

Lab 5

初步分析与思路

考虑到我们已经实现了 CSR 的读写，因此优先考虑 `ecall` 和 `mret` 指令。

在经过查阅资料后，发现在 RISC-V 中，`privilege Mode` 并不对程序可见，因此对于 CPU 设计来说，我们应当自己在某一个位置保存当前流水线的 `privilege Mode`。并且，内存操作应该考虑当前的 `privilege Mode`。

考虑到我们的 CPU 在开始执行时运行在 M 模式，因此肯定会先执行 `mret` 指令。

mret 指令

`mret` 指令的作用是从 M 模式回到先前的模式（S 模式或 U 模式），并恢复之前的上下文。它会将 `mepc` 中保存的地址作为下一条指令的地址，根据 `mstatus` 修改 `privilegeMode`，并修改 `mstatus`。

因此考虑

- 新的 `mstatus` 应该为 `{mstatus[63:8], 1'b1, mstatus[6:4], mstatus[7], mstatus[2:0]}`
- 新的 `privilegeMode` 应该为 `mstatus[12:11]`
- 跳转到 `mepc` 中保存的地址
- 需要刷新流水线

最终：

```
moduleOut.isCSRWrite <= 1;
moduleOut.isCSRWrite2 <= 0;
moduleOut.isCSRWrite3 <= 0;
moduleOut.CSR_addr <= 12'h300; // mstatus
moduleOut.CSR_write_value <=
{mstatus[63:8], 1'b1, mstatus[6:4], mstatus[7], mstatus[2:0]};

priviledgeModeWrite <= 1;
newPriviledgeMode <= mstatus[12:11];
```

ecall 指令

`ecall` 指令的作用是触发一个（同步的）异常，通常用于系统调用或中断处理，会跳到中断向量地址（`mtvec`），并将当前的 `mepc` 保存到 CSR 中。

因此考虑

- 新的 `mepc` 应该为 `pc`
- 新的 `mstatus` 应该为 `{mstatus[63:13], priviledgeMode, mstatus[10:8], mstatus[3], mstatus[6:4], 1'b0, mstatus[2:0]}`
- 新的 `mcause` 应该为 8，考虑到我们只有 U 模式
- 新的 `privilegeMode` 应该为 3（M 模式）

- 跳转到 `mtvec` 中保存的地址
- 需要刷新流水线

最终：

```
moduleOut.isCSRWrite <= 1;
moduleOut.isCSRWrite2 <= 1;
moduleOut.isCSRWrite3 <= 1;
moduleOut.CSR_addr <= 12'h300; // mstatus
moduleOut.CSR_write_value <=
{mstatus[63:13],privilegeMode,mstatus[10:8],mstatus[3],mstatus[6:4],1'b0,m
status[2:0]};
moduleOut.CSR_addr2 <= 12'h341; // mepc
moduleOut.CSR_write_value2 <= moduleIn.instrAddr;
moduleOut.CSR_addr3 <= 12'h342; // mcause
moduleOut.CSR_write_value3 <= 8;

privilegeModeWrite <= 1;
newPrivilegeMode <= 3;
```

CSR 寄存器的读写

注意到以上命令可能执行至多 3 个 CSR 寄存器的读写，因此我们需要在 `writeback` 模块中增加对 `CSR_addr2` 和 `CSR_addr3` 的处理。

MMU

MMU 是本次 Lab 的核心内容（说白了前面两条指令应该是为了跳到 U 模式以便使用 MMU）。

起初我打算在 `memory` 模块中实现 MMU，但仔细思考之后认为这种方式存在问题，假设在 `memory` 中实现 MMU，是可以做的，即多次使用 `dbus` 以访问页表，但存在一个严重问题是：对于指令的读取（`fetch`），我们也需要使用 MMU 来进行地址转换，但 `ibus` 固定为 4 字节，无法访问完整页表，并且我并不想修改已有的 `ibus` 定义。

