# RISC-V 指令集手册

## 第 I 卷：非特权指令集架构

文档版本 20191214-*draft*

编者：安德鲁 · 沃特曼 [1]，克尔斯泰 · 阿桑诺维奇 [1,2]

[1]SiFive 股份有限公司，

[2] 加州伯克利分校，电子工程部，计算机科学与技术系

waterman@eecs.berkeley.edu, krste@berkeley.edu

2022 年 11 月 25 日

本规范的所有版本的贡献者如下，以字母顺序排列（请联系编者以提出更改建议）：阿文，克尔斯泰 · 阿桑诺维奇，里马斯 · 阿维齐尼斯，雅各布 · 巴赫迈耶，克里斯托弗 · F · 巴顿，艾伦 · J · 鲍姆，亚历克斯 · 布拉德伯里，斯科特 · 比默，普雷斯顿 · 布里格斯，克里斯托弗 · 塞利奥，张传华，大卫 · 奇斯纳尔，保罗 · 克莱顿，帕默 · 达贝尔特，肯 · 多克瑟，罗杰 · 埃斯帕萨，格雷格 · 福斯特，谢克德 · 弗勒，斯特凡 · 弗洛伊德伯格，马克 · 高希尔，安迪 · 格鲁，简 · 格雷，迈克尔 · 汉伯格，约翰 · 豪瑟，戴维 · 霍纳，布鲁斯 · 霍尔特，比尔 · 赫夫曼，亚历山大 · 琼诺，奥洛夫 · 约翰森，本 · 凯勒，大卫 · 克鲁克迈尔，李云燮，保罗 · 洛文斯坦，丹尼尔 · 卢斯蒂格，雅廷 · 曼尔卡，卢克 · 马兰杰，玛格丽特 · 马托诺西，约瑟夫 · 迈尔斯，维贾亚南德 · 纳加拉扬，里希尔 · 尼希尔，乔纳斯 · 奥伯豪斯，斯特凡 · 奥雷尔，欧伯特，约翰 · 奥斯特豪特，大卫 · 帕特森，克里斯托弗 · 普尔特，何塞 · 雷诺，乔希 · 谢德，科林 · 施密特，彼得 · 苏厄尔，萨米特 · 萨卡尔，迈克尔 · 泰勒，韦斯利 · 特普斯特拉，马特 · 托马斯，汤米 · 索恩，卡罗琳 · 特里普，雷 · 范德瓦尔克，穆拉里达兰 · 维贾亚拉加万，梅根 · 瓦克斯，安德鲁 · 沃特曼，罗伯特 · 沃森，德里克 · 威廉姆斯，安德鲁 · 赖特，雷诺 · 赞迪克，和张思卓。



引用请使用：“The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-*draft*”，编者：安德鲁 · 沃特曼、克尔斯泰 · 阿桑诺维奇，RISC-V 国际，2019 年 12 月。

# 前言

本文描述了 RISC-V 非特权架构。

当前，标记为“被批准”的指令集架构模块，已经被官方批准了。在被提交批准之前，标记为“冻结”的模块，预计不会有重大改变。在被批准之前，标记为“草案”的模块，预计还会有所改变。

本文包含以下版本的 RISC-V ISA 模块：

| 基础模块 | 版本 | 状态 |
|---|---|---|
| RVWMO | 2.0 | **被批准** |
| **RV32I** | **2.1** | **被批准** |
| **RV64I** | **2.1** | **被批准** |
| *RV32E* | *1.9* | 草案 |
| *RV128I* | *1.7* | 草案 |
| 拓展模块 | 版本 | 状态 |
| **M** | **2.0** | **被批准** |
| **A** | **2.1** | **被批准** |
| **F** | **2.2** | **被批准** |
| **D** | **2.2** | **被批准** |
| **Q** | **2.2** | **被批准** |
| **C** | **2.0** | **被批准** |
| *Counters* | *2.0* | 草案 |
| *L* | *0.0* | 草案 |
| *B* | *0.0* | 草案 |
| *J* | *0.0* | 草案 |
| *T* | *0.0* | 草案 |
| *P* | *0.2* | 草案 |
| *V* | *0.7* | 草案 |
| **Zicsr** | **2.0** | **被批准** |
| **Zifencei** | **2.0** | **被批准** |
| **Zihintpause** | **2.0** | **被批准** |
| *Zihintntl* | *0.2* | 草案 |
| *Zam* | *0.1* | 草案 |
| **Zfh** | **1.0** | **被批准** |
| **Zfhmin** | **1.0** | **被批准** |
| **Zfinx** | **1.0** | **被批准** |
| **Zdinx** | **1.0** | **被批准** |
| **Zhinx** | **1.0** | **被批准** |
| **Zhinxmin** | **1.0** | **被批准** |
| **Zmmul** | **1.0** | **被批准** |
| *Ztso* | *0.1* | 冻结 |

# 对基于已批准的 20191213 版本文档的前言

本文档描述了 RISC-V 非特权架构。

当前，标记为“被批准”的指令集架构模块，已经被官方批准了。在被提交批准之前，标记为“*冻结*”的模块，预计不会有重大改变。在被批准之前，标记为“*草案*”的模块，预计还会有所改变。

本文包含以下版本的 RISC-V ISA 模块：

| 基础模块 | 版本 | 状态 |
|---|---|---|
| RVWMO | 2.0 | **被批准** |
| **RV32I** | **2.1** | **被批准** |
| **RV64I** | **2.1** | **被批准** |
| *RV32E* | *1.9* | 草案 |
| *RV128I* | *1.7* | 草案 |
| 拓展模块 | 状态 | 状态 |
| **M** | **2.0** | **被批准** |
| **A** | **2.1** | **被批准** |
| **F** | **2.2** | **被批准** |
| **D** | **2.2** | **被批准** |
| **Q** | **2.2** | **被批准** |
| **C** | **2.0** | **被批准** |
| *Counters* | *2.0* | 草案 |
| *L* | *0.0* | 草案 |
| *B* | *0.0* | 草案 |
| *J* | *0.0* | 草案 |
| *T* | *0.0* | 草案 |
| *P* | *0.2* | 草案 |
| *V* | *0.7* | 草案 |
| **Zicsr** | **2.0** | **被批准** |
| **Zifencei** | **2.0** | **被批准** |
| *Zam* | *0.1* | 草案 |
| *Ztso* | *0.1* | 冻结 |

此版本文档中的变动包括：

- 现在是 2.1 版本的拓展模块 A，已经在 2019 年 12 月被理事会批准。
- 定义了大端序的 ISA 变体。
- 把用于用户模式中断的 N 拓展模块移入到卷 II 中。

- 定义了暂停提示指令（PAUSE hint instruction）。

## 对基于已批准的 20190608 版本文档的前言

本文档描述了 RISC-V 非特权指令集架构。

此时，RVWMO 内存模型已经被批准了。当前，标记为“被批准”的指令集架构模块，已经被官方批准了。在被提交批准之前，标记为“冻结”的模块，预计不会有重大改变。在被批准之前，标记为“草案”的模块，预计还会有所改变。

本文包含以下版本的 RISC-V ISA 模块：

| 基础模块 | 版本 | 状态 |
|---|---|---|
| RVWMO | 2.0 | **被批准** |
| **RV32I** | **2.1** | **被批准** |
| **RV64I** | **2.1** | **被批准** |
| *RV32E* | *1.9* | 草案 |
| *RV128I* | *1.7* | 草案 |
| 拓展模块 | 版本 | 状态 |
| **Zifencei** | **2.0** | **被批准** |
| **Zicsr** | **2.0** | **被批准** |
| **M** | **2.0** | **被批准** |
| *A* | *2.0* | 冻结 |
| **F** | **2.2** | **被批准** |
| **D** | **2.2** | **被批准** |
| **Q** | **2.2** | **被批准** |
| **C** | **2.0** | **被批准** |
| *Ztso* | *0.1* | 冻结 |
| *Counters* | *2.0* | 草案 |
| *L* | *0.0* | 草案 |
| *B* | *0.0* | 草案 |
| *J* | *0.0* | 草案 |
| *T* | *0.0* | 草案 |
| *P* | *0.2* | 草案 |
| *V* | *0.7* | 草案 |
| *N* | *1.1* | 草案 |
| *Zam* | *0.1* | 草案 |

此版本文档中的变化包括：

- 将在 2019 年初被理事会批准的 ISA 模块的描述，更正为“**被批准**”的。
- 从批准的模块中移除 A 扩展。
- 变更文档版本方案，以避免与 ISA 模块的版本冲突。
- 把基础整数 ISA 的版本号增加到 2.1，以反映：被批准的 RVWMO 内存模型的出现，和先前基础 ISA 中的 FENCE.I、计数器和 CSR 指令的去除。
- 把 F 扩展和 D 扩展的版本号增加到 2.2，以反映：版本 2.1 更改了规范的 NaN [1]；而版本 2.2 定义了 NaN 装箱（NaN-Boxing，意为 NaN 的表示方式）方案，并更改了 FMIN 和 FMAX 指令的定义。
- 将文档的名字改为“非特权的”指令，以此作为将 ISA 规范从平台相关的文件中分离的行动之一。
- 为执行环境、硬件线程[2]、陷入（traps）和内存访问添加了更清晰和更精确的定义。
- 定义了指令集的种类：标准的, 保留的, 自定义的, 非标准的, and 非符合的。
- 移除了隐含着交替字节序操作的相关文本，因为交替字节序操作还没有被 RISC-V 所定义。
- 修改了对未对齐的 load 和 store 行为的描述。规范现在允许执行环境接口中对未对齐地址陷入进行可见的处理，而不是仅仅在用户模式中授权对未对齐的加载和存储进行不可见处理。而且，现在允许报告有关未对齐访问（包括原子访问）的访问异常，而不是仅仅模拟。
- 把 FENCE.I 从强制性的基础模块中移出，编入一个独立的扩展，名为 Zifencei ISA。FENCE.I 曾经被从 Linux 用户 ABI 中去除，它在实现大型非一致性指令和数据缓存时是有问题的。然而，它仍然是仅有的标准的取指一致性机制。
- 去除了禁止 RV32E 和其它扩展一起使用的约束。
- 去除了平台相关的约束条款，即，在 RV32E 和 RV64I 章节中，特定编码会产生非法指令异常
- 计数器/计时器指令现在不被认为是强制性的基础 ISA 的一部分，因此 CSR 指令被移动到独立的章节并被标记为 2.0 版本，同时非特权计数器被移动到另一个独立的章节。计数器由于存在明显的问题（包括计数不精确等），所以还没有准备批准。
- 添加了 CSR 有序访问模型。
- 为 2 位 *fmt* 域中的浮点指令明确地定义了 16 位半精度浮点格式。
- 定义了 FMIN.*fmt* 和 FMAX.*fmt* 的有符号零行为，并改变了它们遇到 NaN 信号（Signaling-NaN）输入时的行为，以符合建议的 IEEE 754-201x 规范中的 minimumNumber 和 maximumNumber 操作规范。
- 定义了内存一致性模型 RVWMO。
- 定义了“Zam”扩展，它允许未对齐的 AMO 并指定它们的语义。

[1] 译者注：NaN 是 Not a Number 的缩写，是计算机科学中数值数据类型的一类值，表示未定义或不可表示的值，常在浮点数运算中使用，参见 IEEE 754-1985 浮点数标准

[2] 译者注：硬件线程 hart 是 RISC-V 引入的一个新概念，具体含义请参考 RISC-V 规范的第 II 卷“特权 ISA”

- 定义了“Ztso”扩展，它执行比 RVWMO 更加严格的内存一致性模型。
- 改善了描述和注释。
- 定义了术语 IALIGN，作为描述指令地址对齐约束的简写。
- 去除了 P 扩展章节的内容，因为它现在已经被活跃的任务组文档所取代。
- 去除了 V 扩展章节的内容，因为它现在已经被独立的向量扩展草案文档所代替。

## 对 2.2 版本文档的前言

这是文档的 2.2 版本，描述了 RISC-V 的用户级架构。文档包括 RISC-V ISA 模块的如下版本：

| 基础模块 | 版本 | 草案被冻结? |
|---|---|---|
| RV32I | 2.0 | 是 |
| RV32E | 1.9 | 否 |
| RV64I | 2.0 | 是 |
| RV128I | 1.7 | 否 |
| 拓展模块 | 版本 | 被冻结? |
| M | 2.0 | 是 |
| A | 2.0 | 是 |
| F | 2.0 | 是 |
| D | 2.0 | 是 |
| Q | 2.0 | 是 |
| L | 0.0 | 否 |
| C | 2.0 | 是 |
| B | 0.0 | 否 |
| J | 0.0 | 否 |
| T | 0.0 | 否 |
| P | 0.1 | 否 |
| V | 0.7 | 否 |
| N | 1.1 | 否 |

到目前为止，此标准还没有任何一部分得到 RISC-V 基金会的官方批准，但是上面标记有“被冻结”标签的组件在批准处理期间，除了解决规范中的模糊不清和漏洞以外，预计不会再有变化。

此版本文档的主要变更包括：

- 此文档的先前版本是最初的作者在知识共享署名 4.0 国际许可证下发布的，当前版本和未来的版本将在相同的许可证下发布
- 重新安排了章节，把所有的扩展按规范次序排列。
- 改进了描述和注释。
- 修改了关于 JALR 的隐式提示的建议，以支持 LUI/JALR 和 AUIPC/JALR 配对的更高效的宏操作融合。
- 澄清了关于加载-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。
- 澄清了 `fcsr` 高位的作用和行为。
- 改正了对 FNMADD.*fmt* 和 FNMSUB.*fmt* 指令的描述，它们曾经给出了错误的零结果的符号。
- 指令 FMV.S.X 和 FMV.X.S 的语义没有变化，但是为了和语义更加一致，它们被分别重新命名为 FMV.W.X 和 FMV.X.W。旧名字仍将继续被工具支持。
- 规定了在较宽的 `f` 寄存器中，使用 NaN 装箱模型保存（小于 FLEN）的浮点值的行为。
- 定义了 FMA 的异常行为（$\infty$, 0, qNaN）。
- 添加注释指出，P 扩展可能会为了使用整数寄存器进行定点操作，而被重新写入一个整数 SMID（packed-SIMD）方案。
- 一个 V 向量指令集扩展的草案。
- 一个 N 用户级陷入扩展的早期草案。
- 扩充了伪指令列表。
- 移除了调用规约章节，它已经被 RISC-V ELF psABI 规范 [2] 所代替。
- C 扩展已经被冻结，并被重新编号为 2.0 版本。

## 对 2.1 版本文档的前言

这是文档的 2.1 版本，描述了 RISC-V 用户级架构。注意被冻结的 2.0 版本的用户级 ISA 基础和扩展 IMAFDQ 比起本文档的先前版本 [27] 还没有发生变化，但是一些规范漏洞已经被修复，文档也被完善了。一些软件的约定已经发生了改变。

- 为评注部分做了大量补充和改进。
- 分割了各章节的版本号。
- 修改为大于 64 位的长指令编码，以避免在非常长的指令格式中移动 *rd* 修饰符。
- CSR 指令现在用基础整数格式来描述，并在此引入了计数寄存器，而不只是稍后在浮点部分（和相应的特权架构手册）中引入。

- SCALL 和 SBREAK 指令已经被分别重命名为 ECALL 和 EBREAK。它们的编码和功能没有变化。
- 澄清了浮点 NaN 的处理，并给出了一个新的规范的 NaN 值。
- 澄清了浮点到整数溢出时的返回值。
- 澄清了 LR/SC 所允许的成功和必要的失败，包括压缩指令在序列中的使用。
- 一个新的基础 ISA 提案 RV32E，用于减少整数寄存器的数目，它支持 MAC 扩展。
- 一个修正的调用约定。
- 为软浮点调用约定放松了栈对齐，并描述了 RV32E 调用约定。
- 一个 1.9 版本的 C 压缩扩展的修正提案。

## 对 2.0 版本文档的前言

这是用户 ISA 规范的第二次发布，而我们试图让基础用户 ISA 和通用扩展（例如，IMAFD）在未来的发展中保持固定。这个 ISA 规范从 1.0 版本 [26] 开始，已经有了如下改变：

- 将 ISA 划分为一个整数基础模块和一些标准扩展模块。
- 重新编排了指令格式，让立即编码更加高效。
- 基础 ISA 按小字节序内存体系定义，而把大字节序或双字节序作为非标准的变体。
- 加载-保留/存储-条件（LR/SC）指令已经加入到原子指令扩展中。
- AMO 和 LR/SC 可以支持释放一致性（release consistency）模型。
- FENCE 指令提供更细粒度的内存和 I/O 排序。
- 为 fetch-and-XOR（AMOXOR）添加了一个 AMO，并修改了 AMOSWAP 的编码来为它腾出空间。
- 用 AUIPC 指令（它向 `pc` 加上一个 20 位的高位立即数）取代了 RDNPC 指令（它只读取当前的 `pc` 值）。这帮助我们显著节省了位置无关的代码。
- JAL 指令现在已经被移动到 U-Type 格式，它带有明确目的寄存器；J 指令被弃用，由 *rd*=`x0` 的 JAL 代替。这样去掉了仅有的一条目的寄存器不明确的指令，也把 J-Type 指令格式从基础 ISA 中移除。这虽然减少了 JAL 的适用范围，但是会明显减少基础 ISA 的复杂性。
- 关于 JALR 指令的静态提示已经被丢弃。对于符合标准调用约定的代码，这些提示与 *rd* 和 *rs1* 寄存器的修饰符放在一起都是多余的。
- 现在，JALR 指令在计算出目标地址之后，清除了它的最低位，以此来简化硬件、以及允许把辅助信息存储在函数指针中。
- MFTX.S 和 MFTX.D 指令已经被分别重命名为 FMV.X.S 和 FMV.X.D。类似地，MXTF.S 和 MXTF.D 指令也已经分别被重命名为 FMV.S.X 和 FMV.D.X。

- MFFSR 和 MTFSR 指令已经被分别重命名为 FRCSR 和 FSCSR。添加了 FRRM、FSRM、FRFLAGS 和 FSFLAGS 指令来独立地访问 `fcsr` 的两个子域：舍入模式和异常标志位。
- FMV.X.S 和 FMV.X.D 指令现在从 *rs1* 获得它们的操作数，而不是 *rs2* 了。这个变化简化了数据通路的设计。
- 添加了 FCLASS.S 和 FCLASS.D 浮点分类指令。
- 采纳了一种更简单的 NaN 生成和传播方案。
- 对于 RV32I，系统性能计数器已经被扩展到 64 位宽，且对于高 32 位和低 32 位分开进行读取访问。
- 定义了规范的 NOP 和 MV 编码。
- 为 48 位、64 位和 64 以上位指令定义了标准指令长度编码。
- 添加了 128 位地址空间的变体——RV128 的描述。
- 32 位基础指令格式中的主要操作码已经被分配给了用户自定义的扩展。
- 改正了一个笔误：建议存储从 *rd* 获得它们的数据，已经更正为从 *rs2* 获取。

# 目录

# 第一章　介绍

RISC-V（发音“risk-five”）是一个新的指令集架构（ISA），它原本是为了支持计算机架构的研究和教育而设计的，但是我们现在希望它也将成为一种用于工业实现的、标准的、免费和开放的架构。我们在定义 RISC-V 方面的目标包括：

- 一个完全开放的 ISA，学术界和工业界可以免费获得它。
- 一个真实的 ISA，适用于直接的原生的硬件实现，而不仅仅是进行模拟或二进制翻译。
- 一个对于特定微架构样式（例如，微编码的、有序的、解耦的、乱序的）或者实现技术（例如，全定制的、ASIC、FPGA）而言，避免了“过度的架构设计”（译者注：避免采用大而全的复杂微架构，超出了需求），但在这些的任何一个中都能高效实现的 ISA。
- 一个 ISA 被分成两个部分：1、一个小型基础整数 ISA，其可以用作定制加速器或教育目的的基础；2、可选的标准扩展，用于支持通用目的的软件环境。
- 支持已修订的 2008 IEEE-754 浮点标准 [8]。
- 一个支持广泛 ISA 扩展和专用变体的 ISA。
- 32 位和 64 位地址空间的变体都可以用于应用程序、操作系统内核、和硬件实现。
- 一个支持高度并行的多核或众核实现（包括异构多处理器）的 ISA。
- 具有可选的可变长度指令，可以扩展可用的指令编码空间，以及支持可选的稠密指令编码，以提升性能、静态编码尺寸和能效。
- 一个完全虚拟化的 ISA，以便简化监控器（Hypervisor）的开发。
- 一个使新特权架构上的实验被简化的 ISA。

---

我们设计决定的注释将采用像本段这样的格式。如果读者只对规范本身感兴趣，这种非正规的文本可以跳过。

---

选用 *RISC-V* 来命名，是为了表示 *UC* 伯克利设计的第五个主要的 *RISC ISA*（前四个是 *RISC-I [16]*、*RISC-II [9]*、*SOAR [23]* 和 *SPUR [12]*）。我们也用罗马字母“*V*”双关表示“变种（*variations*）”和“向量（*vectors*）”，因为，支持包括各种数据并行加速器在内的广泛的架构研究，是此 *ISA* 设计的一个明确的目标。

RISC-V ISA 的设计，尽可能地避免了实现的细节（尽管注解包含了一些由实现所驱动的决策）；它应当作为具有许多种实现的软件可见的接口来阅读，而不是作为某一特定硬件的定制品的设计来阅读。RISC-V 手册的结构分为两卷。这一卷覆盖了基本的非特权 *(unprivileged)* 指令的设计，包括可选的非特权 ISA 扩展。非特权指令是那些在所有权限架构的所有权限模式中，都能普遍可用的指令，不过其行为可能随着权限模式和权限架构而变化。第二卷提供了起初的（“经典的”）特权架构的设计。手册使用 IEC 80000-13:2008 约定，每个字节有 8 位。

---

在非特权 *ISA* 的设计中，我们尝试去除任何依赖于特定微架构的特征，例如高速缓存的行（*cache line*）大小，或者特权架构的细节，例如页转换（*page translation*）。这既是为了简化，也是为选择各种可能的微架构，或各种可能的特权架构保持最大程度的灵活性。

## 1.1 RISC-V 硬件平台术语

一个 RISC-V 硬件平台可以包含：一个或多个兼容 RISC-V 的处理核心（译者注：后续简称为 RISC-V 兼容核心或 RISC-V 核心）与其它不兼容 RISC-V 的核心、固定功能的加速器、各种物理内存结构、I/O 设备，和一个允许各组件通信的互联结构。

如果某个组件包含了一个独立的取指单元，那么它被称为一个核心。一个兼容 RISC-V 兼容的核心可以通过多线程，支持多个 RISC-V 兼容的硬件线程（hardware thread，或称为 hart）。

RISC-V 核心可以有额外的专用指令集扩展，或者一个附加的协处理器（*coprocessor*）。我们使用术语“协处理器（*coprocessor*）”来指代被接到 RISC-V 核心的单元。其大部分时候顺序执行 RISC-V 指令流，但其还包含了额外的架构状态和指令集扩展，并且可能保有与主 RISC-V 指令流相关的一些有限的自主权（译者注：这里指来自协处理器的、独立于主指令流的指令流）。

我们使用术语“加速器（*accelerator*）”来指代一个不可编程的固定功能单元，或者一个虽然能自主操作但是专用于特定任务的核心。在 RISC-V 系统中，我们希望可编程加速器都基于 RISC-V、带有专用指令集扩展和/或定制协处理器的核心。RISC-V 加速器的一个重要类别是 I/O 加速器，它为主应用核心分担了 I/O 处理任务的负载。

一个 RISC-V 硬件平台在系统级别的组织多种多样，范围可以从一个单核心微控制器到一个有数千节点（其中每个节点又都是一个共享内存的众核服务器）的集群。甚至小型片上系统都可能具有多层的多计算机和/或多处理器的结构，以使开发工作模块化，或者提供子系统间的安全隔离。

## 1.2 RISC-V 软件执行环境和硬件线程

一个 RISC-V 程序的行为依赖于它所运行的执行环境。RISC-V 执行环境接口（execution environment interface，EEI）定义了：程序的初始状态、环境中的硬件线程的数量和类型（包括被硬件线程支持的特权模式）、内存和 I/O 区域的可访问性和属性、执行在各硬件线程上的所有合法指令的行为（例如，ISA 就是 EEI 的一个组件），以及在包括环境调用在内的执行期间，任何中断或异常的处理。EEI 的例子包括了 Linux 应用程序二进制接口（ABI），或者 RISC-V 管理级（译者注：我们建议把 supervisor 翻译成监管器，后续的 hypervisor 翻译成超级监管器）二进制接口（SBI）。一个 RISC-V 执行环境的实现可以是纯硬件的、纯软件的、或者是硬件和软件的组合。例如，操作码陷入和软件模拟可以被用于实现硬件里没有提供的功能。执行环境的实际例子包括：

- “裸机”（Bare metal）硬件平台：硬件线程直接通过物理处理器线程实现，指令对物理地址空间有完全访问权限。这个硬件平台定义了一个从加电复位开始的执行环境。
- RISC-V 操作系统：通过将用户级硬件线程多路复用到可用的物理处理器线程上，以及通过虚拟内存来控制对内存的访问，提供了多个用户级别的执行环境。
- RISC-V 监管器（Hypervisor）：为宾客（guest）操作系统提供了多个监管器级别（supervisor-level）的执行环境。
- RISC-V 模拟器（RISC-V emulator）：例如 Spike、QEMU 或 rv8，它们在一个底层 x86 系统上模拟 RISC-V 硬件线程，并提供一个用户级别的或者监管器级别的执行环境。

---

可以考虑将一个裸的硬件平台定义为一个执行环境接口（*EEI*），它由可访问的硬件线程、内存、和其它设备来构成环境，且初始状态是加电复位时的状态。通常，大多数软件被设计为使用更抽象的接口，因为 *EEI* 越抽象，它所提供的跨不同硬件平台的可移植性越好。*EEI* 经常是一层叠着一层的，一个较高层的 *EEI* 使用另一个较低层的 *EEI*。

从软件在给定执行环境中运行的观点看，硬件线程是一种资源，它在执行环境中自动地获取和执行 RISC-V 指令。在这个方面，硬件线程的行为像是一种硬件线程资源——即使被执行环境时分多路复用到了真实的硬件上。一些 EEI 支持额外硬件线程的创建和销毁，例如，通过环境调用来派生新的硬件线程。

执行环境有义务确保它的各个硬件线程的最终向前推进（forward progress）。当硬件线程在执行要明确等待某个事件的机制（例如本规范第二卷中定义的 wait-for-interrupt 指令）时，该责任被挂起；当硬件线程终止时，该责任结束。硬件线程的向前推进是由下列事件构成的：

- 一个指令的引退（retirement）。
- 一个陷入，就像 1.6节中定义的那样。

- 由组成向前推进的扩展所定义的任何其它事件。

---

术语“硬件线程（*hart*）”的引入是在 *Lithe [14, 15]* 上的工作中，是为了提供一个表示一种抽象的执行资源的术语，作为与软件线程编程抽象的对应。

硬件线程（*hart*）与软件线程上下文之间的重要区别是：运行在执行环境中的软件不负责引发执行环境的各硬件线程的推进；那是外部执行环境的责任。因此，从执行环境内部软件的观点看，环境的 *hart* 的操作就像硬件的线程一样。

一个执行环境的实现可能将一组宾客硬件线程（*guest hart*），时间多路复用到由它自己的执行环境提供的更少的宿主硬件线程（*host hart*）上，但是这种做法必须以一种“宾客硬件线程像独立的硬件线程那样操作”的方式进行。特别地，如果宾客硬件线程比宿主硬件线程更多，那么执行环境必须有能力抢占宾客硬件线程，而不是必须无限等待宾客硬件线程上的宾客软件来“让步（*yield*）”对宾客硬件线程的控制。

## 1.3 RISC-V ISA 概览

RISC-V ISA 被定义为一个基础的整数 ISA（在任何实现中都必须出现）和一些对基础 ISA 的可选的扩展。基础整数 ISA 非常类似于早期的 RISC 处理器，除了没有分支延迟槽，但支持可选的变长指令编码。“基础”是被小心地限制在足以为编译器、汇编器、链接器、和操作系统（带有额外特权操作）提供合理目标的一个最小的指令集合的范围内，提供了一个便捷的 ISA 和软件工具链“骨架”，可以围绕它们来构建更多定制的处理器 ISA。

尽管使用“RISC-V ISA”这个词汇很方便，但其实 RISC-V 是一系列相关 ISA 的 ISA 族，族中目前有四个基础 ISA。每个基础整数指令集由不同的整数寄存器宽度、对应的地址空间尺寸和整数寄存器数目作为特征。在第 二章和第 七章描述了两个主要的基础整数变体，RV32I 和 RV64I，它们分别提供了 32 位和 64 位的地址空间。我们使用术语“XLEN”来指代一个整数寄存器的位宽（32 或者 64 位）。第 六章描述了 RV32I 基础指令集的子集变体：RV32E，它已经被添加来支持小型微控制器，具有一半数目的整数寄存器。第 八章概述了基础整数指令集的一个未来变体 RV128I，它将支持扁平的 128 位地址空间（XLEN = 128）。基础整数指令集使用补码来表示有符号的整数值。

---

尽管 *64* 位地址空间是更大的系统的需求，但我们相信在接下来的数十年里，*32* 位地址空间仍然适合许多嵌入式和客户端设备，并有望能够降低内存流量和能量消耗。此外，*32* 位地址空间对于教育目的是足够的。更大的扁平 *128* 位地址空间，也许最终会需要，因此我们要确保它能被容纳到 *RISC-V ISA* 框架之中。

---

*RISC-V* 中的四个基础 *ISA* 被作为不同的基础 *ISA* 对待。一个常见的问题是，为什么没有一个单一的 *ISA*？甚至特别地，为什么 *RV32I* 不是 *RV64I* 的一个严格的子集？一些早期的 *ISA*

设计（*SPARC*、*MIPS*）为了支持已有的 *32* 位二进制在新的 *64* 位硬件上运行，在增加地址空间大小的时候就采用了严格的超集策略。

明确地将基础 *ISA* 分离的主要优点在于，每个基础 *ISA* 可以按照自己的需求而优化，而不需要支持其他基础 *ISA* 需要的所有操作。例如，*RV64I* 可以忽略那些只有 *RV32I* 才需要的、处理较窄寄存器的指令和 *CSR*。*RV32I* 变体则可以使用那些在更宽地址空间变体中需要留给指令的编码空间。

没有作为单一 *ISA* 设计的主要缺点是，它使在一个基础 *ISA* 上模拟另一个时所需的硬件复杂化（例如，在 *RV64I* 上模拟 *RV32I*）。然而，地址和非法指令陷入方面的不同总体上意味着，在任何时候（即使是完全的超集指令编码），硬件也将需要进行一些模式的切换；而不同的 *RISC-V* 基础 *ISA* 是足够相似的，支持多个版本的成本相对较低。虽然有些人已经提出，严格的超集设计将允许将遗留的 *32* 位库链接到 *64* 位代码，但是由于软件调用约定和系统调用接口的不同，即使是兼容编码，这在实践中也是不实际的。

*RISC-V* 权限架构提供了 `misa` 中的域，用以在各级别控制非特权 *ISA*，来支持在相同的硬件上模拟不同的基础 *ISA*。我们注意到，较新的 *SPARC* 和 *MIPS ISA* 修订版已经弃用不经改变就在 *64* 位系统上支持运行 *32* 位代码了。

一个相关的问题是，为什么 *32* 位加法对于 *RV32I*（*ADD*）和 *RV64I*（*ADDW*）有不同的编码？*ADDW* 操作码应当被用于 *RV32I* 中的 *32* 位加法，而 *ADDD* 应当被用于 *RV64I* 中的 *64* 位加法，而不是像现有设计这样，将相同的操作码 *ADD* 用于 *RV32I* 中的 *32* 位加法和 *RV64I* 中的 *64* 位加法、却将一个不同的操作码 *ADDW* 用于 *RV64I* 中的 *32* 位加法。这也将与在 *RV32I* 和 *RV64I* 中对 *32* 位加载使用相同的 *LW* 操作码的做法保持一致性。*RISC-V ISA* 的最早版本的确有这种替代的设计，但是在 *2011* 年 *1* 月，*RISC-V* 的设计变成了如今的选择。我们的关注点在于在 *64* 位 *ISA* 中支持 *32* 位整数，而不在于提供对 *32* 位 *ISA* 的兼容性；并且动机是消除 *RV32I* 中，并非所有操作码都有“**W*”后缀所引起的不对称性（例如，有 *ADDW*，但是 *AND* 没有 *ANDW*）。事后来看，同时设计两个 *ISA*，而不是先设计一个再于其上追加设计另一个，作为如此做法的结果，这可能是不合适的；而且，出于我们必须把平台的需求折进 *ISA* 规范之中的信条，那意味着在 *RV64I* 中将需要所有的 *RV32I* 的指令。虽然现在改变编码已经太晚了，但是由于上述原因，这也几乎没有什么实际后果。

我们也能够将 **W* 变体作为 *RV32I* 系统的一个扩展启用，以提供一种跨 *RV64I* 和未来 *RV32* 变体的常用编码

RISC-V 已经被设计为支持广泛的定制和专用化。每个基础整数 ISA 可以加入一个或多个可选的指令集进行扩展。一个扩展可以被归类为标准的、自定义的，或者不合规的。出于这个目的，我们把每个 RISC-V 指令集编码空间（和相关的编码空间，例如 CSR）划分为三个不相交的种类：标准、保留、和自定义。标准扩展和编码由 RISC-V 国际定义；任何不由 RISC-V 国际定义的扩展都是非标准的的。每个基础 ISA 及其标准扩展仅使用标准编码，并且在它们使用这些编码时不能相互冲突。保留的编码当前还没有被定义，是省下来用于未来的标准扩展的；一旦如此使用，它们将变为标准编码。自定义编码应当永远不被用于标准扩展，而是可用于特定供应商的非标准扩展。非标准扩展或者是仅使用自定义编码的自定义扩展，或者是使用了任何标准或保留编码的非合规的扩展。指令集扩展一般是共享的，但是根据基础 ISA 的不同，也可能提供稍微不同的功能。第 二十

七章描述了扩展 RISC-V ISA 的各种方法。我们也已经为基于 RISC-V 的指令和指令集开发了一套命名约定，那将在第 二十八章进行详细的描述。

为了支持更一般的软件开发，RISC-V 定义了一组标准扩展来提供整数乘法/除法、原子操作、和单精度与双精度浮点运算。基础整数 ISA 被命名为“I”（根据整数寄存器的宽度配以“RV32”或“RV64”的前缀），它包括了整数运算指令、整数加载、整数存储、和控制流指令。标准整数乘法和除法扩展被命名为“M”，并添加了对整数寄存器中的值进行乘法和除法的指令。标准原子指令扩展（用“A”表示）添加了对内存进行原子读、原子修改、和写内存的指令，用于处理器间的同步。标准单精度浮点扩展（表示为“F”）添加了浮点寄存器、单精度运算指令，和单精度的加载和存储。标准双精度浮点扩展（表示为“D”）扩展了浮点寄存器，并添加了双精度运算指令、加载、和存储。标准“C”压缩指令扩展为通常的指令提供了较窄的 16 位形式。

在基础整数 ISA 和这些标准扩展之外，我们相信很少还会有新的指令对所有应用都将提供显著的益处，尽管它也许对某个特定的领域很有帮助。随着对能效的关注迫使更加的专业化，我们相信简化一个 ISA 规范中所必需的部分是很重要的。尽管其它架构通常把它们的 ISA 视为一个单独的实体，这些 ISA 随着时间的推移、指令的添加，而变成一个新的版本；RISC-V 则将努力保持基础和各个标准扩展自始至终的恒定性，新的指令改为作为未来可选的扩展分层。例如，不管任何后续的扩展如何，基础整数 ISA 都将继续作为独立的 ISA 被完全支持。

## 1.4 内存

一个 RISC-V 硬件线程有共计 $2^{\mathrm{XLEN}}$ 字节的单字节可寻址空间，可用于所有的内存访问。内存的一个“**字**（*word*）”被定义为32 位 (4 字节)。对应地，一个“**半字**（*halfword*）”是16 位 (2 字节)，一个“**双字**（*doubleword*）”64 位 (8 字节)，而一个“**四字**（*quadword*）”是128 位 (16 字节)。内存地址空间是环形的，所以位于地址 $2^{\mathrm{XLEN}}-1$ 的字节与位于地址零的字节是相邻的。因此，硬件进行内存地址计算时，忽略了溢出，代之以按模 $2^{\mathrm{XLEN}}$ 环绕。

执行环境决定了硬件资源到硬件线程地址空间的映射。一个硬件线程的地址空间可以有不同地址范围，它可以是（1）空白的，或者（2）包含**主内存**，或者（3）包含一个或多个 *I/O* **设备**。I/O 设备的直接读写会造成可见的副作用，但是访问主内存不会。虽然执行环境有可能把硬件线程地址空间中的所有内容都称作 I/O 设备，但是通常都会把某些部分指定为主内存。

当一个 RISC-V 平台有多个硬件线程时，任意两个硬件线程的地址空间可以是完全相同的，或者完全不同的，或者可以有部分不同但共享资源的一些子集，而这些资源被映射到相同或不同的地址范围。

对于一个纯粹的"裸机"环境，所有的硬件线程可以看到一个完全相同的地址空间，完全由物理地址进行访问。然而，当执行环境包含了带有地址转换的操作系统，通常会给每个硬件线程一个虚拟的地址空间，此空间很大程度上、或者完全就是线程自己的。

执行每个 RISC-V 机器指令涉及了一次或多次内存访问，这进一步可划分为隐式和显式访问。对于每个被执行的指令，进行一次隐式内存读（指令获取）是为了获得已编码指令进行执行。许多 RISC-V 指令在指令获取之外不再进一步地访问内存。在由该指令决定的地址处，有专门的加载指令和存储指令对内存进行显式的读或写。执行环境可能要求指令执行除了非特权 ISA 所文档化的访问之外的其他隐式内存访问（例如进行地址转换）。

执行环境决定了各种内存访问操作可以访问非空地址空间的哪些部分。例如，可以被取指操作隐式读到的位置集合，可能与那些可以被加载（load）指令操作显式读到的位置集合有交叠；以及，可以被存储（store）指令操作显式写到的位置集合，可能只是能被读到的位置集合的一个子集。通常，如果一个指令尝试访问的内存位于一个不可访问的地址处，将因为该指令引发一个异常。地址空间中的空白位置总是不可访问的。

除非特别说明，否则，不引发异常的隐式读可能会任意提前地、试探地发生，甚至是在机器能够证明的确需要读之前发生。例如，一种合法实现方式是，可能会尝试第一时间读取所有的主内存，缓存尽可能多的可获取（可执行）字节以供之后的指令获取，以及避免为了指令获取而再次读主内存（译者注：即通常所说的指令预取）为了确保某些隐式读只在写入相同内存位置之后是有序的，软件必须执行为此目的而定义的、特定的屏障指令或缓存控制指令（例如第 三章里定义的 FENCE.I 指令）。

由一个硬件线程发起的内存访问（隐式或显式），在被另一个硬件线程、或者任何其它可访问相同内存的代理线程所感知时，可能看起来像是以一种不同的顺序发生的。然而，这个被感知到的内存访问重新排序总是受到特定的内存一致性模型的约束。用于 RISC-V 的默认的内存一致性模型是 RISC-V 弱内存排序（RVWMO），定义在第 十七章和附录中。也可以采用更强的模型：全存储排序（Total Store Ordering），定义在第 二十五章中。执行环境也可以添加约束，进一步限制的可感知的内存访问的重排。由于 RVWMO 模型是被任何 RISC-V 实现所允许的最弱的模型，用这个模型写出的软件兼容所有 RISC-V 实现的实际的内存一致性规则。与隐式读一样，除非假定的内存一致性模型和执行环境有其他特别需求，否则软件必须执行屏障或缓存控制指令来确保特定顺序的内存访问。

## 1.5 基础指令长度编码

基础 RISC-V ISA 有固定长度的 32 位指令，必须在 32 位边界上自然地对齐。然而，标准 RISC-V 编码策略被设计为支持具有可变长度指令的 ISA 扩展指令，每条指令在长度上可以是任意

数目的 16 位指令的封装包（*parcel*），指令封装包在 16 位边界自然对齐。第 十八章中描述的标准压缩 ISA 扩展（译者注：即 C 扩展）减少了代码尺寸，通过提供压缩的 16 位指令，以及放松了对齐的限制，允许所有的指令（16 位和 32 位）在任意 16 位边界上对齐，而提升了代码的密度。

我们使用术语“IALIGN”（以位为单位）来表示实现层面所采用的指令空间对齐约束。在基础 ISA 中，IALIGN 是 32 位。但是在某些 ISA 扩展中，包括在压缩 ISA 扩展中，将 IALIG 是宽松的 16 位。IALIGN 不能取除了 16 和 32 以外的任何其它值。

我们使用术语“ILEN”（以位为单位）来表示实现层面所支持的最大指令长度，它总是 IALIGN 的倍数。对于只支持一个基础指令集的实现，ILEN 是 32 位。支持更长指令的具体实现架构也就有更大的 ILEN 值。

图 1.1描绘了标准 RISC-V 指令长度编码约定。基础 ISA 中的所有的 32 位指令都把它们的最低二位设置为“11”。而可选的压缩 16 位指令集扩展，它们的最低二位等于“00”、“01”、或“10”。

### 拓展的指令长度编码

32 位指令编码空间的一部分已经被初步分配给了长度超过 32 位的指令。目前这片空间的整体是被保留的，而且下面的关于超过 32 位编码的提议并没有被认为已被冻结。

带有超过 32 位编码的标准指令集扩展将额外的若干低序位设置为 1（即图 1.1中的 bbb=111，关于 48 位和 64 位长度的约定如图 1.1所示。指令长度在 80 位到 176 位之间的，使用 [14:12] 中 3 位（即图 1.1中的 nnn）来编码，并给出除最先的 5×16 位字（80 位字）以外的 16 位的字的数目（即图 1.1中的 nnn 的实际值）。位 [14:12] 被设置为“111”的编码被保留，用于未来更长的指令编码。

---

考虑到压缩格式的代码尺寸和节能效果，我们希望在 *ISA* 编码策略中构建对压缩格式的支持，而不是事后才想起添加它；但是为了允许更简单的实现，我们不想强制规定压缩格式。我们也希望允许更长的指令，以支持一些实验和更大的指令集扩展。尽管我们这种编码约定要求更严格的核心 *RISC-V ISA* 编码，但是这样做收益良多。

一个标准 *IMAFD ISA* 的实现只需要在指令缓存中持有最主要的 *30* 位（节省了 *6.25%*）。在指令缓存重新填充时，任何遇到有低位被清除的指令，应当在存进缓存之前，重新编码为非法的 *30* 位指令，以保持非法指令异常的行为。

也许更重要的是，通过把我们的基础 *ISA* 凝炼成 *32* 位指令字的子集，我们为非标准的和自定义的指令集扩展留出了更多可用的空间。特别地，基础 *RV32I ISA* 在 *32* 位指令字中使用少于 *1/8* 的编码空间。正如第二十七章中描述的那样，一个不需要支持标准压缩指令扩展的实现，可以将 *3* 个额外的不一致的 *30* 位指令空间映射到 *32* 位固定宽度格式，同时保留对 $\geq$*32* 位标准指令集扩展的支持。甚至，如果实现层面也不需要长度 $>$*32* 位的指令，它可以把这些不一致扩展恢复成另外四种主要的操作码。

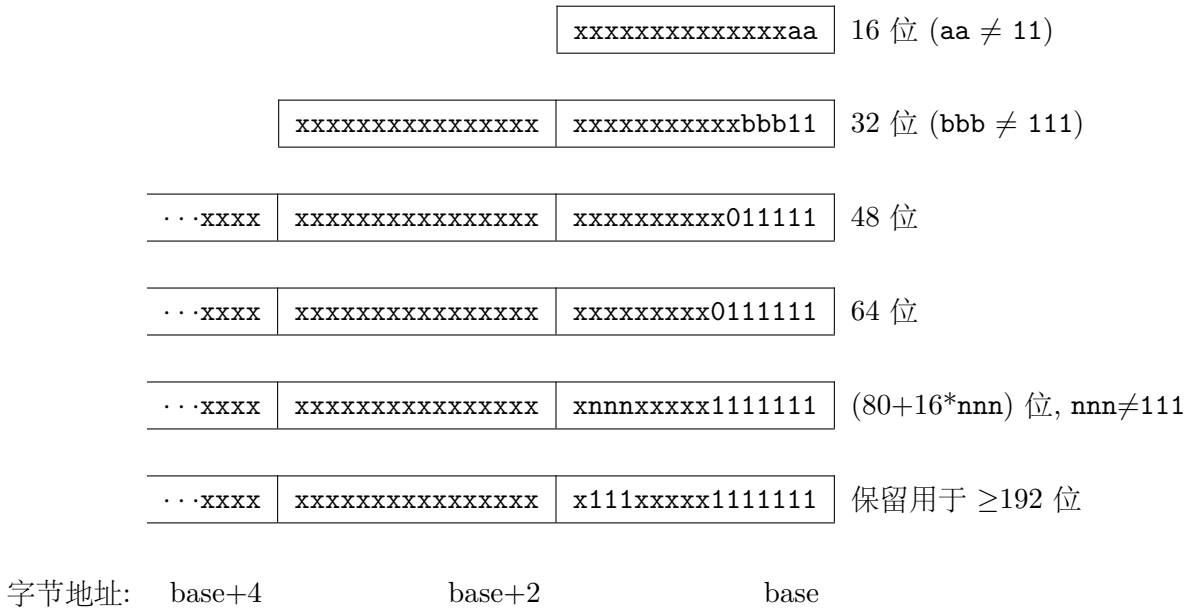



图 1.1: RISC-V 指令长度编码。当前只有 16 位和 32 位编码已被冻结。

位 [15:0] 都是 0 的编码被定义为非法指令。如果存在任何 16 位指令集扩展，则这些指令被认为具有最小的长度：16 位，否则是 32 位。位 [ILEN-1:0] 都是 1 的编码也是非法的；这个指令的长度被认为是 ILEN 位。

---

我们认为有一个特征是，所有位都是“*0*”的任意长度的指令都是不合法的，因为这很快会让陷入处理错误地跳转到零内存区域。类似地，我们也保留了包含所有“*1*”的指令编码作为非法指令，以捕获在无编程的非易失性内存设备、断连的内存总线、或者断开的内存设备上通常观测到的出错模式。

在所有的 *RISC-V* 实现上，软件可以依靠将一个包含“*0*”的自然对齐的 *32* 位字作为一个非法指令，以供明确需要非法指令的软件使用。由于可变长度编码，定义一个相应的全是“*1*”的已知非法值是更加困难的。软件不能一般地使用 *ILEN* 位全是“*1*”的非法值，因为软件可能不知道最终的目标机器的 *ILEN* （例如，如果软件被编译为一个用于许多不同机器的标准二进制库）。我们也考虑了定义一个全是“*1*”的 *32* 位字作为非法指令，因为所有的机器必须支持 *32* 位指令尺寸，但是这需要在 *ILEN>32* 的机器上的指令获取单元报告一个非法指令异常，而不是在这种指令接近保护边界时报告一个访问故障异常，让可变指令长度的取指和解码变得复杂。

RISC-V 基础 ISA 既有小字节序的内存系统，也有大字节序的内存系统，后者需要特权架构进一步定义大字节序的操作。不论内存系统的字节序如何，指令都作为 16 位小字节序的封装包的序列被存储在内存中。构成一个指令的封装包被存储在地址递增的半字地址处，封装包的最低地址持有指令规范中指令的最低若干位。

我们最初为 *RISC-V* 内存系统选择小字节序的字节次序，因为小字节序系统当前在商业上占主导（所有的 *x86* 系统；*iOS*、安卓，和用于 *ARM* 的 *Windows*）。一个小问题是，我们已经发现，小字节序内存系统对于硬件设计者更加自然。但是，考虑到特定的应用领域（例如 *IP* 网络）在大字节序数据结构上的操作，以及基于大字节序处理器构建的特定遗留代码，所以我们也已经定义了 *RISC-V* 的大字节序和双字节序变体。

我们不得不固定指令封装包在内存中存储的顺序，而且不依赖于内存系统的字节序，来确保长度编码位始终以半字地址顺序首先出现。这允许取指单元通过只检查第一个 *16* 位指令包的最初几位，就快速决定可变长度指令的长度。

我们更进一步地把指令封装包本身做成小字节序的，以便从内存系统字节序中把指令编码完全解耦出来。这个设计对软件工具和双字节序硬件都有好处。否则，例如一个 *RISC-V* 汇编器或反汇编器将总是需要预先知道当前运行系统的字节序，尽管在双字节序系统中，字节序的模式可能在执行期间动态变化。与之相反，通过给定指令一个固定的字节序，有时可以让编写软件无需感知字节序，甚至是以二进制形式的软件，就像位置无关代码（*PIC*）一样。

然而，对于编码或解码机器指令的 *RISC-V* 软件来说，选择只有小字节序的指令的确会有后果。例如，大字节序的 *JIT* 编译器在向指令内存处执行存储操作的时候，必须交换字节的次序。

一旦我们已经决定了固定为小字节序指令编码，这将自然地导致把长度编码位放置在指令格式的 *LSB*[1] 的位置，以避免打断操作码域。

## 1.6 异常、陷入和中断

我们使用术语“异常（*Exception*）”来指代一种发生在运行时的不正常的状况，它与当前 RISC-V 硬件线程中的一条指令相关联。我们使用术语“中断（*Interrupt*）”来指代一种外部的异步事件，它可能导致一个 RISC-V 硬件线程经历一次意料之外的控制转移。我们使用术语“陷入（*Trap*）”来指代由一个异常或中断引发的将控制权转移到陷入处理程序的过程。

下面的章节中的指令描述了在指令执行期间可以引发异常的条件。大多数 RISC-V EEI 的通常行为是，当在一个指令上发出异常的信号时，会发生一次到某些处理程序的陷入（标准浮点扩展中的浮点异常除外，那些并不引起陷入）。硬件线程产生中断、中断路由、和中断启用的具体方式依赖于 EEI。

我们使用的“异常”和“陷入”概念与 *IEEE-754* 浮点标准中的相兼容。

陷入是如何处理的，以及对运行在硬件线程上的软件的可见性如何，依赖于外围的执行环境。从运行在执行环境内部的软件的视角，在运行时遭遇硬件线程的陷入将有四种不同的影响：

[1] 译者注：Least Significant Bit，最低有效位

**被控制的陷入：** 这种陷入对于运行在执行环境中的软件可见，并由软件处理。例如，在一个于硬件线程上同时提供监管器模式和用户模式的 EEI 中，用户模式硬件线程的 ECALL 通常将导致控制被转移到运行在相同硬件线程上的一个监管器模式的处理程序。类似地，在相同的环境中，当一个硬件线程被中断，硬件线程上将运行一个监管器模式中的中断处理程序。

**被请求的陷入：** 这种陷入是一个同步的异常，它是对执行环境的一种显式调用，请求了一个代表执行环境内部的软件的动作。一个例子便是系统调用。在这种情况下，执行环境采取了被请求的动作后，硬件线程上的执行可能继续，也可能不会继续。例如，一个系统调用可以移除硬件线程，或者引起整个执行环境的有序终止。

**不可见的陷入：** 这种陷入被执行环境透明地处理了，并且在陷入被处理之后，执行正常继续。例子包括模拟缺失的指令、在按需分页的虚拟内存系统中处理非常驻页故障，或者在多程序机器中为不同的事务处理设备中断。在这些情况中，运行在执行环境中的软件不会意识到陷入（我们忽略了这些定义中的时间影响）。

**致命的陷入：** 这种陷入代表了一个致命的失败，并引发执行环境终止执行。例子包括虚拟内存页保护检查的失败，或者看门狗[1]计时器到期。每个 EEI 应当定义执行应如何被终止，以及如何将其汇报给外部环境。

表 1.1 显示了每种陷入的特点：

| | 被控制的 | 被请求的 | 不可见的 | 致命的 |
|---|---|---|---|---|
| 执行终止 | 否 | 否 1 | 否 | 是 |
| 软件被遗忘 | 否 | 否 | 是 | 是 2 |
| 由环境处理 | 否 | 是 | 是 | 是 |

表 1.1: 陷入的特点。注：1）可以被请求终止. 2）不精确的致命的陷入或许可被软件观测到。

EEI 为每个陷入定义了它是否会被精确处理，尽管通常建议是尽可能地保持精确处理。被控制的陷入和被请求的陷入可以被执行环境内部的软件观测到是不精确的。不可见的陷入，根据定义，不能被运行在执行环境内部的软件观测到是否精确。致命陷入可以被运行在执行环境内部的软件观测到不精确，如果已知错误的指令没有引起直接的终止的话。

因为这篇文档描述了非特权指令，所以陷入是很少被提及的。处理包含陷入的架构性方法被定义在特权架构手册中，伴有支持更丰富 EEI 的其它特征。这里只记录了被单独定义的引发请求陷入的非特权指令。根据不可见的陷入的性质，其超出了这篇文档的讨论范围。没有在本文档中定义的指令编码，和没有被一些其它方式定义的指令编码，可以引起致命陷入。

[1]译者注：watchdog，一种用于访问保护的装置

## 1.7 “未指定的”(UNSPECIFIED) 行为和值

架构完全描述了架构必须做的事和任何关于它们可能做的事的约束。对于那些架构有意不约束实现的情况，会显式地使用术语“未指定的”。

术语“未指定的”指代了一种有意不进行约束的行为或值。这些行为或值对于扩展、平台标准或实现是开放的。对于基础架构定义为“未指定的”的情形，扩展、平台标准或实现文档可以提供规范性内容以进一步约束。

像基础架构一样，扩展架构应当完全描述清楚所允许的行为和值，并使用术语“未指定的”用于有意不做约束的情况。对于这种情况，就可以被其它的扩展、平台标准或实现来约束或定义。

# 第二章　RV32I 基础整数指令集，2.1 版本

本章介绍 RV32I 基础整数指令集。

---

*RV32I* 的设计要足以能够形成一个编译器的目标码，并足以能够支持现代操作系统环境。该 *ISA* 也被设计为在最小化的实现中降低对硬件的需求。*RV32I* 包含 *40* 条各不相同的指令——尽管在某个简单的实现中，可能会用一个总是陷入的 *SYSTEM* 硬件指令来覆盖 *ECALL/EBREAK* 指令，以及可能会把 *FENCE* 指令实现为一个 *NOP*，从而把基础指令数目减少到总计 *38* 条。*RV32I* 可以模拟几乎任何其它的 *ISA* 扩展（除了 *A* 扩展，因为它需要原子性的额外硬件支持）

实际上，一个包含了机器模式（*machine-mode*）特权架构的硬件实现还将需要 *6* 个 *CSR* 指令。

对于教学目的来说，基础整数 *ISA* 的子集可能是非常有用的，但是“基础”已经定义了，应当尽量不要在一个真实的硬件实现中对基础整数 *ISA* 进行子集化——除非想省略掉对非对齐内存访问的支持，或者想把所有的 *SYSTEM* 指令视为一个单独的陷入。

---

标准 *RISC-V* 汇编语言语法的文档在《汇编程序员手册》 *[1]* 中。

---

大多数对 *RV32I* 的注解也适用于 *RV64I* 基础指令集。

## 2.1　基础整数 ISA 的编程模型

表 2.1显示了基础整数 ISA 的非特权状态。对于 RV32I，32 个 `x` 寄存器每个都是 32 位宽，也就是说，XLEN = 32。寄存器 `x0` 的所有位都被硬布线为 0。通用目的寄存器 `x1-x31` 持有数值，这些值被各种指令解释为各种布尔值的集合、或者二进制有符号整数，或者无符号整数的二补码。

还有一个额外的非特权寄存器：程序计数器 `pc`，保存当前指令的地址。

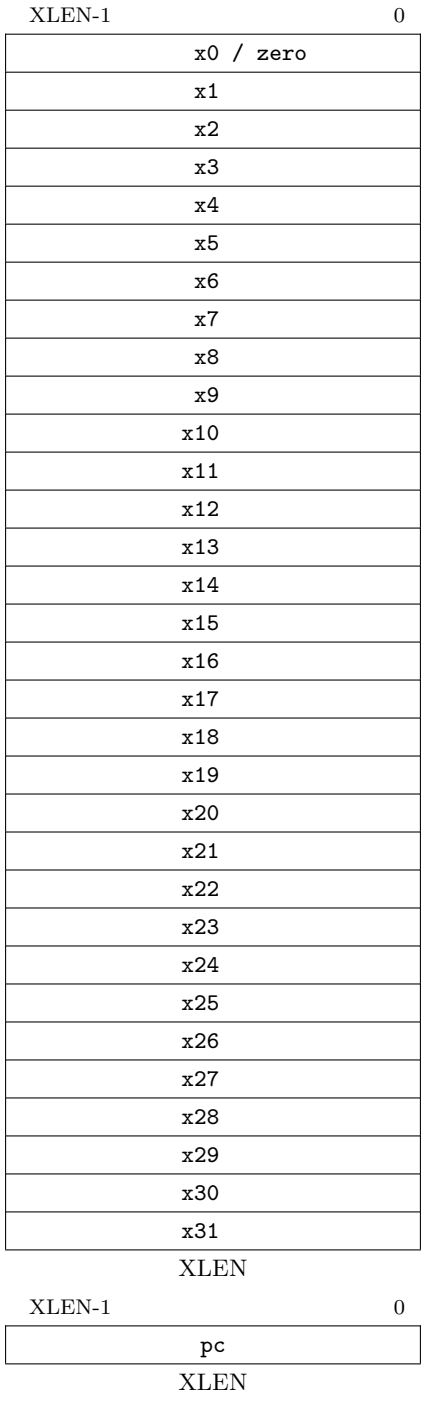



图 2.1: RISC-V 基础非特权整数寄存器状态

---

在基础整数 *ISA* 中没有专门的栈指针或子程序返回地址链接的寄存器；指令编码允许任何的 `x` 寄存器被用于这些目的。然而，标准软件调用约定使用寄存器 `x1` 来持有调用的返回地址，同时寄存器 `x5` 可用作备选的链接寄存器。标准调用约定使用寄存器 `x2` 作为栈指针。

硬件可能选择加速函数调用并使用 x1 或 x5 返回。见 *JAL* 和 *JALR* 指令的描述。

压缩 *16* 位指令格式是围绕着 x1 是返回地址寄存器，而 x2 是栈指针的假设设计的。使用其它约定（非标准约定）的软件虽然可以正确地执行，但是可能会让导致更大的代码尺寸。

---

架构寄存器的可用数目对代码尺寸、性能、和能量消耗有很大的影响。尽管 *16* 个寄存器对于一个整数 *ISA* 来说运行已编译代码理应是足够的，但是使用 *3*-地址格式，在 *16* 位指令中编码一个带有 *16* 个寄存器的完整 *ISA* 仍然是不可能的。尽管 *2*-地址格式是可能的，但是将增加指令数量并降低效率。我们希望避免中间指令尺寸（例如 *Xtensa* 的 *24* 位指令），以简化基础硬件实现。一旦采用了 *32* 位指令尺寸，就可以直接支持 *32* 个整数寄存器。更大数目的整数寄存器也对高性能代码的性能提升有帮助，可以促成循环展开、软件流水线和缓存平铺的广泛使用。

由于这些原因，我们为 *RV32I* 选择了 *32* 个整数寄存器作为约定数量。动态寄存器使用往往由一些频繁访问的寄存器所控制，而 *regfile* 的实现可以优化以减少对频繁访问寄存器的访问能耗 *[22]*。可选的压缩 *16* 位指令格式大多数只访问 *8* 个寄存器，并因此可以提供一种稠密的指令编码；而额外的指令集扩展，如果愿意，可能支持更大的寄存器空间（或者是扁平的，或者是分层的）。

对于资源受限的嵌入式应用，我们已经定义了 *RV32E* 子集，它只有 *16* 个寄存器（第 六章）。

## 2.2 基础指令格式

在基础 RV32I ISA 中，有四个核心指令格式（R/I/S/U），如图 2.2所示。所有这四个格式都是 32 位固定长度。基础 ISA 有 IALIGN = 32 意味着指令必须在内存中对齐到四字节的边界。如果在执行分支或无条件跳转时，目标地址没有按 IALIGN 位对齐，将生成一个指令地址未对齐的异常。这个异常由分支或跳转指令汇报，而不是目标指令。对于还没有被执行的条件分支，不会生成指令地址未对齐异常。

---

当加入了 *16* 位长度的指令扩展或者其它长度为 *16* 位奇数倍的扩展（即，$IALIGN = 16$）时，对基础 *ISA* 指令的对齐约束被放宽到按双字节边界对齐。

导致指令未对齐的分支或跳转，将汇报指令地址未对齐异常，以帮助调试，同时有助于简化 $IALIGN = 32$ 的系统硬件设计，因为这是唯一可能发生未对齐的情况。

解码一个保留指令的行为是“未指定的”（UNSPECIFIED）。

---

一些平台下，解码为标准使用而保留的操作码会引发一个非法指令异常。其它平台可能允许保留的操作码空间被用于非合规的扩展。

为了简化解码，所有格式中，RISC-V ISA 在相同的位置保存源寄存器（*rs1* 和 *rs2*）和目的寄存器（*rd*）。除了 CSR 指令（第 十一章）中使用的 5 位立即数，立即数总是符号扩展的，并且通

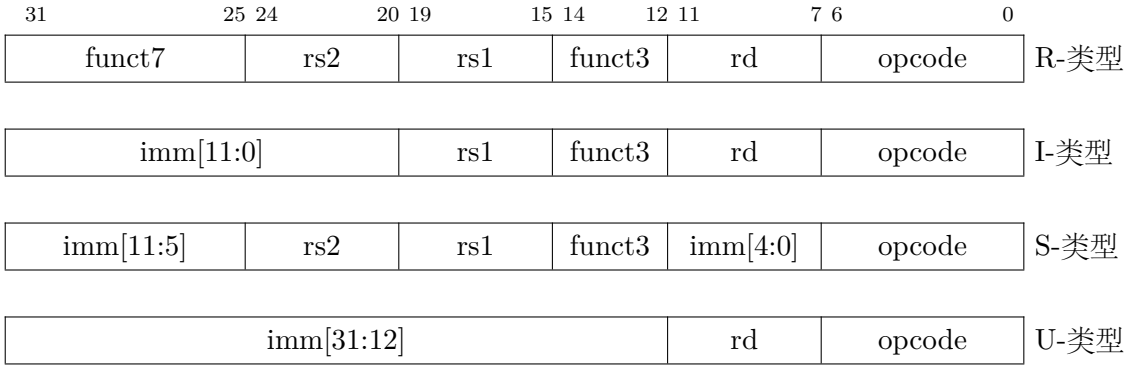



图 2.2: RISC-V 基础指令格式。每个立即数子域都用正被产生的立即数值中的位位置（imm[$x$]）的标签标记，而不是像通常做的那样，用指令立即数域中的位位置。

常在指令中被封装在最左端的可用位，且被提前分配以减少硬件复杂度。特别地，为了加速符号扩展的电路，所有立即数的符号位总是在指令的位 31 处。

---

在实现层面中，解码寄存器标识符通常都是非常关键的路径，因此选择指令格式时，在所有格式中，寄存器标识符都保存在相同的位置上；作为代价，不得不跨格式移动立即数位（一个分享自 *RISC* 第四版 *RISC-IV* 的属性，又称 *SPUR [12]*）。

实际上，大多数立即数或者比较小，或者需要所有的 *XLEN* 位。我们选择了一种不对称的立即数分割方法（常规指令中的 *12* 位加上一个特殊的 *20* 位的“加载上位立即数（*load-upper-immediate*）”指令）来为常规指令增加可用的编码空间。

立即数是符号扩展的，因为对于某些立即数，我们没有观察到使用零扩展的收益（像在 *MIPS ISA* 中），并且想保持 *ISA* 尽可能地简单。

## 2.3 立即数编码变量

基于对立即数的处理，还有两个指令格式的变体（B/J），如图 2.3所示。

S 格式和 B 格式之间唯一的不同是，在 B 格式中，12 位立即数域被用于以 2 的倍数对分支的偏移量进行编码。将中间位（imm[10:1]）和符号位放置在固定的位置，同时 S 格式中的最低位（inst[7]）以 B 格式对高序位进行编码，而不是像传统的做法那样，在硬件中把编码指令立即数中的所有位直接左移一位。

类似地，U 格式和 J 格式之间唯一的不同是，20 位立即数向左移位 12 位形成 U 格式立即数，而向左移 1 位形成 J 格式立即数。选择 U 格式和 J 格式立即数中的指令位的位置，是为了与其它格式和彼此之间有最大程度的交叠。

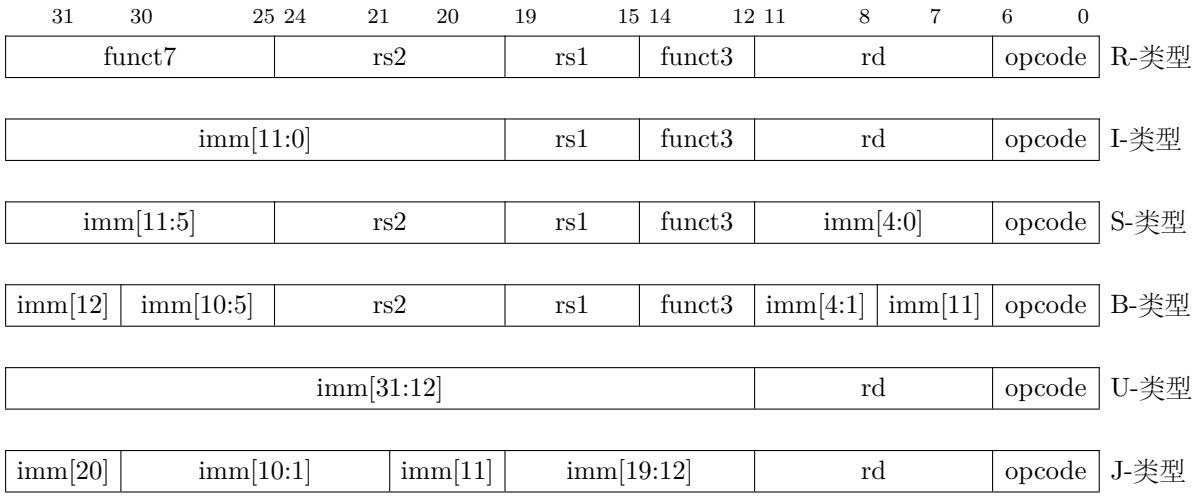



图 2.3: 显式立即数的 RISC-V 基础指令格式。

图 2.4显示了由每个基础指令格式产生的立即数，并用标记显示了立即数值的各个位是由哪个指令位（inst[$y$]）所产生的。

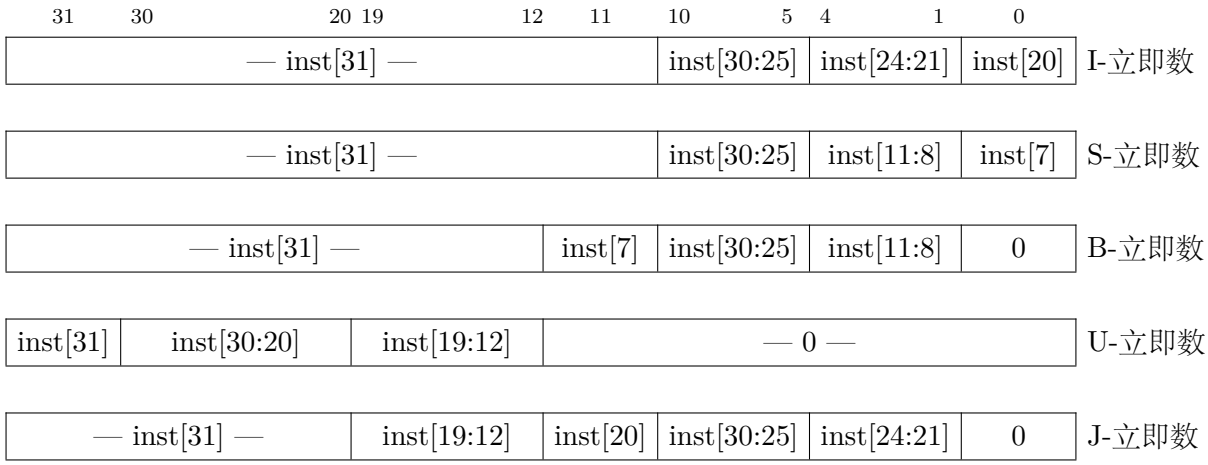



图 2.4: 由 RISC-V 指令产生的立即数的类型。用构造了它们值的指令位对域进行了标记。符号扩展总是使用 inst[31]。

---

符号扩展是最关键的立即数操作之一（特别是对 *XLEN>32*），而在 *RISC-V* 中，所有立即数的符号位总是保持在指令的位 *31*，以允许符号扩展与指令解码并行处理。

虽然更加复杂的实现可能带有用于分支和跳转计算的独立加法器，并且，因为在不同指令类型之间，保持立即数位的位置不变并不能从中获得好处，所以我们希望减少最简单实现的硬件开销。通过旋转由 *B* 格式和 *J* 格式立即数编码的指令中的位，而不是使用动态的硬件多路复

用器（*mux*），来将立即数扩大 *2* 倍，我们减少了大约一半的指令符号扇出（*fanout*）和立即数多路复用的开销。加扰（*scrambled*）立即数编码将对静态编译或事前编译添加微不足道的时间。为了指令的动态生成，虽然有一些小小的额外的开销，但是最常见的短转向分支却有了直接的立即数编码。

## 2.4 整数运算指令

大多数整数运算指令操作所有 XLEN 位的值，这些值保存在整数寄存器文件中。整数运算指令或者被编码为使用 I 类型格式的寄存器-立即数操作，或者被编码为使用 R 类型格式的寄存器-寄存器操作。对于寄存器-立即数指令和寄存器-寄存器指令，目的寄存器都是寄存器 *rd*。整数运算指令不会引发算术异常。

---

我们没有在整数指令集中包括对于在整数算术操作时进行溢出检查的特殊指令集的支持，因为许多溢出检查可以使用 *RISC-V* 分支低成本地实现。对于无符号加法的溢出检查，只需要在加法之后执行一条额外的分支指令： `add t0, t1, t2; bltu t0, t1, overflow`.

