成绩评定日期：

实验报告成绩：

2024～2025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2201

组长：张一航

组员：王梓豪 雷鹏

报告日期：2024.12.31

目录

[1 实验概述 3](#_Toc25236)

[1.1小组成员及任务分工 3](#_Toc3577)

[1.2任务概述 3](#_Toc20522)

[1.3运行环境 3](#_Toc30185)

[2 实验方法与流程 4](#_Toc21352)

[2.1 设计流程 4](#_Toc21414)

[2.2 模块划分 4](#_Toc14829)

[2.3 接口定义 4](#_Toc32390)

[3 代码说明 6](#_Toc1310)

[3.1 IF段 6](#_Toc1051)

[3.2 ID段 6](#_Toc24322)

[3.3 EX段 11](#_Toc22054)

[3.4 MEM段 12](#_Toc27360)

[3.5 WB段 14](#_Toc18736)

[3.6 STALL CTRL段 15](#_Toc30710)

[3.7 HILO寄存器模块 16](#_Toc28914)

[4 感受与改进意见 17](#_Toc3780)

[4.1 张一航的心得体会 17](#_Toc4336)

[4.2 王梓豪的心得体会 17](#_Toc21943)

[4.3 雷鹏的心得体会 18](#_Toc19185)

[5 参考资料 18](#_Toc28739)

# 1 实验概述

## **1.1**小组成员及任务分工

|  |  |  |
| --- | --- | --- |
| 姓名 | 任务分工 | 任务量占比 |
| 张一航 | 参与ID段设计，完成EX、MEM阶段数据通路连接，完成store、load指令的部分逻辑 | 40% |
| 王梓豪 | 主要负责EX和MEM模块，实现高效准确的ALU操作和地址计算。集成并优化乘法和除法模块，确保复杂运算的正确性和效率。 | 40% |
| 雷鹏 | 主要负责WB和CRTL模块 | 20% |

## **1.2**任务概述

本次实验要求完成一个五级流水线的CPU设计。具体任务包括在现有框架代码的基础上，补充逻辑指令、算数指令、跳转指令及数据相关的处理。主要模块包括：

IF.v：取指阶段

ID.v：译码阶段

EX.v：执行阶段

MEM.v：访存阶段

WB.v：回写阶段

CTRL.v：流水线控制模块，负责暂停和恢复流水线

此外，lib文件夹下提供了必要的库文件，CTRL文件用于控制流水线的暂停和继续操作。图示

描述已自动生成

## **1.3**运行环境

· 硬件环境：采用CG服务器进行开发和仿真。

· 软件环境：Vivado 2019版本，用于Verilog代码的编写、综合和仿真。

# 2 实验方法与流程

## 2.1 设计流程

本次实验的设计流程主要包括以下几个步骤：

· 需求分析：明确实验目标，了解五级流水线CPU的基本结构和功能需求。

· 模块设计：根据流水线阶段划分模块，设计各个模块的功能和接口。

· 代码编写：使用Verilog HDL编写各个模块的代码，实现指令的取指、译码、执行、访存和回写功能。

· 模块集成：将各个模块集成到整体CPU架构中，确保模块间的正确连接和数据传输。

· 仿真测试：使用Vivado进行仿真，设计测试用例，验证CPU的功能正确性。

· 调试优化：根据仿真结果，定位和修复设计中的问题，优化性能。

· 文档撰写：整理实验过程和结果，编写实验报告。

## 2.2 模块划分

根据五级流水线的结构，将CPU划分为以下主要模块：

· IF段（取指模块）：负责从指令存储器中取出指令，并计算下一条指令的地址。

· ID段（译码模块）：对取出的指令进行译码，读取操作数，并生成控制信号。

· EX段（执行模块）：执行算术或逻辑运算，计算分支地址或内存地址。

· MEM段（访存模块）：访问数据存储器，进行读写操作。

· WB段（回写模块）：将运算结果写回寄存器。

· CTRL段（控制模块）：负责流水线的暂停和恢复，处理数据冒险和控制冒险。

每个模块之间通过数据总线进行通信，确保指令和数据的正确传输。

## 2.3 接口定义

为了确保模块之间的正确连接和数据传输，需要详细定义各个模块的接口信号。以下是各模块的主要接口定义：

**IF段接口：**

· 输入：时钟信号（clk）、复位信号（rst）、暂停信号（stall）、分支跳转· 信号（br\_bus）

· 输出：IF段到ID段的数据总线（if\_to\_id\_bus）、指令存储器使能信号（inst\_sram\_en）、写使能信号（inst\_sram\_wen）、指令地址（inst\_sram\_addr）、指令数据（inst\_sram\_wdata）

