

Lab2: Forwarding and AXI4-lite

朱熙哲

3220103361

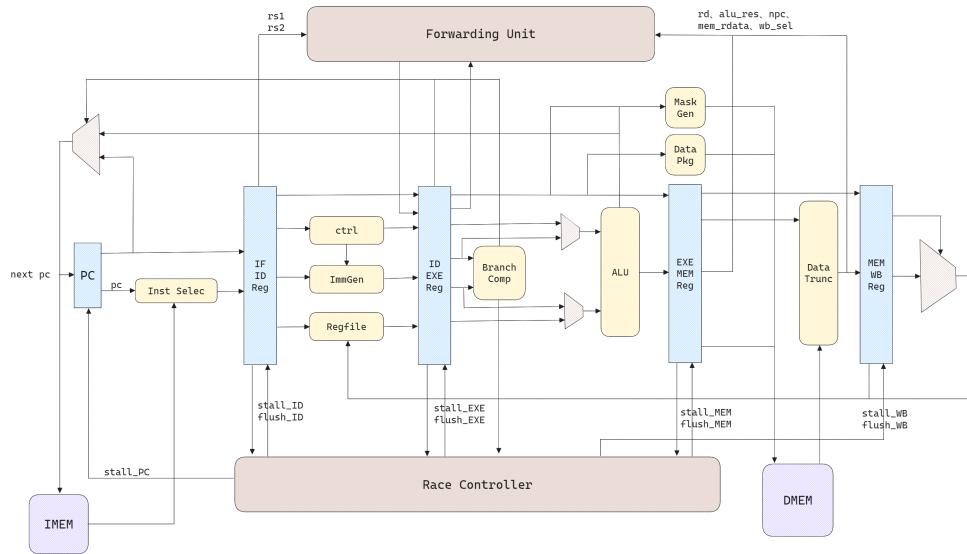
2024 年 1 月 8 日

目录

1	Forwarding 的实现	2
1.1	数据通路	2
1.2	仿真测试	3
1.3	上板验证	3
2	AXI4-lite 总线	5
2.1	模块设计	5
2.1.1	PC	5
2.1.2	RaceController	5
2.1.3	CoreToMem FSM	6
2.2	仿真测试	10
2.3	上板验证	10
3	思考题	12

1 Forwarding 的实现

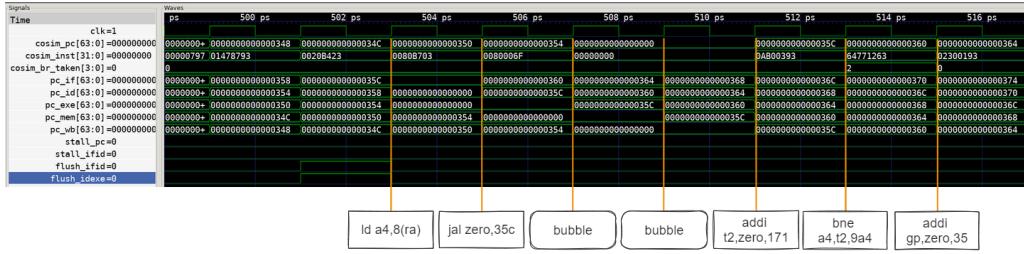
1.1 数据通路



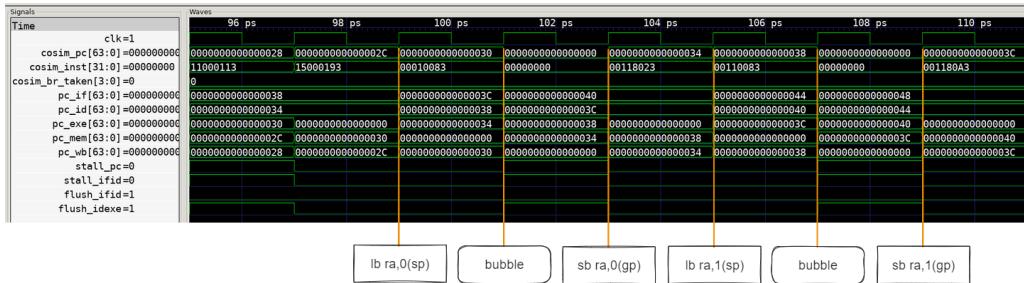
模块设计已在 lab1 的 bonus 中给出。需要说明的是，我的 forwarding 设计是将 EXE 与 MEM 阶段的 rd 经判断传回给 ID/EXE 寄存器，这是为了时序优化，减去了 WB 阶段的 rd 前递的判断。代价是，如果时间裕量不足，来不及在下降沿写回，则依然会导致错误读取。

