清 华 大 学

综 合 论 文 训 练

题目：基于 RISC-V 的用户态中断扩展

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 5 月 27 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：_____ 导师签名：_____ 日　期：_____

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工作的效率。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，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，移植 Rocket Chip 并成功启动 Linux 。实验表明，用户态中断相比于信号机制，可以取得 x 倍的性能提升；相比于 eventfd，可以取得 x 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 cooperation, 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 instruction set architecture. 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 x-fold performance improvement; compared with eventfd, it can achieve x-fold performance improvement.

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

目　　录

插图索引

表格索引

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)

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；
2. `SENDUIPI` 指令需要经过两次读内存和一次写内存操作；
3. 用户态中断有一套独立的控制流程，需要通过 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]。

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。

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。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。通过硬件的参与，我们的设计在 N 扩展的基础上，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
- 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
- 接收方没有在目标核上运行，但发送方发送了用户态中断

2.1　整体架构

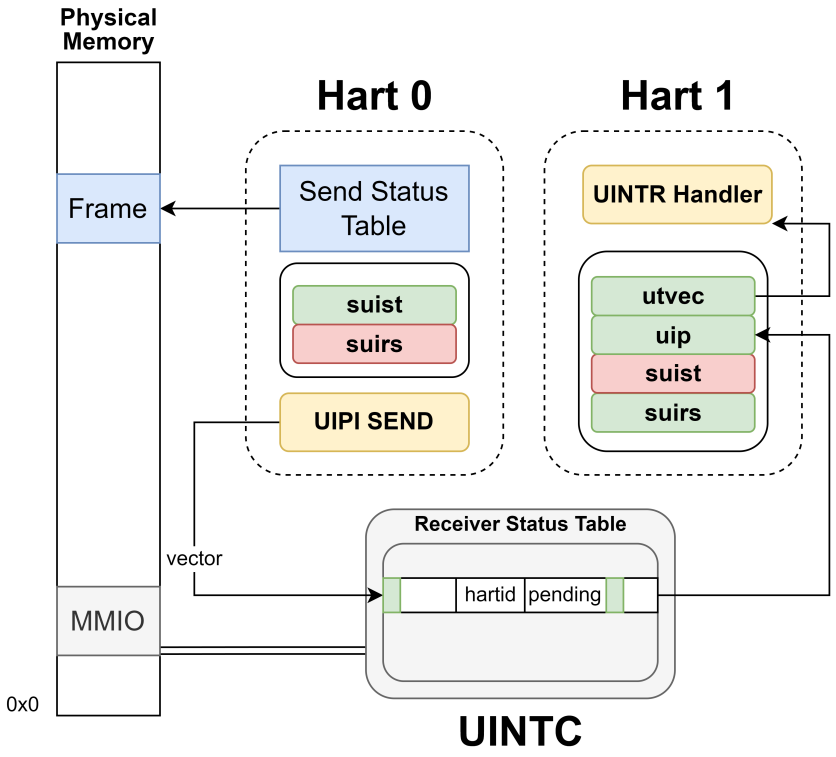

图 2.1　RISC-V 用户态中断扩展整体架构

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：

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

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

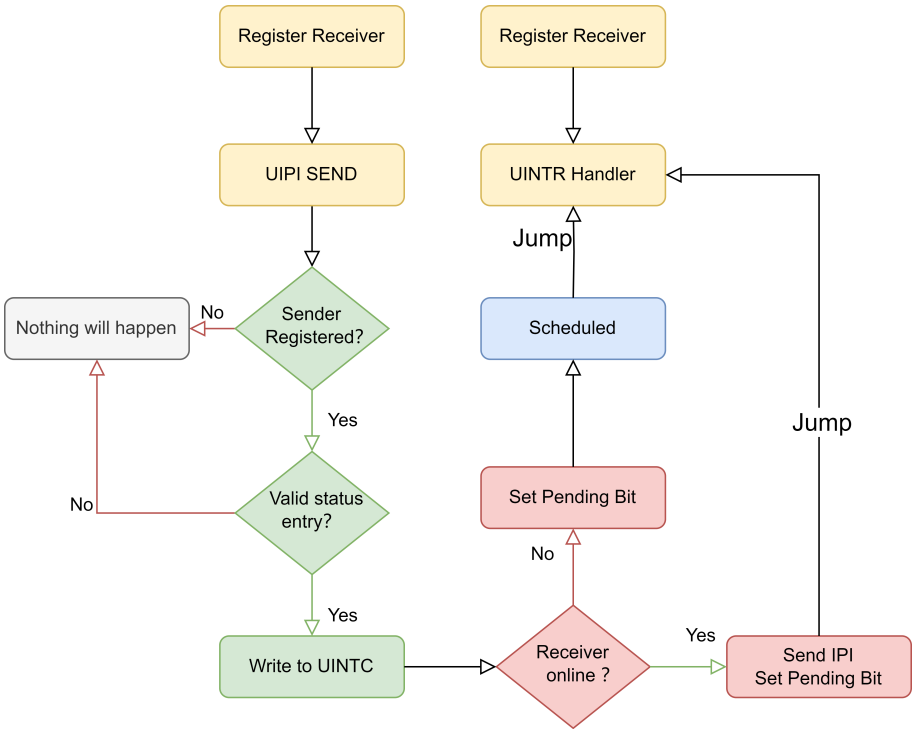

图 2.2 RISC-V 用户态中断工作流程

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，从三个方面介绍了 RISC-V 用户态中断扩展的设计方案。通过添加两个 CPU 控制寄存器、一个外部中断控制器、一条 R 型指令，在 RISC-V N 扩展的基础上实现了用户态中断的发送和处理，同时克服了本章开头提到的潜在的安全问题。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：

```
DEF_HELPER_1(uret, tl, env)
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。

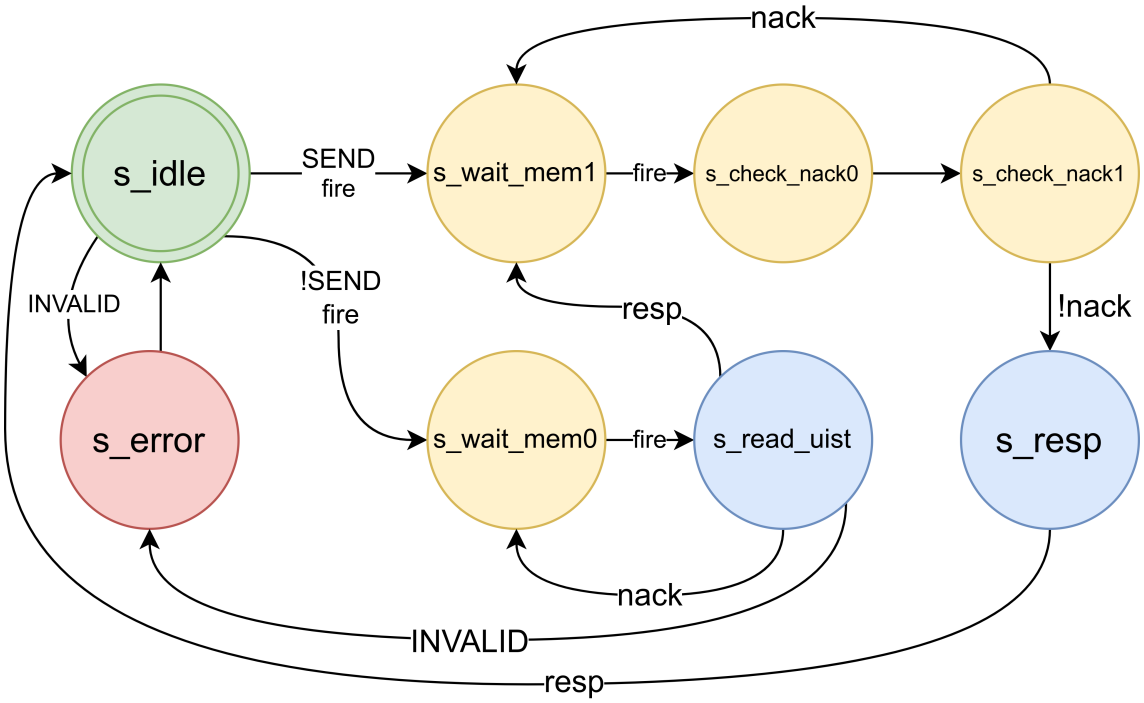

图 3.1　UIPI 协处理器工作流程

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。

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

