浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：李安旭

学　　院：计算机科学与技术学院

系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3220102479

指导教师：姜晓红

2024 年 12 月 24 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：Dynamically Scheduled Pipelines using Scoreboarding/Tomasulo

学生姓名：李安旭 专业：计算机科学与技术 学号：3220102479

同组学生姓名：王晓宇 指导老师：姜晓红

实验地点： 实验日期：2024 年 12 月 24 日

一、 实验目的和要求

Tips: 写出本次实验的目的与要求

- 理解支持多周期操作的流水线原理、设计方法和验证方法
- 理解带有 Scoreboard 的动态调度原理

要求：

- R/I-type：4
- mul：4
- divu：4
- load：4
- store：4
- branch：4
- jal：4
- jalr：4
- lui：4
- auipc：4

- Issue出现FU unit hazard但未出现WAW：5
- Issue出现WAW但未出现FU unit hazard：5
- Issue同时出现FU unit hazard和WAW：5
- Issue后发生RAW stall：5
- Issue后未发生RAW stall：5
- 在某条指令尝试write back的过程中出现WAR stall：5
- 在同一周期内，一条指令WB结束欲清空RRS，另一条指令写同一个寄存器，发现RRS将被清空所以成功Issue，向RRS写入新值：10

- alu unit >= 2 busy：5
- mem unit >= 2 busy：5
- mul unit >= 2 busy：5
- mul unit >= 2 busy：5
- **Renaming： +20**

二、 实验内容和原理

Tips：结合代码实现，简要解释：

发射逻辑以及发射后更新 qj/qk/rj/rk 的逻辑

WB 的逻辑（包括 WAR stall）

Branch 的逻辑（包括 predict not taken）

为实现多 unit 增加的结构

1. 发射逻辑以及发射后更新 qj/qk/rj/rk 的逻辑

```
wire[2:0] use_FU = {3{use_ALU}}  & `FU_ALU  |
                   {3{use_MEM}}  & `FU_MEM  |
                   {3{use_MUL}}  & `FU_MUL  |
                   {3{use_DIV}}  & `FU_DIV  |
                   {3{use_JUMP}} & `FU_JUMP ;

wire[4:0] op = {5{ADD}}          & `ALU_ADD    |      You, 3周前 • Finish Lab5_report ···

wire[4:0] dst = {5{R_valid | I_valid | L_valid | LUI | AUIPC | JAL | JALR}} & rd;
wire[4:0] src1 = {5{R_valid | I_valid | S_valid | L_valid | B_valid | JALR}} & rs1;
wire[4:0] src2 = {5{R_valid | S_valid | B_valid}} & rs2;
wire[2:0] fu1 = RRS[src1];
wire[2:0] fu2 = RRS[src2];
wire rdy1 = ~|fu1;
wire rdy2 = ~|fu2;
```

```
// IS
if (RO_en) begin
    // not busy, no WAW, write info to FUS and RRS
    // Issue指令时候的处理逻辑
    if (|dst) RRS[dst] <= use_FU;//TO_BE_FILLED;
    FUS[use_FU][`BUSY] <= 1'b1;//TO_BE_FILLED;
    FUS[use_FU][`SRC1_H:`SRC1_L] <= src1; //TO_BE_FILLED;
    FUS[use_FU][`SRC2_H:`SRC2_L] <= src2; //TO_BE_FILLED;
    FUS[use_FU][`DST_H:`DST_L] <= dst; //TO_BE_FILLED;
    FUS[use_FU][`OP_H:`OP_L] <= op; //TO_BE_FILLED;
    FUS[use_FU][`FU1_H:`FU1_L] <= fu1;//TO_BE_FILLED;
    FUS[use_FU][`FU2_H:`FU2_L] <= fu2;//TO_BE_FILLED;
    FUS[use_FU][`FU_DONE] <= 1'b0;//new
    FUS[use_FU][`RDY1] <= rdy1;//TO_BE_FILLED;
    FUS[use_FU][`RDY2] <= rdy2;//TO_BE_FILLED;

    IMM[use_FU] <= imm;
    PCR[use_FU] <= PC;
end      You, 3周前 • Finish Lab5_report
```

