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：涂文钊

组员：马浩然

陶锦超

报告日期：2021.12.18

目录

一 组员工作量占比：

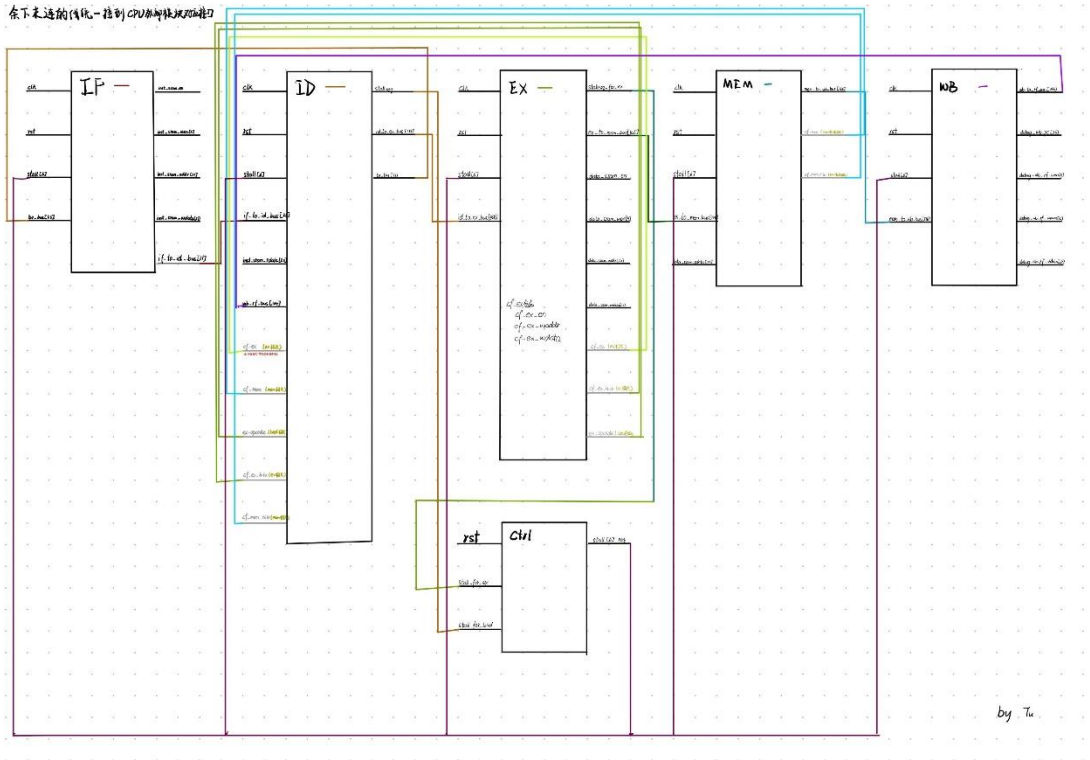
组长 涂文钊 45%

组员 马浩然 25% 陶锦超 30%

二 总体设计

在已有代码上，完善五段流水的 cpu,实现 MIPS 32 指令集基础指令。先添加数据旁路以处理 WAR 相关，再添加气泡处理逻辑解决 load 暂停问题，最后添加跳转处理代码，在 IF 段确定读入指令地址，并添加一些基础运算指令过 pass point 1。添加 hilo 寄存器处理乘法除法指令，为 hilo 寄存器添加数据旁路解决数据相关。最后对对几条 load 与 store 指令添加控制逻辑达到 pass point 64。

三 cpu 模块内部不同流水段连线图



线名标注在线段上，灰色的线实际上为多条线，由于画不下去了，有所缩略。图中未连接的线将接到 cpu 模块与外部连接的线路上，如 clk,rst 都为外部输入。

四 完成指令数与开发环境

完成指令数：共完成 51 条指令，分别为:

inst_ori, inst_lui, inst_addiu, inst_beq, inst_subu,
inst_jr,inst_jal, inst_addu, inst_bne, inst_sll, inst_or inst_sw,
inst_lw, 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_srl,inst_srlv,in
st_bgez,inst_bgtz,inst_srav, inst_blez, inst_bltz,
inst_bgezal,inst_bltzal,inst_jalr,
inst_mflo,inst_div,inst_divu,inst_mfhi ,inst_mult ,
inst_multu ,inst_mthi, inst_mtlo, inst_lb,inst_lbu,inst_lh,inst_lhu,
inst_sb,inst_sh

