实验报告成绩：	成绩评定日期：

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

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

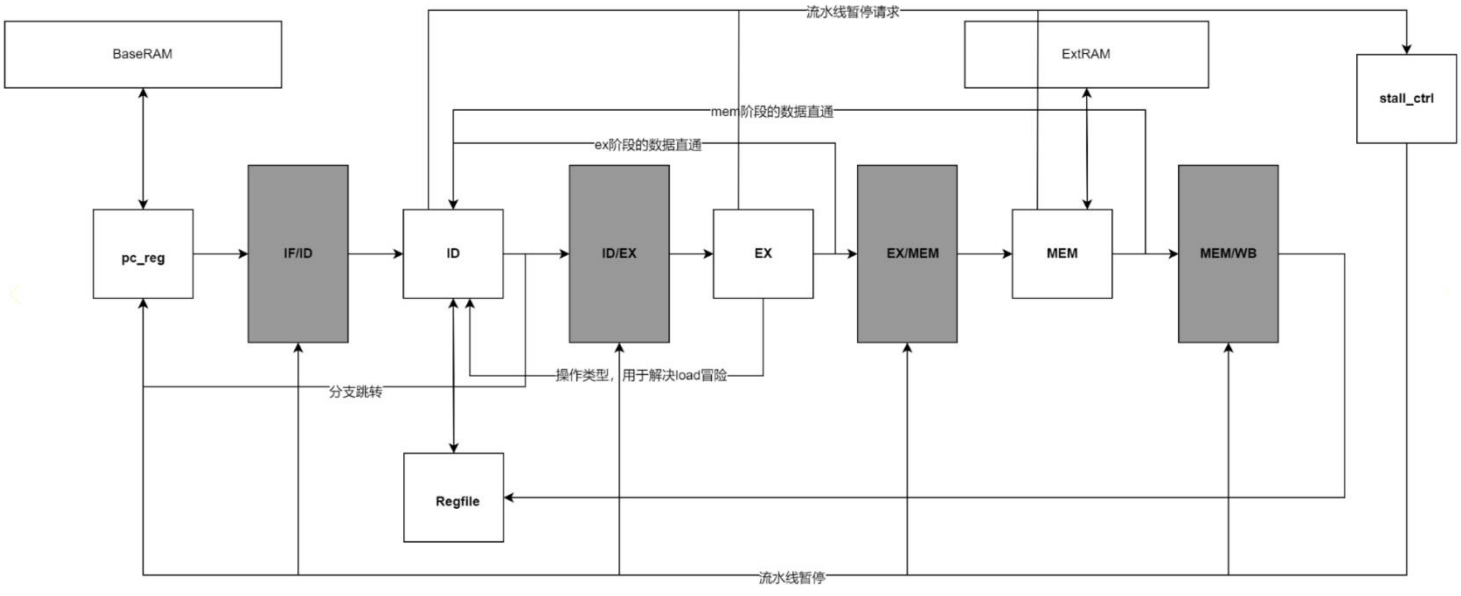
EX.v：执行阶段

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。



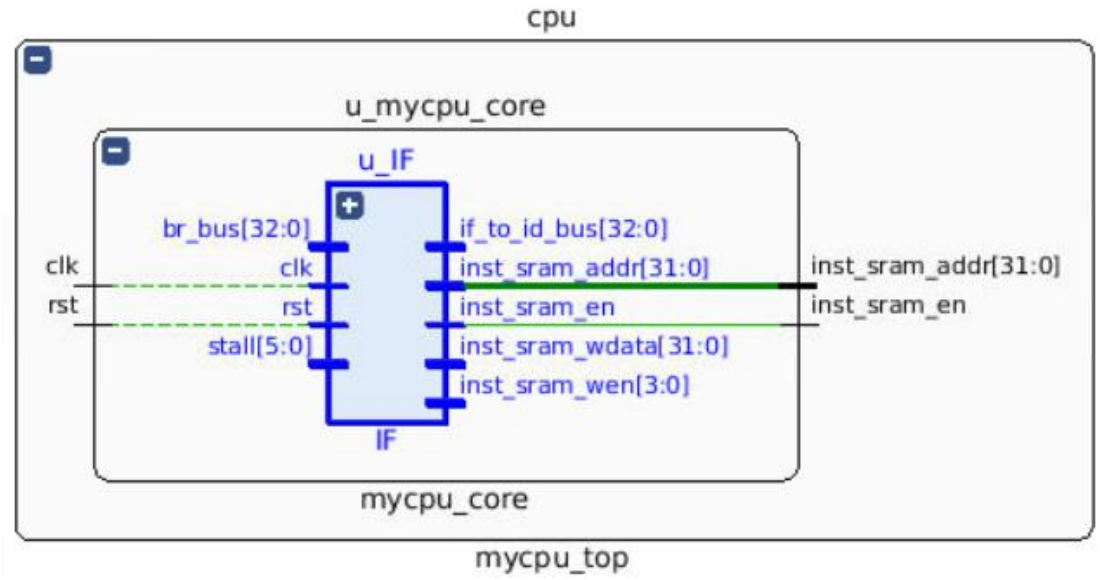
1.3 运行环境

- 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 代码说明

2.1 IF 段

该段的输入输出总线定义如下方图片及表格所示：

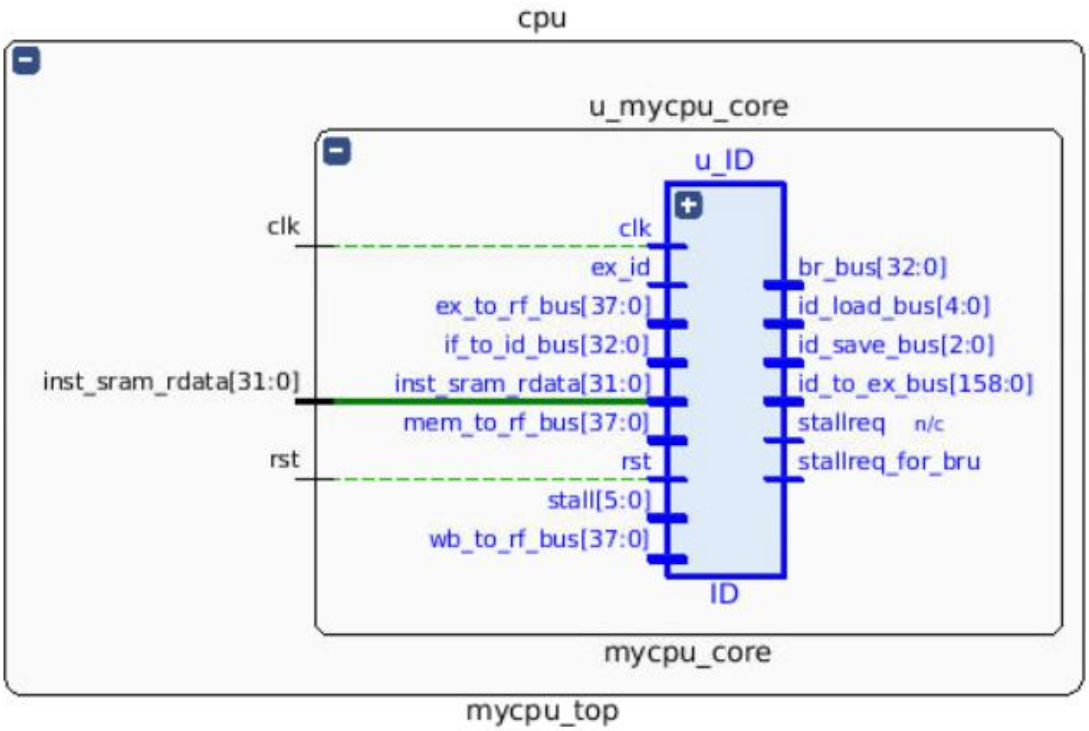


序号	接口名	宽度	输入/输出	作用
1	clk	1	输入	时钟信号
2	rst	1	输入	复位信号
3	stall	6	输入	暂停信号，控制指令是否暂停
4	br_bus	33	输入	分支跳转信号，控制延迟槽是否跳转
5	if_to_id_bus	33	输出	IF 段到 ID 段的数据总线
6	inst_sram_en	1	输出	读写使能信号
7	inst_sram_wen	4	输出	写使能信号
8	inst_sram_addr	32	输出	存放指令寄存器的地址
9	inst_sram_wdata	32	输出	存放指令寄存器的数据

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。

2.2 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	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
12	id_hi_lo_bus	72	输出	ID 段存放进 hilo 寄存器的数据的总
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。

