Lab 2：流水线异常和中断设计

陈祎伟

3230102357

2025年3月9日

1 设计思路

1.1 CSRRegs.v文件

CSRRegs 模块用于实现 CSR 寄存器的读写操作。

在原先 CSR 寄存器读写指令实现的基础上，增加了对于 mepc、mcause、mtval、mstatus 的旁路写操作。

增加相关wire信号输入后，具体实现如下：

```
    else if (bypass_EN) begin
        CSR[0] <= mstatus_bypass_in;
        CSR[9] <= mepc_bypass_in;
        CSR[10] <= mcause_bypass_in;
        CSR[11] <= mtval_bypass_in;
    end
```

其中，mstatus_bypass_in、mepc_bypass_in、mcause_bypass_in、mtval_bypass_in 分别为 mstatus、mepc、mcause、mtval 的旁路写入信号，而 bypass_EN 为旁路写入使能信号。

1.2 ExceptionUnit.v文件

ExceptionUnit 模块用于处理异常和中断。

首先连接 CSRRegs 模块，实现读取与写入：

```
    assign csr_raddr = csr_rw_addr_in;
    assign csr_r_data_out = csr_rdata;
```

```
    assign csr_waddr = csr_rw_addr_in;
    assign csr_wdata = csr_w_imm_mux ? csr_w_data_imm : csr_w_data_reg;
    assign csr_w = csr_rw_in;
    assign csr_wsc = csr_wsc_mode_in;
```

定义寄存器信号 in_trap 用于指示是否处于异常或中断状态。

当发生异常或中断时，将 bypass_EN 置为 1，并将相关信息通过旁路传递给 CSRRegs 模块，具体实现如下：

```
    if (some error type) begin
        bypass_EN <= 1;
        mcause_bypass_in <= MCAUSE_ILLEGAL_INST; // or other error code
            , according to the error type
        mepc_bypass_in <= (1) epc_cur(exception) or (2) epc_next(
            interrupt);
        mtval_bypass_in <= (1) einst(illegal instruction) or (2) eaddr(
            load/store address misaligned) or (3) 32'h0(others);
        mstatus_bypass_in <= {mstatus[31:13], 2'b11, mstatus[10:8],
            mstatus[3], mstatus[6:4], 1'b0, mstatus[2:0]}; // set MPP
            to 11, MPIE(mstatus[7]) to MIE(mstatus[3]), MIE(mstatus[3])
             to 0
        in_trap <= 1; // set in_trap to 1
    end
```

其中，当发生中断时，还需判断 in_trap 信号是否为 1，若为 1 则不再处理中断。

```
    else if (interrupt && !in_trap) begin
```

完成异常或中断处理，进入 mret 指令时，将 in_trap 置为 0，具体实现如下：

```
    else if (mret) begin
        in_trap <= 0;
    end
```

最后完成时钟周期以及流水线寄存器的更新，**最初**具体实现如下：

```
1  assign PC_redirect = (illegal_inst || l_access_fault ||
       s_access_fault || ecall_m || interrupt && !in_trap) ? mtvec :
2                       (mret) ? mepc : 32'h0;
3  assign redirect_mux = (illegal_inst || l_access_fault ||
       s_access_fault || ecall_m || (interrupt && !in_trap) || mret) ?
        1'b1 : 1'b0;
4  assign reg_FD_flush = (illegal_inst || l_access_fault ||
       s_access_fault || ecall_m || (interrupt && !in_trap) || mret) ?
        1'b1 : 1'b0;
5  assign reg_DE_flush = (illegal_inst || l_access_fault ||
       s_access_fault || ecall_m || (interrupt && !in_trap) || mret) ?
        1'b1 : 1'b0;
6  assign reg_EM_flush = (illegal_inst || l_access_fault ||
       s_access_fault || ecall_m || (interrupt && !in_trap) || mret) ?
        1'b1 : 1'b0;
7  assign reg_MW_flush = (illegal_inst || l_access_fault ||
       s_access_fault || ecall_m || (interrupt && !in_trap) || mret) ?
        1'b1 : 1'b0;
8  assign RegWrite_cancel = (illegal_inst || l_access_fault ||
       s_access_fault || ecall_m || (interrupt && !in_trap) || mret) ?
        1'b1 : 1'b0;
9  assign MemWrite_cancel = (illegal_inst || l_access_fault ||
       s_access_fault || ecall_m || (interrupt && !in_trap) || mret) ?
        1'b1 : 1'b0;
```

