实验五：流水线的综合设计（含中断）

课程名称：　计算机组成　　实验类型：　综合

实验项目名称：　流水线综合设计实验

学生姓名:　胡若凡　　学号：　3200102312　　同组学生姓名：　无

实验地点：　紫金港东四 509 室　　实验日期：　2022　年　6　月　1　日

一、操作方法与实验步骤

1.1 实验目的

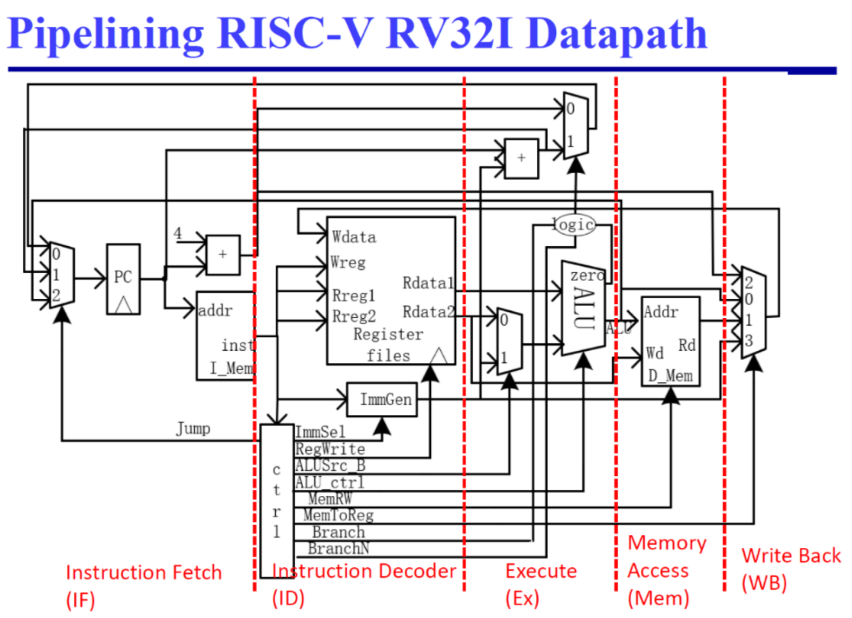
（1）理解流水线 CPU 的基本原理和组织结构

（2）掌握五级流水线的工作过程和设计方法

（3）理解流水线 CPU 停机的原理

（4）设计流水线测试程序

1.2 流水线设计框架



取指：取指阶段涉及程序计数器 PC 和指令存储器 I_Mem 程序计数器输出作为地址从指令存储器中读取指令。实验中，对应的是 IF_reg_ID ：暂存指令和 PC 值，以待下一级使用。

译码：译码阶段涉及寄存器堆 Register Files 和译码器、立即数生成单元(ImmGen)；从寄

存器堆可以读取操作数译码器对指令进行解析产生各种各种控制信号立即数生成单元根据控制信号和输入指令生成各种类型的立即数。对应的是 ID_reg_Ex 寄存器：暂存 PC 值，寄存器读取的数据，立即数和控制信号以待下一级使用。

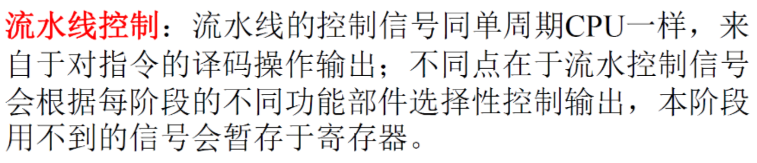
执行：执行阶段涉及运算单元 ALU 它获取操作数并完成指定的算数运算或逻辑运算。具体对应的是 Ex_reg_Mem 寄存器：暂存运算结果和控制信号以待下一级使用。

存储器访问:存储器访问阶段涉及数据存储器 D_Mem; Load/Store 指令对数据存储器进行读或写。具体对应的是 Mem_reg_WB 寄存器：暂存存储器结果和控制信号以待下一级使用。

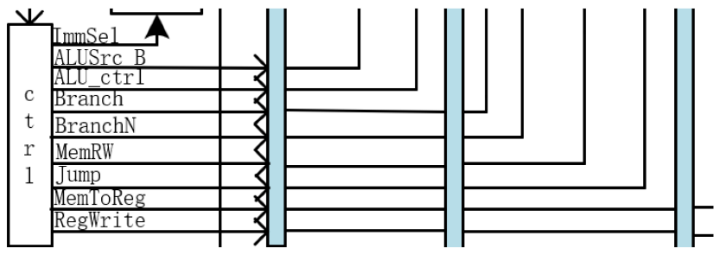
写回：写回阶段涉及寄存器堆（Register Files）；将 ALU 的运算结果、存储器输出结果、PC+4 写回到寄存器堆。当写回阶段结束，一次完整的五级流水操作完成；此时下一次操作进行到存储器访问阶段(如果有)。由于在各级流水线之间插入了寄存器作为数据及控制信号的暂存，从而实现多条指令的重叠而不受影响。

1.3 流水线控制信号设计

流水线所需要的控制信号如下所示：



流水线控制：流水线的控制信号同单周期CPU一样，来自于对指令的译码操作输出；不同点在于流水控制信号会根据每阶段的不同功能部件选择性控制输出，本阶段用不到的信号会暂存于寄存器。

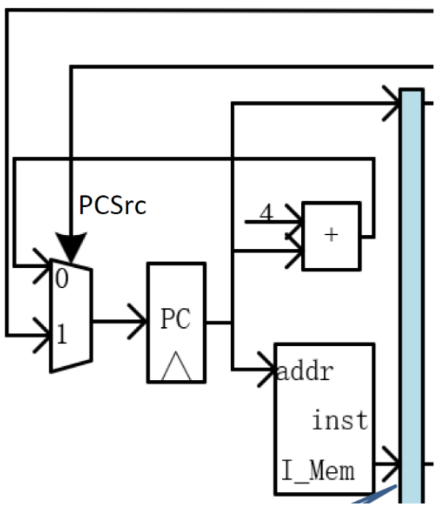


控制信号具体作用如下所示：

信号	源数目	功能定义	赋值0时动作	赋值1时动作	赋值2时动作
ALUSrc_B	2	ALU端口B输入选择	选择源操作数寄存器2数据	选择32位立即数（符号扩展后）	-
MemToReg	3	寄存器写入数据选择	选择ALU输出	选择存储器数据	选择PC+4
Branch		Beq指令目标地址选择	选择PC+4地址	选择转移目的地址PC+imm（zero=1）	-
BranchN		Bne指令目标地址选择	选择PC+4地址	选择转移目的地址PC+imm（zero=0）	-
Jump		Jal指令目标地址选择	选择PC+4地址	选择跳转目的地址	-
PCSrc	2	PC输入选择（分支跳转的衍生）	=(Branch&zero)|(BranchN&(~zero))|Jump		-
RegWrite	-	寄存器写控制	禁止寄存器写	使能寄存器写	-
MemRW	-	存储器读写控制	存储器读使能，存储器写禁止	存储器写使能，存储器读禁止	-
ALU_Control	**000-111**	3位ALU操作控制	参考表ALU_Control（详见实验04）		
ImmSel	**00-11**	2位立即数组合控制	参考表ImmSel（详见实验04）		

1.4 框架设计代码（以下将包含三个实验的模块设计）

1.4.1 IF 微操作部分



Pipeline-IF 设计：

```
    module Pipeline_IF(
    input [31:0]inst_in_IF,
    output reg[31:0]inst_out_IF,
    input clk_IF, //时钟
    input rst_IF, //复位
    input en_IF, //使能
    input [31:0] PC_in_jalr,
    input [31:0] PC4_in_IF,
    input [31:0] PC_in_IF, //取指令 PC 输入
    input [1:0]PCSrc, //PC 输入选择
    input NOP_IFID,
    output  reg[31:0] PC_out_IF //PC 输出
    );
    wire [31:0] new_pc;
    MUX4T1 MUX4T1_0(
    .s(PCSrc),
    .I0(PC4_in_IF),
    .I1(PC_in_IF),
    .I2(PC_in_jalr),
    .I3(0),//PCSrc 不会等于 4 的
    .o(new_pc));
    always@(*) begin
    if(rst_IF==1)begin
       PC_out_IF=0;
     inst_out_IF=0;
     end
     else if (en_IF)begin
     if(NOP_IFID)
     begin
    PC_out_IF=new_pc-4;
    inst_out_IF=inst_in_IF;
     end
     else begin
    PC_out_IF=new_pc;
       inst_out_IF=inst_in_IF;
    end
    end
    end
    endmodule
```

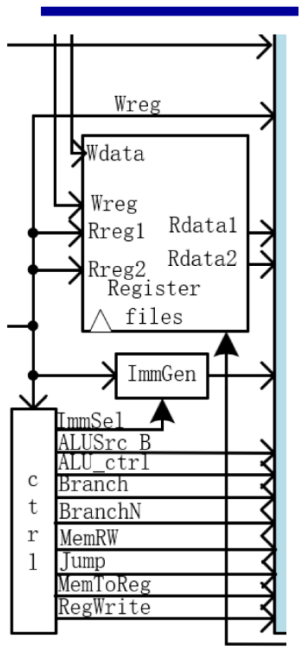
子模块：MUX4T1

```
module MUX4T1(
input[1:0] s,
input [31:0]I0,
input [31:0]I1,
input [31:0]I2,
input [31:0]I3,
output reg[31:0] o
    );
    always@(*)
    begin
    if(s==2'b00)
    o=I0;
    else if(s==2'b01)
    o=I1;
    else if(s==2'b10)
    o=I2;
    else
    o=I3;
    end
endmodule
```

IF_reg_ID 设计：

```
    module IF_Reg_ID(
    input clk_IFID, //寄存器时钟
    input rst_IFID, //寄存器复位
    input en_IFID, //寄存器使能
    input [31:0] PC_in_IFID, //PC 输入
    input [31:0] inst_in_IFID, //指令输入
    input NOP_IFID, //插入 NOP 使能
    output reg [31:0] PC_out_IFID, //PC 输出
    output reg [31:0] inst_out_IFID, //指令输出
    output reg valid_IFID//寄存器有效
    );
        always@(posedge clk_IFID or posedge rst_IFID)
        begin
        if(rst_IFID==1)
        begin
        PC_out_IFID=0;
        inst_out_IFID=0;
        valid_IFID=0;
        end
        else//rst=0
        begin
        if(NOP_IFID)//NOP_IFID=1 设置 valid_IFID=0
        begin
        if(en_IFID==1)
        begin
        inst_out_IFID=32'h00000013;
        valid_IFID=0;
        PC_out_IFID=PC_in_IFID;
        end
        else begin
        inst_out_IFID=32'h00000013;
        valid_IFID=0;
        end
        end
        else//NOP_IFID=0
        begin
        if(en_IFID)//NOP_IFID=0 and en_IFID=1
        begin
        valid_IFID=1;
        PC_out_IFID=PC_in_IFID;
        inst_out_IFID=inst_in_IFID;
        end
     //en_IFID==0 就保持不变
        end
        end
        end
    endmodule
```

