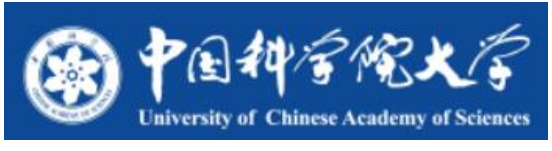
ISCAS
中国科学院软件研究所
Institute of Software,Chinese Academy of Sciences
中国科学院大学
University of Chinese Academy of Sciences

中断、异常与系统调用

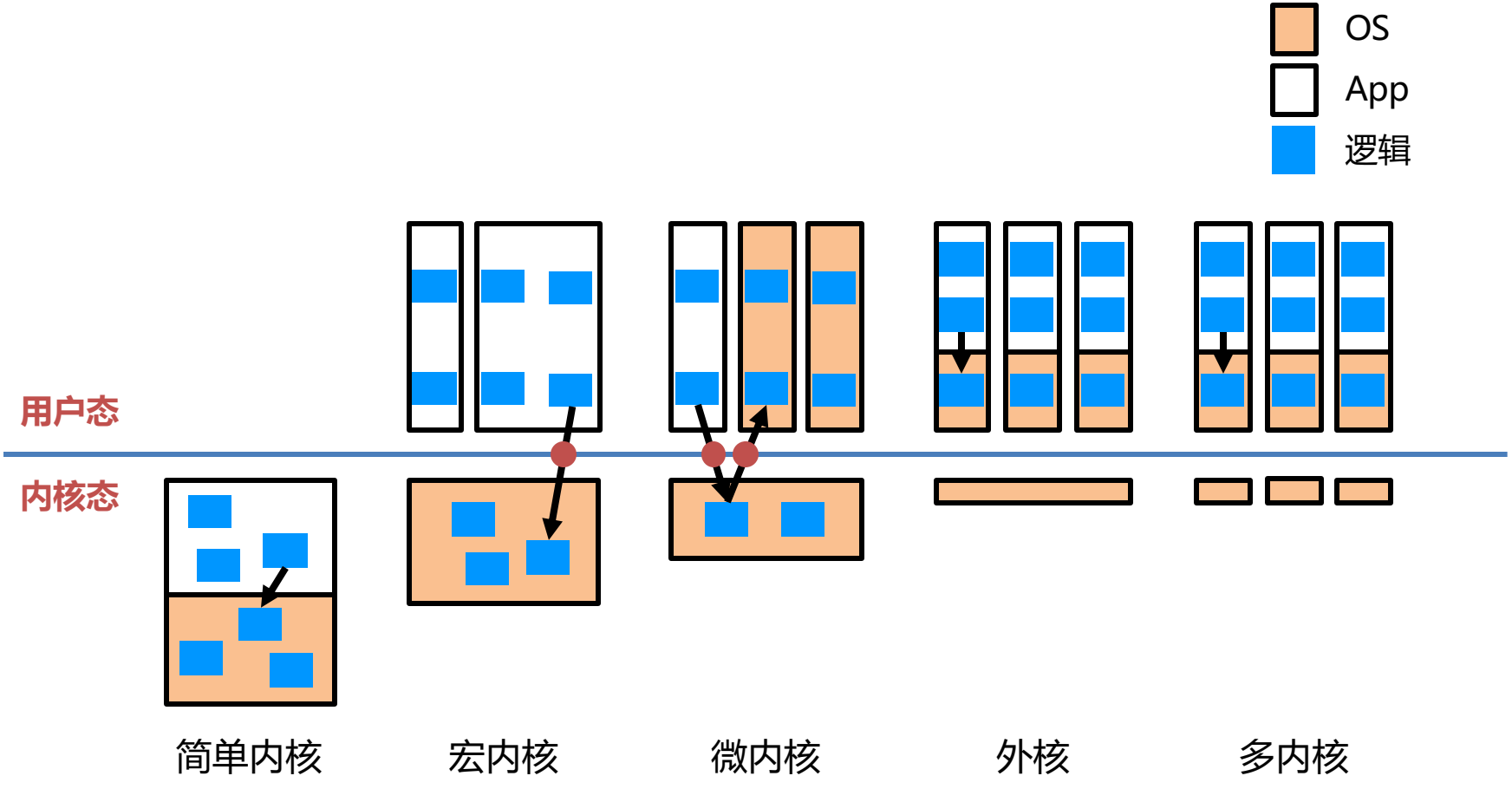
授课教师姓名

改编声明

- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，原课程官网：
  - https://ipads.se.sjtu.edu.cn/courses/os/index.shtml
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。

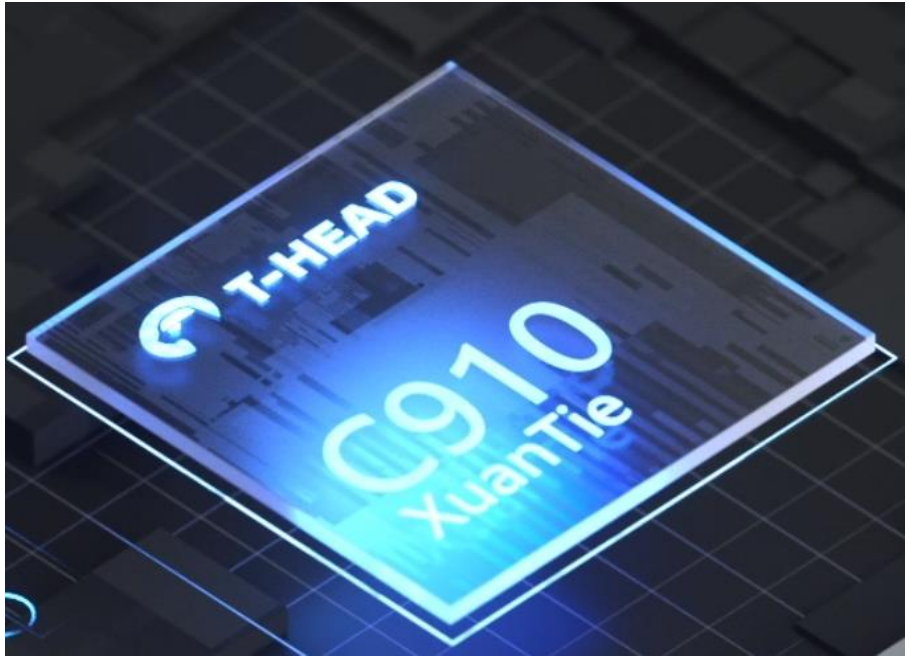


Review: 不同操作系统架构的对比



RISC-V简介

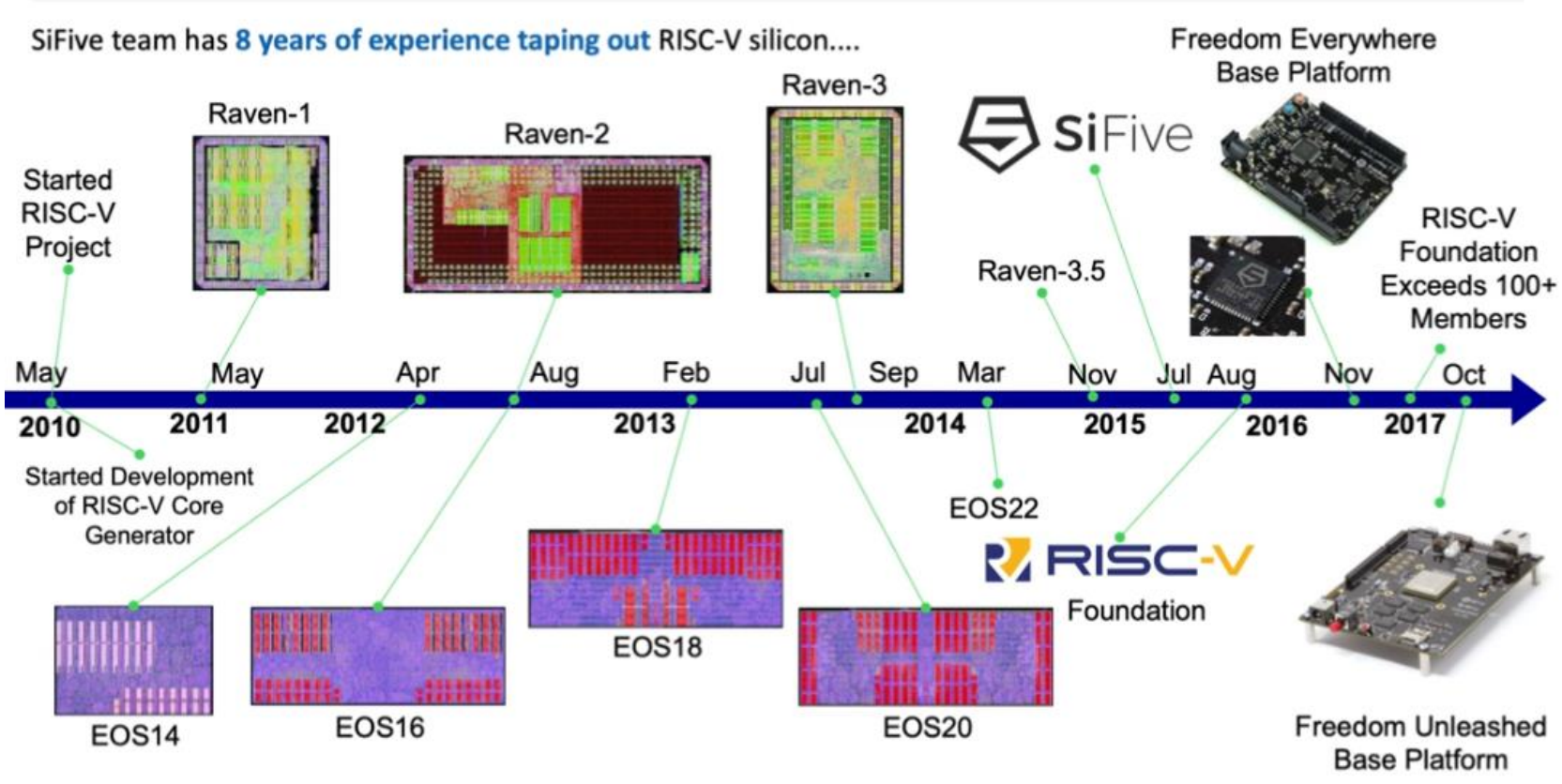
RISC-V: 先进的开源指令集



玄铁C910

兆易创新GD32 RISC-V MCU

RISC-V发展

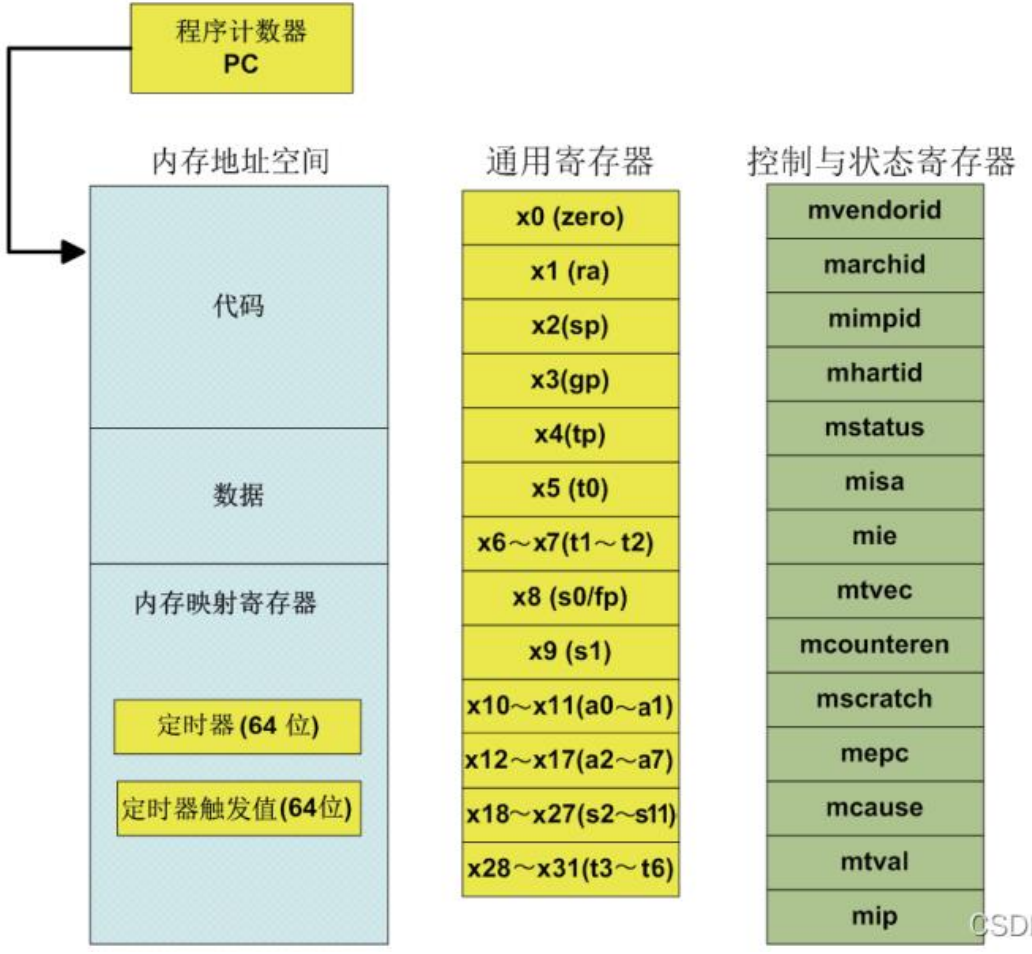

SiFive team has 8 years of experience taping out RISC-V silicon....
Started RISC-V Project
May 2010
Started Development of RISC-V Core Generator
Raven-1
May 2011
Raven-2
Apr 2012
Aug 2012
Raven-3
Feb 2013
Jul 2013
Sep 2014
Mar 2014
EOS22
Raven-3.5
Nov 2015
SiFive
Jul 2015
Freedom Everywhere Base Platform
Aug 2016
Nov 2016
RISC-V Foundation Exceeds 100+ Members
Oct 2017
RISC-V Foundation
EOS14
EOS16
EOS18
EOS20
Freedom Unleashed Base Platform

RISC-V参考文献

- 【参考 1】： The RISC-V Instruction Set Manual，Volume I: Unprivileged ISA
- 【参考 2】: The RISC-V Instruction Set Manual， Volume II: Privileged Architecture
  - https://github.com/isrc-cas/riscv-isa-manual-cn
- 【参考 3】: SiFive FU540-C000 Manual，v1p0
- 【参考 4】: RISC-V Platform-Level Interrupt Controller Specification：
  - https://github.com/riscv/riscv-plic-spec

RISC-V

- PC(64-bit)指向当前执行的指令
- 指令长度相同 (RISC, 16、32、64、128-bit)
- PC指针不占用通用寄存器，而是独立的，程序执行中自动变化，无法通过通用寄存器访问和修改PC值。



CPU寄存器 RISC-V

- **32个通用寄存器**
  - X0-X31
- **1个PC指针寄存器**

寄存器	ABI 名称	说明
x0	zero	0值寄存器，硬编码为0，写入数据忽略，读取数据为0
x1	ra	用于返回地址(return address)
x2	sp	用于栈指针（stack pointer）
x3	gp	用于通用指针 (global pointer)
x4	tp	用于线程指针 （thread pointer）
x5	t0	用于存放临时数据或者备用链接寄存器
x6~x7	t1~t2	用于存放临时数据寄存器
x8	s0/fp	需要保存的寄存器或者帧指针寄存器
x9	s1	需要保存的寄存器
x10~x11	a0~a1	函数传递参数寄存器或者函数返回值寄存器
x12~x17	a2~a7	函数传递参数寄存器
x18~x27	s2-s11	需要保存的寄存器
x28~x31	t3~t6	用于存放临时数据寄存器

寄存器 RISC-V vs X86-64

- **32个通用寄存器**
  - X0-X31
  - 1个只读零值寄存器x0
  - 1个返回地址寄存器
  - 1个堆栈指针寄存器
  - 1个全局指针寄存器
  - 1个线程指针寄存器
  - 7个临时寄存器
  - 12个存保（saved）寄存器
  - 8个函数参数寄存器
- **1个PC寄存器**
- **14个CSR控制和状态寄存器**

思考：X86架构中，切换特权级时rsp是如何保存，以及如何恢复的?

