RISC-V 指令集手册

卷 I：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 9 月 23 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；而版本 2.2 定义了 NaN-boxing 方案，并更改了 FMIN 和 FMAX 指令的定义。
- 将文档的名字改为指代“非特权的”指令，以此作为将 ISA 规范从平台概述授权中分离的移动的一部分。
- 为执行环境、硬件线程、陷入和内存访问添加了更清晰和更精确的定义。
- 定义了指令集的种类：*标准的*, *保留的*, *自定义的*, *非标准的*, and *不合格的*。
- 移除了在交替字节序下的文本隐含操作，因为交替字节序操作还没有被 RISC-V 所定义。
- 修改了对非对齐的加载和存储行为的描述。现在，规范允许在执行环境接口中进行的显式的非对齐地址陷入，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的隐式处理。而且，现在允许报告，由本不应被模拟的非对齐访问（包括原子访问），所引起的访问错误异常。
- 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，它仍然是仅有的用于标准指令获取的一致性机制。
- 去除了禁止 RV32E 和其它扩展一起使用的约束。
- 去除了平台相关的约束，即，在 RV32E 和 RV64I 章节中，特定的编码产生的非法指令异常。
- 计数器/计时器指令现在不被认为是强制性的基础 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，并改变了它们遇到有符号 NaN 输入时的行为，以符合建议的 IEEE 754-201x 规范中的 minimumNumber 和 maximumNumber 操作规范。
- 定义了内存一致性模型 RVWMO。
- 定义了“Zam”扩展，它允许未对齐的 AMO 并指定它们的语义。
- 定义了“Ztso”扩展，它执行比 RVWMO 更加严格的内存一致性模型。
- 改善了描述和注释。
- 定义了术语 IALIGN，作为描述指令地址对齐约束的简写。
- 去除了 P 扩展章节的内容，因为它现在已经被活跃的任务组文档所取代。

- 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）映射的表。
- 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-boxing 模型持有较窄（<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 packed-SIMD 协议。
- 提出了一个 V 向量指令集扩展的草案。
- 提出了一个 N 用户级陷入扩展的早期草案。
- 扩充了伪指令列表。
- 移除了调用规约章节，它已经被 RISC-V ELF psABI 规范 [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 [16] 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转换的返回值。
- 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。
- 为软浮点调用惯例放松了栈对齐，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 [15] 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）。这导致我们显著节省了位置无关的代码。
- JAL 指令现在已经被移动到 U-Type 格式，它带有明确目的寄存器；J 指令被弃用，由 *rd*=`x0` 的 JAL 代替。这样去掉了仅有的目的寄存器不明确的指令，也把 J-Type 指令格式从基础 ISA 中去除。这虽然减少了 JAL 的适用范围，但是会明显减少基础 ISA 的复杂性。
- 关于 JALR 指令的静态提示已经被丢弃。对于符合标准调用约定的代码，这些提示、还有 *rd* 和 *rs1* 寄存器的修饰符，都是多余的。
- 现在，JALR 指令在计算出目标地址之后，清除了它的最低位，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“过度架构”，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 [4]。
- 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，以便简化超管级（hypervisor）的开发。
- 一个简化了新的特权架构设计的实验的 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 [11]*、*RISC-II [5]*、*SOAR [14]* 和 *SPUR [8]*）。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，例如缓存行尺寸，或者特权架构的细节，例如页面转换。这既是为了简化，也是为了提供各种微架构或各种权限架构最大程度的灵活性。

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 的处理核心与其它不兼容 RISC-V 的核心、固定功能加速器、各种物理内存结构、I/O 设备，和一个允许各组件通信的交互结构。

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 的核心可以通过多线程，支持多个兼容 RISC-V 的“硬件线程（*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。

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，我们期望有许多可编程加速器将是基于 RISC-V 的、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，它分担了主应用核心中 I/O 处理任务的负荷。

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，到一个有数千个共享内存的众核服务节点的集群。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（hart）的数量和类型（包

括被硬件线程支持的权限模式)、内存和 I/O 区域的可访问性和属性、执行在各硬件线程上的所有合法指令的行为（例如，ISA 就是 EEI 的一个组件），以及在包括环境调用在内的执行期间，任何中断或异常的处理。EEI 的例子包括了 Linux 应用程序二进制接口（ABI），或者 RISC-V 管理员二进制接口（SBI)。一个 RISC-V 执行环境的实现可以是纯硬件的、纯软件的、或者是硬件和软件的组合。例如，操作码陷入和软件模拟可以被用于实现硬件里没有提供的功能。执行环境实现的例子包括:

- “裸机”硬件平台（”Bare metal” hardware platform)：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（RISC-V operating system)：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 虚拟器（RISC-V hypervisors)：为宾客操作系统（guest operating system）提供了多个管理员级别的执行环境。
- 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管理员级别的执行环境。

可以考虑将一个裸的硬件平台定义为一个执行环境接口（*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比硬件更抽象的接口，因为 *EEI* 越抽象，它所提供的跨不同硬件平台的可移植性越大。*EEI* 经常是一层叠着一层的，一个较高层的 *EEI* 使用另一个较低层的 *EEI*。

从软件在给定的执行环境中运行的观点看，hart 是一种资源，它在该执行环境中自动地获取和执行 RISC-V 指令。在这个方面，hart 的行动像是一种硬件线程资源，即使执行环境将时间多路复用到了真实的硬件上。一些 EEI 支持额外硬件线程的创建和解构，例如，通过环境调用来派生新的 hart。

执行环境负责确保它的各个硬件线程的最终推进。对于一个给定的 hart，当其正在运作要明确等待某个事件的机制（例如本规范卷 II 中定义的 wait-for-interrupt 指令）时，执行环境的职责被暂停；当硬件线程终止时，该责任结束。hart 的推进是由下列事件构成的:

- 一个指令的引退。
- 一个陷入，就像 1.6 节 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“*hart*”的引入是在 *Lithe [9, 10]* 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有）和一些对基础 ISA 的可选的扩展。基础整数 ISA 非常类似于早期的 RISC 处理器，除了没有分支延迟槽，和支持可选的变长指令编码。"基础"是被小心地限制在足以为编译器、汇编器、链接器、和操作系统（带有额外特权操作）提供合理目标的一个最小的指令集合的范围内，并因此提供了一个便捷的 ISA 和软件工具链"骨架"，可以围绕它们来构建更多定制的处理器 ISA。

尽管可以很方便的说这个 RISC-V ISA，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**??**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 **??**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 **??** 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 的最早的版本的确有这种可选择的设计，但是在 2011 年 1 月，RISC-V 的设计变成了如今的选择。我们的关注点在于在 64 位 ISA 中支持 32 位整数，而不在于提供对 32 位 ISA 的兼容性；并且动机是消除 RV32I 中，并非所有操作码都有“*W”后缀所引起的不对称性（例如，有 ADDW，但是 AND 没有 ANDW）。事后来看，同时设计两个 ISA，而不是先设计一个再于其上追加设计另一个，作为如此做法的结果，这可能是不合适的；而且，出于我们必须把平台的需求折进 ISA 规范之中的信条，那意味着在 RV64I 中将需要所有的 RV32I 的指令。虽然现在改变编码已经太晚了，但是由于上述原因，这也几乎没有什么实际意义了。

