

# 实验五 流水线 CPU

**noname**

2020 年 6 月 23 日

## 1 实验目的

1. 理解流水线 CPU 的组成结构和工作原理;
2. 掌握数字系统的设计和调试方法;
3. 熟练掌握数据通路和控制器的设计和描述方法.

## 2 逻辑设计

### 2.1 流水线 CPU(Pipeline CPU)

待设计的流水线 CPU 可以执行如下 6 条指令:

**add:**  $rd \leftarrow rs + rt;$   $op = 000000, funct = 100000$

op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5 bits)	funct(6 bits)
------------	------------	------------	------------	---------------	---------------

图 1: R-type 指令

**addi:**  $rt \leftarrow rs + imm;$   $op = 001000$

**lw:**  $rt \leftarrow M(rs + addr);$   $op = 100011$

**sw:**  $M(rs + addr) \leftarrow rt;$   $op = 101011$

**beq:**  $if(rs = rt) then pc \leftarrow pc + 4 + addr << 2$   
 $else pc \leftarrow pc + 4;$   $op = 000100$

op(6 bits)	rs(5 bits)	rt(5 bits)	addr/immediate(16 bits)
------------	------------	------------	-------------------------

图 2: I-type 指令

**j:**  $pc \leftarrow (pc + 4)[31 : 28]||(add << 2)[27 : 0];$   $op = 000010$

op(6 bits)	addr(26 bits)
------------	---------------

图 3: J-type 指令

根据上述指令的功能, 设计两种流水线 CPU 的不完全的数据通路如图-4 和图-5 所示, 其中橙色部分为控制器。具体实现时 ALU 和寄存器堆可以利用实验 1 和实验 2 设计的模块, 指令存储器 ROM 和数据存储器 RAM 均采用 IP 例化实现, 容量为  $256 \times 32$  位的分布式存储器。

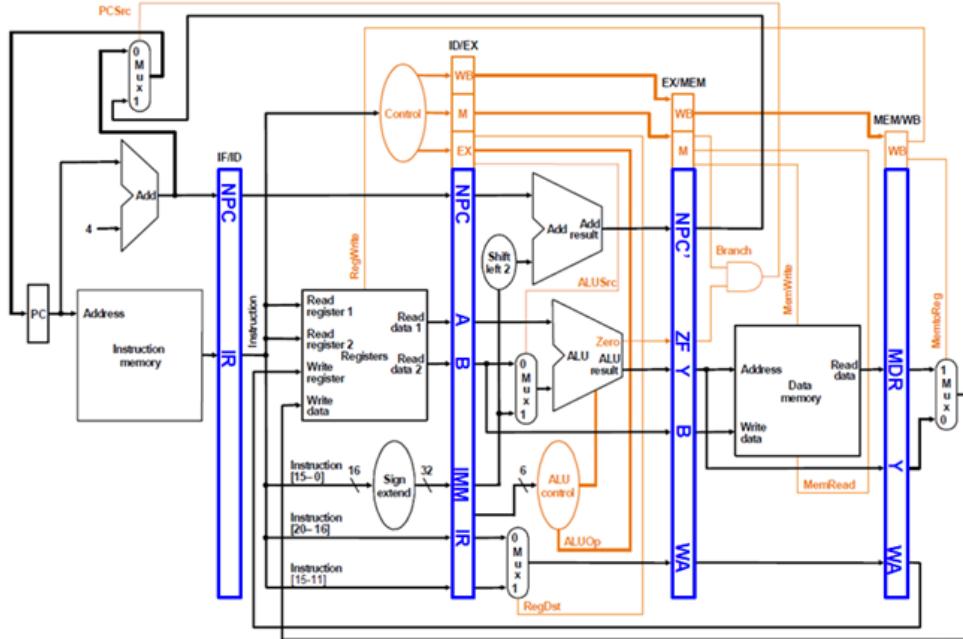


图 4: 无相关处理流水线数据通路

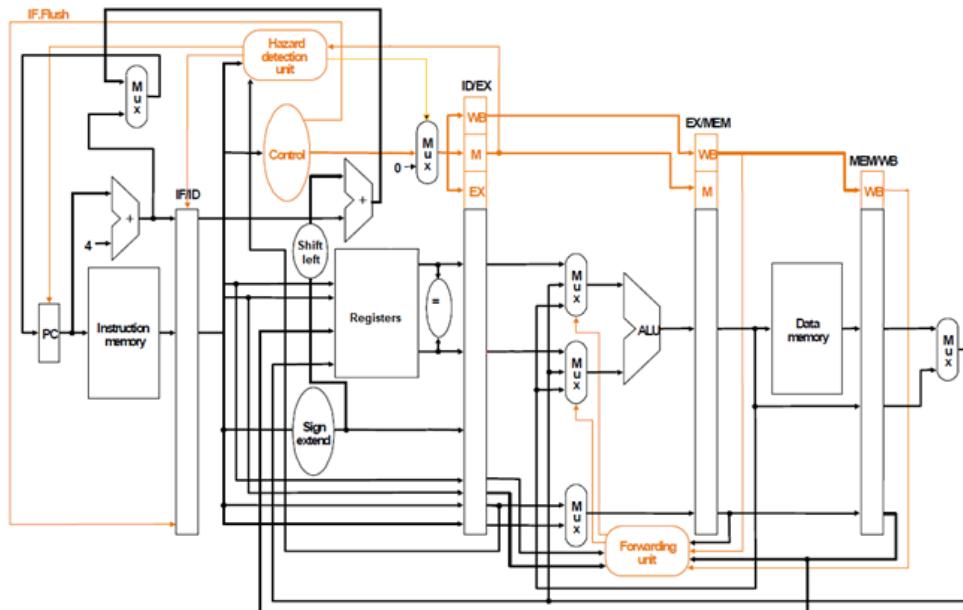


图 5: 有相关处理流水线数据通路

流水线各段操作 (无相关处理):

流水段	任何指令类型		
IF	$IF/ID. IR \leftarrow Mem[PC]; IF/ID. NPC \leftarrow PC+4;$ $PC \leftarrow (if PCSrc \{EX/MEM. NPC'\} else \{PC+4\});$		
ALU 指令(R类/I类)	Load/Store 指令	分支指令	
EX	$EX/MEM. IR \leftarrow ID/EX. IR;$ $EX/MEM. ALUOut \leftarrow ID/EX. A op ID/EX. B;$ 或 $EX/MEM. ALUOut \leftarrow ID/EX. A op ID/EX. Imm;$ $EX/MEM. cond \leftarrow 0;$	$EX/MEM. IR \leftarrow ID/EX. IR;$ $EX/MEM. B \leftarrow ID/EX. B;$ $EX/MEM. ALUOutput \leftarrow ID/EX. A + ID/EX. Imm;$ $EX/MEM. cond \leftarrow 0;$	$EX/MEM. NPC' \leftarrow ID/EX. NPC$ $+ ID/EX. Imm \ll 2;$ $EX/MEM. cond \leftarrow (ID/EX. A == ID/EX. B);$

