Lab 2: RV64 时钟中断处理

学号: 3190105838

姓名: 范钊瑀

4.1 准备工程

在4.1步骤中我们需要将repo中最新的 `stddef.h` 等文件同步到lab1实现的工程中，同时为了达到原来相同效果的输出，我们也需要更改main函数中原本对 `puti` ， `puts` 的调用。

此外，我们需要对 `vmlinux.lds` 以及 `head.S` 文件进行更改，以实现我们的中断服务：

- 需要在 `vmlinux.lds` 文件中加入 `init` 部分，这部分取代了之前的 `entry` 部分，用于存放os的启动内核代码，而原先的 `entry` 部分则用于存放中断处理指令。

```
*(.text.init)
```

- 也要对 `head.S` 存放的位置进行更改，如下所示：

```
.section .text.init
```

4.2 开启异常处理

这一部分是对 `head.S` 进行更改，在这一阶段，我们需要对指定的寄存器进行赋值，以实现正常的中断功能。

首先是要设置 `stvec` 寄存器，它是"中断向量表基址"，存储了中断处理程序的地址，我们本次实验统一采用Direct模式，因此该寄存器中存储的就是 `_traps` 的地址，因此我们可以采用以下的语句先将结果赋到x10中，再利用csr指令将x10的结果赋值到 `stvec` 寄存器中。

```
la x10, _traps
csrw stvec, x10
```

之后我们需要对中断使能进行开启，也即改变 `sie` 寄存器中的 `STIE` 位，由下图我们可以知道，我们需要将 `sie` 的第五位置为1，因此可以同样将值先存到x10中，再通过 `csrs` 指令进行数据的写入。

4.1.5 Supervisor Interrupt Registers (`sip` and `sie`)

The `sip` register is an SXLEN-bit read/write register containing information on pending interrupts, while `sie` is the corresponding SXLEN-bit read/write register containing interrupt enable bits.

SXLEN-1 … 10	9	8	7 … 6	5	4	3 … 2	1	0
WPRI	SEIP	UEIP	**WPRI**	STIP	UTIP	**WPRI**	SSIP	USIP
SXLEN-10	1	1	2	1	1	2	1	1

Figure 4.4: Supervisor interrupt-pending register (`sip`).

SXLEN-1 … 10	9	8	7 … 6	5	4	3 … 2	1	0
WPRI	SEIE	UEIE	**WPRI**	STIE	UTIE	**WPRI**	SSIE	USIE
SXLEN-10	1	1	2	1	1	2	1	1

Figure 4.5: Supervisor interrupt-enable register (`sie`).

具体实现代码如下：

```
li   x10, (1 << 5)
csrs sie, x10
```

再之后我们需要设置第一次中断，由于我们已经写好了 `clock_set_next_event` 函数，我们可以通过直接调用来设置开始一秒后的中断：

```
call clock_set_next_event
```

和sie寄存器类似，我们也需要通过csr指令来对 `sstatus` 寄存器进行赋值，有关 `sstatus` 寄存器中每一位所代表的功能如下：

31	30 … 20	19	18	17	16 … 15	14 … 13	12 … 9	8	7 … 6	5	4	3 … 2	1	0
SD	**WPRI**	MXR	SUM	**WPRI**	XS[1:0]	FS[1:0]	**WPRI**	SPP	**WPRI**	SPIE	UPIE	**WPRI**	SIE	UIE
1	11	1	1	1	2	2	4	1	2	1	1	2	1	1

Figure 4.1: Supervisor-mode status register (`sstatus`) for RV32.

SXLEN-1 … SXLEN-2	34 … 33	32 … 31	30 … 20	19	18	17
SD	**WPRI**	UXL	**WPRI**	MXR	SUM	**WPRI**
1	SXLEN-35	2	12	1	1	1

16 … 15	14 … 13	12 … 9	8	7 … 6	5	4	3 … 2	1	0
XS[1:0]	FS[1:0]	**WPRI**	SPP	**WPRI**	SPIE	UPIE	**WPRI**	SIE	UIE
2	2	4	1	2	1	1	2	1	1

Figure 4.2: Supervisor-mode status register (`sstatus`) for RV64.

