RISC-V 指令集手册

卷 2：特权体系结构（Privileged Architecture）

特权体系结构 1.7 版

文档版本 1.7 版

（翻译：要你命 3000@EETOP 翻译版本 1.0）

警告！这个规范的初稿在成为标准之前，可能会被修改，因此基于此规范初稿的实现，可能与未来的标准规范并不相符。

Andrew Waterman, Yunsup Lee, Rimas Avižienis, David Patterson, Krste Asanović

CS Division, EECS Department, University of California, Berkeley

{waterman|yunsup|rimas|pattrsn|krste}@eecs.berkeley.edu

2015 年 5 月 9 日

该文档同时也是 UCB/EECS-2015-49 技术报告

目录

第1章 介绍

这是一个 RISC-V 特权体系结构描述文档的初始版本。这个版本与我们当前的实现并不相符。欢迎反馈。在最终发布版本之前，可能会修改。

本文档描述了 RISC-V 特权体系结构，它覆盖了除了用户级 ISA 之外所有的 RISC-V 其他方面的内容，包括特权指令、运行操作系统所需的额外功能、接入外部设备。

我们的设计考虑，将出现在类似的文本段落内，如果读者只关心规范，则可以跳过这些段落。

我们清楚地知道，本文档所描述的整个特权级设计，可以被完全不同的另外一个特权级设计所替代，而不需要修改用户级 ISA，甚至不需要改变 ABI。特别地，整个特权级规范被设计成用于运行现有的操作系统，包含了一个传统的基于层次的保护模型。其他的特权规范可以包含其他某些更为灵活的保护域模型。

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者多个 RISC-V 兼容的核心、其他非 RISC-V 兼容的核心、固定功能的加速器、各种物理存储器结构、I/O 设备以及一个允许这些部件相互通信的互联结构。

一个部件被称为“***核心（core）***”，如果它包含了一个独立的指令取指单元。一个 RISC-V 兼容核心可能通过多线程技术支持多个 RISC-V 兼容硬件线程，或者 ***harts***。

一个 RISC-V 核心可能有额外的专有指令集扩展或者一个增加的***协处理器（coprocessor）***。我们使用术语***协处理器***，指的是一个接入到 RISC-V 核心的单元，绝大部分时间被 RISC-V 指令流序列化，但它包含了一些额外的体系结构状态和指令集扩展，并有可能相对于主要的 RISC-V 指令流来说，有一定的自治。

我们使用术语***加速器（accelerator）***，指的是一个要么不能编程的固定功能单元，要么是一个可以自治工作、但专门用于某项任务的核心。在 RISC-V 系统中，我们预期许多可编程加速器将会是基于 RISC-V 核心的，包含专门的指令集扩展，和/或定制化的协处理器。一类重要的 RISC-V 加速器是 I/O 加速器，它将 I/O 处理任务从主要应用核心上卸载负载（offload）。

RISC-V 硬件平台的系统级组织结构，可以从单个核心的微控制器，到一个拥有共享存储器众核（manycore）服务器作为节点的数千个节点的集群系统。即使是小型的片上系统（System on a chip）也可能是结构化的，包含层次化的多个计算机和/或多个处理器，以便能够模块化设计研发或者在不同的子系统之间提供安全隔离。

本文档重点关注运行在一个单处理器或者一个共享存储器多处理器中的每一个 hart（硬件线程）可以看到的特权体系结构。

1.2 RISC-V 特权软件栈术语

本节描述我们用于描述 RISC-V 各种各样可能的特权软件栈中的部件术语。

图 1.1 给出了 RISC-V 体系结构支持的可能的软件栈。最左边的图给出了一个简单的系统，它只支持单个应用程序运行在一个应用执行环境（Application Execution Environment, AEE）上。这个应用程序被编码为与一个特定的应用程序二进制接口（Application Binary Interface, ABI）运行。这个 ABI 包含所支持的用户级 ISA，加上一堆与 AEE 交互的 ABI 调用。ABI 对应用程序隐藏了 AEE 的细节，使得实现 AEE 具有更大的灵活性。同样的 ABI 可被直接实现在多个不同的主机操作系统上，或者被一个用户模式下的仿真环境支持，这个仿真环境运行在一个不同的 ISA 机器上。（**译者注：就像虚拟机，或者指令模拟器**）

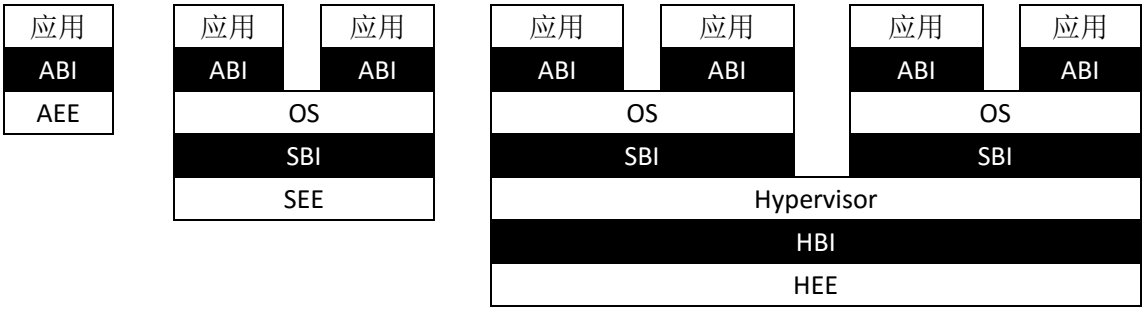


图 1.1：不同实现的栈，支持各种形式的特权执行

在我们的图形表示中，用黑底白字表示抽象接口，将它们与实现接口的具体部件实例相区分开来。

中间的配置显示了一个传统的操作系统（OS），可支持多个应用程序的多道运行。每个应用程序通过 ABI 与 OS 通信，OS 提供了 AEE。正如同应用程序通过 ABI 与 AEE 通信一样，RISC-V 操作系统通过一个管理员二进制接口（Supervisor Binary Interface，SBI）与管理员执行环境（Supervisor Execution Environment，SEE）通信。一个 SBI 包含了用户级和管理员级 ISA，以及一堆 SBI 函数调用。在所有 SEE 实现中，使用单一的 SBI，允许单一的 OS 二进制镜像运行于任何 SEE 之上。在低端的硬件平台上，SEE 可以只是一个简单的 boot loader 和 BIOS 类型的 IO 系统，在高端服务器上，SEE 可以是一个提供 hypervisor 的虚拟机，或者在一个体系结构仿真环境中，SEE 可以是一个运行在主机操作系统上的“很薄的”转换层。

绝大多数管理员级 ISA 定义，并没有将 SBI 和执行环境和/或硬件平台进行区分，使得虚拟化和开发新的硬件平台变得复杂化。

最右侧的配置显示了一个虚拟机监视器配置，此处由一个单一的 hypervisor 支持多个多道操作系统。每个 OS 通过一个 SBI 与 hypervisor 通信，hypervisor 提供了 SEE。Hypervisor 使用一个 hypervisor 二进制接口（Hypervisor Binary Interface，HBI）与 hypervisor 执行环境（Hypervisor Execution Environment，HEE）通信，这将把 hypervisor 与具体的硬件平台细节相隔离。

各种各样的 ABI、SBI 和 HBI 仍然在研究中，但是我们预感到 SBI 和 HBI 通过与 virtio[3]类似的虚拟化设备接口来支持设备，并支持设备发现。通过这种方式，只需要书写一组设备驱动程序，就可以支持任何 OS 或者 hypervisor，并且也可以和 boot 环境共享。

RISC-V ISA 的硬件实现通常需要除了特权 ISA 之外的其他一些特性，才能支持各种各样的执行环境（AEE、SEE、HEE）。我们使用一个硬件抽象层（Hardware Abstract Layer，HAL）将硬件平台所需要的特性和执行环境相分离，如图 1.2 所示。注意到在一个 RISC-V 软件栈中，HAL 并不是必须的，因为一个执行环境可能仅仅是一个由软件仿真提供的，或者不需要抽象而直接书写在一个给定的硬件平台上。

后续章节将详细描述所提出的 RISC-V 硬件平台标准设计。

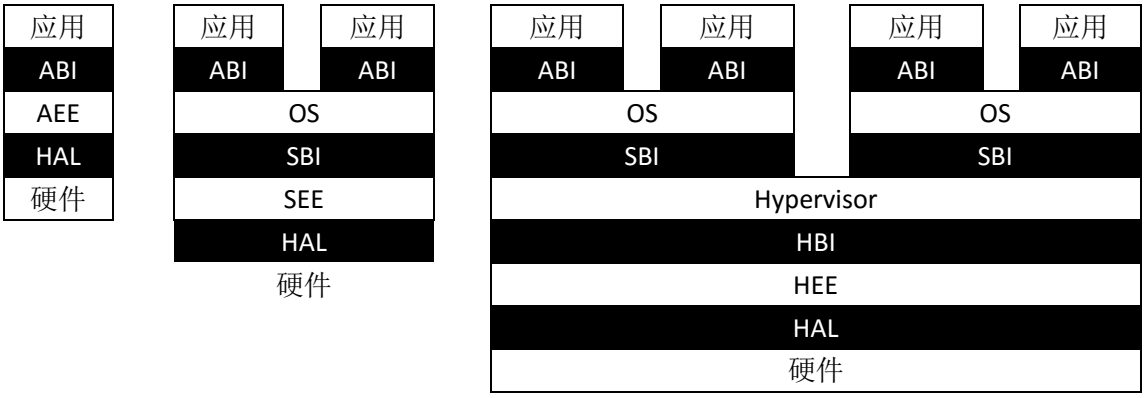


图 1.2：硬件抽象层（HAL）将底层硬件平台从执行环境中进行抽象。

1.3 特权级

任何时候，一个 RISC-V 硬件线程（***hart***）是运行在某个特权级上的，这个特权级被编码到一个或者多个 CSR（control and status register，控制和状态寄存器）中的一种模式。当前定义了四种特权级，如表 1.1 所示。

级别	编码	名字	缩写
0	00	用户/应用程序	U
1	01	管理员	S
2	10	Hypervisor	H
3	11	机器	M

表 1.1：RISC-V 特权级

特权级被用于在不同的软件栈部件之间提供保护，试图执行当前特权模式不允许的操作，将导致一个异常的产生。这些异常通常会导致下层执行环境或者 HAL 产生自陷（trap）。

机器级是最高级特权，也是 RISC-V 硬件平台唯一必须的特权级。运行于机器模式（M-mode）下的代码是固有可信的（inherently trusted），因为它可以在低层次访问机器的实现。用户模式（U-mode）和管理员模式（S-mode）被分别用于传统应用程序和操作系统，而 Hypervisor 模式（H-mode）则是为了支持虚拟机监视器。

每一个特权级都有一组核心的特权 ISA 扩展，以及可选扩展和变种。例如，机器模式支持数种可选的标准变种，支持地址翻译和存储器保护。

虽然现在并没有定义任何一个，未来 hypervisor 级 ISA 扩展将会被加入，以提高虚拟化的性能。一个支持 hypervisor 常见的特性是，从管理员物理地址到 hypervisor 物理地址，提供第二级别的翻译和保护。

实现可以提供从特权级 1 到特权级 4 任意级别开始的特权模式，这需要在隔离性和较小的实现代价之间进行折中，如表 1.2 所示。

级别数量	支持的模式
1	M
2	M，U
3	M，S，U
4	M，H，S，U

表 1.2：支持的特权模式组合

在描述中，我们试图将书写代码的特权级，和代码运行的特权模式，相分离，虽然经常两者是紧密相连的。例如，一个管理员级操作系统可以在一个拥有 3 种特权模式的系统上运行在管理员模式上，但是在一个拥有 2 种或者更多种特权模式的系统上，它也可以在一个经典的虚拟机监视器中运行在用户模式下。在这两种情形中，可以使用同样的管理员级操作系统二进制代码，被编码为一个管理员级 SBI，因此能够使用管理员级特权指令和 CSR。当以用户模式运行一个客户操作系统（guest OS）时，所有管理员级行为将会导致自陷，并由运行在更高特权级的 SEE 进行仿真。

所有硬件实现必须提供 M-mode，因为这是唯一的模式，可以不受限制地访问整个机器。最简单的 RISC-V 实现可以仅提供 M-mode，虽然这样做不能为防止不正确的、恶意应用代码提供保护。许多 RISC-V 实现还支持至少一个用户模式（U-mode），以对系统的其他部分进行保护，防止被应用程序代码破坏。管理员模式（S-mode）可被加入，以在管理员级操作系统和 SEE、HAL 之间提供隔离。Hypervisor 模式（H-mode）将在一个虚拟机监视器和 HEE、运行在机器模式的 HAL 之间提供隔离。

一个 hart（硬件线程）通常在 U-mode 下运行应用程序，直到某些自陷（例如一个管理员调用或者一个定时器中断）强制切换到一个自陷处理函数（trap handler），这个自陷处理函数通常运行在更特权的模式下。然后这个线程将执行这个自陷处理器函数，它最终在 U-mode 下，在引起自陷的指令处或之后，继续线程执行。提升特权基级别的自陷称为垂直自陷（vertical trap），而保持在同样特权级别的自陷称为***水平自陷（horizontal trap）***。RISC-V 特权体系结构提供了将自陷灵活地路由到不同的特权层。

水平自陷可以实现为垂直自陷，它将控制返回给处于较低特权模式的水平自陷处理函数。

第2章 控制和状态寄存器（CSR）

在 RISC-V ISA 中，SYSTEM 主要操作码被用来编码所有的特权指令。这可以分为两类：一类是原子性读-修改-写控制和状态寄存器（CSR）的指令，另一类是其他特权指令。本章我们将描述访问 CSR 的指令，因为它们在所有特权级都很常见。后面章节将根据特权级描述每个 CSR 的功能，以及其他与某个特定特权级紧密相关的特权指令。注意到虽然 CSR 和指令虽然与一个特权级相关，但是它们也可以从更高的特权级进行访问。

