Lab1: Pipeline CPU

朱熙哲

3220103361

2023 年 11 月 7 日

目录

1 数据通路 **3**

2 模块设计 **3**

2.1 PC 3

2.2 BRAM 4

2.3 阶段寄存器 4

2.4 Race Controller 6

3 仿真与板验 **8**

3.1 仿真测试 8

3.2 上板验证 9

4 思考题 **10**

5 Bonus **13**

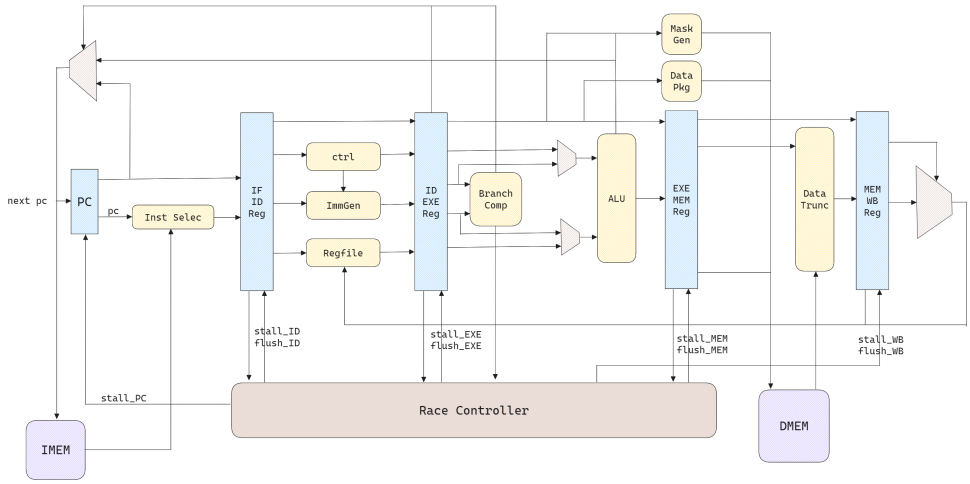
5.1 Forwarding 13

5.1.1 数据通路 13

5.1.2 模块设计 13

5.2 Predict not token 14
5.3 仿真 15

1 数据通路



相比原 SCPU，PipelineCPU 增加了 Race Controller 模块以实现 stall 机制，并且增添了阶段寄存器，以及修改了部分接线，来完成流水线功能并解决冲突问题。

以下仅介绍发生了变化的模块。

2 模块设计

2.1 PC

```
always@(posedge clk or posedge rst) begin
    if(rst) begin
        pc <= 0;
    end
    else begin
        if (stall)begin
            pc <= pc;
        end
        else begin
```

```
            pc <= pc_in;
        end
    end
end
assign npc = rst ? 0 : (stall ? pc : (pc + 4));
assign valid_if = 1;
```

PC 寄存器中主要变化在于，增加了 stall 信号，当 stall 信号有效时，下一拍 pc 维持原值不变，否则依然被赋值为 pc 选择器传入的 next pc。对于 npc，为了配合内存模块的改变，不能直接使用 pc+4，而是当 rst 或者 stall 信号有效时，应也应保持为原 pc，当然这也符合逻辑。对于 valid_if，由于 PC 寄存器并不会被排空，所以总置为 1。

2.2 BRAM

Block RAM 最大的变化在于，由于延迟一拍的内存读写，因此不能直接将 pc 或者 data 传入 MEM 模块。由数据通路图中可见，直接将 next pc 以及 ALU 的结果传入 MEM，we_mem 也直接从 ID/EXE 阶段寄存器传入。

2.3 阶段寄存器

以下是 IF/ID Reg：

```
always @(posedge clk) begin
    if (flush || rst) begin  //重置
        pc_id <= 0;
        npc_id <= 0;
        inst_id <= 0;

        rd_addr_id <= 0;
```

```
            rs1_addr_id <= 0;
            rs2_addr_id <= 0;
            valid_id <= 0;
        end
        else begin      //传递所需信号和数据
            if (stall == 1) begin
                pc_id <= pc_id;
                npc_id <= npc_id;
                inst_id <= inst_id;

                rd_addr_id <= rd_addr_id;
                rs1_addr_id <= rs1_addr_id;
                rs2_addr_id <= rs2_addr_id;
                valid_id <= valid_id;
            end
            else begin
                pc_id <= pc_if;
                npc_id <= npc_if;
                inst_id <= inst_if;

                rd_addr_id <= inst_if[11:7];
                rs1_addr_id <= inst_if[19:15];
                rs2_addr_id <= inst_if[24:20];
                valid_id <= valid_if;
            end
        end
    end
```

