Lab 6 实验报告

一、实验目标

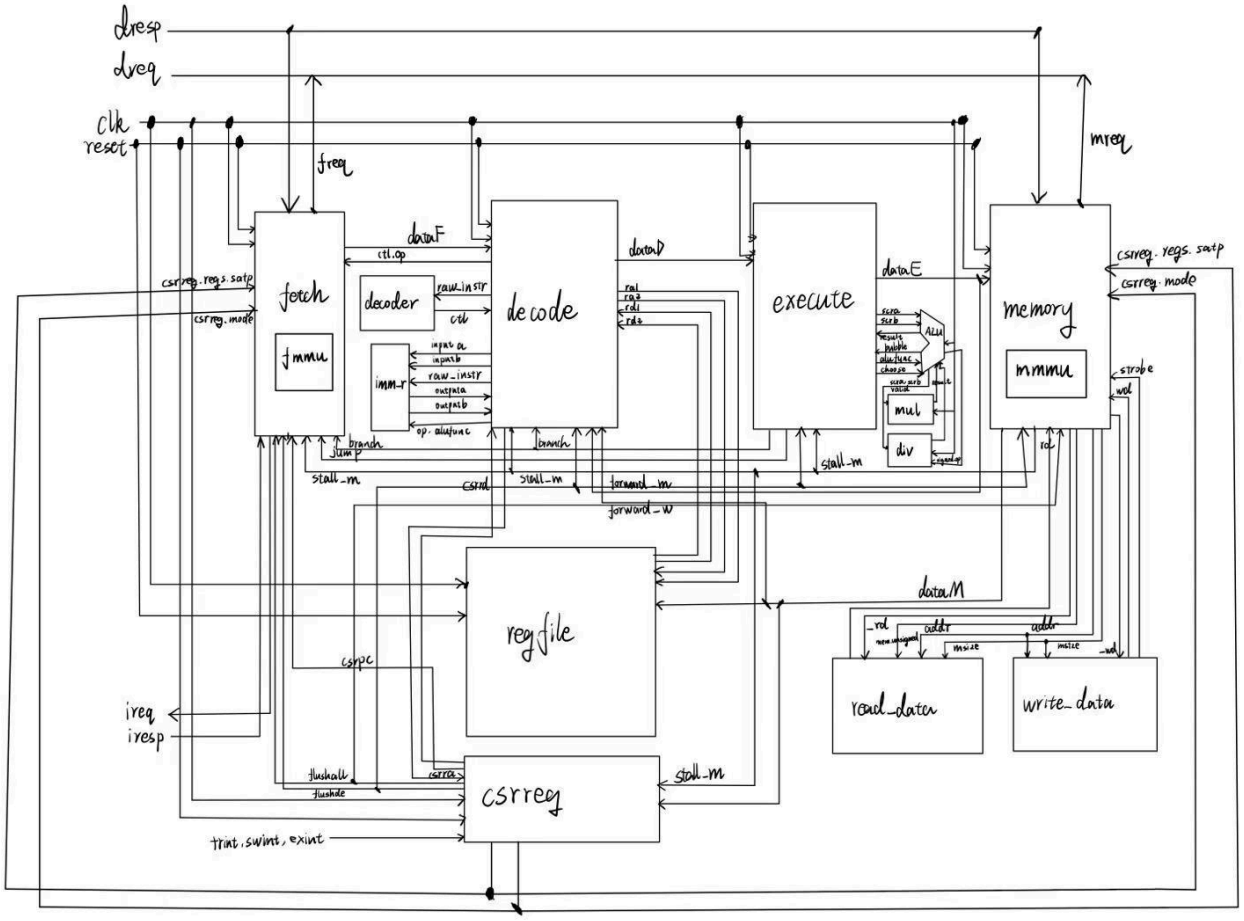
1. 实现异常：
   - 指令地址不对齐
   - 数据地址不对齐
   - 非法指令
   - ecall（lab5已经完成了）
2. 实现中断：
   - 时钟中断（trint）
   - 外部中断（exint）
   - 软件中断（swint）