将所有特权指令放到同一个主要操作码和结构下，可以简化硬件自陷编码，并为虚拟化执行环境做好了准备。

在这个初始版本的规范中，许多 CSR 寄存器包含一些字段，它们的值当前并没有被使用，因此被置为 0，或者它们的值当前仅支持有限范围的设置（译者注：即可能的取值只是有限几个值），但是未来可能支持一个扩展范围的设置。一般来说，软件只能将支持的值写入到字段中，而硬件仅返回指定的缺省值。在此规范正式发布之前，这些地方将被适当的标记，以指明为了保证正确的向前兼容，而必须做出的准确行为。

2.1 访问 CSR 的指令

除了本手册卷 1 描述的用户状态之外，一个实现可以包含额外的 CSR，它们可由某些特权级的子集所访问。下面的指令可被用于原子性的读和修改 CSR。操作 CSR 的指令可能有一些副作用。

31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	zimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	zimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	zimm[4:0]	CSRRCI	dest	SYSTEM

CSRRW（Atomic Read/Write CSR）指令原子性的交换 CSR 和整数寄存器中的值。CSRRW 指令读取在 CSR 中的旧值，将其零扩展到 XLEN 位，然后写入整数寄存器 ***rd*** 中。***rs1*** 寄存器中的值将被写入 CSR 中。

CSRRS（Atomic Read and Set Bit in CSR）指令读取 CSR 的值，将其零扩展到 XLEN 位，然后写入整数寄存器 ***rd*** 中。整数寄存器 ***rs1*** 中的值指明了哪些 CSR 中的位被置为 1。***rs1*** 中的任何为 1 的位，将导致 CSR 中对应位被置为 1，如果 CSR 中该位是可以写的话。CSR 中的其

他位不受影响（虽然当 CSR 被写入时可能有些副作用）。

CSRRC（Atomic Read and Clear Bit in CSR）指令读取 CSR 的值，将其零扩展到 XLEN 位，然后写入整数寄存器 ***rd*** 中。整数寄存器 ***rs1*** 中的值指明了哪些 CSR 中的位被置为 0。***rs1*** 中的任何为 1 的位，将导致 CSR 中对应位被置为 0，如果 CSR 中该位是可以写的话。CSR 中的其他位不受影响。

对于 CSRRS 指令和 CSRRC 指令，如果 ***rs1＝x0***，那么指令将根本不会去写 CSR，因此应该不会产生任何由于写 CSR 产生的副作用（**译者注：某些特殊 CSR 检测是否有人尝试写入，一旦有写入，则执行某些动作。这和写入什么值没什么关系**）。注意如果 ***rs1*** 寄存器包含的值是 0，而不是 ***rs1＝x0***，那么将会把一个不修改的值写回 CSR（**译者注：这时会有一个写 CSR 的操作，但是写入的值就是旧值，因此可能会产生副作用**）。

CSRRWI 指令、CSRRSI 指令、CSRRCI 指令分别于 CSRRW 指令、CSRRS 指令、CSRRC 指令相似，除了它们是使用一个处于 ***rs1*** 字段的、零扩展到 XLEN 位的 5 位立即数（***zimm[4:0]***）而不是使用 ***rs1*** 整数寄存器的值。如果 ***zimm[4:0]***字段是零，那么这些指令将不会写 CSR，因此应该不会产生任何由于写 CSR 产生的副作用。

用于读取 CSR 的汇编语言伪指令 CSRR ***rd, csr*** 被编码为 CSRRS ***rd, csr, x0***。用于写 CSR 的汇编语言伪指令 CSRW ***csr, rs1*** 被编码为 CSRRW ***x0, csr, rs1***，而伪指令 CSRWI ***csr, zimm*** 被编码为 CSRRWI ***x0, csr, zimm***。

还有汇编语言伪指令被定义在不需要 CSR 旧值时，用来设置和清除 CSR 中的位：CSRS/CSRC ***csr, rs1***；CSRSI/CSRCI ***csr, zimm***。

2.2 CSR 地址映射约定

标准 RISC-V ISA 设置了一个 12 位的编码空间（csr[11:0]）可用于 4096 个 CSR。根据约定，CSR 地址的高 4 位（csr[11:8]）用于编码 CSR 根据特权级读写的可访问性，如表 2.1 所示。最高 2 位（csr[11:10]）指示这个寄存器是否是可以读/写（00、01 或者 10），还是只读的（11）。后面 2 位（csr[9:8]）指示了能够访问这个 CSR 所需要的最低特权级（用户级是 00，管理员级是 01）。

CSR 地址约定使用 CSR 地址的高位来编码缺省的访问特权。这简化了硬件中的错误检测，并提供了更大的 CSR 空间，但是也限制了将 CSR 映射进地址空间。

实现也许允许一个更高特权级，对一个较低特权级对不允许的 CSR 进行访问时产生自陷，以便拦截这些访问。这种改变应当对较低特权级软件是透明的。

试图访问一个不存在的 CSR 将产生一个非法指令异常。试图访问一个没有相应特权的 CSR 或者写一个只读寄存器，也将产生一个非法指令异常。一次读/写寄存器可能包含了某些位是只读的，此种情况下，写入只读位被忽略。

表 2.1 也指明了在标准和非标准使用之间分配的 CSR 地址。保留给非标准使用的 CSR 地址，将来标准扩展也不会重新定义使用。阴影（shadow）部分的地址是保留给一些地址，这些地址在低特权级时是只读的，而在更高特权级时是可以修改这些寄存器的。注意到如果一个特权级已经被分配了一个读/写阴影地址，那么任何更高特权级可以使用同样的 CSR 地址来对同样的寄存器进行读/写。

高效的虚拟化，要求在虚拟环境中，尽可能多的指令以直接（natively）的方式执行，而任何特权访问都自陷到虚拟机监视器[1]。在较低特权级是只读的 CSR，如果它们在更高的特权级可以读写的话，则被投影（shadow）到单独的 CSR 地址空间。这避免了对允许的较低特权级访问产生自陷，而同时可以对非法访问产生自陷。

2.3 CSR 列表

表 2.2-表 2.5 列出了当前被分配了 CSR 地址的 CSR。只有定时器、计数器和浮点 CSR 是当前定义的标准用户级 CSR。其他寄存器被特权代码使用，如后续章节所述。注意并不是所有实现都需要所有寄存器。

CSR 地址			十六进制	使用和可访问性
[11:10]	[9:8]	[7:6]		
用户 CSR				
00	00	XX	0x000-0x0FF	标准 读/写
01	00	XX	0x400-0x4FF	标准 读/写
10	00	XX	0x800-0x8FF	非标准 读/写
11	00	00-10	0xC00-0xCBF	标准 只读
11	00	11	0xCC0-0xCFF	标准 只读
管理员 CSR				
00	01	XX	0x100-0x1FF	标准 读/写
01	01	0X	0x500-0x57F	标准 读/写
01	01	1X	0x580-0x5FF	非标准 读/写
10	01	00-10	0x900-0x9BF	标准 读/写 阴影
10	01	11	0x9C0-0x9FF	非标准 读/写 阴影
11	01	00-10	0xD00-0xDBF	标准 只读
11	01	11	0xDC0-0xDFF	非标准 只读
Hypervisor CSR				
00	10	XX	0x200-0x2FF	标准 读/写
01	10	0X	0x600-0x67F	标准 读/写
01	10	1X	0x680-0x6FF	非标准 读/写
10	10	00-10	0xA00-0xABF	标准 读/写 阴影
10	10	11	0xAC0-0xAFF	非标准 读/写 阴影
11	10	00-10	0xE00-0xEBF	标准 只读
11	10	11	0xEC0-0xEFF	非标准 只读
机器 CSR				
00	11	XX	0x300-0x3FF	标准 读/写
01	11	0X	0x700-0x77F	标准 读/写
01	11	1X	0x780-0x7FF	非标准 读/写
10	11	00-10	0xB00-0xBBF	标准 读/写 阴影
10	11	11	0xBC0-0xBFF	非标准 读/写 阴影
11	11	00-10	0xF00-0xFBF	标准 只读
11	11	11	0xFC0-0xFFF	非标准 只读

表 2.1：RISC-V CSR 地址空间分配

地址	特权	名字	描述
用户浮点 CSR			
0x001	URW	fflags	浮点已产生异常
0x002	URW	frm	浮点动态舍入模式
0x003	URW	fcsr	浮点控制和状态寄存器（frm+fflags）
用户计数器/定时器			
0xC00	URO	cycle	RDCYCLE 指令的周期计数器
0xC01	URO	time	RDTIME 指令的定时器
0xC02	URO	instret	RDINSTRET 指令的已退休指令（instruction-retired）计数器
0xC80	URO	cycleh	cycle 的高 32 位，仅 RV32
0xC81	URO	timeh	time 的高 32 位，仅 RV32
0xC82	URO	instreth	instret 的高 32 位，仅 RV32

表 2.2：当前已分配的 RISC-V 用户级 CSR 地址

地址	特权	名字	描述
管理员自陷 Setup			
0x100	SRW	sstatus	管理员状态寄存器
0x101	SRW	stvec	管理员自陷处理函数基地址
0x104	SRW	sie	管理员中断使能寄存器
0x121	SRW	stimecmp	墙钟（Wall-clock）定时器比较值
管理员定时器			
0xD01	SRO	stime	管理员墙钟时间寄存器
0xD81	SRO	stimeh	stime 的高 32 位，仅 RV32
管理员自陷处理			
0x140	SRW	sscratch	管理员自陷处理函数 Scratch 寄存器
0x141	SRW	sepc	管理员异常程序计数器（exception program counter）
0x142	SRO	scause	管理员自陷原因（trap cause）
0x143	SRO	sbadaddr	管理员坏地址（bad address）
0x144	SRW	sip	管理员挂起的中断（interrupt pending）
管理员保护和翻译			
0x180	SRW	sptbr	页表基地址寄存器
0x181	SRW	sasid	地址-空间 ID
管理员读写、用户只读寄存器阴影			
0x900	SRW	cyclew	RDCYCLE 指令的周期计数器
0x901	SRW	timew	RDTIME 指令的定时器
0x902	SRW	instretw	RDINSTRET 指令的已退休指令（instruction-retired）计数器
0x980	SRW	cyclehw	cycle 的高 32 位，仅 RV32
0x981	SRW	timehw	time 的高 32 位，仅 RV32
0x982	SRW	instrethw	instret 的高 32 位，仅 RV32

表 2.3：当前已分配的 RISC-V 管理员级 CSR 地址

地址	特权	名字	描述
Hypervisor 自陷 Setup			
0x200	HRW	hstatus	Hypervisor 状态寄存器
0x201	HRW	htvec	Hypervisor 自陷处理函数基地址
0x202	HRW	htdeleg	Hypervisor 自陷转移（delegation）寄存器
0x221	HRW	htimecmp	Hypervisor 墙钟（Wall-clock）定时器比较值
Hypervisor 定时器			
0xE01	HRO	htime	Hypervisor 墙钟时间寄存器
0xE81	HRO	htimeh	htime 的高 32 位，仅 RV32
Hypervisor 自陷处理			
0x240	HRW	hscratch	Hypervisor 自陷处理函数 Scratch 寄存器
0x241	HRW	hepc	Hypervisor 异常程序计数器（exception program counter）
0x242	HRW	hcause	Hypervisor 自陷原因（trap cause）
0x243	HRW	hbadaddr	Hypervisor 坏地址（bad address）
Hypervisor 保护和翻译			
0x28X	TBD	TBD	TBD（未定义、研发中）
Hypervisor 读写、管理员只读寄存器阴影			
0xA01	HRW	stimew	管理员墙钟定时器
0xA81	HRW	stimehw	管理员墙钟定时器高 32 位，仅 RV32

表 2.4：当前已分配的 RISC-V Hypervisor 级 CSR 地址

地址	特权	名字	描述
机器信息寄存器			
0xF00	MRO	mcpuid	CPU 描述
0xF01	MRO	mimpid	Vendor ID 和版本号
0xF10	MRO	mhartid	硬件线程 ID
机器自陷 Setup			
0x300	MRW	mstatus	机器状态寄存器
0x301	MRW	mtvec	机器自陷处理函数基地址
0x302	MRW	mtdeleg	机器自陷转移（delegation）寄存器
0x304	MRW	mie	机器中断使能寄存器
0x321	MRW	mtimecmp	机器墙钟（Wall-clock）定时器比较值
机器定时器和计数器			
0x701	MRW	mtime	机器墙钟时间寄存器
0x741	MRW	mtimeh	mtime 的高 32 位，仅 RV32
机器自陷处理			
0x340	MRW	mscratch	机器自陷处理函数 Scratch 寄存器
0x341	MRW	mepc	机器异常程序计数器（exception program counter）
0x342	MRW	mcause	机器自陷原因（trap cause）
0x343	MRW	mbadaddr	机器坏地址（bad address）
0x344	MRW	mip	机器挂起的中断（interrupt pending）
机器保护和翻译			
0x380	MRW	mbase	基本寄存器（base register）
0x381	MRW	mbound	绑定寄存器（bound register）
0x382	MRW	mibase	指令基本寄存器
0x383	MRW	mibound	指令绑定寄存器
0x384	MRW	mdbase	数据基本寄存器
0x385	MRW	mdbound	数据绑定寄存器
机器读写、Hypervisor 只读寄存器阴影			
0xB01	MRW	htimew	Hypervisor 墙钟定时器
0xB81	MRW	htimehw	Hypervisor 墙钟定时器高 32 位，仅 RV32
机器主机-目标机接口（非标准 Berkeley 扩展）			
0x780	MRW	mtohost	到主机去的输出寄存器
0x781	MRW	mfromhost	从主机来的输入寄存器

表 2.5：当前已分配的 RISC-V 机器级 CSR 地址

第3章 机器级 ISA

本章描述在机器模式（M-mode）下面可用的机器级操作，机器模式是 RISC-V 最高特权模式。M-mode 是在一个 RISC-V 硬件实现中唯一强制实现的特权模式（**译者注：你可以不实现其他特权模式，但是机器模式是必须的**）。M-mode 是用于低层次的访问一个硬件平台，是上电复位后进入的第一个模式。M-mode 也被用于实现那些硬件太难直接实现或者耗费太大的特性。RISC-V 机器级 ISA 包含一个被扩展的通用核心，这依赖于支持哪些其他的特权级和其他的硬件实现细节。

