浙江大學

本科实验报告

课程名称:	**计算机体系结构**
姓 名:	吴语若
学 院:	计算机科学与技术学院
专 业:	计算机科学与技术
学 号:	3220104111
指导教师:	常瑞

2024年10月5日

浙江大学实验报告

课程名称：计算机体系结构　　实验类型：综合

实验项目名称：流水线RISC-V CPU设计

学生姓名：吴语若　　专业：计算机科学与技术　学号：3220104111

指导老师：常瑞　　助教：秦嘉俊、钟梓航

实验地点：曹西-301　　实验日期：2024年10月5日

一、实验目的和要求

1. 温故流水线CPU设计
2. 了解并实现RV321指令集
3. 理解旁路优化

二、实验过程和原理

1. 根据流水线CPU原理图完成`src/lab1/core/RV32core.v`的代码填空，主要方式即参考给出的原理图进行连线：

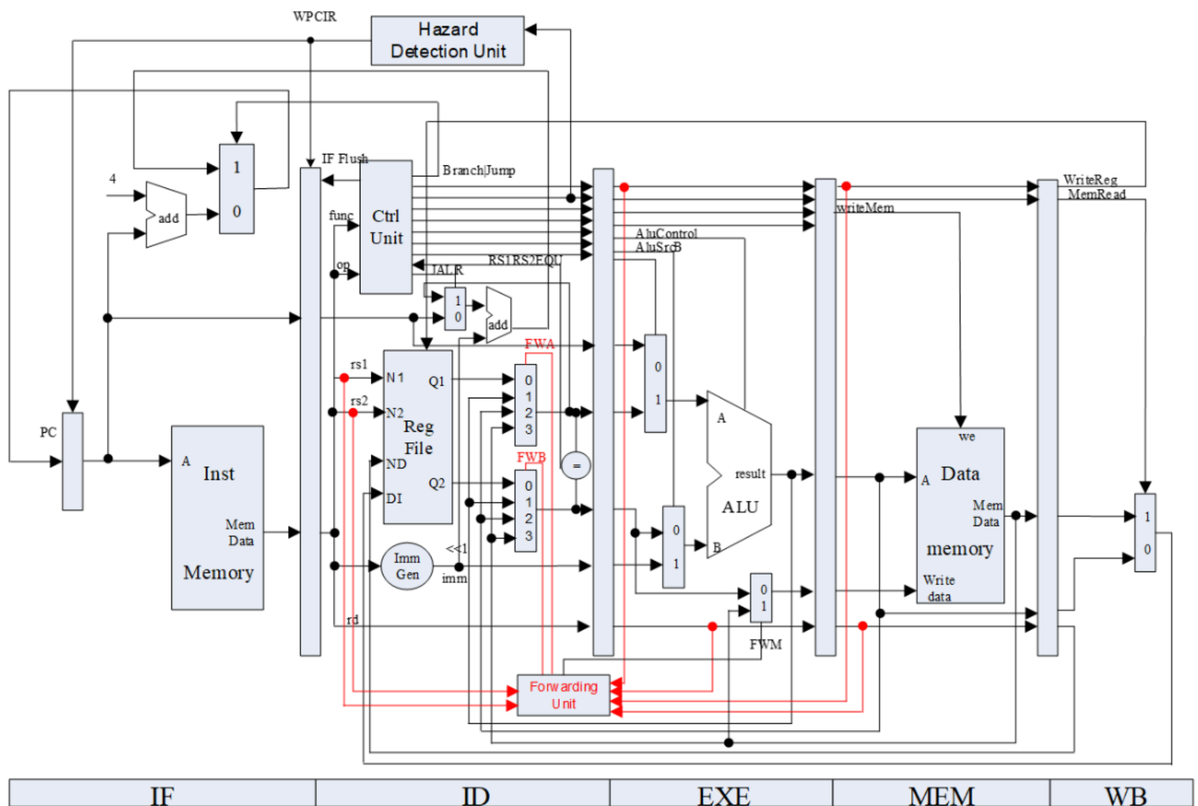

图1 流水线（带forwarding）CPU的基本实现原理

填空的代码如下，具体解释见注释：

```
// IF 阶段取值的选择，根据是否跳转选择取值的地址是跳转后的地址还是 PC + 4
MUX2T1_32 mux_IF(.I0(PC_4_IF),.I1(jump_PC_ID),.s(Branch_ctrl),.o(next_PC_IF));
// ID 阶段根据 Forwarding Unit 对是否发生前递的判断的控制信号，决定传递的操作数来源
MUX4T1_32 mux_forward_A(
.I0(rs1_data_reg),.I1(ALUout_EXE),.I2(ALUout_MEM),.I3(Datain_MEM),
.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),
.s(forward_ctrl_B),.o(rs2_data_ID));
// EX 阶段根据 ALU 的控制信号决定操作数的来源：第一个操作数来自PC地址还是第一个寄存器；第二个操作数来自第二个寄存器还是立即数
MUX2T1_32 mux_A_EXE(.I0(PC_EXE),.I1(rs1_data_EXE),.s(ALUSrc_A_EXE),.o(ALUA_EXE));
MUX2T1_32 mux_B_EXE(.I0(rs2_data_EXE),.I1(Imm_EXE),.s(ALUSrc_B_EXE),.o(ALUB_EXE));
// EX 阶段根据 Forwarding Unit 的 ls前递控制信号决定写入存储器的内容来源为rs2寄存器还是之前ld的数据
MUX2T1_32 mux_forward_EXE(
.I0(rs2_data_EXE),.I1(Datain_MEM),.s(forward_ctrl_ls),.o(Dataout_EXE));
```

