Lab0-Lab1

Challenge 1-3

Q：描述ucore中处理中断异常的流程（从异常的产生开始），其中mov a0，sp的目的是什么？SAVE_ALL中寄寄存器保存在栈中的位置是什么确定的？对于任何中断，__alltraps 中都需要保存所有寄存器吗？请说明理由。

A：产生，跳到stvec保存地址处，进入alltraps,然后保存寄存器，之后jal跳到traps函数处理中断，跳回来后恢复寄存器，然后返回即可。

mov a0 sp的作用是把sp地址放到a0（传参寄存器），trap的唯一参数是指针类型，这样就把trapframe结构体传进去了。

由结构体定义的顺序。

不需要！单一的中断并不需要都保存，但是嵌套中断的话就需要都保存，因为可能在后续中断中被修改。

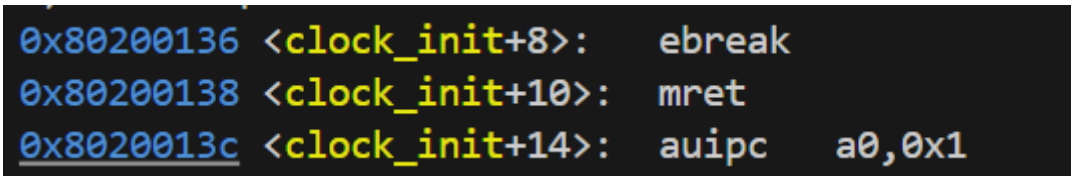
Q：在trapentry.S中汇编代码 csrw sscratch, sp；csrrw s0, sscratch, x0实现了什么操作，目的是什么？save all里面保存了stval scause这些csr，而在restore all里面却不还原它们？那这样store的意义何在呢?

通过sscratch暂存sp的值，并存到s0保留寄存器里，同时sscratch写入0，目的是便于重设当前状态，便于递归发生异常的时候识别内核态。

cause和badaddr确实没有恢复，这些是辅助异常处理的寄存器，处理完后它们的内容就无关紧要了，也就不必恢复了。

Q：编程完善在触发一条非法指令异常 mret和，在 kern/trap/trap.c的异常处理函数中捕获，并对其进行处理，简单输出异常类型和异常指令触发地址

这个的检查关键点在epc的更新上，应该认识到ebreak是2



我们查看对应地址发现，ebreak的指令确实长度为2字节，我们在epc更新的时候应该+2而不是+4.

非法指令异常的话更新+4就可以了。

7个问题

1.makefile中跟架构有关的指令

架构就是指riscv64，所以我们只需要找到makefile中所有带riscv64的字符就可以了，一共是3处，下面分别进行解释：

第一处：

```
GCCPREFIX := riscv64-unknown-elf-
```

PREFIX的作用是配置安装的路径，加上前缀GCC，指的就是GCC编译的前缀，意思就是在GCC前面加上`riscv64-unknown-elf-`

第二处：

```
QEMU := qemu-system-riscv64
```

：=的意思就是赋值，类似于编译系统原理。该行的意思就是用`qemu-system-riscv64`代表`QEMU`

第三处：

```
LDFLAGS := -m elf64lriscv
```

- `LDFLAGS`的作用是进行链接
- `:=`是makefile文件中的赋值操作符
- `-m`的含义是，指定链接器的目标架构
- `elf64lriscv`是指定了生成的ELF文件的架构是64位RISC-V架构的

2.cprintf为什么可以传递可变参数

我们找到该函数

```
int cprintf(const char *fmt, ...) {
    va_list ap;
    int cnt;
    va_start(ap, fmt);
    cnt = vcprintf(fmt, ap);
    va_end(ap);
    return cnt;
}
```

我们发现，`va_list ap`的含义是定义了ap的类型为`va_list`，`va_list`的作用就是**传递可变长度列表**

3.opensbi运行在哪一级

运行在M态

因为opensbi在使用的时候需要使用硬件和软件，所以opensbi为固件

因为固件需要直接访问硬件，所以opensbi运行在M态

4.特权级一共有几种

RISCV有四种**特权级（privilege level）**。但是编码为10的等级目前还没有得到使用。

- 等级0，编码为00，简称U-mode，是用户程序和应用程序的特权级
- 等级1，编码为01，简称S-mode，是操作系统内核的特权级
- 等级2（未得到使用）
- 等级3，编码为11，简称M-mode，是固件的特权级

Level	Encoding	全称	简称
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved(目前未使用，保留)	
3	11	Machine	M

5.qemu频率

$$T = 10^{-7}$$

$$f = 10^{7}$$

代码部分在`lab1\kern\driver\clock.c`

```
static uint64_t timebase = 100000;
void clock_init(void) {
    set_csr(sie, MIP_STIP);
    clock_set_next_event();
    ticks = 0;
    cprintf("++ setup timer interrupts\n");
}
void clock_set_next_event(void) { sbi_set_timer(get_cycles() + timebase); }
```

我们首先设置时钟中断，timer的值每加一次timebase，就会触发一次时钟中断，对于QEMU来说，timer每增加1，就代表时间过去了10^{-7}秒，也就是100ns，所以频率为10的7次

6.内部中断和外部中断

- 外部中断通常是由外部设备（如鼠标、键盘、磁盘驱动器等）或外部事件（如定时器、硬件故障）等触发，如键盘中断、时钟中断；通常需要操作系统的设备驱动程序处理。外部中断是异步的，可以延时处理。
- 内部中断由CPU内部事件触发，如算数溢出、除零错误、非法指令、缺页等；通常由操作系统的异常处理程序处理。内部中断是同步的，必须立即处理。

——所有中断都可以恢复吗?

并非所有中断都可以恢复。外部硬件中断、系统调用中断、缺页中断等，在操作系统处理完这些中断任务后，可以恢复到原来的状态继续执行；而如硬件故障、除零中断这些，在硬件或程序方面本身存在问题，无法继续执行，因此不可恢复。

7.上下文保存恢复机制

进入中断入口点后，首先保存上下文，然后传参并调用中断处理函数进行中断处理，处理结束后再返回并恢复上下文。

- **保存上下文**：将需要保存的32个通用寄存器和4个相关的csr整合成一个结构体trapFrame（该结构体占据36个寄存器大小的空间），定义一个汇编宏SAVE_ALL，将这个trapFrame存到栈顶。

- **调用中断处理程序**：将栈顶指针作为参数传递给trap函数（即中断处理程序），栈顶指针sp中实际就是这个trapFrame，传参以便中断处理时访问。中断处理结束后返回该处，执行恢复上下文的操作。
- **恢复上下文**：定义一个汇编宏RESTORE_ALL，将保存到栈上的寄存器值逆序一一恢复到寄存器中。但是不必恢复所有的csr（如scause、abadaddr），因为它们只在中断处理过程中有用，处理结束后不再需要。

最终退出中断处理（一般用sret指令）。