注：指令按 pass 点顺序添加,并不以功能排序

开发环境：Xilinx Vivado 2019.2_1106_2127

五 各段流水线的详细说明

IF 段:

1.整体功能说明：

IF 段主要负责向 cpu 外部模块发出读取指令的请求，控制向 inst_sram_en,inst_sram_wen,inst_sram_addr 与 inst_sram_wdata 输出的值。外部模块依据这四个值返回指令数据 inst_sram_rdata，其中存放着依据

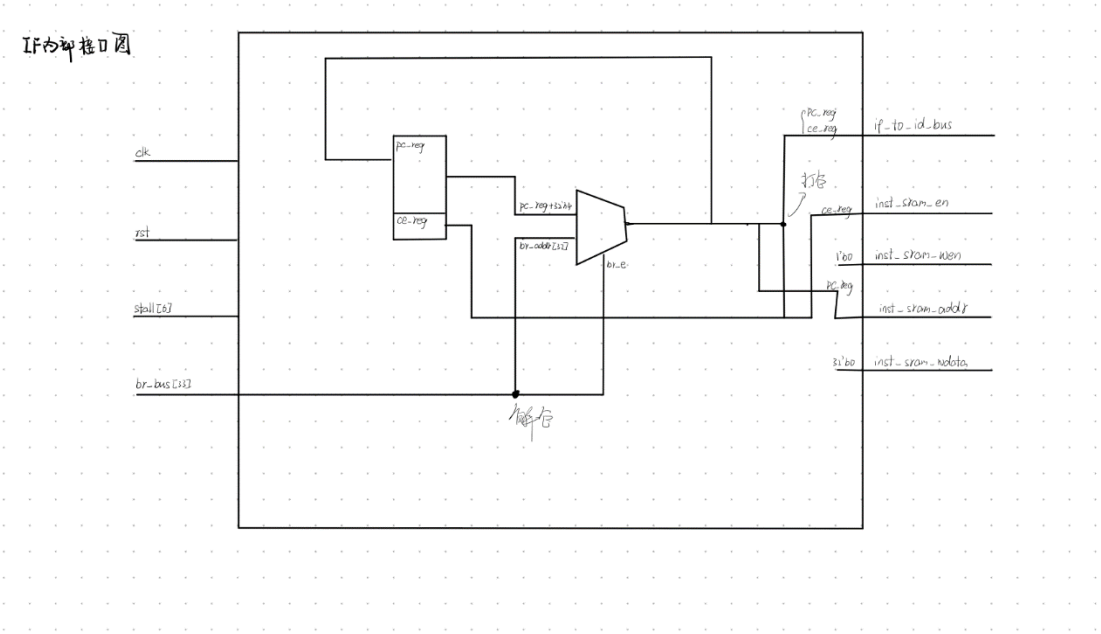
pc 值读取到的 32 位 MIPS 指令码。inst_sram_rdata 通过 cpu 内部连线直接输入到 id 模块的 inst_sram_rdata 中做处理。

由 ID 段输入的 33 位的 br_bus 将在 IF 段解包为 1 位的 br_e 与 32 位的 br_addr。其中 br_e 表示是否发生跳转，br_addr 则存放 id 段算出的跳转地址。IF 段内有一个两路选择器，使用 br_e 作为选择信号。若 br_e 为 0，则使用原 PC 寄存器内的地址加 4 作为新的 PC 地址（next_pc）绑定到 inst_sram_addr 中；若 br_e 为 1，则使用 br_addr 作为新的 PC 地址(next_pc)绑定到 inst_sram_addr 中。

最后是代码中处理好的 rst 与 stall 信号。若 stall[0]为`NOSTOP 则正常运转，将 next_pc 值存入 IF 段的 32 位 pc_reg 寄存器内，并将 1’b1 存放在寄存器 ce_reg 内。由于我们实现的功能里一般不需要将 IF 段暂停，所以并没有实现相应逻辑，有需要可以另行添加。rst 是复位信号,当外部模块将 rst 信号置为 1 时，将 pc_reg 内的值初始化为 32’hbfbf_fffc，将 ce_reg 内的值置为 0。

IF 段将当前 pc_reg 寄存器与 ce_reg 寄存器内的值打包通过 if_to_id_bus 传给 ID 段处理。虽然 id 段只是简单读入 ce 的值，并没有做处理（

