实验报告成绩:	成绩评定日期:

2022 ~ 2023 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（一）2002

组长：袁云

组员：姜昕敏

报告日期：2023.1.2

目录

1.概述

1.1 工作量分配

此次计算机系统的实验成功通过了64个测试点。姜昕敏同学主要负责数据相关、流水线暂停处理、1-36指令的添加、跳转指令和一些基本指令的添加部分。袁云同学主要负责数据相关、36-64指令的添加、访存指令、流水线的控制、HILO寄存器的设计以及一些基本指令的添加部分。

1.2 总体设计

将MIPS流水CPU整体的处理过程分为六部分，其中五部分为流水线的基本组成单元，即取指令阶段（IF）、指令译码阶段（ID）、指令执行阶段（EX）、存储器访问阶段（MEM）和寄存器回写阶段（WB）五级处理过程，并确定各控制部件的控制信号逻辑，即控制段（Ctrl）。同时进行调试，检验各部件功能是否能正常运行，总结解决问题的经验。

IF段根据是否有跳转给下一个PC赋值，并将其值作为地址传给存储器，使存储器向ID段返回32位inst指令，同时将取指令信号变为真。

ID段的主要功能是解析指令，访存寄存器以及计算跳转指令的地址。ID段从内存接收到指令数据流之后，将指令按位拆分并判断指令的具体类别。

EX段的主要功能是进行各种运算和向存储器发出存取信号。EX段在接收到ID段的信号之后，将基本操作组合的alu_op信号、操作数1和操作数2传入内部alu模块进行具体的运算。

MEM段的主要功能是取回访存的结果，获取从存储器读取到的数据，并将数据进行处理得到需要存回寄存器的值，最后将各类寄存器的读写使能信号、地址和写入数据合并为总线传入WB段。

WB段的主要功能是将结果写入通用寄存器堆。它起到了数据传递的作用。

Ctrl段控制流水线的暂停处理，ID段和EX段发送暂停请求，由Ctrl段接收后对来自不同段的暂停请求给予不同的暂停信号stall，并传递给流水线的各个部分，使其按照stall信号进行暂停操作。

1.3 不同流水段之间的连线图

不同流水段之间的连线图如图1所示。

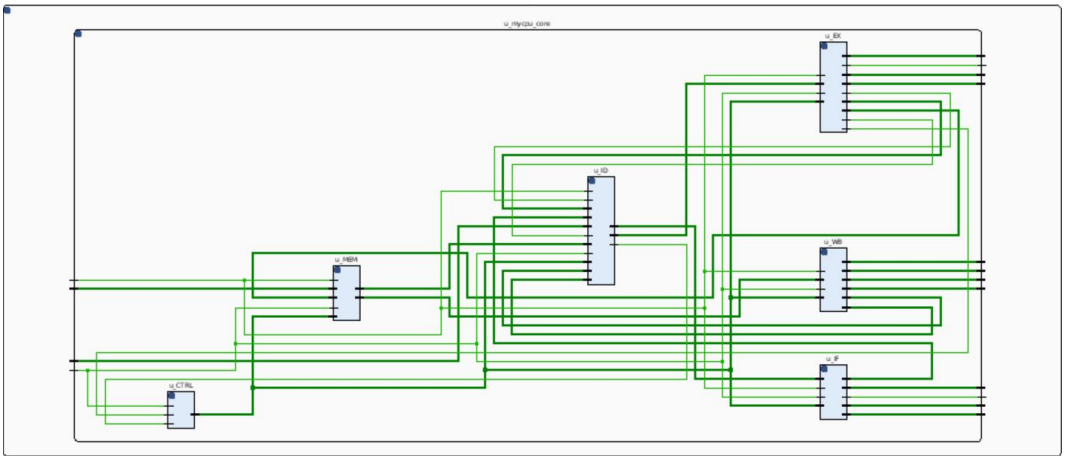

图 1

1.4 完成的指令

完成指令共 52 条，包括算术运算：ADD，ADDI，ADDU，ADDIU，SUB，SUBU，SLT，SLTI，SLTU，SLTIU，DIV，DIVU，MULT，MULTU；逻辑运算：AND，ANDI，LUI，NOR，OR，ORI，XOR，XORI；移位指令：SLLV，SLL，SRAV，SRA，SRLV，SRL；分支跳转：BEQ，BNE，BGEZ，BGTZ，BLEZ，BLTZ，BGEZAL，BLTZAL，J，JAL，JR，JALR；数据移动：MFHI，MFLO，MTHI，MTLO；访存指令：LB，LBU，LH，LHU，LW，SB，SH，SW。