**ID段接口：**

· 输入：时钟信号（clk）、复位信号（rst）、暂停信号（stall）、IF段数据（if\_to\_id\_bus）、指令数据（inst\_sram\_rdata）、来自EX段的数据（ex\_id）、来自WB段的数据（wb\_to\_rf\_bus）、来自EX段的寄存器数据（ex\_to\_rf\_bus）、来自MEM段的寄存器数据（mem\_to\_rf\_bus）、来自EX段的hilo寄存器数据（ex\_hi\_lo\_bus）

· 输出：ID段到EX段的数据总线（id\_to\_ex\_bus）、分支跳转信号（br\_bus）、ID段的hilo寄存器数据（id\_hi\_lo\_bus）、执行load命令的数据总线（id\_load\_bus）、执行save命令的数据总线（id\_save\_bus）、暂停请求信号（stallreq\_for\_bru）

**EX段接口：**

· 输入：时钟信号（clk）、复位信号（rst）、暂停信号（stall）、ID段数据（id\_to\_ex\_bus）、ID段的load数据（id\_load\_bus）、ID段的save数据（id\_save\_bus）

· 输出：EX段到MEM段的数据（ex\_to\_mem\_bus）、EX段到寄存器文件的数据（ex\_to\_rf\_bus）、EX段到hilo寄存器的数据（ex\_hi\_lo\_bus）、对EX段的stall请求（stallreq\_for\_ex）、内存数据的读写使能信号（data\_sram\_en）、内存数据的写使能信号（data\_sram\_wen）、内存数据地址（data\_sram\_addr）、要写入内存的数据（data\_sram\_wdata）、EX段传给ID段的数据（ex\_id）、内存数据选择信号（data\_ram\_sel）、EX段读取的数据（ex\_load\_bus）

**MEM段接口：**

· 输入：时钟信号（clk）、复位信号（rst）、暂停信号（stall）、EX段数据（ex\_to\_mem\_bus）、从内存中读出的数据（data\_sram\_rdata）、内存数据选择信号（data\_ram\_sel）、EX段读取的数据（ex\_load\_bus）

· 输出：MEM段到WB段的数据（mem\_to\_wb\_bus）、MEM段到寄存器文件的数据（mem\_to\_rf\_bus）、对EX段的stall请求（stallreq\_for\_load）

**WB段接口：**

· 输入：时钟信号（clk）、复位信号（rst）、暂停信号（stall）、MEM段数据（mem\_to\_wb\_bus）

· 输出：WB段传给寄存器文件的数据（wb\_to\_rf\_bus）、用于调试的PC值（debug\_wb\_pc）、用于调试的写使能信号（debug\_wb\_rf\_wen）、用于调试的写寄存器地址（debug\_wb\_rf\_wnum）、用于调试的写寄存器数据（debug\_wb\_rf\_wdata）

**CTRL段接口：**

· 输入：时钟信号（clk）、EX段的stall请求（stallreq\_for\_ex）、BRU的stall请求（stallreq\_for\_bru）、Load命令的stall请求（stallreq\_for\_load）

· 输出：暂停信号（stall）

通过详细定义各模块的接口，可以确保模块之间的数据传输和控制信号的正确连接，从而实现流水线CPU的功能。

# 3 代码说明

## **3.1** IF段

该段的输入输出总线定义如下方图片及表格所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk | 1 | 输入 | 时钟信号 |
| 2 | rst | 1 | 输入 | 复位信号 |
| 3 | stall | 6 | 输入 | 暂停信号，控制指令是否暂停 |
| 4 | br\_bus | 33 | 输入 | 分支跳转信号，控制延迟槽是否跳转 |
| 5 | if\_to\_id\_bus | 33 | 输出 | IF段到ID段的数据总线 |
| 6 | inst\_sram\_en  inst\_sram\_en  \_en | 1 | 输出 | 读写使能信号 |
| 7 | inst\_sram\_wen | 4 | 输出 | 写使能信号 |
| 8 | inst\_sram\_addr | 32 | 输出 | 存放指令寄存器的地址 |
| 9 | inst\_sram\_wdata | 32 | 输出 | 存放指令寄存器的数据 |

