Lab 5 实验报告

一、实验目标

1. 实现指令：MRET ECALL

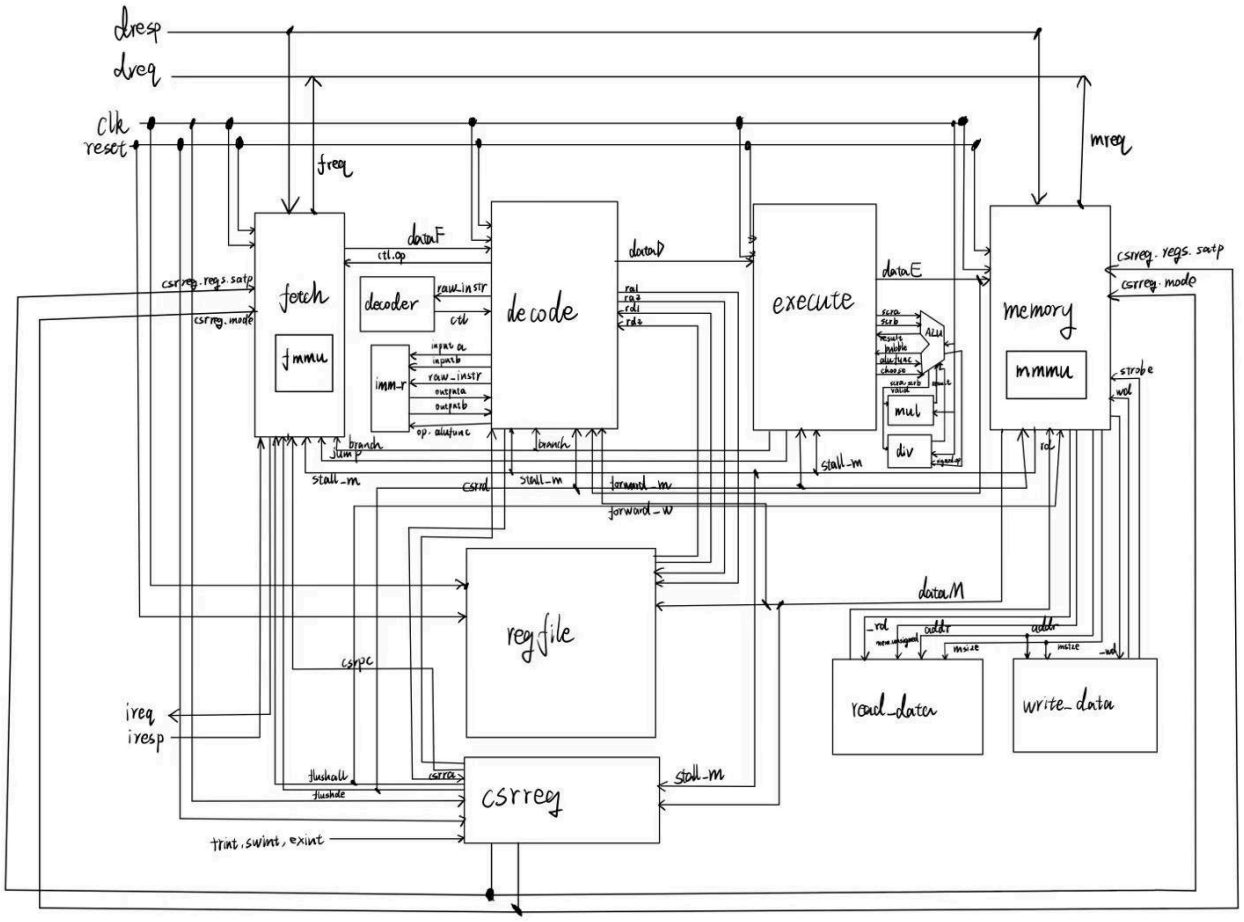
   实现页表功能MMU

2. 完成Vivado仿真与上板

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、实现MRET和ECALL指令

由于新增的两个指令涉及对机器的特权级的修改，故需要先在csrreg中实现mode和mode_next寄存器，然后将mode_next寄存器对应连接到 `DifftestCSRState` 的priviledgeMode接口即可。其在csrreg中大致定义如下：

```
u2 mode;
u2 mode_next;
always_ff @(posedge clk) begin
    if (reset) begin
        mode <= 2'd3;
    end else begin
        mode <= mode_next;
    end
end
```

其中，RISC-V 的特权级定义为：

名称	编码	描述
User Mode	`00`	用户模式
Supervisor Mode	`01`	管理员/操作系统
Reserved	`10`	保留
Machine Mode	`11`	最高特权级，硬件/启动态

