RISC-V 指令集手册

第 I 卷：非特权指令集架构

文档版本 20191214-*draft*

编者：安德鲁 · 沃特曼 [1], 克尔斯泰 · 阿桑诺维奇 [1,2]
[1]SiFive 股份有限公司，
[2] 加州伯克利分校，电子工程部，计算机科学与技术系
waterman@eecs.berkeley.edu, krste@berkeley.edu

2022 年 10 月 11 日

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

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

前言

本文描述了 RISC-V 非特权架构。

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

本文包含以下版本的 RISC-V ISA 模块：

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

对基于已批准的 20191213 版本文档的前言

本文档描述了 RISC-V 非特权架构。

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

本文包含以下版本的 RISC-V ISA 模块：

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

此版本文档中的变动包括：

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

- 定义了暂停提示指令（PAUSE hint instruction）。

对基于已批准的 20190608 版本文档的前言

本文档描述了 RISC-V 非特权指令集架构。

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

本文包含以下版本的 RISC-V ISA 模块：

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

此版本文档中的变化包括：

- 将在 2019 年初被理事会批准的 ISA 模块的描述，更正为“**被批准**”的。
- 从批准的模块中移除 A 扩展。
- 变更文档版本方案，以避免与 ISA 模块的版本冲突。
- 把基础整数 ISA 的版本号增加到 2.1，以反映：被批准的 RVWMO 内存模型的出现，和先前基础 ISA 中的 FENCE.I、计数器和 CSR 指令的去除。
- 把 F 扩展和 D 扩展的版本号增加到 2.2，以反映：版本 2.1 更改了规范的 NaN （译者注：NaN 是 Not a Number 的缩写，是计算机科学中数值数据类型的一类值，表示未定义或不可表示的值，常在浮点数运算中使用，参见 IEEE 754-1985 浮点数标准）；而版本 2.2 定义了 NaN 装箱（NaN-Boxing，意为 NaN 的表示方式）方案，并更改了 FMIN 和 FMAX 指令的定义。
- 将文档的名字改为“非特权的”指令，以此作为将 ISA 规范从平台相关的文件中分离的行动之一。
- 为执行环境、硬件线程（译者注：硬件线程 hart 是 RISC-V 引入的一个新概念，具体含义请参考 RISC-V 规范的第 II 卷“特权 ISA”）、陷入（traps）和内存访问添加了更清晰和更精确的定义。
- 定义了指令集的种类：*标准的*, *保留的*, *自定义的*, *非标准的*, and *非符合的*。
- 移除了隐含着交替字节序操作的相关文本，因为交替字节序操作还没有被 RISC-V 所定义。
- 修改了对未对齐的 load 和 store 行为的描述。规范现在允许执行环境接口中对未对齐地址陷入进行可见的处理，而不是仅仅在用户模式中授权对未对齐的加载和存储进行不可见处理。而且，现在允许报告有关未对齐访问（包括原子访问）的访问异常，而不是仅仅模拟。
- 把 FENCE.I 从强制性的基础模块中移出，编入一个独立的扩展，名为 Zifencei ISA。FENCE.I 曾经被从 Linux 用户 ABI 中去除，它在实现大型非一致性指令和数据缓存时是有问题的。然而，它仍然是仅有的标准的取指一致性机制。
- 去除了禁止 RV32E 和其它扩展一起使用的约束。
- 去除了平台相关的约束条款，即，在 RV32E 和 RV64I 章节中，特定编码会产生非法指令异常
- 计数器/计时器指令现在不被认为是强制性的基础 ISA 的一部分，因此 CSR 指令被移动到独立的章节并被标记为 2.0 版本，同时非特权计数器被移动到另一个独立的章节。计数器由于存在明显的问题（包括计数不精确等），所以还没有准备批准。
- 添加了 CSR 有序访问模型。
- 为 2 位 *fmt* 域中的浮点指令明确地定义了 16 位半精度浮点格式。
- 定义了 FMIN.*fmt* 和 FMAX.*fmt* 的有符号零行为，并改变了它们遇到 NaN 信号（Signaling-NaN）输入时的行为，以符合建议的 IEEE 754-201x 规范中的 minimumNumber 和 maximumNumber 操作规范。
- 定义了内存一致性模型 RVWMO。

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

对 2.2 版本文档的前言

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

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

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

此版本文档的主要变更包括：

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

对 2.1 版本文档的前言

这是文档的 2.1 版本，描述了 RISC-V 用户级架构。注意被冻结的 2.0 版本的用户级 ISA 基础和扩展 IMAFDQ 比起本文档的先前版本 [19] 还没有发生变化，但是一些规范漏洞已经被修复，文档也被完善了。一些软件的约定已经发生了改变。

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

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

对 2.0 版本文档的前言

这是用户 ISA 规范的第二次发布，而我们试图让基础用户 ISA 和通用扩展（例如，IMAFD）在未来的发展中保持固定。这个 ISA 规范从 1.0 版本 [18] 开始，已经有了如下改变：

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

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

目录

第一章　介绍

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

- 一个完全*开放*的 ISA，学术界和工业界可以免费获得它。
- 一个*真实*的 ISA，适用于直接的原生的硬件实现，而不仅仅是进行模拟或二进制翻译。
- 一个对于特定微架构样式（例如，微编码的、有序的、解耦的、乱序的）或者实现技术（例如，全定制的、ASIC、FPGA）而言，避免了“过度的架构设计”(译者注：避免采用大而全的复杂微架构，超出了需求)，但在这些的任何一个中都能高效实现的 ISA。
- 一个 ISA 被分成两个部分：1、一个*小型*基础整数 ISA，其可以用作定制加速器或教育目的的基础；2、可选的标准扩展，用于支持通用目的的软件环境。
- 支持已修订的 2008 IEEE-754 浮点标准 [5]。
- 一个支持广泛 ISA 扩展和专用变体的 ISA。
- 32 位和 64 位地址空间的变体都可以用于应用程序、操作系统内核、和硬件实现。
- 一个支持高度并行的多核或众核实现（包括异构多处理器）的 ISA。
- 具有可选的*可变长度指令*，可以扩展可用的指令编码空间，以及支持可选的*稠密指令编码*，以提升性能、静态编码尺寸和能效。
- 一个完全虚拟化的 ISA，以便简化监控器（Hypervisor）的开发。
- 一个使新特权架构上的实验被简化的 ISA。

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

选用 RISC-V 来命名，是为了表示 UC 伯克利设计的第五个主要的 RISC ISA（前四个是 RISC-I [12]、RISC-II [6]、SOAR [17] 和 SPUR [9]）。我们也用罗马字母“V”双关表示“变种（variations）”和“向量（vectors）”，因为，支持包括各种数据并行加速器在内的广泛的架构研究，是此 ISA 设计的一个明确的目标。

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

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

1.1 RISC-V 硬件平台术语

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

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

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

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

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

1.2 RISC-V 软件执行环境和硬件线程

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

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

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

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

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

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

- 由组成向前推进的扩展所定义的任何其它事件。

术语"硬件线程（hart）"的引入是在 Lithe [10, 11] 上的工作中，是为了提供一个表示一种抽象的执行资源的术语，作为与软件线程编程抽象的对应。

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

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

1.3 RISC-V ISA 概览

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

尽管使用"RISC-V ISA"这个词汇很方便，但其实 RISC-V 是一系列相关 ISA 的 ISA 族，族中目前有四个基础 ISA。每个基础整数指令集由不同的整数寄存器宽度、对应的地址空间尺寸和整数寄存器数目作为特征。在第 2 章 二和第 7 章七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 6 章 六描述了 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 8 章 八 概述了基础整数指令集的一个未来变体 RV128I，它将支持扁平的 128 位地址空间（XLEN = 128）。基础整数指令集使用补码来表示有符号的整数值。

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

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

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

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

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

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

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

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

RISC-V 已经被设计为支持广泛的定制和专用化。每个基础整数 ISA 可以加入一个或多个可选的指令集进行扩展。一个扩展可以被归类为标准的、自定义的，或者不合规的。出于这个目的，我们把每个 RISC-V 指令集编码空间（和相关的编码空间，例如 CSR）划分为三个不相交的种类：标准、保留、和自定义。标准扩展和编码由 RISC-V 国际定义；任何不由 RISC-V 国际定义的扩展都是非标准的的。每个基础 ISA 及其标准扩展仅使用标准编码，并且在它们使用这些编码时不能相互冲突。保留的编码当前还没有被定义，是省下来用于未来的标准扩展的；一旦如此使用，它们将变为标准编码。自定义编码应当永远不被用于标准扩展，而是可用于特定供应商的非标准扩展。非标准扩展或者是仅使用自定义编码的自定义扩展，或者是使用了任何标准或保留编码的非合规的扩展。指令集扩展一般是共享的，但是根据基础 ISA 的不同，也可能提供稍微不同的功能。第 27

章 **??**描述了扩展 RISC-V ISA 的各种方法。我们也已经为基于 RISC-V 的指令和指令集开发了一套命名约定，那将在第 28 章 **??**进行详细的描述。

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

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

1.4 内存

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

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

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

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

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

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

除非特别说明，否则，不引发异常的隐式读可能会任意提前地、试探地发生，甚至是在机器能够证明的确需要读之前发生。例如，一种合法实现方式是，可能会尝试第一时间读取所有的主内存，缓存尽可能多的可获取（可执行）字节以供之后的指令获取，以及避免为了指令获取而再次读主内存（译者注：即通常所说的指令预取）为了确保某些隐式读只在写入相同内存位置之后是有序的，软件必须执行为此目的而定义的、特定的屏障指令或缓存控制指令（例如第 3 章 三里定义的 FENCE.I 指令）。

由一个硬件线程发起的内存访问（隐式或显式），在被另一个硬件线程、或者任何其它可访问相同内存的代理线程所感知时，可能看起来像是以一种不同的顺序发生的。然而，这个被感知到的内存访问重新排序总是受到特定的内存一致性模型的约束。用于 RISC-V 的默认的内存一致性模型是 RISC-V 弱内存排序（RVWMO），定义在第 17 章 十一和附录中。也可以采用更强的模型：全存储排序（Total Store Ordering），定义在第 25 章 ??中。执行环境也可以添加约束，进一步限制的可感知的内存访问的重排。由于 RVWMO 模型是被任何 RISC-V 实现所允许的最弱的模型，用这个模型写出的软件兼容所有 RISC-V 实现的实际的内存一致性规则。与隐式读一样，除非假定的内存一致性模型和执行环境有其他特别需求，否则软件必须执行屏障或缓存控制指令来确保特定顺序的内存访问。

1.5 基础指令长度编码

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

数目的 16 位指令的封装包（*parcel*），指令封装包在 16 位边界自然对齐。第 18 章 **??**中描述的标准压缩 ISA 扩展（译者注：即 C 扩展）减少了代码尺寸，通过提供压缩的 16 位指令，以及放松了对齐的限制，允许所有的指令（16 位和 32 位）在任意 16 位边界上对齐，而提升了代码的密度。

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

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

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

拓展的指令长度编码

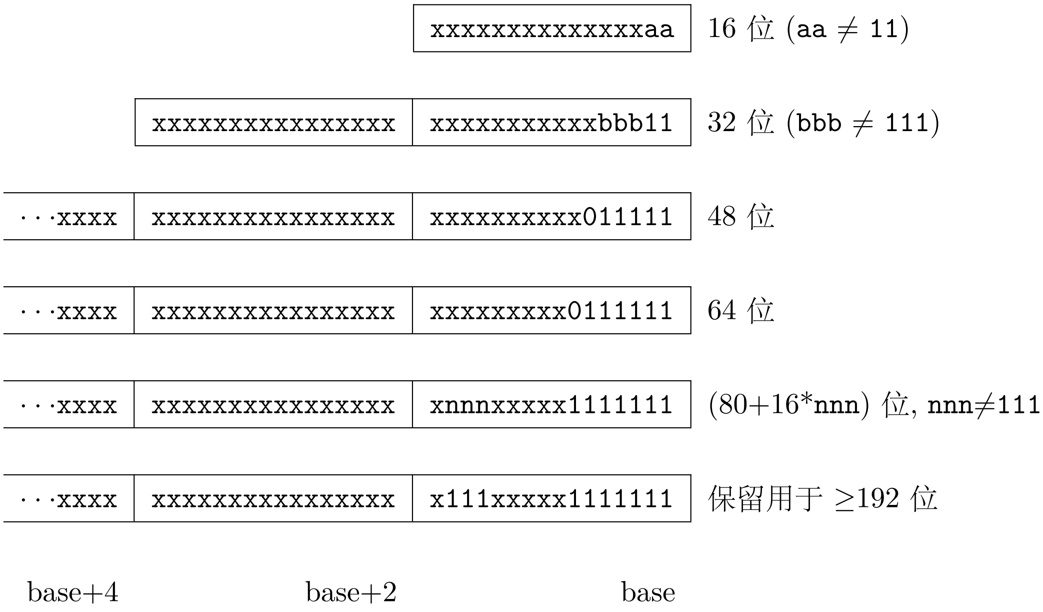
32 位指令编码空间的一部分已经被初步分配给了长度超过 32 位的指令。目前这片空间的整体是被保留的，而且下面的关于超过 32 位编码的提议并没有被认为已被冻结。

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

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

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

也许更重要的是，通过把我们的基础 ISA 凝炼成 32 位指令字的子集，我们为非标准的和自定义的指令集扩展留出了更多可用的空间。特别地，基础 RV32I ISA 在 32 位指令字中使用少于 1/8 的编码空间。正如第 27 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 ≥32 位标准指令集扩展的支持。甚至，如果实现层面也不需要长度 >32 位的指令，它可以把这些不一致扩展恢复成另外四种主要的操作码。



			xxxxxxxxxxxxxxaa	16 位 (aa ≠ 11)
		xxxxxxxxxxxxxxxx	xxxxxxxxxxxbbb11	32 位 (bbb ≠ 111)
	···xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxxx011111	48 位
	···xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxx0111111	64 位
	···xxxx	xxxxxxxxxxxxxxxx	xnnnxxxxx1111111	(80+16*nnn) 位, nnn≠111
	···xxxx	xxxxxxxxxxxxxxxx	x111xxxxx1111111	保留用于 ≥192 位
字节地址:	base+4	base+2	base	

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

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

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

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

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

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

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

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

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

一旦我们已经决定了固定为小字节序指令编码，这将自然地导致把长度编码位放置在指令格式的 *LSB*（译者注：*Least Significant Bit*，最低有效位）的位置，以避免打断操作码域。

1.6 异常、陷入和中断

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

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

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

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

被控制的陷入： 这种陷入对于运行在执行环境中的软件可见，并由软件处理。例如，在一个于硬件线程上同时提供监管器模式和用户模式的 EEI 中，用户模式硬件线程的 ECALL 通常将导

致控制被转移到运行在相同硬件线程上的一个监管器模式的处理程序。类似地，在相同的环境中，当一个硬件线程被中断，硬件线程上将运行一个监管器模式中的中断处理程序。

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

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

致命的陷入： 这种陷入代表了一个致命的失败，并引发执行环境终止执行。例子包括虚拟内存页保护检查的失败，或者看门狗（译者注：watchdog，一种用于访问保护的装置）计时器到期。每个 EEI 应当定义执行应如何被终止，以及如何将其汇报给外部环境。

Table 1.1 显示了每种陷入的特点：

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

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

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

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

1.7 “未指定的”（UNSPECIFIED）行为和值

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

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

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

第二章　RV32I 基础整数指令集，2.1 版本

这章描述了 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，并且可能会把 FENCE 指令实现为一个 NOP，从而把基础指令数目减少到总计 38 条）。RV32I 可以模拟几乎任何其它的 ISA 扩展（除了 A 扩展，因为它需要额外的硬件支持去实现原子性）。

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化– 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。

标准 RISC-V 汇编语言语法的文档在《汇编程序员手册》 [1] 中。

大多数对 RV32I 的注解也适用于 RV64I 基础指令集。

2.1　基础整数 ISA 的编程模型

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

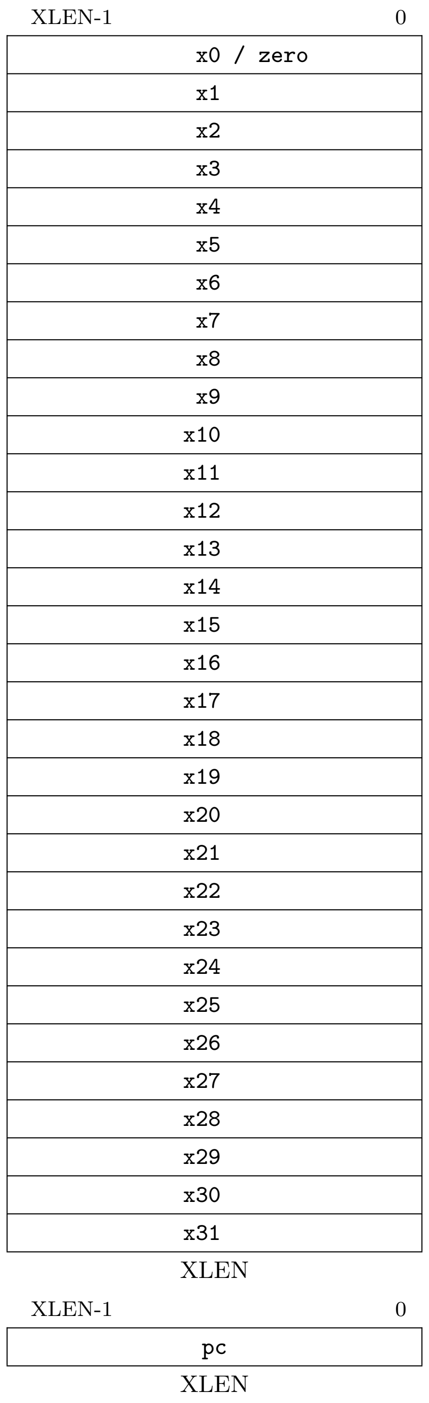


图 2.1: RISC-V 基础非特权整数寄存器状态

在基础整数 *ISA* 中没有专门的栈指针或子程序返回地址链接寄存器；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（非标准约定）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* 位指令中编码一个带有 *16* 个寄存器的完整 *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，可以促成循环展开（*loop unrolling*）、软件管道（*software pipelining*）和缓存平铺（*cache tiling*）的广泛使用。

由于这些原因，我们为 *RV32I* 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 [16]。可选的压缩 *16* 位指令格式大多数只访问 *8* 个寄存器，并因此可以提供一种稠密的指令编码；而额外的指令集扩展，如果愿意，可能支持更大的寄存器空间（或者是扁平的，或者是分层的）。

对于资源受限的嵌入式应用，我们已经定义了 *RV32E* 子集，它只有 *16* 个寄存器（第 *6* 章 六）。

2.2 基础指令格式

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

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

由分支或跳转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，因为这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 11 章 **??**）中使用的 5 位立即数，立即数总是符号扩展的，并且通

<table>
<tr><td>31 25</td><td>24 20</td><td>19 15</td><td>14 12</td><td>11 7</td><td>6 0</td><td></td></tr>
<tr><td>funct7</td><td>rs2</td><td>rs1</td><td>funct3</td><td>rd</td><td>opcode</td><td>R-类型</td></tr>
<tr><td colspan="2">imm[11:0]</td><td>rs1</td><td>funct3</td><td>rd</td><td>opcode</td><td>I-类型</td></tr>
<tr><td>imm[11:5]</td><td>rs2</td><td>rs1</td><td>funct3</td><td>imm[4:0]</td><td>opcode</td><td>S-类型</td></tr>
<tr><td colspan="4">imm[31:12]</td><td>rd</td><td>opcode</td><td>U-类型</td></tr>
</table>

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

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

在实现中，解码寄存器标识符通常在关键路径上。因此在选择指令的格式时，选择在所有格式中的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-IV 的属性，又称 SPUR [9]）。

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

立即数是符号扩展的，因为对于某些立即数（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。

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。

<table>
<tr><th>31</th><th>30 25</th><th>24 21</th><th>20</th><th>19 15</th><th>14 12</th><th>11 8</th><th>7</th><th>6 0</th><th></th></tr>
<tr><td colspan="2">funct7</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td colspan="2">rd</td><td>opcode</td><td>R-类型</td></tr>
<tr><td colspan="4">imm[11:0]</td><td>rs1</td><td>funct3</td><td colspan="2">rd</td><td>opcode</td><td>I-类型</td></tr>
<tr><td colspan="2">imm[11:5]</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td colspan="2">imm[4:0]</td><td>opcode</td><td>S-类型</td></tr>
<tr><td>imm[12]</td><td>imm[10:5]</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td>imm[4:1]</td><td>imm[11]</td><td>opcode</td><td>B-类型</td></tr>
<tr><td colspan="6">imm[31:12]</td><td colspan="2">rd</td><td>opcode</td><td>U-类型</td></tr>
<tr><td>imm[20]</td><td colspan="2">imm[10:1]</td><td>imm[11]</td><td colspan="2">imm[19:12]</td><td colspan="2">rd</td><td>opcode</td><td>J-类型</td></tr>
</table>

图 2.3: 显式立即数的 RISC-V 基础指令格式。

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

<table>
<tr><th>31</th><th>30 20</th><th>19 12</th><th>11</th><th>10 5</th><th>4 1</th><th>0</th><th></th></tr>
<tr><td colspan="4">— inst[31] —</td><td>inst[30:25]</td><td>inst[24:21]</td><td>inst[20]</td><td>I-立即数</td></tr>
<tr><td colspan="4">— inst[31] —</td><td>inst[30:25]</td><td>inst[11:8]</td><td>inst[7]</td><td>S-立即数</td></tr>
<tr><td colspan="3">— inst[31] —</td><td>inst[7]</td><td>inst[30:25]</td><td>inst[11:8]</td><td>0</td><td>B-立即数</td></tr>
<tr><td>inst[31]</td><td>inst[30:20]</td><td>inst[19:12]</td><td colspan="4">— 0 —</td><td>U-立即数</td></tr>
<tr><td colspan="2">— inst[31] —</td><td>inst[19:12]</td><td>inst[20]</td><td>inst[30:25]</td><td>inst[24:21]</td><td>0</td><td>J-立即数</td></tr>
</table>

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

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

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

用器（*MUX*），来将立即数扩大 *2* 倍，我们减少了大约一半的指令符号扇出和立即数多路复用的开销。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，保存在整数寄存器文件中的 XLEN 位的值。整数运算指令或者被编码为使用 I 类型格式的寄存器-立即数操作，或者被编码为使用 R 类型格式的寄存器-寄存器操作。对于寄存器-立即数指令和寄存器-寄存器指令，目的寄存器都是寄存器 *rd*。整数运算指令不会引发算术异常。

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

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

对于一般的有符号加法，在加法之后需要三条额外的指令，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

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

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

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

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

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

按常量移位按照 I 类型格式专门编码。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）。

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

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

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

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

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

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

通过把 *U* 立即数设置为 *0*，可以获得当前 pc。尽管 *JAL +4* 指令也可以被用于获得本地 pc（*JAL* 后续指令的 pc 值），但是，在简单的微架构，它可能引起流水线暂停，或者在更加复杂的微架构中，污染分支目标缓冲区结构。

整数寄存器 - 寄存器操作

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

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

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

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

NOP 指令

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

除了提升 pc 和使任何适用的执行计数器递增以外，NOP 指令不改变任何架构上的可见状态。NOP 被编码为 ADDI *x0, x0, 0*。

NOP 可以被用于把代码段对齐到微架构上的有效地址边界，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 *2.9*）。