X86-64

16个通用寄存器

1个%rip寄存器

1个%rsp寄存器

返回地址压栈

EFLAGS

ISA

- **RISC**
  - 固定长度指令格式
  - 更多的通用寄存器
  - Load/store 结构
  - 简化寻址方式

```
0000000000000000 <bar>:
   0:   81010113                addi    sp,sp,-2032
   4:   7e113423                sd      ra,2024(sp)
   8:   7e813023                sd      s0,2016(sp)
   c:   7f010413                addi    s0,sp,2032
  10:   fff00293                 li      t0,-1
  14:   02029293                 slli    t0,t0,0x20
  18:   7f028293                 addi    t0,t0,2032
  1c:   00510133                 add     sp,sp,t0
  20:   00000097                 auipc   ra,0x0
                        20: R_RISCV_CALL        foo
  24:   000080e7                 jalr    ra # 20 <bar+0x20>
  28:   fea43423                 sd      a0,-24(s0)
  2c:   000007b7                 lui     a5,0x0
```

RISC vs CISC

	RISC (RISC-V)	CISC (x86-64)
指令长度	定长	变长
寻址模式	寻址方式单一	多种寻址方式
内存操作	load/store	mov
实现	增加通用寄存器数量	微码
指令复杂度	简单	复杂
汇编复杂度	复杂	简单
中断响应	快	慢
功耗	低	高
处理器结构	简单	复杂

基地址加偏移量模式

- 引用M[r_b, Offset]处的数据
- 基地址寄存器r_b
- RISC-V中偏移量Offset通常是一个立即数

如： lw x1, 100(x2) # 加载寄存器 x1 中的值，该值位于基地址寄存器 x2 加上立即数 100 的内存位置

如果需要使用寄存器中的值作为偏移量，加载或计算偏移量，然后将其与基地址相加。

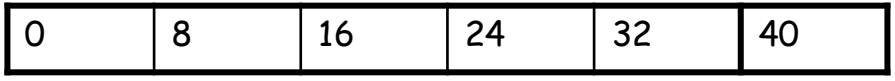
如：add x4, x2, x3 # 计算偏移量

lw x1, 0(x4) # 使用计算得到的偏移量加载数据到 x1

加载寄存器 x1 中的值，该值位于基地址寄存器 x2 加上偏移量寄存器 x3 的内存位置

示例：基地址加偏移量模式

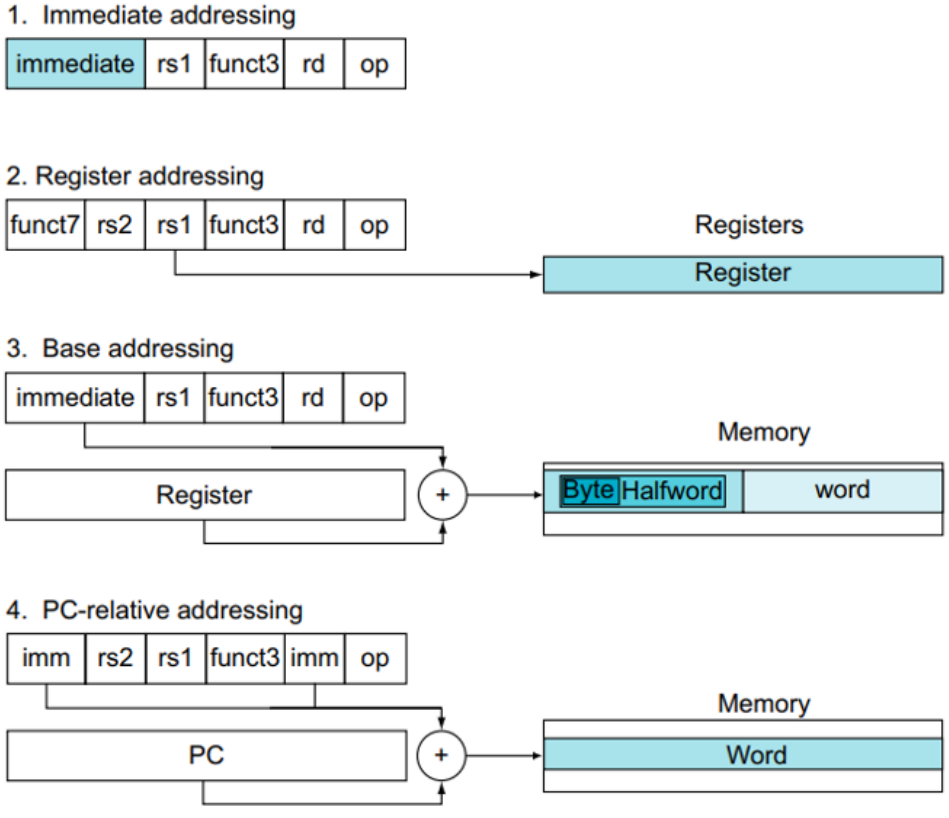
```
long E[6] ;
```



- **假设E是一个整型数组**
  - E的起始地址存放在s2寄存器中
- **访问数组元素E[1]的RISC-V汇编为：**
  - **lb s1,8(s2)**

RISC-V寻址模式小结

1. 立即数寻址
   - 操作数是操作本身的常量
2. 寄存器寻址
   - 操作数在寄存器
3. 基址寻址
   - 操作数于内存中，其地址是寄存器和指令中的常量之和
4. PC 相对寻址
   - 分支地址是 PC和指令中常量之和



函数调用指令（caller调用callee）

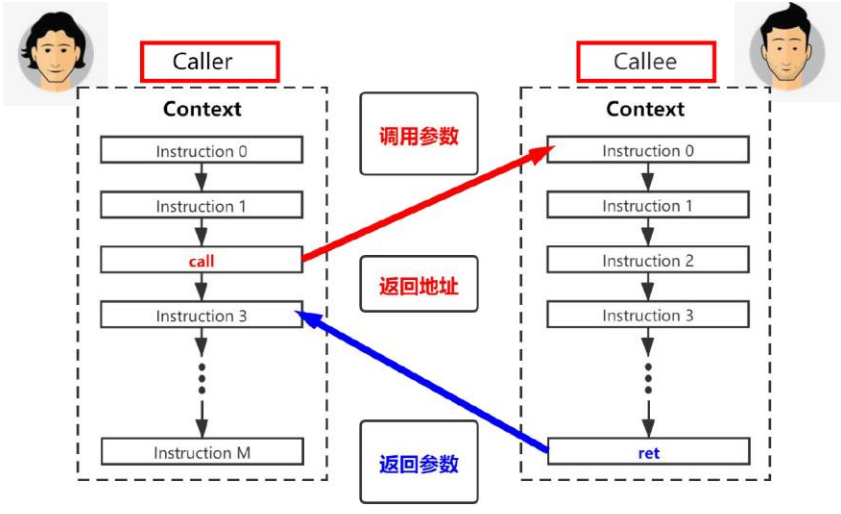
指令

– `jal label`（直接调用，调用函数）

功能

– 将**返回地址(PC+4)**存储在**寄存器或堆栈**

– 跳转到被调用者的**入口地址**



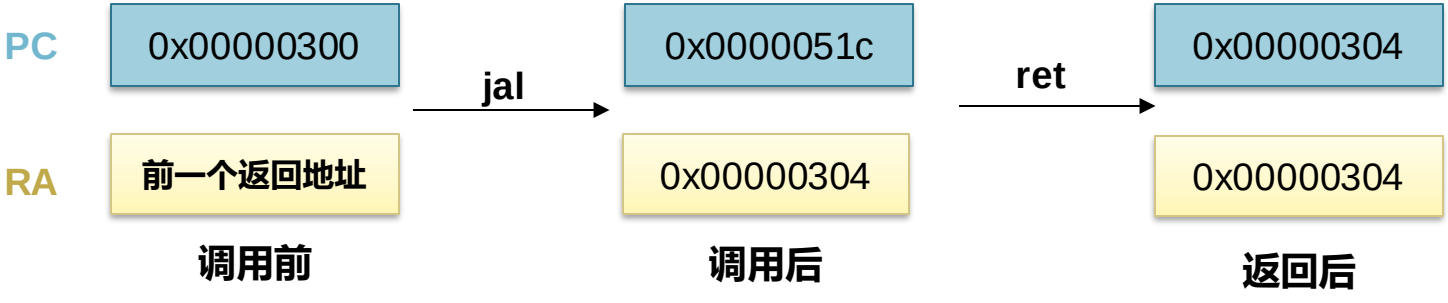
示例：PC与RA的变化

High-Level Code

```
int main() {
        simple( );
        。 。 。
}

// void means the function
// returns no value
void simple(){
        return;
}
```

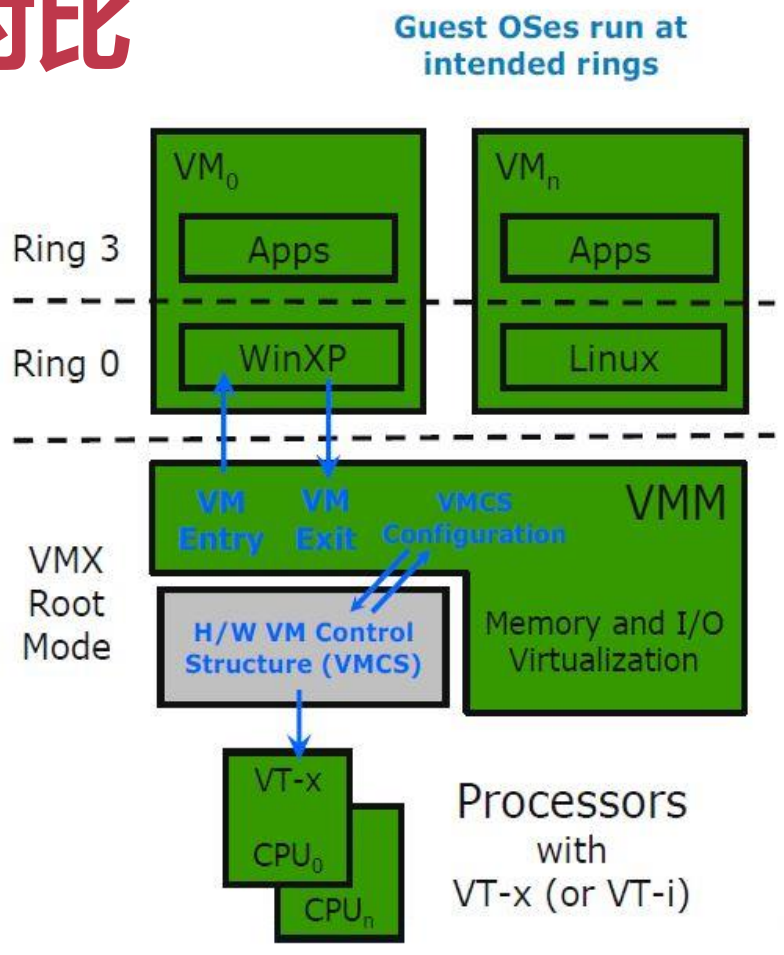
RISC-V Assembly Code

```
0x00000300  main: jal simple  #call function
0x00000304  ...
...

0x0000051c  simple: jr ra  #return
```



特权级：与X86-64对比

- **Non-root：**
  - Ring 3： Guest app
  - Ring 0： Guest OS
- **Root：**
  - Ring 3： App
  - Ring 0： Hypervisor



特权级/RISC-V (Exception Level)

