实验报告成绩：	成绩评定日期：

2021～2022 学年秋季学期

A3705060050《计算机系统》必修课

课程实验报告

班级：人工智能 1902

组长：陈乐奇

组员：汤博皓、董苡廷

报告日期：2021.12.18

目录

一、工作量分配

A 汤博皓

完成了开始端的 EX,MEM,WB 段数据相关并填写指令到 sll。完成乘除法器模块的使用、hilo 寄存器在 regfile 模块的实现，并添加 hilo 寄存器读和写的四条指令

B 陈乐奇

陈乐奇：加了 sll，or，and，lw，sw，xor，add，addi，bne，slt，slti，sltiu，j，srl，srlv，bgtz，lw，sw 指令，共 18 条。完善了 lw，sw 及其指令访存相关的操作，添加了流水线暂停机制，即加气泡操作。 修改了 b 系列指令在 reg1 和 reg2 处赋值的 bug。增加了 32 周期移位乘法器。

C 参与了接线，完成了从 andi 到 bgezal 的指令以及 lb lbu lh lhu sb sh 的指令的添加董苡廷

二、整体设计

五级流水线的各个阶段的主要工作如下：

2.1 取指阶段

从指令存储器读取指令，同时确定下一条指令地址。

PC 段：给出指令地址，其中实现指令指针寄存器 PC，该寄存器的值就是指令地址。

IF 段：实现取值与译码阶段之间的寄存器，将取指阶段的结果、取得的指令、指令地址等信息再下一个使用周期传递到译码阶段。

2.2 译码阶段

对指令进行译码，从通用寄存器中读取要使用的寄存器的值，如果指令中含有立即数，那么还要将立即数进行符号扩展或无符号扩展。如果是转移指令并且满足转移条件，那么给出转移目标作为新的指令地址。

ID 段：对指令进行译码，译码结果包括运算类型、运算所需的原操作数、要写入的目的寄存器等。

Regfile 段：实现了 32 个 32 位的通用整数寄存器，可以同时进行两个寄存器的读操作和一个寄存器的写操作。

2.3 执行阶段

按照译码阶段给出的操作数、运算类型进行运算，给出运算结果，如果是 load store 指令，那么海汇计算 load store 的目标地址。

EX 段：依据译码阶段的结果进行指定的运算，给出运算结果。实现执行预防存阶段之间的寄存器，将执行阶段的结果在下一个使用周期传递到缓存阶段。

DIV 模块：实现出除法操作。

MUL 模块：实现有符号和无符号乘法操作。

2.4 访存阶段

如果是 load store 指令，那么在此阶段会访存数据存储器，反之，只是将执行阶段的结果向下传递到回写阶段，同时在此阶段还要判断是否有异常需要处理，如果有，那么会清除流水线，然后转移到异常处理例程入口地址处继续执行。MEM 段：如果是加载存储指令，那么会对数据存储器进行访问，还会进行异常判断。

2.5 回写阶段

（WB 段）将运算结果保存到目标寄存器。

2.6CTRL 模块

这个模块虽然不属于五段流水线，却担当着接受来自五段流水线暂停信号，控制流水线的暂停与运行，保证五段流水线在某些特殊情况，如读写冲突，乘除法器等占用多个时钟周期的情况依然能正确运行的功能。

其中的流水线暂停机制又叫加气泡操作。在本次实验中，加气泡主要涉及 ID 段和 EX 段。ID 段的加气泡涉及 load 指令的 RAW 冲突操作，而 EX 段的加气泡涉及 32 周期移位乘法器和除法器需要占用多个时钟周期。

在这些情况下需要暂停流水线。直观思想是只需保持取指令地址 PC 的值不变，同时保持流水线各个阶段的寄存器不变。而实验所用的则是采用的是一种改进的方法：假如位于流水线第 n 阶段的指令需要多个时钟周期，进而请求流水线暂停，那么需保持取指令地址 PC 的值不变，同时保持流水线第 n 阶段以及之前的各个阶段的寄存器不变，而第 n 阶段后面的指令继续运行。比如：流水线执行阶段的指令请求流水线暂停，那么保持 PC 不变，同时保持取指、译码、执行阶段的寄存器不变，但是可以允许访存、回写阶段的指令继续运行。

代码中的 CTRL 模块，其作用是接收各阶段如 ID，EX 段传递过来的流水线暂停请求信号，从而控制流水线各阶段的运行。为了实现流水线暂停机制。而 CTRL 的输入 stall 有 6 位，其含义如下：

stall[0]表示取指地址 PC 是否保持不变，为 1 表示保持不变。

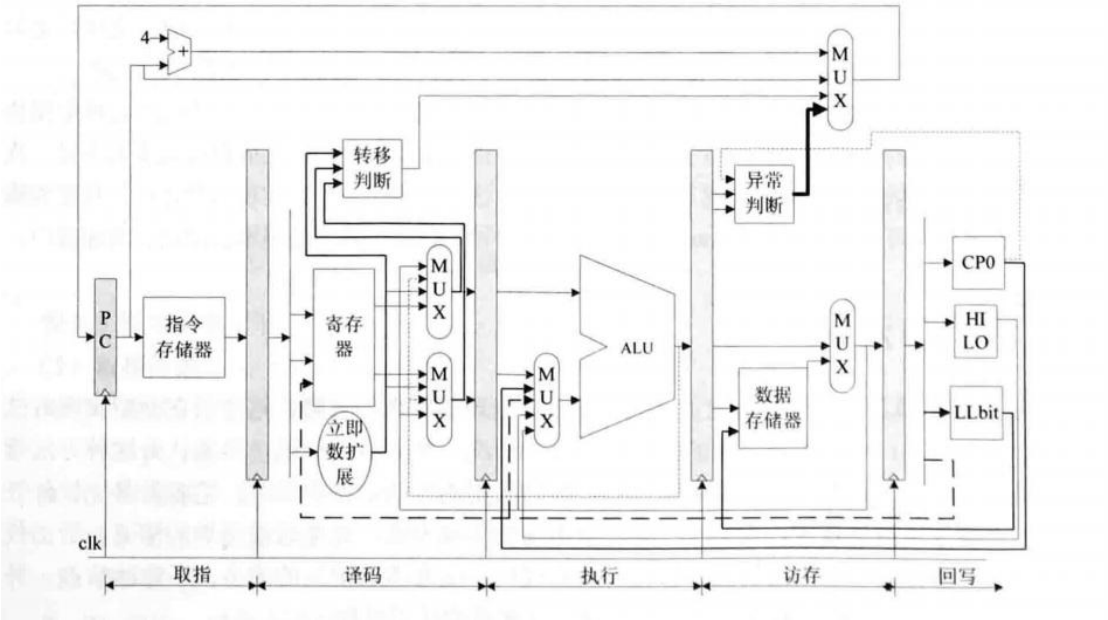
stall[1]表示流水线取指阶段是否暂停，为 1 表示暂停。

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。



2.7 添加 MIPS 指令集

同时本实验按顺序实现了 52 条 MIPS 指令，指令集包括：

ORI，LUI，ADDIU，BEQ，SUBU，JAL，JR，ADDU，SLL，OR，LW，SW，XOR，BNE，SLTU，SLT，SLTI，SLTIU，J，ADD，ADDI，SUB，AND，ANDI，NOR，XORI，SLLV，SRA，SRAV，SRLV，SRL，BGTZ，BGEZ，BLEZ，BLTZ，BLTZAL，BGEZAL，MOVN，MOVZ，MFLO，MTHI，MTLO，DIV，DIVU，MULT，MULU，LB，LBU，LH，LHU，SB，SH

其中分为一下几大类：

2.7.1 逻辑操作指令：

八条指令，and、andi、or、ori、xor 、xori、 nor、 lui，分别表示实现逻辑与、或、异或、或非等运算。

2.7.2 位移操作指令：

六条指令，sll 、sllv 、sra 、srav 、srl 、srlv，分别表示实现逻辑左移、逻辑右移、算术左移和算术右移等运算。

2.7.3 移动操作指令：

六条指令，movn、movz、mfhi、mflo、mtlo、mthi，用于通用寄存器之间的数据移动，以及通用寄存器与 HI、LO 寄存器之间的数据移动。

2.7.4 算术操作指令：

十三条指令，add、addi、addiu、addu、sub、subu、slt、 sltu、 mul、 mult、multu、 div、 divu，实现了加法、减法、乘法、乘累加、除法等运算。

