计算机体系结构 - Exp1 Report

一、实验目的和要求

实验目的

•温故流水线CPU设计

•了解并实现RV32I指令集

•理解旁路优化（Forwarding）

实验要求

1.实现RV32I中所有指令(除了fence, ecall, ebreak)

2.实现流水线中的forwarding

3.通过仿真测试和上板验证

二、实验内容和原理

RV32core

这一部分主要是对照data path进行连线，填补多路选择器的信号等。

```
`timescale 1ns / 1ps


module  RV32core(
        input debug_en,  // debug enable
        input debug_step,  // debug step clock
        input [6:0] debug_addr,  // debug address
        output[31:0] debug_data,  // debug data
        input clk,  // main clock
        input rst,  // synchronous reset
        input interrupter  // interrupt source, for future use
    );

    wire debug_clk;

    debug_clk
clock(.clk(clk),.debug_en(debug_en),.debug_step(debug_step),.debug_clk(debug_clk
));

    wire Branch_ctrl, JALR, RegWrite_ctrl, mem_w_ctrl, MIO_ctrl,
        ALUSrc_A_ctrl, ALUSrc_B_ctrl, DatatoReg_ctrl, rs1use_ctrl, rs2use_ctrl;
    wire[1:0] hazard_optype_ctrl;
    wire[2:0] ImmSel_ctrl, cmp_ctrl;
    wire[3:0] ALUControl_ctrl;

    wire forward_ctrl_ls;
    wire[1:0] forward_ctrl_A, forward_ctrl_B;
```

```
    wire PC_EN_IF;
    wire [31:0] PC_IF, next_PC_IF, PC_4_IF, inst_IF;

    wire reg_FD_EN,reg_FD_stall,reg_FD_flush, cmp_res_ID;
    wire [31:0] jump_PC_ID, PC_ID, inst_ID, Debug_regs, rs1_data_reg,
rs2_data_reg,
        Imm_out_ID, rs1_data_ID, rs2_data_ID, addA_ID;

    wire reg_DE_EN, reg_DE_flush, RegWrite_EXE, mem_w_EXE, MIO_EXE,
        ALUSrc_A_EXE, ALUSrc_B_EXE, ALUzero_EXE, ALUoverflow_EXE, DatatoReg_EXE;
    wire[2:0] u_b_h_w_EXE;
    wire[3:0] ALUControl_EXE;
    wire[4:0] rs1_EXE, rs2_EXE, rd_EXE;
    wire[31:0] ALUout_EXE, PC_EXE, inst_EXE, rs1_data_EXE, rs2_data_EXE,
Imm_EXE,
        ALUA_EXE, ALUB_EXE, Dataout_EXE;

    wire reg_EM_EN, reg_EM_flush, RegWrite_MEM, DatatoReg_MEM, mem_w_MEM,
MIO_MEM;
    wire[2:0] u_b_h_w_MEM;
    wire[4:0] rd_MEM;
    wire[31:0] ALUout_MEM, PC_MEM, inst_MEM, Dataout_MEM, Datain_MEM;



    wire reg_MW_EN, RegWrite_WB, DatatoReg_WB;
    wire[4:0] rd_WB;
    wire [31:0] wt_data_WB, PC_WB, inst_WB, ALUout_WB, Datain_WB;



    // IF
    REG32
REG_PC(.clk(debug_clk),.rst(rst),.CE(PC_EN_IF),.D(next_PC_IF),.Q(PC_IF));

    add_32 add_IF(.a(PC_IF),.b(32'd4),.c(PC_4_IF));

    MUX2T1_32
mux_IF(.I0(PC_4_IF),.I1(jump_PC_ID),.s(Branch_ctrl),.o(next_PC_IF));        //to
fill sth. in ()

    ROM_D inst_rom(.a(PC_IF[8:2]),.spo(inst_IF));



    // ID
    REG_IF_ID
reg_IF_ID(.clk(debug_clk),.rst(rst),.EN(reg_FD_EN),.Data_stall(reg_FD_stall),
        .flush(reg_FD_flush),.PCOUT(PC_IF),.IR(inst_IF),

        .IR_ID(inst_ID),.PCurrent_ID(PC_ID));

    CtrlUnit
ctrl(.inst(inst_ID),.cmp_res(cmp_res_ID),.Branch(Branch_ctrl),.ALUSrc_A(ALUSrc_A
_ctrl),

.ALUSrc_B(ALUSrc_B_ctrl),.DatatoReg(DatatoReg_ctrl),.RegWrite(RegWrite_ctrl),
```

```
.mem_w(mem_w_ctrl),.MIO(MIO_ctrl),.rs1use(rs1use_ctrl),.rs2use(rs2use_ctrl),

.hazard_optype(hazard_optype_ctrl),.ImmSel(ImmSel_ctrl),.cmp_ctrl(cmp_ctrl),
        .ALUControl(ALUControl_ctrl),.JALR(JALR));

    Regs
register(.clk(debug_clk),.rst(rst),.L_S(RegWrite_WB),.R_addr_A(inst_ID[19:15]),
        .R_addr_B(inst_ID[24:20]),.rdata_A(rs1_data_reg),.rdata_B(rs2_data_reg),
        .Wt_addr(rd_WB),.Wt_data(wt_data_WB),
        .Debug_addr(debug_addr[4:0]),.Debug_regs(Debug_regs));

    ImmGen
imm_gen(.ImmSel(ImmSel_ctrl),.inst_field(inst_ID),.Imm_out(Imm_out_ID));

    MUX4T1_32
mux_forward_A(.I0(rs1_data_reg),.I1(ALUout_EXE),.I2(ALUout_MEM),.I3(Datain_MEM),
       //to fill sth. in ()
        .s(forward_ctrl_A),.o(rs1_data_ID));

    MUX4T1_32
mux_forward_B(.I0(rs2_data_reg),.I1(ALUout_EXE),.I2(ALUout_MEM),.I3(Datain_MEM),
       //to fill sth. in ()
        .s(forward_ctrl_B),.o(rs2_data_ID));

    MUX2T1_32 mux_branch_ID(.I0(PC_ID),.I1(rs1_data_ID),.s(JALR),.o(addA_ID));

    add_32 add_branch_ID(.a(addA_ID),.b(Imm_out_ID),.c(jump_PC_ID));

    cmp_32
cmp_ID(.a(rs1_data_ID),.b(rs2_data_ID),.ctrl(cmp_ctrl),.c(cmp_res_ID));

    HazardDetectionUnit
hazard_unit(.clk(debug_clk),.Branch_ID(Branch_ctrl),.rs1use_ID(rs1use_ctrl),

.rs2use_ID(rs2use_ctrl),.hazard_optype_ID(hazard_optype_ctrl),.rd_EXE(rd_EXE),

.rd_MEM(rd_MEM),.rs1_ID(inst_ID[19:15]),.rs2_ID(inst_ID[24:20]),.rs2_EXE(rs2_EXE
),
        .PC_EN_IF(PC_EN_IF),.reg_FD_EN(reg_FD_EN),.reg_FD_stall(reg_FD_stall),

.reg_FD_flush(reg_FD_flush),.reg_DE_EN(reg_DE_EN),.reg_DE_flush(reg_DE_flush),