1.5 程序运行环境及使用工具

本次实验主要在老师为我们在 CG 平台上搭建的服务器里进行，利用了 vivado19.2 和 VScode 等相关软件环境。

2. 单个流水段说明

2.1 IF 段

2.1.1 整体功能

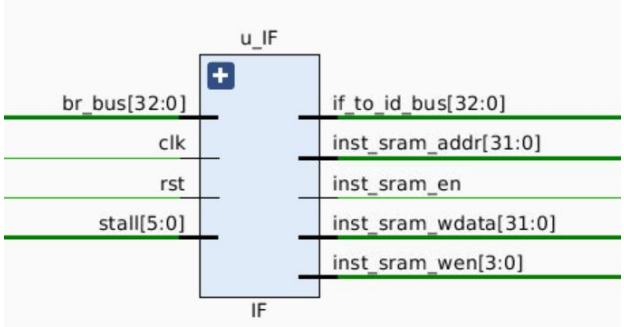

图 2

2.1.1.1 跳转指令

在信号中，建立 pc_reg 存放当前的 pc 指令地址，next_pc 用于存放经过运算后进行跳转的下一条指令地址。首先 IF 段接收从 ID 段传来的跳转指令数据 br_bus。br_bus 中的 br_e 为跳转使能信号，如果为 1，则将 next_pc 中的值赋为跳转后的指令地址；否则，进行正常取指令操作，即 pc_reg + 4。相关代码如下。

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

2.1.1.2 传输指令地址

IF 段的 output 输出是将 PC 的值作为地址发送给存储器，使存储器根据接收到的 PC 地址向 ID 段返回 32 位 inst 指令，同时将 PC 值传给 ID 段方便后面指令的使用。ce_reg 表示存储器取指令使能信号；pc_reg 表示 ID 段当前指令对应的地址。如果需要插入气泡，则 ce_reg 为 0，即在之后不需要进行存储器取指令。相关代码如下。

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

分别对输出到存储器中的信号进行赋值，如 inst_sram_en 中存储着 ce_reg，如果 ce_reg 为零，则说明进行了插入气泡的操作，不需要对存储器进行处理，否则使能信号为 1，进行传出。存储器中的指令地址为当前 pc 指令地址。数据都默认设为 0。

```
assign inst_sram_en    =  ce_reg;
assign inst_sram_wen   =  4'b0;
assign inst_sram_addr  =  pc_reg;
assign inst_sram_wdata =  32'b0;
```

2.1.2 具体信息

2.1.2.1 输入信号

clk 表示时钟信号；rst 表示复位信号；stall 表示 Ctrl 控制的暂停信号；br_bus 表示 ID 传输来的用于判断是否需要进行跳转的指令信号；br_addr 表示跳转指令操作之后所存放的跳转地址。br_bus 是从 ID 段接收的信号，其中 br_e 为跳转使能信号，如果 br_e 为 1，则将 next_pc 中的值赋为跳转后的指令，否则进行正常取指令操作。

```
assign {
        br_e,
```

```
        br_addr
    } = br_bus;
```

2.1.2.2 输出信号

ce_reg 表示存储器取指令使能信号，在经过是否需要插入气泡的判断之后进行赋值；pc_reg 表示存储器取值指令值；next_pc 表示用于计算的下一条 pc 指令的值，经过 br_e 的判断后，next_pc 的值由跳转指令地址或者是当前指令值+4 构成。

```
assign {
        ce_reg,
        pc_reg
} = if_to_id_bus;
```

2.2 ID 段

2.2.1 整体功能

ID 段接口结构如图 3 所示。

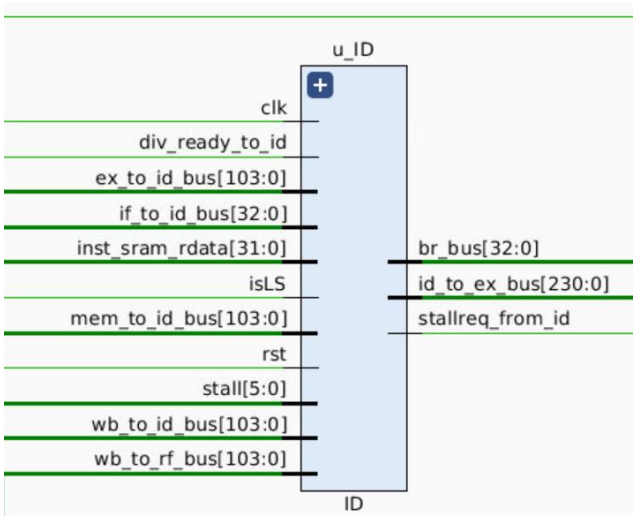

图 3

2.2.1.1 译码

首先对于 32 位的指令进行拆解方便后续指令操作，然后将指令的使能信号设置为能被唯一识别的编码，如 ORI 指令只要求 opcode 段为 00_1101 即可。这样当一条指令被 ID 段接收到之后，就可以只使对应的指令使能信号变为 1，完成译码操作。

2.2.1.2 操作数内容、指令基本操作信号、寄存器存储信号

指令的基本操作类型包括：ADD，SUB，SLT，SLTU，AND，NOR，OR，XOR，SLL，SRL，SRA，LUI。将它们作为使能信号并绑定在 alu_op 的总线中，传入 EX 段中进行运算类型的判断。

寄存器存储信号包括寄存器存储使能信号 rf_we，存入 rd 寄存器使能 sel_rf_dst[0]，存入 rt 寄存器使能 sel_rf_dst[1]，存入 31 号寄存器使能

sel_rf_dst[2]，通过运算得到最终的存入寄存器的地址。在每个可能的信号来源处添加使能信号，不同类别的指令点亮不同的使能信号，通过 sel_alu_src1 和 sel_alu_src2 两条线路传入 EX 段进行下一步运算操作。

```
assign rf_waddr = {5{sel_rf_dst[0]}} & rd
                | {5{sel_rf_dst[1]}} & rt
                | {5{sel_rf_dst[2]}} & 32'd31;
```