```
1    struct uintc_priv {
2        struct cpumask lmask; // 位图记录已连接的 CPU
3        void __iomem *regs;   // UINTC 在内核地址空间映射的起始地址
4        resource_size_t size; // UINTC 地址范围大小
5        u32 nr;               // UINTC 槽位数量
6        void *mask;           // 位图记录已分配的槽位
7        spinlock_t lock;      // 互斥锁
8    };
```

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：

```
1    extern void __handler_entry(struct __uintr_frame* frame, void*
     handler) {
2        uint64_t irqs = uipi_read();
3        csr_clear(CSR_UIP, MIE_USIE);
4        uint64_t (*__handler)(struct __uintr_frame * frame, uint64_t) =
     handler;
5        irqs = __handler(frame, irqs);
6        uipi_write(irqs);
7    }
8    static uint64_t __register_receiver(void* handler) {
9        // 设置中断处理函数入口
10       csr_write(CSR_UTVEC, uintrvec);
11       csr_write(CSR_USCRATCH, handler);
12       // 使能 U 态中断处理
13       csr_set(CSR_USTATUS, USTATUS_UIE);
14       csr_set(CSR_UIE, MIE_USIE);
15       int ret = __syscall0(__NR_uintr_register_receiver);
16       // 使能 UINTC
17       uipi_activate();
18       return ret;
19   }
20   // 用户调用接口
21   #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;
```

```
20          int ret;
21          // 注册接收方中断处理函数
22          if (uintr_register_receiver(uintr_handler))
23              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

