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：

```
    wire BEQ = Bop & funct3_0;
    wire BNE = Bop & funct3_1;
    wire BLT = Bop & funct3_4;
    wire BGE = Bop & funct3_5;
    wire BLTU = Bop & funct3_6;
    wire BGEU = Bop & funct3_7;

    wire LB =  Lop & funct3_0;
    wire LH =  Lop & funct3_1;
    wire LW =  Lop & funct3_2;
    wire LBU = Lop & funct3_4;
    wire LHU = Lop & funct3_5;

    wire SB = Sop & funct3_0;
    wire SH = Sop & funct3_1;
    wire SW = Sop & funct3_2;

    wire LUI   = opcode == 7'b0110111;
    wire AUIPC = opcode == 7'b0010111;

    wire JAL  =  opcode == 7'b1101111;
    assign JALR = (opcode == 7'b1100111) && funct3_0;
    ...
    assign Branch = JAL | JALR | B_valid & cmp_res;
    ...
    localparam cmp_EQ  = 3'b001;
    localparam cmp_NE  = 3'b010;
    localparam cmp_LT  = 3'b011;
    localparam cmp_LTU = 3'b100;
    localparam cmp_GE  = 3'b101;
    localparam cmp_GEU = 3'b110;
    assign cmp_ctrl = BEQ ? cmp_EQ  :
                      BNE ? cmp_NE  :
                      BLT ? cmp_LT  :
                      BLTU? cmp_LTU :
                      BGE ? cmp_GE  :
                      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;
47      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;
4           hazard_optype_EXE <= hazard_optype_ID & {2{~reg_DE_flush}};
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。

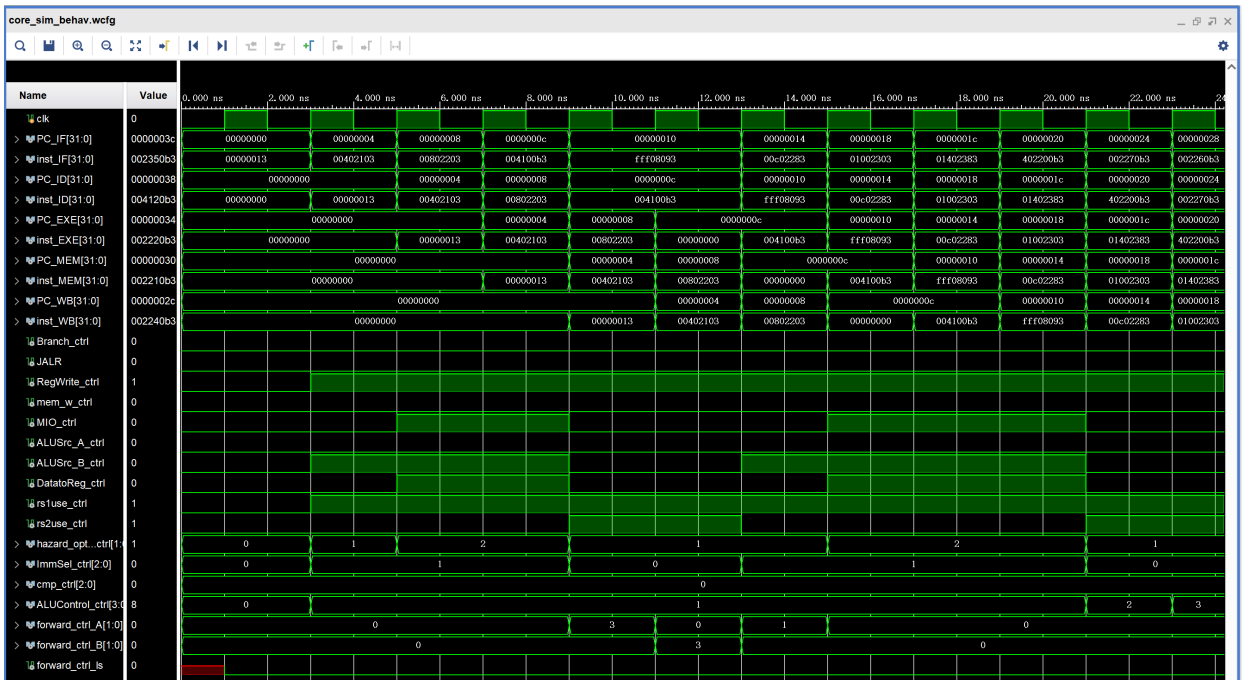
```
    // IF
    MUX2T1_32
mux_IF(.I0(PC_4_IF),.I1(jump_PC_ID),.s(Branch_ctrl),.o(next_PC_IF));
    ...
    // ID
    MUX4T1_32
mux_forward_A(.I0(rs1_data_reg),.I1(ALUout_EXE),.I2(ALUout_MEM),.I3(Datain_
MEM),
        .s(forward_ctrl_A),.o(rs1_data_ID));