选用 *ADDI* 进行 *NOP* 编码是因为，这是在跨多个系统中最可能的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。

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

无条件跳转

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

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寄存器（函数调用时不保存），并且其编码与常规链接寄存器相比只有一位不同。

普通的无条件跳转（汇编器伪指令 J）被编码为 *rd*=`x0` 的 JAL。

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

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

无条件跳转指令都使用 pc 相对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 相对地址范围中的任何地方。

注意 *JALR* 指令不会像条件分支指令那样，把 *12* 位立即数当作 *2* 字节的倍数对待。这回避了硬件中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*2 KiB* 或最高*2 KiB*地址区域的单一指令子例程调用，这可以被用于实现对小型运行时库的快速调用。或者，*ABI* 可以专用于通用目的寄存器，以指向地址空间中任何其它地方的一个库。

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

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

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

一些其它的 *ISA* 把显式的提示位添加到了它们的间接跳转指令上，来指导返回地址栈的操作。我们使用绑定寄存器号码的隐式提示和调用约定，以减少用于这些提示的编码空间。

当两个不同的链接寄存器（x1 和 x5）被给定为 rs1 和 rd 时，接下来 *RAS* 会被同时弹出和推入，以支持协程。如果 rs1 和 rd 是相同的链接寄存器（x1 或着 x5），*RAS* 只把允许宏操作融合推入序列：

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

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

表 2.1: 在 JALR 指令的寄存器操作数中编码的返回地址栈预测提示。

条件分支

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

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

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

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

软件应当被优化为，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流水线，避免了额外的条件代码状态或者临时寄存器的使用，并减少了静态代码的尺寸和动态指令获取的流量。另一点是，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流水线的开销 [*4*, *8*, *7*]，并且已经在商业处理器中被实现 [*14*]。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流水线，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。用于乱序核心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 [*14*]。

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

指令地址未对齐异常不可能发生在支持 *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要引发有可能的任何异常，或其它的副作用。

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。

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

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，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，有效地址不能被以字节为单位的访问大小整除）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模拟，例如，如果对内存区域的访问有副作用，那么可以引发一个访问故障异常而不是一个地址未对齐异常。当 EEI 不保证隐式地处理未对齐的加载和存储时，EEI 必须定义由地址未对齐引起的异常是否导致被包含的陷入（允许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，这些未对齐的访问对应用程序的性能会有帮助。对于允许 *EEI* 选择通过常规的加载和存储指令来支持未对齐的访问，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。特殊指令是难以使用的、让 *ISA* 复杂化的，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

31 28	27	26	25	24	23	22	21	20	19 15	14 12	11 7	6 0
fm	PI	PO	PR	PW	SI	SO	SR	SW	rs1	funct3	rd	opcode
4	1	1	1	1	1	1	1	1	5	3	5	7
FM	前驱				后继				0	FENCE	0	MISC-MEM

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 17 章 十一提供了 RISC-V 内存一致性模型的一个精确的描述。

FENCE 指令也对那些被外部设备发起的内存读写所观察到的、硬件线程发起的内存读和内存写进行排序。然而，FENCE 不对使用任何其它信号机制的外部设备发起的观察事件排序。

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

EEI 将定义什么 I/O 操作是可能的，并且特别地，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。
其他		保留供未来使用。

表 2.2: 屏障模式编码

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

FENCE.TSO 指令被编码为 *fm*=1000、前驱 RW、以及后继 = RW 的 FENCE 指令。FENCE.TSO 把它前驱集合中的所有加载操作排在它后继集合中的所有内存操作之前，并把它前驱

集合中的所有存储操作排在它后继集合中的所有存储操作之前。这使得 FENCE.TSO 的前驱集合中的非 AMO 存储操作与它的后继集合中的非 AMO 加载操作不再有序。

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

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

> 我们选择了一个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 11 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程序。更复杂的实现可能在硬件中执行更多的各系统指令。

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

这两个指令对支持的执行环境引发了一个精确的请求陷入。

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

EBREAK 指令被用于将控制返回到调试环境。

ECALL 和 EBREAK 之前被命名为 SCALL 和 SBREAK。这些指令有相同的功能和编码，但是被重命名了，是为了反映它们可以更一般化地使用，而不只是调用一个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 18 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。像 NOP 指令，除了提升 pc 和任何适用的性能计数器，HINT 不改变任何架构上的可视状态。实现总是被允许忽略已编码的提示。

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

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

作为另一个例子，一个 pred 域为零且 fm 域为零的 FENCE 指令是一个 HINT；succ 域，rs1 域，*and* rd 域编码了 HINT 的参数。一个简单的实现可以把 HINT 作为一个 FENCE 简单

地执行，即，在任何被编码在 succ 域中的后续内存访问之前，对先前内存访问的空集进行排序。由于前驱集和后继集的交集为空，该指令不会施加内存排序，因此它没有架构可见的影响。

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

我们预计标准的提示最终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	$rd=\texttt{x0}$	2^{20}	保留供未来标准使用
AUIPC	$rd=\texttt{x0}$	2^{20}	
ADDI	$rd=\texttt{x0}$, 并且要么 $rs1\neq\texttt{x0}$ 要么 $imm\neq0$	$2^{17}-1$	
ANDI	$rd=\texttt{x0}$	2^{17}	
ORI	$rd=\texttt{x0}$	2^{17}	
XORI	$rd=\texttt{x0}$	2^{17}	
ADD	$rd=\texttt{x0}$, $rs1\neq\texttt{x0}$	$2^{10}-32$	
ADD	$rd=\texttt{x0}$, $rs1=\texttt{x0}$, $rs2\neq\texttt{x2–x5}$	28	
ADD	$rd=\texttt{x0}$, $rs1=\texttt{x0}$, $rs2=\texttt{x2–x5}$	4	($rs2=\texttt{x2}$) NTL.P1 ($rs2=\texttt{x3}$) NTL.PALL ($rs2=\texttt{x4}$) NTL.S1 ($rs2=\texttt{x5}$) NTL.ALL
SUB	$rd=\texttt{x0}$	2^{10}	保留供未来标准使用
AND	$rd=\texttt{x0}$	2^{10}	
OR	$rd=\texttt{x0}$	2^{10}	
XOR	$rd=\texttt{x0}$	2^{10}	
SLL	$rd=\texttt{x0}$	2^{10}	
SRL	$rd=\texttt{x0}$	2^{10}	
SRA	$rd=\texttt{x0}$	2^{10}	
FENCE	$rd=\texttt{x0}$, $rs1\neq\texttt{x0}$, $fm=0$, and either $pred=0$ or $succ=0$	$2^{10}-63$	
FENCE	$rd\neq\texttt{x0}$, $rs1=\texttt{x0}$, $fm=0$, and either $pred=0$ or $succ=0$	$2^{10}-63$	
FENCE	$rd=rs1=\texttt{x0}$, $fm=0$, $pred=0$, $succ\neq0$	15	
FENCE	$rd=rs1=\texttt{x0}$, $fm=0$, $pred\neq\mathrm{W}$, $succ=0$	15	
FENCE	$rd=rs1=\texttt{x0}$, $fm=0$, $pred=\mathrm{W}$, $succ=0$	1	暂停
SLTI	$rd=\texttt{x0}$	2^{17}	指定供自定义使用
SLTIU	$rd=\texttt{x0}$	2^{17}	
SLLI	$rd=\texttt{x0}$	2^{10}	
SRLI	$rd=\texttt{x0}$	2^{10}	
SRAI	$rd=\texttt{x0}$	2^{10}	
SLT	$rd=\texttt{x0}$	2^{10}	
SLTU	$rd=\texttt{x0}$	2^{10}	

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 [15]）。JIT 编译器可以在单个的 FENCE.I 之前生成一大段对指令的追踪，并且通过把翻译过的指令写到已知的没有保留在 I-缓存中的内存区域，分摊任何指令缓存的嗅探/失效负载。

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

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

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

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

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

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

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

FENCE.I 指令中的未使用的域，imm[11:0]、rs1 和 rd，被保留用于未来扩展中的更细粒度的屏障功能。为了向前兼容，基础实现应当忽略这些域，而标准软件应当把这些域置为零。The unused fields in the FENCE.I instruction, *imm[11:0]*, *rs1*, and *rd*, are reserved for finer-grain fences in future extensions. For forward compatibility, base implementations shall ignore these fields, and standard software shall zero these fields.

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

第四章 “Zihintntl”非时间局部性提示（0.2 版本）

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

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

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

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

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

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

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

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

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

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

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

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

表 4.1: 为可移植软件推荐的在各种场景中采用的 NTL 变体。

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

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

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

表 4.2: NTL 变体到各种内存层次的映射。

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

NTL 指令影响除 Zicbom 扩展中缓存管理指令外的所有内存访问指令。

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

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

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

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

为了预取到最内层的缓存中，不要将 NTL 指令作为预取指令的前缀。

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

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

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

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

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

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

第五章 “Zihintpause”暂停提示（2.0 版本）

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

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

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

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

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

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

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

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

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

第六章　RV32E 和 RV64E 基础整数指令集 (1.95 版本)

这章描述了一个 RV32E 和 RV64E 基础整数指令集的建议草案，它们是为嵌入式系统的微控制器设计的。RV32E 和 RV64E 分别是 RV32I 和 RV64I 的简化版本：仅有的改变是把整数寄存器的数目减少到了 16 个。这章仅仅概述了 RV32E/RV64E 和 RV32I/RV64I 之间的不同，并因此应当被放在第 2 章 二和第 7 章七之后阅读。

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

RV32E 和 *RV64E* 可以与所有当前的标准扩展相组合。

6.1　RV32E 和 RV64E 编程模型

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

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

6.2　RV32E 和 RV64E 指令集编码

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

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

第七章　RV64I 基础整数指令集（2.1 版本）

这章描述了 RV64I 基础整数指令集，它是在第 2 章 二中描述的 RV32I 变体之上构建的。这章只呈现了与 RV32I 的不同，所以应当与那篇更早的章节结合着阅读。

7.1　寄存器状态

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

7.2　整数运算指令

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

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

整数寄存器 - 立即数指令

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

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

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

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

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

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

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

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

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

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

整数寄存器—寄存器操作

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

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

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

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

7.3 加载和存储指令

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

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

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

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

7.4 “提示”指令

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

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

Instruction	Constraints	Code Points	Purpose
LUI	*rd*=x0	2^{20}	保留供未来标准使用
AUIPC	*rd*=x0	2^{20}	
ADDI	*rd*=x0, 要么 *rs1*≠x0 要么 *imm*≠0	$2^{17}-1$	
ANDI	*rd*=x0	2^{17}	
ORI	*rd*=x0	2^{17}	
XORI	*rd*=x0	2^{17}	
ADDIW	*rd*=x0	2^{17}	
ADD	*rd*=x0, *rs1*≠x0	$2^{10}-32$	
ADD	*rd*=x0, *rs1*=x0, *rs2*≠x2–x5	28	
ADD	*rd*=x0, *rs1*=x0, *rs2*=x2–x5	4	(*rs2*=x2) NTL.P1 (*rs2*=x3) NTL.PALL (*rs2*=x4) NTL.S1 (*rs2*=x5) NTL.ALL
SUB	*rd*=x0	2^{10}	保留供未来标准使用
AND	*rd*=x0	2^{10}	
OR	*rd*=x0	2^{10}	
XOR	*rd*=x0	2^{10}	
SLL	*rd*=x0	2^{10}	
SRL	*rd*=x0	2^{10}	
SRA	*rd*=x0	2^{10}	
ADDW	*rd*=x0	2^{10}	
SUBW	*rd*=x0	2^{10}	
SLLW	*rd*=x0	2^{10}	
SRLW	*rd*=x0	2^{10}	
SRAW	*rd*=x0	2^{10}	
FENCE	*rd*=x0, *rs1*≠x0, *fm*=0, and either *pred*=0 or *succ*=0	$2^{10}-63$	
FENCE	*rd*≠x0, *rs1*=x0, *fm*=0, and either *pred*=0 or *succ*=0	$2^{10}-63$	
FENCE	*rd*=*rs1*=x0, *fm*=0, *pred*=0, *succ*≠0	15	
FENCE	*rd*=*rs1*=x0, *fm*=0, *pred*≠W, *succ*=0	15	
FENCE	*rd*=*rs1*=x0, *fm*=0, *pred*=W, *succ*=0	1	PAUSE
SLTI	*rd*=x0	2^{17}	

第八章　RV128I 基础整数指令集（1.7 版本）

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

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

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

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

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

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

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

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

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

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

第九章 用于乘法和除法的“M”标准扩展（2.0 版本）

This chapter describes the standard integer multiplication and division instruction extension, which is named “M” and contains instructions that multiply or divide values held in two integer registers.

We separate integer multiply and divide out from the base to simplify low-end implementations, or for applications where integer multiply and divide operations are either infrequent or better handled in attached accelerators.

9.1 Multiplication Operations

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

MUL performs an XLEN-bit×XLEN-bit multiplication of *rs1* by *rs2* and places the lower XLEN bits in the destination register. MULH, MULHU, and MULHSU perform the same multiplication but return the upper XLEN bits of the full 2×XLEN-bit product, for signed×signed, unsigned×unsigned, and signed *rs1*×unsigned *rs2* multiplication, respectively. If both the high and low bits of the same product are required, then the recommended code sequence is: MULH[[S]U] *rdh, rs1, rs2*; MUL *rdl, rs1, rs2* (source register specifiers must be in same order and *rdh* cannot be the same as *rs1* or *rs2*). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies.

MULHSU is used in multi-word signed multiplication to multiply the most-significant word of the multiplicand (which contains the sign bit) with the less-significant words of the multiplier (which are unsigned).

MULW is an RV64 instruction that multiplies the lower 32 bits of the source registers, placing the sign-extension of the lower 32 bits of the result into the destination register.

In RV64, MUL can be used to obtain the upper 32 bits of the 64-bit product, but signed arguments must be proper 32-bit signed values, whereas unsigned arguments must have their upper 32 bits clear. If the arguments are not known to be sign- or zero-extended, an alternative is to shift both arguments left by 32 bits, then use MULH[[S]U].

9.2 Division Operations

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

DIV and DIVU perform an XLEN bits by XLEN bits signed and unsigned integer division of *rs1* by *rs2*, rounding towards zero. REM and REMU provide the remainder of the corresponding division operation. For REM, the sign of the result equals the sign of the dividend.

For both signed and unsigned division, it holds that $dividend = divisor \times quotient + remainder$.

If both the quotient and remainder are required from the same division, the recommended code sequence is: DIV[U] *rdq, rs1, rs2*; REM[U] *rdr, rs1, rs2* (*rdq* cannot be the same as *rs1* or *rs2*). Microarchitectures can then fuse these into a single divide operation instead of performing two separate divides.

DIVW and DIVUW are RV64 instructions that divide the lower 32 bits of *rs1* by the lower 32 bits of *rs2*, treating them as signed and unsigned integers respectively, placing the 32-bit quotient in *rd*, sign-extended to 64 bits. REMW and REMUW are RV64 instructions that provide the corresponding signed and unsigned remainder operations respectively. Both REMW and REMUW always sign-extend the 32-bit result to 64 bits, including on a divide by zero.

The semantics for division by zero and division overflow are summarized in Table 9.1. The quotient of division by zero has all bits set, and the remainder of division by zero equals the dividend. Signed division overflow occurs only when the most-negative integer is divided by -1. The quotient of a signed division with overflow is equal to the dividend, and the remainder is zero. Unsigned division overflow cannot occur.

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

表 9.1: Semantics for division by zero and division overflow. L is the width of the operation in bits: XLEN for DIV[U] and REM[U], or 32 for DIV[U]W and REM[U]W.

We considered raising exceptions on integer divide by zero, with these exceptions causing a trap in most execution environments. However, this would be the only arithmetic trap in the standard ISA (floating-point exceptions set flags and write default values, but do not cause traps) and would require language implementors to interact with the execution environment's trap handlers for this case. Further, where language standards mandate that a divide-by-zero exception must cause an immediate control flow change, only a single branch instruction needs to be added to each divide operation, and this branch instruction can be inserted after the divide and should normally be very predictably not taken, adding little runtime overhead.

The value of all bits set is returned for both unsigned and signed divide by zero to simplify the divider circuitry. The value of all 1s is both the natural value to return for unsigned divide, representing the largest unsigned number, and also the natural result for simple unsigned divider implementations. Signed division is often implemented using an unsigned division circuit and specifying the same overflow result simplifies the hardware.

9.3 Zmmul Extension, Version 1.0

The Zmmul extension implements the multiplication subset of the M extension. It adds all of the instructions defined in Section 9.1, namely: MUL, MULH, MULHU, MULHSU, and (for RV64 only) MULW. The encodings are identical to those of the corresponding M-extension instructions.

The Zmmul extension enables low-cost implementations that require multiplication operations but not division. For many microcontroller applications, division operations are too infrequent to justify the cost of divider hardware. By contrast, multiplication operations are more frequent, making the cost of multiplier hardware more justifiable. Simple FPGA soft cores particularly benefit from eliminating division but retaining multiplication, since many FPGAs provide hardwired multipliers but require dividers be implemented in soft logic.

第十章　用于原子指令的“A”标准扩展（2.1 版本）

The standard atomic-instruction extension, named “A”, contains instructions that atomically read-modify-write memory to support synchronization between multiple RISC-V harts running in the same memory space. The two forms of atomic instruction provided are load-reserved/store-conditional instructions and atomic fetch-and-op memory instructions. Both types of atomic instruction support various memory consistency orderings including unordered, acquire, release, and sequentially consistent semantics. These instructions allow RISC-V to support the RCsc memory consistency model [3].

After much debate, the language community and architecture community appear to have finally settled on release consistency as the standard memory consistency model and so the RISC-V atomic support is built around this model.

10.1 Specifying Ordering of Atomic Instructions

The base RISC-V ISA has a relaxed memory model, with the FENCE instruction used to impose additional ordering constraints. The address space is divided by the execution environment into memory and I/O domains, and the FENCE instruction provides options to order accesses to one or both of these two address domains.

To provide more efficient support for release consistency [3], each atomic instruction has two bits, *aq* and *rl*, used to specify additional memory ordering constraints as viewed by other RISC-V harts. The bits order accesses to one of the two address domains, memory or I/O, depending on which address domain the atomic instruction is accessing. No ordering constraint is implied to accesses to the other domain, and a FENCE instruction should be used to order across both domains.

If both bits are clear, no additional ordering constraints are imposed on the atomic memory operation. If only the *aq* bit is set, the atomic memory operation is treated as an *acquire* access, i.e., no following memory operations on this RISC-V hart can be observed to take place before the acquire memory operation. If only the *rl* bit is set, the atomic memory operation is treated as a *release* access, i.e., the release memory operation cannot be observed to take place before any earlier memory operations on this RISC-V hart. If both the *aq* and *rl* bits are set, the atomic memory operation is *sequentially consistent* and cannot be observed to happen before any earlier memory operations or after any later memory operations in the same RISC-V hart and to the same address domain.

10.2 Load-Reserved/Store-Conditional Instructions

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

Complex atomic memory operations on a single memory word or doubleword are performed with the load-reserved (LR) and store-conditional (SC) instructions. LR.W loads a word from the address in *rs1*, places the sign-extended value in *rd*, and registers a *reservation set*—a set of bytes that subsumes the bytes in the addressed word. SC.W conditionally writes a word in *rs2* to the address in *rs1*: the SC.W succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the SC.W succeeds, the instruction writes the word in *rs2* to memory, and it writes zero to *rd*. If the SC.W fails, the instruction does not write to memory, and it writes a nonzero value to *rd*. Regardless of success or failure, executing an SC.W instruction invalidates any reservation held by this hart. LR.D and SC.D act analogously on doublewords and are only available on RV64. For RV64, LR.W and SC.W sign-extend the value placed in *rd*.

Both compare-and-swap (CAS) and LR/SC can be used to build lock-free data structures. After extensive discussion, we opted for LR/SC for several reasons: 1) CAS suffers from the ABA problem, which LR/SC avoids because it monitors all writes to the address rather than only checking for changes in the data value; 2) CAS would also require a new integer instruction format to support three source operands (address, compare value, swap value) as well as a different memory system message format, which would complicate microarchitectures; 3) Furthermore, to avoid the ABA problem, other systems provide a double-wide CAS (DW-CAS) to allow a counter to be tested and incremented along with a data word. This requires reading five registers and writing two in one instruction, and also a new larger memory system message type, further complicating implementations; 4) LR/SC provides a more efficient implementation of many *primitives as it only requires one load as opposed to two with CAS (one load before the CAS instruction to obtain a value for speculative computation, then a second load as part of the CAS instruction to check if value is unchanged before updating).*

The main disadvantage of LR/SC over CAS is livelock, which we avoid, under certain circumstances, with an architected guarantee of eventual forward progress as described below. Another concern is whether the influence of the current x86 architecture, with its DW-CAS, will complicate porting of synchronization libraries and other software that assumes DW-CAS is the basic machine primitive. A possible mitigating factor is the recent addition of transactional memory instructions to x86, which might cause a move away from DW-CAS.

More generally, a multi-word atomic primitive is desirable, but there is still considerable debate about what form this should take, and guaranteeing forward progress adds complexity to a system.

The failure code with value 1 encodes an unspecified failure. Other failure codes are reserved at this time. Portable software should only assume the failure code will be non-zero.

We reserve a failure code of 1 to mean "unspecified" so that simple implementations may return this value using the existing mux required for the SLT/SLTU instructions. More specific failure codes might be defined in future versions or extensions to the ISA.

For LR and SC, the A extension requires that the address held in *rs1* be naturally aligned to the size of the operand (i.e., eight-byte aligned for 64-bit words and four-byte aligned for 32-bit words). If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

Emulating misaligned LR/SC sequences is impractical in most systems.

Misaligned LR/SC sequences also raise the possibility of accessing multiple reservation sets at once, which present definitions do not provide for.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

Following this model, in systems with memory translation, an SC is allowed to succeed if the earlier LR reserved the same location using an alias with a different virtual address, but is also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only required to invalidate reservations when they overlap the bytes accessed by the LR. These writes are not required to invalidate the reservation when they access other bytes in the reservation set.

The SC must fail if the address is not within the reservation set of the most recent LR in program order. The SC must fail if a store to the reservation set from another hart can be observed to occur between the LR and SC. The SC must fail if a write from some other device to the bytes accessed by the LR can be observed to occur between the LR and SC. (If such a device writes the reservation set but does not write the bytes accessed by the LR, the SC may or may not fail.) An SC must fail if there is another SC (to any address) between the LR and the SC in program order. The precise statement of the atomicity requirements for successful LR/SC sequences is defined by the Atomicity Axiom in Section 11.1.

The platform should provide a means to determine the size and shape of the reservation set.

A platform specification may constrain the size and shape of the reservation set.

A store-conditional instruction to a scratch word of memory should be used to forcibly invalidate any existing load reservation:

- *during a preemptive context switch, and*
- *if necessary when changing virtual to physical address mappings, such as when migrating pages that might contain an active reservation.*

The invalidation of a hart's reservation when it executes an LR or SC imply that a hart can only hold one reservation at a time, and that an SC can only pair with the most recent LR, and LR with the next following SC, in program order. This is a restriction to the Atomicity Axiom in Section 11.1 that ensures software runs correctly on expected common implementations that operate in this manner.

