# Lab3-Extra-Ov

## 问题描述

在 Lab3 的课下，我们主要介绍了 0 号异常（即时钟中断），并使用了 TLB 异常；在本次 Extra 中，我们希望大家实现 12 号异常 Ov 的处理函数。

在 See-MIPS-Run-Linux 中对 Ov 的描述如下：

| 异常号 | 助记符 | 描述 |
| --- | --- | --- |
| 12 | Ov | 自陷形式的整数算术指令（比如说 `add` 但 `addu` 不会）导致的溢出。<br>C 语言中 `unsigned` 类型间的运算不会使用溢出-自陷指令。 |

以下指令在发生算术溢出时，都会触发 Ov 异常：

| 指令 | 语法 | Encoding | 指令说明 |
| --- | --- | --- | --- |
| add | add $d, $s, $t | 0000 00ss ssst tttt dddd d000 0010 0000 | $d = $s + $t |
| sub | sub $d, $s, $t | 0000 00ss ssst tttt dddd d000 0010 0010 | $d = $s - $t |
| addi | addi $t, $s, imm | 0010 00ss ssst tttt iiii iiii iiii iiii | $t = $s + imm |

例如， add $8, $9, $10 的二进制编码为 0000 0001 0010 1010 0100 0000 0010 0000.

在本题中，你需要添加 Ov 异常的处理函数，实现以下的处理方式：

- 若异常由 $add$ 指令触发，则将发生异常的指令替换为 $addu$ 指令，并执行语句 `printk("add ov handled\n");`
- 若异常由 $sub$ 指令触发，则将发生异常的指令替换为 $subu$ 指令，并执行语句 `printk("sub ov handled\n");`
- 若异常由 $addi$ 指令触发，则将异常现场中记录的 $t 寄存器的值改为 $s/2 + imm/2 （这里 / 表示无符号整数除法），EPC 的值改为 EPC + 4，并执行语句 `printk("addi ov handled\n");`

注意上述中的异常处理函数需要完成的输出内容不包含双引号而且应该单独成行，并注意避免其他非必要的输出，否则将影响评测结果。

同时，你还需要记录每个进程发生 Ov 异常的次数。请在 $include/env.h$ 中定义的 `struct Env` 结构体的末尾增加成员 `env_ov_cnt` ，并在每次进入 Ov 异常处理函数时执行 `curenv->env_ov_cnt++;` 来维护当前进程发生异常的次数。

此外，由于评测需要，你还需要将 $kern/env.c$ 中定义的 `pre_env_run` 函数（可以在 $Vim$ 中用 `/pre_env_run` 命令来查找）中的语句 `printk("env %08x reached end pc: 0x%08x, $v0=0x%08x\n", e->env_id, epc, tf->regs[2]);` 改为 `printk("env %08x reached end pc: 0x%08x, $v0=0x%08x, env_ov_cnt=%d\n", e->env_id, epc, tf->regs[2], e->env_ov_cnt);` 以此来输出进程结束前总共发生过的 Ov 异常次数。

可能使用到的指令补充：

| 指令 | 语法 | Encoding | 指令说明 |
| --- | --- | --- | --- |

| 指令 | 语法 | Encoding | 指令说明 |
| --- | --- | --- | --- |
| addu | addu $d, $s, $t | 0000 00ss ssst tttt dddd d000 0010 0001 | $d = $s + $t |
| subu | subu $d, $s, $t | 0000 00ss ssst tttt dddd d000 0010 0011 | $d = $s - $t |

# 提示

在完成异常处理函数时需要注意：

- 可以利用 CP0 中 EPC 寄存器（对应 Trapframe 结构体中的成员 cp0_epc）保存的值，获取发生异常的指令所在的虚拟地址；
- 在Lab2中我们知道， CPU 进行访存时，对于位于 kuseg 区间的虚拟地址需要通过 TLB 来获取物理地址。在 `elf_load_seg` 函数加载ELF文件时，用户程序被加载到了用户空间 kuseg，由于我们设置了程序中保存操作指令代码的.text节权限为只读，这部分空间在页表中仅被映射为 PTE_V，而不带有 PTE_D 权限。如果我们直接利用 EPC 中的虚拟地址去尝试修改指令，由于对应 TLB 项不带有 PTE_D 位，会引发 Lab2 介绍 EntryLo 提及的 TLB 写入异常，内核目前不能处理这种情况。对于该问题，我们推荐用以下思路回避 TLB 机制修改指令：

