第一章 介绍

RISC-V（念作“risc five”）是一个新型的指令集体系结构（ISA），最初它的目的是用于计算机体系结构的研究和教育，但现在我们希望它也能作为一种供工业界用于实现（implementation）的开源免费体系结构。我们在定义 RISC-V 时的目的包括：

- 一个完全开源的 ISA，可供学术、工业界免费使用。
- 一个真正能直接适用于原生硬件实现的 ISA，而不仅仅只是仿真或二进制翻译（binary translation）。
- 避免过度倾向某种特定的微架构风格（如微码、顺序、解耦、乱序）或者实现技术（如 full-custom、ASIC、FPGA），但能高效实现其中任意一种。
- 从 ISA 中分割出一个小的基础整数 ISA，可用于定制加速器（accelerator）或教育目的；以及可选的标准扩展，用于支持通用软件开发。
- 支持修订后的 IEEE-754 2008 浮点标准。
- 支持广泛的 ISA 扩展，及特殊变体。
- 同时支持 32bit 和 64bit 地址空间的变体，便于应用、操作系统内核或硬件的实现。
- 支持高度并行的多核，或者众核的实现，包括异构多核处理器。
- 可选的变长指令，用于对指令编码空间进行扩展，以及通过支持可选的幂集指令编码来改善性能，静态代码大小以及能效。
- 一个完全可虚拟化的 ISA，以简化 hypervisor 的开发。
- 让新型特权架构的设计实践更简单。

我们关于设计决策的评注格式如本段所示，若读者仅对 specification（规范）本身感兴趣，则可跳过这些非规范的文本。

RISC-V 这个名字代表 UC Berkeley 的第五个主要的 RISC ISA 设计（前四个分别是 RISC-I、RISC-II、SOAR、SPUR）。我们还用罗马数字“V”来表示“变体(Variations)”和向量“Vectors”，以及对一系列体系结构研究的支持，包括各种数据并行加速器，就是这个 ISA 设计的直接目标。

RISC-V 的定义会尽可能地规避实现（implementation）细节（尽管在取决于实现（implementation-driven）的决策中包含了注释），应当将其理解为各种实现中的软件可见接口，而非特定的硬件设计。RISC-V manual 分为两卷，本卷介绍基础的非权限设计，其中包括可选的非权限 ISA 扩展。非特权指令通常在所有特权架构

下的所有特权模式下都是可用的，尽管行为可能因特权模式或者特权架构而异。第二卷中提供了首个（“classic”的）特权体系结构设计。本 manuals 采用 IEC 80000-13:2008 conventions，1 Byte 代表 8 bits。

我们会试图在非权限 ISA 的设计中，消除任何关于特定微架构特性（如 cache line 的 size）或特权架构细节（如 page translation）的依赖。这么做既是为了简洁性，也是为了在各个微架构或特权架构之间最大化它的灵活性。

1.1 RISC-V 硬件平台术语（RISC-V Hardware Platform Terminology）

一个 RISC-V 硬件平台（hardware platform）包含一或多个兼容 RISC-V 的处理核心，也可以包含其它不兼容 RISC-V 的核心，固定功能的加速器（accelerator），各种物理内存（memory）结构，I/O 设备，以及用于组件通信的互连结构（interconnect structure）。

如果一个组件（component）包含了一个独立的取指单元（instruction fetch unit），则将其称之为一个**核心（core）**。兼容 RISC-V 的核心可通过多线程来支持多个兼容 RISC-V 的硬件线程，或者说 harts。

RISC-V 核心也能带有额外的专用指令集扩展，或者协处理器（coprocessor）。我们用术语**“协处理器（coprocessor）”**来代指那些与 RISC-V 核心相连接且主要由 RISC-V 指令流排序（sequenced by），但也具有额外体系结构状态与指令集扩展的单元；另外，它们也可能会有一定的受限独立权（相对于 RISC-V 指令流）。

我们用术语**加速器（accelerator）**来代指那些不可编程、功能固定的单元，或者能够自主运行但也专用于某些特定任务的核心。可以预见在 RISC-V 系统中，将会出现许多基于 RISC-V 核心的可编程加速器（带特定的指令集扩展，或者协处理器）。例如，I/O 加速器就是一类重要的 RISC-V 加速器，它们负责减轻主应用程序核心中的 I/O 处理任务负载。

