浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：

学　　院：计算机科学与技术学院

　　系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：

指导教师：卜凯

2022 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：Lab2：Pipelined CPU supporting exception & interrupt

学生姓名：　　专业：计算机科学与技术　　学号：

同组学生姓名：　　指导老师：卜凯

实验地点：曹西 301　　实验日期：2022 年 10 月 26 日

1 Tasks and requirements

1.1 Tasks

The only main task of Lab-2 is:

1. Master the design methods of pipelined CPU supporting **exception** & **interrupt**.

1.2 Requirements

This experiment would be based on SWORD development board, with xc7k325tffg676-2L FPGA.

We are given a Verilog project complementing a CPU core and other components for debugging on board. Most of the parts have been finished. There is only one file we need to complete, that is:

ExceptionUnit.v, in order to deal with **CSR operations** and **traps** (interrupts or exceptions).

When this work is finished, we shall verify both the simulation results and the on-board results of a preset program.

2 Contents and principles

All of the following principles are based on RISC-V 5-stage pipelined CPU.

2.1 RISC-V privilege levels

RISC-V has 3 privilege levels until now:

1. **Machine**, usually for bootloader and other hardware demands,
2. **Supervisor**, usually for operating systems,
3. **User**, usually for applications.

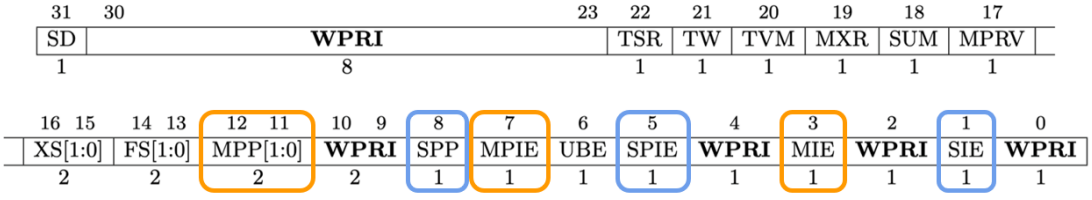
However, only Machine level is necessary; That means, Supervisor and User level may be not implemented in a system.

Switches between privilege levels are realized by **trap**. An **interrupt** or **exception** at a lower privilege level will switch the processor to a higher level to deal with that interrupt or exception. After some dealing process, the privilege level should be recovered to its original.

2.2 CSR (Control and Status Registers)

Control and status registers are one kind of registers built in CPU, storing information about the running conditions of the system. Different privilege level may access different CSRs. In this experiment we mainly intervene following CSRs:

1. **mstatus** (at 0x300). As its name implies, it stores information about the status of the machine. We mainly consider xIE and xPIE bits, whose function is highly relevant to interrupts.



mstatus.*x*IE: Interrupt Enable in *x* mode
mstatus.*x*PIE: Previous Interrupt Enable in *x* mode
mstatus.*x*PP: Previous Priviledge mode up to *x* mode

2. **mtvec** (at 0x305). This register contains information about how to deal with different traps, and may contain addresses for trap handling programs.
3. **mcause** (at 0x342). This register contains information about the cause of the trap. It should be automatically written when trap occurs. Following is a table that defines how **mcause** should be written when entering a trap.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

4. **mepc** (at 0x341). This register should contain the address that the trap handling programs should return to.

Obviously, when entering a trap, some CSRs should be automatically modified by the hardware. Somehow, CSRs can also be written or read by **CSR instructions** explicitly.

2.3 Privileged Instructions

There are some instructions that can only be used for a certain privilege level. In this experiment we will deal with **ecall** and **sret** instructions.

ecall instruction is used to generate a software interrupt. The user should first write some information about the interrupt in some registers to let the system run the corresponded handling program.

mret instruction is used to return to Supervisor level from Machine level. Of

course, it should change some data in the CSRs.

Likewise, there is a **sret** instruction, but we don’t need to implement it in this experiment.

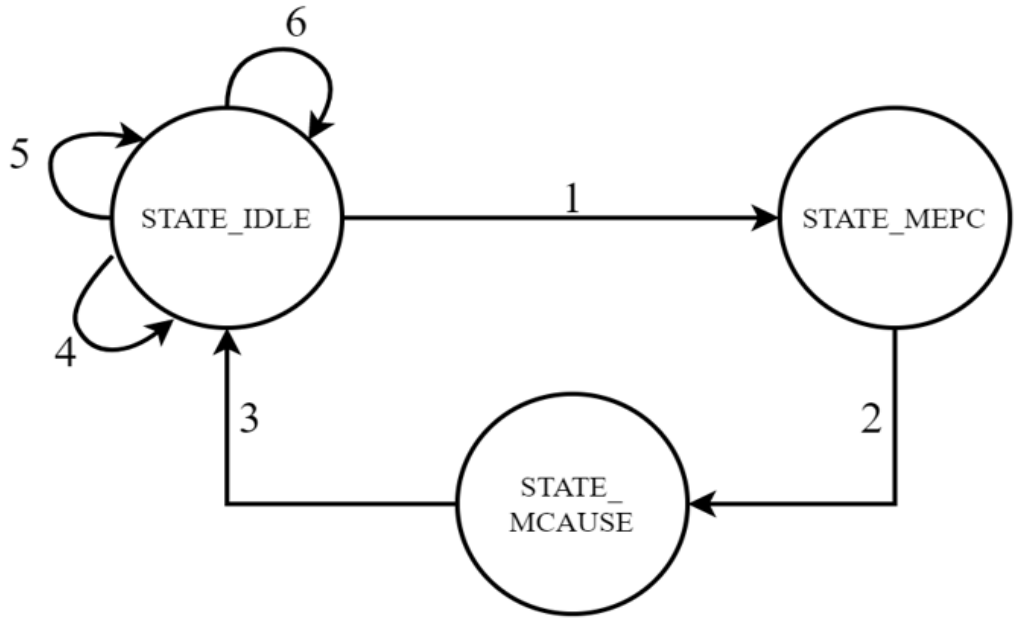
2.4Trap Handling Process & Implement on Pipelined CPU

When a trap occurs, we need to assure the following principles or steps to keep correct running status:

1. Finish all instructions before the trap correctly, and flush the instruction which caused that trap and its following instructions.
2. Store the current running status to CSRs, and jump to the address that **mtvec** defined.
3. After a sret instruction, we remodify the CSRs, and jump to the address that **mepc** defined to recover running status.

In this experiment, we are to find the interrupt instructions at **WB** stage, illegal instructions at **ID** stage, and memory fault at **MEM** stage. That means we need to flush at most 4 instructions. As guidebook suggested, we enter trap at **WB** stage. However, this may cause some problems, and we will discuss it later.

Each trap handling will cost 3 cycles to finish all writings to CSRs, so we introduce an automaton:



The transactions of the automaton will be discussed in more detail afterwards.

3 Steps and data records

3.1 Completed Verilog source files

3.1.1 ExceptionUnit.v

In the predefined parts there contains CSR registers and some inputs and outputs of this unit. As we are to finish other parts, we omit corresponding code.

We first define some constants for use. These constants stand for states of the automaton and addresses of CSRs.

```
parameter STATE_IDLE = 2'b00;
parameter STATE_MEPC = 2'b01;
parameter STATE_MCAUSE = 2'b10;

parameter MSTATUS = 12'h300;
parameter MIE = 12'h304;
parameter MTVEC = 12'h305;
parameter MEPC = 12'h341;
parameter MCAUSE = 12'h342;
parameter MTVAL = 12'h343;
parameter MIP = 12'h344;
```

And then we define one wire to represent trapping in signal, and some registers for storing some useful variables during cycles. After them there is a state register to identify current state of the automaton.

```
wire trap_in = interrupt | illegal_inst | l_access_fault | s_access_fault | ecall_m;

reg[31:0] epc_cur_IDLE;
reg[31:0] epc_next_IDLE;
reg interrupt_IDLE;
reg illegal_inst_IDLE;
```

```
reg l_access_fault_IDLE;
reg s_access_fault_IDLE;
reg ecall_m_IDLE;
reg trap_in_IDLE;

reg interrupt_MEPC;
reg illegal_inst_MEPC;
reg l_access_fault_MEPC;
reg s_access_fault_MEPC;
reg ecall_m_MEPC;

reg[1:0] state;
```

We would store values to use across cycles; And then we also consider state transfers of the automaton, because they should both be implemented under sequential logic.

If reset signal is invalid, then we use the following codes to proceed; or we reset all these values to 0 (also represent **STATE_IDLE**). The codes are all in an **always** block triggered by **posedge clk**.

The automaton works as follows:

If the state is **STATE_IDLE**, then it keeps, until **trap_in** or **mret** signal is set. If set, the state transfers to **STATE_MEPC**.

If the state is **STATE_MEPC**, then it must transfer to **STATE_MCAUSE**.

And, if the state is **STATE_MCAUSE**, then it must transfer to **STATE_IDLE**.

```
always@(posedge clk) begin
  if(rst) begin
    epc_cur_IDLE <= 0;
    epc_next_IDLE <= 0;
    interrupt_IDLE <= 0;
    illegal_inst_IDLE <= 0;
    l_access_fault_IDLE <= 0;
```

```
        s_access_fault_IDLE <= 0;
        ecall_m_IDLE <= 0;
        trap_in_IDLE <= 0;

        interrupt_MEPC <= 0;
        illegal_inst_MEPC <= 0;
        l_access_fault_MEPC <= 0;
        s_access_fault_MEPC <= 0;
        ecall_m_MEPC <= 0;

        state <= 0;
        end
    else begin
        epc_cur_IDLE <= epc_cur;
        epc_next_IDLE <= epc_next;
        interrupt_IDLE <= interrupt;
        illegal_inst_IDLE <= illegal_inst;
        l_access_fault_IDLE <= l_access_fault;
        s_access_fault_IDLE <= s_access_fault;
        ecall_m_IDLE <= ecall_m;
        trap_in_IDLE <= trap_in;

        interrupt_MEPC <= interrupt_IDLE;
        illegal_inst_MEPC <= illegal_inst_IDLE;
        l_access_fault_MEPC <= l_access_fault_IDLE;
        s_access_fault_MEPC <= s_access_fault_IDLE;
        ecall_m_MEPC <= ecall_m_IDLE;

        case(state)
        STATE_IDLE: begin
```

```
        if(trap_in | mret) state <= STATE_MEPC;
        else state <= STATE_IDLE;
      end
      STATE_MEPC: begin
        state <= STATE_MCAUSE;
      end
      STATE_MCAUSE: begin
        state <= STATE_IDLE;
      end
      endcase
    end
end
```

Next are the operations we need to do at each state of the automaton.