2. 根据流水线CPU原理完成不同控制信号下比较器`cmp_32.v`的代码

   在整个CPU内，对`cmp_32`的调用如下：

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

   由此可以推测这个部件的作用为：比较a和b的大小，根据不同的指令对应的比较器控制信号`cmp_ctrl`输出不同的结果`cmp_res_ID`，完善后的代码如下：

```
`timescale 1ns / 1ps

module cmp_32(  input [31:0] a,
                input [31:0] b,
                input [2:0] ctrl,
                output c
    );
    parameter cmp_EQ  = 3'b001;
    parameter cmp_NE  = 3'b010;
    parameter cmp_LT  = 3'b011;
    parameter cmp_LTU = 3'b100;
    parameter cmp_GE  = 3'b101;
    parameter cmp_GEU = 3'b110;

    wire res_EQ  = a == b;
    wire res_NE  = ~res_EQ;
    wire res_LT  = (a[31] & ~b[31]) || (~(a[31] ^ b[31]) && a < b);
    wire res_LTU = a < b;
    wire res_GE  = ~res_LT;
    wire res_GEU = ~res_LTU;

    wire EQ  = ctrl == cmp_EQ ;
    wire NE  = ctrl == cmp_NE ;
    wire LT  = ctrl == cmp_LT ;
    wire LTU = ctrl == cmp_LTU;
    wire GE  = ctrl == cmp_GE ;
    wire GEU = ctrl == cmp_GEU;

    // 比较器输出当前指令对应的结果
    assign c = (EQ & res_EQ) | (NE & res_NE) | (LT & res_LT) |
               (LTU & res_LTU) | (GE & res_GE) | (GEU & res_GEU);
```

```
endmodule
```

3. 完善控制单元CtrlUnit.v代码

- 首先先要根据输入的func和op的值判断指令类型，由RV32I非特权指令布局可以写出对应的解码的代码：

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

图2 代码填空涉及的部分RV32I非特权指令布局

```
wire BEQ = Bop & funct3_0;
wire BNE = Bop & funct3_1;
wire BLT = Bop & funct3_4;
wire BGE = Bop & funct3_5;
wire BLTU = Bop & funct3_6;
wire BGEU = Bop & funct3_7;

wire LB =  Lop & funct3_0;
wire LH =  Lop & funct3_1;
wire LW =  Lop & funct3_2;
wire LBU = Lop & funct3_4;
wire LHU = Lop & funct3_5;

wire SB = Sop & funct3_0;
wire SH = Sop & funct3_1;
wire SW = Sop & funct3_2;

wire LUI   = opcode == 7'b0110111;
wire AUIPC = opcode == 7'b0010111;

wire JAL  = opcode == 7'b1101111;
assign JALR = opcode == 7'b1100111;
```

- 根据不同指令类型以及比较器的结果cmp_res输出对应的控制信号，具体判断方式见代码注释：

```
parameter cmp_EQ  = 3'b001;
parameter cmp_NE  = 3'b010;
parameter cmp_LT  = 3'b011;
parameter cmp_LTU = 3'b100;
parameter cmp_GE  = 3'b101;
parameter cmp_GEU = 3'b110;
```

图3 cmp_32.v内不同指令类型对应的控制信号cmp_ctrl的值

```
// 跳转控制信号为1条件：是B型指令且满足跳转条件(cmp_res == 1)或者是J型指令
assign Branch = (B_valid & cmp_res) | JAL | JALR;

// 根据比较器cmp_32.v判断具体指令的代码给cmp_ctrl赋值
parameter cmp_EQ  = 3'b001;
parameter cmp_NE  = 3'b010;
parameter cmp_LT  = 3'b011;
parameter cmp_LTU = 3'b100;
parameter cmp_GE  = 3'b101;
parameter cmp_GEU = 3'b110;

assign cmp_ctrl = (BEQ ? cmp_EQ : 3'b000) | (BNE ? cmp_NE : 3'b000) |
                  (BLT ? cmp_LT : 3'b000) | (BGE ? cmp_GE : 3'b000) |
                  (BLTU ? cmp_LTU : 3'b000) | (BGEU ? cmp_GEU : 3'b000);

// 根据不同指令判断ALU操作数来源
// ALUSrc_A == 1 - rs1, ALUSrc_A == 0 - PC，只有在JAL、JALR和AUIPC指令下取PC
assign ALUSrc_A = ~(JAL | AUIPC | JALR);
// ALUSrc_B == 1 - Imm, ALUSrc_B == 0 - rs2
// 第二个操作数取立即数的指令有：I型指令，L型指令，S型指令，AUIPC, LUI
assign ALUSrc_B = I_valid | L_valid | S_valid | AUIPC | LUI;

// 根据指令类型，判断寄存器是否读取
assign rs1use =  ~(JAL | LUI | AUIPC);
assign rs2use = R_valid | B_valid | S_valid;
```

- 判断可能的冲突类型

  在五级流水线CPU中，可能的冲突情况有以下几种：

  1. 前一条指令的目标寄存器是下一条指令的源寄存器
  2. 前一条指令的目标寄存器是隔一条指令的源寄存器
  3. `ld`指令读出的结果是下一条`sd`指令写进的数据

