《计算机系统》实验报告

班级：人工智能 2002 班

姓名：王艺茗 20206444
关君然 20206474

日期：2022.12.29

目 录

1. 实验概括

1.1 工作量

王艺茗：50%　　关君然：50%

1.2 总体设计

流水线是数字系统中一种提高系统稳定性和工作速度的方法，广泛应用于高档 CPU 的架构中。根据 MIPS 处理器的特点，将整体的处理过程分为取指令（IF）、指令译码（ID）、执行（EX）、存储器访问（MEM）和寄存器回写（WB）五级，对应多周期的五个处理阶段。

其中，一个指令的执行需要 5 个时钟周期，每个时钟周期的上升沿来临时，此指令所代表的一系列数据和控制信息将转移到下一级处理。

DLX 的基本流水线示意图如下所示：

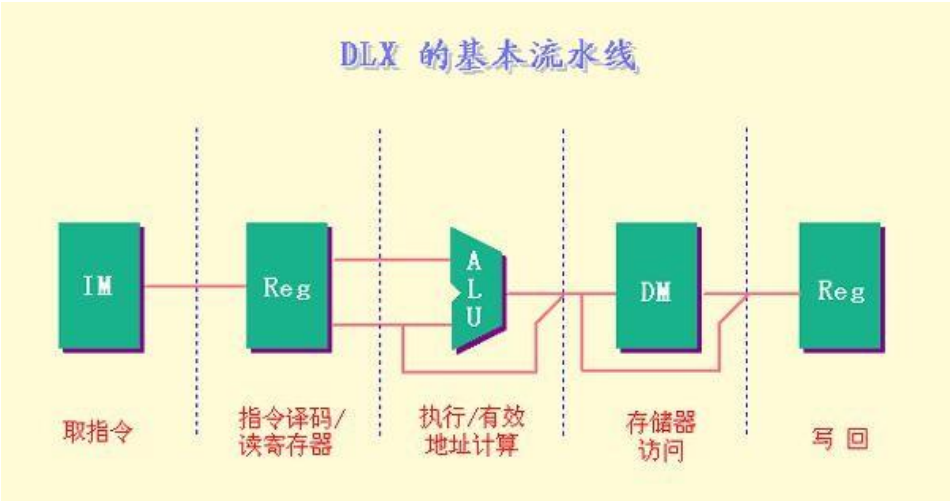


图 1.1 DLX 的流水线

1.2.1 取指令周期（IF）

IR ← Mem[PC]

NPC ← PC＋4

该周期的功能是从指令存储器中取出指令。

本段有一个 PC 寄存器用于存放指令地址，PC 输入是一个选择器选择的结果。从 PC 寄存器中取出指令地址，根据 PC 从指令存储器中取出指令，送入 IF/ID 流水段寄

存器以便后面流水段使用，并吧 PC+4 的值一并存入流水段寄存器。这是用于分支指令的控制冒险用于恢复 PC 所用。

1.2.2 指令译码/读寄存器周期（ID）

A ← Regs[IR6 ..10] (Regs[rs])

B ← Regs[IR11 ..15] (Regs[rt])

Imm ← (IR16)16 ## IR16 ..31

本阶段的主要功能是 ID 段的主要工作为指令的解析，寄存器的访存以及跳转指令的地址计算。将指令进行译码生成控制信号，并从寄存器中取出相应的操作数。是从上一个流水段寄存中取出指令，并送入译码控制器进行译码，译码完成厚将译码结果存入下一个流水段寄存器，并根据译码结果将立即数扩展，扩展方式共有 4 中，分别是逻辑扩展，算数扩展，为 lui 指令进行的扩展，和为分支指令进行的扩展。

1.2.3 执行/有效地址计算周期（EX）

存储器访问（load 和 store）

ALUOutput ← A＋Imm

寄存器－寄存器 ALU 操作

ALUOutput ← A op B

寄存器－立即值 ALU 操作

ALUOutput ← A op Imm

分支操作

ALUOutput ← NPC＋Imm 计算偏移地址

Cond ← (A op 0) 判断分支是否成功，失败则结束

根据指令的编码进行算数或者逻辑运算或者计算条件分支指令的跳转目标地址。

此外 LW、SW 指令所用的 RAM 访问地址也是在本周期上实现。

数据运算是从上一流水段寄存器中取出操作数和控制信号，根据控制信号控制数据通路进行数据的运算，并将运算的结果存入下一个流水段。但是其取出的操作数也是进行了转发选择的数据，转发同样还是采用的按字节转发的模式。

而抵制运算则是为了跳转和分支指令准备的，如果跳转地址会被送回第一部分也即取值模块的 PC 源之一进行选择。跳转指令也会再下一个流水段送回，因为在下一个流水段才能知道是否需要跳转。

1.2.4 存储器访问 / 分支完成周期（MEM）

在该周期处理的 DLX 指令只有 Load、Store 和分支指令。

存储器访问（load 和 store）

LMD ← Mem[ALUOutput]

或 Mem [ALUOutput] ← B

分支操作

if（cond）PC ← ALUOutput　　成功则把计算好的地址放入 PC

else PC ← NPC　　否则不做

其他类型指令均不做

只有在执行 LW、SW 指令时才对存储器进行读写，对其他指令只起到一个周期的作用。

本阶段进行的操作是内存相关的操作，早内存的读写口之前都有一个移位器，只是为 LW 和 SW 等类型的指令准备的，还有一个 condition 的检查单元，该单元是用于判断跳转是否有效和是否真的需要写回寄存器的。

1.2.5 写回周期（WB）

不同指令在该周期完成的工作也不一样

寄存器－寄存器型 ALU 指令

Regs[IR16 ..20] (rd)← ALUOutput

寄存器－立即值型 ALU 指令

Regs[IR11 ..15] (rt)← ALUOutput

Load 指令

Regs[IR11 ..15] (rt)← LMD

该段把指令执行的结果写回到寄存器文件中，即 ALU 运算指令和 load 指令在这个周期把结果数据写入通用寄存器组。ALU 运算指令的结果数据来自 ALU，而 load 指令的结果数据来自存储器。

整个流水线 CPU 的时钟关系，由于每个流水段寄存器都是设计在下降沿进行写入的，而寄存器的读取设计在上升沿，为了解决写回寄存器的时候可能同时对同一地址进行读写引起的结构冲突，所以在设计的时候采用了读写分不同边沿的，那么这里就有可能出现数据的延迟，也就是想要的数据还没写回即要读取，因此这里会使用到一种转发关系。