其运行逻辑如图，

```
              正常运行
                 |
              异常 / 中断 (此处为ECALL指令)
                 |
      +----------v-----------+
      |    mode = 2'b11      | <- 切换到 Machine 模式
      | mstatus.mpp = 原mode |
      +----------+-----------+
                 |
              处理中断
                 |
              执行 MRET(异常返回)
                 |
        +--------v-------+
        |   mode = mpp   | <- 恢复特权模式
        +----------------+
```

故其mode_next的赋值逻辑为：

```
always_comb begin
    mode_next = mode;
    if(~stall_m && dataM.ctl.op == ECALL && dataM.valid)
        mode_next = 2'd3;
    else if(~stall_m && dataM.ctl.op == MRET && dataM.valid)
        mode_next = regs_next.mstatus.mpp;
end
```

MRET

`MRET` 是从机器模式返回到之前的模式的指令。具体修改内容如下：

修改寄存器	修改后含义
`csrpc`	将PC设置为 `regs_next.mepc`，即异常前指令地址
`regs_next.mstatus.mie`	设置为 `regs_next.mstatus.mpie`，恢复异常前中断状态
`regs_next.mstatus.mpie`	置为 1，表示以后能再次触发中断
`regs_next.mstatus.mpp`	设置为 0（用户模式），代表完成特权级返回
`mode_next`	设置为 `regs_next.mstatus.mpp`，更新当前特权级

ECALL

`ECALL` 是用户或系统程序触发陷入时通过引发环境调用异常来请求执行环境，进入异常处理流程的指令。具体修改内容如下：

修改寄存器	修改后含义
`regs_next.mepc`	设置为当前的 `dataM.pc`，即触发 `ecall` 的指令地址
`regs_next.mcause`	设置为异常码：`11`（M 模式下环境调用）或 `8`（U 模式下环境调用）
`regs_next.mstatus.mpie`	设置为当前 `mie` 的值，表示保存中断状态
`regs_next.mstatus.mie`	清 0，关闭中断，防止异常处理中被打断
`regs_next.mstatus.mpp`	设置为当前特权级
`csrpc`	将PC设置为 `regs_next.mtvec` 的值，跳转到异常处理入口地址
`mode_next`	设置为 `3`（M 模式），进入机器模式进行异常处理

其中，`mcause`寄存器的具体设置见下图（来自中文手册），本次lab只涉及Environment call from U-mode和Environment call from M-mode

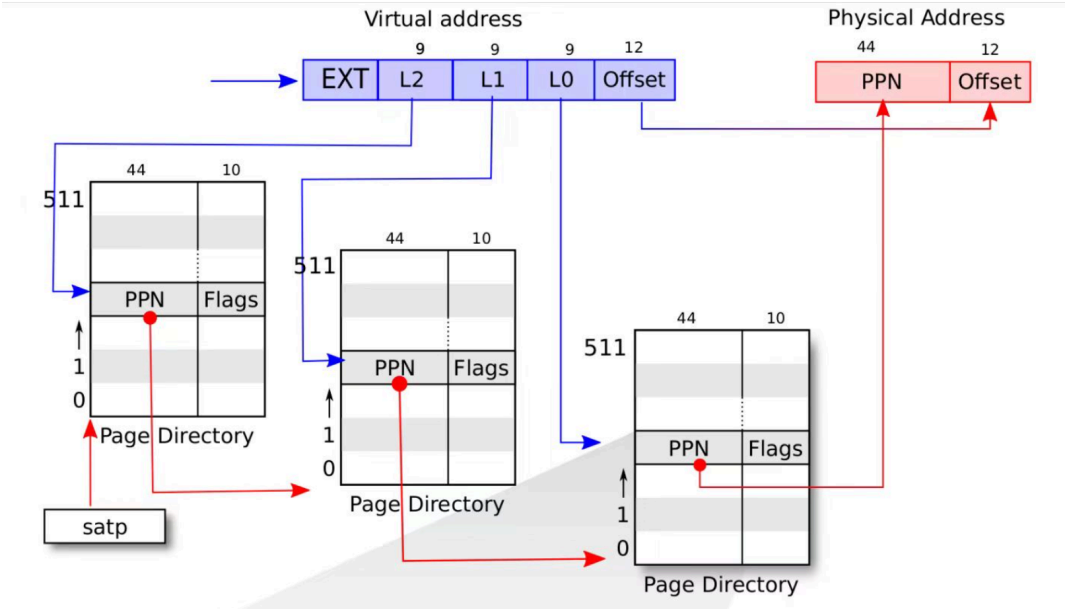
Interrupt / Exception mcause[XLEN-1]	Exception Code mcause[XLEN-2:0]	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
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 address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

