清 华 大 学

综 合 论 文 训 练

题目：RISC-V 用户态中断扩展设计与实现

系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：田凯夫

指导教师：陈　渝　副教授

2023 年 6 月 13 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签　名：田凯夫　导师签名：陈涛　日　期：2023年6月7日

中文摘要

进程间通信是进程间协调合作的基础，其效率往往影响着系统整体的工作效率。尤其在微内核架构中，进程间通信是主要的性能瓶颈之一。传统的进程间通信机制如信号、管道等大多数需要内核的参与，不可避免地引入了切换的开销。共享内存机制虽然减少了切换的开销，但同时也引入了同步互斥的开销，且通信双方无法实时响应消息，仍需要轮询或其他机制辅助。

用户态中断是近年来提出的进程间通信机制，由用户态直接收发中断信号，可以减少陷入内核的开销，而且独立的中断处理流程可以支持异步通信，确保实时性的同时减少等待的开销。本文基于 RISC-V 指令集架构提出了用户态中断扩展方案。不同于 x86 指令集架构的用户态中断扩展，本文的设计方案与 RISC-V 标准指令集结合紧密，并通过用户态中断控制器进一步减小了指令的访存开销。在硬件方面，修改 QEMU 模拟器验证设计方案并在 Rocket Chip 上实现该硬件扩展；在软件方面，修改 Linux 6.0 以适配硬件特性。最终在 FPGA 上搭建 SoC ，成功启动 Linux 并运行测试程序。实验表明，用户态中断相比于信号机制，可以取得 203 倍的性能提升；相比于 eventfd，可以取得 136 倍的性能提升。

关键词：用户态中断；软硬件结合；进程间通信

ABSTRACT

IPC (Inter-Process Communication) is the basis of inter-process coordination and co-operation, and its efficiency often affects the efficiency of the overall work of the system. Especially in microkernel, IPC is one of the main performance bottlenecks. Most of the traditional IPC mechanisms such as Signal and Pipe require the participation of the kernel, which inevitably introduces context switching overhead. Although the shared memory mechanism reduces the overhead of context switching, it also introduces the overhead of synchronous mutual exclusion, and the communication ends cannot respond to messages in real time, so polling or other mechanisms are still needed.

User-Interrupt is an IPC mechanism proposed in recent years. Programs in user mode can directly send and receive interrupt signals, which can reduce the overhead of trapping into the kernel, and the independent interrupt handler can support asynchronous communication, ensuring real-time performance and reducing waiting overhead. This paper proposes a User-Interrupt extension based on the RISC-V ISA. Unlike the User-Interrupt extension of the x86 ISA, the design in this article is closely integrated with the RISC-V standard ISA and further reduces the memory access overhead of instructions through a User-Interrupt Controller. In terms of hardware, the QEMU simulator is modified to verify the design and the hardware extension is implemented on Rocket Chip. In terms of software, Linux 6.0 is modified to adapt to hardware features. Finally we build SoC on FPGA with Rocket Chip and successfully boot Linux. Experiments show that compared with Signal, User-Interrupt can achieve 203x performance improvement; compared with eventfd, it can achieve 136x performance improvement.

Keywords: User-Interrupt; Hardware-Software Co-Design; IPC

目　　录

主要符号表

KPTI	内核页表隔离 (Kernel Pagetable Isolation)
IPC	进程间通信 (Inter-Process Communication)
UINTC	用户态中断控制器 (User-Interrupt Controller)
MMIO	内存映射的输入/输出 (Memory-Mapped Input/Output)
PLIC	平台级中断控制器 (Platform Level Interrupt Controller)
CLINT	核心本地中断器 (Core-Local Interruptor)
IPI	跨核中断 (Inter-Processor Interrupt)
UIPI	用户态跨核中断 (User Inter-Processor Interrupt)
RISC	精简指令集计算机 (Reduced Instruction Set Computer)
ISA	指令集架构 (Instruction Set Architecture)
CSR	控制/状态寄存器 (Control/Status Register)
FPGA	现场可编程逻辑门阵列 (Field Programmable Gate Array)
ILA	集成逻辑分析仪 (Integrated Logic Analyzer)

第 1 章　引言

1.1　研究背景

传统的 IPC（Inter-Process Communication，进程间通信）机制包括信号、管道、命名管道、消息队列和共享内存等[1]，其性能问题主要体现在以下几个方面：

1. 上下文切换开销：进程间通过内核进行通信，保存上下文和切换页表都会带来较大开销，为了解决熔断漏洞时引入的 KPTI 机制进一步增加了陷入内核的开销[2]；
2. 数据拷贝开销：将数据块从一个进程复制到另一个进程的地址空间中，引入较高的延迟和开销；
3. 同步互斥开销：使用锁和信号量等同步机制来保证进程之间访问共享资源的顺序，需要使用原子指令、内存屏障等硬件机制，会引入额外的开销；
4. 安全性问题：进程和进程之间、内核和进程之间资源的共享都有可能产生数据窃取和损坏等问题。

在微内核架构中，内核只提供最基本的服务，例如虚拟存储、IPC 等，大部分的系统服务如网络协议栈、文件系统、设备驱动等都以进程的形式运行在用户空间[3]，IPC 因此成为了微内核中最重要的部分之一，同时也是性能瓶颈之一。

用户态中断（User-Interrupt）是由用户接收和响应的中断，相比于传统的 IPC 机制，用户态直接对中断进行处理可以减少陷入内核带来的开销。无论是在宏内核还是微内核中都具有良好的应用场景。用户态中断的发送方可以是下列中的任何一种：

- 外部设备：用户态驱动[4]相比于内核的干预更加灵活和轻便，具有良好的可移植性和安全性。引入用户态中断后，用户态驱动可以直接接收并处理外部设备发来的中断，进一步提高性能。
- 内核：io_uring[5]是 Linux 5.1 版本引入的一种异步 I/O 机制，支持请求的批量提交、数据零拷贝，已被广泛应用于各种场景。引入用户态中断后，内核在 I/O 操作完成后可以跨核向进程发送用户态中断通知，进一步提高异步唤醒的效率。
- 进程：seL4 的 Notification[6]是一种轻量级的 IPC 机制，基于事件队列和快速路径，进程之间可以实现高效的异步通信，但目标进程需要通过轮询或等待

的方式来处理信息。引入用户态中断后，目标进程可以在执行其他任务的同时立即接收并处理信息。

1.2 相关工作

1.2.1 RISC-V N 扩展

RISC-V 是一种基于 RISC 原则的指令集架构，由加州大学伯克利分校开发，具有模块化、可扩展等特点。RISC-V N 扩展是在 RISC-V 特权级指令规范 v1.12 的草案中提出的，该扩展的设计思路是为 U 态提供一套和 M 态和 S 态类似的中断异常处理机制。截至目前，该扩展已被废除，因为 N 扩展的应用扩展尚不明朗且没有足够的工作去推动其完善和实现。为了引入用户态中断扩展，我们需要复现 RISC-V N 扩展的设计来支持基本的用户态中断处理流程。

名称	描述
`ustatus`	U 态全局中断使能
`utvec`	U 态陷入向量模式与基址
`uip`	U 态待处理中断（时钟中断、软件中断、外部中断）
`uie`	U 态使能中断（时钟中断、软件中断、外部中断）
`uscratch`	U 态暂存寄存器
`uepc`	U 态中断或异常指令 pc
`ucause`	U 态陷入原因
`sideleg`	S 态中断委托
`sedeleg`	S 态异常委托

表 1.1 RISC-V N CSRs

RISC-V N 扩展中的 `uret` 指令与 `mret` 和 `sret` 类似，将 `ustatus` 寄存器的 `UPIE` 赋值给 `UIE` ，然后跳转至 `uepc` 。

尤予阳等人对 RISC-V N 扩展进行了进一步完善，在 PLIC 中为 U 态额外分配一套上下文，并将 PLIC 中断信号与 CPU 的 `USIP` 寄存器连接。他们的工作主要分为两个方面，软件方面对 QEMU 进行修改来验证设计方案，基于 rCore 操作系统应用设计方案；硬件方面对 Rocket Chip 进行修改，并在 FPGA 上进行性能评估。通过将用户态中断应用在用户态串口驱动中，证明其在吞吐率和延时方面的性能

表现优于内核态驱动。

1.2.2 x86 用户态中断

2021 年 5 月发布的 Intel 指令集架构扩展[7] 中加入了基于 x86 指令集架构的用户态中断扩展。x86 的设计主要有以下几个特点：

1. 发送方和接收方状态在内存中进行维护：接收方通过 UPID 来维护使能位等信息，发送方通过 UITT 来维护 UPIDADDR 等信息；
2. SENDUIPI 指令需要经过两次读内存和一次写内存操作：当用户程序执行 SENDUIPI 时，硬件会首先根据 UITTSZ 寄存器检查传入的下标是否合法，然后根据 UITTADDR 从内存中读取对应表项，硬件再根据该表项包含的地址读取内存中的 UPID ，若 UPID 中使能位为 1 ，则向目标 APIC 发送中断信号，并将对应的 Pending 位写入到原来的 UPID 中；
3. 用户态中断的响应有一套独立的控制流程，需要通过几个 MSR 辅助完成。

intel 在 Linux 中加入了对 x86 用户态中断扩展的支持，并通过测试表明基于用户态中断的 IPC 机制相比于信号、eventfd 和管道等机制有明显的性能提升[8]。关于用户态中断可能的应用场景，intel 开发者指出他们正在研究已有的一些库例如 libevent 和 liburing，但目前尚未给出用户态中断的应用接口。项晨东等人在 QEMU 加入了对 x86 用户态中断扩展的支持，成功运行了上述 Linux 分支并复现了性能测试结果。

对比 RISC-V N 扩展和 x86 用户态中断扩展，可以发现以下几个问题：

1. x86 用户态中断扩展的中断处理流程与其他特权级的中断异常处理流程存在差异，而 RISC-V N 扩展与指令集结合的更为紧密，沿用了 M 态和 S 态的中断异常处理流程，相比之下，RISC-V N 扩展的设计更易于理解，且硬件逻辑更简单。
2. x86 用户态中断扩展并没有涉及用户态如何处理外部中断，而尤予阳等人的工作通过沿用 PLIC 中 M 态和 S 态的设计思路，引入额外的 U 态上下文，在未改变原有 RISC-V N 扩展的基础上，加入了用户态外部中断的处理。
3. x86 用户态中断扩展的核心指令是 SENDUIPI，这条指令的逻辑涉及三次访存，设计比较臃肿，虽然为用户程序提供了充分的灵活性，但同时也增加了硬件的复杂性。

1.2.3 Chisel 硬件描述语言

Chisel[9] 是基于 Scala 编程语言的高级硬件描述语言，设计者可以使用 Chisel 编写复杂的、可参数化数字电路生成器，并生成可综合的 Verilog 硬件描述语言。Chisel 既支持高度的抽象化表示，可以使用库函数和 Scala 提供的闭包函数等，同时又保留了对电路细粒度的控制。

Rocket Chip 是基于 Chisel 语言的 SoC 生成器，由加州伯克利分校的计算机科学和人工智能实验室开发。Rocket Chip 最大的特点是高度可配置化，支持生成处理器、内存控制器、外设等其他 SoC 组件。Rocket 处理器基于 RISC-V 指令集架构开发，并支持多种指令集扩展，例如基础的整数指令集、乘除扩展、浮点扩展等。

RoCC (Rocket Custom Coprocessor)[10] 是 Rocket Chip 的扩展内容，允许用户添加自定义的协处理器到 Rocket 处理器中来实现加速或其他特殊功能。RoCC 协处理器通过预定义的接口与 Rocket 处理器进行通信。RoCC 接口定义了一组标准的指令和协议，用于在 Rocket 处理器和 RoCC 协处理器之间传递数据和控制信息，它允许 RoCC 协处理器通过 Rocket 处理器的缓存系统访问内存和外设。

1.3 本文工作

本文的主要工作是设计并验证了 RISC-V 用户态中断扩展。具体地，本文通过软硬件结合的方法，自底向上对 RISC-V 用户态中断进行实现和性能评估，并在指令级别验证了用户态中断相对于传统的跨核通信机制的性能优势。

本文主要分为以下几个部分：

第 2 章介绍 RISC-V 用户态中断扩展的硬件设计方案，包括设计目标、CPU 状态维护、用户态中断控制器、UIPI 指令以及软件接口和工作流程。

第 3 章详细描述了硬件的架构和实现细节，主要包括两个层面：（1）在 QEMU 中加入对 RISC-V 用户态中断的支持，实现功能级的硬件模拟仿真；（2）在 Rocket Chip 中加入对 RISC-V 用户态中断的支持，提供可以在 FPGA 上运行的硬件原型。

第 4 章详细描述了软件的适配和实现细节，主要包括两个层面：（1）在 Linux 6.0 中加入对硬件实现的适配和系统调用接口；（2）添加库函数对系统调用进行封装，为用户态程序提供接口。

第 5 章介绍了在 FPGA 开发板上搭建 SoC 和实验环境进行功能和性能测试，并对性能测试的结果进行详细分析。

第 6 章对本文工作进行总结，并介绍未来可能的研究方向。

第 2 章　设计方案

在 RISC-V N 扩展的基础上，我们提出了 RISC-V 用户态中断扩展，通过引入新的 CSR、指令以及外部中断控制器，可以实现高效的用户态跨核中断。

2.1　整体架构

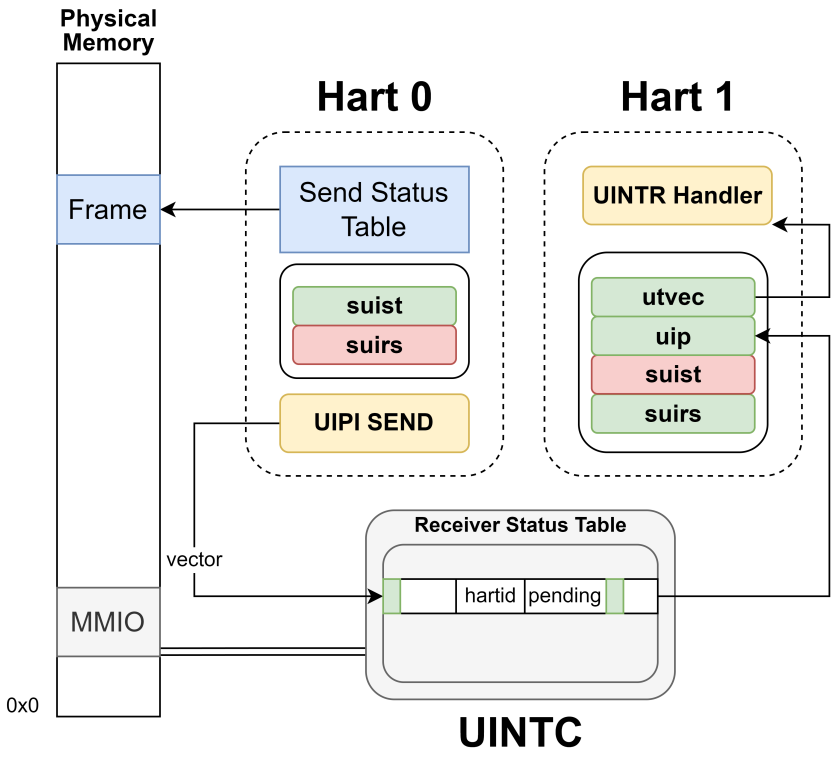


图 2.1　RISC-V 用户态中断扩展整体架构

图 2.1 给出了 RISC-V 用户态中断扩展的整体架构，图中的两个 Hart 上分别运行用户态中断的发送方和接收方，Hart 中给出了一些主要的控制寄存器，其中既包括 RISC-V N 扩展中采用的 `utvec` 和 `uip` 寄存器，也包括本章设计方案引入的 `suirs` 和 `suist` 寄存器（绿色代表使能，红色代表禁用）。此外，用户态中断控制器的 MMIO 访问端口通过地址翻译映射到内核地址空间，发送方的状态表也由内核进行管理。

2.2 CPU 状态

`suirs`(User-Interrupt Receiver Status) 寄存器和 `suist`(User-Interrupt Sender Table) 寄存器分别用来索引接收方和发送方的状态。这两个寄存器均被设置为 U 态不可访问，U 态只能通过 `uipi` 指令间接地应用它们包含的信息。若无特殊说明，以下的描述均基于 **64** 位 RISC-V 指令架构。

对应位	名称	描述
15:0	UIRS Index	接收方序号
62:16	Reserved	保留位，硬件会忽略这些位
63	Enable	使能位，置 1 表示使能