1.2.6 在本方案的实现中：

分支指令需要 4 个时钟周期（移到 ID 段，只需 2 个周期）

store 指令需要 4 个时钟周期

其它指令需要 5 个时钟周期

1.3 MIPS 指令格式

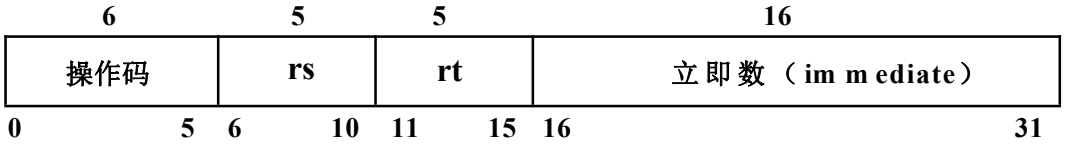
MIPS 指令系统结构有 MIPS-32 和MIPS-64 两种结构，本实验的 MIPS 指令选用 MIPS-32。以下所说的 MIPS 指令均指 MIPS-32，即 MIPS 的指令格式为 32 位。

除个别指令外，所有指令的格式均为立即数型（I-Type）、跳转型（J-Type）和寄存器型（R-Type）三种类型中的一种。三类指令格式如下所示。

1）I 类指令(Immediate)

包括所有的 load 和store 指令、立即数指令、分支指令、寄存器跳转指令、寄存器链接跳转指令。

立即数字段为 16 位，用于提供立即数或偏移量。



其中，I 类指令包括：

load 指令

访存有效地址：Regs[rs]＋immediate

从存储器取来的数据放入寄存器 rt

store 指令

访存有效地址：Regs[rs]＋immediate

要存入存储器的数据放在寄存器 rt 中

立即数指令

Regs[rt] ← Regs[rs] op immediate

分支指令

转移目标地址：Regs[rs]＋immediate，rt 无用

寄存器跳转、寄存器跳转并链接

转移目标地址为 Regs[rs]

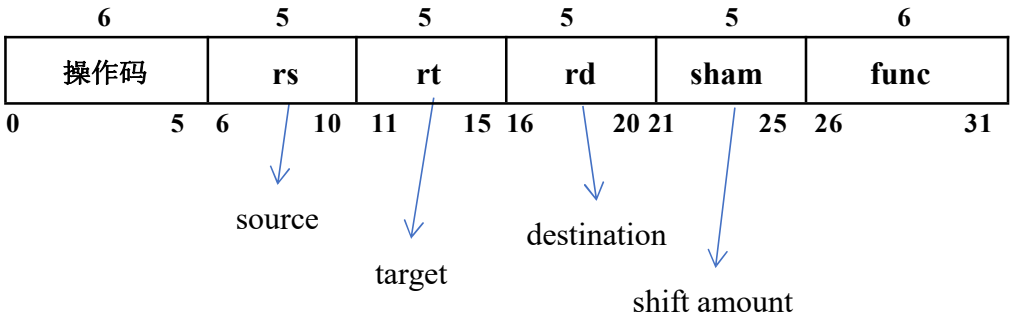
2）R 类指令(Register-to-Register)

包括 ALU 指令、专用寄存器读/写指令、move 指令等。

ALU 指令

egs[rd]← Regs[rs] func Regs[rt]

func 为具体的运算操作编码。



3）J 类指令（Jump）

包括跳转指令、跳转并链接指令、自陷指令、异常返回指令。

在这类指令中，指令字的低 26 位是偏移量，它与 PC 值相加形成跳转的地址。

处理器实现的指令包括除 4 条非对齐指令外的所有 MIPS I 指令以及 MIPS32 中的 ERET 指令，有 14 条算术运算指令、8 条逻辑运算指令，6 条移位指令、8 条分支跳转指令、4 条数据移动指令、12 条访存指令，共计 52 条。

1.4 不同流水段之间的连线图

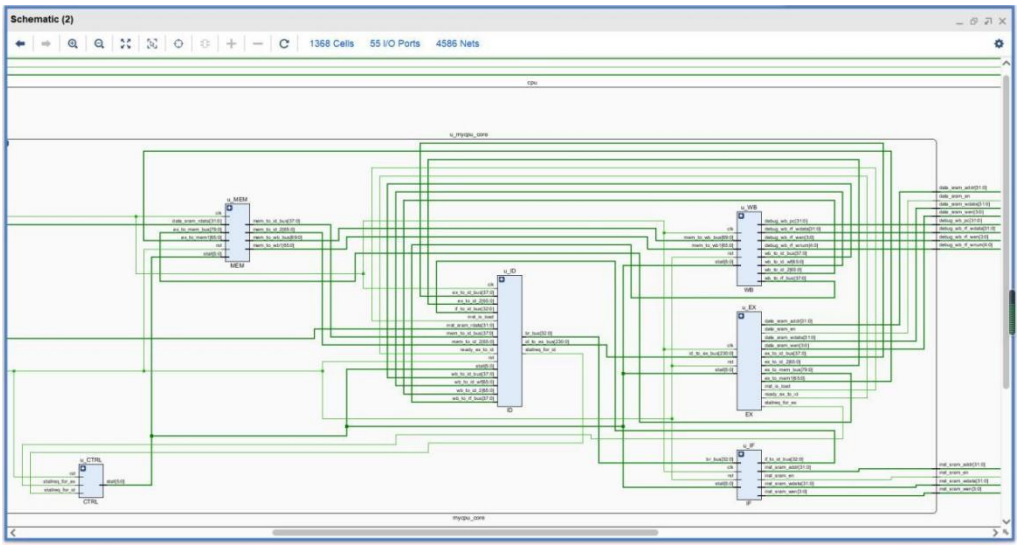


图 1.2 流水线 CPU 总体结构图

1.5 指令完成度

添加完成了如下指令：

算术运算指令、逻辑运算指令、移位指令、分支跳转指令、数据移动指令、访存指令， 共计 49 条指令。

按照写实验的先后循序这些指令顺序如下：

inst_ori, inst_lui, inst_addiu, inst_beq, inst_subu, inst_jr, inst_jal, inst_addu, inst_bne, inst_sll, inst_or,inst_lw, inst_sw, inst_xor ,inst_sltu, inst_slt, inst_slti, inst_sltiu, inst_j, inst_add, inst_addi ,inst_sub, inst_and , inst_andi, inst_nor, inst_xori, inst_sllv, inst_sra, inst_bgez, inst_bltz, inst_bgtz, inst_blez,inst_bgezal,inst_bltzal, inst_jalr, inst_mflo, inst_mfhi, inst_mthi, inst_mtlo, inst_div, inst_divi, inst_mult,inst_multu, inst_lb, inst_lbu, inst_lh, inst_lhu, inst_sb, inst_sh