二、模块层次

```
core
├── fetch          // 取指阶段
    ├── fmmu //实现页表功能
├── decode         // 译码阶段
    ├── decoder
    ├── imm_r
├── execute        // 执行阶段
    ├── alu
        ├── mul
        ├── div
            ├── divu
        ├── divu
├── memory         // 访存阶段
    ├── read_data
    ├── write_data
    ├── mmmu //实现页表功能
├── csr          // CSR寄存器文件
    ├── csrreg
```

```
└── regfile       // 寄存器文件
```

手绘电路图：



三、异常

定义结构`error_t`：

```
typedef enum logic [2:0] {
    NOERROR,           // 无异常
    INSTR_MISALIGN,    // 指令地址不对齐
    EDECODE,           // 非法指令
    LOAD_MISALIGN,     // Load指令数据地址不对齐
    STORE_MISALIGN     // Store指令数据地址不对齐
} error_t;
```

对所有流水线寄存器`fetch_data_t`、`decode_data_t`、`excute_data_t`、`memory_data_t`都加入`error_t error`字段，从而在流水线中逐级传播异常。

每一级的 `error` 字段需遵循以下优先级策略：

- 优先使用前一级传入的 `error`。
- 本级若检测出异常，且前级为 NOERROR，则本级异常生效。

指令地址不对齐

在fetch阶段检测：

```
dataF.error <= (pc[1:0]==2'b0)?NOERROR:INSTR_MISALIGN;
```

RV64 指令按4字节对齐；若最低两位非 `00`，则为地址不对齐异常。

非法指令

在decode阶段检测：

```
dataD.error <=dataF.error != NOERROR ? dataF.error :
                            ctl.op == UNKNOWN ? EDECODE : NOERROR;
```

此处`ctl.op == UNKNOWN`是在decoder中当对`raw_instr`的判断落入default当中时判断的:当 decoder 无法识别该指令格式时，`ctl.op` 被设置为 `UNKNOWN`，进而标记为非法指令。

数据地址不对齐

在memory阶段检测：

```
always_comb begin
        error_next=dataE.error;
        unique case(dataE.ctl.op)
            LD: begin
                error_next=dataE.valid&&(dataE.error==0)&&loaderror?
LOAD_MISALIGN:dataE.error;
            end
            SD: begin
                error_next=dataE.valid&&(dataE.error==0)&&storeerror?
STORE_MISALIGN:dataE.error;
            end
        endcase
    end
```

分LD和SD指令，其中`loaderror`信号在read_data模块中判断，`storeerror`信号在write_data模块中判断。

发生异常时的操作

图片参考实验文档Lab6.pdf

1. `mepc ← pc`
2. `next_pc ← mtvec`
3. `mcause[63] ← 0`表示异常，`mcause[62:0] ← 对应的异常类型`
4. `mstatus.mpie ← mstatus.mie`
5. `mstatus.mie = 0`
6. `mstatus.mpp ← mode`
7. `mode ← M Mode`
8. 清除流水线。取消当周期发起的 `dreq.valid`。已发起的 `dreq` 保留，等到 `data_ok` 后再清除流水线。

修改目标	赋值	作用说明
`flushde`、`flushall`	1	清空流水线中后续还未执行的指令，以防错误执行
`mode_next`	2'd3	切换到最高特权等级 Machine Mode
`csrpc`	`regs_next.mtvec`	设置程序跳转到异常处理入口（`mtvec` 寄存器中存储的地址）
`mepc`	`dataM.pc`	存储异常发生时的 PC，异常返回后从该地址继续执行
`mcause`	分情况，详见下图	存储异常或中断的原因代码
`mstatus.mpie`	`regs_next.mstatus.mie`	保存中断使能状态到 mpie
`mstatus.mie`	1'b0	关闭中断
`mstatus.mpp`	mode	保存原先的特权模式