考虑到之前提到：

- `ibus` 是 `dbus` 的一个子集
- `ibus` 和 `dbus` 在最后实现了一个复用器（实际上到内存的总线只有一根）

因此我们考虑在复用器后面实现 MMU，这样无论是 `ibus` 还是 `dbus` 都会到这里，并且只需要实现一个。

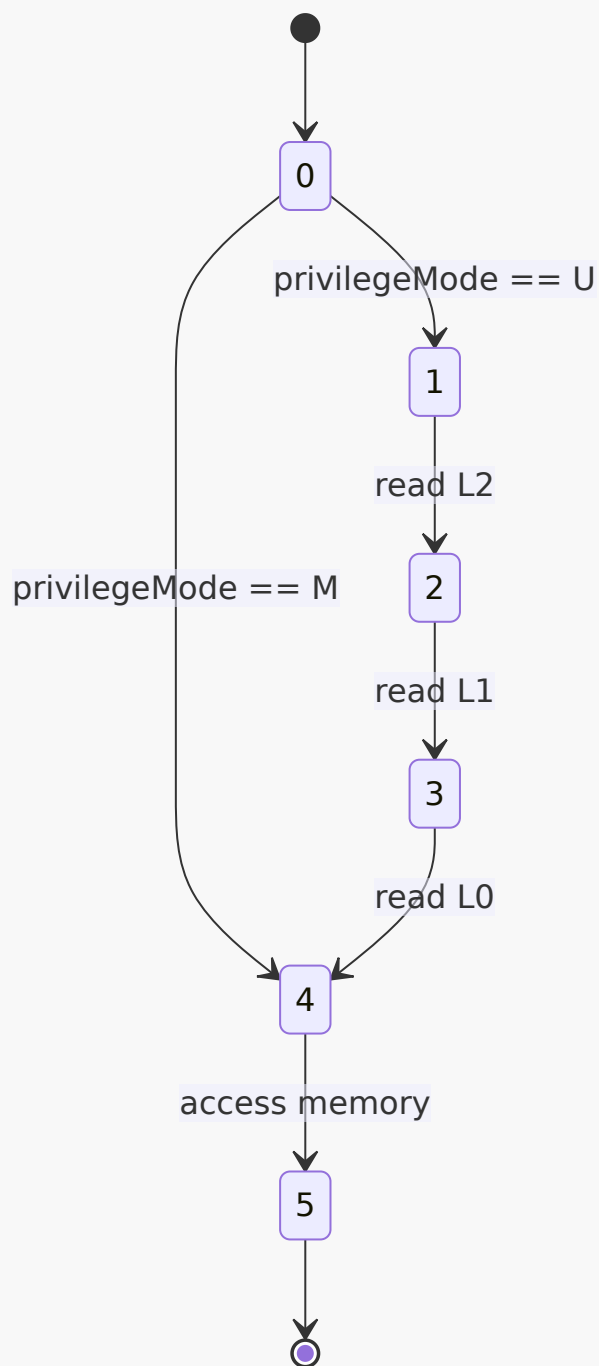
根据 Mode 39 的要求，我们应该实现状态机，考虑为了获取物理地址，我们需要进行 3 次访存，对应三级页表的地址。

- 状态 0：当前 MMU 处于空闲状态，等待指令或数据访问请求
- 状态 1：当前 MMU 正在使用 `satp` 为基址，L2 作为偏移量，读取第一级页表（第二级页表的基址）
- 状态 2：当前 MMU 正在使用第一级页表的值为基址，L1 作为偏移量，读取第二级页表（第三级页表的基址）
- 状态 3：当前 MMU 正在使用第二级页表的值为基址，L0 作为偏移量，读取第三级页表中的物理地址
- 状态 4：当前 MMU 正在使用物理地址访问内存

- 状态 5：访问完毕

状态转移：

- 0 -> 1：接收到指令或数据访问请求，且 privilegeMode 为 U 模式（需要使用地址转换）
- 0 -> 4：接收到指令或数据访问请求，且 privilegeMode 为 M 模式（不需要使用地址转换）
- 1 -> 2：读取第一级页表成功
- 2 -> 3：读取第二级页表成功
- 3 -> 4：读取第三级页表成功
- 4 -> 5：访问内存成功
- 5 -> 0：访问完毕，返回空闲状态（总是执行，因此会在状态 5 停留一个周期）



