《计算机组成与设计》实验报告

姓名：龙永奇
学院：计算机科学与技术学院
专业：计算机科学与技术
邮箱：3220105907@zju.edu.cn

报告日期：2024年6月14日

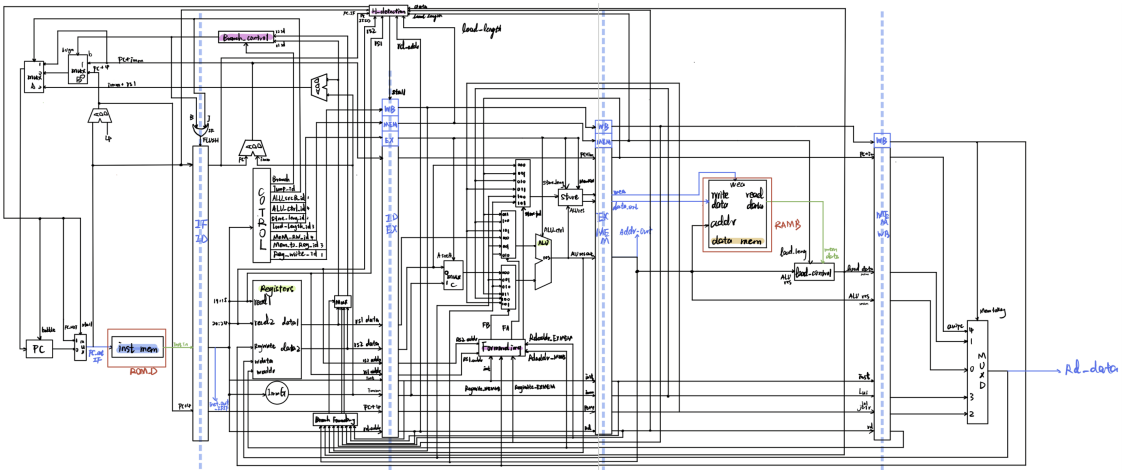
Lab5 < Pipeline CPU>

> <3220105907> <龙永奇> 3220105907@zju.edu.cn

1 操作方法与实验步骤

1.1 流水线 CPU 数据通路设计

- 本实验从实验 5-2 开始，直接实现冲突控制，数据通路整体设计如下：

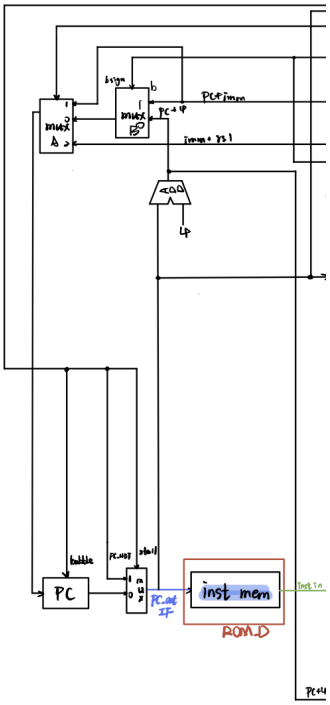


1.2 基本流水线模块实现

1.2.1 Inst_mem Data_mem（此模块不在数据通路结构中）

- 使用 Vivado 提供的 IP 核，参数与 Lab 4-3 一致

1.2.2 Instruction Fetch (IF)



IF 模块由 PC 寄存器以及跳转信号输入组成：

- PC 寄存器

  输入为 MuxA 选择后的数据，为了产生 bubble，额外添加了 bubble 控制信号

  输出结果由 stall 指令控制，用于 Load-use 冒险情况下的停顿

  实现代码如下：

```
module PC(
    input               clk,
    input               rst,
```

```
    input [31:0]        D,
    input               bubble,

    output reg [31:0]   Q
    );

    always @(posedge clk or posedge rst) begin
        if(rst)
            Q <= 32'b0;
        else if(bubble)
            Q <= Q;
        else
            Q <= D;
    end
endmodule
```

- Mux A

  用于控制 J 型指令跳转 PC，输入信号为 MuxB 输出，PC + imm，Rs1 + imm，控制信号由 Controller 产生

- Mux B

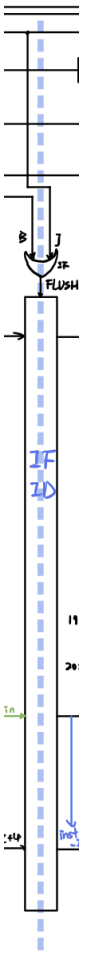
  用于控制 B 型指令跳转 PC，输入信号为 PC + imm，PC + 4，控制信号由 Branch_control 模块产生

整体整体实现代码如下：

```
module IF(
    input               clk_IF,
    input               rst_IF,
    input               Branch_signal_in_IF,
    input [1:0]         Jump_in_IF,
    input [31:0]        PC_plus_imm_in_IF,
    input [31:0]        imm_plus_rs1_in_IF,
    input               Stall_HDT_in_IF,
    input [31:0]        PC_HDT_in_IF,
    input               bubble_in_IF,

    output [31:0]       PC_out_IF,
    output [31:0]       PC_plus_4_out_IF
    );

    wire [31:0] MuxA_out;
    wire [31:0] MuxB_out;
    wire [31:0] Q_out;
    reg [31:0]  PC_out;

    PC IF1(
        .clk(clk_IF),
        .rst(rst_IF),
        .D(MuxA_out),
        .bubble(bubble_in_IF),

        .Q(Q_out)
    );

    always @(*) begin
```

```
        if(Stall_HDT_in_IF == 0)
            PC_out = Q_out;
        else if(Stall_HDT_in_IF == 1)
            PC_out = PC_HDT_in_IF;
    end

    assign PC_out_IF = PC_out;
    assign PC_plus_4_out_IF = PC_out_IF + 32'd4;

    Mux3to1 IF2(
        .data1(MuxB_out),
        .data2(PC_plus_imm_in_IF),
        .data3(imm_plus_rs1_in_IF),
        .control(Jump_in_IF),

        .out(MuxA_out)
    );

    Mux2to1 IF3(
        .data1(PC_plus_4_out_IF),
        .data2(PC_plus_imm_in_IF),
        .control(Branch_signal_in_IF),

        .out(MuxB_out)
    );

endmodule
```

1.2.3 IF_reg_ID



用于储存 IF 模块输出结果

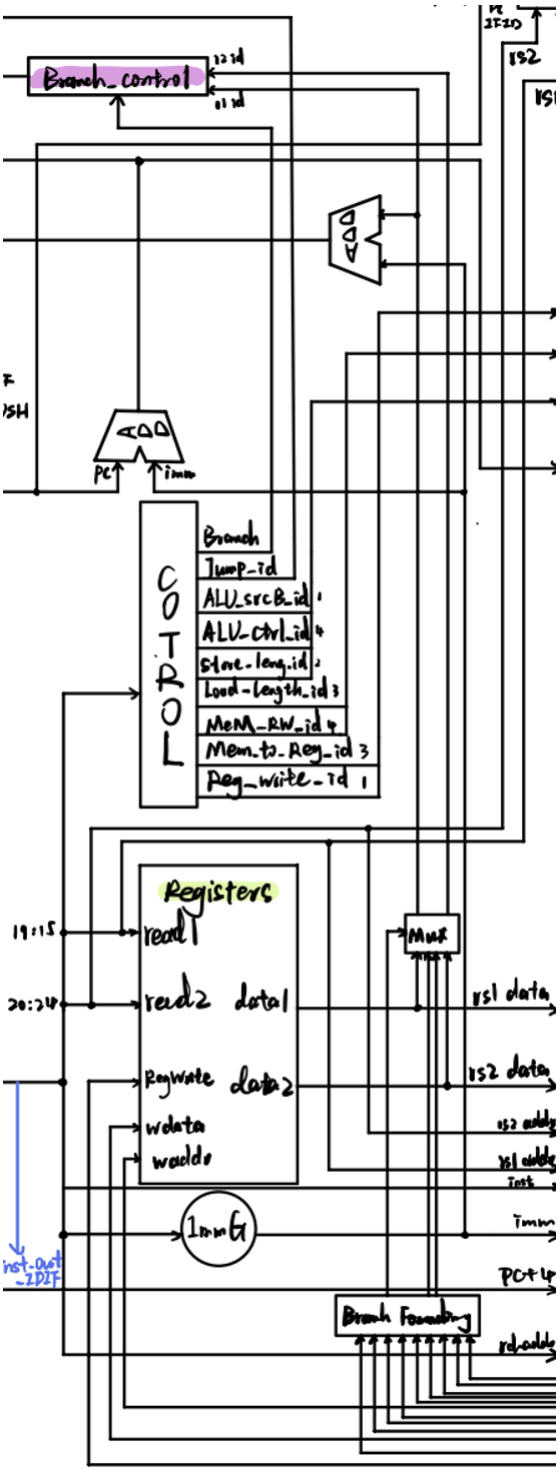
输入信号：

- 跳转指令 B 或 J 类型产生的 Flush 信号，只要发生跳转，就将读入的错误指令改为 nop
- PC，传递给下级流水线
- PC + 4，用于 Jal 或 Jalr 指令

整体实现代码如下：

```
module IF_reg_ID(
    input            clk_IFID,
    input            rst_IFID,
    input [31:0]     PC_in_IFID,
    input [31:0]     inst_in_IFID,
    input [31:0]     PC_plus_4_IFID,
    input            Flush_in_IFID,

    output reg [31:0]    PC_out_IFID,
    output reg [31:0]    inst_out_IFID,
    output reg [31:0]    PC_plus_4_out_IFID
    );

    always @(posedge clk_IFID or posedge rst_IFID) begin
        if (rst_IFID) begin
            PC_out_IFID <= 32'b0;
            inst_out_IFID <= 32'b0;
            PC_plus_4_out_IFID <= 32'b0;
        end
        else if(Flush_in_IFID) begin
            PC_out_IFID <= 32'b0;
            inst_out_IFID <= 32'h00000013;
            PC_plus_4_out_IFID <= 32'b0;
        end
        else begin
            PC_out_IFID <= PC_in_IFID;
            inst_out_IFID <= inst_in_IFID;
            PC_plus_4_out_IFID <= PC_plus_4_IFID;
        end
    end
endmodule
```

1.2.4 Instruciton Decode (ID)



本模块用于产生当前指令所需的控制信号以及寄存器的读写，由以下部分组成：

- Controller

  和 Lab4-3 不同的是改变了 Branch 信号的控制方法，因为需要将跳转判断放在 ID 阶段来完成分支预测，无法使用 ALU 的零位判断，修改部分如下：

```
// B-type
if(OPcode == 5'b11000) begin
    Jump = 0;
    RegWrite = 0;
    ALUSrc_B = 0;
    Load_len = 3'b111;
    case(Fun3)
```

```
        3'b000: Branch = 3'b000;            // beq
        3'b001: Branch = 3'b001;            // bne
        3'b100: Branch = 3'b010;            // blt
        3'b101: Branch = 3'b011;            // bge
        3'b110: Branch = 3'b100;            // bltu
        3'b111: Branch = 3'b101;            // bgeu
    endcase
end
```

将不同类型的 Branch 信号转译为 000 至 101 的信号，传给 Branch_control，不再产生 Branch 以及 BranchN 信号

- Branch_control

  实现代码如下：

```
module Branch_control(
    input [2:0]     Branch,
    input [31:0]    Rs1_data,
    input [31:0]    Rs2_data,

    output          Branch_signal
    );
    reg Branch_signal_tmp;
    always @(*) begin
        case(Branch)
            3'b000: begin
                if(Rs1_data == Rs2_data)
                    Branch_signal_tmp <= 1;
                else
                    Branch_signal_tmp <= 0;
            end
            3'b001: begin
                if(Rs1_data != Rs2_data)
                    Branch_signal_tmp <= 1;
                else
                    Branch_signal_tmp <= 0;
            end
            3'b010: begin
                if($signed(Rs1_data) < $signed(Rs2_data))
                    Branch_signal_tmp <= 1;
                else
                    Branch_signal_tmp <= 0;
            end
            3'b011: begin
                if($signed(Rs1_data) >= $signed(Rs2_data))
                    Branch_signal_tmp <= 1;
                else
                    Branch_signal_tmp <= 0;
            end
            3'b100: begin
                if(Rs1_data < Rs2_data)
                    Branch_signal_tmp <= 1;
                else
                    Branch_signal_tmp <= 0;
            end
```

```
            3'b101: begin
                if(Rs1_data >= Rs2_data)
                    Branch_signal_tmp <= 1;
                else
                    Branch_signal_tmp <= 0;
            end
            default: Branch_signal_tmp <= 0;
        endcase
    end
    assign Branch_signal = Branch_signal_tmp;

endmodule
```

