# 实验1 - 流水线 RISC-V CPU 设计

## 1 实验步骤

### 1.1 实现 RV32I 中的所有指令

这一部分,我们主要是实现了 RV32I 中的所有指令。基于已经给出的代码框架,我们只需要在对应 [to]fill sth in 的地方进行代码填空即可。包括 [ctrlunit.v] 和 [cmp\_32.v]。

#### 1.1.1 Control Unit

CtrlUnit.v 中:

```
1
        wire BEQ = Bop & funct3_0;
 2
        wire BNE = Bop & funct3 1;
 3
        wire BLT = Bop & funct3_4;
 4
        wire BGE = Bop & funct3 5;
 5
        wire BLTU = Bop & funct3 6;
 6
        wire BGEU = Bop & funct3 7;
 7
8
        wire LB = Lop & funct3 0;
9
        wire LH = Lop & funct3_1;
10
        wire LW = Lop & funct3 2;
11
        wire LBU = Lop & funct3 4;
12
        wire LHU = Lop & funct3_5;
13
14
        wire SB = Sop & funct3_0;
15
        wire SH = Sop & funct3 1;
16
        wire SW = Sop & funct3_2;
17
18
        wire LUI = opcode == 7'b0110111;
19
        wire AUIPC = opcode == 7'b0010111;
20
21
        wire JAL = opcode == 7'b1101111;
22
        assign JALR = (opcode == 7'b1100111) && funct3 0;
23
24
        assign Branch = JAL | JALR | B_valid & cmp_res;
25
26
        localparam cmp_EQ = 3'b001;
27
        localparam cmp_NE = 3'b010;
28
        localparam cmp_LT = 3'b011;
29
        localparam cmp_LTU = 3'b100;
30
        localparam cmp_GE = 3'b101;
31
        localparam cmp GEU = 3'b110;
32
        assign cmp_ctrl = BEQ ? cmp_EQ :
33
                          BNE ? cmp_NE :
34
                          BLT ? cmp_LT :
35
                          BLTU? cmp_LTU:
36
                          BGE ? cmp_GE :
37
                          BGEU? cmp_GEU : 3'b000;
```

```
38
39
        assign ALUSrc_A = JAL | JALR | AUIPC;
40
        assign ALUSrc B = I valid | L valid | S valid | LUI | AUIPC;
41
42
        assign rs1use = R_valid | I_valid | B_valid | JALR | L_valid | S_valid
43
        assign rs2use = R valid | B valid | S valid;
44
        localparam hazard optype ALU = 2'd1;
45
        localparam hazard optype LOAD = 2'd2;
46
        localparam hazard_optype_STORE = 2'd3;
        assign hazard_optype = (R_valid | I_valid | JAL | JALR | LUI | AUIPC) ?
    hazard optype ALU :
48
                               (L_valid) ? hazard_optype_LOAD :
49
                                (S_valid) ? hazard_optype_STORE : 2'd0;
```

这里我们主要补充了 Branch、Load、Store 类指令以及 LUI AUIPC JAL JALR 的译码信号。 cmp\_ctrl 信号会传给比较器,我们这里要做的是告诉比较器分支指令需要进行什么样的比较运算。 ALUSrc\_A 如果为 1,那么我们在 ALU 的 A 端口会使用 PC 作为输入而不是 ID 阶段传过去的操作数; ALUSrc\_B 如果为 1,那么我们在 ALU 的 B 端口会使用立即数作为输入而不是 ID 阶段的值。 hazard\_optype, rs1\_use 和 rs2\_use 信号会传给 Forwarding Unit,以便我们判断 hazard。

### 1.1.2 Compare Unit

cmp\_32.v 中:

```
1 | assign c = (EQ & res_EQ) | (NE & res_NE) | (LT & res_LT) | (LTU & res_LTU) | (GE & res_GE) | (GEU & res_GEU);
```

