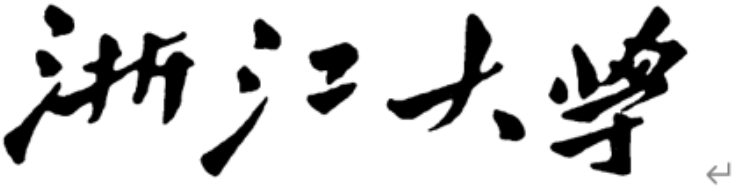


课程名称:	**计算机组成与设计**
姓 名:	杨吉祥
学 院:	计算机科学与技术学院
系:	竺可桢学院图灵班
专 业:	计算机科学与技术
学 号:	3230106222
指导教师:	刘海风

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

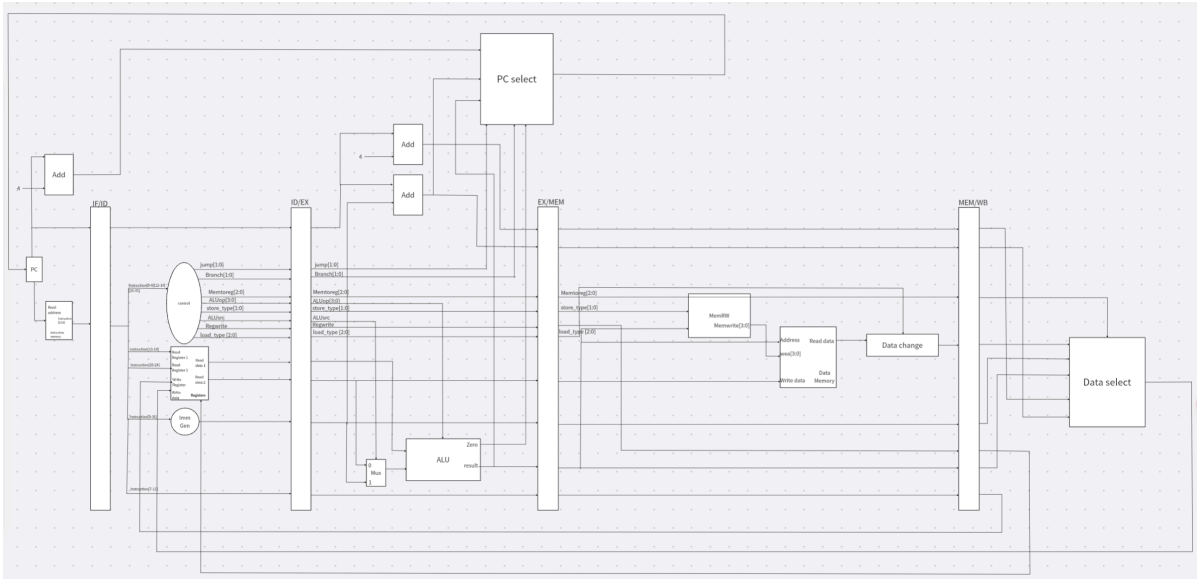
学生姓名： 杨吉祥 专业： 计算机科学与技术 学号： 3230106222

报告日期：12.25

一、操作方法与实验步骤

1.1 datapath图

我将datapath分为IF,ID,EX,MEM,WB五个阶段和IF_ID_reg,ID_EX_reg,EX_MEM_reg,MEM_WB_reg四个寄存器,然后将这九部分分别用代码实现,我的数据冒险是用数据前递解决,控制冒险是用stall解决



1.2 IF.v

该阶段进行取指令,stall_IF为1表示正在执行跳转指令,由于此时的PC_IF已经不是正在执行的跳转指令的地址,PC_ID_EX才是该跳转指令的地址,所以根据stall_IF去给PC1正确赋值

```
`timescale 1ns/1ps
module IF(
    input clk_IF,
    input rst_IF,
    input stall_IF,
    input [31:0]PC_ID_EX,
    input [31:0] PC_in_IF,
    input [31:0]PC1_IF,
    input [31:0]PC2_IF,
    input [1:0]jump_IF,
    input [1:0]branch_IF,
    input zero_IF,
    output [31:0] PC_out_IF
);

wire [31:0]PC_next;
wire [31:0]PC1;
reg [31:0]PC_out;
```

```
assign PC_out_IF = PC_out;
assign PC1=(stall_IF==0)?PC_in_IF:PC_ID_EX;

PC_select X0(
    .PC1(PC1+4),
    .PC2(PC1_IF),
    .ALU_result(PC2_IF),
    .jump(jump_IF),
    .branch(branch_IF),
    .ALU_zero(zero_IF),
    .PC_out(PC_next)
);

always @(posedge clk_IF or posedge rst_IF) begin
    if(rst_IF) begin
        PC_out <= 0;
    end
    else begin
        PC_out <= PC_next;
    end
end
endmodule
```

1.3 IF_ID_reg.v

该寄存器比其他中间寄存器多了一个stall信号,表示正在执行跳转指令,那这时候就不能执行跳转指令接下来的指令,所以当stall为1时,该寄存器输出的指令为32'h13,表示的是add x0,x0,x0,相当于nop

```
`timescale 1ns/1ps
module IF_ID_reg(
    input clk_IF_ID,
    input rst_IF_ID,
    input stall_IF_ID,
    input [31:0]PC_in_IF_ID,
    input [31:0]inst_in_IF_ID,
    output [31:0]PC_out_IF_ID,
    output [31:0]inst_out_IF_ID
);

reg [31:0]PC_out;
reg [31:0]inst_out;

assign PC_out_IF_ID = PC_out;
assign inst_out_IF_ID = inst_out;

always @(posedge clk_IF_ID or posedge rst_IF_ID) begin
    if(rst_IF_ID) begin
        PC_out <= 0;
        inst_out <= 0;
    end
    else begin
        if(stall_IF_ID) begin
            PC_out <= PC_out;
            inst_out <= 32'h00000013;
```

```
        end
        else begin
            PC_out <= PC_in_IF_ID;
            inst_out <= inst_in_IF_ID;
        end
    end
end

endmodule
```

1.4 ID.v

该阶段进行译码,生成多个控制信号,生成立即数以及读写寄存器,需要注意的是因为各个阶段的中间寄存器是在上升沿进行写,所以ID阶段中的寄存器改成在下降沿写,这样WB阶段上升沿准备好要写的数据,下降沿就能将数据写入,不需要再等一个时钟周期

```
`timescale 1ns/1ps
module ID(
    input clk_ID,
    input rst_ID,
    input [31:0]inst_in_ID,
    input [31:0]waddr_ID,
    input [31:0]wdata_ID,
    input Regwrite_ID,
    output [31:0]rs1_ID,
    output [31:0]rs2_ID,
    output [31:0]imm_ID,
    output [1:0]jump_ID,
    output [1:0]branch_ID,
    output [2:0]Memtoreg_ID,
    output [3:0]ALUop_ID,
    output ALUsrc_ID,
    output RegWrite_ID,
    output [1:0]store_type_ID,
    output [2:0]load_type_ID,
    output is_auipc,
    output is_lui,
    output [31:0]reg0,
    output [31:0]reg1,
    output [31:0]reg2,
    output [31:0]reg3,
    output [31:0]reg4,
    output [31:0]reg5,
    output [31:0]reg6,
    output [31:0]reg7,
    output [31:0]reg8,
    output [31:0]reg9,
    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
);

SCPU_ctrl X0(
    .OPcode(inst_in_ID[6:0]),
    .Fun3(inst_in_ID[14:12]),
    .Fun7(inst_in_ID[31:25]),
    .ALUop(ALUop_ID),
    .ALUsrc(ALUsrc_ID),
    .Branch(branch_ID),
    .Jump(jump_ID),
    .MemtoReg(Memtoreg_ID),
    .RegWrite(RegWrite_ID),
    .load_type(load_type_ID),
    .store_type(store_type_ID),
    .is_auipc(is_auipc),
    .is_lui(is_lui)
);

Regs X1(
    .clk(clk_ID),
    .rst(rst_ID),
    .Rs1_addr(inst_in_ID[19:15]),
    .Rs2_addr(inst_in_ID[24:20]),
    .Wt_addr(waddr_ID),
    .Wt_data(wdata_ID),
    .RegWrite(Regwrite_ID),
    .Rs1_data(rs1_ID),
    .Rs2_data(rs2_ID),
    .Reg00(reg0),
    .Reg01(reg1),
    .Reg02(reg2),
    .Reg03(reg3),
    .Reg04(reg4),
    .Reg05(reg5),
    .Reg06(reg6),
    .Reg07(reg7),
    .Reg08(reg8),
    .Reg09(reg9),
    .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)
);

ImmGen X2(
    .inst_in(inst_in_ID),
    .Imm_out(imm_ID)
);

endmodule
```

1.5 ID_EX_reg.v

```
`timescale 1ns/1ps
module ID_EX_reg(
    input clk_ID_EX,
    input rst_ID_EX,
    input [31:0]PC_in_ID_EX,
    input [31:0]inst_in_ID_EX,
    input [1:0]jump_in_ID_EX,
    input [1:0]branch_in_ID_EX,
    input [2:0]MemtoReg_in_ID_EX,
    input [3:0]ALUOp_in_ID_EX,
    input [1:0]store_type_in_ID_EX,
    input [2:0]load_type_in_ID_EX,
    input ALUsrc_in_ID_EX,
    input RegWrite_in_ID_EX,
    input [31:0]rs1data_in_ID_EX,
    input [31:0]rs2data_in_ID_EX,
    input [31:0]imm_in_ID_EX,
    input [4:0]rd_in_ID_EX,
    output [31:0]PC_out_ID_EX,
    output [31:0]inst_out_ID_EX,
    output [1:0]jump_out_ID_EX,
    output [1:0]branch_out_ID_EX,
    output [2:0]MemtoReg_out_ID_EX,
    output [3:0]ALUOp_out_ID_EX,
    output [1:0]store_type_out_ID_EX,
    output [2:0]load_type_out_ID_EX,
```

```
    output ALUsrc_out_ID_EX,
    output RegWrite_out_ID_EX,
    output [31:0]rs1data_out_ID_EX,
    output [31:0]rs2data_out_ID_EX,
    output [31:0]imm_out_ID_EX,
    output [4:0]rd_out_ID_EX
);

reg [31:0]PC_out;
reg [31:0]inst_out;
reg [1:0]jump_out;
reg [1:0]branch_out;
reg [2:0]MemtoReg_out;
reg [3:0]ALUOp_out;
reg [1:0]store_type_out;
reg [2:0]load_type_out;
reg ALUsrc_out;
reg RegWrite_out;
reg [31:0]rs1data_out;
reg [31:0]rs2data_out;
reg [31:0]imm_out;
reg [4:0]rd_out;

assign PC_out_ID_EX = PC_out;
assign inst_out_ID_EX = inst_out;
assign jump_out_ID_EX = jump_out;
assign branch_out_ID_EX = branch_out;
assign MemtoReg_out_ID_EX = MemtoReg_out;
assign ALUOp_out_ID_EX = ALUOp_out;
assign store_type_out_ID_EX = store_type_out;
assign load_type_out_ID_EX = load_type_out;
assign ALUsrc_out_ID_EX = ALUsrc_out;
assign RegWrite_out_ID_EX = RegWrite_out;
assign rs1data_out_ID_EX = rs1data_out;
assign rs2data_out_ID_EX = rs2data_out;
assign imm_out_ID_EX = imm_out;
assign rd_out_ID_EX = rd_out;

always @(posedge clk_ID_EX or posedge rst_ID_EX) begin
    if(rst_ID_EX) begin
        PC_out <= 0;
        inst_out <= 0;
        jump_out <= 0;
        branch_out <= 0;
        MemtoReg_out <= 0;
        ALUOp_out <= 0;
        store_type_out <= 0;
        load_type_out <= 0;
        ALUsrc_out <= 0;
        RegWrite_out <= 0;
        rs1data_out <= 0;
        rs2data_out <= 0;
        imm_out <= 0;
        rd_out <= 0;
    end
    else begin
```

```
        PC_out <= PC_in_ID_EX;
        inst_out <= inst_in_ID_EX;
        jump_out <= jump_in_ID_EX;
        branch_out <= branch_in_ID_EX;
        MemtoReg_out <= MemtoReg_in_ID_EX;
        ALUOp_out <= ALUOp_in_ID_EX;
        store_type_out <= store_type_in_ID_EX;
        load_type_out <= load_type_in_ID_EX;
        ALUsrc_out <= ALUsrc_in_ID_EX;
        RegWrite_out <= RegWrite_in_ID_EX;
        rs1data_out <= rs1data_in_ID_EX;
        rs2data_out <= rs2data_in_ID_EX;
        imm_out <= imm_in_ID_EX;
        rd_out <= rd_in_ID_EX;
    end
end

endmodule
```