1.4.2 ID 微操作部分



Pipeline_ID 设计

```
    module Pipeline_ID(
    `VGA_DBG_RegFile_Outputs
    input clk_ID, //时钟
    input rst_ID, //复位
    input RegWrite_in_ID, //寄存器堆使能
    input [4:0] Rd_addr_ID, //写目的地址输入
    input [31:0] Wt_data_ID, //写数据输入
    input [31:0] Inst_in_ID, //指令输入
    output [4:0] Rd_addr_out_ID, //写目的地址输出
    output wire [31:0] Rs1_out_ID, //操作数 1 输出
    output wire [31:0] Rs2_out_ID, //操作数 2 输出
    output wire [31:0] Imm_out_ID, //立即数输出
    output wire ALUSrc_B_ID, //ALU B 端输入选择
    output wire [31:0] ALU_control_ID,//ALU 控制
    output wire Branch_ID, //Branch 控制
    //output reg BranchN_ID, //Bne 控制 a
    output wire MemRW_ID, //存储器读写
    output wire [1:0] Jump_ID, //Jal,Jalr 控制
    output wire [2:0] MemtoReg_ID, //寄存器写回选择
    output wire RegWrite_out_ID) ;//寄存器堆读写

    wire [2:0]ImmSel;
```

```
assign Rd_addr_out_ID=Inst_in_ID[11:7];

RegFile Regs(
`VGA_DBG_RegFile_Arguments
.clk(~clk_ID),
.rst(rst_ID),
.wen(RegWrite_in_ID),
.rs1(Inst_in_ID[19:15]),
.rs2(Inst_in_ID[24:20]),
.rd(Rd_addr_ID),
.i_data(Wt_data_ID),
.rs1_val(Rs1_out_ID),
.rs2_val(Rs2_out_ID));

ImmGen ImmGen_0(
.ImmSel(ImmSel),
.inst_field(Inst_in_ID),
.Imm_out(Imm_out_ID));


SCPU_ctrl SCPU_ctrl_0(
.inst_field(Inst_in_ID),
.OPcode(Inst_in_ID[6:0]),
.Fun3(Inst_in_ID[14:12]),
.Fun7(Inst_in_ID[30]),
.ImmSel(ImmSel),
.ALUSrc_B(ALUSrc_B_ID),
.MemtoReg(MemtoReg_ID),
.Jump(Jump_ID),
.Branch(Branch_ID),
.RegWrite(RegWrite_out_ID),
.MemRW(MemRW_ID),
.ALU_Control(ALU_control_ID));

Endmodule
```

子模块：RegFile

```
module RegFile(
`VGA_DBG_RegFile_Outputs
input clk,
input rst,
input wen,
input[4:0]rs1,//源寄存器 1 的编号
input[4:0]rs2,
input[4:0]rd,//目的寄存器的编号
input[31:0]i_data,
output[31:0]rs1_val,//源寄存器 1 的输出数据
output[31:0]rs2_val//源寄存器 2 的输出数据
    );
    reg[31:0] regs[1:31];
    `VGA_DBG_RegFile_Assignments
    integer i;
    assign rs1_val=(rs1==0)?0:regs[rs1];
    assign rs2_val=(rs2==0)?0:regs[rs2];
    always@(posedge clk or posedge rst)
    begin
    if(rst==1)begin for(i=1;i<32;i=i+1) regs[i]<=0; end
    else
    begin if((wen==1)&&(rd!=0))
    regs[rd]<=i_data;
    end
    end
endmodule
```

ImmGen 模块

```
module ImmGen(
input wire [2:0] ImmSel,//立即数操作控制
input wire [31:0] inst_field,//指令数据域
output reg [31:0] Imm_out//立即数输出
    );
    always@(*)begin
    case(ImmSel)
    3'b000:Imm_out={{20{inst_field[31]}},inst_field[31:20]};//addi,lw

3'b001:Imm_out={{20{inst_field[31]}},inst_field[31:25],inst_field[11
:7]};//sw

3'b010:Imm_out={{20{inst_field[31]}},inst_field[7],inst_field[30:25]
,inst_field[11:8],1'b0};//beq

3'b011:Imm_out={{12{inst_field[31]}},inst_field[19:12],inst_field[20
],inst_field[30:21],1'b0};//jal
    3'b100:Imm_out={inst_field[31:12],12'b0};
    endcase
    end
endmodule
```

子模块：SCPU_ctrl

```
module SCPU_ctrl(
input[31:0]inst_field,
input[6:0]OPcode, //Opcode------inst[6:0]
input[2:0]Fun3, //Function-----inst[14:12]
input Fun7, //Function-----inst[30]
output reg [2:0]ImmSel, //立即数选择控制
output reg ALUSrc_B, //源操作数 2 选择
output reg [2:0] MemtoReg, //写回数据选择控制
output reg [1:0]Jump, //jal
output reg Branch, //beq
output reg RegWrite, //寄存器写使能
output reg MemRW, //存储器读写使能
output reg [31:0]ALU_Control
    );
    reg [1:0] ALUop;
always @* begin
case(OPcode)
7'b0110111: begin //lui
RegWrite=1;
ImmSel=3'b100;
ALUSrc_B=1;//无关项
Branch=0;
Jump=0;
MemtoReg=3'b011;
MemRW=0;//应该不写也不读此处写 0 也没啥
ALUop=2'b00;//无关项
end
7'b0010111: begin //auipc
RegWrite=1;
ImmSel=3'b100;
ALUSrc_B=1;
Branch=0;
Jump=0;//pc+4
MemtoReg=3'b100;
MemRW=0;
ALUop=2'b00;
end
7'b1100111: begin //jalr
RegWrite=1;
ImmSel=3'b000;
ALUSrc_B=1;
Branch=0;
Jump=2'b10;
MemtoReg=3'b010;
MemRW=0;
ALUop=2'b00;
end
7'b0110011: begin //ALU R-type ok
RegWrite=1;
ImmSel=3'b000;
ALUSrc_B=0;
Branch=0;
Jump=0;
MemtoReg=0;
MemRW=0;
ALUop=2'b10;
end
7'b0000011: begin//load I-type ok
RegWrite=1;
ImmSel=3'b000;
ALUSrc_B=1;
Branch=0;
```

```
Jump=0;
MemtoReg=3'b001;
MemRW=0;
ALUop=2'b00;
//ALU_Control=add
end
7'b0100011: begin//store S-type ok
RegWrite=0;
ImmSel=3'b001;
ALUSrc_B=1;
Branch=0;
Jump=0;
MemtoReg=0;
MemRW=1;
ALUop=2'b00;
end
7'b1100011: begin //beq B-type ok
RegWrite=0;
ImmSel=3'b010;
ALUSrc_B=0;
Branch=1;
Jump=0;
MemtoReg=0;
MemRW=0;
ALUop=2'b01;
end
7'b1101111: begin //jump  J-type
RegWrite=1;
ImmSel=3'b011;
ALUSrc_B=1;
Branch=0;
Jump=1;
MemtoReg=3'b010;
MemRW=0;
ALUop=2'b00;
end
7'b0010011:begin//ALU(addi;;;;) I-type
RegWrite=1;
ImmSel=3'b000;
ALUSrc_B=1;
Branch=0;
Jump=0;
MemtoReg=3'b000;
MemRW=0;
ALUop=2'b11;
//ALU_Control=add
end
default:begin
RegWrite=0;
ImmSel=3'b000;
```

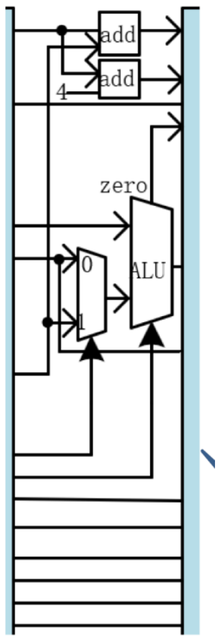
```
ALUSrc_B=1;
Branch=0;
Jump=0;
MemtoReg=3'b000;
MemRW=0;
ALUop=2'b00;
end
endcase
end
assign Fun = {Fun3,Fun7};
always @* begin
case(ALUop)
2'b00:begin ALU_Control =  32'd0; end//add 计算地址   lw,sw
2'b01:begin
case(Fun3)
3'b000:begin ALU_Control=32'd10;  end//beq
3'b001:begin ALU_Control=32'd11;  end//bne
3'b100:begin ALU_Control=32'd12;  end//blt
3'b101:begin ALU_Control=32'd13;  end//bge
3'b110:begin ALU_Control=32'd14;  end//bltu
3'b111:begin ALU_Control=32'd15;  end//bgeu
endcase//sub 比较条件   beq
end
2'b10:begin  //实现了第四行的全部指令
case({Fun3,Fun7}) //R-formats
4'b0000:begin ALU_Control = 32'd0 ; end//add
4'b0001:begin ALU_Control = 32'd1 ; end//sub
4'b0010:begin ALU_Control = 32'd2 ; end//sll
4'b0100:begin ALU_Control = 32'd3 ; end//slt
4'b0110:begin ALU_Control = 32'd4 ; end//sltu
4'b1000:begin ALU_Control = 32'd5 ; end//xor
4'b1010:begin ALU_Control = 32'd6 ; end//srl
4'b1011:begin ALU_Control = 32'd7 ; end//sra
4'b1100:begin ALU_Control = 32'd8 ; end//or
4'b1110:begin ALU_Control = 32'd9 ; end//and
default:begin ALU_Control=32'bx ; end
endcase
end
2'b11:begin
case(Fun3) //I-format 实现了第三行的全部指令
3'b000:begin ALU_Control = 32'd0;end//addi
3'b010:begin ALU_Control = 32'd3;end//slti
3'b011:begin ALU_Control = 32'd4;end//sltiu
3'b100:begin ALU_Control = 32'd5;end//xori
3'b110:begin ALU_Control = 32'd8;end//ori
3'b111:begin ALU_Control = 32'd9;end//andi
3'b101:begin ALU_Control = Fun7==0?32'd6:32'd7;end //srli srai
//case(Fun7)
//1'b0:begin ALU_Control = 32'd6;end//srli
//1'b1:begin ALU_Control = 32'd7;end//srai
```

```
//endcase
3'b001:begin ALU_Control = 32'd2;end//slli
endcase
end
endcase
end
endmodule
```

ID_Reg_EX 设计

```
module ID_Reg_EX(
input clk_IDEX,
input rst_IDEX,
input en_IDEX,
input NOP_IDEX,
input valid_in_IDEX,
input [31:0]PC_in_IDEX,
input [31:0]inst_in_IDEX,
//input [31:0]Wt_addr_IDEX,
input [31:0]Rs1_in_IDEX,
input [31:0]Rs2_in_IDEX,
input [31:0]Imm_in_IDEX,
input ALUSrc_B_in_IDEX,
input [31:0] ALU_control_in_IDEX,
input Branch_in_IDEX,
input MemRW_in_IDEX,
input [1:0]Jump_in_IDEX,
input [2:0]MemtoReg_in_IDEX,
input RegWrite_in_IDEX,
input [4:0] Rd_addr_IDEX,
output reg[31:0]inst_out_IDEX,
output reg[31:0]PC_out_IDEX,
//output reg[4:0]Wt_addr_out_IDEX,
output reg[31:0] Rs1_out_IDEX,
output reg[31:0] Rs2_out_IDEX,
output reg[31:0] Imm_out_IDEX,
output reg ALUSrc_B_IDEX,
output reg[31:0]ALU_control_out_IDEX,
output reg Branch_out_IDEX,
output reg[1:0] Jump_out_IDEX,
output reg MemRW_out_IDEX,
output reg RegWrite_out_IDEX,
output reg [2:0]MemtoReg_out_IDEX,
output reg [4:0] Rd_addr_out_IDEX,
output reg valid_out_IDEX
    );
    always@(posedge clk_IDEX or posedge rst_IDEX) begin
    if(rst_IDEX)
    begin
    PC_out_IDEX=0;
     Rs1_out_IDEX=0;
    Rs2_out_IDEX=0;
    Imm_out_IDEX=0;
    ALUSrc_B_IDEX=0;
    ALU_control_out_IDEX=0;
    Branch_out_IDEX=0;
    Jump_out_IDEX=0;
    MemRW_out_IDEX=0;
    RegWrite_out_IDEX=0;
    MemtoReg_out_IDEX=0;
    Rd_addr_out_IDEX=0;
    inst_out_IDEX=0;
    valid_out_IDEX=0;
    end
    else
    begin
    if(NOP_IDEX)
    begin
//   valid_out_IDEX=0;
//   inst_out_IDEX=32'h00000013;
//   valid_out_IDEX=0;
//disable the RegWrite and MemRW
   PC_out_IDEX=PC_in_IDEX;
    Rs1_out_IDEX=Rs1_in_IDEX;
    Rs2_out_IDEX=Rs2_in_IDEX;
    Imm_out_IDEX=Imm_in_IDEX;
    ALUSrc_B_IDEX=ALUSrc_B_in_IDEX;
    ALU_control_out_IDEX=ALU_control_in_IDEX;
    Branch_out_IDEX=Branch_in_IDEX;
    Jump_out_IDEX=Jump_in_IDEX;
    MemRW_out_IDEX=0;
    RegWrite_out_IDEX=0;
    MemtoReg_out_IDEX=MemtoReg_in_IDEX;
    Rd_addr_out_IDEX=Rd_addr_IDEX;
    inst_out_IDEX=inst_in_IDEX;
    valid_out_IDEX=0;
    end
    else if(en_IDEX)
    begin
    PC_out_IDEX=PC_in_IDEX;
    Rs1_out_IDEX=Rs1_in_IDEX;
    Rs2_out_IDEX=Rs2_in_IDEX;
    Imm_out_IDEX=Imm_in_IDEX;
    ALUSrc_B_IDEX=ALUSrc_B_in_IDEX;
    ALU_control_out_IDEX=ALU_control_in_IDEX;
    Branch_out_IDEX=Branch_in_IDEX;
    Jump_out_IDEX=Jump_in_IDEX;
    MemRW_out_IDEX=MemRW_in_IDEX;
    RegWrite_out_IDEX=RegWrite_in_IDEX;
    MemtoReg_out_IDEX=MemtoReg_in_IDEX;
    Rd_addr_out_IDEX=Rd_addr_IDEX;
    inst_out_IDEX=inst_in_IDEX;
    valid_out_IDEX=valid_in_IDEX;
    end
    end
    end
endmodule
```

