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 30 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。

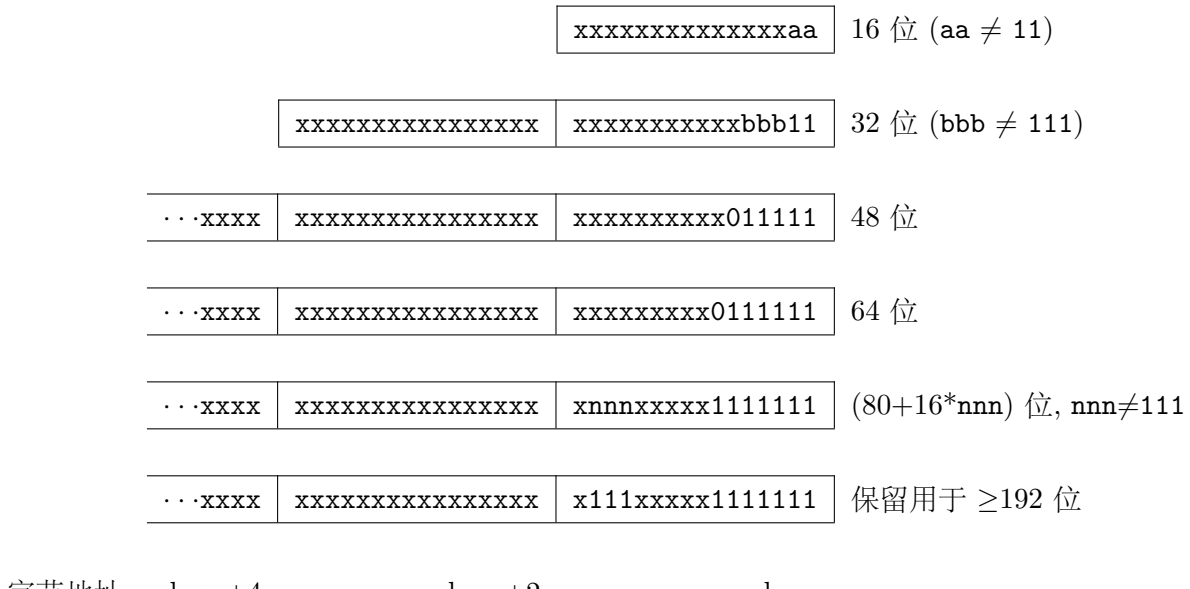


图 1.1: RISC-V 指令长度编码。此时只有 16 位和 32 位编码被认为是被冻结的。

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 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。

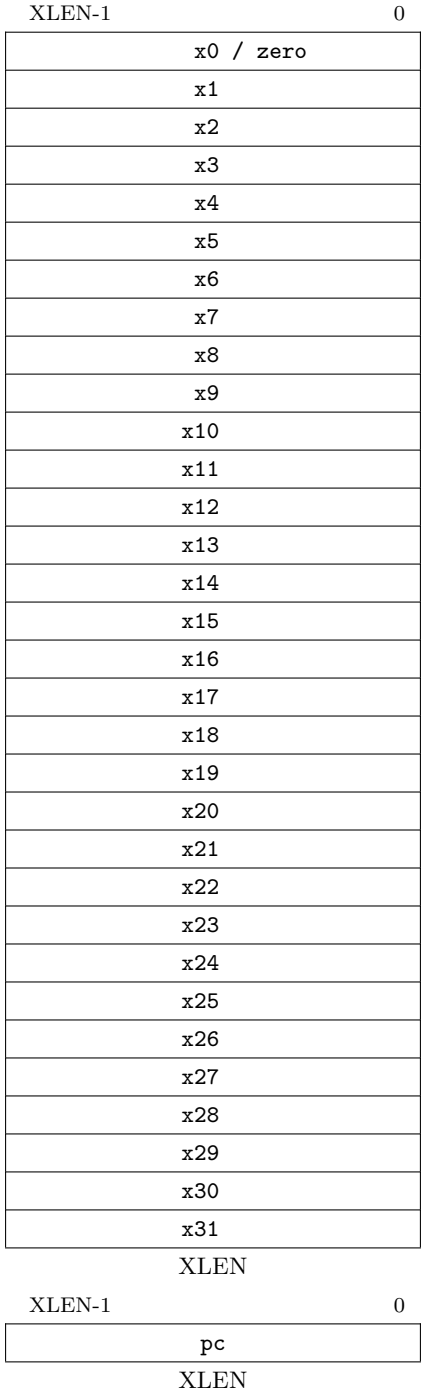


图 2.1: RISC-V 基础非特权整数寄存器状态

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 *[13]*。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 **??**）。