An SC instruction can never be observed by another RISC-V hart before the LR instruction that established the reservation. The LR/SC sequence can be given acquire semantics by setting the *aq* bit on the LR instruction. The LR/SC sequence can be given release semantics by setting the *rl* bit on the SC instruction. Setting the *aq* bit on the LR instruction, and setting both the *aq* and the *rl* bit on the SC instruction makes the LR/SC sequence sequentially consistent, meaning that it cannot be reordered with earlier or later memory operations from the same hart.

If neither bit is set on both LR and SC, the LR/SC sequence can be observed to occur before or after surrounding memory operations from the same RISC-V hart. This can be appropriate when the LR/SC sequence is used to implement a parallel reduction operation.

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set, nor should software set the *aq* bit on an SC instruction unless the *rl* bit is also set. LR.*rl* and SC.*aq* instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

```
    # a0 holds address of memory location
    # a1 holds expected value
    # a2 holds desired value
    # a0 holds return value, 0 if successful, !0 otherwise
cas:
    lr.w t0, (a0)         # Load original value.
    bne t0, a1, fail      # Doesn't match, so fail.
    sc.w t0, a2, (a0)     # Try to update.
    bnez t0, cas          # Retry if store-conditional failed.
    li a0, 0              # Set return to success.
    jr ra                 # Return.
fail:
    li a0, 1              # Set return to failure.
    jr ra                 # Return.
```

图 10.1: Sample code for compare-and-swap function using LR/SC.

LR/SC can be used to construct lock-free data structures. An example using LR/SC to implement a compare-and-swap function is shown in Figure 10.1. If inlined, compare-and-swap functionality need only take four instructions.

10.3 Eventual Success of Store-Conditional Instructions

The standard A extension defines *constrained LR/SC loops*, which have the following properties:

- The loop comprises only an LR/SC sequence and code to retry the sequence in the case of failure, and must comprise at most 16 instructions placed sequentially in memory.

- An LR/SC sequence begins with an LR instruction and ends with an SC instruction. The dynamic code executed between the LR and SC instructions can only contain instructions from the base "I" instruction set, excluding loads, stores, backward jumps, taken backward branches, JALR, FENCE, and SYSTEM instructions. If the "C" extension is supported, then compressed forms of the aforementioned "I" instructions are also permitted.
- The code to retry a failing LR/SC sequence can contain backwards jumps and/or branches to repeat the LR/SC sequence, but otherwise has the same constraint as the code between the LR and SC.
- The LR and SC addresses must lie within a memory region with the *LR/SC eventuality* property. The execution environment is responsible for communicating which regions have this property.
- The SC must be to the same effective address and of the same data size as the latest LR executed by the same hart.

LR/SC sequences that do not lie within constrained LR/SC loops are *unconstrained*. Unconstrained LR/SC sequences might succeed on some attempts on some implementations, but might never succeed on other implementations.

We restricted the length of LR/SC loops to fit within 64 contiguous instruction bytes in the base ISA to avoid undue restrictions on instruction cache and TLB size and associativity. Similarly, we disallowed other loads and stores within the loops to avoid restrictions on data-cache associativity in simple implementations that track the reservation within a private cache. The restrictions on branches and jumps limit the time that can be spent in the sequence. Floating-point operations and integer multiply/divide were disallowed to simplify the operating system's emulation of these instructions on implementations lacking appropriate hardware support.

Software is not forbidden from using unconstrained LR/SC sequences, but portable software must detect the case that the sequence repeatedly fails, then fall back to an alternate code sequence that does not rely on an unconstrained LR/SC sequence. Implementations are permitted to unconditionally fail any unconstrained LR/SC sequence.

If a hart *H* enters a constrained LR/SC loop, the execution environment must guarantee that one of the following events eventually occurs:

- *H* or some other hart executes a successful SC to the reservation set of the LR instruction in *H*'s constrained LR/SC loops.
- Some other hart executes an unconditional store or AMO instruction to the reservation set of the LR instruction in *H*'s constrained LR/SC loop, or some other device in the system writes to that reservation set.
- *H* executes a branch or jump that exits the constrained LR/SC loop.

- *H* traps.

Note that these definitions permit an implementation to fail an SC instruction occasionally for any reason, provided the aforementioned guarantee is not violated.

As a consequence of the eventuality guarantee, if some harts in an execution environment are executing constrained LR/SC loops, and no other harts or devices in the execution environment execute an unconditional store or AMO to that reservation set, then at least one hart will eventually exit its constrained LR/SC loop. By contrast, if other harts or devices continue to write to that reservation set, it is not guaranteed that any hart will exit its LR/SC loop.

Loads and load-reserved instructions do not by themselves impede the progress of other harts' LR/SC sequences. We note this constraint implies, among other things, that loads and load-reserved instructions executed by other harts (possibly within the same core) cannot impede LR/SC progress indefinitely. For example, cache evictions caused by another hart sharing the cache cannot impede LR/SC progress indefinitely. Typically, this implies reservations are tracked independently of evictions from any shared cache. Similarly, cache misses caused by speculative execution within a hart cannot impede LR/SC progress indefinitely.

These definitions admit the possibility that SC instructions may spuriously fail for implementation reasons, provided progress is eventually made.

One advantage of CAS is that it guarantees that some hart eventually makes progress, whereas an LR/SC atomic sequence could livelock indefinitely on some systems. To avoid this concern, we added an architectural guarantee of livelock freedom for certain LR/SC sequences.

Earlier versions of this specification imposed a stronger starvation-freedom guarantee. However, the weaker livelock-freedom guarantee is sufficient to implement the C11 and C++11 languages, and is substantially easier to provide in some microarchitectural styles.

10.4 Atomic Memory Operations

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

The atomic memory operation (AMO) instructions perform read-modify-write operations for multiprocessor synchronization and are encoded with an R-type instruction format. These AMO instructions atomically load a data value from the address in *rs1*, place the value into register *rd*, apply a binary operator to the loaded value and the original value in *rs2*, then store the result back to the original address in *rs1*. AMOs can either operate on 64-bit (RV64 only) or 32-bit words in memory. For RV64, 32-bit AMOs always sign-extend the value placed in *rd*, and ignore the upper 32 bits of the original value of *rs2*.

For AMOs, the A extension requires that the address held in *rs1* be naturally aligned to the size of the operand (i.e., eight-byte aligned for 64-bit words and four-byte aligned for 32-bit words). If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated. The "Zam" extension, described in Chapter **??**, relaxes this requirement and specifies the semantics of misaligned AMOs.

The operations supported are swap, integer add, bitwise AND, bitwise OR, bitwise XOR, and signed and unsigned integer maximum and minimum. Without ordering constraints, these AMOs can be used to implement parallel reduction operations, where typically the return value would be discarded by writing to `x0`.

We provided fetch-and-op style atomic primitives as they scale to highly parallel systems better than LR/SC or CAS. A simple microarchitecture can implement AMOs using the LR/SC primitives, provided the implementation can guarantee the AMO eventually completes. More complex *implementations might also implement AMOs at memory controllers, and can optimize away fetching the original value when the destination is* x0*.*

The set of AMOs was chosen to support the C11/C++11 atomic memory operations efficiently, and also to support parallel reductions in memory. Another use of AMOs is to provide atomic updates to memory-mapped device registers (e.g., setting, clearing, or toggling bits) in the I/O space.

To help implement multiprocessor synchronization, the AMOs optionally provide release consistency semantics. If the *aq* bit is set, then no later memory operations in this RISC-V hart can be observed to take place before the AMO. Conversely, if the *rl* bit is set, then other RISC-V harts will not observe the AMO before memory accesses preceding the AMO in this RISC-V hart. Setting both the *aq* and the *rl* bit on an AMO makes the sequence sequentially consistent, meaning that it cannot be reordered with earlier or later memory operations from the same hart.

The AMOs were designed to implement the C11 and C++11 memory models efficiently. Although the FENCE R, RW instruction suffices to implement the acquire *operation and FENCE RW, W suffices to implement* release*, both imply additional unnecessary ordering as compared to AMOs with the corresponding* aq *or* rl *bit set.*

An example code sequence for a critical section guarded by a test-and-test-and-set spinlock is shown in Figure 10.2. Note the first AMO is marked *aq* to order the lock acquisition before the critical section, and the second AMO is marked *rl* to order the critical section before the lock relinquishment.

```
    li           t0, 1        # Initialize swap value.
again:
    lw           t1, (a0)     # Check if lock is held.
    bnez         t1, again    # Retry if held.
    amoswap.w.aq t1, t0, (a0) # Attempt to acquire lock.
    bnez         t1, again    # Retry if held.
    # ...
    # Critical section.
    # ...
    amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
```

图 10.2: Sample code for mutual exclusion. a0 contains the address of the lock.

We recommend the use of the AMO Swap idiom shown above for both lock acquire and release to simplify the implementation of speculative lock elision [13].

The instructions in the “A” extension can also be used to provide sequentially consistent loads and stores. A sequentially consistent load can be implemented as an LR with both *aq* and *rl* set. A sequentially consistent store can be implemented as an AMOSWAP that writes the old value to x0 and has both *aq* and *rl* set.

第十一章 RVWMO Memory Consistency Model, Version 2.0

This chapter defines the RISC-V memory consistency model. A memory consistency model is a set of rules specifying the values that can be returned by loads of memory. RISC-V uses a memory model called “RVWMO” (RISC-V Weak Memory Ordering) which is designed to provide flexibility for architects to build high-performance scalable designs while simultaneously supporting a tractable programming model.

Under RVWMO, code running on a single hart appears to execute in order from the perspective of other memory instructions in the same hart, but memory instructions from another hart may observe the memory instructions from the first hart being executed in a different order. Therefore, multithreaded code may require explicit synchronization to guarantee ordering between memory instructions from different harts. The base RISC-V ISA provides a FENCE instruction for this purpose, described in Section 2.7, while the atomics extension “A” additionally defines load-reserved/store-conditional and atomic read-modify-write instructions.

The standard ISA extension for misaligned atomics “Zam” (Chapter **??**) and the standard ISA extension for total store ordering “Ztso” (Chapter **??**) augment RVWMO with additional rules specific to those extensions.

The appendices to this specification provide both axiomatic and operational formalizations of the memory consistency model as well as additional explanatory material.

This chapter defines the memory model for regular main memory operations. The interaction of the memory model with I/O memory, instruction fetches, FENCE.I, page table walks, and SFENCE.VMA is not (yet) formalized. Some or all of the above may be formalized in a future revision of this specification. The RV128 base ISA and future ISA extensions such as the “V” vector and “J” JIT extensions will need to be incorporated into a future revision as well.

Memory consistency models supporting overlapping memory accesses of different widths simultaneously remain an active area of academic research and are not yet fully understood.

The specifics of how memory accesses of different sizes interact under RVWMO are specified to the best of our current abilities, but they are subject to revision should new issues be uncovered.

11.1 Definition of the RVWMO Memory Model

The RVWMO memory model is defined in terms of the *global memory order*, a total ordering of the memory operations produced by all harts. In general, a multithreaded program has many different possible executions, with each execution having its own corresponding global memory order.

The global memory order is defined over the primitive load and store operations generated by memory instructions. It is then subject to the constraints defined in the rest of this chapter. Any execution satisfying all of the memory model constraints is a legal execution (as far as the memory model is concerned).

Memory Model Primitives

The *program order* over memory operations reflects the order in which the instructions that generate each load and store are logically laid out in that hart's dynamic instruction stream; i.e., the order in which a simple in-order processor would execute the instructions of that hart.

Memory-accessing instructions give rise to *memory operations*. A memory operation can be either a *load operation*, a *store operation*, or both simultaneously. All memory operations are single-copy atomic: they can never be observed in a partially complete state.

Among instructions in RV32GC and RV64GC, each aligned memory instruction gives rise to exactly one memory operation, with two exceptions. First, an unsuccessful SC instruction does not give rise to any memory operations. Second, FLD and FSD instructions may each give rise to multiple memory operations if XLEN<64, as stated in Section **??** and clarified below. An aligned AMO gives rise to a single memory operation that is both a load operation and a store operation simultaneously.

Instructions in the RV128 base instruction set and in future ISA extensions such as V (vector) and P (SIMD) may give rise to multiple memory operations. However, the memory model for these extensions has not yet been formalized.

A misaligned load or store instruction may be decomposed into a set of component memory operations of any granularity. An FLD or FSD instruction for which XLEN<64 may also be decomposed into a set of component memory operations of any granularity. The memory operations generated by such instructions are not ordered with respect to each other in program order, but they are ordered normally with respect to the memory operations generated by preceding and subsequent instructions in program order. The atomics extension "A" does not require execution environments to support misaligned atomic instructions at all; however, if misaligned atomics are supported via the "Zam" extension, LRs, SCs, and AMOs may be decomposed subject to the constraints of the atomicity axiom for misaligned atomics, which is defined in Chapter **??**.

The decomposition of misaligned memory operations down to byte granularity facilitates emulation on implementations that do not natively support misaligned accesses. Such implementations might, for example, simply iterate over the bytes of a misaligned access one by one.

An LR instruction and an SC instruction are said to be *paired* if the LR precedes the SC in program order and if there are no other LR or SC instructions in between; the corresponding memory operations are said to be paired as well (except in case of a failed SC, where no store operation is generated). The complete list of conditions determining whether an SC must succeed, may succeed, or must fail is defined in Section 10.2.

Load and store operations may also carry one or more ordering annotations from the following set: "acquire-RCpc", "acquire-RCsc", "release-RCpc", and "release-RCsc". An AMO or LR instruction with *aq* set has an "acquire-RCsc" annotation. An AMO or SC instruction with *rl* set has a "release-RCsc" annotation. An AMO, LR, or SC instruction with both *aq* and *rl* set has both "acquire-RCsc" and "release-RCsc" annotations.

For convenience, we use the term "acquire annotation" to refer to an acquire-RCpc annotation or an acquire-RCsc annotation. Likewise, a "release annotation" refers to a release-RCpc annotation or a release-RCsc annotation. An "RCpc annotation" refers to an acquire-RCpc annotation or a release-RCpc annotation. An "RCsc annotation" refers to an acquire-RCsc annotation or a release-RCsc annotation.

In the memory model literature, the term "RCpc" stands for release consistency with processor-consistent synchronization operations, and the term "RCsc" stands for release consistency with sequentially consistent synchronization operations [3].

While there are many different definitions for acquire and release annotations in the literature, in the context of RVWMO these terms are concisely and completely defined by Preserved Program Order rules 5–7.

> *"RCpc" annotations are currently only used when implicitly assigned to every memory access per the standard extension "Ztso" (Chapter* **??***). Furthermore, although the ISA does not currently contain native load-acquire or store-release instructions, nor RCpc variants thereof, the RVWMO model itself is designed to be forwards-compatible with the potential addition of any or all of the above into the ISA in a future extension.*

Syntactic Dependencies

The definition of the RVWMO memory model depends in part on the notion of a syntactic dependency, defined as follows.

In the context of defining dependencies, a "register" refers either to an entire general-purpose register, some portion of a CSR, or an entire CSR. The granularity at which dependencies are tracked through CSRs is specific to each CSR and is defined in Section 11.2.

Syntactic dependencies are defined in terms of instructions' *source registers*, instructions' *destination registers*, and the way instructions *carry a dependency* from their source registers to their destination registers. This section provides a general definition of all of these terms; however, Section 11.3 provides a complete listing of the specifics for each instruction.

In general, a register r other than `x0` is a *source register* for an instruction i if any of the following hold:

- In the opcode of i, *rs1*, *rs2*, or *rs3* is set to r
- i is a CSR instruction, and in the opcode of i, *csr* is set to r, unless i is CSRRW or CSRRWI and *rd* is set to `x0`
- r is a CSR and an implicit source register for i, as defined in Section 11.3
- r is a CSR that aliases with another source register for i

Memory instructions also further specify which source registers are *address source registers* and which are *data source registers*.

In general, a register r other than `x0` is a *destination register* for an instruction i if any of the following hold:

- In the opcode of i, *rd* is set to r

- i is a CSR instruction, and in the opcode of i, *csr* is set to r, unless i is CSRRS or CSRRC and *rs1* is set to `x0` or i is CSRRSI or CSRRCI and uimm[4:0] is set to zero.

- r is a CSR and an implicit destination register for i, as defined in Section 11.3

- r is a CSR that aliases with another destination register for i

Most non-memory instructions *carry a dependency* from each of their source registers to each of their destination registers. However, there are exceptions to this rule; see Section 11.3

Instruction j has a *syntactic dependency* on instruction i via destination register s of i and source register r of j if either of the following hold:

- s is the same as r, and no instruction program-ordered between i and j has r as a destination register

- There is an instruction m program-ordered between i and j such that all of the following hold:

  1. j has a syntactic dependency on m via destination register q and source register r
  2. m has a syntactic dependency on i via destination register s and source register p
  3. m carries a dependency from p to q

Finally, in the definitions that follow, let a and b be two memory operations, and let i and j be the instructions that generate a and b, respectively.

b has a *syntactic address dependency* on a if r is an address source register for j and j has a syntactic dependency on i via source register r

b has a *syntactic data dependency* on a if b is a store operation, r is a data source register for j, and j has a syntactic dependency on i via source register r

b has a *syntactic control dependency* on a if there is an instruction m program-ordered between i and j such that m is a branch or indirect jump and m has a syntactic dependency on i.

Generally speaking, non-AMO load instructions do not have data source registers, and unconditional non-AMO store instructions do not have destination registers. However, a successful SC instruction is considered to have the register specified in rd *as a destination register, and hence it is possible for an instruction to have a syntactic dependency on a successful SC instruction that precedes it in program order.*

Preserved Program Order

The global memory order for any given execution of a program respects some but not all of each hart's program order. The subset of program order that must be respected by the global memory order is known as *preserved program order*.

The complete definition of preserved program order is as follows (and note that AMOs are simultaneously both loads and stores): memory operation a precedes memory operation b in preserved program order (and hence also in the global memory order) if a precedes b in program order, a and b both access regular main memory (rather than I/O regions), and any of the following hold:

- Overlapping-Address Orderings:

  1. b is a store, and a and b access overlapping memory addresses
  2. a and b are loads, x is a byte read by both a and b, there is no store to x between a and b in program order, and a and b return values for x written by different memory operations
  3. a is generated by an AMO or SC instruction, b is a load, and b returns a value written by a

- Explicit Synchronization

  4. There is a FENCE instruction that orders a before b
  5. a has an acquire annotation
  6. b has a release annotation
  7. a and b both have RCsc annotations
  8. a is paired with b

- Syntactic Dependencies

  9. b has a syntactic address dependency on a
  10. b has a syntactic data dependency on a
  11. b is a store, and b has a syntactic control dependency on a

- Pipeline Dependencies

  12. b is a load, and there exists some store m between a and b in program order such that m has an address or data dependency on a, and b returns a value written by m
  13. b is a store, and there exists some instruction m between a and b in program order such that m has an address dependency on a

Memory Model Axioms

An execution of a RISC-V program obeys the RVWMO memory consistency model only if there exists a global memory order conforming to preserved program order and satisfying the *load value axiom*, the *atomicity axiom*, and the *progress axiom*.

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

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

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

The 原子性公理 *theoretically supports LR/SC pairs of different widths and to mismatched addresses, since implementations are permitted to allow SC operations to succeed in such cases. However, in practice, we expect such patterns to be rare, and their use is discouraged.*

Progress Axiom No memory operation may be preceded in the global memory order by an infinite sequence of other memory operations.

11.2 CSR Dependency Tracking Granularity

Name	Portions Tracked as Independent Units	Aliases
`fflags`	Bits 4, 3, 2, 1, 0	`fcsr`
`frm`	entire CSR	`fcsr`
`fcsr`	Bits 7-5, 4, 3, 2, 1, 0	`fflags`, `frm`

表 11.1: Granularities at which syntactic dependencies are tracked through CSRs

Note: read-only CSRs are not listed, as they do not participate in the definition of syntactic dependencies.

11.3 Source and Destination Register Listings

This section provides a concrete listing of the source and destination registers for each instruction. These listings are used in the definition of syntactic dependencies in Section 11.1.

The term “accumulating CSR” is used to describe a CSR that is both a source and a destination register, but which carries a dependency only from itself to itself.

Instructions carry a dependency from each source register in the “Source Registers” column to each destination register in the “Destination Registers” column, from each source register in the “Source Registers” column to each CSR in the “Accumulating CSRs” column, and from each CSR in the “Accumulating CSRs” column to itself, except where annotated otherwise.

