浙江大學

本科实验报告

课程名称：计算机组成与设计

姓　　名：张晋恺

学　　院：竺可桢学院

　　　系：所在系

专　　业：计算机科学与技术

学　　号：3230102400

指导教师：刘海风

2024 年 12 月 20 日

浙江大学实验报告

课程名称：计算机组成与设计　实验类型：综合

实验项目名称: 流水线 CPU 设计

学生姓名：张晋恺　专业：计算机科学与技术 学号：3230102400

同组学生姓名：　指导老师：刘海风

实验地点：东 4-512　实验日期：2024 年 12 月 19 日

一、操作方法与实验步骤

5-1 不解决冲突的 CPU 设计

在实验开始之前，我首先进行了 Datapath 的绘制，实现了不解决冲突的 CPU 设计。Datapath 如下:

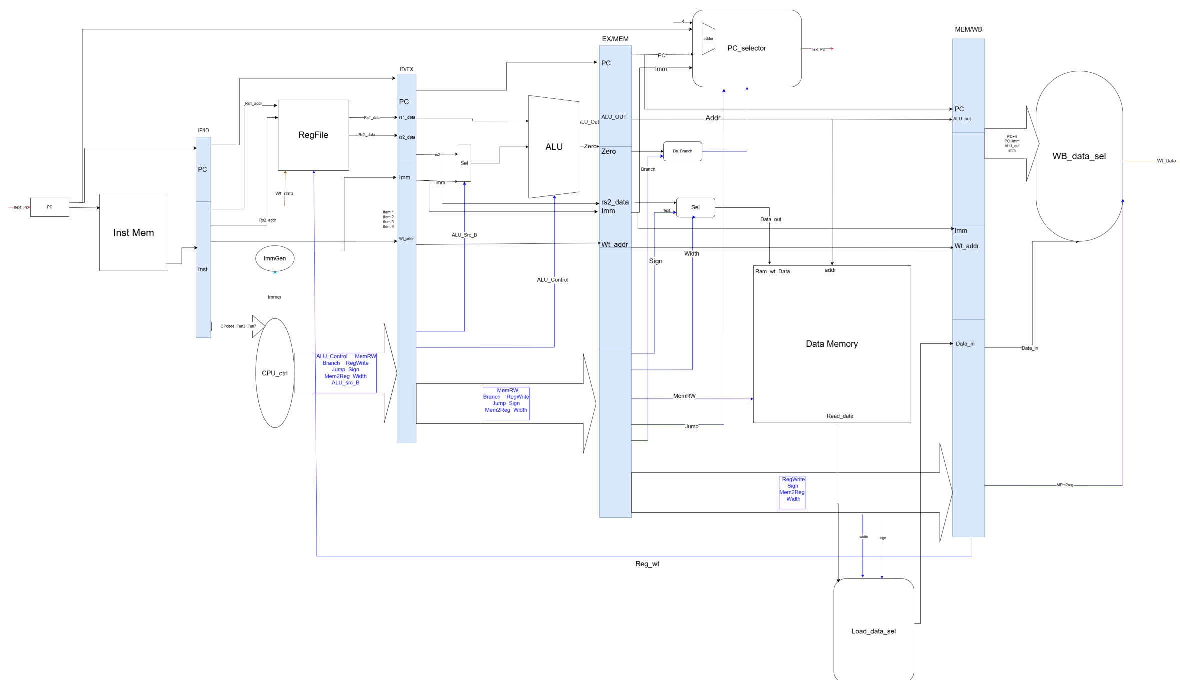

图 1: Datapath

对于不实现 hazard 的 CPU，实际上就是把单周期 CPU 的每个阶段分开，分为以下五个阶段

1. IF: 取指令阶段
2. ID: 指令译码阶段，对指令进行译码，得到指令对应的控制信号，立即数等
3. EX: 执行阶段，根据控制信号和立即数等，进行 ALU 运算
4. MEM: 内存访问阶段，根据控制信号, 进行内存访问，同时进行 Branch 和 Jump 指令的判断
5. WB: 写回阶段，将结果写回寄存器

加上阶段寄存器，然后进行流水线操作，这一部分比较简单，我直接在单周期 CPU 的基础上进行修改，得到了不实现 hazard 的 CPU；

5-2 解决数据冲突

在流水线中，主要存在以下三种冲突：

1. 结构冲突: 当内存访问与 IF 阶段同时进行，会发生冲突；这一部分我们已经通过使用不存的内存来存储指令和数据解决掉了；

2. 数据冲突: 当指令需要使用前一条指令的执行结果时，会发生数据冲突；这一部分我通过 forwarding 来解决

3. 控制冲突: 当 Branch 和 Jump 指令发生时，会发生控制冲突；这一部分我通过将 Branch 和 Jump 指令的执行提前到 ID 阶段来解决

接下来, 我逐步分析解决数据冲突和控制冲突的方法;

Forwarding 解决数据冲突

可能发生数据冲突的代码只有两种可能

1. ID 阶段指令 a 需要使用 EX 阶段指令 b 的执行结果;a 在 EX 阶段时需要 b 从 MEM 阶段前递, 称为 EX-MEM forwarding;

2. ID 阶段指令 a 需要使用 MEM 阶段指令 b 的执行结果;a 在 EX 阶段时需要 b 从 WB 阶段前递, 称为 EX-WB forwarding;

对于 EX-MEM forwarding, 当指令 b 为 load 指令时, 还需要额外插入一个 bubble, 等待 MEM 阶段执行完进入 WB 阶段;

对于 EX-WB forwarding, 则不需要额外插入 bubble;

其对应的图为

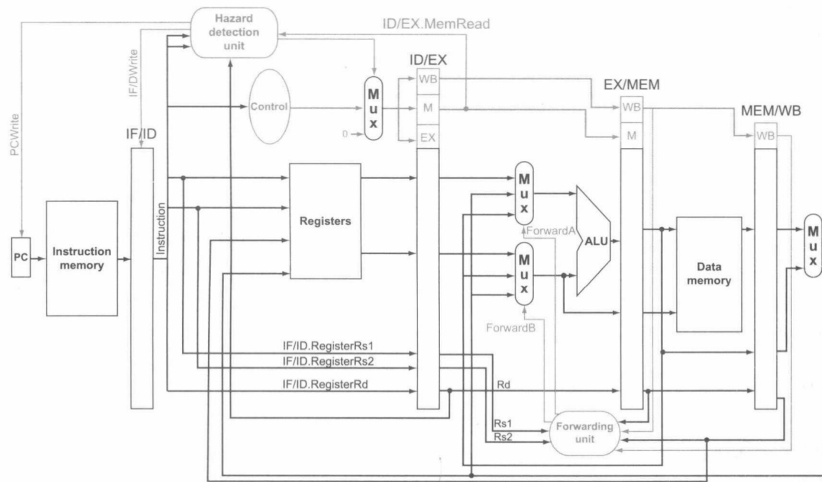

图 2: Forwarding

相关的代码为

```
`timescale 1ns / 1ps



module Forwarding(
    input [4:0] EX_MEM_rd,
    input [4:0] MEM_WB_rd,
    input [4:0] ID_EX_rs1,
    input [4:0] ID_EX_rs2,
    input       EX_MEM_RegWrite,
    input       MEM_WB_RegWrite,
    output reg [1:0] forward_rs1, // 00 for Regs, 01 for EX_MEM, 10 for MEM_WB
    output reg [1:0] forward_rs2 // 00 for Regs, 01 for EX_MEM, 10 for MEM_WB
    );


always @(*) begin
    //forwarding for rs1
    if (EX_MEM_RegWrite == 1 && EX_MEM_rd != 0 && EX_MEM_rd == ID_EX_rs1) begin
        forward_rs1 = 2'b01;//forward from EX_MEM
    end
    else if (MEM_WB_RegWrite == 1 && MEM_WB_rd != 0 && MEM_WB_rd == ID_EX_rs1) begin
        forward_rs1 = 2'b10;//forward from MEM_WB
    end
    else begin
        forward_rs1 = 2'b00;//there is no forwarding
    end
    //forwarding for rs2

    if (EX_MEM_RegWrite == 1 && EX_MEM_rd != 0 && EX_MEM_rd == ID_EX_rs2) begin
        forward_rs2 = 2'b01;//forward from EX_MEM
    end
    else if (MEM_WB_RegWrite == 1 && MEM_WB_rd != 0 && MEM_WB_rd == ID_EX_rs2) begin
        forward_rs2 = 2'b10;//forward from MEM_WB
    end
    else begin
        forward_rs2 = 2'b00;//there is no forwarding
    end
end


endmodule
```