成功通过第 64 个点，并制作了一个 32 时钟周期的乘法器。

1.6 程序运行环境及使用工具

操作系统：Windows 10 64 位(DirectX 12)。

开发平台：Vivado 2019.2。

编程语言：Verilog HDL 硬件描述语言。

2. 单个流水段说明

2.1 IF 段

2.1.1 整体功能说明

取指令周期（IF 段）

IF 段主要功能为：

以程序计数器PC 中的内容作为地址，从存储器中取出指令并放入指令寄存器 IR；同时 PC 值加 4（假设每条指令占 4 个字节），指向顺序的下一条指令。

IF 段负责根据是否有跳转给下一个 PC 赋值，将取指令信号变为真，并将总线传入 WB 段。并将当前计算得到的 PC 值传给指令存储器，让指令存储器在指令译码/读寄存器周期(ID 段)中，得到当前 PC 值所对应指令的 inst 值，使存储器根据 PC 地址向 ID 段返回 32 位 ins 指令，同时将 PC 值传给 ID 段方便后面指令的使用。

2.1.2 端口介绍

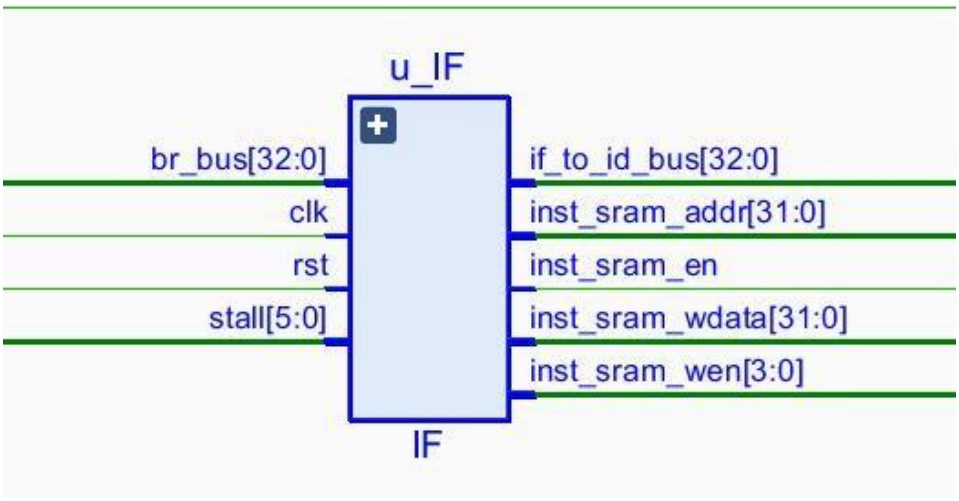


图 2.1 IF 段接口示意图

1)输入端口：br_bus[32:0],clk,rst,stall[5:0]

2) 输 出 端 口 ：
if_to_di_bus[32:0],inst_sram_addr[31:0],inst_sram_en,inst_sram_wdata[31:0]
,inst_sram_wen[3:0]

2.1.3 信号介绍

rst 是接收到的复位信号，clk 是接收到的时钟信号。

br_bus[32:0]是从 ID 段中接收到的跳转指令改变下一条指令的信号。其中包括了 br_e 跳转使能信号，br_addr[31:0]跳转地址值。当 br_e 为1 时，且br_addr[31:0]有值时，则将 br_addr[31:0]赋值给当前指令的 pc 值，并且将此 pc 值发给指令寄存器，从而在 ID 段得出跳转后的指令。

stall[5:0]是从 CTRL.v 文件中接收到的暂停信号，如果 stall[0]==1’b1，则 pc 值保持上一个时钟周期的值不变，使之发生暂停操作。如果不进行暂停，则正常进行 pc 值加 4 或者发生跳转地址操作。

if_to_di_bus[32:0]是将当前得到的 pc 值发送给 ID 段。

inst_sram_en 是进行读指令存储器的使能信号。

inst_sram_addr[31:0]是将存入了当前的 pc 值，并将其发送给指令存储器，指令存储器得到了 pc 值后，将改 pc 值对应下的指令 inst 值发送给 ID 段中，进行译码。

inst_sram_wdata[31:0],inst_sram_wen[3:0]是将值写入指令存储器的操作，因为在 IF 不涉及到存指令存储器的操作，所以在该段将两个值都赋为 0。

2.2 ID 段

2.2.1 整体功能说明

指令译码/读寄存器周期（ID 段）

ID 段主要功能为：

对从指令存储器中得到的指令 inst 值进行译码，并用 IR 中的寄存器地址去访问通用寄存器组，从 32 个通用寄存器 reg_array[31:0]中读出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。如果是转移指令满足转移条件，那么要给出转移目标，作为新的指令地址。

若解析到指令为跳转指令，则计算跳转地址，并将结果通过 br.bus,总线传回 IF 段，以便 IF 段在最短时间拿到下一条指令的地址。另外将各种使能信号以及操作数1和操作数 2 通过 ID_TO_EX 总线传递给下一阶段的 EX。

