体系结构实验 3：例外处理

姓名：禹相祐　学号：2312900　专业：计算机科学与技术

目录

一、 实验目的与要求

本次作业的核心任务是把处理器里关于“异常/例外”的这一条通路真正打通，使得用户程序在执行 `syscall`，或者遇到一条处理器不支持的指令（RI, Reserved Instruction）时，能够自动跳到统一的异常入口，由硬件与软件协同完成记录与恢复。完成后，处理器应当具备下面这些能力：

1. 能在流水线中识别出系统调用指令并产生对应的异常类型。
2. 能在解码阶段发现非支持指令并上报 RI 异常。
3. 在异常发生时，把触发异常的 PC 写入 EPC，同时拉高 Status.EXL，并在 Cause 中写入正确的 ExcCode。
4. 异常向量入口地址不再写死，而是从 CP0 的 EBase 中取得，便于软件后期修改异常入口位置。

5. 执行 `eret` 时能够从 EPC 中取回原程序位置，并把 EXL 复位，使得后续异常还能继续被响应。

6. 引发异常的那条指令本身不能对寄存器或存储器产生实际写回, 以免把错误结果提交出去。

二、 例外实现原理

在一条典型的 MIPS 五级流水里，异常处理往往遵循“前级发现、末级定案”的思路：前面的流水级只负责发现可能的异常并把信息带下去，真正决定“这条指令要不要进异常、要写哪些 CP0 寄存器”是在 WB 末尾完成的。

(一) 基本流程

整体过程可以拆成下面几步：

1. **前端识别**：在 ID/Decode 阶段，解码单元一旦发现这条指令是 `syscall`，或者压根不在支持指令表里，就立刻产生 `syscall` 或 `ri` 标志位。

2. **逐级下传**：这两个标志会随着 ID→EXE、EXE→MEM、MEM→WB 这三段流水寄存器一路往后传，不在中途处理。

3. **WB 阶段处理**：到达 WB 时再统一判断是否需要进入异常，若需要则写 CP0、发冲刷、改 PC。

4. **PC 重定向**：WB 给取指端一个新的 PC 值,若是 `eret` 则回 EPC,若是 `syscall`/RI 则跳到 EBase 加上固定偏移的位置。

(二) CP0 寄存器扩展

我们在 `wb.v` 里增加了对 CP0.EBase 的支持，关键点如下：

- 新增内部寄存器 `cp0r_ebase`，对应 CP0 的 (15,1) 号寄存器地址。
- 上电后给它一个默认值 0，真正的运行地址由软件用 `mtc0` 写进去。
- 后面产生异常跳转时就不再是硬编码地址，而是用 EBase 作为基址，灵活度更高。

(三) 异常入口选择

在 WB 末尾要根据当前到底是哪一类指令来决定下一个 PC：

- 如果当前是 `eret`，说明要从异常返回，那下一个 PC 就是 EPC 的内容；
- 如果是 `syscall` 或 RI 之类的同步异常，就跳到 EBase + offset。

这样所有同步异常都能统一落到同一片异常处理代码里，方便管理。

(四) 写 EPC、Cause、Status

真正使得“异常能进也能出”的，实际上就是下面三个寄存器的配合：

1. **EPC**：记录出问题的那条指令的 PC，方便回去。
2. **Cause.ExcCode**：用来标示是哪一类异常，`syscall` 通常写 8，RI 写 10，其它异常写各自的编码。
3. **Status.EXL**：异常进入时要把 EXL 置位，防止嵌套异常；执行 `eret` 时再清掉。

这三者都正确之后，异常处理流程就能闭环了。

三、 各级流水线改动

(一) ID（decode）阶段

- 在指令解码的同时多做一层判断：是不是 `syscall`，或者是不是一条根本没有实现的指令；如果是，就生成对应的异常标志。
- 因为后面级也要知道这条指令是异常的，所以 ID→EXE 的总线位宽必须加大，把新增的标志位、指令 PC 等一并塞进去。
- 同时把本条指令的 PC 也打包下去，WB 要用它来写 EPC。