2.大致结构图



ID 段:

1.整体功能说明

ID 段主要负责指令的译码，读写寄存器模块，处理数据相关，判断是否处理 load 的 stall,计算跳转地址等诸多功能，是流水线各段中功能最复杂的一段。

2.输入端口介绍

clk	外界输入的时钟信号
rst	外部输入的复位信号
stall	ctrl 模块发出的暂停气泡处理信号
if_to_id_bus	IF 段打包输入到 ID 段的信号
inst_sram_rdata	外界返回的 inst 指令
wb_to_rf_bus	WB 段写回到 ID 段寄存器的信号
cf_ex_we cf_ex_addr cf_ex_data	EX 段到 ID 段的数据旁路，解决指令数据相关问题
cf_mem_we cf_mem_addr cf_mem_data	MEM 段到 ID 段的数据旁路，解决指令数据相关问题
ex_opcode	EX 段返回的操作码，用于在 ID 段进行判断，如果是 load 类指令，则在 ID 段后插入气泡，暂停一周期
cf_ex_we_hi cf_ex_we_lo cf_ex_hi cf_ex_lo	EX 段到 ID 段的数据旁路，解决寄存器 HILO 数据相关问题
cf_mem_we_hi cf_mem_we_lo cf_mem_hi cf_mem_lo	MEM 段到 ID 段的数据旁路，解决寄存器 HILO 数据相关问题

3.输出端口介绍

stallreq	连接到 ctrl 模块的 stall_for_load 接口，如果为一则 ctrl 模块输出的 stall 信号为 00_0111,将 EX,MEM,WB 段暂停一周期。
id_to_ex_bus	ID 段打包发送给 EX 段的信号，包含以下数据： inst_lb, inst_lbu, inst_lh, inst_lhu,//这四个表示当前处理的指令是否为 load 类指令，将原封不动的传输到 MEM 段，判断从 sram 中获取的数据要如何写到 rt 寄存器中 inst_sb, inst_sh,//判断是否为 store 类指令，将原封不动传入到 EX 段，控制从 EX 段写入到 sram 中的数据。 inst_mult, inst_multu, inst_div, inst_divu,//乘除指令，控制 ex 段乘除法器的运作，指定写入到 HILO 寄存器的 id_we_hi, id_we_lo, id_hi, id_lo,//HILO 寄存器的控制读写信号，与从 HILO 寄存器读取出的值 id_pc, // 158:127 当前指令 PC 值 inst, // 126:95 当前指令码

	alu_op, // 94:83 指定 alu 运算器运算类型 sel_alu_src1, // 82:80 alu 源操作数 1 的多路选择指令 sel_alu_src2, // 79:76 alu 源操作数 2 的多路选择指令 data_ram_en, // 75 是否允许当前指令读写 sram 的判断信号 data_ram_wen, // 74:71 是否允许当前指令写 sram 的判断信号 rf_we, // 70 允许当前指令写寄存器的判断信号 rf_waddr, // 69:65 当前指令写寄存器的地址 sel_rf_res, // 64 选择写入寄存器的数据是来自 EX 段算出的数据还是 MEM 段读取的数据 rdata1, // 63:32 rs 寄存器的值 rdata2 // 31:0 rt 寄存器的值

br_bus	ID 段解码的跳转指令控制信号与跳转地址
	br_e:若当前指令为跳转指令且满足跳转条件时为 1
	`assign br_e = (inst_beq & rs_eq_rt)`
	`| inst_jr`
	`| inst_jal`
	`| (inst_bne & ~rs_eq_rt)`
	`| inst_j`
	`| (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)`
	`| inst_jalr`
	`; //`
	br_addr:根据指令操作码，在三类跳转地址选择一种。

3. 核心实现逻辑介绍

数据相关：

```
assign rdata1 = rs==cf_ex_addr && cf_ex_we ? cf_ex_data :
                rs==cf_mem_addr && cf_mem_we ?  cf_mem_data :
                rs==wb_rf_waddr && wb_rf_we ? wb_rf_wdata :
                rdata1_r;
assign rdata2 = rt==cf_ex_addr&&cf_ex_we ? cf_ex_data :
                rt==cf_mem_addr&&cf_mem_we ?  cf_mem_data :
                rt==wb_rf_waddr&&wb_rf_we ? wb_rf_wdata :
                rdata2_r;
assign id_hi = cf_ex_we_hi ? cf_ex_hi :
               cf_mem_we_hi ? cf_mem_hi :
               wb_we_hi ? wb_hi :
               hi_r;
assign id_lo = cf_ex_we_lo ? cf_ex_lo :
               cf_mem_we_lo ? cf_mem_lo :
               wb_we_lo ? wb_lo :
               lo_r;
```