最后，MERT指令和ECALL指令都需要进行刷新，即将`flushde`和`flushall`都置为1。其余decode等处修改与之前的指令实现类似，此处略去。

四、页表功能实现

MMU（Memory Management Unit，内存管理单元）是一种硬件模块，用于在CPU和内存之间实现虚拟内存管理，其主要功能是将虚拟地址转换为物理地址。本次lab只需要实现三级页表机制（Sv39）。Sv39使用4KB大的基页，页表项的大小是8个字节，为了保证页表大小和页面大小一致，树的基数相应地降到2^9，树也变为三层。Sv39将虚拟地址划分为2^9个 1GB大小的吉页 。每个吉页被进一步划分为2^9个2MB大小的巨页。每个巨页再进一步分为2^9个4KB大小的基页。

大致思路见下图：



用于控制内存分页的寄存器为satp，其具体布局为：

```
typedef struct packed {
    u4 mode;  // [63:60]  用于选择分页模式
              // 0：Bare，No translation or protection ；8：Sv39，Page-based 39 bit virtual
addressing
    u16 asid; // [59:44]  暂时不用管，全置零即可
    u44 ppn; //  [43:0]   保存根物理页号，实际值为「根页表物理地址右移 12 位」
} satp_t;
satp_t satp;
```

本次MMU通过一个状态机实现，有三种状态：

```
enum logic [1:0] {
        IDLE,     //等待开始
        READ,     //读取页表项
        DONE      //地址翻译完成
    } state, next_state;
always_ff @(posedge clk) begin
    if (reset) state <= IDLE;
    else state <= next_state;
end
```

`IDLE` 状态：

如果 `satp.mode == 0`（即Bare模式）或当前是 machine mode(`mmode == 'b11`)，则直接输出虚拟地址作为物理地址（`next_pa = va`），MMU 不介入。

否则开始页表遍历，从 `satp.ppn` 开始作为页表基地址；设置当前页表级别为2（Sv39有3级页表：L2, L1, L0）；进入 `READ` 状态。

`READ` 状态：依据不同的PTE情况分情况处理，PTE结构如图10.11（来自中文手册）所示，其中：

1. V位决定了该页表项的其余部分是否有效（V = 1时有效）。若V = 0，则任何遍历到此页表项的虚址转换操作都会导致页错误。
2. R、W和X位分别表示此页是否可以读取、写入和执行。如果这三个位都是0， 那么这个页表项是指向下一级页表的指针，否则它是页表树的一个叶节点。
3. PPN域包含物理页号，这是物理地址的一部分。若这个页表项是一个叶节点，那 么PPN是转换后物理地址的一部分。否则PPN给出下一节页表的地址。

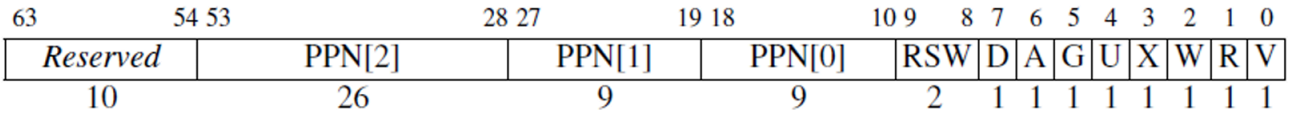


图 10.11：一个 RV64 Sv39 页表项（PTE）。

- **无效PTE**：`pte[0] == 0`（V = 0）→ 无效页表项表示这个地址没有映射，翻译结束，状态进入 `DONE`，直接结束转换流程。
- **叶子PTE**：`pte[3:1] != 'b000`（XWR = 0）→ 是叶子页表项，表示找到了物理页框地址，于是生成物理地址并结束转换：
  - `pte[53:10]` 是PPN域，表示物理页框地址
  - `va[11:0]` 是页内偏移offset
  - 组合得出最终 `next_pa = {8'b0, pte[53:10], va[11:0]};`
  - 状态转为 `DONE`
- **非叶子PTE**：继续遍历下一级页表，直到达到最大层级或遇到错误：
  - 将 PTE 中指向的下一层页表地址提取出来作为新的 `ppn`；
  - 若 `level == 0`，表示已经到最后一级，下一状态转为 `DONE`；
  - 否则，继续读下一层 PTE，状态保持为 `READ`，`level--`。

