浙江大学

计算机体系结构实验报告

课程名称：	计算机体系结构
姓　　名：	周楠
学　　院：	计算机科学与技术学院
专　　业：	计算机科学与技术
学　　号：	3220102535
指导教师：	常瑞

2024 年 10 月 21 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：	流水线异常和中断设计				
学生姓名：	周楠	专业	计算机科学与技术	学号：	3220102535
实验地点：	玉泉实验室	实验时间：	2024.10.22	指导教师：	常瑞

一. 操作方法与实验步骤

1.1 CSRRegs 模块

该模块用于实现控制状态寄存器（CSR, Control and Status Registers）的读写和操作功能。

具体来说，该模块通过输入的地址和数据，控制 CSR 的读取和写入操作，并在异常（trap）和返回（mret）操作时更新特定的寄存器内容。

```
module CSRRegs(
    input clk, rst,
    input[11:0] raddr, waddr,
    input[31:0] wdata,
    input csr_w,
    input[1:0] csr_wsc_mode,
    output[31:0] rdata,
    // 补充信号
    input trap,
    input mret,
    input[31:0] mstatus_in,
    input[31:0] mtvec_in,
    input[31:0] mepc_in,
    input[31:0] mcause_in,
    input[31:0] mtval_in,
    output[31:0] mstatus_out,
    output[31:0] mtvec_out,
    output[31:0] mepc_out,
```

```
19      output[31:0] mcause_out,
20      output[31:0] mtval_out
21  );
22      // You may need to modify this module for better efficiency
23
24      reg[31:0] CSR [0:15];
25
26      // Address mapping. The address is 12 bits, but only 4 bits are
            used in this module.
27      wire raddr_valid = raddr[11:7] == 5'h6 && raddr[5:3] == 3'h0;
28      wire[3:0] raddr_map = (raddr[6] << 3) + raddr[2:0];
29      wire waddr_valid = waddr[11:7] == 5'h6 && waddr[5:3] == 3'h0;
30      wire[3:0] waddr_map = (waddr[6] << 3) + waddr[2:0];
31
32      // 根据 raddr_map 计算得到 mepc, mtvec, mcause, mtval, mstatus对应
            的CSR索引
33      assign mstatus_out = CSR[0];
34      assign mtvec_out = CSR[5];
35      assign mepc_out = CSR[9];
36      assign mcause_out = CSR[10];
37      assign mtval_out = CSR[11];
38
39
40      assign rdata = CSR[raddr_map];
41
42      always@(posedge clk or posedge rst) begin
43          if(rst) begin
44              CSR[0] <= 32'h88;
45              CSR[1] <= 0;
46              CSR[2] <= 0;
47              CSR[3] <= 0;
48              CSR[4] <= 32'hfff;
49              CSR[5] <= 0;
50              CSR[6] <= 0;
51              CSR[7] <= 0;
```

```
52              CSR[8] <= 0;
53              CSR[9] <= 0;
54              CSR[10] <= 0;
55              CSR[11] <= 0;
56              CSR[12] <= 0;
57              CSR[13] <= 0;
58              CSR[14] <= 0;
59              CSR[15] <= 0;
60          end
61          else if(trap) begin
62              CSR[0] <= mstatus_in;
63              CSR[5] <= mtvec_in;
64              CSR[9] <= mepc_in;
65              CSR[10] <= mcause_in;
66              CSR[11] <= mtval_in;
67          end
68          else if(mret) begin
69              CSR[0] <= mstatus_in;
70          end
71          else if(csr_w & !trap & !mret) begin
72              // csr_wsc_mode = inst[13:12]
73              // 当 csr_wsc_mode = 01 时, csr_wdata = wdata
74              // 当 csr_wsc_mode = 10 时, csr_wdata = wdata | wdata
75              // 当 csr_wsc_mode = 11 时, csr_wdata = wdata & ~wdata
76              case(csr_wsc_mode)
77                  2'b01: CSR[waddr_map] = wdata;
78                  2'b10: CSR[waddr_map] = CSR[waddr_map] | wdata;
79                  2'b11: CSR[waddr_map] = CSR[waddr_map] & ~wdata;
80                  default: CSR[waddr_map] = wdata;
81              endcase
82          end
83      end
84  endmodule
```