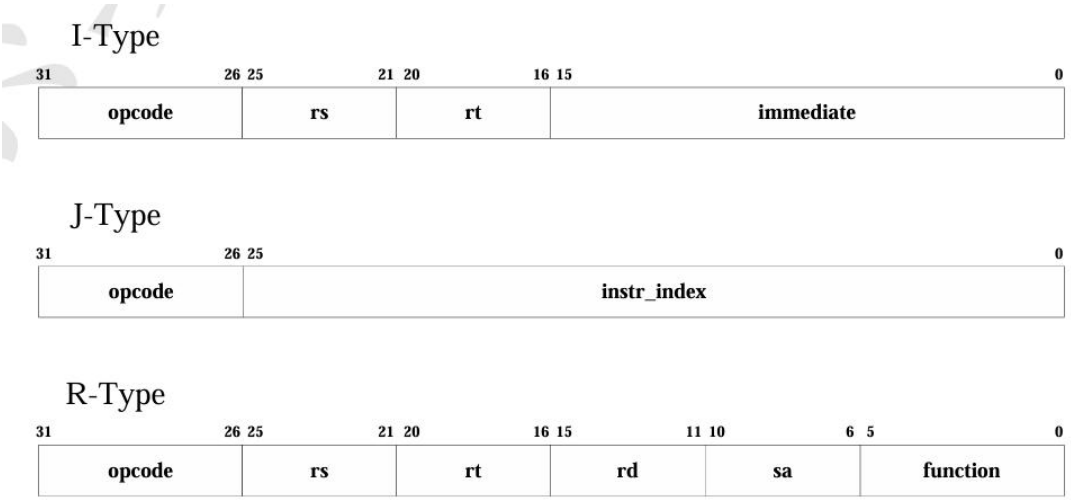


图 3-1 指令格式

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;
```

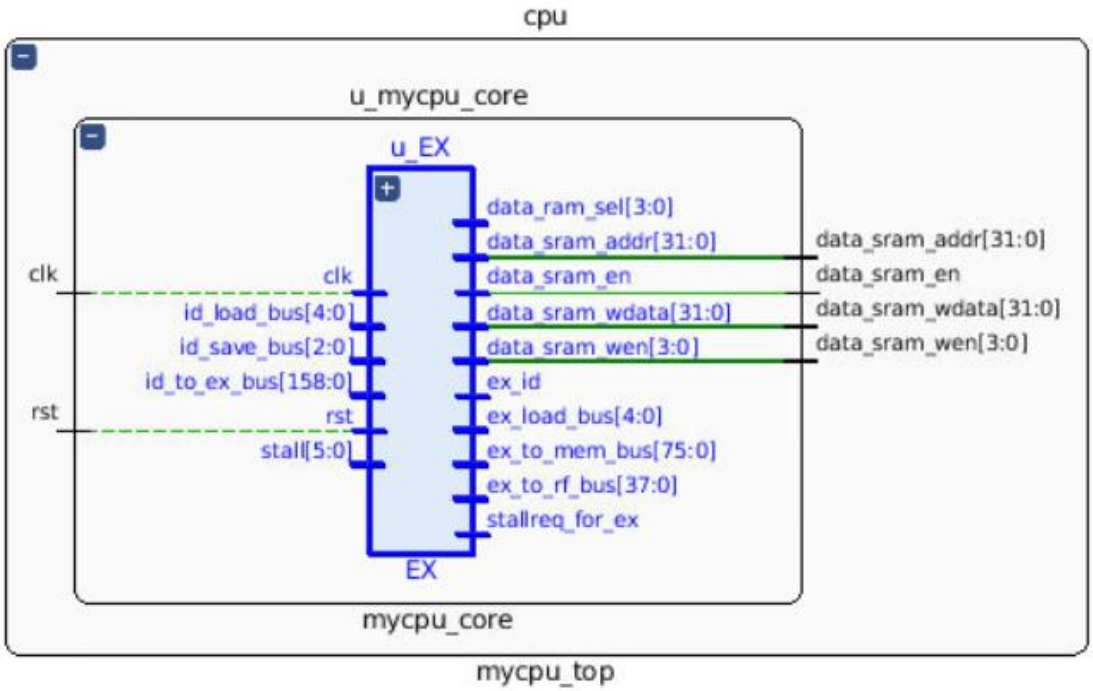
跳转指令：先用 br_e 表示这条指令是否为跳转指令，用 rs_ge_z 表示是否满足 rdata1 的值大于等于 0,用 rs_le_z 表示是否满足 rdata1 的值小于等于 0，用 rs_lt_z 表示是否满足 rdata1 的值小于 0，rs_eq_rt 表示是否满足 rdata1 是否等于 radta2 的值。br_addr 表示跳转后的地址，根据不同的指令对地址做不同的计算，并将结果赋给 br_addr。