If reset signal is valid, we set CSR operations invalid, and do nothing else. If invalid, we operate as the following.

If the state is **STATE_IDLE**, we will consider several circumstances:

If **trap_in** signal is valid, we store MIE to MPIE and set MIE to 0 in **mstatus** register to disable Machine level interrupt.

Or if **mret** is valid, we restore the **mstatus** modified by **trap_in**, and read **mepc** to recover running status.

Or if **csr_rw_in** is valid, we do the CSR operations informed.

Or if all are invalid, we keep CSR operations invalid, and do nothing else.

If the state is **STATE_MEPC**, we set **mepc** to next instruction address (which we stored before) to be run in the original program, and read **mtvec** to jump to the trap handling program.

If the state is **STATE_MCAUSE**, we set **mcause** according to the trap reason we stored before.

```
always@* begin
  if(rst) begin
    csr_w <= 0;
```

```
    csr_raddr <= 0;
end
else begin
    case(state)
    STATE_IDLE: begin
        if(trap_in) begin
            csr_w = 1'b1;
            csr_wsc = 2'b01;
            csr_waddr = MSTATUS;
            csr_wdata = {mstatus[31:8], mstatus[3], mstatus[6:4], 1'b0, mstatus[2:0]};
        end
        else if(mret) begin
            csr_w = 1'b1;
            csr_wsc = 2'b01;
            csr_waddr = MSTATUS;
            csr_raddr = MEPC;
            csr_wdata = {mstatus[31:8], 1'b1, mstatus[6:4], mstatus[7], mstatus[2:0]};
        end
        else if(csr_rw_in) begin
            csr_w = 1'b1;
            csr_wsc = csr_wsc_mode_in;
            csr_raddr = csr_rw_addr_in;
            csr_waddr = csr_rw_addr_in;
            csr_wdata = csr_w_imm_mux ? {{27{0}}, csr_w_data_imm} :
            csr_w_data_reg;
        end
        else begin
            csr_w = 1'b0;
        end
    end
```

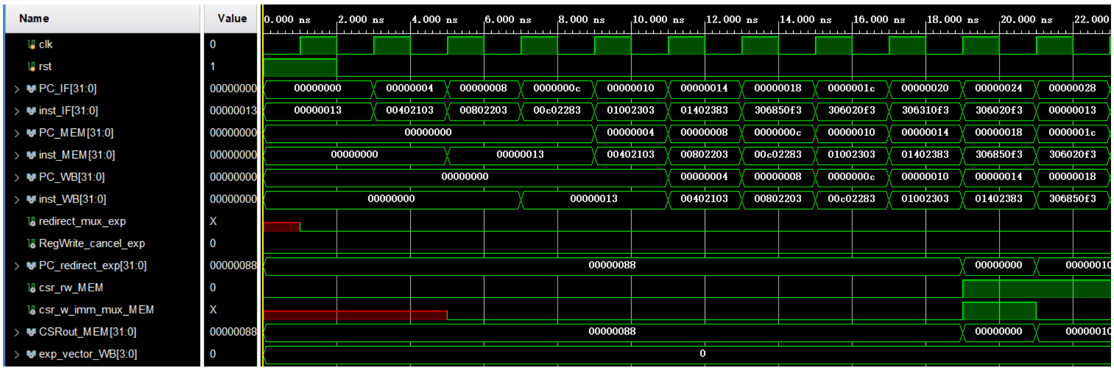
```
        STATE_MEPC: begin
            csr_w = 1'b1;
            csr_wsc = 2'b01;
            csr_waddr = MEPC;
            csr_raddr = MTVEC;
            csr_wdata = interrupt_IDLE ? epc_next_IDLE : epc_cur_IDLE;
        end
        STATE_MCAUSE: begin
            csr_w = 1'b1;
            csr_wsc = 2'b01;
            csr_waddr = MCAUSE;
            csr_wdata = {32{interrupt_MEPC}} & 32'h8000 |
            {32{illegal_inst_MEPC}} & 32'd2 |
            {32{l_access_fault_MEPC}} & 32'd5 |
            {32{s_access_fault_MEPC}} & 32'd7 |
            {32{ecall_m_MEPC}} & 32'd11;
        end
        endcase
    end
end
```

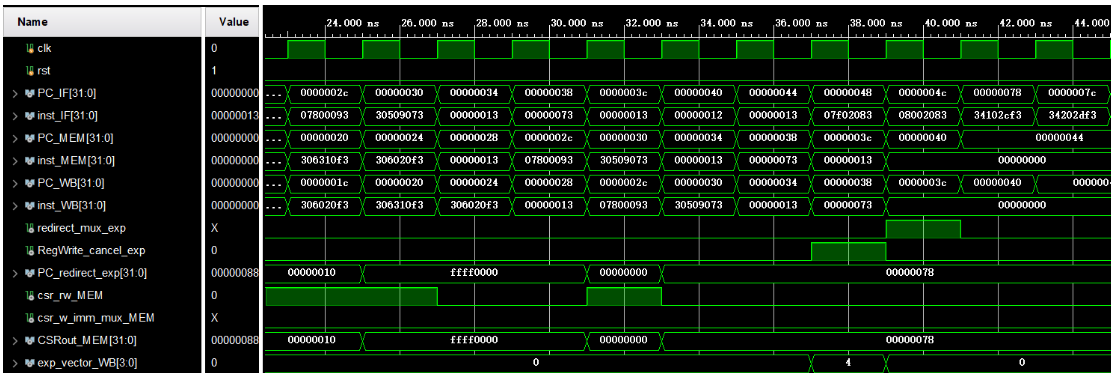
Finally, we set some control signals according to the current state. Mainly we need to flush all wrong instructions and redirect PC.

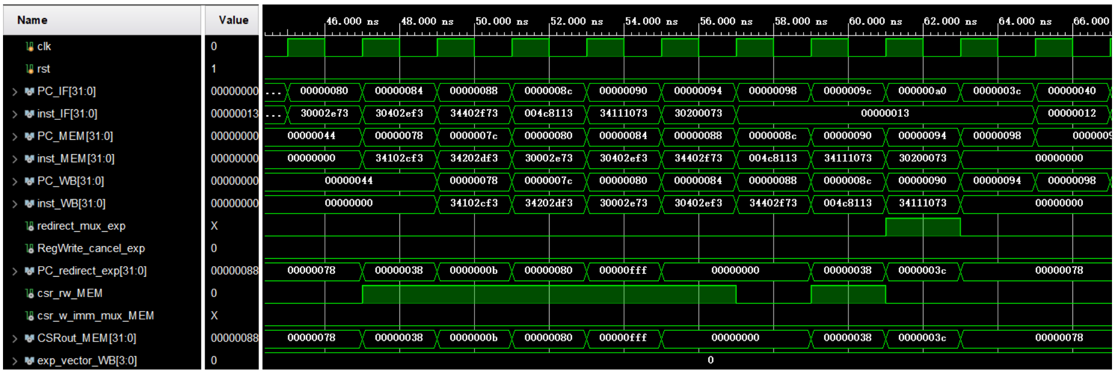
```
assign PC_redirect = csr_r_data_out;
assign redirect_mux = trap_in_IDLE | mret;

assign reg_MW_flush = trap_in | mret;
assign reg_EM_flush = trap_in | mret;
assign reg_DE_flush = trap_in | mret;
assign reg_FD_flush = trap_in | trap_in_IDLE | mret;
assign RegWrite_cancel = trap_in & ~interrupt;
```

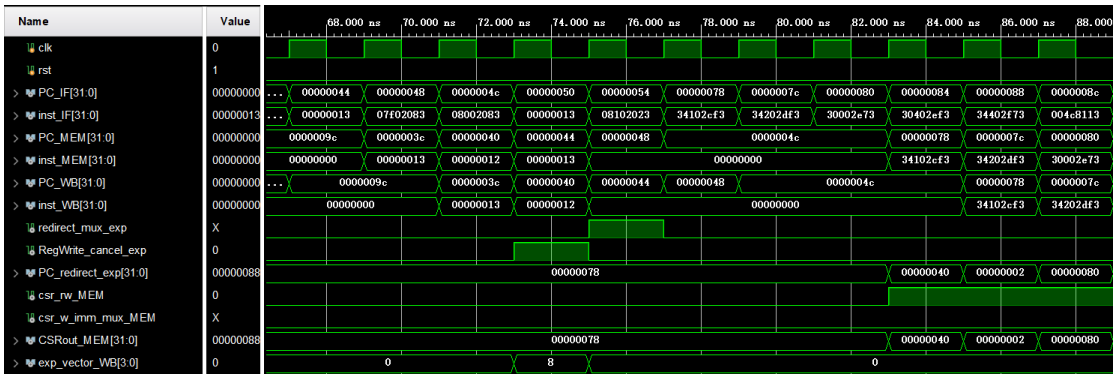
3.2 Implementation results

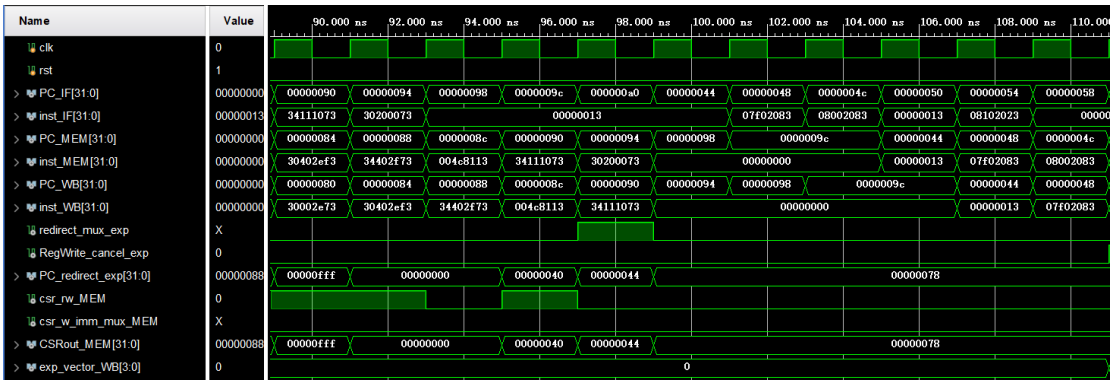
Here we record all our behavioral simulation results. Most will not be used for analysis.