1.6 EX.v

该阶段进行ALU的运算,由于可能出现数据冲突的情况,所以需要用forwardingA和forwardingB来判断是否需要数据前递,为1代表需要数据前递,为0就不用

```
`timescale 1ns/1ps
module EX(
    input [3:0]ALUOp_in_EX,
    input ALUsrc_in_EX,
    input forwardingA,
    input forwardingB,
    input [31:0]ALU_C,
    input [31:0]ALU_D,
    input [31:0]rs1data_in_EX,
    input [31:0]rs2data_in_EX,
    input [31:0]imm_in_EX,
    output zero_out_EX,
    output [31:0]ALU_out_EX
);

wire [31:0]p;
assign p = forwardingB==0?rs2data_in_EX:ALU_D;


ALU x0(
    .A(forwardingA==0?rs1data_in_EX:ALU_C),
    .B(ALUsrc_in_EX==0?p:imm_in_EX),
    .ALU_operation(ALUOp_in_EX),
    .res(ALU_out_EX),
    .zero(zero_out_EX)
);

endmodule
```

1.7 EX_MEM_reg.v

```
`timescale 1ns/1ps
module  EX_MEM_reg(
    input clk_EX_MEM,
    input rst_EX_MEM,
    input [31:0]PC1_in_EX_MEM,
    input [31:0]PC2_in_EX_MEM,
    input [2:0]MemtoReg_in_EX_MEM,
    input [1:0]store_type_in_EX_MEM,
    input RegWrite_in_EX_MEM,
    input [2:0]load_type_in_EX_MEM,
    input [31:0]rs2data_in_EX_MEM,
    input [31:0]imm_in_EX_MEM,
    input zero_in_EX_MEM,
    input [31:0]ALU_in_EX_MEM,
    input [4:0]rd_in_EX_MEM,
    output [31:0]PC1_out_EX_MEM,
    output [31:0]PC2_out_EX_MEM,
    output [2:0]MemtoReg_out_EX_MEM,
    output [1:0]store_type_out_EX_MEM,
    output RegWrite_out_EX_MEM,
    output [2:0]load_type_out_EX_MEM,
    output [31:0]rs2data_out_EX_MEM,
    output [31:0]imm_out_EX_MEM,
    output [31:0]ALU_out_EX_MEM,
    output [4:0]rd_out_EX_MEM
);


reg [31:0] PC1_reg, PC2_reg;
reg [2:0] MemtoReg_reg;
reg [1:0] store_type_reg;
reg RegWrite_reg;
reg [2:0] load_type_reg;
reg [31:0] rs2data_reg, imm_reg;
reg [31:0] ALU_reg;
reg [4:0] rd_reg;

always @(posedge clk_EX_MEM or posedge rst_EX_MEM) begin
    if (rst_EX_MEM) begin
        PC1_reg <= 32'b0;
        PC2_reg <= 32'b0;
        MemtoReg_reg <= 3'b0;
        store_type_reg <= 2'b0;
        RegWrite_reg <= 1'b0;
        load_type_reg <= 3'b0;
        rs2data_reg <= 32'b0;
        imm_reg <= 32'b0;
        ALU_reg <= 32'b0;
        rd_reg <= 5'b0;
    end
    else begin
        PC1_reg <= PC1_in_EX_MEM;
        PC2_reg <= PC2_in_EX_MEM;
        MemtoReg_reg <= MemtoReg_in_EX_MEM;
        store_type_reg <= store_type_in_EX_MEM;
        RegWrite_reg <= RegWrite_in_EX_MEM;
```

```
        load_type_reg <= load_type_in_EX_MEM;
        rs2data_reg <= rs2data_in_EX_MEM;
        imm_reg <= imm_in_EX_MEM;
        ALU_reg <= ALU_in_EX_MEM;
        rd_reg <= rd_in_EX_MEM;
    end
end

assign PC1_out_EX_MEM = PC1_reg;
assign PC2_out_EX_MEM = PC2_reg;
assign MemtoReg_out_EX_MEM = MemtoReg_reg;
assign store_type_out_EX_MEM = store_type_reg;
assign RegWrite_out_EX_MEM = RegWrite_reg;
assign load_type_out_EX_MEM = load_type_reg;
assign rs2data_out_EX_MEM = rs2data_reg;
assign imm_out_EX_MEM = imm_reg;
assign ALU_out_EX_MEM = ALU_reg;
assign rd_out_EX_MEM = rd_reg;

endmodule
```

1.8 MEM.v

该阶段进行内存的读和写,实现与lab4相同,根据ALU_result,load_type和store_type生成四位写信号,并且对读出的数据进行修正

```
`timescale 1ns/1ps
module MEM(
    input [2:0] load_type_MEM,
    input [1:0] store_type_MEM,
    input [31:0] ALU_result_MEM,
    input [31:0] data_in_MEM,
    input [31:0] rs2data_in_MEM,
    output [31:0]Addr_out_MEM,
    output [3:0]MemWrite_out_MEM,
    output [31:0]data_change_out_MEM,
    output [31:0] data_out_MEM
);

reg [31:0]data;

assign data_out_MEM=data;
assign Addr_out_MEM=ALU_result_MEM;

Data_change x0(
    .data_in(data_in_MEM),
    .ALU_result(ALU_result_MEM),
    .load_type(load_type_MEM),
    .data_change(data_change_out_MEM)
);

MemRw x1(
    .ALU_result(ALU_result_MEM),
    .store_type(store_type_MEM),
    .MemWrite(MemWrite_out_MEM)
```

```
);

always@(*)begin
        case(ALU_result_MEM%4)
        0:data=rs2data_in_MEM;
        1:data=rs2data_in_MEM<<8;
        2:data=rs2data_in_MEM<<16;
        3:data=rs2data_in_MEM<<24;
    endcase
end

endmodule
```

1.9 MEM_WB_reg.v

```
`timescale 1ns/1ps
module MEM_WB_reg(
    input clk_MEM_WB,
    input rst_MEM_WB,
    input [31:0]PC1_in_MEM_WB,
    input [31:0]PC2_in_MEM_WB,
    input [2:0]MemtoReg_in_MEM_WB,
    input [31:0]data_change_in_MEM_WB,
    input [31:0]imm_in_MEM_WB,
    input RegWrite_in_MEM_WB,
    input [31:0]ALU_result_in_MEM_WB,
    input [4:0]rd_in_MEM_WB,
    output [31:0]PC1_out_MEM_WB,
    output [31:0]PC2_out_MEM_WB,
    output [2:0]MemtoReg_out_MEM_WB,
    output [31:0]data_change_out_MEM_WB,
    output [31:0]imm_out_MEM_WB,
    output RegWrite_out_MEM_WB,
    output [31:0]ALU_result_out_MEM_WB,
    output [4:0]rd_out_MEM_WB
);

reg [31:0] PC1_reg, PC2_reg, data_change_reg, imm_reg, ALU_result_reg;
reg [2:0] MemtoReg_reg;
reg RegWrite_reg;
reg [4:0] rd_reg;

always @(posedge clk_MEM_WB or posedge rst_MEM_WB) begin
    if (rst_MEM_WB) begin
        PC1_reg <= 32'b0;
        PC2_reg <= 32'b0;
        MemtoReg_reg <= 3'b0;
        data_change_reg <= 32'b0;
        imm_reg <= 32'b0;
        RegWrite_reg <= 1'b0;
        ALU_result_reg <= 32'b0;
        rd_reg <= 5'b0;
    end else begin
        PC1_reg <= PC1_in_MEM_WB;
        PC2_reg <= PC2_in_MEM_WB;
```

```
        MemtoReg_reg <= MemtoReg_in_MEM_WB;
        data_change_reg <= data_change_in_MEM_WB;
        imm_reg <= imm_in_MEM_WB;
        RegWrite_reg <= RegWrite_in_MEM_WB;
        ALU_result_reg <= ALU_result_in_MEM_WB;
        rd_reg <= rd_in_MEM_WB;
    end
end

assign PC1_out_MEM_WB = PC1_reg;
assign PC2_out_MEM_WB = PC2_reg;
assign MemtoReg_out_MEM_WB = MemtoReg_reg;
assign data_change_out_MEM_WB = data_change_reg;
assign imm_out_MEM_WB = imm_reg;
assign RegWrite_out_MEM_WB = RegWrite_reg;
assign ALU_result_out_MEM_WB = ALU_result_reg;
assign rd_out_MEM_WB = rd_reg;

endmodule
```

1.10 WB.v

该阶段执行写寄存器,写入的数据根据之前ID阶段生成的MemtoReg控制信号进行选择,由于寄存器是在下降沿进行写,所以WB阶段时钟周期下降沿就能完成写

```
`timescale 1ns/1ps
module WB(
    input [31:0]PC1_in_WB,
    input [31:0]PC2_in_WB,
    input [31:0]data_change_in_WB,
    input [31:0]imm_in_WB,
    input [31:0]ALU_result_in_WB,
    input [2:0]MemtoReg_WB,
    output [31:0]wdata_out_WB
);

Data_select x0(
    .PC1(PC1_in_WB),
    .PC2(PC2_in_WB),
    .data_change(data_change_in_WB),
    .immediate(imm_in_WB),
    .ALU_result(ALU_result_in_WB),
    .Memtoreg(MemtoReg_WB),
    .data_out(wdata_out_WB)
);

endmodule
```