(二) EXE/MEM 阶段

- 这两级的任务非常简单，就是把 ID 阶段带下来的异常信息照搬往后送。
- 由于上一段总线位宽改了，这两段的寄存器和拆包逻辑也要同步调整，不然会对不上。

(三) WB 阶段（关键部分）

WB 这里是整个异常流程的落点，要做的事稍微多一点：

1. **禁止写回**：一旦确定这条指令是异常，就把最终的寄存器写使能关掉，防止错误数据写进寄存器堆：

$$rf\_wen = rf\_wen\_raw \wedge WB\_over \wedge \neg(ri)$$

   同理也可以把 `syscall` 一起并进去。

2. **写 CP0**：根据异常类型写 EPC、Cause、Status，同时还要兼容正常的 `mtc0` 写入。
3. **冲刷并跳转**：向前级发 `cancel` 信号，把后面没来得及提交的指令清掉，再给 IF 一个新的 PC，把控制流导向异常入口或 EPC。

四、 Vivado 实验结果

(一) 测试程序说明

为了验证流水线异常通路是否按预期工作，我们编写了如下汇编程序。程序的思路是让处理器依次经历下面几种情况：先正常执行并能 eret 回来，然后在 0x18 处触发第一次 RI，再在 0x1C 处触发第二次 RI，最后在 0x20 处触发一次 syscall。这样就能在仿真波形里依次看到“异常产生 →CP0 写 EPC/EXL→ 再次异常覆盖 → 异常码从 0x0a 变 0x08”这几个关键现象。

Listing 1: 测试程序

```
    .text
    .globl _start

# 0x00: 先做一次 eret 场景，证明能从 EPC 返回
_start:
    addiu $30, $0, 0
    mfc0 $30, $14 # $30 <- EPC
    addiu $30, $30, 4 # EPC = EPC + 4
    mtc0 $30, $14
    eret # 返回到后面我们布置的代码
    nop

# 0x18: 第一次 RI，EPC 应写 0x18，Cause 应是 0x0a
    .org 0x18
ri_first:
    .word 0xff000000 # 非法指令

# 0x1C: 第二次 RI，EPC 应被新地址 0x1c 覆盖
    .org 0x1c
ri_second:
    .word 0xff000000

# 0x20: syscall，异常码应从 0x0a 变为 0x08，EPC=0x20
    .org 0x20
syscall_test:
    mtc0 $0, $15 # EBase=0，便于观察异常入口
    li $2, 0
    syscall
```

这段代码运行起来，流水线应该出现这样的“画面”：

- 当 PC 跑到 **0x18** 时，取到的是一条根本不能解码的指令，所以解码阶段会标记 RI，最终 WB 阶段会把 EPC 写成 **0x18**，把 **Status.EXL** 置 1，把 **Cause.ExcCode** 写成 0x0a。对应我们的第二张图。

- 随后 PC 又会到 **0x1c**，再次取到同一条非法指令，这一次 EPC 不应该还是 0x18，而是要被更新成 **0x1c**，说明“后来的异常能覆盖掉前一次的异常地址”。对应我们的第四张图。
- 再往后 PC 到 **0x20**，执行 `syscall`，这时异常码就不该再是 0x0a，而要变成 **0x08**，同时 EPC 写成 **0x20**，说明这一次是系统调用异常。对应我们的第五、六张图。