比较器中只有一条语句需要填充,我们根据传入的 [cmp\_ctrl] 信号,决定我们需要进行什么样的比较运算,并将对应的计算结果输出即可。

### 1.2 实现流水线中的 forwarding

#### 1.2.1 HazardDetectionUnit.v

在 HazardDetectionUnit.v 中,输入如下代码:

```
1
        reg[1:0] hazard_optype_EXE, hazard_optype_MEM;
 2
        always@(posedge clk) begin
 3
            hazard_optype_MEM <= hazard_optype_EXE;</pre>
4
            hazard_optype_EXE <= hazard_optype_ID & {2{~reg_DE_flush}};</pre>
 5
        end
6
7
        localparam hazard optype ALU = 2'd1;
8
        localparam hazard_optype_LOAD = 2'd2;
9
        localparam hazard_optype_STORE = 2'd3;
10
11
        wire load_stall = ((rs1use_ID && rs1_ID == rd_EXE) || (rs2use_ID &&
    rs2_ID == rd_EXE && hazard_optype_ID != hazard_optype_STORE)) && rd_EXE &&
    hazard_optype_EXE == hazard_optype_LOAD;
12
        reg [1:0] forwardA, forwardB;
```

```
13
14
        always @(*) begin
15
            if(rs1use ID && rs1 ID == rd EXE && rd EXE && hazard optype EXE ==
    hazard_optype_ALU) begin
16
                forwardA = 2'd1;
17
            end else if(rs1use_ID && rs1_ID == rd_MEM && rd_MEM &&
    hazard_optype_MEM == hazard_optype_ALU) begin
18
                forwardA = 2'd2;
19
            end else if(rs1use ID && rs1 ID == rd MEM && rd MEM &&
    hazard_optype_MEM == hazard_optype_LOAD) begin
20
                forwardA = 2'd3;
21
            end else begin
22
                forwardA = 2'd0;
23
            end
24
25
            if(rs2use ID && rs2 ID == rd EXE && rd EXE && hazard optype EXE ==
    hazard_optype_ALU) begin
26
                forwardB = 2'd1;
27
            end else if(rs2use ID && rs2 ID == rd MEM && rd MEM &&
    hazard_optype_MEM == hazard_optype_ALU) begin
28
                forwardB = 2'd2;
29
            end else if(rs2use_ID && rs2_ID == rd_MEM && rd_MEM &&
    hazard_optype_MEM == hazard_optype_LOAD) begin
30
                forwardB = 2'd3;
31
            end else begin
32
                forwardB = 2'd0;
33
            end
34
        end
35
36
        assign reg_FD_EN = 1'b1;
37
        assign reg_DE_EN = 1'b1;
38
        assign reg_EM_EN = 1'b1;
39
        assign reg_MW_EN = 1'b1;
40
        assign reg_EM_flush = 1'b0;
41
        assign PC_EN_IF = ~load_stall;
42
        assign reg_FD_stall = load_stall;
43
        assign reg_FD_flush = Branch_ID;
44
        assign reg_DE_flush = load_stall;
45
46
        assign forward_ctrl_A = forwardA;
47
        assign forward ctrl B = forwardB;
48
49
        assign forward_ctrl_ls = rs2_EXE == rd_MEM && hazard_optype_EXE ==
    hazard_optype_STORE
50
                                 && hazard_optype_MEM == hazard_optype_LOAD;
```

这里我们首先添加了两个流水线寄存器,用来存放当前时钟周期的 EXE 和 MEM 阶段的 hazard\_optype 信号。

随后我们判断 [load\_stall] 信号,如果为 1,那么我们需要在 ID 阶段进行暂停,等待 MEM 阶段的指令执行完毕。判断的方法是,如果当前指令需要使用前一条指令的结果,当前指令不为 STORE 而且前一条指令为 LOAD 指令,如果是,那么我们需要暂停 ID 阶段的指令,等待 LOAD 指令的结果。

随后我们判断 [forwardA] 和 [forwardB] 信号。如果当前指令需要使用前一条指令的结果,那么 forward 信号为 1,如果需要用前两条指令的结果,那么我们根据冒险的类型,如果是要将 ALU 的结果 前递 [forward] 信号为 2,如果是要将访存的结果前递那么 [forward] 为 3。否则不会发生冒险, forward 信号为 0。

我们还有 forward\_ctrl\_ls 信号, 适用于 lw r1, xxx sw r1, xxx 的情况。

最后我们还要根据 [load\_stall] 和 [Branch\_ID] 信号,决定是否需要暂停 IF 阶段的指令,以及是否需要清空 ID 阶段的指令。

#### 1.2.2 流水线集成

最后我们在 RV32core.v 中,完成代码填空,将前递的数据和相关控制信号填入 MUX 即可。

```
1
        // IF
 2
        MUX2T1 32
    mux_IF(.I0(PC_4_IF),.I1(jump_PC_ID),.s(Branch_ctrl),.o(next_PC_IF));
 3
 4
        // ID
 5
        MUX4T1 32
   mux_forward_A(.I0(rs1_data_reg),.I1(ALUout_EXE),.I2(ALUout_MEM),.I3(Datain_
   MEM),
 6
            .s(forward_ctrl_A),.o(rs1_data_ID));
 7
 8
        MUX4T1 32
    mux_forward_B(.I0(rs2_data_reg),.I1(ALUout_EXE),.I2(ALUout_MEM),.I3(Datain_
    MEM),
9
            .s(forward_ctrl_B),.o(rs2_data_ID));
10
11
        // EX
12
        MUX2T1 32
    mux_A_EXE(.IO(rs1_data_EXE),.I1(PC_EXE),.s(ALUSrc_A_EXE),.o(ALUA_EXE));
13
14
        MUX2T1 32
    mux_B_EXE(.I0(rs2_data_EXE),.I1(Imm_EXE),.s(ALUSrc_B_EXE),.o(ALUB_EXE));
15
        . . .
```

