

# [BUAA-CO-Lab] P7 MIPS 微体系

*Posted by roife on Sat, Jan 2, 2021*

## 上机总结

课上总共四道题：

1. 弱测：测 P6 的功能
2. 基础测试（中强测）：测 `mtc0`、`mfc0`、`eret` 和中断异常
3. 强测
4. 新增异常 `syscall`

如果课下没问题，前三个可以直接交。第四个只要在 F 级识别特定指令流水异常就好了（和 RI 差不多）。

```
assign F_excCode = F_excAdEL ? `EXC_AdEL :  
                  F_syscall ? `EXC_syscall :  
                  `EXC_None;
```

## 课下总结

### 前言

P7 是个坑，因为教程写得很垃圾，而且 P7 的实现非常灵活，所以一千个人能写出一千零一种 P7。

P7 还不能和 Mars 对拍，只能和小伙伴对拍。并且如果实现不同，那么和小伙伴的对拍也会很艰难。总之我很好奇助教们是怎么测试 P7 的。

一个建议：看完课下教程和 PPT 还是模模糊糊的怎么办？建议分块写。先写 CP0，然后写中断异常，最后处理桥和计时器。如果还不懂就对着本文和学长的代码慢慢琢磨。

## 要做什么

P7 总共要实现 3 个东西：中断异常，协处理器 CP0，桥。其中桥是用来接受中断的，CP0 是来处理中断的。下面分三部分讲解。

因为我们的 CPU 运行可能会出错，那么就需要处理这些错误，这就是 CP0 干的活。当出错时 CP0 会引导 CPU 跳转到特定的程序（Exception handler）处理错误，处理完再跳回去继续执行。有人可能会好奇要如何处理，这个不是我们要考虑的，因为怎么处理是软件的事，也就是在你的测试数据里面考虑的，我们要干的就是出错了跳过去（比如 handler 可以修改 EPC 跳过异常指令，具体怎么做我们不管）。

建议做之前先看高小鹏的 PPT，然后看这个。

首先我们要定好大致的框架：CP0 放在 M 级，中断也从 M 级传入。之所以这么做是因为这么做的人比较多，方便对拍。

## 基础概念

- **宏观 PC**：虽然我们的 CPU 是流水线的，但是我们要将其封装模拟成一个单周期 CPU。而宏观 PC 就是这个单周期 CPU 的 PC 地址。我们要保证宏观 PC 前面的指令都已经完成了，后面的指令都没有执行过，当前的指令正在执行中。我们将 CP0 放在 M 级，因此也以 M 级为界线，规定 M 级的 PC 就是宏观 PC。
- **受害 PC**：即哪条指令出了问题，我们要将这条出问题的指令存入 EPC。对于普通的异常，我们约定受害指令就是发生错误的那条指令。但是对于跳转指令，我们认为受害指令为跳转的目标指令。发生跳转问题时，我们流水线中的指令顺序为 D 错误指令，E 延迟槽，M 跳转指令，此时根据上面的描述，受害 PC 是 D 级的 PC。

首先声明，在 P7，为了简单我们约定以下几件事：

1. 如果发生异常的指令是**延迟槽指令**，那么我们要求返回程序时仍然返回这条指令所属的跳转指令。也就是说“异常延迟槽回到跳转”。
2. 如果发生异常的指令是**跳转指令**，那么我们要求执行完延迟槽。详细解释见上述的“宏观 PC”。

3. 如果发生异常的指令是**乘除指令的下一条**，我们允许乘除指令不被撤回。也就是对于 M错误指令，W乘除指令的情况，此时乘除槽正在计算，本来在异常处理时可能会覆盖乘除槽的结果，但是我们约定不会这么做。但是注意，如果是 E乘除指令，M错误指令，你要保证乘除指令不执行。

## CP0

CP0 要干的事就是接收到中断异常时看看是否允许其发生，允许的话记录一下状态方便 handler 处理。

我们要实现 CP0 中的四个寄存器：SR、Cause、EPC、PrID。

- SR 表示系统的状态，比如能不能发生异常
- Cause 记录异常的信息，比如是否处于延迟槽以及异常的原因
- EPC 记录发生异常的位置，便于处理完中断异常的时候返回
- PrID 是一个可以随便定义的寄存器，表示你的 CPU 型号