3.1 机器级 CSR

除了本节描述的机器级 CSR 之外，M-mode 代码可以访问所有更低特权级的 CSR。

3.1.1 CPU ID 寄存器 mcpuid

mcpuid 寄存器是一个 XLEN 位的只读寄存器，包含了 CPU 实现性能相关的信息。这个寄存器在所有实现中都必须是可读的，但是可以返回一个值 0，表示 CPU ID 特性并没有被实现，需要通过一个单独的非标准机制来判断 CPU 性能。

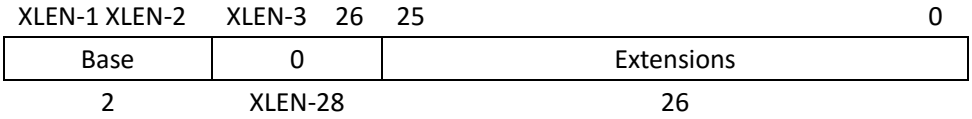


图 3.1：机器 CPU ID 寄存器（mcpuid）

Base 字段编码了原始基本整数 ISA，如表 3.1 所示。对于支持多个 ISA 的实现，Base 字段总是描述最宽支持的 ISA，因为这是复位后进入机器模式时的 ISA。

值	描述
0	RV32I
1	RV32E
2	RV64I
3	RV128I

表 3.1：mcpuid 的 Base 字段编码

基本内核可以通过使用对 mcpuid 返回值的符号位进行分支而快速确定，左移 1 位，然后依据符号位进行第二个分支。这个检查可以写成汇编代码，而不需要知道机器的寄存器宽度（XLEN）。

Extensions 字段编码了存在的标准扩展，每一个单独位代表一个字母序的字母（位 0 表示扩展“A”、位 1 表示扩展“B”、直到位 25 表示未来的标准扩展“Z”）。对 RV32I、RV64I、

RV128I 基本 ISA，“I”位被置为 1，对 RV32E，“E”位被置为 1。

如果支持用户（user）、管理员（supervisor）、hypervisor 特权模式，那么“U”、“S”、“H”位被分别置为 1。

如果有任何非标准的扩展，那么“X”位被置为 1。

mcpuid 寄存器将一些基本的 CPU 特性信息暴露给了机器模式的代码。在机器模式下，更详细的信息可以通过检查其他寄存器、在 boot 过程中检查 ROM 存储器等方式获得。

我们要求较低特权级执行环境的调用，而不是读取 CPU 寄存器，来判定每一个特权级的可用特性。这将使能虚拟化层，使得它可以改变任意级看到的 ISA，并支持异常丰富的命令接口而不增大硬件的负担。

3.1.2 实现 ID 寄存器 mimpid

mimpid 寄存器提供了一个唯一的处理器实现来源和版本编码。这个寄存器在所有实现中都必须是可读的，但是可以返回一个值 0，表示数据字段并没有被实现。

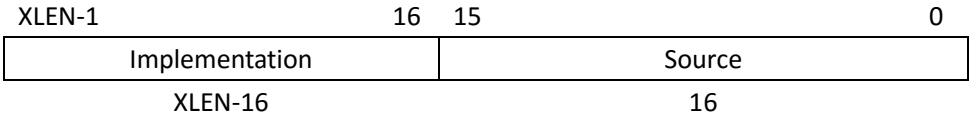


图 3.2：机器实现 ID 寄存器（mimpid）

16 位的 Source 字段用于描述处理器设计的来源，它被划分为两大类：开源的和私有的实现。值 0x0001-0x7FFE 被保留给开源的项目，而值 0x8001-0xFFFE 被保留给闭源的实现。值 0x7FFF 和 0xFFFF 被保留给未来扩展用。值 0x8000 被保留给一个匿名的来源，这可用于在一个 Source ID 被分配之前研发期间使用。

当前 Source 已经被分配的值如表 3.2 所示。

Source 值	描述
0x0000	CPU ID 未实现（**译者注：应为 mimpid**）
0x0001	UC Berkeley Rocket repo
0x0002-0x7FFE	保留给开源实现
0x7FFF	保留给扩展
0x8000	保留给匿名来源
0x8001-0xFFFE	保留给私有实现
0xFFFF	保留给扩展

表 3.2：mimpid 中 Source 字段的编码

mimpid 寄存器剩余的 XLEN-16 位可用于编码实现的详细设计信息，包括微体系结构类型和版本号等。这个字段的格式是由 Source 的提供者自定义的，但是在标准工具中，将其作为一个十六进制字符串打印，因此这个 Implementation 值应该在半个字节的边界对齐，以提高可读性。

mimpid 寄存器的值应该仅仅反映 RISC-V 处理器设计自身的特性，而不是它周围系统的特性。应当使用其他的机制来编码外围系统的细节信息。

意图是使用开源的 ID 来表示这个研发是围绕哪个 repo 开发的，而不是强调某个特定的研究组织。Implementation 字段可用于划分设计的各个分支，包括由那些组织研发。使用开源设计的商业制造应该（可能是由授权所要求的）保留原始的 Source ID。这将减少碎片化和工具支持代价，以及提供归属。开源 ID 只应该在分配后才能发布，用于开源项目，将来很可能由一个基金来管理。商业 ID 很可能由一个商业联盟来分配。

3.1.3 硬件线程 ID 寄存器 mhartid

mhartid 寄存器是一个 XLEN 位的只读寄存器，包含了运行代码的硬件线程 ID 整数值。这个寄存器在所有实现中都必须是可读的。硬件线程 ID 在一个多处理器系统中并不要求一定是连续的，但是至少应该有一个硬件线程，它的 ID 是 0。

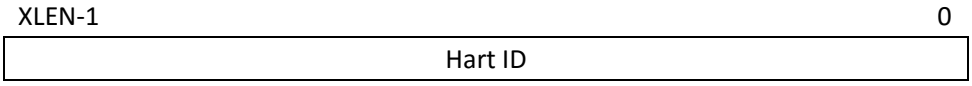


图 3.3：硬件线程 ID 寄存器（mhartid）

在某种情形下，我们必须确保只能有一个硬件线程运行某些代码（例如，在复位时），因此需要一个硬件线程，它具有一个已知的硬件线程 ID=0。

我们并没有使用 CSR 或者特权指令来传输关于底层硬件平台的组织结构信息，因为这样做就需要一个无边界的扩展机制，并且必须包含非 RISC-V 核心和从设备。系统研发人员必须提供一种手段给机器模式代码，使它能够询问平台并发现系统的结构。

3.1.4 机器状态寄存器（mstatus）

mstatus 寄存器是一个 XLEN 位的可读/可写寄存器，其格式如图 3.4 所示。mstatus 寄存器持续跟踪和控制硬件线程的当前操作状态。mstatus 在 H 和 S 特权级 ISA 受限的视图，分别出现在 hstatus 和 sstatus 寄存器中。

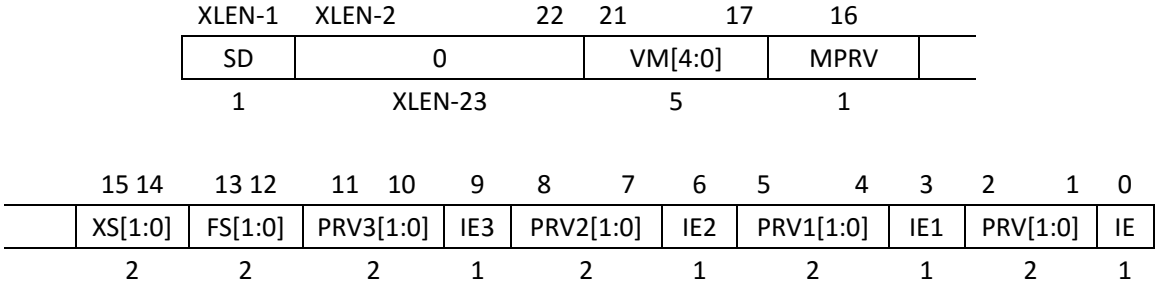


图 3.4：机器模式状态寄存器（mstatus）

3.1.5 mstatus 寄存器中的特权和全局中断使能栈

PRV[1:0]字段保存了硬件线程当前特权模式，其编码如表 1.1 所示。如果实现仅提供 M-mode，那么这两位被硬连线到 11。

IE 位指示了在当前特权模式，是否使能中断（1=使能，0=禁止），其主要用途是禁止中断，以便在与当前特权级的中断处理函数相关时，能够保证原子性。当一个硬件线程运行在一个给定的特权模式时，更高特权模式的中断总是使能的，而更低特权模式的中断总是禁用的。更高特权等级的代码可以在移交控制给较低特权级之前，使用每个中断单独的使能位，来禁用选定的中断。

活跃的 IE 位处于位 0 的位置，使得它可以使用单条 CSR 指令原子性的置位或者清除。

为了支持嵌套的自陷，提供了一个 PRV 和 IE 位的栈，这个栈的深度与支持的特权模式数量相等，此处 PRV0 是活跃的特权模式 PRV（即 PRV0-PRVN 对应 N 级特权模式），除了如果实现仅支持机器模式，在此种情形下，这个栈的深度是 2，并且所有的 PRV 字段都被硬连线为 11。当处理一个自陷时，这个栈被向左边 push，并且 PRV 被设置为活跃自陷处理函数的特权模式，且 IE=0。当从一个自陷处理函数返回时（使用 ERET 指令），栈被向右 pop，并且最左边的项（PRVN）被设置为所支持的最低特权等级，且使能中断（即，在一个只有 M 模式的机器上，PRV1=M 且 IE1=1，而在一个有两个或者更多模式的机器上，当从自陷处理函数返回时，PRVN=U 且 IEN=1）。在通常操作中，这个栈应该包含从左到右单调增加的特权模式（最老的到最新的）。

对于较低的特权模式，任何自陷（同步的或者异步的）通常会在更高的特权模式下处理，并且中断被禁止。更高的特权模式要么处理这个自陷并使用栈信息返回，要么如果不想立即返回被中断的上下文，则在重新使能中断之前保存特权栈，因此只需要每个较低的特权模式一个栈项。

我们考虑过为多个特权级的实现，在特权栈上增加一个额外的级别，以允许 M-mode 软件产生自陷而不需要保存和恢复 mstatus 寄存器。然而，当前产生异常的 M-mode 软件貌似并不因为这个特性而受益。例如，当使用 M-mode 软件仿真缺失的硬件特性时，mstatus 寄存器通常是由于其他原因（例如，为了设置 MPRV 位）而被操作。保存和恢复特权栈可以无代价的折叠（fold into）进这样的操作中。

如果 M-mode 软件希望使能中断，则保存和恢复特权栈的动作，可以类似地被折叠进中断使能/禁止代码序列中。

当栈被 pop 时，最低特权模式且中断被使能，被加入到栈的底部，以帮助捕获导致无效项被 pop 出栈的错误。

PRV 字段只需要保存支持的特权模式。

如果机器只提供 U 和 M 模式，则只需要单个硬件存储位来表示 00 或者 11。然而，软件应当仅对这些字段写入有效值，以保证兼容性。

3.1.6 mstatus 寄存器中的虚拟化管理字段

虚拟化管理字段 VM[4:0]指示了当前活跃的虚拟化方案，包括虚拟存储器翻译和保护。表 3.3 给出了当前定义好的虚拟化方案。对于一个 RISC-V 硬件实现，只有 Mbare 模式是强制要求的。Mbare、Mbb、Mbbid 方案在 3.5-3.6 节进行描述，而基于页面的虚拟存储器方案在后面的章节进行描述。

每个 VM 字段的设置定义了在所有被支持的特权级下面的行为，某些 VM 设置的行为可能依据硬件所支持的特权级而有所不同。

值	缩写	所需要的模式	描述
0	Mbare	M	没有翻译或者保护
1	Mbb	M，U	单个基址和边界
2	Mbbib	M，U	分离的指令和数据基址和边界
3-7	*保留*		
8	Sv32	M，S，U	基于页面的 32 位虚拟寻址
9	Sv39	M，S，U	基于页面的 39 位虚拟寻址
10	Sv48	M，S，U	基于页面的 48 位虚拟寻址
11	Sv57	M，S，U	保留给基于页面的 57 位虚拟寻址
12	Sv64	M，S，U	保留给基于页面的 64 位虚拟寻址
13-31	*保留*		

表 3.3：虚拟化管理字段 VM[4:0]的编码

Mbare 对应于没有存储器管理或翻译，因此所有的有效地址，无论其特权模式，都被认为是机器物理地址。Mbare 模式是复位时进入的模式。

Mbb 是一种“基址和边界”体系结构，它需要系统至少有两个特权级（U 和 M）。Mbb 适用于那些在用户模式代码需要低开销翻译和保护的系统，而不需要按需页面虚拟存储器（支持交换，swapping is supported）。Mbbib 提供了分离的地址（**译者注：应该是指令**）和数据段，允许在进程之间共享一个仅执行（execute-only）的代码段。

Sv32 是一个针对 RV32 系统的基于页面的虚拟存储器体系结构，提供了一个 32 位虚拟地址空间，被设计成支持现代管理员级操作性，包括基于 Unix 的系统。

Sv39 和 Sv48 是针对 RV64 系统的基于页面的虚拟存储器体系结构，提供了一个 39 位或

者 48 位的虚拟地址空间，被设计成支持现代管理员级操作性，包括基于 Unix 的系统。

Sv32、Sv39、Sv48 需要实现支持 M、S 和 U 特权级。如果还支持 H-mode，为 hypervisor 级代码定义了额外的操作，用于支持多个管理员级虚拟机。为支持虚拟机的 H-mode，并没有定义。

现有的 Sv39 和 Sv48 方案可以轻而易举地扩展到 Sv57 和 Sv64 虚拟地址宽度。Sv52、Sv60、Sv68 和 Sv76 虚拟地址空间宽度暂时计划用于 RV128 系统，此处虚拟地址宽度小于 68 位的方案，主要用于那些需要 128 位整数算术但是并不需要更大地址空间的应用程序。

我们当前的虚拟化管理方案定义，在每一个特权级只支持同样的基本体系结构。可以定义虚拟化方案的变种，以在较低的特权级支持较窄的宽度，例如在 RV64 系统上运行 RV32 代码。

3.1.7 mstatus 寄存器中的存储器特权

MPRV 位修改了 load 和 store 执行的特权级。当 MPRV=0 时，翻译和保护如同寻常一样。当 MPRV=1 时，数据存储器地址就如同 PRV 被设置为当前 PRV1 字段的值一样被翻译和保护。指令地址翻译和保护不受影响。