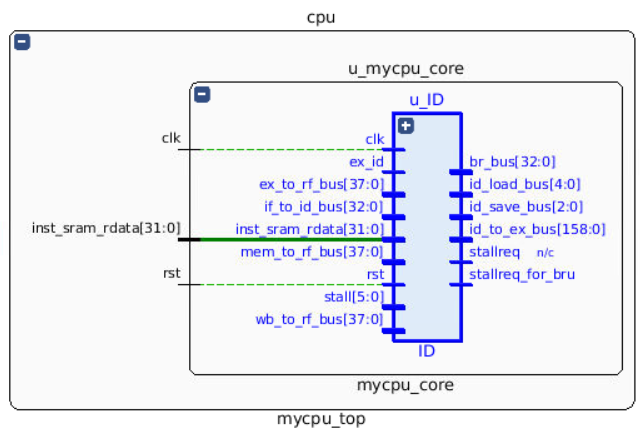
图示

描述已自动生成

这一阶段相对简单，没有太多需要改动的地方，大部分时候负责的工作是从PC+4的地址处获取下一条指令。只需要注意几个跳转指令需要跳到对应的地址读取指令。

## **3.2** ID段

该段的输入输出总线定义如下方图片及表格所示：



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk | 1 | 输入 | 时钟信号 |
| 2 | rst | 1 | 输入 | 复位信号 |
| 3 | stall | 6 | 输入 | 暂停信号，控制指令是否暂停 |
| 4 | stallreq | 1 | 输出 | 暂停请求信号 |
| 5 | if\_to\_id\_bus | 33 | 输入 | IF段到ID段的数据总线 |
| 6 | inst\_sram\_rdata | 1 | 输入 | 读写使能信号 |
| 7 | ex\_id | 1 | 输入 | 写使能信号 |
| 8 | wb\_to\_rf\_bus | 38 | 输入 | WB段存放进寄存器的数据 |
| 9 | ex\_to\_rf\_bus | 38 | 输入 | EX段存放进寄存器的数据 |
| 10 | mem\_to\_rf\_bus | 38 | 输入 | MEM段存放进寄存器的数据 |
| 11 | ex\_hi\_lo\_bus | 66 | 输入 | EX段存放进hilo寄存器的数据的总线 |
| 12 | id\_hi\_lo\_bus | 72 | 输出 | ID段存放进hilo寄存器的数据的总线 |
| 13 | id\_load\_bus | 5 | 输出 | ID段执行load命令的数据总线 |
| 14 | id\_save\_bus | 3 | 输出 | ID段执行save命令的数据总线 |
| 15 | stallreq\_for\_bru | 1 | 输出 | 执行load命令时的暂停请求 |
| 16 | id\_to\_ex\_bus | 159 | 输出 | ID段到EX段的数据总线 |
| 17 | br\_bus | 33 | 输出 | 分支跳转信号，控制延迟槽是否跳转 |

ID段的主要功能是指令解码。

日程表

描述已自动生成

根据32位MIPS指令集类型可获取译码结果。根据I、J、R型指令对应部分读取寄存器文件中地址为 rs（inst[25;21]）以及地址为rt(inst[20:16]的通用寄存器,得到rdata1以及rdata2，并且通过判断是否发生数据相关，从而更改rdata1以及 rdata2的值。（由于数据相关，在这里，传给下一段的寄存器数据是ndata而非rdata）

    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);

同时根据opcode和func部分分析要执行的运算，给对应的ALU的相关控制逻辑赋值，其中，0表示该条指令不采用该 ALU，1 表示采用该ALU，同时将所有的 ALU 标识符组合起来成为 alu\_op，alu\_op为十二位宽，代表 16 种不同的 ALU，并且作为传入 EX 段的一部分。要写入的目的寄存器。rf\_we 代表写使能信号，表示该条指令是否用写入通用寄存器，sel\_rf\_dst[0]表示该指令要将计算结果写入 rd 通用寄存器，sel\_rf\_dst[1]表示该指令要将计算结果写入 rt 通用寄存器,sel\_rf\_dst[2]表示该指令要将计算结果写入 31 号通用寄存器。rf\_waddr 表示要该条指令的计算结果要写入的通用寄存器的地址，data\_ram\_en表示该条指令是否要与内存中取值或者写入值，如果该条指令要从内存中取值或者写入值，那么它将被赋值为 1’b1,data\_ram\_wen 为四位宽，表示该条指令是否要写入寄存器，如果该条指令要将计算结果的第几个字节写入寄存器，那么对应位置的值设为 1。