    MUX4T1_32
mux_forward_B(.I0(rs2_data_reg),.I1(ALUout_EXE),.I2(ALUout_MEM),.I3(Datain_
MEM),
        .s(forward_ctrl_B),.o(rs2_data_ID));
    ...
    // EX
    MUX2T1_32
mux_A_EXE(.I0(rs1_data_EXE),.I1(PC_EXE),.s(ALUSrc_A_EXE),.o(ALUA_EXE));

    MUX2T1_32
mux_B_EXE(.I0(rs2_data_EXE),.I1(Imm_EXE),.s(ALUSrc_B_EXE),.o(ALUB_EXE));

    ...
```

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。

```
addi x0, x0, 0  # PC=0
lw x2, 4(x0)
lw x4, 8(x0)
add x1, x2, x4
addi x1, x1, -1
lw x5, 12(x0)
lw x6, 16(x0)
lw x7, 20(x0)
sub x1, x4, x2
and x1,x4,x2
or  x1,x4,x2    # PC=28
```



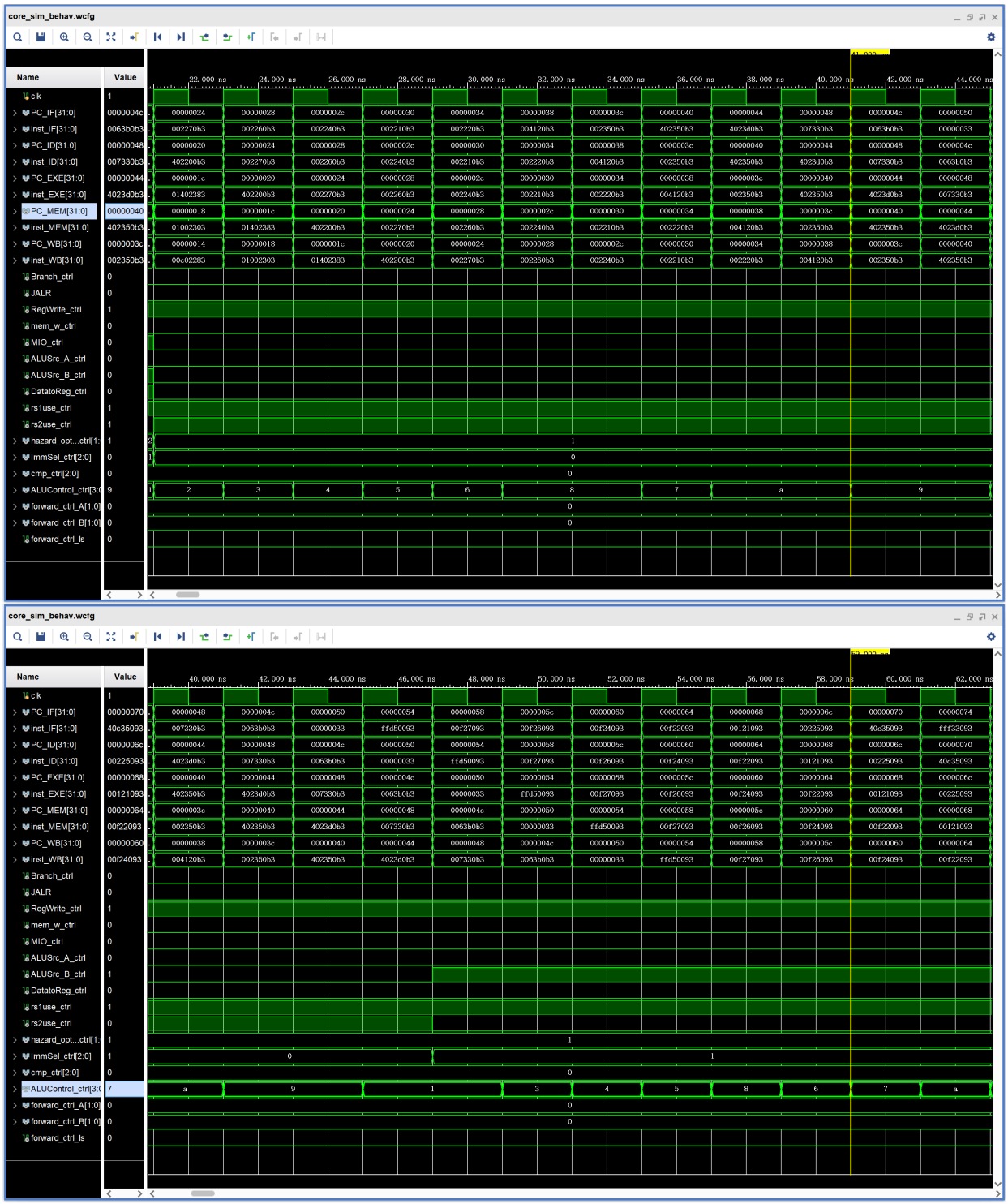
可以看到，在 9ns 时，当前 `lw x4, 8(x0)` 在 EX 阶段（会写入 `x4`），而 `add x1, x2, x4` 在 ID 阶段（要读取 `x4 x2`），产生 load-use stall，因此我们暂停 ID 和 IF 阶段的指令一个周期。而且此时 `lw x2, 4(x0)` 处于 MEM 阶段（会写入 `x2`），因此这里发生了第三类数据冒险，即我们需要把访存的结果前递到 ID 阶段，因此可以看到此时 `forward_ctrl_A=3`。

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。

