实验报告成绩：	成绩评定日期：

2022 ~ 2023 学年秋季学期

《计算机系统》必修课

课程实验报告

班级：人工智能 2302

组长：蔡嘉伟

班级：人工智能 2301

组员：徐本钰

报告日期：2025.12.27

目录

任务量各 50%

GitHub 地址 https://github.com/s0y096/CPU_EXP

1. 总体设计

该 CPU 采用经典的五级流水线架构，包含取指(IF)、译码(ID)、执行(EX)、访存(MEM)和写回(WB)五个阶段。设计采用哈佛架构，指令存储器与数据存储器分离。流水线支持数据前递(forwarding)机制解决数据冒险，通过停顿(stall)机制解决结构冒险。关键特性包括支持 MIPS 指令集的算术逻辑运算、分支跳转、访存指令以及乘除法运算。程序运行环境 windows vivado。

2. 不同流水段之间的连线图

IF 阶段(取指) ---> ID 阶段(译码) ---> EX 阶段(执行) ---> MEM 阶段(访存) ---> WB 阶段(写回)

| | | |

指令存储器 寄存器文件 ALU/乘除单元 数据存储器 写回寄存器

| | | |

PC 计数器　　控制信号生成　　前递数据通路　　访存数据通路　　调试输出

各阶段间通过专用总线传递信息：

IF_TO_ID_WD：IF 到 ID 的总线，包含指令地址和使能信号

ID_TO_EX_WD：ID 到 EX 的总线，包含指令、操作数、控制信号

EX_TO_MEM_WD：EX 到 MEM 的总线，包含计算结果、访存控制信号

MEM_TO_WB_WD：MEM 到 WB 的总线，包含写回数据和地址

3. 单个流水段说明

3.1 IF 阶段（取指阶段）

整体功能 IF 阶段负责从指令存储器中取出指令，并计算下一条指令的地址。支持顺序执行和分支跳转两种取指方式。

端口介绍

输入端口：clk(时钟)、rst(复位)、stall(停顿信号)、br_bus(分支跳转总线)

输出端口：if_to_id_bus(到 ID 阶段的总线)、inst_sram_*(指令存储器接口信号)

信号

pc_reg：程序计数器寄存器，保存当前取指地址

ce_reg：指令存储器使能寄存器

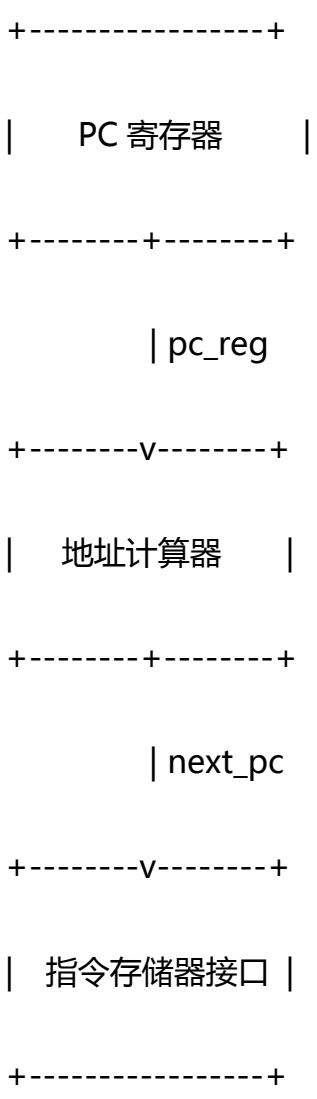
next_pc：下一条指令地址，由分支判断逻辑或顺序地址生成

br_bus :来自 ID 阶段的分支跳转信号 ,包含跳转使能 br_e 和跳转地址 br_addr

功能模块

IF 阶段主要包含程序计数器和地址计算逻辑。程序计数器在时钟上升沿更新，当分支跳转有效时更新为跳转地址，否则顺序加 4。

结构示意图



源码（IF.v）

```
// 程序计数器更新逻辑
always @ (posedge clk) begin
    if (rst) begin
        pc_reg <= 32'hbfbf_fffc;   // 复位地址
    end
    else if (stall[0]==`NoStop) begin   // 如果 IF 阶段不被停顿
        pc_reg <= next_pc;              // 更新 PC
    end
end

// 下一条指令地址计算（选择器示例）
assign next_pc = br_e ? br_addr : pc_reg + 32'h4;
```

这是一个带优先级的二选一选择器，当分支跳转使能 br_e 有效时，选择跳转地址 br_addr，否则选择顺序地址(pc+4)。优先级固定为分支跳转优先。

3.2 ID 阶段（译码阶段）

整体功能 ID 阶段对指令进行译码，读取寄存器文件，生成控制信号，检测数据冒险并实现数据前递，同时进行分支判断。

端口介绍

输入端口：clk、rst、stall、if_to_id_bus、inst_sram_rdata(指令数据)