对于有符号加法，如果一个操作数的符号是已知的，溢出检查只需要在加法之后执行一条分支： `addi t0, t1, +imm; blt t0, t1, overflow`. 这覆盖了带有一个立即操作数的加法的通常情况。

对于一般的有符号加法，在加法之后需要三条额外的指令，利用了这样一个事实：当且仅当某个操作数是负数时，和应当小于另一个操作数。

```
add t0, t1, t2
slti t3, t2, 0
slt t4, t0, t1
bne t3, t4, overflow
```

在 *RV64I* 中，*32* 位有符号加法的检查可以被进一步优化，通过比较在操作数上进行 *ADD* 和 *ADDW* 的结果实现。

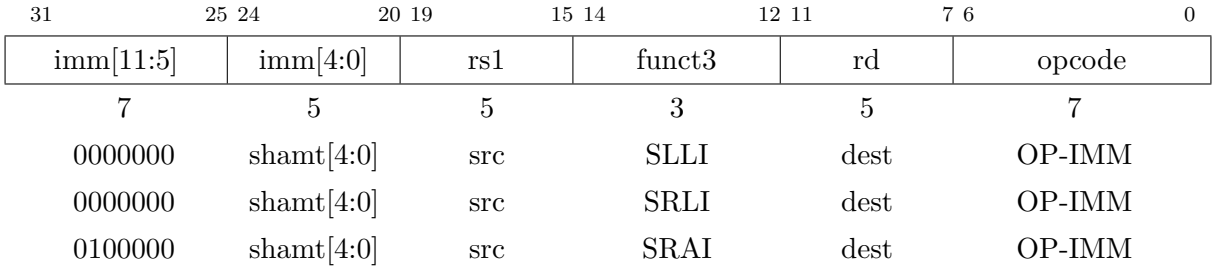
**整数寄存器 - 立即数指令**

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

ADDI 将符号扩展的 12 位立即数加到寄存器 *rs1* 上。简单地将结果的低 XLEN 位当作结果，而忽略了算数溢出。ADDI *rd, rs1, 0* 被用于实现 MV *rd, rs1* 汇编器伪指令。

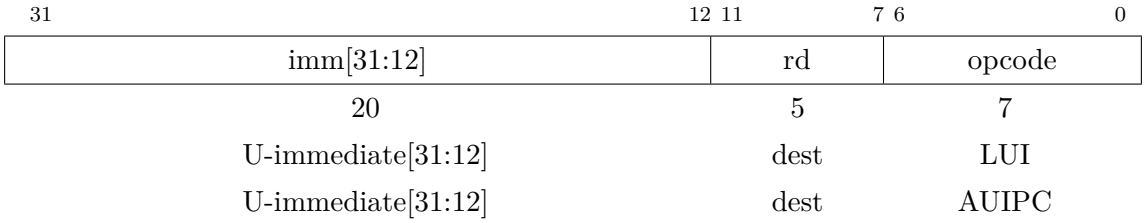
如果寄存器 *rs1* 小于符号扩展的立即数（当二者都被视为有符号数时），SLTI（小于立即数时置 1）指令把值 1 放到寄存器 *rd* 中；否则，该指令把 0 写入 *rd* 中。SLTIU 与之相似，但是将两个值作为无符号数比较（也就是说，前者会把立即数按符号扩展到 XLEN 位，而后者会将其视为无符号数）。注意，如果 *rs1* 等于 0，那么 SLTIU *rd, rs1, 1* 会把 *rd* 设置为 1，否则会把 *rd* 设置为 0（汇编器伪指令 SEQZ *rd, rs*）。

ANDI、ORI、XORI 是在寄存器 *rs1* 和符号扩展的 12 位立即数上执行按位 AND、OR 和 XOR，并把结果放入 *rd* 的逻辑操作。注意，XORI *rd, rs1, -1* 对寄存器 *rs1* 执行按位逻辑反转（汇编器伪指令 NOT *rd, rs*）。



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

按常量移位按照 I 类型格式专门编码。被移位的操作数存放在 *rs1* 中，移位的数目被编码在 I 立即数域的低 5 位。右移类型被编码在位 30。SLLI（Shift Logical Right Immediate）是逻辑左移（零被移位到低位）；SRLI（Shift Logical Left Immediate）是逻辑右移（零被移位到高位）；而 SRAI（Shift Right Arithmetic Immediate）是算数右移（原来的符号位被复制到空出来的高位）。



| 31 12 | 11 7 | 6 0 |
|---|---|---|
| imm[31:12] | rd | opcode |
| 20 | 5 | 7 |
| U-immediate[31:12] | dest | LUI |
| U-immediate[31:12] | dest | AUIPC |

LUI（加载高位立即数）被用于构建 32 位常量，它使用 U 类型格式。LUI 把 32 位 U 立即数值放在目的寄存器 *rd* 中，同时把最低的 12 位用零填充。

AUIPC（加高位立即数到 `pc`）被用于构建 `pc` 相对地址，它使用 U 类型格式。AUIPC 根据 U 立即数形成 32 位偏移量（最低 12 位填零），把这个偏移量加到 AUIPC 指令的地址，然后把结果放在寄存器 *rd* 中。

---

`lui` 和 `auipc` 的汇编语法不代表 *U* 立即数的低 *12* 位，他们总是零。

*AUIPC* 指令支持双指令序列，以便从 `pc` 访问任意的偏移量，用于控制流传输和数据访问。*AUIPC* 与一个 *JALR* 中的 *12* 位立即数的组合可以把控制传输到任何 *32* 位 `pc` 相对地址，而

*AUIPC* 加上常规加载或存储指令中的 *12* 位立即数偏移量可以访问任何 *32* 位 pc 相对数据地址。

通过把 *U* 立即数设置为 *0*，可以获得当前 pc。尽管 *JAL+4* 指令也可以被用于获得本地 pc（*JAL* 后续的指令），但是它可能引起简单微架构中的流水线破坏，或者复杂微架构中的分支目标缓冲区结构污染。

### 整数寄存器 - 寄存器操作

RV32I 定义了一些 R 类型算数操作。所有操作都读取 *rs1* 寄存器和 *rs2* 寄存器作为源操作数，并将结果写入寄存器 *rd*。*funct7* 域和 *funct3* 域制定了操作的类型。

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

ADD 执行 *rs1* 和 *rs2* 的相加。SUB 执行从 *rs1* 中减去 *rs2*。忽略结果的溢出，并把结果的低 XLEN 位写入目的寄存器 *rd*。SLT 和 SLTU 分别执行有符号和无符号的比较，如果 *rs1* < *rs2*，向 *rd* 写入 1，否则写入 0。注意，如果 *rs2* 不等于零，SLTU *rd*, *x0*, *rs2* 把 *rd* 设置为 1，否则把 rd 设置为 0（汇编器伪指令 SNEZ *rd, rs*）。AND、OR 和 XOR 执行按位逻辑操作。

SLL、SLR 和 SRA 对寄存器 *rs1* 中的值执行逻辑左移、逻辑右移、和算数右移，移位的数目保持在寄存器 *rs2* 的低 5 位中。

### NOP 指令

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

除了使 pc 前进，以及使任何适用的性能计数器递增以外，NOP 指令不改变任何架构上的可见状态。NOP 被编码为 ADDI *x0, x0, 0*。

*NOP* 可以被用于把代码段对齐到微架构上的高位有效地址边界，或者为内联代码的修改留出空间。尽管有许多可能的方法来编码 *NOP*，我们定义了一个规范的 *NOP* 编码，来允许微架构优化，以及更具可读性的反汇编输出。其它的 *NOP* 编码形式可用于 *HINT* 指令（第 *2.9* 节 *2.9*）。

选用 *ADDI* 进行 *NOP* 编码是因为，这是在跨多个系统中最可能的、最少资源来执行的方法（如果解码中没有优化的话）。特别是，指令只会读一个寄存器。并且，*ADDI* 功能单元也更可能用于超标量设计，因为加法是最常见的操作。特别是，地址生成（*address-generation*）功能单元可以使用计算基址 + 偏移量地址计算所需的相同硬件来执行 *ADDI*，而寄存器-寄存器 *ADD* 或者逻辑/移位操作都需要额外的硬件。

## 2.5 控制转移指令

RV32I 提供两种类型的控制转移指令：无条件跳转和条件分支。RV32I 中的控制转移指令没有架构上可见的延迟槽。

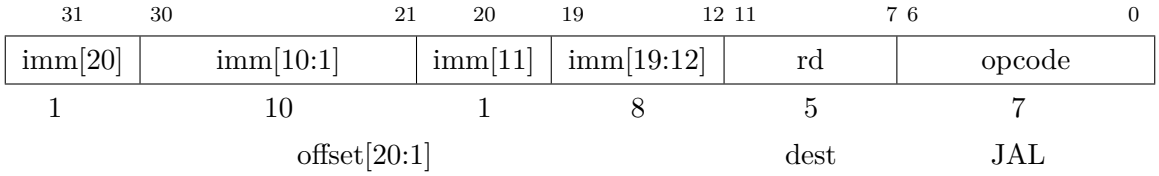
如果在一次跳转或发生转移的目标上发生了一个指令访问故障异常或指令缺页故障异常，该异常会报告在目标指令上，而不是报告在跳转或分支指令上。

### 无条件跳转

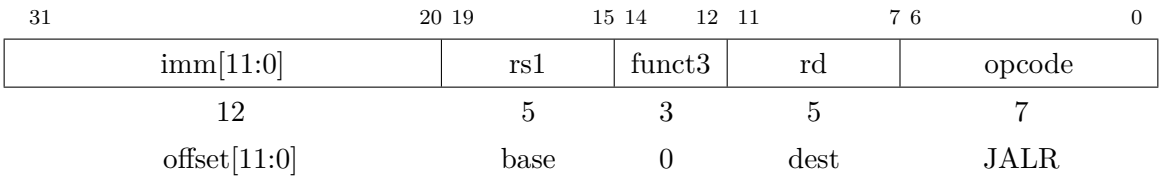
跳转和链接（JAL）指令使用 J 类型格式。J 类型指令把 J 立即数以 2 字节的倍数编码一个有符号的偏移量。偏移量是符号扩展的，加到当前跳转指令的地址上以形成跳转目标地址。跳转可以因此到达的目标范围是 ±1 MiB。JAL 把跟在 JAL 之后的指令的地址（pc+4）存储到寄存器 *rd* 中。标准软件调用约定使用 x1 作为返回地址寄存器，使用 x5 作为备选的链接寄存器。

备选的链接寄存器支持调用 *millicode* 例程（例如，那些在压缩代码中的保存和恢复寄存器的例程），同时保留常规的返回地址寄存器。寄存器 x5 被选为备用链接寄存器，映射到标准调用约定中的一个临时值，其编码与常规链接寄存器相比只有一位不同。

一般的无条件跳转（汇编器伪指令 J）被编码为 *rd*=x0 的 JAL。



间接跳转指令 JALR（跳转和链接寄存器）使用 I 类型编码。通过把符号扩展的 12 位 I 立即数加到寄存器 *rs1* 来获得目标地址，然后把结果的最低有效位设置为零。紧接着跳转的指令的地址（pc+4）被写入寄存器 *rd*。如果不需要结果，寄存器 x0 也可以被用作目的寄存器。




无条件跳转指令都使用 pc 相对地址，以此支持位置无关代码。*JALR* 指令被定义为能够使用双指令序列跳转到 *32* 位绝对地址空间范围内的任何地方。一条 *LUI* 指令可以首先把目标地址的高 *20* 位加载到 rs1，然后 *JALR* 指令可以加上低位。类似地，先用 *AUIPC* 再用 *JALR* 可以跳转到 *32* 位 pc 相对地址范围中的任何地方。

注意 *JALR* 指令不会像条件分支指令那样，把 *12* 位立即数当作 *2* 字节的倍数对待。这回避了硬件中出现的另一种立即数格式。实际上，大多数 *JALR* 的使用，要么有一个零立即数，要么是与 *LUI* 或 *AUIPC* 搭配成对，所以有一点范围减少是无关紧要的。

在计算 *JALR* 目标地址时清理最低有效的位，既稍微简化了硬件，又允许函数指针的低位被用于存储辅助信息。尽管这种情况中，会有一些潜在的错误检查的轻微丢失，但是实际上，跳转到一个不正确的指令地址通常将很快引发一个异常。

当以 rs1=x0 作为基地址使用时，*JALR* 可以被用于实现地址空间中从任何地方到最低*2 KiB*或最高*2 KiB*地址区域的单一指令子例程调用，这可以被用于实现对小型运行时库的快速调用。或者，*ABI* 可以专用于通用目的寄存器，以指向地址空间中任何其它地方的一个库。

如果目标地址没有对齐到 IALIGN 位边界，JAL 和 JALR 指令将产生一个指令地址未对齐（instruction-address-misaligned）异常。

指令地址未对齐异常不会发生在 *IALIGN* = *16* 的机器上，例如那些支持压缩指令集扩展（*C*）的机器。

返回地址预测栈是高性能取指单元的一个常见特征，但是需要精确地探测用于过程调用和有效返回、即将生效的指令。对于 RISC-V，有关指令用途的提示，是通过使用的寄存器号码被隐式地编码的。只有当 *rd* = x1/x5 时，JAL 指令才应当把返回地址推入到返回地址栈（RAS）上。JALR 指令应当压入/弹出一个 RAS 的所有情形如表 2.1所示。

一些其它的 *ISA* 把显式的提示位添加到了它们的间接跳转指令上，来指导返回地址栈的操作。我们使用绑定寄存器号码的隐式提示和调用约定，以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）被给定为 rs1 和 rd 时，接下来 *RAS* 会被同时弹出和推入，以支持协程。如果 rs1 和 rd 是相同的链接寄存器（x1 或者 x5），*RAS* 只是为使能序列中宏操作融合（*macro-op fusion*）而被推入：

`lui ra, imm20; jalr ra, imm12(ra)` 和 `auipc ra, imm20; jalr ra, imm12(ra)`

| *rd* is x1/x5 | *rs1* is x1/x5 | *rd*=*rs1* | RAS action |
|---|---|---|---|
| 否 | 否 | – | 无 |
| 否 | 是 | – | 弹出 |
| 是 | 否 | – | 压入 |
| 是 | 是 | 否 | 弹出，然后压入 |
| 是 | 是 | 是 | 压入 |

表 2.1: 在 JALR 指令的寄存器操作数中编码的返回地址栈预测提示。

**条件分支**

所有的分支指令使用 B 类型指令格式。12 位 B 立即数以 2 字节的倍数编码符号偏移量。偏移量是符号扩展的，加到分支指令的地址上以给出目标地址。条件分支的范围是 ±4 KiB。

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

分支指令对两个寄存器进行比较。BEQ 和 BNE 分别在寄存器 *rs1* 和 *rs2* 相等或不等时采取分支。BLT 和 BLTU 分别使用有符号和无符号的比较，如果 *rs1* 小于 *rs2* 则采取分支。BGE 和 BGEU 分别使用有符号和无符号的比较，如果 rs1 大于或等于 rs2 则采取分支。注意，BGT、BGTU、BLE 和 BLEU 可以分别通过反转 BLT、BLTU、BGE 和 BGEU 的操作数来合成。

---

可以用一条 *BLTU* 指令检查有符号的数组边界，因为任意负数索引都将比任意非负数边界要大。

软件应当按这样的原则来优化编写：按顺序执行的代码路径是占大部分的常见路径，而顺序外的代码路径被采取的频率较低。软件也应当假定，向后的分支将被预测采取，而向前的分支被预测不采取，至少在它们第一次被遇到时如此。动态预测应当快速地学习任何可预测的分支行为。

不像其它的一些架构，对于无条件分支，应当总是使用跳转指令（*rd*=x0 的 JAL），而不是使用一个条件总是真的条件分支指令。RISC-V 的跳转也是 pc 相对的，并支持比分支更宽的偏移量范围，而且将不会污染条件分支预测表。

条件分支被设计为包含两个寄存器之间的算数比较操作（*PA-RISC*、*Xtensa* 和 *MIPS R6* 中也是这样做的），而不是使用条件代码（*x86*、*ARM*、*SPARC*、*PowerPC*），或者只用一个寄存器和零比较（*Alpha*、*MIPS*），又或是只比较两个寄存器是否相等（*MIPS*）。这个设计的动机是观察到：比较与分支的组合指令适合于常规流水线，避免了额外的条件代码状态或者临时寄存器的使用，并减少了静态代码的尺寸和动态指令获取的流量。另一个考虑点是，与零比较需要非平凡的电路延迟（特别是在高级处理中运行流程到达静态逻辑后），并因此与算数量级的比较几乎同样代价高昂。融合的比较与分支指令的另一个优势是，分支可以在前端指令流中被更早地观察到，并因此能够被更早地预测。在基于相同的条件代码可以采取多个分支的情况中，使用条件代码的设计或许有优势，但是我们相信这种情况是相对稀少的。

我们考虑过，但是没有在指令编码中包含静态分支提示。这些虽然可以减少动态预测器的压力，但是需要更多指令编码空间和软件画像来达到最佳结果，并且如果产品的运行没有匹配画像（*profiling*）运行的话，会导致性能变差。

我们考虑过，但是没有包含条件移动或谓词指令，它们可以有效地替换不可预测的短向前分支（*short forward branches*）。条件移动是二者中较简单的，但是难以和条件代码一起使用，因为那会引起异常（内存访问和浮点操作）。谓词会给系统添加额外的标志，添加额外的指令来设置和清除标志，以及在每个指令上增加额外的编码负担。条件移动和谓词指令都会增加乱序微架构的复杂度，因为如果谓词为假，则需要把目的架构寄存器的原始值复制到重命名后的目的物理寄存器，因此会添加隐含的第三个源操作数。此外，静态编译时间决定使用谓词而不是分支，可以导致没有包含在编译器训练集中的输入的性能降低，尤其是考虑到不可预测的分支是稀少的，而且随着分支预测技术的改进会而变得更加稀少。

我们注意到，现存的各种微架构技术会把不可预测的短向前分支转化为内部谓词代码，以避免分支误预测时冲刷流水线的开销 *[7, 11, 10]*，并且已经在商业处理器中被实现 *[19]*。最简单的技术只是通过只冲刷分支阴影（*branch shadown*）中的指令，而不是整个取指流水线，或者通过使用宽指令取指或空闲指令取指槽从两端获取指令，从而减少了从误预测短向前分支恢复的代价。用于乱序核心中的更加复杂的技术是在分支阴影中的指令上添加内部谓词，内部谓词的值由分支指令写入，这允许分支和随后的指令相比于其他代码被推测执行和乱序执行，而与其它代码的执行顺序不一致 *[19]*。

如果目标地址没有对齐到 IALIGN 位边界，并且分支条件评估为真，那么条件分支指令将生成一个指令地址未对齐异常。如果分支条件评估为假，那么指令地址未对齐异常将不会产生。

---

指令地址未对齐异常不会发生在支持 *16* 位对齐指令扩展（例如，压缩指令集扩展 *C*）的机器上。

## 2.6 加载和存储指令

RV32I 是一个“加载-存储”架构，只有加载和存储指令访问内存，而算数指令只操作 CPU 寄存器。RV32I 提供一个 32 位的地址空间，按字节编址。EEI 将定义该地址空间的哪一部分是哪个指

令可以合法访问的（例如，一些地址可能是只读的，或者只支持按字访问）。以 x0 为目的寄存器的加载操作必定会引发某些异常，并引起其它的副作用，即使所加载的值被丢弃。

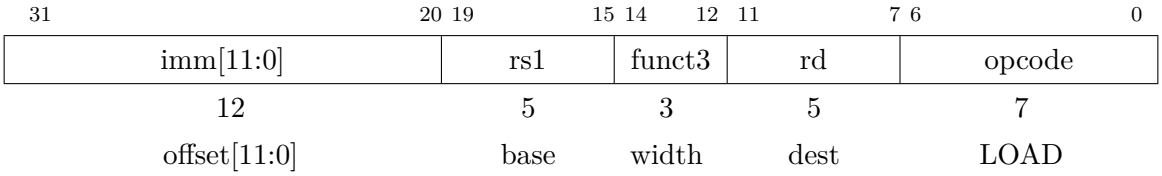
EEI 将定义内存系统是否是小字节序或大字节序的。在 RISC-V 中，字节序是按字节编址的不变量。

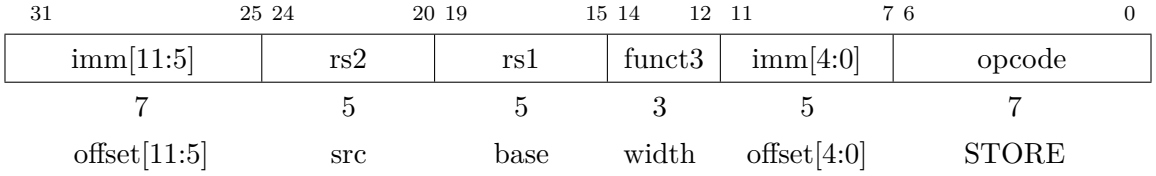
---

在字节序是按字节编址不变量的系统中，有如下的属性：如果一个字节以某个字节序被存储到内存的某个地址，那么从那个地址开始以任何字节序的一个字节大小的加载操作都将返回被存储的值。

在一个小字节序的配置中，对于多字节的存储，在最低内存字节地址处写入寄存器字节的最低有效位，然后按有效位的升序写入其它的寄存器字节。加载类似，把较小有效位的内存字节地址的内容传输到较低有效位的寄存器字节。

在一个大字节序的配置中，对于多字节的存储，在最低内存字节地址处写入寄存器字节的最高有效位，然后按有效位的降序写入其它的寄存器字节。加载类似，把较大有效位的内存字节地址的内容传输到较低有效位的寄存器字节。




加载和存储指令在寄存器和内存之间传输一个值。加载指令被编码为 I 类型格式，存储指令则是 S 类型。通过把寄存器 *rs1* 加到符号扩展的 12 位偏移量，可以获得有效地址。加载指令从内存复制一个值到寄存器 *rd*。存储指令把寄存器 *rs2* 中的值复制到内存。

LW 指令从内存加载一个 32 位的值到 *rd*。LH 先从内存加载一个 16 位的值，然后在存储到 *rd* 中之前，把它符号扩展到 32 位。LHU 先从内存加载一个 16 位的值，然后，在存储到 *rd* 中之前，把它用零扩展到 32 位。LB 和 LBU 被类似地定义于 8 位的值。SW、SH 和 SB 指令从寄存器 *rs2* 的低位将 32 位、16 位和 8 位的值存储到内存。

不管 EEI 如何，有效位地址自然对齐的加载和存储不应当引发地址未对齐的异常。对于有效位地址没有自然对齐到被引用的数据类型的情况（即，有效地址不能被以字节为单位的访问大小整除），其行为取决于 EEI。

EEI 也可以完全支持未对齐的加载和存储，使得运行在执行环境内部的软件将永不会经历包含的或者致命的（译者注：见 1.6 节提到的陷入类型）地址未对齐陷入。在这种情况中，未对齐的加载和存储可以在硬件中被处理，或者通过一个不可见的陷入进入执行环境，或者根据具体地址，可能是硬件和不可见陷入的组合。

EEI 可以不保证未对齐的加载和存储被不可见地处理掉。在这种情况中，没有自然对齐的加载和存储或者可以成功地完成执行，或者可以引发一个异常。所引发的异常可以是一个地址未对齐异常，也可以是一个访问故障异常。对于除了未对齐外都能够完成的内存访问，如果不能模拟未对齐访问（例如，如果对内存区域的访问有副作用），那么可以引发一个访问故障异常而不是一个地址未对齐异常。当 EEI 不保证隐式地处理未对齐的加载和存储时，EEI 必须定义由地址未对齐引起的异常是否导致被包含的陷入（允许运行在执行环境中的软件来处理该陷入），或者导致致命的陷入（终止执行）。

---

当移植遗留代码时，偶尔需要未对齐的访问；在使用某些形式的 *packed-SIMD* 扩展（译者注：打包的 *SIMD* 扩展指令，即 *P* 扩展）、或者处理外部打包的数据结构时，这些未对齐的访问对应用程序的性能会有帮助。对于是否允许 *EEI* 通过常规的加载和存储指令来选择支持未对齐的访问，我们的基本原则是，是否能够简化因额外处理未对齐而引入的硬件电路设计的复杂性。一个选择是，在基础 *ISA* 中将不允许未对齐的访问，然后为未对齐访问提供一些单独的 *ISA* 支持：或者是用一些特殊指令来帮助软件处理未对齐访问，或者是用未对齐访问的新的硬件编址模式。特殊指令是难以使用的，会让 *ISA* 复杂化，并通常增加了新的处理器状态（例如，*SPARC VIS* 对齐地址偏移量寄存器），或是让现有处理器状态的访问复杂化（例如，*MIPS LWL/LWR* 部分寄存器写操作）。此外，对于面向循环的 *packed-SIMD* 代码，当操作数未对齐时，会迫使软件根据操作数的对齐方式提供多种形式的循环，这使代码生成复杂化，并增加了循环启动的负担。新的未对齐硬件编址模式或者会占据相当多的指令编码空间，或者需要非常简化的编址模式（例如，只有寄存器间接寻址模式）。

即使是当未对齐的加载和存储成功完成时，根据实现不同，这些访问也可能运行得极度缓慢（例如，当通过一个不可见的陷入实现时）。此外，自然对齐的加载和存储会被保证原子执行，但未对齐的加载和存储却可能不会，并因此需要额外的同步来保证原子性。

---

我们没有强制要求未对齐访问的原子性，所以执行环境可以使用一种不可见的机器陷入和一个软件处理程序来处理部分或所有的未对齐访问。如果提供了未对齐硬件支持，软件利用它简单地使用常规加载和存储指令就可以。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 内存排序指令


FENCE 指令被用于为其它 RISC-V 硬件线程和外部设备或协处理器所看到的设备 I/O 和内存访问进行排序。设备输入（I）、设备输出（O）、内存读（R）和内存写（W）的任意组合可以与其他同样这些的任意组合进行排序。可以非正式地认为，没有其它的 RISC-V 硬件线程或外部设备可以在 FENCE 之前的*前驱*（*predecessor*）集合中的任何操作之前，观察到 FENCE 之后的*后继*（*successor*）集合中的任何操作。第 17 章 十七提供了 RISC-V 内存一致性模型的一个精确的描述。

如同所观察到的对那些外部设备发起的内存读写进行排序，FENCE 指令也对硬件线程发起的内存读和内存写进行排序。然而，FENCE 不对外部设备使用任何其它信号机制发起的观察事件进行排序。

---

一个设备可能通过某些外部通信机制（例如，一个为中断控制器驱动中断信号的内存映射控制寄存器）观察到对一个内存位置的访问。这个通信是在 *FENCE* 排序机制的视野之外的，因此，*FENCE* 指令不能保证中断信号变化何时能对中断控制器可见。特定的设备可以提供额外的排序保证以减少软件负载，但属于 *RISC-V* 内存模型的范畴之外的话题。

一个设备可能通过某些外部通信机制（例如，一个为中断控制器驱动中断信号的内存映射控制寄存器）观察到对一个内存位置的访问。这个通信是在 *FENCE* 排序机制的视野之外的，因此，*FENCE* 指令不能提供保证，中断信号的变化何时能对中断控制器可见。特定的设备可以提供额外的排序保证以减小软件负载，但是那些机制属于 *RISC-V* 内存模型的范畴之外了。

EEI 将定义什么样的 I/O 操作是允许的，特别是当被加载和存储指令访问时，分别有哪些内存地址将被视为设备输入和设备输出操作、而不是内存读取和写入操作，并以此排序。例如，内存映射 I/O 设备通常被未缓存的加载和存储访问，这些访问使用 I 和 O 位而不是 R 和 W 位进行排序。指令集扩展也可以在 FENCE 中规定同样使用 I 和 O 位排序的新的 I/O 指令。

| *fm* 域 | 助记符 | 含义 |
|---|---|---|
| 0000 | 无 | 一般的屏障 |
| 1000 | TSO | 带有 FENCE RW, RW：排除“写到读”的次序<br>其它的：保留供未来使用。 |
| 其他 | | 保留供未来使用。 |

表 2.2: 屏障模式编码

屏障模式域 *fm* 定义了 FENCE 的语义。一个 *fm*=0000 的 FENCE 把它的前驱集合中的所有内存操作，排在它的后继集合的所有内存操作之前。

FENCE.TSO 指令被编码为 *fm*=1000、前驱 RW、以及后继 = RW 的 FENCE 指令。FENCE.TSO 把它前驱集合中的所有加载操作排在它后继集合中的所有内存操作之前，并把它前驱集合中的所有存储操作排在它后继集合中的所有存储操作之前。这使得 FENCE.TSO 的前驱集合中的非 AMO 存储操作与它的后继集合中的非 AMO 加载操作不再有序。

---

因为*FENCE RW,RW*所施加的排序是 *FENCE.TSO* 所施加排序的一个超集，所以忽略 fm 域并把 *FENCE.TSO* 作为*FENCE RW,RW*实现是正确的。

FENCE 指令中的未使用的域——*rs1* 和 *rd*——被保留用于未来扩展中的更细粒度的屏障。为了向前兼容，基础实现应当忽略这些域，而标准软件应当把这些域置为零。同样地，表 2.2中的许多 *fm* 和前驱/后继集合设置也被保留供将来使用。基础实现应当把所有这些保留的配置视为普通的 *fm* = 0000 的屏障，而标准软件应当只使用非保留的配置。

---

我们选择了一个宽松的内存模型以允许从简单的机器实现和可能的未来协处理器或加速器扩展获得高性能。我们从内存 *R/W* 排序中分离了 *I/O* 排序以避免在一个设备驱动硬件线程中进行不必要的序列化，而且也支持备用的非内存路径来控制额外增加的协处理器或 *I/O* 设备。此外，简单的实现还可以忽略前驱和后继的域，而总是在所有的操作上执行保守的屏障。

## 2.8 环境调用和断点

SYSTEM 指令被用于访问那些可能需要访问特权的系统功能，并且使用 I 类型指令格式进行编码。这些指令可以被划分为两个主要的类别：那些原子性的“读-修改-写”控制和状态寄存器（CSR）相关指令，和所有其它潜在的特权指令。CSR 指令在第 十一章描述，而基础非特权指令在接下来的小节中描述。

---

在简单的实现中，*SYSTEM* 指令被定义为允许更简单的实现总是陷入到一个单独的软件陷入处理程序。更复杂的实现可能在硬件中执行更多的各系统指令。

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

这两个指令对支持的执行环境引发了一个精确的请求陷入。

ECALL 指令被用于向执行环境发起一个服务请求。EEI 将定义服务请求参数传递的方式，但是通常这些参数将处于整数寄存器文件中已定义的位置。

EBREAK 指令被用于将控制返回到调试环境。

---

*ECALL 和 EBREAK 之前被命名为 SCALL 和 SBREAK。这些指令有相同的功能和编码，但是被重命名了，是为了反映它们可以更一般化地使用，而不只是调用一个监管器级别的操作系统或者调试器。*

---

*EBREAK 被主要设计为供调试器使用的，以引发执行停止和返回到调试器中。EBREAK 也被标准 gcc 编译器用来标记可能不会被执行的代码路径。*

*EBREAK 的另一个用处是支持“半宿主”，即，包含调试器的执行环境可以通过围绕 EBREAK 指令构建一套备用系统调用接口来提供服务。因为 RISC-V 基础 ISA 没有提供更多的（多于一个的）EBREAK 指令，RISC-V 半宿主使用一个特殊的指令序列来将半宿主 EBREAK 与调试器插入的 EBREAK 进行区分。*

```
slli x0, x0, 0x1f   # 入口 NOP
ebreak              # 中断到调试器
srai x0, x0, 7      # NOP编码编号为7的半宿主调用
```

*注意这三个指令都必须是 32 位宽的指令，也就是说，它们必须不能出现在第 十八章里描述的压缩 16 位指令之中。*

*移位 NOP 指令仍然被认为可以用作 HINT*

*半宿主是一种服务调用的形式，它将更自然地使用现有 ABI 被编码为 ECALL，但是这将要求调试器有能力拦截 ECALL，那是对调试标准的一个较新的补充。我们试图改为使用带有标准 ABI 的 ECALL，这种情况中，半宿主可以与现有标准分享服务 ABI。*

*我们注意到，ARM 处理器在较新的设计中，对于半宿主调用，也已经转为使用了 SVC 而不再是 BKPT。*

## 2.9 “提示”指令

RV32I 保留了大量的编码空间用于 HINT 指令，这些通常被用于与微架构进行性能提示的交互。像 NOP 指令，除了提升 pc 和任何适用的性能计数器，HINT 不改变任何架构上的可视状态。实现中总是被允许忽略已编码的提示。

大多数 RV32I HINT 被编码为 *rd*=x0 的整数运算指令。其余 RV32I HINT 被编码为没有前驱集和后继集且 $fm = 0$ 的 FENCE 指令。

选择这样的 *HINT* 编码是为了简单的实现可以完全忽略 *HINT*，而把 *HINT* 作为一个常规的、但是恰好不改变架构状态的指令。例如，如果目的寄存器是 x0，那么 *ADD* 就是一个 *HINT*；五位的 rs1 和 rs2 域编码了 *HINT* 的参数。然而，简单的实现中可以简单地把 *HINT* 执行为把 rs1 加 rs2 写入 x0 的 *ADD* 指令，这种没有架构上可见的影响。

作为另一个例子，一个 pred 域为零且 fm 域为零的 *FENCE* 指令是一个 *HINT*；succ 域，rs1 域，*and* rd 域编码了 *HINT* 的参数。一个简单的实现可以把 *HINT* 作为一个 *FENCE* 简单地执行，即，在任何被编码在 succ 域中的后续内存访问之前，对先前内存访问的空集进行排序。由于前驱集和后继集的交集为空，该指令不会施加内存排序，因此它没有架构可见的影响。

表 2.3列出了所有的 RV32I HINT 代码点。91% 的 HINT 空间被保留用于标准 HINT。剩余的 HINT 空间被指定用于自定义的 HINT：在这个子空间中，将永远不会定义标准 HINT。

我们预计标准的提示最终会包含内存系统空间和时间的局部性提示、分支预测提示、线程调度提示、安全性标签、和用于模拟/仿真的仪器标志。

| 指令 | 约束 | 代码点 | 目的 |
|---|---|---|---|
| LUI | *rd*=x0 | $2^{20}$ | 保留供未来标准使用 |
| AUIPC | *rd*=x0 | $2^{20}$ | |
| ADDI | *rd*=x0, 并且要么 *rs1*≠x0 要么 *imm*≠0 | $2^{17}-1$ | |
| ANDI | *rd*=x0 | $2^{17}$ | |
| ORI | *rd*=x0 | $2^{17}$ | |
| XORI | *rd*=x0 | $2^{17}$ | |
| ADD | *rd*=x0, *rs1*≠x0 | $2^{10}-32$ | |
| ADD | *rd*=x0, *rs1*=x0, *rs2*≠x2–x5 | 28 | |
| ADD | *rd*=x0, *rs1*=x0, *rs2*=x2–x5 | 4 | (*rs2*=x2) NTL.P1<br>(*rs2*=x3) NTL.PALL<br>(*rs2*=x4) NTL.S1<br>(*rs2*=x5) NTL.ALL |
| SUB | *rd*=x0 | $2^{10}$ | 保留供未来标准使用 |
| AND | *rd*=x0 | $2^{10}$ | |
| OR | *rd*=x0 | $2^{10}$ | |
| XOR | *rd*=x0 | $2^{10}$ | |
| SLL | *rd*=x0 | $2^{10}$ | |
| SRL | *rd*=x0 | $2^{10}$ | |
| SRA | *rd*=x0 | $2^{10}$ | |
| FENCE | *rd*=x0, *rs1*≠x0, *fm*=0, 且 *pred*=0 或者 *succ*=0 | $2^{10}-63$ | |
| FENCE | *rd*≠x0, *rs1*=x0, *fm*=0, 且 *pred*=0 或者 *succ*=0 | $2^{10}-63$ | |
| FENCE | *rd*=*rs1*=x0, *fm*=0, *pred*=0, *succ*≠0 | 15 | |
| FENCE | *rd*=*rs1*=x0, *fm*=0, *pred*≠W, *succ*=0 | 15 | |
| FENCE | *rd*=*rs1*=x0, *fm*=0, *pred*=W, *succ*=0 | 1 | 暂停 |
| SLTI | *rd*=x0 | $2^{17}$ | 指定供自定义使用 |
| SLTIU | *rd*=x0 | $2^{17}$ | |
| SLLI | *rd*=x0 | $2^{10}$ | |
| SRLI | *rd*=x0 | $2^{10}$ | |
| SRAI | *rd*=x0 | $2^{10}$ | |
| SLT | *rd*=x0 | $2^{10}$ | |
| SLTU | *rd*=x0 | $2^{10}$ | |

表 2.3: RV32I 提示指令。

# 第三章 “Zifencei”指令获取屏障（2.0 版本）

这章定义了“Zifencei”扩展，它包括了 FENCE.I 指令，该指令提供了在相同硬件线程上进行的写指令内存与指令获取之间的显式同步。目前，这个指令是确保对硬件线程可见的存储也将对它的指令获取可见的唯一标准机制。

---

我们考虑过、但是没有包括“存储指令字”指令（像在 *MAJC* 中那样 *[21]*）。*JIT* 编译器可以在单个的 *FENCE.I* 之前生成一大段对指令的追踪，并且通过把翻译过的指令写到已知的没有保留在 *I*-缓存中的内存区域，分摊任何指令缓存的嗅探/失效负载。

---

*FENCE.I* 指令被设计为支持多种实现。简单的实现可以在 *FENCE.I* 被执行的时候冲刷本地指令缓存和指令流水线。更加复杂的实现可以在每个数据（指令）缓存缺失的时候嗅探（*snoop*）指令（数据）缓存，或者在主指令缓存中的某些行正在被本地存储指令写入时，使用一个包容统一的私有 *L2* 缓存使其无效化。如果指令和数据缓存以这种方式保持一致性，或者如果内存系统只由未缓存的 *RAM* 组成，那么只有获取流水线需要在 *FENCE.I* 被冲刷。

*FENCE.I* 指令曾是基础 *I* 指令集的前一部分。受到两个主要问题的驱使，尽管在编写本手册时它仍然是保持指令获取一致性的仅有的标准方法，它还是被移出了强制性基础指令集。

首先，我们已经认识到，在一些系统上，*FENCE.I* 的实现将是昂贵的，在内存模型任务组中正在讨论替代它的机制。特别地，对于拥有非一致性指令缓存和非一致性数据缓存、或者指令缓存的重新填充不会嗅探（*snoop*）一致性数据缓存的设计，在遇到一个 *FENCE.I* 指令时，这两个缓存都必须完全被冲刷。当在一个统一的缓存或较外层内存系统之前有多个级别的 *I* 缓存和 *D* 缓存时，这个问题将更加严重。

第二，该指令并非足够强力能在一个像 *Unix* 那样的操作系统环境中的用户级别可用。*FENCE.I* 只同步本地硬件线程，而 *OS* 可以在 *FENCE.I* 之后把用户硬件线程重新调度到一个不同的物理硬件线程。这将需要 *OS* 执行一个额外的 *FENCE.I* 作为每个上下文迁移的一部分。出于这个原因，标准 *Linux ABI* 已经从用户级别中移除了 *FENCE.I*，现在是需要一个系统调用来保持指令获取的一致性，这允许 *OS* 最小化当前系统上需要执行的 *FENCE.I* 的数目，并为将来改进的指令获取一致性机制提供向前兼容性。

正在讨论的未来的指令获取一致性方法包括，提供更加严格的 *FENCE.I* 版本，它只把 rs1 中指定的地址作为目标，并/或者允许软件使用依赖于机器模式缓存维护操作的 *ABI*。

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

FENCE.I 指令被用于同步指令和数据流。在硬件线程执行 FENCE.I 指令以前，RISC-V 不保证到指令内存的存储将对 RISC-V 硬件线程上的指令获取可见。FENCE.I 指令确保 RISC-V 硬件线程上后续的指令获取将能看到已经对同一 RISC-V 硬件线程可见的任何先前的数据存储。在一个多处理器系统中，FENCE.I 不确保其它 RISC-V 硬件线程的指令获取也将能看到本地硬件线程的存储。为了让对指令内存的存储对于所有的 RISC-V 硬件线程可见，正在写的硬件线程也必须在请求所有的远程 RISC-V 硬件线程执行 FENCE.I 之前执行一次数据 FENCE。

FENCE.I 指令中的未使用的域，imm[11:0]、rs1 和 rd，被保留用于未来扩展中的更细粒度的屏障功能。为了向前兼容，基础实现应当忽略这些域，而标准软件应当把这些域置为零。

---

因为 *FENCE.I* 只使用硬件线程自己的指令获取来给存储排序，如果应用程序线程将不会被迁移到不同的硬件线程，那么应用程序代码应当只依赖 *FENCE.I*。*EEI* 可以提供有效的多处理器指令流同步机制。

# 第四章 “Zihintntl”非时间局部性提示（0.2 版本）

NTL 指令是一种 HINT，它表示直接后继指令（下称“目标指令”）的显式内存访问显现出较差的引用时间局部性。NTL 指令既不改变架构状态，也确实不改变目标指令的架构可见的影响。它提供四种变体：

NTL.P1 指令表示目标指令在内存层次的最内层私有缓存的容量内没有显现出时间局部性。NTL.P1 被编码为ADD *x0, x0, x2*。

NTL.PALL 指令表示目标指令在内存层次的任何私有缓存层次的容量内都没有显现出时间局部性。NTL.PALL 被编码为ADD *x0, x0, x3*。

NTL.S1 指令表示目标指令在内存层次的最内层共享缓存的容量内没有显现出时间局部性。NTL.S1 被编码为ADD *x0, x0, x4*。

NTL.ALL 指令表示目标指令在内存层次的任何缓存层次的容量内都没有显现出时间局部性。NTL.ALL 被编码为ADD *x0, x0, x5*。

---

*NTL* 指令可以被用于在数据流动、或遍历大型数据结构时，避免缓存污染，或者减少生产者－消费者交互中的延迟。

微架构可能使用 *NTL* 指令来通知缓存替换策略，或者决定分配到哪块缓存，或者避免缓存分配。例如，*NTL.P1* 可以表示一个实现不应当申请私有 *L1* 缓存中的一行，但应当在 *L2* 中（不论私有或共享）申请。在另一个实现中，*NTL.P1* 可以申请 *L1* 中的行，但是处于最近最少使用（*LRU*）的状态。

*NTL.ALL* 通常将通知实现不要申请缓存层次中的任何位置。编程人员应当为那些没有可利用的时间局部性的访问使用 *NTL.ALL*。

像任何 *HINT* 一样，这些指令可以被自由地忽略。因此，尽管它们是以基于缓存的内存层次的角度描述的，它们并不强制要求提供缓存。

一些实现的某些内存可能遵从这些 *HINT*，而对其它内存访问忽略它们：例如，通过在 *L1* 中以独占状态获取一个缓存行来实现 *LR/SC* 的实现可能忽略在 *LR* 和 *SC* 上的 *NTL* 指令，但是可能遵从 *AMO* 和常规加载与存储的 *NTL* 指令。

表 4.1列出了一些软件使用情况，以及可移植 软件（即，不会针对任何特定实现的内存层次进行调整的软件）在各情况中应当使用的推荐的 NTL 变体。

| 场景 | 推荐的 NTL 变体 |
|---|---|
| 访问一个64 KiB 256 KiB尺寸的工作集 | NTL.P1 |
| 访问一个256 KiB 1 MiB尺寸的工作集 | NTL.PALL |
| 访问一个尺寸超过1 MiB的工作集 | NTL.S1 |
| 没有可利用的时间局部性的访问（例如，流） | NTL.ALL |
| 访问一个竞争的同步变量 | NTL.PALL |

表 4.1: 为可移植软件推荐的在各种场景中采用的 NTL 变体。

缓存尺寸将在实现之间明显变化，因此表4.1中列出的工作集尺寸仅仅是粗略的指导意见。

表 4.2列出了一些样例内存层次，以及各 NTL 变体如何映射到各缓存层次的建议。该表也推荐了为实现所调整的软件在申请特定层次缓存时应当避免的 NTL 变体。例如，对于一个具有私有 L1 和共享 L2 的系统，表格推荐 NTL.P1 和 NTL.PALL，表示时间局部性不能被 L1 利用，而 NTL.S1 和 NTL.ALL 表示时间局部性不能被 L2 利用。进一步地，为这种系统所调整的软件应当使用 NTL.P1，以表示缺少可以被 L1 利用的时间局部性，或者应当使用 NTL.ALL 表示缺少可以被 L2 利用的时间局部性。

| 内存层次 | NTL 变体到实际缓存层次的推荐映射 | | | | 为显式缓存管理推荐的 NTL 变体 | | | |
|---|---|---|---|---|---|---|---|---|
| | P1 | PALL | S1 | ALL | L1 | L2 | L3 | L4/L5 |
| 常见场景 | | | | | | | | |
| 无缓存 | — | | | | *none* | | | |
| 仅有私有 L1 | L1 | L1 | L1 | L1 | ALL | — | — | — |
| 私有 L1，共享 L2 | L1 | L1 | L2 | L2 | P1 | ALL | — | — |
| 私有 L1，共享 L2/L3 | L1 | L1 | L2 | L3 | P1 | S1 | ALL | — |
| 私有 L1/L2 | L1 | L2 | L2 | L2 | P1 | ALL | — | — |
| 私有 L1/L2; 共享 L3 | L1 | L2 | L3 | L3 | P1 | PALL | ALL | — |
| 私有 L1/L2; 共享 L3/L4 | L1 | L2 | L3 | L4 | P1 | PALL | S1 | ALL |
| 不常见的场景 | | | | | | | | |
| 私有 L1/L2/L3; 共享 L4 | L1 | L3 | L4 | L4 | P1 | P1 | PALL | ALL |
| 私有 L1; 共享 L2/L3/L4 | L1 | L1 | L2 | L4 | P1 | S1 | ALL | ALL |
| 私有 L1/L2; 共享 L3/L4/L5 | L1 | L2 | L3 | L5 | P1 | PALL | S1 | ALL |
| 私有 L1/L2/L3; 共享 L4/L5 | L1 | L3 | L4 | L5 | P1 | P1 | PALL | ALL |

表 4.2: NTL 变体到各种内存层次的映射。

如果提供了 C 扩展，也会提供这些 HINT 的压缩变体：C.NTL.P1 被编码为C.ADD *x0, x2*；C.NTL.PALL 被编码为C.ADD *x0, x3*；C.NTL.S1 被编码为C.ADD *x0, x4*；以及 C.NTL.ALL 被编码为C.ADD *x0, x5*。

NTL 指令影响除 Zicbom 扩展中缓存管理指令外的所有内存访问指令。

---

在撰写本文时，对于这条规则还没有其它的例外，因此 *NTL* 指令会影响在基础 *ISA* 和 *A*、*F*、*D*、*Q*、*C* 及 *V* 标准扩展中定义的所有内存访问指令，也会影响那些在卷 *II* 中 *hypervisor* 扩展中定义的内存访问指令。

*NTL* 指令可以影响除 *Zicbom* 以外的缓存管理操作。例如，*NTL.PALL* 后跟 *CBO.ZERO* 可以表示该行应该在 *L3* 中分配并被清零，但是不能在 *L1* 或 *L2* 中分配。

当一个 NTL 指令被应用到 Zicbop 扩展中的预取提示时，它表示缓存行应当被预取到比 NTL 所指定的层次更外层的缓存中。

---

例如，在一个具有私有 *L1* 和共享 *L2* 的系统中，*NTL.P1* 后跟 *PREFETCH.R* 可以以读意图预取到 *L2* 中。

为了预取到最内层的缓存中，不要将 *NTL* 指令作为预取指令的前缀。

在某些系统中，*NTL.ALL* 后跟一条预取指令可以预取到内存控制器内部的缓存中或者预取缓冲区中。

不鼓励软件在一条 NTL 指令之后跟一条并不明确访问内存的指令。不遵守此建议可能会降低性能，但是没有架构上可见的影响。

在目标指令上发生陷入的事件中，不鼓励实现将 NTL 应用到陷入处理器中的第一条指令。相反，在这种情况下，建议实现忽略 HINT。

---

如果在一条 *NTL* 指令与它的目标指令之间发生了中断，那么执行通常将在目标指令处恢复。不被重新执行的 *NTL* 指令并不会改变程序的语义。

某些实现可能希望在目标指令被发现之前不处理 *NTL* 指令（例如，这样可以使 *NTL* 与其修改的内存访问相融合）。这种实现可能优先在 *NTL* 之前进行中断，而不是在 *NTL* 与内存访问之间中断。

---

由于 *NTL* 指令被编码为 *ADD*，因此它们可以在 *LR/SC* 循环中使用，而不回避向前执行保证。但是，由于在 *LR/SC* 循环中使用其它的加载和存储确实会避开向前执行保证，因此在这种循环中使用 *NTL* 的唯一原因是修改 *LR* 或 *SC*。

# 第五章 “Zihintpause”暂停提示（2.0 版本）

PAUSE 指令是一个表示当前硬件线程的指令引退率应当暂时减少或暂停的 HINT。它影响的持续时间必须是有界的，可以是零。没有架构状态被改变。

---

软件可以在执行自旋等待的代码序列时，使用 *PAUSE* 指令来减少能耗。在执行 *PAUSE* 时，多线程核心可以暂时地放弃执行资源，让给其它硬件线程。建议使 *PAUSE* 指令通常性地包含在自旋等待循环的代码序列中。

未来的扩展可能添加类似于 *x86 MONITOR/MWAIT* 指令的原语，这提供了一种更有效的机制来等待对特定内存位置的写入。然而，这些指令不会取代 *PAUSE*。当轮询非内存事件时、轮询多重事件时、或者软件不确切地知道它正在轮询什么事件时，*PAUSE* 是更合适的。

*PAUSE* 指令效果的持续时间，在实现内部和实现之间可以有显著变化。在典型的实现中，该持续时间应当远小于执行一次上下文切换的时间，可能多于一次片上缓存未命中延迟或一次对主内存无缓存访问的粗略次序。

一系列 *PAUSE* 指令可以被用于创建与 *PAUSE* 指令数目粗略成比例的累积延迟。然而，在可移植代码的自旋等待循环中，在重新评估循环条件之前应当仅使用一条 *PAUSE* 指令，否则硬件线程可能在某些实现中拖延比最优更长的时间，从而降低系统的性能。

PAUSE 被编码为 *pred*=W，*succ*=0，*fm*=0，*rd*=x0，和 *rs1*=x0。

---

*PAUSE* 被编码为 *FENCE* 操作码中的一条提示，因为某些实现有可能故意拖延 *PAUSE* 指令，直到完成尚未完成的内存事务。然而，由于后继集为空，*PAUSE* 并不强制要求任何特定的内存次序——因此，它确实是一个 *HINT*。

像其它 *FENCE* 指令一样，*PAUSE* 不能被用在 *LR/SC* 序列中而不避开向前执行保证。

前驱集 *W* 的选择是任意的，因为后继集为空。其它类似于 *PAUSE* 的 *HINT* 可能与其它前驱集一同编码。

# 第六章　RV32E 和 RV64E 基础整数指令集 (1.95 版本)

这章描述了一个 RV32E 和 RV64E 基础整数指令集的建议草案，它们是为嵌入式系统的微控制器设计的。RV32E 和 RV64E 分别是 RV32I 和 RV64I 的简化版本：仅有的改变是把整数寄存器的数目减少到了 16 个。这章仅仅概述了 RV32E/RV64E 和 RV32I/RV64I 之间的不同，并因此应当被放在第 二章和第 七章之后阅读。

---

*RV32E 被设计为，为嵌入式微控制器提供一个更小的基础核心。RV64E 也有兴趣用于大型 SoC 设计中的微控制器、以及减少高线程（highly thread）64 位处理器的上下文状态。*

*RV32E 和 RV64E 可以与所有当前的标准扩展相组合。*

## 6.1　RV32E 和 RV64E 编程模型

RV32E 和 RV64E 把整数寄存器的数目减少到 16 个通用目的寄存器，（`x0`–`x15`），这里 `x0` 是一个专用的零寄存器。

---

*我们已经发现，在小型 RV32I 内核设计中，较高的 16 个寄存器消费了除内存外的所有内核区域的大约四分之一，因此它们的移除节省了大约 25% 的内存区域，而内核的电量也相应地减少了。*

## 6.2　RV32E 和 RV64E 指令集编码

RV32E 及 RV64E 分别使用与 RV32I 和 RV64I 相同的指令集编码，但是只提供寄存器 `x0 - x15`。所有指定其它寄存器 `x16 - x31` 的编码都是保留的

本章的前一稿将所有使用 x16 - x31 寄存器的编码都可用于自定义的。这一版本采用了一种更加保守的方法，让这些编码被保留，以便以后可以在自定义空间或新的标准编码之间分配它们。

# 第七章　RV64I 基础整数指令集（2.1 版本）

这章描述了 RV64I 基础整数指令集，它是在第 二章中描述的 RV32I 变体之上构建的。这章只呈现了与 RV32I 的不同，所以应当与那篇更早的章节结合着阅读。

## 7.1　寄存器状态

RV64I 把整数寄存器和所支持的用户地址空间拓宽到 64 位（表 2.1中 XLEN=64）。

## 7.2　整数运算指令

大多数整数运算指令在 XLEN 位的值上操作。在 RV64I 中提供了额外的指令变体来操作 32 位的值，通过在操作码上添加“W”后缀来表示。这些“*W”指令忽略了它们的输入的高 32 位，并且总是产生 32 位有符号的值、把它们符号扩展到 64 位，也就是说，从位 XLEN-1 位到位 31 是相等的。

---

编译器和调用约定维持了一种不变性，即在 *64* 位寄存器中，所有的 *32* 位值都以一种符号扩展的格式被保持。甚至 *32* 位无符号整数也会把位 *31* 扩展到位 *63* *32*。因此，在无符号 *32* 位整数和有符号 *32* 位整数之间的转换是一个 *no-op*，从一个有符号 *32* 位整数转换到一个有符号 *64* 位整数也是如此。在这种不变性下，现有的 *64* 位宽 *SLTU* 和无符号分支比较仍然能正确地操作无符号 *32* 位整数。类似地，现有的在 *32* 位符号扩展整数上的 *64* 位宽逻辑操作保留了符号扩展属性。加法和移位需要少量的新指令（*ADD[I]W/SUBW/SxxW*），以确保 *32* 位值的合理的性能。

**整数寄存器 - 立即数指令**

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

ADDIW 是一个 RV64I 指令，它把符号扩展的 12 位立即数加到寄存器 *rs1*，并在 *rd* 中产生合适的 32 位符号扩展的结果。运算结果的低 32 位符号扩展到 64 位作为结果，而忽略了溢出。注意，*rd, rs1, 0* 把寄存器 *rs1* 的低 32 位的符号扩展写入寄存器 *rd*（汇编器伪指令 SEXT.W）。

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

按常量移位被编码为一种专门化的 I 类型格式，它使用与 RV32I 相同的指令操作码。对于 RV64I，被移位的操作数在 *rs1* 中，移位的数目被编码在 I 立即数域的低 6 位中。右移类型被编码在底第 30 位上。SLLI 是逻辑左移（移位后低位补零）；SRLI 是逻辑右移（移位后高位补零）；而 SRAI 是算数右移（原始符号位被复制到空白的高位中）。

SLLIW、SRLIW 和 SRAIW 是 RV64I 中独有的指令，它们的定义类似，但是在 32 位值上操作，并把它们的 32 位结果符号扩展到 64 位。带有 $imm[5] \neq 0$ 的 SLLIW、SRLIW 和 SRAIW 的编码是保留的。

---

先前，$imm[5] \neq 0$ 的 *SLLIW*、*SRLIW* 和 *SRAIW* 被定义为：引发非法指令异常，而现在它们被标记为保留的。这是一个向后兼容的改变。

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

LUI（加载高位立即数）使用与 RV32I 相同的操作码。LUI 把 32 位的 U 立即数放进寄存器 *rd* 中，并把最低的 12 位填充为零。该 32 位结果被符号扩展到 64 位。

AUIPC（加高位立即数到 `pc`）使用与 RV32I 相同的操作码。AUIPC 被用于构建关于 `pc` 的相对地址，并使用 U 类型格式。AUIPC 从 U 立即数形成 32 位偏移量，并把最低的 12 位填充为零，把结果符号扩展到 64 位，把它加到 AUIPC 指令的地址，然后把结果放进寄存器 *rd* 中。

---

注意，可以通过将 *LUI* 与 *LD* 配对、将 *AUIPC* 与 *JALR* 配对等方式形成的偏移量的地址集是 $[-2^{31}-2^{11}, 2^{31}-2^{11}-1]$。

**整数寄存器—寄存器操作**

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

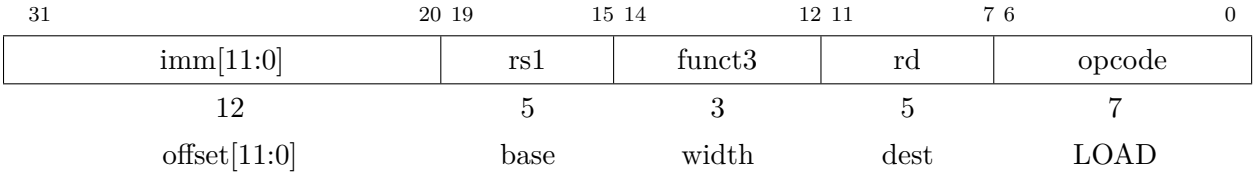
ADDW 和 SUBW 是 RV64I 独有的指令，它们的定义类似于 ADD 和 SUB，但是在 32 位值上操作，并产生有符号的 32 位结果。溢出被忽略，且结果的低 32 位被符号扩展到 64 位，并写到目的寄存器。

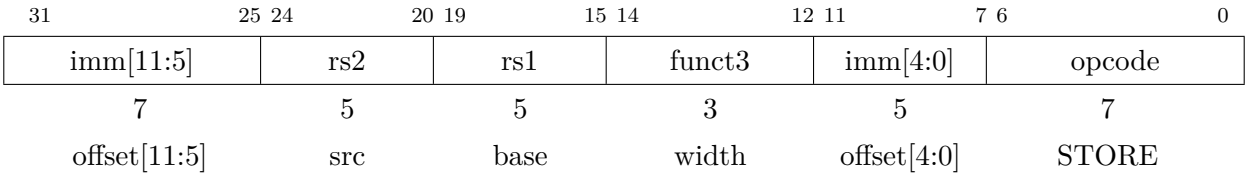
SLL、SRL 和 SRA 对寄存器 *rs1* 中的值实施逻辑左移、逻辑右移和算数右移，移位的数目保持在寄存器 *rs2* 中。在 RV64I 中，只有 *rs2* 的低 6 位被考虑用于移位数目。

SLLW、SRW 和 SRAW 是 RV64I 独有的指令，它们的定义类似，但是在 32 位值上操作，并把它们的 32 位结果符号扩展到 64 位。移位数量由 *rs2[4:0]* 给出。

## 7.3 加载和存储指令

RV64I 把地址空间扩展到了 64 位。执行环境将定义地址空间的哪部分对于访问是合法的。




对于 RV64I，LD 指令从内存加载一个 64 位的值到寄存器 *rd*。

对于 RV64I，LW 指令从内存加载一个 32 位的值，并把它符号扩展到 64 位，然后将其存储到寄存器 *rd*。另一方面，RV64I 的 LWU 指令则会对内存中的 32 位值使用零扩展。类似地，LH 和 LHU 被定义用于 16 位值，以及 LB 和 LBU 用于 8 位值。SD、SW、SH 和 SB 指令分别把寄存器 *rs2* 的低 64 位、32 位、16 位和 8 位值存储到内存。

## 7.4 “提示”指令

所有在 RV32I 中作为微架构 HINT 的指令（见 2.9节）也是 RV64I 中的 HINT。RV64I 中的额外的运算指令同时扩展了标准 HINT 和自定义 HINT 的编码空间。

表 7.1列出了所有的 RV64I HINT 代码点。91% 的 HINT 空间被保留用于标准 HINT，但是目前还没有被定义。其余的 HINT 空间被指定用于自定义 HINT：不会有标准 HINT 将被定义在这个子空间中。

| 指令 | 约束 | 代码点 | 目的 |
|---|---|---|---|
| LUI | *rd*=x0 | $2^{20}$ | 保留供未来标准使用 |
| AUIPC | *rd*=x0 | $2^{20}$ | |
| ADDI | *rd*=x0, 要么 *rs1*≠x0 要么 *imm*≠0 | $2^{17}-1$ | |
| ANDI | *rd*=x0 | $2^{17}$ | |
| ORI | *rd*=x0 | $2^{17}$ | |
| XORI | *rd*=x0 | $2^{17}$ | |
| ADDIW | *rd*=x0 | $2^{17}$ | |
| ADD | *rd*=x0, *rs1*≠x0 | $2^{10}-32$ | |
| ADD | *rd*=x0, *rs1*=x0, *rs2*≠x2–x5 | 28 | |
| ADD | *rd*=x0, *rs1*=x0, *rs2*=x2–x5 | 4 | (*rs2*=x2) NTL.P1<br>(*rs2*=x3) NTL.PALL<br>(*rs2*=x4) NTL.S1<br>(*rs2*=x5) NTL.ALL |
| SUB | *rd*=x0 | $2^{10}$ | 保留供未来标准使用 |
| AND | *rd*=x0 | $2^{10}$ | |
| OR | *rd*=x0 | $2^{10}$ | |
| XOR | *rd*=x0 | $2^{10}$ | |
| SLL | *rd*=x0 | $2^{10}$ | |
| SRL | *rd*=x0 | $2^{10}$ | |
| SRA | *rd*=x0 | $2^{10}$ | |
| ADDW | *rd*=x0 | $2^{10}$ | |
| SUBW | *rd*=x0 | $2^{10}$ | |
| SLLW | *rd*=x0 | $2^{10}$ | |
| SRLW | *rd*=x0 | $2^{10}$ | |
| SRAW | *rd*=x0 | $2^{10}$ | |
| FENCE | *rd*=x0, *rs1*≠x0, *fm*=0, 且 *pred*=0 或者 *succ*=0 | $2^{10}-63$ | |
| FENCE | *rd*≠x0, *rs1*=x0, *fm*=0, 且 *pred*=0 或者 *succ*=0 | $2^{10}-63$ | |
| FENCE | *rd*=*rs1*=x0, *fm*=0, *pred*=0, *succ*≠0 | 15 | |
| FENCE | *rd*=*rs1*=x0, *fm*=0, *pred*≠W, *succ*=0 | 15 | |
| FENCE | *rd*=*rs1*=x0, *fm*=0, *pred*=W, *succ*=0 | 1 | 暂停 |
| SLTI | *rd*=x0 | $2^{17}$ | |
| SLTIU | *rd*=x0 | $2^{17}$ | |
| SLLI | *rd*=x0 | $2^{11}$ | |

# 第八章　RV128I 基础整数指令集（1.7 版本）

“在计算机设计中只可能发生一个难以恢复的错误——没有足够的地址位用于内存编址和内存管理。” – Bell 和 Strecker，ISCA-3，1976 年。

这章描述了 RV128I，一个支持扁平 128 位地址空间的 RISC-V ISA 的变体。该变体是对现有的 RV32I 和 RV64I 设计的一种直接的外扩。

---

*扩展整数寄存器宽度的主要原因是为了支持更大的地址空间。还不清楚什么时候将会需要大于 64 位的扁平地址空间。在编写本手册时，世界上最快的超级计算机，经 Top500 基准的衡量，有超过1 PB的 DRAM，而且如果所有的 DRAM 都保留在单一地址空间中，将需要超过 50 位的地址空间。一些仓储级（warehouse-scale）的计算机甚至已经包含了更大数量的 DRAM，且新型高密度固态非易失性存储器和快速互联技术可能驱使着甚至更大内存空间的需求。超规模系统的研究把100 PB的内存系统作为目标，它占据了 57 位地址空间。根据历史的增长率，很可能在 2030 年以前就需要超过 64 位的地址空间了。*

*历史表明，无论何时，只要对超过 64 位地址空间的需要变得明确，架构师们都将重复关于替代扩展地址空间的激烈辩论，包括分段、96 位地址空间、和软件工作环境，直到最终，扁平 128 位地址空间被采纳为最简单和最佳的解决方案。*

*这时我们还没有冻结 RV128 规范，因为基于 128 位地址空间的实际用途，可能还有需要演化该设计。*

RV128I 以与 RV32I 上构建 RV64I 的相同的方法构建于 RV64I 之上，把整数寄存器扩展到 128 位（也就是说，XLEN = 128）。大多数整数运算指令是没有变化的，因为它们被定义为在 XLEN 位上操作。保留了 RV64I 在寄存器低位的 32 位值上操作的“*W”整数指令，但是现在把它们的结果从位 31 符号扩展到位 127 了。添加了一个新的“*D”整数指令集，它在 128 位整数寄存器的低位中的 64 位值上进行操作，并把结果从位 63 符号扩展到位 127。“*D”指令消耗了标准 32 位编码中的两个主要的操作码（OP-IMM-64 和 OP-64）。

---

*为了提升对 RV64 的兼容性，与处理 RV32 到 RV64 的做法相反，我们可以改变解码方式，比如把 RV64I 的 ADD 重命名为 64 位的 ADDD，并在先前的 OP-64 主操作码（现在重命名为 OP-128 主操作码）中添加一个 128 位的 ADDQ。*

按立即数移位（SLLI/SRLI/SRAI）现在使用 I 立即数的低 7 位进行编码，而可变的移位（SLL/SRL/SRA）使用移位数目源寄存器的低 7 位进行编码。

使用现有的 LOAD 主操作码添加了 LDU（双无符号加载）指令，随着新的 LQ 和 SQ 指令一起加载和存储四字值。SQ 被添加到 STORE 主操作码，同时 LQ 被添加到 MISC-MEM 主操作码。

浮点指令集没有变化，尽管 128 位 Q 浮点扩展现在可以支持 FMV.X.Q 和 FMV.Q.X 指令，以及来往于 T（128 位）整数格式的额外的 FCVT 指令。

# 第九章　用于乘法和除法的“M”标准扩展（2.0 版本）

这章描述了标准整数乘法和除法指令扩展，命名为“M”，包含了将两个整数寄存器中持有的值相乘或相除的指令.