```
and x1,x4,x2    # PC=24
or  x1,x4,x2
xor x1,x4,x2
sll x1,x4,x2
slt x1,x4,x2
slt x1,x2,x4
srl x1, x6, x2
sra x1, x6, x2
sra x1, x7, x2
```

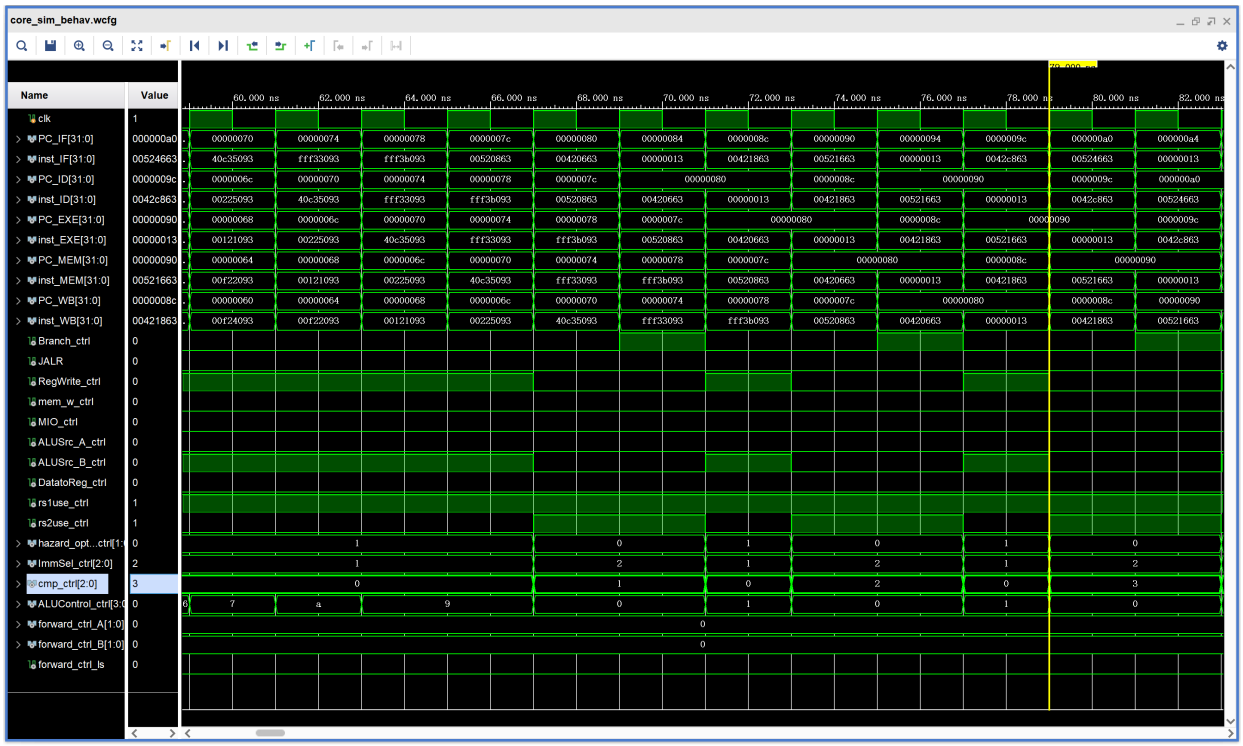
```
sltu x1, x6, x7
sltu x1, x7, x6
add x0,x0,x0    # PC=50
addi x1,x10,-3
andi x1,x4,15
ori  x1,x4,15
xori x1,x4,15
slti x1,x4,15
slli x1,x4,1
srli x1,x4,2
srai x1, x6, 12
sltiu x1, x6, -1    # PC=74
```



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。