RISC-V 硬件平台的系统级组织结构（system-level organization）可小可大，小到单核心微控制器，大到在共享内存众核服务器节点中能含有数千个节点的集群（cluster）。当然即便是小型的片上系统也可能会具有多计算机或多处理器的层次结构，以便模块化开发，或在子系统间提供安全隔离。

1.2 RISC-V 软件执行环境与 Harts（RISC-V Software Execution Environments and Harts）

RISC-V 程序的行为取决于它运行时所在的执行环境。

RISC-V 执行环境接口（EEI, execution environment interface）定义了程序的初始状态、环境中 hart 的数量与类型（包括 hart 所支持的特权模式，可访问的内存、I/O 区域，所有合法指令在每个 hart 上的执行行为（即，ISA 实际是 EEI 的一个组件）），以及该如何处理在执行期间引发的任何中断（intrrupt）和异常（exception）（包括环境调用（environment call））。EEI 的例子包括 Linux 的 ABI（Application Binary Interface），或者 RISC-V 的 SBI（Supervisor Binary Interface）。RISC-V 的执行环境既可以是纯硬件，也可以是纯软件的，亦或者是两者的结合。例如，可以通过操作码陷阱（opcode trap）和软件仿真来实现硬件中没有提供的功能。执行环境的实现示例包括：

- **“裸机（bare metal）”**硬件平台，其中的 hart 直接由处理器物理线程实现，指令能完整访问物理地址空间，硬件平台定义了上电重启（power-on reset）后初始的执行环境。
- **RISC-V 操作系统（RISC-V operating systems）**：将用户级 hart 多路复用到可用的处理器物理线程上，并通过虚拟内存限制内存访问，提供多个 user-level 的执行环境。
- **RISC-V 虚拟机监视器（RISC-V hypervisors）**，为 guest operating system 提供多个 supervisor-level 的执行环境。
- **RISC-V 仿真器（RISC-V emulators）**，如 Spike、QEMU 和 rv8，它们可在 x86 系统上对 RISC-V 的 hart 进行仿真，能提供 user-level 和 supervisor-level 的执行环境。

可将裸硬件平台(bare hardware platform)看作是 EEI 的定义,其中包括可访问的 hart, memory, 及其它环境中的设备, 还有上电重启后的初始状态。大多软件在通常情况下都是基于较高抽象程度的硬件接口来设计的, 从而通过更抽象的 EEI 获得较好的跨硬件平台移植性。EEI 通常是层层相叠的, 其中较高级的 EEI 使用另一个较低级的 EEI。

从（给定执行环境下运行的）软件视角来看，hart 就是在执行环境中，用于自动获取并执行 RISC-V 指令的资源（resource）。这样来看，hart 的行为是比较类似于硬件线程资源的，当然它们实际上是被执行环境时间复用到硬件上的。通过一些

EEI 支持，可以对 hart 进行额外创建（creation）与销毁（destruction），例如通过环境调用来 fork 出新的 hart。

执行环境负责确保每个 hart 都能最终向前进展（forward progress）。对于给定 hart，它的责任（responsibility）会在其执行了一个显式等待某事件的机制时候暂停，例如在在本 specification 的 Volume II 中定义的中断等待指令（wait-for-intrrupt）；然后，在 hart 终止（terminated）时，相应的责任也会结束。以下事件构成了进展前进（forward progress）：

- 一个指令的退休（retirement）
- 一个 trap（在 Section 1.6 中定义）
- 构成进展前进的任何其它被定义事件

hart 一词引自 Lithe 的工作，这是个用于代表执行资源抽象的术语（而非软件线程的编程抽象）。

硬件线程（hardware thread，hart）与软件线程上下文（context）的重要区别在于，执行环节中运行的软件并不担保每个 hart 的进展：因为这是外部执行环境的责任。因而，从执行环境中软件的角度来看，环境的 hart 运行起来就跟硬件线程一样。

具体实现的执行环境可能会把执行环境中数量较少的 host hart 分时复用给更多的 guest hart，但必须保证 guest hart 运行起来就跟独立硬件线程一样。特别是，guest hart 多于 host hart 时，执行环境应当确保能够抢占（preempt）guest hart，以防无期限等待 guest hart 上的 guest software“让出”guest hart 控制权。

