**RISC-V指令集手册**

**第一卷：非特权指令集架构**

文档版本20191213

编者：安德鲁·沃特曼1，克尔斯泰·阿桑诺维奇1,2

1SiFive股份有限公司

2加州大学伯克利分校EECS系CS部

[andrew@sifive.com](mailto:andrew@sifive.com), [krste@berkeley.edu](mailto:krste@berkeley.edu)

2019年12月13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，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。

本文档在知识共享属性4.0国际许可证下发布。

本文档是《RISC-V指令集手册，第一卷：用户级指令集架构 2.1版本》的衍生，该手册在如下许可证下发布：©2010-2017 安德鲁·沃特曼，李云燮，大卫·帕特森，克尔斯泰·阿桑诺维奇，知识共享属性4.0国际许可证。

引用请使用：“RISC-V指令集手册，第一卷：用户级指令级架构，文档版本20191213”，编者安德鲁·沃特曼和克尔斯泰·阿桑诺维奇，RISC-V基金会，2019年12月。

# 前言

本文档描述了RISC-V非特权架构。

标记为“被批准”的ISA模块当前已经被批准了。标记为“被冻结”的模块在被提交批准之前，预计不会有重大的变化。标记为“草案”的模块在被批准之前，预计会有所改变。

文档包含RISC-V ISA模块的如下版本：

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

此版本文档中的变动包括：

* A扩展，现在是2.1版本，已经在2019年12月被理事会批准。
* 定义了大字节序的ISA变体。
* 把用于用户模式中断的N扩展移动到了第二卷。

## 对基于已批准的20190608版本文档的前言

本文档描述了RISC-V非特权架构。

RVWMO内存模型此时已经被批准了。标记为“被批准”的ISA模块目前已经被批准。标记为“被冻结”的模块在被提交批准之前预计不会有显著的改动。标记为“草案”的模块在批准之前预计会有变动。

文档包括RISC-V ISA模块的如下版本：

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

此版本文档中的变化包括：

* 对于2019年年初被理事会批准的ISA模块，将其描述移动到“**被批准的**”。
* 从批准中移除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：已存在被批准的RVWMO内存模型；排除了先前基础ISA中的FENCE.I、计数器和CSR指令。
* 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；而版本2.2定义了NaN装箱方案，并更改了FMIN和FMAX指令的定义。
* 把文档的名字变为引用“非特权的”指令，以此作为将ISA规范从平台概述授权中分离的移动的一部分。
* 为执行环境、硬件线程、陷入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*，和*不合格的*。
* 在交替字节序下移除了文本隐含操作，因为交替字节序操作还没有被RISC-V所定义。
* 修改了未对齐的加载和存储行为的描述。规范现在允许在执行环境界面中的显式的未对齐地址陷入，而不是仅仅在用户模式中授权未对齐的加载和存储的隐式处理。而且，现在允许报告不应被模拟的、有关未对齐访问（包括原子访问）的访问异常。
* 把FENCE.I从强制性基础模块中移出，编入一个独立的扩展，名为Zifencei ISA。FENCE.I。FENCE.I曾经被从Linux用户ABI中去除，造成如今在实现大型非一致性指令和数据缓存中的问题。然而，它仍然是仅有的标准指令获取的一致性机制。
* 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，即，特定的编码在RV32E和RV64I章节中产生的非法指令异常。
* 计数器/计时器指令现在不被认为是强制性基础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，并改变了它们遇到有符号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。
* 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代替。
* 去除了V扩展章节的内容，因为它现在已经被独立的向量枯燥草案文档所代替。

## 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  RV32E  RV64I  RV128I | 2.0  1.9  2.0  1.7 | Y  N  Y  N |
| 扩展模块 | 版本 | 被冻结？ |
| M  A  F  D  Q  L  C  B  J  T  P  V  N | 2.0  2.0  2.0  2.0  2.0  0.0  2.0  0.0  0.0  0.0  0.1  0.7  1.1 | Y  Y  Y  Y  Y  N  Y  N  N  N  N  N  N |

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：

* 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，把所有的扩展放在了规范次序的首位。
* 改进了描述和注释。
* 修改了关于JALR的隐含提示建议，以支持LUI/JALR和AUIPC/JALR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）映射的表。
* 澄清了**fcsr**高位的作用和行为。
* 改正了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装箱模型持有较窄（小于FLEN）浮点值的行为。
* 定义了FMA的异常行为（∞，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的打包SMID协议。
* 提出了一个V向量指令集扩展的草案。
* 提出了一个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[1]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[25]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，计数寄存器在此引入，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转换的返回值。
* 澄清了LR/SC所允许的成功和需要的失败，包括压缩指令在序列中的使用。
* 一个新的RV32E基础ISA提案，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。
* 为软浮点调用惯例放松了栈对齐，并描述了RV32E调用约定。
* 一个1.9版本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[24]开始，已经有了如下改变：

* 该ISA已经被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，让直接编码更加有效。
* 基础ISA按小字节序内存体系定义，而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）指令已经加入到了原子指令扩展中。
* AMO和LR/SC可以支持释放一致性模型。
* 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添加了一个20位的高位立即数）取代了RDNPC指令（它只读取当前的PC值）。这导致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。这样去掉了仅有的目的寄存器不明确的指令，也把J-Type指令格式从基础ISA中去除。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，还有*rd*和*rs1*寄存器的修饰符，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，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**的部分域：取整模式和异常标志位。
* 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位和更多位指令定义了标准指令长度编码。
* 添加了对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

**前言**

**1 介绍**

1.1 RISC-V硬件平台术语

1.2 RISC-V软件执行环境与硬件线程

1.3 RISC-V ISA 概览

1.4 内存

1.5 基础指令长度编码

1.6 异常、陷入和中断

1.7 “未指定的”行为和值

**2 RV32I基础整数指令集（2.1版本）**

2.1 基础整数ISA的编程模型

2.2 基础指令格式

2.3 立即编码变量

2.4 整数运算指令

2.5 控制转移指令

2.6 加载和存储指令

2.7 内存排序指令

2.8 环境调用和断点

2.9 “提示”指令

**3 “Zifencei”指令获取屏障（2.0版本）**

**4 RV32E基础整数指令集（1.9版本）**

4.1 RV32E编程模型

4.2 RV32E指令集

**5 RV64I基础整数指令集（2.1版本）**

5.1 寄存器状态

5.2 整数运算指令

5.3 加载和存储指令

5.4 “提示”指令

**6 RV128I基础整数指令集（1.7版本）**

**7 用于乘法和除法的“M”标准扩展（2.0版本）**

7.1 乘法操作

7.2 除法操作

**8 用于原子指令的“A”标准扩展（2.1版本）**

8.1 指定原子指令的次序

8.2 加载-保留/存储-条件指令

8.3 存储-条件指令的最终正确完成

8.4 原子内存操作

**9 控制与状态寄存器（CSR）指令“Zicsr”（2.0版本）**

9.1 CSR指令

**10 计数器**

10.1 基础计数器和计时器

10.2 硬件性能计数器

**11 用于单精度浮点的“F”标准扩展（2.2版本）**

11.1 寄存器状态

11.2 浮点控制和状态寄存器

11.3 NaN的生成和传播

11.4 亚正常算法

11.5单精度加载和存储指令

11.6 单精度浮点运算指令

11.7 单精度浮点转换和移动指令

11.8 单精度浮点比较指令

11.9 单精度浮点分类指令

**12 用于双精度浮点的“D”标准扩展（2.2版本）**

12.1 D寄存器状态

12.2 较窄值的NaN装箱

12.3 双精度加载和存储指令

12.4 双精度浮点运算指令

12.5 双精度浮点转换和移动指令

12.6 双精度浮点比较指令

12.7 双精度浮点分类指令

**13 用于四精度浮点的“Q”标准扩展（2.2版本）**

13.1 四精度加载和存储指令

13.2 四精度运算指令

13.3 四精度转换和移动指令

13.4 四精度浮点比较指令

13.5 四精度浮点分类指令

**14 RVWMO内存一致性模型（0.1版本）**

14.1 RVWMO内存模型的定义

14.2 CSR依赖跟踪粒度

14.3 源寄存器和目的寄存器列表

**15 用于十进制浮点的“L”标准扩展（0.0版本）**

15.1 十进制浮点寄存器

**16 用于压缩指令的“C”标准扩展（2.0版本）**

16.1 概览

16.2 压缩指令格式

16.3 加载和存储指令

16.4 控制转移指令

16.5 整数运算指令

16.6 C指令在LR/SC序列中的使用

16.7 “提示”指令

16.8 RVC指令集列表

**17 用于位操作的“B”标准扩展（0.0版本）**

**18 用于动态翻译语言的“J”标准扩展（0.0版本）**

**19 用于事务内存的“T”标准扩展（0.0版本）**

**20 用于打包SIMD指令的“P”标准扩展（0.2版本）**

**21 用于向量操作的“V”标准扩展（0.7版本）**

**22 用于非对齐原子的“Zam”标准扩展（0.1版本）**

**23 用于全存储排序的“Ztso”标准扩展（0.1版本）**

**24 RV32/64G指令集列表**

**25 RISC-V汇编编程手册**

**26 扩充的RISC-V**

26.1 扩展术语

26.2 RISC-V扩展设计理念

26.3 定宽32位指令格式下的扩展

26.4 添加对齐的64位指令扩展

26.5 支持VLIW编码

**27 ISA扩展命名约定**

27.1 大小写敏感性

27.2 基础整数ISA

27.3 指令集扩展的命名

27.4 版本号

27.5 着重说明

27.6 附加的标准扩展的命名

27.7 管理员级指令集扩展

27.8 超级管理员级指令集扩展

27.9 机器级指令集枯燥

27.10 非标准扩展的命名

27.11 子集命名约定

**28 历史和鸣谢**

28.1 “为什么要开发一个新的ISA？”伯克利小组的理由

28.2 从ISA手册1.0版的修订历史

28.3 从ISA手册2.0版的修订历史

28.4 从2.1版的修订历史

28.5 从2.2版的修订历史

28.6 2.3版的修订历史

28.7 赞助

**A RVWMO的说明材料（0.1版本）**

A.1 为什么用RVWMO？

A.2 决定性检验

A.3 RVWMO规则的解释

A.3.1 保留的程序次序和全局内存次序

A.3.2 加载值公理

A.3.3 原子性公理

A.3.4 进程公理

A.3.5 重叠地址排序（规则1-3）

A.3.6 屏障（规则4）

A.3.7 显式同步（规则5-8）

A.3.8 句法依赖（规则9-11）

A.3.9 管道依赖（规则12-13）

A.4 超出主存范围

A.4.1 一致性和可缓存性

A.4.2 I/O排序

A.5 代码移植和映射指南

A.6 实现指南

A.6.1 未来可能的扩展

A.7 已知问题

A.7.1 混合尺寸的RSW

B 形式化的内存模型规范（0.1版本）

B.1 Alloy中的形式公理规范

B.2 Herd中的形式公理规范

B.3 一个内存操作模型

B.3.1 指令内的伪码执行

B.3.2 指令实例状态

B.3.3 硬件线程状态

B.3.4 共享内存状态

B.3.5 过渡

B.3.6 局限性

# 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、有序、解耦、乱序）或者实现技术（例如，全定制、ASIC、FPGA）而言，避免了“过度架构”，但在它们之中的任何一个中都能有效实现的ISA。
* 一个ISA被分成两个部分：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；以及可选的标准扩展，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[7]。
* 一个支持广泛的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、操作系统内核、和硬件实现。
* 一个支持高度并行的多核及其实现（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。
* 完全虚拟化的ISA，以便简化超级管理员的开发。
* 一个简化了新的特权架构设计的实验的ISA。

*关于我们的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[15]、RISC-II[8]、SOAR[21]和SPUR[11]）。我们也用罗马字母“V”双关表示“变种”和“向量”，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（尽管注解包含了由实现所驱动的决策）；它应当作为对软件可见的接口来阅读，对应于许多种实现，而不是作为某一特定硬件的定制品来设计。RISC-V手册的结构分为两卷。这一卷覆盖了基本的非特权指令的设计，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特权模式和特权架构而变化。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，例如缓存行尺寸，或者特权架构的细节，例如页面转换。这既是为了简化，也是为了允许各种微架构或各种权限架构的最大程度的灵活性。*

## 1.1 RISC-V硬件平台术语

一个RISC-V硬件平台可以包含：带有其它非兼容RISC-V核的一个或多个兼容RISC-V的处理核、固定功能加速器、各种物理内存结构、I/O设备，还有一个允许各组件通信的交互结构。

如果某个组件包含了一个独立的指令获取单元，那么它被称为一个*核*。一个兼容RISC-V的核可以通过多线程，支持多个兼容RISC-V的硬件线程（或称为：*hart*）。

RISC-V核可以有额外的专用指令集扩展，或者一个附加的*协处理器*。我们使用术语“*协处理器*”来指代某些被接到RISC-V核的单元，它们主要通过RISC-V指令流进行排序，但是包含了额外的架构状态和指令集扩展，并且可能包含与主RISC-V指令流相关的一些有限的自主权。

我们使用术语“*加速器*”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。在RISC-V系统中，我们预计有许多可编程加速器将是基于RISC-V的核，带有专用指令集扩展和/或定制的协处理器。RISC-V加速器的一个重要类别是I/O加速器，它把I/O处理任务的负荷从主应用核中卸了下来。

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微控制器到一个有数千节点的集群，其中每个节点都是一个共享内存多核服务器。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，来模块化开发工作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（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权限模式）、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管理员二进制接口（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，操作码陷入和软件仿真可以被用于实现硬件里没有提供的功能。执行环境实现的例子包括：

* “裸机”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。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超级管理员：为宾客操作系统提供了多个管理员级别的执行环境。
* RISC-V仿真器：例如Spike、QEMU或rv8，在一个底层的x86系统上仿真RISC-V硬件线程，并提供一个用户级别的或者管理员级别的执行环境。

*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比硬件更抽象的接口，因为EEI越抽象，它所提供的跨不同硬件平台的可移植性越大。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。*

从软件在给定的执行环境中运行的观点看，硬件线程是一种资源，它在那个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（hart）行动得像是一种硬件线程资源，即使执行环境将时间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解构，例如，通过环境调用来派生新的硬件线程。

执行环境负责确保它的各个硬件线程的最终推进。对于一个给定的硬件线程，当硬件线程正在运作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。硬件线程的推进是由下列事件构成的：

* 一个指令的引退。
* 一个陷入，就像1.6节定义的那样。
* 由组成硬件进程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上的工作[13, 14]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，环境的硬件线程（hart）的操作就像硬件的线程一样。*

*一个执行环境实现可能将一组宾客硬件线程，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”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有）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，和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，并因此提供了一个便捷的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。

尽管很容易讲出*这个*“RISC-V ISA”，但其实RISC-V是一系列相关ISA的ISA族，其中目前有四个基础ISA。每个基础整数指令集有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2章和第5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4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6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（二补数）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，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。也许最终会需要更大的扁平128位地址空间，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，在增加地址空间尺寸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上仿真另一个时所需的硬件复杂化（例如，在RV64I上仿真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仿真不同的基础ISA。我们注意到，较新的SPARC和MIPS ISA修订版已经强烈反对不经改变就在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的最早的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意义了。*

*已经被注意到，我们能够将\*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已经被设计为支持扩展定制和规范。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，而且我们把每个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。标准编码由基金会定义，不能与用于相同基础ISA的其它标准扩展冲突。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。我们使用术语“非标准的”来描述那些不是由基金会所定义的扩展。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。我们使用术语“非合规的”来描述一个非标准的扩展，如果它使用了标准编码或者保留编码（例如，自定义扩展不是非合规的）。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26章描述了扩展RISC-V ISA的各种方法。我们也已经为基于RISC-V的指令和指令集扩展开发了一个命名约定，那将在第27章进行详细的描述。

为了支持更一般的软件开发，定义了一组标准扩展来提供整数乘法/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、和原子的指令，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和标准GC扩展之外，我们相信很少还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2XLEN字节的单字节可寻址空间，用于所有的内存访问。内存的一个“*字*”被定义为32位（4字节）。对应地，一个“*半字*”是16位（2字节），一个“*双字*”是64位（8字节），而一个“*四字*”是128位（16字节）。内存地址空间是环形的，所以位于地址2XLEN-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2XLEN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设备的读写可以造成可见的副作用，但是访问主内存不可以。虽然执行环境可能把硬件线程地址空间中的所有内容都称作I/O设备，但是通常都会期望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，有专门的加载和存储指令对内存执行*显式*的读或写。执行环境可能强制要求指令的执行实施其他隐式的内存访问（例如实现地址转换）而不是为非特权ISA实施那些记录的内存访问。

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，可以被指令读取隐式读到的位置集合，是否可以与那些可以被加载指令显式读到的位置集合有交叠；以及，可以被存储指令显式写到的位置集合，可能只是能被读到的位置的一个子集。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，不引发异常又没有副作用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。例如，一个有效的实现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。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定义的、特定的屏障或缓存控制指令（例如第3章里定义的FENCE.I指令）。

由硬件线程发起的内存访问（隐式或显式）可能出现以不同的顺序发生，因为注意到被另一个硬件线程、或者任何其它代理所发起的访问也能访问到相同的内存。然而，这个注意到的内存访问重新排序总是受到适用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14章和附录中。或者，一种实现也可以采用更强的模型，全存储排序，定义在第23章中。执行环境也可以添加约束，进一步限制注意到的内存访问的重排。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需要，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扩展的，每条指令在长度上可以是任意数目的16位指令包，指令包在16位边界自然对齐。第16章中描述的压缩ISA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）来表示实现所执行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，放松的IALIGN是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）来表示被实现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。支持更长指令的实现也有更大的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”。

### 扩展的指令长度编码

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，而且不认为下面的关于编码超过32位指令的提议已被冻结。

带有超过32位编码的标准指令集扩展将额外的低序位设置为1，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，使用一个3位的域来编码，在位[14:12]中给出了除最先的5×16位字以外的16位字的数目。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。

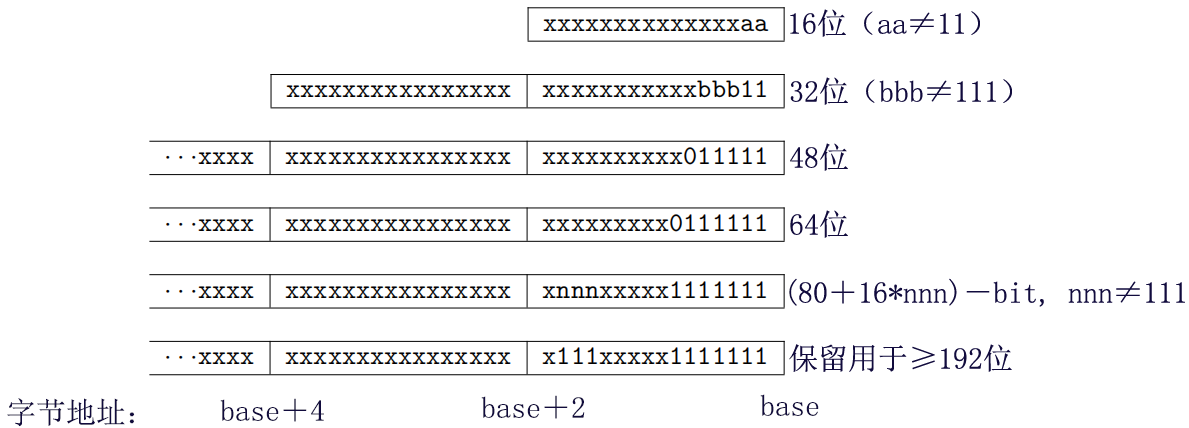


图1.1： RISC-V指令长度编码。此时只有16位和32位编码被认为是被冻结的。

*给定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，我们不想强制使用压缩的格式。我们也希望可选地允许更长的指令，以支持实验和更大的指令集扩展。尽管我们的编码约定需要更严格的核心RISC-V ISA编码，但是这仍然有许多有益的效果。*

*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，任何遭遇有低位被清除的指令应当在存进缓存之前，被重新编码为非法的30位指令，以保留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凝结成32位指令字的子集，我们为非标准的和自定义的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26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，同时保留对标准≥32位指令集扩展的支持。甚至，如果实现也不需要长度＞32位的指令，它可以为不一致的扩展恢复另外四个主要的操作码。*

位[15:0]都是0的编码被定义为非法指令。这些指令被认为具有最小的长度：16位，如果任何16位指令集扩展存在，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观测到的其它通常的样式。*

*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的机器的标准二进制库）。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，让可变指令长度的获取和解码变得复杂。*

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位小字节序的包的序列被存储在内存中。形成一个指令的包被存储在递增的半字地址处，最低地址的包持有指令规范中最低的若干位。

*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，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，以及基于假定大字节序处理器构建的特定遗留代码，所以我们已经定义了RISC-V的大字节序和双字节序变体。*

*我们不得不固定指令包在内存中存储的顺序，独立于内存系统的字节序之外，来确保长度编码位始终以半字地址顺序首先出现。这允许指令获取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大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活动的字节序，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，有时可以让仔细编写的软件的字节序不可知，甚至是以二进制的形式，就像与位置无关的代码一样。*

*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编译器必须在向指令内存存储的时候，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”来指代一种发生在运行时的不寻常的状况，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”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”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者的过程。

下面的章节中的指令描述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者的陷入（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”与IEEE-754浮点标准中的相兼容。*

陷入是如何被处理的，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：

**被包含的陷入**：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管理员模式和用户模式的EEI中，用户模式硬件线程的ECALL通常将导致控制转移到运行在相同硬件线程上的一个管理员模式的处理者。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管理员模式中的中断处理者。

**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。例子包括仿真缺失的指令、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，或者允许监视计时器失效。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。

下面的表格显示了每种陷入的特点：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 被包含 | 被请求 | 不可见的 | 致命的 |
| 执行终止？ | N | N1 | N | Y |
| 软件被遗忘？ | N | N | Y | Y2 |
| 由环境处理？ | N | Y | Y | Y |

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

EEI为每个陷入定义了它是否被精确处理，尽管建议是尽可能地保持精度。被包含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。不可见的陷入，根据它们的性质，超出了这篇文档的讨论范围。没有在此定义的指令编码，和没有被一些其它方法定义的指令编码，可以引起致命陷入。

## 1.7 “未指定的”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，扩展应当完全设计允许的行为和值，并使用术语“未指定的”用于有意不约束的情况。在这些情况中，可以被其它的扩展、平台标准或实现约束或定义。

# 第二章 RV32I基础整数指令集，2.1版本

这章描述了2.1版本的RV32I基础整数指令集。

*RV32I被设计为足以形成编译器目标和支持现代操作系统环境的。该ISA也被设计为在最小的实现中减少对硬件的需求。RV32I包含40条各不相同的指令，尽管在简单的实现中，可能会用一个总是陷入的系统硬件指令来覆盖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可以仿真几乎任何其它的ISA扩展（除了A扩展，因为它需要对原子性的额外的硬件支持）*

*实际上，一个包括了机器模式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的子集可能是有用的，但是“基础”已经定义了，应当很少有动机对一个真实的硬件实现进行子集，除了忽略对非对齐的内存访问的支持，和把所有的系统指令视为一个单独的陷入。*

*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被硬布线为所有位都等于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布尔值的集合、或者二进制有符号整数或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**，保持了当前指令的地址。

|  |  |  |
| --- | --- | --- |
| XLEN-1 | 0 | |
| x0 / 零 | | |
| x1 | | |
| x2 | | |
| x3 | | |
| x4 | | |
| x5 | | |
| x6 | | |
| x7 | | |
| x8 | | |
| x9 | | |
| x10 | | |
| x11 | | |
| x12 | | |
| x13 | | |
| x14 | | |
| x15 | | |
| x16 | | |
| x17 | | |
| x18 | | |
| x19 | | |
| x20 | | |
| x21 | | |
| x22 | | |
| x23 | | |
| x24 | | |
| x25 | | |
| x26 | | |
| x27 | | |
| x28 | | |
| x29 | | |
| x30 | | |
| x31 | | |
| XLEN | | |
| XLEN-1 | | 0 |
| pc | | |
| XLEN | | |

表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来保持一个调用的返回地址，以及寄存器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返回使用x1或x5。见JAL和JALR指令的描述。*

*可选的压缩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。使用其它约定的软件将正确地操作，但是可能会得到更大的代码尺寸。*

*可用的架构寄存器数目可以对代码尺寸、性能、和能量消耗有很大的影响。尽管16个寄存器对于运行已编译的代码的一个整数ISA来说理应是足够的，但是使用3-地址格式在16位指令中使用16个寄存器编码一个完整的ISA还是不可能的。尽管2-地址格式将是可能的，但是它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，并且一旦采用了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有帮助，其中可以有广泛使用的循环展开、软件管道和缓存平铺。*

*由于这些原因，我们为基础ISA选择了一个32个整数寄存器的约定尺寸。动态寄存器使用往往由一些频繁被访问的寄存器所控制，而寄存器文件的实现可以被优化，以减少对频繁访问寄存器的访问能耗[20]。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（或者扁平，或者分层）。*

*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4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，并且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，目标地址没有按四字节对齐，将生成一个指令地址未对齐的异常。这个异常在分支或跳转指令上汇报，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，这个对基础ISA指令的对齐约束被放宽到按两字节边界对齐。*

