

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 11 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

**关键词：**用户态中断；软硬件结合；进程间通信

## ABSTRACT

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

# 目 录

插图索引.....	V
表格索引.....	VI
第 1 章 引言 .....	1
第 2 章 背景介绍 .....	3
2.1 RISC-V N 扩展 .....	3
2.1.1 指令集规范 .....	3
2.1.2 外部中断 .....	3
2.2 x86 用户态中断.....	4
第 3 章 硬件设计 .....	5
3.1 整体架构.....	5
3.2 CPU 状态 .....	6
3.3 用户态中断控制器 .....	7
3.4 UIPI 指令 .....	8
3.5 软件接口.....	9
第 4 章 功能级硬件模拟仿真 .....	11
4.1 指令翻译.....	11
4.2 CPU 状态 .....	12
4.3 核间中断.....	12
第 5 章 硬件实现 .....	14
5.1 Chisel .....	14
5.1.1 Rocket Chip .....	14
5.1.2 RoCC.....	15
5.2 RISC-V N 扩展 .....	15
5.3 UINTC 用户态中断控制器 .....	16
5.4 UIPI 协处理器 .....	17

第 6 章 软件实现 .....	19
6.1 Linux .....	19
6.1.1 UINTC 驱动 .....	19
6.1.2 状态保存与恢复 .....	20
6.1.3 进程管理 .....	20
6.2 libc .....	20
6.2.1 库函数实现 .....	20
6.2.2 应用示例 .....	21
第 7 章 系统评估 .....	23
7.1 仿真测试 .....	23
7.1.1 功能级仿真测试 .....	23
7.1.2 电路级仿真测试 .....	23
7.2 性能测试 .....	23
7.2.1 功能级性能测试 .....	23
7.2.2 电路级性能测试 .....	23
第 8 章 结论 .....	24
参考文献 .....	25
致    谢 .....	27
声    明 .....	28
附录 A 外文资料的书面翻译 .....	29

## 插图索引

图 3.1	RISC-V 用户态中断扩展整体架构.....	5
图 3.2	RISC-V 用户态中断工作流程.....	9
图 5.1	Chisel 工作流程 <sup>[12]</sup> .....	14
图 5.2	Rocket 处理器数据通路 .....	15
图 5.3	UIPI 协处理器工作流程.....	18

## 表格索引

表 2.1	RISC-V N CSRs .....	3
表 3.1	接收方状态寄存器 .....	6
表 3.2	发送方状态寄存器 .....	6
表 3.3	发送方状态 .....	6
表 3.4	接收方状态 .....	7
表 3.5	UINTC 操作码 .....	7
表 3.6	UINTC 地址映射 .....	8



## 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

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<sup>[1]</sup>，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<sup>[2]</sup>；
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，内核值提供最基本的服务，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<sup>[3]</sup>，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<sup>[4]</sup>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<sup>[5]</sup>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<sup>[6]</sup>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。

在前人工作的基础上，笔者设计并验证了 RISC-V 用户态中断扩展，通过 QEMU 模拟器对设计草案进行完善，并在 Linux 中加入了对 RISC-V 用户态中断扩展的支持；此外，笔者还在 Rocket Chip 中加入了对用户态扩展的支持，通过软件仿真验证实现，并在 zcu102 开发板上进行性能测试，证明了用户态中断无论是在模拟环境下还是在真实的硬件环境下，相比于传统的 IPC 机制可以取得明显的性能提升。

本文主要分为以下几个部分：

- **背景介绍：**简要介绍 RISC-V N 扩展和 x86 用户态中断指令规范；
- **设计草案：**RISC-V 用户态中断扩展设计草案；
- **软件实现：**介绍在 QEMU、Linux 等系统软件上关于 RISC-V 用户态中断扩展的实现细节；
- **硬件实现：**介绍在 Rocket Chip 上关于 RISC-V 用户态中断扩展的实现细节；
- **性能评估：**分别对软件实现和硬件实现进行性能测试和分析。

## 第 2 章 背景介绍

### 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。

#### 2.1.1 指令集规范

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
uiip	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 2.1 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 UIPIE 赋值给 UIE，然后跳转至 uepc。

#### 2.1.2 外部中断

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 上对设计方案进行了性能评估。通过将用户态中断应用在用户态串口驱动中，证明了其在吞吐率和延时方面的性能表现优于内核态驱动。