2.7.5 转移指令：

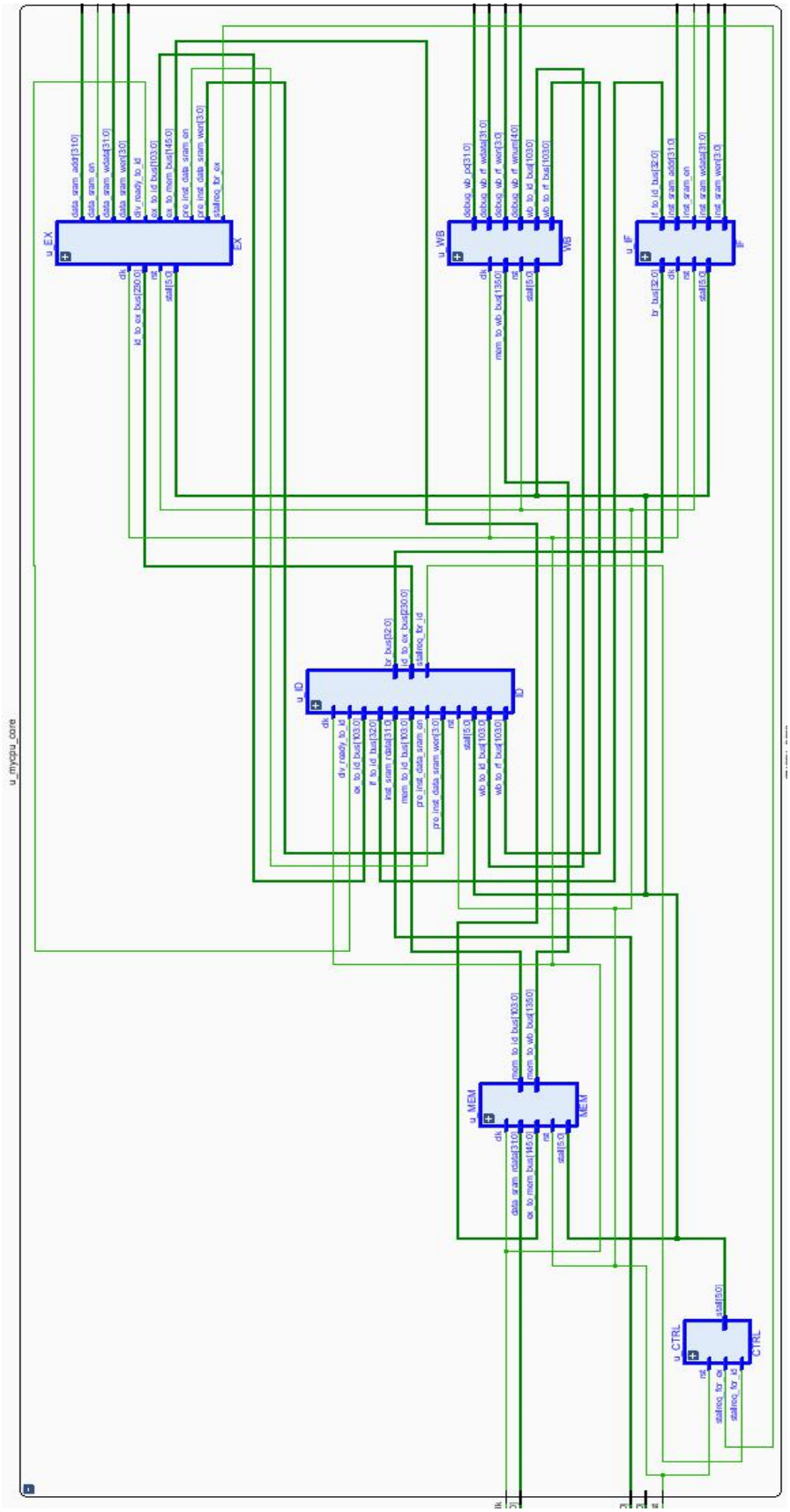
十一条指令 jr、 jalr、j、jal、beq、bgez、bgezal、bgtz、bltz、 bltzal、bne，分为有条件跳转和无条件跳转两种。

2.7.6 加载缓存指令：

八条指令 lw、sw、lb、 lbu、 lh、lhu、sb、sh，“l ”开头的都是加载指令，“s”开头的是存储指令，用于向存储器中读取数据或向存储器中保存数据。

程序运行环境及实用工具：本实验基于 Vivado2019，应用 Verilog HDL 语言对 CPU 底层进行设计与仿真处理。

三、不同流水段的连接图（横图）



四、单个流水段的说明

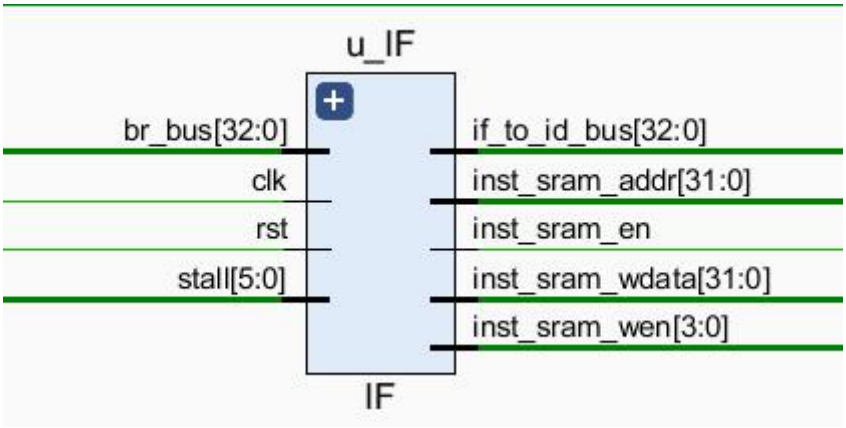
4.1、IF 段

4.1.1 整体功能：

根据 PC 值从存储器中取出指令，并将指令送入指令寄存器中，PC 值增加 4，指向顺序的下一条指令，并将指令地址放入临时寄存器中。

4.1.2 接口描述：

序号	接口名	宽度	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	br_bus[32:0]	33	输入	传入 PC 跳转信息
4	stall[5:0]	6	输入	流水线暂停请求信号
5	if_to_id_bus	33	输出	IF 段输出到 ID 段
6	inst_sram_addr[31:0]	32	输出	
7	inst_sram_en	1	输出	
8	inst_sram_wdata[31:0]	32	输出	
9	Inst_sram_wen[3:0]	4	输出	



IF 段结构示意图

序号	线路	宽度	作用	输入/输出
5	if_to_id_bus	32	将得到的 pc 地址送入 ID	输出

信号包含：

ce_reg:

pc_reg：程序计数器

pc 段作用是给出指令地址。

当复位信号为 1 时，启用 PC 寄存器，如果流水线暂停请求信号为不暂停，那么禁用 PC 寄存器。

```
always @ (posedge clk) begin
    if (rst) begin
        ce_reg <= 1'b0;
    end
    else if (stall[0]==`NoStop) begin
        ce_reg <= 1'b1;
    end
end
```

ce_reg 为 1 时禁用 PC 寄存器。而如果 stall[0]为 Stop 时，表示后面的段有加气泡操作，需要保持此时的 PC 地址不变，为保障气泡前面的段的数据维持不变，起到一个暂停的效果。

序号	线路	宽度	作用	输入/输出
3	br_bus[32:0]	33	传入 PC 跳转信息	输入

信号包含：

br_e:ID 段传输给 IF 段的 PC 地址跳转信号。

br_addr：ID 段传输给 IF 段的地址跳转信息。

跳转关键代码如下：

```
assign next_pc = br_e ? br_addr
                      : pc_reg + 32'h4;
```

跳转信号为 1，则 PC 的下一个地址为跳转的地址信息。否则，PC 地址自增 4。

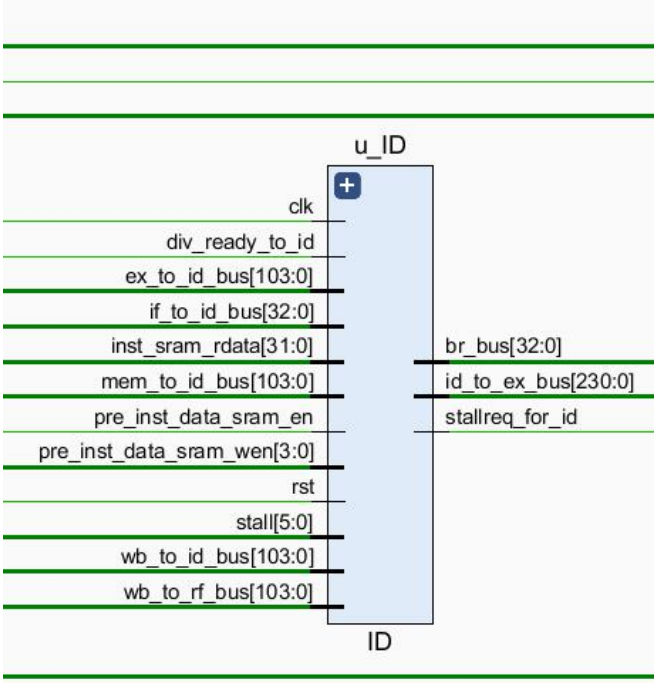
4.2、ID 段

4.2.2 整体功能

操作：进行指令译码，读取指令寄存器，并将读出的结果放入两个临时寄存器 reg1 和 reg2 中。对指令寄存器中内容的低十六位进行符号扩展，将符号扩展后的 32 位立即数保存在临时寄存器 Imm 中。并且在 ID 中加入 regfile 寄存器，同时进行对两个寄存器的读操作和一个寄存器的写操作。

ID 段需要解析指令，判断这个指令是否要读取寄存器，是否要进行有符号扩展展。还是说用立即数。是否要写入寄存器。同时还需要判断该条指定是否要进行 alu 操作，为这条指令分配合适的算术运算。

通俗地讲就是让计算机知道这条指令是要干什么的，同时让计算机得出要使用的寄存器，或者让立即数进行拓展（方便后续指令执行），亦或者（转移指令）是给出转移目的寄存器与转移条件；



ID 段结构示意图

4.2.3、接口描述

序号	接口名	宽度	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	if_to_id_bus[32:0]	33	输入	IF 段输入到 ID 段总线
4	ex_to_id_bus[103:0]	104	输入	EX 段输入到 ID 段总线
5	mem_to_id_bus[103:0]	104	输入	MEM 段输入到 ID 段总线
6	wb_to_id_bus[103:0]	104	输入	WB 段开端

				输入到 ID 段总线
7	wb_to_rf_bus[103:0]	104	输入	WB 段结束回写到 ID 段总线
8	div_ready_to_id	1	输入	除法运算是否结束信号
9	inst_sram_rdata[31:0]	32	输入	ID 段得到的指令信息
10	pre_inst_data_sram_en	1	输入	上一条指令是否有访存操作
11	pre_inst_data_sram_wen[3:0]	4	输入	上一条指令是读还是写
12	stall[5:0]	6	输入	
13	br_bus[32:0]	33	输出	ID 段传输到 IF 段的 PC 跳转信息
14	id_to_ex_bus[230:0]	231	输出	ID 段输出到 EX 段总线
15	stallreq_for_id	1	输出	是否在 ID 段发出加气泡请求