输出端口：id_to_ex_bus、br_bus、stallreq_for_bru(分支停顿请求)

前递输入：wb_to_rf_bus、ex_to_rf_bus、mem_to_rf_bus

信号介绍

ndata1/ndata2：经过数据前递处理后的操作数 rf_we：寄存器写使能信号

rf_waddr：寄存器写地址 br_e：分支跳转使能信号 br_addr：分支跳转地址

功能模块

1. 指令译码器：将 32 位指令解析为各字段

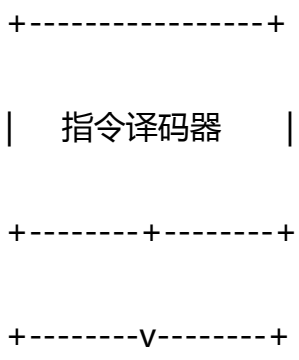
2. 寄存器文件：双端口寄存器堆，支持同时读两个操作数

3. 数据前递逻辑：解决 RAW 数据冒险

4. 分支判断逻辑：计算分支条件并生成跳转信号

5. 控制信号生成：产生 ALU 操作、访存控制等信号

结构示意图



```
        |    寄存器文件      |
        +--------+--------+
        +--------v--------+
        |    数据前递逻辑    |
        +--------+--------+
        +--------v--------+
        | 分支判断与控制   |
        +-----------------+
```

源码解释（ID.v 数据前递部分）

```
// 数据前递选择器（并行选择器示例）
assign ndata1 = (ex_rf_we   && rs == ex_rf_waddr)    ? ex_rf_wdata   :
                (mem_rf_we    &&    rs    ==    mem_rf_waddr)       ?
mem_rf_wdata :
                (wb_rf_we   && rs == wb_rf_waddr)    ? wb_rf_wdata   :
                 rdata1;

assign ndata2 = (ex_rf_we   && rt == ex_rf_waddr)    ? ex_rf_wdata   :
```

(mem_rf_we && rt == mem_rf_waddr) ? mem_rf_wdata :

(wb_rf_we && rt == wb_rf_waddr) ? wb_rf_wdata :

rdata2;

这是三个并行的选择器结构，用于实现数据前递。每个选择器独立判断是否存在数据冒险，优先级为：EX 阶段数据 > MEM 阶段数据 > WB 阶段数据 > 原始寄存器数据。没有采用级联选择器，而是用条件运算符实现，所有判断并行进行。

3.3 EX 阶段（执行阶段）

整体功能说明

EX 阶段执行算术逻辑运算，进行乘除法计算，计算访存地址，生成访存控制信号。

端口介绍

输入端口：clk、rst、stall、id_to_ex_bus、id_load_bus、id_save_bus

输出端口：ex_to_mem_bus、data_sram_*(数据存储器接口)、stallreq_for_ex(执行停顿请求)

