浙江大学

本科实验报告

课程名称：	计算机体系结构
姓　　名：	张天逸
学　　院：	竺可桢学院
专　　业：	计算机科学与技术
学　　号：	3220106424
指导教师：	姜晓红

2024 年 11 月 27 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：Pipelined CPU multi-cycle operations (Out of Order Version)

学生姓名：张天逸　　学号：3220106424

同组学生姓名：朱家骏　　指导老师：姜晓红

一、 实验目的和要求

1. 理解支持多周期操作的流水线的原理。
2. 掌握支持多周期操作的流水线的设计方法。
3. 掌握支持多周期操作的流水线的验证方法。

二、 实验内容和原理

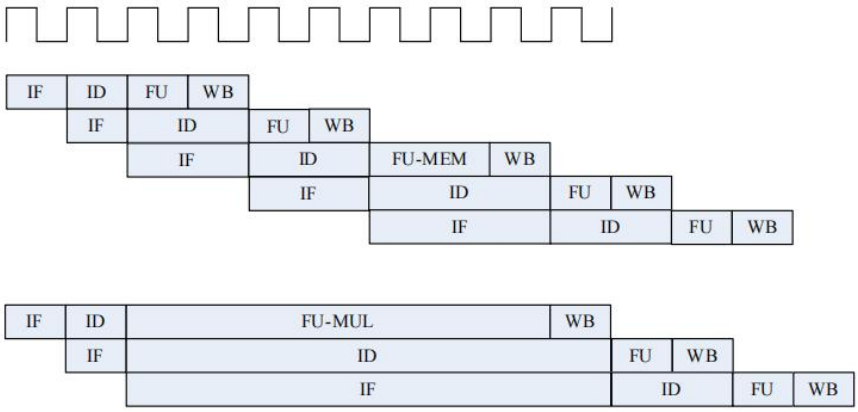
内容：

1. 设计一款支持乱序执行的多周期流水线 CPU。
2. 将 CPU 设计成 IF/ID/FU/WB 四阶段，且 FU 阶段支持多周期。
3. 支持指令乱序完成，并检测和解决 CPU 执行过程的冒险情况。

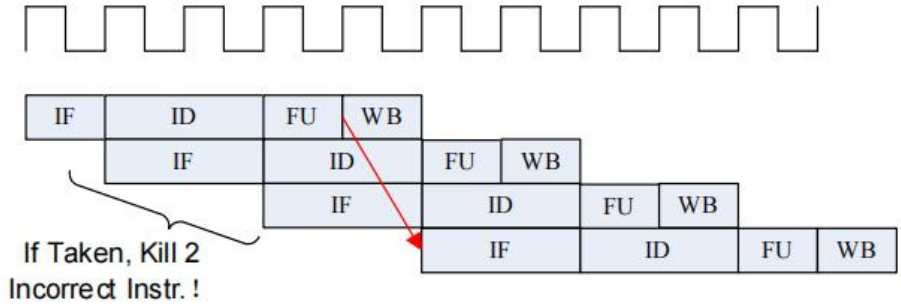
原理：

将 CPU 的功能进行拓展和分拆，在 FU 阶段共有算术逻辑单元、访存单元、乘法单元、除法单元、跳转单元 5 个功能单元，每个单元执行命令所需的时钟周期数量各不相同（比如算术逻辑单元 ALU 需要 1 个周期、访存单元需要 2 个时钟周期，等等）。