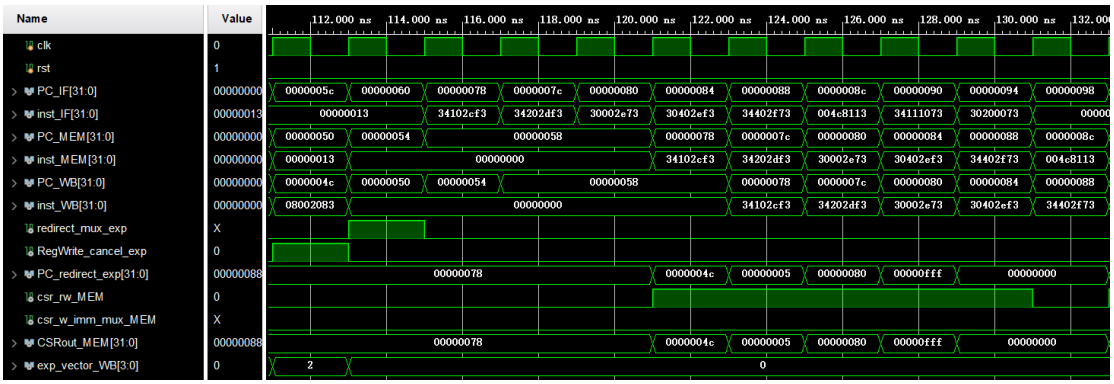


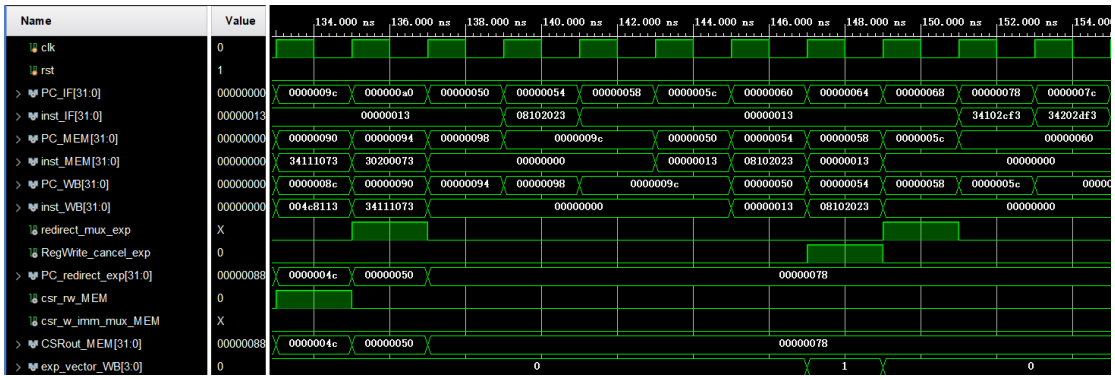


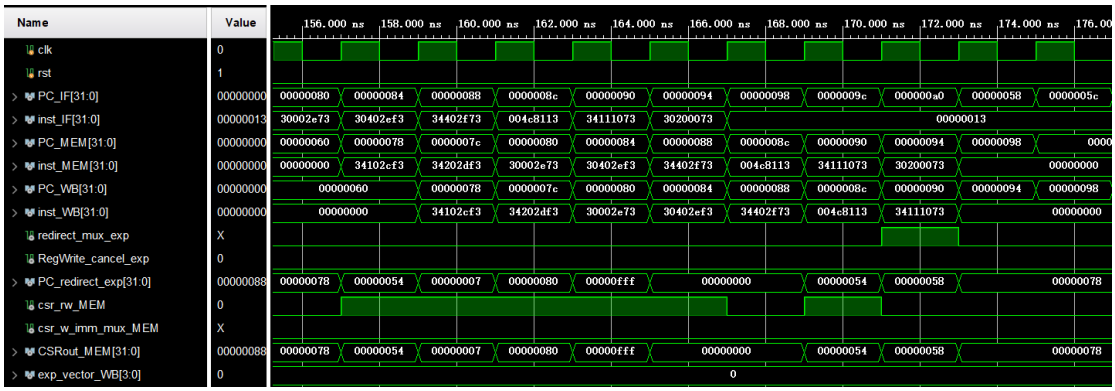


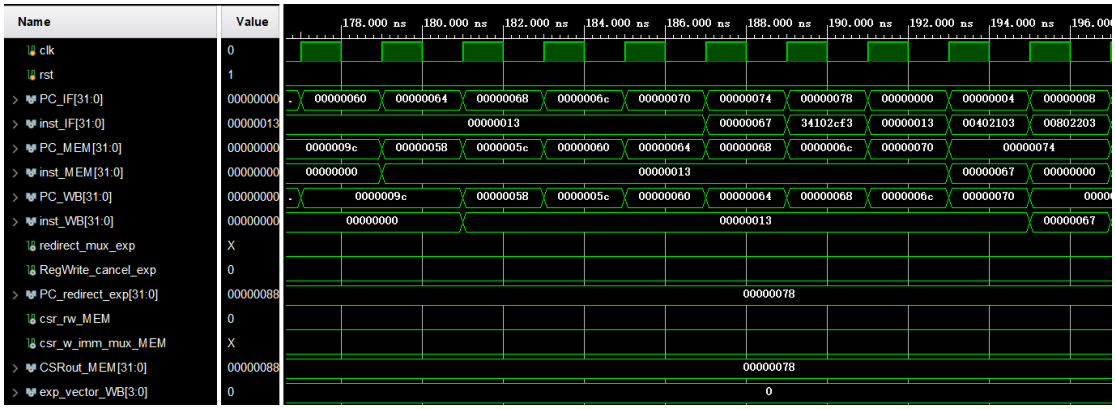

Name
Value
68.000 ns
70.000 ns
72.000 ns
74.000 ns
76.000 ns
78.000 ns
80.000 ns
82.000 ns
84.000 ns
86.000 ns
88.000
clk
rst
PC_IF[31:0]
inst_IF[31:0]
PC_MEM[31:0]
inst_MEM[31:0]
PC_WB[31:0]
inst_WB[31:0]
redirect_mux_exp
RegWrite_cancel_exp
PC_redirect_exp[31:0]
csr_rw_MEM
csr_w_imm_mux_MEM
CSRout_MEM[31:0]
exp_vector_WB[3:0]


Name
Value
90.000 ns
92.000 ns
94.000 ns
96.000 ns
98.000 ns
100.000 ns
102.000 ns
104.000 ns
106.000 ns
108.000 ns
clk
rst
PC_IF[31:0]
inst_IF[31:0]
PC_MEM[31:0]
inst_MEM[31:0]
PC_WB[31:0]
inst_WB[31:0]
redirect_mux_exp
RegWrite_cancel_exp
PC_redirect_exp[31:0]
csr_rw_MEM
csr_w_imm_mux_MEM
CSRout_MEM[31:0]
exp_vector_WB[3:0]


Name
Value
112.000 ns
114.000 ns
116.000 ns
118.000 ns
120.000 ns
122.000 ns
124.000 ns
126.000 ns
128.000 ns
130.000 ns
clk
rst
PC_IF[31:0]
inst_IF[31:0]
PC_MEM[31:0]
inst_MEM[31:0]
PC_WB[31:0]
inst_WB[31:0]
redirect_mux_exp
RegWrite_cancel_exp
PC_redirect_exp[31:0]
csr_rw_MEM
csr_w_imm_mux_MEM
CSRout_MEM[31:0]
exp_vector_WB[3:0]


Name
Value
clk
rst
PC_IF[31:0]
inst_IF[31:0]
PC_MEM[31:0]
inst_MEM[31:0]
PC_WB[31:0]
inst_WB[31:0]
redirect_mux_exp
RegWrite_cancel_exp
PC_redirect_exp[31:0]
csr_rw_MEM
csr_w_imm_mux_MEM
CSRout_MEM[31:0]
exp_vector_WB[3:0]

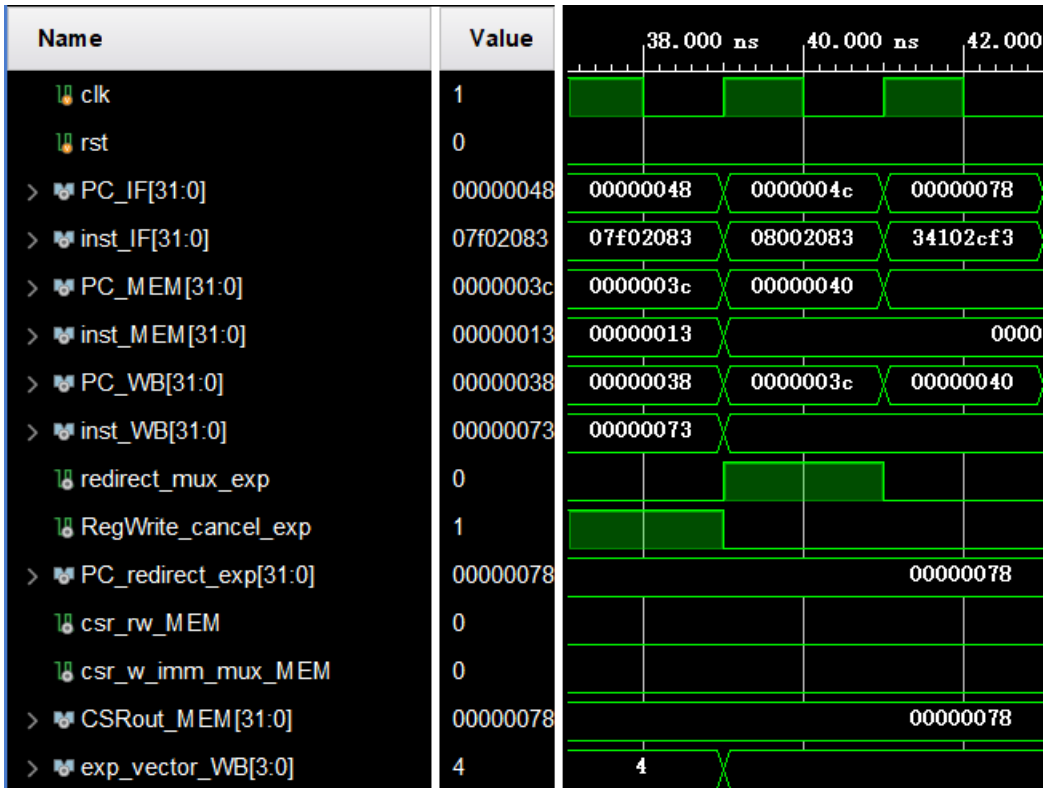

Name
Value
clk
rst
PC_IF[31:0]
inst_IF[31:0]
PC_MEM[31:0]
inst_MEM[31:0]
PC_WB[31:0]
inst_WB[31:0]
redirect_mux_exp
RegWrite_cancel_exp
PC_redirect_exp[31:0]
csr_rw_MEM
csr_w_imm_mux_MEM
CSRout_MEM[31:0]
exp_vector_WB[3:0]


