RISC-V指令集手册

第一卷：用户级ISA

文件版本2.2

编辑：Andrew Waterman[1], Krste Asanovi'c[1,2] [1]SiFive Inc.,
[2]加州大学伯克利分校EECS系CS部门
andrew@sifive.com, krste@berkeley.edu

2017年5月7日

按字母顺序排列所有版本的规范（请联系编辑以建议更正）：Krste Asanovi'c, Rimas Avizienis, Jacob Bachmeyer, Christopher F. Batten, Allen J. Baum, Alex Bradbury, Scott Beamer, Preston Briggs, Christopher Celio, David Chisnall, Paul Clayton, Palmer Dabbelt, Stefan Freudenberger, Jan Gray, Michael Hamburg, John Hauser, David Horner, Olof Johansson, Ben Keller, Yunsup Lee, Joseph Myers, Rishiyur Nikhil, Stefan O'Rear, Albert Ou, John Ousterhout, David Patterson, Colin Schmidt, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn, Ray VanDeWalker, Megan Wachs, Andrew Waterman, Robert Watson和Reinoud Zandijk。

本文档采用知识共享署名4.0国际许可协议发布。

本文档是“RISC-V指令集手册，第I卷：用户级ISA”的衍生产品
版本2.1“根据以下许可证发布：Qc 2010-2017 Andrew Waterman, Yunsup Lee,
大卫帕特森，Krste Asanovi'c。知识共享署名4.0国际许可。

请引用：“RISC-V指令集手册，第I卷：用户级ISA，文档版本2.2”，编辑Andrew Waterman和Krste Asanovi'c, RISC-V基金会，2017年5月。

前言

这是描述RISC-V用户级体系结构的文档的2.2版。该文档包含以下版本的RISC-V ISA模块：

基础	版	冷冻？
rv32i	2.0	Y
RV32I	1.9	N
rv64i	2.0	Y
RV128I	1.7	N
延期	**版**	**冷冻？**
M	2.0	Y
A	2.0	Y
F	2.0	Y
D	2.0	Y
Q	2.0	Y
L	0.0	N
C	2.0	Y
B	0.0	N
J	0.0	N
T	0.0	N
P	0.1	N
V	0.2	N
N	1.1	N

迄今为止，RISC-V基金会尚未正式批准该标准的任何部分，但在批准过程中，上述标记为“冻结”的组件不会发生变化，除非解决规范中的模糊和漏洞。

该版本文件的主要变化包括：

- 本文档的先前版本是根据知识共享署名发布的4.0原始作者的国际许可，本文档的此版本和未来版本将在同一许可下发布。
- 重新排列章节，以规范顺序排列所有扩展名。
- 对说明和评论的改进。
- 修改了JALR的隐式暗示建议，以支持LUI / JALR和AUIPC / JALR对的更有效的宏观操作融合。

- 澄清对负载保留/存储条件序列的约束。
- 一个新的控制和状态寄存器（CSR）映射表。
- 阐明了fcsr的高阶位的目的和行为。
- 更正了FNMADD.fmt和FNMSUB.fmt指令的说明，这些说明建议使用零结果的错误符号。
- 说明FMV.SX和FMV.XS分别重命名为FMV.WX和FMV.XW，以更加符合其语义，但没有改变。工具中将继续支持旧名称。
- 使用NaN-boxing模型在较宽的f寄存器中保存的较窄（<FLEN）浮点值的指定行为。
- 定义了FMA（∞，0，qNaN）的异常行为。
- 添加了注释，指示可以使用整数寄存器将P扩展重新编写为整数打包SIMD提议以进行定点操作。
- V向量指令集扩展的提议草案。
- N个用户级陷阱扩展的早期提案草案。
- 扩展的伪指令列表。
- 删除调用约定章节，该章节已被RISC-V ELF psABI规范[1]取代。
- C扩展已被冻结并重新编号为2.0版。

文件版本2.1的前言

这是描述RISC-V用户级体系结构的文档的2.1版。请注意，冻结的用户级ISA基础和扩展IMAFDQ版本2.0没有从本文档的先前版本[36]更改，但已修复了一些规范漏洞并且文档已得到改进。对软件约定进行了一些更改。

- 评论部分的众多补充和改进。
- 每章的版本号分开。
- 修改长指令编码> 64位以避免以非常长的指令格式移动rd说明符。
- CSR指令现在以引入计数器寄存器的基本整数格式进行描述，而不是稍后在浮点部分（以及随附的特权架构手册）中介绍。
- SCALL和SBREAK指令已分别重命名为ECALL和EBREAK。它们的编码和功能没有变化。
- 澄清浮点NaN处理和新的规范NaN值。
- 澄清由浮点返回的值返回溢出的整数转换。
- LR / SC的澄清允许成功和所需的失败，包括在序列中使用压缩指令。
- 用于减少整数寄存器计数的新RV32E基础ISA提议支持MAC扩展。
- 经修订的召集会议。
- 用于软浮动调用约定的轻松堆栈对齐，以及RV32E调用的描述

惯例。
- 针对C压缩扩展的修订提案，版本1.9。

2.0版前言

这是用户ISA规范的第二个版本，我们打算将基本用户ISA和一般扩展（即IMAFD）的规范保持固定以供将来开发。自ISA规范的1.0版[35]以来，已经进行了以下更改。

- ISA已分为具有多个标准扩展的整数基础。
- 指令格式已重新排列，以使立即编码更有效。
- 基本ISA已定义为具有little-endian内存系统，big-endian或bi-endian作为非标准变体。
- 在原子指令扩展中添加了加载保留/存储条件（LR / SC）指令。
- AMO和LR / SC可以支持发布一致性模型。
- FENCE指令提供更精细的内存和I / O顺序。
- 添加了用于fetch-and-XOR（AMOXOR）的AMO，并且AMOSWAP的编码已更改为腾出空间。
- AUIPC指令向PC添加20位高位立即数，取代RDNPC指令，该指令仅读取当前PC值。这样可以显着节省与位置无关的代码。
- 现在，JAL指令已移至具有显式目标寄存器的U-Type格式，并且Jd指令已被丢弃，由JAL替换为rd = x0。这将删除具有隐式目标寄存器的唯一指令，并从基本ISA中删除J-Type指令格式。随着JAL覆盖范围的减少，基本ISA复杂性显着降低。
- JALR指令的静态提示已被删除。对于符合标准调用约定的代码，rd和rs1寄存器说明符的提示是多余的。
- JALR指令现在清除计算目标地址的最低位，以简化硬件并允许辅助信息存储在函数指针中。
- MFTX.S和MFTX.D指令已分别重命名为FMV.XS和FMV.XD。同样，MXTF.S和MXTF.D指令已分别重命名为FMV.SX和FMV.DX。
- MFFSR和MTFSR指令已分别重命名为FRCSR和FSCSR。添加了FRRM，FSRM，FRFLAGS和FSFLAGS指令以单独访问fcsr的舍入模式和异常标志子字段。
- FMV.XS和FMV.XD指令现在从rs1而不是rs2中获取其操作数。此更改简化了数据路径设计。
- 添加了FCLASS.S和FCLASS.D浮点分类指令。
- 采用了更简单的NaN生成和传播方案。
- 对于RV32I，系统性能计数器已扩展到64位宽，对上部和下部32位具有单独的读访问权限。
- 已经定义了Canonical NOP和MV编码。

- 已为48位，64位和> 64位指令定义了标准指令长度编码。
- 添加了128位地址空间变体RV128的描述。
- 已为用户定义的自定义扩展分配了32位基本指令格式的主要操作码。
- 一个印刷错误表明存储从rd获取数据已被更正为引用rs2。

内容

前言 i

1 介绍 1

1.1 RISC-V ISA概述 3

1.2 指令长度编码 5

1.3 例外，陷阱和中断 7

2 RV32I Base Integer指令集，版本2.0 9

2.1 基本整数子集的程序员模型 9

2.2 基本指令格式 11

2.3 即时编码变体 11

2.4 整数计算指令 13

2.5 控制转移指令 15

2.6 加载和存储指令 18

2.7 记忆模型 20

2.8 控制和状态寄存器指令 21

2.9 环境呼叫和断点 24

3 RV32E基本整数指令集，版本1.9 27

3.1 RV32E程序员模型 27

3.2 RV32E指令集 27

3.3 RV32E扩展 28

翻译狗 www.fanyigou.net www.translateGo.net

4 RV64I基本整数指令集，版本2.0 **29**

4.1 登记国 ... 29

4.2 整数计算指令 ... 29

4.3 加载和存储指令 ... 31

4.4 系统说明 ... 32

5 RV128I基本整数指令集，版本1.7 **33**

6 "M"标准扩展为整数乘法和除法，版本2.0 **35**

6.1 乘法运算 ... 35

6.2 分部运作 ... 36

7 "A"原子指令的标准扩展，版本2.0 **39**

7.1 指定原子指令的排序 ... 39

7.2 加载保留/存储条件指令 ... 40

7.3 原子内存操作 ... 43

8 单精度浮点的"F"标准扩展，版本2.0 **45**

8.1 F注册州 ... 45

8.2 浮点控制和状态寄存器 ... 47

8.3 NaN生成和传播 ... 48

8.4 次正规算术 ... 49

8.5 单精度加载和存储指令 ... 49

8.6 单精度浮点计算指令 ... 49

8.7 单精度浮点转换和移动指令 ... 51

8.8 单精度浮点比较指令 ... 52

8.9 单精度浮点分类指令 ... 53

9 双精度浮点的"D"标准扩展，版本2.0 **55**

9.1 D登记州 ... 55

9.2 NaN拳击更窄的价值观 55
9.3 双精度加载和存储指令 56
9.4 双精度浮点计算指令 57
9.5 双精度浮点转换和移动指令 57
9.6 双精度浮点比较指令 59
9.7 双精度浮点分类指令 59

10 “Q”标准扩展为四精度浮点版本2.0 **61**

10.1 四精度加载和存储指令 61
10.2 四精度计算指令 62
10.3 四精度转换和移动指令 62
10.4 四精度浮点比较指令 63
10.5 四精度浮点分类指令 63

11 “L”十进制浮点的标准扩展，版本0.0 **65**

11.1 十进制浮点寄存器 65

12 “C”压缩指令的标准扩展，版本2.0 **67**

12.1 概观 67
12.2 压缩指令格式 69
12.3 加载和存储指令 71
12.4 控制转移指令 74
12.5 整数计算指令 76
12.6 在LR / SC序列中使用C指令 80
12.7 RVC指令集列表 81

13 用于位操作的“B”标准扩展，版本0.0 **85**

14 动态翻译语言的“J”标准扩展，版本0.0 **87**

15 事务性内存的“T”标准扩展，版本0.0 **89**

16 “P”标准扩展包装SIMD指令，版本0.1 **91**

17 矢量操作的“V”标准扩展，版本0.2 **93**

17.1 矢量单位国家 93

17.2 元素数据类型和宽度 93

17.3 矢量配置寄存器（vcmaxw，vctype，vcp） 95

17.4 矢量长度 97

17.5 快速配置说明 97

18 用户级中断的“N”标准扩展，版本1.1 **101**

18.1 其他CSR 101

18.2 用户状态寄存器（ustatus） 102

18.3 其他CSR 102

18.4 N扩展指令 102

18.5 减少上下文交换开销 102

19 RV32 / 64G指令集清单 **103**

20 RISC-V汇编程序员手册 **109**

21 扩展RISC-V **113**

21.1 扩展术语 113

21.2 RISC-V扩展设计理念 116

21.3 固定宽度32位指令格式的扩展 116

21.4 添加对齐的64位指令扩展 118

21.5 支持VLIW编码 118

22 ISA子集命名约定 **121**

22.1 区分大小写 121

22.2 基本整数ISA 121

22.3 指令扩展名称 121

22.4 版本号 122

22.5 非标准扩展名 122
22.6 监督级指令子集 122
22.7 主管级扩展 122
22.8 子集命名约定 123
23历史和致谢 125
23.1 ISA手册修订版1.0的历史 125
23.2 ISA手册修订版2.0的历史 126
23.3 修订版2.1的历史 128
23.4 修订版2.2的历史 128
23.5 资金 129

第1章

介绍

RISC-V（发音为“risk-five”）是一种新的指令集架构（ISA），最初设计用于支持计算机体系结构研究和教育，但我们现在希望它也将成为行业实现的标准免费开放架构。我们定义RISC-V的目标包括：

- 一个完全开放的ISA，可供学术界和工业界免费使用。
- 真正的ISA适用于直接本机硬件实现，而不仅仅是模拟或二进制转换。
- ISA避免针对特定微体系结构样式（例如，微编码，有序，解耦，无序）或实现技术（例如，全定制，ASIC，FPGA）的“过度架构”，但允许高效任何这些实施。
- ISA分为一个小的基本整数ISA，可用作自定义加速器或教育用途的基础，以及可选的标准扩展，以支持通用软件开发。
- 支持修订的2008 IEEE-754浮点标准[14]。
- ISA支持广泛的用户级ISA扩展和专用变体。
- 适用于应用程序，操作系统内核和硬件实现的32位和64位地址空间变体。
- ISA支持高度并行的多核或多核实现，包括异构多处理器。
- 可选的可变长度指令，用于扩展可用的指令编码空间，并支持可选的密集指令编码，以提高性能，静态代码大小和能效。
- 完全可虚拟化的ISA，可简化虚拟机管理程序开发。
- ISA，通过新的管理程序级和管理程序级ISA设计简化实验。

我们的设计决策评论的格式如本段所述，如果读者只对规范本身感兴趣，可以跳过。

选择RISC-V这个名称来代表加州大学伯克利分校的第五个主要RISC ISA设计（RISC-I [23]，RISC-II [15]，SOAR [32]和SPUR [18]是前四个）。我们也是双关语

翻译狗 www.fanyigou.net www.translateGo.net

使用罗马数字“V”来表示“变化”和“向量”，因为支持一系列架构研究，包括各种数据并行加速器，是ISA设计的明确目标。

我们开发了RISC-V以支持我们自己在研究和教育方面的需求，我们的团队对研究思路的实际硬件实施特别感兴趣（自本规范第一版以来，我们已经完成了11种不同的RISC-V芯片制造），以及为学生提供在课堂上探索的真实实施（RISC-V处理器RTL设计已在伯克利的多个本科和研究生课程中使用）。在我们目前的研究中，我们对特殊和异构加速器的转变特别感兴趣，这是由传统晶体管缩放结束所带来的功率限制所驱动的。我们需要一个高度灵活和可扩展的基础ISA来围绕它建立我们的研究工作。

我们一再提出的问题是“为什么要开发新的ISA？”使用现有商用ISA的最大好处是大型且广泛支持的软件生态系统，包括开发工具和移植应用程序，可用于研究和教学。其他好处包括存在大量文档和教程示例。但是，我们在研究和教学中使用商业教学集的经验是，这些好处在实践中较小，并且不会超过缺点:

- 商*业ISA是专有的。除了SPARC V8，这是一个开放的IEEE标准[2]，商业ISA的大多数所有者都谨慎地保护他们的知识产权，不欢迎免费提供竞争性实施。对于仅使用软件模拟器的学术研究和教学来说，这不是一个问题，但对于希望分享实际RTL实现的团体来说，这是一个主要问题。对于那些不想信任商业ISA实现的少数几个来源但却被禁止创建自己的洁净室实现的实体来说，这也是一个主要问题。我们无法保证所有RISC-V实施都不会受到第三方专利侵权，但我们可以保证我们不会试图起诉RISC-V实施者。*
- 商*业ISA仅在某些市场领域受欢迎。撰写本文时最明显的例子是ARM体系结构在服务器领域得不到很好的支持，英特尔x86体系结构（或者几乎所有其他体系结构）在移动领域得不到很好的支持，尽管两者都是英特尔和ARM正试图进入彼此的细分市场。另一个例子是ARC和Tensilica，它们提供可扩展的核心，但专注于嵌入式空间。这种市场细分削弱了支持特定商业ISA的好处，因为在实践中，软件生态系统仅存在于某些域，并且必须为其他域构建。*
- 商*业ISA来来去去。以前的研究基础设施是围绕不再受欢迎的商业ISA（SPARC，MIPS）或甚至不再生产（Alpha）而建立的。这些都失去了活跃的软件生态系统的好处，围绕ISA和支持工具的挥之不去的知识产权问题干扰了感兴趣的第三方继续支持ISA的能力。开放的ISA也可能失去知名度，但任何有兴趣的一方都可以继续使用和开发生态系统。*
- 流*行的商业ISA很复杂。主流商用ISA（x86和ARM）在硬件中实现非常复杂，无法支持通用软件堆栈和操作系统。更糟糕的是，几乎所有的复杂性都是由于糟糕的，或者至少是过时的ISA设计决策，而不是真正提高效率的功能。*
- 仅*商业ISA不足以提出应用程序。即使我们花费了很多精力来实现商业ISA，但这还不足以运行该ISA的现有应用程序。大多数应用程序需要运行完整的ABI（应用程序二进制接口），而不仅仅是用户级ISA。大多数ABI依赖于库，而库依赖于操作系统支持。要运行现有操作系统，需要实现操作系统所需的管理程序级ISA和设备接口。与用户级ISA相比，这些通常没有明确规定，实现起来要复杂得多。*

- 流*行的商业ISA不是为可扩展性而设计的。主要的商业ISA并非特别设计用于可扩展性，因此随着其指令集的增长，增加了相当多的指令编码复杂性。Tensilica（被Cadence收购）和ARC（被Synopsys收购）等公司围绕可扩展性构建了ISA和工具链，但专注于嵌入式应用而非通用计算系统。*
- 改*进的商业ISA是一种新的ISA。我们的主要目标之一是支持架构研究，包括主要的ISA扩展。即使很小的扩展也会降低使用标准ISA的好处，因为必须修改编译器并从源代码重建应用程序以使用扩展。引入新架构状态的较大扩展也需要修改操作系统。最终，改进的商用ISA成为一个新的ISA，但携带基础ISA的所有传统包袱。*

我们的立场是ISA可能是计算系统中最重要的接口，并且没有理由认为这样一个重要的接口应该是专有的。占主导地位的商业ISA基于30多年前已经众所周知的指令集概念。软件开发人员应该能够针对开放的标准硬件目标，商业处理器设计人员应该在实现质量上展开竞争。

我们远非第一个考虑适合硬件实现的开放式ISA设计。我们还考虑了其他现有的开放式ISA设计，其中最接近我们目标的是OpenRISC架构[22]。出于以下几个技术原因，我们决定不采用OpenRISC ISA:

- *OpenRISC具有条件代码和分支延迟槽，这使得更高性能的实现变得复杂。*
- *OpenRISC使用固定的32位编码和16位立即数，从而排除了更密集的指令编码，并限制了以后扩展ISA的空间。*
- *OpenRISC不支持2008年对IEEE 754浮点标准的修订。*
- *我们开始时，OpenRISC 64位设计还没有完成。*

从一个干净的平板开始，我们可以设计一个满足我们所有目标的ISA，当然，这比我们在一开始就计划的工作要多得多。我们现在已经投入了大量精力来构建RISC-V ISA基础架构，包括文档，编译器工具链，操作系统端口，参考ISA模拟器，FPGA实现，高效ASIC实现，架构测试套件和教学材料。自本手册的最后一版以来，RISC-V ISA在学术界和工业界都有相当大的应用，我们创建了非盈利的RISC-V基金会来保护和推广该标准。RISC-V基金会网站http: // riscv。组织包含有关使用RISC-V的基金会成员资格和各种开源项目的最新信息。

RISC-V手册分为两卷。本卷涵盖了用户级ISA设计，包括可选的ISA扩展。第二个卷提供特权架构。

在此用户级手册中，我们旨在消除对特定微体系结构功能或特权体系结构详细信息的依赖性。这既是为了清晰，也是为了替代实现的最大灵活性。

1.1 RISC-V ISA概述

RISC-V ISA被定义为基本整数ISA，它必须存在于任何实现中，以及基本ISA的可选扩展。基本整数ISA与早期的非常相似

RISC处理器，除了没有分支延迟槽并支持可选的可变长度指令编码。基础被严格限制为一组最小的指令，足以为编译器，汇编器，链接器和操作系统提供合理的目标（具有额外的管理员级操作），因此提供了方便的ISA和软件工具链“骨架”，可以构建更多定制的处理器ISA。

每个基本整数指令集的特征在于整数寄存器的宽度和用户地址空间的相应大小。有两个主要的基本整数变体RV32I和RV64I，在第2章和第4章中描述，它们分别提供32位或64位用户级地址空间。硬件实现和操作系统可能仅为用户程序提供RV32I和RV64I中的一个或两个。第3章介绍RV32I基本指令集的RV32E子集变体，该变量已添加到支持小型微控制器。第5章描述了支持128位用户地址空间的基本整数指令集的未来RV128I变体。基本整数指令集对有符号整数值使用二进制补码表示。

虽然64位地址空间是大型系统的必要条件，但我们相信32位地址空间在未来几十年仍将适用于许多嵌入式和客户端设备，并且需要降低内存流量和能耗。此外，32位地址空间足以用于教育目的。最终可能需要更大的平坦128位地址空间，因此我们确保可以在RISC-V ISA框架内实现这一点。

基本整数ISA可以是硬件实现的子集，但是必须使用更特权层的操作码陷阱和软件仿真来实现硬件不提供的功能。

基本整数ISA的子集对于教学目的可能是有用的，但是已经定义了基础，使得除了省略对未对齐的存储器访问的支持并将所有SYSTEM指令视为单个陷阱之外，应该没有动机将实际硬件实现子集化。

RISC-V旨在支持广泛的定制和专业化。基本整数ISA可以使用一个或多个可选指令集扩展进行扩展，但不能重新定义基本整数指令。我们将RISC-V指令集扩展划分为标准和非标准扩展。标准扩展通常应该是有用的，不应与其他标准扩展冲突。非标准扩展可能是高度专业化的，或者可能与其他标准或非标准扩展冲突。指令集扩展可以提供略微不同的功能，具体取决于基本整数指令集的宽度。第21章介绍了扩展RISC-V ISA的各种方法。我们还为RISC-V基本指令和指令集扩展开发了一个命名约定，详见第22章。

为了支持更通用的软件开发，定义了一组标准扩展以提供整数乘法/除法，原子操作以及单精度和双精度浮点算法。基本整数ISA命名为“I”（前缀为RV32或RV64，取决于整数寄存器宽度），并包含整数计算指令，整数加载，整数存储和控制流指令，并且对于所有RISC-V实现都是必需的。标准整数乘法和除法扩展名为“M”，并添加指令以对整数寄存器中保存的值进行乘法和除法。标准的原子指令扩展，用“A”表示，添加了原子读取，修改和写入存储器的指令，用于处理器间同步。标准的单精度浮点扩展，用“F”表示，增加了浮点数

寄存器，单精度计算指令和单精度加载和存储。标准的双精度浮点扩展，用“D”表示，扩展了浮点寄存器，并添加了双精度计算指令，加载和存储。整数基数加上这四个标准扩展（“IMAFD”）的缩写为“G”，并提供通用标量指令集。RV32G和RV64G是我们编译器工具链的默认目标。后面的章节描述了这些和其他计划的标准RISC-V扩展。

除了基本整数ISA和标准扩展之外，新指令很少会为所有应用程序提供显着的好处，尽管它可能对某个域非常有用。由于能效问题迫使人们更加专业化，我们认为简化ISA规范所需的部分非常重要。虽然其他架构通常将其ISA视为单个实体，随着时间的推移随着指令的增加而变为新版本，RISC-V将努力保持基础和每个标准扩展随时间保持不变，而是将新指令作为进一步可选扩展。例如，基本整数ISA将继续作为完全支持的独立ISA，而不管任何后续扩展。

随着用户ISA规范的2.0版本，我们打算将“RV32IMAFD”和“RV64IMAFD”基础和标准扩展（又名“RV32G”和“RV64G”）保持不变，以便将来开发。

1.2 指令长度编码

基本RISC-V ISA具有固定长度的32位指令，必须在32位边界上自然对齐。但是，标准RISC-V编码方案旨在支持具有可变长度指令的ISA扩展，其中每条指令的长度可以是任意数量的16位指令包，并且包裹在16位边界上自然对齐。第12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，并放宽了对齐约束，允许所有指令（16位和32位）在任何16位边界上对齐，以提高代码密度。

图1.1说明了标准的RISC-V指令长度编码约定。基本ISA中的所有32位指令都将其最低的两位设置为11.可选的压缩16位指令集扩展的最低两位等于00,01或10.标准指令集扩展编码超过32位具有设置为1的附加低位，其中48位和64位长度的约定如图1.1所示。使用比特[14:12]中的3比特字段对80比特和176比特之间的指令长度进行编码，除了前5×16比特字之外还给出16比特字的数量。位[14:12]设置为111的编码保留用于将来更长的指令编码。

考虑到压缩格式的代码大小和节能，我们希望建立对ISA编码方案的压缩格式的支持，而不是将其作为事后补充添加，但为了允许更简单的实现，我们不希望制作压缩格式强制性的。我们还希望允许更长的指令来支持实验和更大的指令集扩展。虽然我们的编码约定要求对核心RISC-V ISA进行更严格的编码，但这有几个有益的效果。

标准G ISA的实现只需要在指令缓存中保存最重要的30位（节省6.25%）。在指令缓存重新填充时，遇到任何指令

字节地址：　基地+ 4　　基地+ 2　　基
础

图1.1：RISC-V指令长度编码。

在低速位清零之前，应将其重新编码为非法的30位指令，然后再存储在高速缓存中以保留非法指令异常行为。

也许更重要的是，通过将我们的基本ISA压缩为32位指令字的子集，我们为自定义扩展留出了更多可用空间。特别是，基本RV32I ISA在32位指令字中使用的编码空间不到1/8。如第21章所述，不需要支持标准压缩指令扩展的实现可以将3个额外的30位指令空间映射为32位固定宽度格式，同时保留对标准> = 32位指令的支持 – 设置扩展名。此外，如果
实现也不需要长度为32位的指令，它可以进一步恢复
四大操作码。

我们认为这是一个特征，任何包含所有零位的指令长度都是不合法的，因为这会快速捕获错误的跳转到零存储区域。类似地，我们还保留包含所有1的指令编码是非法指令，以捕获用未编程的非易失性存储器设备，断开的存储器总线或损坏的存储器设备观察到的其他常见模式。

基础RISC-V ISA具有little-endian内存系统，但非标准版本可以提供big-endian或双端内存系统。指令存储在存储器中，每个16位包裹存储在存储器半字中，根据实现的自然字节顺序。形成一条指令的宗地存储在递增的半字地址中，最低寻址的宗地保持指令规范中编号最小的位，即，无论存储系统的字节顺序如何，指令总是存储在小端的包裹序列中。无论内存系统的字节顺序如何，图1.2中的代码序列都会正确地将32位指令存储到内存中。

我们为RISC-V内存系统选择了little-endian字节排序，因为little-endian系统目前在商业上占主导地位（所有x86系统；iOS，Android和Windows for ARM）。一个小问题是我们还发现小端存储器系统对于硬件设计者来说更自然。但是，某些应用领域，如IP网络，在大端数据结构上运行，因此我们留下了非标准大端或双端系统的可能性。

我们必须修复指令包存储在内存中的顺序，与内存系统字节顺序无关，以确保长度编码位始终显示在

www.fanyigou.net 翻译狗 www.translateGo.net

```
//将x2寄存器中的32位指令存储到x3指向的位置。
sh  x2, 0(x3)    //在第一个包裹中存储低位指令。
srli x2, x2, 16//将高位向下移动到低位， 覆盖x2。
sh  x2, 2(x3)    //在第二个地块中存储高位。
```

图1.2：从寄存器到存储器存储32位指令的推荐代码序列。在大端和小端存储器系统上正确运行，并在与可变长度指令集扩展一起使用时避免错位访问。

半字地址顺序。这允许通过仅检查第一个16位指令包的前几位，由指令获取单元快速确定可变长度指令的长度。一旦我们决定修复小端存储器系统和指令包裹排序，这自然导致将长度编码位置于指令格式的LSB位置，以避免分解操作码字段。

1.3 例外，陷阱和中断

我们使用术语异常来指代在与当前RISC-V线程中的指令相关联的运行时发生的异常情况。我们使用术语陷阱来指代由RISC-V线程中发生的异常情况引起的对陷阱处理程序的控制的同步传输。陷阱处理程序通常在更特权的环境中执行。

我们使用术语“中断”来指代与当前RISC-V线程异步发生的外部事件。当发生必须服务的中断时，选择一些指令来接收中断异常并随后经历陷阱。

以下章节中的指令描述描述了在执行期间引发异常的条件。是否以及如何将这些转换为陷阱取决于执行环境，尽管期望大多数环境在发出异常信号时会采用精确陷阱（浮点异常除外，在标准浮点扩展中，不要造成陷阱）。

我们对“异常”和“陷阱”的使用与IEEE-754浮点标准相匹配。

第2章

RV32I Base Integer指令集，版本2.0

本章介绍RV32I基本整数指令集的2.0版。许多评论也适用于RV64I变体。

---

*RV32I被设计为足以形成编译器目标并支持现代操作系统环境。ISA还旨在减少最小实现中所需的硬件。RV32I包含47条独特的指令，尽管一个简单的实现可能会覆盖8条SCALL / SBREAK / CSRR *指令，只有一条SYSTEM硬件指令总是陷阱并且可能能够将FENCE和FENCE.I指令实现为NOP，从而减少硬件指令数量总计38。RV32I可以模拟几乎任何其他ISA扩展（A扩展除外，它需要额外的硬件支持原子性）。*

