浙江大学

本科实验报告

课程名称：	计算机组成与设计
姓　　名：	熊子宇
学　　号：	3200105278
学　　院：	竺可桢学院
专　　业：	混合班
指导教师：	姜晓红/叶大源
报告日期：	2022 年 5 月 19 日

实验五 Pipelined CPU

一、实验目的

0. 流水线 CPU 性能参数简介与实验思路梳理
1. 添加五级锁存器所需信号
2. 设计 Stall 信号
3. 实现五级锁存器的时序逻辑连线
4. 修改 ID 流水级的组合逻辑连线
5. 修改 EX 流水级的组合逻辑连线
6. 修改 MEM 流水级的组合逻辑连线
7. 修改 WB 流水级的组合逻辑连线
8. 撰写测试程序，上板验证现象

二、实验方法与步骤

0 流水线 CPU 性能参数简介与实验思路梳理

在本实验中，我计划完成一个最基本的流水线 CPU。它的性能参数如下：

1. 五级流水线，分成 IF, ID, EX, MEM, WB 五个阶段。

2. 结构竞争：IMem 和 DMem 分开，从而解决了 IF 和 MEM 阶段对内存访问的结构竞争；RegFile 在负边沿写，从而解决了在同一个时钟周期对 RegFile 既写又读的结构竞争。

3. 数据竞争：顺序执行的流水线 CPU 只会在 RAW(Read After Write)的情况下可能出现数据竞争。数据竞争的情形有两种（如下所示）。而我处理数据竞争的方式是 stall 流水线。

第一种如下。这种情况需要 stall 两拍。

```
sra     t6, a0, a2
and     s2, t6, t2
```

第二种情况如下。这种情况需要 stall 一拍。

```
sll     t2, a2, a2
slt     t3, a7, s2
sw      t2, 0(gp)
```

关于如何 stall 将在第 2 部分详述。

4. 控制竞争：我的 CPU 采取的是最基本的策略：只要在 EX 或 MEM 阶段检测到是 branch_taken/jal/jalr 指令，则 stall 三拍。pc_branch 在 EX 阶段计算出来，在 WB 阶段回写至 PC 中。没有将 Comparator 提前至 ID 阶段，也没有实现 predict not taken 策略。

动手实验之前，首先理清楚本实验的思路。流水线 CPU 利用了单周期 CPU 中的所有部件，如 IMem, DMem, RegFile, ImmGen, Controller, ALU, Comparator。这些部件内部的逻辑全部都不需要改动，可以直接迁移。流水线 CPU 与单周期 CPU 的区别在于：

1. 增加锁存器和控制信号：在单周期 CPU 的数据通路中添加了 4 个 Latch，与 PC 一起实现了五级分级。因此在每一级中都增加了部分信号（包括所需要的控制信号和运算结果），以传递给下一级。

2. 增加 Stall 信号：流水线 CPU 中会出现数据竞争和控制竞争，需要增加 Stall 信号来处理竞争。

3. 修改部件的输入输出信号：虽然每一个部件的内部逻辑都不变，但是留给外界的输入输出接口需要修改为对应流水级的信号。

4. 删除单周期 CPU 中的异常中断组件和信号，包括 CSR 和所有相关信号、异常和中断的判断信号等。这是因为在流水线 CPU 中不要求实现异常和中断，而单周期的异常中断逻辑在流水线中也并不适用。

所以在实验最开始，我先拷贝了一份单周期 CPU 工程，并且删除了所有异常中断的部件和信号。以下所有步骤都是基于“改造单周期 CPU”的思路实现的。

1 添加五级锁存器所需信号

在 Defines.vh 中，助教给出了 vga_debug 信号和实际信号的赋值，为我设计分级信号提供了一定的参考。先将需要在板子上呈现的实际信号在 Core.v 中定义好，并根据实际需要再增加一些信号。

助教提供的 Assignment 提示如下：

```
`define VGA_DBG_Core_Assignments \
   assign dbg_pc = pc; \
   assign dbg_inst = inst; \
   assign dbg_IfId_pc = IfId_pc; \
   assign dbg_IfId_inst = IfId_inst; \
   ...
```

PC Latch（在单周期 CPU 中也有）：

```
reg [31:0] pc;
wire [31:0] inst;
```

IF/ID Latch：

```
reg [31:0] IfId_pc;
reg [31:0] IfId_inst;
reg IfId_valid;
```

这里 valid 信号是用来指示该级是否为 NOP。

ID/EX Latch：

```
reg [31:0] IdEx_pc;
reg [31:0] IdEx_inst;
reg IdEx_valid;
reg [4:0] IdEx_rd;
reg [4:0] IdEx_rs1;
reg [4:0] IdEx_rs2;
reg [31:0] IdEx_rs1_val;
reg [31:0] IdEx_rs2_val;
reg IdEx_reg_wen;
reg [31:0] IdEx_imm;
reg IdEx_mem_wen;
reg IdEx_mem_ren;
reg IdEx_is_branch;
reg IdEx_is_jal;
```

```
reg IdEx_is_jalr;
reg IdEx_is_auipc;
reg IdEx_is_lui;
reg [3:0] IdEx_alu_ctrl;
reg [2:0] IdEx_cmp_ctrl;
reg IdEx_is_imm;
```

除了 pc、inst、valid 以外，可以看到在 ID 阶段由 Controller 译码产生了大量控制信号，包括 is_xxx（opcode 指示）系列，RegFile 相关信号(rd, rs1, rs2, regWen)，DMem 相关信号(mem_wen, mem_ren)，ALU 和 Comparator 相关信号(alu_ctrl, cmp_ctrl)，立即数信号(is_imm, imm[31:0])。以及由 RegFile 读出的 rs1_val 和 rs2_val。

EX/MEM Latch：

```
reg [31:0] ExMa_pc;
reg [31:0] ExMa_inst;
reg ExMa_valid;
reg [4:0] ExMa_rd;
reg ExMa_reg_wen;
reg [31:0] ExMa_mem_w_data;
reg [31:0] ExMa_alu_res;
reg ExMa_mem_wen;
reg ExMa_mem_ren;
reg ExMa_is_jal;
reg ExMa_is_jalr;
reg ExMa_is_lui;
reg ExMa_is_auipc;
reg [31:0] ExMa_imm;
reg ExMa_branch_taken;
reg [31:0] ExMa_rs2_val;
```

除了 pc、inst、valid 以外，分为三类信号：

- EXE 吸收了一部分信号，如 ALU 和 Comparator 控制信号、rs1, rs2, rs1_val 不需要继续传递。

- 还有一部分控制信号要继续传递，如 mem_ren, mem_wen, imm, UJ 指令和 U 指令，这几个信号要在 MEM 用到；rd 和 reg_wen 也要继续传递，因为要在 WB 用到。

- 同时生成了新的信号要送给 MEM 阶段，如 ALU 计算结果 alu_res, branch 是否跳转的 branch_taken 信号，DMEM 写入数据 mem_w_data。

MEM/WB Latch：

```
reg [31:0] MaWb_pc;
reg [31:0] MaWb_inst;
reg MaWb_valid;
reg [4:0] MaWb_rd;
reg MaWb_reg_wen;
reg [31:0] MaWb_reg_w_data;
```