以下是添加上 predict-not-token 后的仿真测试与上板验证。

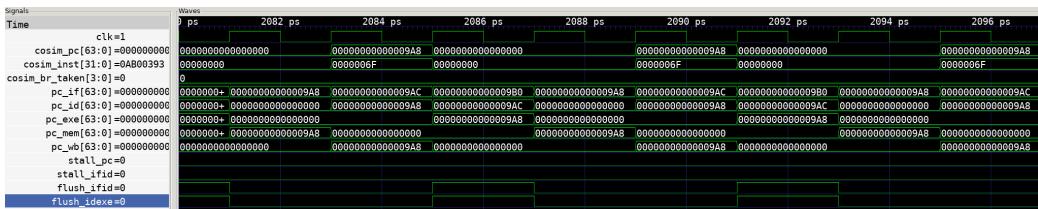
1.2 仿真测试



可以看见 jal 跳转指令后产生了两个 bubble；在第一个 addi 与 bne 之间，由于实现了 forwarding，因此不产生 bubble；bne 后面由于预测不跳转，并且确实没有跳转，则不产生 bubble。

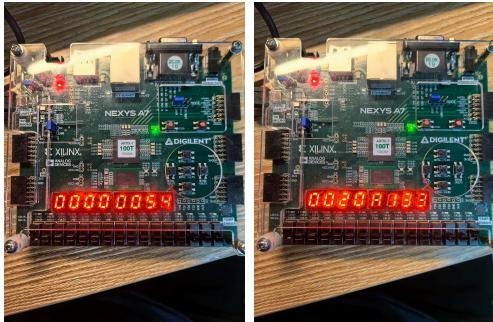


当 EXE 阶段为 load 指令，且与 ID 阶段的指令确实发生数据冲突时，需要 stall 一拍，等到 MEM 阶段取出 DMEM 数据再进行前递，因此会产生一个 bubble。

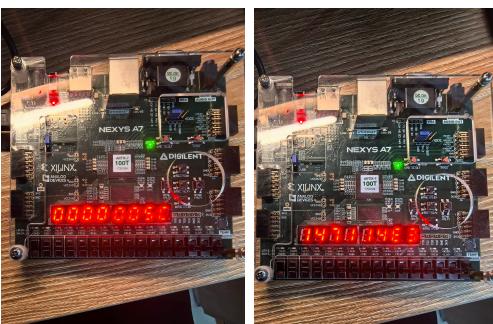


最终在 9a8 循环。

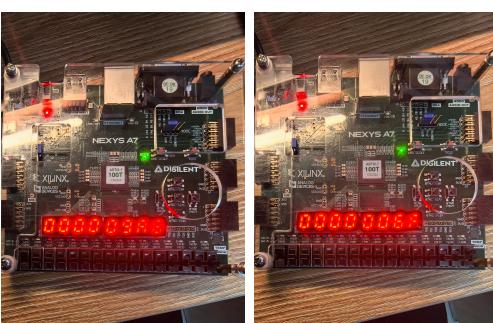
1.3 上板验证



(a) slt sp,ra,sp



(b) bne sp,t2,9a4 <fail>



(c) jal zero,9a8 <pass>

2 AXI4-lite 总线

core 的数据通路与前已实现的一致，只是之前传给 BRAM 的数据，此时将传给 Core2Mem_FSM。

2.1 模块设计

2.1.1 PC

关于 PC，需要额外增加 if_request 的输出：

```
assign if_request = ~rst & ~npc_sel_id & ~npc_sel_exe;
```

当处于 rst 时，不发送请求。由于请求读取指令需要花费大量的时间，相比起成功预测不跳转而节省的拍数而言，预测失败的损失是更加严重的，并且跳转地址还需要额外的存储单元来保存以免在等待请求时流失掉。因此，我选择等到计算完跳转地址再发送请求取指。

又由于 npc_sel_id 会导致 PC 的 stall，因此选择等到跳转指令再走一拍，再发送请求，使得 PC 运行一拍，让 next_pc 能够进入到 PC 中，否则 PC 会持续 stall 不接收跳转地址。

2.1.2 RaceController

```
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) | \
                      stall_IDEXE;

assign stall_PC = stall_IFID | if_stall | npc_sel_id | \
                  (npc_sel_exe & ~br_taken[3] & \
                  br_taken[2:0] != 3'b000);

assign stall_IDEXE = stall_EXEMEM;
```

```

assign stall_EXEMEM = mem_stall;
assign stall_MEMWB = 0;
assign flush_IFID = (stall_PC & ~stall_IFID) | npc_sel_exe;
assign flush_IDEXE = (stall_IFID & ~stall_IDEXE) | npc_sel_exe;
assign flush_EXEMEM = stall_IDEXE & ~stall_EXEMEM;
assign flush_MEMWB = stall_EXEMEM & ~stall_MEMWB;

