P7课下-MIPS微系统设计

产生异常

异常指的是**内部异常**和**外部中断**，前者是**由于指令执行错误导致的“事件”**，后者是**由于外部设备信号导致的“事件”**。为了响应异常，CPU 会自动跳转到某一特定地点（修改 PC 值），然后进行异常处理（由软件实现，不属于 MIPS 微系统）。

异常码 ExcCode

通过访问 CP0 的相应寄存器，可以获得 ExcCode 码（关于 CP0 之后再说），在本次 Project 中，需要实现的异常码有：

异常与中断码	助记符与名称	指令与指令类型	描述
0	`Int`（外部中断）	所有指令	中断请求，来源于计时器与外部中断。
4	`AdEL`（取指异常）	所有指令	PC 地址未字对齐。
4	`AdEL`（取指异常）	所有指令	PC 地址超过 `0x3000 ~ 0x6ffc` 。
4	`AdEL`（取数异常）	`lw`	取数地址未与 4 字节对齐。
4	`AdEL`（取数异常）	`lh`	取数地址未与 2 字节对齐。
4	`AdEL`（取数异常）	`lh, lb`	取 Timer 寄存器的值。
4	`AdEL`（取数异常）	Load 型指令	计算地址时加法溢出。
4	`AdEL`（取数异常）	Load 型指令	取数地址超出 DM、Timer0、Timer1、中断发生器的范围。
5	`AdES`（存数异常）	`sw`	存数地址未 4 字节对齐。
5	`AdES`（存数异常）	`sh`	存数地址未 2 字节对齐。
5	`AdES`（存数异常）	`sh, sb`	存 Timer 寄存器的值。
5	`AdES`（存数异常）	Store 型指令	计算地址加法溢出。
5	`AdES`（存数异常）	Store 型指令	向计时器的 Count 寄存器存值。
5	`AdES`（存数异常）	Store 型指令	存数地址超出 DM、Timer0、Timer1、中断发生器的范围。

异常与中断码	助记符与名称	指令与指令类型	描述
8	`Syscall`（系统调用）	`syscall`	系统调用。
10	`RI`（未知指令）	-	未知的指令码。
12	`Ov`（溢出异常）	`add, addi, sub`	算术溢出。

- 异常与中断码为 `Int` 时，表示此时没有**异常**发生，仅考虑**中断**，因此在没有异常产生时，考虑给 CP0 传入 `Int` ，再根据中断信号判断中断，无需考虑异常；
- **除以 0 是未定义行为，不算异常**，即数据中不应该出现除以 0 的情况；
- **优先处理最早可以探查到的异常。**

处理异常中断

在一个时间周期内，

- CP0 先检查现在是否处理异常中断，再检查 CP0 所在流水级指令是否异常，以及外部是否有未禁用的中断（如有，考虑中断处理而非异常）；
- 如果现在没有处理异常中断，且 CP0 所在流水级存在异常或外部产生了未被禁用的中断，CP0 将向 CPU 发送“跳转到异常中断处理”的信号；
- CPU 接收到这个信号后，对 CP0 所在流水级及之前的流水级进行“清空”，让 PC 跳转到异常中断处理地址 `0x4180` ，CP0 记录下处理完异常后需要返回的地址，以及出现异常的指令是否是延迟槽指令。

完成异常中断处理后，通过 `eret` 回到需要返回的地址（存在 CP0 中）。

> 其实CP0就是一个大号的处于M级的分支判断模块，实际上所有工作都是你原先设计的CPU在做，只是CP0这个东西可以帮你保存一些异常中断的状态信息（包含处理完之后回到哪一条指令），以及帮你判断当前时刻是否需要处理一个异常/中断而已。

