浙江大学

本科实验报告

课程名称：	计算机体系结构
姓　　名：	姜雨童
学　　院：	计算机科学与技术学院
专　　业：	计算机科学与技术
邮　　箱：	3220103450@zju.edu.cn
QQ　　号：	1369218489
电　　话：	18867766468
指导教师：	王小航
报告日期：	2024 年 10 月 10 日

浙江大学实验报告

课程名称：计算机体系结构 实验类型：综合

实验项目名称：Lab02 Pipelined CPU supporting exception & interrupt

学生姓名：姜雨童 学号：33220103450 同组学生姓名：/

实验地点：玉泉曹西 301 实验日期：2024 年 10 月 10 日

一、目标与原理

1-1 实验目标

Understand RISC-V Privilege Levels, CSR and CSR instructions, trap handling process.
Master the design methods of pipelined CPU supporting exception & interrupt.

1-2 实验原理

1-2-1 相关知识

RISC-V 定义了四种特权模式：Machine（M）、Hypervisor（H）、Supervisor（S）、User（U），其中机器模式（M）是最高的特权模式，拥有对底层模式的一切访问权限。
Trap 是造成特权模式切换的一个重要原因。用户（U）模式下发生某些 trap 时，线程会被强制切换到 trap handler 并执行相关指令。但是在跳转之前，需要先保存当前线程的各种信息，这就用到了 CSR。
CSR（Control Status Register）保存了当前线程的各种状态信息，可以用来恢复状态。本实验只涉及到机器模式，因此只用到了几个 M 开头的 CSR（例如 mstatus，mtevc 等）。

1-2-2 具体过程

- 停止执行当前程序
- 跳转至 CSR mtvec 中的 PC 地址继续执行.
- 更新 CSR 寄存器： mcause, mepc, and mstatus
    - mstatus (mstatus[7]=mstatus[3], mstatus[3]= 0)
    - mepc (interrupt: Next PC, exception: Cur PC)
    - mcause (interrupt? ecall ? illegal inst? load fault? store fault?)
- 进入中断异常服务程序

- 停止执行当前程序
- 跳转至 CSR mepc 中的 PC 地址继续执行
- 更新 CSR mstatus.
    - mstatus (mstatus[3] = mstatus[7], mstatus[7] = 1)

1-2-3 流水线中实现

为了实现精确异常（出错指令前的指令需要完成，之后的指令不能改变机器状态），在 WB 阶段开始中断处理。也就是说，异常信息将沿流水线传递，直到写回阶段才真正开始处理中断异常，同时清空流水线的缓存防止后续指令改变机器状态。(但是一些 CSR 指令，比如 mret，会在 MEM 阶段开始处理。)

二、操作方法与实验步骤

实验中只需要补全 ExceptionUnit.v 中的代码。此外，由于实验中的 CSR 寄存器在一个时钟周期内只能读/写一个数据（可以同时读一个写一个），因此定义一个状态机进行多时钟处理。

2-1 定义常量/寄存器

为了书写方便同时增强代码可读性，这里定义了一系列寄存器和常数：中断使能、异常使能、陷入指令使能，表示状态机当前状态与下一个状态的寄存器、状态机三个状态，要写入 CSR 的 epc/cause 值、mcause 含义对照、CSR 中各寄存器的编号(用于地址映射找到对应寄存器)

```
// Trap_in: exception or interrupt
    wire interrupt_en = mstatus[3]; // mie: interrupt enable
    wire exception_en = illegal_inst | l_access_fault | s_access_fault | ecall_m;
    wire trap_in = interrupt & interrupt_en | exception_en;

// State Machine
    reg[1:0] cur_state, next_state;
    localparam STATE_IDLE = 2'b00;
    localparam STATE_MEPC = 2'b01;
    localparam STATE_MCAUSE = 2'b10;

    reg[31:0] epc_to_w, cause_to_w;
    wire[31:0] mcause_value = interrupt ? 32'h8000_0000 :
                              illegal_inst ? 32'd2 :
                              ecall_m ? 32'd11 :
                              s_access_fault ? 32'd7 : 32'd5;
    localparam CSR_mstatus = 12'h300; // CSR number
    localparam CSR_mtvec = 12'h305;
    localparam CSR_mepc = 12'h341;
    localparam CSR_mcause = 12'h342;
```

