浙江大学

本科实验报告

课程名称：	计算机组成与设计
姓　　名：	颜晗
学　　院：	计算机科学与技术学院
专　　业：	计算机科学与技术
指导教师：	
报告日期：	年　月　日

浙江大学实验报告

课程名称：计算机组成与设计 实验类型：综合

实验项目名称：无冲突流水线处理器 CPU 设计（全阶段）

学生姓名：颜晗 学号：3200105515 同组学生姓名：

实验地点：紫金港东四 509 室 实验日期：年 月 日

一、操作方法与实验步骤

本次实验我们将完成 cpu 的流水线实现。通过将各阶段器件封装为一个模块，再在各阶段之间添加多位的信号寄存器模块使需要信号随时钟在各阶段进行传递，即可实现一个流水线化的 cpu，当然由于未处理冲突，本次实验的流水线并无什么实际作用，下一次实验将实现一个真正的流水线可处理冲突的 cpu。

各模块组成：

1. 信号寄存器

由于流水线 cpu 各阶段处理的是不同的指令，使用不同的信号，我们需要使信号能在各阶段之间进行传递且互不影响。我们使用触发器来完成该项任务，引入时钟与复位信号，每个时钟上升沿信号来临时，模块输出信号变成了从上一阶段来的输入信号，下一个阶段就可以利用这些信号来完成该指令的阶段任务。复位信号上升沿时，所有信号输出为 0.所有信号寄存器模块的区别仅在信号的不同，原理是统一的。

```
module IF_reg_ID(
    input clk_IFID,
    input rst_IFID,
    input en_IFID,
    input [31:0]PC_in_IFID,
    input [31:0]inst_in_IFID,

    output reg [31:0]PC_out_IFID,
    output reg [31:0]inst_out_IFID
    );

    always @(posedge clk_IFID or posedge rst_IFID) begin
        if(rst_IFID==1)begin
            PC_out_IFID     = 0;
            inst_out_IFID   = 0;
        end
        else if(en_IFID==1) begin
```

```
            PC_out_IFID     = PC_in_IFID ;
            inst_out_IFID   = inst_in_IFID;
        end
    end
endmodule
```

2. IF 取指阶段

IF 阶段 cpu 根据上一周期的 PC 值计算+4 并接受从 ID 阶段传来的跳转 PC 值和是否跳转信号（PCSrc）来选择下一周期的 PC 值，当时钟上升沿来时，PC 值改变，指令存储器输出的指令进入到 IF/ID 寄存器模块，等待传入其他阶段进行译码与执行。主要器件为选择 PC 的 mux 以及 PC 寄存器。

```
module Pipeline_IF(
    input clk_IF,
    input rst_IF,
    input en_IF,
    input [31:0] PC_in_IF,
    input PCSrc,

    output [31:0] PC_out_IF
    );

    wire [31:0] pc4;
    assign pc4 = PC_out_IF + 32'h4;
    wire [31:0] pc_next;
    assign pc_next = PCSrc ? PC_in_IF : pc4;

    PCReg PC(
        .clk(clk_IF),
        .rst(rst_IF),
        .CE(en_IF),
        .D(pc_next),
        .Q(PC_out_IF)
    );
endmodule
```

3. ID 译码阶段

ID 阶段是整个 cpu 最为重要的阶段，掌握着指令控制信号的译码，寄存器组的读写、立即数的产生还有跳转 PC 的值计算以及是否跳转的判断。

控制信号的译码依旧是利用单周期 cpu 的控制模块，照搬过来进行信号的连接即可，大部分信号都将输出到下面几个阶段帮助指令执行。

寄存器组的读使用当前 ID 阶段的指令译码产生的寄存器地址来读取数据，而写操作将使用来自 WB 阶段的数据、地址以及使能信号，将之前的指令的结果写入寄存器，而当前 ID 阶段的指令的相关信号将输出到下面的几个阶段直到指令执行到 WB 阶段才写回。立即

