

浙江大学

本科实验报告

课程名称：计算机体系结构

姓 名：夏尤楷

学 院：计算机科学与技术学院

系：计算机科学与技术

专 业：计算机科学与技术

学 号：3210104331

指导教师：陈文智

2023 年 10 月 11 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： 支持 RISC-V RV32I 指令的流水线 CPU

学生姓名： 夏尤楷 专业： 计算机科学与技术 学号： 3210104331

同组学生姓名： 来思锐 指导老师： 陈文智

实验地点： 曹西 301 实验日期： 2023 年 10 月 10 日

一、 实验目的和要求

1. 理解 RISC-V RV32I 指令；
2. 掌握执行 RV32I 指令的流水线 CPU 的设计方法；
3. 掌握设计流水线前递检测和旁路单元的方法；
4. 掌握设计不预测分支的 1 周期停顿的方法；
5. 掌握执行 RV32I 指令的流水线 CPU 的程序验证方法。

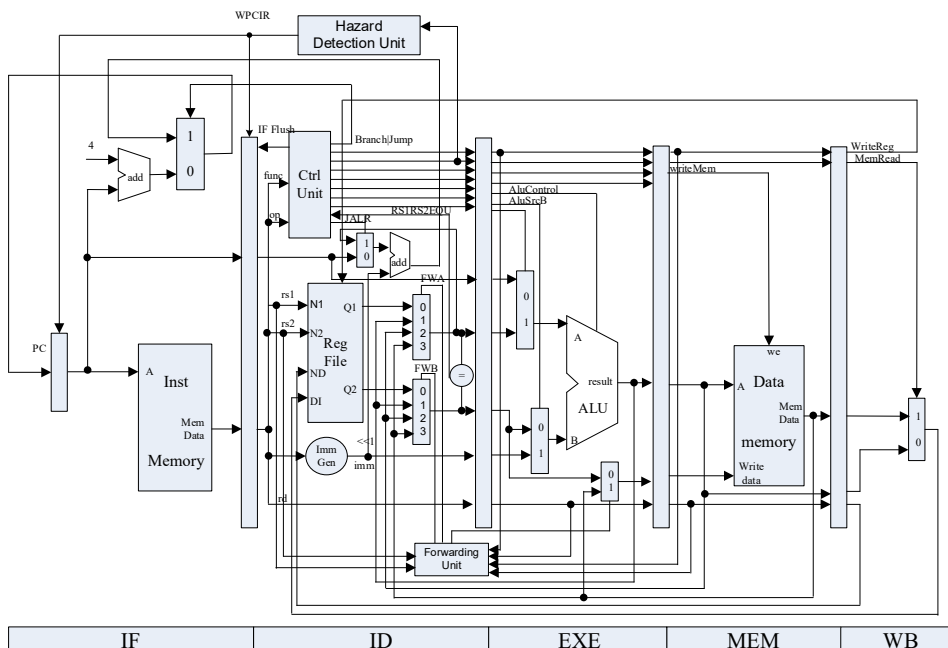
二、 实验内容和原理

（一）内容

1. 设计执行 RV32I 指令的流水线 CPU：
 - （1） 设计数据通路；
 - （2） 设计旁路单元；
 - （3） 设计 CPU 控制器；
2. 用程序验证这个流水线 CPU，并观察程序的执行。

（二）原理

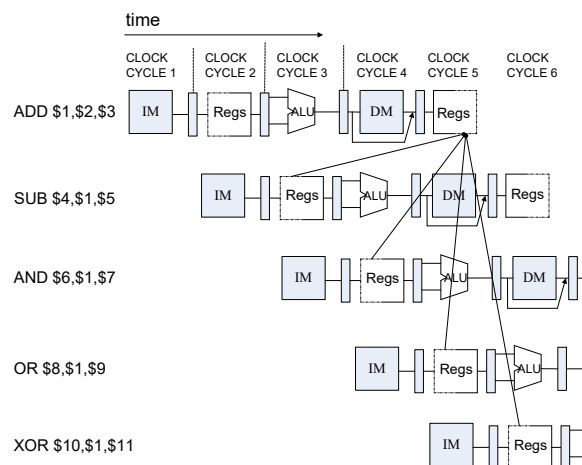
本实验流水线 CPU 结构图大致如下：



其中在设计数据通路的时候，要考虑程序执行时可能出现的数据冒险和控制冒险的情况，并为此专门设计冒险检测模块。

1.数据冒险

数据冒险主要指先执行的指令尚未将计算结果写回目标寄存器时，临近的后续的指令在执行时调用了先前指令的目标寄存器，导致后来的指令取用的寄存器内的值原应为先前指令的计算结果，但实际取用的是未及时被计算结果覆盖掉的旧值，如图。