当出现一个异常时，MPRV 被复位为 0。

MPRV 机制被设想用来提升 M-mode 程序仿真缺失硬件特性时的效率，例如非对齐的 load 和 store。

3.1.8 mstatus 寄存器中的扩展上下文状况

（译者注：为了从汉字区别，下文用状况（status）和状态（state）分别说明）

支持丰富的扩展，是 RISC-V 的一个主要目标，因此我们定义了一个标准接口，允许不改变特权模式代码，特别是一个管理员级操作系统，来支持任意用户模式状态扩展。

FS[1:0]和 XS[1:0]可读/可写字段被用于减少保存和恢复上下文的开销，这是通过设置和跟踪浮点单元和任何其他用户模式扩展的状态来实现的。FS 字段编码了浮点单元的状况，包括 CSR fcsr 和浮点数据寄存器 f0-f31，而 XS 字段编码了任何额外用户模式扩展及相关联的状态的状况。SD 位是一个只读位，指明了是否 FS 字段或者 XS 字段编码了一个脏的状态，需要将扩展的用户上下文写入到存储器中。在没有浮点单元的系统中，FS 字段被硬连线到零，而在没有需要新状态的额外用户扩展的系统中，XS 字段被硬连线到零。如果 FS 字段和 XS 字段都被硬连线到零，那么 SD 也总是零。

FS 字段和 XS 字段使用了相同的的状况编码，如表 3.4 所示，其四个可能的的状况值是 Off、Initial、Clean 和 Dirty。

状态	意义
0	Off
1	Initial
2	Clean
3	Dirty

表 3.4：FS[1:0]和 XS[1:0]字段的编码

到目前为止，还没有标准的扩展定义了额外的状态，除了浮点的 CSR 和数据寄存器。

当的状况被设置为 Off 时，任何指令试图读写相应的状态都会导致一个异常。当的状况是 Initial 时，对应的状态应当具有一个初始的常数值。当的状况是 Clean 时，对应的状态可能与 Initial 时的状态不同，但与上下文切换时保存的最后的值相匹配。当的状况是 Dirty 时，对应的状态可能自上次上下文保存时以来，已经被修改。

当进行一个上下文保存时，负责的特权代码仅在对应的状态的状况是 Dirty 时，才需要将上下文写入存储器，然后就可以复位其状况为 Clean。当进行一个上下文恢复时，仅在状况是 Clean 时（在恢复时，应该永远不会是 Dirty），才需要从存储器中读取上下文。如果状况是 Initial，在恢复上下文时，上下文必须被设置为一个初始值，以避免一个安全漏洞，但这不需要访问存储器即可完成。例如，浮点寄存器可以全部初始化为立即数值 0。

当继续一个用户上下文时，FS 字段和 XS 字段可被特权代码设置，并在保存上下文之前，被特权代码读取。在执行指令时，状况字段也会被更新，而不管特权模式如何。

用户模式 ISA 扩展通常包含额外的用户模式状态，而这些状态可能要比基本整数寄存器大得多。这些扩展可能仅仅用于某些应用，或者仅在某个应用中使用一段很短的时间。为了提高性能，用户模式扩展可以定义额外的指令，以允许用户模式软件将单元置为初始状态或者甚至可以关闭该单元。

例如，一个协处理器在使用前需要被配置（configured），而在使用完之后，需要“去掉配置（unconfigured）”。去掉配置的状态可以表示为上下文保存时的 Initial 状态。如果同一个应用程序一直保持在“去掉配置”、“配置（将把状况设置为 Dirty）”之间运行，那么在去掉配置的指令时，并不需要真正的重新初始化状态，因为所有状态对于用户进程来说都是本地的，也就是说，在上下文恢复时（不是在每一个去掉配置的时候），Initial 状况仅导致协处理器状态被初始化到一个常数值。

执行一条用户模式指令来禁用一个单元，将其置为 Off 状态，那么在它被再次打开之前，任何试图使用该单元的指令，都会导致产生一个非法指令异常。一条用户模式指令打开一个单元，同时必须保证该单元的状态被正确地初始化，因为这个单元可能被另外的上下文期间使用了。

表 3.5 给出了对于 FS 或者 XS 状况位所有可能的状态转变。注意标准浮点扩展并不支持用户模式去掉配置或者禁用/使能指令。

当前状态行为	Off	Initial	Clean	Dirty
在特权代码中保存上下文时				
保存状态？ 下一状态	No Off	No Initial	No Clean	Yes Clean
在特权代码中恢复上下文时				
恢复状态？ 下一状态	No Off	Yes，变为 Initial Initial	Yes，从存储器 Clean	N/A（不存在） N/A（不存在）
执行指令读状态				
动作？ 下一状态	异常 Off	执行 Initial	执行 Clean	执行 Dirty
执行指令修改状态，包括进行配置				
动作？ 下一状态	异常 Off	执行 Dirty	执行 Dirty	执行 Dirty
执行指令去掉配置				
动作？ 下一状态	异常 Off	执行 Initial	执行 Initial	执行 Initial
执行指令禁用单元				
动作？ 下一状态	执行 Off	执行 Off	执行 Off	执行 Off
执行指令使能单元				
动作？ 下一状态	执行 Initial	执行 Initial	执行 Initial	执行 Initial

表 3.5：对于 FS 或者 XS 状况位所有可能的状态转变

提供了标准特权指令来初始化、保存和恢复扩展状态，通过将状态作为朦胧的对象（opaque object）看待，将特权代码与添加扩展的状态具体细节相隔离。

许多协处理器扩展仅被用于有限的上下文，允许软件安全地去掉配置或者完成后禁用掉单元。这样可以减少上下文切换时，具有较大状态的协处理器开销。

我们将浮点状态与其它扩展状态相分离，因为当浮点单元存在时，浮点寄存器是标准调用约定的一部分，因此用户模式软件不知道什么时候可以安全地禁用该浮点单元。

XS 字段为所有添加的扩展状态提供了一个指示，但是可以在扩展内部维护一些额外的微体系结构位，以便进一步减少上下文保存和恢复开销。

SD 位是只读位，如果 FS 或者 XS 的位表明是 Dirty 状况时，SD 位被置为 1（也就是说，SD=((FS==11) OR (XS==11))）。这样就允许特权代码快速的判断什么时候不需要进行额外的上下文保存，除了整数寄存器和 PC 之外。

浮点单元状态总是使用标准指令（F、D 和/或 Q）来初始化、保存和恢复的，而特权代码必须注意 FLEN，以决定为每个 f 寄存器预留多少空间。

在一个管理员级 OS 中，任何额外的用户模式状态，应当使用 SBI 调用来进行初始化、

保存和恢复，这些额外的上下文被当做一个固定大小的朦胧的对象（opaque object）看待。实现初始化、保存、恢复的 SBI 调用，可能需要额外的、实现相关的特权指令，在一个协处理器内实现初始化、保存、恢复微体系结构状态。

所有特权模式共享同一个 FS 和 XS 位。在一个具有多个特权模式的系统中，管理员模式通常将 FS、XS 位直接用于记录与管理员级保存的上下文相关的状况。其他更高级的特权模式应当在保存和恢复相应上下文对应的扩展状态时，更加保守，但是可以根据 Off 状态来避免保存和恢复，根据 Initial 状态来避免保存状态。

做任何合理的使用情形下，在用户和管理员级之间进行的上下文切换次数，要多于在其他特权级之间进行上下文切换的次数。注意协处理器并不需要将它们的上下文进行保存和恢复，以服务一个异步中断，除非这个中断会导致一个用户级上下文切换。

3.1.9 机器自陷向量基址寄存器（mtvex）

mtvec 是一个可读/可写的 XLEN 位寄存器，保存着 M-mode 自陷向量的基址。

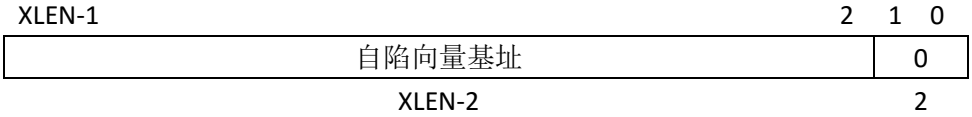


图 3.5：机器自陷向量基址寄存器（mtvex）

mtvec 寄存器必须总是被实现的，但是可以包含一个硬连线只读的值。两个标准值，0xF…FFE00 和 0x0…00100，分别为自陷向量的高地址和低地址，并且在复位后，两个值之一必须出现在 mtvec 中。标准复位向量，对于自陷向量的高地址、低地址来说，要么是 0xF…FFF00 要么是 0x0…00200。

mtvec 寄存器可以被实现为可读/可写的寄存器，以支持可变的自陷向量基址。mtvec 寄存器中可被改写的位，依据实现不同而不同，但是如果任何位都可写，那么必须支持上述两个标准值。mtvec 寄存器中的值必须总是对齐到 4 字节边界的（最低 2 位总是零）。如果任何位都可以写，那么符号位也必须是可写的，而这个符号位必须向下扩展，从符号位一直到下一个可写位。读取一个可变的 mtvec 寄存器的值，必须总是与处理自陷时，生成 PC 基地址的值相一致。

地址	处理函数（Handler）
高自陷向量地址	
0xF…FE00	来自用户模式的自陷
0xF…FE40	来自管理员模式的自陷
0xF…FE80	来自 hypervisor 模式的自陷
0xF…FEC0	来自机器模式的自陷
0xF…FEFC	不可屏蔽中断
0xF…FF00	复位向量
低自陷向量地址	
0x100	来自用户模式的自陷
0x140	来自管理员模式的自陷
0x180	来自 hypervisor 模式的自陷
0x1C0	来自机器模式的自陷
0x1FC	不可屏蔽中断
0x200	复位向量

表 3.6：M-mode 自陷向量地址的标准位置，要么在高存储器位置，要么在低存储器位置

一个在特权级 P 的自陷，导致跳转到地址 mtvec＋P×0x40。不可屏蔽中断导致跳转到地址 mtvec＋0xFC。实现可以定义额外的自陷向量入口点，以允许更加快速地对某些自陷情况进行识别和服务。

我们允许相当灵活地实现自陷向量基地址。一方面我们不希望对低端实现造成大量状态位的负担，但另一方面，我们希望对大系统允许更多的灵活性。不同的系统上下文，可以强制复位和处理代码处在高或者低的存储器位置，因此我们将两者都作为标准硬连线向量。

3.1.10机器自陷转移寄存器（mtdeleg）

缺省情况下，任何特权级的自陷，都是在机器模式下被处理的，尽管一个机器模式处理函数可以使用 mrts 指令和 mrth 指令快速地将自陷重定向回相应的特权级（3.2.2 节）。为了提高性能，实现可以在 mtdeleg 寄存器中提供单独的读/写位，来指示某些自陷可以直接被较低特权级处理。

机器自陷转移寄存器（mtdeleg）是一个可读/可写的 XLEN 位寄存器，必须总是被实现的，但是可以包含一个只读的零值，表明硬件总是将所有自陷定向到机器模式。

如果存在 hypervisor 模式，那么 mtdeleg 寄存器中某个位为 1，则将相应的 U-mode、S-mode 或者 H-mode 自陷定向到 H-mode 自陷处理函数。而 H-mode 可以设置 htdeleg 寄存器中的相应位为 1，将 S-mode 或者 U-mode 发生的自陷转移到 S-mode 自陷处理函数。

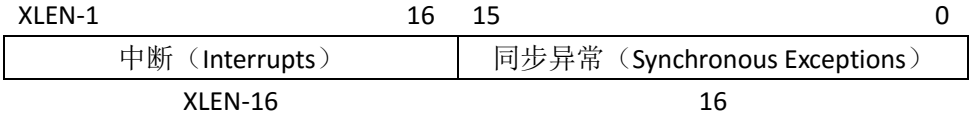


图 3.6：机器自陷转移寄存器（mtdeleg）

如果只存在管理员模式，那么设置 mtdeleg 中的一位，将把相应的 S-mode 或者 U-mode 自陷转移到 S-mode 自陷处理函数。

如果 hypervisor 模式和管理员模式都没有被实现，那么 mtdeleg 寄存器应当被硬连线为零。

一个实现可以选择可转移自陷的子集（来实现），支持转移的位，可以通过首先将所有位置都写 1，然后通过读取 mtdeleg 寄存器的值，来判断哪些位置保留了 1。

mtdeleg 寄存器的低 16 位的位空间被分配给了每一个同步异常，如表 3.7 所示，这些位空间的 index 等于 mcause 寄存器的返回值（例如，设置位 8 为 1，允许用户模式环境调用被转移到一个较低特权自陷处理函数）。

位 16 及以上的位，保存着单个中断的自陷转移位，这些位的 layout 等于 mip 寄存器左移 16 位（例如，STIP 中断转移控制在 mtdeleg 中处于第 21 位）。

3.1.11机器中断寄存器（mip 和 mie）

mip 寄存器是一个可读/可写的 XLEN 位寄存器，包含了挂起中断（pending interrupts）的信息，而 mie 是一个对应的 XLEN 位可读/可写寄存器，包含了中断使能位。mip 中只有对应于软件中断（SSIP、HSIP、MSIP）的较低的位是通过它的 CSR 地址可写的。mip、mie 寄存器的受限视图在 H 和 S 特权级 ISA 中分别出现在 hip/hie 和 sip/sie 中。

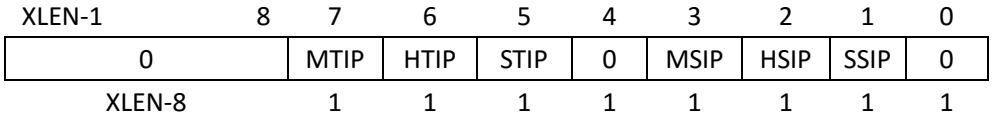


图 3.7：机器中断挂起寄存器（mip）

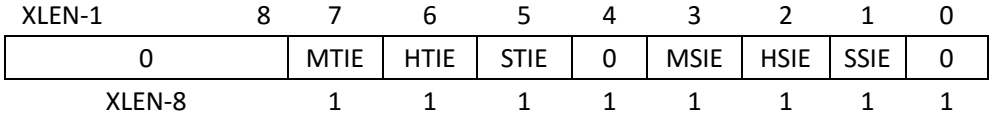


图 3.8：机器中断使能寄存器（mie）

空间保留给未来可能加入的用户级软件中断。

MTIP、HTIP、STIP 位分别对应于机器级、hypervisor、管理员定时器中断挂起位，这些位，分别通过写 mtimecmp、htimecmp、stimecmp 寄存器来清除（**译者注：MTIP、HTIP、STIP 位都是只读的，由硬件自动置位**）。