表 2.1　接收方状态寄存器

对应位	名称	描述
43:0	PPN	发送方状态表基址页号
55:44	Size	发送方状态表页面数量
62:56	Reserved	保留位，硬件会忽略这些位
63	Enable	使能位，置 1 表示使能

表 2.2　发送方状态寄存器

图 2.1 中 Hart 0 运行发送方，`suist` 寄存器被使能；Hart 1 运行接收方，`suirs` 寄存器被使能。

对应位	名称	描述
0	Valid	有效位，置 1 表示有效
15:1	Reserved	保留位，硬件会忽略这些位
31:16	Sender Vector	中断向量
47:32	Reserved	保留位，硬件会忽略这些位
63:48	UIRS Index	接收方序号

表 2.3　发送方状态

发送方状态在内存中进行维护，表项内容如表 2.3 所示。

2.3 用户态中断控制器

用户态中断控制器（UINTC，User-Interrupt Controller）作为设计的核心部分，主要负责维护接收方的状态信息，并响应来自读写端口的请求完成对应的操作。

对应位	名称	描述
0	Active	活跃位，置 1 表示可以向目标核发送中断
1	Mode	默认置 1，置 1 表示 64 位架构，置 0 表示 32 位架构
15:2	Reserved	保留位，硬件会忽略这些位
31:16	Hartid	正在运行该接受方的核号
63:32	Reserved	保留位，硬件会忽略这些位
127:64	Pending Requests	每一位对应一个中断向量，置 1 表示接收到中断请求

表 2.4　接收方状态

UINTC 为每一个接收方分配 32 B 的读写端口，每个操作都有可能从端口读出或向端口写入 8 B 数据，因此总共对应 8 种不同的操作，下表为不同操作对应的地址偏移量：

偏移量	读操作	写操作
0x00	Reserved	SEND
0x08	READ_LOW	WRITE_LOW
0x10	READ_HIGH	WRITE_HIGH
0x18	GET_ACT	SET_ACT

表 2.5　UINTC 操作码

其中 LOW 对应接收方状态的低 64 位，包括 Active，Mode，Hartid 等信息；HIGH 对应接收方状态的高 64 位，也就是 Pending Requests 。

SEND 操作会将数据中包含的中断向量写入到对应接收方状态的 Pending Requests 中，当 Active 为 1 且 Pending Requests 不为 0 时，UINTC 会拉高对应核的 `USIP` 位。

READ_HIGH 操作在读取 Pending Requests 后会将其清 0，而 **WRITE_HIGH** 操作则是将新的数据和原来的 Pending Requests 按位或，这样做是确保读写操作之间的中断请求不会被覆盖。

SET_ACT 操作会默认将新的数据的最低位写入到 Active 中。

CPU 通过执行 `sd` 或 `ld` 指令向总线发送读写请求，读写地址会被转化为不同的接收方序号，以支持 512 个接收方的 UINTC 为例，地址映射如下表所示：

偏移量	位宽	属性	名称	描述
0x00000000	32 B	RW	UIRS0	0 号接收方
0x00000020	32 B	RW	UIRS1	1 号接收方
...	...	...	...	...
0x00003FE0	32 B	RW	UIRS511	511 号接收方

表 2.6　UINTC 地址映射

2.4　UIPI 指令

`uipi` 是可以在 U 态直接执行的 R 型指令，共包括五条不同功能的指令：

0x0 `uipi.send` **rs1**：发送方发送用户态中断

0x1 `uipi.read` **rd**：接收方读取并清空中断等待位

0x2 `uipi.write` **rs1**：接收方写入中断等待位

0x3 `uipi.activate`：接收方准备接收用户态中断

0x4 `uipi.deactivate`：接收方拒绝接收用户态中断

这些指令执行到最后都需要读或写 UINTC 的端口，对 UINTC 中状态的影响与直接访问物理地址读写的影响是一致的。由于指令执行需要直接排除缓存系统访问外设，程序需要考虑指令乱序的问题。

`uipi.send` 指令传入发送方状态表的序号，根据 `suist` 寄存器中发送方状态表基址来读取内存中对应的表项，发送方在执行 `uipi.send` 指令后读到的物理地址为：

$$(\mathrm{PPN} << 0\mathrm{xC}) + (\mathrm{rs1} << 0\mathrm{x3})$$

其中页面大小默认为 4 KB，发送方状态表项的大小默认为 8 字节。若最后计算的地址超出了状态表的最大容量，该指令执行失败。若当前 `suist` 寄存器中使能位为 0，则该指令执行失败。硬件通过读出发送方指定的表项来获取中断向量和接收方序号，并写入 UINTC 对应的地址完成一次中断的发送。

其他的四条指令都需要根据 `suirs` 寄存器中接收方序号来获取 UINTC 读写端

口的物理地址。若当前 `suirs` 寄存器中使能位为 0，则该指令执行失败。

- `uipi.read` 指令直接访问 UINTC HIGH 端口读取数据
- `uipi.write` 指令直接访问 UINTC HIGH 端口写入数据
- `uipi.activate` 指令直接访问 UINTC ACT 端口并向 Active 位写入 1
- `uipi.deactivate` 指令直接访问 UINTC ACT 端口并向 Active 位写入 0

2.5 软件接口

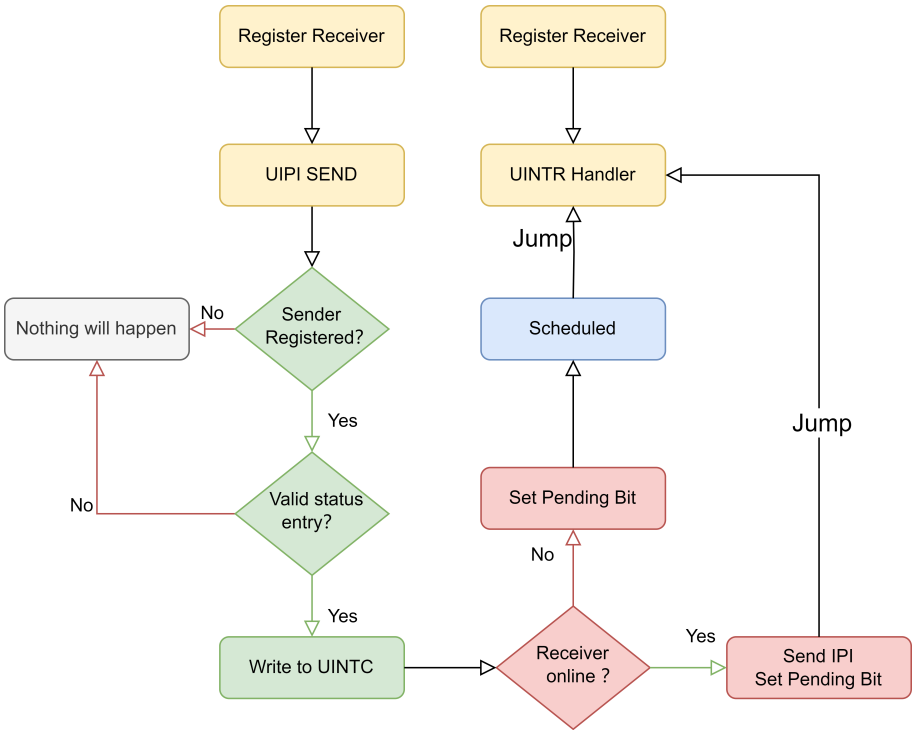


图 2.2 RISC-V 用户态中断工作流程

参考 x86 的用户态中断设计规范[8]，我们使用了类似的系统调用接口：

- `void uintr_register_handler(void *handler)`：接收方注册用户态中断处理函数
- `int uintr_create_fd(int vector)`：接收方分配中断向量，注册并返回文件描述符
- `int uintr_register_sender(int uintr_fd)`：发送方根据文件描述符注册发送方状态表项，返回发送方状态序号

如图 2.2 所示，应用上述系统调用接口，RISC-V 用户态中断的工作流程描述如下：

1. 接收方注册用户态中断处理函数和文件描述符，内核为接收方 UINTC 槽位及对应序号，处理函数的入口地址会被写入到 `utvec` 寄存器中；
2. 发送方根据文件描述符注册发送方状态表项，内核为发送方在内存中分配发送方状态表，中断向量和接收方序号会被写入到发送方状态表项中；
3. 发送方执行 `uipi.send` 指令发送用户态中断，硬件会根据传入的发送方状态序号查找内存中对应的发送方状态表项，并通过 UINTC 写端口发送 SEND 请求，UINTC 在 Pending Requests 中记录中断向量，并根据 Active 位判断是否向对应核发送中断信号：

4a. 若接收方处于正在运行状态，此时 UINTC 中 Active 位是 1，UINTC 向该核发送中断信号，接收方立刻陷入到中断处理函数中，完成后通过 `uret` 指令回到正常的执行流；

4b. 若接收方处于暂停状态，此时 UINTC 中 Active 位是 0，UINTC 不会发送中断信号，接收方被唤醒并准备返回 U 态运行前，内核会对接收方状态进行恢复并将 Active 位置 1，执行 `sret` 指令后会立刻陷入用户态中断处理函数中，完成后通过 `uret` 回到陷入到内核前的执行流；

5. 发送方可以通过和接收方协商来确定何时发送下一个中断，且发送方可以重复执行 `uipi.send` 指令发送中断。

2.6 本章小结

本章从整体架构出发，介绍了 RISC-V 用户态中断扩展的设计方案。通过添加外部中断控制器对接收方状态进行管理减少了 `uipi.send` 指令的访存开销。通过引入 `uipi` 指令和 CPU 控制寄存器，可以对用户态的行为加以限制，因为在普通的中断处理流程中，默认只有 M 态和 S 态可以接收或发送中断，且运行在这些特权态下的软件是可以信任的，但用户态程序的行为并不一定是合法的。我们的设计解决了如下几个问题，这些问题都有可能导致某个核正常执行流程被非法的中断打断：

- 发送方尝试向未注册的目标核发送中断：UINTC 会对 SEND 请求进行判断，如果目标核上未注册接收方，则不会向目标核发送中断。
- 接收方尝试修改自己的控制信息，将来自发送方的中断重定向到其他核：接收方只能通过 `uipi.read` 和 `uipi.write` 指令读写 Pending Requests，当前核号信息对接收方不可见。

- 接收方没有在目标核上运行，但发送方发送了用户态中断：UINTC 会对 SEND 请求进行判断，如果目标核处于非活跃状态，则不会向目标核发送中断。

此外，本章还介绍了系统调用的设计方案，并基于这些软件接口给出了软硬件协同工作的流程。

第 3 章　硬件实现

3.1　QEMU 模拟实现

QEMU[11] 为操作系统和用户态程序提供虚拟的执行环境，通过动态的二进制转换，模拟 CPU 的行为，同时支持多种外设的仿真，在系统开发中扮演着重要角色。QEMU 支持模拟 RISC-V 运行环境，通过对 QEMU 的修改和测试，我们可以不断完善设计方案。对 QEMU 的修改主要分为四个方面：

- 指令翻译：引入对 `uipi` 指令的译码和执行；
- CPU 状态：维护 CSR 寄存器等 CPU 状态；
- 内存读写：`uipi` 指令需要直接访问物理内存和 UINTC 外设，调用 `void cpu_physical_memory_rw(hwaddr addr, void *buf, hwaddr len, bool is_write)` 函数完成对物理地址的读写；
- 核间中断：实现 UINTC 并向各个核发送中断。

3.1.1　指令翻译

QEMU 翻译一条指令的过程为：从客户机指令（Guest Instructions）到中间码（TCG，Tiny Code Generator），最后再到宿主机指令（Host Instructions）。QEMU 的翻译机制类似于 CPU 流水线中的译码阶段，需要定义模式串来帮助 QEMU 在执行到某一指令时调用对应的辅助函数。模式串的定义位于 target/riscv/insn32.decode：

```
uipi_send        0000000 00000 ..... 010 ..... 1111011 @r2
uipi_read        0000001 00000 ..... 010 ..... 1111011 @r2
uipi_write       0000010 00000 ..... 010 ..... 1111011 @r2
uipi_activate    0000011 00000 ..... 010 ..... 1111011 @r2
uipi_deactivate  0000100 00000 ..... 010 ..... 1111011 @r2
```

以 `uret` 这条指令为例，在 target/riscv/insn_trans 目录下，有各种指令的翻译过程，主要用来将指令解析的结果（寄存器，立即数等）传递给辅助函数，将客户机指令拆解为宿主机指令来模拟目标指令的功能。对于 `uret` 指令的执行涉及到较多 CPU 状态的变化，会对 pc，CSR 等产生影响，辅助函数的定义位于 target/riscv/helper.h，通过宏定义 DEF_HELPER_x 来声明辅助函数，例如：

```
1 DEF_HELPER_1(uret, tl, env)
2 DEF_HELPER_4(csrrw, tl, env, int, tl, tl)
```

其中第一个参数对应辅助函数的名称，第二个参数代表函数的返回值类型（tl 表示 target_ulong），后面的参数都是辅助函数传入的参数类型。有了以上的参考，我们可以定义其他辅助函数：

```
DEF_HELPER_2(uipi_write, void, env, tl)
void helper_uipi_write(CPURISCVState *env, target_ulong src) {
    if (uipi_enabled(env, env->suirs)) {
        uint64_t addr = UINTC_REG_HIGH(env->suicfg, SUIRS_INDEX(env->suirs));
        cpu_physical_memory_write(addr, &src, 8);
    }
}
```

3.1.2 CPU 状态维护

CPU 状态的维护位于 target/riscv/cpu.h。这个结构同时考虑了 RV32、RV64、RV128 的情况，这些寄存器都是 CPU 运行时必要的状态。包括但不限于：

- pc
- 整数、浮点寄存器堆
- CSR，有些寄存器是 M 态和 S 态复用的，例如 `mstatus` 、`mip` 等
- PMP 寄存器堆
- 通过 kernel_addr、fdt_addr 等从指定位置加载镜像

在 target/riscv/cpu.h 文件末尾的表中注册 CSR 的操作函数。

中断异常、CSR 等宏定义位于 target/riscv/cpu_bits.h ，我们需要在其中添加和 U 态有关的中断控制位。CPU 中断异常处理函数位于 target/riscv/cpu_helper.c 的最后，这个函数对中断异常原因进行判断，并根据 CPU 当前的特权级做不同的处理。这个函数只给出了 M 态和 S 态的中断异常处理，我们需要额外在此处加入委托给 U 态的中断异常处理，也就是读写 `ustatus` ， `ucause` ， `uepc` 等寄存器。

3.1.3 跨核中断实现

QEMU 支持对不同硬件环境的模拟，需要在 virt 硬件环境中添加 UINTC 外设的配置并生成设备树信息。

UINTC 代码实现位于 hw/intc/riscv_uintc.c ，调用 riscv_uintc_realize 对 UINTC 进行初始化，将 UINTC 外设连接到总线上，并初始化总线地址空间。对外设中的状态寄存器（接收方状态寄存器，中断信号寄存器等）进行内存分配和初始化。通过调用 qdev_connect_gpio_out 默认将 UINTC 的中断信号绑定至每个核的 `uip` 寄存器中的 `USIP` 位。

```
for (i = 0; i < num_harts; i++) {
    CPUState *cpu = qemu_get_cpu(hartid_base + i);
```

```
    RISCVCPU *rvcpu = RISCV_CPU(cpu);
    qdev_connect_gpio_out(dev, i, qdev_get_gpio_in(DEVICE(rvcpu), IRQ_U_SOFT));
}
```

最后完成对 UINTC 读写函数的注册，这样就可以直接通过物理地址访问 UINTC 外设的读写端口：

```
static const MemoryRegionOps riscv_uintc_ops = {
    .read = riscv_uintc_read,
    .write = riscv_uintc_write,
    .endianness = DEVICE_LITTLE_ENDIAN,
    .valid = {
        .min_access_size = 8,
        .max_access_size = 8
    }
};
```

在 QEMU 的 virt 硬件环境中添加设备树生成代码，生成的设备树节点内容如下：

```
uintc@2f10000 {
    interrupts-extended = <0x08 0x00 0x06 0x00 0x04 0x00 0x02 0x00>;
    reg = <0x00 0x2f10000 0x00 0x4000>;
    interrupt-controller;
    compatible = "riscv,uintc0";
};
```

其中各个字段的含义为：

- **interrupts-extended** 连接到每个核 `uip` 寄存器的 `USIP` 位；
- **interrupts-controller** 表示该设备是一个接收中断的控制器，这里加上是为了方便 Linux 识别，实际上 UINTC 并没有接收外部中断；
- **compatible** 表示设备名称，Linux 内注册驱动时应该与之对应。