Key:

^{A}Address source register

^{D}Data source register

†The instruction does not carry a dependency from any source register to any destination register

‡The instruction carries dependencies from source register(s) to destination register(s) as specified

RV32I Base Integer Instruction Set

	Source Registers	Destination Registers	Accumulating CSRs
LUI		*rd*	
AUIPC		*rd*	
JAL		*rd*	
JALR†	*rs1*	*rd*	
BEQ	*rs1*, *rs2*		
BNE	*rs1*, *rs2*		
BLT	*rs1*, *rs2*		
BGE	*rs1*, *rs2*		
BLTU	*rs1*, *rs2*		
BGEU	*rs1*, *rs2*		
LB†	$rs1^{A}$	*rd*	
LH†	$rs1^{A}$	*rd*	
LW†	$rs1^{A}$	*rd*	
LBU†	$rs1^{A}$	*rd*	
LHU†	$rs1^{A}$	*rd*	
SB	$rs1^{A}$, $rs2^{D}$		
SH	$rs1^{A}$, $rs2^{D}$		
SW	$rs1^{A}$, $rs2^{D}$		
ADDI	*rs1*	*rd*	
SLTI	*rs1*	*rd*	
SLTIU	*rs1*	*rd*	
XORI	*rs1*	*rd*	
ORI	*rs1*	*rd*	
ANDI	*rs1*	*rd*	
SLLI	*rs1*	*rd*	
SRLI	*rs1*	*rd*	
SRAI	*rs1*	*rd*	
ADD	*rs1*, *rs2*	*rd*	
SUB	*rs1*, *rs2*	*rd*	
SLL	*rs1*, *rs2*	*rd*	
SLT	*rs1*, *rs2*	*rd*	
SLTU	*rs1*, *rs2*	*rd*	
XOR	*rs1*, *rs2*	*rd*	
SRL	*rs1*, *rs2*	*rd*	
SRA	*rs1*, *rs2*	*rd*	
OR	*rs1*, *rs2*	*rd*	
AND	*rs1*, *rs2*	*rd*	
FENCE			

RV32I Base Integer Instruction Set (continued)

	Source Registers	Destination Registers	Accumulating CSRs	
CSRRW‡	*rs1*, csr^*	*rd*, *csr*		*unless *rd*=`x0`
CSRRS‡	*rs1*, *csr*	rd^*, *csr*		*unless *rs1*=`x0`
CSRRC‡	*rs1*, *csr*	rd^*, *csr*		*unless *rs1*=`x0`

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

RV32I Base Integer Instruction Set (continued)

	Source Registers	Destination Registers	Accumulating CSRs	
CSRRWI‡	csr^*	*rd*, *csr*		*unless *rd*=`x0`
CSRRSI‡	*csr*	*rd*, csr^*		*unless uimm[4:0]=0
CSRRCI‡	*csr*	*rd*, csr^*		*unless uimm[4:0]=0

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

RV64I Base Integer Instruction Set

	Source Registers	Destination Registers	Accumulating CSRs
LWU†	$rs1^A$	*rd*	
LD†	$rs1^A$	*rd*	
SD	$rs1^A$, $rs2^D$		
SLLI	*rs1*	*rd*	
SRLI	*rs1*	*rd*	
SRAI	*rs1*	*rd*	
ADDIW	*rs1*	*rd*	
SLLIW	*rs1*	*rd*	
SRLIW	*rs1*	*rd*	
SRAIW	*rs1*	*rd*	
ADDW	*rs1*, *rs2*	*rd*	
SUBW	*rs1*, *rs2*	*rd*	
SLLW	*rs1*, *rs2*	*rd*	
SRLW	*rs1*, *rs2*	*rd*	
SRAW	*rs1*, *rs2*	*rd*	

RV32M Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs
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 Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs
MULW	*rs1*, *rs2*	*rd*	
DIVW	*rs1*, *rs2*	*rd*	
DIVUW	*rs1*, *rs2*	*rd*	
REMW	*rs1*, *rs2*	*rd*	
REMUW	*rs1*, *rs2*	*rd*	

RV32A Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
LR.W†	$rs1^A$	*rd*		
SC.W†	$rs1^A$, $rs2^D$	*rd**		*if successful
AMOSWAP.W†	$rs1^A$, $rs2^D$	*rd*		
AMOADD.W†	$rs1^A$, $rs2^D$	*rd*		
AMOXOR.W†	$rs1^A$, $rs2^D$	*rd*		
AMOAND.W†	$rs1^A$, $rs2^D$	*rd*		
AMOOR.W†	$rs1^A$, $rs2^D$	*rd*		
AMOMIN.W†	$rs1^A$, $rs2^D$	*rd*		
AMOMAX.W†	$rs1^A$, $rs2^D$	*rd*		
AMOMINU.W†	$rs1^A$, $rs2^D$	*rd*		
AMOMAXU.W†	$rs1^A$, $rs2^D$	*rd*		

RV64A Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
LR.D†	$rs1^A$	rd		
SC.D†	$rs1^A$, $rs2^D$	rd^*		*if successful
AMOSWAP.D†	$rs1^A$, $rs2^D$	rd		
AMOADD.D†	$rs1^A$, $rs2^D$	rd		
AMOXOR.D†	$rs1^A$, $rs2^D$	rd		
AMOAND.D†	$rs1^A$, $rs2^D$	rd		
AMOOR.D†	$rs1^A$, $rs2^D$	rd		
AMOMIN.D†	$rs1^A$, $rs2^D$	rd		
AMOMAX.D†	$rs1^A$, $rs2^D$	rd		
AMOMINU.D†	$rs1^A$, $rs2^D$	rd		
AMOMAXU.D†	$rs1^A$, $rs2^D$	rd		

RV32F Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
FLW†	*rs1*^{A}	*rd*		
FSW	*rs1*^{A}, *rs2*^{D}			
FMADD.S	*rs1*, *rs2*, *rs3*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FMSUB.S	*rs1*, *rs2*, *rs3*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FNMSUB.S	*rs1*, *rs2*, *rs3*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FNMADD.S	*rs1*, *rs2*, *rs3*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FADD.S	*rs1*, *rs2*, frm*	*rd*	NV, OF, NX	*if rm=111
FSUB.S	*rs1*, *rs2*, frm*	*rd*	NV, OF, NX	*if rm=111
FMUL.S	*rs1*, *rs2*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FDIV.S	*rs1*, *rs2*, frm*	*rd*	NV, DZ, OF, UF, NX	*if rm=111
FSQRT.S	*rs1*, frm*	*rd*	NV, NX	*if 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	*if rm=111
FCVT.WU.S	*rs1*, frm*	*rd*	NV, NX	*if 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	*if rm=111
FCVT.S.WU	*rs1*, frm*	*rd*	NX	*if rm=111
FMV.W.X	*rs1*	*rd*		

RV64F Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
FCVT.L.S	*rs1*, frm*	*rd*	NV, NX	*if rm=111
FCVT.LU.S	*rs1*, frm*	*rd*	NV, NX	*if rm=111
FCVT.S.L	*rs1*, frm*	*rd*	NX	*if rm=111
FCVT.S.LU	*rs1*, frm*	*rd*	NX	*if rm=111

RV32D Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
FLD†	*rs1*^{A}	*rd*		
FSD	*rs1*^{A}, *rs2*^{D}			
FMADD.D	*rs1*, *rs2*, *rs3*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FMSUB.D	*rs1*, *rs2*, *rs3*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FNMSUB.D	*rs1*, *rs2*, *rs3*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FNMADD.D	*rs1*, *rs2*, *rs3*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FADD.D	*rs1*, *rs2*, frm*	*rd*	NV, OF, NX	*if rm=111
FSUB.D	*rs1*, *rs2*, frm*	*rd*	NV, OF, NX	*if rm=111
FMUL.D	*rs1*, *rs2*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FDIV.D	*rs1*, *rs2*, frm*	*rd*	NV, DZ, OF, UF, NX	*if rm=111
FSQRT.D	*rs1*, frm*	*rd*	NV, NX	*if rm=111
FSGNJ.D	*rs1*, *rs2*	*rd*		
FSGNJN.D	*rs1*, *rs2*	*rd*		
FSGNJX.D	*rs1*, *rs2*	*rd*		
FMIN.D	*rs1*, *rs2*	*rd*	NV	
FMAX.D	*rs1*, *rs2*	*rd*	NV	
FCVT.S.D	*rs1*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FCVT.D.S	*rs1*	*rd*	NV	
FEQ.D	*rs1*, *rs2*	*rd*	NV	
FLT.D	*rs1*, *rs2*	*rd*	NV	
FLE.D	*rs1*, *rs2*	*rd*	NV	
FCLASS.D	*rs1*	*rd*		
FCVT.W.D	*rs1*, frm*	*rd*	NV, NX	*if rm=111
FCVT.WU.D	*rs1*, frm*	*rd*	NV, NX	*if rm=111
FCVT.D.W	*rs1*	*rd*		
FCVT.D.WU	*rs1*	*rd*		

RV64D Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
FCVT.L.D	*rs1*, frm*	*rd*	NV, NX	*if rm=111
FCVT.LU.D	*rs1*, frm*	*rd*	NV, NX	*if rm=111
FMV.X.D	*rs1*	*rd*		
FCVT.D.L	*rs1*, frm*	*rd*	NX	*if rm=111
FCVT.D.LU	*rs1*, frm*	*rd*	NX	*if rm=111
FMV.D.X	*rs1*	*rd*		

附录 A RVWMO 说明材料（0.1 版本）

这节使用了更加非正式的语言和具体的例子，提供了更多关于 RVWMO（第 17 章 十一）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 17 章 十一和 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 A.7。任何其它的差异性都是无意的。

A.1 为什么用 RVWMO?

内存一致性模型遵循着从弱到强的松散谱系。弱内存模型允许更多的硬件实现的灵活性，并提供理论上比强模型更好的性能、每瓦特的性能、能量、可扩展性，和硬件验证开销，但代价是更复杂的编程模型。强模型提供了更简单的编程模型，但是对于可以在流水线和内存系统中执行的各种（非推测性的）硬件优化，要强加更多的约束开销，并且反过来在能量、区域开销和验证负担方面强加一些成本。

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

为了促进来自其它架构的代码的移植，一些硬件实现可以选择实现 Ztso 扩展，它默认提供了更严格的 RVTSO 次序的语义。为 RVWMO 编写的代码是与 RVTSO 自动且固有地兼容的，但是假定 RVTSO 写的代码不保证在 RVWMO 实现上能够正确地运行。事实上，大多数 RVWMO 实现都将（并且应当）简单地拒绝运行 RVTSO 专用的二进制文件。每个实现必须因此进行选择，或者优先兼容 RVTSO 代码（例如，为了便于来自 x86 的移植），或者反之优先兼容其他实现了 RVWMO 的 RISC-V 核。

在 RVTSO 下，代码中为 RVWMO 所写的一些屏障和/或内存次序注释可能变得冗余；在 Ztso 实现上默认采用 RVWMO 的代价是获取那些在实现上已经变成 no-op 的屏障(例如：FENCE R,RW

和 FENCE RW,W）的增量开销。然而，如果希望兼容非 Ztso 的实现，这些屏障在代码中仍然必须存在。

A.2 Litmus 测试

这章的解释使用了 *litmus* 测试，或者说，为测试或突出显示内存模型的一个特定部分而设计的小型程序。图 A.1显示了带有两个硬件线程的 litmus 测试的一个例子。作为对这个图和对本章中之后所有图的约定，我们假定 `s0` – `s2` 在所有硬件线程中都被预先设置为相同的值，并且 `s0` 持有由 `x` 标签的地址，`s1` 持有 `y` 的，而 `s2` 持有 `z` 的，这里 `x`、`y` 和 `z` 是对齐到 8 字节边界的不相交的内存位置。每张图在左侧显示了 litmus 测试的代码，在右侧则是一个特定的有效或无效执行的可视化。

```
         Hart 0           ||          Hart 1
-------------------------------------------------------
        ⋮                  ||          ⋮
        li t1,1            ||          li t4,4
(a)     sw t1,0(s0)        ||  (e)     sw t4,0(s0)
        ⋮                  ||          ⋮
        li t2,2            ||
(b)     sw t2,0(s0)        ||
        ⋮                  ||          ⋮
(c)     lw a0,0(s0)        ||
        ⋮                  ||          ⋮
        li t3,3            ||          li t5,5
(d)     sw t3,0(s0)        ||  (f)     sw t5,0(s0)
        ⋮                  ||          ⋮
```

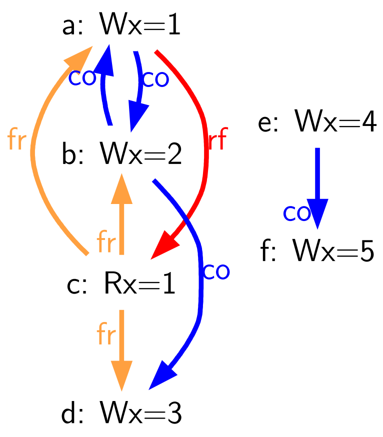


图 A.1: 一个 litmus 测试的示例和一个被禁止的执行（`a0=1`）。

Litmus 测试被用于理解特定具体情境中的内存模型的含义。例如，在图 A.1的 litmus 测试中，根据运行时来自各个硬件线程的指令流的动态交错情况，第一个硬件线程中的 `a0` 的最终的值可以是 2、4 或 5。然而，在这个例子中，硬件线程 0 中的 `a0` 的最终的值将永远都不会是 1 或 3；按直觉，值 1 在加载执行时将不再可见，而值 3 在加载执行时尚未成为可见的。我们下面来分析这个测试和一些其它的测试。

每个 litmus 测试的右侧显示的图展示了正在被考虑的特定执行候选的一个可视化的表示。这些图标使用在内存模型文献中常见的符号，来限制可能的全局内存次序（可能在执行中产生问题）的集合。它也是附录 B.2中展示的 herd 模型的基础。表 A.1中解释了该符号。在列出的关系中，在

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

表 A.1: 在这个附录中绘制的 litmus 测试图表的要点

硬件线程之间、co 边、fr 边和 ppo 边之间的 rf 边直接限制了全局内存次序（正如通过 ppo 也可以限制 fence、addr、data，和一些 ctrl）。其它边（例如 infa-hart rf 边）是信息性的，但是不会限制全局内存次序。

例如，在表 A.1中，a0=1 只可能发生在 (c) 读取由 (a) 写入的值、且下列情形之一为真的时候：

- 在全局内存次序中 (以及在一致性次序 co 中)，(b) 出现在 (a) 之前。然而这违反了 RVWMO PPO 规则 1。从 (b) 到 (a) 的 co 边突出了这一矛盾。
- 在全局内存次序中（以及在一致性次序 co 中），(a) 出现在 (b) 之前。然而，在这种情况中，加载值公理将被违反，因为在程序次序中，(a) 不是先于 (c) 的最近匹配的存储。从 (c) 到 (b) 的 fr 边突出了这一矛盾。

由于这些场景都不满足 RVWMO 公理，结果 a0=1 就被禁止了。

除了在这个附录中描述的内容，在https://github.com/litmus-tests/litmus-tests-riscv中还提供了一套超过七千个的石蕊测试。

litmus 测试项目也提供了关于如何在 RISC-V 硬件上运行 litmus 测试，和如何将结果与操作和公理模型进行比较的指令。

在未来，我们期望把这些关于内存模型的 litmus 测试也改编作为 RISC-V 一致性测试套件的一部分而使用。

A.3 RVWMO 规则的解释

在这节中，我们提供了对所有 RVWMO 规则和公理的解释和例子。

A.3.1 保留程序次序和全局内存次序

保留程序次序代表了必须在全局内存次序中被遵循的程序次序的子集。概念上，从其它硬件线程和/或观察者的角度，来自相同硬件线程的、按照保留程序次序被排序的事件，必须以该次序出现。

非正式地讲，全局内存次序代表了加载和存储所执行的次序。正式的内存模型文献已经从围绕执行概念构建的规范中移出，但是该思想对于建立非正式的直觉仍然是有用的。对于加载，当它的返回值被确定时，它被称为已执行的。对于存储，不是当它在流水线内部被执行时、而是只有当它的值已经被传播到全局可见的存储时，它才被称为已执行的。在这个意义上，全局内存次序也代表了一致性协议和/或余下的内存系统的贡献：把每个硬件线程发出的（可能被重新排序的）内存访问交错到所有硬件线程都赞成的单一的总次序之中。

加载执行的次序并不总是直接对应于那两个加载所返回的值的相对生存时间。特别地，对相同的地址，一个加载 b 可以在另一个加载 a 之前执行（例如，b 可以在 a 之前执行，并且在全局内存次序中，b 可以出现在 a 之前），但是尽管如此，a 可以返回一个比 b 更早旧的值。这种差异性（在其它事情之中）捕获了核心与内存之间安置的缓冲的重新排序效果。例如，b 可能已经返回了 a 存储在存储缓冲区中的一个值，同时 a 可能已经忽略了较新的存储，反而从内存中读取了一个较旧的值。为了解释这个情况，在每次加载执行的时候，它返回的值由加载值公理决定，而不只是通过确定在全局内存次序中最近对相同地址的存储来严格地决定，正如下面描述的那样。

A.3.2 加载值公理

> 加载值公理: 每个加载 i 的各个位所返回的值，由下列存储中在全局内存次序中最近的那个写到该位：
>
> 1. 写该位，并且在全局内存次序中先于 i 的存储
>
> 2. 写该位，并且在程序次序中先于 i 的存储

保留程序次序不需要遵循“在重叠的地址上，一个存储跟随着一个加载”的次序。这种复杂度的提升是因为，在几乎所有实现中存储缓冲区都是随处可见的。非正式地说，当存储仍然在存储缓冲区中的时候，加载可以通过从存储转发来执行（返回一个值），并因此出现在了存储自身的执行（写回到全局可见内存）之前。因此，任何其它的硬件线程将观察到，加载在存储之前执行。

考虑表 A.2的 litmus 测试。当在一个带有存储缓冲区（store buffers）的实现上运行这个程序时，它可能得到 a0=1, a1=0, a2=1, a3=0 的最终输出结果，如下：

- (a) 执行并进入第一个硬件线程的私有存储缓冲区（store buffer）

	Hart 0		Hart 1
	`li t1, 1`		`li t1, 1`
(a)	`sw t1,0(s0)`	(e)	`sw t1,0(s1)`
(b)	`lw a0,0(s0)`	(f)	`lw a2,0(s1)`
(c)	`fence r,r`	(g)	`fence r,r`
(d)	`lw a1,0(s1)`	(h)	`lw a3,0(s0)`

`Outcome: a0=1, a1=0, a2=1, a3=0`

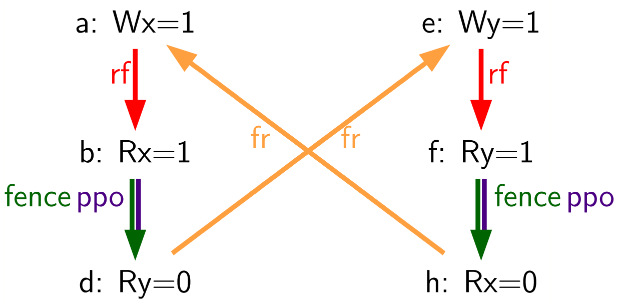


图 A.2: A store buffer forwarding litmus test (outcome permitted)

- (b) 执行并从存储缓冲区中的 (a) 转发它的返回值 1
- (c) 当所有之前的加载操作（例如，(b)）都已经完成时执行
- (d) 执行并从内存读取值 0
- (e) 执行并进入第二个硬件线程的私有存储缓冲区
- (f) 执行并从存储缓冲区中的 (e) 转发它的值 1
- (g) 从所有之前的加载操作（例如，(f)）都已经完成时执行
- (h) 执行并从内存读取值 0
- (a) 从第一个硬件线程的存储缓冲区排放到内存
- (e) 从第二个硬件线程的存储缓冲区排放到内存

因此，内存模型必须能够解释这种行为。

换句话说，假设保留程序次序确实包括了下列假定的规则：在保留的程序次序中，内存访问 a 先于内存访问 b（并因此也在全局内存次序中先于 b），如果在程序次序中 a 先于 b，并且 a 和 b 访问相同的内存位置，a 是一个写，而 b 是一个读。把这个称作“规则 X”。然后我们得到如下结果：

- (a) 先于 (b): 根据规则 X
- (b) 先于 (d): 根据规则 4
- (d) 先于 (e): 根据加载值公理。否则，如果 (e) 先于 (d)，那么将需要 (d) 返回值 1。(这是一个完全合法的执行；它只是并非问题所在)
- (e) 先于 (f): 根据规则 X

- (f) 先于 (h)：根据规则 4
- (h) 先于 (a)：根据加载值公理，同上。

全局内存次序必须是一个总次序，而不能有循环，因为循环将暗示该循环内的每个事件都发生在它自己之前，这是不可能的。因此，上面提出的执行将被禁止，并因此，规则 X 的添加将禁止带有存储缓冲区转发的实现，这显然是不可取的。

尽管如此，即使在全局内存次序中，(b) 先于 (a) 且/或 (f) 先于 (e)，这个例子中唯一合理的可能性也是，对于 (b)，返回由 (a) 所写的值，而 (f) 和 (e) 类似。这种情况的组合导致了加载值公理的定义中的第二个选项。即使在全局内存次序中，(b) 先于 (a)，由于在 (b) 执行的时候 (a) 还位于存储缓冲区中，(a) 将仍然对 (b) 可见。因此，即使在全局内存次序中 (b) 先于 (a)，(b) 也应当返回由 (a) 所写的值，因为在程序次序中 (a) 先于 (b)。对于 (e) 和 (f) 也类似。

```
          Hart 0                  Hart 1
          li t1, 1       ||       li t1, 1
(a)       sw t1,0(s0)    ||       LOOP:
(b)       fence w,w      ||  (d)  lw a0,0(s1)
(c)       sw t1,0(s1)    ||       beqz a0, LOOP
                         ||  (e)  sw t1,0(s2)
                         ||  (f)  lw a1,0(s2)
                         ||       xor a2,a1,a1
                         ||       add s0,s0,a2
                         ||  (g)  lw a2,0(s0)
             结果：a0=1, a1=1, a2=0
```

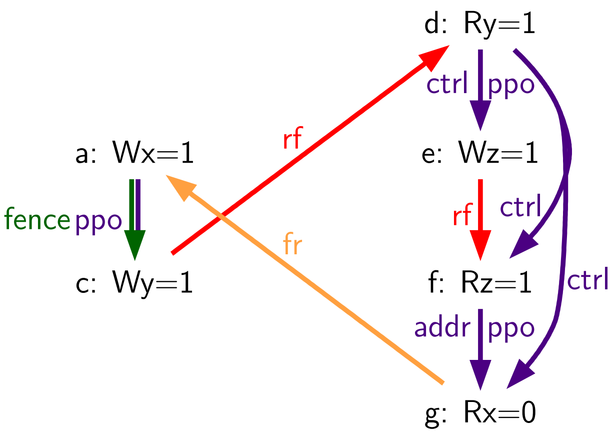


图 A.3: “PPOCA”存储缓冲区转发 litmus 测试（允许的输出结果）

在图 A.3中显示了另一个突出存储缓冲区行为的测试。在这个例子中，由于控制依赖，(d) 的次序排在 (e) 之前，而由于地址依赖，(f) 的次序排在 (g) 之前。然而，即使 (f) 返回了由 (e) 所写的值，(e) 的次序也并不需要排在 (f) 之前。这个可能对应到下列事件序列：

- (e) 推测地执行，并进入第二个硬件线程的私有存储缓冲区（但是没有排放到内存）
- (f) 推测地执行，并从存储缓冲区中的 (e) 转发它的值 1
- (g) 推测地执行，并从内存读取值 0
- (a) 执行，进入第一个硬件线程的私有存储缓冲区，并排放到内存

- (b) 执行，并退场
- (c) 执行，进入第一个硬件线程的私有存储缓冲区，并排放到内存
- (d) 执行，并从内存读取值 1
- (e), (f), 和 (g) 提交，因为推测是正确的
- (e) 从存储缓冲区排放到内存

A.3.3 原子性公理

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

RISC-V 架构把原子性的概念从排序的概念中解耦出来。不像诸如 TSO 的架构，RISC-V 在 RVWMO 下的原子性不会默认采用任何排序需求。排序的语义仅仅由 PPO 规则保证，否则就是适用的。

RISC-V contains two types of atomics: AMOs and LR/SC pairs. These conceptually behave differently, in the following way. LR/SC behave as if the old value is brought up to the core, modified, and written back to memory, all while a reservation is held on that memory location. AMOs on the other hand conceptually behave as if they are performed directly in memory. AMOs are therefore inherently atomic, while LR/SC pairs are atomic in the slightly different sense that the memory location in question will not be modified by another hart during the time the original hart holds the reservation.

```
(a) lr.d a0, 0(s0)     (a) lr.d a0, 0(s0)     (a) lr.w a0, 0(s0)     (a) lr.w a0, 0(s0)
(b) sd   t1, 0(s0)     (b) sw   t1, 4(s0)     (b) sw   t1, 4(s0)     (b) sw   t1, 4(s0)
(c) sc.d t2, 0(s0)     (c) sc.d t2, 0(s0)     (c) sc.w t2, 0(s0)     (c) sc.w t2, 8(s0)
```

图 A.4: In all four (independent) code snippets, the store-conditional (c) is permitted but not guaranteed to succeed

The atomicity axiom forbids stores from other harts from being interleaved in global memory order between an LR and the SC paired with that LR. The atomicity axiom does not forbid loads from being interleaved between the paired operations in program order or in the global memory order, nor does it forbid stores from the same hart or stores to non-overlapping locations from appearing between the paired operations in either program order or in the global memory order. For example, the SC instructions in Figure A.4 may (but are not guaranteed to) succeed. None of those successes would violate the atomicity axiom, because the intervening non-conditional stores are from the same hart as the paired load-reserved and store-conditional instructions. This way, a memory system that tracks memory accesses at cache line granularity (and which therefore will see the four snippets of Figure A.4 as identical) will not be forced to fail a store-conditional instruction that happens to (falsely) share another portion of the same cache line as the memory location being held by the reservation.