2-2 决定状态

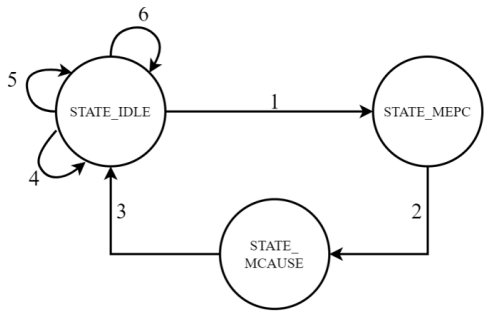
决定状态机初始状态的代码较为简单，这里略去不表。第二个模块是决定状态机下一个状态的模块。根据当前状态（cur_state）和相关信息（如 trap_in）决定状态机执行的操作和其下一个状态。

具体的实现可以分为六种情况（如下图所示）。报告中为了简洁，只展示其中一种情况的代码实现。

这里需要特别关注的是写入 mstatus 的值，以情况 1（代码展示）为例，trap_in 时，mstatus 的第三位 mie 需要置零，第七位 mpie 存储陷入前的 mie 值（mpp 也应该存储陷入前的特权模式，但是实验中陷入前后都是机器模式，不会发生特权模式切换，因此可以忽略这一点）。

Exception Unit

1. STATE_IDLE →(exception or interruption) STATE_MEPC
   - write mstatus
   - flush all the pipeline registers
   - if exception (not interrupt), cancal regwrite
   - record epc and cause
2. STATE_MEPC → STATE_MCAUSE
   - write epc to mepc
   - read mtvec
   - flush pipeline register (FD)
   - set redirect pc mux (next cycle pc → mtvec)
3. STATE_MCAUSE → STATE_IDLE
   - write cause to mcause



4. STATE_IDLE → (mret) STATE_IDLE
   - write mstatus
   - read mepc
   - set redirect pc mux (next cycle pc → mepc)
   - flush pipeline registers (EM, DE, FD)
5. STATE_IDLE → (csr insts) STATE_IDLE
   - csr operations
6. STATE IDLE → (other) STATE IDLE

```
// state transition logic
    always@(*) begin
        case(cur_state)
            STATE_IDLE: begin
                // 1.STATE_IDLE -> (trap_in))STATE_MEPC: write mepc, flush registers*
                if(trap_in) begin
                    csr_waddr = CSR_mstatus;
                    csr_wdata = {mstatus[31:8], mstatus[3], mstatus[6:4], 1'b0,
mstatus[2:0]};
                    csr_w = 1;
                    csr_wsc = 2'b01;
                    next_state = STATE_MEPC;
                end
               . . .// omitted
        endcase
    end
```

2-3 状态转移

第三个模块是根据状态机的下一个状态，对其进行状态转移。另外需要注意的是，发生异常的指令（epc）和异常原因（cause）会随指令执行发生变化，因此需要时时更新，确保在决定状态模块使用的信息都是正确的。

```
// state flip-flop
    always @(posedge clk) begin
        if(cur_state == STATE_IDLE & trap_in) begin
            epc_to_w <= interrupt ? epc_next : epc_cur; //*
            cause_to_w <= mcause_value;
        end
        cur_state <= next_state;
    end
```

2-4 输出

输出 PC 重定向地址和使能、流水线寄存器冲刷使能，以及取消写寄存器使能。

```
// output logic
    assign PC_redirect = csr_r_data_out; // PC <- mepc / mtvec
    assign redirect_mux = mret || cur_state == STATE_MEPC; // whether to redirect PC*

    assign reg_MW_flush = cur_state == STATE_MEPC || trap_in; // flush registers*
    assign reg_EM_flush = reg_MW_flush | mret;
    assign reg_DE_flush = reg_MW_flush | mret;
    assign reg_FD_flush = reg_MW_flush | mret;
    assign RegWrite_cancel = exception_en; // cancel the register write (exception)
```

