

《计算机组成与设计》

实验报告

姓名: 刘韬
学院: 竺可桢学院
专业: 人工智能
邮箱: 3220103422@zju.edu.cn

报告日期: 2024-06-13

1 不解决冲突的五级流水线实现

1.1 数据通路设计

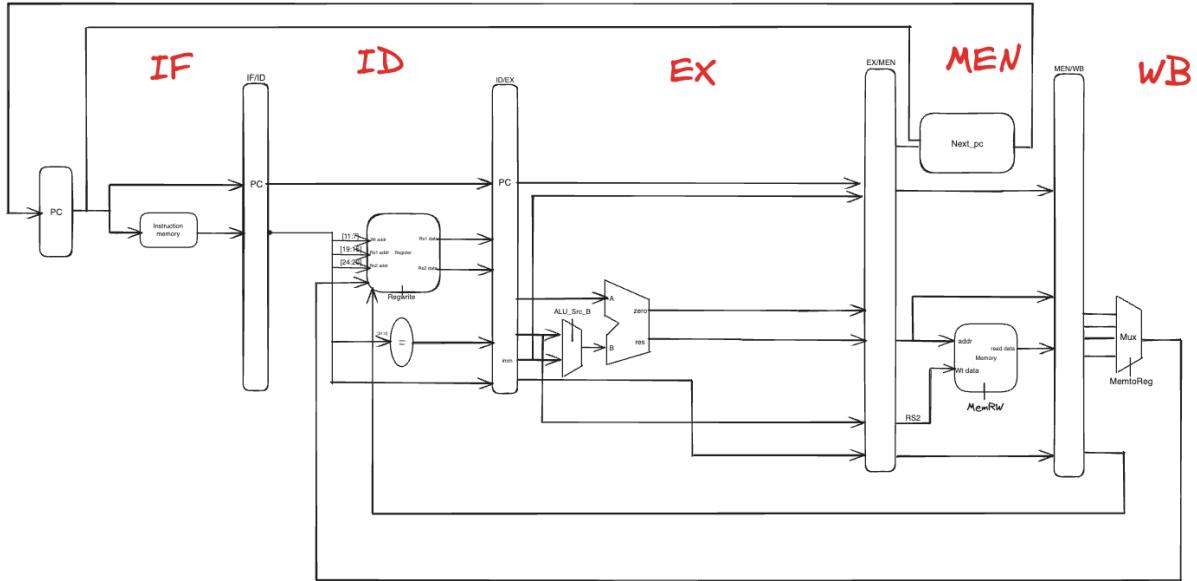


图 1：数据通路

相较于之前单周期 CPU 的设计，流水线 CPU 的主要变化在于每个 stage 的值都需要保存，以便在下一个周期使用。这里我们实现一个 5 级流水线，分别是 IF (Instruction Fetch)、ID (Instruction Decode)、EX (Execution)、MEM (Memory)、WB (Write Back)。在两个 stage 之间的寄存器用于存储这个 stage 产生的需要到后面 stage 使用的数据和信号，并且每个寄存器在 CPU 时钟周期的上升沿进行数据更新，达到了流水线的效果。上图中省略了一些具体的信号传递，在后面具体的模块解析中会进行详细的说明。

1.2 模块代码

考虑到数据通路有了较大的变化，添加了许多寄存器，我们将原先的 datapath.v 直接写入 CPU.v 中，直接在 CPU.v 中实现整个 CPU 的数据通路。

1.2.1 IF 段

IF 段的主要工作是根据 PC 值获取指令，由 PC 和 IF/ID 寄存器两个时序电路分隔开。需要传递到下个 stage 的数据有 pc 值和 inst 值。这里的 PC_next 由 MEN stage 中的 NEXT_PC 产生，inst 通过 ROM 读入。PC 一直保持能写状态，维持更新。

代码如下：

```
1 // IF stage
2 wire [31:0] PC_next;
3 REG32 PC(
4     .clk(clk),
```

```

5      .rst(rst),
6      .CE(1'b1),
7      .D(PC_next),
8      .Q(PC_out)
9  );
10
11 // IF/ID latch
12 reg [31:0] IF_ID_PC;
13 reg [31:0] IF_ID_inst;
14
15 always @(posedge clk or posedge rst) begin
16     if (rst) begin
17         IF_ID_PC <= 32'b0;
18         IF_ID_inst <= 32'b0;
19     end
20     else begin
21         IF_ID_PC <= PC_out;
22         IF_ID_inst <= inst_field;
23     end
24 end

```

1.2.2 ID 段

ID 阶段需要进行寄存器的访问，立即数的生成和指令译码（即控制单元生成控制信号）。需要传递给 ID/EX 寄存器的数据有

PC,inst,rs1_data,rs2_data,imm.write_data

需要传递的控制信号有

ALU_Control,wordtype,MemtoReg,Jump,Branch0,Branch1,RegWrite,MemRW,jalr,utype,ALUSrc_B

根据上面描述，下面是 ID 段的代码：

```

1  // ID stage
2  // SCPU_ctrl
3  wire [3:0] ALU_Control;
4  wire [2:0] ImmSel, wordtype;
5  wire [1:0] MemtoReg;
6  wire Jump, Branch0, Branch1, RegWrite, MemRW, jalr, utype, ALUSrc_B;
7
8  SCPU_ctrl control(
9    .OPcode(IF_ID_inst[6:2]),
10   .Fun3(IF_ID_inst[14:12]),
11   .Fun7(IF_ID_inst[30]),
12   .MIO_ready(1'b1),
13   .ImmSel(ImmSel),
14   .ALUSrc_B(ALUSrc_B),
15   .MemtoReg(MemtoReg),
16   .Jump(Jump),
17   .Branch0(Branch0),
18   .Branch1(Branch1),
19   .RegWrite(RegWrite),
20   .MemRW(MemRW),
21   .ALU_Control(ALU_Control),
22   .jalr(jalr),
23   .utype(utype),
24   .wordtype(wordtype)
25 );

```

```

26
27     // IMMGEM
28     wire [31:0] Imm_out;
29
30     ImmGen_ImmGen(
31         .ImmSel(ImmSel),
32         .inst_field(IF_ID_inst),
33         .Imm_out(Imm_out)
34     );
35
36     // Register File
37     wire [31:0] Rs1_data, Rs2_data;
38
39     Regs_Regs(
40         .clk(clk),
41         .rst(rst),
42         .Rs1_addr(IF_ID_inst[19:15]),
43         .Rs2_addr(IF_ID_inst[24:20]),
44         .Wt_addr(MEM_WB_wt_addr),
45         .Wt_data(write_data),
46         `RegFile_Regs_Arguments
47         .RegWrite(MEM_WB_RegWrite),
48         .Rs1_data(Rs1_data),
49         .Rs2_data(Rs2_data)
50     );

```

需要注意的是，此时使用的数据应是上一个 stage 的数据，所以需要使用 IF/ID 寄存器中的数据。另外在 Regs 中与写回相关的几个数据来自 WB 阶段，所以需要使用 MEM/WB 寄存器中的数据。

下面是 ID/EX 寄存器的代码：

```

1 // ID/EX latch
2     reg [31:0] ID_EX_data1;
3     reg [31:0] ID_EX_data2;
4     reg [4:0] ID_EX_wt_addr;
5     reg [31:0] ID_EX_PC;
6     reg [31:0] ID_EX_imm;
7
8     reg [3:0] ID_EX_ALU_Control;
9     reg ID_EX_ALUSrc_B;
10    reg ID_EX_Jump;
11    reg ID_EX_Branch0;
12    reg ID_EX_Branch1;
13    reg ID_EX_RegWrite;
14    reg ID_EX_MemRW;
15    reg ID_EX_jalr;
16    reg [1:0] ID_EX_MemtoReg;
17    reg [2:0] ID_EX_wordtype;
18    reg ID_EX_utype;
19
20    always @(posedge clk or posedge rst) begin
21        if(rst)begin
22            ID_EX_data1 <= 32'b0;
23            ID_EX_data2 <= 32'b0;
24            ID_EX_wt_addr <= 5'b0;
25            ID_EX_PC <= 32'b0;
26            ID_EX_imm <= 32'b0;
27            ID_EX_ALU_Control <= 4'b0;
28            ID_EX_ALUSrc_B <= 1'b0;