可以看到，在这段代码中, 对 rs1 寄存器的 Forwarding 的条件为 (假设前面的指令为 a, 后面的指令为 b)

```
b.RegWrite == 1 && b.rd != 0 && b.rd == a.rs1
```

对 rs2 寄存器的 Forwarding 的条件也是类似的

```
b.RegWrite == 1 && b.rd != 0 && b.rd == a.rs2
```

并且我首先先判断 EX-MEM forwarding, **再判断** EX-WB forwarding; 这样做的好处是对于如下的例子

```
add x1 x1 x1
add x1 x1 x1
add x1 x1 x1
```

看似两种 Hazard 都发生了, 但实际上我们需要的是 EX-MEM forwarding, 因为这是最新的数据, 所以先进行 EX-MEM forwarding 能保证满足 else if 的判断不会发生；

对于上面提到的 EX-MEM forwarding, 需要插入 bubble 的代码判断如下

```
assign bubble2 = ID_EX_MemRead && (ID_EX_Wt_addr!=0) &&
↪  ((ID_EX_Wt_addr==IF_ID_inst_field[19:15])||(ID_EX_Wt_addr==IF_ID_inst_field[24:20]));
```

bubble 在 b 在 EX,a 在 ID 阶段被检测到，所谓插入 bubble, 就是下一个时钟周期到来时，指令 a 停留在 ID 阶段，指令 b 前进, 将 ID_EX 阶段寄存器控制信号设置为 0；具体来说是以下部分代码

```
always @(posedge clk or posedge rst) begin
    if(rst) begin
        ...
    end
    else begin
        if(bubble) begin//if bubble, do not update PC
            PC <= PC;
            IF_ID_PC <= IF_ID_PC;
            IF_ID_inst_field <= IF_ID_inst_field;
        end
        else if ...
    end
end

...


always @(posedge clk or posedge rst) begin
    if(rst) begin
        ...
    end
    else begin
        if(bubble) begin//set all control signal to 0
           ID_EX_ALU_Control <= 4'b0;
           ID_EX_Branch <= 4'b0;
           ID_EX_MemtoReg <= 3'b0;
           ID_EX_Jump <= 2'b0;
           ID_EX_ALUSrc_B <= 1'b0;
           ID_EX_RegWrite <= 1'b0;
           ID_EX_MemRW <= 1'b0;
           ID_EX_sign <= 1'b0;
           ID_EX_MemRead <= 1'b0;
           ID_EX_width <= 2'b0;
        end
```

```
        else begin
            ...
        end
    end
end
```

注：上述代码只保留了关键部分

有了 Forwarding，接下来只需要在 EX 阶段将译码出来的 rs1 数据和 rs2 数据设置为经过 forwarding 的数据，再进行 ALU 运算即可；同时将 Forwarding 过的 rs2 数据传递到 MEM 阶段；

具体代码如下

```
always @(*) begin
    case(forward_rs1)
        2'b01: rs1_temp = Wt_data_temp;//EX/MEM
        2'b10: rs1_temp = Wt_data;//MEM/WB
        default: rs1_temp = ID_EX_rs1_data;//No forwarding
    endcase
    case(forward_rs2)
        2'b01: rs2_temp = Wt_data_temp;
        2'b10: rs2_temp = Wt_data;
        default: rs2_temp = ID_EX_rs2_data;
    endcase
end

wire [31:0] A,B,ALU_out;
wire zero;
assign A = rs1_temp;
assign B = ID_EX_ALUSrc_B ? ID_EX_imm_data : rs2_temp;
ALU alu(
    .A(A),
    .B(B),
    .ALU_operation(ID_EX_ALU_Control),
    .res(ALU_out),
    .zero(zero)
);
...

always @(posedge clk or posedge rst) begin
    ...
    EX_MEM_rs2_data <= rs2_temp;
    ...
    ...
end
```

这样，我们的数据冲突就解决了；

解决控制冲突

对于控制冲突，我的解决方案是这样的：

将 Branch 和 Jump 指令的执行提前到 ID 阶段，在这一阶段，我设置了一个判断当前指令是否为 BJ-type 的的信号；如果该信号为 1 说明是跳转指令，那么下一个时钟上升沿到达时, 紧跟该 BJ-type 的下一条指令将会被 FLush 掉 (变为 nop) 指令,PC 变为 BJ 指令的指示的下一个 PC(如果做, 那么需要变为 BJ 指令的跳转地址，如果不做，则变为被 FLush 掉的指令的地址)

即我总是认为 BJ-type 指令是执行的;

由于提前了，仍然需要做 forwarding, 但是与数据冲突略有不同, 因为这里只需要对 MEM 阶段进行前递, 如果是 WB 阶段，利用 RegFile 下降沿写的 double pump 可以保证译码结果正确, 如果是 EX 阶段, 则插入 bubble 使其进入 MEM 阶段;

具体代码如下

```
//ID stage do BJ detection
assign is_Branch = (Branch[0] | Branch[1] | Branch[2] | Branch[3]);
assign is_BJ = (Jump[1] | is_Branch);

assign is_fw1 = is_BJ && EX_MEM_RegWrite && (EX_MEM_Wt_addr != 0) && (EX_MEM_Wt_addr ==
↪  IF_ID_inst_field[19:15]);
assign is_fw2 = is_BJ && EX_MEM_RegWrite && (EX_MEM_Wt_addr != 0) && (EX_MEM_Wt_addr ==
↪  IF_ID_inst_field[24:20]);

assign fw_rs1_data = is_fw1 ? Wt_data_temp : rs1_data;//forwarding from MEM stage
assign fw_rs2_data = is_fw2 ? Wt_data_temp : rs2_data;//forwarding from MEM stage

wire do_branch;
wire [31:0] ID_ALU_res;
wire ID_zero;
wire [31:0] PC_remain;

ALU  alu_branch(
    .A(fw_rs1_data),
    .B(fw_rs2_data),
    .ALU_operation(ALU_Control),
    .res(ID_ALU_res),
    .zero(ID_zero)
);

assign do_branch= (Branch[0] & ID_zero) | (Branch[1] & ~ID_zero) | (Branch[2] &
↪  ID_ALU_res[0]) | (Branch[3] & ~ID_ALU_res[0]);
assign PC_remain = is_BJ? PC : PC + 4;

wire [31:0] jalr_addr;
assign jalr_addr = fw_rs1_data + imm_data;
wire [31:0] jump_addr;
assign jump_addr = IF_ID_PC + imm_data;

assign next_PC =  Jump[1]?
           (Jump[0]? jalr_addr : jump_addr) :
           (do_branch ? (IF_ID_PC + imm_data) : PC_remain);

// prepare next pc
...
```

```
assign bubble1 = is_BJ && ID_EX_RegWrite && (ID_EX_Wt_addr!=0) &&
    ((ID_EX_Wt_addr==IF_ID_inst_field[19:15])||(ID_EX_Wt_addr==IF_ID_inst_field[24:20]));

assign bubble = bubble1|bubble2;
```