发射逻辑

如果目的寄存器 dst 不为零，则将 use_FU 写入寄存器重命名表 RRS 中对应的位置。

更新功能单元状态：

- 将 FUS[use_FU][BUSY]` 置为 1，表示我们将使用该功能单元作为此指令的 FU。
- 将源操作数 src1 、src2 和 dst 写入功能单元 FUS 中对应的位置表明使用数据的来源与去向。
- 将操作码 op 写入功能单元 FUS 中对应的位置。
- 将功能单元类型 fu1 和 fu2 写入功能单元 FUS 中表明操作数的 FU 来源。
- 将 FUS[use_FU][FU_DONE]` 置为 0，表示操作尚未完成。
- 将 rdy1 和 rdy2 写入功能单元 FUS ，表示源操作数是否准备好。

更新立即数和程序计数器：

将立即数 imm 写入 IMM 中对应的位置。

将程序计数器 PC 写入 PCR 中对应的位置。

这里我们便将指令成功发射出去，并设置好了对应的逻辑位

更新 qj/qk/rj/rk 的逻辑

对于 q，我们直接填充 RRS 中对应的占用单元 FU，如果此处的值为 0 表示该处数据没有等待的 FU，我们在后文如果成功写入之后要及时清零 RRS 中的 FU 值。

其中 ready 要根据当前两个源操作数是否在先前的指令中都已经得到最新的结果进行赋值，相应的判断准备好的逻辑我们在前文做好了判断——如果源操作数已经准备好则为 1 否则为 0。

2. WB 的逻辑（包括 WAR stall）

WB 逻辑

```
// WB
// WB的处理逻辑，检测何时写入，以及写入的位置
always @ (*) begin
    write_sel = 0;
    reg_write = 0;
    rd_ctrl = 0;

    if (FUS[`FU_JUMP][`FU_DONE] & ~JUMP_WAR) begin//(TO_BE_FILLED) begin
        write_sel = `FU_JUMP-1;//TO_BE_FILLED;
        reg_write = 1'b1;//TO_BE_FILLED;
        rd_ctrl = FUS[`FU_JUMP][`DST_H:`DST_L];//TO_BE_FILLED;
    end
    else if (FUS[`FU_ALU][`FU_DONE] & ~ALU_WAR) begin//(TO_BE_FILLED) begin
        // 这里需要填入多行 Multiple rows need to be filled in here
        write_sel = `FU_ALU-1;//TO_BE_FILLED;
        reg_write = 1'b1;//TO_BE_FILLED;
        rd_ctrl = FUS[`FU_ALU][`DST_H:`DST_L];//TO_BE_FILLED;
    end
    else if (FUS[`FU_MEM][`FU_DONE] & ~MEM_WAR) begin//(TO_BE_FILLED) begin
        // 这里需要填入多行 Multiple rows need to be filled in here      You, 3周前 • Finish
        write_sel = `FU_MEM-1;//TO_BE_FILLED;
        reg_write = 1'b1;//TO_BE_FILLED;
        rd_ctrl = FUS[`FU_MEM][`DST_H:`DST_L];//TO_BE_FILLED;
    end
    else if (FUS[`FU_MUL][`FU_DONE] & ~MUL_WAR) begin//(TO_BE_FILLED) begin
        write_sel = `FU_MUL-1;//TO_BE_FILLED;
        reg_write = 1'b1;//TO_BE_FILLED;
        rd_ctrl = FUS[`FU_MUL][`DST_H:`DST_L];//TO_BE_FILLED;
    end
    else if (FUS[`FU_DIV][`FU_DONE] & ~DIV_WAR) begin//(TO_BE_FILLED) begin
        write_sel = `FU_DIV-1;//TO_BE_FILLED;
        reg_write = 1'b1;//TO_BE_FILLED;
        rd_ctrl = FUS[`FU_DIV][`DST_H:`DST_L];//TO_BE_FILLED;
    end
end
```