其中，每一级生成访问页表项（PTE）的内存地址（`mem_addr`）为：

{ 8'b0, ppn (44 bits), VPN[i] (9 bits), 3'b000 } = PPN << 12 | VPN[i] << 3

> ppn: 当前页表的物理页号（Page Table Base PPN）。
>
> VPN[i] : 当前级别使用的虚拟地址 VPN 段，用作索引。
>
> `{..., 3'b000}`: 页表每项是 8 字节（2^3），所以乘以 8 = 左移3位，表示字节地址。
>
> `{8'b0,...}`: 因为总长为64，所以剩下位置用0填充。

```
always_comb case (level)
    2: mem_addr = {8'b0, ppn, va[38:30], 3'b000};
    1: mem_addr = {8'b0, ppn, va[29:21], 3'b000};
    0: mem_addr = {8'b0, ppn, va[20:12], 3'b000};
endcase
```

`DONE` 状态：

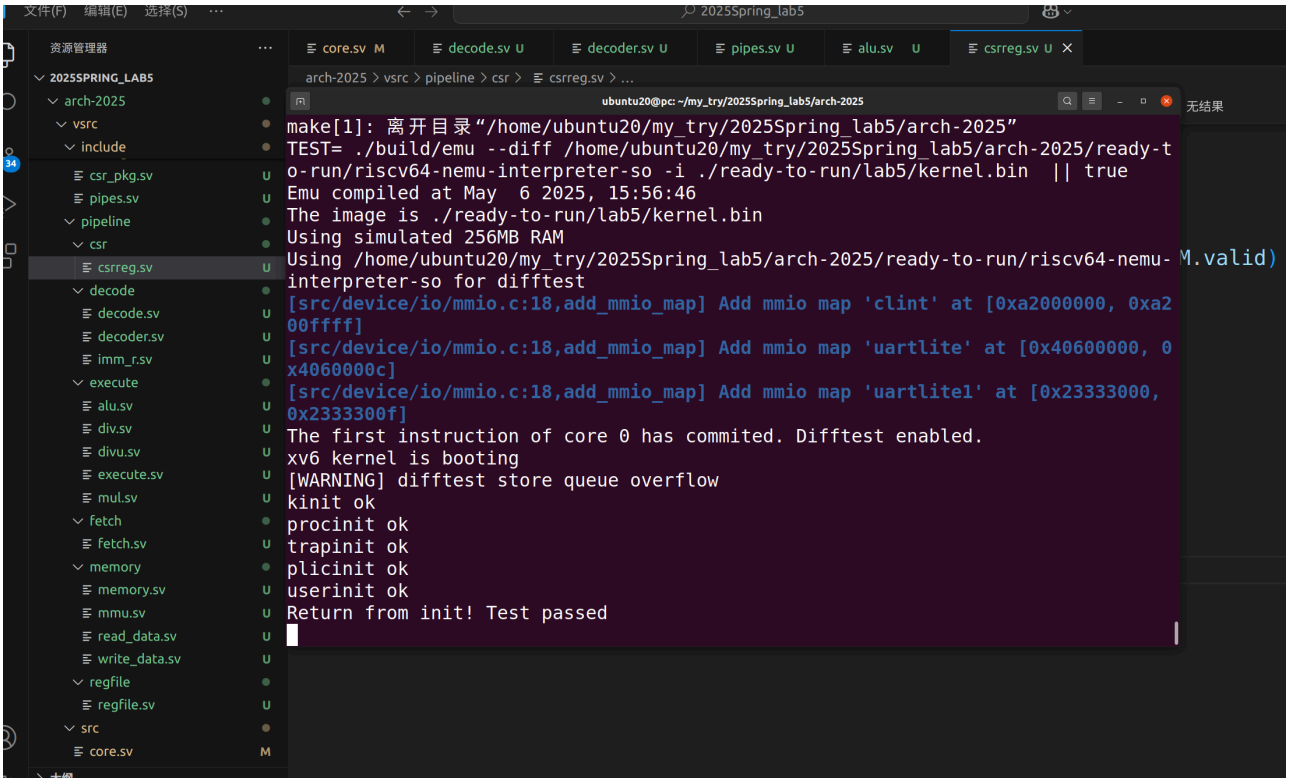
如果 `en` 仍然为 1，就停在 DONE；如果 `en` 拉低，表示翻译请求结束，回到 `IDLE`。

五、实验结果

- 实现指令：MRET ECALL

  实现页表功能MMU

- 通过Lab5测试



- Vivado仿真与上板