```

```

29      ID_EX_Jump <= 1'b0;
30      ID_EX_Branch0 <= 1'b0;
31      ID_EX_Branch1 <= 1'b0;
32      ID_EX_RegWrite <= 1'b0;
33      ID_EX_MemRW <= 1'b0;
34      ID_EX_jalr <= 1'b0;
35      ID_EX_MemtoReg <= 2'b0;
36      ID_EX_wordtype <= 3'b0;
37      ID_EX_utype <= 1'b0;
38  end
39 else begin
40     ID_EX_data1 <= Rs1_data;
41     ID_EX_data2 <= Rs2_data;
42     ID_EX_wt_addr <= IF_ID_inst[11:7];
43     ID_EX_PC <= IF_ID_PC;
44     ID_EX_imm <= Imm_out;
45     ID_EX_ALU_Control <= ALU_Control;
46     ID_EX_ALUSrc_B <= ALUSrc_B;
47     ID_EX_Jump <= Jump;
48     ID_EX_Branch0 <= Branch0;
49     ID_EX_Branch1 <= Branch1;
50     ID_EX_RegWrite <= RegWrite;
51     ID_EX_MemRW <= MemRW;
52     ID_EX_jalr <= jalr;
53     ID_EX_MemtoReg <= MemtoReg;
54     ID_EX_wordtype <= wordtype;
55     ID_EX_utype <= utype;
56 end
57 end

```

1.2.3 EX 段

EX 段主要进行 ALU 计算，需要传递给 EX/MEM 寄存器的数据有这阶段输出的值和下一阶段仍需要的前面传过来的值：

```
pc,ALU_out,Rs2_data,imm,write_addr,zero
```

这一阶段用掉的控制信号有 `ALUSrc_B,ALU_Control`，因此需要传递到下一阶段的控制信号

```
wordtype,MemtoReg,Jump,Branch0,Branch1,RegWrite,MemRW,jalr,utype
```

下面是 EX 段的代码：

```

1 // EX stage
2 // ALU
3 wire [31:0] ALU_B;
4 assign ALU_B = (ID_EX_ALUSrc_B) ? ID_EX_imm : ID_EX_data2;
5 wire [31:0] ALU_out;
6 wire zero;
7
8 ALU ALU(
9     .A(ID_EX_data1),
10    .B(ALU_B),
11    .ALU_operation(ID_EX_ALU_Control),
12    .res(ALU_out),
13    .zero(zero)
14 );

```

需要注意的是使用的数据和信号都应该是上一阶段传递而来的即 ID/EX 寄存器中的数据。下面是 EX/MEM 寄存器的代码：

```

1 // EX/MEM latch
2 reg [31:0] EX_MEM_ALU_out;
3 reg [31:0] EX_MEM_data2;
4 reg [4:0] EX_MEM_wt_addr;
5 reg [31:0] EX_MEM_PC;
6 reg [31:0] EX_MEM_imm;
7 reg EX_MEM_zero;
8
9 reg EX_MEM_Jump;
10 reg EX_MEM_Branch0;
11 reg EX_MEM_Branch1;
12 reg EX_MEM_RegWrite;
13 reg EX_MEM_MemRW;
14 reg EX_MEM_jalr;
15 reg EX_MEM_utype;
16 reg [1:0] EX_MEM_MemtoReg;
17 reg [2:0] EX_MEM_wordtype;
18
19 always @(posedge clk or posedge rst) begin
20     if(rst)begin
21         EX_MEM_ALU_out <= 32'b0;
22         EX_MEM_data2 <= 32'b0;
23         EX_MEM_wt_addr <= 5'b0;
24         EX_MEM_PC <= 32'b0;
25         EX_MEM_imm <= 32'b0;
26         EX_MEM_zero <= 1'b0;
27         EX_MEM_Jump <= 1'b0;
28         EX_MEM_Branch0 <= 1'b0;
29         EX_MEM_Branch1 <= 1'b0;
30         EX_MEM_RegWrite <= 1'b0;
31         EX_MEM_MemRW <= 1'b0;
32         EX_MEM_jalr <= 1'b0;
33         EX_MEM_MemtoReg <= 2'b0;
34         EX_MEM_wordtype <= 3'b0;
35         EX_MEM_utype <= 1'b0;
36     end
37     else begin
38         EX_MEM_ALU_out <= ALU_out;
39         EX_MEM_data2 <= ID_EX_data2;
40         EX_MEM_wt_addr <= ID_EX_wt_addr;
41         EX_MEM_PC <= ID_EX_PC;
42         EX_MEM_imm <= ID_EX_imm;
43         EX_MEM_zero <= zero;
44         EX_MEM_Jump <= ID_EX_Jump;
45         EX_MEM_Branch0 <= ID_EX_Branch0;
46         EX_MEM_Branch1 <= ID_EX_Branch1;
47         EX_MEM_RegWrite <= ID_EX_RegWrite;
48         EX_MEM_MemRW <= ID_EX_MemRW;
49         EX_MEM_jalr <= ID_EX_jalr;
50         EX_MEM_MemtoReg <= ID_EX_MemtoReg;
51         EX_MEM_wordtype <= ID_EX_wordtype;
52         EX_MEM_utype <= ID_EX_utype;
53     end
54 end

```

1.2.4 MEM 段

MEM 段主要进行内存访问，涉及到内存访问的定义不在 SCPU 内部，因此我们只需要给出读取的地址，和写入的数据以及写使能即可。这里我们为了实现 `lw,lb,lh,lbu,lhu` 指令，

对读入的数据和要给出的数据进行了一些处理，这里的处理与之前的单周期 CPU 的处理方式一致。主要是根据写入的数据的长度和写入的地址进行数据的处理。

```
1 // MEM stage
2 // Memory
3 wire [31:0] memory_in;
4 assign memory_in = Data_in >> ({3'b000, EX_MEM_ALU_out[1:0]} << 2'b11);
5 assign Addr_out = EX_MEM_ALU_out;
6 //input
7 always @(*) begin
8     case(EX_MEM_wordtype)
9         3'b000 : reg_in <= {{24{memory_in[7]}}, memory_in[7:0]}; //
10        signed byte
11         3'b100 : reg_in <= {24'b0, memory_in[7:0]}; //unsigned byte
12         3'b001 : reg_in <= {{16{memory_in[15]}}, memory_in[15:0]}; //
13        signed half
14         3'b101 : reg_in <= {16'b0, memory_in[15:0]}; //unsigned half
15         3'b010 : reg_in <= memory_in; //signed word
16         default: reg_in <= 32'b0;
17     endcase
18 end
19 assign Mem_write = EX_MEM_MemRW;
20 //store
21 always @(*) begin
22     case(EX_MEM_wordtype[2:0])
23         3'b000 : begin
24             case(EX_MEM_ALU_out[1:0])
25                 2'b00 : begin
26                     Data_out <= {24'b0, EX_MEMORY_data2[7:0]};
27                     RAM_wt <= 4'b0001;
28                 end
29                 2'b01 : begin
30                     Data_out <= {16'b0, EX_MEMORY_data2[7:0], 8'b0};
31                     RAM_wt <= 4'b0010;
32                 end
33                 2'b10 : begin
34                     Data_out <= {8'b0, EX_MEMORY_data2[7:0], 16'b0};
35                     RAM_wt <= 4'b0100;
36                 end
37                 2'b11 : begin
38                     Data_out <= {EX_MEMORY_data2[7:0], 24'b0};
39                     RAM_wt <= 4'b1000;
40                 end
41             endcase
42         3'b001 : begin
43             case(EX_MEMORY_ALU_out[1:0])
44                 2'b00 : begin
45                     Data_out <= {16'b0, EX_MEMORY_data2[15:0]};
46                     RAM_wt <= 4'b0011;
47                 end
48                 2'b01 : begin
49                     Data_out <= {8'b0, EX_MEMORY_data2[15:0], 8'b0};
50                     RAM_wt <= 4'b0110;
51                 end
52                 2'b10 : begin
53                     Data_out <= {EX_MEMORY_data2[15:0], 16'b0};
54                     RAM_wt <= 4'b1100;
55                 end
56             default : begin
57                 Data_out <= 0;
58                 RAM_wt <= 0;
59             end
60         end
61     end
62 end
```

```

58           end
59       endcase
60   end
61   3'b010 : begin
62       Data_out <= EX_MEM_data2;
63       RAM_wt <= 4'b1111;
64   end
65   default : begin
66       Data_out <= 0;
67       RAM_wt <= 0;
68   end
69 endcase
70 end

```

需要再次强调的是，这里使用的数据是这个周期读入的数据和上一个周期传递过来的数据，因此需要使用 EX/MEM 寄存器中的数据。

同时我们在这个阶段完成了 PC_next 的计算，模块实现如下：

```

1 module PC_NEXT(
2     input [31:0]current_PC,
3     input [31:0]latch_PC,
4     input [31:0]Imm,
5     input [31:0]ALU_out,
6     input zero,
7     input Branch0,
8     input Branch1,
9     input Jump,
10    input jalr,
11    output [31:0]PC_next
12 );
13 wire [31:0] PC_imm;
14 wire s4;
15 assign s4 = (Branch0 & zero) | (Branch1 & ~zero);
16 assign PC_imm = latch_PC + Imm;
17 assign PC_next = (Jump == 1) ? ((jalr == 1) ? ALU_out : PC_imm) : ((s4
18 == 1) ? PC_imm : current_PC + 32'd4);
19 endmodule