1.3 RISC-V ISA 概况（RISC-V ISA Overview）

可将一个 RISC-V ISA 定义为一个基础整数 ISA（base integer ISA），任何实现中都必须要有这个基础整数 ISA。基础整数 ISA 与早期 RISC 处理器中的十分相似，但没有分支延迟槽（branch delay slot），并且支持可选的变长指令编码（variable-length instruction encodings）。

base（基/基础/基本/基础指令/基本指令）是一组由精心挑选的指令所构成的最小指令集（minimal set of instruction），这个 base 足以合理作为编译器、汇编程序、链接器和操作系统（带有额外的权级操作）的目标（target），从而提供了一个非常便利的 ISA 和软件工具链“骨架”，能够围绕它构建出更多定制的处理器 ISA。

尽管在谈论时直接说“RISC-V”很方便，但 RISC-V 实际上是由一族相关的 ISA 所构成的，目前其中有 4 个基础指令集（base ISAs），它们的区别在于整数寄存器的宽度（width），相应的地址空间大小，以及整数寄存器的数目。

第 2、5 章介绍了两个主要的基础整数变体：RV32I 和 RV64I，它们分别提供

了 32bit 和 64bit 的地址空间。**我们用“XLEN”一词代指整数寄存器的宽度（32 或 64），其单位是 bit。**

第 4 章介绍了 RV32I 基础指令的子集变体：RV32E，它是面向小型微控制器的，其缩减了一半的整数寄存器数量也。

第 6 章勾画了面向未来的基础整数指令集变体 RV128I，其支持 128bit 的扁平地址空间（XLEN=128）。

基础整数指令集采用二的补码（2's complement）来表示有符号整数（sigend integer value）。

尽管 64bit 地址空间是大型系统的一项需要，但我们仍坚信 32bit 的地址空间足以在未来的几十年内继续胜任众多嵌入式与客户端设备的需要，并且还能够降低内存流量（traffic）与能耗。另外对于教育目的，32bit 的地址空间也是足够的。
更大的扁平 128bit 地址空间最终也有可能会被用到，我们会确保它能够在 RISC-V 框架下实现。

RISC-V 中有 4 个不同（distinct）的基础 ISA。一个常见问题是，**为什么没有单一的 ISA，尤其是为什么 RV32I 不是 RV64I 的严格子集？**一些早期的 ISA（SPARC，MIPS）在扩展地址空间时，会确保原先的 ISA 是新 ISA 的严格子集，以保证已有的 32bit 二进制文件还能继续在新的 64bit 硬件上运行。

将基础 ISA 显式分离的主要优点是，每个基础 ISA 都能根据自己的需要进行优化，而无需考虑是否支持其它基础 ISA 的操作。例如，RV64I 可以略去那些在 RV32I 中用于处理较窄寄存器的指令，以及各种 CSR。相对地，RV32I 的变体也能更灵活的利用编码空间。
不设计成单个 ISA 的一个主要缺点是，在一个基础 ISA 上模拟另一个 ISA（如在 RV64I 上模拟 RV32I）所需的硬件将更复杂。

然而，对于所有情况，出于寻址与非法指令 trap 的差异，即便用的是完整的超集（superset）指令编码，也同样需要硬件进行一些模式切换（mode switch），况且 RISC-V 的不同基础 ISA 非常相似，因此同时支持多个不同版本的成本相对更低。

尽管有人提出，可以通过严格的超集设计允许传统的 32bit 库与 64bit 的代码链接（link），但考虑到软件调用约定与系统调用接口的差异，这在实践中依旧不切实际，就算采用兼容编码也是如此。

RISC-V 特权架构（privileged architecture）会在 misa 中提供相关的字段（fields），以在每个级别（level）上控制非特权 ISA，从而支持在同一硬件上对不同的基础 ISA 进行模拟。值得一提的是，较新的 SPARC 和 MIPS ISA 修订版中并不建议在 64bit 系统上直接运行不作任何修改的 32bit 代码。

【（这一段原文直翻译起来太难受了，而且相对不重要所以直接转述一遍，大意就是：与此相关的一个问题是：**为什么 RV32I 和 RV64I 中负责 32bit 加法的是编码不同的两条指令？**ADD 指令的功能是进行 XLEN bit 的加法，因此它在 RV32I 中的功能是 32bit 加法（XLEN=32）；在 RV64I 中的功能是 64bit 加法（XLEN=64），要在 RV64I 中进行 32bit 加法，就要用到 RV64I 中新增的指令 ADDW，它的才是 RV64I 中的 32bit 加法。然后又提了一下 LOAD 指令，RV32I 和 RV64I 中 LOAD 指令的功能又是一致的，由 LOAD 指令中的 width 域来指定加载数据的宽度（8、16、32），因此 RV32I 和 RV64I 中用于加载 32bit 数的指令仍然是同一条指令（LW），RV64I 中给 LOAD 新增了另一个 width，用于加载 64bit 数据。然后回到了前面的问题，为什么 ADD 指