       .reg_EM_EN(reg_EM_EN),.reg_EM_flush(reg_EM_flush),.reg_MW_EN(reg_MW_EN),
        .forward_ctrl_ls(forward_ctrl_ls),.forward_ctrl_A(forward_ctrl_A),
        .forward_ctrl_B(forward_ctrl_B));


    // EX
    REG_ID_EX
reg_ID_EX(.clk(debug_clk),.rst(rst),.EN(reg_DE_EN),.flush(reg_DE_flush),.IR_ID(i
nst_ID),

.PCurrent_ID(PC_ID),.rs1_addr(inst_ID[19:15]),.rs2_addr(inst_ID[24:20]),.rs1_dat
a(rs1_data_ID),
```

```
.rs2_data(rs2_data_ID),.Imm32(Imm_out_ID),.rd_addr(inst_ID[11:7]),.ALUSrc_A(ALUS
rc_A_ctrl),

.ALUSrc_B(ALUSrc_B_ctrl),.ALUC(ALUControl_ctrl),.DatatoReg(DatatoReg_ctrl),

.RegWrite(RegWrite_ctrl),.WR(mem_w_ctrl),.u_b_h_w(inst_ID[14:12]),.MIO(MIO_ctrl)
,

        .PCurrent_EX(PC_EXE),.IR_EX(inst_EXE),.rs1_EX(rs1_EXE),.rs2_EX(rs2_EXE),

.A_EX(rs1_data_EXE),.B_EX(rs2_data_EXE),.Imm32_EX(Imm_EXE),.rd_EX(rd_EXE),

.ALUSrc_A_EX(ALUSrc_A_EXE),.ALUSrc_B_EX(ALUSrc_B_EXE),.ALUC_EX(ALUControl_EXE),

.DatatoReg_EX(DatatoReg_EXE),.RegWrite_EX(RegWrite_EXE),.WR_EX(mem_w_EXE),
        .u_b_h_w_EX(u_b_h_w_EXE),.MIO_EX(MIO_EXE));

    MUX2T1_32
mux_A_EXE(.I0(rs1_data_EXE),.I1(PC_EXE),.s(ALUSrc_A_EXE),.o(ALUA_EXE));     //to
fill sth. in ()

    MUX2T1_32
mux_B_EXE(.I0(rs2_data_EXE),.I1(Imm_EXE),.s(ALUSrc_B_EXE),.o(ALUB_EXE));
//to fill sth. in ()

    ALU alu(.A(ALUA_EXE),.B(ALUB_EXE),.Control(ALUControl_EXE),
        .res(ALUout_EXE),.zero(ALUzero_EXE),.overflow(ALUoverflow_EXE));

   MUX2T1_32
mux_forward_EXE(.I0(rs2_data_EXE),.I1(Datain_MEM),.s(forward_ctrl_ls),.o(Dataout
_EXE));        //to fill sth. in ()


    // MEM
    REG_EX_MEM
reg_EXE_MEM(.clk(debug_clk),.rst(rst),.EN(reg_EM_EN),.flush(reg_EM_flush),

.IR_EX(inst_EXE),.PCurrent_EX(PC_EXE),.ALUO_EX(ALUout_EXE),.B_EX(Dataout_EXE),
        .rd_EX(rd_EXE),.DatatoReg_EX(DatatoReg_EXE),.RegWrite_EX(RegWrite_EXE),
        .WR_EX(mem_w_EXE),.u_b_h_w_EX(u_b_h_w_EXE),.MIO_EX(MIO_EXE),


.PCurrent_MEM(PC_MEM),.IR_MEM(inst_MEM),.ALUO_MEM(ALUout_MEM),.Datao_MEM(Dataout
_MEM),

.rd_MEM(rd_MEM),.DatatoReg_MEM(DatatoReg_MEM),.RegWrite_MEM(RegWrite_MEM),
        .WR_MEM(mem_w_MEM),.u_b_h_w_MEM(u_b_h_w_MEM),.MIO_MEM(MIO_MEM));

    RAM_B data_ram(.addra(ALUout_MEM),.clka(debug_clk),.dina(Dataout_MEM),
        .wea(mem_w_MEM),.douta(Datain_MEM),.mem_u_b_h_w(u_b_h_w_MEM));


    // WB
```

```
    REG_MEM_WB
reg_MEM_WB(.clk(debug_clk),.rst(rst),.EN(reg_MW_EN),.IR_MEM(inst_MEM),

.PCurrent_MEM(PC_MEM),.ALUO_MEM(ALUout_MEM),.Datai(Datain_MEM),.rd_MEM(rd_MEM),
        .DatatoReg_MEM(DatatoReg_MEM),.RegWrite_MEM(RegWrite_MEM),


.PCurrent_WB(PC_WB),.IR_WB(inst_WB),.ALUO_WB(ALUout_WB),.MDR_WB(Datain_WB),
        .rd_WB(rd_WB),.DatatoReg_WB(DatatoReg_WB),.RegWrite_WB(RegWrite_WB));

    MUX2T1_32
mux_WB(.IO(ALUout_WB),.I1(Datain_WB),.s(DatatoReg_WB),.o(wt_data_WB));


    wire [31:0] Test_signal;
    assign debug_data = debug_addr[5] ? Test_signal : Debug_regs;

    CPUTEST    U1_3(.PC_IF(PC_IF),
                    .PC_ID(PC_ID),
                    .PC_EXE(PC_EXE),
                    .PC_MEM(PC_MEM),
                    .PC_WB(PC_WB),
                    .PC_next_IF(next_PC_IF),
                    .PCJump(jump_PC_ID),
                    .inst_IF(inst_IF),
                    .inst_ID(inst_ID),
                    .inst_EXE(inst_EXE),
                    .inst_MEM(inst_MEM),
                    .inst_WB(inst_WB),
                    .PCEN(PC_EN_IF),
                    .Branch(Branch_ctrl),
                    .PCSource(Branch_ctrl),
                    .RS1DATA(rs1_data_reg),
                    .RS2DATA(rs2_data_reg),
                    .Imm32(Imm_out_ID),
                    .ImmSel(ImmSel_ctrl),
                    .ALUC(ALUControl_ctrl),
                    .ALUSrc_A(ALUSrc_A_ctrl),
                    .ALUSrc_B(ALUSrc_B_ctrl),
                    .A(ALUA_EXE),
                    .B(ALUB_EXE),
                    .ALU_out(ALUout_MEM),
                    .Datai(Datain_MEM),
                    .Datao(Dataout_MEM),
                    .Addr(Addr),
                    .WR(MWR),
                    .MIO(MIO_MEM),
                    .WDATA(wt_data_WB),
                    .DatatoReg(DatatoReg_WB),
                    .RegWrite(RegWrite_WB),
                    .data_hazard(reg_FD_stall),
                    .control_hazard(Branch_ctrl),

                    .Debug_addr(debug_addr[4:0]),
                    .Test_signal(Test_signal)
```

```
                    );

endmodule
```