---

我们将整数乘法和除法从基础中分离出来，以简化低端的实现，或者用于那些整数乘法和除法操作并不频繁的、或在相接的加速器中能更好地处理的应用。

## 9.1　乘法操作

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

MUL 在 *rs1* 和 *rs2* 上执行 XLEN 位 ×XLEN 位乘法，并将低 XLEN 位放入目的寄存器。MULH、MULHU 和 MULHSU 分别针对有符号数 × 有符号数、无符号数 × 无符号数、有符号 rs1× 无符号 rs2 执行相同的乘法，但是返回 2×XLEN 位结果的高 XLEN 位。如果同一个乘积的高位和低位都需要，那么推荐的代码序列是：MULH[[S]U] *rdh,rs1, rs2*; MUL *rdl, rs1, rs2*（源寄存器标识符必须次序相同，且 *rdh* 不能与 *rs1* 或 *rs2* 相同）。然后微架构可以把这些代码融合进单独的一次乘法操作，而不是两次分离的乘法。

---

*MULHSU* 被用在多字有符号乘法中，将被乘数（包含符号位）的最高位有效字与乘数（无符号的）较低位有效字相乘。

MULW 是一个 RV64 指令，它将源寄存器的低 32 位相乘，把符号扩展的低 32 位结果放入目的寄存器。

---

*在 RV64 中，MUL 可以被用于获得 64 位乘积的高 32 位，但是有符号参数必须是合适的 32 位有符号值，反之无符号参数必须清除它们的高 32 位。如果参数不知道是符号扩展还是零扩展的，一个备选方案是把两个参数都向左移位 32 位，然后使用 MULH[[S]U]。*

## 9.2 除法操作

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

DIV 和 DIVU 在 *rs1* 和 *rs2* 上执行 XLEN 位与 XLEN 位的有符号和无符号整数除法，结果向零取整。REM 和 REMU 提供了对应的除法操作的余数。对于 REM，结果的符号等于被除数的符号。

---

对于有符号除法和无符号除法，都有 被除数 = 除数 × 商 + 余数

如果同一个除法的商和余数都需要，推荐的代码序列是：DIV[U] *rdq, rs1, rs2*; REM[U] *rdr, rs1, rs2* （*rdq* 不能与 *rs1* 或 *rs2* 相同）。然后微架构可以把这些代码融合为单独的一次除法操作，而不是执行两次分离的除法。

DIVW 和 DIVUW 是 RV64 的指令，它们将 *rs1* 的低 32 位与 *rs2* 的低 32 位分别视为有符号整数和无符号整数并相除，把 32 位商放入 *rd*，并符号扩展到 64 位。REMW 和 REMUW 是 RV64 的指令，它们分别提供对应的有符号余数和无符号余数操作。REMW 和 REMUW 都总是把 32 位结果符号扩展到 64 位，包括除数为零时。

表 9.1中总结了除数为零和除法溢出的语义。如果除数为零，商的所有位都被设置为 1，余数等于被除数。有符号的除法溢出只发生在最大负数被 −1 除的时候。溢出的有符号除法的商等于被除数，而余数为零。无符号除法不可能发生溢出。

---

我们考虑过在整数被零除的时候产生异常，这些异常在大多数执行环境中会引发一个陷入。然而，这将是标准 *ISA* 中仅有的算数陷入（浮点异常会设置标志和写默认值，但是不会引起陷入），而对于这种情况，将需要语言解释器来与执行环境的陷入处理程序进行交互。此外，如果

| Condition | 被除数 | 除数 | DIVU[W] | REMU[W] | DIV[W] | REM[W] |
|---|---|---|---|---|---|---|
| Division by zero | $x$ | 0 | $2^L - 1$ | $x$ | $-1$ | $x$ |
| Overflow (signed only) | $-2^{L-1}$ | $-1$ | – | – | $-2^{L-1}$ | 0 |

表 9.1: 除数为零和除法溢出的语义。L 是操作数的位宽度：或者是 XLEN（对于 DIV[U] 和 REM[U]），或者是 32（对于 DIV[U]W 和 REM[U]W）

语言标准强制要求除数为零的异常必须引起控制流的立即改变，那么只需要为每个除法操作添加一条分支指令，而这个分支指令可以在除法之后被插入，并且通常应当非常大概率地被预测为不执行，几乎不增加运行时的负载。

为了简化除法器电路，除数为零时，无论无符号还是有符号除法都返回所有位被设置为 *1* 的值。全 *1* 值既是无符号除法返回的自然值，代表了最大的无符号数，也是简单无符号除法器实现的自然结果。有符号除法经常使用无符号除法电路实现，并指定了相同的溢出结果来简化硬件。

## 9.3 Zmmul 扩展（1.0 版本）

Zmmul 扩展实现了 M 扩展的乘法子集。它添加了定义在第 9.1 节 9.1中的所有指令，即：MUL、MULH、MULHU、MULHSU、和（仅用于 RV64 的）MULW。这些编码与那些对应的 M 扩展的指令是相同的。

---

*Zmmul* 扩展支持那些需要乘法操作但不需要除法操作的低成本实现。对于许多微控制器应用，除法操作太不频繁，导致除法器硬件的开销无法证明是正当的。相比之下，乘法操作更加频繁，使得乘法器硬件的开销更加正当。消除除法但保留乘法，令简单的 *FPGA* 软核特别获益，因为许多 *FPGA* 提供硬布线的乘法器，但是需要在软逻辑中实现除法器。

# 第十章　用于原子指令的“A”标准扩展（2.1 版本）

命名为“A”的标准原子指令扩展包含了原子性的读-修改-写内存指令，以支持运行在相同内存空间中的多个 RISC-V 硬件线程之间的同步。该扩展提供了两种形式的原子指令，即“加载-保留/存储-条件”指令和“原子性获取并操作内存”指令。原子指令的这两种形式都支持各种内存一致性次序，包括无序的、获取的、释放的、和顺序的一致性语义。这些指令允许 RISC-V 支持 RCsc 内存一致性模型 [6]。

---

在大量辩论之后，语言社区和架构社区似乎最终商定了将释放一致性作为标准内存一致性模型，并因此 *RISC-V* 原子性支持就是围绕这个模型构建的。

## 10.1　指定原子指令的次序

基础的 RISC-V ISA 有一个宽松的内存模型，其中 FENCE 指令被用于强制执行额外的次序约束。地址空间被执行环境划分为内存领域和 I/O 领域，而 FENCE 指令提供了选项，以对这两个地址领域中的一个、或两者同时的访问进行排序。

为了对释放一致性 [6] 提供更有效的支持，每个原子指令有两个位，*aq* 和 *rl*，用于指定额外的内存次序约束，正如其它 RISC-V 硬件线程所看到的那样。位次序访问两个地址领域——内存或 I/O——中的哪一个，依赖于原子指令正在访问的地址领域。对另一个领域的访问不隐含次序约束，而应当使用 FENCE 指令跨两个领域排序。

如果两个位都被清除，则在原子内存操作上不会被施加额外的次序约束。如果只设置了 *aq* 位，原子内存操作被视为一次获取访问，也就是说，在这个 RISC-V 硬件线程上，在获取内存操作之前不会观测到随后发生的内存操作。如果只设置了 *rl* 位，原子内存操作被视为一次释放访问，也就是说，在这个 RISC-V 硬件线程上，在任何更早的内存操作之前不会观测到释放内存操作发生。如果 *aq* 位和 *rl* 位都被设置了，原子内存操作是顺序一致性的，在同一个 RISC-V 硬件线程中，对于同一个地址领域，不能在任何更早的内存操作之前、或在任何更迟的内存操作之后观测到该原子内存操作发生。

## 10.2 加载-保留/存储-条件指令

| 31 27 | 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|---|
| funct5 | aq | rl | rs2 | rs1 | funct3 | rd | opcode |
| 5 | 1 | 1 | 5 | 5 | 3 | 5 | 7 |
| LR.W/D | ordering | | 0 | addr | width | dest | AMO |
| SC.W/D | ordering | | src | addr | width | dest | AMO |

加载-保留（LR）和存储-条件（SC）指令在一个内存字或双字上实施复杂的原子内存操作。LR.W 从 *rs1* 中的地址处加载一个字，把符号扩展的值放入 *rd*，并注册一个保留集——归入地址字的字节的一个字节集合。SC.W 有条件地将 *rs2* 中的字写到 *rs1* 中的地址：当且仅当保留仍然有效且保留集包含正在写的字节时，SC.W 成功。如果 SC.W 成功，指令把 *rs2* 中的字写到内存，并把 *rd* 写零。如果 SC.W 失败，指令不会写内存，它向 *rd* 写一个非零值。不管成功还是失败，执行一次 SC.W 指令都会使这个硬件线程持有的任何保留失效。LR.D 和 SC.D 对双字采取类似的行为，并只在 RV64 上可用。对于 RV64，LR.W 和 SC.W 对放入 *rd* 的值进行符号扩展。

---

“比较并交换”（*CAS*）和 *LR/SC* 都能够被用于构建无锁的数据结构。在紧张的讨论之后，我们选择了 *LR/SC*，是由于几个原因：*1*）*LR/SC* 因为监视所有的对地址的访问、而不只是检查数值的变化，可以避免 *CAS* 存在的 *ABA* 问题；*2*）在已经需要一个不同于内存系统的消息格式的基础上，*CAS* 还将需要一个新的整数指令格式来支持三个源操作数（地址、比较的值、交换的值），这将使微架构复杂化；*3*）更进一步地，为了避免 *ABA* 问题，其它系统提供了一个双宽度 *CAS*（*DW-CAS*），以允许计数器可以沿着数据字测试和增长。这需要在一条指令中读五个寄存器并写两个寄存器，而且也需要一个新的更大的内存系统消息格式，远比实现要复杂；*4*）*LR/SC* 为许多原语提供了一个更加有效的实现，因为它只需要加载一次，与之相反的是，*CAS* 需要加载两次（一次在 *CAS* 指令之前加载以获得推测计算的值，然后第二次加载作为 *CAS* 指令的一部分以检查值是否在更新之前被改变了）。

与 *CAS* 相比，*LR/SC* 的主要劣势是活锁，我们在特定环境下通过最终推进的一种结构化的保证来避免它，如下文所描述的那样。另一个问题是，当前 *x86* 架构和它的 *DW-CAS* 的影响是否会将同步库和其它假定 *DW-CAS* 是基本机器原语的软件的移植复杂化。一个可能的缓解因素是，当前向 *x86* 添加的事务内存指令，它可能导致一次从 *DW-CAS* 的迁移。

更一般地，多字原子原语是令人向往的，但是关于这应当采取什么形式仍然有相当大的争议，并且保证向前进度会增加系统的复杂性。我们当前的想法是，沿着原始事务内存提案的内容，包括一个小型的有限容量的事务内存缓冲，作为一个可选的标准扩展“*T*”。

失败代码值 1 编码了一个未指定的失败。其它的失败代码目前被保留，可移植的软件应当仅仅假定，失败代码将是非零的。

---

我们保留了一个失败代码 *1* 来表示“未指定的”，这样简单的实现可以使用 *SLT/SLTU* 指令所需的现有的 *mux* 来返回这个值。*ISA* 的未来版本或扩展中可能会定义更具体的失败代码。

对于 LR 和 SC，A 扩展需要 *rs1* 中持有的地址自然对齐到操作数的尺寸（也就是说，64 位字对齐到 8 字节，32 位字对齐到 4 字节）。如果地址没有自然对齐，将产生一个地址未对齐异常或者一个访问故障异常。如果未对齐的访问不宜被模拟，而除了未对齐之外内存访问都能完成，那么可以为内存访问生成访问故障异常。

---

在大多数系统中，模拟未对齐的 *LR/SC* 序列是不实际的。

未对齐的 *LR/SC* 序列也增加了一次访问多个保留集的可能性，这是现有的定义没有提供的。

实现可以在每个 LR 上注册一个任意大的保留集，提供包括被编址的数据字或双字的所有字节的保留。SC 可以只与程序次序中最相近的 LR 配对。SC 可能成功，当前仅当：没有从另一个硬件线程到该保留集的存储能够在 LR 和 SC 之间被观察到，且以程序次序，在其 LR 和它自己之间没有其它的 SC。SC 可能成功，当且仅当：在 LR 和 SC 之间不会观察到，发生从硬件线程以外的设备到被 LR 指令所访问的字节的写入。注意这个 LR 的有效地址和数据尺寸可能已经不同了，但是保留了 SC 的地址，将之作为保留集的一部分。

---

根据这个模型，在带有内存事务的系统中，如果更早的 *LR* 使用不同的虚拟地址别名预约了相同的位置，那么允许 *SC* 成功；但是如果虚拟地址是不同的，那么也允许失败。

---

为了顾及遗留的设备和总线，从 *RISC-V* 硬件线程以外的设备的写只需要在它们与由 *LR* 所访问的字节重叠时，将预约无效化即可。当它们访问保留集中的其它字节时，这些写不需要将预约无效化。

SC 必定失败，如果：以程序次序，地址没有在最近的 LR 的保留集中。SC 必定失败，如果：在 LR 和 SC 之间可以观察到有从其它硬件线程到保留集的存储发生。SC 必定失败，如果：在 LR 和 SC 之间可以观察到有从其它设备到 LR 所访问的字节的写发生。（如果这个设备写了保留集但是没有写由 LR 所访问的字节，SC 可能失败，也可能不失败。）SC 必定失败，如果：以程序次序，在 LR 和 SC 之间有另一个 SC（对任何地址）。在 17.1节中，“原子性公理”定义了对成功的 LR/SC 序列的原子性需求的精确陈述。

---

平台应当提供一种定义保留集的尺寸和形状的方法。一个平台的规范可能约束保留指令集的尺寸和形状。

---

对内存的划痕字的存储-条件指令应当被用于使任何现有的加载保留强制失效：

- 在抢占式上下文切换期间，和 *during a preemptive context switch, and*
- 如果有必要，在改变虚拟地址到物理地址映射时，例如，当迁移可能包含一个活动的保留的页时。*if necessary when changing virtual to physical address mappings, such as when migrating pages that might contain an active reservation.*

如果一个 *LR* 或 *SC* 暗示，硬件线程当时只持有一个保留，并且以程序次序，*SC* 只能与最接近的 *LR* 配对，且 *LR* 只能与接下来的下一个 *SC* 配对，那么当硬件线程执行该 *LR* 或 *SC* 时，硬件线程的预约被无效化。这是对 *17.1*节中原子性公理的一个限制，该公理确保软件正确地运行在以这种方式操作的期望的常见实现上。

在建立了保留的 LR 指令之前，其它 RISC-V 硬件线程永远不可以观测到 SC 指令。通过设置 LR 指令的 *aq* 位，可以赋予 LR/SC 序列获取的语义。通过设置 SC 指令的 *rl* 位，可以赋予 LR/SC 序列释放的语义。设置 LR 指令的 *aq* 位，并设置 SC 指令的 *aq* 位和 *rl* 位，使 LR/SC 序列顺序一致，意味着它不能被相同硬件线程的更早的或更迟的内存操作重新排序。

如果在 LR 和 SC 上都没有设置任何位，可以在来自同一 RISC-V 硬件线程的周围的内存操作之前或之后观测到 LR/SC 序列的发生。当 LR/SC 序列被用于实现并行规约操作时，这可以是合适的。

软件不应当设置 LR 指令的 *rl* 位，除非也设置了 *aq* 位；软件也不应当设置 SC 指令的 *aq* 位，除非也设置了 *rl* 位。LR.*rl* 和 SC.*aq* 指令不保证提供任何比那些位都被清除的指令更强的次序，但是可能会导致更低的效率。

```
    # a0 持有内存位置的地址  holds address of memory location
    # a1 持有期望的(expected)值  holds expected value
    # a2 持有需要的(desired)值  holds desired value
    # a0 持有返回值，如果成功则为零，否则为非零。  holds return value, 0 if successful, !0
cas:
    lr.w t0, (a0)        # 加载原始值。  Load original value.
    bne t0, a1, fail     # 不匹配，所以失败。  Doesn't match, so fail.
    sc.w t0, a2, (a0)    # 尝试更新。  Try to update.
    bnez t0, cas         # 如果存储条件失败，那么重试  Retry if store-conditional failed.
    li a0, 0             # 设置返回值为成功。  Set return to success.
    jr ra                # 返回。 Return.
fail:
    li a0, 1             # 设置返回值为失败。  Set return to failure.
    jr ra                # 返回。  Return.
```

图 10.1: 使用 LR/SC 的比较与交换功能的样例代码 Sample code for compare-and-swap function using LR/SC.

LR/SC 可以被用于构造无锁的数据结构。图 10.1显示了一个使用 LR/SC 来实现比较与交换功能的例子。如果被内联，比较与交换功能只需要采用四个指令。

## 10.3 存储-条件指令的最终正确完成

标准 A 扩展定义了受约束的 *LR/SC 循环*，它有如下的性质：

- 该循环只包含了一个 LR/SC 序列，和在失败的情况下进行重新尝试该序列的代码，并且必须包含至多 16 条在内存中顺序放置的指令。
- 一个 LR/SC 序列以一条 LR 指令开始，以一条 SC 指令结束。在 LR 和 SC 指令之间执行的动态代码只能包含来自基础“I”指令集的指令，不包括加载、存储、向后跳转、执行向后分支、JALR、FENCE 和 SYSTEM 指令。如果支持“C”扩展，那么前面提到的“I”指令的压缩形式也是被允许的。
- 重新尝试一次失败的 LR/SC 序列的代码可以包含向后跳转和/或分支以重复 LR/SC 序列，但如果不包含，那么与 LR 和 SC 之间的代码受到相同的约束。
- LR 和 SC 的地址必须列于带有 *LR/SC 最终属性*的内存区域之中。具有这种属性的区域的通信由执行环境负责。
- SC 必须与同一硬件线程所执行的最近一次 LR 具有相同的有效地址和相同的数据尺寸。

不在受约束的 LR/SC 循环中的 LR/SC 序列是*不受约束的*。不受约束的 LR/SC 序列可能在某些实现的某些尝试上成功，但是可能在其它实现上永远不成功。

---

*我们限制了 LR/SC 循环的长度来适合基础 ISA 中的 64 个连续指令字节，以避免对指令缓存、TLB 尺寸和关联性的过度限制。类似地，在追踪自由缓存中的预约的简单实现中，我们不允许循环中有其它的加载和存储，以避免限制数据-缓存的关联性。对分支和跳转的约束限制了本可以花费在序列中的时间。在缺少合适的硬件支持的实现上，不允许浮点操作和整数乘法/除法，以简化操作系统对这些指令的模拟。*

*不禁止软件使用不受约束的 LR/SC 序列，但是可移植的软件必须检测序列重复失败的情况，然后退回到不依赖于不受约束的 LR/SC 序列的备用代码序列。实现可以无条件地令任何不受约束的 LR/SC 序列失败。*

如果一个硬件线程 *H* 进入了一个受约束的 LR/SC 循环，执行环境必须保证下列事件之一能最终发生：

- *H* 或某些其它的硬件线程对 *H* 的受约束的 LR/SC 循环中的 LR 指令的保留集执行了一次成功的 SC。
- 某些其它硬件线程对 *H* 的受约束的 LR/SC 循环中的 LR 指令的保留集执行了一次无条件存储或 AMO 指令，或者系统中的某些其它设备写了该保留集。
- *H* 执行了一次分支或跳转而退出了受约束的 LR/SC 循环。*H* executes a branch or jump that exits the constrained LR/SC loop.
- *H* 陷入。*H* traps.

---

注意，只要不违背前面提到的各项保证，这些定义允许实现偶尔以任何原因令 *SC* 指令失败。

---

作为终结性保证的结果，如果执行环境的某些硬件线程正在执行受约束的 *LR/SC* 循环，而没有其它硬件线程或设备在该执行环境中对保留集执行无条件存储或 *AMO*，那么至少一个硬件线程将最终退出它的受约束的 *LR/SC* 循环。反之，如果其它硬件线程或设备持续写保留集，不保证任何硬件线程将退出它的 *LR/SC* 循环。

加载和加载-保留指令本身不会阻碍其它硬件线程的 *LR/SC* 序列的进程。我们注意到这个约束意味着，除此之外，其它硬件线程执行的加载和加载保留指令（可能在同一个核中）不能无限阻碍 *LR/SC* 进程。例如，由共享缓存的另一个硬件线程引起的缓存收回不能无限地阻碍 *LR/SC* 进程。典型地，这意味着对保留的追踪是独立于任何共享缓存的收回的。类似地，由硬件线程内的推测性执行引起的缓存缺失不能无限地阻碍 *LR/SC* 进程。

这些定义承认，即使进程最终完成，*SC* 指令也可能由于实现的原因而貌似失败。

---

*CAS* 的一个优势是，它保证了某些硬件线程最终完成进程，尽管在某些系统上 *LR/SC* 原子性序列可能无限期地活锁。为了避免这个问题，我们为特定的 *LR/SC* 序列添加了一个活锁自由的结构性保证。

这个规范的更早的版本推行了一个更强的饥饿-自由保证。然而，较弱的活锁-自由保证对于实现 *C11* 和 *C++11* 语言已经足够，并且在某些微架构样式中相当更容易被提供。

## 10.4 原子内存操作

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

原子内存操作（AMO）指令为多处理器同步执行读-修改-写操作，并被编码为 R 类型指令格式。这些 AMO 指令从 *rs1* 中的地址处原子性地加载一个数据值，把该值放进寄存器 *rd* 中，对被加载的值和 *rs2* 中的原有值使用一个二进制操作符，然后把结果存回 *rs1* 中的原始地址。AMO 既可以操作在内存中的 64 位字上（仅限 RV64），也可以操作在 32 位字上。对于 RV64，32 位的 AMO 总是把放入 rd 中的值进行符号扩展，并忽略 *rs2* 的原始值的高 32 位。

对于 AMO，A 扩展需要 *rs1* 中持有的地址被自然地对齐到操作数的尺寸（也就是说，对于 64 位字是 8 字节对齐，对于 32 位字是 4 字节对齐）。如果地址没有自然对齐，将生成一个地址未对齐异常或一个访问故障异常。如果未对齐的访问不宜被模拟，而除了未对齐之外的内存访问都能完成，那么可以为内存访问生成访问故障异常。第 二十三章里描述的“Zam”扩展放松了这个需求并指定了未对齐的 AMO 的语义。

支持的操作有：交换、整数加法、按位 AND、按位 OR、按位 XOR、有符号/无符号整数的取最大值/取最小值。如果没有次序约束，这些 AMO 可以被用于实现并行规约操作，通常情况下，返回值将通过写到 `x0` 而被弃置。

---

我们提供了“获取并操作”样式的原子性原语，因为它们比 *LR/SC* 或 *CAS* 更适合高度并行化的系统。一个简单的微架构可以使用 *LR/SC* 原语实现 *AMO*，如果实现提供 *AMO* 最终完成的保证。更复杂的实现也可以在内存控制器出实现 *AMO*，并且能够当目的寄存器是 `x0` 时，对获取原始值进行优化。

选择 *AMO* 的集合以有效地支持 *C11/C++11* 原子性内存操作，也为了支持内存中的并行规约。*AMO* 的另一个使用是提供对 *I/O* 空间中的内存映射设备寄存器的原子性更新（例如，设置位、清除位、或者切换位）。

为了帮助实现多处理器同步，AMO 有选择地提供了释放一致性语义。如果设置了 *aq* 位，那么这个 RISC-V 硬件线程中，不会观测到有比 AMO 更迟的内存操作在 AMO 之前发生。相对地，如果设置了 *rl* 位，那么其它 RISC-V 硬件线程将不会在这个 RISC-V 硬件线程中比 AMO 更早的内存访问之前观测到 AMO。在一个 AMO 上同时设置 *aq*q 和 *rl* 位会使序列具有顺序一致性，意味着它不能与来自相同硬件线程的更早的或更迟的内存操作被重新排序。

---

*AMO* 为高效地实现 *C11* 和 *C++11* 内存模型而设计。尽管 *FENCE R, RW* 指令足以实现获取操作，以及 *FENCE RW, W* 足以实现释放操作，但与设置对应的 aq 或 rl 位的 *AMO* 相比，它们都意味着额外的不必要的排序。

图 10.2中显示了一个通过“测试与测试与设置”自旋锁来保护关键节的示例代码序列。注意第一个 AMO 被标记了 *aq*，是为了将锁的获得排在关键节之前，而第二个 AMO 被标记了 *rl*，是为了将关键节排在锁的释放之前。

---

我们推荐为锁的获取和释放使用上面显示的 *AMO* 交换用语，以简化推测锁省略 *[17]* 的实现。

“A”扩展中的指令也可以被用于提供顺序一致性的加载和存储。顺序一致性加载可以用一个设置了 *aq* 和 *rl* 的 LR 实现。顺序一致性存储可以用一个 AMOSWAP 实现，它把旧的值写到 x0，并设置 *aq* 和 *rl*。

```
    li          t0, 1       # 初始化交换的值。 Initialize swap value.
again:
    lw          t1, (a0)    # 检查锁是否被占用。 Check if lock is held.
    bnez        t1, again   # 如果锁被占用则重试。 Retry if held.
    amoswap.w.aq t1, t0, (a0) # 尝试获取锁。 Attempt to acquire lock.
    bnez        t1, again   # 如果锁被占用则重试。 Retry if held.
    # ...
    # 关键小节 Critical section.
    # ...
    amoswap.w.rl x0, x0, (a0) # 通过存储0来释放锁。 Release lock by storing 0.
```

图 10.2: 互斥的样例代码。a0 包含了锁的地址。Sample code for mutual exclusion. a0 contains the address of the lock.

# 第十一章 控制与状态寄存器（CSR）指令 “Zicsr”(2.0 版本)

RISC-V 定义了一个独立的地址空间，包含与各硬件线程相关联的 4096 个控制和状态寄存器。这章定义了操作在这些 CSR 上的 CSR 指令的完整集合。

---

*CSR* 主要被用于特权架构，但同时也在非特权代码中有一些使用，包括用于计数器和计时器，以及浮点状态。

计数器和计时器不再被认为是标准基础 *ISA* 的强制性部分，因此访问它们所需要的 *CSR* 指令已经从基础 *ISA* 章节 二被移出，进入了这个独立的章节。

## 11.1 CSR 指令

所有的 CSR 指令自动地读取-修改-写入一个单独的 CSR，指令的位 31-20 持有的 12 位 *csr* 域中编码了 CSR 的标识符。立即数形式使用 5 位的零扩展立即数，编码在 rs1 域中。

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

CSRRW（原子性读/写 CSR）指令自动地交换 CSR 和整数寄存器中的值。CSRRW 读取 CSR 的旧值，把该值零扩展到 XLEN 位，然后把它写到整数寄存器 *rd*。*rs1* 中的初始值被写到 CSR。如果 *rd*=x0，那么指令不应当读 CSR，也不应当引起任何可能在读 CSR 时发生的副作用。

CSRRS（原子性读和设置 CSR 位）指令读取 CSR 的值，把该值零扩展到 XLEN 位，然后把它写到整数寄存器 *rd*。整数寄存器 *rs1* 中的初始值被视为位掩码，它指定要在 CSR 中设置的位的位置。任何在 *rs1* 中为高的位将引起 CSR 中对应的位（如果它可写的话）被设置。CSR 中的其它位不会被显式地写入。

CSRRC（原子性读和清除 CSR 位）指令读取 CSR 的值，把该值零扩展到 XLEN 位，然后把它写到整数寄存器 *rd*。整数寄存器 *rs1* 中的初始值被视为位掩码，它指定了要在 CSR 中被清除的位的位置。任何在 *rs1* 中为高的位将引起 CSR 中对应的位（如果它是可写的话）被清除。CSR 中的其它位不会被显式地写入。

对于 CSRRS 和 CSRRC，如果 *rs1*=x0，那么指令将完全不会写 CSR，并因此也应当既不会引起任何只可能在写 CSR 时发生的副作用，也不会在访问只读 CSR 时产生非法指令异常。不管 *rs1* 和 *rd* 域如何设置，CSRRS 和 CSRRC 都总是读取已编址的 CSR，并引起任何读的副作用。注意如果 *rs1* 指定了一个持有 x0 以外的零值的寄存器，那么指令将仍然尝试把未修改的值写回到 CSR，并将引起任何随之而来的副作用。一个 *rs1*=x0 的 CSSRW 将尝试向目的 CSR 写入零。

CSRRWI、CSRRSI 和 CSRRCI 变体分别与 CSRRW、CSRRS 和 CSRRC 相似，除了它们使用一个 XLEN 位的值来更新 CSR，这个值通过零扩展编码在 *rs1* 中的一个 5 位的无符号立即数（uimm[4:0]）域得到，而不是来自一个整数寄存器。对于 CSRRI 和 CSRRCI，如果 uimm[4:0] 域是零，那么这些指令将不会写 CSR，并且应当既不引起任何只可能在写 CSR 时发生的副作用，也不在访问只读 CSR 时产生非法的指令异常。对于 CSRRWI，如果 *rd*=x0，那么指令不应当读 CSR，也不应当跟引起任何可能在读 CSR 时发生的副作用。不论 *rd* 和 *rs1* 域如何，CSRRSI 和 CSRRCI 都将总是读 CSR，和引起任何读的副作用。

表 11.1总结了 CSR 指令在它们是否读和/或写 CSR 方面的行为。

对于任何由于具有特定值的 CSR 而发生事件或后果，如果一次写入 CSR 给了它该值，那么导致的事件或后果被称为该写入的间接效果。RISC-V ISA 不认为一个 CSR 写入的间接效果是该写入的副作用。

---

*CSR 访问的一个副作用的例子是，如果从一个指定 CSR 读取将导致灯泡打开，而向同一个 CSR 写入一个奇数值将导致灯泡关闭。假定写入一个偶数值没有影响。在这种情况下，读取和写入都有控制灯泡是否点亮的副作用，因为这个条件不仅仅根据 CSR 的值决定。（注意，在将一个奇数值写入 CSR 来关闭灯之后，再读来打开灯，重新写相同的奇数值会导致灯再次关闭。因此，在最后一次写入时，不是 CSR 值的变化关掉了灯。）*

| 寄存器操作数 | | | | |
|---|---|---|---|---|
| 指令 | *rd* 是 x0 | *rs1* 是 x0 | 读 CSR | 写 CSR |
| CSRRW | 是 | – | 否 | 是 |
| CSRRW | 否 | – | 是 | 是 |
| CSRRS/CSRRC | – | 是 | 是 | 否 |
| CSRRS/CSRRC | – | 否 | 是 | 是 |
| 立即数操作数 | | | | |
| 指令 | *rd* 是 x0 | *uimm*=0 | 读 CSR | 写 CSR |
| CSRRWI | 是 | – | 否 | 是 |
| CSRRWI | 否 | – | 是 | 是 |
| CSRRSI/CSRRCI | – | 是 | 是 | 否 |
| CSRRSI/CSRRCI | – | 否 | 是 | 是 |

表 11.1: 决定一条 CSR 指令是否读或写指定 CSR 的条件。

另一方面，如果在某个特定 *CSR* 的值为奇数时操纵灯泡，那么打开和关闭灯泡不会被视为写入 *CSR* 的副作用，仅仅是这种写入的间接效果。

更具体地，第二卷中定义的 *RISC-V* 特权架构表明，*CSR* 值的特定组合会导致陷入的发生。当对 *CSR* 的一次显式写入创造了触发陷入的条件时，该陷入不被认为是写入的副作用，而仅仅是间接效果。

标准 *CSR* 没有任何关于读的副作用。标准 *CSR* 可能有关于写的副作用。自定义扩展可能添加某些 *CSR*，其访问具有关于读或写的副作用。

一些 CSR，例如指令引退计数器、指令返回（instret），可能因为指令执行的副作用而被修改。在这些情况中，如果一条 CSR 访问指令读了一个 CSR，它读取值要优先于指令的执行。如果一条 CSR 访问指令写这样的一个 CSR，那么写被完成，而不是执行自增。特别地，被一条指令写到 `instret` 的值将是下一条指令所读到的值。

读 CSR 的汇编器伪指令，CSRR *rd, csr*，被编码为 CSRRS *rd, csr, x0*。写 CSR 的汇编器伪指令，CSRW *csr, rs1*，被编码为 CSRRW *x0, csr, rs1*，同时 *csr, uimm* 被编码为 CSRRWI *x0, csr, uimm*。

当旧的值不需要的时候，进一步定义了设置和清除 CSR 中的位的汇编器伪指令：CSRS/CSRC *csr, rs1*; CSRSI/SCRCI *csr, uimm*。

### CSR 访问排序

每个 RISC-V 硬件线程通常按照程序的执行次序观察自己的 CSR 访问、包括隐式 CSR 访问。特别地，除非另有规定，CSR 访问的执行是在任何按程序次序在先的、其行为将更改 CSR 状态或

被 CSR 状态更改的指令被执行之后，且在任何按程序次序后继的、其行为将更改 CSR 状态或被 CSR 状态更改的指令被执行之前。此外，显式 CSR 读返回指令执行前的 CSR 状态，而显式 CSS 写禁止并重写同一指令对于相同 CSR 的任何隐式写入或修改。

同样，任何来自显式 CSR 访问的副作用通常都被观察到以程序次序同步发生。除非另有规定，任何这种副作用的全部后果都可以被下一条指令观察到，并且先前指令不可能乱序地观察到任何后果。（注意之前对于 CSR 写入的副作用和间接效果所做的区分。）

对于 RVWMO 内存一致性模型（第 十七章），CSR 的访问默认是弱有序的，所以其它硬件线程或设备可以以一种不同于程序次序的次序观测到 CSR 的访问。另外，CSR 的访问并非按照显式内存访问排序，除非 CSR 的访问修改了实施显式内存访问的指令的执行行为，或者除非 CSR 的访问和显式内存访问被内存模型定义的语法依赖或本手册第二卷中内存排序 PMA 节定义的排序需求所排序。为了在所有其它情况中强制排序，软件应当在相关访问之间执行 FENCE 指令。处于 FENCE 指令的目的，CSR 读访问被归类为设备输入（I），而 CSR 写访问被归类为设备输出（O）。

---

非正式地，*CSR* 空间扮演着一个弱排序的内存映射 *I/O* 区域，正如本手册第二卷中内存排序 *PMA* 节所定义的那样。因此，*CSR* 访问的次序与所有其它访问的次序都受到相同机制的约束，该机制将内存映射 *I/O* 访问的次序约束到这样的区域内。

这些 *CSR* 次序约束用于支持那些对设备或其它硬件线程可见、可被影响的 *CSR* 访问中，主内存和内存映射 *I/O* 的有序访问。例子包括 `time`、`cycle` 和 `mcycle` *CSR*，以及反映挂起中断的 *CSR*，如 `mip` 和 `sip`。注意，这类 *CSR* 的隐式读取（例如，由于 `mip` 的改变而采取的中断）也会被作为设备输入而排序。

大多数 *CSR*（包括，例如，`fcsr`）对其它硬件线程是不可见的；它们的访问可以按关于 *FENCE* 指令的全局内存次序自由地重新排序，而不违反本规范。

硬件平台可以把对特定 CSR 的访问定义为强排序的，就像本手册的第二卷中内存排序 PMA 节里定义的那样。对强排序 CSR 的访问相对于对弱排序 CSR 的访问和内存映射 I/O 区域的访问，有更强的次序约束。

---

按全局内存次序进行 *CSR* 访问的重新排序的规则应该大概被移动至第 十七章，关于 *RVWMO* 内存一致性模型。

# 第十二章 “Zicntr”和“Zihpm”计数器

RISC-V ISA 提供了一组至多 32 个 64 位性能计数器和计时器，它们可以通过非特权 XLEN 只读 CSR 寄存器 `0xC00`–`0xC1F` （当 XLEN = 32 时，高 32 位通过 CSR 寄存器 `0xC80`–`0xC9F`）来访问。这些计数器被划分到“Zicntr”和“Zihpm”扩展。

## 12.1 用于基础计数器和计时器的“Zicntr”标准扩展

Zicntr 标准扩展包括这些计数器的前三个（CYCLE、TIME、和 INSTRET），它们具有专门的功能（分别是周期计数、实时时钟、和指令引退）。Zicntr 扩展依赖于 Zicsr 扩展。

---

我们建议在实现中提供这些基本计数器，因为它们对于基本性能分析、适应和动态优化、以及允许应用处理实时流是必需的。单独的 *Zihpm* 扩展中的其它计数器可以帮助诊断性能问题，应当允许用户级应用代码以较低开销访问这些计数器。

某些执行环境可能禁止访问计数器，例如，为了阻止定时侧信道攻击。

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

对于 XLEN≥64 的基础 ISA，CSR 指令可以直接访问所有的 64 位 CSR。特别地，RDCYCLE、RDTIME 和 RDINSTRET 伪指令读取所有的 64 位 `cycle`、`time` 和 `instret` 计数器。

---

计数器伪指令被映射到只读的 `csrrs rd, counter, x0` 典型形式，但是其它的只读 *CSR* 指令形式（基于 *CSRRC/CSRRSI/CSRRCI*）也是读取这些 *CSR* 的合法方式。

对于 XLEN = 32 的基础 ISA，Zicntr 扩展使这三个 64 位只读计数器可以被以 32 位片段的形式访问。RDCYCLE、RDTIME 和 RDINSTRET 伪指令提供低 32 位，而 RDCYCLEH、RDTIMEH 和 RDINSTRETH 伪指令提供对应计数器的高 32 位。

---

我们需要计数器是 *64* 位宽的——即使是在 *XLEN = 32* 的时候。因为，否则的话，软件将很难决定值是否溢出。对于一个低端实现，各计数器的高 *32* 位可以使用软件计数器来实现，通过低 *32* 位的溢出触发陷入处理器进行增长。下面给出的样例代码显示了如何使用独立的 *32* 位宽伪指令安全地读取所有 *64* 位宽的值。

RDCYCLE 伪指令读取 `cycle` CSR 的低 XLEN 位，该 CSR 持有时钟周期数的计数，由从过去任意启动时间运行硬件线程的处理器核执行。RDCYCLEH 仅在 XLEN = 32 时存在，它读取相同的 cycle 计数器的位 63 - 32。实际中，底层 64 位计数器将永远不会溢出。cycle 计数器的推进率将依赖于实现和操作环境。执行环境应当提供一个决定当前 cycle 计数器增加的（周期/秒）率的方法。

---

*RDCYCLE* 试图返回处理器核（而不是硬件线程）的执行周期数。在给定某些实现选择（例如，*AMD Bulldozer*）时，很难精确定义什么是"核心"。给定实现（包括软件模拟）的范围时，精确定义什么是"时钟周期"也是困难的，但是目的在于，*RDCYCLE* 和其它性能计数器一起被用于性能监视。特别地，在有硬件线程/核心的地方，人们会希望有周期计数/已引退指令来测量硬件线程的 *CPI*。

完全不必要将核心暴露给软件，而且实现者可能选择让一个物理核心上的多个硬件线程假装运行在一个硬件线程/核心上的分离的多个核心上，并为各个硬件线程提供独立的周期计数器。这在内部硬件线程的实时交互不存在或者极少的简单桶式处理器中（例如，*CDC 6000* 外围处理器）可能是有道理的。

在有多于一个的硬件线程/核心和动态多线程的地方，通常不可能分离每个硬件线程的周期（尤其是有 *SMT* 时）。或许可能定义一个独立的性能计数器，它试图捕捉一个特定的正在运行的硬件线程的周期数，但是这个定义将不得不非常模糊以覆盖所有可能的线程实现。例如，我们应当只计数任意被发出执行这个硬件线程的指令的周期？和/或任何失效指令的周期？或是包含了这个硬件线程虽然正在占用机器资源、但由于其它硬件线程转入执行而暂停，导致不能执行的周期？可能，需要"以上所有"才能获得可理解的性能统计数据。定义每个硬件线程周期计数的这种复杂性，以及当调整多线程代码时，在任何情况中对每个核心的周期总数计数的需求，导致了每个核心的周期计数器的标准化，这也恰好适用于常见的单硬件线程/核心的情况。

将在"睡眠"期间发生的事情标准化是不实际的，因为"睡眠"的含义不是跨执行环境标准化的，但是如果代码整体被暂停（在深度睡眠中完全门控时钟或断电），那么时钟周期不会执行，且周期计数每次也将不会按规格增加。这里有许多细节，例如，在处理器从断电事件中被唤醒之后，所需要的用于重置处理器的时钟周期是否应当被计数，而这些都被认为是特定于执行环境的细节。

即使没有作用于全平台的精确定义，仍然有对于大多数平台都有用的版本，并且此处有一个不精确的、常用的、"通常是正确的"标准总比没有标准要更好。*RDCYCLE* 的意图主要是性能监视/调整，而规范在编写时考虑了此目标。

RDTIME 伪指令读取 `time` CSR 的低 XLEN 位，其统计了从过去任意时间开始的已经经过的墙上挂钟的真实时间。RDTIMEH 仅在 XLEN = 32 时存在，它读取相同真实时间计数器的位 63 - 32。在实际中，底层 64 位计数器通过实时时钟的每次滴答来增加一，并且，对于实际的实时时钟频率，应当永远不会溢出。执行环境应当提供一种决定计数器滴答（秒/滴答）周期的方法。该周期应当在一个小的误差范围内是恒定的。环境应当提供一种决定时钟精度的方法（即，名义上的和实际的实时时钟周期之间的最大相关误差）。

---

在一些简单的平台上，周期计数可能代表了 *RDTIME* 的一个有效的实现，在这种情况中，*RDTIME* 和 *RDCYCLE* 可能返回相同的结果。

由于实施平台的广泛的多样性，很难提供对时钟周期的严格的强制规定。最大误差范围应当根据平台的需求来设置。

所有硬件线程的实时时钟必须被同步到实时时钟的一个滴答以内。

---

与其它的架构强制规定一样，只要出现硬件线程“好像”被同步到实时时钟的一个滴答以内就足够了，即，软件无法观察到两个硬件线程上的实时时钟值之间存在更大的差异。

RDINSTRET 伪指令读取 `instret` CSR 的低 XLEN 位，其统计本硬件线程从过去某些任意起始点开始的已失效指令的数目。RDINSTRETH 仅在 XLEN = 32 时存在，它读取相同指令计数器的位 63 - 32。在实际中，底层 64 位计数器应当永远不会溢出。

下面的代码序列将把一个有效的 64 位计数器的值读进 `x3:x2`，即使计数器在读取它的上半部分和下半部分之间，溢出了它的下半部分。

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

图 12.1: 当 XLEN = 32 时，读取 64 位周期计数器的样例代码。

## 12.2 用于硬件性能计数器的“Zihpm”标准扩展

Zihpm 扩展包括至多 29 个额外的非特权 64 位硬件性能计数器，`hpmcounter3` - `hpmcounter31`。当 XLEN = 32 时，这些性能计数器的高 32 位可以通过额外的 CSR `hpmcounter3h` - `hpmcounter31h` 进行访问。Zihpm 扩展依赖于 Zicsr 扩展。

在某些应用中，能够在同时立即读取多个计数器是很重要的。当运行在一个多任务环境下时，用户线程在尝试读取计数器的同时可能遭遇一次上下文的切换。对于用户线程，一个解决方案是，事先读取真实时间计数器，然后之后读取其它计数器来决定在这个序列中是否发生了上下文切换，如果是发生切换的情形，可以令读取失效。我们考虑添加输出锁存器来允许用户线程自动对计数器的值进行快照，但是这将增加用户上下文的尺寸，尤其是对于有更多计数器集的实现来说。

这些额外计数器的实现数量和宽度，以及它们所统计事件的集合，是平台相关的。访问一个未实现的或者错误配置的计数器可能引发一个非法指令异常，或者可能返回一个常量值。

执行环境应当提供一种决定计数器实现的数目和宽度的方法，以及一个配置各计数器所统计事件的接口。

对于实现 *RISC-V* 特权执行环境的平台，特权架构手册描述了通过较低特权模式来控制访问这些计数器、和把事件设置为可被计数的特权 *CSR*。

备用的执行环境（例如，仅用户级的软件性能模型）可以提供替代机制来配置被性能计数器计数的事件。

对于统计 *ISA* 级别的度量标准（例如浮点指令执行的数量）和可能的少量常见微架构度量标准（例如“*L1* 指令缓存缺失”）来说，事件设置的最终标准化将是有用的。

用于单精度浮点的“F”标准扩展（2.2 版本）

# 第十三章　用于单精度浮点的“F”标准扩展（2.2 版本）

这章描述了用于单精度浮点的标准指令集扩展（其被命名为“F”），并添加了兼容 IEEE 754-2008 算数标准 [8] 的单精度浮点运算指令。F 扩展依靠“Zicsr”扩展来访问控制和状态寄存器。

## 13.1　F 寄存器状态

F 扩展添加了 32 个浮点寄存器，`f0-f31`，它们每个都是 32 位宽，并添加了一个浮点控制和状态寄存器 `fcsr`，它包含了浮点单元的操作模式和异常状态。这个额外的状态被显示在表 13.1中。我们使用术语 FLEN 来描述 RISC-V ISA 中的浮点寄存器的宽度，而对于 F 单精度浮点扩展，有 FLEN = 32。大多数浮点指令在浮点寄存器文件中的值上进行操作。浮点加载和存储指令在寄存器和内存之间传递浮点值。也提供了把值传入和传出整数寄存器文件的指令。

---

为了简化软件寄存器分配和调用约定，并减少用户状态总数，我们考虑过为整数值和浮点值使用统一的寄存器文件。然而，分离的组织增加了在给定指令宽度时可访问的寄存器的总数，简化了为宽超标量问题进行足够 *regfile* 端口的提供，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。编译器对分离寄存器文件架构的支持和调用约定是很好理解的，而且在浮点寄存器状态上使用脏位可以减少上下文切换的开销。

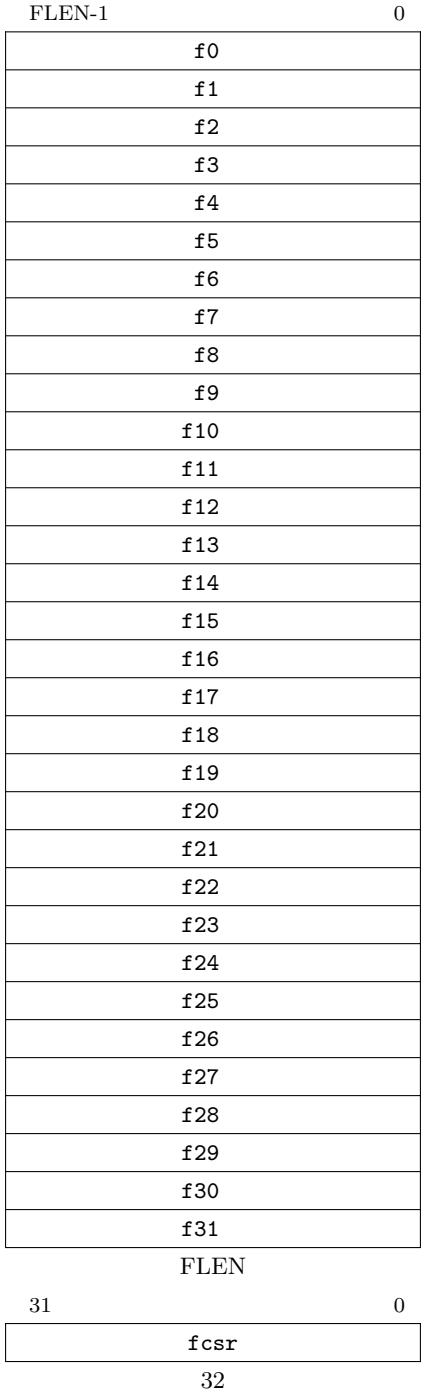



图 13.1: RISC-V 标准 F 扩展单精度浮点状态。

## 13.2 浮点控制和状态寄存器

浮点控制和状态寄存器，fcsr，是一个 RISC-V 控制和状态寄存器（CSR）。它是一个 32 位的读/写寄存器，为浮点算数操作选择动态的舍入模式，并持有累积的异常标志，如图 13.2中显示的那样。

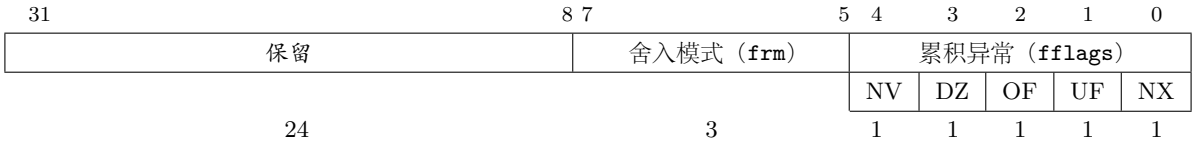



图 13.2: 浮点控制和状态寄存器。

fcsr 寄存器可以使用 FRCSR 和 FSCSR 指令来读取和写入，它们是汇编器伪指令，构建在底层 CSR 访问指令上。FRCSR 通过把 fcsr 复制进整数寄存器 rd 来读取 fcsr。FSCSR 通过把 fcsr 中的原始值复制进整数寄存器 *rd*，然后把从整数寄存器 *rs1* 获得的新值写入 fcsr，来交换 fcsr 中的值。

fcsr 中的域可以通过不同的 CSR 地址来独立地访问，并且为这些访问定义了独立的汇编器伪指令。FRRM 指令读取舍入模式域 frm，并把它复制进整数寄存器 rd 的三个最低有效位，并把所有其它位填零。FSRM 通过把 frm 域中的值复制进整数寄存器 *rd*，然后把从整数寄存器 *rs1* 的三个最低有效位中获得的新值写入 frm，来交换 frm 中的值。对于加速异常标志域 fflags 也类似地定义了 FRFLAGS 和 FSFLAGS。

fcsr 的位 31 - 8 被保留用于其它标准扩展。如果这些扩展尚未存在，那么实现应当忽略对这些位的写入，并在读取的时候提供零值。标准软件应当保留这些位的内容。

浮点操作或者使用编码在指令中的静态舍入模式，或者使用 frm 中持有的动态舍入模式。表 13.1中显示了舍入模式的编码。指令的 *rm* 域中的 111 值选择了 frm 中持有的动态舍入模式。当使用保留的舍入模式执行时，依赖于舍入模式的浮点指令的行为是保留的，包括静态保留舍入模式（101 - 110）和动态保留舍入模式（101 - 111）。某些指令，包括加宽的转换，具有 *rm* 域，但是在数学上并不会被舍入模式影响；软件应把它们的 *rm* 域设置为 RNE（000），但是实现必须把 *rm* 域如常对待（特别地，在涉及解码合法 vs 保留编码时）。

---

*C99 语言标准有效地约束了动态舍入模式寄存器的提供。在典型的实现中，写动态舍入模式 CSR 状态将把管道序列化。静态舍入模式被用于实现专门的算数操作，它们经常不得不在不同的舍入模式之间频繁切换。*

*F 规范的已批准版本强制规定，当一条指令以保留的动态舍入模式执行时，会引发一个非法指令异常。这已经被弱化位保留的，与静态舍入模式指令的行为相匹配。当遇到保留的编码*

| 舍入模式 | 助记符 | 含义 |
|---|---|---|
| 000 | RNE | 就近舍入，关联到偶数 |
| 001 | RTZ | 向零舍入 |
| 010 | RDN | 向下舍入（向 $-\infty$） |
| 011 | RUP | 向上舍入（向 $+\infty$） |
| 100 | RMM | 就近舍入，关联到最大幅度 |
| 101 | | 保留供未来使用。 |
| 110 | | 保留供未来使用。 |
| 111 | DYN | 在指令的 *rm* 域中，选择动态舍入模式；<br>在舍入模式寄存器中，保留。 |

表 13.1: 舍入模式编码。

时，引发一个非法指令异常仍然是有效的行为，所以与已批准的规范兼容的实现也是与弱化的规范相兼容的。

累积异常标志表明了，自从软件上一次重置该域以来，在任何浮点算数指令上已经发生的异常情况，如表 13.2中显示的那样。基础 RISC-V ISA 不支持在浮点异常标志的设置时生成陷入。

| 标志助记符 | 标志含义 |
|---|---|
| NV | 无效的操作 |
| DZ | 除数为零 |
| OF | 溢出 |
| UF | 向下溢出 |
| NX | 不精确的 |

表 13.2: 累积异常标志编码。

---

正如标准所允许的那样，我们不支持在 *F* 扩展中的浮点异常上的陷入，但是需要显式地检查软件中的标志。我们考虑过添加直接通过浮点累积异常标志的内容来控制的分支，但是最终选择了忽略这些指令以保持 *ISA* 的简单。

## 13.3 NaN 的生成和传播

除非另有说明，如果浮点操作的结果是 NaN，那么它是规范的 NaN。规范的 NaN 具有一个正号，并且除了 MSB（或者说，沉默位）以外的所有有效位都被清除。对于单精度浮点，这对应于式样 `0x7fc00000`。

---

我们考虑过传播 *NaN* 的有效载荷，就像标准推荐的那样，但是这个决定将增加硬件开销。并且，由于这个特征在标准中是可选的，它不能被用于可移植的代码。

实现者可以自由地提供一个 *NaN* 有效载荷传播策略，作为被非标准操作模式启用的非标准扩展。然而，上面描述的规范的 *NaN* 策略必须总是被支持的，并且应当成为默认模式

---

在异常情况中，我们需要实现来返回标准所要求的默认值，就用户级软件而言无需进一步干预（不像 *Alpha ISA* 浮点陷入屏障那样）。我们相信异常情况的全硬件处理将变得更加常见，并且因此希望避免让用户级 *ISA* 复杂化，以优化其它的方法。实现可以总是陷入到机器模式软件处理程序来提供异常的默认值。

## 13.4 亚正常算法

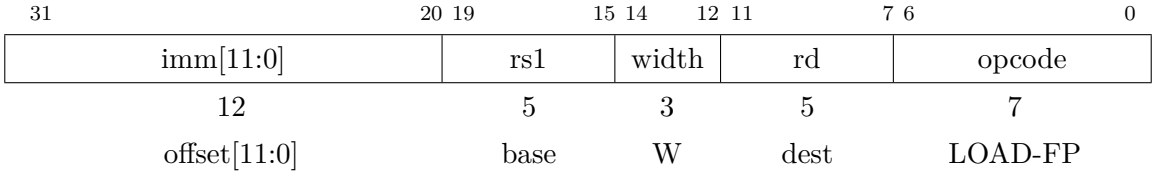
关于亚正常数的操作按照 IEEE754-2008 标准处理。在 IEEE 标准的说法中，极小是在舍入之后检测的。