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

表 1.1：RISC-V 特权级

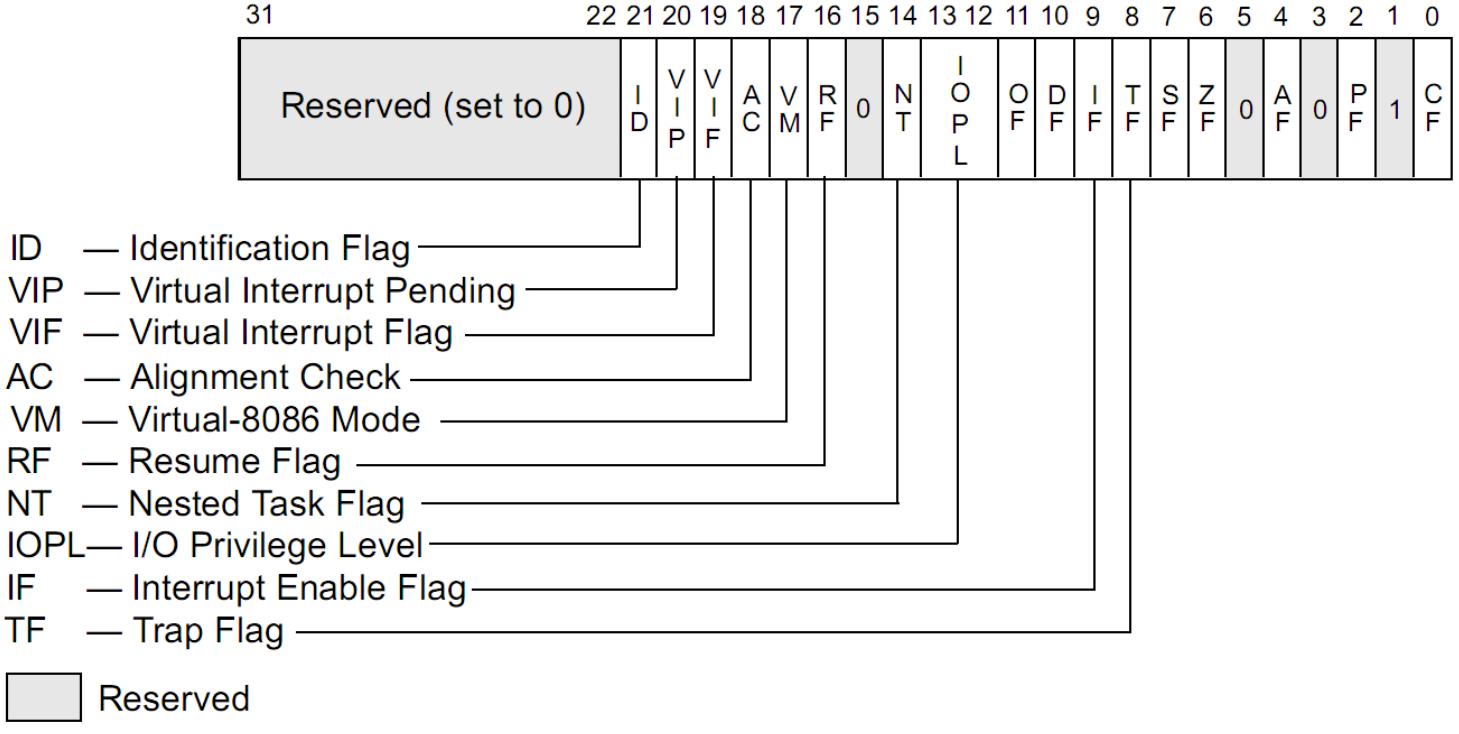
未来 hypervisor 级 ISA 扩展将会被加入

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

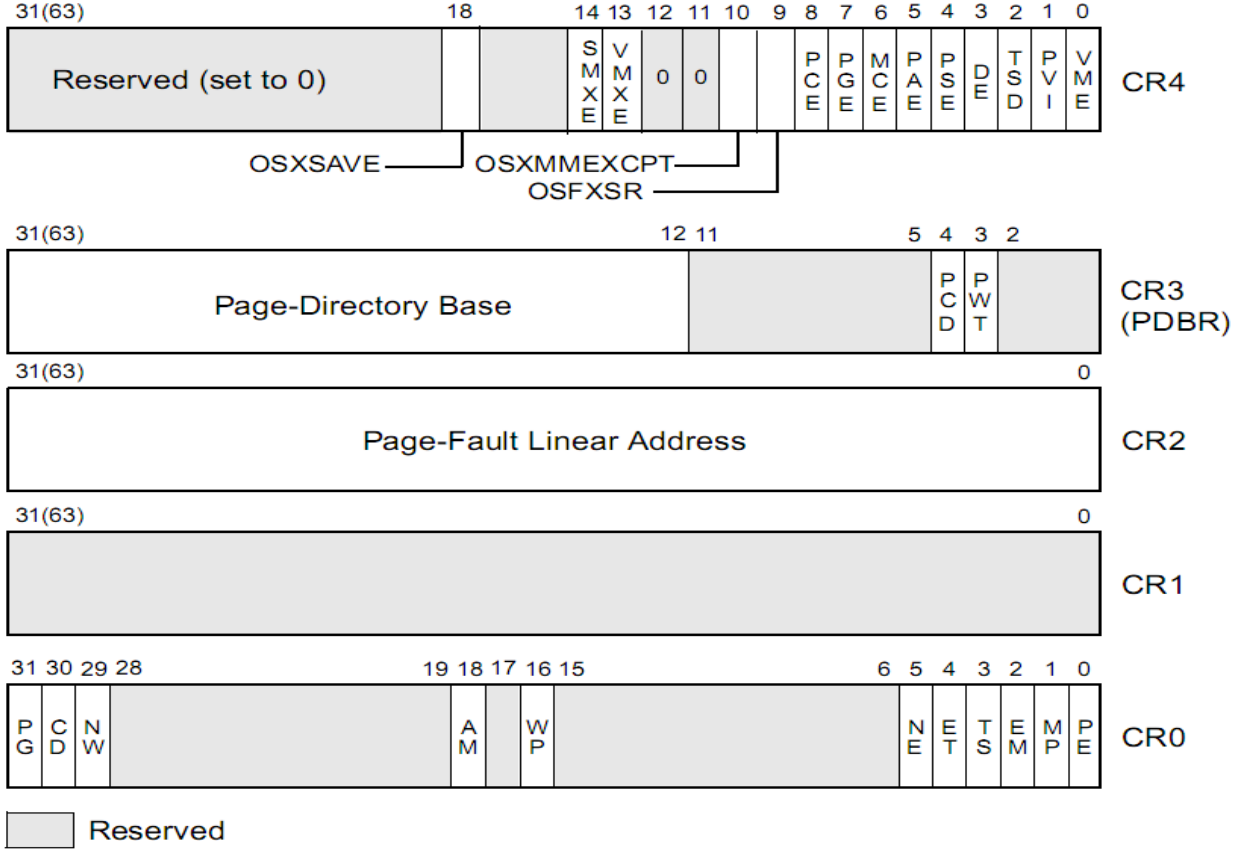
Table 1.2: Supported combinations of privilege modes.

系统状态寄存器：X86-64

- System Flags in the EFLAGS



系统控制寄存器：X86-64



控制状态寄存器：RISC-V

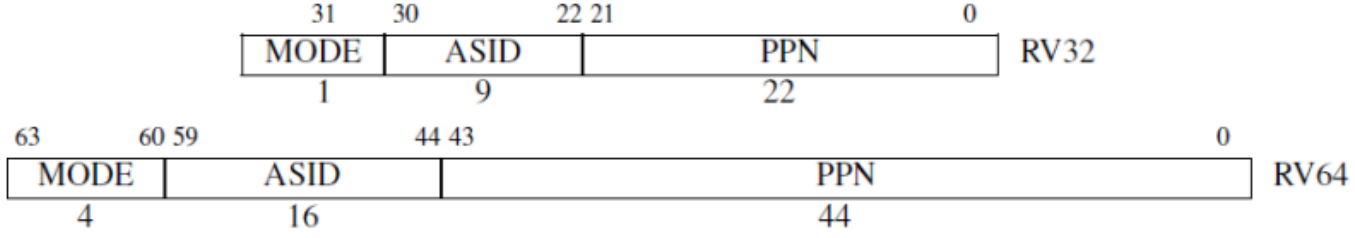
在RISC-V架构中，并没有像传统的处理器架构（如x86或ARM）中的专门的"状态寄存器"，而是通过一组寄存器来实现状态管理和处理器状态的保存。

对于系统的异常处理有8个控制状态寄存器(CSR)。

寄存器	功能
mtvec (Machine Trap Vector)	它保存发生异常时处理器需要跳转到的地址。
mepc (Machine Exception PC)	它指向发生异常的指令。
mcause (Machine Exception Cause)	它指示发生异常的种类。
mie (Machine Interrupt Emable)	它指出处理器目前能处理和必须忽略的中断。
mip (Machine Interrupt Pending)	它列出目前正准备处理的中断。
mtval (Machine Trap Value)	它保存了陷入(trap)的附加信息:地址例外中出错的地址、发生非法指令例外的指令本身，对于其他异常，它的值为0。
mscratch (Machine Scratch)	它暂时存放一-个字大小的数据。
mstatus (Machine Status)	它保存全局中断使能，以及许多其他的状态。
mie (Machine Interrupt Enable)	用于进一步控制（打开和关闭）software interrupt/timer interrupt/external interrupt
mip (Machine Interrupt Pending)	它列出目前已发生等待处理的中断。

内存系统相关寄存器：RISC-V

- satp（Supervisor Address Translation and Protection，监管者地址转换和保护）



图：satp CSR

satp（Supervisor Address Translation and Protection，监管者地址转换和保护）的 S 模式控制状态寄存器控制了分页系统

satp 有三个域。MODE 域可以开启分页并选择页表级数。

ASID（Address Space Identifier，地址空间标识符）域是可选的，它可以用来降低上下文切换的开销。

PPN 字段保存了根页表的物理地址，它以 4 KiB 的页面大小为单位。

对比X86-64:CR3寄存器

地址翻译

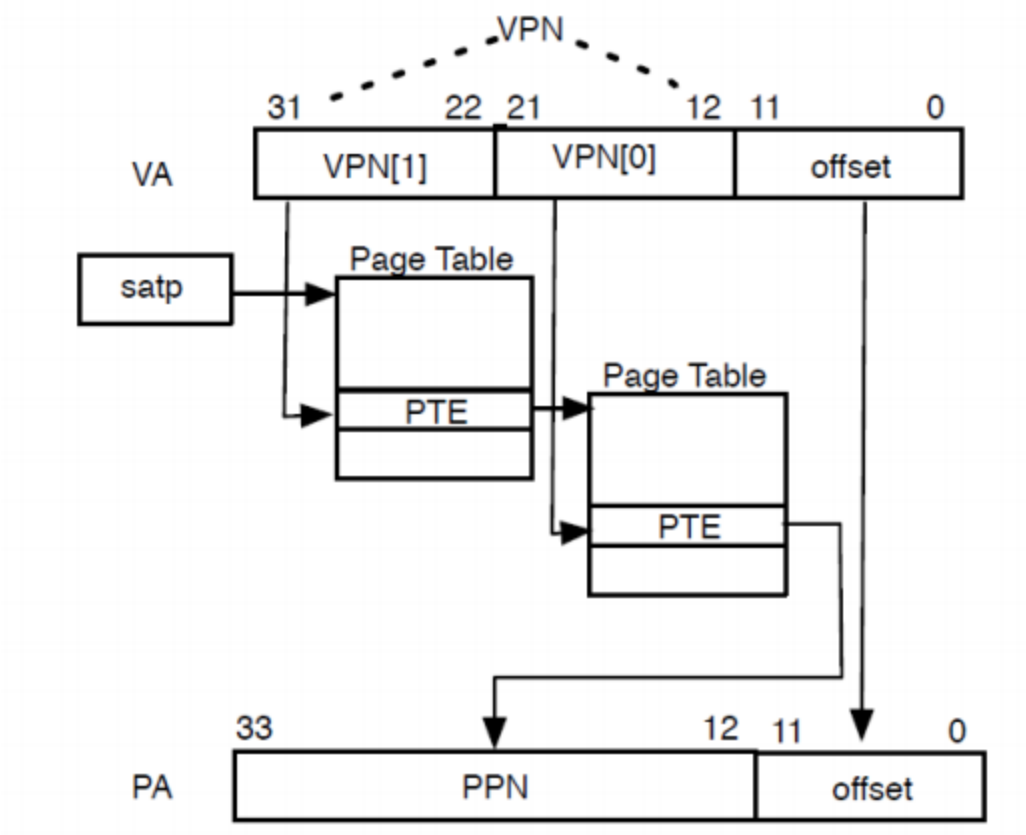


图 10.14：Sv32 中地址转换过程的图示。

RISC-V Linux Kernel SV39虚拟内存布局

开始地址	偏移	结束地址	大小	虚拟内存区域描述
0000000000000000	0	0000003fffffffff	256 GB	用户空间虚拟内存，每个内存管理器不同
0000004000000000	+256 GB	ffffffbfffffffff	~16M TB	... 巨大的、几乎64位宽的直到内核映射的-256G开始偏移的非经典虚拟内存地址空洞。
				内核空间的虚拟内存，在所有进程之间共享:
ffffffc6fee00000	-228 GB	ffffffc6feffffff	2 MB	fixmap
ffffffc6ff000000	-228 GB	ffffffc6ffffffff	16 MB	PCI io
ffffffc700000000	-228 GB	ffffffc7ffffffff	4 GB	vmemmap
ffffffc800000000	-224 GB	ffffffd7ffffffff	64 GB	vmalloc/ioremap space
ffffffd800000000	-160 GB	fffffff6ffffffff	124 GB	直接映射所有物理内存
fffffff700000000	-36 GB	fffffffeffffffff	32 GB	kasan
ffffffff00000000	-4 GB	ffffffff7fffffff	2 GB	modules, BPF
ffffffff80000000	-2 GB	ffffffffffffffff	2 GB	kernel

RISC-V 输入/输出

```c
unsigned int early_uart_recv(void)
{
    while (1) {
        if (early_uart_lsr() & 0x01)
            break;
    }
    return early_get32(AUX_MU_IO_REG) & 0xFF;
}

void early_uart_send(unsigned int c)
{
    while (1) {
        if (early_uart_lsr() & 0x20)
            break;
    }
    early_put32(AUX_MU_IO_REG, c);
}
```

```
BEGIN_FUNC(early_get32)
    lw t3, [s0]
    ret
END_FUNC(early_get32)
```

- **MMIO: 复用lw和sw指令**
  - 映射到物理内存的特殊地址段

```
BEGIN_FUNC(early_get32)
    sw t4, [s0]
    ret
END FUNC(early get32)
```

MMIO与PIO

- **MMIO (Memory-mapped IO)**
  - 将设备映射到连续的物理内存中，使用相同的指令
  - 如，Raspi3映射到0x3F200000
  - 行为与内存不完全一样，读写会有副作用（回忆volatile）
- **PIO (Port IO)**
  - IO设备具有独立的地址空间
  - 使用特殊的指令（如x86中的in/out指令，RISC-V的Claim/Complete）

从键盘看I/O

问题:应用如何通过OS和外设交互的?

- 简单场景:应用是如何获得键盘输入?

OS接受键盘输入

- **键盘等外设具有控制器和缓冲区，将输入存入缓冲区**
- **OS获取该输入的可能方法 ……**
  - 轮询：OS不断去读该缓冲区中的值
  - 中断：当控制器接收到输入后，打断CPU正常执行，OS进行处理

思考：就键盘而言，选用哪种方式更好？为什么？

思考：在哪种场景下，使用“轮询”的效果更好？

中断与轮询

- Cpu速度数量级倍高于I/O
- 只有当I/O就绪时才能进行执行处理

- 周期性检查I/O状态
- 不管I/O是否就绪

- 中断事件发生时，异步通知系统
  - 设备需要马上得到CPU响应，即便CPU处于繁忙状态

应用从OS获取输入

- **为了给应用屏蔽硬件细节、管理硬件，应用一般不直接操控设备**
- **`getchar()`中调用`read()`请求OS将获取的输入返回给用户**
- **应用能否绕过OS直接控制硬件?**
  - 用户态驱动 (但是也需要操作系统实现配置)

通用概念

- **中断（Interrupt）——异步异常**
  - 外部硬件设备所产生的信号
  - 异步：产生原因和当前执行指令无关，如程序被磁盘读打断
- **异常（Exception）——同步异常**
  - 软件的程序执行而产生的事件
  - 包括系统调用（System Call）
    - 用户程序请求操作系统提供服务
  - 同步：产生和当前执行或试图执行的指令相关

Faults

- 指令无法合法正常执行，在执行时可以被检测到

- 示例:
  – 对只读的数据段进行写操作
  – 对不存在的地址空间进行读操作
  – 在用户态执行特权指令

- fault可以被修复，之后可以正常进行
  – 操作系统中有专门的处理fault的软件模块 – 如:缺页处理

Traps

- **Trap是用用于调试等目的专用指令**

- **通过接管trap指令异常，CPU会自动进入调试程序**
  - 通过接管trap指令异常，CPU会自动进入调试程序
  - GDB断点的实现原理

- **Trap指令返回时，程序计数器PC值加1**
  - – 使其能够正常执行后面的程序

Error Exception

- **大部分的错误**
  - – 除零
  - – 非法指令

- **OS的处理**
  - – 简单，给对应进程发送信号，如杀死进程
  - – Force_sig(sig_number, current)

- **内核出现异常时**
  - – die(); //kernel oops

异常的分类

Fault	可纠正的异常，是程序执行的一种被动行为
Trap	可被捕获的陷阱，程序员或编译器的主动行为
Error	错误，一般伴随着进程的终结

异常处理过程

- **强制改变正常的执行流**

- **与进程上下文切换类似，但更轻量级**
  - – 硬件保存上下文状态
  - – 进入内核中的对应入口
    - » 内核判断应该运行哪个中断处理函数
  - – 完成后继续执行
    - » 专用的iret/eret等返回指令

中断与异常比较

类似点

- 处理过程类似，都涉及保存现场
- 有专用事先设计好的处理例程

差异点

- 触发方式
  - 中断是异步的，由外部设备的事件触发
    - 示例：如敲击键盘、网卡收包
  - 异常是同步的，由所执行指令触发
    - 示例：如访存指令地址异常
- 与进程关系
  - 中断与被中断的指令及进程无直接关联，逮到谁是谁
  - 异常关联被中断的的指令与进程，异常的处理可能会阻塞或杀死本进程