Hazard detection

Forward:

一共有三种情况：
ALU+ALU（对应forward_1，rs1_ID == rd_EXE，将ALU在EXE的结果传到ID中）
ALU+任意+ALU(对应forward_2，rs1_ID == rd_MEM，将ALU在MEM的结果传到ID中)

*+Load类型(对应forward_3，rs1_ID == rd_MEM &&hazard_optype_MEM == 2'd2，将MEM的结果传给load去使用，不过要停拍)

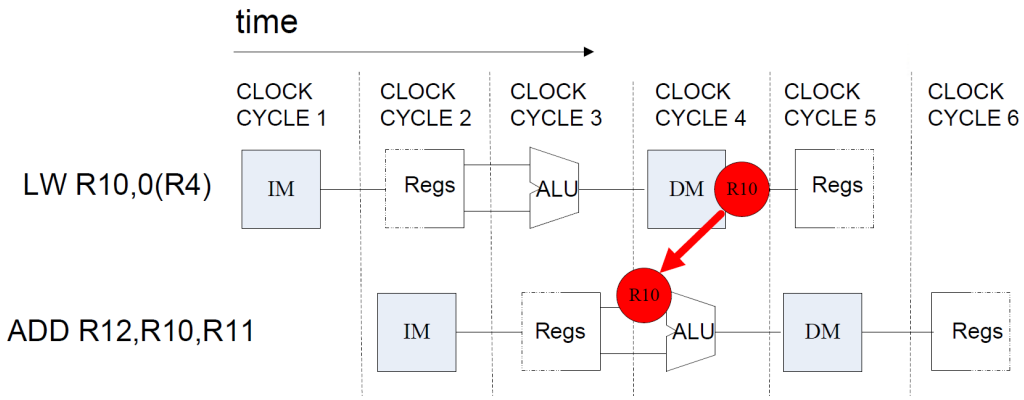
Load+Store类型（对应hazard_optype_MEM == 2'd2 && hazard_optype_EXE==2'd3 && rs2_EXE == rd_MEM时，把MEM的结果传递）