三、实验结果与分析

3-1 仿真验证

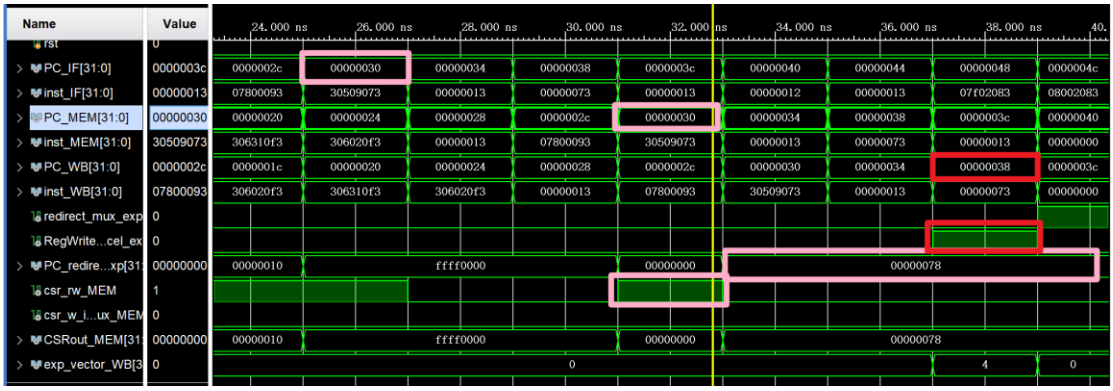
仿真结果完全符合预期，这里分别截取一次中断和异常处理并返回的过程作为佐证：

3-1-1 中断

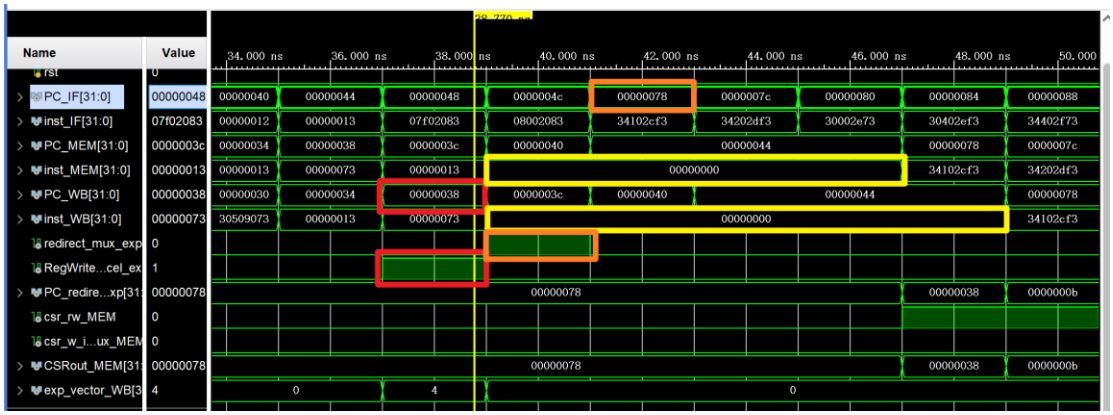
ROM.hex（trap_in）：

12	30509073	30		csrw 0x305, x1	set mtvec=0x78
13	00000013	34		addi x0, x0, 0	
14	00000073	38		ecall	
15	00000013	3C		addi x0, x0, 0	
16	00000012	40		addi x0, x0, 0	# change to illegal

指令 0x30 将 CSR 寄存器中的 mtvec 设置成 trap_handler 的地址 0x78（下图中粉色框）：
CSR 指令在 MEM 阶段执行，图中 PC_MEM 为 0x30 时，CSR 寄存器读写使能为 1，下一个时钟周期时，PC_redirection_exp 已经被成功设置成 0x78。
指令 0x38 调用 ecall（下图中红色框）：
中断异常处理在 WB 阶段开始，图中 PC_WB 为 0x38 时，取消寄存器写的使能。



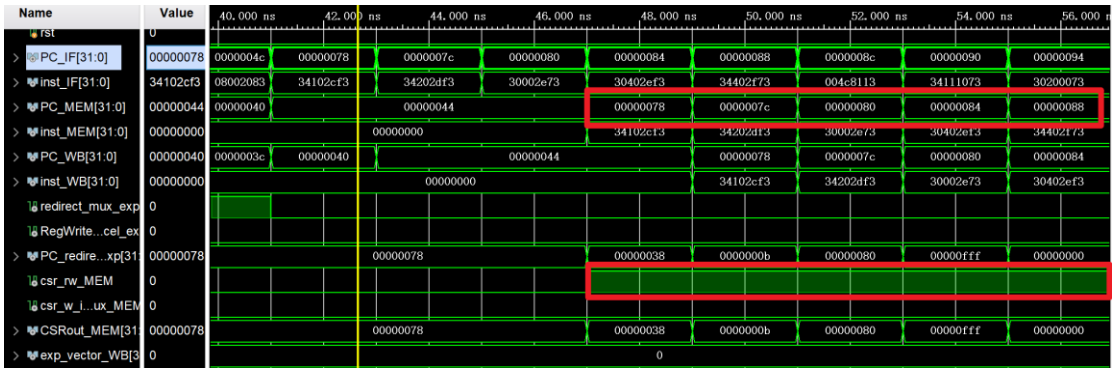
调用 ecall 后 PC 重定向使能置一，并在下一个时钟周期跳转实现陷入（下图中橙色框），同时冲刷流水线指令防止后续指令改变机器状态（下图中黄色框）。



