浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：展翼飞

学　　院：计算机

专　　业：计算机科学与技术

学　　号：3190102196

指导教师：王总辉

2023　年　10　月　20　日

浙江大学实验报告

课程名称：计算机体系结构 实验类型：综合

实验项目名称：Lab1：Pipelined CPU supporting RISC-V RV32I Instructions

学生姓名：展翼飞 专业：计算机科学与技术 学号：3190102196

同组学生姓名： 指导老师：王总辉

实验地点：曹西 301 实验日期：2023 年 10 月 18 日

一、 实验目的和要求

1.1 实验目的

- 理解 RISC-V RV32I 指令
- 掌握执行 RV32I 指令的流水线 CPU 的设计方法
- 掌握 Pipeline Forwarding Detection 和 bypass unit 设计的方法
- 掌握 Predict-not-taken 分支设计的 1 周期停顿的方法
- 掌握执行 RV32I 指令的流水线 CPU 程序验证的方法

1.2 实验要求

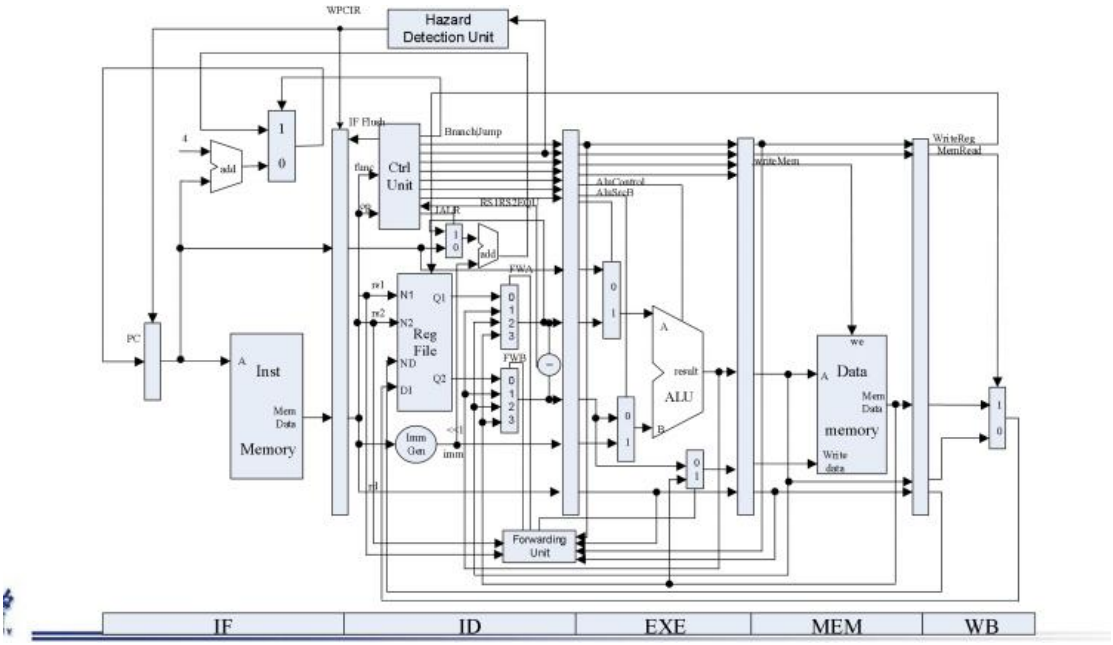
设计 RV32I 指令的流水线 CPU