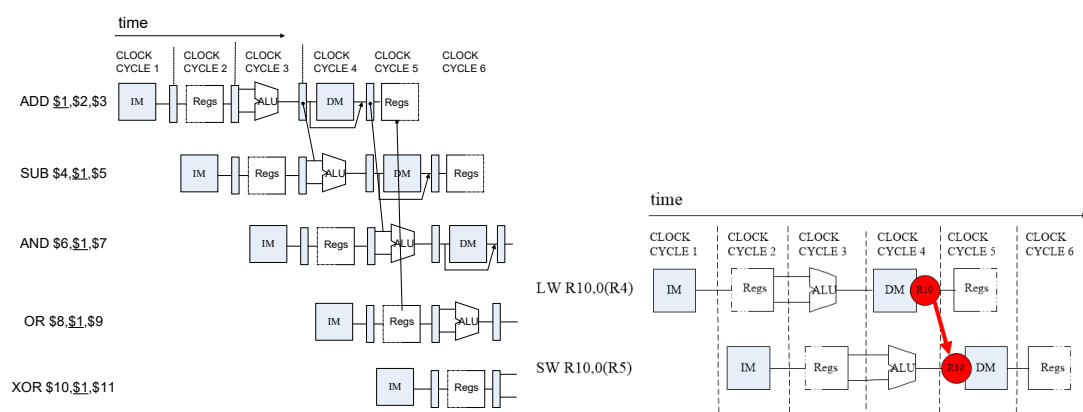


为了解决这一问题，可以在对寄存器进行写回的指令和后续的对同一寄存器进行读取的邻近指令之间插入几个停顿，等待数据写回该寄存器后再执行后续指令。但是这样会浪费时间，降低流水线运行效率。本实验采用前递的方法，即：在指令将运算结果写回该寄存器之前，通过

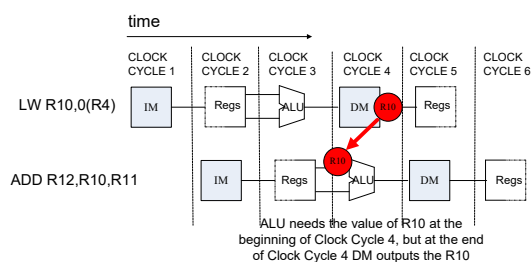
旁路单元，将运算结果提前提供给后续的读取该寄存器的临近指令，替代该指令从该寄存器中读取的未及时变更的旧值，使得整个程序无需等待写回数据即可正确运行，保障了流水线的运行效率。

前递操作有以下几个：

- (1) 无前递发生；
- (2) 先加载入的指令（非 load）处于 EXE 阶段，将其要写回某个寄存器的运算结果值前递到 ID 阶段的指令的该寄存器的位置上；（如下左图第一、二行）
- (3) 先加载入的指令（非 load）处于 MEM 阶段，将其要写回某个寄存器的运算结果值前递到 ID 阶段的指令的该寄存器的位置上；（如下左图第一、三行）
- (4) 先加载入的 load 指令处于 MEM 阶段，将其要写回某个寄存器的运算结果值前递到 ID 阶段的指令的该寄存器的位置上；
- (5) store 指令紧跟着 load 指令被加载入，load 指令处于 MEM 阶段，将从内存中读取出的要写回某个寄存器的数据前递到 EXE 阶段的 store 指令。（如下右图）



不过前递方法并非能够完全去除停顿，当 load 指令的目标寄存器和紧随其后被加载的指令（非 store）的源寄存器之一相同时，如下图：



主要是因为先前指令的结果不能及时地出现在 CPU 中，因而必须进行一个时钟周期的停顿来等待。

2.控制冒险

在原本的流水线中，分支指令在 MEM 阶段后完成分支条件是否满足的判断，这是判断是否要跳转到某特定地址的指令，将其作为下一条执行的指令的基础。然而不管是否要跳转，地址上紧随分支指令后的几条指令会依次执行，这显然不符合我们的愿望。同样，我们可以在分支指令后插入停顿，来等待分支条件是否满足的判断完成，决定出下一条要执行的指令的地址后，捕获并执行下一条指令。但是这要插入三个时钟周期的停顿，浪费的时间过长了，因此，我们将判断分支条件这一步前移到 ID 阶段，这样只需插入一个时钟周期的停顿，用来等待跳转地址被加载到 IF 阶段的 PC 中，再执行下一条指令。

另外，跳转指令（J 型）也需要等待跳转到的地址被加载到 IF 阶段的 PC 中，因而其后也要插入一个时钟周期的停顿。

在处理完冒险后，还要完善 CPU 控制器模块，使之能针对不同指令产生不同的完整的控制信号（其中，要注意指令的 R 型、I 型、S 型、B 型、U 型、J 型这些不同类型之间在控制信号上的差别，各类型指令格式如下图），并完善调用控制信号的数据通路。