*在分支或跳转上汇报的指令地址未对齐异常将导致指令未对齐，以帮助调试，并简化IALIGN＝32的系统的硬件设计，因为该系统是唯一可能发生未对齐的地方。*

上面解码一个保留指令的行为是“未指定的”。

*一些平台可能需要保留的操作码，为标准使用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不合规的扩展。*

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9章）中使用的5位立即数，立即数总是符号扩展的，并且通常在指令中被打包在最左端的可用位，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。

*解码寄存器标识符通常用在实现中的关键路径上，因此选择指令的格式，以便在所有格式中都在相同的位置保存所有的寄存器标识符，作为代价，不得不跨格式移动立即数位（一个分享自RISC-V的属性，又称SPUR[11]）。*

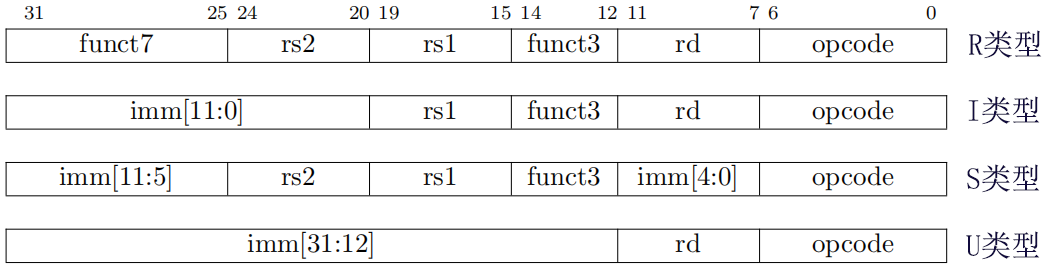


图2.2：RISC-V基础指令格式。每个立即数子域都用正被产生的立即数值中的位位置（imm[x]）的标签标记，而不是像通常做的那样，用指令立即数域中的位位置。

*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”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（像在MIPS ISA中的），我们没有观察到使用零扩展的收益，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。

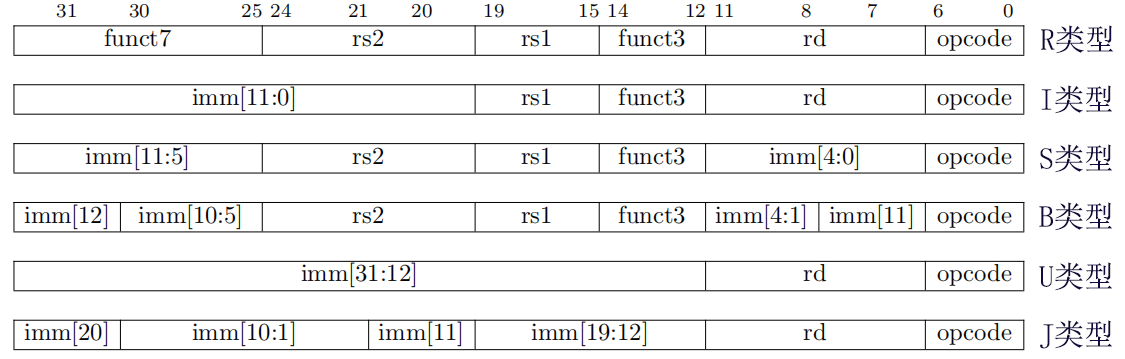


图2.3：显示立即数的RISC-V基础指令格式

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2.4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。

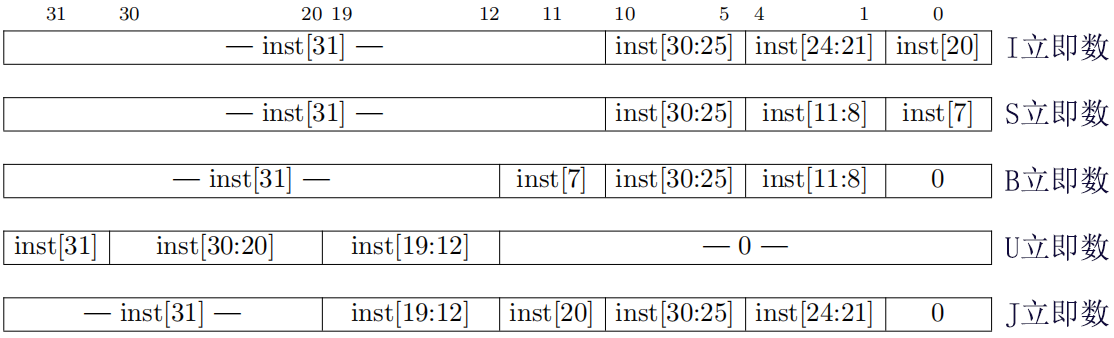


图2.4：由RISC-V指令产生的立即数的类型。用构造了它们值的指令位对域进行了标记。符号扩展总是使用inst[31]。

*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有用于分支和跳转计算的独立加法器，并因此将不能从保持立即数位的位置跨指令类型不变中获得好处，我们仍然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muxes，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和立即数mux的开销。加扰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负载，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。整数运算指令或者编码为使用I类型格式的寄存器-立即数操作，或者编码为使用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，这利用了该观察：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***

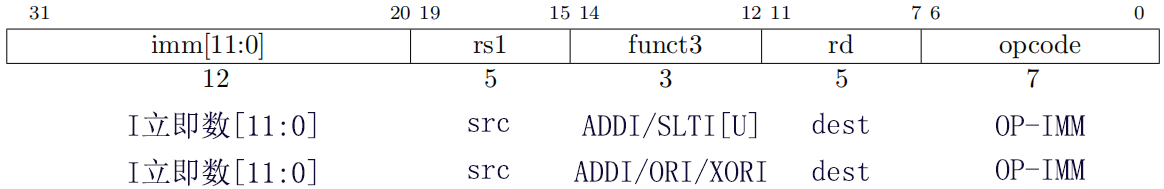
***slti t3, t2, 0***

***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。*

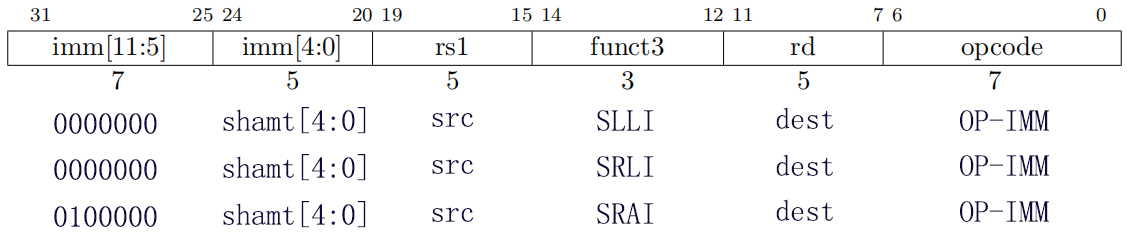
### 整数寄存器 - 立即数指令



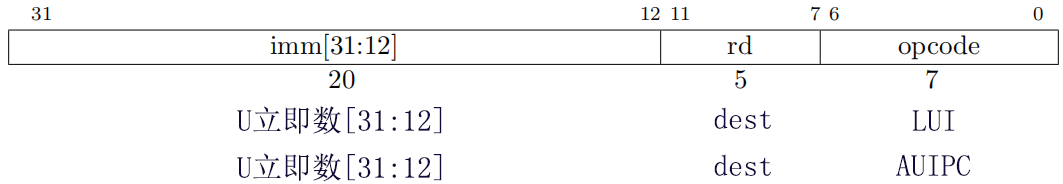
ADDI将符号扩展的12位立即数加到寄存器*rs1*上。简单地将结果的低XLEN位当作结果，而忽略了算数溢出。ADDI *rd*, *rs1*, *0*被用于实现MV *rd*, *rs1*汇编器伪指令。

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（设置少于立即数）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*）。



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



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U立即数值放在目的寄存器*rd*的最高20位中，同时把最低的12位用零填充。

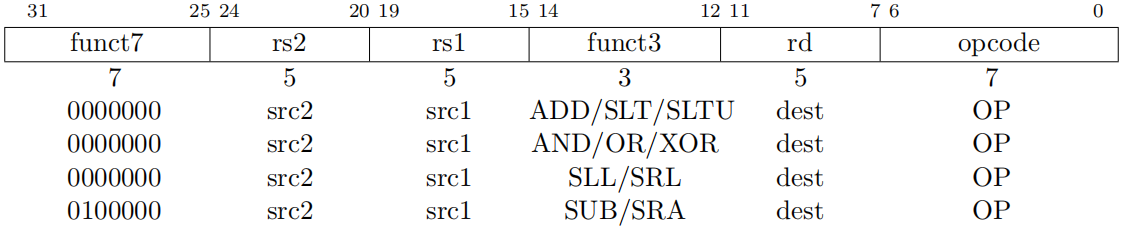
AUIPC（加高位立即数到**pc**）被用于构建**pc**相关的地址，它使用U类型格式。AUIPC根据20位U立即数形成32位偏移量（最低12位填零），把这个偏移量加到AUIPC指令的地址，然后把结果放在寄存器*rd*中。

*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相关的地址，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相关的数据地址。*

*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后续指令的），它可能引起更简单微架构中的管道破裂或者更复杂微架构中的BTB结构污染。*

### 整数寄存器 - 寄存器操作

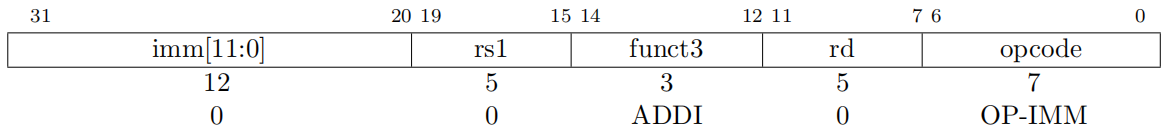
RV32I定义了一些算数R类型操作。所有操作都读取*rs1*寄存器和*rs2*寄存器作为源操作数，并将结果写入寄存器*rd*。*funct7*域和*funct3*域选择了操作的类型。



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*r1*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



除了提升**pc**和递增任何适用的性能计数器以外，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可用于HINT指令（第2.9节）。*

*选用ADDI进行NOP编码是因为，这是在跨多个系统中最可能的采取最少资源来执行的方法（如果解码中没有优化的话）。特别地，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。特别地，地址生成功能单元可以使用相同的基址+偏移量地址计算所需的硬件来执行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

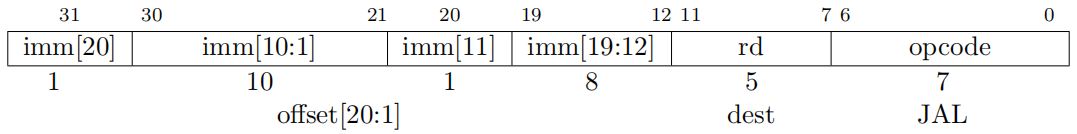
RV32I提供两种类型的控制转移指令：无条件跳转和条件分支。RV32I中的控制转移指令*没*有架构上可见的延迟槽。

### 无条件跳转

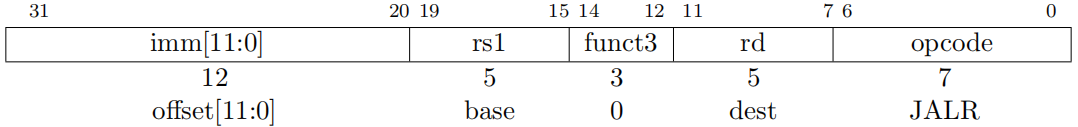
跳转和链接（JAL）指令使用J类型格式，那里J立即数以2字节的倍数编码一个有符号的偏移量。偏移量是符号扩展的，加到跳转指令的地址上以形成跳转目标地址。跳转可以因此到达的目标范围是±1MiB。JAL把跳转随后的指令的地址（**pc**+4）存储到寄存器*rd*中。标准软件调用约定使用**x1**作为返回地址寄存器，使用**x5**作为备用的链接寄存器。

*备用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，因为它映射到了标准调用约定中的一个临时调用，并且其编码与常规链接寄存器相比只有一位不同。*

扁平的无条件跳转（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*。如果结果是不需要的，寄存器**x0**可以被用作目的寄存器。



*无条件跳转指令都使用PC相关的地址来帮助支持位置无关代码。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。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相关的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***基础使用时，JALR可以被用于实现地址空间中从任何地方到最低2KiB或最高2KiB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四字节边界，JAL和JALR指令将产生一个指令地址未对齐异常。

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

返回地址预测栈是高性能指令获取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指令应当如表2.1中显示的那样推入/弹出一个RAS。

*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。*

|  |  |  |  |
| --- | --- | --- | --- |
| *rd* | *rs1* | *rs1*＝*rd* | RAS行为 |
| !*link*  !*link*  *link*  *link*  *link* | !*link*  *link*  !*link*  *link*  *link* | -  -  -  0  1 | 无  弹出  推入  弹出，然后推入  推入 |

表2.1：指令中使用的寄存器说明符中编码的返回地址栈预测提示。上面，当寄存器是**x1**或**x5**时，*link*是true。

*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（或者是***x1***，或者是***x5***），RAS只把允许宏操作融合推入序列：***lui ra, imm20; jalr ra, imm12(ra) and auipc ra, imm20; jalr ra, imm12(ra)***。*

### 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±4KiB。



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。*

软件应当被优化为，按顺序的代码路径是占大部分的常见路径，而线路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相关的，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管道，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。另一点是，与零比较需要非平凡的电路延迟（特别是在高级进程中移动到高级静态逻辑后），并因此与算数等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。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管道的开销[6, 10, 9]，并且已经在商业处理器中被实现[17]。最简单的技术只是通过只冲刷分支阴影中的指令而不是整个获取管道，或者通过使用宽指令获取或空闲指令获取槽从两端获取指令，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被推测性地执行，而与其它代码的执行顺序不一致[17]。*

如果目标地址没有对齐到四字节边界，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可能发生在支持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

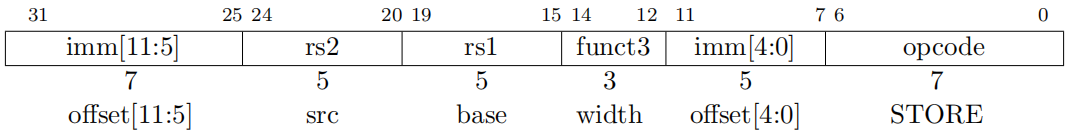
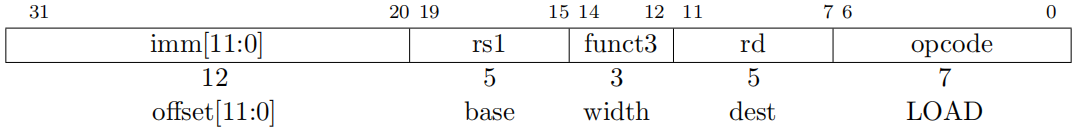
RV32I是一个“加载-存储”架构，那里只有加载和存储指令访问内存，而算数指令只操作CPU寄存器。RV32I提供一个32位的地址空间，按字节编址。EEI将定义该地址空间的哪一部分是哪个指令可以合法访问的（例如，一些地址可能是只读的，或者只支持按字访问）。即使所加载的值被丢弃，以**x0**为目的的加载仍然必须引发任何异常并引起任何其它的副作用。

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：如果一个字节以某些字节序被存储到内存的某些地址，那么从那个地址以任何字节序加载一个字节尺寸都将返回被存储的值。*

*在一个小字节序的配置中，多字节的存储在最低的内存字节地址处写入最低有效位的寄存器字节，然后按它们有效性的升序写入其它的寄存器字节。加载类似，把较小的内存字节地址的内容传输到较低有效性的寄存器字节。*

*在一个大字节序的配置中，多字节的存储在最低的内存字节地址处写入最高有效位的寄存器字节，然后按它们有效性的降序写入其它的寄存器字节。加载类似，把较大的内存字节地址的内容传输到较低有效性的寄存器字节。*



加载和存储指令在寄存器和内存之间传输值。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，但是然后在存储到*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。有效地址没有自然对齐到引用的数据类型的加载和存储（即，32位访问对齐到四字节边界，16位访问对齐到二字节边界），其行为依赖于EEI。

EEI可以保障完全支持未对齐的加载和存储，并因此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实现，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，如果未对齐的访问不应当被仿真，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（允许软件运行在执行环境中以处理该陷入）或者致命陷入（终止执行）。

*当移植遗留代码时，偶尔需要未对齐的访问；且在使用任何形式的打包SIMD扩展、或者处理外部打包的数据结构时，对应用程序的性能有帮助。对于允许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，简化添加额外的未对齐硬件支持。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分离的ISA支持：或者是一些特殊指令来帮助软件处理未对齐访问，或者是一个用于未对齐访问的新的硬件编址模式。特殊指令难以使用、让ISA复杂化，并经常添加新的处理器状态（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）。此外，对于面向循环的打包SIMD代码，当操作数未对齐时的额外负担迫使软件根据操作数的对齐方式提供多种形式的循环，这使代码的生成复杂化，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，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，尽管自然对齐的加载和存储被保证原子执行，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。

*我们没有授权未对齐访问的原子性，所以执行环境实现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。如果提供了硬件未对齐支持，软件可以通过简单地使用常规加载和存储指令利用它。然后，硬件可以根据运行时地址是否对齐自动优化访问。*

## 2.7 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）的任意组合可以与同样这些的任意组合进行排序。非正式地，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*集合中的任何操作之前，观察到FENCE之后的*后继*集合中的任何操作。第14张提供了RISC-V内存一致性模型的一个精确的描述。

EEI将定义什么I/O操作是可能的，且特别地，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中描述同样使用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  其它的：*保留供未来使用。* |
| 其它 | | *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。

可选的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。

*FENCE.TSO编码作为一个可选的扩展被添加到原始的基础FENCE指令编码。基础定义需要实现忽略任何设置位，并把FENCE当作全局的，因此这是一个向后兼容的扩展。*

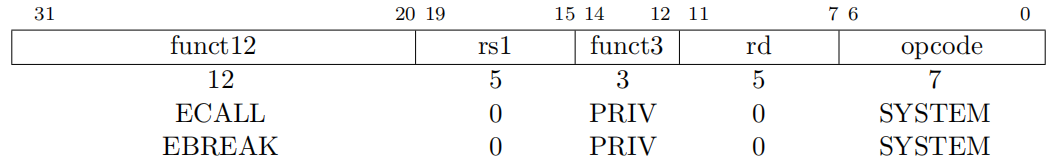
FENCE指令中的未使用的域——*rs1*和*rd*——被保留用于未来扩展中的更细粒度的屏障。为了向前兼容，基础实现应当忽略这些域，而标准软件应当把这些域置为零。同样地，表2.2中的许多*fm*和前驱/后继集合设置也被保留供将来使用。基础实现应当把所有这些保留的配置视为普通的*fm*＝0000的屏障，而标准软件应当只使用非保留的配置。

*我们选择了一个放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添加的协处理器或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权限的系统功能，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），和所有其它潜在的特权指令。CSR指令在第9章描述，而基础非特权指令在接下来的小节中描述。

*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者。更复杂的实现可能在硬件中执行更多的各系统指令。*



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，而不只是调用一个管理员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16章里描述的压缩的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，这些通常被用于向微架构交流性能提示。HINT被编码为*rd*=**x0**的整数运算指令。因此，像NOP指令，HINT不改变任何架构上的可视状态，除了提升**pc**和任何适用的性能计数器。实现总是被允许忽略已编码的提示。

*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运算指令。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。*

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，但是目前那些标准HINT都还没有被定义。剩余的HINT空间被保留用于自定义的HINT：在这个子空间中，将永远不会定义标准HINT。

*现在还没有定义标准的提示。我们预计标准的提示最终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** | 220 | 保留供未来标准使用 |
| AUIPC | *rd*＝**x0** | 220 |
| ADDI | *rd*＝**x0**，并且要么rs1≠x0，要么imm≠0 | 217－1 |
| ANDI | *rd*＝**x0** | 217 |
| ORI | *rd*＝**x0** | 217 |
| XORI | *rd*＝**x0** | 217 |
| ADD | *rd*＝**x0** | 210 |
| SUB | *rd*＝**x0** | 210 |
| AND | *rd*＝**x0** | 210 |
| OR | *rd*＝**x0** | 210 |  |
| XOR | *rd*＝**x0** | 210 |
| SLL | *rd*＝**x0** | 210 |
| SRL | *rd*＝**x0** | 210 |
| SRA | *rd*＝**x0** | 210 |
| FENCE | *pred*＝0或succ＝0 | 25－1 |

|  |  |  |  |
| --- | --- | --- | --- |
| SLTI | *rd*＝**x0** | 217 | 保留供自定义使用 |
| SLTIU | *rd*＝**x0** | 217 |
| SLLI | *rd*＝**x0** | 210 |
| SRLI | *rd*＝**x0** | 210 |
| SRAI | *rd*＝**x0** | 210 |
| SLT | *rd*＝**x0** | 210 |
| SLTU | *rd*＝**x0** | 210 |

表2.3：RV32I HINT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[19]）。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。*

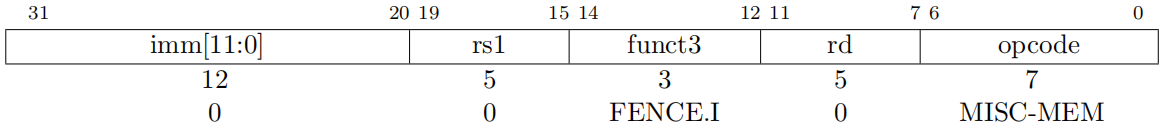
*FENCE.I指令被设计为支持多种实现。简单的实现可以在FENCE.I被执行的时候冲刷本地指令缓存和指令管道。更加复杂的实现可以在每个数据（指令）缓存缺失的时候嗅探指令（数据）缓存，或者在主指令缓存中的某些行正在被本地存储指令写入时，使用一个包容统一的私有L2缓存使其无效化。如果指令和数据缓存以这种方式保持一致性，或者如果内存系统只由未缓存的RAM组成，那么只有获取管道需要在FENCE.I被冲刷。*

*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，对于拥有一致性指令缓存和一致性数据缓存、或者指令缓存的重新填充不会嗅探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。*



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 RV32E基础整数指令集（1.9版本）

这章描述了一个RV32E基础整数指令集的建议草案，它是为嵌入式系统设计的一个RV32I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和RV32I之间的不同，并因此应当被放在第2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。尽管我们已经在此文档的2.0版本中提到了这个可能性，但是我们最初是拒绝定义这个子集的。然而，考虑到对最小可能的32位微控制器的需求，和在这个空间中抢占碎片的利益，我们现在已经定义了RV32E，作为除RV32I、RV64I和RV128I之外的第四个标准基础ISA。为了减少高线程64位处理器的上下文状态，我们也有定义RV64E的兴趣。*

## 4.1 RV32E编程模型

RV32E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内核区域总数中的大约四分之一，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。*

*这个变化需要不同的调用约定和ABI。特别地，RV32E只使用软浮点调用约定。正在考虑一个新的嵌入式ABI，它将跨RV32E和RV32I工作。*

## 4.2 RV32E指令集

RV32E使用与RV32I相同的指令集编码，但是只提供寄存器**x0 - x15**。通过减少寄存器标识符所释放出的指令位，任何未来的标准扩展都不会使用，因而这些指令位可以被用于自定义的扩展。

*RV32E可以与所有当前的标准扩展进行组合。曾经考虑过将F、D和Q扩展定义为，在与RV32E组合时有16项浮点寄存器文件，但是后来决定不这样做。为了支持减少了浮点寄存器状态的系统，我们打算定义一个“Zfinx”扩展，让浮点运算使用整数寄存器，而移除在浮点寄存器和整数寄存器之间的浮点的加载、存储和移动。*

# 第五章 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2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。

## 5.1 寄存器状态

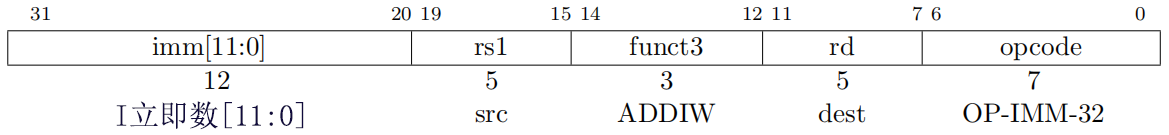
RV64I把整数寄存器和所支持的用户地址空间拓宽到64位（表2.1中XLEN=64）。

## 5.2 整数运算指令

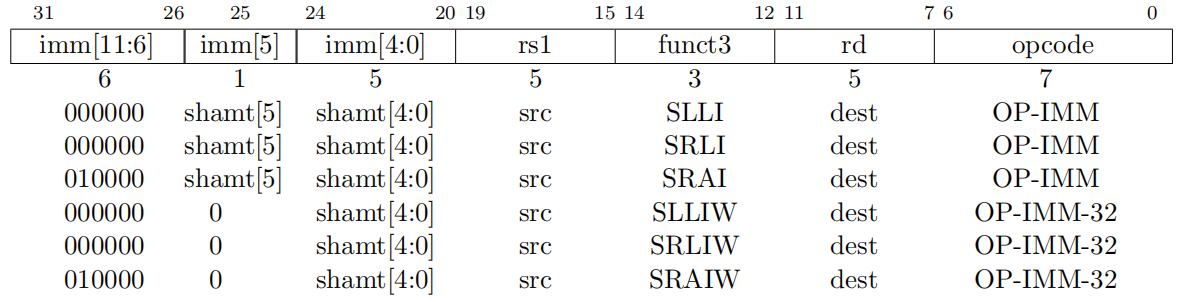
大多数整数运算指令在XLEN位的值上操作。在RV64I中提供了额外的指令变体来操作32位的值，通过在操作码上添加“W”后缀来表示。这些“\*W”指令忽略了它们的输入的高32位，并且总是产生32位有符号的值，也就是说从XLEN-1位到31位是相等的。