已经被注意到，我们能够将“*W”变体作为 RV32I 系统的一个扩展启用，以提供一种跨 RV64I 和未来 RV32 变体的常用编码。

RISC-V 已经被设计为支持广泛的定制和特化。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 的指令和指令集扩展研制了一个命名约定，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，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 $2^{\mathrm{XLEN}} - 1$ 的字节与位于地址零的字节是相邻的。因此，硬件进行内存地址计算时，忽略了溢出，代之以按模 2^{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方式。一个硬件线程的地址空间可以有不同地址范围，它可以是（1）空白的，或者（2）包含*主内存*，或者（3）包含一个或多个 *I/O 设备*。I/O 设备的读写可以造成可见的副作用，但是访问主内存不能。虽然执行环境可能把硬件线程地址空间中的所有内容都称作 I/O 设备，但是通常都会期望把某些部分指定为主内存。

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

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

执行每个 RISC-V 机器指令涉及了一次或多次内存访问，这进一步划分为*隐式*和*显式*访问。对于每个被执行的指令，进行一次*隐式*内存读（指令获取）是为了获得已编码指令进行执行。许多 RISC-V 指令在指令获取之外不再进一步地访问内存。在由专门的加载和存储指令决定的地址处，

有对内存执行显式的读或写。在本非特权 ISA 文档之外，执行环境可能强制要求指令的执行实施其他隐式的内存访问（例如实现地址转换）。

执行环境决定了各种内存访问操作可以访问非空地址空间的哪些部分。例如，可以被指令读取操作隐式读到的位置集合，可能与那些可以被加载指令操作显式读到的位置集合有交叠；以及，可以被存储指令操作显式写到的位置集合，可能只是可读位置的一个子集。通常，如果一个指令尝试访问的内存位于一个不可访问的地址处，将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，一个有效的实现可能会尝试第一时间读取所有的主内存，缓存尽可能多的可获取（可执行）字节以供之后的指令获取，以及避免为了指令获取而再次读主内存。为了确保某些隐式读只在写入相同内存位置之后是有序的，软件必须执行为此目的而定义的、特定的屏障指令或缓存控制指令（例如第 3 章 **??**里定义的 FENCE.I 指令）。

由一个硬件线程发起的内存访问（隐式或显式），在被另一个硬件线程、或者任何其它可访问相同内存的代理所感知时，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适用的内存一致性模型的约束。用于 RISC-V 的默认的内存一致性模型是 RISC-V 弱内存排序（RVWMO），定义在第 17 章 **??**和附录中。或者，一种实现也可以采用更强的模型：全存储排序（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需要，软件必须执行屏障或缓存控制指令来确保特定顺序的内存访问。

1.5 基础指令长度编码

基础 RISC-V ISA 有固定长度的 32 位指令，必须在 32 位边界上自然地对齐。然而，标准 RISC-V 编码策略被设计为支持具有可变长度指令的 ISA 扩展的，每条指令在长度上可以是任意数目的 16 位指令包（*parcel*），指令包在 16 位边界自然对齐。第 18 章 **??**中描述的压缩 ISA 扩展，通过提供压缩的 16 位指令，以及放松了对齐的限制，减少了代码尺寸，允许所有的指令（16 位和 32 位）在任意 16 位边界上对齐而提升了代码的密度。

我们使用术语“IALIGN”（以位为单位）来表示实现所执行的指令空间对齐约束。在基础 ISA 中，IALIGN 是 32 位。但是在某些 ISA 扩展中，包括在压缩 ISA 扩展中，将 IALIGN 放宽到 16 位。IALIGN 不能取除了 16 和 32 以外的任何其它值。

我们使用术语“ILEN”（以位为单位）来表示被实现所支持的最大指令长度，它总是 IALIGN 的倍数。对于只支持一个基础指令集的实现，ILEN 是 32 位。支持更长指令的实现也有更大的 ILEN 值。

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 位编码指令的提议还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低序位设置为 1，关于 48 位和 64 位长度的约定如图 1.1所示。指令长度在 80 位到 176 位之间的，使用一个 3 位的域来编码，在位 [14:12] 中给出了除最先的 5×16 位字以外的 16 位字的数目。位 [14:12] 被设置为“111”的编码被保留，用于未来更长的指令编码。

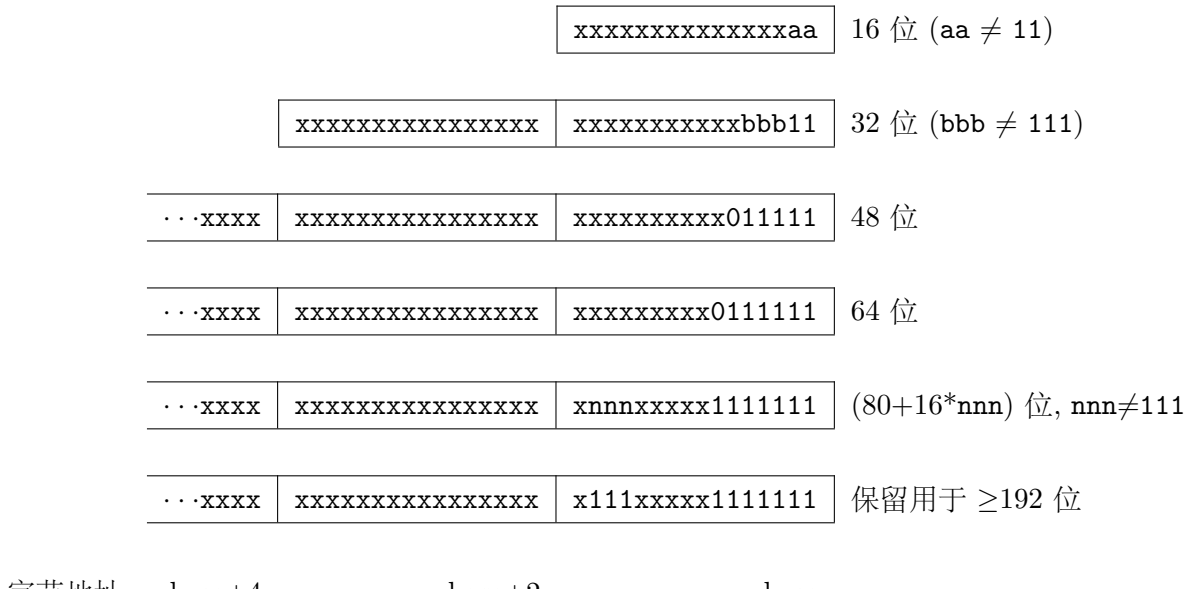


图 1.1: RISC-V 指令长度编码。此时只有 16 位和 32 位编码被认为是被冻结的。

给定压缩格式的代码尺寸和节能效果，我们希望在 *ISA* 编码策略中构建对压缩格式的支持，而不是事后才想起添加它；但是为了允许更简单的实现，我们不想强制使用压缩的格式。我们也希望可选地允许更长的指令，以支持实验和更大的指令集扩展。尽管我们的编码约定需要更严格的核心 *RISC-V ISA* 编码，但是这仍然有许多有益的效果。

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

也许更重要的是，通过把我们的基础 *ISA* 凝结成 *32* 位指令字的子集，我们为非标准的和自定义的扩展留出了更多可用的空间。特别地，基础 *RV32I ISA* 在 *32* 位指令字中使用少于 *1/8* 的编码空间。正如第 *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，同时保留对标准 $\geq$*32* 位指令集扩展的支持。甚至，如果实现也不需要长度 $>$*32* 位的指令，它可以为不一致的扩展恢复另外四个主要的操作码。

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

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

在所有的 *RISC-V* 实现上，软件可以依靠将一个包含“*0*”的自然对齐的 *32* 位字作为一个非法指令，以供明确需要非法指令的软件使用。由于可变长度编码，定义一个相应的全是“*1*”的已知非法值是更加困难的。软件不能一般地使用 *ILEN* 位全是“*1*”的非法值，因为软件可能不知道最终的目标机器的 *ILEN*（例如：如果软件被编译为一个用于许多不同的机器的标准二进制库）。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* 的机器上的取指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，让可变指令长度的获取和解码变得复杂。