序号	线	宽度	输入/输出	作用
3	if_to_id_bus[32:0]	33	输入	IF 段输入到 ID 段总线

信号包含：

br_e:ID 段传输给 IF 段的 PC 地址跳转信号。

br_addr：ID 段传输给 IF 段的地址跳转信息。

序号	线	宽度	输入/输出	作用
4	ex_to_id_bus[103:0]	104	输入	EX 段输入到 ID 段总线

信号包含：

ex_w_hi_we:EX 段传输到 ID 段的 hi 寄存器写信号。

ex_w_hi_i：EX 段传输到 ID 段的 hi 寄存器数据。

ex_w_lo_we：EX 段传输到 ID 段的 lo 寄存器写信号。

ex_w_lo_i：EX 段传输到 ID 段的 lo 寄存器写数据。

ex_to_id_we:EX 段传输到 ID 段的要写入的寄存器的使能信号。

ex_to_id_waddr:EX 段传输到 ID 段的要写入的寄存器地址。

ex_result：EX 段传输到 ID 段的要写入的寄存器数据。

序号	线	宽度	输入/输出	作用
5	mem_to_id_bus[103:0]	104	输入	MEM段输入到ID段总线

信号包含：

mem_w_hi_we:MEM 段传输到 ID 段的 hi 寄存器写信号。

mem_w_hi_i：MEM 段传输到 ID 段的 hi 寄存器数据。

mem_w_lo_we：MEM 段传输到 ID 段的 lo 寄存器写信号。

mem_w_lo_i：MEM 段传输到 ID 段的 lo 寄存器写数据。

mem_to_id_we:MEM 段传输到 ID 段的要写入的寄存器的使能信号。

mem_to_id_waddr:MEM 段传输到 ID 段的要写入的寄存器地址。

mem_result：MEM 段传输到 ID 段的要写入的寄存器数据。

序号	线	宽度	输入/输出	作用
6	wb_to_id_bus[103:0]	104	输入	WB 段开端输入到 ID 段总线

信号包含：

wb_w_hi_we:WB 段传输到 ID 段的 hi 寄存器写信号。

wb_w_hi_i：WB 段传输到 ID 段的 hi 寄存器数据。

wb_w_lo_we：WB 段传输到 ID 段的 lo 寄存器写信号。

wb_w_lo_i：WB 段传输到 ID 段的 lo 寄存器写数据。

wb_to_id_we:WB 段传输到 ID 段的要写入的寄存器的使能信号。

wb_to_id_waddr:WB 段传输到 ID 段的要写入的寄存器地址。

wb_result：WB 段传输到 ID 段的要写入的寄存器数据。

序号	线	宽度	输入/输出	作用
7	wb_to_rf_bus[103:0]	104	输入	WB 段结束回写到 ID 段总线

信号包含：

w_hi_we:WB 段回写到 ID 段的 hi 寄存器写信号。

w_hi_i：WB 段回写到 ID 段的 hi 寄存器数据。

w_lo_we：WB 段回写到 ID 段的 lo 寄存器写信号。

w_lo_i：WB 段回写到 ID 段的 lo 寄存器写数据。

rf_we:WB 段回写到 ID 段的要写入的寄存器的使能信号。

rf_waddr:WB 段回写到 ID 段的要写入的寄存器地址。

rf_wdata：WB 段回写到 ID 段的要写入的寄存器数据。

序号	线	宽度	输入/输出	作用
13	br_bus[32:0]	33	输出	ID 段传输到 IF 段的 PC 跳转信息

信号包含：

br_e:ID 段传输给 IF 段的 PC 地址跳转信号。

br_addr：ID 段传输给 IF 段的地址跳转信息。

序号	线	宽度	输入/输出	作用
14	id_to_ex_bus[230:0]	231	输出	ID 段传输到 IF 段的 PC 跳转信息

data_ram_readen:

hi_read:hi 寄存器读信号

lo_read:lo 寄存器读信号

hi_write:hi 寄存器写信号

lo_write:lo 寄存器写信号

hi_out_file:hi 寄存器读数据

lo_out_file:lo 寄存器读数据

id_pc：程序计数器

inst：指令数据

alu_op：运算类型

sel_alu_src1：记录要读取的 reg1 寄存器

sel_alu_src2：记录要读取的 reg2 寄存器

data_ram_en：记录 load 访存信号

data_ram_wen：记录 store 写存信号
rf_we:regfile 存储使能信号
rf_waddr：数据存储地址
sel_rf_res：判断是使用 alu 运算还是 load 存储模块
rdata1：regfile 模块 reg1 的读数据
rdata2：regfile 模块 reg2 的读数据

4.2.4、功能模块说明：

（1）、ID 段气泡暂停机制：

当复位信号为 1，或者流水线暂停信号 IF 段暂停，ID 段未暂停时，ID 段不接收 IF 段输入总线，并将其置为 0。否则当 IF 段暂停信号是未暂停，那么 ID 段接受 IF 段的输入总线。代码如下：

```
always @ (posedge clk) begin
    inst_reg_en <= 1'b0;
    if (rst) begin
        if_to_id_bus_r <= `IF_TO_ID_WD'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;
    end
    else if (stall[1]==`NoStop) begin
        if_to_id_bus_r <= if_to_id_bus;
    end
end
```

inst_reg_en：寄存器存储 inst 指令信号
inst_reg：存储 inst 指令

当流水线暂停信号 ID 段为暂停，且除法运算没有结束时，寄存器保留上一条指令。否则，不存储上一条运行指令。

```
    //保证 inst 和 PC 同步，inst_reg_en 为 1，则表示 inst_reg 里面的需要用
    always @ (posedge clk) begin
```

```
        if(stall[2]==`Stop && div_ready_to_id==1'b0)begin
             inst_reg <= inst;
             inst_reg_en <= 1'b1;
        end
        else begin
            inst_reg <= 32'b0;
            inst_reg_en <= 1'b0;
        end
    end
```

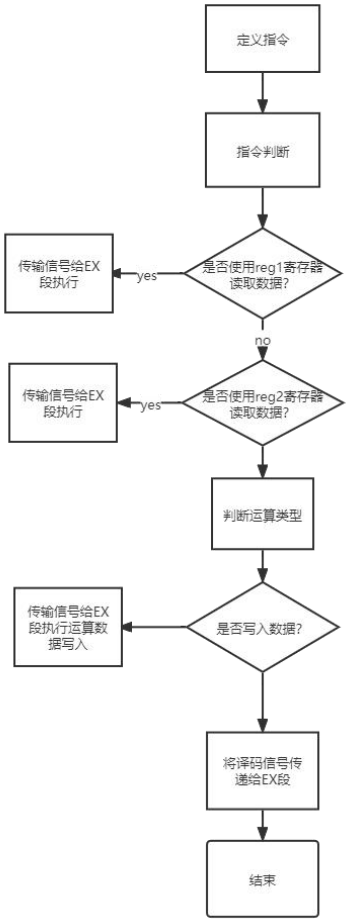
当存储信号为 0 时，ID 段 inst 识别接受下一条指令，否则，inst 使用存储器存储的指令信号，如下图。

```
assign inst = (inst_reg_en == 1'b0) ? inst_sram_rdata    : inst_reg;
```

保证 inst 和 PC 同步，inst_reg_en 为 1，则表示 inst_reg 读入 inst。

、

（2）ID 段译码识别指令：



译码大致流程

指令定义：

根据 A03 手册定义指令名称，类型为 wire，二进制 1 位长。

指令定义示例：

wire inst_ori, inst_lui, inst_addiu, inst_addu, inst_beq, inst_jr,inst_jal, inst_subu, inst_sll;

指令类型判断：

根据 32 位 inst 判断其满足哪一个指令条件，如果满足某一条指令条件，那么这条指令为 1。

指令判断示例：

```
assign inst_ori        = op_d[6'b00_1101];
assign inst_lui        = op_d[6'b00_1111] & (rs == 5'b0_0000);
//assign inst_lui        = op_d[6'b00_1111];
assign inst_addiu      = op_d[6'b00_1001];
```

```
assign inst_beq          = op_d[6'b00_0100];
```

hilo 寄存器模块根据指令定义对 hilo 寄存器的读或写操作。

```
assign hi_read = inst_mfhi;
assign lo_read = inst_mflo;
assign hi_write = inst_mthi;
assign lo_write = inst_mtlo;
```

寄存器读取数据：

wire [2:0] sel_alu_src1:

wire [3:0] sel_alu_src2:

定义 sel_alu_src1 和 sel_alu_src2 分别判断是否使用读取 reg1 和 reg2 寄存器，并指出读取的是 reg 中具体哪一个寄存器。

sel_alu_src1[0]：指 reg1 中的 rs 寄存器，也可能是 base 寄存器（但对应 rs 寄存器）

sel_alu_src1[1]：指 reg1 中的 pc 寄存器

sel_alu_src1[2]：指 reg1 中的 sa 指令寄存器

sel_alu_src2[0]：指 reg2 中的 rt 寄存器

sel_alu_src2[1]：指 reg2 中的立即数寄存器，从零读（有符号拓展到 32 位）

sel_alu_src2[2]：指 reg2 中的立即数寄存器，按位读取（32 位立即数）

sel_alu_src2[3]：指 reg2 中的判断移位寄存器（无符号拓展到 32 位）

生成三位的寄存器一和四位的寄存器二。

```
// rs to reg1 指令不一定显示是rs，可能是base寄存器，但对应rs寄存器
assign sel_alu_src1[0] = inst_ori | inst_addiu | inst_subu | inst_addu | inst_or | inst_lw | inst_sw | inst_xor
    | inst_and | inst_sltu | inst_slt | inst_slti | inst_sltiu | inst_srlv | inst_beq | inst_bne | inst_bgtz|inst_bltz
    | inst_add | inst_jr | inst_addi | inst_sub|inst_andi| inst_nor|inst_xori|inst_sllv| inst_srav|inst_bgez |inst_blez
    | inst_div | inst_divu | inst_mult | inst_multu
    | inst_mthi| inst_mtlo | inst_sh   | inst_sb   | inst_lhu
    | inst_lh | inst_lb | inst_lbu ;

// pc to reg1
assign sel_alu_src1[1] =inst_jal| inst_j|inst_bltzal|inst_bgezal|inst_jalr ;

// sa_zero_extend to reg1
assign sel_alu_src1[2] = inst_sll | inst_srl|inst_sra;
```

reg1 中存放的数据类型分为三类 rs 寄存器，pc 寄存器和 sa 指令寄存器。

```
// rt to reg2 不是看到rt就加在这里，有时rt是要存的，而不是读取的，不能加在这里
assign sel_alu_src2[0] = inst_subu |inst_addu | inst_sll | inst_or | inst_xor | inst_and | inst_sltu
    | inst_slt | inst_srlv | inst_srl | inst_beq | inst_bne | inst_add | inst_sub|inst_nor|inst_sllv |inst_sra| inst_srav
    | inst_div | inst_divu| inst_mult| inst_multu
    |inst_sh | inst_sb | inst_lhu | inst_lh | inst_lb | inst_lbu ;
// 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_sh | inst_sb | inst_lhu | inst_lh | inst_lbu|inst_lb;
// 32'b8 to reg2
assign sel_alu_src2[2] = inst_jal|inst_j|inst_bltzal|inst_bgezal|inst_jalr  ;
// imm_zero_extend to reg2
assign sel_alu_src2[3] = inst_ori|inst_andi|inst_xori;
```

reg2 中存放的数据类型分为四类 rt 寄存器，立即数寄存器（从零读取，按位读取），判断移位寄存器。

操作运算符的判断：

定义 12 种运算符操作

```
assign alu_op = {op_add, op_sub, op_slt, op_sltu,
                 op_and, op_nor, op_or, op_xor,
                 op_sll, op_srl, op_sra, op_lui};
```