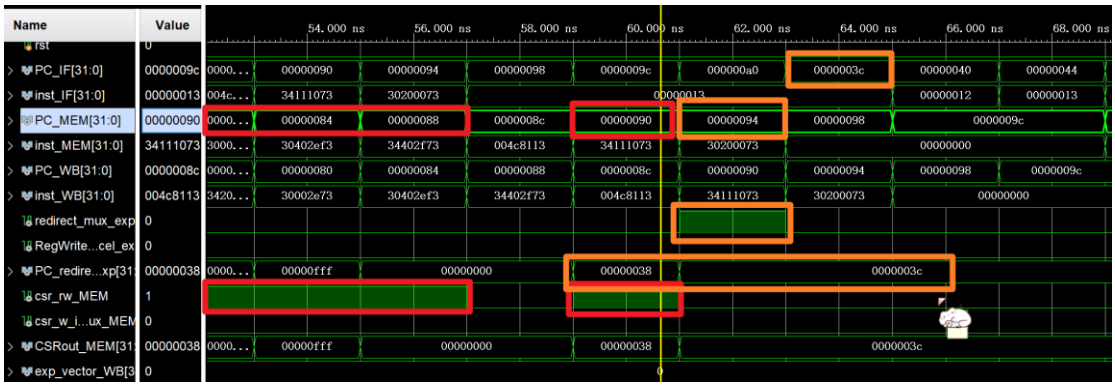
ROM.hex（mret）：

NO.	Instruction	Addr.	Label	ASM	Comment
30	34102cf3	78	trap:	csrr x25, 0x341	# mepc
31	34202df3	7C		csrr x27, 0x342	# mcause
32	30002e73	80		csrr x28, 0x300	# mstatus
33	30402ef3	84		csrr x29, 0x304	# mie
34	34402f73	88		csrr x30, 0x344	# mip
35	004c8113	8C		addi x2, x25, 4	
36	34111073	90		csrw 0x341, x2	# mepc = mepc + 4
37	30200073	94		mret	# 30200073 mret

陷入后执行 trap_handler 指令（从 **0x78** 开始直到 **0x94** 结束），其中 CSR 指令在 MEM 阶段执行（下图中红色框）：



0x90 将 mepc 加四（因为本次陷入为外部中断，不需要重复执行发生中断时所执行的指令），**0x94** 是 mret 指令，修改 PC 等使程序返回陷入前的状态（下图中橙色框）：

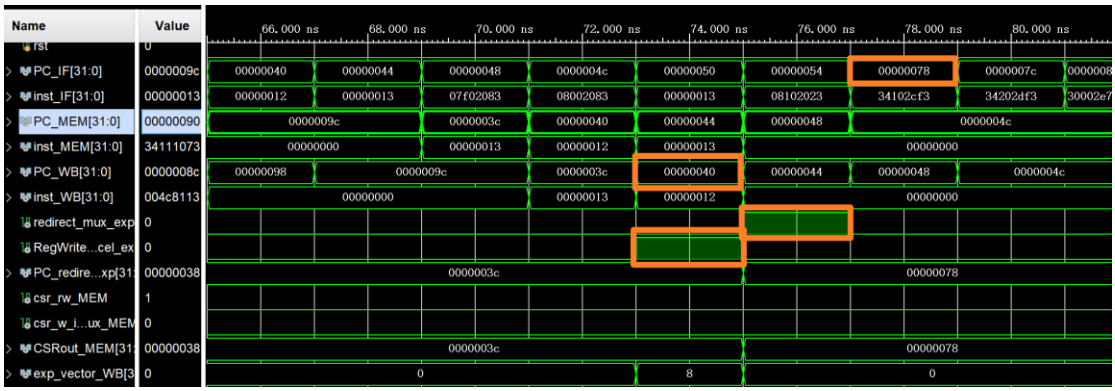


3-1-2 异常

ROM.hex：

16	00000012	40		addi x0, x0, 0	# change to illegal
17	00000013	44		addi x0, x0, 0	
18	07f02083	48		lw x1, 127(x0)	
19	08002083	4C		lw x1, 128(x0)	# l access fault
20	00000013	50		addi x0, x0, 0	
21	08102023	54		sw x1, 128(x0)	# s access fault
22	00000013	58		addi x0, x0, 0	
23	00000013	5C		addi x0, x0, 0	