最后得到一个统一的forwarding号

stall

判断是否需要停顿，即判断是否为load+ALU的情况（EXE需要的内容和MEM在一起时，必须停一拍，和forward_3对应）

最后结合两个得到是否需要stall




```
`timescale 1ps/1ps

module HazardDetectionUnit(
    input clk,
    input Branch_ID, rs1use_ID, rs2use_ID,
    input[1:0] hazard_optype_ID,
    input[4:0] rd_EXE, rd_MEM, rs1_ID, rs2_ID, rs2_EXE,
    output PC_EN_IF, reg_FD_EN, reg_FD_stall, reg_FD_flush,
        reg_DE_EN, reg_DE_flush, reg_EM_EN, reg_EM_flush, reg_MW_EN,
    output forward_ctrl_ls,
    output[1:0] forward_ctrl_A, forward_ctrl_B
);
    //according to the diagram, design the Hazard Detection Unit
    reg[1:0] hazard_optype_EXE, hazard_optype_MEM;
    always @(posedge clk) begin
    hazard_optype_MEM <= hazard_optype_EXE;
    hazard_optype_EXE <= hazard_optype_ID & {2{~reg_DE_flush}};
```

```
    // ID-EXE之间的reg不需要flush的时候才将ID阶段的hazard_optype传给EXE阶段
    end
    // forward_A_1代表的是ALU+ALU出现hazard的时候的forwarding信号
    // 此时需要将ALU的结果在EXE阶段直接送到ID阶段选操作数的四选一MUX里
    wire forward_A_1 = rd_EXE && rs1use_ID && rs1_ID == rd_EXE &&
    hazard_optype_EXE == 2'd1;
    // forward_A_2代表的是ALU+*+ALU出现hazard的时候的forwarding信号
    // 此时需要将ALU的结果在MEM阶段直接送到ID阶段选操作数的四选一MUX里
    wire forward_A_2 = rd_MEM && rs1use_ID && rs1_ID == rd_MEM &&
    hazard_optype_MEM == 2'd1;
    // forward_A_3代表的是load+ALU出现hazard的时候的forwarding信号
    // 此时需要停顿一个周期，然后将MEM阶段读出的数据送到ID阶段选操作数的四选一MUX里
    wire forward_A_3 = rd_MEM && rs1use_ID && rs1_ID == rd_MEM &&
    hazard_optype_MEM == 2'd2;
    // A_stall是判断是否需要停顿的信号：即判断是否为load+ALU的情况

    wire A_stall = rd_EXE && rs1use_ID && rs1_ID == rd_EXE && hazard_optype_EXE
    == 2'd2 && hazard_optype_ID != 2'd3;

    //下面四条指令和上面四条道理完全相同
    wire forward_B_1 = rd_EXE && rs2use_ID && rs2_ID == rd_EXE &&
    hazard_optype_EXE == 2'd1;
    wire forward_B_2 = rd_MEM && rs2use_ID && rs2_ID == rd_MEM &&
    hazard_optype_MEM == 2'd1;
    wire forward_B_3 = rd_MEM && rs2use_ID && rs2_ID == rd_MEM &&
    hazard_optype_MEM == 2'd2;

    wire B_stall = rd_EXE && rs2use_ID && rs2_ID == rd_EXE && hazard_optype_EXE
    == 2'd2 && hazard_optype_ID != 2'd3;
    // forward_ctrl_ls是判断是否为load+store的信号，此时可以直接将从memory读出的数据
    //forward到EXE阶段，准备写入memory

    assign forward_ctrl_ls = hazard_optype_MEM == 2'd2 && hazard_optype_EXE
    ==2'd3 && rs2_EXE == rd_MEM;

    // forward_ctrl_A(forward_ctrl_B)是四选一mux的选择信号
    assign forward_ctrl_A = {2{forward_A_1}} & 2'd1 |
    {2{forward_A_2}} & 2'd2 |
    {2{forward_A_3}} & 2'd3;
    assign forward_ctrl_B = {2{forward_B_1}} & 2'd1 |
    {2{forward_B_2}} & 2'd2 |
    {2{forward_B_3}} & 2'd3;
    // 当需要停顿的时候(A_stall | B_stall == 1)
    assign reg_FD_stall = A_stall | B_stall;
    assign reg_DE_flush = A_stall | B_stall;
    // 当需要跳转的时候则需要将IF-ID阶段的reg flush掉
    assign reg_FD_flush = Branch_ID;
    // 不需要停顿的时候，则取指令的使能信号置1
    assign PC_EN_IF = ~(A_stall | B_stall);
    assign reg_FD_EN = 1'b1;
    assign reg_DE_EN = 1'b1;
    assign reg_EM_EN = 1'b1;
    assign reg_MW_EN = 1'b1;
endmodul
```

CPUctrl

这一块主要进行代码补全，分为以下几个部分

sb,lb等，这部分通过指令类型和func3部分完成

```
wire BEQ = Bop & funct3_0; //to fill sth.in
wire BNE = Bop & funct3_1; //to fill sth.in
wire BLT = Bop & funct3_4; //to fill sth.in
wire BGE = Bop & funct3_5; //to fill sth.in
wire BLTU = Bop & funct3_6; //to fill sth.in
wire BGEU = Bop & funct3_7; //to fill sth.in
wire LB = Lop & funct3_0; //to fill sth.in
wire LH = Lop & funct3_1; //to fill sth.in
wire LW = Lop & funct3_2; //to fill sth.in
wire LBU = Lop & funct3_4; //to fill sth.in
wire LHU = Lop & funct3_5; //to fill sth.in
wire SB = Sop & funct3_0; //to fill sth.in
wire SH = Sop & funct3_1; //to fill sth.in
wire SW = Sop & funct3_2; //to fill sth.in
```

LUI等，这一部分是对opcode的完善补全。

```
wire LUI = opcode == 7'b0110111; //to fill sth.in
wire AUIPC = opcode == 7'b0010111; //to fill sth.in
wire JAL = opcode == 7'b1101111;//to fill sth.in
assign JALR = (opcode == 7'b1100111) && funct3_0;//to fill sth.in
```

Branch跳转，这一块要么是满足branch的条件（B_valid）要么是JAL或者JALR

```
 assign Branch = B_valid & cmp_res | JAL | JALR; //to fill sth.in
```