1.4.3 EX 执行模块



Pipeline_EX 设计

```
module Pipeline_EX(
input[31:0] PC_in_EX, //PC 输入
input[31:0] Rs1_in_EX, //操作数 1 输入
input[31:0] Rs2_in_EX, //操作数 2 输入
input[31:0] Imm_in_EX , //立即数输入
input ALUSrc_B_in_EX , //ALU B 选择
input[31:0] ALU_control_in_EX, //ALU 选择控制
output [31:0] PC_out_EX, //PC 输出
output [31:0] PC4_out_EX, //PC+4 输出
output zero_out_EX, //ALU 判 0 输出
output [31:0] ALU_out_EX, //ALU 计算输出
output [31:0] Rs2_out_EX, //操作数 2 输出
output [31:0] dMem_out_EX
);

assign Rs2_out_EX=Rs2_in_EX;
wire null;

wire [31:0] ALU_nextdata;
add32 add_32_1(
.a(PC_in_EX),
.b(Imm_in_EX),
.c(PC_out_EX));
```

```
add32 add_32_2(
.a(PC_in_EX),
.b(4),
.c(PC4_out_EX));

MUX2T1 MUX2T1_32_1(
.I0(Rs2_in_EX),
.I1(Imm_in_EX),
.s(ALUSrc_B_in_EX),
.o(ALU_nextdata));

ALU ALU_0(
.a_val(Rs1_in_EX),
.b_val(ALU_nextdata),
.ctrl(ALU_control_in_EX),
.result(ALU_out_EX),
.zero(zero_out_EX));

ALU ALU_1(
.a_val(Rs1_in_EX),
.b_val(ALU_nextdata),
.ctrl(ALU_control_in_EX),
.result(dMem_out_EX),
.zero(null));
Endmodule
```

子模块：add32

```
module add32(
input [31:0] a,
input [31:0] b,
output [31:0] c

   );
  assign c=a+b;
endmodule
```

子模块：MUX2T1

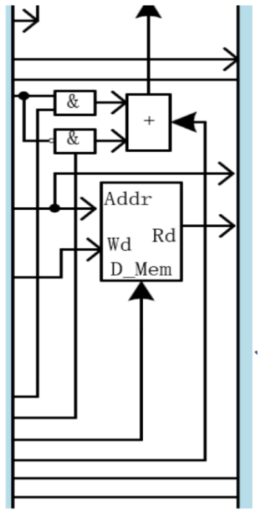
```
module MUX2T1(
input s,
input [31:0]I0,
input [31:0]I1,
output [31:0]o
   );
   assign o=(s==0)?I0:I1;
endmodule
```

子模块：ALU

```
module ALU(
input [31:0] a_val,
input [31:0] b_val,
input [31:0] ctrl,
output reg [31:0] result,
output reg zero
);
   always@(*)
   begin
   case(ctrl)
   32'd0:result=a_val+b_val;
   32'd1:result=a_val-b_val;
   32'd2:result=(a_val<<b_val[4:0]);//逻辑左移 SLL
   32'd3:if(a_val[31]==b_val[31])//slt
   begin
   if(a_val<b_val) result=1;
   else result=0;
   end
   else if(a_val[31]==1)
   result=1;
   else
   result=0;
   32'd4:if(a_val<b_val) result=1;//sltu
   else result=0;
   32'd5:result=a_val^b_val;//异或
   32'd6:result=a_val>>b_val[4:0];//逻辑右移 SRL
   32'd7:result=($signed(a_val))>>>b_val[4:0];//算术右移
   32'd8:result=a_val|b_val;//或
   32'd9:result=a_val&b_val;//且
   32'd10:begin//beq
   if(a_val==b_val)
   zero=1;
   else
   zero=0;
   end
   32'd11:begin//bne
   if(a_val!=b_val)
   zero=1;
   else
   zero=0;
   end
   32'd12:begin//blt
   if(a_val[31]==b_val[31])//slt
   begin
   if(a_val<b_val) zero=1;
   else zero=0;
   end
   else if(a_val[31]==1)
   zero=1;
   else
   zero=0;
   end
```

```
        32'd13:begin//blge
        if(a_val[31]==b_val[31])//slt
        begin
        if(a_val<b_val) zero=0;
        else zero=1;
        end
        else if(a_val[31]==1)
        zero=0;
        else
        zero=1;
        end
        32'd14:begin
        if(a_val<b_val)
        zero=1;
        else
        zero=0;
        end
        32'd15:
        if(a_val>=b_val)
        zero=1;
        else
        zero=0;
        endcase
        end
    endmodule
```

1.4.4 Mem 执行模块



EX_reg_Mem 设计

```
    EX_reg_Mem
    module Ex_reg_Mem(
    input [31:0]PC_cur_in_EXMem,
    output reg [31:0]PC_cur_out_EXMem,
    input valid_in_EXMem,
    input clk_EXMem, //寄存器时钟
    input rst_EXMem, //寄存器复位
```

```
input [31:0]dMem_out_EX,
output reg[31:0]dMem_out_EXMem,
input en_EXMem, //寄存器使能
input[31:0] PC_in_EXMem, //PC 输入
input[31:0] PC4_in_EXMem, //PC+4 输入
input [4:0] Rd_addr_EXMem, //写目的寄存器地址输入
input zero_in_EXMem, //zero
input[31:0] ALU_in_EXMem, //ALU 输入
input[31:0] Rs2_in_EXMem ,//操作数 2 输入
input Branch_in_EXMem, //Branch
input MemRW_in_EXMem, //存储器读写
input [1:0]Jump_in_EXMem, //Jal、Jalr
input [2:0] MemtoReg_in_EXMem, //写回
input RegWrite_in_EXMem, //寄存器堆读写
input [31:0]Imm_in_EXMem,
input [31:0]inst_in_EXMem,
output reg valid_out_EXMem,
output reg[31:0] PC_out_EXMem, //PC 输出
output reg[31:0] PC4_out_EXMem, //PC+4 输出
output reg[4:0] Rd_addr_out_EXMem, //写目的寄存器输出
output reg zero_out_EXMem, //zero
output reg[31:0] ALU_out_EXMem, //ALU 输出
output reg[31:0] Rs2_out_EXMem, //操作数 2 输出
output reg Branch_out_EXMem, //Branch
output reg MemRW_out_EXMem, //存储器读写
output reg[1:0] Jump_out_EXMem, //Jal、Jalr
output reg [2:0] MemtoReg_out_EXMem, //写回
output reg RegWrite_out_EXMem,//寄存器堆读写
output reg [31:0]Imm_out_EXMem,
output reg [31:0]inst_out_EXMem
);
always@(posedge clk_EXMem or posedge rst_EXMem)begin
if(rst_EXMem) begin
PC_out_EXMem=0;
PC4_out_EXMem=0;
Rd_addr_out_EXMem=0;
zero_out_EXMem=0;
ALU_out_EXMem=0;
Rs2_out_EXMem=0;
Branch_out_EXMem=0;
MemRW_out_EXMem=0;
Jump_out_EXMem=0;
MemtoReg_out_EXMem=0;
RegWrite_out_EXMem=0;
Imm_out_EXMem=0;
inst_out_EXMem=0;
valid_out_EXMem=0;
PC_cur_out_EXMem=0;
dMem_out_EXMem=0;
end
```

```
else begin
if(en_EXMem)begin
PC_out_EXMem=PC_in_EXMem;
PC4_out_EXMem=PC4_in_EXMem;
Rd_addr_out_EXMem=Rd_addr_EXMem;
zero_out_EXMem=zero_in_EXMem;
ALU_out_EXMem=ALU_in_EXMem;
Rs2_out_EXMem=Rs2_in_EXMem;
Branch_out_EXMem=Branch_in_EXMem;
MemRW_out_EXMem=MemRW_in_EXMem;
Jump_out_EXMem=Jump_in_EXMem;
MemtoReg_out_EXMem=MemtoReg_in_EXMem;
RegWrite_out_EXMem=RegWrite_in_EXMem;
Imm_out_EXMem=Imm_in_EXMem;
inst_out_EXMem=inst_in_EXMem;
valid_out_EXMem=valid_in_EXMem;
dMem_out_EXMem=dMem_out_EX;
PC_cur_out_EXMem=PC_cur_in_EXMem;
end
end
end
endmodule
```

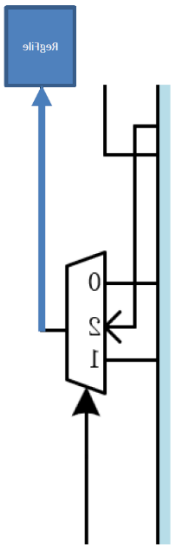
Pipeline_Mem 模块

```
module Pipeline_Mem(
input zero_in_Mem, //zero
input Branch_in_Mem, //beq
input [1:0]Jump_in_Mem, //jal
output reg[1:0]PCSrc//PC 选择控制输出
);

wire isBranch;
always@*begin
if(isBranch|Jump_in_Mem[0])
PCSrc=2'b01;
else if(Jump_in_Mem[1])
PCSrc=2'b10;
else
PCSrc=2'b00;
end

and_2 and_2_1(
.Op1(Branch_in_Mem),
.Op2(zero_in_Mem),
.Res(isBranch));

Endmodule
```

子模块：and_2

```
module and_2(
input  wire Op1,
input wire Op2,
output wire Res
   );
   assign Res=Op1&Op2;
endmodule
```

1.4.4 WB 执行模块



Pipeline_WB 设计

```
module Pipeline_WB(
input[31:0] PC4_in_WB, //PC+4 输入
input[31:0] ALU_in_WB, //ALU 结果输出
input[31:0] PC_in_WB,//新 pc,auipc
input [31:0] Imm_in_WB,//立即数输出,lui
input[31:0] Dmem_data_WB, //存储器数据输入
input[2:0] MemtoReg_in_WB, //写回选择控制
output [31:0] Data_out_WB //写回数据输出
);
   MUX5T1 MUX5T1_32_0(
   .s(MemtoReg_in_WB),
   .I0(ALU_in_WB),
   .I1(Dmem_data_WB),
   .I2(PC4_in_WB),
   .I3(Imm_in_WB),
   .I4(PC_in_WB),
   .o(Data_out_WB));
endmodule
```