## 2.2 x86 用户态中断

2021 年 5 月发布的 Intel 指令集架构扩展<sup>[7]</sup> 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。

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<sup>[8]</sup>。

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。

## 第 3 章 硬件设计

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

- 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

### 3.1 整体架构

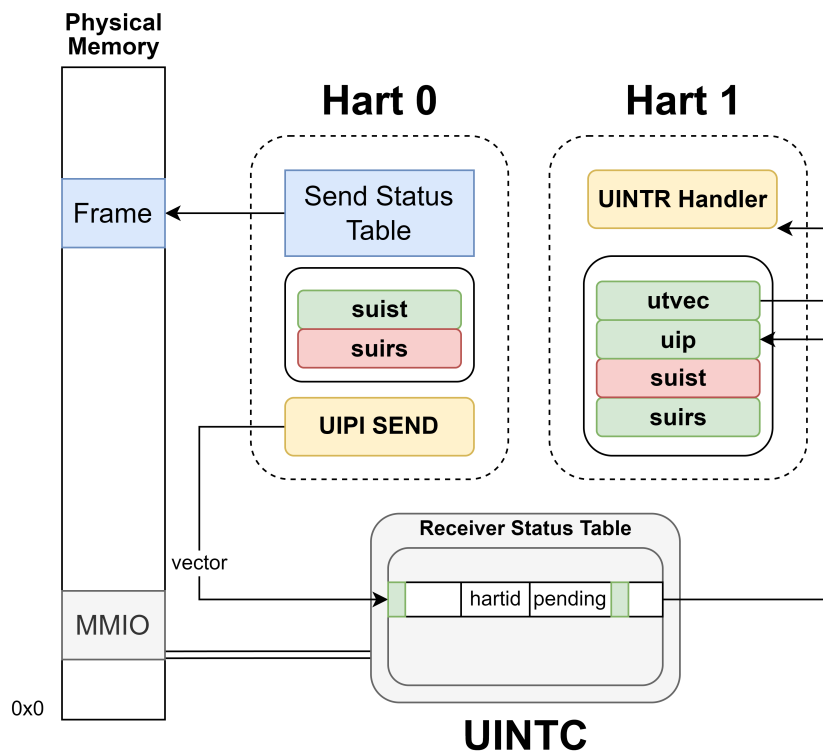


图 3.1 RISC-V 用户态中断扩展整体架构

### 3.2 CPU 状态

`suir`s(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 `ui`pi 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 3.1 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 3.2 发送方状态寄存器

图 3.1 中 Hart 0 运行发送方，`suist` 寄存器被使能；Hart 1 运行接收方，`suir`s 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 3.3 发送方状态

发送方状态在内存中进行维护，表项内容如表 3.3 所示。

### 3.3 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 3.4 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 3.5 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 3.6 **UINTC** 地址映射