1. 异常处理机制

   此处将 trap 和 mret 两个信号作为控制信号，由于两种情况下修改的 CSR 寄存器不同，因此将其分开处理

   - 当发生异常（trap 信号有效）时，模块会将输入的 mstatus_in、mtvec_in、mepc_in、mcause_in 和 mtval_in 等寄存器值写入对应的 CSR 寄存器。
   - 当处理器从异常返回（mret 信号有效）时，更新 mstatus 寄存器的值。

2. CSR 操作模式

   根据 csr_wsc_mode 信号，该模块支持不同的写操作模式：

   - 01：直接写入 wdata。
   - 10：按位“或”写入，CSR[waddr_map] = CSR[waddr_map] | wdata。
   - 11：按位“与非”写入，CSR[waddr_map] = CSR[waddr_map] & wdata。

1.2 ExceptionUnit 模块

1.2.1 exception 与 trap 信号的设置

```
1    wire trap;
2    wire exception;
3    assign exception = (illegal_inst | l_access_fault |
        s_access_fault | ecall_m);
4    assign trap = (exception | interrupt) & mie;
```

由于 exception 和 interrupt 在异常处理结束后返回的 PC 地址不同，因此将其分开处理。当出现非法指令异常 (illegal_inst), 访问错误异常（l_access_fault 和 s_access_fault）, 系统调用异常 (ecall_m) 时，我们设置 exception 信号为 1

trap 信号还需要额外考虑中断使能信号 mie，当 mie 为 1 时，trap 信号有效，否则无效。

1.2.2 CSR 寄存器的读写

```
1    assign csr_raddr = csr_rw_addr_in;
2    assign csr_waddr = csr_rw_addr_in;
3    assign csr_w = csr_rw_in;
4    assign csr_wsc = csr_wsc_mode_in;
5    // 修改1
6    assign csr_wdata = csr_w_imm_mux ? {27'b0, csr_w_data_imm} :
        csr_w_data_reg;
7    assign csr_r_data_out = csr_rdata;
8
9    // mepc_in 仅仅用于trap发生时，要么exception，要么interrupt
10   assign mepc_in = exception ? epc_cur : epc_next;
11   // 当trap发生时，PC跳转到mtvec，mret返回时，PC跳转到mepc
12   assign PC_redirect = (trap) ? mtvec : mepc;
13   // 根据 trap 来决定是否发生 redirect_mux
14   assign redirect_mux = trap | mret;
15   assign mtval_in = (illegal_inst) ? inst_WB :
16                  (l_access_fault | s_access_fault) ? addr_WB : 32'
                       b0;
17
18
19   assign mcause_in = (illegal_inst) ? 32'h00000002 :
20                     (l_access_fault) ? 32'h00000005 :
21                     (s_access_fault) ? 32'h00000007 :
22                     (ecall_m) ? 32'h0000000b :
23                     (interrupt) ? 32'h8000000b : 32'b0;
24   assign mtvec_in = mtvec;
```

1. mepc_in 的设置

   - 当 trap 发生时，需要保存指定的 PC 值到 mepc_in 中。对于 exception，mepc 指向导致异常的指令；对于 interrupt，它指向中断处理后应该恢复执行的位置。也就是说，如果 exception 信号为 1，mepc_in = epc_cur，如果 interrupt 信号为 1，mepc_in = epc_next。
   - 在设置完 mepc_in 之后，我们需要完成指令的跳转，对于 trap 的情况，跳转到 mtvec，对于 mret 的情况，跳转到 mepc。
   - redirect_mux 信号用于决定是否要发生 PC 的跳转。