下面依次给出仿真图，并说明它们与上面指令的对应关系。

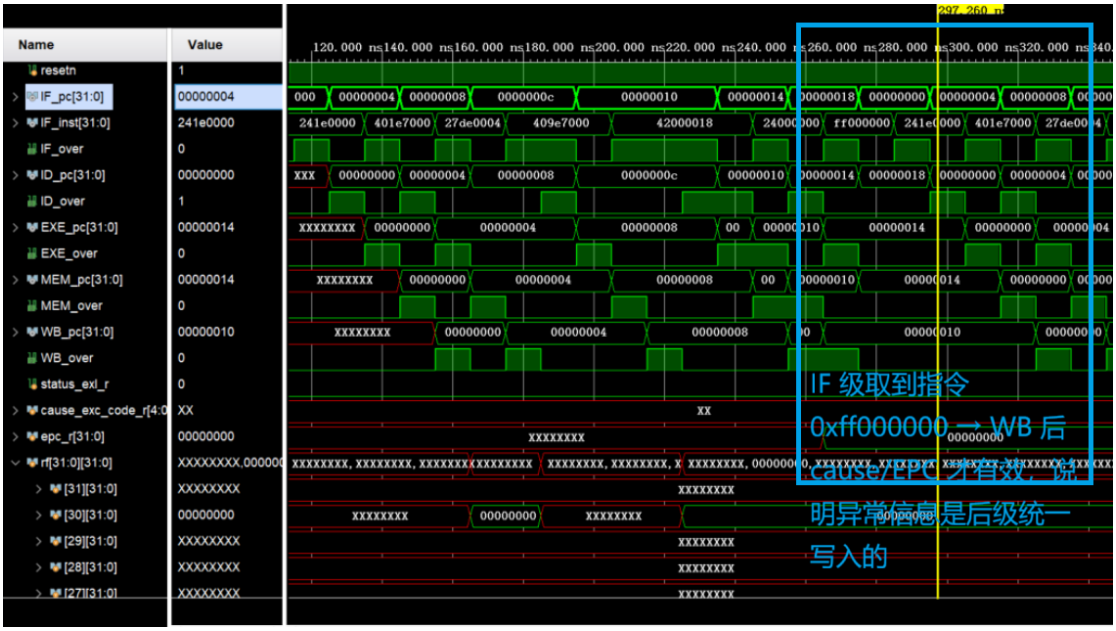

图 1: IF 级取到非法指令后，最终在 WB 阶段才写入 Cause/EPC，说明异常由后级统一处理。

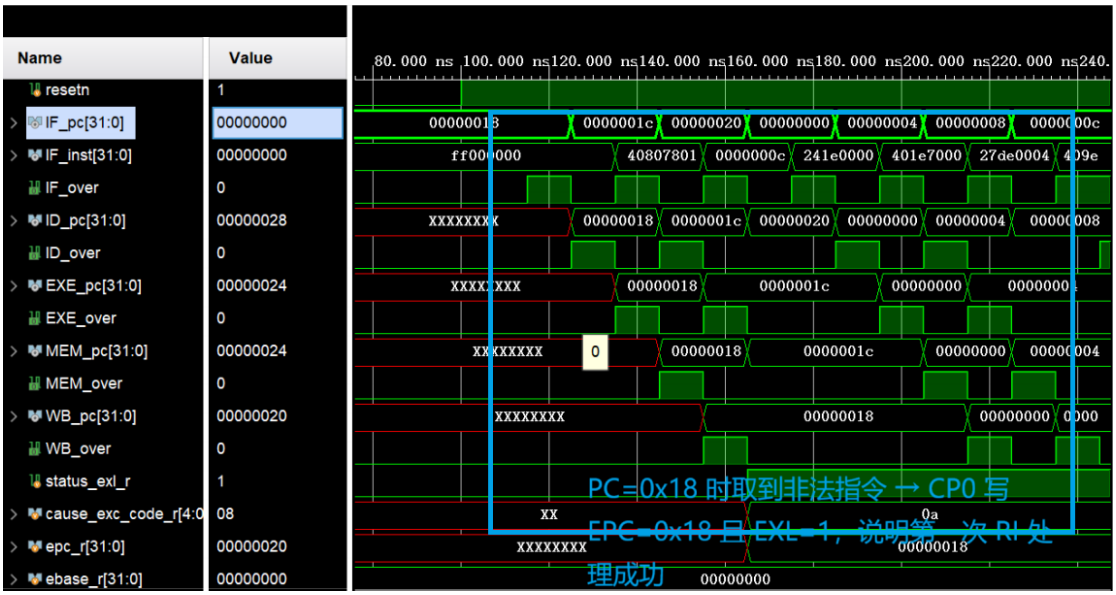

图 2: PC=0x18 时取到非法指令，CP0 写 EPC=0x18 且 EXL=1，说明第一次 RI 触发成功。

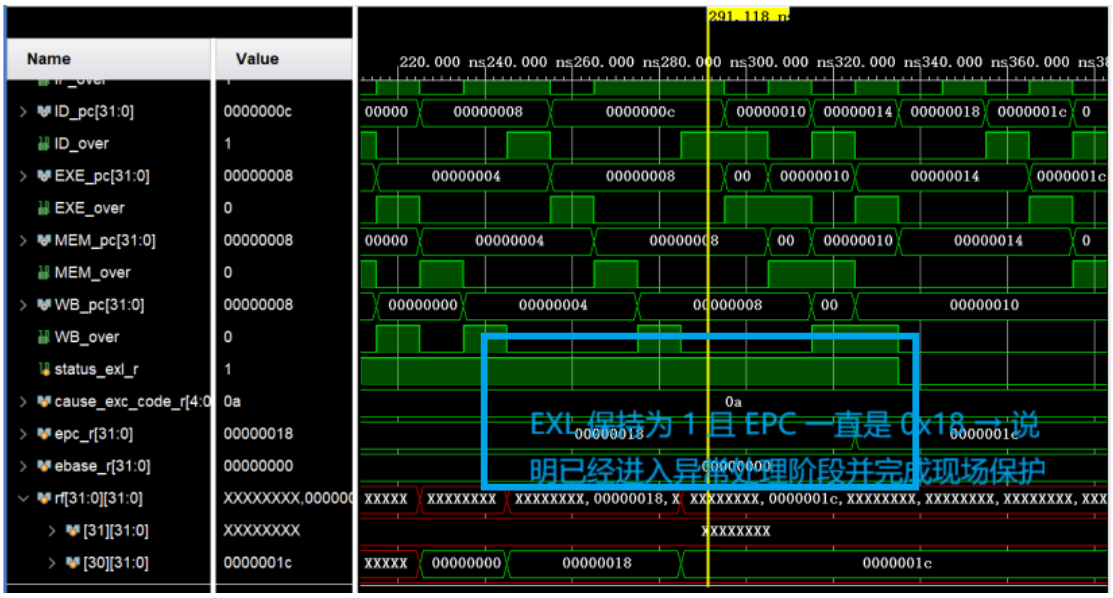

图 3: EXL 一直为 1 且 EPC 保持 0x18，说明已经进入异常服务阶段并完成现场保护。

图 4: 再次取到 0xff000000 后，EPC 由 0x18 更新为 0x1c，说明新的 RI 会覆盖之前记录的异常地址。