```

这是对之前的单周期 CPU 中的计算下一 PC 值的改进，进行了模块的封装，也是因为此时写入 rd 数据的计算有所不同，这里计算得到的数据不能够进行复用。还需要注意的一点是这里用的 PC+4 是当前状态的 PC 而不是寄存器一级一级传递得到的 PC，这样才是流水线 CPU 的实现。

下面是 MEM/WB 寄存器的代码：

```

1 // MEM/WB latch
2 reg [31:0] MEM_WB_ALU_out;
3 reg [31:0] MEM_WB_reg_in;
4 reg [31:0] MEM_WB_imm;
5 reg [31:0] MEM_WB_PC;
6 reg [4:0] MEM_WB_wt_addr;
7
8 reg MEM_WB_RegWrite;
9 reg [1:0] MEM_WB_MemtoReg;
10 reg MEM_WB_utype;
11
12 wire [31:0] write_data;

```

```

13
14    always @(posedge clk or posedge rst) begin
15        if(rst)begin
16            MEM_WB_ALU_out <= 32'b0;
17            MEM_WB_reg_in <= 32'b0;
18            MEM_WB_imm <= 32'b0;
19            MEM_WB_PC <= 32'b0;
20            MEM_WB_wt_addr <= 5'b0;
21            MEM_WB_RegWrite <= 1'b0;
22            MEM_WB_MemtoReg <= 2'b0;
23            MEM_WB_utype <= 1'b0;
24        end
25        else begin
26            MEM_WB_ALU_out <= EX_MEMORY_ALU_out;
27            MEM_WB_reg_in <= reg_in;
28            MEM_WB_imm <= EX_MEMORY_imm;
29            MEM_WB_PC <= EX_MEMORY_PC;
30            MEM_WB_wt_addr <= EX_MEMORY_wt_addr;
31            MEM_WB_RegWrite <= EX_MEMORY_RegWrite;
32            MEM_WB_MemtoReg <= EX_MEMORY_MemtoReg;
33            MEM_WB_utype <= EX_MEMORY_utype;
34        end
35    end

```

1.2.5 WB 段

这一阶段进行寄存器堆的写回操作，先进行写回数据的计算与选择；根据 Risc-V 32I 中所列举的，写回寄存器的值有五个来源，分别是 `alu_out,mem_out,pc+4,imm,pc+imm`，这里我们根据 `MemtoReg` 信号进行选择。同时我们需要根据 `RegWrite` 信号进行写入操作。最后的 `imm` 和 `pc+imm` 是为了实现 U-type 指令的，为此我们将其在 `MemtoReg` 中的编码统一，通过之前产生的额外信号 `utype` 进行区分。

```

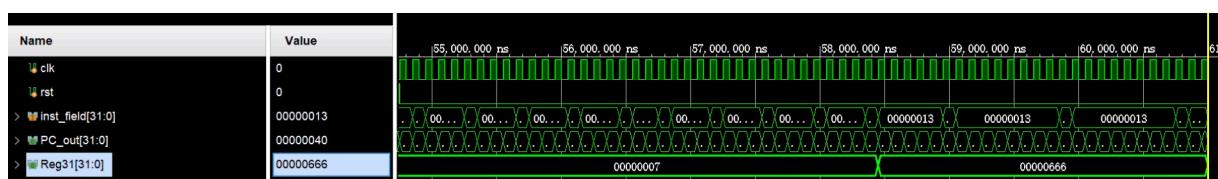
1 // WB stage
2 MUX4T1_32 m3(
3     .I0(MEM_WB_ALU_out),
4     .I1(MEM_WB_reg_in),
5     .I2(MEM_WB_PC + 32'd4),
6     .I3((MEM_WB_utype == 1) ? (MEM_WB_PC + MEM_WB_imm) : MEM_WB_imm),
7     .s(MEM_WB_MemtoReg),
8     .o(write_data)
9 );

```

这里的 `write_data` 已经在前面接入了寄存器堆。

1.3 仿真测试

使用提供的测试文件进行仿真，可以看到仿真结果如下：



查看 `reg31` 寄存器的值，可以看到结果是 666，因此仿真验证通过。

1.4 下板验证

将信号接入串口，通过串口查看输出结果，可以看到输出结果与仿真结果一致。

```
RV32I Single Cycle CPU

pc: 0000003C    inst: 00000013

x0: 00000000    ra: FFFFFFFF    sp: 00000000    gp: 40000000    tp: 40000000
t0: F8000000    t1: C0000000    t2: 80000000    s0: 00000001    s1: 00000001
a0: 00000000    a1: C0000000    a2: 00000001    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000020    s3: 00000000
s4: 0000009EC   s5: 0000009EC   s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000   s11: 000000D0   t3: 000000D0    t4: 00D0CBA0
t5: 00000A28   t6: 00000666

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 01 reg_i_data: 00000000    reg_wen: 0

is_imm: 0      is_auipc: 0     is_lui: 0       imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 1    cmp_ctrl: 0
alu_res: 00000000           cmp_res: 0

is_branch: 0    is_jal: 0      is_jalr: 0
do_branch: 0    pc_branch: 00000000

mem_wen: 0      mem_ren: 0
dmem_o_data: F0000000   dmem_i_data: 00000000   dmem_addr: 00000000

csr_wen: 0      csr_ind: 000    csr_ctrl: 0      csr_r_data: 00000000
mstatus: 00000000   mcause: 00000000   mepc: 00000000   mtval: 00000000
mtvec: 00000000   mie: 00000000   mip: 00000000
```

2 解决冲突的五级流水线实现

2.1 模块实现

2.1.1 PC 计算前移

首先是 datapath 发生的变化，我们将产生 PC_next 的部分前移到了 ID 阶段，这样我们就只需要等待一个周期就可以获得新的 PC 值，这样在执行分支指令和跳转指令的时候就只需要 bubble 一个周期，这在实现上提供了一些便利。

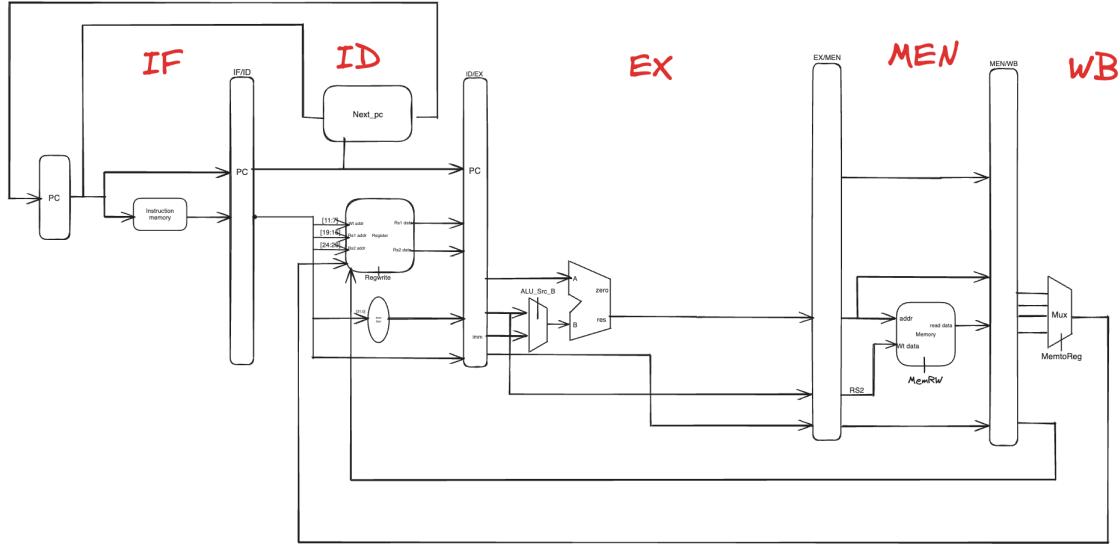


图 4: 修改后的流水线 datapath

2.1.2 解决数据冲突 (data hazard)

数据冲突发生的原因是：当前指令读取的寄存器值在上一条或者上上一条指令中发生了变化但还未来得及写回到寄存器堆中，这导致了数据的不一致。我们在检测到这种情况以后，可以通过暂停流水线即 stall 的方式进行解决，也可以直接将产生的数据通过 forwarding 的方式传递给需要的地方，这样就不需要暂停流水线，这样可以提高流水线的效率。我们采用 forwarding 的方式解决数据冲突：