CP0 （协处理器）的实现

CP0 的任务是**对异常进行配置**，以及**记录异常的信息**，因此需要一些寄存器（32 位）：

寄存器	编号	功能
SR	12	配置异常的功能。
Cause	13	记录异常发生的原因和情况。
EPC	14	记录异常处理结束后需要返回的 PC 。

这些寄存器只有特定的位发挥作用：

寄存器	功能域	位域	解释
SR（State Register）	IM（Interrupt Mask）	15:10	分别对应六个外部中断，1 表示允许中断，0 表示禁止中断。只能通过 `mtc0` 这个指令修改，通过修改这个功能域来屏蔽一些中断。
SR（State Register）	EXL（Exception Level）	1	异常发生时置位，强制进入核心态（进入异常处理程序），禁止中断。
SR（State Register）	IE（Interrupt Enable）	0	全局中断使能，1 表示允许中断，0 表示禁止中断。
Cause	BD（Branch Delay）	31	置 1 表示 EPC 指向当前指令的前一条指令（一定为跳转），否则指向当前指令。
Cause	IP（Interrupt Pending）	15:10	6 位待决的中断位，分别对应 6 个外部中断，1 表示有中断，0 表示无中断，每个周期修改一次，修改的内容来自计时器和外部中断。
Cause	ExcCode	6:2	异常编码，记录当前发生什么异常。
EPC	-	-	记录异常处理结束后需要返回的 PC。

CP0 模块的端口定义如下：

端口	方向	位数	解释
clk	I	1	时钟信号
reset	I	1	复位信号
en	I	1	写使能信号
CP0Add	I	5	寄存器地址
CP0In	I	32	CP0 写入数据
CP0Out	O	32	CP0 读出数据
VPC	I	32	受害 PC
BDIn	I	1	是否是延迟槽指令
ExcCodeIn	I	5	记录异常类型
HWInt	I	6	输入中断信号
EXLClr	I	1	复位 EXL
EPCOut	O	32	EPC 的值
Req	O	1	进入处理程序请求

读写状态寄存器

用来实现 `mfc0, mtc0` 。
CP0 的读写功能使用 `en`，`CP0Add`，`CP0In`，`CP0Out` 端口完成，仿照 GRF。
CP0 的三个状态寄存器仅有部分位有用，对于无用位初始化置 0，后续不对其操作。

进入异常中断处理程序

用来实现异常中断处理。
`VPC, BDIn, ExcCodeIn, HWInt` 是用来判断是否进入异常中断处理程序的所有输入端口。

- CP0 先检查 `EXL` 是否为 1 （正在处理异常中断），再检查 `ExcCodeIn` 是否不为 0 （当前是否存在异常）和 `HWInt & IM` 是否不为 0 且 `IE` 不为 0 （存在未被禁用的中断）；
- 如果现在 `EXL` 为 0，并且 `ExcCodeIn` 不为 0 或“ `HWInt & IM` 不为 0 且 `IE` 不为 0”，那么将 `Req` 置 1，同时记录 `EPC, BD, EXL, ExcCode` 的值，否则置 0；
- CPU 接收到 `Req` 为 1 后，对 CP0 所在流水级及之前流水级清空（之后的流水级正常进行），让 PC 直接跳转到异常中断处理地址 `0x4180` （这是软件决定的功能，即该地址之后的指令是外部写好的，用于处理异常）。

除此之外，只要没有 `reset` 信号，每个时钟周期都要记录一次 `IP` 。

这里的 `VPC` 接入的是“宏观 PC”，即 CPU 在外部表现为“一个有延迟槽的单周期 CPU”，每条指令只有两个状态：执行完了和没执行。由于 CP0 设置在了 M 级，异常处理由 M 级来做，因此宏观 PC 的值等于 M 级 PC 的值，也即是 `VPC` 的来源。（宏观 PC 具体在下面讲到）

CP0 记录什么值?