信号介绍

alu_src1/alu_src2：ALU 操作数

alu_result：ALU 计算结果

ex_result：最终执行结果（可能是 ALU 结果或乘除法结果）

hi_we/lo_we：HI/LO 寄存器写使能

stallreq_for_div：除法器停顿请求

功能模块说明

1. ALU：算术逻辑单元，支持 12 种运算操作

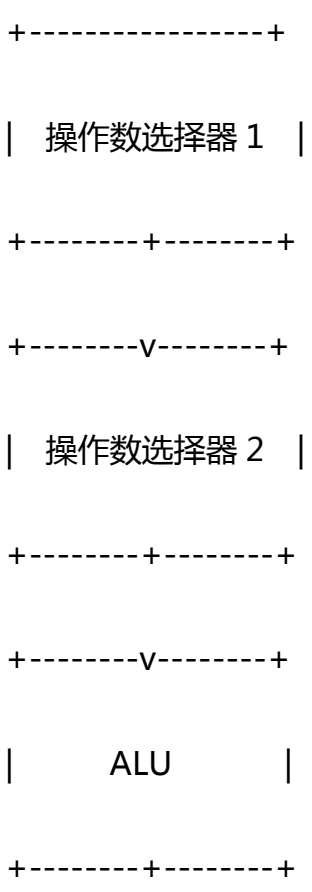
2. 乘法器：32 位有符号/无符号乘法器

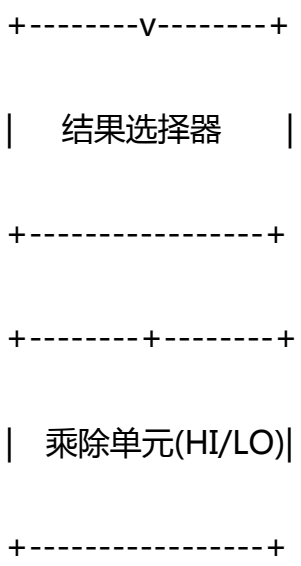
3. 除法器：32 位有符号/无符号除法器（多周期）

4. 操作数选择器：选择 ALU 输入操作数

5. HI/LO 寄存器管理：处理乘除法相关指令

结构示意图





源码（EX.v 操作数选择器）

```
// ALU 操作数选择器 1
assign alu_src1 = sel_alu_src1[1] ? ex_pc :
                  sel_alu_src1[2] ? sa_zero_extend : rf_rdata1;

// ALU 操作数选择器 2
assign alu_src2 = sel_alu_src2[1] ? imm_sign_extend :
                  sel_alu_src2[2] ? 32'd8 :
                  sel_alu_src2[3] ? imm_zero_extend : rf_rdata2;
```

这是两个多路选择器，采用编码控制方式。sel_alu_src1 和 sel_alu_src2 是控制信号编码，每个位对应一种输入源选择。这种设计使用二进制编码而非独热码，通过译码逻辑在 ID 阶段生成控制信号。

3.4 MEM 阶段（访存）

整体功能 MEM 阶段访问数据存储器，完成加载(Load)和存储(Store)操作，将执行结果或加载数据传递给 WB 阶段。

端口

输入端口：clk、rst、stall、ex_to_mem_bus、data_sram_rdata、ex_load_bus

输出端口：mem_to_wb_bus、mem_to_rf_bus（前递数据）

信号

data_ram_en：数据存储器使能 data_ram_wen：数据存储器写使能

sel_rf_res：选择写回数据来源（ALU 结果或访存数据）rf_wdata：最终写回数据

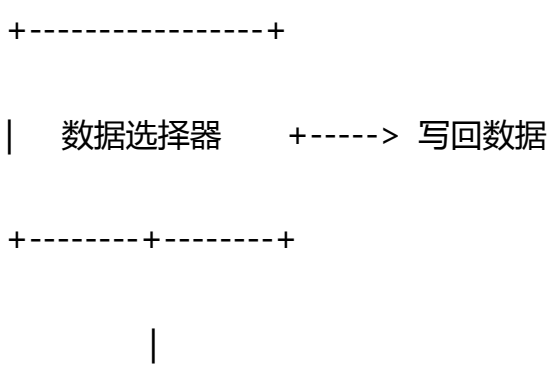
功能模块

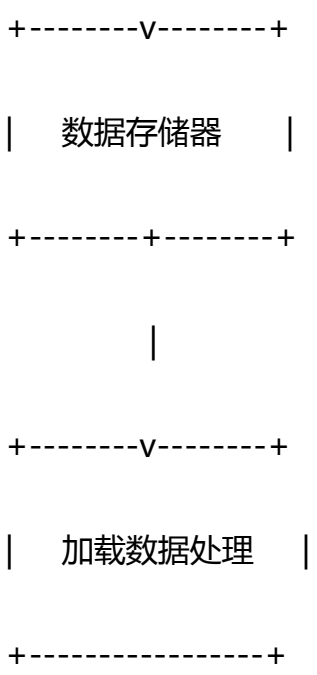
1. 访存接口：连接数据存储器

2. 数据选择器：选择写回数据源

3. 总线暂存：流水线寄存器，暂存阶段间传递的数据

结构示意图





源码解释（MEM.v 数据选择器）

assign rf_wdata = (sel_rf_res & data_ram_en) ? mem_result : ex_result;

这是一个简单的二选一选择器，用于决定写回数据的来源。当指令是加载指令(sel_rf_res=1)且访存使能(data_ram_en=1)时，选择从存储器读取的数据(mem_result)，否则选择执行阶段的计算结果(ex_result)。