The atomicity axiom also technically supports cases in which the LR and SC touch different addresses and/or use different access sizes; however, use cases for such behaviors are expected to be rare in practice. Likewise, scenarios in which stores from the same hart between an LR/SC pair actually overlap the memory location(s) referenced by the LR or SC are expected to be rare compared to scenarios where the intervening store may simply fall onto the same cache line.

A.3.4 Progress Axiom

> Progress Axiom: No memory operation may be preceded in the global memory order by an infinite sequence of other memory operations.

The progress axiom ensures a minimal forward progress guarantee. It ensures that stores from one hart will eventually be made visible to other harts in the system in a finite amount of time, and that loads from other harts will eventually be able to read those values (or successors thereof). Without this rule, it would be legal, for example, for a spinlock to spin infinitely on a value, even with a store from another hart waiting to unlock the spinlock.

The progress axiom is intended not to impose any other notion of fairness, latency, or quality of service onto the harts in a RISC-V implementation. Any stronger notions of fairness are up to the rest of the ISA and/or up to the platform and/or device to define and implement.

The forward progress axiom will in almost all cases be naturally satisfied by any standard cache coherence protocol. Implementations with non-coherent caches may have to provide some other mechanism to ensure the eventual visibility of all stores (or successors thereof) to all harts.

A.3.5 Overlapping-Address Orderings (Rules 1–3)

> Rule 1: b is a store, and a and b access overlapping memory addresses
> Rule 2: a and b are loads, x is a byte read by both a and b, there is no store to x between a and b in program order, and a and b return values for x written by different memory operations
> Rule 3: a is generated by an AMO or SC instruction, b is a load, and b returns a value written by a

Same-address orderings where the latter is a store are straightforward: a load or store can never be reordered with a later store to an overlapping memory location. From a microarchitecture perspective, generally speaking, it is difficult or impossible to undo a speculatively reordered store if the speculation turns out to be invalid, so such behavior is simply disallowed by the model. Same-address orderings from a store to a later load, on the other hand, do not need to be enforced. As discussed in Section A.3.2, this reflects the observable behavior of implementations that forward values from buffered stores to later loads.

Same-address load-load ordering requirements are far more subtle. The basic requirement is that a younger load must not return a value that is older than a value returned by an older load in the same hart to the same address. This is often known as “CoRR” (Coherence for Read-Read pairs), or as part of a broader “coherence” or “sequential consistency per location” requirement. Some architectures in the past have relaxed same-address load-load ordering, but in hindsight this is generally considered to complicate the programming model too much, and so RVWMO requires CoRR ordering to be enforced. However, because the global memory order corresponds to the order in which loads perform rather than the ordering of the values being returned, capturing CoRR requirements in terms of the global memory order requires a bit of indirection.

Consider the litmus test of Figure A.5, which is one particular instance of the more general “fri-rfi” pattern. The term “fri-rfi” refers to the sequence (d), (e), (f): (d) “from-reads” (i.e., reads from an earlier write than) (e) which is the same hart, and (f) reads from (e) which is in the same hart.

From a microarchitectural perspective, outcome `a0=1`, `a1=2`, `a2=0` is legal (as are various other less subtle outcomes). Intuitively, the following would produce the outcome in question:

- (d) stalls (for whatever reason; perhaps it’s stalled waiting for some other preceding instruction)
- (e) executes and enters the store buffer (but does not yet drain to memory)

	Hart 0		Hart 1
	`li t1, 1`		`li  t2, 2`
(a)	`sw t1,0(s0)`	(d)	`lw  a0,0(s1)`
(b)	`fence w, w`	(e)	`sw  t2,0(s1)`
(c)	`sw t1,0(s1)`	(f)	`lw  a1,0(s1)`
		(g)	`xor t3,a1,a1`
		(h)	`add s0,s0,t3`
		(i)	`lw  a2,0(s0)`

`Outcome: a0=1, a1=2, a2=0`

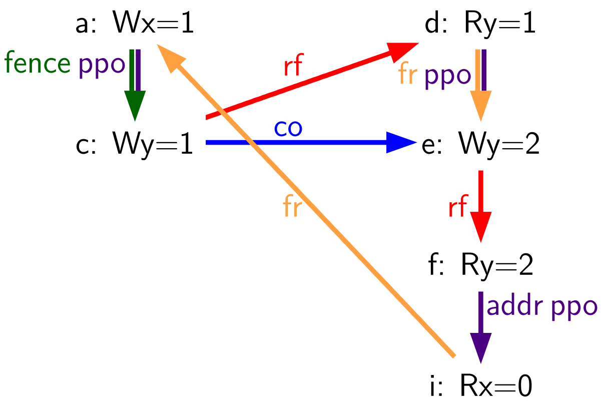


图 A.5: Litmus test MP+fence.w.w+fri-rfi-addr (outcome permitted)

- (f) executes and forwards from (e) in the store buffer
- (g), (h), and (i) execute
- (a) executes and drains to memory, (b) executes, and (c) executes and drains to memory
- (d) unstalls and executes
- (e) drains from the store buffer to memory

This corresponds to a global memory order of (f), (i), (a), (c), (d), (e). Note that even though (f) performs before (d), the value returned by (f) is newer than the value returned by (d). Therefore, this execution is legal and does not violate the CoRR requirements.

Likewise, if two back-to-back loads return the values written by the same store, then they may also appear out-of-order in the global memory order without violating CoRR. Note that this is not the same as saying that the two loads return the same value, since two different stores may write the same value.

Consider the litmus test of Figure A.6. The outcome `a0=1`, `a1=`v, `a2=`v, `a3=0` (where v is some value written by another hart) can be observed by allowing (g) and (h) to be reordered. This might be done speculatively, and the speculation can be justified by the microarchitecture (e.g., by snooping for cache invalidations and finding none) because replaying (h) after (g) would return the value written by the same store anyway. Hence assuming `a1` and `a2` would end up with the same value written by the same store anyway, (g) and (h) can be legally reordered. The global memory order corresponding to this execution would be (h),(k),(a),(c),(d),(g).

	Hart 0		Hart 1
	`li t1, 1`	(d)	`lw a0,0(s1)`
(a)	`sw t1,0(s0)`	(e)	`xor t2,a0,a0`
(b)	`fence w, w`	(f)	`add s4,s2,t2`
(c)	`sw t1,0(s1)`	(g)	`lw a1,0(s4)`
		(h)	`lw a2,0(s2)`
		(i)	`xor t3,a2,a2`
		(j)	`add s0,s0,t3`
		(k)	`lw a3,0(s0)`

Outcome: a0=1, a1=v, a2=v, a3=0

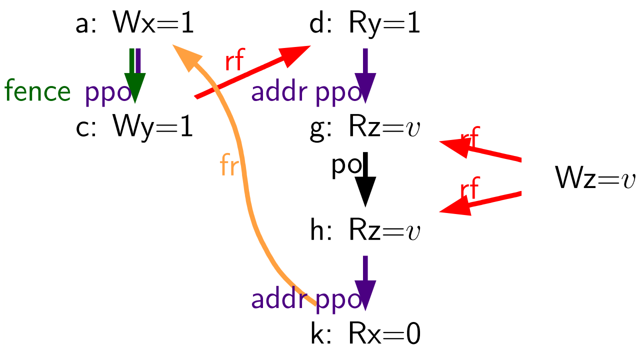


图 A.6: Litmus test RSW (outcome permitted)

Executions of the test in Figure A.6 in which `a1` does not equal `a2` do in fact require that (g) appears before (h) in the global memory order. Allowing (h) to appear before (g) in the global memory order would in that case result in a violation of CoRR, because then (h) would return an older value than that returned by (g). Therefore, PPO rule 2 forbids this CoRR violation from occurring. As such, PPO rule 2 strikes a careful balance between enforcing CoRR in all cases while simultaneously being weak enough to permit "RSW" and "fri-rfi" patterns that commonly appear in real microarchitectures.

There is one more overlapping-address rule: PPO rule 3 simply states that a value cannot be returned from an AMO or SC to a subsequent load until the AMO or SC has (in the case of the SC, successfully) performed globally. This follows somewhat naturally from the conceptual view that both AMOs and SC instructions are meant to be performed atomically in memory. However, notably, PPO rule 3 states that hardware may not even non-speculatively forward the value being stored by an AMOSWAP to a subsequent load, even though for AMOSWAP that store value is not actually semantically dependent on the previous value in memory, as is the case for the other AMOs. The same holds true even when forwarding from SC store values that are not semantically dependent on the value returned by the paired LR.

The three PPO rules above also apply when the memory accesses in question only overlap partially. This can occur, for example, when accesses of different sizes are used to access the same object. Note also that the base addresses of two overlapping memory operations need not necessarily be the same for two memory accesses to overlap. When misaligned memory accesses are being used, the overlapping-address PPO rules apply to each of the component memory accesses independently.

A.3.6 Fences (Rule 4)

> Rule 4: There is a FENCE instruction that orders *a* before *b*

By default, the FENCE instruction ensures that all memory accesses from instructions preceding the fence in program order (the “predecessor set”) appear earlier in the global memory order than memory accesses from instructions appearing after the fence in program order (the “successor set”). However, fences can optionally further restrict the predecessor set and/or the successor set to a smaller set of memory accesses in order to provide some speedup. Specifically, fences have PR, PW, SR, and SW bits which restrict the predecessor and/or successor sets. The predecessor set includes loads (resp. stores) if and only if PR (resp. PW) is set. Similarly, the successor set includes loads (resp. stores) if and only if SR (resp. SW) is set.

The FENCE encoding currently has nine non-trivial combinations of the four bits PR, PW, SR, and SW, plus one extra encoding FENCE.TSO which facilitates mapping of “acquire+release” or RVTSO semantics. The remaining seven combinations have empty predecessor and/or successor sets and hence are no-ops. Of the ten non-trivial options, only six are commonly used in practice:

- FENCE RW,RW
- FENCE.TSO
- FENCE RW,W
- FENCE R,RW
- FENCE R,R
- FENCE W,W

FENCE instructions using any other combination of PR, PW, SR, and SW are reserved. We strongly recommend that programmers stick to these six. Other combinations may have unknown or unexpected interactions with the memory model.

Finally, we note that since RISC-V uses a multi-copy atomic memory model, programmers can reason about fences bits in a thread-local manner. There is no complex notion of “fence cumulativity” as found in memory models that are not multi-copy atomic.

A.3.7 Explicit Synchronization (Rules 5–8)

> Rule 5: *a* has an acquire annotation
> Rule 6: *b* has a release annotation
> Rule 7: *a* and *b* both have RCsc annotations
> Rule 8: *a* is paired with *b*

An *acquire* operation, as would be used at the start of a critical section, requires all memory operations following the acquire in program order to also follow the acquire in the global memory order. This ensures, for example, that all loads and stores inside the critical section are up to date with respect to the synchronization variable being used to protect it. Acquire ordering can be enforced in one of two ways: with an acquire annotation, which enforces ordering with respect to just the synchronization variable itself, or with a FENCE R,RW, which enforces ordering with respect to all previous loads.

```
    sd           x1, (a1)     # Arbitrary unrelated store
    ld           x2, (a2)     # Arbitrary unrelated load
    li           t0, 1        # Initialize swap value.
again:
    amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
    bnez         t0, again    # Retry if held.
    # ...
    # Critical section.
    # ...
    amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
    sd           x3, (a3)     # Arbitrary unrelated store
    ld           x4, (a4)     # Arbitrary unrelated load
```

图 A.7: A spinlock with atomics

Consider Figure A.7. Because this example uses *aq*, the loads and stores in the critical section are guaranteed to appear in the global memory order after the AMOSWAP used to acquire the lock. However, assuming `a0`, `a1`, and `a2` point to different memory locations, the loads and stores in the critical section may or may not appear after the “Arbitrary unrelated load” at the beginning of the example in the global memory order.

Now, consider the alternative in Figure A.8. In this case, even though the AMOSWAP does not enforce ordering with an *aq* bit, the fence nevertheless enforces that the acquire AMOSWAP appears earlier in the global memory order than all loads and stores in the critical section. Note,

```
        sd           x1, (a1)     # Arbitrary unrelated store
        ld           x2, (a2)     # Arbitrary unrelated load
        li           t0, 1        # Initialize swap value.
    again:
        amoswap.w    t0, t0, (a0) # Attempt to acquire lock.
        fence        r, rw        # Enforce "acquire" memory ordering
        bnez         t0, again    # Retry if held.
        # ...
        # Critical section.
        # ...
        fence        rw, w        # Enforce "release" memory ordering
        amoswap.w    x0, x0, (a0) # Release lock by storing 0.
        sd           x3, (a3)     # Arbitrary unrelated store
        ld           x4, (a4)     # Arbitrary unrelated load
```

图 A.8: A spinlock with fences

however, that in this case, the fence also enforces additional orderings: it also requires that the "Arbitrary unrelated load" at the start of the program appears earlier in the global memory order than the loads and stores of the critical section. (This particular fence does not, however, enforce any ordering with respect to the "Arbitrary unrelated store" at the start of the snippet.) In this way, fence-enforced orderings are slightly coarser than orderings enforced by *.aq*.

Release orderings work exactly the same as acquire orderings, just in the opposite direction. Release semantics require all loads and stores preceding the release operation in program order to also precede the release operation in the global memory order. This ensures, for example, that memory accesses in a critical section appear before the lock-releasing store in the global memory order. Just as for acquire semantics, release semantics can be enforced using release annotations or with a FENCE RW,W operation. Using the same examples, the ordering between the loads and stores in the critical section and the "Arbitrary unrelated store" at the end of the code snippet is enforced only by the FENCE RW,W in Figure A.8, not by the *rl* in Figure A.7.

With RCpc annotations alone, store-release-to-load-acquire ordering is not enforced. This facilitates the porting of code written under the TSO and/or RCpc memory models. To enforce store-release-to-load-acquire ordering, the code must use store-release-RCsc and load-acquire-RCsc operations so that PPO rule 7 applies. RCpc alone is sufficient for many use cases in C/C++ but is insufficient for many other use cases in C/C++, Java, and Linux, to name just a few examples; see Section A.5 for details.

PPO rule 8 indicates that an SC must appear after its paired LR in the global memory order. This will follow naturally from the common use of LR/SC to perform an atomic read-modify-write operation due to the inherent data dependency. However, PPO rule 8 also applies even when the value being stored does not syntactically depend on the value returned by the paired LR.

Lastly, we note that just as with fences, programmers need not worry about "cumulativity" when analyzing ordering annotations.

A.3.8 Syntactic Dependencies (Rules 9–11)

> Rule 9: *b* has a syntactic address dependency on *a*
> Rule 10: *b* has a syntactic data dependency on *a*
> Rule 11: *b* is a store, and *b* has a syntactic control dependency on *a*

Dependencies from a load to a later memory operation in the same hart are respected by the RVWMO memory model. The Alpha memory model was notable for choosing *not* to enforce the ordering of such dependencies, but most modern hardware and software memory models consider allowing dependent instructions to be reordered too confusing and counterintuitive. Furthermore, modern code sometimes intentionally uses such dependencies as a particularly lightweight ordering enforcement mechanism.

The terms in Section 11.1 work as follows. Instructions are said to carry dependencies from their source register(s) to their destination register(s) whenever the value written into each destination register is a function of the source register(s). For most instructions, this means that the destination register(s) carry a dependency from all source register(s). However, there are a few notable exceptions. In the case of memory instructions, the value written into the destination register ultimately comes from the memory system rather than from the source register(s) directly, and so this breaks the chain of dependencies carried from the source register(s). In the case of unconditional jumps, the value written into the destination register comes from the current `pc` (which is never considered a source register by the memory model), and so likewise, JALR (the only jump with a source register) does not carry a dependency from *rs1* to *rd*.

```
(a) fadd  f3,f1,f2
(b) fadd  f6,f4,f5
(c) csrrs a0,fflags,x0
```

图 A.9: (c) has a syntactic dependency on both (a) and (b) via `fflags`, a destination register that both (a) and (b) implicitly accumulate into

The notion of accumulating into a destination register rather than writing into it reflects the behavior of CSRs such as `fflags`. In particular, an accumulation into a register does not clobber any previous writes or accumulations into the same register. For example, in Figure A.9, (c) has a syntactic dependency on both (a) and (b).

Like other modern memory models, the RVWMO memory model uses syntactic rather than semantic dependencies. In other words, this definition depends on the identities of the registers being accessed by different instructions, not the actual contents of those registers. This means that an address, control, or data dependency must be enforced even if the calculation could seemingly be "optimized away". This choice ensures that RVWMO remains compatible with code that uses these false syntactic dependencies as a lightweight ordering mechanism.

```
ld  a1,0(s0)
xor a2,a1,a1
add s1,s1,a2
ld  a5,0(s1)
```

图 A.10: A syntactic address dependency

For example, there is a syntactic address dependency from the memory operation generated by the first instruction to the memory operation generated by the last instruction in Figure A.10, even though `a1` XOR `a1` is zero and hence has no effect on the address accessed by the second load.

The benefit of using dependencies as a lightweight synchronization mechanism is that the ordering enforcement requirement is limited only to the specific two instructions in question. Other non-dependent instructions may be freely reordered by aggressive implementations. One alternative would be to use a load-acquire, but this would enforce ordering for the first load with respect to *all* subsequent instructions. Another would be to use a FENCE R,R, but this would include all previous and all subsequent loads, making this option more expensive.

```
      lw  x1,0(x2)
      bne x1,x0,next
      sw  x3,0(x4)
next: sw  x5,0(x6)
```

图 A.11: A syntactic control dependency

Control dependencies behave differently from address and data dependencies in the sense that a control dependency always extends to all instructions following the original target in program order. Consider Figure A.11: the instruction at `next` will always execute, but the memory operation generated by that last instruction nevertheless still has a control dependency from the memory operation generated by the first instruction.

```
      lw  x1,0(x2)
      bne x1,x0,next
next: sw  x3,0(x4)
```

图 A.12: Another syntactic control dependency

Likewise, consider Figure A.12. Even though both branch outcomes have the same target, there is still a control dependency from the memory operation generated by the first instruction in this snippet to the memory operation generated by the last instruction. This definition of control dependency is subtly stronger than what might be seen in other contexts (e.g., C++), but it conforms with standard definitions of control dependencies in the literature.

Notably, PPO rules 9–11 are also intentionally designed to respect dependencies that originate from the output of a successful store-conditional instruction. Typically, an SC instruction will be followed by a conditional branch checking whether the outcome was successful; this implies that there will be a control dependency from the store operation generated by the SC instruction to any memory operations following the branch. PPO rule 11 in turn implies that any subsequent store operations will appear later in the global memory order than the store operation generated by the SC. However, since control, address, and data dependencies are defined over memory operations, and since an unsuccessful SC does not generate a memory operation, no order is enforced between unsuccessful SC and its dependent instructions. Moreover, since SC is defined to carry dependencies from its source registers to *rd* only when the SC is successful, an unsuccessful SC has no effect on the global memory order.

`Initial values: 0(s0)=1; 0(s2)=1`

Hart 0		Hart 1	
(a)	ld a0,0(s0)	(e)	ld a3,0(s2)
(b)	lr a1,0(s1)	(f)	sd a3,0(s0)
(c)	sc a2,a0,0(s1)		
(d)	sd a2,0(s2)		

`Outcome: a0=0, a3=0`

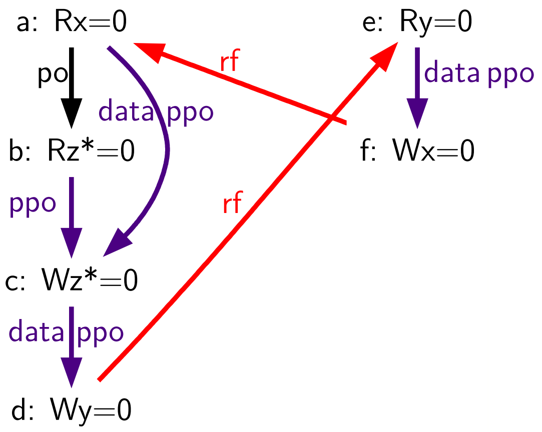


图 A.13: A variant of the LB litmus test (outcome forbidden)

In addition, the choice to respect dependencies originating at store-conditional instructions ensures that certain out-of-thin-air-like behaviors will be prevented. Consider Figure A.13. Suppose a hypothetical implementation could occasionally make some early guarantee that a store-conditional operation will succeed. In this case, (c) could return 0 to `a2` early (before actually executing), allowing the sequence (d), (e), (f), (a), and then (b) to execute, and then (c) might execute (successfully) only at that point. This would imply that (c) writes its own success value to `0(s1)`! Fortunately, this situation and others like it are prevented by the fact that RVWMO respects dependencies originating at the stores generated by successful SC instructions.

We also note that syntactic dependencies between instructions only have any force when they take the form of a syntactic address, control, and/or data dependency. For example: a syntactic dependency between two "F" instructions via one of the "accumulating CSRs" in Section 11.3 does *not* imply that the two "F" instructions must be executed in order. Such a dependency would only serve to ultimately set up later a dependency from both "F" instructions to a later CSR instruction accessing the CSR flag in question.

A.3.9 Pipeline Dependencies (Rules 12–13)

> Rule 12: b is a load, and there exists some store m between a and b in program order such that m has an address or data dependency on a, and b returns a value written by m
> Rule 13: b is a store, and there exists some instruction m between a and b in program order such that m has an address dependency on a

	Hart 0		Hart 1
	`li t1, 1`	(d)	`lw a0, 0(s1)`
(a)	`sw t1,0(s0)`	(e)	`sw a0, 0(s2)`
(b)	`fence w, w`	(f)	`lw a1, 0(s2)`
(c)	`sw t1,0(s1)`		`xor a2,a1,a1`
			`add s0,s0,a2`
		(g)	`lw a3,0(s0)`

`Outcome: a0=1, a3=0`

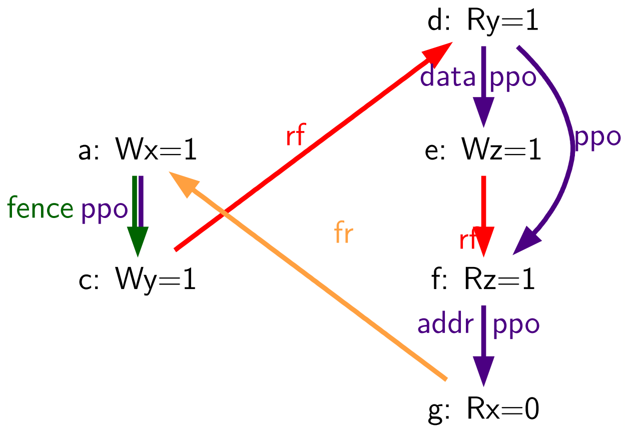


图 A.14: Because of PPO rule 12 and the data dependency from (d) to (e), (d) must also precede (f) in the global memory order (outcome forbidden)

PPO rules 12 and 13 reflect behaviors of almost all real processor pipeline implementations. Rule 12 states that a load cannot forward from a store until the address and data for that store are known. Consider Figure A.14: (f) cannot be executed until the data for (e) has been resolved, because (f) must return the value written by (e) (or by something even later in the global memory order), and the old value must not be clobbered by the writeback of (e) before (d) has had a chance to perform. Therefore, (f) will never perform before (d) has performed.