### 3.4 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：

$$(PPN \ll 0xC) + (rs1 \ll 0x3)$$

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

### 3.5 软件接口

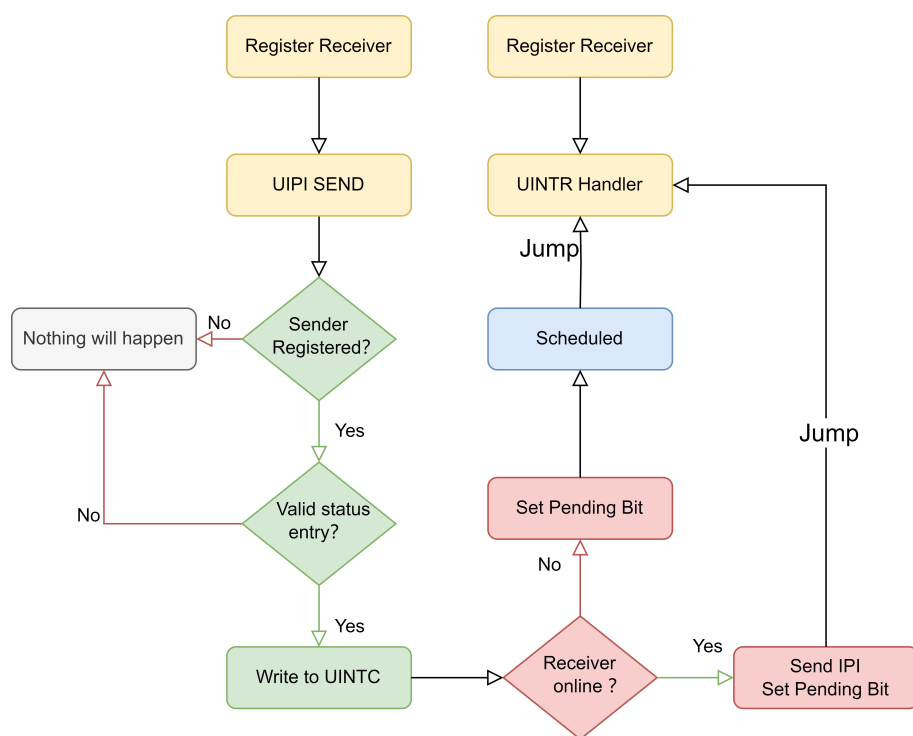


图 3.2 RISC-V 用户态中断工作流程

参考 x86 的用户态中断设计规范<sup>[8]</sup>，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 3.2 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。

## 第 4 章 功能级硬件模拟仿真

QEMU<sup>[9]</sup> 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草案。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。

### 4.1 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`：

<code>uipi_send</code>	<code>0000000</code>	<code>00000</code>	<code>.....</code>	<code>010</code>	<code>.....</code>	<code>1111011</code>	<code>@r2</code>
<code>uipi_read</code>	<code>0000001</code>	<code>00000</code>	<code>.....</code>	<code>010</code>	<code>.....</code>	<code>1111011</code>	<code>@r2</code>
<code>uipi_write</code>	<code>0000010</code>	<code>00000</code>	<code>.....</code>	<code>010</code>	<code>.....</code>	<code>1111011</code>	<code>@r2</code>
<code>uipi_activate</code>	<code>0000011</code>	<code>00000</code>	<code>.....</code>	<code>010</code>	<code>.....</code>	<code>1111011</code>	<code>@r2</code>
<code>uipi_deactivate</code>	<code>0000100</code>	<code>00000</code>	<code>.....</code>	<code>010</code>	<code>.....</code>	<code>1111011</code>	<code>@r2</code>

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：

---

<sup>1</sup> `DEF_HELPER_1(uret, tl, env)`

<sup>2</sup> `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：

---

```
1 DEF_HELPER_2(uiپی_write, void, env, tl)
2 void helper_uپی_write(CPURISCState *env, target_ulong src) {
3     if (uپی_enabled(env, env->suirs)) {
4         uint64_t addr = UINTC_REG_HIGH(env->suicfg, SUIRS_INDEX(env->suirs));
5         cpu_physical_memory_write(addr, &src, 8);
6     }
7 }
```

---

## 4.2 CPU 状态

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。

## 4.3 核间中断

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_uintr.c`，调用 `riscv_uintr_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。

---

```
1 for (i = 0; i < num_harts; i++) {
2     CPUState *cpu = qemu_get_cpu(hartid_base + i);
3     RISCV_CPU *rvcpu = RISCV_CPU(cpu);
4     qdev_connect_gpio_out(dev, i, qdev_get_gpio_in(DEVICE(rvcpu), IRQ_U_SOFT));
5 }
```

---

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 外设的读写端口了：

---

```
1 static const MemoryRegionOps riscv_uintc_ops = {
2     .read = riscv_uintc_read,
3     .write = riscv_uintc_write,
4     .endianness = DEVICE_LITTLE_ENDIAN,
5     .valid = {
6         .min_access_size = 8,
7         .max_access_size = 8
8     }
9 };
```

---

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，从目前的设计草案来看，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<sup>[10]</sup>，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：

---

```
1 if (riscv_has_ext(env, RVN)
2     && prev_priv == PRV_U
3     && get_field(env->mip, MIP_USIP)
4     && get_field(env->mstatus, MSTATUS_UIE)
5     && get_field(env->sideleg, MIP_USIP)) {
6     retpc = env->utvec; // 直接跳转到U态中断处理入口
7     env->uepc = env->sepc; // 指定 \Iuret 到同一条指令
8     mstatus = env->mstatus;
9     mstatus = set_field(mstatus, MSTATUS_UPIE, 1);
10    mstatus = set_field(mstatus, MSTATUS_UIE, 0);
11    env->mstatus = mstatus;
12 }
```