2. mtval_in 的设置

   - 访问错误异常时，写入错误的地址。
   - 非法指令异常时，写入错误的指令。

   需要在 ExceptionUnit 模块中额外添加所需的信号，inst_WB 表示当前的非法指令，addr_WB 表示非法地址。

3. mcause_in 的设置根据异常类型的不同，设置不同的 mcause_in。

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved for future standard use*
1	11	Machine external interrupt
1	12–15	*Reserved for future standard use*
1	≥16	*Reserved 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 for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

Table 3.6: Machine cause register (`mcause`) values after trap.

图 1: mcause_in 的设置

4. mtvec_in 的设置 mtvec_in 的值为指定的地址，不需要发生改变

1.2.3 状态管理和恢复

```
1     wire mie;
2     assign mie = mstatus[3];
3     reg[1:0] MPP;
4     reg MPIE;
5     always @(posedge clk or posedge rst) begin
6         if (rst) begin
7             MPP <= 2'b11;
8             MPIE <= 1'b1;
9         end
10        else if (mret) begin
11            MPP <= mstatus[12:11];
12            MPIE <= mstatus[3];
13        end
14        else if (trap) begin
15            MPP <= 2'b11;
16            MPIE <= 1'b1;
17        end
18        else begin
19            MPP <= MPP;
20            MPIE <= MPIE;
21        end
22    end
23    assign mstatus_in = (mret) ? {mstatus[31:4], MPIE, mstatus[2:0]}
          : {mstatus[31:13], 2'b11, mstatus[10:8], mstatus[3], mstatus
          [6:4], 1'b0, mstatus[2:0]} ;
```

- 异常或中断发生时，mstatus 寄存器的 MPP 和 MPIE 位会被保存，并且相应地设置为新值。
- 在执行 mret 指令时，处理器恢复先前保存的 MPP 和 MPIE 状态，从而继续正常的程序执行。
- 对于 mstatus, 当 mret 发生时，此时的 mie 要用 mpie 替换，表示此时能够发生中断。否则当 trap 发生时，先记录当前的特权模式 MPP，也就是 M mode，

然后设置 mpie 为 mie，保存进入 trap 处理程序前的 mie 状态，让 mie=0，表示进入 trap 处理程序后无法再次发生中断。

1.2.4 流水线清除与取消

```
1    // 接下来处理 reg_FD_flush，reg_DE_flush，reg_EM_flush，
       reg_MW_flush
2    // 当trap发生时，需要将后面的指令全部取消
3    // 当mret发生时，需要将IF，ID，EXE阶段的指令全部取消，MEM阶段的指令
       保存
4    assign reg_FD_flush = trap | mret;
5    assign reg_DE_flush = trap | mret;
6    assign reg_EM_flush = trap | mret;
7    assign reg_MW_flush = trap;
8
9    // 接下来处理 RegWrite_cancel，MemWrite_cancel
10   assign RegWrite_cancel = exception;
11   assign MemWrite_cancel = trap;
```

根据 trap 和 mret 设置 flush 以及 cancle 信号：

1. exception 从 WB 阶段传递过来，对应的是非法指令和访问非法地址，因此此时不能执行 RegWrite 操作，并且 WB 阶段的指令全部取消。因此 Reg_MW_flush = 1，RegWrite_cancel = 1
2. 发生 trap 时，MEM 阶段的指令不能写入内存，因为返回的时候仍需执行该条指令，会导致重复错误。
3. 对于 trap 发生时，需要停止正在执行的指令，因此 reg_FD_flush = 1, reg_DE_flush = 1, reg_EM_flush = 1, reg_MW_flush = 1
4. 对于 mret 情况，mret 信号在 MEM 阶段发出，因此执行 mret 时不能 flush MEM 阶段

