

实验 4-1 报告

学号：2016K8009929029

姓名：张丽玮

一、实验任务（10%）

这个实验主要目的在于使得 CPU 支持例外中断。第一部分主要实现三个指令：MTC0、MFC0、ERET。同时要对 cp0 寄存器的作用有所了解，并且实现 CP0 寄存器 STATUS、CAUSE、EPC。除此之外还有 SYSCALL 指令，也就是要完成对系统调用的支持。

二、实验设计（30%）

总的来说，是添加 mtc0、mfc0、eret 和 syscall 这几个指令，为了实现这些指令，需要增加 cp0 寄存器。这里 epc 用于返回系统调用发生的地址，从而在 eret 的时候回到 epc 寄存器的地址；cause 是用于判断是哪一种例外（通过 exccode 域），而 status 寄存器的 IM7-IM0 位可以控制中断的屏蔽打开。

域名称	位	功能描述	读/写	复位值
0	31..23	只读恒为 0。	0	0
Bev	22	恒设为 1	R	1
0	21..16	只读恒为 0。	0	0
IM7..IM0	15..8	中断屏蔽位。每一位分别控制一个外部中断、内部中断或软件中断的使能。 1：使能；0：屏蔽。	R/W	无
0	7..2	只读恒为 0。	0	0
EXL	1	例外级。当发生例外时该位被置 1。0：正常级；1：例外级。 当 EXL 位置为 1 时： ◆ 处理器自动处于核心态 ◆ 所有硬件与软件中断被屏蔽 ◆ EPC、Cause _{BD} 在发生新的例外时不做更新。	R/W	0x0
IE	0	全局中断使能位。 0：屏蔽所有硬件和软件中断； 1：使能所有硬件和软件中断。	R/W	0x0

域名称	位	功能描述	读/写	复位值
BD	31	标识最近发生例外的指令是否处于分支延迟槽。1：在延迟槽中；0：不在延迟槽中	R	0x0
TI	30	计时器中断指示。1：有待处理的计时器中断；0：没有计时器中断。	R	0x0
0	30..16	只读恒为 0。	0	0
IP7..IP2	15..10	待处理硬件中断标识。每一位对应一个中断线，IP7~IP2 依次对应硬件中断 5~0。 1：该中断线上有待处理的中断；0：该中断线上无中断。	R	0x0
IP1..IP0	9..8	待处理软件中断标识。每一位对应一个软件中断，IP1~IP0 依次对应软件中断 1~0。 软件中断标识位可由软件设置和清除。	R/W	0x0
0	7	只读恒为 0。	0	0
ExcCode	6..2	例外编码。详细描述请见表 6-6。		
0	1..0	只读恒为 0。	0	0

1、fetch 模块/next_pc

fetch 模块并没有增添信号，因为目前的例外并不会在取指阶段发生，也并不会受到影响。

但是计算 nextpc 的逻辑需要改变，如下：

```
assign Jump_addr = JSRC ? JR_target_DE : J_target_DE;
assign inst_addr = realtrap ? except_addr :
                  is_eret ? epc : PC_mux; //new 判断如果是例外中断就到例外处理入口地址，如果是eret就到epc否则正常pc
assign inst_scan_addr = PCWrite ? inst_addr : nextpc;
```

如果是 trap 例外发生，那么就跳转到例外发生地址

```
except_addr = 32'hbfc00380
```

如果是 eret 指令（这里 is_eret 是一个信号），那么需要回到 epc 地址。

这样通过 nextpc 基本完成了一个例外中断的地址跳转。

2、decode 模块

decode 模块是一个译码操作模块，增加的接口较多：

```
output wire      eret_DE,
output wire      trap_DE,
output wire      cp0_Write,
output wire [ 4:0] rd,

input  wire [31:0] cp0Rdata_DE, //new
```

这里实际上进行了一个 cp0 寄存器内容的输入（cp0 寄存器内容在 cp0reg 模块有处理），然后通过 control 处理输出一些例外中断信号（判断是否是 eret, 是否是 trap）

因此这一部分主要在 control 模块里详细叙述。

3、execute 模块

这是一个执行模块，增加了如下几个接口：

```
output wire [31:0] Bypass_EX, // Bypass

input  wire [31:0] cp0Rdata_DE_EX,
input  wire      mfc0_DE_EX,

output reg [31:0] cp0Rdata_EX_MEM,
output reg      mfc0_EX_MEM

input  wire      is_signed_DE_EX,
```

增添的逻辑语句仅仅是这一句：

```
assign Bypass_EX = mfc0_DE_EX ? cp0Rdata_DE_EX : ALUResult_EX;
```

判断是否是 mfc0 指令，如果是，就取出 cp0Rdata 的值，否则正常输出 ALU 的结果。

4、memory 模块

简单明了的存指操作，接口命名可以看出数据来源和去向。主体部分就是一个时钟节拍控制的赋值操作。

主要增加的接口和 EX 模块类似：

```
output wire [31:0] Bypass_MEM, //Bypass

input wire [31:0] cp0Rdata_EX_MEM,
input wire mfc0_EX_MEM,
output reg [31:0] cp0Rdata_MEM_WB,
output reg mfc0_MEM_WB
```