RISC-V 基础 ISA 既有小字节序的内存系统，也有大字节序的内存系统，后者需要特权架构进一步定义大字节序的操作。不论内存系统的字节序如何，指令都作为 16 位小字节序的包的序列被存储在内存中。形成一个指令的包被存储在递增的半字地址处，最低地址的包在指令规范中持有最低的若干位。

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

我们不得不固定指令包在内存中存储的顺序，独立于内存系统的字节序之外，来确保长度编码位始终以半字地址顺序首先出现。这允许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活动的字节序，尽管在双字节序系统中，字节序的模式可能在执行期间动态变化。与之相反，通过给定指令一个固定的字节序，有时可以让仔细编写的软件的字节序不可知，甚至是以二进制的形式，就像位置无关的代码一样。

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

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

1.6 异常、陷入和中断

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

下面的章节中的指令描述描述了在指令执行期间可以引发异常的条件。大多数 RISC-V EEI 的通常行为是，当在一个指令上发出异常的信号时，会发生一次到某些处理程序的陷入（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。

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

被包含的陷入： 这种陷入对于运行在执行环境中的软件可见，并由软件处理。例如，在一个于硬件线程上同时提供管理员模式和用户模式的 EEI 中，用户模式硬件线程的 ECALL 通常，将导致控制被转移到运行在相同硬件线程上的一个管理员模式的处理程序。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管理员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，或者允许监视计时器失效。每个 EEI 应当定义执行应如何被终止，以及如何将其汇报给外部环境。

Table 1.1 显示了每种陷入的特点：

	被包含的	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被精确处理，尽管建议是尽可能地保持精度。被包含的陷入和被请求的陷入可以被执行环境内部的软件观测到是不精确的。不可见的陷入，根据定义，不能被运行在执行环境内部的软件观测到是否精确。致命陷入可以被运行在执行环境内部的软件观测到不精确，如果已知错误的指令没有引起直接的终止的话。

因为这篇文档描述了非特权指令，所以陷入是很少被提及的。处理包含陷入的架构性方法被定义在特权架构手册中，伴有支持更丰富 EEI 的其它特征。这里只记录了被单独定义的引发请求陷入的非特权指令。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”行为和值

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

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

像基础架构一样，扩展应当完全设计允许的行为和值，并使用术语“未指定的”用于有意不做约束的情况。在这些情况中，可以被其它的扩展、平台标准或实现约束或定义。

第二章 RV32I Base Integer Instruction Set, Version 2.1

This chapter describes the RV32I base integer instruction set.

RV32I was designed to be sufficient to form a compiler target and to support modern operating system environments. The ISA was also designed to reduce the hardware required in a minimal implementation. RV32I contains 40 unique instructions, though a simple implementation might cover the ECALL/EBREAK instructions with a single SYSTEM hardware instruction that always traps and might be able to implement the FENCE instruction as a NOP, reducing base instruction count to 38 total. RV32I can emulate almost any other ISA extension (except the A extension, which requires additional hardware support for atomicity).

In practice, a hardware implementation including the machine-mode privileged architecture will also require the 6 CSR instructions.

Subsets of the base integer ISA might be useful for pedagogical purposes, but the base has been defined such that there should be little incentive to subset a real hardware implementation beyond omitting support for misaligned memory accesses and treating all SYSTEM instructions as a single trap.

The standard RISC-V assembly language syntax is documented in the Assembly Programmer's Manual [1].

Most of the commentary for RV32I also applies to the RV64I base.

2.1 Programmers' Model for Base Integer ISA

Figure 2.1 shows the unprivileged state for the base integer ISA. For RV32I, the 32 `x` registers are each 32 bits wide, i.e., XLEN=32. Register `x0` is hardwired with all bits equal to 0. General

purpose registers x1–x31 hold values that various instructions interpret as a collection of Boolean values, or as two's complement signed binary integers or unsigned binary integers.

There is one additional unprivileged register: the program counter pc holds the address of the current instruction.

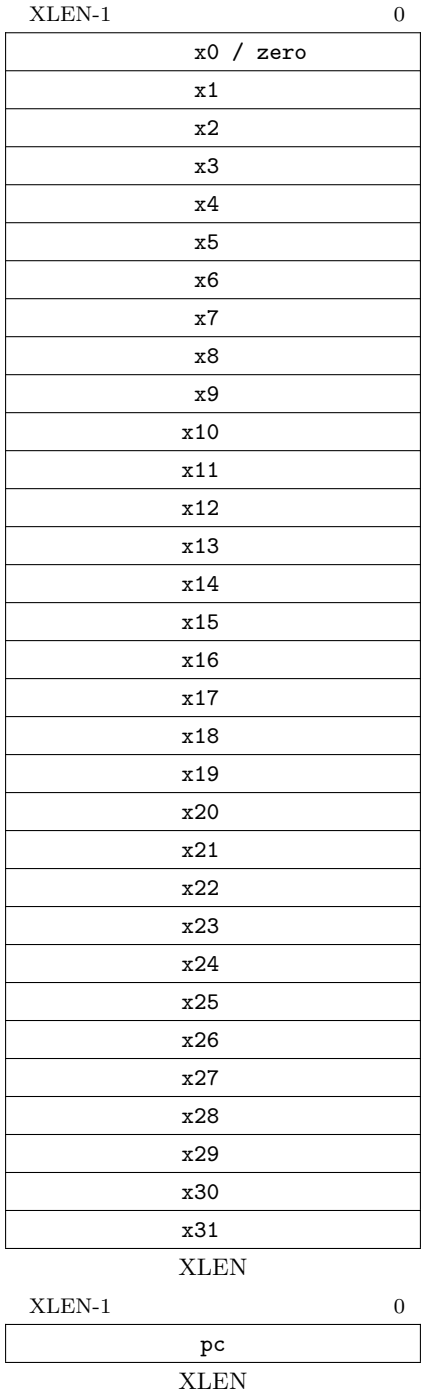


图 2.1: RISC-V base unprivileged integer register state.

There is no dedicated stack pointer or subroutine return address link register in the Base Integer ISA; the instruction encoding allows any `x` *register to be used for these purposes. However, the standard software calling convention uses register* `x1` *to hold the return address for a call, with register* `x5` *available as an alternate link register. The standard calling convention uses register* `x2` *as the stack pointer.*

Hardware might choose to accelerate function calls and returns that use `x1` *or* `x5`*. See the descriptions of the JAL and JALR instructions.*

The optional compressed 16-bit instruction format is designed around the assumption that `x1` *is the return address register and* `x2` *is the stack pointer. Software using other conventions will operate correctly but may have greater code size.*

The number of available architectural registers can have large impacts on code size, performance, and energy consumption. Although 16 registers would arguably be sufficient for an integer ISA running compiled code, it is impossible to encode a complete ISA with 16 registers in 16-bit instructions using a 3-address format. Although a 2-address format would be possible, it would increase instruction count and lower efficiency. We wanted to avoid intermediate instruction sizes (such as Xtensa's 24-bit instructions) to simplify base hardware implementations, and once a 32-bit instruction size was adopted, it was straightforward to support 32 integer registers. A larger number of integer registers also helps performance on high-performance code, where there can be extensive use of loop unrolling, software pipelining, and cache tiling.