  为了判断可能的冲突类型，我们需要知道当前的指令类型是`ld`还是`sd`还是ALU处（除了B型指令不使用ALU以及`ld`、`sd`指令计算的是固定的地址，其他源寄存器均有可能来自上一条指令的目标寄存器）可能发生冲突的指令，再将这个控制信号传入`HazardDetectionUnit.v`进行深入判断。代码如下：

```
// 判断可能的冲突类型
parameter ALU_hazard = 2'b01;
parameter ld_hazard = 2'b10;
parameter sd_hazard = 2'b11;
assign hazard_optype = (L_valid ? ld_hazard : 2'b00) | (S_valid ? sd_hazard : 2'b00) |
                       ((R_valid | I_valid | JAL | JALR | LUI | AUIPC) ?
                       ALU_hazard : 2'b00);
```

4. 完成`HazardDetectionUnit.v`

根据上述提到的ID段Forwarding基本实现原理图，这一模块输出的变量含义分别为：

- `forward_ctrl_ls` - `EX`阶段执行`sd`且`MEM`阶段执行`ld`且`ld`指令读出的结果是`sd`指令写进的数据，则需要将`forward_ctrl_ls`置1，控制`MEM`阶段读出的结果前递至`EX`阶段要`sd`的`data`;
- `forward_ctrl_A` / `forward_ctrl_B`
  - `2'b00` - 表示没有前递，正常取出来自`rs1` / `rs2`寄存器的值即可
  - `2'b01` - 将`EX`段ALU的计算结果前递至`rs1` / `rs2`寄存器的值，表示在`EX`阶段执行的指令（上一条指令）的目标寄存器是在`ID`阶段执行的指令的源寄存器（`rs1` / `rs2`）
  - `2'b10` - 将`MEM`段ALU的计算结果前递至`rs1` / `rs2`寄存器的值，表示在`MEM`阶段执行的指令（上上条指令）的目标寄存器是在`ID`阶段执行的指令的源寄存器（`rs1` / `rs2`）

    - `2'b11` - 将MEM段存储器读取的结果前递至rs1 / rs2寄存器的值，表示在EX/MEM阶段执行的ld指令的目标寄存器是在ID阶段执行的指令的源寄存器（rs1 / rs2）。**因此如果ID阶段执行的不是sd指令且上一条指令是ld且ld的目标寄存器是ID阶段指令的源寄存器，则需要stall一个时钟周期。**
    - 上述指令均在涉及rd寄存器的时候才需要forwarding，因此都需要添加rd寄存器的非零判断（被使用）条件。
- `reg_FD_EN` / `reg_DE_EN` / `reg_EM_EN` / `reg_MW_EN` - 流水线各级间寄存器使能信号，置1
- `PC_EN_IF` - 在不需要stall的时候置1，取新的PC地址对应的指令
- `reg_FD_stall` - 在stall的时候置1
- `reg_FD_flush` / `reg_DE_flush` / `reg_EM_flush` - flush为该级间寄存器控制竞争清除并等待的控制信号，为1的时候需要把寄存器之前的内容清除并等待新的内容；IDEX寄存器在stall的时候等待，IFID寄存器在跳转发生(`Branch_ID == 1'b1`)的时候等待(采取predict-not-taken策略)，EXMEM寄存器不需要等待

由此，我们在判断输出的时候需要使用寄存器来记录前两条指令的竞争类型`hazard_optype`，并且每次都在时钟正边沿更新：

```
parameter ALU_hazard = 2'b01;
parameter ld_hazard = 2'b10;
parameter sd_hazard = 2'b11;

// 记录竞争类型的寄存器
reg[1:0] hazard_optype_EX;
reg[1:0] hazard_optype_MEM;
always @(posedge clk) begin
    hazard_optype_MEM <= reg_EM_flush ? 2'b00 : hazard_optype_EX;
    hazard_optype_EX <= reg_DE_flush ? 2'b00 : hazard_optype_ID;
end
```

