OS Lab3

练习1：完善中断处理 （需要编程）

代码实现

在`kern/trap/trap.c`文件的`interrupt_handler`函数中，针对`IRQ_S_TIMER`进行了如下实现：

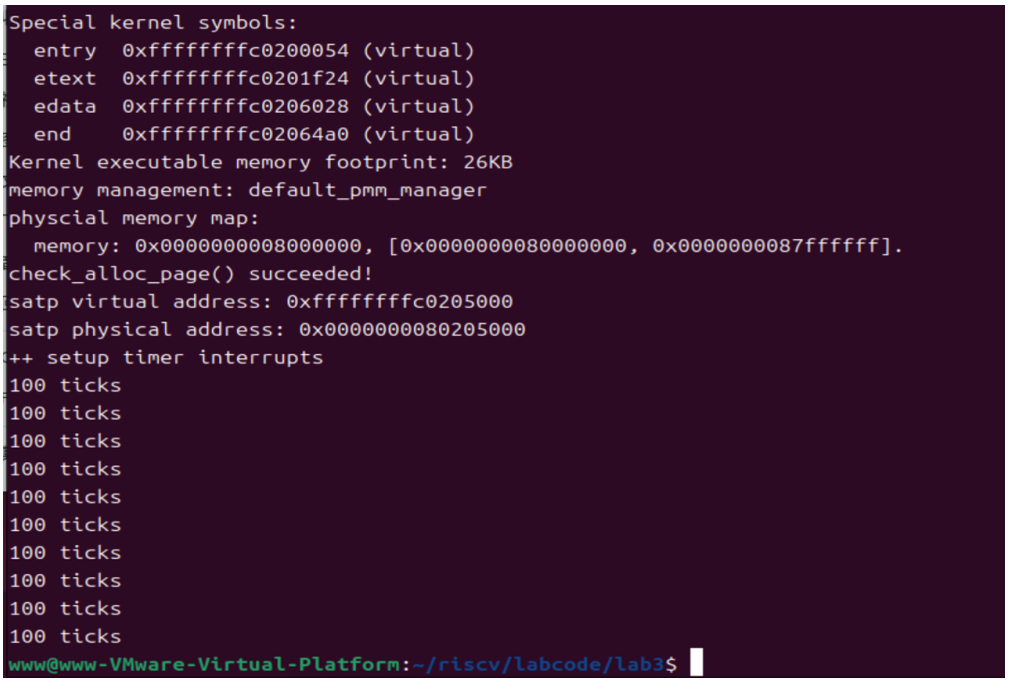
```
case IRQ_S_TIMER:
    // (1) 设置下次时钟中断
    clock_set_next_event();

    // (2) 计数器加一
    ticks++;

    // (3) 每100次中断输出信息并增加打印次数
    if (ticks % TICK_NUM == 0) {
        print_ticks();
        num++;

        // (4) 打印10次后关机
        if (num == 10) {
            sbi_shutdown();
        }
    }
    break;
```

- `ticks`：全局时钟中断计数器（在`clock.h`中声明为`extern volatile size_t ticks;`）
- `num`：局部静态变量，记录打印次数
- `TICK_NUM`：宏定义，值为100，表示每100次中断打印一次

首先通过`clock_set_next_event()`函数设置下一次时钟中断，当每次中断发生时`ticks`计数器加1，当`ticks`达到100的倍数时，调用`print_ticks()`输出"100 ticks"，最后当打印次数达到10次时，调用`sbi_shutdown()`关机。

运行结果

我们使用命令`make qemu`，可以得到如下结果，成功打印了10行的100 ticks。



定时器中断处理流程

1.中断触发阶段

RISC-V的定时器硬件在预定时间到达时产生中断信号，之后CPU跳转到`stvec`寄存器指定的异常处理入口地址（`__alltraps`）。并在`trapentry.S`中保存所有寄存器状态到栈中，构建`trapframe`结构。

2.中断分发阶段

从汇编代码跳转到C语言的`trap()`函数，在函数`trap_dispatch()`根据`tf->cause`判断中断类型，由于`cause`最高位为1，识别为中断，调用`interrupt_handler()`。