令没有像 LOAD 那样统一 功能-编码 关系呢？RISC-V 的最早期版本中确实存在这种可选设计变体，但随后在 2011 年 1 月就改成了现在这个样子)】

我们设计的出发点，是在 64bit 的 ISA 中支持 32bit 整数，而并非确保其与 32bit ISA 的兼容性，从而避免 *W 后缀的不对称性（例如 ADD 有 ADDW，但是 AND 就没有 ANDW）。

事后来看，这样的理由可能并不充分，因为它们实际上是两个同时设计的 ISA，而非其中一者建立在另一者基础之上。当然这还有另外一个原因，就是我们得把平台的要求给纳入到 ISA spec 中，即 RV64I 要有全部的 RV32I 指令。况且，要改变编码的话也已经晚了，并且出于上述原因也没什么实际意义。

值得注意的是，我们可以通过将 *W 变体作为 RV32I 系统的一个扩展，来实现在 RV64I 和未来的 RV32I 变体间通用的编码。

RISC-V 广泛支持各种定制化（customization）与规范（specification）。每个基础整数 ISA 都可搭配一或多个可选的指令集扩展。扩展可能被归类为标准（standard），定制（custom）或者非规范（non-conforming）三类之一。**为此，我们将每个 RISC-V 指令集编码空间（及其相关的其它编码空间，如 CSRs）划分为了三个正交的类别：标准（standard）、保留（reserve）、自定义（custom）。**

标准扩展与编码（standard extensions and encodings）皆由基金会定义；任何并非基金会定义的扩展都是不标准的（non-standard）。每个 base ISA 和它的标准扩展都仅会使用标准编码，并且这些编码的使用不得冲突。

保留编码（reserved encodings）暂时未定义，保留给未来的标准扩展；一旦因此而使用（即用于标准扩展），其就会立即成为标准编码。

自定义编码（custom encodings）不会被用于标准扩展，它们留给供应商做特定**非标准扩展（non-standard extension）**。

非标准扩展（non-standard extension）要么是仅使用了自定义编码的扩展，要么就没按照要求使用任何标准或保留编码的**非标准扩展（non-conforming）**。

指令集扩展一般是通用的，但可能会按照基础 ISA 的不同而在所提供功能上稍有差异。

Chapter 26 介绍了对 RISC-V ISA 进行扩展的各种办法。此外我们还为 RISC-V 的基础指令和指令集扩展确立了一个命名协定（convention），详见 Chapter 27。

为了更加通用地支持软件开发，这里我们还定义了一套标准扩展，用于支持 整数乘/除法、原子操作、单/双精度浮点算术。

- 基础整数 ISA 命名为“I”（前缀是 RV32 或 RV64），其中包含整数计算指令，整数 Load，整数 Store，以及控制流指令。
- 标准整数乘除扩展命名为“M”，其添加了整数乘除法指令（对整数寄存器

中的值进行乘除）。

- 标准原子指令扩展（用“A”表示）添加了用于在处理器间进行同步的 原子读、原子修改、原子写指令。
- 标准单精度浮点扩展用“F”表示，其添加了浮点寄存器，单精度计算指令，以及单精度的 Load 和 Store。
- 标准双精度浮点扩展用“D”表示，其扩展了浮点寄存器，并添加了双精度的计算，Load，Store 指令。
- 标准压缩指令扩展“C”提供了通用指令的 16bit 版本。

我们认为，除了基础整数 ISA 以及标准 GC 扩展，之外的新指令都很难为所有应用程序都带来明显的好处（尽管可能对某些领域非常有益）。考虑到能效（energy efficiency）趋势，我们认为，对 ISA spec 所需的部分进行简化是相当重要的。

而其它的体系结构则通常将 ISA 视为一个单一的实体，并随着指令的添加更新版本。与此不同，RISC-V 则会尽可能地努力确保每个标准扩展都能随着时间保持不变，并将新的指令作为进一步可选的扩展层。例如，基础整数 ISA 在今后也将继续作为完全支持的独立 ISAs，而无需考虑任何后续扩展。

1.4 内存（Memory）

RISC-V hart 具有一个 2^{XLEN} bytes 的单字节可寻址空间。内存的一个字（word）定义为 32bits (4 bytes)。相应地，半字（half word）为 16bits(2 bytes)，双字（double word）为 64bits(8 bytes)，四字为 128bits(16 bytes)。