根据不同类型 Branch 指令，将传入的值进行比较，判断是否跳转

ID 模块需要将 Branch 信号以及 Jump 信号传回至 IF 模块，用于 PC 的选择

- 在实现中将传给后续模块的控制信号整合为 `WB`，`MEM`，`EX`，对应于三个不同阶段

```
assign WB_out_ID = {MemtoReg_ID, RegWrite_out_ID}; //高->低
assign MEM_out_ID = {Load_len_ID};
assign EX_out_ID = {ALUSrc_B_ID, ALU_Control_ID, Store_len_ID,
MemRW_ID};
```

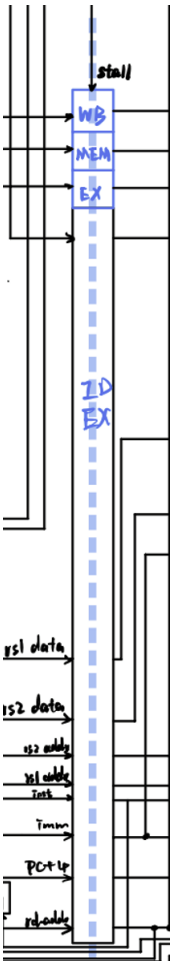
整体实现代码如下：

```
`timescale 1ps/1ps

module ID(
    input               clk_ID,
    input               rst_ID,
    input [31:0]        PC_in_ID,
    input [31:0]        inst_in_ID,
    input [31:0]        PC_plus_4_in_ID,
    input               RegWrite_in_ID,
    input [4:0]         Rd_addr_in_ID,
    input [31:0]        Rd_data_in_ID,
    input               BFWD_A_in_ID,
    input               BFWD_B_in_ID,
    input [31:0]        BFWD_A_data_in_ID,
    input [31:0]        BFWD_B_data_in_ID,

    output              Branch_signal_out_ID,
    output [1:0]        Jump_out_ID,
    output [31:0]       imm_plus_rs1_out_ID,
    output [3:0]        WB_out_ID,
    output [2:0]        MEM_out_ID,
    output [10:0]       EX_out_ID,
    output [31:0]       PC_plus_imm_out_ID,
    output [31:0]       Rs1_data_out_ID,
    output [31:0]       Rs2_data_out_ID,
    output [31:0]       imm_out_ID,
    output [31:0]       PC_plus_4_out_ID,
    output [4:0]        Rd_addr_out_ID,
    output [31:0]       inst_out_ID,
    output              Flush_out_ID,
```

```
    output [31:0]       Reg00,      output [31:0]   Reg01,
    output [31:0]       Reg02,      output [31:0]   Reg03,
    output [31:0]       Reg04,      output [31:0]   Reg05,
    output [31:0]       Reg06,      output [31:0]   Reg07,
    output [31:0]       Reg08,      output [31:0]   Reg09,
    output [31:0]       Reg10,      output [31:0]   Reg11,
    output [31:0]       Reg12,      output [31:0]   Reg13,
    output [31:0]       Reg14,      output [31:0]   Reg15,
    output [31:0]       Reg16,      output [31:0]   Reg17,
    output [31:0]       Reg18,      output [31:0]   Reg19,
    output [31:0]       Reg20,      output [31:0]   Reg21,
    output [31:0]       Reg22,      output [31:0]   Reg23,
    output [31:0]       Reg24,      output [31:0]   Reg25,
    output [31:0]       Reg26,      output [31:0]   Reg27,
    output [31:0]       Reg28,      output [31:0]   Reg29,
    output [31:0]       Reg30,      output [31:0]   Reg31
    );

    wire [2:0]  Branch_ID;
    wire        RegWrite_out_ID;
    wire        ALUSrc_B_ID;
    wire [1:0]  Store_len_ID;
    wire [3:0]  ALU_Control_ID;
    wire [2:0]  MemtoReg_ID;
    wire [3:0]  MemRW_ID;
    wire [2:0]  Load_len_ID;

    assign imm_plus_rs1_out_ID = imm_out_ID + Rs1_data_out_ID;
    assign WB_out_ID = {MemtoReg_ID, RegWrite_out_ID}; //高->低
    assign MEM_out_ID = {Load_len_ID};
    assign EX_out_ID = {ALUSrc_B_ID, ALU_Control_ID, Store_len_ID,
MemRW_ID};
    assign PC_plus_imm_out_ID = PC_in_ID + imm_out_ID;
    assign PC_plus_4_out_ID = PC_plus_4_in_ID;
    assign Rd_addr_out_ID = inst_in_ID[11:7];
    assign inst_out_ID = inst_in_ID;
    assign Flush_out_ID = Branch_signal_out_ID | Jump_out_ID;

    reg [31:0] Rs1_data_tmp;
    reg [31:0] Rs2_data_tmp;
    always @(*)begin
        if(BFWD_A_in_ID)
            Rs1_data_tmp = BFWD_A_data_in_ID;
        else
            Rs1_data_tmp = Rs1_data_out_ID;
        if(BFWD_B_in_ID)
            Rs2_data_tmp = BFWD_B_data_in_ID;
        else
            Rs2_data_tmp = Rs2_data_out_ID;
    end

    Controller ID1(
        .OPcode(inst_in_ID[6:2]),
        .Fun3(inst_in_ID[14:12]),
        .Fun7(inst_in_ID[30]),
        .Jump(Jump_out_ID),
```

```
        .Branch(Branch_ID),
        .RegWrite(RegWrite_out_ID),
        .ALUSrc_B(ALUSrc_B_ID),
        .Store_len(Store_len_ID),
        .ALU_Control(ALU_Control_ID),
        .MemtoReg(MemtoReg_ID),
        .MemRW(MemRW_ID),
        .Load_len(Load_len_ID)
    );

    Branch_control ID2(
        .Branch(Branch_ID),
        .Rs1_data(Rs1_data_tmp),
        .Rs2_data(Rs2_data_tmp),
        .Branch_signal(Branch_signal_out_ID)
    );

    Regs ID3(
        .clk(clk_ID),
        .rst(rst_ID),
        .Rs1_addr(inst_in_ID[19:15]),
        .Rs2_addr(inst_in_ID[24:20]),
        .Wt_addr(Rd_addr_in_ID),
        .Wt_data(Rd_data_in_ID),
        .RegWrite(RegWrite_in_ID),
        .Rs1_data(Rs1_data_out_ID),
        .Rs2_data(Rs2_data_out_ID),
        .Reg00(Reg00),.Reg01(Reg01),.Reg02(Reg02),.Reg03(Reg03),
.Reg04(Reg04),.Reg05(Reg05),.Reg06(Reg06),.Reg07(Reg07),
.Reg08(Reg08),.Reg09(Reg09),.Reg10(Reg10),.Reg11(Reg11),
.Reg12(Reg12),.Reg13(Reg13),.Reg14(Reg14),.Reg15(Reg15),
.Reg16(Reg16),.Reg17(Reg17),.Reg18(Reg18),.Reg19(Reg19),
.Reg20(Reg20),.Reg21(Reg21),.Reg22(Reg22),.Reg23(Reg23),
.Reg24(Reg24),.Reg25(Reg25),.Reg26(Reg26),.Reg27(Reg27),
.Reg28(Reg28),.Reg29(Reg29),.Reg30(Reg30),.Reg31(Reg31)
    );

    ImmG ID4(
        .inst_imm(inst_in_ID),
        .Imm_out(imm_out_ID)
    );
endmodule
```

1.2.5 ID_reg_EX



用于储存 IF 模块输出结果，输入信号如下：

- WB、MEM、EX

  分别用于对应的三个阶段的控制信号

- PC + imm

  auipc 指令结果

- Rs1 Rs2 data

  用于 EX 阶段计算

- inst

  ID 阶段当前指令，用于后续解决冲突

- imm

  用于 EX 阶段计算或 lui 指令结果

- PC + 4

  jal 或 jalr 结果

- Rd_data

  用于写回寄存器以及后续解决冲突

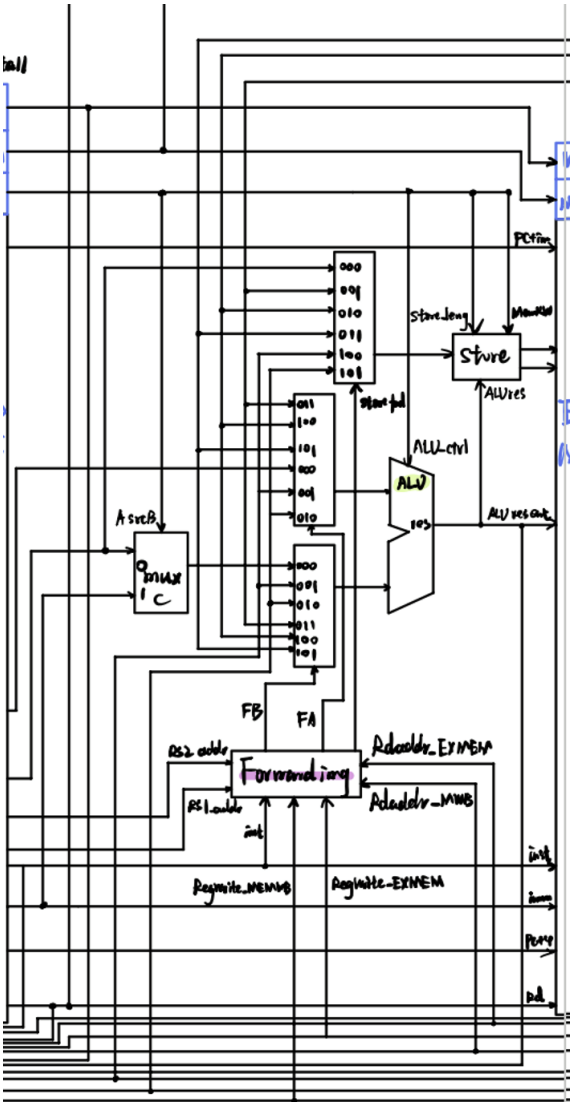
整体实现代码如下：

```
module ID_reg_EX(
    input                   clk_IDEX,
    input                   rst_IDEX,
```

```
    input [3:0]             WB_in_IDEX,
    input [2:0]             MEM_in_IDEX,
    input [10:0]            EX_in_IDEX,
    input [31:0]            PC_plus_imm_in_IDEX,
    input [31:0]            Rs1_data_in_IDEX,
    input [31:0]            Rs2_data_in_IDEX,
    input [31:0]            imm_in_IDEX,
    input [31:0]            PC_plus_4_in_IDEX,
    input [4:0]             Rd_addr_in_IDEX,
    input                   Stall_HDT_in_IDEX,
    input [4:0]             Rs1_addr_in_IDEX,
    input [4:0]             Rs2_addr_in_IDEX,
    input [31:0]            inst_in_IDEX,

    output reg [3:0]        WB_out_IDEX,
    output reg [2:0]        MEM_out_IDEX,
    output reg [10:0]       EX_out_IDEX,
    output reg [31:0]       PC_plus_imm_out_IDEX,
    output reg [31:0]       Rs1_data_out_IDEX,
    output reg [31:0]       Rs2_data_out_IDEX,
    output reg [31:0]       imm_out_IDEX,
    output reg [31:0]       PC_plus_4_out_IDEX,
    output reg [4:0]        Rd_addr_out_IDEX,
    output reg [4:0]        Rs1_addr_out_IDEX,
    output reg [4:0]        Rs2_addr_out_IDEX,
    output reg [31:0]       inst_out_IDEX
    );
    always @(posedge clk_IDEX or posedge rst_IDEX) begin
        if(rst_IDEX | Stall_HDT_in_IDEX) begin
            WB_out_IDEX <= 4'b0;
            MEM_out_IDEX <= 3'b0;
            EX_out_IDEX <= 11'b0;
            PC_plus_imm_out_IDEX <= 32'b0;
            Rs1_data_out_IDEX <= 32'b0;
            Rs2_data_out_IDEX <= 32'b0;
            imm_out_IDEX <= 32'b0;
            PC_plus_4_out_IDEX <= 32'b0;
            Rd_addr_out_IDEX <= 5'b0;
            Rs1_addr_out_IDEX <= 5'b0;
            Rs2_addr_out_IDEX <= 5'b0;
            inst_out_IDEX <= 0;
        end
        else begin
            WB_out_IDEX <= WB_in_IDEX;
            MEM_out_IDEX <= MEM_in_IDEX;
            EX_out_IDEX <= EX_in_IDEX;
            PC_plus_imm_out_IDEX <= PC_plus_imm_in_IDEX;
            Rs1_data_out_IDEX <= Rs1_data_in_IDEX;
            Rs2_data_out_IDEX <= Rs2_data_in_IDEX;
            imm_out_IDEX <= imm_in_IDEX;
            PC_plus_4_out_IDEX <= PC_plus_4_in_IDEX;
            Rd_addr_out_IDEX <= Rd_addr_in_IDEX;
            Rs1_addr_out_IDEX <= Rs1_addr_in_IDEX;
            Rs2_addr_out_IDEX <= Rs2_addr_in_IDEX;
            inst_out_IDEX <= inst_in_IDEX;
        end
```

```
    end
endmodule
```