当 flush 信号或者 rst 信号传入时，下一拍所有传出信号置 0，也就是传出 nop 指令。当 stall 信号有效时，保持原值。否则将 IF 阶段信号接入。

注意此处没有使用 posedge rst 作为异步复位信号，因为这些寄存器接受 flush 与 stall 等多个复位信号控制，可能会生成 latch，或者信号波动导致异常排空。实际在上板过程中也确实发生，stall 信号传入后，多条指令被排空而没有执行。

后面的 ID/EXE、EXE/MEM、MEM/WB 阶段寄存器使用完全一致的设计，不一一介绍。只需注意各阶段间信号的传递即可。

2.4 Race Controller

```
module RaceController(
    input [4:0]rs1_addr_id,  input [4:0]rs2_addr_id,
    input use_rs1_id,        input use_rs2_id,
    input [4:0]rd_addr_exe,  input [4:0]rd_addr_mem,
    input we_reg_exe,        input we_reg_mem,
    input npc_sel_id,        input npc_sel_exe,
    input [3:0]br_taken,
    output wire stall_PC,    output wire stall_IFID,
    output wire stall_IDEXE, output wire stall_EXEMEM,
    output wire stall_MEMWB, output wire flush_IFID,
    output wire flush_IDEXE, output wire flush_EXEMEM,
    output wire flush_MEMWB
);
    assign stall_IFID = (rs1_addr_id == rd_addr_exe &\
                         we_reg_exe & rs1_addr_id != 0 & use_rs1_id) |\
                        (rs2_addr_id == rd_addr_exe &\
                         we_reg_exe & rs2_addr_id != 0 & use_rs2_id) |\
                        (rs1_addr_id == rd_addr_mem &\
                         we_reg_mem & rs1_addr_id != 0 & use_rs1_id) |\
                        (rs2_addr_id == rd_addr_mem &\
```

```
                        we_reg_mem & rs2_addr_id != 0 & use_rs2_id);
    assign stall_PC = stall_IFID | npc_sel_id;
    assign stall_IDEXE = 0;
    assign stall_EXEMEM = 0;
    assign stall_MEMWB = 0;
    assign flush_IFID = (stall_PC & ~stall_IFID) | br_taken[3] |\
                       (br_taken[2:0] == 0 & npc_sel_exe);
    assign flush_IDEXE = stall_IFID & ~stall_IDEXE;
    assign flush_EXEMEM = stall_IDEXE & ~stall_EXEMEM;
    assign flush_MEMWB = 0;
endmodule
```

Race Controller 用于发出 stall 与 flush 信号以解决冲突。

- 数据冲突
  不关心具体的指令类别，仅关注 ID 阶段的 rs1,rs2 与 EXE 和 MEM 阶段的 rd 是否有竞争即可。比如 ID 阶段的 rs1 与 EXE 阶段的 rd 相同，如果 rd 会被写回，并且 rs1 确实会被使用，则需发出 stall 信号给 IF/ID 阶段寄存器。
  use_rs1_id 与 use_rs2_id 是从 control 额外接出的信号，表示该指令类型使用了该寄存器。
  ID 阶段寄存器 stall 后，之前的 PC 寄存器也一并 stall。
  如果发生 stall，则从左往右第一个尚未 stall 的阶段寄存器下一拍应该排空。flush 信号检测上一阶段 stall 并且当前阶段没有 stall，则置 1。
  注意由于在 regfile 中使用下降沿写回，则不需考虑 WB 阶段的数据冲突。