但在实际测试中，发现中断发生时，当前进行的指令未能正常完成，经分析是RegWrite_cancel信号在interrupt时被错误地置为了1，导致了寄存器写入被取消，因此需要对RegWrite_cancel信号进行修改，如下：

```
1  assign RegWrite_cancel = (illegal_inst || l_access_fault ||
       s_access_fault || ecall_m || mret) ? 1'b1 : 1'b0;
```

其中，发生异常时，将 PC 重定向到 mtvec，同时将流水线寄存器清空，取消 RegWrite 和 MemWrite 操作。

发生中断时，将 PC 重定向到 mtvec，同时将流水线寄存器清空，取消 MemWrite 操作，但保留 RegWrite 操作。

进行 mret 指令时，将 PC 重定向到 mepc，同时将 in_trap 置为 0，完成异常或中断处理。

经过仿真分析以及下板验证，代码实现正确，能够正常处理异常和中断。

2 思考题

2.1 精确异常和非精确异常的区别是什么？

精确异常要求硬件能够精确定位触发异常的指令，确保该指令之前的所有操作均已完成且状态生效，后续指令完全取消，系统可回滚至一致状态适合对可靠性要求高的通用处理器（如CPU）。

非精确异常则允许异常发生后系统状态存在不确定性（如后续指令可能已部分执行），无法精准定位异常点，需软件介入修复，硬件设计简化但牺牲了状态一致性，常见于高吞吐场景（如GPU并行计算）或异步中断处理。

两者的选择本质上是硬件可靠性（精确）与执行效率（非精确）的权衡。

2.2 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？

第一条导致 Trap 的指令是 ecall 指令。(PC = 0x70)

进入 Trap 后，依次将 mepc、mcause、mstatus、mtval 寄存器的值写入 x26、x27、x28、x29 寄存器中；接着判断 x27 (mcause) 是否小于 0, 若小于 0 说明发生了中断，否则发生异常，异常时将 x26 (mepc) 加 4. 最后将 x26 (mepc) 的值写回 mepc 寄存器中，并调用 mret 指令，返回异常发生点，结束异常处理。

2.3 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？

访问 M-mode 受保护的 CSR 触发非法指令异常。

测试代码中，csr_test 访问了一些 M-mode 级别的 CSR（如 mscratch、mtvec、mepc、mcause 等）。在 U-mode 运行时，尝试访问这些寄存器会触发非法指令异常。

例如以下代码片段：

```
csrrwi x1, mscratch, 0x10 # PC = 0x2C, CSR[8] = 0x00000010
csrr x1, mscratch         # PC = 0x30, x1 = 0x00000010
csrrw x2, mscratch, x6    # PC = 0x34, CSR[8] = 0xffff0000, x2 = 0x00000010
csrr x1, mscratch         # PC = 0x38, x1 = 0xffff0000
```

2.4 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？

我认为可以不需要等到 WB 段后才送入异常处理模块，但需要谨慎处理。

当在对应阶段识别出异常时，可以立即将异常信息传递给异常处理模块，同时指明异常发生的具体位置；

对于发生异常之后的流水线阶段（之前的指令），则不进行 flush 等处理，使先前的指令继续执行，直到异常处理完成后再进行流水线寄存器的更新。

而对于发生异常之前的流水线阶段（之后的指令），则进行 flush 等处理，取消后续指令的执行，直到异常处理完成后再进行流水线寄存器的更新。

这样可以更快地响应异常，减少异常处理的延迟，提高异常处理的效率。