### 2 实验评估

### 2.1 仿真

这里我们使用实验框架给出的代码进行仿真。

```
1 addi x0, x0, 0 # PC=0
2 lw x2, 4(x0)
3 lw x4, 8(x0)
4 add x1, x2, x4
5 addi x1, x1, -1
6 lw x5, 12(x0)
7 lw x6, 16(x0)
8 lw x7, 20(x0)
9 sub x1, x4, x2
10 and x1,x4,x2
11 or x1,x4,x2 # PC=28
```

ore_sim_behav.wcfg																									_ 0 7
a <u> </u> @   @	56 HF	и ы	-0	20 4	r I fa I	ar la	4																		
1 =   1   1		11 /1	_		1.	41   1																			
Name	Value	0.000 ns		2.000 ns		4. 000 ns		6.000 ns		8.000 ns		10.000 n	s	12.000 n		14.000 n		16.000 n		18.000 n		20.000 n		22. 000 r	ıs
¼ clk	0																								
₩PC_IF[31:0]	0000003c	0	0000000		0000	0004	000	80000	000	0000c		0000	0010		0000	0014	0000	0018	0000	0001c	0000	10020	0000	0024	0000002
₩inst_IF[31:0]	002350b3	00000013		00402103		00802203		004100ь3		fff08093		8093	00c0228		2283	01002303		01402383		4022	:00РЗ	0022	70ь3	002260t	
₩PC_ID[31:0]	00000038	00000000					00000004		80000000		0000000c			00000010		00000014		00000018		0000	1001c	0000	0020	0000002	
₩ inst_ID[31:0]	004120b3	00000000			00000013		00402103		00802203		004100b3			fff08093		00c02283		01002303		0140	2383	40220	00ь3	002270t	
♥PC_EXE[31:0]	00000034		000000				000			00000004 0		00000008 00		0000	0000c		00000010		00000014		00000018		00000	001c	0000002
⊌inst_EXE[31:0]	002220b3		00000000			00000013		00402103		0080	00802203		000000 004100ь3		00Ь3	fff08093		00c02283		01002303		0140	2383	402200t	
₱ PC_MEM[31:0]	00000030		00000000						0000004			0000	0000 80000000		0000c		00000010		00000014		0000	018	000000		
<b>₩</b> inst_MEM[31:0]	002210b3		00000000				000	00000013 00402103		0080	2203	00000000		004100ь3		fff08093		00c02283		0100	2303	0140238			
♥ PC_WB[31:0]	0000002c		00000000								0000	0004	0000	8000	0000		0000c		00000010		00000014		000000		
inst_WB[31:0]	002240b3					00000000					0000	0013	0040	2103	0080	2203	0000	0000	004	100ь3	fff(	8093	00c0	2283	0100230
18 Branch_ctrl	0																								
18 JALR	0																								
18 RegWrite_ctrl	1																								
16 mem_w_ctrl	0																								
18 MIO_ctrl	0																								
1 ALUSrc_A_ctrl	0																								
18 ALUSrc_B_ctrl	0																								
1 DatatoReg_ctrl	0																								
16 rs1use_ctrl	1																								
16 rs2use_ctrl	1																								
♦ hazard_optctrl[1:	1		0			i			2											2				1	
ImmSel_ctrl[2:0]	0		0					1					ò						1					0	
cmp_ctrl[2:0]	0													0											
₩ ALUControl_ctrl[3:0	8		0																				2		3
⊌ forward_ctrl_A[1:0]	0					0						3	(	)		ı					0				
forward_ctrl_B[1:0]	0						0						:	}						0					
18 forward_ctrl_ls	0																								

可以看到,在 9ns 时,当前  $1w \times 4$ ,8(x0) 在 EX 阶段(会写入 x4),而  $add \times 1$ ,x2,x4 在 ID 阶段(要读取  $x4 \times 2$ ),产生  $1v \times 1$ 0 小  $1v \times 2$ 0,在 EX 阶段(会写入  $1v \times 1$ 0,在 EX N 的),在 EX N 的  $1v \times 1$ 0,在 EX N 的),在 EX N 的  $1v \times 1$ 0,在 EX N 的),在 EX N 的),在 EX N 的  $1v \times 1$ 0,在 EX N