*编译器和调用约定维持了一个不变量，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。在这个不变量下，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



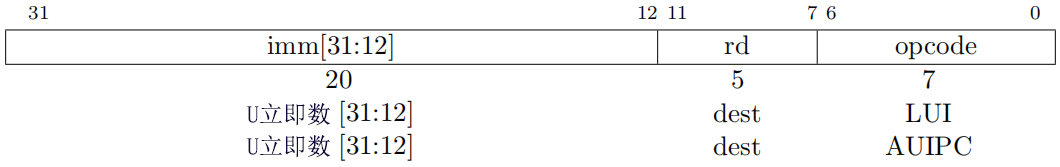
ADDIW是一个RV64I指令，它把符号扩展的12位立即数加到寄存器*rs1*，并在*rd*中产生合适的32位符号扩展的结果。运算结果的低32位符号扩展到64位作为结果，而忽略了溢出。注意，ADDIW *rd*, *rs1*, *0*把寄存器*rs1*的低32位的符号扩展写入寄存器*rd*（汇编器伪指令SEXT.W）。



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。右移类型被编码在位30中。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，并产生有符号的32位结果。带有imm[5]≠0的SLLIW、SRLIW和SRAIW的编码是保留的。

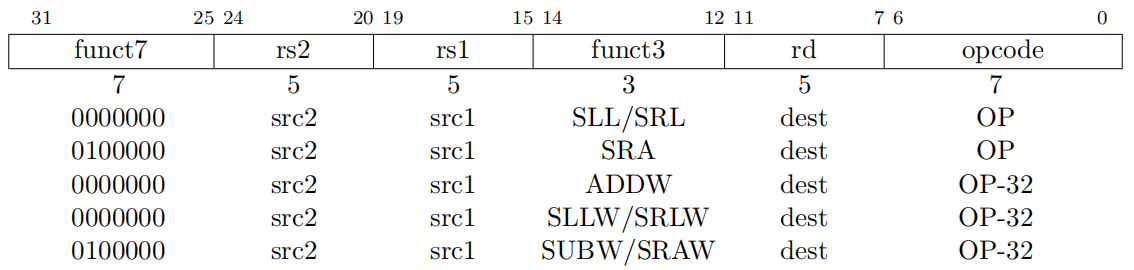
*先前，*imm[5]≠0*的SLLIW、SRLIW和SRAIW被定义为，将引发非法指令异常，而现在它们被标记为保留的。这是一个向后兼容的改变。*



LUI（加载高位立即数）使用与RV32I相同的操作码。LUI把20位的U立即数放进寄存器*rd*的位31~12中，并把最低的12位置零。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被用于构建与pc有关的地址，并使用U类型格式。AUIPC将12个低阶零位追加到20位U立即数，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。

**整数寄存器—寄存器操作**



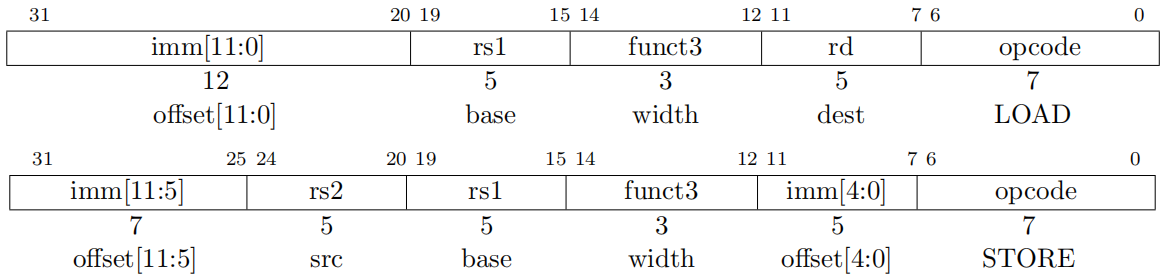
ADDW和SUBW是RV64I独有的指令，它们的定义类似于ADD和SUB，但是在32位值上操作，并产生有符号的32位结果。溢出被忽略，且结果的低32位被符号扩展到64位，并写到目的寄存器。

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，并产生有符号的32位结果。移位数量由*rs2[4:0]*给出。

## 5.3 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。

## 5.4 “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5.1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空间被保留用于自定义HINT：标准HINT将永远不会被定义在这个子空间中。

|  |  |  |  |
| --- | --- | --- | --- |
| 指令 | 约束 | 代码点 | 目的 |

|  |  |  |  |
| --- | --- | --- | --- |
| LUI | *rd*＝**x0** | 220 | 保留供未来标准使用 |
| AUIPC | *rd*＝**x0** | 220 |
| ADDI | *rd*＝**x0**，并且要么rs1≠x0，要么imm≠0 | 217－1 |
| ANDI | *rd*＝**x0** | 217 |
| ORI | *rd*＝**x0** | 217 |
| XORI | *rd*＝**x0** | 217 |
| ADDIW | *rd*＝**x0** | 217 |
| ADD | *rd*＝**x0** | 210 |
| SUB | *rd*＝**x0** | 210 |
| AND | *rd*＝**x0** | 210 |
| OR | *rd*＝**x0** | 210 |
| XOR | *rd*＝**x0** | 210 |
| SLL | *rd*＝**x0** | 210 |
| SRL | *rd*＝**x0** | 210 |
| SRA | *rd*＝**x0** | 210 |
| ADDW | *rd*＝**x0** | 210 |
| SUBW | *rd*＝**x0** | 210 |
| SLLW | *rd*＝**x0** | 210 |
| SRLW | *rd*＝**x0** | 210 |
| SRAW | *rd*＝**x0** | 210 |
| FENCE | *pred*＝0或succ＝0 | 25－1 |

|  |  |  |  |
| --- | --- | --- | --- |
| SLTI | *rd*＝**x0** | 217 | 保留供自定义使用 |
| SLTIU | *rd*＝**x0** | 217 |
| SLLI | *rd*＝**x0** | 211 |
| SRLI | *rd*＝**x0** | 211 |
| SRAI | *rd*＝**x0** | 211 |
| SLLIW | *rd*＝**x0** | 210 |
| SRLIW | *rd*＝**x0** | 210 |
| SRAIW | *rd*＝**x0** | 210 |
| SLT | *rd*＝**x0** | 210 |
| SLTU | *rd*＝**x0** | 210 |

表 5.1：RV64I HINT 指令。

# 第六章 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1PB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。一些仓库规模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100PB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（加载双无符号）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。

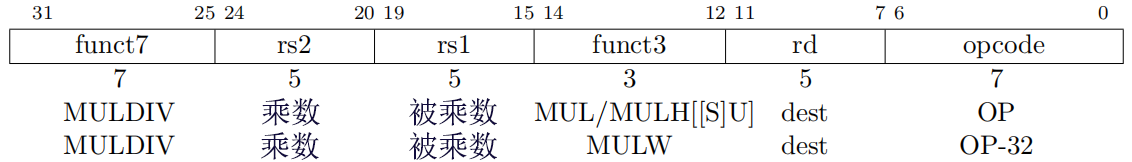
浮点指令集没有变化，尽管128位Q浮点扩展现在可以支持FMV.X.Q和FMV.Q.X指令，以及来往于T（128位）整数格式的额外的FCVT指令。

# 第七章 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或在相接的加速器中能更好地处理的应用。

## 7.1 乘法操作



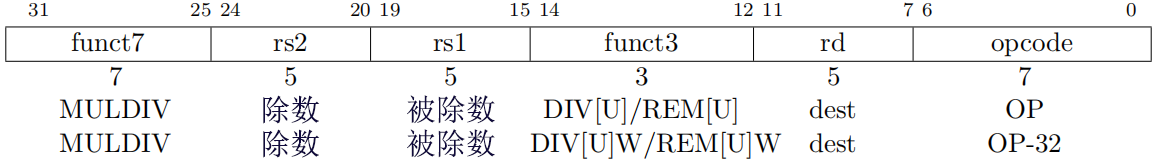
MUL在*rs1*和*rs2*上实施XLEN位×XLEN位乘法，并将低XLEN位放入目的寄存器。MULH、MULHU和MULHSU分别针对有符号数×有符号数、无符号数×无符号数、有符号*rs1*×无符号*rs2*实施相同的乘法，但是返回2xXLEN位结果的高XLEN位。如果同一个乘积的高位和低位都需要，那么推荐的代码序列是：MULH[[S]U] *rdh*, *rs1*, *rs2*; MUL *rdl*, *rs1*, *rs2*（源寄存器标识符必须次序相同，且*rdh*不能与*rs1*或*rs2*相同）。然后微架构可以把这些代码融合进单独的一次乘法操作，而不是两次分离的乘法。

*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，然而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]。*

## 7.2 除法操作



DIV和DIVU在*rs1*和*rs2*上实施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进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7.1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复数被－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。无符号除法溢出不可能发生。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 条件 | 被除数 | 除数 |  | DIVU[W] | REMU[W] | DIV[W] | REM[W] |
| 除以零  溢出（仅有符号） | x  －2L－1 | 0  －1 |  | 2L－1  —— | x  —— | －1  －2L－1 | x  0 |

表7.1：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）。

*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。*

# 第八章 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。提供了两种形式的原子指令，是“加载-预约/存储条件”指令和“原子性获取和操作内存”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[5]。

*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。*

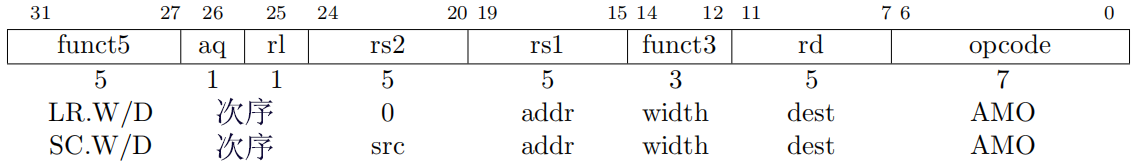
## 8.1 指定原子指令的次序

基础的RISC-V ISA有一个宽松的内存模型，其中FENCE指令被用于推动额外的次序约束。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同时的访问进行排序。

为了对释放一致性[5]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。

## 8.2 加载-预约/存储-条件指令



加载-预约（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*预约集*——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：当且仅当预约仍然有效且预约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预约失效。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”。*

值1的失败代码被保留来编码未指定的失败。其它失败代码目前被保留，可移植的软件应当只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仿真，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，仿真未对齐的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预约集的可能性，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预约集，提供包括被编址的数据字或双字的所有字节的预约。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预约集的存储能够在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，将之作为预约集的一部分。

*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。当它们访问预约集中的其它字节时，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的预约集中。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预约集的存储发生。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。（如果这个设备写了预约集但是没有写由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14.1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预约集的尺寸和形状的方法。*

*平台规范可以约束预约集的尺寸和形状。例如，期望Unix平台需要主内存的预约集是固定尺寸的、连续的、自然对齐的，并且不大于虚拟内存页的尺寸。*

*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预约强制失效：*

* *在抢占式上下文切换期间，和*
* *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预约的页时。*

*如果一个LR或SC暗示，硬件线程当时只持有一个预约，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14.1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。*

在建立了预约的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**

**# a1 持有期望的(expected)值**

**# a2 持有需要的(desired)值**

**# a0 持有返回值，如果成功则为零，否则为非零。**

**cas:**

**lr.w t0, (a0) # 加载原始值。**

**bne t0, a1, fail # 不匹配，所以失败。**

**sc.w t0, a2, (a0) # 尝试更新。**

**bnez t0, cas # 如果存储条件失败，那么重试**

**li a0, 0 # 设置返回值为成功。**

**jr ra # 返回。**

**fail:**

**li a0, 1 # 设置返回值为失败。**

**jr ra # 返回。**

图8.1：使用LR/SC的比较与交换功能的样例代码。

LR/SC可以被用于构造无锁的数据结构。图8.1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。

## 8.3 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、FENCE.I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*终结性（eventuality）*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仿真。*

*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。允许实现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指令的预约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指令的预约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预约集。
* *H*执行了一次分支或跳转而退出了受约束的LR/SC循环。
* *H*陷入了。

*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预约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预约集，不保证任何硬件线程将退出它的LR/SC循环。*

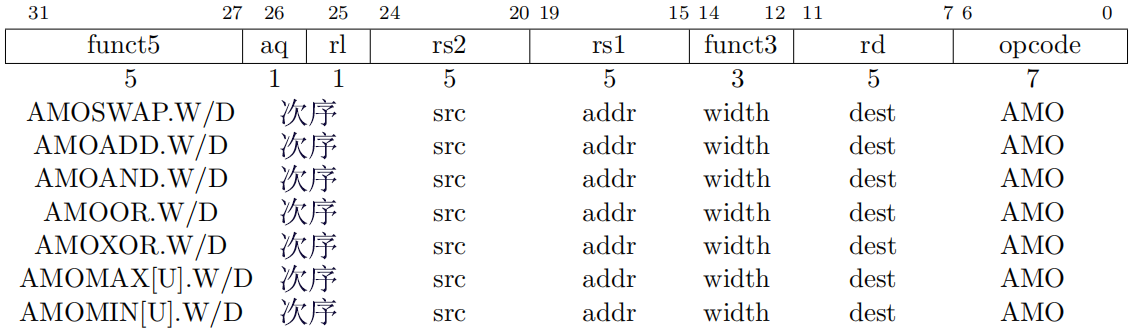
*加载和加载-预约指令本身不会阻碍其它硬件线程的LR/SC序列的进程。我们注意到这个约束意味着，除此之外，其它硬件线程执行的加载和加载预约指令（可能在同一个核中）不能无限阻碍LR/SC进程。例如，由共享缓存的另一个硬件线程引起的缓存收回不能无限地阻碍LR/SC进程。典型地，这意味着对预约的追踪是独立于任何共享缓存的收回的。类似地，由硬件线程内的推测性执行引起的缓存缺失不能无限地阻碍LR/SC进程。*

*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，即使在某些系统上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。*

## 8.4 原子内存操作



原子内存操作（AMO）指令为多处理器同步实施读-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。

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仿真，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22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“获取和操作”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。相反，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*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被设计为有效地实现C11和C++11内存模型。尽管FENCE R, RW指令足以实现获取操作，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8.2中显示了一个通过“测试和测试和设置”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[16]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。**

**again:**

**lw t1, (a0) # 检查锁是否被占用。**

**bnez t1, again # 如果锁被占用则重试。**

**amoswap.w.aq t1, t0, (a0) # 尝试获取锁。**

**bnez t1, again # 如果锁被占用则重试。**

**# ...**

**# 关键小节**

**# ...**

**amoswap.w.r1 x0, x0, (a0) # 通过存储0来释放锁。**

图8.2：互斥的样例代码。**a0**包含了锁的地址。

# 第九章 控制与状态寄存器（CSR）指令“Zicsr”（2.0版本）

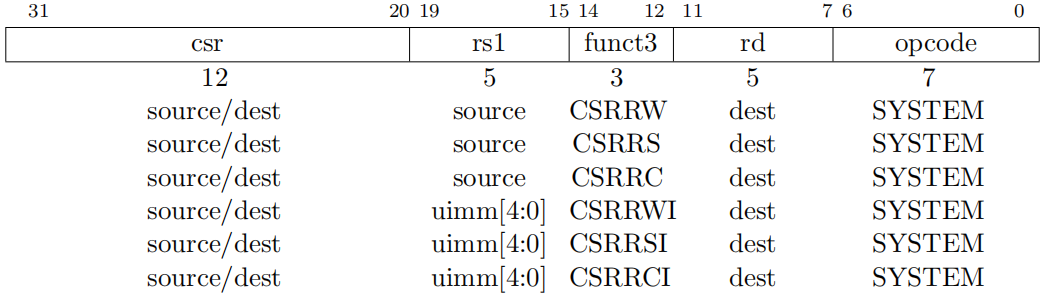
RISC-V定义了一个独立的地址空间，包含与各硬件线程相关联的4096个控制和状态寄存器。这章定义了操作在这些CSR上的CSR指令的完整集合。

*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章节被移出，进入了这个独立的章节。*

## 9.1 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。立即数形式使用一个5位的零扩展立即数，编码在rs1域中。



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。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 寄存器操作数 | | | | |
| 指令 | rd | rs1 | 读CSR？ | 写CSR？ |
| CSRRW  CSRRW  CSRRS/C  CSRRS/C | x0  !x0  ——  —— | ——  ——  x0  !x0 | 否  是  是  是 | 是  是  否  是 |
| 立即数操作数 | | | | |
| 指令 | rd | uimm | 读CSR？ | 写CSR？ |
| CSRRWI  CSRRWI  CSRRS/CI  CSRRS/CI | x0  !x0  ——  —— | ——  ——  0  !0 | 否  是  是  是 | 是  是  否  是 |

表9.1：显示了CSR指令是否读取或写入一个给定CSR的表。CSRRS和CSRRC指令有相同的行为，所以在表中被显示为CSRRS/C。

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中的其它位不受影响（虽然CSR可能在写的时候会有副作用）。

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中的其它位不受影响。

对于CSRRS和CSRRC，如果*rs1*＝**x0**，那么指令将完全不会写CSR，并因此也应当不会引起任何只可能在写CSR时发生的副作用，例如在访问只读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，并且不应当引起任何只可能在写CSR时发生的副作用。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9.1总结了CSR指令在它们是否读和/或写CSR方面的行为。

*迄今为止定义的CSR，除了在不允许的访问上产生非法指令异常，在读取方面没有任何架构方面的副作用。自定义扩展可能添加在读取方面有副作用的CSR。*

一些CSR，例如指令失效计数器、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CSRWI *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

在一个给定的硬件线程上，显式和隐式的CSR访问以程序次序实施，至于那些指令，其执行行为受到被访问的CSR的状态影响。特别地，CSR访问的实施，或者在任何程序次序先前的指令执行之后，或者在任何程序次序后续的指令执行之前（如果这些先前/后续的指令的行为是修改CSR状态或者被CSR状态修改的话）。此外，CSR读访问指令是在指令执行之前返回被访问的CSR状态，而CSR写访问指令则是在指令执行之后更新被访问的CSR状态。

在上面的程序次序不成立的地方，CSR的访问是弱有序的，并且本地硬件线程或者其它硬件线程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。*

*推行这些CSR次序主要用于支持主内存和内存映射I/O访问关于读***time** *CSR的排序。除了***time***、***cycle***、和***mcycle** *CSR以外，在本规范的第一卷和第二卷中到目前为止定义的CSR不能直接访问其它硬件线程或设备，也不能引起对其它硬件线程或设备的可见的副作用。因此，除了之前提到的三个，对CSR的访问可以自由地根据FENCE指令重排次序而不会违背本规范。*

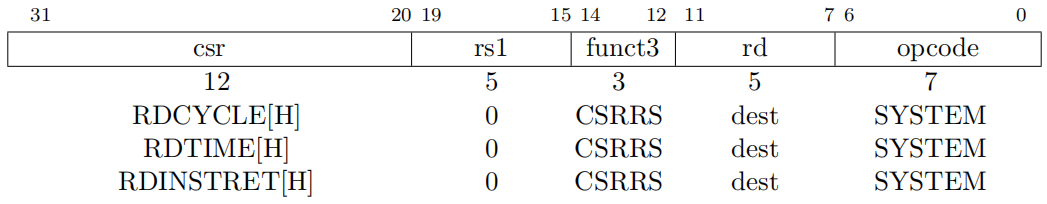
对于引起副作用的CSR访问，上面的排序约束适用于那些副作用的初始次序，但是不一定适用于那些副作用的完成次序。

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。

# 第十章 计数器

RISC-V ISA提供了一组至多32×64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**（在RV32上，高32位通过CSR寄存器**0xC80 - 0xC9F**）来访问。它们中的前三个（CYCLE、TIME和INSTRET）有专门的功能（分别是周期计数、实时时钟，和指令退出），而剩余的计数器（如果被实现了）提供可编程事件的计数。

## 10.1 基础计数器和计时器



RV32I提供了许多64位只读的用户级计数器，它们被映射进12位CSR地址空间，使用CSRRS指令以32位片段访问。在RV64I中，CSR指令可以操作64位CSR。特别地，RDCYCLE、RDTIME和RDINSTRET伪指令读取**cycle**计数器、**time**计数器和**instret**计数器的全64位。因此，RDCYCLEH、RDTIMEH和RDINSTRETH指令在RV64I中是不需要的。

*一些执行环境可能会保护对计数器的访问来阻止定时侧信道攻击。*

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是一个RV32I指令，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”。给定实现（包括软件仿真）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/核的地方，人们会希望有失效的周期计数/指令来测量硬件线程的CPI。*

*完全不必要将核暴露给软件，而且实现者可能选择让一个物理核上的多个硬件线程假装运行在一个硬件线程/核上的分离的多个核上，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，在任何情况中对总共每个核的周期计数的需求，导致了每个核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挂钟时间的真实时间。RDTIMEH是一个RV32I专有的指令，它读取相同真实时间计数器的位**63 - 32**。在实际中，底层64位计数器应当永远不会溢出。执行环境应当提供一种决定真实时间计数器（秒/滴答）周期的方法。该周期必须是恒定的。在一个单独的用户应用中的所有硬件线程的真实时间时钟应当被同步到真实时钟的一个滴答内。环境应当提供一种决定时钟精度的方法。

*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，但是在这种情况中，平台应当把RDTIME指令实现为RDCYCLE的一个别名，以使代码更具有可移植性，而不是使用RDCYCLE来衡量挂钟时间。*

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是一个RV32I专有的指令，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10.1：RV32中读取64位周期计数器的样例代码。

*我们推荐在实现中提供这些基本的计数器，因为它们对于基本性能分析、自适应和动态优化是必要的，并且允许应用使用实时流。应当提供额外的计数器来帮助诊断性能问题，并且用户级应用程序代码应当可以低负载地访问这些计数器。*

*我们要求计数器是64位宽的，即使在RV32上也是如此，否则软件将非常困难来决定值是否已经溢出。对于一个低端的实现，每个计数器的高32位可以使用软件计数器来实现，其增加通过低32位溢出触发的陷入实现。上面描述的样例代码显示了全64位宽的值是如何能够使用独立的32位指令安全地读取的。*

*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。*

## 10.2 硬件性能计数器

为29个额外的非特权64位硬件性能计数器，**hpmcounter3 - hpmcounter31**，分配了CSR空间。对于RV32，可通过额外的CSR **hpmcounter3h - hpmcounter31h**来访问这些性能计数器的高32位。这些计数器统计与平台相关的事件，并通过额外的特权寄存器来配置。这些额外计数器的数目和宽度，以及它们计数的事件集合都是特定于平台的。

*特权架构手册描述了控制访问这些计数器和把事件设置为可被计数的特权CSR。*

*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。*

# 第十一章 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[7]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。

## 11.1 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11.1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。*

|  |
| --- |
| FLEN－1 0 |
| f0 |
| f1 |
| f2 |
| f3 |
| f4 |
| f5 |
| f6 |
| f7 |
| f8 |
| f9 |
| f10 |
| f11 |
| f12 |
| f13 |
| f14 |
| f15 |
| f16 |
| f17 |
| f18 |
| f19 |
| f20 |
| f21 |
| f22 |
| f23 |
| f24 |
| f25 |
| f26 |
| f27 |
| f28 |
| f29 |
| f30 |
| f31 |
| FLEN |
| 31 0 |
| fcsr |
| 32 |

图11.1：RISC-V标准F扩展单精度浮点状态。

## 11.2 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11.2中显示的那样。

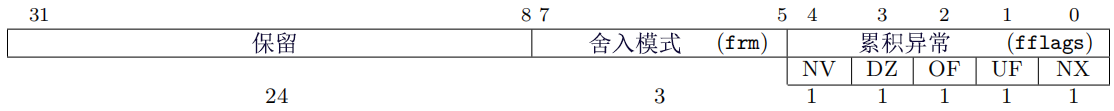


图11.2：浮点控制和状态寄存器。

**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，包括用于十进制浮点的“L”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11.1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。如果**frm**被设置为无效值（101 - 111），任何尝试使用动态舍入模式执行浮点操作的子序列都将引起一个非法指令异常。某些指令，包括拓宽转换，尽管有*rm*域但是不会被舍入模式影响；软件应当把它们的*rm*域设置为RNE（000）。

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。

|  |  |  |
| --- | --- | --- |
| 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－∞） |
| 011 | RUP | 向上舍入（向＋∞） |
| 100 | RMM | 就近舍入，关联到最大幅度 |
| 101 |  | *无效，保留供未来使用。* |
| 110 |  | *无效，保留供未来使用。* |
| 111 | DYN | 在指令的rm域中，选择动态舍入模式；  在舍入模式寄存器中，*无效*。 |

表11.1：舍入模式编码。

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11.2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11.2：累积异常标志编码。