注：指令译码和读寄存器是并行进行的。之所以能做到这一点，是因为在 DLX 指令格式中，操作码在固定位置。这种技术也称为固定字段译码。

2.2.2 端口介绍

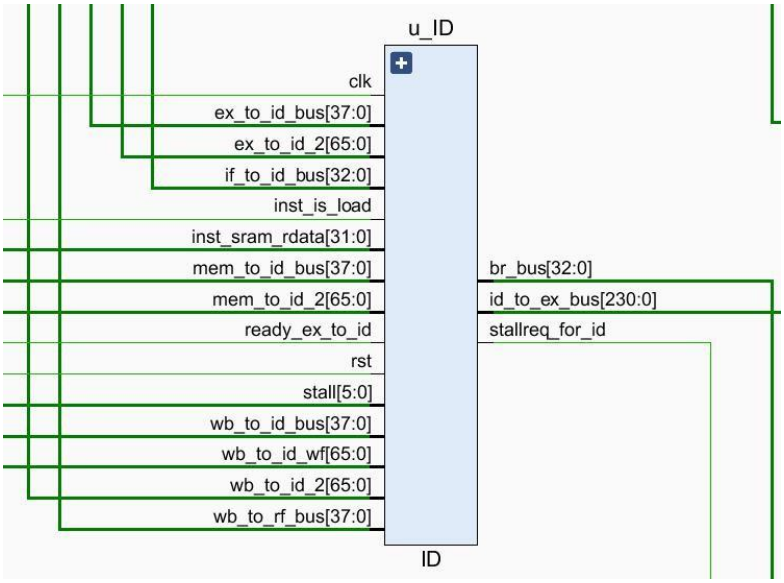


图 2.2 ID 段接口示意图

1)输 入 端 口 ：

clk,rst,stall[5:0],ex_to_id_bus[37:0],mem_to_id_bus[37:0],wb_to_id_bus[37:0],ex_to_id_2[65:0],mem_to_id_2[65:0],wb_to_id_2[65:0],

if_to_id_bus[32:0],inst_sram_rdata[31:0],inst_is_load,wb_to_rf_bus[37:0], wb_to_id_wf[65:0], ready_ex_to_id

2)输出端口：br_bus[32:0],id_to_ex_bus[230:0],stallreg_for_id

2.2.3 信号介绍

rst 是接收到的复位信号，clk 是接收到的时钟信号。

if_to_id_bus[32:0]是从 IF 传过来的当前流水线所处理的 pc 值，并将其[31:0]的值附给 inst。

inst_sram_rdata[31:0]是从指令存储器接收到的当前pc 值对应指令的inst 值，当 ID 段没有被暂停时，则将此值赋给 inst 变量并进行指令判断，找出 inst 所对应的指令，以备进行后续的指令操作。

ex_to_id_bus[37:0],mem_to_id_bus[37:0],wb_to_id_bus[37:0],ex_to_id_2[65:0],mem_to_id_2[65:0],wb_to_id_2[65:0]这些指令都是跟数据相关有关的指令，当当前指令需要取前面还未存入寄存器的值的时候，由 ex 段、mem 段、wb 段提前发给 ID 段，再由 ID 段发送给 regfile.v 文件中，进行赋值给 rs 和 rt 所需要的寄存器的值。

stall[5:0]是从 CTRL.v 文件中接收到的暂停信号，如果 stall[2]==1’b1，则将寄存器 if_to_id_bus_r 的值赋值为 0，以来保证从 if 段传过来的值不能在当前时钟周期赋值，则保证了下一个时钟周期不会得到从当前时钟周期传入到下一个阶段的值，从而将当前时钟周期的操作暂停。

```
always @ (posedge clk) begin      //如果 ID 段需要暂停，则将当前的 inst 值赋值给
        inst_stall_en<=1'b0;      //inst_stall 临时寄存器中
        inst_stall <=32'b0;
        if(stall[1] == 1'b1 & ready_ex_to_id ==1'b0)begin
        inst_stall <= inst;
        inst_stall_en<=1'b1;
        end
    end
```

```
assign inst_stall1 = inst_stall;//将上一个时钟周期 inst_stall 的值赋
    assign inst_stall_en1 = inst_stall_en ;// 给当前的 inst 值
    assign inst = inst_stall_en1 ? inst_stall1  :inst_sram_rdata;
```

同时将当前时钟周期的 inst 保存在临时寄存器 inst_stall[31:0]中，同时将该寄存器的使能信号 inst_stall_en 赋值为 1’b1。并在下一个时钟周期将该寄存器的值赋值给下一个时钟周期的 inst 值，在 stall[2]==1’b0 后则将该寄存器的值变为 32’b0，其该寄存器的使能信号 inst_stall_en 也变为 1’b0。以此来达到暂存当前时钟周期的 inst 的目的。

ready_ex_to_id 是从 EX 段传过来的的一个信号，用来接收到 EX 段中乘除法操作是否完成，如果没有完成，则该信号值为 0，如果完成该信号的值为 1，如果在 ex 段进行的指令是乘除法，因为他们要进行 32 个时钟周期，所以要将后面的流水段进行暂停，所以 ID 段需要一直保存上个时钟周期的 inst 值。

wb_to_id_wf[65:0]和 wb_to_rf_bus[37:0]是从 WB 进行写回寄存器的值，在 ID 段中进行接收，并将其直接传入到 regfile.v 文件中去，从而进行写寄存器的操作。

inst_is_load 是从EX 段接收到的信号，用来判断 EX 的当前指令是否是LW 指令，如果是，则该值为 1，如果不是，则该值为 0。并且与 rs 寄存器和 rt 寄存器中的地址进行判断，如果 EX 段的 lw 要写入的寄存器的地址与当前 ID 段的指令要读取的寄存器的地址有相同的，则 stallreg_for_id 为1’b1，并将此值赋值给 CTRL.v 中， 在此文件中发出暂停 ID 段和 IF 的暂停信号。

br_bus[32:0]是跳转指令所要跳转到的 pc 值，将跳转后的 pc 值和跳转使能信号发送给 ID 段，从而让下一个流水线进行跳转到目标指令。

id_to_ex_bus[230:0]是 ID 段要发送给 EX 的值，其中包括了：当前指令的 pc 值和inst 值，进行alu_op 操作的信号，进行 alu 操作数 1 的目标值选择sel_alu_src1 和进行 alu 操作数 2 的目标值选择 sel_slu_src2，对存储器进行访存操作的使能信号 data_ram_en，对存储器进行操作的 data_sram_wen[3:0],对寄存器 reg_array[31:0]进行写操作的使能信号 rf_we,对寄存器 reg_array[31:0]进行写操作的寄存器的地址，寄存器 rs 和寄存器 rt 所存的数值 rdata1 和rdata2，对 hilo 寄存器进行读写操作的 lo_hi_r[1:0]和 lo_hi_w[1:0]，寄存器 hilo 所对应的数值 lo_o 和hi_o，对寄存器进行操作的 data_ram_read[3:0]。

2.2.4 regfile 功能模块介绍

在 ID 段，可以对 regfile.v 文件进行操作，regfile.v 文件中包括了寄存器 reg_array[31:0]和存乘除法结果的 hilo 寄存器。