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

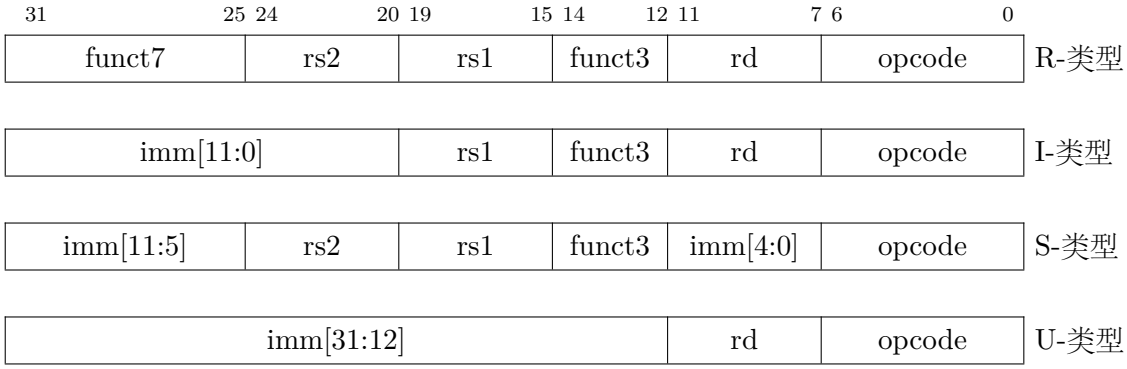


图 2.2: RISC-V 基础指令格式。每个立即数子域都用正被产生的立即数值中的位位置（imm[x]）的标签标记，而不是像通常做的那样，用指令立即数域中的位位置。