2.2.1.3 寄存器访存与数据相关

寄存器模块被设置在 ID 段内部，其中包括 32 个 32 位寄存器，1 个 32 位 HI 寄存器和 1 个 32 位 LO 寄存器。寄存器模块与 ID 连接的线路如图 4 所示。

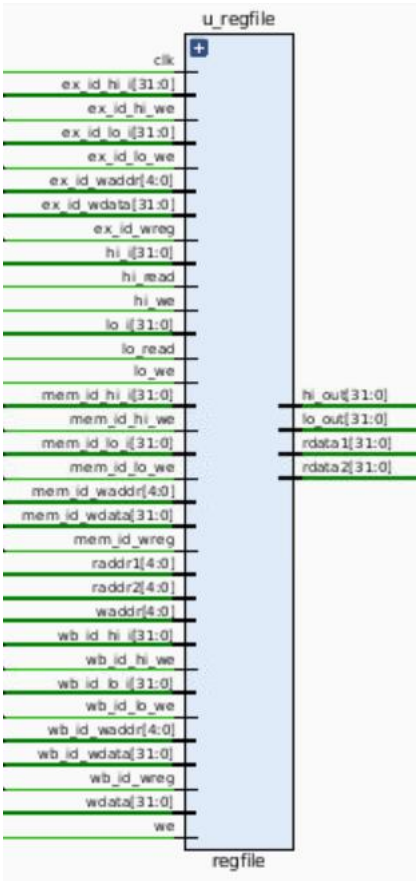

图 4

寄存器模块主要解决的问题为寄存器数据的取存和数据相关问题，其中有关寄存器数据存入的线路有写使能信号 we，写入数据 wdata，写入地址 waddr，HI 寄存器写使能 hi_we，HI 寄存器写入数据 hi_i，LO 寄存器写使能 lo_we，LO 寄存器写入数据 lo_i，该部分只需正常给对应地址赋值即可。在寄存器数据读取时会遇到数据相关问题，EX 段结果是否要存入寄存器的信号 ex_id_wreg、MEM 段结果是否要存入寄存器的信号 mem_id_wreg、WB 段结果是否要存入寄存器的信号 wb_id_wreg 如果为 1，则它们需要在 WB 段之后才能存入寄存器内，若在此之前需要取它们的结果就会导致 RAW 数据相关，此处采用 forwarding 技术可以部分解决该问题。读取数据时，若读取的地址恰好与 EX、MEM 或者 WB 段要写回的

寄存器地址相同，同时它们的写回寄存器的使能信号为真，这时就可以不从寄存器中取值，直接将 EX、MEM 或者 WB 段要存回的值赋给取值信号，如取 rs 寄存器数据部分操作如下。

```
assign rdata1 = (raddr1 == 5'b0) ? 32'b0 :
 ((ex_id_wreg==1'b1)&&(ex_id_waddr==raddr1))?ex_id_wdata:
 ((mem_id_wreg==1'b1)&&(mem_id_waddr==raddr1))?mem_id_wdata:
 ((wb_id_wreg==1'b1)&&(wb_id_waddr==raddr1))?wb_id_wdata:     reg_array[raddr1];
```

2.2.1.4 跳转指令

与跳转指令有关的主要信号有：跳转使能信号 br_e，跳转地址 br_addr，非跳转下一条指令地址 pc_plus_4。当有指令是跳转指令且符合条件时，跳转使能信号就会变为 1，并根据指令的跳转规则给 br_addr 赋值。如 beq 跳转指令，若寄存器 rs 的值等于寄存器 rt 的值，则转移，否则顺序执行。转移目标由立即数 offset 左移 2 位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的 PC 计算得到。当指令的 opcode 段与 000100 匹配，则 beq 指令被点亮。若此时从 rs 取出的值 rdata1 和 rt 取出的值 rdata2 也相等，则 br_e 跳转使能信号将会被点亮。对 offset 左移两位也就相当于在其后面拼接两个 0，有符号扩展就是按照最高位进行填充，最后将三部分拼接得到{{14{inst[15]}}，inst[15：0]，2'b0}，再与 pc_plus_4 相加得到跳转后的指令地址 br_addr。

```
assign inst_beq     = op_d[6'b00_0100];
assign rs_eq_rt = (rdata1 == rdata2);
assign br_e = (inst_beq & rs_eq_rt)|...
assign br_addr = inst_beq ? (pc_plus_4 + {{14{inst[15]}}, inst[15：0], 2'b0}) :
```

2.2.1.5 请求暂停

当涉及到 LW 指令时，由于访问存储器需要在 EX 段发出访问地址，在 MEM 段才能返回数据，这就导致数据无法及时地送回到 ID 段，此时就需要加入暂停信号。首先在 EX 段判断当前的指令是否为 LW，若指令为 LW，让 isLS 信号值为 1 并传回 ID 段。ID 段接收该值之后，再判断当前指令读取的寄存器地址是否与 LW 执行完之后需要存入的寄存器地址相同。如果两个地址相同，由于此时 LW 取存储器的值还未存入寄存器，会导致读取数据错误，所以要将请求暂停信号 stallreq_from_id 变为 1 并传给 CTRL 段。

```
assign stallreq_from_id=(isLS&((rs==ex_id_waddr)|(rt==ex_id_waddr)))? `Stop: `NoStop;
```

