浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：夏尤楷

学　　院：计算机科学与技术学院

　　系：计算机科学与技术

专　　业：计算机科学与技术

学　　号：3210104331

指导教师：陈文智

2024 年 1 月 15 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：支持多周期操作的流水线 CPU

学生姓名：夏尤楷 专业：计算机科学与技术 学号：3210104331

同组学生姓名：来思锐 指导老师：陈文智

实验地点：曹西 301 实验日期：2023 年 11 月 21 日

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

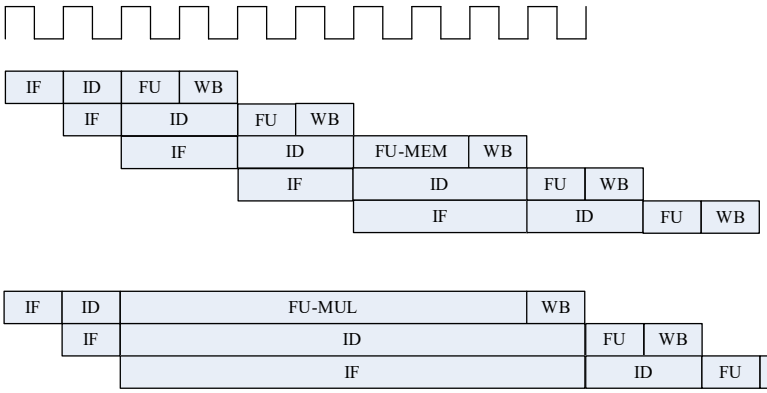
内容：

1. 将流水线重新设计为带 IF/ID/FU/WB 四阶段的模式，其中 FU 阶段支持多周期操作。
2. 重新设计 CPU 控制器。
3. 用程序验证流水线 CPU，观察程序的执行。

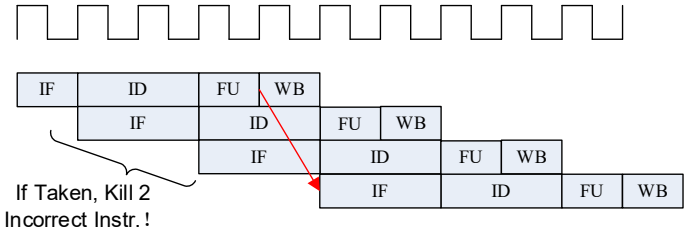
原理：

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）。

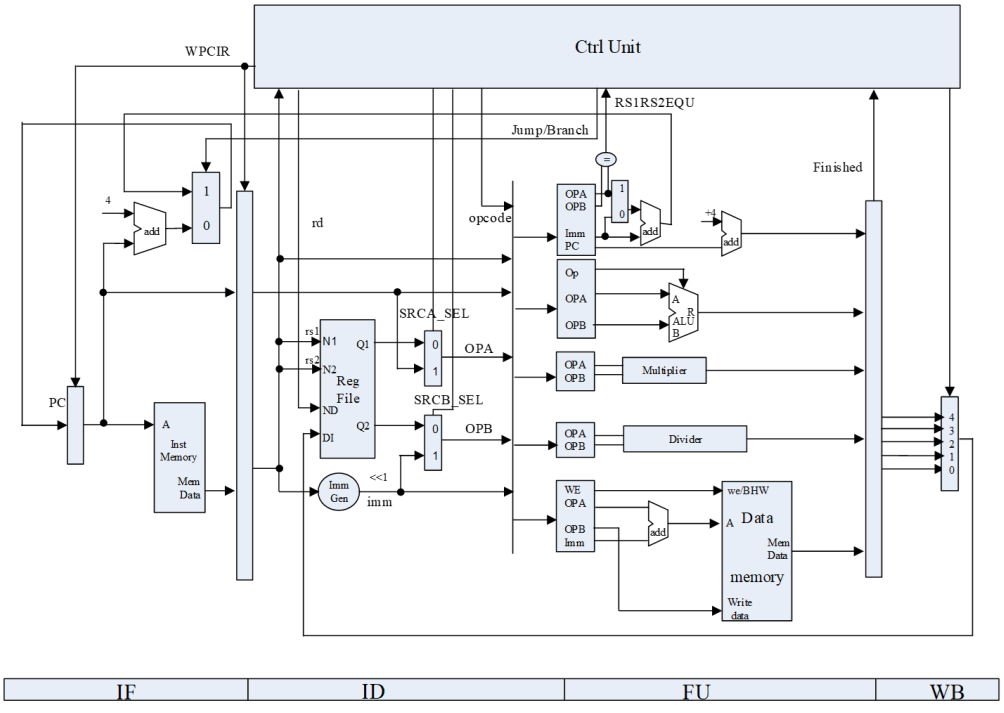
在处理数据冲突上，需要前面指令结果的指令需等前面的指令将结果写回后（即 WB 阶段结束），才能进入 FU 阶段。如下图。