值得一提的是，在这里我引入了一个额外的 ALU, 用于判断 Branch 指令是否需要跳转;

然后，在选择 nextPC 的阶段, 最后不做 BJ-type 指令的情况有两种，一种是这条指令是 BJ-type 指令，但是没有跳转，另一种是这条指令不是 BJ-type 指令;

所以 PCremain 就是判断当前指令是否为 BJ-type 指令，如果为 1，则 PCremain 为不满足跳转的条件，下一个 PC 为被 flush 掉的指令的地址 (即当前 PC)，否则, 这条指令不是 BJ-type 指令，那么 PCremain 为当前 PC+4;

最后，在这里引入的 bubble1 与前面 forwarding 引入的 bubble2 进行或运算，得到最后总体的 bubble 信号;

对指令 flush 的代码如下

```

always @(posedge clk or posedge rst) begin
    if(rst) begin
        ...
    end
    else begin
        if(bubble) begin//if bubble, do not update PC
            ...
        end
        else if (is_BJ) begin
            PC <= next_PC;
            IF_ID_PC <= PC;
            IF_ID_inst_field <= 32'h00000013;//nop
        end
        else begin
            ...//normal process
        end
    end
end
```

即 PC 赋值为 BJ-type 选出来的 nextPC,IF_ID_inst_field 赋值为 nop,IF_ID_PC 赋值为 PC;

仿真测试

除了利用验收代码的仿真之外, 我还使用了以下代码进行了仿真测试

```
    auipc x1, 0
j      pass_0              # 00
dummy:
nop                      # 04
nop                      # 08
nop                      # 0C
nop                      # 10
nop                      # 14
nop                      # 18
nop                      # 1C
j      dummy

pass_0:
li     x31, 1
li     x1, -1            # x1=FFFFFFFF
xori   x3, x1, 1         # x3=FFFFFFFE
# use-use hazard
add    x3, x3, x3        # x3=FFFFFFFC
add    x3, x3, x3        # x3=FFFFFFF8
add    x3, x3, x3        # x3=FFFFFFF0
# Mem stage use-use hazard
xori   x4, x1, 1         # x4=FFFFFFFE
add    x5, x4, x3        # x5=FFFFFFEE
addi   x5, x5, 2         # x5=FFFFFFF0
beq    x5, x3, pass_1
auipc x30, 0
j      dummy

pass_1:
li   x31, 2
# load-use hazard
li   x20, 0x20
sw   x5, 0(x20)
lh   x6, 0(x20)            # x6=FFFFFFF0
add  x6, x6, x6            # x6=FFFFFFE0
addi x6, x6, 0x10          # x6=FFFFFFF0
beq  x6, x5, pass_2
```

```
38 auipc x30, 0
39 j    dummy
40
41 pass_2:
42 li   x31, 3
43 # use-store hazard
44 add  x6, x6, x6          # x6=FFFFFFE0
45 sw   x6, 0(x20)
46 lw   x7, 0(x20)          # x7=FFFFFFE0
47 bne  x7, x5, pass_3
48 auipc x30, 0
49 j    dummy
50
51 pass_3:
52 li   x31, 4
53 li   x1,  4
54 loop:
55 addi x1, x1, -1
56 bne  x1, x0, loop
57 beq  x1, x0, pass_4
58 auipc x30, 0
59 j    dummy
60
61 pass_4:
62 lui  x3, 0x80000     # x3=80000000
63 add  x3, x1, x1      # x3=00000000
64 beq  x3, x0, pass_5
65 auipc x30, 0
66 j    dummy
67
68 pass_5:
69 li   x31, 5
70 addi x6, x6, 0x20    # x6=FFFFFF00
71 sw   x6, 0(x20)
72 lb   x7, 0(x20)      # x7=00000000
73 beq  x7, x0, pass_6
74 auipc x30, 0
75 j    dummy
```

```
76
77 pass_6:
78 li      x31, 0x666
79 j       dummy
```

具体的仿真结果在下一章节分析;