2.1 基本整数子集的程序员模型

图2.1显示了基本整数子集的用户可见状态。有31个通用寄存器x1-x31，它们保存整数值。寄存器x0硬连线到常数0.没有硬连线子程序返回地址链接寄存器，但标准软件调用约定使用寄存器x1来保存调用的返回地址。对于RV32，x寄存器为32位宽，对于RV64，它们为64位宽。本文档使用术语XLEN来指代x寄存器的当前宽度（以位数为32或64）。

还有一个用户可见的寄存器：程序计数器pc保存当前指令的地址。

可用的架构寄存器数量会对代码大小，性能和能耗产生很大影响。虽然16个寄存器对于运行编译代码的整数ISA来说可能是足够的，但是使用3地址格式在16位指令中编码具有16个寄存器的完整ISA是不可能的。虽然2地址格式是可能的，但它会增加指令数量并降低效率。我们希望避免使用中间指令大小（例如Xtensa的24位指令）来简化基本硬件实现，并且一次

翻译狗 www.fanyigou.net www.translateGo.net

采用32位指令大小，直接支持32个整数寄存器。大量的整数寄存器也有助于高性能代码的性能，可以广泛使用循环展开，软件流水线和缓存切片。

出于这些原因，我们为基本ISA选择了常规大小的32个整数寄存器。动态寄存器的使用倾向于由少数经常访问的寄存器控制，并且可以优化regfile实现以减少频繁访问的寄存器的访问能量[31]。可选的压缩16位指令格式大多只能访问8个寄存器，因此可以提供密集的指令编码，而额外的指令集扩展可以支持更大的寄存器空间（平面或分层），如果需要的话。

对于资源受限的嵌入式应用程序，我们定义了RV32E子集，它只有16个寄存器（第3章）。

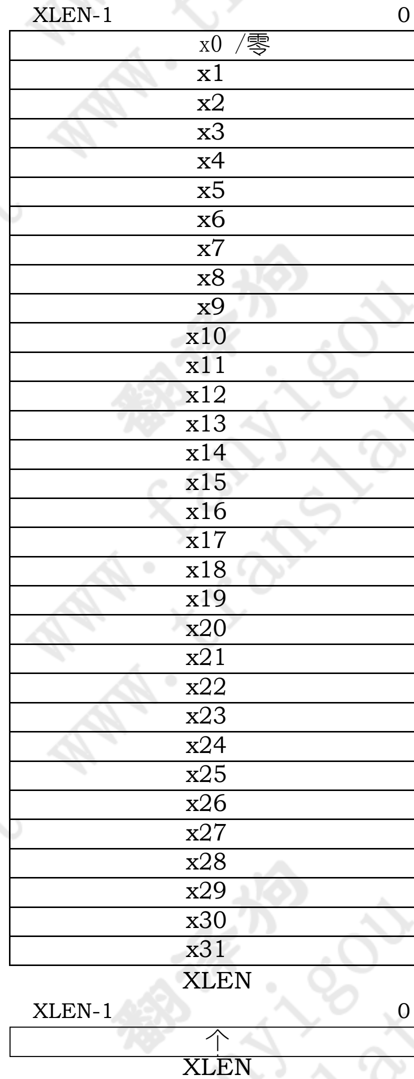


图2.1：RISC-V用户级基本整数寄存器状态。

2.2 基本指令格式

在基础ISA中，有四种核心指令格式（R / I / S / U），如图2.2所示。所有都是固定的32位长度，必须在内存中的四字节边界上对齐。如果目标地址不是四字节对齐，则在采用的分支或无条件跳转上生成指令地址未对齐的异常。对于未采用的条件分支，不生成指令获取未对齐异常。

当添加具有16位长度或16位长度的其他奇数倍的指令扩展时，基本ISA指令的对齐约束被放宽到两字节边界。

图2.2：RISC-V基本指令格式。每个立即子字段都标有正在生成的立即值中的位位置（imm [x]），而不是通常所做的指令立即字段中的位位置。

RISC-V ISA将源（rs1和rs2）和目标（rd）寄存器保持在所有格式的相同位置，以简化解码。除了CSR指令中使用的5位立即数（第2.8节）之外，立即数总是符号扩展，并且通常打包到指令中最左边的可用位，并且已被分配以降低硬件复杂性。特别是，所有立即数的符号位始终位于加速符号扩展电路的指令的第31位。

解码寄存器说明符通常位于实现中的关键路径上，因此选择指令格式是为了将所有寄存器指定符保持在所有格式的相同位置，代价是必须跨格式移动立即位（与RISC-IV共享的属性）又名.SPUR [18]）。

在实践中，大多数中间体要么很小，要么需要所有XLEN位。我们选择了非对称立即分割（常规指令中的12位加上具有20位的特殊加载上部立即指令）以增加可用于常规指令的操作码空间。

Immediates是符号扩展的，因为我们没有观察到对MIPS ISA中的某些immed使用零扩展的好处，并希望尽可能简化ISA。

2.3 即时编码变体

基于对中间体的处理，还有两种指令格式（B / J）变体，如图2.3所示。

S和B格式之间的唯一区别是12位立即数字段用于编码B格式中2的倍数的分支偏移。中间位（imm [10: 1]）和符号位保持在固定位置，而不是将指令编码的所有位在硬件中左移一个硬件，而S位格式的最低位（inst [inst] 7]）以B格式编码高阶位。

类似地，U和J格式之间的唯一区别在于，20位立即数向左移位12位以形成U immediates，并通过1位移位以形成J immediates。选择U和J格式中的指令比特的位置以最大化与其他格式的重叠并且彼此最大化。

31	30 25	24 21	20	19 15	14 12	11 8	7	6 0
功能7		rs2		rs1	功能3	rd		操作码
imm[11:0]				rs1	功能3	rd		操作码
imm [5:11]		rs2		rs1	功能3	imm[4:0]		操作码
imm[12]	IMM [5]	rs2		rs1	功能3	imm[4:1]	imm[11]	操作码
imm[31:12]						rd		操作码
imm[20]	imm [1:10]		imm[11]	imm[19:12]		rd		操作码

R型I
型S型
B型U
型J型

图2.3：显示直接变体的RISC-V基本指令格式。

图2.4显示了每种基本指令格式产生的立即数，并标记为显示哪个指令位（inst [y]）产生立即值的每个位。

31	30 20	19 12	11	10 5	4 1	0
– inst [31] –				研究所 [30:25]	研究所 [24:21]	研究所 [20]
– inst [31] –				研究所 [30:25]	研究所 [11: 8]	研究所[7]
– inst [31] –			研究所[7]	研究所 [30:25]	研究所 [11: 8]	0
研究所 [31]	研究所 [30:20]	研究所 [19:12]	— 0 —			
– inst [31] –		研究所 [19:12]	研究所 [20]	研究所 [30:25]	研究所 [24:21]	0

i- 当 务 之
急 b 型直
接 u 型直
接式 u 型
直接式 u
型直接式
u 型直接
式 u 型直
接 u 型 u
型直接 u
型双即时
双即时

图2.4：RISC-V指令立即生成的类型。字段标有用于构造其值的指令位。符号扩展总是使用inst [31]。

符号扩展是对临界值的最关键操作之一（特别是在RV64I中），在RISC-V中，所有立即数的符号位始终保存在指令的第31位，以允许符号扩展与指令解码并行进行。

虽然更复杂的实现可能具有用于分支和跳转计算的单独加法器，因此不会从保持立即位的位置不变中受益

在教学类型方面，我们希望降低最简单实现的硬件成本。通过在B和J的指令编码中旋转位而不是使用动态硬件多路复用器将立即数乘以2，我们将指令信号扇出和立即多路复用成本减少大约2倍。加扰的立即编码将增加可忽略的时间。静态或提前编译。对于动态生成指令，存在一些小的额外开销，但最常见的短前向分支具有直接的即时编码。

2.4 整数计算指令

大多数整数计算指令对整数寄存器文件中保存的值的XLEN位进行操作。整数计算指令或者使用I型格式编码为寄存器立即操作，或者使用R型格式编码为寄存器寄存器操作。寄存器立即寄存器和寄存器寄存器指令的寄存器rd。没有整数计算指令会导致算术异常。

我们没有为基本指令集中的整数算术运算的溢出检查提供特殊指令集支持，因为可以使用RISC-V分支廉价地实现许多溢出检查。无符号加法的溢出检查在加法后只需要一个附加的分支指令：add t0, t1, t2;bltu t0, t1, 溢出。

对于带符号的加法，如果已知一个操作数的符号，则溢出检查只需要一个
分支后添加：addi t0, t1, + imm;blt t0, t1, 溢出。这包括使用立即操作数添加的常见情况。

对于一般符号加法，需要在加法后添加三个附加指令，当且仅当另一个操作数为负时，利用观察结果应该小于一个操作数。

```
添加t0， t1，
t2 slti t3，
t2,0    slt
t4， t0， t1
t3， t4， 溢出
```

在RV64中，通过比较操作数上ADD和ADDW的结果，可以进一步优化32位带符号加法的检查。

整数寄存器 - 即时指令

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	功能3	rd	操作码
12	5	3	5	7
I-即时[11：0]	src	ADDI /的 SLT I [U]	dest手	OP-IMM
I-即时[11：0]	src	ANDI-TIMESEXORI	dest手	OP-IMM

ADDI将符号扩展的12位立即数添加到寄存器rs1。算术溢出被忽略，结果只是结果的低XLEN位。ADDI rd, rs1,0用于实现MV rd, rs1汇编器伪指令。

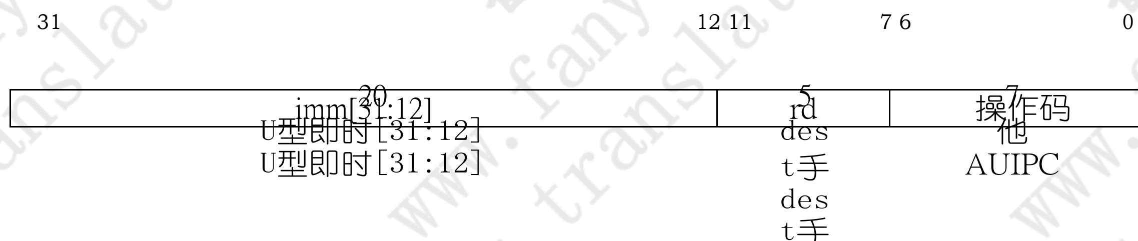
如果寄存器rs1小于符号扩展立即数，当两者都被视为有符号数时，SLTI（设置小于立即数）将值1置于寄存器rd中，否则将0写入rd。SLTIU是

类似但将值作为无符号数进行比较（即，立即首先对符号扩展为XLEN位，然后将其视为无符号数）。注意，如果rs1等于零，则SLTIU rd，rs1,1将rd设置为1，否则将rd设置为0（汇编器伪操作SEQZ rd，rs）。

ANDI，ORI，XORI是逻辑运算，在寄存器rs1和符号扩展的12位立即执行按位AND，OR和XOR，并将结果放在rd中。注意，XORI rd，rs1，-1执行寄存器rs1的逐位逻辑反转（汇编伪指令NOT rd，rs）。

31 25	24 20	19 15	14 12	11 7	6 0
imm [5:11]	imm[4:0]	rs1	功能3	rd	操作码
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest手	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest手	OP-IMM
0100000	shamt[4:0]	src	rias	dest手	OP-IMM

按常数移位被编码为I类型格式的特化。要移位的操作数在rs1中，移位量在I-immediate字段的低5位中编码。右移类型在I-immediate的高位编码。SLLI是逻辑左移（零被移位到低位）;SRLI是一个逻辑右移（零被移入高位）;SRAI是算术右移（原始符号位被复制到空出的高位）。



LUI（load upper immediate）用于构建32位常量并使用U型格式。LUI将U-immediate值放在目标寄存器rd的前20位，用零填充最低的12位。

AUIPC（将上部立即添加到pc）用于构建pc相对地址并使用U型格式。AUIPC与20位U-immediate形成32位偏移，用零填充最低12位，将该偏移添加到pc，然后将结果放入寄存器rd。

AUIPC指令支持双指令序列，以便从PC访问任意偏移，用于控制流传输和数据访问。AUIPC和JALR中的12位立即数的组合可以将控制转移到任何32位PC相对地址，而AUIPC加上常规加载或存储指令中的12位立即偏移可以访问任何32位PC - 相关数据地址。

通过将U-immediate设置为0可以获得当前的PC。虽然也可以使用JAL +4指令来获取PC，但它可能导致更简单的微体系结构中的管道中断或者在更复杂的微体系结构中污染BTB结构。

整数寄存器 - 寄存器操作

RV32I定义了几种算术R类型的操作。所有操作都将rs1和rs2寄存器读作源操作数，并将结果写入寄存器rd。funct7和funct3字段选择

翻译狗 www.fanyigou.net www.translateGo.net

操作类型。

31 25	24 20	19 15	14 12	11 7	6 0
功能7	rs2	rs1	功能3	rd	操作码
7	5	5	3	5	7
0000000	错误预测	上述	ADD/SLTL	dest手	操作
0000000	错误预测	上述	安德/奥罗	dest手	操作
0000000	错误预测	上述	SLL /公司	dest手	操作
0100000	错误预测	上述	子/ SRA	dest手	操作

ADD和SUB分别执行加法和减法。忽略溢出，并将结果的低XLEN位写入目标。SLT和SLTU分别执行有符号和无符号比较，如果rs1 <rs2则写入1到rd，否则为0。注意，如果rs2不等于零，则SLTU rd，x0，rs2将rd设置为1，否则将rd设置为零（汇编伪伪运算SNEZ rd，rs）。AND，OR和XOR执行按位逻辑运算。

SLL，SRL和SRA通过寄存器rs2的低5位中保持的移位量对寄存器rs1中的值执行逻辑左，逻辑右和算术右移。

NOP指令

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	功能3	rd	操作码
12	5	3	5	7
0	0	阿迪	0	OP-IMM

除了推进pc之外，NOP指令不会改变任何用户可见状态。NOP编码为ADDI x0，x0,0。

NOP可用于将代码段与微架构上重要的地址边界对齐，或为内联代码修改留出空间。尽管有许多可能的方法来编码NOP，但我们定义了规范的NOP编码，以允许微架构优化以及更可读的反汇编输出。

2.5 控制转移指令

RV32I提供两种类型的控制传输指令：无条件跳转和条件分支。RV32I中的控制传输指令没有体系结构可见的延迟槽。

无条件跳转

跳转和链接（JAL）指令使用J类型格式，其中J-immediate以2个字节的倍数对带符号的偏移进行编码。偏移是符号扩展并添加到pc以形成跳转目标地址。因此，跳跃可以达到±1 MiB范围。JAL将跳转后的指令地址（pc + 4）存储到寄存器rd中。标准软件调用约定使用x1作为返回地址寄存器，x5作为备用链接寄存器。

备用链路寄存器支持调用毫代码例程（例如，用于在压缩代码中保存和恢复寄存器的例程），同时保留常规返回地址寄存器。寄存器x5被选为备用链接寄存器，因为它映射到标准调用约定中的临时寄存器，并且其编码与常规链接寄存器只有一位不同。

普通的无条件跳转（汇编器伪操作J）被编码为具有rd = x0的JAL。

31	30 ... 21	20	19 ... 12	11 ... 7	6 ... 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	操作码
1	10	1	8	5	7
	偏移[20:1]			dest手	日航

间接跳转指令JALR（跳转和链接寄存器）使用I型编码。通过将12位有符号I-immediate添加到寄存器rs1，然后将结果的最低有效位设置为零来获得目标地址。跳转后的指令地址（pc + 4）被写入寄存器rd。如果不需要结果，寄存器x0可用作目标。

31 ... 20	19 ... 15	14 ... 12	11 ... 7	6 ... 0
imm[11:0]	rs1	功能3	rd	操作码
12	5	3	5	7
偏移[11：0]	基础	0	dest手	JALR

无条件跳转指令都使用PC相对寻址来帮助支持与位置无关的代码。定义JALR指令使双指令序列能够跳转到32位绝对地址范围内的任何位置。LUI指令可以首先使用目标地址的高20位加载rs1，然后JALR可以添加低位。类似地，AUIPC然后JALR可以跳转到32位pc相对地址范围内的任何地方。

请注意，JALR指令不会将12位立即数视为2个字节的倍数，
与条件分支指令不同。这避免了硬件中的一种立即格式。在实践中，JALR的大多数用途将立即为零或与LUI或AUIPC配对，因此范围的轻微减少并不显着。

JALR指令忽略计算目标地址的最低位。这既略微简化了硬件，又允许低位的函数指针用于存储辅助信息。虽然在这种情况下可能会略微丢失错误检查，但实际上跳转到错误的指令地址通常会很快引发异常。

当与基数rs1 = x0一起使用时，JALR可用于从地址空间中的任何位置实现对最低2 KiB或最高2 KiB地址区域的单指令子例程调用，这可用于实现对小运行时的快速调用图书馆。

如果目标地址未与四字节边界对齐，则JAL和JALR指令将生成未对齐的指令获取异常。

在支持具有16位对齐指令的扩展的机器上，例如压缩指令集扩展C，不能执行指令取指错位异常。

返回地址预测栈是高性能指令获取单元的常见特性，但需要准确检测用于过程调用的指令并使返回有效。对于RISC-V，关于指令使用的提示通过使用的寄存器编号隐式编码。只有当rd = x1 / x5时，JAL指令才应将返回地址推送到返回地址堆栈（RAS）。JALR指令应按表/表2.1中所示推送/弹出RAS。

rd	*rs1*	*RS1 = 路*	RAS行动
！*链接* ！*链接* *链接* *链接* *链接*	！*链接* *链接* ！*链接* *链接* *链接*	- - - 0 1	没有 流行 推动 推和弹 推

表2.1：在指令中使用的寄存器说明符中编码的返回地址堆栈预测提示。在上面，当寄存器是x1或x5时，链接为真。

其他一些ISA为其间接跳转指令添加了显式提示位，以指导返回地址堆栈操作。我们使用与寄存器编号绑定的隐式提示和调用约定来减少用于这些提示的编码空间。

当两个不同的链接寄存器（x1和x5）作为rs1和rd给出时，RAS被推送和弹出以支持协同程序。如果rs1和rd是相同的链接寄存器（x1或x5），则仅推送RAS以启用序列的宏操作融合：lui ra, imm20;jalr ra, ra, imm12和auipc ra, imm20;jalr, im12,

条件分支

所有分支指令都使用B类指令格式。12位B-immediate以2的倍数对有符号偏移进行编码，并添加到当前pc以给出目标地址。条件分支范围是±4 KiB。

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	IMM［5］	rs2	rs1	功能3	imm[4:1]	imm[11]	操作码
1	6	5	5	3	4	1	7
偏移12,10：5]		错误预测	上述	贝克/BNE	偏移11,4：1]		科
偏移12,10：5]		错误预测	上述	BLT[U]	偏移11,4：1]		科
偏移12,10：5]		错误预测	上述	BGE[U]	偏移11,4：1]		科

分支指令比较两个寄存器。如果寄存器rs1和rs2分别相等或不相等，则BEQ和BNE取分支。如果rs1小于rs2，BLT和BLTU将使用分支，分别使用有符号和无符号比较。如果rs1大于或等于rs2，则BGE和BGEU分别使用有符号和无符号比较来获取分支。注意，BGT，BGTU，BLE和BLEU可以通过将操作数分别反转为BLT，BLTU，BGE和BGEU来合成。

可以使用单个BLTU指令检查带符号的数组边界，因为任何负指数都将比任何非负边界更大。

应对软件进行优化，使顺序代码路径成为最常见的路径，并将不常用的代码路径置于行外。软件还应该假设后向分支将被预测，而前向分支将被取消，至少在第一次遇到它们

时。动态预测器应该快速学习任何可预测的分支行为。

与其他一些体系结构不同，RISC-V跳转（带有rd = x0的JAL）指令应始终用于无条件分支，而不是具有始终为真条件的条件分支指令。RISC-V跳转也是PC相关的，并且支持比分支更宽的偏移范围，并且不会对条件分支预测表施加压力。

条件分支设计为包括两个寄存器之间的算术比较操作（也在PA-RISC和Xtensa ISA中完成），而不是使用条件代码（x86，ARM，SPARC，PowerPC），或仅将一个寄存器与零进行比较（ Alpha，MIPS），或两个仅用于相等的寄存器（MIPS）。这种设计的动机是观察到组合的比较和分支指令适合常规流水线，避免了额外的条件代码状态或使用临时寄存器，并减少了静态代码大小和动态指令获取流量。另一点是，与零的比较需要非平凡的电路延迟（特别是在高级过程中转移到静态逻辑之后），因此几乎与算术幅度比较一样昂贵。融合比较和分支指令的另一个优点是在前端指令流中较早地观察到分支，因此可以更早地预测。在基于相同条件代码可以采用多个分支的情况下，具有条件代码的设计可能具有优势，但是我们认为这种情况相对较少。

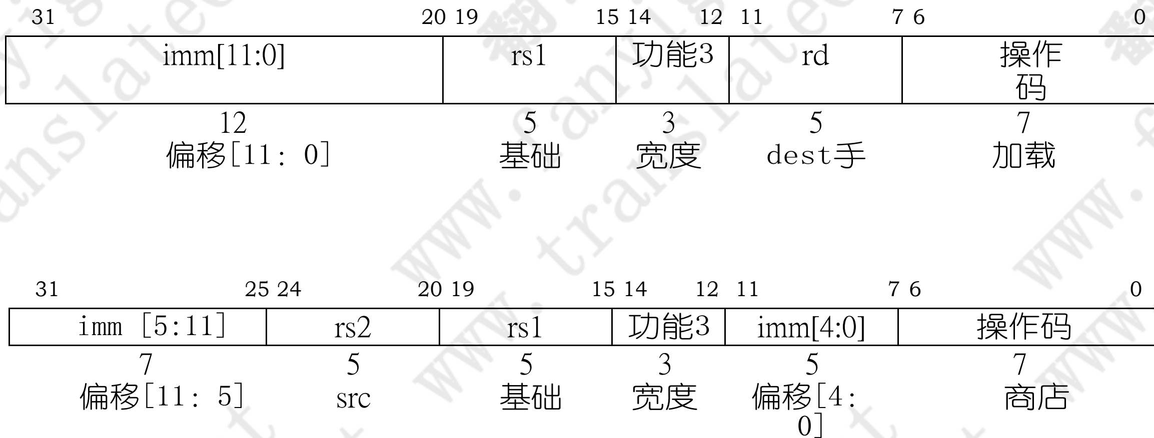
我们考虑但在指令编码中没有包含静态分支提示。这些可以减轻动态预测变量的压力，但需要更多的指令编码空间和软件分析以获得最佳结果，并且如果生产运行与分析运行不匹配，则可能导致性能不佳。

我们考虑但不包括条件移动或谓词指令，它们可以有效地替换不可预测的短前向分支。条件移动是两者中较简单的，但很难用于可能导致异常的条件代码（内存访问和浮点运算）。预测为系统添加额外的标志状态，设置和清除标志的附加指令以及每条指令的额外编码开销。条件移动和谓词指令都增加了无序微架构的复杂性，添加了隐式的第三源操作数，因为如果谓词为假，则需要将目标架构寄存器的原始值复制到重命名的目标物理寄存器中。此外，使用预测而不是分支的静态编译时决策可能导致编译器训练集中未包含的输入的性能降低，特别是考虑到不可预测的分支很少，并且随着分支预测技术的改进而变得越来越少。

我们注意到存在各种微体系结构技术来动态地将不可预测的短前向分支转换为内部预测代码，以避免在分支错误预测上冲洗管道的成本[13,17,16]并且已经在商业处理器中实现[27]。最简单的技术只是通过仅刷新分支阴影中的指令而不是整个获取流水线，或者通过使用宽指令获取或空闲指令获取时隙从两端获取指令来减少从错误预测的短前向分支中恢复的代价。用于无序内核的更复杂的技术在分支阴影中的指令上添加内部谓词，其中内部谓词值由分支指令写入，允许分支和后续指令以推测方式和无序方式执行。其他代码[27]。

2.6 加载和存储指令

RV32I是一种加载存储体系结构，其中只有加载和存储指令访问存储器，算术指令仅在CPU寄存器上运行。RV32I提供32位用户地址空间，该地址空间为字节寻址和小端。执行环境将定义地址空间的哪些部分合法访问。目标为x0的加载必须仍然引发任何异常并执行任何其他副作用，即使废弃了加载值。



加载和存储指令在寄存器和存储器之间传输值。载荷以I型格式编码，存储为S型。通过将寄存器rs1添加到符号扩展的12位偏移量来获得有效字节地址。加载从内存复制值到寄存器rd。存储将寄存器rs2中的值复制到内存中。

LW指令将32位值从内存加载到rd。LH从存储器加载一个16位值，然后在存储到rd之前符号扩展到32位。LHU从存储器加载一个16位值，然后在存储到rd之前将零扩展到32位。LB和LBU类似地定义为8位值。SW，SH和SB指令存储从寄存器rs2的低位到存储器的32位，16位和8位值。

为了获得最佳性能，所有加载和存储的有效地址应自然地针对每种数据类型进行对齐（即，对于32位访问使用4字节边界，对于16位访问使用2字节边界）。基本ISA支持未对齐的访问，但这些访问可能会非常缓慢地运行，具体取决于实现。此外，自然对齐的加载和存储保证以原子方式执行，而未对齐的加载和存储可能不会，因此需要额外的同步以确保原子性。

移植遗留代码时偶尔会需要未对齐的访问，并且在使用任何形式的压缩SIMD扩展时，对于许多应用程序的良好性能至关重要。我们通过常规加载和存储指令支持未对齐访问的基本原理是简化未对齐硬件支持的添加。一种选择是禁止基本ISA中的未对齐访问，然后为未对齐访问提供一些单独的ISA支持，或者是帮助软件处理未对齐访问的特殊指令，或者是用于未对齐访问的新硬件寻址模式。特殊指令难以使用，使ISA复杂化，并且经常添加新的处理器状态（例如，SPARC VIS对齐地址偏移寄存器）或使对现有处理器状态的访问变得复杂（例如，MIPS LWL / LWR部分寄存器写入）。此外，对于面向循环的打包SIMD代码，操作数未对齐时的额外开销会促使软件根据操作数对齐提供多种形式的循环，这会使代码生成复杂化并增加循环启动开销。新的未对准硬件寻址模式在指令编码中占用相当大的空间或者需要非常简化的寻址模式（例如，仅仅寄存器间接寻址）。

我们不要求未对齐访问的原子性，因此简单的实现只能使用机器陷阱和软件处理程序来处理部分或全部未对齐的访问。如果提供了硬件未对齐支持，则软件可以通过简单地使用常规加载和存储指令来利用它。然后，硬件可以根据运行时地址是否对齐自动优化访问。

2.7 记忆模型

由于RISC-V内存模型目前正在修订，以确保它可以有效地支持当前的编程语言内存模型，因此本节已过时。修改后的基本内存模型将包含进一步的排序约束，至少包括来自同一哈特的相同地址的加载不能被重新排序，并且遵守指令之间的语法数据依赖性。

基本RISC-V ISA在单个用户地址空间内支持多个并发执行线程。每个RISC-V硬件线程或哈特都有自己的用户寄存器状态和程序计数器，并执行独立的顺序指令流。执行环境将定义如何创建和管理RISC-V harts。RISC-V harts可以通过对执行环境的调用与其他harts进行通信和同步，这些hast在每个执行环境的规范中单独记录，或者直接通过共享内存系统。RISC-V harts还可以与I / O设备交互，并通过加载和存储间接地相互交互，分配给分配给I / O的地址空间部分。

我们使用术语hart来明确和简洁地描述硬件线程而不是软件管理的线程上下文。

在基础RISC-V ISA中，每个RISC-V hart都会观察自己的内存操作，就像它们按程序顺序执行一样。RISC-V在harts之间有一个宽松的内存模型，需要一个显式的FENCE指令来保证来自不同RISC-V harts的内存操作之间的顺序。第7章描述了可选的原子内存指令扩展“A”，它提供了额外的同步操作。

31 28	27	26	25	24	23	22	21	20	19 15	14 12	11 7	6 0
0	圆周率	人事军官	公共关系	PW	硅	所以	锶	西南部	rs1	功能3	rd	操作码
4	1	1	1	1	1	1	1	1	5	3	5	7
0	前任				接班人				0	围栏	0	MISC-MEM

FENCE指令用于命令设备I / O和存储器访问，如其他RISC-V harts和外部设备或协处理器所查看的那样。设备输入（I），设备输出（O），存储器读取（R）和存储器写入（W）的任何组合可以相对于它们的任何组合进行排序。非正式地，在FENCE之前设置的前一个操作中的任何操作之前，没有其他RISC-V或外部设备可以在FENCE之后观察后继集中的任何操作。执行环境将定义可能的I / O操作，特别是哪些加载和存储指令可以分别作为设备输入和设备输出操作处理和排序，而不是存储器读取和写入。例如，通常使用未缓存的加载和存储来访问存储器映射的I / O设备，这些加载和存储使用I和O位而不是R和W位进行排序。指令集扩展还可以描述新的协处理器I / O指令，这些指令也将使用FENCE中的I和O位进行排序。

FENCE指令中未使用的字段imm [11: 8]，rs1和rd，在未来的扩展中保留用于更细粒度的栅栏。为了向前兼容，基本实现应忽略这些字段，标准软件应将这些字段归零。

我们选择了一个宽松的内存模型来实现简单机器实现的高性能，但是完全放松的内存模型太弱而无法支持编程语言内存模型，因此内存模型正在收紧。

宽松的内存模型也可能与未来的协处理器或加速器扩展最兼容。我们将内存R / W排序中的I / O排序分开，以避免在设备驱动程序中进行不必要的序列化，并支持备用非内存路径来控制添加的协处理器或I / O设备。简单实现可以另外忽略前导和后继字段，并且总是在所有操作上执行保守的栅栏。