再过一个周期后(11ns 时),因为 stall 我们 ID 阶段的指令并没有发生改变,而此时处于 MEM 阶段的指令变成了 lw x4,8(x0)(要写入 x4),同样发生了第三类数据冒险,因此可以看到此时forward\_ctrl\_B=3。

在 13ns, 当前 add x1, x2, x4 处于 EX 阶段 (会写入 x1), addi x1, x1, -1 处于 ID 阶段 (要读取 x1), 发生第一类数据冒险,可以看到此时 forward\_ctrl\_A=1,通过前递解决了这次冲突。

```
1 and x1,x4,x2 # PC=24
2 or x1,x4,x2
3 xor x1,x4,x2
4 sll x1,x4,x2
5 slt x1,x4,x2
6 slt x1,x2,x4
7 srl x1, x6, x2
8 sra x1, x6, x2
9 sra x1, x7, x2
```

```
10 sltu x1, x6, x7

11 sltu x1, x7, x6

12 add x0,x0,x0  # PC=50

13 addi x1,x10,-3

14 andi x1,x4,15

15 ori x1,x4,15

16 xori x1,x4,15

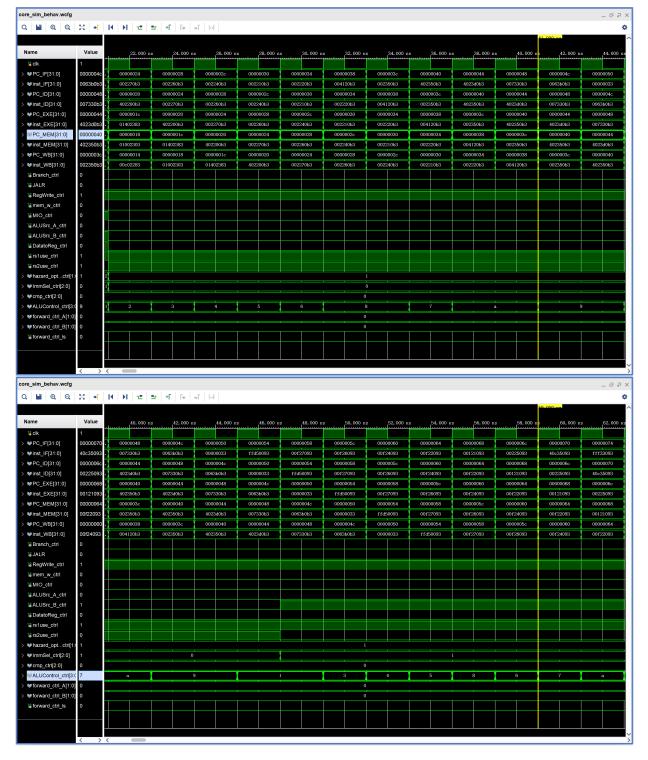
17 slti x1,x4,15

18 slli x1,x4,1

19 srli x1,x4,2

20 srai x1, x6, 12

21 sltiu x1, x6, -1  # PC=74
```



根据代码可知这里没有发生冒险,因此不会有前递和 stall,符合波形结果。

```
1
   srai x1, x6, 12
                       # PC=70
 2
   sltiu x1, x6, -1
 3
    sltiu x1, x7, -1
4
    beq x4,x5,label0
 5
    beq x4,x4,label0
 6
    addi x0,x0,0
 7
    addi x0,x0,0
                      # PC=84
8
   label0:
9
    bne x4,x4,label1
10
    bne x4,x5,label1 # PC=8C
```

```
11 addi x0,x0,0

12 addi x0,x0,0 # PC=94

13 blt x5,x4,label2

14 blt x4,x5,label2 # PC=9C

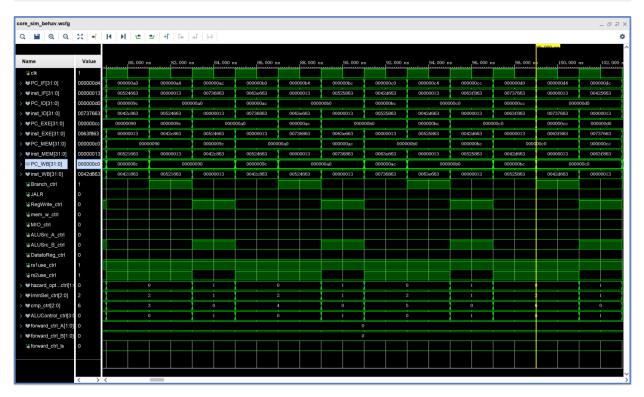
15 addi x0,x0,0 # PC=A4
```

