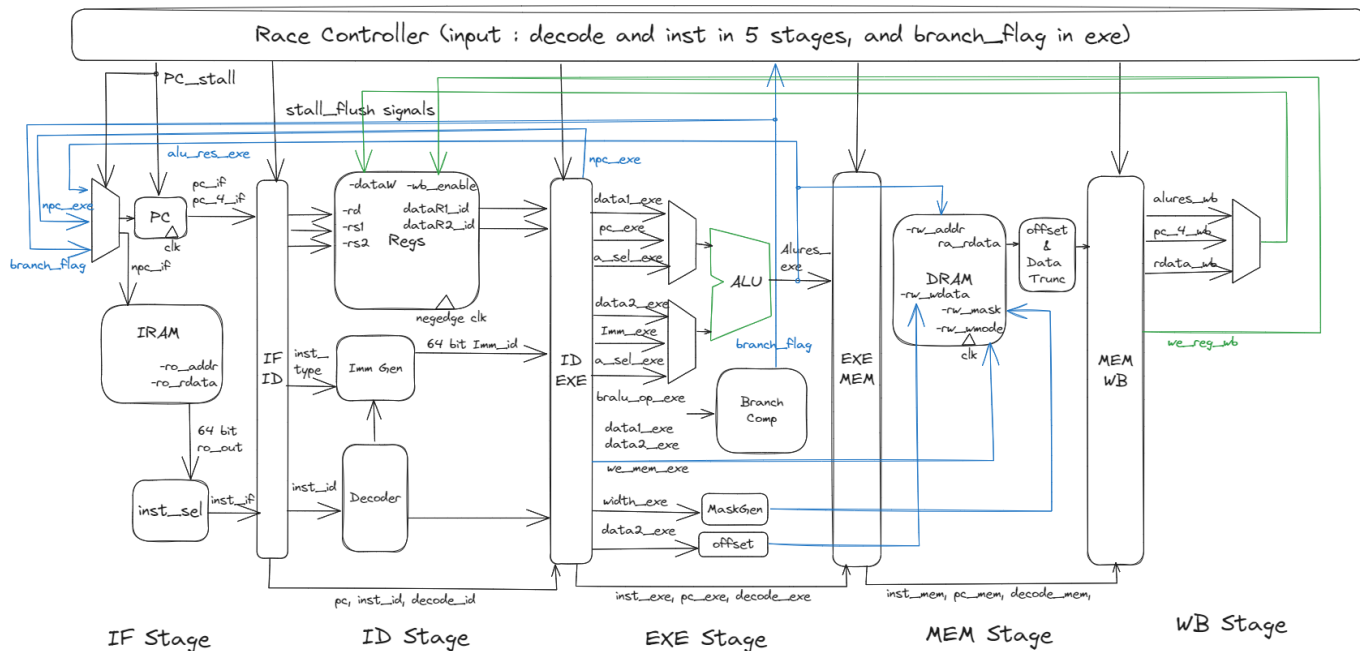


# 1. 理解 PipelineCPU

## 1.1 绘制数据通路

便于自己理解逻辑和接线，绘制了分成了五个阶段，添加了中间寄存器的 CPU 数据通路。引入 BRAM 解决。（Excalidraw 好文明）



## 1.2 决定 RaceController 处理冲突的方法

### RaceController 控制逻辑

最开始采用了助教 PPT 上的方法处理冲突，遇到跳转命令就 stall，数据冲突优先于控制冲突处理。后来修改了 RaceController 的逻辑，确定跳转再 stall，控制冲突优先于数据冲突处理。

### 1.2.1 最开始设想的逻辑

#### Stall 处理 Data Hazard:

如果 ID 阶段发现 EXE 阶段和 MEM 阶段写回的 rd 和自己的 rs1、rs2 是一致的，那么此时 regfile 内的数据是过时的数据，不可以使用，必须等待 EXE 和 MEM 阶段写回才可以，所以一旦检测到如上情况 IF 和 ID 需要 stall。

伪代码如下：

```
if ((ID.rd == ID.rs1 && EXE.reg_we) || (EXE.rd == ID.rs2 && EXE.reg_we)) {
    IF.stall = 1;
    ID.stall = 1;
    EXE.flush = 1;
}
if ((ID.rd == ID.rs1 && MEM.reg_we) || (MEM.rd == ID.rs2 && MEM.reg_we)) {
    IF.stall = 1;
    ID.stall = 1;
    EXE.flush = 1;
    MEM.flush = 1;
}
```