对于每一个支持的非用户特权模式，都有一个单独的定时器中断使能位，对于 M-mode、

H-mode 和 S-mode，分别叫做 MTIE、HTIE 和 STIE。如果不支持某个特权级，则相应的中断使能位被硬连线到零。

空间保留给未来可能加入的用户级定时器中断。

对于每一个支持的非用户特权模式，都有一个单独的软件中断挂起位（MSIP、HSIP、SSIP），它们可以被运行在对应特权级或者更高特权级的本地硬件线程（hart）的运行代码，通过访问 CSR 来读和写。如果不支持某个特权级，则相应的中断挂起位被硬连线到零。机器级的 MSIP 位也可以被远程硬件线程修改，以提供 M-mode 处理器间中断。较低特权级的处理器间中断，是通过对 SEE 或者 HEE 进行 SBI 或者 HBI 调用来实现的，它们最终可能导致一次 M-mode 的写，以接收硬件线程的 MSIP 位。

当硬件线程运行在某个特权级或者更高特权级时，对应特权级的软件中断，如果 mie 中相应的 SIE 位被清零或者 mstatus 寄存器中的全局 IE 位被清零，则被禁止。

我们仅允许硬件线程当运行在 hypervisor 或者管理员模式时，才能直接写它自己的 HSIP 和 SSIP 位，因为其他 hypervisor 级或管理员级硬件线程可能被虚拟化并可能被更高特权级重新调度（descheduled）。因为这个原因，我们依靠 SBI 和 HBI 调用来提供处理器间中断。M-mode 硬件线程是不会被虚拟化的，并可以通过设置其他硬件线程的 MSIP 位来直接中断其他的硬件线程，这通常通过非缓存的（uncached）写存储器映射控制寄存器来完成，可能是在一个硬件平台的一个全局中断控制器中实现。

实现可能对这些寄存器加入额外的机器级中断源。

不可屏蔽中断并不通过 mip 寄存器可见，因为它在执行 NMI 自陷处理函数时是隐藏已知的。

3.1.12机器定时器寄存器（mtime、mtimecmp）

M-mode 包含一个由 mtimecmp 寄存器和实时计数器 mtime 提供的定时器机构。硬件平台必须提供一种机制来判断 mtime 的时基（timebase），这个时基必须运行在一个固定的频率。

mtimecmp 寄存器在所有 RV32、RV64 和 RV128 系统上都是 32 位精度的。当 mtime 寄存器的低 32 位与 mtimecmp 寄存器的低 32 位相同时，将产生一个定时器中断。这个中断将持续存在，直到通过写 mtimecmp 寄存器它才被清除。这个中断只有在当中断被使能且 mie 寄存器中的 MTIE 被置位时，才被处理。

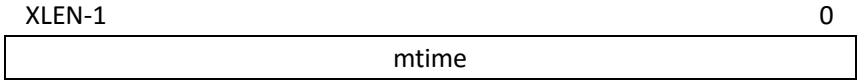


图 3.9：机器定时器寄存器

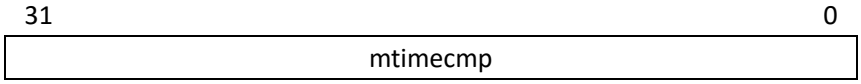


图 3.10：机器定时器比较寄存器

这个定时器机制被定义为使用墙钟定时器（wall-clock time）而不是一个周期计数器（cycle counter），以支持现代处理器通过动态电压和频率调节，运行在高度变化的时钟频率上以节约能量。简单的固定频率系统可以对周期计数和墙钟定时器使用同一个时钟。

提供真正的实时时钟（RTC）相对来说代价比较昂贵（需要一个晶体振荡器或者 MEMS 振荡器），并且需要在系统断电之后还要持续运行，因此在一个系统中通常只有一个。通过提供一个底层的实时时钟（RTC），我们就可以通过在静态（就是非增长的）寄存器中保存增量值（delta value），来实现任何的虚拟定时器。通过一个保存 mtime，实现将真实地读取 RTC，将预期的 RTC 值减去当前 RTC 值，并对每一个硬件线程将这个差值保存到 mtime 寄存器中。当读取 mtime 时，底层的 RTC 再次被读，保存在 mtime 中的差值和 RTC 的读取值相加，形成指令的返回结果。相同的方法可被用于每一特权级的各种墙钟定时器，以及计算正确的定时器比较值。

*在可变频率系统中的一个问题是，实时时钟（RTC）和比较寄存器通常被保存在一个不同于处理器的单独时钟域中，因此访问这个寄存器以及中断信号将受到时钟域穿越（clock-domain crossing）的影响（**译者注：电路上，为了避免不同时钟域之间通信出现亚稳态，通常采用两级或者多级同步器机制来处理，这将导致信号从一个时钟域传到另外一个时钟域，要花额外多的时间**）。*

3.1.13机器 Scratch 寄存器（mscratch）

mscratch 寄存器是一个 XLEN 位可读/可写的寄存器，专用于机器模式。典型的，它被用于保存一个指向机器模式硬件线程本地的上下文空间的指针，并在一个 M-mode 自陷处理函数入口处，与一个用户寄存器进行交换。

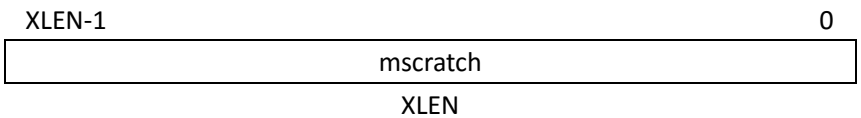


图 3.11：机器模式 scratch 寄存器

MIPS ISA 分配了两个用户寄存器（k0/k1）用于操作系统。虽然 MIPS 的方案提供了一种快速而简单的实现，但是它也减少了可用用户寄存器，并且不能扩展到更多的特权级，或者扩展到嵌套自陷。它也需要在返回用户级之前清除这两个寄存器，以避免潜在的安全漏洞并提供确定性的调试行为。

RISC-V 用户 ISA 被设计成支持许多可能的特权系统环境，因此我们并不想由于任何操作系统相关的特性而更改用户级 ISA。RISC-V CSR 交换指令可以快速地对 mscratch 寄存器进行保存/恢复。与 MIPS 设计不同，OS 可以在用户上下文运行期间依靠 mscratch 寄存器保存的一个值。

*对于硬实时系统，某些系统使用一种更为复杂的寄存器分组（banking）方案，它将分离的中断上下文寄存器分组映射进体系结构寄存器命名空间，以提供非常低的中断处理延迟（译**者注：例如 ARM 中在不同模式下，r0 有不同的物理寄存器，比如 r0_fiq 等**）。我们正在为硬实时系统探索一种不同的方法，而不是提供多个完全的寄存器上下文，来减少软件复杂性和提高性能。*

3.1.14机器异常程序计数器（mepc）

mepc 是一个 XLEN 位可读/可写寄存器，其格式如图 3.12 所示。mepc 的最低位（mepc[0]）永远是零。在那些并不支持 16 位指令对齐的指令集扩展实现上，mepc 的最低 2 位（mepc[1:0]）永远是零。

mepc 寄存器永远不能保存一个导致指令地址非对齐异常的 PC 值。

当处理一个自陷时，mepc 被写入碰到异常的那条指令的虚拟地址。

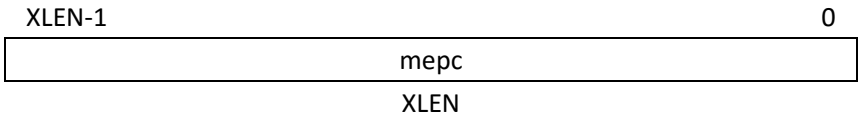


图 3.12：机器异常程序计数器（mepc）

3.1.15机器原因寄存器（mcause）

mcause 是一个 XLEN 位可读/可写寄存器，其格式如图 3.13 所示。如果是中断引起的异常，则 Interrupt 位被置为 1。Exception Code 字段保存了最近一次异常的标识代码。中间的位读取时返回 0，并且应当写入 0，以支持未来对 Exception Code 字段的扩展。表 3.7 列出了可能的机器级异常代码。

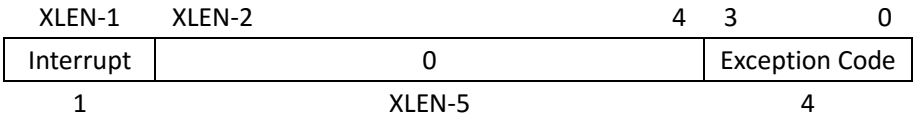


图 3.13：机器原因寄存器 mcause

我们并不区分特权指令异常和非法操作码异常。这简化了体系结构并隐藏

了一个实现所支持的更高特权级指令细节。在特权级处理自陷时，可以实现一种策略以决定是否需要区分，如果需要，一个给定的操作码是否应当被认为是非法的还是特权的。

中断和其他自陷区分，可以通过一个在 mcause 寄存器值符号位上的分支指令来完成。一次左移就可以去掉 interrupt 位，并将 Exception Code 扩展为一个自陷向量表的索引。

中断（Interrupt）	异常代码（Exception Code）	描述
0	0	指令地址未对齐
0	1	指令访问失效（fault）
0	2	非法指令
0	3	断点
0	4	Load 地址未对齐
0	5	Load 访问失效（fault）
0	6	Store/AMO 地址未对齐
0	7	Store/AMO 访问失效（fault）
0	8	从 U-mode 进行环境调用
0	9	从 S-mode 进行环境调用
0	10	从 H-mode 进行环境调用
0	11	从 M-mode 进行环境调用
0	≥12	*保留*
1	0	软件中断
1	1	定时器中断
1	≥2	*保留*

表 3.7：机器原因寄存器（mcause）值

3.1.16机器坏地址寄存器（mbadaddr）

mbadaddr 是一个 XLEN 位可读/可写寄存器，其格式如图 3.14 所示。当出现一个取指地址未对齐异常、取指访问异常、load/store 地址未对齐异常、load/store 访问异常时，mbadaddr 寄存器被写入导致失效的地址（faulting address）。对于其他异常，mbadaddr 寄存器的值是未定义的。

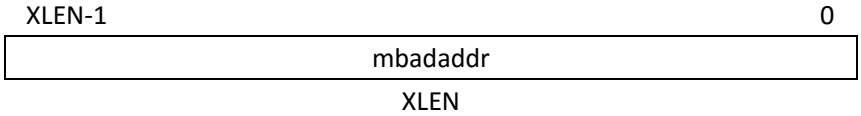


图 3.14：机器坏地址寄存器

在具有可变指令的 RISC-V 系统上，取指访问失效时，mbadaddr 指向引起失效的指令部分，而 mepc 指向该指令的起始。

3.2 机器模式特权指令

3.2.1 改变特权级的指令

改变特权级的指令被编码到 PRIV 次要操作码之下。ECALL（环境调用，Environment Call）指令和 EBREAK（环境断点，Environment Breakpoint）指令在所有特权级下均可用，而 ERET（环境返回，Environment Return）仅在特权级 S、H 和 M 下可用。

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
ERET	0	PRIV	0	SYSTEM

ECALL 指令被用于向更高特权级发起请求。执行环境的二进制接口将定义该请求的参数是如何传递的，但通常情况下这是通过在整数寄存器文件中定义位置来实现的。执行一条 ECALL 指令将导致一个环境调用异常（Environment Call exception）。

我们将用户 ISA 中的 SCALL 指令重命名为 ECALL 指令，使其更加通用。这个重命名并不改变用户模式指令的操作码编码或者功能，但是需要对汇编器/反汇编器进行修改以支持新的名字。

EBREAK 指令被调试器用于将控制转回到调试环境。执行一条 EBREAK 指令将导致一个断点异常（Breakpoint exception）。

本标准并不允许在 EBREAK 指令编码中使用未使用的位来编码调试信息，因为调试信息最好保存在一个由 epc 寄存器索引的哈希表中。

当处理完一个自陷后，ERET 指令被用于返回到自陷产生的特权级。除了如 3.1.5 节所述操作特权栈之外，ERET 指令还将 pc 设置为 Xepc 寄存器保存的值，此处 X 是 ERET 指令被执行时的特权模式（S、H 或 M）。

3.2.2 自陷重定向指令

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

MRTS 指令（机器重定向自陷到管理员，Machine Redirect Trap to Supervisor）将自陷处理从 M-mode 转移到 S-mode。MRTS 指令将特权模式改变为 S，并且将 pc 设置为管理员的自陷处理函数，（这个自陷处理函数入口）被保存在 stvec 寄存器中。另外，mepc、mcause、mbadaddr 寄存器中的值被分别复制到 sepc、scause、sbadaddr 寄存器中。

MRTH 指令（机器重定向自陷到 Hypervisor，Machine Redirect Trap to Hypervisor）的定义类似，但是是将自陷处理转移到 H-mode 的 htvec。mepc、mcause、mbadaddr 寄存器被分别复制到 hepc、hcause、hbadaddr 寄存器中。

简单的实现可以将所有的自陷定向到一个 M-mode 自陷处理函数，即使它们的目标是一个较低的特权模式。自陷重定向指令允许 M-mode 自陷处理函数快速地将控制传输到一个较低特权模式的自陷处理函数。

操作码空间被预留给HRTS 指令，该指令将把自陷从H-mode 定向到S-mode。为了帮助水平方向的用户模式自陷，我们也为 MRTU、HRTU 和 SRTU 指令预留了空间。

3.2.3 等待中断

等待中断指令（WFI）为实现提供了一个提示，即当前硬件线程可以被暂停直到需要处理一个中断。WFI 指令的执行也可用于告知硬件平台，合适的中断应该路由到这个硬件线程。WFI 指令可在 S、H 和 M 特权级使用。