不同体系结构术语的对应关系

通用概念	产生原因	RISC-V		x86-64
中断	硬件异步	异常	异步异常（中断）	中断（可屏蔽/不可屏蔽）
异常	软件同步		同步异常	异常（Fault/Trap/Abort）

- **之后提到的“中断”、“异常”均为通用概念意义**

x86-64术语

- **中断（设备产生、异步）**
  - 可屏蔽：设备产生的信号，通过中断控制器与处理器相连，可被暂时屏蔽（如，键盘、 网络事件）
  - 不可屏蔽： 一些关键硬件的崩溃（如，内存校验错误）
- **异常（软件产生、同步）**
  - **错误（Fault）：**如缺页异常（可恢复）、段错误（不可恢复）等
  - **陷阱（Trap）：**无需恢复，如断点（int 3）、系统调用（int 80）
  - **中止（Abort）：**严重的错误，不可恢复（机器检查）

异常处理—RISC-V

CPU的执行逻辑

- **CPU的执行逻辑很简单**
  1. 以PC的值为地址从内存中获取一条指令并执行
  2. PC+=4，goto 1
- **执行过程中可能发生两种情况**
  1. 指令执行出现错误，比如除零或缺页（同步异常）
  2. 外部设备触发中断（异步异常）
- **这两种情况在RISC-V平台称为「异常」和「中断」**
  - 陷入指由一个异常或中断引发的将控制权转移到陷入处理程序的过程
  - 陷入处理完后，执行流需要恢复到之前被打断的地方继续运行

控制流（Control Flow）和 Trap

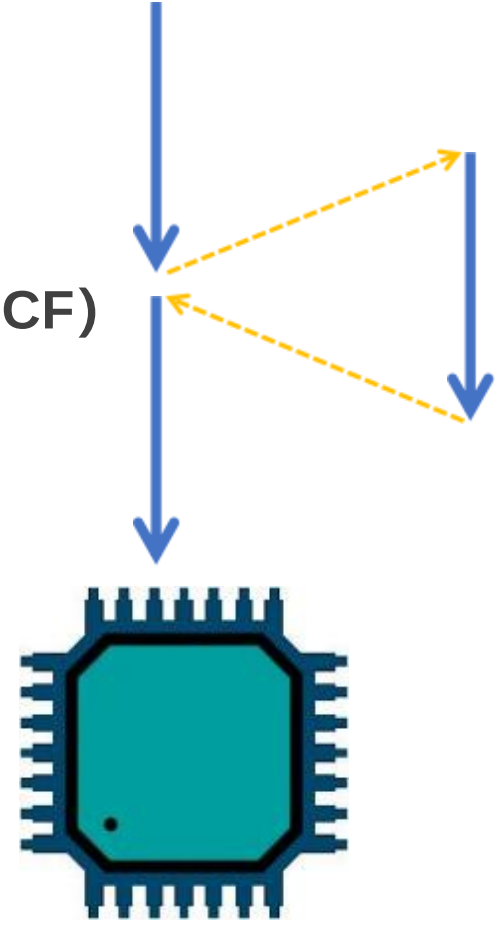
- **控制流（Control Flow）**

  Branch、Jump

- **异常控制流(Exceptional Control Flow， ECF)**

  1. exception
  2. interrupt

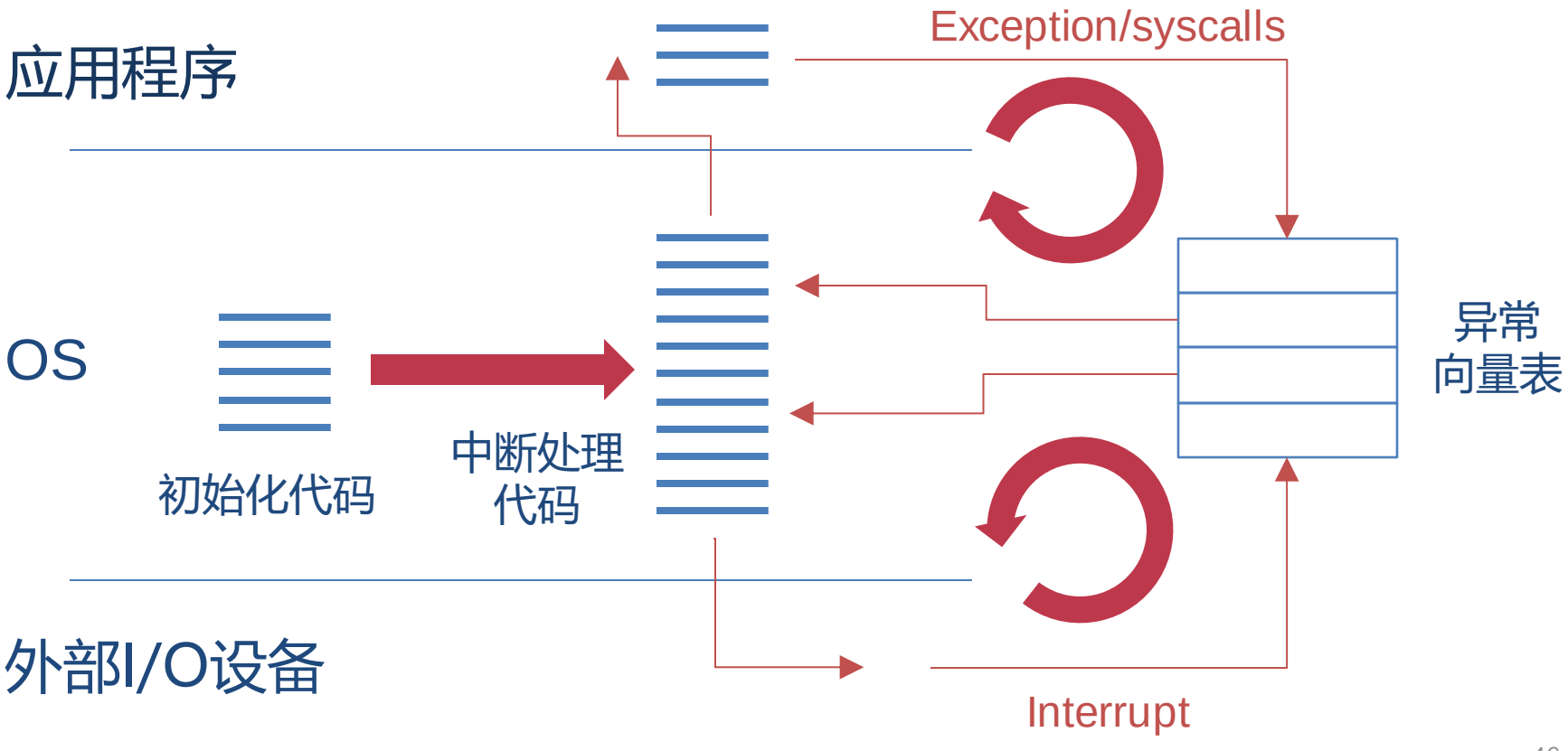
- **RISC-V 把 ECF 统称为 Trap**



操作系统的执行流可对应地分为两部分

- **一、实现对异常向量表的设置**
  - CPU上电后立即执行，是系统初始化的主要工作之一
  - 在开启中断和启动第一个应用之前
- **二、实现对不同异常（中断）的处理函数**
  - 处理应用程序出错的情况：如除零、缺页
    - Q：内核如果自己运行出错怎么办?
  - 一类特殊的同步异常：系统调用，由应用主动触发
  - 处理来自外部设备的中断：如收取网络包、获取键盘输入等

操作系统的执行流（简化版）



RISC-V Trap 处理中涉及的寄存器

- **Machine 模式下的 CSR 列表**

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	MRO	mvendorid	Vendor ID.
0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
Machine Trap Setup			
0x300	MRW	mstatus	Machine status register.
0x301	MRW	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
0x306	MRW	mcounteren	Machine counter enable.
Machine Trap Handling			
0x340	MRW	mscratch	Scratch register for machine trap handlers.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mtval	Machine bad address or instruction.
0x344	MRW	mip	Machine interrupt pending.
Machine Memory Protection			
0x3A0	MRW	pmpcfg0	Physical memory protection configuration.
		⋮	
0x3BF	MRW	pmpaddr15	Physical memory protection address register.

Table 2.4: Currently allocated RISC-V machine-level CSR addresses.

RISC-V Trap 处理中涉及的寄存器

寄存器	用途说明
mtvec（Machine Trap-Vector Base-Address)	它保存发生异常时处理器需要跳转到的地址。
mepc（Machine Exception Program Counter)	当 trap 发生时， hart 会将发生 trap 所对应的指令的地址值（pc）保存在 mepc 中。
mcause（Machine Cause)	当 trap 发生时， hart 会设置该寄存器通知我们 trap 发生的原因。
mtval（Machine Trap Value)	它保存了 exception 发生时的附加信息： 譬如访问地址出错时的地址信息 、或者执行非法指令时的指令本身， 对于其他异常， 它的值为 0。
mstatus（Machine Status)	用于跟踪和控制 hart 的当前操作状态（特别地， 包括关闭和打开全局中断）。
mscratch（Machine Scratch)	Machine 模式下专用寄存器， 我们可以自己定义其用法， 譬如用该寄存器保存当前在 hart 上运行的 task 的上下文（context）的地址。
mie（Machine Interrupt Enable)	用于进一步控制（打开和关闭）software interrupt/timer interrupt/external interrupt
mip（Machine Interrupt Pending)	它列出目前已发生等待处理的中断。

mtvec（Machine Trap-Vector Base-Address）

Write Any Values, Reads Legal Values (WARL)
【 参考2】 2.3 CSR Field Specifications

Figure 3.8: Machine trap-ve

BASE：trap 入口函数的基地址，必须保证四字节对齐。

MODE：进一步用于控制入口函数的地址配置方式：

- Direct：所有的 exception 和 interrupt 发生后 PC 都跳转到 BASE 指定的地址处。
- Vectored：exception 处理方式同 Direct；但 interrupt 的入口地址以数组方式排列。

Value	Name	Description
0	Direct	All exceptions set pc to BASE.
1	Vectored	Asynchronous interrupts set pc to BASE+4×cause.
≥2	—	*Reserved*

【参考 2】Table 3.5: Encoding of mtvec MODE field.

mepc (Machine Exception Program Counter)

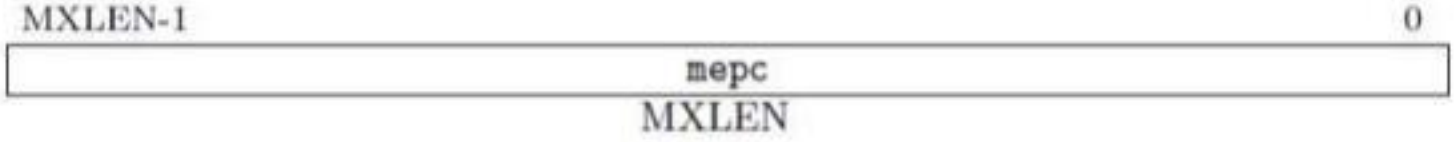
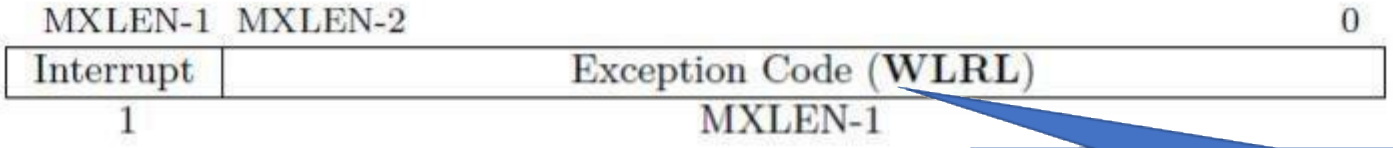


Figure 3.21: Machine exception program counter register.

- 当trap发生时，pc会被替换为mtvec设定的地址，同时hart会设置mepc为当前指令或者下一条指令的地址，当我们需要退出trap时可以调用特殊的mret指令，该指令会将mepc中的值恢复到pc中(实现返回的效果)。
- 在处理trap的程序中我们可以修改mepc的值达到改变mret返回地址的目的。

mcause（Machine Cause)



【参考 2】Figure 3.22: Machine Cause registe

Write/Read Only Legal Values (WLRL)
【 参考2】 2.3 CSR Field Specifications

- 当 trap 发生时，hart会设置该寄存器通知我们 trap 发生 的原因。
- 最高位Interrupt为1时标识了当前trap为interrupt，否则是exception。
- 剩余的 Exception Code 用于标识具体的 interrupt 或者 exception 的种类。

mcause（Machine Cause)

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved for future standard use*
1	11	Machine external interrupt
1	12–15	*Reserved for future standard use*
1	≥16	*Reserved for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

中断

异常

【 参考 2】 Table 3.6: Machine cause register (mcause) values after trap.

mtval（Machine Trap Value)

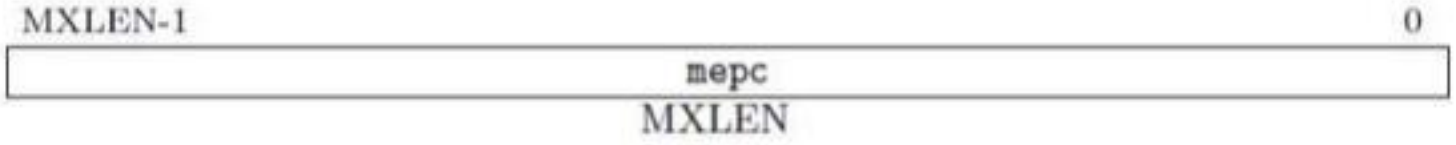


Figure 3.21: Machine exception program counter register.

- 当trap发生时，除了通过mcause可以获取exception的种类code值外，hart还提供了mtval来提供exception的其他信息来辅助我们执行更进一步的操作。
- 具体的辅助信息由特定的硬件实现定义，RISC-V规范没有定义具体的值。但规范定义了一些行为，譬如访问地址出错时的地址信息、或者执行非法指令时的指令本身等，具体阅读【参考2】。

mstatus（Machine Status）

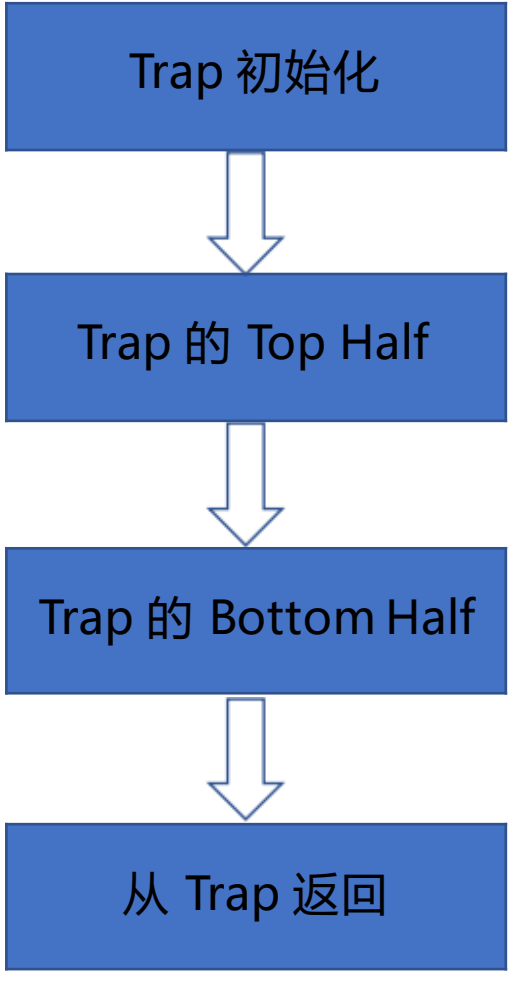
Figure 3.6: Machine

- **xIE(x=M/S/U):分别用于打开(1)或者关闭(0)M/S/U模式下的全局中断。当trap发生时，hart会自动将xIE设置为0。**
- **xPIE(x=M/S/U):当 trap 发生时用于保存trap发生之前的 xIE 值。**
- **xPP(x=M/S):当trap发生时用于保存trap发生之前的权限级别值。注意没有UPP。**
- **其他标志位涉及内存访问权限、虚拟内存控制等，暂不考虑。**

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

Table 1.1: RISC-V privilege levels.