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	功能3	rd	操作码
12	5	3	5	7
0	0	我的篱笆。	0	MISC-MEM

FENCE.I指令用于同步指令和数据流。RISC-V不保证在执行FENCE.I指令之前，指令存储器的存储将对同一RISC-V hart上的指令读取可见。FENCE.I指令仅确保在RISC-V hart上的后续指令获取将看到任何先前的数据存储已经对同一RISC-V hart可见。FENCE.I不确保其他RISC-V harts的指令提取将观察多处理器系统中的本地hart存储。为了使存储到所有RISC-V harts可见的指令存储器，写入hart必须在请求所有远程RISC-V harts执行FENCE.I之前执行数据FENCE。

FENCE.I指令中未使用的字段imm［11：0］，rs1和rd，在未来的扩展中保留用于更细粒度的栅栏。为了向前兼容，基本实现应忽略这些字段，标准软件应将这些字段归零。

FENCE.I指令旨在支持各种实现。一个简单的实现可以在执行FENCE.I时刷新本地指令高速缓存和指令流水线。更复杂的实现可能会在每个数据（指令）高速缓存未命中时监听指令（数据）高速缓存，或者在由本地存储指令写入时使用包含的统一专用L2高速缓存来使来自主指令高速缓存的行无效。如果指令和数据高速缓存以这种方式保持一致，则只需要在FENCE.I处刷新管道。

我们考虑但没有包含“商店指令字”指令（如MAJC［30］）。JIT编译器可以在单个FENCE.I之前生成大量指令，并通过将已转换的指令写入已知不驻留在I-cache中的内存区域来分摊任何指令缓存侦听/无效开销。

2.8 控制和状态寄存器指令

SYSTEM指令用于访问可能需要特权访问的系统功能，并使用I类型指令格式进行编码。这些可分为两大类：原子读取 - 修改 - 写入控制和状态寄存器（CSR），以及所有其他可能特权的指令。本节将介绍CSR指令，以及下一节中介绍的另外两个用户级SYSTEM指令。

翻译狗 www.fanyigou.net www.translateGo.net

定义SYSTEM指令以允许更简单的实现始终陷阱到单个软件陷阱处理程序。更复杂的实现可能会在硬件中执行更多的每个系统指令。

CSR指令

我们在这里定义了完整的CSR指令集，尽管在标准的用户级基础ISA中，只有少数只读的计数器CSR可以访问。

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
企业社会责任	rs1	功能3	rd	操作码
12	5	3	5	7
源/目标	资源	ccrw	dest手	系统
源/目标	资源	CSRRS	dest手	系统
源/目标	资源	中国证监会	dest手	系统
源/目标	uimm[4:0]	CSRRWI	dest手	系统
源/目标	uimm[4:0]	CSRRSI	dest手	系统
源/目标	uimm[4:0]	CSRRCI	dest手	系统

CSRRW（原子读/写CSR）指令以原子方式交换CSR和整数寄存器中的值。CSRRW读取CSR的旧值，将值零扩展为XLEN位，然后将其写入整数寄存器rd。rs1中的初始值写入CSR。如果rd = x0，则指令不应读取CSR，并且不应引起CSR读取时可能发生的任何副作用。

CSRRS（CSR中的原子读取和设置位）指令读取CSR的值，将值零扩展为XLEN位，并将其写入整数寄存器rd。整数寄存器rs1中的初始值被视为位掩码，指定要在CSR中设置的位位置。如果CSR位是可写的，则rs1中的任何位都将导致在CSR中设置相应的位。CSR中的其他位不受影响（尽管CSR在写入时可能会产生副作用）。

CSRRC（CSR中的原子读取和清除位）指令读取CSR的值，将值零扩展为XLEN位，并将其写入整数寄存器rd。整数寄存器rs1中的初始值被视为位掩码，指定要在CSR中清除的位位置。如果CSR位是可写的，则rs1中的任何位都将导致在CSR中清除相应的位。CSR中的其他位不受影响。

对于CSRRS和CSRRC，如果rs1 = x0，则该指令根本不会写入CSR，因此不会导致CSR写入可能产生的任何副作用，例如在访问时引发非法指令异常只读CSR。请注意，如果rs1指定的寄存器保持x0以外的零值，则该指令仍将尝试将未修改的值写回CSR，并将导致任何附带的副作用。

CSRRWI，CSRRSI和CSRRCI变体分别类似于CSRRW，CSRRS和CSRRC，除了它们使用通过零扩展5位无符号立即数（uimm [4：0]）字段获得的XLEN位值更新CSR在rs1字段中编码而不是整数值

寄存器。对于CSRRSI和CSRRCI，如果uimm [4：0]字段为零，则这些指令不会写入CSR，也不会导致CSR写入时可能出现的任何副作用。对于CSRRWI，如果rd = x0，则指令不应读取CSR，也不应引起CSR读取时可能出现的任何副作用。

一些CSR，例如指令退休计数器，instret，可以被修改为指令执行的副作用。在这些情况下，如果CSR访问指令读取CSR，则它在执行指令之前读取值。如果CSR访问指令写入CSR，则在执行指令之后进行更新。特别地，由一条指令写入instret的值将是由以下指令读取的值（即，在写入新值之前由第一条指令退休引起的instret的增量）。

用于读取CSR的汇编器伪指令CSRR rd，csr被编码为CSRRS rd，csr，x0。用于编写CSR的汇编伪指令CSRW csr rs1编码为CSRRW x0，csr，rs1，而CSRWI csr，uimm编码为CSRRWI x0，csr，uimm。

当不需要旧值时，定义更多汇编器伪指令以设置和清除CSR中的位：CSRS / CSRC csr，rs1;CSRSI / CSRCI csr，uimm。

计时器和计数器

31 20	19 15	14 12	11 7	6 0
企业社会责任	rs1	功能3	rd	操作码
12	5	3	5	7
RDCCLE [H]	0	CSRRS	dest手	系统
RDTIME[H]	0	CSRRS	dest手	系统
RDINSTRET[H]	0	CSRRS	dest手	系统

RV32I提供了许多64位只读用户级计数器，这些计数器映射到12位CSR地址空间，并使用CSRRS指令以32位进行访问。

RDCYCLE伪指令读取循环CSR的低XLEN位，其保持从过去的任意开始时间运行哈特的处理器核执行的时钟周期数的计数。RDCYCLEH是仅用于RV32I的指令，它读取同一周期计数器的第63-32位。底层的64位计数器在实践中永远不会溢出。循环计数器的进展速度取决于实施和运行环境。执行环境应该提供一种确定循环计数器递增的当前速率（周期/秒）的方法。

RDTIME伪指令读取CSR时间的低XLEN位，它计算过去任意开始时间过去的挂钟实时。RDTIMEH是仅RV32I指令，读取同一实时计数器的第63-32位。底层的64位计数器在实践中永远不会溢出。执行环境应该提供一种确定实时计数器周期（秒/刻度）的方法。期间必须是恒定的。单个用户应用程序中所有harts的实时时钟应该在实时时钟的一个时钟内同步。环境应该提供一种确定时钟准确性的方法。

RDINSTRET伪指令读取instret CSR的低XLEN位，这些位计数

这个哈特从过去任意一个起点退出的指令数量。RDINSTRETH是一条仅RV32I指令，读取同一指令计数器的第63-32位。底层的64位计数器，在实践中永远不会溢出。

以下代码序列将有效的64位周期计数器值读入x3：x2，即使计数器在读取其上半部分和下半部分之间溢出。

```
再次：
    雷西莱      x3
    rdcyle      x2
    雷西莱      x4
    本          x3, x4, 再次
```

图2.5：用于读取RV32中64位周期计数器的示例代码。

我们要求在所有实现中提供这些基本计数器，因为它们对于基本性能分析，自适应和动态优化至关重要，并允许应用程序使用实时流。应提供附加计数器以帮助诊断性能问题，这些应该可以从用户级应用程序代码中以低开销访问。我们要求计数器为64位宽，即使在RV32上，否则软件很难确定值是否溢出。对于低端实现，每个计数器的高32位可以使用由低32位溢出触发的陷阱处理程序递增的软件计数器来实现。上面描述的示例代码显示了如何完整可以使用单独的32位指令安全地读取64位宽度值。

在某些应用程序中，能够在同一时刻读取多个计数器非常重要。在多任务环境下运行时，用户线程在尝试读取计数器时可能会遇到上下文切换。一种解决方案是用户线程在读取其他计数器之前和之后读取实时计数器以确定是否在序列的中间发生了上下文切换，在这种情况下可以重试读取。我们考虑添加输出锁存器以允许用户线程以原子方式对计数器值进行快照，但这会增加用户上下文的大小，尤其是对于具有更丰富计数器集的实现。

2.9 环境呼叫和断点

31–20	19–15	14–12	11–7	6–0
功能12	rs1	功能3	rd	操作码
12	5	3	5	7
ECALL	0	证人	0	系统
ebak	0	证人	0	系统

ECALL指令用于向支持执行环境发出请求，该环境通常是操作系统。系统的ABI将定义如何传递环境请求的参数，但通常这些参数将位于整数寄存器文件中的已定义位置。

调试器使用EBREAK指令将控制权转移回调试环境。

ECALL和EBREAK之前被命名为SCALL和SBREAK。说明书有

相同的功能和编码，但重命名以反映它们可以比调用管理程序级操作系统或调试程序更常用。

翻译狗 www.fanyigou.net www.translateGo.net

第3章

RV32E基本整数指令集，版本1.9

本章介绍RV32E基本整数指令集，它是为嵌入式系统设计的RV32I的简化版本。主要的变化是将整数寄存器的数量减少到16，并删除RV32I中必需的计数器。本章仅概述RV32E和RV32I之间的差异，因此应在第2章后阅读。

RV32E旨在为嵌入式微控制器提供更小的基础内核。虽然我们在本文档的2.0版本中提到了这种可能性，但我们最初拒绝定义此子集。然而，鉴于对最小可能的32位微控制器的需求，以及为了在这个空间中抢占碎片，我们现在已经将RV32E定义为除RV32I，RV64I和RV128I之外的第四个标准基础ISA。E变体仅针对32位地址空间宽度进行标准化。

3.1 RV32E程序员模型

RV32E将整数寄存器计数减少到16个通用寄存器（x0-x15），其中x0是专用零寄存器。

我们发现，在小型RV32I内核设计中，上面的16个寄存器占用了核心总面积的四分之一（不包括存储器），因此它们的移除节省了大约25%的核心区域，相应的核心功耗降低。

此更改需要不同的调用约定和ABI。特别是，RV32E仅用于软浮点调用约定。具有硬件浮点的系统必须使用I base。

3.2 RV32E指令集

RV32E使用与RV32I相同的指令集编码，但使用寄存器说明符除外
指令中的x16-x31将导致引发非法指令异常。

任何未来的标准扩展都不会使用由简化的寄存器指定字段释放的指令位，因此这些扩展可用于非标准扩展。

进一步的简化是计数器指令（rdcycle [h]，rdtime [h]，rdinstret [h]）不再是强制性的。

强制计数器需要额外的寄存器和逻辑，并且可以用更多特定于应用程序的工具替换。

3.3 RV32E扩展

RV32E可以使用M，A和C用户级标准扩展进行扩展。

我们不打算使用RV32E子集支持硬件浮点。在硬件浮点单元的情况下，减少寄存器计数所节省的成本可以忽略不计，我们希望减少ABI的扩散。

RV32E系统的特权体系结构可以包括用户模式以及机器模式，以及第II卷中描述的物理内存保护方案。

我们不打算使用RV32E子集支持完整的Unix风格的操作系统。在支持OS的核心环境中，减少寄存器数量所节省的成本可以忽略不计，我们希望避免操作系统碎片化。

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，它基于第2章中描述的RV32I变体。本章仅介绍与RV32I的不同之处，因此应与前一章一起阅读。

4.1 登记国

RV64I将整数寄存器和支持的用户地址空间扩展为64位（图2.1中的XLEN = 64）。

4.2 整数计算指令

提供了附加的指令变体来操作RV64I中的32位值，由操作码的“W”后缀表示。这些“* W”指令忽略其输入的高32位，并始终产生32位有符号值，即位XLEN-1到31相等。它们在RV32I中导致非法指令异常。

编译器和调用约定保持一个不变量，即所有32位值都以64位寄存器中的符号扩展格式保存。甚至32位无符号整数也将位31扩展为位63到32. 因此，无符号和有符号32位整数之间的转换是无操作，从有符号的32位整数到带符号的64位整数的转换也是如此。现有的64位宽SLTU和无符号分支比较在此不变量下仍然可以在无符号32位整数上正确运行。类似地，32位符号扩展整数上的现有64位宽逻辑运算保留了符号扩展属性。添加和移位需要一些新指令（ADD [I] W / SUBW / SxxW）以确保32位值的合理性能。

翻译狗 www.fanyigou.net www.translateGo.net

整数寄存器 - 即时指令

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	功能3	rd	操作码
12	5	3	5	7
I-即时[11：0]	src	ADDIW	dest手	OP-IMM-32

ADDIW是一条仅用于RV64I的指令，它将符号扩展的12位立即数添加到寄存器rs1，并在rd中生成32位结果的正确符号扩展。忽略溢出，结果是结果符号的低32位扩展到64位。注意，ADDIW rd，rs1,0将寄存器rs1的低32位的符号扩展写入寄存器rd（汇编器伪操作SEXT.W）。

31 26	25	24 20	19 15	14 12	11 7	6 0
IMM［6］	imm[5]	imm[4:0]	rs1	功能3	rd	操作码
6	1	5	5	3	5	7
000000	萨拉姆［5］	shamt[4:0]	src	SLLI	dest手	OP-IMM
000000	萨拉姆［5］	shamt[4:0]	src	SRLI	dest手	OP-IMM
010000	萨拉姆［5］	shamt[4:0]	src	rias	dest手	OP-IMM
000000	0	shamt[4:0]	src	SLLIW	dest手	OP-IMM-32
000000	0	shamt[4:0]	src	SRLIW	dest手	OP-IMM-32
010000	0	shamt[4:0]	src	SRAIW	dest手	OP-IMM-32

使用与RV32I相同的指令操作码将常数移位编码为I类型格式的特化。要移位的操作数在rs1中，移位量在RV64I的I-immediate字段的低6位中编码。右移位类型在位30中编码.SLLI是逻辑左移位（零移位到低位）;SRLI是一个逻辑右移（零被移入高位）;SRAI是算术右移（原始符号位被复制到空出的高位）。对于RV32I，如果imm［5］/ = 0，则SLLI，SRLI和SRAI会生成非法指令异常。

SLLIW，SRLIW和SRAIW是仅限RV64I的指令，它们是类似定义的，但在32位值上运行并产生带符号的32位结果。如果imm［5］/ = 0，SLLIW，SRLIW和SRAIW会生成非法指令异常。

31 12	11 7	6 0
imm[31:12]	rd	操作码
20	5	7
U型即时[31:12]	dest手	他
U型即时[31:12]	dest手	AUIPC

LUI（load upper immediate）使用与RV32I相同的操作码。LUI将20位U-immediate置于寄存器rd的31-12位，并将0置于最低12位。32位结果符号扩展为64位。

AUIPC（将up immediate添加到pc）使用与RV32I相同的操作码。AUIPC（将上部立即添加到pc）用于构建pc相对地址并使用U型格式。AUIPC附12个低 -

将0位命令到20位U-immediate，将结果符号扩展为64位，然后将其添加到pc
并将结果放在寄存器rd中。

整数寄存器 - 寄存器操作

31 25	24 20	19 15	14 12	11 7	6 0
功能7	rs2	rs1	功能3	rd	操作码
7	5	5	3	5	7
0000000	错误预测	上述	SLL /公司	dest手	操作
0100000	错误预测	上述	SRA	dest手	操作
0000000	错误预测	上述	ADDW	dest手	on-32
0000000	错误预测	上述	SLLW/SRLW	dest手	on-32
0100000	错误预测	上述	SUBW /近程突击武器	dest手	on-32

ADDW和SUBW是仅RV64I指令，其定义类似于ADD和SUB，但在32位值上运行并产生带符号的32位结果。忽略溢出，结果的低32位符号扩展为64位并写入目标寄存器。

SLL，SRL和SRA通过寄存器rs2中保持的移位量对寄存器rs1中的值执行逻辑左，逻辑右和算术右移。在RV64I中，只考虑rs2的低6位用于移位量。

SLLW，SRLW和SRAW是仅限RV64I的指令，它们是类似定义的，但在32位值上运行并产生带符号的32位结果。移位量由rs2 [4: 0]给出。

4.3 加载和存储指令

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合法访问。

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	功能3	rd	操作码
12	5	3	5	7
偏移[11: 0]	基础	宽度	dest手	加载

31 25	24 20	19 15	14 12	11 7	6 0
imm [5:11]	rs2	rs1	功能3	imm[4:0]	操作码
7	5	5	3	5	7
偏移[11: 5]	src	基础	宽度	偏移[4: 0]	商店

LD指令将64位值从存储器加载到RV64I的寄存器rd中。

LW指令从存储器加载32位值，并将其符号扩展为64位，然后将其存储在RV64I的寄存器rd中。另一方面，LWU指令对32位值进行零扩展

来自RV64I的内存。LH和LHU类似地定义为16位值，对于8位值也是LB和LBU。SD，SW，SH和SB指令分别存储从寄存器rs2的低位到存储器的64位，32位，16位和8位值。

4.4 系统说明

在RV64I中，CSR指令可以操作64位CSR。特别是，RDCYCLE，RD-TIME和RDINSTRET伪指令读取循环，时间和instret计数器的完整64位。因此，RDCYCLEH，RDTIMEH和RDINSTRETH指令不是必需的，在RV64I中是非法的。

翻译狗 www.fanyigou.net www.translateGo.net

第5章

RV128I基本整数指令集，版本1.7

“只有一个错误可以在计算机设计中难以恢复 – 没有足够的地址位用于内存寻址和内存管理。” Bell和Strecker, ISCA-3,1976。

本章介绍RV128I，它是RISC-V ISA的一种变体，支持扁平的128位地址空间。该变体是对现有RV32I和RV64I设计的直接推断。

扩展整数寄存器宽度的主要原因是支持更大的地址空间。目前尚不清楚何时需要大于64位的扁平地址空间。在撰写本文时，通过Top500基准测量的世界上最快的超级计算机拥有超过1 PB的DRAM，如果所有DRAM都位于单个地址空间中，则需要超过50位的地址空间。一些仓库规模的计算机已经包含更多的DRAM，新的密集固态非易失性存储器和快速互连技术可能需要更大的存储空间。Exascale系统研究的目标是100个PB存储系统，占用57位地址空间。按照历史增长率，2030年之前可能需要大于64位的地址空间。

历史表明，每当需要超过64位的地址空间时，架构师将重复关于扩展地址空间的替代方案的激烈辩论，包括分段，96位地址空间和软件解决方案，直到最后，平坦128 – 位地址空间将被采用作为最简单和最佳的解决方案。

我们目前尚未冻结RV128规范，因为可能需要根据128位地址空间的实际使用情况来改进设计。

RV128I建立在RV64I之上，与RV64I构建在RV32I上的方式相同，整数寄存器扩展到128位（即XLEN = 128）。大多数整数计算指令不变，因为它们被定义为在XLEN位上操作。保留寄存器低位32位值的RV64I “* W” 整数指令，以及一组新的“* D”整数指令，这些指令操作64位值保存在低位添加了128位整数寄存器。“* D”指令使用标准32位编码中的两个主要操作码（OP-IMM-64和OP-64）。

立即（SLLI / SRLI / SRAI）的移位现在使用I-immediate的低7位进行编码，可变移位（SLL / SRL / SRA）使用移位量源寄存器的低7位。

翻译狗 www.fanyigou.net www.translateGo.net

使用现有的LOAD主要操作码添加LDU（加载双无符号）指令，以及用于加载和存储四字值的新LQ和SQ指令。SQ被添加到STORE主要操作码，而LQ被添加到MISC-MEM主要操作码。

浮点指令集保持不变，尽管128位Q浮点扩展现在可以支持FMV.XQ和FMV.QX指令，以及来自T（128位）整数格式的附加FCVT指令。

第6章

“M”标准扩展为整数乘法和除法，版本 2.0

本章描述了标准整数乘法和除法指令扩展，它被命名为“M”，包含对两个整数寄存器中保存的值进行乘法或除法的指令。

我们将整数乘法与基数分开以简化低端实现，或者对于整数乘法和除法运算在连接的加速器中不频繁或更好处理的应用程序。

6.1 乘法运算

31 25	24 20	19 15	14 12	11 7	6 0
功能7	rs2	rs1	功能3	rd	操作码
7	5	5	3	5	7
mudiv	乘数	被乘数	MUL / MULH [[S] U]	dest手	操作
mudiv	乘数	被乘数	穆洛	dest手	on-32

MUL执行XLEN位×XLEN位乘法，并将较低的XLEN位置于目标寄存器中。MULH，MULHU和MULHSU执行相同的乘法运算，但返回完整2×XLEN位乘积的上XLEN位，分别用于有符号×有符号，无符号×无符号和有符号×无符号乘法。如果同一产品的高位和低位都需要，则建议的代码序列为：MULH [[S] U] rdh，rs1，rs2;MUL rdl，rs1，rs2（源寄存器说明符必须处于相同的顺序且rdh不能与rs1或rs2相同）。然后，微体系结构可以将这些融合到单个乘法运算中，而不是执行两个单独的乘法运算。

MULW仅对RV64有效，并将源寄存器的低32位相乘，将结果的低32位的符号扩展名放入目标寄存器。MUL可以用来

获取64位乘积的高32位，但有符号参数必须是正确的32位有符号值，而无符号参数必须清除其高32位。

6.2 分部运作

31 25	24 20	19 15	14 12	11 7	6 0
功能7	rs2	rs1	功能3	rd	操作码
7	5	5	3	5	7
mudiv	除数	股利	DIV[U]/REM[U]	dest手	操作
mudiv	除数	股利	DIV [U] W / REM [U]W	dest手	on-32

DIV和DIVU通过XLEN位执行XLEN位的有符号和无符号整数除法。REM和REMU提供相应除法运算的其余部分。如果同一分区需要商和余数，则推荐的代码序列为：DIV [U] rdq, rs1, rs2;REM [U] rdr, rs1, rs2 (rdq不能与rs1或rs2相同）。然后，微体系结构可以将这些融合到单个除法运算中，而不是执行两个单独的除法。

表6.1总结了除零和除法溢出的语义。除零的商具有所有位设置，即2^{XLEN} - 1表示无符号除法，或-1表示有符号除法。除以零的余数等于股息。仅当最负的整数-2^{XLEN-1}除以-1时，才会发生有符号除法溢出。有符号除法溢出的商等于被除数，余数为零。无法进行无符号除法溢出。

条件	股利	除数	两	热木	DIV	REM
被零除	x	0	$2^{XLEN}-1$	x	−1	x
溢出（仅限签名）	-2^{XLEN-1}	−1	-	-	-2^{XLEN-1}	0

表6.1：除零和除法溢出的语义。

我们考虑将整数除以异常提高零，这些异常会导致大多数执行环境中的陷阱。但是，这将是标准ISA中唯一的算术陷阱（浮点异常设置标志和写入默认值，但不会导致陷阱），并且需要语言实现者与执行环境的陷阱处理程序进行交互。此外，在语言标准要求零除异常必须导致立即控制流更改的情况下，只需要将单个分支指令添加到每个除法运算，并且该分支指令可以在除法之后插入并且通常应该是非常可预见地没有采用，增加了很少的运行时开销。

无符号和有符号除以零的所有位的值都被返回，以简化分频器电路。所有1的值都是无符号除法返回的自然值，表示最大的无符号数，也是简单无符号除法实现的自然结果。有符号除法通常使用无符号除法电路实现，并指定相同的溢出结果可简化硬件。

DIVW和DIVUW指令仅对RV64有效，并将rs1的低32位除以rs2的低32位，将它们分别视为有符号和无符号整数，将32位商置于rd，符号扩展为64位。REMW和REMUW指令仅对RV64有效，并分别提供相应的有符号和无符号余数运算。

REMW和REMUW都将32位结果符号扩展为64位，包括除以零。

翻译狗 www.fanyigou.net www.translateGo.net

第七章

“A”原子指令的标准扩展，版本2.0

由于RISC-V内存模型目前正在修订，以确保它可以有效地支持当前的编程语言内存模型，因此本节有些过时了。修改后的基本内存模型将包含进一步的排序约束，至少包括来自同一哈特的相同地址的加载不能被重新排序，并且遵守指令之间的语法数据依赖性。

标准原子指令扩展由指令子集名称“A”表示，并包含原子读取 - 修改 - 写入存储器的指令，以支持在同一存储器空间中运行的多个RISC-V harts之间的同步。提供的两种形式的原子指令是加载保留/存储条件指令和原子提取和操作存储器指令。两种类型的原子指令都支持各种内存一致性排序，包括无序，获取，释放和顺序一致的语义。这些指令允许RISC-V支持RCsc内存一致性模型[10]。

经过多次辩论后，语言社区和架构社区似乎终于将发布一致性作为标准内存一致性模型，因此围绕此模型构建了RISC-V原子支持。

7.1 指定原子指令的排序

基础RISC-V ISA具有宽松的内存模型，FENCE指令用于强加额外的排序约束。地址空间由执行环境划分为内存和I / O域，FENCE指令提供了对这两个地址域中的一个或两个的访问顺序的选项。

为了提供更高效的发布一致性支持[10]，每个原子指令都有两个位，aq和rl，用于指定其他RISC-V harts所看到的额外内存排序约束。位顺序访问两个地址域之一，内存或I / O，具体取决于原子指令正在访问的地址域。对其他域的访问不暗示排序约束，并且应该使用FENCE指令在两个域之间进行排序。

如果两个位都清零，则不会对原子存储器操作施加额外的排序约束。如果仅设置了aq位，则原子存储器操作被视为获取访问，即，在获取存储器操作之前，不能观察到此RISC-V上的后续存储器操作。如果仅设置了rl位，则原子存储器操作被视为释放访问，即，在此RISC-V之前的任何早期存储器操作之前，不能观察到释放存储器操作。如果设置了aq和rl位，则原子存储器操作是顺序一致的，并且不能在任何先前的存储器操作之前或在同一RISC-V中的任何后续存储器操作之后观察到，并且只能由任何其他操作观察到hart以相同的全局顺序将所有顺序一致的原子内存操作发送到同一地址域。

从理论上讲，aq和rl位的定义允许在没有全局存储原子性的情况下实现。但是，当设置了aq和rl位时，我们需要原子操作的完全顺序一致性，这意味着除了获取和释放语义之外还有全局存储原子性。实际上，硬件系统通常用全局存储原子性来实现，体现在本地处理器排序规则以及单写入器高速缓存一致性协议中。

7.2 加载保留/存储条件指令

31 27	26	25	24 20	19 15	14 12	11 7	6 0
功能5	水性	rl	rs2	rs1	功能3	rd	操作码
5	1	1	5	5	3	5	7
LR	排序		0	地址	宽度	destdest	我爱你
联合国安全理事会	排序		src	地址	宽度	destdest	我爱你

使用加载保留（LR）和存储条件（SC）指令执行对单个存储器字的复杂原子存储器操作。LR从rs1中的地址加载一个字，将符号扩展值放在rd中，并在内存地址上注册一个预留。SC将rs2中的一个字写入rs1中的地址，前提是该地址仍然存在有效的保留。SC在成功时写入0到rd，在失败时写入非零代码。

比较和交换（CAS）和LR / SC都可用于构建无锁数据结构。经过广泛讨论后，我们选择了LR / SC有以下几个原因：1）CAS遇到了ABA问题，LR / SC避免了这个问题，因为它监视对地址的所有访问，而不是只检查数据值的变化;2）CAS还需要一个新的整数指令格式来支持三个源操作数（地址，比较值，交换值）以及不同的存储系统消息格式，这会使微体系结构复杂化;3）此外，为了避免ABA问题，其他系统提供双宽CAS（DW-CAS）以允许计数器与数据字一起被测试和递增。这需要读取五个寄存器并在一个指令中写入两个，以及一个新的更大的存储器系统消息类型，这进一步使实现复杂化;4）LR / SC提供了更高效的多个原语的实现，因为它只需要一个负载而不是两个带CAS（在CAS指令之前一个负载获得推测计算的值，然后第二个负载作为CAS指令的一部分）在更新之前检查值是否保持不变）。

LR / SC优于CAS的主要缺点是活锁，我们通过如下所述的最终前进进度的架构保证来避免。另一个问题是当前x86架构及其DW-CAS的影响是否会使同步移植变得复杂

假设DW-CAS的库和其他软件是基本的机器原语。一个可能的缓解因素是最近向x86添加了事务性内存指令，这可能导致远离DW-CAS。

值1的故障代码保留用于编码未指定的故障。此时保留其他故障代码，便携式软件应仅假设故障代码不为零。LR和SC在内存中以自然对齐的64位（仅RV64）或32位字运行。未对齐的地址将生成未对齐的地址异常。