图 6: 流水线各段各指令操作 (无相关处理)

流水段	任何指令类型		
ALU 指令	Load/Store 指令	分支指令	
MEM	$MEM/WB. IR \leftarrow EX/MEM. IR;$ $MEM/WB. ALUOut \leftarrow EX/MEM. ALUOut;$	$MEM/WB. IR \leftarrow EX/MEM. IR;$ $MEM/WB. MDR \leftarrow Mem[EX/MEM. ALUOut];$ 或 $Mem[EX/MEM. ALUOut] \leftarrow EX/MEM. B;$	$PCSrc \leftarrow EX/MEM. cond \& EX/MEM. Branch;$
WB	$Regs[MEM/WB. IR_{15..11}] \leftarrow MEM/WB. ALUOut; (R)$ 或 $Regs[MEM/WB. IR_{20..16}] \leftarrow MEM/WB. ALUOut; (I)$	$Regs[MEM/WB. IR_{20..16}] \leftarrow MEM/WB. MDR;$	

图 7: 流水线各段各指令操作 (无相关处理)(续)

根据数据通路, 有相关处理的流水线分支指令均在 ID 段结束, 其余各指令操作与无相关处理流水线基本相同

下按照顺序介绍各个模块的功能，代码在后面展示：

**pipeline CPU:** 主要负责连接各个模块成为一个完整的 CPU 以及接收 CLK 和 RST 信号

#### IF:

**MUX\_PC:** 用来选择 IF 段的 NPC。输入数据分别为 ID\_NPC(即 IF\_PC\_Plus4),ID\_NPC\_Branch,ID\_NPC\_Jump. 选择信号为 ID\_PCSrc.

**DFF\_PC:** 用来更新 PC.

无相关处理流水线每周期都要更新, 即 D 触发器 EN 恒为 1,RST 仅在按下 RST 按键时为 1  
有相关处理流水线根据 Hazard 模块译出控制信号 (PC\_EN) 选择是否更新或清零 PC

**InsMem:** 指令存储器.256\*32 位 ROM, 用来读出 PC 所对应指令 PC\_IR

**IF\_ID:** 段间寄存器.

根据 Hazard 模块译出控制信号 (IF\_ID\_FLUSH,IF\_ID\_EN) 将 IF 段数据传入 ID 段, 防止数据读取错误, 实现流水的基础. **ID\_EXE, EXE\_MEM, MEM\_WB:** 段间寄存器, 与 IF\_ID 功能相似.

#### ID:

**InsCut:** 指令分割. 将 32 位的指令分割成 ID\_OP, ID\_Rs, ID\_Rt, ID\_Rd, ID\_Imm, ID\_Addr, ID\_Funct 等, 便于使用

**ControlUnit:** 控制单元 (译码模块), 输入 ID\_Opcode, 对各控制信号赋值

**RegFile:** 寄存器堆 (两异步读, 一同步写). 用来存取数据 ID\_A, ID\_B. 如 WB\_RegWrite 为 1, 则在前半个周期写入 WB\_WriteData 到 WB\_WA 地址, 且增加限制条件使得 0 号寄存器值恒为 0

**ALU\_BEQ:** 相当于比较器, 用来判断 Rs 与 Rt 读出数据是否相等, 得出 ID\_Zero

**Hazard:** 冲突处理模块. 输入 Opcode 和

ID\_Rs, ID\_Rt, EXE\_WA, MEM\_WA, EXE\_MemRead, MEM\_MemRead.

如果是 BEQ 指令, 流水线暂停两个周期. 如果有相关 (LW 读出数据未写回且无法通过 Forward 解决), 则暂停一周期.

Jump 指令也暂停一周期.

#### EXE:

**Forward:** 定向路径模块, 用来解决未写回寄存器堆便使用数据的可用定向路径解决的数据相关.

如果 EXE\_Rs 或 EXE\_Rt 与 MEM\_WA 或 WB\_WA 相同, 则根据指令类型选择 ALUSrcA 和 ALUSrcB 为 MEM\_Y 或 WB\_WriteData 这两个将要写回但未写回 RegFile 的数据.

**注意! 若为 0 号寄存器, 则不定向**

**MUX\_WA:** 根据 EXE\_RegDst 译出 EXE\_WA

**ALU\_EXE:** EXE 阶段的 ALU 模块. 用来完成 ALU 运算.

#### MEM:

**DataMem:** 数据存储器.256\*32 位单口 RAM, 用来读出 ALU\_Res 地址所对应数据

MEM\_ReadData. 如 MEM\_MemWrite 为 1, 则写入 EXE\_B(寄存器堆读出数据)

#### WB:

**MUX\_MemtoReg:** 根据 WB\_MemtoReg 选择 WB\_WriteData

## 2.2 核心代码展示

仅展示有相关处理的流水线 CPU 核心代码。无相关处理流水线直接提交到附件中  
pipeline CPU: 端口定义部分不展示, 展示各模块连线情况和例化的单元

```

1 //IF
2 mux4 MUXPC(.sel(ID_PCSrc), .d0(IF_PC_Plus), .d1(ID_NPC_Branch), .d2
3     (ID_NPC_Jump), .d3(32'b0), .out(IF_NPC)); //选择NPC
4
5 DFF DFFPC(.CLK(~CLK), .en(PC_EN), .RST(RST), .DIn(IF_NPC), .DOOut(
6     IF_PC)); //更新PC
7 InsMem IM(.a(IF_PC[9:2]), .spo(IF_IR)); //读指令
8
9 ALU ALUPC_plus_4(.a(IF_PC), .b(32'h0004), .ALUOp(3'b000), .y(
10    IF_PC_Plus)); //计算PC+4
11
12 IF_ID IFID(
13     .CLK(CLK), .RST(RST || /*ID_EXE_FLUSH*/IF_ID_FLUSH)), .EN(
14         IF_ID_EN),
15     .IF_PC(IF_PC), .IF_IR(IF_IR), .IF_NPC(IF_PC_Plus), //这里写入PC+4为
16         了计算Beq和Jump
17     .ID_PC(ID_PC), .ID_IR(ID_IR), .ID_NPC(ID_NPC));
18
19 ///////////////////////////////////////////////////
20 //ID
21 //Jump和Beq在ID段写回新指令
22 InsCut IC(
23     .instruction(ID_IR), .op(ID_Opcode), .rs(ID_Rs), .rt(ID_Rt),
24     .rd(ID_Rd), .addr(ID_ADDR), .immediate(ID_IMM), .funct(ID_Funct));
25
26 Control_Unit CU(
27     .CLK(CLK), .RST(RST), .opcode(ID_Opcode), .funct(ID_Funct),
28     .RegDst(ID_RegDst), .ALUSrc(ID_ALUSrc),
29     .ALUOp(ID_ALUOp),
30     .Branch(ID_Branch), .Jump(ID_Jump), .MemWrite(ID_MemWrite), .
31         MemRead(ID_MemRead),
32     .RegWrite(ID_RegWrite), .MemtoReg(ID_MemtoReg));
33
34 assign ID_ReadReg1 = ID_Rs;
35 assign ID_ReadReg2 = ID_Rt;
36 assign ID_EXT_IMM = ID_IMM[15] ? {16'hffff, ID_IMM} : {16'h0000,