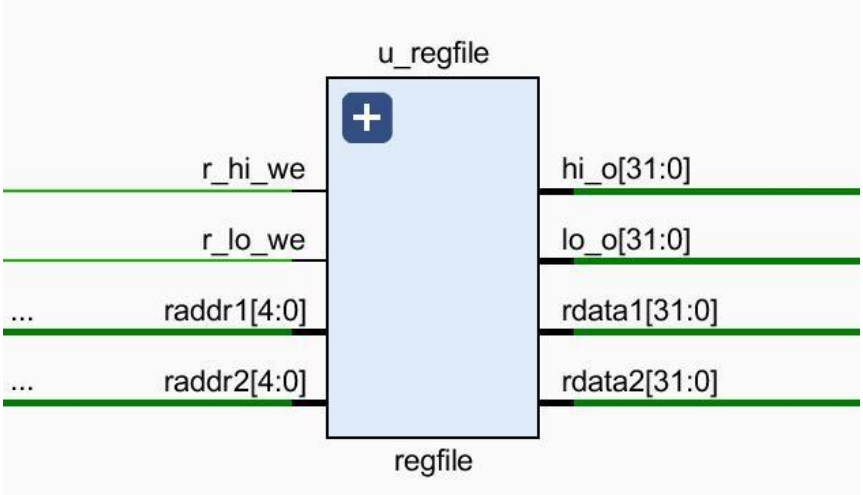


图 2.3 regfile 模块接口示意图

regfile.v 文件中，

主要的输入信号是：r_hi_we,r_lo_we,raddr1[4:0],raddr2[4:0]

主要的输出信号是：hi_o[31:0],lo_o[31:0],rdata1[31:0],radta2[31:0]

其中 r_hi_we 和 r_lo_we 是读取 hilo 寄存器的信号，其从 hilo 寄存器中读出的结果赋值给 hi_o[31:0]和 lo_o[31:0]返回给 ID 段。

其中 raddr1[4:0],raddr2[4:0]是要读取 reg_array[31:0]寄存器中的地址，其从 reg_array[31:0]寄存器读出的结果赋值给 rdata1[31:0]和 radta2[31:0]返回给 ID 段。

在 regfile.v 中，在这里需要解决数据相关的问题，将从 EX、MEM、WB 段需要写到寄存器的值提前传入到 regfile.v 中，如果当前指令中 rs 和 rt 的值与即将要写入寄存器的地址相同，则将传回要写入寄存器的值提前赋给 rdata1[31:0]或者 rdata2[31:0],从而避免了读取到的寄存器值不正确的情况，从而解决了数据相关的问题，代码如下：

```
assign rdata1 = (raddr1 == 5'b0) ? 32'b0 :
    ((raddr1 == ex_rf_waddr)&& ex_rf_we) ? ex_result :
    ((raddr1 == mem_rf_waddr)&& mem_rf_we) ? mem_rf_wdata :
    ((raddr1 == wb1_rf_waddr)&& wb1_rf_we) ? wb1_rf_wdata :
    reg_array[raddr1];
    // read out2
    assign rdata2 = (raddr2 == 5'b0) ? 32'b0 :
    ((raddr2 == ex_rf_waddr)&& ex_rf_we) ? ex_result :
    ((raddr2 == mem_rf_waddr)&& mem_rf_we) ? mem_rf_wdata :
    ((raddr2 == wb1_rf_waddr)&& wb1_rf_we) ? wb1_rf_wdata :
    reg_array[raddr2];
    assign hi_o = hi_ex_we ? hi_ex:
                  hi_mem_we ? hi_mem:
                  hi_wb_we ? hi_wb:
```

```
                hi;
assign lo_o = lo_ex_we ? lo_ex:
                lo_mem_we ? lo_mem:
                lo_wb_we ? lo_wb:lo;
```

2.3 EX 段

2.3.1 整体功能说明

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，那么还会计算 load/store 的目标地址。

在这个周期，不同的指令有不同的操作。

load 指令

访存有效地址：Regs[rs]＋immediate

ALU 把指令中所指定的寄存器的内容与偏移量相加，形成访存有效地址，将从存储器取来的数据放入寄存器 rt 中。

store 指令

访存有效地址：Regs[rs]＋immediate

ALU 把指令中所指定的寄存器的内容与偏移量相加，形成访存有效地址，将要存入存储器的数据放在寄存器 rt 中。

寄存器－寄存器 ALU 指令

ALU 按照操作码指定的操作对从通用寄存器组中读出的数据进行运算。

寄存器－立即数 ALU 指令

ALU 按照操作码指定的操作对从通用寄存器组中读出的操作数和指令中给出的立即数进行运算。

立即数指令

Regs[rt] ← Regs[rs] op immediate

分支指令

转移目标地址：Regs[rs]＋immediate，rt 无用

ALU 把指令中给出的偏移量与 PC 值相加，形成转移目标的地址。同时，对在前一个周期读出的操作数进行判断，确定分支是否成功。

寄存器跳转、寄存器跳转并链接

转移目标地址为 Regs[rs]

2.3.2 端口介绍

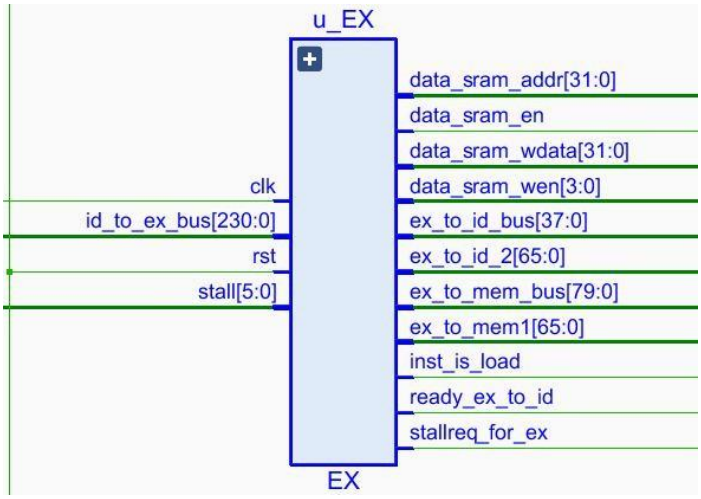


图 2.4 EX 段接口示意图

1)输入端口：clk,rst,id_to_ex_bus[230:0],stall[5:0]