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0	
funct7				rs2		rs1	funct3		rd			opcode		R-type	
imm[11:0]						rs1	funct3		rd			opcode		I-type	
imm[11:5]				rs2		rs1	funct3		imm[4:0]			opcode		S-type	
imm[12]	imm[10:5]			rs2		rs1	funct3		imm[4:1]	imm[11]	opcode		B-type		
imm[31:12]										rd			opcode		U-type
imm[20]	imm[10:1]			imm[11]		imm[19:12]			rd			opcode		J-type	

三、实验过程和数据记录

1. 完善控制信号模块

在这一模块中，完善代码，使得模块可以显示该指令是 B 型、S 型、U 型、J 型和 load 类指令中的哪一种具体指令：

```
wire BEQ = Bop & funct3_0;
wire BNE = Bop & funct3_1;
wire BLT = Bop & funct3_4;
```

```

wire BGE = Bop & funct3_5;
wire BLTU = Bop & funct3_6;
wire BGEU = Bop & funct3_7;

wire LB = Lop & funct3_0;
wire LH = Lop & funct3_1;
wire LW = Lop & funct3_2;
wire LBU = Lop & funct3_4;
wire LHU = Lop & funct3_5;

wire SB = Sop & funct3_0;
wire SH = Sop & funct3_1;
wire SW = Sop & funct3_2;

wire LUI = (opcode == 7'b0110111);
wire AUIPC = (opcode == 7'b0010111);

wire JAL = (opcode == 7'b1101111);
assign JALR = (opcode == 7'b1100111) & funct3_0;

```

在 B 型指令方面，外部有一个大小比较模块，通过对两个寄存器内的值进行大小比较，来判断是否满足分支跳转的大小条件，并将该结果输出。控制模块对其输出命令提供的大小关系条件（大于等于、小于或等于）；并从其接收是否满足跳转条件的判断结果，来输出信号控制程序是否跳转：

```

assign cmp_ctrl = funct3;

assign Branch = (B_valid&cmp_res) | (JAL&cmp_res) | (JALR&cmp_res);

```

ALU 的两个运算数的来源在不同类型的指令下是不同的，所以要输出对运算数来源进行选择的控制信号，信号的生成方式设计如下：

```

assign ALUSrc_A = AUIPC | JAL | JALR;

assign ALUSrc_B = L_valid | S_valid | I_valid | AUIPC | LUI;

```

控制模块还需要输出指令中是否含有 Rs1 寄存器和 Rs2 寄存器，并输出该指令可能面临的冒险类型（根据指令类型来区分），这些输出信号提供给冒险检测单元：

```

assign rsluse = R_valid | B_valid | JALR | L_valid | S_valid | I_valid;
assign rs2use = R_valid | S_valid | B_valid;

wire [1:0] alu_haz, ld_haz, sd_haz, no_haz;
assign no_haz = 2'b00;
assign alu_haz = 2'b01;
assign ld_haz = 2'b10;
assign sd_haz = 2'b11;
assign hazard_optype = {2{R_valid | I_valid | B_valid | JALR | AUIPC | JAL}} & alu_haz |
    {2{L_valid}} & ld_haz |
    {2{S_valid}} & sd_haz;

```

2. 大小比较模块

该模块会比较分支命令中两寄存器内值的大小关系，判断其大小关系是否满足命令提供的大小关系条件，是则输出 1，否则输出 0：

```
assign c = ((EQ & res_EQ) | (NE & res_NE) | (LT & res_LT)
| (LTU & res_LTU) | (GE & res_GE) | (GEU & res_GEU));
```