RISC-V Trap 处理流程



异常处理函数

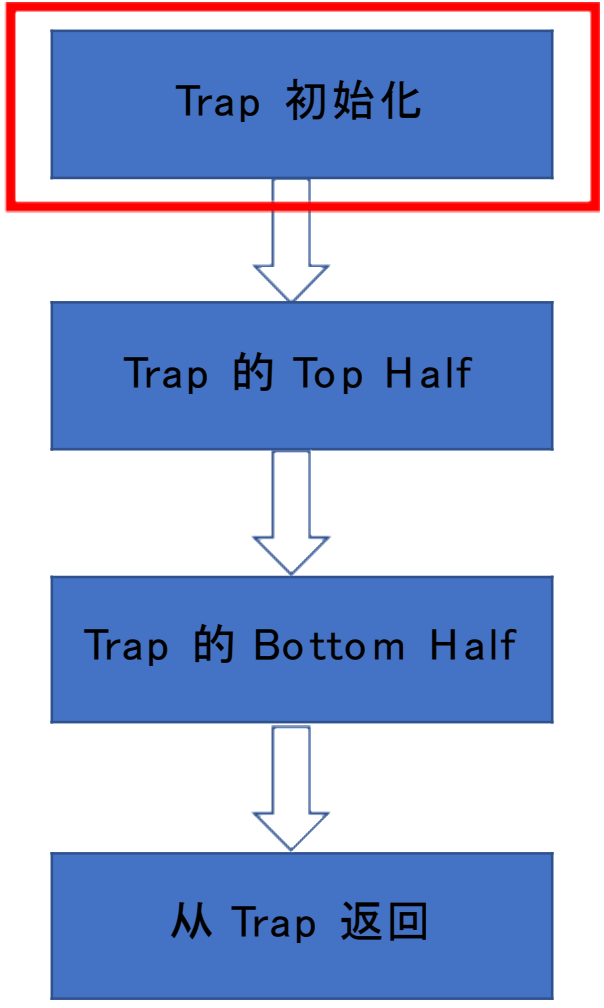
- **异常处理函数运行在内核态**
  - 可以不受限制地访问所有资源

- **处理器将异常类型存储在指定寄存器中 (cause)**
  - 表明发生的是哪一种异常
  - 异常处理函数根据异常类型执行不同逻辑

异常处理函数

- **当异常处理函数完成异常处理后，将通过下述操作之一转移控制权：**
  - 回到发生异常时正在执行的指令
  - 回到发生异常时的下一条指令
  - 结束当前进程

RISC-V Trap 处理流程

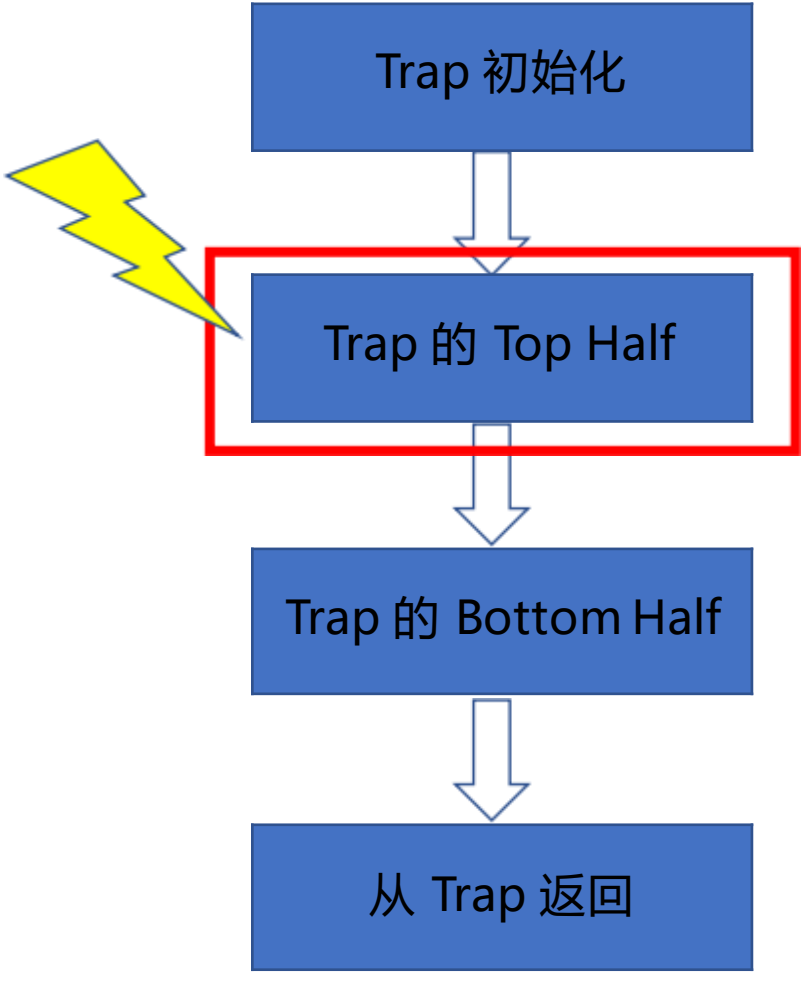


Trap的初始化

```
void trap_init()
{
        /*
         * set the trap-vector base-address for machine-mode
         */
        w_mtvec((reg_t)trap_vector);
}
```

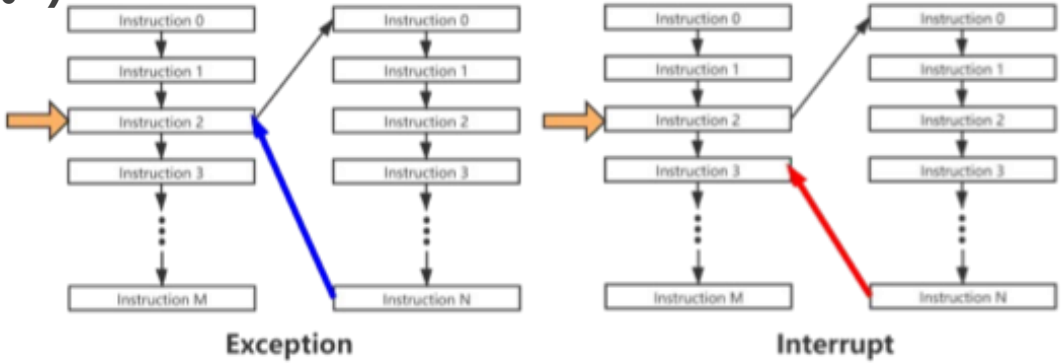
```
trap_vector:
        # save context(registers).
        csrrw   t6, mscratch, t6        # swap t6 and mscratch
        reg_save t6

        # Save the actual t6 register, which we swapped into
        # mscratch
        mv      t5, t6          # t5 points to the context of current task
        csrr    t6, mscratch    # read t6 back from mscratch
        sw      t6, 120(t5)     # save t6 with t5 as base
```

RISC-V Trap 处理流程



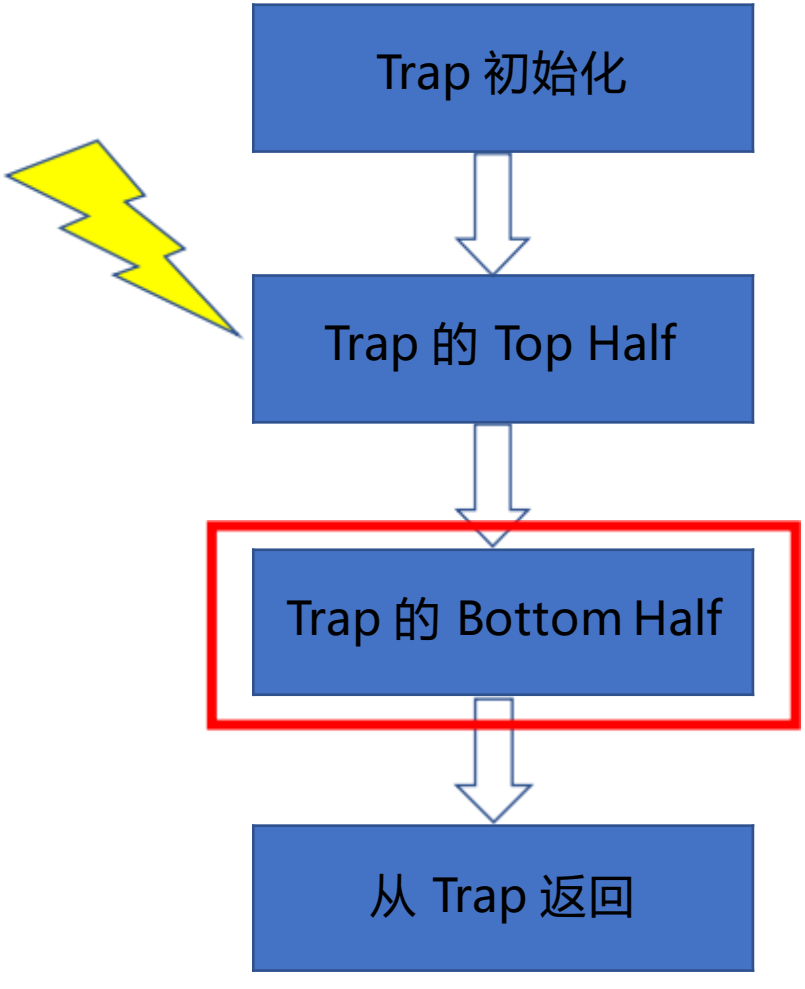
Trap发生，Hart自动执行如下状态转换

- 把 mstatus 的 MIE 值复制到 MPIE 中，清除mstatus中的MIE标志位，效果是中断被禁止。
- 设置mepc，同时PC被设置为mtvec。(需要注意的是，对于exception，mepc 指向导致异常的指令；对于 interrupt，它指向被中断的指令的下一条指令的位置。)



- 根据 trap 的种类设置mcause，并根据需要为 mtval 设置附加信息。
- 将trap发生之前的权限模式保存在mstatus的MPP域中，再把hart权限模式更改为M(也就是说无论在任何Level下触发trap，hart首先切换到Machine模式)。

RISC-V Trap 处理流程

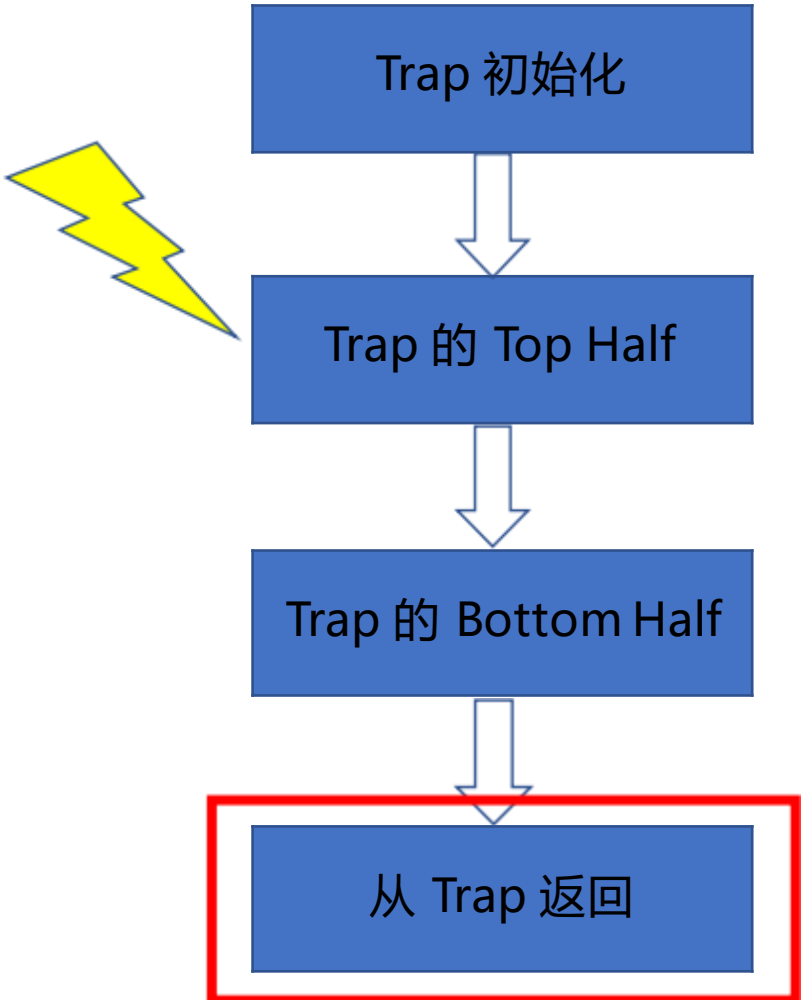


trap handler：软件需要做的事情

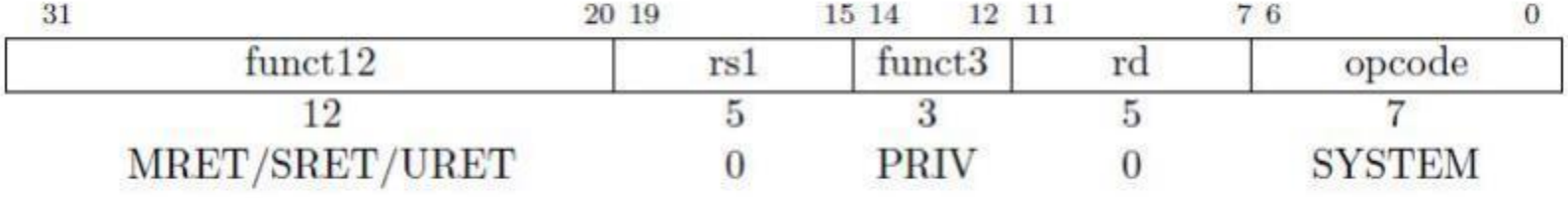
- 保存(save)当前控制流的上下文信息 (利用mscratch)
- 调用 C 语言的 trap handler
- 从 trap handler 函数返回，mepc 的值有可能需要调整
- 恢复（restore）上下文的信息
- 执行MRET指令返回到trap 之前的状态

```
trap_vector:
        # save context(registers).
        csrrw   t6, mscratch, t6        # swap t6 and mscratch
        reg_save t6

        # Save the actual t6 register, which we swapped into
        # mscratch
        mv      t5, t6          # t5 points to the context of current task
        csrr    t6, mscratch    # read t6 back from mscratch
        sw      t6, 120(t5)     # save t6 with t5 as base

        # Restore the context pointer into mscratch
        csrw    mscratch, t5

        # call the C trap handler in trap.c
        csrr    a0, mepc
        csrr    a1, mcause
        call    trap_handler

        # trap_handler will return the return address via a0.
        csrw    mepc, a0

        # restore context(registers).
        csrr    t6, mscratch
        reg_restore t6

        # return to whatever we were doing before trap.
        mret
```