3.5 WB 阶段（写回阶段）

整体功能说明

WB 阶段将数据写回寄存器文件，并输出调试信息用于验证和调试。

端口介绍

输入端口：clk、rst、stall、mem_to_wb_bus

输出端口：wb_to_rf_bus、debug_*（调试信号）

信号介绍

wb_pc：写回阶段的指令地址（调试用）rf_we：寄存器写使能

rf_waddr：寄存器写地址 rf_wdata：寄存器写数据

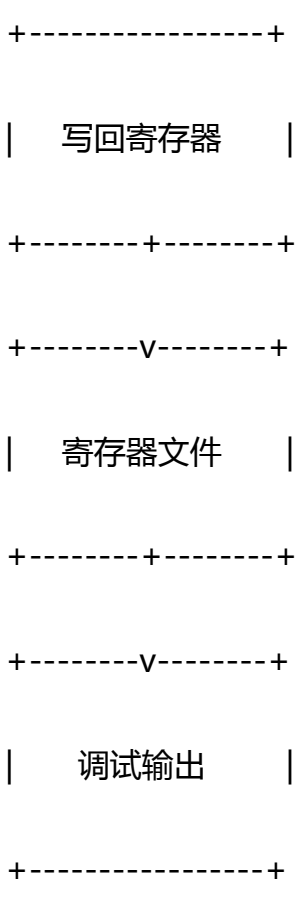
功能模块说明

1. 写回寄存器：暂存写回数据

2. 调试接口：输出处理器状态信息

3. 总线驱动：驱动写回数据到寄存器文件

结构示意图



源码（WB.v）

```
// 写回数据打包
assign wb_to_rf_bus = {
    rf_we,
    rf_waddr,
    rf_wdata
};
// 调试信号生成
assign debug_wb_pc = wb_pc;
assign debug_wb_rf_wen = {4{rf_we}};  // 扩展为 4 位写使能
assign debug_wb_rf_wnum = rf_waddr;
assign debug_wb_rf_wdata = rf_wdata;
```

WB 阶段主要完成数据格式的转换和打包。调试信号将内部信号转换为标准调试接口格式，其中 debug_wb_rf_wen 通过复制操作将 1 位写使能扩展为 4 位，符合字节使能格式。

4. 流水线控制机制

停顿控制（CTRL.v）

```
always @ (*) begin
    if (rst) begin
```

```
        stall = `StallBus'b0;
    end
    else if (stallreq_for_ex) begin    // EX 阶段停顿请求（如除法）
        stall = `StallBus'b001111;      //  停顿 IF,ID,EX 阶段
    end
    else if (stallreq_for_bru) begin //  分支停顿请求
        stall = `StallBus'b000111;      //  停顿 IF,ID,EX 阶段
    end
    else begin
        stall = `StallBus'b0;              //  无停顿
    end
end
```

停顿控制器采用优先级编码，EX 阶段的停顿请求优先级最高（如除法运算），其次是分支停顿请求。停顿信号 stall 是一个位向量，每个位对应一个流水段的停顿控制。

5. 数据前递机制

数据前递通过三级前递路径实现：

1. EX 阶段前递：解决相邻指令的数据依赖

2. MEM 阶段前递：解决间隔一条指令的数据依赖

3. WB 阶段前递：解决间隔两条指令的数据依赖

前递逻辑在 ID 阶段实现，通过比较源寄存器地址和写回寄存器地址，选择最新的数据值。

第一级：EX 阶段前递

来源：EX 阶段刚计算出的结果

解决：RAW（写后读），当前指令在 ID 阶段需要前一条指令的 EX 结果

时机：前一条指令在 EX 阶段，当前指令在 ID 阶段

第二级：MEM 阶段前递

来源：MEM 阶段的数据（可能是 ALU 结果或刚加载的数据）

解决：RAW 冒险，当前指令需要前前条指令的 MEM 结果

时机：前前条指令在 MEM 阶段，当前指令在 ID 阶段

第三级：WB 阶段前递

来源：WB 阶段即将写回的数据

解决：常规的数据传递，优先级最低

时机：数据已计算完成，正在写回

ID.v 中的前递逻辑：

// 三级前递选择器（并行结构）

assign ndata1 = (ex_rf_we && rs == ex_rf_waddr) ? ex_rf_wdata :

//第一级：EX 阶段

(mem_rf_we && rs == mem_rf_waddr) ? mem_rf_wdata： // 第二级：

MEM 阶段

(wb_rf_we && rs == wb_rf_waddr) ? wb_rf_wdata : // 第三级：WB

阶段 rdata1; // 无冒险：直接读取

assign ndata2 = (ex_rf_we && rt == ex_rf_waddr) ? ex_rf_wdata :

(mem_rf_we && rt == mem_rf_waddr) ?

mem_rf_wdata :

(wb_rf_we && rt == wb_rf_waddr) ? wb_rf_wdata :

rdata2;