*正如标志所允许的那样，我们不支持在基础ISA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。*

## 11.3 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。*

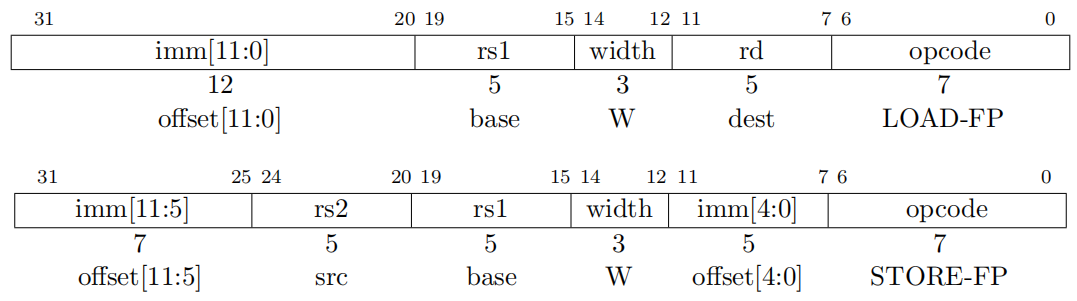
## 11.4 亚正常算法

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。*

## 11.5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，一个在寄存器*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。

## 11.6 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*之间实施单精度浮点加法和乘法。FSUB.S实施从*rs1*中减去*rs2*的单精度浮点减法。FDIV.S实施*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11.3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）。

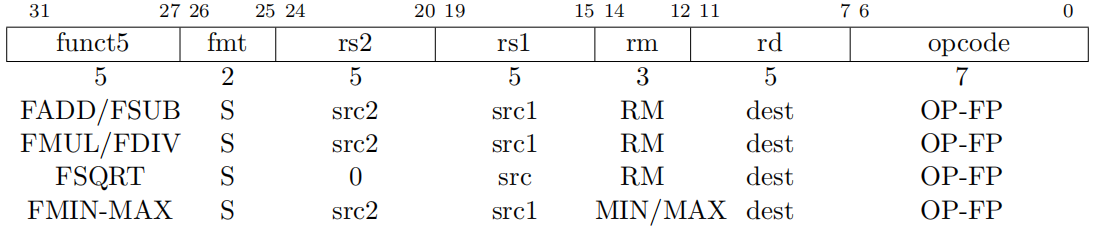
|  |  |  |
| --- | --- | --- |
| *fmt*域 | 助记符 | 含义 |
| 00 | S | 32位单精度 |
| 01 | D | 64位双精度 |
| 10 | H | 16位半精度 |
| 11 | Q | 128位四精度 |

表11.3：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11.1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。*



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。

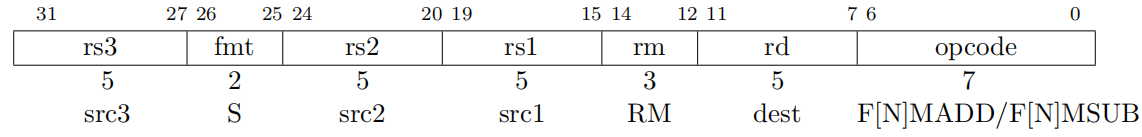
FMADD.S将*rs1*和*rs2*中的值相乘，加上*rs3*中的值，并把最终结果写到*rd*。FMADD.S计算(*rs1*×*rs2*)＋rs3。

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。*



*融合乘加（FMA）指令会消费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∞和零时，融合乘加指令必须设置无效操作异常标志，即使加数是沉默的NaN时也需如此。

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

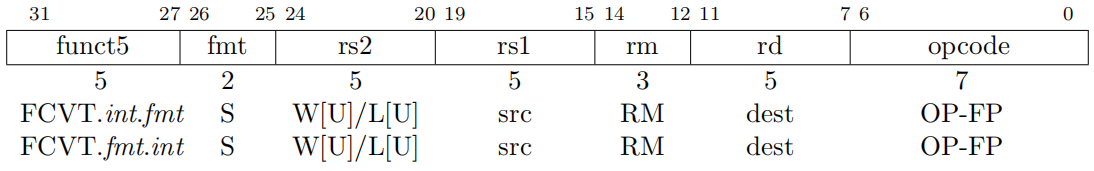
## 11.7 单精度浮点转换和移动指令

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11.4给出了FCVT.int.S的有效输入的范围和无效输入的行为。

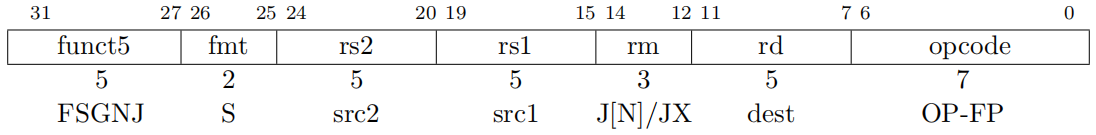
所有的浮点到整数转换指令和整数到浮点转换指令都根据*rm*域进行舍入。浮点寄存器可以使用FCVT.S.W *rd*, *x0*被初始化为浮点正零，它将永远不会设置任何异常标志。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | FCVT.W.S | FCVT.WU.S | FCVT.L.S | FCVT.LU.S |
| 最小有效输入（舍入后）  最大有效输入（舍入后） | －231  231－1 | 0  232－1 | －263  263－1 | 0  264－1 |
| 对于超出范围的负数输入的输出  对于－∞的输出  对于超出范围的正数输入的输出  对于＋∞或NaN的输出 | －231  －231  231－1  231－1 | 0  0  232－1  232－1 | －263  －263  263－1  263－1 | 0  0  264－1  264－1 |

表11.4：浮点到整数转换的域和对于无效输入的行为。



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*）。

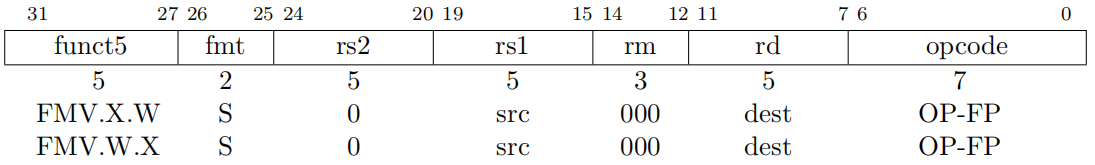


*符号注入指令提供了浮点MV、ABS和NEG，也支持了少量其它的操作，包括IEEE copySign操作和超越数学函数库中的符号操作。尽管MV、ABS和NEG只需要一个寄存器操作数，而FSGNJ指令需要两个，但大多数微架构将不太可能添加优化，来从读取这些相对不频繁的指令的寄存器数目的减少中受益。甚至在这种情况中，微架构也可以为FSGNJ指令做简单地检测，当两个源寄存器是相同的时，只读取一份拷贝。*

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。*

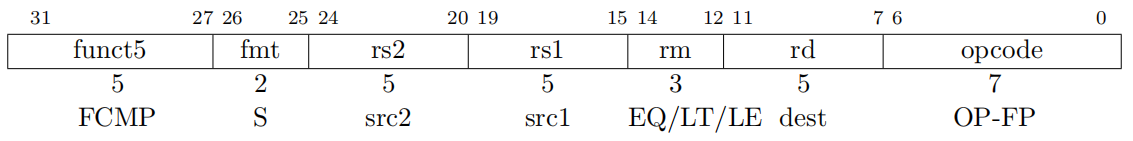


*基础浮点ISA被如此定义，是为了允许实现在寄存器中采用浮点格式的内部重新编码，以简化对亚正常值的处理，并可能减少功能单元的延迟。为此，基础ISA避免在浮点寄存器中，通过定义直接读写整数寄存器文件的转化和比较操作来表示整数值。这也去除了许多常见的需要在整数寄存器和浮点寄存器之间显式移动的情况，为常见的混合格式代码序列减少了指令计数和关键路径。*

## 11.8 单精度浮点比较指令

浮点比较指令（FEQ.S、FLT.S、FLE.S）在浮点寄存器之间实施特定的比较（*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。

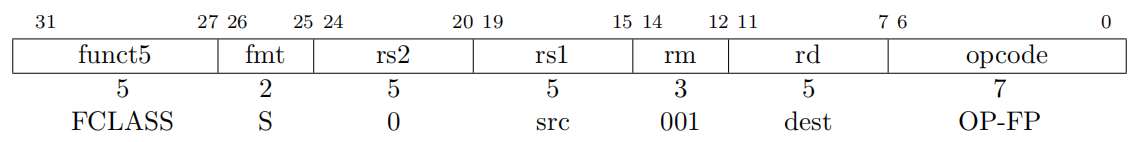
FLT.S和FLE.S实施IEEE 754-2008标准所提及的信号比较：即，如果某个输入是NaN，它们设置无效操作异常标志。FEQ.S实施沉默比较：它只在某个输入是发信号的NaN时设置无效操作异常标志。对于所有这三个指令，如果有操作数是NaN，那么结果就是0。



*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。*

## 11.9 单精度浮点分类指令

FCLASS.S指令检测浮点寄存去*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11.5中。如果属性位真，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。



|  |  |
| --- | --- |
| *rd*位 | 含义 |
| 0 | *rs1*是－∞ |
| 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*是一个正的亚正常的数 |
| 6 | *rs1*是一个正的正常的数 |
| 7 | *rs1*是＋∞ |
| 8 | *rs1*是发信号的NaN |
| 9 | *rs1*是沉默的NaN |

表11.5：FCLASS指令的结果的格式。

# 第十二章 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。

## 12.1 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11.1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12.2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。*

## 12.2 较窄值的NaN装箱

当支持多个浮点精度时，较窄的*n*位类型（*n*＜FLEN）的有效值表示在一个FLEN位NaN值的低*n*位中，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*≤FLEN）,有效的NaN装箱的*n*位值表现为负的沉默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。*

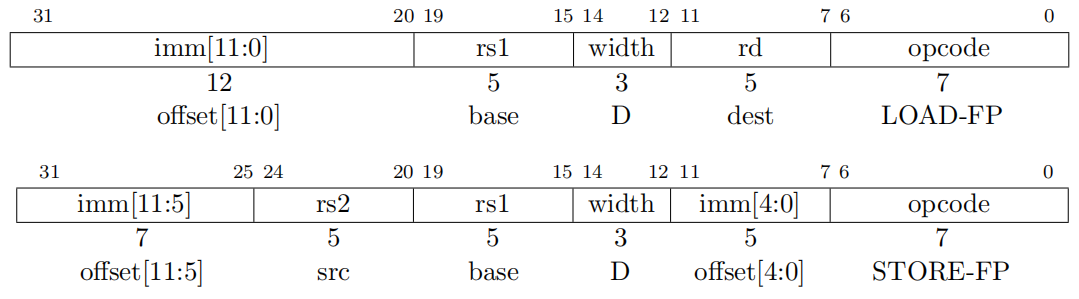
*非重新编码的实现在每个浮点指令的输入和输出上把操作数按IEEE标准格式解包和打包。对一个非重新编码的实现的NaN装箱开销主要在于，检查较窄操作的高位是否表示了一个合法的NaN装箱的值，以及把结果的高位全写成1。*

*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。*

## 12.3 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。*

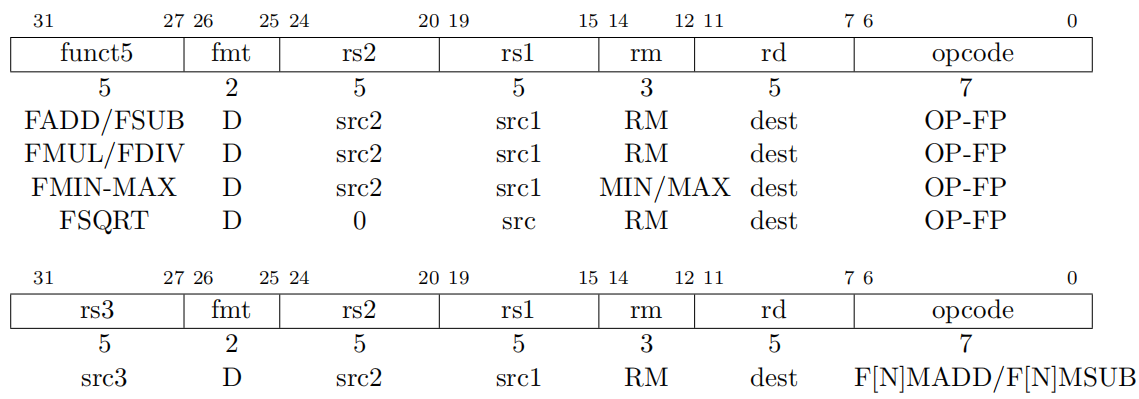


只有当有效的地址被自然对齐，并且XLEN≥64时，FLD和FSD才保证原子性执行。

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。

## 12.4 双精度浮点运算指令

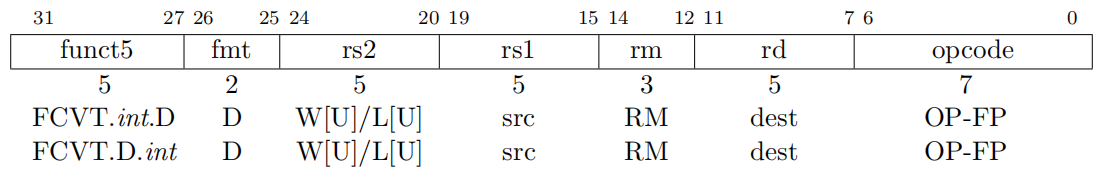
双精度浮点运算指令的定义与它们对应的单精度指令的定义类似，但是操作在双精度操作数上，并产生双精度的结果。



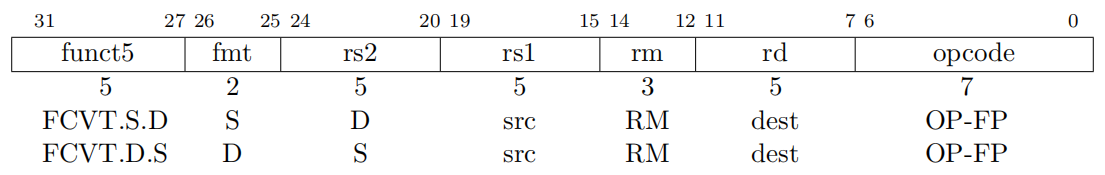
## 12.5 双精度浮点转换和移动指令

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去rd中。FCVT.WU.D，FCVT.LU.D，FCVT.D.WU和FCVT.D.LU变体转化或转化为无符号整数值。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。

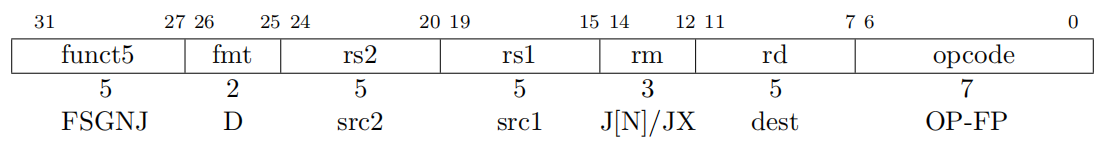
所有的浮点到整数转换指令和整数到浮点转换指令都根据rm域进行舍入。注意FCVT.D.W[U]总是产生确切的结果，而不会被舍入模式影响。



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。

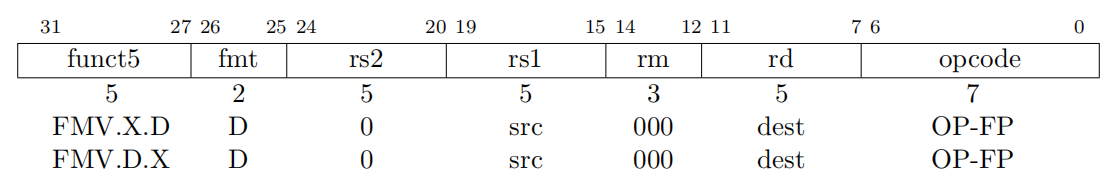


浮点到浮点的符号注入指令，FSGNJ.D、FSGNJN.D和FSGNJX.D的定义与对应的单精度符号注入指令的定义类似。



只当XLEN≥64时，提供了在浮点和整数寄存器之间按位式样移动的指令。FMV.X.D把浮点寄存去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。



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，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。

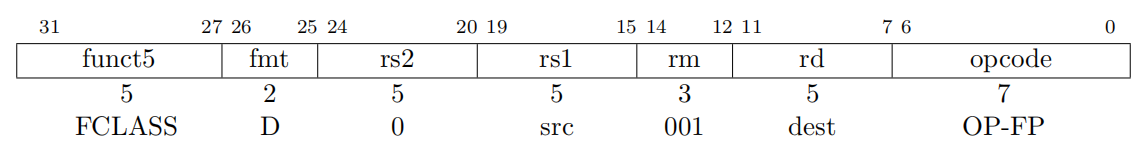
我们注意到，对于实现了64位浮点单元（包括融合的乘加支持）和64位浮点加载与存储的系统来说，从32位整数移动到64位整数的数据路径的外围硬件开销较低，而支持32位宽地址空间和指针的软件ABI可以被用于避免静态数据的增长和动态内存的拥塞。

## 12.6 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，但是操作在双精度操作数上。

## 12-612.7 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，但是操作在双精度操作数上。

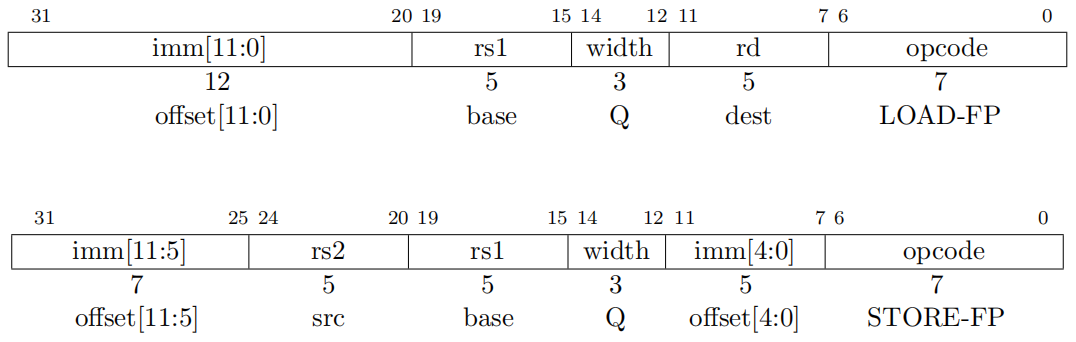


# 第十三章 用于四精度浮点的“Q”标准扩展（2.2版本）

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12.2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。

## 13.1 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。

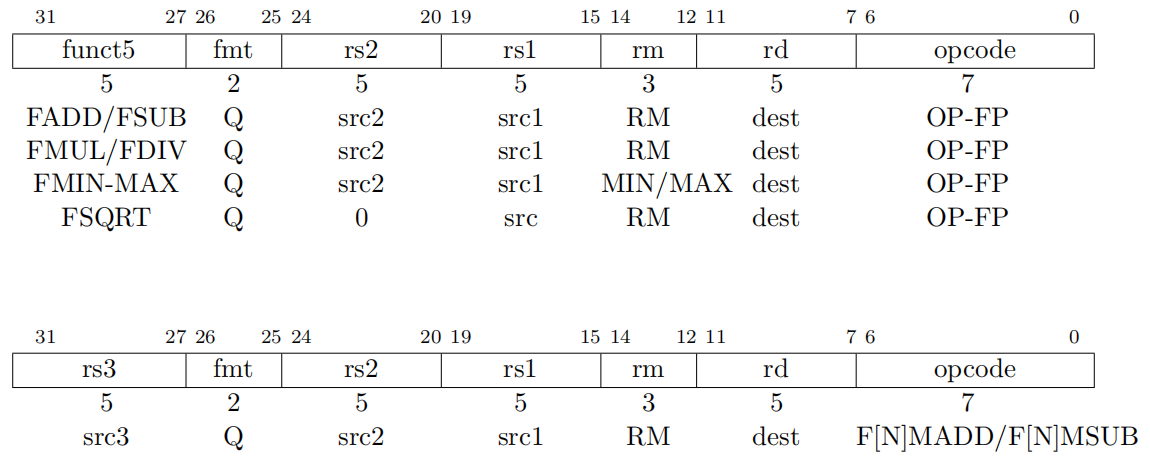
## 13.2 四精度运算指令

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13.1中显示的那样。

|  |  |  |
| --- | --- | --- |
| 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 |

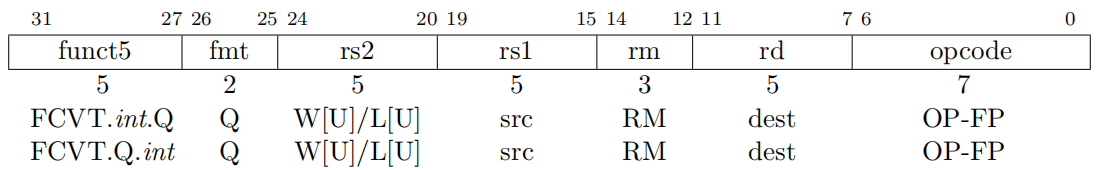
表13.1：格式域编码。

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，但是操作在四精度操作数上，并产生四精度的结果。

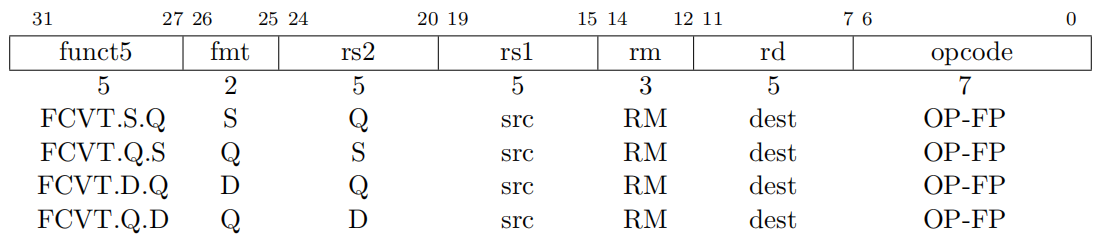


## 13.3 四精度转换和移动指令

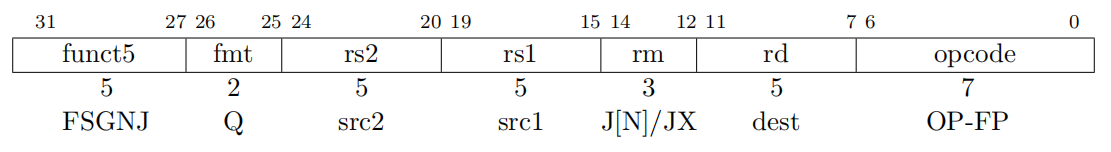
添加了新的浮点到整数转化指令和整数到浮点转化指令。这些指令的定义与双精度到整数转化指令和整数到双精度转化指令的定义类似。FCVT.W.Q或者FCVT.L.Q分别把一个四精度浮点数转化为一个有符号的32位或64位整数。FCVT.Q.W或FCVT.Q.L分别把一个32位或64位有符号整数转化为一个四精度浮点整数。FCVT.WU.Q、FCVT.LU.Q、FCVT.Q.WU和FCVT.Q.LU变体转化或转化为无符号整数值。FCVT.L[U].Q和FCVT.Q.L[U]是RV64独有的指令。



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。



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。

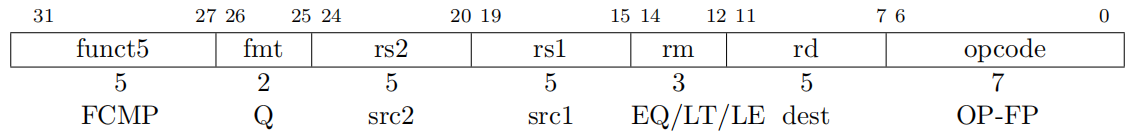


在RV32或RV64中不提供FMV.X.Q和FMV.Q.X指令，所以四精度位式样必须经过内存被移动到整数寄存器。

*RV128将在Q扩展中支持FMV.X.Q和FMV.Q。*

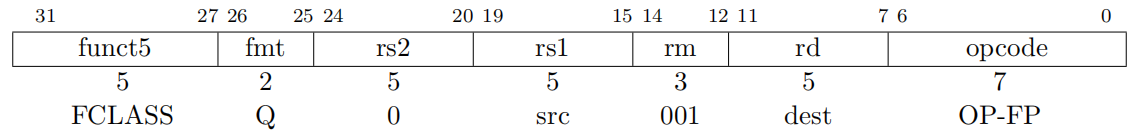
## 13.4 四精度浮点比较指令

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，但是操作在四精度操作数上。



## 13.5 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，但是操作在四精度操作数上。



# 第十四章 RVWMO内存一致性模型（0.1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22章）和用于全存储排序的标准ISA扩展“Ztso”（第23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、“T”事务内存扩展和“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。*

## 14.1 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12.3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22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仿真。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8.2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[5]。*

*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23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14.2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，14.3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，寄存器r是源寄存器，而不是x0：

* 在i的操作码中，rs1、rs2或者rs3倍设置为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14.3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，寄存器r是目的寄存器，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14.3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14.3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

* 管道依赖

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的各字节都返回某个存储所写入的字节，该存储为下列存储中在全局次序里最近的那个：

1. 写该字节，并且在全局内存次序中先于i的存储

2. 写该字节，并且在程序次序中先于i的存储

### 原子性公理

如果r和w是由硬件线程h中的对齐的LR和SC指令生成的成对的加载和存储操作，s是一个对字节x的存储，r返回s写入的值，那么在全局内存次序中，s必须先于w，并且以全局内存次序在s之后和w之前，除了h到字节x的存储之外，没有其它的存储。