在 UINTC 的实现中，中断是通过每次写入 UINTC 的端口来触发的，这和真实的硬件实现其实存在差异。例如在 U 态，从目前的设计方案来看，需要同时满足以下几个条件才可以触发中断：

- 当前特权级为 S 态；
- `ustatus` 中 `UIE` 位是 1；
- `uie` 中 `USIE` 位是 1；
- `uip` 中 `USIP` 位是 1。

在硬件实现中，可以看成是几个信号的与操作，当其他所有信号都拉高时，任何一个信号从低电平拉高都会触发中断，根据 RISC-V 的特权态规范[12]，`sret` 会将特权态从 S 态切换回 U 态，`uret` 会将 `ustatus` 中的 `UIE` 位设置为 `UPIE` 位，在这两条指令后执行的第一条指令都有可能被中断打断并立刻进入中断处理的流程，

因此我们需要在 QEMU 中模拟这个过程，在 `sret` 和 `uret` 指令中直接对上述条件进行判断和处理，例如在 `sret` 的辅助函数中：

```
if (riscv_has_ext(env, RVN)
    && prev_priv == PRV_U
    && get_field(env->mip, MIP_USIP)
    && get_field(env->mstatus, MSTATUS_UIE)
    && get_field(env->sideleg, MIP_USIP)) {
    retpc = env->utvec;     // 直接跳转到U态中断处理入口
    env->uepc = env->sepc;  // 指定 \Iuret 到同一条指令
    mstatus = env->mstatus;
    mstatus = set_field(mstatus, MSTATUS_UPIE, 1);
    mstatus = set_field(mstatus, MSTATUS_UIE, 0);
    env->mstatus = mstatus;
}
```

3.2 Rocket Chip 硬件实现

3.2.1 CPU 状态维护

参考设计方案，在 rocket/CSR.scala 中添加读写 CSR 的逻辑，需要注意 `uip` 等多个特权级共用的寄存器需要设置对应特权级的屏蔽位。

参考 M 态中断和异常委托的逻辑，添加 S 态委托到 U 态的逻辑，此外需要在 S 态将已委托给 U 态的中断屏蔽。

在 rocket/IDecode.scala 中添加 `uret` 的指令解码并将指令解码注册到 `decode_table` 中。`uret` 的功能逻辑比较简单，只需要设置 `ustatus` 中的使能位并重新设置 pc 即可。

```
    // 读取 uip 寄存器
    val read_uip = read_mip & read_sideleg
    // 委托给 U 态的中断
    val delegateU = Bool(usingUser) && reg_mstatus.prv === PRV.U && delegate && read_sideleg(
    cause_lsbs) && cause(xLen - 1)
    // U 态的中断
    val u_interrupts = Mux(nmie && reg_mstatus.prv === PRV.U && reg_mstatus.uie,
    pending_interrupts & read_sideleg, UInt(0))
    // uret 的逻辑
    when (Bool(usingUser) && !io.rw.addr(9) && !io.rw.addr(8)) {
      reg_mstatus.uie := reg_mstatus.upie
      reg_mstatus.upie := true
      ret_prv := PRV.U
      io.evec := readEPC(reg_uepc)
    }
```

3.2.2 UINTC 外设实现与接入

参考 CLINT 和 PLIC 实现 UINTC 外设，首先定义 device 并指定名称和 compatible，和 QEMU 中生成设备树的逻辑类似，在这里需要指定 UINTC 连接到 intc

，且需要指定为 interrupt-controller 让 linux 完成初始化。

```
    val device = new SimpleDevice("uintc", Seq("riscv,uintc0")) {
        override val alwaysExtended: Boolean = true
        override def describe(resources: ResourceBindings): Description = {
            val Description(name, mapping) = super.describe(resources)
            val extra = Map("interrupt-controller" -> Nil,
                "#interrupt-cells" -> Seq(ResourceInt(1)))
            Description(name, mapping ++ extra)
        }
    }
```

定义 node 并配置 UINTC 寄存器的读写端口，定义一系列的 `RegField` 实现读写操作，调用 `node.regmap(opRegFields)` 进行注册：

```
    val node: TLRegisterNode = TLRegisterNode(
        address = Seq(params.address),
        device = device,
        beatBytes = beatBytes,
        concurrency = 1)

    // 注册 SEND 操作的写端口
    val opRegFields = uirs.zipWithIndex.flatMap { case (x, i) =>
      Seq(sendOffset(i) -> Seq(RegField(64, (),
        RegWriteFn { (valid, data) =>
            x.pending := x.pending | (valid << data(5, 0)).asUInt
            Bool(true)
        }))}
```

定义 `intnode` 连接到 CPU，在上述 `SEND` 操作里将 `ipi` 寄存器对应位置位：

```
    val intnode: IntNexusNode = IntNexusNode(
        sourceFn = { _ => IntSourcePortParameters(Seq(IntSourceParameters(1,
            Seq(Resource(device, "int"))))) },
        sinkFn = { _ => IntSinkPortParameters(Seq(IntSinkParameters())) },
        outputRequiresInput = false)
    // 拉高 ipi 寄存器
    val ipi = Seq.fill(nHarts) { RegInit(0.U) }
    ipi.zipWithIndex.foreach { case (hart, i) =>
        hart := uirs.map(x => x.pending =/= 0.U && x.active && hartId(x.hartid) === i.asUInt).
     reduce(_ || _)
    }
    val (intnode_out, _) = intnode.out.unzip
    intnode_out.zipWithIndex.foreach { case (int, i) =>
        int(0) := ShiftRegister(ipi(i)(0), params.intStages) // usip
    }
```

将中断信号连接到对应核的 `USIP`，参考其他信号的处理方法，连接 `core.interrupts` 和 `intSinkNode`：

```
    // freechips.rocketchip.tile.SinksExternalInterrupts
    def csrIntMap: List[Int] = {
        // ...
        val usip = if (usingUser) Seq(0) else Nil
        List(65535, 3, 7, 11) ++ seip ++ usip ++ List.tabulate(nlips)(_ + 16)
    }
    // freechips.rocketchip.subsystem.CanAttachTile
```

```
    // 将 intSinkNode 的输入连接到 Rocket 处理器
    def decodeCoreInterrupts(core: TileInterrupts): Unit = {
        // ...
        val usip = if (core.usip.isDefined) Seq(core.usip.get) else Nil
        // ...
        val (interrupts, _) = intSinkNode.in(0)
        (async_ips ++ periph_ips ++ seip ++ usip ++ core_ips).zip(interrupts).foreach { case(c,
      i) => c := i }
    }
```

3.2.3 UIPI 协处理器实现

UIPI 协处理器负责处理自定义 `uipi` 指令，基于 RoCC 实现，应用 RoCC 的访存端口处理 `uipi` 指令涉及的访存请求。

RoCC 位于流水线的写回阶段，读写 CSR 时需要考虑写后读冲突，RoCC 读 `suirs` 和 `suist` 寄存器时需要增加前传逻辑。

```
    // 前传逻辑，以 suirs 寄存器为例
    when (decoded_addr(CSRs.suirs)) {
        val new_suirs = new SUIRS().fromBits(wdata)
        reg_suirs := new_suirs
        io.uintr.suirs := new_suirs
    }.otherwise {
        io.uintr.suirs := reg_suirs
    }
```

根据 Rocket Chip 的配置方法，在 Tile 中添加 UIPI 协处理器：

```
    class WithUIPI extends Config((_, _, _) => {
        case BuildRoCC => Seq((p: Parameters) => {
            // 指定该 RoCC 处理符合 OpcodeSet.custom3 格式的指令
            val module = LazyModule(new UIPI(OpcodeSet.custom3)(p))
            module
        })
    })
```

UIPI 协处理器接收译码结果并根据操作码来执行不同处理流程，状态机各个状态描述如下：

- **`s_idle`** 默认状态，等待接收处理器传来的操作请求；
- **`s_wait_mem0`** `uipi.send` 指令发起读内存请求并等待响应；
- **`s_read_uist`** `uipi.send` 指令处理响应并根据返回数据计算出访问 UINTC 的地址，根据数据缓存的 `s2_nack` 信号判断是否需要重新发起请求；
- **`s_wait_mem1`** 发起读或写 UINTC 请求并等待响应；
- **`s_check_nack0`** 等待数据缓存响应；
- **`s_check_nack1`** 根据数据缓存的 `s2_nack` 信号判断是否需要重新发起请求；
- **`s_resp`** 响应处理器请求，`uipi.read` 需要返回写入目标寄存器的内容；

- `s_error` 指令格式错误、访存错误、权限错误、发送方状态表项错误。

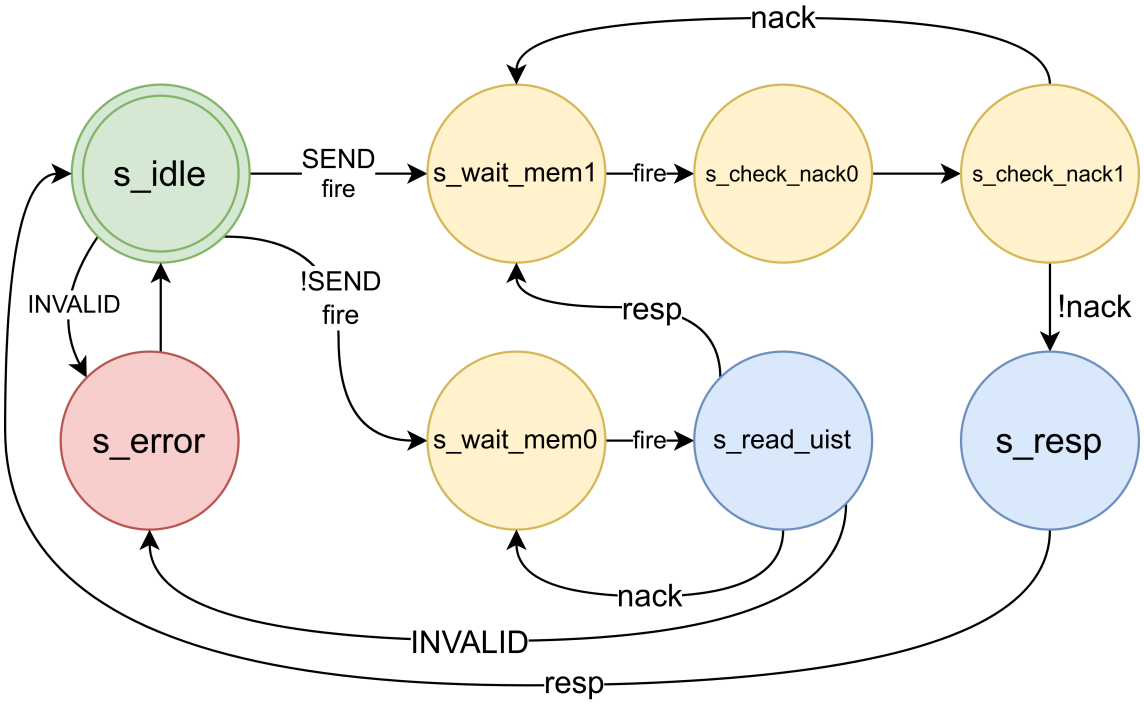


图 3.1　UIPI 协处理器工作流程

UIPI 协处理器共发出三类访存请求：读内存，读外设，写外设，其中读内存请求可能因为数据缓存不命中需要重新发起请求，读写外设请求可能因为数据缓存事务繁忙需要重新发起请求，因此在处理流程中加入了对数据缓存 `s2_nack` 信号的处理。此外，在 Rocket Chip 中存在一个缓存多个 RoCC 读写请求的队列模块，支持失败读写请求的重试，然而对于写外设请求，队列会持续等待响应信号并阻塞。因为目前只有一个 UIPI 协处理器，所以在架构中移除了这个模块并直接将 UIPI 协处理器连接到数据缓存的仲裁器。

在 UIPI 协处理器的实现中，由于涉及的状态较多且与处理器通信信号比较复杂，需要借助于行为仿真观察波形来判断当前逻辑是否正确，此外，由于 Rocket Chip 本身逻辑复杂，在代码层面很难直观地获取信息，同样需要借助于行为仿真来整理思路。关于 Verilator 行为仿真的具体内容参见第 5.2.1 节。

3.3　本章小结

本章分两个层面介绍了硬件实现的方法，无论是在 QEMU 中还是在 Rocket Chip 中，改动都主要涉及三个方面，这三个方面分别对应设计方案的三部分内容，

即 CPU 控制寄存器和状态维护，用户态中断控制器和 UIPI 指令。通过对 QEMU 的分析和改动，可以不断完善设计方案中的漏洞，最后将设计方案落实到真实的硬件中。Chisel 硬件描述语言强大的功能和丰富的抽象使得硬件逻辑更加清晰，帮助我们在开发过程中减少了使用传统硬件描述语言时遇到的各种错误。

第 4 章　软件适配

4.1　Linux 内核扩展

Linux[13] 是世界上最著名的开源操作系统。Linux 是标准的宏内核，可以在其上运行多种应用。目前 Linux 已经支持在 RISC-V 硬件平台上运行，也能在 QEMU 模拟的 RISC-V 硬件环境上运行。为了支持用户态中断并对其性能进行更深入的分析，我们对 Linux 6.0 版本进行了修改。针对 Linux 的修改主要分为三个部分：

- 添加 UINTC 驱动并让 Linux 识别 UINTC 的设备树信息；
- 接收方陷入内核的状态保存与恢复；
- 注册并实现系统调用。

4.1.1　UINTC 驱动支持

```
struct uintc_priv {
    struct cpumask lmask; // 位图记录已连接的 CPU
    void __iomem *regs;   // UINTC 在内核地址空间映射的起始地址
    resource_size_t size; // UINTC 地址范围大小
    u32 nr;               // UINTC 槽位数量
    void *mask;           // 位图记录已分配的槽位
    spinlock_t lock;      // 互斥锁
};
```

驱动代码位于 drivers/irqchip/irq-riscv-uintc.c ，Linux 对 UINTC 进行的初始化过程如下：

- 解析设备树获取外设对应的物理地址范围；
- 初始化全局控制结构 4.1.1 并保存外设信息；
- 调用 `void __iomem *ioremap(phys_addr_t addr, size_t size)` 完成内核物理地址到虚拟地址的映射，内核可以通过虚拟地址直接访问 UINTC 读写端口；
- 初始化全局位图管理 UINTC 中已分配的槽位。

4.1.2　状态保存与恢复

arch/riscv/kernel/entry.S 是 Linux 在 RISC-V 硬件平台上运行时的陷入入口，涉及上下文的保存与恢复、针对不同陷入原因跳转到对应的处理函数入口、针对不同系统调用号跳转到系统调用向量表对应的入口等。

考虑到在某些负载环境下，接收方进程可能在不同核上迁移，需要对用户态

的控制状态进行保存与恢复，主要对 utvec 、uscratch 、uepc 三个寄存器进行保存。此外还需要在陷入时将接收方状态的 Active 位置 0 确保当这个核运行其他进程时不会被中断。接收方被唤醒并准备在某个核上运行前，除了对上述三个寄存器进行恢复外，还需要设置 suirs 寄存器。

4.1.3 进程管理

Linux 中线程也被称为轻量级进程（LWP, Light-weight process）[14]，如无特殊说明，描述中默认使用进程指代进程或线程。用户进程通过系统调用注册成为发送方或接收方（可以同时是发送方和接收方），内核需要在进程控制块中维护发送方和接收方的状态。

接收方状态包括如下内容：

- **uirs_index** 对应 UINTC 的接收方状态表中的下标；
- **uvec_mask** 发送方中断向量位图，记录已分配的向量。

发送方状态包括如下内容：

- **uist_ctx** 发送方状态表状态，指向内存中的发送方状态表，此外还包括锁和引用计数等变量；
- **uist_mask** 发送方状态表位图，记录已分配的发送方状态表项。

在第二章第五节介绍的软件接口中，发送方和接收方通过文件描述符共享某些状态，包括如下内容：

- **recv** 指向创建该文件描述符的接收方状态；
- **uvec** 该文件描述符对应的中断向量。

Linux 在系统调用表中注册系统调用号和处理函数，并通过宏定义系统调用处理函数。

系统调用 `void uintr_register_handler(void *handler)` 由接收方发起，用于注册接收方中断处理函数。内核查询 UINTC 并分配空闲项，并在接收方状态中保存相关信息，若当前无空闲项，则返回错误码。此外，内核还需要将 UINTC 中的接收方状态表项清空，以免出现未知的错误。如前面小节所述，suirs 等寄存器的初始化均放在系统调用处理完毕和返回用户态前之间进行。