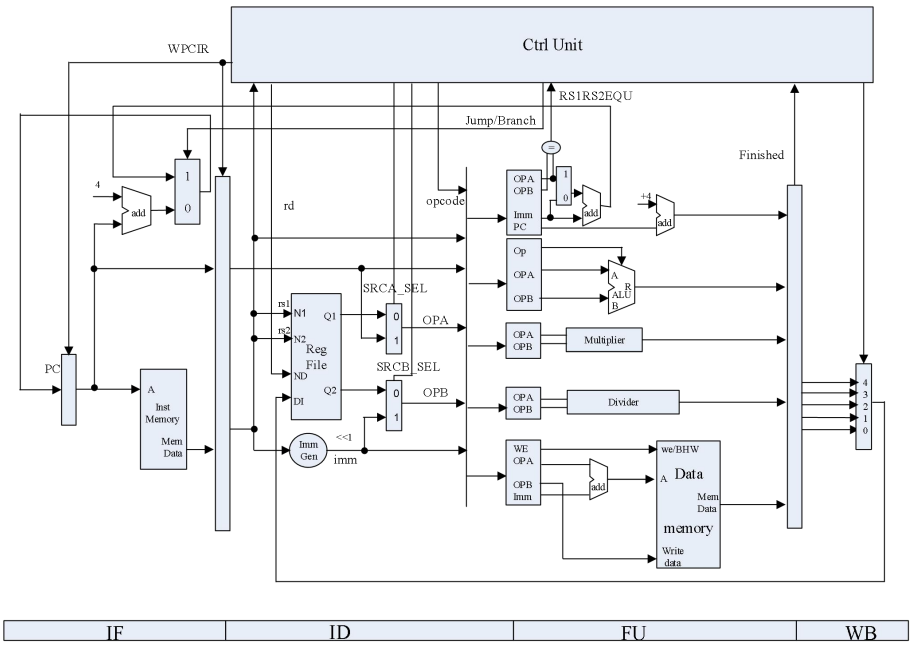
在处理数据冲突上，需要前面指令结果的指令需等前面的指令将结果写回后（即 WB 阶段结束），才能进入 FU 阶段。如下图。



在控制冲突上，采用预测不跳转，如果分支、跳转指令在 FU 阶段结束后发现预测错误，则将在该指令之后捕获的指令（一共 2 条）全部删除，重新从正确的位置开始捕获新的指令。如下图。



本实验设计的支持多周期操作的流水线 CPU 的构造如下图：



三、 实验过程和数据记录及结果分析

1. 补全框架内代码文件

(1) 补全各功能单元的逻辑代码

各功能单元中，都包含了状态的切换，当功能单元中的状态转变为某一指定状态时，表明操作已结束，结果已经成功生成。

I. 访存单元 (FU_mem.v 中，操作结束时 state == 1)

这里不仅是要补充将输入保存至相应寄存器的逻辑，还要补充开始执行时状态和状态切换的逻辑，以及得到要访问的地址的运算方式。代码如下：

```
always @(posedge clk) begin
    if (EN && state[0] == 0) begin
        rs1_data_reg <= rs1_data;
        rs2_data_reg <= rs2_data;
        imm_reg <= imm;
        mem_w_reg <= mem_w;
        bhw_reg <= bhw;
        state <= 2'b01;
    end
    else if(state[0] == 1)begin
        rs1_data_reg_cur <= rs1_data_reg;
        rs2_data_reg_cur <= rs2_data_reg;
        imm_reg_cur <= imm_reg;
        mem_w_reg_cur <= mem_w_reg;
        bhw_reg_cur <= bhw_reg;
        state <= 2'b10;
    end
    else begin
        state <= 2'b00;
    end
end
```

访存单元需 2 个时钟周期执行完成。

II. 乘法单元 (FU_mul.v 中，操作结束时 state == 1)

这里补充将输入保存至相应寄存器、开始执行时状态和状态切换的逻辑（还有一个从乘法运算器中获得运算结果的 wire mulres[63:0]的声明），代码如下：

```
always@(posedge clk) begin
    if(EN & ~|state) begin
        A_reg <= A;
        B_reg <= B;
        state <= 7'b1000000;
    end
    else state <= {1'b0, state[6:1]};
end
```

```
assign res = mulres[31:0];
```

乘法单元需 7 个时钟周期执行完成。

III. 除法单元 (FU_div.v 中，在工作时 state == 1)

这里补充将输入保存至相应寄存器、开始执行时状态和状态切换的逻辑。其中，A 为被除数，A_valid 表示被除数是否有效（恒有效，恒赋值为 1）；B 为被除数，B_valid 表示被除数是否有效（若 B 不为 0 则有效，赋值为 1；否则赋值为 0）。除法单元执行需要的时钟周期视除法器运行情况而定。代码如下：

```
always@(posedge clk) begin
    if(EN & ~state) begin
        A_reg <= A;
        B_reg <= B;
        A_valid <= 1;
        B_valid <= 1;
        state <= 1;
    end
    else if(res_valid) begin
        A_valid <= 0;
        B_valid <= 0;
        state <= 0;
    end
end
```

```
assign res = divres[63:32];
```