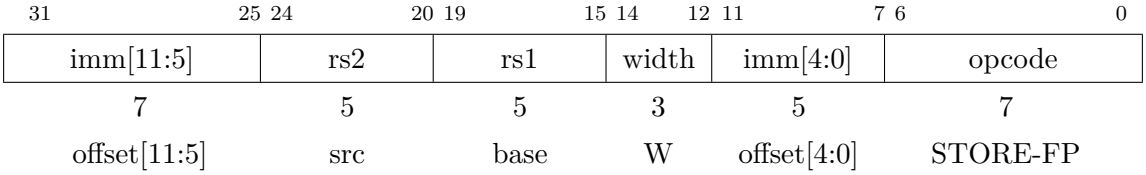
---

在舍入之后检测极小导致了更少的貌似的向下溢出信号。

## 13.5 单精度加载和存储指令

浮点加载和存储使用相同的“基址 + 偏移量”编址模式，就像整数基础 ISA 那样，一个寄存器 *rs1* 中的基地址与一个 12 位的有符号字节偏移量。FLW 指令从内存加载一个单精度浮点值，并把它放入浮点寄存器 *rd*。FSW 把浮点寄存器 *rs2* 中的一个单精度值存储到内存。




只有在有效地址自然对齐的时候，才保证 FLW 和 FSW 的原子性执行

FLW 和 FSW 不修改正在被传递的位；特别地，非规范的 NaN 的有效载荷被保留。

正如第 2.6节描述的那样，EEI 定义了未对齐的浮点加载和存储是被隐式地处理，还是引发一个包含的或致命的陷入。

## 13.6 单精度浮点运算指令

带有一个或两个源操作数的浮点算数指令使用带有 OP-FP 主操作码的 R 类型格式。FADD.S 和 FMUL.S 分别在 *rs1* 和 *rs2* 之间执行单精度浮点加法和乘法。FSUB.S 执行从 *rs1* 中减去 *rs2* 的单精度浮点减法。FDIV.S 执行 *rs1* 除以 *rs2* 的单精度浮点除法。FSQRT.S 计算 *rs1* 的平方根。在每种情况中，结果都被写入 *rd*。

2 位浮点格式域 *fmt* 按照表 13.3中显示的那样编码。对于 F 扩展中的所有指令，它都被设置为 *S*（00）。

| *fmt* 域 | Mnemonic | Meaning |
|---|---|---|
| 00 | S | 32 位单精度 |
| 01 | D | 64 位双精度 |
| 10 | H | 16 位半精度 |
| 11 | Q | 128 位四精度 |

表 13.3: 格式域编码。

所有执行舍入的浮点操作都可以使用 *rm* 域来选择舍入模式，*rm* 域的编码显示在表 13.1中。

浮点最小数和最大数指令 FMIN.S 和 FMAX.S 分别把 *rs1* 和 *rs2* 中的较小者或较大者写到 *rd*。仅对于这些指令的目的而言，值 −0.0 被认为小于值 +0.0。如果两个输入都是 NaN，结果是规范的 NaN。如果只有一个操作数是 NaN，结果是那个非 NaN 的操作数。发信号的 NaN 输入会设置无效操作异常标志，即使当结果不是 NaN 时也是如此。

---

*注意，在 F 扩展的 2.2 版本中，FMIN.S 和 FMAX.S 指令被修正为实现所提出的 IEEE 754-201x 的 mininumNumber 和 maximumNumber 操作，而不是 IEEE 754-2008 的 minNum 和 maxNum 操作。这些操作的区别在于它们对发信号的 NaN 的处理。*

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

浮点融合乘加指令需要一个新的标志指令格式。R4 类型指令指定三个源寄存器（*rs1*、*rs2* 和 *rs3*）和一个目的寄存器（*rd*）。这个格式只被浮点融合乘加指令使用。

FMADD.S 将 *rs1* 和 *rs2* 中的值相乘，加上 *rs3* 中的值，并把最终结果写到 *rd*。FMADD.S 计算 *(rs1×rs2)+rs3*。

FMSUB.S 将 *rs1* 和 *rs2* 中的值相乘，减去 *rs3* 中的值，并把最终结果写到 *rd*。FMSUB.S 计算 *(rs1×rs2)-rs3*。

FNMSUB.S 将 *rs1* 和 *rs2* 中的值相乘，取乘积的相反数，加上 *rs3* 中的值，并把最终结果写到 *rd*。FNMSUB.S 计算*-(rs1×rs2)+rs3*。

FNMADD.S 将 *rs1* 和 *rs2* 中的值相乘，取乘积的相反数，减去 *rs2* 中的值，并把最终结果写到 *rd*。FNMADD.S 计算*-(rs1×rs2)-rs3*。

---

*FNMSUB 和 FNMADD 指令的命名是反直觉的，是由于 MIPS-IV 中对应指令的命名。MIPS 指令被定义为对总和的取负，而不像 RISC-V 的指令做的那样只对乘积取负，所以当时的命名策略更合理。这两个定义对于有符号的零的结果是有区别的。RISC-V 的定义符合 x86 和 ARM 融合乘加指令的行为，但与 x86 和 ARM 相比，RISC-V FNMSUB 和 FNMADD 指令的名字被不幸地交换了。*

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

---

*融合乘加（FMA）指令会消耗 32 位指令编码空间的一大部分。考虑过某些替代方案来限制 FMA 只使用动态舍入模式，但是静态舍入模式在利用了缺少乘积舍入的代码中是有用的。另一个备选方案将使用 rd 来提供 rs3，但是这在一些常见的序列中将需要额外的移动指令。当前的设计仍然使 32 位编码空间的大部分保持开放，同时避免让 FMA 是非正交的。*

当被乘数是 $\infty$ 和零时，融合乘加指令必须设置无效操作异常标志，即使加数是静默的 NaN 时也需如此。

---

*IEEE 754-2008 标准允许（但是不必须）为$\infty \times 0 + qNaN$操作产生无效异常。*

## 13.7 单精度浮点转换和移动指令

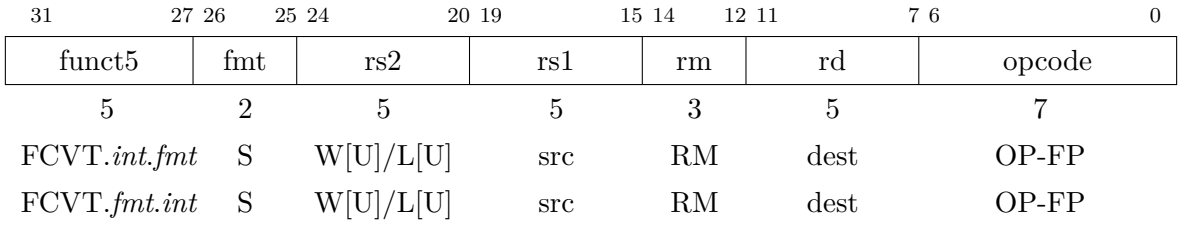
浮点到整数转换指令和整数到浮点转换指令被编码在 OP-FP 主操作码空间中。FCVT.W.S 或 FCVT.L.S 将一个浮点寄存器 *rs1* 中的浮点数分别转化为一个有符号的 32 位或 64 位整数，并将其放入整数寄存器 *rd* 中。FCVT.S.W 或 FCVT.S.L 分别把整数寄存器 *rs1* 中的一个 32 位或 64 位有符号整数转化为一个浮点数，并把它放入浮点寄存器 *rd* 中。FCVT.WU.S、FCVT.LU.S、FCVT.S.WU 和 FCVT.S.LU 的变体转化或转换为无符号整数值。对于大于 32 的 XLEN，FCVT.W[U].S 把 32 位结果符号扩展到目的寄存器的宽度。FCVT.L[U].S 和 FCVT.S.L[U] 是 RV64 独有的指令。如果舍入的结果不能以目的格式表示，它将被裁剪为最接近的值，并且设置无效标志。表 13.4给出了 FCVT.*int*.S 的有效输入的范围和无效输入的行为。

| | FCVT.W.S | FCVT.WU.S | FCVT.L.S | FCVT.LU.S |
|---|---|---|---|---|
| 最小有效输入（舍入后） | $-2^{31}$ | 0 | $-2^{63}$ | 0 |
| 最大有效输入（舍入后） | $2^{31}-1$ | $2^{32}-1$ | $2^{63}-1$ | $2^{64}-1$ |
| 对于超出范围的负数输入的输出 | $-2^{31}$ | 0 | $-2^{63}$ | 0 |
| 对于 $-\infty$ 的输出 | $-2^{31}$ | 0 | $-2^{63}$ | 0 |
| 对于超出范围的正数输入的输出 | $2^{31}-1$ | $2^{32}-1$ | $2^{63}-1$ | $2^{64}-1$ |
| 对于 $+\infty$ 或 NaN 的输出 | $2^{31}-1$ | $2^{32}-1$ | $2^{63}-1$ | $2^{64}-1$ |

表 13.4: 浮点到整数转换的域和对于无效输入的行为。

所有的浮点到整数转换指令和整数到浮点转换指令都根据 *rm* 域进行舍入。浮点寄存器可以使用 FCVT.S.W *rd*, `x0` 被初始化为浮点正零，它将永远不会设置任何异常标志。

如果舍入结果与操作数的值不同，并且没有设置无效异常标志，那么所有的浮点转换指令都会设置不精确异常标志。



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

浮点到浮点的符号注入指令，FSGNJ.S、FSGNJN.S 和 FSGNJX.S，产生的结果是取 *rs1* 的除了符号位的所有位。对于 FSGNJ，结果的符号位是 *rs2* 的符号位；对于 FSGNJN，结果的符号位是 *rs2* 的符号位取反；而对于 FSGNJX，该符号位是 *rs1* 和 *rs2* 的符号位的 XOR 结果。符号注入指令既不设置浮点异常标志，它们也不会将 NaN 规范化。注意，FSGNJ.S *rx, ry, ry* 把 *ry* 移动到 *rx*（汇编器伪指令 FMV.S *rx, ry*）；FSGNJN.S *rx, ry, ry* 把 *ry* 的相反数移动到 *rx*（汇编器伪指令 FNEG.S *rx, ry*）；而 FSGNJX.S *rx, ry, ry* 把 *ry* 的绝对值移动到 *rx*（汇编器伪指令 FABS.S *rx, ry*）。

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

---

符号注入指令提供了浮点 *MV*、*ABS* 和 *NEG*，也支持了少量其它的操作，包括 *IEEE copySign* 操作和超越数学函数库中的符号操作。尽管 *MV*、*ABS* 和 *NEG* 只需要一个寄存器操作数，而 *FSGNJ* 指令需要两个，但大多数微架构将不太可能添加优化，来从读取这些相对不频繁的指令的寄存器数目的减少中受益。甚至在这种情况中，微架构也可以为 *FSGNJ* 指令做简单地检测，当两个源寄存器是相同的时，只读取一份拷贝。

提供了在浮点寄存器和整数寄存器之间移动位式样的指令。FMV.X.W 把浮点寄存器 *rs1* 中的以 IEEE 754-2008 编码表示的单精度值移动到整数寄存器 *rd* 的低 32 位。在转移中这些位不会被修改，并且特别地，非规范的 NaN 的有效载荷也被保留。对于 RV64，目的寄存器的高 32 位被填充为浮点数的符号位的拷贝。

FMV.W.X 把整数寄存器 *rs1* 的低 32 位中的以 IEEE 754-2008 标准编码的单精度值移动到浮点寄存器 *rd*。在转移中这些位不会被修改，并且特别地，非规范的 NaN 的有效载荷被保留。

---

*FMV.W.X* 和 *FMV.X.W* 指令之前被称作 *FMV.S.X* 和 *FMV.X.S*。*W* 的使用更符合它们作为单纯移动 *32* 位而不对其进行解释的指令的语义。这在定义了 *NaN* 装箱之后变得更加清晰。为了避免干扰现有的代码，*W* 版本和 *S* 版本都将被工具支持。

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

基础浮点 *ISA* 被如此定义，是为了允许实现在寄存器中采用浮点格式的内部重新编码，以简化对亚正常值的处理，并可能减少功能单元的延迟。为此，*F* 扩展避免在浮点寄存器中，通过定义直接读写整数寄存器文件的转化和比较操作来表示整数值。这也去除了许多常见的需要在整数寄存器和浮点寄存器之间显式移动的情况，为常见的混合格式代码序列减少了指令计数和关键路径。

## 13.8 单精度浮点比较指令

浮点比较指令(FEQ.S、FLT.S、FLE.S)在浮点寄存器之间执行特定的比较(*rs1* = *rs2*, *rs1* < *rs2*, *rs1* ≤ *rs2*)，并且如果条件满足，向整数寄存器 *rd* 写入 1，否则写入 0。

FLT.S 和 FLE.S 执行 IEEE 754-2008 标准所提及的信号比较：即，如果某个输入是 NaN，它们设置无效操作异常标志。FEQ.S 实施静默比较：它只在某个输入是发信号的 NaN 时设置无效操作异常标志。对于所有这三个指令，如果有操作数是 NaN，那么结果就是 0。

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

*F* 扩展提供了一个 ≤ 比较，而基础 *ISA* 提供了一个 ≥ 分支比较。因为 ≤ 可以从 ≥ 中合成，反之亦然，所以对于这种不一致性没有性能上的影响，但是在 *ISA* 中，它仍然是一种不幸的不协调。

## 13.9 单精度浮点分类指令

FCLASS.S 指令检测浮点寄存器 *rs1* 中的值，并向整数寄存器 *rd* 写入一个 10 位的掩码，它表示该浮点数的类别。掩码的格式被描述在表 13.5中。如果某个属性为真，那么将设置 *rd* 中的对应位并清除其它位。*rd* 中的所有其它位被清除。注意在 *rd* 中将恰好只有一位会被设置。FCLASS.S 不设置浮点异常标志。

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

| *rd* bit | Meaning |
|---|---|
| 0 | *rs1* 是 $-\infty$. |
| 1 | *rs1* 是一个负的正常的数 |
| 2 | *rs1* 是一个负的亚正常的数 |
| 3 | *rs1* 是 $-0$. |
| 4 | *rs1* 是 $+0$. |
| 5 | *rs1* 是一个负的亚正常的数 |
| 6 | *rs1* 是一个负的正常的数 |
| 7 | *rs1* 是 $+\infty$. |
| 8 | *rs1* 是发信号的 NaN |
| 9 | *rs1* 是沉默的 NaN |

表 13.5: FCLASS 指令的结果的格式。

# 第十四章　用于双精度浮点的“D”标准扩展（2.2 版本）

这章描述了标准双精度浮点指令集扩展，该扩展被命名为“D”，并添加了兼容 IEEE 754-2008 算数标准的双精度浮点运算指令。D 扩展依赖于基础单精度指令子集 F。

## 14.1　D 寄存器状态

D 扩展把 32 位浮点寄存器 `f0` - `f31` 拓宽到 64 位（在图 13.1中 FLEN = 64）。`f` 寄存器现在既可以持有 32 位浮点值，也可以持有 64 位浮点值，正如下面在 14.2节中描述的那样。

---

*根据 F 扩展、D 扩展和 Q 扩展被支持的情况，FLEN 可以是 32、64 或者 128。可以至多支持四个不同的浮点精度，包括 H、F、D 和 Q。*

## 14.2　较窄值的 NaN 装箱

当支持多个浮点精度时，在一个 FLEN 位 NaN 值的低 $n$ 位中表示较窄的 $n$ 位类型（$n$ <FLEN）的有效值，这个过程术语叫做 NaN 装箱。一个有效的 NaN 装箱的值的高位必须全是 1。因此，当被视为任意更宽的 m 位值时（$n < m \leq$ FLEN），有效的 NaN 装箱的 $n$ 位值表现为负的静默 NaN（qNaN）。任何把较窄的结果写到一个 `f` 寄存器的操作都必须把最高的 FLEN$-n$ 位全写成 1，以产生一个合法的 NaN 装箱的值。

---

*软件可能不知道存储在一个浮点寄存器中的数据的当前类型，但是必须能够保存和恢复寄存器的值，因此不得不定义使用较宽的操作来转移较窄的值的结果。一个常见的情况是用于由调用者保存的寄存器，但是对于包括 varargs、用户级线程库、虚拟机迁移、和调试在内的特征，标准约定也是值得的。*

浮点 $n$ 位转移操作把以 IEEE 标准格式保持的外部值移进和移出 `f` 寄存器，并包含浮点加载和存储（FL$n$/FS$n$）和浮点移动指令（FMV.$n$.X/FMV.X.$n$）。把一个较窄的 $n$ 位（$n <$ FLEN）转移进 f 寄存器将创造一个有效的 NaN 装箱的值。把一个较窄的 $n$ 位转移出浮点寄存器时，将转移该寄存器的低 $n$ 位，而忽略高 FLEN $-$ $n$ 位。

除了在之前段落中描述的转移操作，所有其它的关于操作较窄 $n$ 位的浮点操作（$n <$ FLEN）都将检查输入操作数是否被正确地 NaN 装箱，或者说，所有的 FLEN$-n$ 位是否都是 1。如果的确如此，输入的最低 $n$ 个有效位被作为输入值使用，否则输入值被视为一个 $n$ 位的规范 NaN。

---

这个文档的较早的版本没有定义把较窄或较宽操作数的结果送进操作的行为，除非要求较宽的保存和恢复将保留较窄操作数的值。新的定义移除了这个与实现有关的行为，但仍然采纳了浮点单元的非重新编码的实现和重新编码的实现。如果没有正确地使用值，新的定义也帮助抓取由传播 *NaN* 引起的软件错误。

非重新编码的实现在每个浮点指令的输入和输出上把操作数按 *IEEE* 标准格式解包和打包。对一个非重新编码的实现的 *NaN* 装箱开销主要在于，检查较窄操作的高位是否表示了一个合法的 *NaN* 装箱的值，以及把结果的高位全写成 *1*。

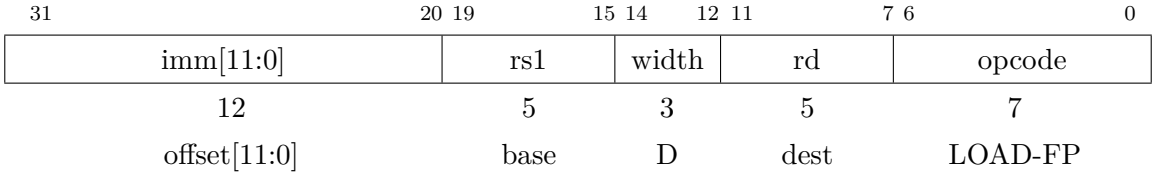
重新编码的实现使用一个更加方便的内部格式来表示浮点值，它添加了一个指数位来允许所有的值的标准化保持。重新编码的实现的开销主要在于，为了追踪内部类型和符号位所需要的额外的标签工作，但是这可以通过在指数域中内部地重新编码 *NaN* 来完成，而不用添加新的状态位。用于把值转移进出重新编码格式的管道需要一些小的改动，但是数据路径和延迟开销是很小的。在任何情况中，对于宽操作数，重新编码的过程都必须处理输入亚正常值的移位，而提取 *NaN* 装箱的值是一个与标准化相似的过程，除了要跳过领头的 *1* 位而不是 *0* 位以外，从而允许共享数据路径的 *muxing*。

## 14.3 双精度加载和存储指令

FLD 指令从内存加载一个双精度浮点值到浮点寄存器 *rd* 中。FSD 把浮点寄存器中的一个双精度值存储到内存中。

---

该双精度值可以是一个 *NaN* 装箱的单精度值。



| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|
| imm[11:5] | rs2 | rs1 | width | imm[4:0] | opcode |
| 7 | 5 | 5 | 3 | 5 | 7 |
| offset[11:5] | src | base | D | offset[4:0] | STORE-FP |

只有当有效的地址被自然对齐，并且 XLEN≥64 时，FLD 和 FSD 才保证原子性执行。

FLD 和 FSD 不修改正在被转移的位；特别地，非规范的 NaN 的有效载荷被保留。

## 14.4 双精度浮点运算指令

双精度浮点运算指令的定义与它们对应的单精度指令的定义类似，但是在双精度操作数上操作，并产生双精度的结果。

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

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

## 14.5 双精度浮点转换和移动指令

浮点到整数转换指令和整数到浮点转换指令被编码在 OP-FP 主操作码空间中。FCVT.W.D 或 FCVT.L.D 把浮点寄存器 *rs1* 中的双精度浮点数分别转化为一个有符号的 32 位或 64 位整数，并将其放入整数寄存器 *rd* 中。FCVT.D.W 或 FCVT.D.L 分别把整数寄存器 *rs1* 中的 32 位或 64 位有符号整数转换为一个双精度浮点数，并将其放入浮点寄存器 *rd* 中。FCVT.WU.D，FCVT.LU.D，

FCVT.D.WU 和 FCVT.D.LU 变体转化为无符号整数值、或从无符号整数值转化。对于 RV64，FCVT.W[U].D 把 32 位结果进行符号扩展。FCVT.L[U].D 和 FCVT.D.L[U] 是 RV64 独有的指令。FCVT.*int*.D 的有效输入范围和无效输入行为与 FCVT.*int*.S 相同。

所有的浮点到整数转换指令和整数到浮点转换指令都根据 *rm* 域进行舍入。注意 FCVT.D.W[U] 总是产生确切的结果，而不会被舍入模式影响。

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

双精度到单精度的转化指令 FCVT.S.D 和单精度到双精度的转化指令 FCVT.D.S 被编码在 OP-FP 主操作码空间中，并且源寄存器和目的寄存器都是浮点寄存器。*rs2* 域编码了源寄存器的数据类型，而 *fmt* 域编码了目的寄存器的数据类型。FCVT.S.D 根据 RM 域进行舍入，FCVT.D.S 将永远不会舍入。

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

浮点到浮点的符号注入指令，FSGNJ.D、FSGNJN.D 和 FSGNJX.D 的定义与对应的单精度符号注入指令的定义类似。

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

只有当 XLEN≥64 时，提供在浮点和整数寄存器之间按位式样移动的指令。FMV.X.D 把浮点寄存器 *rs1* 中的双精度值移动到一个以 IEEE 754-2008 标准编码表示的整数寄存器 *rd* 中。FMV.D.X 从整数寄存器 *rs1* 中把以 IEEE 754-2008 标准编码编码的双精度值移动到浮点寄存器 *rd* 中。

FMV.X.D 和 FMV.D.X 不修改正在被转移的位；特别地，非规范的 NaN 的有效载荷被保留。

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

*RISC-V ISA* 的早期版本有额外的指令来允许 *RV32* 系统在一个 *64* 位浮点寄存器的高位部分和低位部分与一个整数寄存器之间进行转移。然而这将成为仅有的部分寄存器写入指令，并将增加实现在重新编码浮点或寄存器重命名时的复杂性，因为需要一个管道"读-修改-写"序列。如果要遵循这个式样，为 *RV32* 和 *RV64* 增加到处理四精度也将需要额外的指令。*ISA* 被定义为，通过把转化和比较的结果写入合适的寄存器文件，来减少整数到浮点寄存器的显式移动的数目，因此我们希望这些指令的收益能够比其它的 *ISA* 更低。

我们注意到，对于实现了 *64* 位浮点单元（包括融合的乘加支持）和 *64* 位浮点加载与存储的系统来说，从 *32* 位整数移动到 *64* 位整数的数据路径的外围硬件开销较低，而支持 *32* 位宽地址空间和指针的软件 *ABI* 可以被用于避免静态数据的增长和动态内存的拥塞。

## 14.6 双精度浮点比较指令

双精度浮点比较指令的定义与它们对应的单精度指令的定义类似，但是在双精度操作数上操作。

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

## 14.7 双精度浮点分类指令

双精度浮点分类指令，FCLASS.D，其定义与对应的单精度指令的定义类似，但是在双精度操作数上操作。

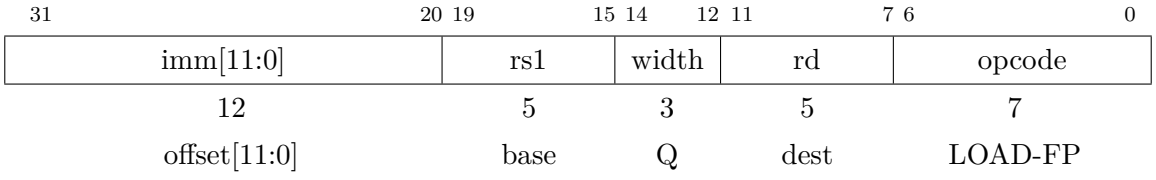
| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|
| funct5 | fmt | rs2 | rs1 | rm | rd | opcode |
| 5 | 2 | 5 | 5 | 3 | 5 | 7 |
| FCLASS | D | 0 | src | 001 | dest | OP-FP |

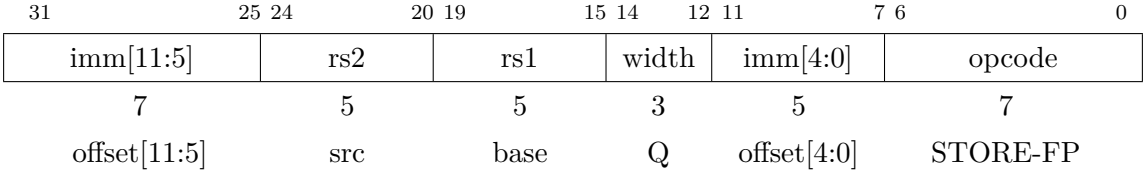
# 第十五章　用于四精度浮点的“Q”标准扩展（2.2 版本）

这章描述了用于 128 位四精度二进制浮点指令的、兼容 IEEE 754-2008 算数标准的 Q 标准扩展。四精度二进制浮点指令集扩展被命名为“Q”；它依赖于双精度浮点扩展 D。浮点寄存器现在被扩展为保持单精度、双精度、或者四精度的浮点值（FLEN = 128）。在 14.2节中描述的 NaN 装箱策略现在被递归地扩展，以允许单精度值被 NaN 装箱在一个双精度值中，而该双精度值自己被 NaN 装箱在一个四精度值中。

## 15.1　四精度加载和存储指令

添加了 LOAD-FP 和 STORE-FP 指令的新的 128 位变体，使用 funct3 宽度域的新值进行编码。




只有有效的地址被自然对齐并且 XLEN = 128 时，才保证 FLQ 和 FSQ 原子性地执行。

FLQ 和 FSQ 不会修改正在被转移的位；特别地，非规范的 NaN 的有效载荷被保留。

## 15.2 四精度运算指令

为大多数指令的格式域添加了一个新的被支持的格式，如表 15.1中显示的那样。

| *fmt* 域 | Mnemonic | Meaning |
|---|---|---|
| 00 | S | 32 位单精度 |
| 01 | D | 64 位双精度 |
| 10 | H | 16 位半精度 |
| 11 | Q | 128 位四精度 |

表 15.1: 格式域编码。

四精度浮点运算指令的定义与他们对应的双精度指令的定义类似，但是在四精度操作数上操作，并产生四精度的结果。

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

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

## 15.3 四精度转换和移动指令

添加了新的浮点到整数转化指令和整数到浮点转化指令。这些指令的定义与双精度到整数转化指令和整数到双精度转化指令的定义类似。FCVT.W.Q 或者 FCVT.L.Q 分别把一个四精度浮点数转化为一个有符号的 32 位或 64 位整数。FCVT.Q.W 或 FCVT.Q.L 分别把一个 32 位或 64 位有符号整数转化为一个四精度浮点整数。FCVT.WU.Q、FCVT.LU.Q、FCVT.Q.WU 和 FCVT.Q.LU 变体转化为无符号整数值、或从无符号整数值转化。FCVT.L[U].Q 和 FCVT.Q.L[U] 是 RV64 独有的指令。注意 FCVT.Q.L[U] 总是产生确切的结果，而不会被舍入模式影响。

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

添加了新的浮点到浮点转化指令。这些指令的定义与双精度浮点到浮点转化指令的定义类似。FCVT.S.Q 把一个四精度浮点数转化为一个单精度浮点数，FCVT.Q.S 与之相反。FCVT.D.Q 把一个四精度浮点数转化为一个双精度浮点数，FCVT.Q.D 与之相反。

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

浮点到浮点的符号注入指令，FSGNJ.Q、FSGNJN.Q 和 FSGNJX.Q 的定义与双精度符号注入指令的定义类似。

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

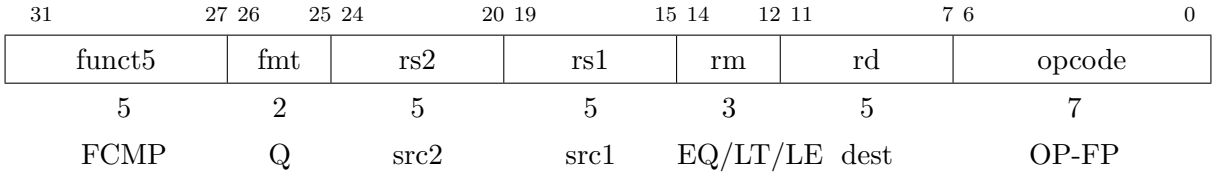
在 RV32 或 RV64 中不提供 FMV.X.Q 和 FMV.Q.X 指令，所以四精度位式样必须通过内存被移动到整数寄存器。

---

*RV128 将在 Q 扩展中支持 FMV.X.Q 和 FMV.Q。*

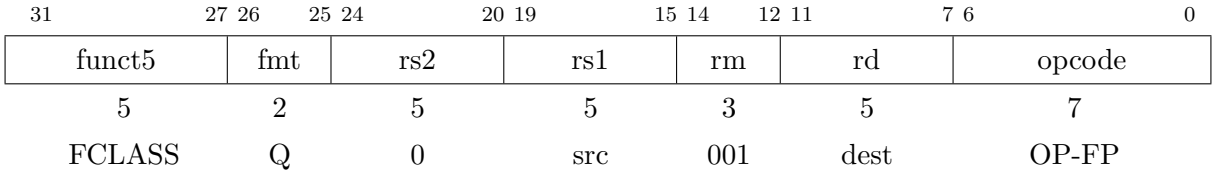
## 15.4 四精度浮点比较指令

四精度浮点比较指令的定义与它们对应的双精度指令的定义类似，但是在四精度操作数上操作。




## 15.5 四精度浮点分类指令

四精度浮点分类指令，FCLASS.Q，其定义与它对应的双精度指令类似，但是在四精度操作数上操作。



# 第十六章　用于半精度浮点的“Zfh”和“Zfhmin”标准扩展（1.0 版本

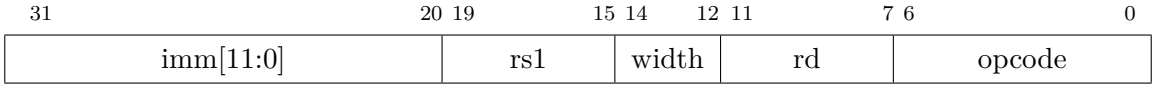
本章描述了用于 16 位半精度二元浮点指令的 Zfh 标准扩展，兼容 IEEE 754-2008 算术标准。Zfh 扩展依赖于单精度浮点扩展，F。在第 14 章描述的 NaN 装箱策略被扩展为，允许在一个单精度值中对一个半精度值 NaN 装箱（当 D 扩展或 Q 扩展存在时，其可以在双精度或四精度值中递归地 NaN 装箱）。

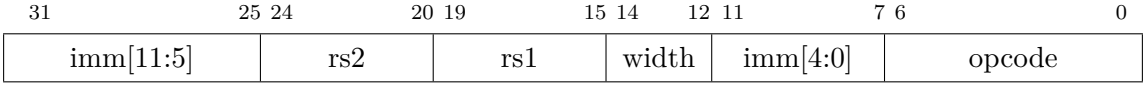
---

这个扩展主要提供了消费半精度操作数并产生半精度结果的指令。然而，在计算半精度数据中使用更高的中间精度也是很常见的。尽管这个扩展提供了足以实现该样式的显式转换指令，未来的扩展也可能用额外的指令进一步加速此类运算，那些额外的指令会隐式地加宽其操作数——例如，半 × 半 + 单 → 单——或者隐式地收窄其结果——例如，半 + 单 → 半。

## 16.1　半精度加载和存储指令

添加了 LOAD-FP 和 STORE-FP 指令的新的 16 位变体，并为 funct3 宽度域编码了新的值。




只有当有效地址被自然对齐时，才保证 FLH 和 FSH 原子地执行。

FLH 和 FSH 不改变正在被传输的位；特别地，保留了非典型 NaN 的有效载荷。FLH 将写入 *rd* 的结果进行 NaN 装箱，而 FSH 忽略 *rs2* 中除低 16 位之外的所有位。

## 16.2 半精度运算指令

向大多数指令的 format 域添加了一个新的支持格式，如表 16.1所示。

| *fmt* 域 | 助记符 | 含义 |
|---|---|---|
| 00 | S | 32 位单精度 |
| 01 | D | 64 位双精度 |
| 10 | H | 16 位半精度 |
| 11 | Q | 128 位四精度 |

表 16.1: 格式域编码。

半精度浮点运算指令的定义类似于它们对应的单精度指令，但是在半精度操作数上操作，并产生半精度结果。

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

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

## 16.3 半精度转换和移动指令

添加了新的浮点到整数转换指令和整数到浮点转换指令。这些指令的定义与单精度到整数转换指令及整数到单精度转换指令类似。FCVT.W.H 或 FCVT.L.H 分别把一个半精度浮点数转换为一

个有符号的 32 位或 64 位整数。FCVT.H.W 或 FCVT.H.L 分别把一个 32 位或 64 位有符号整数转换为一个半精度浮点数。FCVT.WU.H、FCVT.LU.H、FCVT.H.WU 和 FCVT.H.LU 的变体与无符号整数值相互转换。FCVT.L[U].H 和 FCVT.H.L[U] 是只在 RV64 中使用的指令。

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

添加了新的浮点到浮点转换指令。这些指令的定义与双精度浮点到浮点转换指令类似。FCVT.S.H 或 FCVT.H.S 分别把一个半精度浮点数转换为一个单精度浮点数、以及相反的操作。如果存在 D 扩展，FCVT.D.H 或 FCVT.H.D 分别把一个半精度浮点数转换为一个双精度浮点数、以及相反的操作。如果存在 Q 扩展，FCVT.Q.H 或 FCVT.H.Q 分别把一个半精度浮点数转换为一个四精度浮点数、以及相反的操作。

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

浮点到浮点的符号注入指令，FSGNJ.H、FSGNJN.H 和 FSGNJX.H 的定义与单精度符号注入指令类似。

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

提供了在浮点和整数寄存器之间移动位式样的指令。FMV.X.H 把浮点寄存器 *rs1* 中的半精度值按照 IEEE 754-2008 标准编码的一种表示形式移动到整数寄存器 *rd* 中，并把浮点数的符号位复制填充到高 XLEN-16 位。

FMV.H.X 把按照 IEEE 754-2008 标准编码的半精度值从整数寄存器 *rs1* 的低 16 位移动到浮点寄存器 *rd*，并把结果 NaN 装箱。

FMV.X.H 和 FMV.H.X 不改变正在被传输的位；特别地，非典型 NaN 的有效载荷被保留。

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

## 16.4 半精度浮点比较指令

半精度浮点比较指令的定义与其对应的单精度版本类似，但是在半精度操作数上操作。

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

## 16.5 半精度浮点分类指令

半精度浮点分类指令，FCLASS.H，其定义与其对应的单精度版本类似，但是在半精度操作数上操作。

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

## 16.6 用于最小半精度浮点支持的“Zfhmin”标准扩展

本节描述了 Zfhmin 标准扩展，它为 16 位半精度二进制浮点指令提供了最低限度的支持。Zfhmin 扩展是 Zfh 扩展的一个子集，仅由数据传输与转换指令组成。像 Zfh 一样，Zfhmin 扩展依赖

于单精度浮点扩展，F。预想的 Zfhmin 软件主要使用半精度格式存储，而以更高精度执行大多数运算。

Zfhmin 扩展包括下列来自 Zfh 的指令：FLH、FSH、FMV.X.H、FMV.H.X、FCVT.S.H 和 FCVT.H.S。如果 D 扩展存在，也包括 FCVT.D.H 和 FCVT.H.D 指令。如果 Q 扩展存在，还额外包括了 FCVT.Q.H 和 FCVT.H.Q 指令。

---

*Zfhmin 不包括 FSGNJ.H 指令，因为将半精度值在浮点寄存器之间移动，使用 FSGNJ.S 指令代替就足够了。*

---

半精度加法、减法、乘法、除法，和平方根操作可以通过把半精度操作数转化为单精度、使用单精度算术执行操作再转化回单精度 *[18]* 来忠实地模拟。对于 *RNE* 和 *RMM* 舍入模式，使用这种方法执行半精度融合乘加，在某些输入时会产生 *1-ulp* 的误差。

把 *8* 位或 *16* 位整数转化为半精度，可以通过先转化为单精度、再转化到半精度来模拟。从 *32* 位整数的转换可以用首先转化为双精度来模拟。如果 *D* 扩展不存在，而且在 *RNE* 或 *RMM* 模式下 *1-ulp* 的误差是可容忍的，那么 *32* 位整数也可以先被转化为单精度。同样的标注也适用于没有 *Q* 扩展时的从 *64* 位整数的转换。

# 第十七章　RVWMO 内存一致性模型（2.0 版本）

这章定义了 RISC-V 内存一致性模型。内存一致性模型是一组规则的集合，它指定了可以被内存的加载所返回的值。RISC-V 使用一个叫做“RVWMO”（RISC-V 弱内存次序）的内存模型，它被设计来为架构提供构建高性能可扩展设计的灵活性，并同时支持易处理的编程模型。

在 RVWMO 下，从同一硬件线程的其它内存指令的视角来看，运行在单一硬件线程的代码看似有序地执行，但是从另一个硬件线程的内存指令，可能观察到第一个硬件线程的内存指令正在以一种不同的次序被执行。因此，多线程代码可能需要显式的同步，来保证来自不同硬件线程的内存指令之间的次序。基础 RISC-V ISA 出于这个目的，提供了一个 FENCE 指令，它描述在 2.7节中，同时原子扩展“A”额外定义了“加载-保留”/“存储-条件”和原子性“读-修改-写”指令。

用于未对齐原子性的标准 ISA 扩展“Zam”（第 二十三章）和用于全存储排序的标准 ISA 扩展“Ztso”（第 二十五章）为 RVWMO 增加了特定于那些扩展的额外的规则。

这个规范的附录提供了公理化的和操作规范化的内存一致性以及补充说明材料。

---

这章定义了用于规则的主内存操作的内存模型。使用 *I/O* 内存、指令获取、*FENCE.I*、页表游走和 *SFENCE.VMA* 的内存模型交互还没有被规范化。上述中的一些或全部可能在这个规范未来的修订中被规范化。*RV128* 基础 *ISA* 和未来的 *ISA* 扩展（例如“*V*”向量扩展和“*J*”*JIT* 扩展）也将需要被并入未来的修订中。

支持不同宽度同时进行重叠的内存访问的内存一致性模型仍然是学术研究的一个积极区域，并且还没有被完全理解。关于不同尺寸的内存访问如何在 *RVWMO* 下交互的细节是我们当前能做到的最好的，但是当新问题被揭露后，它们将不得不再修订。

## 17.1　RVWMO 内存模型的定义

*全局内存次序*，即所有硬件线程所产生的内存操作的总体次序，据此定义了 RVWMO 内存模型。总的来说，一个多线程程序有许多种不同可能的执行，而每种执行有其自己相应的全局内存次序。

全局内存次序定义在由内存指令生成的原始的加载和存储操作之上。然后它将受到本章余下部分定义的约束的限制。任何满足所有内存模型约束的执行都是合法的执行（至少在内存模型所关注的方面如此）。

### 内存模型原语

在内存操作上的*程序次序*反映了生成每个加载和存储的指令在硬件线程的动态指令流中的次序；即，简单有序处理器将执行的该硬件线程的指令的次序。

内存访问指令造成了内存操作。*内存操作*可以是一个*加载操作*、*存储操作*，或者是二者同时。所有的内存操作都是"单拷贝原子"的：它们可以永远不会被观察到处于一种部分完成的状态。

在 RV32GC 和 RV64GC 的指令之中，每个对齐的内存指令都确切造成一次内存操作和两个异常。首先，一次不成功的 SC 指令不会造成任何内存操作。第二，如果 XLEN<64，就像 14.3节中所陈述的和下面澄清的那样，那么 FLD 和 FSD 指令每次可以造成多个内存操作。一个对齐的 AMO 指令造成单次内存操作，它同时是一个加载操作和一个存储操作。

---

*RV128* 基础指令集中的指令和诸如 *V*（向量）和 *P*（*SMID*）的未来 *ISA* 扩展中的指令可能造成多个内存操作。然而对于这些扩展的内存模型还没有被规范化。

未对齐的加载或存储指令可能被分解为任意粒度的组件内存操作的集合。对于 XLEN<64 的 FLD 或 FSD 指令也可能被分解为任意粒度的组件内存操作的集合。通过这样的指令生成的内存操作并不按照相互间的程序次序被排序，但是它们可以根据"在程序次序中位于先前或后续指令所生成的内存操作"来正常地排序。原子扩展"A"完全不需要执行环境提供未对齐的原子指令；然而，如果通过"Zam"扩展支持未对齐的原子指令，那么 LR、SC 和 AMO 可以按照未对齐原子指令的原子性公理的约束而被分解，该约束定义在第 二十三章中。

---

将未对齐内存操作分解下到字节粒度有利于在原本不支持未对齐访问的实现上进行模拟。例如，这种实现可能简单地逐个迭代未对齐的访问的字节。

如果在程序次序中，LR 先于 SC，并且在它们之间没有其它的 LR 或 SC 指令，那么 LR 指令和 SC 指令被称作成对的；对应的内存操作也被称为成对的（除了在 SC 失败的情况中，那里没有产生存储操作）。决定一个 SC 是否必定成功、可能成功、或者必定失败的条件的完整列表定义在 10.2节中。

加载和存储操作也可以从下列集合中携带一个或多个次序注释："acquire-RCpc"、"acquire-RCsc"、"release-RCpc"和"release-RCsc"。一个设置了 *aq* 的 AMO 或 LR 指令具有"acquire-RCsc"注释。一个设置了 *rl* 的 AMO 或 SC 指令具有"releaseRCsc"注释。同时设置了 *aq* 和 *rl* 的 AMO、LR 或 SC 指令也同时有"acquire-RCsc"和"release-RCsc"注释。

为了方便，我们使用术语“acquire 注释”来指代一个 acquire-RCpc 注释或者一个 acquire-RCsc 注释。类似地，用“release 注释”指代一个 release-RCpc 注释或者一个 release-RCsc 注释。用“RCpc 注释”指代一个 acquire-RCpc 注释或者一个 releaseRCpc 注释。用“RCsc 注释”指代一个 acquire-RCsc 注释或者一个 release-RCsc 注释。

---

在内存模型文献中，术语“*RCpc*”代表带有与处理器一致的同步操作的释放一致性，而术语“*RCsc*”代表带有顺序一致的同步操作的释放一致性 *[6]*。

虽然在文献中对于 *acquire* 注释和 *release* 注释有许多不同的定义，在 *RVWMO* 的上下文中，这些术语由保留的程序次序规则5−7简洁而完整地定义。

“*RCpc*”注释目前只被用在各标准扩展“*Ztso*”（第 二十五章）被隐式地分配给每个内存访问时。甚至，尽管 *ISA* 目前既不包括原生的“加载-获取”或者“存储-释放”指令，也因此不包括其中的 *RCpc* 变量，*RVWMO* 模型本身被设计为向前兼容的，可以在未来的扩展中把上面的任何或者所有的潜在的附件兼容进 *ISA* 中。

### 句法依赖

RVWMO 内存模型的定义部分依赖于句法依赖的概念，后者定义如下。

在定义的依赖的上下文中，“寄存器”或者指代一个完整的通用目的寄存器，或者指代 CSR 的某些部分，或者指代一个完整的 CSR。通过 CSR 追踪的依赖的粒度特定于每个 CSR，并在 17.2节中定义。

句法依赖的定义依据于指令的源寄存器、指令的目的寄存器，和指令从它们的源寄存器到目的寄存器携带依赖的方式。本节提供了一个所有这些术语的通用的定义；然而， 17.3节提供了每个指令的详细信息的一个完整的列表。

总体上，对于一个指令 $i$，如果满足任意下列条件，源寄存器是寄存器 $r$，而不是 `x0`：

- 在 $i$ 的操作码中，*rs1*、*rs2* 或者 *rs3* 被设置为 $r$
- $i$ 是一个 CSR 指令，并且在 $i$ 的操作码中，*csr* 被设置为 $r$（除非 $i$ 是 CSRRW 或 CSRRWI，并且 *rd* 被设置为 *x0*）
- $r$ 是一个 CSR，且对于 $i$，$r$ 是一个隐式的源寄存器，就像 17.3节中定义的那样
- 对于 $i$，$r$ 是一个作为另一个源寄存器的别名的 CSR

内存指令也进一步指定了哪个源寄存器是地址源寄存器，以及哪个是数据源寄存器。

总体上，对于一个指令 $i$，如果满足任意下列条件，目的寄存器是寄存器 $r$，而不是 *x0*：

- 在 i 的操作码中，rd 被设置为 r
- $i$ 是一个 CSR 指令，且在 $i$ 的操作码中，*csr* 被设置为 $r$（除非 $i$ 是 CSRRS 或 CSRRC，并且 *rs1* 被设置为 *x0*；或者 $i$ 是 CSRRI 或 CSRRCI，并且 uimm[4:0] 被设置为 0）
- $r$ 是一个 CSR，并且对于 $i$，$r$ 是一个隐式的目的寄存器，正如 17.3节中定义的那样
- 对于 $i$，$r$ 是一个作为另一个目的寄存器的别名的 CSR

大多数非内存指令携带有从它们的每个源寄存器到它们的每个目的寄存器的依赖。然而，对于这个规则是有例外的；见 17.3节。

如果满足下列之一，通过 $i$ 的目的寄存器 $s$ 和 $j$ 的源寄存器 $r$，指令 $j$ 有一个关于指令 $i$ 的**句法依赖**：

- $s$ 与 $r$ 相同，并且按程序次序，在 i 和 j 之间没有指令把 $r$ 作为目的寄存器
- 按程序次序，在 $i$ 和 $j$ 之间有指令 $m$，使得满足所有下列条件：
  1. 通过目的寄存器 $q$ 和源寄存器 $r$，$j$ 有一个关于 $m$ 的句法依赖
  2. 通过目的寄存器 $s$ 和源寄存器 $p$，$m$ 有一个关于 $i$ 的句法依赖
  3. $m$ 携带有从 $p$ 到 $q$ 的依赖

最后，在下面的定义中，$a$ 和 $b$ 是两个内存操作，而 $i$ 和 $j$ 是分别生成 $a$ 和 $b$ 的指令。

$b$ 有一个关于 a 的**句法地址依赖**，如果 $r$ 是 $j$ 的一个地址源寄存器，而 $j$ 通过源寄存器 $r$ 有一个关于 $i$ 的句法依赖

$b$ 有一个关于 $a$ 的**句法数据依赖**，如果 $b$ 是一个存储操作，$r$ 是 $j$ 的一个数据源寄存器，且 $j$ 通过源寄存器 $r$ 有一个关于 $i$ 的句法依赖 $b$ has a *syntactic data dependency* on $a$ if $b$ is a store operation, $r$ is a data source register for $j$, and $j$ has a syntactic dependency on $i$ via source register $r$

$b$ 有一个关于 $a$ 的**句法控制依赖**，如果按程序次序，在 $i$ 和 $j$ 之间有一个指令 $m$，使 $m$ 是一个分支、或者间接跳转，并且 $m$ 有一个关于 $i$ 的句法依赖。

---

总的来说，非 *AMO* 加载指令没有数据源寄存器，而无条件非 *AMO* 存储指令没有目的寄存器。然而，一个成功的 *SC* 指令被认为在 rd 中指定了寄存器作为目的寄存器，并因此对于一条指令，可能有一个关于程序次序中先于它的成功 *SC* 指令的句法依赖。

### 保留的程序次序

对于任意给定的程序执行，全局内存次序遵循着各个硬件线程的内存次序中的一部分（但不是所有）。必须被全局内存次序所遵循的程序次序子集被称为保留的程序次序。

保留的程序次序的完整定义如下（且注意 AMO 是同时进行加载和存储）：在保留的程序次序中，内存操作 $a$ 先于内存操作 $b$（并因此在全局内存次序中也是如此），如果以程序次序 $a$ 先于 $b$，$a$ 和 $b$ 都访问常规的主内存（而不是 I/O 区域），并且满足任何下列：

- 重叠地址次序：
  1. $b$ 是一个存储操作，且 $a$ 和 $b$ 访问了重叠的内存地址
  2. $a$ 和 $b$ 是加载操作，$x$ 是 $a$ 和 $b$ 都读取的一个字节，以程序次序在 $a$ 和 $b$ 之间没有对 $x$ 的存储操作，并且 $a$ 和 $b$ 返回由不同的内存操作所写入的 $x$ 的值
  3. $a$ 是由 AMO 或 SC 指令生成的操作，$b$ 是加载操作，且 $b$ 返回一个由 $a$ 写入的值
- 显示同步
  4. 在 $b$ 之前有一个排序 $a$ 的 FENCE 指令
  5. $a$ 有一个 acquire 注释
  6. $b$ 有一个 release 注释
  7. $a$ 和 $b$ 都有 RCsc 注释
  8. $a$ 与 $b$ 是成对的
- 句法依赖
  9. b 有一个关于 a 的句法地址依赖
  10. $b$ 有一个关于 $a$ 的句法数据依赖
  11. $b$ 是一个存储操作，且 $b$ 有一个关于 $a$ 的句法控制依赖
- 流水线依赖
  12. $b$ 是一个加载操作，且按程序次序，在 $a$ 和 $b$ 之间存在某些存储操作 $m$，使得 $m$ 有一个关于 $a$ 的地址依赖或数据依赖，而 $b$ 返回一个由 $m$ 写入的值
  13. $b$ 是一个存储操作，且按程序次序，在 $a$ 和 $b$ 之间存在某些指令 $m$，使 $m$ 有一个关于 $a$ 的地址依赖

**内存模型公理**

只有当存在一个符合保留的程序次序并且满足加载值公理、原子性公理和进度公理的全局内存次序时，RISC-V 程序的执行遵循 RVWMO 内存一致性模型。

**加载值公理** 每个加载 $i$ 的各个位所返回的值，由下列存储中在全局内存次序中最近的那个写到该位：

1. 写该位，并且在全局内存次序中先于 $i$ 的存储
2. 写该位，并且在程序次序中先于 $i$ 的存储

**原子性公理** 如果 $r$ 和 $w$ 是由一个硬件线程 $h$ 中对齐的 LR 和 SC 指令所生成的配对的加载和存储操作，$s$ 是一个对于字节 $x$ 的存储，而 $r$ 返回 $s$ 所写的值，那么在全局内存次序中，$s$ 必须先于 $w$。并且在全局内存次序中，在 $s$ 之后、$w$ 之前，没有来自同一硬件线程的不同于 $h$ 的存储。

---

原子性公理理论上支持不同宽度的 *LR/SC* 对，以及不匹配的地址，因为允许实现在这种情况中使 *SC* 操作成功。然而，在实际中，我们希望这种式样是稀有的，并且不鼓励使用它们。

**进度公理** 在全局内存次序之中，任何内存操作之前都不能有其它内存操作的无限序列。

## 17.2 CSR 依赖跟踪粒度

| 名称 | 作为独立单元被追踪的部分 | 别称 |
|---|---|---|
| `fflags` | Bits 4, 3, 2, 1, 0 | `fcsr` |
| `frm` | CSR 整体 | `fcsr` |
| `fcsr` | 位 7-5, 4, 3, 2, 1, 0 | `fflags`, `frm` |

表 17.1: 通过 CSR 跟踪的句法依赖的粒度

注意：没有列出只读的 CSR，因为它们不参与句法依赖的定义。

## 17.3 源寄存器和目的寄存器列表

这节提供了每个指令的源寄存器和目的寄存器的具体列表。这些列表被用于定义 17.1节中的句法依赖。

术语“累积 CSR”被用于描述一种 CSR，它同时是一个源寄存器和一个目的寄存器，但是只携带一个从它自己到它自己的依赖。

除非另有说明，指令携带的依赖是：从“源寄存器”列中的各个源寄存器到“目的寄存器”列中的各个目的寄存器的依赖、从“源寄存器”列中的各个源寄存器到“累积 CSR”列中的各个 CSR 的依赖，以及从“累积 CSR”列中的各个 CSR 到其自身的依赖。

要点：Key:

$^{A}$ 地址源寄存器

$^{D}$ 数据源寄存器

† 指令没有携带从任何源寄存器到任何目的寄存器的依赖

‡ 指令按照指定携带了从源寄存器到目的寄存器的依赖

**RV32I 基础整数指令集**

| | 源<br>寄存器 | 目的<br>寄存器 | 累积<br>CSR |
|---|---|---|---|
| LUI | | *rd* | |
| AUIPC | | *rd* | |
| JAL | | *rd* | |
| JALR† | *rs1* | *rd* | |
| BEQ | *rs1*, *rs2* | | |
| BNE | *rs1*, *rs2* | | |
| BLT | *rs1*, *rs2* | | |
| BGE | *rs1*, *rs2* | | |
| BLTU | *rs1*, *rs2* | | |
| BGEU | *rs1*, *rs2* | | |
| LB† | $rs1^{A}$ | *rd* | |
| LH† | $rs1^{A}$ | *rd* | |
| LW† | $rs1^{A}$ | *rd* | |
| LBU† | $rs1^{A}$ | *rd* | |
| LHU† | $rs1^{A}$ | *rd* | |
| SB | $rs1^{A}$, $rs2^{D}$ | | |
| SH | $rs1^{A}$, $rs2^{D}$ | | |
| SW | $rs1^{A}$, $rs2^{D}$ | | |
| ADDI | *rs1* | *rd* | |
| SLTI | *rs1* | *rd* | |

| | | | |
|---|---|---|---|
| SLTIU | *rs1* | *rd* | |
| XORI | *rs1* | *rd* | |
| ORI | *rs1* | *rd* | |
| ANDI | *rs1* | *rd* | |
| SLLI | *rs1* | *rd* | |
| SRLI | *rs1* | *rd* | |
| SRAI | *rs1* | *rd* | |
| ADD | *rs1*, *rs2* | *rd* | |
| SUB | *rs1*, *rs2* | *rd* | |
| SLL | *rs1*, *rs2* | *rd* | |
| SLT | *rs1*, *rs2* | *rd* | |
| SLTU | *rs1*, *rs2* | *rd* | |
| XOR | *rs1*, *rs2* | *rd* | |
| SRL | *rs1*, *rs2* | *rd* | |
| SRA | *rs1*, *rs2* | *rd* | |
| OR | *rs1*, *rs2* | *rd* | |
| AND | *rs1*, *rs2* | *rd* | |
| FENCE | | | |
| FENCE.I | | | |
| ECALL | | | |
| EBREAK | | | |

**RV32I 基础整数指令集（续）**

| | 源<br>寄存器 | 目的<br>寄存器 | 累积<br>CSR | |
|---|---|---|---|---|
| CSRRW‡ | *rs1*, *csr** | *rd*, *csr* | | * 除非 *rd*=x0 |
| CSRRS‡ | *rs1*, *csr* | *rd**, *csr* | | * 除非 *rs1*=x0 |
| CSRRC‡ | *rs1*, *csr* | *rd**, *csr* | | * 除非 *rs1*=x0 |

‡carries a dependency from *rs1* to *csr* and from *csr* to *rd*

**RV32I 基础整数指令集（续）**

| | 源<br>寄存器 | 目的<br>寄存器 | 累积<br>CSR | |
|---|---|---|---|---|
| CSRRWI‡ | *csr** | *rd*, *csr* | | *unless *rd*=x0 |
| CSRRSI‡ | *csr* | *rd*, *csr** | | *unless uimm[4:0]=0 |
| CSRRCI‡ | *csr* | *rd*, *csr** | | *unless uimm[4:0]=0 |

‡carries a dependency from *csr* to *rd*

**RV64I 基础整数指令集**

| | 源<br>寄存器 | 目的<br>寄存器 | 累积<br>CSR |
|---|---|---|---|
| LWU† | $rs1^A$ | *rd* | |
| LD† | $rs1^A$ | *rd* | |
| SD | $rs1^A$, $rs2^D$ | | |
| SLLI | *rs1* | *rd* | |
| SRLI | *rs1* | *rd* | |
| SRAI | *rs1* | *rd* | |
| ADDIW | *rs1* | *rd* | |
| SLLIW | *rs1* | *rd* | |
| SRLIW | *rs1* | *rd* | |
| SRAIW | *rs1* | *rd* | |
| ADDW | *rs1*, *rs2* | *rd* | |
| SUBW | *rs1*, *rs2* | *rd* | |
| SLLW | *rs1*, *rs2* | *rd* | |
| SRLW | *rs1*, *rs2* | *rd* | |
| SRAW | *rs1*, *rs2* | *rd* | |

**RV32M 标准扩展**

| | 源<br>寄存器 | 目的<br>寄存器 | 累积<br>CSR |
|---|---|---|---|
| MUL | *rs1*, *rs2* | *rd* | |
| MULH | *rs1*, *rs2* | *rd* | |
| MULHSU | *rs1*, *rs2* | *rd* | |
| MULHU | *rs1*, *rs2* | *rd* | |
| DIV | *rs1*, *rs2* | *rd* | |
| DIVU | *rs1*, *rs2* | *rd* | |
| REM | *rs1*, *rs2* | *rd* | |
| REMU | *rs1*, *rs2* | *rd* | |

**RV64M 标准扩展**

| | 源<br>寄存器 | 目的<br>寄存器 | 累积<br>CSR |
|---|---|---|---|
| MULW | *rs1*, *rs2* | *rd* | |
| DIVW | *rs1*, *rs2* | *rd* | |
| DIVUW | *rs1*, *rs2* | *rd* | |
| REMW | *rs1*, *rs2* | *rd* | |
| REMUW | *rs1*, *rs2* | *rd* | |

**RV32A 标准扩展**

| | 源<br>寄存器 | 目的<br>寄存器 | 累积<br>CSR |
|---|---|---|---|
| LR.W† | $rs1^A$ | *rd* | |
| SC.W† | $rs1^A$, $rs2^D$ | *rd** | |
| AMOSWAP.W† | $rs1^A$, $rs2^D$ | *rd* | |
| AMOADD.W† | $rs1^A$, $rs2^D$ | *rd* | |
| AMOXOR.W† | $rs1^A$, $rs2^D$ | *rd* | |
| AMOAND.W† | $rs1^A$, $rs2^D$ | *rd* | |
| AMOOR.W† | $rs1^A$, $rs2^D$ | *rd* | |
| AMOMIN.W† | $rs1^A$, $rs2^D$ | *rd* | |
| AMOMAX.W† | $rs1^A$, $rs2^D$ | *rd* | |
| AMOMINU.W† | $rs1^A$, $rs2^D$ | *rd* | |
| AMOMAXU.W† | $rs1^A$, $rs2^D$ | *rd* | |

* 如果成功

**RV64A 标准扩展**

| | 源<br>寄存器 | 目的<br>寄存器 | 累积<br>CSR | |
|---|---|---|---|---|
| LR.D† | $rs1^A$ | $rd$ | | |
| SC.D† | $rs1^A$, $rs2^D$ | $rd$* | | * 如果成功 |
| AMOSWAP.D† | $rs1^A$, $rs2^D$ | $rd$ | | |
| AMOADD.D† | $rs1^A$, $rs2^D$ | $rd$ | | |
| AMOXOR.D† | $rs1^A$, $rs2^D$ | $rd$ | | |
| AMOAND.D† | $rs1^A$, $rs2^D$ | $rd$ | | |
| AMOOR.D† | $rs1^A$, $rs2^D$ | $rd$ | | |
| AMOMIN.D† | $rs1^A$, $rs2^D$ | $rd$ | | |
| AMOMAX.D† | $rs1^A$, $rs2^D$ | $rd$ | | |
| AMOMINU.D† | $rs1^A$, $rs2^D$ | $rd$ | | |
| AMOMAXU.D† | $rs1^A$, $rs2^D$ | $rd$ | | |

**RV32F 标准扩展**

| | 源寄存器 | 目的寄存器 | 累积CSR | |
|---|---|---|---|---|
| FLW† | *rs1*$^A$ | *rd* | | |
| FSW | *rs1*$^A$, *rs2*$^D$ | | | |
| FMADD.S | *rs1*, *rs2*, *rs3*, frm* | *rd* | NV, OF, UF, NX | * 如果 rm=111 |
| FMSUB.S | *rs1*, *rs2*, *rs3*, frm* | *rd* | NV, OF, UF, NX | * 如果 rm=111 |
| FNMSUB.S | *rs1*, *rs2*, *rs3*, frm* | *rd* | NV, OF, UF, NX | * 如果 rm=111 |
| FNMADD.S | *rs1*, *rs2*, *rs3*, frm* | *rd* | NV, OF, UF, NX | * 如果 rm=111 |
| FADD.S | *rs1*, *rs2*, frm* | *rd* | NV, OF, NX | * 如果 rm=111 |
| FSUB.S | *rs1*, *rs2*, frm* | *rd* | NV, OF, NX | * 如果 rm=111 |
| FMUL.S | *rs1*, *rs2*, frm* | *rd* | NV, OF, UF, NX | * 如果 rm=111 |
| FDIV.S | *rs1*, *rs2*, frm* | *rd* | NV, DZ, OF, UF, NX | * 如果 rm=111 |
| FSQRT.S | *rs1*, frm* | *rd* | NV, NX | * 如果 rm=111 |
| FSGNJ.S | *rs1*, *rs2* | *rd* | | |
| FSGNJN.S | *rs1*, *rs2* | *rd* | | |
| FSGNJX.S | *rs1*, *rs2* | *rd* | | |
| FMIN.S | *rs1*, *rs2* | *rd* | NV | |
| FMAX.S | *rs1*, *rs2* | *rd* | NV | |
| FCVT.W.S | *rs1*, frm* | *rd* | NV, NX | * 如果 rm=111 |
| FCVT.WU.S | *rs1*, frm* | *rd* | NV, NX | * 如果 rm=111 |
| FMV.X.W | *rs1* | *rd* | | |
| FEQ.S | *rs1*, *rs2* | *rd* | NV | |
| FLT.S | *rs1*, *rs2* | *rd* | NV | |
| FLE.S | *rs1*, *rs2* | *rd* | NV | |
| FCLASS.S | *rs1* | *rd* | | |
| FCVT.S.W | *rs1*, frm* | *rd* | NX | * 如果 rm=111 |
| FCVT.S.WU | *rs1*, frm* | *rd* | NX | * 如果 rm=111 |
| FMV.W.X | *rs1* | *rd* | | |

**RV64F 标准扩展**

| | 源寄存器 | 目的寄存器 | 累积CSR | |
|---|---|---|---|---|
| FCVT.L.S | *rs1*, frm* | *rd* | NV, NX | * 如果 rm=111 |
| FCVT.LU.S | *rs1*, frm* | *rd* | NV, NX | * 如果 rm=111 |
| FCVT.S.L | *rs1*, frm* | *rd* | NX | * 如果 rm=111 |
| FCVT.S.LU | *rs1*, frm* | *rd* | NX | * 如果 rm=111 |