  1. 通过查询 curenv 的页表，获得用户虚拟地址对应的物理地址
  2. 将该物理地址转化至 kseg0 区间中对应的虚拟地址从而可以直接去访存

  步骤（1）（2）中涉及的地址转化，均可使用既有的函数去实现。
- addi 溢出处理函数需要修改保存现场的 Trapframe 结构体（定义在 include/trap.h 中）中记录的对应通用寄存器的值，这样才能保证从异常返回恢复现场后目的寄存器的值是我们规定的值；

注意异常处理函数中要求的用 printk 输出的内容！

注意对记录异常处理次数输出的修改要求！

# 样例输出 & 本地测试

对于创建运行如下代码的进程:

```
int main() {
    unsigned int src1, src2, dst;
    /*
     * 测试 add
     */
    src1 = 0x80000000;
    src2 = 0x80000000;
    asm("add %0,%1,%2\n\t"
         : "=r"(dst)             /* 输出操作数，也是第0个操作数 %0 */
         : "r"(src1), "r"(src2) /* 输入操作数，也是第1个操作数和第2个操作数 %1, %2 */
        );
    /* 上面的 asm 执行了一条 add 指令，其中源操作数来自 src1 和 src2，目的操作数将储存到
dst 中。*/
    if (dst != (src1 + src2)) {
        return -1;              //如果异常处理结果不正确进程将运行结束并返回-1
    }

    /*
     * 测试 sub
     */
    src1 = 0x80000000;
```

```
    src2 = 0x70000000;
    asm("sub %0,%1,%2\n\t"
         : "=r"(dst)             /* 输出操作数，也是第0个操作数 %0 */
        : "r"(src1), "r"(src2) /* 输入操作数，也是第1个操作数和第2个操作数 %1, %2 */
        );
    /* 上面的 asm 执行了一条 sub 指令，其中源操作数来自 src1 和 src2 ，目的操作数将储存到
dst 中。*/
    if (dst != (src1 - src2)) {
        return -1;              //如果异常处理结果不正确进程将运行结束并返回-1
    }

    /*
     * 测试 addi
     */
    src1 = 0x7ffffff0;
    asm("addi %0, %1, 20\n\t"
         : "=r"(dst)        /* 输出操作数，也是第0个操作数 %0 */
         : "r"(src1)        /* 输入操作数，也是第1个操作数 %1*/
        );
    /* 上面的 asm 执行了一条 addi 指令，其中源操作数来自 src1 ，目的操作数将储存到 dst 中。
*/
    if (dst != (src1 / 2 + 10)) {
        return -1;              //如果异常处理结果不正确进程将运行结束并返回-1
    }
    return dst;
}
```

应当输出：

```
add ov handled
sub ov handled
addi ov handled
env 00000800 reached
end pc: 0x00400180, $v0=0x40000002, env_ov_cnt=3 [00000800]
free env 00000800
i am killed ...
```

输出说明：

- 前三行表示进程运行中进入了三次 Ov 异常处理函数
  第四行中$v0=0x40000002表示该进程运行结束时的返回值，env_ov_cnt=3表示该进程运行过程中发生过的 Ov 异常次数。
  上述进程中，对于异常处理后的结果进行了判断，如果正确执行了异常处理，进程的最终结束返回值$v0应当为变量dst的最终值，env_ov_cnt应该为进程发生过的异常总数，如果输出中的$v0与变量dst在进程结束最终值不相等或者异常发生次数与预期值不同，则说明异常未能按照题目要求正确处理。

你可以使用：

```
make test lab=3_ov && make run 在本地测试上述样例（调试模式）
MOS_PROFILE=release make test lab=3_ov && make run 在本地测试上述样例（开启优化）
```

# 实验代码提示

本节提示一种可以实现上述功能的代码结构：

- 在kern/genex.S中补充以下代码，用 BUILD_HANDLER 宏来构建自己的异常处理函数 handle_ov，`BUILD_HANDLER ov do_ov`。并在kern/traps.c中仿照其他函数，将该异常处理函数加入异常向量组中的对应位置。
- `void do_ov(struct Trapframe *tf) { // 你需要在此处实现问题描述的处理要求 }`