IV. 跳转单元 (FU_jump.v 中，结束时 state == 1)

这里补充计算得到跳转的目的地址的逻辑（对于 jalr 指令，目的地址为源寄存器 1 的值加上立即数；其他则是 PC 的值加上立即数）。对于 branch 指令，调用已有模块 cmp_32，判断两个源寄存器中的值是否满足跳转条件，输出满足与否的结果供外部使用，以实现条件跳转。代码如下：

```
cmp_32 cmp(.a(rs1_data_reg), .b(rs2_data_reg), .ctrl(cmp_ctrl_reg), .c(cmp_res));

add_32 add(.a(JALR_reg ?rs1_data_reg:PC_reg), .b(imm_reg), .c(PC_jump));//第一个空有坑

add_32 add2(.a(PC_reg), .b(32'd4), .c(PC_wb));
```

跳转单元只需 1 个时钟周期就执行完成。

(2) 补充控制单元和框架连线

I. 控制单元

我们定义了几个用于检测硬件流水线中潜在冒险（hazards）的信号。WAW 检测写后写冒险。RAW_rs1 和 RAW_rs2 分别检测 rs1 和 rs2 的读后写冒险，WB_structure_hazard 检测写回阶段的结构冒险，FU_structure_hazard 检测功能单元的结构冒险，FU_hazard 综合所有冒险信号，最终确定是否存在冒险。代码如下：

```
wire WAW = rd && ((rd == FU_write_to[1] && !fu_en_alu) || (rd == FU_write_to[2] && !fu_en_mem) ||
        (rd == FU_write_to[3] && !fu_en_mul) || (rd == FU_write_to[4] && !fu_en_div) ||
        (rd == FU_write_to[5] && !fu_en_jump));    //这里判断预约寄存器的代码可能非常长
wire RAW_rs1 = ((rs1 == FU_write_to[1]) || (rs1 == FU_write_to[2]) || (rs1 == FU_write_to[3]) ||
        (rs1 == FU_write_to[4]) || (rs1 == FU_write_to[5])) && rs1;
wire RAW_rs2 = ((rs2 == FU_write_to[1]) || (rs2 == FU_write_to[2]) || (rs2 == FU_write_to[3]) ||
        (rs2 == FU_write_to[4]) ||(rs2 == FU_write_to[5])) && rs2;
wire WB_structure_hazard = (reservation_reg[0] != use_FU && FU_status[use_FU] == 1 );
wire FU_structure_hazard = reservation_reg[FU_delay_cycles[use_FU]] == 0 ? 0 : 1;
wire FU_hazard = WAW|RAW_rs1|RAW_rs2|WB_structure_hazard|FU_structure_hazard;
```

我们在控制单元（CtrlUnit）中处理功能单元（FU）的状态更新逻辑。具体来说，当保留寄存器 reservation_reg[0]的值不为 0 时，会执行一系列操作来更新功能单元的状态。操作包括禁用对应功能单元的写回使能，清除对应功能单元的状态，清除对应功能单元的写入目标。代码如下：

```
if (reservation_reg[0] != 0) begin
    FU_writeback_en[reservation_reg[0]] <= 1'b0;
    FU_status[reservation_reg[0]] <= 0;
    FU_write_to[reservation_reg[0]] <= 0;
end
```

下面处理功能单元（FU）的使用情况和指令有效性的逻辑。具体来说，当功能单元未被使用且指令有效时，会执行一系列操作来更新保留寄存器和其他相关信号。操作包括将 reservation_reg 数组中的所有元素向左移动一位，清空 reservation_reg 数组中最后一个位置的值，清除功能单元的输入 B 和输入 J。代码如下：

```
if (use_FU == 0 && valid_ID) begin
    for (i = 0; i < 31; i=i+1) begin
        reservation_reg[i] <= reservation_reg[i + 1];
    end
    reservation_reg[31] <= 0;
    B_in_FU <= 0;
    J_in_FU <= 0;
end
```