用sel\_alu\_src1和sel\_alu\_src2来判断操作数来源，第一个操作数有三种来源，第二个操作数有四种来源，通过区分不同的指令进而分辨不同的操作数的来源。

传值用ID段得到的数据，分别给id\_to\_ex\_bus和br\_bus赋值，其中br\_bus是传给IF段的用于传输跳转指令的判断信号和跳转的地址。

    // rs to reg1

    assign sel\_alu\_src1[0] =    inst\_lw | inst\_sw |inst\_lb| inst\_lbu  | inst\_lh | inst\_lhu| inst\_sb | inst\_sh |

                                inst\_ori | inst\_addiu | inst\_or | inst\_xor | inst\_and  | inst\_andi| inst\_nor | inst\_xori |

                                inst\_sub | inst\_subu | inst\_add | inst\_addi | inst\_addu |

                                inst\_jr | inst\_bgezal | inst\_bltzal |

                                inst\_slti | inst\_or | inst\_srav | inst\_sltu | inst\_slt | inst\_sltiu | inst\_sllv| inst\_srlv

                                ;

    // pc to reg1

    assign sel\_alu\_src1[1] = inst\_jal | inst\_jalr | inst\_bltzal | inst\_bgezal;

    // sa\_zero\_extend to reg1

    assign sel\_alu\_src1[2] = inst\_sll | inst\_sra | inst\_srl;

    // rt to reg2

    assign sel\_alu\_src2[0] =    inst\_sub | inst\_subu | inst\_addu | inst\_sll | inst\_or | inst\_xor | inst\_sra | inst\_srl |

                                inst\_srlv | inst\_sllv| inst\_sra | inst\_srav| inst\_sltu | inst\_slt  | inst\_add | inst\_and| inst\_nor ;

                                // inst\_div | inst\_divu |inst\_mult | inst\_multu;

    // imm\_sign\_extend to reg2

    assign sel\_alu\_src2[1] =    inst\_lui | inst\_addiu | inst\_lw | inst\_sw | inst\_slti| inst\_sltiu | inst\_addi |

                                inst\_lb  | inst\_lbu   | inst\_lh  | inst\_lhu | inst\_sh | inst\_sb;

    // 32'b8 to reg2

    assign sel\_alu\_src2[2] = inst\_jal | inst\_jalr | inst\_bgezal | inst\_bltzal;

    // imm\_zero\_extend to reg2

    assign sel\_alu\_src2[3] = inst\_ori | inst\_andi | inst\_xori;

跳转指令：先用br\_e表示这条指令是否为跳转指令，用rs\_ge\_z表示是否满足 rdata1的值大于等于 0，用rs\_le\_z表示是否满足rdata1的值小于等于0，用rs\_lt\_z表示是否满足 rdata1的值小于 0，rs\_eq\_rt表示是否满足rdata1是否等于radta2的值。br\_addr 表示跳转后的地址，根据不同的指令对地址做不同的计算，并将结果赋给br\_addr。

    assign br\_e = inst\_beq & rs\_eq\_rt | inst\_j | inst\_jalr | | inst\_jr | inst\_jal | inst\_bne & ~rs\_eq\_rt |

                  inst\_bgez & rs\_ge\_z | inst\_bgtz & rs\_gt\_z |inst\_blez & rs\_le\_z | inst\_bltz & rs\_lt\_z |

                  inst\_bgezal & rs\_ge\_z | inst\_bltzal & rs\_lt\_z ;

    assign br\_addr =

                        (inst\_beq       ? (pc\_plus\_4 + {{14{inst[15]}},inst[15:0],2'b0}) : 32'b0)   |

                        (inst\_jr        ? ndata1 : 32'b0)                                           |

                        (inst\_jal       ? {pc\_plus\_4[31:28],instr\_index,2'b0} : 32'b0)              |