包含数据路径设计，Bypass 单元设计和 CPU 控制器设计。

二、 实验内容和原理

实现模块： Hazard detection， rv32i core（主要为多路选择器）， cpu controller

Hazard 原理：



1.结构冒险

* 原因：在 WB 阶段写寄存器时同时有 ID 阶段的指令读寄存器

* 解决方式：寄存器在时钟上边沿写，下边沿读

2.数据冒险

* 原因：当前指令需要写回寄存器的结果在 WB 阶段前前就要被之后的指令读相应寄存器

* 解决方法；

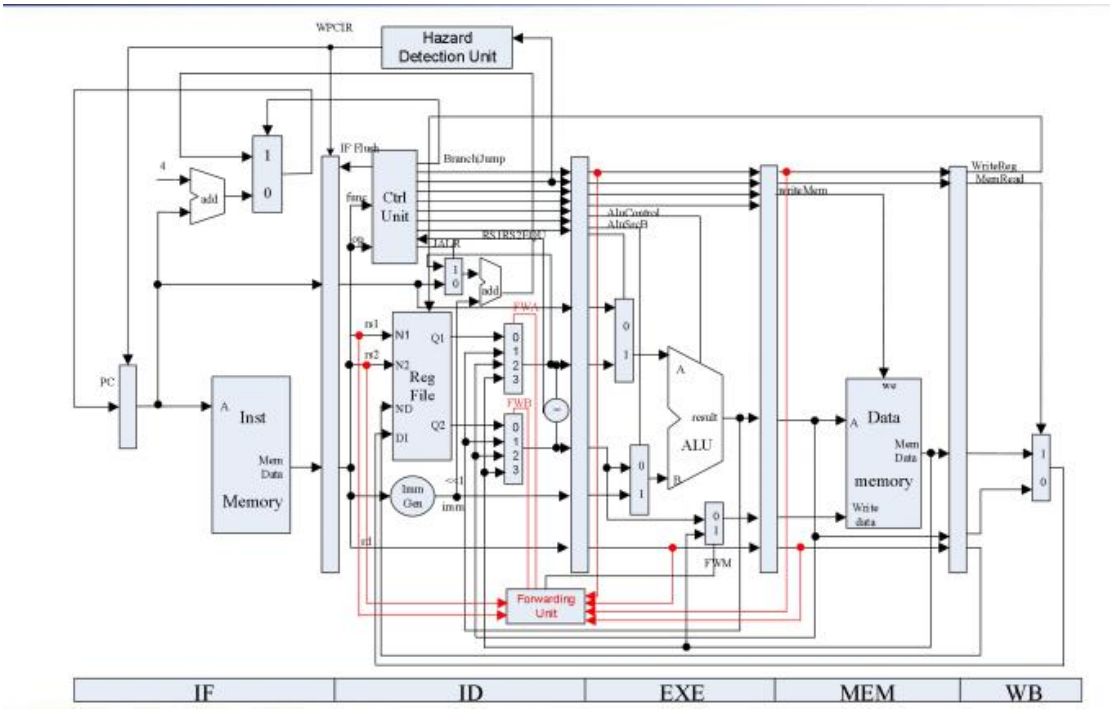
* R | I | J 或 LUI, AUIPC 后读相应寄存器，则从 EXE 或 MEM 阶段的 ALU 结果 forward 到 ID（对应 forward control unit 控制 ID 多路选择器的线 1 2）

* L 类型指令后紧接 S 类型指令读相应寄存器，则从 MEM 阶段的取回值 forward 到 EXE（对应 EXE 阶段下方准备传给 MEM WriteData 的多路选择器线 1）

* L 类型指令一周期后接读相应寄存器指令，则从 MEM 阶段的取回值 forward 到 ID（对应 forward control unit 控制 ID 多路选择器的线 3）

