Lab 2: Pipelined CPU Supporting Exceptions & Interrupts

Student ID: 3220106039
Name: 李瀚轩(Hanxuan Li)

1. Overview of the Experiment

In this experiment, we implemented an **Exception Unit** that handles control status register (CSR) instructions, as well as exceptions and interrupts in a pipelined CPU. The goal was to extend the functionality of our pipelined processor by adding support for exception and interrupt handling, ensuring proper execution of CSR instructions, and correctly managing pipeline flushing and program counter (PC) redirection when these events occur.

The primary module implemented in this experiment is the `ExceptionUnit`. The design supports detection of exceptions such as illegal instructions, load/store access faults, system calls, and machine mode interrupts, along with the handling of return from exception (MRET) instructions.

---

2. Steps of the Experiment

2.1. Implementation of the Exception Unit

The `ExceptionUnit` module was designed to manage exceptions and interrupts, handle CSR accesses, and control pipeline flushing. The core components include:

- **CSR Register Access**: The module interfaces with a CSR register file, allowing it to read and write to CSR registers like `mstatus`, `mepc`, `mcause`, and `mtvec`.
- **State Machine**: The design features a state machine that transitions through different states to handle exception processing steps such as storing the exception cause, updating the program counter, and flushing pipeline stages.
- **Pipeline Control**: The unit controls the flushing of various pipeline stages (FD, DE, EM, MW) and cancels writes to the register file when necessary.
- **PC Redirection**: Based on the type of exception or interrupt, the module redirects the PC to either the exception handler address or the saved `mepc` for returning from an exception.

2.1.1 Module Code

```
module ExceptionUnit(
    input clk, rst,
    input csr_rw_in,
    input[1:0] csr_wsc_mode_in,
    input csr_w_imm_mux,
    input[11:0] csr_rw_addr_in,
    input[31:0] csr_w_data_reg,
    input[4:0] csr_w_data_imm,
    output[31:0] csr_r_data_out,

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

    input mret,

    input[31:0] epc_cur,
    input[31:0] epc_next,
    output[31:0] PC_redirect,
    output redirect_mux,

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

The above code initializes the inputs and outputs necessary for the Exception Unit to interact with the CPU pipeline and CSR registers. Notably, it receives signals related to exceptions, interrupts, and the current execution state (`epc_cur` , `epc_next`), and outputs signals to control PC redirection, pipeline flushing, and CSR access.

2.2. State Machine Design

The state machine manages the sequence of operations during exception or interrupt handling. It transitions between the following states:

- **State 0 (IDLE)**: In this state, the system checks for exceptions or interrupts. If an exception occurs, the state machine transitions to State 1. If a return from exception (`mret`) instruction is detected, it handles the restoration of the previous machine state and returns to normal execution.
- **State 1 (MEPC Update)**: The program counter at the time of the exception is saved into the `mepc` register. The state machine then transitions to State 2.
- **State 2 (MCAUSE Update)**: The cause of the exception (stored in the `cause` register) is written to the `mcause` CSR. After this, the state machine transitions back to State 0 (IDLE), where it waits for the next exception or interrupt.

State Transitions

```
always @(posedge clk) begin
        state <= next_state;
        if((state == 0) && ((mstatus[3] && interrupt) | illegal_inst |
l_access_fault | s_access_fault | ecall_m)) begin
            epc <= epc_cur;
            cause <= interrupt ? 1 << 31 :
                     illegal_inst ? 2 :
                     l_access_fault ? 5 :
                     s_access_fault ? 7 :
                     ecall_m ? 11 : 0;
        end
    end
```

In this block, we check if any exception or interrupt is detected. If an exception is found, the current `epc_cur` (the PC value where the exception occurred) is saved in the `epc` register, and the appropriate cause code is recorded in the `cause` register.

2.3. CSR Operations

The unit handles CSR read and write operations based on the `csr_rw_in` signal and the associated control signals (`csr_wsc_mode_in` , `csr_w_imm_mux`). For exceptions, the relevant registers (`mepc` , `mcause` , `mstatus`) are updated in sequence as part of the state machine's transitions.