                        (inst\_j         ? ({ pc\_plus\_4[31:28],inst[25:0],2'b0}) : 32'b0)            |

                        (inst\_jalr      ? ndata1:32'b0)                                             |

                        (inst\_bne       ? (pc\_plus\_4 + {{14{inst[15]}},inst[15:0],2'b0}) : 32'b0)   |

                        (inst\_bgez      ? (pc\_plus\_4 + {{14{inst[15]}},inst[15:0],2'b0}) : 32'b0)   |

                        (inst\_bgtz      ? (pc\_plus\_4 + {{14{inst[15]}},inst[15:0],2'b0}) : 32'b0)   |

                        (inst\_blez      ? (pc\_plus\_4 + {{14{inst[15]}},inst[15:0],2'b0}) : 32'b0)   |

                        (inst\_bltz      ? (pc\_plus\_4 + {{14{inst[15]}},inst[15:0],2'b0}) : 32'b0)   |

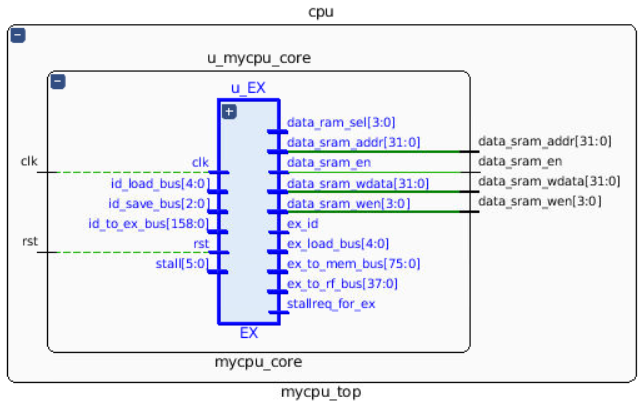
                        (inst\_bgezal    ? (pc\_plus\_4 + {{14{inst[15]}},inst[15:0],2'b0}) : 32'b0)   |

                        (inst\_bltzal    ? (pc\_plus\_4 + {{14{inst[15]}},inst[15:0],2'b0}) : 32'b0)

                        ;

## 3.3 EX段

该段的输入输出总线定义如下方图片及表格所示：

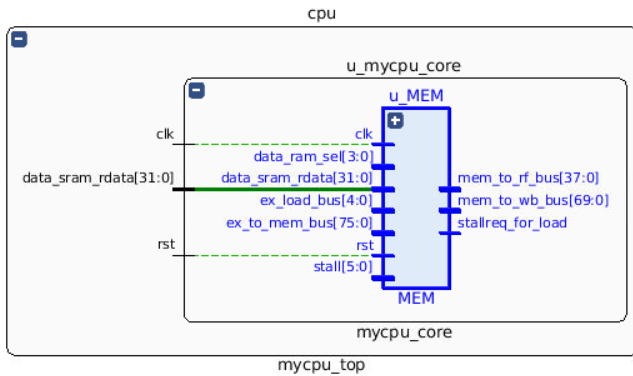


|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk | 1 | 输入 | 时钟信号 |
| 2 | rst | 1 | 输入 | 复位信号 |
| 3 | stall | 6 | 输入 | 控制暂停信号 |
| 4 | id\_to\_ex\_bus | 169 | 输入 | ID 段传给 EX 段的数据 |
| 5 | id\_load\_bus | 5 | 输入 | ID段传递读的数据 |
| 6 | id\_save\_bus | 3 | 输入 | ID段传递写的数据 |
| 7 | ex\_to\_mem\_bus | 80 | 输出 | EX 段传给 MEM 段的数据 |
| 8 | ex\_to\_rf\_bus | 38 | 输出 | EX 段传给regfile段的数据 |
| 9 | ex\_hi\_lo\_bus | 66 | 输出 | EX 段传给hilo段的数据 |
| 10 | stallreq\_for\_ex | 1 | 输出 | 对EX段的stall请求 |
| 11 | data\_sram\_en | 1 | 输出 | 内存数据的读写使能信号 |
| 12 | data\_sram\_wen | 4 | 输出 | 内存数据的写使能信号 |
| 13 | data\_sram\_addr | 32 | 输出 | 内存数据存放的地址 |
| 14 | data\_sram\_wdata | 32 | 输出 | 要写入内存的数据 |
| 15 | ex\_id | 38 | 输出 | EX 段传给 ID 段的数据 |
| 16 | data\_ram\_sel | 4 | 输出 | 内存数据的选择信号 |
| 17 | ex\_load\_bus | 5 | 输出 | EX 段读取的数据 |