子模块：MUX5T1

```
module MUX5T1(
input[2:0] s,
input [31:0]I0,
input [31:0]I1,
input [31:0]I2,
input [31:0]I3,
input [31:0]I4,
output reg[31:0] o
   );
   always@(*)
   begin
   if(s==3'b000)
   o=I0;
   else if(s==3'b001)
   o=I1;
   else if(s==3'b010)
   o=I2;
   else if(s==3'b011)
   o=I3;
   else
   o=I4;
   end
endmodule
```

Mem_reg_WB 设计

```
    module Mem_reg_WB(
    input [31:0]PC_cur_in_MemWB,
    output reg[31:0]PC_cur_out_MemWB,
    input clk_MemWB, //寄存器时
    input rst_MemWB, //寄存器复位
    input en_MemWB, //寄存器使能
    input valid_in_MemWB,
    input[31:0] PC_in_MemWB,
    input [31:0]inst_in_MemWB,
    input[31:0] PC4_in_MemWB, //PC+4 输入
    input[4:0] Rd_addr_MemWB, //写目的地址输入
    input[31:0] ALU_in_MemWB, //ALU 输入
    input[31:0] Dmem_data_MemWB, //存储器数据输入
    input[2:0] MemtoReg_in_MemWB, //写回
    input[31:0]Imm_in_MemWB,
    input RegWrite_in_MemWB, //寄存器堆读写
    output reg valid_out_MemWB,
    output reg[31:0]inst_out_MemWB,
    output reg[31:0] PC_out_MemWB,
    output reg[31:0] PC4_out_MemWB, //PC+4 输出
    output reg[4:0] Rd_addr_out_MemWB, //写目的地址输出
    output reg[31:0] ALU_out_MemWB, //ALU 输出
    output reg[31:0] DMem_data_out_MemWB,//存储器数据输出
    output reg[2:0] MemtoReg_out_MemWB, //写回
    output reg RegWrite_out_MemWB,//寄存器堆读写
    output reg [31:0]Imm_out_MemWB,
    input [1:0] PCSrc_in_MemWB,
    output reg [1:0]PCSrc_out_MemWB
```

```
);
always@(posedge clk_MemWB or posedge rst_MemWB)begin
if(rst_MemWB)
begin
PC4_out_MemWB=0;
Rd_addr_out_MemWB=0;
ALU_out_MemWB=0;
DMem_data_out_MemWB=0;
MemtoReg_out_MemWB=0;
RegWrite_out_MemWB=0;
Imm_out_MemWB=0;
PC_out_MemWB=0;
inst_out_MemWB=0;
valid_out_MemWB=0;
PC_cur_out_MemWB=0;
PCSrc_out_MemWB=0;
end
else
begin
if(en_MemWB)begin
PC4_out_MemWB=PC4_in_MemWB;
Rd_addr_out_MemWB=Rd_addr_MemWB;
ALU_out_MemWB=ALU_in_MemWB;
DMem_data_out_MemWB=Dmem_data_MemWB;
MemtoReg_out_MemWB=MemtoReg_in_MemWB;
RegWrite_out_MemWB=RegWrite_in_MemWB;
Imm_out_MemWB=Imm_in_MemWB;
PC_out_MemWB=PC_in_MemWB;
valid_out_MemWB=1;
inst_out_MemWB=inst_in_MemWB;
PC_cur_out_MemWB=PC_cur_in_MemWB;
PCSrc_out_MemWB=PCSrc_in_MemWB;
end
end
end
endmodule
```

Pipeline_WB 设计

```
module Pipeline_WB(
input[31:0] PC4_in_WB, //PC+4 输入
input[31:0] ALU_in_WB, //ALU 结果输出
input[31:0] PC_in_WB,//新 pc,auipc
input [31:0] Imm_in_WB,//立即数输出,lui
input[31:0] Dmem_data_WB, //存储器数据输入
input[2:0] MemtoReg_in_WB, //写回选择控制
output [31:0] Data_out_WB //写回数据输出
);
   MUX5T1 MUX5T1_32_0(
   .s(MemtoReg_in_WB),
   .I0(ALU_in_WB),
   .I1(Dmem_data_WB),
   .I2(PC4_in_WB),
   .I3(Imm_in_WB),
   .I4(PC_in_WB),
   .o(Data_out_WB));
Endmodule
```

子模块：MUX5T1

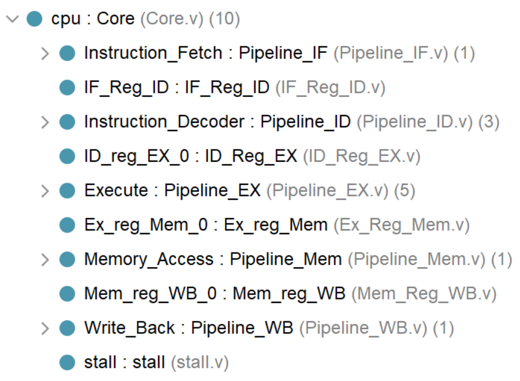
```
        module MUX5T1(
        input[2:0] s,
        input [31:0]I0,
        input [31:0]I1,
        input [31:0]I2,
        input [31:0]I3,
        input [31:0]I4,
        output reg[31:0] o
           );
           always@(*)
           begin
           if(s==3'b000)
           o=I0;
           else if(s==3'b001)
           o=I1;
           else if(s==3'b010)
           o=I2;
           else if(s==3'b011)
           o=I3;
           else
           o=I4;
           end
        endmodule
```

stall 设计：

```
module stall(
    input rst_stall, //复位
    input RegWrite_out_IDEX, //执行阶段寄存器写控制
    input [4:0]Rd_addr_out_IDEX, //执行阶段寄存器写地址
    input RegWrite_out_EXMem, //访存阶段寄存器写控制
    input [4:0]Rd_addr_out_EXMem, //访存阶段寄存器写地址
    input [4:0]Rs1_addr_ID, //译码阶段寄存器读地址 1
    input [4:0]Rs2_addr_ID, //译码阶段寄存器读地址 2
    input Rs1_used, //Rs1 被使用
    input Rs2_used, //Rs2 被使用
    input Branch_ID, //译码阶段 b-type
    input [1:0]Jump_ID, //译码阶段 jal
    input Branch_out_IDEX, //执行阶段 b-type
    input [1:0]Jump_out_IDEX, //执行阶段 jal
    input [1:0]PCSrc_out_MemWB,
    input Branch_out_EXMem, //访存阶段 b-type
    input [1:0]Jump_out_EXMem, //访存阶段 jal
    input [4:0]Rs1_addr_IDEX,
    input [4:0]Rs2_addr_IDEX,
    output reg en_IF, //流水线寄存器的使能及 NOP 信号
    output reg en_IFID,
    output reg NOP_IFID,
    output reg NOP_IDEx
    );
    always@* begin
    if(rst_stall)begin
    //按理来说 rst 之后是要能正常用的
    en_IF=1;
    en_IFID=1;
    NOP_IFID=0;
    NOP_IDEx=0;
    end
    else begin
    //Data_hazard
    //if(RegWrite_out_EXMem&&Rs1_used&&Rs1_addr_ID!=0&&Rd_addr_out_EXMem
    ==Rs1_addr_ID)
    NOP_IFID=0;
    NOP_IDEx=0;
    en_IF=1;
    en_IFID=1;
    if((Rs1_used&&Rs1_addr_ID!=0&&Rd_addr_out_EXMem==Rs1_addr_ID)||(Rs1_
    used&&Rs1_addr_ID!=0&&Rd_addr_out_IDEX==Rs1_addr_ID)||(Rs2_used&&Rs2
    _addr_ID!=0&&Rd_addr_out_IDEX==Rs2_addr_ID)||(Rs2_used&&Rs2_addr_ID!
    =0&&Rd_addr_out_EXMem==Rs2_addr_ID)||(Rs2_used&&Rs2_addr_IDEX!=0&&Rd
    _addr_out_EXMem==Rs2_addr_IDEX)||(Rs1_used&&Rs1_addr_IDEX!=0&&Rd_add
    r_out_EXMem==Rs1_addr_IDEX))
    begin
    //插入 stall
    NOP_IDEx=1;
    en_IF=0;
    en_IFID=0;
    end

    //Control
    hazard||(Branch_out_EXMem==1||Jump_out_EXMem==2'b01||Jump_out_EXMem=
    =2'b10)
    else if((Branch_ID==1||Jump_ID==2'b01||Jump_ID==2'b10))
    begin
    NOP_IFID=1;
    en_IF=0;
    en_IFID=0;
    end
    else
    if((Branch_out_IDEX==1||Jump_out_IDEX==2'b01||Jump_out_IDEX==2'b10))
    begin
    NOP_IFID=1;
```

```
//NOP_IDEx=1;
en_IF=0;
en_IFID=0;
end
else
if(Branch_out_EXMem==1||Jump_out_EXMem==2'b01||Jump_out_EXMem==2'b10)
begin
NOP_IFID=1;
//NOP_IDEx=1;
en_IF=0;
en_IFID=0;
end
else if(PCSrc_out_MemWB!=0)
begin
NOP_IFID=1;
en_IF=1;
en_IFID=1;
end
end
end
endmodule
```

这里是添加后的框架展示图。



```
cpu : Core (Core.v) (10)
    Instruction_Fetch : Pipeline_IF (Pipeline_IF.v) (1)
    IF_Reg_ID : IF_Reg_ID (IF_Reg_ID.v)
    Instruction_Decoder : Pipeline_ID (Pipeline_ID.v) (3)
    ID_reg_EX_0 : ID_Reg_EX (ID_Reg_EX.v)
    Execute : Pipeline_EX (Pipeline_EX.v) (5)
    Ex_reg_Mem_0 : Ex_reg_Mem (Ex_Reg_Mem.v)
    Memory_Access : Pipeline_Mem (Pipeline_Mem.v) (1)
    Mem_reg_WB_0 : Mem_reg_WB (Mem_Reg_WB.v)
    Write_Back : Pipeline_WB (Pipeline_WB.v) (1)
    stall : stall (stall.v)
```