2)输出端口：
data_sram_addr[31:0],data_sram_en,data_sram_wdata[31:0],data_sram_wen[3:0],ex_to_id_bus[37:0],
ex_to_id2[65:0],ex_to_mem_bus[79:0],ex_to_mem1[65:0],inst_is_load,ready_ex_to_id,stall_for_ex

2.3.3 信号介绍

rst 是接收到的复位信号，clk 是接收到的时钟信号。

id_to_ex_bus[230:0]是 ID 段要发送给 EX 的值，其中包括了：当前指令的 pc 值和inst 值，进行alu_op 操作的信号，进行 alu 操作数 1 的目标值选择sel_alu_src1 和进行 alu 操作数 2 的目标值选择 sel_slu_src2，对存储器进行访存操作的使能信号 data_ram_en，对存储器进行操作的 data_sram_wen[3:0],对寄存器 reg_array[31:0]进行写操作的使能信号 rf_we,对寄存器 reg_array[31:0]进行写操作的寄存器的地址，寄存器 rs 和寄存器 rt 所存的数值 rdata1 和rdata2，对 hilo 寄存器进行读写操作的 lo_hi_r[1:0]和 lo_hi_w[1:0]，寄存器 hilo 所对应的数值 lo_o 和hi_o，对寄存器进行操作的 data_ram_read[3:0]。

stall[5:0]是从 CTRL.v 文件中接收到的暂停信号，如果 stall[3]==1’b1，则代表要暂停 EX 段，当 EX 段的指令为乘法或除法时，因为乘除法需要 32 个时钟周期

的计算时间，所以需要将 IF、ID、EX 段进行暂停操作，直到乘除法指令结束后流水线才正常运行。

data_sram_en 时对存储器的访存使能信号，当需要对存储器进行操作时，则令其值为 1’b1。

data_sram_addr[31:0]将 EX 段中算出的结果传给存储器进行寻址，并将寻址得到的值通过 data_sram_rdata 传递到 MEM 段中。

data_sram_wen[3:0]通过传递当前指令对存储器的操作指令，在 data_sram_wdata[31:0]中控制不同类型的将要写入存储器的值。

ex_to_mem_bus[79:0]是 EX 段要发送给 MEM 的值，其中包括了：当前指令的 pc 值，对存储器进行访存操作的使能信号 data_ram_en，对存储器进行操作的 data_sram_wen[3:0],对寄存器 reg_array[31:0]进行写操作的使能信号 rf_we,对寄存器 reg_array[31:0]进行写操作的寄存器的地址 rf_waddr，EX 段中算出的结果 ex_result[31:0],对寄存器进行操作的 data_ram_read[3:0]。

ex_to_meml[65:0]是 EX 段要发送给 MEM 的值，其包括了写 hi 和 lo 寄存器的使能信号，用于判断是否进行写寄存器的操作，还有包括了将要写入 hi 和 lo 寄存器的值，如果不写，则此处为 0，且使能信号为 0。

ex_to_id_2[65:0]是 EX 段要发送给 ID 段中的 regfile.v，用于解决下一条指令要用到上一条指令存入 hilo 寄存器值的问题。

inst_is_load 是 EX 段发送到 ID 段的信号，用来判断 EX 的当前指令是否是 LW 指令，如果是，则该值为 1，如果不是，则该值为 0。并且与 rs 寄存器和 rt 寄存器中的地址进行判断，如果 EX 段的 lw 要写入的寄存器的地址与当前 ID 段的指令要读取的寄存器的地址有相同的，则 stallreg_for_id 为 1’b1，并将此值赋值给 CTRL.v 中，在此文件中发出暂停 ID 段和 IF 的暂停信号。

ready_ex_to_id 是 EX 段发送到 ID 段的一个信号，用来接收到 EX 段中乘除法操作是否完成，如果没有完成，则该信号值为 0，如果完成该信号的值为 1，如果在 ex 段进行的指令是乘除法，因为他们要进行 32 个时钟周期，所以要将后面的流水段进行暂停，所以 ID 段需要一直保存上个时钟周期的 inst 值。

stallreq_for_ex 是从 EX 段发送给 CTRL.v 的一个请求暂停指令，这个操作是由于乘除法器需要 32 个时钟周期计算而导致的，需要让 CTRL.v 发送到 IF、ID、EX 段进行暂停操作，直到乘除法指令结束后流水线才正常运行。

ex_to_id_bus[37:0]是跟数据相关有关的指令，当当前指令需要取前面还未存入寄存器的值的时候，EX 段提前发给 ID 段，再由 ID 段发送给 regfile.v 文件中，进行赋给 rs 和 rt 所需要的寄存器的值。

2.3.4 mul_plus 功能模块介绍

mul_plus 是一个.v 文件，它通过 EX 进行连接

```
mul_plus u_mul_plus(                          //连接 mul_plus.v
    .clk        (clk            ),
    .start_i      (mul_begin),
    .mul_sign     (mul_signed),
        .opdata1_i    (  rf_rdata1    ),
        .opdata2_i    (  rf_rdata2    ),
        .result_o     (mul_result     ),
        .ready_o      (mul_ready_i      )
    );
```

其中 mul_begin 用来判断乘法的开始，如果为 1’b1，则进行乘法操作，并且通过指令来判断此次乘法为有符号乘法还是无符号乘法并传输进 mul_plus.v 中。将 ID 段传到 EX 段的两个操作数传入到乘法器中，并进行二进制数的乘法操作，最后将结果返回到 EX 段中，并分为高位[63:32]和低位[31:0]分别存入 hi 和 lo 寄存器中。

我们学习了两种实现乘法器的方法：

第一种是串行乘法器，即两个 N 位二进制数 x、y 的乘积用简单的方法计算就是利用移位操作来实现。

串行乘法器的乘法功能是正确的，但计算一次乘法需要 8 个周期。因此可以看出串行乘法器速度比较慢、时延大，但这种乘法器的优点则是所占用的资源是所有类型乘法器中最少的，在低速的信号处理中有着广泛的应用。

另外一种就是流水线乘法器，一般的快速乘法器通常采用逐位并行的迭代阵列结 构，将每个操作数的 N 位都并行地提交给乘法器。但是一般对于 FPGA 来讲，进位的 速度快于加法的速度，这种阵列结构并不是最优的。所以可以采用多级流水线的形式， 将相邻的两个部分乘积结果再加到最终的输出乘积上，即排成一个二叉树形式的结构， 这样对于 N 位乘法器需要 lb（N）级来实现。