我们保留失败代码1表示“未指定”，以便简单实现可以使用SLT / SLTU指令所需的现有mux返回此值。在ISA的未来版本或扩展中可能会定义更具体的故障代码。

在标准A扩展中，某些受约束的LR / SC序列最终保证成功。LR / SC序列的静态代码加上在发生故障时重试序列的代码必须包含至少16个顺序放置在存储器中的整数指令。为了保证序列最终成功，LR和SC指令之间执行的动态代码只能包含来自基本“I”子集的其他指令，不包括加载，存储，向后跳转或后向分支，FENCE，FENCE.I和SYSTEM指令。重试失败的LR / SC序列的代码可以包含反向跳转和/或分支以重复LR / SC序列，但是具有相同的约束。SC必须与最新LR执行的地址相同。不满足这些约束的LR / SC序列可能会在某些实现的某些尝试上完成，但无法保证最终成功。

CAS的一个优点是它保证了一些hart最终取得进展，而LR / SC原子序列可以在某些系统上无限期地锁定。为了避免这种担忧，我们为LR / SC原子序列添加了前向进展的架构保证。对LR / SC序列内容的限制允许实现捕获LR上的高速缓存行并通过在有限的短时间内保持远程高速缓存干预来完成LR / SC序列。中断和TLB未命中可能导致保留丢失，但最终原子序列可以完成。我们限制LR / SC序列的长度以适应基本ISA中的64个连续指令字节，以避免对指令高速缓存和TLB大小和关联性的不适当限制。同样，我们不允许序列中的其他加载和存储，以避免限制数据缓存关联性。对分支和跳转的限制限制了可以在序列中花费的时间。不允许浮点运算和整数乘法/除法，以简化操作系统在缺乏适当硬件支持的实现上对这些指令的仿真。

实现可以在每个LR上保留存储器空间的任意子集，并且对于单个哈特，多个LR预留可以同时是活动的。如果在SC和本哈特中的最后一个LR之间没有发现从其他地址到该地址的访问以保留该地址，则SC可以成功。请注意，此LR可能具有不同的地址参数，但保留SC的地址作为内存子集的一部分。遵循此模型，在具有内存转换的系统中，如果较早的LR使用具有不同虚拟地址的别名保留相同位置，则允许SC成功，但如果虚拟地址不同，则也允许SC失败。如果存在从另一个hart到该地址的可观察的存储器访问，或者如果在该hart上存在中间上下文切换，或者如果在此期间hart执行了特权异常返回指令，则SC必须失败。

该规范明确允许实现支持具有更广泛保证的更强大的实现，前提是它们不会使受约束序列的原子性保证无效。

LR / SC可用于构造无锁数据结构。使用LR / SC实现比较和交换功能的示例如图7.1所示。如果内联，比较和交换功能只需要三个指令。

```
    #a0保存内存位置的地址
    #a1保持预期值
    #a2保持所需的值
    #a0保存返回值，如果成功则保持为0，否则为0：
否则为：
    LR。W TO（AO）    #加载原始值。b0，
    a1，失败        #不匹配，所以失败。
    sc.w AO，A2（AO）#尝试更新。
    吉拉             #返回。
失败：
    李AO、1          #设置返回失败。
    吉拉             #返回。
```

图7.1：使用LR / SC进行比较和交换功能的示例代码。

在紧接在前的LR之前，另一个RISC-V hart永远不会观察到SC指令。由于LR / SC序列的原子性质，在LR和成功的SC之间没有观察到任何来自任何hart的存储器操作。通过在SC指令上设置aq位，可以为LR / SC序列提供获取语义。通过设置LR指令上的rl位，可以给LR / SC序列释放语义。在LR指令上设置aq和rl位，并在SC指令上设置aq位使得LR / SC序列与其他顺序一致的原子操作顺序一致。

如果在LR和SC上都没有设置位，则可以观察到LR / SC序列在来自相同RISC-V哈特的周围存储器操作之前或之后发生。当LR / SC序列用于实现并行缩减操作时，这是适当的。

一般来说，一个多字原子基元是可取的，但是关于这应该采用什么形式仍然存在相当大的争议，并且保证向前进展增加了系统的复杂性。我们目前的想法是在原始事务存储器提议的行中包括一个小的有限容量事务存储缓冲器作为可选的标准扩展“T”。

7.3 原子内存操作

31 27	26	25	24 20	19 15	14 12	11 7	6 0
功能5	水性	rl	rs2	rs1	功能3	rd	操作码
5	1	1	5	5	3	5	7
amoswap。瓦德	排序		src	地址	宽度	destdest	我爱你
amoadd。瓦德	排序		src	地址	宽度	destdest	我爱你
美国。瓦德	排序		src	地址	宽度	destdest	我爱你
部长们瓦德	排序		src	地址	宽度	destdest	我爱你
amoxor。瓦德	排序		src	地址	宽度	destdest	我爱你
AMOMAX [U]。瓦德	排序		src	地址	宽度	destdest	我爱你
AMOMIN [U]。瓦德	排序		src	地址	宽度	destdest	我爱你

原子存储器操作（AMO）指令对多处理器同步执行读 - 修改 - 写操作，并用R类指令格式编码。这些AMO指令以原子方式从rs1中的地址加载数据值，将值放入寄存器rd，将二进制运算符应用于加载的值和rs2中的原始值，然后将结果存储回rs1中的地址。AMO可以在64位（仅RV64）或内存中的32位字上运行。对于RV64,32位AMO始终对rd中的值进行符号扩展。rs1中保存的地址必须与操作数的大小自然对齐（即，对于64位字对齐8字节，对32位字对齐4字节）。如果地址不是自然对齐的，则会生成未对齐的地址异常。

支持的操作包括交换，整数添加，逻辑AND，逻辑OR，逻辑XOR以及有符号和无符号整数最大值和最小值。在没有排序约束的情况下，这些AMO可用于实现并行缩减操作，其中通常通过写入x0来丢弃返回值。

我们提供了fetch-and-op样式的原子基元，因为它们比LR / SC或CAS更好地扩展到高度并行的系统。一个简单的微体系结构可以使用LR / SC原语实现AMO。更复杂的实现也可以在内存控制器上实现AMO，并且可以在目标为x0时优化远程获取原始值。

选择这组AMO来支持C11 / C ++ 11原子存储器操作效率
有利地，并且还支持并行减少内存。AMO的另一个用途是为I / O空间中的存储器映射设备寄存器（例如，设置，清除或切换位）提供原子更新。

为了帮助实现多处理器同步，AMO可选择提供发布一致性语义。如果设置了aq位，则在AMO之前不能观察到此RISC-V中的后续存储器操作。相反，如果设置了rl位，那么在此RISC-V中，在AMO之前的存储器访问之前，其他RISC-V harts将不会观察到AMO。

AMO旨在有效地实现C11和C ++ 11内存模型。尽管FENCE R，RW指令足以实现获取操作和FENCE RW，W足以实现释放，但与具有相应的aq或rl位集的AMO相比，这两者都意味着额外的不必要的排序。

图7.2显示了由测试和设置的自旋锁保护的临界区的示例代码序列。请注意，第一个AMO标记为aq以在关键部分之前命令锁定获取，第二个AMO标记为rl以在锁定放弃之前命令关键部分。

```
    里          t0, 1         #初始化交换值。再
次:
    amoswap.w.aq t0, t0,  (a0) #尝试获取锁定。
    电磁电压太低             t0, 再次    #重试
    如果举行。
    # ...
    #关键部分。
    # ...
    amoswap.w.rl x0, x0,  (a0) #通过存储0释放锁定。
```

图7.2: 互斥的示例代码。a0包含锁的地址。

我们建议使用上面显示的AMO Swap习惯用于锁定获取和释放，以简化推测性锁定省略的实现[25]。

存在使原子操作的实现复杂化的风险，如果锁定值与交换值匹配，则微体系结构可以在获取交换内消除存储，以避免弄脏处于共享或独占清除状态的高速缓存行。效果类似于测试和测试和设置锁定，但代码路径更短。

“A”扩展中的指令也可用于提供顺序一致的加载和存储。可以将顺序一致的负载实现为具有aq和rl集合的LR。顺序一致的存储可以实现为AMOSWAP，它将旧值写入x0并且同时设置aq和rl。

第8章

单精度浮点的“F”标准扩展，版本 2.0

本章描述了单精度浮点的标准指令集扩展，它被命名为“F”，并添加了符合IEEE 754-2008算法标准的单精度浮点计算指令[14]。

8.1 F注册州

F扩展添加了32个浮点寄存器f0-f31，每个32位宽，以及浮点控制和状态寄存器fcsr，它包含浮点单元的操作模式和异常状态。这种附加状态如图8.1所示。我们使用术语FLEN来描述RISC-V ISA中浮点寄存器的宽度，并使用FLEN = 32来表示F单精度浮点扩展。大多数浮点指令对浮点寄存器文件中的值进行操作。浮点加载和存储指令在寄存器和存储器之间传输浮点值。还提供了向整数寄存器文件传输值的指令。

我们考虑了整数和浮点值的统一寄存器文件，因为这简化了软件寄存器分配和调用约定，并减少了总用户状态。但是，拆分组织会增加使用给定指令宽度可访问的寄存器总数，从而简化了针对宽超标量问题提供足够的regfile端口，支持解耦的浮点单元架构，并简化了内部浮点编码技术的使用。很好地理解了分离寄存器文件体系结构的编译器支持和调用约定，并且在浮点寄存器文件状态上使用脏位可以减少上下文切换开销。

www.fanyigou.net 翻译狗 www.translateGo.net

图8.1：RISC-V标准F扩展单精度浮点状态。

翻译狗 www.fanyigou.net www.translateGo.net

图8.2：浮点控制和状态寄存器。

8.2 浮点控制和状态寄存器

浮点控制和状态寄存器fcsr是RISC-V控制和状态寄存器（CSR）。它是一个32位读/写寄存器，为浮点算术运算选择动态舍入模式，并保存应计异常标志，如图8.2所示。

可以使用FRCSR和FSCSR指令读取和写入fcsr寄存器，这些指令是基于CSR访问指令构建的汇编伪操作。FRCSR通过将fcsr复制到整数寄存器rd来读取fcsr。FSCSR通过将原始值复制到整数寄存器rd，然后将从整数寄存器rs1获得的新值写入fcsr来交换fcsr中的值。

fcsr中的字段也可以通过不同的CSR地址单独访问，并为这些访问定义单独的汇编程序伪操作。FRRM指令读取舍入模式字段frm并将其复制到整数寄存器rd的最低有效三位，其他所有位均为零。FSRM通过将原始值复制到整数寄存器rd中来交换frm中的值，然后将从整数寄存器rs1的三个最低有效位获得的新值写入frm。FRFLAGS和FSFLAGS的类似定义为Accrued Exception Flags字段fflags。附加伪指令FSRMI和FSFLAGSI使用立即值而不是寄存器rs1交换值。

fcsr的第31-8位保留用于其他标准扩展，包括十进制浮点的“L”标准扩展。如果不存在这些扩展，则实现应忽略对这些位的写入，并在读取时提供零值。标准软件应保留这些位的内容。

浮点运算使用在指令中编码的静态舍入模式，或者使用frm中保存的动态舍入模式。舍入模式的编码如表8.1所示。指令的rm字段中的值111选择frm中保持的动态舍入模式。如果将frm设置为无效值（101-111），则任何后续尝试使用动态舍入模式执行浮点运算都将导致非法指令陷阱。但是，具有rm字段的一些指令不受舍入模式的影响；他们应该将他们的rm字段设置为RNE（000）。

C99语言标准有效地要求提供动态舍入模式寄存器。

舍入模式	助记符	含义
000	RNE	回合最近，与Even的联系
001	RTZ	向零回合
010	RDN	向下舍入（朝向-∞）
011	RUP	向上（向+∞）
100	RMM	回合最近，与Max Magnitude联系
101		*无效。保留供将来使用。*
110		*无效。保留供将来使用。*
111		在指令的rm字段中，选择动态舍入模式； 在舍入模式寄存器中，无效。

表8.1：舍入模式编码。

应计异常标志表示自上次由软件重置字段以来对任何浮点算术指令产生的异常条件，如表8.2所示。

国旗助记符	国旗的意思
内华达州	操作无效
0J L	除以零
的	溢出
超 滤	潜流
NX	不精确

表8.2：应计异常标志编码。

在标准允许的情况下，我们不支持基本ISA中的浮点异常的陷阱，而是需要在软件中显式检查标志。我们考虑添加由浮点累积异常标志的内容直接控制的分支，但最终选择省略这些指令以保持ISA简单。

8.3 NaN生成和传播

除非另有说明，否则如果浮点运算的结果是NaN，则它是规范的NaN。规范的NaN具有正号并且除了MSB（即安静位）之外所有有效位都清楚。对于单精度浮点，这对应于模式0x7fc00000。

对于FMIN和FMAX，如果至少一个输入是信令NaN，或者如果两个输入都是安静的NaN，则结果是规范的NaN。如果一个操作数是一个安静的NaN而另一个不是NaN，则结果是非NaN操作数。

符号注入指令（FSGNJ，FSGNJN，FSGNJX）不规范化NaN;他们直接操纵底层的位模式。

我们考虑按标准推荐的方式传播NaN有效载荷，但这一决定会增加硬件成本。此外，由于该功能在标准中是可选的，因此不能用于便携式代码。

实现者可以自由地提供NaN有效载荷传播方案作为由非标准操作模式启用的非标准扩展。但是，必须始终支持上述规范的NaN方案，并且应该是默认模式。

我们要求实现在异常条件下返回标准规定的默认值，而不需要用户级软件的任何进一步干预（与Alpha ISA浮点陷阱屏障不同）。我们相信特殊情况的完整硬件处理将变得更加普遍，因此希望避免使用户级ISA复杂化以优化其他方法。实现总是可以陷入机器模式的软件处理程序，以提供出色的默认值。

8.4 次正规算术

对正常数字的操作根据IEEE 754-2008标准进行处理。按照IEEE标准的说法，在舍入后检测到细微之处。

在舍入之后检测到细微性导致更少的虚假下溢信号。

8.5 单精度加载和存储指令

浮点加载和存储使用与整数基ISA相同的基址+偏移量寻址模式，寄存器rs1中的基址和12位带符号字字节偏移量。FLW指令将单精度浮点值从内存加载到浮点寄存器rd。FSW将浮点寄存器rs2的单精度值存储到存储器中。

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	宽度	rd	操作码
12	5	3	5	7
偏移[11：0]	基础	W	destdest	LOAD-FP

31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
IMM［5］	rs2	rs1	宽度	imm[4:0]	操作码
7	5	5	3	5	7
偏移[11：5]	src	基础	W	偏移[4：0]	STORRE-FP

如果有效地址自然对齐，则FLW和FSW仅保证以原子方式执行。

8.6 单精度浮点计算指令

具有一个或两个源操作数的浮点算术指令将R型格式与OP-FP主要操作码一起使用。FADD.S，FSUB.S，FMUL.S和FDIV.S分别在rs1和rs2之间执行单精度浮点加法，减法，乘法和除法，

将结果写入rd。FMIN.S和FMAX.S分别写入较小或较大的rs1和
rs2到rd。FSQRT.S计算rs1的平方根并将结果写入rd。

2位浮点格式字段fmt的编码如表8.3所示。对于F扩展中的所有指令，它被设置为S（00）。

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

表8.3：格式字段编码。

执行舍入的所有浮点运算都可以使用rm选择舍入模式
字段编码如表8.1所示。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
FADD /多分类器	S	错误预测	上述	RM	destdest	OP-FP
浮点乘法/讯号	S	错误预测	上述	RM	destdest	OP-FP
FMIN-MAX	S	错误预测	上述	MIN/MAX	destdest	OP-FP
FSQRT	S	0	src	RM	destdest	OP-FP

浮点融合乘法 - 加法指令需要新的标准指令格式。R4类型指令指定三个源寄存器（rs1，rs2和rs3）和目标寄存器（rd）。此格式仅由浮点融合乘加指令使用。融合乘法 - 加法指令将rs1和rs2中的值相乘，可选择否定乘积，然后在rs3中加或减值，将最终结果写入rd。FMADD.S计算rs1×rs2 + rs3;FMSUB.S计算rs1×rs2-rs3;FNMSUB.S计算-rs1×rs2 + rs3;和FNMADD.S计算-rs1×rs2-rs3。

当被乘数为∞且为零时，融合的乘加指令必须引发无效操作异常，即使加数是一个安静的NaN。

IEEE 754-2008标准允许但不要求引起操作∞×0 + qNaN的无效异常。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
src3	S	错误预测	上述	RM	destdest	F [ ] [ ] msub MADD / F

8.7 单精度 浮点 转变 和 移动 说明

浮点到整数和整数到浮点的转换指令在OP-FP主要操作码空间中编码。FCVT.WS或FCVT.LS将浮点寄存器rs1中的浮点数分别转换为整数寄存器rd中的带符号32位或64位整数。FCVT.SW或FCVT.SL分别将整数寄存器rs1中的32位或64位有符号整数转换为浮点寄存器rd中的浮点数。FCVT.WU.S，FCVT.LU.S，FCVT.S.WU和FCVT.S.LU变体转换为无符号整数值或从无符号整数值转换。FCVT.L［U］.S和FCVT.SL［U］在RV32中是非法的。如果舍入结果在目标格式中无法表示，则会将其剪裁为最接近的值并设置无效标志。表8.4给出了FCVT.int.S的有效输入范围和无效输入的行为。

	fcvt。WS	fcvt。五。S	fcvt。LS	fcvt。路的。
最小有效输入（舍入后）	-2^{31}	0	-2^{63}	0
最大有效输入（舍入后）	$2^{[illegible]}-1$	$2^{[illegible]}-1$	$2^{[illegible]}-1$	$2^{[illegible]}-1$
输出超出范围的负输入	-2^{31}	0	-2^{63}	0
输出为-∞	-2^{31}	0	-2^{63}	0
输出超出范围的正输入	$2^{[illegible]}-1$	$2^{[illegible]}-1$	$2^{[illegible]}-1$	$2^{[illegible]}-1$
输出为+∞或NaN	$2^{[illegible]}-1$	$2^{[illegible]}-1$	$2^{[illegible]}-1$	$2^{[illegible]}-1$

表8.4：浮点到整数转换的域和无效输入的行为。

根据rm字段舍入所有浮点到整数和整数到浮点的转换指令。可以使用FCVT.SW rd，x0将浮点寄存器初始化为浮点正零，这将永远不会引发任何异常。

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

浮点到浮点符号注入指令FSGNJ.S，FSGNJN.S和FSGNJX.S产生的结果除了来自rs1的符号位之外的所有位。对于FSGNJ，结果的符号位是rs2的符号位;对于FSGNJN，结果的符号位与rs2的符号位相反;对于FSGNJX，符号位是rs1和rs2的符号位的XOR。符号注入指令不设置浮点异常标志。注意，FSGNJ.S rx，ry，ry将ry移动到rx（汇编程序伪操作FMV.S rx，ry）;FSGNJN.S rx，ry，ry将ry的否定移到rx（汇编程序伪操作FNEG.S rx，ry）;和FSGNJX.S rx，ry，ry将ry的绝对值移动到rx（汇编程序伪操作FABS.S rx，ry）。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
FSGNJ	S	错误预测	上述	f］［N］/ JX	destdest	OP-FP

符号注入指令提供浮点MV，ABS和NEG，以及支持一些其他操作，包括IEEE copySign操作和超越数学函数库中的符号操作。虽然MV，ABS和NEG只需要一个寄存器操作数，而FSGNJ指令需要两个，但大多数微体系结构都不太可能增加优化，以便从这些相对不频繁的指令的寄存器读取次数减少中受益。即使在这种情况下，微架构也可以简单地检测两个源寄存器何时对于FSGNJ指令是相同的，并且只读取一个副本。

提供了用于在浮点和整数寄存器之间移动位模式的指令。FMV.XW将IEEE 754-2008编码中表示的浮点寄存器rs1中的单精度值移动到整数寄存器rd的低32位。对于RV64，目标寄存器的高32位填充了浮点数符号位的副本。FMV.WX将以IEEE 754-2008标准编码编码的单精度值从整数寄存器rs1的低32位移动到浮点寄存器rd。在传输中不修改比特，并且特别地，保留非规范NaN的有效载荷。

FMV.WX和FMV.XW指令以前称为FMV.SX和FMV.XS W的使用与它们的语义更一致，因为它是一个移动32位而不解释它们的指令。在定义NaN拳击后，这变得更加清晰。为避免干扰现有代码，工具将支持W和S版本。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
fmv。XW	S	0	src	000	destdest	OP-FP
fmv。WX	S	0	src	000	destdest	OP-FP

定义基本浮点ISA以便允许实现在寄存器中使用浮点格式的内部重新编码以简化次正常值的处理并且可能减少功能单元等待时间。为此，基本ISA通过定义直接读取和写入整数寄存器文件的转换和比较操作，避免在浮点寄存器中表示整数值。这也消除了许多需要在整数和浮点寄存器之间进行显式移动的常见情况，从而减少了常见混合格式代码序列的指令数和关键路径。

8.8 单精度浮点比较指令

浮点比较指令执行浮点寄存器rs1和rs2之间的指定比较（等于，小于或小于或等于），并将布尔结果记录在整数寄存器rd中。

FLT.S和FLE.S执行IEEE 754-2008标准所指的信令比较：即，如果任一输入是NaN，则引发无效操作异常。FEQ.S执行安静的比较：仅信令NaN输入导致无效操作异常。对于所有三条指令，如果任一操作数为NaN，则结果为0。

31 — 27	26 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
钙镁磷肥	S	错误预测	上述	EQ/LT-LE	destdest	OP-FP

8.9 单精度浮点分类指令

FCLASS.S指令检查浮点寄存器rs1中的值，并向整数寄存器rd写入一个10位掩码，指示浮点数的类。表8.5中描述了掩码的格式。如果属性为true，则将设置rd中的相应位，否则将清除。rd中的所有其他位都被清除。请注意，rd中只有一位将被设置。

31 — 27	26 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
FCLASS	S	0	src	001	destdest	OP-FP

*rd*位	含义
0	*rs1是*$-\infty$。
1	*r*s1是负正常数。rs1是负的次正
2	规数。rs1是-0。
3	*r*s1是+0。
4	*r*s1是正次正规数。rs1是正的正
5	常数。rs1是$+\infty$。
6	*r*s1是信号NaN。
7	*r*s1是一个安静的NaN。
8	
9	

表8.5：FCLASS指令结果的格式。

第9章

双精度浮点的“D”标准扩展，版本 2.0

本章描述了标准的双精度浮点指令集扩展，它被命名为“D”，并添加了符合IEEE 754-2008算术标准的双精度浮点计算指令。D扩展取决于基本单精度指令子集F.

9.1 D登记州

D扩展将32个浮点寄存器f0-f31扩展为64位（图8.1中的FLEN = 64）。f寄存器现在可以保存32位或64位浮点值，如9.2节所述。

FLEN可以是32,64或128，具体取决于支持哪种F，D和Q扩展。最多可支持四种不同的浮点精度，包括H，F，D和Q.仅支持V向量扩展时，才支持半精度H标量值。

9.2 NaN拳击更窄的价值观

当支持多个浮点精度时，在称为NaN-boxing的过程中，在FLEN位NaN值的低n位中表示较窄的n位类型的有效值n <FLEN。有效NaN盒装值的高位必须全为1。因此，当被视为任何更宽的m位值时，有效的NaN加框的n位值表现为负静态NaN（qNaNs），n <m≤ FLEN。

软件可能不知道存储在浮点寄存器中的当前数据类型，但必须知道

www.fanyigou.net 翻译狗 www.translateGo.net

能够保存和恢复寄存器值，因此必须定义使用更宽的操作来传输更窄的值的结果。常见的情况是被调用者保存的寄存器，但标准惯例也适用于包括varargs，用户级线程库，虚拟机迁移和调试在内的功能。

浮点n位传输操作将以IEEE标准格式保存的外部值移入和移出f寄存器，并包含浮点加载和存储（FLn / FSn）和浮点移动指令（FMV.nX / FMV.Xn） 。f寄存器中较窄的n位传输n <FLEN将通过将目标f寄存器的所有高位FLEN-n位设置为1来创建有效的NaN盒值。从浮点运算出更窄的n位传输点寄存器将传输寄存器的低n位，忽略FLEN-n的高位。

浮点计算和符号注入操作根据f寄存器中保存的FLEN位值计算结果。窄n位操作，其中n <FLEN，检查输入操作数是否正确NaN-boxed，即所有高FLEN-n位为1.如果是，则输入的n个最低有效位用作输入值，否则输入值被视为n位规范NaN。将n位浮点结果写入目标f寄存器的n个最低有效位，将所有1写入最高FLEN-n位以产生合法的NaN盒值。

从整数到浮点的转换（例如，FCVT.SX），NaN-box将比FLEN更窄的任何结果填充FLEN位目标寄存器。从较窄的n位浮点值到整数（例如，FCVT.XS）的转换将检查合法的NaN装箱，并且如果不是合法的n位值，则将输入视为n位规范NaN。

本文档的早期版本没有定义将较窄或较宽操作数的结果输入操作的行为，除非要求更宽的保存和恢复将保留较窄操作数的值。新定义删除了此特定于实现的行为，同时仍适用于浮点单元的非重新编码和重新编码的实现。如果值使用不正确，新定义还可以通过传播NaN来捕获软件错误。

非重新编码的实现在每个浮点操作的输入和输出上解压缩并将操作数打包为IEEE标准格式。对非重新编码实现的NaN装箱成本主要在于检查较窄操作的高位是否表示合法的NaN盒值，以及将所有1写入结果的高位。

重新编码的实现使用更方便的内部格式来表示浮点值，并添加指数位以允许所有值保持标准化。重新编码实现的成本主要是跟踪内部类型和符号位所需的额外标记，但这可以通过在指数字段内部重新编码NaN而无需添加新的状态位来完成。用于将值传入和传出重新编码格式的管道需要进行小的修改，但数据路径和延迟成本很小。在任何情况下，重新编码过程必须处理宽操作数的输入子正常值的移位，并且提取NaN加框值是与归一化类似的过程，除了跳过前导1比特而不是跳过前导0比特，允许要共享的数据路径多路复用。

9.3 双精度加载和存储指令

FLD指令从存储器加载一个双精度浮点值到浮点寄存器rd。FSD将浮点寄存器的双精度值存储到存储器中。

双精度值可以是NaN盒装的单精度值。

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	宽度	rd	操作码
12	5	3	5	7
偏移[11：0]	基础	D	destdest	LOAD-FP

31 25	24 20	19 15	14 12	11 7	6 0
IMM［5］	rs2	rs1	宽度	imm[4:0]	操作码
7	5	5	3	5	7
偏移[11：5]	src	基础	D	偏移[4:0]	STORRE-FP

如果有效地址自然对齐且XLEN≥64，则FLD和FSD仅保证以原子方式执行。

9.4 双精度浮点计算指令

双精度浮点计算指令的定义类似于它们的单精度浮点计算指令，但是在双精度操作数上操作并产生双精度结果。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
FADD /多分类器	D	错误预测	上述	RM	destdest	OP-FP
浮点乘法/讯号	D	错误预测	上述	RM	destdest	OP-FP
FMIN-MAX	D	错误预测	上述	MIN/MAX	destdest	OP-FP
FSQRT	D	0	src	RM	destdest	OP-FP

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
src3	D	错误预测	上述	RM	destdest	F［］［］msub MADD / F

9.5 双精度浮点转换和移动指令

浮点到整数和整数到浮点的转换指令在OP-FP主要操作码空间中编码。FCVT.WD或FCVT.LD将浮点寄存器rs1中的双精度浮点数分别转换为整数寄存器rd中的带符号32位或64位整数。FCVT.DW或FCVT.DL分别将整数寄存器rs1中的32位或64位有符号整数转换为浮点寄存器rd中的双精度浮点数。

FCVT.WU.D，FCVT.LU.D，FCVT.D.WU和FCVT.D.LU变体转换为无符号整数值或从无符号整数值转换。FCVT.L［U］.D和FCVT.DL［U］在RV32中是非法的。FCVT.int.D的有效输入范围和无效输入的行为与FCVT.int.S相同。

根据rm字段舍入所有浮点到整数和整数到浮点的转换指令。注意FCVT.DW［U］始终产生精确结果，不受舍入模式的影响。

31–27	26–25	24–20	19–15	14–12	11–7	6–0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
FCVT.*int*.D	D	W[U]/L[U]	src	RM	destdest	OP-FP
fcvt。D.int	D	W[U]/L[U]	src	RM	destdest	OP-FP

双精度到单精度和单精度到双精度的转换指令FCVT.SD和FCVT.DS在OP-FP主要操作码空间中编码，源和目标都是浮点寄存器。rs2字段对源的数据类型进行编码，fmt字段对目标的数据类型进行编码。根据RM领域的FCVT.SD轮次;FCVT.DS永远不会圆。

31–27	26–25	24–20	19–15	14–12	11–7	6–0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
fcvt。SD	S	D	src	RM	destdest	OP-FP
fcvt。DS	D	S	src	RM	destdest	OP-FP

浮点到浮点符号注入指令FSGNJ.D，FSGNJN.D和FSGNJX.D的定义类似于单精度符号注入指令。

31–27	26–25	24–20	19–15	14–12	11–7	6–0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
FSGNJ	D	错误预测	上述	f]［N］/ JX	destdest	OP-FP

仅对于RV64，提供了在浮点和整数寄存器之间移动位模式的指令。FMV.XD将浮点寄存器rs1中的双精度值移动到整数寄存器rd中的IEEE 754-2008标准编码中的表示。FMV.DX将以IEEE 754-2008标准编码编码的双精度值从整数寄存器rs1移动到浮点寄存器rd。

31–27	26–25	24–20	19–15	14–12	11–7	6–0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
fmv。XD	D	0	src	000	destdest	OP-FP
fmv。DX	D	0	src	000	destdest	OP-FP