**RV32D 标准扩展**

| | 源<br>寄存器 | 目的<br>寄存器 | 累积<br>CSR | |
|---|---|---|---|---|
| FLD† | *rs1*$^{A}$ | *rd* | | |
| FSD | *rs1*$^{A}$, *rs2*$^{D}$ | | | |
| FMADD.D | *rs1*, *rs2*, *rs3*, frm* | *rd* | NV, OF, UF, NX | * 如果 rm=111 |
| FMSUB.D | *rs1*, *rs2*, *rs3*, frm* | *rd* | NV, OF, UF, NX | * 如果 rm=111 |
| FNMSUB.D | *rs1*, *rs2*, *rs3*, frm* | *rd* | NV, OF, UF, NX | * 如果 rm=111 |
| FNMADD.D | *rs1*, *rs2*, *rs3*, frm* | *rd* | NV, OF, UF, NX | * 如果 rm=111 |
| FADD.D | *rs1*, *rs2*, frm* | *rd* | NV, OF, NX | * 如果 rm=111 |
| FSUB.D | *rs1*, *rs2*, frm* | *rd* | NV, OF, NX | * 如果 rm=111 |
| FMUL.D | *rs1*, *rs2*, frm* | *rd* | NV, OF, UF, NX | * 如果 rm=111 |
| FDIV.D | *rs1*, *rs2*, frm* | *rd* | NV, DZ, OF, UF, NX | * 如果 rm=111 |
| FSQRT.D | *rs1*, frm* | *rd* | NV, NX | * 如果 rm=111 |
| FSGNJ.D | *rs1*, *rs2* | *rd* | | |
| FSGNJN.D | *rs1*, *rs2* | *rd* | | |
| FSGNJX.D | *rs1*, *rs2* | *rd* | | |
| FMIN.D | *rs1*, *rs2* | *rd* | NV | |
| FMAX.D | *rs1*, *rs2* | *rd* | NV | |
| FCVT.S.D | *rs1*, frm* | *rd* | NV, OF, UF, NX | * 如果 rm=111 |
| FCVT.D.S | *rs1* | *rd* | NV | |
| FEQ.D | *rs1*, *rs2* | *rd* | NV | |
| FLT.D | *rs1*, *rs2* | *rd* | NV | |
| FLE.D | *rs1*, *rs2* | *rd* | NV | |
| FCLASS.D | *rs1* | *rd* | | |
| FCVT.W.D | *rs1*, frm* | *rd* | NV, NX | * 如果 rm=111 |
| FCVT.WU.D | *rs1*, frm* | *rd* | NV, NX | * 如果 rm=111 |
| FCVT.D.W | *rs1* | *rd* | | |
| FCVT.D.WU | *rs1* | *rd* | | |

**RV64D 标准扩展**

| | 源<br>寄存器 | 目的<br>寄存器 | 累积<br>CSR | |
|---|---|---|---|---|
| FCVT.L.D | *rs1*, frm* | *rd* | NV, NX | * 如果 rm=111 |
| FCVT.LU.D | *rs1*, frm* | *rd* | NV, NX | * 如果 rm=111 |
| FMV.X.D | *rs1* | *rd* | | |
| FCVT.D.L | *rs1*, frm* | *rd* | NX | * 如果 rm=111 |
| FCVT.D.LU | *rs1*, frm* | *rd* | NX | * 如果 rm=111 |
| FMV.D.X | *rs1* | *rd* | | |

# 第十八章　用于压缩指令的“C”标准扩展 (2.0 版本)

这章描述了 RISC-V 标准压缩指令集扩展，命名为“C”，它通过为常见的操作添加短 16 位指令编码，减少了静态和动态的代码尺寸。C 扩展可以被添加到任何基础 ISA（RV32、RV64、RV128），而我们使用通用术语“RVC”来涵盖所有这些加入了 C 扩展的 ISA。通常，程序中的 50% 60% 的 RISC-V 指令可以被 RVC 指令替代，从而减少了 25% 30% 的代码尺寸。

## 18.1　概览

RVC 使用了一个简单的压缩策略，它提供常见 32 位 RISC-V 指令的较短的 16 位版本，当：

- 立即数或地址偏移量较小，或者
- 其中一个寄存器是零寄存器（x0）、ABI 链接寄存器（x1），或者 ABI 栈指针（x2），或者
- 目的寄存器和第一个源寄存器完全相同，或者
- 使用的寄存器是 8 个最流行的寄存器。

C 扩展与所有其它的标准指令扩展相兼容。C 扩展允许 16 位指令与 32 位指令自由混合，其中 32 位指令现在可以从任何 16 位边界开始，也就是说，IALIGN = 16。除了 C 扩展外，没有指令可以引发指令地址未对齐异常。

---

*在原始的 32 位指令上移除 32 位对齐限制，可以显著提高代码密度。*

压缩的指令编码大多在 RV32C、RV64C 和 RV128C 之间通用，但是如表 18.4中显示的那样，根据基础 ISA，也有少量操作码被用于不同的目的。例如，较宽的地址空间 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，然而这些相同的操作码在 RV32C 和 RV64C 中被用于双精度浮点值的加载和存储。如果 C 扩展被实现了，不论相关的标准浮点扩展（F 和/或 D）是否也被实现，都必须提供合适的压缩的浮点加载和存储指令。此外，RV32C 包括了一个压缩的跳转和链接指令，以压缩较短范围的子例程调用，而在 RV64C 和 RV128C 中，相同的操作码被用于压缩 ADDIW。

双精度加载和存储是静态和动态指令的一个重要部分，因此有必要将其包含在 *RV32C* 和 *RV64C* 的编码中。

尽管对于当前支持 *ABI* 的编译的基准，单精度加载和存储不是静态或动态压缩的一个重要来源，但是对于只提供硬件单精度浮点单元、并且有只支持单精度浮点数的 *ABI* 的微控制器来说，在衡量基准中，单精度加载和存储的使用至少与双精度加载和存储频率相同。因此，在 *RV32C* 中，有必要为这些操作提供压缩的支持。

对于微控制器，较短范围的子例程调用更可能出现在小型二进制代码中，因此有必要在 *RV32C* 中包括这些。

尽管在不同基础 *ISA* 下，为了不同的目的重用操作码，会增加文档的复杂性，然而即使是对于支持多个基础 *ISA* 的设计，对其实现的复杂性的影响也很小。压缩的浮点加载和存储变体使用与较宽的整数加载和存储相同的指令格式，带有相同的寄存器修饰符。

RVC 的设计有一个约束，每个 RVC 指令扩展到某个基础 ISA（RV32I/E、RV64I 或 RV128I）或者现有的 F 和 D 标准扩展中的一个单独的 32 位指令。采用这个约束有两个主要的好处：

- 硬件设计可以在解码期间简单地扩展 RVC 指令，简化了验证并最小化了对现存微架构的修改。
- 编译器可以感知不到 RVC 扩展，而把代码压缩留给汇编器和链接器，即使一个能感知到压缩的编译器通常将能够产生更好的结果。

我们感觉，在 *C* 和基础 *IFD* 指令之间的简单一对一映射的所减少的多重复杂度远远超出了稍微更密集的编码的潜在收益，这种编码或者添加了额外的只能在 *C* 扩展中支持的指令，或者允许在一个 *C* 指令中进行多重 *IFD* 指令的编码。

注意，C 扩展并没有被设计为一个独立的 ISA，意味着它必须随着一个基础 ISA 使用，这一点很重要。

可变长度的指令集长期被用来改进代码密度。例如，*IBM Stretch [5]*，开发于 *1950* 年代晚期，有一个带有 *32* 位和 *64* 位指令的 *ISA*，那里某些 *32* 位指令是完整的 *64* 位指令的压缩版本。*Stretch* 也采用了限制寄存器集的概念，这些寄存器在某些较短的指令格式中是可编址的，而短分支指令只能引用一个索引寄存器。稍后的 *IBM 360* 架构 *[4]* 支持了一个简单的可变长度指令编码，包括 *16* 位、*32* 位或 *48* 位指令格式。

在 *1963* 年，*CDC* 介绍了 *Cray* 设计的 *CDC 6600 [20]*，一个 *RISC* 架构的前身，它引入了带有两种长度（*15* 位和 *30* 位）指令的富寄存器的加载-存储架构。稍后的 *Cray-1* 设计使用了非常相似的指令格式，带有 *16* 位和 *32* 位指令长度。

在 *1980* 年代的最初的 *RISC ISA*，都将性能放在第一位，代码尺寸放在第二位，这对于工作站环境是合理的，但是对于嵌入式环境则不然。因此，*ARM* 和 *MIPS* 随后都推出了提供更小代码尺寸的 *ISA* 版本，通过提供备选的 *16* 位宽指令集来代替标准的 *32* 位宽指令。压缩的 *RISC ISA* 相对于它们的起点，减少了大约 *25 30%* 的代码尺寸，生成的代码显著小于 *80x86*。

这个结果让一些人感到惊讶，因为他们的直觉是，可变长度的 *CISC ISA* 应当比只提供了 *16* 位和 *32* 位格式的 *RISC ISA* 更小。

由于原始的 *RISC ISA* 没有留出足够的操作码空间来自由地包括这些计划之外的压缩指令，它们转而作为完整的新的 *ISA* 进行开发。这意味着编译器需要不同的代码生成器用于独立的压缩 *ISA*。第一代压缩 *RISC ISA* 扩展（例如，*ARM Thumb* 和 *MIPS16*）只使用了固定的 *16* 位指令尺寸，这很好地减少了静态代码尺寸，但是引起了动态指令数目的增长，这导致了与原始的定宽 *32* 位指令尺寸相比更低的性能。这引起了第二代压缩 *RISC ISA* 设计的发展，使用混合的 *16* 位和 *32* 位指令长度（例如，*ARM Thumb2*、*microMIPS*、*PowerPC VLE*），因此性能与纯 *32* 位指令相似，但是显著节省了代码尺寸。不幸的是，这些不同代际的压缩 *ISA* 是互相不兼容的，也与原始的未压缩的 *ISA* 不兼容，导致了文档、实现和软件工具支持中的明显的复杂性。

在常见的使用 *64* 位的 *ISA* 中，只有 *PowerPC* 和 *microMIPS* 目前支持压缩指令格式。奇怪的是，大多数流行的 *64* 位移动平台（*ARM v8*）的 *64* 位 *ISA* 并没有包括压缩指令格式，而静态代码尺寸和动态指令获取带宽对它们来说是很重要的指标。尽管静态代码尺寸在较大系统中不是主要关心的问题，但是指令获取带宽可能成为运行商业工作负载的服务器（它们经常含有大量的指令工作集）中的主要瓶颈。

得益于 *25* 年的事后观察，*RISC-V* 从一开始就被设计为支持压缩指令的，为 *RVC* 留出了足够的操作码空间，来（与许多其它的扩展一起）作为一个简单的扩展被添加到基础 *ISA* 之上。*RVC* 的哲学是为嵌入式应用减少代码尺寸，并为所有应用提升性能和能源效率以减少指令缓存的缺失。*Waterman* 显示 *RVC* 获取的指令位减少了 *25% 30%*，这减少了 *20% 25%* 的指令缓存缺失，或者说，与将指令缓存尺寸翻倍的性能影响大致相同 *[24]*。

## 18.2 压缩指令格式

表 18.1显示了九个压缩指令格式。CR、CI 和 CSS 可以任意使用 32 个 RVI 寄存器，但是 CIW、CL、CS、CA 和 CB 被限制为只能使用其中的 8 个。表 18.2列出了这些常用的寄存器，它们对应于寄存器 `x8` 到 `x15`。注意，使用栈指针作为基地址寄存器的加载和存储指令有各自独立的版本，因为保存到栈和从栈中恢复是如此普遍，以至于它们要使用 CI 和 CSS 格式，以允许访问所有的 32 个数据寄存器。对于 ADDI4SPN 指令，CIW 支持一个 8 位的立即数。

---

*RISC-V ABI* 被更改为把频繁使用的寄存器映射到寄存器 `x8` — `x15`。这简化了解压缩的解码器，因为它有一组连续的自然对齐的寄存器号，并且也与 *RV32E* 基础 *ISA* 兼容，后者只有 *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，带有八个映射到 `f8` 到 `f15` 的寄存器。

---

标准 *RISC-V* 调用约定把最频繁使用的浮点寄存器映射到寄存器 `f8` 到 `f15`，这将允许使用与整数寄存器号相同的寄存器进行解压缩解码。

在所有的指令中，格式都被设计为，将两个寄存器源修饰符位保持在相同位置，而目的寄存器域可以移动。当存在完整的 5 位目的寄存器修饰符时，它位于与 32 位 RISC-V 编码中的相同位置。如果立即数是被符号扩展的，符号扩展总是从位 12 开始。正如在基础规范中的那样，立即数域已经被加扰，以减少必需的立即数 mux 的数目。

---

在指令格式中，立即数域是加扰的，而不是按顺序的，这样在每个指令中，可以使尽可能多的位位于相同位置，因此简化了实现。

对于许多 RVC 指令，值为零的立即数是不允许的，且 `x0` 并非是一个有效的 5 位寄存器修饰符。这些限制为其它的需要更少的操作数位的指令释放了编码空间。

| 格式 | 含义 | 15 14 13 | 12 | 11 10 | 9 8 | 7 6 | 5 4 3 2 | 1 0 |
|---|---|---|---|---|---|---|---|---|
| CR | 寄存器 | funct4 | | rd/rs1 | | rs2 | | op |
| CI | 立即数 | funct3 | imm | rd/rs1 | | imm | | op |
| CSS | 栈相关存储 | funct3 | imm | | | rs2 | | op |
| CIW | 宽立即数 | funct3 | imm | | | | rd′ | op |
| CL | 加载 | funct3 | imm | | rs1′ | imm | rd′ | op |
| CS | 存储 | funct3 | imm | | rs1′ | imm | rs2′ | op |
| CA | 算术 | funct6 | | | rd′/rs1′ | funct2 | rs2′ | op |
| CB | 分支/算术 | funct3 | offset | | rd′/rs1′ | offset | | op |
| CJ | 跳转 | funct3 | jump target | | | | | op |

表 18.1: 压缩的 16 位 RVC 指令格式。

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

表 18.2: 通过 CIW、CL、CS、CA 和 CB 格式的 3 位的 *rs1′*、*rs2′* 和 *rd′* 域指定的寄存器。

## 18.3 加载和存储指令

为了增加 16 位指令的访问范围，数据转移指令使用零扩展的立即数，它按照数据的字节尺寸进行缩放：字 ×4，双字 ×8，四字 ×16。

RVC 提供了加载和存储的两个变体。一个使用 ABI 栈指针 `x2` 作为基础地址，而可以把任意数据寄存器作为目标。另一个可以引用 8 个基础地址寄存器中的一个和 8 个数据寄存器中的一个。

### 基于栈指针的加载和存储

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

这些指令使用 CI 格式。

C.LWSP 从内存把一个 32 位的值加载到寄存器 *rd* 中。它通过将零扩展的偏移量扩大 4 倍，加到栈指针 `x2` 上，来计算出有效地址。它扩展到 `lw rd, offset(x2)`。C.LWSP 只有在 *rd*≠`x0` 时有效；*rd*=`x0` 的代码点被保留。

C.LDSP 是一个 RV64C/RV128C 独有的指令，它从内存加载一个 64 位的值到寄存器 *rd* 中。它通过将零扩展的偏移量扩大 8 倍，加到栈指针 `x2` 上，来计算出有效地址。它扩展到 `ld rd, offset(x2)`。C.LDSP 只有在 *rd*≠`x0` 时有效；*rd*=`x0` 的代码点被保留。

C.LQSP 是一个 RV128C 独有的指令，它从内存加载一个 128 位的值到寄存器 *rd* 中。它通过将零扩展的偏移量扩大 16 倍，加到栈指针 `x2` 上，来计算出有效地址。它扩展到 `lq rd, offset(x2)`。C.LQSP 只在 *rd*≠`x0` 时有效；*rd*=`x0` 的代码点被保留。

C.FLWSP 是一个 RV32FC 独有的指令，它从内存加载一个单精度浮点值到浮点寄存器 *rd* 中。它通过将零扩展的偏移量扩大 4 倍，加到栈指针 `x2` 上，来计算出有效地址。它扩展到 `flw rd, offset(x2)`。

C.FLDSP 是一个 RV32DC/RV64DC 独有的指令，它从内存加载一个双精度浮点值到浮点寄存器 *rd* 中。它通过将零扩展的偏移量扩大 8 倍，加到栈指针 `x2` 上，来计算出有效地址。它扩展到 `fld rd, offset(x2)`。

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

这些指令使用 CSS 格式。

C.SWSP 把一个 32 位的值存储到寄存器 *rs2* 中。它通过将零扩展的偏移量扩大 4 倍，加到栈指针 `x2` 上，来计算出有效地址。它扩展到 `sw rs2, offset(x2)`。

C.SDSP 是一个 RV64C/RV128 独有的指令，它把寄存器 *rs2* 中的一个 64 位的值存储到内存。它通过将零扩展的偏移量扩大 8 倍，加到栈指针 `x2` 上，来计算出有效地址。它扩展到 `sd rs2, offset(x2)`。

C.SQSP 是一个 RV128C 独有的指令，它把寄存器 *rs2* 中的一个 128 位的值存储到内存。它通过将零扩展的偏移量扩大 16 倍，加到栈指针 `x2` 上，来计算出有效地址。它扩展到 `sq rs2, offset(x2)`。

C.FSWSP 是一个 RV32FC 独有的指令，它把浮点寄存器 *rs2* 中的一个单精度浮点值存储到内存。它通过将零扩展的扩大 4 倍，加到栈指针 `x2` 上，来计算出有效地址。它扩展到 `fsw rs2, offset(x2)`。

C.FSDSP 是一个 RV32DC/RV64DC 独有的指令，它把浮点寄存器 *rs2* 中的一个双精度浮点数存储到内存。它通过将零扩展的偏移量扩大 8 倍，加到栈指针 `x2` 上，来计算出有效地址。它扩展到 `fsd rs2, offset(x2)`。

---

在函数的入口/出口处的寄存器保存/恢复代码代表了静态代码尺寸的一大部分。在 *RVC* 中，基于栈指针的压缩的加载和存储可以有效地减少两倍的保存/恢复静态代码尺寸，同时通过减少动态指令带宽来提升性能。

为了进一步减少保存/恢复代码尺寸，在其它 *ISA* 中使用的一个常见的机制是多重加载和多重存储指令。我们考虑过为 *RISC-V* 采用这些指令，但是注意到这些指令的如下缺点：

- 这些指令让处理器的实现复杂化。
- 对于虚拟内存系统，有些数据访问可能驻留在物理内存中，而有些不能，需要为部分执行的指令使用一种新的重启机制。

- 不像其余的 *RVC* 指令，没有等价于多重加载和多重存储的 *IFD*。
- 不像其余的 *RVC* 指令，编译器将不得不注意这些指令，来生成指令和按次序分配寄存器，以最大化它们被保存和存储的机会——因为它们要按顺序次序被保存和存储。
- 简单微架构的实现将限制如何围绕加载多重和存储多重指令调度其它指令，导致潜在的性能损失。
- 期望的顺序寄存器分配可能与为 *CIW*、*CL*、*CS*、*CA* 和 *CB* 格式选择的特征寄存器冲突。*The desire for sequential register allocation might conflict with the featured registers selected for the CIW, CL, CS, CA, and CB formats.*

此外，在软件中，通过 *[25]* 的 *5.6* 节中描述的一种技术，使用子例程调用公共的序言和结语代码，替换序言和结语代码，可以实现大多数收益。

虽然合理的架构可能得出不同的结论，我们决定忽略加载多重和存储多重，代之使用纯软件的方法，调用保持/恢复 *millicode* 例程以获得最大程度的代码尺寸减少。

## 基于寄存器的加载和存储

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

这些指令使用 CL 格式。

C.LW 从内存加载一个 32 位的值到寄存器 *rd′* 中。它通过将零扩展的偏移量扩大 4 倍，加到寄存器 *rs1′* 中的基地址上，来计算出有效地址。它扩展到 `lw rd′, offset(rs1′)`。

C.LD 是一个 RV64C/RV128C 独有的指令，它从内存加载一个 64 位的值到寄存器 *rd′* 中。它通过将零扩展的偏移量扩大 8 倍，加到寄存器 *rs1′* 中的基地址上，来计算出有效地址。它扩展到 `ld rd′, offset(rs1′)`。

C.LQ 是一个 RV128C 独有的指令，它从内存加载一个 128 位的值到寄存器 *rd′* 中。它通过将零扩展的偏移量扩大 16 倍，加到寄存器 *rs1′* 中的基地址上，来计算出有效地址。它扩展到 `lq rd′, offset(rs1′)`。

C.FLW 是一个 RV32FC 独有的指令，它从内存加载一个单精度浮点值到浮点寄存器 *rd′* 中。它通过将零扩展的偏移量扩大 4 倍，加到寄存器 *rs1′* 中的基地址上，来计算出有效地址。它扩展到 `flw rd′, offset(rs1′)`。

C.FLD 是一个 RV32DC/RV64DC 独有的指令，它从内存加载一个双精度浮点值到浮点寄存器 *rd′* 中。它通过将零扩展的偏移量扩大 8 倍，加到寄存器 *rs1′* 中的基地址上，来计算出有效地址。它扩展到 `fld rd′, offset(rs1′)`。

| 15 13 | 12 10 | 9 7 | 6 5 | 4 2 | 1 0 |
|---|---|---|---|---|---|
| funct3 | imm | rs1′ | imm | rs2′ | op |
| 3 | 3 | 3 | 2 | 3 | 2 |
| C.SW | offset[5:3] | base | offset[2\|6] | src | C0 |
| C.SD | offset[5:3] | base | offset[7:6] | src | C0 |
| C.SQ | offset[5\|4\|8] | base | offset[7:6] | src | C0 |
| C.FSW | offset[5:3] | base | offset[2\|6] | src | C0 |
| C.FSD | offset[5:3] | base | offset[7:6] | src | C0 |

这些指令使用 CS 格式。

C.SW 把寄存器 *rs2′* 中的一个 32 位的值存储到内存。它通过将零扩展的偏移量扩大 4 倍，加到寄存器 *rs1′* 中的基地址上，来计算出有效地址。它扩展到 `sw rs2′, offset(rs1′)`。

C.SD 是一个 RV64C/RV128C 独有的指令，它把寄存器 *rs2′* 中的一个 64 位的值存储到内存。它通过将零扩展的偏移量扩大 8 倍，加到寄存器 *rs1′* 中的基地址上，来计算出有效地址。它扩展到 `sd rs2′, offset(rs1′)`。

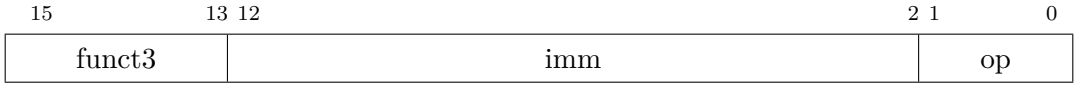
C.SQ 是一个 RV128C 独有的指令，它把寄存器 *rs2′* 中的一个 128 位的值存储到内存。它通过将零扩展的偏移量扩大 16 倍，加到寄存器 *rs1′* 中的基地址上，来计算出有效地址。它扩展到 `sq rs2′, offset(rs1′)`。

C.FSW 是一个 RV32FC 独有的指令，它把浮点寄存器 *rs2′* 中的一个单精度浮点值存储到内存。它通过将零扩展的偏移量扩大 4 倍，加到寄存器 *rs1′* 中的基地址上，来计算出有效地址。它扩展到 `fsw rs2′, offset(rs1′)`。

C.FSD 是一个 RV32DC/RV64DC 独有的指令，它把浮点寄存器 *rs2′* 中的一个双精度浮点值存储到内存。它通过将零扩展的偏移量扩大 8 倍，加到寄存器 *rs1′* 中的基地址上，来计算出有效地址。它扩展到 `fsd rs2′, offset(rs1′)`。

## 18.4 控制转移指令

RVC 提供了无条件跳转指令和条件分支指令。因为带有基础 RVI 指令，所有的 RVC 控制转移指令的偏移量都是 2 字节的倍数。




| 3 | 11 | 2 |
|---|---|---|
| C.J | offset[11\|4\|9:8\|10\|6\|7\|3:1\|5] | C1 |
| C.JAL | offset[11\|4\|9:8\|10\|6\|7\|3:1\|5] | C1 |

这些指令使用 CJ 格式。

C.J 实施无条件控制转移。偏移量被符号扩展，并被加到 `pc` 以形成跳转的目标地址。C.J 因此可以有 ±2 KiB的目标范围。C.J 扩展到 `jal x0, offset`。

C.JAL 是一个 RV32C 独有的指令，它实施与 C.J 相同的操作，但是额外地把跳转（`pc` + 2）之后的指令的地址写到链接寄存器 `x1`。C.JAL 扩展到 `jal x1, offset`。


| 4 | 5 | 5 | 2 |
|---|---|---|---|
| C.JR | src≠0 | 0 | C2 |
| C.JALR | src≠0 | 0 | C2 |

这些指令使用 CR 格式。

C.JR（跳转寄存器）实施到寄存器 *rs1* 中的地址的无条件控制转移。C.JR 扩展到 `jalr x0, 0(rs1)`。C.JR 只有在 *rs1*≠`x0` 时有效；*rs1*=`x0` 的代码点被保留。

C.JALR（跳转和链接寄存器）实施与 C.JR 相同的操作，但是额外把跳转（`pc` + 2）之后的指令的地址写到链接寄存器 `x1`。C.JALR 扩展到 `jalr x1, 0(rs1)`。C.JALR 只有在 *rs1*≠`x0` 时有效；*rs1*=`x0` 的代码点对应于 C.EBREAK 指令。

---

*严格地讲，C.JALR 并不确切地扩展到某个基础 RVI 指令，因为加到 pc 形成链接地址的值是 2，而不像基础 ISA 中那样是 4，但是同时支持 2 字节和 4 字节的偏移量对于基础微架构只是一个非常微小的改变。*

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

这些指令使用 CB 格式。

C.BEQZ 实施条件控制转移。偏移量被符号扩展，并被加到 `pc` 以形成分支目标地址。它因此可以有 ±256 B的目标范围。如果寄存器 *rs1′* 中的值是零，C.BEQZ 采取其分支。它扩展到 `beq rs1′, x0, offset`。

C.BNEZ 的定义类似，但是它采取其分支，是在 *rs1′* 包含一个非零的值时。它扩展到 `bne rs1′, x0, offset`。

## 18.5 整数运算指令

RVC 提供了用于整数运算和常量生成的一些指令。

### 整数常量-生成指令

这两个常量生成指令都使用 CI 指令格式，并且能够把任何整数寄存器作为目标。

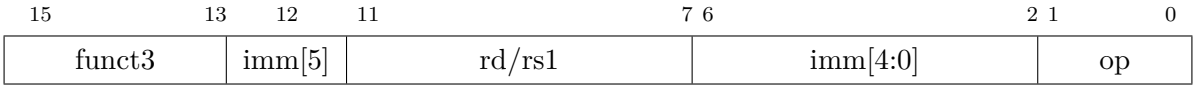
| 15–13 | 12 | 11–7 | 6–2 | 1–0 |
|---|---|---|---|---|
| funct3 | imm[5] | rd | imm[4:0] | op |
| 3 | 1 | 5 | 5 | 2 |
| C.LI | imm[5] | dest≠0 | imm[4:0] | C1 |
| C.LUI | nzimm[17] | dest≠{0, 2} | nzimm[16:12] | C1 |

C.LI 把符号扩展的 6 位立即数 *imm* 加载进寄存器 *rd* 中。C.LI 扩展到 `addi rd, x0, imm`。C.LI 只有当 *rd*≠`x0` 时有效；带有 *rd*=`x0` 的代码点编码了 HINT。

C.LUI 把非零的 6 位立即数域加载到目的寄存器的位 17 - 12，清除底部的 12 位，并把位 17 符号扩展到目的寄存器的所有更高位。C.LUI 扩展到 `lui rd, nzimm`。C.LUI 只有当 *rd*≠{`x0`, `x2`}，并且当立即数不等于零时有效。*nzimm*=0 的代码点被保留；余下的 *rd*=`x0` 的代码点是 HINT；余下的 *rd*=`x2` 的代码点对应于 C.ADDI16SP 指令。

### 整数寄存器-立即数操作

这些整数寄存器-立即数操作以 CI 格式编码，并实施在整数寄存器和 6 位立即数上的操作。




| 3 | 1 | 5 | 5 | 2 |
|---|---|---|---|---|
| C.ADDI | nzimm[5] | dest≠0 | nzimm[4:0] | C1 |
| C.ADDIW | imm[5] | dest≠0 | imm[4:0] | C1 |
| C.ADDI16SP | nzimm[9] | 2 | nzimm[4\|6\|8:7\|5] | C1 |

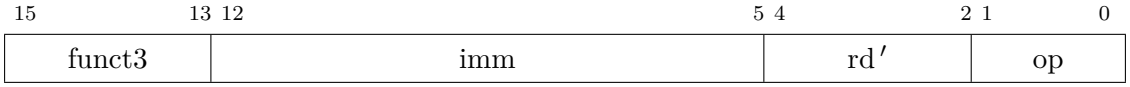
C.ADDI 把非零的符号扩展的 6 位立即数加到寄存器 *rd* 中的值，然后把结果写到 *rd*。C.ADDI 扩展到 `addi rd, rd, nzimm`。C.ADDI 只有当 `addi rd, rd, nzimm`≠`x0` 且 *nzimm*≠0 时有效。*rd*=`x0` 的代码点编码了 C.NOP 指令；余下的 *nzimm*=0 的代码点编码了 HINT。

C.ADDIW 是一个 RV64C/RV128C 独有的指令，它执行相同的计算，但是产生一个 32 位的结果，然后把结果符号扩展到 64 位。C.ADDIW 扩展到 `addiw rd, rd, imm`。对于 C.ADDIW，立即数可以是零，这对应于 `sext.w rd`。C.ADDIW 只有在 *rd*≠`x0` 时有效；*rd*=`x0` 的代码点被保留。

C.ADDI16SP 与 C.LUI 共享操作码，但是有一个目的域 `x2`。C.ADDI16SP 把非零的符号扩展的 6 位立即数加到栈指针中的值（`sp`=`x2`），那里立即数被缩放来代表范围 $(-512, 496)$ 中的 16 的倍数。C.ADDI16SP 被用于调整过程序言和结语中的栈指针。它扩展到 `addi x2, x2, nzimm`。C.ADDI16SP 只有当 *nzimm*≠0 时有效；*nzimm*=0 的代码点被保留。

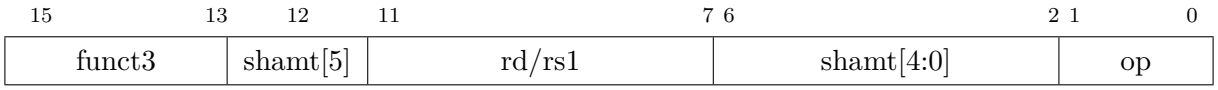
---

在标准的 *RISC-V* 调用约定中，栈指针 `sp` 总是 *16* 位对齐的。




| 3 | 8 | 3 | 2 |
|---|---|---|---|
| C.ADDI4SPN | nzuimm[5:4\|9:6\|2\|3] | dest | C0 |

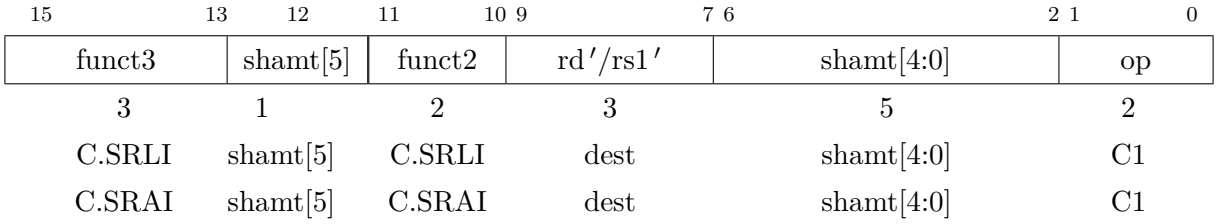
C.ADDI4SPN 是一个 CIW 格式的指令，它把一个零扩展的非零立即数，扩大 4 倍，加到栈指针 `x2` 上，并把结果写到 `rd′`。这个指令被用于生成指向栈分配变量的指针，且扩展到 `addi rd′, x2, nzuimm`。C.ADDI4SPN 只在 *nzuimm*≠0 时生效；*nzuimm*=0 的代码点被保留。




| 3 | 1 | 5 | 5 | 2 |
|---|---|---|---|---|
| C.SLLI | shamt[5] | dest≠0 | shamt[4:0] | C2 |

SLLI 是一个 CI 格式的指令，它对寄存器 *rd* 中的值执行逻辑左移，然后把结果写到 *rd*。移位的数目被编码在 *shamt* 域之中。对于 RV128C，移位数目零被用于编码 64 的移位。C.SLLI 扩展到 `slli rd, rd, shamt`；但 `shamt=0` 的 RV128C 除外，它扩展到 `slli rd, rd, 64`。

对于 RV32C，*shamt[5]* 必须是零；*shamt[5]*=1 的代码点被指定用于自定义扩展。对于 RV32C 和 RV64C，移位的数目必须是非零的；*shamt*=0 的代码点是 HINT。对于所有的基础 ISA，除了 RV32C 中那些 *shamt[5]*=1 的之外，*rd*=`x0` 的代码点都是 HINT。



| 15–13 | 12 | 11–10 | 9–7 | 6–2 | 1–0 |
|---|---|---|---|---|---|
| funct3 | shamt[5] | funct2 | rd′/rs1′ | shamt[4:0] | op |
| 3 | 1 | 2 | 3 | 5 | 2 |
| C.SRLI | shamt[5] | C.SRLI | dest | shamt[4:0] | C1 |
| C.SRAI | shamt[5] | C.SRAI | dest | shamt[4:0] | C1 |

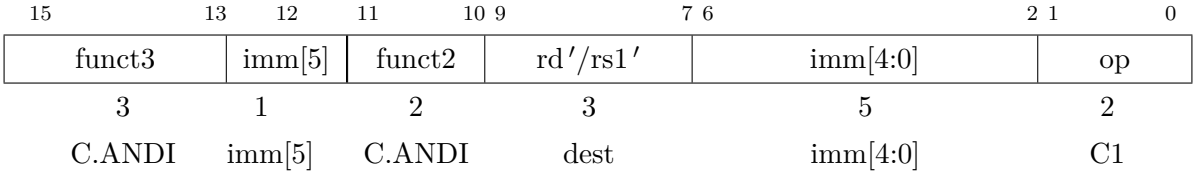
C.SRLI 是一个 CB 格式的指令，它对寄存器 *rd′* 中的值实施逻辑右移，然后把结果写到 *rd′*。移位的数目被编码在 *shamt* 域之中。对于 RV128C，移位数目零被用于编码 64 的移位。甚至，对于 RV128C，移位数目被符号扩展，并因此合法的移位数目是 1 - 13，64，和 96 - 127。C.SRLI 扩展到 `srli rd′, rd′, shamt`；但 `shamt=0` 的 RV128C 除外，它扩展到 `srli rd′, rd′, 64`。

对于 RV32C，*shamt[5]* 必须是零；*shamt[5]*=1 的代码点被指定用于自定义扩展。对于 RV32C 和 RV64C，移位数目必须是非零的；*shamt*=0 的代码点是 HINT。

C.SRAI 的定义与 C.SRLI 类似，但是执行的是算数右移。C.SRAI 扩展到 `srai rd′, rd′, shamt`。

---

*左移通常比右移更加频繁，因为左移被频繁用于放缩地址的值。右移因此被赋予较少的编码空间，并被放置在一个编码象限中，那里所有其它的立即数都是被符号扩展的。对于 RV128，作出该决策是为了让 6 位移位数目立即数也被符号扩展。除了减少解码的复杂度，我们相信 96 -127 的右移数目比 64 - 95 的数目更加有用，以允许提取位于 128 位地址指针的高部分中的标签。我们注意到 RV128C 将不会与 RV32C 和 RV64C 被冻结在同一点，以允许评估 128 位地址空间代码的常见用法。*



| 15–13 | 12 | 11–10 | 9–7 | 6–2 | 1–0 |
|---|---|---|---|---|---|
| funct3 | imm[5] | funct2 | rd′/rs1′ | imm[4:0] | op |
| 3 | 1 | 2 | 3 | 5 | 2 |
| C.ANDI | imm[5] | C.ANDI | dest | imm[4:0] | C1 |

C.ANDI 是一个 CB 格式的指令，它计算寄存器 *rd′* 中的值与符号扩展的 6 位立即数的按位 AND，然后把结果写到 *rd′*。C.ANDI 扩展到 `andi rd′, rd′, imm`。

### 整数寄存器-寄存器操作

### Integer Register-Register Operations

| 15 12 | 11 7 | 6 2 | 1 0 |
|---|---|---|---|
| funct4 | rd/rs1 | rs2 | op |
| 4 | 5 | 5 | 2 |
| C.MV | dest≠0 | src≠0 | C2 |
| C.ADD | dest≠0 | src≠0 | C2 |

这些指令使用 CR 格式。

C.MV 把寄存器 *rs2* 中的值复制到寄存器 *rd* 中。C.MV 扩展到 `add rd, x0, rs2`。C.MV 只在 rs2 = x0 时有效；*rs2*=`x0` 的代码点对应于 C.JR 指令。*rs2*≠`x0` 和 *rd*=`x0` 的代码点是 HINT。

---

*C.MV 扩展到与典型的 MV 伪指令（其使用 ADDI）不同的指令。专门处理 MV 的实现，例如，使用寄存器重命名的硬件，可能会发现把 C.MV 扩展到 MV 而不是 ADD 会更加方便，只需轻微的额外的硬件开销。*

C.ADD 把寄存器 *rd* 和 *rs2* 中的值相加，并把结果写到寄存器 *rd*。C.ADD 扩展到 `add rd, rd, rs2`。C.ADD 只在 *rs2*≠`x0` 时有效；r*s2*=`x0` 的代码点对应于 C.JALR 和 C.EBREAK 指令。*rs2*≠`x00` 和 *rd*=`x0` 的代码点是 HINT。

| 15 10 | 9 7 | 6 5 | 4 2 | 1 0 |
|---|---|---|---|---|
| funct6 | rd′/rs1′ | funct2 | rs2′ | op |
| 6 | 3 | 2 | 3 | 2 |
| C.AND | dest | C.AND | src | C1 |
| C.OR | dest | C.OR | src | C1 |
| C.XOR | dest | C.XOR | src | C1 |
| C.SUB | dest | C.SUB | src | C1 |
| C.ADDW | dest | C.ADDW | src | C1 |
| C.SUBW | dest | C.SUBW | src | C1 |

这些指令使用 CA 格式。

C.AND 计算寄存器 *rd′* 和 *rs2′* 中的值的按位 AND，然后把结果写到寄存器 *rd′*。C.AND 扩展到 `and rd′, rd′, rs2′`。

C.OR 计算寄存器 *rd′* 和 *rs2′* 中的值的按位 OR，然后把结果写到寄存器 *rd′*。C.OR 扩展到 `or rd′, rd′, rs2′`。

C.XOR 计算寄存器 *rd′* 和 *rs2′* 中的值的按位 XOR，然后把结果写到寄存器 *rd′*。C.XOR 扩展到 `xor rd′, rd′, rs2′`。

C.SUB 从寄存器 *rd′* 中的值中减去寄存器 *rs2′* 中的值，然后把结果写到寄存器 *rd′*。C.SUB 扩展到 `sub rd′, rd′, rs2′`。

C.ADDW 是一个 RV64C/4V128C 独有的指令，它把寄存器 *rd′* 和 *rs2′* 中的值相加，然后在把结果写到寄存器 *rd′* 之前，对和的低 32 位进行符号扩展。C.ADDW 扩展到 `addw rd′, rd′, rs2′`。

C.SUBW 是一个 RV64C/RV128 独有的指令，它从寄存器 *rd′* 中的值中减去寄存器 *rs2′* 中的值，然后在把结果写到寄存器 *rd′* 之前，对差的低 32 位进行符号扩展。C.SUBW 扩展到 `subw rd′, rd′, rs2′`。

---

这组的六个指令虽然不会各自提供（对资源的）大量节省，但是不会占据太多的编码空间，而且易于实现；并且作为一个组，在静态和动态压缩方面提供了值得改进的地方。

### Defined Illegal Instruction

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

所有位都是零的 16 位指令被永久性地保留为一个非法指令。

---

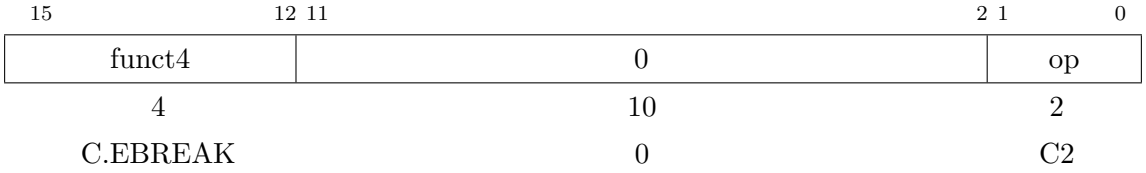
我们把全零指令保留为非法指令，以帮助对尝试执行内存空间中以零结尾的、或者不存在的部分进行陷入。在任何非标准的扩展中，全零的值都应当被重新定义。类似地，我们保留了所有位都设为 *1* 的指令（对应于 *RISC-V* 可变长度编码策略中的非常长的指令）作为非法指令，以捕获另一些常见的在不存在的内存区域中的值。

### NOP 指令

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

C.NOP 是一个 CI 格式的指令，除了提升 pc 和增加任何适用的性能计数器，它不改变任何用户可见的状态。C.NOP 扩展到 nop。C.NOP 只有在 *imm*=0 时有效；*imm*≠0 的代码点编码了 HINT。

### 断点指令




调试器可以使用 C.EBREAK 指令，它扩展到 ebreak，以造成控制被转移回调试环境。C.EBREAK 与 C.ADD 共享操作码，但是 *rd* 和 *rs2* 都是零，因此也可以使用 CR 格式。

## 18.6 C 指令在 LR/SC 序列中的使用

在支持 C 扩展的实现上，在受限的 LR/SC 序列内部允许 I 指令的压缩形式，就像 10.3节中描述的那样，也允许在受限的 LR/SC 序列中使用。

---

这意味着，任何声称同时支持 *A* 扩展和 *C* 扩展的实现都必须确保：包含有效 *C* 指令的 *LR/SC* 序列将最终完成。

## 18.7 “提示”指令

RVC 编码空间的一部分被保留用于微架构 HINT。像在 RV32I 基础 ISA 中的 HINT（见 2.9节），这些指令除了增加 pc 和任何适用的性能计数器，不修改任何架构状态。在实现上，HINT 作为 no-op 执行而忽略它们。

RVC HINT 被编码为不修改架构状态的运算指令，或者是因为 *rd*=x0（例如，C.ADD *x0*, *t0*），或者是因为 *rd* 被它自己的拷贝所覆写（例如，C.ADDI *t0*, 0）。

---

选择这样的 *HINT* 编码，使得简单的实现可以忽略全部的 *HINT*，代替为把 *HINT* 作为一个恰好不会改变架构状态的常规运算指令来执行。

没有必要把 RVC HINT 扩展到它们对应的 RVI HINT。例如，C.ADD *x0*, *a0*可能不会编码为与ADD *x0*, *x0*, *a0*相同的 HINT。

不需要把 *RVC HINT* 扩展到 *RVI HINT* 的主要原因是，*HINT* 不可能以与底层运算指令相同的方式被压缩。并且，解耦 *RVC* 和 *RVI HINT* 的映射可以使稀缺的 *RVC HINT* 空间被分配给最常用的 *HINT*，特别地，分配给适用于宏操作融合的 *HINT*。

表 18.3列出了所有的 RVC HINT 代码点。对于 RV32C，78% 的 HINT 空间被保留给标准 HINT。余下的 HINT 空间被指定给自定义 HINT：永远不会有标准 HINT 将被定义在这个子空间中

| 指令 | 约束 | 代码点 | 用途 |
|---|---|---|---|
| C.NOP | *nzimm*≠0 | 63 | 保留供未来标准使用 |
| C.ADDI | *rd*≠x0, *nzimm*=0 | 31 | |
| C.LI | *rd*=x0 | 64 | |
| C.LUI | *rd*=x0, *nzimm*≠0 | 63 | |
| C.MV | *rd*=x0, *rs2*≠x0 | 31 | |
| C.ADD | *rd*=x0, *rs2*≠x0, *rs2*≠x2–x5 | 27 | |
| C.ADD | *rd*=x0, *rs2*=x2–x5 | 4 | (*rs2*=x2) C.NTL.P1<br>(*rs2*=x3) C.NTL.PALL<br>(*rs2*=x4) C.NTL.S1<br>(*rs2*=x5) C.NTL.ALL |
| C.SLLI | *rd*=x0, *nzimm*≠0 | 31 (RV32)<br>63 (RV64/128) | 指定为自定义使用 |
| C.SLLI64 | *rd*=x0 | 1 | |
| C.SLLI64 | *rd*≠x0, RV32 和 RV64 独有 | 31 | |
| C.SRLI64 | RV32 和 RV64 独有 | 8 | |
| C.SRAI64 | RV32 和 RV64 独有 | 8 | |

表 18.3: RVC HINT 指令。

## 18.8 RVC 指令集列表

表 18.4显示了 RVC 主操作码的映射。表的每一行对应于编码空间的一个象限。最后一个象限，其设置了两个最小有效位，对应于宽度超过 16 位的指令，包括那些在基础 ISA 中的指令。一些指令只对特定的操作数有效；当无效的时候，它们被标记为 *RES* 来表示该操作码被保留用于未来的标准扩展；或者标记为 *Custom* 来表示该操作码被指定用于自定义扩展；或者标记为 *HINT* 来表示该操作码被保留用于微架构提示（见 18.7节）。

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

表 18.4: RVC 操作码映射

表 18.5 - 18.7 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 | 非法指令 |
| 000 | nzuimm[5:4\|9:6\|2\|3] | | | rd′ | 00 | C.ADDI4SPN *(RES, nzuimm=0)* |
| 001 | uimm[5:3] | rs1′ | uimm[7:6] | rd′ | 00 | C.FLD *(RV32/64)* |
| 001 | uimm[5:4\|8] | rs1′ | uimm[7:6] | rd′ | 00 | C.LQ *(RV128)* |
| 010 | uimm[5:3] | rs1′ | uimm[2\|6] | rd′ | 00 | C.LW |
| 011 | uimm[5:3] | rs1′ | uimm[2\|6] | rd′ | 00 | C.FLW *(RV32)* |
| 011 | uimm[5:3] | rs1′ | uimm[7:6] | rd′ | 00 | C.LD *(RV64/128)* |
| 100 | — | | | | 00 | 保留 |
| 101 | uimm[5:3] | rs1′ | uimm[7:6] | rs2′ | 00 | C.FSD *(RV32/64)* |
| 101 | uimm[5:4\|8] | rs1′ | uimm[7:6] | rs2′ | 00 | C.SQ *(RV128)* |
| 110 | uimm[5:3] | rs1′ | uimm[2\|6] | rs2′ | 00 | C.SW |
| 111 | uimm[5:3] | rs1′ | uimm[2\|6] | rs2′ | 00 | C.FSW *(RV32)* |
| 111 | uimm[5:3] | rs1′ | uimm[7:6] | rs2′ | 00 | C.SD *(RV64/128)* |

表 18.5: RVC 的指令列表，第 0 象限。

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

表 18.6: RVC 的指令列表，第 1 象限。

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

表 18.7: RVC 的指令列表，第 2 象限。

# 第十九章　用于位操作的“B”标准扩展（0.0 版本）

这章是为未来的提供位操作指令的标准扩展占位的，包括了插入、提取和测试位域的指令，和用于旋转、漏斗型移位、位与字节置换的指令。

---

尽管位操作指令在一些应用领域（特别是当处理外部打包的数据结构时）非常有效，我们仍然把它们排除在基础 *ISA* 之外，因为它们不是在所有领域中都有用的，并且为了支持所有所需的操作数，会增加额外的复杂度或指令格式。

我们预计 *B* 扩展将成为基础 *30* 位指令空间中编码的一个棕色地带，等待着重新的发掘。

# 第二十章　用于动态翻译语言的“J”标准扩展（0.0版本）

这章是为未来的支持动态翻译语言的标准扩展占位的。

---

许多流行的语言通常通过动态翻译实现，包括 *Java* 和 *Javascript*。这些语言可以从对动态检查和垃圾回收的额外的 *ISA* 支持中获益。

# 第二十一章　用于打包 SIMD 指令的“P”标准扩展 (0.2 版本)

---

第五次 *RISC-V* 研讨会的讨论表示，希望放弃这个用于浮点寄存器的打包 *SIMD* 的提案，转而支持在 *V* 扩展上对大型浮点 *SIMD* 操作的标准化。然而，在小型 *RISC-V* 实现的整数寄存器中，还存在对打包 *SIMD* 定点操作的使用兴趣。一个任务组正在为了定义新的 *P* 扩展而工作着。

# 第二十二章　用于向量操作的“V”标准扩展（0.7 版本）

当前的工作组草案被寄放在了 https://github.com/riscv/riscv-v-spec。

---

基础向量扩展意图在 32 位指令编码空间中为数据并行执行提供通用的支持，而后续的向量扩展会针对特定领域支持更丰富的功能性。

# 第二十三章　用于非对齐原子的“Zam”标准扩展 (0.1 版本)

本章定义了“Zam”扩展，它通过对未对齐的原子内存操作（AMO）进行标准化支持，扩展了“A”扩展。在实现了“Zam”的平台，未对齐的 AMO 只需要针对相同地址和相同尺寸的其它访问（包括非原子的加载和存储）进行原子化地执行。更确切地说，实现了“Zam”的执行环境服从下列公理：

**未对齐原子的原子性公理**　如果 $r$ 和 $w$ 是来自硬件线程 $h$ 的成对的未对齐的加载和存储指令，它们具有相同的地址和相同的尺寸，那么以全局内存次序，在 $r$ 和 $w$ 生成的内存操作之间，不会再有某个存储指令 $s$ 生成的存储操作——其中 $s$ 满足：来自除了 $h$ 之外的硬件线程，并且与 $r$ 和 $w$ 具有相同地址和相同尺寸。并且，以全局内存次序，在 $r$ 或者 $w$ 生成的两个内存操作之间，不会再有某个加载指令 $l$ 生成的加载操作——其中 $l$ 满足：来自除了 $h$ 之外的硬件线程，并且与 $r$ 和 $w$ 具有相同地址和相同尺寸。

原子性的这个受限制的形式试图在应用的需求（其需要支持未对齐原子）与实现的能力（确实提供必要程度的原子性）之间进行平衡。

在“Zam”下，对齐的指令继续按照它们在 RVWMO 下的行为正常工作。

---

“*Zam*”的意图是，它可以用两种方式实现：

1. 对于相关的地址和尺寸的原子未对齐访问，在原生支持该访问的硬件上（例如，对于在单一缓存行中进行的未对齐访问）：通过简单地遵循与对齐的 *AMO* 所应用的相同的规则来实现。
2. 对于相关的地址和尺寸的未对齐访问，在缺少对其原生支持的硬件上：通过用该地址和尺寸陷入所有指令（包括加载指令），并在一个 *mutex*（它是一个给定了内存地址和访问尺寸的函数）中执行它们（通过任意数目的内存操作）。可以通过把它们分割为独立的加载和存储指令来模拟 *AMO*，但是所有保留的程序次序规则（例如，传入和传出的语法依赖项）的行为必须像 *AMO* 仍然是一个单一的内存操作时一样。

# 第二十四章 用于在整数寄存器中使用浮点的“Zfinx”、“Zdinx”、“Zhinx”、“Zhinxmin”标准扩展(1.0 版本)

本章定义了“Zfinx”扩展（发音为“z-f-in-x”），为单精度浮点指令提供了与标准浮点 F 扩展类似的指令，但是操作在 `x` 寄存器而不是 `f` 寄存器上。本章还定义了“Zdinx”、“Zhinx”和“Zhinxmin”扩展，为其他浮点精度提供了类似的指令。

---

*F 扩展使用单独的 `f` 寄存器进行浮点计算，以减少寄存器压力并为宽超标量简化寄存器文件端口的提供。然而，额外的128 B架构状态增加了最小实现成本。通过消除 `f` 寄存器，Zfinx 扩展极大地减少了支持带有浮点指令集的简单 RISC-V 实现的成本。Zfinx 也降低了上下文切换成本。*

*一般来说，假定 F 扩展存在的软件与假定 Zfinx 扩展存在的软件是不兼容的，反之亦然。*

除了转移指令 FLW、FSW、FMV.W.X、FMV.X.W、C.FLW[SP] 和 C.FSW[SP] 外，Zfinx 扩展添加了 F 扩展所增加的所有指令。

---

*Zfinx 软件使用整数的加载与存储在内存之间传递浮点值。在寄存器之间的传输既可以使用整数算术指令，也可以使用浮点符号注入指令。*

这些 F 扩展指令的 Zfinx 变体具有相同的语义，只是当这样一条指令将访问 `f` 寄存器时，改为访问具有相同编号的 `x` 寄存器。

## 24.1 处理更窄的值

宽度$w$ < XLEN bits位的浮点操作数占据 `x` 寄存器的位$w$-1:0。对于 $w$ 位的浮点操作会忽略位XLEN-1:$w$。

产生$w$ < XLEN-bit位结果的浮点操作用位$w$-1（符号位）填充位XLEN-1:$w$。

f 寄存器中采用的 *NaN* 装箱策略的设计足够支持重新编码的浮点格式。不过，由于浮点操作数和整数操作数占据相同的寄存器，重新编码对于 *Zfinx* 不太实用。因此，减少了对 *NaN* 装箱的需求。

当符号扩展的 *32* 位浮点数保存在 *RV64* 的 x 寄存器时，匹配现有的 *RV64* 调用约定，这需要所有的 *32* 位类型在传递到 *x* 寄存器、或从 x 寄存器返回时，都被符号扩展。为了使架构更加规范，我们在 *RV32* 和 *RV64* 中都把这个式样扩展到了 *16* 位浮点数。

## 24.2 Zdinx

Zdinx 扩展提供了类似于双精度浮点的指令。Zdinx 扩展需要 Zfinx 扩展。

除了转移指令 FLD、FSD、FMV.D.X、C.FLD[SP] 和 C.FSD[SP]，Zdinx 扩展添加了 D 扩展增加的所有指令。

这些 D 扩展指令的 Zdinx 变体具有相同的语义，只是当这样一条指令将访问 f 寄存器时，改为访问具有相同编号的 x 寄存器。

## 24.3 处理更宽的值

RV32Zdinx 中的双精度操作数保存在对齐的 x 寄存器对中，也就是说，寄存器数量必须是偶数。为双宽度浮点操作数使用未对齐的（奇数数量的）寄存器是保留的。

不论字节序如何，较低编号的寄存器都保存低序位，而较高编号的寄存器保存高序位：例如，RV32Zdinx 中，双精度操作数的位 31:0 可能保存在寄存器 x14 中，同时该操作数的位 63:32 保存在 x15 中。

当一个双宽度浮点结果写入 x0 时，整个写入是无效的：例如，对于 RV32Zdinx，向 x0 写入一个双精度结果并不会导致 x1 被写入。

当 x0 被用作一个双宽度浮点操作数时，完整的操作数是零——即，x1 是未访问的。

没有提供按对加载和按对存储的指令，所以在 *RV32Zdinx* 中从内存或向内存转移双精度操作数需要两次加载或存储。然而，寄存器的移动只需要一个单独的 *FSGNJ.D* 指令。

## 24.4 Zhinx

Zhinx 扩展提供了类似于半精度浮点的指令。Zhinx 扩展需要 Zfinx 扩展。

除了转移指令 FLH、FSH、FMV.H.X 和 FMV.X.H 外，Zhinx 扩展添加了所有 Zfh 扩展增加的指令。

这些 Zfh 扩展指令的 Zhinx 变体具有相同的语义，只是在这样一条指令将访问 `f` 寄存器时，改为访问具有相同编号的 `x` 寄存器。

## 24.5 Zhinxmin

Zhinxmin 扩展提供了对操作在 `x` 寄存器上的 16 位半精度浮点指令的最小限度的支持。Zhinxmin 扩展需要 Zfinx 扩展。

Zhinxmin 扩展包含来自 Zhinx 扩展的下列指令：FCVT.S.H 和 FCVT.H.S。如果 Zdinx 扩展存在，也包含了 FCVT.D.H 和 FCVT.H.D 指令。

---

在未来，可能定义与 *RV32Zdinx* 类似的 *RV64Zqinx* 四精度扩展。*RV32Zqinx* 扩展也可能被定义，但是需要四寄存器组（译者注：指每组包含共同工作的四个寄存器，原文 *quad-register groups*）。

## 24.6 特权架构的影响

在第二卷定义的标准特权架构中，如果实现了 Zfinx 扩展，`mstatus` 域 FS 被硬布线为零，并且 FS 不再影响浮点指令或 `fcsr` 访问的陷入行为。

当实现了 Zfinx 扩展时，`misa` 位 F、D 和 Q 被硬布线为零。

---

未来的发现机制可能被用于探查 *Zfinx*、*Zhinx* 和 *Zdinx* 扩展的存在性。

# 第二十五章　用于全存储排序的“Ztso”标准扩展 (0.1 版本)

本章定义了用于 RISC-V 全存储排序（RVTSO）内存一致性模型的“Ztso 扩展”。RVTSO 被定义为与 RVWMO（定义在 17.1章中）有差异的部分。

---

*Ztso 扩展旨在帮助原本为带有 TSO 模型的架构（例如 x86 或某些版本的 SPARC）写的代码进行移植，而这两种代码都默认使用 TSO。它也支持那些固有地提供 RVTSO 行为并希望将此事实暴露给软件的实现。*

RVTSO 对 RVWMO 做了如下调整：

- 所有的加载操作的行为如同它们有 acquire-RCpc 注释一样
- 所有的存储操作的行为如同它们有 release-RCpc 注释一样
- 所有的 AMO 行为如同它们同时有 acquire-RCsc 和 release-RCsc 注释一样

---

*这些规则让除了4-7之外的所有 PPO 规则变得多余。它们也让任何没有同时设置了 PW 和 SR 的非 I/O 屏障变得多余。最终，它们也暗示了，不会有内存操作将被重新排序而在任何方向上超越一个 AMO。*

*在 RVTSO 的上下文中，就像是 RVWMO 的情况那样，通过 PPO 规则5-7简明而完整地定义了存储次序。在这两个内存模型中，都是加载值公理使得硬件线程能够将一个来自它的存储缓冲区的值发送到一个后续的（以程序次序）加载——那就是说，存储可以在它们对其它硬件线程可见之前，被本地发送。*

此外，如果实现了 Ztso 扩展，则 V 扩展和 Zve 系列扩展中的向量内存指令在指令级遵循 RVTSO。Ztso 扩展不强化指令内部元素访问的顺序。

尽管事实上 Ztso 没有向 ISA 添加新的指令，假定 RVTSO 写出的代码也将不能正确运行在不支持 Ztso 的实现上。编译的二进制只能运行在 Ztso 下，这应当通过二进制中的一个标志被表示出来，使得没有实现 Ztso 的平台可以简单地拒绝运行它们。

# 第二十六章　RV32/64G 指令集列表

RISC-V 项目的一个目标是，将它作为一个稳定的软件开发目标来使用。为了这个目的，我们定义了一个基础 ISA（RV32I 或 RV64I）加上选择的标准扩展（IMAFD、Zicsr、Zifencei）的组合，作为一个“通用目的”的 ISA，而且，我们使用缩写 G 来表示指令集扩展的 IMAFDZicsr Zifencei 组合。本章展示了为 RV32G 和 RV64G 列出的操作码映射和指令集。

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

表 26.1: RISCV-V 基础操作码映射，instinst[1:0]=11

表 26.1显示了 RVG 的主要操作码的映射。设置了 3 或更低位的主要操作码被保留用于超过 32 位的指令长度。标记为保留的的操作码应当被避免用于自定义的指令集扩展，因为它们可能会用在未来的标准扩展中。标记为自定义− *0* 和自定义− *1* 的主操作码应当被避免用于未来的标准扩展，并且建议用于具有基础 32 位指令格式的自定义指令集扩展。标记为自定义− *2/rv128* 和自定义− *3/rv128* 的操作码被保留，以供 RV128 未来使用，但如果不是 RV128，那么它们将避免被用于标准扩展，并因此也可以被用于 RV32 和 RV64 中的自定义指令集扩展。

我们相信 RV32G 和 RV64G 为大量通用目的计算提供了简单但完整的指令集。可以添加可选的在 十八章中描述的压缩指令集（形成 RV32GC 和 RV64GC）来改善性能、代码尺寸和能量效率，尽管这会带来额外的硬件复杂度。

随着我们的脚步超越了 IMAFDC，走进进一步的指令集扩展，添加的指令更加倾向于特定领域，而只对严格的某类应用（例如，多媒体应用或者安全应用）提供收益。不像大多数商业化的 ISA，RISC-V ISA 的设计将基础 ISA 和广泛可用的标准扩展与这些更特定化的额外部分清晰地分离开。第 二十七章对于向 RISC-V ISA 添加扩展的方法进行了更加广泛的讨论。

| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | |
|---|---|---|---|---|---|---|---|
| funct7 | | rs2 | rs1 | funct3 | rd | opcode | R-type |
| imm[11:0] | | | rs1 | funct3 | rd | opcode | I-type |
| imm[11:5] | | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type |
| imm[12\|10:5] | | rs2 | rs1 | funct3 | imm[4:1\|11] | opcode | B-type |
| imm[31:12] | | | | | rd | opcode | U-type |
| imm[20\|10:1\|11\|19:12] | | | | | rd | opcode | J-type |

**RV32I 基础指令集**

| | | | | | |
|---|---|---|---|---|---|
| imm[31:12] | | | rd | 0110111 | LUI |
| imm[31:12] | | | rd | 0010111 | AUIPC |
| imm[20\|10:1\|11\|19:12] | | | rd | 1101111 | JAL |
| imm[11:0] | | rs1 | 000 | rd | 1100111 | JALR |
| imm[12\|10:5] | rs2 | rs1 | 000 | imm[4:1\|11] | 1100011 | BEQ |
| imm[12\|10:5] | rs2 | rs1 | 001 | imm[4:1\|11] | 1100011 | BNE |
| imm[12\|10:5] | rs2 | rs1 | 100 | imm[4:1\|11] | 1100011 | BLT |
| imm[12\|10:5] | rs2 | rs1 | 101 | imm[4:1\|11] | 1100011 | BGE |
| imm[12\|10:5] | rs2 | rs1 | 110 | imm[4:1\|11] | 1100011 | BLTU |
| imm[12\|10:5] | rs2 | rs1 | 111 | imm[4:1\|11] | 1100011 | BGEU |
| imm[11:0] | | rs1 | 000 | rd | 0000011 | LB |
| imm[11:0] | | rs1 | 001 | rd | 0000011 | LH |
| imm[11:0] | | rs1 | 010 | rd | 0000011 | LW |
| imm[11:0] | | rs1 | 100 | rd | 0000011 | LBU |
| imm[11:0] | | rs1 | 101 | rd | 0000011 | LHU |
| imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 | SB |
| imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 | SH |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |
| imm[11:0] | | rs1 | 000 | rd | 0010011 | ADDI |
| imm[11:0] | | rs1 | 010 | rd | 0010011 | SLTI |
| imm[11:0] | | rs1 | 011 | rd | 0010011 | SLTIU |
| imm[11:0] | | rs1 | 100 | rd | 0010011 | XORI |
| imm[11:0] | | rs1 | 110 | rd | 0010011 | ORI |
| imm[11:0] | | rs1 | 111 | rd | 0010011 | ANDI |
| 0000000 | shamt | rs1 | 001 | rd | 0010011 | SLLI |
| 0000000 | shamt | rs1 | 101 | rd | 0010011 | SRLI |
| 0100000 | shamt | rs1 | 101 | rd | 0010011 | SRAI |
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | SUB |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 | SLL |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 | SLT |
| 0000000 | rs2 | rs1 | 011 | rd | 0110011 | SLTU |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | XOR |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 | SRL |
| 0100000 | rs2 | rs1 | 101 | rd | 0110011 | SRA |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 | OR |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | AND |

| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | |
|---|---|---|---|---|---|---|---|
| funct7 | | rs2 | rs1 | funct3 | rd | opcode | R-type |
| imm[11:0] | | | rs1 | funct3 | rd | opcode | I-type |
| imm[11:5] | | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type |