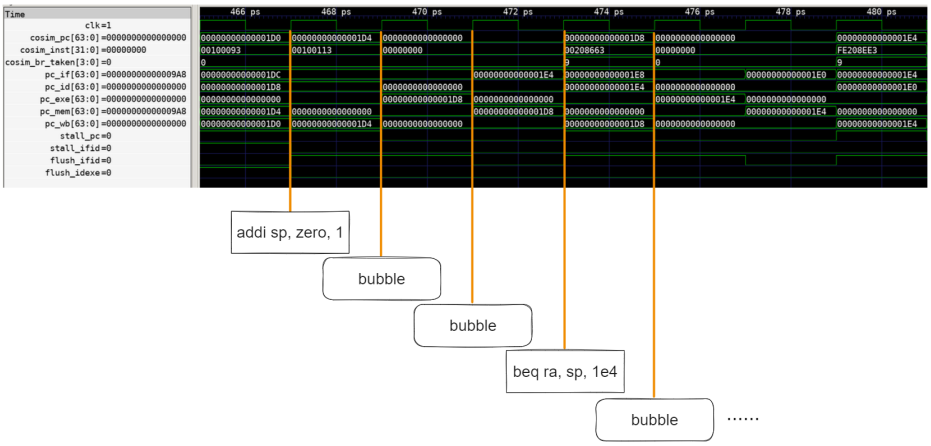
- 控制冲突
  npc_sel_id 表示该 ID 阶段的指令是否为跳转指令，若是，则将 PC 寄存器 stall，等到 EXE 阶段计算完成跳转结果。

br_taken[3] 表示 Btype 的跳转，若跳转需要额外排空已经从 PC 进入 IF/ID 的下一条指令。br_taken[2:0] == 0 & npc_sel _exe 表示 jal 或 jalr 指令跳转，此时也许排空。