EXE阶段ALU处的MUX信号，ALUSrc_A 是通过 0，1 来区分第一个操作数是 PC 还是 reg，AUIPC,JAL,JALR 需要在该阶段进行 pc 的运算.而对于ALUSrc_B ，是判断第二个操作数是否为立即数

```
assign ALUSrc_A = JAL | JALR | AUIPC; //to fill sth. in
assign ALUSrc_B = I_valid | L_valid | S_valid | LUI | AUIPC; //to fill sth. in
```

是否使用RS1,RS2操作数的寄存器读数判断

```
assign rs1use = R_valid | I_valid | S_valid | B_valid | L_valid | JALR;
assign rs2use = R_valid | S_valid | B_valid;
```

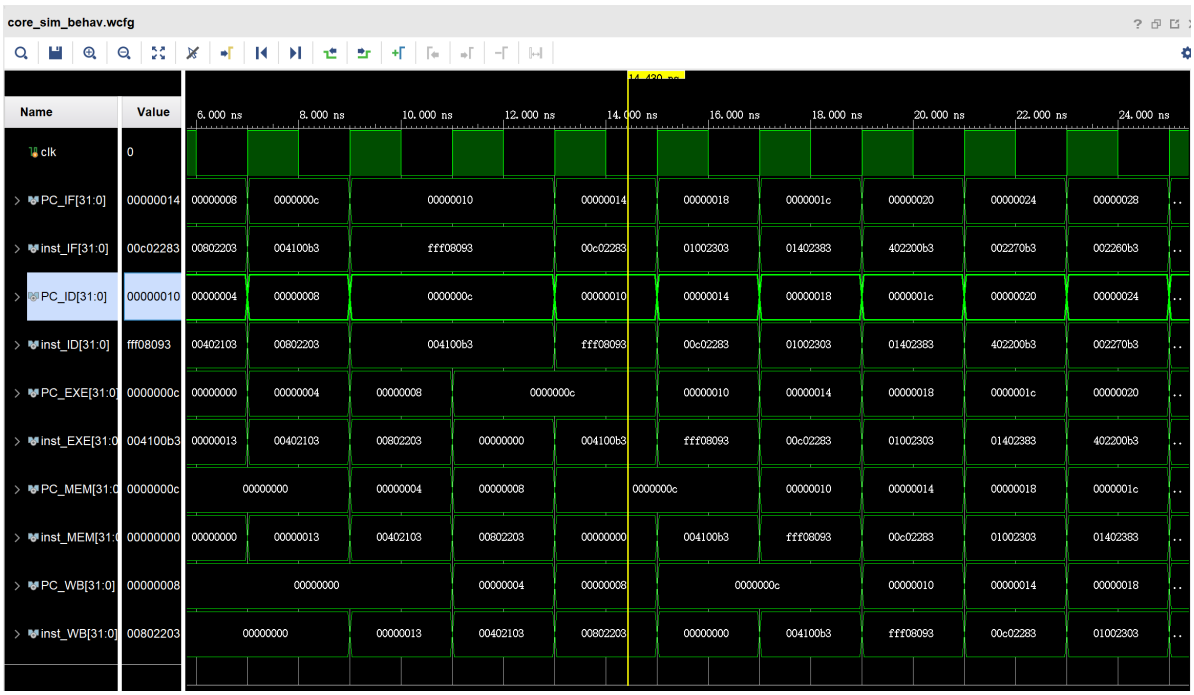
Hazard分为3种情况，一种是ALU的result在EXE的phase，一种是ALU result在MEM的phase，一种是Mem Data out，又在Memory Read后，在MEM的phase

```
assign hazard_optype = {2{R_valid | I_valid | JAL | JALR | LUI | AUIPC}}
& hazard_optype_ALU | // Others
{2{L_valid}} & hazard_optype_LOAD | // LOAD
{2{S_valid}} & hazard_optype_STORE; // STORE
```

