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：张沛全　　专业：计算机科学与技术　学号：3190102214

同组学生姓名：陈德瀚　　指导老师：卜凯

实验地点：曹光彪西楼301　实验日期：2021年 11月 1日

1. 实验目的和要求

Purpose

- 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.

Task

- Design of Pipelined CPU supporting exception & interrupt.
    - Design datapath
    - Design Co-processor & Controller
- Verify the Pipelined CPU with program and observe the execution of program

2. 实验内容和原理

Privilege

There are 3 privilege levels in RISC-V: user mode, supervisor mode and machine mode. Machine mode is the most privileged one. When there is some important interrupt or exception, the system should turn to machine mode and handle the situation.

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

CSR

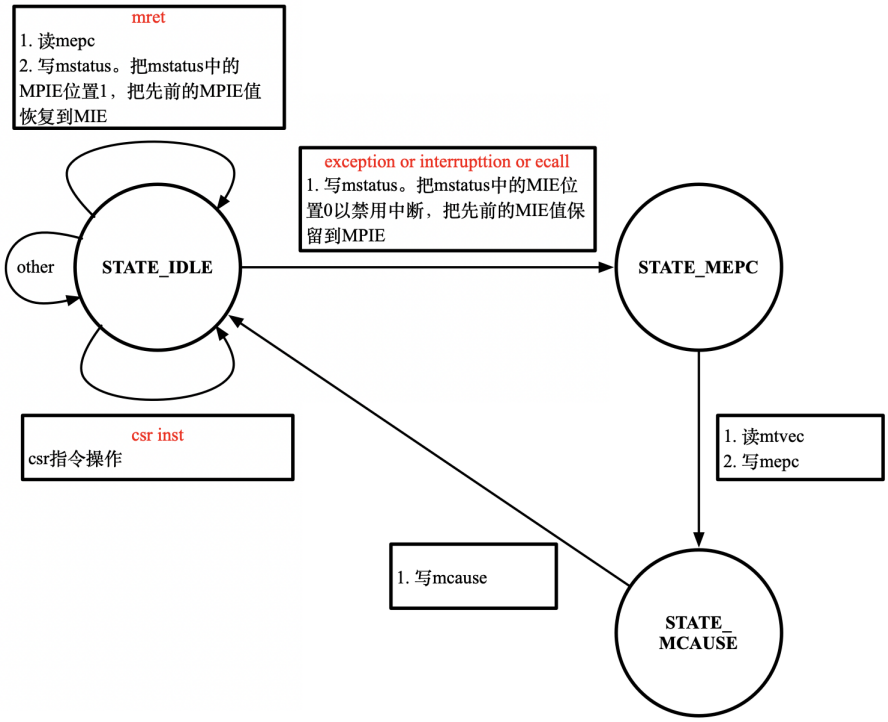
CSR represents Control Status Register, which supports the implementation of different modes. When handling an exception or interrupt, CSR records and controls the status and execution mechanism of CPU. Some important ones are listed as follow:

Name	Full Name	Description
mtvec	Machine Trap-Vector Base-Address Register	定义进入异常的程序PC地址
mcause	Machine Cause Register	反映进入异常的原因
mtval	Machine Trap Value Register	反映进入异常的信息
mepc	Machine Exception Program Counter	用于保存异常的返回地址
mstatus	Machine Status Register	处理器状态，mstatus的MIE域和MPIE域用于反映全局中断使能
mie	Machine Interrupt Enable Registers	用于控制不同类型中断的局部中断使能
mip	Machine Interrupt Pending Registers	反映不同类型中断的等待状态

Some instructions are designed to control CSR, which are explained in the guidance.

Handling

Only 4 kinds of exceptions are required to handle in this experiment, which are intsruction `ecall`, load access fault, store access fault and illegal instruction. Since direct addressing mode is used rather than vectored addressing mode here, interrupt handling is also easy to implement. There is only one trap handler in the program. When entering the trap handler, we can use the state machine as the following diagram has shown:



3. 实验过程和数据记录

There is only one module detecting and handling exception needs to be filled out.

ExceptionUnit.v

At first we figure out the exact usage of each input and output.

After that, we use some sequential logic to implement the state machine. We add a register named `state` to indicate the current state. Then according to the diagram, we simply assign values to control signals and outputs.