### Stall 处理 Control Hazard:

如果一条branch指令或者j指令还没有在EXE更新PC，那么ID、IF得到的PC可能就是错误的，那么得到的内容也是无效的，因此要阻止在b、j指令执行完之前的IF、ID的数据读入。如果我IF的指令是b、j指令，那么在它EXE执行完之前，就不可以加载的指令。

```
if (ID.is_btype & ID.is_jtype | EXE.is_btype & EXE.is_jtype) {
    IF.stall = 1;
    ID.flush = 0;
}
```

### Stall 处理 Control Hazard 和 Data Hazard 同时出现:

采用数据竞争的方式。

#### 1.1.2 RaceController: 在 EXE 阶段确定需要跳转再 stall, Control Hazard 先于 Data Hazard 处理。

由于 BRAM 延后一拍取指令，所以对于冲突的处理不仅在 RaceController 中处理，取 inst\_if 的指令时，采用这样的方法：

```
wire [63:0] ro_addr = (branch_flag_exe & decode_exe[19]) ? alu_out_exe[11:3] : (stall_PC ?
pc_if[11:3] : npc_if[11:3]);

/*
    exe 阶段的 inst 需要跳转时，ro_addr 读入 inst_exe 的跳转地址；不需要跳转时，如果有数据冲突，
    传入 pc_if 实现 stall，如果没有数据冲突，就提前一拍传入 pc，即输入 npc_if。
*/
```

在 RaceController 做 Data Hazard 和 Control Hazard 的判断以及阶段寄存器 stall 和 flush 操作的设置:

```
//
wire controlrace = branch_flag_exe & decode_exe[19];

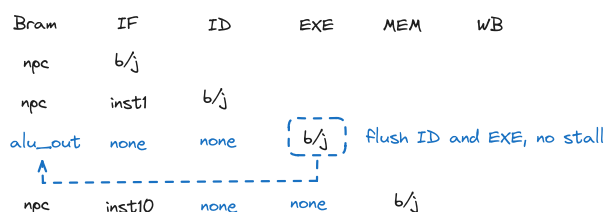
assign stall_PC = datarace & ~controlrace;
assign stall_IFID = datarace & ~controlrace;
assign stall_IDEXE = datarace & ~controlrace;

assign flush_IFID = controlrace;
assign flush_IDEXE = datarace | controlrace;

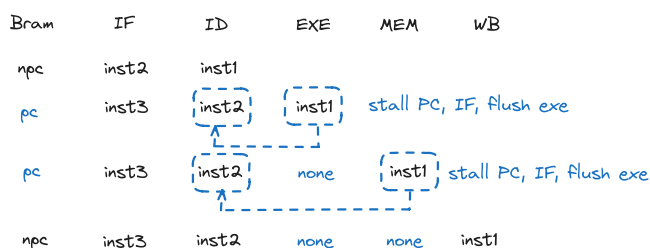
/* 别的 stall flush 接口都直接 assign 为 0 */
```

几种冲突情况如下:

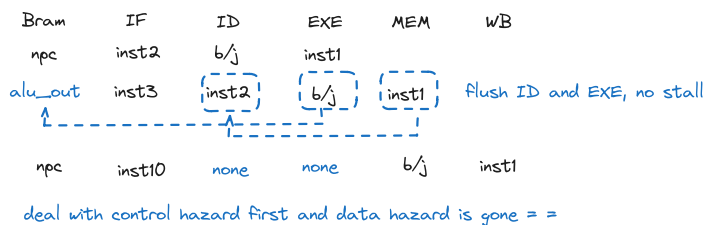
#### Control Hazard Only



#### Data Hazard Only



#### Data Hazard and Control Hazard



## 思考题

1. 对于 `syn.asm` 的 fibonacci (13 - 19 行), 请计算该 loop 在流水线 CPU 的 CPI, 再用 lab0 的单周期 CPU 运行 test1, 对比二者的 CPI;

```

0:  00100093          addi    ra,zero,1
4:  00100113          addi    sp,zero,1
8:  00500213          addi    tp,zero,5
# ra = 1, sp = 1, tp = 5
000000000000000c <fibonacci>:
c:  002081b3          add     gp,ra,sp
10: 003100b3          add     ra,sp,gp # data hazard
14: 00308133          add     sp,ra,gp # data hazard
18: fff20213          addi    tp,tp,-1
1c: fe4018e3          bne     zero,tp,c <fibonacci> # control hazar
20: 63d00293          addi    t0,zero,1597
24: 0c511e63          bne     sp,t0,100 <fail> # data hazard

