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 1901

组长：穆奕博

组员：高天，朱梓铭

报告日期：2021.12.18

目录

一、总体设计简介.....3
1.1 总体设计.....3
1.2 实验环境.....3
1.3 组内分工.....3
1.4 流水段简介.....3
二、单个流水段说明.....5
2.1 取值（IF）段.....5
2.2 译码（ID）段.....7
2.3 执行（EX）段.....10
2.4 访存（MEM）段.....14
2.5 回写（WB）段.....16
2.6 Regfile 寄存器的实现：.....18
三、流水线暂停机制的实现.....19
3.1 整体设计.....19
3.2 模块接口.....20
3.3 模块代码.....20
四、乘除法器的实现.....22
4.1 整体设计.....22
4.2 模块接口.....22
4.3 模块代码.....23
五、转移与分支指令实现.....28
5.1 整体设计.....28
5.2 代码实现.....29
六、加载与存储指令实现.....31
6.1 总体设计.....31
6.2 详细设计与相关代码.....31
七、附加模块说明.....33
八、组员实验感受.....35
九、参考资料.....36

一、总体设计简介

1.1 总体设计

本实验共计完成算术与逻辑运算、分支跳转、移位、数据移动、访存等共计51条指令，可以完成上述指令的功能，并通过64个点的仿真软件与实验板测试。实验仓库的地址为 https://github.com/Sswjm/NEU-CPUdesign。

1.2 实验环境

本实验仿真环境为vivado 2019.2，使用vscode编写verilog代码

1.3 组内分工

穆奕博：完成数据相关通路连接，完成subu、addu、jal、jr、sll、or等算术和跳转指令，完成hilo寄存器的添加，完成乘法和除法指令，完成访存指令并实现流水线暂停。

高天：完成xor、sltu、bne、slt、slti、sltiu、j、add、addi、sub、and、andi、nor、xori指令的添加

朱梓铭：完成sllv、sra、srav、srl、srlv、bgez、bgtz、blez、bltz、bltzal、bgezal、jalr指令的添加

1.4 流水段简介

流水线是指将计算机指令处理过程拆分为多个步骤，并通过多个硬件处理单元并行执行来加快指令执行速度。

流水线广泛应用于高档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，对应周期的五个处理阶段。如图1所示，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。

	1	2	3	4	5	6	7	8	9
I1	IF	ID	EX	MEM	WB				
I2		IF	ID	EX	MEM	WB			
I3			IF	ID	EX	MEM	WB		
I4				IF	ID	EX	MEM	WB	
I5					IF	ID	EX	MEM	WB

图1 流水线流水作业示意图

本实验流水线为五级流水段，如图 2 所示，分别为 IF（取指）、ID（译码）、EX（执行）、MEM（访存）、WB（回写）。

取指阶段：从指令存储器读出指令，同时确定下一条指令地址。

译码阶段：对指令进行译码，从通用寄存器中读出要使用的寄存器的值，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。

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，那么还会计算 load/store 的目标地址。

访存阶段：如果是 load/store 指令，那么在此阶段回访问数据存储器，反之，只是将执行阶段的结果向下传递到回写阶段。同时，在此阶段还要判断是否有异常需要处理，如果有，那么回清除流水线，然后转移到异常处理例程入口地址处继续执行。