而实际山增添的操作也就是对于是否是 mfc0 指令的判断。

```
assign Bypass_MEM = mfc0_EX_MEM ? cp0Rdata_EX_MEM : ALUResult_EX_MEM;
```

5、writeback 模块

一个写回模块。不需要时钟信号控制，直接赋值即可。

写回模块类似上面的操作，这里主要影响了写回的 regwdata 寄存器，同样是判断是否是 mfc0 寄存器，是就取 cp0Rdata 的值。

```
assign RegWdata_WB = |MFHL_MEM_WB ? HI_LO_out : (MemToReg_WB ? MemRdata_Final : (mfc0_MEM_WB ? cp0Rdata_MEM_WB : ALUResult_MEM_WB));
assign RegWdata_Bypass_WB = |MFHL_MEM_WB ? HI_LO_out : (mfc0_MEM_WB ? cp0Rdata_MEM_WB : ALUResult_MEM_WB);
```

6、control 模块

decode 的主要功能模块。这次主要增加了这几个接口输出：

```
output wire trap,
output wire eret,
output wire cp0_Write,
output wire mfc0,
output wire is_signed,
output wire in_inst_set
```

同样也增加了这几条指令的判断信号：

```
// Exception and Interrupt related instructions
wire inst_mtc0    = (op == 6'b010000) && (rs == 5'b00100);
wire inst_mfc0    = (op == 6'b010000) && (rs == 5'b00000);
wire inst_syscall = (op == 6'b000000) && (func == 6'b001100);
wire inst_eret    = (op == 6'b010000) && (func == 6'b011000);
wire inst_break   = (op == 6'b000000) && (func == 6'b001101);
```

除了 mfc0 指令在 regwrite 信号里增添，其他都在下方额外增加语句判断信号。

```
//new
assign mfc0 = ~rst & inst_mfc0;
assign eret = ~rst & inst_eret;
assign trap = ~rst & (inst_syscall | inst_break);

assign cp0_Write = ~rst & (inst_mtc0 | inst_syscall | inst_break);

assign is_signed = ~rst & (inst_add | inst_sub | inst_addi);

assign in_inst_set = rst
| inst_lw | inst_beq | inst_bne | |
| inst_sw | inst_addiu | inst_jr | inst_sll |
| inst_sltiu | inst_lui | inst_addu | inst_addi |
| inst_or | inst_slt | inst_xori | inst_add |
| inst_andi | inst_ori | inst_sltu | inst_and |
| inst_sub | inst_subu | inst_sllv | inst_sra |
| inst_nor | inst_xor | inst_srlv | inst_div |
| inst_srav | inst_srl | inst_srlv | inst_div |
| inst_divu | inst_mult | inst_multu | inst_mfhi |
| inst_mflo | inst_mthi | inst_mtlo | inst_jalr |
| inst_bgtz | inst_blez | inst_bltz | inst_bgez |
| inst_bltzal | inst_bgezal | inst_lb | inst_lbu |
| inst_lh | inst_lhu | inst_lwl | inst_lwr |
| inst_sb | inst_sh | inst_swl | inst_swr |
| inst_mtc0 | inst_mfc0 | inst_syscall | inst_eret |
| inst_j | inst_jal | inst_slti | inst_break ;
```

trap 用来判断是否是 syscall 或者 break 指令，这两个都表示要返回 epc 地址。Is_signed 和 in_inst_set 在

4-1 目前没有用到，适用于判断 4-2 的 ALU 运算的整形溢出例外。

7、Bypass 模块

这是一个为了实现指令相关时延迟取指的旁路模块，

这一模块只增加了对 trap 的逻辑操作：


```

assign IRWrite = ~(DE_EX_Stall | realtrap)

assign realtrap = trap & ~trap_flag;

always @ (posedge clk)
if (rst)
    trap_flag <= 1'b0;
else if (trap | trap_flag)
    trap_flag <= ~trap_flag;
else
    trap_flag <= trap_flag;

```

这是一个对陷入的追加判断。一旦陷入，陷入处理程序被规定在各自的进程上下文中执行。

8、divider 模块

一个运用迭代的除法器。busy 和 done 信号分别表示除法是否正在运行和是否已经做完，从而在除法进行时对于其他操作阻塞处理。

实际上是一个辗转相除法的处理，利用时钟周期，和 count，完成一个 busy 和 done 的判断从而达到阻塞效果。

9、Multiplier 模块

此模块原本实现了 booth 算法的乘法器，但是由于是和 div 除法器一样利用了时钟周期，进行了一个状态机的操作，而导致在测试 mul_tb 的时候由于时钟延迟，没有办法获得正确结果。后来只是简单采用了乘号。（具体代码可见 multiply.v 文件）。