其中 SR 我们也只要实现一部分：SR[15:10] 表示允许发生的中断；SR[1] 表示是否处于中断异常中（是的话就不能发生中断异常）；SR[0] 表示是否允许中断。Cause 也只要实现一部分：Cause[31] 表示延迟槽标记；Cause[15:10] 表示发生了哪个中断；Cause[6:2] 表示异常原因。为了方便我们定义一些宏。

```
`define IM SR[15:10]
`define EXL SR[1]
`define IE SR[0]
`define BD Cause[31]
`define hwint_pend Cause[15:10]
`define ExcCode Cause[6:2]
```

首先讲一下异常和中断的条件（按照 PPT 说的来）：

```
wire IntReq = ((HWInt & `IM)) & !`EXL & `IE; // 允许当前中断 且 不在中断异常中 且 允许中断
wire ExcReq = (!ExcCodeIn) & !`EXL; // 存在异常 且 不在中断中
assign Req = IntReq | ExcReq;
```

发生异常的处理方法：

```

if (Req) begin // int|exc
    `ExcCode <= IntReq ? 5'b0 : ExcCodeIn;
    `EXL <= 1'b1;
    EPCreg <= tempEPC;
    `BD <= bdIn;
end

```

这里讲一下 BD 是干啥的。如果异常发生在延迟槽，那么按照要求我们返回的时候要返回跳转指令。所以如果 BD 信号为真时应该输出上一条指令的 PC。

```

wire [31:2] tempEPC = (Req) ? (bdIn ? PC[31:2]-1 : PC[31:2])
                        : EPCreg;

assign EPCout = {tempEPC, 2'b0};

```

然后就是记得每个时钟上升沿都要更新 HWInt：

```

`hwint_pend <= HWInt;

```

退出异常的条件是识别到了 `eret`，我们直接把 `EXLClr` 接上 `M_eret` 就好。

## 中断异常

我们要实现的异常有 5 种：

1. 中断 Int（看成特殊的异常，异常码为 0）
2. 读取异常 AdEL（读取错误的指令，DM 读取错误，读取外设错误）
3. 写入异常 AdES（DM 写入错误，写入外设错误）
4. 非法指令 RI（识别不出来的指令）
5. 算术溢出 Ov

发生异常时要清空流水线，也就是传入流水线寄存器一个 `req` 信号，如果发生了异常中断则类似于 `reset`。

## AdEL / AdES

读取错误的指令可以在 F 级判断后流水

```
assign excAdEL = ((|pc[1:0]) | (pc < `StartInstr) | (pc > `EndInstr)) && !D_eret;
```

DM 存取错误其实差不多，可以放一起写。就是要注意如果 lw 发生了溢出，那么算作读取错误而不是 ov：

```
wire ErrAlign = ((DMType == `DM_w) && (|addr[1:0])) ||  
                (((DMType == `DM_h) || (DMType == `DM_hu)) && (addr[0]));  
wire ErrOutOfRange = !( ((addr >= `StartAddrDM) && (addr <= `EndAddrDM)) ||  
                        ((addr >= `StartAddrTC1) && (addr <= `EndAddrTC1)) ||  
                        ((addr >= `StartAddrTC2) && (addr <= `EndAddrTC2)));  
wire ErrTimer = (DMType != `DM_w) && (addr >= `StartAddrTC1);  
wire ErrSaveTimer = (addr >= 32'h0000_7f08 && addr <= 32'h0000_7f0b) ||  
                   (addr >= 32'h0000_7f18 && addr <= 32'h0000_7f1b); // cannot access c  
assign excAdEL = load && (ErrAlign || ErrOutOfRange || ErrTimer || excOvDM);  
assign excAdES = store && (ErrAlign || ErrOutOfRange || ErrTimer || ErrSaveTimer || excO  
// excOvDM 信号是 E 级流水过来的
```