EX段的任务也比较简单，主要负责计算从ID段传来的各类数据（存储于寄存器中，包括地址、数据等）。ALU已在库文件中提供，只需设置相关控制信号即可。

## **3.4** MEM段

该段的输入输出总线定义如下方图片及表格所示：



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk | 1 | 输入 | 时钟信号 |
| 2 | rst | 1 | 输入 | 复位信号 |
| 3 | stall | 6 | 输入 | 控制暂停信号 |
| 4 | ex\_to\_mem\_bus | 80 | 输入 | EX 传给 MEM 段的数据 |
| 5 | data\_sram\_rdata | 32 | 输入 | 从内存中读出来要写入寄存器的值 |
| 6 | data\_ram\_sel | 4 | 输入 | 内存数据的选择信号 |
| 7 | ex\_load\_bus | 5 | 输入 | EX 段读取的数据 |
| 8 | stallreq\_for\_load | 1 | 输出 | 对EX段的stall请求 |
| 9 | mem\_to\_wb\_bus | 70 | 输出 | MEM 传给 WB 段的数据 |
| 10 | mem\_to\_rf\_bus | 38 | 输出 | MEM段传给regfile段的数据 |

主要任务是访存取数，核心逻辑如下：

    assign b\_data = data\_ram\_sel[3] ? data\_sram\_rdata[31:24] :

                    data\_ram\_sel[2] ? data\_sram\_rdata[23:16] :

                    data\_ram\_sel[1] ? data\_sram\_rdata[15: 8] :

                    data\_ram\_sel[0] ? data\_sram\_rdata[ 7: 0] : 8'b0;

    assign h\_data = data\_ram\_sel[2] ? data\_sram\_rdata[31:16] :

                    data\_ram\_sel[0] ? data\_sram\_rdata[15: 0] : 16'b0;

    assign w\_data = data\_sram\_rdata;

    assign mem\_result = inst\_lb     ? {{24{b\_data[7]}},b\_data} :

                        inst\_lbu    ? {{24{1'b0}},b\_data} :

                        inst\_lh     ? {{16{h\_data[15]}},h\_data} :

                        inst\_lhu    ? {{16{1'b0}},h\_data} :

                        inst\_lw     ? w\_data : 32'b0;

    assign rf\_wdata = sel\_rf\_res & data\_ram\_en ? mem\_result : ex\_result;

## **3.5** WB段

该段的输入输出总线定义如下方图片及表格所示：

图片包含 文本

描述已自动生成

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk | 1 | 输入 | 时钟信号 |
| 2 | rst | 1 | 输入 | 复位信号 |
| 3 | stall | 6 | 输入 | 控制暂停信号 |
| 4 | mem\_to\_wb\_bus | 70 | 输入 | MEM 传给 WB 的数据 |
| 5 | wb\_to\_rf\_bus | 38 | 输出 | WB 传给 rf 的数据 |
| 6 | debug\_wb\_pc | 32 | 输出 | 用来 debug 的 pc 值 |
| 7 | debug\_wb\_rf\_wen | 4 | 输出 | 用来 debug 的写使能信号 |
| 8 | debug\_wb\_rf\_wnum | 5 | 输出 | 用来 debug 的写寄存器地址 |
| 9 | debug\_wb\_rf\_wdata | 32 | 输出 | 用来 debug 的写寄存器数据 |

该段任务是将数据写回内存，没有什么需要改动的地方。

核心逻辑：提取从MEM传入的相关控制信号，然后将写回信号发送给寄存器堆完成内存写入操作：

    assign {

        wb\_pc,

        rf\_we,

        rf\_waddr,

        rf\_wdata

    } = mem\_to\_wb\_bus\_r;

    assign wb\_to\_rf\_bus = {

        rf\_we,

        rf\_waddr,

        rf\_wdata

    };

## **3.6** STALL CTRL段

该段的输入输出总线定义如下方图片及表格所示：

图示

中度可信度描述已自动生成

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk | 1 | 输入 | 时钟信号 |
| 2 | stallreq\_for\_ex | 1 | 输入 | 执行阶段的指令是否请求流水线暂停 |
| 3 | stallreq\_for\_bru | 5 | 输入 | Load命令是否请求流水线暂停 |
| 4 | stall | 6 | 输出 | 暂停信号 |