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

（一）补全各功能单元的逻辑

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。

1. ALU 单元（FU_ALU.v 中，操作结束时 state == 1）

这里要补充的逻辑很简单，只要将输入的两个操作数和运算种类（加、减、且、或等）保存至相应的寄存器，使得输入消失后仍能通过寄存器获取相应的运算结果，补全后代码如下：

```
reg[3:0] Control;
reg[31:0] A, B;

always@(posedge clk) begin
    if(EN & ~state) begin // state == 0
        A <= ALUA;
        B <= ALUB;
        Control <= ALUControl;           //to fill sth.in
        state <= 1;
    end
    else state <= 0;
end
```

ALU 仅需一个时钟周期就可以执行完成。

2. 访存单元（FU_mem.v 中，操作结束时 state[0] == 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) begin
        bhw_reg <= bhw;
        mem_w_reg <= mem_w;
        rs1_data_reg <= rs1_data;
        rs2_data_reg <= rs2_data;
        imm_reg <= imm;
        state[1] <= 1'b1;
    end
    else
        state <= state >> 1'b1;
end

assign addr = rs1_data_reg + imm_reg;         //to fill sth.in
```

访存单元需 2 个时钟周期执行完成。

3. 乘法单元（FU_mul.v 中，操作结束时 state[0] == 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：

```
wire [63:0] mulres;

always @(posedge clk) begin
    if (EN && ~state) begin
```

```
        A_reg <= A;
        B_reg <= B;
        state[6] <= 1'b1;
    end else
        state <= state >> 1'b1;
end         //to fill sth.in
```

乘法单元需 7 个时钟周期执行完成。

4. 除法单元（FU_div.v 中，操作结束信号由除法器返回，状态 state 仅表示除法器是否在工作中，state == 1 表示在工作）

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，代码如下：

```
wire res_valid;
wire[63:0] divres;

reg state;
assign finish = res_valid & state;
initial begin
    state = 0;
end

reg A_valid, B_valid;
reg[31:0] A_reg, B_reg;

always @(posedge clk) begin
    if (EN & ~state) begin
        A_reg <= A;
        A_valid <= 1'b1;
        B_reg <= B;
        B_valid <= B ? 1'b1 : 1'b0;
        state <= 1;
    end else if (finish)
        state <= 0;
end            //to fill sth.in
```

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。

5. 跳转单元（在 FU_jump.v 中，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，同时还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：

```
always @(posedge clk) begin
    if (EN & ~state) begin
       state <= 1'b1;
       rs1_data_reg <= rs1_data;
       rs2_data_reg <= rs2_data;
       imm_reg <= imm;
       PC_reg <= PC;
       cmp_ctrl_reg <= cmp_ctrl;
       JALR_reg <= JALR;
   end else
       state <= 1'b0;
end