最后是项目的源代码 (除去已经展示过的 Forwarding 模块,ALU 和 Regfile,CPUctrl 模块)

```
1  `timescale 1ns / 1ps
2
3
4  `include "SCPU_header.vh"
5
6  module MyScpu(
7      input clk,
8      input rst,
9      // input MIO_ready,
10     input [31:0] inst_in,
11     input [31:0] Data_in,
12     `RegFile_Regs_Outputs
13     `Pip_Regs_Outputs
14     output reg [3:0] RAM_wt_bits,
15     output reg [31:0] Data_out,
16     output reg [31:0] PC_out,
17     output reg [31:0] Addr_out,
18     output reg [31:0] Wt_data,
19     output reg MemRW
20     // output reg CPU_MIO
21     );
22
23 //Instruction Fetch
24 // IF/ID (PC,inst)
25 reg [31:0] PC;
26 wire [31:0] next_PC;
27
28 always @(*) begin
29     PC_out = PC;
30 end
31
32 always @(posedge clk or posedge rst) begin
33     if(rst) begin
34         PC <= 32'h0;
35         IF_ID_PC <= 32'h0;
36         IF_ID_inst_field <= 32'h0;
37     end
38     else begin
39         if(bubble) begin//if bubble, do not update PC
40             PC <= PC;
41             IF_ID_PC <= IF_ID_PC;
42             IF_ID_inst_field <= IF_ID_inst_field;
43         end
44         else if (is_BJ) begin
```

```
45                 PC <= next_PC;
46                 IF_ID_PC <= PC;
47                 IF_ID_inst_field <= 32'h00000013;//nop
48             end
49             else begin
50                 PC <= next_PC;
51                 IF_ID_PC <= PC;
52                 IF_ID_inst_field <= inst_in;
53             end
54         end
55     end
56
57     //end of Instruction Fetch
58
59
60
61     // Instruction Decode
62
63     wire [31:0] rs1_data, rs2_data, imm_data;
64     wire [31:0] rs1_index, rs2_index;
65     wire [3:0]  ALU_Control;
66     wire [3:0]  Branch;
67     wire [2:0]  MemtoReg, ImmSel;
68     wire [1:0]  Jump;
69     wire [1:0]  width;
70     wire        ALUSrc_B, RegWrite, MemRW_temp,sign;
71     wire        Memread_temp;
72     wire        is_BJ;
73     wire        is_Branch;
74     wire [31:0] fw_rs1_data, fw_rs2_data;//forwarding for ID stage
75     wire        is_fw1, is_fw2;//forwarding for ID stage
76
77
78     assign rs1_index = IF_ID_inst_field[19:15];
79     assign rs2_index = IF_ID_inst_field[24:20];
80
81     Regs regfile(
82         .clk(clk),
83         .rst(rst),
84         .RegWrite(MEM_WB_RegWrite),
85         .Rs1_addr(rs1_index),
86         .Rs2_addr(rs2_index),
87         .Wt_addr(MEM_WB_Wt_addr),
88         .Wt_data(Wt_data),
89         `RegFile_Regs_Arguments
90         .Rs1_data(rs1_data),
91         .Rs2_data(rs2_data)
92     );
93
94     ScpuCtrl ctrl(
95         .OPcode(IF_ID_inst_field[6:2]),
96         .Fun3(IF_ID_inst_field[14:12]),
97         .Fun7(IF_ID_inst_field[30]),
98         // .MIO_ready(MIO_ready),
99         .ImmSel(ImmSel),
100        .ALUSrc_B(ALUSrc_B),
101        .MemtoReg(MemtoReg),
```

```
102      .Jump(Jump),
103      .Branch(Branch),
104      .RegWrite(RegWrite),
105      .MemRW(MemRW_temp),
106      .ALU_Control(ALU_Control),
107      // .CPU_MIO(CPU_MIO),
108      .signal(sign),
109      .width(width),
110      .Memread(Memread_temp)//for stall
111  );
112
113  // ID stage do BJ detection
114  assign is_Branch = (Branch[0] | Branch[1] | Branch[2] | Branch[3]);
115  assign is_BJ = (Jump[1] | is_Branch);
116
117  assign is_fw1 = is_BJ && EX_MEM_RegWrite && (EX_MEM_Wt_addr != 0) && (EX_MEM_Wt_addr ==
     ↪   IF_ID_inst_field[19:15]);
118  assign is_fw2 = is_BJ && EX_MEM_RegWrite && (EX_MEM_Wt_addr != 0) && (EX_MEM_Wt_addr ==
     ↪   IF_ID_inst_field[24:20]);
119
120  assign fw_rs1_data = is_fw1 ? Wt_data_temp : rs1_data;//forwarding from MEM stage
121  assign fw_rs2_data = is_fw2 ? Wt_data_temp : rs2_data;//forwarding from MEM stage
122
123  wire do_branch;
124  wire [31:0] ID_ALU_res;
125  wire ID_zero;
126  wire [31:0] PC_remain;
127
128  ALU  alu_branch(
129      .A(fw_rs1_data),
130      .B(fw_rs2_data),
131      .ALU_operation(ALU_Control),
132      .res(ID_ALU_res),
133      .zero(ID_zero)
134  );
135
136  assign do_branch= (Branch[0] & ID_zero) | (Branch[1] & ~ID_zero) | (Branch[2] &
     ↪   ID_ALU_res[0]) | (Branch[3] & ~ID_ALU_res[0]);
137  // assign do_branch = (EX_MEM_Branch[0] & EX_MEM_ALU_zero) | (EX_MEM_Branch[1] &
     ↪   ~EX_MEM_ALU_zero) | (EX_MEM_Branch[2] & EX_MEM_ALU_out[0]) | (EX_MEM_Branch[3] &
     ↪   ~EX_MEM_ALU_out[0]);
138  assign PC_remain = is_BJ? PC : PC + 4;
139
140  wire [31:0] jalr_addr;
141  assign jalr_addr = fw_rs1_data + imm_data;
142  wire [31:0] jump_addr;
143  assign jump_addr = IF_ID_PC + imm_data;
144
145  assign next_PC =  Jump[1]?
146              (Jump[0]? jalr_addr : jump_addr) :
147              (do_branch ? (IF_ID_PC + imm_data) : PC_remain);
148
149  // prepare next pc
150
151  Immgen imm(
152      .inst_field(IF_ID_inst_field),
153      .ImmSel(ImmSel),
```

```
154        .Imm_out(imm_data)
155    );
156
157    wire bubble;
158
159    wire bubble1;
160    wire bubble2;
161
162    assign bubble1 = is_BJ && ID_EX_RegWrite && (ID_EX_Wt_addr!=0) &&
       ↪  ((ID_EX_Wt_addr==IF_ID_inst_field[19:15])||(ID_EX_Wt_addr==IF_ID_inst_field[24:20]));
163
164
165
166    //ID/EX (PC,rs1_data,rs2_data,imm_data,ALU_Control,Wt_addr
167    //,Branch,MemtoReg,Jump,ALUSrc_B,RegWrite,MemRW,sign)
168    always @(posedge clk or posedge rst) begin
169        if(rst) begin
170            ID_EX_PC <= 32'h0;
171            ID_EX_rs1 <= 5'b0;
172            ID_EX_rs2 <= 5'b0;
173            ID_EX_rs1_data <= 32'h0;
174            ID_EX_rs2_data <= 32'h0;
175            ID_EX_imm_data <= 32'h0;
176            ID_EX_ALU_Control <= 4'b0;
177            ID_EX_Wt_addr <= 5'b0;
178            ID_EX_Branch <= 4'b0;
179            ID_EX_MemtoReg <= 3'b0;
180            ID_EX_Jump <= 2'b0;
181            ID_EX_ALUSrc_B <= 1'b0;
182            ID_EX_RegWrite <= 1'b0;
183            ID_EX_MemRW <= 1'b0;
184            ID_EX_sign <= 1'b0;
185            ID_EX_MemRead <= 1'b0;
186            ID_EX_width <= 2'b0;
187        end
188        else begin
189            if(bubble) begin//set all control signal to 0
190               ID_EX_ALU_Control <= 4'b0;
191               ID_EX_Branch <= 4'b0;
192               ID_EX_MemtoReg <= 3'b0;
193               ID_EX_Jump <= 2'b0;
194               ID_EX_ALUSrc_B <= 1'b0;
195               ID_EX_RegWrite <= 1'b0;
196               ID_EX_MemRW <= 1'b0;
197               ID_EX_sign <= 1'b0;
198               ID_EX_MemRead <= 1'b0;
199               ID_EX_width <= 2'b0;
200            end
201            else begin
202                ID_EX_PC <= IF_ID_PC;
203                ID_EX_rs1 <= IF_ID_inst_field[19:15];
204                ID_EX_rs2 <= IF_ID_inst_field[24:20];
205                ID_EX_rs1_data <= rs1_data;
206                ID_EX_rs2_data <= rs2_data;
207                ID_EX_imm_data <= imm_data;
208                ID_EX_Wt_addr <= IF_ID_inst_field[11:7];
209                ID_EX_ALU_Control <= ALU_Control;
```

```
210                 ID_EX_Branch <= Branch;
211                 ID_EX_MemtoReg <= MemtoReg;
212                 ID_EX_Jump <= Jump;
213                 ID_EX_width <= width;
214                 ID_EX_ALUSrc_B <= ALUSrc_B;