* L 类型指令紧接 R | I | J 或 LUI，AUIPC 后读相应寄存器，无法 forward（当

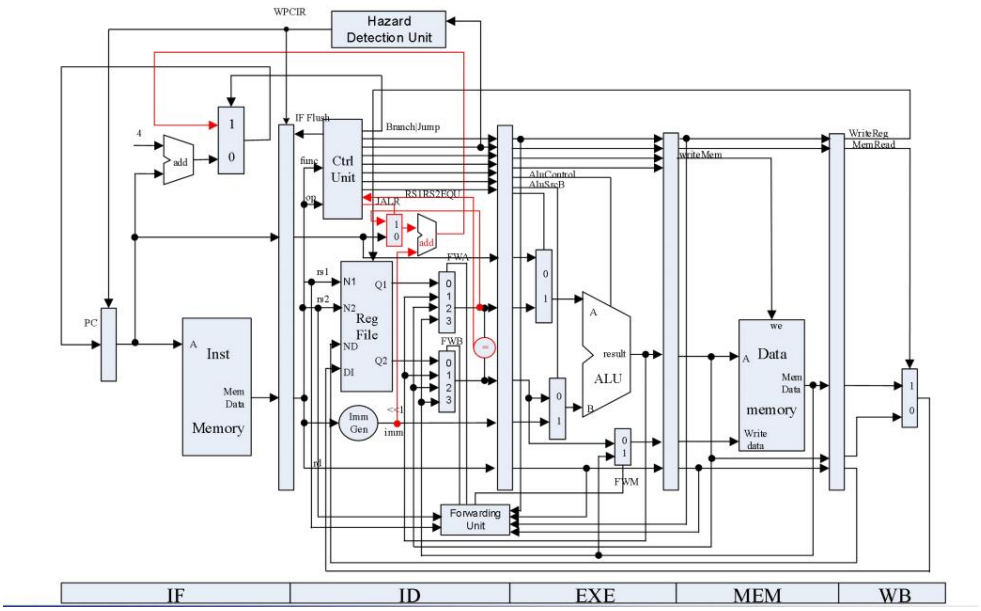
前指令 load 时下一条已经在 exe），需要 Flush ID-EXE 并 Stall 之前的寄存器（L 指令在 EXE 阶段时判断）



3. 控制冒险

策略：Predict branch taken

ID 阶段在检测到跳转指令后由 Hazard Control Unit 输出控制信号，Flush 掉 IF-ID 寄存器的一个 cycle 使一个周期后 IF 取到的地址为 ID 阶段当前时钟末上边沿产出的需要跳转正确地址



三、　重要部分代码和分析（主要介绍**你写的**代码的逻辑，贴代码+文字描述）

1. RV32core 模块

主要为多路选择器的线路填写，理解信号名，各个模块代表的含义，按照流水线 CPU 线路原理图填写即可

```
// IF
REG32 REG_PC(.clk(debug_clk),.rst(rst),.CE(PC_EN_IF),.D(next_PC_IF),.Q(PC_IF));

add_32 add_IF(.a(PC_IF),.b(32'd4),.c(PC_4_IF));

MUX2T1_32 mux_IF(.I0(PC_4_IF),.I1(jump_PC_ID),.s(Branch_ctrl),.o(next_PC_IF));        //to fill sth. in ()
```

分析：IF 阶段取指令地址的多路选择器，输入 PC+4 与 ID 阶段得到的跳转地址以及分支控制信号，输出下一条指令的地址

```
MUX4T1_32 mux_forward_A(.I0(rs1_data_reg),.I1(ALU_out_EXE),.I2(ALU_out_MEM),.I3(Datain_MEM),        /
    .s(forward_ctrl_A),.o(rs1_data_ID));

MUX4T1_32 mux_forward_B(.I0(rs2_data_reg),.I1(ALU_out_EXE),.I2(ALU_out_MEM),.I3(Datain_MEM),        /
    .s(forward_ctrl_B),.o(rs2_data_ID));
```

分析：ID 阶段 forward unit 控制的两个多路选择器，输入寄存器读取内容与三种 forward 信息与控制信号，输出选择器结果，三种 forward 分别是 EXE 阶段 ALU 结果，MEM 阶段的 ALU 结果和 MEM 阶段的内存读取结果

```
MUX2T1_32 mux_A_EXE(.I0(PC_EXE),.I1(rs1_data_EXE),.s(ALUSrc_A_EXE),.o(ALUA_EXE));     //to fill sth. in ()

MUX2T1_32 mux_B_EXE(.I0(rs2_data_EXE),.I1(Imm_EXE),.s(ALUSrc_B_EXE),.o(ALUB_EXE));       //to fill sth. in (
```