*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

在全局内存次序中，任何内存操作之前都不能有其它内存操作的无限序列。

## 14.2 CSR依赖跟踪粒度

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

表14.1：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。

## 14.3 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14.1节中的句法依赖。

术语“累积CSR”被用于描述一个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、从“源寄存器”列中的各个源寄存器到“累积CSR”列中的各个CSR，以及从“累积CSR”列中的各个CSR到其自身。

要点：

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 源寄存器 | 目的寄存器 | 累积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† | rs1A | rd |  |  |
| LH† | rs1A | rd |  |  |
| LW† | rs1A | rd |  |  |
| LBU† | rs1A | rd |  |  |
| LHU† | rs1A | rd |  |  |
| SB | rs1A,rs2D |  |  |  |
| SH | rs1A,rs2D |  |  |  |
| SW | rs1A,rs2D |  |  |  |
| 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）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 源寄存器 | 目的寄存器 | 累积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** |

### RV32I基础整数指令集（续）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 源寄存器 | 目的寄存器 | 累积CSR |  |
| CSRRWI‡ | csr\* | rd,csr |  | \*除非rd＝**x0** |
| CSRRSI‡ | csr | rd,csr\* |  | \*除非uimm[4:0]＝0 |
| CSRRCI‡ | csr | rd,csr\* |  | \*除非uimm[4:0]＝0 |

### RV64I基础整数指令集

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 源寄存器 | 目的寄存器 | 累积CSR |  |
| LWU† | rs1A | rd |  |  |
| LD† | rs1A | rd |  |  |
| SD | rs1A,rs2D |  |  |  |
| 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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 源寄存器 | 目的寄存器 | 累积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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 源寄存器 | 目的寄存器 | 累积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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 源寄存器 | 目的寄存器 | 累积CSR |  |
| LR.W† | rs1A | rd |  |  |
| SC.W† | rs1A,rs2D | rd\* |  | \*如果成功 |
| AMOSWAP.W† | rs1A,rs2D | rd |  |  |
| AMOADD.W† | rs1A,rs2D | rd |  |  |
| AMOXOR.W† | rs1A,rs2D | rd |  |  |
| AMOAND.W† | rs1A,rs2D | rd |  |  |
| AMOOR.W† | rs1A,rs2D | rd |  |  |
| AMOMIN.W† | rs1A,rs2D | rd |  |  |
| AMOMAX.W† | rs1A,rs2D | rd |  |  |
| AMOMINU.W† | rs1A,rs2D | rd |  |  |
| AMOMAXU.W† | rs1A,rs2D | rd |  |  |

### RV64A标准扩展

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 源寄存器 | 目的寄存器 | 累积CSR |  |
| LR.D† | rs1A | rd |  |  |
| SC.D† | rs1A,rs2D | rd\* |  | \*如果成功 |
| AMOSWAP.D† | rs1A,rs2D | rd |  |  |
| AMOADD.D† | rs1A,rs2D | rd |  |  |
| AMOXOR.D† | rs1A,rs2D | rd |  |  |
| AMOAND.D† | rs1A,rs2D | rd |  |  |
| AMOOR.D† | rs1A,rs2D | rd |  |  |
| AMOMIN.D† | rs1A,rs2D | rd |  |  |
| AMOMAX.D† | rs1A,rs2D | rd |  |  |
| AMOMINU.D† | rs1A,rs2D | rd |  |  |
| AMOMAXU.D† | rs1A,rs2D | rd |  |  |

### RV32F标准扩展

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 源寄存器 | 目的寄存器 | 累积CSR |  |
| FLW† | rs1A | rd |  |  |
| FSW | rs1A,rs2D |  |  |  |
| 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,UF,NX | \*如果rm＝111 |
| FSUB.S | rs1,rs2,frm\* | rd | NV,OF,UF,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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 源寄存器 | 目的寄存器 | 累积CSR |  |
| FLD† | rs1A | rd |  |  |
| FSD | rs1A,rs2D |  |  |  |
| 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,UF,NX | \*如果rm＝111 |
| FSUB.D | rs1,rs2,frm\* | rd | NV,OF,UF,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 | NX | \*如果rm＝111 |
| FCVT.D.S | rs1,frm\* | rd | NX | \*如果rm＝111 |
| 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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 源寄存器 | 目的寄存器 | 累积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 |  |  |

# 第十五章 用于十进制浮点的“L”标准扩展（0.0版本）

这章是一个还没有被基金会批准的提案草案。

这章是为命名为“L”的标准扩展的规范占位的，该扩展是为支持IEEE 754-2008标准中定义的十进制浮点算数而设计的。

## 15.1 十进制浮点寄存器

现存的浮点计数器被用于持有64位和128位十进制浮点值，而现存的浮点加载和存储指令被用于把值移到或移出内存。

*由于融合的乘加指令需要大量的操作码空间，十进制浮点指令扩展将需要30位编码空间中的五个25位主操作码。*

# 第十六章 用于压缩指令的“C”标准扩展（2.0版本）

这章描述了当前对于RISC-V标准压缩指令集扩展的命名为“C”的提案，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。典型地，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。

## 16.1 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16.4中显示的那样，根据基础ISA宽度，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中，因此有必要在RV32C中包括这些。*

*尽管在不同基础寄存器宽度下，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寄存器宽度的设计，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。
* 编译器可以意识不到RVC扩展，而把代码压缩留给汇编器和链接器，即使一个能意识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[4]，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索引寄存器中的一个。稍后的IBM 360架构[3]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[18]，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[22]。

## 16.2 压缩指令格式

表16.1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16.2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。例如，立即数位17－10总是源自相同指令位位置。五个其它的立即数位（5、4、3、1和0）仅仅有两个源指令位，同时还有四个立即数位（9、7、6和2）有三个源，以及一个立即数位（8）有四个源。*

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 | 算数 | funct6 | | | | | | rd'/rs1' | | | funct2 | | rs2 | | | op | |
| CB | 分支 | funct3 | | | offset | | | rs1' | | | offset | | | | | op | |
| CJ | 跳转 | funct3 | | | jump target | | | | | | | | | | | op | |

表16.1：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 |

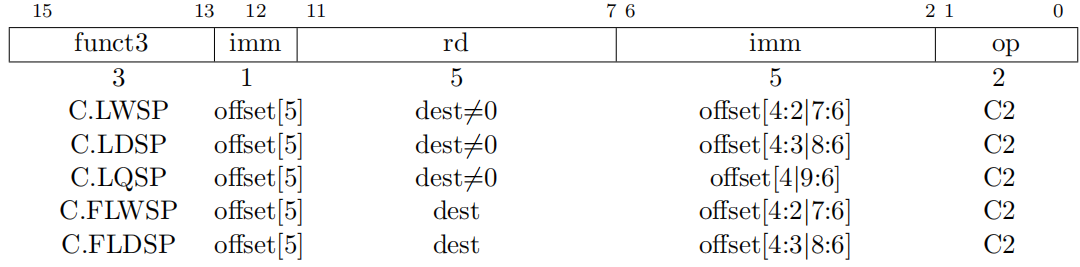
表16.2：通过CIW、CL、CS、CA和CB格式的三位的rs1’、rs2和rd’域指定的寄存器。

## 16.3 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



这些指令使用CI格式。

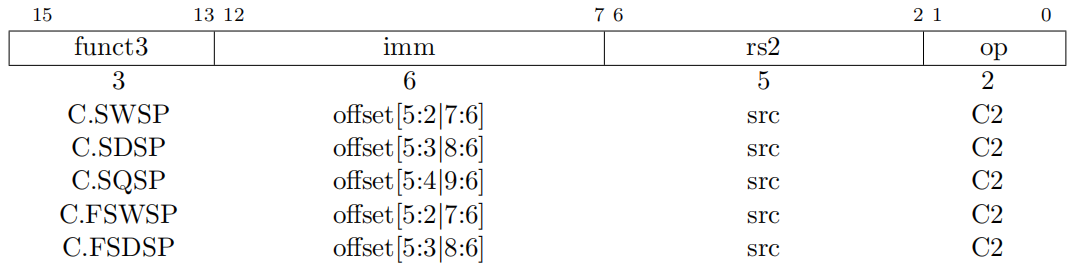
C.LWSP从内存把一个32位的值加载到寄存器rd中。它通过将零扩展的偏移量扩大4倍，加到栈指针x2上，来计算出有效地址。它扩展到lw rd, offset[7:2](x2)。C.LWSP只有在rd＝x0时有效；rd＝x0的代码点被保留。

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[8:3](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RV128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[9:4](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[7:2](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[8:2](x2)。



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[7:2](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[8:3](x2)。

C.SQSP是一个RV128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[9:4](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[7:2](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[8:3](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较大部分。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。*

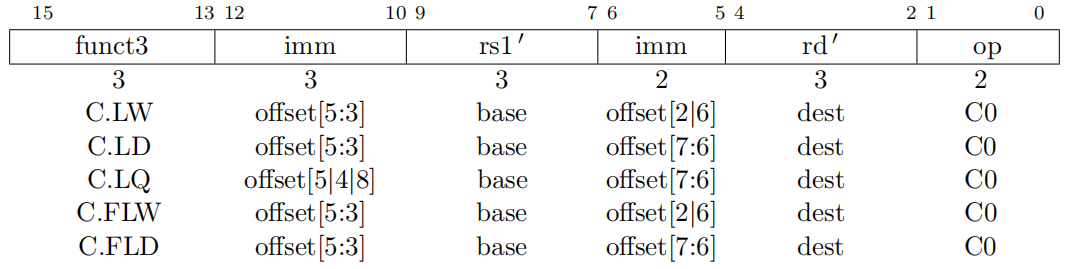
*为了进一步减少保存/恢复代码尺寸，在其它ISA中使用的一个常见的机制是多重加载和多重存储指令。我们考虑过为RISC-V采用这些指令，但是注意到这些指令的如下缺点：*

* *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。*

此外，在软件中，通过[23]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



这些指令使用CL格式。

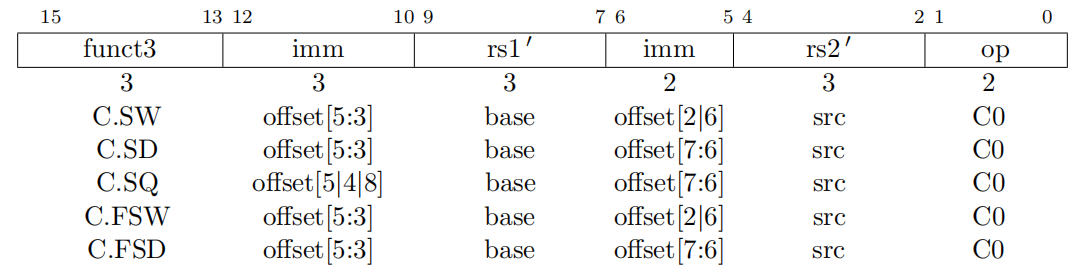
C.LW从内存加载一个32位的值到寄存器rd’中。它通过将符号扩展的偏移量扩大4倍，加到寄存器rs1’中的基地址上，来计算出有效地址。它扩展到lw rd’, offset[6:2](rs1’)。

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。它通过将符号扩展的偏移量扩大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[7:3](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[8:4](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[6:2](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[7:3](rs1’)。



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[6:2](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[7:3](rs1’)。

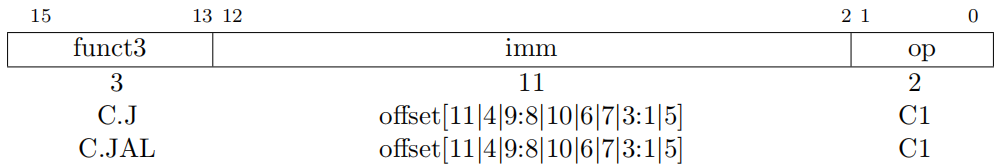
C.SQ是一个RV128C独有的指令，它把寄存器rs2’中的一个128位的值存储到内存。它通过将零扩展的偏移量扩大16倍，加到寄存器rs1’中的基地址上，来计算出有效地址。它扩展到sq rs2’, offset[8:4](rs1’)。

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[6:2](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[7:3](rs1’)。

## 16.4 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。



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±2KiB的目标范围。C.J扩展到jal x0, offset[11:1]。

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[11:1]。

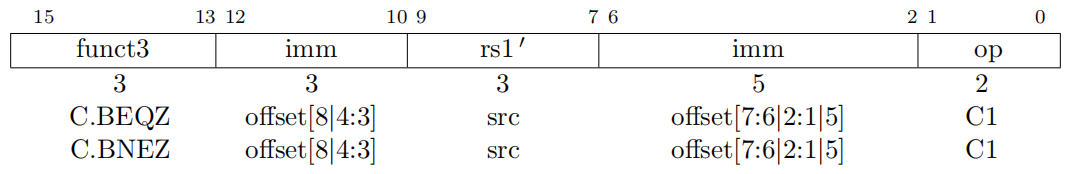


这些指令使用CR格式。

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。*



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±256B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[8:1]。

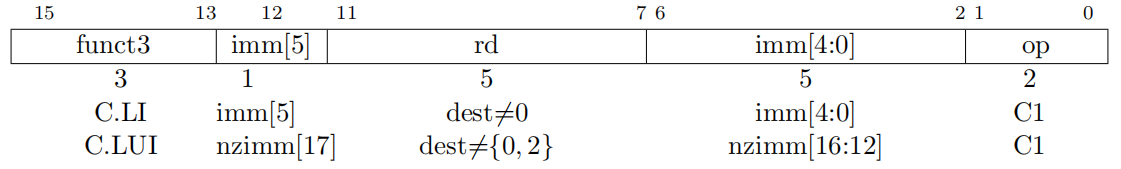
C.BNEZ的定义类似，但是它采取其分支是在rs1’包含一个非零的值时。它扩展到bne rs1’, x0, offset[8:1]。

## 16.5 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。

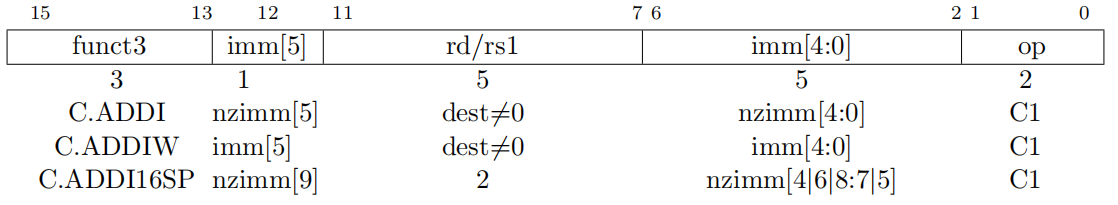


C.LI把符号扩展的6位立即数imm加载进寄存器rd中。C.LI扩展到addi rd, x0, imm[5:0]。C.LI只有当rd＝x0时有效；带有rd＝x0的代码点编码了HINT。

C.LUI把非零的6位立即数域加载进目的寄存器的位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的更高位。C.LUI扩展到lui rd, nzimm[17:12]。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。

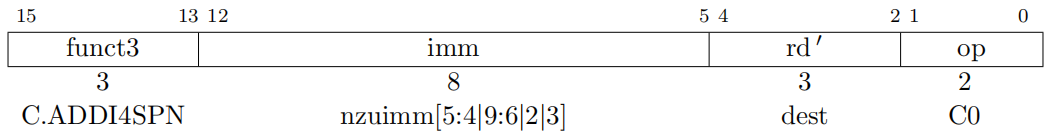


C.ADDI把非零的符号扩展的6位立即数加到寄存器rd中的值，然后把结果写到rd。C.ADDI扩展到addi rd, rd, nzimm[5:0]。C.ADDI只有当rd＝x0且nzimm＝0时有效。rd＝0的代码点编码了C.NOP指令；余下的nzimm＝0的代码点编码了HINT。

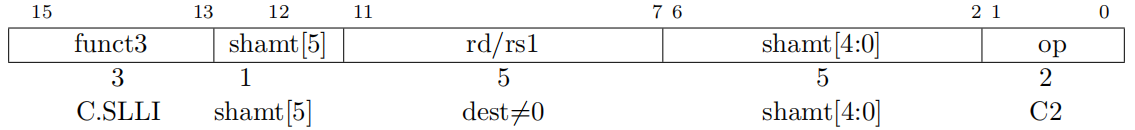
C.ADDIW是一个RV64C/RV128C独有的指令，它实施相同的计算，但是产生一个32位的结果，然后把结果符号扩展到64位。C.ADDIW扩展到addiw rd, rd, imm[5:0]。对于C.ADDIW，立即数可以是零，这对应于sext.w rd。C.ADDIW只有在rd＝x0时有效；rd＝x0的代码点被保留。

C.ADDI16SP与C.LUI共享操作码，但是有一个x2的目的域。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[9:4]。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。*

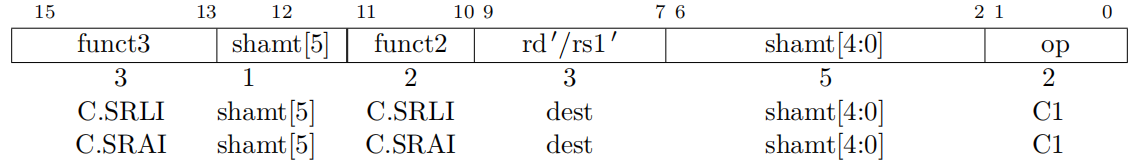


C.ADDI4SPN是一个CIW格式的指令，它把一个零扩展的非零立即数，扩大4倍，加到栈指针x2上，并把结果写到rd’。这个指令被用于生成指向栈分配变量的指针，且扩展到addi rd’, x2, nzuimm[9:2]。C.ADDI4SPN只在nzuimm＝0时生效；nzuimm＝0的代码点被保留。



SLLI是一个CI格式的指令，它对寄存器rd中的值实施逻辑左移，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[5:0]，除了shamt＝0的RV128C，它扩展到slli rd, rd, 64。

对于RV32C，shamt[5]必须是零；shamt[5]＝1的代码点被保留用于自定义扩展。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。

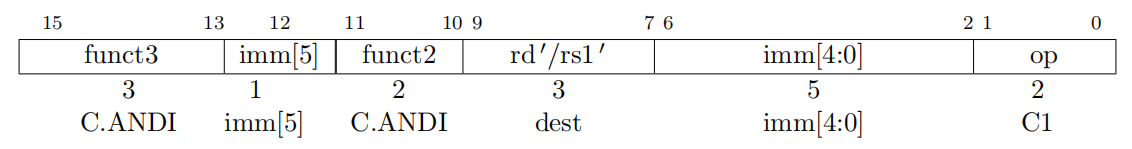


C.SRLI是一个CB格式的指令，它对寄存器rd’中的值实施逻辑右移，然后把结果写到rd’。移位的数目被编码在shamt域之中。对于RV128C，移位数目零被用于编码64的移位。甚至，对于RV128C，移位数目被符号扩展，并因此合法的移位数目是1 - 13，64，和96 - 127。C.SRLI扩展到srli rd’, rd’, shamt[5:0]，除了shamt＝0的RV128C，它扩展到srli rd’, rd’, 64。

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

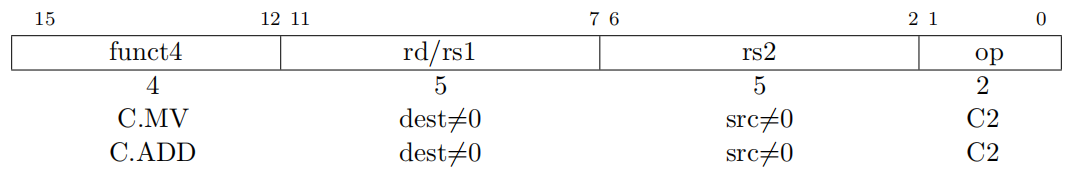
C.SRAI的定义与C.SRLI类似，但是执行的是算数右移。C.SRAI扩展到srai rd’, rd’, shamt[5:0]。

*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位地址空间代码的典型用法。*



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[5:0]。

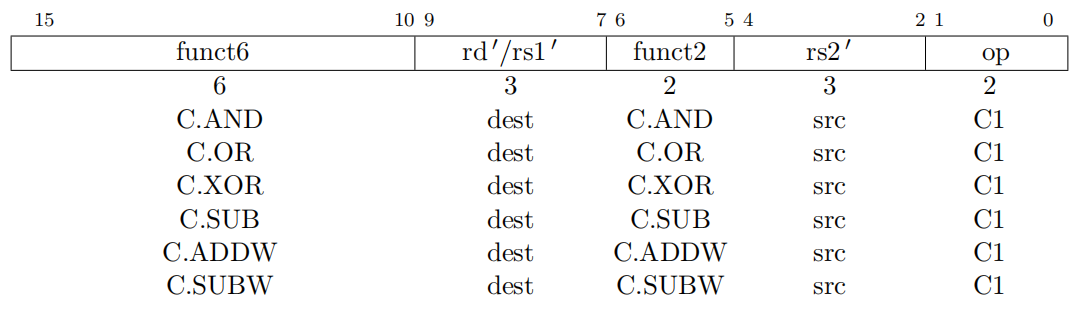
### 整数寄存器-寄存器操作

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；rs2＝x0的代码点对应于C.JALR和C.EBREAK指令。rs2＝x0和rd＝x0的代码点是HINT。



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’。

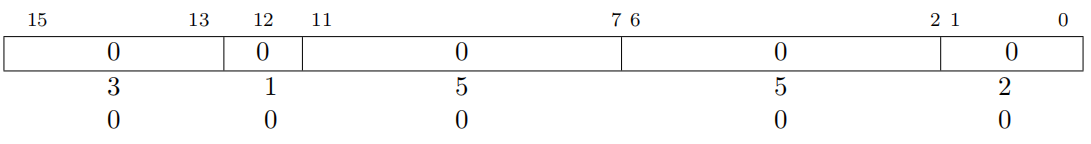
C.SUB从寄存器rd’中的值中减去寄存器rs2’中的值，然后把结果写到寄存器rd’。C.SUB扩展到sub rd’, rd’, rs2’。

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’。

*这组的六个指令各自不提供（对资源的）大量节省，但是不会占据太多的编码空间，并且易于实现，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。*

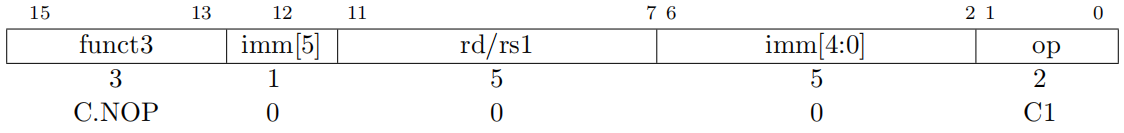
### 已定义的非法指令



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。

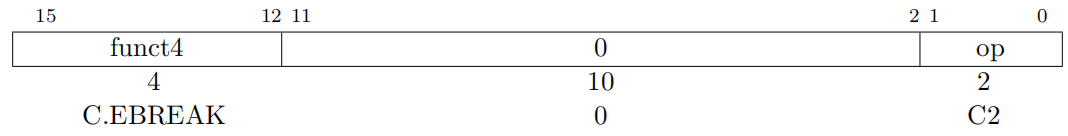
*我们把全零指令保留为非法指令，以帮助对尝试执行内存空间中以零结尾的或者不存在的部分进行陷入。在任何非标准的扩展中，全零的值都应当被重新定义。类似地，我们保留了所有位都设为1的指令（对应于RISC-V可变长度编码策略中的非常长的指令）作为非法指令，以捕获另一些常见的在不存在的内存区域中的值。*

### NOP指令



C.NOP是一个CI格式的指令，它不改变任何用户可见的状态，除了提升pc和增加任何适用的性能计数器。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。

## 16.6 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8.3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。*

## 16.7 “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, t0可能不会编码为与ADD x0, x0, t0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的映射允许稀缺的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16.3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，但是现在还没有被定义。余下的HINT空间被保留给自定义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** | 31 |
| C.SLLI | *rd*＝**x0**，*nzimm*≠0 | 31（RV32）  63（RV64/128） | 保留供自定义使用 |
| 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16.3：RVC HINT指令。

## 16.8 RVC指令集列表

表16.4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NSE来表示该操作码被保留用于自定义扩展；或者标记为HINT来表示该操作码被保留用于微架构提示（见16.7节）。

