异常处理程序

eret 没有延迟槽

`eret` 承担了跳转功能，但是 `eret` 是没有延迟槽的。也就是说测试数据中可能出现 `eret` 指令后紧跟另一条非 `nop` 指令的情况。你的设计应该保证 `eret` 的后续指令不被执行。

异常处理程序的结构

异常处理程序是由软件实现的，我们只需要提供接口而无需自己实现。同时，了解异常处理程序是十分有必要的。

异常和中断处理流程可以概括成如下步骤（需要强调的是，这些步骤只是为了让同学们更好的理解处理程序的结构，我们在实际测评中并不保证下述的步骤都执行，也不保证不包含在下述步骤里的结构不出现）：

- Step 1：构造异常处理环境，保存现场。
- Step 2：读取 `Cause` 和 `EPC` 寄存器，判断错误类型。
- Step 3：根据异常类型和其他属性执行对应处理。
- Step 4：恢复现场。
- Step 5：使用 `eret` 指令从异常处理返回。

下面列出了一个简要的发生算数溢出时的程序，同学们可以结合源码进行参考和理解。

```
# 程序首先从这里运行
.text
    # 只允许外部中断
    ori $t0, $0, 0x1001
    mtc0 $t0, $12

    # 算术溢出
    lui $t0, 0x7fff
    lui $t1, 0x7fff
    add $t2, $t0, $t1

end:
    beq $0, $0, end
    nop

.ktext 0x4180
_entry:
    # 保存上下文
```

```
    j _save_context
    nop

_main_handler:
    # 取出 ExcCode
    mfc0 $k0, $13
    ori $k1, $0, 0x7c
    and $k0, $k0, $k1

    # 如果是中断，直接恢复上下文
    beq $k0, $0, _restore_context
    nop

    # 将 EPC + 4，即处理异常的方法就是跳过当前指令
    mfc0 $k0, $14
    addu $k0, $k0, 4
    mtc0 $k0, $14
    j _restore_context
    nop

_exception_return:
    eret

_save_context:
    ori $k0, $0, 0x1000     # 在栈上找一块空间保存现场
    addiu $k0, $k0, -256
    sw $sp, 116($k0)        # 最先保存栈指针
    move $sp, $k0

    # 依次保存通用寄存器（注意要跳过 $sp）、HI 和 LO
    sw $1, 4($sp)
    sw $2, 8($sp)
    # ......
    sw $31, 124($sp)
    mfhi $k0
    mflo $k1
    sw $k0, 128($sp)
    sw $k1, 132($sp)

    j _main_handler
    nop

_restore_context:
    # 依次恢复通用寄存器（注意要跳过 $sp）、 HI 和 LO
    lw $1, 4($sp)
    lw $2, 8($sp)
    # ......
    lw $31, 124($sp)
    lw $k0, 128($sp)
    lw $k1, 132($sp)
    mthi $k0
    mtlo $k1

    # 最后恢复栈指针
    lw $sp, 116($sp)
```

```
    j _exception_return
    nop
```

利用 MARS 验证异常处理框架

尽管在 MARS 中，我们只能针对内部异常进行模拟，无法模拟外部中断。但由我们对内部异常与外部中断的了解可以知道，两者的处理是类似的。因此我们可以在 MARS 中先验证中断/异常处理的框架是否正确（我们可以构造一条产生异常的指令，如溢出，再观察 MARS 能否进入 Exception Handler），至于我们如何处理这个错误，则是次要问题。