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

如果当硬件线程被暂停时，出现一个使能的中断或者后来出现了一个，将会在下一条指令处执行中断异常，即自陷处理函数继续执行，并且 mepc＝pc＋4。

在下一条指令处执行中断异常和自陷，因此从自陷处理函数返回的时候，将在 WFI 指令之后继续执行代码。

WFI 指令仅是一个提示，并且一个合法的实现可以将 WFI 指令实现为一条 NOP 指令。

如果实现并没有在执行该指令时暂停硬件线程，那么中断将在包含 WFI 指令的空循环的某条指令处进行处理，并且从处理函数的一个返回，空循环将继续执行。

当 WFI 指令被执行时，中断可以被禁用，但是在任何（使能的或者禁用的）中断挂起时或变成挂起时，被暂停的硬件线程必须继续执行。如果任何被屏蔽的中断挂起或变成挂起，执行将在 pc＋4 处继续，并且软件必须决定对任何挂起的中断采取什么样的动作。

通过允许当中断被禁用时唤醒，可以调用另一个中断处理函数的入口点，

并且不需要保存当前上下文，因为在 WFI 指令被执行之前，当前上下文可以被保存或者丢弃。

可以查询 mip、hip、sip 寄存器来判断在机器、hypervisor、管理员模式下任何中断是否出现。

由于实现可以自由地将 WFI 指令实现为 NOP 指令，因此软件在 WFI 指令之后的代码中必须明确地检测任何相关但是被禁用的中断，并且在没有检测到合适的中断时，循环回 WFI 指令。

同样的“等待事件”模板可能在将来的扩展用于等待一个存储器位置变化或者消息达到。

3.3 物理存储器属性

访问系统的物理存储器是由机器模式完成的。由于物理存储器空间的布局是高度系统相关的，因此本节描述为每一个物理地址区间指定属性的整体方法，而不是对某个给定的硬件平台特定的细节。

一个完整系统的物理存储器 map 包括各种存储器区域和各种存储器映射控制寄存器。一些存储器区域可能并不存在，一些存储器区域可能是只读的，一些可能不支持 subword 甚至 subblock 访问，一些可能不支持原子性操作，以及一些可能不支持 cache 一致性。类似的，存储器映射控制寄存器随着它们支持的访问宽度不同而不同，以及读、写访问是否有相关的副作用。

尽管许多系统在虚拟存储器页表中指定了这些属性，但是这将平台相关的信息注入到了一个虚拟化的层次中，并可导致系统错误，除非这些属性在每一个平台物理存储器区域的每一个页表项中被正确地初始化。另外，如果在物理存储器空间中指定属性，那么可用的页面大小可能不是最优的。

对于 RISC-V，我们将机器物理存储器属性规范分解成一个单独的定制硬件结构。许多情况下，每个物理地址区域的属性，在系统设计时，是已知的，并在每一个 RISC-V 处理器系统中可以被硬连线到存储器数据通路中。或者，可以提供机器级控制寄存器，在平台上每一个区域对应的粒度上，指定这些属性（例如，一个片上的 SRAM 可以灵活地划分为 cacheable 和 uncacheable 使用）。这些属性作用于任何物理存储器区域的访问，包括那些在底层有虚拟到物理存储器翻译的访问。

为了帮助系统调试，我们强烈地建议 RISC-V 处理器对非法物理存储器访问在核心内进行精确自陷，而不是从存储器子系统中将它们报告为非精确的机器检测错误。

3.4 物理存储器访问控制

为了减少失效（contain faults）和支持安全处理，最好限制一个在较低特权上下文中运行的硬件线程，对物理地址的访问。与前一节描述的物理存储器属性相类似，一个 RISC-V 系统应当提供每物理线程（per-hart）的控制寄存器，以允许可对每一块物理存储器空间指定其物理存储器访问特权（读、写、执行）。访问控制设置的粒度随着在硬件平台上的物理存储器空间不同而不同，并且某些区域的特权可以是被硬连线的。

这些机器模式物理存储器访问控制，可作用于一个运行在 U、S 或者 H 模式的硬件线程

的所有访问，还作用于硬件线程运行在 M 模式且 mstatus 寄存器中的 MPRV 位被置为 1 时的 load 和 store。与前一节描述的物理地址属性类似，非法物理存储器访问应当被精确地自陷。

3.5 Mbare 寻址环境

可以通过复位或者任何时候写入 mstatus 的 VM 字段，进入 Mbare 环境。

在 Mbare 环境中，所有的虚拟地址不经翻译直接转换为物理地址，去掉任何超出的高位（**译者注：例如物理地址只有 16 位，则虚拟地址超出 16 位以上的高位被丢弃**）。前些章节描述的物理存储器属性和访问控制可用于约束这些访问。

3.6 基址-边界环境

本节描述 Mbb 虚拟化环境，它提供了一种基址-边界翻译和保护机制。基址-边界有两种变种，Mbb 和 Mbbid，根据其是否只有一个基址-边界（Mbb）还是对指令取指和数据访问采用分离的基址-边界（Mbbid）。这种简单的翻译和保护机制具有低复杂度和确定的高性能的优点，因为在操作时永远不会存在 TLB 缺失。

3.6.1 Mbb：单个基址-边界寄存器（mbase，mbound）

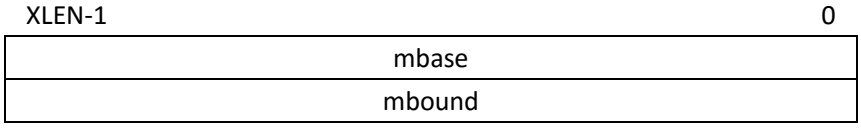


图 3.15：单个基址-边界寄存器

较简单的 Mbb 系统具有单个基址 mbase 和单个边界 mbound 寄存器。Mbb 可以通过在 mstatus 寄存器的 VM 字段写入 1 来使能。

基址-边界寄存器定义了一个连续的虚拟地址段，开始于虚拟地址 0 处，长度由 mbound 的值给出（字节）。这个虚拟地址段被映射到一个连续的物理地址段，其起始物理地址由 mbase 寄存器给出。

当 Mbb 工作时，所有较低特权模式（U、S、H）指令取指地址和数据地址被转换，这是通过将虚拟地址加上 mbase 获得物理地址的。同时虚拟地址与边界寄存器中的值进行比较。如果虚拟地址大于等于保存在 mbound 寄存器中的虚拟地址限制，则产生一个地址失效异常。

机器模式指令取指和数据访问并没有在 Mbb 中被翻译或者检查（除了当 mstatus 寄存器中的 MPRV 位被置为 1 时的 load 和 store 之外），因此机器模式的有效地址被当做物理地址。

3.6.2 Mbbid：分离的指令和数据基址-边界寄存器

Mbbid 方案将指令取指和数据访问的虚拟地址段相分离，允许单个物理指令段可被两个或者多个用户级虚拟地址空间共享，而每个分配了分离的数据段。Mbbid 可以通过在 mstatus 寄存器的 VM 字段写入 2 来使能。

图 3.16：分离的指令和数据基址-边界寄存器

分离的指令和数据基址-边界方案被用于著名的 Cray 超级计算机上，在将段放入物理存储器时，它避免了绝大多数与翻译和保护相关的运行时开销。

mibase、mibound 分别定义了指令段的物理起始地址和长度，而 mdbase、mdbound 分别指定了数据段的物理起始地址和长度。

数据虚拟地址段开始于地址 0，而指令虚拟地址段开始于虚拟地址空间的一半处，在最前面是 1 而后面是 XLEN-1 个 0 的地址处（例如，对于 32 位地址空间系统，是 0x8000_0000）。较低特权模式的指令取指的虚拟地址被首先检查，确保它们的最高位被置位，如果没有，将产生一个异常。然后最高被被当做 0，加上虚拟地址的基址，再检查边界。

数据和指令虚拟地址段不应当重叠，并且我们觉得保持潜在的零页面数据访问（使用来自于寄存器 x0 的一个 12 位偏移量）要比使用 JALR 及 x0 支持指令入口点更为重要。特别地，一条 JAL 指令可以直接访问所有 2MB 代码段。

为了简化链接，指令虚拟地址段起始地址应当是一个常数，与完整二进制的长度无关。放置在虚拟存储器的中点，可最小化分为两个段所需要的电路。

提供了 Mbbid 的系统必须同时提供 Mbb。写入 mbase 相关的 CSR 地址，应当将同样的值写入 mibase 和 mdbase，写入 mbound 应当将同样的值写入 mibound 和 mdbound，以提供兼容的行为。读 mbase 应该返回 mdbase 的值，而读 mbound 应当返回 mdbound 的值。当 VM 位被设置为 Mbb 时，指令取指将不再检查虚拟地址的最高位，并且在加上基址和检查边界时，不再将最高位复位为 0。

*虽然分离的机制允许单个物理指令段可被多个用户进程实例共享，它也有效地阻止了用户程序写入指令段（数据 store 被单独翻译），阻止从数据段中执行指令（指令取指被单独翻译）。这些限制可以阻止某些形式的安全攻击（**译者注：防止了在栈中执行缓冲区溢出攻击的代码**）。*

另一方面，许多现代编程系统要求，或者受益于某些形式的运行时生成代码，因此这些时候应当使用单个段的较简单的 Mbb 模式，这也是为什么在提

供 Mbbid 时需要支持 Mbb 模式。

第4章 管理员级 ISA

本章描述 RISC-V 管理员级体系结构，它包含一个公共核心，以及各种管理员级地址翻译和保护机制。管理员模式总是在一个由机器模式 mstatus 寄存器 VM 字段定义的虚拟存储器方案中运作的。管理员级代码是针对一个给定的 VM 方案书写的，在使用时不能更改 VM 方案。

管理员级代码依赖于一个管理员执行环境，初始化这个环境并进入管理员代码，这个管理员代码入口点，是由系统二进制接口（SBI）定义的。SBI 同时还定义了为管理员级代码提供管理员环境服务的功能函数入口点。

管理员模式被故意地限制了与底层物理硬件的交互，例如物理存储器和设备中断，以支持清晰的虚拟化。一个更传统、虚拟化不友好（virtualization-unfriendly）的操作系统，可以这样移植：通过使用 M-mode 来初始地将物理存储器映射入管理员虚拟存储器地址空间、并将设备中断转移到 S-mode。

4.1 管理员 CSR

为管理员提供了一系列的 CSR。

---

*管理员应当仅仅看到那些管理员级操作系统可见的 CSR 状态。特别地，没有存在（或者不存在）更高特权级（hypervisor 或机器级）可见的 CSR，而管理员访问性的信息（**译者注：即管理员压根不知道更高特权级有没有这些 CSR**）。更高特权级可见的额外 CSR，将被编码，如果处理器正在比管理员级更高的特权级上运行。*

许多管理员 CSR 是对应机器模式 CSR 的子集，应当首先阅读机器模式章节以帮助理解管理员级 CSR 描述。

4.1.1 管理员状态寄存器（sstatus）

sstatus 是一个 XLEN 位可读/可写寄存器，其格式如图 4.1 所示。sstatus 寄存器追踪了处理器当前的操作状态。

XLEN-1	XLEN-2 17	16	15 14	13 12	11 5	4	3	2 1	0
SD	0	MPRV	XS[1:0]	FS[1:0]	0	PS	PIE	0	IE
1	XLEN-18	1	2	2	7	1	1	2	1

图 4.1：管理员模式状态寄存器

PS 位表明了在进入管理员模式之前，一个硬件线程运行在哪一个特权级。当处理一个自陷时，如果这个自陷原来来自与用户模式，则 PS 被置为 0，否则被置为 1。当执行一条

ERET 指令（参考 3.2.1 节）从自陷处理函数返回时，如果 PS 为 0，则特权级被置为用户模式，否则如果 PS 为 1，则被置为管理员模式。

IE 位使能或者禁用管理员模式下的所有中断。当 IE 被清零时，当处于管理员模式时中断并不被处理。当硬件线程运行在用户模式时，IE 的值被忽略，并且管理员级中断是使能的。管理员可以使用 sie 寄存器禁用单个中断源。

PIE 位表明了在进入管理员模式前中断是否是使能的。当处理一个自陷时，PIE 被置为 IE，而 IE 被置为 0。当执行一条 ERET 指令时，IE 被设置为 PIE。

4.1.2 sstatus 寄存器中的存储器特权

MPRV 位修饰了 load 和 store 执行的特权级。当 MPRV＝0 时，存储器被如同通常一样保护。当 MPRV＝1 时，数据存储器的保护就如同 PS 位给出了当前的特权级（也就是说，当 PS＝0 时是用户级，当 PS＝1 时是管理员级）。指令存储器保护不受 MPRV 位的影响。

当出现一个异常时，MPRV 被复位为 0。

*MPRV 机制允许管理员软件以用户方式来引用存储器，而不需要从用户级访问保护的存储器。（**译者注：就像是管理员软件以用户身份来访问数据存储器。**）*

4.1.3 管理员中断寄存器（sip 和 sie）

sip 寄存器是一个 XLEN 位可读/可写寄存器，包含了挂起中断的信息，而 sie 寄存器是一个对应的 XLEN 位可读/可写寄存器，包含了中断使能位。

定义了两种类型的中断：软件中断和定时器中断。当前硬件线程通过向 sip 寄存器中的软件中断挂起（SSIP）位写入 1 来触发一个软件中断。一个挂起的软件中断，可以通过向 sip 中的 SSIP 位写入 0 来清除。sip 寄存器中其他的位都是只读的。当 sie 寄存器中的软件中断使能（SSIE）位被清除时，软件中断被禁用。

处理器间中断是通过 SBI 调用的方式发向其他硬件线程的，这最终会导致接收方硬件线程的 sip 寄存器中的 SSIP 位被置为 1。

如果 sip 寄存器中的 STIP 位被置为 1，那么将挂起一个定时器中断。当 sie 寄存器中的 STIE 位被清除时，定时器中断被禁用。通过写入 stimecmp 寄存器来清除一个挂起的定时器中断。

