Lab6 中断与异常

为什么我们需要中断、异常和特权模式

特权模式

特权模式的想法很直观。一些指令很危险，我们不希望用户编写的代码能够执行这些指令；CSR寄存器保存了特权模式的信息，我们也不希望用户代码能够修改CSR寄存器。因此我们需要不同的特权模式来区别有权限执行的指令和有权限修改的寄存器。

RISCV指令集设计了三种特权模式，权限由高到低分别是

- `Machine Mode`，拥有最高权限
- `Supervisor Mode`，操作系统运行在这个级别
- `User Mode`，普通用户程序运行在这个级别

异常

异常处理

软件/硬件难免会出错。例如：

- 数组越界，你写的程序访问了不该访问的内存
- 程序被病毒感染了，(x86 CPU)指令被修改为 `HLT`，直接把CPU关掉
- 程序被病毒感染了，指令被修改为RISCV指令集里面不存在的一条指令
- 内存的某些单元损坏了

这时，计算机系统应该如何应对？

对于前两种情况，或许存在一些软件层面的办法。例如一些高级语言(`Java`)能够监测数组越界；操作系统或者杀毒软件能够检查软件中的恶意代码。但是很显然这些办法会带来额外的性能开销。

更麻烦的是后两种情况。在软件层面上，这些问题是难以被检测到的。这种“异常”需要在CPU实际执行这条指令时才会被发现。

上面列举的四情况实际上都属于CPU意义上的“异常”。具体来说，异常可以包括：

- 非法内存访问（非法地址、地址没有对齐等）
- 非法指令
- 越权指令
- ...

既然我们确定了“异常”需要在CPU执行时被检测，随之而来的问题就是，遇到异常后CPU应该如何处理。

直接忽略异常显然是不行的。那么如何让用户知道异常发生了，并进行修复呢？我们可以回想 `python` 等编程语言中的办法：

```
try:
    do_something()
except KeyError:
    fix_key_error()
except ValueError:
    fix_value_error()
except Exception as e:
    print(f"Uncaught error: {e}")
```

我们预先将可能发生的异常定义出来，然后对于每一种异常，指定对应的处理代码。

类似地，在CPU中我们也能这么做！回想一下，CPU只是一个能不断从内存取指令然后执行的机器。我们可以预先在内存的一部分区域保存好处理某种异常的代码，然后告诉CPU“异常类型→对应handle代码的内存地址”的映射关系。当CPU遇到异常时，首先判断异常对应的类型，然后跳转到对应的内存地址开始执行我们的异常处理代码。这就是CPU中的异常处理机制。

异常处理的过程中，CPU的特权级别会相应地提升，以便操作系统进行一些需要特权的操作。在异常处理返回后，特权级别会再下降到之前的级别。

资源抽象

异常并非都是程序出了问题，也可以用于用户主动切换到操作系统内核态请求服务。

举例来说，`print`的实际实现是相当复杂的。程序需要知道屏幕的光标位置、处理对应的显卡绘制逻辑，并且在不同的硬件上，这一系列操作都不相同。与各种显卡驱动交互的操作逻辑显然不应该集成在每一个`print("Hello World")`的程序里，而是由操作系统暴露出一个抽象的资源接口，供程序直接调用。

在这种情况下，用户程序可以通过执行一个特殊的指令`ecall`主动引发“异常”，跳转到操作系统预留好的接口，请求服务。这个过程又称为系统调用(System Call)。

中断

中断与异常类似，都是通过某种信号，强制CPU跳转到另一段代码执行。区别在于，异常的信号是同步的，而中断的信号是异步的。发生中断时，也需要短暂跳转到更高的特权等级，由操作系统处理。

时钟中断

即使在单核CPU上，一个操作系统也能够“同时”运行QQ、微信，播放音乐，浏览网页。这实际上是很神奇的一件事情。这种“同时”运行多个程序的能力实际上是操作系统通过调度机制营造出的假象。