流水线顶层设计：

```
module Core(
`VGA_DBG_Core_Outputs
input wire clk,
input wire rst,
input wire [31:0] imem_o_data,
input wire [31:0] dmem_o_data,
output wire MemRW,//
output wire [31:0]imem_addr,//
output wire [31:0]dmem_i_data,//
//output wire [31:0]PC_out_IF,/
//output wire [31:0] PC_out_EX_,//////
//output wire [31:0] PC4_out_EX_,//////
//output wire [1:0] PC_SRC_Mem,
output wire [31:0]dmem_addr
);

wire [31:0]PC_out_IF;
wire [31:0]PC_out_IDEX;
wire [31:0]PC_out_EX;
wire [31:0]PC_out_EXMem;
wire [31:0]PC_out_MemWB;
wire [31:0]PC4_out_EX;
wire [31:0]PC4_out_EXMem;
wire [31:0]PC4_out_MemWB;

wire [31:0]inst_ID;
wire [31:0]Data_out_WB;//WB 的输出

wire [31:0] Rs1_out_ID;
wire [31:0] Rs1_out_IDEX;
wire [31:0] Rs2_out_ID;
wire [31:0] Rs2_out_IDEX;
wire [31:0] Rs2_out_EX;
wire [31:0] Rs2_out_EXMem;
wire [4:0] Rd_addr_out_ID;
wire [4:0] Rd_addr_out_IDEX;
wire [4:0] Rd_addr_out_EXMem;
wire [4:0] Rd_addr_out_MemWB;
wire [31:0]Imm_out_ID;
wire [31:0]Imm_out_IDEX;
wire [31:0]Imm_out_EXMem;
wire [31:0]Imm_out_MemWB;
wire ALUSrc_B_ID;
wire ALUSrc_B_IDEX;
wire [31:0]ALU_control_ID;
wire [31:0]ALU_control_IDEX;
wire Branch_ID;
wire Branch_out_IDEX;
wire Branch_out_EXMem;
wire [1:0]Jump_ID;
wire [1:0]Jump_out_IDEX;
wire MemRW_ID;
wire MemRW_IDEX;
wire [2:0]MemtoReg_ID;
wire [2:0]MemtoReg_IDEX;
wire RegWrite_out_ID;
wire RegWrite_out_IDEX;
wire zero_out_EX;
```

```
    wire zero_out_EXMem;
    wire [31:0]ALU_out_EX;
    wire [31:0]ALU_out_EXMem;
    wire [31:0]ALU_out_MemWB;

    wire Branch_our_EXMem;
    wire [1:0]Jump_out_EXMem;
    wire [2:0]MemtoReg_out_EXMem;
    wire [2:0]MemtoReg_out_MemWB;
    wire RegWrite_out_EXMem;
    wire RegWrite_out_MemWB;
    wire [1:0]PCSrc;
    wire [31:0]DMem_data_out_MemWB;
    wire [31:0]inst_out_IDEX;
    wire [31:0]inst_out_EXMem;
    wire [31:0]inst_out_MemWB;

    wire valid_IFID;
    wire valid_out_IDEX;
    wire valid_out_EXMem;
    wire valid_out_MemWB;
    wire en_IF;
    wire en_IFID;
    wire NOP_IFID;
    wire NOP_IDEx;

//    assign PC_out_EX_=PC_out_EXMem;
//    assign PC4_out_EX_=PC4_out_EXMem;
       wire [1:0] PCSrc_out_MemWB;
       wire [31:0] inst_out_IF;

    Pipeline_IF  Instruction_Fetch(
    .inst_in_IF(imem_o_data),
    .inst_out_IF(inst_out_IF),
    .clk_IF(clk),
    .NOP_IFID(NOP_IFID),
    .rst_IF(rst),
    .en_IF(en_IF),
    .PC4_in_IF(imem_addr+4),
    .PC_in_IF(PC_out_MemWB),
    .PC_in_jalr(ALU_out_MemWB),
    .PCSrc(PCSrc_out_MemWB),
    .PC_out_IF(PC_out_IF));
```

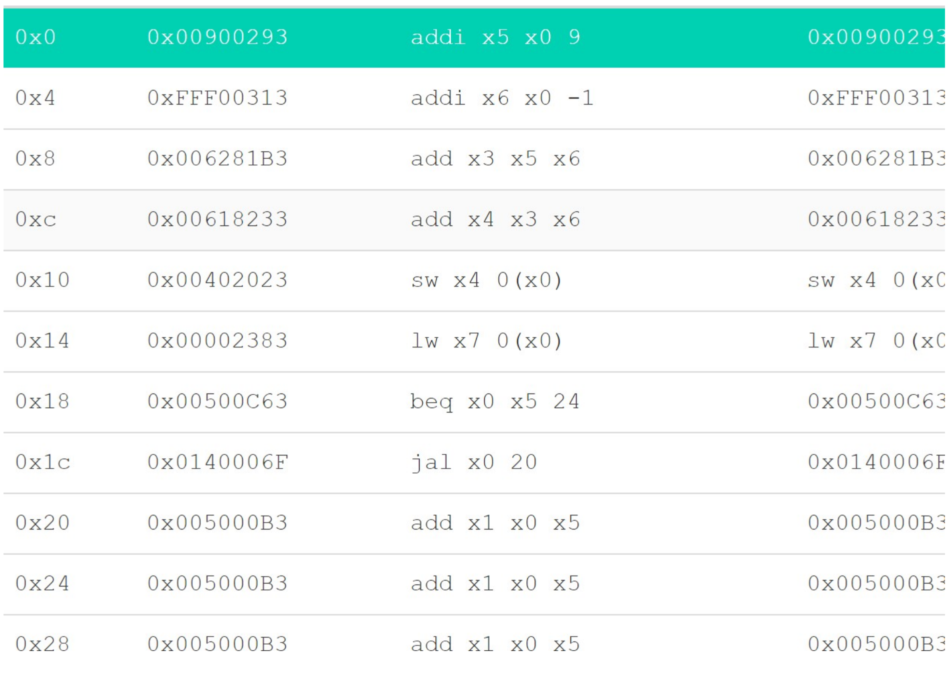
```
IF_Reg_ID IF_Reg_ID(
.clk_IFID(clk),
.rst_IFID(rst),
.en_IFID(en_IFID),
.PC_in_IFID(PC_out_IF),
.inst_in_IFID(inst_out_IF),
.NOP_IFID(NOP_IFID),
.PC_out_IFID(imem_addr),
.inst_out_IFID(inst_ID),
.valid_IFID(valid_IFID));

Pipeline_ID Instruction_Decoder(
`VGA_DBG_RegFile_Arguments
.clk_ID(clk),
.rst_ID(rst),
.RegWrite_in_ID(RegWrite_out_MemWB),
.Rd_addr_ID(Rd_addr_out_MemWB),//要写的 rd
.Wt_data_ID(Data_out_WB),
.Inst_in_ID(inst_ID),
.Rd_addr_out_ID(Rd_addr_out_ID),//译码产生的 rd
.Rs1_out_ID(Rs1_out_ID),
.Rs2_out_ID(Rs2_out_ID),
.Imm_out_ID(Imm_out_ID),
.ALUSrc_B_ID(ALUSrc_B_ID),
.ALU_control_ID(ALU_control_ID),
.Branch_ID(Branch_ID),
.MemRW_ID(MemRW_ID),
.Jump_ID(Jump_ID),
.MemtoReg_ID(MemtoReg_ID),
.RegWrite_out_ID(RegWrite_out_ID));

ID_Reg_EX ID_reg_EX_0(
.valid_in_IDEX(valid_IFID),
.valid_out_IDEX(valid_out_IDEX),
.clk_IDEX(clk),
.rst_IDEX(rst),
.en_IDEX(1),
.NOP_IDEX(NOP_IDEx),
.PC_in_IDEX(imem_addr),
.Rd_addr_IDEX(Rd_addr_out_ID),
.inst_in_IDEX(inst_ID),
.Rs1_in_IDEX(Rs1_out_ID),
.Rs2_in_IDEX(Rs2_out_ID),
.Imm_in_IDEX(Imm_out_ID),
.ALUSrc_B_in_IDEX(ALUSrc_B_ID),
.ALU_control_in_IDEX(ALU_control_ID),
.Branch_in_IDEX(Branch_ID),
.MemRW_in_IDEX(MemRW_ID),
.Jump_in_IDEX(Jump_ID),
.MemtoReg_in_IDEX(MemtoReg_ID),
.RegWrite_in_IDEX(RegWrite_out_ID),
.PC_out_IDEX(PC_out_IDEX),
.inst_out_IDEX(inst_out_IDEX),
.Rd_addr_out_IDEX(Rd_addr_out_IDEX),
.Rs1_out_IDEX(Rs1_out_IDEX),
.Rs2_out_IDEX(Rs2_out_IDEX),
.Imm_out_IDEX(Imm_out_IDEX),
.ALUSrc_B_IDEX(ALUSrc_B_IDEX),
.ALU_control_out_IDEX(ALU_control_IDEX),
.Branch_out_IDEX(Branch_out_IDEX),
.Jump_out_IDEX(Jump_out_IDEX),
.MemRW_out_IDEX(MemRW_IDEX),
.MemtoReg_out_IDEX(MemtoReg_IDEX),
.RegWrite_out_IDEX(RegWrite_out_IDEX));

wire [31:0]dMem_out_EX;
Pipeline_EX Execute(
.PC_in_EX(PC_out_IDEX),
.Rs1_in_EX(Rs1_out_IDEX),
.Rs2_in_EX(Rs2_out_IDEX),
.Imm_in_EX(Imm_out_IDEX),
.ALUSrc_B_in_EX(ALUSrc_B_IDEX),
.ALU_control_in_EX(ALU_control_IDEX),
.PC_out_EX(PC_out_EX),//加了立即数的 PC
.PC4_out_EX(PC4_out_EX),//加了 4 的 PC
.zero_out_EX(zero_out_EX),
.ALU_out_EX(ALU_out_EX),
.dMem_out_EX(dMem_out_EX),
.Rs2_out_EX(Rs2_out_EX));

wire [31:0]PC_cur_out_EXMem;
Ex_reg_Mem Ex_reg_Mem_0(
.dMem_out_EX(dMem_out_EX),
.dMem_out_EXMem(dmem_addr),
.PC_cur_in_EXMem(PC_out_IDEX),
.PC_cur_out_EXMem(PC_cur_out_EXMem),
.valid_in_EXMem(valid_out_IDEX),
.valid_out_EXMem(valid_out_EXMem),
.clk_EXMem(clk),
.rst_EXMem(rst),
.en_EXMem(1),
.inst_in_EXMem(inst_out_IDEX),
.PC_in_EXMem(PC_out_EX),//加了立即数的 PC
.PC4_in_EXMem(PC4_out_EX),
.Rd_addr_EXMem(Rd_addr_out_IDEX),
.zero_in_EXMem(zero_out_EX),
.ALU_in_EXMem(ALU_out_EX),
.Rs2_in_EXMem(Rs2_out_EX),
.Imm_in_EXMem(Imm_out_IDEX),
.Branch_in_EXMem(Branch_out_IDEX),
.MemRW_in_EXMem(MemRW_IDEX),
.Jump_in_EXMem(Jump_out_IDEX),
.MemtoReg_in_EXMem(MemtoReg_IDEX),
.RegWrite_in_EXMem(RegWrite_out_IDEX),
.PC_out_EXMem(PC_out_EXMem),
.PC4_out_EXMem(PC4_out_EXMem),
.Rd_addr_out_EXMem(Rd_addr_out_EXMem),
.zero_out_EXMem(zero_out_EXMem),
.ALU_out_EXMem(ALU_out_EXMem),
.Rs2_out_EXMem(dmem_i_data),
.Branch_out_EXMem(Branch_out_EXMem),
.MemRW_out_EXMem(MemRW),
.Jump_out_EXMem(Jump_out_EXMem),
.MemtoReg_out_EXMem(MemtoReg_out_EXMem),
.Imm_out_EXMem(Imm_out_EXMem),
.inst_out_EXMem(inst_out_EXMem),
.RegWrite_out_EXMem(RegWrite_out_EXMem));

Pipeline_Mem Memory_Access(
.zero_in_Mem(zero_out_EXMem),
.Branch_in_Mem(Branch_out_EXMem),
.Jump_in_Mem(Jump_out_EXMem),
.PCSrc(PCSrc));

wire [31:0]PC_cur_out_MemWB;

Mem_reg_WB Mem_reg_WB_0(
.PCSrc_in_MemWB(PCSrc),
.PCSrc_out_MemWB(PCSrc_out_MemWB),
.PC_cur_in_MemWB(PC_cur_out_EXMem),
.PC_cur_out_MemWB(PC_cur_out_MemWB),
.valid_in_MemWB(valid_out_EXMem),
.valid_out_MemWB(valid_out_MemWB),
.clk_MemWB(clk),
.rst_MemWB(rst),
.en_MemWB(1),
.inst_in_MemWB(inst_out_EXMem),
.PC_in_MemWB(PC_out_EXMem),
.PC4_in_MemWB(PC4_out_EXMem),
.Rd_addr_MemWB(Rd_addr_out_EXMem),
.Imm_in_MemWB(Imm_out_EXMem),
.ALU_in_MemWB(ALU_out_EXMem),
.Dmem_data_MemWB(dmem_o_data),
.MemtoReg_in_MemWB(MemtoReg_out_EXMem),
.RegWrite_in_MemWB(RegWrite_out_EXMem),
.PC_out_MemWB(PC_out_MemWB),
.PC4_out_MemWB(PC4_out_MemWB),
.Rd_addr_out_MemWB(Rd_addr_out_MemWB),
.ALU_out_MemWB(ALU_out_MemWB),
.Imm_out_MemWB(Imm_out_MemWB),
.DMem_data_out_MemWB(DMem_data_out_MemWB),
.MemtoReg_out_MemWB(MemtoReg_out_MemWB),
.inst_out_MemWB(inst_out_MemWB),
.RegWrite_out_MemWB(RegWrite_out_MemWB));

Pipeline_WB Write_Back(
.PC4_in_WB(PC4_out_MemWB),
.ALU_in_WB(ALU_out_MemWB),
.PC_in_WB(PC_out_MemWB),
.Imm_in_WB(Imm_out_MemWB),
.Dmem_data_WB(DMem_data_out_MemWB),
.MemtoReg_in_WB(MemtoReg_out_MemWB),
.Data_out_WB(Data_out_WB));


stall stall(
.rst_stall(rst),
.RegWrite_out_IDEX(RegWrite_out_IDEX), //执行阶段寄存器写控制
.Rd_addr_out_IDEX(Rd_addr_out_IDEX), //执行阶段寄存器写地址
.RegWrite_out_EXMem(RegWrite_out_EXMem), //访存阶段寄存器写控制
.Rd_addr_out_EXMem(Rd_addr_out_EXMem), //访存阶段寄存器写地址
.Rs1_addr_ID(inst_ID[19:15]), //译码阶段寄存器读地址 1
.Rs2_addr_ID(inst_ID[24:20]), //译码阶段寄存器读地址 2
.Rs1_addr_IDEX(inst_out_IDEX[19:15]),
.Rs2_addr_IDEX(inst_out_IDEX[24:20]),
    .Rs1_used(1), //Rs1 被使用
    .Rs2_used(1), //Rs2 被使用
    .Branch_ID(Branch_ID), //译码阶段 b-type
    .Jump_ID(Jump_ID), //译码阶段 jal
    .Branch_out_IDEX(Branch_out_IDEX), //执行阶段 b-type
    .Jump_out_IDEX(Jump_out_IDEX), //执行阶段 jal
    .Branch_out_EXMem(Branch_out_EXMem), //访存阶段 b-type
    .Jump_out_EXMem(Jump_out_EXMem), //访存阶段 jal
     .PCSrc_out_MemWB(PCSrc_out_MemWB),
     .en_IF(en_IF), //流水线寄存器的使能及 NOP 信号
    .en_IFID(en_IFID),
    .NOP_IFID(NOP_IFID),
    .NOP_IDEx(NOP_IDEx) );


//    assign dbg_pc=PC_out_IF;
    assign dbg_pc=PC_out_IF;
    assign dbg_inst= imem_o_data;
    assign dbg_IfId_pc = imem_addr;
    assign dbg_IfId_inst =inst_ID;
    assign dbg_IfId_valid = valid_IFID;
    assign dbg_IdEx_pc = PC_out_IDEX;
    assign dbg_IdEx_inst = inst_out_IDEX;
    assign dbg_IdEx_valid = valid_out_IDEX;
    assign dbg_IdEx_rd = Rd_addr_out_IDEX;
    assign dbg_IdEx_rs1 =inst_out_IDEX[19:15];
    assign dbg_IdEx_rs2 =inst_out_IDEX[24:20];
    assign dbg_IdEx_rs1_val =Rs1_out_IDEX;
    assign dbg_IdEx_rs2_val =Rs2_out_IDEX;
    assign dbg_IdEx_reg_wen = RegWrite_out_IDEX;
    assign dbg_IdEx_is_imm =ALUSrc_B_IDEX;
    assign dbg_IdEx_imm =Imm_out_IDEX;
    assign dbg_IdEx_mem_wen = MemRW_IDEX;
    assign dbg_IdEx_mem_ren = ~MemRW_IDEX;
    assign dbg_IdEx_is_branch = Branch_out_IDEX;
    assign dbg_IdEx_is_jal =( Jump_out_IDEX==2'b01)?1:0;
    assign dbg_IdEx_is_jalr = (Jump_out_IDEX==2'b10)?1:0;
    assign dbg_IdEx_is_auipc = (inst_out_IDEX[6:0]==7'b0010111)?1:0;
    assign dbg_IdEx_is_lui =  (inst_out_IDEX[6:0]==7'b0110111)?1:0;
    assign dbg_IdEx_alu_ctrl =ALU_control_IDEX;
    assign dbg_IdEx_cmp_ctrl = 0;
    assign dbg_ExMa_pc = PC_cur_out_EXMem;
    assign dbg_ExMa_inst =inst_out_EXMem;
    assign dbg_ExMa_valid = valid_out_EXMem;
    assign dbg_ExMa_rd = Rd_addr_out_EXMem;
    assign dbg_ExMa_reg_wen = RegWrite_out_EXMem;
    assign dbg_ExMa_mem_w_data = dmem_i_data;
    assign dbg_ExMa_alu_res = ALU_out_EXMem;
    assign dbg_ExMa_mem_wen = MemRW;
    assign dbg_ExMa_mem_ren = ~MemRW;
    assign dbg_ExMa_is_jal = (Jump_out_EXMem==2'b01)?1:0;
    assign dbg_ExMa_is_jalr = (Jump_out_EXMem==2'b10)?1:0;
    assign dbg_MaWb_pc =PC_cur_out_MemWB;
    assign dbg_MaWb_inst = inst_out_MemWB;
    assign dbg_MaWb_valid = valid_out_MemWB;
    assign dbg_MaWb_rd = Rd_addr_out_MemWB;
    assign dbg_MaWb_reg_wen = RegWrite_out_MemWB;
    assign dbg_MaWb_reg_w_data = Data_out_WB;
endmodule
```