3.时钟中断处理阶段

在`interrupt_handler()`中，通过`switch(cause)`匹配到`IRQ_S_TIMER`

具体处理：

- 调用`clock_set_next_event()`重新设置定时器
- `ticks`计数器递增
- 判断是否达到打印条件
- 判断是否达到关机条件

4.中断返回阶段

处理完成后，从`trapframe`恢复所有寄存器状态，之后通过`sret`指令从中断返回用户模式，继续执行被中断的程序。

扩展练习 Challenge1：描述与理解中断流程

（一）ucore 中处理中断与异常的整体流程

在 ucore 中，中断或异常的处理流程从 CPU 检测到异常或中断信号开始，体现了硬件快速响应与软件逻辑处理的协同设计。总体来看，可以分为三个阶段来理解。

可以的，这一部分可以稍微扩展，把“硬件捕获中断”和“入口跳转”的过程讲得更详细，并结合相关代码展示。下面是整理后的解析：

1. 中断检测与入口跳转阶段

在 ucore 中，当 CPU 执行程序时，如果发生异常（比如非法指令、页错误）或者收到外部中断信号（比如定时器中断、外设中断），CPU 硬件会立即捕获这一事件。这一步是**硬件自动触发**的，不需要软件干预。

硬件捕获后会做两件关键的事情：

（1）保存当前执行状态

- CPU 会把一些最基本的状态信息保存起来，比如程序计数器（PC）、异常类型标志、模式信息等。
- 这保证了中断或异常处理完成后，可以正确返回到被打断的程序继续执行。

（2）跳转到中断/异常处理入口

- CPU 会根据**中断向量表**找到对应的入口地址，然后跳转执行。
- 在 ucore 中，这个统一入口在系统初始化阶段设置，通过如下 C 代码完成（`trap.c` 中的 `idt_init()`）：

```
extern void __alltraps(void);

/* 初始化异常向量 */
void idt_init(void) {
    /* 设置 sscratch 寄存器为 0，表示当前执行在内核中 */
    write_csr(sscratch, 0);

    /* 设置异常/中断统一入口地址 */
    write_csr(stvec, &__alltraps);
}
```

这里的关键点：

- `write_csr(stvec, &__alltraps)`：把 CSR（控制状态寄存器）`stvec` 设置成 `__alltraps` 标签地址。
  - `stvec` 是 RISC-V 中的**异常向量寄存器**，它告诉 CPU 当发生 trap（中断或异常）时跳到哪里执行。
  - `__alltraps` 是汇编标签，定义在 `trapentry.S` 中，负责保存寄存器现场并转到 C 层 `trap()` 函数处理。
- 所有类型的中断或异常都会跳到同一个入口 `__alltraps`，这让内核可以**统一管理和分发**，而不需要针对每一种事件写不同的入口处理代码。

2. 现场保存与分发处理阶段

控制权从正在执行的程序切换到内核的统一入口 `__alltraps` 之后，在这个阶段，核心目标是**先把当前 CPU 的状态完整保存**，再把控制权交给 C 层的 `trap()` 函数处理。整个过程可以分成以下几步：

（1）保存寄存器现场

在 `__alltraps` 中，最重要的就是执行 `SAVE_ALL` 宏：

```
.macro SAVE_ALL
    csrw sscratch, sp               # 将当前栈指针保存到 sscratch（防止递归异常）
    addi sp, sp, -36 * REGBYTES     # 栈空间下降，为保存寄存器留位置

    # 保存通用寄存器 x0~x31
    STORE x0, 0*REGBYTES(sp)
    STORE x1, 1*REGBYTES(sp)
    ...
    STORE x31, 31*REGBYTES(sp)

    # 保存关键 CSR 寄存器
    csrrw s0, sscratch, x0  # 读取 sscratch
    csrr s1, sstatus        # 当前状态寄存器
    csrr s2, sepc            # 异常发生的指令地址
    csrr s3, sbadaddr        # 错误地址或非法访问地址
    csrr s4, scause          # 异常/中断原因

    STORE s0, 2*REGBYTES(sp)
    STORE s1, 32*REGBYTES(sp)
    STORE s2, 33*REGBYTES(sp)
    STORE s3, 34*REGBYTES(sp)
    STORE s4, 35*REGBYTES(sp)
.endm
```

解释：

1.通用寄存器（x0~x31）

- 保存了 CPU 的工作数据，包括临时值（`t0~t6`）、函数调用保存寄存器（`s0~s11`）、函数参数/返回值寄存器（`a0~a7`）等。
- 保存顺序固定，这样恢复时可以一一对应，保证上下文一致。