数发生器使用当前译码的信号产生立即数，随其余数据信号传入下面的阶段使用。

为了减少下一步实验跳转指令相关的 stall 数量，ID 阶段还产生下周期的 PC 地址以及跳转的判断信号。因此直接加入了一个 ALU 模块（可针对 branch 指令的计算专门设计）对 branch 指令的跳转条件进行运算判断。Jal 信号和 branch 信号的跳转地址都是 PC+imm，而 jalr 的跳转地址为 rs1+imm，对两个信号选择后与 PCSrc 信号一同输出到 ID 阶段。

```
 assign ImmtoReg_ID = Inst_in_ID[5];
    assign Rd_addr_out_ID = Inst_in_ID[11:7];
    //寄存器组
    RegFile32 Regs(.clk(clk_ID),
                   .rst(rst_ID),
                   .RegWrite(RegWrite_in_ID),
                   .Rs1_addr(Inst_in_ID[19:15]),
                   .Rs2_addr(Inst_in_ID[24:20]),
                   .Wt_addr(Rd_addr_ID),
                   .Wt_data(Wt_data_ID),
                   .Rs1_data(Rs1_out_ID),
                   .Rs2_data(Rs2_out_ID),
                   ……vga 显示信号忽略
//控制模块
cpu_ctrl ctrl_unit(
        .OPcode(Inst_in_ID[6:2]),
        .Fun3(Inst_in_ID[14:12]),
        .Fun7(Inst_in_ID[30]),
        .ImmSel(ImmSel),
        .ALUSrc_B(ALUSrc_B_ID),
        .MemtoReg(MemtoReg_ID),
        .Jump(Jump_ID),
        .Branch(Branch_ID),
        .RegWrite(RegWrite_out_ID),
        .MemRW(MemRW_ID),
        .ALU_Control(ALU_control_ID)
    );
    //立即数
    ImmGen immgen(
        .ImmSel(ImmSel),
        .inst_field(Inst_in_ID),
        .Imm_out(Imm_out_ID)
    );
//以下为 PC 判断
    ALUT alu_B(
        .A(Rs1_out_ID),
        .B(Rs2_out_ID),
        .operator(ALU_control_ID),
        .res(ALU_out),
```

```
        .zero(zero)
    );
    assign is_branch = |(Branch_ID[5:0]
&{{~ALU_out[0]},ALU_out[0],{~ALU_out[0]},ALU_out[0],{~zero},zero});
    assign PC_imm = PC_in_ID + Imm_out_ID;
    assign PC_jalr = Rs1_out_ID + Imm_out_ID;
    assign PC_cal_ID = Jump_ID[1] ? PC_jalr :PC_imm;
    assign PCSrc_ID = |{is_branch,Jump_ID[1:0]};
```

4. Ex 计算阶段

该阶段主要对 ID 阶段传过来的数据（Rs1，Rs2，imm）进行计算，另外，作为 jal 和 jalr 写回寄存器的一个来源，还需要计算 PC+4 的值传下去。

```
    assign PC4_out_EX = PC_in_EX + 4;
    wire [31:0] ALU_B;
    assign ALU_B = ALUSrc_B_in_EX ? Imm_in_EX : Rs2_in_EX ;
    ALUT alut(
        .A(Rs1_in_EX),
        .B(ALU_B),
        .operator(ALU_control_in_EX),
        .res(ALU_out_EX),
        .zero(zero_out_EX)
    );
    assign Rs2_out_EX = Rs2_in_EX;
```

5. Mem 内存取数据写数据阶段

以 ALU 计算的结果作为地址，进入总线后对应的地址范围会传进 RAM，如果是 load 指令，我们要根据 fun3 对传入的数据进行截取和扩展（零扩展和符号扩展），再传入 WB 阶段；而对于 Store 指令，由于无法更改 Top 的总线模块，我们将不再重新引入 cpu 处理后再次输出而是在 Top 模块对将输出到 ROM 的数据进行截取、移位处理，使其适应 sb、sh 指令的需求。