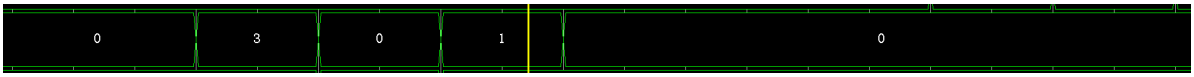
三、实验过程和数据记录

仿真验证

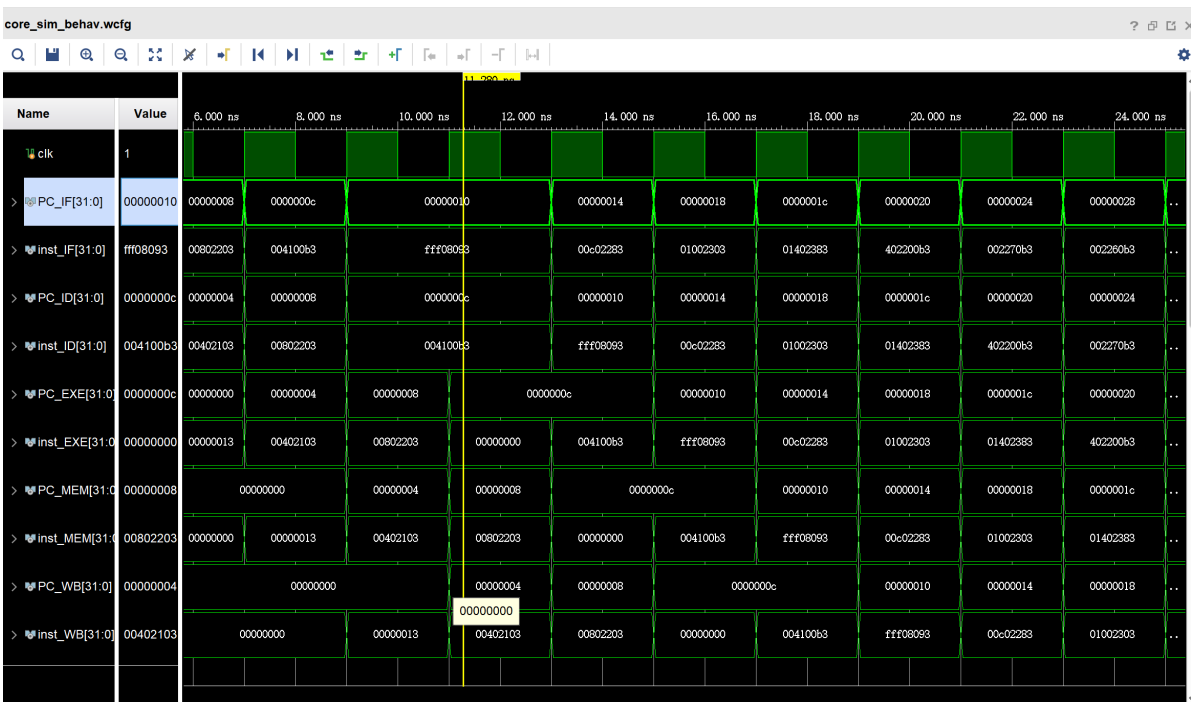
ALU+ALU的前置类



此时ID阶段的指令是addi x1, x1, -1。EXE阶段的指令是add x1, x2, x4。此时ALU计算出的x1的值应该forwarding到ID阶段的src_A。并且应该有forward被置1，如图，为1



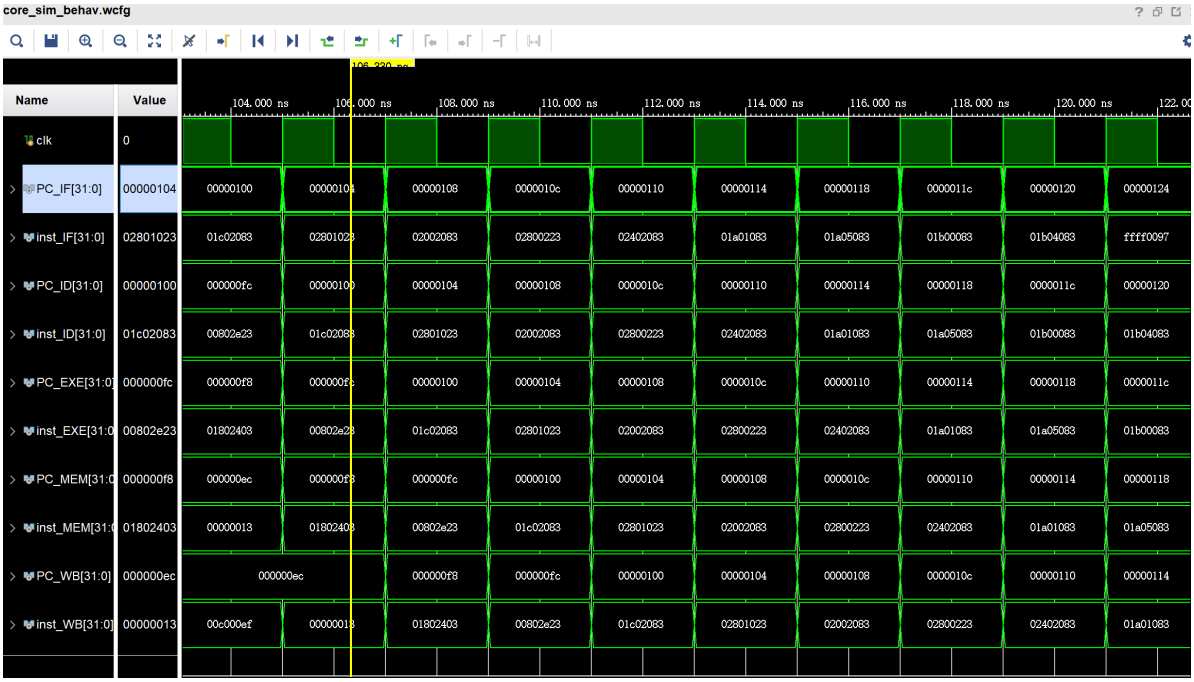
Load+ALU的停拍类



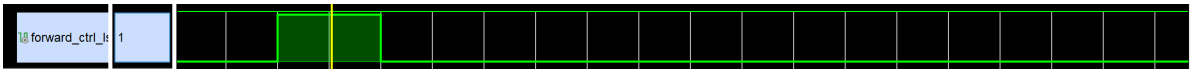
此时，接连着lw x4, 8(x0)与add x1, x2, x4两条指令，mem阶段无法给exe做forward，所以会停一拍