当指令需要某条运算符的时候，即给该运算符赋值为 1。

op_add，加法运算

op_sub，减法运算

op_slt，有符号小于比较

op_sltu，无符号小于比较

op_and，逻辑与运算

op_nor，逻辑或非运算

op_or，逻辑或运算

op_xor，逻辑异或运算

op_sll，逻辑左移

op_srl，逻辑右移

op_sra，算数右移

op_lui，寄存器高半部分置立即数

访存模块：

定义：

data_ram_en：数据加载，存储使能信号

data_ram_wen：写入存储器使能信号

```
assign data_ram_readen =     inst_lw    ? 4'b1111
                            :inst_lb    ? 4'b0001
                            :inst_lbu ? 4'b0010
```

```
:inst_lh    ? 4'b0011
:inst_lhu ? 4'b0100
:inst_sb    ? 4'b0101
:inst_sh    ? 4'b0111
:4'b0000;
```

写入寄存器数据模块：

assign rf_we = inst_ori | inst_lui | inst_addiu | inst_subu | inst_jal |inst_addu | inst_sll | inst_or | inst_lw.....

定义 rf_we：表示 regfile 模块存储使能信号

定义 wire sel_rf_dst[2:0]：标记存储在哪个寄存器中

sel_rf_dst[0]：存入 rd 寄存器中

sel_rf_dst[1]：存入 rt 寄存器中

sel_rf_dst[2]：存入 31 号通用寄存器中

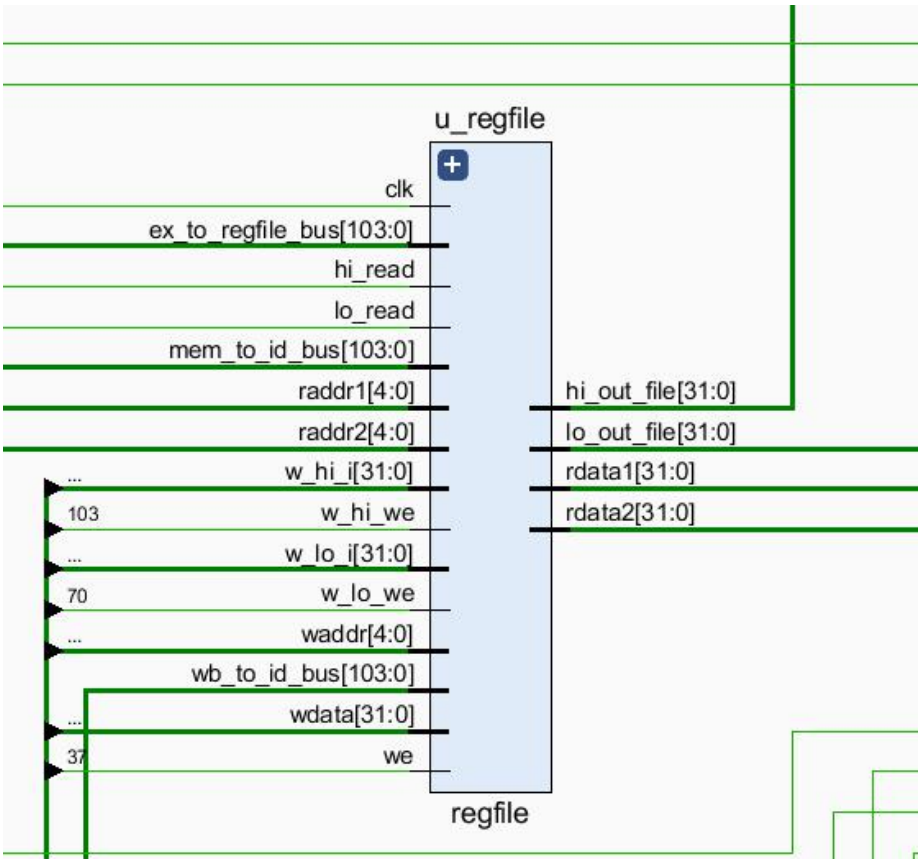
跳转指令模块：

定义：

br_e：为跳转信号，不同的跳转指令判断条件不同。

br_addr：为跳转地址，不同的跳转指令所跳转的地址可能需要地址运算。才可以得到 br_addr

（3）Regfile 模块：



regfile 结构示意图

接口介绍

序号	接口名	宽度	输入/输出	作用
1	clk	1	输入	时钟信号
2	we	1	输入	写使能信号
3	waddr	5	输入	要写入的寄存器地址
4	wdata	32	输入	要写入的数据
5	raddr1	5	输入	第一个读寄存器端口要读取的寄存器的地址
6	raddr2	5	输入	第二个读寄存器端口要读取的寄存器的地址
7	ex_to_regfile_bus	104	输入	EX 段输入到 regfile 总线
8	mem_to_id_bus	104	输入	MEM 段输入到 regfile 总线
9	wb_to_id_bus	104	输入	WB 段输入到 regfile 总线
10	hi_read	1	输入	hi 寄存器读信号

11	lo_read	1	输入	lo 寄存器读信号
12	w_hi_we	1	输入	Hi 寄存器写信号
13	w_lo_we	1	输入	Lo 寄存器写信号
14	w_hi_i	32	输入	Hi 寄存器写数据
15	w_lo_i	32	输入	Lo 寄存器写数据
16	hi_out_file	32	输出	Hi 寄存器读数据
17	lo_out_file	32	输出	Lo 寄存器读数据
18	rdata1	32	输出	从 regfile 寄存器 1 中读取数据
19	rdata2	32	输出	从 regfile 寄存器 2 中读取数据

信号介绍

序号	线	宽度	输入/输出	作用
7	ex_to_regfile_bus	104	输入	ex 段数据传入 regfile 段

ex_w_hi_we:EX 段传输到 regfile 段的 hi 寄存器写信号。

ex_w_hi_i：EX 段传输到 regfile 段的 hi 寄存器数据。

ex_w_lo_we：EX 段传输到 regfile 段的 lo 寄存器写信号。

ex_w_lo_i：EX 段传输到 regfile 段的 lo 寄存器写数据。

ex_to_id_we:EX 段传输到 regfile 段的要写入的寄存器的使能信号。

ex_to_id_waddr:EX 段传输到 regfile 段的要写入的寄存器地址。

ex_result:EX 段传输到 regfile 段的要写入的寄存器数据。

序号	线	宽度	输入/输出	作用
8	mem_to_id_bus	104	输入	mem 段数据传入 regfile 段

mem_w_hi_we:MEM 段传输到 regfile 段的 hi 寄存器写信号。

mem_w_hi_i：MEM 段传输到 regfile 段的 hi 寄存器数据。

mem_w_lo_we：MEM 段传输到 regfile 段的 lo 寄存器写信号。

mem_w_lo_i：MEM 段传输到 regfile 段的 lo 寄存器写数据。

mem_to_id_we:MEM 段传输到 regfile 段的要写入的寄存器的使能信号。

mem_to_id_waddr:MEM 段传输到 regfile 段的要写入的寄存器地址。

mem_result：MEM 段传输到 regfile 段的要写入的寄存器数据。

序号	线	宽度	输入/输出	作用
9	wb_to_id_bus	104	输入	wb 段数据传入 regfile 段

wb_w_hi_we:WB 段传输到 regfile 段的 hi 寄存器写信号。

wb_w_hi_i：WB 段传输到 regfile 段的 hi 寄存器数据。

wb_w_lo_we：WB 段传输到 regfile 段的 lo 寄存器写信号。

wb_w_lo_i：WB 段传输到 regfile 段的 lo 寄存器写数据。

wb_to_id_we:WB 段传输到 regfile 段的要写入的寄存器的使能信号。

wb_to_id_waddr:WB 段传输到 regfile 段的要写入的寄存器地址。

wb_result：WB 段传输到 regfile 段的要写入的寄存器数据。

模块说明：

Regfile 模块有两个读寄存器和一个写寄存器。

写寄存器：

定义 32 个三十二位寄存器 reg[31:0] reg_array[31:0]

```
    // write
    always @ (posedge clk) begin
        if (we && waddr!=5'b0) begin
            reg_array[waddr] <= wdata;
        end
    end
```

如果 we 和 waddr 都不是 0，则将要写入的数据写入三十二位寄存器。

读寄存器 1：

```
assign rdata1 = (raddr1 == 5'b0) ? 32'b0 :
                    ((raddr1==  ex_to_id_waddr)  &  ex_to_id_we)  ?
ex_result :
                    ((raddr1==  mem_to_id_waddr)  &  mem_to_id_we)  ?
mem_result :
                    ((raddr1==  wb_to_id_waddr)  &  wb_to_id_we)  ?
wb_to_id_result : reg_array[raddr1];
```

处理数据相关：

当读寄存器的地址为 0 则，数据为 0。否则判断读取的地址是否和前面流水线某一段地址相关，如果相关则读入该段数据。如果都不是则读取改地址对应寄存器中的数据。

读寄存器 2 同理。

hilo 寄存器模块：

```
reg [31:0] reg_array_hi;
reg [31:0] reg_array_lo;
```

定义两个寄存器 hi 和 lo

写寄存器：

```
always @ (posedge clk) begin
    if (w_hi_we) begin
        reg_array_hi <= w_hi_i;
    end
end
always @ (posedge clk) begin
    if (w_lo_we) begin
        reg_array_lo <= w_lo_i;
    end
end
```