RISC-V Trap 处理流程



退出 trap：编程调用 MRET 指令

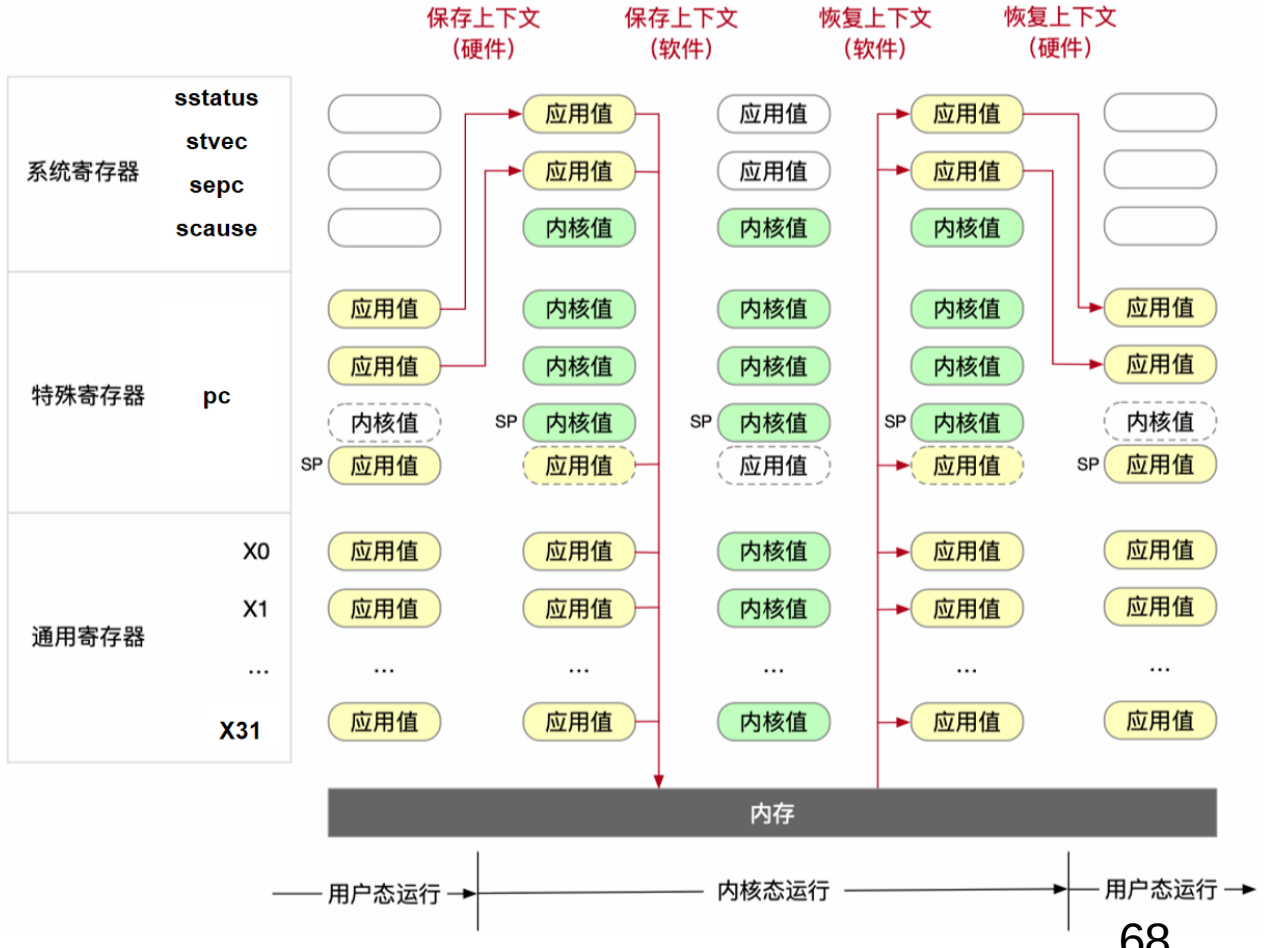


【参考 2】 3.2.2 Trap-Return Instructions

- 针对不同权限级别下如何退出 trap 有各自的返回指令xRET(x=M/S/U)
- 以在M模式下执行mret指令为例，会执行如下操作:
- 当前Hart的权限级别 = mstatus.MPP; mstatus.MPP = U(如果hart不支持U则为M)
- mstatus.MIE = mstatus.MPIE; mstatus.MPIE = 1
- pc = mepc

内核态与用户态的切换

用户态/内核态切换时的处理器状态变化



处理器在切换过程中的任务

1. **将发生异常事件的指令地址保存在sepc中**
2. **将异常事件的原因保存在scause寄存器中**
   - 例如，是执行ecall指令导致的，还是访存缺页导致的
3. **sstatus寄存器中的SPP值为0切换特权级别为用户级**
4. **将引发缺页异常的内存地址保存在stvec中**
5. **切换用户栈为内核栈**
   - 可以使用sscratch来保存当前进程task struct的地址，异常发生时取出
6. **sstatus寄存器中的SPP值为1切换特权级别为监管级**
7. **找到异常处理函数的入口地址，并将该地址写入PC，开始运行操作系统**
   - 根据stvec寄存器中保存的异常向量表基地址，以及发生异常事件的类型确定

思考题

- 为什么操作系统不能直接使用应用程序在用户态的栈呢?

处理器的这些操作都是必要的

- **PC寄存器的值必须由处理器保存**
  - 否则当操作系统开始执行时，PC将被覆盖
- **栈的切换也必须由硬件完成**
  - 否则操作系统有可能使用用户态的栈，导致安全问题

sret：从内核态返回到用户态

1. **将当前特权级别设置为sstatus.SPP**
   - sstatus.SPP=0，切换到用户模式
2. **切换内核栈为用户栈**
   - 使用sscratch保存进程task struct的地址，异常时取出
   - 退出异常时将task struct存回sscratch中
3. **将sepc中的地址写入PC，并执行应用程序代码**

操作系统在切换过程中的任务

- **主要任务：将属于应用程序的 CPU 状态保存到内存中**
  - 用于之后恢复应用程序继续运行
- **应用程序需要保存的运行状态称为处理器上下文**
  - 处理器上下文（Processor Context）：应用程序在完成切换后恢复执行所需的最小处理器状态集合
  - 处理器上下文中的寄存器具体包括：
    - 通用寄存器 X0-X31
    - 系统寄存器，在不同的特权模式下有相对应的寄存器
    - 额外的非特权寄存器PC

系统调用

常见的Linux的系统调用

编号	名称	描述.	编号	名称	名称.
17	getcwd	Get current working directory	129	kill	Send signal to a process
23	dup	Duplicate a file descriptor	172	getpid	Get process ID
56	openat	Open a file	214	brk	Set the top of heap
57	close	Close a file	215	munmap	Unmap a file from memory
63	read	Read a file	220	clone	Create a process
64	write	Write a file	221	execve	Execute a program
80	fstat	Get file status	222	mmap	Map a file into memory
93	_exit	Terminate the process	260	wait4	Wait for process to stop

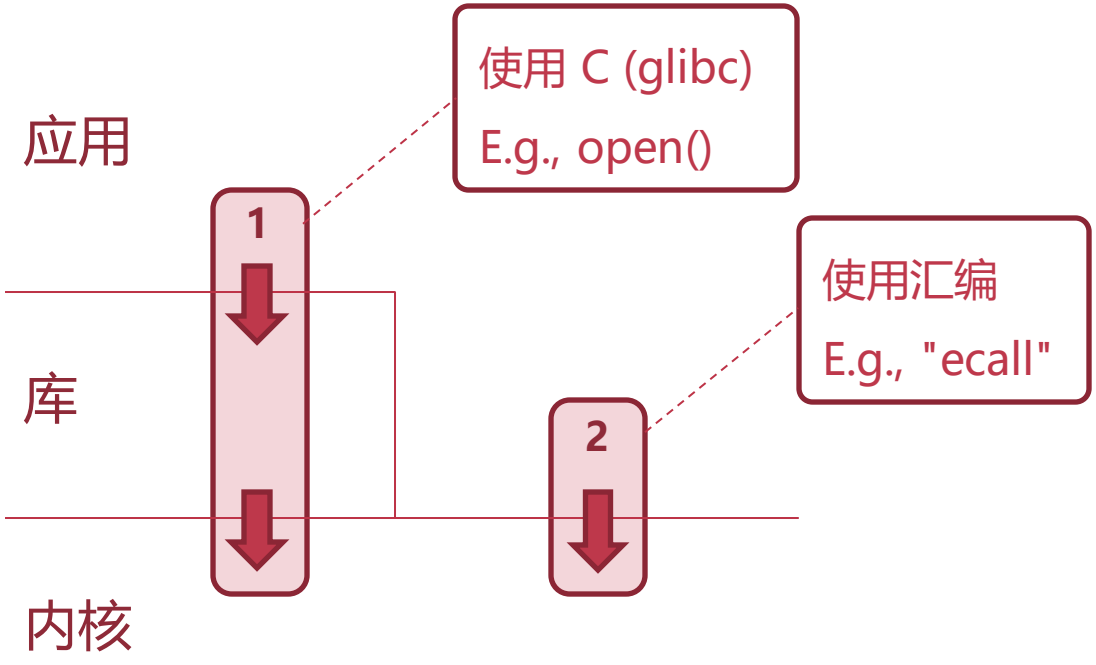
如何跟踪系统调用?

```
int main() {
  write(1, "Hello world!\n", 13);
}
```

```
$ strace -o hello.out ./hello
```

```
execve("./hello2", ["./hello2"], [/* 59 vars */]) = 0
uname({sys="Linux", node="kiwi", ...}) = 0
brk(0) = 0xca9000
brk(0xcaa1c0) = 0xcaa1c0
arch_prctl(ARCH_SET_FS, 0xca9880) = 0
brk(0xccb1c0) = 0xccb1c0
brk(0xccc000) = 0xccc000
write(1, "Hello world!\n", 13) = 13
exit_group(13) = ?
```

程序员角度看系统调用



ecall指令：从用户态进入内核态

- **CPU的行为**
  - sstatus寄存器跟踪处理器的当前运行状态，SPP位设置为0
  - 将用户程序中ecall后的第一条指令所在地址保存到寄存器sepc
  - 切换用户栈为内核栈
  - 根据异常向量表中的配置，执行对应异常向量条目所配置的代码

sret指令：从内核态返回用户态

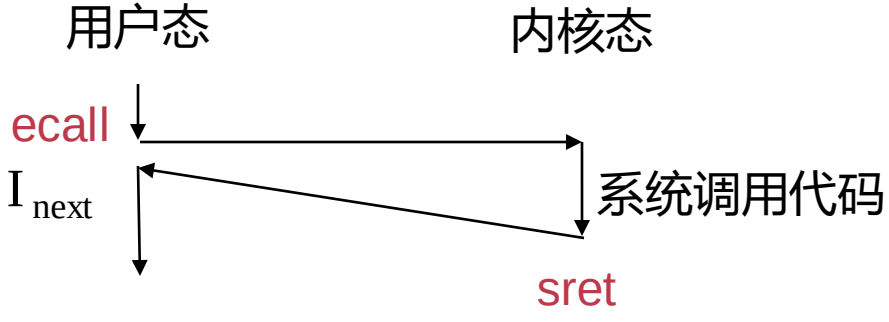
- **CPU的行为**
  - 根据sstatus中SPP的值（此时为0）将权限级别设置为用户模式
  - 切换内核栈为用户栈
  - 将sepc寄存器中所保存的返回地址重设到程序计数器PC中，执行应用程序中的代码

Questions

- **内核自己是否可以调用syscall?**
  - 如果可以，是否需要换栈?

系统调用的参数与返回值

- **参数传递**
  - 最多允许8个参数：a0-a7
  - a7用于存放系统调用编号
  - 调用者保存的寄存器必须在用户态保存
- **返回值存放于a0寄存器中**



系统调用返回值与errno

- **一般库函数**
  - 出错时返回-1，并设置全局变量errno为具体的错误值
- **系统调用通过寄存器向应用传递返回值**
  - 一般设置为 -errno
  - 库对系统调用的 wrapper code 会将系统调用的返回值转换为库函数形式的返回值

Q: 如果寄存器放不下参数怎么办?

- **寄存器放不下，只能通过内存传参**
  - 将参数放在内存中，将指针放在寄存器中传给内核
  - 内核通过指针访问相关参数
- **想一想，可能有什么问题?**
  - 情况-1：指针指向了内核区域（攻击！）
  - 情况-2：指针指向的区域被swap-out了
    - 导致内核访问时出现了page fault，怎么处理?
  - 情况-3：指针指向了未映射区域
    - 导致segmentation fault，会怎么处理?

如何验证用户态的指针合法性?

- **如果仔细检查的话太费时**
  - 需要检查指针是否来自所有的合法内存区域（VMA）
- **Linux的方案**
  - 仅仅做一个简单的检查，判断是否在最大的VMA
  - 即使检查过了，指针依然有可能不合法
  - 在内核态发生非法内存访问，一般会被认为是内核bug而触发Oops，并kill掉相关进程

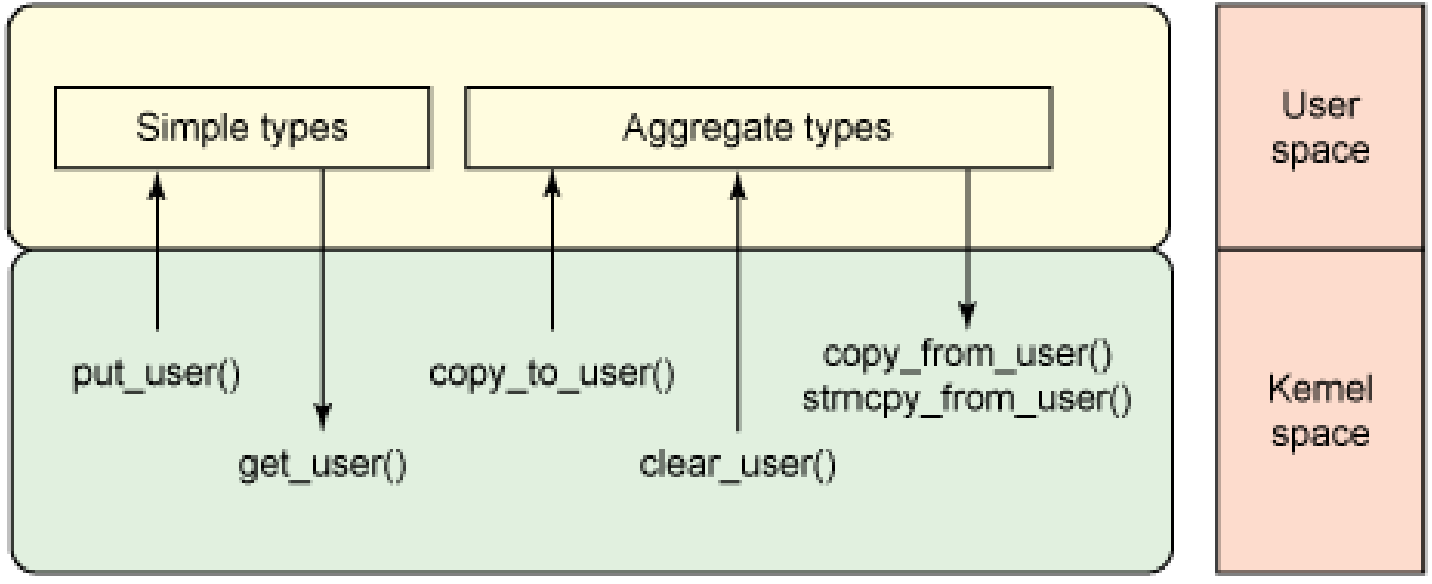
处理由用户指针而带来的fault

- **内核代码通过一组特定的routine来访问用户内存**
  - 例如：copy_from_user
- **当发生page fault的时候，内核会检查PC的值**
  - 如果PC的值在这组routine中，则不会Oops，而是会尝试运行fixup代码，去解决page fault
- **在内核中，这是一个常常会出现漏洞的地方**
  - 回顾下x86的 SMAP 特性：防止内核访问用户态内存