系统调用 `int uintr_create_fd(int vector)` 由接收方发起，用于注册某一中断向量对应的文件描述符。内核首先分配一个文件描述符，然后根据接收方状态中的 uvec_mask 分配传入的中断向量，若该向量已被分配，则返回错误码。执行成功后，系统调用返回新创建的文件描述符。

系统调用 `int uintr_register_sender(int uintr_fd)` 由发送方发起，用于发送方根据文件描述符注册状态表项。内核首先判断该发送方是否注册过状态表，如果尚未注册，则在内存中分配一定数量的页作为发送方状态表，通过 `uist_mask` 分配状态表中的一项，若当前无空闲项，则返回错误码。如表 2.3 所示，内核将文件描述符中包含的信息写入到状态表中。

为了防止内存泄漏，需要在进程退出时释放资源。Linux 提供了 `exit_thread` 接口，需要添加配置选项来定义这个函数（否则默认为空）。在这个函数中，需要判断当前进程是否为发送方或接收方，发送方需要释放状态表，接收方需要释放占用的 UINTC 表项。此外，文件描述符信息中包含了指针指向发送方状态，为了防止出现野指针，需要在 `file_operations` 中注册 `release` 函数清空指针。

4.2 库函数实现

为方便用户态程序使用系统调用接口，以库函数的形式对系统调用进一步封装，默认支持包括设置 U 态 CSR 、上下文保存与恢复以及读取并更新 UINTC 中的 Pending Requests 等功能：

```
extern void __handler_entry(struct __uintr_frame* frame, void* handler) {
    uint64_t irqs = uipi_read();
    csr_clear(CSR_UIP, MIE_USIE);
    uint64_t (*__handler)(struct __uintr_frame * frame, uint64_t) = handler;
    irqs = __handler(frame, irqs);
    uipi_write(irqs);
}
static uint64_t __register_receiver(void* handler) {
    // 设置中断处理函数入口
    csr_write(CSR_UTVEC, uintrvec);
    csr_write(CSR_USCRATCH, handler);
    // 使能 U 态中断处理
    csr_set(CSR_USTATUS, USTATUS_UIE);
    csr_set(CSR_UIE, MIE_USIE);
    int ret = __syscall0(__NR_uintr_register_receiver);
    // 使能 UINTC
    uipi_activate();
    return ret;
}
// 用户调用接口
#define uintr_register_receiver(handler) __register_receiver(handler)
```

注意到上述代码并没将用户注册的函数直接赋值给 `utvec` 寄存器，而是赋值给了 `uscratch` 。参考下面给出的汇编代码，需要先将通用寄存器保存在栈上，然后才能开始执行用户注册的函数。用户态中断处理函数 `__handler_entry` 执行完毕后跳转到 `jal` 的下一条指令也就是 `uintrret` 的入口，恢复原来执行流的通用寄

存器。

```
    .section .text
    .align 4
    .globl uintrvec
uintrvec:
    # 分配栈空间
    addi sp, sp, -248
    # 保存通用寄存器
    sd ra, 0(sp)
    # ...此处省略...
    # 跳转至用户态中断处理函数
    mv a0, sp
    csrr a1, uscratch
    jal __handler_entry

    .align 4
    .globl uintrret
uintrret:
    # 从栈上恢复通用寄存器
    ld ra, 0(sp)
    # ...此处省略...
    # 释放栈空间
    addi sp, sp, 248
    # 返回原来的执行流中
    uret
```

4.3 用户态程序实现

根据图 2.2 中 RISC-V 用户态中断的工作流程，可以实现一个简单的进程间通信程序。

```
    volatile unsigned int uintr_received; // volatile 避免编译优化
    unsigned int uintr_fd;

    uint64_t uintr_handler(struct __uintr_frame *ui_frame, uint64_t irqs) {
        uintr_received = 1; // 设置标志位
        return 0;
    }

    void *sender_thread(void *arg) {
        int uipi_index;
        // 注册发送方状态表项
        uipi_index = uintr_register_sender(uintr_fd);
        // 发送用户态中断
        uipi_send(uipi_index);
        return NULL;
    }

    int main() {
        pthread_t pt;
        int ret;
        // 注册接收方中断处理函数
        if (uintr_register_receiver(uintr_handler))
            exit(EXIT_FAILURE);
```

```
24          // 注册文件描述符
25          ret = uintr_create_fd(1);
26          if (ret < 0) exit(EXIT_FAILURE);
27          uintr_fd = ret;
28          // 创建发送方
29          if (pthread_create(&pt, NULL, &sender_thread, NULL))
30              exit(EXIT_FAILURE);
31          // 忙等待标志位
32          while (!uintr_received);
33          pthread_join(pt, NULL);
34          close(uintr_fd);
35          // 正常退出
36          exit(EXIT_SUCCESS);
37      }
```

在 `main` 函数中，首先由接收方注册中断处理函数，注册文件描述符，创建发送方线程，并忙等待标志位；线程创建默认地址空间共享，因此发送方线程可以根据文件描述符注册，发送一个中断便直接退出；接收方收到中断并陷入中断处理函数后设置标志位，回到正常流程发现标志位已被设置，继续执行直到退出。

4.4 本章小结

本章介绍了软件上各个层面针对硬件的适配。为了使测试结果更具有说服力，我们选择对 Linux 进行修改和扩展以适配硬件中添加的有关用户态中断的特性。为了使用户程序编写起来更加方便，我们编写了库函数对系统调用和汇编代码进行封装。为了更清晰地阐明第二章中软硬件协同工作的流程，我们编写了一个简单的用户态测试程序，尽可能覆盖各种特性的同时，验证软硬件协同工作的正确性。

第 5 章　系统评估

本章重点介绍在 FPGA 上对系统的构建和评估，实验的目标主要分为以下几个方面：

- 验证硬件实现的正确性；
- 验证软件实现的正确性；
- 验证软硬件协同设计的用户态中断的性能优势；
- 分析测试结果与硬件行为间的联系。

5.1　实验环境

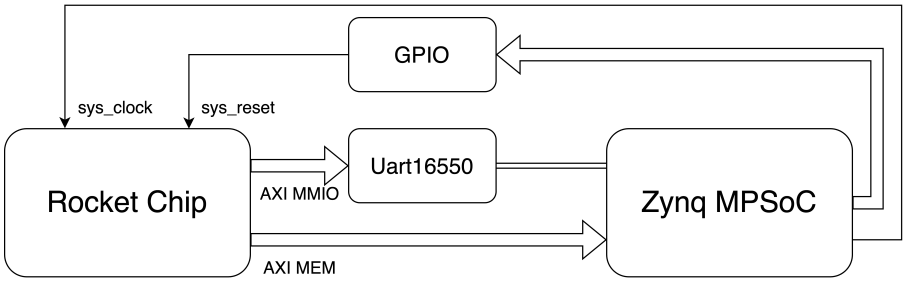


图 5.1　SoC 整体架构

本文实验主要在 Zynq UltraScale+ MPSoC ZCU102 开发板上开展。ZCU102 分为两个部分，分别为处理器子系统（PS）和可编程逻辑（PL）。PS 提供一款四核 ARM® Cortex®-A53、双核 Cortex-R5F 实时处理器，可以直接对开发板上的资源进行控制；PL 端可以通过 DDR4 组件访问内存资源。

首先，应用 Xilinx 提供的 petalinux SDK 构建在 PS 上运行的 Linux 系统，并挂载 SD 卡作为系统的存储外设。如图 5.1 所示，引入 AXI GPIO IP 核并在 PS 的设备树给出相关信息，GPIO 的出端口连接到 Rocket Chip 内部，可以在 PS 上操作 /sys/class/gpio/* 来写入这个端口对 Rocket Chip 进行复位。

利用 Rocket Chip 支持的自定义端口和自定义配置参数构建顶层模块。自定义配置包括加入第三章最后一节介绍的 UIPI 协处理器，设置 BootROM 的地址和加载内容。Rocket Chip 顶层模块需要对外暴露两个 AXI master 接口，MEM 端口访问位于 PS 端的 DDR 控制器，MMIO 端口访问 PL 上引入的 Uart16550 IP 核。关于两个端口的地址映射，需要将 Rocket Chip 配置的映射和 Block Design 构建时指定

的地址映射相对应。

进一步地，定义系统的顶层模块对 Rocket Chip 顶层模块和 Block Design 模块进一步封装，该模块会对 AXI 接口的位宽进行处理，此外，由于 Rocket Chip 默认看到的内存起始地址为 0x80000000 ，而 Block Deisgn 中 DDR 控制器访问端口的地址映射起始地址为 0 ，需要在模块中对地址高位进行截断。

为了与 Rocket Chip 上运行的系统软件进行交互，引入 Uart16550 IP 核，虽然 Rocket Chip 能够自动生成与内部配置有关的设备树信息，但缺少 PL 额外引入的设备的设备信息，需要手动加入 Uart16550 的设备树节点。

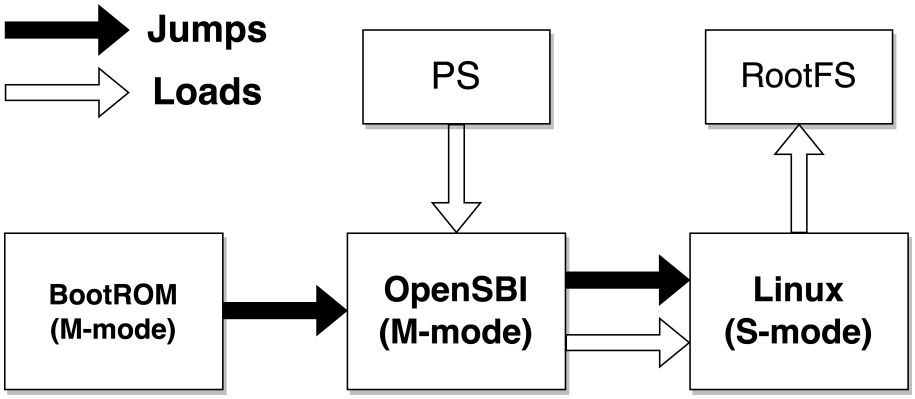


图 5.2　系统软件启动流程

系统软件的启动流程如图 5.2 所示。BootROM 为 Rocket Chip 内置的启动代码，在构建时会提前写入到模块中，跳转地址默认为内存的起始地址。

OpenSBI (RISC-V Open Source Supervisor Binary Interface)[15] 是 RISC-V 架构下的 Bootloader，支持通用 SBI 接口，支持 dynamic、jump、payload 三种启动方式。本文的实验采用 payload 启动方式，OpenSBI 镜像会根据 Linux 镜像和设备树的文件位置一起构建。OpenSBI 会在运行时将 Linux 镜像加载到目标地址，并将设备树放在指定的地址。跳转到 Linux 镜像前，OpenSBI 会将核号、设备树地址等信息传递给 Linux 用于 Linux 进一步的初始化。

Buildroot[16] 是一个构建嵌入式 Linux 系统的框架。本文的实验使用 Buildroot 构建根文件系统，并在 Linux 的编译选项中指定文件系统镜像的位置，让 Linux 镜像与文件系统镜像一起构建，并将该文件系统作为 initramfs 。此外，为了让 Linux 在启动时可以打印日志，需要在设备树中添加启动参数指定 earlycons 。

如前文所述，PS 可以对 Rocket Chip 进行复位，在复位前，需要把构建好的 OpenSBI 镜像拷贝到内存中，这部分内存以 reserved-memory 的形式挂载到 PS 的 /dev/mem 文件上，PS 可以直接对该文件进行 mmap 操作完成镜像的写入。

Linux 启动后，可以在主机上连接开发板的串口并看到输出：

图 5.3　启动 Linux

5.2 功能测试

5.2.1 Verilator 仿真测试

Verilator[17] 是一款开源的硬件描述语言仿真器，可以生成 C++ 仿真代码并在主机上模拟硬件电路的行为。相比于传统的基于事件的仿真器，Verilator 可以更快速、准确地模拟大型电路。Rocket Chip 集成了 Verilator 仿真器，可以在综合前对 Rocket Chip 进行行为仿真，通过观察 Verilator 生成的波形，可以提早发现并解决硬件逻辑问题，减少硬件调试带来的开销。

riscv-tests[18] 是一个开源的验证 RISC-V 处理器正确性的测试集，由 RISC-V 基金会开发和维护，包括了 RISC-V 指令集的大部分指令和特性的测试。Rocket Chip 集成该项目并利用 Verilator 对这些测例进行仿真。基于这一项目，我们实现了多个针对 RISC-V 用户态中断的测例，覆盖了读写 CSR 、中断处理、UIPI 指令等特性。

5.2.2 软件接口测试

```
[    7.313724] clocksource: Switched to clocksource riscv_clocksource
[   21.338406] workingset: timestamp_bits=62 max_order=16 bucket_order=0
[   28.154498] io scheduler mq-deadline registered
[   28.237624] io scheduler kyber registered
[   43.562542] Serial: 8250/16550 driver, 4 ports, IRQ sharing disabled
[   46.009692] printk: console [ttyS0] disabled
[   46.110334] 60000000.serial: ttyS0 at MMIO 0x60001000 (irq = 0, base_baud = 6250000) is a 16550A
[   46.274488] printk: console [ttyS0] enabled
[   46.274488] printk: console [ttyS0] enabled
[   46.426498] printk: bootconsole [ns16550a0] disabled
[   46.426498] printk: bootconsole [ns16550a0] disabled
[  251.130402] Freeing unused kernel image (initmem) memory: 8392K
[  251.294326] Run /init as init process
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Saving 256 bits of non-creditable seed for next boot
Starting network: ip: socket: Function not implemented
ip: socket: Function not implemented
FAIL

Welcome to Buildroot
buildroot login: root
login[69]: root login on 'ttyS0'
# cd /tests/ipc-bench/
# ./uintrfd/uipi-sample
Basic test: uipi_sample
[  728.402060] [CPU 1] uintr: [sys_uintr_register_receiver        ]: receiver=71 entry=0
[  728.730386] [CPU 1] uintr: [__do_sys_uintr_create_fd           ]: receiver=71 uvec=1 uintrfd=3
Receiver enabled interrupts
[  729.310398] [CPU 0] uintr: [__do_sys_uintr_register_sender     ]: sender=72 entry=0 va=ffffffd801211000
Sending IPI from sender thread 0
        -- User Interrupt handler --
        Pending User Interrupts: 2
[  730.505420] [CPU 1] uintr: [uintr_free                         ]: freed sender=72
[  731.286284] [CPU 0] uintr: [uintrfd_release                    ]: release uintrfd for uvec=1
Success
[  735.437400] [CPU 1] uintr: [uintr_free                         ]: freed receiver=71 entry=0
#
CTRL-A Z for help | 57600 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline | ttyUSB3
```

图 5.4　应用程序输出

在第四章最后一节中，我们介绍了一个简单的用户态进程间通信程序。分别在 QEMU 模拟器和 Rocket Chip 上启动修改后的 Linux 并运行该程序，可以看到

发送方和接收方均正常退出，并释放占用的内核资源，包括文件描述符、发送方状态表等。

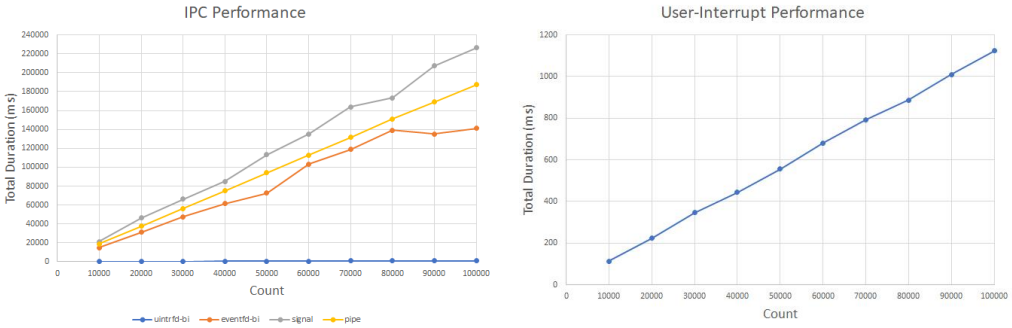
5.3 性能测试

ipc-bench[19] 是一个用来的测试 Linux 和 OS X 操作系统上 IPC 性能的开源项目，包括信号、管道、FIFO、共享内存、TCP 等 IPC 机制。为减少用户程序层面的干扰，对于不同的 IPC 机制采用相同的 Ping-Pong 通信模型，通信的双方会忙等待对方的消息。每个测例会根据配置参数决定某一方收或发的总次数和消息的大小，执行结束后在命令行打印计算得到的性能指标，包括通信的总时间，平均每次通信的时间，吞吐量，标准差。但是实际在统计测试结果时，我们并未完全采纳输出的信息，后续会给出具体的分析。