具体来说，操作系统使用了“时间片轮转”的技术，将CPU时间切割为毫秒级别的片段。每个程序轮流获得一个时间片执行。时间满后当前程序暂停，切换回操作系统，再指定下一个程序执行。由于切换时间非常快，用户会感觉所有程序在同时运行。

这里的问题在于某个用户程序执行到规定时间后，如何将控制权切换回操作系统，以便执行下一条指令。我们能够想到下面两个办法：

1. 通过“君子约定”，要求每个程序内部都包含下面的逻辑：每隔一段时间，将控制权还给操作系统
2. 通过某种外部的方式强制打断运行

第一种方法有着很大的问题：暂且不提程序会不会遵守这种约定，如果程序陷入了死循环，根本无法执行到“切换”部分，那么操作系统就无计可施了。

因此，我们引入了“时钟中断”机制。在上一个lab中，我们实现了每个周期自增的`mcycle`寄存器。操作系统只需要在切换到用户程序前，告诉CPU“我希望这段程序执行多久”以及“时间满后跳转到哪里执行（操作系统自身的调度器代码）”。CPU会自动比对`mcycle`寄存器，当达到要求时发起一个时钟中断，强制跳转回操作系统。

外部中断、软件中断

计组课程上讲到过“中断”与“轮询”的区别。外部中断正是实现了这里的“中断”机制。

在RISCV中，“键盘输入”这种外设信号会被中断处理器处理，然后作为中断信号发送给CPU。当程序需要等待这类耗时不确定的外部信号时，不需要占用CPU时间反复轮询检查，而是只需要将自己注册为“键盘输入”这个中断信号的handler，等待信号到来时被唤醒。

软件中断的概念与之类似，只不过信号来自于内部软件，而不是外部设备。

实现细节

根据上面的思考，我们大概理解了中断和异常的原理：在特定情况下，CPU暂停当前的指令执行，跳转到预先定义好的另一段代码进行处理。

我们来考虑一些细节：

不同的中断异常类型需要不同的处理代码，CPU如何知道跳转到哪里？

一种办法是在CPU中维护一个寄存器组，按照中断异常类型的下标来取对应的地址。但是RISCV里面有超过64种中断异常类型，这个寄存器组会非常庞大而昂贵。

CPU设计是money-performance trade-off的艺术。RISCV选择的办法是在CPU中只维护一个跳转地址 `mtvec`(Machine Trap-Vector Base-Address Register)，指向的地址作为一个统一的入口，再根据中断异常类型跳转到不同的实际处理代码。

如何告诉处理程序中断异常的细节？

我们需要一系列信息：最后执行的的PC地址和中断异常的类型分别保存到`mepc`(Machine Exception Program Counter)和`mcause`，其他琐碎的信息（出错的特权等级类型等）保存到`mstatus`。

处理之后，CPU如何跳转回去？

我们之前将PC保存到了`mepc`，只需要取出来重新执行即可。同时将`mstatus`保存的各种信息复原。这一系列操作由`mret`指令实现。

中断信息如何保存？

CPU可能会同时收到多个中断信息，或者在一个中断处理没结束时收到另一个中断信息。因此我们需要提供一种方式，将中断信息保存下来，等待方便时处理。

`mip`(Machine Interrupt Pending)寄存器的每一位代表了一类中断信息。当收到中断时，我们将对应的位置1，等待后续处理。

注意：在本次实验中（以及大多数现代CPU中），中断是一个持续的信号，而不是一个短暂的上升沿。因此你可以直接用`=`对`mip`的对应位进行赋值。并且在中断处理结束后不需要手动清空`mip`，中断信号会自行停止。（实际上是中断处理程序告诉外界停止发送信号）

CPU需要处理所有中断吗？

在某些场景下，我们不希望CPU处理中断（例如，我们已经在中断异常处理过程中，或者正在进行对性能要求很高的计算）。

RISCV提供了一系列寄存器来控制CPU当前是否需要相应中断。`mstatus.mie`是全局中断控制位。在`mstatus.mie=1`的情况下，`mie`(Machine Interrupt Enable)某位为1，表示CPU会处理对应类型的中断，否则不会理会这个中断。