继续向 WB 阶段传递 rd, reg_wen 信号，同时也要传递在 MEM 阶段得到的 RegFile 写入数据 reg_w_data。

2 设计 Stall 信号

2.1 Data Hazard

出现数据竞争的原因是发生 RAW。粗略地说，是 EX 级或 MEM 级的 rd 信号等于 ID 级的 rs1 或者 rs2。但是，仅有一部分指令解码出的 rd, rs1, rs2 是有效的。

EX/MEM 级的 rd 是否有效可以用对应级的 reg_wen 信号判断。

有 rs1 的指令：R, I, S, B Type

有 rs2 的指令：R, S, B Type

此外，还要注意 rd 应不为 0，因为 x0 寄存器只读不写。

总结来看，data_hazard 信号应如下：

```
    assign data_hazard = (IdEx_reg_wen | ExMa_reg_wen)
        && ( ((is_aluRR || is_aluRI || mem_ren || is_jalr || mem_wen ||
is_branch) && ((IdEx_rd != 0 && IdEx_rd == rs1) || (ExMa_rd != 0 && ExMa_rd
== rs1)) )
        || ( (is_aluRR || mem_wen || is_branch) && ((IdEx_rd != 0 && IdEx_rd
== rs2) || (ExMa_rd != 0 && ExMa_rd == rs2)) ) );
```

注意两点：第一，is_load 可以用 mem_ren 替代，is_store 可以用 mem_wen 替代；第二，这里逻辑写得比较复杂，是为了方便理解。实际上可以用卡诺图或布尔代数的方式简化表达式。

2.2 Control Hazard

控制竞争的逻辑较为简单。若 EX/MEM 级将发生有条件/无条件跳转，则产生 control_hazard 信号。如下：

```
    assign control_hazard
    = (IdEx_is_branch & cmp_res) | ExMa_branch_taken | IdEx_is_jal |
ExMa_is_jal | IdEx_is_jalr | ExMa_is_jalr;
```

注意 cmp_res 信号是 EX 级实时生成的，而其他信号都是由上一级锁存器传递而来的。

定义 stall 信号：

```
    assign stall = data_hazard | control_hazard;
```

3 实现五级锁存器的时序逻辑连线

3.1 PC Latch

PC 锁存器与单周期中基本一致，只是多了 data hazard 的判断。当 rst 信号为 1 时全部信号置 0。当 clk 正边沿且不为数据竞争时，将 pc <= pc_branch。这里需要注意，当发生控制竞争时，应当允许 pc 改变，否则无法做到 branch 的 WB 阶段和跳转地址指令的 IF 阶段同时执行。

```
/** PC Latch */
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            pc <= 0;
        end
        else begin
            if (~data_hazard) pc <= pc_branch; // 如果发生 data_hazard 则保
持不变, control_hazard 可以改变
        end
    end
```

3.2 IF/ID Latch

如下是 IF/ID Latch 的 clk 部分（rst 重置部分省略）。

当未发生竞争时，`IfId_pc <= pc; IfId_inst <= inst;`

当发生数据竞争时，`IfId_pc` 和 `IfId_inst` 保持不变。

当发生控制竞争时，`ID`阶段的指令应当清零。`IfId_pc <= 0;IfId_inst <= `NOP_INST;` 而且 `valid` 信号置 `0`，表明由硬件插入 `NOP`。

```
/** IF/ID Latch */
IfId_valid <= ~control_hazard; // 只有 control_hazard 会清空 ID
if (control_hazard) begin
    IfId_pc <= 0;
    IfId_inst <= `NOP_INST;
end
    else if (~stall) begin
    IfId_pc <= pc;
    IfId_inst <= inst;
end
// 如果是 data hazard 则保持不变
```

3.3 ID/EX Latch

如下是 ID/EX Latch 的 clk 部分（rst 重置部分省略）。

当未发生竞争时:

- pc, inst 接收 ID 传递的同名信号。

- rd, rs1, ...控制信号赋值为 ID 阶段产生的同名组合逻辑信号。注意这些组合逻辑信号将由 IfId_inst 产生。

当发生竞争时（数据竞争和控制竞争），`EX` 阶段的指令应当清零。`Valid` 信号置 `0`，表明由硬件插入 `NOP`。

```
IdEx_valid <= ~stall;
if (stall) begin
   IdEx_pc <= 0;
   IdEx_inst <= `NOP_INST;
   IdEx_rd <= 0;
   // ... 省略清 0
end
else begin
   IdEx_pc <= IfId_pc;
   IdEx_inst <= IfId_inst;
   IdEx_rd <= IfId_inst[11:7];
   IdEx_rs1 <= IfId_inst[19:15];
   IdEx_rs2 <= IfId_inst[24:20];
   IdEx_rs1_val <= rs1_val;
   IdEx_rs2_val <= rs2_val;
   // ... IdEx_yyy <= yyy; 同名变量赋值省略
 end
```

3.4 EX/MEM Latch

如下是 EX/MEM Latch 的 clk 部分（rst 重置部分省略）。

EX 阶段与竞争无关，因此 EX/MEM Latch 就是对应信号的赋值。此外还有在 EX 阶段新生成的 alu_res, branch_taken 信号要传递给 MEM 阶段。

```
ExMa_pc <= IdEx_pc;
ExMa_inst <= IdEx_inst;
ExMa_valid <= IdEx_valid;
ExMa_rd <= IdEx_rd;
ExMa_reg_wen <= IdEx_reg_wen;
```

```
ExMa_mem_w_data <= dmem_o_data;
ExMa_alu_res <= alu_res;
ExMa_mem_wen <= IdEx_mem_wen;
ExMa_mem_ren <= IdEx_mem_ren;
ExMa_is_jal <= IdEx_is_jal;
ExMa_is_jalr <= IdEx_is_jalr;
ExMa_is_lui <= IdEx_is_lui;
ExMa_is_auipc <= IdEx_is_auipc;
ExMa_imm <= IdEx_imm;
ExMa_branch_taken <= IdEx_is_branch & cmp_res;
ExMa_rs2_val <= IdEx_rs2_val;
```

3.5 MEM/WB Latch

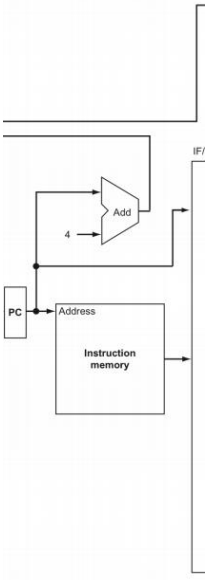
如下是 MEM/WB Latch 的 clk 部分（rst 重置部分省略）。

WB 阶段是最简单的。在前四个阶段，所有信号已经生成。要传递给 WB 阶段的只有寄存器的写相关信号。

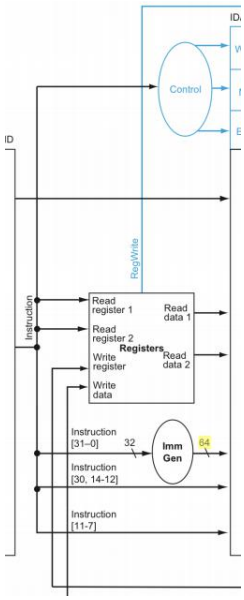
```
MaWb_pc <= ExMa_pc;
MaWb_inst <= ExMa_inst;
MaWb_valid <= ExMa_valid;
MaWb_rd <= ExMa_rd;
MaWb_reg_wen <= ExMa_reg_wen;
MaWb_reg_w_data <= reg_i_data;
```