6. WB 寄存器组写回阶段

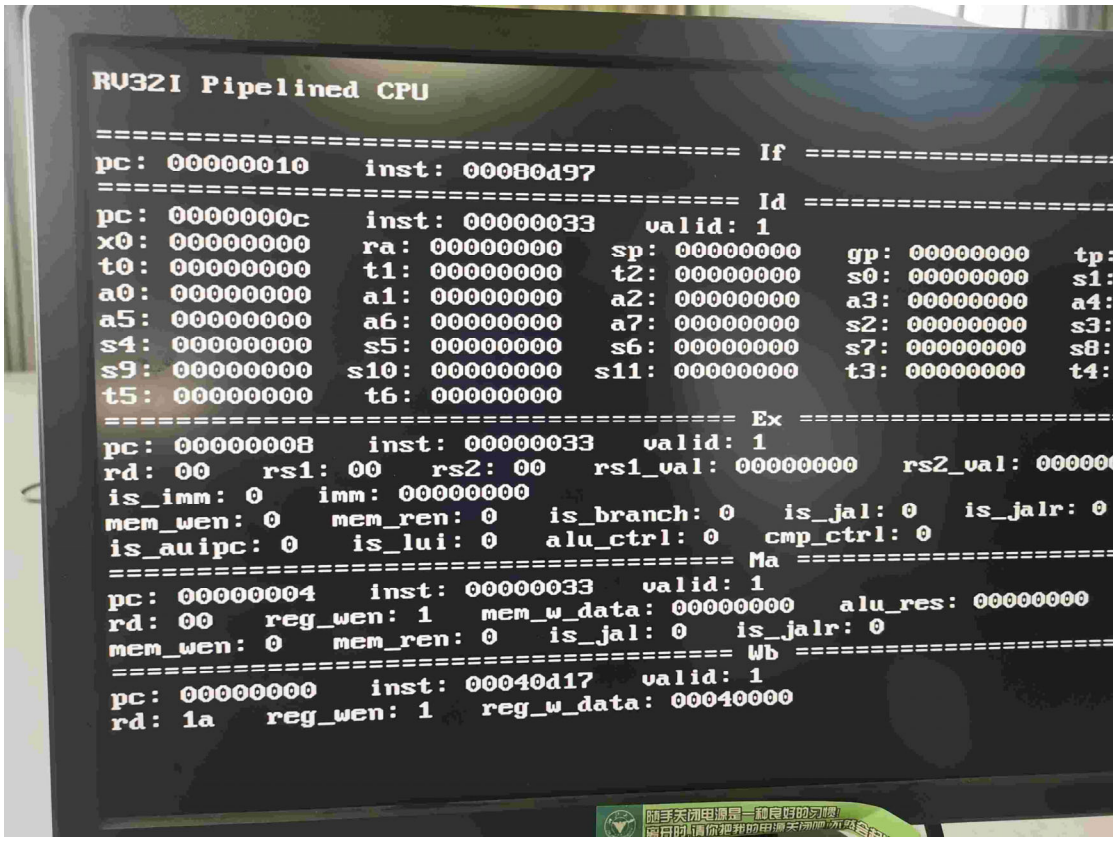
本质即几个 mux 模块，从 PC+4（jalr，jal），PC+imm（U 型，auipc），RAM_out（I 型），ALU_res（R 型，I 型部分），imm（U 型，lui）中选出写回寄存器的数据，再随目的寄存器和写使能信号一起传回 ID 阶段。

```
assign Data_reg_WB=Immtoreg_out_MemWB ?Imm_out_MemWB:PCcal_out_MemWB;
    assign Data_out_WB = MemtoReg_out_MemWB[1]?
             (MemtoReg_out_MemWB[0]?Data_reg_WB : PC4_out_MemWB)
         :(MemtoReg_out_MemWB[0]?DMem_data_out_MemWB : ALU_out_MemWB);
```