访问用户态内存的一组routine

Function	Action
get_user(), __get_user()	reads integer (1,2,4 bytes)
put_user(), __put_user()	writes integer (1,2,4 bytes)
copy_from_user(), __copy_from_user	copy a block from user space
copy_to_user(), __copy_to_user()	copy a block to user space
strncpy_from_user(), __strncpy_from_user()	copies null-terminated string from user space
strnlen_user(), __strnlen_user()	returns length of null-terminated string in user space
clear_user(), __clear_user()	fills memory area with zeros

访问用户态内存的一组routine



Virtual Dynamic Shared Object

VDSO

Motivation

- **系统调用的时延不可忽略**
  - 尤其是调用非常频繁的那些
    - 如 gettimeofday()
- **如何降低系统调用的时延?**
  - 大部分时延都是由于U->S/M的模式切换带来的
  - 如果没有模式切换，那么就不需要保存回复状态
  - **想象一下切换都带来了什么样的开销?**

The Code of *gettimeofday()*

- **内核定义**
  - 在编译时作为内核的一部分
- **用户态运行**
  - 将gettimeofday的代码加载到一块与应用共享的内存页
  - 这个页称为：vDSO
    - Virtual Dynamic Shared Object
  - Time 的值同样映射到用户态空间（只读）
    - 只有在内核态才能更新这个值
- **Q：和以前的gettimeofday相比有什么区别?**

vDSO的共享页在哪儿?

```
$ ldd `which bash`
  linux-vdso.so.1  (0x0000003f98eb1000)
  libm.so.6 => /lib/libm.so.6 (0x0000003f98e41000)
  libc.so.6 => /lib/libc.so.6 (0x0000003f98d17000)
  /lib/ld-linux-riscv64-p64d.so.1 (0x0000003f98eb2000)
```

The source can be found in arch/riscv/kernel/vdso/vgettimeofday.c

Flexible System Call Scheduling with Exception-Less System Calls, OSDI'10

FLEX-SC

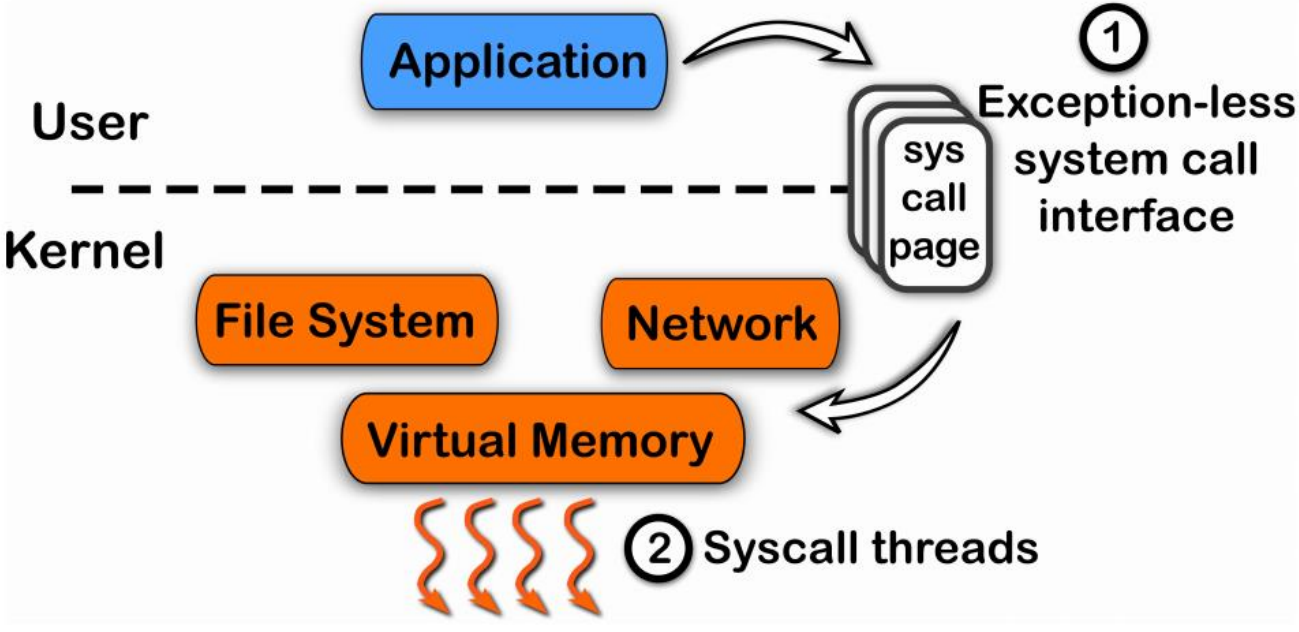
Motivation

- **如何进一步降低系统调用的时延?**
  - 不仅仅是 gettimeofday()
- **"时间都去哪儿了? "**
  - 大部分是用来做状态的切换
    - 保存和恢复状态 + 权限的切换
  - Cache pollution
- **是否有可能在不切换状态的情况下实现系统调用?**

Flexible System Call

- **一种新的syscall机制**
  - 引入 **system call page** ，由 user & kernel 共享
  - User threads 可以将系统调用的请求 **push** 到 system call page
  - kernel threads 会从system call page **poll** system call 请求
- **Exception-less syscall**
  - 将系统调用的调用和执行解耦，可分布到不同的CPU核

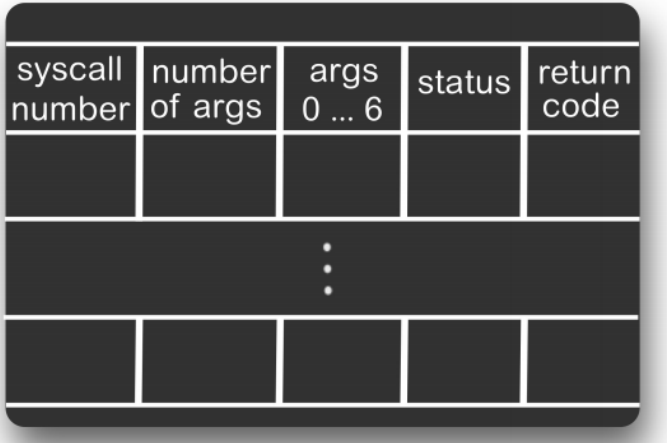
System Call的另一种方法



Exception-less System Call

```
write(fd, buf, 4096);
```

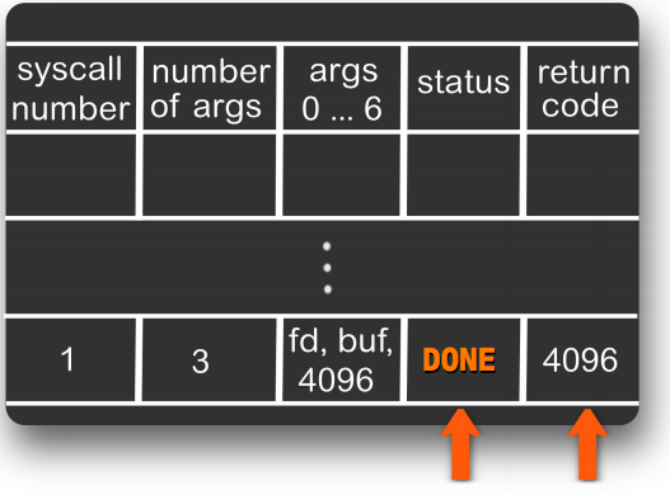
```
entry = free_syscall_entry();

/* write syscall */
entry->syscall = 1;
entry->num_args = 3;
entry->args[0] = fd;
entry->args[1] = buf;
entry->args[2] = 4096;
entry->status = SUBMIT;

while (entry->status != DONE)
   do_something_else();

return entry->return_code;
```



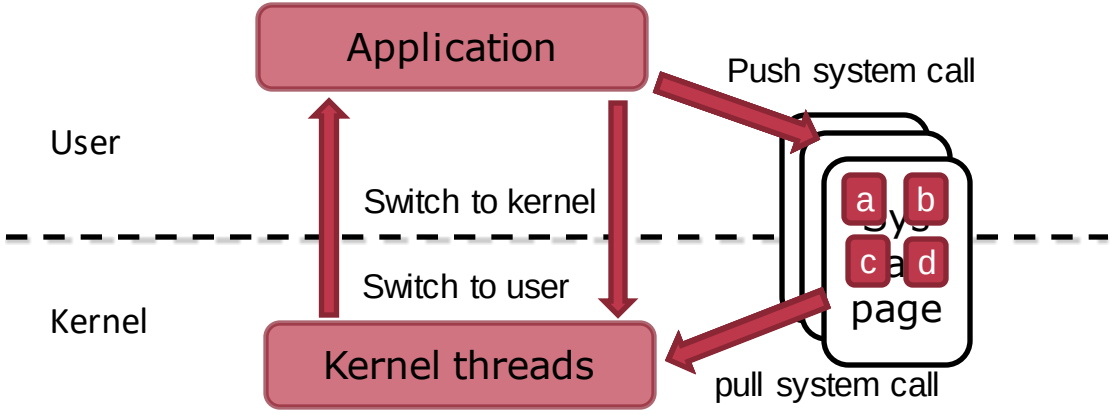
Kernel填充syscall的返回值

```
write(fd, buf, 4096);
```

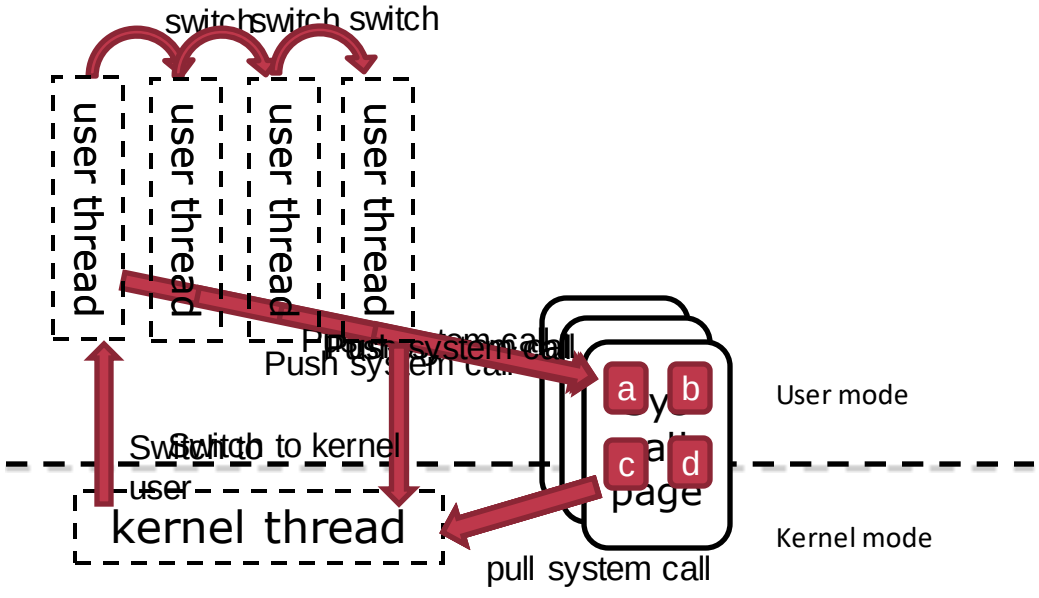
```
entry = free_syscall_entry();

/* write syscall */
entry->syscall = 1;
entry->num_args = 3;
entry->args[0] = fd;
entry->args[1] = buf;
entry->args[2] = 4096;
entry->status = SUBMIT;

while (entry->status != DONE)
   do_something_else();

return entry->return_code;
```



单核上：Single Threads



单核上：Multiple Threads



FlexSC: Flexible System Call Scheduling with Exception-Less System Calls

消息式系统调用

- GenterOS
- XOS

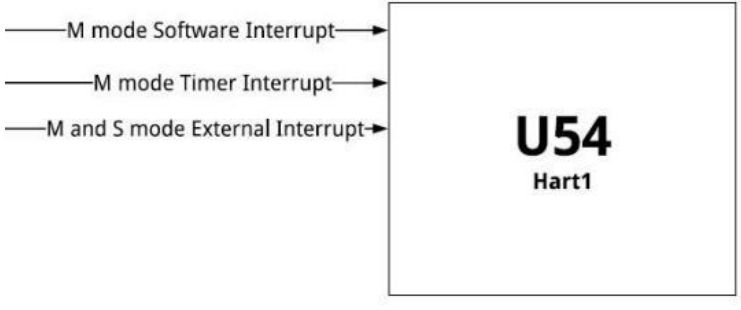
中断处理-RISC-V

RISC-V 中断（Interrupt）的分类

- **本地（Local）中断**
  - software interrupt
  - timer interrupt
- **全局（Global）中断**
  - externel interrupt

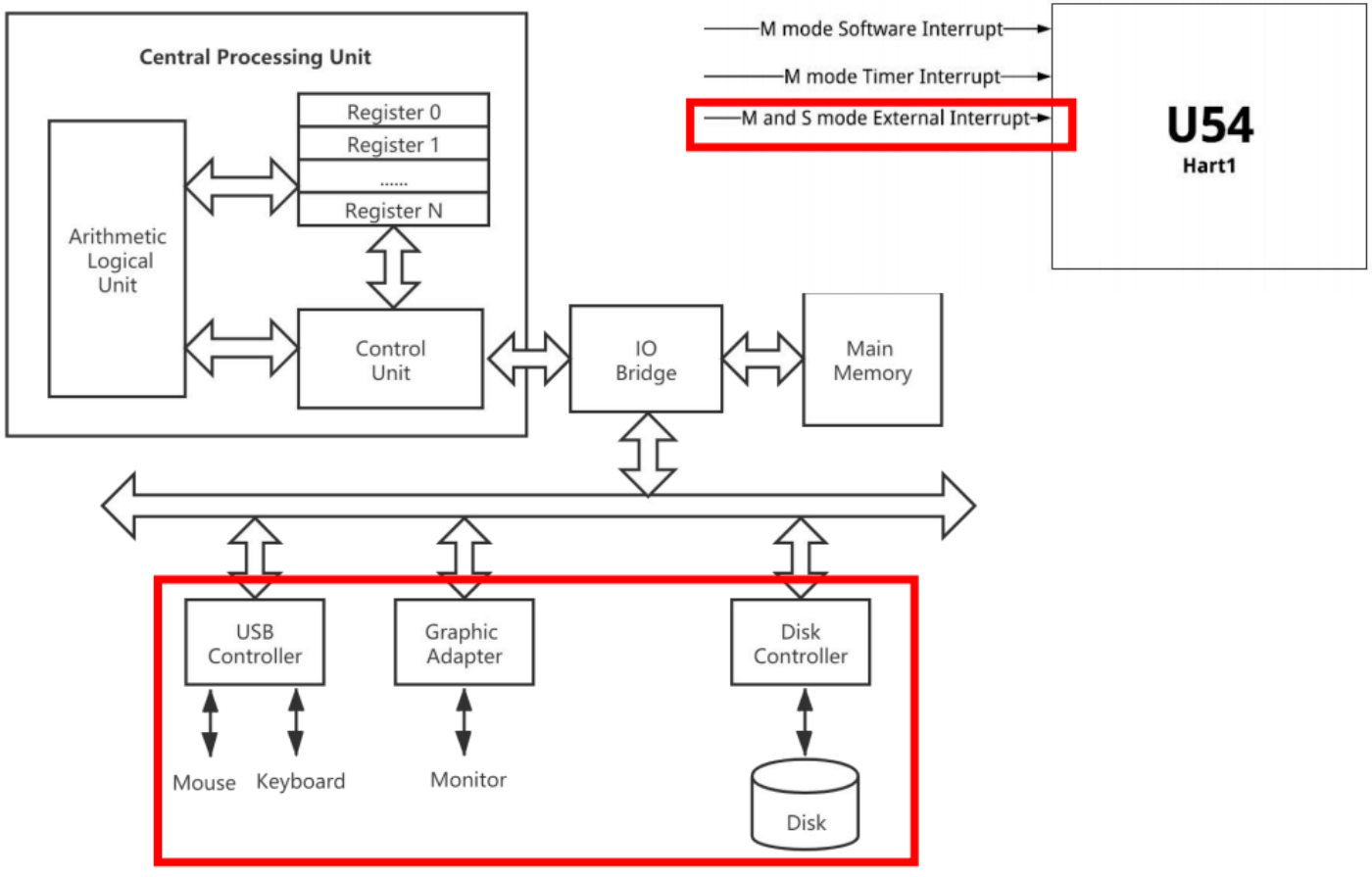
Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved for future standard use*
1	11	Machine external interrupt
1	12–15	*Reserved for future standard use*
1	≥16	*Reserved for platform use*

Table 3.6: Machine cause register (`mcause`) values after trap.