同样的，WB 阶段处理 0x40 指令带来的异常（非法更改），取消寄存器写使能、重定向 PC、冲刷流水线指令等，最后成功陷入 trap_handler（下图中橙色框）：

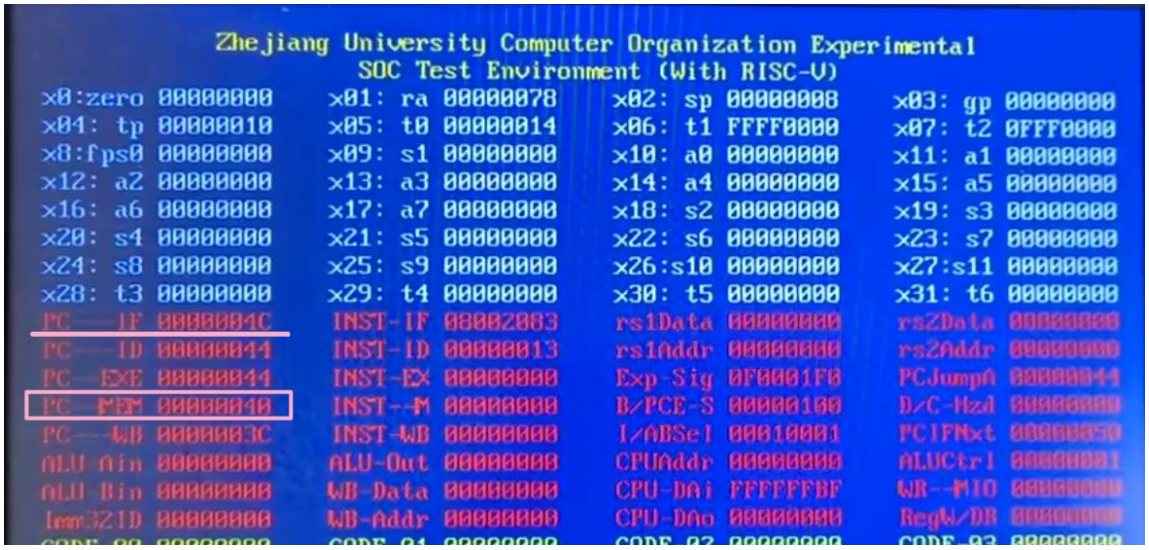


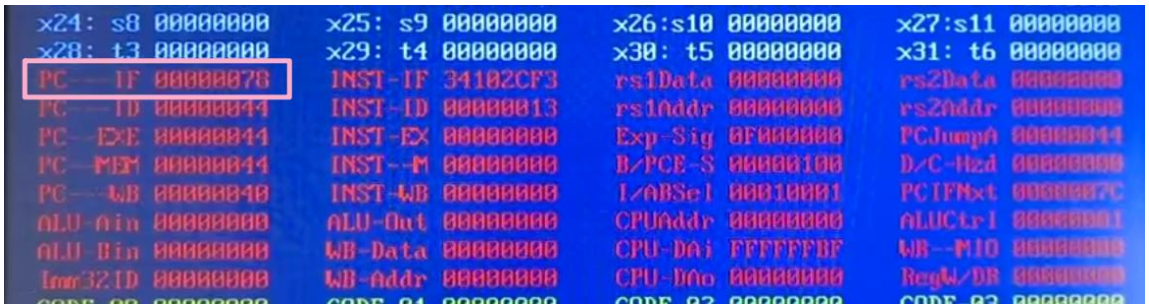
返回过程与中断类似，这里不做赘述。（需要注意的是，一般发生类似于 page fault 的异常后，需要返回发生异常的指令，将其重新执行一遍，这是中断和异常在返回这一步的一个不同点。但是本次实验中处理的异常均为指令本身的错误，重新执行指令反而会陷入死循环。）