```
csr_waddr = 12'h300;// mstatus
csr_wdata = {mstatus[31:8], mstatus[3], mstatus[6:4], 1'b0, mstatus[2:0]};
csr_w = 1;// write enable
csr_wsc = 2'b01;// write mode
```

This snippet shows how the `mstatus` register is updated during the handling of an exception. The `mstatus` CSR is responsible for storing and controlling the machine mode's interrupt enable and exception handling.

2.4. Pipeline Flush and PC Redirection

When an exception or interrupt occurs, the pipeline needs to be flushed to prevent erroneous instructions from being executed. The following signals control the flushing of the pipeline stages:

```
FD_flush = 1;
DE_flush = 1;
EM_flush = 1;
MW_flush = 1;
```

At the same time, the PC is redirected to either the exception handler or the return address (`mepc`), depending on the type of exception or interrupt.

```
  reg_PC_redirect = csr_r_data_out;
  reg_redirect_mux = 1;
```

If a `mret` instruction is detected, the PC is redirected to the value stored in `mepc`, allowing the CPU to return to the normal execution flow after handling the exception.

Complete Code:

```
`timescale 1ns / 1ps

module ExceptionUnit(
    input clk, rst,
    input csr_rw_in,
    input[1:0] csr_wsc_mode_in,
    input csr_w_imm_mux,
    input[11:0] csr_rw_addr_in,
    input[31:0] csr_w_data_reg,
    input[4:0] csr_w_data_imm,
    output[31:0] csr_r_data_out,

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

    input mret,

    input[31:0] epc_cur,
    input[31:0] epc_next,
    output[31:0] PC_redirect,
    output redirect_mux,

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

    reg[11:0] csr_raddr, csr_waddr;
    reg[31:0] csr_wdata;
    reg csr_w;
    reg[1:0] csr_wsc;

    wire[31:0] mstatus;

    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_w
sc));

    reg[1:0] state = 0;
    reg[1:0] next_state = 0;
    reg[31:0] cause = 0;
    reg[31:0] epc = 0;
    reg FD_flush = 0;
    reg DE_flush = 0;
    reg EM_flush = 0;
    reg MW_flush = 0;
    reg reg_RegWrite_cancel = 0;
    reg reg_redirect_mux = 0;
    reg[31:0] reg_PC_redirect = 0;

    always @(posedge clk) begin
        state <= next_state;
        if((state == 0) && ((mstatus[3] && interrupt) | illegal_inst |
l_access_fault | s_access_fault | ecall_m)) begin
            epc <= epc_cur;
            cause <= interrupt ? 1 << 31 :
                     illegal_inst ? 2 :
                     l_access_fault ? 5 :
                     s_access_fault ? 7 :
                     ecall_m ? 11 : 0;
        end
    end