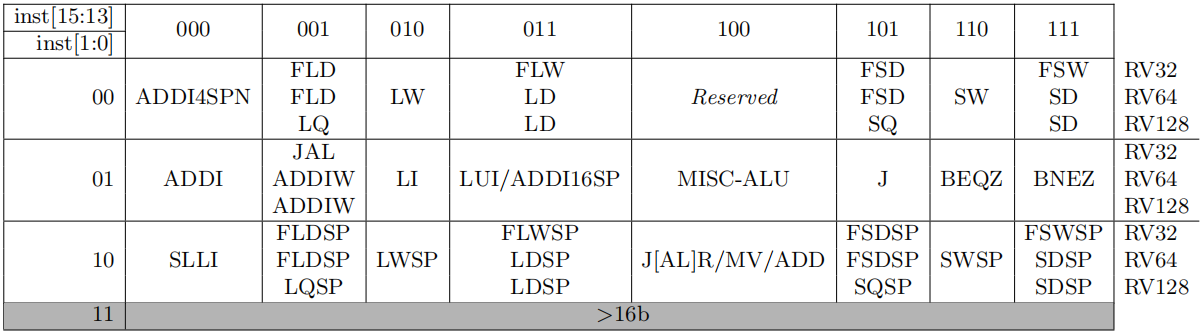


表16.4：RVC操作码映射

表16.5 - 16.7 列出了RVC指令。

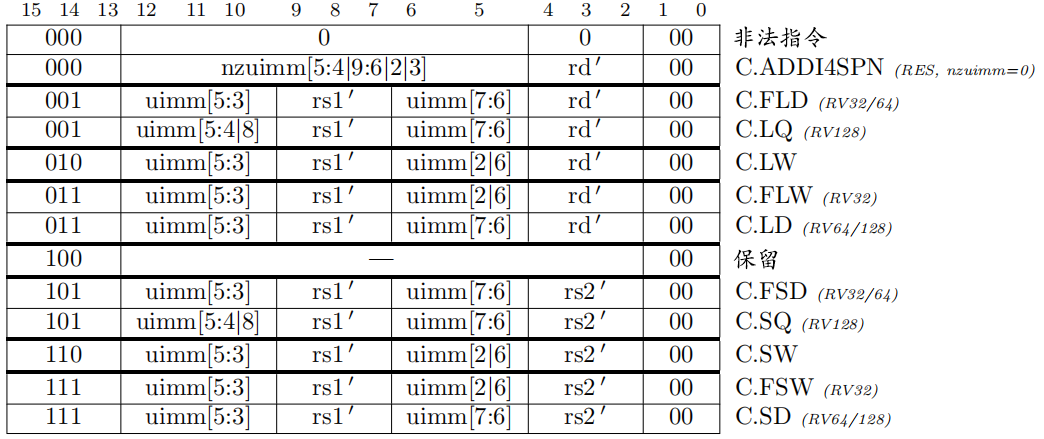


表16.5：RVC的指令列表，第0象限。

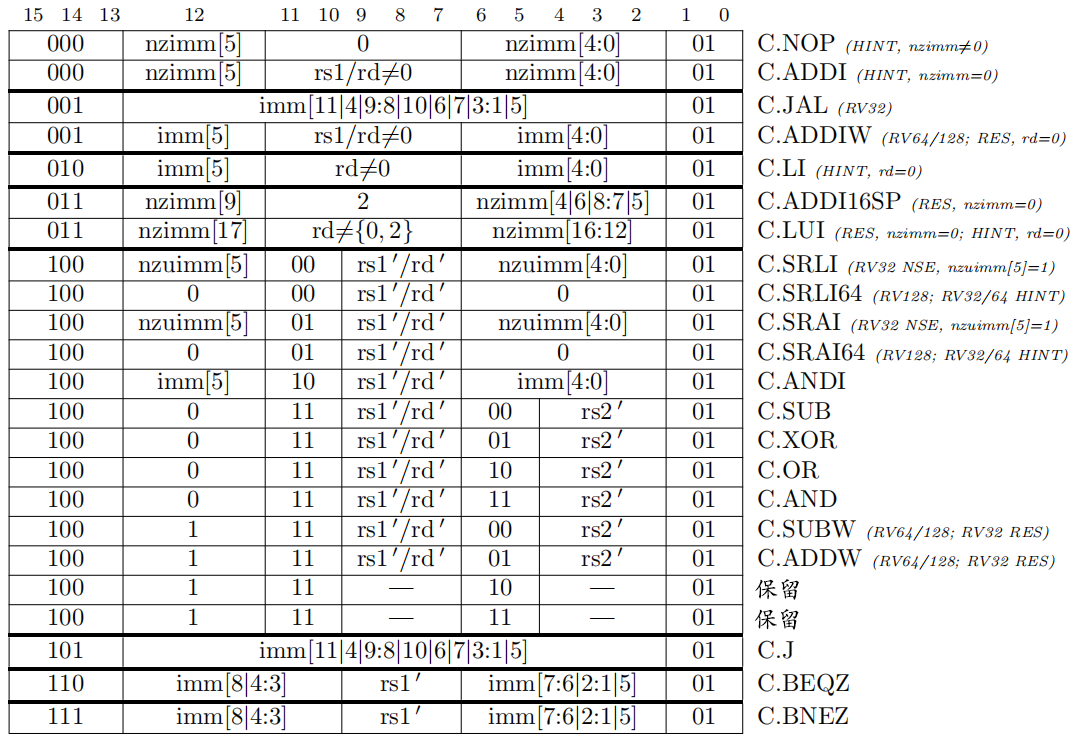


表16.6：RVC的指令列表，第1象限。

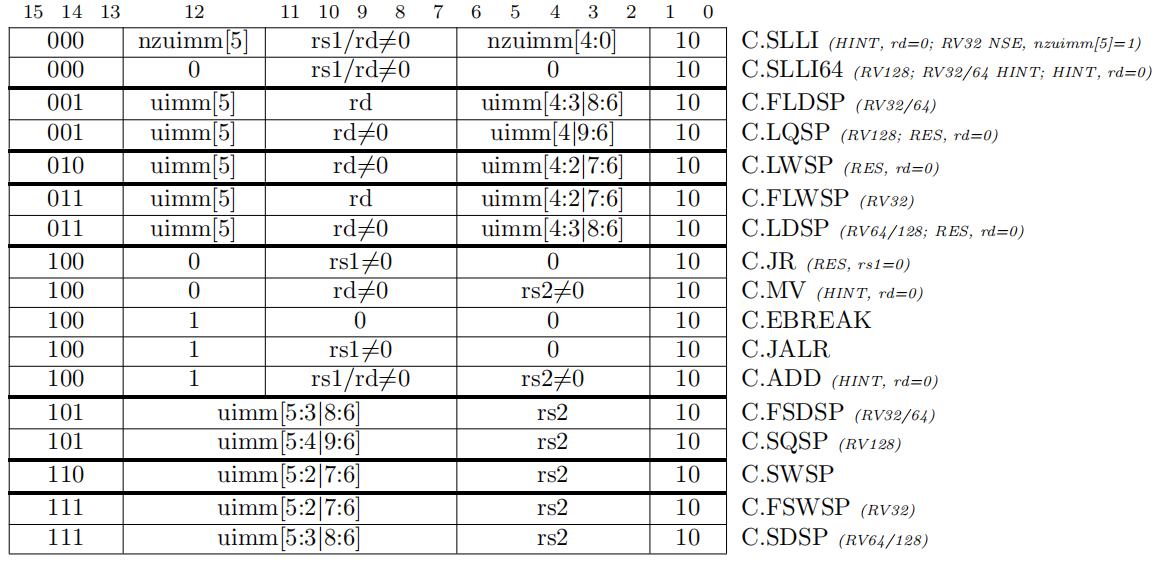


表16.7：RVC的指令列表，第2象限。

# 第十七章 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，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。*

# 第十八章 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。*

# 第十九章 用于事务内存的“T”标准扩展（0.0版本）

这章是为未来的提供事务内存操作的标准扩展占位的。

*尽管过去的20年进行了大量研究和初始的商业实现，关于支持涉及多个地址的的原子性操作的最佳方法仍然有许多争议。*

*我们当前的想法是，沿着原有的事务内存提案的思路，包括了一个小型有限容量的事务内存缓冲区。*

# 第二十章 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，对打包SIMD定点操作的使用还存在兴趣。一个任务组正在为了定义新的P扩展而工作着。*

# 第二十一章 用于向量操作的“V”标准扩展（0.7版本）

当前的工作草案被寄放在了https://github/riscv/riscv-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，而后续的向量扩展支持针对特定领域的更丰富的功能性。*

# 第二十二章 用于非对齐原子的“Zam”标准扩展（0.1版本）

这章定义了“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。

“Zam”下的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、在原生支持对相关的地址和尺寸的原子未对齐访问的硬件上（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、在没有原生支持对相关的地址和尺寸的未对齐访问的硬件上：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）。AMO可以通过把它们分割为独立的加载和存储指令而被仿真，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。*

# 第二十三章 用于全存储排序的“Ztso”标准扩展（0.1版本）

这章定义了用于RISC-V全存储排序（RVTSO）内存一致性模型的“Ztso扩展”。RVTSO被定义为RVWMO（定义在14.1章中）中有差异的部分。

*Ztso扩展旨在帮助原本为x86或SPARC架构写的代码进行移植，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。*

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。

# 第二十四章 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。

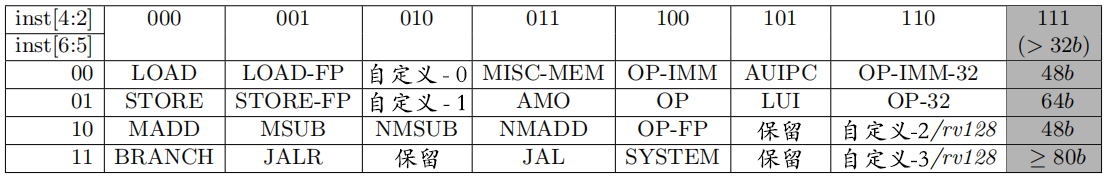


表24.1：RISC-V基础操作码映射，inst[1:0]＝11

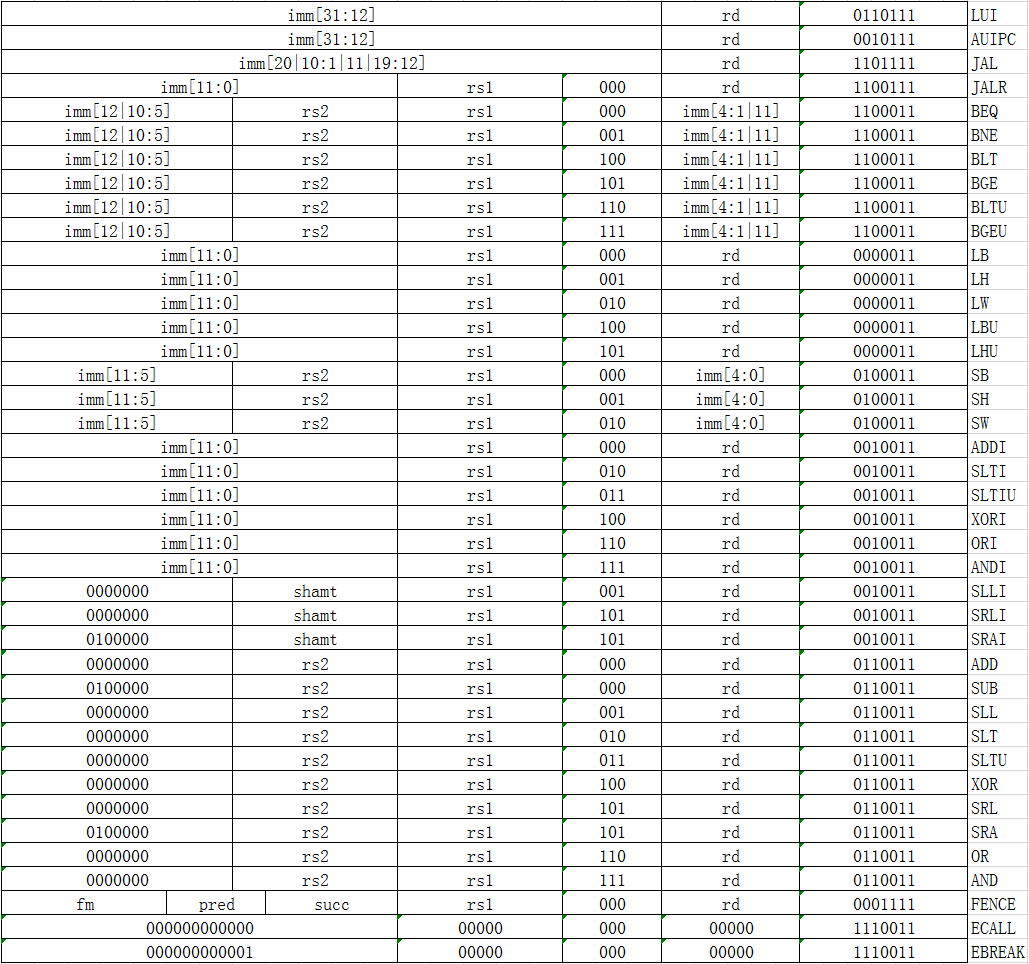
表24.1显示了RVG的主要操作码的映射。设置了3或更低位的主要操作码被保留用于超过32位的指令长度。标记为保留的的操作码应当被避免用于自定义的指令集扩展，因为它们可能会用在未来的标准扩展中。标记为自定义－0和自定义－1的主操作码应当被避免用于未来的标准扩展，并且建议用于具有基础32位指令格式的自定义指令集扩展。标记为自定义－2/rv128和自定义－3/rv128的操作码被保留，以供RV128未来使用，但如果不是RV128，那么它们将避免用于标准扩展，并因此也可以被用于RV32和RV64中的自定义指令集扩展。

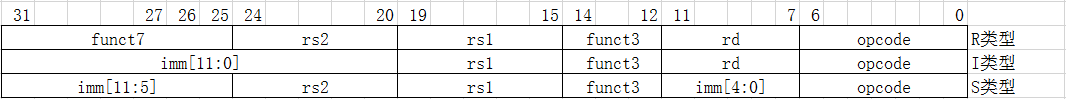
我们相信RV32G和RV64G为大量通用目的计算提供了简单但完整的指令集。可以添加可选的在16章中描述的压缩指令集（形成RV32GC和RV64GC）来改善性能、代码尺寸和能量效率，尽管这会带来额外的硬件复杂度。

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26章对于向RISC-V ISA添加扩展的方法进行了更加广泛的讨论。

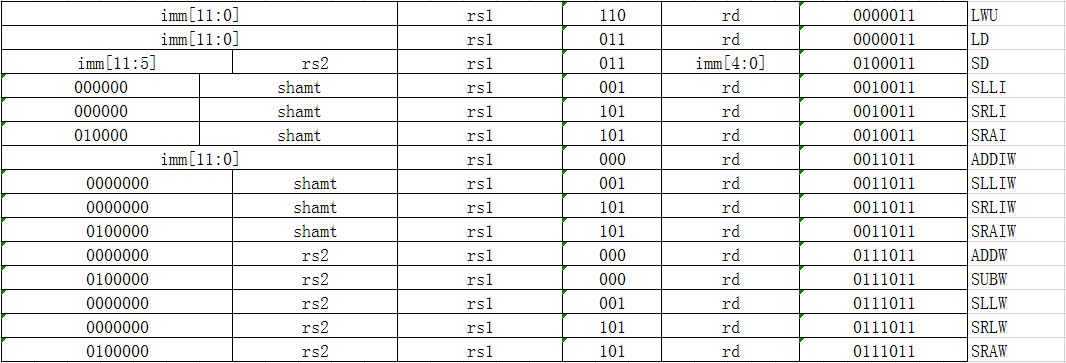


RV32I基础指令集





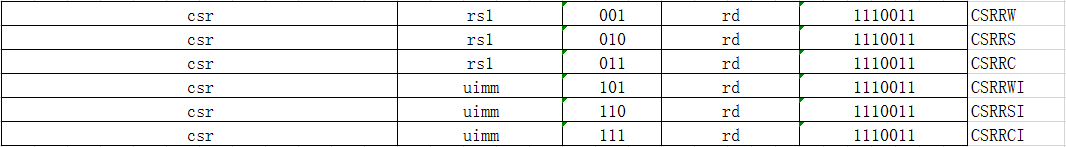
RV64基础指令集（RV32I之外的部分）



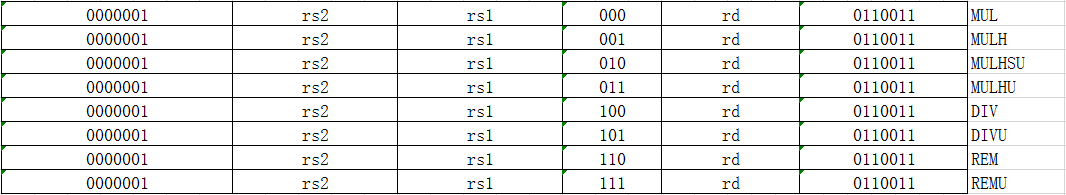
RV32/RV64 *Zifencei* 标准扩展

表24-5

RV32/RV64 *Zicsr* 标准扩展



RV32M标准扩展



RV64M标准扩展（RV32M之外的部分）

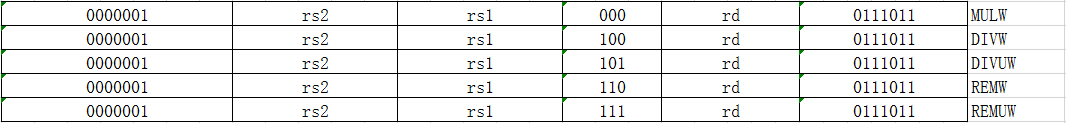
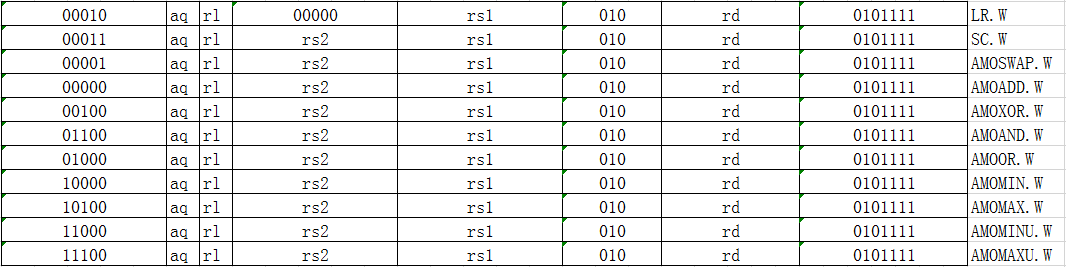
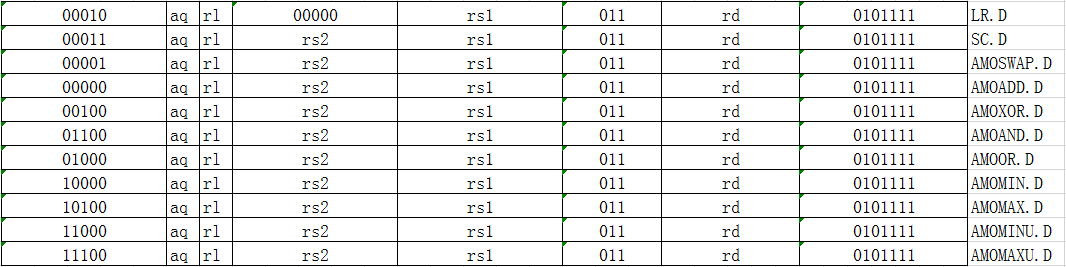


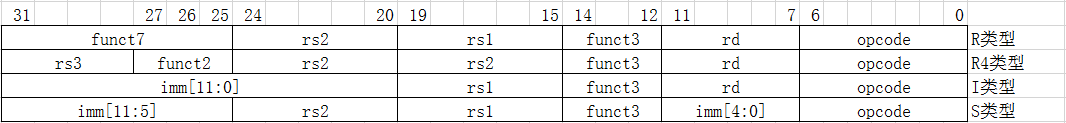
表24-2-9

RV32A标准扩展

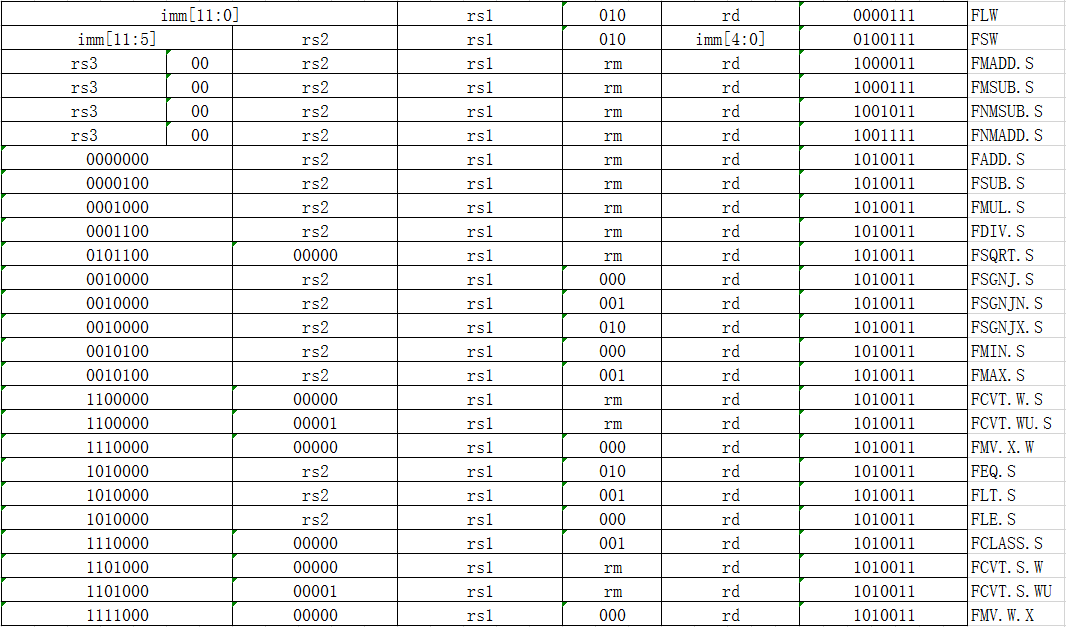


RV64A标准扩展（RV32A之外的部分）

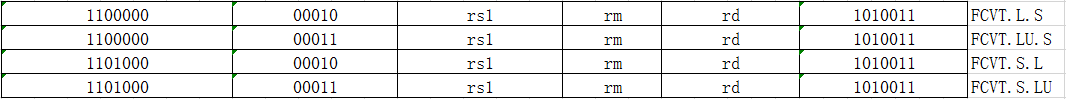


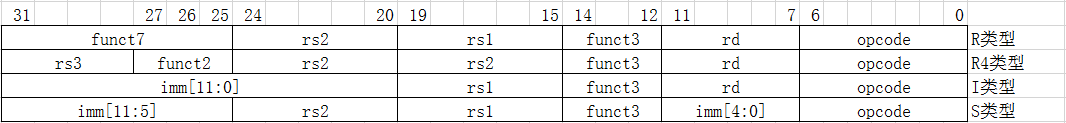


RV32F标准扩展

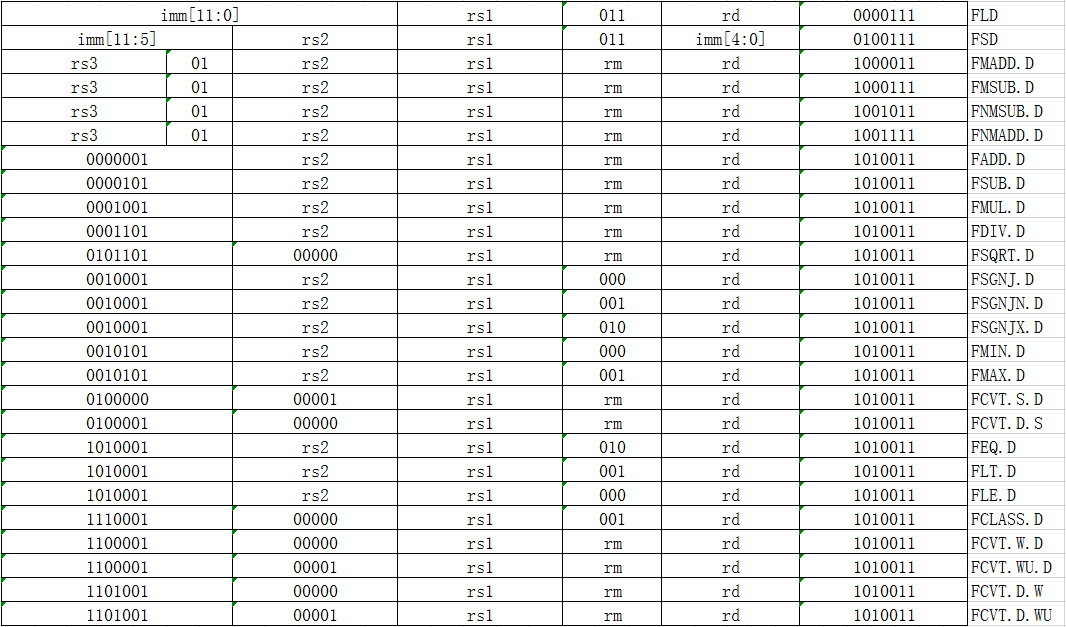


RV64F标准扩展（RV32F之外的部分）

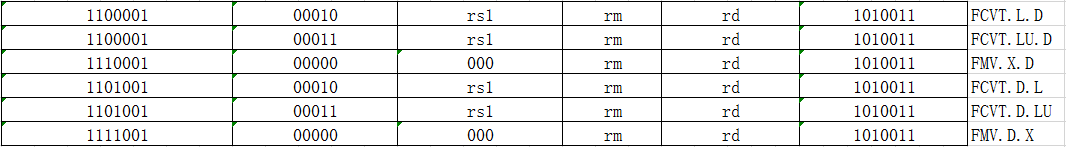


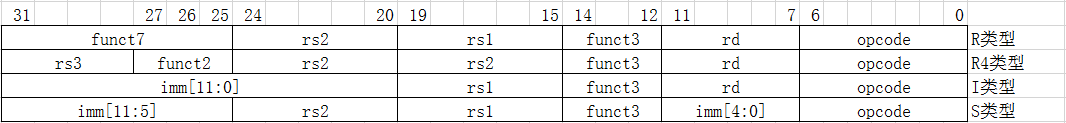


RV32D标准扩展

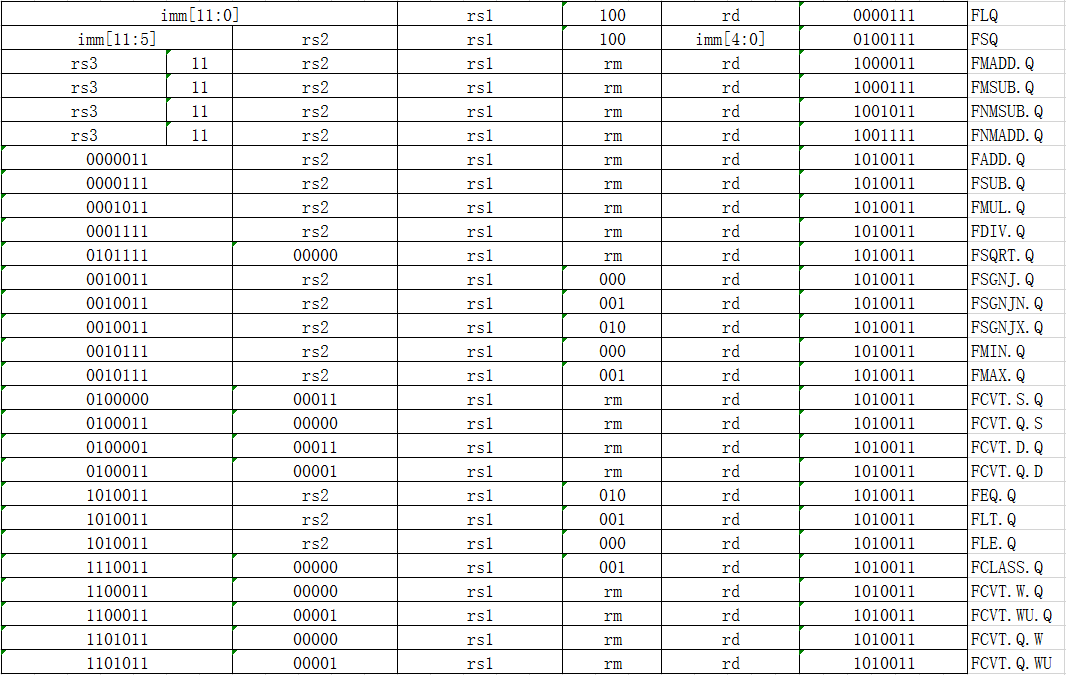


RV64D标准扩展（RV32D之外的部分）





RV32Q标准扩展



RV64Q标准扩展（RV32Q之外的部分）

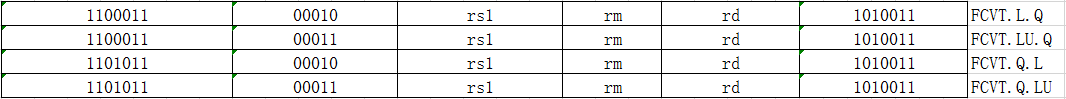


表24.2：RISC-V指令列表

表24.3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24.3：RISC-V控制和状态寄存器（CSR）地址映射。