基于第四章第二节中的库函数，利用 ipc-bench 项目提供的环境，我们实现了针对用户态中断的测试程序。为满足 Ping-Pong 通信模型，该测试程序的通信两端需要同时作为接收方和发送方。在通信开始前，两个线程需要进行一系列的配置，这部分配置不会被包含在总通信时间中。正如上一节所介绍的，我们运行测试的硬件环境是 Rocket Chip，ipc-bench 项目在统计通信时间时，还统计了每次通信的时间来分析标准差等性能指标，需要在每次通信时调用 `timespec_get` 函数。这个函数会执行系统调用陷入内核，内核为了获取硬件寄存器中包含的时间信息，执行了 `rdtime` 这条伪指令，会进一步陷入 OpenSBI 中读取 CLINT 硬件寄存器，整个流程涉及大量的上下文切换，成为了实际的性能瓶颈。对于信号和管道等测例来说可能影响并不是很显著，但是对于 eventfd 和用户态中断等机制来说，每次通信都获取时间会降低测试的准确性。因此在后续的测试中，除用户态中断的测试程序以外，我们也修改了性能对比中用到的测试程序，包括信号、管道和 eventfd，具体的修改方案为移除每次通信的时间统计，只在通信循环的开始和结束获取时间并计算出通信的总时间。此外，在用户态中断中，消息的大小恒为 1 bit，因此在运行其他测例时，也指定消息大小为 1 bit。

5.4 测试结果与分析

总体上来看，用户态中断的性能明显优于其他 IPC 机制，说明用户态中断从发出到响应的延迟很低；具体到指令周期，通过 ILA 抓取 Rocket Chip 中的 `pc` 寄存器，从发送方所在的核执行 `uipi.send` 这条指令开始，到接收方所在的核触发



(a) IPC 性能对比　　(b) uintrfd-bi 性能测试结果

图 5.5　性能测试结果，总通信次数最大为 100000 次，步长为 10000 次

中断处理流程，`pc` 跳转到 `utvec` ，总共需要约 100 个时钟周期；根据第四章第二节中对于库函数的介绍，在执行用户指定的中断处理函数之前，需要进行通用寄存器的保存，大概需要约 400 个时钟周期。也就是说从发送方发送用户态中断到接收方响应并开始处理共需要约 500 个时钟周期，这已经远小于应用程序陷入内核再返回的开销了。此外，可以从性能测试结果计算出平均每次收发用户态中断需要的延迟为 11230 ns，也就是大约 560 个时钟周期（主频为 50 M，每个时钟周期 20 ns），符合我们通过 ILA 观察到的结果。

在实验的设计中，我们尽可能地减少其他因素对结果的影响，可以将通信总时间与通信次数近似地看成线性关系，从而可以计算出回归方程，其中斜率为单次通信的时间，因为实验开始时要获取时间，且缓存系统也需要预热，所以截距不为 0 。对比单次通信的时间，可以计算出用户态中断相对于其他 IPC 机制的延迟比：

IPC Type	Relative Latency (normalized to User-Interrupt)
User-Interrupt	1.00
Signal	203.84
Eventfd	136.69
Pipe	167.78

表 5.1　IPC 相对延迟比

相比于 x86 的测试结果[8] ，上述实验的结果提升了一个数量级，如前文所述，我们的实验移除了对单次通信时间的统计，因此性能优势更加明显，而 intel 的测试则保留了单次通信时间的统计。

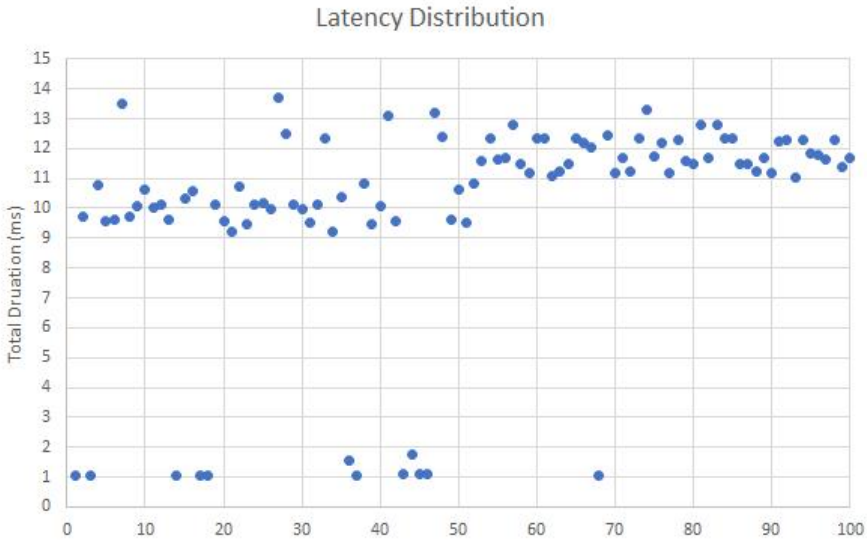


图 5.6　性能测试结果，运行 100 次 uintrfd-bi 测试，每次测试的通信次数为 100 次

相比于其他的 IPC 机制，用户态中断的性能存在一定的波动。在进行软件适配时，需要考虑到接收方是否正在某个核上运行，理想情况下是接收方一直在某个核上运行，用户态中断的延迟可以达到上述的指令周期数；当目标核正处于内核态或运行其他进程时，只有 UINTC 的 Pending Requests 位被写入，UINTC 不会触发用户态中断。当目标接收方被调度时，内核才会在返回用户态前将当前核号写入 UINTC 并将 Active 位置为 1，从而在执行 `sret` 返回用户态的第一条指令触发用户态中断并陷入到用户态中断处理函数中。接收方不在核上运行的流程可以类比用户为某个信号注册处理函数的情况，二者都是经历了**恢复-陷入-恢复**的过程，在这种情况下，用户态中断的响应延迟会增加，但一般来说会比信号的处理更快，因为信号处理需要两次切换特权级，而用户态中断只需要一次切换特权级，所以用户态中断的处理相比信号的处理对于页表和缓存系统来说更加友好。在图 5.6 中可以观察到测试结果的波动情况，根据前面的分析可以大致估计 100 次通信需要的总时间为 1 ms，然而大部分测试的结果为 10 ms 左右，这部分开销正是来自于时钟中断的陷入和恢复过程（首次陷入内核缓存缺失的开销较大），但是考虑到时钟中断引入的额外开销会均摊到每一次通信上，所以随着通信总次数的增加对总体性能的影响较小。

5.5 本章小结

本章展示了我们针对前面几章在硬件和软件上的实现进行的实验，实验结果表明我们设计的 RISC-V 用户态中断扩展可以在软硬件协同下正常工作，且相比于传统的 IPC 机制有极大的性能提升。此外，我们的实验基于 Rocket Chip 的最新发布版本，因此实验环境的构建也给其他想在 ZCU102 上移植 Rocket Chip 并启动 Linux 的开发者提供了非常有价值的参考。

第 6 章　结论

本文提出了 RISC-V 用户态中断扩展方案，并通过软硬件协同的方式，对设计方案进行了验证和测试。实验结果表明，在本文搭建的测试环境中，用户态中断相比于信号机制，可以取得 203 倍的性能提升；相比于 eventfd，可以取得 136 倍的性能提升。

本文的工作聚焦于用户态中断扩展方案的设计与实现，重点对用户态中断的性能进行评估。在未来的工作中，可以更多地探索用户态中断的异步机制，并通过更复杂的应用场景如微内核等来发挥用户态中断的优势。

插图索引

表格索引

参考文献

[1] Tanenbaum A S, Bos H. Modern operating systems[M]. Prentice Hall Press, 2015.

[2] Lipp M, Schwarz M, Gruss D, et al. Meltdown and spectre[J]. Linux Weekly News, 2018, 8 (2): 1-7.

[3] Liedtke J. Towards real microkernels[J]. Communications of the ACM, 1996, 39(9): 70-77.

[4] Dong Y, Xu M, Dai Y, et al. User-space device drivers: achievements and challenges[C]//2013 13th International Conference on Computational Science and Its Applications. IEEE, 2013: 112-121.

[5] Viro A, Rago E, Kogan P. io_uring: The linux aio replacement[C]//Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19). USENIX Association, 2019: 441-454.

[6] Klein G, Elphinstone K, Heiser G, et al. sel4: Formal verification of an os kernel[J]. ACM SIGOPS Operating Systems Review, 2009, 43(3): 207-220.

[7] Intel® architecture instruction set extensions and future features programming reference [M/OL]. Intel Corporation, 2021: 11-1. https://www.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html.

[8] Mehta S. x86 User Interrupts support[EB/OL]. (2021-09-13)[2021-09-13]. https://lwn.net/Articles/869140/.

[9] Bachrach J, Vo H, Richards B, et al. Chisel: constructing hardware in a Scala embedded language[C/OL]//Groeneveld P, Sciuto D, Hassoun S. The 49th Annual Design Automation Conference (DAC 2012). San Francisco, CA, USA: ACM, 2012: 1216-1225. http://dl.acm.org/citation.cfm?id=2228360.

[10] Liu X, Zhao Y, Asanović K, et al. Automatic code generation for rocket chip rocc accelerators [C]//2016 IEEE 34th International Conference on Computer Design (ICCD). IEEE, 2016: 43-50.

[11] Bellard F, the QEMU team. About QEMU[EB/OL]. 2023. https://www.qemu.org/docs/master/about/index.html.

[12] Waterman A, Lee Y, Avizienis R, et al. The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10[R/OL]. EECS Department, University of California, Berkeley, 2017. https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf.

[13] Linux Foundation. Linux kernel[EB/OL]. 1991–present. https://www.kernel.org/.

[14] Bovet D, Cesati M. Understanding the linux kernel 3rd.[M]. Oreilly & Associates Inc, 2005.

[15] SiFive. OpenSBI: An Open Source SBI Implementation for RISC-V[EB/OL]. 2023. https://github.com/riscv/opensbi.

[16] Buildroot contributors. Buildroot GitHub Repository[EB/OL]. 2023. https://github.com/buildroot/buildroot.

[17] Verilator contributors. Verilator GitHub Repository[EB/OL]. 2023. https://github.com/verilator/verilator.

[18] RISC-V International. riscv-tests[EB/OL]. 2023. https://github.com/riscv/riscv-tests.

[19] Yang K. ipc-bench[EB/OL]. 2021. https://github.com/kclyu/ipc-bench.

致　　谢

大半年的工作告一段落，论文也接近完成，我想对所有帮助过我的人表示感谢。

首先，我想感谢陈渝老师和向勇老师。老师们用专业知识为我指明了研究方向，引领我一步步深入探索，在我遇到困难时给我支持和帮助，在我研究的过程中提出了很多宝贵的建议。

其次，我想感谢我的父母和家人们。一路走来，他们一直是我坚强的后盾和精神支柱，是我克服困难，克服骄躁的动力源泉。

此外，我还要感谢实验室的尤予阳学长，学长丰富的经验和灵感帮助我解决了很多开发和调试过程中遇到的困难。

最后，我想感谢我的室友们。大学四年，和室友们一起熬夜奋战，一起肆意快活，一起畅想未来的时光给我留下了美好的回忆。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：田凯夫　日　期：2023年6月7日.

附录 A　外文资料的书面翻译

RISC-V 高级中断架构（第一章至第三章第六节）

目录

A.1　介绍

本文档指定了 RISC-V 高级中断架构，包括：(a) 对 RISC-V 标准特权架构的扩展（RISC-V 指令集手册第二卷中指定的 Harts）;(b) 用于 RISC-V 系统的两个

标准中断控制器: 高级平台级中断控制器 (APLIC, Advanced Platform-Level Interrupt Controller) 和消息驱动中断控制器 (IMSIC, Incoming Message-Signaled Interrupt Controller); 和 (c) 与中断有关的其他系统部件的要求。

本章内容主要是关于我们的设计决策，实现选项和应用场景的说明，读者如果只对规范感兴趣，可以跳过这章。

A.1.1 目标

RISC-V 高级中断架构有以下目标：

- 以 RISC-V 的中断处理功能为基础构建特权架构，最大限度地减少对现有功能的替换。
- 除了基本的有线中断外，为 RISC-V 系统提供直接使用 PCI Express 和其他设备标准所采用的消息驱动中断 (MSIs, message-signaled interrupt) 的设施。
- 对于有线中断，定义一个新的平台级中断控制器 (高级 PLIC 或 APLIC)，为每个特权级 (RISC-VM 和 S 特权级) 提供独立的控制界面，并且可以将有线中断转换为系统支持的消息驱动中断。
- 在 RISC-V Hart 上扩展本地中断的框架。
- 可选地允许软件将所有中断源的相对优先级配置到 RISC-V Hart (包括标准的时钟中断和软件中断等)，而不是受限于由一个单独的中断控制器处理外部中断的优先级。
- 当 Hart 实现特权架构的虚拟化扩展时，为虚拟机中断虚拟化提供足够的支持。
- 借助用于重定向 MSI 的 IOMMU (I/O 内存管理单元)，最大限度地为客户操作系统提供对设备直接进行控制的能力，同时最小化虚拟机监管程序的参与。
- 避免虚拟机数量受到中断硬件的限制。
- 尽可能地权衡速度、效率和灵活性并实现上述所有功能。

高级中断架构的最初版本主要针对更大型、高性能的 RISCV 系统的需求。目前还没有定义对下列以下中断处理特性的支持，这些特性可以在所谓的“实时“系统中减小中断响应时间，但不太适合高速处理器：

- 给每个中断源一个单独的陷入入口地址;
- 在陷入时自动保存和恢复级寄存器；
- 基于优先级的抢占式级联中断。

这些特性的目标是优化更小的或实时系统，可以作为后续的指令扩展，或单独作为高级中断架构的未来版本。

A.1.2 限制

在当前版本中，RISC-V 高级中断架构可以支持高达 16384 个 Hart 的 RISC-V SMP 系统。如果 Hart 是 64 位（RV64）并实现虚拟化扩展，并且实现高级中断架构的所有功能，那么对于每个物理 Hart 可能有多达 63 个活跃虚拟 Hart 和潜在的数千个额外的空闲（被换出）虚拟 Hart，其中每个虚拟 Hart 直接控制一个或多个物理设备。

表 A.1 总结了 Hart 数量的主要限制，包括物理和虚拟的 Hart，以及高级中断架构可能支持的不同中断标识的数量。

我们假设任何具有数千个物理 *hart* 的单个 *RISC-V* 计算机（或集群或分布式系统中的任何单个节点）需要一个适应机器特定组织的中断设施，对此我们不尝试预测。

	Maximum	Requirements
Physical harts	16,384	
Active virtual harts having direct control of a device, per physical hart	31 for RV32, 63 for RV64	RISC-V hypervisor extension; IMSICs with guest interrupt files; and an IOMMU
Idle (swapped-out) virtual harts having direct control of a device, per physical hart	potentially thousands	An IOMMU with support for memory-resident interrupt files
Wired interrupts at a single APLIC	1023	
Distinct identities usable for MSIs at each hart (physical or virtual)	2047	IMSICs

表 A.1　系统中 Hart 数量和中断标识数量的绝对限制。独立的实现可能有更小的限制。

A.1.3 主要组件概述

一个 RISC-V 系统的中断信号总体架构取决于它是主要用于 MSI 还是更传统的有线中断。在完全支持 MSI 的系统中，每个 Hart 都有一个 IMSIC，它作为这个 hart 的私有中断控制器，负责外部中断。相反，在主要基于传统有线中断的系统中，Hart 没有 IMSIC。较大的系统，特别是带有 PCI 设备的系统，需要给 Hart 提供 IMSIC 来完全支持 MSI，而许多较小的系统可能仍然最适合使用有线中断和没有 IMSIC 的更简单的 Hart。

A.1.3.1 无 IMSIC 的外部中断

当 RISC-V Hart 没有 IMSIC 时，外部中断通过专用线路发送到 Hart。在这种情况下，一个 APLIC 充当传统的中央中断集线器，为每个 Hart 路由和按优先级处理外部中断，如图 A.1 所示。中断可以有选择地路由到每个 Hart 的 M 特权级或 S 特权级。APLIC 在第四章中进行了规定。