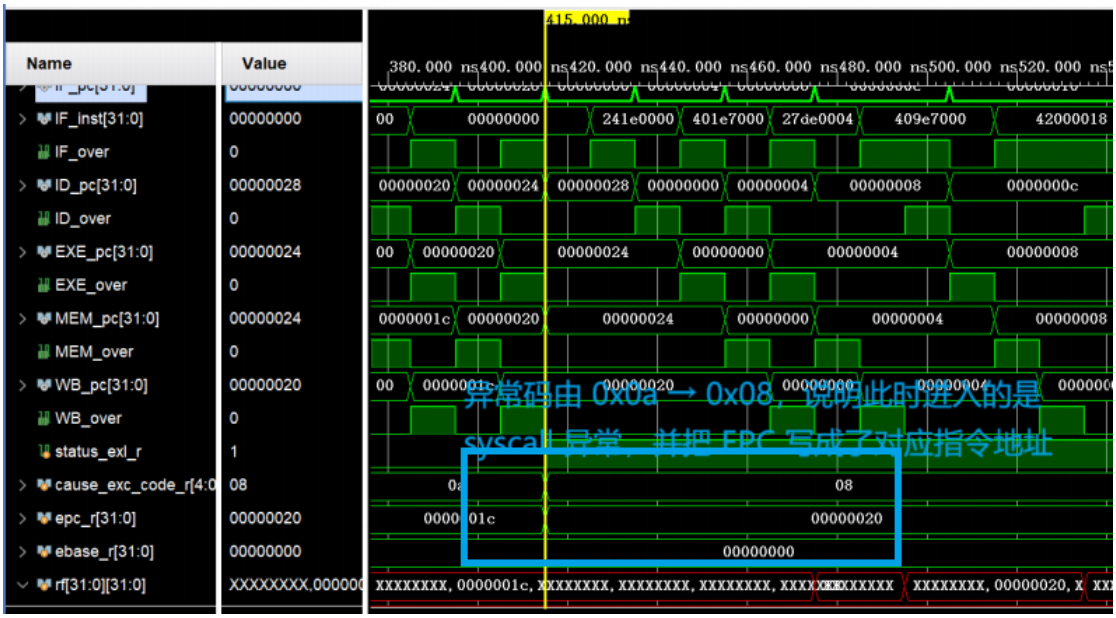

图 5: 异常码从 0x0a 变为 0x08，表明此时触发的是 syscall 异常，并写入对应指令地址。

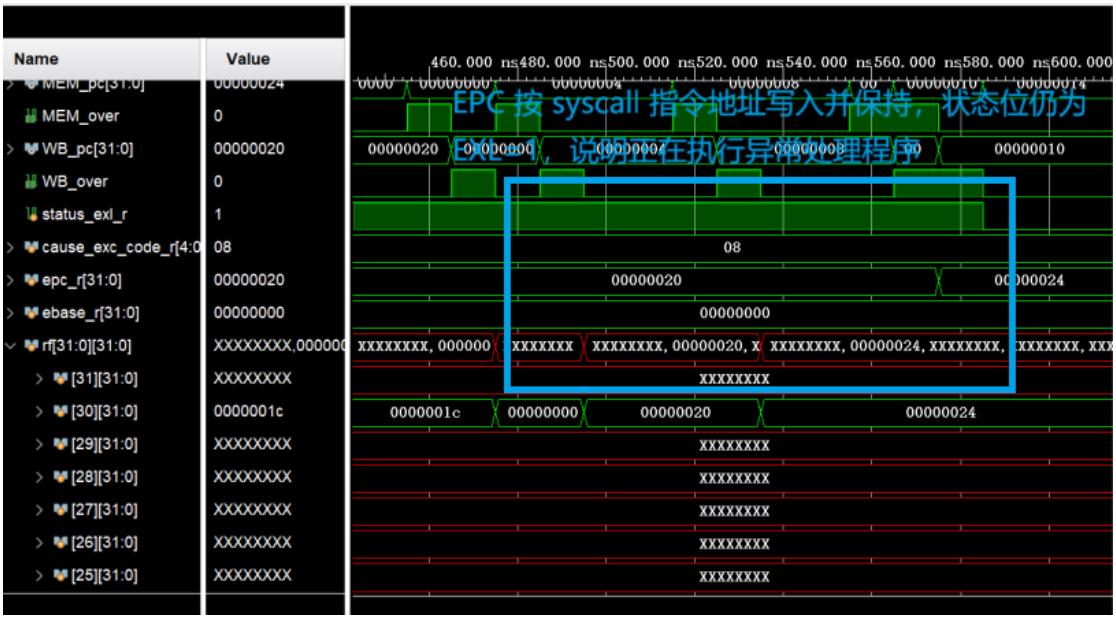

图 6: EPC 按 syscall 指令地址写入并保持，EXL 保持为 1，说明处理器正在执行异常处理程序。

五、 总结

本次实验的核心目标是让流水线在出现 `syscall` 和 RI 时，能够正确进入异常、写入 CP0（EPC/Cause/Status），并能通过 `eret` 返回原指令继续执行。从仿真结果看，异常的发现、传递、写入和返回这几步已经能对上我们设计的测试程序。

碰到的难点主要有两点，如下：

- **异常标志的贯通性问题**：异常是在 ID 被识别、在 WB 被处理的，中间任意一级流水寄存器（ID→EXE、EXE→MEM、MEM→WB）没加字段，WB 就拿不到异常标志，表现就是“取到了非法指令但 EPC 不写”。这个问题通过统一扩大三段总线并固定字段顺序解决。
- **异常后的写回与冲刷问题**：只禁止异常指令本身写回还不够，还要在 WB 判定到异常时同步向前级发 `cancel`，把已经在路上的指令清掉，否则会出现异常已经进入但前一条仍写寄存器的现象。这个问题通过在 WB 统一裁决并广播冲刷信号解决。

综合起来，本次实验完成了：

1. 解码阶段即可识别 `syscall` 和 RI；
2. 异常标志能传到 WB，在同一处写 EPC、Cause、Status；
3. 异常入口从 CP0.EBase 获取，非写死地址；
4. `eret` 能从 EPC 返回并清除 EXL；

5. 异常指令不会写回寄存器堆，流水线状态保持正确。