接下来我们需要根据不同阶段和寄存器的状态判断不同情况下forward及stall（根据上述的输出分析）：

```
// 判断stall: ID阶段执行的不是sd指令且上一条指令是ld且ld的目标寄存器是ID阶段指令的源寄存器
assign reg_FD_stall = (hazard_optype_ID != sd_hazard) & (hazard_optype_EX == ld_hazard) &
                      (rs1use_ID & rd_EXE == rs1_ID | rs2use_ID & rd_EXE == rs2_ID);
// 根据stall情况为部分控制信号赋值
assign PC_EN_IF = ~reg_FD_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;
assign reg_DE_flush = reg_FD_stall;
assign reg_FD_flush = Branch_ID;
assign reg_EM_flush = 1'b0;

// 判断forward
// forward_ctrl_ls: EX阶段执行sd且MEM阶段执行ld且ld指令读出的结果是sd指令写进的数据
assign forward_ctrl_ls = (hazard_optype_EX == sd_hazard) & (hazard_optype_MEM ==
ld_hazard) &
                         (rs2_EXE == rd_MEM);
// rs1的forward
assign forward_ctrl_A = ((rd_EXE && rs1use_ID && rd_EXE == rs1_ID && hazard_optype_EX ==
ALU_hazard) ? 2'b01 : 2'b00 ) | // 在EX阶段执行的指令的目标寄存器是在ID阶段执行的指令的源寄存器
rs1
                        ((!(rd_EXE && rs1use_ID && rd_EXE == rs1_ID && hazard_optype_EX ==
ALU_hazard)) && (rd_MEM && rs1use_ID && rd_MEM == rs1_ID && hazard_optype_MEM ==
ALU_hazard) ? 2'b10 : 2'b00 ) | // 在MEM阶段执行的指令的目标寄存器是在ID阶段执行的指令的源寄存
器rs1
```

```
                        ((!(rd_EXE && rs1use_ID && rd_EXE == rs1_ID && hazard_optype_EX ==
ALU_hazard)) && (hazard_optype_MEM == ld_hazard && rs1use_ID && rd_MEM == rs1_ID) ? 2'b11
: 2'b00 );  // 在MEM阶段执行的ld指令的目标寄存器是在ID阶段执行的指令的源寄存器rs1

// rs2的forward
assign forward_ctrl_B = ((rd_EXE && rs2use_ID && rd_EXE == rs2_ID && hazard_optype_EX ==
ALU_hazard) ? 2'b01 : 2'b00 ) | // 在EX阶段执行的指令的目标寄存器是在ID阶段执行的指令的源寄存器
rs2
                        ((!(rd_EXE && rs1use_ID && rd_EXE == rs1_ID && hazard_optype_EX ==
ALU_hazard)) && (rd_MEM && rs2use_ID && rd_MEM == rs2_ID && hazard_optype_MEM ==
ALU_hazard) ? 2'b10 : 2'b00 ) | // 在MEM阶段执行的指令的目标寄存器是在ID阶段执行的指令的源寄存
器rs2
                        ((!(rd_EXE && rs1use_ID && rd_EXE == rs1_ID && hazard_optype_EX ==
ALU_hazard)) && (hazard_optype_MEM == ld_hazard && rs2use_ID && rd_MEM == rs2_ID) ? 2'b11
: 2'b00 );  // 在MEM阶段执行的ld指令的目标寄存器是在ID阶段执行的指令的源寄存器rs2
```

5. 进行仿真测试和上板验证（见实验结果分析）

四、实验结果分析

1. **仿真测试**

- forward机制以及其减少stall延迟的验证

```
lw x2, 4(x0) # PC = 0x4, x2 = 0x00000008
lw x4, 8(x0) # PC = 0x8, x4 = 0x00000010
add x1, x2, x4 # PC = 0xC, x1 = 0x00000018
addi x1, x1, -1 # PC = 0x10, x1 = 0x00000017
```

如仿真图所示，在这段代码执行的过程中，发生了三处forward。

1. 代码第一行和代码第三行：在MEM阶段执行的ld指令的目标寄存器x2是在ID阶段执行的指令的源寄存器rs1 = x2，forward_ctrl_A = 3，前递存储器读取出来的data，不需要stall，减少了第一条指令WB后第三条指令再ID的1个时钟周期的stall。
2. 代码第二行和代码第三行：在EX阶段执行的ld指令的目标寄存器x4是在ID阶段执行的指令的源寄存器rs2 = x4，forward_ctrl_B = 3，stall一个时钟周期后前递存储器读取出来的data，原本第二条指令WB后第三条指令才能ID，需要stall2个时钟周期，减少了1个时钟周期的stall。
3. 代码第三行和代码第四行：在EX阶段执行的ALU指令的目标寄存器x1是在ID阶段执行的指令的源寄存器rs1 = x1，forward_ctrl_A = 1，前递ALU的计算结果，不需要stall，减少了第三条指令WB后第四条指令再ID的1个时钟周期的stall。

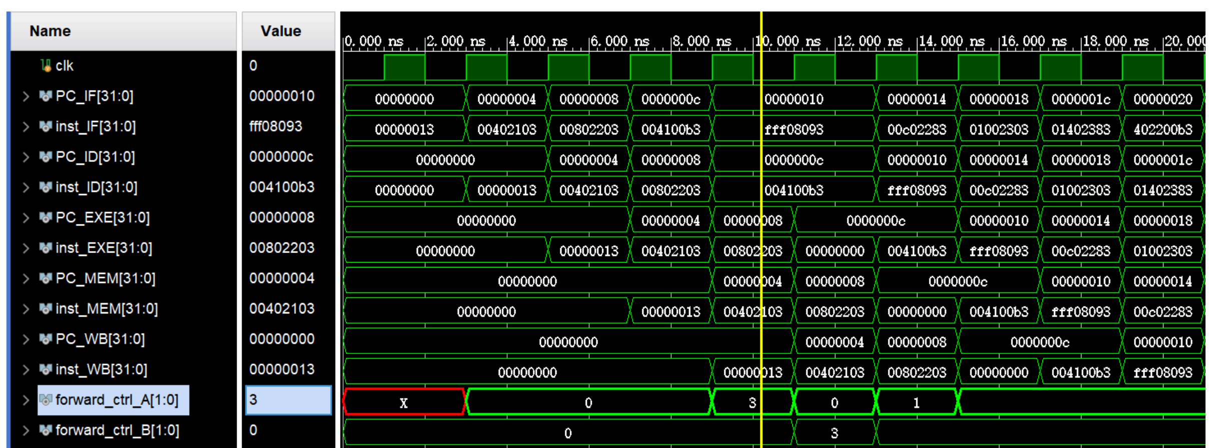

图4 forward_ctrl_A、forward_ctrl_B起作用的仿真

```
lw   x8, 24(x0) # PC = 0xF8, x8 = 0xFF000F0F
sw   x8, 28(x0) # PC = 0xFC
```