For these reasons, we chose a conventional size of 32 integer registers for RV32I. Dynamic register usage tends to be dominated by a few frequently accessed registers, and regfile implementations can be optimized to reduce access energy for the frequently accessed registers [13]. The optional compressed 16-bit instruction format mostly only accesses 8 registers and hence can provide a dense instruction encoding, while additional instruction-set extensions could support a much larger register space (either flat or hierarchical) if desired.

For resource-constrained embedded applications, we have defined the RV32E subset, which only has 16 registers (Chapter **??***).*

2.2 Base Instruction Formats

In the base RV32I ISA, there are four core instruction formats (R/I/S/U), as shown in Figure 2.2. All are a fixed 32 bits in length. The base ISA has IALIGN=32, meaning that instructions must be aligned on a four-byte boundary in memory. An instruction-address-misaligned exception is generated on a taken branch or unconditional jump if the target address is not IALIGN-bit aligned. This exception is reported on the branch or jump instruction, not on the target instruction. No instruction-address-misaligned exception is generated for a conditional branch that is not taken.

The alignment constraint for base ISA instructions is relaxed to a two-byte boundary when instruction extensions with 16-bit lengths or other odd multiples of 16-bit lengths are added (i.e., IALIGN=16).

Instruction-address-misaligned exceptions are reported on the branch or jump that would cause instruction misalignment to help debugging, and to simplify hardware design for systems with IALIGN=32, where these are the only places where misalignment can occur.

The behavior upon decoding a reserved instruction is 未指定的.

Some platforms may require that opcodes reserved for standard use raise an illegal-instruction exception. Other platforms may permit reserved opcode space be used for non-conforming extensions.

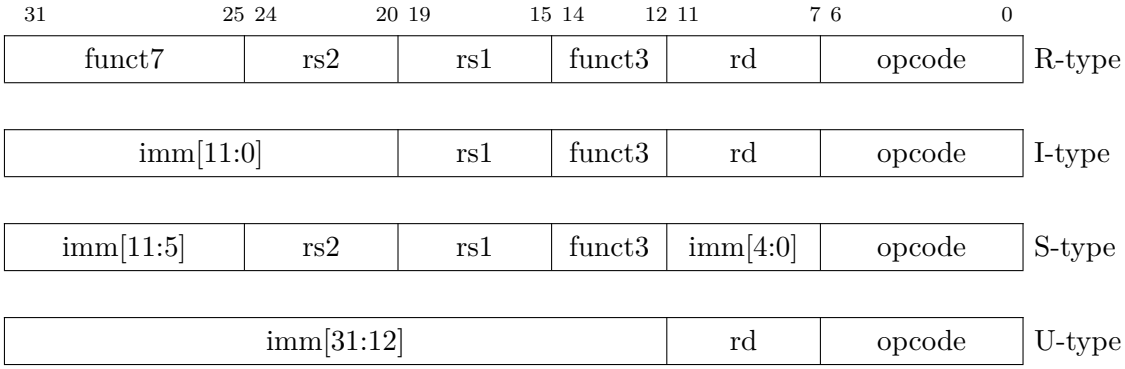


图 2.2: RISC-V base instruction formats. Each immediate subfield is labeled with the bit position (imm[x]) in the immediate value being produced, rather than the bit position within the instruction's immediate field as is usually done.

The RISC-V ISA keeps the source (*rs1* and *rs2*) and destination (*rd*) registers at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions (Chapter **??**), immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.

Decoding register specifiers is usually on the critical paths in implementations, and so the instruction format was chosen to keep all register specifiers at the same position in all formats at the expense of having to move immediate bits across formats (a property shared with RISC-IV aka. SPUR [8]).

In practice, most immediates are either small or require all XLEN bits. We chose an asymmetric immediate split (12 bits in regular instructions plus a special load-upper-immediate instruction with 20 bits) to increase the opcode space available for regular instructions.

Immediates are sign-extended because we did not observe a benefit to using zero-extension for some immediates as in the MIPS ISA and wanted to keep the ISA as simple as possible.

2.3 Immediate Encoding Variants

There are a further two variants of the instruction formats (B/J) based on the handling of immediates, as shown in Figure 2.3.

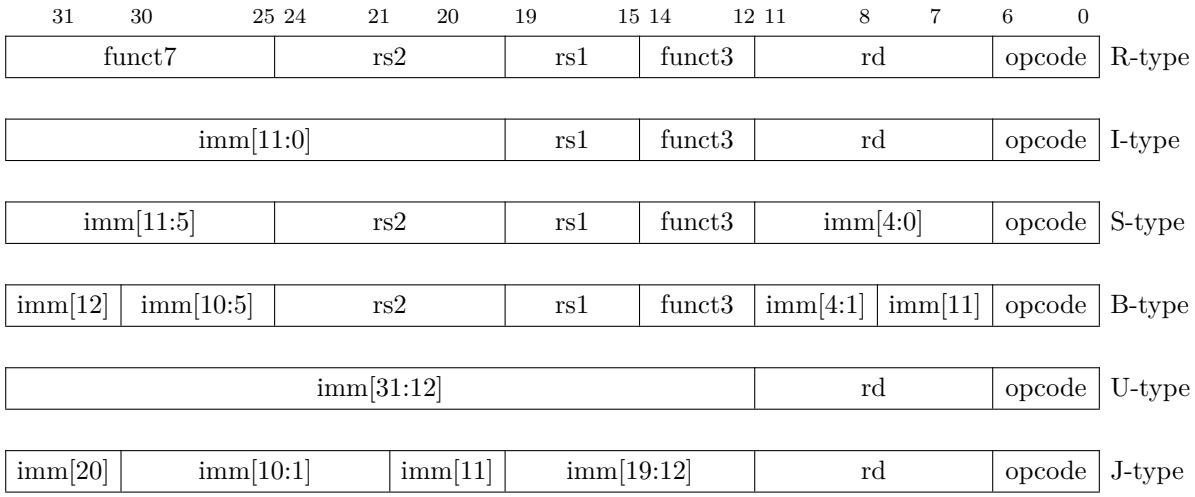


图 2.3: RISC-V base instruction formats showing immediate variants.

The only difference between the S and B formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware as is conventionally done, the middle bits (imm[10:1]) and sign bit stay in fixed positions, while the lowest bit in S format (inst[7]) encodes a high-order bit in B format.

Similarly, the only difference between the U and J formats is that the 20-bit immediate is shifted left by 12 bits to form U immediates and by 1 bit to form J immediates. The location of instruction bits in the U and J format immediates is chosen to maximize overlap with the other formats and with each other.

Figure 2.4 shows the immediates produced by each of the base instruction formats, and is labeled to show which instruction bit (inst[y]) produces each bit of the immediate value.