## RI

枚举所有的指令。注意考虑 nop。

```
assign excRI = !( beq | bne | bgez | bgtz | blez | bltz |  
                  j | jal | jalr | jr |  
                  lb | lbu | lh | lhu | lw | sb | sh | sw |  
                  lui | addi | addiu | andi | ori | xori | slti | sltiu |  
                  add | addu | sub | subu | And | Nor | Or | Xor | ori | slt | sltu |  
                  sll | sllv | sra | srav | srl | srlv |  
                  ((opcode == 6'b000000) && (func==6'b000000)) |  
                  div | divu | mfhi | mflo | mthi | mtlo | mult | multu |  
                  mtc0 | mfc0 | eret);
```

## Ov

注意两种指令的溢出要分开来。

```

assign ALUAriOverflow = add | addi | sub;
assign ALUDMOverflow = lb | lbu | lh | lhu | lw | sb | sh | sw;
wire [32:0] exA = {A[31], A}, exB = {B[31], B};
wire [32:0] exAdd = exA + exB, exSub = exA - exB;
assign excOvAri = ALUAriOverflow && (
    ((ALUControl == `ALU_add) && (exAdd[32] != exAdd[31])) ||
    ((ALUControl == `ALU_sub) && (exSub[32] != exSub[31]))
);
assign excOvDM = ALUDMOverflow && (
    ((ALUControl == `ALU_add) && (exAdd[32] != exAdd[31])) ||
    ((ALUControl == `ALU_sub) && (exSub[32] != exSub[31]))
);

```

## 一个关于空泡的问题

我们会遇到一个问题：在阻塞时我们会往流水线中插入 nop，这个 nop 的 pc 和 bd 信号都是 0。此时宏观 PC 会显示错误的值。并且如果此时发生了中断，就会导致 EPC 存入错误的值。

按照道理来讲，如果插入了 nop，它的 pc 和 bd 应该是目前这条指令的值。（想一想，为什么？）比如指令序列是 D-add, E-nop, M-lw，那么 nop 的 pc 是 add。这样当 nop 到 M 级时的宏观 PC 才是正确的。

因此我们在流水时最好把这两个信号一起流水。

```

// D_reg
always @(posedge clk) begin
    if (reset | req) begin
        instr_out    <= 0;
        pc_out       <= req ? 32'h0000_4180 : 0;
        excCode_out  <= 0;
        bd_out       <= 0;
    end else if(WE) begin
        instr_out    <= instr_in;
        pc_out       <= pc_in;
        excCode_out  <= excCode_in;
        bd_out       <= bd_in;
    end
end

// E_reg
always @(posedge clk) begin
    if (reset || stall || req) begin
        instr_out    <= 0;
        pc_out       <= stall ? pc_in : (req ? 32'h0000_4180 : 0);
        excCode_out  <= 0;
        bd_out       <= stall ? bd_in : 0;
        // ...
    end else if(WE) begin
        instr_out    <= instr_in;
        pc_out       <= pc_in;
        excCode_out  <= excCode_in;
        bd_out       <= bd_in;
        // ...
    end
end

// M_reg
always @(posedge clk) begin
    if (reset || req) begin
        instr_out    <= 0;
        pc_out       <= req ? 32'h0000_4180 : 0;
        bd_out       <= 0;
        // ...
    end else if(WE) begin
        instr_out    <= instr_in;
        pc_out       <= pc_in;
        bd_out       <= bd_in;
        // ...
    end
end

// W_reg 不用管，因为宏观 PC 在 M 级

```

这里有一句话需要注意：  $pc\_out < req ? 32'h0000\_4180 : 0;=$ 。在发生异常时，我们需要立即跳到异常响应代码，并且清空流水线内还没有执行完的指令。但是如果直接将 `pc_out` 置为 0，会导致在异常代码到达 M 级之前的宏观 PC 都变成了 0（因为这几条指令都被清空了）。所以发生异常时我们在这里将 `pc_out` 置为异常代码地址 `32'h0000_4180`，避免宏观 PC 出现突然变成 0 的情况。


## mfc0/mtc0/eret

这三个指令用来处理异常。

记得修改阻塞单元。个人没写 EPC 转发，感觉没必要，只写了 `eret` 阻塞。

`eret` 的阻塞条件为：

```
wire stall_eret = D_eret & ((E_mtc0 & (E_rd_addr == 5'd14)) || (M_mtc0 & (M_rd_addr == 5
```