9.6 双精度浮点比较指令

双精度浮点比较指令的定义类似于它们的单精度对应，但是对双精度操作数进行操作。

31–27	26–25	24–20	19–15	14–12	11–7	6–0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
钙镁磷肥	D	错误预测	上述	EQ/LT-LE	destdest	OP-FP

9.7 双精度浮点分类指令

双精度浮点分类指令FCLASS.D的定义类似于其单精度对应，但是在双精度操作数上运行。

31–27	26–25	24–20	19–15	14–12	11–7	6–0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
FCLASS	D	0	src	001	destdest	OP-FP

www.fanyigou.net 翻译狗 www.translateGo.net

第10章

“Q”标准扩展为四精度浮点版本 2.0

本章介绍符合IEEE 754-2008算法标准的128位二进制浮点指令的Q标准扩展。128位或四精度二进制浮点指令子集名为“Q”，需要RV64IFD。浮点寄存器现在扩展为保持单精度，双精度或四精度浮点值（FLEN = 128）。第9.2节中描述的NaN-boxing方案现在递归地扩展，以允许单精度值在双精度值内进行NaN加框，该双精度值本身是在四精度值内的NaN框。

10.1 四精度加载和存储指令

添加了新的128位LOAD-FP和STORE-FP指令变体，并使用funct3宽度字段的新值进行编码。

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	宽度	rd	操作码
12	5	3	5	7
偏移[11：0]	基础	Q	destdest	LOAD-FP

31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
IMM［5］	rs2	rs1	宽度	imm[4:0]	操作码
7	5	5	3	5	7
偏移[11：5]	src	基础	Q	偏移[4：0]	STORRE-FP

如果有效地址自然对齐且XLEN = 128，则FLQ和FSQ仅保证以原子方式执行。

翻译狗 www.fanyigou.net www.translateGo.net

10.2 四精度计算指令

在大多数指令的格式字段中添加了一种新的支持格式，如表10.1所示。

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

表10.1：格式字段编码。

四精度浮点计算指令的定义类似于它们的双精度浮点计算指令，但是在四精度操作数上操作并产生四精度结果。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
FADD /多分类器	Q	错误预测	上述	RM	destdest	OP-FP
浮点乘法/讯号	Q	错误预测	上述	RM	destdest	OP-FP
FMIN-MAX	Q	错误预测	上述	MIN/MAX	destdest	OP-FP
FSQRT	Q	0	src	RM	destdest	OP-FP

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
src3	Q	错误预测	上述	RM	destdest	F [ ] [ ] msub MADD / F

10.3 四精度转换和移动指令

31 27	26 25	24 20	19 15	14 12	11 7	6 0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
fcvt。int.q	Q	W[U]/L[U]	src	RM	destdest	OP-FP
fcvt。Q.int	Q	W[U]/L[U]	src	RM	destdest	OP-FP

添加了新的浮点到浮点转换指令FCVT.SQ，FCVT.QS，FCVT.DQ，FCVT.QD。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
FCVT.SQ	S	Q	src	RM	destdest	OP-FP
fcvt。QS	Q	S	src	RM	destdest	OP-FP
fcvt。DQ	D	Q	src	RM	destdest	OP-FP
fcvt。QD	Q	D	src	RM	destdest	OP-FP

浮点到浮点符号注入指令FSGNJ.Q，FSGNJN.Q和FSGNJX.Q的定义类似于双精度符号注入指令。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
FSGNJ	Q	错误预测	上述	f]［N］/ JX	destdest	OP-FP

未提供FMV.XQ和FMV.QX指令，因此必须通过存储器将四精度位模式移动到整数寄存器。

RV128在Q扩展中支持FMV.XQ和FMV.QX。

10.4 四精度浮点比较指令

浮点比较指令执行浮点寄存器rs1和rs2之间的指定比较（等于，小于或小于或等于），并将布尔结果记录在整数寄存器rd中。

31 27	26 25	24 20	19 15	14 12	11 7	6 0
功能5	fmt	rs2	rs1	rm	rd	操作码
5	2	5	5	3	5	7
钙镁磷肥	Q	错误预测	上述	EQ/LT-LE	destdest	OP-FP

10.5 四精度浮点分类指令

四精度浮点分类指令FCLASS.Q的定义类似于其双精度对应指令，但是对四精度操作数进行操作。

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

Chapter 11

“L”十进制浮点的标准扩展，版本0.0

本章是一个占位符，用于指定名为“L”的标准扩展，旨在支持IEEE 754-2008标准中定义的十进制浮点运算。

11.1 十进制浮点寄存器

现有的浮点寄存器用于保存64位和128位十进制浮点值，现有的浮点加载和存储指令用于将值移入和移出内存。

由于融合乘加指令所需的大操作码空间，十进制浮点指令扩展将在30位编码空间中需要五个25位主要操作码。

Chapter 12

"C"压缩指令的标准扩展，版本2.0

本章描述了RISC-V标准压缩指令集扩展的当前草案提案，名为"C"，它通过为常见操作添加短16位指令编码来减少静态和动态代码大小。C扩展可以添加到任何基础ISA（RV32，RV64，RV128），我们使用通用术语"RVC"来涵盖其中任何一个。通常，程序中50%-60%的RISC-V指令可以用RVC指令替换，从而减少25%-30%的代码大小。

12.1 概观

RVC使用简单的压缩方案，在以下情况下提供更短的16位版本的常见32位RISC-V指令：

- 立即数或地址偏移量很小，或者
- 其中一个寄存器是零寄存器（x0），ABI链接寄存器（x1）或ABI堆栈指针（x2），或者
- 目标寄存器和第一个源寄存器是相同的，或
- 使用的寄存器是8个最受欢迎的寄存器。

C扩展与所有其他标准指令扩展兼容。C扩展允许16位指令与32位指令自由混合，后者现在能够在任何16位边界上启动。通过添加C扩展，JAL和JALR指令将不再引发指令未对齐异常。

www.fanyigou.net 翻译狗 www.translateGo.net

删除原始32位指令上的32位对齐约束可以显着提高代码密度。

压缩指令编码在RV32C，RV64C和RV128C中最常见，但如表12.3所示，根据基本ISA宽度，一些操作码用于不同目的。例如，更宽的地址空间RV64C和RV128C变体需要额外的操作码来压缩64位整数值的加载和存储，而RV32C使用相同的操作码来压缩加载和存储单精度浮点值。类似地，RV128C需要额外的操作码来捕获128位整数值的加载和存储，而这些相同的操作码用于RV32C和RV64C中的双精度浮点值的加载和存储。如果实现了C扩展，则只要还实现了相关的标准浮点扩展（F和/或D），就必须提供适当的压缩浮点加载和存储指令。此外，RV32C包括压缩跳转和链接指令以压缩短程子程序调用，其中相同的操作码用于压缩RV64C和RV128C的ADDIW。

双精度加载和存储是静态和动态指令的重要部分，因此将它们包含在RV32C和RV64C编码中的动机。

虽然单精度加载和存储不是为当前支持的ABI编译的基准测试的静态或动态压缩的重要来源，但对于仅提供硬件单精度浮点单元且具有仅支持单精度浮点的ABI的微控制器在点数中，单精度加载和存储将至少与双精度加载和存储一样频繁地用于测量基准。因此，在RV32C中为这些提供压缩支持的动机。

短程子程序调用更可能出现在微控制器的小二进制文件中，因此有动力将这些调用包含在RV32C中。

尽管为不同的基址寄存器宽度重用不同目的的操作码会增加文档的复杂性，但即使对于支持多个基本ISA寄存器宽度的设计，对实现复杂性的影响也很小。压缩的浮点加载和存储变体使用相同的指令格式，具有与更宽的整数加载和存储相同的寄存器说明符。

RVC的设计是在每个RVC指令扩展为基本ISA（RV32I / E，RV64I或RV128I）或F和D标准扩展（存在）中的单个32位指令的约束下。采用这种约束有两个主要好处：

- 硬件设计可以在解码期间简单地扩展RVC指令，简化验证并最小化对现有微体系结构的修改。
- 编译器可能不知道RVC扩展并将代码压缩留给汇编器和链接器，尽管压缩感知编译器通常能够产生更好的结果。

我们认为C和基本IFD指令之间简单的一对一映射的复杂性降低远远超过了稍微更密集的编码的潜在收益，这些编码增加了仅在C扩展中支持的附加指令，或者允许在一个扩展中编码多个IFD指令C指令。

值得注意的是，C扩展并非设计为独立的ISA，而是与基本ISA一起使用。

可变长度指令集一直用于提高代码密度。例如，20世纪50年代后期开发的IBM Stretch [6]具有带32位和64位指令的ISA，其中一些32位指令是完整64位指令的压缩版本。Stretch还采用了限制在一些较短指令格式中可寻址的寄存器集的概念，其中短分支指令只能引用其中一个索引寄存器。后来的IBM 360架构[3]支持使用16位，32位或48位指令格式的简单可变长度指令编码。

1963年，CDC推出了Cray设计的CDC 6600 [28]，这是RISC架构的前身，它引入了一个富含寄存器的加载 - 存储架构，其指令有两个长度，15位和30位。后来的Cray-1设计使用了非常相似的指令格式，具有16位和32位指令长度。

20世纪80年代的初始RISC ISA都选择了代码大小的性能，这对于工作站环境是合理的，但对于嵌入式系统则不然。因此，ARM和MIPS随后通过提供替代的16位宽指令集而不是标准的32位宽指令来制作提供更小代码尺寸的ISA版本。压缩的RISC ISA将相对于起始点的代码大小减少了大约25-30%，从而产生的代码明显小于80x86。这个结果让人感到惊讶，因为他们的直觉是可变长度CISC ISA应该小于仅提供16位和32位格式的RISC ISA。

由于最初的RISC ISA没有留下足够的操作码空间来包含这些未计划的压缩指令，因此它们被开发为完整的新ISA。这意味着编译器需要为不同的压缩ISA提供不同的代码生成器。第一个压缩的RISC ISA扩展（例如，ARM Thumb和MIPS16）仅使用固定的16位指令大小，这大大减少了静态代码大小，但导致动态指令数量增加，导致与原始指令相比性能降低固定宽度的32位指令大小。这导致开发了第二代压缩RISC ISA设计，具有混合的16位和32位指令长度（例如，ARM Thumb2，microMIPS，PowerPC VLE），因此性能类似于纯32位指令，但具有节省了大量代码。遗憾的是，这些不同代的压缩ISA与原始的未压缩ISA不兼容，导致文档，实现和软件工具支持的显着复杂性。

在常用的64位ISA中，只有PowerPC和microMIPS目前支持压缩指令格式。令人惊讶的是，最受欢迎的移动平台64位ISA（ARM v8）不包含压缩指令格式，因为静态代码大小和动态指令获取带宽是重要的指标。虽然静态代码大小不是大型系统中的主要问题，但是指令获取带宽可能是运行商业工作负载的服务器的主要瓶颈，这些工作负载通常具有大的指令工作集。

得益于25年的后见之明，RISC-V旨在从一开始就支持压缩指令，为RVC留下足够的操作码空间作为基础ISA上的简单扩展（以及许多其他扩展）。RVC的理念是减少嵌入式应用程序的代码大小，并提高所有应用程序的性能和能效，因为指令缓存中的错失更少。Waterman表明，RVC的指令位减少了25%-30%，这使指令缓存未命中减少了20%-25%，或者与指令缓存大小加倍大致相同的性能影响[33]。

12.2 压缩指令格式

表12.1显示了八种压缩指令格式。CR，CI和CSS可以使用32个RVI寄存器中的任何一个，但CIW，CL，CS和CB仅限于其中的8个。表12.2列出了这些流行的寄存器，它们对应于寄存器x8到x15。请注意，有一个单独的版本

翻译狗 www.fanyigou.net www.translateGo.net

加载和存储使用堆栈指针作为基址寄存器的指令，因为保存到堆栈并从堆栈恢复是如此普遍，并且它们使用CI和CSS格式来允许访问所有32个数据寄存器。CIW为ADDI4SPN指令提供8位立即数。

RISC-V ABI已更改为使常用寄存器映射到寄存器x8-x15。这通过具有连续的自然对齐的寄存器编号集来简化解压缩解码器，并且还与RV32E子集基本规范兼容，其仅具有16个整数寄存器。

基于压缩寄存器的浮点加载和存储也分别使用CL和CS格式，其中8个寄存器映射到f8到f15。

标准RISC-V调用约定将最常用的浮点寄存器映射到寄存器f8到f15，这允许与整数寄存器号相同的寄存器解压缩解码。

这些格式旨在将两个寄存器源指定符的位保留在所有指令中的相同位置，而目标寄存器字段可以移动。当存在完整的5位目标寄存器指定符时，它与32位RISC-V编码位于同一位置。在对符号进行符号扩展的情况下，符号扩展始终来自第12位。立即字段已被加扰，如基本规范中所示，以减少所需的立即多路复用器的数量。

立即字段以指令格式而不是按顺序加扰，使得尽可能多的位在每个指令中处于相同位置，从而简化了实现。例如，立即位17-10总是来自相同的指令位位置。其他五个立即位（5,4,3,1和0）只有两个源指令位，而四个（9,7,6和2）有三个源，一个（8）有四个源。

对于许多RVC指令，不允许零值immediates，x0不是有效的5位寄存器说明符。这些限制为需要较少操作数位的其他指令释放了编码空间。

格式	含义
铬	寄存器
那里	立即CSS
	堆栈相对商店CIW
	宽立即CL
	加载
反恐精英	商店
CB	科
CJ	跳

15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
功能4		rd/rs1		rs2		运
功能3	imm	rd/rs1		imm		运
功能3	imm			rs2		运
功能3	imm				rd′	运
功能3	imm		rs1′	imm	rd′	运
功能3	imm		rs1′	imm	rs2′	运
功能3	抵消		rs1	抵消		运

表12.1：压缩的16位RVC指令格式。

RVC寄存器编号整数寄存器编号整数寄存器ABI名称
浮点寄存器号浮点寄存器ABI名称

000	001	010	011	100	101	110	111
x8	x9	x10	x11	x12	x13	x14	x15
s0	s1	a0	a1	a2	a3	a4	a5
f8	f9	f10	f11	f12	f13	f14	f15
fs0	fs1	粮	fa1	2	法	4	fa5

表12.2：由CIW，CL，CS和CB格式的三位rs1'，rs2'和rd'字段指定的寄存器。

12.3 加载和存储指令

为了增加16位指令的范围，数据传输指令使用零扩展的立即数，按字节数据的大小进行缩放：单词为×4，双字为×8，四字为×16。

RVC提供两种加载和存储变体。一个使用ABI堆栈指针x2作为基址，可以定位任何数据寄存器。另一个可以引用8个基址寄存器之一和8个数据寄存器之一。

基于堆栈指针的负载和存储

15 13	12	11 7	6 2	1 0
功能3	imm	rd	imm	运
3	1	5	5	2
C.LWSP	偏移[5]	接待/ = 0	偏移[4：2 ｜ 7：6]	C2
C.LDSP	偏移[5]	接待/ = 0	偏移[4：3 ｜ 8：6]	C2
C.LQSP	偏移[5]	接待/ = 0	偏移[4 ｜ 9：6]	C2
C.FLWSP	偏移[5]	destdest	偏移[4：2 ｜ 7：6]	C2
C.FLDSP	偏移[5]	destdest	偏移[4：3 ｜ 8：6]	C2

这些说明使用CI格式。

C.LWSP从存储器加载32位值到寄存器rd。它通过将按比例缩放4的零扩展偏移量添加到堆栈指针x2来计算有效地址。它扩展到lw rd，偏移[7：2]（x2）。

C.LDSP是一种仅用于RV64C / RV128C的指令，它将64位值从存储器加载到寄存器rd中。它通过将标度为8的零扩展偏移量添加到堆栈指针x2来计算其有效地址。它扩展到ld rd，偏移[8：3]（x2）。

C.LQSP是一个仅RV128C指令，它将128位值从存储器加载到寄存器rd。它通过将按比例缩放16的零扩展偏移量添加到堆栈指针x2来计算其有效地址。它扩展到lq rd，偏移[9：4]（x2）。

翻译狗 www.fanyigou.net www.translateGo.net

C.FLWSP是一个仅RV32FC指令，它将单精度浮点值从存储器加载到浮点寄存器rd。它通过将按比例缩放4的零扩展偏移量添加到堆栈指针x2来计算其有效地址。它扩展到flw rd，偏移[7：2]（x2）。

C.FLDSP是一个RV32DC / RV64DC-only指令，它将从存储器的双精度浮点值加载到浮点寄存器rd。它通过将标度为8的零扩展偏移量添加到堆栈指针x2来计算其有效地址。它扩展到fld rd，偏移[8：3]（x2）。

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

这些说明使用CSS格式。

C.SWSP将寄存器rs2中的32位值存储到存储器中。它通过将按比例缩放4的零扩展偏移量添加到堆栈指针x2来计算有效地址。它扩展为sw rs2，offset [7：2]（x2）。

C.SDSP是仅用于RV64C / RV128C的指令，它将寄存器rs2中的64位值存储到存储器中。它通过将标度为8的零扩展偏移量添加到堆栈指针x2来计算有效地址。它扩展为sd rs2，偏移[8：3]（x2）。

C.SQSP是一个仅RV128C指令，它将寄存器rs2中的128位值存储到存储器中。它通过将按比例缩放16的零扩展偏移量添加到堆栈指针x2来计算有效地址。它扩展到sq rs2，偏移[9：4]（x2）。

C.FSWSP是一条仅用于RV32FC的指令，它将浮点寄存器rs2中的单精度浮点值存储到存储器中。它通过将按比例缩放4的零扩展偏移量添加到堆栈指针x2来计算有效地址。它扩展为fsw rs2，offset [7：2]（x2）。

C.FSDSP是一种仅用于RV32DC / RV64DC的指令，它将浮点寄存器rs2中的双精度浮点值存储到存储器中。它通过将标度为8的零扩展偏移量添加到堆栈指针x2来计算有效地址。它扩展为fsd rs2，偏移[8：3]（x2）。

函数入口/出口处的寄存器保存/恢复代码代表静态代码大小的重要部分。RVC中基于堆栈指针的压缩加载和存储有效地将保存/恢复静态代码大小减少了2倍，同时通过减少动态指令带宽来提高性能。

在其他ISA中用于进一步减少保存/恢复代码大小的常用机制是加载多个和存储多个指令。我们考虑将这些用于RISC-V，但注意到这些指令存在以下缺点：

- *这些指令使处理器实现复杂化。*

- *对于虚拟内存系统，一些数据访问可能驻留在物理内存中，而另一些则不能，这需要一个新的重启机制来执行部分执行的指令。*
- *与其他RVC指令不同，没有IFD等同于加载多个和存储多个。*
- *与其余的RVC指令不同，编译器必须知道这些指令，以生成指令并按顺序分配寄存器，以最大限度地保存和存储它们，因为它们将被保存和恢复。顺序。*
- *简单的微体系结构实现将限制如何在加载周围安排其他指令并存储多条指令，从而导致潜在的性能损失。*
- *对顺序寄存器分配的期望可能与为CIW，CL，CS和CB格式选择的特征寄存器冲突。*

此外，通过用对常见序言和结尾代码的子程序调用替换序言和结尾代码，可以在软件中实现许多增益，这是[34]的第5.6节中描述的技术。

虽然合理的架构师可能得出不同的结论，但我们决定省略加载和存储多个，而是使用纯软件方法调用save / restore millicode例程以获得最大的代码大小减少。

基于寄存器的载荷和商店

15–13	12–10	9–7	6–5	4–2	1–0
功能3	imm	rs1′	imm	rd′	运
3	3	3	2	3	2
C.LW	偏移[5：3]	基础	偏移[2 ｜ 6]	destdest	C0
C.LD	偏移[5：3]	基础	偏移[7：6]	destdest	C0
C.LQ	偏移[5 ｜ 4 ｜ 8]	基础	偏移[7：6]	destdest	C0
C.FLW	偏移[5：3]	基础	偏移[2 ｜ 6]	destdest	C0
C.FLD	偏移[5：3]	基础	偏移[7：6]	destdest	C0

这些说明使用CL格式。

C.LW将32位值从存储器加载到寄存器rd′。它通过将按比例缩放4的零扩展偏移量添加到寄存器rs1′中的基址来计算有效地址。它扩展到lw rd′，偏移[6：2]（rs1′）。

C.LD是一个RV64C / RV128C指令，它将64位值从存储器加载到寄存器rd′。它通过将缩放为8的零扩展偏移量添加到寄存器rs1′中的基址来计算有效地址。它扩展到ld rd′，偏移[7：3]（rs1′）。

C.LQ是一个仅RV128C指令，它将128位值从存储器加载到寄存器rd′。它通过将缩放为16的零扩展偏移量添加到寄存器rs1′中的基址来计算有效地址。它扩展到lq rd′，偏移[8：4]（rs1′）。

C.FLW是一个仅RV32FC指令，它将单精度浮点值从存储器加载到浮点寄存器rd′。它通过将缩放为4的零扩展偏移量添加到寄存器rs1′中的基址来计算有效地址。它扩展到flw rd′，偏移[6：2]（rs1′）。

C.FLD是一个RV32DC / RV64DC-only指令，它将一个双精度浮点值从存储器加载到浮点寄存器rd′。它通过将缩放为8的零扩展偏移量添加到寄存器rs1′中的基址来计算有效地址。它扩展到fld rd′，偏移[7：3]（rs1′）。

15 13	12 10	9 7	6 5	4 2	1 0
功能3	imm	RSI′	imm	rs2′	运
3	3	3	2	3	2
C.SW	偏移[5：3]	基础	偏移[2 ｜ 6]	src	C0
C.SD.	偏移[5：3]	基础	偏移[7：6]	src	C0
C.SQ	偏移[5 ｜ 4 ｜ 8]	基础	偏移[7：6]	src	C0
C.FSW	偏移[5：3]	基础	抵消[2/6]	src	C0
消防处	偏移[5：3]	基础	抵消[7:6]	src	C0

这些说明使用CS格式。

C.SW将32位值存储在寄存器rs2′中。它通过将缩放为4的零扩展偏移量添加到寄存器rs1′中的基址来计算有效地址。它扩展到sw rs2′，偏移[6：2]（rs1′）。

C.SD是仅用于RV64C / RV128C的指令，它将寄存器rs2′中的64位值存储到存储器中。它通过将缩放为8的零扩展偏移量添加到寄存器rs1′中的基址来计算有效地址。它扩展为sd rs2′，偏移[7：3]（rs1′）。

C.SQ是一个仅RV128C指令，它将寄存器rs2′中的128位值存储到存储器中。它通过将缩放为16的零扩展偏移量添加到寄存器rs1′中的基址来计算有效地址。它扩展到sq rs2′，偏移[8：4]（rs1′）。

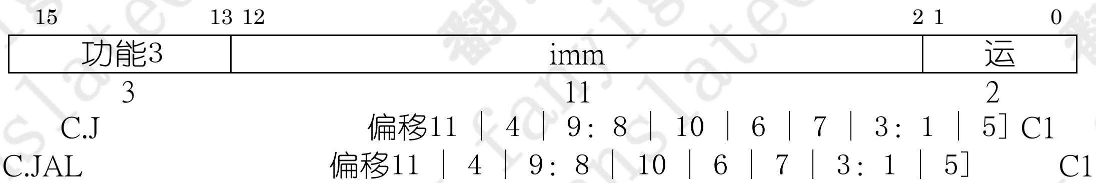
C.FSW是一个仅RV32FC指令，它将浮点寄存器rs2′中的单精度浮点值存储到存储器中。它通过将按比例缩放4的零扩展偏移量添加到寄存器rs1′中的基址来计算有效地址。它扩展到fsw rs2′，偏移[6：2]（rs1′）。

C.FSD是一个RV32DC / RV64DC-only指令，它将浮点寄存器rs2′中的双精度浮点值存储到存储器中。它通过将缩放为8的零扩展偏移量添加到寄存器rs1′中的基址来计算有效地址。它扩展到fsd rs2′，偏移[7：3]（rs1′）。

12.4 控制转移指令

RVC提供无条件跳转指令和条件分支指令。与基本RVI指令一样，所有RVC控制传输指令的偏移量都是2个字节的倍数。

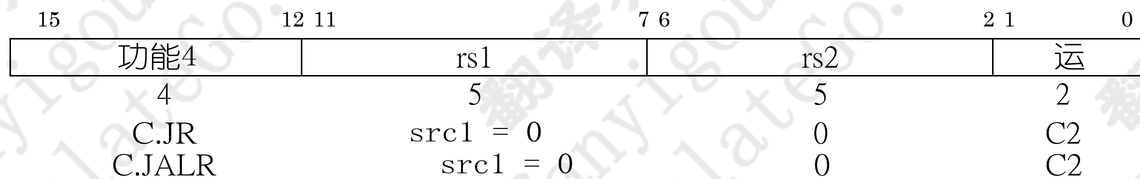
翻译狗 www.fanyigou.net www.translateGo.net



这些说明使用CJ格式。

CJ执行无条件控制转移。偏移是符号扩展并添加到pc以形成跳转目标地址。因此，CJ可以达到±2 KiB范围。CJ扩展到jal x0，偏移[11：1]。

C.JAL是一个RV32C指令，它执行与CJ相同的操作，但另外将跳转后的指令地址（pc + 2）写入链接寄存器x1。C.JAL扩展到jal x1，偏移[11：1]。

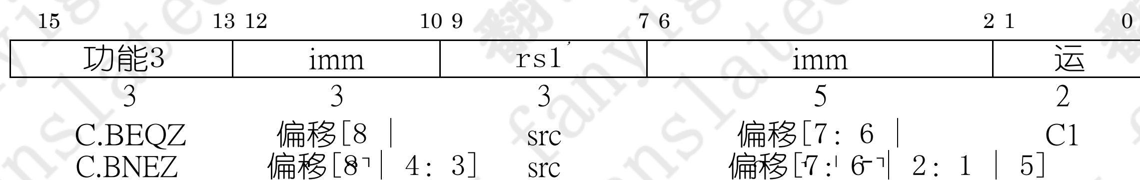


这些说明使用CR格式。

C.JR（跳转寄存器）执行无条件控制转移到寄存器rs1中的地址。C.JR扩展为jalr x0，rs1，0。

C.JALR（跳转和链接寄存器）执行与C.JR相同的操作，但另外将跳转后的指令地址（pc + 2）写入链接寄存器x1。C.JALR扩展到jalr x1，rs1，0。

严格来说，C.JALR并没有完全扩展到基本RVI指令，因为添加到PC以形成链接地址的值是2而不是基本ISA中的4，但是支持2和4字节的偏移量只是一个基础微体系结构的微小变化。



C1这些指令使用CB格式。

C.BEQZ执行条件控制转移。偏移是符号扩展并添加到pc以形成分支目标地址。因此它可以达到256 B范围。如果寄存器rs1′ 中的值为零，则C.BEQZ采用分支。它扩展为beq rs1′，x0，偏移[8：1]。

C.BNEZ的定义类似，但如果rs1′ 包含非零值则需要分支。它扩展为rs1′，x0，偏移[8：1]。

12.5 整数计算指令

RVC提供了几个整数运算和常量生成的指令。

整数常数生成指令

这两个常量生成指令都使用CI指令格式，并且可以定位任何整数寄存器。

15 13	12	11 7	6 2	1 0
功能3	imm[5]	rd	imm[4:0]	运
3	1	5	5	2
李国宝	imm[5]	接待/ = 0	imm[4:0]	C1
C.LUI	nzuimm[17]	dest/={0, 2} dest/={0, 2}	nzuimm [16点十二]	C1

C.LI将符号扩展的6位立即数imm加载到寄存器rd中。C.LI仅在rd / = x0时有效。C.LI扩展为addi rd，x0，imm [5：0]。

C.LUI将非零的6位立即数字段加载到目标寄存器的第17-12位，清除最后的12位，并将第17位符号扩展到目标的所有高位。C.LUI仅在rd / = {x0，x2}时有效，并且当立即数不等于零时有效。C.LUI扩展到lui rd，nzuimm [17:12]。

整数寄存器 - 即时操作

这些整数寄存器立即操作以CI格式编码，并对任何非x0整数寄存器和6位立即数执行操作。直接不能为零。

15 13	12	11 7	6 2	1 0
功能3	imm[5]	rd/rs1	imm[4:0]	3
日	1	5	5	2
C.ADDI	nzimm[5]	destdest	nzimm：0 [4]	C1
C. ADDEW	imm[5]	接待/ = 0	imm[4:0]	C1
C.ADDI16SP [9]		2	nzimm〔4 | 8 7 6 5 |:	
|]	C1			

C.ADDI将非零符号扩展的6位立即数添加到寄存器rd中的值，然后将结果写入rd。C.ADDI扩展到addi rd，rd，nzimm [5：0]。

C.ADDIW是一个仅RV64C / RV128C指令，它执行相同的计算但产生32位结果，然后将结果符号扩展为64位。C.ADDIW扩展为addiw rd，rd，imm [5：0]。对于C.ADDIW，立即数可以为零，其中这对应于sext.w rd。

C.ADDI16SP与C.LUI共享操作码，但目标字段为x2。C.ADDI16SP将非零符号扩展的6位立即数添加到堆栈指针中的值（sp = x2），其中立即数被缩放以表示范围（-512,496）中的16的倍数。C.ADDI16SP用于调整程序序言和结尾中的堆栈指针。它扩展为addi x2，x2，nzimm [9：4]。

在标准RISC-V调用约定中，堆栈指针sp始终是16字节对齐的。

15 13	12 5	4 2	1 0
功能3	imm	rd′	运
3	8	3	2
C.ADDI4SPN	nzuimm[5:4|9:6|2|3]	destdest	C0