等之后时间富裕之后再添加信号 busy 和 done 等尝试完成这个乘法器。

10、Cp0reg 模块

Cp0 寄存器的处理模块，也可以说是这次例外中断的核心模块之一。

```

module cp0reg(
    input clk,
    input rst,
    input wen,
    input eret,
    input trap,
    input [5:0] int,
    input [`ADDR_WIDTH - 1:0] waddr,
    input [`ADDR_WIDTH - 1:0] raddr,
    input [`DATA_WIDTH - 1:0] wdata,
    output [`DATA_WIDTH - 1:0] rdata,
    output [`DATA_WIDTH - 1:0] epc_value
);

```

输入以下信号，用以判断是否是 `eret` 或者陷入指令，并且会返回 `epc` 的值。

具体操作的时候，将 `status` 和 `cause` 寄存器每部分单独罗列出来判断再整合（列举 `status` 的操作）

```
if (rst)
begin
    status_IM7    <= 1'b0;
    status_IM6    <= 1'b0;
    status_IM5    <= 1'b0;
    status_IM4    <= 1'b0;
    status_IM3    <= 1'b0;
    status_IM2    <= 1'b0;
    status_IM1    <= 1'b0;
    status_IM0    <= 1'b0;
    status_EXL    <= 1'b0;
    status_IE     <= 1'b0;
end
else begin
    if (eret)
        status_EXL <= 1'b0;
    else if (trap)
        status_EXL <= 1'b1;
    else
        status_EXL <= status_EXL;

    if (wen && waddr==5'd12)
    begin
        status_IM7 <= wdata[ 15];
        status_IM6 <= wdata[ 14];
        status_IM5 <= wdata[ 13];
        status_IM4 <= wdata[ 12];
        status_IM3 <= wdata[ 11];
        status_IM2 <= wdata[ 10];
        status_IM1 <= wdata[  9];
        status_IM0 <= wdata[  8];
        status_IE  <= wdata[  0];
    end
end
end
```

除了对于 `status` 和 `cause` 寄存器的操作，还有 `epc` 寄存器

```
if (rst)
    epc <= 32'd0;
else if (!status_EXL)
    epc <= epc;
else if (wen && waddr==5'd14)
    epc <= wdata[31:0];
```

一旦 `EXL` 位为 1 那么在发生新的例外的时候 `epc` 的值并不进行更新。

最后通过 rdata 将值输出出去。

三、实验过程（60%）

（一）实验流水账

11月14日，那两天在写12306并没有动cpu

11月15日，回顾一下ppt，看了一下MIPS指令手册以及cp0寄存器的详细情况

11月16日，下午5点到晚上12点，开始动工cpu

11月17日，下午4点到晚上9点，中断例外基本框架写完，开始跑测试，失败

11月18日，下午4点到晚上11点，debug，通过69条测试

11月19日，下午5点到晚上10点，企图增加后续指令，结果69测试点都跑不过了……回退回去

（二）错误记录

具体描述实验过程中的错误，环境问题、仿真阶段、上板阶段的都可以记录。

1、错误1

（1）错误现象

```
[1704717 ns] Error!!!  
reference: PC = 0xbfc0ed4c, wb_rf_wnum = 0x15, wb_rf_wdata = 0x00000002  
mycpu      : PC = 0xbfc0ed4c, wb_rf_wnum = 0x15, wb_rf_wdata = 0xxxxxxxxx
```



第35个测试点开始报错

（2）分析定位过程

有Z调Z，HI和LO的连接在更改过程中出了问题

（3）错误原因

删除注释冗余内容时不注意删掉了之前代码的寄存器定义

（4）修正效果

增添回来，完成。

（5）归纳总结（可选）

代码删改需备份，需谨慎。

2、错误2

（1）错误现象

```
[3312457 ns] Error!!!
```

```
reference: PC = 0xbfc00400, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00000020
```

```
mycpu      : PC = 0xbfc00400, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x0000xX20
```

前 68 测试点全过之后第 69 个测试点过程中的问题

(2) 分析定位过程

寻找中间部分 xx 的原因

(3) 错误原因

这是在我加了 break 信号和其他例外之后出现的问题，初步判断为新加入的信号扰乱了之前的逻辑。

(4) 修正效果

将后增添的部分先行删除，回退到原先版本。

(5) 归纳总结（可选）

其实应该更仔细分析，这个 xx 一定不是因为这条指令出错，而是之前某条指令开始写 ram 的时候出现了问题，导致这次取 data 出现 xx。因为这周比较爆炸所以没来得及处理这一问题，留待下次解决。

四、实验总结（可选）

这一次的实验报告写的真的非常简陋了……因为要准备人工智能期中考试和数据库系统 12306 的实验验收（orz），实验做的比较仓促，里面可能还有些东西自己也还没完全弄明白，可能需要拖到 4-2 的时候再自己搞清楚。因为实际上本来有直接把 4-2 也写了的野心，但是写了之后发现甚至连 4-1 的测试都跑不过了，说明理解方面可能还是有些问题。

按部就班来，以及代码注意备份。说实话当时写了后面内容之后改回原本能过的版本花了我很久很久……要是改不回来就凉了啊 orz。

因为用的 control 模块总体控制，每级一个模块，很多信号，越来越多的信号要逐层传递，无论是写的时候还是 debug 的时候都非常麻烦，可能是时候考虑一下老师给的代码风格了。