二、实验结果与分析

1. 下板验证

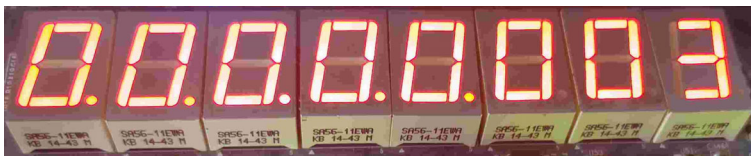
Vga 显示，可以看出两指令之间插入了三条无意义指令，因目前无处理冲突的能力。



开机画面

矩形动画

计数器



学号

2. 仿真验证

无了……

三、讨论与心得

本次实验完成了无冲突处理的流水线 cpu，由于不同阶段数据信号同时存在，还将部分 vga 显示的信号单独赋了值，因此主要是信号不匹配导致的 bug，vga 显示正常后根据出错指令的数据通路猜测并检查错误信号可以发现诸如使用的信号未声明，位数不匹配、信号接错地方等错误，包括后面发现自己一个 Rd 地址的线路本应是[4:0]声明为了[31:0]，但也能运行……可能还有一些不影响运行的 bug。

浙江大学实验报告

课程名称：计算机组成与设计 实验类型：综合

实验项目名称：带冲突处理的流水线处理器 CPU 设计

学生姓名：颜晗 学号：3200105515 同组学生姓名：

实验地点：紫金港东四 509 室 实验日期：年 月 日

一、操作方法与实验步骤

本次实验我们将实现流水线 cpu 的冲突处理，具体为添加 stall 处理模块与 forwarding 模块，并从原先的模块中再引出部分信号用于判断。

1. Forwarding 模块

使用 forwarding 可以简化多种指令（R 型与 R 型、R 型与 S 型 I 型……）之间 stall 的插入并且提升 cpu 的效率，本身也足够简单。

Rd 地址和写使能信号在前一个实验已经引出，而要实现 forwording 模块，我们还要从 ID 中引出 Rs1 和 Rs2 的地址以及判断这两个地址是否有意义（是否被使用）的信号传入 Ex 模块。不同指令格式对于源寄存器的使用是固定的，因此我们只需要在 ID 中判断 opcode 即可判断源寄存器地址是否有意义。

```
assign Rs1_used = (OPcode == 5'b01100) | (OPcode == 5'b00100) |
( OPcode == 5'b00000) |(OPcode == 5'b01000)  | (OPcode == 5'b11000) |
(OPcode == 5'b11001);
//R, Imm, ld,  sd,  branch, jalr
assign Rs2_used = (OPcode == 5'b01100) | (OPcode == 5'b01000) |
(OPcode == 5'b11000) ;
//R, sd, branch
assign Rs1_addr = Inst_in_ID[19:15];
assign Rs2_addr = Inst_in_ID[24:20];
```

Forwarding 模块的逻辑很简单，对于 ALU 的两个源数据，可能因为前面的数据还未执行结束写回到寄存器就已经要使用了（判断 used 以及 Ex 的源寄存器是否等于下面阶段的目的寄存器），那么我们将后面模块的对应数据直接引进作为一个输入并使用对应的信号判断是否使用该输入即可，forwarding 的数据可能来自 Mem 或 WB 阶段，除需要从存储器取出的数据外，其余写回寄存器的数据源（PC+4、PC+imm、imm、ALU_res）都可以从 EX 阶段进行 forwarding，而 WB 阶段新增了一个从存储器取出的数据作为源数据。

```
module Forwarding(
    input [4:0] Rs1_addr_IDEX,
    input [4:0] Rs2_addr_IDEX,
    input Rs1_used,
    input Rs2_used,
    input [4:0] Rd_addr_EXMem,
    input RegWrite_ExMem,
    input [4:0] Rd_addr_MemWB,
    input RegWrite_MemWB,

    output reg [1:0] ForwardA,
    output reg [1:0] ForwardB

    );

    always @* begin
        if(Rs1_used & RegWrite_ExMem &(Rs1_addr_IDEX
==Rd_addr_EXMem))begin
            ForwardA = 2'b01;//从 Mem 阶段前递
        end
        else if(Rs1_used & RegWrite_ExMem &(Rs1_addr_IDEX
==Rd_addr_MemWB)) begin
            ForwardA = 2'b10;//从 WB 阶段前递
        end
        else begin
            ForwardA = 2'b00;//无前递
        end

        if(Rs2_used & RegWrite_ExMem &(Rs2_addr_IDEX
==Rd_addr_EXMem))begin
            ForwardB = 2'b01;//从 Mem 阶段前递
        end
        else if(Rs2_used & RegWrite_ExMem &(Rs2_addr_IDEX
==Rd_addr_MemWB)) begin
            ForwardB = 2'b10;//从 WB 阶段前递
        end
        else begin
            ForwardB = 2'b00;//无前递
        end
    end

endmodule
```