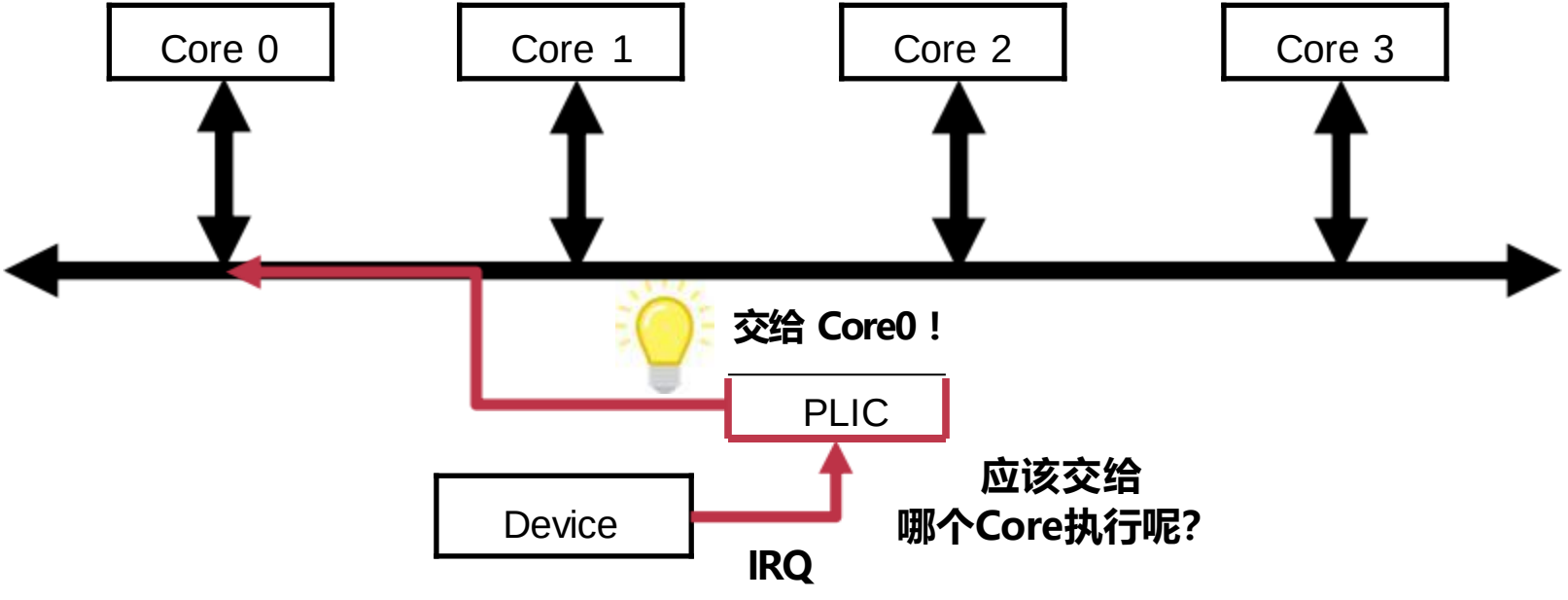
【参考 3】 Figure 3: FU540-C000 Interrupt Architecture Block Diagram.

外部中断

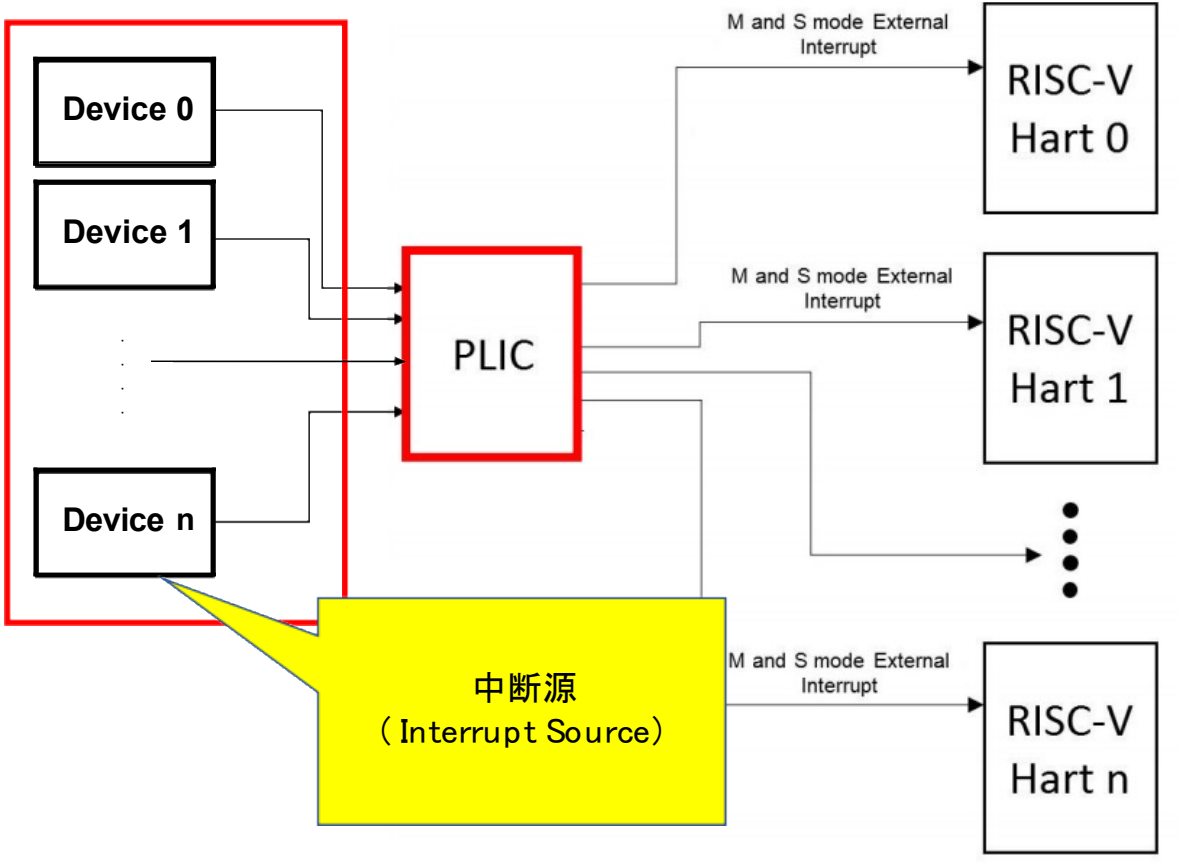

Central Processing Unit
Register 0
Register 1
.....
Register N
Arithmetic Logical Unit
Control Unit
IO Bridge
Main Memory
M mode Software Interrupt
M mode Timer Interrupt
M and S mode External Interrupt
U54
Hart1
USB Controller
Graphic Adapter
Disk Controller
Mouse
Keyboard
Monitor
Disk

问题：多核CPU如何处理中断?

- 中断：如何避免打断所有核呢?



PLIC（Platform-level Interrupt Controller）



中断源

- Interrupt Source ID 范围：　1 ~ 53（0x35)
- 0 预留不用

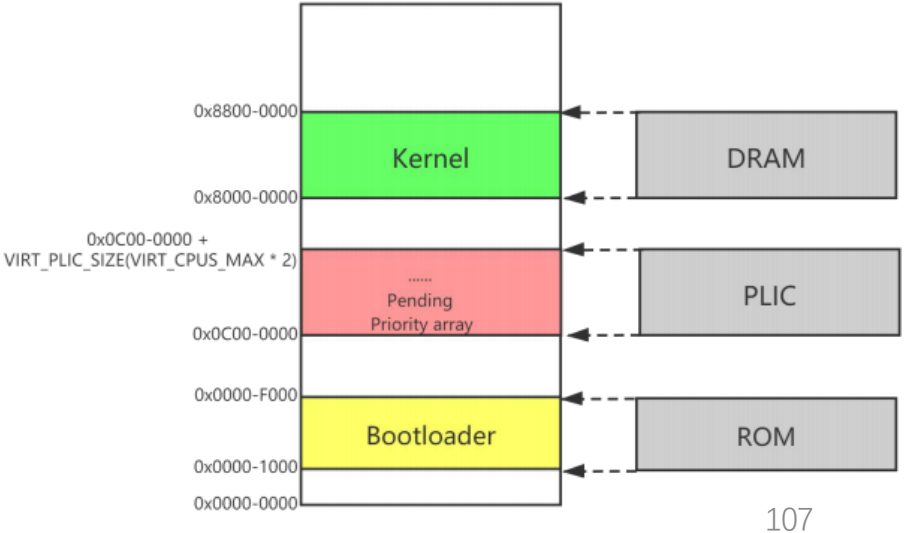
```
enum {
    UART0_IRQ = 10,
    RTC_IRQ = 11,
    VIRTIO_IRQ = 1, /* 1 to 8 */
    VIRTIO_COUNT = 8,
    PCIE_IRQ = 0x20, /* 32 to 35 */
    VIRTIO_NDEV = 0x35 /* Arbitrary maximum number of interrupts */
};
```

PLIC编程接口—寄存器

- RISC-V 规范规定，PLIC的寄存器编址采用内存映射（memory map) 方式。每个寄存器的宽度为 32-bit。
- 具体寄存器编址采用 base + offset 的格式，且 base 由各个特定 platform 自己定义。

```
static const MemMapEntry virt_memmap[] = {
    [VIRT_DEBUG] =       {        0x0,         0x100 },
    [VIRT_MROM] =        {     0x1000,        0xf000 },
    [VIRT_TEST] =        {   0x100000,        0x1000 },
    [VIRT_RTC] =         {   0x101000,        0x1000 },
    [VIRT_CLINT] =       {  0x2000000,       0x10000 },
    [VIRT_PCIE_PIO] =    {  0x3000000,       0x10000 },
    [VIRT_PLIC] =        {  0xc000000, VIRT_PLIC_SIZE(VIRT_CPUS_MAX * 2) },
    [VIRT_UART0] =       { 0x10000000,         0x100 },
    [VIRT_VIRTIO] =      { 0x10001000,        0x1000 },
    [VIRT_FLASH] =       { 0x20000000,     0x4000000 },
    [VIRT_PCIE_ECAM] =   { 0x30000000,    0x10000000 },
    [VIRT_PCIE_MMIO] =   { 0x40000000,    0x40000000 },
    [VIRT_DRAM] =        { 0x80000000,           0x0 },
};
```

```
#define PLIC_BASE 0x0c000000L
```



中断控制器需要考虑的问题

- **如何指定不同中断的优先级**
  - 低优先级中断处理中，出现了高优先级的中断
  - 嵌套中断
- **中断交给谁处理**
- **如何与软件协同**

中断控制器

- **中断类型变多，将中断分发给不同的核（对称或非对称）进行处理**

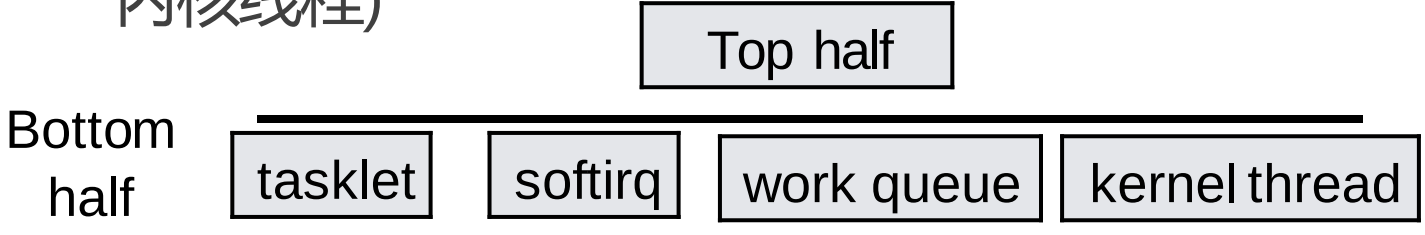
- **主要功能**
  - 分发：管理所有中断、 决定优先级、路由
  - CPU接口：给每个CPU核有对应的接口

讨论：中断处理不能做太多的事情?

- 怎么办?

案例：Linux的中断处理理念

- **在中断处理中做尽量少的事**
- **推迟非关键行为**
- **结构：Top half & Bottom half**
  - Top half : 做最少的工作后返回
    - Bottom half ： 推迟处理 (softirq, tasklets, 工作队列，内核线程)



Top Half：马上做

- **最小的、 公共行为**
  - 保存寄存器、 屏蔽其他中断
  - 恢复寄存器，返回原来场景
- **最重要：调用合适的由硬件驱动提供的中断处理handler**
- **因为中断被屏蔽，所以不要做太多事情（时间、空间）**
- **使用将请求放入队列，或者设置标志位将其他处理推迟到bottom half**

Top Half：找到handler

- 现代处理器中，多个I/O设备共享一个IRQ和中断向量
- 多个ISR (interrupt service routines)可以结合在一个向量上
- 调用每个设备对应该IRQ的ISR

Bottom Half：延迟完成

- **提供一些推迟完成任务的机制**
  - softirqs
  - tasklets （建立在softirqs之上)
  - 工作队列
  - 内核线程
- **这些工作可以被中断**

注意：中断处理没有进程上下文

为什么?

- 中断（和异常相比）和具体的某条指令无关
- 也和中断时正在跑的进程、用户程序无关
- 中断处理handler不能睡眠!

中断处理中的一些约束

- **不能睡眠**
  - 或者调用可能会睡眠的任务
- **不能调用`schedule()`调度**
- **不能释放信号或调用可能睡眠的操作**
- **不能和用户地址空间交换数据**