2.2.2 具体信息

2.2.2.1 输入信号

clk 表示时钟信号；rst 表示复位信号；stall 表示 Ctrl 控制的暂停信号；div_ready_to_id 判断当前是否需要因除法运算而暂停周期；isLS 判定 ID 当前

周期是否需要申请暂停。if_to_id_bus 由 IF 段传到 ID 段，ce 表示存储器取指令使能信号，id_pc 表示 ID 段当前指令对应的地址。

```
   assign {
       ce,
       id_pc
} = if_to_id_bus_r;
```

inst_sram_rdata 表示存储器根据 IF 段提供的 PC 地址取出的 32 位指令；ex_to_id_bus，mem_to_id_bus，wb_to_id_bus 都是利用 forwarding 技术为了解决数据相关问题设置的。ex_to_id_bus 由 EX 段传到 ID 段，其中的 ex_id_wreg 表示 EX 段计算出的结果是否需要存入寄存器；ex_id_waddr 表示 EX 段计算的结果需要存放在寄存器中的地址；ex_id_wdata 表示 EX 段计算结果的值；ex_id_hi_we 表示 EX 段计算出的结果是否要存入 HI 寄存器；ex_id_lo_we 表示 EX 段计算出的结果是否要存入 LO 寄存器；ex_id_hi 表示 EX 段计算出结果的数值；ex_id_lo 表示 EX 段计算出结果的数值。

```
   assign {
       ex_id_wreg,
       ex_id_waddr,
       ex_id_wdata,
       ex_id_hi_we,
       ex_id_lo_we,
       ex_id_hi,
       ex_id_lo
}=ex_to_id_bus;
```

mem_to_id_bus 由 MEM 段传到 ID 段，其中的 mem_id_wreg 表示 MEM 段计算出的结果是否需要存入寄存器；mem_id_waddr 表示 MEM 段计算的结果需要存放在寄存器中的地址；mem_id_wdata 表示 MEM 段计算结果的值；mem_id_hi_we 表示 MEM 段计算出的结果是否要存入 HI 寄存器；mem_id_lo_we 表示 MEM 段计算出的结果是否要存入 LO 寄存器；mem_id_hi 表示 MEM 段计算出结果的数值；mem_id_lo 表示 MEM 段计算出结果的数值。

```
  assign {
       mem_id_wreg,
       mem_id_waddr,
       mem_id_wdata,
       mem_id_hi_we,
       mem_id_lo_we,
       mem_id_hi,
       mem_id_lo
   }=mem_to_id_bus;
```

wb_to_id_bus 由 WB 段传到 ID 段，其中的 wb_id_wreg 表示 WB 段计算出的结果是否需要存入寄存器，wb_id_waddr 表示 WB 段计算的结果需要存放在寄存

器中的地址；wb_id_wdata 表示 WB 段计算结果的值；wb_id_hi_we 表示 WB 段计算出的结果是否要存入 HI 寄存器；wb_id_lo_we 表示 WB 段计算出的结果是否要存入 LO 寄存器；wb_id_hi 表示 WB 段计算出结果的数值；wb_id_lo 表示 WB 段计算出结果的数值。

```
   assign {
       wb_id_wreg,
       wb_id_waddr,
       wb_id_wdata,
       wb_id_hi_we,
       wb_id_lo_we,
       wb_id_hi,
       wb_id_lo
   }=wb_to_id_bus;
```

wb_to_rf_bus 由 WB 段传入 ID 段，将之前计算的结果存入对应寄存器，其中的 ex_rf_hi_we 表示 HI 寄存器存入使能；ex_rf_lo_we 表示 LO 寄存器存入使能；ex_rf_hi 表示 HI 寄存器存入的数据；ex_rf_lo 表示 LO 寄存器存入的数据；wb_rf_we 表示普通寄存器存入使能；wb_rf_waddr 表示普通寄存器存入地址；wb_rf_wdata 表示存入数据。

```
assign {
       ex_rf_hi_we,
       ex_rf_lo_we,
       ex_rf_hi,
       ex_rf_lo,
       wb_rf_we,
       wb_rf_waddr,
       wb_rf_wdata
   } = wb_to_rf_bus;
```

2.2.2.2 输出信号

br_bus 由 ID 传出到 IF 段，起到下一条 PC 值跳转的作用；br_e 表示 PC 跳转使能信号，br_addr 表示 PC 跳转地址。

```
   assign br_bus = {
       br_e,
       br_addr
};
```

id_to_ex_bus 由 ID 段传出到 EX 段，其中的 data_ram_readen 表示区分不同类别的访存指令；hi_write 表示 HI 寄存器写入信号；lo_write 表示 LO 寄存器写入信号；hi_read 表示 HI 寄存器读取信号；lo_read 表示 LO 寄存器读取信号；hi_out_file 表示 HI 寄存器读取的数据；lo_out_file 表示 LO 寄存器读取的数据；id_pc 与 IF 传入的值相同，表示当前指令对应的地址；inst 表示当前 32 位指令；alu_op 表示基本运算类型的综合；sel_alu_src1 表示操作数 1 的类