分析：EXE 阶段控制 ALU 输入的两个多路选择器，控制信号由 Control unit 产生并通过阶段寄存器传递到 ALUA_EXE 与 ALUB_EXE

```
MUX2T1_32 mux_forward_EXE(.I0(rs2_data_EXE),.I1(Datain_MEM),.s(forward_ctrl_ls),.o(Dataout_EXE));        //t
```

分析：控制 load 后紧接 store 指令的 forward

2. ControlUnit 模块

需要根据 ID 阶段的 32 位指令读取相应信息，分析指令类型，获取立即数，生成跳转，ALU 前多路选择器，冒险类型等控制信号

```
wire BEQ = Bop & funct3_0;                                  //to fill sth. in
//beq指令只有在操作数寄存器rs1中的数值与操作数寄存器rs2的数值相等时，才会跳转跳,转
wire BNE = Bop & funct3_1;                                  //to fill sth. in
//bne指令只有在操作数寄存器rs1中的数值与操作数寄存器rs2的数值不相等时，才会跳转,转
wire BLT = Bop & funct3_4;                                  //to fill sth. in
//blt指令只有在操作数寄存器rs1中的数值小于操作数寄存器rs2的数值时（有符号数），才
wire BGE = Bop & funct3_5;                                  //to fill sth. in
//bge指令只有在操作数寄存器rs1中的数值大于或等于操作数寄存器rs2的数值时（有符号数)
wire BLTU = Bop & funct3_6;                                 //to fill sth. in
//bltu指令只有在操作数寄存器rs1中的数值小于操作数寄存器rs2的数值时（无符号数），才
wire BGEU = Bop & funct3_7;                                 //to fill sth. in
//bgeu指令只有在操作数寄存器rs1中的数值大于或等于操作数寄存器rs2的数值时（无符号数

wire LB =  Lop & funct3_0;                                  //to fill sth. in
//从地址 x[rs1] + sign-extend(offset)读取一个字节，经符号位扩展后写入 x[rd]
wire LH =  Lop & funct3_1;                                  //to fill sth. in
//从地址 x[rs1] + sign-extend(offset)读取两个字节，经符号位扩展后写入 x[rd]
wire LW =  Lop & funct3_2;                                  //to fill sth. in
//从地址 x[rs1] + sign-extend(offset)读取四个字节，写入 x[rd]。对于 RV64I，结果
wire LBU = Lop & funct3_4;                                  //to fill sth. in
//从地址 x[rs1] + sign-extend(offset)读取一个字节，经零扩展后写入 x[rd]
wire LHU = Lop & funct3_5;                                  //to fill sth. in
//从地址 x[rs1] + sign-extend(offset)读取两个字节，经零扩展后写入 x[rd]。
```

分析：这部分位指令种类读取，根据它们所属类型与 function3，opcode 等信号即可判断

```
assign Branch = BEQ | BNE | BLT | BGE | BLTU | BGEU | JAL | JALR;
```

分析：跳转控制信号，为 B 类型指令加上 JAL,JALR

```
assign cmp_ctrl = 3'b001 & {3{BEQ}} |
                  3'b010 & {3{BNE}} |
                  3'b011 & {3{BLT}} |
                  3'b100 & {3{BLTU}} |
                  3'b101 & {3{BGE}} |
                  3'b110 & {3{BGEU}};                    //to fill sth. in
```

分析：compare 控制信号，与 compare Unit 对应即可

```
assign rs1use =  R_valid | I_valid | S_valid | B_valid | L_valid | JALR;                    //to fill

assign rs2use = R_valid | S_valid | B_valid;                    //to fill sth. in
```

分析：当前指令两条寄存器读取线路使用信号

```
//hazard_optype
assign hazard_optype =  2'b01 & {2{R_valid | I_valid | JAL | JALR | LUI | AUIPC}} |
                        2'b10 & {2{L_valid}}|
                        2'b11 & {2{S_valid}};                    //to fill sth. in
```

分析：Hazard 类型控制信号，01 意为非 L 或 S 类型的指令，10 意为 L 类型指令，11 意为 S 类型指令，后续输入 HazardDetection 模块，便于辨别冒险类型，生成冒险相关控制信号