4 修改 ID 流水级的组合逻辑连线

IF 阶段组件只有 PC 和 IMEM，没有需要修改之处。



IF 级组件：PC 和 IMEM



ID 级组件：Controller, RegFile, ImmGen

ID 阶段组件有 Controller, RegFile, ImmGen 三者。以下分别修改。

4.1 修改 Controller

在单周期 CPU 的基础上，调整所需要的控制信号如下：

```
/* Control Unit Signal */
wire reg_wen;
wire is_imm;
wire mem_wen; //MemWrite
wire mem_ren; //MemRead
// wire m2reg;  可以把 alu_res 和 dmem_o_data 二选一使用 mem_ren 来选择，
// 在 MEM 阶段就选好，则可以省略 m2reg 信号
wire [2:0] immType;
wire is_aluRI;
wire is_aluRR;
wire is_branch;
wire is_jal;
wire is_jalr;
wire is_auipc;//add upper imm20 to pc
wire is_lui;
wire [2:0] cmp_ctrl;
wire [3:0] alu_ctrl;
```

注意单周期 CPU 中的 m2reg 可以省略，因为 m2reg = mem_ren。少一个信号则可以减轻三个锁存器的负载。

实例化 Controller 模块如下。最需要注意的是在单周期 CPU 中的 inst 信号应该改为 IfId_inst。

```
Controller control_unit(
       .inst(IfId_inst),
       .alu_ctrl(alu_ctrl),
       .reg_wen(reg_wen),
       .is_imm(is_imm),
       .mem_wen(mem_wen),
       .mem_ren(mem_ren),
       .immType(immType),
       .is_branch(is_branch),
       .is_jal(is_jal),
       .is_lui(is_lui),
       .is_jalr(is_jalr),
       .is_auipc(is_auipc),
       .cmp_ctrl(cmp_ctrl),
       .is_aluRR(is_aluRR),
       .is_aluRI(is_aluRI)
);
```

Controller.v 内部的逻辑没有发生任何改变（除去删除不需要的信号），在 Lab4 的报告中已经比较详尽的写明，此处便省略。

4.2 修改 ImmGen

立即数发生器则更加简单，如下：

```
/** ImmGen */
wire [31:0] imm;

ImmGen imm_generator(
    .inst(IfId_inst),
    .immType(immType),
    .imm(imm)
);
```

4.3 修改 RegFile

RegFile 参与了 ID 和 WB 两阶段。在这里先定义好所有变量：

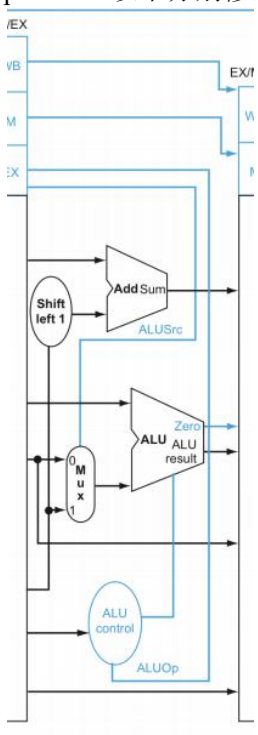
```
/** regFile*/
wire [4:0] rs1, rs2;
wire [31:0] rs1_val, rs2_val;
reg [31:0] reg_i_data;
```

RegFile 参与了 ID 和 WB 两阶段。在这里先定义好所有变量：RegFile 的读口、读出数据与 ID 级有关，先修改实例中的与读有关的参数：

```
assign rs1 = IfId_inst[19:15];
assign rs2 = IfId_inst[24:20];

RegFile reg_file(
       `VGA_DBG_RegFile_Arguments
       .clk(~clk),
       .rst(rst),
       .wen(?), //WB 部分再补充
       .rs1(rs1),
       .rs2(rs2),
       .rd(?),
       .i_data(?),
       .rs1_val(rs1_val),
       .rs2_val(rs2_val)
   );
```

5 修改 EX 流水级的组合逻辑连线

EX 阶段组件只有 ALU 和 Comparator。以下分别修改。



EX 级组件：ALU, Comparator

定义所需变量如下：

```
/** ALU and Comparator */
wire [31:0] a_val, b_val;
wire [31:0] alu_res;
wire cmp_res;
```

实例化 ALU 和 Comparator 模块如下。所有组合逻辑都没有改变，只不过 a_val, b_val

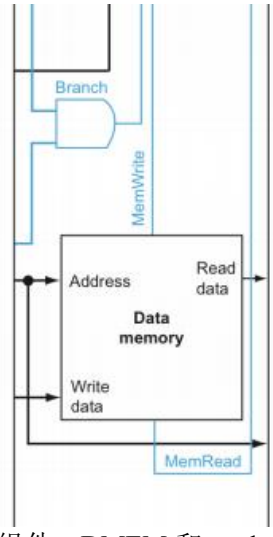
是由 ID 级的信号产生，ALU 和 Comparator 的控制信号也需要由 ID 级的信号控制。

```
assign a_val = IdEx_rs1_val;
assign b_val = IdEx_is_imm ? IdEx_imm : IdEx_rs2_val;

Alu alu(
    .a_val(a_val),
    .b_val(b_val),
    .ctrl(IdEx_alu_ctrl),
    .result(alu_res)
);

Comparator comparator(
    .a_val(a_val),
    .b_val(b_val),
    .ctrl(IdEx_cmp_ctrl),
    .result(cmp_res)
);
```

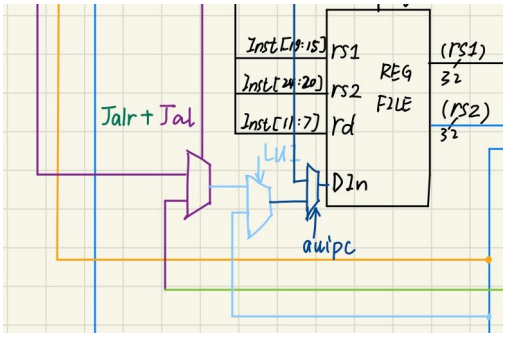
Controller.v 内部的逻辑没有发生任何改变（除去删除不需要的信号），在 Lab4 的报告中已经比较详尽的写明，此处便省略。

6 修改 MEM 流水级的组合逻辑连线

MEM 级组件：DMEM 和 pc_branch 计算、reg_i_data 计算。在图上只有 DMEM 和 pc_branch 计算，而我将图中 WB 阶段的 reg_i_data 计算也移到了 MEM 级中。