2.关键控制状态寄存器（CSR）

- `sstatus`：记录当前 CPU 状态（用户态/内核态、是否允许中断等）。
- `sepc`：异常或中断发生的指令地址，处理完后需要返回这里继续执行。
- `sbadaddr`：异常访问的地址，比如页错误时的虚拟地址。
- `scause`：记录异常或中断类型，比如时钟中断、非法指令等。

3.为什么先保存 sscratch

1. 防止在处理异常的过程中又发生异常（递归 trap）时，内核知道当前是在内核中，不会错误覆盖现场。

(2) 切换到 C 层进行逻辑处理

保存完现场后，`__alltraps` 会把当前栈指针传给 C 层的 `trap()` 函数：

```
move a0, sp  # 将当前栈指针放入 a0，作为参数传递给 trap()
jal trap      # 跳转到 C 层 trap() 处理
```

对应 C 代码（`trap.c`）：

```
void trap(struct trapframe *tf) {
    // 根据 trap 类型分发到具体处理函数
    trap_dispatch(tf);
}
```

这里的 `trap_dispatch()` 会判断是中断还是异常：

```
static inline void trap_dispatch(struct trapframe *tf) {
    if ((intptr_t)tf->cause < 0) {
        // 中断
        interrupt_handler(tf);
    } else {
        // 异常
        exception_handler(tf);
    }
}
```

再进一步，中断进入 `interrupt_handler()`，根据具体类型调用对应的处理逻辑（比如时钟中断、软件中断等）；异常进入 `exception_handler()`，处理非法指令、断点等。

(3) 保存现场的作用

- **防止上下文丢失**：中断发生时，用户程序或内核程序正在使用的寄存器、状态都会被保存到栈上。
- **支持嵌套中断**：因为现场完整保存，可以处理更高优先级的中断，处理完再恢复。
- **C 层逻辑处理更安全**：trap() 可以放心修改数据，而不会破坏原程序的运行状态。

3. 现场恢复与返回阶段

中断或异常处理完毕后，会跳转到 `__trapret` 标签处执行 `RESTORE_ALL` 宏，将之前保存的寄存器和状态恢复到中断发生前的状态。

```
.macro RESTORE_ALL
    LOAD s1, 32*REGBYTES(sp)
    LOAD s2, 33*REGBYTES(sp)
    csrw sstatus, s1
    csrw sepc, s2
    ...
    LOAD x2, 2*REGBYTES(sp)    # 最后恢复栈指针
.endm
```

最后通过 `sret` 指令返回到原来的执行流，继续执行被打断的程序。

```
__trapret:
    RESTORE_ALL
    # return from supervisor call
    sret
```

总体来说，ucore 的中断处理机制体现了软硬件协作的思想：硬件负责快速响应和状态保存，而软件负责逻辑分发和资源管理。

（二）`mov a0, sp` 的作用与目的

在 `__alltraps` 宏执行完保存寄存器后，会有一条指令：

```
move a0, sp
```

这条指令的作用是将当前的栈指针（`sp`）的值放入 `a0` 寄存器中。由于 RISC-V 调用约定规定函数的第一个参数是通过 `a0` 传递的，因此这里其实是**把当前中断上下文的起始地址作为参数传递给 C 函数 `trap()`**。

在进入 `trap()` 之后，它的函数参数 `struct trapframe *tf` 实际上就是当前栈顶地址，也就是刚才在 `SAVE_ALL` 里保存的那一堆寄存器的起始位置。`trap()` 内部访问 `tf->cause`、`tf->epc` 等信息时，实际上就是在直接读取当时保存的寄存器内容。

换句话说，这个 `mov a0, sp` 的目的就是**完成从汇编层中断现场到 C 函数参数传递的桥接**。我觉得这一步虽然简单，但很关键，它使得中断现场信息能方便地传递给后续的 C 层逻辑，从而实现了硬件和软件的无缝衔接。

（三）`SAVE_ALL` 中寄存器在栈中位置的确定方式

在 `SAVE_ALL` 宏中，每个寄存器都被按顺序以固定偏移量压入栈中，比如：

```
STORE x0, 0*REGBYTES(sp)
STORE x1, 1*REGBYTES(sp)
STORE x2, 2*REGBYTES(sp)
...
STORE x31, 31*REGBYTES(sp)
```

这些偏移量的顺序实际上是根据 RISC-V 的寄存器编号和 `struct pushregs` 的定义严格对应的。也就是说，栈中的布局与 C 语言中 `trapframe` 结构体的内存布局是完全一致的。这样设计可以保证在 C 层读取时能够正确通过结构体偏移访问到对应寄存器。