可以发现 `SIE` 为 `sstatus` 寄存器的第1位，因此我们可以通过以下的赋值语句进行：

```
li   x10, (1 << 1)
csrs sstatus, x10
```

将这些寄存器赋值完毕后，我们便可以跳转到内核的启动模块，即执行以下的语句：

```
jal x0, start_kernel
```

4.3 实现上下文切换

这里我们需要操作 `stack pointer` 进行压栈操作，以保存进入中断前所有寄存器的值，以便处理完毕中断后能够恢复“事发现场”，在下面的代码中，我们先分配栈空间，将寄存器依次保存到栈空间中，同时也要对sepc进行保存，这里存储了返回之后继续执行的地址，因此保存的时候需要注意不能将其丢弃。

```
addi sp, sp, -248
addi sp, sp, -8

sd x0,  248(sp)
sd x1,  240(sp)
sd x3,  232(sp)
sd x4,  224(sp)
sd x5,  216(sp)
sd x6,  208(sp)
sd x7,  200(sp)
sd x8,  192(sp)
sd x9,  184(sp)
sd x10, 176(sp)
sd x11, 168(sp)
sd x12, 160(sp)
sd x13, 152(sp)
sd x14, 144(sp)
sd x15, 136(sp)
sd x16, 128(sp)
sd x17, 120(sp)
sd x18, 112(sp)
sd x19, 104(sp)
sd x20, 96(sp)
```

```
sd x21, 88(sp)
sd x22, 80(sp)
sd x23, 72(sp)
sd x24, 64(sp)
sd x25, 56(sp)
sd x26, 48(sp)
sd x27, 40(sp)
sd x28, 32(sp)
sd x29, 24(sp)
sd x30, 16(sp)
sd x31, 8(sp)

csrrs x10, sepc, x0 # get sepc to x10
sd x10, 0(sp)
```

在save阶段结束后，我们需要将 `scause` 和 `sepc` 放在函数参数的位置，并且调用 `trap_handler` 以进入中断处理程序。需要注意的是，在RISC-V中，从x10开始是函数调用时的默认参数寄存器。

```
csrr x10, scause
csrr x11, sepc
call trap_handler
```

在程序从trap恢复后，我们需要做的是恢复trap发生之前的现场，也就是将原先保存的寄存器依次恢复到正确的位置处：

```
ld x10, 0(sp)
csrrw x0, sepc, x10 # restore sepc

ld x0,  248(sp)
ld x1,  240(sp)
ld x3,  232(sp)
ld x4,  224(sp)
ld x5,  216(sp)
ld x6,  208(sp)
ld x7,  200(sp)
ld x8,  192(sp)
ld x9,  184(sp)
ld x10, 176(sp)
ld x11, 168(sp)
ld x12, 160(sp)
ld x13, 152(sp)
ld x14, 144(sp)
```

```
    ld x15, 136(sp)
    ld x16, 128(sp)
    ld x17, 120(sp)
    ld x18, 112(sp)
    ld x19, 104(sp)
    ld x20, 96(sp)
    ld x21, 88(sp)
    ld x22, 80(sp)
    ld x23, 72(sp)
    ld x24, 64(sp)
    ld x25, 56(sp)
    ld x26, 48(sp)
    ld x27, 40(sp)
    ld x28, 32(sp)
    ld x29, 24(sp)
    ld x30, 16(sp)
    ld x31, 8(sp)

    addi sp, sp, 248
    addi sp, sp, 8
```

之后再利用 `sepc` 的值来从trap中返回正常执行的阶段，我们处于supervisor模式，因此使用：

```
    sret
```

4.4 实现异常处理函数

这里的 `trap_handler` 函数需要接受传入的 `scause` 和 `sepc` 值，对不同类型的中断进行处理，因此我们首先需要清楚 `scause` 寄存器中每一位存储的信息：

4.1.10 Supervisor Cause Register (scause)

The scause register is an SXLEN-bit read-write register formatted as shown in Figure 4.9. When a trap is taken into S-mode, scause is written with a code indicating the event that caused the trap. Otherwise, scause is never written by the implementation, though it may be explicitly written by software.

The Interrupt bit in the scause register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception. Table 4.2 lists the possible exception codes for the current supervisor ISAs. The Exception Code is a **WLRL** field, so is only guaranteed to hold supported exception codes.

