浙江大学

# 本科实验报告

课程名称：计算机体系结构

姓　　名：汪辉

学　　院：计算机科学与技术学院

系：

专　　业：计算机科学与技术

学　　号：3190105609

指导教师：陈文智

2021 年　10 月　26 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：**Pipelined CPU supporting exception & interrupt**

学生姓名：汪辉 专业：计算机科学与技术 学号：3190105609

同组学生姓名：王嘉豪 指导老师：陈文智

实验地点：曹西 301 实验日期：2021 年 10 月 26 日

## 一、 实验目的和要求

- **Understand the principle of CPU exception & interrupt and its processing procedure.**
- **Master the design methods of pipelined CPU supporting exception & interrupt.**
- **master methods of program verification of Pipelined CPU supporting exception & interrupt.**

## 二、 实验内容和原理

- **Design of Pipelined CPU supporting exception & interrupt.**
  - Design exception unit
- **Verify the Pipelined CPU with program and observe the execution of program**

## 三、 实验过程和数据记录

### 1、理解发生异常后的操作

CSR 寄存器的地址为 12 位，但是实验只用到几个寄存器，分别是 0x300 的 status，0x305 的 mtvec，0x341 的 mepc，0x342 的 mcause 和 0x343 的 mtval，实际上实验中需要在 exception 发生时修改的寄存器就是 status、pc 和 cause 三个。

实验涉及到的 exception 一共有 3 类：ecall、非定义指令和存取错误。当 exception 发生时，无论是什么类型的，都需要经过以下步骤：取出 mtvec 里的内容作为跳转的 pc，将发生 exception 的 pc 存入 mepc，记录 status 并写入 mstatus 以及记录 cause 写入 mcause。

程序执行到跳转的 pc 后，进入一段 trap 指令段，遇到 mret 指令则跳回到发生 mepc 内的地址继续执行剩下的指令。

2、理解 CSR 寄存器

（1） Mtvec

Store the interruption handler entrance address
The base can be explained according to mode code

（2） Mepc

Save the instruction address when exception raised or interruption happens.

a) the PC indicate the instruction that raise the exception
b) the instruction need to be executed after back from interruption

（3） Mcause

If mcause[31] == 1 then the trap was caused by an interruption.
Exception code field: a code identifying the last exception.

（4） Mstatus

When an interruption/ exception raised: mstatus[7] ← mstatus[3]
When MRET is executed: mstatus[3] ← mstatus[7]

3、理解 3 类 CSR 操作指令

- **csrrw rd, csr, rs1: t←CSRs[csr], CSRs[csr]←x[rs1], x[rd]←t**
- **csrrs rd, csr, rs1: t←CSRs[csr], CSRs[csr] ←t | x[rs1], x[rd]←t**
- **csrrc rd, csr, rs1: t←CSRs[csr], CSRs[csr] ←t & ~x[rs1], x[rd]←t**
- **csrrwi rd, csr, zimm[4..0]: x[rd] ←CSRs[csr], CSRs[csr]←zimm**
- **cssrrsi rd,csr,zimm[4..0]: t←CSRs[csr], CSRs[csr]←t | zimm; x[rd]←t**
- **csrrci rd, csr, zimm[4..0]: t←CSRs[csr], CSRs[csr]←t&~zimm; x[rd]←t**

4、设计完成 Exception 模块

实验中关于 exception 信号的处理和核心模块的信号传递都已经完成，只需要根据接口信息处理发生异常后的对应寄存器。并且 Exception 单元也设计好了 CSR 寄存器模块及其读写，只需要设计合理时序使得模块有序地完成各个寄存器的读和写即可。以下结合实际代码解读：

（1）实验操作的各个寄存器的地址

```
//  the address of several CSR registers operated in the stage of exception
    parameter mie_addr = 12'h304 ;
    parameter mip_addr = 12'h344 ;
    parameter mepc_addr = 12'h341 ;
    parameter mtvec_addr = 12'h305 ;
    parameter mcause_addr = 12'h342 ;
    parameter mstatus_addr = 12'h300 ;
```

（2）模块用到的各个状态量

```
//  status to control the stage of writing in CSR
    reg[1:0] write_signal = 2'b00 ;
    reg[1:0] next_signal = 2'b00 ;

//  used to temporarily store data for writing in CSRs when exception
    reg[31:0] exception_reg = 32'b0 ;
    reg[31:0] epc_reg = 32'b0 ;
    reg[31:0] mstatus_reg = 32'b0 ;

//  used to flush the next stages of exception
    reg flush_reg = 0 ;
    reg flush_FD_reg, flush_DE_reg, flush_EM_reg, flush_MW_reg ;

//  for redirecting in exception
    reg redirect_reg = 0 ;
```