需要注意的是，由于沟通 `ibus`、`dbus` 与内存总线的是用 `SimTop`（和 `VTop`，用于 Vivado），因此我们需要在 `SimTop`（和 `VTop`）中增加一根线，用于传输当前的 `privilegeMode`，并且需要在 `CBusArbiter` 重定向其内存总线的处理方式。

另外，仍然考虑之前 Lab 3 提到的，需要对内存访问的第 31 位进行判断，以判定其访问的是内存还是外设，这里的判断其实是物理地址而非虚拟地址，因此不能再使用 ALU 的输出作为判定依据，而是要在 MMU 中根据最后页表的输出进行判断。这又需要增加几根线。

总体时序问题修复

当增加了以上内容之后，流水线的时序出现了一些小问题：

CSR 刷新流水线的问题

之前考虑的是：CSR 写入的影响会从 decode 阶段开始影响某一条指令，因此是这样做的：

memory 阶段根据当前是否为 CSR 写入，产生一个跳转信号，若跳转，则在上升沿：

- 作废即将从 `fetch` 阶段到 `decode` 阶段的指令
- 作废即将从 `decode` 阶段到 `execute` 阶段的指令
- 作废即将从 `execute` 阶段到 `memory` 阶段的指令
- 设置 `fetch` 下一周期获取的为 `mtvec` 中的地址
- 传递当前指令给 `writeback` 阶段，写寄存器和 CSR

在之前的 Lab 4 是没有问题的，但是请注意：这里的 CSR 寄存器实际上是在下一个周期中才开始写入的，这个周期 `decode execute memory` 都在空转，无需在意，但是，`fetch` 可是在获取下一条指令（即冲刷流水线之后重新进入流水线的第一条指令），注意进入这个周期时 CSR 还没完成写入。

Lab 5 的问题是：CSR 从 fetch 阶段就开始影响指令执行了

这个的原因是需要确定这条指令取指时是否要使用 MMU 进行地址转换。

因此，我们增加一条信号，如果是 CSR 写入，`fetch` 下一个周期也不能取指，应该 stall 一个周期。

注意到这个逻辑和之前为了避免 Load-use Hazard 的逻辑是一样的，因此我们可以将其合并为一个逻辑。

fetch 中

```
if(JumpEn) begin
    // ...
    stall <= csrJump;
end else if(~lwHold & ~stall) begin
    // ...
end else begin
    // ...
    stall <= 0;
end
```

memory 中

```
assign csrJump = moduleIn.isCSRWrite & moduleIn.valid;
```

上板

每次上板都会出现一堆问题，以下按遇到和解决的时间顺序记录：

(遇到) 卡在 xv6 kernel is booting

在上板时，只输出 xv6 kernel is booting。

遂尝试仿真

(遇到) 仿真只跑了 3 个指令就不跑了

注意到 Vivado 的仿真对以下写法不友好：

```
u3 funct3 = instr[14:12];  
u7 funct7 = instr[31:25];
```