```
    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)
                                    ;
```

2.3 EX 段

该段的输入输出总线定义如下方图片及表格所示：

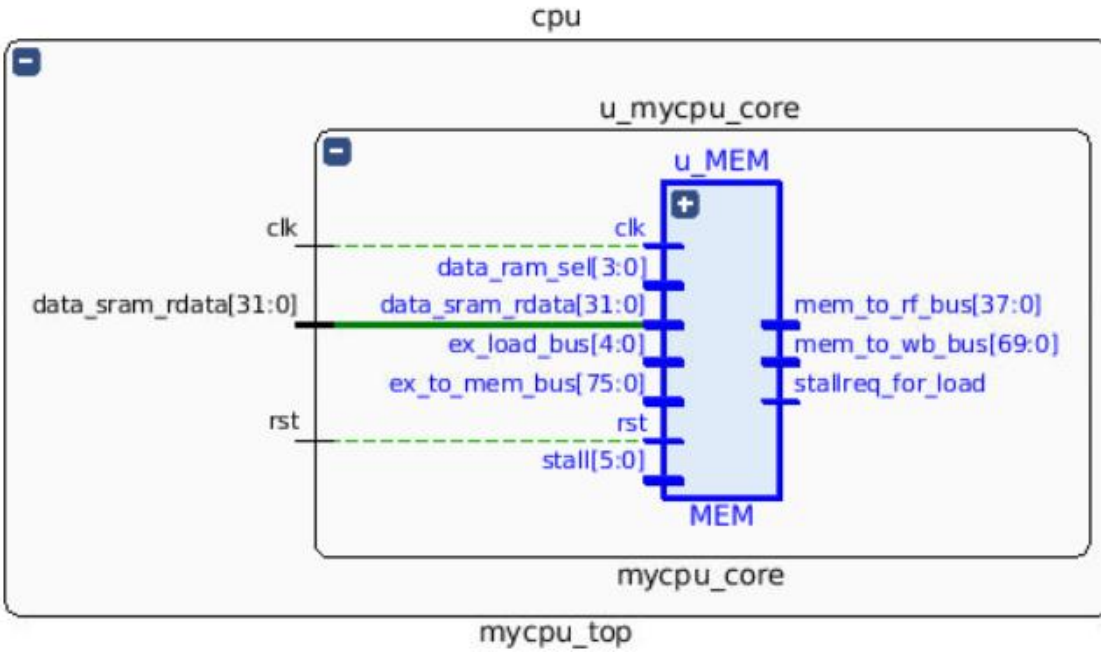


序号	接口名	宽度	输入/输出	作用
1	clk	1	输入	时钟信号
2	rst	1	输入	复位信号
3	stall	6	输入	控制暂停信号
4	id_to_ex_bus	169	输入	ID 段传给 EX 段的数据
5	id_load_bus	5	输入	ID 段传递读的数据
6	id_save_bus	3	输入	ID 段传递写的数据
7	ex_to_mem_bus	80	输出	EX 段传给 MEM 段的数据
8	ex_to_rf_bus	38	输出	EX 段传给 regfile 段的数据
9	ex_hi_lo_bus	66	输出	EX 段传给 hilo 段的数据
10	stallreq_for_ex	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。

2.4 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

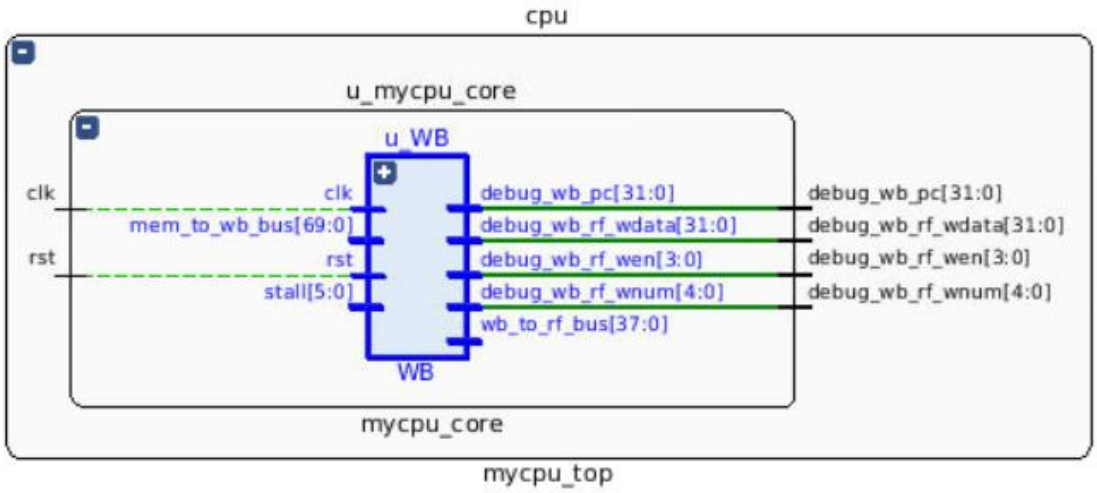
主要任务是访存取数，核心逻辑如下：