（3）exception 发生时的变量

```
wire exception = illegal_inst | l_access_fault | s_access_fault | ecall_m ;
wire[31:0] exception_code = illegal_inst ? {1'b0,31'd2} :
                            l_access_fault ? {1'b0,31'd5} :
                            s_access_fault ? {1'b0,31'd7} :
                            ecall_m ? {1'b0,31'd11} : 32'd0 ;
```

其中 exception_code 为 mcause 的值。

实验设计了两个 always 模块组合而成的电路，一个用于处理各个状态寄存器，在时钟上升沿操作，另一个在*操作，设定各个要写进 CSR 寄存器模块的值。其中，在时钟下降沿单独处理状态机的寄存器，为了能够在上升沿即让 CSRregs 模块完成读和写，这样节省了 exception 发生后的操作周期。

（4）时钟上升沿操作

```
// posedge of clk, do : flush, redirecting and updating write_signal, turn if need
    always @(posedge clk) begin
        if ( flush_reg )
            begin
                    flush_FD_reg <= 0 ;
                    flush_DE_reg <= 0 ;
                    flush_EM_reg <= 0 ;
                    flush_MW_reg <= 0 ;
                    flush_reg <= 0 ;
            end
        if ( exception ) // if exception it costs one more cycle to get into trap
            begin
                flush_reg <= 1 ;
                flush_FD_reg <= 1 ;
                flush_DE_reg <= 1 ;
                flush_EM_reg <= 1 ;
                flush_MW_reg <= 1 ;
            end

        redirect_reg <= exception ; // redirecting after one cycle when exception
        write_signal <= next_signal ; // status of signal for writing CSRs during exception
        turn_on <= next_turn ; // status of turn
    end
```

（5） 时钟下降沿

```
// negedge of clk, do : write to CSRs(mepc, mstatus, mcause) when exception -- turn_on
    if ( ~clk ) begin
            if ( turn_on ) begin
                case (write_signal )
                    2'b00:begin //write reg mepc
                        csr_waddr <= mepc_addr ;
                        csr_wdata <= epc_reg ;
                    end
                    2'b01:begin //write reg mstatus
                        csr_waddr <= mstatus_addr ;
                        csr_wdata <= mstatus_reg ;
                    end
                    2'b10:begin //write reg mcause
                        csr_waddr <= mcause_addr ;
                        csr_wdata <= exception_reg ;
                    end
                    2'b11:begin
                        next_turn <=  0;
                    end
                endcase
                 next_signal <= write_signal+1 ;
            end
    end
```

（6） Always@*

```
csr_wsc = csr_rw_in ? csr_wsc_mode_in : 2'b00 ; //mode the regs of CSR
csr_w = csr_rw_in | turn_on | mret; //enable the regs to write

if ( csr_rw_in & ~turn_on ) // for instructions of operating CSRs
    begin
        csr_waddr = csr_rw_addr_in ;
        csr_raddr = csr_rw_addr_in ;
        if (~csr_w_imm_mux)   csr_wdata = csr_w_data_reg ;
         else    csr_wdata = {27'b0,csr_w_data_imm} ;
   end
else if ( exception & ~turn_on ) // into the cycles of turn_on to write the CSRs when exception
    begin
        next_turn = 1 ; // get turn_on up next posedge of clk
        csr_raddr = mtvec_addr ; // get the instruction in the mtvec, which is the beginning of trap
        epc_reg = epc_cur ; // reg the current pc to write in CSR in next cycles
        exception_reg = exception_code ; // reg the data of mcause
        mstatus_reg = {mstatus[31:8],mstatus[3],mstatus[6:4],1'b0,mstatus[2:0]} ; // reg the status
    end
 else if ( mret & ~turn_on ) // instruction mret happens do one read and one write
    begin
        csr_raddr = mepc_addr ; // get the mepc which is the instruction to return
        csr_waddr = mstatus_addr ; // write the current status data in CSR mstatus
        csr_wdata = {mstatus[31:4],mstatus[7],mstatus[2:0]} ;
    end
```

（7） 输出信号处理

```
assign PC_redirect = csr_r_data_out ;
assign redirect_mux = redirect_reg | mret ;

assign reg_FD_flush = flush_FD_reg | exception | mret ;
assign reg_DE_flush = flush_DE_reg | exception | mret ;
assign reg_EM_flush = flush_EM_reg | exception | mret ;
assign reg_MW_flush = flush_MW_reg | exception | mret ;

assign RegWrite_cancel = exception ;
```