XLEN-1	6	5	4	2	1	0
0		STIP	0		SSIP	0
XLEN-6		1	3		1	1

图 4.2：管理员中断挂起寄存器（sip）

XLEN-1	6	5	4	2	1	0
0		STIE	0		SSIE	0
XLEN-6		1	3		1	1

图 4.3：管理员中断使能寄存器（sie）

4.1.4 管理员定时器寄存器（stime，stimecmp）

管理员模式包含一个由 stimecmp 寄存器和实时计数器 stime（stime 是用户只读 time 寄存器的管理员可读/可写版本）提供的定时器机构。SBI 必须提供一种机制来判断 stime 的时基（timebase），这个时基必须运行在一个固定的频率。

stimecmp 寄存器在所有 RV32、RV64 和 RV128 系统上都是 32 位精度的。当 stime 寄存器的低 32 位与 stimecmp 寄存器的低 32 位相同时，将产生一个定时器中断。这个中断将持续存在，直到通过写 stimecmp 寄存器它才被清除。这个中断只有在当中断被使能且 sie 寄存器中的 STIE 被置位时，才被处理。

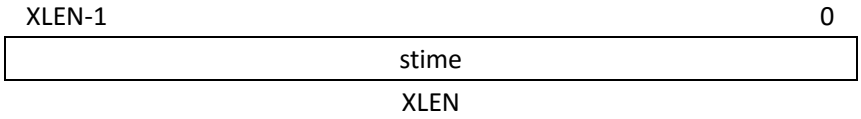


图 4.4：管理员定时器寄存器

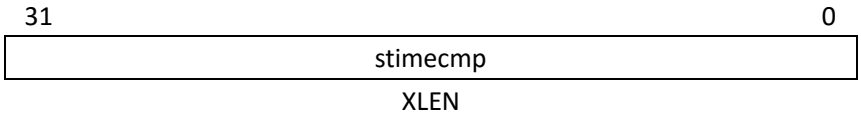


图 4.5：管理员定时器比较寄存器

4.1.5 管理员 scratch 寄存器（sscratch）

sscratch 寄存器是一个 XLEN 位可读/可写的寄存器，专用于管理员模式。典型的，当一个硬件线程在执行用户代码的时候，sscratch 被用于保存一个指向管理员模式硬件线程本地的上下文空间的指针。在一个自陷处理函数入口处，sscratch 与一个提供初始工作寄存器的用户寄存器进行交换。

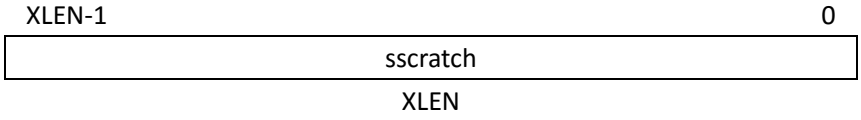


图 4.6：管理员模式 scratch 寄存器

4.1.6 管理员异常程序计数器（sepc）

sepc 是一个 XLEN 位可读/可写寄存器，其格式如图 4.7 所示。sepc 的最低位（sepc[0]）永远是零。在那些并不支持 16 位指令对齐的指令集扩展实现上，sepc 的最低 2 位（sepc[1:0]）永远是零。

当处理一个自陷时，sepc 被写入碰到异常的那条指令的虚拟地址。

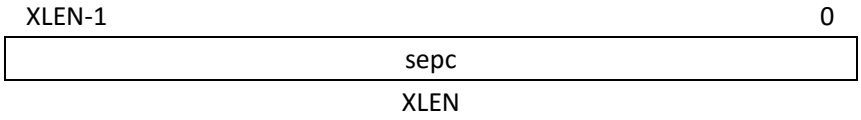


图 4.7：管理员异常程序计数器（sepc）

4.1.7 管理员原因寄存器（scause）

scause 是一个 XLEN 位只读寄存器，其格式如图 4.8 所示。如果是中断引起的异常，则 Interrupt 位被置为 1。Exception Code 字段保存了最近一次异常的标识代码。中间的位读取时返回 0，并且应当写入 0，以支持未来对 Exception Code 字段的扩展。表 4.1 列出了可能的管理员级异常代码。

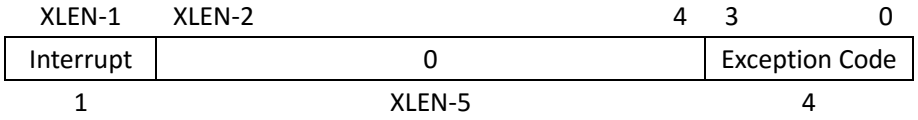


图 4.8：管理员原因寄存器 scause

中断（Interrupt）	异常代码（Exception Code）	描述
0	0	指令地址未对齐
0	1	指令访问失效（fault）
0	2	非法指令
0	3	断点
0	4	保留
0	5	Load 访问失效（fault）
0	6	AMO 地址未对齐
0	7	Store/AMO 访问失效（fault）
0	8	环境调用
0	≥9	*保留*
1	0	软件中断
1	1	定时器中断
1	≥2	*保留*

表 4.1：管理员原因寄存器（scause）值

4.1.8 管理员坏地址寄存器（sbadaddr）

sbadaddr 是一个 XLEN 位只读寄存器，其格式如图 4.9 所示。当出现一个取指地址未对齐异常、取指访问异常、AMO 地址未对齐异常、load/store 访问异常时，sbadaddr 寄存器被写入导致失效的地址（faulting address）。对于其他异常，sbadaddr 寄存器的值是未定义的。

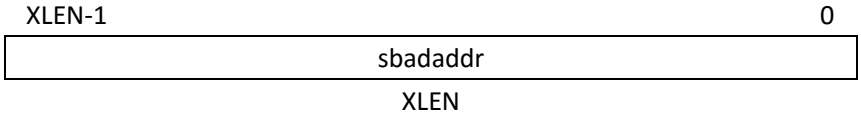


图 4.9：管理员坏地址寄存器

在具有可变指令的 RISC-V 系统上，取指访问失效时，sbadaddr 指向引起失效的指令部分，而 sepc 指向该指令的起始。

4.1.9 管理员页表基址寄存器（sptbr）

sptbr 是一个 XLEN 位可读/可写寄存器，其格式如图 4.10 所示。sptbr 寄存器仅出现在支持分页虚拟存储器的系统中。该寄存器保存了当前根页表（root page table）的管理员物理地址，它必须在 4KB 边界对齐。

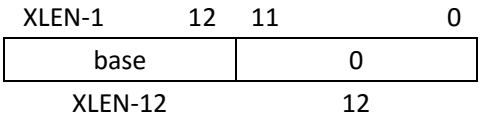


图 4.10：管理员页表基址寄存器 sptbr

对许多应用来说，页面大小的选择对性能有巨大的影响。较大的页面大小，提升了 TLB 命中率并放松了虚拟-索引（virtually-indexed）、物理-标签（physically-tagged）Cache 上的关联性约束。同时，较大的页面加剧了内部碎片化，浪费了物理存储器和可能的 Cache 容量。

经过大量的思考，我们选择在 RV32 和 RV64 上都采用传统的 4KB 页面大小。我们预期这个决定将简化底层运行时软件和设备驱动的移植。TLB 命中问题在现代操作系统中支持通过透明超页（superpage）来加以改善[2]。另外，多级 TLB 层次相比起它们映射的地址空间的多级 cache 层次而言，要便宜得多。

4.1.10管理员地址空间 ID 寄存器（sasid）

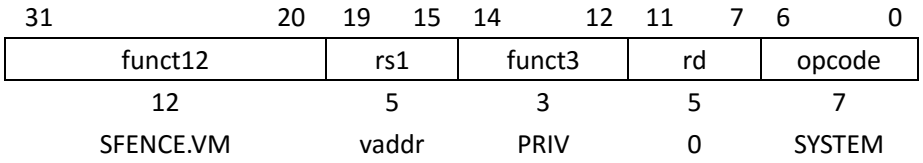
sasid 是一个 ASIDLEN 位的可读/可写寄存器，其格式如图 4.11 所示，仅出现在支持分页虚拟存储器的系统中。该寄存器指明了在一个每-地址-空间（per-address-space）基础上利用地址-翻译栅栏的当前地址空间（This register specifies the current address space to facilitate address-translation fences on a per-address-space basis.）。SBI 应该提供一种机制以获取 ASIDLEN，它是实现定义的，如果不支持 ASID，那么这个值可能是零。

图 4.11：管理员地址-空间 ID 寄存器

4.2 管理员指令

除了 3.2.1 节定义的 ECALL、EBREAK、ERET 指令之外，提供了一条新的管理员级指令。

4.2.1 管理员存储器管理栅栏指令



管理员存储器管理栅栏指令 SFENCE.VM 被用于存储器中存储器管理数据结构（in-memory memory-management data structures）和当前执行的同步更新。SFENCE.VM 指令保证了任何对存储器中存储器管理数据结构（例如，页表）的更新，在这个 RISC-V 线程未来执行的指令时已经生效。

SFENCE.VM 指令被用于刷新（flush）任何与地址翻译相关的本地硬件 cache。这条指令被认为是一个栅栏而不是一条 TLB 刷新（flush），以提供更清晰的语义表明哪些指令受到这个刷新操作的影响，以支持更广泛丰富的动态 cache 结构和存储器管理方案。SFENCE.VM 指令也被更高特权级用于同步页表写和地址翻译硬件。

注意该指令对其他 RISC-V 线程的（地址）翻译并没有影响，这需要另外通知。一种方法是使用 1）一个本地数据栅栏来确保本地写全局可见，然后 2）一个到别的线程的处理器间中断，然后 3）一个在远程线程的中断处理函数中的本地 SFENCE.VM 指令，最后 4）signal 回原来的线程告知操作已经完成。当然，这个 RISC-V 相当于 TLB 作废（This is, of course, the RISC-V analog to a TLB shootdown.）。或者，实现可以提供直接的硬件以支持远程 TLB 作废。TLB 作废由一个 SBI 调用来处理以屏蔽实现的细节。

SFENCE.VM 指令的行为依赖于 sasid 寄存器的当前值。如果 sasid 是非零值，SFENCE.VM 指令只对当前地址空间内的地址翻译起作用。如果 sasid 是零，SFENCE.VM 指令对所有地址空间的地址翻译都起作用。在这种情形下，它也影响全局映射（global mappings），这将在 4.5.1 节描述。

寄存器操作数 rs1 包含了一个可选的虚拟地址参数。如果 rs1＝x0，这个栅栏将影响所有的虚拟地址翻译。常见情况下，翻译数据结构的修改仅仅对单个地址映射（例如，一个页面）起作用时，rs1 可以被指定为这个映射内部的一个虚拟地址，使得翻译栅栏仅仅影响这个映射。

较为简单的实现可以忽略 sasid 寄存器的 ASID 值和 rs1 的虚拟地址，总是执行一个全局的栅栏。

4.3 管理员在 Mbare 环境中的操作

当在 mstatus 的 VM 字段选择 Mbare 环境时（3.1.6 节），管理员模式虚拟地址被截断并直接映射到管理员级物理地址。随后管理员物理地址在被直接转换成机器级物理地址之前，被各种物理存储器保护结构（3.3-3.4 节）检测。

4.4 管理员在基址边界环境中的操作

当在 mstatus 的 VM 字段选择 Mbb 或者 Mbbid 环境时（3.1.6 节），管理员模式虚拟地址根据相应的机器级基址和边界寄存器，被翻译和检测。结果的管理员级物理地址随后在被直接转换成机器级物理地址之前，被各种物理存储器保护结构（3.3-3.4 节）检测。

4.5 Sv32：基于页面的 32 位虚拟存储器系统

当 Sv32 被写入 mstatus 寄存器的 VM 字段时，管理员将工作在 32 位分页的虚拟存储器系统中。Sv32 被 RV32 系统支持，并被设计成包含了足够支持现代基于 Unix 操作系统的机制。

初始版本的 RISC-V 分页虚拟存储器体系结构被设计为直截了当支持现有的操作系统。我们构建了页表布局以支持一个硬件页表 walker。在高性能系统上，软件 TLB refill 是性能瓶颈，并且在去耦合专用协处理器时特别麻烦。一个实现可以选择将软件 TLB refill 使用一个机器模式自陷处理函数来实现，作为 M-mode 的一个扩展。

4.5.1 寻址和存储器保护

Sv32 实现支持一个 32 位虚拟地址空间，分割成 4KB 页面。一个 Sv32 虚拟地址被切分为一个虚拟页编号（virtual page number，VPN）和页内偏移量（offset），如图 4.12 所示。当在 mstatus 的 VM 字段选择 Sv32 虚拟存储器模式时，管理员虚拟地址通过一个两级页表被翻译为管理员物理地址。20 位的 VPN 被翻译为一个 22 位的物理页编号（physical page number，PPN），而 12 位的页内偏移量不被翻译。结果的管理员级物理地址随后在被直接转换成机器级物理地址之前，被各种物理存储器保护结构（3.3-3.4 节）检测。

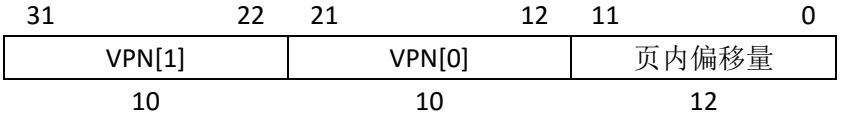


图 4.12：Sv32 虚拟地址

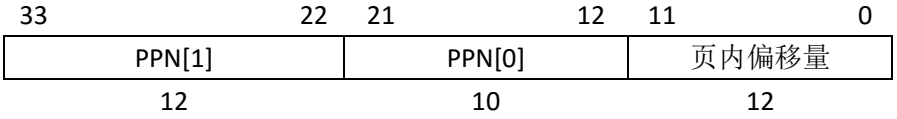


图 4.13：Sv32 物理地址

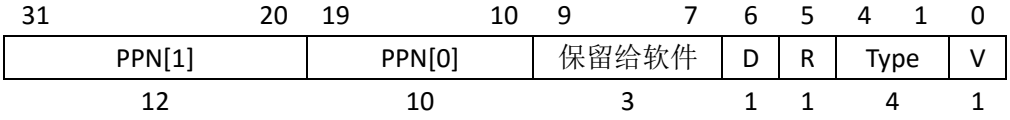


图 4.14：Sv32 页表项

Sv32 页表包含 2^{10} 个页表项（PTE），每个页表项为 4 个字节。一个页表刚好是一个页的大小，并且永远对齐到页的边界。根页表的物理地址被保存在 sptbr 寄存器中。