```
srai x1, x6, 12      # PC=70
sltiu x1, x6, -1
sltiu x1, x7, -1
beq  x4,x5,label0
beq  x4,x4,label0
addi x0,x0,0
addi x0,x0,0         # PC=84
label0:
bne  x4,x4,label1
bne  x4,x5,label1    # PC=8C
```

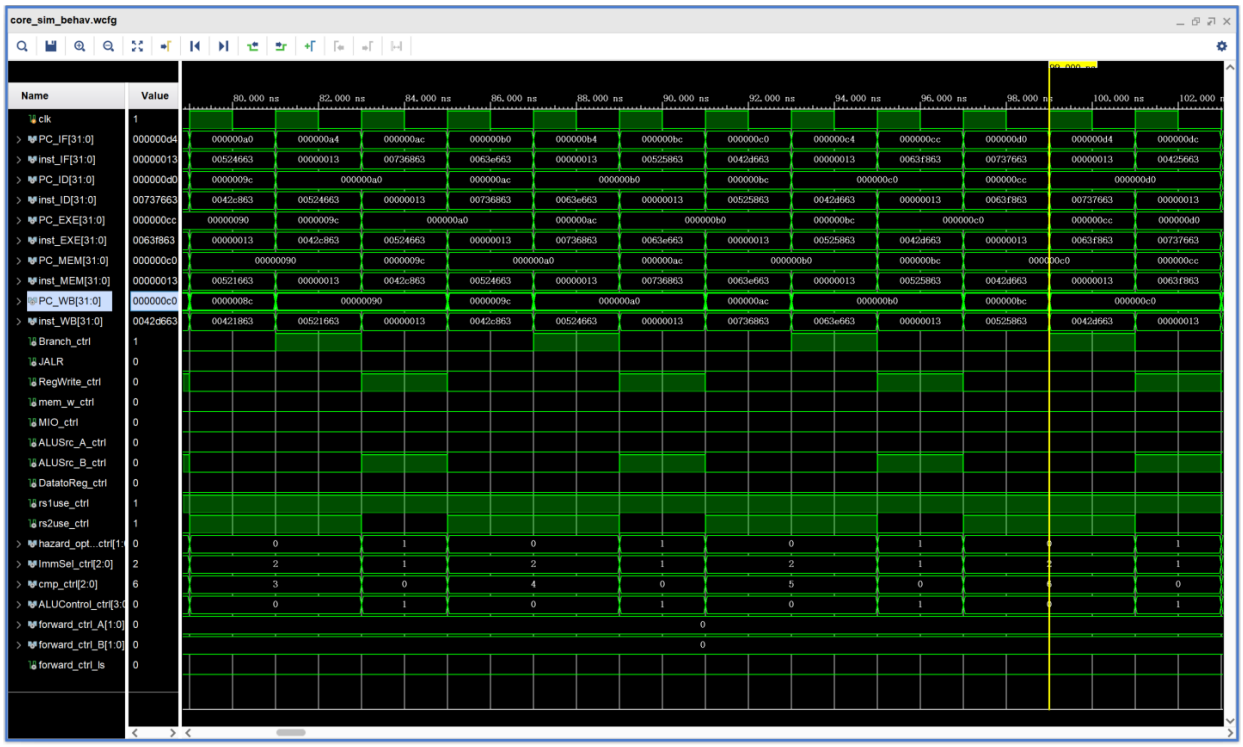
```
addi x0,x0,0
addi x0,x0,0          # PC=94
blt  x5,x4,label2
blt  x4,x5,label2     # PC=9C
addi x0,x0,0
addi x0,x0,0          # PC=A4
```



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。