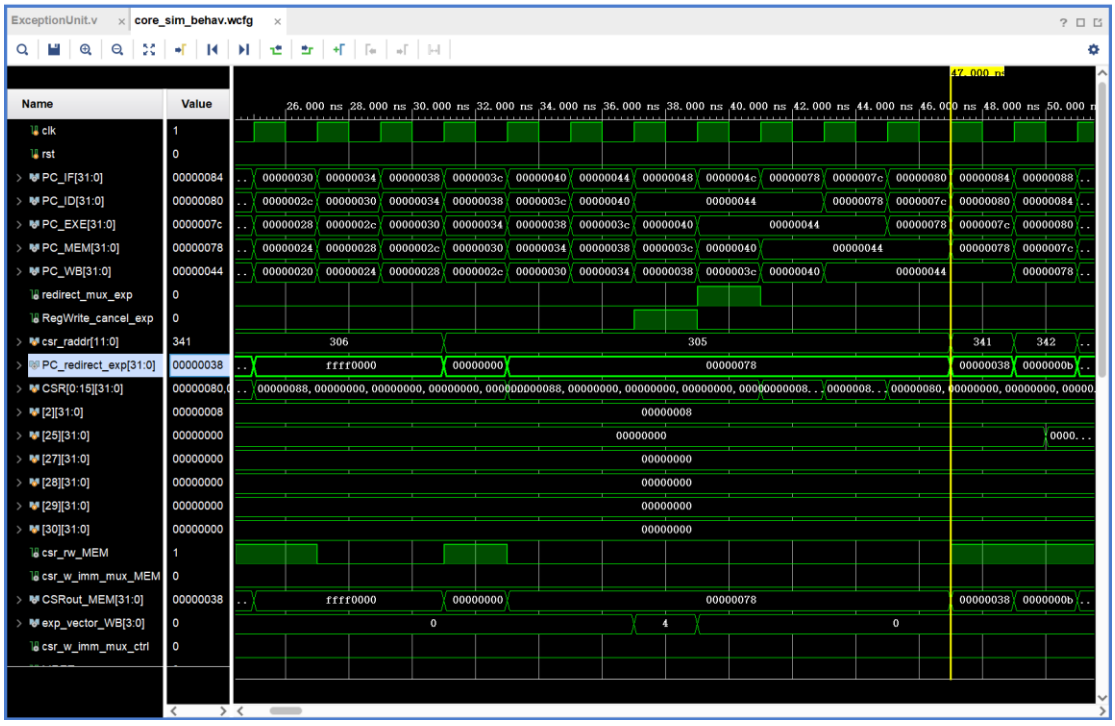
5、一些时序设计的解释

关于 flush 输出信号的处理，用到了寄存器暂存要清空的周期，这是在实验中通过仿真信号调整设计的，当 exception 发生时，该指令已经执行到了 WB 阶段，而前面阶段则相应地取到了 exception 后的 3 条指令且执行了 1 到 3 个阶段。这些阶段的中转寄存器里的内容在这个周期被 flush 掉，而后相应的执行结果仍会被存入，需要再 flush 一个周期来清空内容。Mret 指令则不同，因为实验测试指令中 mret 后设计了对应的空指令来帮助跳转，即使不做更多的 flush 也不会影响流水线跳转后的运行。

