS16芯片遭受IBM pad库的一个bug影响，EOS18和EOS20成功地运行在1.35GHz。

Raven测试芯片的贡献者包括Yunsup Lee、Andrew Waterman、Rimas Avižienis、Brian Zimmer、Jaehwa Kwak、Ruzica Jevtić、Milovan Blagojević、Alberto Puggelli、Steven Bailey、Ben Keller、Pi-Feng Chiu、Brian Richards、Borivoje Nikolić和Krste Asanović。

EOS测试芯片的贡献者包括Yunsup Lee、Rimas Avižienis、AndrewWaterman、Henry Cook、Huy Vo、Daiwei Li、Chen Sun、Albert Ou、Quan Nguyen、Stephen Twigg、Vladimir Stojanović和Krste Asanović。

Andrew Waterman和Yunsup Lee开发了C++ ISA仿真器“Spike”，作为研发过程中的黄金模型，其名字来源于在US横贯大陆的铁路竣工庆典上所使用的黄金铁道钉。Spike已经成为一个BSD开源项目。

AndrewWaterman完成了一个具有初步RISC-V压缩指令集设计的硕士论文[28]。

完成了各种各样的RISC-V FPGA实现，主要是Par Lab项目研究的集成演示的一部分。最新的FPGA设计具有3个cache一致性的RV64IMA处理器，运行了一个研究用的操作系统。FPGA实现的贡献者包括Andrew Waterman、Yunsup Lee、Rimas Avižienis和Krste Asanović。

RISC-V处理器被用于UC Berkeley的数门课程。Rocket被用于2011年秋的CS250，作为

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

课程项目的基础，Brian Zimmer是课程TA。在2012年春的本科生CS152课程上，Christopher Celio使用ChiSel语言书写了一系列适合教学用途的RV32处理器，命名为“Sodor”，这是“托马斯小火车”和他的小伙伴们生活的小岛。这个套件包括一个微编码内核（microcoded core）、一个非流水内核、2级流水内核、3级流水内核、5级流水内核，并在BSD授权下公开共享。这个套件于2013年春在CS152课程中得到更新和再度使用，Yunsup Lee是课程TA，2014年春Eric Love是课程TA。Christopher Celio也开发了一个乱序执行的RV64设计，称为BOOM（Berkeley Out-of-Order Machine），并具有流水线可视化功能，被用于CS152课程。CS152课程还使用了由Andrew Waterman和Henry Cook开发的具有cache一致性版本的Rocket内核。

2013年夏，定义了RoCC（Rocket Custom Coprocessor）接口，以简化向Rocket内核添加定制加速器的工作。在2013年秋，Rocket和RoCC接口被大量应用于CS250 VLSI课程，教师是Jonathan Bachrach，好几个学生加速器项目是构建在RoCC接口之上的。Hwacha向量单元作为RoCC协处理器被重写。

两名Berkeley本科生，Quan Nguyen和Albert Ou，在2013年春，成功的将Linux移植到RISC-V上。

在2014年1月，Colin Schmidt成功的完成了RISC-V 2.0的LLVM后端。

在2014年3月，Bluespec的Darius Rad贡献了软浮点ABI支持的GCC移植。

我们也注意到几个其他的RISC-V内核实现，包括一个由Tommy用Verilog书写，一个由Rishiyur Nikhil用Bluespec书写。

致谢

感谢Christopher F. Batten、Preston Briggs、Christopher Celio、David Chisnall、Stefan Freudenberger、John Hauser、Ben Keller、Rishiyur Nikhil、Michael Taylor、Tommy Thorn和Robert Watson为ISA 2.0版本规范的初稿提出意见。

22.3 版本 2.1 的历史

自从2014年5月2.0版本冻结之后，RISC-V ISA被非常快速地采用，不能在如此短小的章节里记录下太多的活动。可能其中最重要的事件是在2015年8月成立了非盈利的RISC-V基金会。基金会现在接管了官方RISC-V标准的管理工作，而官方网站riscv.org是获取有关RISC-V标准新闻和更新最好的去处。

致谢

感谢Scott Beamer、Allen J. Baum、Christopher Celio、David Chisnall、Paul Clayton、Palmer Dabbelt、Jan Gray、Michael Hamburg和John Hauser为ISA 2.0版本规范提出意见。

22.4 资助

RISC-V体系结构和实现的研发，部分的由下列赞助商资助。

- **Par Lab:** 研究受Microsoft (Award #024263)、Intel (Award #024894)资助，并由U.C.