如仿真图所示，在这段代码执行的过程中，发生了EX阶段执行sw且MEM阶段执行lw且lw指令读出的结果是sw指令写进的数据的冲突，所以`forward_ctrl_ls = 1`，MEM阶段读出的结果前递至EX阶段要sd的data，减少了等待ld的指令WB后再执行sd指令ID阶段读取寄存器的2个时钟周期的stall。

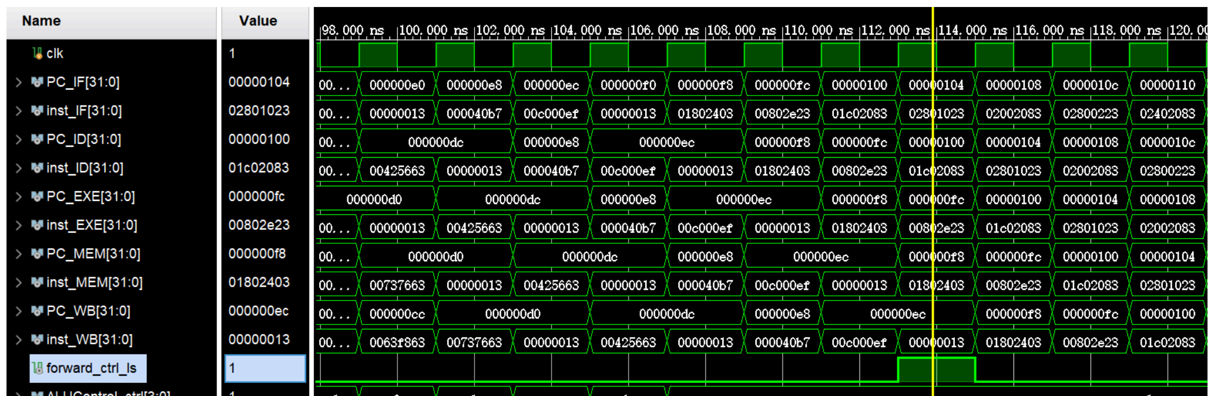

图5 forward_ctrl_ls起作用的仿真

- predict-not-taken跳转机制

```
beq  x4,x4,label0 # PC = 0x80 branch
addi x0,x0,0 # PC = 0x84
addi x0,x0,0 # PC = 0x88

label0:
bne  x4,x4,label1 # PC = 0x8C not branch
bne  x4,x5,label1 # PC = 0x90 branch
addi x0,x0,0 # PC = 0x94
addi x0,x0,0 # PC = 0x98

label1:
blt  x5,x4,label2 # PC = 0x9C not branch
blt  x4,x5,label2 # PC = 0xA0 branch
addi x0,x0,0 # PC = 0xA4
addi x0,x0,0 # PC = 0xA8

label2:
bltu x6,x7,label3 # PC = 0xAC not branch
bltu x7,x6,label3 # PC = 0xB0 branch
addi x0,x0,0 # PC = 0xB4
addi x0,x0,0 # PC = 0xB8

label3:
bge x4,x5,label4 # PC = 0xBC not branch
bge x5,x4,label4 # PC = 0xC0 branch
addi x0,x0,0 # PC = 0xC4
addi x0,x0,0 # PC = 0xC8

label4:
bgeu x7,x6,label5 # PC = 0xCC not branch
bgeu x6,x7,label5 # PC = 0xD0 branch
addi x0,x0,0 # PC = 0xD4
addi x0,x0,0 # PC = 0xD8
```

从仿真图中可以看出，当跳转条件不满足的时候（例如`PC = 0x8C`），不产生stall，正常执行下一条指令；当跳转条件满足发生跳转的时候（例如`PC = 0x90`），`Branch_ctrl = 1`，产生一个时钟周期的stall（注入一个NOP），之后执行跳转后的指令。仿真符合跳转逻辑的预期。

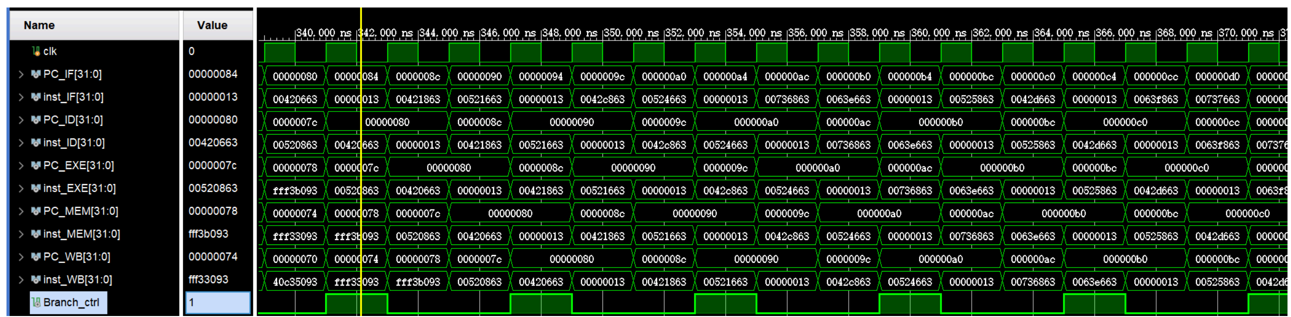

图6 B型指令predict-not-taken跳转机制验证

```
jalr x1,0(x0) # PC = 0x128, x1 = 0x12C
```

从仿真图可以看出，和B型指令一样，JALR指令的跳转也在插入一个NOP后正常执行，（只不过不需要判断直接`Branch_ctrl = 1`），符合跳转逻辑的预期。