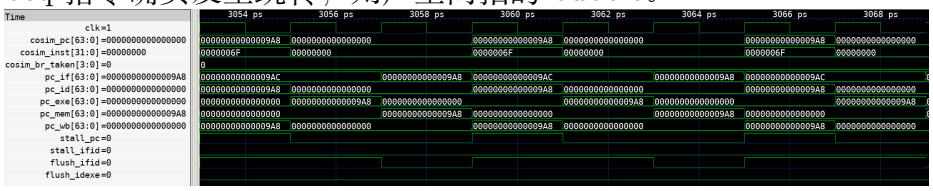
综上可见，若跳转，则损失两拍，否则损失一拍。

3 仿真与板验

3.1 仿真测试

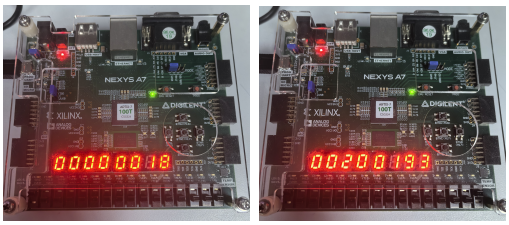


可以看到 addi 与 beq 指令间发生数据竞争，则 stall 两个周期直至写回；beq 指令确实发生跳转，则产生两拍的 bubble。

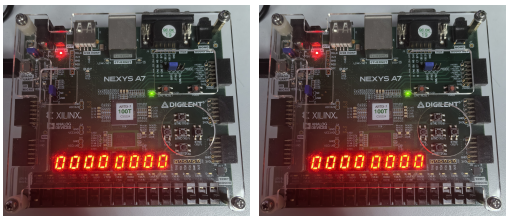


最终在 9a8 循环，每次 jal 都会产生两个 bubble。

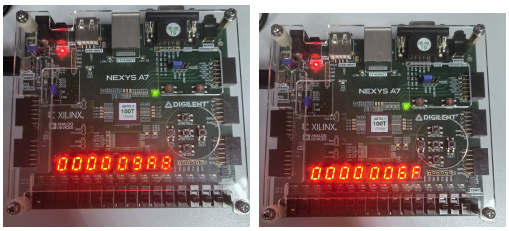
3.2 上板验证



(a) addi gp,zero,2



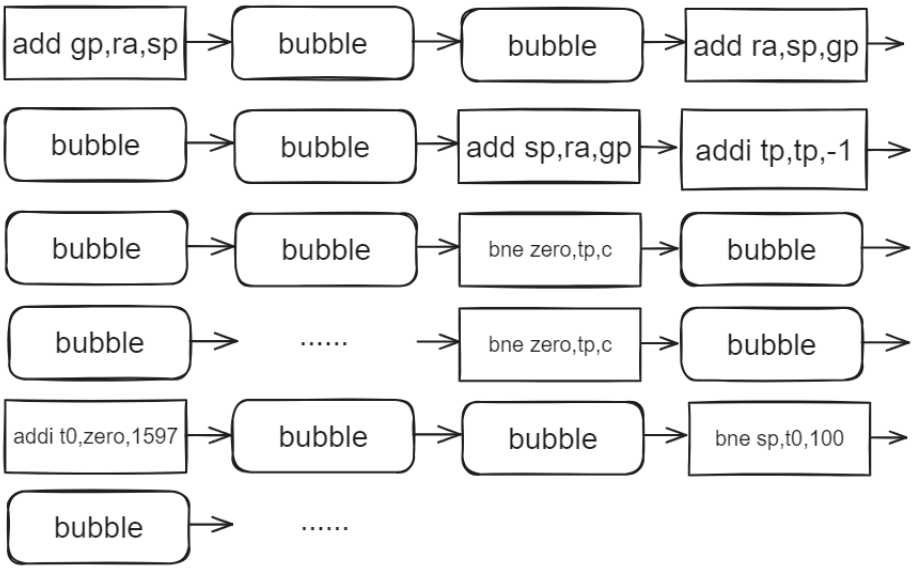
(b) bubble



(c) jal zero,9a8 <pass>

4 思考题

1. 由图可见，每个循环要引入 8 个 bubble，总共 5 个循环，最后一个循环是 7 个 bubble，再加上离开 fibonacii 的 5 拍，总共需要 69 个时钟周期，总运行代码条数为 27，则 $fibonacci_CPI_{PCPU} = \frac{32}{9}$。显然，单周期 CPU 的 CPI 就为 1。



PCPU fibonacci

2. 第二个 addi 与 lb 间会产生 1 个 bubble，之后每个 lb 和其下一个 sb 之间会产生 2 个 bubble。不过可以直接分析波形图得到总周期数为 50，总运行代码数为 26，则 $test2_CPI_{PCPU} = \frac{25}{13}$。单周期 CPU 的 CPI 仍为 1。

3. - 数据冲突
    - RAW：在指令 1 写入前，指令 2 先读取了数据，这是本 lab 唯一涉及的数据冲突类型，也是最常见的数据冲突类型。即 EXE/MEM 阶段的 rd 与 ID 阶段的 rs1/rs2 编号相同，并且

rd 会被写回，而 rs1/rs2 会被使用，且不是 zero 寄存器。6 种指令间均可能发生该冲突。

  - WAW：在指令 1 写入前，指令 2 先写入。发生条件为①流水线不止一个段可以写回寄存器组。②指令发生优化重排序。
  - WAR：在指令 1 读取之前，指令 2 先写入了数据。当指令被重新排序乱序执行时，可能发生该冲突。