    always @(*) begin
        case(state)
            0: begin// STATE_IDLE
                // write mstatus
                if((mstatus[3] && interrupt) | illegal_inst | l_access_fault |
s_access_fault | ecall_m) begin
                    csr_waddr = 12'h300;// mstatus
                    csr_wdata = {mstatus[31:8], mstatus[3], mstatus[6:4],
1'b0, mstatus[2:0]};
                    csr_w = 1;// write enable
                    csr_wsc = 2'b01;// write mode
                // flush all the pipeline registers
                    FD_flush = 1;
                    DE_flush = 1;
                    EM_flush = 1;
                    MW_flush = 1;
                // record epc and cause
                    // epc = epc_cur;
                    // cause = interrupt ? 1 << 31 :
                    //          illegal_inst ? 2 :
                    //          l_access_fault ? 5 :
                    //          s_access_fault ? 7 :
                    //          ecall_m ? 11 : 0;
                // if exception (not interrupt), cancel regwrite
                    reg_RegWrite_cancel = interrupt ? 0 : 1;
                    next_state = 1;// to STATE_MEPC
                    csr_raddr = 0;
                    reg_redirect_mux = 0;
                    reg_PC_redirect = 0;
                end
                else if(mret) begin
                    //write mstatus
                    csr_waddr = 12'h300;// mstatus
                    csr_wdata = {mstatus[31:8], 1'b1, mstatus[6:4],
mstatus[7], mstatus[2:0]};
                    csr_w = 1;// write enable
                    csr_wsc = 2'b01;// write mode
                    //read mepc
                    csr_raddr = 12'h341;// mepc
                    //set redirect pc mux (next cycle pc �? mepc)
                    reg_redirect_mux = 1;
                    reg_PC_redirect = csr_r_data_out;
                    //flush pipeline registers (EM, DE, FD)
                    FD_flush = 1;
                    DE_flush = 1;
                    EM_flush = 1;
                    MW_flush = 0;
                    // epc = 0;
                    // cause = 0;
                    reg_RegWrite_cancel = 0;
                    next_state = 0;
                end
                else if(csr_rw_in) begin
                    //csr operations
                    csr_raddr = csr_rw_addr_in;

                    csr_waddr = csr_rw_addr_in;
                    if (csr_w_imm_mux) begin
                        // write immediate number
                        csr_wdata = {27'b0, csr_w_data_imm};
                    end
                    else begin
                        // write reg data
                        csr_wdata = csr_w_data_reg;
                    end
                    csr_w = csr_wsc_mode_in == 1 ? 1 : 0;
                    csr_wsc = csr_wsc_mode_in;
                    FD_flush = 0;
                    DE_flush = 0;
                    EM_flush = 0;
                    MW_flush = 0;
                    // epc = 0;
                    // cause = 0;
                    reg_RegWrite_cancel = 0;
                    next_state = 0;
                    reg_redirect_mux = 0;
                    reg_PC_redirect = 0;
                end
                else begin
                    csr_waddr = 0;
                    csr_wdata = 0;
                    csr_w = 0;
                    FD_flush = 0;
                    DE_flush = 0;
                EM_flush = 0;
                MW_flush = 0;
                reg_RegWrite_cancel = 0;
                reg_redirect_mux = 0;
                reg_PC_redirect = 0;
                // cause = 0;
                // epc = 0;
                next_state = 0;
                csr_raddr = 0;
            end
        end
        1: begin// STATE_MEPC
            //write epc tp mepc
            csr_waddr = 12'h341;// mepc
            csr_wdata = epc;
            csr_w = 1;
            csr_wsc = 2'b01;
            //read mtvec
            csr_raddr = 12'h305;// mtvec
            //flush pipeline registers (FD)
            FD_flush = 1;
            DE_flush = 0;
            EM_flush = 0;
            MW_flush = 0;
            //set redirect pc mux (next cycle pc �? mtvec)
            reg_PC_redirect = csr_r_data_out;
            reg_redirect_mux = 1;
            next_state = 2;
            // epc = 0;
            // cause = 0;
            reg_RegWrite_cancel = 0;
        end
        2: begin// STATE_MCAUSE
            //write cause to mcause
            reg_redirect_mux = 0;
            csr_waddr = 12'h342;// mcause
            csr_wdata = cause;
            csr_w = 1;
            csr_wsc = 2'b01;
            next_state = 0;
            FD_flush = 0;
            DE_flush = 0;
            EM_flush = 0;
            MW_flush = 0;
            reg_RegWrite_cancel = 0;
            reg_PC_redirect = 0;
            // epc = 0;
            // cause = 0;
            csr_raddr = 0;
        end
    endcase
end