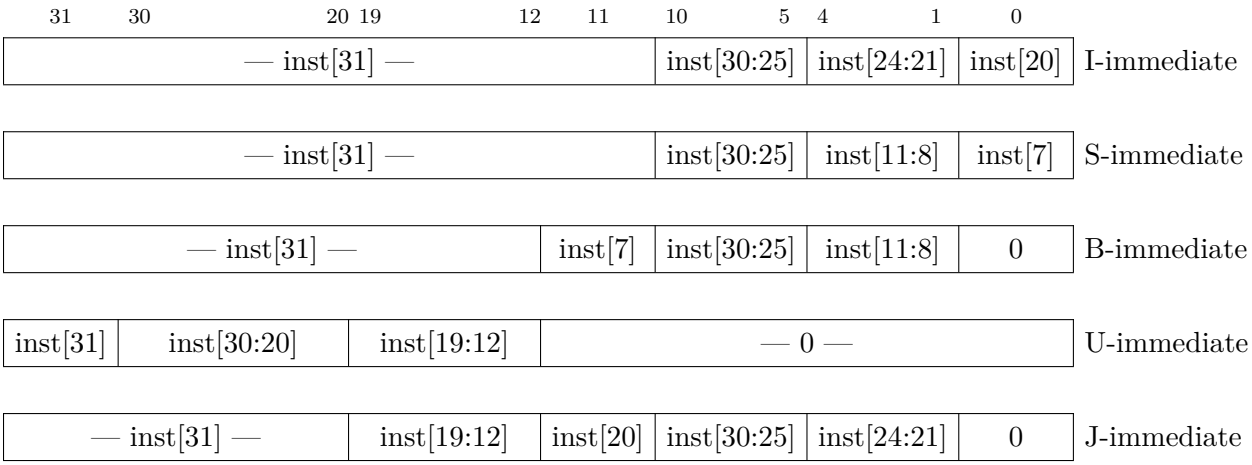


图 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

Sign-extension is one of the most critical operations on immediates (particularly for XLEN>32), and in RISC-V the sign bit for all immediates is always held in bit 31 of the instruction to allow sign-extension to proceed in parallel with instruction decoding.

Although more complex implementations might have separate adders for branch and jump calculations and so would not benefit from keeping the location of immediate bits constant across types of instruction, we wanted to reduce the hardware cost of the simplest implementations. By rotating bits in the instruction encoding of B and J immediates instead of using dynamic hardware muxes to multiply the immediate by 2, we reduce instruction signal fanout and immediate mux costs by around a factor of 2. The scrambled immediate encoding will add negligible time to static or ahead-of-time compilation. For dynamic generation of instructions, there is some small additional overhead, but the most common short forward branches have straightforward immediate encodings.

2.4 Integer Computational Instructions

Most integer computational instructions operate on XLEN bits of values held in the integer register file. Integer computational instructions are either encoded as register-immediate operations using the I-type format or as register-register operations using the R-type format. The destination is register *rd* for both register-immediate and register-register instructions. No integer computational instructions cause arithmetic exceptions.

We did not include special instruction-set support for overflow checks on integer arithmetic operations in the base instruction set, as many overflow checks can be cheaply implemented using RISC-V branches. Overflow checking for unsigned addition requires only a single additional branch instruction after the addition: `add t0, t1, t2; bltu t0, t1, overflow`.

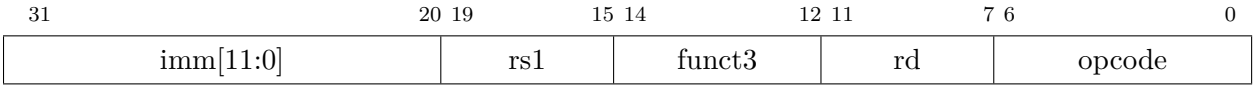
For signed addition, if one operand's sign is known, overflow checking requires only a single branch after the addition: `addi t0, t1, +imm; blt t0, t1, overflow`. *This covers the common case of addition with an immediate operand.*

For general signed addition, three additional instructions after the addition are required, leveraging the observation that the sum should be less than one of the operands if and only if the other operand is negative.

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

In RV64I, checks of 32-bit signed additions can be optimized further by comparing the results of ADD and ADDW on the operands.

Integer Register-Immediate Instructions



12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

ADDI adds the sign-extended 12-bit immediate to register *rs1*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI *rd, rs1, 0* is used to implement the MV *rd, rs1* assembler pseudoinstruction.

SLTI (set less than immediate) places the value 1 in register *rd* if register *rs1* is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to *rd*. SLTIU is similar but compares the values as unsigned numbers (i.e., the immediate is first sign-extended to XLEN bits then treated as an unsigned number). Note, SLTIU *rd, rs1, 1* sets *rd* to 1 if *rs1* equals zero, otherwise sets *rd* to 0 (assembler pseudoinstruction SEQZ *rd, rs*).

ANDI, ORI, XORI are logical operations that perform bitwise AND, OR, and XOR on register *rs1* and the sign-extended 12-bit immediate and place the result in *rd*. Note, XORI *rd, rs1, -1* performs a bitwise logical inversion of register *rs1* (assembler pseudoinstruction 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

Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 5 bits of the I-immediate field. The right shift type is encoded in bit 30. SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).

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 (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the 32-bit U-immediate value into the destination register *rd*, filling in the lowest 12 bits with zeros.

AUIPC (add upper immediate to `pc`) is used to build `pc`-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the address of the AUIPC instruction, then places the result in register *rd*.

The assembly syntax for `lui` *and* `auipc` *does not represent the lower 12 bits of the U-immediate, which are always zero.*

The AUIPC instruction supports two-instruction sequences to access arbitrary offsets from the `pc` *for both control-flow transfers and data accesses. The combination of an AUIPC and the 12-bit immediate in a JALR can transfer control to any 32-bit* `pc`*-relative address, while an AUIPC plus the 12-bit immediate offset in regular load or store instructions can access any 32-bit* `pc`*-relative data address.*

The current `pc` *can be obtained by setting the U-immediate to 0. Although a JAL +4 instruction could also be used to obtain the local* `pc` *(of the instruction following the JAL), it might cause pipeline breaks in simpler microarchitectures or pollute branch-target buffer structures in more complex microarchitectures.*

Integer Register-Register Operations

RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.

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 performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. Note, SLTU *rd*, *x0*, *rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudoinstruction SNEZ *rd, rs*). AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.

NOP Instruction

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

The NOP instruction does not change any architecturally visible state, except for advancing the `pc` and incrementing any applicable performance counters. NOP is encoded as ADDI *x0, x0, 0*.

NOPs can be used to align code segments to microarchitecturally significant address boundaries, or to leave space for inline code modifications. Although there are many possible ways to encode a NOP, we define a canonical NOP encoding to allow microarchitectural optimizations as well as

for more readable disassembly output. The other NOP encodings are made available for HINT instructions (Section 2.9).

ADDI was chosen for the NOP encoding as this is most likely to take fewest resources to execute across a range of systems (if not optimized away in decode). In particular, the instruction only reads one register. Also, an ADDI functional unit is more likely to be available in a superscalar design as adds are the most common operation. In particular, address-generation functional units can execute ADDI using the same hardware needed for base+offset address calculations, while register-register ADD or logical/shift operations require additional hardware.

2.5 Control Transfer Instructions

RV32I provides two types of control transfer instructions: unconditional jumps and conditional branches. Control transfer instructions in RV32I do *not* have architecturally visible delay slots.

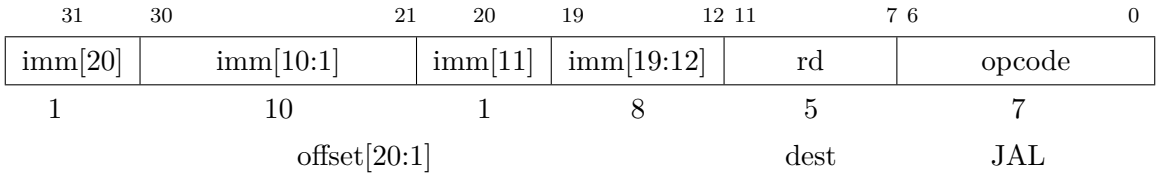
If an instruction access-fault or instruction page-fault exception occurs on the target of a jump or taken branch, the exception is reported on the target instruction, not on the jump or branch instruction.