1.11 forwarding.v

该部分用来解决数据冲突的情况,当写入寄存器尚未完成时就读该寄存器,就会发生数据冲突,由于ID阶段读寄存器,下一个EX阶段接收读出的值,所以我们要通过MEM和WB阶段来判断是否需要将数据前递给EX阶段, `inst_ID_EX[19:15]` 和 `inst_ID_EX[24:20]` 分别表示读取的寄存器1和2,当MEM或WB阶段的rd寄存器不为0且等于rs1或rs2,并且RegWrite为1,这时候就要将要写入的数据前递,又因为MEM阶段执行的指令是在WB阶段执行的指令之后,所以如果MEM和WB阶段的rd寄存器相同,那前递的数据应该选择MEM阶

段的数据,前递的数据按照WB阶段那样根据MemtoReg控制信号进行选择就行。forwardingA为1表示rs1值需要前递,forwardingB为1表示rs2值需要前递,ALU_C表示前递的rs1值,ALU_D表示前递的rs2值

```
`timescale 1ns/1ps
module forwarding(
    input [31:0]inst_ID_EX,
    input ALUsrc_ID_EX,
    input RegWrite_EX_MEM,
    input [2:0]Memtoreg_EX_MEM,
    input [4:0]rd_EX_MEM,
    input [31:0]PC1_EX_MEM,
    input [31:0]PC2_EX_MEM,
    input [31:0]ALU_EX_MEM,
    input [31:0]imm_EX_MEM,
    input [31:0]data_change_MEM,
    input RegWrite_MEM_WB,
    input [2:0]Memtoreg_MEM_WB,
    input [4:0]rd_MEM_WB,
    input [31:0]PC1_MEM_WB,
    input [31:0]PC2_MEM_WB,
    input [31:0]ALU_MEM_WB,
    input [31:0]imm_MEM_WB,
    input [31:0]data_change_MEM_WB,
    output forwardingA,
    output forwardingB,
    output [31:0]ALU_C,
    output [31:0]ALU_D
);

reg A,B;
reg [31:0]C,D;
wire [31:0]data1,data2;

assign forwardingA = A;
assign forwardingB = B;
assign ALU_C = C;
assign ALU_D = D;

Data_select x0(
    .data_change(data_change_MEM),
    .ALU_result(ALU_EX_MEM),
    .immediate(imm_EX_MEM),
    .PC1(PC1_EX_MEM),
    .PC2(PC2_EX_MEM),
    .Memtoreg(Memtoreg_EX_MEM),
    .data_out(data1)
);

Data_select x1(
    .data_change(data_change_MEM_WB),
    .ALU_result(ALU_MEM_WB),
    .immediate(imm_MEM_WB),
    .PC1(PC1_MEM_WB),
    .PC2(PC2_MEM_WB),
    .Memtoreg(Memtoreg_MEM_WB),
```

```
    .data_out(data2)
);


always @(*) begin
    if(RegWrite_EX_MEM && rd_EX_MEM != 0 && rd_EX_MEM == inst_ID_EX[19:15]) begin
        A = 1;
        C = data1;
    end
    else if(RegWrite_MEM_WB && rd_MEM_WB != 0 && rd_MEM_WB == inst_ID_EX[19:15])
begin
        A = 1;
        C = data2;
    end
    else begin
        A = 0;
        C = 32'b0;
    end
    if(RegWrite_EX_MEM && rd_EX_MEM != 0 && rd_EX_MEM == inst_ID_EX[24:20]) begin
        B = 1;
        D = data1;
    end
    else if(RegWrite_MEM_WB && rd_MEM_WB != 0 && rd_MEM_WB == inst_ID_EX[24:20])
begin
        B = 1;
        D = data2;
    end
    else begin
        B = 0;
        D = 32'b0;
    end
end

endmodule
```