MEM 级组件：DMEM 和 pc_branch 计算



reg_i_data 计算

6.1 修改 DMEM

在单周期 CPU 的基础上，调整所需要的控制信号如下：

```
/* DMem */
assign dmem_addr = ExMa_alu_res;
assign dmem_i_data = ExMa_rs2_val;
assign dmem_wen = ExMa_mem_wen;
assign dmem_ren = ExMa_mem_ren;
```

主要的区别在于，赋值的信号是 EX/MEM 锁存器传递的。

6.2 pc_branch 计算

依据 opcode 信号和 branch_taken 信号，修改 pc_branch 如下：

```
reg [31:0] pc_branch;
always@* begin
    if (ExMa_is_jalr) pc_branch = ExMa_alu_res;
```

```
        else if (ExMa_branch_taken | ExMa_is_jal) pc_branch = ExMa_pc
+ ExMa_imm;
        else pc_branch = pc + 4;
    end
```

6.3 reg_i_data 计算

如上图所示，写口数据来源比较多，由很多二选一多路选择器组成。在 Verilog 中可以用 if 语句表现五种分支选择。

```
    always@* begin
        if (ExMa_mem_ren) reg_i_data = dmem_o_data;
        else if (ExMa_is_jal | ExMa_is_jalr) reg_i_data = ExMa_pc + 4;
        else if (ExMa_is_lui) reg_i_data = ExMa_imm;
        else if (ExMa_is_auipc) reg_i_data = ExMa_imm + ExMa_pc;
        else reg_i_data = ExMa_alu_res;
    end
```

7 修改 WB 流水级的组合逻辑连线

一者是 pc <= pc_branch，另一个是 RegFile 的回写。补充完整 RegFile 的实例：

```
    RegFile reg_file(
        `VGA_DBG_RegFile_Arguments
        .clk(~clk),
        .rst(rst),
        .wen(MaWb_reg_wen),
        .rs1(rs1),
        .rs2(rs2),
        .rd(MaWb_rd),
        .i_data(MaWb_reg_w_data),
        .rs1_val(rs1_val),
        .rs2_val(rs2_val)
    );
```

至此，所有修改已经完成，第 0 部分预期设定的功能也已完成。

8 撰写测试程序，上板验证现象

8.1 测试 I Type

首先做好准备，将 gp = 0xFE000000，以便于 LED 显示。

```
    lw      gp, 8(x0)           # gp = FE000000
```

I Type 的测试程序如下。测试了 addi 负数、sltiu 负数、负数的算术左右移等特殊情况。最后用 sw r1, 0(gp)指令将相应寄存器值的后八位显示在 LED 上便于观察。

在这些指令中，设置了一次 stall 的情况，用红色字体标出。引起的结果应是暂停 2 拍。

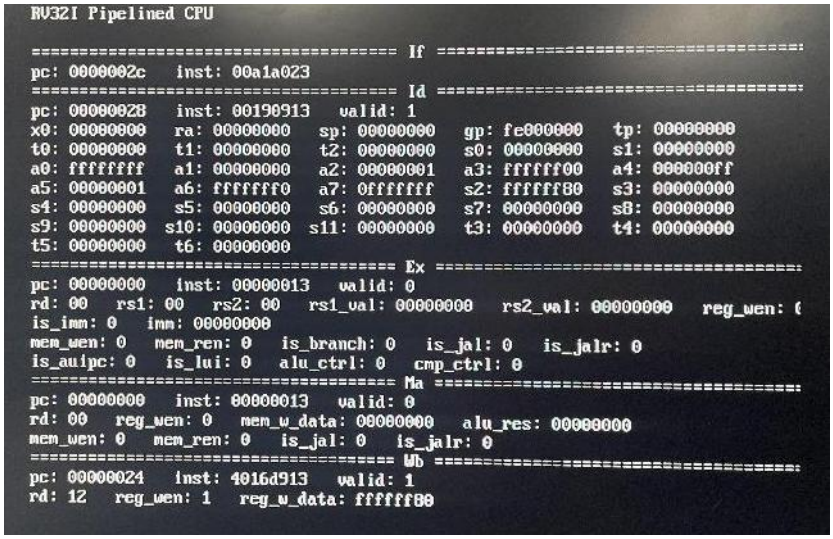
```
    # I type
    # 充满流水线，每一拍流出一个值
    # 没有 stall
        addi    a0, zero, -1        # a0 = 0xFFFF_FFFF
        slti    a1, zero, -2        # a1 = 0x0000_0000
        sltiu   a2, zero, -1        # a2 = 0x0000_0001
        xori    a3, a0, 0xFF        # a3 = 0xFFFF_FF00
        ori     a4, a1, 0xFF        # a4 = 0x0000_00FF
        andi    a5, a0, 0x1         # a5 = 0x0000_0001
        slli    a6, a0, 4           # a6 = 0xFFFF_FFF0
    # 负数的算术右移和逻辑右移
        srli    a7, a0, 4           # a7 = 0x0FFF_FFFF
        srai    s2, a3, 1           # s2 = 0xFFFF_FF80
```

```
# nop
# nop
addi    s2, s2, 1          # s2 = 0xFFFF_FFB1
sw      a0, 0(gp)          # LED = 8'b1111_1111
sw      a1, 0(gp)          # LED = 8'b0000_0000
sw      a2, 0(gp)          # LED = 8'b0000_0001
sw      a3, 0(gp)          # LED = 8'b0000_0000
sw      a4, 0(gp)          # LED = 8'b1111_1111
sw      a5, 0(gp)          # LED = 8'b0000_0001
sw      a6, 0(gp)          # LED = 8'b1111_0000
sw      a7, 0(gp)          # LED = 8'b1111_1111
sw      s2, 0(gp)          # LED = 8'b1000_0001
```

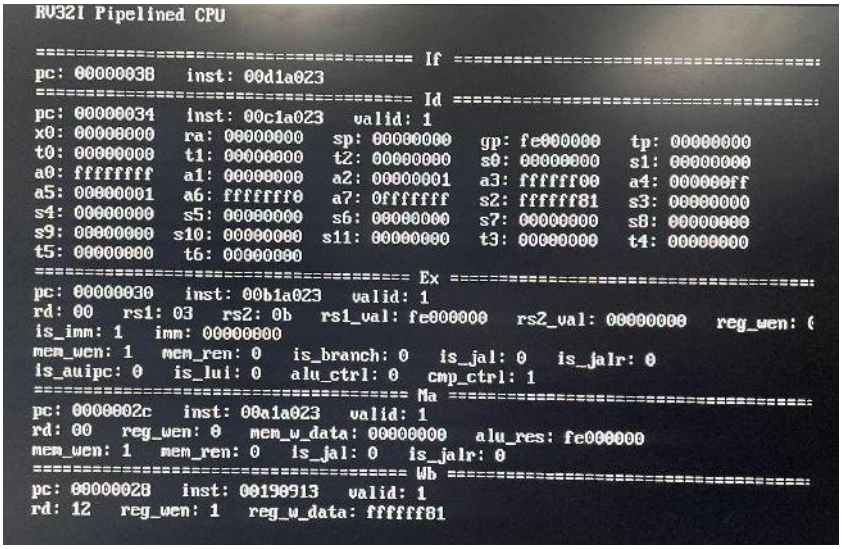
PC、inst 和指令对应如下：

```
0x00000000    0x00802183    lw x3 8(x0)
0x00000004    0xFFF00513    addi x10 x0 -1
0x00000008    0xFFE02593    slti x11 x0 -2
0x0000000C    0xFFF03613    sltiu x12 x0 -1
0x00000010    0x0FF54693    xori x13 x10 255
0x00000014    0x0FF5E713    ori x14 x11 255
0x00000018    0x00157793    andi x15 x10 1
0x0000001C    0x00451813    slli x16 x10 4
0x00000020    0x00455893    srli x17 x10 4
0x00000024    0x4016D913    srai x18 x13 1
0x00000028    0x00190913    addi x18 x18 1
0x0000002C    0x00A1A023    sw x10 0(x3)
0x00000030    0x00B1A023    sw x11 0(x3)
0x00000034    0x00C1A023    sw x12 0(x3)
0x00000038    0x00D1A023    sw x13 0(x3)
0x0000003C    0x00E1A023    sw x14 0(x3)
0x00000040    0x00F1A023    sw x15 0(x3)
0x00000044    0x0101A023    sw x16 0(x3)
0x00000048    0x0111A023    sw x17 0(x3)
0x0000004C    0x0121A023    sw x18 0(x3)
```