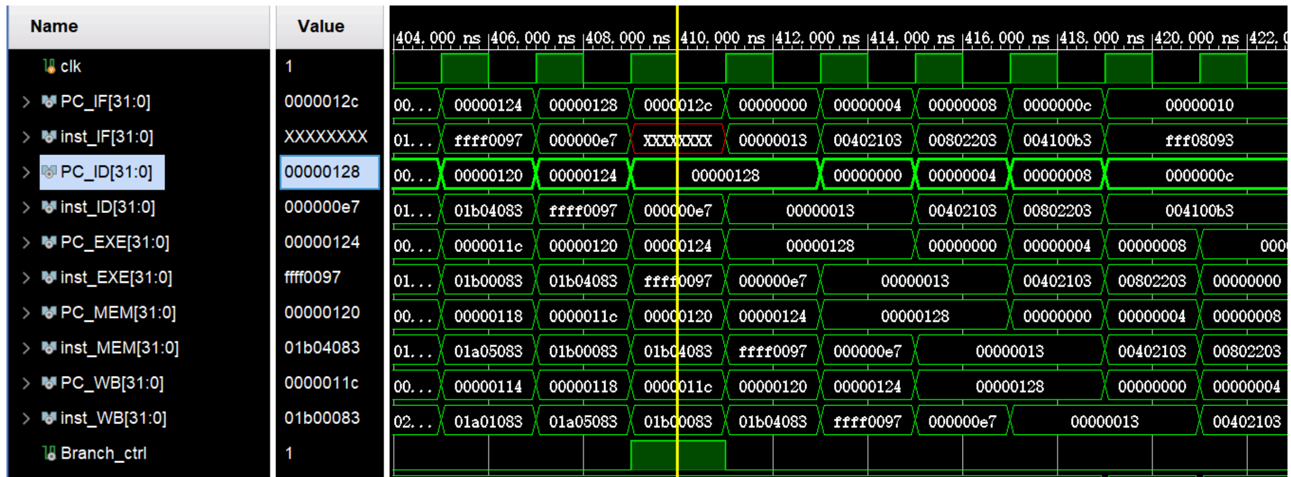

图7 JALR指令perdict-not-taken跳转机制验证

2. **上板验证**

   上板后验证指令执行顺序与寄存器的值，发现每一步都与`all_test.S`的注释相符合，所有寄存器结果正确且`forward`机制正常起作用。（由于篇幅限制，这里贴出最后两个时钟周期的寄存器的值的截图，以及采取的是predict-not-taken策略，所以在执行`jalr`指令的时候`stall`了一个时钟周期，故两个时钟周期的`PC_WB`的值都为`0x128`）

```
x01=0xFFFF0124
x02=0x00000008
x03=0x00000000
x04=0x00000010
x05=0x00000014
x06=0xFFFF0000
x07=0x0FFF0000
x08=0xFF000F0F
x09=0x00000000
x10=0x00000000
x11=0x00000000
x12=0x00000000
x13=0x00000000
x14=0x00000000
x15=0x00000000
x16=0x00000000
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x00000000
x26=0x00000000
x27=0x00000000
x28=0x00000000
x29=0x00000000
x30=0x00000000
x31=0x00000000
PC_WB  =0x00000128
INST   =0x000000E7
MEMADDR=0x00000000
MEMDATA=0x00000001
```

```
x01=0x0000012C
x02=0x00000008
x03=0x00000000
x04=0x00000010
x05=0x00000014
x06=0xFFFF0000
x07=0x0FFF0000
x08=0xFF000F0F
x09=0x00000000
x10=0x00000000
x11=0x00000000
x12=0x00000000
x13=0x00000000
x14=0x00000000
x15=0x00000000
x16=0x00000000
x17=0x00000000
x18=0x00000000
x19=0x00000000
x20=0x00000000
x21=0x00000000
x22=0x00000000
x23=0x00000000
x24=0x00000000
x25=0x00000000
x26=0x00000000
x27=0x00000000
x28=0x00000000
x29=0x00000000
x30=0x00000000
x31=0x00000000
PC_WB  =0x00000128
INST   =0x00000013
MEMADDR=0x00000000
MEMDATA=0x00000001
```

图8、9 寄存器值符合预期

五、思考题

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。（只需要贴出一次 Forwarding 机制起效的仿真图片即可）
   - 见实验结果分析 - 仿真测试 - forward机制以及其减少stall延迟的验证。
2. 在我们的框架中，比较器 `cmp_32` 处于 ID 段。请说明比较器在 ID 对比比较器在 EX 的优劣。（提示：可以从时延的角度考虑）
   - 首先，从时延角度，就本次实验的框架而言，遇到跳转指令的情况下，如果比较器在ID阶段会带来一个时钟周期的stall，而如果比较器在EX阶段则只有跳转指令到达EX阶段的时候才会出是否跳转的结果，需要两个时钟周期的stall，跳转指令部分会增加一个时钟周期的时延；而就算是采用其他的策略，在预测没有hit的时候，EX阶段比较器都会比ID阶段比较器多一个时钟周期的时延。
   - 其次，从硬件复杂度角度，EX阶段本身就有可以用于比较的ALU部件，将比较器放EX阶段可以降低硬件复杂度的同时提升部件的利用率，而如果将比较器放在ID阶段则会需要额外的比较器，相比之下硬件复杂度较高。