此外，`SAVE_ALL` 在保存完通用寄存器后，还会把几个关键的控制寄存器（`sstatus`、`sepc`、`sbadaddr`、`scause`）依次压栈，并留出相应空间。这些寄存器代表中断发生时的特权状态、异常指令地址、错误地址和异常原因，是恢复执行所必须的信息。

我理解这个栈布局其实就是**trapframe 的物理实现形式**。通过这种严格固定的布局，ucore 可以方便地在不同层（汇编层和 C 层）共享同一份中断上下文数据结构，实现统一管理与恢复。

（四）是否所有中断都需要在 `__alltraps` 中保存全部寄存器

是的，从设计上看，`__alltraps` 在进入时无论何种中断或异常，都会执行 `SAVE_ALL`，即保存所有寄存器。这么做的确存在一定的性能开销，但它带来的好处是**统一、简单且安全**。

原因主要有两点：

1.在中断发生时，我们无法预先知道是哪种类型的中断（时钟、外设、异常等），更无法保证中断处理代码中是否会使用到所有寄存器。

2.某些中断可能会打断内核代码执行，如果不完整保存寄存器，就可能导致返回时寄存器被破坏，程序无法正确恢复。

因此，ucore 选择在 `__alltraps` 统一保存所有寄存器，这是**一种为简化实现与保证安全性所作的权衡。** 当然，在实际的操作系统（比如 Linux）中，会对不同类型的中断采用不同的保存策略——比如对快速中断只保存必要寄存器，以优化性能。但 ucore 作为教学内核，追求的是结构清晰与可理解性，所以我们的实验代码采用了“保存全部”的保守做法。

扩增练习 Challenge2：理解上下文切换机制

```
csrw sscratch, sp      # 将当前sp保存到sscratch寄存器
csrrw s0, sscratch, x0 # 将sscratch的值读到s0，同时将sscratch设为0
```

总目的：实现了一个安全的**栈指针交换与清零操作**，其核心目的是在**异常发生时保存原始栈指针并标记异常来源。**

1、`csrw sscratch, sp` 的行为与目的

在 RISC-V 架构的异常处理入口阶段，`csrw sscratch, sp` 指令执行**将当前栈指针寄存器 sp 的值写入 sscratch 控制寄存器的操作**。这一操作在异常处理机制中具有关键作用。

行为：该指令完成 sp 寄存器到 sscratch 寄存器的直接数据传送。此时的 sp 寄存器状态取决于异常发生时的上下文环境，可能指向用户栈或内核栈空间。其中，`sscratch` 是一个特殊的控制寄存器，通常被用作一个临时存储和指针交换的场所。

目的：该指令实现了异常处理初期的**状态保存功能**。在异常向量入口处，保存当前栈指针为后续的栈切换和上下文保存提供基础。sscratch 作为 RISC-V 特权架构专门为异常处理设计的临时存储寄存器，在此场景下承担起临时保存关键指针数据的职责。这一保存操作确保了在后续异常处理流程中，即使栈指针发生变化，仍能通过 sscratch 获取原始的栈指针值，为完整的上下文保存建立必要前提。

2、`csrrw s0, sscratch, x0` 的行为与目的

`csrrw s0, sscratch, x0` 指令实现原子性的读-修改-写操作。该指令在一个执行周期内完成两个独立操作：**读取 sscratch 寄存器的当前值到 s0 通用寄存器，同时将零值写入 sscratch 寄存器。**

这一指令序列的设计包含多重**目的**。

- 在数据处理层面，实现了栈指针从 sscratch 专用寄存器到 s0 通用寄存器的安全转移，为后续通过 `STORE s0, 2*REGBYTES(sp)` 指令将栈指针保存到异常帧中的预定位置做好准备。
- 在状态管理层面，将 sscratch 寄存器清零具有重要的状态标识意义。清零后的 sscratch 寄存器明确标识系统当前处于内核异常处理状态。这一状态标识为处理递归异常提供了关键判断依据：
    - 当 sscratch 为非零值时，表明异常来自用户态，可继续使用标准的内核栈处理流程；

    - 当 sscratch 为零值时，则表明异常发生在内核态内部，需要采取特殊处理措施，如使用备用栈空间或执行错误处理流程。
- 在并发安全层面，选择使用 csrrw 原子指令而非分开的 csrr 和 csrw 指令序列，确保了状态读取与设置操作的原子性。这种设计消除了在两条独立指令执行间隙发生嵌套异常而导致状态不一致的可能性，保证了异常处理入口的可靠性。