An important tip is that the control signals for CSR and outputs should be implemented in combinational logic. If not, all the assignments happen at the positive edge of a clock cycle, the following problems may occur:

- CSR instructions cannot read the proper register out, since the proper address has not been assigned before the positive edge arrives
- (If important output like `PC_redirect` is written in sequential logic) There are 2 clock cycles between the instruction causing exception and the trap, and an instruction will remain within these 2 cycles and be fully executed

- After the trap, CPU could not correctly return to the program, since the proper `mepc` cannot be read

```
`timescale 1ns / 1ps

module ExceptionUnit(
    input clk, rst,
    input csr_rw_in,            // whether a r/w intruction
    input[1:0] csr_wsc_mode_in, // 01: rw  10: rs  11: rc
    input csr_w_imm_mux,        // csr instructions with immediate
    input[11:0] csr_rw_addr_in, // address of the csr
    input[31:0] csr_w_data_reg, // data to write into a csr
    input[4:0] csr_w_data_imm,  // unextended zimm
    output[31:0] csr_r_data_out,// data to write into a register

    input interrupt,
    input illegal_inst,
    input l_access_fault,
    input s_access_fault,
    input ecall_m,

    input mret,

    input[31:0] epc_cur,    // PC of WB
    input[31:0] epc_next,   // next PC that would not be flushed
    output [31:0] PC_redirect,   // PC to fetch in IF
    output redirect_mux,    // select PC for IF

    output reg_FD_flush, reg_DE_flush, reg_EM_flush, reg_MW_flush,
    output RegWrite_cancel
);


    //According to the diagram, design the Exception Unit
    parameter idle = 0;
    parameter mepc = 1;
    parameter mcause = 2;
    reg[1:0] state;
    reg[31:0] _mepc;
    reg[5:0] _cause;
    wire[31:0] mstatus;

    assign exception = illegal_inst | l_access_fault | s_access_fault;
    assign _interrupt = (interrupt & mstatus[3]);
    // PC
    assign PC_redirect = ((state == mepc || mret) && ~rst) ? csr_r_data_out : 0;
    assign redirect_mux = ((state == mepc || mret) && ~rst) ? 1 : 0;


    wire[11:0] csr_raddr =  {12{state == idle && mret}} & 12'h341 | //mepc read
                            {12{state == mepc}} & 12'h305 | // mtvec read
                            {12{state == idle && !mret}} & csr_rw_addr_in;  //
normal
```