- 记录 `ExcCode` 时，如果当前有中断，不应该直接记录 `ExcCodeIn` ，而是记录 `Int` ；
- 当 `BDIn` 为 1 时，说明发生异常中断的指令是延迟槽内的指令，让 `EPC` 记录 `VPC - 4` ，即延迟槽前一条指令（某个分支跳转指令），否则记录 `VPC` ；
- `EXL` 置为 1；
- `BD` 置为 `BDIn` 。

退出异常中断处理程序

用来实现 `eret` 。
该指令需要产生一个信号，将 `EXLClr` 置 1，此时在 CP0 中， `EXL` 修改为 0 .这样便解除了 `EXL` 记录的异常中断处理状态。
在 D 级判断，需要置位 PC。

封装成单周期 CPU

宏观 PC

以某一个流水级为界限，作为输出端口输出出来，一般为 CP0 所在的流水级（M 级）。

精确异常

精确指出哪一条指令导致了异常，称为**异常受害指令**。

清空流水线

在异常发生时，需要清空流水线，避免宏观 PC 之后的指令被执行。清空宏观 PC 之后的指令所在的流水线寄存器，即插入 `nop` 。
信号优先级如下：

信号	优先级
reset	最高，复位大于一切
Req	次高，中断请求比内部阻塞重要
flush / Stall	最低，流水线信号，外部人员看不到

确定 CP0 的位置

E、M 级都可以，这里采用 M 级。

流水异常码

异常信号 ExcCode 应该流水到 CP0 所在流水级，而不能直接提交到 CP0 。

写入 EPC

发生异常的重要行为是将中断指令的 PC 写入 EPC ，如果是延迟槽指令就写入 PC - 4，如果不是就写入 PC。这里的 PC 是宏观 PC ，在上面处理异常部分也提到了。
这样，返回的时候将重新执行异常指令。

异常处理程序

异常处理程序由软件实现，该项目只需要提供接口。

新指令

针对异常处理以及其他需求，需要增加几条指令：`mfc0, mtc0, syscall, eret`

- `mfc0`：读 CP0 寄存器，`mfc0 rt, rd` ；GPR[rt] <- CP0[rd]； （COP0+rs）
- `mtc0`：写 CP0 寄存器，`mtc0 rt, rd` ；CP0[rd] <- GPR[rt]； （COP0+rs）
- `syscall`：系统调用，产生系统调用异常；（special+func）
- `eret`：异常返回，将保存在 CP0 的 EPC 寄存器中的现场（被中断的下一条地址）写入 PC，实现从异常中断处理程序返回。（COP0+func）