二、实验结果验证

2.1 仿真展示

因为有无冒险的流水线设计本质是多了一个模块，采取的仿真指令设计与结果如下：

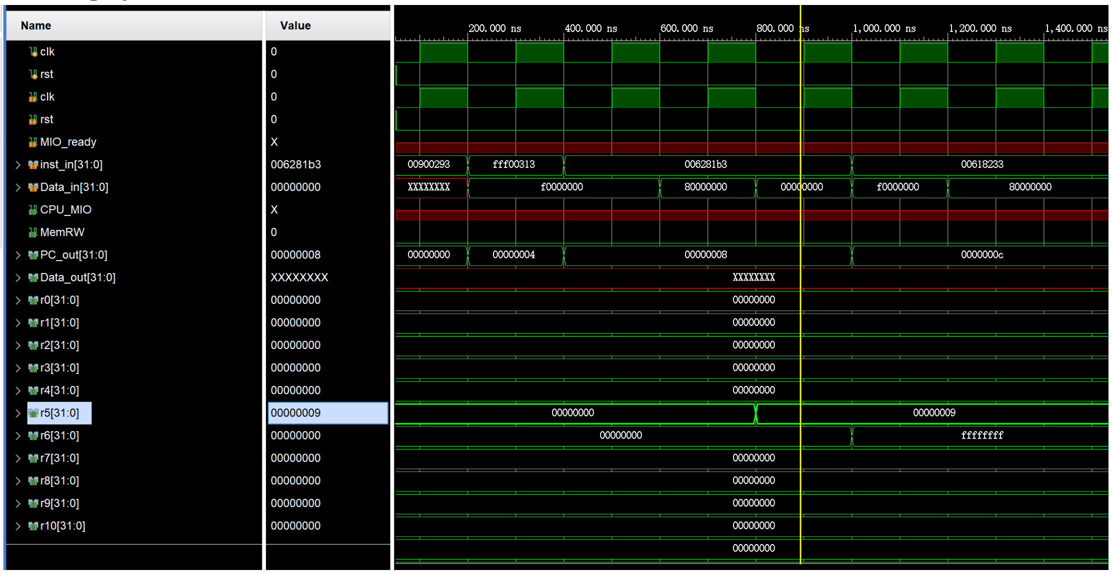
自己的代码如图：



0x0	0x00900293	addi x5 x0 9	0x0090029
0x4	0xFFF00313	addi x6 x0 -1	0xFFF0031
0x8	0x006281B3	add x3 x5 x6	0x006281B
0xc	0x00618233	add x4 x3 x6	0x0061823
0x10	0x00402023	sw x4 0(x0)	sw x4 0(x
0x14	0x00002383	lw x7 0(x0)	lw x7 0(x
0x18	0x00500C63	beq x0 x5 24	0x00500C6
0x1c	0x0140006F	jal x0 20	0x0140006
0x20	0x005000B3	add x1 x0 x5	0x005000B
0x24	0x005000B3	add x1 x0 x5	0x005000B
0x28	0x005000B3	add x1 x0 x5	0x005000B

首先测试第四条指令，测试 data hazard，之后测试连续的 hazard，比如 sw，lw 指令。最后

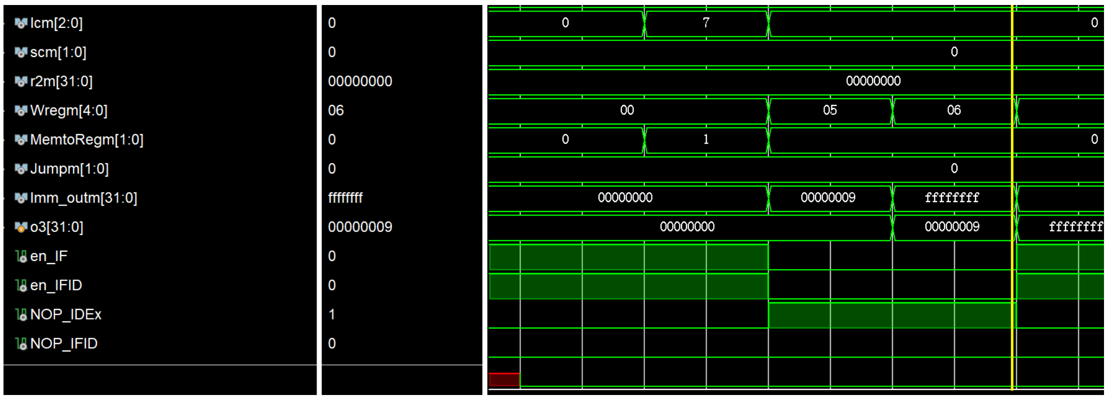
测试 beq，jal。测试如图：



```
Name             Value
clk              0
rst              0
clk              0
rst              0
MIO_ready        X
inst_in[31:0]    006281b3
Data_in[31:0]    00000000
CPU_MIO          X
MemRW            0
PC_out[31:0]     00000008
Data_out[31:0]   XXXXXXXX
r0[31:0]         00000000
r1[31:0]         00000000
r2[31:0]         00000000
r3[31:0]         00000000
r4[31:0]         00000000
r5[31:0]         00000009
r6[31:0]         00000000
r7[31:0]         00000000
r8[31:0]         00000000
r9[31:0]         00000000
r10[31:0]        00000000
```