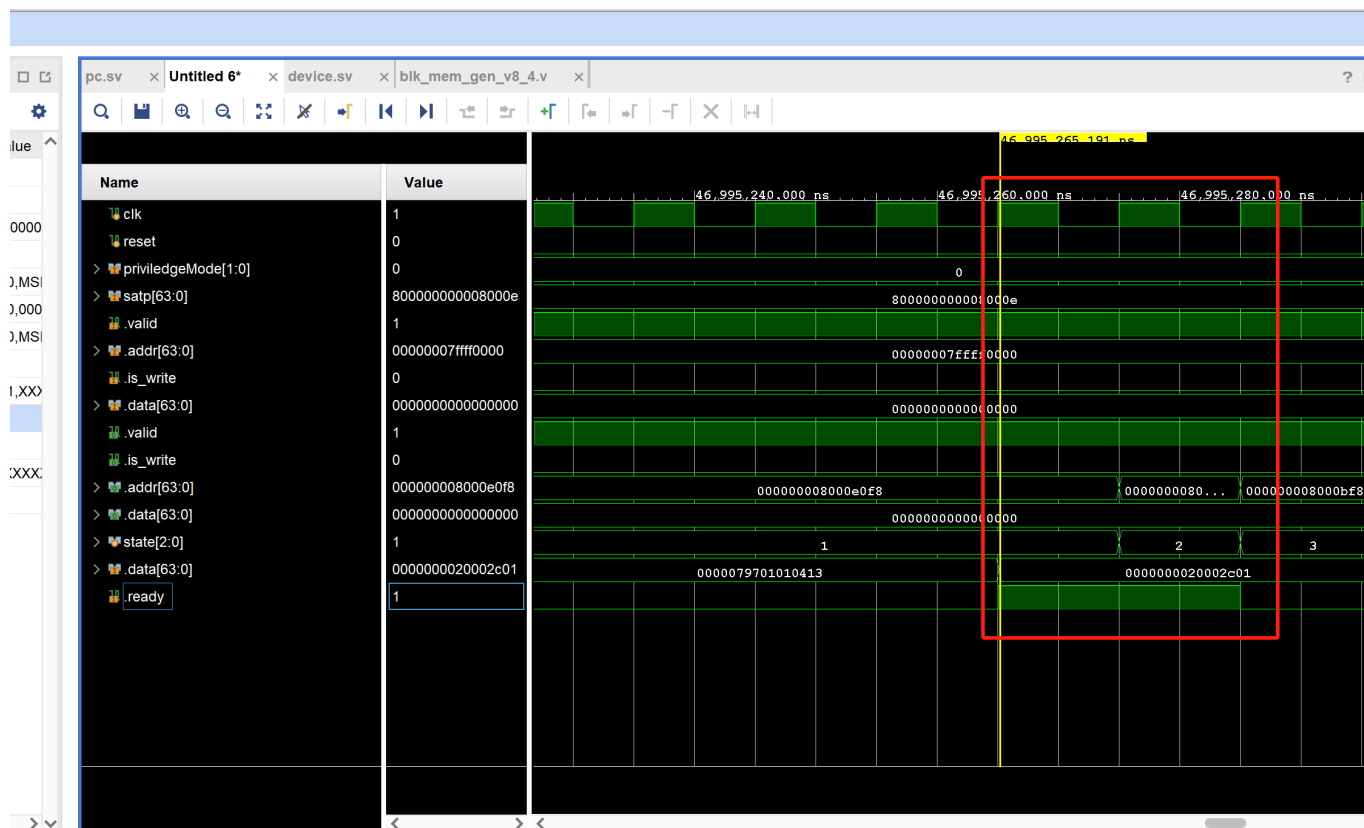
(解决) 仿真只跑了 3 个指令就不跑了

故改为

```
u3 funct3;  
assign funct3 = instr[14:12];  
u7 funct7;  
assign funct7 = instr[31:25];
```

(遇到) 切换到用户模式后无法访存（页表访问存在问题）

注意到



在上板使用的内存总线中，当内存访问完毕后 **.ready** 和 **.last** 信号会保持**两个周期**的高电平，这会导致状态机直接跳过一个状态（图中为 2）。

(解决) 切换到用户模式后无法访存（页表访问存在问题）

为了避免影响 Verilator 的速度，使用宏来针对性地解决这个问题：

```
`ifndef VERILATOR
logic ok_2_state;
u64 temp;
`endif
```

当在 Vivado 内：

```
if (cbus_resp_from_mem.ready&&cbus_resp_from_mem.last) begin
    if(ok_2_state) begin
        ok_2_state <= 0;
        // and state <= next state
        ... // use temp instead of cbus_resp_from_mem.data
    end else begin
        ok_2_state <= 1;
        temp <= cbus_resp_from_mem.data;
    end
end else
```

此时，Vivado 仿真已正常（即使是 Post-Implementation）：

```
› launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:18 . Memory (MB): peak = 7099.848 ; gain
› run all
  xv6 kernel is booting
  kinit ok
  procinit ok
  trapinit ok
  plicinit ok
  userinit ok
  Return from init! Test passed
› run: Time (s): cpu = 00:00:28 ; elapsed = 00:00:22 . Memory (MB): peak = 7099.848 ; gain = 0.000
```