上板验证现象：



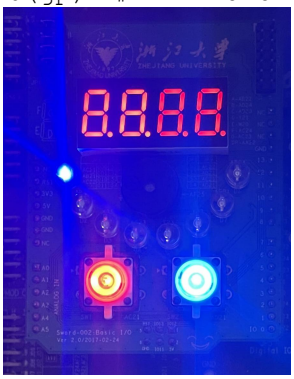
Wb_pc = 24，插入两个 bubble，Id_pc = 28



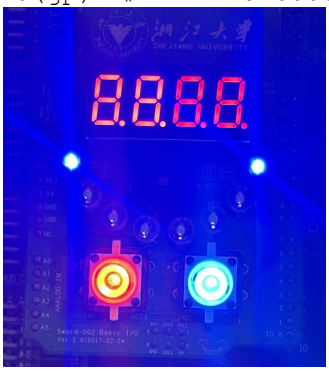
无竞争，充满流水线。Id 寄存器显示 a0~a7 及 s2 的值与预期一致

```
sw   a0, 0(gp)  # LED = 8'b1111_1111
```

```
sw   a1, 0(gp)  # LED = 8'b0000_0000
```



```
sw   a2, 0(gp)  # LED = 8'b0000_0001
```



```
sw   s2, 0(gp) # LED = 8'b1000_0001
```

8.2 测试 R Type 以及数据竞争

R Type 的测试程序如下。除了测试各指令是否正确以外，还着重测试了各种数据竞争的情况。包括 alu + sw，stall 2 拍；alu + other + sw，stall 1 拍；还有 alu + alu 的 stall 情况。

所有 stall 的部分均以红色字体标出。

最后同样是用 LED 灯显示各个寄存器的后 8 位的值。

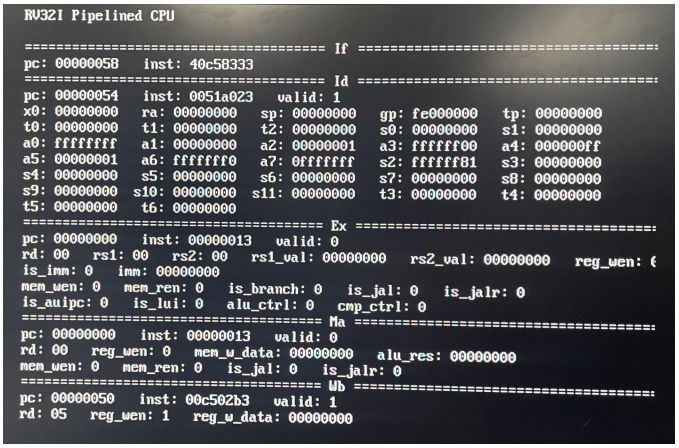
```
# R Type
# test alu + sw stall 2
# test alu + other + sw, stall 1
# test alu + alu, stall 2
    add     t0, a0, a2          # t0 = 0
    # nop
    # nop
    sw      t0, 0(gp)           # LED = 8'b0000_0000, alu + sw, stall 2
```

```
(insert nop, nop)
       sub     t1, a1, a2          # t1 = 0xFFFF_FFFF
       sll     t2, a2, a2          # t2 = 0x0000_0002
       slt     t3, a7, s2          # t3 = 0
       # nop
       sw      t2, 0(gp)          # LED = 8'b0001_0000, alu + other + sw,
stall 1 (insert nop)
       sltu    t4, a7, s2          # t4 = 1
       srl     t5, a4, a2          # t5 = 0x0000_007F
       sra     t6, a0, a2          # t6 = 0xFFFF_FFFF
       # nop
       # nop
       and     s2, t6, t2          # s2 = 0x0000_0002 , alu + alu, stall 2
       sw      t0, 0(gp)          # LED = 8'b0000_0000
       sw      t1, 0(gp)          # LED = 8'b1111_1111
       sw      t2, 0(gp)          # LED = 8'b0000_0010
       sw      t3, 0(gp)          # LED = 8'b0000_0000
       sw      t4, 0(gp)          # LED = 8'b0000_0001
       sw      t5, 0(gp)          # LED = 8'b0111_1111
       sw      t6, 0(gp)          # LED = 8'b1111_1111
       sw      s2, 0(gp)          # LED = 8'b0000_0010
```

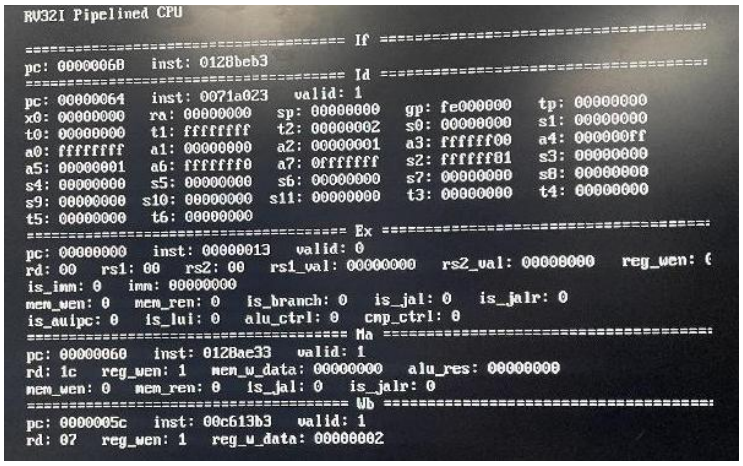
PC、inst 和指令对应如下：

```
0x00000050    0x00C502B3    add x5 x10 x12
0x00000054    0x0051A023    sw x5 0(x3)
0x00000058    0x40C58333    sub x6 x11 x12
0x0000005C    0x00C613B3    sll x7 x12 x12
0x00000060    0x0128AE33    slt x28 x17 x18
0x00000064    0x0071A023    sw x7 0(x3)
0x00000068    0x0128BEB3    sltu x29 x17 x18
0x0000006C    0x00C75F33    srl x30 x14 x12
0x00000070    0x40C55FB3    sra x31 x10 x12
0x00000074    0x007FF933    and x18 x31 x7
0x00000078    0x0051A023    sw x5 0(x3)
0x0000007C    0x0061A023    sw x6 0(x3)
0x00000080    0x0071A023    sw x7 0(x3)
0x00000084    0x01C1A023    sw x28 0(x3)
0x00000088    0x01D1A023    sw x29 0(x3)
0x0000008C    0x01E1A023    sw x30 0(x3)
0x00000090    0x01F1A023    sw x31 0(x3)
0x00000094    0x0121A023    sw x18 0(x3)
```