二. 思考题

1. 精确异常和非精确异常的区别是什么？

   精确异常意味着在异常发生时，处理器的状态（如寄存器、内存等）完全一致，程序可以被“正确”地中断并进入异常处理程序。处理器能够保证所有在异常发生之前的指令都已经完整执行，而所有在异常之后的指令都没有开始执行。

   非精确异常是指当异常发生时，处理器无法保证所有之前的指令都已经执行完毕，或者后续的指令可能已经部分执行。

   - 指令执行顺序: 精确异常确保异常发生前的指令已经执行，之后的指令未执行；非精确异常无法确保异常点的指令完全执行顺序
   - 状态一致性：精确异常发生时系统状态可完全恢复；非精确异常状态可能不一致，恢复复杂。
   - 异常处理难度：精确异常异常点清晰，易于恢复和调试；非精确异常异常点不明确，调式和恢复复杂

2. 阅读测试代码，第一次导致 trap 的指令是哪条？trap 之后的指令做了什么？如果实现了 U mode，并以 U mode 从头开始执行测试指令，会出现什么新的异常？

   第一次 trap 为 ecall 指令

   trap 发生后，处理器将会进入异常处理程序。读取 CSR 寄存器到寄存器堆中，更新 mepc，执行 mret。

```
1        34102cf3 csrr x25, 0x341 # mepc PC = 0x78
2        34202df3 csrr x27, 0x342 # mcause PC = 0x7c
3        30002e73 csrr x28, 0x300 # mstatus PC = 0x80
4        34302ef3 csrr x29, 0x343 # mtval PC = 0x84
5        34402f73 csrr x30, 0x344 # mip PC = 0x88
6        004c8113 addi x2, x25, 4 PC = 0x8c
7        34111073 csrw 0x341, x2 PC = 0x90
8        30200073 mret PC = 0x94
9        00000013 addi x0, x0, 0 PC = 0x98
10       00000013 addi x0, x0, 0 PC = 0x9c
11       00000013 addi x0, x0, 0 PC = 0xa0
12       00000013 addi x0, x0, 0 PC = 0xa4
```

在 U mode 下，用户态程序尝试访问某些不允许用户态读取或写入的 CSR 寄存器时（例如 mstatus、mepc 等机器模式特权寄存器），处理器会认为该操作是非法操作，触发 Illegal Instruction Exception（非法指令异常）

在 U mode 下执行 ecall 指令，会触发环境调用异常，其 mcause 将被设置为 8，表示 ecall 来自用户模式。

3. 为什么异常要传到最后一段即 WB 段后，才送入异常处理模块？可不可以一旦在某一段流水线发现了异常就送入异常处理模块，如果可以请说明异常处理模块应该如何处理异常；如果不可以，请说明理由。

   异常需要在 WB 阶段后处理的原因：

   (a) 保证异常的精确性: 存储访问错误或算术溢出等异常可能会在 EX（执行）阶段或 MEM（内存访问）阶段被发现，但在确定结果写回寄存器前，流水线尚未完全处理该指令的全部影响。如果在某个较早的阶段就直接进行异常处理，会导致前面的指令有可能已经部分执行，而后面的指令状态还未确定，影响到整个流水线的准确性。

   (b) 避免错误的指令取消: 如果在较早的阶段（如 ID 或 EX）发现异常并立刻触发异常处理机制，那么后续阶段可能已经开始执行的指令会被误取消，这会影响到程序的正确执行。等到 WB 阶段后再触发异常处理，确保了前面所有的依赖已经被正确解决。

   如何实现在早期阶段处理异常：

   (a) 精确异常恢复机制：在早期阶段捕捉到异常时，处理器必须具备恢复未完成指令的机制，即从异常指令开始，正确恢复所有寄存器、内存状态。这要求在流水线的每个阶段都维护寄存器和内存的回滚信息，确保异常处理完成后，可以精确恢复到异常发生之前的状态。