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

assign PC_jump = JALR_reg ? (rs1_data_reg + imm_reg) : (PC_reg + imm_reg);
assign PC_wb = PC_reg + 3'd4;          //to fill sth.in
```

跳转单元只需 1 个时钟周期就执行完成。

（二）补充 CPU 各部分的连线

ImmGen 模块连接如下（很自然，不用多解释）：

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

为 ALU 选择运算数 A 时，当选择信号为 0 时，选择源寄存器 A 的值；当选择信号为 1 时，选择 PC 的值。故作如下连线：

```
MUX2T1_32 mux_imm_ALU_ID_A(.I0(rs1_data_ID),.I1(PC_ID),.s(ALUSrcA_ctrl),.o(ALUA_ID));
```

为 ALU 选择运算数 B 时，当选择信号为 0 时，选择源寄存器 B 的值；当选择信号为 1 时，选择立即数。故作如下连线：

```
MUX2T1_32 mux_imm_ALU_ID_B(.I0(rs2_data_ID),.I1(Imm_out_ID),.s(ALUSrcB_ctrl),.o(ALUB_ID));
```

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(.I0(),.I1(ALUout_WB),.I2(mem_data_WB),.I3(mulres_WB),
              .I4(divres_WB),.I5(PC_wb_WB),.I6(),.I7(),.s(DatatoReg_ctrl),.o(wt_data_WB));
```

四、 实验结果分析

使用如下指令进行测试：

NO.	Instruction	Addr.	Label	ASM	Comment
0	00000013	0	__start:	addi x0, x0, 0	
1	00402103	4		lw x2, 4(x0)	x2=0x8
2	00802203	8		lw x4, 8(x0)	x4=0x10
3	004100b3	C		add x1, x2, x4	x1=0x18
4	fff08193	10		addi x3, x1, -1	x3=0x17
5	00c02283	14		lw x5, 12(x0)	x5=0x14
6	01002303	18		lw x6, 16(x0)	x6=0xFFFF0000
7	01402383	1C		lw x7, 20(x0)	x7=0x0FFF0000