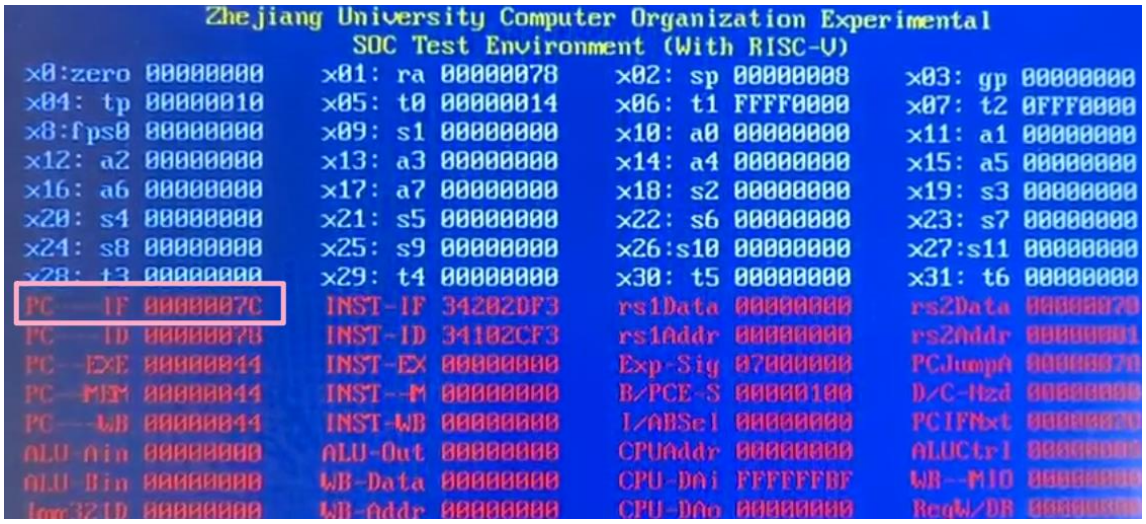
3-2 上板验证

3-1-1 中断

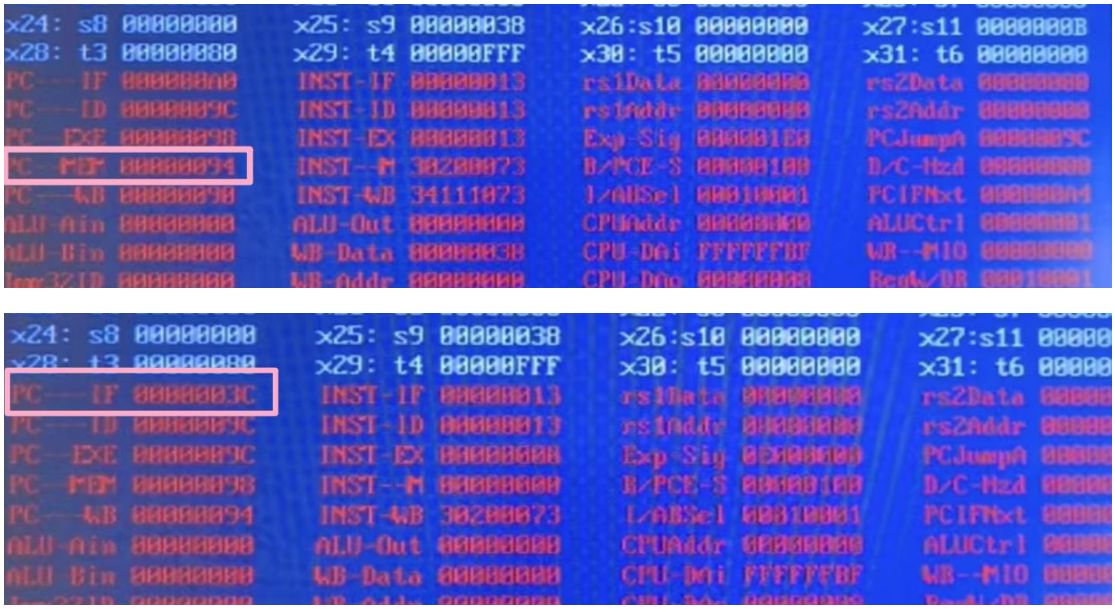
下面截取了发生中断时的几帧画面，可以看到 PC_WB 读取 0x38 的 ecall 指令，并在 PC_MEM 执行到 0x40 的指令时完成状态的保存，开启 PC 重定向（图 1），跳转到 trap_handler（图 2），随后执行中断异常处理程序（图 3）。