```

```

31      ID_IMM} ;

32      mux2 #(5) MUX_WA(. sel(ID_RegDst) , .d0(ID_IR[20:16]) , .d1(ID_IR
33          [15:11]) , .out(ID_WA)) ;

34      RegFile RF(. clk(CLK) , .ra0(ID_ReadReg1) , .ra1(ID_ReadReg2) ,
35          .rd0(ID_ReadData1) , .rd1(ID_ReadData2) ,
36          .wa(WB_WA) , .we(WB_RegWrite) , .wd(WB_WriteData)) ;

37      ALU ALU_ID_BEQ(.a(ID_ReadData1) , .b(ID_ReadData2) , .ALUOp(3'b001) ,
38          .zero(ID_Zero)) ;

39      assign ID_PCSrc = ID_Jump ? 2'b10 : (ID_Zero ? (ID_Branch ? 2'b01
40          : 2'b00) : 2'b00) ;
41      assign ID_NPC_Branch = ID_NPC + (ID_EXT_IMM<<2) ;
42      assign ID_NPC_Jump = {ID_NPC[31:28] , ID_IR[26:0] , 2'b00} ;
43      //ID段的相关处理通过Hazard暂停实现
44      Hazard Hazard(
45          //冲突处理
46          .CLK(CLK) , .RST(RST) , .ID_Opcode(ID_Opcode) ,
47          .ID_Rs(ID_Rs) , .ID_Rt(ID_Rt) , .EXE_WA(EXE_WA) , .MEM_WA(MEM_WA) ,
48          .EXE_MemRead(EXE_MemRead) , .MEM_MemRead(MEM_MemRead) ,
49          .PC_EN(PC_EN) , .IF_ID_EN(IF_ID_EN) , .ID_EXE_FLUSH(ID_EXE_FLUSH) ,
50          .Count(Count)) ;

51      assign IF_ID_FLUSH = (ID_PCSrc == 2'b00) ? 1'b0 : //正常
52          跳转不 stall
53          (ID_PCSrc == 2'b10) ? 1'b1 : //Jump stall一个周期
54          (Count == 3) ? 1'b1 : 1'b0 ; //Beq stall若有相关 stall两个周期

55      ID_EXE IDEXE(
56          .CLK(CLK) , .RST(RST || ID_EXE_FLUSH) ,
57          .ID_IR(ID_IR) , .ID_NPC(ID_NPC) ,
58          .ID_ReadData1(ID_ReadData1) , .ID_ReadData2(ID_ReadData2) , .
59              ID_EXT_IMM(ID_EXT_IMM) ,
60          .ID_RegDst(ID_RegDst) , .ID_ALUOp(ID_ALUOp) ,
61          .ID_ALUSrc(ID_ALUSrc) ,
62          .ID_MemWrite(ID_MemWrite) , .ID_MemRead(ID_MemRead) ,
63          .ID_RegWrite(ID_RegWrite) , .ID_MemtoReg(ID_MemtoReg) ,
64          .EXE_A(EXE_A) , .EXE_B(EXE_B) , .EXE_IR(EXE_IR) ,

```

```

65      .EXE_NPC(EXE_NPC) , .EXE_EXT_IMM(EXE_EXT_IMM) ,
66      .EXE_RegDst(EXE_RegDst) , .EXE_ALUOp(EXE_ALUOp) ,
67      .EXE_ALUSrc(EXE_ALUSrc) ,
68      .ID_WA(ID_WA) , .EXE_WA(EXE_WA) ,
69      .EXE_MemWrite(EXE_MemWrite) , .EXE_MemRead(EXE_MemRead) ,
70      .EXE_RegWrite(EXE_RegWrite) , .EXE_MemtoReg(EXE_MemtoReg) );
71      /////////////////////////////////
72
73      //EXE
74      assign EXE_Rs = EXE_IR[25:21];
75      assign EXE_Rt = EXE_IR[20:16];
76
77      Forwarding Forward(
78          //定向路径
79          .ID_Rs(ID_Rs) , .ID_Rt(ID_Rt) ,
80          .EXE_Rs(EXE_Rs) , .EXE_Rt(EXE_Rt) ,
81          .MEM_RegWrite(MEM_RegWrite) , .MEM_MemtoReg(MEM_MemtoReg) ,
82          .WB_RegWrite(WB_RegWrite) , .WB_MemtoReg(WB_MemtoReg) ,
83          .WB_WriteData(WB_WriteData) , .MEM_Y(MEM_Y) ,
84          .MEM_WA(MEM_WA) , .WB_WA(WB_WA) ,
85          .EXE_Forward_SelA(EXE_Forward_SelA) , .EXE_Forward_SelB(
86              EXE_Forward_SelB) ,
87          .ID_Forward_SelA(ID_Forward_SelA) , .ID_Forward_SelB(
88              ID_Forward_SelB)
89      );
90
91      mux4 #(32) MUX_Forward_A(
92          .sel(EXE_Forward_SelA) , .d0(EXE_A) , .d1(MEM_Y) ,
93          .d2(WB_WriteData) , .d3(32'b0) , .out /*EXE_Forward_A*/ /EXE_ALUA));
94      mux4 #(32) MUX_Forward_B(
95          .sel(EXE_Forward_SelB) , .d0(EXE_B) , .d1(MEM_Y) ,
96          .d2(WB_WriteData) , .d3(32'b0) , .out (EXE_Forward_B));
97      mux2 #(32) MUX_ALUB(.sel(EXE_ALUSrc) , .d0(EXE_Forward_B) , .d1(
98          EXE_EXT_IMM) , .out (EXE_ALUB));
99
100     EXE_MEMORY EXEMEM(
101         .CLK(CLK) , .RST(RST) ,