2. Stall 控制

流水线运行时会遇到难以避免的冲突，此时我们不得不停下部分流水线，使前面的指令执行完后才能执行后面的指令。由于提前了跳转判断并添加了 forwarding，需要插入 stall 情况只有三种：

2.1 branch 指令条件跳转，需要两个源寄存器的值进行比较，但是可能前面的指令未执行完而且正好修改了需要的寄存器，此时我们需要等到前面的指令都执行完后，寄存器的值已被写回才能继续执行 branch 指令。而这种情况我们连停三拍。

2.2 跳转，这种情况下我们需要重新更新 IF 阶段的 PC 值及其所取出的指令，因此停一拍，并在 IF/ID 寄存器中插入 nop，等待更新后继续执行。而不跳转的话，PC 值依旧是 PC+4，就不需要做任何操作了。

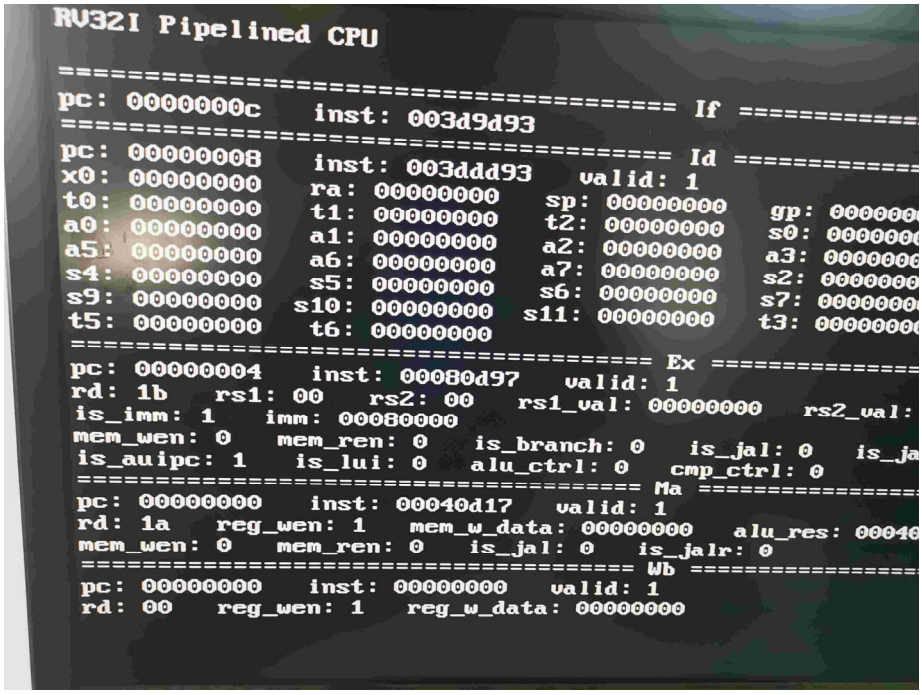
2.3 Load_use，由于取数据需要时间，我们不能在 Mem 阶段将数据前递至 Ex 阶段，因此如果 load 指令后一条指令需要对应的数据，我们就需要停一拍。在 load 指令执行至 Ex 阶段时，下一条指令也在 ID 译码结束，判断源寄存器和目的寄存器即可判断是否需要 stall。

```
 always @* begin
        if(OPcode_ID == 5'b11000 | OPcode_ID == 5'b11011 | OPcode_ID
== 5'b11001) begin
            if( RegWrite_IDEX & Rs1_used & (Rs1_addr_ID ==
Rd_addr_IDEX) & Rs1_addr_ID != 0
              | RegWrite_IDEX & Rs2_used & (Rs2_addr_ID ==
Rd_addr_IDEX) & Rs2_addr_ID != 0
              | RegWrite_ExMem & Rs1_used & (Rs1_addr_ID ==
Rd_addr_EXMem) & Rs1_addr_ID != 0
              | RegWrite_ExMem & Rs2_used & (Rs2_addr_ID ==
Rd_addr_EXMem) & Rs2_addr_ID != 0
              | RegWrite_MemWB & Rs1_used & (Rs1_addr_ID ==
Rd_addr_MemWB) & Rs1_addr_ID != 0
              | RegWrite_MemWB & Rs2_used & (Rs2_addr_ID ==
Rd_addr_MemWB) & Rs2_addr_ID != 0    ) begin
                en_IF <= 0;
                en_IFID <= 0;
                NOP_IFID <= 0;
                NOP_IDEX <=1;
                //跳转指令对前面有数据依赖，没用前递，要等 WB 完成
                //if 停止取指令，ifid，id 停止运行，idex 使用 NOP
            end
            else if(PCSrc_ID) begin
                en_IF <= 1;
                en_IFID <= 1;
                NOP_IFID <= 1;
                NOP_IDEX <=0;
                //确定跳转，
                //if 重新取指令，ifid 向前插入一个 NOP 运行下去
            end
```