3. HazardDetection 模块

通过输入各阶段对寄存器的读写情况，判断冒险类型，输出各阶段寄存器使能与 Flush 信号

```
reg[1:0] hazard_optype_EXE, hazard_optype_MEM;
always@(posedge clk) begin
    hazard_optype_MEM <= hazard_optype_EXE;
    hazard_optype_EXE <= hazard_optype_ID & {2{~reg_DE_flush}};
end
```

分析：通过寄存器将 ID 阶段产生的 hazard_optype 传递并保留到 EXE, MEM 阶段，便于判断冒险类型

```
//1.数据竞争   01 非L类型需要读写寄存器指令 10 L类型 11 S类型
//EXE阶段forward到ID阶段
wire rs1_forward_1     = rs1use_ID && rs1_ID == rd_EXE && rd_EXE && hazard_optype_EXE == 2'b01;
//如果时L类型指令后紧接非S类型读相应寄存器指令，需要stall 1个cycle
wire rs1_forward_stall = rs1use_ID && rs1_ID == rd_EXE && rd_EXE && hazard_optype_EXE == 2'b10
                       && hazard_optype_ID != 2'b11;
//第一种类型的hazard后接读相应寄存器指令，MEM阶段forward到ID阶段
wire rs1_forward_2     = rs1use_ID && rs1_ID == rd_MEM && rd_MEM && hazard_optype_MEM == 2'b01;
//L类型指令一周期后接读相应寄存器指令，从MEMforward到ID
wire rs1_forward_3     = rs1use_ID && rs1_ID == rd_MEM && rd_MEM && hazard_optype_MEM == 2'b10;

wire rs2_forward_1     = rs2use_ID && rs2_ID == rd_EXE && rd_EXE && hazard_optype_EXE == 2'b01;
wire rs2_forward_stall = rs2use_ID && rs2_ID == rd_EXE && rd_EXE && hazard_optype_EXE == 2'b10
                       && hazard_optype_ID != 2'b11;
wire rs2_forward_2     = rs2use_ID && rs2_ID == rd_MEM && rd_MEM && hazard_optype_MEM == 2'b01;
wire rs2_forward_3     = rs2use_ID && rs2_ID == rd_MEM && rd_MEM && hazard_optype_MEM == 2'b10;

//L类型指令后紧接S类型指令
assign forward_ctrl_ls = rs2_EXE == rd_MEM && hazard_optype_EXE == 2'b11
                      && hazard_optype_MEM == 2'b10;
```

```
//L类型指令后紧接S类型指令
assign forward_ctrl_ls = rs2_EXE == rd_MEM && hazard_optype_EXE == 2'b11
                      && hazard_optype_MEM == 2'b10;

assign forward_ctrl_A = {2{rs1_forward_1}} & 2'd1 |
                        {2{rs1_forward_2}} & 2'd2 |
                        {2{rs1_forward_3}} & 2'd3 ;

assign forward_ctrl_B = {2{rs2_forward_1}} & 2'd1 |
                        {2{rs2_forward_2}} & 2'd2 |
                        {2{rs2_forward_3}} & 2'd3 ;
```

分析：通过 ID 阶段寄存器读取情况与后续阶段寄存器写入情况判断 forward 类型，生成 forward 与 stall 的控制信号

R | I | J 或 LUI, AUIPC 后读相应寄存器，则从 EXE 或 MEM 阶段的 ALU 结果 forward 到 ID（对应 forward control unit 控制 ID 多路选择器的线 1 2）

L 类型指令后紧接 S 类型指令读相应寄存器，则从 MEM 阶段的取回值 forward 到 EXE（对应 EXE 阶段下方准备传给 MEM WriteData 的多路选择器线 1）

L 类型指令一周期后接读相应寄存器指令，则从 MEM 阶段的取回值 forward 到 ID（对应 forward control unit 控制 ID 多路选择器的线 3）