**RV64I 基础指令集（RV32I 之外的部分）**

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

**RV32/RV64 *Zifencei* 标准扩展**

| | | | | | |
|---|---|---|---|---|---|
| imm[11:0] | rs1 | 001 | rd | 0001111 | FENCE.I |

**RV32/RV64 *Zicsr* 标准扩展**

| | | | | | |
|---|---|---|---|---|---|
| csr | rs1 | 001 | rd | 1110011 | CSRRW |
| csr | rs1 | 010 | rd | 1110011 | CSRRS |
| csr | rs1 | 011 | rd | 1110011 | CSRRC |
| csr | uimm | 101 | rd | 1110011 | CSRRWI |
| csr | uimm | 110 | rd | 1110011 | CSRRSI |
| csr | uimm | 111 | rd | 1110011 | CSRRCI |

**RV32M 标准扩展**

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

**RV64M 标准扩展（RV32M 之外的部分）**

| | | | | | | |
|---|---|---|---|---|---|---|
| 0000001 | rs2 | rs1 | 000 | rd | 0111011 | MULW |
| 0000001 | rs2 | rs1 | 100 | rd | 0111011 | DIVW |

| 31 27 | 26 | 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | |
|---|---|---|---|---|---|---|---|---|
| funct7 | | | rs2 | rs1 | funct3 | rd | opcode | R-type |

**RV32A 标准扩展**

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

**RV64A 标准扩展（RV32A 之外的部分）**

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

| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | |
|---|---|---|---|---|---|---|---|
| funct7 | | rs2 | rs1 | funct3 | rd | opcode | R-type |
| rs3 | funct2 | rs2 | rs1 | funct3 | rd | opcode | R4-type |
| imm[11:0] | | | rs1 | funct3 | rd | opcode | I-type |
| imm[11:5] | | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type |

**RV32F 标准扩展**

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

**RV64F 标准扩展（RV32F 之外的部分）**

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

| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | |
|---|---|---|---|---|---|---|---|
| funct7 | | rs2 | rs1 | funct3 | rd | opcode | R-type |
| rs3 | funct2 | rs2 | rs1 | funct3 | rd | opcode | R4-type |
| imm[11:0] | | | rs1 | funct3 | rd | opcode | I-type |
| imm[11:5] | | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type |

**RV32D 标准扩展**

| | | | | | | |
|---|---|---|---|---|---|---|
| imm[11:0] | | | rs1 | 011 | rd | 0000111 | FLD |
| imm[11:5] | | rs2 | rs1 | 011 | imm[4:0] | 0100111 | FSD |
| rs3 | 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.S.D |
| 0100001 | | 00000 | rs1 | rm | rd | 1010011 | FCVT.D.S |
| 1010001 | | rs2 | rs1 | 010 | rd | 1010011 | FEQ.D |
| 1010001 | | rs2 | rs1 | 001 | rd | 1010011 | FLT.D |
| 1010001 | | rs2 | rs1 | 000 | rd | 1010011 | FLE.D |
| 1110001 | | 00000 | rs1 | 001 | rd | 1010011 | FCLASS.D |
| 1100001 | | 00000 | rs1 | rm | rd | 1010011 | FCVT.W.D |
| 1100001 | | 00001 | rs1 | rm | rd | 1010011 | FCVT.WU.D |
| 1101001 | | 00000 | rs1 | rm | rd | 1010011 | FCVT.D.W |
| 1101001 | | 00001 | rs1 | rm | rd | 1010011 | FCVT.D.WU |

**RV64D 标准扩展（RV32D 之外的部分）**

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

| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | |
|---|---|---|---|---|---|---|---|
| funct7 | | rs2 | rs1 | funct3 | rd | opcode | R-type |
| rs3 | funct2 | rs2 | rs1 | funct3 | rd | opcode | R4-type |
| imm[11:0] | | | rs1 | funct3 | rd | opcode | I-type |
| imm[11:5] | | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type |

**RV32Q 标准扩展**

| | | | | | | |
|---|---|---|---|---|---|---|
| imm[11:0] | | | rs1 | 100 | rd | 0000111 | FLQ |
| imm[11:5] | | rs2 | rs1 | 100 | imm[4:0] | 0100111 | FSQ |
| rs3 | 11 | rs2 | rs1 | rm | rd | 1000011 | FMADD.Q |
| rs3 | 11 | rs2 | rs1 | rm | rd | 1000111 | FMSUB.Q |
| rs3 | 11 | rs2 | rs1 | rm | rd | 1001011 | FNMSUB.Q |
| rs3 | 11 | rs2 | rs1 | rm | rd | 1001111 | FNMADD.Q |
| 0000011 | | rs2 | rs1 | rm | rd | 1010011 | FADD.Q |
| 0000111 | | rs2 | rs1 | rm | rd | 1010011 | FSUB.Q |
| 0001011 | | rs2 | rs1 | rm | rd | 1010011 | FMUL.Q |
| 0001111 | | rs2 | rs1 | rm | rd | 1010011 | FDIV.Q |
| 0101111 | | 00000 | rs1 | rm | rd | 1010011 | FSQRT.Q |
| 0010011 | | rs2 | rs1 | 000 | rd | 1010011 | FSGNJ.Q |
| 0010011 | | rs2 | rs1 | 001 | rd | 1010011 | FSGNJN.Q |
| 0010011 | | rs2 | rs1 | 010 | rd | 1010011 | FSGNJX.Q |
| 0010111 | | rs2 | rs1 | 000 | rd | 1010011 | FMIN.Q |
| 0010111 | | rs2 | rs1 | 001 | rd | 1010011 | FMAX.Q |
| 0100000 | | 00011 | rs1 | rm | rd | 1010011 | FCVT.S.Q |
| 0100011 | | 00000 | rs1 | rm | rd | 1010011 | FCVT.Q.S |
| 0100001 | | 00011 | rs1 | rm | rd | 1010011 | FCVT.D.Q |
| 0100011 | | 00001 | rs1 | rm | rd | 1010011 | FCVT.Q.D |
| 1010011 | | rs2 | rs1 | 010 | rd | 1010011 | FEQ.Q |
| 1010011 | | rs2 | rs1 | 001 | rd | 1010011 | FLT.Q |
| 1010011 | | rs2 | rs1 | 000 | rd | 1010011 | FLE.Q |
| 1110011 | | 00000 | rs1 | 001 | rd | 1010011 | FCLASS.Q |
| 1100011 | | 00000 | rs1 | rm | rd | 1010011 | FCVT.W.Q |
| 1100011 | | 00001 | rs1 | rm | rd | 1010011 | FCVT.WU.Q |
| 1101011 | | 00000 | rs1 | rm | rd | 1010011 | FCVT.Q.W |
| 1101011 | | 00001 | rs1 | rm | rd | 1010011 | FCVT.Q.WU |

**RV64Q 标准扩展（RV32Q 之外的部分）**

| | | | | | | |
|---|---|---|---|---|---|---|
| 1100011 | 00010 | rs1 | rm | rd | 1010011 | FCVT.L.Q |
| 1100011 | 00011 | rs1 | rm | rd | 1010011 | FCVT.LU.Q |
| 1101011 | 00010 | rs1 | rm | rd | 1010011 | FCVT.Q.L |
| 1101011 | 00011 | rs1 | rm | rd | 1010011 | FCVT.Q.LU |

| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | |
|---|---|---|---|---|---|---|---|
| funct7 | | rs2 | rs1 | funct3 | rd | opcode | R-type |
| rs3 | funct2 | rs2 | rs1 | funct3 | rd | opcode | R4-type |
| imm[11:0] | | | rs1 | funct3 | rd | opcode | I-type |
| imm[11:5] | | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type |

**RV32Zfh 标准扩展**

| | | | | | | |
|---|---|---|---|---|---|---|
| imm[11:0] | | | rs1 | 001 | rd | 0000111 | FLH |
| imm[11:5] | | rs2 | rs1 | 001 | imm[4:0] | 0100111 | FSH |
| rs3 | 10 | rs2 | rs1 | rm | rd | 1000011 | FMADD.H |
| rs3 | 10 | rs2 | rs1 | rm | rd | 1000111 | FMSUB.H |
| rs3 | 10 | rs2 | rs1 | rm | rd | 1001011 | FNMSUB.H |
| rs3 | 10 | rs2 | rs1 | rm | rd | 1001111 | FNMADD.H |
| 0000010 | | rs2 | rs1 | rm | rd | 1010011 | FADD.H |
| 0000110 | | rs2 | rs1 | rm | rd | 1010011 | FSUB.H |
| 0001010 | | rs2 | rs1 | rm | rd | 1010011 | FMUL.H |
| 0001110 | | rs2 | rs1 | rm | rd | 1010011 | FDIV.H |
| 0101110 | | 00000 | rs1 | rm | rd | 1010011 | FSQRT.H |
| 0010010 | | rs2 | rs1 | 000 | rd | 1010011 | FSGNJ.H |
| 0010010 | | rs2 | rs1 | 001 | rd | 1010011 | FSGNJN.H |
| 0010010 | | rs2 | rs1 | 010 | rd | 1010011 | FSGNJX.H |
| 0010110 | | rs2 | rs1 | 000 | rd | 1010011 | FMIN.H |
| 0010110 | | rs2 | rs1 | 001 | rd | 1010011 | FMAX.H |
| 0100000 | | 00010 | rs1 | rm | rd | 1010011 | FCVT.S.H |
| 0100010 | | 00000 | rs1 | rm | rd | 1010011 | FCVT.H.S |
| 0100001 | | 00010 | rs1 | rm | rd | 1010011 | FCVT.D.H |
| 0100010 | | 00001 | rs1 | rm | rd | 1010011 | FCVT.H.D |
| 0100011 | | 00010 | rs1 | rm | rd | 1010011 | FCVT.Q.H |
| 0100010 | | 00011 | rs1 | rm | rd | 1010011 | FCVT.H.Q |
| 1010010 | | rs2 | rs1 | 010 | rd | 1010011 | FEQ.H |
| 1010010 | | rs2 | rs1 | 001 | rd | 1010011 | FLT.H |
| 1010010 | | rs2 | rs1 | 000 | rd | 1010011 | FLE.H |
| 1110010 | | 00000 | rs1 | 001 | rd | 1010011 | FCLASS.H |
| 1100010 | | 00000 | rs1 | rm | rd | 1010011 | FCVT.W.H |
| 1100010 | | 00001 | rs1 | rm | rd | 1010011 | FCVT.WU.H |
| 1110010 | | 00000 | rs1 | 000 | rd | 1010011 | FMV.X.H |
| 1101010 | | 00000 | rs1 | rm | rd | 1010011 | FCVT.H.W |
| 1101010 | | 00001 | rs1 | rm | rd | 1010011 | FCVT.H.WU |
| 1111010 | | 00000 | rs1 | 000 | rd | 1010011 | FMV.H.X |

**RV64Zfh 标准扩展（RV32Zfh 之外的部分）**

| | | | | | | |
|---|---|---|---|---|---|---|
| 1100010 | | 00010 | rs1 | rm | rd | 1010011 | FCVT.L.H |
| 1100010 | | 00011 | rs1 | rm | rd | 1010011 | FCVT.LU.H |
| 1101010 | | 00010 | rs1 | rm | rd | 1010011 | FCVT.H.L |
| 1101010 | | 00011 | rs1 | rm | rd | 1010011 | FCVT.H.LU |

表 26.3列出了当前被分配了 CSR 地址的 CSR。计时器、计数器，和浮点 CSR 是这个规范中定义的仅有的 CSR。

| 编号 | 权限 | 名称 | 描述 |
|---|---|---|---|
| 浮点控制和状态寄存器 | | | |
| 0x001 | 读/写 | fflags | 浮点增长异常。 |
| 0x002 | 读/写 | frm | 浮点动态舍入模式。 |
| 0x003 | 读/写 | fcsr | 浮点控制和状态寄存器（frm + fflags）。 |
| 计数器和计时器 | | | |
| 0xC00 | 只读 | cycle | 用于 RDCYCLE 指令的周期计数器。 |
| 0xC01 | 只读 | time | 用于 RDTIME 指令的计时器。 |
| 0xC02 | 只读 | instret | 用于 RDINSTRET 指令的指令退场计数器。 |
| 0xC80 | 只读 | cycleh | cycle 的高 32 位，RV32I 专用。 |
| 0xC81 | 只读 | timeh | time 的高 32 位，RV32I 专用。 |
| 0xC82 | 只读 | instreth | instret 的高 32 位，RV32I 专用。 |

表 26.3: RISC-V 控制和状态寄存器（CSR）地址映射。

# 第二十七章　扩充的 RISC-V

除了支持标准通用目的软件开发，RISC-V 的另一个目标是为更加专门的指令集扩展或更加定制的加速器提供一个基础。指令编码空间和可选的可变长度指令编码的设计使得在构建更加定制的处理器时，更容易利用那些用于标准 ISA 工具链的软件开发工作。例如，意图为只使用标准 I 基础的实现持续提供完全的软件支持，也许还有许多非标准的指令集扩展。

这章描述了各种可以扩展基础 RISC-V ISA 的方法，以及管理由独立工作组开发的指令集扩展的策略。本卷只考虑非特权 ISA，尽管相同的方法和术语也被用于第二卷中描述的监管器级别的扩展。

## 27.1　扩展术语

本节定义了一些用于描述 RISC-V 扩展的标准术语。

### 标准扩展 vs 非标准扩展

任何 RISC-V 处理器实现必须支持一个基础整数 ISA（RV32I、RV32E、RV64I 或 RV128I）。此外，一个实现可以支持一个或更多的扩展。我们把扩展划分为两个宽泛的种类：*标准扩展* vs *非标准扩展*。

- 一个标准扩展是，一种通常有用的扩展，它被设计为不与任何其它标志扩展相冲突。当前，在这本手册的其它章节中描述的“MAFDQLCBTPV”，或者是完整的、或者是计划中的标准扩展。
- 一个非标准扩展，可以是高度专用的、并且可以与其它标准或非标准扩展冲突的扩展。我们预计随着时间的推移，将会开发出多种多样的非标准扩展，而其中的某些最终会被提升为标准扩展。

### 指令编码空间和前缀

指令编码空间是一定数目的指令位，在这些指令位中编码了基础 ISA 或 ISA 扩展。RISC-V 支持多种指令长度，但是即使在单一指令长度中，也有各种尺寸的可用编码空间。例如，基础 ISA 被定义在一个 30 位编码空间（32 位指令的位 31 − 2）之中，同时原子扩展“A”容纳在一个 25 位编码空间（位 31 − 7）之中。

我们使用术语“前缀”来指代一个指令编码空间的*右边*的位（因为 RISC-V 中的指令获取是小字节序的，右边的位被存储在更早的内存地址处，因此形成了一个按照指令获取次序的前缀）。标准基础 ISA 编码的前缀是两位“11”域，在 32 位字的位 1 − 0 中，而标准原子扩展“A”的前缀是七位的“0101111”域，保持在代表 AMO 主操作码的 32 位字的位 6 − 0 中。编码格式的一个怪癖是，在 32 位指令格式中，用于编码次要操作码的 3 位的 funct3 域虽然不与主操作码位相接，但是却被认为是 22 位指令空间的前缀的一部分。

尽管一个指令编码空间可以是任何尺寸的，采用一组较小的常见尺寸将简化把独立开发的扩展打包进单一的全局编码中的过程。表 27.1为 RISC-V 给出了建议的尺寸。

| Size | Usage | # 可用的标准指令长度 | | | |
|---|---|---|---|---|---|
| | | 16-bit | 32-bit | 48-bit | 64-bit |
| 14 位 | 压缩 16 位编码的象限 | 3 | | | |
| 22 位 | 基础 32 位编码中的次要操作码 | | $2^8$ | $2^{20}$ | $2^{35}$ |
| 25 位 | 基础 32 位编码中的主要操作码 | | 32 | $2^{17}$ | $2^{32}$ |
| 30 位 | 基础 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 位编码中的主要操作码 | | | 32 | $2^{20}$ |
| 40 位 | 48 位编码的象限 | | | 4 | $2^{17}$ |
| 45 位 | 64 位编码中的子级次要操作码 | | | | $2^{12}$ |
| 48 位 | 64 位编码中的次要操作码 | | | | $2^9$ |
| 52 位 | 64 位编码中的主要操作码 | | | | 32 |

表 27.1: 建议的标准 RISC-V 指令编码空间尺寸。

### 绿色地带扩展 vs 棕色地带扩展

我们使用术语*绿色地带扩展*来描述一个这样的扩展，它在一个新的指令编码空间开始发展，并因此只能在前缀级别引起编码冲突。我们使用术语*棕色地带扩展*来描述一个扩展，它符合在先前定义的指令空间中的现有编码。一个棕色地带扩展必须联系到一个特定的绿色地带父级编码，而对于

相同的绿色地带父级编码，可能有多个棕色地带扩展。例如，基础 ISA 是一个 30 位指令空间的绿色地带编码，同时 FDQ 浮点扩展都是添加到父级基础 ISA 的 30 位编码空间的棕色地带扩展。

注意，我们认为标准 A 扩展具有绿色地带编码，因为它在全 32 位基础指令编码的最左侧的位中定义了一个全新的、先前为空的 25 位编码空间，即使它的标准前缀把它定位在了父基础 ISA 的 30 位编码空间之中。仅仅改变它的一个 7 位前缀可以把 A 扩展移动到一个不同的 30 位编码空间，同时只需要担心前缀级别的冲突，而在编码空间自身之中不会有冲突。

| | 添加状态 | 没有新状态 |
|---|---|---|
| 绿色地带编码 | RV32I(30), RV64I(30) | A(25) |
| 棕色地带编码 | F(I), D(F), Q(D) | M(I) |

表 27.2: 标准指令集扩展的二维特征。

表 27.2显示了置于一种简单的二维分类中的基础和标准扩展。一个轴是该扩展属于绿色地带的还是棕色地带的，而另一个轴是该扩展是否添加了架构的状态。对于绿色地带扩展，括号中给出的是指令编码空间的尺寸。对于棕色地带扩展，括号中给出的是其构建所基于的扩展的名字（绿色地带或者棕色地带）。额外的用户级架构状态通常暗示了对监管器级别系统的改变，或者对标准调用约定的可能的改变。

注意 RV64I 并不被认为是 RV32I 的一个扩展，而是一种不同的完整的基础编码。

### 标准-可兼容全局编码

对于一个确实的 RISC-V 实现，一个 ISA 的完整的或*全局的*编码必须为每个所包含的指令编码空间分配一个唯一的不冲突的前缀。基础扩展和每个标准扩展各拥有一个已分配的标准前缀，以确保它们都可以在全局编码中共存。

*标准-可兼容*全局编码是，一种基础扩展和每个所包含的标准扩展都拥有它们的标准前缀的编码。标准-可兼容全局编码可以含有与所包含的标准扩展不相冲突的非标准扩展。标准-可兼容全局编码也可以为非标准扩展使用标准前缀，如果相关联的标准扩展并没有被包含在全局编码之中。换句话说，一个标准扩展如果被包含在一个标准-可兼容全局编码之中，那么必须使用它的标准前缀，但是否则的话，它的前缀可以自由地重新分配。这些限制允许常见的工具链把任何 RISC-V 标准-可兼容全局编码的标准子集作为目标。

### 保证的非标准编码空间

为了支持专有的自定义扩展的开发，部分编码空间被保证永远不会被标准扩展使用。

## 27.2 RISC-V 扩展设计理念

我们试图通过鼓励扩展开发者们在指令编码空间中操作，以及通过提供工具来为这些分配独有的前缀来将其打包进标准-可兼容全局编码，来支持大量的独立开发的扩展。某些扩展被自然地实现为现有扩展的棕色地带扩充，而将分享分配给它们的父级绿色地带扩展的任何前缀。标准扩展前缀避免了核心功能编码中的虚假的不兼容，同时允许定制更加深奥的扩展。

这种把 RISC-V 扩展重新打包进不同的标准-可兼容全局编码的能力可以有多种使用的方式。

一种使用情况是，开发高度特化的定制加速器，是为了运行来自重要应用领域的内核而设计。这些加速器可能希望丢弃除了基础整数 ISA 之外的所有 ISA，而只加入手头任务所需要的扩展。基础 ISA 被设计为，呈现了关于一个硬件实现的最低需求，而被编码为只使用了 32 位指令编码空间一小部分。

另一种使用情况是，为一种新的类型的指令集扩展构建一个研究原型。研究人员可能不想把工作扩展到实现一个可变长度的指令获取单元，并因此愿意使用一个简单的 32 位定宽指令编码来构建他们的扩展的原型。然而，这个新的扩展可能太大，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的标准扩展，标准-可兼容全局编码可以丢弃不使用的标准扩展，而重用它们的前缀，以把所提出的扩展放置在非标准位置中，来简化研究原型的工程。标准工具将仍然能够把基础扩展和任何存在的标准扩展作为目标，以减少开发时间。一旦指令集扩展被评估和优化过，然后它就可以被打包进一个更大的、可变长度的编码空间而变得可用，以避免与所有标准扩展冲突。

下面的章节描述了使用新指令集扩展开发实现的越来越复杂的策略。这些策略主要是为了用于高度定制的、教育的、或者实验的架构，而不是用于 RISC-V ISA 开发的主线。

## 27.3 定宽 32 位指令格式下的扩展

在这节中，我们对向只支持基础定宽 32 位指令格式的实现添加扩展的内容进行了讨论。

---

我们预计，最简单的定宽 *32* 位编码将在许多受限的加速器和研究原型中变得流行。

### 可用的 30 位指令编码空间

在标准编码中，可用的 30 位指令编码空间中的三个（2 位前缀是 00、01 和 10 的那些）被用于启用可选的压缩指令扩展。然而，如果压缩指令集扩展是不需要的，那么这三个额外的 30 位编码空间就变得可用了。这使 32 位格式中的可用编码空间变成了四倍。

### 可用的 25 位指令编码空间

一个 25 位指令编码空间对应于基础和标准扩展编码中的一个主要操作码。

有四个主要的操作码被明确指定用于自定义扩展（表 26.1），它们中的每个都代表一个 25 位编码空间。

两个保留用于 RV64 的主要操作码（OP-IMM-32 和 OP-32）也可以被只用于 RV32 的非标准扩展。

如果实现不需要浮点，那么七个保留用于标准浮点扩展的主要的操作码（LOAD-FP、STORE-FP、MADD、MSUB、NMSUB、NMADD、OP-FP）可以被重用于非标准扩展。类似地，AMO 主操作码可以被重用，如果不需要标准原子扩展的话。

如果实现不需要超过 32 位长的指令，那么额外的四个主要的操作码是可用的（在表 26.1中被标记为灰色的那些）。

基础 RV32I 编码只使用 11 个主要的操作码和 3 个保留的操作码，留给了扩展 18 个可用的操作码。基础 RV64I 编码只使用 13 个主要的操作码和 3 个保留的操作码，留给了扩展 16 个可用的操作码。

### 可用的 22 位指令编码空间

一个 22 位编码空间对应于基础和标准扩展编码中的一个 funct3 次要操作码空间。一些主要的操作码有一个没有被完全占用的 funct3 域的次要操作码，留下了一些可用的 22 位编码空间。

通常一个主要的操作码在指令余下的位中选择用于编码操作数的格式，并且理想情况下，扩展应当遵循主要的操作码的操作数格式，以简化硬件解码。

### 其它空间

在特定的主要的操作码下可以使用更小的空间，并且不是所有的次要操作码都被完全填满。

## 27.4 添加对齐的 64 位指令扩展

对于基础 32 位定宽指令格式来说，为太大的扩展提供空间的最简单的方法是添加自然对齐的 64 位指令。该实现仍然必须支持 32 位基础指令格式，但是可以要求 64 位指令在 64 位边界对齐，以简化指令获取，必要时使用 32 位 NOP 指令作为对齐的填充。

为了简化标准工具的使用，64 位指令的编码应当像表 1.1中描述的那样。然而，实现可能为 64 位指令选择了一个非标准的指令长度编码，同时为 32 位指令保留了标准编码。例如，如果压缩指令是不需要的，那么 64 位指令可以在指令的前两位中使用一个或更多的零位来编码。

---

我们期待生产指令获取单元的处理器产生器能够自动地处理任何所支持的可变长度指令编码的组合

## 27.5 支持 VLIW 编码

尽管 RISC-V 并不是作为一个纯 VLIW 机器的基础而设计，也可以使用一些替代的方法，将 VLIW 编码作为扩展添加。但是在所有情况中，都必须支持基础 32 位编码，以允许任何标准软件工具的使用。

### 固定尺寸指令组

最简单的方法是，在编码的 VLIW 操作中定义一个单一的、大型的、自然对齐的指令格式（例如，128 位）。在一个传统的 VLIW 中，这个方法往往会浪费指令内存来容纳 NOP，但是一个兼容 RISC-V 的实现也将不得不支持基础 32 位指令，从而限制了将 VLIW 代码尺寸扩张到 VLIW 加速的函数。

### 编码长度组

另一个方法是，使用表 1.1中的标准长度编码来编码并行的指令组，这允许 NOP 被压缩在 VLIW 指令之外。例如，一个 64 位指令可以容纳两个 28 位操作，同时一个 96 位指令可以容纳三个 28 位操作，等等。或者，一个 48 位指令可以容纳一个 42 位操作，同时一个 96 位指令可以容纳两个 42 位操作，等等。

这个方法具有为容纳单一操作的指令保留了基础 ISA 编码的优势，但是劣势在于，需要为 VLIW 指令中的操作、以及更大指令组的未对齐的指令获取，使用新的 28 位或 42 位编码。一个简化方法是，不允许 VLIW 指令跨越特定的微架构的重要边界（例如，缓存行或者虚拟内存页）。

### 固定尺寸指令扎

另一个方法，与 Itanium 类似，是使用一个较大的自然对齐的固定指令扎尺寸（例如，128 位），并行操作组在该指令扎上进行编码。这个方法简化了指令获取，但是把复杂度转移给了组执行引擎。为了保持 RISC-V 的兼容性，将必须仍然支持基础 32 位指令。

### 前缀中的 End-Of-Group 位

上述方法没有一个为 VLIW 指令中的单独操作保留了 RISC-V 编码。然而另一个方法是在定宽 32 位编码中重新赋予两个前缀位新的用途。一个前缀位可以被用于在被设置时指示“组的结束”，同时第二个位可以在被清除时表示正在谓词下执行。由 VLIW 扩展感知不到的工具生成的标准 RISC-V 32 位指令将把两个前缀位都设置为（11），并因此具有正确的语义：每条指令都位于组的结尾并且都不是谓词。

这个方法的主要劣势是，基础 ISA 缺少复杂的谓词支持，而那在一个激进的 VLIW 系统中通常是需要的；并且在标准 30 位编码空间中，很难增加空间以指定更多的谓词寄存器。

# 第二十八章　ISA 扩展命名约定

这章描述了 RISC-V ISA 扩展的命名策略，它被用于简洁地描述一个硬件实现中现有指令的集合，或者被一个应用二进制接口（ABI）所使用的指令的集合。

---

*RISC-V ISA 为了支持多种多样的实现而设计，带有各种实验的指令集扩展。我们已经发现，一个有组织的命名策略对软件工具和文档具有简化作用。*

## 28.1　大小写敏感性

ISA 命名字符串是大小写敏感的。

## 28.2　基础整数 ISA

RISC-V ISA 字符串以 RV32I、RV32E、RV64I 或 RV128I 开始，表示了对于基础整数 ISA 所支持的地址空间尺寸的位数。

## 28.3　指令集扩展的命名

标准 ISA 被赋予了由单个字母组成的命名。例如，最初的四个对于整数基础的标准扩展是：用于整数乘法和除法的“M”，用于原子内存指令的“A”，用于单精度浮点指令的“F”，和用于双精度浮点指令的“D”。任何 RISC-V 指令集的变体可以被简洁地描述为，将基础整数前缀与所包含的扩展的命名的结合，例如，“RV64IMAFD”。

我们也定义了一个缩写“G”来代表“IMAFDZicsr_Zifencei”基础和扩展，因为这是为了表示我们的标准通用目的 ISA。

对 RISC-V ISA 的标准扩展被赋予了其它保留的字母，例如用于四精度浮点的“Q”，或者用于 16 位压缩指令格式的“C”。

有些 ISA 扩展依赖于其它扩展的存在，例如，“D”扩展依赖于“F”扩展，而“F”扩展依赖于“Zicsr”扩展。这些依赖可能隐含在 ISA 命名之中：例如，RV32IF 等价于 RV32IFZicsr，而 RV32ID 等价于 RV32IFD 和 RV32IFDZicsr。

## 28.4 版本号

认识到指令集可能随着时间而扩展或改变，我们在扩展的名字后面编码了扩展的版本号。版本号被划分为主版本号和次版本号，使用“p”分割。如果次版本是“0”，那么“p0”可以从版本字符串中被忽略。主版本号的改变表示了一种向后兼容性的损失，而只改变次版本号必须是向后兼容的。例如，在这本手册的 1.0 发布版本中定义的原始的 64 位标准 ISA 可以被写全为“RV64I1p0M1p0A1p0F1p0D1p0”，而“RV64I1M1A1F1D1”是更简洁的写法。

我们在第二个发布版本中介绍了版本号策略。因此我们把一个标准扩展的默认版本定义为在那个时间的当前版本，例如，“RV32I”等价于“RV32I2”。

## 28.5 着重说明

下划线“_”可以被用于分割 ISA 扩展，以增强可读性，和提供歧义消除，例如“RV32I2_M2_A2”。

因为用于打包 SIMD 的“P”扩展可能会与版本号中的小数点相混淆，所以如果它跟在一个数字后面，那么在它前面必须加下划线。例如，“rv32i2p2”意味着 RV32I 的 2.2 版本，而“rv32i2_p2”意味着带有 P 扩展的 2.0 版本的 RV32I 的 2.0 版本。

## 28.6 附加的标准扩展的命名

标准扩展也可以使用一个“Z”、后面跟着一个按字母顺序排列的名字和一个可选的版本号来命名。例如，“Zifencei”命名了第 三章中描述的指令获取屏障扩展；“Zifencei2”和“Zifencei2p0”描述了相同扩展的 2.0 版本。

跟在“Z”后面的第一个字母通常暗示了相关字母顺序最近的扩展种类，IMAFDQCVH。例如，对于用于未对齐原子的“Zam”扩展，字母“a”表示该扩展与“A”标准扩展相关。如果命名了多个“Z”扩展，它们应当首先按种类排序，然后在每个种类中按字母顺序排序——例如，“Zicsr_Zifencei_Zam”。

使用“Z”前缀的扩展必须使用一个下划线与其它多字母的扩展分割，例如，“RV32IMACZicsr_Zifencei”。

## 28.7 监管器级指令集扩展

标准监管器级指令集扩展在第二卷中定义，但是在命名时使用“S”作为前缀，后面跟着按字母顺序排列的名称和一个可选的版本号。监管器级别的扩展必须通过一个下划线与其他多字母的扩展进行分割。

标准监管器级扩展应当被列在标准非特权扩展之后。如果列出了多个监管器级别的扩展，它们应当按字母顺序排列。

## 28.8 机器级指令集扩展

标准机器级指令集扩展使用三个字母“Zxm”作为前缀。

标准机器级扩展应当被列在标准更低特权级扩展之后。如果列出了多个机器级扩展，它们应当按字母顺序排列。

## 28.9 非标准扩展的命名

非标准扩展使用一个单独的“X”、后面跟着一个按字母顺序排列的名字和一个可选的版本号来命名。例如，“Xhwacha”命名了 Hwacha 向量获取 ISA 扩展；“Xhwacha2”和“Xhwacha2p0”命名了相同扩展的 2.0 版本。

非标准扩展必须被列在所有的标准扩展之后。它们必须通过一条下划线来与其它多字母的扩展分割。例如，一个带有非标准扩展 Argle 和 Bargle 的 ISA 可以被命名为“RV64IZifencei_Xargle_Xbargle”。

如果列出了多个非标准扩展，它们应当按照字母顺序排序。

## 28.10 子集命名约定

表 28.1总结了标准化的扩展命名。

| 子集 | 命名 | 隐含 |
|---|---|---|
| 基础 ISA | | |
| 整数 | I | |
| 约简的整数 | E | |
| 标准非特权扩展 | | |
| 整数乘法和除法 | M | |
| 原子性 | A | |
| 单精度浮点 | F | Zicsr |
| 双精度浮点 | D | F |
| 通用 | G | IMAFDZicsr_Zifencei |
| 四精度浮点 | Q | D |
| 16 位压缩指令 | C | |
| 打包的 SIMD 扩展 | P | |
| 向量扩展 | V | D |
| 监视级扩展 | H | |
| 控制和状态寄存器访问 | Zicsr | |
| 指令获取屏障 | Zifencei | |
| 未对齐原子性 | Zam | A |
| 全存储排序 | Ztso | |
| 标准监管器级扩展 | | |
| 监管器级扩展“def” | Sdef | |
| 标准机器级扩展 | | |
| 机器级扩展“jkl” | Zxmjkl | |
| 非标准扩展 | | |
| 非标准扩展“mno” | Xmno | |

表 28.1: 标准 ISA 扩展命名。该表也定义了扩展的名字在命名字符串中必须出现在的传统次序，表中从上到下的顺序表示命名字符串中从开始到结束的顺序，例如，RV32IMACV 是合法的，而 RV32IMAVC 是不合法的。

# 第二十九章　历史和鸣谢

## 29.1 “为什么要开发一个新的 ISA?”伯克利小组的理由

我们开发 RISC-V 来支持我们自己在研究和教育中的需求，这里，我们组尤其对研究想法的实际硬件实现（自从这个规范的初次编辑以来，我们已经完成了 11 个不同的 RISC-V 的硅制品）、以及为学生们在课堂中的探索提供真实实现（RISC-V 处理器 RTL 设计已经用在了伯克利的多个本科和研究生课程之中）感兴趣。在我们目前的研究中，受到传统晶体管规模的终结所引发的能量受限的驱使，我们特别感兴趣于转向专用化和异构化的加速器。我们希望有一种高度灵活的和可扩展的基础 ISA，围绕它来构建我们的研究工作。

我们被重复问到的一个问题是“为什么要开发一个新的 ISA?”使用一个现有的商业 ISA 的最显而易见的好处是有大而广泛的软件生态系统的支持，无论是开发工具还是移植的应用，都可以被用于研究和教学。其它好处包括存在大量的文档和教程样例。然而，我们使用商业指令集用于研究和教学的经验是，这些好处在实际中是比较小的，并且比不过其劣势：

- **商业 ISA 是有专利的**。除了 SPARC V8，它是一个开放的 IEEE 标准 [3]，大多数商业 ISA 的所有者都小心地保卫着他们的知识产权，而不欢迎自由地提供有竞争力的实现。这对于只使用软件模拟器的学术研究和教学来说还不是太大的问题，但是对于那些希望分享确实的 RTL 实现的团队来说却是主要关注的事情。对于不愿意相信几乎没有源代码的商业 ISA 实现的群体来说，这也是个主要的问题——但是他们被禁止创造他们自己的干净实现。我们不能保证所有的 RISC-V 实现都将免受第三方专利的侵权，但是我们可以保证我们不会尝试起诉 RISC-V 的实现者。
- **商业 ISA 只流行于特定的市场领域**。在编写时，最显而易见的例子是，ARM 架构对服务器空间的支持并不好，而英特尔 x86 架构（或者同样地，几乎其余的所有架构）都不能很好地支持移动空间——尽管英特尔和 ARM 都在尝试进入相互的市场段。另一个例子是 ARC 和 Tensilica，它们提供可扩展的核心，但是聚焦于嵌入式空间。这种市场分割冲淡了支持特定商业 ISA 的好处，因为实际上，软件生态系统只针对特定领域而存在，而不得不为了其它领域而构建。

- **商业 ISA 来来往往**。先前的研究基础设施围绕着商业 ISA 构建，那些商业 ISA 已经不再流行（SPARC、MIPS）或者甚至不再生产了（Alpha）。这些都失去了一个活跃的软件生态系统的好处，而围绕 ISA 和支持工具的持续不断的知识产权问题一直在阻碍着感兴趣的第三方继续支持 ISA 的能力。一个开放的 ISA 可能也会失去流行性，但是任何感兴趣的团体都可以继续使用和开发这个生态系统。

- **流行的商业 ISA 是复杂的**。对于在硬件中的支持常见软件栈和操作系统的级别，居主导的商业 ISA（x86 和 ARM）的实现都是非常复杂的。更糟的是，几乎所有的复杂度都是由于坏的、或者至少是过时的 ISA 设计决策，而不是由于真正的提高效率的特性。

- **只有商业 ISA 自己是不足以带起应用的**。即使我们扩展了实现一个商业 ISA 的工作，这也仍然不足以使 ISA 运行现有的应用。大多数应用需要一个完整的 ABI（应用程序二进制接口）来运行，而不仅仅是用户级 ISA。大多数 ABI 依赖于库，而这又反过来依赖于操作系统的支持。为了运行一个现有的操作系统，需要实现监管器级别的 ISA 和操作系统期望的设备接口。与用户级 ISA 相比，这些通常更加不明确，并且实现起来也要更复杂得多。

- **流行的商业 ISA 并非为了扩展而设计**。居主导的商业 ISA 不会特地为扩展设计，并且因此，随着它们的指令集的增长，增加了相当多的指令编码复杂度。诸如 Tensilica（被 Cadence 收购）和 ARC（被 Synopsys 收购）的公司虽然围绕着可扩展性构建了 ISA 和工具链，但是更加聚焦于嵌入式应用，而不是通用目的的计算系统。

- **修改后的商业 ISA 是一种新的 ISA**。我们的目标之一是支持架构研究，包括主要 ISA 扩展在内。甚至小型扩展也会减少使用标准 ISA 的收益，因为，为了使用扩展，不得不修改编译器和从源代码重新构建应用。更大的扩展引入了新的架构状态，也需要对操作系统的修改。最终，修改后的商业 ISA 变成了一个新的 ISA，但是它仍然保留了基础 ISA 的所有的遗留的包袱。

我们的立场是，ISA 或许是计算系统中最重要的接口，并且这样一个重要的接口应当没有理由成为专利。居主导的商业 ISA 是基于 30 年前就已经广为人知的指令集概念。软件开发者应当能够瞄准一种开放的标准硬件目标，而商业处理器设计者应当在实现质量上竞争。

我们远不是第一个考虑适合硬件实现的开发 ISA 设计的。我们也考虑过其它的开放 ISA 设计，其中最接近我们目标的是 OpenRISC 架构 [13]。但我们决定不采用 OpenRISC ISA，是由于一些技术原因：

- OpenRISC 具有条件代码和分支延迟槽，这使更高性能的实现复杂化。

- OpenRISC 使用固定 32 位编码和 16 位立即数，这妨碍了更密集的指令编码，并限制了 ISA 的后续扩展的空间。

- OpenRISC 不支持 2008 年修订的 IEEE 754 浮点标准。
- 在我们开始时，OpenRISC 的 64 位设计还没有完成。

通过从一片空白开始，我们可以设计一个满足我们所有目标的 ISA，尽管理所当然地，这比我们一开始所计划的要采取的工作量要大得多。我们现在已经调查了在构建 RISC-V ISA 基础设施方面相当多的工作，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、有效的 ASIC 实现、架构测试套件和教学材料。从这本手册的上一次编辑开始，RISC-V ISA 在学术和工业中已经有了相当大的应用，而我们已经创造了非营利性的 RISC-V 基金会来保护和推广该标准。RISC-V 基金会网站在https://riscv.org，它包括了关于基金会成员和各种使用 RISC-V 的开源项目的最新信息。

## 29.2 从 ISA 手册 1.0 版的修订历史

RISC-V ISA 和指令集手册构建在一些较早的项目之上。监管器级机器的某些方面和手册的整体格式可以追溯到开始于 1992 年 UC 伯克利和 ICSI 的 T0（Torrent-0）向量微处理器项目。T0 是一个基于 MIPS-II ISA 的向量处理器，由克尔斯泰 · 阿桑诺维奇作为主要的架构和 RTL 设计者，以及布莱恩 · 金斯伯里和伯特兰 · 伊里索作为主要的 VLSI 实现者。ICSI 的大卫 · 约翰逊是对 T0 ISA 设计、尤其是监管器模式，以及手册文本的主要贡献者。约翰 · 豪瑟也提供了关于 T0 ISA 设计的相当多的反馈。

麻省理工学院在 2000 年开始的 Scale（用于低能耗的软件控制架构）项目，在 T0 项目基础设施上构建，改良了监管器级的接口；并通过丢弃分支延迟槽，移除了 MIPS 标量 ISA。罗尼 · 克拉辛斯基和克里斯托弗 · 巴顿是麻省理工学院的 Scale 向量线程处理器的主要架构师，而马克 · 汉普顿为 Scale 移植了基于 GCC 的编译器基础设施和工具。

在 2002 年秋季学期，T0 MIPS 标量处理器规范的一个轻微编辑的版本（MIPS-6371）被用于教授 MIT 6.371 的 VLSI 系统入门课程，由克里斯 · 特曼和克尔斯泰 · 阿桑诺维奇作为讲师。克里斯 · 特曼为课程（当时没有 TAI）贡献了大部分的实验材料。2005 年春季，在麻省理工学院，6.371 课程演化为实验性的 6.884 复杂数字设计课程，由阿文和克尔斯泰 · 阿桑诺维奇教授，该课程成为一个常规的春季课程 6.375。在 6.884/6.375 中使用了基于 Scale MIPS 标量 ISA 的一个约简版本，命名为 SMIPS。克里斯托弗 · 巴顿是早期提供了这些的课程的助教，围绕 SMIPS ISA 开发了相当大量的文档和实验材料。这个相同的 SMIPS 实验材料被助教李云燮采纳和强化，用于 UC 伯克利 2009 年秋季的 CS250 VLSI 系统设计课程，该课程由约翰 · 沃兹内克、克尔斯泰 · 阿桑诺维奇和约翰 · 拉扎罗教授。

Maven（向量线程引擎的可塑阵列）项目是一种第二代向量线程架构。它由克里斯托弗 · 巴顿主导设计，当时他是 UC 伯克利的一名开始于 2007 年夏季的交换学者。青木秀田，一名来自日立

的客座工业研究员，给出了关于早期 Maven ISA 和微架构设计的大量反馈。Maven 基础设施是基于 Scale 基础设施，但是 Maven ISA 进一步地移除了 Scale 中定义的 MIPS ISA 变体，而使用一个统一的浮点和整数寄存器文件。设计 Maven 是为了支持带有备用数据并行加速器的实验。李云燮是各种 Maven 向量单元的主要实现者，同时里马斯 · 阿维齐尼斯是各种 Maven 标量单元的主要实现者。李云燮和克里斯托弗 · 巴顿将 GCC 移植到新的 Maven ISA 中共同工作。克里斯托弗 · 塞利奥提供了 Maven 的一种传统的向量指令集（“Flood”）变体的原始定义。

基于所有这些先前项目的经验，在 2010 年夏天，开始了 RISC-V ISA 的定义，由安德鲁 · 沃特曼、李云燮、克尔斯泰 · 阿桑诺维奇和大卫 · 帕特森作为主要设计者。RISC-V 32 位指令子集的一个最初版本被用于 UC 伯克利 2010 年秋季 CS250 VLSI 系统设计课程之中，由李云燮作为助教。RISC-V 与较早的 MIPS 灵感的设计有明显的不同。约翰 · 豪瑟贡献了浮点 ISA 的定义，包括符号注入指令和一个寄存器编码策略，它允许浮点值的内部重新编码。

## 29.3 从 ISA 手册 2.0 版的修订历史

已经完成了 RISC-V 处理器的多个实现，包括一些硅制品，就像表 29.1中显示的那样。

| 名称 | 流片日期 | 处理 | ISA |
|---|---|---|---|
| Raven-1 | 2011 年 5 月 29 日 | ST 28nm FDSOI | RV64G1_Xhwacha1 |
| EOS14 | 2012 年 4 月 1 日 | IBM 45nm SOI | RV64G1p1_Xhwacha2 |
| EOS16 | 2012 年 8 月 17 日 | IBM 45nm SOI | RV64G1p1_Xhwacha2 |
| Raven-2 | 2012 年 8 月 22 日 | ST 28nm FDSOI | RV64G1p1_Xhwacha2 |
| EOS18 | 2013 年 2 月 6 日 | IBM 45nm SOI | RV64G1p1_Xhwacha2 |
| EOS20 | 2013 年 7 月 3 日 | IBM 45nm SOI | RV64G1p99_Xhwacha2 |
| Raven-3 | 2013 年 9 月 26 日 | ST 28nm SOI | RV64G1p99_Xhwacha2 |
| EOS22 | 2014 年 3 月 7 日 | IBM 45nm SOI | RV64G1p9999_Xhwacha3 |

表 29.1: 已制造的 RISC-V 测试芯片。

第一个被制造的 RISC-V 处理器是用 Verilog 编写的，并且在 2011 年作为 Raven-1 测试芯片，以一种从 ST 预先生产的28 纳米FDSOI 技术制造。在克尔斯泰 · 阿桑诺维奇的建议下，李云燮和安德鲁 · 沃特曼共同开发和制造了两个核心：1) 一个带有错误检测反转的 RV64 标量核心，和 2) 一个带有附着 64 位浮点向量单元的 RV64 核心。第一个微架构被非正式地称之为“TrainWreck”，因为可用来完成设计的时间较短，而且使用了不成熟的设计库。

随后，在克尔斯泰 · 阿桑诺维奇的建议下，安德鲁 · 沃特曼、里马斯 · 阿维齐尼斯和李云燮开发了一个干净的微架构，用于有序解耦的 RV64 核，并且，延续了铁路的主题，以乔治 · 史蒂芬森的成功的蒸汽机车设计“Rocket”为代号。Rocket 是用 Chisel 编写的，后者是 UC 伯克利开发的一

种新的硬件设计语言。Rocket 使用的 IEEE 浮点单元由约翰 · 豪瑟、安德鲁 · 沃特曼和布莱恩 · 理查兹开发。在这之后，Rocket 被进一步精炼和开发，并以28 纳米FDSOI 又制造了两次（Raven-2,、Raven-3），并为了一个光电项目以 IBM 45 纳米SOI 技术制造了五次（EOS14、EOS16、EOS18、EOS20、EOS22）。让 Rocket 的设计可以用作一种参数化的 RISC-V 处理器生成器的工作正在进行中。

EOS14-EOS22 芯片包括了 Hwacha 的早期版本，它是一个 64 位的 IEEE 浮点向量单元，在克尔斯泰 · 阿桑诺维奇的建议下，由李云燮、安德鲁 · 沃特曼、海 · 沃、欧伯特、全阮和史蒂芬 · 提格开发。EOS16-EOS22 芯片包括了带有缓存一致协议的两个核，该协议是在克尔斯泰 · 阿桑诺维奇的建议下，由亨利 · 库克和安德鲁 · 沃特曼开发的。EOS14 硅已经成功地以1.25 GHz运行了。EOS16 硅遇到了一个来自 IBM 焊接点库的故障。EOS18 和 EOS20 也已经成功地以1.35 GHz运行。

Raven 测试芯片的贡献者包括李云燮、安德鲁 · 沃特曼、里马斯 · 阿维齐尼斯、布莱恩 · 齐默、扎瓦 · 夸克、鲁齐卡 · 杰夫蒂、米洛万 · 布拉戈耶维奇、阿尔贝托 · 普盖利、斯蒂芬 · 贝利、本 · 凯勒、皮凤秋、布莱恩 · 理查兹、鲍里沃耶 · 尼科利和克尔斯泰 · 阿桑诺维奇。

EOS 测试芯片的贡献者包括李云燮、里马斯 · 阿维齐尼斯、安德鲁 · 沃特曼、亨利 · 库克、海 · 沃、李代伟、孙晨、欧伯特、全阮、史蒂芬 · 提格、弗拉基米尔 · 斯托亚诺维奇和克尔斯泰 · 阿桑诺维奇。

安德鲁 · 沃特曼和李云燮开发了 C++ ISA 模拟器“Spike”，用作开发中的一个黄金模型，并且以用于庆祝美国横贯大陆铁路的完成的黄金钉来命名。Spike 已经可以作为一个 BSD 开源项目而获得了。

安德鲁 · 沃特曼完成了一篇 RISC-V 压缩指令集的初步设计的硕士论文 [24]。

已经完成了 RISC-V 的各种 FPGA 的实现，主要作为 Par 实验室项目研究撤离的集成演示的一部分。最大的 FPGA 设计是运行一个研究操作系统的三个缓存一致 RV64IMA 处理器。FPGA 实现的贡献者包括安德鲁 · 沃特曼、李云燮、里马斯 · 阿维齐尼斯和克尔斯泰 · 阿桑诺维奇。

RISC-V 处理器已经用在了 UC 伯克利的一些课程之中。Rocket 被用在 2011 年秋季推出的 CS250 中，作为班级项目的基础，布莱恩 · 齐默担任助教。对于 2012 年春季的本科 CS152 课程，克里斯托弗 · 塞利奥使用 Chisel 来写了一个教育用 RV32 处理器的套件，以“坦克引擎 Thomas”和朋友们居住的岛屿命名为“Sodor”。该套件包括了一个微代码核，一个无管道核，和 2 级、3 级与 5 级流水线核，并在一个 BSD 许可证下公开可用。该套件后续再次被更新和使用是在 2013 年春季的 CS152 中，李云燮担任助教，以及 2014 年春季，由埃里克 · 洛夫担任助教。克里斯托弗 · 塞利奥也开发了一个乱序的 RV64 设计，称之为 BOOM（伯克利乱序机器），伴有流水线可视化，它被用在 CS152 课程之中。CS152 课程也使用了由安德鲁 · 沃特曼和亨利 · 库克开发的 Rocket 核的缓存一致版本。

整个 2013 年的夏天，RoCC（Rocket 定制协处理器）接口被定义为简化了定制加速器向 Rocket 核的添加。Rocket 和 RoCC 接口在乔纳森 · 巴赫拉赫教授的 2013 年秋季 CS250 VLSI 课程中得到了广泛的使用，为 RoCC 接口构建了一些学生加速器项目。Hwacha 向量单元已经被重写为一个 RoCC 协处理器。

在 2013 年春天，两个伯克利的本科生，全阮和欧伯特，已经成功地将 Linux 移植到 RISC-V 上运行。

在 2014 年 1 月，科林 · 施密特成功地为 RISC-V 2.0 完成了一个 LLVM 后端。

在 2014 年 3 月，大流士 · 拉德在 Bluespec 为 GCC 的移植贡献了软浮点 ABI 支持。

约翰 · 豪瑟贡献了浮点分类指令的定义

我们还了解了一些其它的 RISC-V 核的实现，包括一个由汤米 · 索恩用 Verilog 的实现，和一个由里希尔 · 尼希尔用 Bluespec 的实现。

## 鸣谢

感谢克里斯托弗 · F · 巴顿、普雷斯顿 · 布里格斯、克里斯托弗 · 塞利奥、大卫 · 奇斯纳尔、斯特凡 · 弗洛伊德伯格、约翰 · 豪瑟、本 · 凯勒、里希尔 · 尼希尔、迈克尔 · 泰勒、汤米 · 索恩和罗伯特 · 沃森关于规范 2.0 版本草案 ISA 的评论。

## 29.4 从 2.1 版的修订历史

从引入 2014 年 5 月冻结的 2.0 版本依赖，RISC-V ISA 的应用已经非常迅速，在这样一个简短的历史小节中有太多的活动要记录。或许最重要的单一事件就是，在 2015 年 8 月，非盈利 RISC-V 基金会的成立。基金会现在将接管官方 RISC-V ISA 标准的管理工作，而官方网站 `riscv.org` 是获得关于 RISC-V 标准的新闻和更新的最佳场所。

## 鸣谢

感谢斯科特 · 比默、艾伦 · J · 鲍姆、克里斯托弗 · 塞利奥、大卫 · 奇斯纳尔、保罗 · 克莱顿、帕默 · 达贝尔特、简 · 格雷、迈克尔 · 汉伯格和约翰 · 豪瑟对 2.0 版本规范的评论。

## 29.5 从 2.2 版的修订历史

## 鸣谢

感谢雅各布 · 巴赫迈耶、亚历克斯 · 布拉德伯里、戴维 · 霍纳、斯特凡 · 奥雷尔和约瑟夫 · 迈尔斯对 2.1 版本规范的评论。

## 29.6 2.3 版的修订历史

RISC-V 的应用持续以惊人的速度发展着。

基于保罗 · 邦齐尼的一个提议，约翰 · 豪瑟和安德鲁 · 沃特曼贡献了一个超管级 ISA 扩展。

丹尼尔 · 卢斯蒂格、阿文、克尔斯泰 · 阿桑诺维奇、谢克德 · 弗勒、保罗 · 洛文斯坦、雅廷 · 曼尔卡、卢克 · 马兰杰、玛格丽特 · 马托诺西、维贾亚南德 · 纳加拉扬、里希尔 · 尼希尔、乔纳斯 · 奥伯豪斯、克里斯托弗 · 普尔特、何塞 · 雷诺、彼得 · 苏厄尔、萨米特 · 萨卡尔、卡罗琳 · 特里普、穆拉里达兰 · 维贾亚拉加万、安德鲁 · 沃特曼、德里克 · 威廉姆斯、安德鲁 · 赖特和张思卓贡献了内存一致模型。

## 29.7 赞助

部分 RISC-V 架构和实现的开发由如下赞助者所赞助：

- **Par 实验室**：研究由微软（Award #024263）和英特尔（Award #024894）赞助，并由 U.C.Discovery（Award #DIG07-10227）提供匹配资助。额外的支持来自于 Par 实验室附属的诺基亚、英伟达、甲骨文和三星。
- **项目 Isis**：DoE Award DE-SC0003624。
- **ASPIRE 实验室**：DARPA PERFECT 工程，Award HR0011-12-2-0016。DARPA POEM 工程 Award HR0011-11-C-0100。未来架构研究中心（C-FAR），一个由半导体研究公司资助的 STARnet 中心。额外的支持来自于 ASPIRE 工业赞助者，英特尔，和 ASPIRE 附属，谷歌，惠普企业，华为，诺基亚，英伟达，甲骨文，和三星。

本文的内容并不能必然地反映出美国政府的立场和政策，并且不应被推断出官方的认可。

# 附录 A　RVWMO 说明材料（0.1 版本）

这节使用了更加非正式的语言和具体的例子，提供了更多关于 RVWMO（第 十七章）的解释。这些解释都是为了澄清该公理和保留的程序次序规则的含义和目的。这个附录应当被视为评注；而所有的规范性材料都在第 十七章和 ISA 规范的主体的其余部分中提供。当前的所有已知的差异性都被列在了第 A.7节。任何其它的差异性都是无意的。

## A.1　为什么用 RVWMO?

内存一致性模型遵循着从弱到强的松散谱系。弱内存模型允许更多的硬件实现的灵活性，并提供理论上比强模型更好的性能、每瓦特的性能、能量、可扩展性，和硬件验证开销，但代价是更复杂的编程模型。强模型提供了更简单的编程模型，但是对于可以在流水线和内存系统中执行的各种（非推测性的）硬件优化，要强加更多的约束开销，并且反过来在能量、区域开销和验证负担方面强加一些成本。

RISC-V 选择了 RVWMO 内存模型，它是释放一致性的一个变体。这将它置于了内存模型谱系的两个极端之间。RVWMO 内存模型使架构师能够构建简单的实现、激进的实现，将实现深深地嵌入到一个更大的系统之中，并服务于复杂的内存系统交互，或者任何其它的可能性，所有这些同时又能够以足够强大的高性能支持编程语言内存模型。

为了促进来自其它架构的代码的移植，一些硬件实现可以选择实现 Ztso 扩展，它默认提供了更严格的 RVTSO 次序的语义。为 RVWMO 编写的代码是与 RVTSO 自动且固有地兼容的，但是假定 RVTSO 写的代码不保证在 RVWMO 实现上能够正确地运行。事实上，大多数 RVWMO 实现都将（并且应当）简单地拒绝运行 RVTSO 专用的二进制文件。每个实现必须因此进行选择，或者优先兼容 RVTSO 代码（例如，为了便于来自 x86 的移植），或者反之优先兼容其他实现了 RVWMO 的 RISC-V 核。

在 RVTSO 下，代码中为 RVWMO 所写的一些屏障和/或内存次序注释可能变得冗余；在 Ztso 实现上默认采用 RVWMO 的代价是获取那些在实现上已经变成 no-op 的屏障（例如：FENCE R,RW

和 FENCE RW,W）的增量开销。然而，如果希望兼容非 Ztso 的实现，这些屏障在代码中仍然必须存在。

## A.2 Litmus 测试

这章的解释使用了 *litmus* 测试，或者说，为测试或突出显示内存模型的一个特定部分而设计的小型程序。图 A.1显示了带有两个硬件线程的 litmus 测试的一个例子。作为对这个图和对本章中之后所有图的约定，我们假定 `s0` – `s2` 在所有硬件线程中都被预先设置为相同的值，并且 `s0` 持有由 `x` 标签的地址，`s1` 持有 `y` 的，而 `s2` 持有 `z` 的，这里 `x`、`y` 和 `z` 是对齐到 8 字节边界的不相交的内存位置。每张图在左侧显示了 litmus 测试的代码，在右侧则是一个特定的有效或无效执行的可视化。

| | Hart 0 | | Hart 1 |
|---|---|---|---|
| | ⋮ | | ⋮ |
| | `li t1,1` | | `li t4,4` |
| (a) | `sw t1,0(s0)` | (e) | `sw t4,0(s0)` |
| | ⋮ | | ⋮ |
| | `li t2,2` | | |
| (b) | `sw t2,0(s0)` | | |
| | ⋮ | | ⋮ |
| (c) | `lw a0,0(s0)` | | |
| | ⋮ | | ⋮ |
| | `li t3,3` | | `li t5,5` |
| (d) | `sw t3,0(s0)` | (f) | `sw t5,0(s0)` |
| | ⋮ | | ⋮ |

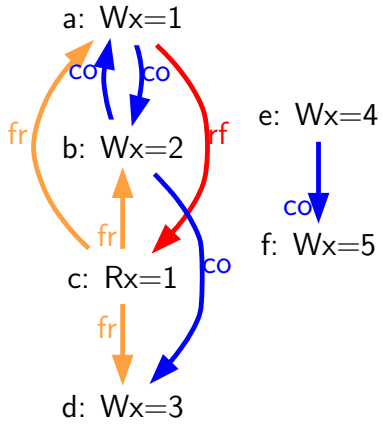



图 A.1: 一个 litmus 测试的示例和一个被禁止的执行（`a0=1`）。

Litmus 测试被用于理解特定具体情境中的内存模型的含义。例如，在图 A.1的 litmus 测试中，根据运行时来自各个硬件线程的指令流的动态交错情况，第一个硬件线程中的 `a0` 的最终的值可以是 2、4 或 5。然而，在这个例子中，硬件线程 0 中的 `a0` 的最终的值将永远都不会是 1 或 3；按直觉，值 1 在加载执行时将不再可见，而值 3 在加载执行时尚未成为可见的。我们下面来分析这个测试和一些其它的测试。

每个 litmus 测试的右侧显示的图展示了正在被考虑的特定执行候选的一个可视化的表示。这些图标使用在内存模型文献中常见的符号，来限制可能的全局内存次序（可能在执行中产生问题）的集合。它也是附录 B.2中展示的 herd 模型的基础。表 A.1中解释了该符号。在列出的关系中，在

| 边 | 全名 (和解释) |
|---|---|
| rf | 读从（Reads From）(从各存储到返回该存储写入值的加载) |
| co | 一致性（Coherence）(关于存储到各地址的一个总的次序) |
| fr | 从读（From-Reads）(从各加载到读取加载所返回值的存储的共同后继) |
| ppo | 保留程序次序（Preserved Program Order） |
| fence | 通过一个 FENCE 指令强行采取的排序 |
| addr | 地址依赖（Address Dependency） |
| ctrl | 控制依赖（Control Dependency） |
| data | 数据依赖（Data Dependency） |

表 A.1: 在这个附录中绘制的 litmus 测试图表的要点

硬件线程之间、co 边、fr 边和 ppo 边之间的 rf 边直接限制了全局内存次序（正如通过 ppo 也可以限制 fence、addr、data，和一些 ctrl）。其它边（例如 infa-hart rf 边）是信息性的，但是不会限制全局内存次序。

例如，在表 A.1中，a0=1 只可能发生在 (c) 读取由 (a) 写入的值、且下列情形之一为真的时候：

- 在全局内存次序中（以及在一致性次序 co 中），(b) 出现在 (a) 之前。然而这违反了 RVWMO PPO 规则 1。从 (b) 到 (a) 的 co 边突出了这一矛盾。
- 在全局内存次序中（以及在一致性次序 co 中），(a) 出现在 (b) 之前。然而，在这种情况中，加载值公理将被违反，因为在程序次序中，(a) 不是先于 (c) 的最近匹配的存储。从 (c) 到 (b) 的 fr 边突出了这一矛盾。

由于这些场景都不满足 RVWMO 公理，结果 a0=1 就被禁止了。

除了在这个附录中描述的内容，在https://github.com/litmus-tests/litmus-tests-riscv中还提供了一套超过七千个的石蕊测试。

---

*litmus 测试项目也提供了关于如何在 RISC-V 硬件上运行 litmus 测试，和如何将结果与操作和公理模型进行比较的指令。*

---

*在未来，我们期望把这些关于内存模型的 litmus 测试也改编作为 RISC-V 一致性测试套件的一部分而使用。*

## A.3 RVWMO 规则的解释

在这节中，我们提供了对所有 RVWMO 规则和公理的解释和例子。

### A.3.1 保留程序次序和全局内存次序

保留程序次序代表了必须在全局内存次序中被遵循的程序次序的子集。概念上，从其它硬件线程和/或观察者的角度，来自相同硬件线程的、按照保留程序次序被排序的事件，必须以该次序出现。

非正式地讲，全局内存次序代表了加载和存储所执行的次序。正式的内存模型文献已经从围绕执行概念构建的规范中移出，但是该思想对于建立非正式的直觉仍然是有用的。对于加载，当它的返回值被确定时，它被称为已执行的。对于存储，不是当它在流水线内部被执行时、而是只有当它的值已经被传播到全局可见的存储时，它才被称为已执行的。在这个意义上，全局内存次序也代表了一致性协议和/或余下的内存系统的贡献：把每个硬件线程发出的（可能被重新排序的）内存访问交错到所有硬件线程都赞成的单一的总次序之中。

加载执行的次序并不总是直接对应于那两个加载所返回的值的相对生存时间。特别地，对相同的地址，一个加载 $b$ 可以在另一个加载 $a$ 之前执行（例如，$b$ 可以在 $a$ 之前执行，并且在全局内存次序中，$b$ 可以出现在 $a$ 之前），但是尽管如此，$a$ 可以返回一个比 $b$ 更早旧的值。这种差异性（在其它事情之中）捕获了核心与内存之间安置的缓冲的重新排序效果。例如，$b$ 可能已经返回了 $a$ 存储在存储缓冲区中的一个值，同时 $a$ 可能已经忽略了较新的存储，反而从内存中读取了一个较旧的值。为了解释这个情况，在每次加载执行的时候，它返回的值由加载值公理决定，而不只是通过确定在全局内存次序中最近对相同地址的存储来严格地决定，正如下面描述的那样。