1.12 SCPU.v

该部分将上面这些部分连接起来

```
`timescale 1ns/1ps
module SCPU(
    input clk,
    input rst,
    input [31:0]Data_in,
    input [31:0]inst_in,
    output [31:0]Addr_out,
    output [31:0]Data_out,
    output [31:0]PC_out,
    output [31:0]pc_if,
    output [31:0]pc_id,
    output [31:0]inst_id,
    output [31:0]pc_ex,
    output [4:0]rd_ex,
    output [4:0]rs1,
    output [4:0]rs2,
```

```
    output [31:0]rs1_val,
    output [31:0]rs2_val,
    output reg_wen_ex,
    output is_imm,
    output [31:0]imm,
    output is_branch,
    output is_jal_ex,
    output is_jalr_ex,
    output is_auipc,
    output is_lui,
    output [3:0]alu_ctrl,
    output [4:0]rd_mem,
    output [31:0]alu_res,
    output [4:0]rd_wb,
    output [31:0]reg_w_data,
    output [3:0]MemWrite,
    output [31:0]reg0,
    output [31:0]reg1,
    output [31:0]reg2,
    output [31:0]reg3,
    output [31:0]reg4,
    output [31:0]reg5,
    output [31:0]reg6,
    output [31:0]reg7,
    output [31:0]reg8,
    output [31:0]reg9,
    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
);
//forwarding
wire forwardingA;
wire forwardingB;
wire [31:0]ALU_C;
wire [31:0]ALU_D;
//IF
wire [31:0]PC_out_IF;
```

```
//IF_ID
wire [31:0]PC_out_IF_ID;
wire [31:0]inst_out_IF_ID;
//ID
wire [31:0]rs1_ID;
wire [31:0]rs2_ID;
wire [31:0]imm_ID;
wire [1:0]jump_ID;
wire [1:0]branch_ID;
wire [2:0]Memtoreg_ID;
wire [3:0]ALUop_ID;
wire ALUsrc_ID;
wire RegWrite_ID;
wire [1:0]store_type_ID;
wire [2:0]load_type_ID;
//ID_EX
wire [31:0]PC_out_ID_EX;
wire [31:0]inst_out_ID_EX;
wire [1:0]jump_out_ID_EX;
wire [1:0]branch_out_ID_EX;
wire [2:0]MemtoReg_out_ID_EX;
wire [3:0]ALUOp_out_ID_EX;
wire [1:0]store_type_out_ID_EX;
wire [2:0]load_type_out_ID_EX;
wire ALUsrc_out_ID_EX;
wire RegWrite_out_ID_EX;
wire [31:0]rs1data_out_ID_EX;
wire [31:0]rs2data_out_ID_EX;
wire [31:0]imm_out_ID_EX;
wire [4:0]rd_out_ID_EX;
//EX
wire zero_out_EX;
wire [31:0]ALU_out_EX;
//EX_MEM
wire [31:0]PC1_out_EX_MEM;
wire [31:0]PC2_out_EX_MEM;
wire [2:0]MemtoReg_out_EX_MEM;
wire [1:0]store_type_out_EX_MEM;
wire RegWrite_out_EX_MEM;
wire [2:0]load_type_out_EX_MEM;
wire [31:0]rs2data_out_EX_MEM;
wire [31:0]imm_out_EX_MEM;
wire [31:0]ALU_out_EX_MEM;
wire [4:0]rd_out_EX_MEM;
//MEM
wire [31:0]Addr_out_MEM;
wire [3:0]MemWrite_out_MEM;
wire [31:0]data_change_out_MEM;
wire [31:0]data_out_MEM;
//MEM_WB
wire [31:0]PC1_out_MEM_WB;
wire [31:0]PC2_out_MEM_WB;
wire [2:0]MemtoReg_out_MEM_WB;
wire [31:0]data_change_out_MEM_WB;
wire [31:0]imm_out_MEM_WB;
wire RegWrite_out_MEM_WB;
```

```
wire [31:0]ALU_result_out_MEM_WB;
wire [4:0]rd_out_MEM_WB;
//WB
wire [31:0]wdata_out_WB;

wire stall;
wire stall_IF;
wire stall_IF_ID;



assign stall=(jump_ID==2'b10)||(jump_ID==2'b01)||(branch_ID[1]==1'b1)||
(jump_out_ID_EX==2'b10)||(jump_out_ID_EX==2'b01)||(branch_out_ID_EX[1]==1'b1);
assign stall_IF=stall;
assign stall_IF_ID=stall;


assign Addr_out=Addr_out_MEM;
assign Data_out=data_out_MEM;
assign PC_out=PC_out_IF;

assign pc_if=PC_out_IF;
assign pc_id=PC_out_IF_ID;
assign inst_id=inst_out_IF_ID;
assign pc_ex=PC_out_ID_EX;
assign rd_ex=rd_out_ID_EX;
assign rs1=inst_out_IF_ID[19:15];
assign rs2=inst_out_IF_ID[24:20];
assign rs1_val=rs1_ID;
assign rs2_val=rs2_ID;
assign reg_wen_ex=RegWrite_out_ID_EX;
assign is_imm=ALUsrc_out_ID_EX;
assign imm=imm_out_ID_EX;
assign is_branch=branch_out_ID_EX[1];
assign is_jal_ex=(jump_out_ID_EX==2'b01);
assign is_jalr_ex=(jump_out_ID_EX==2'b10);
assign alu_ctrl=ALUOp_out_ID_EX;
assign rd_mem=rd_out_EX_MEM;
assign alu_res=ALU_out_EX_MEM;
assign rd_wb=rd_out_MEM_WB;
assign reg_w_data=wdata_out_WB;
assign MemWrite=MemWrite_out_MEM;


forwarding a0(
    .inst_ID_EX(inst_out_ID_EX),
    .ALUsrc_ID_EX(ALUsrc_out_ID_EX),
    .RegWrite_EX_MEM(RegWrite_out_EX_MEM),
    .Memtoreg_EX_MEM(MemtoReg_out_EX_MEM),
    .rd_EX_MEM(rd_out_EX_MEM),
    .PC1_EX_MEM(PC1_out_EX_MEM),
    .PC2_EX_MEM(PC2_out_EX_MEM),
    .ALU_EX_MEM(ALU_out_EX_MEM),
    .imm_EX_MEM(imm_out_EX_MEM),
    .data_change_MEM(data_change_out_MEM),
    .RegWrite_MEM_WB(RegWrite_out_MEM_WB),
```

```
    .Memtoreg_MEM_WB(MemtoReg_out_MEM_WB),
    .rd_MEM_WB(rd_out_MEM_WB),
    .PC1_MEM_WB(PC1_out_MEM_WB),
    .PC2_MEM_WB(PC2_out_MEM_WB),
    .ALU_MEM_WB(ALU_result_out_MEM_WB),
    .imm_MEM_WB(imm_out_MEM_WB),
    .data_change_MEM_WB(data_change_out_MEM_WB),
    .forwardingA(forwardingA),
    .forwardingB(forwardingB),
    .ALU_C(ALU_C),
    .ALU_D(ALU_D)
);

IF x0(
    .clk_IF(clk),
    .rst_IF(rst),
    .stall_IF(stall_IF),
    .PC_ID_EX(PC_out_ID_EX),
    .PC_in_IF(PC_out_IF),
    .PC1_IF(PC_out_ID_EX+imm_out_ID_EX),
    .PC2_IF(ALU_out_EX),
    .jump_IF(jump_out_ID_EX),
    .branch_IF(branch_out_ID_EX),
    .zero_IF(zero_out_EX),
    .PC_out_IF(PC_out_IF)
);

IF_ID_reg x1(
    .clk_IF_ID(clk),
    .rst_IF_ID(rst),
    .stall_IF_ID(stall_IF_ID),
    .PC_in_IF_ID(PC_out_IF),
    .inst_in_IF_ID(inst_in),
    .PC_out_IF_ID(PC_out_IF_ID),
    .inst_out_IF_ID(inst_out_IF_ID)
);

ID x2(
    .clk_ID(clk),
    .rst_ID(rst),
    .inst_in_ID(inst_out_IF_ID),
    .waddr_ID(rd_out_MEM_WB),
    .wdata_ID(wdata_out_WB),
    .Regwrite_ID(RegWrite_out_MEM_WB),
    .rs1_ID(rs1_ID),
    .rs2_ID(rs2_ID),
    .imm_ID(imm_ID),
    .jump_ID(jump_ID),
    .branch_ID(branch_ID),
    .Memtoreg_ID(Memtoreg_ID),
    .ALUop_ID(ALUop_ID),
    .ALUsrc_ID(ALUsrc_ID),
    .RegWrite_ID(RegWrite_ID),
    .store_type_ID(store_type_ID),
    .load_type_ID(load_type_ID),
    .is_auipc(is_auipc),
```

```
    .is_lui(is_lui),
    .reg0(reg0),
    .reg1(reg1),
    .reg2(reg2),
    .reg3(reg3),
    .reg4(reg4),
    .reg5(reg5),
    .reg6(reg6),
    .reg7(reg7),
    .reg8(reg8),
    .reg9(reg9),
    .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)
);

ID_EX_reg x3(
    .clk_ID_EX(clk),
    .rst_ID_EX(rst),
    .PC_in_ID_EX(PC_out_IF_ID),
    .inst_in_ID_EX(inst_out_IF_ID),
    .jump_in_ID_EX(jump_ID),
    .branch_in_ID_EX(branch_ID),
    .MemtoReg_in_ID_EX(Memtoreg_ID),
    .ALUOp_in_ID_EX(ALUop_ID),
    .store_type_in_ID_EX(store_type_ID),
    .load_type_in_ID_EX(load_type_ID),
    .ALUsrc_in_ID_EX(ALUsrc_ID),
    .RegWrite_in_ID_EX(RegWrite_ID),
    .rs1data_in_ID_EX(rs1_ID),
    .rs2data_in_ID_EX(rs2_ID),
    .imm_in_ID_EX(imm_ID),
    .rd_in_ID_EX(inst_out_IF_ID[11:7]),
    .PC_out_ID_EX(PC_out_ID_EX),
    .inst_out_ID_EX(inst_out_ID_EX),
    .jump_out_ID_EX(jump_out_ID_EX),
    .branch_out_ID_EX(branch_out_ID_EX),
```

```
    .MemtoReg_out_ID_EX(MemtoReg_out_ID_EX),
    .ALUOp_out_ID_EX(ALUOp_out_ID_EX),
    .store_type_out_ID_EX(store_type_out_ID_EX),
    .load_type_out_ID_EX(load_type_out_ID_EX),
    .ALUsrc_out_ID_EX(ALUsrc_out_ID_EX),
    .RegWrite_out_ID_EX(RegWrite_out_ID_EX),
    .rs1data_out_ID_EX(rs1data_out_ID_EX),
    .rs2data_out_ID_EX(rs2data_out_ID_EX),
    .imm_out_ID_EX(imm_out_ID_EX),
    .rd_out_ID_EX(rd_out_ID_EX)
);

EX x4(
    .ALUOp_in_EX(ALUOp_out_ID_EX),
    .ALUsrc_in_EX(ALUsrc_out_ID_EX),
    .forwardingA(forwardingA),
    .forwardingB(forwardingB),
    .ALU_C(ALU_C),
    .ALU_D(ALU_D),
    .rs1data_in_EX(rs1data_out_ID_EX),
    .rs2data_in_EX(rs2data_out_ID_EX),
    .imm_in_EX(imm_out_ID_EX),
    .zero_out_EX(zero_out_EX),
    .ALU_out_EX(ALU_out_EX)
);

EX_MEM_reg x5(
    .clk_EX_MEM(clk),
    .rst_EX_MEM(rst),
    .PC1_in_EX_MEM(PC_out_ID_EX+4),
    .PC2_in_EX_MEM(PC_out_ID_EX+imm_out_ID_EX),
    .MemtoReg_in_EX_MEM(MemtoReg_out_ID_EX),
    .store_type_in_EX_MEM(store_type_out_ID_EX),
    .RegWrite_in_EX_MEM(RegWrite_out_ID_EX),
    .load_type_in_EX_MEM(load_type_out_ID_EX),
    .rs2data_in_EX_MEM(forwardingB==0?rs2data_out_ID_EX:ALU_D),
    .imm_in_EX_MEM(imm_out_ID_EX),
    .ALU_in_EX_MEM(ALU_out_EX),
    .rd_in_EX_MEM(rd_out_ID_EX),
    .PC1_out_EX_MEM(PC1_out_EX_MEM),
    .PC2_out_EX_MEM(PC2_out_EX_MEM),
    .MemtoReg_out_EX_MEM(MemtoReg_out_EX_MEM),
    .store_type_out_EX_MEM(store_type_out_EX_MEM),
    .RegWrite_out_EX_MEM(RegWrite_out_EX_MEM),
    .load_type_out_EX_MEM(load_type_out_EX_MEM),
    .rs2data_out_EX_MEM(rs2data_out_EX_MEM),
    .imm_out_EX_MEM(imm_out_EX_MEM),
    .ALU_out_EX_MEM(ALU_out_EX_MEM),
    .rd_out_EX_MEM(rd_out_EX_MEM)
);

MEM x6(
    .load_type_MEM(load_type_out_EX_MEM),
    .store_type_MEM(store_type_out_EX_MEM),
    .ALU_result_MEM(ALU_out_EX_MEM),
    .data_in_MEM(Data_in),
```

```
    .rs2data_in_MEM(rs2data_out_EX_MEM),
    .Addr_out_MEM(Addr_out_MEM),
    .MemWrite_out_MEM(MemWrite_out_MEM),
    .data_change_out_MEM(data_change_out_MEM),
    .data_out_MEM(data_out_MEM)
);

MEM_WB_reg x7(
    .clk_MEM_WB(clk),
    .rst_MEM_WB(rst),
    .PC1_in_MEM_WB(PC1_out_EX_MEM),
    .PC2_in_MEM_WB(PC2_out_EX_MEM),
    .MemtoReg_in_MEM_WB(MemtoReg_out_EX_MEM),
    .data_change_in_MEM_WB(data_change_out_MEM),
    .imm_in_MEM_WB(imm_out_EX_MEM),
    .RegWrite_in_MEM_WB(RegWrite_out_EX_MEM),
    .ALU_result_in_MEM_WB(ALU_out_EX_MEM),
    .rd_in_MEM_WB(rd_out_EX_MEM),
    .PC1_out_MEM_WB(PC1_out_MEM_WB),
    .PC2_out_MEM_WB(PC2_out_MEM_WB),
    .MemtoReg_out_MEM_WB(MemtoReg_out_MEM_WB),
    .data_change_out_MEM_WB(data_change_out_MEM_WB),
    .imm_out_MEM_WB(imm_out_MEM_WB),
    .RegWrite_out_MEM_WB(RegWrite_out_MEM_WB),
    .ALU_result_out_MEM_WB(ALU_result_out_MEM_WB),
    .rd_out_MEM_WB(rd_out_MEM_WB)
);

WB x8(
    .PC1_in_WB(PC1_out_MEM_WB),
    .PC2_in_WB(PC2_out_MEM_WB),
    .data_change_in_WB(data_change_out_MEM_WB),
    .imm_in_WB(imm_out_MEM_WB),
    .ALU_result_in_WB(ALU_result_out_MEM_WB),
    .MemtoReg_WB(MemtoReg_out_MEM_WB),
    .wdata_out_WB(wdata_out_WB)
);

endmodule
```

二、实验结果与分析

仿真波形分析

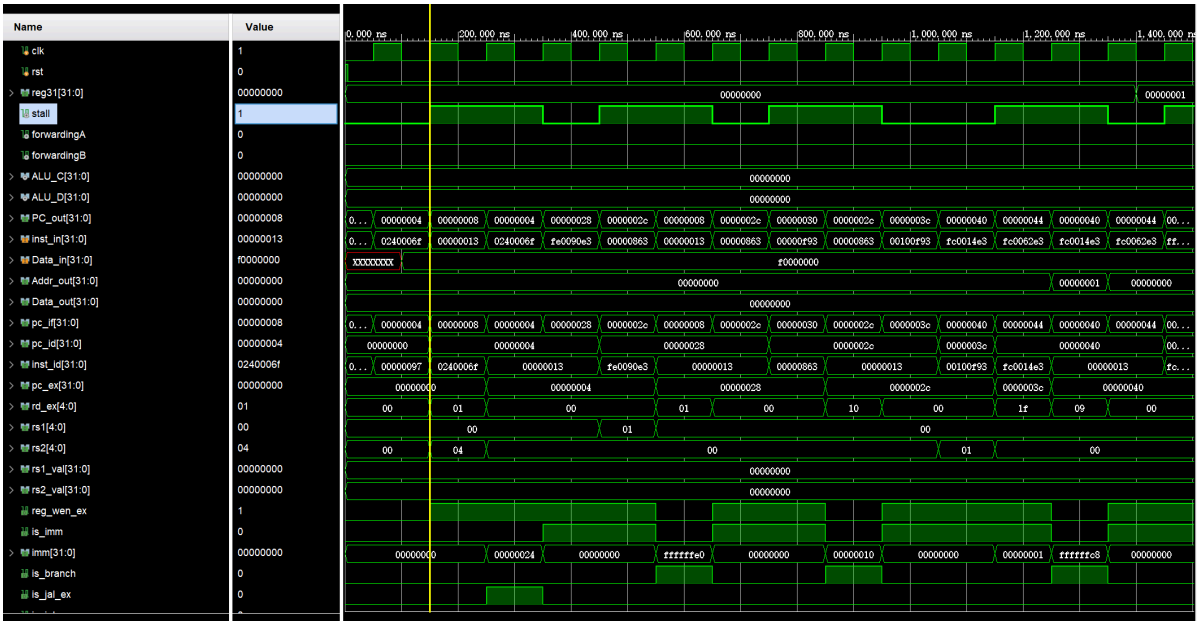
仿真代码用的就是lab4给的代码,并未修改

控制冒险

我是通过stall来处理控制冒险,而stall信号是由ID和EX阶段的jump和branch指令决定,需要在两个时钟周期后才能决定是否跳转,同时也可以发现下面的波形图中,在stall为1时会有两个时钟周期的inst_id为0x13,即nop指令