Name
Value
clk
rst
PC_IF[31:0]
inst_IF[31:0]
PC_MEM[31:0]
inst_MEM[31:0]
PC_WB[31:0]
inst_WB[31:0]
redirect_mux_exp
RegWrite_cancel_exp
PC_redirect_exp[31:0]
csr_rw_MEM
csr_w_imm_mux_MEM
CSRout_MEM[31:0]
exp_vector_WB[3:0]

4 Analysis of the results

Here we only consider some typical results.

4.1 Trap by ecall



We will see the simulation in 37-43ns. During 37-39ms, the instructions in the pipeline is:

IF 0x07f02083, **lw x1, 127(x0)**

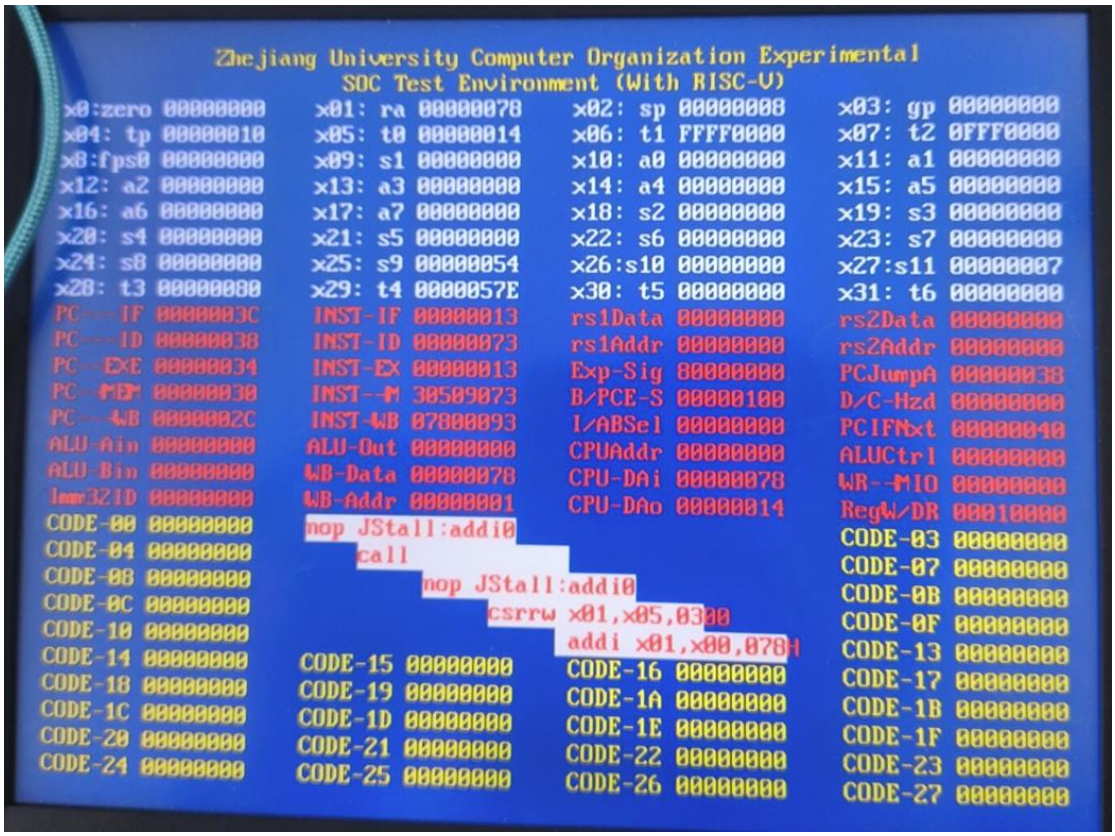
MEM 0x00000013, **addi x0, x0, 0**

WB 0x00000073, **ecall**

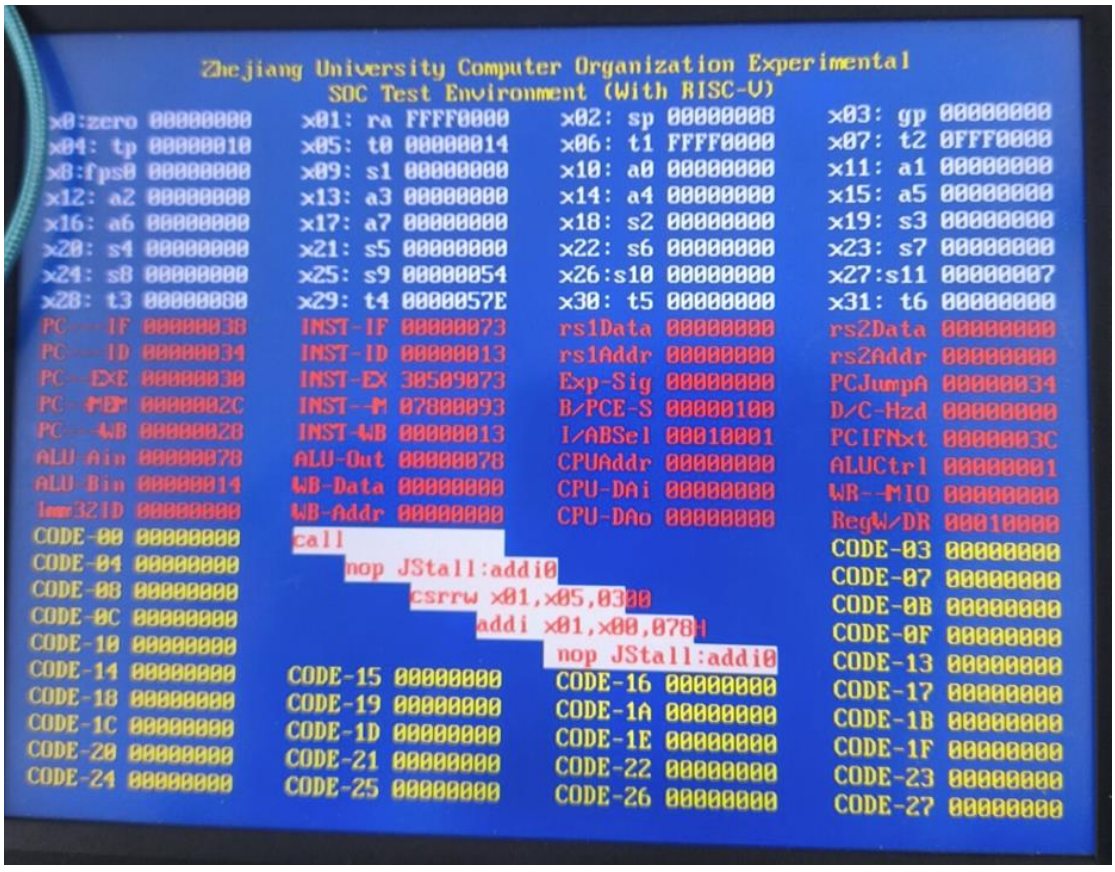
As We detected trap at WB stage, **RegWrite_cancel** was immediately set to invalid, and flush all other instructions (**STATE_IDLE**). Next cycle we see **redirect_mux_exp** set to valid, in order to begin the trap handling program, and flush all other instructions as well (**STATE_MEPC**). Finally at 41-43ns, we read out the first instruction of the trap handling program (**STATE_MCAUSE**). That first instruction is:

IF 0x34102cf3, **csrr x25, 0x341**

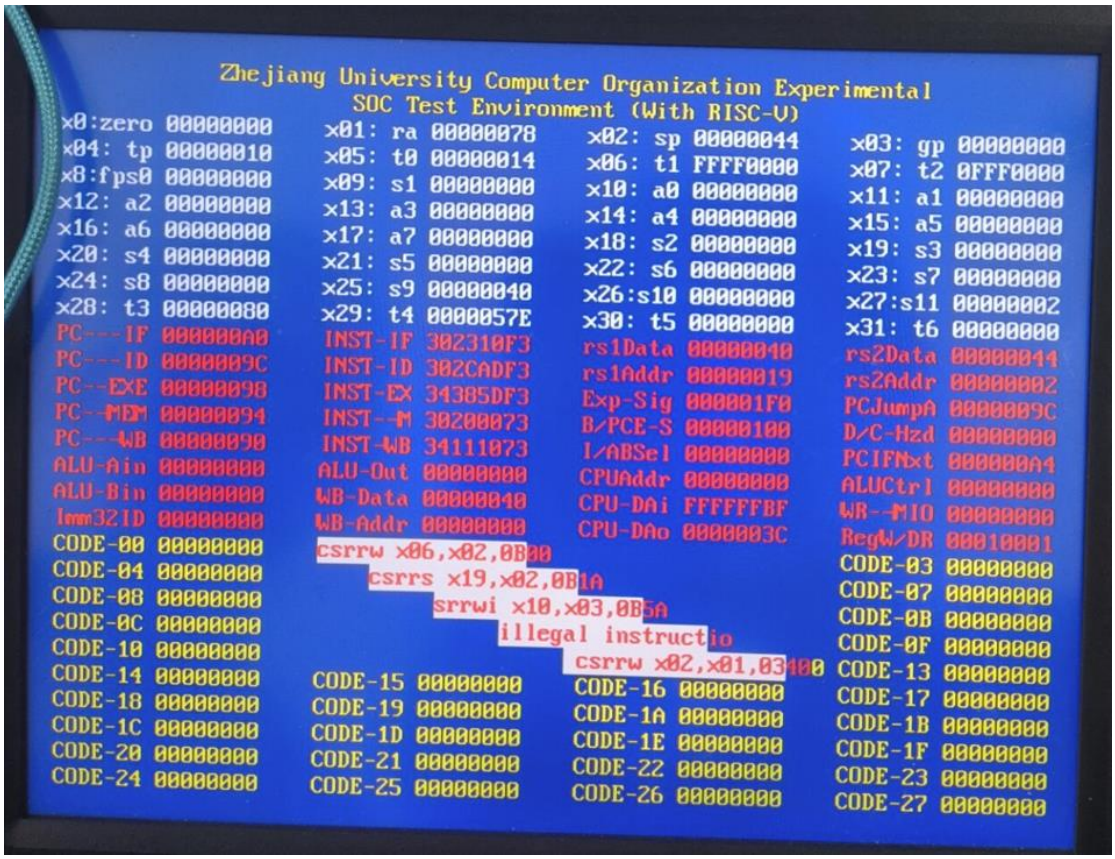
This process run correctly as we could see.



The ecall at MEM stage.

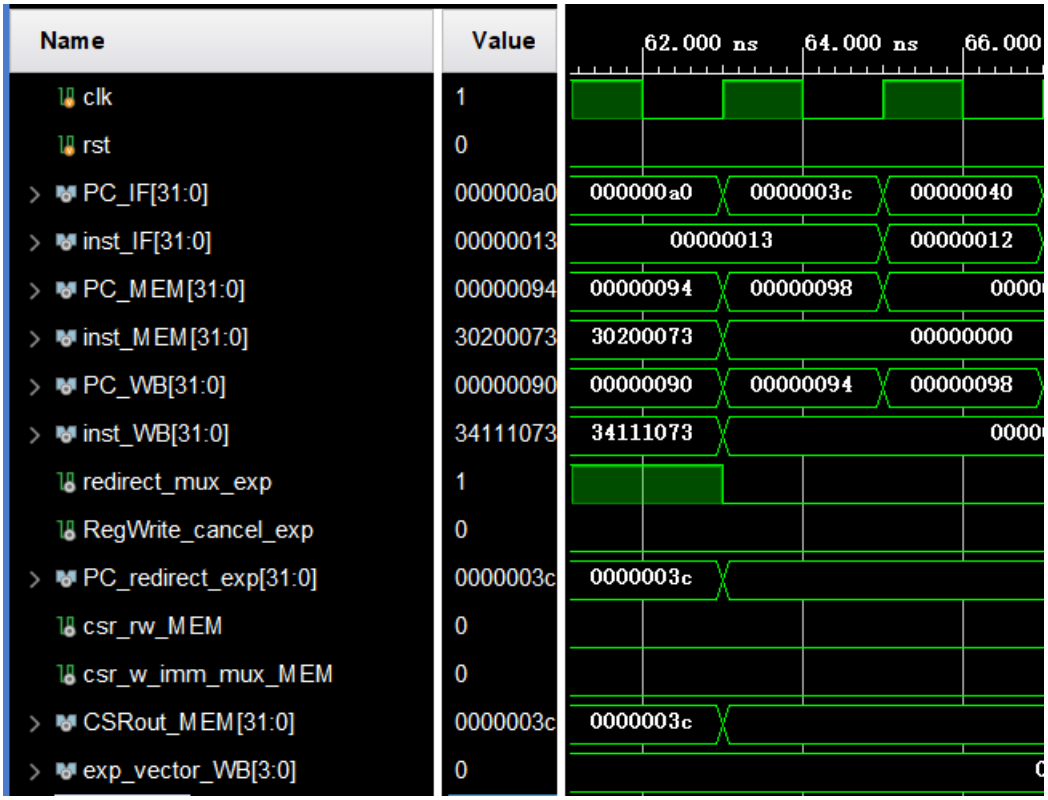


The ecall at WB stage (**STATE_IDLE**).



Trap handling at running, using CSR operations.

4.2 Returning from Trap



We will see the simulation in 61-67ns. That trap handling was caused by ecall at address 0x38. During 61-63ms, the instructions in the pipeline is:

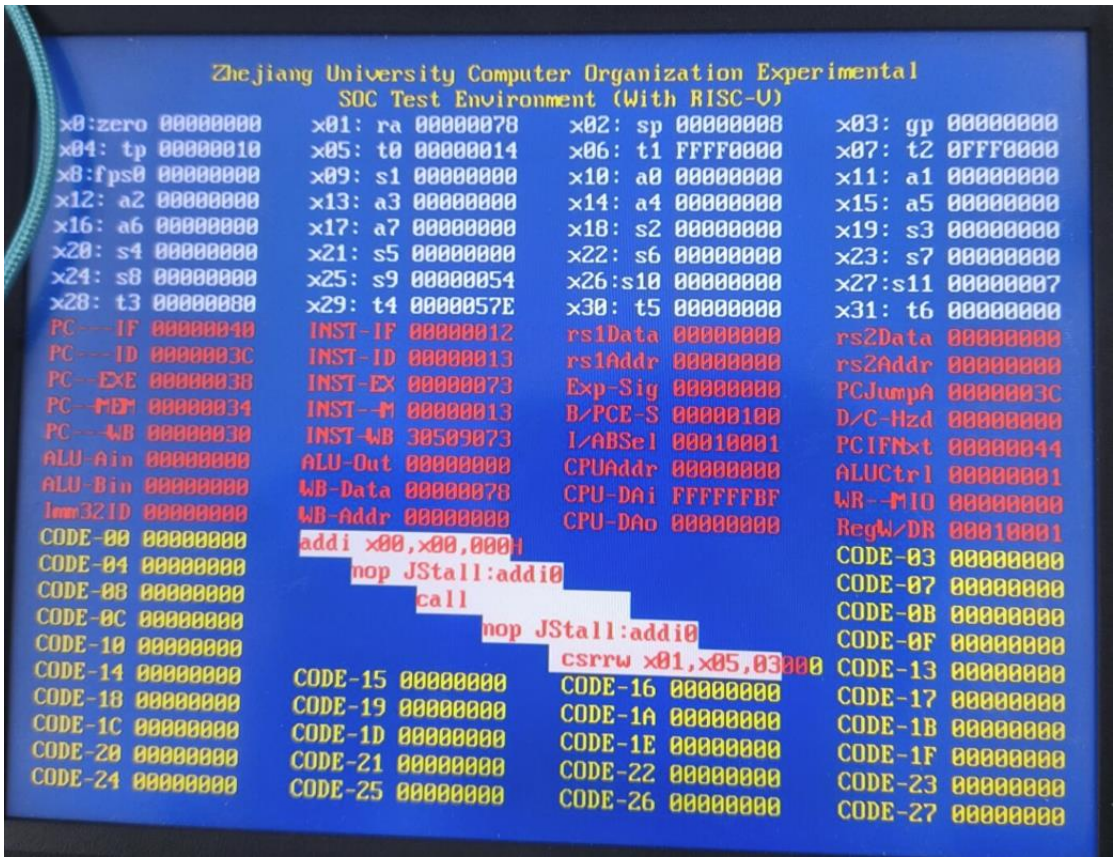
IF 0x00000013, **addi x0, x0, 0**

MEM 0x30200073, **mret**

WB 0x34111073, **csrw 0x341, x2**

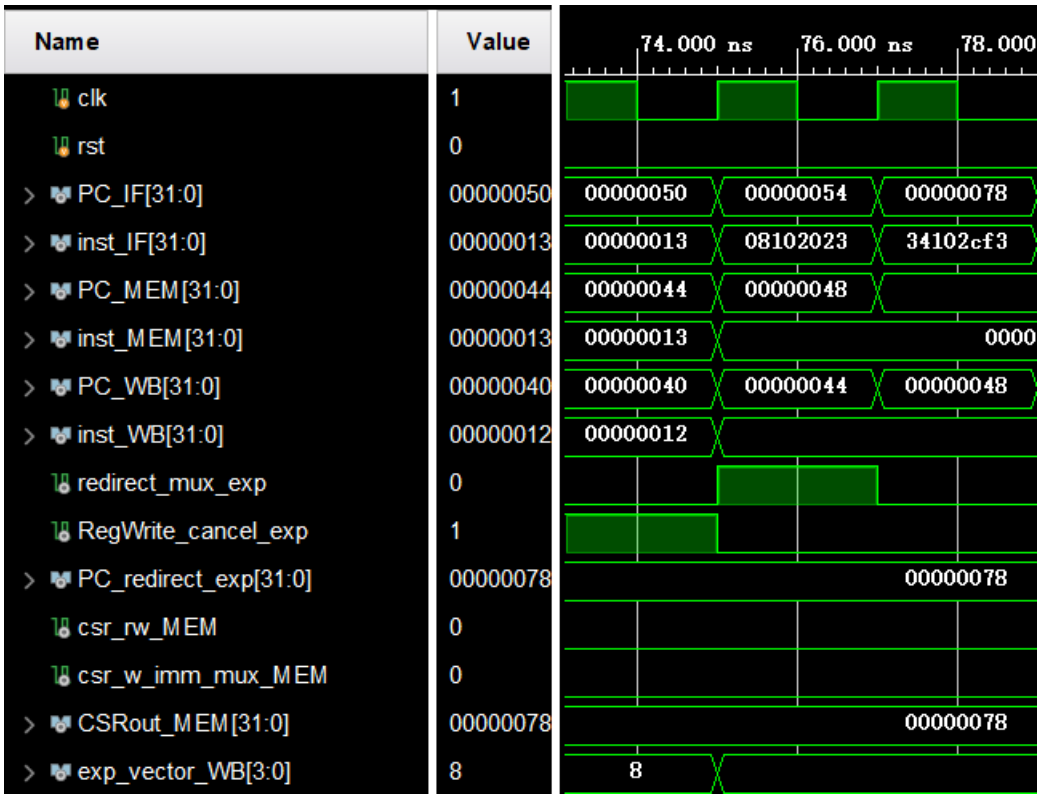
As We detected a **mret** instruction at MEM stage, we immediately set **redirect_mux_exp** to valid, and flush other instructions (**STATE_IDLE**). We read out mepc and jumped back to the original program (**STATE_MEPC**). After that we do nothing significantly intervening the running conditions of our program (**STATE_MCAUSE**).