### A.3.2 加载值公理

> 加载值公理：每个加载 $i$ 的各个位所返回的值，由下列存储中在全局内存次序中最近的那个写到该位：
>
> 1. 写该位，并且在全局内存次序中先于 $i$ 的存储
> 2. 写该位，并且在程序次序中先于 $i$ 的存储

保留程序次序不需要遵循“在重叠的地址上，一个存储跟随着一个加载”的次序。这种复杂度的提升是因为，在几乎所有实现中存储缓冲区都是随处可见的。非正式地说，当存储仍然在存储缓冲区中的时候，加载可以通过从存储转发来执行（返回一个值），并因此出现在了存储自身的执行（写回到全局可见内存）之前。因此，任何其它的硬件线程将观察到，加载在存储之前执行。

考虑表 A.2的 litmus 测试。当在一个带有存储缓冲区（store buffers）的实现上运行这个程序时，它可能得到 `a0=1`, `a1=0`, `a2=1`, `a3=0` 的最终输出结果，如下：

- (a) 执行并进入第一个硬件线程的私有存储缓冲区（store buffer）

| | Hart 0 | | Hart 1 |
|---|---|---|---|
| | li t1, 1 | | li t1, 1 |
| (a) | sw t1,0(s0) | (e) | sw t1,0(s1) |
| (b) | lw a0,0(s0) | (f) | lw a2,0(s1) |
| (c) | fence r,r | (g) | fence r,r |
| (d) | lw a1,0(s1) | (h) | lw a3,0(s0) |

输出结果 a0=1, a1=0, a2=1, a3=0

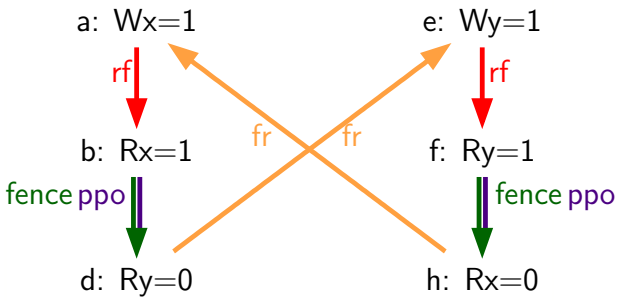



图 A.2: 一个存储缓冲区转发石蕊测试（允许的输出结果）

- (b) 执行并从存储缓冲区中的 (a) 转发它的返回值 1
- (c) 当所有之前的加载操作（例如，(b)）都已经完成时执行
- (d) 执行并从内存读取值 0
- (e) 执行并进入第二个硬件线程的私有存储缓冲区
- (f) 执行并从存储缓冲区中的 (e) 转发它的值 1
- (g) 从所有之前的加载操作（例如，(f)）都已经完成时执行
- (h) 执行并从内存读取值 0
- (a) 从第一个硬件线程的存储缓冲区排放到内存
- (e) 从第二个硬件线程的存储缓冲区排放到内存

因此，内存模型必须能够解释这种行为。

换句话说，假设保留程序次序确实包括了下列假定的规则：在保留的程序次序中，内存访问 $a$ 先于内存访问 $b$（并因此也在全局内存次序中先于 $b$），如果在程序次序中 $a$ 先于 $b$，并且 $a$ 和 $b$ 访问相同的内存位置，$a$ 是一个写，而 $b$ 是一个读。把这个称作“规则 X”。然后我们得到如下结果：

- (a) 先于 (b): 根据规则 X
- (b) 先于 (d): 根据规则 4
- (d) 先于 (e): 根据加载值公理。否则，如果 (e) 先于 (d)，那么将需要 (d) 返回值 1。(这是一个完全合法的执行；它只是并非问题所在)
- (e) 先于 (f): 根据规则 X

- (f) 先于 (h)：根据规则 4
- (h) 先于 (a)：根据加载值公理，同上。

全局内存次序必须是一个总次序，而不能有循环，因为循环将暗示该循环内的每个事件都发生在它自己之前，这是不可能的。因此，上面提出的执行将被禁止，并因此，规则 X 的添加将禁止带有存储缓冲区转发的实现，这显然是不可取的。

尽管如此，即使在全局内存次序中，(b) 先于 (a) 且/或 (f) 先于 (e)，这个例子中唯一合理的可能性也是，对于 (b)，返回由 (a) 所写的值，而 (f) 和 (e) 类似。这种情况的组合导致了加载值公理的定义中的第二个选项。即使在全局内存次序中，(b) 先于 (a)，由于在 (b) 执行的时候 (a) 还位于存储缓冲区中，(a) 将仍然对 (b) 可见。因此，即使在全局内存次序中 (b) 先于 (a)，(b) 也应当返回由 (a) 所写的值，因为在程序次序中 (a) 先于 (b)。对于 (e) 和 (f) 也类似。

| Hart 0 | | Hart 1 | |
|---|---|---|---|
| | li t1, 1 | | li t1, 1 |
| (a) | sw t1,0(s0) | | LOOP: |
| (b) | fence w,w | (d) | lw a0,0(s1) |
| (c) | sw t1,0(s1) | | beqz a0, LOOP |
| | | (e) | sw t1,0(s2) |
| | | (f) | lw a1,0(s2) |
| | | | xor a2,a1,a1 |
| | | | add s0,s0,a2 |
| | | (g) | lw a2,0(s0) |

输出结果：a0=1, a1=1, a2=0

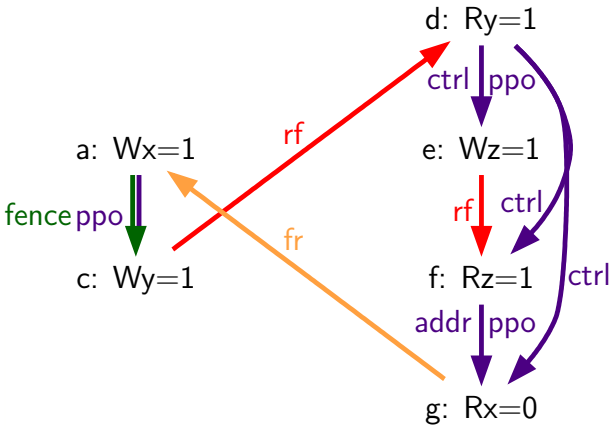



图 A.3：“PPOCA”存储缓冲区转发 litmus 测试（允许的输出结果）

在图 A.3中显示了另一个突出存储缓冲区行为的测试。在这个例子中，由于控制依赖，(d) 的次序排在 (e) 之前，而由于地址依赖，(f) 的次序排在 (g) 之前。然而，即使 (f) 返回了由 (e) 所写的值，(e) 的次序也并不需要排在 (f) 之前。这个可能对应到下列事件序列：

- (e) 推测地执行，并进入第二个硬件线程的私有存储缓冲区（但是没有排放到内存）
- (f) 推测地执行，并从存储缓冲区中的 (e) 转发它的值 1
- (g) 推测地执行，并从内存读取值 0
- (a) 执行，进入第一个硬件线程的私有存储缓冲区，并排放到内存

- (b) 执行，并退场
- (c) 执行，进入第一个硬件线程的私有存储缓冲区，并排放到内存
- (d) 执行，并从内存读取值 1
- (e), (f), 和 (g) 提交，因为推测是正确的
- (e) 从存储缓冲区排放到内存

### A.3.3 原子性公理

> 原子性公理 (对于对齐的原子)：如果 $r$ 和 $w$ 是由一个硬件线程 $h$ 中对齐的 LR 和 SC 指令所生成的配对的加载和存储操作，$s$ 是一个对于字节 $x$ 的存储，而 $r$ 返回 $s$ 所写的值，那么在全局内存次序中，$s$ 必须先于 $w$。并且在全局内存次序中，在 $s$ 之后、$w$ 之前，没有来自同一硬件线程的不同于 $h$ 的存储。

RISC-V 架构把原子性的概念从排序的概念中解耦出来。不像诸如 TSO 的架构，RISC-V 在 RVWMO 下的原子性不会默认采用任何排序需求。排序的语义仅仅由 PPO 规则保证，否则就是适用的。

RISC-V 包含两种类型的原子性：AMO 和 LR/SC 对。通过下列方式，它们在概念上有不同的表现。LR/SC 的行为就像是，旧值被带到核心，修改，然后写回到内存，所有这些保留都维持在该内存位置。另一方面，AMO 在概念上表现得像是，它们直接在内存中执行。AMO 因此有固有的原子性，而 LR/SC 对的原子性在某种意义上略有不同，即在内存位置方面，在起初的硬件线程持有该保留的期间，不会被另一个硬件线程所修改。

```
(a) lr.d a0, 0(s0)      (a) lr.d a0, 0(s0)      (a) lr.w a0, 0(s0)      (a) lr.w a0, 0(s0)
(b) sd   t1, 0(s0)      (b) sw   t1, 4(s0)      (b) sw   t1, 4(s0)      (b) sw   t1, 4(s0)
(c) sc.d t2, 0(s0)      (c) sc.d t2, 0(s0)      (c) sc.w t2, 0(s0)      (c) sc.w t2, 8(s0)
```

图 A.4: 在所有的四个（独立的）代码片段中，存储条件 (c) 是被允许的，但是不保证成功

原子性公理禁止在全局内存次序中，来自其它硬件线程的存储，在一个 LR 核、与该 LR 配对的 SC 之间交错。原子性公理没有禁止在程序次序或全局内存次序中，加载在成对的操作之间交错，也没有禁止在程序次序或全局内存次序中，来自相同硬件线程的存储或者对非重叠位置的存储出现在成对的操作之间。例如，图 A.4中的 SC 指令可以（但是不保证）成功。那些成功没有一个将违背原子性公理，因为其间的非条件存储是与成对的加载-存储指令和存储-条件指令来自相同的硬件线程。这样，一个在缓存行粒度追踪内存访问（并因此将看到图 A.4中的四个片段是完全相同的）的

内存系统将不会强制让碰巧（假）共享了相同缓存行另一部分作为保留所正在持有的内存位置的存储-条件指令失败。

这个原子性公理也技术性地支持了 LR 和 SC 接触不同地址和/或使用不同访问尺寸的情形；然而，在实际中，预计这种行为的使用情形会很稀少。同样地，那种在一个 LR/SC 对之间，来自相同硬件线程的存储与该 LR 或 SC 引用的内存位置实际重叠的情景，与其间的存储可能简单地落在相同的缓存行上的情景相比，也是稀少的。

### A.3.4 进度公理

> 进度公理: 在全局内存次序之中，任何内存操作之前都不能有其它内存操作的无限序列。

进程公理确保了一个最小的向前进程保证。它确保了来自一个硬件线程的存储将在有限数量的时间之内，最终变得对于系统中的其它硬件线程可见，并且来自其它硬件线程的加载将最终能够读取那些值（或由该值而来的后继值）。没有这个规则的话，举个例子，一个自旋锁无限地在一个值上旋转，将变得合法，甚至是在有来自另一个硬件线程的存储正在等待该自旋锁解锁的时候。

进程公理并不试图在一个 RISC-V 实现中的硬件线程上采用任何其它的公平、延迟或者服务质量的概念。任何更强的公平性概念都取决于剩余的 ISA 和/或平台和/或设备的定义和实现。

在几乎所有的情况中，向前进程公理都将被任何标准的缓存一致协议所满足。带有非一致性缓存的实现可能不得不提供一些其它的机制来确保所有的存储（或者由此而来的后继者）对于所有硬件线程的最终可见性。

### A.3.5 重叠地址排序（规则 1-3）

> Rule 1: $b$ 是一个存储操作，且 $a$ 和 $b$ 访问了重叠的内存地址
> Rule 2: $a$ 和 $b$ 是加载操作，$x$ 是 $a$ 和 $b$ 都读取的一个字节，以程序次序在 $a$ 和 $b$ 之间没有对 $x$ 的存储操作，并且 $a$ 和 $b$ 返回由不同的内存操作所写入的 $x$ 的值
> Rule 3: $a$ 是由 AMO 或 SC 指令生成的操作，$b$ 是加载操作，且 $b$ 返回一个由 $a$ 写入的值

相同地址排序，如果后者是一个存储，那么是简单的：一个加载或存储永远不会被重新排序到一个与后面的存储重叠的内存位置。从微架构的视角，总的来说，如果推测被证明是无效的，很难或者说不可能来撤销一个推测性重排的存储，因此这种行为被模型简单地禁止了。换句话说，不需要从一个存储到后一个加载的相同地址排序。正如在 A.3.2节中讨论的那样，这反映了将值从缓冲的存储转发到之后的加载的实现的可观察的行为。

相同地址的加载-加载排序的要求要微妙得多。基础要求是，较新的加载一定不能返回比同一个硬件线程中对相同地址进行的较旧的加载所返回的值更旧的值。这通常被称为“CoRR”（读-读对的一致性），或者称为更宽泛的“一致性”或者“各位置的顺序连贯性”需求的一部分。过去，一些架构已经放松了相同地址的加载-加载排序，但是事后看来，这通常会让编程模型变得过于复杂，并且因此 RVWMO 需要强制执行 CoRR 排序。然而，因为全局内存次序对应于加载执行的次序，而不是值被返回的次序，所以，从全局内存次序的角度，捕获 CoRR 的需求需要一点间接性。

| Hart 0 | | Hart 1 | |
|---|---|---|---|
| | li t1, 1 | | li t2, 2 |
| (a) | sw t1,0(s0) | (d) | lw a0,0(s1) |
| (b) | fence w, w | (e) | sw t2,0(s1) |
| (c) | sw t1,0(s1) | (f) | lw a1,0(s1) |
| | | (g) | xor t3,a1,a1 |
| | | (h) | add s0,s0,t3 |
| | | (i) | lw a2,0(s0) |

输出结果：a0=1, a1=2, a2=0

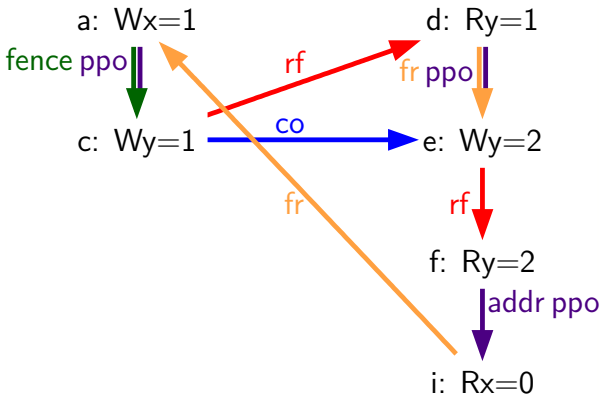



图 A.5: 石蕊测试 MP + fence.w.w + fir-rfi-addr（允许的输出结果）。

考虑图 A.5的石蕊测试，它是更一般的“fri-rfi”式样的一个特别的实例。术语“fri-rfi”表示序列 (d)、(e)、(f)：(d)“从读取”来自相同硬件线程的 (e)（例如，从一个比 (e) 更早的写读取），而 (f) 从来自相同硬件线程的 (e) 读取。

从微架构的视角，输出结果 a0=1，a1=2，a2=0 是合法的（比起各种其它更加不怎么微妙的输出结果）。直观地讲，下列将产生上述提及的输出结果：

- (d) 暂停（不论出于什么原因；或许它在等待一些其它先前的指令时就暂停了）
- (e) 执行，并进入存储缓冲区（但是还没有排放到内存）
- (f) 执行，并从存储缓冲区中的 (e) 转发
- (g)、(h) 和 (i) 执行
- (a) 执行并排放到内存，(b) 执行，且 (c) 执行并排放到内存
- (d) 解除暂停并执行
- (e) 从存储缓冲区排放到内存

这个对应于全局内存次序 (f)、(i)、(a)、(c)、(d)、(e)。注意，即使 (f) 在 (d) 之前执行，由 (f) 返回的值也比由 (d) 返回的值更新。因此，这个执行是合法的，并且不会违背 CoRR 需求。

类似地，如果两个背靠背的加载返回了相同存储所写入的值，那么在全局内存次序中，它们也可以乱序出现而不违背 CoRR。注意这与说两个加载返回相同的值是不相同的，因为两个不同的存储也可以写入相同的值。

| | Hart 0 | | Hart 1 |
|---|---|---|---|
| | `li t1, 1` | (d) | `lw a0,0(s1)` |
| (a) | `sw t1,0(s0)` | (e) | `xor t2,a0,a0` |
| (b) | `fence w, w` | (f) | `add s4,s2,t2` |
| (c) | `sw t1,0(s1)` | (g) | `lw a1,0(s4)` |
| | | (h) | `lw a2,0(s2)` |
| | | (i) | `xor t3,a2,a2` |
| | | (j) | `add s0,s0,t3` |
| | | (k) | `lw a3,0(s0)` |

输出结果：`a0=1, a1=`$v$`, a2=`$v$`, a3=0`

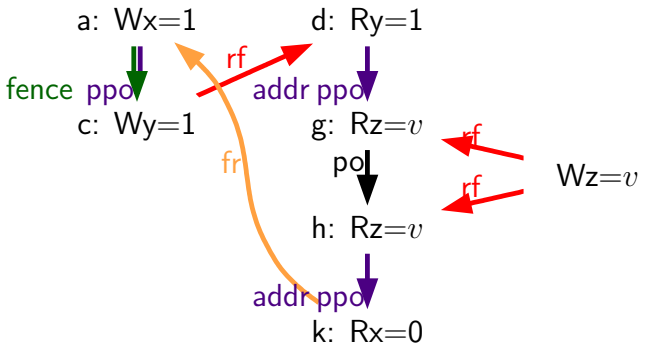



图 A.6: 石蕊测试 RSW（允许的输出结果）

考虑图 A.6的石蕊测试。输出结果 `a0=1`、`a1=`$v$、`a2=`$v$、`a3=0`（这里 $v$ 是由另一个硬件线程所写入的某个值）可以通过允许 (g) 和 (h) 重排而被观察到。这个做法可能是推测性的，并且该推测可以被微架构证明（例如，通过监视缓存失效情况而没有发现），因为无论如何，在 (g) 之后重新执行 (h) 都将返回相同存储所写入的值。因此假设，无论如何 a1 和 a2 都将最终由相同的存储写入相同的值，(g) 和 (h) 可以被合法地重新排序。对应这个执行的全局内存次序将是 (h)、(k)、(a)、(c)、(d)、(g)。

图 A.6中 `a1` 不等于 `a2` 的测试的执行实际上需要 (g) 在全局内存次序中出现在 (h) 之前。允许 (h) 在全局内存次序中出现在 (g) 之前，那种情况中将导致违反 CoRR，因为接下来 (h) 将返回一个比 (g) 所返回的更旧的值。因此，PPO 规则 2禁止这种 CoRR 违背的发生。严格来说，PPO 规则 2在所有情况中执行 CoRR 都达成了一种小心的平衡，同时又足够弱，以允许在真实的微架构中经常出现的“RSW”和“fri-rfi”式样。

还有一个重叠地址规则：PPO 规则 3简单地陈述了，一个值不能从 AMO 或 SC 返回到后续的加载，直到 AMO 或 SC 已经全局执行（在 SC 的情况中，还要是成功的全局执行）。从概念的观点，AMO 和 SC 指令理应在内存中原子性地执行，这有点是顺理成章的。然而，特别地，PPO 规则 3陈述了，硬件可能甚至不会无意识地转发正在被 AMOSWAP 存储到后续加载的值，即使对于 AMOSWAP，该存储的值实际在语义上并不依赖于内存中的先前的值，就像对于其它 AMO 的情况一样。即使当从 SC 转发存储的值在语义上不依赖于所配对的 LR 所返回的值时也同样如此。

上面这三个 PPO 规则也应用在当上述提到的内存访问只有部分重叠的情况中。这是可能发生的，例如，当使用了不同尺寸的访问来访问相同的对象。也要注意，为了两个内存访问重叠，两个重叠的内存操作的基地址不需要必定是相同的。当使用了未对齐的内存访问的时候，重叠地址 PPO 规则独立地应用到每个组件内存访问。

### A.3.6 屏障（规则 4）

> Rule 4: 在 $b$ 之前有一个排序 $a$ 的 FENCE 指令

默认情况下，FENCE 指令确保程序次序中所有的来自屏障之前的指令的内存访问（即，“前趋集”）在全局内存次序中，早于程序次序中来自屏障之后的指令的内存访问（即，“后继集”）出现。然而，屏障可以选择性地把前趋集和/或后继集进一步限制到一个更小的内存访问集合，以提供某些加速。特别地，屏障拥有限制前趋集和/或后继集的 PR、PW、SR 和 SW 位。当且仅当设置了 PR 位（对应于 PW）时，前趋集包括加载（对应于存储）。类似地，当且仅当设置了 SR（对应于 SW）时，后继集包括加载（对应于存储）。

当前的 FENCE 编码有关于四个位 PR、PW、SR 和 SW 的九个非平凡的组合，加上一个额外的编码 FENCE.TSO，它有助于“获取 + 释放”或 RVTSO 语义的映射。剩余的七个组合没有前趋集和/或后继集，并因此都是 no-op。对于十个非平凡的选项，只有六个是在实际中经常使用的：

- FENCE RW,RW
- FENCE.TSO
- FENCE RW,W
- FENCE R,RW
- FENCE R,R
- FENCE W,W

使用 PR、PW、SR 和 SW 的任何其它组合的 FENCE 指令是被保留的。我们强烈建议编程人员坚持使用这六个。其它的组合可能与内存模型有未知的或者不期望的交互。

最后，我们注意到，由于 RISC-V 使用一种多重拷贝原子的内存模型，编程人员因此可以以一种线程本地的方式来推断屏障位。在非多重拷贝原子的内存模型中，没有“屏障累积性”的复杂性概念。

### A.3.7 显式同步（规则 5-8）

> 规则 5: $a$ 有一个 acquire 注释
> 规则 6: $b$ 有一个 release 注释
> 规则 7: $a$ 和 $b$ 都有 RCsc 注释
> 规则 8: $a$ 与 $b$ 是成对的

一个获取操作，正如应当被用在临界区开始处那样，需要在程序次序中的所有之后的内存操作也都在全局内存次序中在获取操作之后。这确保了，例如，临界区之内的所有的加载和存储，相对于正在保护它们的同步变量，都是最新的。获取次序可以通过两种方式之一而采用：通过一个 acquire 注释，它采用只相对于同步变量自身的次序，或者通过一个 FENCE R, RW，它采用相对于所有先前的加载的次序。

```
    sd           x1, (a1)     # 任意不相关的存储
    ld           x2, (a2)     # 任意不相关的加载
    li           t0, 1        # 初始化交换值
again:
    amoswap.w.aq t0, t0, (a0) # 尝试获取锁
    bnez         t0, again    # 如果被占用则重试
    # ...
    # Critical section.
    # ...
    amoswap.w.rl x0, x0, (a0) # 通过存入0来释放锁
    sd           x3, (a3)     # 任意不相关的存储
    ld           x4, (a4)     # 任意不相关的加载
```

图 A.7: 带原子性的自旋锁

考虑图 A.7。因为这个例子使用 $aq$，临界区中的加载和存储被保证在全局内存次序中出现在用于获取锁的 AMOSWAP 之后。然而，假设 `a0`、`a1` 和 `a2` 指向不同的内存位置，临界区中的加载和存储可能会、或可能不会在全局内存次序中，出现在例子开始的“任意不相干的加载”之后。

现在，考虑图 A.8中的替代方案。在这种情况中，即使 AMOSWAP 不会采用带有 $aq$ 位的次序，尽管如此，屏障也会使获取 AMOSWAP 在全局内存次序中早于临界区中的所有加载和存储出现。然而，注意，在这种情况中，屏障也会强制采用额外的次序：它也需要程序开始处的“任意不相干的加载”在全局内存次序中比临界区的加载和存储出现得更早。（然而，这个特别的屏障并不强制采用任何相对于片段开始处的“任意不相干的存储”的次序。）通过这种方式，屏障强加的次序比通过.$aq$ 采用的次序会稍微地更粗糙些。

```
    sd          x1, (a1)     # 任意不相关的存储
    ld          x2, (a2)     # 任意不相关的加载
    li          t0, 1        # 初始化交换值
again:
    amoswap.w   t0, t0, (a0) # 尝试获取锁
    fence       r, rw        # 强制采用“acquire”内存次序
    bnez        t0, again    # 如果被占用则重试
    # ...
    # Critical section.
    # ...
    fence       rw, w        # 强制采用“release”内存次序
    amoswap.w   x0, x0, (a0) # 通过存入0来释放锁
    sd          x3, (a3)     # 任意不相关的存储
    ld          x4, (a4)     # 任意不相关的加载
```

图 A.8: 带屏障的自旋锁

释放次序和获取次序的效果完全相同，只是方向相反。释放的语义需要在程序次序中所有的先于释放操作的加载和存储也要在全局内存次序中先于释放操作。这确保了，例如，在临界区中的内存访问在全局内存次序中出现在锁释放存储之前。正如和获取的语义一样，释放的语义可以使用 relase 注释或者用 FENCE RW, W 操作来强制采用。使用相同的例子，临界区中的加载和存储和代码片段末尾处的“任意不相干存储”之间的次序只由图 A.8的 FENCE RW,W 采用，而不是图 A.7中的 *rl*。

单独使用 RCpc 注释，存储-释放到加载-获取的次序是不会被强制采用的。这有助于在 TSO 和/或 RCpc 内存模型下所写的代码的移植。为了强制采用存储-释放到加载-获取的次序，代码必须使用 store-release-RCsc 和 load-acquire-RCsc 操作，以便应用 PPO 规则7。对于许多 C/C++ 中的使用情形，只有 RCpc 来举一些例子是足够的，但是对于许多 C/C++、Java 和 Linux 中的其它的使用情形是不够的；详情请见 A.5节。

PPO 规则 8说明了，一个 SC 必须在全局内存次序中出现在它所配对的 LR 之后。由于固有的数据依赖，这将自然地从 LR/SC 的常见使用开始去执行一个原子的读-修改-写操作。然而，PPO 规则 8也会应用，即使当正在存储的值在句法上并不依赖于所配对的 LR 所返回的值。

最后，我们注意到，只使用屏障，编程人员在分析排序注释的时候，不需要担心“累积性”。

### A.3.8 句法依赖（规则 9-11）

> 规则 9: b 有一个关于 a 的句法地址依赖
> 规则 10: *b* 有一个关于 *a* 的句法数据依赖
> 规则 11: *b* 是一个存储操作，且 *b* 有一个关于 *a* 的句法控制依赖

从一个加载到相同硬件线程中的后续内存操作的依赖是 RVWMO 内存模型所考虑的。Alpha 内存模型由于选择不强制采用这些依赖而著名，但是大多数现代硬件和软件内存模型都考虑允许依赖指令被重新排序，是过于混乱和违反直觉的。此外，现代代码有时会故意使用这种依赖，作为一种特别轻量级的排序实施机制。

第 17.1节中的术语工作如下。无论何时，当写入每个目的寄存器的值是源寄存器的函数的时候，指令被称为携带了从它们的源寄存器到它们的目的寄存器的依赖。对于大多数指令，这意味着，目的寄存器携带了一个来自所有源寄存器的依赖。然而，也有一些著名的例外。在内存指令的情形中，写入目的寄存器的值最终来自于内存系统，而不是直接来自源寄存器，并因此这样打破了所携带的来自源寄存器的依赖链。在无条件跳转的情形中，写入目的寄存器的值来自于当前的 `pc`（它永远不会被内存模型认为是一个源寄存器），并因此类似地，JALR（仅有的带有源寄存器的跳转）不会携带一个从 *rs1* 到 *rd* 的依赖。

```
(a) fadd  f3,f1,f2
(b) fadd  f6,f4,f5
(c) csrrs a0,fflags,x0
```

图 A.9: 通过 `fflags`，一个 (a) 和 (b) 都会隐含地累积进去的目的寄存器，(c) 有一个关于 (a) 和 (b) 的句法依赖。

累积到一个目的寄存器，而不是写入它，这个概念反应了类似 `fflags` 的 CSR 的行为。特别地，累积进一个寄存器不会冲击到任何先前的写入或对相同寄存器的累积。例如，在图 A.9中，(c) 有一个同时关于 (a) 和 (b) 的句法依赖。

类似其它现代内存模型，RVWMO 内存模型使用句法依赖，而不是语义依赖。换句话说，这个定义依赖于正在被不同指令访问的寄存器的标识，而不是那些寄存器的实际的内容。这意味着地址依赖、控制依赖、或者内存依赖必须是强制采用的，即使计算看起来是可以“优化掉”的。这个选择确保了 RVWMO 保留了与使用这些假句法依赖作为轻量级排序机制的代码的兼容性。

```
ld  a1,0(s0)
xor a2,a1,a1
add s1,s1,a2
ld  a5,0(s1)
```

图 A.10: 一个句法地址依赖

例如，在图 A.10中，存在一个从第一个指令生成的内存操作到最后一个指令生成的内存操作的句法地址依赖，即使 a1 XOR a1 是零，并因此不会影响到第二个加载所访问的地址。

使用依赖作为轻量级同步机制的好处是，排序的强制性需求仅仅被限制在上述提及的特定的两个指令。其它非依赖的指令可以被激进的实现自由地重排。一个替代方案是使用一个加载-获取，但是这将强制第一个加载相对于所有后继指令的次序。另一个替代方案是使用 FENCE R, R，但是这将包含所有先前的加载和*所有*后继的加载，使这个选择更加昂贵。

```
      lw  x1,0(x2)
      bne x1,x0,next
      sw  x3,0(x4)
next: sw  x5,0(x6)
```

图 A.11: 一个句法控制依赖

一个控制依赖总是扩展到程序次序中跟在原始目标之后的所有的指令，在这个意义上，控制依赖的表现不同于地址依赖和数据依赖。考虑表 A.11：尽管下一个指令将总是执行，但是由上一个指令生成的内存操作仍然有一个来自第一个指令生成的内存操作的控制依赖。

```
      lw  x1,0(x2)
      bne x1,x0,next
next: sw  x3,0(x4)
```

图 A.12: 另一个句法控制依赖

类似地，考虑图 A.12。即使两个分支的最终结果都有相同的目标，仍然存在从这个片段的第一个指令生成的内存操作，到最后一个指令生成的内存操作的一个控制依赖。控制依赖的这个定义比其它环境中（例如，C++）可能看到的要强一些，但是它符合文献中控制依赖的标准定义。

显然，PPO 规则9 - 11也是有意设计的，以尊重来源于成功的存储条件指令的输出的依赖。通常，一个 SC 指令将跟随一个检测输出结果是否成功的条件分支；这暗示了将会有一个从 SC 指令生成的存储操作到分支随后的任何内存操作的控制依赖。PPO 规则 11反过来暗示了，任何后继的存储操作将在全局内存次序中比 SC 生成的存储操作出现得更晚。然而，由于控制依赖、地址依赖和数据依赖是定义在内存操作上的，并且由于一次不成功的 SC 不会生成内存操作，所以在不成功的 SC 和它的依赖指令之间不会强制排序。并且，由于只有当 SC 成功的时候，SC 才被定义为携带从它的源寄存器到 *rd* 的依赖，一次不成功的 SC 不会影响全局内存次序。

此外，选择尊重源自于存储-条件指令的依赖确保了，特定的类似无中生有的行为会被阻止。考虑图 A.13。假设一个假想的实现可以偶然地做到提前保证存储-条件操作将会成功。在这种情形中，(c) 将提前返回 0 到 a2（在实际执行之前），从而允许序列 (d)、(e)、(f)、(a)、然后是 (b) 的执行，接着 (c) 可能只在那一点（成功地）执行。这将表示 (c) 把它自己的成功的值写到了 0(s1)！幸运

初始值：0(s0)=1; 0(s2)=1

| | Hart 0 | | Hart 1 |
|---|---|---|---|
| (a) | ld a0,0(s0) | (e) | ld a3,0(s2) |
| (b) | lr a1,0(s1) | (f) | sd a3,0(s0) |
| (c) | sc a2,a0,0(s1) | | |
| (d) | sd a2,0(s2) | | |

输出结果：a0=0, a3=0

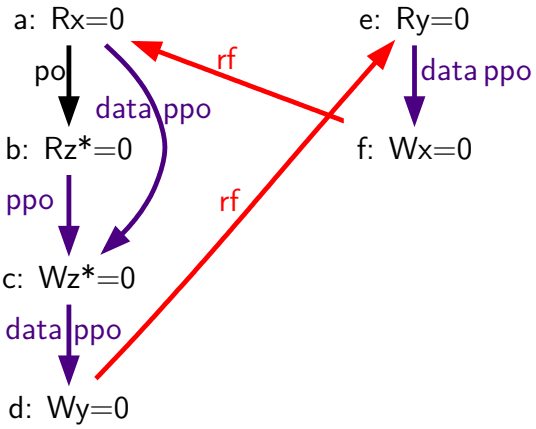



图 A.13: LB 石蕊测试的一种变体（禁止的输出结果）

的是，由于 RVWMO 尊重源自于由成功的 SC 指令生成的存储的依赖的事实，这个情形和其它类似的情形被阻止了。

我们也注意到，指令之间的句法依赖，只有当它们采取句法地址依赖、句法控制依赖，和/或句法数据依赖的形式时才会有效力。例如， 17.3节中，在两个“F”指令之间通过“累积 CSR”形成的句法依赖并不表示这两个“F”指令必须按次序执行。这种依赖将只会用于之后最终建立从两个“F”指令到之后的上述提及的访问 CSR 标志的 CSR 指令的依赖。

### A.3.9 流水线依赖（规则 12-13）

> 规则 12: $b$ 是一个加载操作，且按程序次序，在 $a$ 和 $b$ 之间存在某些存储操作 $m$，使得 $m$ 有一个关于 $a$ 的地址依赖或数据依赖，而 $b$ 返回一个由 $m$ 写入的值
> 规则 13: $b$ 是一个存储操作，且按程序次序，在 $a$ 和 $b$ 之间存在某些指令 $m$，使 $m$ 有一个关于 $a$ 的地址依赖

PPO 规则 12和13反应了几乎所有的真实的处理器流水线实现的行为。规则 12陈述了一个加载不能从一个存储转发，直到那个存储的地址和数据是已知的时候。考虑图 A.14：在 (e) 的数据被决定之前，(f) 不能执行，因为 (f) 必须返回由 (e) 写入的值（或者在全局内存次序中由某些更加靠后的所写入的值），而在 (d) 有机会执行之前，旧的值必须不能被 (e) 的写回所冲击。因此，(f) 将永远不会在 (d) 的执行之前执行。

如果在 (e) 和 (f) 之间，有另一个针对相同地址的存储，就像图 A.15中的那样，那么 (f) 将不再依赖于 (e) 正在被决定的数据，并因此 (f) 关于为 (e) 生产数据的 (d) 的依赖将被打破。

| | Hart 0 | | Hart 1 |
|---|---|---|---|
| | li t1, 1 | (d) | lw a0, 0(s1) |
| (a) | sw t1,0(s0) | (e) | sw a0, 0(s2) |
| (b) | fence w, w | (f) | lw a1, 0(s2) |
| (c) | sw t1,0(s1) | | xor a2,a1,a1 |
| | | | add s0,s0,a2 |
| | | (g) | lw a3,0(s0) |

输出结果：a0=1, a3=0

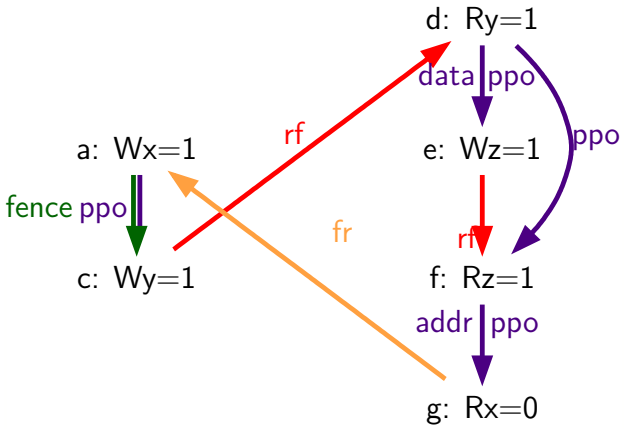



图 A.14：根据 PPO 规则 12和从 (d) 到 (e) 的数据依赖，在全局内存次序中，(d) 必须也先于 (f)（禁止的输出结果）

| | Hart 0 | | Hart 1 |
|---|---|---|---|
| | li t1, 1 | | li t1, 1 |
| (a) | sw t1,0(s0) | (d) | lw a0, 0(s1) |
| (b) | fence w, w | (e) | sw a0, 0(s2) |
| (c) | sw t1,0(s1) | (f) | sw t1, 0(s2) |
| | | (g) | lw a1, 0(s2) |
| | | | xor a2,a1,a1 |
| | | | add s0,s0,a2 |
| | | (h) | lw a3,0(s0) |

输出结果：a0=1, a3=0

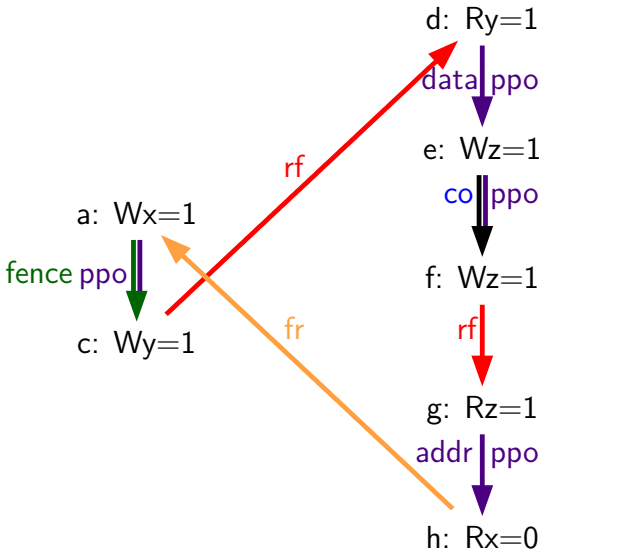



图 A.15：根据 PPO 规则 12 和从 (d) 到 (e) 的数据依赖，在全局内存次序中，(d) 必须也先于 (f)（禁止的输出结果）

规则 13制定了一个和之前规则相似的观点：一个存储不能在内存执行，直到所有的先前可能访问相同地址的加载自身已经被执行。这样的加载看起来必须在存储之前执行，但是如果存储在加载有机会读取旧值之前就覆写了内存中的值，它就不能这么做了。类似地，一个存储通常不能执行，直到它已知了先前的指令不会由于地址解析失败而引发异常，而从这个意义上讲，规则 13可以被视作规则 11的某种特殊情况。

考虑图 A.16：在 (e) 的地址被决定之前，(f) 不能执行，因为它可能会导致地址匹配；也就是说，a1=s0。因此，在 (d) 已经执行并证实地址是否确实重叠之前，(f) 不能被送到内存。

| Hart 0 | | Hart 1 | |
|---|---|---|---|
| | | | li t1, 1 |
| (a) | lw a0,0(s0) | (d) | lw a1, 0(s1) |
| (b) | fence rw,rw | (e) | lw a2, 0(a1) |
| (c) | sw s2,0(s1) | (f) | sw t1, 0(s0) |

输出结果：a0=1, a1=t

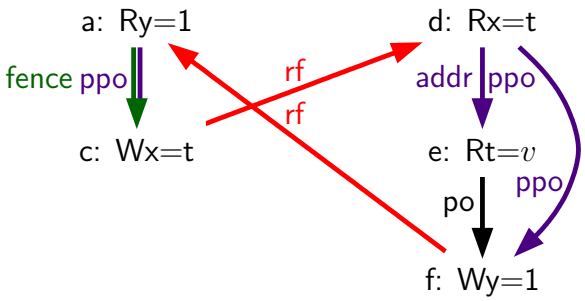



图 A.16: 因为在 (e) 和 (g) 之间的额外的存储，(d) 不再必须先于 (g) 了（允许的输出结果）

## A.4 超出主存范围

RVWMO 当前不会尝试正式地描述 FENCE.I、SFENCE.VMA、I/O 屏障和 PMA 的行为如何。所有这些的行为都将在未来的形式化中描述。与之同时，FENCE.I 的行为描述在第 三章中，SFENCE.VMA 的行为描述在 RISC-V 指令集特权架构手册之中，而 I/O 屏障的行为和 PMA 的效果将在下面描述。

### A.4.1 一致性和可缓存性

RISC-V 特权 ISA 定义了物理内存属性（PMA），除此之外，它指定了地址空间的各部分是否是一致的和/或可缓存的。完整的细节见 RISC-V 特权 ISA 规范。这里，我们简单地讨论每个 PMA 中的各种细节是如何关系到内存模型的：

- 主内存 vs I/O，以及 I/O 内存次序 PMA：定义的内存模型适用于主内存区域。I/O 次序在下面讨论。
- 支持的访问类型和原子性 PMA：内存模型简单地被应用在每个区域所支持的任何原语之上。
- 可缓存性 PMA：可缓存性 PMA 总体上不会影响内存模型。非可缓存的区域可能比可缓存的区域有更多的限制性行为，但是所允许的行为集合无论如何都不会再变化。然而，一些平台相关的和/或设备相关的可缓存性设置可能有区别。
- 一致性 PMA：在 PMA 中标记为非一致性的内存区域，其内存一致性模型当前是平台相关的和/或设备相关的：加载值公理、原子性公理和进程公理都可能被非一致性内存所违背。然而请注意，一致性内存不需要硬件缓存一致性协议。RISC-V 特权 ISA 规范建议，主内存的硬件不一致区域是不鼓励的，但是内存模型与硬件一致性、软件一致性、由只读性内存导致的隐含一致性、由只有一个拥有权限的代理导致的隐含一致性，或者其它的一致性，相兼容。

- 幂等性 PMA：幂等性 PMA 被用于指定那些加载和/或存储可能有副作用的内存区域，而这反过来被微架构用来决定，例如，预取是否合法。这个区别不影响内存模型。

### A.4.2 I/O 排序

对于 I/O，通常不会应用加载值公理和原子性公理，因为读和写都可能有设备相关的副作用，并可能把由最近的存储所“写”的值以外的值返回到相同的地址。无论如何，下面保留的程序次序规则通常仍然适用于对 I/O 内存的访问：在全局内存次序中，内存访问 $a$ 先于内存访问 $b$，如果在程序次序中 $a$ 先于 $b$，并且满足如下的一个或多个条件：

1. 在如第 十七章中定义的保留的程序次序中，$a$ 先于 $b$，除了只适用于从一个内存操作到另一个内存操作、和从一个 I/O 操作到另一个 I/O 操作的获取和释放次序注释的例外，但从一个内存操作到一个 I/O 操作，或者反过来，则不是。
2. $a$ 和 $b$ 是对于一个 I/O 区域中重叠地址的访问
3. $a$ 和 $b$ 是对于相同的强排序的 I/O 区域的访问
4. $a$ 和 $b$ 是对于 I/O 区域的访问，且关联到被 $a$ 或 $b$ 所访问的 I/O 区域的通道是通道 1
5. $a$ 和 $b$ 是对于关联到相同通道（除了通道 0）的 I/O 区域的访问

注意 FENCE 指令在其前驱集和后继集之中区分了主内存操作和 I/O 操作。为了强制在 I/O 操作和主内存操作之间排序，代码必须使用一个带有 PI、PO、SI 和/或 SO，加上 PR、PW、SR 和/或 SW 的 FENCE。例如，为了强制在一个对主内存的写和一个对设备寄存器的 I/O 写之间排序，需要一个 FENCE W, O 或者更强的指令。

```
sd t0, 0(a0)
fence w,o
sd a0, 0(a1)
```

图 A.17: 有序的内存和 I/O 访问

当一个屏障被实际使用时，实现必须假定设备可能尝试在接收到 MMIO 信号之后立即访问内存，以及来自该设备到内存的后继的内存访问必须观察到所有次序优先于该 MMIO 操作的访问的影响。换句话说，在图 A.17中，假设 `0(a0)` 是在主内存中的，而 `0(a1)` 是在 I/O 内存中的一个设备寄存器的地址。如果设备在接收到 MMIO 写的时候访问 `0(a0)`，那么根据 RVWMO 内存模型的规则，在概念上，该加载必须出现在第一个对 0（a0）的存储之后。在一些实现中，确保这一点的仅有的方法将是要求第一个存储确实在 MMIO 写被发出之前完成。其它实现可能找到了更加激进的方式，同时其它实现仍然可能不需要对 I/O 和主内存访问做任何完全不同的事。无论如何，

RVWMO 内存模型不在这些选项中做区分；它只是简单地提供了一种与实现无关的机制来指定必须强制采用的次序。

许多架构包括了“次序”和“完成”屏障的独立概念，尤其是当它与 I/O 相关时（与常规的主内存相反）。次序屏障简单地确保了内存操作保持有序，而完成屏障确保了，在任何后继的访问变得可见之前，前趋的访问都已经被完成。RISC-V 不会明确地区分次序屏障和完成屏障。反之，这种区分是从 FENCE 位的不同用法简单地推断出来的。

对于遵守 RISC-V Unix 平台规范的实现，I/O 设备和 DMA 操作被要求一致性地访问内存，并且通过强排序的 I/O 通道完成。因此，访问常规的主内存区域，如果该区域同时被外部设备访问，那么也可以使用标准同步机制。不遵守 Unix 平台规范的实现和/或在不会一致性地访问内存的设备中，将需要使用机制（这目前是平台相关的或者设备相关的）来强制一致性。

地址空间中的 I/O 区域应当被考虑为在那些区域的 PMA 中的非可缓存的区域。这种区域可以被 PMA 认为是一致性的，如果它们还没有被任何代理缓存的话。

这一节中的次序保证可能不适用于在 RISC-V 核心和设备之间的平台相关的边界。特别地，经过外部总线（例如，PCIe）发送的 I/O 访问可能在它们到达它们的最终目的地之前被重新排序。在那种情景中，必须根据那些外部设备和总线的平台相关的规则来强制实行排序。

## A.5 代码移植和映射指南

| x86/TSO 操作 | RVWMO 映射 |
|---|---|
| 加载 | `l{b\|h\|w\|d}; fence r,rw` |
| 存储 | `fence rw,w; s{b\|h\|w\|d}` |
| 原子 RMW | `amo<op>.{w\|d}.aqrl` OR<br>`loop: lr.{w\|d}.aq; <op>; sc.{w\|d}.aqrl; bnez loop` |
| 屏障 | `fence rw,rw` |

表 A.2: 从 TSO 操作到 RISC-V 操作的映射

表 A.2提供了一份从 TSO 内存操作到 RISC-V 内存指令的映射。通常的 x86 加载和存储都是固有的 acquire-RCpc 和 release-RCpc 操作：TSO 默认强制所有的加载-加载、加载-存储，和存储-存储排序。因此，在 RVWMO 下，所有的 TSO 加载必须被映射到随后跟有 FENCE R, RW 的加载，而所有的 TSO 存储必须被映射到跟在存储之后的 FENCE RW, W。TSO 原子读-修改-写和使用 LOCK 前缀的 x86 指令是完全排序的，并且可以或者通过一个同时设置了 *aq* 和 *rl* 的 AMO 实现，或者通过一个设置了 *aq* 的 LR、上述提及的算数操作、一个同时设置了 *aq* 和 *rl* 的 SC，还

有一个检查成功条件的条件分支来实现。在最后一种情况中，在 LR 上的 *rl* 注释（由于不明的原因）是多余的，并且可以被省略。

表 A.2的替代方案也是可行的。一个 TSO 存储可以被映射到设置了 *rl* 的 AMOSWAP 上。然而，由于 RVWMO PPO 规则 3禁止值从 AMO 到后继加载的转发，对存储使用 AMOSWAP 可能对性能产生负面的影响。一个 TSO 加载可以使用设置了 *aq* 的 LR 来映射：所有的这种 LR 指令将是无配对的，但是事实上本身并不排除使用 LR 进行加载。然而，再次强调，这种映射也可能对性能有负面影响，如果它把比最初意图更多的压力放在了保留机制上的话。

| Power 操作 | RVWMO 映射 |
|---|---|
| 加载 | `l{b\|h\|w\|d}` |
| 加载-保留 | `lr.{w\|d}` |
| 存储 | `s{b\|h\|w\|d}` |
| 存储-条件 | `sc.{w\|d}` |
| `lwsync` | `fence.tso` |
| `sync` | `fence rw,rw` |
| `isync` | `fence.i; fence r,r` |

表 A.3: 从 Power 操作到 RISC-V 操作的映射

表 A.3提供了一份从 Power 内存操作到 RISC-V 内存指令的映射。Power ISYNC 在 RISC-V 上映射到一个后跟有 FENCE R, R 的 FENCE.I 上；后一个屏障是必须的，因为 ISYNC 被用于定义一种“控制 + 控制屏障”的依赖，而它在 RVWMO 中是不存在的。

| ARM 操作 | RVWMO 映射 |
|---|---|
| 加载 | `l{b\|h\|w\|d}` |
| 加载-获取 | `fence rw, rw; l{b\|h\|w\|d}; fence r,rw` |
| 加载-独占 | `lr.{w\|d}` |
| 加载-获取-独占 | `lr.{w\|d}.aqrl` |
| 存储 | `s{b\|h\|w\|d}` |
| 存储-释放 | `fence rw,w; s{b\|h\|w\|d}` |
| 存储-独占 | `sc.{w\|d}` |
| 存储-释放-独占 | `sc.{w\|d}.rl` |
| `dmb` | `fence rw,rw` |
| `dmb.ld` | `fence r,rw` |
| `dmb.st` | `fence w,w` |
| `isb` | `fence.i; fence r,r` |

表 A.4: 从 ARM 操作到 RISC-V 操作的映射

表 A.4提供了一份从 ARM 内存操作到 RISC-V 内存指令的映射。由于 RISC-V 目前没有带 *aq* 或 *rl* 注释的不修饰的加载和存储的操作码，ARM 加载-获取和存储-释放操作应当代之以使用屏障来映射。而且，为了强制采用存储-释放到加载-获取的次序，在存储-释放和加载-获取之间必须有一个 FENCE RW, RW；表 A.4通过把屏障放置在每个获取操作之前，强制采用了这个次序。ARM 的 load-exclusive 和 store-exclusive 指令可以类似地映射到与它们对等的 RISC-V LR 和 SC 上，但是并非把 FENCE RW, RW 放在设置了 *aq* 的 LR 之前，而是我们也简单地用设置 *rl* 代替。ARM ISB 在 RISC-V 上映射到后跟有 FENCE R, R 的 FENCE.I 上，类似于 ISYNC 映射 Power 的方式。

| Linux 操作 | RVWMO 映射 |
|---|---|
| `smp_mb()` | `fence rw,rw` |
| `smp_rmb()` | `fence r,r` |
| `smp_wmb()` | `fence w,w` |
| `dma_rmb()` | `fence r,r` |
| `dma_wmb()` | `fence w,w` |
| `mb()` | `fence iorw,iorw` |
| `rmb()` | `fence ri,ri` |
| `wmb()` | `fence wo,wo` |
| `smp_load_acquire()` | `l{b\|h\|w\|d}; fence r,rw` |
| `smp_store_release()` | `fence.tso; s{b\|h\|w\|d}` |
| Linux 构造 | RVWMO AMO 映射 |
| `atomic_<op>_relaxed` | `amo<op>.{w\|d}` |
| `atomic_<op>_acquire` | `amo<op>.{w\|d}.aq` |
| `atomic_<op>_release` | `amo<op>.{w\|d}.rl` |
| `atomic_<op>` | `amo<op>.{w\|d}.aqrl` |
| Linux 构造 | RVWMO LR/SC 映射 |
| `atomic_<op>_relaxed` | `loop: lr.{w\|d}; <op>; sc.{w\|d}; bnez loop` |
| `atomic_<op>_acquire` | `loop: lr.{w\|d}.aq; <op>; sc.{w\|d}; bnez loop` |
| `atomic_<op>_release` | `loop: lr.{w\|d}; <op>; sc.{w\|d}.aqrl*; bnez loop` OR `fence.tso; loop: lr.{w\|d}; <op>; sc.{w\|d}*; bnez loop` |
| `atomic_<op>` | `loop: lr.{w\|d}.aq; <op>; sc.{w\|d}.aqrl; bnez loop` |

表 A.5: 从 Linux 内存原语到 RISC-V 原语的映射。其它的构造（例如自旋锁）应当相应地服从。非一致性 DMA 的平台或设备可能需要额外的同步（例如缓存冲刷或无效性机制）；当前任何这样的额外同步都将是设备相关的。

表 A.5提供了一份 Linux 内存排序宏到 RISC-V 内存指令的映射。Linux 屏障 `dma_rmb()` 和 `dma_wmb()` 分别映射到 FENCE R, R 和 FENCE W, W，因为 RISC-V Unix 平台需要一致性 DMA，但是在非一致性 DMA 平台上将分别被映射到 FENCE RI, RI 和 FENCE WO, WO。非一致性 DMA 的平台也可以要求一种"缓存行可以被冲刷和/或无效化"的机制。这种机制将是设备相关的、和/或在未来对 ISA 的扩展中标准化的。

Linux 对于释放操作的映射可能看起来比必要的更强，但是需要这些映射去覆盖某些 Linux 需要比更直观的映射将提供的更强的次序的情况。特别地，在本文正在被编写的时候，Linux 正在积极地讨论，在一个临界区中的访问和相同硬件线程中的一个后继的临界区中的访问之间，是否需要加载-加载、加载-存储，和存储-存储的次序，并由相同的同步对象进行保护。不是所有的 FENCE RW, W/FENCE R, RW 映射和 *aq*/*rl* 映射的组合都能提供这种次序。围绕这个问题有这样一些方法，包括：

1. 永远使用 FENCE RW, W/FENCE R, RW，并且永远不使用 *aq*/*rl*。这是足够的，但是不可取，因为它违背了 *aq*/*rl* 修饰符的目的。
2. 永远使用 *aq*/*rl*，并永远不使用 FENCE RW, W/FENCE R, RW。这目前不会起作用，因为缺少带有 *aq* 和 *rl* 修饰符的加载和存储操作码。
3. 加强释放操作的映射，使得它们将在现有的任何种类的获取映射中强制采用充分的次序。这是当前推荐的方案，该方案展示在表 A.5中。

```
                                RVWMO 映射：
                                (a) lw           a0, 0(s0)
Linux 代码：                    (b) fence.tso  // vs. fence rw,w
(a)  int r0 = *x;               (c) sd           x0,0(s1)
(bc) spin_unlock(y, 0);             ...
     ...                            loop:
     ...                        (d) amoswap.d.aq a1,t1,0(s1)
(d)  spin_lock(y);                  bnez         a1,loop
(e)  int r1 = *z;               (e) lw           a2,0(s2)
```

图 A.18: Linux 中临界区之间的次序

例如，Linux 社区当前正在讨论临界区次序规则，该规则将要求图 A.18中的 (a) 被排序在 (e) 之前。如果确实那样要求，那么把 (b) 映射为 FENCE RW, W 将是不充分的。也就是说，随着 Linux 内核内存模型的演化，这些映射也将随之变化。

表 A.6提供了一份 C11/C++11 原子操作到 RISC-V 内存指令的映射。如果引入了带有 *aq* 和 *rl* 修饰符的加载和存储操作码，那么表 A.7中的映射就将足够了。然而要注意，只有当原子的

| C/C++ 构造 | RVWMO 映射 |
|---|---|
| Non-atomic load | l{b\|h\|w\|d} |
| atomic_load(memory_order_relaxed) | l{b\|h\|w\|d} |
| atomic_load(memory_order_acquire) | l{b\|h\|w\|d}; fence r,rw |
| atomic_load(memory_order_seq_cst) | fence rw,rw; l{b\|h\|w\|d}; fence r,rw |
| 非原子性存储 | s{b\|h\|w\|d} |
| atomic_store(memory_order_relaxed) | s{b\|h\|w\|d} |
| atomic_store(memory_order_release) | fence rw,w; s{b\|h\|w\|d} |
| atomic_store(memory_order_seq_cst) | fence rw,w; s{b\|h\|w\|d} |
| atomic_thread_fence(memory_order_acquire) | fence r,rw |
| atomic_thread_fence(memory_order_release) | fence rw,w |
| atomic_thread_fence(memory_order_acq_rel) | fence.tso |
| atomic_thread_fence(memory_order_seq_cst) | fence rw,rw |
| C/C++ 构造 | RVWMO AMO 映射 |
| atomic_<op>(memory_order_relaxed) | amo<op>.{w\|d} |
| atomic_<op>(memory_order_acquire) | amo<op>.{w\|d}.aq |
| atomic_<op>(memory_order_release) | amo<op>.{w\|d}.rl |
| atomic_<op>(memory_order_acq_rel) | amo<op>.{w\|d}.aqrl |
| atomic_<op>(memory_order_seq_cst) | amo<op>.{w\|d}.aqrl |
| C/C++ 构造 | RVWMO LR/SC 映射 |
| atomic_<op>(memory_order_relaxed) | loop: lr.{w\|d}; <op>; sc.{w\|d}; bnez loop |
| atomic_<op>(memory_order_acquire) | loop: lr.{w\|d}.aq; <op>; sc.{w\|d}; bnez loop |
| atomic_<op>(memory_order_release) | loop: lr.{w\|d}; <op>; sc.{w\|d}.rl; bnez loop |
| atomic_<op>(memory_order_acq_rel) | loop: lr.{w\|d}.aq; <op>; sc.{w\|d}.rl; bnez loop |
| atomic_<op>(memory_order_seq_cst) | loop: lr.{w\|d}.aqrl; <op>; sc.{w\|d}.rl; bnez loop |

表 A.6: 从 C/C++ 原语到 RISC-V 原语的映射。

atomic_<op>(内存 _ 次序 _seq_cst) 被使用一个同时设置了 *aq* 和 *rl* 的 LR 映射时，这两个映射才会正确地互通。

| C/C++ 构造 | RVWMO 映射 |
|---|---|
| 非原子性加载 | `l{b\|h\|w\|d}` |
| `atomic_load(memory_order_relaxed)` | `l{b\|h\|w\|d}` |
| `atomic_load(memory_order_acquire)` | `l{b\|h\|w\|d}.aq` |
| `atomic_load(memory_order_seq_cst)` | `l{b\|h\|w\|d}.aq` |
| 非原子性存储 | `s{b\|h\|w\|d}` |
| `atomic_store(memory_order_relaxed)` | `s{b\|h\|w\|d}` |
| `atomic_store(memory_order_release)` | `s{b\|h\|w\|d}.rl` |
| `atomic_store(memory_order_seq_cst)` | `s{b\|h\|w\|d}.rl` |
| `atomic_thread_fence(memory_order_acquire)` | `fence r,rw` |
| `atomic_thread_fence(memory_order_release)` | `fence rw,w` |
| `atomic_thread_fence(memory_order_acq_rel)` | `fence.tso` |
| `atomic_thread_fence(memory_order_seq_cst)` | `fence rw,rw` |

| C/C++ 构造 | RVWMO AMO 映射 |
|---|---|
| `atomic_<op>(memory_order_relaxed)` | `amo<op>.{w\|d}` |
| `atomic_<op>(memory_order_acquire)` | `amo<op>.{w\|d}.aq` |
| `atomic_<op>(memory_order_release)` | `amo<op>.{w\|d}.rl` |
| `atomic_<op>(memory_order_acq_rel)` | `amo<op>.{w\|d}.aqrl` |
| `atomic_<op>(memory_order_seq_cst)` | `amo<op>.{w\|d}.aqrl` |

| C/C++ 构造 | RVWMO LR/SC 映射 |
|---|---|
| `atomic_<op>(memory_order_relaxed)` | `lr.{w\|d}; <op>; sc.{w\|d}` |
| `atomic_<op>(memory_order_acquire)` | `lr.{w\|d}.aq; <op>; sc.{w\|d}` |
| `atomic_<op>(memory_order_release)` | `lr.{w\|d}; <op>; sc.{w\|d}.rl` |
| `atomic_<op>(memory_order_acq_rel)` | `lr.{w\|d}.aq; <op>; sc.{w\|d}.rl` |
| `atomic_<op>(memory_order_seq_cst)` | `lr.{w\|d}.aq*; <op>; sc.{w\|d}.rl` |

为了能与表 A.6中的各个代码映射相互操作，* 必须是 `lr.{w|d}.aqrl`

表 A.7: 假设的从 C/C++ 原语到 RISC-V 原语的映射，如果引入了原生的加载-获取和存储释放操作码的话。

任何 AMO 可以通过一个 LR/SC 对来模拟，但是必须注意确保任何源自于 LR 的 PPO 次序也是源自于 SC 的，并且任何在 SC 终止的 PPO 次序也会使得在 LR 处终止。例如，LR 必须也要服从任何 AMO 拥有的数据依赖，使得加载操作否则将没有任何数据依赖的概念。类似地，相同硬件线程中的其它地方的一个 FENCE R, R 的影响也必须适用于 SC，否则将不会遵从该屏障。模拟器可以系通过简单地把 AMO 映射到 `lr.aq; <op>; sc.aqrl` 上来达成这个效果，与其它地方的用于全排序原子性的映射相匹配。

这些 C11/C++11 映射需要平台为所有内存提供下列物理内存属性（正如 RISC-V 特权 ISAS 中定义的那样）：

- 主内存
- 连贯性
- AMOArithmetic
- RsrvEventual

具有不同属性的平台可能需要不同的映射，或者需要平台相关的 SW（例如，内存映射 I/O）。

## A.6 实现指南

RVWMO 和 RVTSO 内存模型绝不排除微架构采用复杂的推测技术或其它形式的优化来提供更高的性能。模型也不采用任何需要使用任何一个特定的缓存层次的需求，甚至一点也不使用缓存一致性协议。相反，这些模型只指定了可以暴露给软件的行为。微架构可以自由地使用任何流水线设计，任何一致性或非一致性缓存层次，任何片上互连，等等，只要设计只认可满足内存模型规则的执行。也就是说，为了帮助人们理解内存模型的实际实现，本节中我们提供了一些关于架构师和编程人员应当如何解释模型的规则的指导。

RVWMO 和 RVTSO 都是多重拷贝原子性（或者“其它多重拷贝原子性”）的：任何对一个硬件线程（除了最初发出它的那个硬件线程）可见的存储的值，在概念上必须也对系统中的所有的其它硬件线程可见。换句话说，硬件线程可以从它们自己的先前的存储进行转发，时机在那些存储变得对所有的硬件线程全局可见之前，但是提前在硬件线程之间进行转发是不被允许的。多重拷贝原子性可以通过多种方式被采用。它可能由于缓存和存储缓冲区的物理设计而固有地存在，它可以通过一种单一写者/多重读者的缓存一致性协议来采用，或者它可以由于某些其它机制而存在。

尽管多重拷贝原子性的确在微架构上采用了一些限制，但是它是保持内存模型免于变得极度复杂的关键属性之一。例如，一个硬件线程不可以从一个邻居硬件线程的私有存储缓冲区合法地转发一个值（当然，除非它这么做，不会有新的非法行为变得架构可见）。一个缓存一致性协议也不能在其已经无效化了所有来自其它缓存的更旧的拷贝之前，从一个硬件线程向另一个硬件线程转发一个值。当然，微架构可以（并且高性能实现可能会）通过推测或其它的优化来暗中违背这些规则，只要任何不合规的行为都不会暴露给编程人员。

作为一份解释 RVWMO 中的 PPO 规则的粗略指南，从软件的角度，我们期待看到以下情形：

- 编程人员将有规律地和积极地使用 PPO 规则1和4 – 8。

- 专业的编程人员将使用 PPO 规则9 – 11来加速重要数据结构的关键路径。
- 即使是专业的编程人员也将很少直接使用 PPO 规则2 – 3和12 – 13，如果它们有的话。