以上代码主要实现解决数据相关的多路选择器的构建，若 ex 段指令算出的结果要写，且与当前 ID 段指令要读的寄存器相同，则用 ex 段算出的结果代替。若 mem 段指令的结果要写，且与当前 ID 段指令要读的寄存器相同，则用 mem 段算出的结果代替。若 wb 段指令的结果要写，且与当前 ID 段指令要读的寄存器相同，则用 wb 段算出的结果代替。若没有以上数据相关，就使用从寄存器中读取到的数据。

store 气泡：

```
wire load_stall;

assign load_stall = (( cf_ex_addr == rs) | ( cf_ex_addr == rt ));

assign stallreq = is_load & load_stall;
```

以上代码主要实现 load 插入气泡逻辑的处理。如果指令为 load 类型指令（is_load）且写入的地址与当前指令地址相同，向 ctrl 模块发出暂停信号。

HILO 寄存器：

```
module hilo(

input wire clk,

output wire [31:0] r_hi,
```

```
    output wire [31:0] r_lo,
    input wire we_hi,
    input wire we_lo,
    input wire [31:0] wb_hi,
    input wire [31:0] wb_lo
);
```

总体实现逻辑与 regfile 类似，将从 HILO 寄存器读取的值顺序传下去。在 ex，mem 段添加数据旁路解决数据相关问题。

EX 段:

1. 整体概述：

ex 段主要负责对 ID 段传来的数据进行运算，若传来的是 store 指令，ex 段负责依据运算得到的结果写出到 sram 中。

2.端口介绍：

```
input wire clk,//时钟信号
input wire rst,//复位信号
input wire [`StallBus-1:0] stall,//ctrl 传出的 stall 暂停控制信号
input wire [`ID_TO_EX_WD-1:0] id_to_ex_bus,//id 段打包发送给 ex 段的数据信号
output wire [`EX_TO_MEM_WD-1:0] ex_to_mem_bus, //ex 段打包发送给 mem 段的数据信号
output wire data_sram_en,//数据访问控制信号
output wire [3:0] data_sram_wen,//数据写访问信号
output wire [31:0] data_sram_addr,//数据读取地址
output wire [31:0] data_sram_wdata,//要写入的数据
//解决 ex 数据相关
output wire [4:0] cf_ex_addr,
output wire [31:0] cf_ex_data,
```

```
output wire cf_ex_we,

output wire [5:0] ex_opcode,

output wire cf_ex_we_hi,

output wire cf_ex_we_lo,

output wire [31:0] cf_ex_hi,

output wire [31:0] cf_ex_lo,

//处理 div 的时钟暂停

output wire stallreq_for_ex
```

3.核心逻辑介绍：

对源操作数的选择：

```
assign alu_src1 = sel_alu_src1[1] ? ex_pc :
                  sel_alu_src1[2] ? sa_zero_extend :
                  rf_rdata1;
assign alu_src2 = sel_alu_src2[1] ? imm_sign_extend :
                  sel_alu_src2[2] ? 32'd8 :
                  sel_alu_src2[3] ? imm_zero_extend :
                  sel_alu_src2[4] ? ex_lo_i :
                  sel_alu_src2[5] ? ex_hi_i:
                  sel_alu_src2[6] ? 32'b0:
                  rf_rdata2;
```