上板验证现象（主要表现 alu + sw 的竞争，为了节约篇幅省略 LED 的变化）：



alu + sw stall 2 拍



alu + other + sw，stall 1 拍

8.3 测试 lw、sw 以及数据竞争

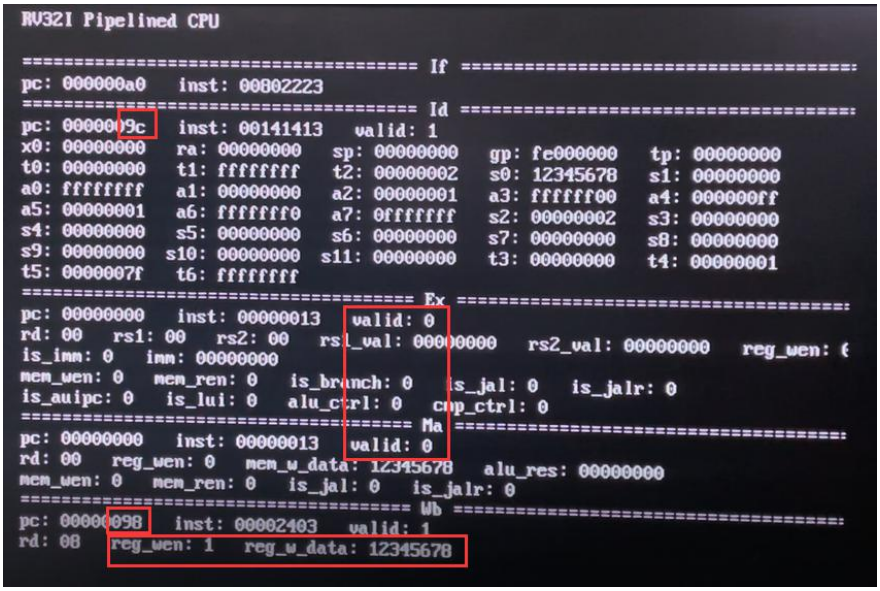
lw、sw 的测试程序如下。除了测试各指令是否正确以外，还着重测试了各种数据竞争的情况。包括 lw + alu，stall 2 拍；lw + other + sw，stall 1 拍。所有 stall 的部分均以红色字体标出。

```
# test LW and SW
    lw      s0, 0(zero)     # s0 = 0x1234_5678
    # nop
    # nop
    slli    s0, s0, 1       # s0 = 0x2468_acf0
    # nop
    # nop
    sw      s0, 4(zero)     # (* no GPRs modified *)
    lw      a0, 4(zero)     # a0 = 0x2468_acf0
    addi    t0, zero, 1
    # nop
    sw      a0, 0(gp)       # LED = 8'b1111_0000
```

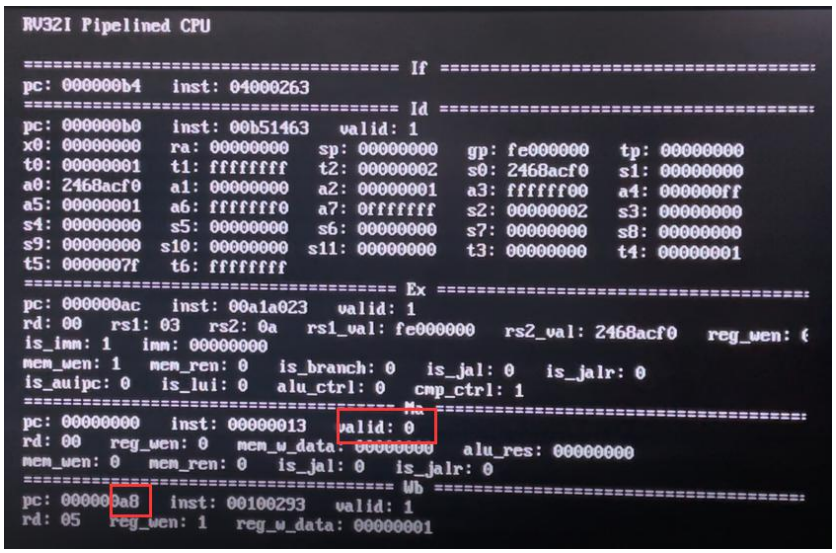
PC、inst 和指令对应如下：

```
0x00000098    0x00002403    lw x8 0(x0)
0x0000009C    0x00141413    slli x8 x8 1
0x000000A0    0x00802223    sw x8 4(x0)
0x000000A4    0x00402503    lw x10 4(x0)
0x000000A8    0x00100293    addi x5 x0 1
0x000000AC    0x00A1A023    sw x10 0(x3)
```

上板验证现象：




```
lw      s0, 0(zero)     # s0 = 0x1234_5678
# nop
# nop
slli    s0, s0, 1       # s0 = 0x2468_acf0
```




```
lw      a0, 4(zero)     # a0 = 0x2468_acf0
addi    t0, zero, 1
# nop
sw      a0, 0(gp)       # LED = 8'b1111_0000
```

8.4 测试 branch 以及控制竞争

branch 的部分测试程序如下。由于 branch 指令包含 beq, bne, bge, blt 等，大同小异，所以仅以 beq 和 bne 的跳转、不跳转情况为示例。若跳转，在本流水线 CPU 中要 stall 3 拍。

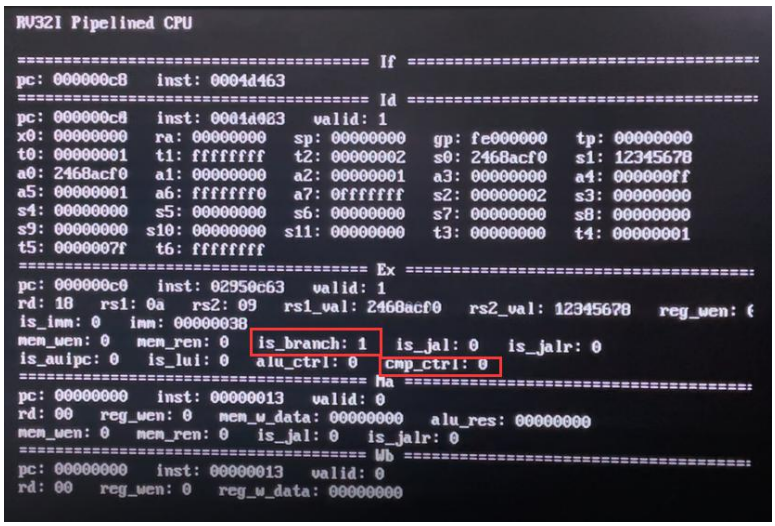
```
# BRANCH

bne_target:
    addi    a3, zero, 0     # a3 = 0x0
    lw      s1, 0(zero)     # s1 = 0x1234_5678
    beq     a0, s1, end_b   # 不跳转，继续执行 LED
    sw      a3, 0(gp)       # LED = 8'b0000_0000
```

```
    bge     s1, zero, bge_target # 跳转至 bge_target
    # nop
    # nop
    # nop
    beq     zero, zero, end_b   # 不执行

bge_target:
```