```

```

101      .EXE_IR(EXE_IR) , .EXE_B(EXE_B) ,
102      .EXE_ALURes(EXE_ALURes) ,
103      .EXE_WA(EXE_WA) ,
104      .EXE_MemWrite(EXE_MemWrite) , .EXE_MemRead(EXE_MemRead) ,
105      .EXE_RegWrite(EXE_RegWrite) , .EXE_MemtoReg(EXE_MemtoReg) ,
106      .MEM_IR(MEM_IR) , .MEM_Y(MEM_Y) , .MEM_B(MEM_B) ,
107      .MEM_WA(MEM_WA) ,
108      .MEM_MemWrite(MEM_MemWrite) , .MEM_MemRead(MEM_MemRead) ,
109      .MEM_RegWrite(MEM_RegWrite) , .MEM_MemtoReg(MEM_MemtoReg) ) ;
110
111
112      ///////////////////////////////////////////////////
113      //MEM
114
115      DataMem DM(.clk(CLK) , .we(MEM_MemWrite) , .a(MEM_Y[9:2]) ,
116      .d(MEM_B) , .spo(MEM_ReadData)) ;
117
118      MEM_WB MEMWB(
119      .CLK(CLK) , .RST(RST) ,
120      .MEM_ReadData(MEM_ReadData) , .MEM_Y(MEM_Y) ,
121      .MEM_WA(MEM_WA) ,
122      .MEM_RegWrite(MEM_RegWrite) , .MEM_MemtoReg(MEM_MemtoReg) ,
123      .WB_Y(WB_Y) , .WB_MDR(WB_MDR) ,
124      .WB_WA(WB_WA) ,
125      .WB_RegWrite(WB_RegWrite) , .WB_MemtoReg(WB_MemtoReg) ) ;
126
127
128      ///////////////////////////////////////////////////
129      //WB
130      mux2 Mux_MemtoReg( .sel(WB_MemtoReg) , .d0(WB_Y) , .d1(WB_MDR) , .out(
131          WB_WriteData)) ;
        endmodule

```

**Hazard:** 冲突处理

```

1 module Hazard(
2     input CLK,RST,
3     input [5:0] ID_Opcode,
4     input [4:0] ID_Rs, ID_Rt, EXE_WA, MEM_WA, WB_WA,
5     input ID_Branch, EXE_MemRead, MEM_MemRead, WB_MemRead,
6     output PC_EN, IF_ID_EN, ID_EXE_FLUSH,
7     output reg [2:0] Count

```

```

8 );
9
10      assign ID_EXE_FLUSH =
11      (
12          (((EXE_WA==ID_Rs) || (EXE_WA==ID_Rt)) && (EXE_MemRead||ID_Opcode
13              ==6'b000100))
14          ||
15          (((MEM_WA==ID_Rs) || (MEM_WA ==ID_Rt)) && (MEM_MemRead || ID_Opcode
16              ==6'b000100))
17      )
18      &&
19      (Count != 3); // 如果是LW相关或(BEQ且未暂停两周期),刷新寄存器堆数据
20          为0
21
22      assign IF_ID_EN = ~ID_EXE_FLUSH; // 条件成立则为0,保持寄存器不更新
23      assign PC_EN = ~ID_EXE_FLUSH; // 条件成立则为0,保持寄存器不更新
24
25      always@(posedge CLK or posedge RST)
26      begin
27          if (RST)
28              Count <= 0;
29          else if (((EXE_WA == ID_Rs) || (EXE_WA == ID_Rt))&&
30                  (ID_Opcode == 6'b000100) && (Count != 1) && (Count != 2)) // 实现2个
31                  周期暂停
32              Count <= 1;
33          else if (Count == 1)
34              Count <= 2;
35          else if (Count == 2)
36              Count <= 3;
37          else if (Count == 3)
38              Count <= 0;
39      end
40      endmodule

```

**Forward:** 定向路径

```

1 module Forwarding(
2     input [4:0] EXE_Rs, EXE_Rt, ID_Rs, ID_Rt,
3     input MEM_RegWrite, MEM_MemtoReg,
4     WB_RegWrite, WB_MemtoReg,
5     input [31:0] WB_WriteData, MEM_Y,
6     input [4:0] MEM_WA, WB_WA,

```

```

7 | output reg [1:0] EXE_Forward_SelA , EXE_Forward_SelB ,
8 | ID_Forward_SelA , ID_Forward_SelB
9 | );
10 |
11 // 根据情况给EXE_Forward_SelA和EXE_Forward_SelB赋值，用来选择ALUA和ALUB.
12 always @(*)
13 begin
14 EXE_Forward_SelA [1] = WB_RegWrite && (WB_WA != 0) && (MEM_WA != EXE_Rs) &&
15 (WB_WA == EXE_Rs) ;
16 EXE_Forward_SelA [0] = MEM_RegWrite && (MEM_WA != 0) && (MEM_WA == EXE_Rs) ;
17 EXE_Forward_SelB [1] = WB_RegWrite && (WB_WA != 0) && (MEM_WA != EXE_Rt) &&
18 (WB_WA == EXE_Rt) ;
19 EXE_Forward_SelB [0] = MEM_RegWrite && (MEM_WA != 0) && (MEM_WA == EXE_Rt) ;
20 |
21 end
22 endmodule