三级前递的优先级设计

优先级顺序：EX 阶段 > MEM 阶段 > WB 阶段 > 寄存器文件

EX 阶段数据最新：刚刚计算完成，但还未写入寄存器

MEM 阶段数据次新：已经过一个周期，可能已从存储器加载

WB 阶段数据最旧：即将写回，但还在流水线中

前递条件：

/1. 前一条指令要写寄存器（ex_rf_we=1）

2. 当前指令的源寄存器（rs）等于前一条指令的目的寄存器（ex_rf_waddr）

3. 如果条件满足，使用前递数据（ex_rf_wdata），否则继续检查下一级

1：连续算术指令（EX 前递）

周期 1：ADD R1, R2, R3　　EX 阶段计算 R2+R3，结果在 EX_out

周期 2：SUB R4, R1, R5　　ID 阶段需要 R1，从 EX 阶段前递获取

使用第一级前递（EX 阶段）

2：加载指令后使用（MEM 前递）

周期 1：LW R1, 0(R2)　　MEM 阶段从存储器加载数据到 R1

周期 2：ADD R3, R1, R4　　ID 阶段需要 R1，从 MEM 阶段前递获取

使用第二级前递（MEM 阶段）

3：间隔两条指令（WB 前递）

周期 1：ADD R1, R2, R3　　WB 阶段准备写回 R1

周期 2：NOP　　空操作

周期 3：OR R5, R1, R6　　ID 阶段需要 R1，从 WB 阶段前递获取

开始

五级流水，mycpu_core.v 实现布线相关,regfile.v 32 位寄存器相关，module ID(

```
    input wire clk,
```

```
    input wire rst,
    // input wire flush,
    input wire [`StallBus-1:0] stall,

    output wire stallreq,

    input wire [`IF_TO_ID_WD-1:0] if_to_id_bus,

    input wire [31:0] inst_sram_rdata,

    input wire ex_id, // EX阶段传回信号 ,EX阶段传回上一个指令的sel_rf_res信号

    input wire [`WB_TO_RF_WD-1:0] wb_to_rf_bus,

    input wire [`EX_TO_RF_WD-1:0] ex_to_rf_bus, // 从 EX 阶段到 regfile 的数据总线
```

input wire [`MEM_TO_RF_WD-1:0] mem_to_rf_bus, // 从 MEM 阶段到 regfile 的数据总线

output wire stallreq_for_bru, // 为跳转分支指令请求停顿的信号

output wire [`LoadBus-1:0] id_load_bus, // ID 阶段传递的 Load 型指令信号

output wire [`SaveBus-1:0] id_save_bus, // ID 阶段传递的 Save 型指令信号

output wire [`ID_HI_LO_WD-1:0] id_hi_lo_bus, //ID 段传递的 hi_lo 寄存器指令信号

input wire [`EX_HI_LO_WD-1:0] ex_hi_lo_bus, // EX 段传递的 hi_lo 寄存器信号

output wire [`ID_TO_EX_WD-1:0] id_to_ex_bus,

output wire [`BR_WD-1:0] br_bus

);

Id.v input 接收气泡，output 输出数据，if _to_id_bus 表 if 段传给 id 段

```
always @ (posedge clk) begin
        if (rst) begin
            if_to_id_bus_r <= `IF_TO_ID_WD'b0;
            flag <= 1'b0;
            buf_inst <= 32'b0;
        end
        // else if (flush) begin
        //        ic_to_id_bus <= `IC_TO_ID_WD'b0;
        // end
        else if (stall[1]==`Stop && stall[2]==`NoStop) begin
            if_to_id_bus_r <= `IF_TO_ID_WD'b0;
            flag <= 1'b0;
        end
        else if (stall[1]==`NoStop) begin
            if_to_id_bus_r <= if_to_id_bus;
            flag <= 1'b0;
        end
```

```
        else if (stall[1]==`Stop && stall[2]==`Stop && flag==1'b0) begin
            flag <= 1'b1;
            buf_inst <= inst_sram_rdata; //停顿，缓存当前指令
        end
    end
```

Always 段在延迟槽中，需在下一周期中打开，本周期不执行

```
assign id_to_ex_bus = {
    id_pc,          // 158:127
    inst,           // 126:95
    alu_op,         // 94:83
    sel_alu_src1,   // 82:80
    sel_alu_src2,   // 79:76
    data_ram_en,    // 75
    data_ram_wen,   // 74:71
    rf_we,          // 70
    rf_waddr,       // 69:65
    sel_rf_res,     // 64
    rdata1,         // 63:32
    rdata2          // 31:0
};
```

打包是有顺序的，所以解包时需要一一对应，名称和长度可以将许多数据打包通过一根线发送 Wire 临时变量

```
regfile u_regfile(
    .clk    (clk    ),
    .raddr1 (rs ),
    .rdata1 (rdata1 ),
```

```
        .raddr2 (rt ),
        .rdata2 (rdata2 ),
        .we         (wb_rf_we         ),
        .waddr    (wb_rf_waddr    ),
        .wdata    (wb_rf_wdata    )
);括号中为寄存器名称，表对哪个寄存器进行操作，
  assign inst_ori         = op_d[6'b00_1101];
        assign inst_lui         = op_d[6'b00_1111];
        assign inst_addiu      = op_d[6'b00_1001];
表判断是否相等后面数字可在 A03 文件中找到，6 表 6 位
  assign opcode = inst[31:26];
        assign rs = inst[25:21];
        assign rt = inst[20:16];
        assign rd = inst[15:11];
        assign sa = inst[10:6];
        assign func = inst[5:0];
        assign imm = inst[15:0];
        assign instr_index = inst[25:0];
```

```
assign code = inst[25:6];

assign base = inst[25:21];

assign offset = inst[15:0];

assign sel = inst[2:0];
```