```
addi x0,x0,0          # PC=A0
addi x0,x0,0
label2:
bltu x6,x7,label3
bltu x7,x6,label3
addi x0,x0,0
addi x0,x0,0
label3:
bge x4,x5,label4
bge x5,x4,label4
addi x0,x0,0
addi x0,x0,0
label4:
bgeu x7,x6,label5
bgeu x6,x7,label5
addi x0,x0,0
addi x0,x0,0
label5:
```

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



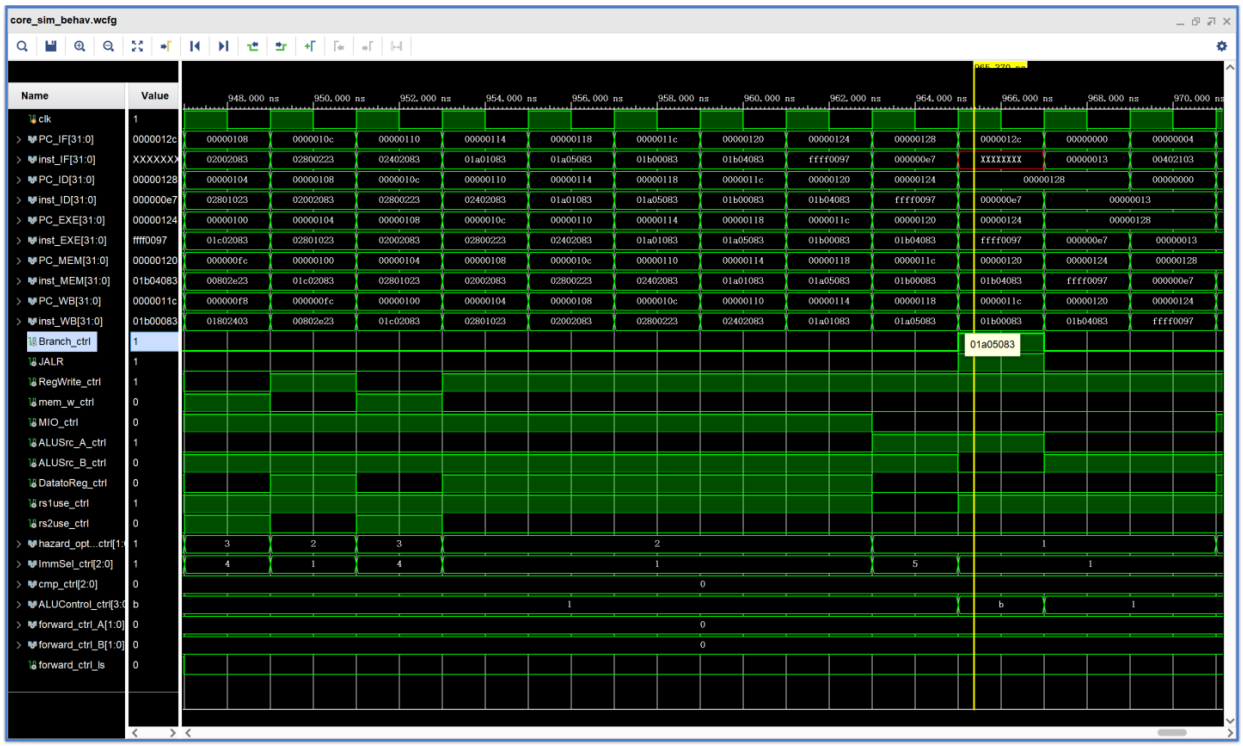
这里在 81ns，87ns，93ns 也发生了跳转，原因同上，这里不再重复。