如果 regfile 模块得到 hi 或 lo 写信号，则将数据写入寄存器中

读寄存器

```
// read hi
assign hi_out_file = (ex_w_hi_we) ? ex_w_hi_i:
                     (mem_w_hi_we) ? mem_w_hi_i:
                     (wb_w_hi_we) ? wb_w_hi_i :
                     reg_array_hi;
//read lo
assign lo_out_file = (ex_w_lo_we) ? ex_w_lo_i:
                     (mem_w_lo_we) ? mem_w_lo_i:
                     (wb_w_lo_we) ? wb_w_lo_i :
                     reg_array_lo;
```

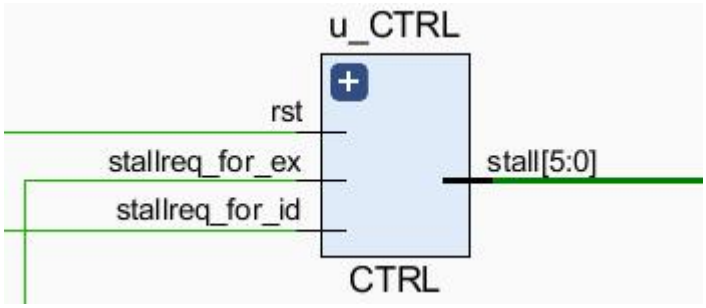
处理数据相关：如果要读的 hilo 寄存器数据在上一段流水线的处理过程中，可以通过判断上一段流水线传来的写信号，哪一段写信号被激活，则直接读取对应段的数据。

4.3、CTRL 段和多段交互

4.3.1 访存指令与暂停机制

ID 段设计一个重要的暂停机制就是读写冲突相关的暂停机制。看一下这段代码。

```
reg [31:0] inst_reg;
reg inst_reg_en;
always @ (posedge clk) begin
    inst_reg_en <= 1'b0;
    if (rst) begin
        if_to_id_bus_r <= `IF_TO_ID_WD'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;
    end
    else if (stall[1]==`NoStop) begin
        if_to_id_bus_r <= if_to_id_bus;
    end
end
```



CTRL

这边引用一下 CTRL 段的代码：

```
always @ (*) begin
    if (rst) begin
        stall = `StallBus'b0;
    end else if(stallreq_for_id == `Stop) begin
        stall = 6'b000_111;
    end else if(stallreq_for_ex == `Stop) begin
        stall = 6'b001_111;
    end
    else begin
        stall = `StallBus'b0;
    end
```

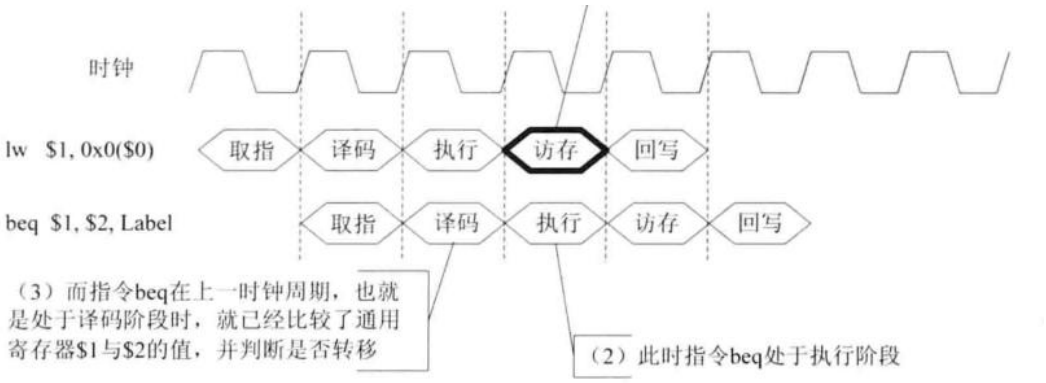
end

可以看出，当 CTRL 接收到 ID 段的 stallreq_for_id 时，就会发出 000111 的信号，正对应了上述 ID 段的代码 stall[1]==`Stop && stall[2]==`NoStop 的时候就会向 EX 段输出空信号。按照前面所说的，如果 stall 为这样的信号的话，IF 段，就会持续输出相同的 PC，而这时 ID 段会向 EX 段输出空信号，EX 段及其后面的段不受 stall 影响，会继续工作。而也正因为 EX 段接受 ID 段的空信号，EX 段不会做出新的动作，而是把当前工作传递后一个段而停止工作，或者是在持续完成 EX 段未完成的工作。

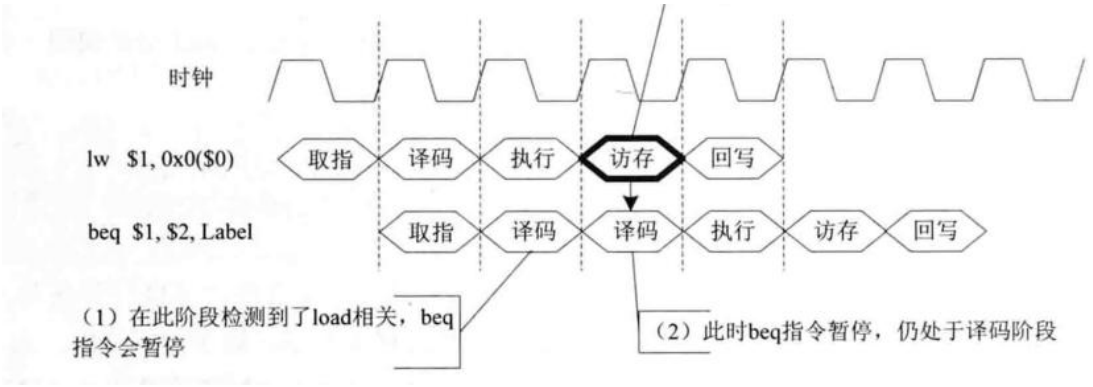
那什么时候 ID 段需要发出一个暂停申请呢？请看如下指令。

```
lw   $1, 0x0($0)     // 从数据存储器的地址 0x0 处加载字，保存到通用寄存器$1
beq  $1, $2, Label   // 比较通用寄存器$1 与$2，如果相等，那么转移到 Label 处
```

正常情况下，上一条指令在 EX 段完成相关操作后，得到的数据通过数据相关的连线直接给当前的指令使用，而 lw 指令不同，它需要在 MEM 段才能得到真正所需要的数据，比 EX 段晚了一个周期。加载指令 lw 会在访存阶段从数据存储器读取数据，也就是在访存阶段才能获得要写入通用寄存器 S1 的值，这个值是 S1 的最新值，此时紧接着的转移指令 beq 处于执行阶段，而 beq 在上一周期译码阶段时，就已经对寄存器 S1 与 S2 的值进行了比较，并判断是否转移，显然这个判断依据的寄存器 S1 的值不是 lw 指令加载得到的值，所以程序并没有按照意图运行。如下图所示：



我们需要做的在译码阶段检查当前指令与上一条指令是否存在 load 相关，如果存在 load 相关，那么就让流水线的译码、取指阶段暂停，而执行、访存、回写阶段继续，相当于插入一个空指令，即加气泡操作，这样处于执行阶段的加载指令会继续运行，不受影响，当其运行到访存阶段时，将加载得到的数据前推到译码阶段，然后，流水线可以继续运行。按如下图所示：



那如何判断上一条指令与当前指令是否存在 load 相关冲突呢？先来看一下 ID 段对 load 类指令的特殊处理。

```
//和访存有关
// load and store enable
assign data_ram_en = inst_lw | inst_sw | inst_sh | inst_sb | inst_lhu | inst_lh | inst_lbu |inst_lb;
// write enable  写存储器则传入 1111，不是则是 0000，调换顺序会报错
assign data_ram_wen = inst_sw ? 4'b1111:4'b0000;
```

这两根线都会传到 EX 段，那么只需要在 EX 段把这两根线回传，然后判断 data_ram_en 是不是 1，且 data_ram_wen 是不是 0000，就可以知道上一条指令是否为 load 指令了。顺带一提，如果是 store 相关指令，则需要把 data_ram_wen 设为 1111，但 store 类指令不涉及冲突操作。到了上一条指令和加载内存有关。我需要判断一下上一条指令，它所要存储的寄存器。跟当前指令所需要访问的寄存器是否是同一个地址。如果，那么就存在一个读写相关，即 load 相关。

查看如下代码

```
//判断是否加气泡
//要读取的 reg1 是否与上一条指令存在 load 相关
wire stallreq_for_reg1_dataRelate;
//要读取的 reg2 是否与上一条指令存在 load 相关
wire stallreq_for_reg2_dataRelate;
//上一条指令是否为 load
wire pre_inst_is_load;
assign pre_inst_is_load = pre_inst_data_sram_en & (pre_inst_data_sram_wen == 4'b0000);
assign stallreq_for_reg1_loadRelate = (rs == pre_ex_to_id_waddr) & sel_alu_src1[0];
assign stallreq_for_reg2_loadRelate = (rt == pre_ex_to_id_waddr) & sel_alu_src2[0];
assign stallreq_for_id = (stallreq_for_reg1_loadRelate | stallreq_for_reg2_loadRelate) &
pre_inst_is_load;
```

EX 段除了传回上面两根线之外，还需要传回 EX 段所需要访问的。寄存器的地址，如上面代码是比较上一条指令所需要写入的寄存器的地址。跟当前指令所需要读的寄存器的地址是否一致。如果有其中一个是一样的，并且上一条指令确定是 load 类指令，那么的话，就需要发出一个加气泡申请。