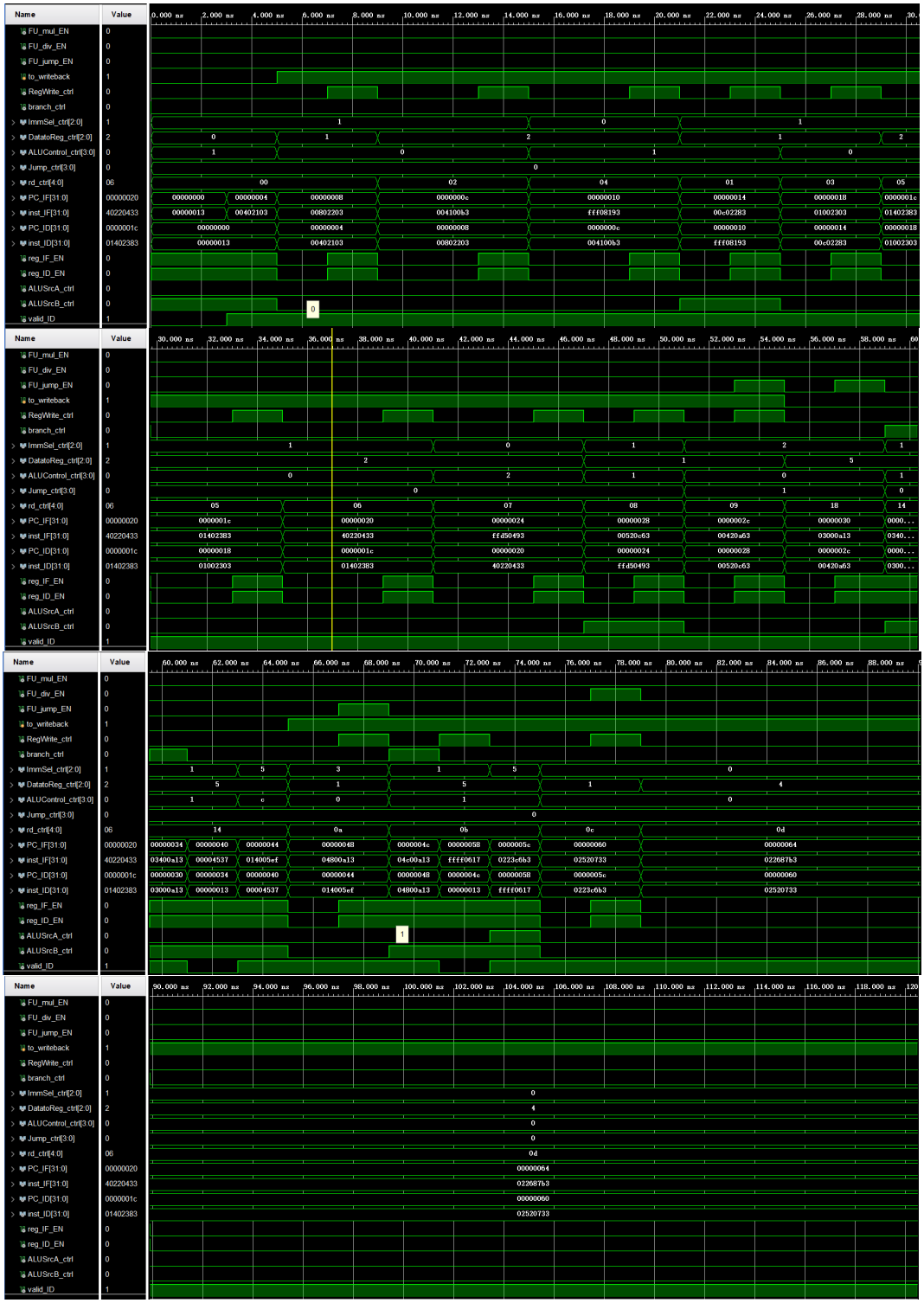
8	40220433	20		sub x8, x4, x2	x8=0x8
9	ffd50493	24		addi x9, x10,-3	x9=0xFFFFFFFD
10	00520c63	28		beq x4, x5, label0	no jump
11	00420a63	2C		beq x4, x4, label0	jump to 0x40
12	03000a13	30		addi x20,x0, 48	not executed
13	03400a13	34		addi x20,x0, 52	not executed
14	03800a13	38		addi x20,x0, 56	not executed
15	03c00a13	3C		addi x20,x0,60	not executed
16	00004537	40	label0:	lui x10,4	x10=0x4000
17	014005ef	44		jal x11,20	x11=0x48
18	04800a13	48		addi x20,x0,72	not executed
19	04c00a13	4C		addi x20,x0, 76	not executed
20	05000a13	50		addi x20,x0, 80	not executed
21	05400a13	54		addi x20,x0, 84	not executed
22	ffff0617	58		auipc x12, 0xffff0	x12=0xFFFF0058
23	0223c6b3	5C		div x13, x7, x2	x13=0x01FFE000
24	02520733	60		mul x14, x4, x5	x14=0x140
25	022687b3	64		mul x15, x13,x2	x15=0x0FFF0000
26	00400813	68		addi x16, x0, 4	x16=0x4
27	000008e7	6C		jalr x17, 0(x0)	x17=0x70