## 乘除槽

注意如果发生了异常，乘除法就不能执行，即：

```
always @(posedge clk) begin
    if (reset) begin
        // ...
    end else if (!req) begin // req 表示中断或者异常
        // ...
    end
end
```

## 异常识别

注意异常的优先顺序：

- 同一个指令如果发生多个异常，则优先考虑最先发生的异常（例如同时在 D、E 级发生异常，就先考虑 D 级发生的异常）
- 如果多条指令都会发生异常，那么也是优先考虑最先发生的异常（比如 D、E、M 级的指令都发生异常，则先考虑 M 级指令导致的异常）



表现在流水线上就是：

- 对于同一条指令的多个异常，越靠近 F 级的**异常**优先级越高（因为先发生）
- 对于多条指令的异常，越靠近 M 级的**指令发生的异常**优先级越高（也是因为先发生）

```
assign F_excCode = F_excAdEL ? `EXC_AdEL : `EXC_None;

assign D_excCode = D_old_excCode ? D_old_excCode : // D_old_excCode 是 F 级发生的异常，后
D_excRI ? `EXC_RI :
`EXC_None;

assign E_excCode = E_old_excCode ? E_old_excCode :
E_excOvAri ? `EXC_Ov :
`EXC_None;

assign M_excCode = M_old_excCode ? M_old_excCode :
M_excAdEL ? `EXC_AdEL :
M_excAdES ? `EXC_AdES :
`EXC_None;
```

## eret

发生 eret 的时候我们有多种做法：

1. 当 M 级识别到 eret 时清空 FDEM 级
2. 当 D 级识别到时插入一个 nop（还记得 P5 的清空延迟槽吗）
3. 直接强制就如 EPC

我用第三种做法，快一点

```
assign pc = D_eret ? EPC : tempPC;
assign instr = (excAdEL) ? 0 : im[pc[14:2] - 12'hc00];

// NPC
assign npc = req ? 32'h0000_4180 :
eret ? EPC + 4 : // 注意此时 F 是 EPC，我们的 NPC 要变成 EPC + 4
// ...
F_pc + 4;
```

## 桥和 DM

我们的 CPU 把 DM 中一块特殊的区域用来作为与外设交互的接口，中间通过桥来连接。这个没啥好说的，看懂 PPT 就行，直接放代码吧。

```
wire selTC1 = (`RAddr >= `StartAddrTC1) && (`RAddr <= `EndAddrTC1),
    selTC2 = (`RAddr >= `StartAddrTC2) && (`RAddr <= `EndAddrTC2);
wire TCwe1 = selTC1 && PrWE,
    TCwe2 = selTC2 && PrWE;
wire [31:0] TCout1, TCout2;
wire IRQ1, IRQ2;
assign PrRD =    selTC1 ? TCout1 :
                  selTC2 ? TCout2 :
                  0;
wire [5:0] HWInt = {3'b0, interrupt, IRQ2, IRQ1};
```

## DM

注意 DM 写入的条件（WE 接口）为 M\_WE & (!req)。

内部 always@(posedge clk) 中的也要改（考虑到外设）。

```
if (WE && (addr >= `StartAddrDM) && (addr <= `EndAddrDM)) begin
    // ...
end
```

注意下一级寄存器（W\_reg）传入 DM 数据时要判断是否是外设的数据。

```
W_REG W_reg(
    // ...
    .DM_in((M_ALUout >= 32'h0000_7f00) ? PrRD : M_DMout),
    // ...
);
```

## 计时器

这个东西给了源码和文档，很快就看懂了。

### PREVIOUS

[BUAA-CO-LAB] P6 流水线 CPU-LITE2  
(**/POSTS/BUAA-CO-LAB-P6/**)

### NEXT

使用 UTTERANCES 为静态博客添加评论  
(**/POSTS/USE-UTTERANCES-FOR-BLOG-COMMENT/**)



(<https://github.com/roife>)



(<https://twitter.com/roifex>)



(<mailto:roifewu@gmail.com>)



(</index.xml>)

Copyright © ROIFE BLOG 2023 | Powered by Hugo (<https://gohugo.io>) | Made with Emacs  
(<https://www.gnu.org/software/emacs/>)