下面处理流水线停顿（stall）的逻辑。具体来说，当检测到冒险（hazard）、ID 阶段刷新（flush）或下一个 ID 阶段刷新，并且当前指令有效时，会执行一系列操作以确保流水线的正确运行。代码如下：

```
else if ((FU_hazard | reg_ID_flush | reg_ID_flush_next)&&valid_ID) begin   // stall
    for (i = 0; i < 31; i=i+1) begin
        if (reservation_reg[FU_delay_cycles[use_FU]]) begin
            if(reservation_reg[FU_delay_cycles[use_FU]]==0) begin
                fu_en_empty<= 1'b1;
            end
            else if(reservation_reg[FU_delay_cycles[use_FU]]==1) begin
                fu_en_alu <= 1'b1;
            end
            else if(reservation_reg[FU_delay_cycles[use_FU]]==2) begin
                fu_en_mem <= 1'b1;
            end
            else if(reservation_reg[FU_delay_cycles[use_FU]]==3) begin
                fu_en_mul <= 1'b1;
            end
            else if(reservation_reg[FU_delay_cycles[use_FU]]==4) begin
                fu_en_div <= 1'b1;
            end
            else if(reservation_reg[FU_delay_cycles[use_FU]]==5) begin
                fu_en_jump <= 1'b1;
            end
        end
        reservation_reg[i] <= reservation_reg[i + 1];
    end
    reservation_reg[31] <= 0;
    B_in_FU <= 0;
    J_in_FU <= 0;
end
```

下面处理指令有效（valid_ID）的情况。具体来说，当指令有效时，会执行一系列操作来更新功能单元（FU）的状态和保留寄存器。操作包括清除所有功能单元的使能信号，更新当前使用功能单元的状态，更新写回使能和目标寄存器，更新保留寄存器，确保在延迟周期结束后释放功能单元，更新功能单元的输入信号。

```
else if(valid_ID)begin
        fu_en_empty <= 0;
        fu_en_alu <= 0;
        fu_en_div <= 0;
        fu_en_mem <= 0;
        fu_en_mul <= 0;
        fu_en_jump <= 0;
        FU_status[use_FU] <= 1;
        FU_writeback_en[use_FU] <= 1'b1 & rd_used;
        FU_write_to[use_FU] <= rd & {5{rd_used}};
        reservation_reg[FU_delay_cycles[use_FU] - 1] <= use_FU;
        for (i = 0; i < 31; i=i+1) begin
            if(i != FU_delay_cycles[use_FU] - 1) begin
                reservation_reg[i] <= reservation_reg[i + 1];
            end
        end
        reservation_reg[31] <= 0;
        B_in_FU <= B_valid;
        J_in_FU <= JAL | JALR;
end
```

II. 框架连线

选择一个功能单元，将其结果写回寄存器时，当选择信号为 1 时，选择 ALU；当选择信号为 2 时，选择访存功能单元；当选择信号为 3 时，选择乘法功能单元；当选择信号为 4 时，选择除法功能单元；当选择信号为 5 时，选择跳转功能单元。故连线如下：

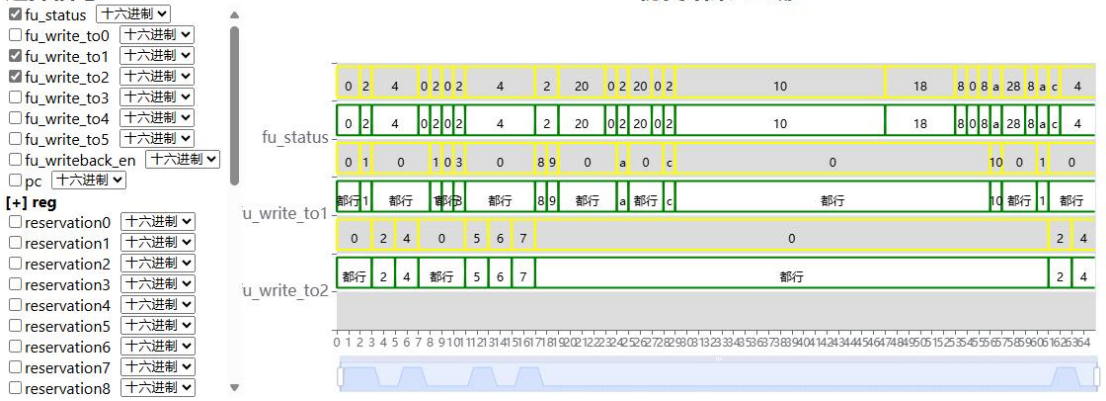
```
MUX8T1_32 mux_DtR(
.s(DatatoReg_ctrl),
.I0(0),
.I1(ALUout_FU),
.I2(mem_data_FU),
.I3(mulres_FU),
.I4(divres_FU),
.I5(PC_wb_FU),
.I6(0),
.I7(0),
.o(wt_data_WB));
```