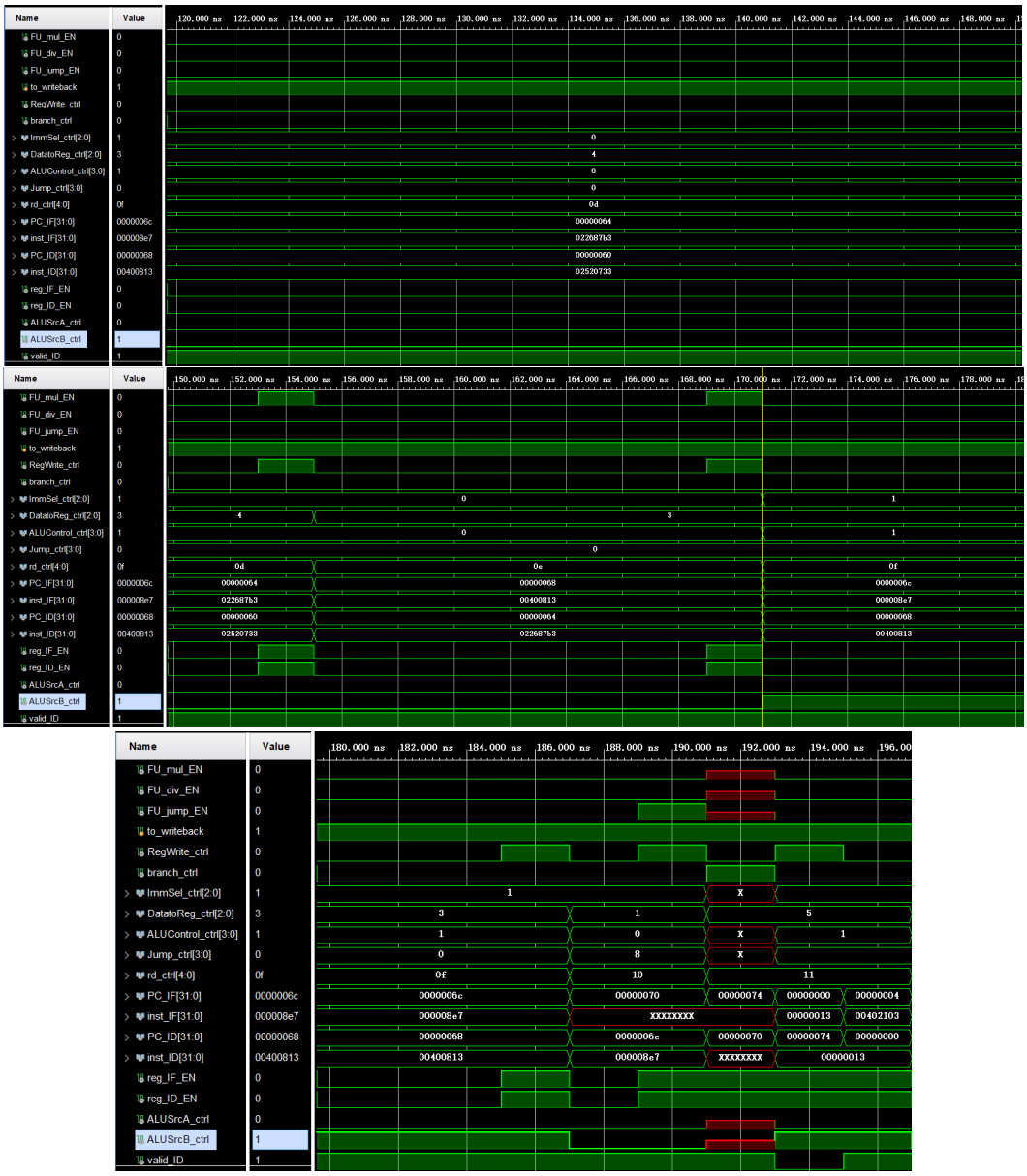
使用如下内存数据进行测试：

NO.	Data	Addr.	NO.	Data	Addr.
0	000080BF	0	16	00000000	40
1	00000008	4	17	00000000	44
2	00000010	8	18	00000000	48
3	00000014	C	19	00000000	4C
4	FFFF0000	10	20	A3000000	50
5	0FFF0000	14	21	27000000	54
6	FF000F0F	18	22	79000000	58
7	F0F0F0F0	1C	23	15100000	5C
8	00000000	20	24	00000000	60
9	00000000	24	25	00000000	64
10	00000000	28	26	00000000	68
11	00000000	2C	27	00000000	6C
12	00000000	30	28	00000000	70