215                 ID_EX_RegWrite <= RegWrite;
216                 ID_EX_MemRW <= MemRW_temp;
217                 ID_EX_MemRead <= Memread_temp;
218                 ID_EX_sign <= sign;
219             end
220         end
221     end
222 
223     //end of Instruction Decode
224 
225     assign bubble2 = ID_EX_MemRead && (ID_EX_Wt_addr!=0) &&
    ↪   ((ID_EX_Wt_addr==IF_ID_inst_field[19:15])||(ID_EX_Wt_addr==IF_ID_inst_field[24:20]));
226 
227     assign bubble = bubble1 | bubble2;
228 
229     //Execution
230 
231     wire [1:0] forward_rs1, forward_rs2;
232 
233     Forwarding fw_inst(
234         .EX_MEM_rd(EX_MEM_Wt_addr),
235         .MEM_WB_rd(MEM_WB_Wt_addr),
236         .ID_EX_rs1(ID_EX_rs1),
237         .ID_EX_rs2(ID_EX_rs2),
238         .EX_MEM_RegWrite(EX_MEM_RegWrite),
239         .MEM_WB_RegWrite(MEM_WB_RegWrite),
240         .forward_rs1(forward_rs1),
241         .forward_rs2(forward_rs2)
242     );
243 
244     reg [31:0] rs1_temp, rs2_temp;
245 
246     always @(*) begin
247         case(forward_rs1)
248             2'b01: rs1_temp = Wt_data_temp;//EX/MEM
249             2'b10: rs1_temp = Wt_data;//MEM/WB
250             default: rs1_temp = ID_EX_rs1_data;//No forwarding
251         endcase
252         case(forward_rs2)
253             2'b01: rs2_temp = Wt_data_temp;
254             2'b10: rs2_temp = Wt_data;
255             default: rs2_temp = ID_EX_rs2_data;
256         endcase
257     end
258 
259     wire [31:0] A,B,ALU_out;
260     wire zero;
261     assign A = rs1_temp;
262     assign B = ID_EX_ALUSrc_B ? ID_EX_imm_data : rs2_temp;
263     ALU alu(
264         .A(A),
265         .B(B),
```

```
    .ALU_operation(ID_EX_ALU_Control),
    .res(ALU_out),
    .zero(zero)
);


//EX/MEM (PC,ALU_out(mem_rw_addr),rs2_data(For s-type),
//imm_data(lui auipc),Wt_addr(wb),Branch,MemtoReg,Jump,RegWrite,
//MemRW,sign,ALU_zero)


always @(posedge clk or posedge rst) begin
    if(rst) begin
        EX_MEM_PC <= 32'b0;
        EX_MEM_ALU_out <= 32'b0;
        EX_MEM_imm_data <= 32'b0;
        EX_MEM_Wt_addr <= 5'b0;
        EX_MEM_rs2_data <= 4'b0;
        EX_MEM_Branch <= 4'b0;
        EX_MEM_MemtoReg <= 3'b0;
        EX_MEM_Jump <= 2'b0;
        EX_MEM_width <= 2'b0;
        EX_MEM_RegWrite <= 1'b0;
        EX_MEM_MemRW <= 1'b0;
        EX_MEM_sign <= 1'b0;
        EX_MEM_ALU_zero <= 1'b0;
    end
    else begin
        EX_MEM_PC <= ID_EX_PC;
        EX_MEM_ALU_out <= ALU_out;
        EX_MEM_rs2_data <= rs2_temp;
        EX_MEM_imm_data <= ID_EX_imm_data;
        EX_MEM_Wt_addr <= ID_EX_Wt_addr;
        EX_MEM_Branch <= ID_EX_Branch;
        EX_MEM_MemtoReg <= ID_EX_MemtoReg;
        EX_MEM_Jump <= ID_EX_Jump;
        EX_MEM_width <= ID_EX_width;
        EX_MEM_RegWrite <= ID_EX_RegWrite;
        EX_MEM_MemRW <= ID_EX_MemRW;
        EX_MEM_sign <= ID_EX_sign;
        EX_MEM_ALU_zero <= zero;
    end
end
//end of Execution

//Memory Access

// wire  do_branch;



// sb sh sw choose data
always @(*) begin
    Addr_out = EX_MEM_ALU_out;
    MemRW = EX_MEM_MemRW;
    case({EX_MEM_sign, EX_MEM_width, EX_MEM_ALU_out[1:0]})
       {`FUNC_BYTE, `MOD_ZERO}: begin
```

```
           Data_out = {24'b0, EX_MEM_rs2_data[7:0]};
           RAM_wt_bits = 4'b0001;
       end
       {`FUNC_BYTE, `MOD_ONE}: begin
           Data_out = {16'b0, EX_MEM_rs2_data[7:0], 8'b0};
           RAM_wt_bits = 4'b0010;
       end
       {`FUNC_BYTE, `MOD_TWO}: begin
           Data_out = {8'b0, EX_MEM_rs2_data[7:0], 16'b0};
           RAM_wt_bits = 4'b0100;
       end
       {`FUNC_BYTE, `MOD_THREE}: begin
           Data_out = {EX_MEM_rs2_data[7:0], 24'b0};
           RAM_wt_bits = 4'b1000;
       end
       {`FUNC_HALF, `MOD_ZERO}: begin
           Data_out = {16'b0, EX_MEM_rs2_data[15:0]};
           RAM_wt_bits = 4'b0011;
       end
       {`FUNC_HALF, `MOD_ONE}: begin
           Data_out = {8'b0, EX_MEM_rs2_data[15:0], 8'b0};
           RAM_wt_bits = 4'b0110;
       end
       {`FUNC_HALF, `MOD_TWO}: begin
           Data_out = {EX_MEM_rs2_data[15:0], 16'b0};
           RAM_wt_bits = 4'b1100;
       end
       {`FUNC_WORD, `MOD_ZERO}: begin
           Data_out = EX_MEM_rs2_data;
           RAM_wt_bits = 4'b1111;
       end
       default: begin
           Data_out = 32'b0;
           RAM_wt_bits = 4'b0000;
       end
    endcase
    end
//pass data to next stage

reg [31:0] Mem_data_in;
always @(*) begin
    case({EX_MEM_sign, EX_MEM_width, EX_MEM_ALU_out[1:0]})
        {`FUNC_BYTE, `MOD_ZERO}: Mem_data_in = {{24{Data_in[7]}}, Data_in[7:0]};
        {`FUNC_BYTE, `MOD_ONE}: Mem_data_in = {{24{Data_in[15]}}, Data_in[15:8]};
        {`FUNC_BYTE, `MOD_TWO}: Mem_data_in = {{24{Data_in[23]}}, Data_in[23:16]};
        {`FUNC_BYTE, `MOD_THREE}: Mem_data_in = {{24{Data_in[31]}}, Data_in[31:24]};
        {`FUNC_HALF, `MOD_ZERO}: Mem_data_in = {{16{Data_in[15]}}, Data_in[15:0]};
        {`FUNC_HALF, `MOD_ONE}: Mem_data_in = {{16{Data_in[23]}}, Data_in[23:8]};
        {`FUNC_HALF, `MOD_TWO}: Mem_data_in = {{16{Data_in[31]}}, Data_in[31:16]};
        {`FUNC_WORD, `MOD_ZERO}: Mem_data_in = Data_in;
        {`FUNC_BYTE_UNSIGNED, `MOD_ZERO}: Mem_data_in = {{24{1'b0}}, Data_in[7:0]};
        {`FUNC_BYTE_UNSIGNED, `MOD_ONE}: Mem_data_in = {{24{1'b0}}, Data_in[15:8]};
        {`FUNC_BYTE_UNSIGNED, `MOD_TWO}: Mem_data_in = {{24{1'b0}}, Data_in[23:16]};
        {`FUNC_BYTE_UNSIGNED, `MOD_THREE}: Mem_data_in = {{24{1'b0}}, Data_in[31:24]};
        {`FUNC_HALF_UNSIGNED, `MOD_ZERO}: Mem_data_in = {{16{1'b0}}, Data_in[15:0]};
        {`FUNC_HALF_UNSIGNED, `MOD_ONE}: Mem_data_in = {{16{1'b0}}, Data_in[23:8]};
        {`FUNC_HALF_UNSIGNED, `MOD_TWO}: Mem_data_in = {{16{1'b0}}, Data_in[31:16]};
```

```
380             default: Mem_data_in = 32'b0;
381         endcase
382     end
383 
384 
385     //prepare for wt_data
386     reg [31:0] Wt_data_temp;
387     always @(*) begin
388         case (EX_MEM_MemtoReg)
389             `MEM2REG_ALU: Wt_data_temp = EX_MEM_ALU_out;
390             `MEM2REG_MEM: Wt_data_temp = Mem_data_in;
391             `MEM2REG_PC_PLUS: Wt_data_temp = EX_MEM_PC + 4;
392             `MEM2REG_LUI: Wt_data_temp = EX_MEM_imm_data;
393             `MEM2REG_AUIPC: Wt_data_temp = EX_MEM_imm_data + EX_MEM_PC;
394             default: Wt_data_temp = 32'b0;
395         endcase
396     end
397 
398 
399     //MEM/WB (PC,ALU_out,imm_data,Data_in,Wt_addr,MemtoReg,sign,RegWrite)
400     always @(posedge clk or posedge rst) begin
401         if(rst) begin
402             MEM_WB_PC <= 32'b0;//just for debug
403             MEM_WB_Wt_addr <= 5'b0;
404             MEM_WB_RegWrite <= 1'b0;//
405             MEM_WB_Data_in <= 32'b0;
406             MEM_WB_MemtoReg <= 3'b0;//debug
407         end else begin
408             Wt_data <= Wt_data_temp;
409             MEM_WB_PC <= EX_MEM_PC;//just for debug
410             MEM_WB_Wt_addr <= EX_MEM_Wt_addr;
411             MEM_WB_RegWrite <= EX_MEM_RegWrite;
412             MEM_WB_Data_in <= Mem_data_in;
413             MEM_WB_MemtoReg <= EX_MEM_MemtoReg;//debug
414         end
415     end
416 
417     // Writeback
418 
419     //end of pipeline
420 
421     endmodule
```