写回的时候，如果当前对应 FUS 项的 done 为 1 表示 FU 已经执行完毕，并且没有出现 WAR 竞争时，就会把对应的写回使能打开，以及传出正确的 rd 寄存器的编号以便寄存器堆进行改写。

WAR stall

```
    // WAR的检测，这里的代码有点长
    wire ALU_WAR = ~(
        (FUS[`FU_MEM][`SRC1_H:`SRC1_L] == FUS[`FU_ALU][`DST_H:`DST_L] && FUS[`FU_MEM][`RDY1])  ||
        (FUS[`FU_MEM][`SRC2_H:`SRC2_L] == FUS[`FU_ALU][`DST_H:`DST_L] && FUS[`FU_MEM][`RDY2])  ||
        (FUS[`FU_MUL][`SRC1_H:`SRC1_L] == FUS[`FU_ALU][`DST_H:`DST_L] && FUS[`FU_MUL][`RDY1])  ||
        (FUS[`FU_MUL][`SRC2_H:`SRC2_L] == FUS[`FU_ALU][`DST_H:`DST_L] && FUS[`FU_MUL][`RDY2])  ||
        (FUS[`FU_DIV][`SRC1_H:`SRC1_L] == FUS[`FU_ALU][`DST_H:`DST_L] && FUS[`FU_DIV][`RDY1])  ||
        (FUS[`FU_DIV][`SRC2_H:`SRC2_L] == FUS[`FU_ALU][`DST_H:`DST_L] && FUS[`FU_DIV][`RDY2])  ||
        (FUS[`FU_JUMP][`SRC1_H:`SRC1_L] == FUS[`FU_ALU][`DST_H:`DST_L] && FUS[`FU_JUMP][`RDY1]) ||
        (FUS[`FU_JUMP][`SRC2_H:`SRC2_L] == FUS[`FU_ALU][`DST_H:`DST_L] && FUS[`FU_JUMP][`RDY2])
    );//TO_BE_FILLED;
    wire MEM_WAR = ~(
        (FUS[`FU_ALU][`SRC1_H:`SRC1_L] == FUS[`FU_MEM][`DST_H:`DST_L] && FUS[`FU_ALU][`RDY1]) ||
        (FUS[`FU_ALU][`SRC2_H:`SRC2_L] == FUS[`FU_MEM][`DST_H:`DST_L] && FUS[`FU_ALU][`RDY2]) ||
        (FUS[`FU_MUL][`SRC1_H:`SRC1_L] == FUS[`FU_MEM][`DST_H:`DST_L] && FUS[`FU_MUL][`RDY1]) ||
        (FUS[`FU_MUL][`SRC2_H:`SRC2_L] == FUS[`FU_MEM][`DST_H:`DST_L] && FUS[`FU_MUL][`RDY2]) ||
        (FUS[`FU_DIV][`SRC1_H:`SRC1_L] == FUS[`FU_MEM][`DST_H:`DST_L] && FUS[`FU_DIV][`RDY1]) ||
        (FUS[`FU_DIV][`SRC2_H:`SRC2_L] == FUS[`FU_MEM][`DST_H:`DST_L] && FUS[`FU_DIV][`RDY2]) ||
        (FUS[`FU_JUMP][`SRC1_H:`SRC1_L] == FUS[`FU_MEM][`DST_H:`DST_L] && FUS[`FU_JUMP][`RDY1]) ||
        (FUS[`FU_JUMP][`SRC2_H:`SRC2_L] == FUS[`FU_MEM][`DST_H:`DST_L] && FUS[`FU_JUMP][`RDY2])
    );//TO_BE_FILLED;

    wire MUL_WAR = ~(…
    ); //TO_BE_FILLED;

    wire DIV_WAR = ~(…
    );//TO_BE_FILLED;

    wire JUMP_WAR = ~(…
    );//TO_BE_FILLED;
```

```
            // 对于WAR的处理逻辑
            // WB
            if (FUS[`FU_JUMP][`FU_DONE] & JUMP_WAR) begin…
            end
            // ALU
            else if (FUS[`FU_ALU][`FU_DONE] & ALU_WAR) begin
                // 这里需要填入多行 Multiple rows need to be filled in here
                //TO_BE_FILLED <= 0;
                FUS[`FU_ALU] <= 32'b0;
                RRS[FUS[`FU_ALU][`DST_H:`DST_L]] <= 3'b0;

                if (FUS[`FU_JUMP][`FU1_H:`FU1_L] == `FU_ALU) FUS[`FU_JUMP][`RDY1]<=1'b1;
                if (FUS[`FU_MEM][`FU1_H:`FU1_L] == `FU_ALU) FUS[`FU_MEM][`RDY1]<=1'b1;
                if (FUS[`FU_MUL][`FU1_H:`FU1_L] == `FU_ALU) FUS[`FU_MUL][`RDY1]<=1'b1;
                if (FUS[`FU_DIV][`FU1_H:`FU1_L] == `FU_ALU) FUS[`FU_DIV][`RDY1]<=1'b1;

                if (FUS[`FU_JUMP][`FU2_H:`FU2_L] == `FU_ALU) FUS[`FU_JUMP][`RDY2]<=1'b1;
                if (FUS[`FU_MEM][`FU2_H:`FU2_L] == `FU_ALU) FUS[`FU_MEM][`RDY2]<=1'b1;
                if (FUS[`FU_MUL][`FU2_H:`FU2_L] == `FU_ALU) FUS[`FU_MUL][`RDY2]<=1'b1;
                if (FUS[`FU_DIV][`FU2_H:`FU2_L] == `FU_ALU) FUS[`FU_DIV][`RDY2]<=1'b1;
            end
            // MEM
            else if (FUS[`FU_MEM][`FU_DONE] & MEM_WAR) begin…
            end
            // MUL
            else if (FUS[`FU_MUL][`FU_DONE] & MUL_WAR) begin…
            end
            // DIV
            else if (FUS[`FU_DIV][`FU_DONE] & DIV_WAR) begin…
            end
        end
    end
```