assign reg_FD_flush = FD_flush;
```

```
    assign reg_DE_flush = DE_flush;
    assign reg_EM_flush = EM_flush;
    assign reg_MW_flush = MW_flush;
    assign RegWrite_cancel = reg_RegWrite_cancel;
    assign redirect_mux = reg_redirect_mux;
    assign PC_redirect = reg_PC_redirect;
endmodule
```

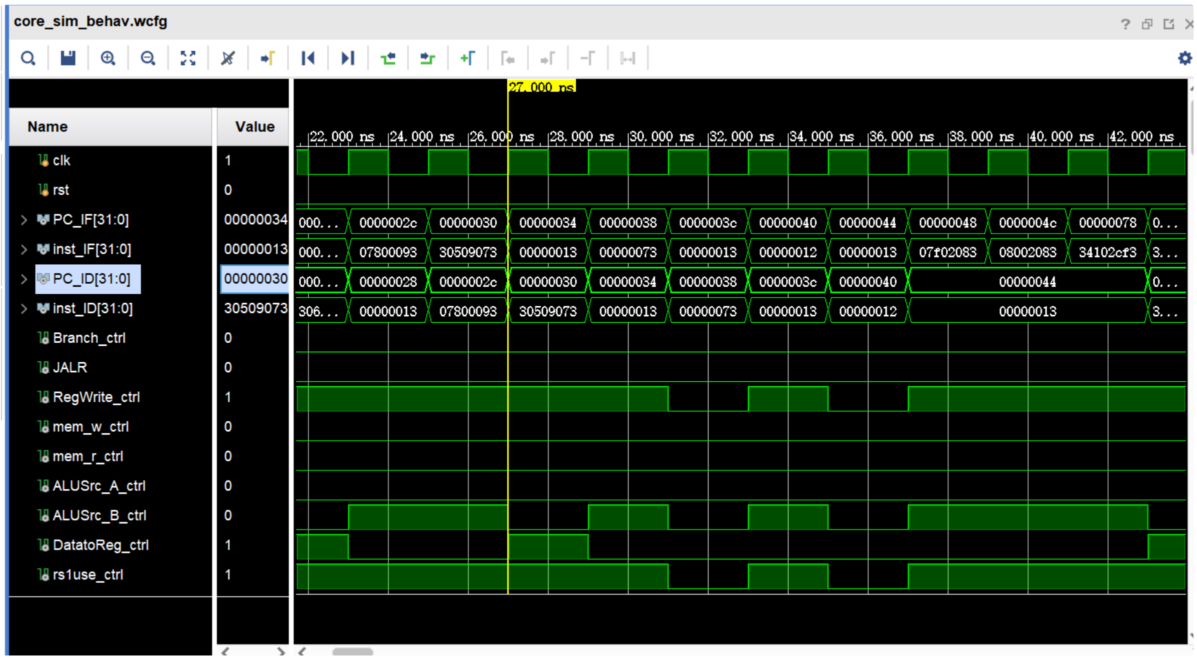
2.5 CtrlUnit