---

## 第 5 章 硬件实现

### 5.1 Chisel

Chisel<sup>[11]</sup> 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，并生成综合的 Verilog。Chisel 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，同时保留了对电路细粒度的控制。

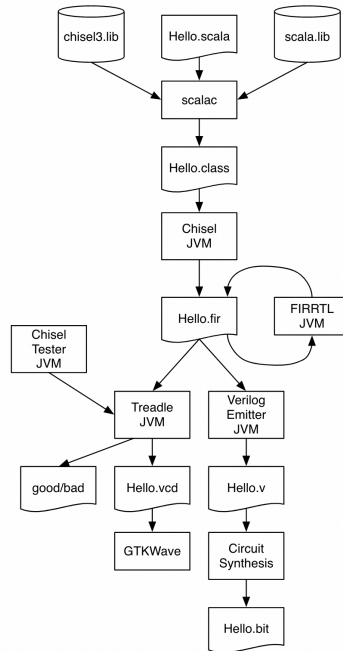


图 5.1 Chisel 工作流程<sup>[12]</sup>

#### 5.1.1 Rocket Chip

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。

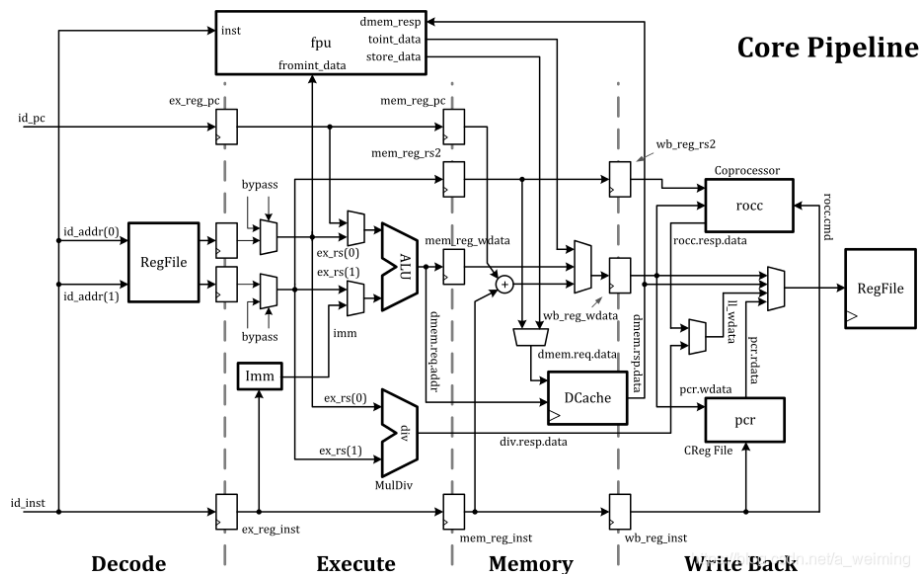


图 5.2 Rocket 处理器数据通路

### 5.1.2 RoCC

RoCC (Rocket Custom Coprocessor)<sup>[13]</sup> 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。

## 5.2 RISC-V N 扩展

参考设计规范和设计草案，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。

```

1 // 读取 uip 寄存器
2 val read_uip = read_mip & read_sideleg
3 // 委托给 U 态的中断
4 val delegateU = Bool(usingUser) && reg_mstatus.prv === PRV.U && delegate && read_sideleg(
  cause_lsbs) && cause(xLen - 1)

```



```

5 // U 态的中断
6 val u_interrupts = Mux(nmie && reg_mstatus.prv === PRV.U && reg_mstatus.uie,
    pending_interrupts & read_sideleg, UInt(0))
7 // uret 的逻辑
8 when (Bool(usingUser) && !io.rw.addr(9) && !io.rw.addr(8)) {
9     reg_mstatus.uie := reg_mstatus.upie
10    reg_mstatus.upie := true
11    ret_prv := PRV.U
12    io.evec := readEPC(reg_uepc)
13 }

```

---

## 5.3 UINTC 用户态中断控制器

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。

```

1 val device = new SimpleDevice("uintc", Seq("riscv,uintc0")) {
2     override val alwaysExtended: Boolean = true
3     override def describe(resources: ResourceBindings): Description = {
4         val Description(name, mapping) = super.describe(resources)
5         val extra = Map("interrupt-controller" -> Nil,
6             "#interrupt-cells" -> Seq(ResourceInt(1)))
7         Description(name, mapping ++ extra)
8     }
9 }