```

其余部分代码在附件内展示

### 2.3 test1 仿真波形结果分析

具体仿真结果的正确性在录制视频内已经讲过了，可以看出是按照汇编文件逐条执行的，且执行结果均正确

这里只放出仿真波形并进行简要分析

四条 ADDI 接两条 ADD 接 LW, 用到了定向路径

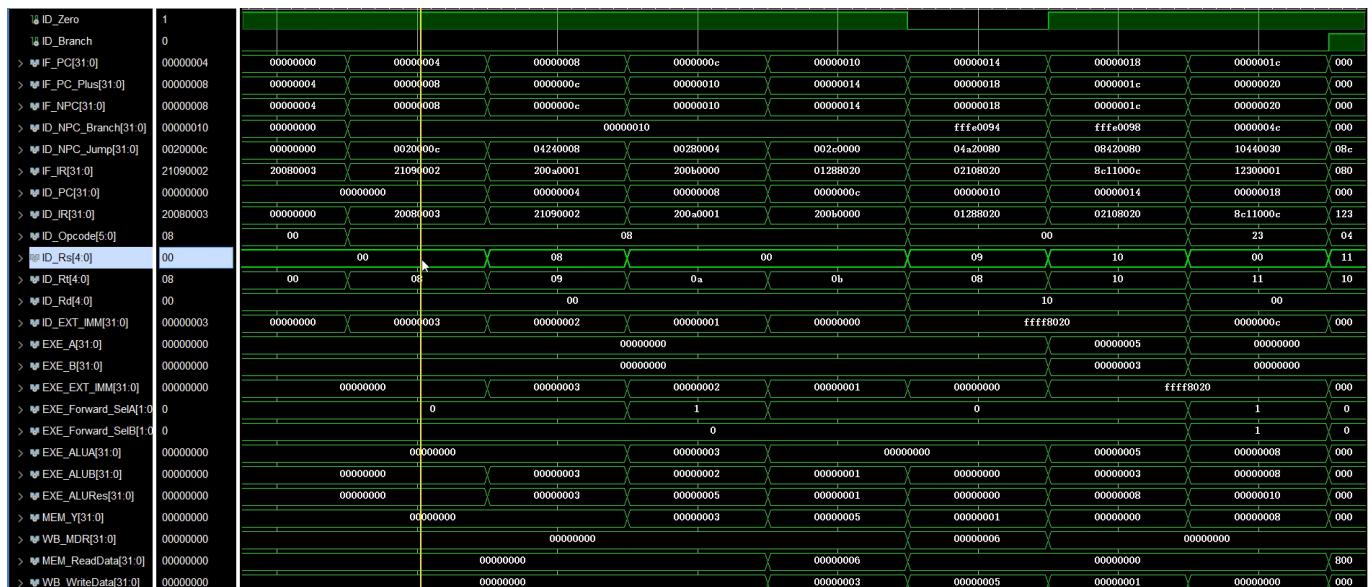


图 8: TEST1\_1

## BEQ 暂停两个周期并把 ID 段内容清空接两条 LW 指令

> IF_IR[31:0]	08000011	12300001	08000011		8c080010	8c090014	01288020	01288020	8c110018
> ID_PC[31:0]	0000001c	00000018	0000001c		00000000	00000024	00000028	00000028	0000002c
> ID_R[31:0]	12300001	8c11000c	12300001		00000000	8c080010	8c090014	01288020	01288020
> ID_Opcode[5:0]	04	23	04		00		23		00
> ID_Rs[4:0]	11	00	11		00		00		09
> ID_Rt[4:0]	10	11	10		00		08		08
> ID_Rd[4:0]	00	11	10		00		08		10
> ID_EXT_IMM[31:0]	00000001	0000000c	00000001		00000000	00000010	00000014	ffff8020	
> EXE_A[31:0]	00000000	00000000	00000000		00000010	00000000	00000000	00000000	00000000
> EXE_B[31:0]	00000000	00000000	00000000		00000010	00000000	00000003	00000005	00000000
> EXE_EXT_IMM[31:0]	00000000	ffff8020	0000000c		00000001	00000000	00000010	00000014	00000000
> EXE_Fwd_SelA[1:0]	0	1	0		0		0		0
> EXE_Fwd_SelB[1:0]	0	1	0		0		0		0
> EXE_ALUA[31:0]	00000000	00000008	00000000		00000010	00000000	00000000	00000000	00000000
> EXE_ALUB[31:0]	00000000	00000008	0000000c		00000000	00000000	00000014	00000000	00000000
> EXE_ALURes[31:0]	00000000	00000010	00000000		00000000	00000000	00000014	00000000	00000000
> MEM_Y[31:0]	00000000	00000008	0000000c		00000000	00000000	00000010	00000011	00000000
> WB_MDR[31:0]	00000010	00000000	80000000		00000000	00000000	00000000	00000000	80000000
> MEM_ReadData[31:0]	00000000	00000000	00000010		00000000	00000000	00000000	80000000	80000000
> WB_WriteData[31:0]	00000010	00000000	00000008		00000000	00000000	00000000	00000000	00000000
> EXE_WA[4:0]	00	10	11		00		x0		00
> WB_WA[4:0]	11	0b	10		00		x0		08
> MEM_WA[4:0]	00	10	11		00		x0		09
> ID_ReadData[31:0]	00000010	00000000	00000008		00000010	00000000	00000005	00000005	00000003
> ID_ReadData2[31:0]	00000010	00000000	00000000		00000010	00000000	00000003	00000005	00000003

图 9: TEST1\_2

两条 LW 指令接 ADD 指令, 由于 LW+RTYPE 相关, 暂停等待正确数据计算得出再定向路径, 再接 LW+BEQ

> ID_NPC_Jump[31:0]	04a20080	00000000	10200040	10240050		04a20080		10440060	08c00004
> IF_IR[31:0]	8c110018	8c080010	8c090014	01288020		8c110018		12300001	08000011
> ID_PC[31:0]	0000002c	00000000	00000024	00000028		00000024		00000030	00000034
> ID_R[31:0]	01288020	00000000	8c080010	8c090014		01288020		8c110018	12300001
> ID_Opcode[5:0]	00	00	23	00		00		23	04
> ID_Rs[4:0]	09	00	00	09		09		00	11
> ID_Rt[4:0]	08	00	08	09		08		11	10
> ID_Rd[4:0]	10	00	00	10		10		00	00
> ID_EXT_IMM[31:0]	ffff8020	00000010	00000014	00000000		ffff8020		00000018	00000001
> EXE_A[31:0]	00000000	00000010	00000000	00000000		00000000		80000000	00000000
> EXE_B[31:0]	00000000	00000010	00000000	00000003		00000000		80000000	00000010
> EXE_EXT_IMM[31:0]	00000000	00000001	00000000	00000010		00000000		ffff8020	00000018
> EXE_Fwd_SelA[1:0]	0	0	0	0		0		0	0
> EXE_Fwd_SelB[1:0]	0	0	0	0		0		0	0
> EXE_ALUA[31:0]	00000000	00000010	00000000	00000000		00000000		80000000	00000000
> EXE_ALUB[31:0]	00000000	00000000	00000000	00000000		00000000		80000000	00000000
> EXE_ALURes[31:0]	00000000	00000000	00000000	00000000		00000000		00000000	00000000
> MEM_Y[31:0]	00000000	00000000	00000000	00000000		00000000		00000000	00000000
> WB_MDR[31:0]	80000000	00000000	00000000	00000000		80000000		00000000	00000000
> MEM_ReadData[31:0]	00000000	00000000	00000000	00000000		00000000		00000000	00000000
> WB_WriteData[31:0]	80000000	00000000	00000000	00000000		80000000		00000000	00000000
> EXE_WA[4:0]	00	x0	00	08		09		10	11
> WB_WA[4:0]	09	00	x0	00		08		00	10
> MEM_WA[4:0]	00	00	x0	00		09		00	11
> ID_ReadData[31:0]	80000000	00000000	00000000	00000005		80000000		00000000	00000010
> ID_ReadData2[31:0]	80000000	00000000	00000003	00000005		80000000		00000000	00000010