store 指令控制：

```
assign data_sram_wen =   inst_sb & alu_result[1:0] == 2'b00 ? 4'b0001:
                         inst_sb & alu_result[1:0] == 2'b01 ? 4'b0010:
                         inst_sb & alu_result[1:0] == 2'b10 ? 4'b0100:
                         inst_sb & alu_result[1:0] == 2'b11 ? 4'b1000:
                         inst_sh & alu_result[1:0] == 2'b00 ? 4'b0011:
                         inst_sh & alu_result[1:0] == 2'b10 ? 4'b1100:
                         data_ram_wen;
```

```
assign data_sram_wdata = data_sram_wen==4'b0001 ? {rf_rdata2[7:0],rf_rdata2[7:0],rf_rdata2[7:0],rf_rdata2[7:0]}:
                         data_sram_wen==4'b0010 ? {rf_rdata2[7:0],rf_rdata2[7:0],rf_rdata2[7:0],rf_rdata2[7:0]}:
                         data_sram_wen==4'b0100 ? {rf_rdata2[7:0],rf_rdata2[7:0],rf_rdata2[7:0],rf_rdata2[7:0]}:
                         data_sram_wen==4'b1000 ? {rf_rdata2[7:0],rf_rdata2[7:0],rf_rdata2[7:0],rf_rdata2[7:0]}:
                         data_sram_wen==4'b0011 ? {rf_rdata2[15:0],rf_rdata2[15:0]}:
                         data_sram_wen==4'b1100 ? {rf_rdata2[15:0],rf_rdata2[15:0]}:
                         rf_rdata2;
```

对乘除指令运算结果的选择：

```
assign ex_hi=ex_we_hi & (inst_div| inst_divu)? div_result[63:32] :
             ex_we_hi & (inst_mult| inst_multu)? mul_result[63:32] :
             ex_we_hi ? alu_result :
             ex_hi_i;
assign ex_lo=ex_we_lo & (inst_div | inst_divu) ? div_result[31:0] :
             ex_we_lo & (inst_mult | inst_multu) ? mul_result[31:0] :
             ex_we_lo ? alu_result :
             ex_lo_i;
```

MEM 段:

1.整体介绍：

mem 段要做的事就很少了，处理数据相关逻辑，依据从 id 段一路传回的 lb,lub,lh,lhu 四条指令与 ex 段运算得到的结果进行选择，控制读入数据的处理方式。

2.端口介绍：

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

```
input wire [`EX_TO_MEM_WD-1:0] ex_to_mem_bus,
input wire [31:0]data_sram_rdata,//读取到的数据
output wire [`MEM_TO_WB_WD-1:0] mem_to_wb_bus,
//解决 mem 段数据相关问题
output wire [4:0] cf_mem_addr,
output wire [31:0] cf_mem_data,
output wire cf_mem_we,
output wire cf_mem_we_hi,
output wire cf_mem_we_lo,
output wire [31:0] cf_mem_hi,
output wire [31:0] cf_mem_lo
```

3.核心功能介绍：

依据指令与读取地址，决定读取方式

```
wire [3:0] data_sram_ren;
assign data_sram_ren =    inst_lb & ex_result[1:0] == 2'b00 ? 4'b0001:
                          inst_lb & ex_result[1:0] == 2'b01 ? 4'b0010:
                          inst_lb & ex_result[1:0] == 2'b10 ? 4'b0100:
                          inst_lb & ex_result[1:0] == 2'b11 ? 4'b1000:
                          inst_lh & ex_result[1:0] == 2'b00 ? 4'b0011:
                          inst_lh & ex_result[1:0] == 2'b10 ? 4'b1100:
                          inst_lbu & ex_result[1:0] == 2'b00 ? 4'b0001:
                          inst_lbu & ex_result[1:0] == 2'b01 ? 4'b0010:
                          inst_lbu & ex_result[1:0] == 2'b10 ? 4'b0100:
                          inst_lbu & ex_result[1:0] == 2'b11 ? 4'b1000:
                          inst_lhu & ex_result[1:0] == 2'b00 ? 4'b0011:
                          inst_lhu & ex_result[1:0] == 2'b10 ? 4'b1100:
```

```
                                    4'b1111;
```

依据 data_sram_ren 里的值判断 mem 写入逻辑：

```
assign mem_result =     inst_lbu & data_sram_ren==4'b0001 ? {24'b0,data_sram_rdata[7:0]}:
                        inst_lbu &  data_sram_ren==4'b0010 ? {24'b0,data_sram_rdata[15:8]}:
                        inst_lbu &  data_sram_ren==4'b0100 ? {24'b0,data_sram_rdata[23:16]}:
                        inst_lbu &  data_sram_ren==4'b1000 ? {24'b0,data_sram_rdata[31:24]}:
                        inst_lhu &  data_sram_ren==4'b0011 ? {16'b0,data_sram_rdata[15:0]}:
                        inst_lhu &  data_sram_ren==4'b1100 ? {16'b0,data_sram_rdata[31:16]}:
                        inst_lb &  data_sram_ren==4'b0001 ? {{24{data_sram_rdata[7]}},data_sram_rdata[7:0]}:
                        inst_lb &  data_sram_ren==4'b0010 ? {{24{data_sram_rdata[15]}},data_sram_rdata[15:8]}:
                        inst_lb &  data_sram_ren==4'b0100 ? {{24{data_sram_rdata[23]}},data_sram_rdata[23:16]}:
                        inst_lb &  data_sram_ren==4'b1000 ? {{24{data_sram_rdata[31]}}, data_sram_rdata[31:24]}:
                        inst_lh &  data_sram_ren==4'b0011 ? {{16{data_sram_rdata[15]}},data_sram_rdata[15:0]}:
                        inst_lh &  data_sram_ren==4'b1100 ? {{16{data_sram_rdata[31]}},data_sram_rdata[31:16]}:
                        data_sram_rdata;
```