1.2.6 Execution (EX)



本模块用于 ALU 计算以及生成 Store 指令所需的控制信号及数据，由以下部分组成：

- ALU

  同 Lab 4-3 相比，删去了零位判断，将 Branch 跳转判断提前至 ID 阶段

- Store

  生成 Store 指令所需的控制信号及数据，实现代码如下：

```
module Store_control(
    input [31:0]        Regs_Rs2_data,      // 要存储的数据
    input [1:0]         Store_control,      // 00 sb, 01 sh, 10 sw
    input [31:0]        ALU_res,            // 地址
    input [3:0]         MemRW,              // MemRW
    output reg [31:0]   store_control_out,  // 整合后的存储数据
    output reg [3:0]    store_control_wea   // 控制 wea
    );
    always @(*) begin
        case(Store_control)
            2'b00: begin        // sb
                if(ALU_res[1:0] == 0) begin
```

```
                    store_control_out = {24'b0,Regs_Rs2_data[7:0]};
                    store_control_wea = 4'b0001;
                end
                else if(ALU_res[1:0] == 2'b01) begin
                    store_control_out = {16'b0, Regs_Rs2_data[7:0],
8'b0};
                    store_control_wea = 4'b0010;
                end
                else if(ALU_res[1:0] == 2'b10) begin
                    store_control_out = {8'b0, Regs_Rs2_data[7:0],
16'b0};
                    store_control_wea = 4'b0100;
                end
                else if(ALU_res[1:0] == 2'b11) begin
                    store_control_out = {Regs_Rs2_data[7:0],24'b0};
                    store_control_wea = 4'b1000;
                end
            end
            2'b01: begin        // sh
                if(ALU_res[1:0] == 0) begin
                    store_control_out = {16'b0,Regs_Rs2_data[15:0]};
                    store_control_wea = 4'b0011;
                end
                else if(ALU_res[1:0] == 2'b10) begin
                    store_control_out = {Regs_Rs2_data[15:0],16'b0};
                    store_control_wea = 4'b1100;
                end
            end
            2'b10: begin        // sw
                    store_control_out = {Regs_Rs2_data[31:0]};
                    store_control_wea = 4'b1111;
                end
        endcase
        if(MemRW != 4'b1111) begin
            store_control_wea = 4'b0000;
        end
    end
endmodule
```

与 Lab4-3 不同的是，在 Controller 发出的 `MemRW` 信号为 `1111` 时，本模块将 `store_control_wea` 直接作为最终传入至内存的 `wea` 信号，即根据 `byte`，`halfbyte`，`word` 控制不同字节的写入使能

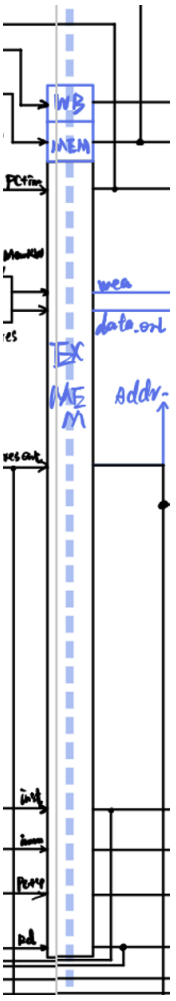
整体实现代码如下：

```
`timescale 1ns / 1ps

module EX(
    input [10:0]            EX_in_EX,
    input [31:0]            PC_plus_imm_in_EX,
    input [31:0]            Rs1_data_in_EX,
    input [31:0]            Rs2_data_in_EX,
    input [31:0]            imm_in_EX,
    input [31:0]            PC_plus_4_in_EX,
    input [4:0]             Rd_addr_in_EX,
```

```
    input [2:0]             Forwarding_A_in_EX,
    input [2:0]             Forwarding_B_in_EX,
    input [2:0]             store_fwd_in_EX,
    input [31:0]            Rd_data_in_EX,
    input [31:0]            P_ALU_res_in_EX,
    input [31:0]            imm_EXMEM_in_EX,
    input [31:0]            PC_plus_4_EXMEM_in_EX,
    input [31:0]            PC_plus_imm_EXMEM_in_EX,
    input [31:0]            inst_in_EX,

    output [31:0]           PC_plus_imm_out_EX,
    output [3:0]            store_control_wea_out_EX,
    output [31:0]           store_control_data_out_EX,
    output [31:0]           ALU_res_out_EX,
    output [31:0]           imm_out_EX,
    output [31:0]           PC_plus_4_out_EX,
    output [4:0]            Rd_addr_out_EX,
    output [31:0]           inst_out_EX
    );

    wire [31:0] MuxC_out;
    wire [31:0] A_out;
    wire [31:0] B_out;
    wire [4:0]  ALU_ctrl = EX_in_EX[9:6];
    wire [1:0]  St_ctrl = EX_in_EX[5:4];
    wire [3:0]  MemR = EX_in_EX[3:0];
    wire        ALUsrcB = EX_in_EX[10];
    wire [31:0] Regs_Rs2_data_in;

    assign PC_plus_imm_out_EX = PC_plus_imm_in_EX;
    assign imm_out_EX = imm_in_EX;
    assign PC_plus_4_out_EX = PC_plus_4_in_EX;
    assign Rd_addr_out_EX = Rd_addr_in_EX;
    assign inst_out_EX = inst_in_EX;
    assign Regs_Rs2_data_in =
        store_fwd_in_EX == 3'b000 ? Rs2_data_in_EX :
        store_fwd_in_EX == 3'b001 ? Rd_data_in_EX :
        store_fwd_in_EX == 3'b010 ? P_ALU_res_in_EX :
        store_fwd_in_EX == 3'b011 ? PC_plus_imm_EXMEM_in_EX :
        store_fwd_in_EX == 3'b100 ? imm_EXMEM_in_EX :
        store_fwd_in_EX == 3'b101 ? PC_plus_4_EXMEM_in_EX : 0;

    Store_control EX1(
        .Regs_Rs2_data(Regs_Rs2_data_in),
        .Store_control(St_ctrl),
        .ALU_res(ALU_res_out_EX),
        .MemRW(MemR),
        .store_control_out(store_control_data_out_EX),
        .store_control_wea(store_control_wea_out_EX)
    );

    Mux2to1 EX2(
        .data1(Rs2_data_in_EX),
        .data2(imm_in_EX),
        .control(ALUsrcB),
        .out(MuxC_out)
```

```
    );

    Mux6to1 MuxFA(
        .data1(Rs1_data_in_EX),
        .data2(Rd_data_in_EX),              // i小恩惠结果
        .data3(P_ALU_res_in_EX),            // 前一个alu结果
        .data4(PC_plus_imm_EXMEM_in_EX),    // auipc
        .data5(imm_EXMEM_in_EX),            // lui
        .data6(PC_plus_4_EXMEM_in_EX),      // jalr jal
        .control(Forwarding_A_in_EX),
        .out(A_out)
    );

    Mux6to1 MuxFB(
        .data1(MuxC_out),
        .data2(Rd_data_in_EX),
        .data3(P_ALU_res_in_EX),
        .data4(PC_plus_imm_EXMEM_in_EX),
        .data5(imm_EXMEM_in_EX),
        .data6(PC_plus_4_EXMEM_in_EX),
        .control(Forwarding_B_in_EX),
        .out(B_out)
    );

    ALU EX3(
        .A(A_out),
        .B(B_out),
        .ALU_Control(ALU_ctrl),
        .res(ALU_res_out_EX)
    );
endmodule
```

1.2.7 EX_reg_MEM



本模块用于储存 EX 阶段的计算结果以及 MEM 与 WB 阶段的控制信号

输入信号包括：

- WB MEM
- PC + imm、inst、imm、PC + 4、Rd addr
- ALU_res

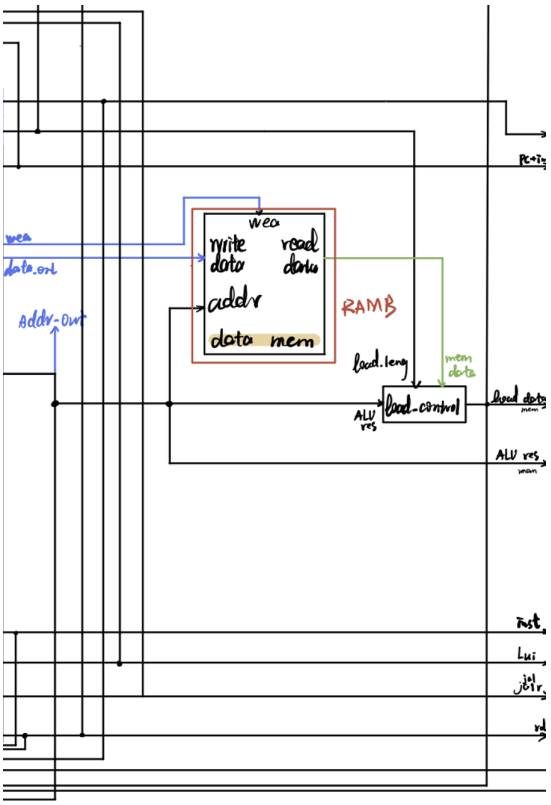
  此处 ALU 计算结果包括 Load 信号所需地址以及运算结果，前者用于 Load_control 模块生成不同类型 Load 指令所需不同数据

整体实现代码如下：

```
`timescale 1ns / 1ps

module EX_reg_MEM(
    input               clk_EXMEM,
    input               rst_EXMEM,
    input [3:0]         WB_in_EXMEM,
    input [2:0]         MEM_in_EXMEM,
    input [31:0]        PC_plus_imm_in_EXMEM,
    input [3:0]         store_control_wea_in_EXMEM,
    input [31:0]        store_control_data_in_EXMEM,
    input [31:0]        ALU_res_in_EXMEM,
    input [31:0]        imm_in_EXMEM,
    input [31:0]        PC_plus_4_in_EXMEM,
    input [4:0]         Rd_addr_in_EXMEM,
```

```
    input [31:0]        inst_in_EXMEM,

    output reg [3:0]    WB_out_EXMEM,
    output reg [2:0]    MEM_out_EXMEM,
    output reg [31:0]   PC_plus_imm_out_EXMEM,
    output reg [3:0]    store_control_wea_out_EXMEM,
    output reg [31:0]   store_control_data_out_EXMEM,
    output reg [31:0]   ALU_res_out_EXMEM,
    output reg [31:0]   imm_out_EXMEM,
    output reg [31:0]   PC_plus_4_out_EXMEM,
    output reg [4:0]    Rd_addr_out_EXMEM,
    output reg [31:0]   inst_out_EXMEM
    );

    always @(posedge clk_EXMEM or posedge rst_EXMEM) begin
        if(rst_EXMEM) begin
            WB_out_EXMEM <= 4'b0;
            MEM_out_EXMEM <= 3'b0;
            PC_plus_imm_out_EXMEM <= 32'b0;
            store_control_wea_out_EXMEM <= 4'b0;
            store_control_data_out_EXMEM <= 32'b0;
            ALU_res_out_EXMEM <= 32'b0;
            imm_out_EXMEM <= 32'b0;
            PC_plus_4_out_EXMEM <= 32'b0;
            Rd_addr_out_EXMEM <= 5'b0;
            inst_out_EXMEM <= 0;
        end
        else begin
            WB_out_EXMEM <= WB_in_EXMEM;
            MEM_out_EXMEM <= MEM_in_EXMEM;
            PC_plus_imm_out_EXMEM <= PC_plus_imm_in_EXMEM;
            store_control_wea_out_EXMEM <= store_control_wea_in_EXMEM;
            store_control_data_out_EXMEM <=store_control_data_in_EXMEM;
            ALU_res_out_EXMEM <= ALU_res_in_EXMEM;
            imm_out_EXMEM <= imm_in_EXMEM;
            PC_plus_4_out_EXMEM <= PC_plus_4_in_EXMEM;
            Rd_addr_out_EXMEM <= Rd_addr_in_EXMEM;
            inst_out_EXMEM <= inst_in_EXMEM;
        end
    end

endmodule
```