3. 冒险检测模块

该模块检测数据冒险是否会发生，并产生相应的数据前递信号（共三个，前递目标分别为 ID 阶段源寄存器 1、ID 阶段源寄存器 2 和 EXE 阶段 store 指令存储被写数据的源寄存器）和程序停顿的控制信号，使得数据冒险一旦发生，相应的数据前递和程序停顿操作能够进行。不同的前递操作和程序停顿操作的触发条件和执行方式已经在介绍实验原理中的数据冒险时有详细说明。另外，当 ID 阶段检测出具体代码实现如下：

```
module HazardDetectionUnit(
    input clk,
    input Branch_ID, rsluse_ID, rs2use_ID,
    input[1:0] hazard_optype_ID,
    input[4:0] rd_EXE, rd_MEM, rs1_ID, rs2_ID, rs2_EXE,

    output PC_EN_IF, reg_FD_EN, reg_FD_stall, reg_FD_flush,
           reg_DE_EN, reg_DE_flush, reg_EM_EN, reg_EM_flush, reg_MW_EN,
    output forward_ctrl_ls,
    output [1:0] forward_ctrl_A, forward_ctrl_B
);
    reg [1:0] hazard_optype_EXE, hazard_optype_MEM;
    always @(posedge clk) begin
        hazard_optype_MEM <= hazard_optype_EXE;
        hazard_optype_EXE <= {2{~reg_DE_flush}} & hazard_optype_ID;
    end

    wire forward_rs1_EXE = rsluse_ID & rs1_ID != 5'b0 & rs1_ID == rd_EXE
        & hazard_optype_EXE == 2'b01;
    wire forward_rs1_MEM = rsluse_ID & rs1_ID != 5'b0 & rs1_ID == rd_MEM
        & hazard_optype_MEM == 2'b01;
    wire forward_rs1_LD = rsluse_ID & rs1_ID != 5'b0 & rs1_ID == rd_MEM
        & hazard_optype_MEM == 2'b10;

    wire forward_rs2_EXE = rs2use_ID & rs2_ID != 5'b0 & rs2_ID == rd_EXE
        & hazard_optype_EXE == 2'b01;
    wire forward_rs2_MEM = rs2use_ID & rs2_ID != 5'b0 & rs2_ID == rd_MEM
        & hazard_optype_MEM == 2'b01;
    wire forward_rs2_LD = rs2use_ID & rs2_ID != 5'b0 & rs2_ID == rd_MEM
        & hazard_optype_MEM == 2'b10;

    wire stall = (rsluse_ID & rs1_ID != 5'b0 & rs1_ID == rd_EXE &
        hazard_optype_EXE == 2'b10 & hazard_optype_ID != 2'b11) |
        (rs2use_ID & rs2_ID != 5'b0 & rs2_ID == rd_EXE &
        hazard_optype_EXE == 2'b10 & hazard_optype_ID != 2'b11);

    assign reg_FD_stall = stall;

    assign PC_EN_IF = ~stall;
    assign reg_FD_EN = 1'b1;
    assign reg_DE_EN = 1'b1;
    assign reg_EM_EN = 1'b1;
    assign reg_MW_EN = 1'b1;

    assign reg_DE_flush = stall;
    assign reg_FD_flush = Branch_ID;
    assign reg_EM_flush = 1'b0;

    assign forward_ctrl_A = {2{forward_rs1_EXE}} & 2'b01 |
        {2{forward_rs1_MEM}} & 2'b10 |
        {2{forward_rs1_LD}} & 2'b11;
    assign forward_ctrl_B = {2{forward_rs2_EXE}} & 2'b01 |
        {2{forward_rs2_MEM}} & 2'b10 |
        {2{forward_rs2_LD}} & 2'b11;

    assign forward_ctrl_ls = (rs2_EXE == rd_MEM) & (hazard_optype_EXE == 2'b11)
```

```

                                & (hazard_optype_MEM == 2'b10); //sd after ld
endmodule

```

4. 数据通路连接（包括旁路单元）

根据之前指令的执行情况决定下一条指令的地址到底是紧随在后还是另作跳转：

```

MUX2T1_32 mux_IF(.IO(PC_4_IF),.I1(jump_PC_ID),.s(Branch_ctrl | JALR),.o(next_PC_IF));

```

读取前递控制信号，采取不同的前递操作（针对 ID 阶段源寄存器 1、

ID 阶段源寄存器 2 和 EXE 阶段 store 指令存储被写数据的源寄存器）：

```

MUX4T1_32 mux_forward_A(.IO(rs1_data_reg),.I1(ALUout_EXE),.I2(ALUout_MEM),.I3(Datain_MEM),
    .s(forward_ctrl_A),.o(rs1_data_ID));

```

```

MUX4T1_32 mux_forward_B(.IO(rs2_data_reg),.I1(ALUout_EXE),.I2(ALUout_MEM),.I3(Datain_MEM),
    .s(forward_ctrl_B),.o(rs2_data_ID));

```

```

MUX2T1_32
mux_forward_EXE(.IO(rs2_data_EXE),.I1(Datain_MEM),.s(forward_ctrl_ls),.o(Dataout_EXE));

```

选择运算数 A 是来自 PC 还是来自寄存器组：

```

MUX2T1_32 mux_A_EXE(.IO(rs1_data_EXE),.I1(PC_EXE),.s(ALUSrc_A_EXE),.o(ALUA_EXE));

```

选择运算数 B 是来自寄存器组还是来自立即数：

```

MUX2T1_32 mux_B_EXE(.IO(rs2_data_EXE),.I1(Imm_EXE),.s(ALUSrc_B_EXE),.o(ALUB_EXE));