```
assign rs1use = CSR_valid | R_valid | I_valid | B_valid | L_valid |
S_valid | JALR;
assign hazard_optype = (CSR_valid | R_valid | I_valid | JAL | JALR |
LUI | AUIPC) ? 2'b01 :
L_valid ? 2'b10 :
S_valid ? 2'b11 : 2'b00;
```

Only the two signals mentioned above may involve `rs1` forwarding due to CSR instructions. Therefore, we need to add a CSR_valid signal to ensure that CSR instructions can also be forwarded correctly. Other parts are the same as that in lab1.

2. Experimental Evaluation

2.1 Simulation


core_sim_behav.wcfg

Name	Value
clk	1
rst	0
PC_IF[31:0]	00000034
inst_IF[31:0]	00000013
PC_ID[31:0]	00000030
inst_ID[31:0]	30509073
Branch_ctrl	0
JALR	0
RegWrite_ctrl	1
mem_w_ctrl	0
mem_r_ctrl	0
ALUSrc_A_ctrl	0
ALUSrc_B_ctrl	0
DatatoReg_ctrl	1
rs1use_ctrl	1

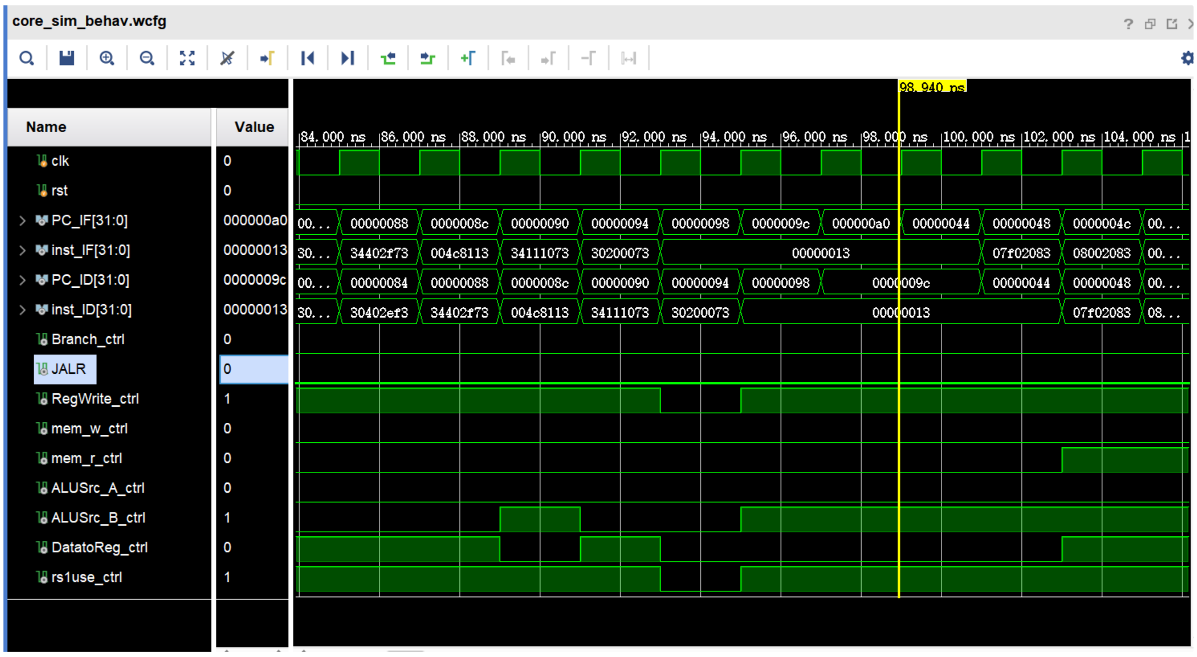
First, regarding the read and write operations for privileged instructions, we observe the following:

- When the address is `0x18`, a value of `0x10` is successfully written to the `mtvec` register at address `0x306`.

- When the address is `0x20`, a value of `0xffff0000` (the value stored in register `x6`) is written to the same `0x306` register. Both write and read operations correctly reflect the expected values.

Next, when the address is `0x30`, the `mtvec` register is updated to `0x78`.

Finally, when the instruction at address `0x38`, which is an `ecall` instruction, is executed, the processor enters exception handling mode. The instruction causes a jump to the handler address `0x78`, the `mepc` register is updated to `0x38` (the address of the `ecall` instruction), and the `mcause` register holds the value `0xb`, indicating the cause of the exception. All these behaviors are consistent with expected results.



During the exception handling process, we observe that reading from the `mepc`, `mcause`, `mstatus`, `mie`, and `mip` registers works as expected. The `mepc + 4` operation functions correctly, and the `mret` instruction properly returns from the exception.

Other types of exceptions are handled correctly as well, following the same procedure. Therefore, they are not demonstrated individually in this discussion.

2.2 Onboard Result

This behavior was already demonstrated to the teacher during the lab session, so there is no need to elaborate further.