```

由于取内存的耗时，基本上关于 load 的数据冲突的 stall_IFID 也就没有用武之地了。对于 PC，if_stall 会导致 stall，ID 阶段的跳转指令也会使其 stall，需要注意的是如果是未发生跳转的 Btype 指令，会导致 next_pc 载入 pc+4，然而由于先前的等待策略，当前 pc 这条指令尚未发送请求取值，因此额外 stall 一拍使得 Btype 指令返回的 npc 流失掉，然后 next_pc 载入当前 pc，发送请求取指。

由于 PC 需要运行一拍来载入 next_pc，因此需要对 ID 和 EXE 进行 flush 来排空进入到下一阶段的一条只有 pc 的空指令。

2.1.3 CoreToMem FSM

使用三段式来书写这个有限状态机。

一 状态转移

```

parameter IDLE = 3'b100,
           INST = 3'b010,
           DATA = 3'b001;
reg [2:0] state, next_state;;
reg [63:0] pc_reg;
reg if_stall_reg, mem_stall_reg;

always @(posedge clk or negedge rstn) begin
  if (!rstn) begin
    state = IDLE;
    pc_reg = 64'h0;
  end
  else begin
    state = next_state;
    pc_reg = pc_reg + 1;
  end
end

```

```

        state <= IDLE;
    end
    else begin
        state <= next_state;
    end
end

```

首先将状态用独热码编码，再定义状态与寄存器，然后使用非阻塞赋值描述状态转移。

二 状态判断与 stall 信号释放

```

case(state)
    IDLE: begin
        next_state = wen_cpu | ren_cpu ? DATA :
                                if_request      ? INST :
                                IDLE;
        if_stall_reg = if_request;
        mem_stall_reg = wen_cpu | ren_cpu;
    end
    DATA: begin
        next_state = valid_mem ? IDLE : DATA;
        if_stall_reg = if_request;
        mem_stall_reg = ~valid_mem;
    end
    INST: begin
        next_state = valid_mem ? IDLE : INST;
        if_stall_reg = ~valid_mem;
        mem_stall_reg = wen_cpu | ren_cpu;
    end
    default: begin

```

```

    next_state = IDLE;
    if_stall_reg = 0;
    mem_stall_reg = 0;
end
endcase //省略 always 块细节，仅保留 case 判断

```

使用组合逻辑，按照给出的有限状态机表格来判断下一个状态，需要注意的是 if_stall 信号和 mem_stall 信号也需要在该 always 块中判断，因此需要两个 stall 寄存器。如果直接使用输入信号来 assign 赋值，一方面代码复杂，其次有可能在 core 和 FSM 间产生组合回路，导致锁存器出现。

注意在 DATA 状态和 INST 状态也需要接受 core 发来的 if_request 信号和 ren/wen 信号并返回 stall 信号。持续 stall 直至从总线传回 valid 信号并选择关闭 mem 或者 if 的 stall 信号。

三 状态输出

```

case(next_state)
DATA:begin
if(state == DATA)begin
    address_mem <= address_mem;
    ren_mem <= ren_mem;
    wen_mem <= wen_mem;
    wmask_mem <= wmask_mem;
    wdata_mem <= wdata_mem; //也可以不保存这些数据
end
else begin
    address_mem <= address_cpu;
    ren_mem <= ren_cpu;
    wen_mem <= wen_cpu;
    wmask_mem <= wmask_cpu;
end