这时候对应的是forward_3的内容，对应也为3

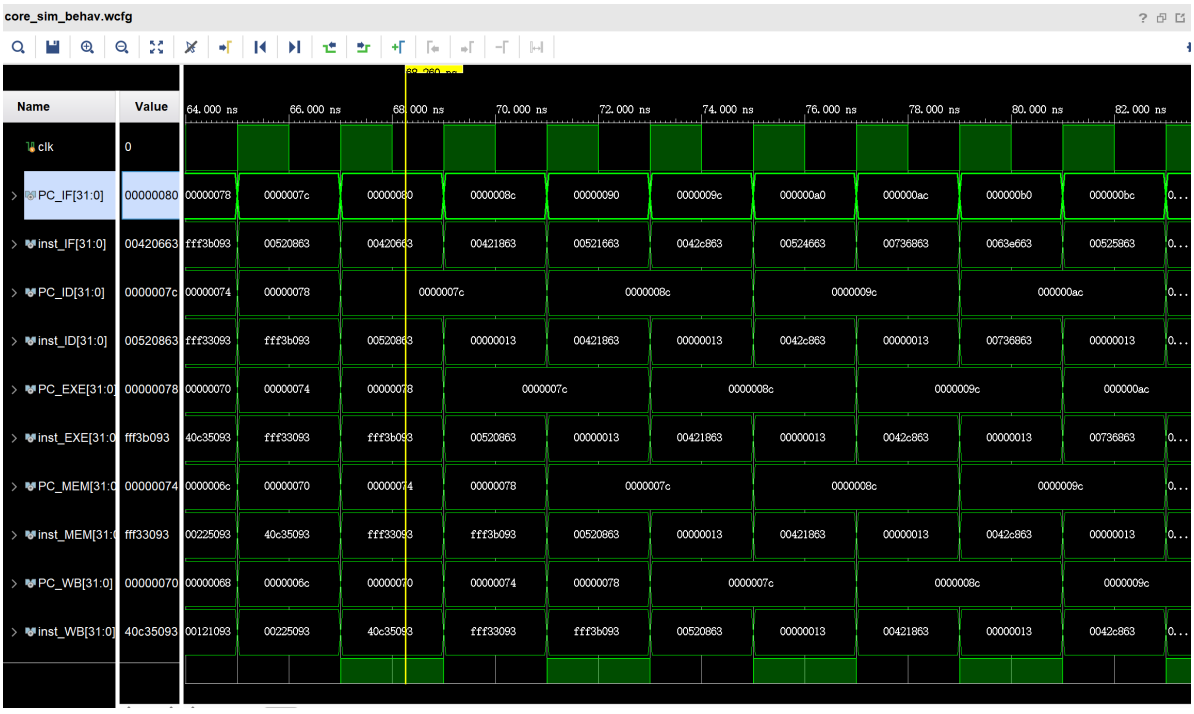
load+store



MEM阶段的指令是sw x8, 28(x0)。EXE阶段的指令是lw x8, 24(x0)。此时forward_ctrl_ls信号置1，选择了从memory中forwarding回来的值



Branch

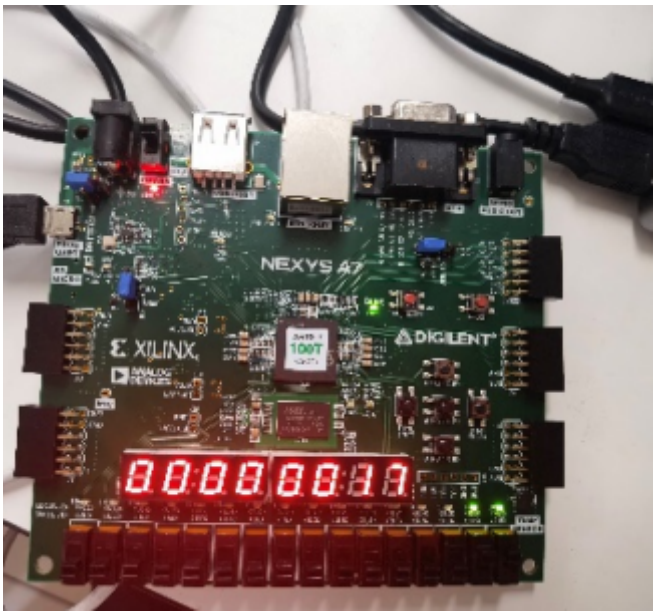


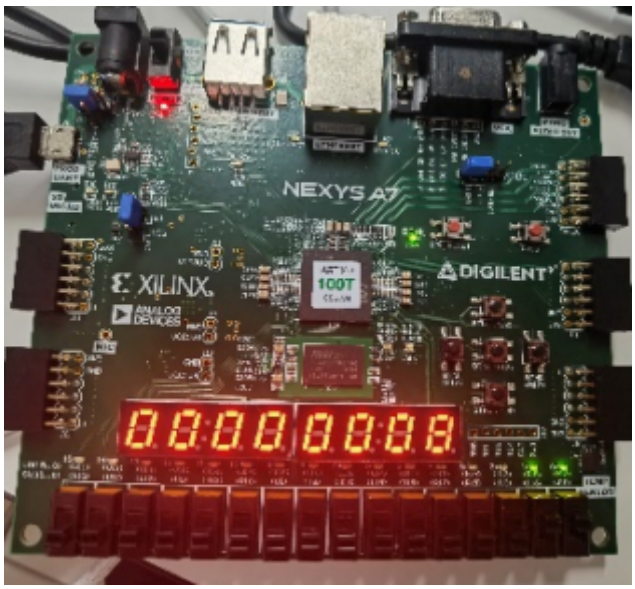
在黄圈部分ID阶段的指令是beq x4,x4,label0。此时需要跳转到0x8c，可以看到流水线stall了一个周期，fetch到了0x8c。