图 10: TEST1\_3

## LW 接 BEQ,BEQ 暂停两周期跳转成功, 后接两条 ADD 和 BEQ

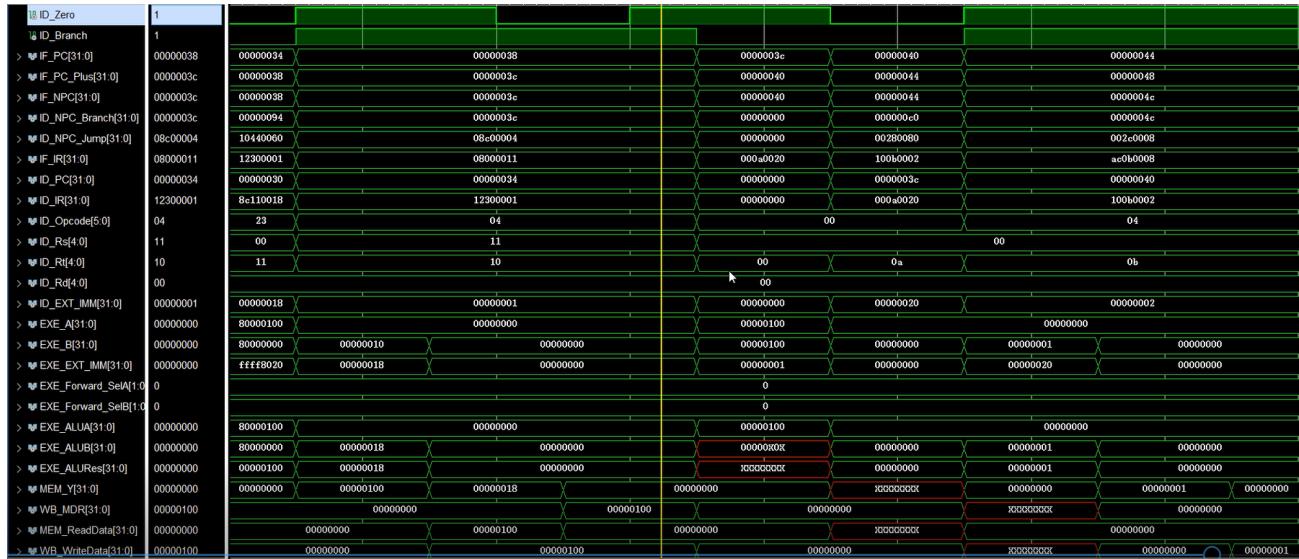


图 11: TEST1\_4

BEQ 跳转成功, 最后 CPU 在 SW 和 JUMP 间循环执行, 观察 MEM\_\_ReadData, 可知:0x08  
数据为 1, 测试通过



图 12: TEST1\_5

## 2.4 test2 仿真波形结果分析

ADD 接 JUMP 暂停一周期接两条 ADDI 接 ADD(用到了定向路径)