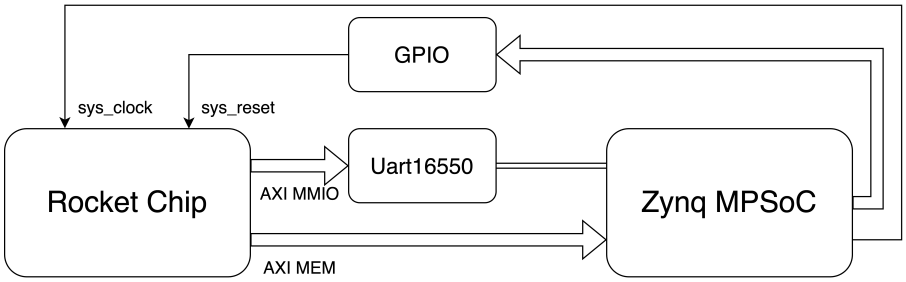

图 5.1　SoC 整体架构

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。

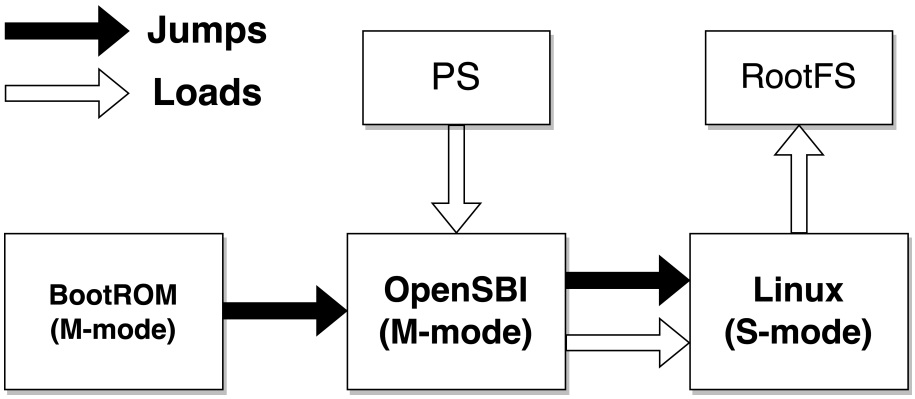

图 5.2　系统软件启动流程

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：

```
OpenSBI v1.2
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name             : rocket-chip-zcu102
Platform Features         : medeleg
Platform HART Count       : 2
Platform IPI Device       : aclint-mswi
Platform Timer Device     : aclint-mtimer @ 10000000Hz
Platform Console Device   : uart8250
Platform HSM Device       : ---
Platform PMU Device       : ---
Platform Reboot Device    : ---
Platform Shutdown Device  : ---
Firmware Base             : 0x80000000
Firmware Size             : 140 KB
Runtime SBI Version       : 1.0

Domain0 Name              : root
Domain0 Boot HART         : 0
Domain0 HARTs             : 0*,1*
Domain0 Region00          : 0x0000000002008000-0x000000000200bfff (I)
Domain0 Region01          : 0x0000000002000000-0x0000000002007fff (I)
Domain0 Region02          : 0x0000000080000000-0x0000000008003ffff ()
Domain0 Region03          : 0x0000000000000000-0xffffffffffffffff (R,W,X)
Domain0 Next Address      : 0x0000000080200000
Domain0 Next Arg1         : 0x0000000082200000
Domain0 Next Mode         : S-mode
Domain0 SysReset          : yes

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART Priv Version    : v1.11
Boot HART Base ISA        : rv64imafdcx
Boot HART ISA Extensions  : none
Boot HART PMP Count       : 8
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 30
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000333
Boot HART MEDELEG         : 0x000000000000b109
sbi_hart_switch_mode 0x80200000 0x1 0x0 0x82200000
[    0.000000] Linux version 6.0.0-gf975e2f3f358-dirty (oslab@oslab) (riscv64-unknown-linux-gnu
-gcc (g2ee5e430018) 12.2.0, GNU ld (GNU Binutils) 2.40.0.20230214) #65 SMP Thu May 25 17:09:29
CST 2023
[    0.000000] OF: fdt: Ignoring memory range 0x80000000 - 0x80200000
[    0.000000] Machine model: freechips,rocket-chip-zcu102
[    0.000000] earlycon: ns16550a0 at MMIO 0x0000000060001000 (options '115200n8')
[    0.000000] printk: bootconsole [ns16550a0] enabled
[    0.000000] efi: UEFI not found.
[    0.000000] Zone ranges:
[    0.000000]   DMA32    [mem 0x0000000080200000-0x000000008fffffff]
[    0.000000]   Normal   empty
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000000080200000-0x000000008fffffff]
[    0.000000] Initmem setup node 0 [mem 0x0000000080200000-0x000000008fffffff]
[    0.000000] SBI specification v1.0 detected
[    0.000000] SBI implementation ID=0x1 Version=0x10002
[    0.000000] SBI TIME extension detected
```

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

如图是用户态与不同 IPC 机制的性能对比结果。

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。

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。时钟中断会导致用户程序陷入到内核，从而导致用户态中断的响应延迟增加，因此用户态中断性能存在一定波动。

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 机制的加速比：

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 机制有很大的性能提升。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 x 倍的性能提升；相比于 eventfd，可以取得 x 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。

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

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：_____ 日　期：_____

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）

目录

A.1　引言

这篇文档给出了 RISC-V 高级中断架构的规范，包括：

(a) 针对 RISC-V 特权架构规范的扩展；

(b) 两个标准的 RISC-V 中断控制器：高级平台级中断控制器（APLIC）和

A.1.1　目标

A.1.2　限制