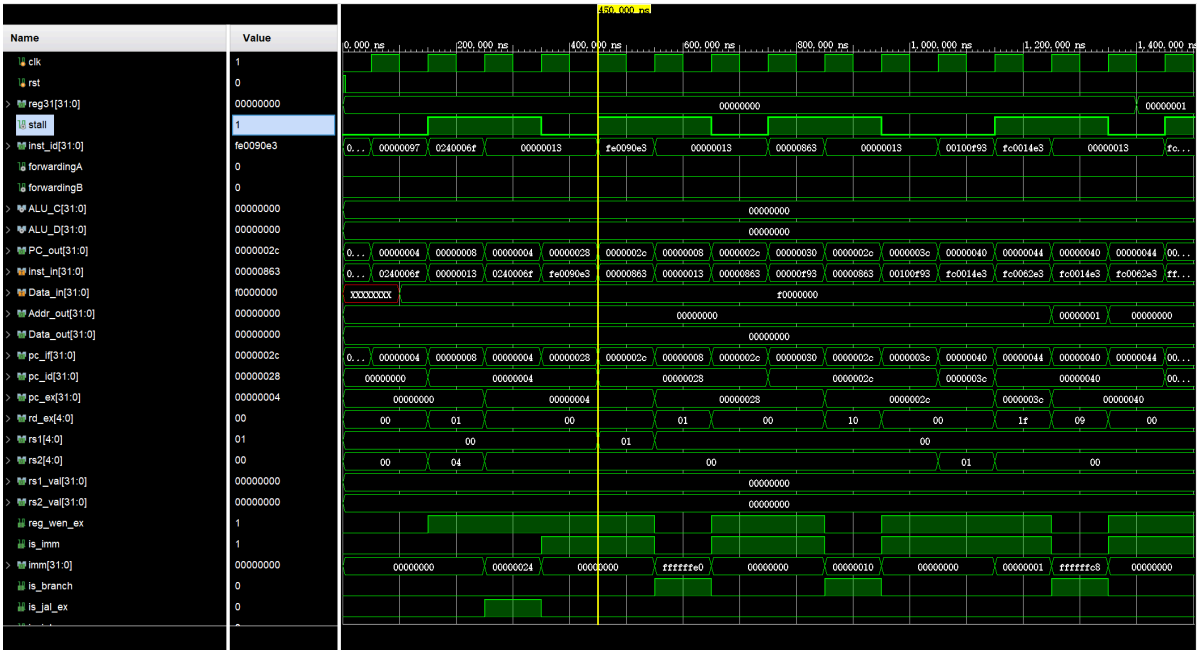
- **jal x0 36**

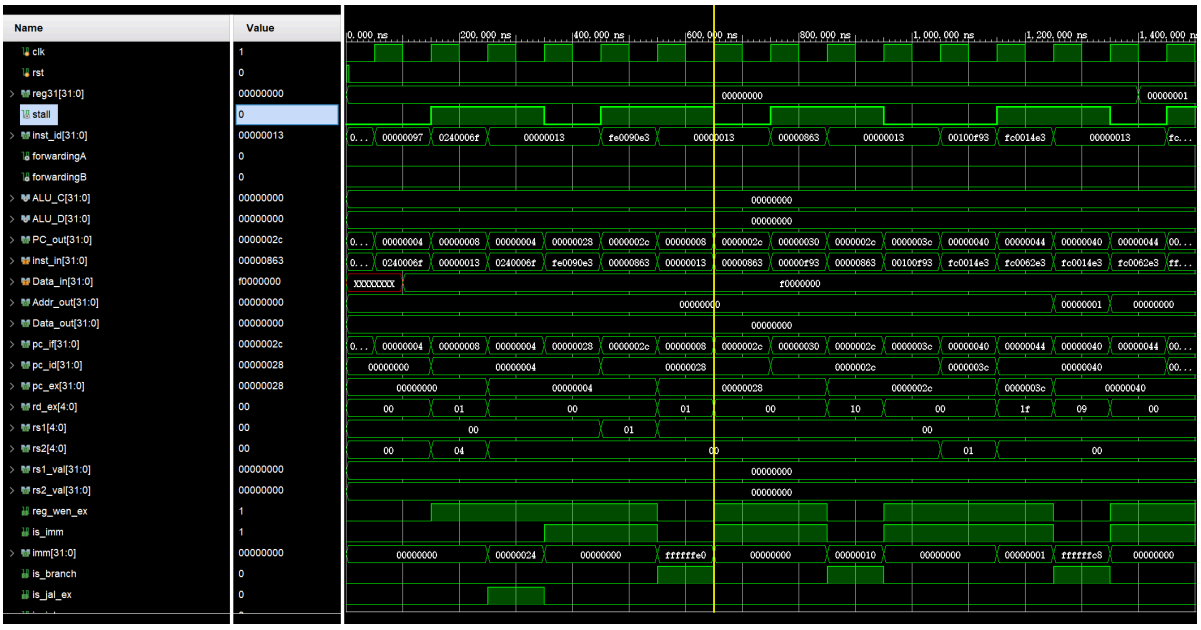
PC	Machine Code	Basic Code	Original Code
0x0	0x00000097	auipc x1 0	auipc x1, 0
0x4	0x0240006F	jal x0 36	j start # 00
0x8	0x00000013	addi x0 x0 0	nop # 04
0xc	0x00000013	addi x0 x0 0	nop # 08
0x10	0x00000013	addi x0 x0 0	nop # 0C
0x14	0x00000013	addi x0 x0 0	nop # 10
0x18	0x00000013	addi x0 x0 0	nop # 14



- **bne x1 x0 -32**

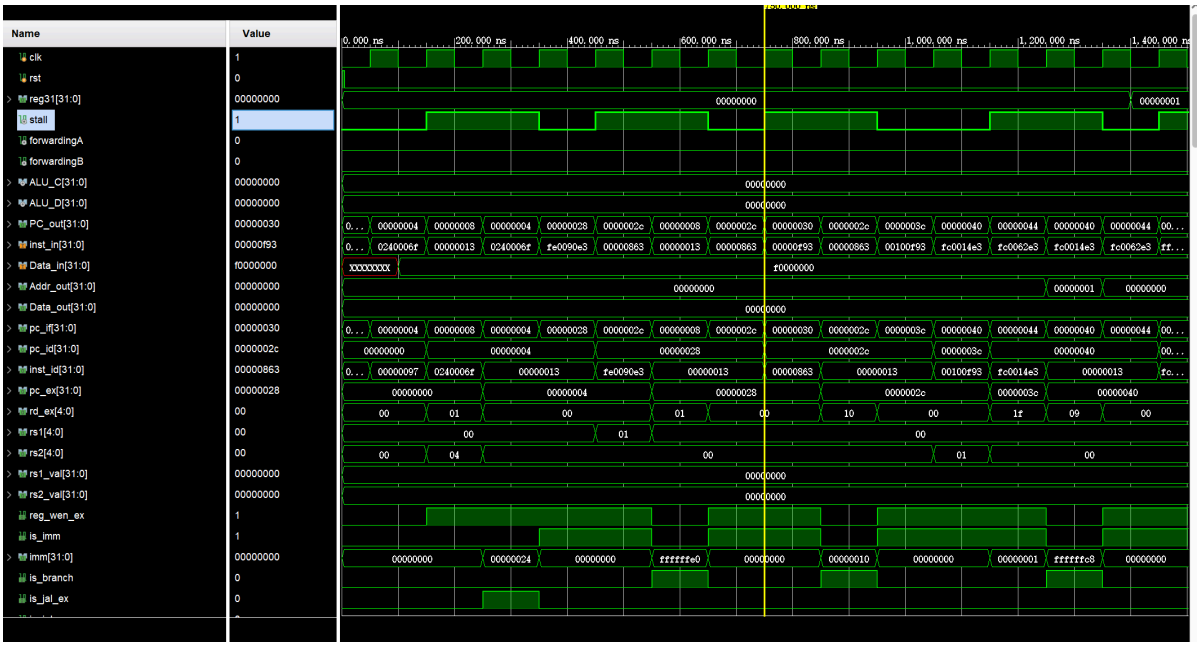
0x20	0x00000013	addi x0 x0 0	nop # 1C
0x24	0xFE5FF06F	jal x0 -28	j dummy
0x28	0xFE0090E3	bne x1 x0 -32	bnez x1, dummy
0x2c	0x00000863	beq x0 x0 16	beq x0, x0, pass_0
0x30	0x00000F93	addi x31 x0 0	li x31, 0

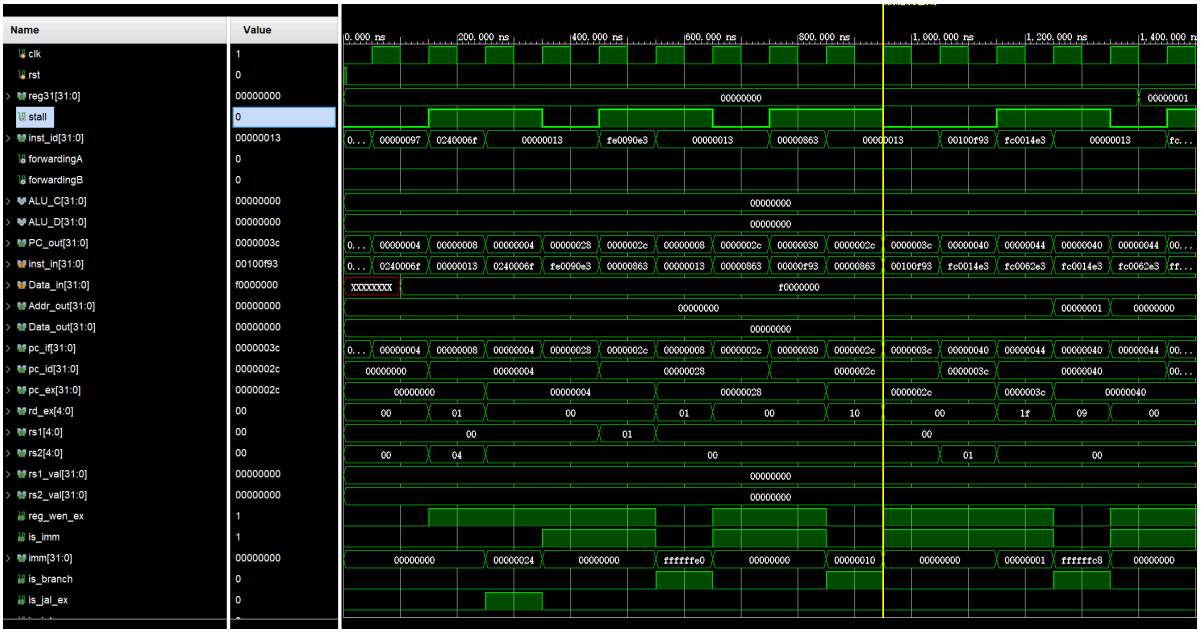




- **beq x0 x0 16**

	0x20	0x00000013	addi x0 x0 0	nop # 1C
	0x24	0xFE5FF06F	jal x0 -28	j dummy
	0x28	0xFE0090E3	bne x1 x0 -32	bnez x1, dummy
●	0x2c	0x00000863	beq x0 x0 16	beq x0, x0, pass_0
	0x30	0x00000F93	addi x31 x0 0	li x31, 0
	0x34	0x00000F17	auipc x30 0	auipc x30, 0
	0x38	0xFD1FF06F	jal x0 -48	j dummy