PC、inst 和指令对应如下：

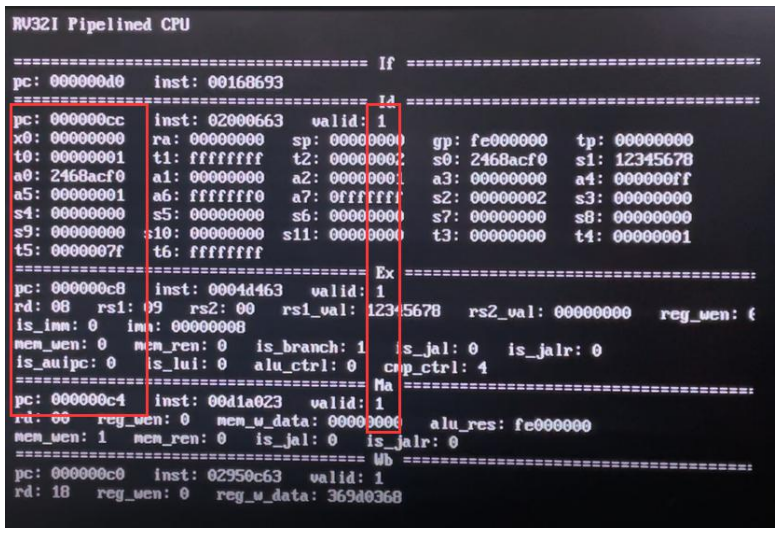
```
0x000000B8     0x00000693     addi x13 x0 0
0x000000BC     0x00002483     lw x9 0(x0)
0x000000C0     0x02950C63     beq x10 x9 56
0x000000C4     0x00D1A023     sw x13 0(x3)
0x000000C8     0x0004D463     bge x9 x0 8
0x000000CC     0x02000663     beq x0 x0 44
0x000000D0     0x00168693     addi x13 x13 1
```

上板验证现象：



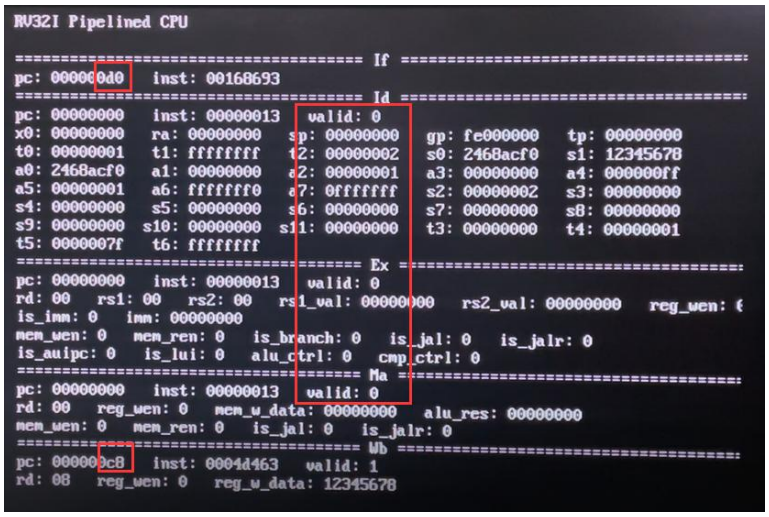

```
beq     a0, s1, end_b   # 不跳转，继续执行 LED
```

处于 EX 级，检测到不跳转




```
beq     a0, s1, end_b   # 不跳转，继续执行 LED
```

不跳转，继续执行




```
bge     s1, zero, bge_target # pc = 0xc8, 跳转至 bge_target
    # nop
    # nop
    # nop
    beq     zero, zero, end_b   # 不执行
bge_target: #pc = 0xd0
```

8.5 测试 U Type

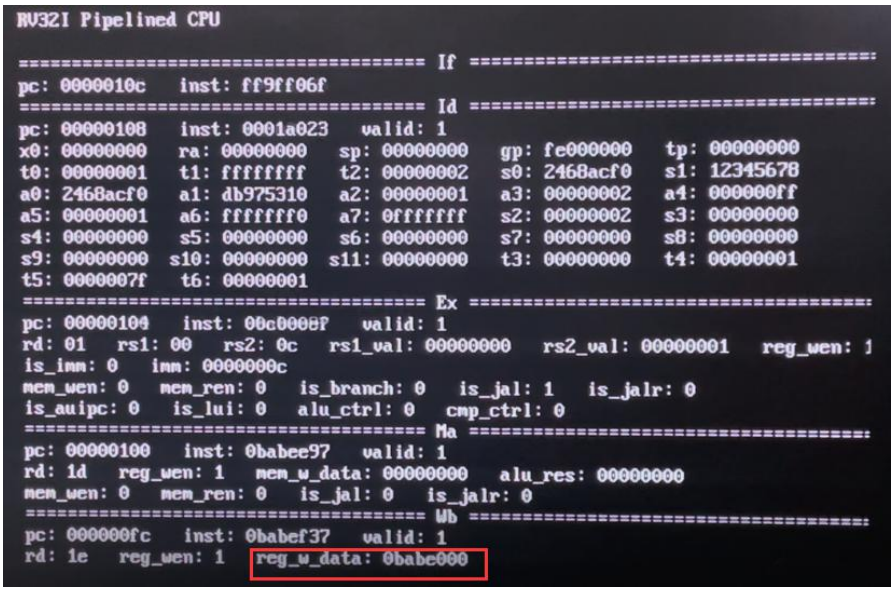
U Type 测试较为简单，代码以及理论结果如下：

```
lui     t5, 0xbabe      # t5 = 0x0babe000
auipc   t4, 0xbabe      # t4 = 0x0babe100
```

