期末检查

王艺杭
2023202316

流水线旁路机制分析

1. 结合你的代码，讲一讲下面的机制是怎么体现的
    1. 怎么识别后执行的指令是否用了前执行的指令还没写回寄存器的内容?
        1. Decode 阶段把当前指令的 `rs/rt/rd` 拆出来，并产生“将来是否写回(`reg_write`)、写到哪个寄存器(`dest`)”等控制信号
        2. 在 `ForwardUnit` 中，如果当前（ID 或 EX）要读的寄存器号 `id_rs/id_rt` 或 `ex_rs/ex_rt`，等于前面某一级将要写回的 `dest`，且该级 `*_reg_write` 为 1，且寄存器号不是 `$0`，则判定存在 RAW 相关。然后输出 `fwd_*_sel`（选择从哪一级把值旁路过来）。
        3. 即使不靠 `ForwardUnit`，寄存器堆也做了一个小旁路：同一拍里如果 WB 正在写 `waddr`，而读口读的也是这个寄存器，就直接把 `wdata` 回送到读数据
    2. 指令 stall 在你的代码里什么地方涉及？你的代码通过什么方法判断需要 stall 几个周期的?
        1. 哪里涉及 stall：
            - 产生 stall/flush/bubble 控制信号：`HazardUnit`
            - 用这些控制信号去“冻结寄存器/插泡/停 PC”：`PipelineCPU`（再加上 `PipeReg*` 的 `en/flush` 机制）
        2. 通过什么方法判断需要 stall 几个周期?
           stall 持续多久取决于 `HazardUnit` 的条件在连续多少个周期为真。
    3. 旁路机制是怎么跨流水线转发的？你的代码通过什么方法判断需要使用旁路数据而不是真的寄存器数据?
        1. 旁路数据从哪来：
            - EX 刚算出的结果：用组合路径 `ex_mem_d.ex_result`（还没写进 EX/MEM 寄存器）提供给 ID 级分支/跳转比较用
            - MEM 级“可转发值”：`mem_fwd_value = load ? mem_load_data : ex_mem_q.ex_result`
            - WB 级“可转发值”：`wb_fwd_value = mem_wb_q.wb_data`
        2. 怎么判断“用旁路而不是用寄存器堆读值”

           在 `ForwardUnit` 中对每个源寄存器（`id_rs/id_rt` 或 `ex_rs/ex_rt`）做比较
            - 条件形如：`src != 0 && stage_reg_write && stage_dest != 0 && stage_dest == src`，命中则输出选择信号 `fwd_*_sel`，表示“从哪个流水级的结果旁路过来”。
            - 优先级写死在 if/else 顺序里，ID 级 mux：`fwd_id_rs_sel/fwd_id_rt_sel` 决定送给 Branch/Jump 单元的比较操作数；EX 级 mux：`fwd_ex_rs_sel/fwd_ex_rt_sel` 覆盖 `id_ex_q.rs_val/rt_val`（或 MDU stall 时的 base），决定 ALU/MDU 的真实输入
            - WB 同周期写回且 ID 同周期读同一寄存器时，寄存器堆直接返回 `wdata`

乘法单元机制分析

我们提供的乘法器 (MDU) 运算需要多个周期。因此，将乘法器和旁路机制结合有独特的挑战。结合你的代码回答以下问题（Lx 代表下面代码的第 x 行）

```
mult $1, $2
addu $10, $11, $12 # 一个无关指令
mflo $3
mfhi $4
addu $3, $3, $4 # 一个用了乘法输出的指令
mult $1, $2 # 一个乘法
mflo $3
mfhi $4
mult $5, $6 # 另一个乘法
mflo $7
mfhi $8
addu $3, $3, $7 # 处理结果
```

1. 你的乘法器被占用时，不使用乘法器的指令（L2）可以正常执行吗？如果能正常执行，你是怎么做的？如果不能，进行什么修改能让你的 CPU 在进行乘法运算时可以执行其他不使用乘法器的指令？请结合你代码的 stall 生成机制作答。

   可以。

   生成 MDU stall 的条件是：`mdu_hazard = ex_is_mdu && mdu_busy`，也就是说必须同时满足：

   - `ex_is_mdu`：EX 阶段这条指令是 MDU 类（mult/div/mflo/mfhi/mtlo/mthi）
   - `mdu_busy`：乘法器正忙

   `mult` 在它进入 EX 的那一拍，如果 `!mdu_busy`，就会产生一次 `mdu_start` 脉冲。此时 `mdu_busy` 还是 0，所以 `mdu_hazard` 不成立，不会 stall。下一拍开始 `mdu_busy` 会变成 1，但这时 EX 阶段已经是 L2（`addu`），它不是 MDU 指令，所以 `ex_is_mdu=0`，`mdu_hazard` 仍然不成立 → 不会因为 MDU 忙而停 IF/ID/EX，L2 正常走 ALU

2. 实验手册要求乘法器被占用时，任何使用乘法器的指令都会被卡在 EX 级。请结合你的代码，构思一种方法使得上面的代码在执行时，不卡在 L3 L4，而卡在 L6。