我们这里每个 FU 单元均有 WAR 检测，我们这里仅对 ALU 单元的 WAR 检测和处理进行解释：

WAR 是当前指令的目的操作数与之前的 FUS 中的源操作数相同，并且之前指令的源操作数对应的 ready 位为 0，这样就会产生 WAR 冲突。这里得到判断逻辑是判断各个 FU 的无 WAR 逻辑后取反，我们单独处理了当 WAR 冲突发生的操作，这里的操作主要是将使用该写回数据的 FU 的状态位改写，主要是 RDY 位的设置。

3. Branch 的逻辑（包括 predict not taken）

```
// normal stall: structural hazard or WAW
assign normal_stall = (use_ALU & (FUS[`FU_ALU][`BUSY] ) )  |
                      (use_MEM & (FUS[`FU_MEM][`BUSY] )) |
                      (use_MUL & (FUS[`FU_MUL][`BUSY] )) |
                      (use_DIV & (FUS[`FU_DIV][`BUSY] )) |
                      (use_JUMP& (FUS[`FU_JUMP][`BUSY]) )|
                      (|(dst & RRS[dst]));


// 1 Enable (或0 Stall)  IS 和 RO.
assign IS_en = IS_flush | (~normal_stall & ~ctrl_stall);//TO_BE_FILLED;
assign RO_en = (~IS_flush & ~normal_stall & ~ctrl_stall);//TO_BE_FILLED;

always @ (posedge clk or posedge rst) begin      You, 3周前 • Finish Lab5_report
    if (rst) begin
        ctrl_stall <= 0;
    end
    else begin
        // IS
        if (RO_en & (use_FU == `FU_JUMP)) begin
            ctrl_stall <= 1;
        end
        else if (JUMP_done) begin
            ctrl_stall <= 0;
        end
    end
end

always @ (posedge clk or posedge rst) begin
    if (rst) begin
        IS_flush <= 0;
    end
    else if (JUMP_done & is_jump) begin
        IS_flush <= 1;
    end
    else begin
        IS_flush <= 0;
    end
end
```