```
addi x0,x0,0        # PC=D4
addi x0,x0,0
label5:
bge  x4,x4,label6   # PC=DC
addi x0,x0,0        # PC=E0
addi x0,x0,0
label6:
lui  x1,4           # PC=E8
jal  x1,12
addi x0,x0,0        # PC=F0
addi x0,x0,0
lw   x8, 24(x0)     # PC=F8
sw   x8, 28(x0)
lw   x1, 28(x0)
sh   x8, 32(x0)
lw   x1, 32(x0)
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`。

```
lw   x1, 32(x0)      # PC=108
sb   x8, 36(x0)
lw   x1, 36(x0)
lh   x1, 26(x0)
lhu  x1, 26(x0)
lb   x1, 27(x0)
lbu  x1, 27(x0)
auipc x1, 0xffff0
jalr x1,0(x0)        # PC=128
```

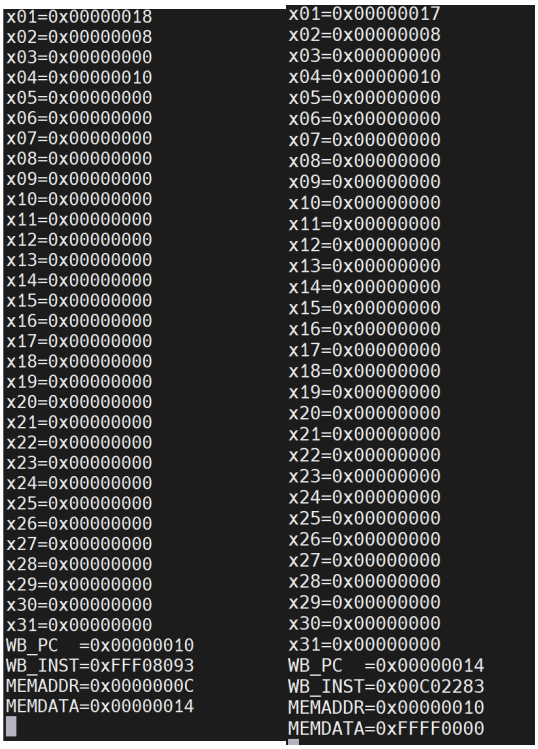


这里在 965ns 时，`jalr x1,0(x0)` 处于 ID 阶段，可以看到 `JALR=1`，随后这条指令在 EX 阶段被执行，计算出了要跳转的地址 0x0，因此可以看到 967ns 时 PC 回到了 0x0。