不加气泡申请之后，CTRL 接收到来自 ID 段的 stallreq_for_id，就会发出 000111，

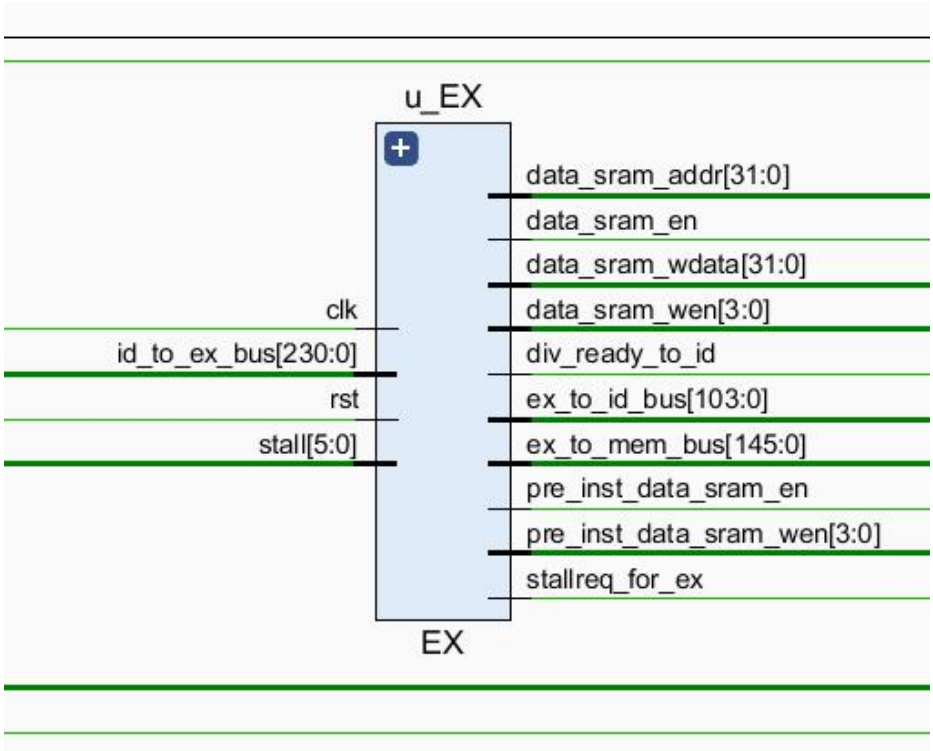
IF 段 PC 不变，ID 段向 EX 段输出空信号，EX 段的 load 指令运行完后数据到了 MEM 工作时 EX 段会接收空信号，不工作。而此时 EX 段接收空信号，传回的值也不再是上面的值，也就不存在 load 相关了，ID 段的 stallreq_for_id 为 0，CTRL 发出 000000，流水线恢复工作，而此时的 ID 段便可以通过数据相关接收到来自 MEM 的正确数据了。

需要注意的是，尽管 IF 段的 PC 维持不变，但 ID 段的有一个变量还是会产生改变，即 inst。Inst 代表着当前 ID 段的指令，为了保持 ID 段的 inst 和 PC 同步不变。需要一个寄存器暂时把它存起来。

这里定义两个寄存器，reg [31:0] inst_reg，reg inst_reg_en;分别用于存储指令和表示当前的 inst_reg 是否有存储指令。如果检测到存在读写相关的话，那么就得及时把当前的指定存入前一个寄存器。并且把后一个寄存器设为一。而如果没有检测到，则持续为两个寄存器赋 0 值。

在流水线恢复正常工作时，只需要判断 inst_reg_en 是否为 1。如果是，则使用 inst_reg 里的指令，不是，则正常使用 inst_sram_rdata 即可。

4.4、EX 段



4.4.1 接口名称

序号	接口名	宽度	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	id_to_ex_bus	231	输入	ID 段输入到 EX 段总线
4	stall	6	输入	控制指令
5	data_sram_addr	32	输出	
6	data_sram_en	1	输出	
7	data_sram_wdata	32	输出	
8	data_sram_wen	4	输出	
9	div_ready_to_id	1	输出	判断除法是否结束
10	ex_to_id_bus	104	输出	EX 段写入 ID 段
11	ex_to_mem_bus	146	输出	EX 段写入 MEM 段
12	pre_inst_data_sram_en	1	输出	
13	pre_inst_data_sram_wen	4	输出	
14	stallreq_for_ex	1	输出	

序号	线	宽度	输入/输出	作用
3	id_to_ex_bus	231	输入	ID 数据传入 EX 段

信号包含：

data_ram_readen

hi_read：hi 寄存器读信号

lo_read:lo 寄存器读信号

hi_write:hi 寄存器写信号

lo_write:lo 寄存器写信号

hi_out_file:hi 寄存器读数据

lo_out_file：lo 寄存器读数据

ex_pc:PC 值

inst:32 位指令

alu_op：运算符

sel_alu_src1：运算寄存器 1 访问

sel_alu_src2 运算寄存器 2 访问

data_ram_en：访存使能信号

data_ram_wen:

rf_we：读数据使能信号

rf_rdata1:regfile 读取数据 1

rf_rdata2:regfile 读取数据 2

序号	线	宽度	输入/输出	作用
10	ex_to_id_bus	104	输入	EX 数据传入 MEM 段

w_hi_we：hi 寄存器写信号

w_hi_i ：hi 寄存器数据

w_lo_we：lo 寄存器写信号

w_lo_i:lo 寄存器数据

rf_we：使能信号

rf_waddr ：运算结果所在地址

ex_result：运算得到的结果

序号	线	宽度	输入/输出	作用
10	ex_to_mem_bus	146	输入	EX 数据传入 MEM 段

data_ram_readen：设计访存指令类型的判断

w_hi_we:hi 寄存器写信号

w_hi_i:hi 寄存器写数据

w_lo_we：lo 寄存器写信号

w_lo_i：lo 寄存器写数据

ex_pc:PC 值

data_ram_en：

data_ram_wen：

sel_rf_res：

rf_we：使能信号

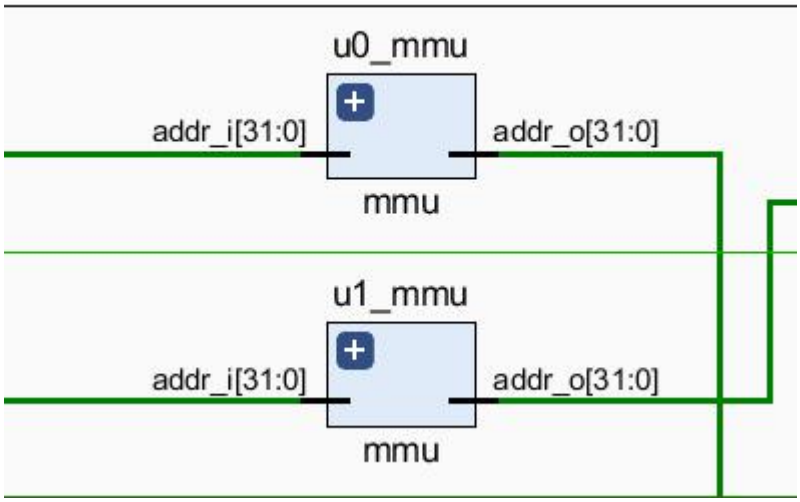
rf_waddr：运算结果地址

ex_result：运算结果

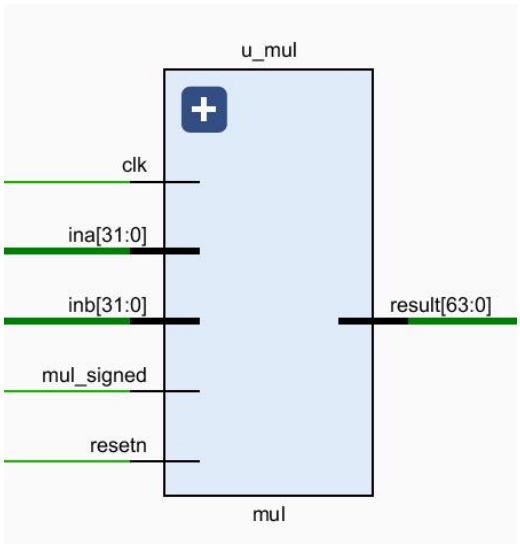
4.4.2 模块说明

乘法器，除法器

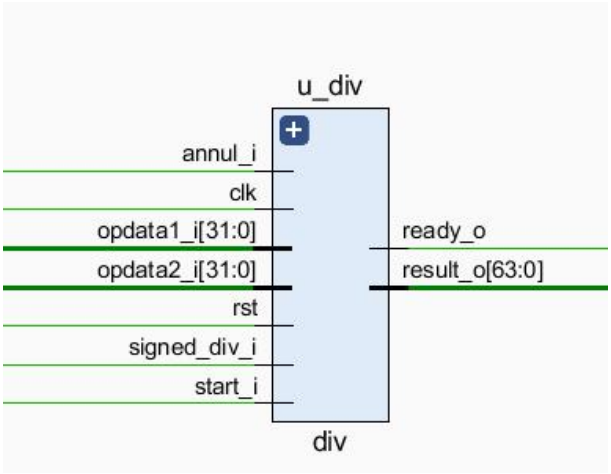
MMU



MUL



DIV



首先是使用给定 mul 模块：

```
wire [63:0] mul_result;
wire mul_signed; // 有符号乘法标记
wire [31:0] mul_src1;
wire [31:0] mul_src2;
assign inst_mult = (inst[31:26] == 6'b00_0000) & (inst[15:6] == 10'b00000_00000) & (inst[5:0] == 6'b01_1000);
assign inst_multu = (inst[31:26] == 6'b00_0000) & (inst[15:6] == 10'b00000_00000) & (inst[5:0] == 6'b01_1001);

assign mul_signed = inst_mult;//判断有符号/无符号乘法
assign mul_src1 =(inst_mult | inst_multu) ? rf_rdata1 : 32'd0;
assign mul_src2 =(inst_mult | inst_multu) ? rf_rdata2 : 32'd0;

mul u_mul(
    .clk            (clk            ),
    .resetn         (~rst           ),
    .mul_signed     (mul_signed     ),
    .ina            (mul_src1       ), // 乘法源操作数 1
    .inb            (mul_src2       ), // 乘法源操作数 2
    .result         (mul_result     ) // 乘法结果 64bit
);
```

定义 inst_mult 和 inst_multu 指令用来判断是否使用乘法模块。

定义 mul_signed 判断有符号/无符号乘法

给 mul_src1 和 mul_src2 根据是否为乘法指令判断条件，赋予 rs/rt 值。若不是乘

法指令，置 0。

同理给定 div 模块

```
// DIV part
wire [63:0] div_result;
wire inst_div, inst_divu;
wire div_ready_i;
reg stallreq_for_div;
assign stallreq_for_ex = stallreq_for_div  &&(div_ready_i==1'b0);
```