二、实验结果与分析

首先，如果代码一切执行正常, 最后 x31 的值会变为 0x666，如下

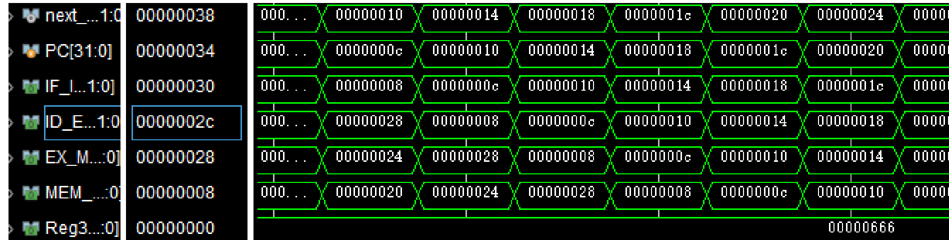

图 3: 通过仿真

接下来重点分析 Forwarding 和分支处理的仿真波形

Forwarding

use-use hazard

以这部分代码为例

```
li     x31, 1
li     x1, -1             # x1=FFFFFFFF
xori   x3, x1, 1          # x3=FFFFFFFE
# use-use hazard
add    x3, x3, x3         # x3=FFFFFFFC
add    x3, x3, x3         # x3=FFFFFFF8
add    x3, x3, x3         # x3=FFFFFFF0
# Mem stage use-use hazard
xori   x4, x1, 1          # x4=FFFFFFFE
add    x5, x4, x3         # x5=FFFFFFEE
addi   x5, x5, 2          # x5=FFFFFFF0
beq    x5, x3, pass_1
auipc  x30, 0
j      dummy
```

EX-MEM Forwarding

首先进行 xori 操作, 紧接着 add 操作需要使用 xori 操作的结果, 因此在 add 的 EX 阶段，同时存在 rs1 和 rs2 的 forwarding，都来自 MEM 阶段

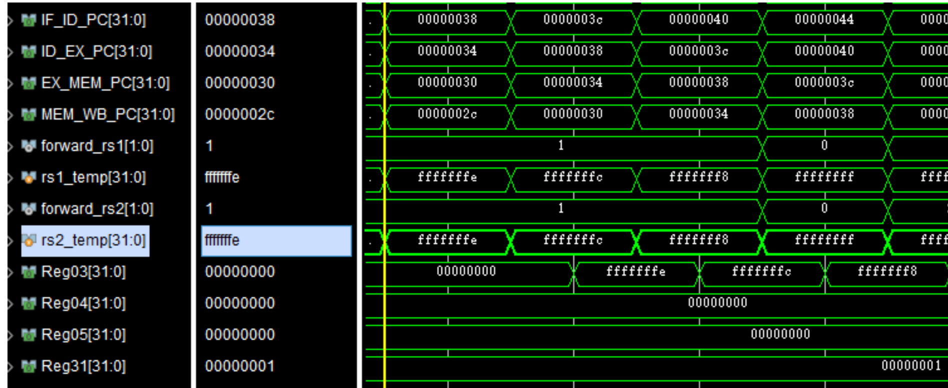

图 4: EX-MEM Forwarding

在 xori 之后,x3 的值为 FFFFFFFE, 可以看到，在这个值被写回寄存器之前,add 操作在 EX 阶段的 forwarding 已经生效，值为 1 代表来自 MEM 阶段，此时 rs1temp 和 rs2temp 的值都为 FFFFFFFE，进行了正确的运算，之后的连续两条 add 操作的 forwarding 也是类似的；最后确实按照预期改变了 x3 的值；

MEM-WB Forwarding

在接下来的 add x5 x4 x3 操作中,x3 需要 WB 阶段的前递，所以此时的 rs2 的 forwarding 信号为 2，代表来自 WB 阶段，此时 rs2temp 的值为 FFFFFFF0，进行正确的运算，最后确实按照预期改变了 x5 的值；

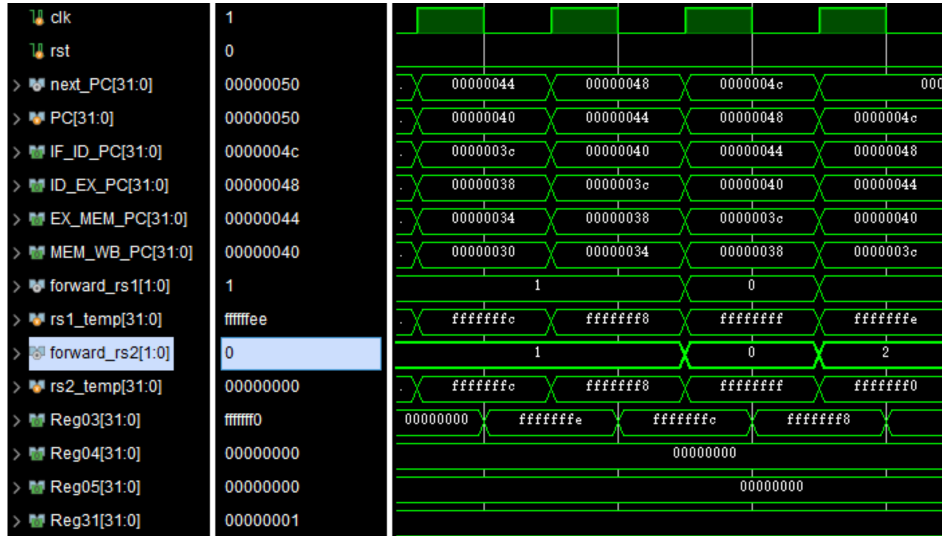

图 5: MEM-WB Forwarding

load-use hazard

这部分代码测试了 load-use hazard

```
li    x31, 2
li    x20, 0x20
sw    x5, 0(x20)
lh    x6, 0(x20)          # x6=FFFFFFF0
add   x6, x6, x6          # x6=FFFFFFE0
addi  x6, x6, 0x10        # x6=FFFFFFF0
beq   x6, x5, pass_2
auipc x30, 0
j     dummy
```