ime	Value	60.000	ns 62.000 i	ns 64.000 n	s 66.000 n:	s _68.000 n	s 70.000 s	ns ,72,000 n	ıs ,74.000 n	s 76.000 n	s ,78.000 n	zo 000 ee ; 80.000 n:	s ,82.000
clk	1 4												
PC_IF[31:0]	000000a0 .	00000070	00000074	00000078	0000007c	0800000	00000084	0000008c	00000090	00000094	0000009с	000000a0	000000a4
inst_IF[31:0]	00524663	40c35093	fff33093	fff3b093	00520863	00420663	00000013	00421863	00521663	00000013	0042c863	00524663	00000013
PC_ID[31:0]	0000009c	0000006с	00000070	00000074 00000078		0000007c	000	00080	0000008c 000		00090	0000009с	000000a0
inst_ID[31:0]	0042c863 .	00225093	40c35093	fff33093 fff3b093		00520863	00420663	00000013	00421863 00521663		00000013	0042c863	00524663
PC_EXE[31:0]	00000090	00000068	0000006c	00000070	00000074	00000078	0000007c	0000	0000008c		0000	0090	0000009с
inst_EXE[31:0]	00000013	00121093	00225093	40c35093	fff33093	fff3b093	00520863	00420663	00000013 00421863		00521663	00000013	0042c863
PC_MEM[31:0]	00000090	00000064	00000068	0000006с	0000070	0000074	00000078	0000007с	00000080		0000008c	0000	0090
inst_MEM[31:0]	00521663	00f22093	00121093	00225093	40c35093	fff33093	fff3b093	00520863	00420663	00000013	00421863	00521663	00000013
PC_WB[31:0]	0000008c .	00000060	0000064	00000068	0000006c	00000070	00000074	0000078	0000007c	0000	0800	0000008c	00000090
inst_WB[31:0]	00421863 .	00f24093	00f22093	00121093	00225093	40c35093	fff33093	fff3b093	00520863	00420663	00000013	00421863	00521663
Branch_ctrl	0												
JALR	0												
RegWrite_ctrl	0												
mem_w_ctrl	0												
MIO_ctrl	0												
ALUSrc_A_ctrl	0												
ALUSrc_B_ctrl	0												
DatatoReg_ctrl	0												
rs1use_ctrl	1												
rs2use_ctrl	1												
hazard_optctrl[1:	_			1				1	<del></del>	0	1		
ImmSel_ctrl[2:0]	2					-	2	1	<b>≬</b> ————	2	1	. 2	
cmp_ctrl[2:0]	3							0	1	2	0		
ALUControl_ctrl[3:0	<b>=</b> 4	7	а		•	(		1		0	1		
forward_ctrl_A[1:0]	_												
forward_ctrl_B[1:0]								0					
forward_ctrl_ls	0												

在 69ns 时,beq x4, x4, label0 处于 ID 阶段,此时我们在 ID 阶段已经通过比较器提前判断出此条指令的跳转是要发生的,因此我们可以看到 Branch\_ctrl=1,同时插入了一条 bubble (71~73 ns 的 inst\_ID 为 nop, 这里只刷新了指令内容,没有刷新 PC),此外 IF 阶段取的指令也被丢弃,而是在下一个周期从跳转的目标地址取值(这里的目标地址是 0x8c)。后续 75ns, 81ns 原因同上,这里不再重复。

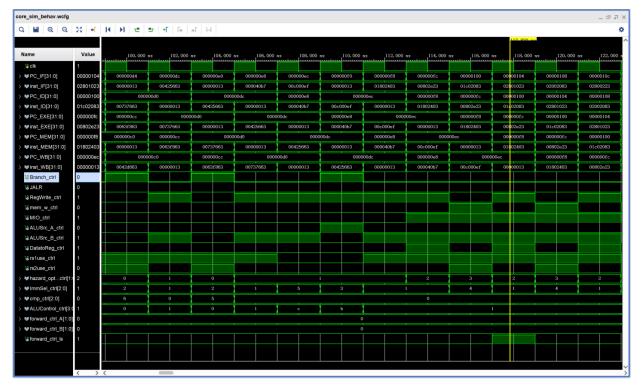
```
1 addi x0,x0,0
                  # PC=A0
2 addi x0,x0,0
 3 label2:
4 bltu x6,x7,label3
5 bltu x7,x6,label3
 6 addi x0,x0,0
7 addi x0,x0,0
8 label3:
9 bge x4,x5,label4
10 bge x5,x4,label4
11 addi x0,x0,0
12 addi x0,x0,0
13 label4:
14 bgeu x7,x6,label5
15 bgeu x6,x7,label5
16 addi x0,x0,0
17 addi x0,x0,0
18 label5:
```

```
19 bge x4,x4,label6
20 addi x0,x0,0 # PC=DC
```