# 第二十五章 RISC-V汇编编程手册

这章是为汇编编程手册占位的。

表25.1列出了关于x寄存器和f寄存器的汇编助记符，以及它们在首次标准调用约定中的角色。

|  |  |  |  |
| --- | --- | --- | --- |
| 寄存器 | ABI名称 | 描述 | 角色 |
| x0 | zero | 硬布线零 | —— |
| x1 | ra | 返回地址 | 调用者 |
| x2 | sp | 栈指针 | 被调用者 |
| x3 | gp | 全局指针 | —— |
| x4 | tp | 线程指针 | —— |
| x5 | t0 | 临时/备用链接寄存器 | 调用者 |
| x6－7 | t1－2 | 临时 | 调用者 |
| x8 | s0/fp | 保存寄存器/帧指针 | 被调用者 |
| x9 | s1 | 保存寄存器 | 被调用者 |
| x10－11 | a0－1 | 函数参数/返回值 | 调用者 |
| x12－17 | a2－7 | 函数参数 | 调用者 |
| x18－27 | s2－11 | 保存寄存器 | 被调用者 |
| x28－31 | t3－6 | 临时 | 调用者 |
| f0－7 | ft0－7 | FP临时 | 调用者 |
| f8－9 | fs0－1 | FP保存寄存器 | 被调用者 |
| f10－11 | fa0－1 | FP参数/返回值 | 调用者 |
| f12－17 | fa2－7 | FP参数 | 调用者 |
| f18－27 | fs2－11 | FP保存寄存器 | 被调用者 |
| f28－31 | ft8－11 | FP临时 | 调用者 |

表25.1：用于RISC-V整数和浮点寄存器的汇编助记符，和它们在首次标准调用约定中的角色。

*未来可能会有不同的调用约定，但是请注意，寄存器x1、x2和x5有编码在标准ISA和/或压缩扩展中的特殊含义。*

表25.2和25.3包含了标准RISC-V伪指令列表。

|  |  |  |
| --- | --- | --- |
| 伪指令 | 基础指令 | 含义 |
| la rd, symbol (non-PIC) | auipc rd, delta[31:12] + delta[11]  addi rd, rd, delta[11:0] | 加载绝对地址，其中delta=symbol-pc |
| la rd, symbol (PIC) | auipc rd, delta[31:12] + delta[11]  l{w|d} rd, rd, delta[11:0] | 加载绝对地址，其中delta=GOT[symbol]-pc |
| lla rd, symbol | auipc rd, delta[31:12] + delta[11]  addi rd, rd, delta[11:0] | 加载局部地址，其中delta=symbol-pc |
| l{b|h|w|d} rd, symbol | auipc rd, delta[31:12] + delta[11]  l{b|h|w|d} rd, delta[11:0](rd) | 加载global |
| s{b|h|w|d} rd, symbol, rt | auipc rt, delta[31:12] + delta[11]  s{b|h|w|d} rd, delta[11:0](rt) | 存储global |
| fl{w|d} rd, symbol, rt | auipc rt, delta[31:12] + delta[11]  fl{w|d} rd, delta[11:0](rt) | 浮点加载global |
| fs{w|d} rd, symbol, rt | auipc rt, delta[31:12] + delta[11]  fs{w|d} rd, delta[11:0](rt) | 浮点存储global |

基础指令使用pc相关的地址，所以链接器从符号中减去pc来得到差值delta。链接器把delta[11]加到高20位部分，抵消了低12位部分的符号扩展。

|  |  |  |
| --- | --- | --- |
| nop | addi x0, x0, 0 | 无操作 |
| li rd, immediate | 数不清的序列 | 加载立即数 |
| mv rd, rs | addi rd, rs, 0 | 拷贝寄存器 |
| not rd, rs | xori rd, rs, -1 | 一补数 |
| neg rd, rs | sub rd, x0, rs | 二补数 |
| negw rd, rs | subw rd, x0, rs | 二补数字 |
| sext.w rd, rs | addiw rd, rs, 0 | 符号扩展字 |
| seqz rd, rs | sltiu rd, rs, 1 | 如果为零则设置 |
| snez rd, rs | sltu rd, x0, rs | 如果不为零则设置 |
| sltz rd, rs | slt rd, rs, x0 | 如果小于零则设置 |
| sgtz rd, rs | slt rd, x0, rs | 如果大于零则设置 |
| fmv.s rd, rs | fsgnj.s rd, rs, rs | 拷贝单精度寄存器 |
| fabs.s rd, rs | fsgnjx.s rd, rs, rs | 单精度绝对值 |
| fneg.s rd, rs | fsgnjn.s rd, rs, rs | 单精度取反 |
| fmv.d rd, rs | fsgnj.d rd, rs, rs | 拷贝双精度寄存器 |
| fabs.d rd, rs | fsgnjx.d rd, rs, rs | 双精度绝对值 |
| fneg.d rd, rs | fsgnjn.d rd, rs, rs | 双精度取反 |
| beqz rs, offset | beq rs, x0, offset | 如果为零则分支 |
| bnez rs, offset | bne rs, x0, offset | 如果不为零则分支 |
| blez rs, offset | bge x0, rs, offset | 如果小于等于零则分支 |
| bgez rs, offset | bge rs, x0, offset | 如果大于等于零则分支 |
| bltz rs, offset | blt rs, x0, offset | 如果小于零则分支 |
| bgtz rs, offset | blt x0, rs, offset | 如果大于零则分支 |
| bgt rs, rt, offset | blt rt, rs, offset | 如果大于则分支 |
| ble rs, rt, offset | bge rt, rs, offset | 如果小于等于则分支 |
| bgtu rs, rt, offset | bltu rt, rs, offset | 如果无符号大于则分支 |
| bleu rs, rt, offset | bgeu rt, rs, offset | 如果无符号小于等于则分支 |

表25.2：RISC-V伪指令。

|  |  |  |
| --- | --- | --- |
| 伪指令 | 基础指令 | 含义 |
| j offset | jal x0, offset | 跳转 |
| jal offset | jal x1, offset | 跳转和链接 |
| jr rs | jalr x0, 0(rs) | 跳转寄存器 |
| jalr rs | jalr x1, 0(rs) | 跳转和链接寄存器 |
| ret | jalr x0, 0(x1) | 从子程序返回 |
| call offset | auipc x1,offset[31:12]+offset[11]  jalr x1, offset[11:0](x1) | 调用远程子程序 |
| tail offset | auipc x6,offset[31:12]+offset[11]  jalr x0, offset[11:0](x6) | 尾调用远程子程序 |
| fence | fence iorw, iorw | 阻隔所有的内存和I/O |
| rdinstret[h] rd | csrrs rd, instret[h], x0 | 读指令退场计数器 |
| rdcycle[h] rd | csrrs rd, cycle[h], x0 | 读周期计数器 |
| rdtime[h] rd | csrrs rd, time[h], x0 | 读实时时钟 |
| csrr rd, csr | csrrs rd, csr, x0 | 读CSR |
| csrw csr, rs | csrrw x0, csr, rs | 写CSR |
| csrs csr, rs | csrrs x0, csr, rs | 设置CSR中的位 |
| csrc csr, rs | csrrc x0, csr, rs | 清除CSR中的位 |
| csrwi csr, imm | csrrwi x0, csr, imm | 写CSR，立即数 |
| csrsi csr, imm | csrrsi x0, csr, imm | 设置CSR中的位，立即数 |
| csrci csr, imm | csrrci x0, csr, imm | 清除CSR中的位，立即数 |
| frcsr rd | csrrs rd, fcsr, x0 | 读FP控制/状态寄存器 |
| fscsr rd, rs | csrrw rd, fcsr, rs | 交换FP控制/状态控制器 |
| fscsr rs | csrrw x0, fcsr, rs | 写FP控制/状态寄存器 |
| frrm rd | csrrs rd, frm, x0 | 读FP舍入模式 |
| fsrm rd, rs | csrrw rd, frm, rs | 交换FP舍入模式 |
| fsrm rs | csrrw x0, frm, rs | 写FP舍入模式 |
| frflags rd | csrrs rd, fflags, x0 | 读FP异常标志 |
| fsflags rd, rs | csrrw rd, fflags, rs | 交换FP异常标志 |
| fsflags rs | csrrw x0, fflags, rs | 写FP异常标志 |

表25.3：RISC-V伪指令。

# 第二十六章 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管理员级别的扩展。

## 26.1 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或RV64I）。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”，是或者完整或者计划中的标准扩展。
* 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

指令编码空间是一些数目的指令位，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26.1为RISC-V给出了建议的尺寸。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 尺寸 | 用处 | # 可用的标准指令长度 | | | |
| 16位 | 32位 | 48位 | 64位 |
| 14位 | 压缩16位编码的象限 | 3 |  |  |  |
| 22位 | 基础32位编码中的次要操作码 |  | 28 | 220 | 235 |
| 25位 | 基础32位编码中的主要操作码 |  | 32 | 217 | 232 |
| 30位 | 基础32位编码的象限 |  | 1 | 212 | 227 |
| 32位 | 48位编码中的次要操作码 |  |  | 210 | 225 |
| 37位 | 48位编码中的主要操作码 |  |  | 32 | 220 |
| 40位 | 48位编码的象限 |  |  | 4 | 217 |
| 45位 | 64位编码中的子级次要操作码 |  |  |  | 212 |
| 48位 | 64位编码中的次要操作码 |  |  |  | 29 |
| 52位 | 64位编码中的主要操作码 |  |  |  | 32 |

表26.1：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，它适用于周围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26.2：标准指令集扩展的二维特征。

表26.2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管理员级别系统的改变，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不与所包含的标准扩展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。

## 26.2 RISC-V扩展设计理念

我们试图通过鼓励扩展开发者们再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。

另一个使用情况是，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。

## 26.3 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保留用于自定义扩展（表24.1），它们中的每个都代表一个25位编码空间。其中的两个（将是OP-IMM-64和OP-64）为RV128基础编码的最终使用而保留，但是可以被用于RV32和RV64的标准或非标准扩展。两个保留用于RV64的操作码（OP-IMM-32和OP-32）也可以被只用于RV32的标准和非标准扩展。

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24.1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。

## 26.4 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。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位指令应当像表1.1中描述的那样被编码。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。*

## 26.5 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，可以使用一些替代的方法，将VLIW编码作为扩展添加。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指令中的操作使用新的28位或42位编码，以及未对齐对更大指令组的指令获取。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

上述方法中，没有一个为VLIW指令中的单独的操作保留RISC-V编码。还有另一个方法是，在定宽32位编码中重新调整两个前缀位。如果设置了一个前缀位，它可以被用于指示“组结束”，而如果清除了第二个位，可以表示在一个谓词下的执行。通过意识不到VLIW扩展的工具生成的标准RISC-V 32位指令将把两个前缀位都进行设置（11），并因此具有正确的语义，其中每条指令都在组结束处，而不是谓词（在组结束处）。

这个方法的主要劣势是，基础ISA缺少复杂的谓词支持，这在一个激进的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。

# 第二十七章 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的指令的集合，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被设计为支持多种多样的实现，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的简化作用。

## 27.1 大小写敏感性

ISA命名字符串是大小写敏感的。

## 27.2 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。

## 27.3 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幕，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。

## 27.4 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”。

## 27.5 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，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。

## 27.6 附加的标准扩展的命名

标准扩展也可以使用一个“Z”、后面跟着一个字母顺序的名字和一个可选的版本号来命名。例如，“Zifencei”命名了第3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，IMAFDQLCBJTPVN。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，“RV32IMAC\_Zicsr\_Zifencei”。

## 27.7 管理员级指令集扩展

标准管理员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的名称和一个可选的版本号。管理员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。

标准管理员级扩展应当被列在标准非特权扩展之后。如果列出了多个管理员级别的扩展，它们应当按字母顺序排列。

## 27.8 超级管理员级指令集扩展

标准超级管理员级指令集扩展的命名和管理员级别的扩展相像，但是以字母“H”开始，而不是字母“S”。

标准超级管理员级扩展应当被列在标准更低特权级扩展之后。如果列出了多个超级管理员级别的扩展，它们应当按字母顺序排列。

## 27.9 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。

## 27.10 非标准扩展的命名

非标准扩展使用一个单独的“X”、后面跟着一个字母顺序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。

## 27.11 子集命名约定

表27.1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 | IMADZifencei |
| 四精度浮点 | Q | D |
| 十进制浮点 | L |  |
| 16位压缩指令 | C |  |
| 位操作 | B |  |
| 动态语言 | J |  |
| 事务性内存 | T |  |
| 打包的SIMD扩展 | P |  |
| 向量扩展 | V |  |
| 用户级中断 | N |  |
| 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 |  |
| 标准管理员级扩展 | | |
| 管理员级扩展“def” | Sdef |  |
| 标准超级管理员级扩展 | | |
| 超级管理员级扩展“ghi” | Hghi |  |
| 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27.1：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。

# 第二十八章 历史和鸣谢

## 28.1 “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感兴趣（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的需求（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）。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动，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[2]，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使用起诉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，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，需要实现管理员级别的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，包括主要的ISA扩展。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[12]。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。

## 28.2 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。管理员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、尤其是管理员模式，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，改良了管理员级的接口，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TA（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TA（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TA（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。

## 28.3 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28.1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28.1：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·沃特曼开发了两个核，并共同制造了：1)一个带有错误检测反转的RV64标量核，和2)一个带有附着64位浮点向量单元的RV64核。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，它是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1.25GHz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1.35GHz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[22]。

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TA（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TA，以及2014年春季，由埃里克·洛夫担任TA。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。

## 28.4 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。

## 28.5 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。

## 28.6 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级管理员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。

## 28.7 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反映出美国政府的立场和政策，并且不应被推断出官方认可。

# 附录A RVWMO的说明材料（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14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14张和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频谱。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，但是对于可以在管道核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。

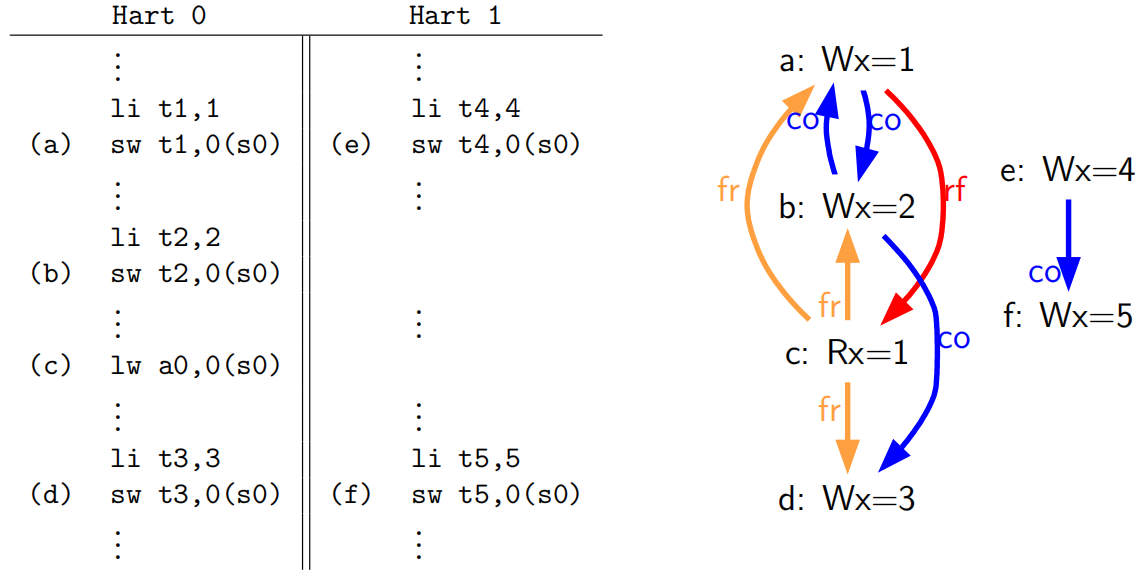
RISC-V选择了RVWMO内存模型，它是释放一致性的一个变体。这将它置于了内存模型频谱的两个极端之间。RVWMO内存模型使架构师能够构建简单的实现、激进的实现，将实现深深地嵌入到一个更大的系统之中并服务于复杂的内存系统交互，或者任何其它的可能性，所有这些的同时又足够强大，能够以高性能支持编程语言内存模型。

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 决定性检验

这章的解释使用了石蕊测试，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。表A.1显示了带有两个硬件线程的石蕊测试的一个例子。作为对这个表和对本章中之后所有表的约定，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。每张表在左侧显示了石蕊测试的代码，在右侧则是一个特定的有效或无效执行的可视化。



图A.1：一个示例石蕊测试和一个被禁止的执行（a0＝1）。

石蕊测试被用于理解特定具体情境中的内存模型的含义。例如，在表A.1的石蕊测试中，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在加载执行时将还没有成为可见的。我们下面来分析这个测试和许多其它的测试。

每个石蕊测试的右侧显示的图表显示了正在被考虑的特定的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讨论中可能产生执行的可能的全局内存次序的集合。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，在硬件线程、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也可以限制屏障、addr、数据，和一些控制边）。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。

|  |  |
| --- | --- |
| 边 | 全名（和解释） |
| rf | 读从（从各存储到返回该存储写入值的加载） |
| co | 一致性（关于存储到各地址的一个总的次序） |
| fr | 从读（从各加载到读取加载所返回值的存储的共同后继） |
| ppo | 保留的程序次序 |
| fence | 通过一个FENCE指令强行采取的排序 |
| addr | 地址依赖 |
| ctrl | 控制依赖 |
| data | 数据依赖 |

表A.1：在这个附录中绘制的石蕊测试图表的要点

例如，在表A.1中，a0＝1只可能发生在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)不是在(c)之前的最近匹配的存储。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。

*石蕊测试项目也提供了关于如何在RISC-V硬件上运行石蕊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，我们期望把这些内存模型也改编作为RISC-V抽检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 保留的程序次序和全局内存次序**

保留的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。

换句话说，来自相同硬件线程的没有按保留程序次序排序的事件，从其它硬件线程和/或观察者的角度，可以以新的次序出现。

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，不是当它在管道内部被执行时，而是只有当它的值已经被传播到全局可见的内存时，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年龄。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，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：

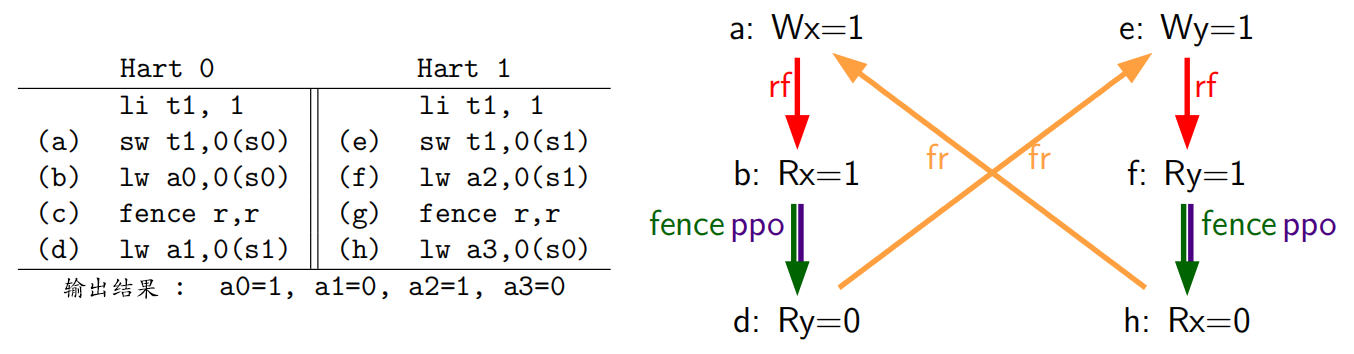
1、写该位，并且在全局内存次序中先于i的存储

2、写该位，并且在程序次序中先于i的存储

保留的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的石蕊测试。当在一个带有存储缓冲区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
* (b)执行并从存储缓冲区中的(a)转发它的返回值1
* (c)从所有之前的加载（例如，(b)）都已经完成时执行



图A.2：一个存储缓冲区转发石蕊测试（允许的输出结果）

* (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（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

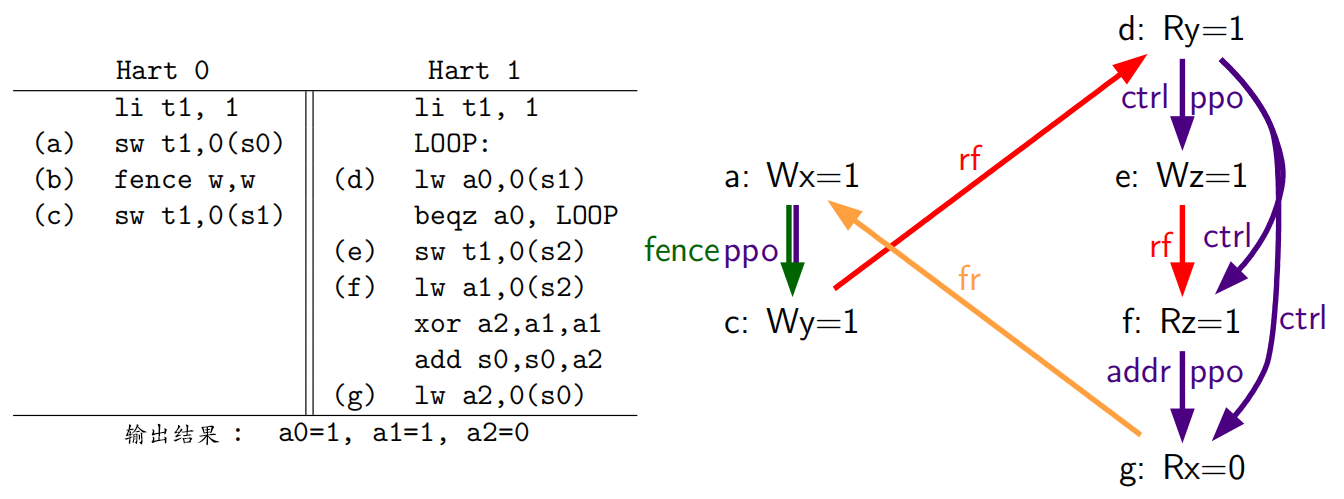
因此，内存模型必须能够解释这种行为。

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。



图A.3：“PPOCA”存储缓冲区转发石蕊测试（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：

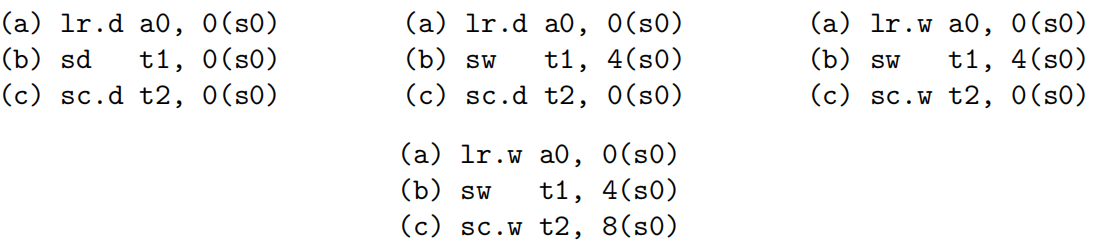
* (e)推测地执行，并进入第二个硬件线程的私有存储缓冲区（但是没有排放到内存）
* (f)推测地执行，并从存储缓冲区中的(e)转发它的值1。
* (g)推测地执行，并从内存读取值0。
* (a)执行，进入第一个硬件线程的私有存储缓冲区，并排放到内存
* (b)执行，并退场
* (c)执行，进入第一个硬件线程的私有存储缓冲区，并排放到内存
* (d)执行，并从内存读取值1
* (e), (f)和(g)提交，因为推测是正确的
* (e)从存储缓冲区排放到内存