   > 注意：L6 从 EX 级恢复执行前，往 $3 $4 写入应当重新插入回流水线的 WB 级中。流水线开始正常流动前要保证一级流水线上只有一条指令。

   1. 新增一个“小队列 + scoreboarding”
      增加一个很小的 pending 队列：
      - `pending_q[i] = {valid, dest_reg, is_lo}`（is_lo=1 表示 mflo，0 表示 mfhi）
      - 同时维护一个 `pending_mask[31:0]`：某个寄存器是否“有一个来自 mflo/mfhi 的写回还没发生”
   2. L3/L4：遇到 mflo/mfhi 且 MDU 忙时，不让它们变成 EX 的 MDU 指令

      在 ID→EX（构造 `id_ex_d`）的地方做“摘除”：当前指令是 `mflo/mfhi`（可用 `id_mdu_cmd==MDUCMD_READ_LO/HI` 判定），且 `mdu_busy==1`：把 `{dest, is_lo}` 推入 `pending_q`，并置 `pending_mask[dest]=1`，同时把这条指令在流水线里“消掉”：令 `id_ex_d.valid=0` 或者至少 `id_ex_d.reg_write=0`、`id_ex_d.mdu_cmd=MDUCMD_NONE`，使它不会在 EX 变成 `ex_is_mdu=1`，从而不会触发现有的 `mdu_hazard`。这样 L3/L4 就不会卡在 EX（因为它们

根本不会以“MDU 指令形态”进入 EX），流水可以继续把后续指令推到 EX。如果 pending 队列满了：就退化成老行为——让 mflo/mfhi 正常进入 EX，然后按 `mdu_hazard` 卡住（这样不会破坏正确性）。

3. L6：消费者在 EX 触发 stall

   当 L6 `addu $3,$3,$4` 进入 EX 时，如果它要读的源寄存器中，有任意一个在 `pending_mask` 里为 1（比如 `$3` 或 `$4`），就说明它需要的值“还没真正写回”，此时 **让它在 EX 卡住。**目前已经有“卡住 EX 并且让 MEM/WB 继续走”的模式：MDU stall 用的是 `stall_idex + bubble_exmem` 和 EX/MEM 插泡，可以复用同一套控制输出：`stall_pc=1, stall_ifid=1, stall_idex=1, bubble_exmem=1`，这会把 L6 固定在 EX，前端不再推进，同时避免 EX/MEM 重复执行副作用。为了满足“恢复前流水线上只有一条指令”，在进入这个“等待 replay”模式时，把 IF/ID 也清空

4. MDU 结束后：把 pending 写回“重新注入 WB”，再放行 L6

   当 `mdu_busy` 变 0 时，MDU 的 HI/LO 已经更新完成（MDU 在 `remainingCycleCount==1` 时写 hi/lo，下一拍可读；`dataRead` 是根据 `operation` 组合读 hi/lo）。此时进入一个“WB replay 状态机”，做两件事：

   - 逐条把 pending 写回塞进 MEM/WB（就像它们是正常指令提交一样）：正常写回通路是 `wb_we/wb_waddr/wb_wdata` 来自 `mem_wb_q`，所以 replay 时可以“覆盖 `mem_wb_d` / `mem_wb_q` 的输入来源”：连续两个周期构造合成的 `mem_wb_d`：
     - 第 1 个周期：写 `$3`（mflo），`wb_data = LO`
     - 第 2 个周期：写 `$4`（mfhi），`wb_data = HI`
       每注入一条，就清对应的 `pending_mask[dest]=0` 并弹出队列项
   - 在 replay 完成前继续卡住 L6，直到 `$3/$4` 的 replay WB 都完成，再解除 `stall_idex`，让 L6 在 EX 继续执行

3. （附加题）如果你有两个乘法器，在什么文件作出什么修改能让你的 CPU 同时进行两个乘法，让上面代码在 L8-L13 不阻塞，在 L14 阻塞？

   - 在 `PipelineCPU.sv` 新增一个“HI/LO 版本号/顺序号” `hilo_seq`（每发射一次 mult/div 就 `hilo_seq++`，并把这个 seq 入 FIFO），当解码到 `mflo rd` / `mfhi rd` 时，不再强制它变成 `ex_is_mdu=1` 的指令去碰 MDU；而是在 ID（或 ID→EX 生成阶段）登记一条 **pending 写回**：记录：`pending.valid=1, pending.dest=rd, pending.is_lo/is_hi, pending.seq=hilo_seq(最近一次mult的seq)`；同时把这条 `mflo/mfhi` 在流水线里“消掉副作用”（例如让其 `valid=0` 或 `reg_write=0`），这样它不会触发当前 HazardUnit 的 `ex_is_mdu && mdu_busy` 卡 EX 机制。当某个 `seq` 在 FIFO 头被提交，`arch_hi/arch_lo` 更新后，如果有 pending 写回在等这个 seq，就把它**重新注入 WB**
   - 在 `HazardUnit.sv` 增加一个新的 hazard 类型（“pending-reg hazard”），给 HazardUnit 增加输入：`ex_uses_rs/ex_uses_rt + ex_rs/ex_rt`，维护一个 `pending_mask[31:0]`（哪些 GPR 还没被 replay 写回），当 EX 阶段指令要读的源寄存器命中 `pending_mask` 时，输出类似 MDU stall 的控制：`stall_pc=1, stall_ifid=1, stall_idex=1, bubble_exmem=1`，把消费者卡在 EX，直到对应 pending 写回完成