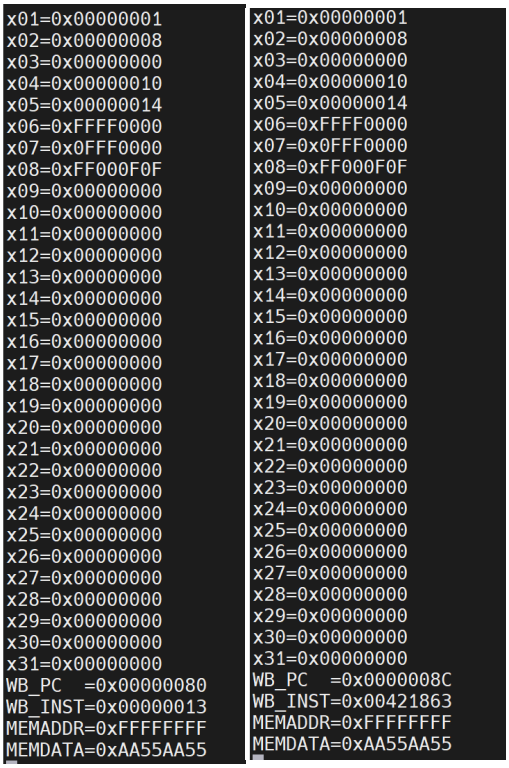
2.2 上板结果

`PC=4` 对应的指令是 `lw 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。

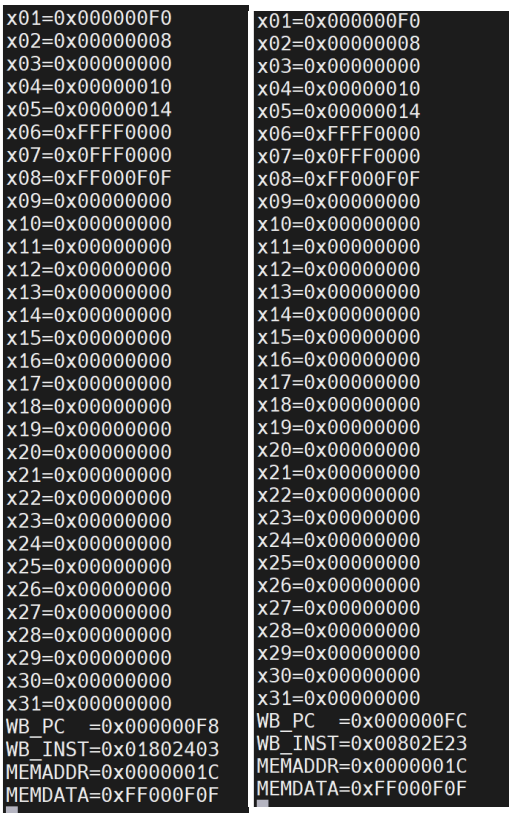
`PC=10` 对应的指令是 `add x1, x2, x4`，`PC=14` 对应的指令是 `addi x1, x1, -1`。这里发生了第一类数据冒险，可以看到第一条执行后 `x1` 的值为 18（`x4` 的值为 0x10，这也说明我们 `x2` 的数据冒险也成功解决），而第二条执行后 `x1` 的值为 17，说明我们解决了这个冒险。



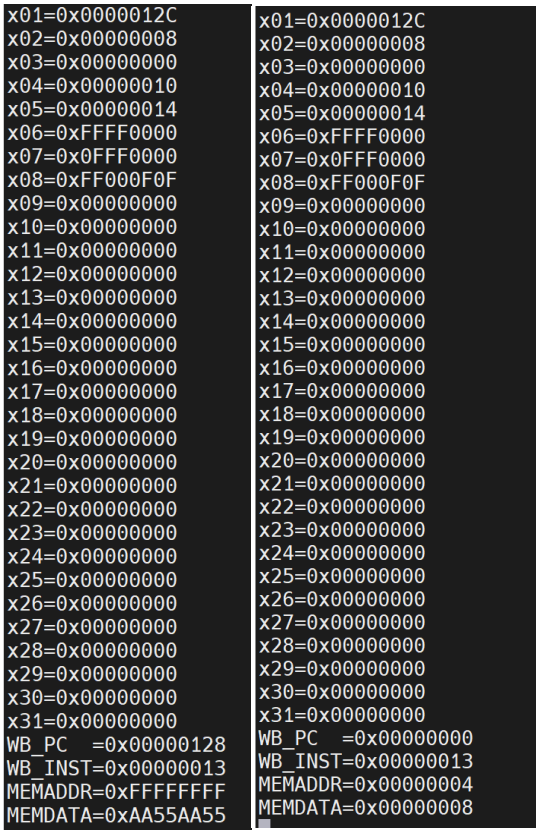
`PC=80` 对应的指令是 `beq x4 x4 12`，这里因为发生了跳转，因此我们需要插入一条 bubble，我们可以看到此时的 inst 为 NOP，说明我们的确插入了一条 bubble（这里我们没有改变 PC）。同时可以看到这条指令的下一条指令是 `PC=8C`，说明我们的确执行了跳转。