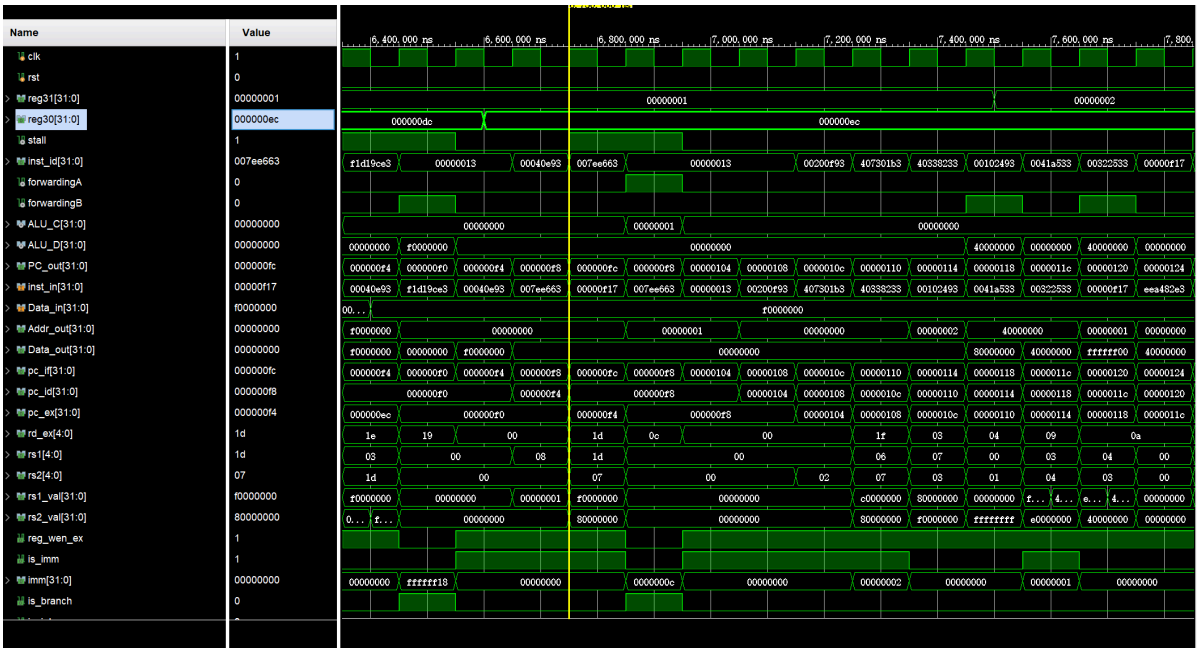




- **bltu x29 x7 12**

  可以发现执行该指令后成功跳转,并且x30的值不变说明没有执行下一条指令

	0xf0	0xF1D19CE3	bne x3 x29 -232	bne x3, x29, dummy
	0xf4	0x00040E93	addi x29 x8 0	mv x29, x8 # x29=x8=00000001
●	0xf8	0x007EE663	bltu x29 x7 12	bltu x29, x7, pass_1 # unsigned 00000001 < 80000000
	0xfc	0x00000F17	auipc x30 0	auipc x30, 0
	0x100	0xF09FF06F	jal x0 -248	j dummy
	0x104	0x00000013	addi x0 x0 0	nop
	0x108	0x00200F93	addi x31 x0 2	li x31, 2



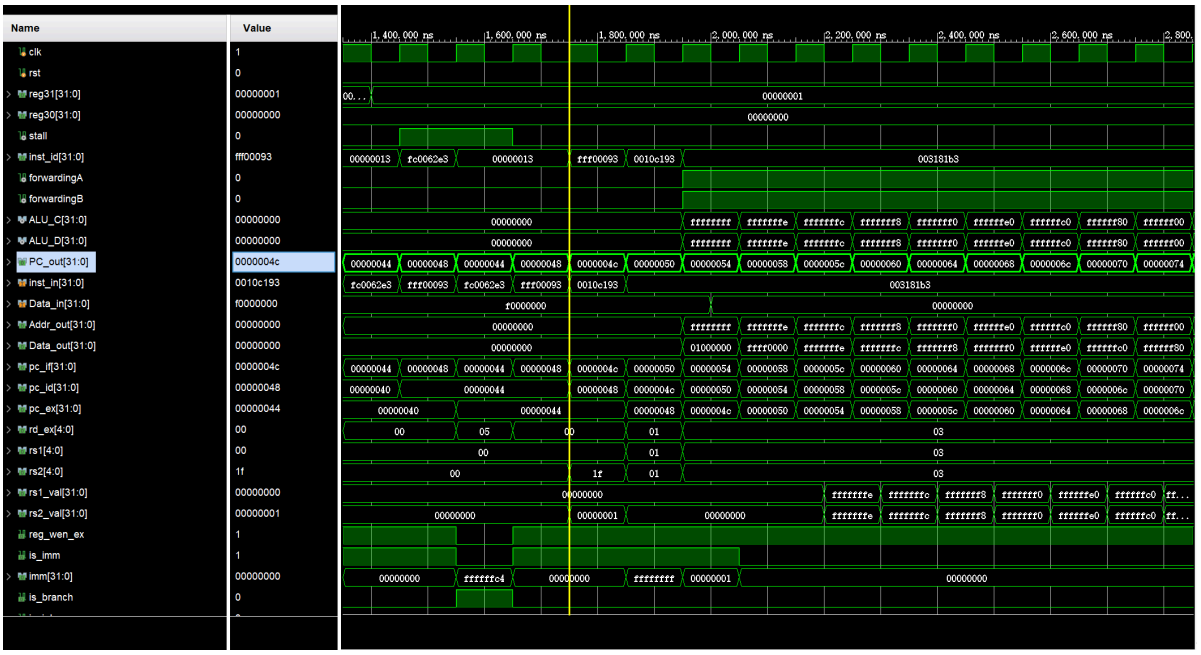
数据冒险

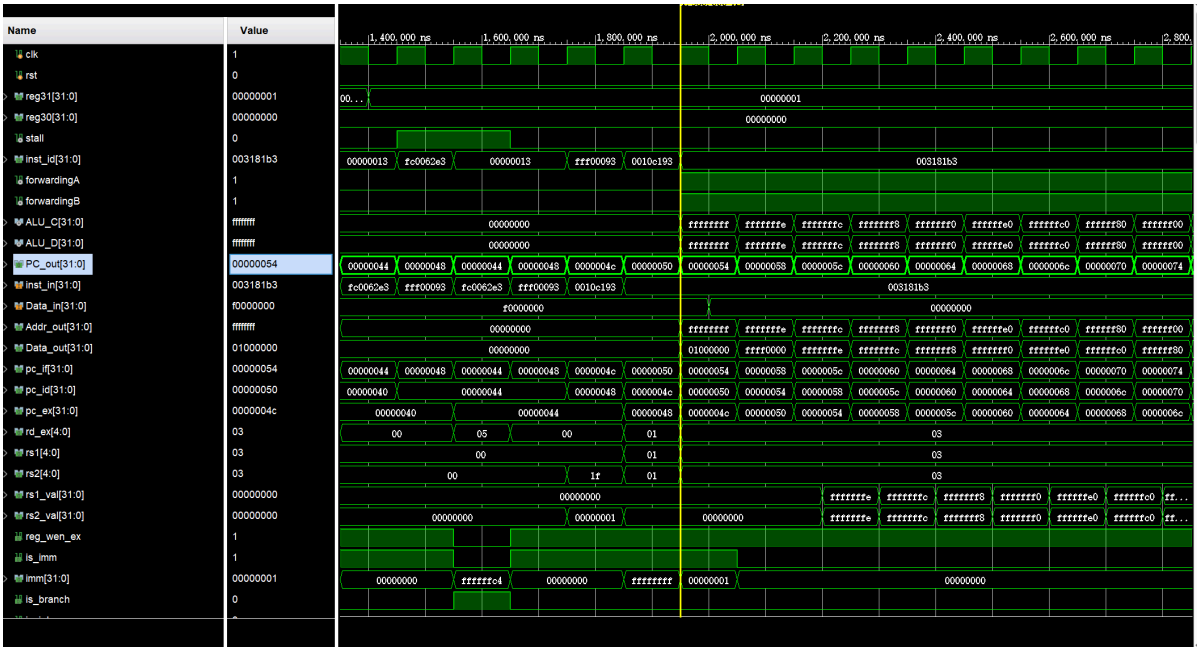
- **addi x1 x0 -1**

  xori x3 x1 1

当**xori x3 x1 1**执行到EX阶段时,**addi x1 x0 -1**执行到MEM阶段,由于**xori x3 x1 1**指令的立即数是1,而我们forwarding的判断依据是rd是否等于inst[19:15]或inst[24:20],所以此时我们的forwardingA和forwardingB都是1,ALU_C和ALU_D都等于x1,也就是-1,但我们之后对ALU计算的时候会根据ALU_scr去判断是立即数还是寄存器的值,所以此时forwardingB为1不会产生什么影响

	0x44	0xFC0062E3	bltu x0 x0 -60	bltu x0, x0, dummy
●	0x48	0xFFF00093	addi x1 x0 -1	li x1, -1 # x1=FFFFFFFF
	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

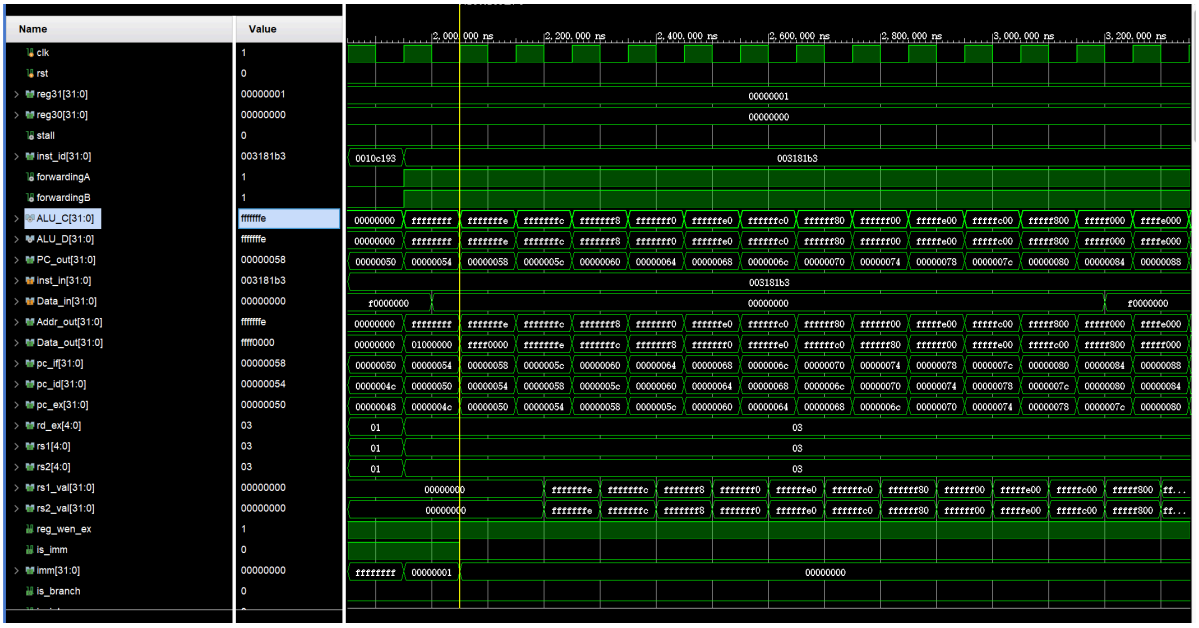




- **add x3 x3 x3**

  从0x50开始执行了多条**add x3 x3 x3**指令,可以看到仿真波形中forwardingA和forwardingB一直为1,并且ALU_C和ALU_D的值依次为0xFFFFFFFE,0xFFFFFFFC,0xFFFFFFF8,0xFFFFFFF0,0xFFFFFFE0,0xFFFFFFC0,0xFFFFFF80,0xFFFFFF00,0xFFFFFE00,正是x3的值