```

- Pipeline: data hazard flush 两拍 (相当于stall 2 拍), 确定跳转的 control hazard stall 两拍

$$Instruction = 5 \times 5 + 2 = 27$$

$$Time\ Cycle = inst + 2 \times datahazard + 2 \times controlhazard$$

$$TimeCycle = 3 + (5 + 2 \times 3 + 2) \times 4 + 2 = 68$$

IF	ID	EXE	MEM	WB
a1	ai3	ai2	ai1	
a2	a1	ai3	ai2	ai1
a2	a1	ai3	none	ai2

3

a3	a2	a1	ai3	none
a3	a2	none	a1	ai3
a3	a2	none	none	a1
ai	a3	a2	none	none
ai	a3	none	a2	none
ai	a3	none	none	a2
bne	ai	a3	none	none
ai	bne	ai	a3	none
ai	bne	none	ai	a3
ai	bne	none	none	ai
none	none	bne	ai	a3
a1	none	none	bne	ai
a2	a1	none	none	bne

13 x 4 times

a3 a2 a1 none none

...

a3	a2	a1	none	none
a3	a2	none	a1	none
a3	a2	none	none	a1
ai	a3	a2	none	none
ai	a3	none	a2	none
ai	a3	none	none	a2
bne	ai	a3	none	none
ai	bne	ai	a3	none

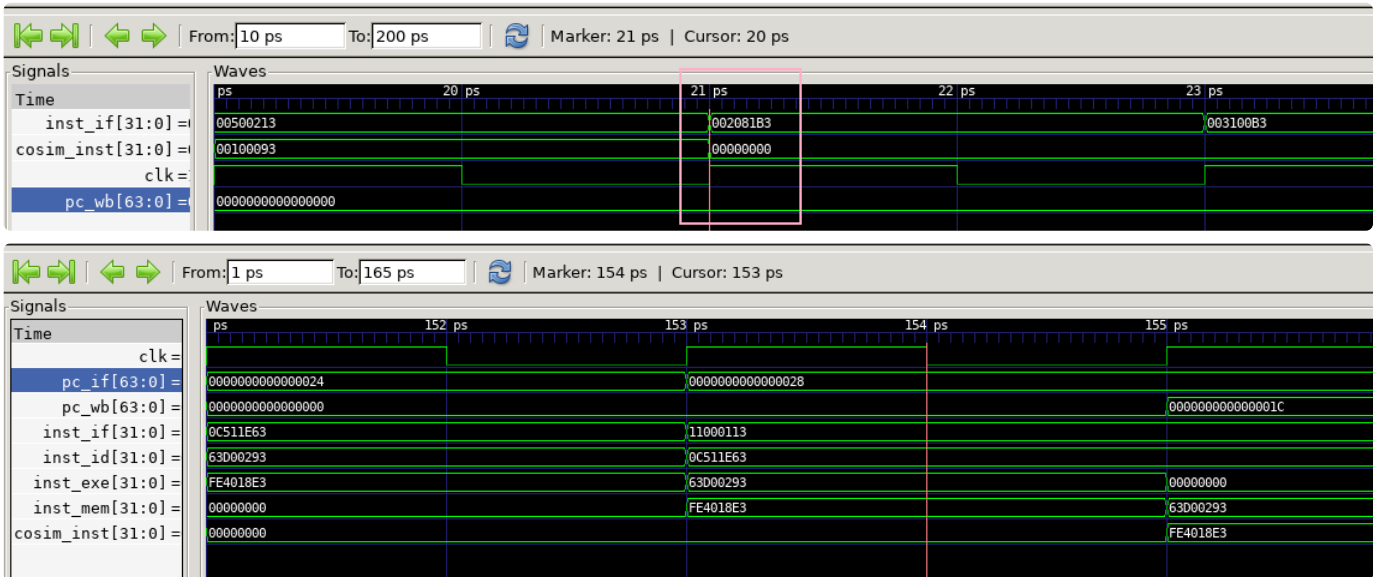
8

bne2    addi    bne    ai    a3  
           bne2    addi    bne    ai  
                   bne2    addi    bne