13	00000000	34	29	00000000	74
14	00000000	38	30	00000000	78
15	00000000	3C	31	00000000	7C

仿真结果截图如下：





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。

对仿真中个指令的执行情况解读如下：

（1）3ns~5ns：尚未有指令在 FU 阶段。ID 阶段指令为 32’h00000013（addi x0, x0, 0），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。

（2）5ns~9ns：在 ID 阶段，指令为 32’h00402103（lw x2, 4(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’h00000013（addi x0, x0, 0）进入 ALU 被执行，这条指令的结果要写回寄存器 0，所以 rd_ctrl == 5’h0，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。

（3）9ns~15ns：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。

（4）15ns~21ns：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。

（5）21ns~25ns：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。

（6）25ns~29ns：在 ID 阶段，指令为 32'h00c02283（lw x5, 12(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'hfff08193（addi x3, x1, -1）进入 ALU 被执行，这条指令的结果要写回寄存器 3，所以 rd_ctrl == 5'h3，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。

（7）29ns~35ns：在 ID 阶段，指令为 32'h01002303（lw x6, 16(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'h00c02283（lw x5, 12(x0)）进入访存单元被执行，这条指令的结果要写回寄存器 5，所以 rd_ctrl == 5'h5，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。

（6）35ns~41ns：在 ID 阶段，指令为 32'h01402383（lw x7, 20(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'h01002303（lw x6, 16(x0)）进入访存单元被执行，这条指令的结果要写回寄存器 6，所以 rd_ctrl == 5'h6，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。

（7）41ns~47ns：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。

（8）47ns~51ns：在 ID 阶段，指令为 32'hffd50493（addi x9, x10, -3），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'h40220433（sub x8, x4, x2）进入 ALU 被执行，这条指令的结果要写回寄存器 8，所以 rd_ctrl == 5'h8，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。

（9）51ns~55ns：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。

（10）55ns~59ns：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。

（11）59ns~63ns：前两纳秒内，在 ID 阶段，指令为 32'h03000a13（addi x20, x0, 48），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'h00420a63（beq x4, x4, 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。易得 x4 == x4，所以要作跳转，branch_ctrl == 1，这使得 reg_IF_EN 和 reg_ID_EN 均维持为 1，使得 CPU 能够捕获新指令并清除 ID 阶段的指令。然后，在后两纳秒内，PC_IF 跳变为 label0 的地址 32'h00000040，从此处取得新的指令；清除 ID 阶段的指令，valid_ID 变为 0，因为 ID 阶段的指令已经无效了。同时 branch_ctrl 也变为 0，表示跳转结束了。

（12）63ns~65ns：此时尚未有有效指令在 FU 阶段。ID 阶段指令为 32'h00004537（lui x10, 4），只有一个操作数 B，来自立即数，所以 ALUSrcB_ctrl == 1；其对应的 ALUControl_ctrl = 12 == 4'hc；此指令为 U 型指令，所以对于立即数格式，有 ImmSel_ctrl == 5；此指令有效，故 valid_ID == 1。

（13）65ns~69ns：在 ID 阶段，指令为 32'h014005ef（jal x11, 20），此指令为 J 型指令，所以对于立即数格式，有 ImmSel_ctrl == 3；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'h00004537（lui x10, 4）进入 ALU 被执行，这条指令的结果要写回寄存器 10，所以 rd_ctrl == 10 == 5'ha，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。