这是一个多路选择器，其中指令条件可以不加，此处为了看到清晰加入到条件中。

WB 段:

1. 整体功能介绍：

wb 段的功能最为简单，将前面传来的数据写入 debug 线内方便 debug，并将相应的值传回到 ID 段。（除 HILO 寄存器走线外，没有添加什么逻辑，不做介绍）

六 实验心得体会

马浩然：在本次实验中，我们学习了如何编写简单的 CPU，并进行指令的添加和数据相关的处理。对此次实验，我深有感想和体会，同时我也认识到了我在计算机学习方面的不足。

我们小组首要面临的问题就是 point 1 的编写。前面在理论课上老师已经给我们讲解了有关流水线的基础知识，所以我们认为虽然对 Verilog 并不是很熟悉，但通过自己的学习和理解，也是可以有所进展。接着我们就借助《Verilog 语法基础》PPT 中讲解的内容，进行对 Verilog 语言的初步了解，学习其中对变量的定义、组合逻辑、操作符等内容。根据所给的示例代码，一边尝试进行理解，一边学着其代码格式进行自我编写，明白每个变量和句子的含义。最终在助教和老师的帮助下，我们成功完成了 point 1 的编写，为接下来的进展打好基础。

我负责的是在 point 1-36 中添加部分指令。我通过《自己动手做 CPU》这本书中第一条指令 ori 的实现，尝试增添其他指令。每通过一条指令，我就将下一条指令的二进制码复制下来，找到下一条我该添加的指令，得知其指令格式和运算逻辑。有些指令的实现需要进行条件判断，因此我需要添加额外的 wire 变量来判断。做实验的过程少不了组长的帮助，在他的点拨下，那些较为复杂的指令也成功编写。

最开始添加指令时，我完全不明白要如何操作。我仅仅照着书中的内容，按部就班地完成一个较为简单的指令添加。然而，当我费劲地完成第一条指令添加后，我发现这只是一个机械式重复罢了，我仍然不明白其中的原理，在编写中我没有任何自己的想法。这让我在后续添加中遇到了诸多挫折。因此，我当时试着添加几条语句，尽管他们是错误的，但这突破性的尝试，让我在后续的指令编写中能够明白其中的原理，从而成功编写后续指令。

在整个实验中我认识到了很多。实验的成与败并不重要，当然当我们成功的完成实验的时候，那也是一份难得的快乐！但是我们仍然在许多细节上出错了，所以我们在实验过程中，我们应该尽量减少操作的盲目性提高实验效率的保证，不要过于着急。实验中我就是犯了一个的错误，我添加指令时，误将第一个与第二个读寄存器端口要读取的寄存器地址填反了，导致我们小组检查时，浪费了较长时间。这应该是不认真导致的后果，在实验中极为需要的是我们认真严谨、大胆、自信，还有团队合作精神。

这次实验我们受益匪浅，不求以后能有多大的改进，只求自己能坚持每天都在进步，哪怕一点点就好。同时也希望能在今后的课程中学到更多，熟练地掌握所学的知识，并应用于日常生活当中。

陶锦超：在实现的代码的过程中，因为事先提供的代码并没有完成数据相关方面的处理，所以为了解决这一部分，我们花费了大量时间。而在解决这一问题后，我参与的指令添加环节，让我在实践中对计算机体系结构有了更加深