实验细节

准备工作

实现 `mode` 寄存器，并将其连接到 `DifftestCSRState.priviledgeMode`

Encoding	Mode
00	U
01	S（本次实验不需要实现）
11	M

本次实验只考虑U-Mode和M-Mode的切换，不需要考虑S Mode。CPU从M Mode初始启动。

异常

实现下列异常：

- 指令地址不对齐
- 数据地址不对齐
- 非法指令
- `ecall`

异常存在优先级（见下表）。发生异常时，要进行下列操作：

1. `mepc ← pc`
2. `next_pc ← mtvec`
3. `mcause[63] ← 0`表示异常，`mcause[62:0] ← 对应的异常类型`
4. `mstatus.mpie ← mstatus.mie`
5. `mstatus.mie = 0`
6. `mstatus.mpp ← mode`
7. `mode ← M Mode`
8. 清除流水线。取消当周期发起的 `dreq.valid`。已发起的 `dreq` 保留，等到 `data_ok` 后再清除流水线。

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*Reserved*
1	≥16	*Designated 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*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

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

Priority	Exc. Code	Description
Highest	3	Instruction address breakpoint
	12, 1	During instruction address translation: First encountered page fault or access fault
	1	With physical address for instruction: Instruction access fault
	2 0 8, 9, 11 3 3	Illegal instruction Instruction address misaligned Environment call Environment break Load/store/AMO address breakpoint
	4, 6	Optionally: Load/store/AMO address misaligned
	13, 15, 5, 7	During address translation for an explicit memory access: First encountered page fault or access fault
	5, 7	With physical address for an explicit memory access: Load/store/AMO access fault
Lowest	4, 6	If not higher priority: Load/store/AMO address misaligned

Table 3.7: Synchronous exception priority in decreasing priority order.

中断

实现时钟中断、外部中断、软件中断

我们提供的`trint`， `exint`， `swint`分别表示三种中断信号。

与异常不同，中断的处理是有条件的。

> An interrupt i will trap to M-mode (causing the privilege mode to change to M-mode) if all of the following are true: (a)either the current privilege mode is M and the MIE bit in the `mstatus` register is set,or the current privilege mode has less privilege than M-mode;(b)bit `i` is set in both `mip` and `mie`.
>
> These conditions for an interrupt trap to occur must be evaluated in a bounded amount of time from when an interrupt becomes, or ceases to be, pending in `mip`, and must also be evaluated immediately following the execution of an `xRET` instruction or an explicit write to a CSR on which these interrupt trap conditions expressly depend(including `mip`, `mie`, `mstatus`)

省流：

中断处理**实际发生**的条件是同时满足 (1)当前是M Mode且`mstatus.mie=1` 或者 当前不是M Mode (2) `mip[i]=1` 且 `mie[i]=1`

中断处理**evaluate**（进行上面的检查，来确定要不要跳转到中断向量）的条件是满足下面之一 (1) 刚收到一个中断信号 (2) 刚执行过`mret` (3) `mip`， `mie`， `mstatus`刚被CSR写入修改过。

本次Lab我们只要求在 (1) 刚收到一个中断信号 时执行中断`evalue`

> Bonus: 严格按照上面的要求，在三个条件任一满足时evaluate中断

处理中断时，除了第三步要将`mcause[63]`赋值为1外，其他进行的操作与异常处理相同。

其他指令

`mret`

1. `mstatus.mie ← mstatus.mpie`
2. `mstatus.mpie = 1`
3. `mstatus.mpp ← mode`
4. `mode ← mstatus.mpp`
5. `mstatus.xs ← 0`

Bonus

我们使用`trint`模拟了时钟中断。Bouns内容是自行实现`mtimecmp`寄存器，向其写入一个数值后，当`mcycle`大于这个数值时，持续发出时钟中断信号，直到中断处理程序将`mtimecmp`设置为一个更大的值。