（14）69ns~73ns：前两纳秒内，在 ID 阶段，指令为 32'h04800a13（addi x20, x0, 72），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'h014005ef（jal x11, 20）进入跳转单元被执行，这条指令的结果要写回寄存器 11，所以 rd_ctrl == 11 == 5'hb，to_writeback == 1；结果来自跳转单元，所以 DatatoReg_ctrl == 5。必作跳转，branch_ctrl == 1，这使得 reg_IF_EN 和 reg_ID_EN 均维持为 1，使得 CPU 能够捕获新指令并清除 ID 阶段的指令。然后，在后两纳秒内，PC_IF 在 jal x11, 20 指令的地址上增加 20 变为 32'h00000058，从此处取得新的指令；清除 ID 阶段的指令，valid_ID 变为 0，因为 ID 阶段的指令已经无效了；branch_ctrl 也变为 0，表示跳转结束了；指令的结果计算完成，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。

（15）73ns~75ns：此时尚未有有效指令在 FU 阶段。ID 阶段指令为 32'hffff0617（auipc x12, 0xffff0），操作数 A 来自该指令的 PC，所以 ALUSrcA_ctrl == 1；操作数 B 来自立即数，所以 ALUSrcB_ctrl == 1；使用加法，所以 ALUControl_ctrl = 12 == 4'hc；此指令为 U 型指令，所以对于立即数格式，有 ImmSel_ctrl == 5；此指令有效，故 valid_ID == 1。

（16）75ns~79ns：在 ID 阶段，指令为 32'h0223c6b3（div x13, x7, x2），所以 FU_div_EN 在最后一个时钟周期被设置为 1，使该指令之后进入除法单元执行；此指令有效，故 valid_ID == 1。在 FU 阶段，指令 32'hffff0617（auipc x12, 0xffff0）进入 ALU 被执行，这条指令的结果要写回寄存器 12，所以 rd_ctrl == 12 == 5'hc，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。

（17）79ns~155ns：在 ID 阶段，指令为 32'h02520733（mul x14, 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。

（18）155ns~171ns：在 ID 阶段，指令为 32'h022687b3（mul x15, x13, x2），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'h02520733（mul x14, x4, x5）进入乘法单元被执行，这条指令的结果要写回寄存器 14，所以 rd_ctrl == 14 == 5'he，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，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。

（19）171ns~187ns：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。

（20）187ns~191ns：在 ID 阶段，指令为 32'h000008e7（jalr x17, 0(x0)），此指令为 I 型指令，所以对于立即数格式，有 ImmSel_ctrl == 1；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'h00400813（addi x16, x0, 4）进入 ALU 被执行，这条指令的结果要写回寄存器 16，所以 rd_ctrl == 16 == 5'h10，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。

（21）192ns~196ns：前两纳秒内，ID 阶段已无指令；在 FU 阶段，指令 32'h000008e7（jalr x17, 0(x0)）进入跳转单元被执行，这条指令的结果要写回寄存器 17，所以 rd_ctrl == 17 == 5'h11，to_writeback == 1；结果来自跳转单元，所以 DatatoReg_ctrl == 5。必作跳转，branch_ctrl == 1，这使得 reg_IF_EN 和 reg_ID_EN 均维持为 1，使得 CPU 能够捕获新指令并清除 ID 阶段的指令。然后，在后两纳秒内，PC_IF 变为 x0 + 0 = 32'h00000000，取得此处指令；清除 ID 阶段的指令，valid_ID 变为 0，因为 ID 阶段的指令已经无效了（虽然本来也没有指令）；branch_ctrl 也变为 0，表示跳转结束了；指令的结果计算完成，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。

至此，程序的一个循环被执行完毕了。

在物理验证时，无论程序跑多久，只要使程序跑完两个循环及以上， x1~x17 各寄存器的值就是固定的，就可以通过这些寄存器的值是否正确来直接判断 CPU

设计是否正确。

物理验证时，程序跑完几个循环后，屏幕显示如下：

各寄存器值正确，说明 CPU 设计正确。

五、　讨论与心得

通过这次实验，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。