```

```

        wdata_mem <= wdata_cpu;
    end
end

INST:begin
    if(state == INST)begin
        pc_reg <= pc_reg;
        address_mem <= address_mem;
        ren_mem <= ren_mem;
    end
    else begin
        pc_reg <= pc;
        address_mem <= pc;
        ren_mem <= if_request;
    end
end

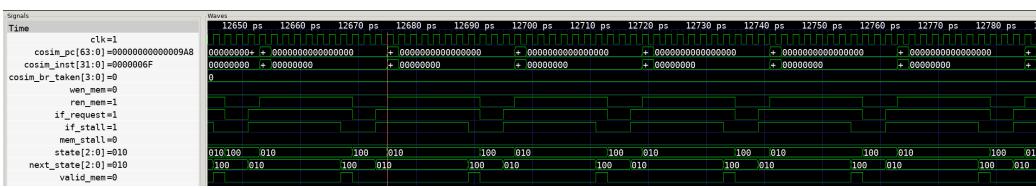
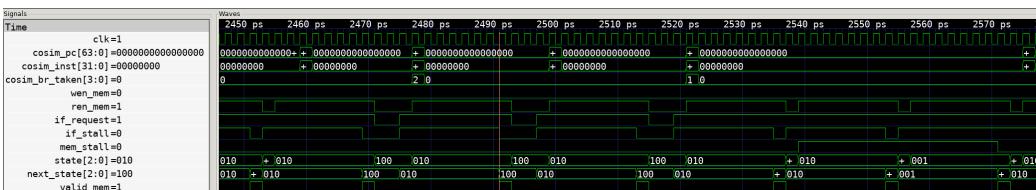
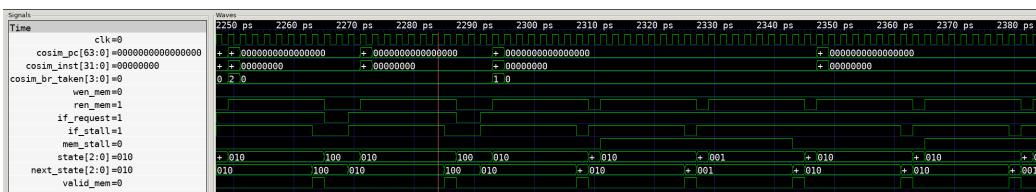
IDLE:begin
    pc_reg <= 0;
    address_mem <= 0;
    ren_mem <= 0;
    wen_mem <= 0;
    wmask_mem <= 0;
    wdata_mem <= 0;
end
endcase //省略 always 块细节，仅保留 case 输出判断，详见源码
assign if_stall = if_stall_reg;
assign mem_stall = mem_stall_reg;
assign rdata_cpu = rdata_mem;
assign inst[31:0] = rdata_cpu[pc_reg[2:0]*8 +:32];

```

发出给总线的数据使用非阻塞赋值。主要需要注意的是，将 pc 保存

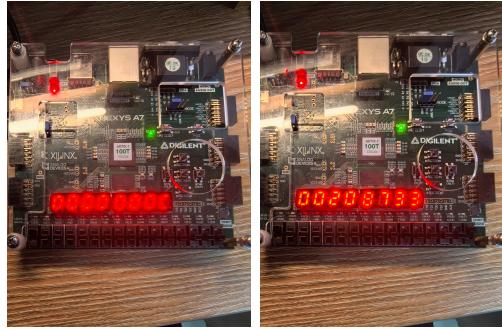
在寄存器中，当 valid 返回时，pc_reg 依然保存着上一条请求的 pc，因此可以确定上一条指令，否则传入的 pc 已然变为 next_pc 了，导致取指错误。