```

5. 仿真和物理验证

采用如下指令：

NO.	Instruction	Addr.	Label	ASM
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
15	002350b3	3C		srl x1, x6, x2
16	402350b3	40		sra x1, x6, x2
17	4023d0b3	44		sra x1, x7, x2
18	007330b3	48		sltu x1, x6, x7
19	0063b0b3	4C		sltu x1, x7, x6

20	00000033	50		add x0,x0,x0
21	ffd50093	54		addi x1,x10,-3
22	00f27093	58		andi x1,x4,15
23	00f26093	5C		ori x1,x4,15
24	00f24093	60		xori x1,x4,15
25	00f22093	64		slti x1,x4,15
26	00121093	68		slli x1,x4,1
27	00225093	6C		srli x1,x4,2
28	40c35093	70		srai x1, x6, 12
29	fff33093	74		sltiu x1, x6, -1
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
45	00000013	B4		addi x0,x0,0
46	00000013	B8		addi x0,x0,0
47	00525863	BC	label3:	bge x4,x5,label4
48	0042d663	C0		bge x5,x4,label4
49	00000013	C4		addi x0,x0,0
50	00000013	C8		addi x0,x0,0
51	0063f863	CC	label4:	bgeu x7,x6,label5
52	00737663	D0		bgeu x6,x7,label5
53	00000013	D4		addi x0,x0,0
54	00000013	D8		addi x0,x0,0
55	00425663	DC	label5:	bge x4,x4,label6
56	00000013	E0		addi x0,x0,0
57	00000013	E4		addi x0,x0,0
58	000040b7	E8	label6:	lui x1,4
59	00c000ef	EC		jal x1,12

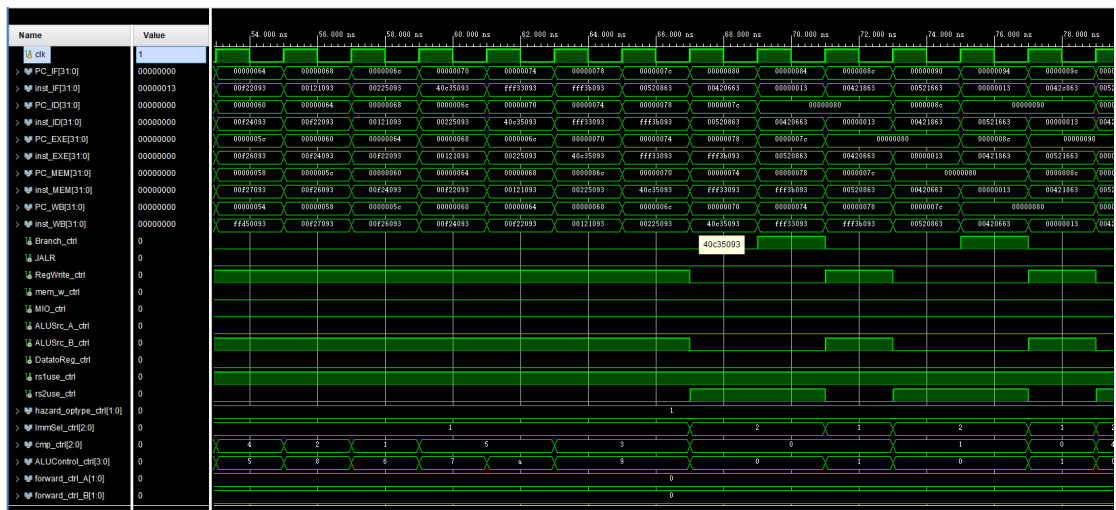
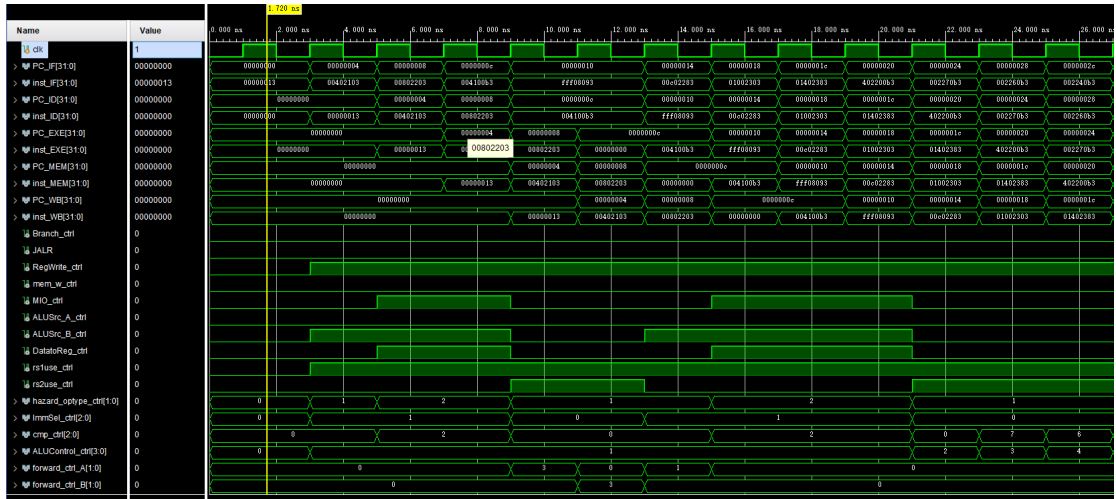
60	00000013	F0		addi x0,x0,0
61	00000013	F4		addi x0,x0,0
62	01802403	F8		lw x8, 24(x0)
63	00802e23	FC		sw x8, 28(x0)
64	01c02083	100		lw x1, 28(x0)
65	02801023	104		sh x8, 32(x0)
66	02002083	108		lw x1, 32(x0)
67	02800223	10C		sb x8, 36(x0)
68	02402083	110		lw x1, 36(x0)
69	01a01083	114		lh x1, 26(x0)
70	01a05083	118		lhu x1, 26(x0)
71	01b00083	11C		lb x1, 27(x0)
72	01b04083	120		lbu x1, 27(x0)
73	ffff0097	124		auipc x1, 0xffff0
74	000000e7	128		jalr x1,0(x0)