1.2.8 Memory (MEM)



本模块用于内存数据的读写，组成模块如下：

- Load_control

  该模块接收从内存中读取的数据，并根据 ALU 计算的地址，整合不同类型 Load 指令所需的数据，实现代码如下：

```
module Load_control(
    input [31:0]        data_mem_out,       // 要读的数据
    input [2:0]         Load_control,
    input [31:0]        ALU_res,            // 地址
    output reg [31:0]   Load_control_data    // 整合后的数据
    );

    always @(*) begin
        case(Load_control)
            3'b000: begin       // lb
                if(ALU_res[1:0] == 0) begin
                    if(data_mem_out[7] == 0)
                        Load_control_data = {24'b0,
                                             data_mem_out[7:0]};
                    else
                        Load_control_data = {24'hFF_FFFF,
                                             data_mem_out[7:0]};
                end
                else if(ALU_res[1:0] == 2'b01) begin
                    if(data_mem_out[15] == 0)
                        Load_control_data = {24'b0,
                                             data_mem_out[15:8]};
                    else
                        Load_control_data = {24'hFF_FFFF,
                                             data_mem_out[15:8]};
```

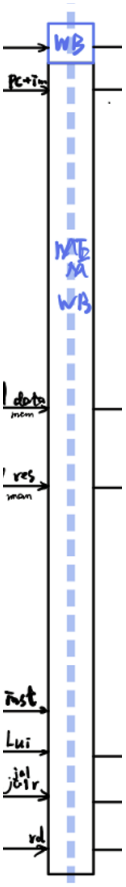
```
                end
                else if(ALU_res[1:0] == 2'b10) begin
                    if(data_mem_out[23] == 0)
                        Load_control_data = {24'b0,
                                    data_mem_out[23:16]};
                    else
                        Load_control_data = {24'hFF_FFFF,
                                    data_mem_out[23:16]};
                end
                else if(ALU_res[1:0] == 2'b11) begin
                    if(data_mem_out[31] == 0)
                        Load_control_data = {24'b0,
                                    data_mem_out[31:24]};
                    else
                        Load_control_data = {24'hFF_FFFF,
                                    data_mem_out[31:24]};
                end
            end
            3'b010: begin       // lh
                if(ALU_res[1:0] == 0) begin
                    if(data_mem_out[15] == 0)
                        Load_control_data = {16'b0,
                                     data_mem_out[15:0]};
                    else
                        Load_control_data = {16'hFFFF,
                                       data_mem_out[15:0]};
                end
                else if(ALU_res[1:0] == 2'b10) begin
                    if(data_mem_out[31] == 0)
                        Load_control_data = {16'b0,
                                    data_mem_out[31:16]};
                    else
                        Load_control_data = {16'hFFFF,
                                      data_mem_out[31:16]};
                end
            end
            3'b100: begin       // lw
                Load_control_data = {data_mem_out[31:0]};
            end
            3'b001: begin       // lbu
                if(ALU_res[1:0] == 0) begin
                    Load_control_data = {24'b0,data_mem_out[7:0]};
                end
                else if(ALU_res[1:0] == 2'b01) begin
                    Load_control_data = {24'b0,data_mem_out[15:8]};
                end
                else if(ALU_res[1:0] == 2'b10) begin
                    Load_control_data = {24'b0,data_mem_out[23:16]};
                end
                else if(ALU_res[1:0] == 2'b11) begin
                    Load_control_data = {24'b0,data_mem_out[31:24]};
                end
            end
            3'b011: begin       // lhu
                if(ALU_res[1:0] == 0) begin
                    Load_control_data = {16'b0,data_mem_out[15:0]};
```

```
                end
                else if(ALU_res[1:0] == 2'b10) begin
                    Load_control_data = {16'b0,data_mem_out[31:16]};
                end
            end
        endcase
    end
endmodule
```

整体实现代码如下：

```
module MEM(
    input [2:0]         MEM_in_MEM,
    input [31:0]        PC_plus_imm_in_MEM,
    input [31:0]        ALU_res_in_MEM,
    input [31:0]        imm_in_MEM,
    input [31:0]        PC_plus_4_in_MEM,
    input [4:0]         Rd_addr_in_MEM,
    input [31:0]        RAM_B_data_in_MEM,
    input [31:0]        inst_in_MEM,
    output [31:0]       PC_plus_imm_out_MEM,
    output [31:0]       Load_control_data_out_MEM,
    output [31:0]       ALU_res_out_MEM,
    output [31:0]       imm_out_MEM,
    output [31:0]       PC_plus_4_out_MEM,
    output [4:0]        Rd_addr_out_MEM,
    output [31:0]       inst_out_MEM
    );

    assign PC_plus_imm_out_MEM = PC_plus_imm_in_MEM;
    assign ALU_res_out_MEM = ALU_res_in_MEM;
    assign imm_out_MEM = imm_in_MEM;
    assign PC_plus_4_out_MEM = PC_plus_4_in_MEM;
    assign Rd_addr_out_MEM = Rd_addr_in_MEM;
    assign inst_out_MEM = inst_in_MEM;

    Load_control MEM1(
        .data_mem_out(RAM_B_data_in_MEM),
        .Load_control(MEM_in_MEM),
        .ALU_res(ALU_res_in_MEM),
        .Load_control_data(Load_control_data_out_MEM)
    );

endmodule
```

1.2.9 MEM_reg_WB



本模块用于储存 MEM 阶段读取的数据

包含以下信号：

- WB MEM
- PC + imm、inst、imm、PC + 4、Rd addr
- ALU_res
- Load_data

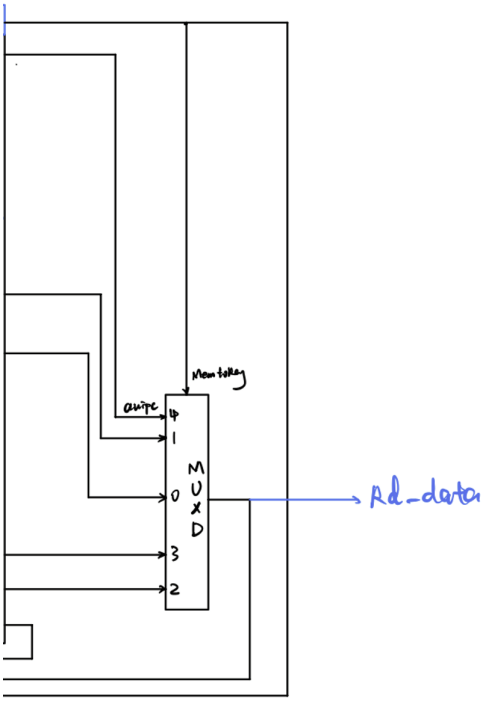
  即 Load_control 模块根据地址以及内存中读取的数据，产生的整合后的数据

整体实现如下：

```
module MEM_reg_WB(
    input               clk_MEMWB,
    input               rst_MEMWB,
    input [3:0]         WB_in_MEMWB,
    input [31:0]        PC_plus_imm_in_MEMWB,
    input [31:0]        Load_control_data_in_MEMWB,
    input [31:0]        ALU_res_in_MEMWB,
    input [31:0]        imm_in_MEMWB,
    input [31:0]        PC_plus_4_in_MEMWB,
    input [4:0]         Rd_addr_in_MEMWB,
    input [31:0]        inst_in_MEMWB,

    output reg [3:0]    WB_out_MEMWB,
    output reg [31:0]   PC_plus_imm_out_MEMWB,
    output reg [31:0]   Load_control_data_out_MEMWB,
    output reg [31:0]   ALU_res_out_MEMWB,
```

```
    output reg [31:0]   imm_out_MEMWB,
    output reg [31:0]   PC_plus_4_out_MEMWB,
    output reg [4:0]    Rd_addr_out_MEMWB,
    output reg [31:0]   inst_out_MEMWB
    );
    always @(posedge clk_MEMWB or posedge rst_MEMWB) begin
        if(rst_MEMWB) begin
            WB_out_MEMWB <= 4'b0;
            PC_plus_imm_out_MEMWB <= 32'b0;
            Load_control_data_out_MEMWB <= 32'b0;
            ALU_res_out_MEMWB <= 32'b0;
            imm_out_MEMWB <= 32'b0;
            PC_plus_4_out_MEMWB <= 32'b0;
            Rd_addr_out_MEMWB <= 5'b0;
            inst_out_MEMWB <= 0;
        end
        else begin
            WB_out_MEMWB <= WB_in_MEMWB;
            PC_plus_imm_out_MEMWB <= PC_plus_imm_in_MEMWB;
            Load_control_data_out_MEMWB <= Load_control_data_in_MEMWB;
            ALU_res_out_MEMWB <= ALU_res_in_MEMWB;
            imm_out_MEMWB <= imm_in_MEMWB;
            PC_plus_4_out_MEMWB <= PC_plus_4_in_MEMWB;
            Rd_addr_out_MEMWB <= Rd_addr_in_MEMWB;
            inst_out_MEMWB <= inst_in_MEMWB;
        end
    end
endmodule
```

1.2.10 Write Back (WB)



本模块用于选择前面各模块产生的计算结果，传至 Rd 寄存器

由以下模块组成：

- Mux5to1

共选择五种数据

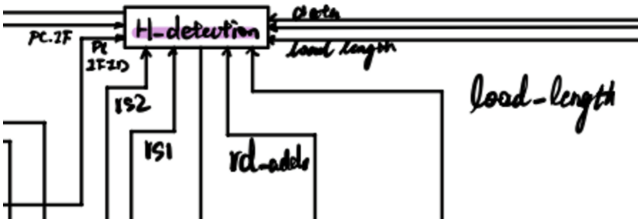
- ALU 计算结果
- PC + imm，auipc 指令结果
- Load_data，load 指令从内存中读取的数据
- imm，lui 指令结果
- PC + 4，jal、jalr 指令返回寄存器的值
- Rd_addr，写入寄存器的地址

整体实现如下：

```
module WB(
    input [3:0]         WB_in_WB,
    input [31:0]        PC_plus_imm_in_WB,
    input [31:0]        Load_control_data_in_WB,
    input [31:0]        ALU_res_in_WB,
    input [31:0]        imm_in_WB,
    input [31:0]        PC_plus_4_in_WB,
    input [31:0]        inst_in_WB,
    output [31:0]       Rd_data_out_WB,
    output              RegWrite_out_WB
    );

    assign RegWrite_out_WB = WB_in_WB[0];
    Mux5to1 WB1(
        .data1(ALU_res_in_WB),
        .data2(Load_control_data_in_WB),
        .data3(PC_plus_4_in_WB),
        .data4(imm_in_WB),
        .data5(PC_plus_imm_in_WB),
        .control(WB_in_WB[3:1]),

        .out(Rd_data_out_WB)
    );
endmodule
```