上板验证

经历以下的指令，观察R1寄存器结果，符合要求

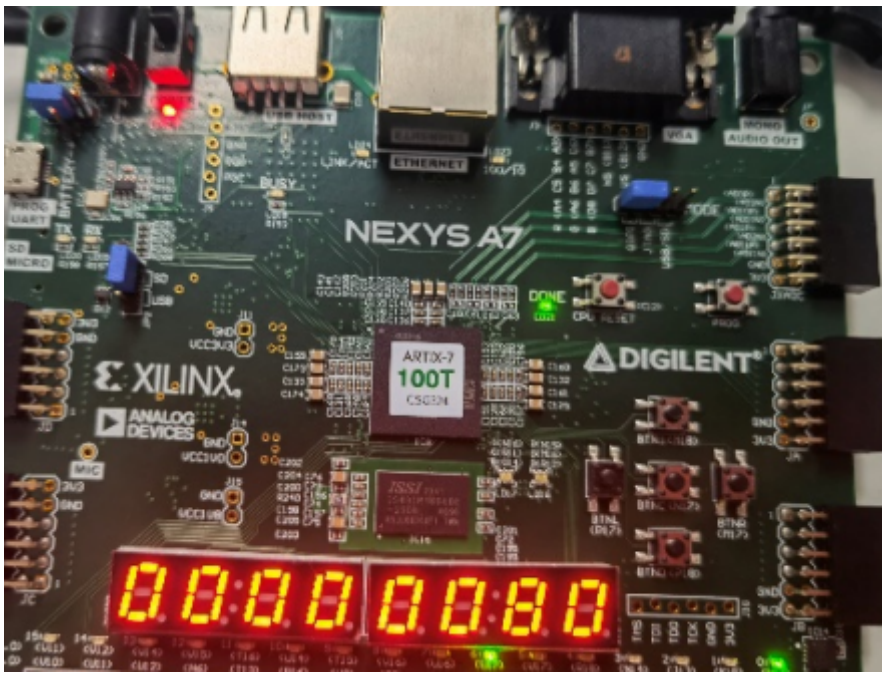
NO.	Instruction	Addr.	Label	ASM
0	00000013	0	__start:	addi x0, x0, 0
1	00402103	4		lw x2, 4(x0)
2	00802203	8		lw x4, 8(x0)
3	004100b3	C		add x1, x2, x4
4	fff08093	10		addi x1, x1, -1

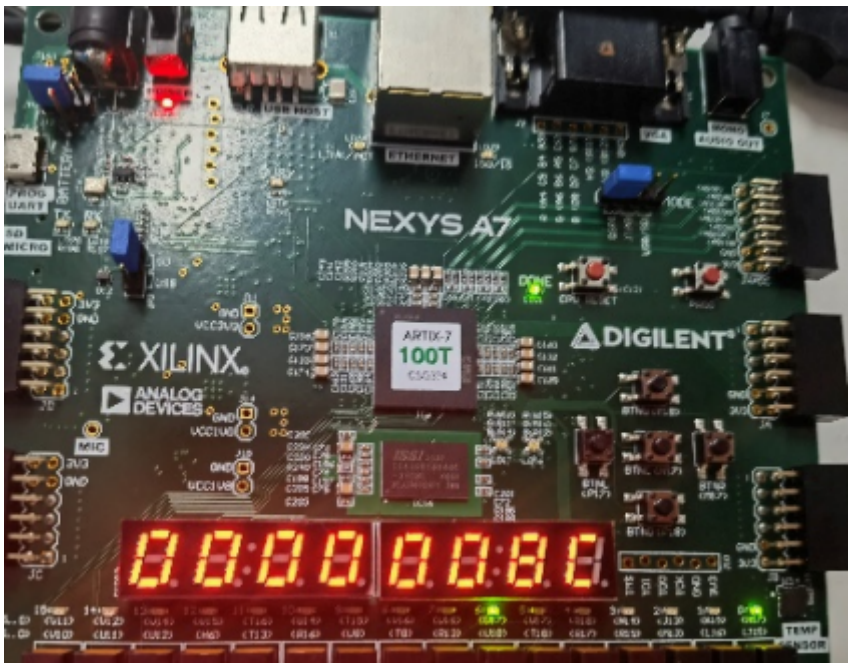




实现以下指令，测试branch的跳转,PC跳转8C，符合要求：

32	00420663	80		beq x4,x4,label0
33	00000013	84		addi x0,x0,0
34	00000013	88		addi x0,x0,0
35	00421863	8C	label0:	bne x4,x4,label1





四.实验心得

心得

这次的实验是对计组实验的回顾，因为我基础较差，在这次实验中自己写了基础的指令后，在一些模块还是碰到了不少的问题（麻烦助教学长了）。我在写代码的时候，主要的问题是体现在hazard_detection的部分，由于有ALU, Load和Store三种类型指令相关的Hazard，需要使用一个2位的控制信号输入到HazardDetection模块作为区分。另外，关于forward_ctrl_ls信号，我一开始不是很明白，后来问了同学才知道是Load+Store类型的Hazard，这里也感谢同学给了我一个别的班的ppt，照着图给我讲解。今年的验收和之前的验收确实有些不一样，感觉需要对自己提出更高的要求，我会在后续的学习中加强对实验的重视，力争能尽量多的去理解，去学习。

思考题部分

1.添加了forwarding机制后，是否观察到了stall延迟减少的情况？请在测试程序中给出forwarding机制起到实际作用的位置，并给出仿真图加以证明。

回答：加入 forwarding 后，stall 除了 load-used，control hazard，其他情况不会发生，可以减少 stall 的情况。图的话看仿真验证就可以证明

2.有没有办法避免注意事项中提到的由 Load 所导致得需要额外 stall 一个周期或者两个周期 这样的情况，即有没有办法做到只用 forwarding 解决 data hazard，不用额外的 stall 来 解决 data hazard。如果有，请说明方法和利弊；如果没有，请说明理由。

回答：无法避免。因为ld指令，在MEM阶段中才能从Memory中取数据，但是ld后面的R指令等ID阶段就使用，相差两个阶段无法实现前递。