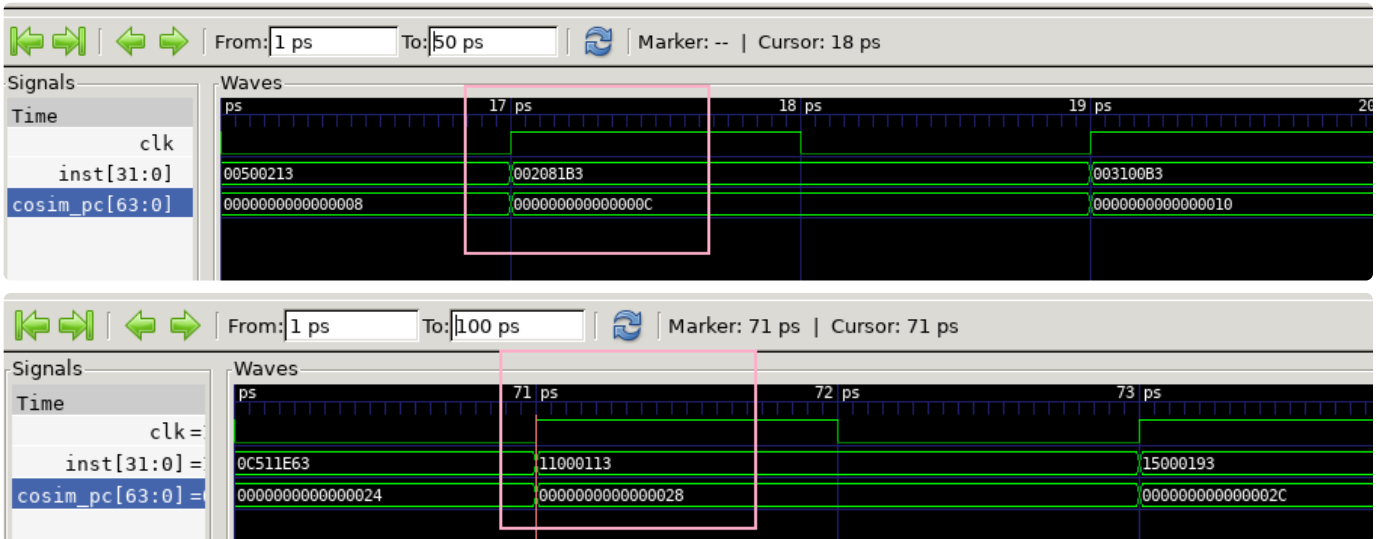
5

bne2     addi  
bne2

观察 Pipeline 的波形验证, 1 clk = 2 ps, start = 21 ps. end = 153 ps. Time Cycle = 68, CPI = 68/27



观察 SCPU 的波形, 1 clk= 2 ps, start = 17 ps, end = 71 ps, Time Cycle = 27, CPI =1



所以 PipelineCPU 的 CPI 为 68/27, 单周期的 CPI 为 1。

2. 对于 test2 (21-47 行), 请计算你的 CPU 的 CPI, 再用 lab0 的单周期 CPU 运行 test2, 对比二者的 CPI;

```
000000000000028 <test2>:
28: 11000113      addi    sp,zero,272
2c: 15000193      addi    gp,zero,336
30: 00010083      lb      ra,0(sp) # data hazard
34: 00118023      sb      ra,0(gp) # data hazard
38: 00110083      lb      ra,1(sp)
3c: 001180a3      sb      ra,1(gp) #
```

```

40: 00211083      lh      ra,2(sp)
44: 00119123      sh      ra,2(gp) #
48: 00412083      lw      ra,4(sp)
4c: 0011a223      sw      ra,4(gp) #
50: 00813083      ld      ra,8(sp)
54: 0011b423      sd      ra,8(gp) #
58: 00014083      lbu     ra,0(sp)
5c: 0011b823      sd      ra,16(gp) #
60: 00010083      lb      ra,0(sp)
64: 0011bc23      sd      ra,24(gp) #
68: 00015083      lhu     ra,0(sp)
6c: 0211b023      sd      ra,32(gp) #
70: 00011083      lh      ra,0(sp)
74: 0211b423      sd      ra,40(gp) #
78: 00016083      lwu     ra,0(sp)
7c: 0211b823      sd      ra,48(gp) #
80: 00012083      lw      ra,0(sp)
84: 0211bc23      sd      ra,56(gp) #
88: 00800093      addi    ra,zero,8
8c: 00800293      addi    t0,zero,8