综合效果分析

这两条指令构成的序列建立了完整的异常入口处理机制：

- 实现了栈指针状态的完整保存
- 建立了明确的内核异常处理状态标识
- 确保了关键操作的原子性和可靠性
- 为递归异常检测和处理提供了基础支持

该设计体现了系统底层异常处理机制对状态保存、原子操作和错误恢复的全面考虑，是构建可靠操作系统内核的重要基础组件。

3、为什么先用 `csrw sscratch, sp` 然后再 `csrrw` 取回?

在异常处理的设计中，我们采用先执行`csrw sscratch, sp`再通过`csrrw s0, sscratch, x0`取回的序列，这体现了**防御性编程**的思想。

如果直接使用`mv s0, sp`指令，虽然能够获取当前的栈指针值，但无法同时完成设置sscratch标志位的关键操作，也无法正确处理从用户态进入异常时栈指针已经切换的复杂情况。

在实际的异常处理流程中，当异常发生时硬件会自动将栈指针切换到内核栈，此时执行`csrw sscratch, sp`保存的实际上是进入异常后的内核栈指针，这一行为初看似乎与预期目的不符。然而，在RISC-V标准异常处理规范中，通常约定在用户态运行时sscratch寄存器保存着内核栈指针，在异常入口处直接交换sp和sscratch的值。

但我们实现的代码采用了不同的策略：在异常入口时栈指针可能已经处于内核栈状态，通过`csrw sscratch, sp`先保存当前的内核栈指针，然后借助`csrrw s0, sscratch, x0`指令实际上是将sscratch作为临时存储介质来使用，最终目的是清空sscratch并设置内核态处理的标志位，从而为可能的递归异常处理提供可靠的检测机制。

4、SAVE_ALL 保存内容与顺序分析

保存的完整上下文包括：

- 所有通用寄存器（x0-x31），其中x2(sp)通过特殊路径保存
- 关键控制状态寄存器：
    - `sstatus`：特权级、中断使能等状态
    - `sepc`：保存了**被中断指令的虚拟地址**
    - `stval`：提供了**与异常相关的附加信息**
    - `scause`：记录了一个编码，精确指出了**中断或异常的具体原因**

保存顺序如下所示：

```
csrw sscratch, sp           # 暂存当前sp到sscratch
addi sp, sp, -36*REGBYTES   # 在内核栈上分配异常帧
# 保存除x2外的所有通用寄存器
csrrw s0, sscratch, x0      # 原子操作：读取原始sp到s0，清空sscratch
# 读取其他CSR到s1-s4
STORE s0, 2*REGBYTES(sp)    # 将原始sp保存到异常帧的x2位置
```

5、为什么保存stval/scause但不还原？存储的意义何在？

保存 `stval` / `scause` 但不还原它们，是因为它们属于“一次性”的状态寄存器，其核心作用是向操作系统描述“刚刚发生了什么”，而不是一个需要保持的配置。存储它们的意义，在于为高级语言编写的异常处理程序提供决策所需的信息。

具体来说：

1. 保存的目的：为C处理函数提供信息 在异常发生的瞬间，硬件会自动将异常原因写入 `scause`，将附加信息（如出错的内存地址）写入 `stval`。汇编入口代码将它们保存到栈上的“陷阱帧”中，是为了让后续用C语言编写的 `trap()` 函数能够安全地读取这些信息。C函数无法直接、高效地访问CSR，通过陷阱帧传递是标准做法。处理函数正是根据 `scause` 来判断这是系统调用、页错误还是时钟中断，并根据 `stval` 来获取具体的错误地址。

2. 不还原的原因：它们是只读的瞬时状态 `scause` 和 `stval` 是硬件设置的**只读**寄存器，它们记录的是一个已经发生的异常事件的属性。当CPU执行 `sret` 指令返回时，这个异常事件已经被处理完毕，它们的使命也就结束了。在返回前将它们还原到CPU是毫无意义的，因为：

- 这些值无法影响返回操作：`sret` 指令只关心 `sepc`（返回地址）和 `sstatus`（状态）。
- 它们会被自动覆盖：当下一次异常发生时，硬件会自动用新的值覆盖它们，旧值没有保留的必要。