我们的 forwarding 分为两个部分，一部分是传递给 ALU 的，用作数据处理的，另一部分是传递给 Next_PC 的，用来生成下一条指令的 PC 值，这一部分也涉及到了控制冲突的处理，我们统一放在下一部分进行讨论。解决 ALU 数据冲突的思路是：

1. 修改 ALU 的输入数据源，根据各阶段的 MemtoReg 信号和写回寄存器的地址，选择需要的数据源。当 MEM 阶段和 EX 阶段同时冲突时，选择 EX 阶段的数据，因为这是最新的数据。
2. 通过 bubble_stop 信号判断是否发生 load-use 的情况，如果发生了，就需要 bubble 一个周期，等待 load 指令加载完毕。
3. bubble 的做法是暂停 IF, ID 两个阶段，并且把 ID 阶段的控制信号全部置 0，后面的阶段照常进行。

对于第一部分，我们通过下面的方式检测是否发生数据冲突：

```

1 module forwarding(
2     input [4:0] EX_MEM_rd,
3     input [4:0] MEM_WB_rd,
4     input [4:0] ID_EX_rs1,
5     input [4:0] ID_EX_rs2,
6     input EX_MEM_RegWrite,
7     input MEM_WB_RegWrite,
8     output reg [1:0] forwardA, //00: no forward, 01: forward from MEM_WB,
9     output reg [1:0] forwardB

```

```

10 );
11     always @(*) begin
12         if(EX_MEM_RegWrite && (EX_MEM_rd != 0) && (EX_MEM_rd == ID_EX_rs1))
13             forwardA <= 2'b10;
14         else if(MEM_WB_RegWrite && (MEM_WB_rd != 0) && (MEM_WB_rd ==
15             ID_EX_rs1))
16             forwardA <= 2'b01;
17         else
18             forwardA <= 2'b00;
19         if(EX_MEM_RegWrite && (EX_MEM_rd != 0) && (EX_MEM_rd == ID_EX_rs2))
20             forwardB <= 2'b10;
21         else if(MEM_WB_RegWrite && (MEM_WB_rd != 0) && (MEM_WB_rd ==
22             ID_EX_rs2))
23             forwardB <= 2'b01;
24         else
25             forwardB <= 2'b00;
26     end
27 endmodule

```

如果 ID 段的读取的寄存器和 EX 或 MEM 阶段要写入的寄存器一致，并且这个寄存器不是 0 号寄存器，那么就发生了数据冲突，这时我们就需要进行 forwarding。这个模块传出的信号告知具体是哪个地方发生了冲突。

我们发现，这里的写回值不仅仅是 ALU_OUT 这一个来源，还有可能是 PC+4(跳转指令)，imm(lui)，PC+imm(U-type 指令)，而这恰好是由 MemtoReg 信号控制的，而这个信号在原本不处理冲突的流水线设计中，直到最后一个 WB 阶段才会被使用，现在我们提前利用这个信号，将每个阶段中的产生的需要写回的值，也就是目标寄存器应该变成的值，传递给需要的阶段，那么这样就解决了数据冲突。

```

1 MUX4T1_32 m3(
2     .I0(MEM_WB_ALU_out),
3     .I1(MEM_WB_reg_in),
4     .I2(MEM_WB_PC + 32'd4),
5     .I3((MEM_WB_utype == 1) ? (MEM_WB_PC + MEM_WB_imm) : MEM_WB_imm),
6     .s(MEM_WB_MemtoReg),
7     .o(write_data)
8 );
9
10 wire [31:0] EX_MEMORY_write_data;
11 MUX4T1_32 m4(
12     .I0(EX_MEMORY_ALU_out),
13     .I1(32'b0),
14     .I2(EX_MEMORY_PC + 32'd4),
15     .I3((EX_MEMORY_utype == 1) ? (EX_MEMORY_PC + EX_MEMORY_imm) : EX_MEMORY_imm),
16     .s(EX_MEMORY_MemtoReg),
17     .o(EX_MEMORY_write_data)
18 );
19
20 MUX4T1_32 ALU_A(
21     .I0(ID_EX_data1),
22     .I1(write_data), // MEM_WB_data_out
23     .I2(EX_MEMORY_write_data),
24     .I3(32'b0),
25     .s(forwardA),
26     .o(ALU_data_A)
27 );

```

```

28
29 wire [31:0] ALU_data_B0;
30 MUX4T1_32 ALU_B(
31     .I0(ID_EX_data2),
32     .I1(write_data),
33     .I2(EX_MEM_write_data),
34     .I3(32'b0),
35     .s(forwardB),
36     .o(ALU_data_B0)
37 );

```

上面的 m3 和 m4 两个模块是用来选择需要写回的数据的，这里的 write_data 是在 WB 阶段产生的，而 EX_MEM_write_data 是在 MEM 阶段产生的，下面是根据 forward 的结果选择需要的数据。

以上的处理可以解决大部分的数据冲突，但是还有一种数据冲突是必须要 bubble_stop 才可以解决的。load-use 类的数据冲突，即在 load 指令后面的指令中使用了 load 指令的结果，这种情况下，我们需要 bubble 直到数据写回，这样才能保证数据的正确性。因为读取地址的产生在 ID 阶段，但是要到 MEM 阶段才能得到数据，这样就会导致数据的不一致。我们在 ID 阶段检测是否发生了这种情况，如果发生了，就需要 bubble 直到数据写回，这里就没有使用 forwarding 的方式，而是直接 bubble。

```

1 assign bubble_stop = (ID_EX_Mem_read && (ID_EX_wt_addr == IF_ID_inst[19:15]
2 || ID_EX_wt_addr == IF_ID_inst[24:20])
3 || ((EX_MEM_Mem_read) && (EX_MEM_wt_addr == IF_ID_inst[19:15] ||
4 EX_MEM_wt_addr == IF_ID_inst[24:20]))
5 || (branch && ID_EX_RegWrite && ID_EX_wt_addr != 0 && (ID_EX_wt_addr ==
IF_ID_inst[19:15] || ID_EX_wt_addr == IF_ID_inst[24:20]));

```

上面的代码利用了 MEM_read 信号，这个信号是在 cpu_ctrl 模块中产生的，当 load 指令的时候，这个信号会被置为 1，这样我们就可以检测是否发生了 load-use 类的数据冲突。我们检测一个周期和两个周期前发生 load 指令的情况，这样就可以保证数据的正确性。最后一部分是下面控制冲突中需要用到的 bubble_stop 信号。

这里的时序更新如下：

```

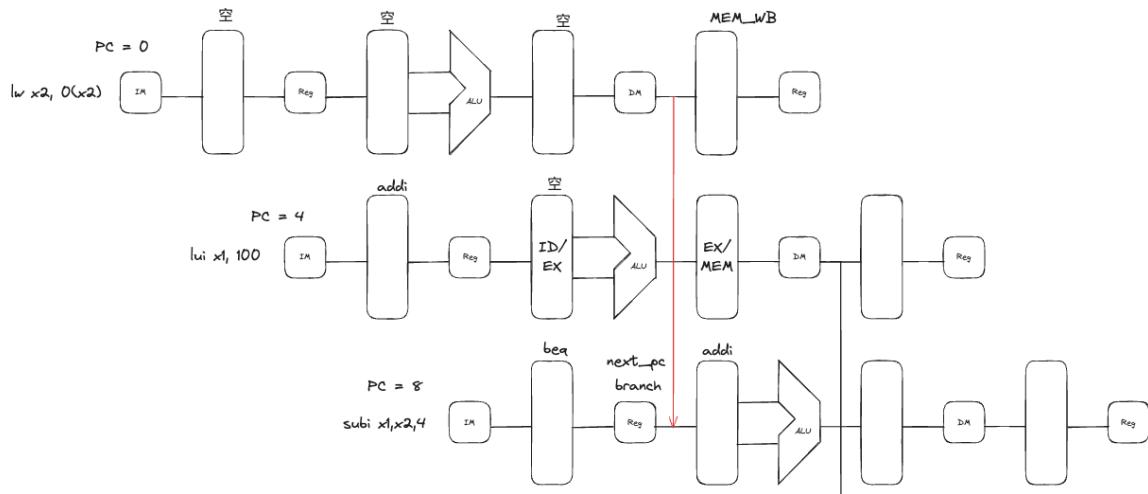
1 always @(posedge clk or posedge rst) begin
2     if(rst)begin
3         ...
4     end
5     else begin
6         else begin
7             if (bubble_stop) begin
8                 PC_out <= PC_out;
9
10                IF_ID_PC <= IF_ID_PC;
11                IF_ID_inst <= IF_ID_inst;
12
13                ID_EX_ALU_Control <= 4'b0;
14                ID_EX_ALUSrc_B <= 1'b0;
15                ID_EX_Jump <= 1'b0;
16                ID_EX_Branch0 <= 1'b0;