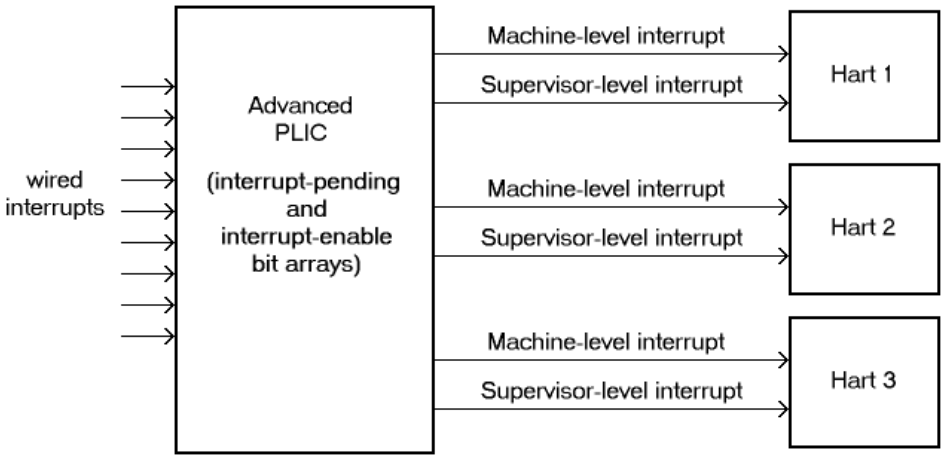


图 A.1　传统的有线中断发送方法，用于不支持 MSI 的 Hart

如果没有 IMSIC，当前的高级中断架构不支持直接将外部中断信号发送到虚拟机，即使 RISC-V Hart 实现了特权架构的虚拟化扩展也是如此。相反，必须将中断发送到相关的虚拟机监管程序，然后该虚拟机监管程序可以选择向虚拟机注入虚拟中断。

如果 *Hart* 实现了虚拟化扩展，目前正在研究的一个问题是，是否应该允许 *APLIC* 将外部中断路由为虚拟化扩展的客户机外部中断，从而允许直接将中断发送到虚拟机，而无需在虚拟机监管程序级别处理每个中断。目前，我们假设需要直接向虚拟机发出外部中断信号的系统具有 *IMSIC*。

A.1.3.2 有 IMSIC 的外部中断

为了能够接收消息驱动中断，每个 RISC-V Hart 必须具有一个 IMSIC ，如图 A.2 所示。基本上，消息驱动中断只是对硬件特定地址进行的内存写操作。为此，每个 IMSIC 在机器的地址空间中分配一个或多个不同的地址，当以预期的格式对其中一个地址进行写入时，对应的 IMSIC 将写入解释为相应 Hart 的外部中断。

由于所有 IMSIC 在机器的物理地址空间中具有唯一的地址，因此每个 IMSIC 可以从任何具有写入权限的代理 (Hart 或设备) 接收 MSI 写入。IMSIC 为针对 M 特

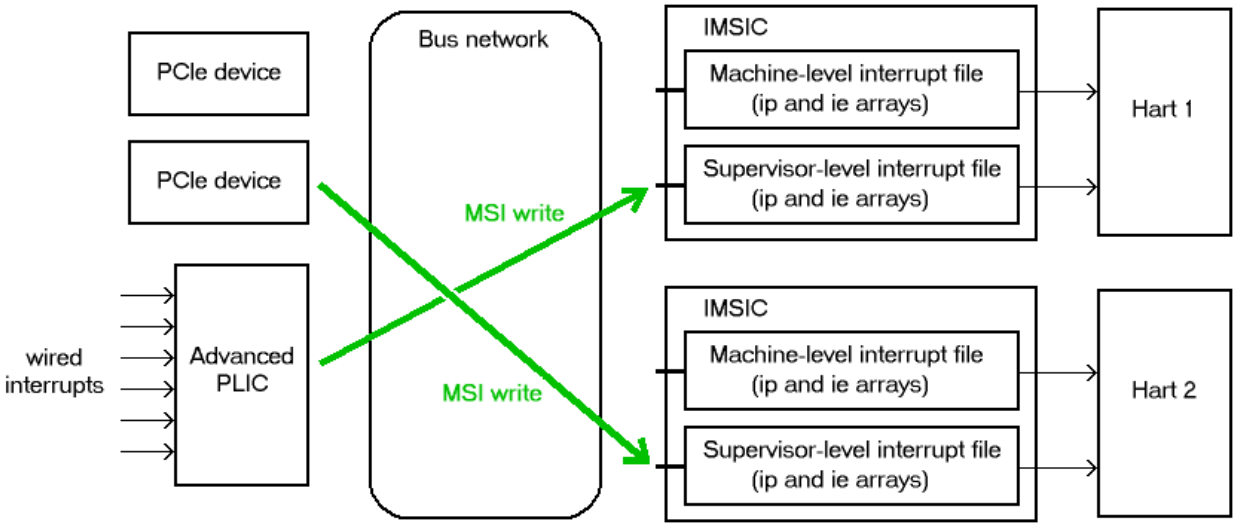


图 A.2　当 harts 具有 IMSIC 以接收 MSI 时，使用 MSI 进行中断发送。

权级和 S 特权级的 MSI 分别提供地址，一方面是为了通过控制不同地址的写入权限来单独授予或拒绝每个特权级上发送中断的能力，一方面是为了更好地支持虚拟化 (假设某特权级是更高特权级)。针对特定特权级的 MSI 被记录在 IMSIC 的中断文件 (interrupt file) 中，该文件主要由表示中断挂起位 (interrupt-pending bits) 的数组和表示该 Hart 当前准备接收哪个中断的中断使能位数组组成。

IMSIC 单元在第三章中完全定义。RISC-V 高级中断架构使用的 MSI 格式在该章节的第二节中进行了描述。

当 RISC-V 系统中的 Hart 具有 IMSIC 时，系统通常仍包含一个 APLIC，但其角色已改变。与图 A.1 中直接通过线路向 Hart 发送中断信号不同，APLIC 将传入的有线中断转换为 MSI 写入，并通过它们的 IMSIC 单元将其发送到 Hart。每个 MSI 根据软件设置的 APLIC 配置被发送到单个目标 Hart。

如果 RISC-V Hart 实现了特权架构的虚拟化扩展，IMSICs 可能具有额外的客户机中断文件，用于向虚拟机传递中断。除了 IMSIC 的第三章，还可以查看专门介绍向虚拟机发送中断的第六章。如果系统还包含用于执行 I/O 设备内存访问的地址翻译的 IOMMU，则来自这些设备的 MSI 可能需要特殊处理。这个问题在第 8 章中得到了解决，即“用于向虚拟机传递 MSI 的 IOMMU 支持”。

A.1.3.3　其他中断

除了来自 I/O 设备的外部中断外，RISC-V 特权架构还指定了一些其他主要类别的中断用于 Hart。特权架构的时钟中断仍然得到充分支持，软件中断得到至少

部分支持，尽管它们都没有出现在图 A.2 和 A.1 中。有关软件中断的具体信息，请参阅第七章“处理器间中断 (IPI, Inter-Processor Interrupts)”。

高级中断架构在 Hart 上添加了相当多的本地中断支持，即 Hart 在响应异步事件时本质上会自我中断，这些异步事件通常是错误。本地中断仍然包含在 Hart 内（或接近它），因此与标准的 RISC-V 时钟中断和软件中断一样，它们不会通过 APLIC 或 IMSIC。

A.1.4 硬件中断标识

RISC-V 特权架构为 Hart 的每个中断原因分配了一个独特的主要标识号 (major identity number)，它是在中断陷入时自动写入 CSR `mcause` 或 `scause` 的异常代码。由特权架构标准化的中断原因的主要标识在 0-15 范围内，而编号 16 及以上则正式供平台标准或自定义使用。高级中断架构声称进一步控制范围在 16-23 和 32-47 的标识号，留下 24-31 范围内的编号和所有 48 及以上的主要标识号自定义使用。表 A.2 描述了所有主要中断标识与此扩展的特征。

大多数 I/O 设备的中断通过 Hart 的外部中断控制器传递到 Hart，它可以是 Hart 的 IMSIC（图 A.2）或 APLIC（图 A.1）。如表 A.2 所示，给定特权级别的外部中断都共享一个单独的主要标识号：M 特权级为 11，S 特权级为 9，VS 特权级为 10。来自不同原因的外部中断通过外部中断控制器提供的次要标识号在 Hart 上互相区分。

除了外部中断之外，其他中断原因也可能有自己的次要标识号。然而，本文档只需要讨论外部中断的次要标识号。

高级中断架构定义的本地中断及其处理主要在第五章中讨论。

A.1.5 选择接收中断的 Hart

每个中断只会被传递给一个特权级别下的一个 Hart，通常由软件以某种方式确定。与其他某些架构不同，RISC-V 高级中断架构没有提供用于将中断广播或组播到多个 Hart 的标准硬件机制。

对于本地中断以及任何软件向 Hart 的较低特权级别注入的“虚拟”中断，这些中断完全是 Hart 本地的事务，其他 Hart 永远不可见。RISC-V 特权架构的时钟中断也与各自的 Hart 唯一绑定。

对于由 Hart 外部源接收的其他中断，软件会将每个中断信号（无论是通过线路还是通过 MSI 传递）配置为仅发送到单个 Hart。

要向多个 Hart 发送处理器间中断（IPI），发起 Hart 只需要执行一个循环，向

Major identity	Minor identity	
0	–	*Reserved by Privileged Architecture*
1	–	Supervisor software interrupt
2	–	Virtual supervisor software interrupt
3	–	Machine software interrupt
4	–	*Reserved by Privileged Architecture*
5	–	Supervisor timer interrupt
6	–	Virtual supervisor timer interrupt
7	–	Machine timer interrupt
8	–	*Reserved by Privileged Architecture*
9	Determined by	Supervisor external interrupt
10	external interrupt	Virtual supervisor external interrupt
11	controller	Machine external interrupt
12	–	Supervisor guest external interrupt
13	–	Counter overflow interrupt
14–15	–	*Reserved by Privileged Architecture*
16–23	–	*Reserved for standard local interrupts*
24–31	–	*Designated for custom use*
32–34	–	*Reserved for standard local interrupts*
35	–	Low-priority RAS event interrupt
36–42	–	*Reserved for standard local interrupts*
43	–	High-priority RAS event interrupt
44–47	–	*Reserved for standard local interrupts*
⩾ 48	–	*Designated for custom use*

表 A.2　每个 hart 的所有中断原因的主要和次要标识。主要标识号 0-15 属于 RISC-V 特权架构的标准。

每个目标 Hart 发送单个 IPI 即可。对于单个目标 Hart 的 IPI，请参见第七章。

发送单个 *IPI* 到多个目的地时，源 *Hart* 所花费的工作量肯定会被接收 *Hart* 处理这些中断所花费的总工作量所压倒。因此，最好只期望提供自动化的 *IPI* 组播机制能在最好的情况下略微减少系统的总体工作量。对于非常大量的 *Hart*，*IPI* 组播的硬件机制必须解决软件如何在每次使用时指定预期目标集合的问题，而且实际的 *IPI* 物理传递与软件版本可能没有太大区别。

我们不排除未来有可选的 *IPI* 组播硬件机制的可能性，但仅当在实际使用中

可以证明具有显著优势时才会考虑。截至 *2020* 年，在拥有 *IPI* 组播硬件的系统上观察到 *Linux* 并没有使用该硬件机制。

在极少数情况下，需要将来自 I/O 设备的单个中断发送给多个 Hart，这个中断必须发送到一个单独的 Hart，然后该 Hart 可以通过 IPI 向其他 Hart 发出中断。

我们认为，*I/O* 中断通知多个 *Hart* 的情况非常罕见，因此在这种情况下不能证明标准化硬件支持组播的必要性。

除了组播传递之外，其他架构还支持“*1-of-N*”中断传递选项，其中硬件从配置的 *N* 个 *Hart* 中选择一个单一的目标 *Hart*，旨在在 *Hart* 之间自动平衡中断处理的负载。*2010* 年代的实验对 *1-of-N* 模式的效用提出了质疑，在实践中显示出软件通常可以比实际芯片中实现的硬件算法更好地平衡负载。因此，*Linux* 被修改为停止在拥有 *1-of-N* 中断传递的系统上使用该功能。

我们仍然认为，硬件负载平衡中断处理可能对某些专业市场（如网络）有益，但是迄今为止在这方面所做的声明不能证明要求所有 *RISC-V* 服务器支持 *1-of-N* 传递。有了更多的证据，某些 *1-of-N* 传递机制可能成为未来的选择。

最初的 *RISC-V* 平台级中断控制器（*PLIC*）是可配置的，因此每个中断源可以向任何 *Hart* 子集（可能是所有 *Hart*）发出外部中断信号。当多个 *Hart* 从 *PLIC* 收到单个原因的外部中断时，首先在 *PLIC* 上“声明”中断的 *Hart* 将负责对其进行服务。通常，这会产生竞争，其中配置为接收多播中断的 *Hart* 子集同时接收外部中断并竞争成为在 *PLIC* 上首先声明中断的 *Hart*。意图是提供一种 *1-of-N* 中断传递形式。但是，对于所有未能获得声明的 *Hart*，其中断陷入的开销都会被浪费。

出于已经给出的原因，*APLIC* 支持将每个中断仅发送到由软件选择的单个 *Hart*，而不是多个 *Hart*。

A.1.6 ISA 扩展 Smaia 和 Ssaia

高级中断架构（AIA）为 RISC-V 指令集架构（ISA）的扩展定义了两个名称，一个用于 M 特权级，另一个用于 S 特权级。对于 M 特权级，扩展名 **Smaia** 涵盖了 AIA 为 Hart 指定的所有特权级别上所有添加的 CSR 和所有修改的中断响应行为。对于 S 特权级，扩展名 **Ssaia** 与 Smaia 基本相同，但排除了 M 特权级的 CSR 和 S 特权级看不到的行为。

扩展名 Smaia 和 Ssaia 仅涵盖对 Hart 的 ISA 产生影响的那些 AIA 特性。虽然下文将 APLIC、IOMMU 和除写入 IMSIC 之外的任何启动处理器间中断的机制描述或讨论为 AIA 的一部分，但它们不被 Smaia 或 Ssaia 提及，因为这些组件被归类为非 ISA 的组件。

正如后续章节所展示的，AIA 具体添加的 CSR 和行为，即 Smaia 或 Ssaia 所提及的，取决于基础 ISA 的 XLEN（RV32 或 RV64），是否实现了 S-mode 和虚拟化扩展，以及 Hart 是否具有 IMSIC。但并没有为每个可能的有效子集提供单独的 AIA 扩展名。相反，不同的组合可以从指示的特性（例如 RV64I+S-mode+Smaia，但不包括虚拟化扩展）的交集中推断出来。

像编译器和汇编器这样的软件开发工具不需要关心 IMSIC 是否存在，而是应该只允许尝试访问 IMSIC CSR（在第二章和第三章中描述），如果指示了 Smaia 或 Ssaia。如果没有实际的 IMSIC，这样的尝试可能会触发异常并陷入，但对于开发工具来说，这不是一个问题。

A.2 Hart 中添加的控制状态寄存器（CSR）

对于 RISC-V Hart 可以接收中断并陷入的每个特权级别，高级中断架构都添加了用于中断控制和处理的 CSR。

A.2.1 M 特权级 CSR

表 A.3 列出了在 M 特权级添加的 CSR 以及大小被高级中断架构更改的现有的 M 特权级 CSR。现有的 CSR `mie`、`mip` 和 `mideleg` 被扩展为 64 位以支持总共 64 个中断原因。

对于 RV32，表中列出的高半部 CSR 允许访问寄存器 `mideleg`、`mie`、`mvien`、`mvip` 和 `mip` 的高 32 位。高级中断架构要求这些高半部 CSR 存在于 RV32 中，但它们访问的位可能都是只读的零。

CSR `miselect` 和 `mireg` 提供了访问表 A.3中以外的多个寄存器的窗口。`miselect` 的值确定当前可以通过别名 CSR `mireg` 访问哪个寄存器。`miselect` 是一个 **WARL** 寄存器，它必须支持一定范围的最小值，这取决于实现的特性。当没有实现 IMSIC 时，`miselect` 必须能够至少容纳 0 到 `0x3F` 范围内的任何 6 位值。当实现了 IMSIC 时，`miselect` 必须能够容纳 0 到 `0xFF` 范围内的任何 8 位值。目前，范围在 0 到 `0xFF` 之间的 `miselect` 值被分配为以下子范围：