六、讨论与心得

1. 在`RV32core.v`代码填空的过程中由于整体原理图比较复杂，变量名很多又与原理图标注的不一样，然后框架上还有一些变量名和语义不匹配的情况（比如下图的in和out是不是反了），在写代码和debug的时候要格外细心否则很容易出错。

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

图10 RV32core.v部分代码

2. 在最开始的时候我的forward逻辑直接对三种情况取了或，没有考虑两种情况都满足的优先级和覆盖的情况，导致在仿真过程中`forward_ctrl_A`的值不符合预期。因为EX阶段的前递一定是优先于MEM阶段的，且这两种情况都符合的时候（最开始的仿真图第二个`forward_ctrl_A = 3`的时候）由于采取了或运算会导致MEM阶段的前递控制信号覆盖了EX阶段的前递控制信号，因此应该在判断MEM前递的条件中加一条不发生EX前递。修

改后仿真结果正常。

```
// rs1的forward
assign forward_ctrl_A = ((rd_EXE && rs1use_ID && rd_EXE == rs1_ID && hazard_optype_EX == ALU_hazard) ? 2'b01 : 2'b00 )
                        ((rd_MEM && rs1use_ID && rd_MEM == rs1_ID && hazard_optype_MEM == ALU_hazard) ? 2'b10 : 2'b00 )
                        ((hazard_optype_MEM == ld_hazard && rs1use_ID && rd_MEM == rs1_ID) ? 2'b11 : 2'b00 );  // 在MEM
```

图11 修改前的代码

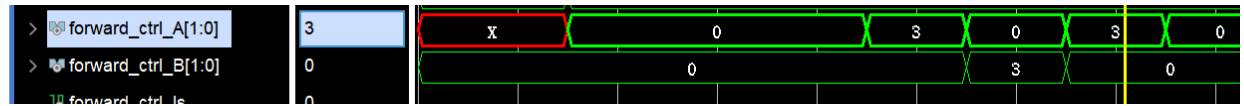

图12 修改前的仿真

```
assign forward_ctrl_A = ((rd_EXE && rs1use_ID && rd_EXE == rs1_ID && hazard_optype_EX == ALU_hazard) ? 2'b01 : 2'b00 ) |
                        ((!(rd_EXE && rs1use_ID && rd_EXE == rs1_ID && hazard_optype_EX == ALU_hazard)) && (rd_MEM && rs1use_ID && rd_MEM == rs1_ID && hazard_optype_MEM == ALU_hazard) ? 2'b10 : 2'b00 ) |
                        ((!(rd_EXE && rs1use_ID && rd_EXE == rs1_ID && hazard_optype_EX == ALU_hazard)) && (hazard_optype_MEM == ld_hazard && rs1use_ID && rd_MEM == rs1_ID) ? 2'b11 : 2'b00 );
```

图13 修改后的代码

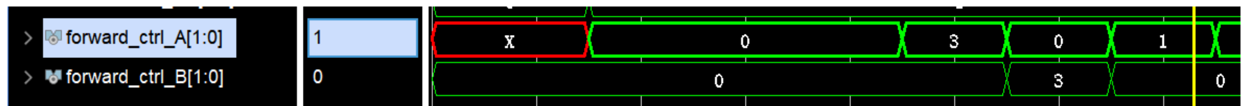

图14 修改后的仿真

3. 本次实验复习了计组的流水线CPU，并且巩固了forward和stall这些当时并没有学得特别明白的知识点。和计组实验不同，这次实验是在给出的代码框架上写一些填空，刚开始的时候面对比较大的工程有点无从下手，但是后来和实验文档的原理图的各个模块一一对应之后也就适应了填空这种方式。ps. 用VScode找一个变量都在哪里出现真的方便好多。
4. 我在做这个实验最开始的时候理解起来会有些别扭，是因为计组的前递是在EX阶段实现的，而实验文档和给出的框架的前递是在ID阶段实现的，虽然从原理上讲两个地方都可以，但是我最开始理解这个前递方式的时候有“ID阶段需要数据的时候EX阶段的数据是否已经准备好了，是否需要额外stall一个时钟周期”的困惑，后来经过思考和查证发现，EX阶段的计算在这个时钟周期的前半段，而ID阶段获取EX阶段的数据并写入IDEX寄存器发生在这个时钟周期的后半段，所以应该是可以顺利完成前递不需要stall的。现在从实现效果上来看，这两个前递方式应该是等价的（吧。