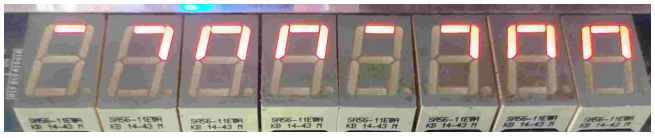
```
            else begin
                en_IF <= 1;
                en_IFID <= 1;
                NOP_IFID <= 0;
                NOP_IDEX <=0;
                //普通情况，执行下去
            end
        end
        else begin //判断 load_use
            if( (OPcode_IDEX == 5'b00000) & Rs1_addr_ID != 0
&Rs2_addr_ID != 0
              & ((Rs1_used & Rs1_addr_ID == Rd_addr_IDEX)
              | (Rs2_used & Rs2_addr_ID == Rd_addr_IDEX)) ) begin
                en_IF <= 0;
                en_IFID <= 0;
                NOP_IFID <= 0;
                NOP_IDEX <= 1;
                //load_use
                // if ifid 停摆，相当于 id 阶段下一周期依旧执行前一条指令
                // ex 插入 nop
              end
              else begin
                en_IF <= 1;
                en_IFID <= 1;
                NOP_IDEX <= 0;
                NOP_IFID <= 0;
                //普通情况
              end
        end

    end
```

二、实验结果与分析

1. 下板验证
Vga 显示，不需要人工插入气泡。



开机动画



矩形动画

计数器


学号


2. 仿真验证

无了……

三、讨论与心得

本次实验我们完成了流水线的冲突处理，由于指令相比课堂更加全面且，我们需要自行判断 forwarding 与 stall 所需的数据和信号。注意在设计添加完两个模块后还需要加入 CPU 模块中。此次遇到的问题主要有：

1. 由于复位时各阶段间寄存器的信号都是 0，导致开始时即一直产生 load_use 的 stall 插入，cpu 就不动了。因此后续在 stall 控制模块中添加了寄存器地址非 0 的条件判断，由于 0 号寄存器无法修改，也不会影响正常指令的条件判断。
2. 由于偷懒，前置的 cpu 设计中的几乎所有 if-else 以及 case 语句都没有 else 或 default 情况，导致了 latch 的产生，但是之前一直没有出现过问题，就没有解决。但是此次测试助教代码时 sb 指令部分一直有问题，甚至不同的运行方式（按钮单步运行、时钟）会导致结果不同。尝试补上所有的 else 以及 default 情况后结果恢复正常……只能说硬件确实神奇。