Stall信号共6位，每一位分别代表流水线中某一段的暂停信号。stall[0]表示取指地址PC是否保持不变，为1表示保持不变；stall[1]表示流水线取指阶段是否暂停，为1表示暂停；stall[2]表示流水线译码阶段是否暂停，为1表示暂停；stall[3]表示流水线执行阶段是否暂停，为1表示暂停。stall[4]表示流水线访存阶段是否暂停，为1表示暂停；stall[5]表示流水线回写阶段是否暂停，为1表示暂停。

核心逻辑：

    always @ (\*) begin

        if (rst) begin

            stall = `StallBus'b0;

        end

        //stallreq\_for\_ex, stallreq\_for\_bru, stallreq\_for\_load

        else if (stallreq\_for\_ex) begin

            stall = `StallBus'b001111;

        end

        //

        else if (stallreq\_for\_bru) begin

            stall = `StallBus'b000111;

        end

        else if (stallreq\_for\_load) begin

            stall = `StallBus'b000011;

        end

        else begin

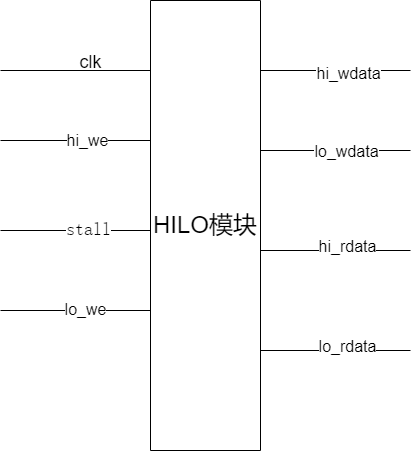
            stall = `StallBus'b0;

        end

    end

## 3.7 HILO寄存器模块

**整体说明：**

hi和lo属于协处理器，不在通用寄存器的范围内，这两个寄存器主要是在用来处理乘法和除法。以乘法作为示例，如果两个整数相乘，那么乘法的结果低位保存在lo寄存器，高位保存在hi寄存器。 当然，这两个寄存器也可以独立进行读取和写入。读的时候，使用mfhi、mflo；写入的时候，用mthi、mtlo。和通用寄存器不同，mfhi、mflo是在执行阶段才开始从hi、lo寄存器获取数值的。写入则和通用寄存器一样，也是在写回的时候完成的。

可以直接改写lib下的regfile.v，也可以添加hiloreg.v，创建u\_hi\_lo\_reg，但是MEM、WB也要跟着改，这里我们采用第二种方法，即添加hiloreg.v文件。接口如右图所示。

表 8 HILO寄存器输入输出

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 接口名 | 宽度 | 输入/输出 | 作用 |
| 1 | clk | 1 | 输入 | 时钟信号 |
| 2 | stall | 6 | 输入 | 控制暂停信号 |
| 3 | hi\_we | 1 | 输入 | hi寄存器的写使能信号 |
| 4 | lo\_we | 1 | 输入 | lo寄存器的写使能信号 |
| 5 | hi\_wdata | 32 | 输出 | Hi寄存器写的数据 |
| 6 | lo\_wdata | 32 | 输出 | Lo寄存器写的数据 |
| 7 | hi\_rdata | 32 | 输出 | Hi寄存器读的数据 |
| 8 | lo\_rdata | 32 | 输出 | Lo寄存器读的数据 |

**功能说明：**

当hi\_we和lo\_we均为1时，寄存器reg\_hi和reg\_lo同时将hi\_wdata和lo\_wdata写入。当hi\_we为0，lo\_we为1时，reg\_lo将lo\_wdata写入；当hi\_we为1，lo\_we为0时，reg\_hi将hi\_wdata写入。hi\_rdata和lo\_rdata分别输出reg\_hi和reg\_lo中的数据。

# 4 感受与改进意见

## 4.1 张一航的心得体会

在这次实验中，我负责了IF和ID模块的设计与实现。通过这个过程，我对五级流水线的工作原理有了更深入的理解，特别是在处理数据前递和数据冒险时，我学到了如何通过调整控制信号来避免冲突，确保指令顺利执行。尽管这一部分实现比较复杂，但通过反复调试，我逐渐掌握了流水线控制信号的细节。