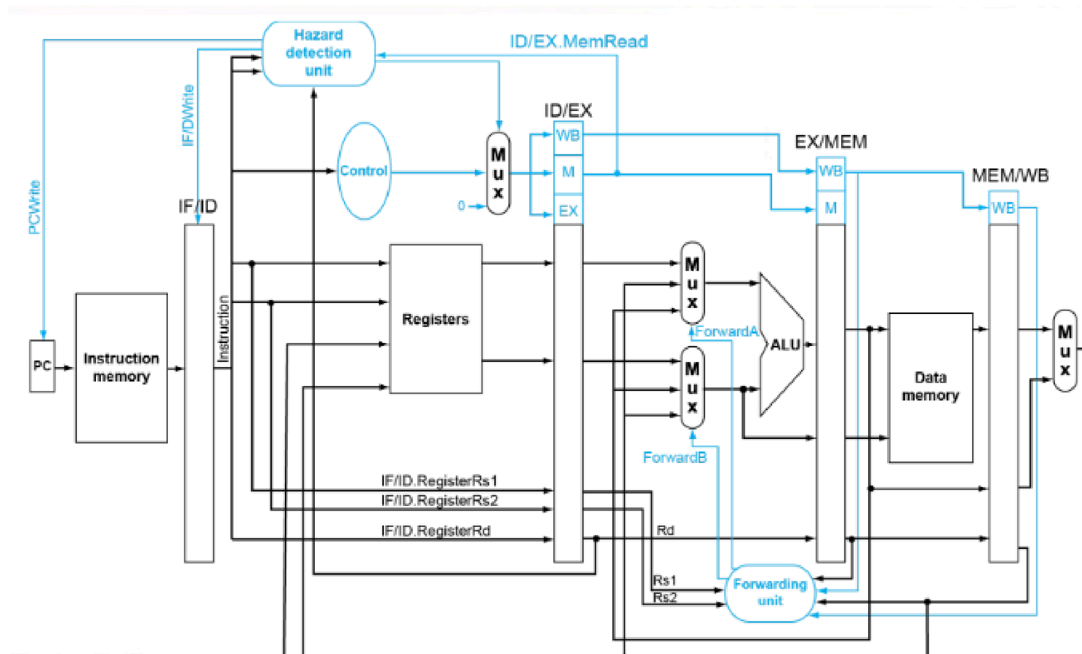

图15 计组的前递在EX阶段

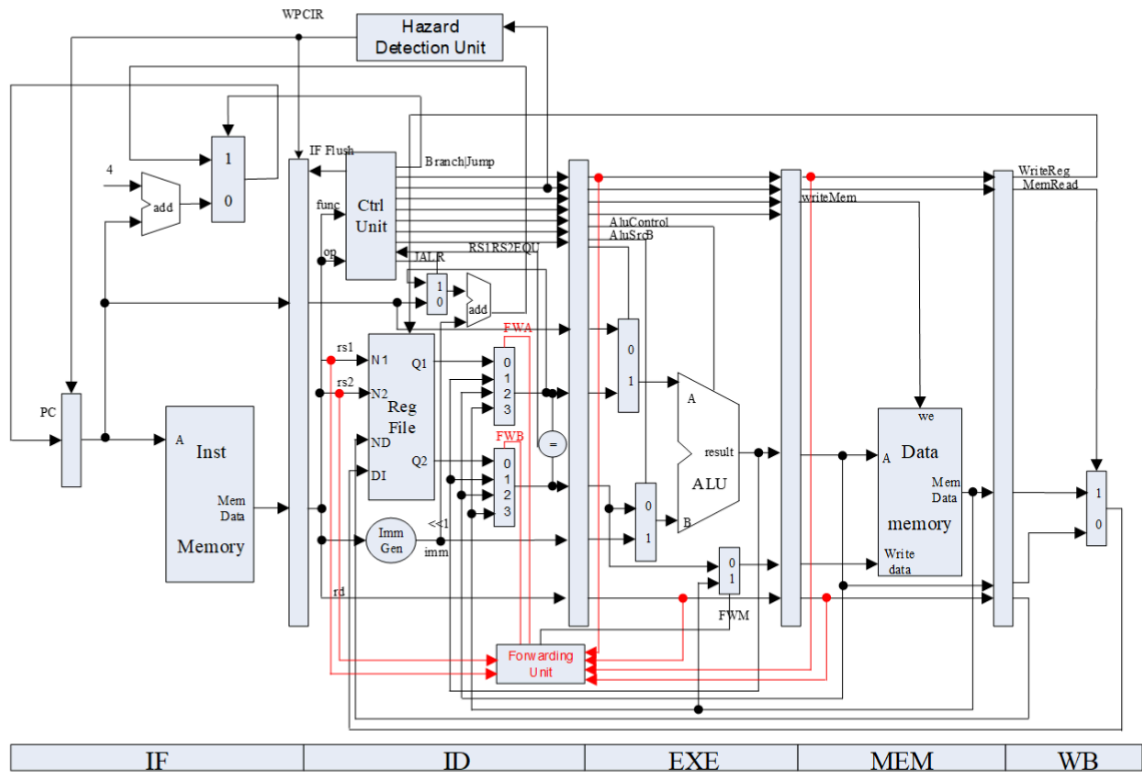

图16 体系实验的前递在ID阶段

5. 最开始仿真的时候所有的指令forward都是正确的，我一度以为要收工了，一上板之后发现`lw x4, 8(x0)`指令开始部分寄存器的值不对，排查好久之后发现是我的`RV32core.v`里面的ALU选择变量写成了`ID`阶段的选择变量而不是`EXE`阶段的（VScode自动补全害人），以后代码填空的时候一定要好好看上下文，确认清楚每个相似变量的含义。

```
MUX2T1_32 mux_A_EXE(.I0(PC_EXE),.I1(rs1_data_EXE),.s(ALUSrc_A_ctrl),.o(ALUA_EXE));
MUX2T1_32 mux_B_EXE(.I0(rs2_data_EXE),.I1(Imm_EXE),.s(ALUSrc_B_ctrl),.o(ALUB_EXE));
```

图17 导致寄存器值错误的原始代码