从硬件的角度，我们也希望看到下列情况：

- PPO 规则1和3 – 6反映了好理解的规则，应当不会给架构师带来什么惊喜。
- PPO 规则2反映了一个自然的和常见的硬件优化，但那是一个非常微妙的优化，因此值得仔细地复查。
- PPO 规则7可能对于架构师来说不会立刻感到显然，但是它是标准内存模型的需求。
- 加载值公理、原子性公理，和 PPO 规则8 – 13反映了大多数硬件实现将自然地采用的规则，除非它们包含了极度的优化。当然，尽管如此，实现应当确保复查这些规则。硬件也必须确保句法依赖不会“被优化掉”。

架构可以自由地实现任何的内存模型规则，正如它们选择的那样保守。例如，一个硬件实现可以选择做到下列中的任何或者所有：

- 无论各位实际如何设置，把所有的屏障都按它们是 FENCE RW, RW（或者 FENCE IORW, IORW，如果涉及了 I/O）来解释
- 把所有的带 PW 和 SR 的屏障都按它们是 FENCE RW, RW（或者 FENCE IORW, IORW，如果涉及了 I/O）来实现，因为无论如何，带 SR 的 PW 都是四个可能的主内存次序组件中最昂贵的
- 像 A.5节中描述的那样模拟 *aq* 和 *rl*
- 强制所有的相同地址的加载-加载次序，即使存在在诸如“fri-rfi”和“RSW”等式样
- 禁止任何从存储缓冲区中的存储到相同地址的后继的 AMO 或 LR 的值的转发
- 禁止任何从存储缓冲区中的 AMO 或 SC 到相同地址的后继的加载的值的转发
- 在所有内存访问上实现 TSO，并忽略任何不包括 PW 和 SR 次序的主内存屏障（例如，就像 Ztso 实现会做的那样）
- 把所有的原子性实现为 RCsc 或者甚至是全排序的，无论注释如何

实现了 RVTSO 的架构可以安全地做到下列事情：

- 忽略所有不同时具有 PW 和 SR 的屏障（除非该屏障也排序了 I/O）

- 忽略所有除了规则4至7之外的 PPO 规则，因为在 RVTSO 的假设下，剩下的规则与其它的 PPO 规则是多余的

其它的一般注意事项：

- 从内存模型的观点来看，静默存储（例如，写入与内存位置已经存在的值相同的值的存储）的行为与任何其它的存储相像。类似地，不实际改变内存中的值的 AMO（例如，一个 *rs2* 中的值比内存中当前的值更小的 AMOMAX）仍然在语义上被认为是存储操作。尝试实现静默存储的微架构必须小心地确保仍然服从内存模型，特别是在诸如 RSW（章节 A.3.5）的情形中，它们往往与静默存储不兼容。
- 写可以被融合（即，对相同地址的两个连续的写可以被融合）或归并（即，对于相同地址的两个背靠背的写，可以省略较早的那个），只要结果行为不会违反内存模型语义。

写归并的问题可以从下面的例子来理解：

| | Hart 0 | | Hart 1 |
|---|---|---|---|
| | `li t1, 3` | | `li t3, 2` |
| | `li t2, 1` | | |
| (a) | `sw t1,0(s0)` | (d) | `lw a0,0(s1)` |
| (b) | `fence w, w` | (e) | `sw a0,0(s0)` |
| (c) | `sw t2,0(s1)` | (f) | `sw t3,0(s0)` |

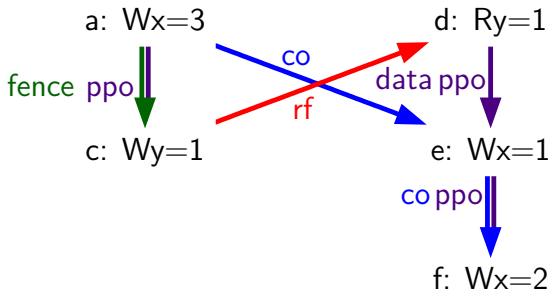



图 A.19: 写归并石蕊测试，允许的执行。

如前所写，如果加载 (d) 读取值 1，然后 (a) 在全局内存次序中必须先于 (f)：

- 由于规则 2，在全局内存次序中，(a) 先于 (c)
- 由于加载值公理，在全局内存次序中，(c) 先于 (d)
- 由于规则 7，在全局内存次序中，(d) 先于 (e)
- 由于规则 1，在全局内存次序中，(e) 先于 (f)

换句话说，地址在 `s0` 中的内存位置的最终取值必定是 2（由存储 (f) 写入的值），而不能是 3（由存储 (a) 写入的值）。

一个非常激进的微架构可能会错误地决定丢弃 (e)，因为 (f) 取代了它，而这可能反过来导致微架构打破 (d) 和 (f) 之间现在已消除的依赖（并因此也会打破 (a) 和 (f) 之间的）。这将违反内存模型规则，因而它是被禁止的。写归并可能在其他情形中是合法的，如果，例如在 (d) 和 (e) 之间没有数据依赖的话。

### A.6.1 未来可能的扩展

我们希望任何或所有的下列可能的未来扩展都将与 RVWMO 内存模型兼容：

- 'V'向量 ISA 扩展
- 'J'JIT 扩展
- 用于设置了 *aq* 和 *rl* 的加载和存储操作码的原生编码
- 限制到特定地址的屏障
- 缓存的写回/冲刷/无效化/等等……指令

## A.7 已知问题

### A.7.1 混合尺寸的 RSW

| Hart 0 | | Hart 1 | |
|---|---|---|---|
| | li t1, 1 | | li t1, 1 |
| (a) | lw a0,0(s0) | (d) | lw a1,0(s1) |
| (b) | fence rw,rw | (e) | amoswap.w.rl a2,t1,0(s2) |
| (c) | sw t1,0(s1) | (f) | ld a3,0(s2) |
| | | (g) | lw a4,4(s2) |
| | | | xor a5,a4,a4 |
| | | | add s0,s0,a5 |
| | | (h) | sw a2,0(s0) |

输出结果：a0=1, a1=1, a2=0, a3=1, a4=0

图 A.20: 混合尺寸的差异（被公理模型所允许，被操作模型所禁止）

在混合尺寸的 RSW 变体家族中，操作规范和公理规范之间有一种已知的差异性，显示在表 A.20—A.22中。为了解决这个差异性，我们可以选择添加一些像是下列新的 PPO 规则的东西：在保留的程序次序中（并因此也在全局内存次序中），内存操作 $a$ 先于内存操作 $b$，如果在程

| | Hart 0 | | Hart 1 |
|---|---|---|---|
| | li t1, 1 | | li t1, 1 |
| (a) | lw a0,0(s0) | (d) | ld a1,0(s1) |
| (b) | fence rw,rw | (e) | lw a2,4(s1) |
| (c) | sw t1,0(s1) | | xor a3,a2,a2 |
| | | | add s0,s0,a3 |
| | | (f) | sw a2,0(s0) |

输出结果：a0=0, a1=1, a2=0

图 A.21: 混合尺寸的差异（被公理模型所允许，被操作模型所禁止）

| | Hart 0 | | Hart 1 |
|---|---|---|---|
| | li t1, 1 | | li t1, 1 |
| (a) | lw a0,0(s0) | (d) | sw t1,4(s1) |
| (b) | fence rw,rw | (e) | ld a1,0(s1) |
| (c) | sw t1,0(s1) | (f) | lw a2,4(s1) |
| | | | xor a3,a2,a2 |
| | | | add s0,s0,a3 |
| | | (g) | sw a2,0(s0) |

Outcome: a0=1, a1=0x100000001, a1=1

图 A.22: 混合尺寸的差异（被公理模型所允许，被操作模型所禁止）

序次序中 a 先于 b，a 和 b 都访问常规主内存（而不是 I/O 区域），$a$ 是一个加载，$b$ 是一个存储，那么在 $a$ 和 $b$ 之间存在一个加载 $m$，$a$ 和 $m$ 都读取一个位 $x$，在 $a$ 和 $m$ 之间没有写 $x$ 的存储，且在 PPO 中 $m$ 先于 $b$。换句话说，在 herd 语法中，我们可以选择向 PPO 添加“`(po-loc & rsw);ppo;[W]`”。许多实现已经自然地采用了这种次序。严格来说，即使这个规则不是官方的，尽管如此，我们也推荐实现者采用它，以便确保未来可能把这个规则添加到 RVWMO 的向前兼容性。

# 附录 B　形式化的内存模型规范（0.1 版本）

为了便于对 RVWMO 的形式化分析，这章使用不同的工具和建模方法呈现了一组形式化。任何差异性都是无意的；希望这些模型确实描述了相同的合法行为集合。

这个附录应当被视为注释；所有的规范材料都提供在第 十七章和 ISA 规范主体的剩余部分中。第 A.7节中列出了所有当前已知的差异性。任何其它的差异性都是无意的。

## B.1 Alloy 中的形式公理规范

我们在 Alloy（http://alloy.mit.edu）中呈现了 RVWMO 内存模型的一种形式规范。这个模型可以从https://github.com/daniellustig/riscv-memory-model在线获得。

该线上材料也包含了一些石蕊测试和一些关于 Alloy 可以怎样被用于对第 A.5节中的一些映射进行模型检查的例子。

```
////////////////////////////////////////////////////////////////////////////////
// =RVWMO PPO=

// 保留的程序次序
fun ppo : Event->Event {
  // same-address ordering
  po_loc :> Store
  + rdw
  + (AMO + StoreConditional) <: rfi

  // 显式同步
  + ppo_fence
  + Acquire <: ^po :> MemoryEvent
  + MemoryEvent <: ^po :> Release
  + RCsc <: ^po :> RCsc
  + pair

  // 句法依赖
  + addrdep
  + datadep
  + ctrldep :> Store

  // 管道依赖
  + (addrdep+datadep).rfi
  + addrdep.^po :> Store
}

// 全局内存次序尊重保留的程序次序
fact { ppo in ^gmo }
```

图 B.1: Alloy 中的形式化 RVWMO 内存模型（1/5：PPO）

```
////////////////////////////////////////////////////////////////////////////////
// =RVWMO 公理=

// 加载值公理
fun candidates[r: MemoryEvent] : set MemoryEvent {
  (r.~^gmo & Store & same_addr[r]) // 在gmo中写前趋r
  + (r.^~po & Store & same_addr[r]) // 在po中写前趋r
}

fun latest_among[s: set Event] : Event { s - s.~^gmo }

pred LoadValue {
  all w: Store | all r: Load |
    w->r in rf <=> w = latest_among[candidates[r]]
}

// 原子性公理
pred Atomicity {
  all r: Store.~pair |              // 从lr开始，
    no x: Store & same_addr[r] |  // 对于相同的地址，没有存储x
      x not in same_hart[r]       // 使得x来自不同的硬件线程，
      and x in r.~rf.^gmo         // 在gmo中x跟着(存储r的“读从”)，
      and r.pair in x.^gmo        // 以及在gmo中，r跟着x
}

// 进程公理被隐去：Alloy只考虑有限的执行

pred RISCV_mm { LoadValue and Atomicity /* and Progress */ }
```

图 B.2: Alloy 中的形式化 RVWMO 内存模型（2/5：公理）

```
////////////////////////////////////////////////////////////////////////////////
// 内存的基础模型

sig Hart {  // 硬件线程
  start : one Event
}
sig Address {}
abstract sig Event {
  po: lone Event // 程序次序
}

abstract sig MemoryEvent extends Event {
  address: one Address,
  acquireRCpc: lone MemoryEvent,
  acquireRCsc: lone MemoryEvent,
  releaseRCpc: lone MemoryEvent,
  releaseRCsc: lone MemoryEvent,
  addrdep: set MemoryEvent,
  ctrldep: set Event,
  datadep: set MemoryEvent,
  gmo: set MemoryEvent,  // 全局内存次序
  rf: set MemoryEvent
}
sig LoadNormal extends MemoryEvent {} // l{b|h|w|d}
sig LoadReserve extends MemoryEvent { // lr
  pair: lone StoreConditional
}
sig StoreNormal extends MemoryEvent {}       // s{b|h|w|d}
// 模型中所有的StoreConditionals 都假定是成功的
sig StoreConditional extends MemoryEvent {}  // sc
sig AMO extends MemoryEvent {}               // amo
sig NOP extends Event {}

fun Load : Event { LoadNormal + LoadReserve + AMO }
fun Store : Event { StoreNormal + StoreConditional + AMO }

sig Fence extends Event {
  pr: lone Fence, // 操作位
  pw: lone Fence, // 操作位
  sr: lone Fence, // 操作位
  sw: lone Fence  // 操作位
}
sig FenceTSO extends Fence {}

/* Alloy编码细节：操作码位要么被设置（被编码，例如 as f.pr in iden），
 * 要么不被设置（f.pr not in iden）。这些位不能被用作其它任何用途 */
fact { pr + pw + sr + sw in iden }
// 对排序注释也是类似的
fact { acquireRCpc + acquireRCsc + releaseRCpc + releaseRCsc in iden }
// 不要试图通过pr/pw/sr/sw来编码FenceTSO；只把它当做is那样使用
fact { no FenceTSO.(pr + pw + sr + sw) }
```

```
////////////////////////////////////////////////////////////////////////////
// = 基本模型规则 =

// 次序注释组
fun Acquire : MemoryEvent { MemoryEvent.acquireRCpc + MemoryEvent.acquireRCsc }
fun Release : MemoryEvent { MemoryEvent.releaseRCpc + MemoryEvent.releaseRCsc }
fun RCpc : MemoryEvent { MemoryEvent.acquireRCpc + MemoryEvent.releaseRCpc }
fun RCsc : MemoryEvent { MemoryEvent.acquireRCsc + MemoryEvent.releaseRCsc }

// 没有像store - acquire或者load - release那样的东西，除非两者结合
fact { Load & Release in Acquire }
fact { Store & Acquire in Release }

// FENCE PPO
fun FencePRSR : Fence { Fence.(pr & sr) }
fun FencePRSW : Fence { Fence.(pr & sw) }
fun FencePWSR : Fence { Fence.(pw & sr) }
fun FencePWSW : Fence { Fence.(pw & sw) }

fun ppo_fence : MemoryEvent->MemoryEvent {
    (Load  <: ^po :> FencePRSR).(^po :> Load)
  + (Load  <: ^po :> FencePRSW).(^po :> Store)
  + (Store <: ^po :> FencePWSR).(^po :> Load)
  + (Store <: ^po :> FencePWSW).(^po :> Store)
  + (Load  <: ^po :> FenceTSO) .(^po :> MemoryEvent)
  + (Store <: ^po :> FenceTSO) .(^po :> Store)
}

// 辅助定义
fun po_loc : Event->Event { ^po & address.~address }
fun same_hart[e: Event] : set Event { e + e.^~po + e.^po }
fun same_addr[e: Event] : set Event { e.address.~address }

// 初始化存储
fun NonInit : set Event { Hart.start.*po }
fun Init : set Event { Event - NonInit }
fact { Init in StoreNormal }
fact { Init->(MemoryEvent & NonInit) in ^gmo }
fact { all e: NonInit | one e.*~po.~start }  // 各事件都确实地在one硬件线程中
fact { all a: Address | one Init & a.~address } // one初始化各地址的存储
fact { no Init <: po and no po :> Init }
```

图 B.4: Alloy 中的形式化 RVWMO 内存模型（4/5：基础模型规则）

```
// po
fact { acyclic[po] }

// gmo
fact { total[^gmo, MemoryEvent] } // gmo是在所有MemoryEvents上的总次序

//rf
fact { rf.~rf in iden } // 每个读返回唯一写的值
fact { rf in Store <: address.~address :> Load }
fun rfi : MemoryEvent->MemoryEvent { rf & (*po + *~po) }

//dep
fact { no StoreNormal <: (addrdep + ctrldep + datadep) }
fact { addrdep + ctrldep + datadep + pair in ^po }
fact { datadep in datadep :> Store }
fact { ctrldep.*po in ctrldep }
fact { no pair & (^po :> (LoadReserve + StoreConditional)).^po }
fact { StoreConditional in LoadReserve.pair } // 假设所有的SC都成功了

// rdw
fun rdw : Event->Event {
  (Load <: po_loc :> Load)  // start with all same_address load-load pairs,
  - (~rf.rf)                // subtract pairs that read from the same store,
  - (po_loc.rfi)            // and subtract out "fri-rfi" patterns
}

// 过滤出冗余的实例和/或可视化效果
fact { no gmo & gmo.gmo } // 保持可视化效果整洁
fact { all a: Address | some a.~address }

////////////////////////////////////////////////////////////////////////////////
// =可选：操作码编码约束=

// 神圣的屏障列表
fact { Fence in
  Fence.pr.sr
  + Fence.pw.sw
  + Fence.pr.pw.sw
  + Fence.pr.sr.sw
  + FenceTSO
  + Fence.pr.pw.sr.sw
}

pred restrict_to_current_encodings {
  no (LoadNormal + StoreNormal) & (Acquire + Release)
}

////////////////////////////////////////////////////////////////////////////////
// =Alloy捷径 =
pred acyclic[rel: Event->Event] { no iden & ^rel }
pred total[rel: Event->Event, bag: Event] {
```

## B.2 Herd 中的形式公理规范

工具 herd 把一个内存模型和一个石蕊测试作为输入，并在内存模型顶端模拟测试的执行。内存模型使用领域专用语言 Cat 写成。这节提供了两个 RVWMO 的 Cat 内存模型。第一个模型，图 B.7，对于一个 Cat 模型而言尽可能多地遵循了全局内存次序、第 十七章、RVWMO 的定义。第二个模型，图 B.8，是一个等价的、更加有效的的、基于部分次序的 RVWMO 模型。

模拟器 herd 是 DIY 工具套件的一部分——软件和文件见 http://diy.inria.fr 。模型和更多内容可以从 urlhttp://diy.inria.fr/cats7/riscv/ 在线获得。

```
(*************)
(* Utilities *)
(*************)

(* All fence relations *)
let fence.r.r = [R];fencerel(Fence.r.r);[R]
let fence.r.w = [R];fencerel(Fence.r.w);[W]
let fence.r.rw = [R];fencerel(Fence.r.rw);[M]
let fence.w.r = [W];fencerel(Fence.w.r);[R]
let fence.w.w = [W];fencerel(Fence.w.w);[W]
let fence.w.rw = [W];fencerel(Fence.w.rw);[M]
let fence.rw.r = [M];fencerel(Fence.rw.r);[R]
let fence.rw.w = [M];fencerel(Fence.rw.w);[W]
let fence.rw.rw = [M];fencerel(Fence.rw.rw);[M]
let fence.tso =
  let f = fencerel(Fence.tso) in
  ([W];f;[W]) | ([R];f;[M])

let fence =
  fence.r.r | fence.r.w | fence.r.rw |
  fence.w.r | fence.w.w | fence.w.rw |
  fence.rw.r | fence.rw.w | fence.rw.rw |
  fence.tso

(* Same address, no W to the same address in-between *)
let po-loc-no-w = po-loc \ (po-loc?;[W];po-loc)
(* Read same write *)
let rsw = rf^-1;rf
(* Acquire, or stronger  *)
let AQ = Acq|AcqRel
(* Release or stronger *)
and RL = RelAcqRel
(* All RCsc *)
let RCsc = Acq|Rel|AcqRel
(* Amo events are both R and W, relation rmw relates paired lr/sc *)
let AMO = R & W
let StCond = range(rmw)

(*************)
(* ppo rules *)
(*************)

(* Overlapping-Address Orderings *)
let r1 = [M];po-loc;[W]
and r2 = ([R];po-loc-no-w;[R]) \ rsw
and r3 = [AMO|StCond];rfi;[R]
(* Explicit Synchronization *)
and r4 = fence
and r5 = [AQ];po;[M]
and r6 = [M];po;[RL]
and r7 = [RCsc];po;[RCsc]
```

```
Total

(* Notice that herd has defined its own rf relation *)

(* Define ppo *)
include "riscv-defs.cat"

(*********************************)
(* Generate global memory order *)
(*********************************)

let gmo0 = (* precursor: ie build gmo as an total order that include gmo0 *)
  loc & (W\FW) * FW | # Final write after any write to the same location
  ppo |               # ppo compatible
  rfe                 # includes herd external rf (optimization)

(* Walk over all linear extensions of gmo0 *)
with  gmo from linearizations(M\IW,gmo0)

(* Add initial writes upfront -- convenient for computing rfGMO *)
let gmo = gmo | loc & IW * (M\IW)

(**********)
(* Axioms *)
(**********)

(* Compute rf according to the load value axiom, aka rfGMO *)
let WR = loc & ([W];(gmo|po);[R])
let rfGMO = WR \ (loc&([W];gmo);WR)

(* Check equality of herd rf and of rfGMO *)
empty (rf\rfGMO)|(rfGMO\rf) as RfCons

(* Atomicity axiom *)
let infloc = (gmo & loc)^-1
let inflocext = infloc & ext
let winside  = (infloc;rmw;inflocext) & (infloc;rf;rmw;inflocext) & [W]
empty winside as Atomic
```

图 B.7: riscv.cat，RVWMO 内存模型的一个 Herd 版本（2/3）

```
Partial

(***************)
(* Definitions *)
(***************)

(* Define ppo *)
include "riscv-defs.cat"

(* Compute coherence relation *)
include "cos-opt.cat"

(**********)
(* Axioms *)
(**********)

(* Sc per location *)
acyclic co|rf|fr|po-loc as Coherence

(* Main model axiom *)
acyclic co|rfe|fr|ppo as Model

(* Atomicity axiom *)
empty rmw & (fre;coe) as Atomic
```

图 B.8: `riscv.cat`，RVWMO 内存模型的一种备选的 Herd 表示（3/3）

## B.3 一个内存操作模型

这是操作风格中对 RVWMO 内存模型的一个备选的表示。它旨在确实地承认与公理化表示相同的扩展行为：对于任何给定的程序，承认一个执行，当且仅当公理化表示也允许它。

公理化表示被定义为关于完整候选执行的谓词。相比之下，这种操作表示具有一种抽象的微架构风味：它被表示为一个状态机，其中状态是对于硬件机器状态的一种抽象表示，并且带有显式的乱序执行和推测性执行（但是从更实现相关的微架构细节中抽象出来，例如寄存器重命名、存储缓冲区、缓存层次、缓存协议，等等）。尽管如此，它可以提供有用的直觉。它也可以增量地构造执行，使交互地和随机地探索更大样例的行为成为可能，同时公理化模型需要完整的候选执行，在此之上，公理可以得到检查。

操作表示覆盖了混合尺寸的执行，可能带有 2 的不同乘幂字节尺寸的重叠的内存访问。未对齐的访问被打断为单字节访问。

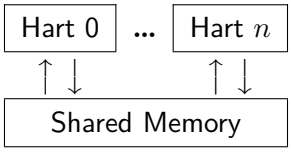
操作模型，与 RISC-V ISA 语义片段（RV64I 和 A）一起，被集成进 `rmem` 探究工具中（urlhttps://github.com/rems-project/rmem）。`rmem` 可以彻底地、伪随机地和交互地探究石蕊测试（见A.2）和小型 ELF 二进制文件。在 `rmem` 中，ISA 语义使用 Sail 显示表达（关于 Sail 语言，见 urlhttps://github.com/rems-project/sail，以及 RISC-V ISA 模型，见 urlhttps://github.com/rems-project/sail-riscv），而并发语义使用 Lem 表达（关于 Lem 语言，见 urlhttps://github.com/rems-project/lem）。

`rmem` 有一个命令行接口和一个网络接口。网络接口完全运行在客户端，并且与一个石蕊测试库一起在线提供：`http//www.cl.cam.ac.uk/~pes20/rmem`。命令行接口比网络接口更快，尤其是在穷举模式中。

下面是关于模型状态和过渡的一个非正式的介绍。正式的模型描述在下一个小节开始。

术语：与公理化表示相对，这里每个内存操作都或者是一个加载，或者是一个存储。因此，AMO 产生了两种截然不同的内存操作，加载和存储。当与“指令”连起来使用时，术语“加载”和“存储”指代产生此种内存操作的指令。因此，二者都包括 AMO 指令。术语“获取”指代一种带有 acquire-RCpc 或 acquire-RCsc 注释的指令（或者它的内存操作）。术语“释放”指代一种带有 release-RCpc 或 release-RCsc 注释的指令（或者它的内存操作）。

**模型状态** 一个模型状态由一份共享内存和一组硬件线程状态组成。



共享内存状态记录所有的至今已经传播的内存存储操作，以它们传播的次序记录（这可以变得更加高效，但是为了简化表示，我们采用了这种方式）。

每个硬件线程状态主要由一个指令实例树组成，其中一些已经*完成*了，而另一些还没有完成。未完成的指令实例可能被*重新启动*，例如，如果它们依赖于一个乱序的或者推测的加载，结果该加载出现了错误。

在指令树中，条件分支和间接跳转指令可能多个后继。当这种指令完成时，任何没有被采用的备选路径都被丢弃。

指令树中的每个指令实例都有一个状态，它包括了内部指令语义（用于该指令的 ISA 伪代码）的执行状态。模型在 Sail 中对内部指令语义使用了一种形式化描述。可以将一条指令的执行状态想成伪代码控制状态、伪代码调用栈和局部变量值的一种表示。一个指令实例状态也包括了关于该实例的内存和寄存器足迹、它的寄存器读写、它的内存操作、它是否完成，等等的信息。

**模型过渡** 对于任何模型状态，模型都定义了允许的过渡集合，每个过渡都是一个到达新的抽象机器状态的单一原子步骤。一条指令的执行往往将涉及许多过渡，而它们在操作模型执行中与来自其它指令引发的过渡进行交错。每个过渡由一个单独的指令实例引发；它将改变那个实例的状态，并且它可能依赖于、或者改变它的余下的硬件线程状态和共享内存状态，但是它不会依赖于其它硬件线程的状态，并且它也不会去改变它们：下面引入的过渡定义在 B.3.5节中，每个过渡都带有一个先决条件和一个过渡后模型状态的构造。

用于所有指令的过渡：

- 获取指令: 这个过渡代表了一个新指令实例的获取和解码，作为一个先前的获取指令实例（或者初始获取地址）的一个程序次序后继。

  模型假定指令内存是固定的；它不描述自我修改的代码的行为。特别地，获取指令过渡不会生成内存加载操作，而在过渡中也不会涉及共享内存。反而，模型依赖于在给定一个内存位置时提供操作码的外部指示。

  - 寄存器写: 这是对一个寄存器值的写。
  - 寄存器读: 这是对一个寄存器值的写，该值来自写该寄存器的最近的程序次序前趋指令实例。
  - 伪代码内部步骤: 这覆盖了伪代码内部运算：算数、函数调用，等等。
  - 完成指令: 在指令伪代码完成的这一时刻，指令不能被重启，内存访问不能被丢弃，而所有的内存效果已经发生。对于条件分支和间接跳转指令，任何从未被写到 *pc* 寄存器的地址获取的程序次序后继，与它们之后的指令实例的子树一起，都被丢弃，

专用于加载指令的过渡：

- 初始化内存加载操作：在这一时刻，加载指令的内存足迹是暂时已知的（它可能发生改变，如果较早的指令被重启的话），并且它的单个内存加载指令被满足而可以开始。
- 通过从未传播的状态转发来满足内存加载操作：通过转发，这会从程序次序中之前的内存存储操作，部分地或完全地满足单个内存加载操作。
- 从内存满足内存加载操作：这会从内存完全地满足单个内存加载操作的未完成部分。
- 完整加载操作：在这一时刻，指令的所有的内存加载操作都已经被完全地满足，且指令的伪代码可以继续执行。一个加载指令可以被重启，直到完成指令过渡。但是，在某些条件下，模型可能把一个加载指令视为不可重启的，即使是在它结束之前（例如，见传播存储操作）。

专用于存储指令的过渡：

- 初始化内存存储操作足迹：在这一时刻，存储的内存足迹是暂时已知的。
- 初始化内存存储操作的值：在这一时刻，内存存储操作具有它们的值，并且程序次序后继的内存加载操作可以通过从这些值转发而被满足。
- 提交存储指令：在这一时刻，存储指令被保证发生（指令不再会被重启或丢弃），而它们可以通过被传播到内存而开始。
- 传播存储操作：这会把单个的内存存储操作传播到内存。
- 完整存储操作：在这一时刻，指令的所有的内存存储操作都已经被传播到内存，而指令的伪代码可以继续执行。

专用于 `sc` 指令的过渡：

- 早期 `sc` 失败：这会引起 `sc` 失败，或者是一个自发的失败，或者是因为没有与程序次序之前的 `lr` 配对。
- 配对的 `sc`：这个过渡表示 `sc` 与一个 `lr` 配对了，并且可能会成功。
- 提交和传播一个 `sc` 的存储操作：这是对提交提交存储指令和传播存储操作操作过渡的一个原子性的执行，它只有当存储来自于 `lr` 读取的内容没有被覆写时才会被启用。
- Late `sc` fail：这会引起 `sc` 失败，或者是一个自发的失败，或者是因为存储来自于 `lr` 读取的内容已经被覆写。

专用于 AMO 指令的过渡：

- 满足、提交和传播一个 AMO 的操作：这是一个所有需要去满足加载操作、执行必要的算术、和传播存储操作的过渡的原子性操作。

专用于屏障指令的过渡：

- ○ 提交屏障

标记有 ○ 标签的过渡，只要它们的先决条件被满足，那么总是可以立即执行，而不需要排除其它的行为；而 ● 不行。尽管获取指令指令被标记为 ●，但是只要它不是无限地多次执行，它就可以被立即执行。

一个非 AMO 加载指令的实例，在被获取之后，往往将按这个次序经历如下的过渡：

1. 寄存器读
2. 初始化内存加载操作
3. 通过从未传播的状态转发来满足内存加载操作 和/或 从内存满足内存加载操作 (与满足实例的所有加载操作的需求数目相同)
4. 完整加载操作
5. 寄存器写
6. 完成指令

在上述过渡之前、之间和之后，可能出现任意数目的伪代码内部步骤过渡。此外，在下一个程序位置中的用于获取指令的 Fetch 指令过渡将一直可用，直到其被执行为止。

这样结束了关于操作模型的非正式描述。接下来的章节描述了正式的操作模型。

### B.3.1 指令内的伪码执行

每个指令实例的指令内语义被表达为一个状态机，它本质上运行指令伪代码。给定一个伪代码执行状态，它会计算下一个状态。大多数状态标识了一个由伪代码所请求的、挂起的内存或寄存器状态，这是内存模型必须做的。这些状态是（这是一个带标签的联合；标签用小写字母表示）：

| | | |
|---|---|---|
| LOAD_MEM(*kind*, *address*, *size*, *load_continuation*) | - | 内存加载操作 |
| EARLY_SC_FAIL(*res_continuation*) | - | 允许 `sc` 提前失败 |
| STORE_EA(*kind*, *address*, *size*, *next_state*) | - | 内存存储有效地址 |
| STORE_MEMV(*mem_value*, *store_continuation*) | - | 内存存储值 |
| FENCE(*kind*, *next_state*) | - | 屏障 |
| READ_REG(*reg_name*, *read_continuation*) | - | 寄存器读 |
| WRITE_REG(*reg_name*, *reg_value*, *next_state*) | - | 寄存器写 |
| INTERNAL(*next_state*) | - | 伪代码内部步骤 |
| DONE | - | 伪代码的结束 |

Here:

- *mem_value* 和 *reg_value* 是字节的列表；
- *address* 是一个 XLEN 位的整数；
- 对于加载/存储，*kind* 标识了它是否是 `lr/sc`、acquire-RCpc/release-RCpc、acquire-RCsc/release-RCsc、acquire-release-RCsc；
- 对于屏障，*kind* 标识了它是否是一个普通的屏障还是 TSO 屏障，以及（对于普通屏障）前趋和后继次序的位；
- *reg_name* 标识了一个寄存器和它的一部分（开始和结束位的索引）；以及
- continuation 描述了指令实例将如何继续处理每个可能由周围内存模型所提供的值（*load_continuation* 和 *read_continuation* 取得从内存加载的值和读取从先前寄存器写入的值，*store_continuation* 对于一个失败的 sc 取假，而在所有其它情形中取真，而如果 `sc` 失败，*res_continuation* 取假，否则取真）。

---

例如，给定加载指令`lw x1,0(x2)`，一次执行往往将如下运行。初始执行状态将从给定的操作码的伪代码计算。这可能预计是 READ_REG(`x2`, *read_continuation*)。把寄存器 `x2` 最近写入的值（如果有必要，指令语义将被阻塞，直到寄存器的值可用），也就是 `0x4000`，输入到 *read_continuation*，返回 LOAD_MEM(`plain_load`, `0x4000`, `4`, *load_continuation*)。把从内存地址 `0x4000` 加载的 `4` 字节的值，也就是 `0x42`，输入到 *load_continuation*，返回 WRITE_REG(`x1`, `0x42`, DONE)。在上述状态之间可能出现许多 INTERNAL(*next_state*) 状态。

注意，写内存被分为两个步骤，STORE_EA 和 STORE_MEMV：第一个步骤制造存储暂时已知的内存足迹，而第二个步骤添加要被存储的值。我们确保这些在伪代码中是成对的（STORE_EA 后面跟着 STORE_MEMV），但是在它们之间可能会有其它的步骤。

---

可以观察到，STORE_EA 可以发生在要存储的值被决定之前。例如，对于操作模型所允许的石蕊测试 *LB + fence.r.rw + data-po*（正如 *RVWMO* 所允许的那样），硬件线程 *1* 中的第一个存储必须在它的值被决定之前就采取 STORE_EA 步骤，以便第二个存储可以看到它是一个非重叠的内存足迹，以允许第二个存储被乱序提交而不违反一致性。

除了明确执行一次加载和一次存储的 AMO，每个指令的伪代码最多执行一次存储或一次加载。那些内存访问然后通过硬件线程语义被分割为架构上的原子性单元（见下面的初始化内存加载操作和初始化内存存储操作足迹）。

非正式地，一个寄存器读的每一位都应当由来自最近的（以程序次序）可以写该位的指令实例的一个寄存器写（或者来自硬件线程的初始寄存器状态，如果没有这样的写的话）来满足。因此，有必要知道每个指令实例的寄存器写的足迹，我们在指令实例被创建时（见下面的获取指令的动作）计算这个足迹。我们确保在伪代码中，每个指令对每个寄存器位最多执行一次寄存器写，并且也保证不会尝试读取它刚刚写入的寄存器的值。

每个寄存器读必须等待合适的寄存器写被执行（正如上面描述的那样），从这个事实浮现了模型中的数据流依赖（地址和数据）。

### B.3.2 指令实例状态

每个指令实例 $i$ 都拥有一个状态，包括：

- *program_loc*，指令被从此获取的内存地址；
- *instruction_kind*，识别这是否是一个加载、存储、AMO、屏障、分支/跳转，或者一个‘简单的’指令（这也包括了一种类似于描述伪指令执行状态的类型）；
- *src_regs*，源 *reg_name* 的集合（包括系统寄存器），由指令的伪代码静态决定；
- *dst_regs*，目的 *reg_name*（包括系统寄存器），由指令的伪代码静态决定；
- *pseudocode_state*（或者有时只简写为‘state’），如下之一（这是一个带标签的联合；标签用小写字母表示）：

  | | | |
  |---|---|---|
  | PLAIN(*isa_state*) | - | 准备制造一个伪代码过渡 |
  | PENDING_MEM_LOADS(*load_continuation*) | - | 请求内存加载操作 |
  | PENDING_MEM_STORES(*store_continuation*) | - | 请求内存存储操作 |

- *reg_reads*，寄存器读实例已经被执行，对于每个实例，包括寄存器写它从中读取的片段；
- *reg_writes*，寄存器写实例已经被执行；
- *mem_loads*，一组内存加载操作的集合，并且对于每个加载操作，是尚未满足的片段（还没有被满足的字节的索引），而对于已满足的片段，则是满足它的存储片段（每个片段由一个内存存储操作和它的字节索引的子集组成）。

每个内存加载操作包括了一个内存足迹（地址和尺寸）。每个内存存储操作包括了一个内存足迹和（当值可用时）一个值。

一个带有非空 *mem_loads* 的加载指令实例，如果所有的加载操作都被满足（换句话说，没有未满足的加载片段），那么被称之为被完全满足的。

非正式地，一个指令实例被称为具有**完全决定的**数据，如果为它的源寄存器提供输入的加载（和 `sc`）指令被完成了。类似地，它被称为具有完全决定的内存足迹，如果为它的内存操作地址寄存器提供输入的加载（和 `sc`）指令被完成了。正式地讲，我们首先定义了**完全决定的寄存器写**的概念：一个来自指令实例 $i$ 的 *reg_writes* 的 $w$ 被称之为**完全决定的**，如果满足了下列条件之一：

1. i 被完成了；或者
2. w 所写的值不会被 i 制造的内存操作所影响（即，一个从内存加载的值或者 sc 的结果），并且，对于 i 已经制造的每个影响 w 的寄存器读，源于 i 读取的寄存器写都是完全决定的（或者 i 从初始寄存器状态读取）。

现在，一个指令实例 $i$ 被称为具有**完全决定的**数据，如果对于每个来自 *reg_reads* 的寄存器读 $r$，$r$ 从中读取的寄存器写都是完全决定的。一个指令实例 $i$ 被称为**具有完全决定的内存足迹**，如果对于来自 *reg_reads* 的每个输入到 $i$ 的内存操作地址的寄存器读 $r$，$r$ 从中读取的寄存器写都是完全决定的。

---

对于每次寄存器写，`rmem` 工具都会记录，在执行写的这一时刻，该指令已经读取的来自其它指令的寄存器写的集合。通过小心地安排由工具覆盖的指令的伪代码，我们能够做到这点，使得这确实是这次写所依赖的寄存器写的集合。

### B.3.3 硬件线程状态

单个硬件线程的模型状态包括：

- *hart_id*，一个关于硬件线程的唯一的标识符；
- *initial_register_state*，各寄存器的初始寄存器状态；
- *initial_fetch_address*，初始指令获取地址；
- *instruction_tree*，以程序次序，一个已经被获取到的（并且没有被丢弃的）指令实例的树。

### B.3.4 共享内存状态

共享内存的模型状态包括一个内存存储操作的列表，按它们传播到共享内存的次序排序。

当一个存储操作被传播到共享内存时，它被简单地添加到列表的末端。当一个加载操作从内存被满足时，对于加载操作的每一个字节，将返回最近对应的存储片段。

---

对于大多数目的，将共享内存想象为一个数组会更简单，即，一个从内存位置到内存存储操作片段的映射，这里每个内存位置被映射到一个最近的针对该位置的内存存储操作的 *1* 字节的片段。然而，对于适当地处理 `sc` 指令，这种抽象不够详细。*RVWMO*原子性公理允许来自与 `sc` 相同的硬件线程的存储操作，以在 `sc` 的存储操作和与之配对的 `lr` 将读取自的存储操作之间进行干预。为了允许这种存储操作进行干预，而禁止其它的，数组抽象必须被扩展以记录更多的信息。这里，我们使用一个列表，因为它很简单，但是更有效和可扩展的实现应当更可能使用某些更好的东西。

### B.3.5 过渡

下面的每个段落都描述了一种系统过渡。描述开始于一个基于当前系统状态之上的条件。只有当条件被满足时，过渡可以发生在当前状态。条件后面跟着一个行动，它在过渡发生时应用到该状态，以生成新的系统状态。

**获取指令** 可以从地址 *loc* 获取指令实例 $i$ 的一个可能的程序次序后继，如果：

1. 它还没有被获取到，即，在硬件线程的指令树中，$i$ 没有直接的来自 *loc* 的后继；并且
2. 如果 $i$ 的伪代码已经向 *pc* 写入了一个地址，那么 *loc* 必须是那个地址，否则 *loc* 就是：
   - 对于一个条件分支，是后继地址或分支目标地址；
   - 对于一个（直接的）跳转和链接指令（`jal`），是目标地址；
   - 对于一个间接跳转指令（`jalr`），是任何地址；以及
   - 对于任何其它的指令，是 $i.program\_loc + 4$。

行动：在 *loc* 处的程序内存中为指令构建一个新的初始化的指令实例 $i$'，它带有从指令伪代码计算的状态 Plain(*isa_state*)，包括从伪代码可以获得的静态信息，例如它的 *instruction_kind*，*src_regs*，和 *dst_regs*，并把 $i$' 添加到硬件线程的指令树作为 $i$ 的一个后继。

---

可能的下一个获取地址（*loc*）在获取 $i$ 之后是立即可用的，并且模型不需要等待伪代码写到 pc；这允许乱序执行，并且推测过去的条件分支和跳转。对于大多数指令，这些地址很容易

从指令的伪代码获得。唯一的例外是间接跳转指令（jalr），那里地址依赖于寄存器中持有的值。原则上，数学模型应当允许这里推测到任意的地址。在 rmem 工具中的穷举搜索通过多次运行穷举搜索来处理这个问题，对于每个间接跳转，都带有不断增长的、可能的下一个获取地址的集合。最初的搜索使用空集合，因此在间接跳转指令之后没有获取，直到指令的伪代码写到 pc，然后我们使用该值来获取下一条指令。在开始穷举搜索的下一次迭代之前，我们为每个间接跳转（按代码位置分组）收集了它在先前搜索迭代中的所有执行中写到 pc 的值的集合，并使用它作为指令的可能的下一个获取的地址。当没有检测到新的获取地址时，这个过程终止。

**初始化内存加载操作** 一个状态 PLAIN(LOAD_MEM(*kind*, *address*, *size*, *load_continuation*)) 中的指令实例 $i$ 总是可以初始化对应的内存加载操作。行动：

1. 构造合适的内存加载操作 *mlos*：
   - 如果 *address* 被对齐到 *size*，那么 *mlos* 是一个从 *address* 开始的 *size* 字节的单个内存加载操作；
   - 否则，*mlos* 是一组数量为 *size* 的内存加载操作的集合，每个加载操作负责地址 $address \ldots address + size - 1$ 中的一个字节。
2. 把 $i$ 的 *mem_loads* 设置到 *mlos*；并且
3. 更新 $i$ 的状态为 PENDING_MEM_LOADS(*load_continuation*)。

---

在 17.1节中说到，未对齐的内存访问可能在任意粒度被分解。这里我们把它们分解为一个字节的访问，因为这个粒度包含了所有其它的粒度。

**通过从未传播的状态转发来满足内存加载操作** 对于一个状态 PENDING_MEM_LOADS(*load_continuation*) 中的非 AMO 加载指令实例 $i$，和 $i.mem\_loads$ 中的一个具有未满足片段的内存加载操作 *mlo*，内存加载操作可以通过从未被程序次序之前的存储指令实例所传播的内存存储操作转发，而被部分地或完全地满足，如果：

1. 所有的程序次序在先的设置了.sr 和.pw 的 fence 指令都完成了；
2. 对于每个程序次序在先的设置了.sr 和.pr，但是没有设置.pw 的 fence 指令，$f$，如果 $f$ 没有完成，那么所有的程序次序在 $f$ 之前的加载指令都被完全地满足了；
3. 对于每个没有完成的程序次序在先的 fence.tso 指令，$f$，所有的程序次序先于 $f$ 的加载指令都被完全地满足了；
4. 如果 $i$ 是一个 load-acquire-RCsc，所有的程序次序在先的 store-relaase-RCsc 都完成了；

5. 如果 $i$ 是一个 load-acquire-release，所有的程序次序在先的指令都完成了；
6. 所有的未完成的程序次序在先的 load-acquire 指令都完全地被满足了；并且
7. 所有的程序次序在先的 store-acquire-release 指令都被满足了；

令 *msoss* 是所有来自程序次序先于 $i$ 的、并且已经计算出要存储的值的非 `sc` 存储指令实例的未被转发的内存存储操作片段的集合，它们与 *mlo* 的未被满足的片段重叠，并且不会被干扰的存储操作、或从一个干扰的加载读取的存储操作所替代。最后一个条件要求，对于 *msoss* 中的每个来自指令 $i$' 的内存存储操作片段 *msos*：

- 没有程序次序在 $i$ 和 $i$' 之间的存储指令带有与 *msos* 重叠的内存存储操作；并且
- 没有程序次序在 $i$ 和 $i$' 之间的加载指令被来自不同的硬件线程的重叠的内存存储操作片段所满足。

行动：

1. 更新 *i.mem_loads*，以表示 *mlo* 被 *msoss* 满足了；并且
2. 重启任何违背一致性的推测性指令，作为这个的结果，也就是说，对于每个未完成的作为 i 的程序次序后继的指令 $i$'，和 $i$' 的每个由 *msoss*' 满足的内存加载操作 *mlo*'，如果在 *msoss*' 中存在一个内存存储操作片段 *msos*'，和一个来自 *msoss* 中一个不同的内存存储操作的重叠的内存存储操作片段，并且 *msos*' 并非来自 $i$ 的一个程序次序后继指令，那么重新启动 $i$' 和它的**重启依赖**。

此处，指令 $j$ 的**重启依赖**是指：

- $j$ 的程序次序后继，如果它有关于 $j$ 的寄存器写的数据流依赖；
- $j$ 的程序次序后继，如果它有一个内存加载操作，其读取自 j 的一个内存存储操作（通过转发）；
- 如果 $j$ 是一个 load-acquire，那么是 $j$ 的所有的程序次序后继；
- 如果 $j$ 是一个加载，对于每个设置了`.sr` 和`.pr`，但是没有设置`.pw` 的 `fence`，$f$，如果它是 $j$ 的一个程序次序后继，那么是所有的是 $f$ 的程序次序后继的加载指令；
- 如果 $j$ 是一个加载，对于每个 `fence.tso`，$f$，如果是 $j$ 的一个程序次序后继，那么是所有的是 $f$ 的程序次序后继的加载指令；以及
- （递归地）所有的上述指令实例的所有重启依赖。

---

向一个内存加载转发内存存储操作可能只满足该加载的某些片段，而使其它片段仍是未满足的。

一个程序次序在先的不可用的存储操作，当它变得可用时，在采用上面的过渡的时候，可能使得 *msoss* 暂时不可靠（违反一致性）。那样的存储将阻止加载的完成（见完成指令），并将在存储操作被传播时，导致它重启（见传播存储操作）。

上述过渡条件的结果就是，*store-release-RCsc* 内存存储操作不能被转发到 *load-acquire-RCsc* 指令：*msoss* 不包括来自已完成存储的内存存储操作（因为那些必定是已传播的内存存储操作），而且当加载是 *acquire-RCsc* 的时候，上述条件需要所有的程序次序在先的 *store-release-RCsc* 都被完成。

**从内存满足内存加载操作** 对于一个非 AMO 加载指令或者一个“AMO 的满足、提交和传播操作”过渡上下文中的 AMO 指令的指令实例 $i$，任何 $i.mem\_loads$ 中的有未满足片段的内存加载操作 mlo，可以从内存被满足，如果通过未传播的存储转发来满足内存加载操作的所有条件都被满足了的话。行动：令 *msoss* 是来自覆盖了 *mlo* 的未满足片段的内存的内存存储操作片段，并应用通过未传播的存储转发满足内存加载操作的行为。

---

注意通过未传播的存储转发来满足内存加载操作可能会使内存加载操作的某些片段无法满足，那些片段将不得不通过再次采取过渡来满足，或者采取从内存满足内存加载操作来满足。另一方面，从内存满足内存加载操作将总是满足内存加载操作的所有的未满足片段。

**完整加载操作** 状态 Pending_mem_loads(*load_continuation*) 中的一个加载指令实例 $i$ 可以是完整的（不要与完成相混淆），如果所有的内存加载操作 $i.mem\_loads$ 被完全地满足了（即，没有未满足的片段）。行动：将 i 的状态更新为 Plain(*load_continuation(mem_value)*)，这里 *mem_value* 是从所有满足 $i.mem\_loads$ 的内存存储操作片段集合而成的。

**早期 `sc` 失败** 状态 Plain(Early_sc_fail(*res_continuation*)) 中的一个 `sc` 指令实例 $i$ 可以总是被造成失败。行动：把 $i$ 的状态更新到 Plain(*res_continuation(false)*)。

**配对的 `sc`** 状态 Plain(Early_sc_fail(*res_continuation*)) 中的一个 `sc` 指令实例 $i$ 可以继续它的（可能成功的）执行，如果 $i$ 与一个 `lr` 配对的话。行动：把 $i$ 的状态更新到 Plain(*res_continuation(true)*)。

**初始化内存存储操作足迹** 状态 Plain(Store_ea(*kind*, *address*, *size*, *next_state*)) 中的一个指令实例 $i$ 可以总是宣布它的挂起的内存操作足迹。行动：

1. 构造合适的内存存储操作 *msos*（不带有存储的值）：
   - 如果 *address* 对齐到 *size*，那么 *msos* 是一个单独的针对 *address* 的 *size* 字节的内存存储操作；
   - 否则，*msos* 是一组数量为 *size* 的内存存储操作集合，每个长度 1 字节，针对地址 $address \ldots address + size - 1$。

2. 把 *i.mem_stores* 设置为 *msos*；并且
3. 把 *i* 的状态更新到 PLAIN(*next_state*)。

---

注意，在采取上述过渡之后，内存存储操作还没有拥有它们的值。把这个过渡从下面的过渡分离出来的重要性在于，它允许其它的程序次序后继的存储指令观察到这个指令的内存足迹，并且如果它们不重叠的话，尽可能早地乱序传播（即，在数据寄存器的值变得可用之前）。

**初始化内存存储操作的值** 状态 PLAIN(STORE_MEMV(*mem_value* 中的一个指令实例 i 可以总是初始化内存存储操作 *i.mem_stores* 的值。行动：

1. 在内存存储操作 *mem_value* 之间分割出 *i.mem_stores*；以及
2. 把 *i* 的状态更新到 PENDING_MEM_STORES(*store_continuation*)。

**提交存储指令** 一个非 `sc` 存储指令，或者一个在“提交和传播一个 `sc` 的存储操作”的上下文中的 `sc` 指令，如果在状态 PENDING_MEM_STORES(*store_continuation*) 之中，那么可以被提交（不要与传播相混淆），如果：

1. *i* 具有完全决定的数据；
2. 所有的程序次序在先的条件分支和间接跳转指令都完成了；
3. 所有的程序次序在先的设置了`.sw` 的 `fence` 指令都完成了；
4. 所有的程序次序在先的 `fence.tso` 指令都完成了；
5. 所有的程序次序在先的 load-acquire 指令都完成了；
6. 所有的程序次序在先的 store-acquire-release 指令都完成了；
7. 如果 *i* 是一个 store-release，所有的程序次序在先的指令都完成了；
8. 所有的程序次序在先的内存访问指令具有完全决定的内存足迹；
9. 所有的程序次序在先的存储指令，除了失败的 `sc`，都已经初始化并因此具有非空的 *mem_stores*；并且
10. 所有的程序次序在先的加载指令都已经初始化并因此具有非空的 *mem_loads*。

行动：记录 i 被提交了。

注意，如果条件8被满足了，条件9和10也被满足，或者将在采取某些立即的过渡之后被满足。因此，对它们的要求并不会增强模型。通过要求它们，我们保证了先前的内存访问指令已经采取了足够的过渡，使得它们的内存操作对传播存储操作的条件检查可见，这是指令将采取的下一个过渡，使得那个条件更加简单。

**传播存储操作** 对于状态 PENDING_MEM_STORES(*store_continuation*) 中的一个提交的指令实例 $i$，和一个在 $i.mem\_stores$ 之中的未传播的内存存储操作 $mso$，$mso$ 可以被传播，如果：

1. 所有的与 $mso$ 重叠的程序次序在先的存储指令的内存存储操作都已经传播；
2. 所有的与 $mso$ 重叠的程序次序在先的加载指令的内存加载操作都已经被满足，并且（加载指令）是不可重启的（见下面的定义）；并且
3. 所有的通过转发 $mso$ 被满足的内存加载操作都被完全地满足。

此处，一个未完成的指令实例 $i$ 是不可重启的，如果：

1. 不存在一个存储指令 s 和 s 的一个未传播的内存存储操作 mso，使得对 mso 应用“传播存储操作”过渡的行为将导致 j 的重启；并且
2. 不存在一个未完成的加载指令 l 和 l 的一个内存加载操作 mlo，使得对 mlo 应用“通过从未传播的存储转发来满足内存加载操作”/“从内存满足内存加载操作”过渡（甚至 mlo 已经被满足）将导致 j 的重启。

行动：

1. 使用 $mso$ 更新共享内存的状态；
2. 更新 $i.mem\_stores$，以表示 $mso$ 被传播了；以及
3. 重启任何因为这个的结果而已经违背了一致性的推测性指令，也就是说，对于每个程序次序晚于 $i$ 的未完成的指令 $i'$ 和 $msoss'$ 中的与 $mso$ 重叠而不来自 $mso$ 的每个 $i'$ 的内存加载操作 $mlo'$，并且 $msos'$ 不是来自 $i$ 的一个程序次序后继，重启 $i'$ 和它的重启依赖（见：通过从未传播的状态转发来满足内存加载操作）。

**提交和传播一个 sc 的存储操作** 一个来自硬件线程 $h$ 的、状态 PENDING_MEM_STORES(*store_continuation*) 中的、未提交的 `sc` 指令实例 $i$，带有一个配对的 `lr` $i'$，其已经被某些存储片段 $msoss$ 所满足，可以被同时提交和传播，如果：

1. $i$' 被完成；
2. 每个已经转发到 $i$' 的内存存储操作都是已传播的；
3. 提交存储指令的条件被满足；
4. 传播存储操作的条件被满足（注意一个 `sc` 指令只能有一个内存存储操作）；以及
5. 在共享内存中，对于每个来自 *msoss* 的存储片段 *msos*，从 *msos* 被传播到内存后的任何时刻，*msos* 都还没有被来自不是 $h$ 的硬件线程的存储覆写。

行动：

1. 应用提交存储指令的行动；以及
2. 应用传播存储操作的行动。

**Late sc fail** 状态 Pending_mem_stores(*store_continuation*) 中的一个 `sc` 指令实例 $i$，如果还没有传播它的内存存储操作，那么总是可以被造成失败。行动：

1. 清除 $i$.*mem_stores*；以及
2. 把 $i$ 的状态更新到 Plain(*store_continuation(false)*)。

---

为了效率，`rmem` 工具只在不可能采用一个提交和传播一个 `sc` 存储操作过渡的时候，才允许这个过渡。这不影响所允许的最终状态的集合，但是当交互式地探究时，如果 *sc* 应当失败，那么应当使用早期 *sc* 失败过渡，而不是期待这个过渡。

**完整存储操作** 状态 Pending_mem_stores(*store_continuation*) 中的一个存储指令实例 $i$，如果在 $i$.*mem_stores* 中的所有内存存储操作都已经被传播，那么可以总是完整的（不要与完成相混淆）。行动：把 $i$ 的状态更新到 Plain(*store_continuation(true)*)。

**满足、提交和传播一个 AMO 的操作** 状态 Pending_mem_loads(*load_continuation*) 中的一个 AMO 指令实例 $i$ 可以执行它的内存访问，如果可以执行下列顺序的过渡而没有干扰的过渡：

1. 从内存满足内存加载操作
2. 完整加载操作

3. 伪代码内部步骤 (zero or more times)
4. 初始化内存存储操作的值
5. 提交存储指令
6. 传播存储操作
7. 完整存储操作

并且额外地，完成指令的条件，除了不要求 $i$ 在状态 Plain(Done 之中，其余在这些过渡之后保持不变。行动：一个接一个地、不加干扰过渡地，执行上述序列的过渡（这不包括完成指令），

---

注意，程序次序在先的存储不能被转发到一个 *AMO* 的加载。这纯粹是因为上面的过渡序列不包括转发过渡。但是即使它确实包含了转发过渡，当尝试执行传播存储操作过渡的时候，该序列也将失败，因为这个过渡需要所有的程序次序在先的针对重叠的内存足迹的存储指令都被传播，而传播需要存储操作是未被传播的。

此外，一个 *AMO* 的存储不能被转发到一个程序次序后继的加载。在采取上面的过渡之前，*AMO* 的存储指令没有拥有它的值，并因此不能被转发；在采取上面的过渡之后，存储操作被传播，并因此不能被转发。

**提交屏障** 在状态 Plain(Fence(*kind*, *next_state*)) 中的一个屏障指令实例 $i$ 可以被提交，如果：

1. 如果 $i$ 是一个普通 fence，并且它设置了`.pr`，那么所有的程序次序在先的加载指令都是完成的；
2. 如果 $i$ 是一个普通 fence，并且它设置了`.pw`，那么所有的程序次序在先的存储指令都是完成的；以及
3. 如果 $i$ 是一个 `fence.tso`，那么所有的程序次序在先的加载和存储指令都是完成的。

行动：

1. 记录 $i$ 被提交了；以及
2. 把 $i$ 的状态更新到 Plain(*next_state*)。

**寄存器读** 状态 Plain(Read_reg(*reg_name* 中的一个指令实例 $i$ 可以执行一次 *reg_name* 的寄存器读，如果它需要读取自的每个指令实例都已经执行了所期待的 *reg_name* 寄存器写。

对于 *reg_name* 的每一位，令 *read_sources* 包括，可以写到该位的最近（以程序次序）的指令实例对于该位的写入（如果有的话）。如果没有这样的指令，来源就是来自 *initial_register_state* 的初始的寄存器值。令 *reg_value* 为从 *read_sources* 集合的值。行动：

1. 把 *reg_name* 添加到带有 *read_sources* 和 *reg_value* 的 *i.reg_reads*；以及
2. 把 *i* 的状态更新为 Plain(*read_cont(reg_value)*)。

**寄存器写** 状态 Plain(Write_reg(*reg_name* 中的一个指令实例 *i* 总是可以执行一个 *reg_name* 寄存器写。行动：

1. 把 *reg_name* 添加到带有 *deps* 和 *reg_value* 的 *i.reg_writes*；以及
2. 把 *i* 的状态更新到 Plain(*next_state*)。

此处 *deps* 是所有来自 *i.reg_reads* 的读取来源的集合，和一个标志的配对，这个标志为真，当且仅当 *i* 是一个已经被完全地满足的加载指令实例。

**伪代码内部步骤** 状态 Plain(Internal(*next_state*)) 中的一个指令实例 *i* 总是可以执行那个伪代码内部的步骤。行动：把 *i* 的状态更新到 Plain(*next_state*)。

**完成指令** 状态 Plain(Done) 中的一个未完成的指令实例 *i* 可以被完成，如果：

1. 如果 *i* 是一个加载指令：
   (a) 所有的程序次序在先的 load-acquire 指令都被完成了；
   (b) 所有的程序次序在先的设置了.sr 的 `fence` 指令都被完成了；
   (c) 对于每个程序次序在先的没有完成的 `fence.tso` 指令，*f*，所有的程序次序在 *f* 之前的加载指令都被完成了；并且
   (d) 保证 *i* 的内存加载指令所读取的值将不会引起对一致性的违背，即，对于任何程序次序在先的指令实例 *i'*，令 *cfp* 是来自程序次序在 *i* 和 *i'* 之间的存储指令的已传播的内存存储操作，和从程序次序在 *i* 和 *i'* 之间、包括 *i'* 的存储指令转发到 *i* 的固定的内存存储操作，这二者的组合，并令 $\overline{cfp}$ 是 *i* 的内存足迹中的 *cfp* 的补。如果 $\overline{cfp}$ 是不空的：
      i. *i'* 具有一个完全决定的内存足迹；
      ii. *i'* 没有与 $\overline{cfp}$ 重叠的未传播的内存存储操作；并且

iii. 如果 $i$' 是一个带有与 $\overline{cfp}$ 重叠的内存足迹的加载，那么与 $\overline{cfp}$ 重叠的 $i$' 的所有的内存加载操作都被满足，并且 $i$' 是不可重启的（对于如何决定一个指令是否是不可重启的，见传播存储操作过渡）。

这里，一个内存存储操作被称为固定的，如果存储指令具有完全决定的数据。

2. $i$ 具有一个完全决定的数据；并且
3. 如果 $i$ 不是一个屏障，那么所有的程序次序在先的条件分支和间接跳转指令都被完成了。

行动：

1. 如果 $i$ 是一个条件分支或者间接跳转指令，丢弃任何未采取的执行路径，即，移除 *instruction_tree* 中的所有的不可被分支/跳转采用而达到的指令实例；并且
2. 记录该指令为完成的，即，设置 *finished* 为真。

### B.3.6 局限性

- 模型覆盖用户级 RV64I 和 RV64A。特别地，它不支持未对齐的原子性扩展“Zam”或者总存储排序扩展“Ztso”。使模型适应 RV32I/A 和 G、Q 还有 C 扩展应当是轻而易举的，但是我们从未尝试过它。这将主要涉及，为指令写 Sail 代码，同时对并发模型的改变（如果有的话）最小。
- 模型只覆盖了一般的内存访问（它不处理 I/O 访问）。
- 模型没有覆盖 TLB 相关的效果。
- 模型假设指令内存是固定的。特别地，Fetch 指令过渡不会生成内存加载操作，而共享内存不会在过渡中被涉及。反而，模型依赖于一个在给定内存位置时提供操作码的外部指示。
- 模型没有覆盖异常、陷入和中断。

# 参考文献


[1] RISC-V Assembly Programmer's Manual. https://github.com/riscv/riscv-asm-manual.

[2] RISC-V ELF psABI Specification. https://github.com/riscv/riscv-elf-psabi-doc/.

[3] IEEE standard for a 32-bit microprocessor. IEEE Std. 1754-1994, 1994.

[4] G. M. Amdahl, G. A. Blaauw, and Jr. F. P. Brooks. Architecture of the IBM System/360. *IBM Journal of R. & D.*, 8(2), 1964.

[5] Werner Buchholz, editor. *Planning a computer system: Project Stretch*. McGraw-Hill Book Company, 1962.

[6] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[7] Timothy H. Heil and James E. Smith. Selective dual path execution. Technical report, University of Wisconsin - Madison, November 1996.

[8] ANSI/IEEE Std 754-2008, IEEE standard for floating-point arithmetic, 2008.

[9] Manolis G.H. Katevenis, Robert W. Sherburne, Jr., David A. Patterson, and Carlo H. Séquin. The RISC II micro-architecture. In *Proceedings VLSI 83 Conference*, August 1983.

[10] Hyesoon Kim, Onur Mutlu, Jared Stark, and Yale N. Patt. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 43–54, 2005.

[11] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, Washington, DC, USA, 1998.

[12] David D. Lee, Shing I. Kong, Mark D. Hill, George S. Taylor, David A. Hodges, Randy H. Katz, and David A. Patterson. A VLSI chip set for a multiprocessor workstation–Part I: An RISC microprocessor with coprocessor interface and support for symbolic processing. *IEEE JSSC*, 24(6):1688–1698, December 1989.

[13] OpenCores. OpenRISC 1000 architecture manual, architecture version 1.0, December 2012.

[14] Heidi Pan, Benjamin Hindman, and Krste Asanović. Lithe: Enabling efficient composition of parallel libraries. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism (HotPar '09)*, Berkeley, CA, March 2009.

[15] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with Lithe. In *31st Conference on Programming Language Design and Implementation*, Toronto, Canada, June 2010.

[16] David A. Patterson and Carlo H. Séquin. RISC I: A reduced instruction set VLSI computer. In *ISCA*, pages 443–458, 1981.

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

[18] Pierre Roux. Innocuous Double Rounding of Basic Arithmetic Operations. *Journal of Formalized Reasoning*, 7(1):131–142, November 2014.

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

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

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

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

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

[24] Andrew Waterman. Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed. Master's thesis, University of California, Berkeley, 2011.

[25] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, University of California, Berkeley, 2016.

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

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