1.3 冲突解决模块实现

1.3.1 Harzard Detection



本模块用于处理 Load use 情况下，产生 stall，分为两种情况：

- Load 指令后紧跟一条指令需要使用 load 结果

```
if(Load_len_IDEX_in_HDT != 3'b111 & (Rs1 == Rd_EX | Rs2 == Rd_EX))
    Stall_out_HDT_tmp = 1;
```

  - `Load_len_IDEX_in_HDT != 3'b111` 该指令为 load 指令
  - `(Rs1 == Rd_EX | Rs2 == Rd_EX)` 当前指令所需寄存器和 load 指令目标寄存器相同

  此时将 `Stall` 信号置为 1，使 `IDEX` 寄存器停一拍即可

- Load 指令后第二条指令需要使用 load 结果

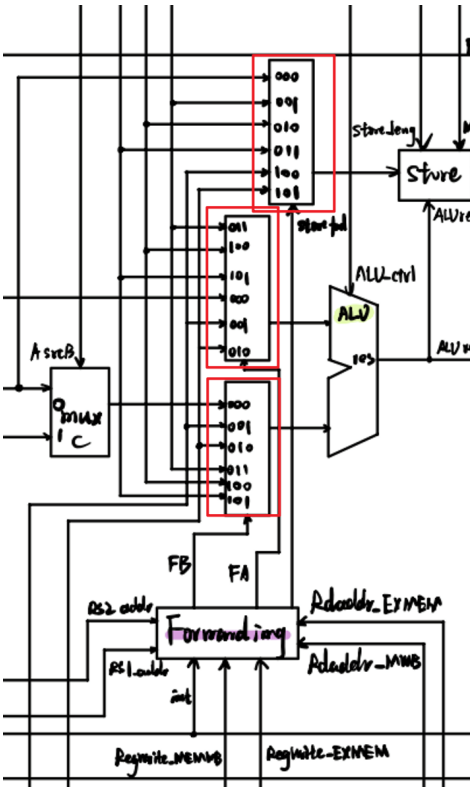
```
if(Load_len_EX_in_HDT != 3'b111 & (Rs1 == Rd_MEM | Rs2 == Rd_MEM))
    bubble_tmp = 1
```

  此时将 `bubble` 信号置为 1，使 PC 寄存器停一拍即可，相当于前一种情况的 `stall` 往前顺延一个周期

整体实现代码如下：

```
module Harzard_detection(
    input [4:0]     Rs1,                        // ID发出
    input [4:0]     Rs2,                        // ID发出
    input [4:0]     Rd_EX,                      // EX发出
    input [4:0]     Rd_MEM,                     // MEM发出
    input [2:0]     Load_len_IDEX_in_HDT,       // IDEX发出
    input [2:0]     Load_len_EXMEM_in_HDT,      // EXMEM发出
    input [31:0]    PC_IF_in_HDT,               // IF发出
    input [31:0]    PC_IFID_in_HDT,             // IFID发出

    output          Stall_out_HDT,              // 给IDEX的stall
    output [31:0]   PC_out_HDT,                 // 给IF的输出
    output          bubble
    );

    reg         Stall_out_HDT_tmp;
    reg         bubble_tmp;

    always @(*) begin       // 普通指令的load_len是3'b111
        if(Load_len_ID_in_HDT != 3'b111 & (Rs1 == Rd_EX | Rs2 ==
                                                    Rd_EX))
            Stall_out_HDT_tmp = 1;
        else if(Load_len_EX_in_HDT != 3'b111 & (Rs1 == Rd_MEM | Rs2 ==
                                                    Rd_MEM))
            bubble_tmp = 1;
        else begin
            Stall_out_HDT_tmp = 0;
            bubble_tmp = 0;
        end
    end

    assign Stall_out_HDT = Stall_out_HDT_tmp;
    assign PC_out_HDT = PC_IFID_in_HDT;
    assign bubble = bubble_tmp;

endmodule
```

1.3.2 Forwarding



本模块用于处理需要数据前递时产生的冲突，共五种情况：

- FA(FB ST) = 010

```
if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs1_addr_IDEX_in_FWD & (OPcode_EXMEM ==     5'b01100
| OPcode_EXMEM == 5'b00100))
```

  上一个周期 alu 结果

- FA(FB ST) = 011

```
if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs1_addr_IDEX_in_FWD & OPcode_EXMEM ==      5'b00101)
```

  上一个周期 auipc 结果

- FA(FB ST) = 100

```
if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs1_addr_IDEX_in_FWD & OPcode_EXMEM ==      5'b01101)
```

  上一个周期 lui 结果

- FA(FB ST) = 101

```
if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs1_addr_IDEX_in_FWD & (OPcode_EXMEM ==     5'b11011
| OPcode_EXMEM == 5'b11001))
```

  上一个周期 jal 或 jalr 结果

- FA(FB ST) = 001

```
if(RegWrite_MEMWB_in_FWD & Rd_addr_MEMWB_in_FWD != 0 & (~
(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs1_addr_IDEX_in_FWD)) &
Rd_addr_MEMWB_in_FWD == Rs1_addr_IDEX_in_FWD)
```

  WB 阶段写回结果

其中 FA、FB 信号分别用于选择 ALU 的两个输入，ST 信号用于选择 Store 指令的存储数据

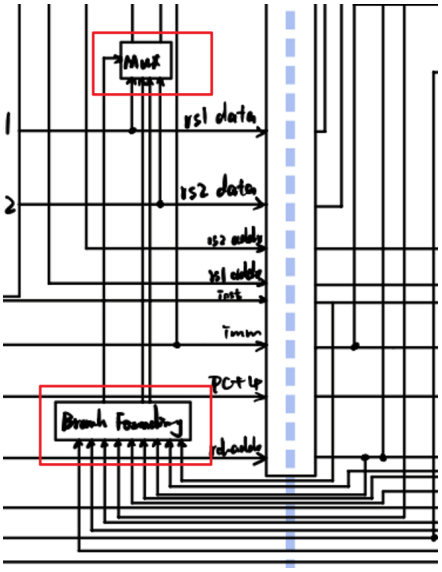
整体实现代码如下：

```
module Forwarding(
    input [4:0]     Rs1_addr_IDEX_in_FWD,       // IDEX发出
    input [4:0]     Rs2_addr_IDEX_in_FWD,       // IDEX发出
    input [4:0]     Rd_addr_EXMEM_in_FWD,       // EXMEM发出
    input [4:0]     Rd_addr_MEMWB_in_FWD,       // MEMWB发出
    input           RegWrite_EXMEM_in_FWD,      // EXMEM发出
    input           RegWrite_MEMWB_in_FWD,      // MEMWB发出
    input [31:0]    inst_EXMEM_in_FWD,
    input [31:0]    inst_IDEX_in_FWD,

    output [2:0]    Forwarding_A_out_FWD,       // FA
    output [2:0]    Forwarding_B_out_FWD,       // FB
    output [2:0]    store_fwd_out_FWD           // ST
    );
    reg [2:0]   FA;
    reg [2:0]   FB;
    reg [2:0]   ST;
    wire [4:0]  OPcode_IDEX = inst_IDEX_in_FWD[6:2];
    wire [4:0]  OPcode_EXMEM = inst_EXMEM_in_FWD[6:2];

    always @(*) begin
        // 2 上一个 alu 结果
        if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs1_addr_IDEX_in_FWD & (OPcode_EXMEM == 5'b01100 |
OPcode_EXMEM == 5'b00100))
            FA = 3'b010;
        // 3 上个 auipc 的结果
        else if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs1_addr_IDEX_in_FWD & OPcode_EXMEM == 5'b00101)
            FA = 3'b011;
        // 4 上个 lui 的结果
        else if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs1_addr_IDEX_in_FWD & OPcode_EXMEM == 5'b01101)
            FA = 3'b100;
        // 5 上个 jal 或 jalr 的结果
        else if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs1_addr_IDEX_in_FWD & (OPcode_EXMEM == 5'b11011 |
OPcode_EXMEM == 5'b11001))
            FA = 3'b101;
        // 1 WB阶段写回结果
```

```
        else if(RegWrite_MEMWB_in_FWD & Rd_addr_MEMWB_in_FWD != 0 & (~
(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 & Rd_addr_EXMEM_in_FWD ==
Rs1_addr_IDEX_in_FWD)) & Rd_addr_MEMWB_in_FWD == Rs1_addr_IDEX_in_FWD)
            FA = 3'b001;
        // 0
        else
            FA = 3'b000;

        if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs2_addr_IDEX_in_FWD & (OPcode_EXMEM == 5'b01100 |
OPcode_EXMEM == 5'b00100) & OPcode_IDEX == 5'b01100)
            FB = 3'b010;
        else if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs2_addr_IDEX_in_FWD & OPcode_EXMEM == 5'b00101 &
OPcode_IDEX == 5'b01100)
            FB = 3'b011;
        else if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs2_addr_IDEX_in_FWD & OPcode_EXMEM == 5'b01101 &
OPcode_IDEX == 5'b01100)
            FB = 3'b100;
        else if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs2_addr_IDEX_in_FWD & (OPcode_EXMEM == 5'b11011 |
OPcode_EXMEM == 5'b11001) & OPcode_IDEX == 5'b01100)
            FB = 3'b101;
        else if(RegWrite_MEMWB_in_FWD & Rd_addr_MEMWB_in_FWD != 0 & (~
(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 & Rd_addr_EXMEM_in_FWD ==
Rs2_addr_IDEX_in_FWD)) & Rd_addr_MEMWB_in_FWD == Rs2_addr_IDEX_in_FWD &
OPcode_IDEX == 5'b01100)
            FB = 3'b001;
        else
            FB = 3'b000;

        // STORE 指令
        if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs2_addr_IDEX_in_FWD & (OPcode_EXMEM == 5'b01100 |
OPcode_EXMEM == 5'b00100) & OPcode_IDEX == 5'b01000)
            ST = 3'b010;
        else if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs2_addr_IDEX_in_FWD & OPcode_EXMEM == 5'b00101 &
OPcode_IDEX == 5'b01000)
            ST = 3'b011;
        else if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs2_addr_IDEX_in_FWD & OPcode_EXMEM == 5'b01101 &
OPcode_IDEX == 5'b01000)
            ST = 3'b100;
        else if(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 &
Rd_addr_EXMEM_in_FWD == Rs2_addr_IDEX_in_FWD & (OPcode_EXMEM == 5'b11011 |
OPcode_EXMEM == 5'b11001) & OPcode_IDEX == 5'b01000)
            ST = 3'b101;
        else if(RegWrite_MEMWB_in_FWD & Rd_addr_MEMWB_in_FWD != 0 & (~
(RegWrite_EXMEM_in_FWD & Rd_addr_EXMEM_in_FWD != 0 & Rd_addr_EXMEM_in_FWD ==
Rs2_addr_IDEX_in_FWD)) & Rd_addr_MEMWB_in_FWD == Rs2_addr_IDEX_in_FWD &
OPcode_IDEX == 5'b01000)
            ST = 3'b001;
        else
            ST = 3'b000;
```

```
    end
    assign Forwarding_A_out_FWD = FA;
    assign Forwarding_B_out_FWD = FB;
    assign store_fwd_out_FWD = ST;
endmodule
```

1.3.3 Branch Forwarding



本模块用于 Branch 指令所需数据的前递，共两种情况：

- Branch 所需数据来自上一个周期 ALU 计算结果

  ```
  if(Rs1_addr == Rd_addr_IDEX_in_BFWD & Rd_addr_IDEX_in_BFWD
  != 0 & RegWrite_IDEX_in_BFWD)
  ```

  BFWD_A (BFWD_B) = 1

  BFWD_A_data (BFWD_B_data) = ALU_res_EX

- Branch 所需数据来自上上个周期 ALU 计算结果

  ```
  if(Rs1_addr == Rd_addr_EXMEM_in_BFWD &
  Rd_addr_EXMEM_in_BFWD != 0 & RegWrite_EXMEM_in_BFWD)
  ```