定义 div_result 除法运算结果和 inst_div、inst_divu 除法指令。由于除法需要进行 32 个周期，所以需要定义 div_ready_i 使能信号和 stallreq_for_div，判断除法是否完成。根据使能信号为 1 和 stallreq_for_div 表示除法结束，0 和 stallreq_for_div 表示仍在进行。

```
//WAIT
assign div_ready_to_id = div_ready_i;
assign inst_div = (inst[31:26] == 6'b00_0000) & (inst[15:6] == 10'b00000_00000) & (inst[5:0] == 6'b01_1010);
assign inst_divu = (inst[31:26] == 6'b00_0000) & (inst[15:6] == 10'b00000_00000) & (inst[5:0] == 6'b01_1011);

reg [31:0] div_opdata1_o;
reg [31:0] div_opdata2_o;
reg div_start_o;
reg signed_div_o;

//div part
div u_div(
    .rst          (rst          ),
    .clk          (clk          ),
    .signed_div_i (signed_div_o ),
    .opdata1_i    (div_opdata1_o    ),
    .opdata2_i    (div_opdata2_o    ),
    .start_i      (div_start_o      ),
    .annul_i      (1'b0         ),
```

```
        .result_o       (div_result       ), //  除法结果  64bit
        .ready_o        (div_ready_i      )
    );
```

div_opdata1_o 和 div_opdata2_o 分别为被除数，除数，div_result 为 div 模块返回的除法结果。

32 位移位乘法器

该乘法器为自己实现的 32 周期移位乘法器，使用方式和项目中的 div.v 试商法除法器使用类似。如下面代码所示，乘法器几个端口的功能，跟它的注释是一样的。

```
    input wire rst,                         //复位
    input wire clk,                         //时钟
    input wire signed_mul_i,                //是否为有符号乘法运算，输入 1'b1 表示有符号
    input wire[31:0] opdata1_i,             //乘数 1
    input wire[31:0] opdata2_i,             //乘数 2
    input wire start_i,                     //是否开始乘法运算
    input wire annul_i,                     //是否取消乘法运算，输入 1'b1 表示取消
    output reg[63:0] result_o,              //乘法运算结果
    output reg ready_o                      //乘法运算是否结束
```

被乘数和乘数输入均为 32 位数。而输出结果的为 64 位。就是因为 32 位数相乘，其结果最大不超过 64 位。

在乘法器为空闲状态的时候，即 MulFree 状态下，若输入为 start_i == 1’b1 && annul_i == 1'b0，则乘法器开始工作，输出结果的状态为 MulResultNotReady，表示乘法器的，结果还没运算好，EX 段不能向后输出数据。

此时乘法器会自先判断两个乘数是否有一个为 0，如果时，则进入 MulByZero，重置部分储存器后直接跳转到乘法器结束状态。并且输出结果零。

若两个乘数都不是 0，乘法器会进入初始化工作，此时状态为 MulOn。并且判断两个乘数是否为有符号数。如果是有符号数，会判断它是否为负数，如果是负数，就会补码进行运算。

乘法器的主要变量如下：

```
    reg [5:0] cnt;                          //记录乘法进行了几轮
    reg [63:0] product;                     //移位累加的结果
    reg [1:0] state;                        //乘法器处于的状态
    reg[63:0] multiplicand;                 //被乘数，初始值为 temp_op1，有
```

移位操作

```
reg[31:0] temp_op1;                    //处理后的被乘数
reg[31:0] temp_op2;                    //处理后的乘数
```

处理后的被乘数和乘数指的是如果它是有符号数会进行一个补码运算。

乘法器的原理大致如下：

这里两个移位操作，一个是被乘数的左移，一个是乘数的右移。二进制的乘法和十进制的乘法原理一样，都是按位进行相乘，然后累加。二进制数只有一和零，所以如果、对应的位是 1，这时候把被乘数照抄下来，如果是 0，则抄写却为零。代码中 product 变量记录的是每次移位累加的结果，multiplicand 初始值为被乘数，每次和 product 进行一次加法操作后，都需要经过一次左移操作。而 temp_op2 作为乘数，每次 multiplicand 和 product 进行一次加法操作后，都需要右移一位。而上述所说的按位乘和加，这里参考的位是 temp_op2 最低位。弄完一个位后，都需要对它进行一个右移操作来更新，最低位。而且因为更新后的最低位实际上在原来的位中是高了一位。所以被乘数也需要左移一位。

而 EX 段对乘法器的链接和操作代码如下：

```
mul u_mul(
    .rst                (rst                ),
    .clk                (clk                ),
    .signed_mul_i (signed_mul_o ),
    .opdata1_i      (mul_opdata1_o      ),
    .opdata2_i      (mul_opdata2_o      ),
    .start_i          (mul_start_o          ),
    .annul_i          (1'b0          ),
    .result_o       (mul_result       ), //  除法结果  64bit
    .ready_o         (mul_ready_i          )
);

always @ (*) begin
    if (rst) begin
        stallreq_for_mul = `NoStop;
        mul_opdata1_o = `ZeroWord;
        mul_opdata2_o = `ZeroWord;
        mul_start_o = `MulStop;
        signed_mul_o = 1'b0;
    end
    else begin
        stallreq_for_mul = `NoStop;
        mul_opdata1_o = `ZeroWord;
        mul_opdata2_o = `ZeroWord;
        mul_start_o = `MulStop;
        signed_mul_o = 1'b0;
```

```
case ({inst_mult,inst_multu})
    2'b10:begin
        if (mul_ready_i == `MulResultNotReady) begin
            mul_opdata1_o = rf_rdata1;
            mul_opdata2_o = rf_rdata2;
            mul_start_o = `MulStart;
            signed_mul_o = 1'b1;
            stallreq_for_mul = `Stop;
        end
        else if (mul_ready_i == `MulResultReady) begin
            mul_opdata1_o = rf_rdata1;
            mul_opdata2_o = rf_rdata2;
            mul_start_o = `MulStop;
            signed_mul_o = 1'b1;
            stallreq_for_mul = `NoStop;
        end
        else begin
            mul_opdata1_o = `ZeroWord;
            mul_opdata2_o = `ZeroWord;
            mul_start_o = `MulStop;
            signed_mul_o = 1'b0;
            stallreq_for_mul = `NoStop;
        end
    end
    2'b01:begin
        if (mul_ready_i == `MulResultNotReady) begin
            mul_opdata1_o = rf_rdata1;
            mul_opdata2_o = rf_rdata2;
            mul_start_o = `MulStart;
            signed_mul_o = 1'b0;
            stallreq_for_mul = `Stop;
        end
        else if (mul_ready_i == `MulResultReady) begin
            mul_opdata1_o = rf_rdata1;
            mul_opdata2_o = rf_rdata2;
            mul_start_o = `MulStop;
            signed_mul_o = 1'b0;
            stallreq_for_mul = `NoStop;
        end
        else begin
            mul_opdata1_o = `ZeroWord;
            mul_opdata2_o = `ZeroWord;
            mul_start_o = `MulStop;
            signed_mul_o = 1'b0;
```

```
                    stallreq_for_mul = `NoStop;
                end
            end
            default:begin
            end
        endcase
    end
end
```

这里用 case 来判断当前的乘法，指定是否为有符号数乘法，来为连接到乘法器的负责判断是否为有符号数的寄存器 signed_mul_o 赋值。这里最重要的是判断是否为 MulResultNotReady，如果是，表示乘法器在工作，运算还没结束。这时候需要一直给乘法器的被乘数和乘数，赋上对应的寄存器的值。而且只要运算没结束，来自乘法器的加气泡申请 stallreq_for_mul 就需要一直为 Stop 状态，等到运算结束后，需要及时更改它为 NoStop，让流水线继续运行。

经过 div/mul 运算和对 hilo 寄存器的指令操作，其读写代码如下：

```
assign w_hi_we = inst_mult | inst_multu | inst_div | inst_divu | inst_mthi;
assign w_lo_we = inst_mult | inst_multu | inst_div | inst_divu | inst_mtlo;
assign w_hi_i = (inst_mult | inst_multu) ?    mul_result[63:32] :
                        (inst_div | inst_divu ) ? div_result[63:32] :
                        (hi_write) ? rf_rdata1:32'b0;
assign w_lo_i = (inst_mult | inst_multu) ?    mul_result[31:0] :
                        (inst_div | inst_divu ) ? div_result[31:0] :
                        (lo_write) ? rf_rdata1 : 32'b0;
```

如果是除法运算，hi 寄存器存储 div_result 的高 32 位，lo 寄存器存储 div_result 的低 32 位。

如果是乘法运算，hi 寄存器存储 mul_result 的高 32 位，lo 寄存器存储 mul_result 的低 32 位。

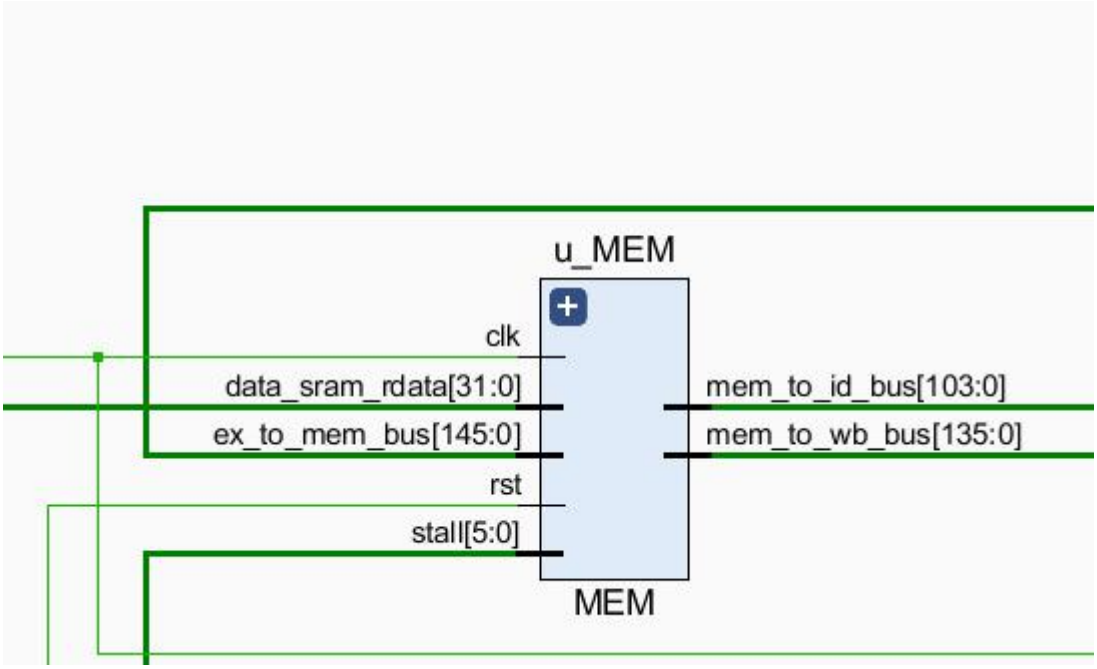
如果是写操作，hi 寄存器或 lo 寄存器根据写信号判定是否写入 rf_rdata1。