```
      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;
```

2.5 WB 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	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

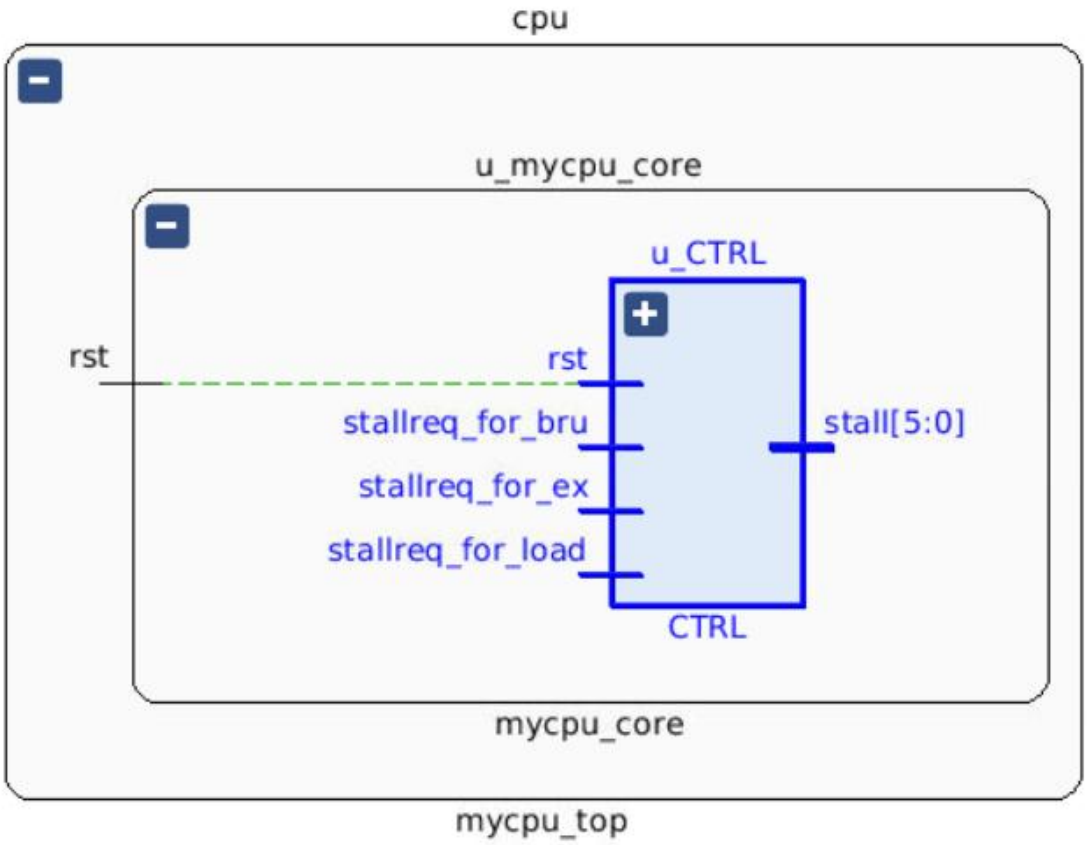
该段任务是将数据写回内存，没有什么需要改动的地方。

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
    };
```

2.6 STALL CTRL 段

该段的输入输出总线定义如下方图片及表格所示：



序号	接口名	宽	输入/输	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
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
```

2.7 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。

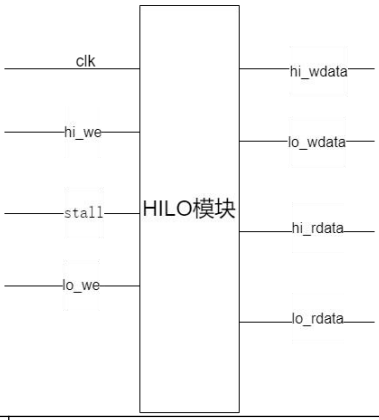


表 8 HILO 寄存器输入输出

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。

3 感受与改进意见

3.1 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。

3.2 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。

3.3 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。

4 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。