2. 实验结果分析

使用课程框架中提供的汇编指令转化为指令码存入 rom.hex，使用 ram.hex 内存数据进行测试。仿真结果截图如下所示：

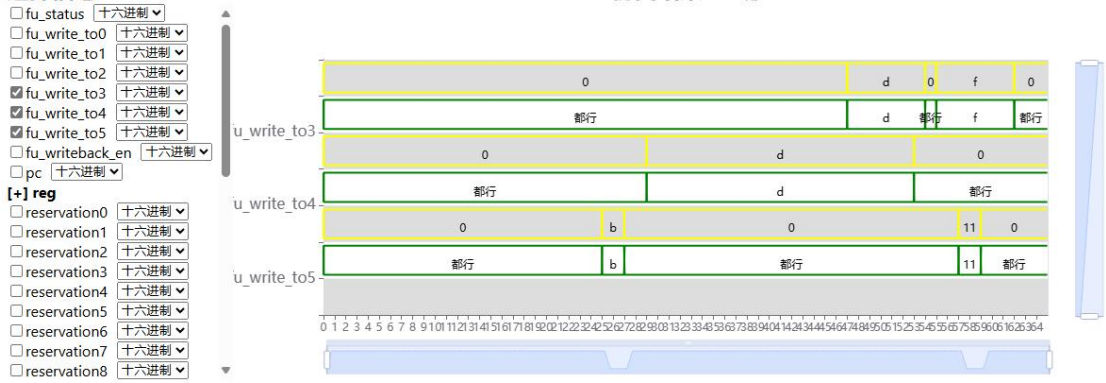
选择信号

仿真结果: 正确



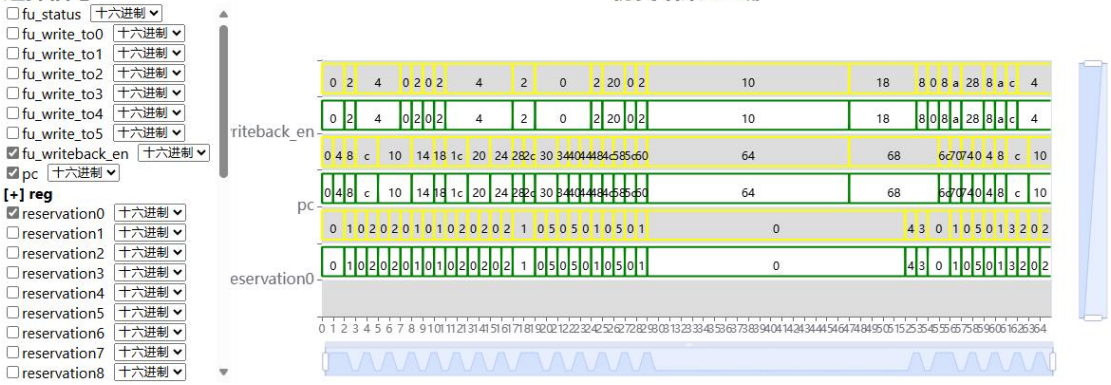
选择信号

仿真结果: 正确



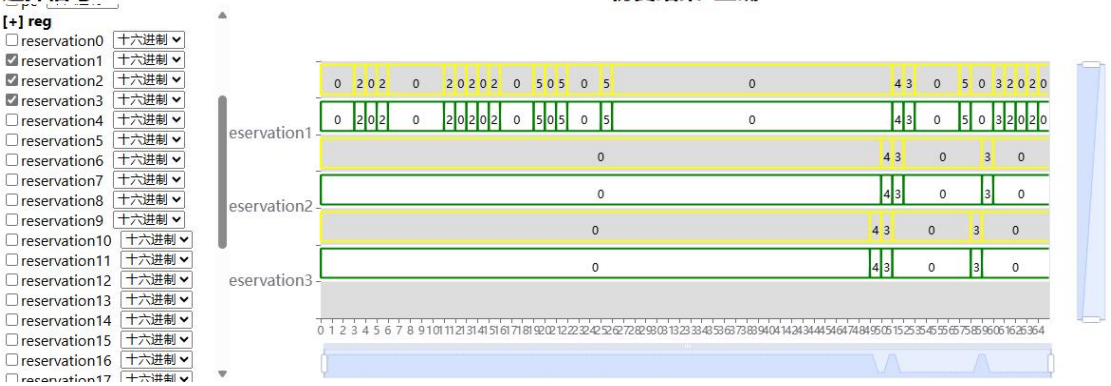
选择信号

仿真结果: 正确



选择信号

仿真结果: 正确



其中，每当功能单元处于执行阶段时，除非要实现跳转，信号 reg_IF_EN 和 reg_ID_EN 的值会设为 0，以停止继续捕获与解析下一条指令，只有当执行完成时，reg_IF_EN 和 reg_ID_EN 才会变成 1，继续捕获和解析下一条指令。当 ID 阶段的指令不是 ALU 指令时，ALUSrcA_ctrl、ALUSrcB_ctrl 和 ALUControl_ctrl 都会保持为 0。当 ID 阶段的指令不涉及立即数时，ImmSel_ctrl 保持为 0。

在有指令在 FU 阶段后，对仿真中几条重点指令的执行情况解读如下：

1. FU unit hazard：在 ID 阶段，指令为 32’h00802203（lw x4, 8(x0)），此指令为 I 型指令，所以对于立即数格式，有 ImmSel_ctrl == 1；此指令有效，故 valid_ID == 1。在 FU 阶段，指令 32’h00402103（lw x2, 4(x0)）进入访存单元被执行，这条指令的结果要写回寄存器 2，所以 rd_ctrl == 5’h2，to_writeback == 1；结果来自访存单元，所以 DatatoReg_ctrl == 2。经历 2 个时钟周期后，执行完成，RegWrite_ctrl 由 0 变为 to_writeback 的 1，在这个时钟周期结尾将结果写回寄存器中，然后 RegWrite_ctrl 再变回 0。

2. RAW：在 ID 阶段，指令为 32’h004100b3（add x1, x2, x4），操作数 A 来自寄存器，故 ALUSrcA_ctrl == 0；操作数 B 也来自寄存器，所以 ALUSrcB_ctrl == 1；做的是加法，所以 ALUControl_ctrl == 1；此指令有效，故 valid_ID == 1。在 FU 阶段，指令 32’h00802203（lw x4, 8(x0)）进入访存单元被执行，这条指令的结果要写回寄存器 4，所以