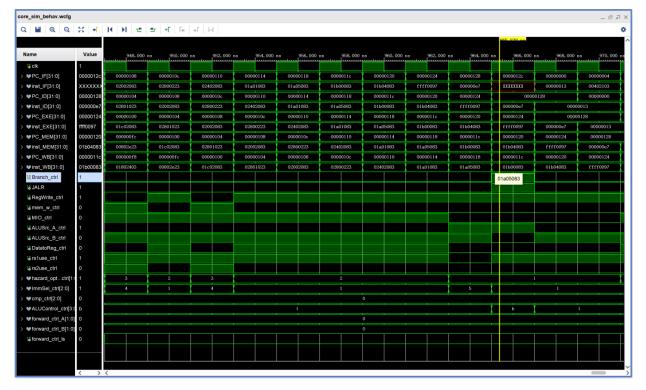
这里在 81ns, 87ns, 93ns 也发生了跳转,原因同上,这里不再重复。

```
1 addi x0,x0,0 # PC=D4
2 addi x0,x0,0
3 label5:
4 bge x4,x4,label6 # PC=DC
              # PC=E0
5 addi x0,x0,0
6
  addi x0,x0,0
7
   label6:
8
  lui x1,4
                  # PC=E8
9
   jal x1,12
  addi x0,x0,0 # PC=F0
10
11
  addi x0,x0,0
12
  lw x8, 24(x0) # PC=F8
13 sw x8, 28(x0)
14 lw x1, 28(x0)
15 sh x8, 32(x0)
16 lw x1, 32(x0)
17 sb x8 36(x0) # PC=10C
```



这里在 99ns, 103ns, 109ns 也发生了跳转,原因同上,这里不再重复。 在 117ns,此时 sw x8, 28(x0) 处于 EX 阶段 (要存入 x8 的值), lw x8, 24(x0) 处于 MEM 阶段 (要读取 x8 的值), 因此我们需要将内存中读出来的值前递,可以看到此时 forward\_ctrl\_ls=1。

```
1 1w
     x1, 32(x0)
                  # PC=108
2
  sb x8, 36(x0)
3
  lw x1, 36(x0)
4
  lh x1, 26(x0)
5
  lhu x1, 26(x0)
6
  1b
     x1, 27(x0)
7
  lbu x1, 27(x0)
8 auipc x1, 0xffff0
9 jalr x1,0(x0) # PC=128
```



这里在 965ns 时,jalr x1,0(x0) 处于 ID 阶段,可以看到 JALR=1,随后这条指令在 EX 阶段被执行,计算出了要跳转的地址 0x0,因此可以看到 967ns 时 PC 回到了 0x0。

### 2.2 上板结果

PC=4 对应的指令是 Tw x2, 4(x0), 这里要把地址 0x4 存放的值 0x8 写入寄存器 x2。而 PC=10 对应的指令是 add x1, x2, x4。这里发生了第三类数据冒险,我们需要把访存取出的值前递到 ID 阶段。从后面的图我们可以看到前递成功而且没有多余的 stall。

PC=10 对应的指令是 add x1, x2, x4, PC=14 对应的指令是 addi x1, x1, -1。这里发生了第一类数据冒险,可以看到第一条执行后 x1 的值为 18 (x4 的值为 0x10, 这也说明我们 x2 的数据冒险也成功解决),而第二条执行后 x1 的值为 17,说明我们解决了这个冒险。

```
x01=0x00000017
x01=0x00000018
x02=0x00000008
                             x02=0x00000008
x03=0x00000000
                             x03=0x00000000
                             x04=0x000000010
x04=0x00000010
x05=0x000000000
                             x05=0x00000000
                             x06=0x00000000
x06=0x00000000
x07=0x00000000
                             x07=0x00000000
x08=0x00000000
                             x08=0x000000000
x09=0x00000000
                             x09=0x00000000
x10=0x00000000
                             x10=0x00000000
x11=0x00000000
                            x11=0x00000000
x12=0x000000000
                            x12=0x000000000
x13=0x000000000
                            x13=0x00000000
x14=0x000000000
                            x14=0x000000000
x15=0x00000000
                            x15=0x00000000
x16=0x00000000
                            x16=0x000000000
x17=0x00000000
                            x17=0x000000000
x18=0x00000000
                            x18=0x000000000
x19=0x00000000
                            x19=0x00000000
x20=0x000000000
                            x20=0x00000000
x21=0x000000000
                            x21=0x00000000
x22=0x000000000
                            x22=0x000000000
x23=0x000000000
                            x23=0x000000000
x24=0x000000000
                            x24=0x000000000
x25=0x00000000
                            x25=0x000000000
x26=0x000000000
                            x26=0x000000000
x27=0x000000000
                            x27=0x000000000
x28=0x00000000
                            x28=0x00000000
x29=0x000000000
                            x29=0x00000000
x30=0x00000000
                            x30=0x00000000
x31=0x00000000
                            x31=0x00000000
WB PC =0x00000010
WB_INST=0xFFF08093
MEMADDR=0x00000000C
                            WB_PC =0x00000014
WB_INST=0x00C02283
MEMDATA=0x00000014
                            MEMADDR=0x00000010
                            MEMDATA=0xFFFF0000
```