内存地址空间是循环的，因此地址 $2^{XLEN}-1$ 处的字节与地址 0 处的字节是相邻的。相应地，硬件上关于内存地址的计算也会忽略溢出（相当 $\bmod 2^{XLEN}$ ）。

执行环境（execution environment）决定了硬件资源到 hart 地址空间的映射关系。hart 地址空间上的一段地址范围可能：

（1）未占用（vacant）。

（2）包含了主存（main memory）

（3）包含了一或多个 I/O 设备。

对 I/O 设备的读写可能导致可见（visible）的副作用（side effects），但对主存的访问不会。尽管执行环境可将 hart 地址空间中的所有内容都调给 I/O 设备，但通

常的预期是其中的某些部分会被指定为主存。

当一个 RISC-V 平台上有多个 HART 时，其中任意的两个 HART 的地址空间可能：完全相同或者完全不同，又或者部分不同但又共享某些资源子集（resource subset），被映射到相同或者不同的地址范围。

对于纯“裸机（bare metal）”环境，所有 Hart 都可能看到完全相同的地址空间（全是物理寻址访问）。然而，一旦执行环境纳入了具有地址转换的操作系统，操作系统通常就会为每个 hart 提供 基本上，或者是完全属于它们自己的虚拟地址空间。

每个 RISC-V 机器指令的执行都会涉及一或多个内存访问，具体细分为隐式访问（implicit access）和显式访问（explicit access）。每个指令都会涉及一个隐式访存（即获取指令），来获得待执行指令的编码。大多 RISC-V 指令在取指之后并不会进行更多访存。特别地，Load 和 Store 指令会对指令所确定的地址进行显式的读取或写入。在非特权 ISA 文档之外，执行环境可能会要求指令在执行时候进行额外的隐式访存（如进行地址转换）。

执行环境决定了，对于每种内存访问，哪些非空地址空间（non-vacant address）是可访问的（accessible）。例如，取指（隐式读取）的可访位置集 与 Load 指令（显式读取）的可访位置集 两者间可能相互重叠，也可能不存在任何重叠；以及，Store 指令（显式写入）的可访位置集可能只是可读位置集的子集。正常情况下，指令在试图访问内存中不可访问的地址时，将会引起关于该指令的异常（exception）。**地址空间中的未占用位置（vacant locations）永远不可访问（never accessible）。**

除非另加说明（specified otherwise），否则不引起任何异常的隐式读取可以任意提前或者推测地执行，甚至在机器被证明可能需要读取之前都行。具体例如，可以在实现中尝试尽可能地尽早读取所有主存中的数据，缓存尽可能多的可取（可执行）（fetchable (executable)）字节，以便后续的取指不必再次访问主存。

为确保某些隐式读取能排序到同地址上的写入后面，软件必须执行相应的 fence 或者用于此目的的 cache 控制指令（如 Chapter 3 中所定义的 FENCE.I 指令）。

hart 对某地址进行的访存操作（显式或者隐式）后，如果其它 hart 或者硬件实体（agent）可访问该位置的话，则有可能在该位置上看到不一样的访问操作顺序。不过，这种内存访问的重排序始终都会受到所适内存一致性模型（memory consistency model）的约束。

RISC-V 默认的内存一致性模型是 Chapter 16 和 Appendix 中所定义的 RVWMO（ RISC-V Weak Memory Ordering）。Chapter 24 中定义了另一个可选

模型，即 TSO（Total Store Ordering），它对内存序有着更强的约束。

RVWMO 模型是任何 RISC-V 实现中所允许的最弱模型，因而基于该模型所编写的软件能与所有采用其它内存一致性规则的 RISC-V 实现相兼容。

类似前面的隐式读取，要在内存一致性模型与执行环境的假设之外进一步保持特定访存顺序的话，软件就必须主动地执行 fence 或者 cache 控制指令。

1.5 基础指令长度编码（Base Instruction-Length Encoding）

基础 RISC-V 指令集（base RISC-V ISA）具有 32bit 的定长指令，其必须对其到 32bit 边界上。

不过，RISC-V 标准编码方案（standard RISC-V encoding scheme）的设计也支持带有变长指令的 ISA 扩展（ISA extensions with variable-length instructions），其中每条指令都由一或多个 16bit 的指令字片（parcel）构成，这些字片都是自然对齐到 16bit 边界上的。