经过对比，发现流水线乘法器比串行乘法器的速度快很多很多，在非高速的信号处理中有广泛的应用。至于高速信号的乘法一般需要利用 FPGA 芯片中内嵌的硬核DSP 单元来实现。

自制乘法器说明：

步骤：

1 将操作数 1 和操作数 2 进行取绝对值操作，如果其首位为 1 且当前乘法为有符号乘法，则将此操作数取反后加 1 作为原操作数的绝对值。

2 将操作数 1 的绝对值作为被乘数，其[63:32]位赋值为 0，[31:0]位赋值位操作数 1 的绝对值，再将操作数 2 作为乘数[31:0];

3 每经过一个时钟周期，被乘数左移一位，最低位补 0，最高位舍弃；

4 每经过一个时钟周期，乘数右移一位，最高位补 0，最低为舍弃；

5 如果当前时钟周期的乘数最低位为 1，则将当前时钟周期的被乘数与乘法的临时结果相加赋值给临时结果，如果为 0，临时结果保持不变；

6 如果乘数不为 0，则跳回到 3，如果乘数为 0，则结束循环；

7 如果是有符号乘法，则将原操作码 1 和 2 的符号位进行取非或，得到最终结果的符号，如果是有符号的，则将得到的临时结果取反加 1，否者临时结果直接赋值给输出。

2.4 MEM 段

2.4.1 整体功能说明

如果是 load/store 指令，那么在此阶段会访问数据存储器，反之，只是将执行阶段的结果向下传递到回写阶段。如果存储器传回来了数值，则需要进行通过相关的指令如 lb、lbu、lh、lhu、sb、sh 来判断进行取值操作，再进一步将存储器得到的值存入到寄存器中。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 段最后会将各类寄存器的读写使能信号、地址和写入数据合并为 MEM_TO_WB 总线，并传入 WB 段。

2.4.2 端口介绍

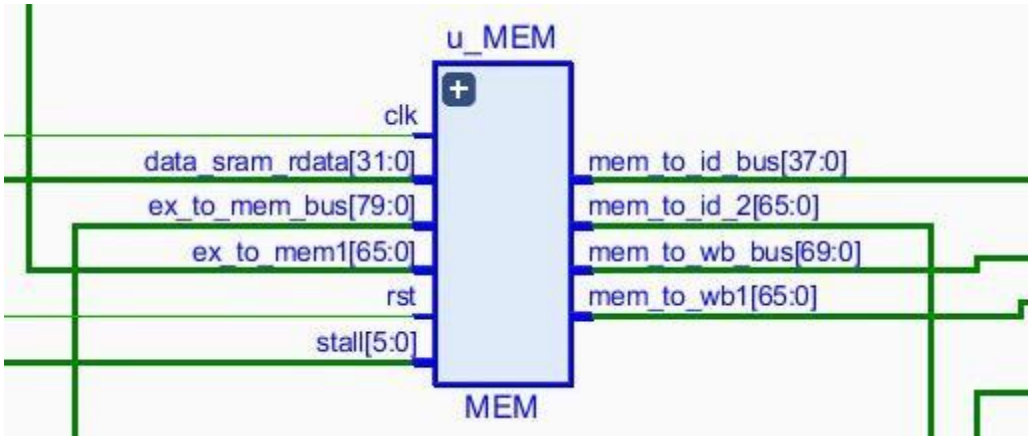


图 2.5 MEM 段接口示意图

1)输入端口：

data_sram_rdata[31:0],ex_to_mem_bus[79:0],ex_to_mem1[65:0],stall[5:0],rst,clk

2)输 出 端 口 ：
mem_to_id_bus[37:0],mem_to_id_2[65:0],mem_to_wb_bus[69:0],mem_to_wb1[65:0]

2.4.3 信号介绍

rst 是接收到的复位信号，clk 是接收到的时钟信号。data_sram_rdata[31:0]是从存储器中在 EX 读取到的数值，在 MEM 接收得到。

ex_to_mem_bus[79:0]是 MEM 段接收到的从 EX 段发送过来的值，其中包括了：当前指令的 pc 值，对存储器进行访存操作的使能信号 data_ram_en，对存储器进行操作的data_sram_wen[3:0],对寄存器reg_array[31:0]进行写操作的使能信号rf_we,对寄存器 reg_array[31:0]进行写操作的寄存器的地址 rf_waddr，EX 段中算出的结果ex_result[31:0],对寄存器进行操作的 data_ram_read[3:0]。

ex_to_mem1[65:0]是 MEM 接收到从 EX 段发送过来的值，其包括了写 hi 和 lo 寄存器的使能信号，用于判断是否进行写寄存器的操作，还有包括了将要写入 hi 和 lo 寄存器的值，如果不写，则此处为 0，且使能信号为 0。

mem_to_id_2[65:0]是 mem 段要发送给 ID 段中的 regfile.v，用于解决下一条指令要用到上一条指令存入 hilo 寄存器值的问题。

mem_to_wb1[65:0]是 MEM 段要发送给 WB 的值，其包括了写 hi 和 lo 寄存器的使能信号，用于判断是否进行写寄存器的操作，还有包括了将要写入 hi 和 lo 寄存器的值，如果不写，则此处为 0，且使能信号为 0。

mem_to_id_bus[37:0]是跟数据相关有关的指令，当当前指令需要取前面还未存入寄存器的值的时候，由 MEM 段提前发给 ID 段，再由 ID 段发送给 regfile.v 文件中，进行赋给 rs 和 rt 所需要的寄存器的值。

mem_to_wb_bus[69:0]是 MEM 段要发送给 WB 段的值，其中包括了：当前指令的 pc 值，对寄存器 reg_array[31:0]进行写操作的使能信号 rf_we,对寄存器 reg_array[31:0]进行写操作的寄存器的地址 rf_waddr，EX 段中算出的结果 ex_result[31:0]。

在 MEM 段中，会进行判断一下，最终写入寄存器中的值是从 EX 段传过来的 ex_result[31:0]还是从存储器中传下来的 data_sram_rdata[31:0],判断后再传给 rf_wdata。

```
    assign mem_result = data_sram_rdata;
   assign rf_wdata =  (data_ram_read==4'b1111 && data_ram_en==1'b1) ? mem_result :
                       (data_ram_read==4'b0001 && data_ram_en==1'b1 &&
ex_result[1:0]==2'b00) ?({{24{mem_result[7]}},mem_result[7:0]}):
```