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 [8]*）。

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。

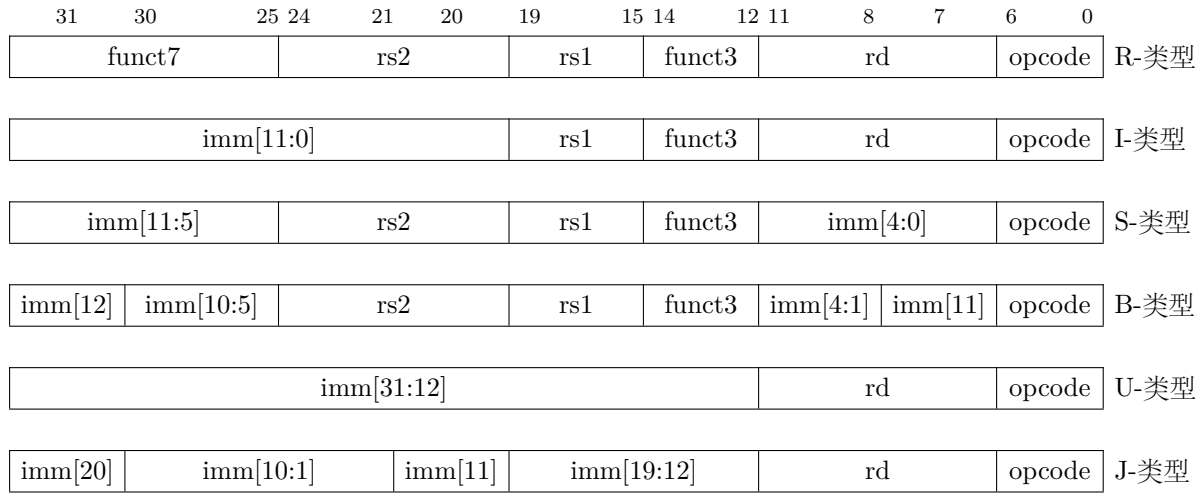


图 2.3: 显式立即数的 RISC-V 基础指令格式。

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

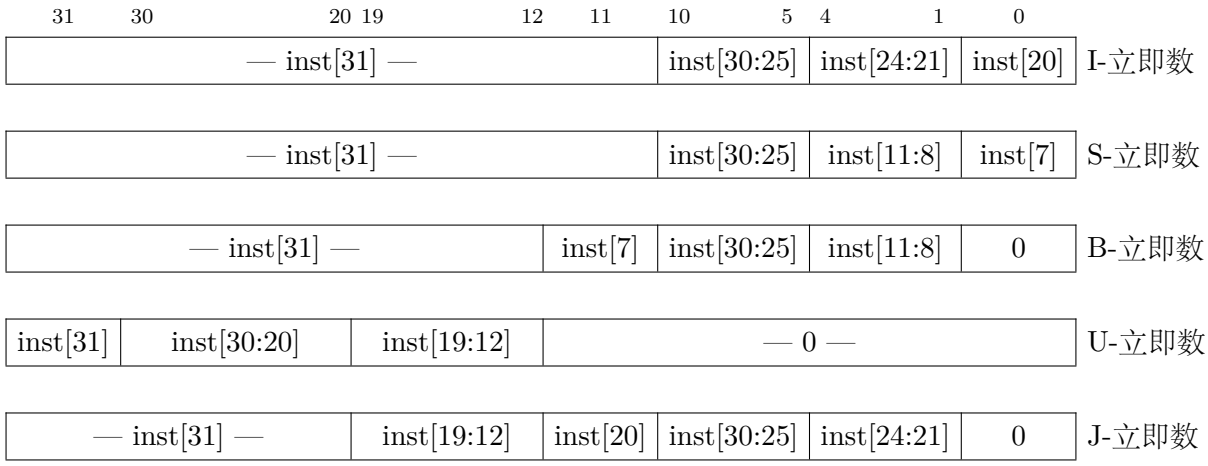


图 2.4: 由 RISC-V 指令产生的立即数的类型。用构造了它们值的指令位对域进行了标记。符号扩展总是使用 inst[31]。

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

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

用器（*MUX*），来将立即数扩大 *2* 倍，我们减少了大约一半的指令符号扇出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。

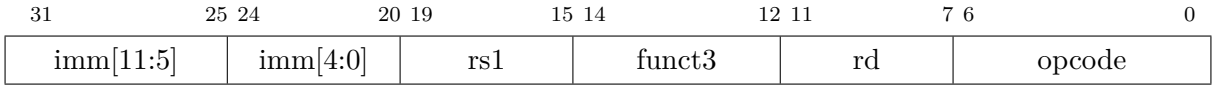
整数寄存器 - 立即数指令

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。

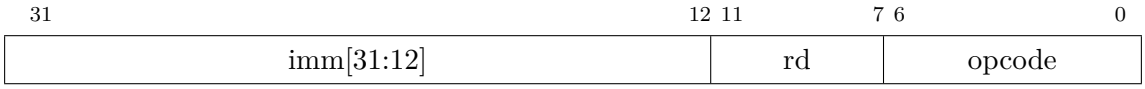
如果寄存器 *rs1* 小于符号扩展的立即数（当二者都被视为有符号数时），SLTI（小于立即数时置 1）指令把值 1 放到寄存器 *rd* 中；否则，该指令把 0 写入 *rd* 中。SLTIU 与之相似，但是将两个值作为无符号数比较（也就是说，前者会把立即数按符号扩展到 XLEN 位，而后者会将其视为无符号数）。注意，如果 *rs1* 等于 0，那么 SLTIU *rd, rs1, 1* 会把 *rd* 设置为 1，否则会把 *rd* 设置为 0（汇编器伪指令 SEQZ *rd, rs*）。

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*）。



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）。



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。

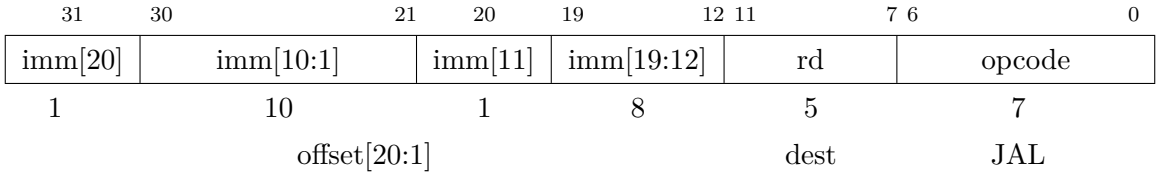
如果在一次跳转或发生转移的分支上发生了一个指令访问故障异常或指令缺页故障异常，该异常会报告在目标指令上，而不是报告在跳转或分支指令上。

无条件跳转

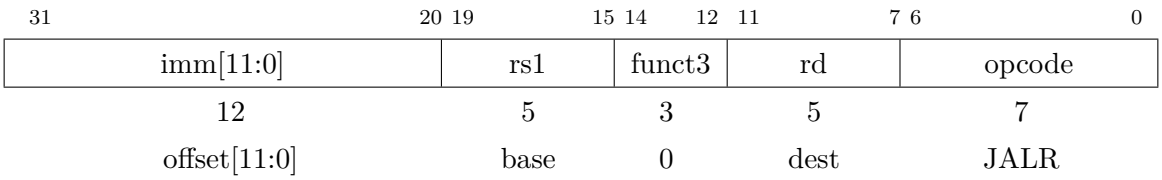
跳转和链接（JAL）指令使用 J 类型格式。J 类型指令把 J 立即数以 2 字节的倍数编码一个有符号的偏移量。偏移量是符号扩展的，加到当前跳转指令的地址上以形成跳转目标地址。跳转可以因此到达的目标范围是 ±1 MiB。JAL 把跟在 JAL 之后的指令的地址（pc+4）存储到寄存器 *rd* 中。标准软件调用约定使用 x1 作为返回地址寄存器，使用 x5 作为备用的链接寄存器。