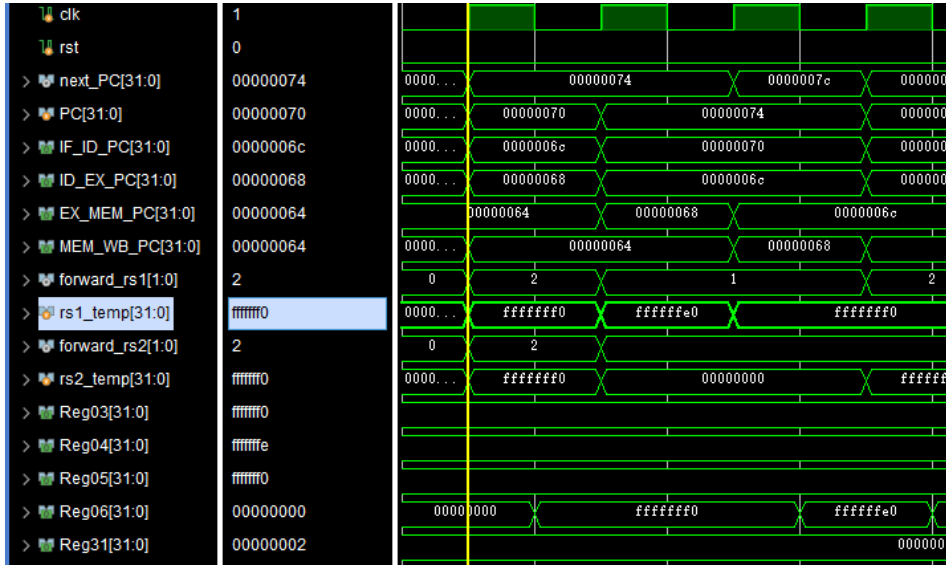

图 6: load-use hazard

如图，在 add x6 x6 x6 操作的 EX 阶段，由于前面监测到 load-use hazard，因此插入了一个 bubble, 所以此时发生的前递是在 lh 指令的 WB 阶段发生，此时 rs1 和 rs2 的 forwarding 信号为 2，代表来自 WB 阶段，此时 rs1temp 和 rs2temp 的值都为 FFFFFFF0，这就是即将要写回寄存器的值；说明我们的 forwarding 是正确的；

use-store hazard

这部分代码测试了 use-store hazard

```
1    add  x6, x6, x6          # x6=FFFFFFE0
2    sw   x6, 0(x20)
3    lw   x7, 0(x20)          # x7=FFFFFFE0
4    bne  x7, x5, pass_3
```

在 add 指令未执行完时,sw 指令需要将计算过的 x6 写入内存

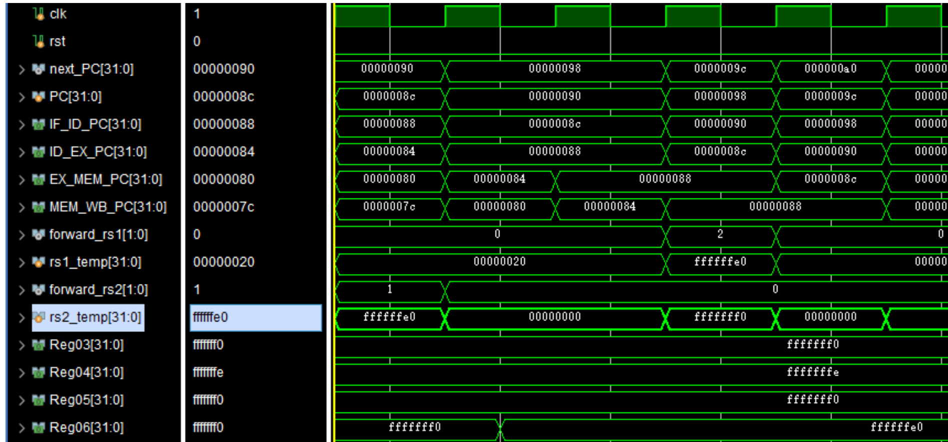

图 7: use-store hazard

可以看到，此时 rs1 不存在 forwarding，rs2 的 forwarding 信号为 1，代表来自 MEM 阶段，此时 rs2temp 的值为 FFFFFFE0，这正是即将要写入 x6 的值，说明我们的 forwarding 是正确的；最后我们将写入的值 load 回来，可以看到，x7 的值为 FFFFFFE0，与 x6 的值相同，说明我们成功写入；

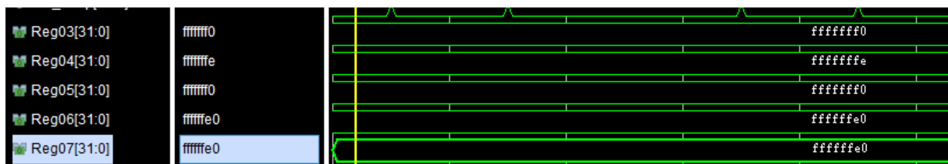

图 8: use-store hazard

接下来我们分析分支处理的阶段

分支处理

分支正常跳转

```
1    add    x5, x4, x3        # x5=FFFFFFEE
2    addi   x5, x5, 2         # x5=FFFFFFF0
3    beq    x5, x3, pass_1
```

以这部分代码为例, 在 beq 指令的 EX 阶段，需要用到 x5 和 x3 的值，但是由于会插入 bubble, 因此在 ID 阶段, addi 指令已经处于 MEM 阶段,add 指令已经处于 WB 阶段, 通过 double pump 可以得到正确的结果；所以只需要对 MEM 阶段进行 forwarding。

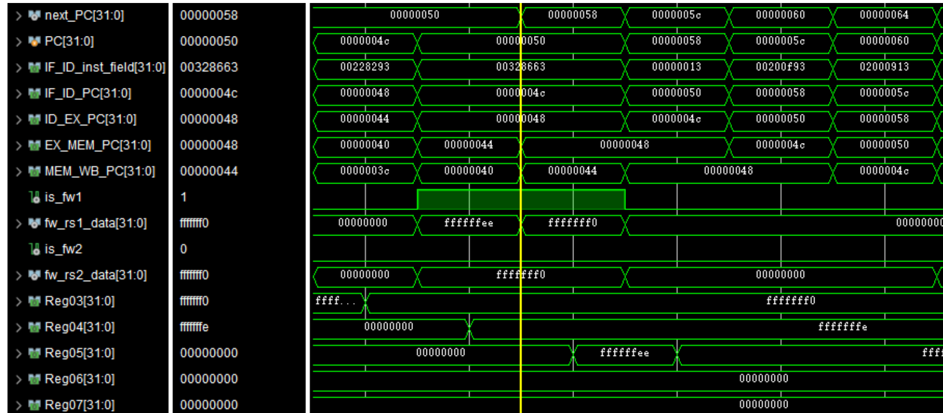

图 9: 分支正常跳转

观察波形图可以看到 is_fw1 的前递信号持续了两个周期，第一个周期是未产生 bubble 时，前递了处于 MEM 阶段的 add 指令的 x5 值，为 FFFFFFEE，第二个周期是插入 bubble 后，前递了处于 EX 阶段的 addi 指令的 x5 值，为 FFFFFFF0，也可以看到这一部分的 PC 和 IF_ID_inst_field 都不变，这正是插入 bubble 的体现；此时 (fwdata=FFFFFFF0 的时候), 判断发生了跳转, 所以 nextPC 为跳转地址 0x58，在下一个时钟周期到来时，PC 赋值为 nextPC，即 0x58，同时跟在 beq 指令后面的指令仍然正常进入 ID, 但是指令内容已经变为 nop(00000013)

分支不跳转

```
1   loop:
2   addi x1, x1, -1
3   bne  x1, x0, loop
4   beq  x1, x0, pass_4
```