As we could see, the address we return to is 0x3c, which was not so correct in theory because we add 4 to **mepc** in our trap handling program. And this will be discussed later.



The program control returned.

4.3Trap by Illegal Instruction



We see the simulation in 73-79ns. However, as the process is very similar to trap by ecall, we won't explain it so much.

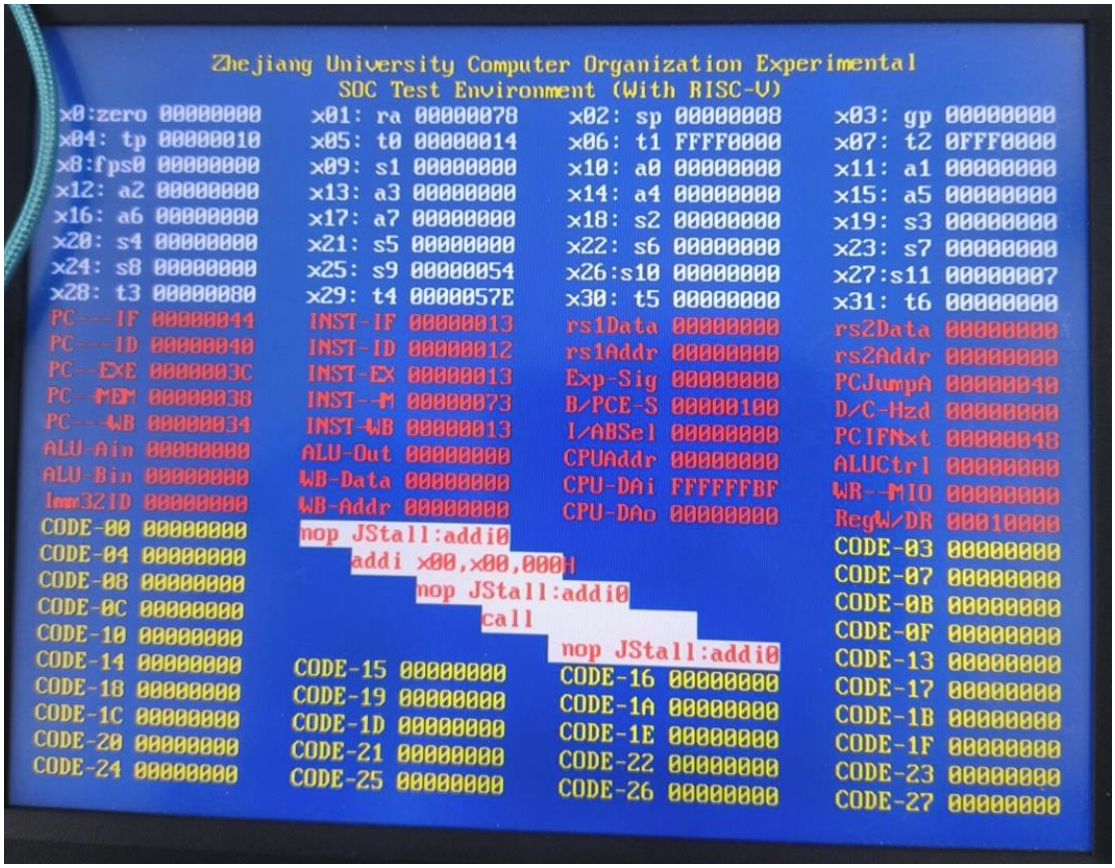
We can see the following instruction caused the trap:

WB 0x00000012, **illegal instruction**

We start automaton transfer at this time, and just as same as how we deal with trap by ecall, we do the same operations in following cycles.

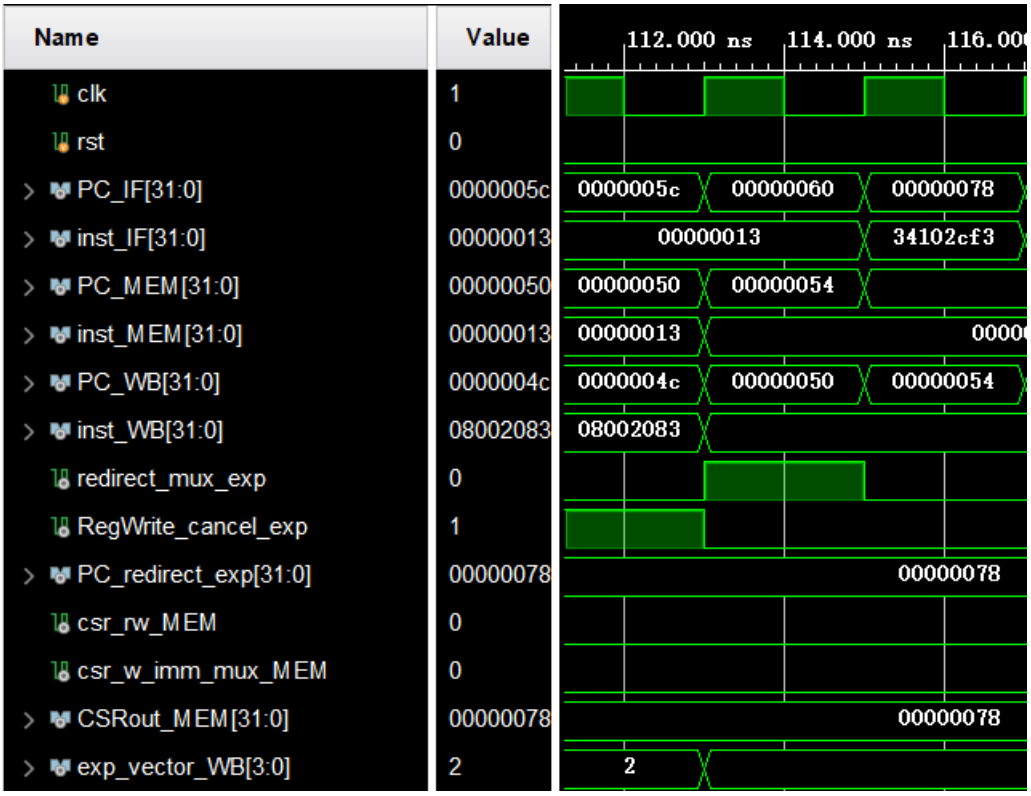
When returned, the return address is 0x44, which was a correct result in theory.

Here are some FPGA running results:



The illegal instruction at MEM stage.

4.4Trap by Memory Fault



We see the simulation in 111-117ns. However, as the process is very similar to traps above, we won't explain it so much too.

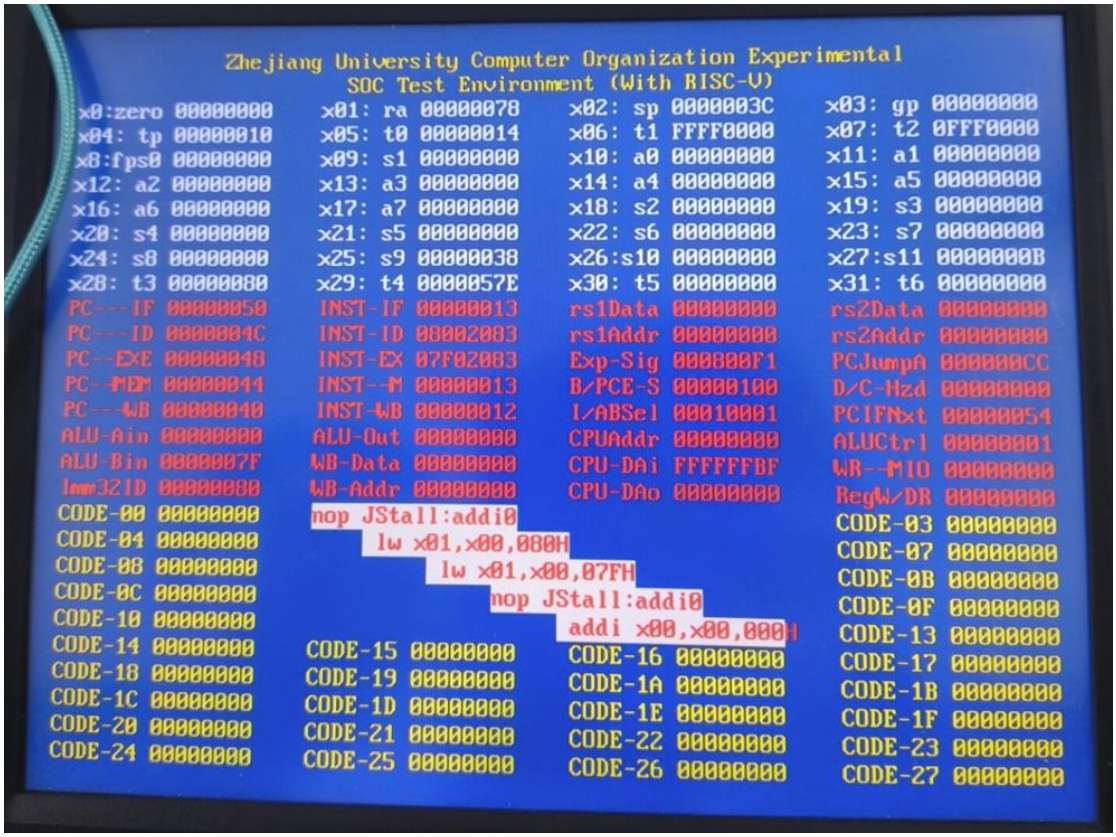
We can see the following instruction caused the trap:

WB 0x08002083, **lw x1, 128(x0)**

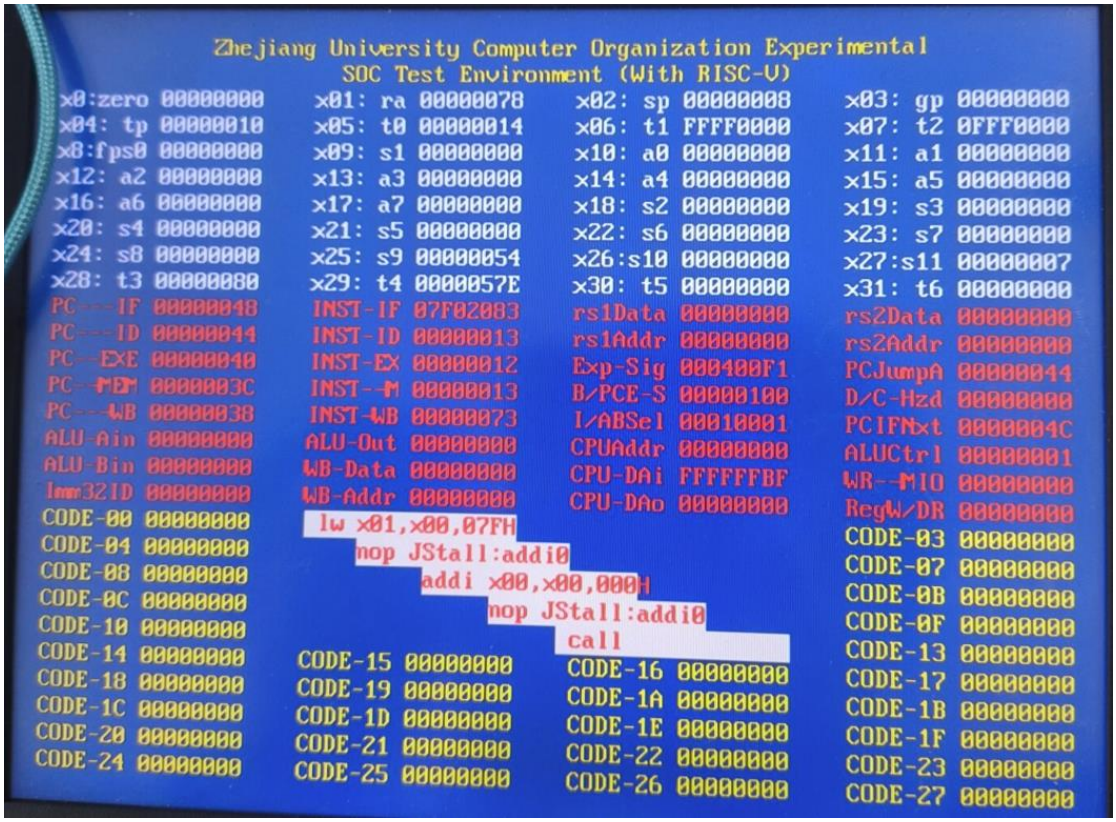
That is a memory fault which will cause a exception. We start automaton transfer at this time, and just as same as how we deal with traps above, we do the same operations in following cycles.

When returned, the return address is 0x60, which was a correct result in theory.

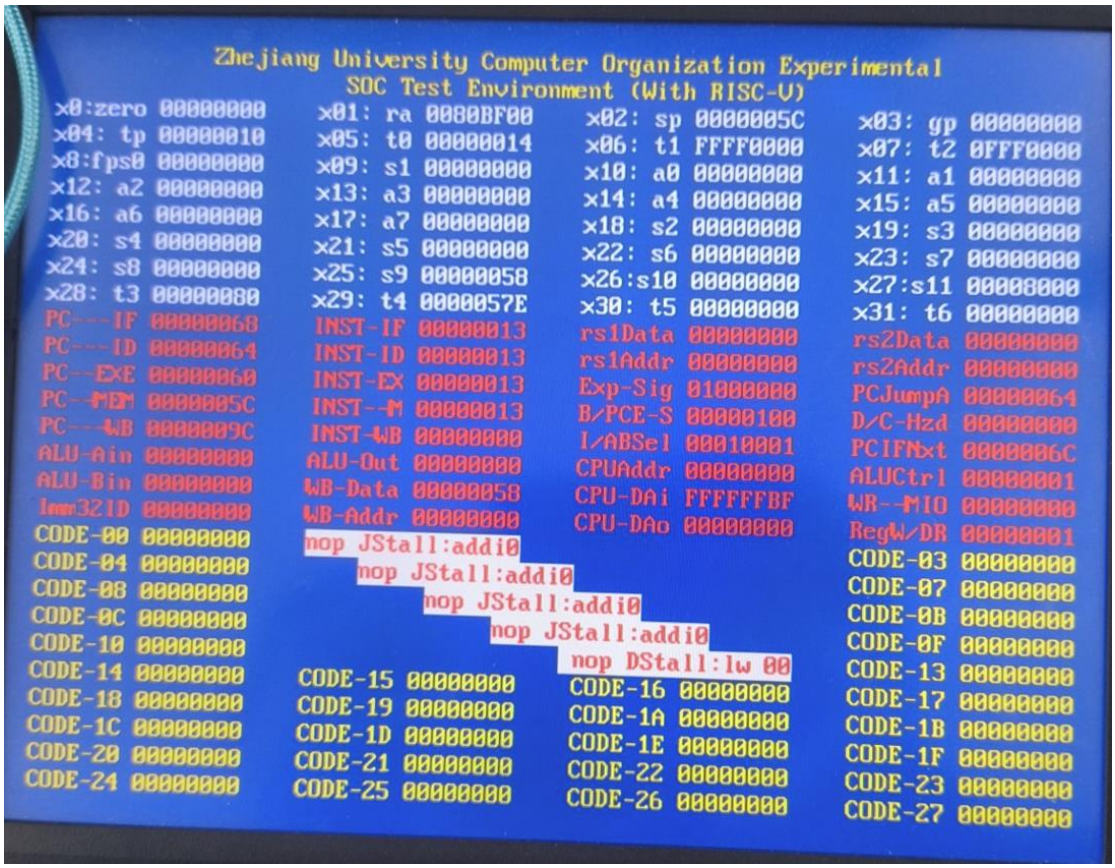
Here are some FPGA running results:



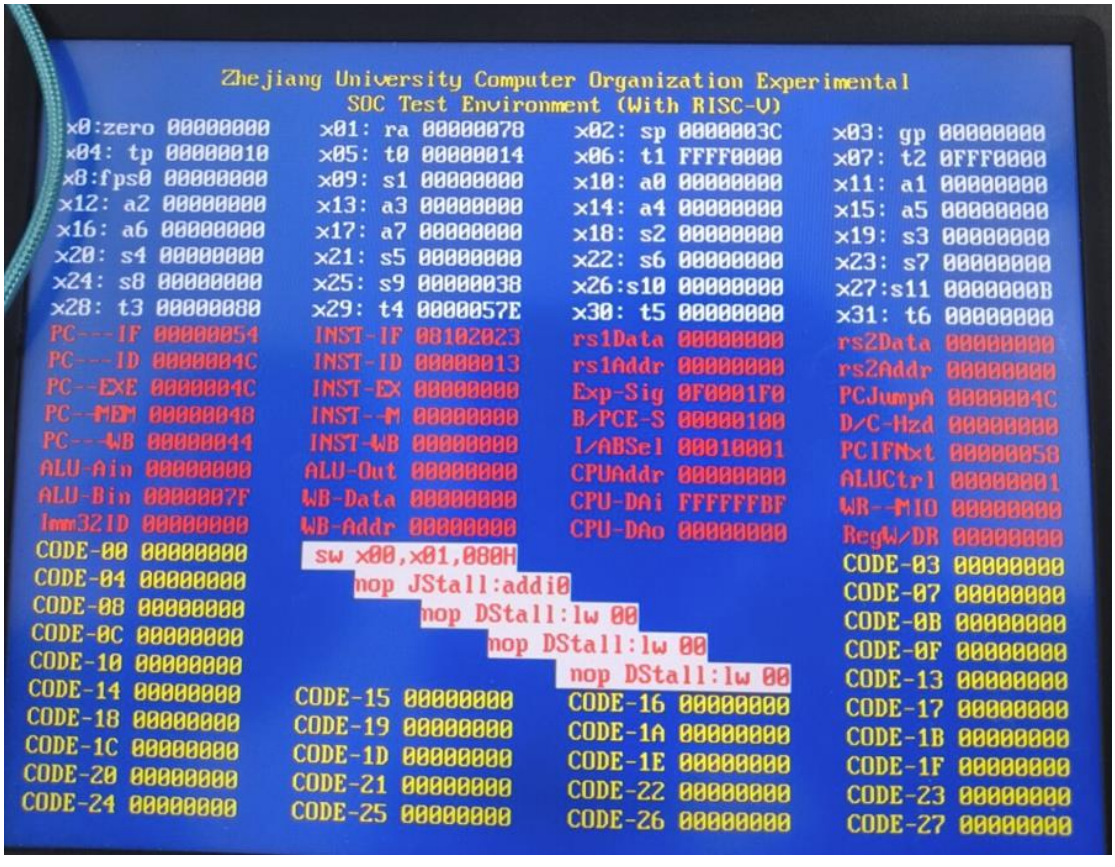
The fault instruction is at MEM.