PC=80 对应的指令是 beq x4 x4 12 , 这里因为发生了跳转,因此我们需要插入一条 bubble , 我们可以看到此时的 inst 为 NOP , 说明我们的确插入了一条 bubble (这里我们没有改变 PC) 。同时可以看到这条指令的下一条指令是 PC=8C , 说明我们的确执行了跳转。

```
x01=0x00000001
                         x02=0x00000008
x02=0x00000008
                        x03=0x00000000
x03=0x000000000
                        x04=0x00000010
x04=0x00000010
                        x05=0x00000014
x05=0x00000014
                        x06=0xFFFF0000
x06=0xFFFF0000
                        x07=0x0FFF0000
x08=0xFF000F0F
x07=0x0FFF0000
x08=0xFF000F0F
                        x09=0x00000000
x09=0x00000000
x10=0x00000000
                        x10=0x00000000
x11=0x00000000
                        x11=0x00000000
                        x12=0x000000000
x12=0x000000000
                        x13=0x00000000
x13=0x00000000
                        x14=0x000000000
x14=0x000000000
                        x15=0x000000000
x15=0x00000000
                        x16=0x00000000
x16=0x000000000
x17=0x00000000
                        x17=0x000000000
x18=0x000000000
                        x18=0x00000000
                        x19=0x00000000
x19=0x000000000
                        x20=0x000000000
x20=0x000000000
                        x21=0x00000000
x21=0x00000000
                        x22=0x00000000
x22=0x000000000
                        x23=0x00000000
x23=0x00000000
                        x24=0x00000000
x24=0x000000000
                        x25=0x00000000
x25=0x000000000
                        x26=0x00000000
x26=0x000000000
                        x27=0x00000000
x27=0x00000000
                        x28=0x00000000
x28=0x00000000
                        x29=0x00000000
x29=0x00000000
                        x30=0x00000000
x30=0x00000000
                         x31=0x000000000
x31=0x00000000
                        WB PC =0x0000008C
WB_PC =0x00000080
WB_INST=0x00000013
                        WB_INST=0x00421863
                        MEMADDR=0xFFFFFFF
MEMADDR=0xFFFFFFFF
MEMDATA=0xAA55AA55
                         MEMDATA=0xAA55AA55
```

PC=F8 对应的指令是 lw x8, 24(x0), PC=FC 对应的指令是 sw x8, 28(x0)。这里我们可以看到两条指令之间并没有 bubble,说明我们利用前递解决了 load 后面 store 的情况。