`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。



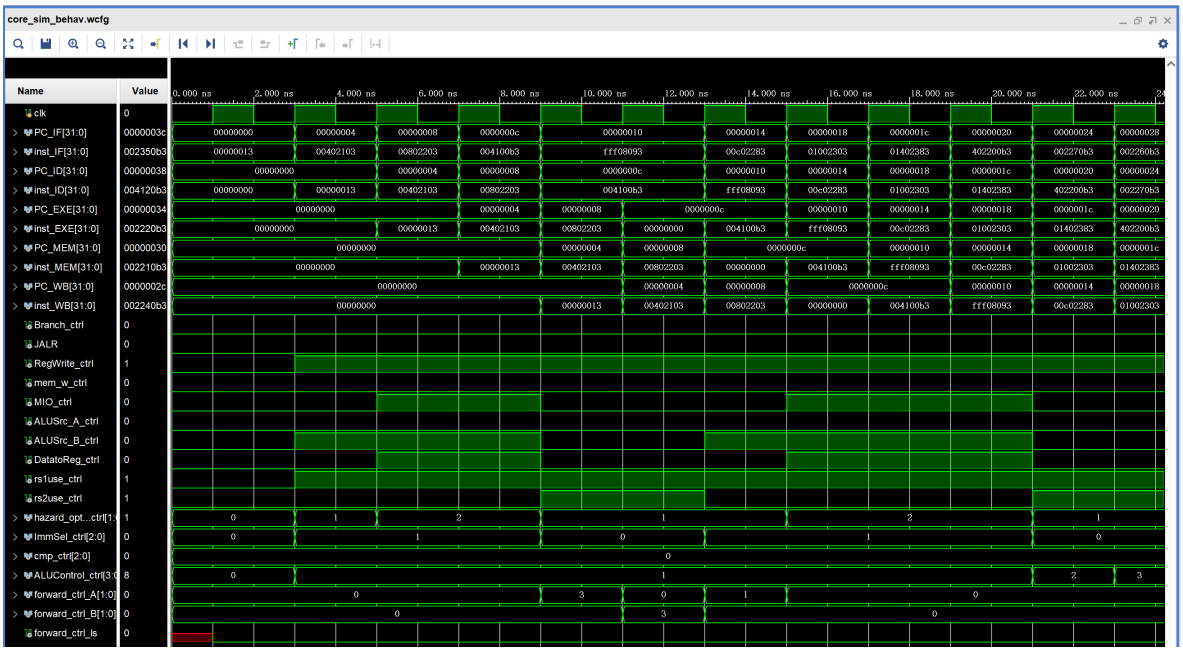
`PC=128` 对应的指令是 `jalr x1, 0(x0)`，这里也发生了跳转，因此插入了一条 `bubble`，同时我们可以看到下一条指令 `PC` 又变为了 `0x0`，说明跳转成功，而且我们将返回地址 `PC+4=12C` 写入了寄存器 `x1`。



3 思考题

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：

```
add x1, x2, x4          # PC=0xC
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：

```
lw x1, 0(x0)          # need a nop
add x2, x1, x1
```