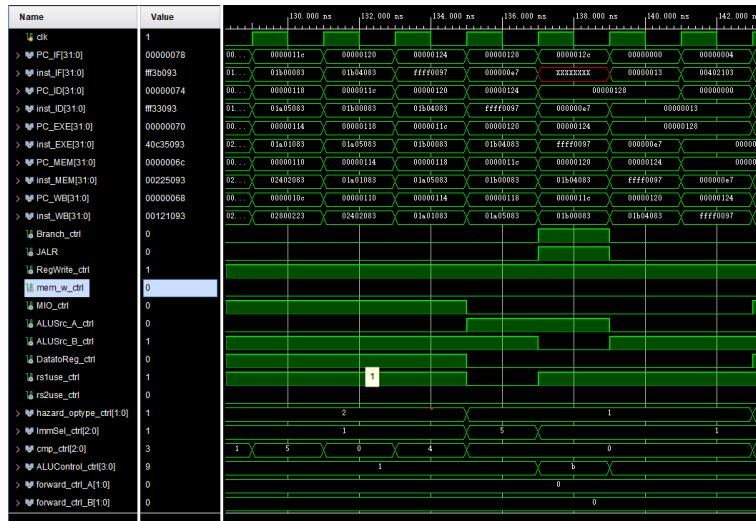
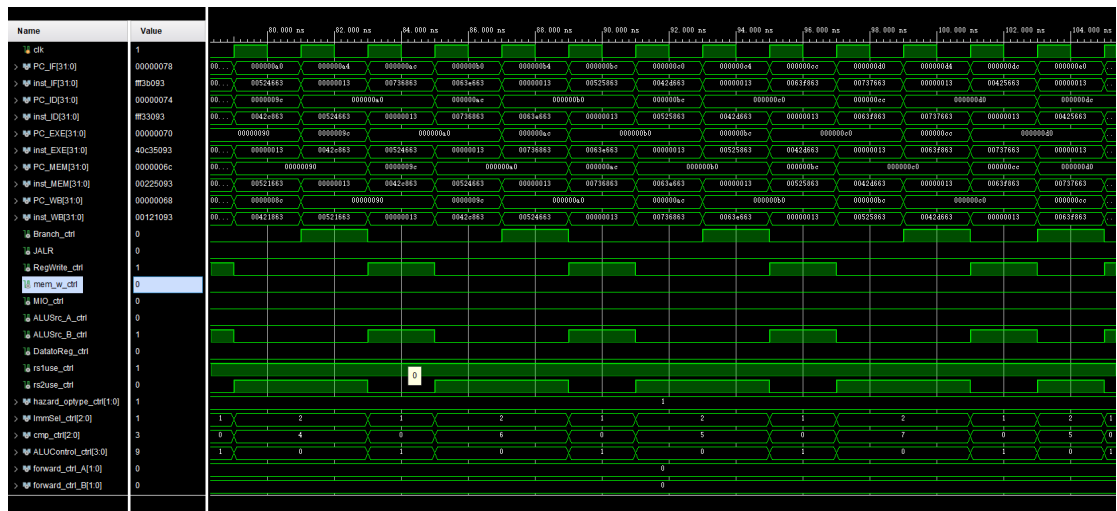
内存中存储的数据如下：