  BFWD_A (BFWD_B) = 1

  BFWD_A_data (BFWD_B_data) = ALU_res_MEM

- Branch 所需数据来自上上个周期 Load 指令结果

  ```
  if(Opcode_EXMEM == 5'b00000)
  ```

  BFWD_A_data (BFWD_B_data) = Load_data

整体实现代码如下：

```
module Branch_Forwarding(
    input [4:0]     Rd_addr_IDEX_in_BFWD,           // IDEX发出
    input [4:0]     Rd_addr_EXMEM_in_BFWD,          // EXMEM发出
    input [31:0]    ALU_res_EX_in_BFWD,             // EX发出
    input [31:0]    ALU_res_MEM_in_BFWD,            // MEM发出
```

```
    input [31:0]    Load_data_in_BFWD,              // load_control发出
    input           RegWrite_IDEX_in_BFWD,          // IDEX发出
    input           RegWrite_EXMEM_in_BFWD,         // EXMEM发出
    input [31:0]    inst_EXMEM_in_FWD,              // EXMEM发出
    input [31:0]    inst_in_BFWD,                   // IDEX发出

    output          BFWD_A,
    output          BFWD_B,
    output [31:0]   BFWD_A_data,
    output [31:0]   BFWD_B_data
    );

    reg         BFWD_A_tmp;
    reg         BFWD_B_tmp;
    reg [31:0]  BFWD_A_data_tmp;
    reg [31:0]  BFWD_B_data_tmp;
    wire [4:0]  Opcode = inst_in_BFWD[6:2];
    wire [4:0]  Opcode_EXMEM = inst_EXMEM_in_FWD[6:2];
    wire [4:0]  Rs1_addr = inst_in_BFWD[19:15];
    wire [4:0]  Rs2_addr = inst_in_BFWD[24:20];

    always @(*) begin
        if(Opcode == 5'b11000)begin
            if(Rs1_addr == Rd_addr_IDEX_in_BFWD & Rd_addr_IDEX_in_BFWD
!= 0 & RegWrite_IDEX_in_BFWD)               begin
                BFWD_A_tmp = 1;
                BFWD_A_data_tmp = ALU_res_EX_in_BFWD;
            end
            else if(Rs1_addr == Rd_addr_EXMEM_in_BFWD &
      Rd_addr_EXMEM_in_BFWD != 0 & RegWrite_EXMEM_in_BFWD)
            begin
                BFWD_A_tmp = 1;
                BFWD_A_data_tmp = ALU_res_MEM_in_BFWD;
                if(Opcode_EXMEM == 5'b00000)
                    BFWD_A_data_tmp = Load_data_in_BFWD;
                end
            else
                BFWD_A_tmp = 0;

            if(Rs2_addr == Rd_addr_IDEX_in_BFWD & Rd_addr_IDEX_in_BFWD
    != 0 & RegWrite_IDEX_in_BFWD)
            begin
                BFWD_B_tmp = 1;
                BFWD_B_data_tmp = ALU_res_EX_in_BFWD;
            end
            else if(Rs2_addr == Rd_addr_EXMEM_in_BFWD &
  Rd_addr_EXMEM_in_BFWD != 0 & RegWrite_EXMEM_in_BFWD)
            begin
                BFWD_B_tmp = 1;
                BFWD_B_data_tmp = ALU_res_MEM_in_BFWD;
                if(Opcode_EXMEM == 5'b00000)
                    BFWD_B_data_tmp = Load_data_in_BFWD;
                end
            else
                BFWD_B_tmp = 0;
        end
```

```
    end
    assign BFWD_A = BFWD_A_tmp;
    assign BFWD_B = BFWD_B_tmp;
    assign BFWD_A_data = BFWD_A_data_tmp;
    assign BFWD_B_data = BFWD_B_data_tmp;
endmodule
```

1.4 CSSTE 设计

本模块未使用 VGA 模块。

根据流水线的特性，将寄存器值、不同阶段的指令输入至 COM 串口用于 Debug

在 CSSTE 中增加了 `tx` 输出，串口模块如下：

```
module CSSTE(
    input         clk_100mhz,
    input         RSTN,
    input  [3:0]  BTN_y,
    input  [15:0] SW,
    output [3:0]  Blue,
    output [3:0]  Green,
    output [3:0]  Red,
    output        HSYNC,
    output        VSYNC,
    output [15:0] LED_out,
    output [7:0]  AN,
    output [7:0]  segment,
    output wire   tx
    );
    UART uart_inst(
        .clk        (clk_100mhz),
        .rst        (U9_rst),
        .pc_if      (U1_PC_out),

        .inst_if    (U2_spo),
        .inst_id    (inst_ID),
        .inst_ex    (inst_EX),
        .inst_mem   (inst_MEM),
        .inst_wb    (inst_WB),
        .mem_w_data (U1_Data_out),      // 写入mem数据
        .alu_res    (U1_Addr_out),      // 写入mem地址
        .mem_wen_mem(U1_MemRW),         // mem写使能
        .rd_wb      (Write_back_data), //写回数据
        .x0(Reg00), .ra(Reg01), .sp(Reg02), .gp(Reg03), .tp(Reg04),
.t0(Reg05), .t1(Reg06), .t2(Reg07), .s0(Reg08), .s1(Reg09),         .a0(Reg10),
.a1(Reg11), .a2(Reg12), .a3(Reg13), .a4(Reg14),         .a5(Reg15), .a6(Reg16),
.a7(Reg17), .s2(Reg18), .s3(Reg19),         .s4(Reg20), .s5(Reg21), .s6(Reg22),
.s7(Reg23), .s8(Reg24),         .s9(Reg25), .s10(Reg26), .s11(Reg27), .t3(Reg28),
.t4(Reg29),         .t5(Reg30), .t6(Reg31),
        .tx(tx)
    );
```

1.5 仿真测试文件

1.5.1 Pipeline CPU 仿真

为测试 Pipeline CPU，需要建立 Testbench 模块，连接 PipelineCPU、Data_mem、Inst_mem 模块

实现代码如下：

```
module Testbench(
    input clk,
    input rst
);
    /* Pipeline_CPU 中接出 */
    wire [3:0]  wea;
    wire [31:0] Addr_out;
    wire [31:0] Data_out;
    wire [31:0] PC_out_IF;
    /* RAM 接出 */
    wire [31:0] douta;
    /* ROM 接出 */
    wire [31:0] spo;

    Pipline_CPU U0(
        .clk(clk),
        .rst(rst),
        .Data_in(douta),
        .inst_in(spo),
        .wea(wea),
        .Addr_out(Addr_out),
        .Data_out(Data_out),
        .PC_out_IF(PC_out_IF)
    );

    RAM_B U1(
        .clka(~clk),
        .wea(wea),
        .addra(Addr_out[11:2]),
        .dina(Data_out),
        .douta(douta)
    );

    ROM_D U2(
        .a(PC_out_IF[11:2]),
        .spo(spo)
    );

endmodule
```

建立仿真测试文件：

```
`timescale 1ns / 1ps
module Testbench_tb();
```

```
    reg clk;
    reg rst;

    Testbench m0(.clk(clk), .rst(rst));

    initial begin
        clk = 1'b0;
        rst = 1'b1;
        #5;
        rst = 1'b0;
    end

    always #50 clk = ~clk;

endmodule
```

本次实验所使用的测试代码为 Lab4-3 所使用的代码

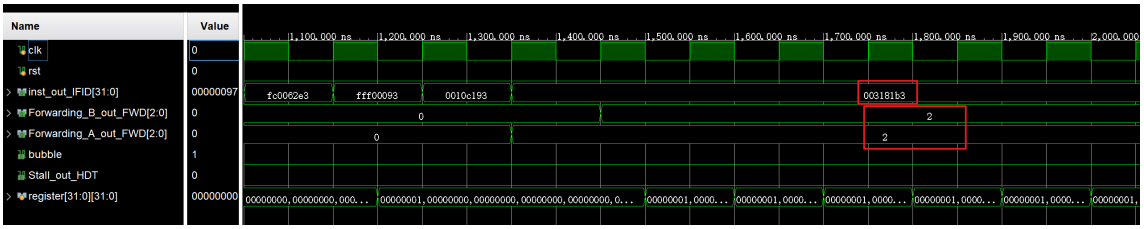
测试结果见"实验结果分析"部分

2 实验结果与分析

2.1 仿真结果分析

2.1.1 Forwarding

- 上一周期 ALU 运算结果 Forwarding



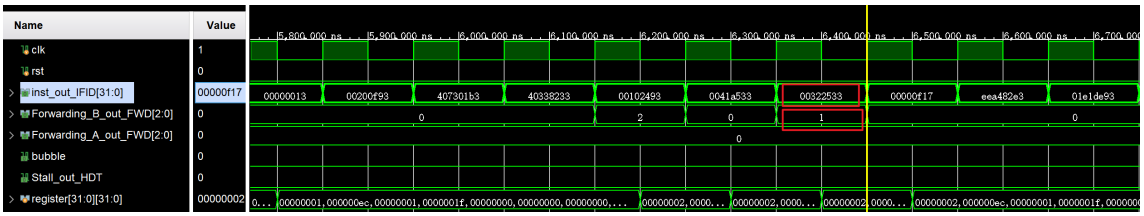
此时开始执行连续加法指令：

0x4c	0x0010C193	xori x3 x1 1	xori x3, x1, 1 # x3=FFFFFFFE
0x50	0x003181B3	add x3 x3 x3	add x3, x3, x3 # x3=FFFFFFFC
0x54	0x003181B3	add x3 x3 x3	add x3, x3, x3 # x3=FFFFFFF8
0x58	0x003181B3	add x3 x3 x3	add x3, x3, x3 # x3=FFFFFFF0
0x5c	0x003181B3	add x3 x3 x3	add x3, x3, x3 # x3=FFFFFFE0
0x60	0x003181B3	add x3 x3 x3	add x3, x3, x3 # x3=FFFFFFC0
0x64	0x003181B3	add x3 x3 x3	add x3, x3, x3 # x3=FFFFFF80
0x68	0x003181B3	add x3 x3 x3	add x3, x3, x3 # x3=FFFFFF00

需要将上一周期的计算结果同时前递至当前周期运算数据 Rs1 和 Rs2

因此 FA 和 FB 均为 2，符合预期

- 上上个周期 Rd 寄存器结果 Forwarding

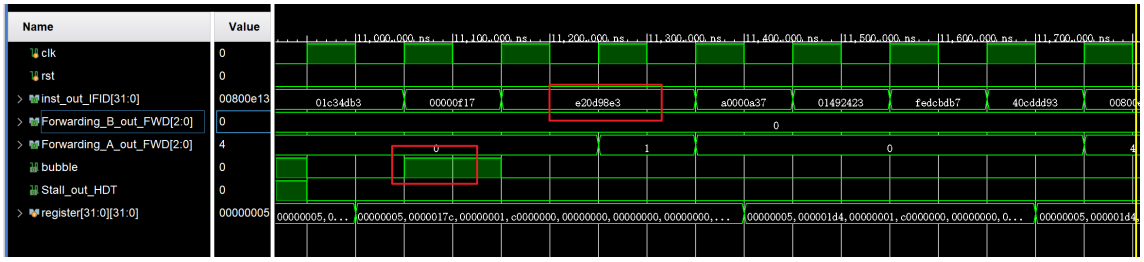


此时指令为：

0x110	0x40338233	sub x4 x7 x3	sub x4, x7, x3 # x4=40000000
0x114	0x00102493	slti x9 x0 1	slti x9, x0, 1 # x9=00000001
0x118	0x0041A533	slt x10 x3 x4	slt x10, x3, x4
0x11c	0x00322533	slt x10 x4 x3	slt x10, x4, x3 # x10=00000000
0x120	0x00000F17	auipc x30 0	auipc x30, 0

`x3` 寄存器的计算结果此时在 `WB` 阶段，需要前递给 `EX` 阶段

- Branch 指令 Forwarding

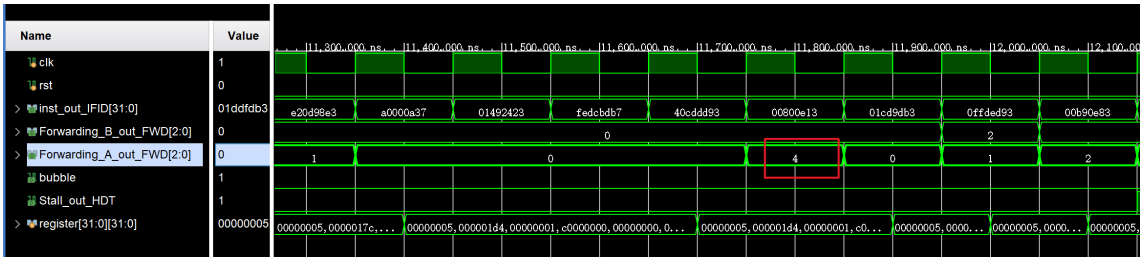


此时指令为：

			x28=mem[0x20]=C0000000
0x1d0	0x01C34DB3	xor x27 x6 x28	xor x27, x6, x28 # x27=00000000
0x1d4	0x00000F17	auipc x30 0	auipc x30, 0
0x1d8	0xE20D98E3	bne x27 x0 -464	bnez x27, dummy
0x1dc	0xA0000A37	lui x20 655360	lui x20, 0xA0000 # x20=A0000000