```

- Pipeline CPU:

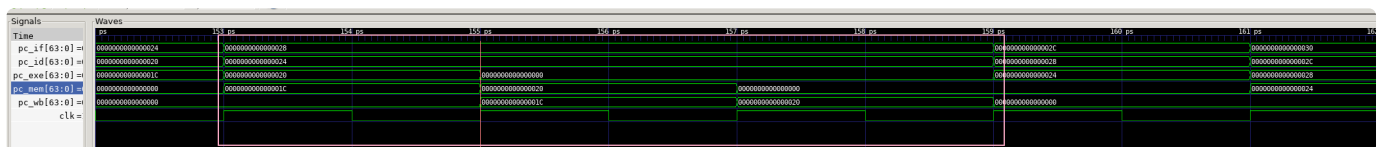
理论计算:

$Instructioncount : 26$

$ClockCycle : 4 + 26 + 12 \times 2 = 54$

观察 Pipeline 的波形验证, 1 clk = 2 ps, start = 153 ps. end = 265 ps. Time Cycle = 56。不符合理论计算 0.0。观察了波形发现在 0x28 处的 test2 的第一条指令进入 IF 阶段时就由于之前 0x20 处的指令被 stall 了, 加上这个影响计算的 Clock Cycle 应该是 56, 理论结果符合逻辑。

$CPI = 56/26 = 28/13$



- SCPU: CPI = 1

### 3. 请你对数据冲突,控制冲突情况进行分析归纳, 试着将他们分类列出;

我对于控制冲突处理的方式是预测都不跳转, 在 EXE 阶段确认了跳转才是 Control Hazard, 处理方式是 flush 掉 IF 和 ID 阶段的指令, 重新读入 ro\_addr; 对于数据冲突是判断 ID 阶段读的寄存器和 EXE/MEM 阶段确认需要写回的寄存器相同, stall IFID 和 IDEXE。由于都是确认了需要做操作才判断为 hazard, 所以出现 Data Hazard 和 Control Hazard 的情况比较单一。

- 数据冲突：寄存器读写冲突

1. ID 阶段的指令读取的寄存器 id 与 EXE 阶段写回的寄存器地址相同
2. ID 阶段的指令读取的寄存器 id 与 MEM 阶段写回的寄存器地址相同

- 控制冲突：发生跳转。由于我是预测都不跳转，所以控制冲突只有一种情况

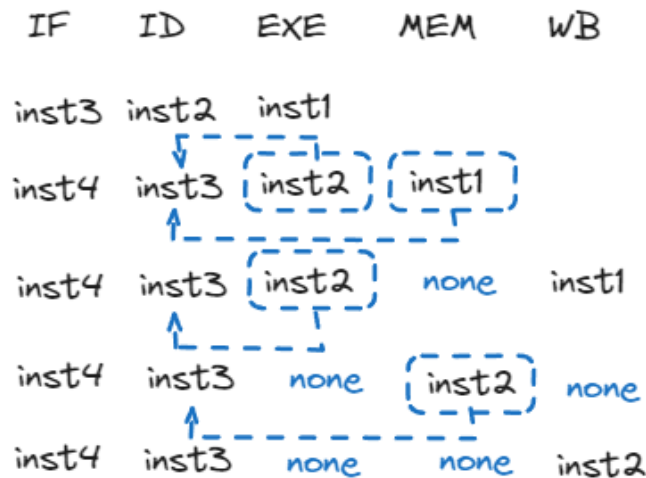
- EXE 阶段的指令确认进行跳转、

- 控制冲突和读写冲突同时发生：

- 由于预测都不跳转，只检测了 EXE 阶段的 inst 是否确认跳转，在 control hazard 出现的情况下，进行跳转之后，之后 IF ID 阶段的 inst 都会变化，data hazard 就会自己消失了。