型，rs、pc、sa 无符号扩展；sel_alu_src2 表示操作数 2 的类型，rt、imm 有符号扩展、imm 无符号扩展、数字 8 的 32 位表示；data_ram_en 表示存储器读取使能；data_ram_wen 表示存储器写使能；rf_we 表示寄存器存储使能；rf_waddr 表示寄存器写入地址；sel_rf_res 表示寄存器写入类别，rt、rd、31 号寄存器；rdata1 表示从寄存器取出的 rs 的值；rdata2 表示从寄存器取出的 rt 的值。

```
assign id_to_ex_bus = {
         data_ram_readen,
         hi_write,
         lo_write,
         hi_read,
         lo_read,
         hi_out_file,
         lo_out_file,
         id_pc,          // 158: 127
       inst,           // 126: 95
       alu_op,         // 94: 83
       sel_alu_src1,   // 82: 80
       sel_alu_src2,   // 79: 76
       data_ram_en,    // 75
       data_ram_wen,   // 74: 71
       rf_we,          // 70
       rf_waddr,        // 69: 65
       sel_rf_res,       // 64
       rdata1,          // 63: 32
       rdata2           // 31: 0
};
```

2.3 EX 段

2.3.1 整体功能

在 EX 段中，主要进行运算操作以及向存储器发出存取信号，同时还会向下一层 MEM 段传输 HILO 相关信号。首先 EX 的 input 为 id_to_ex_bus 和 stall 信号，ID 段中判断指令是否需要进行添加气泡操作，然后将此使能信号传送给 EX 段，EX 段接受从 ID 段传输过来的解析过后的指令以及是否 stall 的使能信号。id_to_ex_bus 信号中包含了 ID 解析完指令后的所有数据以及与 HILO 寄存器之间的信号。EX 的 output 输出接口主要为 EX 段向 MEM 段发送的数据，向 ID 段传送的数据，存储器返回数据 data_sram 的相关数据以及一些是否进行乘除法和添加气泡的使能信号传输。

2.3.1.1 alu 模块

运算部分主要依赖于 EX 段内嵌的 alu 模块，模块接口如图 5 所示。

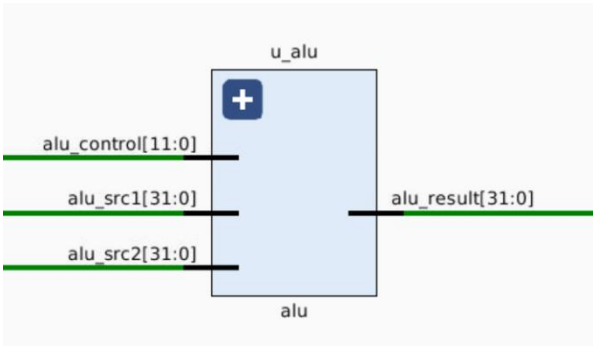

图 5

alu_src1 和 alu_src2 分别为两个操作数，alu_control 由 ID 段的 alu_op 而来，存储了 12 种基本指令的使能信号，只有其中一个为 1。由于从 ID 段传来的只有 sel_alu_src1 和 sel_alu_src2 两个使能信号组，所以在 EX 段需要判断 alu_src1 和 alu_src2 应当存储的内容，可以通过选择器进行选择。

```
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 : rf_rdata2;
```

由于 imm 属于操作数 2 alu_src2 的内容，所以只需要取 alu_src2 的低 16 位与 16 个 0 拼接即可。

```
assign lui_result = {alu_src2[15: 0], 16'b0};
```

算完结果后，用拆解出来所有的使能信号与其对应结果进行与运算，这样就只有当使能信号为真的时候计算结果才有效，再将这些与运算之后的结果进行或运算得到最后的 alu_result 由 alu 模块传回 EX 段。

2.3.1.2 内存访问信号传递

接口如图 6 所示。

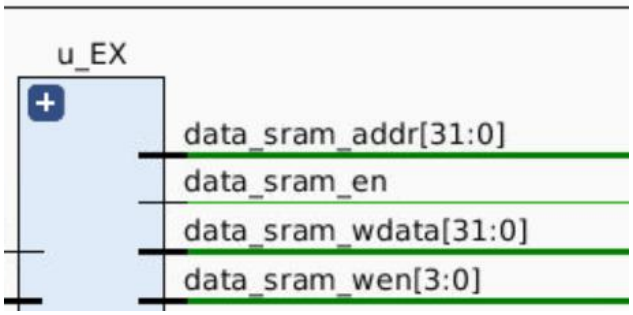

图 6

EX 段向内存发送的信号包括内存读取使能 data_sram_en，内存写入使能 data_sram_wen，内存访问地址 data_sram_addr，内存写入数值 data_sram_wdata。内存读取使能信号由 ID 段的 data_ram_en 传来，直接赋值即可。内存写使能信号的值与具体指令和 EX 段计算结果的最后两位有关。内存访问地址便是 alu 模块计算出的结果。内存写入数值与内存写入信号的类型有关。