	Hart 0		Hart 1
	`li t1, 1`		`li t1, 1`
(a)	`sw t1,0(s0)`	(d)	`lw a0, 0(s1)`
(b)	`fence w, w`	(e)	`sw a0, 0(s2)`
(c)	`sw t1,0(s1)`	(f)	`sw t1, 0(s2)`
		(g)	`lw a1, 0(s2)`
			`xor a2,a1,a1`
			`add s0,s0,a2`
		(h)	`lw a3,0(s0)`

`Outcome: a0=1, a3=0`

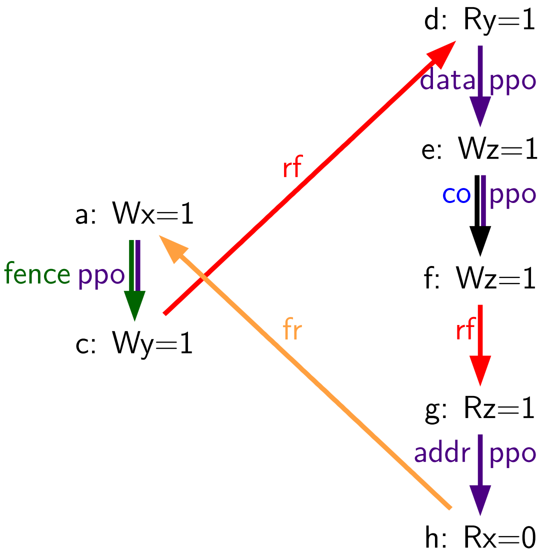


图 A.15: Because of the extra store between (e) and (g), (d) no longer necessarily precedes (g) (outcome permitted)

If there were another store to the same address in between (e) and (f), as in Figure A.15, then (f) would no longer be dependent on the data of (e) being resolved, and hence the dependency of (f) on (d), which produces the data for (e), would be broken.

Rule 13 makes a similar observation to the previous rule: a store cannot be performed at memory until all previous loads that might access the same address have themselves been performed. Such a load must appear to execute before the store, but it cannot do so if the store were to overwrite the value in memory before the load had a chance to read the old value. Likewise, a store generally cannot be performed until it is known that preceding instructions will not cause an exception due to failed address resolution, and in this sense, rule 13 can be seen as somewhat of a special case of rule 11.

Consider Figure A.16: (f) cannot be executed until the address for (e) is resolved, because it may turn out that the addresses match; i.e., that `a1=s0`. Therefore, (f) cannot be sent to memory before (d) has executed and confirmed whether the addresses do indeed overlap.

Hart 0		Hart 1	
			li t1, 1
(a)	lw a0,0(s0)	(d)	lw a1, 0(s1)
(b)	fence rw,rw	(e)	lw a2, 0(a1)
(c)	sw s2,0(s1)	(f)	sw t1, 0(s0)

Outcome: a0=1, a1=t

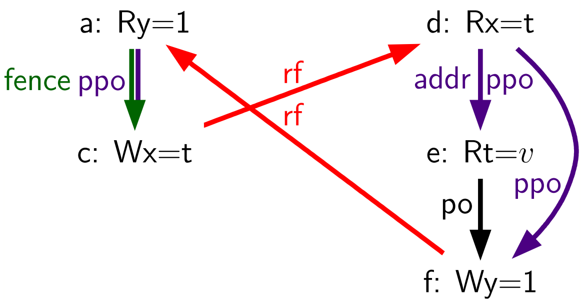


图 A.16: Because of the address dependency from (d) to (e), (d) also precedes (f) (outcome forbidden)

A.4 Beyond Main Memory

RVWMO does not currently attempt to formally describe how FENCE.I, SFENCE.VMA, I/O fences, and PMAs behave. All of these behaviors will be described by future formalizations. In the meantime, the behavior of FENCE.I is described in Chapter 三, the behavior of SFENCE.VMA is described in the RISC-V Instruction Set Privileged Architecture Manual, and the behavior of I/O fences and the effects of PMAs are described below.

A.4.1 Coherence and Cacheability

The RISC-V Privileged ISA defines Physical Memory Attributes (PMAs) which specify, among other things, whether portions of the address space are coherent and/or cacheable. See the RISC-V Privileged ISA Specification for the complete details. Here, we simply discuss how the various details in each PMA relate to the memory model:

- Main memory vs. I/O, and I/O memory ordering PMAs: the memory model as defined applies to main memory regions. I/O ordering is discussed below.
- Supported access types and atomicity PMAs: the memory model is simply applied on top of whatever primitives each region supports.
- Cacheability PMAs: the cacheability PMAs in general do not affect the memory model. Non-cacheable regions may have more restrictive behavior than cacheable regions, but the set of allowed behaviors does not change regardless. However, some platform-specific and/or device-specific cacheability settings may differ.

- Coherence PMAs: The memory consistency model for memory regions marked as non-coherent in PMAs is currently platform-specific and/or device-specific: the load-value axiom, the atomicity axiom, and the progress axiom all may be violated with non-coherent memory. Note however that coherent memory does not require a hardware cache coherence protocol. The RISC-V Privileged ISA Specification suggests that hardware-incoherent regions of main memory are discouraged, but the memory model is compatible with hardware coherence, software coherence, implicit coherence due to read-only memory, implicit coherence due to only one agent having access, or otherwise.

- Idempotency PMAs: Idempotency PMAs are used to specify memory regions for which loads and/or stores may have side effects, and this in turn is used by the microarchitecture to determine, e.g., whether prefetches are legal. This distinction does not affect the memory model.

A.4.2 I/O Ordering

For I/O, the load value axiom and atomicity axiom in general do not apply, as both reads and writes might have device-specific side effects and may return values other than the value "written" by the most recent store to the same address. Nevertheless, the following preserved program order rules still generally apply for accesses to I/O memory: memory access a precedes memory access b in global memory order if a precedes b in program order and one or more of the following holds:

1. a precedes b in preserved program order as defined in Chapter 十一, with the exception that acquire and release ordering annotations apply only from one memory operation to another memory operation and from one I/O operation to another I/O operation, but not from a memory operation to an I/O nor vice versa

2. a and b are accesses to overlapping addresses in an I/O region

3. a and b are accesses to the same strongly ordered I/O region

4. a and b are accesses to I/O regions, and the channel associated with the I/O region accessed by either a or b is channel 1

5. a and b are accesses to I/O regions associated with the same channel (except for channel 0)

Note that the FENCE instruction distinguishes between main memory operations and I/O operations in its predecessor and successor sets. To enforce ordering between I/O operations and main memory operations, code must use a FENCE with PI, PO, SI, and/or SO, plus PR, PW, SR, and/or SW. For example, to enforce ordering between a write to main memory and an I/O write to a device register, a FENCE W,O or stronger is needed.

```
sd t0, 0(a0)
fence w,o
sd a0, 0(a1)
```

图 A.17: Ordering memory and I/O accesses

When a fence is in fact used, implementations must assume that the device may attempt to access memory immediately after receiving the MMIO signal, and subsequent memory accesses from that device to memory must observe the effects of all accesses ordered prior to that MMIO operation. In other words, in Figure A.17, suppose `0(a0)` is in main memory and `0(a1)` is the address of a device register in I/O memory. If the device accesses `0(a0)` upon receiving the MMIO write, then that load must conceptually appear after the first store to `0(a0)` according to the rules of the RVWMO memory model. In some implementations, the only way to ensure this will be to require that the first store does in fact complete before the MMIO write is issued. Other implementations may find ways to be more aggressive, while others still may not need to do anything different at all for I/O and main memory accesses. Nevertheless, the RVWMO memory model does not distinguish between these options; it simply provides an implementation-agnostic mechanism to specify the orderings that must be enforced.

Many architectures include separate notions of "ordering" and "completion" fences, especially as it relates to I/O (as opposed to regular main memory). Ordering fences simply ensure that memory operations stay in order, while completion fences ensure that predecessor accesses have all completed before any successors are made visible. RISC-V does not explicitly distinguish between ordering and completion fences. Instead, this distinction is simply inferred from different uses of the FENCE bits.

For implementations that conform to the RISC-V Unix Platform Specification, I/O devices and DMA operations are required to access memory coherently and via strongly ordered I/O channels. Therefore, accesses to regular main memory regions that are concurrently accessed by external devices can also use the standard synchronization mechanisms. Implementations that do not conform to the Unix Platform Specification and/or in which devices do not access memory coherently will need to use mechanisms (which are currently platform-specific or device-specific) to enforce coherency.

I/O regions in the address space should be considered non-cacheable regions in the PMAs for those regions. Such regions can be considered coherent by the PMA if they are not cached by any agent.

The ordering guarantees in this section may not apply beyond a platform-specific boundary between the RISC-V cores and the device. In particular, I/O accesses sent across an external bus (e.g., PCIe) may be reordered before they reach their ultimate destination. Ordering must be enforced in such situations according to the platform-specific rules of those external devices and buses.

A.5 Code Porting and Mapping Guidelines

x86/TSO Operation	RVWMO Mapping
Load	`l{b|h|w|d}; fence r,rw`
Store	`fence rw,w; s{b|h|w|d}`
Atomic RMW	`amo<op>.{w|d}.aqrl` OR `loop: lr.{w|d}.aq; <op>; sc.{w|d}.aqrl; bnez loop`
Fence	`fence rw,rw`

表 A.2: Mappings from TSO operations to RISC-V operations

Table A.2 provides a mapping from TSO memory operations onto RISC-V memory instructions. Normal x86 loads and stores are all inherently acquire-RCpc and release-RCpc operations: TSO enforces all load-load, load-store, and store-store ordering by default. Therefore, under RVWMO, all TSO loads must be mapped onto a load followed by FENCE R,RW, and all TSO stores must be mapped onto FENCE RW,W followed by a store. TSO atomic read-modify-writes and x86 instructions using the LOCK prefix are fully ordered and can be implemented either via an AMO with both *aq* and *rl* set, or via an LR with *aq* set, the arithmetic operation in question, an SC with both *aq* and *rl* set, and a conditional branch checking the success condition. In the latter case, the *rl* annotation on the LR turns out (for non-obvious reasons) to be redundant and can be omitted.

Alternatives to Table A.2 are also possible. A TSO store can be mapped onto AMOSWAP with *rl* set. However, since RVWMO PPO Rule 3 forbids forwarding of values from AMOs to subsequent loads, the use of AMOSWAP for stores may negatively affect performance. A TSO load can be mapped using LR with *aq* set: all such LR instructions will be unpaired, but that fact in and of itself does not preclude the use of LR for loads. However, again, this mapping may also negatively affect performance if it puts more pressure on the reservation mechanism than was originally intended.

Table A.3 provides a mapping from Power memory operations onto RISC-V memory instructions. Power ISYNC maps on RISC-V to a FENCE.I followed by a FENCE R,R; the latter fence is

Power Operation	RVWMO Mapping
Load	`l{b|h|w|d}`
Load-Reserve	`lr.{w|d}`
Store	`s{b|h|w|d}`
Store-Conditional	`sc.{w|d}`
`lwsync`	`fence.tso`
`sync`	`fence rw,rw`
`isync`	`fence.i; fence r,r`

表 A.3: Mappings from Power operations to RISC-V operations

needed because ISYNC is used to define a "control+control fence" dependency that is not present in RVWMO.

ARM Operation	RVWMO Mapping
Load	`l{b|h|w|d}`
Load-Acquire	`fence rw, rw; l{b|h|w|d}; fence r,rw`
Load-Exclusive	`lr.{w|d}`
Load-Acquire-Exclusive	`lr.{w|d}.aqrl`
Store	`s{b|h|w|d}`
Store-Release	`fence rw,w; s{b|h|w|d}`
Store-Exclusive	`sc.{w|d}`
Store-Release-Exclusive	`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: Mappings from ARM operations to RISC-V operations

Table A.4 provides a mapping from ARM memory operations onto RISC-V memory instructions. Since RISC-V does not currently have plain load and store opcodes with *aq* or *rl* annotations, ARM load-acquire and store-release operations should be mapped using fences instead. Furthermore, in order to enforce store-release-to-load-acquire ordering, there must be a FENCE RW,RW between the store-release and load-acquire; Table A.4 enforces this by always placing the fence in front of each acquire operation. ARM load-exclusive and store-exclusive instructions can likewise map onto their RISC-V LR and SC equivalents, but instead of placing a FENCE RW,RW in front of an LR with *aq* set, we simply also set *rl* instead. ARM ISB maps on RISC-V to FENCE.I followed by FENCE R,R similarly to how ISYNC maps for Power.

Linux Operation	RVWMO Mapping
`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 Construct	RVWMO AMO Mapping
`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 Construct	RVWMO LR/SC Mapping
`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: Mappings from Linux memory primitives to RISC-V primitives. Other constructs (such as spinlocks) should follow accordingly. Platforms or devices with non-coherent DMA may need additional synchronization (such as cache flush or invalidate mechanisms); currently any such extra synchronization will be device-specific.

Table A.5 provides a mapping of Linux memory ordering macros onto RISC-V memory instructions. The Linux fences `dma_rmb()` and `dma_wmb()` map onto FENCE R,R and FENCE W,W, respectively, since the RISC-V Unix Platform requires coherent DMA, but would be mapped onto FENCE RI,RI and FENCE WO,WO, respectively, on a platform with non-coherent DMA. Platforms with non-coherent DMA may also require a mechanism by which cache lines can be flushed and/or invalidated. Such mechanisms will be device-specific and/or standardized in a future extension to the ISA.

The Linux mappings for release operations may seem stronger than necessary, but these mappings are needed to cover some cases in which Linux requires stronger orderings than the more intuitive mappings would provide. In particular, as of the time this text is being written, Linux is actively debating whether to require load-load, load-store, and store-store orderings between accesses in one critical section and accesses in a subsequent critical section in the same hart and protected by the same synchronization object. Not all combinations of FENCE RW,W/FENCE R,RW mappings with *aq*/*rl* mappings combine to provide such orderings. There are a few ways around this problem, including:

1. Always use FENCE RW,W/FENCE R,RW, and never use *aq*/*rl*. This suffices but is undesirable, as it defeats the purpose of the *aq*/*rl* modifiers.
2. Always use *aq*/*rl*, and never use FENCE RW,W/FENCE R,RW. This does not currently work due to the lack of load and store opcodes with *aq* and *rl* modifiers.
3. Strengthen the mappings of release operations such that they would enforce sufficient orderings in the presence of either type of acquire mapping. This is the currently recommended solution, and the one shown in Table A.5.

```
                              RVWMO Mapping:
                              (a) lw           a0, 0(s0)
Linux code:                   (b) fence.tso  // vs. fence rw,w
(a)  int r0 = *x;             (c) sd           x0,0(s1)
(bc) spin_unlock(y, 0);           ...
     ...                          loop:
     ...                      (d) amoswap.d.aq a1,t1,0(s1)
(d)  spin_lock(y);                bnez         a1,loop
(e)  int r1 = *z;             (e) lw           a2,0(s2)
```

图 A.18: Orderings between critical sections in Linux

For example, the critical section ordering rule currently being debated by the Linux community would require (a) to be ordered before (e) in Figure A.18. If that will indeed be required, then it would be insufficient for (b) to map as FENCE RW,W. That said, these mappings are subject to change as the Linux Kernel Memory Model evolves.

Table A.6 provides a mapping of C11/C++11 atomic operations onto RISC-V memory instructions. If load and store opcodes with *aq* and *rl* modifiers are introduced, then the map-

C/C++ Construct	RVWMO Mapping
Non-atomic load	`l{b|h|w|d}`
`atomic_load(memory_order_relaxed)`	`l{b|h|w|d}`
`atomic_load(memory_order_acquire)`	`l{b|h|w|d}; fence r,rw`
`atomic_load(memory_order_seq_cst)`	`fence rw,rw; l{b|h|w|d}; fence r,rw`
Non-atomic store	`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++ Construct	RVWMO AMO Mapping
`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++ Construct	RVWMO LR/SC Mapping
`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: Mappings from C/C++ primitives to RISC-V primitives.

pings in Table A.7 will suffice. Note however that the two mappings only interoperate correctly if `atomic_<op>(memory_order_seq_cst)` is mapped using an LR that has both *aq* and *rl* set.

C/C++ Construct	RVWMO Mapping
Non-atomic load	l{b|h|w|d}
atomic_load(memory_order_relaxed)	l{b|h|w|d}
atomic_load(memory_order_acquire)	l{b|h|w|d}.aq
atomic_load(memory_order_seq_cst)	l{b|h|w|d}.aq
Non-atomic store	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++ Construct	RVWMO AMO Mapping
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++ Construct	RVWMO LR/SC Mapping
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

*must be lr.{w|d}.aqrl in order to interoperate with code mapped per Table A.6

表 A.7: Hypothetical mappings from C/C++ primitives to RISC-V primitives, if native load-acquire and store-release opcodes are introduced.

Any AMO can be emulated by an LR/SC pair, but care must be taken to ensure that any PPO orderings that originate from the LR are also made to originate from the SC, and that any PPO orderings that terminate at the SC are also made to terminate at the LR. For example, the LR must also be made to respect any data dependencies that the AMO has, given that load operations do not otherwise have any notion of a data dependency. Likewise, the effect a FENCE R,R elsewhere in the same hart must also be made to apply to the SC, which would not otherwise respect that fence.

The emulator may achieve this effect by simply mapping AMOs onto `lr.aq; <op>; sc.aqrl`, matching the mapping used elsewhere for fully ordered atomics.

These C11/C++11 mappings require the platform to provide the following Physical Memory Attributes (as defined in the RISC-V Privileged ISA) for all memory:

- main memory
- coherent
- AMOArithmetic
- RsrvEventual

Platforms with different attributes may require different mappings, or require platform-specific SW (e.g., memory-mapped I/O).

A.6 Implementation Guidelines

The RVWMO and RVTSO memory models by no means preclude microarchitectures from employing sophisticated speculation techniques or other forms of optimization in order to deliver higher performance. The models also do not impose any requirement to use any one particular cache hierarchy, nor even to use a cache coherence protocol at all. Instead, these models only specify the behaviors that can be exposed to software. Microarchitectures are free to use any pipeline design, any coherent or non-coherent cache hierarchy, any on-chip interconnect, etc., as long as the design only admits executions that satisfy the memory model rules. That said, to help people understand the actual implementations of the memory model, in this section we provide some guidelines on how architects and programmers should interpret the models' rules.

Both RVWMO and RVTSO are multi-copy atomic (or "other-multi-copy-atomic"): any store value that is visible to a hart other than the one that originally issued it must also be conceptually visible to all other harts in the system. In other words, harts may forward from their own previous stores before those stores have become globally visible to all harts, but no early inter-hart forwarding is permitted. Multi-copy atomicity may be enforced in a number of ways. It might hold inherently due to the physical design of the caches and store buffers, it may be enforced via a single-writer/multiple-reader cache coherence protocol, or it might hold due to some other mechanism.

Although multi-copy atomicity does impose some restrictions on the microarchitecture, it is one of the key properties keeping the memory model from becoming extremely complicated. For example, a hart may not legally forward a value from a neighbor hart's private store buffer (unless of course it is done in such a way that no new illegal behaviors become architecturally visible). Nor may a cache coherence protocol forward a value from one hart to another until the coherence protocol has invalidated all older copies from other caches. Of course, microarchitectures may (and high-performance implementations likely will) violate these rules under the covers through speculation or other optimizations, as long as any non-compliant behaviors are not exposed to the programmer.

As a rough guideline for interpreting the PPO rules in RVWMO, we expect the following from the software perspective:

- programmers will use PPO rules 1 and 4–8 regularly and actively.
- expert programmers will use PPO rules 9–11 to speed up critical paths of important data structures.
- even expert programmers will rarely if ever use PPO rules 2–3 and 12–13 directly. These are included to facilitate common microarchitectural optimizations (rule 2) and the operational formal modeling approach (rules 3 and 12–13) described in Section B.3. They also facilitate the process of porting code from other architectures that have similar rules.

We also expect the following from the hardware perspective:

- PPO rules 1 and 3–6 reflect well-understood rules that should pose few surprises to architects.
- PPO rule 2 reflects a natural and common hardware optimization, but one that is very subtle and hence is worth double checking carefully.
- PPO rule 7 may not be immediately obvious to architects, but it is a standard memory model requirement
- The load value axiom, the atomicity axiom, and PPO rules 8–13 reflect rules that most hardware implementations will enforce naturally, unless they contain extreme optimizations. Of course, implementations should make sure to double check these rules nevertheless. Hardware must also ensure that syntactic dependencies are not "optimized away".

Architectures are free to implement any of the memory model rules as conservatively as they choose. For example, a hardware implementation may choose to do any or all of the following:

- interpret all fences as if they were FENCE RW,RW (or FENCE IORW,IORW, if I/O is involved), regardless of the bits actually set
- implement all fences with PW and SR as if they were FENCE RW,RW (or FENCE IORW,IORW, if I/O is involved), as PW with SR is the most expensive of the four possible main memory ordering components anyway
- emulate *aq* and *rl* as described in Section A.5
- enforcing all same-address load-load ordering, even in the presence of patterns such as "fri-rfi" and "RSW"
- forbid any forwarding of a value from a store in the store buffer to a subsequent AMO or LR to the same address
- forbid any forwarding of a value from an AMO or SC in the store buffer to a subsequent load to the same address
- implement TSO on all memory accesses, and ignore any main memory fences that do not include PW and SR ordering (e.g., as Ztso implementations will do)
- implement all atomics to be RCsc or even fully ordered, regardless of annotation

Architectures that implement RVTSO can safely do the following:

- Ignore all fences that do not have both PW and SR (unless the fence also orders I/O)
- Ignore all PPO rules except for rules 4 through 7, since the rest are redundant with other PPO rules under RVTSO assumptions

Other general notes:

- Silent stores (i.e., stores that write the same value that already exists at a memory location) behave like any other store from a memory model point of view. Likewise, AMOs which do not actually change the value in memory (e.g., an AMOMAX for which the value in *rs2* is smaller than the value currently in memory) are still semantically considered store operations. Microarchitectures that attempt to implement silent stores must take care to ensure that the memory model is still obeyed, particularly in cases such as RSW (Section A.3.5) which tend to be incompatible with silent stores.

- Writes may be merged (i.e., two consecutive writes to the same address may be merged) or subsumed (i.e., the earlier of two back-to-back writes to the same address may be elided) as long as the resulting behavior does not otherwise violate the memory model semantics.

The question of write subsumption can be understood from the following example:

	Hart 0		Hart 1
	`li t1, 3`		`li t3, 2`
	`li t2, 1`		
(a)	`sw t1,0(s0)`	(d)	`lw a0,0(s1)`
(b)	`fence w, w`	(e)	`sw a0,0(s0)`
(c)	`sw t2,0(s1)`	(f)	`sw t3,0(s0)`

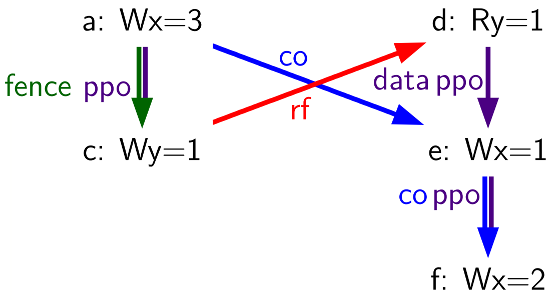


图 A.19: Write subsumption litmus test, allowed execution.