```

```

17   ID_EX_Branch1 <= 1'b0;
18   ID_EX_RegWrite <= 1'b0;
19   ID_EX_MemRW <= 1'b0;
20   ID_EX_jalr <= 1'b0;
21   ID_EX_MemtoReg <= 2'b0;
22   ID_EX_wordtype <= 3'b0;
23   ID_EX_utype <= 1'b0;
24   ID_EX_rs1 <= 5'b0;
25   ID_EX_rs2 <= 5'b0;
26   ID_EX_Mem_read <= 1'b0;
27 end
28 ...
29 end
30 end
31 end

```



实际上，在上一个指令 load 的情况下，我们可以不 stop，而是直接 forward，但是为了实现的方便，我们在这里 stop 一个周期。

2.1.3 解决控制冲突（control hazard）

我们还生成了一个 branch 信号来表示是否进行跳转指令，即 B,jal,jalr 的时候，branch 信号为 1。解决控制冲突的思路是：

生成 branch 信号后，此时跳转指令在 ID 阶段，我们保持 PC 更新，等到下一个时钟上升沿到来时，我们根据 branch 信号的值来决定 PC_next 的值。这里分为跳转和不跳转的两种情况；如果跳转的话，那么 PC_next 的值就是跳转的地址，否则的，PC_next 的值就是 current_PC。这是因为由于跳转指令进入 ID 阶段以后，在同时 PC 是按照上一个的 PC_next 来变化的，也就是说这条指令是下一条指令，如果不做跳转的话就应该按照顺序来执行这一条指令。另外如果遇到前面的数据冲突的情况，就还需要 bubble 一个周期。在 branch 信号为 1 时，PC 继续更新，IF_ID_inst 的值变为 nop，这是为了清空流水线，等到跳转指令结束以后才能继续流水线。

1. 在数据冲突并且不跳转的情况下，PC 将会停留三个周期；
2. 在数据冲突且跳转的情况下，PC 将会停留两个周期；
3. 在数据不冲突且不跳转的情况下，PC 将会停留两个周期；

4. 在数据不冲突且跳转的情况下，PC 将会停留一个周期。

下面是 PC_next 的模块中的 forwarding：因为 b-type 指令是否跳转需要根据寄存器值的比较来进行判断，因此这里也需要用到 forwarding。除此以外，跳转指令的目标地址的计算也是根据当前的 PC 值和 imm 值来进行计算的，这样就需要将 imm 值进行 forwarding。下面是 PC_next 模块的代码：

```
1 // PC_NEXT
2     wire [31:0] jalr_addr, jalr_addr;
3     wire [31:0] reg1, reg2;
4     assign reg1 = (branch && EX_MEM_RegWrite && (EX_MEM_wt_addr != 0) &&
5       (EX_MEM_wt_addr == IF_ID_inst[19:15])) ? EX_MEM_write_data
6       : (branch && MEM_WB_RegWrite && (MEM_WB_wt_addr != 0) &&
7       (MEM_WB_wt_addr == IF_ID_inst[19:15])) ? write_data : Rs1_data;
8     assign reg2 = (branch && EX_MEM_RegWrite && (EX_MEM_wt_addr != 0) &&
9       (EX_MEM_wt_addr == IF_ID_inst[24:20])) ? EX_MEM_write_data
10      : (branch && MEM_WB_RegWrite && (MEM_WB_wt_addr != 0) &&
11       (MEM_WB_wt_addr == IF_ID_inst[24:20])) ? write_data : Rs2_data;
12     assign jalr_addr = IF_ID_PC + Imm_out;
13     assign jalr_addr = reg1 + Imm_out;
14
15     assign s3 = (Branch0 & zero) | (Branch1 & ~zero);
16     assign PC_next = (s3 == 1) ? jalr_addr
17       : (Jump == 1) ? ((jalr == 1) ? jalr_addr : (jal_addr))
18       : (branch == 1) ? PC_out
19       : (PC_out + 32'd4);
```

上面的代码中，与之前同样的方式决定了 reg1 和 reg2 的值，并且计算了跳转地址。我们这里的 zero 是通过一个 ALU 计算得到的，这个 ALU 计算的输入是 ID 阶段的数据。对于这个 PC_next 的值，s3 代表我们要进行分支跳转，那么跳转地址就是 jalr_addr；如果是跳转指令，那么要根据是 JAL 还是 JALR 来决定跳转地址；这里的 branch 信号是在 cpu_ctrl 模块中产生的，当发生分支指令的时候，或者跳转指令的时候，我们暂停 PC，并且将 IF_ID_PC 的值变为 nop，即暂停一个周期。当下一个时钟上升沿到来时，cpu_ctrl 模块会检测到这个 nop，然后将 PC_next 的值变为 PC+4，此时的 PC 值是上一条指令执行的结果，这样就实现了跳转。

2.1.4 时序逻辑调整

前文讨论了 forwarding 的方法和计算 PC 值的模块。要解决流水线的冲突，我们还需要对时序逻辑进行调整，对时序的更新。之前已经展示了 bubble_stop 信号的处理，我们还生成了一个 branch 信号来表示是否进行跳转指令，对于这二者的优先级，我们优先处理 bubble_stop，这是因为还有 load-use 的情况，这种冲突一定是发生在分支指令之前的，而且 stall 一个周期对指令的执行没有影响，而且这样可以保证数据的正确性。因此我们在处理时序逻辑的时候，优先处理 bubble_stop，然后再处理 branch 信号。branch 信号做的事情是解析跳转指令后，ID 阶段的下个周期将被清空，这样就可以保证不会多执行一条指令。

```
1 if (bubble_stop) begin
2   ...
3 end
4 else if (branch) begin
```

```

5   PC_out <= PC_next;
6
7   IF_ID_PC <= PC_out;
8   IF_ID_inst <= 32'h00000013;
9
10  ID_EX_data1 <= Rs1_data;
11  ID_EX_data2 <= Rs2_data;
12  ID_EX_wt_addr <= IF_ID_inst[11:7];
13  ID_EX_PC <= IF_ID_PC;
14  ID_EX_imm <= Imm_out;
15  ID_EX_ALU_Control <= ALU_Control;
16  ID_EX_ALUSrc_B <= ALUSrc_B;
17  ID_EX_Jump <= Jump;
18  ID_EX_Branch0 <= Branch0;
19  ID_EX_Branch1 <= Branch1;
20  ID_EX_RegWrite <= RegWrite;
21  ID_EX_MemRW <= MemRW;
22  ID_EX_jalr <= jalr;
23  ID_EX_MemtoReg <= MemtoReg;
24  ID_EX_wordtype <= wordtype;
25  ID_EX_utype <= utype;
26  ID_EX_rs1 <= IF_ID_inst[19:15];
27  ID_EX_rs2 <= IF_ID_inst[24:20];
28  ID_EX_Mem_read <= mem_read;
29 end
30 else begin
31     ...          //正常更新流水线
32 end

```

2.2 仿真测试

为了测试我们的冲突解决策略是否正常实现，我们分别使用了尽可能多的冲突和单周期 CPU 的测试文件进行测试。前者可以验证我们解决冲突的策略和过程是否符合预期，后者说明我们的流水线和单周期 CPU 可以实现同样的功能。

2.2.1 冲突测试

以下是仿真代码：

```

1 test1:
2     addi x1, x0, 1
3     addi x2, x0, 1
4     addi x4, x0, 5
5 fibonacci:
6     add x3, x1, x2
7     add x1, x2, x3
8     add x2, x1, x3
9     addi x4, x4, -1
10    bne x0, x4, fibonacci
11    addi x5, x0, 0x63D
12    bne x2, x5, fail
13
14 test2:
15    addi x1, x0, 5
16    addi x2, x0, 0
17    addi x3, x0, 0x100
18    addi x5, x0, 4
19 memcpy:

```

```

20    beq x1, x0, exit1
21    lw x4, 0(x2)
22    sub x4, x4, x3
23    sw x4, 0(x3)
24    add x2, x2, x5
25    add x3, x3, x5
26    addi x1, x1, -1
27    bne x1, x0, memcp
28 exit1:
29    addi x1, x0, 5
30    addi x2, x0, 0
31    addi x3, x0, 0x100
32    addi x5, x0, 4
33 memcmp:
34    beq x1, x0, test3
35    lw x4, 0(x2)
36    sub x4, x4, x3
37    lw x6, 0(x3)
38    add x2, x2, x5
39    add x3, x3, x5
40    addi x1, x1, -1
41    bne x4, x6, fail
42    j memcmp
43
44
45 test3:
46    lui x1, 0xDEADB      # 0xDEADB000
47    ori x2, x0, 0xEF    # 0x000000EF
48    add x3, x1, x2      # 0xDEADB0EF
49    sub x1, x2, x1      # 0x215250EF
50    addi x2, x0, 1       # 0x00000001
51    srl x4, x3, x2      # 0x6F56D877
52    and x2, x1, x4      # 0x21525067
53    lui x1, 0x21525      # 0x21525000
54    addi x1, x1, 0x67    # 0x21525067
55    bne x2, x1, fail
56    addi x1, x0, 0xbc
57    jalr x1, x1, 0
58    addi x2, x0, 0xbc
59    bne x1, x2, fail
60
61 pass:
62    j pass
63
64
65 fail:
66    j fail
67

```

上面的测试代码如果没有异常，应该会进入 pass 标签，地址是 0xc4，如果有异常，会进入 fail 标签，地址是 0xc8。以下是仿真结果：

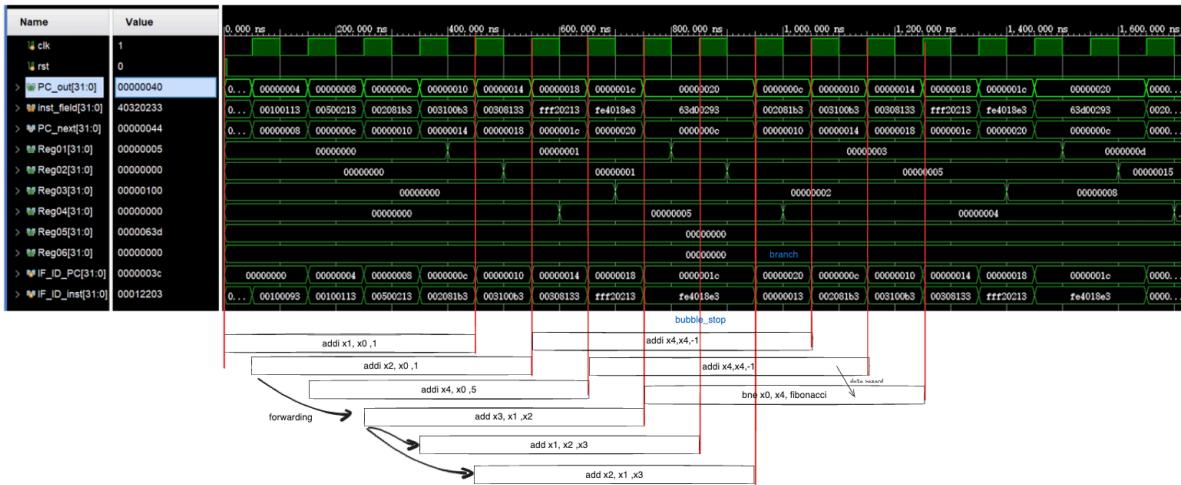


图 6: 仿真分析 1

我们是通过 forwarding 的方式进行，所以对 data hazard 并不会改变 PC 流，其显示与单周期相同。我们分析 PC 流发生变化，的指令部分。我们可以看到在 $\text{PC_out} = 20$ 的时候，PC 停留了一个周期，这是因为 $\text{PC_out} = 1C$ 的指令是 $\text{bne } x0, x4, -16$ ，并且这条指令还用到了前一条指令，这是发生了数据冲突，因此 bubble 了一个周期等待所需要的值计算完毕，进行判断，然后进行了跳转。

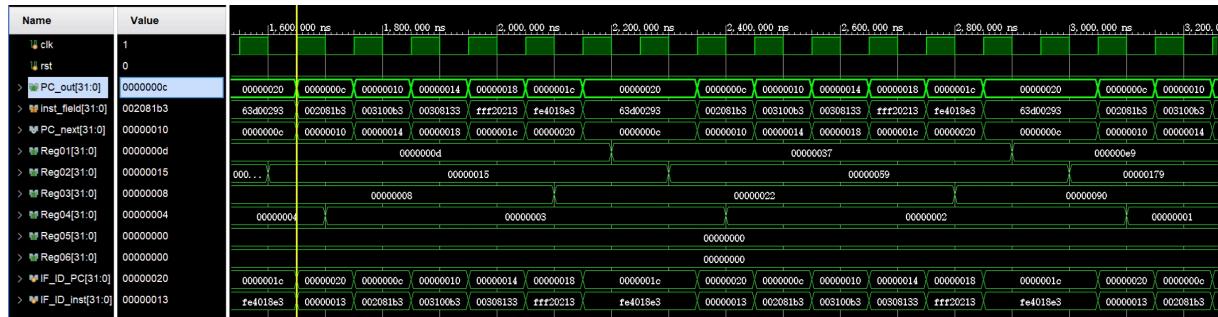


图 7: 仿真分析 2

上面是进行循环计算斐波那契数列，直到 reg4 为 0。

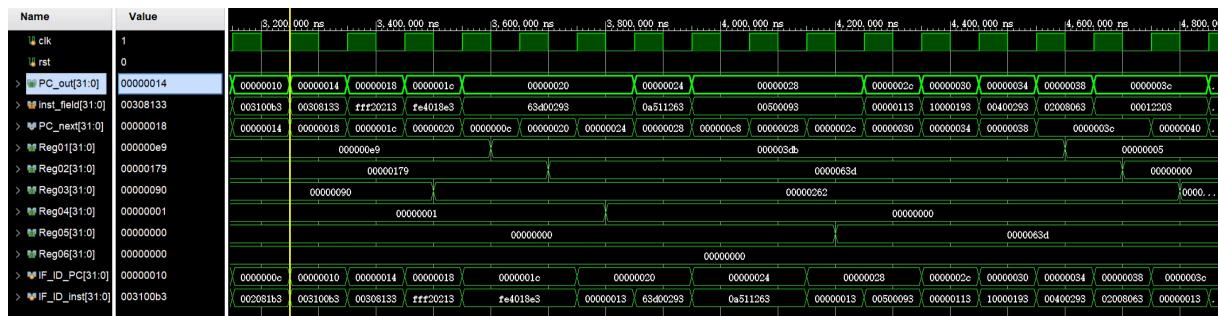


图 8: 仿真分析 3

上图中， $\text{PC_out} = 20$ 停留了三个周期，注意看下面的 IF_ID_inst ，在跳转指令停留了两个周期，这是因为发生了数据冲突，进行了 bubble_stop，接下来进行了跳转判断，将 IF_ID_inst 置为 nop，决定是不跳转，所以 PC_out 还是之前的 PC ，整体来看就是 PC_out 停留了三个周期。