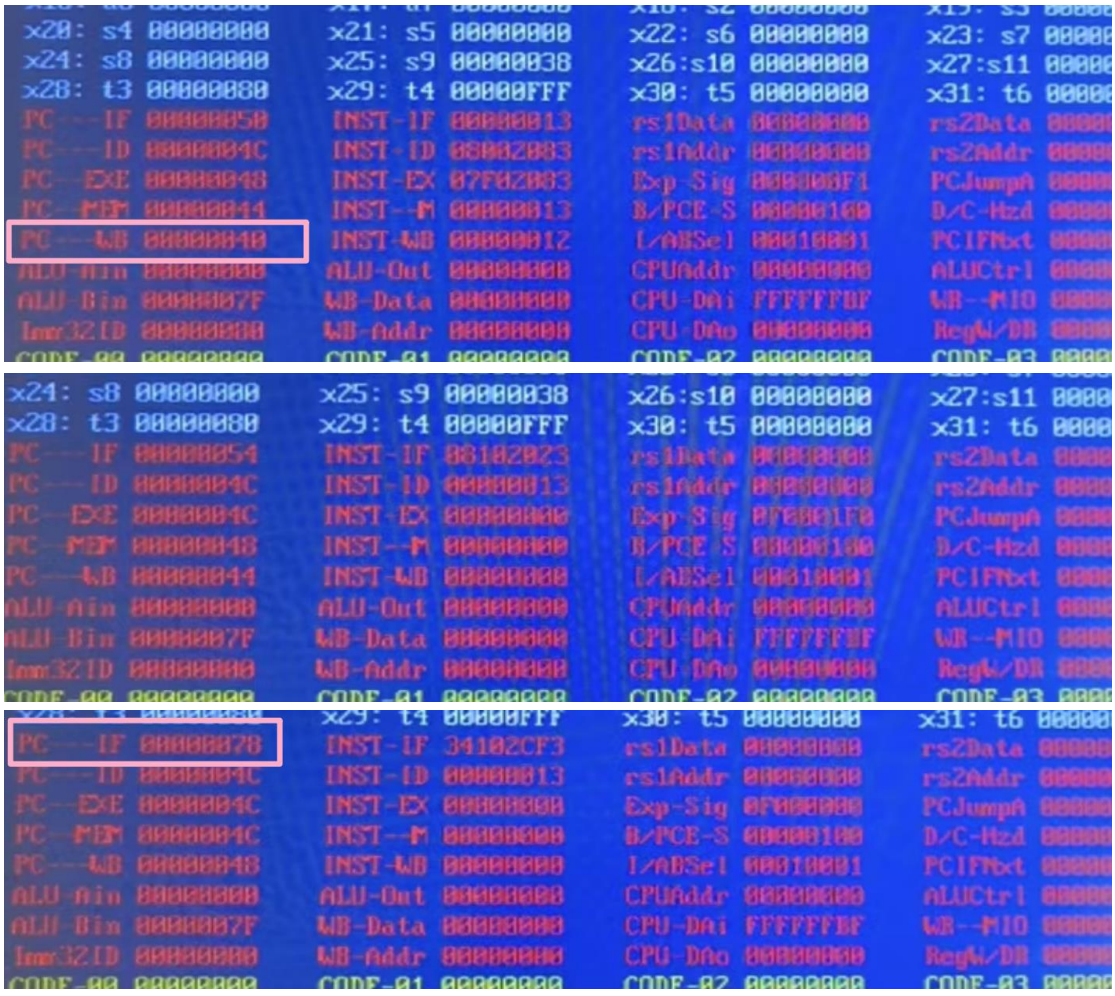


返回时，PC_MEM 指向 0x94 的 mret 指令，（图 1），恢复状态后将 PC 重定向到原来状态，跳出中断异常处理程序（图 2）。



3-1-2 异常

PC_WB 读取 0x40 的异常指令（图 1），保存状态后实现跳转（图 3）。



四、讨论、心得

本次实验给的资料内容非常详尽，包括状态机的书写也给出了例子，加上部分内容也在操作系统课上较为同步地学过，因此实验较为简单，跟着实验指导便可以完成。

其中遇到的一个小麻烦是第一次仿真的时候出现了跑通了一小部分，后面出错的问题。因为前面跑通了而且验证后发现是正确的，因此首先考虑了是 ROM 中指令的问题，在和实验指导上的指令比对后发现是 ROM.hex 最后少了几条指令导致出错。