如果条件都不满足，hi 和 lo 的写数据都置为 0。

MEM

操作：存储器访问操作，分支操作

4.5MEM 段



4.5.1 接口介绍

序号	接口名	宽度	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	data_sram_rdata	32	输入	
4	ex_to_mem_bus	146	输入	EX 段传入 MEM 段数据
5	stall	6	输入	
6	mem_to_id_bus	104	输出	MEM 段传入 ID 段数据
7	mem_to_wb_bus	136	输出	MEM 段传入 WB 段数据

信号介绍

序号	线	宽度	输入/输出	作用
4	ex_to_mem_bus	146	输入	ex 数据传入 mem 段

信号包含

data_ram_readen：设计访存指令类型的判断

w_hi_we:hi 寄存器写信号

w_hi_i:hi 寄存器写数据

w_lo_we：lo 寄存器写信号

w_lo_i：lo 寄存器写数据

ex_pc:PC 值

data_ram_en：

data_ram_wen：

sel_rf_res：

rf_we：使能信号

rf_waddr：运算结果地址

ex_result：运算结果

序号	线	宽度	输入/输出	作用
6	mem_to_id_bus	104	输出	mem 数据传入 id 段

信号包含

w_hi_we:hi 写 hi 寄存器使能信号

w_hi_i:hi 寄存器写数据

w_lo_we：写 lo 寄存器使能信号

w_lo_i：lo 寄存器写数据

rf_we:

rf_waddr

rf_wdata

序号	线	宽度	输入/输出	作用
7	mem_to_wb_bus	136	输出	mem 数据传入 wb 段

信号包含

w_hi_we

w_hi_i

w_lo_we

w_lo_i

mem_pc

rf_we

rf_waddr

rf_wdata

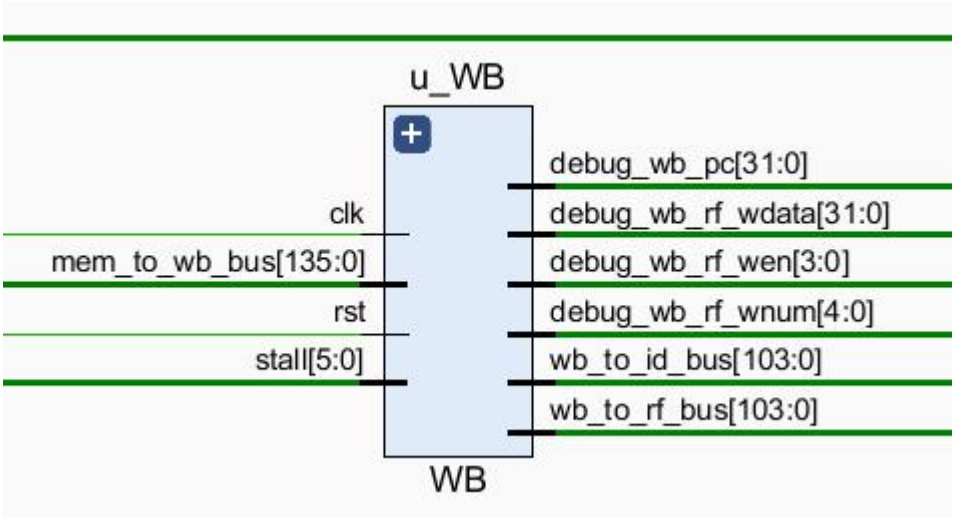
4.5.2 模块功能

rf_wdata 根据 data_ram_readen 进行和 data_ram_en 的情况，完成相关数据写入内存。判断代码如下：

```
assign rf_wdata = (data_ram_readen==4'b1111 && data_ram_en==1'b1) ? data_sram_rdata
                : (data_ram_readen==4'b0001 && data_ram_en==1'b1 && ex_result[1:0]==2'b00) ?({{24{data_sram_rdata[7]}},data_sram_rdata[7:0]})
                : (data_ram_readen==4'b0001 && data_ram_en==1'b1 && ex_result[1:0]==2'b01) ?({{24{data_sram_rdata[15]}},data_sram_rdata[15:8]})
                : (data_ram_readen==4'b0001 && data_ram_en==1'b1 && ex_result[1:0]==2'b10) ?({{24{data_sram_rdata[23]}},data_sram_rdata[23:16]})
                : (data_ram_readen==4'b0001 && data_ram_en==1'b1 && ex_result[1:0]==2'b11) ?({{24{data_sram_rdata[31]}},data_sram_rdata[31:24]})
                : (data_ram_readen==4'b0010 && data_ram_en==1'b1 && ex_result[1:0]==2'b00) ?({24'b0,data_sram_rdata[7:0]})
                : (data_ram_readen==4'b0010 && data_ram_en==1'b1 && ex_result[1:0]==2'b01) ?({24'b0,data_sram_rdata[15:8]})
                : (data_ram_readen==4'b0010 && data_ram_en==1'b1 && ex_result[1:0]==2'b10) ?({24'b0,data_sram_rdata[23:16]})
                : (data_ram_readen==4'b0010 && data_ram_en==1'b1 && ex_result[1:0]==2'b11) ?({24'b0,data_sram_rdata[31:24]})
                : (data_ram_readen==4'b0011 && data_ram_en==1'b1 && ex_result[1:0]==2'b00) ?({{16{data_sram_rdata[15]}},data_sram_rdata[15:0]})
                : (data_ram_readen==4'b0011 && data_ram_en==1'b1 && ex_result[1:0]==2'b10) ?({{16{data_sram_rdata[31]}},data_sram_rdata[31:16]})
                : (data_ram_readen==4'b0100 && data_ram_en==1'b1 && ex_result[1:0]==2'b00) ?({16'b0,data_sram_rdata[15:0]})
                : (data_ram_readen==4'b0100 && data_ram_en==1'b1 && ex_result[1:0]==2'b10) ?({16'b0,data_sram_rdata[31:16]})
                : ex_result;
```

4.6、WB 段

操作：将寄存器—寄存器 ALU 指令，寄存器—立即型 ALU 指令，LOAD 指令结果写入寄存器中，由操作码决定送入寄存器中。



序号	接口名	宽度	输入/输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	mem_to_wb_bus	136	输入	mem 数据写入 wb 段
4	stall	6	输入	
5	wb_to_id_bus	104	输出	Wb 段数据开端写入 ID 段
6	wb_to_rf_bus	104	输出	Wb 段数据结束回写 regfile 段
7	Debug_wb_pc	32	输出	
8	Debug_wb_rf_wdata	32	输出	
9	Debug_wb_rf_wen	4	输出	
10	Debug_wb_rf_wnum	5	输出	

信号介绍

序号	线	宽度	输入/输出	作用
3	Mem_to_wb_bus	136	输入	Wb 段数据开端写入 ID 段

信号包含：

w_hi_we,

w_hi_i,

w_lo_we,

w_lo_i,

mem_pc,

rf_we,

rf_waddr,

rf_wdata

序号	线	宽度	输入/输出	作用
5	Wb_to_id_bus	104	输出	wb段开始直接写入id段中的 regfile 模块

信号包含

w_hi_we,

w_hi_i,

w_lo_we,

w_lo_i,

rf_we,

rf_waddr

rf_wdata

序号	线	宽度	输入/输出	作用
6	Wb_to_rf_bus	104	输出	wb 数据回写进入 ID 段

五、组员感受和改进意见

汤博皓：

在这次实验中，我们组进行五级流水线的指令添加，完成数据相关处理，气泡和乘除法器等功能。在编写代码初期，给我的感受就是读不懂，无从下手。经过和组员的讨论，不断地修改 bug。我对五级流水线中的代码越来越了解。在 EX,MEM,WB 段接线的过程中，我对数据相关也有了更深入领悟。而在乘除法器中，在编写 hilo 寄存器的过程中，我遇到了除法 32 个周期时间停止错位等现象。为了找到问题并解决，我掌握了如何看波形图调 bug。总之通过这次实验，能看到自己对计算机系统知识的把握有肉眼的提升，不过还是有很多知识点需要去学习。

陈乐奇：

这次的实验工作量比较大，好在比较顺利地完成了。在计算机系统这门课课中，我感觉比较难的就是五级流水线结构，还有五级流水线之间的数据相关，以及插气泡机制。尽管老师在讲这一部分的时候仔细给我们讲得很仔细，但我们还是听得不太懂。而这次实验给我们提供了一个很好的机会。去实践，了解真正的五级流水线是怎么工作的，并且学会自己加指令加数据相关，加气泡，还有自己实现一个乘除法器，也带领我们走入了硬件的世界，体会到不一样的魅力。虽然过程很累，但抽象的课本知识通过实验变成了形象具体的 CPU，也是一种收获。至于课程建议，我个人感觉可以在做实验之前带领一下学生怎么入手，毕竟很多计算机专业学生都没有 verilog 编程基础。其次，可以在说明文档上说明一下大致做到如何程度可以通过那些点，这样做实验做起来有目标一些。最后，如果能在实验中加入一些前沿的技术，就更好了。

董苡廷：

本次实验，我们组完成了五级流水线结构。组员内部分工明确，合作愉快。在这次实验中我了解了五级流水线的实现过程，将上课讲的内容进行完整地实现，更深入地理解了知识内容。在实验过程中我学习了加指令，接线，数据相关，还有如何加气泡实现 loadstore 等内容。初步认识了 Verilog 语言，虽然过程很不容易，大家零基础开始学起，遇到各种各样的困难，但是在助教和同学的帮助下都一一解决了。实验圆满成功。