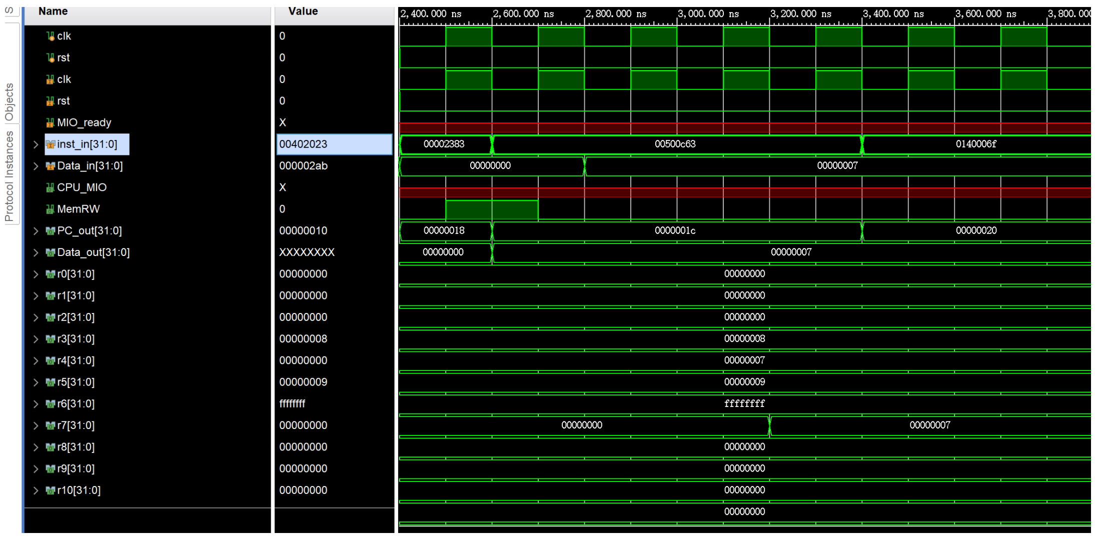
200.000 ns
400.000 ns
600.000 ns
800.000 ns
1,000.000 ns
1,200.000 ns
1,400.000 ns
00900293 fff00313 006281b3 00618233
XXXXXXXX f0000000 80000000 00000000 f0000000 80000000
00000000 00000004 00000008 0000000c
00000000 00000009
00000000 ffffffff

可以看见两个 add 指令都停了两拍，结果也存入。



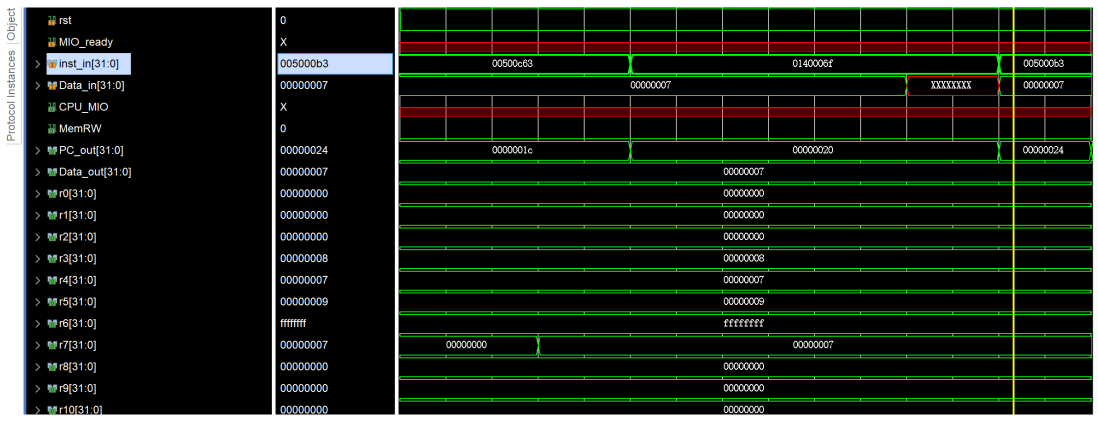
```
lcm[2:0]         0
scm[1:0]         0
r2m[31:0]        00000000
Wregm[4:0]       06
MemtoRegm[1:0]   0
Jumpm[1:0]       0
Imm_outm[31:0]   ffffffff
o3[31:0]         00000009
en_IF            0
en_IFID          0
NOP_IDEx         1
NOP_IFID         0
```

NOP正确。



```
Name             Value
clk              0
rst              0
clk              0
rst              0
MIO_ready        X
inst_in[31:0]    00402023
Data_in[31:0]    000002ab
CPU_MIO          X
MemRW            0
PC_out[31:0]     00000010
Data_out[31:0]   XXXXXXXX
r0[31:0]         00000000
r1[31:0]         00000000
r2[31:0]         00000000
r3[31:0]         00000008
r4[31:0]         00000000
r5[31:0]         00000009
r6[31:0]         ffffffff
r7[31:0]         00000000
r8[31:0]         00000000
r9[31:0]         00000000
r10[31:0]        00000000
```

如图，beq 和jal 对应的指令正确，停顿了三拍，符合 control hazard 的结果。



```
rst              0
MIO_ready        X
inst_in[31:0]    005000b3    00500c63 | 0140006f | 005000b3
Data_in[31:0]    00000007    00000007 | XXXXXXXX | 00000007
CPU_MIO          X
MemRW            0
PC_out[31:0]     00000024    0000001c | 00000020 | 00000024
Data_out[31:0]   00000007    00000007
r0[31:0]         00000000    00000000
r1[31:0]         00000000    00000000
r2[31:0]         00000000    00000000
r3[31:0]         00000008    00000008
r4[31:0]         00000007    00000007
r5[31:0]         00000009    00000009
r6[31:0]         ffffffff    ffffffff
r7[31:0]         00000007    00000000 | 00000007
r8[31:0]         00000000    00000000
r9[31:0]         00000000    00000000
r10[31:0]        00000000    00000000
```

Jal 也正确。

2.2 指令展示

由于 5.4 的使用的指令，实际上就是 5.3 的测试文件加上了中断的情况，因此这里仅选取了中断的 coe 文件展示流水线运行的正确性。

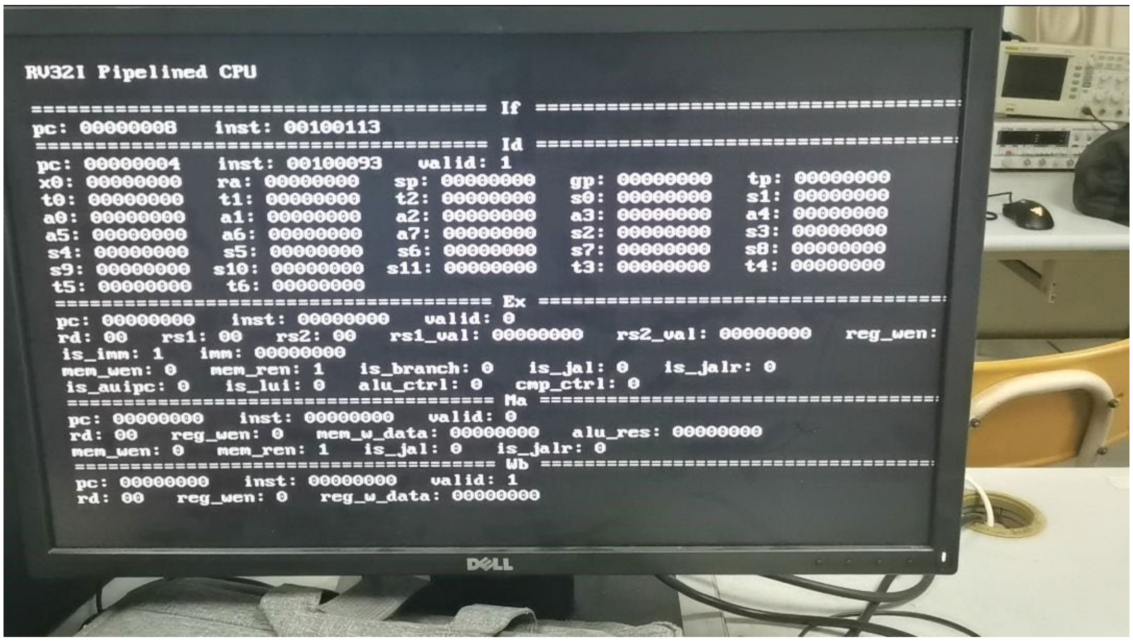
```
memory_initialization_radix=16;
memory_initialization_vector=
00000013,
00100093,
00100113,
00100193,
00100213,
00802283,
00128333,
0020C3B3,
40708433,
FFF1E493,
00327533,
00502223,
005325B3,
0AA3C613,
00818663,
00000013,
00000033,
0012D6B3,
00147713,
0034E7B3,
00A50833,
0085C8B3,
00402903,
004629B3,
0016DA13,
00A77AB3,
00C71463,
00000013,
40128B33,
00150B93,
00986C33,
00B9CCB3,
0FFA7D13,
00390DB3,
002A5E33,
0AF9EE93,
001A0F33,
00802F83,
F69FF06F,
001A8A93,
001B8B93,
001C8C93;
```

指令即为：

```
main:
addi x0,x0,0x0
addi x1,x0,0x1
#x1 = 0x1
addi x2,x0,0x1
#x2 = 0x1
addi x3,x0,0x1
#x3 = 0x1
addi x4,x0,0x1
#x4 = 0x1
lw x5,0x8(x0)
#x5 = 0x80000000
add x6,x5,x1
#x6 = 0x80000001
xor x7,x1,x2
#x7 = 0
sub x8,x1,x7
#x8 = 0x1
lw x9,0x5c(x0)
x0)#x9 = 0xFFFFFFFF
and x10,x4,x3
#x10= 0x1
sw x5,0x4(x0) #mem(1)=0x8000000
slt x11,x6,x5
#x11= 0x0
xori x12,x7,0xAA #x12= 0xAA
beq x3,x8,loop1
addi x0,x0,0x0
add x0,x0,x0
```

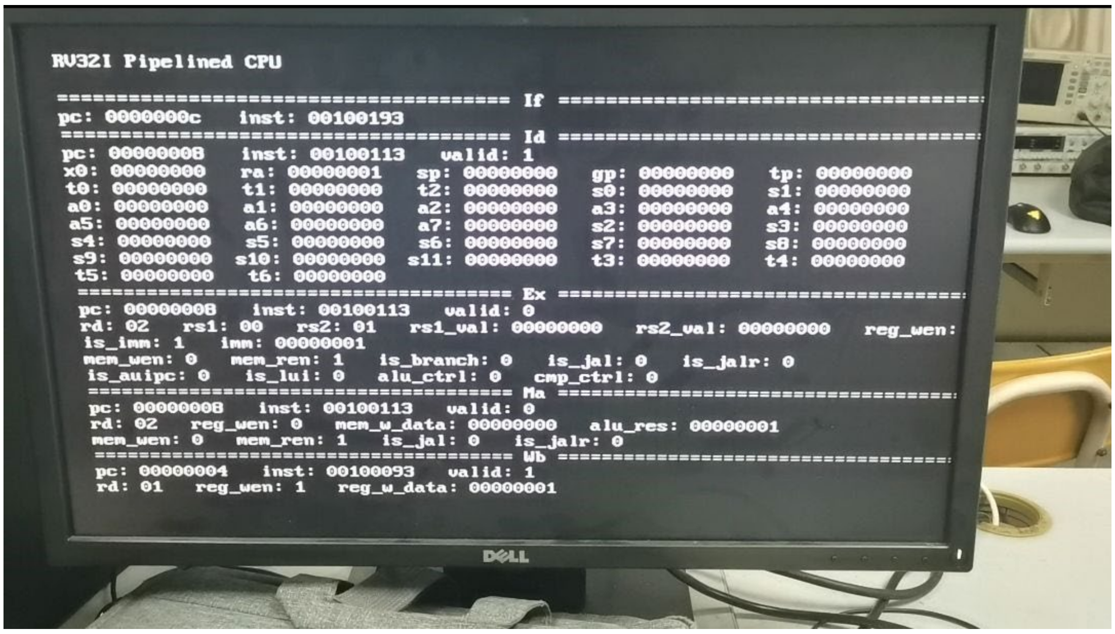
```
loop1:
srl x13,x5,x1 #x13= 0x40000000
andi x14,x8,0x1
#x14= 0x1
or x15,x9,x3
#x15= 0xFFFFFFFF
add x16,x10,x10 #x16= 0x2
xor x17,x11,x8 #x17= 0x1
lw x18,0x4(x0) #x18= 0x80000000
slt x19,x12,x4 #x19= 0
srli x20,x13,0x1 #x20= 0x20000000
and x21,x14,x10 #x21= 0x1
bne x14,x12,loop2
addi x0,x0,0x0
loop2:sub x22,x5,x1 #x22= 0x7FFFFFFF
addi x23,x10,0x1 #x23= 0x2
or x24,x16,x9 #x24= 0xFFFFFFFF
xor x25,x19,x11 #x25= 0x0
andi x26,x20,0xFF #x26= 0x200000FF
add x27,x18,x3 #x27= 0x80000001
srl x28,x20,x2 #x28= 0x10000000
ori x29,x19,0xAF #x29= 0xAF
add x30,x20,x1 #x30= 0x20000001
lw x31,0x8(x0) #x31= 0x80000000
jal x0,main
addi x21,x21,0
x1
addi x23,x23,0x1
adid x25,x25,0x1
```