4. 如果 EX/MEM/WB 段中不止一个段的写寄存器与 ID 段的读寄存器发生了冲突，该如何处理？

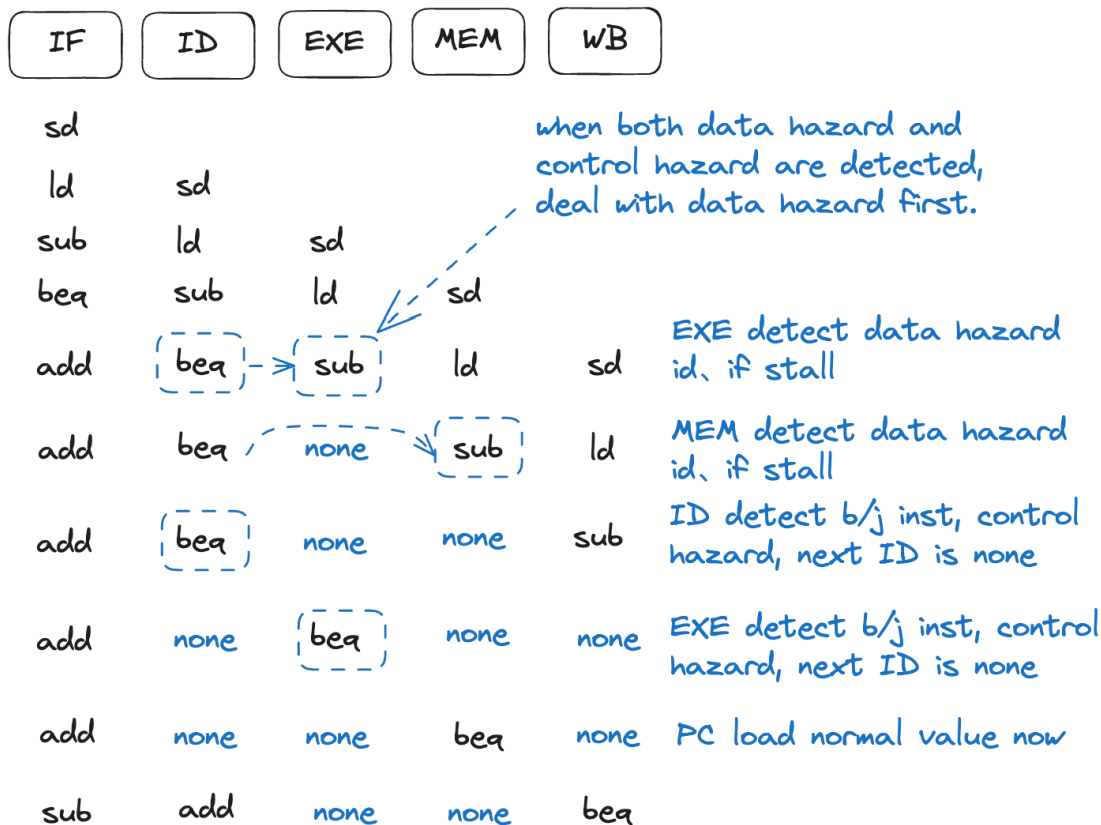
。。TT 由于我的 reg 设置为下降沿触发，所以不用判断 WB 阶段的写寄存器对于 ID 阶段的影响。而在 EX/MEM 段中不止一个段的写寄存器与 ID 段的读寄存器发生了冲突，不需要多余的处理，按照 data hazard 的检测依次 stall 即可。