Sv32 的 PTE 格式如图 4.14 所示。V 位表明 PTE 是否是有效的；如果 V 是 0，则 PTE 的位 31-1 是无关项（don't care），可被软件自由使用。否则，Type 字段表明了 PTE 是一个指向下级页表的指针，还是一个叶子 PTE（leaf PTE）。如果是后者，Type 字段同时还编码了访问权限。表 4.2 给出了 Type 字段的编码。

可选的另外一种 PTE 格式独立地指明管理员和用户权限，这样解释较为简单，但是需要更多的位来编码。这样做将减少一个 32 位 PTE 能够寻址的物理空间大小。

管理员页映射可以在 Type 字段标明为全局的。全局的映射指的是那些存在于所有地址空间的映射。对于非叶子 PTE，全局设置隐含地说明其下级的所有页表映射都是全局的。注意如果没有将一个全局映射标记为全局的，只是降低了性能，然而如果把一个不是全局的映射标记为全局的，则是一个错误。

全局映射被设计用来减少上下文切换的开销。当一条 SFENCE.VM 指令被执行且 sasid 值为非零时，全局映射不需要被刷新（flush）出一个实现的地址翻译 Cache。

每个叶子 PTE 维护一个被引用（referenced，R）位和脏（dirty，D）位。当一个虚拟页被读、写或者取（fetched from）时，实现将设置对应 PTE 的 R 位为 1。当一个虚拟页被写时，实现将额外地设置对应 PTE 的 D 位为 1。导致 R 和/或 D 位被置位的访问操作，不能在更新 PTE 之前出现（The access that causes the R and/or D bit to be set must not appear to precede the update of the PTE.）。另外，PTE 的更新相对于其他写 PTE 的操作来说，必须是原子的。

管理员　　用户

Type	含义	Global	R	W	X	R	W	X
0	指向下一级页表的指针	●	——					
1	指向下一级页表的指针——全局映射							
2	管理员只读，用户读-执行页面		●			●		●
3	管理员读写，用户读-写-执行页面		●	●		●	●	●
4	管理员和用户只读页面		●			●		
5	管理员和用户读-写页面		●	●		●	●	
6	管理员和用户读-执行页面		●		●	●		●
7	管理员和用户读-写-执行页面		●	●	●	●	●	●
8	管理员只读页面		●					
9	管理员读-写页面		●	●				
10	管理员读-执行页面		●		●			
11	管理员读-写-执行页面		●	●	●			
12	管理员只读页面——全局映射	●	●					
13	管理员读-写页面——全局映射	●	●	●				
14	管理员读-执行页面——全局映射	●	●		●			
15	管理员读-写-执行页面——全局映射	●	●	●	●			

表 4.2：PTE Type 字段编码

R 位和 D 位永远不会被实现清零。如果管理员软件不依赖于被引用和/或脏位，例如，它不会将页面交换到二级存储，它应当总是将 PTE 的这两位设置为 1。于是实现就可以避免发出存储器访问以设置这两位。

任何级别的 PTE 都可能是一个叶子 PTE，因此除了 4KB 页面外，Sv32 支持 4MB 的*巨页（**megapage**）*。一个巨页必须虚拟的、物理的对齐到一个 4MB 边界。

4.5.2 虚拟地址翻译过程

一个虚拟地址 ***va*** 被翻译成一个物理地址 ***pa***，如下所示：

1. 设 sptbr 的值为 ***a***，设 ***i***＝LEVELS-1。（对于 Sv32，LEVELS 等于 2）
2. 设地址 ***a*＋*va.vpn[i]*×*PTESIZE*** 处 PTE 的值为 ***pte***。（对于 Sv32，PTESIZE 等于 4）
3. 如果 ***pte.v***＝0，停止并报告一个地址错误。
4. 否则，***pte.v***＝1。如果 ***pte.type***≥2，继续到步骤 5。否则，这个 PTE 是一个指向下级页表的指针。令 ***i***＝***i***－1。如果 ***i***＜0，停止并报告一个地址错误。否则，令 ***a***＝***pte.ppn***×***PAGESIZE***，转到步骤 2。（对于 Sv32，PAGESIZE 等于 2^{12}）
5. 找到一个叶子 PTE。通过 ***pte.type*** 判断所请求的存储器访问是否允许。如果不允许，停止并报告一个地址错误。否则，翻译成功。设 ***pte.r*** 等于 1，并且如果存储器访问是一个 store，设 ***pte.d*** 等于 1。翻译后的物理地址如下：

   pa.pgoff＝***va.pgoff***。

   如果 ***i***＞0，则这是一个超页翻译并且 ***pa.ppn[i－1:0]***＝***va.vpn[i－1:0]***。

   pa.ppn[LEVELS－1***:i]***＝***pte.ppn[***LEVELS－1***:i]***。

4.6 Sv39：基于页面的 39 位虚拟存储器系统

本节描述一个简单的分页虚拟存储器系统，它是为 RV64 所设计的，它支持 39 位虚拟地址空间。Sv39 的设计整体上按照 Sv32 的方案，本节仅描述两者之间不同的细节。

4.6.1 寻址和存储器保护

Sv39 实现支持一个 39 位虚拟地址空间，分割成 4KB 页面。一个 Sv39 虚拟地址被切分为如图 4.15 所示。Load 和 Store 的有效地址是 64 位的，必须使 63-39 位等于第 38 位，否则将会产生一个地址异常。27 位的 VPN 通过一个三级的页表被翻译为一个 38 位 PPN，而 12 位的页内偏移量不被翻译。

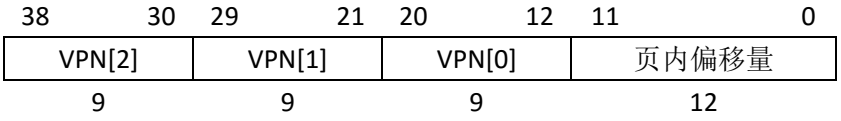


图 4.15：Sv39 虚拟地址

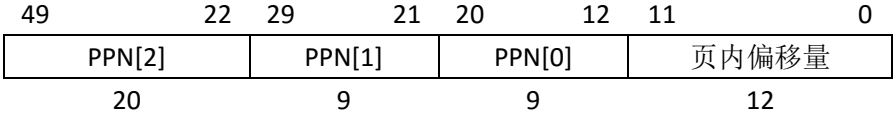


图 4.16：Sv39 物理地址

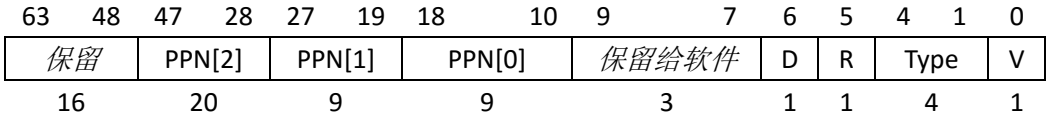


图 4.17：Sv39 页表项

Sv39 页表包含 2^9 个页表项（PTE），每个页表项为 8 个字节。一个页表刚好是一个页的大小，并且永远对齐到页的边界。根页表的物理地址被保存在 sptbr 寄存器中。

Sv39 的 PTE 格式如图 4.17 所示。位 9-0 具有 Sv32 一样的含义。位 63-48 被保留给未来使用，为了前向兼容性，软件必须将它们置为 0。

我们为一个可能的扩展保留了几个 PTE 的位，这个扩展通过允许跳过页表级别，提高对稀疏地址空间的支持，降低了存储器使用量和 TLF refill 延迟。这些保留位也可用于帮助研究实验。其代价就是减少了物理地址空间，但是 1PB 现在来说是足够了。如果到了不够用的时候，那些未被分配的保留位可用于扩展物理地址空间。

任何级别的 PTE 都可能是一个叶子 PTE，因此除了 4KB 页面外，Sv39 支持 4MB 的***巨页（megapage）***和 1GB 的***吉页（gigapage）***。两者都必须虚拟的、物理的对齐到它们各自大小的边界。

虚拟到物理地址的翻译算法与 4.5.2 节的一样，除了 LEVELS 等于 3 而 PTESIZE 等于 8。

4.7 Sv48：基于页面的 48 位虚拟存储器系统

本节描述一个简单的分页虚拟存储器系统，它是为 RV64 所设计的，它支持 48 位虚拟地址空间。Sv48 是为了那些 39 位虚拟地址空间不是足够的系统而设计的。它的设计与 Sv39 非常相似，只是简单的加入了额外一级页表，因此本节仅描述两者之间不同的细节。

支持 Sv48 的实现也应当支持 Sv39。

我们为 RV64 指定两种虚拟存储器系统是为了缓解提供大的地址空间和最小化地址翻译开销的矛盾。对于许多系统来说，512GB 虚拟地址空间是足够的了，因此 Sv39 便能满足。Sv48 将虚拟地址空间增大到 512TB，但也增大了页表占用的物理空间容量，增大了遍历页表的延迟，增大了保存虚拟地址的硬件结构。

已经支持 Sv48 的系统，可以基本上不耗代价的支持 Sv39，因此应当这么做（支持 Sv39）以支持那些假定了 Sv39 的管理员软件。

4.7.1 寻址和存储器保护

Sv48 实现支持一个 48 位虚拟地址空间，分割成 4KB 页面。一个 Sv48 虚拟地址被切分为如图 4.18 所示。Load 和 Store 的有效地址是 64 位的，必须使 63-48 位等于第 47 位，否则将会产生一个地址异常。36 位的 VPN 通过一个四级的页表被翻译为一个 38 位 PPN，而 12 位的页内偏移量不被翻译。

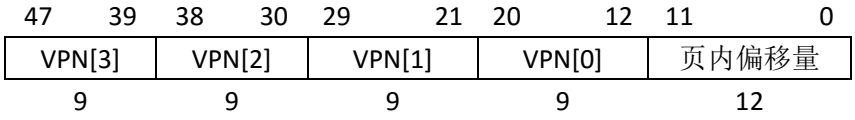


图 4.18：Sv48 虚拟地址

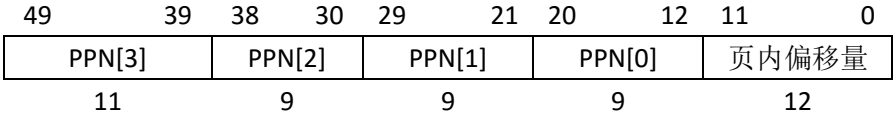


图 4.19：Sv48 物理地址

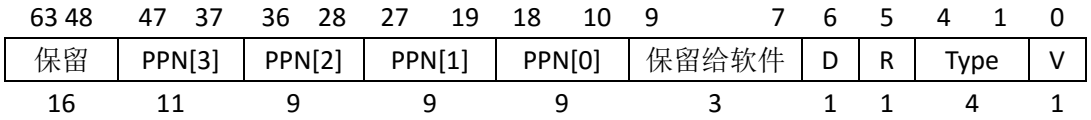


图 4.20：Sv48 页表项

Sv48 的 PTE 格式如图 4.20 所示。位 9-0 具有 Sv32 一样的含义。任何级别的 PTE 都可能是一个叶子 PTE，因此除了 4KB 页面外，Sv48 支持 4MB 的***巨页（megapage）***、1GB 的***吉页（gigapage）***和 512GB 的***特页（terapage）***。三者都必须虚拟的、物理的对齐到它们各自大小的边界。

虚拟到物理地址的翻译算法与 4.5.2 节的一样，除了 LEVELS 等于 4 而 PTESIZE 等于 8。

第5章 Hypervisor 级 ISA

本章是未来RISC-V hypervisor级公共核心规范的预留位置。

特权体系结构被设计成简化经典虚拟化技术的使用，此时一个 guest OS 运行于用户级，少数几条特权指令可以很容易被检测并自陷。

第6章 RISC-V 特权指令集列表

本章给出了 RISC-V 特权体系结构定义的所有指令的指令集列表。

31 20	19 15	14 12	11 7	6 0	
imm[11:0]	rs1	funct3	rd	opcode	I 类

访问 CSR 的指令

csr	rs1	001	rd	1110011	CSRRW rd,csr,rs1
csr	rs1	010	rd	1110011	CSRRS rd,csr,rs1
csr	rs1	011	rd	1110011	CSRRC rd,csr,rs1
csr	zimm	101	rd	1110011	CSRRWI rd,csr,rs1
csr	zimm	110	rd	1110011	CSRRSI rd,csr,rs1
csr	zimm	111	rd	1110011	CSRRCI rd,csr,rs1

改变特权级的指令

000000000000	00000	000	00000	1110011	ECALL
000000000001	00000	000	00000	1110011	EBREAK
000100000000	00000	000	00000	1110011	ERET

自陷重定向指令

001100000101	00000	000	00000	1110011	MRTS
001100000110	00000	000	00000	1110011	MRTH
001000000101	00000	000	00000	1110011	HRTS

中断管理指令

000100000010	00000	000	00000	1110011	WFI

存储器管理指令

000100000001	rs1	000	00000	1110011	SFENCE.VM rs1

表 6.1：RISC-V 特权指令

第7章 历史

致谢

感谢 Christopher Celio、David Chisnall、Palmer Dabbelt、Matt Thomas 和 Albert Ou 对特权规范的反馈。

7.1 资助

RISC-V体系结构和实现的研发，部分的由下列赞助商资助。

- **Par Lab**：研究受Microsoft (Award #024263)、Intel (Award #024894)资助，并由U.C. Discovery (Award #DIG07-10227)配套资助。其他支持来自Par Lab的伙伴Nokia、NVIDIA、Oracle和Samsung。
- **Project Isis**：DoE Award DE-SC0003624。
- **Silicon Photonics**：DARPA POEM program Award HR0011-11-C-0100。
- **ASPIRE Lab**：DARPA PERFECT program，Award HR0011-12-2-0016。The Center for Future Architectures Research (C-FAR)，Semiconductor Research Corporation资助的STARnet中心。其他支持来自ASPIRE产业赞助商、Intel和ASPIRE的伙伴Google、Nokia、NVIDIA、Oracle和Samsung。

本文的内容并不代表US政府的立场或者政策，并且没有暗示官方的认可。

参考文献

[1] Robert P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34-45, June 1974.

[2] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89-104, December 2002.

[3] Rusty Russell. Virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95-103, July 2008.