### **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是一个到字节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。这些在概念上有不同的表现，以下列方式。LR/SC的行为就像是，旧值被带到核，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。



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-条件指令是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（假地）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，与期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 进程公理**

进程公理：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，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）**

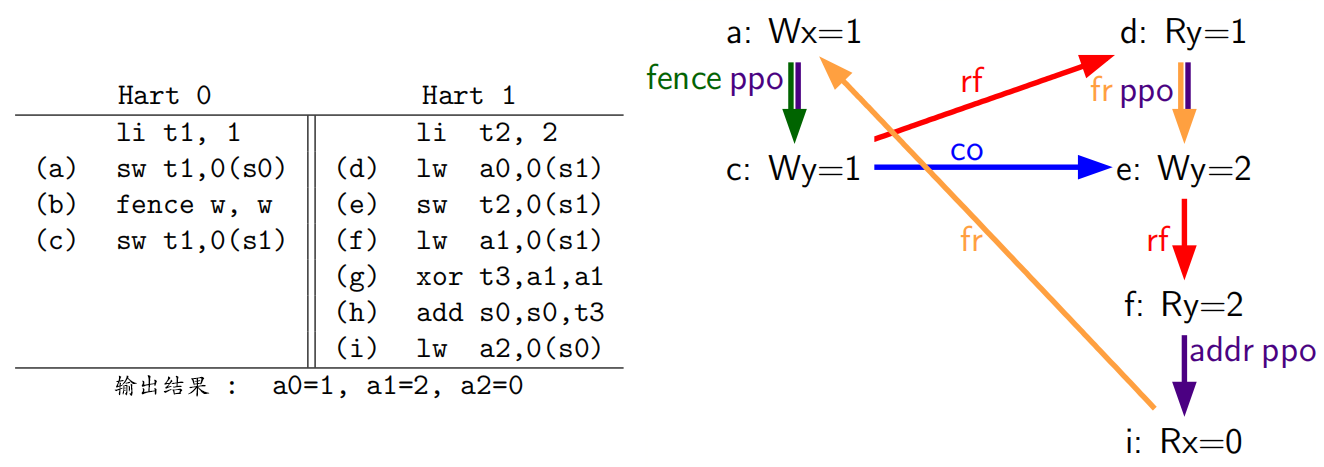
规则1：b是一个存储，且a和b访问重叠的内存地址

规则2：a和b是加载，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，而a和b返回由不同内存操作写入的x的值。

规则3：a是由一个AMO或SC指令生成的，b是一个加载，且b返回由a写入的一个值。

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与后面的存储被重新排序到一个重叠的内存位置。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。



表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。

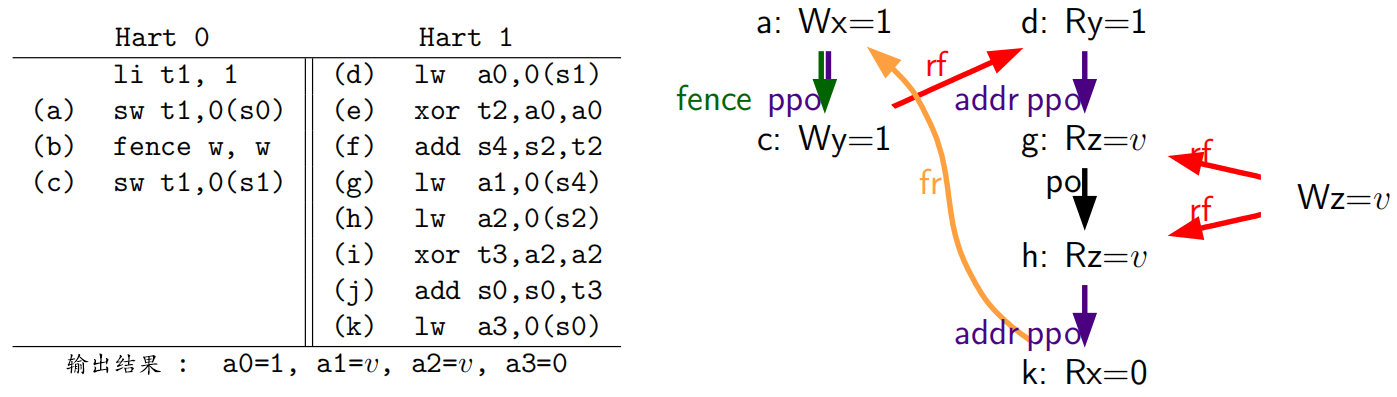
从微架构的视角，输出结果a0＝1，a1＝2，a2＝0是合法的（比起各种其它更加不怎么微妙的输出结果）。直观地讲，下列将产生上述提及的输出结果：

* (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。

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：石蕊测试RSW（允许的输出结果）

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）**

规则4：存在一个将a排序在b之前的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是与b配对的

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

# ...

# 临界区

# ...

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。

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

# ...

# 临界区

# ...

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

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：它也需要诚信开始处的“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采用的次序会稍微更粗糙些。

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。这个确保了，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中的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，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。

14.1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，但是浙江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)

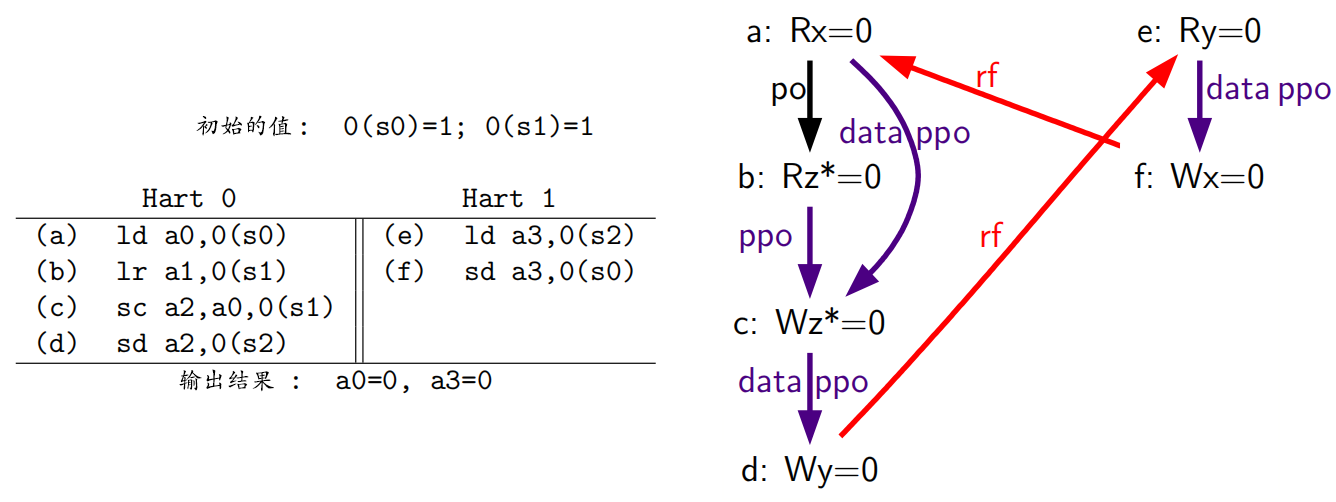
bne x1, x0, next

next: sw x3, 0(x4)

表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。



图A.13：LB石蕊测试的一种变体（禁止的输出结果）

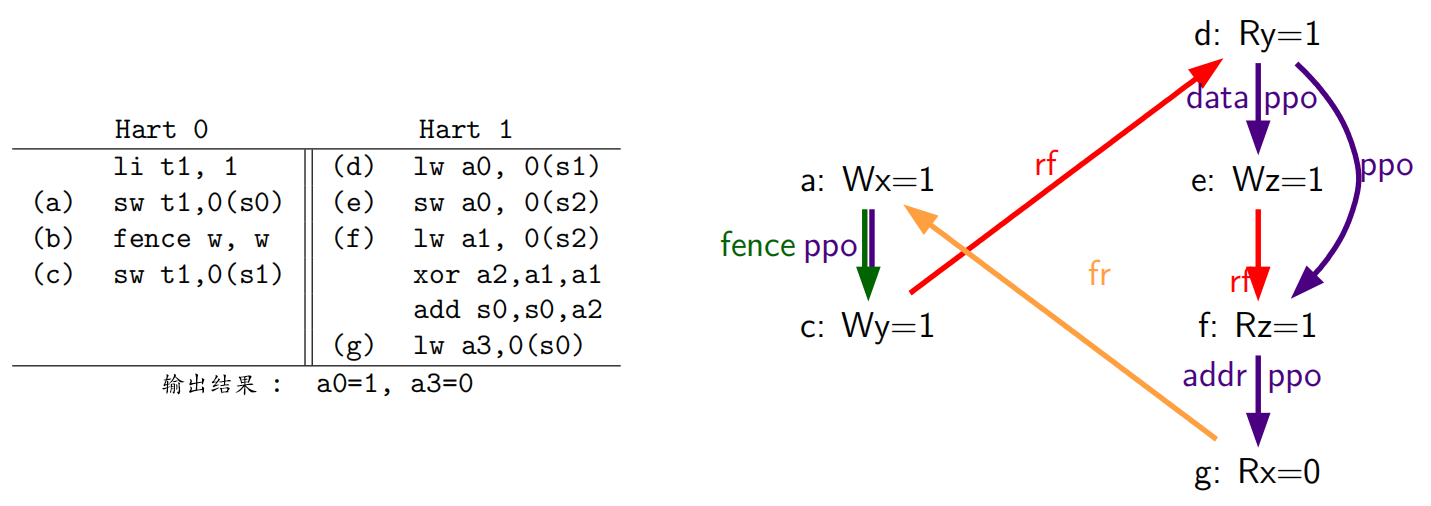
此外，选择尊重源自于存储-条件指令的依赖确保了特定的类似无中生有的行为会被阻止。考虑图A.13。假设一个假想的实现可以偶然地做到提前保证存储-条件操作将会成功。在这种情形中，(c)将提前返回0到a2（在实际执行之前），从而允许序列(d)、(e)、(f)、(a)、然后是(b)的执行，接着(c)可能只在那一点（成功地）执行。这将表示(c)把它自己的成功的值写到了0（s1）！幸运的是，由于RVWMO尊重源自于由成功的SC指令生成的存储的依赖的事实，这个情形和其它类似的情形被阻止了。

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，14.3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 管道依赖（规则12-13）**

规则12：b是一个加载，并且按照程序次序，在a和b之间存在一些存储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，并且按照程序次序，在a和b之间存在一些指令m，使得m有一个关于a的地址依赖。



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）

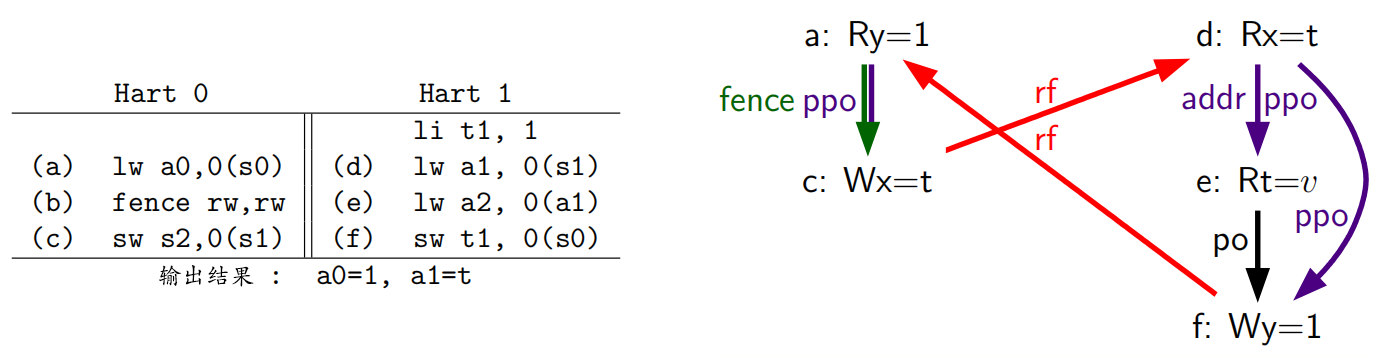
PPO规则12和13反应了几乎所有的真实的处理器管道实现的行为。规则12陈述了一个加载不能从一个存储转发，直到那个存储的地址和数据是已知的时候。考虑图A.14：在(e)的数据被决定之前，(f)不能执行，因为(f)必须返回由(e)写入的值（或者在全局内存次序中由某些更加靠后的所写入的值），而在(d)有机会执行之前，旧的值必须不能被(e)的写回所冲击。因此，(f)将永远不会在(d)的执行之前执行。

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。

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，直到所有的先前的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。



图A.15：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）



图A.16：因为从(d)到(e)的地址依赖，(d)也先于(f)了（禁止的输出结果）

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。

## 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2.7节中，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14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。

这一节中的次序保证可能不适用于在RISC-V核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  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加载必须被映射到之后跟有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。

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：

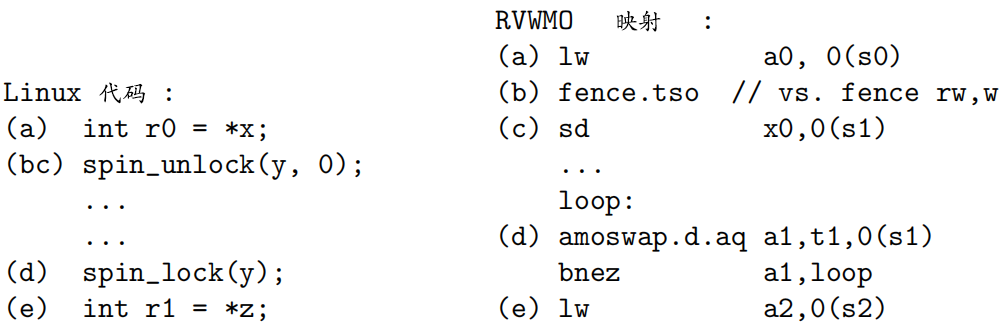
1. 永远使用FENCE RW, W/FENCE R, RW，并且永远不使用aq/rl。这是足够的，但是不可取，因为它违背了aq/rl修饰符的目的。

|  |  |
| --- | --- |
| 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。

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。



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。

|  |  |
| --- | --- |
| 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}; 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。

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<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。

任何AMO可以通过一个LR/SC对来仿真，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。仿真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。

|  |  |
| --- | --- |
| 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。

## 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，我们希望以下内容：

* 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。

包含这些规则是为了方便在B.3节中描述的常见微架构优化（规则2）和操作形式化模型方法（规则3和12－13）。它们也有助于从拥有相似规则的其它架构移植代码的进程。

从硬件的角度，我们也希望下列内容：

* 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节中描述的那样仿真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

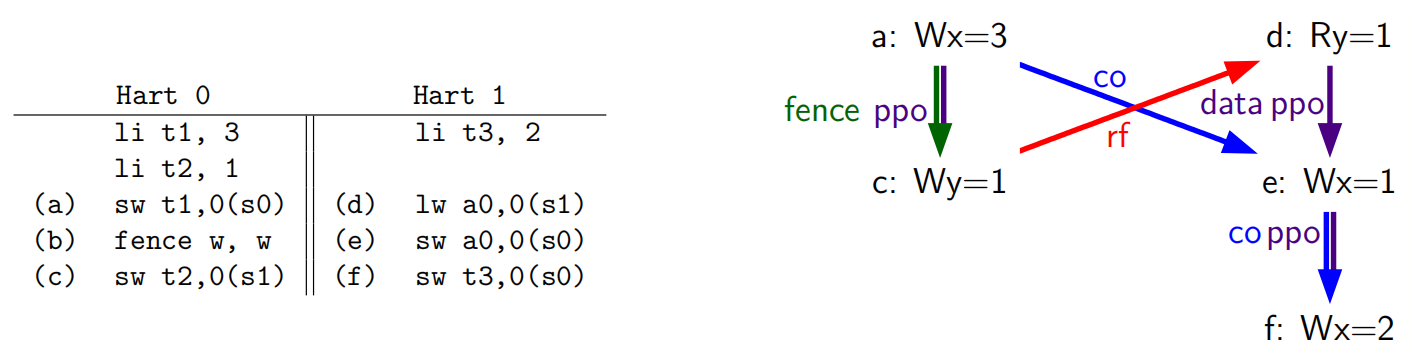
其它的一般注意事项：

* 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：

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)



图A.19：写归并石蕊测试，允许的执行。

* 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)

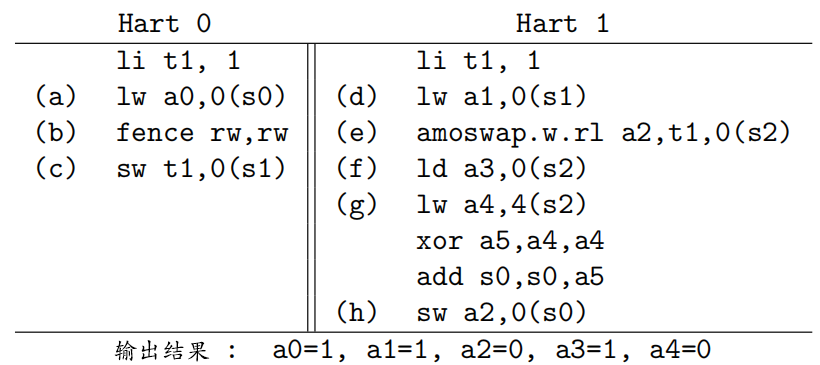
换句话说，地址在s0中的内存位置的最终取值必定是2（由存储(f)写入的值），而不能是3（由存储(a)写入的值）。

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。

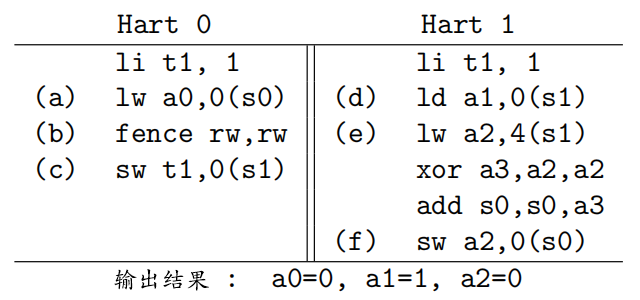
### A.6.1 未来可能的扩展

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：

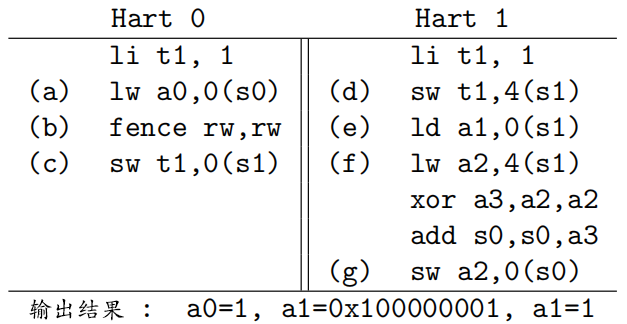
* ‘V’向量ISA扩展
* ‘T’ISA扩展的一个事务内存子集
* ‘J’JIT扩展
* 用于设置了aq和rl的加载和存储操作码的原生编码
* 限制到特定地址的屏障
* 缓存的写回/冲刷/无效化/等等……指令



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）



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）



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）

## A.7 已知问题

### A.7.1 混合尺寸的RSW

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，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14章和ISA规范主体的剩余部分中。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/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 | // starting from the lr ,

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 ) }

图B.3：Alloy中的形式化RVWMO内存模型（3/5：内存的模型）

// //////////////////////////////////////////////////////////////////////////////

// = 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 ] {

all disj e , e ’: bag | e - >e ’ in rel + ~ rel

acyclic [ rel ]

}

图B.5：Alloy中的形式化RVWMO内存模型（5/5：辅助内容）

## 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14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 <http://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 ]

and r8 = rmw

(\* Syntactic Dependencies \*)

and r9 = [ M ]; addr ;[ M ]

and r10 = [ M ]; data ;[ W ]

and r11 = [ M ]; ctrl ;[ W ]

(\* Pipeline Dependencies \*)

and r12 = [ R ];( addr | data );[ W ]; rfi ;[ R ]

and r13 = [ R ]; addr ;[ M ]; po ;[ W ]

let ppo = r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9 | r10 | r11 | r12 | r13

图B.6：riscv-defs.cat，一个关于保留程序次序的Herd定义（1/3）

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（https://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https://github.com/rems-project/sail，以及RISC-V ISA模型，见https://github.com/rems-project/sail-riscv），而并发语义使用Lem表达（关于Lem语言，见https://github.com/rems-project/lem）。

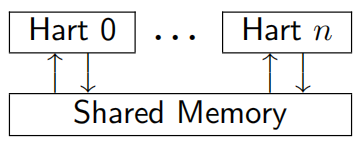
rmem有一个命令行接口和一个网络接口。网络接口完全运行在客户端，并且与一个石蕊测试库一起在线提供：http//www.cl.cam.ac.uk/~pes20/rmem。命令行接口比网络接口更快，尤其是在穷举模式中。

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。

〇 寄存器写：这是对一个寄存器值的写。

〇 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。

〇 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。

〇 结束指令：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：

〇 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。

* 通过从未传播的存储转发而满足内存加载操作：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。

〇 完成加载操作：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，直到结束指令过渡。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：

〇 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。

〇 初始化内存存储操作值：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。

〇 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：这把单个的内存存储操作传播到内存。

〇 完成存储操作：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：这是对提交存储指令和传播存储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。
* 晚期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：

* 一个AMO的满足、提交和传播操作：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：

〇 提交屏障

标记有〇标签的过渡，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Fetch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. 通过从未传播的存储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. 完成加载操作

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，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挂起的内存或寄存器状态，由伪代码所请求，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

这里：

* *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\_names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\_names（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。最初的收缩使用空集合，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*...*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14.1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*...*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, store\_continuation*))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*i.mem\_stores*之间分割出*mem\_value*；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*（见通过从未传播的存储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。

**晚期sc失败**

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（零次或多次）

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, read\_cont*))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, reg\_value, next\_state*))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*cfp*是*i*的内存足迹中的*cfp*的补。如果*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*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*cfp*重叠的内存足迹的加载，那么与*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 ELF psABI Specification. https://github.com/riscv/riscv-elf-psabi-doc/.

[2] IEEE standard for a 32-bit microprocessor. IEEE Std. 1754-1994, 1994.

[3] 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.

[4] Werner Buchholz, editor. Planning a computer system: Project Stretch. McGraw-Hill Book Company, 1962.

[5] 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.

[6] Timothy H. Heil and James E. Smith. Selective dual path execution. Technical report, University of Wisconsin - Madison, November 1996.

[7] ANSI/IEEE Std 754-2008, IEEE standard for floating-point arithmetic, 2008.

[8] Manolis G.H. Katevenis, Robert W. Sherburne, Jr., David A. Patterson, and Carlo H. S´equin. The RISC II micro-architecture. In Proceedings VLSI 83 Conference, August 1983.

[9] 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.

[10] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for nonpredicated instruction set architectures. In Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, PACT ’98, Washington, DC, USA, 1998.

[11] 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.

[12] OpenCores. OpenRISC 1000 architecture manual, architecture version 1.0, December 2012.

[13] Heidi Pan, Benjamin Hindman, and Krste Asanovi´c. 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.

[14] Heidi Pan, Benjamin Hindman, and Krste Asanovi´c. Composing parallel software efficiently with Lithe. In 31st Conference on Programming Language Design and Implementation, Toronto, Canada, June 2010.

[15] David A. Patterson and Carlo H. S´equin. RISC I: A reduced instruction set VLSI computer. In ISCA, pages 443–458, 1981.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] Andrew Waterman. Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed. Master’s thesis, University of California, Berkeley, 2011.

[23] Andrew Waterman. Design of the RISC-V Instruction Set Architecture. PhD thesis, University of California, Berkeley, 2016.

[24] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovi´c. 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.

[25] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovi´c. 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.