对于 Branch 逻辑，当出现 jump 指令时，我们需要利用 JUMPFU 来进行跳转预测，将结果返回到 CtrlUnit 中判断是否要对流水线进行刷新跳转，具体包括有数值判断跳转和 JALR 等指令

```
reg TO_BE_FILLED = 0;
wire cmp_res;
reg state;
assign finish = (state == 1'b1);//TO_BE_FILLED;
initial begin
    state = 0;
end

reg JALR_reg;
reg[3:0] cmp_ctrl_reg = 0;
reg[31:0] rs1_data_reg = 0, rs2_data_reg = 0, imm_reg = 0, PC_reg = 0;

always@(posedge clk) begin
    if(EN & ~state) begin // state == 0
        // 这里需要填入多行 Multiple rows need to be filled in here
        //TO_BE_FILLED <= 0;
        JALR_reg <= JALR;
        cmp_ctrl_reg <= cmp_ctrl;
        rs1_data_reg <= rs1_data;
        rs2_data_reg <= rs2_data;
        imm_reg <= imm;      You, 上周 • Finish Lab6
        PC_reg <= PC;
        state <= 1;
    end
    else state <= 0;
end

cmp_32 cmp(.a(rs1_data_reg),.b(rs2_data_reg),.ctrl(cmp_ctrl_reg[3:1]),.c(cmp_res));

add_32 a(.a(JALR_reg ? rs1_data_reg : PC_reg),.b(imm_reg),.c(PC_jump));

add_32 b(.a(PC_reg),.b(32'd4),.c(PC_wb));

assign is_jump = (cmp_ctrl_reg[0] | cmp_res) ;//& finish;//TO_BE_FILLED;
```

我们这里是 predict not taken 当结果为需要跳转时，IS_flush 将跳转指令之后的数据刷新，表示预测错误需要 flush 掉先前一条错误指令，如果预测正确则不需要 flush。

4. 为实现多 unit 增加的结构

我们为了统筹所有 FU 的信息联通和同步信息方便，我们设置了 FUS、RRS、IMM 等共享数据寄存器单元，方便我们进行查询 FU 占用信息等等功能

```
reg[31:0] FUS[1:5];
reg[31:0] IMM[1:5];

// records which FU will write corresponding reg at WB
reg[2:0] RRS[0:31];

// sometimes an instruction needs PC to execute
// pc record
reg[31:0] PCR[1:5];
```

FUS 是一个 32 位宽的寄存器数组，用于存储功能单元（Functional Units，FU）的状态信息。每个功能单元都有一个对应的 FUS 寄存器，用于记录该功能单元的各种状态和控制信号。具体来说，FUS 寄存器包含以下信息：

- BUSY 标志：指示功能单元是否正在使用。
- SRC1 和 SRC2：源操作数寄存器编号。
- DST：目的操作数寄存器编号。
- OP：操作码，指示功能单元要执行的操作。
- FU_DONE 标志：指示功能单元的操作是否完成。
- RDY1 和 RDY2：指示源操作数是否准备好。

IMM 是一个 32 位宽的寄存器数组，用于存储立即数（Immediate Values）。立即数是指令中直接包含的常数值，通常用于算术运算或地址计算。每个功能单元都有一个对应的 IMM 寄存器，用于存储当前指令的立即数。