(解决) 卡在 xv6 kernel is booting

发现 Timing 报 Warning: `ireq.addr` 有 `no_clock` 错误。

注意到 Fetch 这个地方：

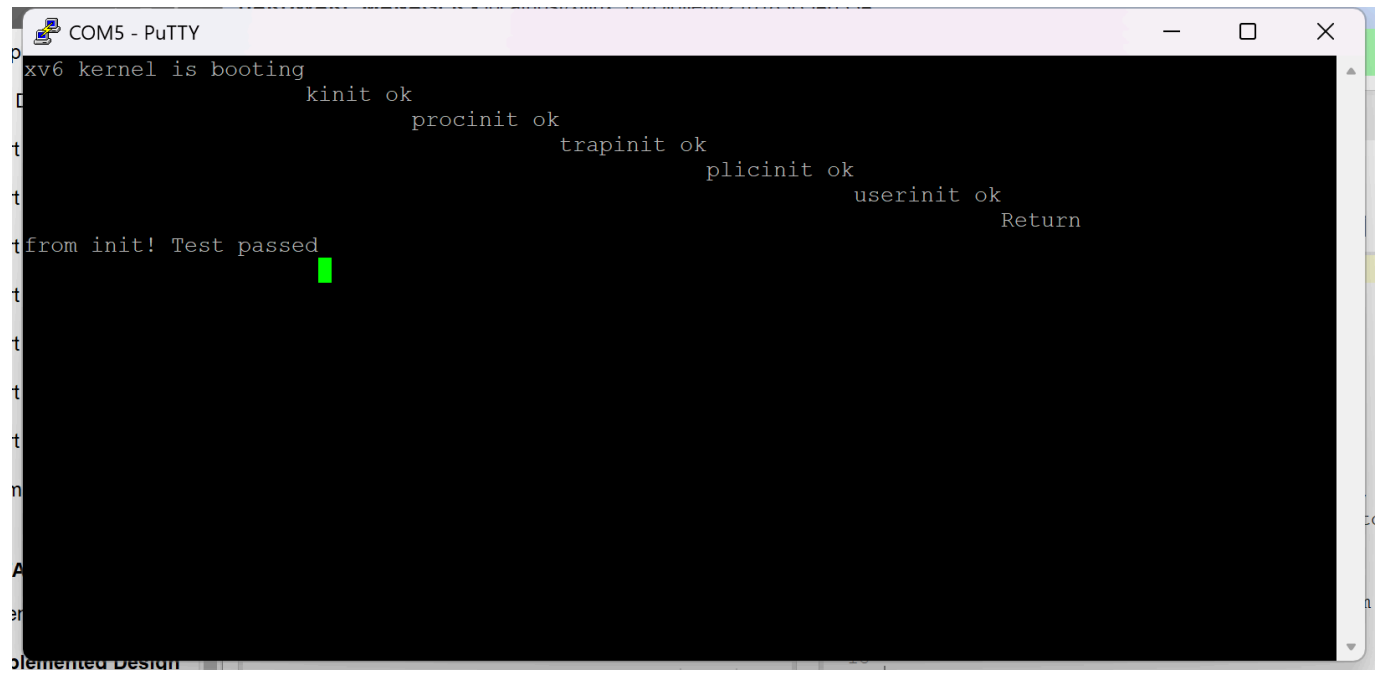
```
if(rst) begin
    curPC <= PCINIT;
    ...
    ibus_req.addr <= curPC;
    ...
end
```

是有问题的，会导致 `ibus_req.addr` 和 `curPC` 被处理时出现问题（况且这个地方本来逻辑也是错的）

因此改为：

```
if(rst) begin
    curPC <= PCINIT;
    ...
    ibus_req.addr <= PCINIT;
    ...
end
```

最终：



```
COM5 - PuTTY
xv6 kernel is booting
    kinit ok
        procinit ok
            trapinit ok
                plicinit ok
                    userinit ok
                        Return
from init! Test passed
```