`0x00`–`0x2F`	reserved
`0x30`–`0x3F`	major interrupt priorities
`0x40`–`0x6F`	reserved
`0x70`–`0xFF`	external interrupts (only with an IMSIC)

`miselect` 也可能支持范围在 `0x00`–`0xFF` 之外的值，尽管目前没有标准寄存器分配给超过 `0xFF` 的值。

具有最高有效位（XLEN-1 = 1）的 `miselect` 值指定为自定义使用，可能是用于通过 `mireg` 访问自定义寄存器。如果 XLEN 更改，则 `miselect` 的最高有效位移动到新位置，保留其之前的值。实现不需要支持 `miselect` 的任何自定义值。

当 `miselect` 是保留范围内的数字（当前为 `0x00`–`0x2F`，`0x40`–`0x6F`，或未指定

Number	Privilege	Width	Name	Description
Machine-Level Window to Indirectly Accessed Registers				
0x350	MRW	XLEN	miselect	Machine indirect register select
0x351	MRW	XLEN	mireg	Machine indirect register alias
Machine-Level Interrupts				
0x304	MRW	64	mie	Machine interrupt-enable bits
0x344	MRW	64	mip	Machine interrupt-pending bits
0x35C	MRW	MXLEN	mtopei	Machine top external interrupt (only with an IMSIC)
0xFB0	MRO	MXLEN	mtopi	Machine top interrupt
Delegated and Virtual Interrupts for Supervisor Level				
0x303	MRW	64	mideleg	Machine interrupt delegation
0x308	MRW	64	mvien	Machine virtual interrupt enables
0x309	MRW	64	mvip	Machine virtual interrupt-pending bits
Machine-Level High-Half CSRs (RV32 only)				
0x313	MRW	32	midelegh	Upper 32 bits of of mideleg (only with S-mode)
0x314	MRW	32	mieh	Upper 32 bits of mie
0x318	MRW	32	mvienh	Upper 32 bits of mvien (only with S-mode)
0x319	MRW	32	mviph	Upper 32 bits of mvip (only with S-mode)
0x354	MRW	32	miph	Upper 32 bits of mip

表 A.3　高级中断体系结构添加或扩展的 M 特权级 CSR

为自定义使用的数字以上 0xFF），尝试访问 mireg 通常会引发非法指令异常。

通常，仅当实现了 IMSIC 时，外部中断的范围 0x70–0xFF 才会被填充。否则，在 miselect 处于此范围时尝试访问 mireg 也会导致非法指令异常。外部中断区域的内容在 IMSIC 章节（第三章）中有所说明。

当实现了 IMSIC 时，CSR mtopei 也只存在于 IMSIC 间接访问的寄存器中，因此在 IMSIC 章节（第三章）中有所说明。

CSR mtopi 报告了挂起并启用的优先级最高的机器级中断，如第五章所述。

当实现 S-mode 时，CSR mvien 和 mvip 支持监管模式的中断过滤和虚拟中断。这些功能在第五章第三节中解释。

A.2.2 S 特权级 CSR

如果 Hart 实现 S-mode，表 A.4 列出了在 S 特权级添加的 CSR 以及大小被高级中断架构更改的现有的 S 特权级 CSR。这些寄存器的功能都与 M 特权级对应。

Number	Privilege	Width	Name	Description
Supervisor-Level Window to Indirectly Accessed Registers				
0x150	SRW	XLEN	siselect	Supervisor indirect register select
0x151	SRW	XLEN	sireg	Supervisor indirect register alias
Supervisor-Level Interrupts				
0x104	SRW	64	sie	Supervisor interrupt-enable bits
0x144	SRW	64	sip	Supervisor interrupt-pending bits
0x15C	SRW	SXLEN	stopei	Supervisor top external interrupt (only with an IMSIC)
0xDB0	SRO	SXLEN	stopi	Supervisor top interrupt
Supervisor-Level High-Half CSRs (RV32 only)				
0x114	SRW	32	sieh	Upper 32 bits of sie
0x154	SRW	32	siph	Upper 32 bits of sip

表 A.4　高级中断体系结构添加或扩展的 S 特权级 CSR

可通过 siselect/sireg 窗口访问的寄存器空间与 M 特权级的寄存器空间是独立的，但是与之平行，用于 S 特权级中断而不是 M 特权级中断。范围在 0 到 0xFF 内的 siselect 分配的值如下

0x00–0x2F　reserved

0x30–0x3F　major interrupt priorities

0x40–0x6F　reserved

0x70–0xFF　external interrupts (only with an IMSIC)

为了最大限度地提高兼容性，建议 siselect 至少支持 9 位的范围，即 0 到 0x1FF，无论是否存在 IMSIC。

由于 VS CSR vsiselect（*第二章第三节*）*始终具有至少 9 位，而且像其他 VS CSR 一样，在虚拟机（VS-mode或VU-mode）中执行时，*vsiselect *代替* siselect*，实现较小的* siselect *范围允许软件发现它没有在虚拟机中运行。*

与 miselect 一样，具有最高有效位（XLEN-1 = 1）的 siselect 值指定为自定义使用。如果 XLEN 更改，则 siselect 的最高有效位移动到新位置，保留其之

前的值。实现不需要支持 siselect 的任何自定义值。

当 siselect 是保留范围内的数字（当前为 0x00–0x2F，0x40–0x6F，或未指定为自定义使用的数字以上 0xFF），或在没有 IMSIC 的情况下处于 0x70–0xFF 范围内时，尝试访问 sireg 应尽可能引发非法指令异常（除非在虚拟机中执行，下一节讨论）。

请注意，siselect 和 sireg 的宽度始终为当前的 XLEN，而不是 SXLEN。因此，例如，如果 MXLEN = 64 且 SXLEN = 32，则当当前特权模式为 M（运行 RV64 代码）时，这些寄存器为 64 位，但特权模式为 S（RV32 代码）时为 32 位。

CSR stopei 在 IMSIC 章节（第三章）中有所描述。

寄存器 stopi 报告了挂起并启用的优先级最高的监管级别中断，如第五章所述。

A.2.3 Hypervisor 和 VS 特权级 CSR

如果 Hart 实现了特权架构的虚拟化扩展，则表 A.5 中列出的虚拟化管理程序（Hypervisor）和 VS CSR 也将被添加或扩展为 64 位。

表中的新 VS CSR（vsiselect，vsireg，vstopei 和 vstopi）都与 S 特权级 CSR 匹配，并在虚拟机（在VS-mode或VU-mode中）执行时替换这些 S 特权级 CSR。

CSR vsiselect 需要支持至少 0 到 0x1FF 范围的 9 位，无论是否实现 IMSIC。与 siselect 类似，vsiselect 的具有最高有效位（XLEN-1 = 1）的值指定为自定义使用。如果 XLEN 更改，则 vsiselect 的最高有效位移动到新位置，保留其之前的值。

与 siselect 和 sireg 一样，vsiselect 和 vsireg 的宽度始终为当前的 XLEN，而不是 VSXLEN。因此，例如，如果 HSXLEN = 64 且 VSXLEN = 32，则当在 HS 模式下运行 RV64 代码的虚拟化管理程序访问这些寄存器时，它们为 64 位，但在在 VS 模式下运行 RV32 代码的客户操作系统访问这些寄存器时为 32 位。

vsiselect 可选择的寄存器空间比 M 特权级和 S 特权级更有限：

0x000–0x02F	reserved
0x030–0x03F	inaccessible
0x040–0x06F	reserved
0x070–0x0FF	external interrupts (IMSIC only), or inaccessible
0x100–0x1FF	reserved

当 vsiselect 具有不可访问寄存器的编号时，尝试从 M 模式或 HS 模式访问 vsireg 会引发非法指令异常，而尝试从 VS 模式或 VU 模式访问 sireg（实际上是

Number	Privilege	Width	Name	Description
Delegated and Virtual Interrupts, Interrupt Priorities, for VS Level				
0x603	HRW	64	hideleg	Hypervisor interrupt delegation
0x608	HRW	64	hvien	Hypervisor virtual interrupt enables
0x609	HRW	HSXLEN	hvictl	Hypervisor virtual interrupt control
0x645	HRW	64	hvip	Hypervisor virtual interrupt-pending bits
0x646	HRW	64	hviprio1	Hypervisor VS-level interrupt priorities
0x647	HRW	64	hviprio2	Hypervisor VS-level interrupt priorities
VS-Level Window to Indirectly Accessed Registers				
0x250	HRW	XLEN	vsiselect	Virtual supervisor indirect register select
0x251	HRW	XLEN	vsireg	Virtual supervisor indirect register alias
VS-Level Interrupts				
0x204	HRW	64	vsie	Virtual supervisor interrupt-enable bits
0x244	HRW	64	vsip	Virtual supervisor interrupt-pending bits
0x25C	HRW	VSXLEN	vstopei	Virtual supervisor top external interrupt (only with an IMSIC)
0xEB0	HRO	VSXLEN	vstopi	Virtual supervisor top interrupt
Hypervisor and VS-Level High-Half CSRs (RV32 only)				
0x613	HRW	32	hidelegh	Upper 32 bits of hideleg
0x618	HRW	32	hvienh	Upper 32 bits of hvien
0x655	HRW	32	hviph	Upper 32 bits of hvip
0x656	HRW	32	hviprio1h	Upper 32 bits of hviprio1
0x657	HRW	32	hviprio2h	Upper 32 bits of hviprio2
0x214	HRW	32	vsieh	Upper 32 bits of vsie
0x254	HRW	32	vsiph	Upper 32 bits of vsip

表 A.5　高级中断体系结构添加或扩展的 Hypervisor 和 VS 特权级 CSR 。参数 HSXLEN 只是扩展的 S 模式的 SXLEN 的另一个名称。

vsireg）会引发虚拟指令异常。类似地，当 vsiselect 具有保留值时，包括未设置最高有效位的 0x1FF 以上的值，从 M 模式或 HS 模式访问 vsireg 应最好引发非法指令异常，而尝试从 VS 模式或 VU 模式访问 sireg（实际上是 vsireg）应最好引发虚拟指令异常。

要求 vsiselect 的范围为 0–0x1FF，即使大部分或全部空间都被保留或不可访问，也允许虚拟化管理程序模拟已实现范围内间接访问的寄存器，包括可能在未来标准化 0x100–0x1FF 的寄存器。

外部中断寄存器的位置（编号为 0x70–0xFF）仅在 hstatus 的 VGEIN 字段是已实现的客户机外部中断号，而不是 0 时才可访问。如果 VGEIN 不是已实现的客户机外部中断号（包括未实现 IMSIC 的情况），则所有非自定义值的 vsiselect 要么保留，要么指定不可访问的寄存器。

同样地，当 hstatus.VGEIN 不是已实现的客户机外部中断号时，尝试从 M 模式或 HS 模式访问 CSR vstopei 会引发非法指令异常，而尝试从 VS 模式访问 stopei 会引发虚拟指令异常。

表 A.5 中的新虚拟化管理程序 CSR（hvien，hvictl，hviprio1 和 hviprio2）增强了向 VS 级别注入中断的 hvip。这些寄存器的使用在第六章虚拟机中断有所涉及。

A.2.4 虚拟指令异常

遵循虚拟机扩展的默认规则，从 VS 模式尝试直接访问虚拟化管理程序或 VS CSR，或从 VU 模式访问任何 S 特权级的 CSR（包括虚拟化管理程序和 VS CSR），通常不会引发非法指令异常，而是引发虚拟指令异常。有关详细信息，请参阅 RISC-V 特权架构。

A.2.5 通过状态使能寄存器的访问控制

如果扩展 Smstateen 与高级中断架构（AIA）一起实现，那么状态使能寄存器 mstateen0 的三个位将控制特权模式低于 M 模式的状态访问 AIA 添加的状态。

bit 60 CSRs siselect, sireg, vsiselect, and vsireg

bit 59 all other state added by the AIA and not controlled by bits 60 and 58

bit 58 all IMSIC state, including CSRs stopei and vstopei

如果 mstateen0 中的其中一个位为零，则尝试从低于 M 模式的特权模式访问相应的状态将导致非法指令异常。与往常一样，状态使能 CSR 不影响在 M 模式下的任何状态的可访问性，只在低于 M 模式的特权模式下生效。有关更多说明，请参阅 ISA 扩展 Smstateen 的文档。

位 59 控制对 AIA CSR siph、sieh、stopi、hidelegh、hvien/hvienh、hviph、hvictl、hviprio1/hviprio1h、hviprio2/hviprio2h、vsiph、vsieh 和 vstopi 的访问，以及通过 siselect + sireg 访问的监管级中断优先级（第五章的 iprio 数组）。

位 58 仅在 Hart 有 IMSIC 时在 mstateen0 中实现。如果 Hypervisor 扩展也被实现，则该位不影响 Hypervisor CSR hgeip 和 hgeie 或 hstatus 的 VGEIN 字段的行为或可访问性。特别是，即使 mstateen0 的位 58 为零，IMSIC 中的客户机外部中断仍将在 HS-mode 中的 hgeip 中可见。

Smstateen 的早期一个未经批准的草案指出，当 mstateen0 的位 58 为 0 时，hgeip、hgeie 和 hstatus 的 VGEIN 字段都是只读的 0 。这种影响不再正确。

如果 Hypervisor 扩展被实现，那么相同的三个位也在 Hypservisor CSR hstateen0 中定义，但只涉及在特权模式 VS 和 VU 中执行的虚拟机可能可访问的状态：

bit 60 CSRs siselect and sireg (really vsiselect and vsireg)

bit 59 CSRs siph and sieh (RV32 only) and stopi (really vsiph, vsieh, and vstopi)

bit 58 all state of IMSIC guest interrupt files, including CSR stopei (really vstopei)

如果 hstateen0 中的一个位为零，并且相应的位在 mstateen0 中为一，则尝试从 VS 或 VU-mode 访问相应的状态将引发虚拟指令异常。（但请注意，对于高半部 CSR siph 和 sieh，仅当 XLEN = 32 时才适用。当 XLEN > 32 时，访问 siph 或 sieh 将像通常一样引发非法指令异常，而不是虚拟指令异常。）

位 58 仅在 hart 有 IMSIC 时在 hstateen0 中实现。此外，即使有 IMSIC，如果 IMSIC 没有用于客户机外部中断的客户机中断文件（第三章），位 58 也可能（或可能不）在 hstateen0 中为只读零。当此位为零时（无论是只读零还是设为零），虚拟机将无法访问 Hart 的 IMSIC，就像 hstatus.VGEIN = 0 一样。

扩展 Ssstateen 被定义为 Smstateen 的 S 特权级视图。因此，Ssaia 和 Ssstateen 的组合包含了上面为 hstateen0 定义的位，但不包括 mstateen0 的位，因为 M 特权级 CSRs 对 S 特权级不可见。

A.3 IMSIC

传入 MSI 控制器（IMSIC）是一个可选的 RISC-V 硬件组件，与一个 Hart 紧密耦合，每个 Hart 有一个 IMSIC。IMSIC 接收和记录 Hart 收到的消息驱动中断（MSI），并在有待处理的已启用中断时向 Hart 发送中断信号。

IMSIC 在机器的地址空间中有一个或多个内存映射寄存器用于接收 MSI。除了这些内存映射寄存器之外，软件主要通过连接 IMSIC 的 Hart 上几个 RISC-V CSR

与 IMSIC 进行交互。

A.3.1 中断文件和中断标识

在 RISC-V 系统中，MSI 不仅可以定向到特定的 Hart，还可以定向到特定 Hart 的特定特权级，例如 M 特权级或 S 特权级。此外，当 Hart 实现虚拟化扩展时，IMSIC 可以选择允许 MSI 定向到 VS 特权级的特定虚拟 Hart。

对于可以将 MSI 定向到 Hart 的每个特权级和每个虚拟 Hart，Hart 对应的 IMSIC 都包含一个单独的中断文件。假设 Hart 实现了 S 模式，它的 IMSIC 至少有两个中断文件，一个用于 M 特权级，另一个用于 S 特权级。当 Hart 还实现了虚拟化扩展时，它的 IMSIC 可能会有额外的中断文件用于虚拟 Hart，称为客户机中断文件。IMSIC 为虚拟 Hart 提供的客户机中断文件的数量正好是 GEILEN ，即由 RISC-V 特权架构定义的虚拟化扩展支持的客户机外部中断数量。