C.ADDI4SPN是一种CIW格式的RV32C / RV64C指令，它将一个零扩展的非零立即数（按4缩放）添加到堆栈指针x2，并将结果写入rd′。该指令用于生成指向堆栈分配变量的指针，并扩展为addi rd′，x2，nzuimm [9：2]。

15 13	12	11 7	6 2	1 0
功能3	shamt[5]	rd/rs1	shamt[4:0]	运
3	1	5	5	2
C.SLLI	shamt[5]	接待/ = 0	shamt[4:0]	C2

C.SLLI是一种CI格式指令，它对寄存器rd中的值执行逻辑左移，然后将结果写入rd。移位量在shamt字段中编码，其中shamt [5]对于RV32C必须为零。对于RV32C和RV64C，移位量必须为非零。对于RV128C，移位量为零用于编码64的移位.C.SLLI扩展为slli rd，rd，shamt [5：0]，除了具有shamt = 0的RV128C，其扩展为s rd，rd， 64。

15 13	12	11 10	9 7	6 2	1 0
功能3	shamt[5]	功能2	rd′/rs1′	shamt[4:0]	运
3	1	2	3	5	2
C.SRLI	shamt[5]	C.SRLI	destdest	shamt[4:0]	C1
C.SRAI	shamt[5]	C.SRAI	destdest	shamt[4:0]	C1

C.SRLI是CB格式指令，它执行寄存器rd′中值的逻辑右移，然后将结果写入rd′。移位量在shamt字段中编码，其中shamt [5]对于RV32C必须为零。对于RV32C和RV64C，移位量必须为非零。对于RV128C，使用零移位量来对64的移位进行编码。此外，对于RV128C，移位量是符号扩展的，因此合法移位量是1-31,64和96-127。C.SRLI扩展为srli rd′，rd′，shamt [5：0]，除了具有shamt = 0的RV128C，其扩展到srli rd′，rd′，64。

C.SRAI的定义类似于C.SRLI，而是执行算术右移。C.SRAI扩展到srai rd′，rd′，shamt [5：0]。

左移通常比右移更频繁，因为左移经常用于缩放地址值。因此，右移已被授予较少的编码空间，并被放置在编码象限中，其中所有其他的中间符号都是符号扩展的。对于RV128，决定立即对6位移位量进行符号扩展。除了降低解码复杂度之外，我们认为右移数量96-127将比64-95更有用，以允许提取位于128位地址指针的高位部分的标签。我们注意到RV128C不会与RV32C和RV64C在同一点冻结，以便评估128位地址空间代码的典型用法。

15 13	12	11 10	9 7	6 2	1 0
功能3	imm[5]	功能2	rd′/rs1′	imm[4:0]	运
3	1	2	3	5	2
C.ANDI	imm[5]	C.ANDI	destdest	imm[4:0]	C1

C.ANDI是一个CB格式的指令，它计算寄存器rd′ 中的值的按位AND和符号扩展的6位立即数，然后将结果写入rd′。C.ANDI扩展到andi rd′，rd′，imm [5: 0]。

整数寄存器 – 寄存器操作

15 12	11 7	6 2	1 0
功能4	rd/rs1	rs2	运
4	5	5	2
C.MV	接待/ = 0	src1 = 0	C2
总和4.	接待/ = 0	src1 = 0	C2

这些说明使用CR格式。

C.MV将寄存器rs2中的值复制到寄存器rd中。C.MV扩展为添加rd，x0，rs2。

C.ADD在寄存器rd和rs2中添加值，并将结果写入寄存器rd。C.ADD扩展为添加rd，rd，rs2。

15 10	9 7	6 5	4 2	1 0
功能6	rd′/rs1′	本功能	rs2′	运
6	3	2	3	2
C.	destdest	C.	src	C1
C	destdest	C	src	C1
C.XOR	destdest	C.XOR	src	C1
苏先生	destdest	苏先生	src	C1
C.ADDW	destdest	C.ADDW	src	C1
C.SUBW	destdest	C.SUBW	src	C1

这些说明使用CS格式。

C.AND计算寄存器rd′ 和rs2′中值的按位AND，然后将结果写入寄存器rd′。C.and扩展到和rd′，rd′，rs2′。

翻译狗 www.fanyigou.net www.translateGo.net

C.OR计算寄存器rd′ 和rs2′中值的按位OR，然后将结果写入寄存器rd′。C.OR扩展到或者′，rd′，rs2′。

C.XOR计算寄存器rd′ 和rs2′中值的按位异或，然后将结果写入寄存器rd′。C.XOR扩展为xor rd′，rd′，rs2′。

C.SUB从寄存器rd′中的值中减去寄存器rs2′ 中的值，然后将结果写入寄存器rd′。C.SUB扩展到sub rd′，rd′，rs2′。

C.ADDW是一个RV64C / RV128C指令，它将寄存器rd′ 和rs2′中的值相加，然后在将结果写入寄存器rd′之前对该和的低32位进行符号扩展。C.ADDW扩展为addw rd′，rd′，rs2′。

C.SUBW是一个RV64C / RV128C指令，它从寄存器rd′中的值中减去寄存器rs2′ 中的值，然后在将结果写入寄存器rd之前对差值的低32位进行符号扩展′。C.SUBW扩展到subw rd′，rd′，rs2′。

这组六条指令不单独提供大量节省，但不占用太多编码空间并且易于实现，并且作为一组在静态和动态压缩方面提供了有价值的改进。

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

所有位为零的16位指令永久保留为非法指令。

我们保留全零指令是非法指令，以帮助捕获尝试执行内存空间的零或不存在部分。不应在任何非标准扩展中重新定义全零值。类似地，我们保留将所有位设置为1（对应于RISC-V可变长度编码方案中的非常长的指令）的指令作为非法捕获在不存在的存储器区域中看到的另一个公共值。

NOP指令

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

C.NOP是一种CI格式的指令，除了推进pc之外，它不会改变任何用户可见的状态。C.NOP编码为c.addi x0,0，因此扩展为addi x0，x0,0。

断点指令

15–12	11–2	1–0
功能4	0	运
4	10	2
C. EBAK	0	C2

调试器可以使用扩展为ebreak的C.EBREAK指令，以使控制转移回调试环境。C.EBREAK与C.ADD指令共享操作码，但rd和rs2都为零，因此也可以使用CR格式。

12.6 在LR / SC序列中使用C指令

在支持C扩展的实现上，可以使用LR / SC序列中允许的I指令的压缩形式，同时保留最终成功的保证，如第7.2节所述。

这意味着任何声称支持A和C扩展的实现必须确保包含有效C指令的LR / SC序列最终完成。

翻译狗 www.fanyigou.net www.translateGo.net

12.7 RVC指令集列表

表12.3显示了RVC主要操作码的映射。设置低两位的操作码对应于宽于16位的指令，包括基本ISA中的指令。几条指令仅对某些操作数有效;当无效时，它们被标记为RES以指示操作码被保留用于将来的标准扩展;NSE表示操作码保留用于非标准扩展;或提示表明操作码是为未来的标准微架构提示保留的。标记为HINT的指令必须在对提示无效的实现上执行为no-ops。

HINT指令旨在支持将来可能影响性能但不影响架构状态的微架构提示。选择了HINT编码，以便简单的实现可以忽略HINT编码并将HINT作为不改变体系结构状态的常规操作来执行。例如，如果目标寄存器是x0，则C.ADD是一个提示，其中五位rs2字段编码HINT的细节。但是，一个简单的实现可以简单地执行HINT作为寄存器x0的添加，这将不起作用。

研究所[15:13] 研究所[1:0]	000	001	010	011	100	101	110	111	
00	ADDI4SPN	FLD FLD LQ	LW	FLW LD	*保留的*	消防 FSD 平 方	申银 万国	搅拌摩 擦焊 sd 卡	RV32 RV64 RV128
01	阿迪	日航 已执行	李	他/ addi16sp	MISC-ALU	J	BEQZ	贝内特	RV32 RV64 RV128
10	SLLI	FLDSP FLDSP LQ	镀锌	FLWSP 模板 LDSP	f] [AL] R / MV / ADD	FSDSP fsdsp 平方	溶胀 聚合	FSWSP 菱形 菱形	
11	>16b								

表12.3：RVC操作码映射

表12.4-12.6列出了RVC指令。

15 14 13	12 11 10	9 8 7	6 5	4 3 2	1 0
000	0			0	00
000	nzuimm[5:4|9:6|2|3]			rd^t	00
001	uimm[5:3]	rs1$_t$	在此［ 6 ］	rd^t	00
001	［ 8 ］ |在此5:4	RSI$_t$	在此［ 6 ］	rd^t	00
010	uimm[5:3]	RSI$_t$	uimm[2|6]	rd^t	00
011	uimm[5:3]	rs1$_t$	uimm[2|6]	rd^t	00
011	uimm[5:3]	rs1$_t$	在此［ 6 ］	rd^t	00
100	—				00
101	uimm[5:3]	RSI$_t$	在此	rs2	00

非法指令C.ADDI4SPN (RES, nzuimm = 0) C.FLD (RV32 / 64)
C.LQ *(RV128)*
C.LW
C.FLW *(RV32)*
C. *ldc (rv64/128)*
保留的
C.FSD *(RV32/64)*
C.SQ *(RV128)*
C.SW
C.FSW *(RV32)*
C.SD *(RV64/128)*

表12.4：RVC的指令列表，象限0。

15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
000	0	0		0		01
000	nzimm[5]	↓/研发/ = 0		nzimm: 0［ 4 ］		01
001	【11 | IMM 4 | 9:8 | 10 | 6 | 7 | 3:1 | 5 】					01
001	imm[5]	↓/研发/ = 0		imm[4:0]		01
010	imm[5]	rd≠0		imm[4:0]		01
011	nzimm[9]	2		nzimm［4 | 8 7 6 5 |: | ］		01
011	nzimm[17]	{0, 2}		nzimm [16点12分]		01
100	nzuimm[5]	00	rs1^t/rd^t	nzuimm[4:0]		01
100	0	00	RSI^t/研发^t	0		01
100	nzuimm[5]	01	RSI^t/研发^t	nzuimm[4:0]		01
100	0	01	RSI^t/研发^t	0		01
100	imm[5	[illegible]	RSI^t/研发^t	imm[4:0]		01
100	0	11	RSI^t/研发^t	00	rs2^t	01
100	0	11	rs1^t/rd^t	01	rs2^t	01

C.NOP
C. *ADDI (提示, nzimm = 0)*
C.JAL *(RV32)*
C. *ADDIW (rv64 / 128; RES, RD = 0)*
c.li (提示, RD = 0) c.addi16sp (RES, nzimm = 0) c.lui (RES, nzimm = 0; 暗示, RD = 0) c.srli (RV32 NSE、nzuimm［ 5 ］= 1) c.srli64 (rv128; RV32 / 64提示) c.srai (RV32 NSE、nzuimm［ 5 ］= 1) c.srai64 (rv128; RV32 / 64提示)
c.andi
c. sub
c. xor
C或C
C. *subw (rv64128;RV32 RES)*
C.ADDW *(RV64/128; RV32 RES)*
保留的
*保留*CJ
C.BEQZ
C.BNEZ

表12.5：RVC，象限1的指令列表。

15 14 13	12	11 10 9 8 7	6 5 4 3 2	1 0
000	nzuimm[5]	↓/研发 / = 0	nzuimm[4:0]	10
000	0	↓/研发 / = 0	0	10
001	在此 [5]	rd	在此[4:3 | 8:6】	10
001	在此 [5]	rd≠0	在此| 9:6】【4	10
010	在此 [5]	rd≠0	在此[4:2 | 7:6】	10
011	在此 [5]	rd	在此[4:2 | 7:6】	10
011	在此 [5]	rd≠0	在此[4:3 | 8:6】	10
100	0	↓/ = 0	0	10
100	0	rd≠0	rs2\ 0	10
100	1	0	0	10
100	1	↓/ = 0	0	10
100	1	↓/研发	rs2\ 0	10

C. *SLLI (提示, RD = 0; RV32 NSE、nzuimm [5] = 1)*
c.slli64 (rv128; RV32 / 64提示; 提示, RD = 0)
c.fldsp (RV32 / 64)
c. *lqsp (rv128);RES, rd=0)*
C. *LWSP (RES, rd = 0)*
C. FLWSP (RV32)
C.LDSP *(RV64/128; RES, rd=0)*
c. jr (res, rs1=0) c. mv (int, rdct. 0) c. eb瑞 k
c. jalr
C. *ADD (提示, rd = 0)*
C. FSDSP (RV32 / 64) C. SQSP (RV128)
C. SWSP C. FSWSP (RV32) C. SDSP (RV64 / 128)

表12.6：RVC, 象限2的指令列表。

翻译狗 www.fanyigou.net www.translateGo.net

用于位操作的“B”标准扩展，版本0.0

本章是未来标准扩展的占位符，用于提供位操作指令，包括插入，提取和测试位字段以及旋转，漏斗移位以及位和字节排列的指令。

尽管位操作指令在某些应用程序域中非常有效，特别是在处理外部打包数据结构时，我们将它们从基本ISA中排除，因为它们在所有域中都没用，并且可以添加额外的复杂性或指令格式来提供所有需要的操作数。

我们预计B扩展将是基本30位指令空间内的棕色域编码。

翻译狗 www.fanyigou.net www.translateGo.net

Chapter 13

翻译狗 www.fanyigou.net www.translateGo.net

动态翻译语言的“J”标准扩展，版本0.0

本章是未来标准扩展的占位符，以支持动态翻译的语言。

许多流行语言通常通过动态翻译实现，包括Java和Javascript。这些语言可以受益于动态检查和垃圾收集的额外ISA支持。

Chapter 14

翻译狗 www.fanyigou.net www.translateGo.net

Chapter 15

事务性内存的“T”标准扩展，版本0.0

本章是未来标准扩展的占位符，用于提供事务内存操作。

尽管在过去二十年中进行了大量研究，并进行了初步的商业实施，但仍然存在很多关于支持涉及多个地址的原子操作的最佳方法的争论。

我们目前的想法是在原始事务内存提议中包含一个小的有限容量事务内存缓冲区。

翻译狗 www.fanyigou.net www.translateGo.net

Chapter 16

“P”标准扩展包装SIMD指令，版本0.1

在第五届RISC-V研讨会上的讨论表明，希望放弃这个针对浮点寄存器的打包SIMD提议，转而支持对大型浮点SIMD操作的V扩展进行标准化。然而，有兴趣将打包SIMD定点操作用于小型RISC-V实现的整数寄存器。

在本章中，我们概述了RISC-V的标准打包SIMD扩展。我们为未来的标准打包SIMD扩展集保留了指令子集名称“P”。许多其他扩展可以构建在压缩SIMD扩展上，利用与整数单元分开的宽数据寄存器和数据路径。

最初与林肯Labs TX-2 [9]一起推出的Packed-SIMD扩展已经成为一种在数据并行代码上提供更高吞吐量的流行方法。早期的商用微处理器实现包括Intel i860，HP PA-RISC MAX [19]，SPARC VIS [29]，MIPS MDMX [12]，PowerPC AltiVec [8]，Intel x86 MMX / SSE [24,26]，而最近的设计包括Intel x86 AVX [20]和ARM Neon [11]。我们在本章中描述了一个用于添加packed-SIMD的标准框架，但并没有积极地处理这样的设计。我们认为，在重用现有的宽数据路径资源时，压缩SIMD设计代表了一个合理的设计点，但如果要将大量额外资源用于数据并行执行，那么基于传统矢量架构的设计是更好的选择，应该使用V延期。

RISC-V压缩SIMD扩展重用浮点寄存器（f0-f31）。这些寄存器可以定义为FLEN = 32到FLEN = 1024的宽度。标准浮点指令子集需要宽度为32位（“F”），64位（“D”）或128位（“Q”）的寄存器。

将浮点寄存器用于打包SIMD值而不是整数寄存器（PA-RISC和Alpha打包SIMD扩展）是很自然的，因为它可以释放控制和地址值的整数寄存器，简化了标量浮点的重用用于SIMD浮点执行的单元，自然导致解耦的整数/浮点硬件设计。浮点加载和存储指令编码还有空间来处理更宽的压缩SIMD寄存器。但是，对打包SIMD值重用浮点寄存器会使重新编码的内部格式更难用于浮点值。

翻译狗 www.fanyigou.net www.translateGo.net

现有的浮点加载和存储指令用于从存储器向f寄存器加载和存储各种大小的字。基本ISA支持32位和64位加载和存储，但LOAD-FP和STORE-FP指令编码允许编码8种不同的宽度，如表16.1所示。当与打包SIMD操作一起使用时，希望支持非自然对齐的加载和存储在硬件中。

宽度字段	码	位大小
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16.1：LOAD-FP和STORE-FP宽度编码。

Packed-SIMD计算指令对f寄存器中的打包值进行操作。每个值可以是8位，16位，32位，64位或128位，并且可以支持整数和浮点表示。例如，64位打包SIMD扩展可将每个寄存器视为1×64位，2×32位，4×16位或8×8位打包值。

简单的压缩SIMD扩展可能适用于未使用的32位指令操作码，但更广泛的压缩SIMD扩展可能需要专用的30位指令空间。

www.fanyigou.net 翻译狗 www.translateGo.net

Chapter 17

矢量操作的“V”标准扩展，版本0.2

本章介绍了RISC-V向量指令集扩展的建议。向量扩展支持可配置的向量单元，以便根据可用的最大向量长度权衡架构向量寄存器的数量和支持的元素宽度。向量扩展旨在允许相同的二进制代码在物理向量存储容量和数据路径并行性不同的各种硬件实现中高效工作。

矢量扩展基于Seymour Cray在20世纪70年代引入的矢量寄存器架构的风格，而不是早期的封装SIMD方法，1957年林肯实验室TX-2引入，现在被大多数其他商业指令集采用。

向量指令集包含早期研究项目中开发的许多功能，包括Berkeley T0和VIRAM矢量微处理器，MIT Scale矢量线程处理器以及Berkeley Maven和Hwacha项目。

17.1 矢量单位国家

附加矢量单元架构状态包括32个矢量数据寄存器（v0-v31），8个矢量谓词寄存器（vp0-vp7）和XLEN位WARL矢量长度CSR，vl。另外，矢量单元的当前配置保持在设定矢量配置CSR（vcmaxw，vctype，vcnpred）中，如下所述。该实现确定了vcmaxw和vcnpred寄存器中保存的当前配置的可用最大向量长度（MVL）。还有一个3位定点舍入模式CSR vxrm和一位定点饱和状态CSR vxsat。

17.2 元素数据类型和宽度

V扩展支持的数据类型和操作取决于基本标量ISA和支持的扩展，可能包括8位，16位，32位，64位和128位整数和定点数据类型（X8，分别为X16，X32，X64和X128），以及16位，32位，64位和128位

翻译狗 www.fanyigou.net www.translateGo.net

CSR名称	数	基础ISA
vl	0x020	RV32, RV64, RV128
vxrm	0x020	RV32, RV64, RV128
vxsat	0x020	RV32, RV64, RV128
vcsr	0x020	RV32, RV64, RV128
vcnpred	0x020	RV32, RV64, RV128
韦克马克夫	0x020	RV32, RV64, RV128
vcmaxw1	0x020	RV32
vcmaxw2	0x020	RV32, RV64
vcmaxw3	0x020	RV32
vctype	0x020	RV32, RV64, RV128
vctype1	0x020	RV32
vctype2	0x020	RV32, RV64
vctype3	0x020	RV32
vctype 0	0x020	RV32, RV64, RV128
vctype 1	0x020	RV32, RV64, RV128
...		
vctype 31	0x020	RV32, RV64, RV128

表17.1：向量扩展CSR。

支持的定点宽度	
RV32I	X8, X16, X32
RV64I	X8, X16, X32, X64
RV128I	X8, X16, X32, X64, X128
支持的浮点宽度	
F	F16, F32
FD	F16, F32, F64
外债配额	F16, F32, F64, F128

表17.2：支持的数据元素宽度取决于基本整数ISA和支持的浮点扩展。请注意，支持给定的浮点宽度要求支持所有较窄的浮点宽度。

浮点类型（分别为F16，F32，F64和F128）。添加V扩展时，它必须支持由表17.2定义的支持的标量类型隐含的向量数据元素类型。支持的最大元素宽度：

elen = 最大值（xl，flen）

当标量和向量指令都支持任何硬件支持的数据类型时，编译器对矢量化的支持大大简化。

将向量扩展添加到具有浮点支持的任何机器都增加了对IEEE标准半精度16位浮点数据类型的支持。这包括第??节中描述的一组标量半精度指令。标量半精度指令遵循其他浮点精度的模板，但使用迄今未使用的fmt字段编码为10。

我们只支持标量半精度浮点类型作为向量扩展的一部分，如

当使用分摊每操作控制开销的向量指令时，获得半精度的主要好处。不支持单独的标量半精度浮点扩展也减少了标准指令集变体的数量。

17.3 矢量配置寄存器（vcmaxw，vctype，vcp）

必须在使用前配置矢量单元。每个架构矢量数据寄存器（v0-v31）配置有该矢量数据寄存器的每个元素中允许的最大位数，或者可以被禁用以释放其他架构矢量数据寄存器的物理矢量存储。可用矢量谓词寄存器的数量也可以独立设置。

可用的MVL取决于配置设置，但MVL对于给定实现上的相同配置参数必须始终具有相同的值。实现必须为所有支持的配置设置提供至少四个元素的MVL。

每个矢量数据寄存器的当前最大宽度保存在vcmaxw中的单独的四位字段中
CSR，编码如表17.3所示。

宽度	编码
残	0000
8	1000
16	1001
32	1010
64	1011
128	1100

表17.3：vcmaxw字段的编码。保留所有其他值。

几个早期的矢量机器能够将物理矢量寄存器存储配置为更多的短矢量或更短数量的长矢量，特别是富士通VP系列[21]。

此外，每个矢量数据寄存器都有一个相关的动态类型字段，该字段保存在vctype CSR中的四位字段中，编码如表17.4所示。向量数据寄存器的动态类型字段被约束为仅保持具有与该向量数据寄存器的对应vcmaxw字段中的值相等或更小宽度的类型。对vctype的更改不会改变MVL。

向量数据寄存器具有最大元素宽度和当前元素数据类型以支持向量函数调用，其中调用者不知道被调用者所需的类型，如下所述。

要减少配置时间，写入vcmaxw字段也会写入相应的vctype字段。vcmaxw字段可以写入表17.4中类型编码的任何值，但只有表17.3中所示的宽度信息将记录在vcmaxw字段中，而完整类型信息将记录在相应的vctype字段中。

尝试写入宽度大于实现支持的任何vcmaxw字段将引发非法指令异常。允许实现记录
vcmaxw

类型	vctype编码	vcmaxw相当于
残	0000	0000
F16	0001	1001
F32	0010	1010
F64	0011	1011
F128	0100	1100
X8	1000	1000
X16	1001	1001
X32	1010	1010
X64	1011	1011
X128	1100	1100

表17.4: vctype字段的编码。第三列显示写入vcmaxw字段时将保存的值。保留所有其他值。

值大于请求的值。特别是，实现可以选择硬连线
vcmaxw字段到支持的最大宽度。

尝试编写不受支持的类型或需要超过当前vcmaxw的类型
宽度到vctype字段将引发异常。

对vcmaxw寄存器中的字段的任何写入都配置向量单元并使所有向量数据寄存器归零并且所有向量谓词寄存器被设置，并且向量长度寄存器vl被设置为支持的最大向量长度。

对vctype字段的任何写入仅将关联的向量数据寄存器归零，而使其他向量单元状态不受干扰。尝试将需要比相应vcmaxw值更多位的类型写入vctype字段将引发非法指令异常。

向量寄存器在重新配置时归零以防止安全漏洞并避免暴露不同实现如何管理物理向量寄存器存储之间的差异。

有序实现将使用每个寄存器的标志位来进行多路复用，而不是每个源上的垃圾值，直到被覆盖为止。对于有序机器，由于预测或矢量长度小于MVL的部分写入使这种归零变得复杂，但是这些情况可以通过采用硬件读 – 修改 – 写入，每个元素添加零位或陷阱来处理 – 模式陷阱处理程序，如果配置为部分后第一次写入访问无序机器只能将初始重命名表指向物理零寄存器。

在RV128中，vcmaxw是一个包含32个4位宽度字段的CSR。位（4N + 3） – （4N）保持矢量数据寄存器N的最大宽度。在RV64中，vcmaxw2 CSR提供对vcmaxw的高64位的访问。在RV32中，vcmaxw1 CSR提供对vcmaxw的第63-32位的访问，而vcmax3 CSR提供对位127-96的访问。

vcnpred CSR包含一个4位WLRL字段，给出启用的架构谓词寄存器的数量，介于0和8之间。任何对vcnpred的写操作都将所有向量数据寄存器置零，设置可见向量谓词寄存器中的所有位，并设置向量长度寄存器vl到支持的最大矢量长度。尝试将大于8的值写入vcnpred会引发非法指令异常。

AVL价值	vl设置
AVL ≥ 2 MVL	MVL
2 MVL > AVL > MVL	1av1 DBA/ 2J
MVL ≥ AVL	AVL

表17.5: setvl指令的操作，以基于所请求的应用矢量长度（AVL）和当前最大矢量长度（MVL）来设置矢量长度寄存器vl。

17.4 矢量长度

有效向量长度保存在XLEN位WARL向量长度CSR vl中，该长度仅能保存0到MVL之间的值。对最大配置寄存器（vcmaxw或vcnpred）的任何写入都会导致vl使用MVL进行初始化。写入vctype不会影响vl。

有效向量长度通常用setvl指令写入，该指令被编码为vl CSR编号的csrrw指令。csrrw的source参数是请求的应用程序向量长度（AVL），作为无符号的XLEN位整数。setvl指令根据表17.5计算要分配给vl的值。

设置vl寄存器的规则有助于在过度循环的最后两次迭代中保持向量管道的完整性。之前在Cray设计的机器中使用过类似的规则[7]。

此计算的结果也作为setvl指令的结果返回。请注意，与常规csrrw指令不同，写入整数寄存器rd的值不是原始CSR值，而是修改后的值。

具有实现定义的向量长度的想法至少可以追溯到IBM 3090向量工具[5]，它使用特殊的“加载向量计数和更新”（VLVCU）指令来控制条带化循环。这里包含的setvl指令基于Asanovic [4]引入的更简单的setvlr指令。

setvl指令通常在stripmined循环的每次迭代开始时使用，以设置在随后的循环迭代中操作的向量元素的数量。可以通过执行具有所有位设置的源参数（最大无符号整数）的setvl来获得当前MVL。

当vl = 0时，不对任何向量指令执行元素操作。

17.5 快速配置说明

可能需要几条指令才能将vcmaxw，vctype和vcnpred设置为给定配置。为了加速配置向量单元，添加了专门的vcfg指令，这些指令被编码为对CSR的写入，其编码的立即值在vcmaxw，vctype和vncpred配置寄存器中设置多个字段。

vcfgd指令被编码为CSRRW，它采用如图17.2所示编码的寄存器值，并返回目标寄存器中的相应MVL。对应 -

```
#Vector-vector 32位添加循环。
#假设向量单元配置了正确的类型。
#a0持有N.
#a1保存指向结果向量的指针
#a2保存指向第一个源向量的指针
#a3保存指向第二个源向量的指针。环：
  setvl t0, 0
  v0 v0, a2        #加载第一个向
  量sll t1, t0,2#乘以字节加a2,
  t1               #Bump指针
  v1 v1, a3        #加载第二个向量
  添加a3, t1       #Bump指针
  vadd vadd, vadd    #添加元素
  子A0, T0         #减少元素已完成vst
       v0,a1       #存储结果向量
  添加a1, t1       #Bump指针
  bnez a0, 循环 #还有吗？
```

图17.1：示例矢量矢量添加循环。

将vcfgdi指令编码为CSRRWI，它采用5位立即值来设置配置，并在目标寄存器中返回MVL。

---

vcfgdi的主要用途之一是使用单字节元素向量配置向量单元，以便在memcpy和memset例程中使用。单个指令可以为这些操作配置向量单元。

vcfgd指令还清除vcnpred寄存器，因此不分配谓词寄存器。

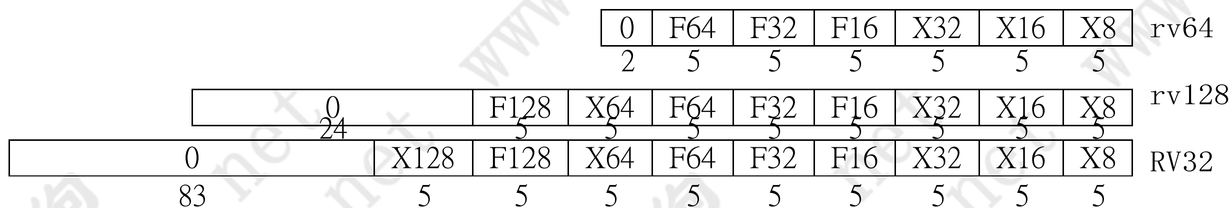


图17.2：不同基础ISA的vcfgd值格式，为每种支持的类型保存5位向量寄存器编号。字段必须包含0，表示没有为该类型分配向量寄存器，或者向量寄存器编号大于右边的全部。两个非零字段之间的所有向量寄存器编号被分配给具有较高向量寄存器编号的类型。

vcfgd值指定分配每种数据类型的向量寄存器的数量，并分为5位字段，每个受支持的数据类型一个。字段中的值0表示不分配该类型的寄存器。非零值表示最高向量

vcfgd值中的每个5位字段必须包含0，表示没有为该类型分配向量寄存器，或者向量寄存器编号大于低位位置中的所有字段，表示包含相关类型的最高向量寄存器。这种编码可以紧凑