PC、inst 和指令对应如下：

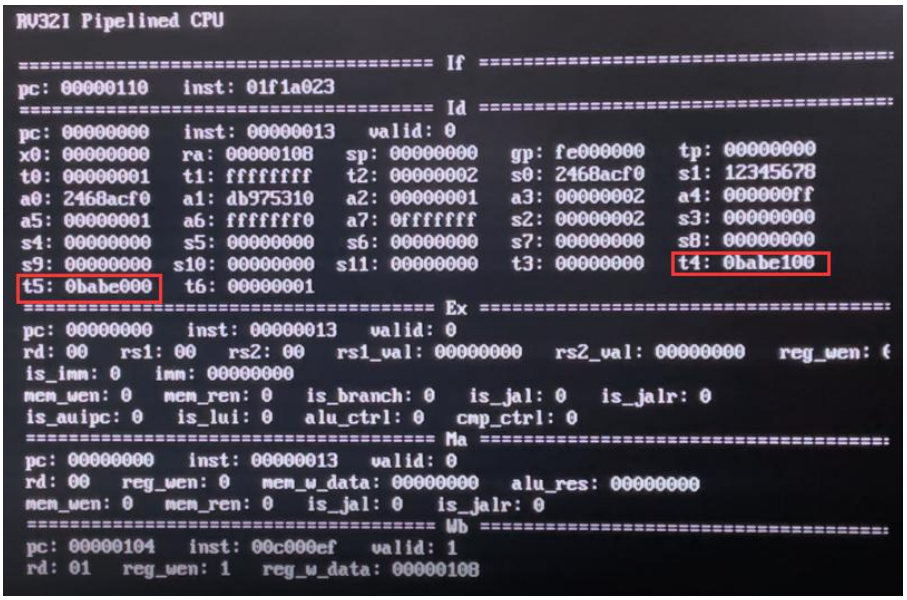
```
0x000000FC     0x0BABEF37     lui x30 47806
0x00000100     0x0BABEE97     auipc x29 47806
```

上板验证现象：




```
lui     t5, 0xbabe      # t5 = 0x0babe000
```

处于 wb 阶段，可见 reg_wen = 1, reg_w_data = 0x0babe000



pc = 0xfc, pc = 0x100 均已流走，t4, t5 发生对应变化

8.6 测试 jal 和 jalr 以及控制竞争

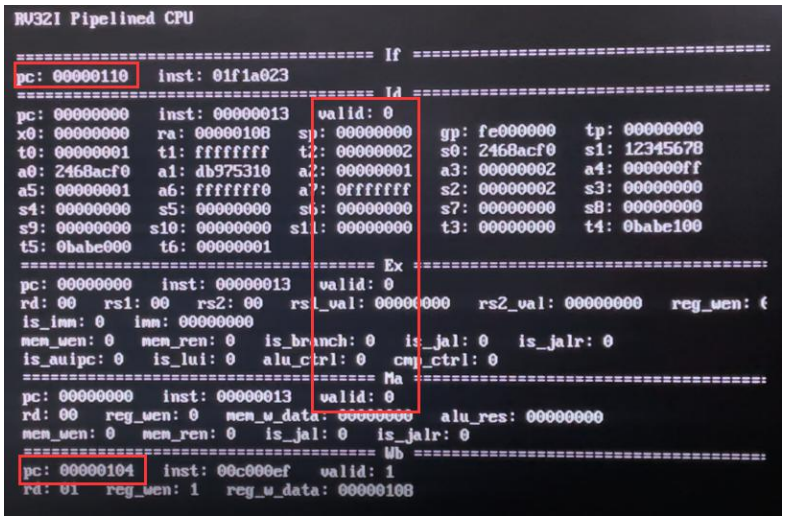
在测试程序结尾，使用 jal 和 jalr 写出死循环，如下：

```
end:
    jal     x1, func        # 无条件跳转至 func
    # nop
    # nop
    # nop
    sw      zero, 0(gp)     # LED = 8'b0000_0000
    jal     zero, end
    # nop
    # nop
    # nop
    # 跳转

func:
    # nop
    # nop
    # nop
    sw      t6, 0(gp)       # LED = 8'b0000_0001
    jalr    zero, x1, 0     # 无条件跳转 sw zero, 0(gp)
```

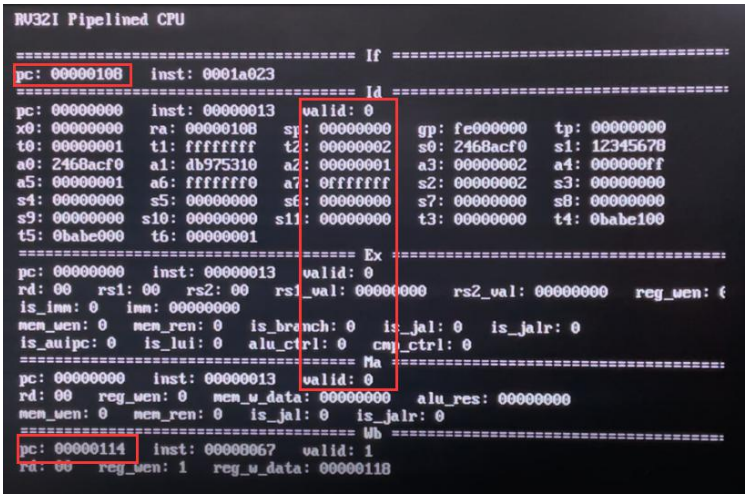
PC、inst 和指令对应如下：

```
0x00000104    0x00C000EF    jal x1 12
0x00000108    0x0001A023    sw x0 0(x3)
0x0000010C    0xFF9FF06F    jal x0 -8
0x00000110    0x01F1A023    sw x31 0(x3)
0x00000114    0x00008067    jalr x0 x1 0
```

上板验证现象：




```
jal       x1, func        # 无条件跳转至 func, pc = 0x110
```




```
jalr    zero, x1, 0     # 无条件跳转 sw zero, 0(gp)
```

综上可见，上板验证现象与预期一致，流水线 CPU 正常工作。

三、实验心得

这一部分主要叙述我做 Lab5 时遇到的 Bug 或困难。

问题一：定义变量时位数错误，导致仿真验证时该变量显示为 XXXXXXXX。

在定义 alu_ctrl 时，忘记其为 4 位，定义成了 alu_ctrl[31:0]，导致无法正常使用 ALU。

问题二：pc 的修改逻辑。

起初 pc 的修改写作：if (~stall) pc <= pc_branch; 但是发现有 bug：发生控制竞争时，MEM 中计算出了 pc_branch，但是无法在 WB 时写回给 pc，导致 WB 和欲跳转的指令的 IF 无法重叠，这与最初的设计本意有所矛盾。修改成如下形式即可解决问题：

```
if (~data_hazard) pc <= pc_branch; // 如果发生 data_hazard 则保持不变，control_hazard 可以改变
```

问题三：IfId_valid 的赋值。

起初写作：IfId_valid <= ~stall; 但数据竞争不会清空 ID 级，只有控制竞争会。修改成 IfId_valid <= ~control_hazard;即可解决问题。

实验心得

总的来说，在设计简单流水线 CPU 时，我没有遭遇很大困难，也没有遇到多少 bug，令我比较惊讶。因为我预想中流水线 CPU 会比单周期更加困难，但实验中，单周期的异常和中断更加困难一些。在单周期 CPU 的各组件基础上，只需要稍微修改组合和时序逻辑，就可以完成 stall 版本的流水线 CPU。

另外，流水线 CPU 的仿真并不直观，不如上板直接观察方便。所以我没有在本次实验中使用仿真验证的方式。

如果想要优化性能，还需要继续添加 forwarding path 和 predict-taken。可以在未来继续探索尝试。