下面进行分析：



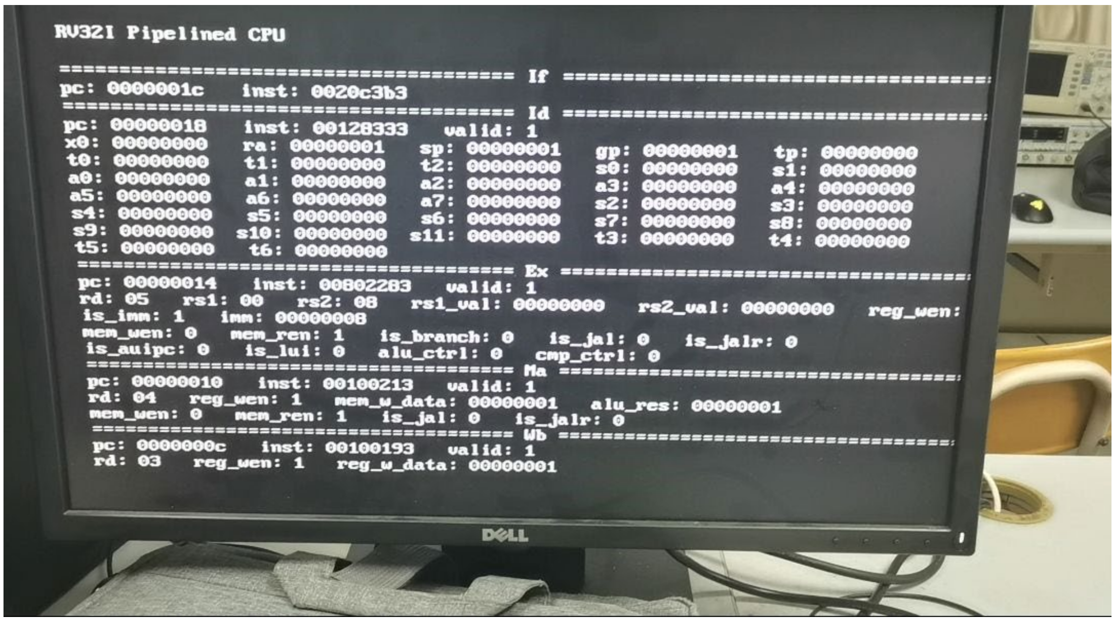
```
RV32I Pipelined CPU

======================================= If ======================================
pc: 00000008   inst: 00100113
======================================= Id ======================================
pc: 00000004   inst: 00100093   valid: 1
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:
is_imm: 1   imm: 00000000
mem_wen: 0   mem_ren: 1   is_branch: 0   is_jal: 0   is_jalr: 0
is_auipc: 0   is_lui: 0   alu_ctrl: 0   cmp_ctrl: 0
======================================= Ma ======================================
pc: 00000000   inst: 00000000   valid: 0
rd: 00   reg_wen: 0   mem_w_data: 00000000   alu_res: 00000000
mem_wen: 0   mem_ren: 1   is_jal: 0   is_jalr: 0
======================================= Wb ======================================
pc: 00000000   inst: 00000000   valid: 1
rd: 00   reg_wen: 0   reg_w_data: 00000000
```

这里是初始阶段，开始取指并且开始转化指令命令。IF 处的 PC 为 00000008，Id 处的 PC 为 00000004，寄存器值为 0，valid 为 1，一切正常。



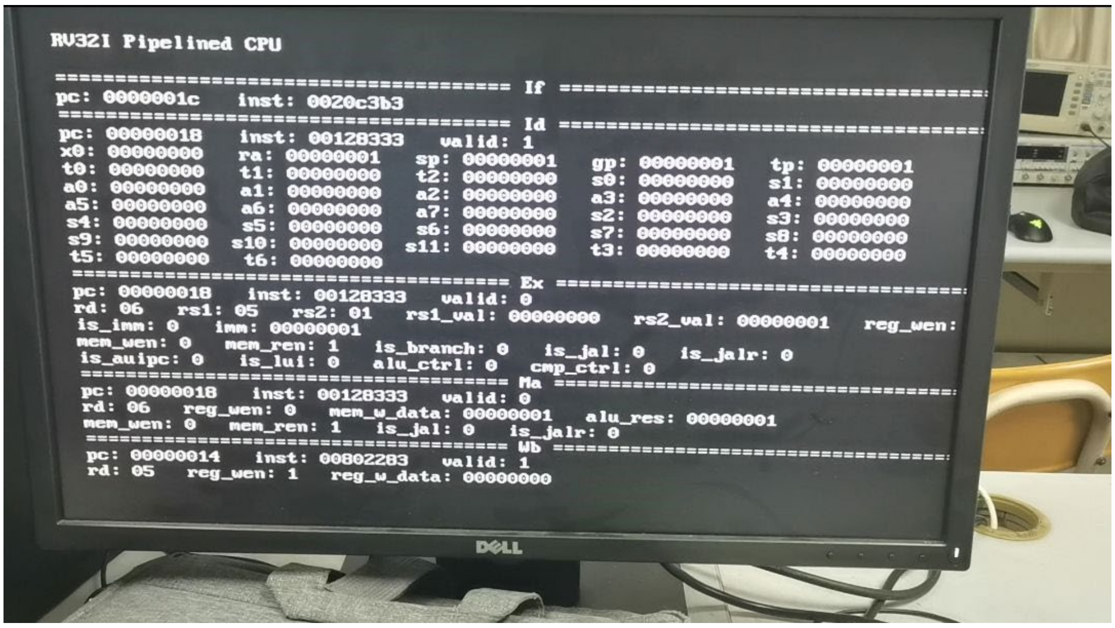
```
RV32I Pipelined CPU

======================================= If ======================================
pc: 0000000c   inst: 00100193
======================================= Id ======================================
pc: 00000008   inst: 00100113   valid: 1
x0: 00000000   ra: 00000001   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: 00000008   inst: 00100113   valid: 0
rd: 02   rs1: 00   rs2: 01   rs1_val: 00000000   rs2_val: 00000000   reg_wen:
is_imm: 1   imm: 00000001
mem_wen: 0   mem_ren: 1   is_branch: 0   is_jal: 0   is_jalr: 0
is_auipc: 0   is_lui: 0   alu_ctrl: 0   cmp_ctrl: 0
======================================= Ma ======================================
pc: 00000008   inst: 00100113   valid: 0
rd: 02   reg_wen: 0   mem_w_data: 00000000   alu_res: 00000001
mem_wen: 0   mem_ren: 1   is_jal: 0   is_jalr: 0
======================================= Wb ======================================
pc: 00000004   inst: 00100093   valid: 1
rd: 01   reg_wen: 1   reg_w_data: 00000001
```

这里是流水线已将第一条指令给运行完毕的时候，可以看到，Inst 为 00100093 这条指令，已经被送到了 WB 阶段，而对用的 add x1,x0,1 这句话已经被完整执行，可以看到，在 ra 寄存器中，值已经从上图的 00000000 改为了目前的 00000001，可以说明，流水线的运行是正常的。



```
RV32I Pipelined CPU

====================================== If ======================================
pc: 0000001c   inst: 0020c3b3
====================================== Id ======================================
pc: 00000018   inst: 00128333   valid: 1
x0: 00000000   ra: 00000001   sp: 00000001   gp: 00000001   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: 00000014   inst: 00802283   valid: 1
rd: 05   rs1: 00   rs2: 08   rs1_val: 00000000   rs2_val: 00000000   reg_wen:
is_imm: 1   imm: 00000008
mem_wen: 0   mem_ren: 1   is_branch: 0   is_jal: 0   is_jalr: 0
is_auipc: 0   is_lui: 0   alu_ctrl: 0   cmp_ctrl: 0
====================================== Ma ======================================
pc: 00000010   inst: 00100213   valid: 1
rd: 04   reg_wen: 1   mem_w_data: 00000001   alu_res: 00000001
mem_wen: 0   mem_ren: 1   is_jal: 0   is_jalr: 0
====================================== Wb ======================================
pc: 0000000c   inst: 00100193   valid: 1
rd: 03   reg_wen: 1   reg_w_data: 00000001
```

从 pc=00000018 开始，展示 stall 的功能，这里的指令为 00128333，即 add x6,x5,x1，而前一个 pc=00000014 时，inst 为 00802283，即 lw x5,0x8(x0)。由于 lw 指令的存在，我们可以推测，需要停顿三拍进行等待。



```
RV32I Pipelined CPU

====================================== If ======================================
pc: 0000001c   inst: 0020c3b3
====================================== Id ======================================
pc: 00000018   inst: 00128333   valid: 1
x0: 00000000   ra: 00000001   sp: 00000001   gp: 00000001   tp: 00000001
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: 00000018   inst: 00128333   valid: 0
rd: 06   rs1: 05   rs2: 01   rs1_val: 00000000   rs2_val: 00000001   reg_wen:
is_imm: 0   imm: 00000001
mem_wen: 0   mem_ren: 1   is_branch: 0   is_jal: 0   is_jalr: 0
is_auipc: 0   is_lui: 0   alu_ctrl: 0   cmp_ctrl: 0
====================================== Ma ======================================
pc: 00000018   inst: 00128333   valid: 0
rd: 06   reg_wen: 0   mem_w_data: 00000001   alu_res: 00000001
mem_wen: 0   mem_ren: 1   is_jal: 0   is_jalr: 0
====================================== Wb ======================================
pc: 00000014   inst: 00802283   valid: 1
rd: 05   reg_wen: 1   reg_w_data: 00000000
```

可以看到，这里确实停了三拍，我们可以看到 pc 在 00000018 处停留了三拍，所以 stall 的设计是正确的。其余的实验结果在验收时进行了展示，完成了后续冒险的指令。

三、讨论、心得

本实验书写难度不大，就是在4.3的基础上加上几个寄存器。由于4.3使用了很多.v文

件，因此诸如ALU，SCPU_Ctrl等模块都不需要进行更改，如果有bug也不会考虑是这些模块的问题。需要考虑的就是所有的时序问题：reg在上升沿还是下降沿处理、PC的处理情况，32位寄存器的读写时序以及RAM,ROM的读写情况等等。理论上分析不够，需要多种情况进行讨论，分析，进行不断的下板尝试得出结论。

5.4在处理 hazard上面难度不高，主要是对理论课的复刻。关键问题有两点，第一点是hazard情况复杂，需要传入很多参数，需要很多实现方法。因此非常容易出现bug。第二点是由于5.3的实现比较多样，指令在hazard方面会出现很多情况。简单实现方法是所有的ld指令全部 nop 三拍，可能效率较低。对于stall的实现，最初我的理解有问题，关于NOP和EN的情况欠缺考虑，后来在重新学习了理论课才掌握。

本实验最难的一点就是自己的代码的独特情况的处理，包括5.3的调试也是，技巧性和重复性较高，知识性较低，关键是掌握调试技巧。