0	F64	F32	F16	X32	X16	X8
0	18	12	0	1	0	0

矢量寄存器	韦克马克夫	vctype	类型
v31-v19	0000	0000	残
v18-v13	1011	0011	F64
v12-v2	1010	0010	F32
v1-v0	1010	1010	X32

图17.3：使用vcfgd值设置配置的示例。

表示向量寄存器到数据类型的任意分配，除了必须至少有两个向量寄存器（v0和v1）分配给最窄的所需类型。示例分配如图17.3所示。

分别使用CSRRW和CSRRWI编码提供单独的vcfgp和vcfgpi指令，这些指令将源值写入vcnpred寄存器并返回新的MVL。这些写操作还清除向量数据寄存器，设置分配的谓词寄存器中的所有位，并设置vl = MVL。在vcfgd之后可以使用vcfgp或vcfgpi指令来完成向量单元的重新配置。

如果为vcgfd赋予零参数，则将取消配置向量单元而不启用寄存器，并且将为MVL返回值0。只有配置寄存器vcmaxw和vcnpred可以在此状态下直接访问或通过vcfgd，vcfgdi，vcfgp或vcfgpi指令访问。其他向量指令将引发非法指令异常。

为了快速更改向量寄存器的各个类型，每个向量数据寄存器n都有一个专用的CSR地址来访问其vctype字段，名为vctypevn。vcfgt和vcfgti指令是常规CSRRW和CSRRWI指令的汇编程序伪指令，用于更新类型字段并返回原始值。vcfgti指令通常用于在一条指令中记录前一种类型时更改为所需类型，而vcfgt指令用于恢复为已保存的类型。

Chapter 18

“N”标准扩展
用户级中断，版本1.1

这是一个占位符，用于更完整地编写N扩展，并形成讨论的基础。

本章介绍了添加RISC-V用户级中断和异常处理的建议。当存在N扩展，并且外部执行环境已将指定的中断和异常委派给用户级时，则硬件可以将控制直接转移到用户级陷阱处理程序而不调用外部执行环境。

用户级中断主要用于支持仅存在Mmode和U模式的安全嵌入式系统，但也可以在运行类Unix操作系统的系统中支持，以支持用户级陷阱处理。

在Unix环境中使用时，用户级中断可能不会取代传统的信号处理，但可以用作进一步扩展的构建块，以生成用户级事件，例如垃圾收集障碍，整数溢出，浮点陷阱。

18.1 其他CSR

表18.1中列出了为支持N扩展而添加的用户可见CSR。

数	名称	描述
0x000	乌斯塔图斯	用户状态寄存器。
0x004	洋葱	用户中断使能寄存器。
0x005	乌特韦茨	用户陷阱处理程序基地址。
0x040	乌斯克拉奇	用户陷阱处理程序的Scratch寄存器。
0x041	乌埃普茨	用户异常程序计数器。
0x042	乌库斯	用户陷阱原因。
0x043	选择	用户不良地址或指令。
0x044	乌伊普	用户中断待处理。

表18.1：N扩展的CSR。

翻译狗 www.fanyigou.net www.translateGo.net

18.2 用户状态寄存器（ustatus）

ustatus寄存器是一个XLEN位读/写寄存器，格式如图18.1所示。该ustatus寄存器跟踪并控制哈特当前的运行状态。

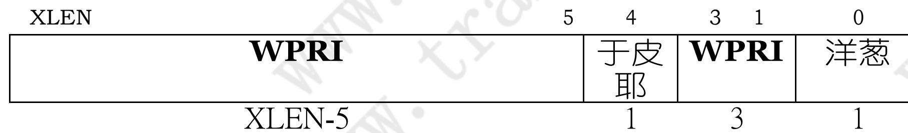


图18.1：用户模式状态寄存器（ustatus）。

用户中断使能位UIE在清零时禁用用户级中断。当采用用户级陷阱时，UIE的值被复制到UPIE中，并且UIE的值被设置为零，以便为用户级陷阱处理程序提供原子性。

没有UPP位来保持先前的权限模式，因为它只能是用户模式。

URET指令用于从U模式的陷阱返回，URET将UPIE复制到UIE，然后设置UPIE。

在弹出UPIE / UIE堆栈之后设置UPIE以启用中断并帮助捕获编码错误。

18.3 其他CSR

其余的CSR以与为M模式和S模式定义的陷阱处理寄存器类似的方式运行。

一个更完整的文章要遵循。

18.4 N扩展指令

添加URET指令以执行MRET和SRET的类似功能。

18.5 减少上下文交换开销

用户级中断处理寄存器为用户级上下文添加了相当大的状态，但在正常使用中通常很少处于活动状态。特别是，uepc，ucause和utval仅在执行陷阱处理程序时有效。

可以按照FS和XS字段的格式将NS字段添加到mstatus和sstatus，以在值不活动时减少上下文切换开销。执行URET会将uepc，ucause和utval重新置于初始状态。

Chapter 19

RV32 / 64G指令集清单

RISC-V项目的一个目标是将其用作稳定的软件开发目标。为此，我们将基本ISA（RV32I或RV64I）和选定的标准扩展（IMAFD）的组合定义为“通用”ISA，并且我们使用缩写G作为指令集扩展的IMAFD组合。本章介绍RV32G和RV64G的操作码映射和指令集列表。

研究所[4：2] 研究所[6：5]	000	001	010	011	100	101	110	111 (> 32*b*)
00	加载	LOAD-FP	*定制0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48*b*
01	商店	STORRE-FP	*定制1*	我爱你	操作	他	on-32	64*b*
10	MADD	mmub	NMSUB	NMADD	OP-FP	*保留的*	*定制-2 / rv128*	48*b*
11	科	JALR	*保留的*	日航	系统	*保留的*	*定制-3 / rv128*	≥ 80*b*

表19.1：RISC-V基本操作码映射，inst [1：0] = 11

表19.1显示了RVG主要操作码的映射。设置了3个或更多低位的主要操作码保留用于大于32位的指令长度。对于自定义指令集扩展，应避免使用标记为保留的操作码，因为未来的标准扩展可能会使用这些操作码。未来的标准扩展将避免标记为custom-0和custom-1的主要操作码，建议在基本32位指令格式中使用自定义指令集扩展。标记为custom-2 / rv128和custom-3 / rv128的操作码保留供RV128将来使用，但是对于标准扩展将被禁用，因此也可用于RV32和RV64中的自定义指令集扩展。

我们相信RV32G和RV64G为广泛的通用计算提供简单但完整的指令集。可以添加第12章中描述的可选压缩指令集（形成RV32GC和RV64GC），以提高性能，代码大小和能效，但有一些额外的硬件复杂性。

随着我们超越IMAFDC进一步扩展到指令集扩展，添加的指令往往更加特定于域，并且仅为受限类应用程序提供好处，例如，用于多媒体或安全性。与大多数商业ISA不同，RISC-V ISA设计明确区分了基础ISA和广泛适用的标准扩展，以及这些更专业的附加功能。第21章更广泛地讨论了向RISC-V ISA添加扩展的方法。

RV32I基本指令集

31 27 26 25	24 20	19 15	14 12	11 7	6 0
功能7	rs2	rs1	功能3	rd	操作码
imm[11:0]		rs1	功能3	rd	操作码
IMM [5]	rs2	rs1	功能3	imm[4:0]	操作码
imm [12:10: 5]	rs2	rs1	功能3	imm [4: 1:11]	操作码
imm[31:12]				rd	操作码
imm[31:12]				rd	0110111
imm[31:12]				rd	0010111
imm [20:10: 1: 11:19:12]				rd	1101111
imm[11:0]		rs1	000	rd	1100111
imm [12:10: 5]	rs2	rs1	000	imm [4: 1:11]	1100011
imm [12:10: 5]	rs2	rs1	001	imm [4: 1:11]	1100011
imm [12:10: 5]	rs2	rs1	100	imm [4: 1:11]	1100011
imm [12:10: 5]	rs2	rs1	101	imm [4: 1:11]	1100011
imm [12:10: 5]	rs2	rs1	110	imm [4: 1:11]	1100011
imm [12:10: 5]	rs2	rs1	111	imm [4: 1:11]	1100011
imm[11:0]		rs1	000	rd	0000011
imm[11:0]		rs1	001	rd	0000011
imm[11:0]		rs1	010	rd	0000011
imm[11:0]		rs1	100	rd	0000011
imm[11:0]		rs1	101	rd	0000011
IMM [5]	rs2	rs1	000	imm[4:0]	0100011
IMM [5]	rs2	rs1	001	imm[4:0]	0100011
IMM [5]	rs2	rs1	010	imm[4:0]	0100011
imm[11:0]		rs1	000	rd	0010011
imm[11:0]		rs1	010	rd	0010011
imm[11:0]		rs1	011	rd	0010011
imm[11:0]		rs1	100	rd	0010011
imm[11:0]		rs1	110	rd	0010011
imm[11:0]		rs1	111	rd	0010011
0000000	萨拉姆	rs1	001	rd	0010011
0000000	萨拉姆	rs1	101	rd	0010011
0100000	萨拉姆	rs1	101	rd	0010011
0000000	rs2	rs1	000	rd	0110011
0100000	rs2	rs1	000	rd	0110011
0000000	rs2	rs1	001	rd	0110011
0000000	rs2	rs1	010	rd	0110011
0000000	rs2	rs1	011	rd	0110011
0000000	rs2	rs1	100	rd	0110011
0000000	rs2	rs1	101	rd	0110011
0100000	rs2	rs1	101	rd	0110011
0000000	rs2	rs1	110	rd	0110011
0000000	rs2	rs1	111	rd	0110011
0000 预计值 苏克奇		00000	000	00000	0001111
0000 0000 0000		00000	001	00000	0001111
000000000000		00000	000	00000	1110011
000000000001		00000	000	00000	1110011
企业社会		rs1	001	rd	1110011

R型I 型S型 B型U 型J型

LUI
AUIPC
JAL
JALR
BEQ
BNE
BLT
BGE
BLTU
BGEU
LB.
LH
LW
LBU
LHU
SH
SW
ADDI
SLTI
sltiu
xori
ORI安
迪slli
srli
Srai添
加子
SLL
SLT浇
异或
SRL
SRA和
栅栏
栅栏。
我回忆
ebreak
csrrw
CSRRs
csrrc
csrrwi
csrrsi
csrrci

31 27	26 25	24 20	19 15	14 12	11 7	6 0
功能7		rs2	rs1	功能3	rd	操作码
imm[11:0]			rs1	功能3	rd	操作码
IMM [5]		rs2	rs1	功能3	imm[4:0]	操作码

10100	水性	rl	r
11000	水性	rl	r
11100	水性	rl	r

RV64I基本指令集（除RV32I外）

imm[11:0]		rs1	110	rd	0000011
imm[11:0]		rs1	011	rd	0000011
IMM [5]	rs2	rs1	011	imm[4:0]	0100011
000000	萨拉姆	rs1	001	rd	0010011
000000	萨拉姆	rs1	101	rd	0010011
010000	萨拉姆	rs1	101	rd	0010011
imm[11:0]		rs1	000	rd	0011011
0000000	萨拉姆	rs1	001	rd	0011011
0000000	萨拉姆	rs1	101	rd	0011011
0100000	萨拉姆	rs1	101	rd	0011011
0000000	rs2	rs1	000	rd	0111011
0100000	rs2	rs1	000	rd	0111011
0000000	rs2	rs1	001	rd	0111011
0000000	rs2	rs1	101	rd	0111011
0100000	rs2	rs1	101	rd	0111011

RV32M标准扩展

0000001	rs2	rs1	000	rd	0110011
0000001	rs2	rs1	001	rd	0110011
0000001	rs2	rs1	010	rd	0110011
0000001	rs2	rs1	011	rd	0110011
0000001	rs2	rs1	100	rd	0110011
0000001	rs2	rs1	101	rd	0110011
0000001	rs2	rs1	110	rd	0110011
0000001	rs2	rs1	111	rd	0110011

RV64M标准扩展（除RV32M外）

0000001	rs2	rs1	000	rd	0111011
0000001	rs2	rs1	100	rd	0111011
0000001	rs2	rs1	101	rd	0111011
0000001	rs2	rs1	110	rd	0111011
0000001	rs2	rs1	111	rd	0111011

RV32A标准扩展

00010	水性	rl	00000	rs1	010	rd	0101111
00011	水性	rl	rs2	rs1	010	rd	0101111
00001	水性	rl	rs2	rs1	010	rd	0101111
00000	水性	rl	rs2	rs1	010	rd	0101111
00100	水性	rl	rs2	rs1	010	rd	0101111
01100	水性	rl	rs2	rs1	010	rd	0101111
01000	水性	rl	rs2	rs1	010	rd	0101111
10000	水性	rl	rs2	rs1	010	rd	0101111

R型I型S型

激光焊接装置LD SD slli srli Srai
addiw slliw srliw sraiw addw地铁sllw SRLW sRAW

多多mulhsu mulhu div DIVU REM热木

多个DIVW DIVUW REMW REMUW

LR。W与W
amoswap amoadd。W。W amoxor amoand。W。W amomin Amoor。W。W amomax
amominu。W。W W amomaxu。

www.fanyigou.net 翻译狗 www.translateGo.net

31 27	26 25	24 20	19 15	14 12	11 7	6 0
功能7		rs2	rs1	功能3	rd	操作码
rs3	功能2	rs2	rs1	功能3	rd	操作码
imm[11:0]			rs1	功能3	rd	操作码
IMM［5］		rs2	rs1	功能3	imm[4:0]	操作码

R型R4
型I型
S型

RV64A标准扩展（除RV32A外）

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00010	水性	rl	00000	rs1	011	rd	0101111
00011	水性	rl	rs2	rs1	011	rd	0101111
00001	水性	rl	rs2	rs1	011	rd	0101111
00000	水性	rl	rs2	rs1	011	rd	0101111
00100	水性	rl	rs2	rs1	011	rd	0101111
01100	水性	rl	rs2	rs1	011	rd	0101111
01000	水性	rl	rs2	rs1	011	rd	0101111
10000	水性	rl	rs2	rs1	011	rd	0101111
10100	水性	rl	rs2	rs1	011	rd	0101111
11000	水性	rl	rs2	rs1	011	rd	0101111
11100	水性	rl	rs2	rs1	011	rd	0101111

LR。
D SC.
D
amoswap。D
AMOD。D
AMOXOR。D 美
国人。D 部长。
D AMOMIN。D
AMOMAX。D
AMOMOU。D
AMOMAX。(d)

RV32F标准扩展

31 27	26 25	24 20	19 15	14 12	11 7	6 0
imm[11:0]			rs1	010	rd	0000111
IMM［5］		rs2	rs1	010	imm[4:0]	0100111
rs3	00	rs2	rs1	rm	rd	1000011
rs3	00	rs2	rs1	rm	rd	1000111
rs3	00	rs2	rs1	rm	rd	1001011
rs3	00	rs2	rs1	rm	rd	1001111
0000000		rs2	rs1	rm	rd	1010011
0000100		rs2	rs1	rm	rd	1010011
0001000		rs2	rs1	rm	rd	1010011
0001100		rs2	rs1	rm	rd	1010011
0101100		00000	rs1	rm	rd	1010011
0010000		rs2	rs1	000	rd	1010011
0010000		rs2	rs1	001	rd	1010011
0010000		rs2	rs1	010	rd	1010011
0010100		rs2	rs1	000	rd	1010011
0010100		rs2	rs1	001	rd	1010011
1100000		00000	rs1	rm	rd	1010011
1100000		00001	rs1	rm	rd	1010011
1110000		00000	rs1	000	rd	1010011
1010000		rs2	rs1	010	rd	1010011
1010000		rs2	rs1	001	rd	1010011
1010000		rs2	rs1	000	rd	1010011
1110000		00000	rs1	001	rd	1010011
1101000		00000	rs1	rm	rd	1010011
1101000		00001	rs1	rm	rd	1010011
1111000		00000	rs1	000	rd	1010011

FW FSW
FMADD。S
FMSUB。S
fnmsub。S
FNMADD。S
FADD。S
FSUB。S
FMUL。S
FDIV。S
FSQRT。S
FSGNJ。S
FSGNJN。S
FSGNJX。S
FMIN。S
FMAX。S
FCVT。WS
FCVT 。 五 。
S FMV。XW
FEQ。S
FLT。 S LE。
S FCLASS 。
S FCVT.
SW FCVT。
吴国集团
WX

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
功能7		rs2	rs1	功能3	rd	操作码	R型R4
rs3	功能2	rs2	rs1	功能3	rd	操作码	型I型
imm[11:0]			rs1	功能3	rd	操作码	S型
IMM［5］		rs2	rs1	功能3	imm[4:0]	操作码	

RV64F标准扩展（除RV32F外）

1100000	00010	rs1	rm	rd	1010011
1100000	00011	rs1	rm	rd	1010011
1101000	00010	rs1	rm	rd	1010011
1101000	00011	rs1	rm	rd	1010011

fcvt。LS FCVT。路。S FCVT. SL FCVT。卢晓明

RV32D标准扩展

imm[11:0]			rs1	011	rd	0000111	FLD
IMM［5］		rs2	rs1	011	imm[4:0]	0100111	FSD
rs3	01	rs2	rs1	rm	rd	1000011	FMADD. D
rs3	01	rs2	rs1	rm	rd	1000111	FMSUB. D
rs3	01	rs2	rs1	rm	rd	1001011	FNMSUB. D
rs3	01	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. SD
0100001		00000	rs1	rm	rd	1010011	FCVT. DS
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. WD
1100001		00001	rs1	rm	rd	1010011	FCVT. WU. D
1101001		00000	rs1	rm	rd	1010011	FCVT. DW
1101001		00001	rs1	rm	rd	1010011	FCVT. D. WU

RV64D标准扩展（除RV32D外）

1100001	00010	rs1	rm	rd	1010011
1100001	00011	rs1	rm	rd	1010011
1110001	00000	rs1	000	rd	1010011
1101001	00010	rs1	rm	rd	1010011
1101001	00011	rs1	rm	rd	1010011
1111001	00000	rs1	000	rd	1010011

fcvt。LD FCVT。路。D FMV。XD FCVT。DL FCVT。D. LU FMV。DX

表19.2：RISC-V的指令列表

翻译狗 www.fanyigou.net www.translateGo.net

表19.3列出了当前已分配CSR地址的CSR。定时器，计数器和浮点CSR是本规范中定义的唯一CSR。

数	特权	名称	描述
		浮点控制和状态寄存器	
0x001 0x002 0x003	读/写读/写 读/写	弗尔夫勒弗 带进位反馈移位寄存器	浮点应计异常。浮点动态舍入模式。 浮点控制和状态寄存器（frm + fflags）。
		计数器和计时器	
0xC00 0xC01 0xC02 0xC80 0xC81 0xC82	只读只读 只读只读 只读 只读	循环时间 instret cycleh timeh 因斯特雷思	RDCYCLE指令的循环计数器。RDTIME指令的定时器。 指令 - RDINSTRET指令的退役计数器。循环的高32位，仅RV32I。 高32位时间，仅RV32I。 高位32位的instret，仅限RV32I。

表19.3：RISC-V控制和状态寄存器（CSR）地址映射。

翻译狗 www.fanyigou.net www.translateGo.net

第20章

RISC-V汇编程序员手册

本章是汇编程序员手册的占位符。

表20.1列出了x和f寄存器的汇编程序助记符及其在标准调用约定中的作用。

寄存器	ABI名称	描述	节电器
x0	零	硬接线零	—
x1	类风湿性关节炎	退货地址	呼叫者
x2	服务提供商	堆栈指针	被呼叫者
x3	gp	全局指针	—
x4	胡志明市	线程指针	—
x5	t0	临时/备用链接寄存器	呼叫者
x6-7	t1-2	临时工	呼叫者
x8	s/fp	保存的寄存器/帧指针	被呼叫者
x9	s1	保存的注册	被呼叫者
x10-11	a0-1	函数参数/返回值	呼叫者
x12-17	a2-7	函数参数	呼叫者
x18-27	s2-11	保存的寄存器	被呼叫者
x28-31	t3-6	临时工	呼叫者
f0-7	ft0-7	FP临时	呼叫者
f8-9	fs0-1	FP保存寄存器	被呼叫者
f10-11	fa0-1	FP参数/返回值	呼叫者
f12-17	fa2-7	FP参数	呼叫

www.fanyigou.net 翻译狗 www.translateGo.net

			者
f18-27	fs2−11	FP保存寄存器	被呼叫者
f28-31	ft8−11	FP临时	呼叫者

表20.1：RISC-V整数和浮点寄存器的汇编器助记符。

表20.2和20.3包含标准RISC-V伪指令的列表。

翻译狗 www.fanyigou.net www.translateGo.net

伪指令	基础指令	含义
香格里拉路，象征	auipc rd, symbol[31:12] 阿迪，RD，RD，符号[11:0]	加载地址
l{b|h|w|d} rd，符号	auipc rd, symbol[31:12] l { B | H | W | D} RD，符号[11:0] (RD)	加载全局
s{b|h|w|d} 路，象征，RT	auipc rt, symbol[31:12] { B的| H | W | D} RD，符号[11:0] (RT)	存储全球
佛罗里达州| D W 点加载全局	路，象征，RT FL { W | D} RD，符号[11:0] (RT)	auipc rt, symbol[31:12] 浮
FS{ | }D W 点存储全局	路，象征，RT FS { W | D} RD，符号[11:0] (RT)	auipc rt, symbol[31:12] 浮
诺普	迪 x0，x0，0	没有操作
立刻	*无数的序列*	立即加载
mv rd, rs	迪德，rs，0	复制寄存器
不是，rs	xori rd, rs, -1	一个补充
否定，rs	第，x0，rs	两个补码
否定，rs	sw rd，x0，rs	两个补充词
sext.w rd, rs	加法，女士，0	签署扩展词
塞茨路	sltiu RD、RS 1	设置if = 0
偷偷摸摸地说	现场采访，x0，RS	设置if = 0
SLTZ型RD、RS	SLT RD、RS，x0	设置为<零
sgtz RD、RS	slt rd, x0, rs	设置if> 0
FMV的RD、RS。	fsgnj的RD、RS，RS。	复制单精度寄存器fabs. s
rd，rs	fsgnjx的RD、RS，RS。	单精度绝对值fneg. s rd，
rs	fsgnjn的RD、RS，RS。	单精度否定
FMV。D RD、RS	fsgnj。D RD、RS，RS	复制双精度寄存器fabs. d
rd，rs	fsgnjx。D RD、RS，RS	双精度绝对值fneg. d rd，
rs	fsgnjn。D RD、RS，RS	双精度否定
beqz RS，偏移	beq rs，x0，偏移量	分支if =零
bnez rs，抵消	bne rs，x0，偏移量	分支if =零
布莱兹，偏移量	bge x0，rs，偏移量	分支如果≤零
bgez RS，偏移	BGE RS，x0，偏移	分支如果≥零
RS系列，偏移	BLT RS，x0，偏移	分支如果<零
bgtz RS，偏移	blt x0，rs，偏移量	分支如果>零
bgt rs，rt，偏置	blt rt，rs，偏置	分支如果>
瓣辭，rt，偏移量	bge rt，rs，偏置	分支如果≤
bgtu RS，RT，偏移	bltu RT、RS、偏移	分 支 如 果 >，
unsigned bleu rs，rt，offset 移	分支如果≤，无符号j偏移量	bgeu RT、RS、偏
移量	跳	jal x0，偏
日航偏移	jal x1，偏移量	跳转和链接
红外遥感	贾尔 x0, Rs 0	跳转寄存器
贾尔 rs	贾尔 x1, Rs 0	跳转和链接寄存器
对	贾尔 x0, x1, 0	从子程序返回
呼叫抵消	auipc x6，offset[31:12] jalr x1，x6，偏移[11：0]	调用远程子程序
尾部偏移	auipc x6，offset[31:12] jalr x0、x0、offset[11:0]	尾调用远程子程序
篱笆	篱笆iorw，iorw	所有内存和I / O上的栅栏

表20.2：RISC-V伪指令。

伪指令	基础教学	含义
rdinstret [H]路		克斯特罗斯， instret [h]， x0 阅
读说明 - 退休柜台		
rdcycle [H]路		CSRRs RD， 周
期[H]， x0	读周期计数器rdtime [h]	rd CSRRs
RD， 时间[H]， x0		读取实时时钟
csrr rd， csr	csrrs rd, csr, x0	阅读CSR
csrw csr, rs	csrrw x0, csr, rs	写CSR
CSR企业社会责任， RS		csrrs x0, csr, rs 在CSR中设置位
证监会的社会责任， RS		csrc x0， csr， rs 清除CSR中的位
csrwi csr, imm	csrwi x0， csr， imm	立即写CSR
csrsi csr, imm	csrsi x0， csr， imm	设置CSR中的位， 立即
csrci csr， imm	csrrci x0, csr, imm	CSR中的清除位， 立即
frcsr rd	csrrs rd, fcsr, x0	读FP控制/状态寄存器
fscsr rd， rs	csrw rd， fcsr， rs	交换FP控制/状态寄存器
fscsr rs	csrrw x0， fcsr， rs	写FP控制/状态寄存器frrm
rd	csrrs rd, frm, x0	读取FP舍入模式
FSRM RD、RS	csrw rd， frm， rs	交换FP舍入模式
fsrm rs	csrrw x0, frm, rs	写FP舍入模式
fsrmi rd, imm	csrwi rd， frm， imm	交换FP舍入模式， 立即fsrmi
imm	csrwi x0， frm， imm	写FP舍入模式， 立即frflags
rd	csrrs rd， fflags， x0	读取FP异常标志
fsflags rd, rs	弗斯特勒尔	交换FP异常标志
fsflags rs	csrrw x0， fflags， rs	写FP异常标志
fs弗拉菲西第，	伊姆山	csrrwi rd， fflags， imm 立即交换FP异常标
志		
fs弗拉菲西 imm	csrrwi x0， fflags， imm	写入FP异常标志， 立即

表20.3：访问控制和状态寄存器的伪指令。

翻译狗 www.fanyigou.net www.translateGo.net

第21章

扩展RISC-V

除了支持标准通用软件开发之外，RISC-V的另一个目标是为更专业的指令集扩展或更多定制加速器提供基础。指令编码空间和可选的可变长度指令编码旨在使构建更多自定义处理器时更容易利用标准ISA工具链的软件开发工作。例如，目的是继续为仅使用标准I库的实现提供完全软件支持，可能还有许多非标准指令集扩展。

本章介绍了可以扩展基本RISC-V ISA的各种方法，以及管理由独立组开发的指令集扩展的方案。该卷仅处理用户级ISA，尽管第二卷中描述的管理程序级扩展使用相同的方法和术语。

21.1 扩展术语

本节定义了一些用于描述RISC-V扩展的标准术语。

标准与非标准扩展

任何RISC-V处理器实现都必须支持基本整数ISA（RV32I或RV64I）。此外，实现可以支持一个或多个扩展。我们将扩展分为两大类：标准与非标准。

- 标准扩展是一种通常有用的扩展，旨在与任何其他标准扩展不冲突。目前，本手册其他章节中描述的“MAFDQLCBTPV”是完整的或计划的标准扩展。
- 非标准扩展可能是高度专业化的，可能与其他标准或非标准扩展冲突。我们预计随着时间的推移将开发各种各样的非标准扩展，其中一些最终将被提升为标准扩展。

指令编码空间和前缀

指令编码空间是一些指令位，其中编码基本ISA或ISA扩展。RISC-V支持不同的指令长度，但即使在单个指令长度内，也有各种大小的编码空间可用。例如，基本ISA定义在30位编码空间（32位指令的位31-2）内，而原子扩展“A”适合25位编码空间（位31-7）。

我们使用术语前缀来指代指令编码空间右侧的位（因为RISC-V是小端，右边的位存储在较早的存储器地址中，因此在指令获取顺序中形成前缀）。标准基本ISA编码的前缀是在32位字的位1-0中保存的两位“11”字段，而标准原子扩展“A”的前缀是七位“0101111”字段保存在代表AMO主要操作码的32位字的位6-0中。编码格式的一个怪癖是用于编码次操作码的3位funct3字段与32位指令格式中的主要操作码位不相邻，但被认为是22位指令空间的前缀的一部分。

虽然指令编码空间可以是任何大小，但采用较小的通用大小集简化了将独立开发的扩展打包成单个全局编码。表21.1给出了RISC-V的建议大小。

尺寸	用法	#提供标准指令长度			
		16位	32位	48位	64位
14位	压缩16位编码的象限	3			
22位	基本32位编码的次要操作码		2^8	2^{20}	2^{35}
25位	基本32位编码的主要操作码		32	2^{17}	2^{32}
30位	基本32位编码的象限		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位编码的主要操作码			32	2^{20}
40位	48位编码的象限			4	2^{17}
45位	64位编码的子小操作码				2^{12}
48位	64位编码中的次要操作码				2^9
52位	64位编码的主要操作码				32

表21.1：建议的标准RISC-V指令编码空间大小。

格林菲尔德与布朗菲尔德扩展

我们使用术语绿域扩展来描述开始填充新指令编码空间的扩展，因此只能在前缀级别引起编码冲突。我们使用术语棕色字段扩展来描述适合于先前定义的指令空间中的现有编码的扩展。棕色区域扩展必然与特定的绿地父编码相关联，并且可能存在对同一绿地父编码的多个棕色区域扩展。例如，基本ISA是30位指令空间的绿地编码，而FDQ浮点扩展都是棕色域扩展，增加了父基本ISA 30位编码空间。