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

2.5 WB 段

2.5.1 整体功能说明

将运算结果保存到目标寄存器。

2.5.2 端口介绍

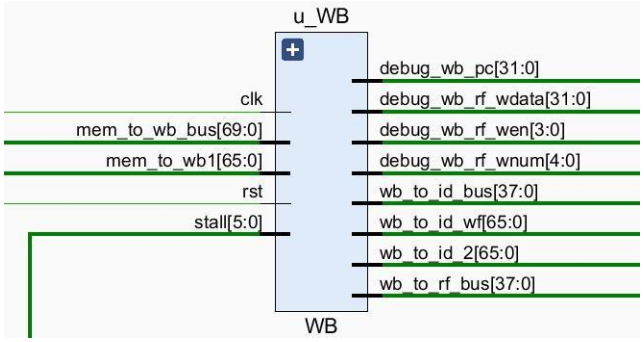


图 2.6 WB 段接口示意图

1)输入端口：clk,rst,mem_to_wb_bus[69:0],mem_to_wb1[65:0],stall[5:0]

2)输出端口：
wb_to_id_bus[37:0],wb_to_id_wf[65:0],wb_to_id_2[65:0],wb_to_rf_bus[37:0]

2.5.2 信号介绍

rst 是接收到的复位信号，clk 是接收到的时钟信号。

mem_to_wb_bus[69:0]是 WB 段接收到的从 MEM 段发过来的值，其中包括了：当前指令的 pc 值，对寄存器 reg_array[31:0]进行写操作的使能信号 rf_we,对寄存器 reg_array[31:0]进行写操作的寄存器的地址 rf_waddr，EX 段中算出的结果 ex_result[31:0]。

mem_to_wb1[65:0]是 WB 段接收到的从 MEM 段发过来的值，其包括了写 hi 和 lo 寄存器的使能信号，用于判断是否进行写寄存器的操作，还有包括了将要写入 hi 和 lo 寄存器的值，如果不写，则此处为 0，且使能信号为 0。

wb_to_id_2[65:0]是 WB 段要发送给 ID 段中的 regfile.v，用于解决下一条指令要用到上一条指令存入 hilo 寄存器值的问题。

wb_to_id_bus[37:0]是跟数据相关有关的指令，当当前指令需要取前面还未存入寄存器的值的时候，由 WB 段提前发给 ID 段，再由 ID 段发送给 regfile.v 文件中，进行赋给 rs 和 rt 所需要的寄存器的值。

wb_to_rf_bus[37:0]是 wb 要写回 reg_array[31:0]的值，其中包括了对寄存器 reg_array[31:0]进行写操作的使能信号 rf_we,对寄存器 reg_array[31:0]进行写操作的寄存器的地址 rf_waddr，EX 段中算出的结果 ex_result[31:0]。

wb_to_id_wf[65:0] 是 WB 段要发送给 regfile.v 的值，其包括了写 hi 和 lo 寄存器的使能信号，用于判断是否进行写寄存器的操作，还有包括了将要写入 hi 和 lo 寄存器的值，如果不写，则此处为 0，且使能信号为 0。

3. 实验感受及改进意见

3.1 王艺茗

本次实验让我感受最深的就是入门难，这是在我目前上过的实验课中最难上手的一门，也许是这是因为是一门偏硬件的一门课，同时结合了写代码的一些工作，需要我们去设计流水线，去完成一条比较完整的流水线。其实经过了实验的过程，我感觉自己学到了很多东西，对流水线有了比较深的理解，在理论部分还没能特别理解，在实验后就有了额外的加深。

在实验的过程中，时常遇到困难，同时在改自己程序的 bug 中，也增长了自己的能力。在改 bug 的过程中，我学到了许多方法，需要自己耐心去看代码，去找每个变量的定义，同时最重要的是要去看波形图，一个时钟周期的去找自己发生错误的地方是哪里。当然错误的地方不一定是真正要改的地方，这在 C++代码中也有深刻的体会，所以 bug 的调试是我们都避不开的问题，要联系好上下文才能真正找的 bug 的错误来源。我认为我在调试 bug 的过程中找到了对实验更深的理解，一帆风顺反而不能学到太多的东西。

从一开始的小组作业开始，因为我们专业之前从来没有一门硬件方面的课，导致入手起来非常困难，我和组内成员，一边学习流水线，一边恶补相关知识，从一开始的接线，到过第一条指令我们用了很长时间，遇到了无数的困难，在老师和同学的帮助下一一克服，通过第一个点后，豁然开朗，后面的代码编写也更加轻车熟路，虽然也遇到乘法器添加等难题，也尽我们所能去完成，这也使我们对自己更有信心。

3.2 关君然

通过此次的 CPU 设计实验，让我们对 CPU 内部组成以及指令在 CPU 部件上如何运作有了一个更深的理解。老师在课堂上面讲课的时候感觉对于流水线不算很熟悉，但是明白一些东西，但是等到动手实操，自己完成一个简单的 CPU 的制作时，还是会遇到许多问题。

在实验过程中，我们小组遇到了各种各样的问题。在最一开始老师布置下来的 CPU 任务的时候，我们小组完全是懵的，虽因为 CPU 器件和指令运算只在课本上学习，从来没有真正实践过，现在需要自己设计 CPU 的各个部件，而且要将指令在器件上运行，感觉很复杂。虽然之前的一些小作业，我们也学会了如何使用 vivado 进行简单的半加器、全加器以及模 10 计数器的实现与仿真，但是如何从头开始进行 CPU 的设计对于我们来说，很难缕清这一团乱麻的源头。

但在接下来的日子，我们没有因为不

会而放弃，而是努力专心去设计好每个部件，对每个部件的功能进行模拟仿真，确保这一部件模块不出错，在设计过程中，感觉慢慢可以理清思路，也明白了下一步需要设计的东西。

通过此次实验，让我们对 CPU 有了更深的理解，而不只是纸上谈兵。其中 LW 指令以及 SW 指令的添加与理解、以及后面乘除法器的一些设计让我们都头疼了很久， 经过很长时间的摸索，终于找到了正确的解决方案，并且顺利的设计出合理的指令， 达到了预期的目的。

4. 参考资料

[1] 雷思磊.《自己动手做 cpu_雷思磊》[M/CD].

[2] 《“系统能力培养大赛”MIPS 指令系统规范_v1.01》[M/CD].