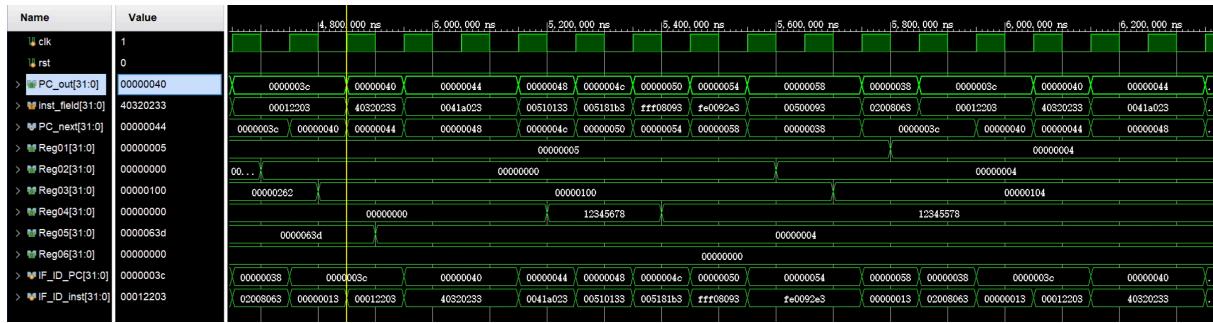


图 9: 仿真分析 4

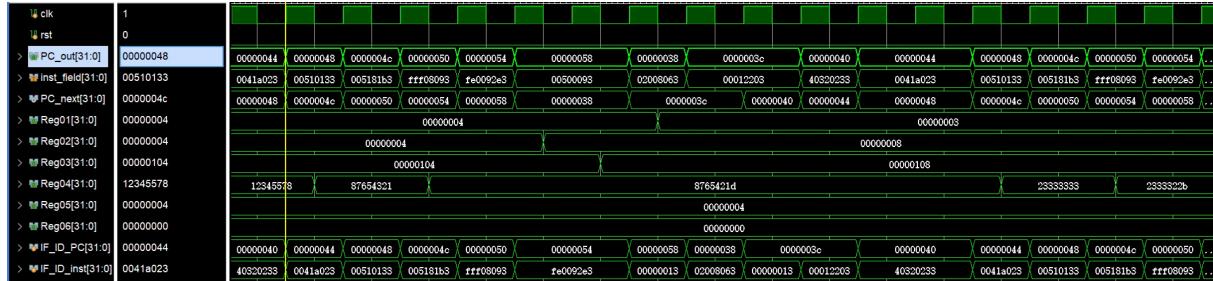


图 10: 仿真分析 5

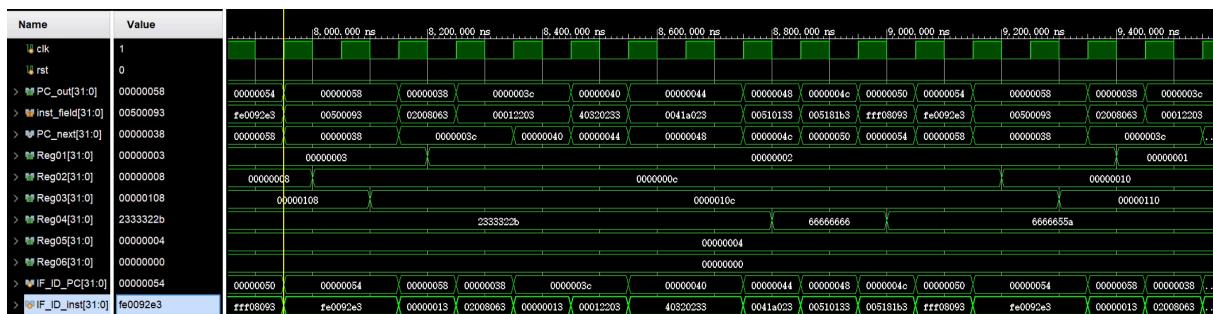


图 11: 仿真分析 6

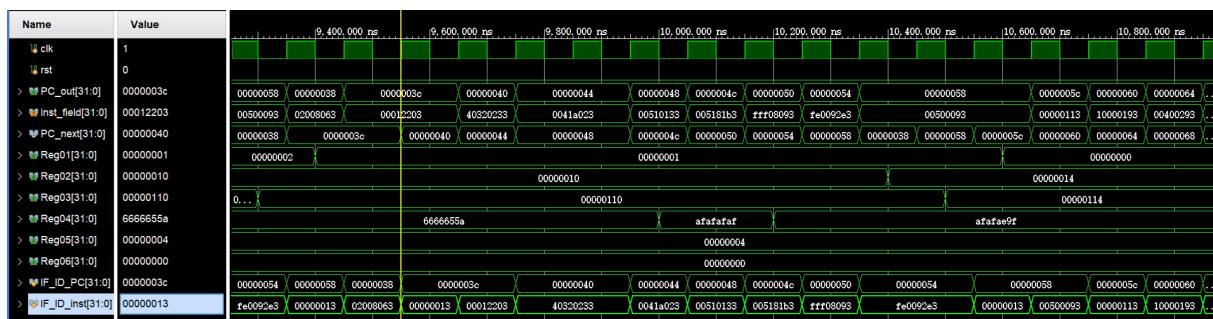


图 12: 仿真分析 7

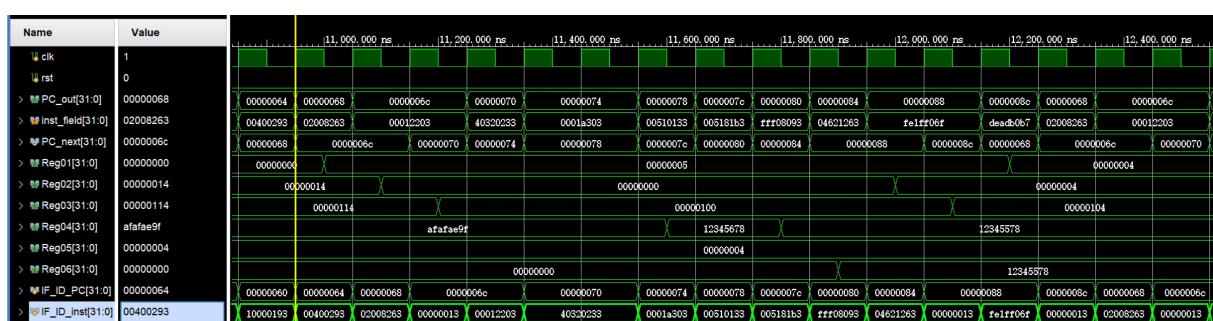


图 13: 仿真分析 8

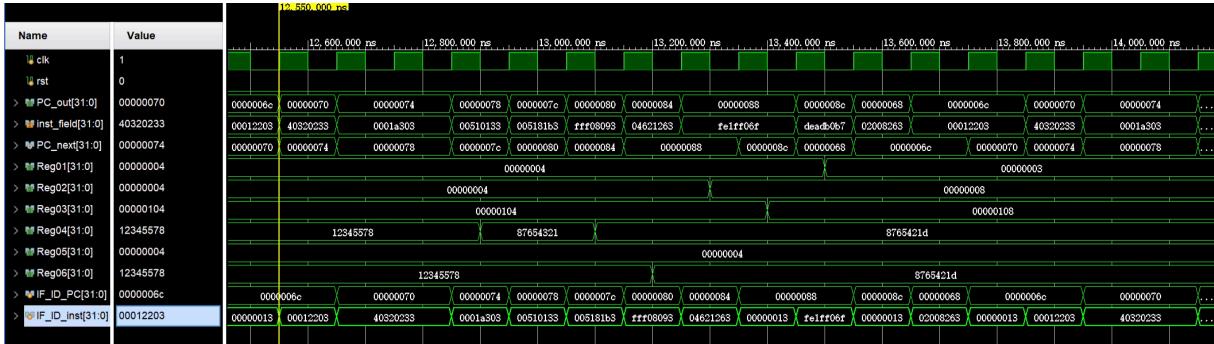


图 14: 仿真分析 9

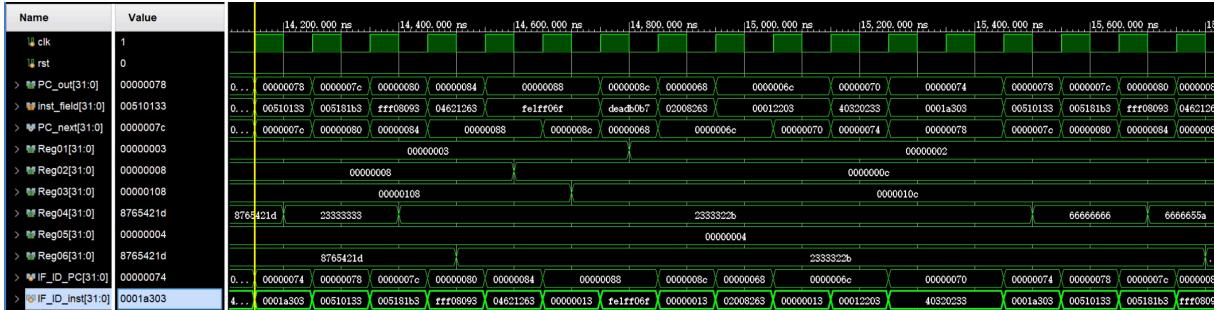


图 15: 仿真分析 10

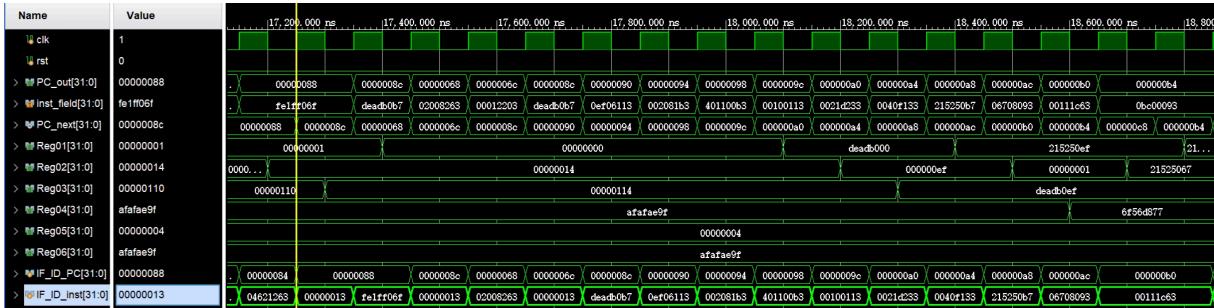


图 16: 仿真分析 11

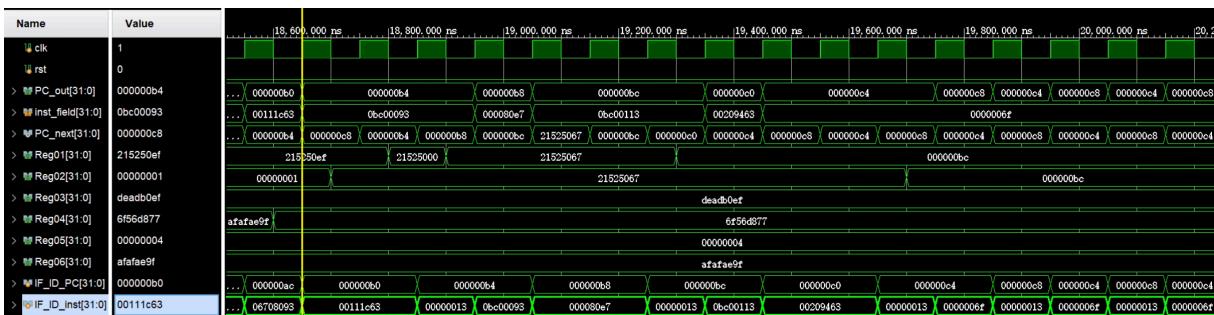


图 17: 仿真分析 12

上面的几张仿真包括了部分循环，PC 的变化已经在前面写出，我们看到在最后的 PC 值是在 0xc4，是 pass 标签的位置，因此我们实现的 PCPU 通过冲突仿真测试。

2.2.2 CPU 功能测试

这里使用和单周期 CPU 同样的验收代码进行测试，我们可以看到最后 reg31 的值是 666，说明验收通过。

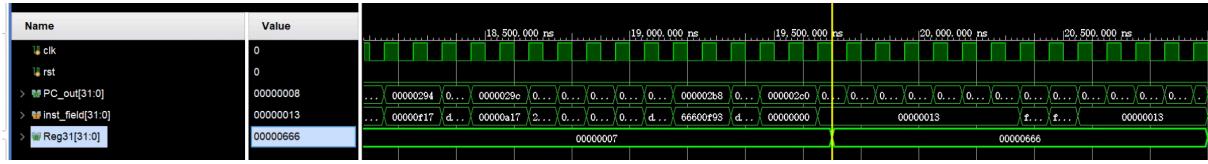


图 18：功能仿真

这里我们与 5-1 中不解决冲突的比较，可以发现 5-1 的指令流中插入了大量的 nop，到最后完成全部测试时，已经过了 58000ns，我们解决了冲突，花费的时间是 19700ns，这比单周期 CPU 通过测试花费的时间略长，原因是为了数据的正确性，采用了 stall 的方式。在实际实现时，时钟频率一定是流水线的可以比单周期的要快；这是因为单周期的最长路径经历了流水线的五个阶段，而流水线将五个阶段拆分，使得各部分在周期内独立更新，而不用等到全部执行完毕再做更新，这样的时钟延迟会更低，因此即使我们添加了一些 stall 使得整体的用时增加，但是在实际实现中，pcpu 是效率更高的。

2.3 下板验证

```
RV32I Single Cycle CPU

pc: 00000024    inst: FE5FF06F

x0: 00000000    ra: FFFFFFFF    sp: 00000000    gp: 40000000    tp: 40000000
t0: F8000000    t1: C0000000    t2: 80000000    s0: 00000001    s1: 00000001
a0: 00000000    a1: C0000000    a2: 00000001    a3: 00000000    a4: 00000000
a5: 00000000    a6: 00000000    a7: 00000000    s2: 00000020    s3: 00000000
s4: 000002A4    s5: 000002A4    s6: 00000000    s7: 00000000    s8: 00000000
s9: 00000000    s10: 00000000    s11: 000000D0    t3: 000000D0    t4: 00D0CBA0
t5: 000002B0    t6: 00000666

rs1: 00 rs1_val: 00000000