L 类型指令紧接 R | I | J 或 LUI，AUIPC 后读相应寄存器，无法 forward（当前指令 load 时下一条已经在 exe），需要 Flush ID-EXE 并 Stall 之前的寄存器（L 指令在 EXE 阶段时判断）

```
wire load_stall = rs1_forward_stall | rs2_forward_stall;

assign PC_EN_IF = ~load_stall;
assign reg_FD_EN = ~load_stall;
assign reg_FD_stall = load_stall;
assign reg_FD_flush = Branch_ID;
assign reg_DE_EN = ~load_stall;
assign reg_DE_flush = load_stall;
assign reg_EM_EN = 1;
assign reg_EM_flush = 0;
assign reg_MW_EN = 1;
```

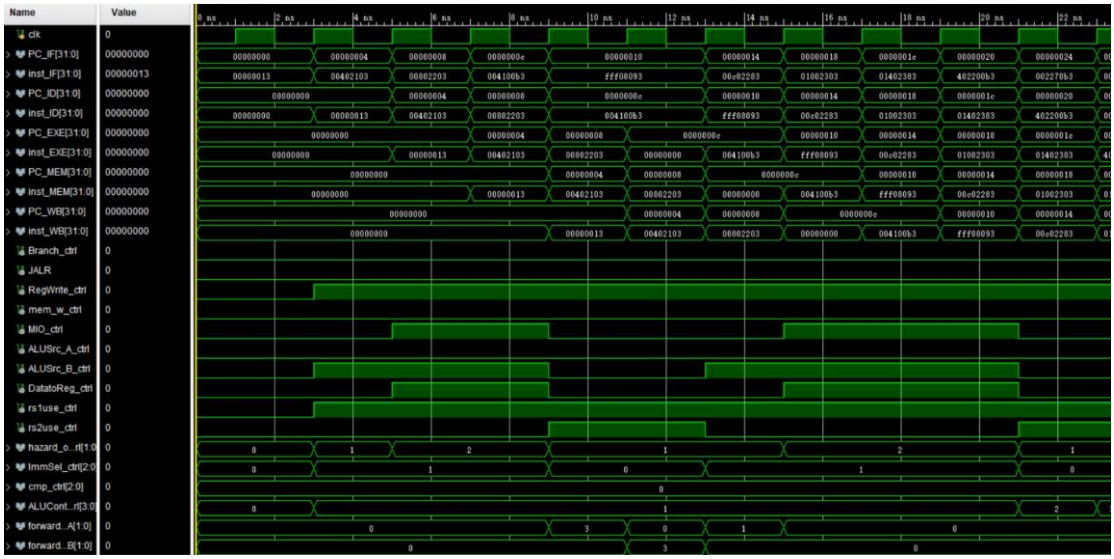
通过 forward stall 与 ID 阶段的跳转控制信号得到各阶段寄存器是否需要 stall 或 flush。只有跳转和分支外仅有 load 后紧接 store 指令后需要 stall。

如果 load 后紧接 store，则 L 指令在 EXES 指令在 ID 阶段时得到判断，则需要 EXE 指令后的 IF 到 ID，ID 到 EXE 均 Stall 一个 cycle，即阶段寄存器使能置 0。

如果 ID 阶段判断到需要跳转，则采取 Branch predict taken 策略，flush IF 到 ID 的阶段寄存器并使 PC 取指令寄存器 stall 一个 cycle 等待分支控制信号与正确的指令地址。