指令	op5	op4	op3	op2	op1	op0
“COP0”	0	1	0	0	0	0
"special"	0	0	0	0	0	0

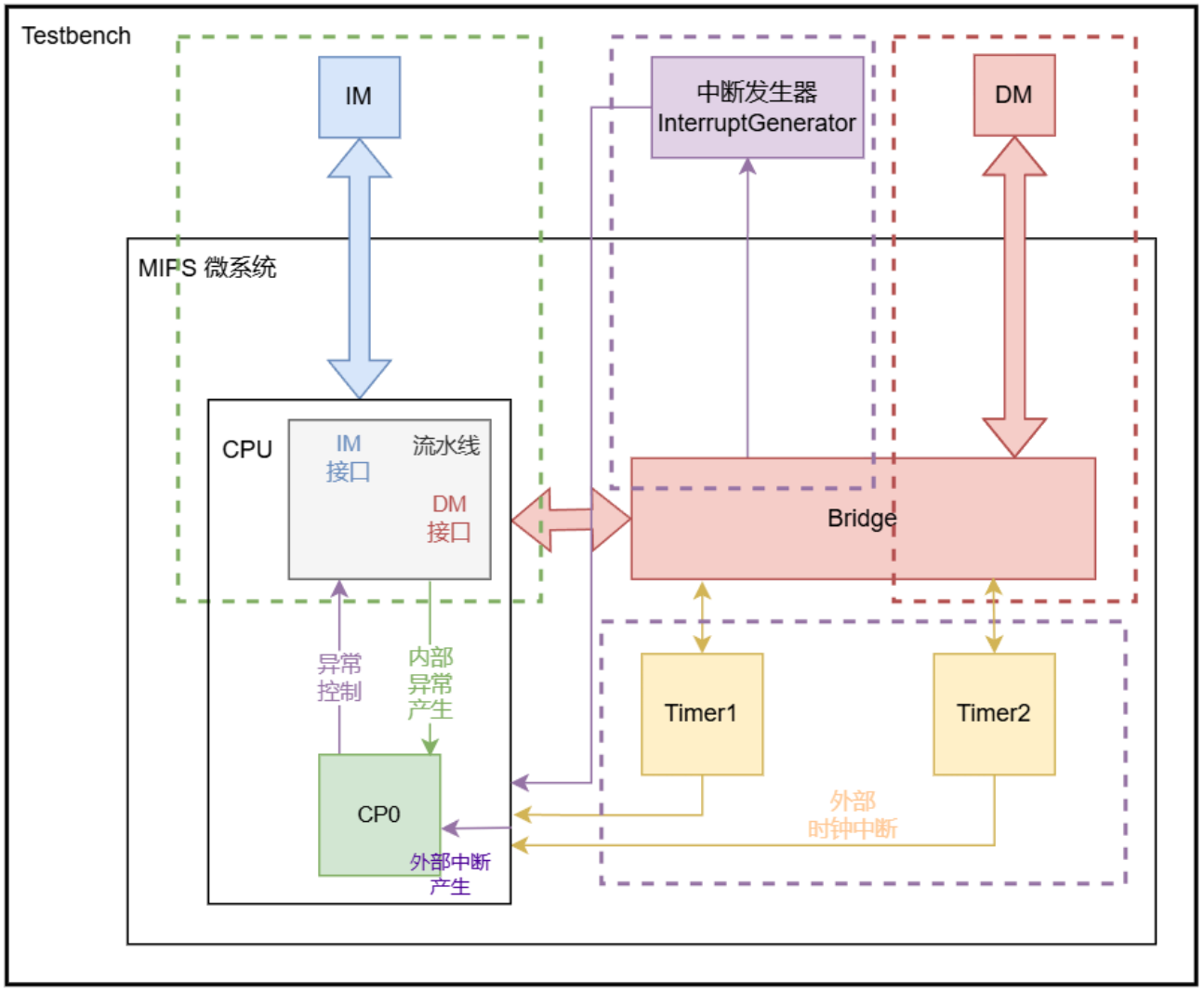
指令	rs4	rs3	rs2	rs1	rs0
mfc0	0	0	0	0	0
mtc0	0	0	1	0	0

指令	func5	func4	func3	func2	func1	func0
syscall	0	0	1	1	0	0

指令	func5	func4	func3	func2	func1	func0
eret	0	1	1	0	0	0

外设的实现

外设是独立于之前实现的 CPU （P6）的部分，最后的架构应该是这样：



绿色虚线是已经实现的部分，紫色虚线是新增的部分，红色虚线为改变后的 DM 接口。

计时器（Timer）

计时器的主要功能是**根据设定的时间来定时产生中断信号**，是系统中断的来源之一。
课程组给出了标准实现代码：

```
`timescale 1ns / 1ps
`define IDLE 2'b00
`define LOAD 2'b01
`define CNT  2'b10
`define INT  2'b11

`define ctrl   mem[0]
`define preset mem[1]
```

```
`define count  mem[2]
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    21:43:39 12/28/2017
// Design Name:
// Module Name:    TC
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module TC(
    input clk,
    input reset,
    input [31:2] Addr,
    input WE,
    input [31:0] Din,
    output [31:0] Dout,
    output IRQ
    );

        reg [1:0] state;
        reg [31:0] mem [2:0];

        reg _IRQ;
        assign IRQ = `ctrl[3] & _IRQ;

        assign Dout = mem[Addr[3:2]];

        wire [31:0] load = Addr[3:2] == 0 ? {28'h0, Din[3:0]} : Din;

        integer i;
        always @(posedge clk) begin
                if(reset) begin
                        state <= 0;
                        for(i = 0; i < 3; i = i+1) mem[i] <= 0;
                        _IRQ <= 0;
                end
                else if(WE) begin
                        // $display("%d@: *%h <= %h", $time, {Addr, 2'b00}, load);
                        mem[Addr[3:2]] <= load;
                end
                else begin
                        case(state)
```

```
                                `IDLE : if(`ctrl[0]) begin
                                        state <= `LOAD;
                                        _IRQ <= 1'b0;
                                end
                                `LOAD : begin
                                        `count <= `preset;
                                        state <= `CNT;
                                end
                                `CNT  :
                                        if(`ctrl[0]) begin
                                                if(`count > 1) `count <= `count-1;
                                                else begin
                                                        `count <= 0;
                                                        state <= `INT;
                                                        _IRQ <= 1'b1;
                                                end
                                        end
                                        else state <= `IDLE;
                                default : begin
                                        if(`ctrl[2:1] == 2'b00) `ctrl[0] <= 1'b0;
                                        else _IRQ <= 1'b0;
                                        state <= `IDLE;
                                end
                        endcase
                end
        end

endmodule
```