`mcause` 具体赋值情况：

0	Instruction address misaligned
1	Instruction access fault
2	Illegal instruction
4	Load address misaligned
5	Load access fault
6	Store/AMO address misaligned
8	Environment call from U-mode
9	Environment call from S-mode
10	*Reserved*
11	Environment call from M-mode

四、中断

定义中断信号：

中断处理**实际发生**的条件是同时满足（1)当前是M Mode且`mstatus.mie=1` 或者 当前不是M Mode（2）`mip[i]=1` 且 `mie[i]=1`

```
assign interupt=dataM.valid&&regs.mstatus.mie &&((trint&&regs.mie[7])||
(swint&&regs.mie[3])||(exint&&regs.mie[11]));
```

在中断信号有效时，执行对应操作：

中断上下文寄存器配置表

修改目标	赋值	作用说明
`flushde`、`flushall`	`1`	清空流水线后续指令，防止中断响应期间的指令误执行
`mode_next`	`2'd3`	强制切换至Machine模式处理中断，确保特权隔离
`csrpc`	`regs_next.mtvec`	跳转至中断向量基地址（`mtvec`），启动中断服务程序
`mepc`	`dataM.pc`	保存中断返回地址，确保`MRET`指令能正确恢复执行流
`mcause`	分情况，详见下图	记录中断触发原因，指导中断服务程序的分支处理
`mstatus.mpie`	`regs.mstatus.mie`	保存原中断使能状态，为中断返回时的状态恢复提供依据
`mstatus.mie`	`1'b0`	关闭全局中断使能，防止中断嵌套
`mstatus.mpp`	`mode`	记录原特权模式，支持`MRET`后的模式恢复

`mcause` 具体赋值情况：

1	Supervisor software interrupt
3	Machine software interrupt
5	Supervisor timer interrupt
7	Machine timer interrupt
9	Supervisor external interrupt
11	Machine external interrupt

五、实验结果

- 实现异常与中断
- 通过Lab6测试（中间循环的地方略去）

```
Emu compiled at May 15 2025, 10:55:41
The image is ./ready-to-run/lab6/lab6-test.bin
Using simulated 256MB RAM
Single test passed.
Run sys-test
trap here, epc 00, cause 0
Test ecall_u [OK]
trap here, epc 00, cause 0
trap here, epc 00, cause 0
Test instr_misalign [OK]
trap here, epc 00, cause 0
trap here, epc 00, cause 0
Test load_misalign [OK]
trap here, epc 00, cause 0
trap here, epc 00, cause 0
Test store_misalign [OK]
trap here, epc 00, cause 0
trap here, epc 00, cause 00
```

```
Test instr_misalign [OK]
trap here, epc 00, cause 0
trap here, epc 00, cause 0
Test load_misalign [OK]
trap here, epc 00, cause 0
trap here, epc 00, cause 0
Test store_misalign [OK]
trap here, epc 00, cause 0
trap here, epc 00, cause 00
Test timer_intr [OK]
trap here, epc 00, cause 0
trap here, epc 00, cause 00
Test software_intr [OK]
trap here, epc 00, cause 0
Timer interrupt in test_trap, this should happen 50 times.
Timer interrupt in test_trap, this should happen 50 times.
Timer interrupt in test_trap, this should happen 50 times.
Timer interrupt in test_trap, this should happen 50 times.
Timer interrupt in test_trap, this should happen 50 times.
Timer interrupt in test_trap, this should happen 50 times.
Timer interrupt in test_trap, this should happen 50 times.
```

```
Timer interrupt in test_trap, this should happen 50 times.
Timer interrupt in test_trap, this should happen 50 times.
Timer interrupt in test_trap, this should happen 50 times.
Timer interrupt in test_trap, this should happen 50 times.
Timer interrupt in test_trap, this should happen 50 times.
Timer interrupt in test_trap, this should happen 50 times.
Timer interrupt in test_trap, this should happen 50 times.
Test m_trap [OK]
Privileged test finished.
m_trap_test [X]
---TEST FAILED---
m_trap_test [X]
---TEST FAILED---
m_trap_test [X]
```