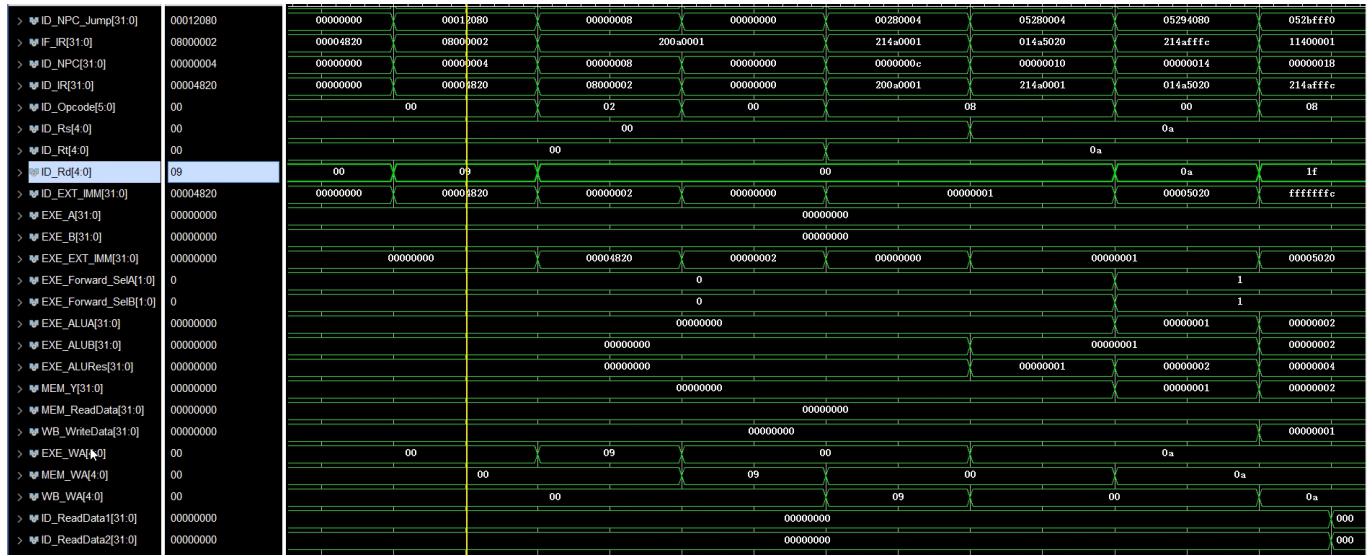


图 13: TEST2\_1

ADDI 接 BEQ 暂停两周期接 ADDI 接 JUMP

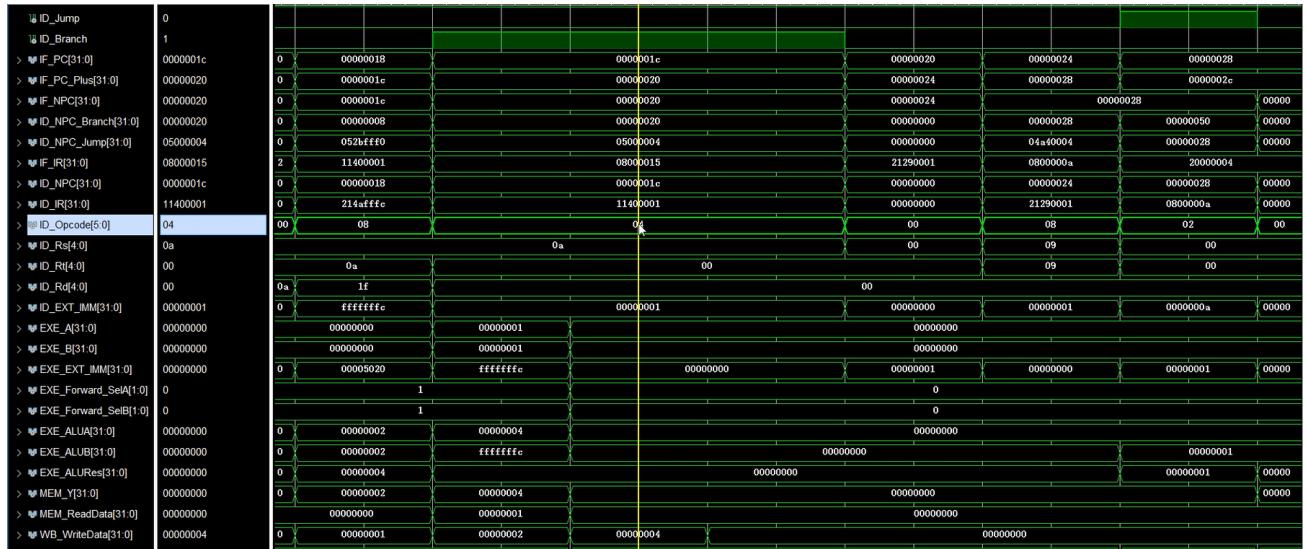


图 14: TEST2\_2

## JUMP 暂停一周期接 ADDI 接两条 LW+ADD

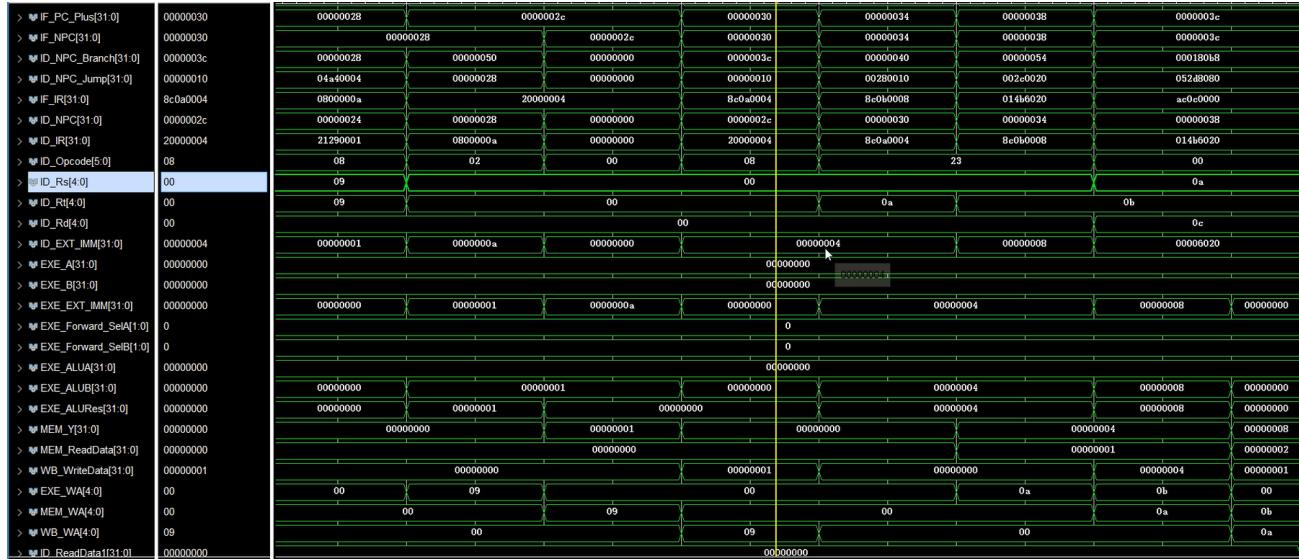


图 15: TEST2\_3

## ADD 暂停等待 LW 算出结果后接 SW 和两条 LW

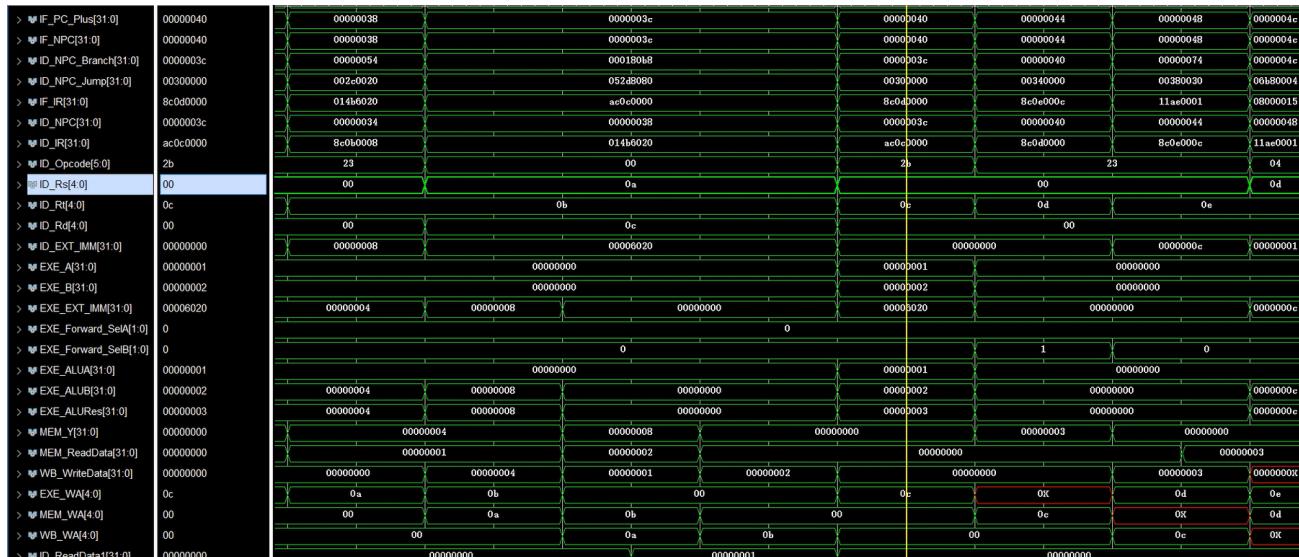


图 16: TEST2\_4

## 两条 LW 接 BEQ,BEQ 暂停 2 周期 Rs,Rt 数据正确, 成功跳转

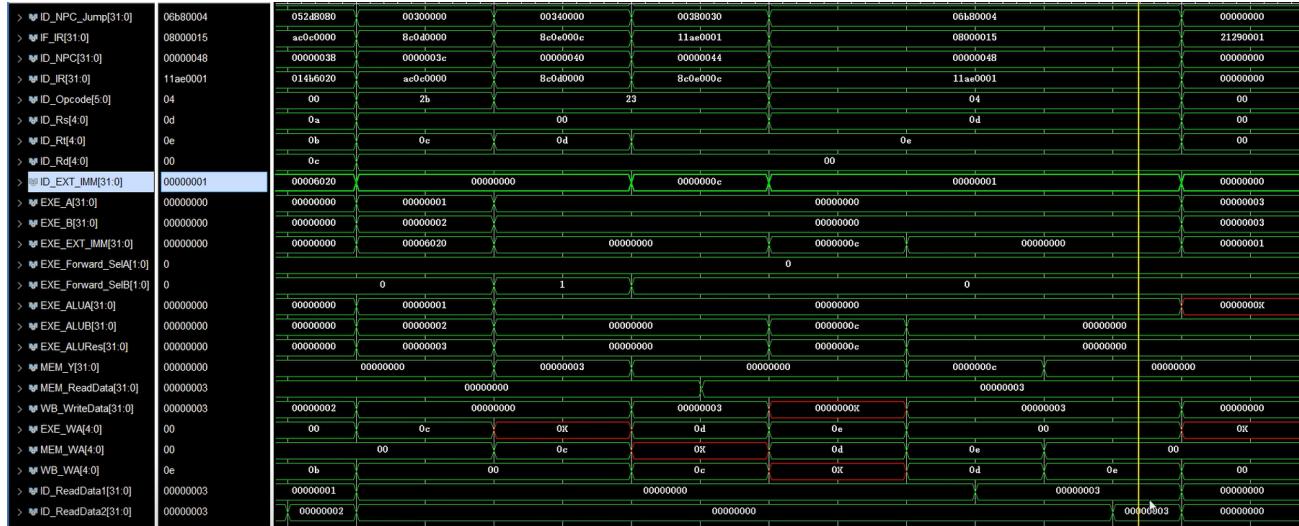


图 17: TEST2\_5

## BEQ 跳转成功执行一条 ADDI 指令, 之后两条 JUMP 指令循环跳转

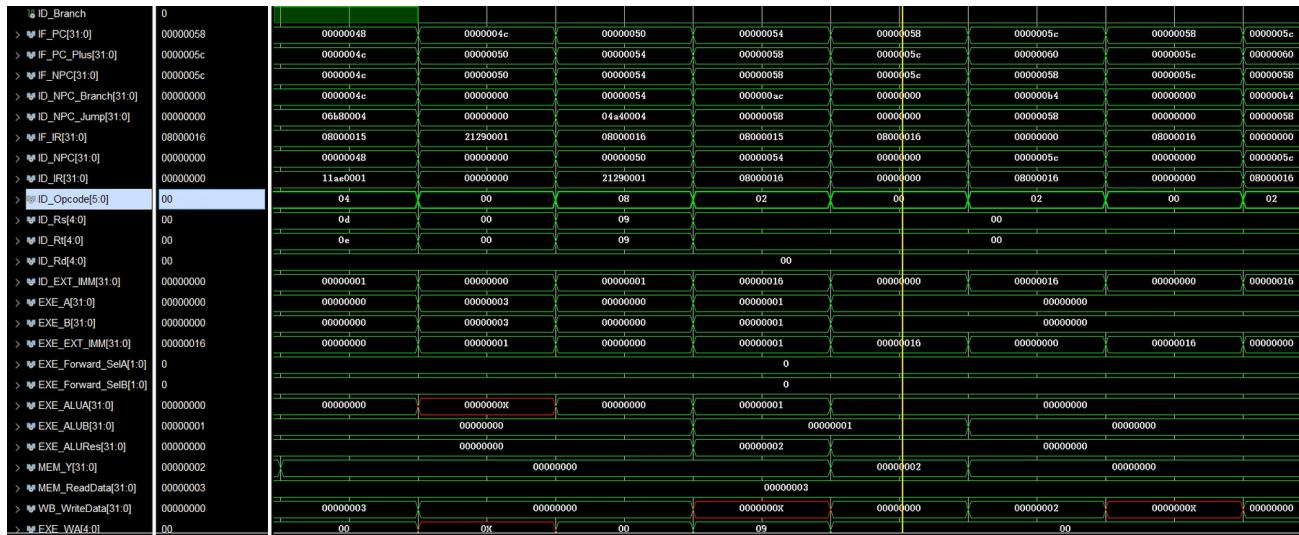


图 18: TEST2\_6

## 2.5 调试单元 (DeBug Unit)

为了方便下载调试，设计一个调试单元 DBU，该单元的功能和结构与实验四类似，可以用于控制 CPU 的运行方式，显示运行过程的中间状态和最终运行结果。DBU 的端口与 CPU 以及 FPGA 开发板外设 (拨动/按钮开关、LED 指示灯、7-段数码管) 的连接如下图所示。为了 DBU 在不影响 CPU 运行的情况下，随时监视 CPU 运行过程中寄存器堆和数据存储器的内容，可以为寄存器堆和数据存储器增加 1 个用于调试的读端口。

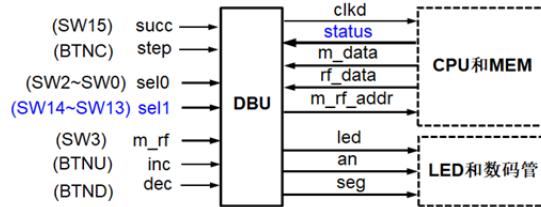


图 19: DBU 连线

### 控制 CPU 运行方式

succ = 1: clkd 输出连续的周期性脉冲信号，可以作为 CPU 的时钟信号，控制 CPU 连续执行指令  
 succ = 0: 每按动 step 一次，clkd 输出一个脉冲信号，可以作为 CPU 的时钟信号，控制 CPU 执行一个时钟周期