RRS 是一个 3 位宽的寄存器数组，用于实现寄存器重命名（Register Renaming）。寄存器重命名表记录了每个逻辑寄存器当前映射到的功能单元编号。

- 每个逻辑寄存器（0 到 31）对应的功能单元编号（0 到 7），表示该逻辑寄存器的值将由哪个功能单元在写回阶段（WB）写入。

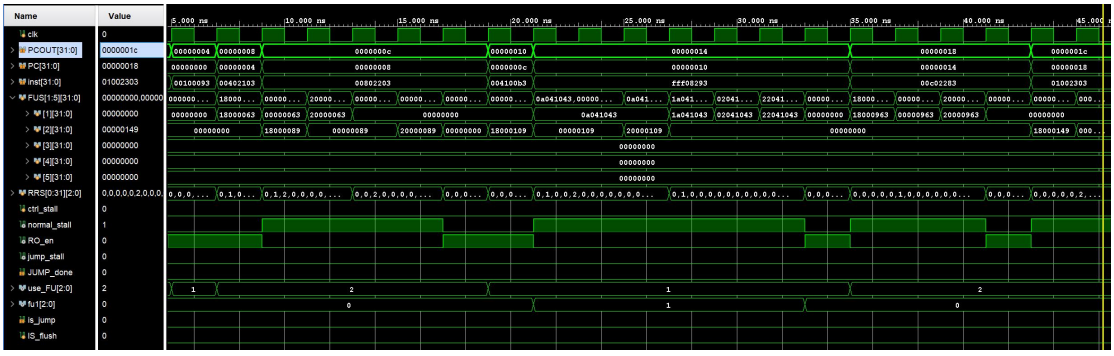
- 通过使用 RRS，可以避免写后写（WAW）和写后读（WAR）数据冒险，提高指令级并行性。

```
// function unit      You, 3周前 • Finish Lab5_
`define FU_BLANK    3'd0
`define FU_ALU      3'd1
`define FU_MEM      3'd2
`define FU_MUL      3'd3
`define FU_DIV      3'd4
`define FU_JUMP     3'd5

// immediate types
`define Imm_type_I 3'b001
`define Imm_type_B 3'b010
`define Imm_type_J 3'b011
`define Imm_type_S 3'b100
`define Imm_type_U 3'b101

// bits in FUS
`define BUSY    0
`define OP_L    1
`define OP_H    5
`define DST_L   6
`define DST_H   10
`define SRC1_L  11
`define SRC1_H  15
`define SRC2_L  16
`define SRC2_H  20
`define FU1_L   21
`define FU1_H   23
`define FU2_L   24
`define FU2_H   26
`define RDY1    27
`define RDY2    28
`define FU_DONE 29
```

这里是设置对应宏以优化我们寄存器对应位的显示

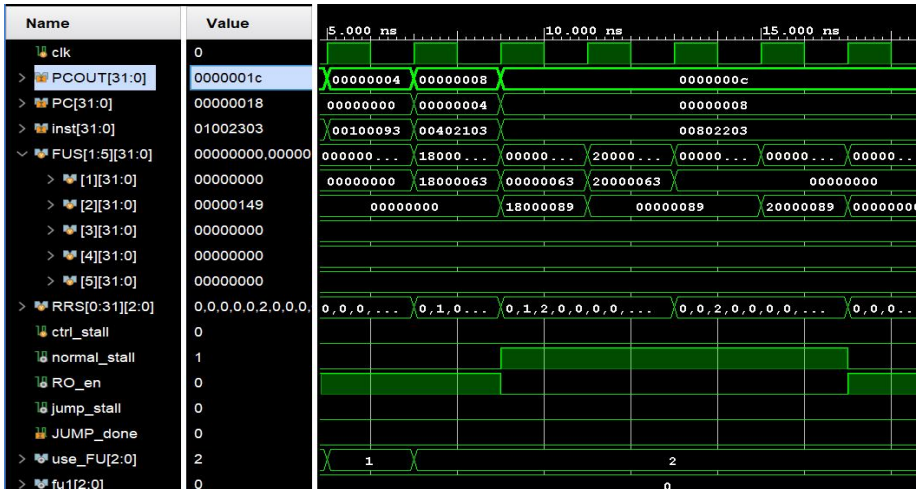
三、 实验过程和数据记录及结果分析

Tips：请给出本次实验仿真关键信号截图，并结合波形简要解释各种 Hazard 发生和解决的逻辑



- FU_Hazard:

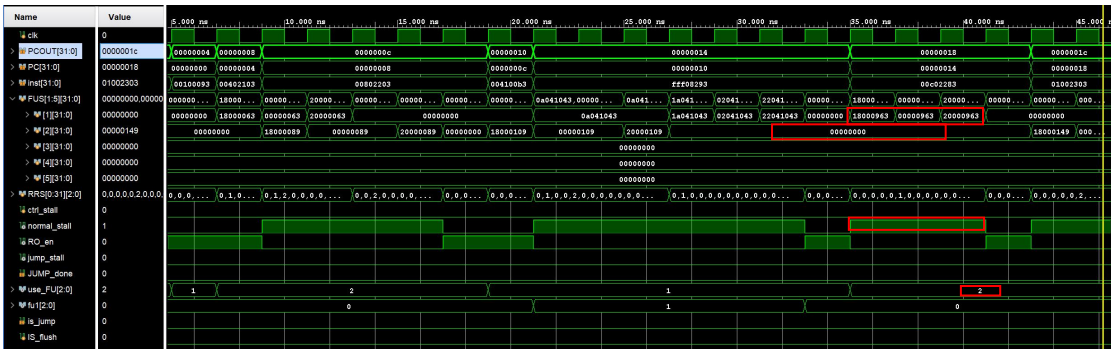
```
0x4     0x00402103    lw x2          lw x2, 4(x0)
                      4(x0)

0x8     0x00802203    lw x4          lw x4, 8(x0)
                      8(x0)
```



如图所示，此时 Ctrl_Unit 正在执行 lw 指令，但是此时通过 FUS 可知前面的 lw 指令还没有执行完毕，所以发生 FU_Hazard，nomal_stall 也因此变成 1，流水线 stall。

- WAW_Hazard

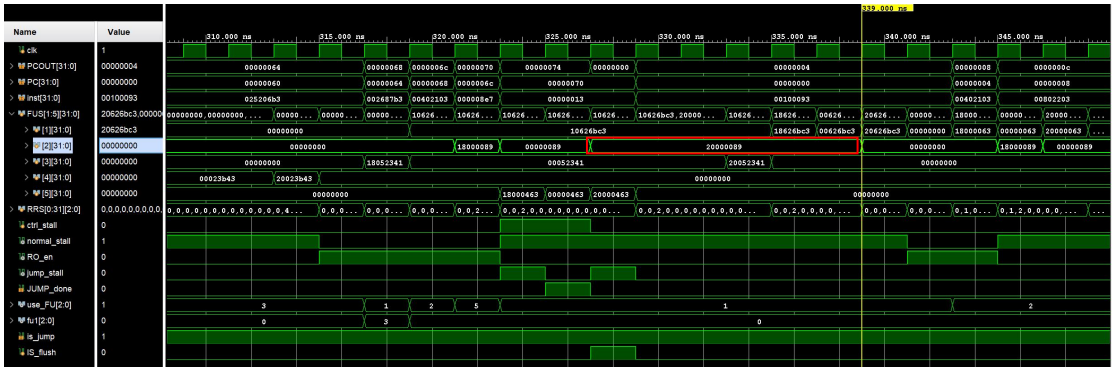
```
0x10   0xFFF08293   addi x5 x1 -1   addi x5, x1, -1

0x14   0x00C02283   lw x5 12(x0)    lw x5, 12(x0)
```