注意，我们认为标准A扩展具有绿域编码，因为它在完整的32位基本指令编码的最左边的位中定义了一个新的先前空的25位编码空间，即使它的标准前缀位于30-基址ISA的位编码空间。仅更改其单个7位前缀可以将A扩展移动到不同的30位编码空间，同时仅担心前缀级别的冲突，而不是编码空间本身内的冲突。

	添加状态	没有新的国家
格林菲尔德	rv32i（30）、（30）rv64i	A(25)
布朗菲尔德	F(I), D(F), Q(D)	M(I)

表21.2：标准指令集扩展的二维表征。

表21.2显示了简单二维分类中的基础和标准扩展。一个轴是扩展是绿地还是棕色，而另一个轴是扩展是否添加了建筑状态。对于绿地扩展，指令编码空间的大小在括号中给出。对于棕色地带扩展，它所构建的扩展名（绿地或棕色地带）的名称在括号中给出。额外的用户级体系结构状态通常意味着对主管级系统或可能对标准调用约定的更改。

请注意，RV64I不被视为RV32I的扩展，而是不同的完整基本编码。

标准兼容的全局编码

对于实际RISC-V实现，ISA的完整或全局编码必须为每个包含的指令编码空间分配唯一的非冲突前缀。基础和每个标准扩展都有一个标准前缀，以确保它们可以全局编码共存。

标准兼容的全局编码是基础和每个包含的标准扩展具有其标准前缀的编码。标准兼容的全局编码可以包括不与所包含的标准扩展冲突的非标准扩展。如果相关的标准扩展不包括在全局编码中，则标准兼容的全局编码也可以使用非标准扩展的标准前缀。换句话说，标准扩展必须使用其标准前缀（如果包含在标准兼容的全局编码中），否则其前缀可以自由重新分配。这些约束允许通用工具链以任何RISC-V标准兼容全局编码的标准子集为目标。

保证非标准编码空间

为了支持专有自定义扩展的开发，保证编码空间的某些部分永远不会被标准扩展使用。

翻译狗 www.fanyigou.net www.translateGo.net

21.2 RISC-V扩展设计理念

我们打算通过鼓励扩展开发人员在指令编码空间内操作来支持大量独立开发的扩展，并通过提供工具将这些扩展打包成标准兼容的全局编码，方法是分配唯一的前缀。一些扩展更自然地实现为现有扩展的棕色域扩充，并将共享分配给其父绿域扩展的任何前缀。标准扩展前缀避免了核心功能编码中的虚假不兼容性，同时允许自定义打包更多深奥的扩展。

将RISC-V扩展重新打包成不同的标准兼容全局编码的这种能力可以以多种方式使用。

一个用例是开发高度专业化的自定义加速器，旨在运行来自重要应用程序域的内核。这些可能希望删除除基本整数ISA之外的所有内容，并仅添加手头任务所需的扩展。基本ISA设计用于对硬件实现提出最低要求，并且已被编码为仅使用32位指令编码空间的一小部分。

另一个用例是为新类型的指令集扩展构建研究原型。研究人员可能不想花费精力来实现可变长度的指令获取单元，因此希望使用简单的32位固定宽度指令编码来对其扩展进行原型设计。但是，这个新扩展可能太大而不能与32位空间中的标准扩展共存。如果研究实验不需要所有标准扩展，则标准兼容的全局编码可能会丢弃未使用的标准扩展并重用其前缀以将建议的扩展放置在非标准位置以简化研究原型的工程。标准工具仍然可以定位基础和任何标准扩展，以减少开发时间。一旦评估和改进了指令集扩展，就可以将其打包到更大的可变长度编码空间中，以避免与所有标准扩展冲突。

以下部分描述了使用新的指令集扩展开发实现的日益复杂的策略。这些主要用于高度定制，教育或实验性架构，而不是RISC-V ISA开发的主线。

21.3 固定宽度32位指令格式的扩展

在本节中，我们将讨论为仅支持基本固定宽度32位指令格式的实现添加扩展。

我们预计最简单的固定宽度32位编码将受到许多限制加速器和研究原型的欢迎。

可用的30位指令编码空间

在标准编码中，三个可用的30位指令编码空间（具有2位前缀00,01和10的空间）用于启用可选的压缩指令扩展。但是，如果不需要压缩指令集扩展，那么这三个另外的30位编码空间变得可用。这使32位格式的可用编码空间翻了两番。

可用的25位指令编码空间

25位指令编码空间对应于基本和标准扩展编码中的主要操作码。

有四个主要操作码明确保留用于自定义扩展（表19.1），每个操作码代表一个25位编码空间。其中两个保留用于RV128基本编码（将是OP-IMM-64和OP-64），但可用于RV32和RV64的标准或非标准扩展。

为RV64保留的两个操作码（OP-IMM-32和OP-32）也可用于RV32的标准和非标准扩展。

如果实现不需要浮点，那么为标准浮点扩展（LOAD-FP, STORE-FP, MADD, MSUB, NMSUB, NMADD, OP-FP）保留的七个主要操作码可以重用于非标准扩展。类似地，如果不需要标准原子扩展，则可以重用AMO主要操作码。

如果实现不需要超过32位的指令，则可以使用另外四个主要操作码（表19.1中以灰色标记的操作码）。

基本RV32I编码仅使用11个主要操作码和3个保留操作码，最多可用于扩展。基本RV64I编码仅使用13个主要操作码和3个保留操作码，最多可使用16个扩展。

可用的22位指令编码空间

22位编码空间对应于基本和标准扩展编码中的funct3次要操作码空间。几个主要操作码都有一个funct3字段次要操作码，它没有被完全占用，只剩下几个22位编码空间。

通常，主要操作码选择用于在指令的剩余位中编码操作数的格式，理想情况下，扩展应遵循主要操作码的操作数格式以简化硬件解码。

其他空间

在某些主要操作码下可以使用较小的空间，并非所有的小操作码都是完全填充的。

21.4 添加对齐的64位指令扩展

为基本32位固定宽度指令格式提供过大扩展空间的最简单方法是添加自然对齐的64位指令。该实现仍然必须支持32位基本指令格式，但可以要求64位指令在64位边界上对齐以简化指令获取，并在必要时使用32位NOP指令作为对齐填充。

为简化标准工具的使用，64位指令应按图1.1所述进行编码。但是，实现可能会为64位指令选择非标准指令长度编码，同时保留32位指令的标准编码。例如，如果不需要压缩指令，则可以使用指令的前两位中的一个或多个零位来编码64位指令。

我们期望生成指令获取单元的处理器生成器能够自动处理支持的可变长度指令编码的任何组合。

21.5 支持VLIW编码

尽管RISC-V并非设计为纯VLIW机器的基础，但可以使用多种替代方法将VLIW编码添加为扩展。在所有情况下，必须支持基本32位编码以允许使用任何标准软件工具。

固定大小的指令组

最简单的方法是定义单个大的自然对齐的指令格式（例如，128位），其中VLIW操作被编码。在传统的VLIW中，这种方法倾向于浪费指令存储器来保存NOP，但RISC-V兼容的实现也必须支持基本32位指令，将VLIW代码大小扩展限制为VLIW加速功能。

编码长度组

另一种方法是使用图1.1中的标准长度编码来编码并行指令组，从而允许NOP从VLIW指令中压缩出来。例如，64位指令可以保存两个28位操作，而96位指令可以保存三个28位操作，依此类推。或者，48位指令可以保存一个42位操作，而96位指令可以保存两个42位操作，依此类推。

这种方法的优点是为保持单个操作的指令保留基本ISA编码，但缺点是需要对VLIW指令内的操作进行新的28位或42位编码，以及对较大的组进行未对齐的指令获取。一种简化是不允许VLIW指令跨越某些微架构上重要的边界（例如，高速缓存行或虚拟存储器页面）。

固定大小的指令包

类似于Itanium的另一种方法是使用较大的自然对齐的固定指令束大小（例如，128位），其中并行操作组被编码。这简化了指令获取，但将复杂性转移到组执行引擎。要保持RISC-V兼容，仍然必须支持基本32位指令。

前缀中的组结束位

上述方法都不保留VLIW指令内的各个操作的RISC-V编码。另一种方法是在固定宽度32位编码中重新利用两个前缀位。如果设置，则一个前缀位可用于发信号通知“组末尾”，而如果清除，则第二位可指示谓词下的执行。由不知道VLIW扩展的工具生成的标准RISC-V 32位指令将具有前缀位设置（11），因此具有正确的语义，每个指令在组的末尾而不是预测。

这种方法的主要缺点是基本ISA缺乏在激进的VLIW系统中通常需要的复杂预测支持，并且很难增加空间以在标准30位编码空间中指定更多谓词寄存器。

第22章

ISA子集命名约定

本章描述了RISC-V ISA子集命名方案，该方案用于简明地描述硬件实现中存在的指令集，或应用程序二进制接口（ABI）使用的指令集。

RISC-V ISA旨在通过各种实验指令集扩展支持各种实现。我们发现有组织的命名方案简化了软件工具和文档。

22.1 区分大小写

ISA命名字符串不区分大小写。

22.2 基本整数ISA

RISC-V ISA字符串以RV32I，RV32E，RV64I或RV128I开头，表示基本整数ISA支持的地址空间大小（以位为单位）。

22.3 指令扩展名称

标准ISA扩展名由一个字母组成。例如，整数基数的前四个标准扩展是：“M”表示整数乘法和除法，“A”表示原子存储器指令，“F”表示单精度浮点指令，“D”表示双精度浮点指令精确浮点指令。通过将基本整数前缀与包含的扩展名相连接，可以简洁地描述任何RISC-V指令集变体。例如，“RV64IMAFD”。

我们还定义了缩写“G”来表示“IMAFD”基础和扩展，因为它旨在表示我们的标准通用ISA。

RISC-V ISA的标准扩展给出了其他保留字母，例如，“Q”表示四精度浮点，“C”表示16位压缩指令格式。

22.4 版本号

认识到指令集可能随着时间的推移而扩展或改变，我们编码子集名称后面的子集版本号。版本号分为主版本号和次版本号，用“p”分隔。如果次要版本为“0”，则可以从版本字符串中省略“p0”。主要版本号的更改意味着向后兼容性的丢失，而仅次要版本号的更改必须向后兼容。例如，本手册1.0版中定义的原始64位标准ISA可以完整地写为“RV64I1p0M1p0A1p0F1p0D1p0”，更简洁地写为“RV64I1M1A1F1D1”，或者更简洁地写为“RV64G1”。G ISA子集可以写为“RV64I2p0M2p0A2p0F2p0D2p0”，或者更简洁地写为“RV64G2”。

我们在第二版中引入了版本编号方案，我们也打算成为永久标准。因此，我们将标准子集的默认版本定义为本文档时存在的默认版本，例如，“RV32G”等同于“RV32I2M2A2F2D2”。

22.5 非标准扩展名

非标准子集使用单个“X”命名，后跟名称以字母和可选版本号开头。例如，“Xhwacha”命名为Hwacha vector-fetch ISA扩展；“Xhwacha2”和“Xhwacha2p0”的名称版本2.0相同。

非标准扩展必须通过单个下划线与其他多字母扩展分开。例如，具有非标准扩展Argle和Bargle的ISA可以命名为“RV64GXargle Xbargle”。

-

22.6 监督级指令子集

标准管理程序指令子集在第II卷中定义，但使用“S”作为前缀命名，后跟以字母和可选版本号开头的管理程序子集名称。

主管扩展必须通过单个下划线与其他多字母扩展分开。

22.7 主管级扩展

使用“SX”前缀定义超级用户级ISA的非标准扩展。

22.8 子集命名约定

表22.1总结了标准化的子集名称。

子集	名称
标准通用ISA	
整数 整数乘法和除法原子 单精度浮点 双精度浮点	I M A F D
一般	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 动态语言事务内存包 装-SIMD扩展矢量扩展 用户级中断	Q L C B J T P V N
非标准用户级扩展	
非标准扩展“abc”	徐州奥森回转支承制造有限公司
标准主管级ISA	
主管延期“def”	山西柴油机厂
非标准主管级别扩展	
主管延期“ghi”	SXghi

表22.1：标准ISA子集名称。该表还定义了子集名称必须出现在名称字符串中的规范顺序，表中的top-to-bottom指示名称字符串中的倒数第一，例如，RV32IMAFDQC是合法的，而RV32IMAFDCQ则不是。

第23章

历史和致谢

23.1 ISA手册修订版1.0的历史

RISC-V ISA和指令集手册构建了几个早期项目。监控级机器的几个方面和手册的整体格式可以追溯到1992年开始的加州大学伯克利分校和ICSI的T0（Torrent-0）矢量微处理器项目.T0是基于MIPS-II ISA的矢量处理器由Krste Asanovi'c担任主要设计师和RTL设计师，Brian Kingsbury和Bertrand Irrisou担任主要VLSI实施者。ICSI的David Johnson是T0 ISA设计的主要贡献者，特别是管理员模式和手册文本。John Hauser还就T0 ISA设计提供了大量反馈。

MIT的规模（低能耗软件控制架构）项目始于2000年，基于T0项目基础设施，改进了管理程序级接口，并通过删除分支延迟槽从MIPS标量ISA移开。Ronny Krashinsky和Christopher Batten是麻省理工学院Scale Vector-Thread处理器的主要架构师，而Mark Hampton则为Scale提供了基于GCC的编译器基础架构和工具。

T0 MIPS标量处理器规范（MIPS-6371）的轻微编辑版本用于教授2002年秋季学期的新版MIT 6.371 VLSI系统入门课程，Chris Terman和Krste Asanovic担任讲师。Chris Terman贡献了该课程的大部分实验材料（没有TA！）。6.371课程演变为麻省理工学院的6.884复杂数字设计课程，由Arvind和Krste Asanovi'c于2005年春季授课，成为常规的春季课程6.375。在6.884 / 6.375中使用了基于Scale MIPS的标量ISA的简化版本，名为SMIPS。Christopher Batten是这些课程早期课程的助教，并开发了大量基于SMIPS ISA的文档和实验材料。TA Yunsup Lee针对由John Wawrzynek，Krste Asanovic和John Lazzaro教授的加州大学伯克利分校2009年秋季CS250 VLSI系统设计课程改编和增强了相同的SMIPS实验室材料。

Maven（Malleable Array of Vector-thread ENgines）项目是第二代矢量线程架构。它的设计由克里斯托弗•巴滕（Christopher Batten）领导，他于2007年夏季开始在加州大学伯克利分校担任交流学者。来自日立的访问工业研究员Hidetaka Aoki对早期的Maven ISA和微架构设计提供了大量反馈。Maven

翻译狗 www.fanyigou.net www.translateGo.net

基础架构基于Scale基础架构，但Maven ISA进一步远离Scale中定义的MIPS ISA变体，具有统一的浮点和整数寄存器文件。Maven旨在支持替代数据并行加速器的实验。Yunsup Lee是各种Maven矢量单元的主要实现者，而Rimas Avizienis是各种Maven标量单元的主要实现者。Yunsup Lee和Christopher Batten将GCC移植到新的Maven ISA。Christopher Celio提供了Maven传统矢量指令集（“Flood”）变体的初始定义。

根据以往所有项目的经验，RISC-V ISA定义于2010年夏季开始。在2012年加州大学伯克利分校CS250 VLSI系统设计课程中使用RISC-V 32位指令子集的初始版本，并使用Yunsup李作为TA。RISC-V与早期的MIPS灵感设计完全不同。John Hauser为浮点ISA定义做出了贡献，包括符号注入指令和允许内部重新编码浮点值的寄存器编码方案。

23.2 ISA手册修订版2.0的历史

已经完成了RISC-V处理器的多种实现，包括几种硅制造，如图23.1所示。

名称	流片日期	处理	ISA
乌鸦-1	2011年5月29日	st 28 纳米 fdsoi	rv64g1 xwacha1
EOS14	2012年4月1日	IBM的45nm SOI	RV64G1p1 Xhwacha2
EOS16	2012年8月17日	IBM的45nm SOI	RV64G1p1 Xhwacha2
乌鸦-2	2012年8月22日	st 28 纳米 fdsoi	RV64G1p1 Xhwacha2
EOS18	2013年2月6日	IBM的45nm SOI	RV64G1p1 Xhwacha2
EOS20	2013年7月3日	IBM的45nm SOI	RV64G1p99 Xhwacha2
乌鸦3	2013年9月26日	ST 28nm SOI	RV64G1p99 Xhwacha2
EOS22	2014年3月7日	IBM的45nm SOI	RV64G1p9999 Xhwacha3

表23.1：制造的RISC-V测试芯片。

第一批RISC-V处理器采用Verilog编写，采用ST生产的28 nm FDSOI技术，于2011年作为Raven-1测试芯片制造。两个内核由Yunsup Lee和Andrew Waterman开发，由Krste Asanovic建议，一起制造：1）具有错误检测触发器的RV64标量核心，以及2）具有附加的64位浮点矢量单元的RV64核心。第一个微体系结构被非正式地称为“TrainWreck”，因为用不成熟的设计库完成设计的时间很短。

随后，安德鲁●沃特曼，Rimas Avizienis和Yunsup Lee（由Krste Asanovi'c建议）开发了一个清洁的微架构，用于有序解耦的RV64核心，继续铁路主题后，代号为“Rocket”的乔治●斯蒂芬森成功的蒸汽之后机车设计。Rocket是用Chisel编写的，Chisel是加州大学伯克利分校开发的一种新的硬件设计语言。Rocket中使用的IEEE浮点单元由John Hauser，Andrew Waterman和Brian Richards开发。Rocket已经进一步改进和开发，并且已经在28 nm FDSOI（Raven-2，Raven-3）中制造了两次，在IBM 45 nm SOI技术（EOS14，EOS16，EOS18，EOS20，EOS22）中制造了5次用于光子学项目。工作正在进行中

火箭设计可用作参数化RISC-V处理器发生器。

EOS14-EOS22芯片包括早期版本的Hwacha，一种64位IEEE浮点矢量单元，由Yunsup Lee，Andrew Waterman，Huy Vo，Albert Ou，Quan Nguyen和Stephen Twigg开发，由Krste Asanovi'c建议。EOS16-EOS22芯片包括具有缓存一致性协议的双核，由Henry Cook和Andrew Waterman开发，由Krste Asanovi'c建议。EOS14芯片已成功运行在1.25 GHz。EOS16芯片遭受了IBM pad库中的错误。EOS18和EOS20已成功运行在1.35 GHz。

Raven测试芯片的贡献者包括Yunsup Lee，Andrew Waterman，Rimas Avizienis，Brian Zimmer，Jaehwa Kwak，Ruzica Jevti'c，Milovan Blagojevi'c，Alberto Puggelli，Steven Bailey，Ben Keller，Pi-Feng Chiu，Brian Richards，Borivoje Nikolic和Krste Asanovi'c。

EOS测试芯片的贡献者包括Yunsup Lee，Rimas Avizienis，Andrew Waterman，Henry Cook，Huy Vo，Daiwei Li，Chen Sun，Albert Ou，Quan Nguyen，Stephen Twigg，Vladimir Sto-janovic和Krste Asanovic。

Andrew Waterman和Yunsup Lee开发了C ++ ISA模拟器“Spike”，用作开发中的黄金模型，并以用于庆祝美国横贯大陆铁路完工的金色尖峰命名。Spike已作为BSD开源项目提供。

Andrew Waterman完成了硕士论文，初步设计了RISC-V压缩指令集[33]。

RISC-V的各种FPGA实现已经完成，主要作为Par Lab项目研究撤退的集成演示的一部分。最大的FPGA设计有3个缓存一致的RV64IMA处理器，运行研究操作系统。FPGA实现的贡献者包括Andrew Waterman，Yunsup Lee，Rimas Avizienis和Krste Asanovi'c。

RISC-V处理器已在加州大学伯克利分校的几个班级中使用。Rocket被用于2011年秋季的CS250作为课程项目的基础，Brian Zimmer作为TA。对于2012年春季的本科CS152课程，Christopher Celio使用Chisel编写了一套教育RV32处理器，名为“Sodor”，位于“Thomas the Tank Engine”和朋友们居住的岛屿之后。该套件包括一个微芯片核心，一个非流水线核心，以及2,3和5级流水线核心，并在BSD许可下公开提供。该套件随后在2013年春季更新并再次在CS152中使用，Yunsup Lee作为TA，2014年春季，Eric Love担任TA。Christopher Celio还开发了一种无序RV64设计，称为BOOM（伯克利无序机器），并附带管道可视化，用于CS152类。CS152类还使用了Andrew Waterman和Henry Cook开发的Rocket核心的缓存一致版本。

2013年夏天，RoCC（Rocket Custom Coprocessor）界面被定义为简化向Rocket核心添加自定义加速器的过程。Rocket和RoCC接口广泛用于Jonathan Bachrach教授的2013年秋季CS250 VLSI课程，其中有几个学生加速器项目建立在RoCC界面上。Hwacha向量单元已被重写为RoCC协处理器。

两位伯克利本科生，Quan Nguyen和Albert Ou已经成功移植Linux，以便在2013年春季在RISC-V上运行。

2014年1月, Colin Schmidt成功完成了RISC-V 2.0的LLVM后端。2014年3月,

Bluespec的Darius Rad为GCC端口提供了软浮动ABI支持.John Hauser提供了浮

点分类指令的定义。

我们了解其他几个RISC-V核心实现, 包括Tommy Thorn在Verilog中的一个, 以及Rishiyur Nikhil在Bluespec中的一个。

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对ISA 2.0规范草案的评论。

23.3 修订版2.1的历史

自2014年5月引入冻结版本2.0以来, RISC-V ISA的使用速度非常快, 在此类短期历史记录中记录的活动太多。也许最重要的单项活动是2015年8月成立非营利性RISC-V基金会。基金会现在将接管官方RISC-V ISA标准的管理, 官方网站riscv.org是最好的获取RISC-V标准的新闻和更新。

致谢

感谢Scott Beamer, Allen J. Baum, Christopher Celio, David Chisnall, Paul Clayton, Palmer Dabbelt, Jan Gray, Michael Hamburg和John Hauser对2.0版规范的评论。

23.4 修订版2.2的历史致谢

感谢Jacob Bachmeyer, Alex Bradbury, David Horner, Stefan O'Rear和Joseph Myers对2.1版规范的评论。

23.5 资金

RISC-V架构和实施的开发部分由以下赞助商资助。

- Par Lab：由Microsoft（奖励#024263）和英特尔（奖励#024894）资助并通过UC Discovery（奖励#DIG07-10227）资助的研究。其他支持来自Par Lab附属公司诺基亚，NVIDIA，甲骨文和三星。
- 项目Isis：DoE奖DE-SC0003624。
- ASPIRE实验室：DARPA PERFECT计划，奖项HR0011-12-2-0016。DARPA POEM计划奖HR0011-11-C-0100。未来架构研究中心（C-FAR），由半导体研究公司资助的STARnet中心。ASPIRE工业赞助商，英特尔和ASPIRE附属公司，谷歌，惠普企业，华为，诺基亚，NVIDIA，甲骨文和三星的额外支持。

本文的内容不一定反映美国政府的立场或政策，也不应推断出官方认可。

参考书目

[1] RISC-V ELF psABI规范。https://github.com/riscv/riscv-elf-psabi-doc/.

[2] IEEE标准的32位微处理器。IEEE标准1754-1994,1994。

[3] GM Amdahl, GA Blaauw和Jr. FP Brooks。IBM System / 360的体系结构。*IBM Journal of R.&D., 8 (2), 1964。*

[4] Krste Asanovi'c。矢量微处理器。博士论文, 加州大学伯克利分校, 1998年5月。可用作技术报告UCB / CSD-98-1014。

[5] W. Buchholz。IBM System / 370矢量架构。IBM Systems Journal, 25 (1) : 51-62,1986。

[6] 编辑Werner Buchholz。规划计算机系统: Project Stretch。McGraw-Hill Book Company, 1962。

[7] Cray Inc. *用于Cray X1系统的Cray汇编语言 (CAL) 参考手册, 1.1版, 2003年6月。*

[8] K. Diefendorff, PK Dubey, R。Hochsprung和H. Scale。AltiVec对PowerPC的扩展加速了媒体处理。IEEE Micro, 20 (2) : 85-95,2000。

[9] John M. Frankovich和H. Philip Peterson。林肯TX-2计算机的功能描述。在西部联合计算机会议上, 加利福尼亚州洛杉矶, 1957年2月。

[10] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta和John Hennessy。可扩展共享内存多处理器中的内存一致性和事件排序。在“第17届计算机体系结构年度国际研讨会论文集”, 第15-26页, 1990年。

[11] J. Goodacre和AN Sloss。并行性和ARM指令集架构。计算机, 38 (7) : 42 - 50, 2005。

[12] Linley Gwennap。Digital, MIPS增加了多媒体扩展。微处理器报告, 1996年。

[13] Timothy H. Heil和James E. Smith。选择性双路径执行。技术报告, 威斯康星大学麦迪逊分校, 1996年11月。

[14] ANSI / IEEE Std 754-2008, IEEE浮点运算标准, 2008。

[15] Manolis GH Katevenis, Robert W. Sherburne, Jr., David A. Patterson和Carlo H. Sequin。RISC II微架构。在Proceedings VLSI 83 Conference, 1983年8月。

[16] Hyesoon Kim, Onur Mutlu, Jared Stark和Yale N. Patt。愿望分支：结合条件分支和自适应预测执行的预测。在第38届IEEE / ACM国际微体系结构研讨会论文集，MICRO 38，第43-54页，2005年。

[17] A. Klauser，T。Austin，D。Grunwald和B. Calder。非预测指令集架构的动态吊床预测。在1998年国际并行结构和编译技术会议论文集，PACT '98，华盛顿特区，美国，1998年。

[18] David D. Lee，Shing I. Kong，Mark D. Hill，George S. Taylor，David A. Hodges，Randy H. Katz和David A. Patterson。用于多处理器工作站的VLSI芯片组 - 第I部分：具有协处理器接口并支持符号处理的RISC微处理器。IEEE JSSC，24（6）：1688-1698，1989年12月。

[19] 李李。与MAX-2的子字并行。IEEE Micro，16（4）：51-59，1996年8月。

[20] 克里斯罗蒙特。英特尔高级矢量扩展简介。英特尔白皮书，2011年。

[21] Kenichi Miura和Keiichiro Uchida。FACOM矢量处理器系统：VP-100 / VP-200。在Kawalik，编辑，北约高速研究高级研究讲习班，第F7卷。Springer-Verlag，1984。同样刊登：IEEE Tutorial Supercomputers：Design and Applications。Kai Hwang（编辑），第59-73页。

[22] OpenCores的。OpenRISC 1000架构手册，架构版本1。0，2012年12月。

[23] David A. Patterson和Carlo H. S'equin。RISC I：精简指令集VLSI计算机。在ISCA，第443-458页，1981年。

[24] A. Peleg和U. Weiser。MMX技术扩展到英特尔架构。IEEE Micro，16（4）：42 - 50，1996年8月。

[25] Ravi Rajwar和James R. Goodman。推测锁省略：启用高度并发的多线程执行。在第34届年度ACM / IEEE国际微体系结构研讨会论文集，MICRO 34，第294-305页。IEEE计算机学会，2001年。

[26] SK Raman，V。Pentkovski和J. Keshava。在Pentium-III处理器上实现流式SIMD扩展。IEEE Micro，20（4）：47 -57,2000。

[27] Balaram Sinharoy，R。Kalla，WJ Starke，HQ Le，R。Cargnoni，JA Van Norstrand，
BJ Ronchetti，J。Stuecheli，J。Leenstra，GL Guthrie，DQ Nguyen，B。Blaner，CF Marino，E。Retter和P. Williams。IBM POWER7多核服务器处理器。IBM Journal of Research and Development，55（3）：1-1,2011。

[28] 詹*姆斯E.桑顿。控制数据6600中的并行操作。在1964年10月27日至29日的秋季联合计算机会议论文集，第二部分：超高速计算机系统，AFIPS '64（秋季，第二部分），第33-40页，1965年。*

[29] M. Tremblay，JM O'Connor，V。Narayanan和梁鹤。VIS加速了新媒体处理。
*I*EEE Micro，16（4）：10-20，1996年8月。

[30] Marc Tremblay，Jeffrey Chan，Shailender Chaudhry，Andrew W. Conigliaro和Shing Sheung Tse。MAJC架构：并行性和可伸缩性的综合。IEEE Micro，20（6）：12-25,2000。

[31] J. Tseng和K. Asanovi'c。节能的注册访问。在Proc。第13届集成电路与系统设计研讨会，第377-384页，巴西马瑙斯，2000年9月。

[32] David Ungar, Ricki Blau, Peter Foley, Dain Samples和David Patterson。SOAR架构：RISC上的Smalltalk。在ISCA，188-197页，Ann Arbor，MI，1984。

[33] 安德鲁沃特曼。利用RISC-V压缩提高能效并减小代码尺寸。硕士论文，加州大学伯克利分校，2011年。

[34] 安德鲁沃特曼。RISC-V指令集架构的设计。博士论文，加州大学伯克利分校，2016年。

[35] Andrew Waterman, Yunsup Lee, David A. Patterson和Krste Asanovi'c。RISC-V指令集手册，第I卷：基本用户级ISA。技术报告UCB / EECS-2011-62，EECS Department, University of California, Berkeley, 2011年5月。

[36] Andrew Waterman, Yunsup Lee, David A. Patterson和Krste Asanovic。RISC-V指令集手册，第I卷：基本用户级ISA版本2.0。技术报告UCB / EECS-2014-54，EECS部门，加州大学伯克利分校，2014年5月。