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。



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



无条件跳转指令都使用 pc 相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：

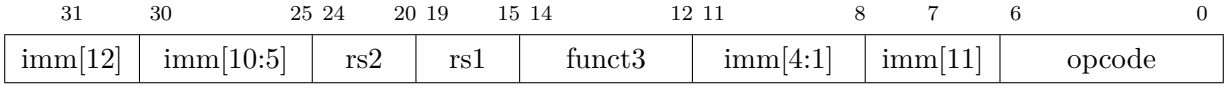
`lui ra, imm20; jalr ra, imm12(ra)` 和 `auipc ra, imm20; jalr ra, imm12(ra)`

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。



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 *[3, 7, 6]*，并且已经在商业处理器中被实现 *[12]*。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 *[12]*。

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

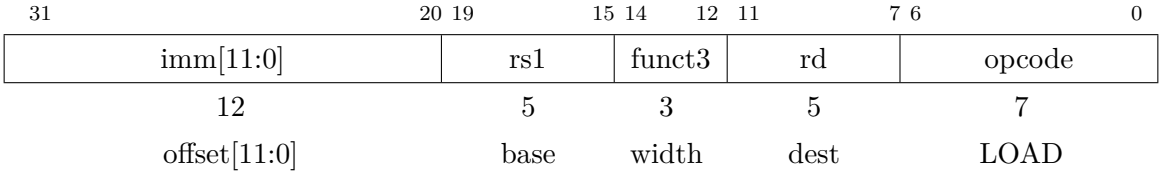
RV32I 是一个“加载-存储”架构，那里只有加载和存储指令访问内存，而算数指令只操作 CPU 寄存器。RV32I 提供一个 32 位的地址空间，按字节编址。EEI 将定义该地址空间的哪一部分是哪个指令可以合法访问的（例如，一些地址可能是只读的，或者只支持按字访问）。即使所加载的值被丢弃，以 `x0` 为目的的加载仍然必须要引发有可能的任何异常，或其它的副作用。

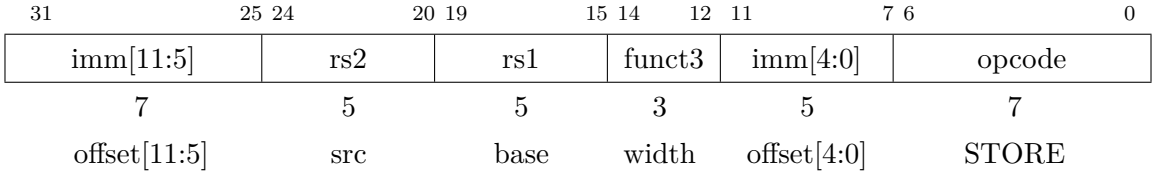
EEI 将定义内存系统是否是小字节序或大字节序的。在 RISC-V 中，字节序是按字节编址的不变量。

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。





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 **??**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*=x0	2^{20}	保留供未来标准使用
AUIPC	*rd*=x0	2^{20}	
ADDI	*rd*=x0, 并且要么 *rs1*≠x0 要么 *imm*≠0	$2^{17}-1$	
ANDI	*rd*=x0	2^{17}	
ORI	*rd*=x0	2^{17}	
XORI	*rd*=x0	2^{17}	
ADD	*rd*=x0, *rs1*≠x0	$2^{10}-32$	
ADD	*rd*=x0, *rs1*=x0, *rs2*≠x2–x5	28	
ADD	*rd*=x0, *rs1*=x0, *rs2*=x2–x5	4	(*rs2*=x2) NTL.P1 (*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}	
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	暂停
SLTI	*rd*=x0	2^{17}	指定供自定义使用
SLTIU	*rd*=x0	2^{17}	
SLLI	*rd*=x0	2^{10}	
SRLI	*rd*=x0	2^{10}	
SRAI	*rd*=x0	2^{10}	
SLT	*rd*=x0	2^{10}	
SLTU	*rd*=x0	2^{10}	

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.