不过，在实现过程中，我发现对于某些复杂的指令执行，控制信号的设计仍然存在一定的挑战，尤其是在遇到多级数据冒险时，如何保证流水线稳定性和高效性仍需要更精确的调试。建议在未来的实验中可以提前模拟更复杂的指令集，避免在实际实现时出现较大的设计调整。

此外，实验让我对Verilog语言的使用更加熟悉，尤其是在硬件设计中的应用，通过编写代码并调试，提升了我的编程能力和问题分析能力。然而，在调试过程中，Verilog的语法和调试工具仍然存在一定的难度，未来可以更多学习仿真工具的使用，提高调试效率。

团队协作方面，我们每个人的分工非常明确，及时的沟通和反馈让我体会到团队合作的重要性。但有时在模块整合时，代码风格和实现方式的差异会导致一些小问题，建议团队在开始之前可以提前达成一些设计规范和代码标准，以提高整体的协作效率。

## 4.2 王梓豪的心得体会

作为EX和MEM模块的负责人，我的主要任务是设计ALU操作和地址计算模块，并优化乘法与除法模块。在EX阶段，我学到了如何设计高效的ALU，并通过调整控制信号来确保不同指令可以正确计算。尽管设计过程挑战性大，但通过不断的调试，我掌握了运算模块的设计思路，并在性能优化方面积累了经验。

然而，在处理乘法和除法时，遇到了一些瓶颈，尤其是在硬件实现时，如何保证计算的准确性与速度是一个比较复杂的问题。我认为，未来可以更多地借助硬件加速方法（如流水线或并行处理）来优化乘除法操作的性能。此外，乘法和除法模块的调试过程较为繁琐，未来可以考虑增加更完善的测试覆盖，提前发现潜在问题。

在MEM阶段，我设计的访存模块让我对内存与寄存器之间的数据交换有了更深入的认识。尤其是在处理不同宽度数据访问时，我理解到如何合理地设计数据选择和信号控制逻辑来提高系统性能。但在实现过程中，我意识到不同访存操作的处理方式还有提升空间，建议以后可以深入探讨各种数据存取模式，优化访存性能。

通过这次实验，我也加强了与组员的沟通与合作，大家互相支持、分享思路，共同解决问题。但有时在工作分配时还存在一定的重叠，建议在分工时更加细致，避免后期出现重复劳动，提升整体效率。

## 4.3 雷鹏的心得体会

在实验中，我负责了WB和CTRL模块的设计与实现。WB模块的设计相对简单，主要任务是将计算结果写回寄存器，尽管操作简单，但我意识到确保数据传递准确性对于整个流水线的稳定性至关重要。通过设计和调试WB模块，我加深了对寄存器文件操作的理解，确保数据能够正确无误地写回。

在CTRL模块的设计中，遇到了较多挑战，特别是如何有效处理来自EX段和BRU的多种stall请求。我通过设计合理的控制逻辑，优化了暂停和恢复机制，提高了流水线的稳定性和性能。然而，在面对复杂的流水线暂停时，CTRL模块的设计有时会变得复杂，我认为未来可以考虑使用更高效的状态机设计来进一步简化控制逻辑。

在设计过程中，我也遇到了如何有效管理多种stall请求的挑战，经过多次调整，我设计了一套合理的控制策略，确保不同暂停请求能够及时得到响应。调试过程中，我利用Vivado仿真工具发现了很多潜在问题，并且通过快速定位和修复问题，提升了设计的可靠性。未来可以进一步加强对工具的使用，提升调试效率。

最后，团队合作上，尽管我们有较好的沟通与协调，但在模块整合时，还是遇到了一些兼容性问题。建议在后续的实验中，能更早地进行模块的预集成和接口测试，以减少模块间的差异，保证最终集成的顺利进行。

# 5 参考资料

1、张晨曦 著《计算机体系结构》（第二版） 高等教育出版社

2、雷思磊 著《自己动手写 CPU》 电子工业出版社

3、（美）DavidA.Patterson、John L.Hennessy 著 《计算机组成与设计：硬件、软件接口（原书第 4 版）》

4、Yale N.Patt 著 《计算机系统概论（原书第 2 版）》

5、龙芯杯官方的参考文档

6、Verilog HDL官方文档及教程。