Branch 指令需要使用上上个周期指令的计算结果，因此需要在 Branch 指令之前产生一个 bubble，防止读取当前寄存器错误数据而发生不必要的跳转

- 上周期 Lui 指令 Forwarding



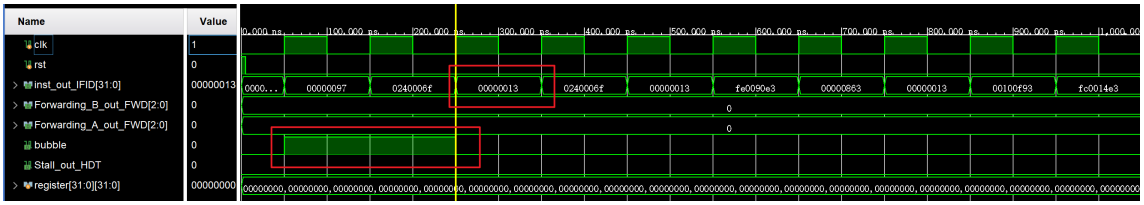
此时指令为：

0x1e0	0x01492423	sw x20 8(x18)	sw x20, 8(x18) # mem[0x28]=A0000000
0x1e4	0xFEDCBDB7	lui x27 1043915	lui x27, 0xFEDCB # x27=FEDCB000
0x1e8	0x40CDDD93	srai x27 x27 12	srai x27, x27, 12 # x27=FFFFEDCB
0x1ec	0x00800E13	addi x28 x0 8	li x28, 8

需要将上一个周期 Lui 指令结果前递至当前周期，因此 FA = 4

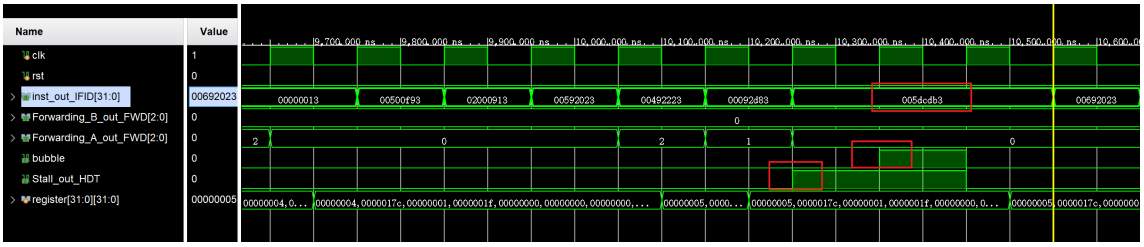
2.1.2 Stall & Bubble

- jal 指令的 Flush



如果 jal 发生跳转，需要停顿并且将 IF 已经获取到的错误指令 Flush，改为 nop 指令

- Load-use 指令

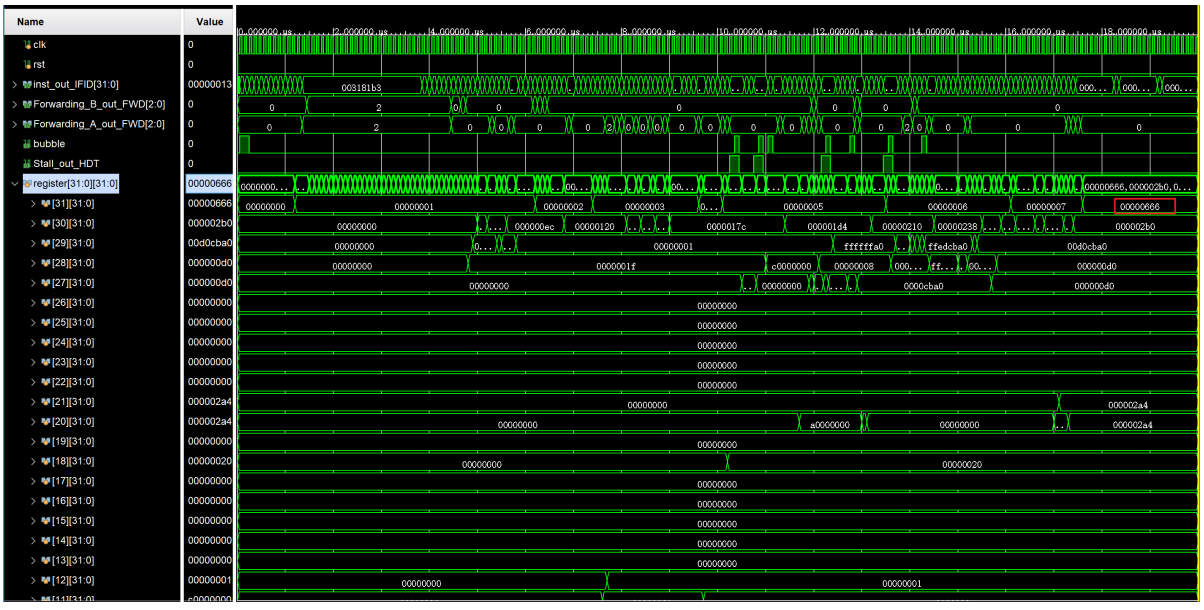


此时指令为：

```
0x1bc    0x00492223    sw x4 4(x18)      sw x4, 4(x18) # mem[0x24]=40000000
0x1c0    0x00092D83    lw x27 0(x18)     lw x27, 0(x18) #
                                         x27=mem[0x20]=F8000000
0x1c4    0x005DCDB3    xor x27 x27 x5    xor x27, x27, x5 # x27=00000000
0x1c8    0x00692023    sw x6 0(x18)      sw x6, 0(x18) # mem[0x20]=C0000000
```

由于 Load 指令从内存中读取数据需要一个周期，因此只使用前递无法解决 Load-use 冒险，需要停顿一个周期，再进行前递

2.1.3 整体仿真结果



最后 x31 寄存器值保持在 x666，符合预期结果

2.2 上板验证结果分析

- 初始状态

```
RV32I Pipelined CPU

==================================== If ======================================
pc: 00000000    inst: 00000097
==================================== Id ======================================
pc: 00000000    inst: 00000000  valid: 0
x0: 00000000    ra: 00000000    sp: 00000000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000000
==================================== Ex ======================================
pc: 00000000    inst: 00000000  valid: 0
rd:  00 rs1: 00 rs2: 00 rs1_val: 00000000       rs2_val: 00000000       reg_wen: 0
is_imm: 0       imm: 00000000
mem_wen: 0      mem_ren: 0      is_branch: 0    is_jal: 0       is_jalr: 0
is_auipc: 0     is_lui: 0       alu_ctrl: 0     cmp_ctrl: 0
==================================== Ma ======================================
pc: 00000000    inst: 00000000  valid: 0
rd:  00 reg_wen: 0      mem_w_data: 00000000    alu_res: 00000000
mem_wen: 0      mem_ren: 0      is_jal: 0       is_jalr: 0
==================================== Wb ======================================
pc: 00000000    inst: 00000000  valid: 0
rd:  00 reg_wen: 0      reg_w_data: 00000000
```

- 跳转

```
RV32I Pipelined CPU

==================================== If ======================================
pc: 0000002C    inst: 00000863
==================================== Id ======================================
pc: 00000000    inst: FE0090E3  valid: 0
x0: 00000000    ra: 00000000    sp: 00000000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000000
==================================== Ex ======================================
pc: 00000000    inst: 00000013  valid: 0
rd:  00 rs1: 00 rs2: 00 rs1_val: 00000000       rs2_val: 00000000       reg_wen: 0
is_imm: 0       imm: 00000000
mem_wen: 0      mem_ren: 0      is_branch: 0    is_jal: 0       is_jalr: 0
is_auipc: 0     is_lui: 0       alu_ctrl: 0     cmp_ctrl: 0
==================================== Ma ======================================
pc: 00000000    inst: 0240006F  valid: 0
rd:  00 reg_wen: 0      mem_w_data: 00000000    alu_res: 00000000
mem_wen: 0      mem_ren: 0      is_jal: 0       is_jalr: 0
==================================== Wb ======================================
pc: 00000000    inst: 00000013  valid: 0
rd:  00 reg_wen: 0      reg_w_data: 00000000
```

- x31 = 1

```
RV32I Pipelined CPU

==================================== If ======================================
pc: 00000054    inst: 003181B3
==================================== Id ======================================
pc: 00000000    inst: 003181B3  valid: 0
x0: 00000000    ra: 00000000    sp: 00000000    gp: 00000000    tp: 00000000
t0: 00000000    t1: 00000000    t2: 00000000    s0: 00000000    s1: 00000000
a0: 00000000    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000000    t3: 00000000    t4: 00000000
t5: 00000000    t6: 00000001
==================================== Ex ======================================
pc: 00000000    inst: 0010C193  valid: 0
rd:  00 rs1: 00 rs2: 00 rs1_val: 00000000       rs2_val: 00000000       reg_wen: 0
is_imm: 0       imm: 00000000
mem_wen: 0      mem_ren: 0      is_branch: 0    is_jal: 0       is_jalr: 0
is_auipc: 0     is_lui: 0       alu_ctrl: 0     cmp_ctrl: 0
==================================== Ma ======================================
pc: 00000000    inst: FFF00093  valid: 0
rd:  00 reg_wen: 0      mem_w_data: 01000000    alu_res: FFFFFFFF
mem_wen: 0      mem_ren: 0      is_jal: 0       is_jalr: 0
==================================== Wb ======================================
pc: 00000000    inst: FC0062E3  valid: 0
rd:  00 reg_wen: 0      reg_w_data: 00000000
```

- x31 = 2

```
RV32I Pipelined CPU

==================================== If ======================================
pc: 00000124    inst: EEA482E3
==================================== Id ======================================
pc: 00000000    inst: 00000F17  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: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000000    t3: 0000001F    t4: 00000001
t5: 000000EC    t6: 00000002
==================================== Ex ======================================
pc: 00000000    inst: 00322533  valid: 0
rd:  00 rs1: 00 rs2: 00 rs1_val: 00000000       rs2_val: 00000000       reg_wen: 0
is_imm: 0       imm: 00000000
mem_wen: 0      mem_ren: 0      is_branch: 0    is_jal: 0       is_jalr: 0
is_auipc: 0     is_lui: 0       alu_ctrl: 0     cmp_ctrl: 0
==================================== Ma ======================================
pc: 00000000    inst: 0041A533  valid: 0
rd:  00 reg_wen: 0      mem_w_data: 00000000    alu_res: 00000000
mem_wen: 0      mem_ren: 0      is_jal: 0       is_jalr: 0
==================================== Wb ======================================
pc: 00000000    inst: 00102493  valid: 0
rd:  01 reg_wen: 0      reg_w_data: 00000000
```

- x31 = 3

```
RV32I Pipelined CPU

====================================== If ======================================
pc: 00000150    inst: 00B57533
====================================== Id ======================================
pc: 00000000    inst: 0FF57513  valid: 0
x0: 00000000    ra: FFFFFFFF    sp: 00000000    gp: 40000000    tp: 40000000
t0: F8000000    t1: C0000000    t2: 80000000    s0: 00000001    s1: 00000001
a0: 00000001    a1: 00000000    a2: 00000000    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000000    t3: 0000001F    t4: 00000001
t5: 00000120    t6: 00000003
====================================== Ex ======================================
pc: 00000000    inst: 0030A633  valid: 0
rd:  00 rs1: 00 rs2: 00 rs1_val: 00000000       rs2_val: 00000000       reg_wen: 0
is_imm: 0       imm: 00000000
mem_wen: 0      mem_ren: 0      is_branch: 0    is_jal: 0       is_jalr: 0
is_auipc: 0     is_lui: 0       alu_ctrl: 0     cmp_ctrl: 0
====================================== Ma ======================================
pc: 00000000    inst: 0012A5B3  valid: 0
rd:  00 reg_wen: 0      mem_w_data: 0000FF00    alu_res: 00000001
mem_wen: 0      mem_ren: 0      is_jal: 0       is_jalr: 0
====================================== Wb ======================================
pc: 00000000    inst: 0030A513  valid: 0
rd:  01 reg_wen: 0      reg_w_data: 00000000
```