将 32 位码进行拆解，每几位对应一个意思，以便于对不同的部分进行操作利用。inst_ori, inst_lui, inst_addiu, inst_beq 起源四指令，理解以完成第一步数据相关。前三个指令为操作数指令较为简单，beq 为跳转指令较复杂。

任何信号都不能无值，无值是错误的。

数据相关部分

Version0.1 到 0.0 的修改实现了数据相关（在 version0.4 中重新进行了修改详情见 github）

下述代码修改均在 ex.v 文件中进行。

output wire [37:0] ex_to_id_bus 接输出线从 ex 到 id 段

```
 assign ex_to_id_bus = {

        rf_we,          // 37

        rf_waddr,       // 36:32

        ex_result       // 31:0
```

}; rf_we 为信号一位 ,rf_waddr 为地址 ,ex_result 为传输数据即结果。

下述修改均在 id.v 中 input wire [37:0] ex_to_id_bus,

input wire [37:0] mem_to_id_bus,

input wire [37:0] wb_to_id_bus,接入接收线分别来自三个段 ex mem wb

.rdata2 (rdata2),

.we (wb_rf_we),

.waddr (wb_rf_waddr),

.wdata (wb_rf_wdata)

.wdata (wb_rf_wdata),

.ex_to_id_bus (ex_to_id_bus),

.mem_to_id_bus (mem_to_id_bus),

.wb_to_id_bus (wb_to_id_bus)

);加粗为添加部分，

下述修改均在 regfile.v

input wire [37:0] ex_to_id_bus,

input wire [37:0] mem_to_id_bus,

input wire [37:0] wb_to_id_bus,添加输入线

```
  wire ex_rf_we;
  wire [4:0] ex_rf_waddr ;
wire [31:0] ex_result;
wire mem_rf_we;
wire [4:0] mem_rf_waddr;
wire [31:0] mem_result;
wire wb_rf_we;
wire [4:0] wb_rf_waddr;
wire [31:0] wb_result;
assign {
      ex_rf_we,
      ex_rf_waddr,
      ex_result
} = ex_to_id_bus;
assign {
      mem_rf_we,
      mem_rf_waddr,
      mem_result
```

} = mem_to_id_bus;

assign {

wb_rf_we,

wb_rf_waddr,

wb_result

} = wb_to_id_bus;分别定义并打开将一根线上信号拆分，上述为 ex，mem，wb 段传入 regfile

((raddr1 == ex_rf_waddr) && ex_rf_we) ? ex_result :

((raddr1 == mem_rf_waddr) && mem_rf_we) ? mem_result :

((raddr1 == wb_rf_waddr) && wb_rf_we) ? wb_result :

((raddr2 == ex_rf_waddr) && ex_rf_we) ? ex_result :

((raddr2 == mem_rf_waddr) && mem_rf_we) ? mem_result :

((raddr2 == wb_rf_waddr) && wb_rf_we) ? wb_result :

reg_array[raddr2];

粗体为添加判断部分，地址是否相同并且信号值为 1，再进行三目运算符进行判断并操作。

output wire [37:0] mem_to_id_bus

assign mem_to_id_bus = {

rf_we, // 37

rf_waddr, // 36:32

ex_result // 31:0

};在 mem.v 中进行相同操作

output wire [37:0] wb_to_id_bus,

assign wb_to_id_bus = {

rf_we, // 37

rf_waddr, // 36:32

rf_wdata // 31:0

};在 wb.v 中进行相同操作

下述操作均在 core.v 中进行

wire [37:0] ex_to_id_bus;

wire [37:0] mem_to_id_bus;

wire [37:0] wb_to_id_bus;

.br_bus (br_bus),

.ex_to_id_bus (ex_to_id_bus),

.mem_to_id_bus (mem_to_id_bus),

.wb_to_id_bus (wb_to_id_bus)

.ex_to_id_bus (ex_to_id_bus)

.mem_to_id_bus (mem_to_id_bus)

.wb_to_id_bus (wb_to_id_bus)在 core 中定义

添加指令代码（github）

.s 文件查找指令值，定义，判断，操作数赋值，（注意跳转指令）

1.Subu

2.assign inst_addiu = op_d[6'b00_1001]; // 不用 decoder 函数也可写为 inst_addiu = (inst[31:26]==6'b00_1001)

3. assign inst_beq = op_d[6'b00_0100]; // 写为 op_d[6'b000100]也可

4. assign inst_subu = op_d[6'b00_0000] && (inst[10:6]==5'b00_000) && (inst[5:0]==6'b10_0011);

5. assign sel_alu_src1[0] = inst_ori | inst_addiu | inst_subu;

2.br

1. assign br_e = (inst_beq & rs_eq_rt) | inst_jr;

2. assign br_addr = inst_beq ? (pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b0}) :