中断发生器（InterruptGenerator）

这是本课程抽象简化现实外设后得到的一种外设（键盘、鼠标等的抽象），会在不确定的时刻产生中断信号，并持续置高，直到微系统做出响应，变回低位。
对中断发生器的响应通过系统桥实现，通过 Store 类指令访问地址 `0x7F20` 。
中断发生器在测试 tb 上实现，无需自己实现，只需要通过以下功能：

- 微系统可以通过外部端口接受外部中断信号（在计时器部分已经实现了）。
- 微系统可以通过访问地址 `0x7F20` 的 `store` 类指令，改变对应的微系统输出信号（`m_int_addr`，`m_int_byteen`），即系统桥实现正确。

系统桥（Bridge）

系统桥传入对地址的访问请求，通过系统桥内部的转换代码，将请求转变为对应寄存器的读写操作。

思考题

> 1、请查阅相关资料，说明鼠标和键盘的输入信号是如何被 CPU 知晓的?

鼠标和键盘产生中断信号，进入中断处理区的对应位置，将输入信号从鼠标和键盘中读入寄存器。

> 2、请思考为什么我们的 CPU 处理中断异常必须是已经指定好的地址？如果你的 CPU 支持用户自定义入口地址，即处理中断异常的程序由用户提供，其还能提供我们所希望的功能吗？如果可以，请说明这样可能会出现什么问题？否则举例说明。（假设用户提供的中断处理程序合法）

因为中断异常处理程序是独立于其他正常部分的，因此需要单独一块地址空间，避免错误访问。应该是可以的，但要保证接入地址仍然是 0x4180 ，即接口一致，且异常处理方式合理。

> 3、为何与外设通信需要 Bridge？

系统桥是连接 CPU 和外设的功能设备，它会给 CPU 提供一种接口，使得 CPU 可以像读写普通存储器一样（即按地址读写）来读写复杂多变的外设。系统桥统一且简化了 CPU 的对外接口，CPU 不必为每种外设单独提供接口，符合高内聚，低耦合的设计思想。

> 4、请阅读官方提供的定时器源代码，阐述两种中断模式的异同，并分别针对每一种模式绘制状态移图。

略（）

> 5、倘若中断信号流入的时候，在检测宏观 PC 的一级如果是一条空泡（你的 CPU 该级所有信息均为空）指令，此时会发生什么问题？在此例基础上请思考：在 P7 中，清空流水线产生的空泡指令应该保留原指令的哪些信息？