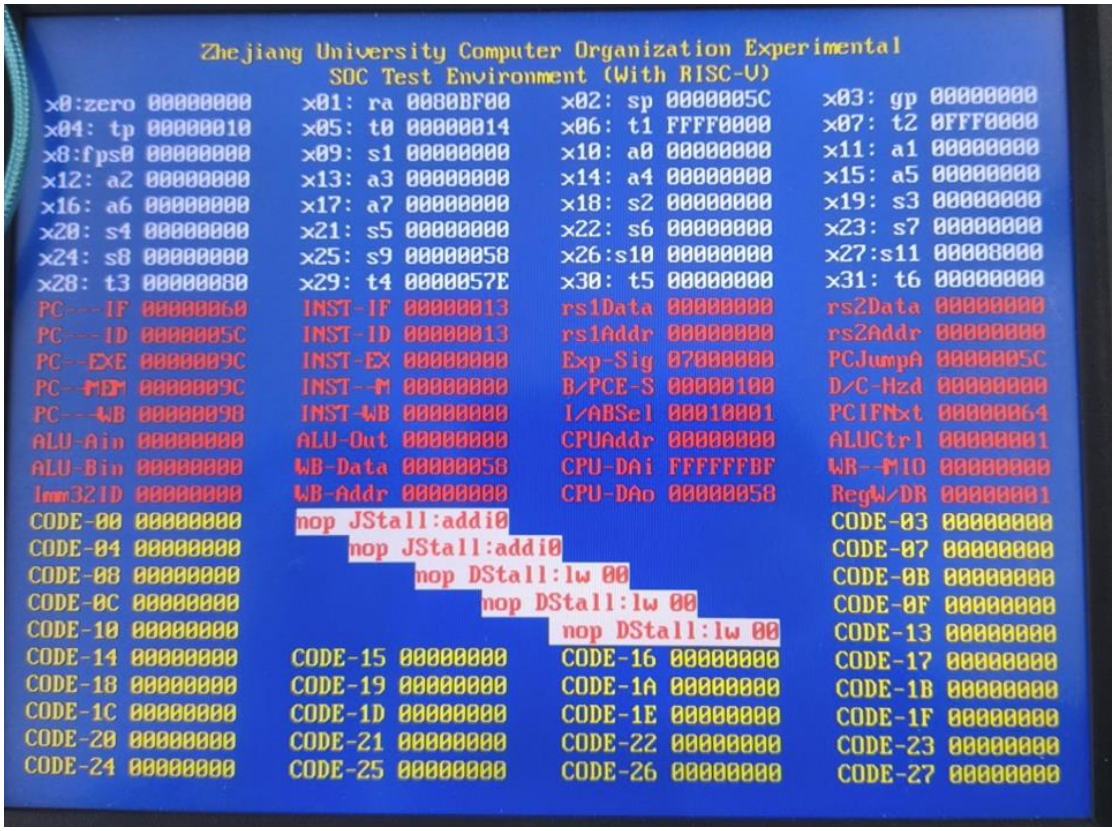
Here **lw x1, 128(x0)** is at WB stage (**STATE_IDLE**).



The program control is returned to User.



At 0x54, the program triggers a save memory fault (**STATE_IDLE**).

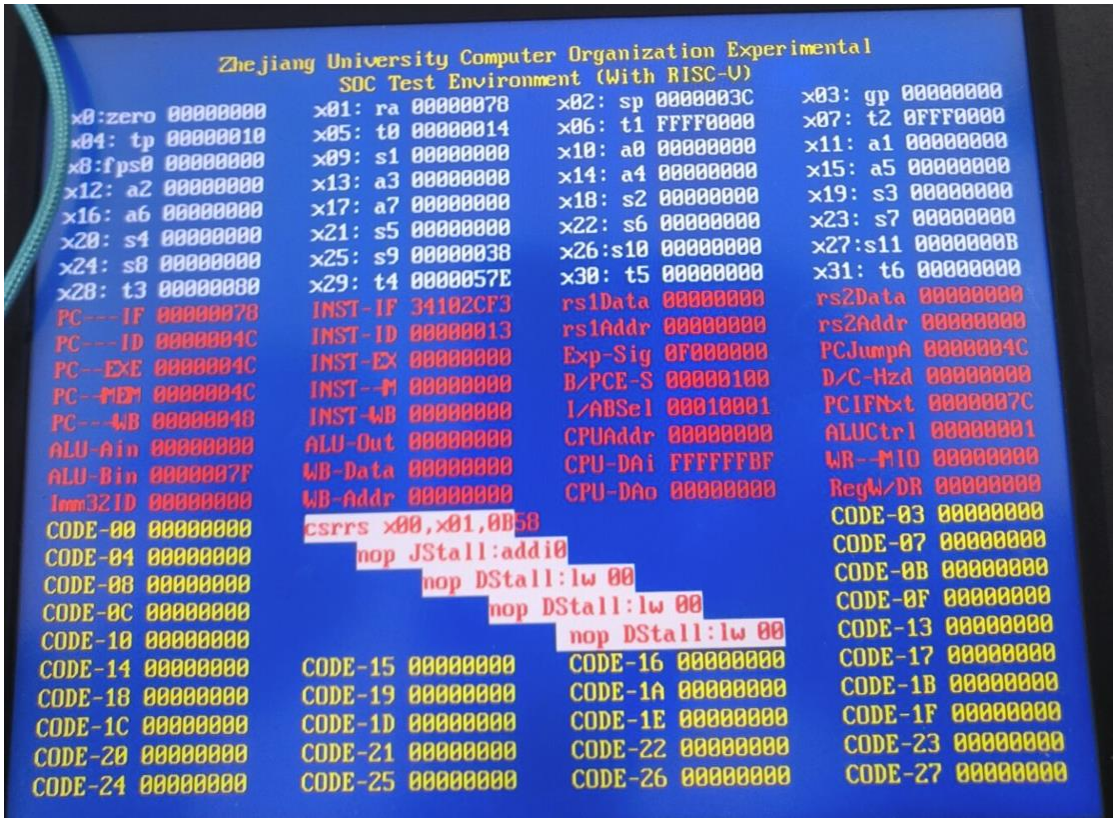


The exception returned to 0x5c(**STATE_MCAUSE**).

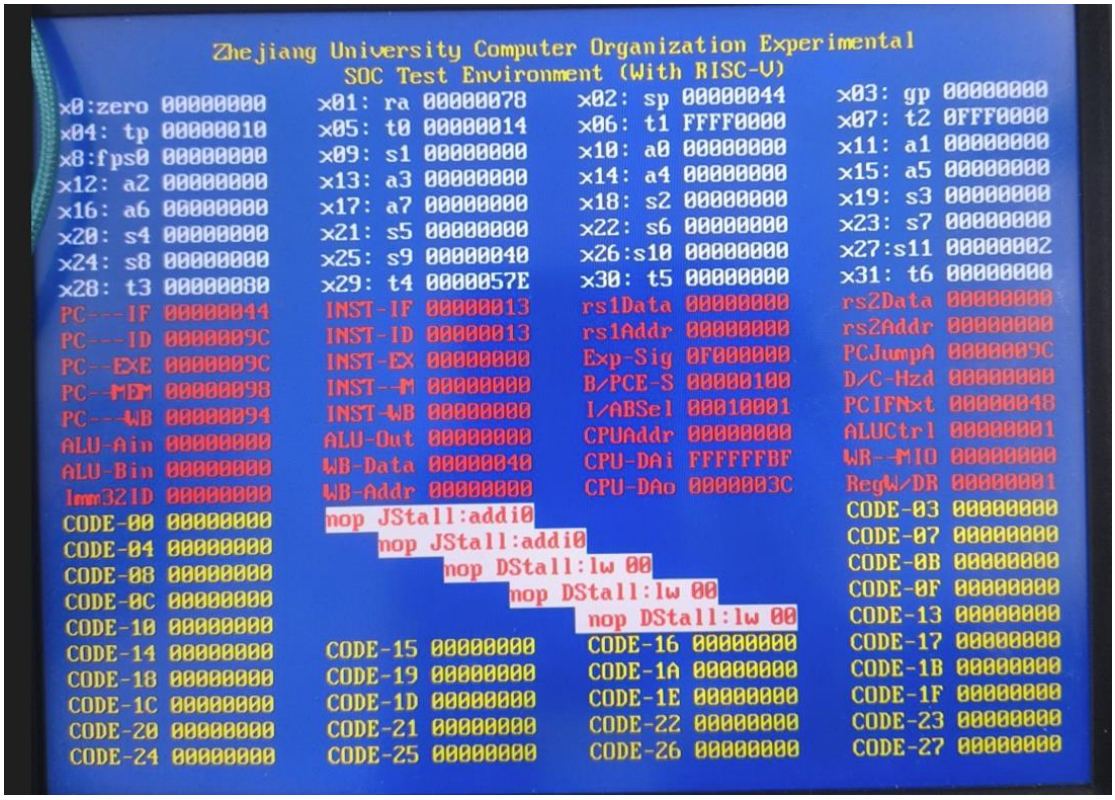
4.5 Hardware Interrupt

As hardware interrupts have not much relation with software, there are no results we can show in behavioral simulation.

Hardware interrupts in our experiment are made by a switch on board. The dealing process are also similar to the traps we mentioned above. So, we will just show the FPGA results.



Entering hardware interrupt at 0x40, several instructions flushed.



Returing to 0x44.

5 Discussion and Conclusion

5.1 Problems

5.1.1 Problems Concerning Verilog Language

Problem 1. always@ Block

As what was generally discussed in the DingTalk group, **always@(posedge clk)**, **always@(negedge clk)** and **always@*** would possibly operate very differently when running on FPGA board.

That's generally because at the positive edge of clock signal the values of many signals are not stable. To solve this problem, logic in **always@*** blocks should be as simple as possible, and the coder should avoid using <= and = operators in the same block.

Also, we could consider using negative edges as trigger to obtain stable signals. I put the <= operators in **always@(posedge clk)** block and = operators in **always@*** block and then solved the problem.

5.1.2 Problems Concerning the Experiment Guides

Problem 1. Detection at WB Stage

Trap detection at WB stage will surely cause some problems. Consider the following codes:

MEM sd x1, 0(x0)

WB ecall

When **ecall** is at WB stage, the MEM stage instruction has been dealt, so the memory could be wrongly set before entering the trap. The experiment guide didn't solve the problem, and I also can hardly think of a method that could solve this on 5-stage pipelined CPU. Maybe we could try to detect the trap at EXE and insert some **bubble** instructions?

Problem 2. Returning Address

In our experiment, the trap handling process will add 4 to **mepc**. That is to avoid we rerun the instruction which will cause an exception. However, software interrupts

are also affected, so if we enter the trap handling process with **mepc** set to next instruction of an **ecall**, the program will return to a wrong address.

We have to overcome the problem by omitting the difference of a software interrupt from an exception.

5.2 Achievements and conclusion

In this experiment, I implemented the trap handling process of a CPU, and learned the basic knowledge about RISC-V CSRs and control levels. The experiment is successful as a result.