sel0 = 0: 查看 CPU 运行结果 (存储器或者寄存器堆内容)

m\_rf: 1, 查看存储器 (MEM); 0, 查看寄存器堆 (RF)

m\_rf\_addr: MEM/RF 的调试读口地址 (字地址)，复位时为零

inc/dec: m\_rf\_addr 加 1 或减 1

rf\_data/m\_data: 从 RF/MEM 读取的数据字

16 个 LED 指示灯显示 m\_rf\_addr

8 个数码管显示 rf\_data/m\_data

sel0 = 1 ~ 7: 查看 CPU 运行状态 (status)

根据 sel0 (选择流水段) 和 sel1 (选择相应段中寄存器)，选择一个 32 位数据显示在 8 个数码管上  
 sel0 = 1: PC, 程序计数器

sel0 = 2: IR/ID, sel1=0, NPC; sel1=1, IR

sel0 = 3: ID//EX, ....

sel0 = 4: EX/MEM, ....

sel0 = 5: MEM/WB, ....

sel0 = 6:

sel0 = 7:

根据需要在 16 个 LED 指示灯 (SW15~SW0) 上显示选中流水段的控制信号

未要求仿真，作为附件

## 3 实验结果

经波形分析，流水线 CPU 通过了两组测试，各指令各周期功能均正确实现

## 4 思考题

### 4.1 支持分支预测的流水线 CPU 的设计，并进行功能仿真和下载测试

分支预测仅需将 BEQ 暂停的周期 IF 段读出 PC+4 即可，如果 BEQ 跳转失败，那么继续执行。如果跳转成功，则清空 IF\_ID，ID\_EXE 段间寄存器内容，执行跳转成功 PC 对应的指令。

仅提供思路，未实现。

## 5 意见与建议

本次实验难度很大，数据通路不完整且未提供无相关处理流水线的 test 文件。BEQ 和 JUMP 提到 ID 段执行与使得 BEQ 的 Forward 只得通过暂停实现，加大了 Hazard 模块设计难度。DeBug 难度较大，高低电平的选择和莫名其妙的数据更新错误很让人崩溃。建议数据通路给全和给水平不高的同学无相关处理的测试文件，使大家都能正确完成实验。