> 6、为什么 `jalr` 指令为什么不能写成 `jalr $31, $31`？

测试数据（神中神！）

```
.text
    # 允许外部中断，Timer中断
    ori $t0, $0, 0x1c01
    mtc0 $t0, $12
    # 请在宏观PC为30c0和30d0的时候设置两次中断

    #lw取数地址未字对齐
    addi $t1 $0 0x3333
    sw $t1 0($0)
    lw $t1 3($0)

    #lh取数未半字对齐
    lh $t1 3($0)

    #lh、lb取Timer寄存器的值
    lh $t1 0x7F00($0)
    lb $t2 0x7F02($0)

    #load类指令计算地址加法溢出
    lui $t1 0xf000
    lw $t2 -0x7fff($t1)
    lh $t2 -0x7fff($t1)
    lb $t2 -0x7fff($t1)

    #load类指令取数地址越界
    ori $t1 $0 0x7F1C
```

```
lw $t2 0($t1)
lh $t2 0($t1)
lb $t2 0($t1)

#sw存数地址未字对齐
addi $t1 $0 0x4444
sw $t1 2($0)

#sh存数未半字对齐
sh $t1 3($0)

#sh、sb存Timer寄存器的值
sh $t1 0x7F00($0)
sb $t2 0x7F02($0)

#store类指令计算地址加法溢出
lui $t1 0xf000
sw $t2 -0x7fff($0)
sh $t2 -0x7fff($0)
sb $t2 -0x7fff($0)

#store类指令存数地址越界
ori $t1 $0 0x7F0C
sw $t2 0($t1)
sh $t2 0($t1)
sb $t2 0($t1)

#store类指令向计时器的 Count 寄存器存值
sw $t1 0x7F08($0)

#系统调用
ori $v0 $0 1
syscall

#未知指令
addu $t1 $t2 $t3

#溢出异常
lui $t1 0x7fff
lui $t2 0x7fff
add $t3 $t1 $t2
lui $t1 0xf000
addi $t2 $t1 -0x7000
lui $t1 0x7fff
lui $t2 0xf000
sub $t3 $t1 $t2

#正好处于E级的乘除法指令是否开始
lw $t3 1($0)
mult $t1 $t2

#已经开始的乘除法是否未受影响
mult $t1 $t2
```

```
    lw $t3 1($0)
    mflo $t4

    #正好处于E级的mthi与mtlo是否写入
    lw $t3 1($0)
    mthi $t4
    mfhi $t4

    #阻塞的时候发生中断(请在宏观PC为0x30c0的时候中断)
    lw $t1 0($0)
    jal kk1
    nop

kk1:
    #检测到异常时发生中断(请在宏观PC为0x30d0时发生中断)
    lw $t3 2($0)

    #阻塞过程中发生异常
    lw $t3 3($0)
    lw $t1 3($0)
    lw $t2 0($t1)

    #测试Timer的中断功能及其响应是否正确、中断优先级是否正确
    ori $t1 $0 9 #设置计时使能和中断屏蔽
    sw $t1 0x7F00($0)
    ori $t1 $0 1 #设置计时数
    sw $t1 0x7F04($0)
    nop
    nop
    lw $t1 2($0)
    nop
    nop
    nop
    nop

    #测试mtc0和eret指令的转发、以及清空延迟槽
    ori $t1 $0 0x3120
    mtc0 $t1 $14
    nop
    eret
    ori $t1 $0 0x444
    ori $t1 $0 0x3138
    sw $t1 0($0)
    lw $t2 0($0)
    mtc0 $t2 $14
    eret
    ori $t1 $0 0x444

    #延迟槽指令出错时是否正确存入EPC（如果处理不对，会在这两条指令死循环）
    beq $t1 $t1 jump1
    lw $t2 1($0)
jump1:
```

```
    #跳转到不对齐地址时，EPC是否正确（pc地址未字对齐、越界）
    ori $t1 $0 0x1111
    jr $t1
    nop
    ori $s0 $0 0x4444

    #未达到条件跳转时是否正确存入EPC
    beq $0 $t1 jump2
    lw $t2 1($0)
jump2:


end:
    beq $0, $0, end
    nop
_bad:
    ori $s2 $0 0x4444
    beq $0, $0, end
    nop

## 中间省略很多个nop
## 直到地址 0x4180 处

.ktext 0x4180
_entry:

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

    # 如果是中断，对$s1寄存器赋值0x222，并且响应中断
    beq $k0, $0, zhongduan
    nop
    #j _error
    beq $0 $0 _error
    nop

zhongduan:
    ori $s1 $0 0x222
    mfc0 $t1 $13
    andi $t1 $t1 0x0400
    beq $t1 $0 _waibuzhongduan  #处理外部中断
    nop
    #处理计时器1中断(同时测试对计时器读写是否正常)
    lw $t1 0x7F00($0)
    andi $t1 $t1 0xfff7
    sw $t1 0x7F00($0)
    #j _exception_return
    beq $0 $0 _exception_return
    nop
```

```
_waibuzhongduan:
    sb $0, 0x7f20($0)
    #j _exception_return
    beq $0 $0 _exception_return
    nop

_error:
    #判断是否是pc地址未对齐
    mfc0 $t1 $14
    ori $t2 $0 4
    div $t1 $t2
    mfhi $t1
    beq $t1 $0 _other_error #其他错误
    nop
    mfc0 $t1 $14
    ori $t2 $0 0x1111
    bne $t1 $t2 _bad #如果出错会写入0x4444
    nop
    ori $t1 $0 0x3150 #注意这个地方是特判，如果修改了之前的代码，会出错
    mtc0 $t1 $14
    beq $0 $0 _exception_return
    nop


_other_error:
    #判断是否为延迟槽指令
    mfc0 $t1 $13
    lui $t2 0x8000
    and $t1 $t1 $t2
    beq $0 $t1 _not_BD_error
    nop
    mfc0 $t1 $14
    ori $t2 $0 0x3138
    beq $t1 $t2 _right1
    nop
    ori $t2 $0 0x3150
    beq $t1 $t2 _right2
    nop
    beq $0 $0 _not_BD_error #如果BD的记录error,会陷入死循环
    nop
_right1:
    ori $t1 $0 0x3140
    mtc0 $t1 $14
    beq $0 $0 _exception_return
    nop
_right2:
    ori $t1 $0 0x3158
    mtc0 $t1 $14
    beq $0 $0 _exception_return
    nop

    # 将 EPC + 4，即处理异常的方法就是跳过当前指令,并且对$s1寄存器赋值0x111
_not_BD_error:
```

```
    ori $s1 $0 0x111
    mfc0 $k0, $14
    addi $k0, $k0, 4
    mtc0 $k0, $14
    beq $0 $0 _exception_return
    nop

_exception_return:
    eret
    # 测试eret后的指令是否会被执行
    ori $s0 $0 0x444
```