NO.	Data	Addr.	NO.	Data	Addr.
0	000080BF	0	16	00000000	40
1	00000008	4	17	00000000	44
2	00000010	8	18	00000000	48
3	00000014	C	19	00000000	4C
4	FFFF0000	10	20	A3000000	50
5	0FFF0000	14	21	27000000	54
6	FF000F0F	18	22	79000000	58
7	F0F0F0F0	1C	23	15100000	5C
8	00000000	20	24	00000000	60
9	00000000	24	25	00000000	64
10	00000000	28	26	00000000	68
11	00000000	2C	27	00000000	6C
12	00000000	30	28	00000000	70
13	00000000	34	29	00000000	74
14	00000000	38	30	00000000	78
15	00000000	3C	31	00000000	7C

四、实验结果分析

以下是仿真结果图：





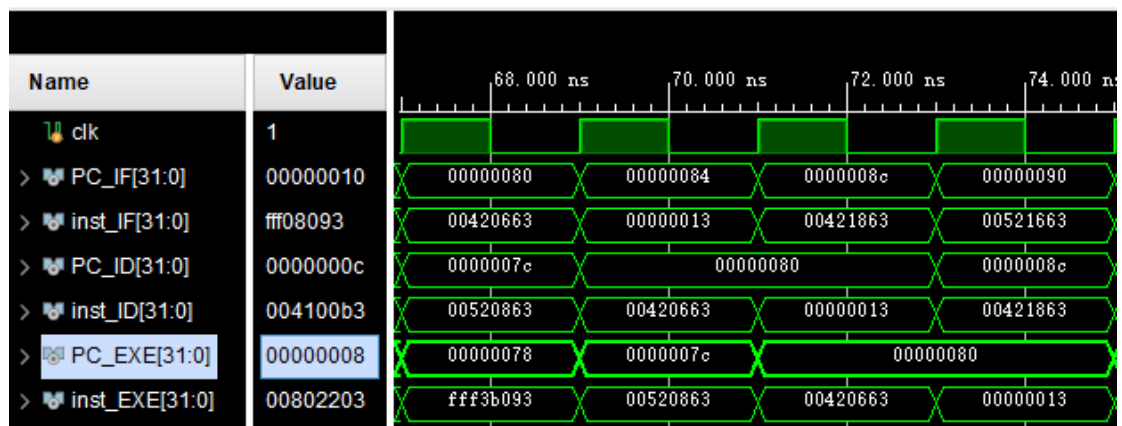
其中针对 CPU 运行的一些情况进行说明：

(1) 分支跳转时的单周期停顿：

当 PC_ID = 32'h00000080 时，ID 阶段的汇编指令为 beq x4,x4,label0，

该指令的分支跳转必定执行。可以看到，此时 ID 阶段停顿了一个周期，表

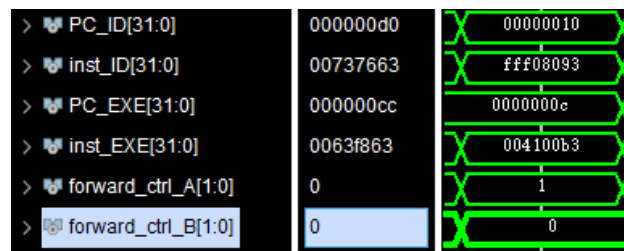
现为 PC_ID 不动，但指令内容被清除为 32'h00000013（即 add x0,x0,x0）；
同时 PC_IF 跳变为 label0 的地址 32'h0000008c。如下图。



(2) 前递各情况分析：

①. 从 EXE 阶段的非 load 指令前递至 ID 阶段

如右图。此时 EXE 阶段汇编指令为 add x1, x2, x4，而 ID 阶段汇编指令为



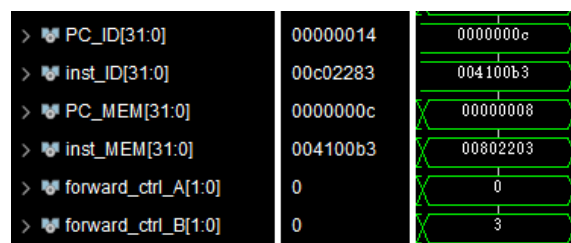
addi x1, x1, -1，此时 forward_ctrl_A 信号值为 1，表示将 EXE 阶段指令的运算结果前递到 ID 阶段的 A 数据源（原本读取了 x1 寄存器的值）进行替代，同时 forward_ctrl_B 信号值为 0，表示 ID 阶段的 B 数据源的值（立即数-1）不被前递替代。