3. inst_jr ? (rdata1) :

4. 32'b0;

Jal，Addu，Sll 同理不赘述

6.Or

assign br_e = (inst_beq & rs_eq_rt) | inst_jr | inst_jal | (inst_ben & (~rs_eq_rt));Code has comments. Press enter to view.

assign br_addr = inst_beq ? (pc_plus_4 + {{14{inst[15]}},inst[15:0],2'b0}) :

inst_jr ? (rdata1) :

inst_jal ? ({pc_plus_4[31:28], inst[25:0], 2'b0}) :

inst_ben ? (pc_plus_4 + {{14{inst[15]}}, inst[15:0], 2'b0}) :

32'b0;后面和 beq 相同

Lw 和 sw（一存一取）

1.数据通路重新命名

wire [37:0] ex_to_id_bus;

wire [37:0] mem_to_id_bus;

wire [37:0] wb_to_id_bus;

->

wire [37:0] ex_to_rf_bus;

wire [37:0] mem_to_rf_bus;

wire [37:0] wb_to_rf_bus;

2. 添加代码

assign mem_result = data_sram_rdata;等（GitHub）

加气泡

IF ID EX ME WB

IF ID EX ME WB

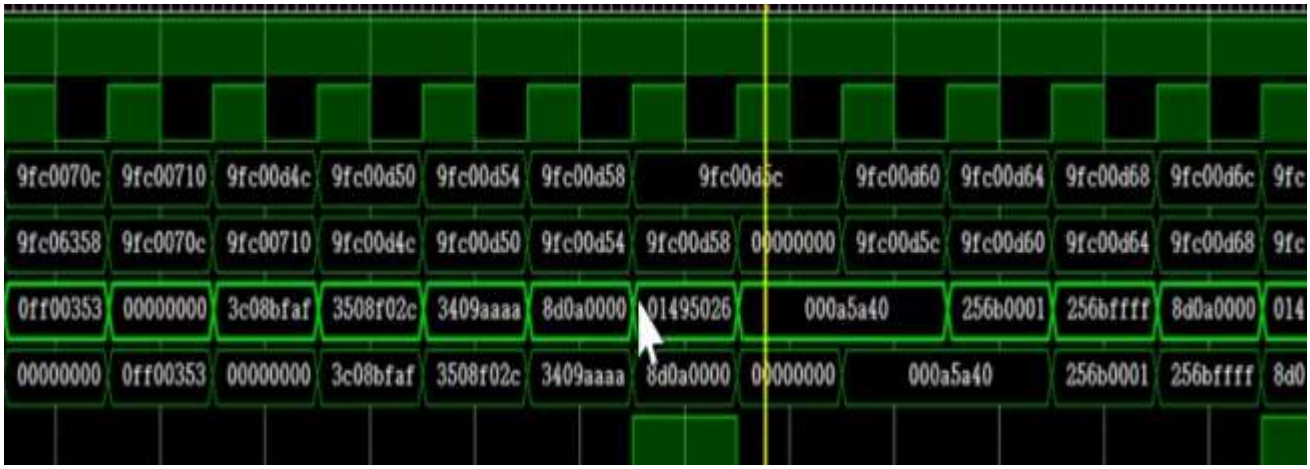
IF ID EX ME WB

IF ID EX ME WB

IF ID EX ME WB

便于视觉对齐效果将 MEM 缩略为 ME

需要气泡来进行相当与占位延时的操作，以便于正确的段接收到正确的数据。加在其中一级的气泡会同样影响下一级，所以下一级也需要加。



注意需要保存 inst 值否则并不会自动保存，导致 inst 值传递错误

增添宏定义

define EX_TO_RF_WD 38

define MEM_TO_RF_WD 38

define LoadBus 5

define SaveBus 3

添加数据选择信号// 选择最新的数据进行处理

```
    assign ndata1 = ((ex_rf_we && rs == ex_rf_waddr) ? ex_rf_wdata : 32'b0) |
                    ((!(ex_rf_we && rs == ex_rf_waddr) && (mem_rf_we && rs == mem_rf_waddr)) ? mem_rf_wdata : 32'b0) |
```

((!(ex_rf_we && rs == ex_rf_waddr) && !(mem_rf_we && rs == mem_rf_waddr) && (wb_rf_we && rs == wb_rf_waddr)) ? wb_rf_wdata : 32'b0) |

(((ex_rf_we && rs == ex_rf_waddr) || (mem_rf_we && rs == mem_rf_waddr) || (wb_rf_we && rs == wb_rf_waddr)) ? 32'b0 : rdata1);

assign ndata2 = ((ex_rf_we && rt == ex_rf_waddr) ? ex_rf_wdata : 32'b0) |

((!(ex_rf_we && rt == ex_rf_waddr) && (mem_rf_we && rt == mem_rf_waddr)) ? mem_rf_wdata : 32'b0) |