rd_ctrl == 5’h4，to_writeback == 1；结果来自访存单元，所以 DatatoReg_ctrl == 2。经历 2 个时钟周期后，执行完成，RegWrite_ctrl 由 0 变为 to_writeback 的 1，在这个时钟周期结尾将结果写回寄存 器中，然后 RegWrite_ctrl 再变回 0。

3. FU unit hazard && RAW：在 ID 阶段，指令为 32’hfff08193（addi x3, x1, -1），操作数 A 来自寄存器，故 ALUSrcA_ctrl == 0；操作数 B 来自立即数，所以 ALUSrcB_ctrl== 1；做的是加法，所以 ALUControl_ctrl == 1；此指令为 I 型指令，所以对于立即数格式，有 ImmSel_ctrl == 1；此指令有效，故 valid_ID == 1。在 FU 阶段，指令 32’h004100b3（add x1, x2, x4）进入 ALU 被执行，这条指令的结果要写回寄存器 1，所以 rd_ctrl == 5’h1，to_writeback == 1；结果来自 ALU，所以 DatatoReg_ctrl == 1。经历 1 个时钟周期后，执行完成，RegWrite_ctrl 由 0 变为 to_writeback 的 1，在这个时钟周期结尾将结果写回寄存器中，然后 RegWrite_ctrl 再变回 0。

4. Reservation hazard：在 ID 阶段，指令为 32’h40220433（sub x8, x4, x2），操作数 A 来自寄存器，故 ALUSrcA_ctrl == 0；操作数 B 也来自寄存器，所以 ALUSrcB_ctrl == 0；做的是减法，所以 ALUControl_ctrl == 2；此指令有效，故 valid_ID == 1。在 FU 阶段，指令 32’h01402383（lw x7, 20(x0)）进入访存单元被执行，这条指令的结果要写回寄存器 7，所以 rd_ctrl == 5’h7，to_writeback == 1； 结果来自访存单元，所以 DatatoReg_ctrl == 1。经历 2 个时钟周期后，执行完成，RegWrite_ctrl 由 0 变为 to_writeback 的 1，在这个时钟周期结尾将结果写回寄存器中，然后 RegWrite_ctrl 再变回 0。