刻的理解。由于CPU相关指令涉及到寄存器、内存，所以在添加指令的时候，我们需要理解寄存器、内存相关数据的存储形式，存取规范等。尽管在课程学习时我们了解了不同架构设计的规范不一样，但是本次实验采用的是常见的MIPS32指令集架构，对应的寄存器也是常见的32个通用寄存器，其中0位寄存器一般作为常量0。实际使用时，还会有些需要遵循的一系列约定，这是课程学习时没有提到的约定。事实上，硬件上没有强制指定寄存器的使用规则。而在一些特殊功能实现时，我们还会考虑到特殊的寄存器，比如PC(Program Counter程序计数器)、HI(乘除结果高位寄存器)、LO(乘除结果低位寄存器)。这是因为在实现乘除法器时，我们需要利用到HI和LO保存乘除法运算的结果。实际操作时，乘法利用到HI，LO分别存储结果高32位和低32位；除法利用HI，LO分别存储结果余数和商。这在实验中添加乘除法器是最要紧的前提。而添加指令的环节中，也加深了我对指令的理解。我们会给具体指令集中的指令规定固定的格式，以便让译码器理解代码，CPU译码的过程相当于就是查字典，在规定的有限的指令集内找到与输入的机器码相匹配的情况，并执行相应的操作。而为了方便指令的添加，我们需要对字节序列有着清楚的了解，指令的每个部分所占的字节数以及所在位置，都是该指令最关键的特征。不过好在，本实验的预处理中，已经将32位的指令长度每个会使用的区域都进行的事先的声明和定义，后续的操作中主要是利用各指令的不同，而进行添加和使用。这让我们很清晰的看到了一条指令的定义过程，揭秘了在CPU运行时，复杂系统的每一次调用的“魔法”。但是困难随着自己的理解也迫近而来，真实的指令由于种类不同，实现调用的功能部件不一样，导致往往在添加几条指令后，便会遇见新的困难。实验中我们添加了逻辑操作指令、移位操作指令、移动操作指令、算数操作指令、转移指令、加载存储指令等。

在实验中，我们不仅强化了课堂中学习的计算机体系结构的知识，还在实际操作中，掌握了一些软件和语言的使用，在本次实验中，我们运用Verilog语言进行简单CPU的代码实现，对于Verilog语言有了一些基础的了解和使用，实际上Verilog语言本身就是在C语言的基础上发展而来的，语法结构上继承了很多C语言的语法结构，所以在学习使用Verilog时，还是很容易上手的。但是在使用vivado进行仿真调试时为了看懂波形图着实费了些力气。不过好在结合书本知识，了解CPU按照时钟频率工作，而时钟会产生的一个周期信号，所以结合每一个时钟周期以及提供的test实际操作流程，就可以看懂真实运行时在哪一步指令出现的错误，以方便准确的定位到错误的发生位置。而正是在实验中，可以看到的时钟周期波形图的变化，让我们对五段流水线的理解大大加深。课程中学习的取指、译码、执行、访存、回写步骤，ALU（Arithmetic Logic Unit）算术逻辑单元，RAM等等都可以成为我们键盘下的

代码，而理解他们其中的起承转合正是课程想要为我们讲授的知识，想要为我们构建的计算机体系结构、计算机系统的知识逻辑框架。

罗马非一日建成。每一个复杂的功能部件都来自每一行代码的堆砌。而作为新时代的计算机、人工智能专业的学生，不积跬步无以至千里，不积小流无以成江海，千里之行要始于足下。从复杂的处理器的每一小处入手，就是需要一步一步，一点一点实现。而不能望难生畏，利用简单的地方入手，逐步的去增加部件，完善设计，丰富内容，通过每一次修改代码，补充代码和不停地修改代码，不停地补充代码，一行一行的去书写调试，利用每次设立的小目标，汇聚团队的力量去实现一个较庞大复杂的设计。

涂文钊：在这次历时近 4 周的实验中，我们组经历了许多挫折，小小的数据旁路，我们组就加错 5 个版本。理论与工程的冲突，verilog 教学与具体仿真的冲突都让我们备受煎熬。但我们组做出来了，通过实验报告回顾发现，曾经阻拦我们的高峰大泽，回顾起来反而异常简单。从早上试错到深夜，与组员们一起攻克的经历，现在回忆起来像是闪烁着星光。数据旁路，stall 气泡，HILO,乘除处理，load,store 信号处理。一步步走来，我们对加线愈发熟悉（，对 cpu 的实现也更有兴趣。乘除法器的实验，mmu,cache,TLB，甚至是超标量，指令发射，条件预测。新的领域向我们打开。可惜，相聚的时光总是短暂，更多的内容要留待后言了。