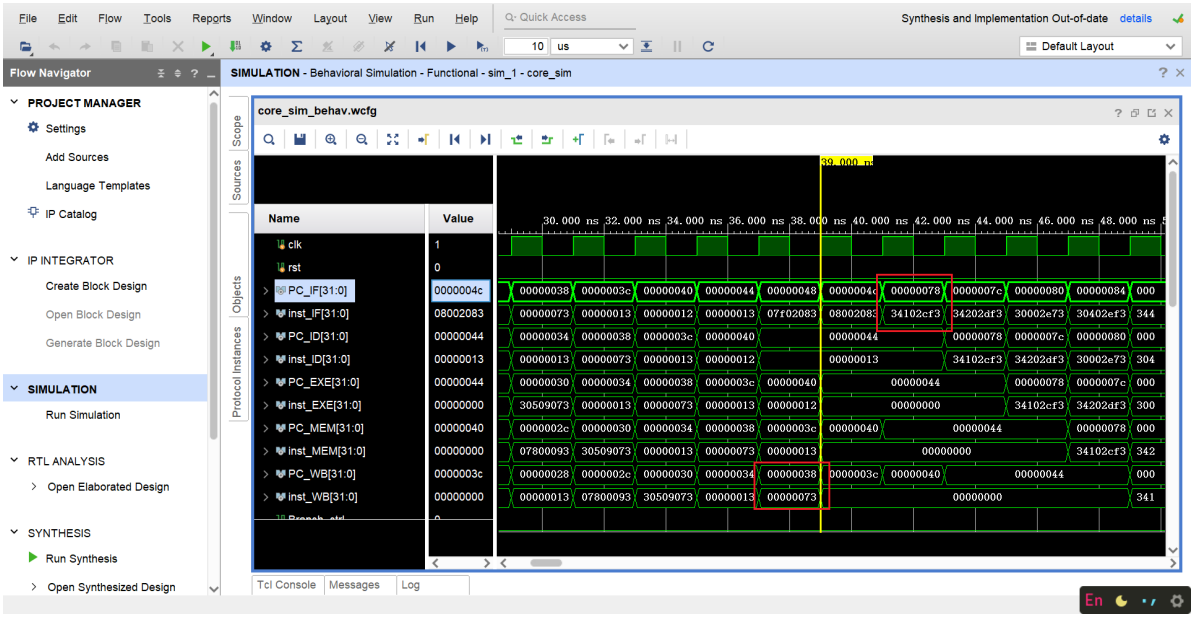
```
    wire[11:0] csr_waddr =  {12{state == idle && (exception | ecall_m | mret)}}
& 12'h300           |    // write mstatus
                           {12{state == mepc}} & 12'h341
                    |      // write mepc
                           {12{state == mcause}} & 12'h342
                    |      // write mcause
                           {12{state == idle && !(exception | ecall_m | mret)}}
& csr_rw_addr_in   ;   // normal

    wire[31:0] csr_wdata =  {32{state == idle && (exception | ecall_m)}} &
{mstatus[31:8], mstatus[3], mstatus[6:4], 1'b0, mstatus[2:0]} |
                           {32{state == idle && (mret)}} & {mstatus[31:8],
1'b1, mstatus[6:4], mstatus[7], mstatus[2:0]}           |
                           {32{state == mepc}} & _mepc |
                           {32{state == mcause}} & _cause |
                           {32{state == idle && !(exception | ecall_m | mret)}}
& (csr_w_imm_mux ? {27'b0, csr_w_data_imm} : csr_w_data_reg)    ;

    wire csr_w = {(state == idle && (exception | ecall_m | mret)) || state ==
mepc || state == mcause}      |
                    {state == idle && !(exception | ecall_m | mret)} & csr_rw_in
& csr_w_data_imm!=5'b00000 ;

    wire[1:0] csr_wsc = {2{state == idle && !(exception | ecall_m | mret)}} &
csr_wsc_mode_in           |
                        {(state == idle && (exception | ecall_m | mret)) ||
state == mepc || state == mcause};


    assign reg_FD_flush = state == idle & (exception | ecall_m | _interrupt) |
state == mepc ? 1 : mret ? 1 : 0;
    assign reg_DE_flush = state == idle & (exception | ecall_m | _interrupt) |
state == mepc ? 1 : mret ? 1 : 0;
    assign reg_EM_flush = state == idle & (exception | ecall_m | _interrupt) |
state == mepc ? 1 : mret ? 1 : 0;
    assign reg_MW_flush = state == idle & (exception | ecall_m) | state == mepc;
    assign RegWrite_cancel = state == idle & (exception | ecall_m) | state ==
mepc;


    always @(posedge clk or posedge rst) begin
        if(rst)begin
            state <= idle;
            _mepc <= 0;
            _cause <= 0;
        end
        else if(state == idle && (exception | ecall_m | _interrupt)) begin
            state <= mepc;
            _mepc <= epc_cur;
            if(ecall_m)
                _cause <= 11;
            else if(illegal_inst)
                _cause <= 2;
            else if (l_access_fault)
                _cause <= 5;
```

```
            else if (s_access_fault)
                _cause <= 7;
            else if (_interrupt)
                _cause <= 32'h80000000;
        end
        else if(state == mepc) begin
            state <= mcause;
            _mepc <= _mepc;
            _cause <= _cause;
        end
        else if(state == mcause) begin
            state <= idle;
            _mepc <= 0;
            _cause <= _cause;
        end
        else if(state == idle && mret) begin
            state <= idle;
            _mepc <= 0;
            _cause <= _cause;
        end
        else begin
            state <= idle;
            _mepc <= 0;
            _cause <= _cause;
        end
    end

    CSRRegs
csr(.clk(clk),.rst(rst),.csr_w(csr_w),.raddr(csr_raddr),.waddr(csr_waddr),

.wdata(csr_wdata),.rdata(csr_r_data_out),.mstatus(mstatus),.csr_wsc_mode(csr_wsc
));


endmodule
```