2.2 仿真测试

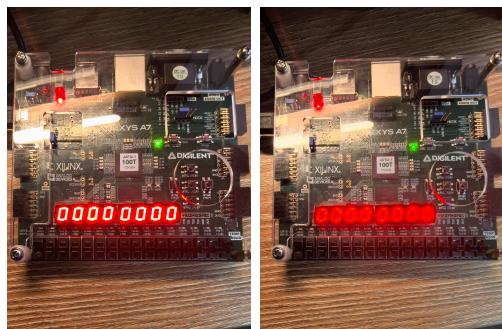


可以发现主要的时间开销都在读取内存中。

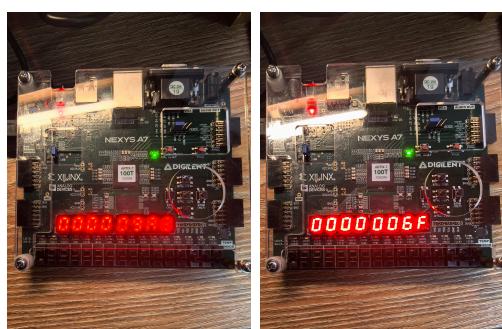
2.3 上板验证



(a) add a4,ra,sp



(b) bubble



(c) jal zero,9a8 <pass>

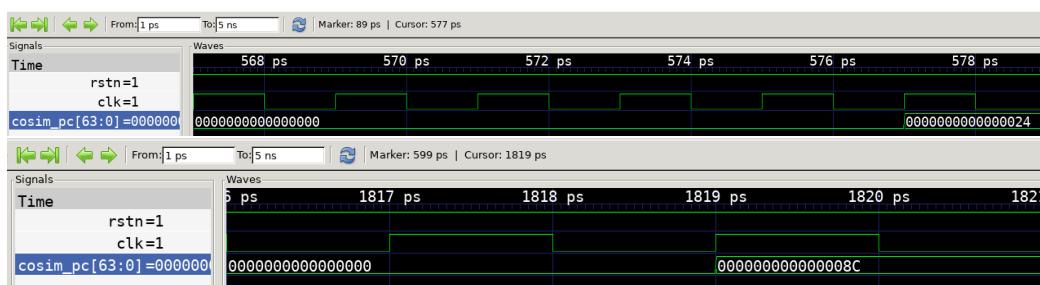
3 思考题

1. stall 机制依然需要。当 EXE 阶段为 load 指令，并且与 ID 阶段指令发生数据冲突时，此时尚未取到内存中的数据，无法前递，必须 stall 一拍。当然也可以通过优化指令顺序来避免该情况发生。另外在引入总线后必然需要 stall 机制。
2. forwarding 机制的加入会导致多路选择器的增加，进而增加电路复杂度，并且增加了从 ID 阶段向 EXE 阶段传递 rs1 与 rs2 的信号延迟。相比单纯 stall,forwarding 的缺点是无法解决 load-use 冲突，并且必须依靠 predict(-not)-token 与 flush 来解决控制冲突。
3. ①对于所有 use-use 冲突，停顿的拍数从 1 或 2 拍均减为 0 拍。②对于 use-store 冲突，停顿的拍数从 1 或 2 拍均减为 0 拍。③对于 load-use 冲突，停顿拍数从 1 或 2 拍减为 0 或 1 拍。
4. 通过波形计算 syn.asm 中 fibonacci 阶段和 test2 阶段的 cpi。

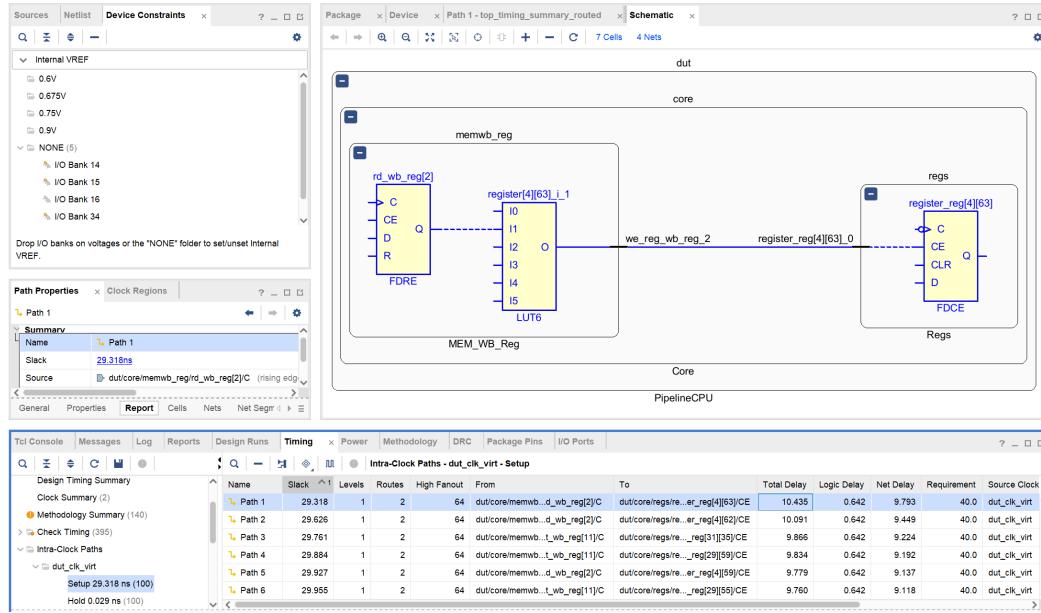
$$CPI_{fibonacci} = \frac{244}{27} \approx 9.0, CPI_{test2} = \frac{610}{26} \approx 23.5$$

可以发现取指和存取内存是制约 cpi 的主要因素，尤其是大量的 load 和 store 指令会极大地增加时间消耗，导致 cpi 相当可怜。

可能的提升方法有：①提高总线的传输效率，减少传输时延。②增加 RAM 数量，将 IMEM 与 DMEM 分离，分别请求数据，避免结构竞争导致 stall。



5. 对于仅添加 Forwarding 未引入总线的电路，可以看见关键路径依然是写回阶段 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 10.435ns，裕量为 29.318ns。发现相比纯 stall 实现，延迟有所增加。



Critical Path