四、 仿真结果和分析

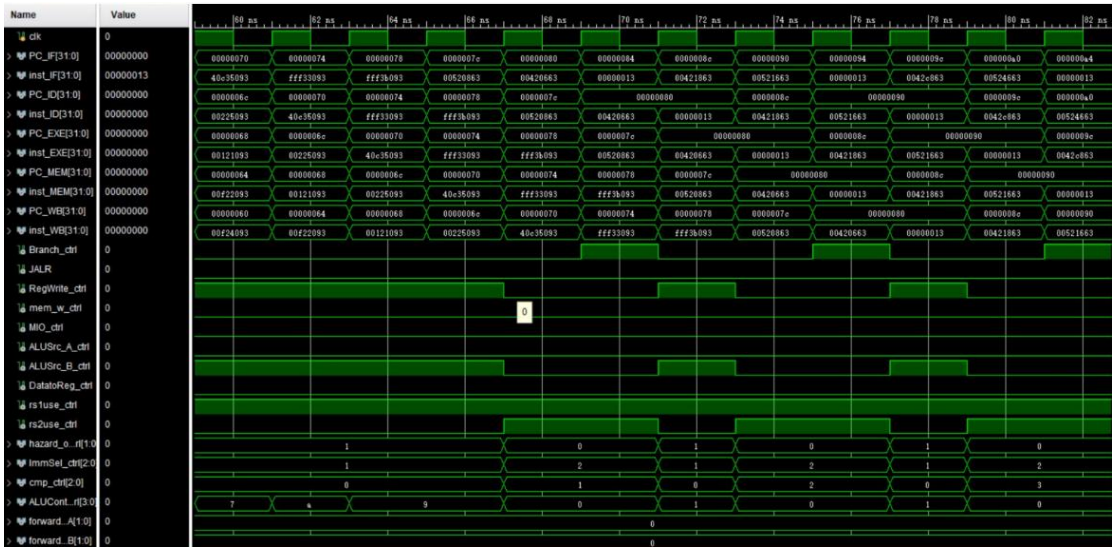
数据冒险：



NO.	Instruction	Addr.	Label	ASM	Comment
0	00000013	0	__start:	addi x0, x0, 0	
1	00402103	4		lw x2, 4(x0)	
2	00802203	8		lw x4, 8(x0)	
3	004100b3	C		add x1, x2, x4	
4	fff08093	10		addi x1, x1, -1	
5	00c02283	14		lw x5, 12(x0)	
6	01002303	18		lw x6, 16(x0)	
7	01402383	1C		lw x7, 20(x0)	
8	402200b3	20		sub x1,x4,x2	
9	002270b3	24		and x1,x4,x2	
10	002260b3	28		or x1,x4,x2	
11	002240b3	2C		xor x1,x4,x2	
12	002210b3	30		sll x1,x4,x2	
13	002220b3	34		slt x1,x4,x2	
14	004120b3	38		slt x1,x2,x4	

lw 后第二条指令读相应寄存器：从上图可以看出 t = 9ns 时指令 add x1, x2, x4 正处于 ID 阶段，且 lw x2, 4(x0)处于 MEM 阶段，发生了 lw 后的第二条指令读相应寄存器，hazard optype 为 1，forward 为 3，与理论相对应

控制冒险：



NO.	Instruction	Addr.	Label	ASM	Comment
30	fff3b093	78		sltiu x1, x7, -1	
31	00520863	7C		beq x4,x5,label0	
32	00420663	80		beq x4,x4,label0	
33	00000013	84		addi x0,x0,0	
34	00000013	88		addi x0,x0,0	
35	00421863	8C	label0:	bne x4,x4,label1	
36	00521663	90		bne x4,x5,label1	
37	00000013	94		addi x0,x0,0	
38	00000013	98		addi x0,x0,0	
39	0042c863	9C	label1:	blt x5,x4,label2	
40	00524663	A0		blt x4,x5,label2	
41	00000013	A4		addi x0,x0,0	
42	00000013	A8		addi x0,x0,0	
43	00736863	AC	label2:	bltu x6,x7,label3	
44	0063e663	B0		bltu x7,x6,label3	

分析：t = 69s 时 beq x4, x5 label0 处于 ID 阶段，分支控制信号 Branch_ctrl = 1，信号无误，此时判断发生跳转，PC 在下一周期跳到 8C，即 label0，结果无误

五、 下板结果（简要概括即可）

因基础较差，未能及时完成实验结果下板，但可以预测与仿真结果大体一致

六、 讨论与心得（optional）

1. 本次实验主要为理解五级流水线 CPU 各级的信号通路，理解三种冒险与解决方法，熟悉基本的指令与指令解码方式。重点和难点在于信号通路的梳理与冒险的解决，要合理设计控制信号，注意时钟周期与阶段寄存器的使能与冲刷控制。