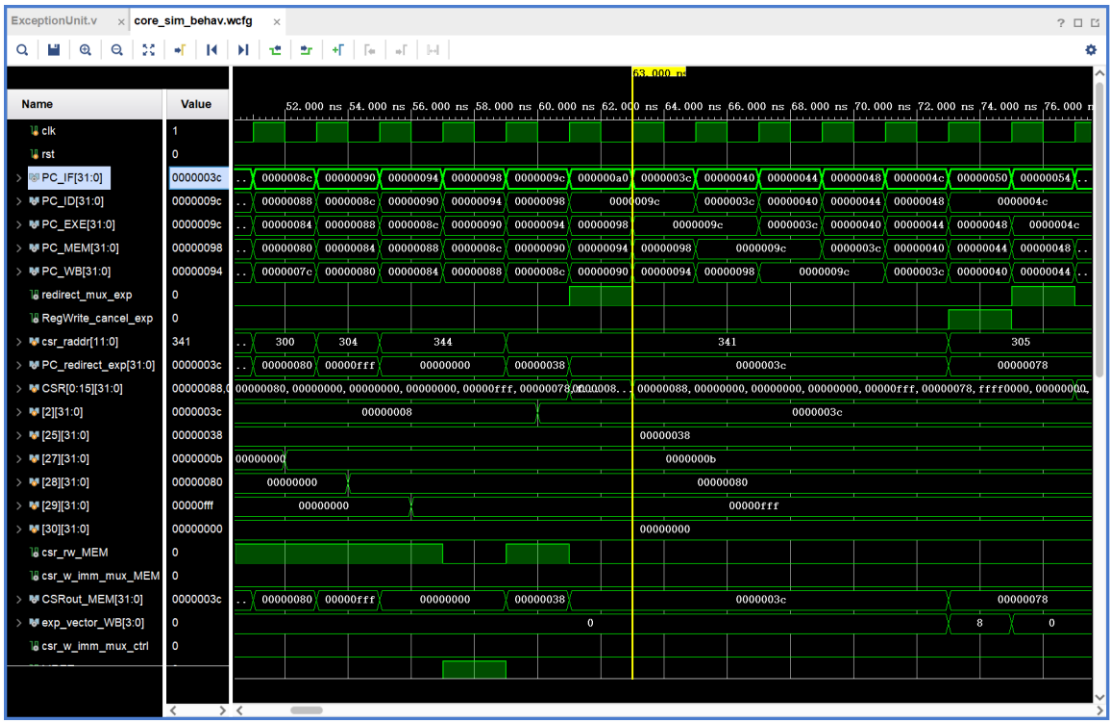
## 四、　实验结果分析

### 1. 仿真波形

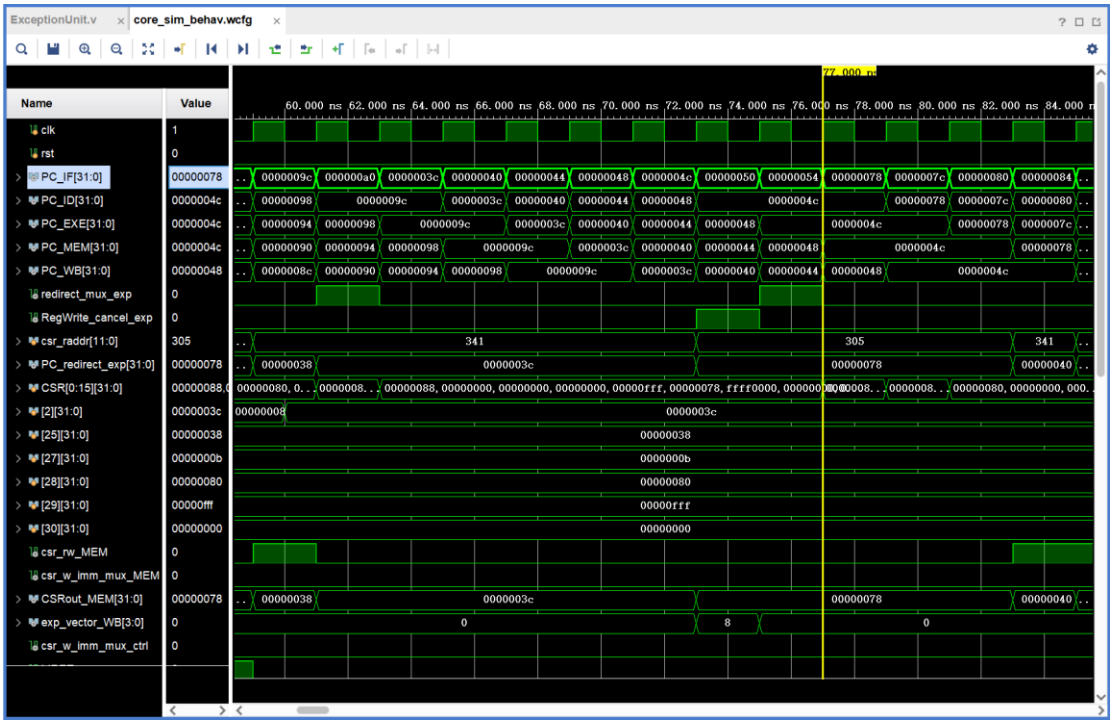
Ecall：38 跳转到 78



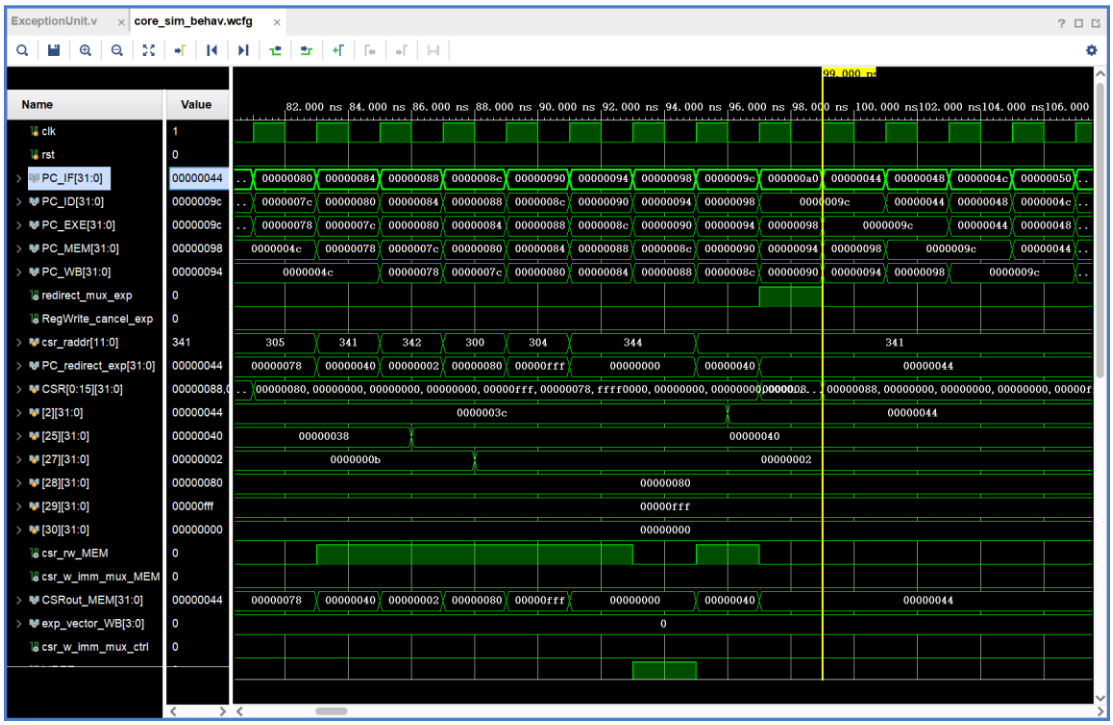

Mret：跳转回 3c




Illegal：40 跳转到 78

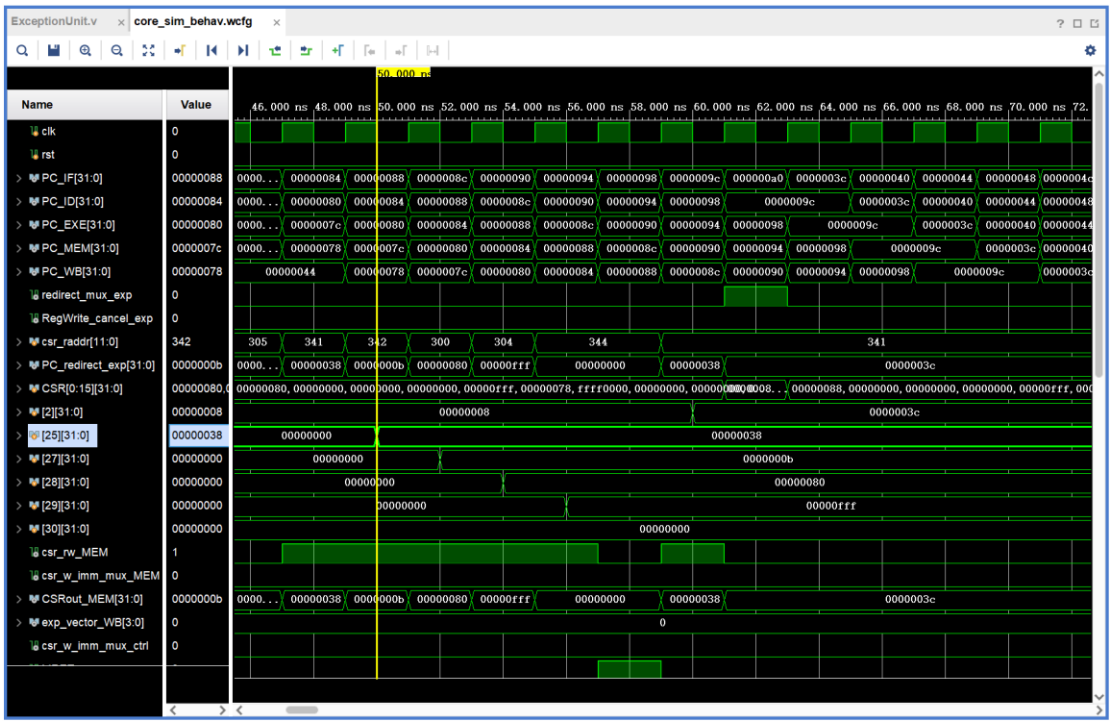


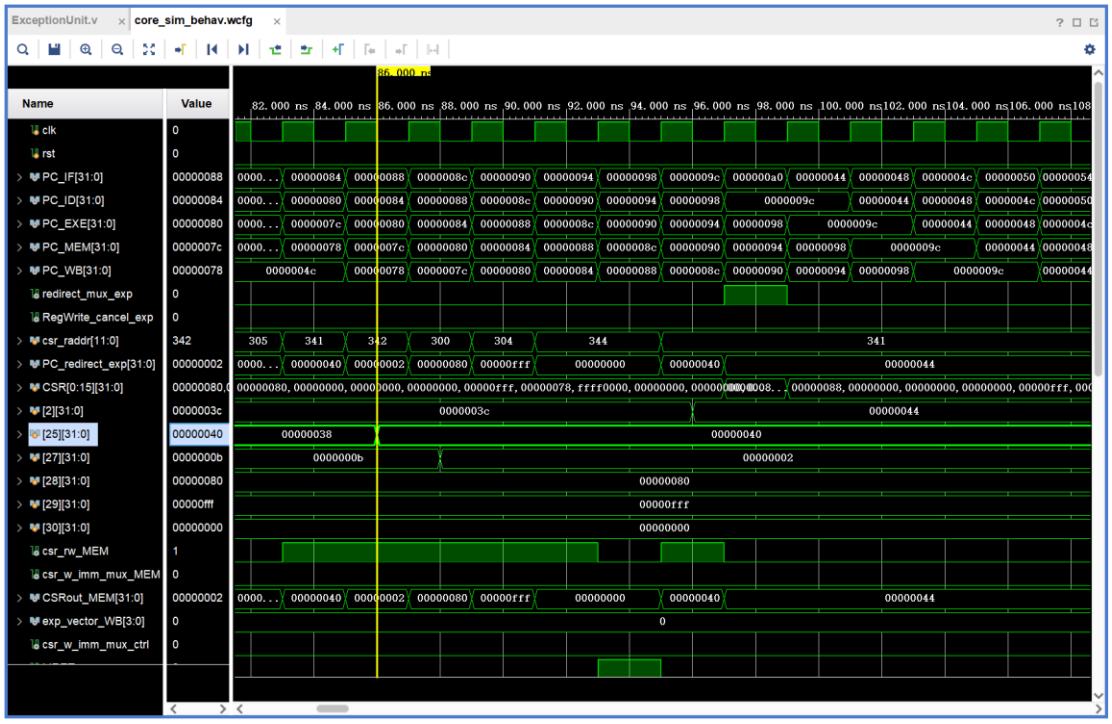