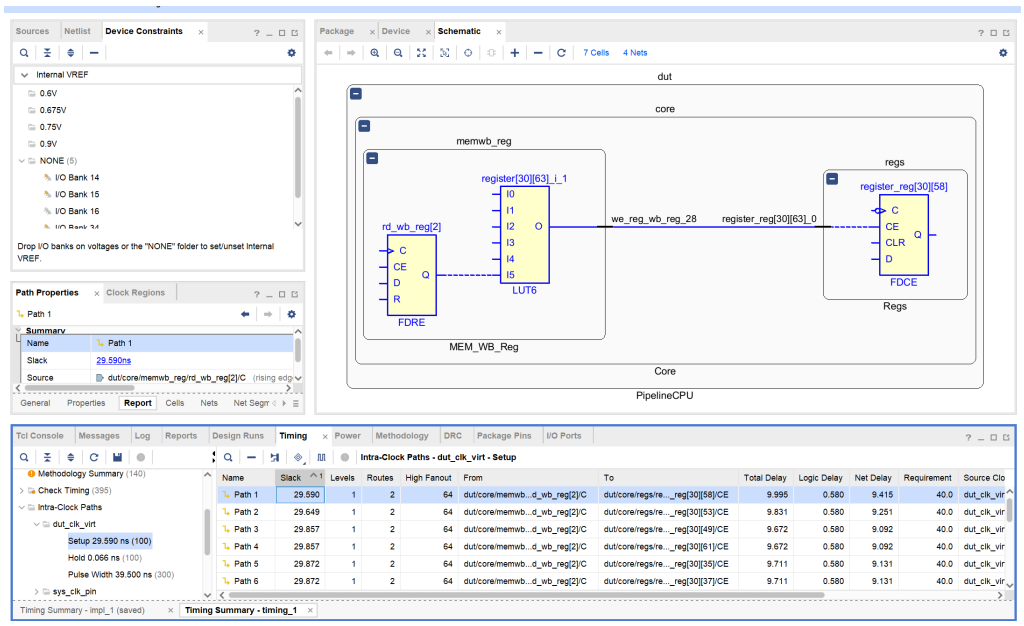
- 控制冲突
  - 发生跳转：jal 与 jalr 指令，以及有跳转的 Btype 指令。根据本 lab 设计，会产生两个 bubble。
  - 不发生跳转：无跳转的 Btype 指令。根据本 lab 设计，会产生一个 bubble。

4. 首先，由于采用下降沿写回，WB 阶段与 ID 阶段寄存器冲突不会导致异常读取；其次，若 EX 和 MEM 段寄存器均与 ID 段寄存器发生冲突，依然 stall 两拍，等到 EX 段的指令进行到 WB 段后结束 stall，流水线继续运行。

5. 优先解决数据冲突，否则可能导致跳转异常。即当 ID 阶段 Btype 指令与 EXE 阶段或 MEM 阶段指令发生数据冲突，先 stall 直至该指令走到 WB 阶段，再处理接下来的控制冲突。

6. 对于通用寄存器组 Regfiles 可以。变化在于，读取寄存器时，信号不通过 IF/ID 阶段寄存器而直接传给 Regfiles；写入时也不经过 MEM/WB 阶段寄存器，而直接从 EXE/MEM 阶段寄存器经过 WB_MUX 传给 Regfiles；并且要使用寄存器来缓存读取内容。
   对于流水线中间寄存器组，由于端口太多，不可使用 BRAM 实现。

7. 由图可见，关键路径是写回阶段 rd 从 MEM/WB 阶段寄存器组传输到 Regs 寄存器组的过程。最大延迟为 9.995ns，裕量为 29.590ns，因此还可以把时钟频率调快，从 12.5MHz 调至 25MHz 以上。