5. Branch Not taken：在 ID 阶段，指令为 32’h00520c63（beq x4, x5, label0），此指令为 B 型指令，所以对于立即数格式，有 ImmSel_ctrl == 2；FU_jump_EN 在最后一个时钟周期被设置为 1，使该指令之后进入跳转单元执行；此指令有效，故 valid_ID == 1。在 FU 阶段，指令 32’hffd50493（addi x9, x10, -3）进入 ALU 被执行，这条指令的结果要写回寄存器 9，所以 rd_ctrl == 5’h9，to_writeback == 1；结果来自 ALU，所以 DatatoReg_ctrl == 1。经历 1 个时钟周期后，执行完成，RegWrite_ctrl 由 0 变为 to_writeback 的 1，在这个时钟周期结尾将结果写回寄存器中，然后 RegWrite_ctrl 再变回 0。

6. Reservation hazard, taken：在 ID 阶段，指令为 32’h00420a63（beq x4, x4, label0），此指令为 B 型指令，所以对于立即数格式，有 ImmSel_ctrl == 2；FU_jump_EN 在最后一个时钟周期被设置为 1，使该指令之后进入跳转单元执行；此指令有效，故 valid_ID == 1。在 FU 阶段，指令 32’h00520c63（beq x4, x5, label0）进入跳转单元被执行，没有结果要写回寄存器，所以 to_writeback == 0。由于 x4 内值为 0x10，x5 内值为 0x14，两者不相

等，所以 branch_ctrl == 0，不作跳转。

7. WAW：在 ID 阶段，指令为 32’h025206b3（mul x13, x4, x5），所以 FU_mul_EN 在最后一个时钟周期被设置为 1，使该指令之后进入乘法单元执行；此指令有效，故 valid_ID == 1。在 FU 阶段，指令 32’h0223c6b3（div x13, x7, x2）进入除法单元被执行，这条指令的结果要写回寄存器 13，所以 rd_ctrl == 13 == 5’hd，to_writeback == 1；结果来自除法模块，所以 DatatoReg_ctrl == 4。经历许多时钟周期后，执行完成，在最后一个时钟周期时 RegWrite_ctrl 由 0 变为 to_writeback 的 1，在这个时钟周期结尾将结果写回寄存器中，然后 RegWrite_ctrl 再变回 0。

8. WB quicklier：在 ID 阶段，指令为 32’h00400813（addi x16, x0, 4），操作数 A 来自寄存器，故 ALUSrcA_ctrl == 0；操作数 B 来自立即数，所以 ALUSrcB_ctrl == 1；做的是加法，所以 ALUControl_ctrl == 1；此指令为 I 型指令，所以对于立即数格式，有 ImmSel_ctrl == 1；此指令有效，故 valid_ID == 1。在 FU 阶段，指令 32’h022687b3（mul x15, x13, x2）进入乘法单元被执行，这条指令的结果要写回寄存器 15，所以 rd_ctrl == 14 == 5’hf，to_writeback == 1；结果来自乘法模块，所以 DatatoReg_ctrl == 3。经历 7 个时钟周期后，执行完成，在最后一个时钟周期时 RegWrite_ctrl 由 0 变为 to_writeback 的 1，在这个时钟周期结尾将结果写回寄存器中，然后 RegWrite_ctrl 再变回 0。

四、 讨论与心得

通过这次计算机体系结构的实验，我深刻理解了支持多周期操作的流水线的原理，掌握支持多周期操作的流水线的设计与验证方法。

在实验过程中，我遇到了一些挑战。我们在处理数据冲突时，需要前面指令结果的指令需等前面的指令将结果写回后（即 WB 阶段结束），才能进入 FU 阶段。而在控制冲突上，采用预测不跳转，如果分支、跳转指令在 FU 阶段结束后发现预测错误，则将在该指令之后捕获的指令（一共 2 条）全部删除，重新从正确的位置开始捕获新的指令。上述多周期操作流水线操作使得 CPU 的设计复杂度明显上升，需要对系统进行合理的模块化设计以完成恰当的逻辑判定。

总的来说，我对支持多周期操作的流水线 CPU 的实现方式的理解更为深入了。这种 CPU 将不同类型的指令交给不同的功能单元去执行，从而增加了同一 CPU 能执行的指令的种类，大大提升了 CPU 的能力。这次实验是一次宝贵的学习经历，它不仅提高了我的技术技能，也增强了我的问题解决能力。