Mret：跳转回 44




后续跳转与之前相同，均正常跳转。

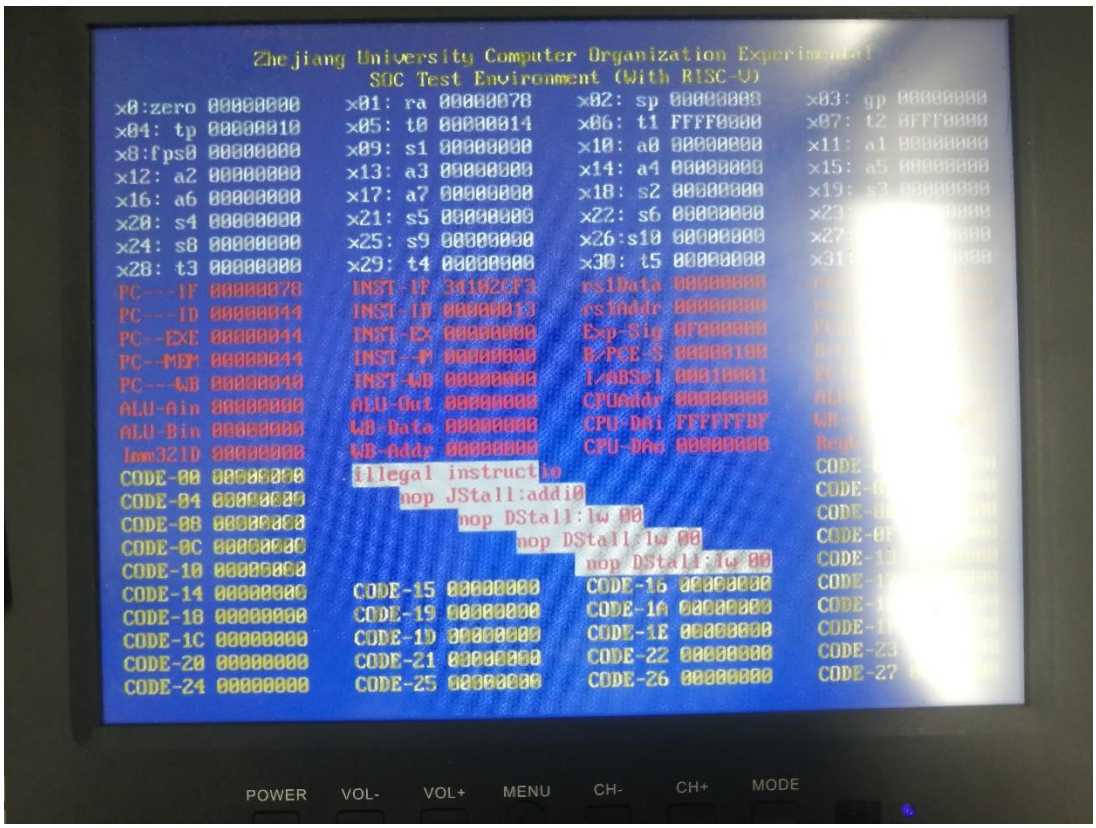
寄存器观察：


ExceptionUnit.v
core_sim_behav.wcfg
Name
Value
clk
rst
PC_IF[31:0]
PC_ID[31:0]
PC_EXE[31:0]
PC_MEM[31:0]
PC_WB[31:0]
redirect_mux_exp
RegWrite_cancel_exp
csr_raddr[11:0]
PC_redirect_exp[31:0]
CSR[0:15][31:0]
[2][31:0]
[25][31:0]
[27][31:0]
[28][31:0]
[29][31:0]
[30][31:0]
csr_rw_MEM
csr_w_imm_mux_MEM
CSRout_MEM[31:0]
exp_vector_WB[3:0]
csr_w_imm_mux_ctrl


ExceptionUnit.v
core_sim_behav.wcfg
Name
Value
clk
rst
PC_IF[31:0]
PC_ID[31:0]
PC_EXE[31:0]
PC_MEM[31:0]
PC_WB[31:0]
redirect_mux_exp
RegWrite_cancel_exp
csr_raddr[11:0]
PC_redirect_exp[31:0]
CSR[0:15][31:0]
[2][31:0]
[25][31:0]
[27][31:0]
[28][31:0]
[29][31:0]
[30][31:0]
csr_rw_MEM
csr_w_imm_mux_MEM
CSRout_MEM[31:0]
exp_vector_WB[3:0]
csr_w_imm_mux_ctrl