Unconditional Jumps

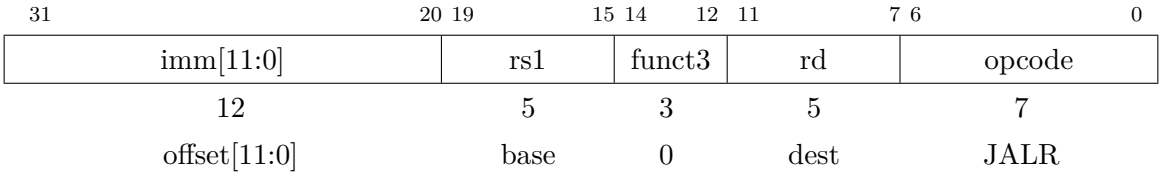
The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Jumps can therefore target a ±1 MiB range. JAL stores the address of the instruction that follows the JAL (`pc`+4) into register *rd*. The standard software calling convention uses `x1` as the return address register and `x5` as an alternate link register.

The alternate link register supports calling millicode routines (e.g., those to save and restore registers in compressed code) while preserving the regular return address register. The register `x5` *was chosen as the alternate link register as it maps to a temporary in the standard calling convention, and has an encoding that is only one bit different than the regular link register.*

Plain unconditional jumps (assembler pseudoinstruction J) are encoded as a JAL with *rd*=`x0`.



The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register *rs1*, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (pc+4) is written to register *rd*. Register x0 can be used as the destination if the result is not required.



The unconditional jump instructions all use pc*-relative addressing to help support position-independent code. The JALR instruction was defined to enable a two-instruction sequence to jump anywhere in a 32-bit absolute address range. A LUI instruction can first load* rs1 *with the upper 20 bits of a target address, then JALR can add in the lower bits. Similarly, AUIPC then JALR can jump anywhere in a 32-bit* pc*-relative address range.*

Note that the JALR instruction does not treat the 12-bit immediate as multiples of 2 bytes, unlike the conditional branch instructions. This avoids one more immediate format in hardware. In practice, most uses of JALR will have either a zero immediate or be paired with a LUI or AUIPC, so the slight reduction in range is not significant.

Clearing the least-significant bit when calculating the JALR target address both simplifies the hardware slightly and allows the low bit of function pointers to be used to store auxiliary information. Although there is potentially a slight loss of error checking in this case, in practice jumps to an incorrect instruction address will usually quickly raise an exception.

When used with a base rs1=x0*, JALR can be used to implement a single instruction subroutine call to the lowest 2 KiB or highest 2 KiB address region from anywhere in the address space, which could be used to implement fast calls to a small runtime library. Alternatively, an ABI could dedicate a general-purpose register to point to a library elsewhere in the address space.*

The JAL and JALR instructions will generate an instruction-address-misaligned exception if the target address is not aligned to an IALIGN-bit boundary.

Instruction-address-misaligned exceptions are not possible on machines with IALIGN=16, such as those that support the compressed instruction-set extension, C.

Return-address prediction stacks are a common feature of high-performance instruction-fetch units, but require accurate detection of instructions used for procedure calls and returns to be effective. For RISC-V, hints as to the instructions' usage are encoded implicitly via the register numbers used. A JAL instruction should push the return address onto a return-address stack (RAS) only when *rd* is x1 or x5. JALR instructions should push/pop a RAS as shown in the Table 2.1.

rd is x1/x5	*rs1* is x1/x5	*rd*=*rs1*	RAS action
No	No	–	None
No	Yes	–	Pop
Yes	No	–	Push
Yes	Yes	No	Pop, then push
Yes	Yes	Yes	Push

表 2.1: Return-address stack prediction hints encoded in the register operands of a JALR instruction.

Some other ISAs added explicit hint bits to their indirect-jump instructions to guide return-address stack manipulation. We use implicit hinting tied to register numbers and the calling convention to reduce the encoding space used for these hints.

*When two different link registers (*x1 *and* x5*) are given as* rs1 *and* rd*, then the RAS is both popped and pushed to support coroutines. If* rs1 *and* rd *are the same link register (either* x1 *or* x5*), the RAS is only pushed to enable macro-op fusion of the sequences:* `lui ra, imm20; jalr ra, imm12(ra)` *and* `auipc ra, imm20; jalr ra, imm12(ra)`

Conditional Branches

All branch instructions use the B-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2 bytes. The offset is sign-extended and added to the address of the branch instruction to give the target address. The conditional branch range is ±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

Branch instructions compare two registers. BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively. BLT and BLTU take the branch if *rs1* is less than *rs2*, using signed and unsigned comparison respectively. BGE and BGEU take the branch if *rs1* is greater than or equal to *rs2*, using signed and unsigned comparison respectively. Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

Signed array bounds may be checked with a single BLTU instruction, since any negative index will compare greater than any nonnegative bound.

Software should be optimized such that the sequential code path is the most common path, with less-frequently taken code paths placed out of line. Software should also assume that backward branches will be predicted taken and forward branches as not taken, at least the first time they are encountered. Dynamic predictors should quickly learn any predictable branch behavior.

Unlike some other architectures, the RISC-V jump (JAL with *rd*=`x0`) instruction should always be used for unconditional branches instead of a conditional branch instruction with an always-true condition. RISC-V jumps are also `pc`-relative and support a much wider offset range than branches, and will not pollute conditional-branch prediction tables.

The conditional branches were designed to include arithmetic comparison operations between two registers (as also done in PA-RISC, Xtensa, and MIPS R6), rather than use condition codes (x86, ARM, SPARC, PowerPC), or to only compare one register against zero (Alpha, MIPS), or two registers only for equality (MIPS). This design was motivated by the observation that a combined compare-and-branch instruction fits into a regular pipeline, avoids additional condition code state or use of a temporary register, and reduces static code size and dynamic instruction fetch traffic. Another point is that comparisons against zero require non-trivial circuit delay (especially after the move to static logic in advanced processes) and so are almost as expensive as arithmetic magnitude compares. Another advantage of a fused compare-and-branch instruction is that branches are observed earlier in the front-end instruction stream, and so can be predicted earlier. There is perhaps an advantage to a design with condition codes in the case where multiple branches can be taken based on the same condition codes, but we believe this case to be relatively rare.

We considered but did not include static branch hints in the instruction encoding. These can reduce the pressure on dynamic predictors, but require more instruction encoding space and software profiling for best results, and can result in poor performance if production runs do not match profiling runs.

We considered but did not include conditional moves or predicated instructions, which can effectively replace unpredictable short forward branches. Conditional moves are the simpler of the two, but are difficult to use with conditional code that might cause exceptions (memory accesses and floating-point operations). Predication adds additional flag state to a system, additional instructions to set and clear flags, and additional encoding overhead on every instruction. Both conditional move and predicated instructions add complexity to out-of-order microarchitectures, adding an implicit third source operand due to the need to copy the original value of the destination architectural register into the renamed destination physical register if the predicate is false. Also, static compile-time decisions to use predication instead of branches can result

in lower performance on inputs not included in the compiler training set, especially given that unpredictable branches are rare, and becoming rarer as branch prediction techniques improve.

We note that various microarchitectural techniques exist to dynamically convert unpredictable short forward branches into internally predicated code to avoid the cost of flushing pipelines on a branch mispredict [3, 7, 6] and have been implemented in commercial processors [12]. The simplest techniques just reduce the penalty of recovering from a mispredicted short forward branch by only flushing instructions in the branch shadow instead of the entire fetch pipeline, or by fetching instructions from both sides using wide instruction fetch or idle instruction fetch slots. More complex techniques for out-of-order cores add internal predicates on instructions in the branch shadow, with the internal predicate value written by the branch instruction, allowing the branch and following instructions to be executed speculatively and out-of-order with respect to other code [12].