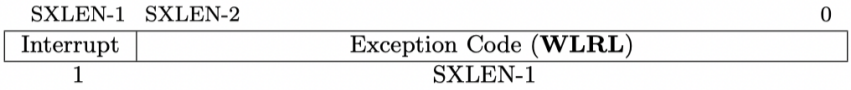

Figure 4.9: Supervisor Cause register scause.

首先为了得到是否为 `Interrupt` 的信息，我们需要获得 `scause` 寄存器的最高位，因此我们可以利用 `dec2bit` 函数，利用简单的移位实现。

```
int dec2bit(unsigned long num, int index) {
    return (num>>(index-1)) & 1;
}
```

在处理完最高位是否为 `Interrupt` 之后，我们需要得到当前的 `Exception Code`，因此需要继续解码：

```
void trap_handler(unsigned long scause, unsigned long sepc) {
    // 通过 `scause` 判断trap类型
    // 如果是interrupt 判断是否是timer interrupt
    // 如果是timer interrupt 则打印输出相关信息, 并通过
`clock_set_next_event()` 设置下一次时钟终端
    // `clock_set_next_event()` 见 4.5 节
    // 其他interrupt / exception 可以直接忽略

    // # YOUR CODE HERE
    // scause 最高位1 代表是interrupt
    int interrupt = dec2bit(scause, 64);
    if(interrupt == 1){
        unsigned long exception_code = scause - (1UL << 63);
        if(exception_code == 5){
            printk("[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
        }
    }
}
```

如果`exception_code`是5，也就代表了`Supervisor timer interrupt`。数字5可以从下表获得：

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

4.5 实现时钟中断相关函数

`clock.c`中需要实现获取当前时间戳的`get_cycles ( )`函数和利用`sbi_ecall`进行下一次中断设置的`clock_set_next_event ( )`函数。
`get_cycles ( )`函数使用内联汇编，利用`rdtime`指令将时间获取并返回。
`clock_set_next_event ( )`函数对`sbi_ecall`的调用中，需要注意的是，`Function ID`和`Extension ID`都是0，见下表：

Function Name	Function ID	Extension ID
sbi_set_timer （设置时钟相关寄存器）	0	0x00
sbi_console_putchar （打印字符）	0	0x01
sbi_console_getchar （接收字符）	0	0x02
sbi_shutdown （关机）	0	0x08

```
unsigned long get_cycles() {
    // 使用 rdtime 编写内联汇编，获取 time 寄存器中（也就是mtime 寄存器 )的值并返回
    // # YOUR CODE HERE
    unsigned long _time;
    __asm__ volatile (
        "rdtime %[time]\n"
        :[time] "=r" (_time)
```

```
        :
        :
    );
    return _time;
}

void clock_set_next_event() {
    // 下一次 时钟中断 的时间点
    unsigned long next = get_cycles() + TIMECLOCK;

    // 使用 sbi_ecall 来完成对下一次时钟中断的设置
    // # YOUR CODE HERE
    printk("kernel is running!\n");
    sbi_ecall(0x0, 0x0, next, 0, 0, 0, 0, 0);
}
```

4.6 编译及测试

在执行 `make run` 后，工程能够正常编译并且每隔一秒将中断设置与内核运行状态打印到屏幕上。

思考题

1.

尽管所有权限级别的中断都可以在Machine Mode处理，但是根据种类，有些可以被委托给较低权限的模式处理以提高性能，这两个寄存器表示某些异常和中断应该由较低的特权级别直接处理：

`mideleg` :machine interrupt delegation register

因为 MIDELEG[15:0] == 16'b0000_0010_0010_0010

即　MIDELEG[9] == MIDELEG[5] == MIDELEG[1] == 1'b1 那么根据下表，这就意味着这三种Interrupt：Supervisor software interrupt, Supervisor timer interrupt, Supervisor external interrupt 被委托给S-Mode执行，也即只能够在S-Mode执行，如果在更高权限的模式，如M-Mode产生了这三种Interrupt，那么中断不会执行。

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

`medeleg` :machine exception delegation register

MEDELEG[15:0] == 16'b1011_0001_0000_1001

MEDELEG[15] == MEDELEG[13] == MEDELEG[12] == MEDELEG[8] == MEDELEG[3] == MEDELEG[1] == 1'b1

The meaning of `medeleg` is similar to the previous one, which means that the following exceptions with the corresponding Exception code must be handled in S-Mode.

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*