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
0x6c	0x003181B3	add x3 x3 x3	add x3, x3, x3 # x3=FFFFFE00
0x70	0x003181B3	add x3 x3 x3	add x3, x3, x3 # x3=FFFFFC00
0x74	0x003181B3	add x3 x3 x3	add x3, x3, x3 # x3=FFFFF800
0x78	0x003181B3	add x3 x3 x3	add x3, x3, x3 # x3=FFFFF000
0x7c	0x003181B3	add x3 x3 x3	add x3, x3, x3 # x3=FFFFE000

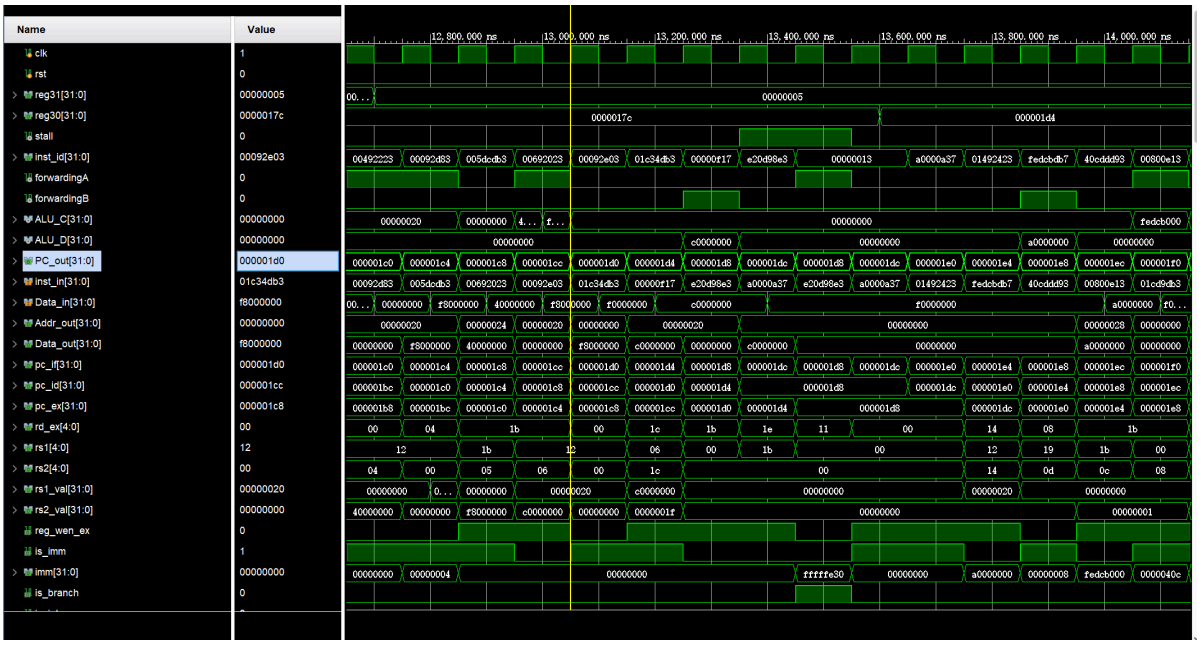


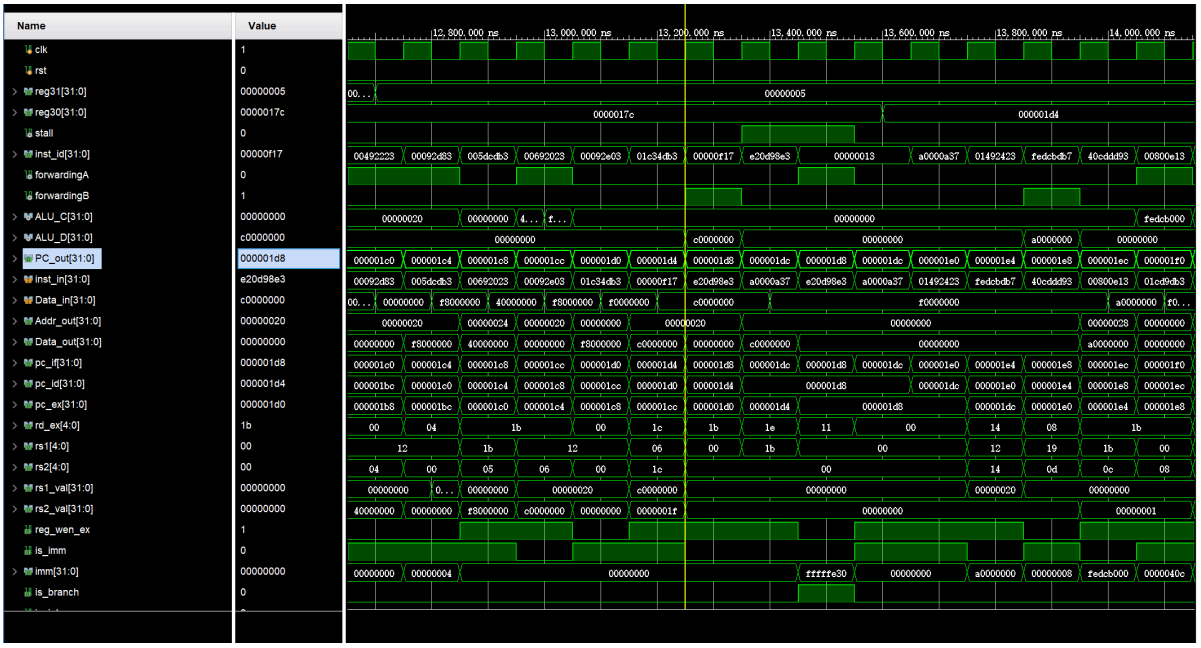
- **lw x28 0(x18)**

  xor x27 x6 x28

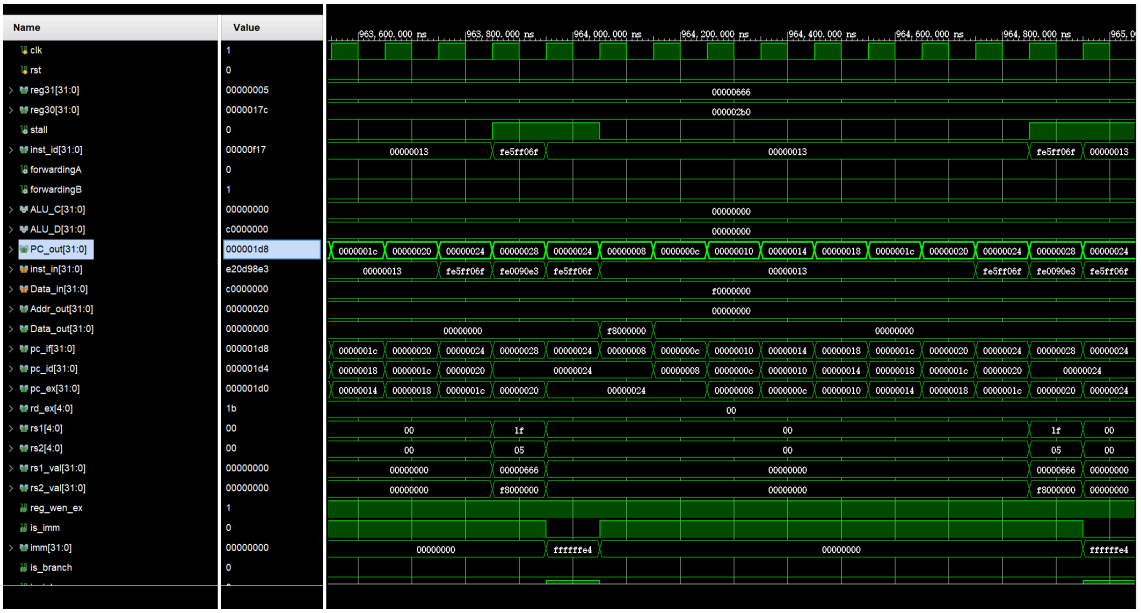
可以看到两个时钟周期后**lw x28 0(x18)**执行到MEM阶段,**xor x27 x6 x28**执行到EX阶段,此时forwardingB=1,ALU_D=0xC0000000,表示rs2数据需要前递,前递数据为0xC0000000,正是load指令执行后x28的值

0x1c8	0x00692023	sw x6 0(x18)	sw x6, 0(x18) # mem[0x20]=C0000000
0x1cc	0x00092E03	lw x28 0(x18)	lw x28, 0(x18) # 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
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
0x1f0	0x01CD9DB3	sll x27 x27 x28	sll x27, x27, x28 # x27=FFEDCB00
0x1f4	0x0FFDED93	ori x27 x27 255	ori x27, x27, 0xff # x27=FFEDCBFF