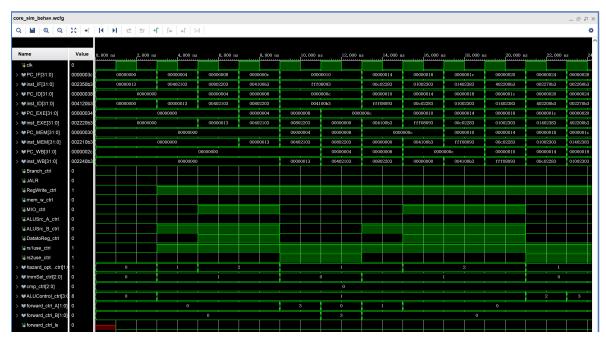
```
x01=0x000000F0
                       x01=0x000000F0
x02=0x00000008
                       x02=0x00000008
x03=0x00000000
                       x03=0x00000000
x04=0x00000010
                       x04=0x00000010
x05=0x00000014
                       x05=0x00000014
x06=0xFFFF0000
                       x06=0xFFFF0000
x07=0x0FFF0000
                       x07=0x0FFF0000
x08=0xFF000F0F
                       x08=0xFF000F0F
x09=0x00000000
                       x09=0x00000000
x10=0x00000000
                       x10=0x00000000
                       x11=0x000000000
x11=0x00000000
x12=0x00000000
                       x12=0x00000000
x13=0x000000000
                      x13=0x00000000
                       x14=0x000000000
x14=0x00000000
x15=0x00000000
                       x15=0x000000000
                       x16=0x000000000
x16=0x00000000
                      x17=0x00000000
x17=0x000000000
                       x18=0x00000000
x18=0x00000000
x19=0x00000000
                       x19=0x000000000
x20=0x00000000
                       x20=0x000000000
                       x21=0x000000000
x21=0x00000000
                      x22=0x000000000
x22=0x000000000
                       x23=0x000000000
x23=0x00000000
                      x24=0x000000000
x24=0x00000000
                       x25=0x000000000
x25=0x00000000
                       x26=0x000000000
x26=0x000000000
                       x27=0x00000000
x27=0x000000000
                       x28=0x00000000
x28=0x00000000
                       x29=0x000000000
x29=0x000000000
                       x30=0x00000000
x30=0x00000000
                       x31=0x000000000
x31=0x00000000
                      WB_PC =0x000000FC
WB INST=0x00802E23
WB_PC = 0x000000F8
WB_INST=0x01802403
                      MEMADDR=0x0000001C
MEMADDR=0x0000001C
                       MEMDATA=0xFF000F0F
MEMDATA=0xFF000F0F
```

PC=128 对应的指令是 jalr x1,0(x0),这里也发生了跳转,因此插入了一条 bubble,同时我们可以看到下一条指令 PC 又变为了 0x0,说明跳转成功,而且我们将返回地址 PC+4=12C 写入了寄存器 x1。

```
x01=0x0000012C
                      x01=0x0000012C
x02=0x00000008
                      x02=0x00000008
x03=0x00000000
                      x03=0x000000000
x04=0x00000010
                      x04=0x00000010
x05=0x00000014
                      x05=0x00000014
x06=0xFFFF0000
                      x06=0xFFFF0000
x07=0x0FFF0000
                      x07=0x0FFF0000
x08=0xFF000F0F
                      x08=0xFF000F0F
x09=0x00000000
                      x09=0x00000000
x10=0x00000000
                      x10=0x000000000
x11=0x00000000
                      x11=0x000000000
x12=0x00000000
                      x12=0x00000000
x13=0x00000000
                      x13=0x00000000
x14=0x000000000
                      x14=0x000000000
x15=0x00000000
                      x15=0x000000000
                      x16=0x000000000
x16=0x00000000
x17=0x00000000
                      x17=0x000000000
x18=0x00000000
                      x18=0x00000000
                      x19=0x00000000
x19=0x00000000
                      x20=0x000000000
x20=0x000000000
x21=0x00000000
                      x21=0x000000000
                      x22=0x000000000
x22=0x000000000
                      x23=0x00000000
x23=0x00000000
x24=0x000000000
                      x24=0x000000000
                      x25=0x00000000
x25=0x000000000
                      x26=0x000000000
x26=0x000000000
x27=0x00000000
                      x27=0x00000000
                      x28=0x00000000
x28=0x00000000
                      x29=0x00000000
x29=0x00000000
                      x30=0x00000000
x30=0x00000000
                      x31=0x000000000
x31=0x00000000
                      WB PC =0 \times 000000000
WB_PC =0 \times 00000128
                      WB_INST=0x00000013
WB_INST=0x00000013
                      MEMADDR=0x00000004
MEMADDR=0xFFFFFFFF
                      MEMDATA=0x00000008
MEMDATA=0xAA55AA55
```

1. 添加 forwarding 机制后, stall 的数量减少。现在只有在 load-use (即 ld reg xxx 后的下一条指令就要使用 reg 的值)的情况下才会需要 stall, 其他时候都可以由前递解决。例如我们的代码:

```
1 add x1, x2, x4 # PC=0xC
2 addi x1, x1, -1 # PC=0X10
```



从图中可以看到,在 13ns 时,forward\_ctrl\_A=1,说明这时候 ID 阶段 A 的值会来自 ALU 阶段的前递,可以看到后续没有 stall 操作,而结合上板结果可知我们的操作正确, x1 被写入了正确的值,因此前递操作正确且有效。

2. 没有办法。因为 load-use hazard 的情况下,我们需要的数据直到 MEM 阶段结束才能用于前递,而这时 EX 阶段的指令也已经执行完毕,前递无法在 EX 阶段执行完成前进行,也就无法起到前递的效果。

如这样的例子:

```
1 | 1w x1, 0(x0)  # need a nop
2 | add x2, x1, x1
```