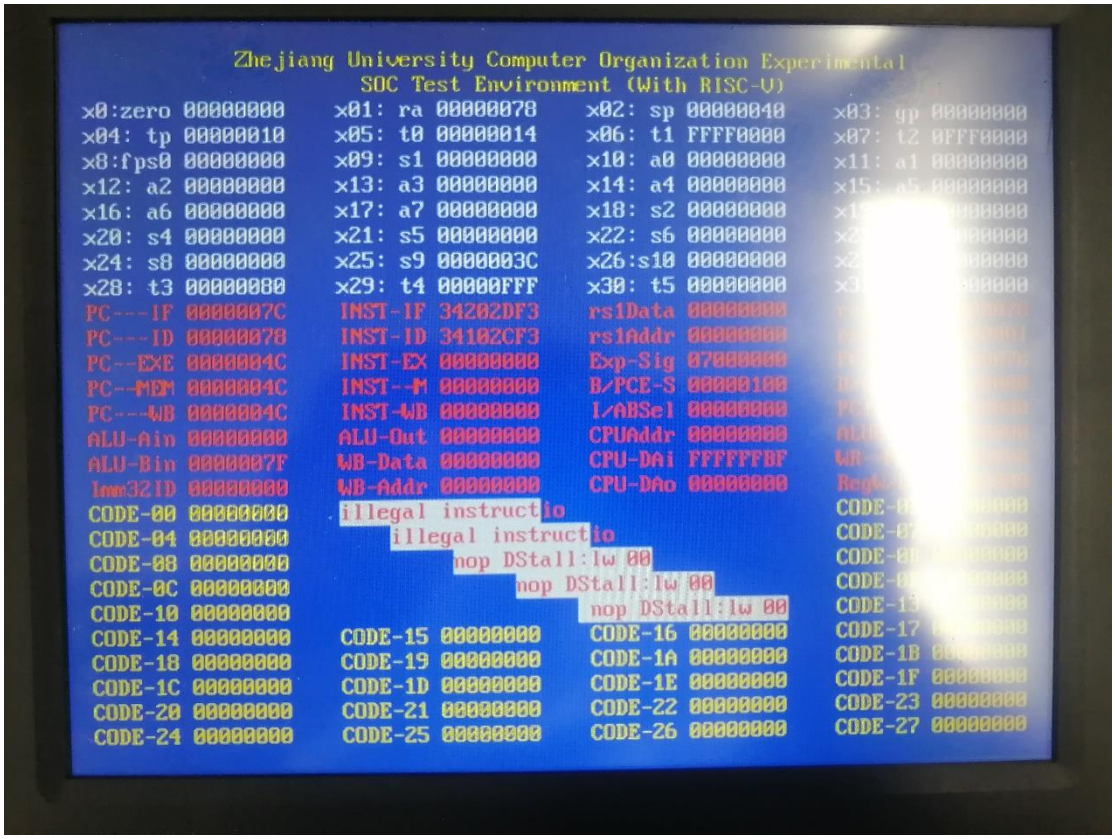
2. 实验箱记录

- 第一次跳转 ecall

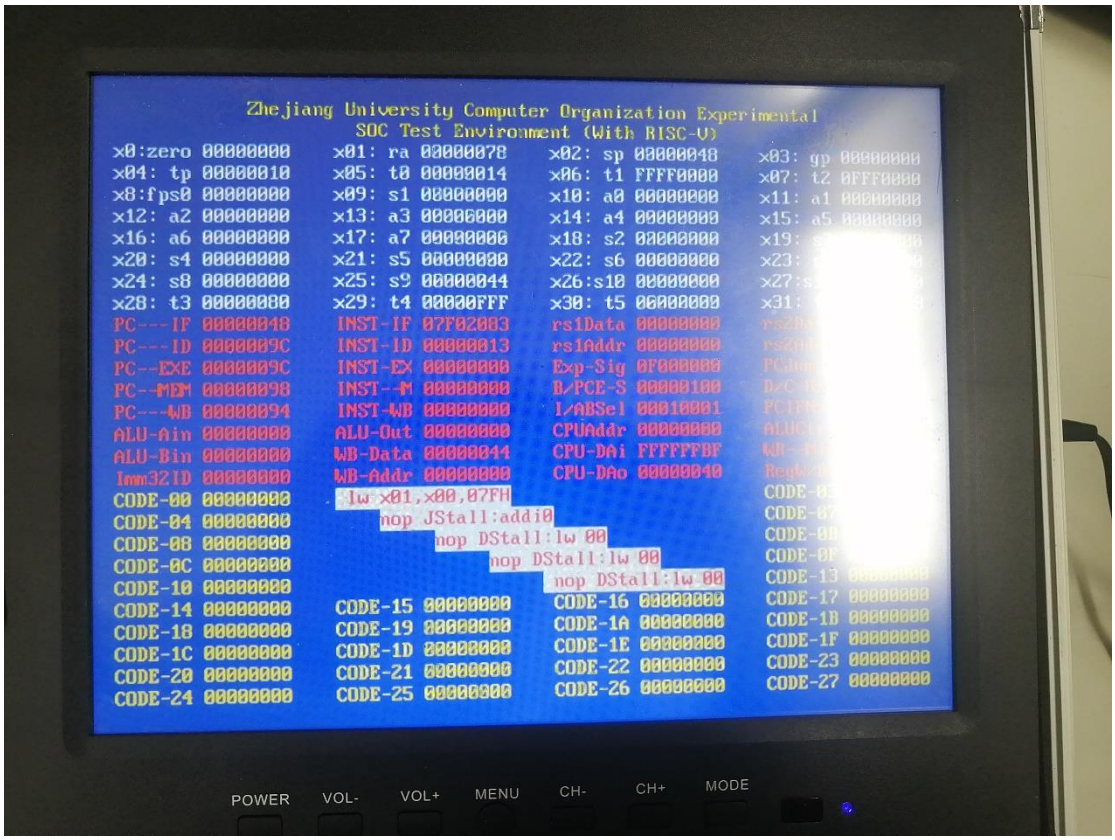



- mret

• illegal 指令跳转




• 第二次 mret



五、 讨论与心得

起初实验箱结果与仿真结果不一致，在 exception 发生时跳转失败，从助教了解到可能是由于改用了下降沿来处理信号，但是自己思考后又觉得下降沿处理的信号并没有很大的影响。仿真对实际电路的逻辑没有很严格的要求，只要信号处理合理就能得到正确的信号，后续改动了一些不严格的逻辑后才又成功在实验箱中观察到正常的跳转。一个可能的原因是我在不同的 always 模块里改动了同一个信号，这在实际电路中是会产生竞争冲突的。

此外，自己之前对于 Verilog 设计的理解不充分，也写出过 *case X 2'b00:X<=X+1;* 这样违背电路逻辑的设计，这样的写法导致了仿真在遇到这个处理时直接产生了 fatal error 而直接全部中断。经过这一次实验后，自己对 Verilog 的设计又有了进一步的理解。