As written, if the load (d) reads value 1, then (a) must precede (f) in the global memory order:

- (a) precedes (c) in the global memory order because of rule 2
- (c) precedes (d) in the global memory order because of the Load Value axiom
- (d) precedes (e) in the global memory order because of rule 7
- (e) precedes (f) in the global memory order because of rule 1

In other words the final value of the memory location whose address is in `s0` must be 2 (the value written by the store (f)) and cannot be 3 (the value written by the store (a)).

A very aggressive microarchitecture might erroneously decide to discard (e), as (f) supersedes it, and this may in turn lead the microarchitecture to break the now-eliminated dependency between (d) and (f) (and hence also between (a) and (f)). This would violate the memory model rules, and hence it is forbidden. Write subsumption may in other cases be legal, if for example there were no data dependency between (d) and (e).

A.6.1 Possible Future Extensions

We expect that any or all of the following possible future extensions would be compatible with the RVWMO memory model:

- ‘V’ vector ISA extensions
- ‘J’ JIT extension
- Native encodings for load and store opcodes with *aq* and *rl* set
- Fences limited to certain addresses
- Cache writeback/flush/invalidate/etc. instructions

A.7 Known Issues

A.7.1 Mixed-size RSW

	Hart 0		Hart 1
	`li t1, 1`		`li t1, 1`
(a)	`lw a0,0(s0)`	(d)	`lw a1,0(s1)`
(b)	`fence rw,rw`	(e)	`amoswap.w.rl a2,t1,0(s2)`
(c)	`sw t1,0(s1)`	(f)	`ld a3,0(s2)`
		(g)	`lw a4,4(s2)`
			`xor a5,a4,a4`
			`add s0,s0,a5`
		(h)	`sw a2,0(s0)`

`Outcome: a0=1, a1=1, a2=0, a3=1, a4=0`

图 A.20: Mixed-size discrepancy (permitted by axiomatic models, forbidden by operational model)

	Hart 0		Hart 1
	`li t1, 1`		`li t1, 1`
(a)	`lw a0,0(s0)`	(d)	`ld a1,0(s1)`
(b)	`fence rw,rw`	(e)	`lw a2,4(s1)`
(c)	`sw t1,0(s1)`		`xor a3,a2,a2`
			`add s0,s0,a3`
		(f)	`sw a2,0(s0)`

`Outcome: a0=0, a1=1, a2=0`

图 A.21: Mixed-size discrepancy (permitted by axiomatic models, forbidden by operational model)

There is a known discrepancy between the operational and axiomatic specifications within the family of mixed-size RSW variants shown in Figures A.20–A.22. To address this, we may choose to add something like the following new PPO rule: Memory operation *a* precedes memory operation *b* in preserved program order (and hence also in the global memory order) if *a* precedes *b* in program

	Hart 0		Hart 1
	`li t1, 1`		`li t1, 1`
(a)	`lw a0,0(s0)`	(d)	`sw t1,4(s1)`
(b)	`fence rw,rw`	(e)	`ld a1,0(s1)`
(c)	`sw t1,0(s1)`	(f)	`lw a2,4(s1)`
			`xor a3,a2,a2`
			`add s0,s0,a3`
		(g)	`sw a2,0(s0)`

`Outcome: a0=1, a1=0x100000001, a1=1`

图 A.22: Mixed-size discrepancy (permitted by axiomatic models, forbidden by operational model)

order, a and b both access regular main memory (rather than I/O regions), a is a load, b is a store, there is a load m between a and b, there is a byte x that both a and m read, there is no store between a and m that writes to x, and m precedes b in PPO. In other words, in herd syntax, we may choose to add "`(po-loc & rsw);ppo;[W]`" to PPO. Many implementations will already enforce this ordering naturally. As such, even though this rule is not official, we recommend that implementers enforce it nevertheless in order to ensure forwards compatibility with the possible future addition of this rule to RVWMO.

附录 B Formal Memory Model Specifications, Version 0.1

To facilitate formal analysis of RVWMO, this chapter presents a set of formalizations using different tools and modeling approaches. Any discrepancies are unintended; the expectation is that the models describe exactly the same sets of legal behaviors.

This appendix should be treated as commentary; all normative material is provided in Chapter 十一 and in the rest of the main body of the ISA specification. All currently known discrepancies are listed in Section A.7. Any other discrepancies are unintentional.

B.1 Formal Axiomatic Specification in Alloy

We present a formal specification of the RVWMO memory model in Alloy (http://alloy.mit.edu). This model is available online at https://github.com/daniellustig/riscv-memory-model.

The online material also contains some litmus tests and some examples of how Alloy can be used to model check some of the mappings in Section A.5.

```
////////////////////////////////////////////////////////////////////////////////
// =RVWMO PPO=

// Preserved Program Order
fun ppo : Event->Event {
  // same-address ordering
  po_loc :> Store
  + rdw
  + (AMO + StoreConditional) <: rfi

  // explicit synchronization
  + ppo_fence
  + Acquire <: ^po :> MemoryEvent
  + MemoryEvent <: ^po :> Release
  + RCsc <: ^po :> RCsc
  + pair

  // syntactic dependencies
  + addrdep
  + datadep
  + ctrldep :> Store

  // pipeline dependencies
  + (addrdep+datadep).rfi
  + addrdep.^po :> Store
}

// the global memory order respects preserved program order
fact { ppo in ^gmo }
```

图 B.1: The RVWMO memory model formalized in Alloy (1/5: PPO)

```
////////////////////////////////////////////////////////////////////////////////
// =RVWMO axioms=

// Load Value Axiom
fun candidates[r: MemoryEvent] : set MemoryEvent {
  (r.~^gmo & Store & same_addr[r]) // writes preceding r in gmo
  + (r.^~po & Store & same_addr[r]) // writes preceding r in po
}

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]]
}

// Atomicity Axiom
pred Atomicity {
  all r: Store.~pair |            // starting from the lr,
    no x: Store & same_addr[r] |  // there is no store x to the same addr
      x not in same_hart[r]       // such that x is from a different hart,
      and x in r.~rf.^gmo         // x follows (the store r reads from) in gmo,
      and r.pair in x.^gmo        // and r follows x in gmo
}

// Progress Axiom implicit: Alloy only considers finite executions

pred RISCV_mm { LoadValue and Atomicity /* and Progress */ }
```

图 B.2: The RVWMO memory model formalized in Alloy (2/5: Axioms)

```
////////////////////////////////////////////////////////////////////////////////
// Basic model of memory

sig Hart {  // hardware thread
  start : one Event
}
sig Address {}
abstract sig Event {
  po: lone Event // program order
}

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,  // global memory order
  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}
// all StoreConditionals in the model are assumed to be successful
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, // opcode bit
  pw: lone Fence, // opcode bit
  sr: lone Fence, // opcode bit
  sw: lone Fence  // opcode bit
}
sig FenceTSO extends Fence {}

/* Alloy encoding detail: opcode bits are either set (encoded, e.g.,
 * as f.pr in iden) or unset (f.pr not in iden).  The bits cannot be used for
 * anything else */
fact { pr + pw + sr + sw in iden }
// likewise for ordering annotations
fact { acquireRCpc + acquireRCsc + releaseRCpc + releaseRCsc in iden }
// don't try to encode FenceTSO via pr/pw/sr/sw; just use it as-is
```

```
////////////////////////////////////////////////////////////////////////////////
// =Basic model rules=

// Ordering annotation groups
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 }

// There is no such thing as store-acquire or load-release, unless it's both
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)
}

// auxiliary definitions
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 }

// initial stores
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 }  // each event is in exactly one hart
fact { all a: Address | one Init & a.~address } // one init store per address
fact { no Init <: po and no po :> Init }
```

图 B.4: The RVWMO memory model formalized in Alloy (4/5: Basic model rules)

```
// po
fact { acyclic[po] }

// gmo
fact { total[^gmo, MemoryEvent] } // gmo is a total order over all MemoryEvents

//rf
fact { rf.~rf in iden } // each read returns the value of only one write
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 } // assume all SCs succeed

// 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
}

// filter out redundant instances and/or visualizations
fact { no gmo & gmo.gmo } // keep the visualization uncluttered
fact { all a: Address | some a.~address }

////////////////////////////////////////////////////////////////////////////////
// =Optional: opcode encoding restrictions=

// the list of blessed fences
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 shortcuts=
pred acyclic[rel: Event->Event] { no iden & ^rel }
pred total[rel: Event->Event, bag: Event] {
```

B.2 Formal Axiomatic Specification in Herd

The tool herd takes a memory model and a litmus test as input and simulates the execution of the test on top of the memory model. Memory models are written in the domain specific language Cat. This section provides two Cat memory model of RVWMO. The first model, Figure B.7, follows the *global memory order*, Chapter 十一, definition of RVWMO, as much as is possible for a Cat model. The second model, Figure B.8, is an equivalent, more efficient, partial order based RVWMO model.

The simulator herd is part of the diy tool suite — see http://diy.inria.fr for software and documentation. The models and more are available online at http://diy.inria.fr/cats7/riscv/.

```
(*************)
(* Utilities *)
(*************)

(* All fence relations *)
let fence.r.r = [R];fencerel(Fence.r.r);[R]
let fence.r.w = [R];fencerel(Fence.r.w);[W]
let fence.r.rw = [R];fencerel(Fence.r.rw);[M]
let fence.w.r = [W];fencerel(Fence.w.r);[R]
let fence.w.w = [W];fencerel(Fence.w.w);[W]
let fence.w.rw = [W];fencerel(Fence.w.rw);[M]
let fence.rw.r = [M];fencerel(Fence.rw.r);[R]
let fence.rw.w = [M];fencerel(Fence.rw.w);[W]
let fence.rw.rw = [M];fencerel(Fence.rw.rw);[M]
let fence.tso =
  let f = fencerel(Fence.tso) in
  ([W];f;[W]) | ([R];f;[M])

let fence =
  fence.r.r | fence.r.w | fence.r.rw |
  fence.w.r | fence.w.w | fence.w.rw |
  fence.rw.r | fence.rw.w | fence.rw.rw |
  fence.tso

(* Same address, no W to the same address in-between *)
let po-loc-no-w = po-loc \ (po-loc?;[W];po-loc)
(* Read same write *)
let rsw = rf^-1;rf
(* Acquire, or stronger  *)
let AQ = Acq|AcqRel
(* Release or stronger *)
and RL = RelAcqRel
(* All RCsc *)
let RCsc = Acq|Rel|AcqRel
(* Amo events are both R and W, relation rmw relates paired lr/sc *)
let AMO = R & W
let StCond = range(rmw)

(*************)
(* ppo rules *)
(*************)

(* Overlapping-Address Orderings *)
let r1 = [M];po-loc;[W]
and r2 = ([R];po-loc-no-w;[R]) \ rsw
and r3 = [AMO|StCond];rfi;[R]
(* Explicit Synchronization *)
and r4 = fence
and r5 = [AQ];po;[M]
and r6 = [M];po;[RL]
and r7 = [RCsc];po;[RCsc]
```

```
Total

(* Notice that herd has defined its own rf relation *)

(* Define ppo *)
include "riscv-defs.cat"

(********************************)
(* Generate global memory order *)
(********************************)

let gmo0 = (* precursor: ie build gmo as an total order that include gmo0 *)
  loc & (W\FW) * FW | # Final write after any write to the same location
  ppo |               # ppo compatible
  rfe                 # includes herd external rf (optimization)

(* Walk over all linear extensions of gmo0 *)
with  gmo from linearizations(M\IW,gmo0)

(* Add initial writes upfront -- convenient for computing rfGMO *)
let gmo = gmo | loc & IW * (M\IW)

(**********)
(* Axioms *)
(**********)

(* Compute rf according to the load value axiom, aka rfGMO *)
let WR = loc & ([W];(gmo|po);[R])
let rfGMO = WR \ (loc&([W];gmo);WR)

(* Check equality of herd rf and of rfGMO *)
empty (rf\rfGMO)|(rfGMO\rf) as RfCons

(* Atomicity axiom *)
let infloc = (gmo & loc)^-1
let inflocext = infloc & ext
let winside  = (infloc;rmw;inflocext) & (infloc;rf;rmw;inflocext) & [W]
empty winside as Atomic
```

图 B.7: riscv.cat, a herd version of the RVWMO memory model (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`, an alternative herd presentation of the RVWMO memory model (3/3)

B.3 An Operational Memory Model

This is an alternative presentation of the RVWMO memory model in operational style. It aims to admit exactly the same extensional behavior as the axiomatic presentation: for any given program, admitting an execution if and only if the axiomatic presentation allows it.

The axiomatic presentation is defined as a predicate on complete candidate executions. In contrast, this operational presentation has an abstract microarchitectural flavor: it is expressed as a state machine, with states that are an abstract representation of hardware machine states, and with explicit out-of-order and speculative execution (but abstracting from more implementation-specific microarchitectural details such as register renaming, store buffers, cache hierarchies, cache protocols, etc.). As such, it can provide useful intuition. It can also construct executions incrementally, making it possible to interactively and randomly explore the behavior of larger examples, while the axiomatic model requires complete candidate executions over which the axioms can be checked.

The operational presentation covers mixed-size execution, with potentially overlapping memory accesses of different power-of-two byte sizes. Misaligned accesses are broken up into single-byte accesses.

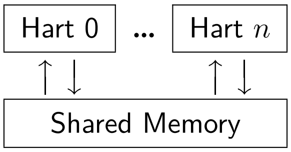
The operational model, together with a fragment of the RISC-V ISA semantics (RV64I and A), are integrated into the `rmem` exploration tool (`https://github.com/rems-project/rmem`). `rmem` can explore litmus tests (see A.2) and small ELF binaries exhaustively, pseudo-randomly and interactively. In `rmem`, the ISA semantics is expressed explicitly in Sail (see `https://github.com/rems-project/sail` for the Sail language, and `https://github.com/rems-project/sail-riscv` for the RISC-V ISA model), and the concurrency semantics is expressed in Lem (see `https://github.com/rems-project/lem` for the Lem language).

`rmem` has a command-line interface and a web-interface. The web-interface runs entirely on the client side, and is provided online together with a library of litmus tests: `http://www.cl.cam.ac.uk/~pes20/rmem`. The command-line interface is faster than the web-interface, specially in exhaustive mode.

Below is an informal introduction of the model states and transitions. The description of the formal model starts in the next subsection.

Terminology: In contrast to the axiomatic presentation, here every memory operation is either a load or a store. Hence, AMOs give rise to two distinct memory operations, a load and a store. When used in conjunction with “instruction”, the terms “load” and “store” refer to instructions that give rise to such memory operations. As such, both include AMO instructions. The term "acquire" refers to an instruction (or its memory operation) with the acquire-RCpc or acquire-RCsc annotation. The term "release" refers to an instruction (or its memory operation) with the release-RCpc or release-RCsc annotation.

Model states A model state consists of a shared memory and a tuple of hart states.



The shared memory state records all the memory store operations that have propagated so far, in the order they propagated (this can be made more efficient, but for simplicity of the presentation we keep it this way).

Each hart state consists principally of a tree of instruction instances, some of which have been *finished*, and some of which have not. Non-finished instruction instances can be subject to *restart*, e.g. if they depend on an out-of-order or speculative load that turns out to be unsound.

Conditional branch and indirect jump instructions may have multiple successors in the instruction tree. When such instruction is finished, any un-taken alternative paths are discarded.

Each instruction instance in the instruction tree has a state that includes an execution state of the intra-instruction semantics (the ISA pseudocode for this instruction). The model uses a formalization of the intra-instruction semantics in Sail. One can think of the execution state of an instruction as a representation of the pseudocode control state, pseudocode call stack, and local variable values. An instruction instance state also includes information about the instance's memory and register footprints, its register reads and writes, its memory operations, whether it is finished, etc.

Model transitions The model defines, for any model state, the set of allowed transitions, each of which is a single atomic step to a new abstract machine state. Execution of a single instruction will typically involve many transitions, and they may be interleaved in operational-model execution with transitions arising from other instructions. Each transition arises from a single instruction instance; it will change the state of that instance, and it may depend on or change the rest of its hart state and the shared memory state, but it does not depend on other hart states, and it will not change them. The transitions are introduced below and defined in Section B.3.5, with a precondition and a construction of the post-transition model state for each.

Transitions for all instructions:

- Fetch instruction: This transition represents a fetch and decode of a new instruction instance, as a program order successor of a previously fetched instruction instance (or the initial fetch address).

  The model assumes the instruction memory is fixed; it does not describe the behavior of self-modifying code. In particular, the Fetch instruction transition does not generate memory load operations, and the shared memory is not involved in the transition. Instead, the model depends on an external oracle that provides an opcode when given a memory location.

- ○ Register write: This is a write of a register value.

- ○ Register read: This is a read of a register value from the most recent program-order-predecessor instruction instance that writes to that register.

- ○ Pseudocode internal step: This covers pseudocode internal computation: arithmetic, function calls, etc.

- ○ Finish instruction: At this point the instruction pseudocode is done, the instruction cannot be restarted, memory accesses cannot be discarded, and all memory effects have taken place. For conditional branch and indirect jump instructions, any program order successors that were fetched from an address that is not the one that was written to the *pc* register are discarded, together with the sub-tree of instruction instances below them.

Transitions specific to load instructions:

- ○ Initiate memory load operations: At this point the memory footprint of the load instruction is provisionally known (it could change if earlier instructions are restarted) and its individual memory load operations can start being satisfied.

- Satisfy memory load operation by forwarding from unpropagated stores: This partially or entirely satisfies a single memory load operation by forwarding, from program-order-previous memory store operations.

- Satisfy memory load operation from memory: This entirely satisfies the outstanding slices of a single memory load operation, from memory.

- ○ Complete load operations: At this point all the memory load operations of the instruction have been entirely satisfied and the instruction pseudocode can continue executing. A load instruction can be subject to being restarted until the Finish instruction transition. But, under some conditions, the model might treat a load instruction as non-restartable even before it is finished (e.g. see Propagate store operation).

Transitions specific to store instructions:

○ Initiate memory store operation footprints: At this point the memory footprint of the store is provisionally known.

○ Instantiate memory store operation values: At this point the memory store operations have their values and program-order-successor memory load operations can be satisfied by forwarding from them.

○ Commit store instruction: At this point the store operations are guaranteed to happen (the instruction can no longer be restarted or discarded), and they can start being propagated to memory.

- Propagate store operation: This propagates a single memory store operation to memory.

○ Complete store operations: At this point all the memory store operations of the instruction have been propagated to memory, and the instruction pseudocode can continue executing.

Transitions specific to `sc` instructions:

- Early `sc` fail: This causes the `sc` to fail, either a spontaneous fail or because it is not paired with a program-order-previous `lr`.
- Paired `sc`: This transition indicates the `sc` is paired with an `lr` and might succeed.
- Commit and propagate store operation of an `sc`: This is an atomic execution of the transitions Commit store instruction and Propagate store operation, it is enabled only if the stores from which the `lr` read from have not been overwritten.
- Late `sc` fail: This causes the `sc` to fail, either a spontaneous fail or because the stores from which the `lr` read from have been overwritten.

Transitions specific to AMO instructions:

- Satisfy, commit and propagate operations of an AMO: This is an atomic execution of all the transitions needed to satisfy the load operation, do the required arithmetic, and propagate the store operation.

Transitions specific to fence instructions:

- ○ Commit fence

The transitions labeled ○ can always be taken eagerly, as soon as their precondition is satisfied, without excluding other behavior; the ● cannot. Although Fetch instruction is marked with a ●, it can be taken eagerly as long as it is not taken infinitely many times.

An instance of a non-AMO load instruction, after being fetched, will typically experience the following transitions in this order:

1. Register read
2. Initiate memory load operations
3. Satisfy memory load operation by forwarding from unpropagated stores and/or Satisfy memory load operation from memory (as many as needed to satisfy all the load operations of the instance)
4. Complete load operations
5. Register write
6. Finish instruction

Before, between and after the transitions above, any number of Pseudocode internal step transitions may appear. In addition, a Fetch instruction transition for fetching the instruction in the next program location will be available until it is taken.

This concludes the informal description of the operational model. The following sections describe the formal operational model.

B.3.1 Intra-instruction Pseudocode Execution

The intra-instruction semantics for each instruction instance is expressed as a state machine, essentially running the instruction pseudocode. Given a pseudocode execution state, it computes the next state. Most states identify a pending memory or register operation, requested by the pseudocode, which the memory model has to do. The states are (this is a tagged union; tags in small-caps):

Load_mem(*kind*, *address*, *size*, *load_continuation*)	-	memory load operation
Early_sc_fail(*res_continuation*)	-	allow `sc` to fail early
Store_ea(*kind*, *address*, *size*, *next_state*)	-	memory store effective address
Store_memv(*mem_value*, *store_continuation*)	-	memory store value
Fence(*kind*, *next_state*)	-	fence
Read_reg(*reg_name*, *read_continuation*)	-	register read
Write_reg(*reg_name*, *reg_value*, *next_state*)	-	register write
Internal(*next_state*)	-	pseudocode internal step
Done	-	end of pseudocode

Here:

- *mem_value* and *reg_value* are lists of bytes;
- *address* is an integer of XLEN bits;
- for load/store, *kind* identifies whether it is `lr/sc`, acquire-RCpc/release-RCpc, acquire-RCsc/release-RCsc, acquire-release-RCsc;
- for fence, *kind* identifies whether it is a normal or TSO, and (for normal fences) the predecessor and successor ordering bits;
- *reg_name* identifies a register and a slice thereof (start and end bit indices); and
- the continuations describe how the instruction instance will continue for each value that might be provided by the surrounding memory model (the *load_continuation* and *read_continuation* take the value loaded from memory and read from the previous register write, the *store_continuation* takes *false* for an `sc` that failed and *true* in all other cases, and *res_continuation* takes *false* if the `sc` fails and *true* otherwise).

For example, given the load instruction `lw x1,0(x2)`*, an execution will typically go as follows. The initial execution state will be computed from the pseudocode for the given opcode. This can be expected to be* Read_reg*(*`x2`*, read_continuation). Feeding the most recently written value of register* `x2` *(the instruction semantics will be blocked if necessary until the register value is available), say* `0x4000`*, to read_continuation returns* Load_mem*(*`plain_load`*,* `0x4000`*,* `4`*, load_continuation). Feeding the 4-byte value loaded from memory location* `0x4000`*, say* `0x42`*, to load_continuation returns* Write_reg*(*`x1`*,* `0x42`*,* Done*). Many* Internal*(next_state) states may appear before and between the states above.*

Notice that writing to memory is split into two steps, Store_ea and Store_memv: the first one makes the memory footprint of the store provisionally known, and the second one adds the value to be stored. We ensure these are paired in the pseudocode (Store_ea followed by Store_memv), but there may be other steps between them.

It is observable that the STORE_EA *can occur before the value to be stored is determined. For example, for the litmus test LB+fence.r.rw+data-po to be allowed by the operational model (as it is by RVWMO), the first store in Hart 1 has to take the* STORE_EA *step before its value is determined, so that the second store can see it is to a non-overlapping memory footprint, allowing the second store to be committed out of order without violating coherence.*

The pseudocode of each instruction performs at most one store or one load, except for AMOs that perform exactly one load and one store. Those memory accesses are then split apart into the architecturally atomic units by the hart semantics (see Initiate memory load operations and Initiate memory store operation footprints below).