((!(ex_rf_we && rt == ex_rf_waddr) && !(mem_rf_we && rt == mem_rf_waddr) && (wb_rf_we && rt == wb_rf_waddr)) ? wb_rf_wdata : 32'b0) |

(((ex_rf_we && rt == ex_rf_waddr) || (mem_rf_we && rt == mem_rf_waddr) || (wb_rf_we && rt == wb_rf_waddr)) ? 32'b0 : rdata2);

添加 slt、slti、sltiu、J、add、addi、sub、and、andi、nor、xori、sllv、sra、srav、srl、srlv

添加指令 bgez、bgtz、blez、bltz

//跳转分支请求停顿信号 （当操作数地址是 EX 阶段要写回的寄存器地址时停顿使能有效）

assign stallreq_for_bru = (ex_id & (ex_rf_we & (rs == ex_rf_waddr | rt == ex_rf_waddr))) ? `Stop : `NoStop;指令都在 id.v 中进行的修改，bgez,bgtz,blez,bltz 均为跳转指令。

添加 HI/LO 寄存器实现（GitHub）

6. 总结

五级流水线 CPU 设计实现了完整的指令执行流水线，具备以下特点：采用哈佛架构，指令数据分离，支持完整的数据前递机制，具备多级流水线停顿控制，支持乘除法等复杂运算，提供完善的调试接口，采用模块化设计，各阶段职责清晰，通过合理的流水线划分和控制机制，在提高指令吞吐率的同时，有效处理了各种数据冒险和结构冒险。

7.实验感受及改进意见通过这次五级流水线 CPU 的设计与实现实验，我获得了深刻而全面的技术体验和工程感悟，这些感受涵盖了从理论认知到实践操作，从模块设计到系统集成的多个层面。

在理论理解方面，这次实验让我真正体会到了流水线技术如何在实践中提升处理器性能。在单周期 CPU 中，所有指令共享一个较长的时钟周期，而五级流水线将指令执行过程分解为取指、译码、执行、访存和写回五个相对独立的阶段，每个阶段可以在一个更短的时钟周期内完成。这种设计允许多条指令在同一时刻处于不同的执行阶段，形成了指令级并行。

模块化设计的重要性在本次实验中得到了充分体现。五个流水段各自独立又相互连接，通过定义清晰的总线接口进行通信。每个模块内部又有更细粒度的功能单元划分，如执行阶段的 ALU、乘法器、除法器等。这种层次化的设计不仅便于调试和验证，也使得功能扩展更加容易。当我需要增加新的指令支持时，通常只需要在相关模块中添加相应的译码逻辑和执行逻辑，而不需要改动整体架构。

调试过程是这次实验中最困难的环节。流水线 CPU 的调试远比单周期 CPU 复杂，因为错误可能源于多个指令在不同阶段的交互。我建立了系统的调试方法：首先编写小型测试程序，验证基本功能；然后逐步增加指令组合复杂度；最后用完整的应用程序测试。在波形调试中，我学会了设置有意义的观测信号，如各阶段的指令内容、寄存器值、控制信号等，通过跟踪指令在流水线中的流动，定位问题所在。最困难的一次调试是发现数据前递在特定序列下会传递错误的值，最终发现是因为在乘除法多周期操作时，前递逻辑没有正确识别数据就绪状态。

哈佛结构的优势在实际运行中得到了验证。指令存储器和数据存储器的分离使得取指和访存可以并行进行，这在波形中清晰可见——当执行阶段在进行计算时，取指阶段可以同时读取下一条指令，访存阶段可以访问数据存储器。这种并行性显著提高了系统整体带宽，但也带来了新的设计考虑，如两个存储器端口的仲裁和优先级处理。

总而言之，这次五级流水线 CPU 设计实验是一次从理论到实践的完整旅程。它让我不仅理解了流水线的工作原理，更体验了硬件设计的工程实践过程。每一个模块的调试成功，每一个测试用例的通过，都带来了巨大的成就感。这种将抽象

概念转化为实际可运行硬件的经历，是我学习计算机体系结构过程中最宝贵的收获。

最后非常感谢学长提供的学习资料和在学习过程中的疑难解答，让我获得了长足的进步。

参考资料：

【MIPS 实现 CPU 中的五级流水线 1 — 开端】https://www.bilibili.com/video/BV1Wb4y1E7tz?vd_source=e70616df3af1f4b08a573880675b816e 及其剩余四个视频共五个，jisuanjixitongshiyan\nscscc2022_group_v0.01\nscscc-group\doc_v0.01\A03_“系统能力培养大赛”MIPS 指令系统规范_v1.01.pdf

《自己动手做 cpu》《手把手教你学 FPGA 设计》菜鸟教程 verilog 语言