2.3.1.3 请求暂停

EX 段的暂停请求来源于乘除法操作，由于该操作执行时间多于一个周期，所以当前指令为乘除法且乘除法准备信号为 0 时就需要请求暂停。

2.3.1.4 数据相关

EX 段有最新得到的 ex_result，hi_ex 和 lo_ex 的运算结果，为了尽快将数据提供给后续要取该寄存器内容指令使用，需要添加 ex_to_id_bus 总线。由于需要知道该数据最后属于哪一个寄存器以及是否有效，所以在总线中必须要有寄存器地址和使能信号。ex_to_id_bus 内容如下。

```
assign ex_to_id_bus={
        rf_we,
        rf_waddr,
        ex_result,
        hi_we,
        lo_we,
        hi_ex,
        lo_ex
    };
```

2.3.2 具体信息

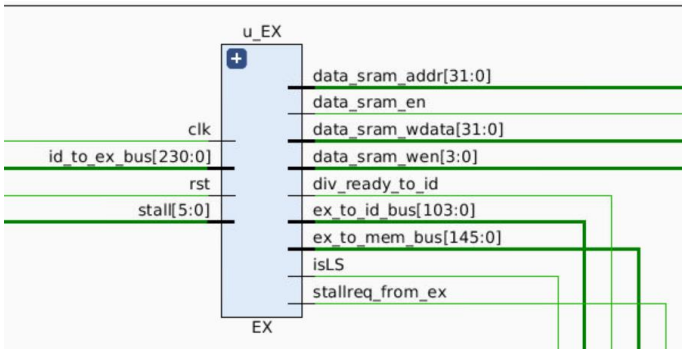

图 7

2.3.2.1 输出信号

alu_op 用于接收从 ID 段传输而来的操作信号。sel_alu_src1 和 sel_alu_src2 为接收的操作数选择信号。从 ID 段接收之后进行判断，alu_src1 操作数是 ex_pc 还是 sel_alu_src[1]，alu_src1[2]操作数是判断 sa 零扩展还是 rf_rdata1。sel_alu_src1[1]可以判断是否为立即数符号扩展，若是立即数符号扩展，则 alu_src2 变为该值；sel_alu_src1[2]判断是否为立即数符号扩展，sel_alu_src1[3]则判断是立即数零扩展还是寄存器数据 rf_rdata2。最终将操作信号和操作数传入 alu 处理。气泡信号为 isLS 信号，是根据接收的 inst[31:26]判断是否为 6’b100011，如果是，则说明需要添加气泡操作，否则不需要。然后将 isLS 信号传入 MEM 段。添加气泡的使能信号 stall 从 ID 段接收之后，在

always 执行周期时会进行判断，如果 stall[2]==`Stop && stall[3]==`NoStop 这个判断语句成立，则将 id_to_ex_bus_r 赋值为 0，否则就会继续执行操作。

乘除法器部分首先添加乘除法指令，定义 mul_result 和 mul_signed 变量来存储乘法运算的符号以及最终运算得到的值，if_mul 用来存储指令，如果 if_mul 为不为零，则将寄存器里取到的值 rf_rdata1 传入 rf_rdata_mul1，rf_rdata2 传入 rf_rdata_mul2，接下来进行乘法运算。除法同理。

HILO 的 EX 模块阶段，译码阶段的结果会传递到 EX 段并据此进行计算。考虑到 EX 段需要读写 HI、LO 寄存器，还要解决 HI、LO 寄存器带来的相关问题，定义相关变量来接收存储 HI、LO 寄存器的值以及是否要写入 HILO 寄存器，同时打包向后阶段发送的 HILO 信号。hi_read、lo_read 为读取的当前 hi、io 寄存器内的值。HI_write 和 LO_write 为向 HILO 寄存器输出的值。如果从 ID 段接收到需要向 HILO 寄存器读写的指令，则会自动判断，然后将 ex_result 的值写为 hi_read 或者 lo_read；如果没有接收到 HILO 寄存器的处理指令，则为 alu_result。

2.4 MEM 段

2.4.1 整体功能

MEM 段功能比较单一，主要是接收处理内存返回的查询结果。接口示意图如图 8 所示。

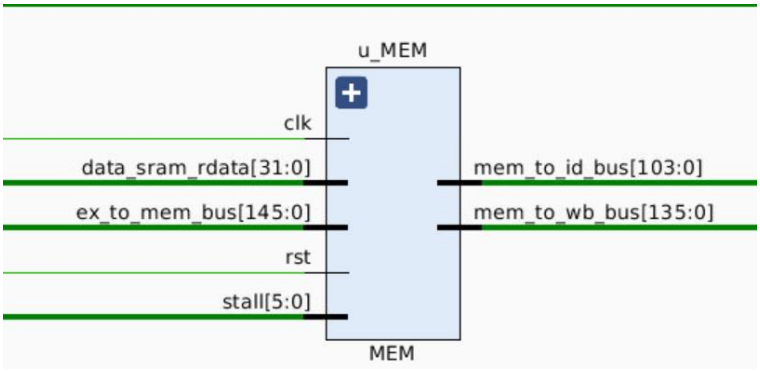

图 8

2.4.1.1 内存返回结果处理

内存返回查询结果 data_sram_rdata，不同的指令需要的查询结果内容不同，所以需要对 ex_to_mem_bus 总线里的 data_ram_readen、data_ram_en、ex_result 信号进行综合判断，截取对应指令想要的查询结果。ex_to_mem_bus_r 内容如图 9 所示。

```
assign {
    data_ram_readen,
    hi_we,
    lo_we,
    hi_ex,
    lo_ex,
    mem_pc,
    data_ram_en,
    data_ram_wen,
    sel_rf_res,
    rf_we,
    rf_waddr,
    ex_result
} =  ex_to_mem_bus_r;
```