The conditional branch instructions will generate an instruction-address-misaligned exception if the target address is not aligned to an IALIGN-bit boundary and the branch condition evaluates to true. If the branch condition evaluates to false, the instruction-address-misaligned exception will not be raised.

Instruction-address-misaligned exceptions are not possible on machines with IALIGN=16, such as those that support the compressed instruction-set extension, C.

2.6 Load and Store Instructions

RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers. RV32I provides a 32-bit address space that is byte-addressed. The EEI will define what portions of the address space are legal to access with which instructions (e.g., some addresses might be read only, or support word access only). Loads with a destination of `x0` must still raise any exceptions and cause any other side effects even though the load value is discarded.

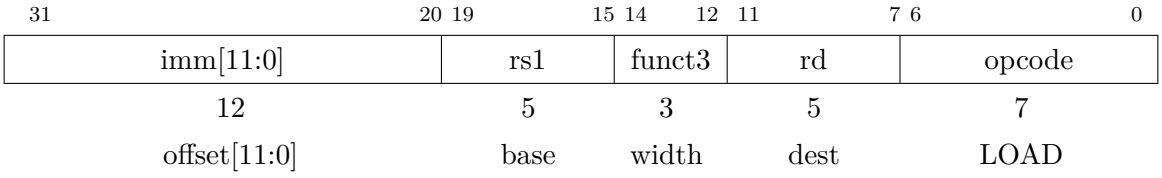
The EEI will define whether the memory system is little-endian or big-endian. In RISC-V, endianness is byte-address invariant.

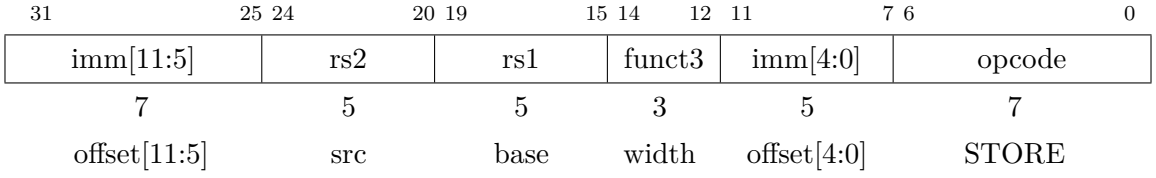
In a system for which endianness is byte-address invariant, the following property holds: if a byte is stored to memory at some address in some endianness, then a byte-sized load from that address in any endianness returns the stored value.

In a little-endian configuration, multibyte stores write the least-significant register byte at the lowest memory byte address, followed by the other register bytes in ascending order of their

significance. Loads similarly transfer the contents of the lesser memory byte addresses to the less-significant register bytes.

In a big-endian configuration, multibyte stores write the most-significant register byte at the lowest memory byte address, followed by the other register bytes in descending order of their significance. Loads similarly transfer the contents of the greater memory byte addresses to the less-significant register bytes.





Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective address is obtained by adding register *rs1* to the sign-extended 12-bit offset. Loads copy a value from memory to register *rd*. Stores copy the value in register *rs2* to memory.

The LW instruction loads a 32-bit value from memory into *rd*. LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in *rd*. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in *rd*. LB and LBU are defined analogously for 8-bit values. The SW, SH, and SB instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register *rs2* to memory.

Regardless of EEI, loads and stores whose effective addresses are naturally aligned shall not raise an address-misaligned exception. Loads and stores whose effective address is not naturally aligned to the referenced datatype (i.e., the effective address is not divisible by the size of the access in bytes) have behavior dependent on the EEI.

An EEI may guarantee that misaligned loads and stores are fully supported, and so the software running inside the execution environment will never experience a contained or fatal address-misaligned trap. In this case, the misaligned loads and stores can be handled in hardware, or via an invisible trap into the execution environment implementation, or possibly a combination of hardware and invisible trap depending on address.

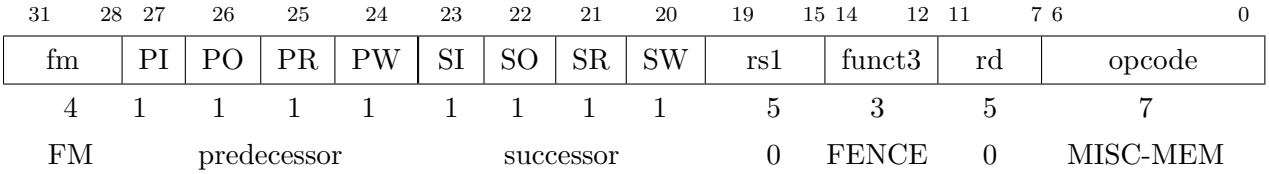
An EEI may not guarantee misaligned loads and stores are handled invisibly. In this case, loads and stores that are not naturally aligned may either complete execution successfully or raise an exception. The exception raised can be either an address-misaligned exception or an access-fault exception. For a memory access that would otherwise be able to complete except for the misalignment, an access-fault exception can be raised instead of an address-misaligned exception if the misaligned access should not be emulated, e.g., if accesses to the memory region have side effects. When an EEI does not guarantee misaligned loads and stores are handled invisibly, the EEI must define if exceptions caused by address misalignment result in a contained trap (allowing software running inside the execution environment to handle the trap) or a fatal trap (terminating execution).

Misaligned accesses are occasionally required when porting legacy code, and help performance on applications when using any form of packed-SIMD extension or handling externally packed data structures. Our rationale for allowing EEIs to choose to support misaligned accesses via the regular load and store instructions is to simplify the addition of misaligned hardware support. One option would have been to disallow misaligned accesses in the base ISAs and then provide some separate ISA support for misaligned accesses, either special instructions to help software handle misaligned accesses or a new hardware addressing mode for misaligned accesses. Special instructions are difficult to use, complicate the ISA, and often add new processor state (e.g., SPARC VIS align address offset register) or complicate access to existing processor state (e.g., MIPS LWL/LWR partial register writes). In addition, for loop-oriented packed-SIMD code, the extra overhead when operands are misaligned motivates software to provide multiple forms of loop depending on operand alignment, which complicates code generation and adds to loop startup overhead. New misaligned hardware addressing modes take considerable space in the instruction encoding or require very simplified addressing modes (e.g., register indirect only).

Even when misaligned loads and stores complete successfully, these accesses might run extremely slowly depending on the implementation (e.g., when implemented via an invisible trap). Furthermore, whereas naturally aligned loads and stores are guaranteed to execute atomically, misaligned loads and stores might not, and hence require additional synchronization to ensure atomicity.

We do not mandate atomicity for misaligned accesses so execution environment implementations can use an invisible machine trap and a software handler to handle some or all misaligned accesses. If hardware misaligned support is provided, software can exploit this by simply using regular load and store instructions. Hardware can then automatically optimize accesses depending on whether runtime addresses are aligned.

2.7 Memory Ordering Instructions



The FENCE instruction is used to order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V hart or external device can observe any operation in the *successor* set following a FENCE before any operation in the *predecessor* set preceding the FENCE. Chapter **??** provides a precise description of the RISC-V memory consistency model.

The FENCE instruction also orders memory reads and writes made by the hart as observed by memory reads and writes made by an external device. However, FENCE does not order observations of events made by an external device using any other signaling mechanism.

A device might observe an access to a memory location via some external communication mechanism, e.g., a memory-mapped control register that drives an interrupt signal to an interrupt controller. This communication is outside the scope of the FENCE ordering mechanism and hence the FENCE instruction can provide no guarantee on when a change in the interrupt signal is visible to the interrupt controller. Specific devices might provide additional ordering guarantees to reduce software overhead but those are outside the scope of the RISC-V memory model.