rs2: 00 rs2_val: 00000000
rd: 01 reg_i_data: FFFFFFFA0    reg_wen: 0

is_imm: 0        is_auipc: 0      is_lui: 0        imm: 00000000
a_val: 00000000 b_val: 00000000 alu_ctrl: 1      cmp_ctrl: 0
alu_res: 00000000          cmp_res: 0

is_branch: 0     is_jal: 0       is_jalr: 0
do_branch: 0     pc_branch: 00000000

mem_wen: 0        mem_ren: 0
dmem_o_data: FFEDCBA0    dmem_i_data: 00000000    dmem_addr: 00000000

csr_wen: 0        csr_ind: 000    csr_ctrl: 0        csr_r_data: 00000000
mstatus: 00000000    mcause: 00000000    mepc: 00000000    mtval: 00000000
mtvec: 00000000    mie: 00000000    mip: 00000000
```

图 19：下班验证

接入串口看到 reg31 的值是 666，说明通过测试。

3 思考题

3.1 思考题 1

3.1.1 TP-0

这段指令在我的 pcpu 中运行不存在冲突，没有出现使用未写回的数据的情况，因此不会发生 data hazard。总共 12 条指令，需要 15.5 个周期，不存在冲突的情况，采用下降沿写回的方式，所以最后 WB 阶段只用半个周期，考虑到完整的周期，所以总共需要 16 个周期。CPI 为 $16/12=1.3333$ 。

3.1.2 TP-1

```
1 addi    x1, x0, 1
2 addi    x2, x1, 2
3 addi    x3, x1, 3
4 addi    x4, x3, 4
```

第二条与第一条指令冲突，第三条与第一条指令冲突，第四条与第三条冲突，都是发生了数据冲突，在我的流水线中使用 forwarding 解决，因此不会发生 bubble，CPI 为 $8/4=2$ 。

3.2 思考题 2

3.2.1 支持 forwarding

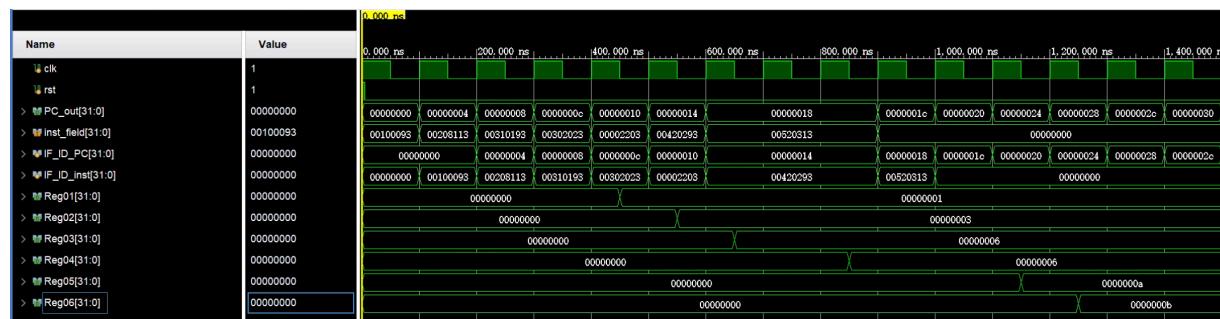


图 20：思考题

完成所有的指令使用了 13 个周期，这里 stall 了两个周期等待 load 指令执行完毕。

4 实验总结

PCPU 是计组中最抓马的一个实验了，相比较开始写 SCPU 时候的茫然，写 PCPU 时已经对整体架构比较熟悉了，datapath 也是在之前的基础上进行了较大的修改，进行了阶段的分割和阶段寄存器的添加。调试的过程很痛苦，vivado 的仿真也很麻烦，好在最后全部都完成了，也顺利地通过了验收代码的测试。相较于 5-1 的实现，我在这里整合了所有的时序逻辑，方便修改和调整。也重构了 PC_next 的计算，使得计算的过程更加清晰。但是我的实现也是存在一些问题的，比如忽略了资源的开销和延迟，尽管能够成功地运行，但仍旧有很多需要改进的地方。

不过，计组课的实验就到这里了，提前开个香槟吧，感觉后面看时间安排决定要不要写 cache 了，感谢调到多次崩溃的 vivado，感谢助教，感谢老师，感谢熬了好几个大夜最终完成的自己。（补写思考题的时候 vivado 又爆了。。。）



图 21：这里有点空，总得放点什么（逃