```

---

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：

```

1 val node: TLRegisterNode = TLRegisterNode(
2     address = Seq(params.address),
3     device = device,
4     beatBytes = beatBytes,
5     concurrency = 1)
6
7 // 注册 SEND 操作的写端口
8 val opRegFields = uirs.zipWithIndex.flatMap { case (x, i) =>
9     Seq(sendOffset(i) -> Seq(RegField(64, (),
10         RegWriteFn { (valid, data) =>
11             x.pending := x.pending | (valid << data(5, 0)).asUInt
12             Bool(true)
13         })))}

```

---

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

```

1 val intnode: IntNexusNode = IntNexusNode(
2     sourceFn = { _ => IntSourcePortParameters(Seq(IntSourceParameters(1,
3         Seq(Resource(device, "int"))))) },
4     sinkFn = { _ => IntSinkPortParameters(Seq(IntSinkParameters())) },
5     outputRequiresInput = false)
6 // 拉高 ipi 寄存器
7 val ipi = Seq.fill(nHarts) { RegInit(0.U) }

```

```

8     ipi.zipWithIndex.foreach { case (hart, i) =>
9         hart := uirs.map(x => x.pending != 0.U && x.active && hartId(x.hartid) == i.asUInt).
10        reduce(_ || _)
11    }
12    val (intnode_out, _) = intnode.out.unzip
13    intnode_out.zipWithIndex.foreach { case (int, i) =>
14        int(0) := ShiftRegister(ipi(i)(0), params.intStages) // usip
15    }

```

关键的是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:

```

1    // freechips.rocketchip.tile.SinksExternalInterrupts
2    def csrIntMap: List[Int] = {
3        // ...
4        val usip = if (usingUser) Seq(0) else Nil
5        List(65535, 3, 7, 11) ++ seip ++ usip ++ List.tabulate(nlips)(_ + 16)
6    }
7    // freechips.rocketchip.subsystem.CanAttachTile
8    // 将 intSinkNode 的输入连接到 Rocket 处理器
9    def decodeCoreInterrupts(core: TileInterrupts): Unit = {
10        // ...
11        val usip = if (core.usip.isDefined) Seq(core.usip.get) else Nil
12        // ...
13        val (interrupts, _) = intSinkNode.in(0)
14        (async_ips ++ periph_ips ++ seip ++ usip ++ core_ips).zip(interrupts).foreach { case (c,
15        i) => c := i }
16    }

```

## 5.4 UIPI 协处理器

根据 Rocket 处理器的数据通路 5.2，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。

```

1    // 前传逻辑，以 suirs 寄存器为例
2    when (decoded_addr(CSRs.suir)) {
3        val new_suir = new SUIRS().fromBits(wdata)
4        reg_suir := new_suir
5        io.uintr.suir := new_suir
6    }.otherwise {
7        io.uintr.suir := reg_suir
8    }

```

在配置中添加 UIPI 协处理器:

```

1    class WithUIPI extends Config((_, _, _) => {
2        case BuildRoCC => Seq((p: Parameters) => {
3            val module = LazyModule(new UIPI(OpcodeSet.custom3)(p))
4            module
5        })
6    })

```

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:

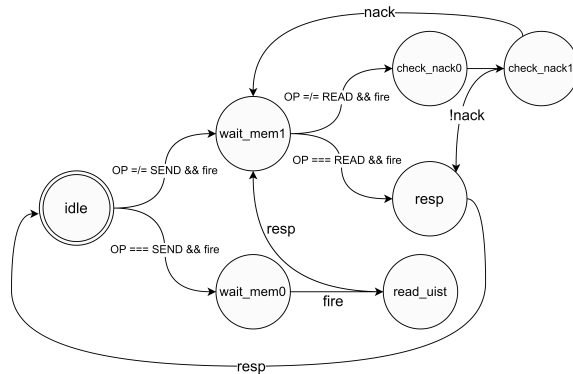


图 5.3 UIPI 协处理器工作流程

根据设计草案中对 `uipi` 指令的描述，UIPI 协处理器要发起至多两次访存请求，其中第二次一定为不经过缓存系统的读或写 UINTC 请求，由于缓存系统考虑了访问外设的请求，在实现逻辑中两次请求会依次经 RoCC 的访存端口发出。此外，对于写外设请求，数据缓存不会拉高 `resp` 信号，需要在 UIPI 中记录 `cycle` 并根据数据缓存的响应信号判断最近一次写请求是否成功，若失败则重新发送之前的请求。