②. 从 MEM 阶段的非 load 指令前递至 ID 阶段

这一点在测试程序中没有体现。

③. 从 MEM 阶段的 load 指令将内存数据前递至 ID 阶段

如右图。此时 MEM 阶段汇编指令为 lw x4, 8(x0)，而 ID 阶段汇编指令为 add x1, x2, x4，此时



forward_ctrl_B 信号值为 3，表示将 MEM 阶段从内存中读取出的数据前递到 ID 阶段的 B 数据源（原本读取了 x4 寄存器的值）进行替代，同时 forward_ctrl_B 信号值为 0，表示 ID 阶段的 B 数据源的值







(立即数-1) 不被前递替代。

- ④. 从 MEM 阶段的 load 指令将内存数据前递至 EXE 阶段的 store 指令

如右图。此时 MEM 阶段的指令为 lw x8, 24(x0)，而 EXE 阶段的指令为 sw x8, 28(x0)，此时 forward_ctrl_ls 信号值为 1，表示将 MEM 阶段从内存中读取出的数据前递到 EXE 阶段，替换 sw 指令从 x8 读出的要写进内存的值。

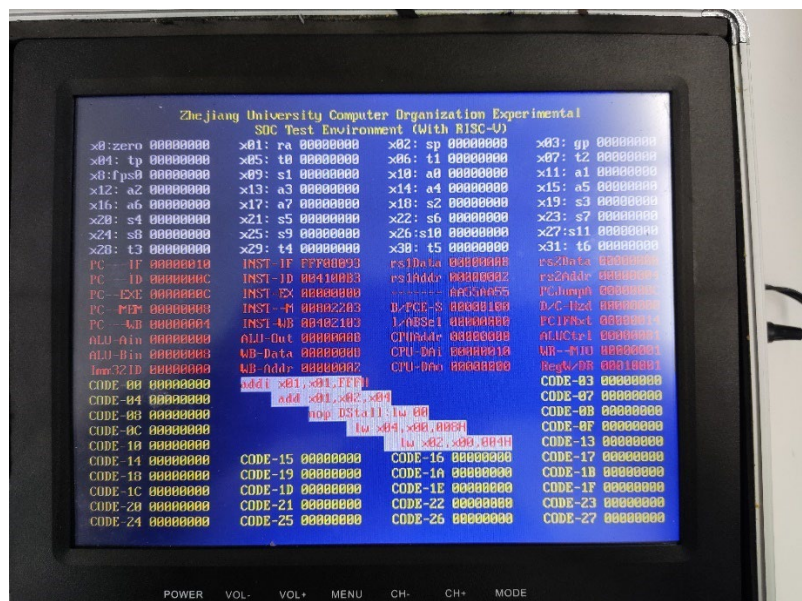
PC_EXE[31:0]	00000004	000000fc
inst_EXE[31:0]	00402103	00802e23
PC_MEM[31:0]	00000000	000000f8
inst_MEM[31:0]	00000013	01802403
forward_ctrl_ls	0	

(3) 数据冲突的停顿分析

> PC_ID[31:0]	00000014				
> inst_ID[31:0]	00c02283				
> PC_EXE[31:0]	00000010				
> inst_EXE[31:0]	fff08093				
> PC_MEM[31:0]	0000000c				
> inst_MEM[31:0]	004100b3				

如上图，当 32'h0000000c 处的指令 add x1, x2, x4 进入 ID 阶段时，在其之前进入 CPU 的指令 lw x4, 8(x0)还处于 EXE 阶段，而该指令要用来修改寄存器 x4 内容的值必须要到 MEM 阶段才能从内存中读取出来，所以这里在 ID 阶段处进行了一个时钟周期的停顿，在这个停顿过程中，指令 lw x4, 8(x0)仍然照常进入下一阶段，并用 32'h00000000（即 lw x0, 0(x0)）填充 EXE 阶段的指令空位。

在物理验证中的这一现象也表现了这一停顿行为。如下图。



在物理验证中，所有指令均能够正确执行。

五、讨论与心得

本次实验让我对流水线 CPU 的机理有了更深的理解。

在处理冒险时，对数据冒险尽可能采用了前递（旁路）的方式来解决，这相比于在上学期的计算机组成的实验中一味用停顿去解决，更好地保障了流水线 CPU 的效率。在具体实现前递（旁路）的过程中，我对这一操作的运行原理有了更深的理解。

在处理分支跳转指令的控制冒险时，在原本计算机组成的实验里，分支指令的条件判断和决定是否跳转这一步是在 MEM 阶段才完成的，因此要停顿 3 个时钟周期之后，才能知道在分支指令之后下一条应当加载的指令是什么。本实验将判断和决定的步骤前移到 ID 阶段，减少了 2 个需要停顿的周期，保障了流水线运行效率。

另外，本次实验没有对分支预测作要求，因而也没作实现。在 CPU 中实现分支预测，对流水线 CPU 的效率的影响，还难下定论，有待探究。