如图所示，此时 Ctrl_Unit 正要执行 lw 指令，但是此时前一条指令同样也要写 x5，所以导致 WAW_stall，由于这个 stall 被包含在 normal_stall 中，所以并不能直接看出来，但是通过排除法，此时并未发生 struct_stall，可以说明必定发生了 WAW_stall，导致了流水线在 PC=0x14 时 stall

- WAR_Hazard

0x60	0x025206B3	mul x13 x4 x5	mul x13, x4, x5
0x64	0x002687B3	add x15 x13 x2	add x15, x13, x2
0x68	0x00402103	lw x2 4(x0)	lw x2, 4(x0)

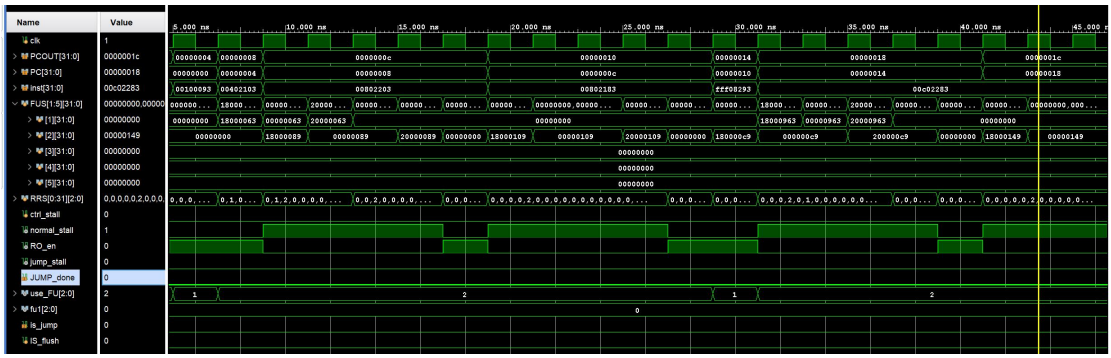


如图所示，此时 Ctrl_Unit 执行 lw 指令，由于 add 指令的 x15 在等待 mul 指令的计算值，所以 add 指令不能开始读操作数，此时 lw 指令会先一步进入 WB 阶段，此时会检测到要写入的 rd 和 ALU 部件的一个 ready 的 src 冲突了，写入就会暂停，直到 add 指令开始运行

- RAW_Hazard

0x60	0x025206B3	mul x13 x4 x5	mul x13, x4, x5
0x64	0x002687B3	add x15 x13 x2	add x15, x13, x2
0x68	0x00402103	lw x2 4(x0)	lw x2, 4(x0)

如图，add 指令的 x13 依赖于前面的 mul 指令，所以会产生 RAW 冲突，此时 FUS 中的 add 部件的 rdy1 状态就会被设置 0，表示该寄存器需要等待前面的指令的计算值，等到 mul 指令计算完毕，add 部件就准备完毕，下个周期开始执行。



• FU_Hazard 和 WAW_Hazard 一起发生

0x1c	0x01402383	lw x7 20(x0)	lw x7, 20(x0)
0x20	0x40220433	sub x8 x4 x2	sub x8,x4,x2
0x24	0x00402403	lw x8 4(x0)	lw x8,4(x0)

如图所示，此时 FU_hazard 和 WAW 同时发生，我们可以看到此时 Ctrl_Unit 要执行 lw 指令，此时前面的 FU_ALU 部件还未写回，会发生 WAW 冲突，而 FU_MEM 部件也在被占用，会发生 FU_Hazard，所以流水线会 stall 5 个周期后。

四、 讨论与心得

Tips: 请写出对本次实验内容的深入讨论或者本次实验的心得体会。

亲手实现了 scoreboard 使我对课上讲的知识理解变得更加深刻，之前对于 WAR 的出现一直不太理解，是在分析波形的时候才彻底理解。本实验也帮助我理解了整个 scoreboard 的数据流。