`SimpleHellaCacheIFReplayQueue` 是一个缓存队列模块，用于处理来自 Rocket 处理器和 DMA 设备的读写请求，在默认的实现中被用来缓存多个 RoCC 的读写请求，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 `resp` 信号并阻塞。因为目前只有一个 UIPI 协处理器，所以在实际的实现中移除了这个队列。

## 第 6 章 软件实现

### 6.1 Linux

Linux<sup>[14]</sup> 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
- 接收方陷入内核前的状态的保存与恢复
- 注册并实现系统调用

#### 6.1.1 UINTC 驱动

在 QEMU 的 virt 硬件环境中添加设备树生成代码，最后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

驱动代码位于 `drivers/irqchip/irq-riscv-uintr.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 6.1.1 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

---

```

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  };

```

---

### 6.1.2 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。

Linux 中线程也被称为轻量级进程（LWP, Light-weight process）<sup>[15]</sup>，如无特殊说明，描述中默认使用进程指代进程或线程。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，注意到当进程主动让权时，内核会两次进入 `resume_userspace` 这个代码段，因此只能在 `restore_all` 这个代码段重新将接收方状态的 `Active` 位置 1 确保被唤醒的是真正运行在这个核上的进程。

### 6.1.3 进程管理

在 arch/riscv/kernel/uintr.c 中添加用户态中断相关代码，主要涉及以下内容：

- 控制结构定义和初始化：在进程控制块中维护发送方和接收方的状态
- 进程资源分配和回收：文件描述符，进程控制块内的状态，发送方状态表等
- 系统调用实现：在系统调用向量表中注册系统调用号，并通过 `SYSCALL_DEFINEx` 宏声明并实现系统调用函数

## 6.2 libc

### 6.2.1 库函数实现

对系统调用进一步封装，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：

---

```

1  #include "uintr.h"

```

---

```

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

## 6.2.2 应用示例

根据图 3.2 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应用。

---

```

1     volatile unsigned int uintr_received; // volatile 避免编译优化
2     unsigned int uintr_fd;
3
4     uint64_t uintr_handler(struct __uintr_frame *ui_frame, uint64_t irqs) {
5         uintr_received = 1; // 设置标志位
6         return 0;
7     }
8
9     void *sender_thread(void *arg) {
10        int uipi_index;
11        // 注册发送方状态表项
12        uipi_index = uintr_register_sender(uintr_fd);
13        // 发送用户态中断
14        uipi_send(uipi_index);
15        return NULL;
16    }
17
18    int main() {
19        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 }
```

---

接收方注册中断处理函数，注册文件描述符，创建发送方进程，并忙等待标志位；发送方进程注册后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。

## 第 7 章 系统评估

### 7.1 仿真测试

#### 7.1.1 功能级仿真测试

#### 7.1.2 电路级仿真测试

基于 `riscv-tests` 项目<sup>[16]</sup>编写关于 RISC-V 用户态中断扩展的测试程序，用来测试 UINTC 外设和 UIPI 协处理器能否正常工作。

Rocket Chip 提供了多种仿真工具和模拟器，我们采用 `verilator` 对 RTL 和测试程序进行仿真，可以在仿真结果中查看指令流，还可以查看波形来精确地了解每一个信号的状态。

### 7.2 性能测试

#### 7.2.1 功能级性能测试

#### 7.2.2 电路级性能测试



## 第 8 章 结论

## 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] Bellard F, the QEMU team. About QEMU[EB/OL]. 2023. <https://www.qemu.org/docs/master/about/index.html>.
- [10] 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>.
- [11] 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>.
- [12] Schoeberl M. Digital design with chisel 4th[M]. Kindle Direct Publishing, 2019.
- [13] 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.

- [14] Linux Foundation. Linux kernel[EB/OL]. 1991–present. <https://www.kernel.org/>.
- [15] Bovet D, Cesati M. Understanding the linux kernel 3rd.[M]. Oreilly & Associates Inc, 2005.
- [16] RISC-V International. riscv-tests[EB/OL]. 2021. <https://github.com/riscv/riscv-tests>.

## 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

### 书面翻译题目

### 目录

本科生的外文资料书面翻译。