于 Chapter 17 中描述的标准压缩 ISA 扩展（standard compressed ISA extension）提供了压缩到了 16bit 的指令，以减少代码大小；这个扩展放松了对齐约束，允许所有指令（16bit/32bit）对齐到任意 16bit 的边界，从而提高代码密度。

我们用术语 **IALIGN**（单位为 bit）来表示实现强制执行的指令地址对齐约束。base ISA 中的 IALIGN 是 32bit 的，但某些 ISA，包括压缩 ISA 扩展，将 IALIGN 放宽到了 16bit。**IALIGN 只能等于 16 或者 32，其它取值都不行。**

我们用术语 **ILEN**（单位为 bit）来表示实现（implementation）所支持的最大指令长度，它始终都是 IALIGN 的倍数。对于那些仅支持 base ISA 的实现，其 ILEN 为 32bit。支持较长指令的实现会有较大的 ILEN 值。

Figure 1.1 阐明了 RISC-V 标准指令长度编码协定（standard RISC-V instruction-length encoding convention）。base ISA 中的所有 32bit 指令中最低 2bit 都是 11。对于可选的 16bit 压缩指令集扩展，则最低 2bit 可以等于 00, 01 或者 10。

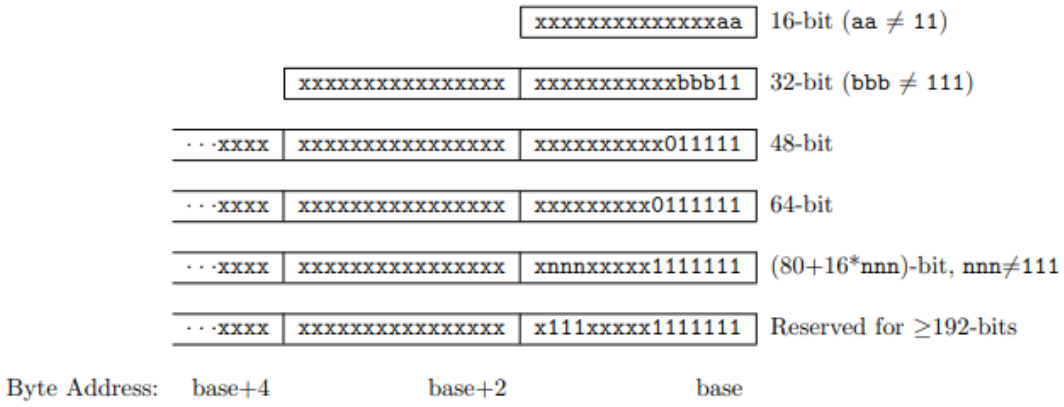

Figure1.1: RISC-V 指令长度编码。现在其中只有 16bit 和 32bit 的编码被冻结了

指令长度扩展编码（Expanded Instruction-Length Encoding）

32bit 指令编码空间中的一部分已被暂时分配给了那些长于 32bit 的指令，目前这部分空间被整个地保留了（reserved）， 以下关于超过 32bit 的指令编码的提议目前并未冻结。

48bit 和 64bit 长度的协定如 Figure1.1 所示，在指令集标准编码扩展中，长于 32bit 的指令会将更多的低位 bit 设为 1。80~176bit 之间的指令长度使用 3bit 字段 [14:12] 来编码其多出于 5*16bit 之外的 16bit 字数量。[14:12]被设为 111 的编码则保留给将来更长的指令编码。

考虑到压缩格式所节约的代码大小与能耗，我们便想直接在 ISA 编码方案中建立对压缩格式的支持，而非作为事后考虑添加。但为了允许更简单的实现，我们并不强制要求使用压缩格式。

我们还想要支持更长的指令，以用于实验以及更大的指令集扩展。尽管我们的协定对核心 RISC-V ISA 的编码要求更为严格，但它也带来了几个好处。

标准 IMAFD ISA 的实现仅需在指令 cache 中缓存指令的高 30bit（节约了 6.25%），在重填指令 cache 时，如遇低 2bit 不规范的指令，则将先其重编码为非法的 30bit 指令，再填入 cache 中，从而保留非法指令的异常行为。