简而言之，这是一个“只读不写”的设计：保存是为了“询问”硬件发生了什么，不还原是因为这些信息已经“用过即弃”。我们只还原那些真正控制CPU执行流和状态的寄存器（如 `sepc`, `sstatus` 和通用寄存器），以确保程序能正确回到中断点继续执行。

6、关于递归异常/内核栈安全性的补充说明

（1）递归异常检测机制

在异常处理入口处，sscratch寄存器被设置为0，这一操作建立了明确的"内核异常处理状态"标志。该标志的核心作用是区分异常的来源：当sscratch值为0时，表明系统当前已经处于内核异常处理过程中，新发生的异常属于递归异常（即内核态内部触发的异常）；当sscratch值非0时，表明异常来自用户态，属于正常的异常入口场景。

（2）递归异常处理策略

检测到递归异常后，系统必须采取特殊处理措施。标准处理流程包括：

- 立即停止使用当前内核栈，避免破坏已有的异常帧和上下文信息
- 切换到预先分配的紧急栈空间继续处理，或者直接触发系统panic
- 确保递归异常不会导致关键内核数据的损坏或丢失 这种保守策略保障了系统在复杂异常场景下的稳定性，防止因栈空间重复使用而引发的状态混乱。

（3）多核与多进程环境下的扩展应用

在多处理器和多进程环境中，sscratch寄存器的使用需要进一步扩展：

- 每个硬件线程独立维护自己的sscratch寄存器值

- 在进程上下文切换时，操作系统需要更新sscratch以指向新进程的内核栈指针
- 这种设计确保了异常返回时能够准确恢复到对应进程的正确内核栈环境 通过这种灵活的sscratch管理机制，系统能够在复杂的多任务环境下维持正确的异常处理语义，保证每个进程和每个处理器核心都能获得独立的、正确的异常处理上下文。

扩展练习Challenge3：完善异常中断

代码实现

- 在`kern/trap/trap.c`文件的`exception_handler`函数中，针对非法指令异常和断点异常进行了如下实现

```
case CAUSE_ILLEGAL_INSTRUCTION:// 非法指令异常处理
/* LAB3 CHALLENGE3   2312711 2314003 2314081 :  */
/*(1)输出指令异常类型（ Illegal instruction）
             *(2)输出异常指令地址
             *(3)更新 tf->epc寄存器*/
             cprintf("Exception type:Illegal instruction\n");
             cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);
             // 更新epc寄存器，跳过当前非法指令
             tf->epc += 4;
             break;
case CAUSE_BREAKPOINT://断点异常处理
/* LAB3 CHALLLENGE3   2312711 2314003 2314081 :  */
            /*(1)输出指令异常类型（ breakpoint）
             *(2)输出异常指令地址
             *(3)更新 tf->epc寄存器*/
            cprintf("Exception type: breakpoint\n");
            cprintf("ebreak caught at 0x%08x\n", tf->epc);
            // 更新epc寄存器，跳过断点指令
            tf->epc += 2;
            break;
```

非法指令异常处理

首先输出异常类型：`"Exception type:Illegal instruction"`之后输出异常地址：`"Illegal instruction caught at 0x%08x"`，其中`%08x`为`tf->epc`的值最后更新EPC寄存器：`tf->epc += 4`，跳过当前非法指令，继续执行下一条指令

断点异常处理

同样我们先输出异常类型：`"Exception type: breakpoint"`然后输出异常地址：`"ebreak caught at 0x%08x"`，其中`%08x`为`tf->epc`的值。最后更新EPC寄存器：`tf->epc += 2`，跳过断点指令，继续执行.

注意：RISC-V指令集是可变长编码，这里的ebreak指令的编码长度是2字节（16位），所以这里应该是加2.

测试代码：之后我们需要在kern/init/init.c中写入如下两行代码来进行测试

```
asm("mret");
asm("ebreak");
```

`asm("mret");` 是 RISC-V 架构中的机器模式异常返回指令，用于从异常、中断或系统调用处理程序返回到被中断的代码位置继续执行。该指令会恢复之前保存的程序计数器、处理器状态和特权级别，是异常处理流程的收尾操作。

`asm("ebreak");` 是环境断点指令，用于主动触发一个断点异常，通常被调试器用于设置软件断点，或在程序中进行自我调试检查，使处理器陷入异常处理程序以便进行调试干预。

运行结果

程序成功捕获了非法指令异常和断点异常，并输出正确的异常类型和触发地址信息。在这之后正常恢复执行，没有陷入异常循环