- x31 = 4

```
RV32I Pipelined CPU

====================================== If ======================================
pc: 000001B8    inst: 00592023
====================================== Id ======================================
pc: 00000000    inst: 02000913  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: 00000000    s3: 00000000
s4: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000000    t3: 0000001F    t4: 00000001
t5: 0000017C    t6: 00000004
====================================== Ex ======================================
pc: 00000000    inst: 00500F93  valid: 0
rd:  00 rs1: 00 rs2: 00 rs1_val: 00000000       rs2_val: 00000000       reg_wen: 0
is_imm: 0       imm: 00000000
mem_wen: 0      mem_ren: 0      is_branch: 0    is_jal: 0       is_jalr: 0
is_auipc: 0     is_lui: 0       alu_ctrl: 0     cmp_ctrl: 0
====================================== Ma ======================================
pc: 00000000    inst: 00000013  valid: 0
rd:  00 reg_wen: 0      mem_w_data: 00000000    alu_res: 00000000
mem_wen: 0      mem_ren: 0      is_jal: 0       is_jalr: 0
====================================== Wb ======================================
pc: 00000000    inst: 00000013  valid: 0
rd:  00 reg_wen: 0      reg_w_data: 00000000
```

- x31 = 5

```
RV32I Pipelined CPU

====================================== If ======================================
pc: 000001C8    inst: 00692023
====================================== Id ======================================
pc: 00000000    inst: 005DCDB3  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: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:00000000    t3: 0000001F    t4: 00000001
t5: 0000017C    t6: 00000005
====================================== Ex ======================================
pc: 00000000    inst: 00000000  valid: 0
rd:  00 rs1: 00 rs2: 00 rs1_val: 00000000       rs2_val: 00000000       reg_wen: 0
is_imm: 0       imm: 00000000
mem_wen: 0      mem_ren: 0      is_branch: 0    is_jal: 0       is_jalr: 0
is_auipc: 0     is_lui: 0       alu_ctrl: 0     cmp_ctrl: 0
====================================== Ma ======================================
pc: 00000000    inst: 00092D83  valid: 0
rd:  00 reg_wen: 0      mem_w_data: 00000000    alu_res: 00000020
mem_wen: 0      mem_ren: 0      is_jal: 0       is_jalr: 0
====================================== Wb ======================================
pc: 00000000    inst: 00492223  valid: 0
rd:  04 reg_wen: 0      reg_w_data: 00000000
```

- x31 = 6

```
RV32I Pipelined CPU

====================================== If ======================================
pc: 00000228    inst: 01CEEEB3
====================================== Id ======================================
pc: 00000000    inst: 01CEEEB3  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: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:0000CBA0    t3: 000000ED    t4: 000000FF
t5: 00000210    t6: 00000006
====================================== Ex ======================================
pc: 00000000    inst: 008E9E93  valid: 0
rd:  00 rs1: 00 rs2: 00 rs1_val: 00000000       rs2_val: 00000000       reg_wen: 0
is_imm: 0       imm: 00000000
mem_wen: 0      mem_ren: 0      is_branch: 0    is_jal: 0       is_jalr: 0
is_auipc: 0     is_lui: 0       alu_ctrl: 0     cmp_ctrl: 0
====================================== Ma ======================================
pc: 00000000    inst: 00000000  valid: 0
rd:  00 reg_wen: 0      mem_w_data: 00000000    alu_res: 00000000
mem_wen: 0      mem_ren: 0      is_jal: 0       is_jalr: 0
====================================== Wb ======================================
pc: 00000000    inst: 00B94E83  valid: 0
rd:  1F reg_wen: 0      reg_w_data: 00000000
```

- x31 = 7

```
RV32I Pipelined CPU

==================================== If ======================================
pc: 00000290    inst: D6107CE3
==================================== Id ======================================
pc: 00000000    inst: 00000F17  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: 00000000    s5: 00000000    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10:00000000    s11:000000D0    t3: 000000D0    t4: 00D0CBA0
t5: 00000278    t6: 00000007
==================================== Ex ======================================
pc: 00000000    inst: 00000013  valid: 0
rd:  00 rs1: 00 rs2: 00 rs1_val: 00000000       rs2_val: 00000000       reg_wen: 0
is_imm: 0       imm: 00000000
mem_wen: 0      mem_ren: 0      is_branch: 0    is_jal: 0       is_jalr: 0
is_auipc: 0     is_lui: 0       alu_ctrl: 0     cmp_ctrl: 0
==================================== Ma ======================================
pc: 00000000    inst: 00145663  valid: 0
rd:  00 reg_wen: 0      mem_w_data: 000000FF    alu_res: 00000000
mem_wen: 0      mem_ren: 0      is_jal: 0       is_jalr: 0
==================================== Wb ======================================
pc: 00000000    inst: D800D6E3  valid: 0
rd:  1F reg_wen: 0      reg_w_data: 00000000
```

- x31 = 666

```
RV32I Pipelined CPU

==================================== If ======================================
pc: 00000014    inst: 00000013
==================================== Id ======================================
pc: 00000000    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: 00000000    inst: 00000013  valid: 0
rd:  00 rs1: 00 rs2: 00 rs1_val: 00000000       rs2_val: 00000000       reg_wen: 0
is_imm: 0       imm: 00000000
mem_wen: 0      mem_ren: 0      is_branch: 0    is_jal: 0       is_jalr: 0
is_auipc: 0     is_lui: 0       alu_ctrl: 0     cmp_ctrl: 0
==================================== Ma ======================================
pc: 00000000    inst: 00000013  valid: 0
rd:  00 reg_wen: 0      mem_w_data: 00000000    alu_res: 00000000
mem_wen: 0      mem_ren: 0      is_jal: 0       is_jalr: 0
==================================== Wb ======================================
pc: 00000000    inst: 00000013  valid: 0
rd:  00 reg_wen: 0      reg_w_data: 00000000
```

寄存器 x31 的值维持在 x666 不变，符合预期

3 讨论、心得

3.1 本次实验所遇问题

本次实验所遇到的问题基本均集中在解决冲突的过程中

3.1.1 Forwarding 的理解

在完成基础流水线 CPU 后，根据理论课中记录的笔记实现了第一个版本的 Forwarding，该版本仅包含对上个周期以及上上个周期 ALU 计算结果的前递，完成后便开始实现 Harzard detection 模块，在所有冲突模块完成后的仿真过程中，发现了许多问题，但无论怎么更改均无法解决。最后在同学的提示下，发现问题根本出现在最先完成的 Forwarding 模块未考虑上个周期 jal、lui、auipc 指令结果的前递，即

完成了不完整的 Forwarding 模块，在添加不同情况下的前递后问题得到解决。

应该少参考，根据自己的理解来设计，防止另一种思维产生的思维惯性造成不良后果，在此过程中花费了大量的时间，甚至额外设计出三种 forward，最后一团乱麻

3.1.2 Flush 与 Stall 的理解

由于本次实验最开始仅实现了 load-use 的 stall，因此无法控制 Branch 指令在跳转的情况下读入的错误指令，最后了解到需要在 IFID 寄存器输入 Flush，将错误指令冲掉，其控制来源是发生跳转，即 J 型指令或 B型指令

3.2 心得

本次实验主要内容为实现一个 RISC-V 指令集架构的五级流水线 CPU，相比于 Lab4 单周期的设计，流水线 CPU 的设计难度上升很多档次，从基本的五个周期模块，四个流水线寄存器实现基本的流水线 CPU，较为简单，只需在完成数据通路图后进行连线即可。

但在完成 Harzard Detection，Forwarding，Branch Forwarding 三个解决冲突模块的过程中，难度较大，可参考的资料很少，在仿真和调试花费了很多时间（仿真超过 150 次），对流水线 CPU 的特性与理解，以及Verilog 语言熟练度要求很高。

通过本次实验，深入了解了流水线 CPU 的原理及特性，是对理论课内容的绝佳实现和延拓，课程与实验内容结合良好，在提高工程能力的同时，加深了对课程内容的理解，收获颇丰，对考试以及之后的计算机体系结构课程学习大有裨益，成就感满满！非常感谢两位助教在实验过程中的悉心指导与解答，同时也感谢同学们的帮助！

4 思考题

1. **基于你完成的流水线，对于以下两段代码分别分析：不同指令之间是否存在冲突（如果有，请逐条列出）、在你的流水线上运行的 CPI 为何**

```
addi    x1, x0, 0
addi    x2, x0, -1
addi    x3, x0, 1
addi    x4, x0, -1
addi    x5, x0, 1
addi    x6, x0, -1
addi    x1, x1, 0
addi    x2, x2, 1
addi    x3, x3, -1
addi    x4, x4, 1
addi    x5, x5, -1
addi    x6, x6, 1
```

无冲突，CPI = 5

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

存在冲突：

- `addi  x1, x0, 1` 与 `addi  x2, x1, 2` 和 `addi  x3, x1, 3` 指令

  需要将 `x1` 前递至指令 2 的 `Rs1`，此时 FA = 2，FB = 0

  需要将 `x1` 前递至指令 3 的 `Rs1`，此时 FA = 2，FB = 0

- `addi  x3, x1, 3` 与 `addi  x4, x3, 4` 指令

  需要将 `x3` 前递至指令 2 的 `Rs1`，此时 FA = 2，FB = 0

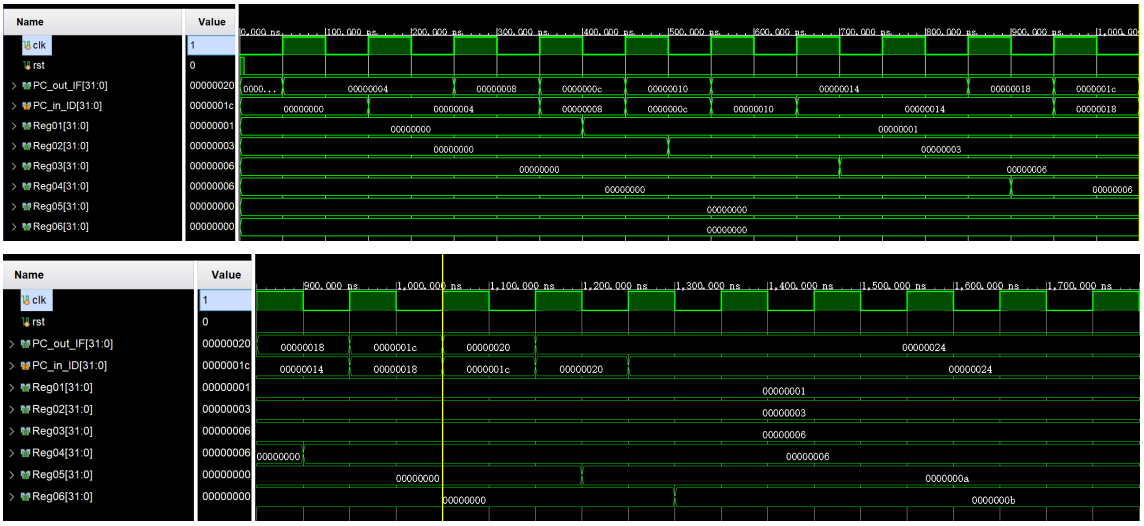
CPI = 5

2. **请根据你的实现，在 testbench 上仿真以下代码，给出仿真结果，并写出完成所有指令用了多少拍，必须给出的信号有 `clk, IF-PC, ID-PC` 以及所有用到的寄存器值。 请务必注意调整数制为十六进制，缩放能够看到所有信号值！！！**

   如果你实现的流水线支持 Forwarding

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

仿真结果如下：



完成所有指令共使用 13 拍