也许更重要的一点是，通过将 base ISA 压缩到 32bit 指令字的一个子集，我们为那些非标准和自定义的扩展留出了更多空间。特别是，base RV32I ISA 仅仅只用了不到 1/8 的 32bit 指令字编码空间。正如 Chapter 26 所述，对于那些不支持标准压缩指令扩展的实现，还可以将另外 3 个不符合规范的 30bit 指令空间映射至 32bit 定长格式，并同时依旧保留对≥32bit 标准指令集的支持。此外，如果该实现也不需要长度大于 32bit 的指令，它还可以为那些不一致扩展（non-conforming）恢复使用另外 4 个主操作码（major opcodes）。

字段[15:0]为全 0 的编码被定义为非法指令（illegal instructions）。应当将这些（字段[15:0]为全 0 的）指令看作是指令长度为最小长度的指令：如果存在任何 16bit 指令集扩展则将其看作 16bits 的指令，其它情况则看作 32bits 的指令。

字段[ILEN-1:0]为全 1 的编码也是非法指令，且应将其视为 ILEN bit 长的指

令。

任何长度的全 0 编码指令都会被看作是非法的，这使得一旦跳转到全 0 的内存区域，就会直接造成 trap。类似的，我们也将全 1 编码的指令保留为非法指令，以便 catch 到其它来自未编程的非易失性内存设备（unprogrammed non-volatile memory devices），断开的内存总线或者是损坏的内存设备的常见 pattern。

如果要用到显式非法指令，软件可以将自然对齐且值为 0 的，格式为 32bit word 作为非法指令使用，这在任何 RISC-V 实现上都是可行的。由于变长编码，我们很难为全 1 编码确定一个统一的已知非法数值，软件通常并不能将 ILEN bits 的全 1 编码作为非法数值使用，因为软件可能会不知道最终目标机器的 ILEN（例如，要将软件编译到多个不同机器所使用的标准二进制库中时）。

我们也考虑过将 32bit 全 1 的 word 定义为非法，因为所有机器都必须支持 32bit 的指令尺寸，然而这样的话就会要求，在 ILEN>32 的机器上，这样的指令与保护边界（borders a protection boundary）相邻时，取指单元报告非法指令异常（illegal instruction exception）而不是访问错误异常（access-fault exception）了，使得变长指令的取指和解码变得更复杂。

RISC-V 基础 ISAs 既有小端序的内存系统，也有大端序的，更进一步地，特权体系结构中还定义了双端序操作（bi-endian operation）。**指令在内存中是以小端序字片（parcel）序列的形式存储的，与内存系统的端序无关。**构成一条指令的字片以半字递增的地址存储，其中地址最低的字片存有指令规范中编号最低的 bits。

出于小端序在商业上的主导地位（所有 x86 系统；IOS，Android，以及 Windows for ARM），我们最初为 RISC-V 内存系统选择了小端序字节排序。并且另一个小细节是，我们还发现对于硬件设计师来说小端序存储系统要更自然。

然而，大端序对于某些应用依旧是需要的，例如 IP 网络就是在大端序的数据结构上运行的，另外某些遗留的代码库也是在大端序处理器上构建的，因此我们也定义了 RISC-V 的大端序和双端序变体。

为确保用于编码指令长度的 bits 能够始终出现在半字地址顺序的开头，我们固定了指令字片在内存中的存储顺序，使其独立于内存系统的端序。从而取指单元只需检查首个 16bit 指令字片中的前几 bit 就能确定变长指令的长度。

我们还进一步将指令字片本身小端序化，以将指令编码完全与内存系统端序解耦。这种设计既有利于软件工具，也有利于双端序硬件。例如，RISC-V 汇编器或者反汇编器能够直接确定预期的工作端序（尽管在双端序系统中，端序模式还是可能会在执行时动态地改变）。通过固定指令端序，有时甚至还能写出端序无知（endianness-agnostic）的软件（甚至在二进制形式下），就像位置无关代码一样。

不过，仅支持小端序指令确实可能对用于编/解码机器指令的 RISC-V 软件造成影响。例如，大端序 JIT 编译器在写入指令内存时就必须交换字节序了。

确定指令的固定小端序后，指令长度编码便很自然地被放在了指令格式中的 LSB 位置，以免需要拆分操作码字段。

1.6 异常，陷阱，和中断（Exceptions, Traps, and Interrupts）

我们使用术语“异常（exception）”，来代指运行时与当前 RISC-V hart 关联的

一条指令的一个非正常状况（unusual condition）。

我们使用“中断（intrrupt）”一词来代指可能导致 RISC-V hart 经历非预期控制转移的外部异步事件。

我们使用术语“陷阱（trap）”来代指由异常或者中断引起的，到 trap handler 的控制转移。

可能造成异常的各个情况会在后续章节的指令描述中介绍。大多数 EEI 的通常行为是，一条指令发出异常信号时，将陷入某个 handler（浮点异常（floating-point exceptions）除外：在标准浮点扩展中，浮点异常并不导致 trap）。hart 产生，路由到，以及启用中断的方式则取决于 EEI。