Informally, each bit of a register read should be satisfied from a register write by the most recent (in program order) instruction instance that can write that bit (or from the hart's initial register state if there is no such write). Hence, it is essential to know the register write footprint of each instruction instance, which we calculate when the instruction instance is created (see the action of Fetch instruction below). We ensure in the pseudocode that each instruction does at most one register write to each register bit, and also that it does not try to read a register value it just wrote.

Data-flow dependencies (address and data) in the model emerge from the fact that each register read has to wait for the appropriate register write to be executed (as described above).

B.3.2 Instruction Instance State

Each instruction instance i has a state comprising:

- *program_loc*, the memory address from which the instruction was fetched;
- *instruction_kind*, identifying whether this is a load, store, AMO, fence, branch/jump or a 'simple' instruction (this also includes a *kind* similar to the one described for the pseudocode execution states);
- *src_regs*, the set of source *reg_name*s (including system registers), as statically determined from the pseudocode of the instruction;
- *dst_regs*, the destination *reg_name*s (including system registers), as statically determined from the pseudocode of the instruction;
- *pseudocode_state* (or sometimes just 'state' for short), one of (this is a tagged union; tags in small-caps):

Plain(*isa_state*)	-	ready to make a pseudocode transition
Pending_mem_loads(*load_continuation*)	-	requesting memory load operation(s)
Pending_mem_stores(*store_continuation*)	-	requesting memory store operation(s)

- *reg_reads*, the register reads the instance has performed, including, for each one, the register write slices it read from;
- *reg_writes*, the register writes the instance has performed;
- *mem_loads*, a set of memory load operations, and for each one the as-yet-unsatisfied slices (the byte indices that have not been satisfied yet), and, for the satisfied slices, the store slices (each consisting of a memory store operation and subset of its byte indices) that satisfied it.
- *mem_stores*, a set of memory store operations, and for each one a flag that indicates whether it has been propagated (passed to the shared memory) or not.
- information recording whether the instance is committed, finished, etc.

Each memory load operation includes a memory footprint (address and size). Each memory store operations includes a memory footprint, and, when available, a value.

A load instruction instance with a non-empty *mem_loads*, for which all the load operations are satisfied (i.e. there are no unsatisfied load slices) is said to be *entirely satisfied*.

Informally, an instruction instance is said to have *fully determined data* if the load (and `sc`) instructions feeding its source registers are finished. Similarly, it is said to have a *fully determined memory footprint* if the load (and `sc`) instructions feeding its memory operation address register are finished. Formally, we first define the notion of *fully determined register write*: a register write w from *reg_writes* of instruction instance i is said to be *fully determined* if one of the following conditions hold:

1. i is finished; or
2. the value written by w is not affected by a memory operation that i has made (i.e. a value loaded from memory or the result of `sc`), and, for every register read that i has made, that affects w, the register write from which i read is fully determined (or i read from the initial register state).

Now, an instruction instance i is said to have *fully determined data* if for every register read r from *reg_reads*, the register writes that r reads from are fully determined. An instruction instance i is said to have a *fully determined memory footprint* if for every register read r from *reg_reads* that feeds into i's memory operation address, the register writes that r reads from are fully determined.

The `rmem` *tool records, for every register write, the set of register writes from other instructions that have been read by this instruction at the point of performing the write. By carefully arranging the pseudocode of the instructions covered by the tool we were able to make it so that this is exactly the set of register writes on which the write depends on.*

B.3.3 Hart State

The model state of a single hart comprises:

- *hart_id*, a unique identifier of the hart;
- *initial_register_state*, the initial register value for each register;
- *initial_fetch_address*, the initial instruction fetch address;
- *instruction_tree*, a tree of the instruction instances that have been fetched (and not discarded), in program order.

B.3.4 Shared Memory State

The model state of the shared memory comprises a list of memory store operations, in the order they propagated to the shared memory.

When a store operation is propagated to the shared memory it is simply added to the end of the list. When a load operation is satisfied from memory, for each byte of the load operation, the most recent corresponding store slice is returned.

For most purposes, it is simpler to think of the shared memory as an array, i.e., a map from memory locations to memory store operation slices, where each memory location is mapped to a one-byte slice of the most recent memory store operation to that location. However, this abstraction is not detailed enough to properly handle the `sc` *instruction. The RVWMO* 原子性公理 *allows store operations from the same hart as the* `sc` *to intervene between the store operation of the* `sc` *and the store operations the paired* `lr` *read from. To allow such store operations to intervene, and forbid others, the array abstraction must be extended to record more information.*

Here, we use a list as it is very simple, but a more efficient and scalable implementations should probably use something better.

B.3.5 Transitions

Each of the paragraphs below describes a single kind of system transition. The description starts with a condition over the current system state. The transition can be taken in the current state only if the condition is satisfied. The condition is followed by an action that is applied to that state when the transition is taken, in order to generate the new system state.

Fetch instruction A possible program-order-successor of instruction instance *i* can be fetched from address *loc* if:

1. it has not already been fetched, i.e., none of the immediate successors of *i* in the hart's *instruction_tree* are from *loc*; and
2. if *i*'s pseudocode has already written an address to *pc*, then *loc* must be that address, otherwise *loc* is:
   - for a conditional branch, the successor address or the branch target address;
   - for a (direct) jump and link instruction (`jal`), the target address;
   - for an indirect jump instruction (`jalr`), any address; and
   - for any other instruction, $i.program\_loc + 4$.

Action: construct a freshly initialized instruction instance i' for the instruction in the program memory at *loc*, with state Plain(*isa_state*), computed from the instruction pseudocode, including the static information available from the pseudocode such as its *instruction_kind*, *src_regs*, and *dst_regs*, and add i' to the hart's *instruction_tree* as a successor of *i*.

The possible next fetch addresses (loc) are available immediately after fetching i and the model does not need to wait for the pseudocode to write to pc*; this allows out-of-order execution, and speculation past conditional branches and jumps. For most instructions these addresses are easily obtained from the instruction pseudocode. The only exception to that is the indirect jump instruction (*`jalr`*), where the address depends on the value held in a register. In principle the mathematical model should allow speculation to arbitrary addresses here. The exhaustive search in the* `rmem` *tool handles this by running the exhaustive search multiple times with a growing set of possible next fetch addresses for each indirect jump. The initial search uses empty sets, hence* *there is no fetch after indirect jump instruction until the pseudocode of the instruction writes to* pc*, and then we use that value for fetching the next instruction. Before starting the next iteration of exhaustive search, we collect for each indirect jump (grouped by code location) the set of values it wrote to* pc *in all the executions in the previous search iteration, and use that as possible next fetch addresses of the instruction. This process terminates when no new fetch addresses are detected.*

Initiate memory load operations An instruction instance *i* in state Plain(Load_mem(*kind*, *address*, *size*, *load_continuation*)) can always initiate the corresponding memory load operations. Action:

1. Construct the appropriate memory load operations *mlos*:
   - if *address* is aligned to *size* then *mlos* is a single memory load operation of *size* bytes from *address*;
   - otherwise, *mlos* is a set of *size* memory load operations, each of one byte, from the addresses $address \ldots address + size - 1$.
2. set *mem_loads* of *i* to *mlos*; and
3. update the state of *i* to Pending_mem_loads(*load_continuation*).

In Section 11.1 it is said that misaligned memory accesses may be decomposed at any granularity. Here we decompose them to one-byte accesses as this granularity subsumes all others.

Satisfy memory load operation by forwarding from unpropagated stores For a non-AMO load instruction instance *i* in state Pending_mem_loads(*load_continuation*), and a memory load operation *mlo* in *i.mem_loads* that has unsatisfied slices, the memory load operation can be partially or entirely satisfied by forwarding from unpropagated memory store operations by store instruction instances that are program-order-before *i* if:

1. all program-order-previous `fence` instructions with `.sr` and `.pw` set are finished;
2. for every program-order-previous `fence` instruction, *f*, with `.sr` and `.pr` set, and `.pw` not set, if *f* is not finished then all load instructions that are program-order-before *f* are entirely satisfied;
3. for every program-order-previous `fence.tso` instruction, *f*, that is not finished, all load instructions that are program-order-before *f* are entirely satisfied;

4. if i is a load-acquire-RCsc, all program-order-previous store-releases-RCsc are finished;

5. if i is a load-acquire-release, all program-order-previous instructions are finished;

6. all non-finished program-order-previous load-acquire instructions are entirely satisfied; and

7. all program-order-previous store-acquire-release instructions are finished;

Let *msoss* be the set of all unpropagated memory store operation slices from non-`sc` store instruction instances that are program-order-before i and have already calculated the value to be stored, that overlap with the unsatisfied slices of *mlo*, and which are not superseded by intervening store operations or store operations that are read from by an intervening load. The last condition requires, for each memory store operation slice *msos* in *msoss* from instruction i':

- that there is no store instruction program-order-between i and i' with a memory store operation overlapping *msos*; and
- that there is no load instruction program-order-between i and i' that was satisfied from an overlapping memory store operation slice from a different hart.

Action:

1. update *i.mem_loads* to indicate that *mlo* was satisfied by *msoss*; and

2. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' that is a program-order-successor of i, and every memory load operation mlo' of i' that was satisfied from $msoss'$, if there exists a memory store operation slice $msos'$ in $msoss'$, and an overlapping memory store operation slice from a different memory store operation in *msoss*, and $msos'$ is not from an instruction that is a program-order-successor of i, restart i' and its *restart-dependents*.

Where, the *restart-dependents* of instruction j are:

- program-order-successors of j that have data-flow dependency on a register write of j;
- program-order-successors of j that have a memory load operation that reads from a memory store operation of j (by forwarding);
- if j is a load-acquire, all the program-order-successors of j;
- if j is a load, for every `fence`, f, with `.sr` and `.pr` set, and `.pw` not set, that is a program-order-successor of j, all the load instructions that are program-order-successors of f;
- if j is a load, for every `fence.tso`, f, that is a program-order-successor of j, all the load instructions that are program-order-successors of f; and

- (recursively) all the restart-dependents of all the instruction instances above.

Forwarding memory store operations to a memory load might satisfy only some slices of the load, leaving other slices unsatisfied.

A program-order-previous store operation that was not available when taking the transition above might make msoss provisionally unsound (violating coherence) when it becomes available. That store will prevent the load from being finished (see Finish instruction), and will cause it to restart when that store operation is propagated (see Propagate store operation).

A consequence of the transition condition above is that store-release-RCsc memory store operations cannot be forwarded to load-acquire-RCsc instructions: msoss does not include memory store operations from finished stores (as those must be propagated memory store operations), and the condition above requires all program-order-previous store-releases-RCsc to be finished when the load is acquire-RCsc.

Satisfy memory load operation from memory For an instruction instance i of a non-AMO load instruction or an AMO instruction in the context of the "Satisfy, commit and propagate operations of an AMO" transition, any memory load operation *mlo* in i.*mem_loads* that has unsatisfied slices, can be satisfied from memory if all the conditions of Satisfy memory load operation by forwarding from unpropagated stores are satisfied. Action: let *msoss* be the memory store operation slices from memory covering the unsatisfied slices of *mlo*, and apply the action of Satisfy memory load operation by forwarding from unpropagated stores.

Note that Satisfy memory load operation by forwarding from unpropagated stores might leave some slices of the memory load operation unsatisfied, those will have to be satisfied by taking the transition again, or taking Satisfy memory load operation from memory. Satisfy memory load operation from memory, on the other hand, will always satisfy all the unsatisfied slices of the memory load operation.

Complete load operations A load instruction instance i in state PENDING_MEM_LOADS(*load_continuation*) can be completed (not to be confused with finished) if all the memory load operations i.*mem_loads* are entirely satisfied (i.e. there are no unsatisfied slices). Action: update the state of i to PLAIN(*load_continuation(mem_value)*), where *mem_value* is assembled from all the memory store operation slices that satisfied i.*mem_loads*.

Early `sc` fail An `sc` instruction instance i in state PLAIN(EARLY_SC_FAIL(*res_continuation*)) can always be made to fail. Action: update the state of i to PLAIN(*res_continuation(false)*).

Paired `sc` An `sc` instruction instance i in state Plain(Early_sc_fail(*res_continuation*)) can continue its (potentially successful) execution if i is paired with an `lr`. Action: update the state of i to Plain(*res_continuation(true)*).

Initiate memory store operation footprints An instruction instance i in state Plain(Store_ea(*kind*, *address*, *size*, *next_state*)) can always announce its pending memory store operation footprint. Action:

1. construct the appropriate memory store operations *msos* (without the store value):
   - if *address* is aligned to *size* then *msos* is a single memory store operation of *size* bytes to *address*;
   - otherwise, *msos* is a set of *size* memory store operations, each of one-byte size, to the addresses $\mathit{address} \ldots \mathit{address} + \mathit{size} - 1$.
2. set *i.mem_stores* to *msos*; and
3. update the state of i to Plain(*next_state*).

Note that after taking the transition above the memory store operations do not yet have their values. The importance of splitting this transition from the transition below is that it allows other program-order-successor store instructions to observe the memory footprint of this instruction, and if they don't overlap, propagate out of order as early as possible (i.e. before the data register value becomes available).

Instantiate memory store operation values An instruction instance i in state Plain(Store_memv(*mem_value*, *store_continuation*)) can always instantiate the values of the memory store operations *i.mem_stores*. Action:

1. split *mem_value* between the memory store operations *i.mem_stores*; and
2. update the state of i to Pending_mem_stores(*store_continuation*).

Commit store instruction An uncommitted instruction instance i of a non-`sc` store instruction or an `sc` instruction in the context of the "Commit and propagate store operation of an `sc`" transition, in state Pending_mem_stores(*store_continuation*), can be committed (not to be confused with propagated) if:

1. *i* has fully determined data;
2. all program-order-previous conditional branch and indirect jump instructions are finished;
3. all program-order-previous `fence` instructions with `.sw` set are finished;
4. all program-order-previous `fence.tso` instructions are finished;
5. all program-order-previous load-acquire instructions are finished;
6. all program-order-previous store-acquire-release instructions are finished;
7. if *i* is a store-release, all program-order-previous instructions are finished;
8. all program-order-previous memory access instructions have a fully determined memory footprint;
9. all program-order-previous store instructions, except for `sc` that failed, have initiated and so have non-empty *mem_stores*; and
10. all program-order-previous load instructions have initiated and so have non-empty *mem_loads*.

Action: record that *i* is committed.

Notice that if condition 8 is satisfied the conditions 9 and 10 are also satisfied, or will be satisfied after taking some eager transitions. Hence, requiring them does not strengthen the model. By requiring them, we guarantee that previous memory access instructions have taken enough transitions to make their memory operations visible for the condition check of Propagate store operation, which is the next transition the instruction will take, making that condition simpler.

Propagate store operation For a committed instruction instance *i* in state PENDING_MEM_STORES(*store_continuation*), and an unpropagated memory store operation *mso* in *i.mem_stores*, *mso* can be propagated if:

1. all memory store operations of program-order-previous store instructions that overlap with *mso* have already propagated;
2. all memory load operations of program-order-previous load instructions that overlap with *mso* have already been satisfied, and (the load instructions) are *non-restartable* (see definition below); and

3. all memory load operations that were satisfied by forwarding *mso* are entirely satisfied.

Where a non-finished instruction instance j is *non-restartable* if:

1. there does not exist a store instruction s and an unpropagated memory store operation *mso* of s such that applying the action of the "Propagate store operation" transition to *mso* will result in the restart of j; and

2. there does not exist a non-finished load instruction l and a memory load operation *mlo* of l such that applying the action of the "Satisfy memory load operation by forwarding from unpropagated stores"/"Satisfy memory load operation from memory" transition (even if *mlo* is already satisfied) to *mlo* will result in the restart of j.

Action:

1. update the shared memory state with *mso*;

2. update *i.mem_stores* to indicate that *mso* was propagated; and

3. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' program-order-after i and every memory load operation *mlo*$'$ of i' that was satisfied from *msoss*$'$, if there exists a memory store operation slice *msos*$'$ in *msoss*$'$ that overlaps with *mso* and is not from *mso*, and *msos*$'$ is not from a program-order-successor of i, restart i' and its *restart-dependents* (see Satisfy memory load operation by forwarding from unpropagated stores).

Commit and propagate store operation of an `sc` An uncommitted `sc` instruction instance i, from hart h, in state Pending_mem_stores(*store_continuation*), with a paired `lr` i' that has been satisfied by some store slices *msoss*, can be committed and propagated at the same time if:

1. i' is finished;

2. every memory store operation that has been forwarded to i' is propagated;

3. the conditions of Commit store instruction is satisfied;

4. the conditions of Propagate store operation is satisfied (notice that an `sc` instruction can only have one memory store operation); and

5. for every store slice *msos* from *msoss*, *msos* has not been overwritten, in the shared memory, by a store that is from a hart that is not *h*, at any point since *msos* was propagated to memory.

Action:

1. apply the actions of Commit store instruction; and
2. apply the action of Propagate store operation.

Late `sc` fail An `sc` instruction instance *i* in state PENDING_MEM_STORES(*store_continuation*), that has not propagated its memory store operation, can always be made to fail. Action:

1. clear *i.mem_stores*; and
2. update the state of *i* to PLAIN(*store_continuation(false)*).

For efficiency, the `rmem` *tool allows this transition only when it is not possible to take the Commit and propagate store operation of an* `sc` *transition. This does not affect the set of allowed final states, but when explored interactively, if the* `sc` *should fail one should use the Early* `sc` *fail transition instead of waiting for this transition.*

Complete store operations A store instruction instance *i* in state PENDING_MEM_STORES(*store_continuation*), for which all the memory store operations in *i.mem_stores* have been propagated, can always be completed (not to be confused with finished). Action: update the state of *i* to PLAIN(*store_continuation(true)*).

Satisfy, commit and propagate operations of an AMO An AMO instruction instance *i* in state PENDING_MEM_LOADS(*load_continuation*) can perform its memory access if it is possible to perform the following sequence of transitions with no intervening transitions:

1. Satisfy memory load operation from memory
2. Complete load operations
3. Pseudocode internal step (zero or more times)
4. Instantiate memory store operation values

5. Commit store instruction

6. Propagate store operation

7. Complete store operations

and in addition, the condition of Finish instruction, with the exception of not requiring i to be in state Plain(Done), holds after those transitions. Action: perform the above sequence of transitions (this does not include Finish instruction), one after the other, with no intervening transitions.

Notice that program-order-previous stores cannot be forwarded to the load of an AMO. This is simply because the sequence of transitions above does not include the forwarding transition. But even if it did include it, the sequence will fail when trying to do the Propagate store operation transition, as this transition requires all program-order-previous store operations to overlapping memory footprints to be propagated, and forwarding requires the store operation to be unpropagated.

In addition, the store of an AMO cannot be forwarded to a program-order-successor load. Before taking the transition above, the store operation of the AMO does not have its value and therefore cannot be forwarded; after taking the transition above the store operation is propagated and therefore cannot be forwarded.

Commit fence A fence instruction instance i in state Plain(Fence(*kind*, *next_state*)) can be committed if:

1. if i is a normal fence and it has `.pr` set, all program-order-previous load instructions are finished;

2. if i is a normal fence and it has `.pw` set, all program-order-previous store instructions are finished; and

3. if i is a `fence.tso`, all program-order-previous load and store instructions are finished.

Action:

1. record that i is committed; and

2. update the state of i to Plain(*next_state*).

Register read An instruction instance *i* in state Plain(Read_reg(*reg_name*, *read_cont*)) can do a register read of *reg_name* if every instruction instance that it needs to read from has already performed the expected *reg_name* register write.

Let *read_sources* include, for each bit of *reg_name*, the write to that bit by the most recent (in program order) instruction instance that can write to that bit, if any. If there is no such instruction, the source is the initial register value from *initial_register_state*. Let *reg_value* be the value assembled from *read_sources*. Action:

1. add *reg_name* to *i.reg_reads* with *read_sources* and *reg_value*; and
2. update the state of *i* to Plain(*read_cont(reg_value)*).

Register write An instruction instance *i* in state Plain(Write_reg(*reg_name*, *reg_value*, *next_state*)) can always do a *reg_name* register write. Action:

1. add *reg_name* to *i.reg_writes* with *deps* and *reg_value*; and
2. update the state of *i* to Plain(*next_state*).

where *deps* is a pair of the set of all *read_sources* from *i.reg_reads*, and a flag that is true iff *i* is a load instruction instance that has already been entirely satisfied.

Pseudocode internal step An instruction instance *i* in state Plain(Internal(*next_state*)) can always do that pseudocode-internal step. Action: update the state of *i* to Plain(*next_state*).

Finish instruction A non-finished instruction instance *i* in state Plain(Done) can be finished if:

1. if *i* is a load instruction:
   (a) all program-order-previous load-acquire instructions are finished;
   (b) all program-order-previous `fence` instructions with `.sr` set are finished;
   (c) for every program-order-previous `fence.tso` instruction, *f*, that is not finished, all load instructions that are program-order-before *f* are finished; and
   (d) it is guaranteed that the values read by the memory load operations of *i* will not cause coherence violations, i.e., for any program-order-previous instruction instance i', let *cfp*

be the combined footprint of propagated memory store operations from store instructions program-order-between i and i', and *fixed memory store operations* that were forwarded to i from store instructions program-order-between i and i' including i', and let $\overline{cfp}$ be the complement of *cfp* in the memory footprint of i. If $\overline{cfp}$ is not empty:

  i. i' has a fully determined memory footprint;

  ii. i' has no unpropagated memory store operations that overlap with $\overline{cfp}$; and

  iii. if i' is a load with a memory footprint that overlaps with $\overline{cfp}$, then all the memory load operations of i' that overlap with $\overline{cfp}$ are satisfied and i' is *non-restartable* (see the Propagate store operation transition for how to determined if an instruction is non-restartable).

Here, a memory store operation is called fixed if the store instruction has fully determined data.

2. i has a fully determined data; and

3. if i is not a fence, all program-order-previous conditional branch and indirect jump instructions are finished.

Action:

1. if i is a conditional branch or indirect jump instruction, discard any untaken paths of execution, i.e., remove all instruction instances that are not reachable by the branch/jump taken in *instruction_tree*; and

2. record the instruction as finished, i.e., set *finished* to *true*.

B.3.6 Limitations

- The model covers user-level RV64I and RV64A. In particular, it does not support the misaligned atomics extension "Zam" or the total store ordering extension "Ztso". It should be trivial to adapt the model to RV32I/A and to the G, Q and C extensions, but we have never tried it. This will involve, mostly, writing Sail code for the instructions, with minimal, if any, changes to the concurrency model.

- The model covers only normal memory accesses (it does not handle I/O accesses).

- The model does not cover TLB-related effects.

- The model assumes the instruction memory is fixed. In particular, the Fetch instruction transition does not generate memory load operations, and the shared memory is not involved in the transition. Instead, the model depends on an external oracle that provides an opcode when given a memory location.
- The model does not cover exceptions, traps and interrupts.

参考文献

[1] RISC-V Assembly Programmer's Manual. https://github.com/riscv/riscv-asm-manual.

[2] RISC-V ELF psABI Specification. https://github.com/riscv/riscv-elf-psabi-doc/.

[3] 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.

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

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

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

[7] 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.

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

[9] 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.

[10] Heidi Pan, Benjamin Hindman, and Krste Asanović. Lithe: Enabling efficient composition of parallel libraries. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism (HotPar '09)*, Berkeley, CA, March 2009.

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

[12] David A. Patterson and Carlo H. Séquin. RISC I: A reduced instruction set VLSI computer. In *ISCA*, pages 443–458, 1981.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

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