每个单独的中断文件主要由两个相同大小的位数组组成，一个数组用于记录到达但尚未服务的 MSI（中断挂起位），另一个数组用于指定 Hart 当前可接受哪些中断（中断使能位）。每个位在两个数组中对应着一个不同的中断标识号，用于区分来自不同源的 MSI。由于 IMSIC 是附加到 Hart 的外部中断控制器，因此中断文件的中断标识成为连接的 Hart 的外部中断的次要标识。

中断文件支持的中断标识数（即每个数组中的活动位数）比 64 的倍数少一个，最少为 63，最多为 2047。

平台标准可以增加每个中断文件必须实现的最小中断标识数。

当一个中断文件支持 N 个不同的中断标识时，有效的标识号在 1 到 N 之间（包括 1 和 N）。在这个范围内的标识号被认为是中断文件实现的；范围外的号码则没有被实现。数字 0 永远不是一个有效的中断标识。

IMSIC 硬件不会假设一个中断文件中的中断标识号与另一个中断文件中的中断标识号之间存在任何连接。通常期望软件将相同的中断标识号分配给不同的中断文件中不同的 MSI 源，而无需跨中断文件进行协调。因此，在系统中可以单独区分的 MSI 源的总数可能是单个中断文件中中断标识数乘以系统中所有 Hart 上中断文件的总数的乘积。

并非所有系统中的中断文件都具有相同的大小（实现相同数量的中断标识）。对于给定的 Hart，客户机外部中断的中断文件必须具有相同的大小，但是 M 特权级和 S 特权级的中断文件的大小可能与客户机外部中断的中断文件的大小不同且互不相同。同样，不同 Hart 的中断文件大小可能不同。

平台可能提供一种方式让软件配置 IMSIC 中中断文件的数量和/或大小，例如允许在 M 特权级上使用较小的中断文件来交换更大的 S 特权级的中断文件，反之亦然。任何这样的可配置性都不在本规范的讨论范围之内，但是建议只有 M 特权级才能控制 IMSIC 中中断文件的数量和大小。

A.3.2 MSI 编码

已经建立的标准（特别是 PCI 和 PCI Express）规定，设备发出的每个单独的消息驱动中断（MSI）都采用设备进行自然对齐的 32 位写入形式，地址和值都由软件在设备（或设备控制器）上进行配置。根据设备或控制器符合的标准版本，地址可能被限制在较低的 4-GiB（32 位）范围内，并且写入的值可能限制在 16 位范围内，其中高 16 位始终为零。

当 RISC-V Hart 具备 IMSIC 时，设备发出的 MSI 通常直接发送到由软件选择处理中断的特定 Hart（可能基于某种中断亲和策略）。MSI 通过存在于接收 Hart 的 IMSIC 中的相应中断文件来定向到特定特权级或虚拟 Hart。MSI 写入地址是连接到目标中断文件的特定字长寄存器的物理地址。MSI 写入数据只是要在该中断文件中挂起的中断的标识号（最终成为对应 Hart 的外部中断的次要标识）。

通过在设备上配置 MSI 的地址和数据，系统软件完全控制：（a）哪个 Hart 接收特定设备中断，（b）目标特权级别或虚拟 Hart，（c）代表目标中断文件中 MSI 的标识号。其中 a 和 b 由 MSI 写入地址对应的中断文件确定，而 c 由 MSI 写入数据确定。

由于 *IMSIC* 支持的最大中断标识号为 *2047*，因此对 *MSI* 数据的 *16* 位限制没有问题。

当实现了虚拟化扩展并且设备直接由客户操作系统管理时，设备发出的 MSI 地址最初是客户机物理地址，因为它们是由客户机操作系统在设备上配置的。这些客户机地址必须由 IOMMU 进行翻译，由虚拟机监控程序配置 IOMMU 来将这些 MSI 重定向到正确的客户机外部中断的中断文件中。有关此主题的更多信息，请参见第八章。

A.3.3 中断优先级

在单个中断文件中，中断优先级直接由中断标识号确定。较低的标识号具有较高的优先级。

由于 *MSI* 让软件完全控制中断文件中标识号的分配，因此软件可以自由选择反映所需中断相对优先级的标识号。

确实，如果中断文件包括要分配给每个中断标识的优先级数字的数组，软件

可以更动态地调整中断优先级。然而，我们认为这种额外的灵活性不会经常被利用，以至于不能证明其所需的额外硬件开销。实际上，对于许多当前使用 *MSI* 的系统，软件通常会完全忽略中断优先级，并表现得好像所有中断具有相同的优先级。

中断文件的最低标识号被赋予了最高的优先级，而不是反向顺序，因为仅对于最高优先级的中断，优先级顺序可能需要仔细管理，而保证在所有系统中存在的是较低的编号 *1* 到 *63*（或者可能是 *1* 到 *255*）。例如，考虑中断文件的最高优先级中断，最紧急的中断总是标识号 *1*。如果优先顺序被反转，最高优先级的中断将在不同的机器上具有不同的标识号，具体取决于中断文件实现了多少个标识号。软件能够为最高优先级的中断分配固定的标识号，这被认为是值得的，即使中断优先级与自然数字顺序相反可能有些奇怪。

A.3.4 复位和状态

当 IMSIC 复位时，其所有中断文件的状态都变得有效和一致，但除此之外的状态是未指定的，可能会有例外情况，例如 M 特权级和 S 特权级中断文件的 `eidelivery` 寄存器，详见第八节。

如果 IMSIC 包含 S 特权级中断文件，并且连接的硬件上的软件启用了之前被禁用的 S-mode（例如通过将 CSR `misa` 的位 S 从 0 更改为 1），则 S 特权级中断文件的所有状态都是有效和一致的，但除此之外的状态是未指定的。

同样地，如果 IMSIC 包含客户机中断文件，并且连接的硬件上的软件启用了之前被禁用的虚拟化扩展（例如通过将 `misa` 的位 H 从 0 更改为 1），则 IMSIC 的客户机中断文件的所有状态都是有效和一致的，但除此之外的状态是未指定的。

A.3.5 中断文件的地址范围

IMSIC 中的每个中断文件都有一个或两个内存映射的 32 位 寄存器，用于接收 MSI 写入。这些内存映射的寄存器位于一个自然对齐的 4-KiB 物理地址空间区域内（一页），即每个中断文件一个页面。

中断文件的地址范围布局如下：

offset	size	register name
`0x000`	4 bytes	`seteipnum_le`
`0x004`	4 bytes	`seteipnum_be`

中断文件的 4-KiB 地址范围中的所有其他字节都是保留的，并且必须实现为只读零。在中断文件的地址范围内，仅支持自然对齐的 32 位 简单读写。写入只读字节会被忽略。对于其他形式的访问（其他大小、不对齐的访问或 AMO），IMSIC 实现应该优先报告访问故障或总线错误，否则必须忽略该访问。

如果 *i* 是一个实现的中断标识号，将值 *i* 以小端字节顺序写入 `seteipnum_le`

（按编号设置外部中断挂起位，小端字节顺序），会导致中断 *i* 的挂起位被设置为 1。如果写入的值不是以小端字节顺序表示的实现的中断标识号，则会忽略对 `seteipnum_le` 的写入。

对于支持大端字节顺序的系统，如果 *i* 是实现的中断标识号，则将值 *i* 以大端字节顺序写入 `seteipnum_be`（按编号设置外部中断挂起位，大端字节顺序）会导致中断 *i* 的挂起位被设置为 1。如果写入的值不是以大端字节顺序表示的实现的中断标识号，则会忽略对 `seteipnum_be` 的写入。仅支持小端字节顺序的系统可以选择忽略对 `seteipnum_be` 的所有写入。

在大多数系统中，`seteipnum_le` 是用于定向到该中断文件的 MSI 的写入端口。对于主要为大端字节顺序构建的系统，`seteipnum_be` 可能为某些设备定向到该中断文件的 MSI 的写入端口。

无论何种情况，对 `seteipnum_le` 或 `seteipnum_be` 的读取都返回零。

当不被忽略时，对中断文件的地址范围的写入保证最终反映在中断文件中，但不一定是立即的。对于单个中断文件，对其地址范围的多个写入（存储），虽然具有任意的延迟，但总是按照 RISC-V 非特权指令集所定义的存储指令的全局内存顺序 (global memory order) 发生。

在大多数情况下，对中断文件地址范围的写入完成后，写入对中断文件的影响的任何延迟都与内存系统中的其他延迟无法区分。然而，如果一个 *Hart* 向自己的 *IMSIC* 的 `seteipnum_le` 或 `seteipnum_be` 寄存器写入数据，则存储指令完成和中断文件中相应的中断挂起位设置之间的延迟可能对 *Hart* 可见。

A.3.6 多个中断文件的地址布局

IMSIC 实现的每个中断文件都有自己的内存区域，如前一节所述，占用机器地址空间的一个 4-KiB 页面。在实践中，所有 IMSIC 的 M 特权级中断文件的内存页应该位于地址空间的一个部分中，并且所有 S 特权级和客户机中断文件的内存页应该类似地位于地址空间的另一个部分中，按照以下规则。

将 *M* 特权级中断文件与地址空间中的其他中断文件分开的主要原因是，实现物理内存保护（*PMP*）的 *Hart* 可以仅使用一个 *PMP* 表项授予 *S* 特权级访问所有 *S* 特权级和客户机中断文件。如果 *M* 特权级中断文件的内存页面与低特权级的中断文件相间排列，那么在授予 *S* 特权级访问所有非机器级中断文件时所需的 *PMP* 表项数量可能等于系统中的 *Hart* 数量。

如果系统规定 Hart 被分成多个组，每个组被分配到自己的地址空间区域中，那么最好的做法是将每个 Hart 组的 M 特权级中断文件分别放置在一起，并将每个 Hart 组的 S 特权级和客户机中断文件分别放置在一起。下面稍后会进一步讨论这种情况。

如果系统在地址空间中将 *Hart* 分成组，那么可能是因为每个组存在于单独的芯片（或多芯片模块中的芯片块）上，将多个芯片的地址空间编织在一起是不可行的。在这种情况下，授予 *S* 特权级访问所有非机器级中断文件需要每个组一个 *PMP* 表项。

为了在地址空间中定位中断文件的内存页面，假设每个 Hart（或每个 Hart 组）都有一个唯一的 Hart 号，该号码可能与 RISC-V 特权架构分配给 Hart 的唯一 Hart 标识符（Hart IDs）相关或无关。为了方便寻址，所有 M 特权级中断文件的内存页面（或单个 Hart 组的所有 M 特权级中断文件的内存页面）应该按照以下公式进行排列，使得 Hart 号码为 h 的 M 特权级中断文件的地址为 $A + h \times 2^C$，其中 A 和 C 是整数常数。如果最大的 Hart 号码是 $h_{\max}$，则 $k = \lceil \log_2(h_{\max} + 1) \rceil$，它是表示任何 Hart 号码所需的位数。基址 A 应该对齐到 2^{k+C} 地址边界，使得 $A + h \times 2^C$ 总是等于 $A | (h \times 2^C)$，其中竖线（|）表示按位逻辑或。

C 可以取到的最小值为 12，其中 2^C 是一个 4-KiB 页面的大小。如果 $C > 12$，则每个 M 特权级中断文件的内存页面的起始地址不仅对齐到 4-KiB 页面，而是对齐到更严格的 2^C 地址边界。在 A 到 $A + 2^{k+C} - 1$ 的 2^{k+C} 大小的地址范围内，任何未被 M 特权级中断文件占用的 4-KiB 页面都应该用只读零值的 32-bit 字填充，使得任何对齐的字的读取都返回零，任何对齐的字的写入都被忽略。

所有 S 特权级中断文件的内存页面（或单个 Hart 组的所有 S 特权级中断文件的内存页面）应该以类似的方式排列，以便 Hart 号码为 h 的 S 特权级中断文件的地址为 $B + h \times 2^D$，其中 B 和 D 是整数常数，并且基址 B 对齐到 2^{k+D} 地址边界。

如果 IMSIC 实现了客户机中断文件，那么 IMSIC 的 S 特权级中断文件和其客户机中断文件的内存页面应该是连续的，从最低地址的 S 特权级中断文件开始，然后按客户机中断号排序的顺序依次是每个客户机中断文件。因此，常数 D 可以取的最小值是 $\lceil \log_2(\text{maximum GEILEN} + 1) \rceil + 12$，其中 GEILEN 是 IMSIC 实现的客户机中断文件数量。

在 B 到 $B + 2^{k+D} - 1$ 的 2^{k+D} 大小的地址范围内，任何未被中断文件（S 特权级或客户机）占用的 4-KiB 页面都应该用只读零值的 32-bit 字填充。

当一个系统将 Hart 分成组，每个组都位于地址空间独立的部分中时，中断文件的内存页面地址应该遵循以下公式 $g \times 2^E + A + h \times 2^C$ 用于 M 特权级中断文件，$g \times 2^E + B + h \times 2^D$ 用于 S 特权级中断文件，其中 g 是 组号，h 是相对于组的 Hart 号，E 是另一个整数常数 $\geqslant k + \max(C, D)$，但通常要大得多。如果最大的组号是 $g_{\max}$，则 $j = \lceil \log_2(g_{\max} + 1) \rceil$，它是表示任何组号所需的位数。除了分别是 2^{k+C} 和 2^{k+D} 的倍数之外，A 和 B 还应该被选择为

$\left((2^j - 1) \times 2^E\right)$ & $A = 0$ and $\left((2^j - 1) \times 2^E\right)$ & $B = 0$

其中和号（&）表示按位逻辑与。这确保了:

$g \times 2^E + A + h \times 2^C$ always equals $(g \times 2^E)$ | A | $(h \times 2^C)$, and

$g \times 2^E + B + h \times 2^D$ always equals $(g \times 2^E)$ | B | $(h \times 2^D)$.

只有在每个组内部，而不是在分离的组之间，才会使用只读零页面进行填充。具体来说，如果 g 是 0 到 $2^j - 1$ 之间（包括两端）的任何整数，那么在地址范围:

$g \times 2^E + A$ through $g \times 2^E + A + 2^{k+C} - 1$, and

$g \times 2^E + B$ through $g \times 2^E + B + 2^{k+D} - 1$,

内，任何未被中断文件占用的页面都应该是只读零。

另请参见第四章，用于确定发出 MSIs 的目标地址的默认算法，这些地址应该是 IMSIC 中断文件的地址。

综合论文训练记录表

学生姓名	田凯夫	学号	2019011264	班级	计 93
论文题目	RISC-V 用户态中断扩展设计与实现				
主要内容以及进度安排	用户态中断是近年提出的通信机制，可以减少传统 IPC 机制存在的上下文切换的开销。现有的工作包括两个方面，分别为在 RISC-V N 扩展的基础上实现的用户态外部中断和 intel 提出的 x86 用户态中断扩展。我的工作内容是聚焦进程间跨核通信问题，设计 RISC-V 用户态中断扩展，修改 QEMU 和 Rocket Chip 以支持设计方案，最后搭建真实硬件环境对设计方案进行验证和评估，对比用户态中断和传统 IPC 机制的性能，撰写论文并完成最终答辩。 进度安排： 1.12 - 2.5：给出完整设计文档，在 QEMU 中实现用户态中断的功能。 2.5 - 4.1：在 OS 中支持用户态中断，实现用户库和测试程序。 4.1 - 5.1：在 Rocket Chip 中支持用户态中断，并通过行为仿真进行验证。 5.1 – 6.7：在开发板上搭建实验环境，分析测试结果，和传统的 IPC 进行性能对比，争取发表论文。 指导教师签字： 考核组组长签字： 2023 年 1 月 12 日				

中期考核意见	通过 考核组组长签字： 2023 年 4 月 15 日
指导教师评语	论文对用户态中断进行了深入的分析，并且在软件仿真和FPGA上进行了实现。通过对Rocket和Linux操作系统事进行软硬件协同优化，在实验上取得了显著的性能提升。论文主题明确，内容详实，体现了作者深入的理解和扎实的实践能力。是一篇优秀的毕业论文 指导教师签字： 2023年 6 月 7 日
评阅教师评语	作者在RISC-V指令集上提出了用户态中断方案，通过软硬件结合的方式对方案进行了验证和评估，实验思路清晰，实验内容完整，该设计方案最终取得了很不错的性能提升，论文撰写规范，内容详实，有理有据，充分体现了作者的研究过程和研究方法，是一篇很好的论文。 评阅教师签字： 2023年 6 月 7 日

答辩小组评语	通过 答辩小组组长签字：[illegible] 2023 年 6 月 7 日

总成绩：A-

教学负责人签字：[illegible]

2023年 6 月 12日