图 9

以 LB 指令为例，LB 指令需要读取内存中一个字节的有符号扩展数据，其对应的信号 data_ram_readen 为 4 位的 0001，data_ram_en 为 1，ex_result 的最后两位为 00，当这些条件都满足时就将 data_sram_rdata 的后 8 位取出并做有符号扩展，将结果存储在 rf_wdata 中等待存回寄存器。

```
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;
```

图 10

2.4.1.2 数据相关

MEM 段产生最新计算结果 rf_wdata 后，添加 mem_to_id_bus 总线，原因是要尽快将数据传给后续正好要取该寄存器内容的指令使用。在总线中还要有寄存器地址和使能信号，这样可以知道该数据最后属于哪一个寄存器以及是否有效。mem_to_id_bus 内容如图 11 所示。

```
assign mem_to_id_bus={
    rf_we,
    rf_waddr,
    rf_wdata,
    hi_we,
    lo_we,
    hi_ex,
    lo_ex
};
```

图 11

2.5 WB 段

2.5.1 整体功能

WB 段的接口示意图如图 12 所示。

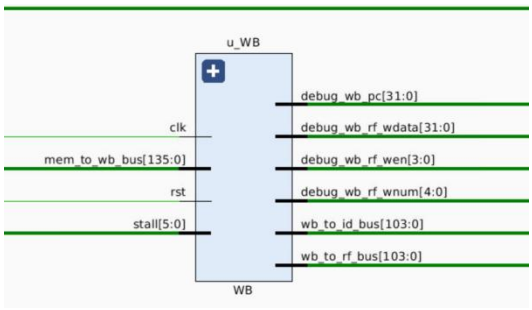

图 12

WB 段接收 MEM 段传输过来的 HILO 寄存器相关的存储指令，同时要判断是否进行 HILO 写入。通过 hi_we，lo_we，hi_ex，lo_ex，HILO 指令的相关信息可以从 EX 段一直传输到 WB 段，再传送到 HILO 模块，并且据此修改 HILO 寄存器的值。另一个传输操作为正常接收 MEM 段传输到 WB 段的信号并向寄存器和 ID 段传输。

```
always @ (posedge clk) begin
    if (rst) begin
        mem_to_wb_bus_r <= `MEM_TO_WB_WD'b0;
    end
    else if (stall[4]==`Stop && stall[5]==`NoStop) begin
        mem_to_wb_bus_r <= `MEM_TO_WB_WD'b0;
    end
    else if (stall[4]==`NoStop) begin
        mem_to_wb_bus_r <= mem_to_wb_bus;
    end
end
```

图 13

如果复位信号为 1 或者需要进行插入气泡，将 mem_to_wb_bus_r 赋值为 0。分析后可知不需要插入气泡，于是将 mem_to_wb_bus 的值给 mem_to_wb_bus_r。

2.5.2 具体信息

2.5.2.1 输入信号

将 HILO 相关操作信号传给 HILO 模块，相关寄存器操作转给 ID 段。最后输出 debug 的信号。

clk 表示时钟信号；rst 表示复位信号；stall 表示 Ctrl 控制的暂停信号；hi_we，lo_we 表示判断是否需要向 HILO 寄存器中写入的使能信号；hi_ex 和 lo_ex 表示访存阶段要写入 HILO 寄存器的值。mem_to_wb_bus 表示 MEM 段向 WB 段传输的信号，内部结构如图 14 所示。

```
assign mem_to_wb_bus = {
    hi_we,
    lo_we,
    hi_ex,
    lo_ex,
    mem_pc,
    rf_we,
    rf_waddr,
    rf_wdata
};
```

图 14

2.5.2.2 输出信号

wb_to_rf_bus 表示 WB 段向寄存器段传递的信号，内部结构如图 15 所示。hi，lo 都与 HILO 寄存器相关指令有关；rf_we 表示向寄存器传输信号的使能信号；rf_waddr 表示传输信号的地址；rf_wdata 表示所传输的数据。

wb_to_id_bus 表示 WB 段向 ID 段回写的信号，内部结构为如图 16 所示。rf_we 表示 WB 段向寄存器取的使能信号；rf_waddr 表示取的地址；rf_wdata 表示数据的存放。wb_to_id_bus 里面主要是通过 ID 段来传输，其余信号为 HILO 相关操作信号。

```
assign wb_to_rf_bus = {
    hi_we,
    lo_we,
    hi_ex,
    lo_ex,
    rf_we,
    rf_waddr,
    rf_wdata
};
```

图 15

```
assign wb_to_id_bus={
    rf_we,
    rf_waddr,
    rf_wdata,
    hi_we,
    lo_we,
    hi_ex,
    lo_ex
};
```