机器码：

```
34081c01
40886000
20093333
ac090000
8c090003
84090003
84097f00
800a7f02
3c09f000
8d2a8001
852a8001
812a8001
34097f1c
8d2a0000
852a0000
812a0000
20094444
ac090002
a4090003
a4097f00
a00a7f02
3c09f000
ac0a8001
a40a8001
a00a8001
34097f0c
ad2a0000
a52a0000
a12a0000
ac097f08
34020001
0000000c
014b4821
```

```
3c097fff
3c0a7fff
012a5820
3c09f000
212a9000
3c097fff
3c0af000
012a5822
8c0b0001
012a0018
012a0018
8c0b0001
00006012
8c0b0001
01800011
00006010
8c090000
0c000c34
00000000
8c0b0002
8c0b0003
8c090003
8d2a0000
34090009
ac097f00
34090001
ac097f04
00000000
00000000
8c090002
00000000
00000000
00000000
00000000
34093120
40897000
00000000
42000018
34090444
34093138
ac090000
8c0a0000
408a7000
42000018
34090444
11290001
```

```
8c0a0001
34091111
01200008
00000000
34104444
10090001
8c0a0001
1000ffff
00000000
34124444
1000fffc

// 很多个 00000000

401a6800
341b007c
035bd024
13400003
00000000
1000000e
00000000
34110222
40096800
31290400
11200006
00000000
8c097f00
3129fff7
ac097f00
1000002e
00000000
a0007f20
1000002b
00000000
40097000
340a0004
012a001a
00004810
11200009
00000000
40097000
340a1111
152afbdb
00000000
34093150
40897000
```

```
1000001d
00000000
40096800
3c0a8000
012a4824
10090012
00000000
40097000
340a3138
112a0006
00000000
340a3150
112a0007
00000000
10000009
00000000
34093140
40897000
1000000b
00000000
34093158
40897000
10000007
00000000
34110111
401a7000
235a0004
409a7000
10000001
00000000
42000018
34100444
```