The EEI will define what I/O operations are possible, and in particular, which memory addresses when accessed by load and store instructions will be treated and ordered as device input and device output operations respectively rather than memory reads and writes. For example, memory-mapped I/O devices will typically be accessed with uncached loads and stores that are ordered using the I and O bits rather than the R and W bits. Instruction-set extensions might also describe new I/O instructions that will also be ordered using the I and O bits in a FENCE.

The fence mode field *fm* defines the semantics of the FENCE. A FENCE with *fm*=0000 orders all memory operations in its predecessor set before all memory operations in its successor set.

fm field	Mnemonic	Meaning
0000	*none*	Normal Fence
1000	TSO	With FENCE RW,RW: exclude write-to-read ordering Otherwise: *Reserved for future use.*
other		*Reserved for future use.*

表 2.2: Fence mode encoding.

The FENCE.TSO instruction is encoded as a FENCE instruction with *fm*=1000, *predecessor*=RW, and *successor*=RW. FENCE.TSO orders all load operations in its predecessor set before all memory operations in its successor set, and all store operations in its predecessor set before all store operations in its successor set. This leaves non-AMO store operations in the FENCE.TSO's predecessor set unordered with non-AMO loads in its successor set.

Because FENCE RW,RW imposes a superset of the orderings that FENCE.TSO imposes, it is correct to ignore the fm *field and implement FENCE.TSO as FENCE RW,RW.*

The unused fields in the FENCE instructions—*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. Likewise, many *fm* and predecessor/successor set settings in Table 2.2 are also reserved for future use. Base implementations shall treat all such reserved configurations as normal fences with *fm*=0000, and standard software shall use only non-reserved configurations.

We chose a relaxed memory model to allow high performance from simple machine implementations and from likely future coprocessor or accelerator extensions. We separate out I/O ordering from memory R/W ordering to avoid unnecessary serialization within a device-driver hart and also to support alternative non-memory paths to control added coprocessors or I/O devices. Simple implementations may additionally ignore the predecessor *and* successor *fields and always execute a conservative fence on all operations.*

2.8 Environment Call and Breakpoints

SYSTEM instructions are used to access system functionality that might require privileged access and are encoded using the I-type instruction format. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), and all other

potentially privileged instructions. CSR instructions are described in Chapter **??**, and the base unprivileged instructions are described in the following section.

The SYSTEM instructions are defined to allow simpler implementations to always trap to a single software trap handler. More sophisticated implementations might execute more of each system instruction in hardware.

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

These two instructions cause a precise requested trap to the supporting execution environment.

The ECALL instruction is used to make a service request to the execution environment. The EEI will define how parameters for the service request are passed, but usually these will be in defined locations in the integer register file.

The EBREAK instruction is used to return control to a debugging environment.

ECALL and EBREAK were previously named SCALL and SBREAK. The instructions have the same functionality and encoding, but were renamed to reflect that they can be used more generally than to call a supervisor-level operating system or debugger.

EBREAK was primarily designed to be used by a debugger to cause execution to stop and fall back into the debugger. EBREAK is also used by the standard gcc compiler to mark code paths that should not be executed.

Another use of EBREAK is to support "semihosting", where the execution environment includes a debugger that can provide services over an alternate system call interface built around the EBREAK instruction. Because the RISC-V base ISAs do not provide more than one EBREAK instruction, RISC-V semihosting uses a special sequence of instructions to distinguish a semihosting EBREAK from a debugger inserted EBREAK.

```
slli x0, x0, 0x1f   # Entry NOP
ebreak              # Break to debugger
srai x0, x0, 7      # NOP encoding the semihosting call number 7
```

Note that these three instructions must be 32-bit-wide instructions, i.e., they mustn't be among the compressed 16-bit instructions described in Chapter **??**.

The shift NOP instructions are still considered available for use as HINTs.

Semihosting is a form of service call and would be more naturally encoded as an ECALL using an existing ABI, but this would require the debugger to be able to intercept ECALLs, which is a newer addition to the debug standard. We intend to move over to using ECALLs with a standard ABI, in which case, semihosting can share a service ABI with an existing standard.

We note that ARM processors have also moved to using SVC instead of BKPT for semihosting calls in newer designs.

2.9 HINT Instructions

RV32I reserves a large encoding space for HINT instructions, which are usually used to communicate performance hints to the microarchitecture. Like the NOP instruction, HINTs do not change any architecturally visible state, except for advancing the `pc` and any applicable performance counters. Implementations are always allowed to ignore the encoded hints.

Most RV32I HINTs are encoded as integer computational instructions with *rd*=`x0`. The other RV32I HINTs are encoded as FENCE instructions with a null predecessor or successor set and with *fm*=0.

These HINT encodings have been chosen so that simple implementations can ignore HINTs altogether, and instead execute a HINT as a regular instruction that happens not to mutate the architectural state. For example, ADD is a HINT if the destination register is `x0`*; the five-bit* rs1 *and* rs2 *fields encode arguments to the HINT. However, a simple implementation can simply execute the HINT as an ADD of* rs1 *and* rs2 *that writes* `x0`*, which has no architecturally visible effect.*

As another example, a FENCE instruction with a zero pred *field and a zero* fm *field is a HINT; the* succ*,* rs1*, and* rd *fields encode the arguments to the HINT. A simple implementation can simply execute the HINT as a FENCE that orders the null set of prior memory accesses before whichever subsequent memory accesses are encoded in the* succ *field. Since the intersection of the predecessor and successor sets is null, the instruction imposes no memory orderings, and so it has no architecturally visible effect.*

Table 2.3 lists all RV32I HINT code points. 91% of the HINT space is reserved for standard HINTs. The remainder of the HINT space is designated for custom HINTs: no standard HINTs will ever be defined in this subspace.

We anticipate standard hints to eventually include memory-system spatial and temporal locality hints, branch prediction hints, thread-scheduling hints, security tags, and instrumentation flags for simulation/emulation.

Instruction	Constraints	Code Points	Purpose
LUI	*rd*=x0	2^{20}	*Reserved for future standard use*
AUIPC	*rd*=x0	2^{20}	
ADDI	*rd*=x0, and either *rs1*≠x0 or *imm*≠0	$2^{17}-1$	
ANDI	*rd*=x0	2^{17}	
ORI	*rd*=x0	2^{17}	
XORI	*rd*=x0	2^{17}	
ADD	*rd*=x0, *rs1*≠x0	$2^{10}-32$	
ADD	*rd*=x0, *rs1*=x0, *rs2*≠x2–x5	28	
ADD	*rd*=x0, *rs1*=x0, *rs2*=x2–x5	4	(*rs2*=x2) NTL.P1 (*rs2*=x3) NTL.PALL (*rs2*=x4) NTL.S1 (*rs2*=x5) NTL.ALL
SUB	*rd*=x0	2^{10}	*Reserved for future standard use*
AND	*rd*=x0	2^{10}	
OR	*rd*=x0	2^{10}	
XOR	*rd*=x0	2^{10}	
SLL	*rd*=x0	2^{10}	
SRL	*rd*=x0	2^{10}	
SRA	*rd*=x0	2^{10}	
FENCE	*rd*=x0, *rs1*≠x0, *fm*=0, 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}	*Designated for custom use*
SLTIU	*rd*=x0	2^{17}	
SLLI	*rd*=x0	2^{10}	
SRLI	*rd*=x0	2^{10}	
SRAI	*rd*=x0	2^{10}	
SLT	*rd*=x0	2^{10}	
SLTU	*rd*=x0	2^{10}	

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] David A. Patterson and Carlo H. Séquin. RISC I: A reduced instruction set VLSI computer. In *ISCA*, pages 443–458, 1981.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.