图 16

2.6 Ctrl 段

2.6.1 整体功能

Ctrl 主要负责控制流水线的暂停处理。它接收来自 ID 段和 EX 段的暂停请求，对于来自不同段的暂停请求给予不同的暂停信号 stall，并传递给流水线的各个部分，使其按照 stall 信号进行暂停操作。接口示意图如图 17 所示。

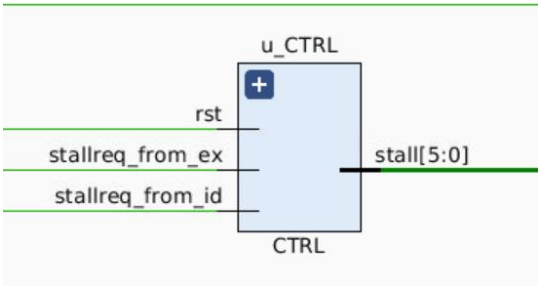

图 17

如图 17 所示，一开始暂停信号设定为 0，如果从 ID 段/EX 段接收到 stallreq_from_id/ex 信号为暂停信号，则给 stall 信号赋值为 6'b00_0111，以后三位为 1 来判断在其他段的暂停操作/给 stall 信号赋值为 6'b00_1111，以后四位为 1 来判断在其他段的暂停操作。

```
always @ (*) begin
    if (rst) begin
        stall = `StallBus'b0;
    end else if(stallreq_from_id==`Stop) begin
        stall= 6'b00_0111;
    end else if(stallreq_from_ex==`Stop) begin
        stall=6'b00_1111;
    end
    else begin
        stall = `StallBus'b0;
    end
end
```

图 18

2.6.2 具体信息

2.6.2.1 输入信号

rst 表示复位信号；stallreq_from_id 表示从 ID 段传输过来的暂停信号；stallreq_from_ex 表示从 EX 段传输过来的暂停信号。

2.6.2.2 输出信号

stall 暂停信号。

3. 实验感受和改进意见

3.1 袁云

通过本次实验，我们完成了流水线 CPU 的设计，并解决了一些数据相关、分支跳转等问题，对流水线 CPU 的基本结构有了更深刻的认识，对流水线 CPU 的工作原理和相关问题的解决方式有了更直观的了解。在这次实验中，一开始我的心里非常迷茫，感觉无从下手，但是老师一直督促着我们的进度，促进我们的学习，所以我们不断向老师与同学请教，阅读老师发的指导材料，在服务器上一遍又一遍尝试，实验渐渐有了些眉目，各个模块的功能开始清晰起来了，顺着控制台的提示和波形图等相关信息理清了思路，一步一步去做，最终成功完成了实验。在此次实验中我深刻感受到了团队合作的重要性，我和队友昕敏一直在努力 debug，有时候自己卡半天想不出来的问题与队友一起探讨后灵感很快就出现了。

总的来说，能完成实验我真的非常开心，因为投入了大量的时间与精力，完成那一刻，成就感一下子就上来了。感谢老师和学长给予我们的帮助，是您们的不厌其烦与悉心指导让我们能成功完成这次实验。也要感谢我的队友昕敏一直与我一起并肩作战，毫不退缩！希望老师下次在实验开始前能够有一个系统的指导，大致介绍一下实验的具体流程，比如说可以以某一指令的更改作为演示，让我们能够尽快找到下手的点。

3.2 姜昕敏

在本次实验中，我主要负责处理数据相关、流水线暂停处理、跳转指令和一些基本指令的添加，以及 1 至 36 条指令的任务。

这个学期我选了很多门课，在 CPU 实验的期间，我还有很多其他实验要去做，因此在前期没有花时间在实验上面。我刚开始看到需要 debug 的代码的时候，丝毫没有头绪，所以就先去做 ddl 比较紧张的实验了。老师开始统计进度的时候，发现我们组的进度比较慢，我和袁云就开始一起讨论去做。最开始看到出错误的指令也没有头绪如何去改，后来看到 GitHub 上的 debug 建议后知道，需要先添加一些数据相关的基本指令，通过仔细查看 IF、ID、EX、MEM、WB 的代码，对比其中相似的点和数据之后，学会了如何添加指令。其中 lw 和 sw 的指令是我们开始添加的有点困难的指令，研究明白这两个指令之后，我们就开始添加气泡和其他一些指令。过了一号点之后，我们就搁置了实验，去忙其他的事情了。后来到了 21 号，才开始接着 1 号点，继续往下做，直到 29 号凌晨，才把 64 个点都过完。我希望以后的实验可以给同学们有体系的讲述一下实验代码的大体结构，每一个部分都在做什么；给我们大概讲一下几种指令如何去加，让我们有一个参考的例子。对于我们从来没接触过这门语言以及这样的实验的同学，在开始的时候比较头痛。

4.参考资料

[1]雷思磊.自己动手做 cpu [M]. 北京:科学出版社

[2]AO3_“系统能力培养大赛”MIPS 指令系统规范_v1.01

[3]AO7_vivado 使用说明_v1.00

[4]AO9_CPU 仿真调试说明_v1.00

[5]A11_Trace 比对机制使用说明_v1.00

[6]CPU 设计实战 CPU Design and Practice (汪文祥, 邢金璋)