在这个循环的最后一次，发生了分支不跳转

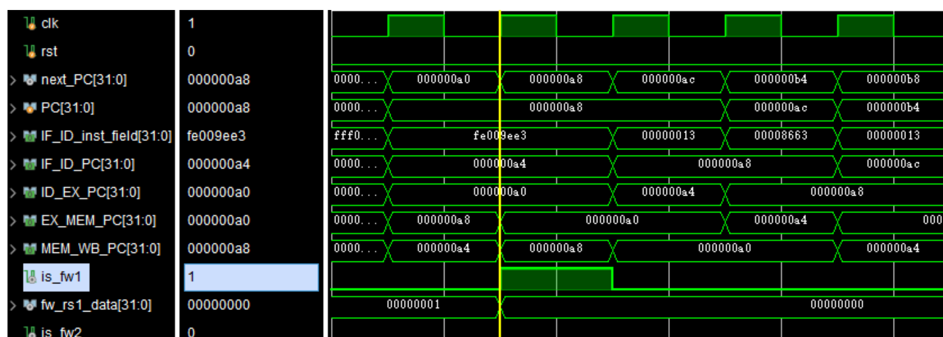

图 10: 分支不跳转

我们从 is_fw1 的信号升高开始看，此时 x1 得到了正确的值为 0,x1 和 x0 的值相等，所以不跳转，判断 nextPC 为 0xa8, 即跟在 bne 后面的指令；时钟上升沿到来时，原本的 0xa8 处的指令进入 ID 阶段指令变为 nop(00000013), 同时由于分支不执行,PC 变为 0x58 将它再次引进来, 再下一个时钟上升沿到来时，它进入 ID 阶段，指令正常传输进来执行；所以会看到这个地址再流水线中出现两次 (不考虑前面由于数据冲突 bubble 的那次)，这就是预测失败的代价;

最后我还测试了 branch 指令其它 forwarding 的情况，现象也均符合预期；此处不再赘述；

上板现象

仿真测试通过后，使用 Lab4-3 的代码进行上板测试，成功跑进 dummy 循环，最后 x31 的值为 0x666，符合预期；

```
RV32I Pipelined CPU

====================================== If ======================================
pc: 0000000C    inst: 00000013
====================================== Id ======================================
pc: 00000008    inst: 00000013  valid: 0
x0: 00000000    ra: FFFFFFFF    sp: 00000000    gp: 40000000    tp: 40000000
t0: F8000000    t1: C0000000    t2: 80000000    s0: 00000001    s1: 00000001
a0: 00000000    a1: C0000000    a2: 00000001    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000020    s3: 00000000
s4: 000002A4    s5: 000002A4    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:000000D0    t3: 000000D0    t4: 00D0CBA0
t5: 000002B0    t6: 00000666
====================================== Ex ======================================
pc: 00000028    inst: 00000000  valid: 0
rd:  00 rs1: 00 rs2: 00 rs1_val: 00000000       rs2_val: 00000000       reg_wen: 1
is_imm: 0       imm: 00000000
mem_wen: 0      mem_ren: 1      is_branch: 0    is_jal: 0       is_jalr: 0
is_auipc: 0     is_lui: 0       alu_ctrl: 0     cmp_ctrl: 0
====================================== Ma ======================================
pc: 00000024    inst: 00000000  valid: 0
rd:  00 reg_wen: 1      mem_w_data: 00000000    alu_res: F8000666
mem_wen: 0      mem_ren: 1      is_jal: 1       is_jalr: 0
====================================== Wb ======================================
pc: 00000020    inst: 00000000  valid: 0
rd:  00 reg_wen: 1      reg_w_data: 00000000
```

图 11: 上板现象

至此，我的流水线 CPU 实验结束。

三、讨论与实验心得

作为除去 Bonus 之外的最后一个实验，流水线的挑战确实很大, 完成了 5-1 部分后，我首先去重新看了一遍教材，对冲突的处理了然于胸后再开始修改代码, 在这一过程中也由于一开始不够细心，导致前递的逻辑没有写好,bubble 时的操作也出现了问题，用了比较长的时间去 debug; 但是磕磕绊绊，还是完成了实验，通过了最后的验收; 计组的实验也到来了尾声，通过这一学期与硬件的亲密接触，如今它已经不是上学期那个仿佛黑盒子般神秘的东西，对于硬件的调试也变得不是这么痛苦.

总而言之, 收获颇丰, 希望未来也能在硬件实验中取得更多的进步.

思考题

分析不同指令是否存在数据冲突

第一部分

```
1   addi    x1, x0, 0
2   addi    x2, x0, -1
3   addi    x3, x0, 1
4   addi    x4, x0, -1
5   addi    x5, x0, 1
6   addi    x6, x0, -1
7   addi    x1, x1, 0
8   addi    x2, x2, 1
9   addi    x3, x3, -1
10  addi    x4, x4, 1
11  addi    x5, x5, -1
12  addi    x6, x6, 1
```

这一部分代码在我的流水线 CPU 中没有出现数据冲突；对这部分代码进行仿真，增加计算时钟周期数的 count；

当 x6 变为 0 时,count 的值为 0xf, 即 16 个时钟周期；

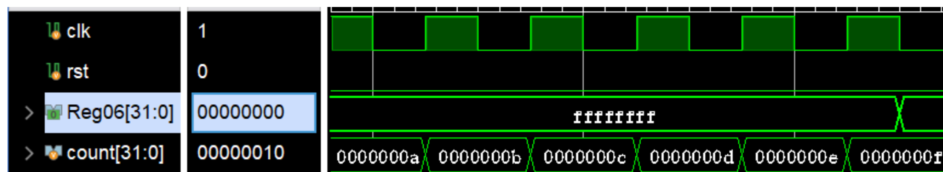

图 12: 第一部分

$$CPI = \frac{16}{12} = 1.33$$

第二部分

```
addi    x1, x0, 1  # x1=1
addi    x2, x1, 2  # x2=3
addi    x3, x1, 3  # x3=4
addi    x4, x3, 4  # x4=8
```

在这一部分中存在的冲突有：

- 第二条指令与第一条指令在 x1 上存在数据冲突
- 第三条指令与第一条指令在 x1 上存在数据冲突
- 第四条指令与第三条指令在 x3 上存在数据冲突

当 x4 变为 8 时,count 的值为 0x7, 即 8 个时钟周期；

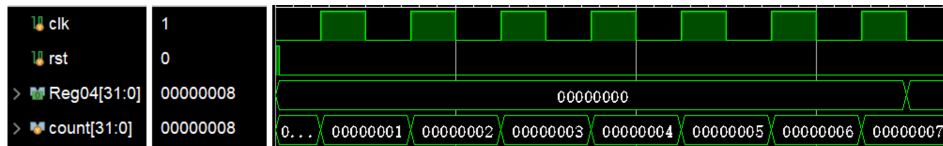

图 13: 第二部分

$$CPI = \frac{8}{4} = 2$$

对代码进行仿真

```
addi    x1, x0, 1  # x1=1
addi    x2, x1, 2  # x2=3
addi    x3, x2, 3  # x3=6
sw      x3, 0(x0)  # 0(x0)=6
lw      x4, 0(x0)  # x4=6
addi    x5, x4, 4  # x5=10
addi    x6, x4, 5  # x6=11
```

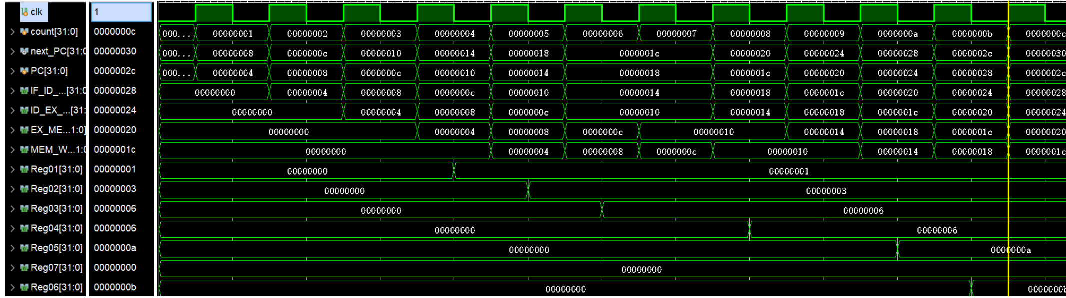

图 14: 第三部分

当 x6 变为 11 时,count 的值为 0xb, 即 12 个时钟周期；

$$CPI = \frac{12}{7} = 1.71$$