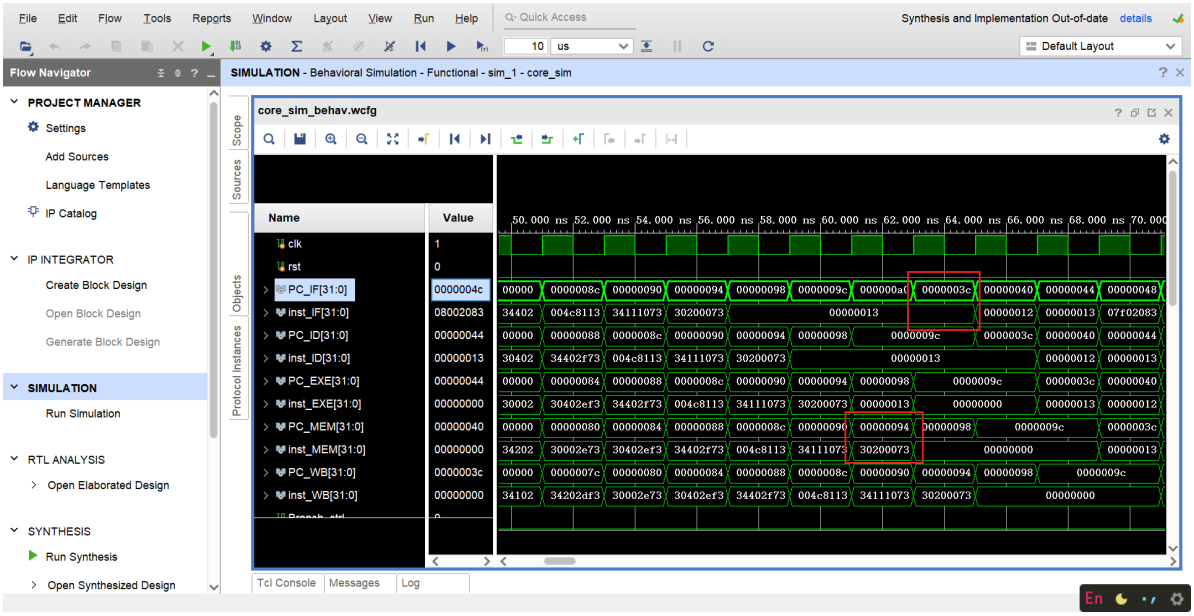
4. 实验结果分析

Simulation

Run the simulation and we can see the result below. When the instruction `ecall` comes to phase WB, CPU enters the trap right after one clock cycle, which is consistent with the guidance. That is correct.



Then, let's check whether CPU works correctly when returning from the trap. When instruction `mret` comes to phase MEM, we can see the instruction right after `ecall` is fetched, which means the program runs correctly. Analysis for other exceptions are similar and so is omitted here.



Program Device

Run the program on a FPGA board and check some points. The following figure shows the interval of one clock cycle between `ecall` at 0x38 and the trap.

Then we can see the first instruciton of the trap is fetched.

After the trap is done, the instruciton right after `ecall` at 0x3C is fetched, and the program continues working normally.

The intruction at 0x54 triggers a store access fault. The following 3 figures shows the process more clearly. The instruction comes to phase WB.

Then, there is an interval. In this interval, CPU is exactly transitioning from state mepc to state mcause, reading `mtvec` and writing `mepc`.

Finally, the first instruction of the trap is fetched. CPU begins to handle the exception.

5. 讨论与心得

The biggest problem is to use sequential logic or combinational logic. Since the refence code includes some register definition, I adopted sequential logic at first. When I was coding, I thought that sequential may cause some problem. Then the simulation result showed that it did. Then I explored the problem with my partner, and combinational logic solves the problem.

The second problem is why simply using conbinational logic works. Since only one CSR register can be written at a time and there are 3 states and 3 transitions before really entering the trap, we thought the interval should be of 2 clock cycles at first. However, the last transition happens in the phase IF of the first instruction of the trap. An instruction cannot write or read a CSR register in phase IF, and this time can be utilized for the last transition, which saves a cycle improving efficiency.

At last, I want to say something about the questions about why there is only one port for writing in the normal register set raised in the groupchat. In short, I think such design is of low cost performance for RISC-V. I think designing multiple writing port is more suitable for CPU or ISA supporting multiple data stream operations. However, there is no SIMD instructions in RISC-V and there is no hardware support either (I think simply adding writing ports involves multiple modifications of other hardware units). If there is no other benefits, adding more writing ports for faster context switching does not seem so economic.