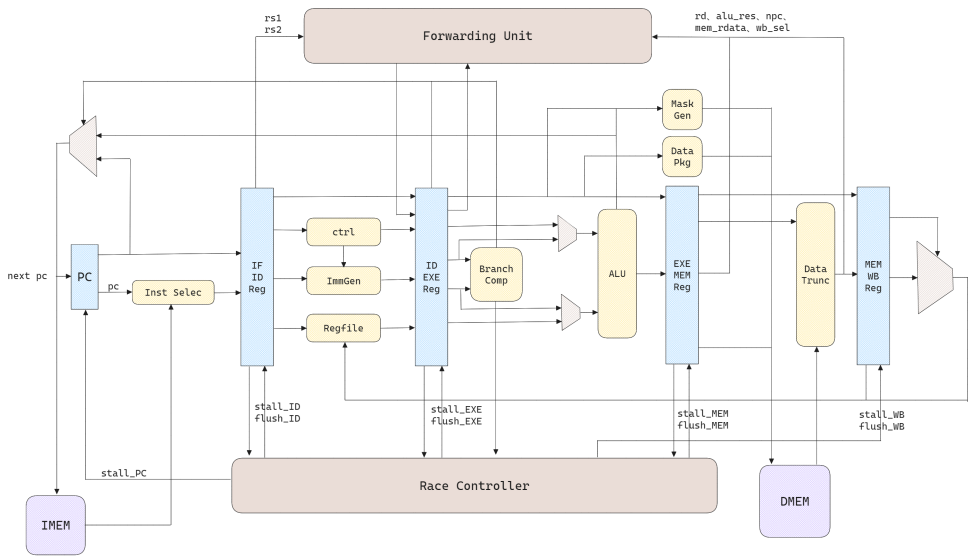


Critical Path

5 Bonus

5.1 Forwarding

5.1.1 数据通路



增添了 Forwarding Unit 模块来前递数据，该模块使用组合逻辑。

5.1.2 模块设计

```
always@(*) begin
    if (rs1_addr_id == rd_addr_exe && \
        rs1_addr_id != 0 && we_reg_exe == 1) begin
        case(wb_sel_exe)
            2'b00: rs1_data_idexe = 0;
            2'b01: rs1_data_idexe = alu_res_exe;
            2'b11: rs1_data_idexe = npc_exe;
            default: rs1_data_idexe = 0;
        endcase
```

```
    end
    else if (rs1_addr_id == rd_addr_mem && \
             rs1_addr_id != 0 && we_reg_mem == 1) begin
        case(wb_sel_mem)
            2'b00: rs1_data_idexe = 0;
            2'b01: rs1_data_idexe = alu_res_mem;
            2'b10: rs1_data_idexe = dmem_mem;
            2'b11: rs1_data_idexe = npc_mem;
            default: rs1_data_idexe = 0;
        endcase
    end
    else begin
        rs1_data_idexe = rs1_data_id;
    end
end
```

为避免代码过长，仅展示 rs1 的判断，而 rs2 的前递判断完全一致。

先判断当 ID 阶段的 rs1/rs2 与 EXE 阶段的 rd 发生冲突，根据 wb_sel 选择数据写回到 ID/EXE 阶段寄存器。再判断 MEM 阶段的冲突并写回。**注意**当 EXE 阶段需要写回 mem_rdata 时，RaceController 发出 stall 信号，等到指令进行到 MEM 阶段，得到了数据再前递。

由于选择写回 ID/EXE 寄存器，当依赖于 MEM 的数据而发生 stall 时，WB 阶段自然会进行下降沿写回而将数据传递回来，不需考虑暂存。

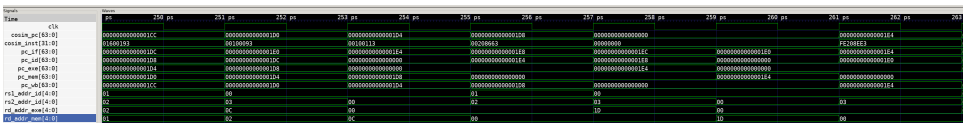
5.2 Predict not token

```
assign stall_IFID = (is_load_exe == 1 & rs1_addr_id == rd_addr_exe & \
                     we_reg_exe & rs1_addr_id != 0) | \
                    (is_load_exe == 1 & rs2_addr_id == rd_addr_exe & \
                    we_reg_exe & rs2_addr_id != 0);
```

```
assign stall_PC = stall_IFID;
assign stall_IDEXE = 0;
assign stall_EXEMEM = 0;
assign stall_MEMWB = 0;
assign flush_IFID = (stall_PC & ~stall_IFID) | br_taken[3] | \
                    (br_taken[2:0] == 0 & npc_sel_exe);
assign flush_IDEXE = (stall_IFID & ~stall_IDEXE) | br_taken[3] | \
                     (br_taken[2:0] == 0 & npc_sel_exe);
assign flush_EXEMEM = stall_IDEXE & ~stall_EXEMEM;
assign flush_MEMWB = 0;
```

在引入 forwarding 后对 RaceController 作出修改的基础上，去掉 npc_sel_id 对 stall_PC 的控制，即总预测不跳转，而如果发生跳转，则需要排空 IF/ID 与 ID/EXE 两个阶段寄存器，将载入流水线的两条错误指令排空。

5.3 仿真



可以发现除了发生跳转的指令后产生两个 bubble，其余 Itype 指令间不再有 bubble。

上板验证留待 lab2 一并进行。