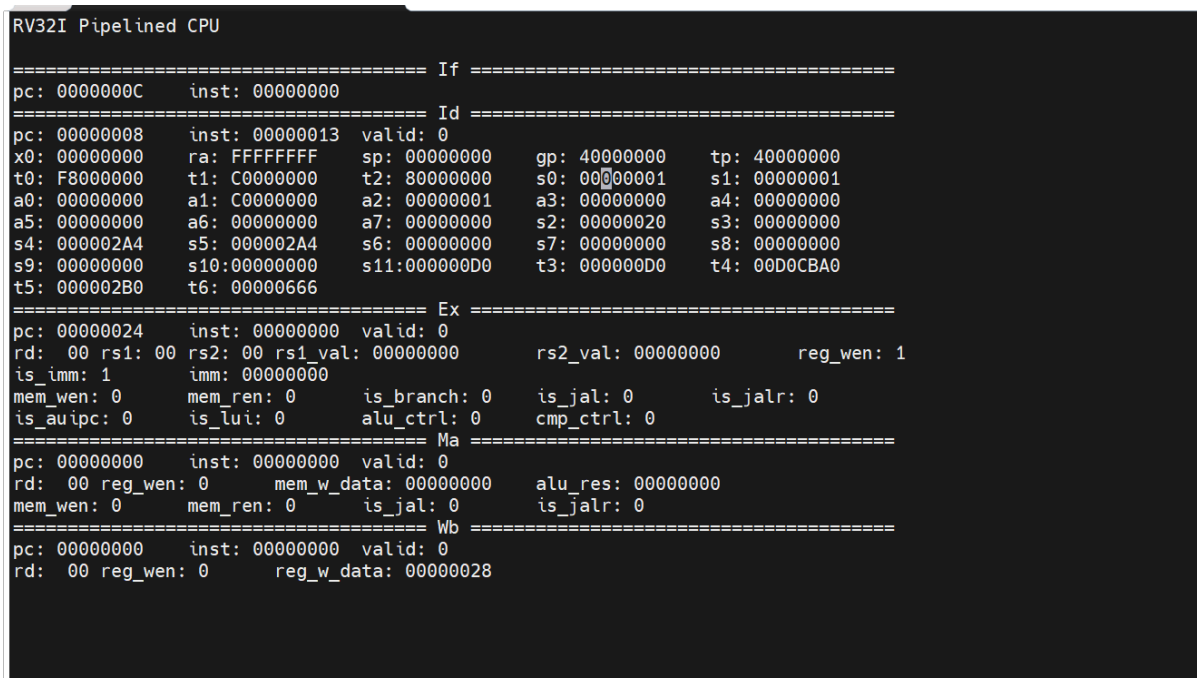


- 最终x31值为0x666



上板验证

由于数据前递的信号以及stall信号这些没法显示出来,所以最后就只放了x31=0x666的图片



三、讨论、心得

思考题

1.1

TP-0

```
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
```

因为写入寄存器是在WB阶段发生,所以在一条指令执行五个周期后就能将数据写入寄存器,这里可能发生冲突的指令之间相隔了五条指令,寄存器的值已经写入,所以不会发生冲突,$CPI = 16/12 = 1.33$

1.2

TP-1

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

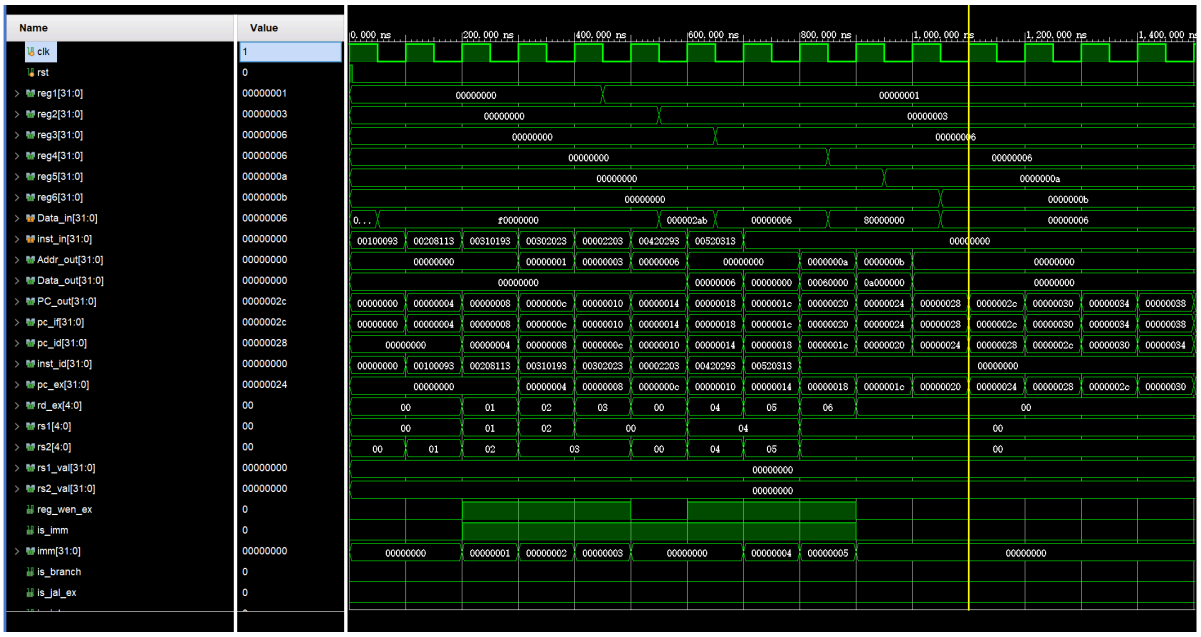
RISC-V 流水线数据冲突归纳后其实只有一种,即在尚未写入寄存器时读取其值（RAW）,1和2,1和3,3和4会发生数据冲突,因为我的流水线是用数据前递解决数据冲突,所以不会有stall,$CPI = 8/4 = 2$

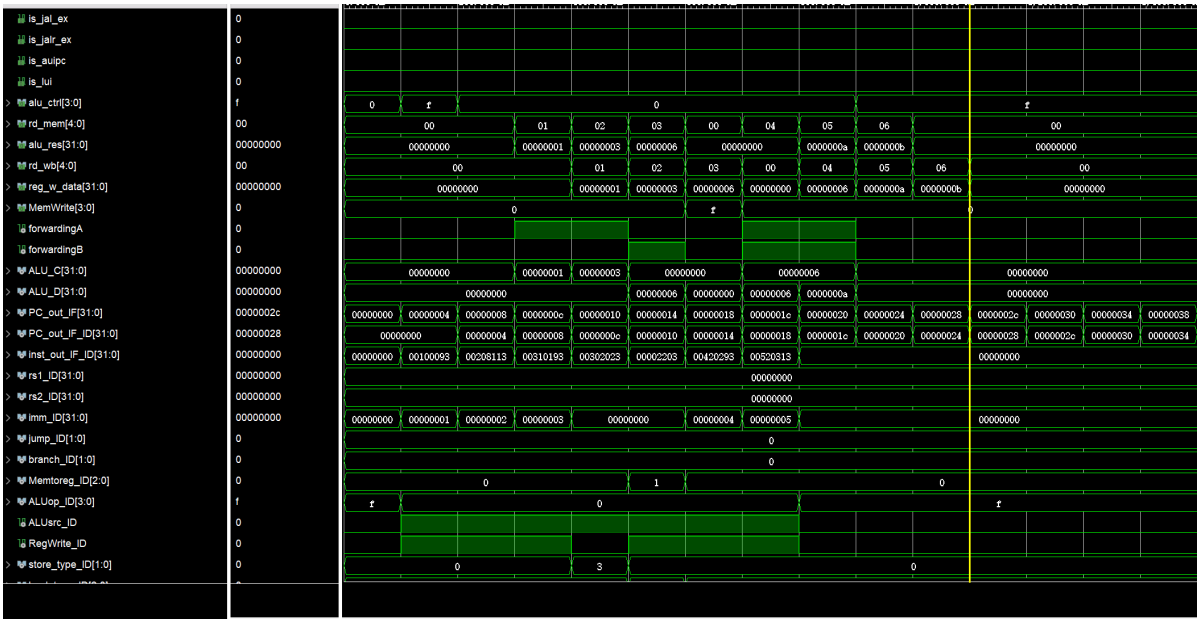
2

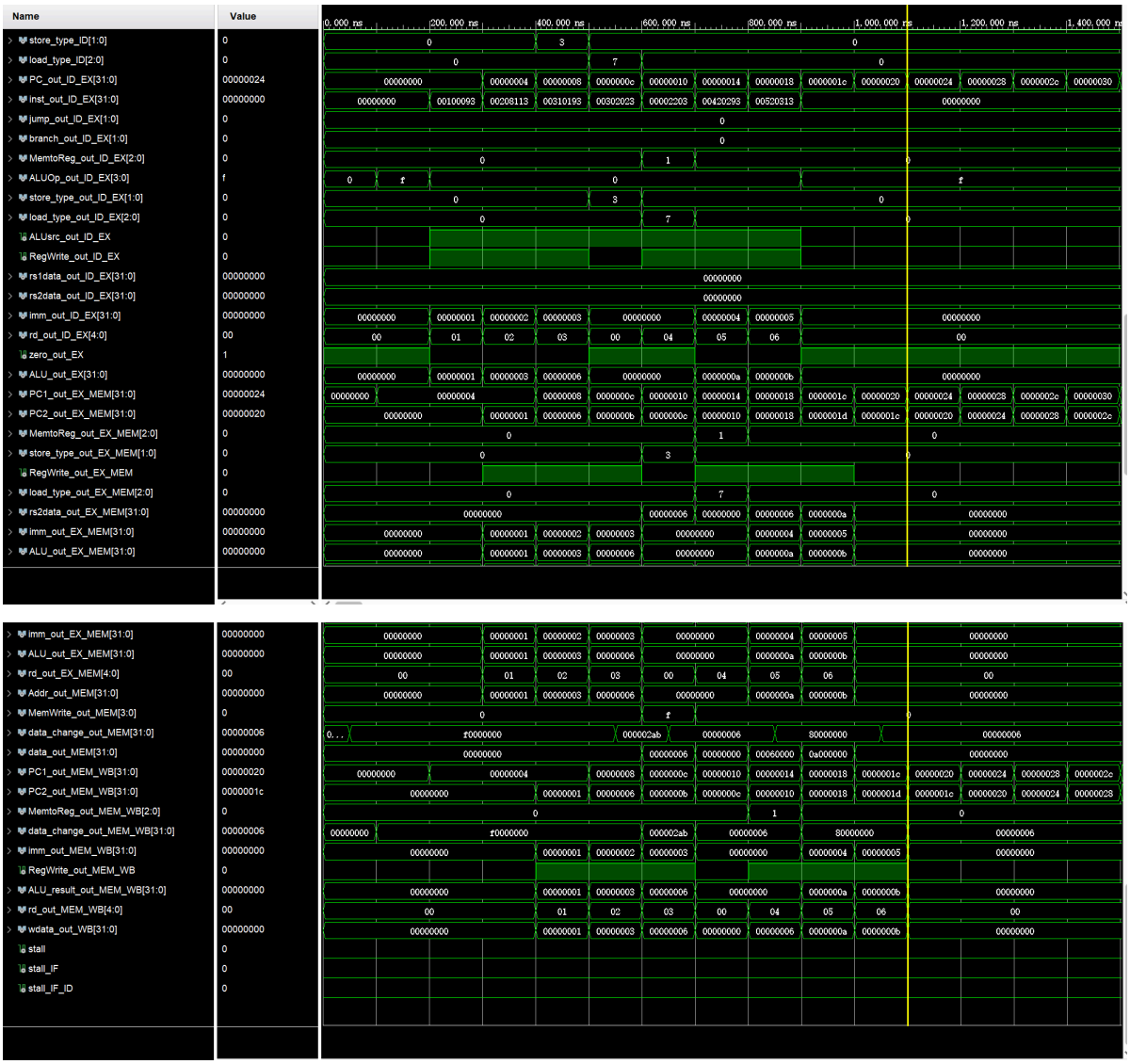
完成所有指令用了11个时钟周期

TP-3

```
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
```







心得

在上次完成单周期cpu后,这次写流水线cpu就觉得相对轻松许多,将数据通路画出来后就显得很清晰,麻烦的是连线部分,总的来说,此次实验还是顺利完成啦,完成后感觉如释重负,这学期的收获还是很多的,希望以后也能如此吧。