5. 如果数据冲突和控制冲突同时发生应该如何处理呢？

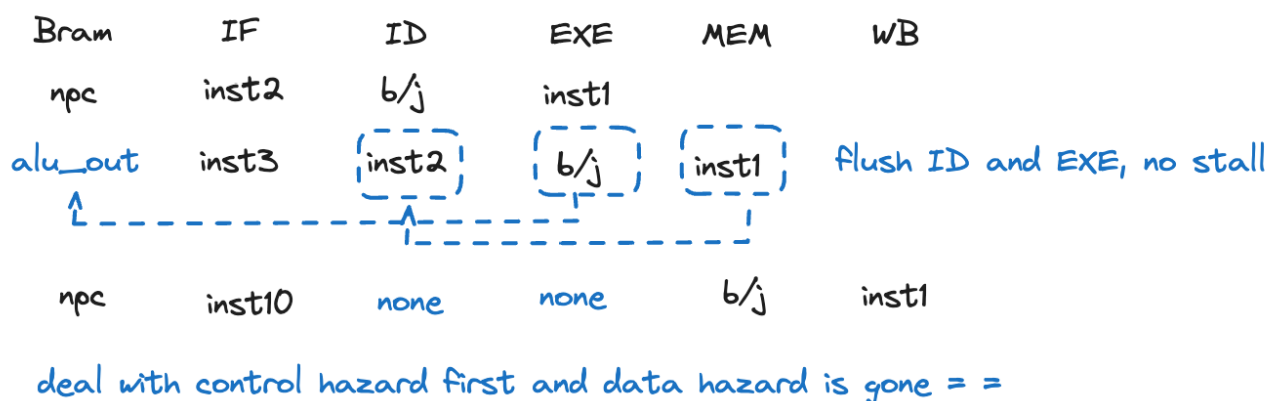


- 根据 1.2.1 设计的逻辑，优先解决数据冲突。



- 根据 1.1.2 设计的逻辑，优先解决控制冲突。因为我设计的流水线预测都不跳转，等到 **b/j** 指令到 **EXE** 阶段了之后确定跳转了再判断发生了控制冲突。解决了控制冲突后数据冲突就消失了。

### Data Hazard and Control Hazard



- 能否使用 BRAM 来实现寄存器组，如果不行是出于什么原因，如果可以需要怎么修改？  
可以。如果使用 BRAM 来实现寄存器组，需要提前一拍传入数据。由于 ID 阶段才解码得到 rs 等信息，所以修改将 Regs 放在 EXE 阶段，提前一拍传入所需要的 rs 和 rd。
- 尝试分析整体设计的关键路径，也就是最长路径，可以参考自己的 implement 报告。

Intra-Clock Paths - dut_clk_virt - Setup													
Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exception	Clock Uncertainty
↳ Path 1	29.562	1	64	dut/memwb/decode_wb_reg[21]/C	dut/Regs/register_reg[22][1]/CE	10.182	0.580	9.602	40.0	dut_clk_virt	dut_clk_virt		0.035
↳ Path 2	29.562	1	64	dut/memwb/decode_wb_reg[21]/C	dut/Regs/register_reg[22][20]/CE	10.182	0.580	9.602	40.0	dut_clk_virt	dut_clk_virt		0.035
↳ Path 3	29.562	1	64	dut/memwb/decode_wb_reg[21]/C	dut/Regs/register_reg[22][22]/CE	10.182	0.580	9.602	40.0	dut_clk_virt	dut_clk_virt		0.035
↳ Path 4	29.562	1	64	dut/memwb/decode_wb_reg[21]/C	dut/Regs/register_reg[22][24]/CE	10.182	0.580	9.602	40.0	dut_clk_virt	dut_clk_virt		0.035
↳ Path 5	29.562	1	64	dut/memwb/decode_wb_reg[21]/C	dut/Regs/register_reg[22][34]/CE	10.182	0.580	9.602	40.0	dut_clk_virt	dut_clk_virt		0.035
↳ Path 6	29.562	1	64	dut/memwb/decode_wb_reg[21]/C	dut/Regs/register_reg[22][37]/CE	10.182	0.580	9.602	40.0	dut_clk_virt	dut_clk_virt		0.035
↳ Path 7	29.717	1	64	dut/memwb/inst_wb_reg[8]/C	dut/Regs/register_reg[31][45]/CE	10.008	0.580	9.428	40.0	dut_clk_virt	dut_clk_virt		0.035
↳ Path 8	29.717	1	64	dut/memwb/inst_wb_reg[8]/C	dut/Regs/register_reg[31][47]/CE	10.008	0.580	9.428	40.0	dut_clk_virt	dut_clk_virt		0.035
↳ Path 9	29.723	1	64	dut/memwb/decode_wb_reg[21]/C	dut/Regs/register_reg[22][29]/CE	9.994	0.580	9.414	40.0	dut_clk_virt	dut_clk_virt		0.035
↳ Path 10	29.896	1	64	dut/memwb/decode_wb_reg[21]/C	dut/Regs/register_reg[22][31]/CE	9.814	0.580	9.234	40.0	dut_clk_virt	dut_clk_virt		0.035

# Appendix: Source Code