我们所使用的“异常”和“陷阱”与 IEEE-754 floating-point standard 相兼容。

如何处理（handle）trap，以及 trap 对软件的可见效果则取决于包围它的执行环境。从执行环境中所运行软件的角度来看，hart 在运行时所遭遇的 trap 可能有 4 种不同效果：

- **包含陷阱（Contained Trap）：**trap 对执行环境运行中的软件可见，并且也由运行中的软件处理。例如，在 harts 上同时提供 supervisor-mode 和 user-mode 的 EEI 中，user-mode 的 ECALL 通常会在同一 hart 上将控制转移给所运行的 supervisor-mode handler。类似地，在同样的环境下，当 hart 被中断时候，interrupt handler 也将以 supervisor-mode 运行在同一 hart 上。
- **请求陷阱（Requested Trap）：**这种 trap 是对执行环境的显式调用，是同步异常（sync exception），作为执行环境中软件的请求操作。例子如系统调用（system call）。这种情况下，执行环境在执行了请求的操作后，可能会让 hart 恢复执行（execution resume），但也可能不。例如系统调用可能会直接将 hart 移除，或者有序终止整个执行环境。
- **不可见陷阱 （Invisible Trap）：**执行环境会透明地（transparently）处理此类陷阱，并在处理后正常恢复执行（execution resume）具体例子如，模拟丢失指令（missing instructions），处理按需分页虚拟内存系统（demand-paged virtual-memory system）中的非驻留页错误（non-resident page faults），或者是处理多道程序机器上不同作业的设备中断。在这类情况下，执行环境中的运行软件是无法感知到 trap 的（在这些定义中，我们忽略时序影响）。
- **致命陷阱（Fatal Trap）：**这类 trap 代表致命故障（fatal failure），并将导致

执行环境终止执行。例子包括虚拟内存页面保护检查失败，或者允许watchdog timer 到期失效。所有 EEI 都应该定义执行是如何被终止的，以及向外界环境进行报告。

下表 Table 1.1 展示了各类 trap 的特征：

	Contained	Requested	Invisible	Fatal
Execution terminates	No	No[1]	No	Yes
Software is oblivious	No	No	Yes	Yes[2]
Handled by environment	No	Yes	Yes	Yes

Table 1.1: 各类 Trap 的特征。注：1)可能会要求进行终止（Termination）2)软件可能察觉到非精确致命陷阱。

EEI 定义了每个 trap 是否是被精确处理（handled precisely），尽管建议上是尽可能保持精确性。不精确的包含陷阱和请求陷阱可能会被执行环境内部的软件感知到是不精确的。对于不可见的陷阱，根据定义，则不应该被执行环境下的运行软件观察到是否精确。如果致命对于致命陷阱，如果已知的错误指令并不会导致立即终止（termination），则执行环境下的运行软件可能会观察到致命陷阱是不精确的。

由于本文档描述的是非特权指令，因此很少会提及 trap。特权架构手册在体系结构水平上定义了 contained trap 该如何处理，以及其它用于支持更多 EEI 的其它特性。这个文档仅记录那些在定义上只能造成请求陷阱的非特权指令。不可见中断本质上超出了这个文档的范围。至于此处未定义，且在别处也没定义的指令编码，则可能造成致命陷阱。

1.7 非规范行为与非规范值（UNSPECIFIED Behaviors and Values）

体系结构（architecture）充分地描述了，实现（implementations）该做什么，以及它们在可能做什么时的约束（constraints）。术语“**非规范（UNSPECIFIED）**”用于指明那些，在体系结构上有意不约束实现的情况。

术语 UNSPECIFIED 是指，那些刻意不约束的行为或者值。这些行为或值的定义是开放给扩展，平台标准或者实现的。扩展，平台标准或者实现的文档中可提供相关标准，以进一步限制基础架构中未规范的情况。

与基础体系结构（base architecture）一样，扩展也应当充分描述允许的行为和值，并在刻意不约束的情况下使用术语 UNSPECIFIED。这些情况可能在其它扩展，平台标准或实现中被约束/定义。