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

Discovery (Award #DIG07-10227)配套资助。其他支持来自Par Lab的伙伴Nokia、NVIDIA、Oracle和Samsung。

- **Project Isis：**DoE Award DE-SC0003624。
- **Silicon Photonics：**DARPA POEM program Award HR0011-11-C-0100。
- **ASPIRE Lab：**DARPA PERFECT program，Award HR0011-12-2-0016。DARPA POEM program Award HR0011-11-C-0100。The Center for Future Architectures Research (C-FAR)，Semiconductor Research Corporation资助的STARnet中心。其他支持来自ASPIRE产业赞助商、Intel和ASPIRE的伙伴Google、Hewlett Packard Enterprise、Huawei、Nokia、NVIDIA、Oracle和Samsung。

本文的内容并不代表US政府的立场或者政策，并且没有暗示官方的认可。

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

参考文献

[1] IEEE standard for a 32-bit microprocessor. IEEE Std. 1754-1994, 1994.
[2] G. M. Amdahl, G. A. Blaauw, and Jr. F. P. Brooks. Architecture of the IBM System/360. *IBM Journal of R. & D.*, 8(2), 1964.
[3] Werner Buchholz, editor. *Planning a computer system: Project Stretch*. McGraw-Hill Book Company, 1962.
[4] K. Diefendorff, P.K. Dubey, R. Hochsprung, and H. Scale. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85-95, 2000.
[5] John M. Frankovich and H. Philip Peterson. A functional description of the Lincoln TX-2 computer. In *Western Joint Computer Conference*, Los Angeles, CA, February 1957.
[6] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15-26, 1990.
[7] J. Goodacre and A.N. Sloss. Parallelism and the ARM instruction set architecture. *Computer*, 38(7):42 -50, 2005.
[8] Linley Gwennap. Digital, MIPS add multimedia extensions. Microprocessor Report, 1996.
[9] Timothy H. Heil and James E. Smith. Selective dual path execution. Technical report, University of Wisconsin - Madison, November 1996.
[10] ANSI/IEEE Std 754-2008, IEEE standard for floating-point arithmetic, 2008.
[11] Manolis G.H. Katevenis, Robert W. Sherburne, Jr., David A. Patterson, and Carlo H. Sé quin. The RISC II micro-architecture. In *Proceedings VLSI 83 Conference*, August 1983.
[12] Hyesoon Kim, Onur Mutlu, Jared Stark, and Yale N. Patt. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 43-54, 2005.
[13] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, Washington, DC, USA, 1998.
[14] David D. Lee, Shing I. Kong, Mark D. Hill, George S. Taylor, David A. Hodges, Randy H. Katz, and David A. Patterson. A VLSI chip set for a multiprocessor workstation-Part I: An RISC microprocessor with coprocessor interface and support for symbolic processing. *IEEE JSSC*, 24(6):1688-1698, December 1989.
[15] R.B. Lee. Subword parallelism with MAX-2. *IEEE Micro*, 16(4):51 -59, August 1996.
[16] Chris Lomont. Introduction to Intel Advanced Vector Extensions. Intel White Paper, 2011.
[17] OpenCores. OpenRISC 1000 architecture manual, architecture version 1.0, December 2012.
[18] David A. Patterson and Carlo H. Séquin. RISC I: A reduced instruction set VLSI computer. In *ISCA*, pages 443-458, 1981.
[19] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42 -50, August 1996.

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.

[20] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294-305. IEEE Computer Society, 2001.

[21] S.K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming SIMD extensions on the Pentium-III processor. *IEEE Micro*, 20(4):47 -57, 2000.

[22] Balaram Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. IBM POWER7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1-1, 2011.

[23] James E. Thornton. Parallel operation in the Control Data 6600. In *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*, AFIPS '64 (Fall, part II), pages 33-40, 1965.

[24] M. Tremblay, J.M. O'Connor, V. Narayanan, and Liang He. VIS speeds new media processing. *IEEE Micro*, 16(4):10 -20, August 1996.

[25] Marc Tremblay, Jeffrey Chan, Shailender Chaudhry, Andrew W. Conigliaro, and Shing Sheung Tse. The MAJC architecture: A synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12- 25, 2000.

[26] J. Tseng and K. Asanović. Energy-efficient register access. In *Proc. of the 13th Symposium on Integrated Circuits and Systems Design*, pages 377-384, Manaus, Brazil, September 2000.

[27] David Ungar, Ricki Blau, Peter Foley, Dain Samples, and David Patterson. Architecture of SOAR: Smalltalk on a RISC. In *ISCA*, pages 188-197, Ann Arbor, MI, 1984.

[28] Andrew Waterman. Improving energy efficiency and reducing code size with RISC-V compressed. Master's thesis, University of California, Berkeley, 2011.

[29] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V instruction set manual, Volume I: Base user-level ISA. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.

[30] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V instruction set manual, Volume I: Base user-level ISA version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

Copyright ©2010-2016, The Regents of the University of California. All rights reserved.