回写阶段：将运算结果保存到目标寄存器。

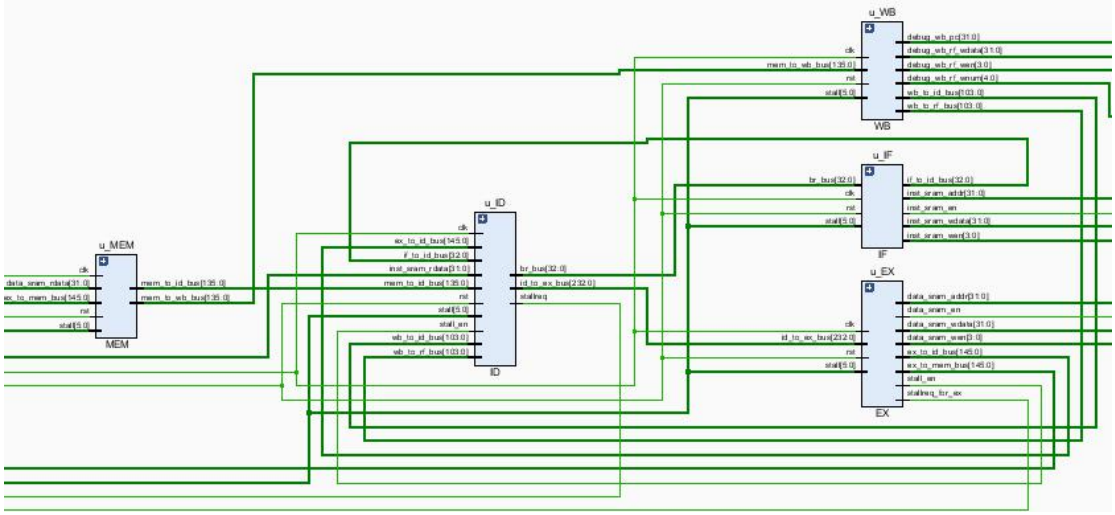


图 2 五级流水段总体接线示意图

流水线中经常有一些被称为“相关”的情况发生，它使得指令序列中下一条指令无法按照设计的时钟周期执行，这些“相关”会降低流水线的性能。流水线中的相关分为以下三种类型。

（1）结构相关：指的是在指令执行的过程中，由于硬件资源满足不了指令执行的要求，发生硬件资源冲突而产生的相关。例如指令和数据都共享一个存储器，在某个时钟周期，流水线既要完成某条指令对存储器中数据的访问操作，又要完成后续的取指令操作，这样就会发生存储器访问冲突，产生结构相关。

（2）数据相关：指的是在流水线中执行的几条指令中，一条指令依赖于前面指令的执行结果。

（3）控制相关：指的是流水线中的分支指令或者其他需要改写 PC 的指令造成的相关。

流水线数据相关又分为三种情况：RAW，WAR、WAW

RAW：假设指令 j 是在指令 i 后面执行的指令，RAW 表示指令 i 将数据写入寄存器后，指令 j 才能从这个寄存器读取数据。如果指令 j 在指令 i 写入寄存器前尝试读出该寄存器的内容，将得到不正确的数据。

WAR：假设指令 j 是在指令 i 后面执行的指令，WAR 表示指令 i 读出数据后，指令 j 才能写这个寄存器。如果指令 j 在指令 i 读出数据前就写入该寄存器，将

使得指令 i 读出的数据不正确。

WAW：假设指令 j 是在指令 i 后面执行的指令，WAW 表示指令 i 将数据写入寄存器后，指令 j 才能将数据写入这个寄存器。如果指令 j 在指令 i 之前写入该寄存器，将使得该寄存器的值不是最新值。

二、单个流水段说明

2.1 取值（IF）段

2.1.1 整体功能说明

取指阶段取出指令存储器中的指令，同时，PC 值递增，准备取下一条指令。

2.1.2 端口介绍

（1）PC 模块接口描述：

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	stall	6	输入	暂停信号
4	br_bus	33	输入	跳转信号总线
5	if_to_id_bus	33	输出	IF 段连接 ID 段总线
6	inst_sram_en	1	输出	存储使能信号
7	inst_sram_wen	4	输出	存储写使能信号
8	inst_sram_addr	32	输出	存储写入地址
9	inst_sram_wdata	32	输出	存储写入数据

2.1.3 包含功能模块介绍

（1）暂停部分

控制 IF 段流水线暂停。

（2）PC 模块

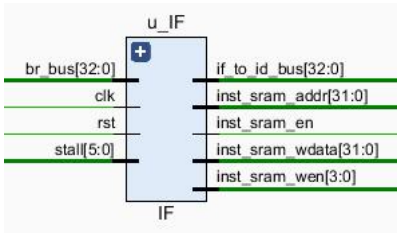
控制 PC 的值，包括增加与跳转。

2.1.4 对应代码与结构示意图

```
3       reg [31:0] pc_reg;
4       reg ce_reg;
5       wire [31:0] next_pc;
6       wire br_e;
```

```
7       wire [31:0] br_addr;
8
9       assign {
10          br_e,
11          br_addr
12      } = br_bus;
13
14      always @ (posedge clk) begin
15          if (rst) begin
16              pc_reg <= 32'hbfbf_fffc;
17          end
18          else if (stall[0]==`NoStop) begin
19              pc_reg <= next_pc;
20          end
21      end
22
23      always @ (posedge clk) begin
24          if (rst) begin
25              ce_reg <= 1'b0;
26          end
27          else if (stall[0]==`NoStop) begin
28              ce_reg <= 1'b1;
29          end
30      end
31
32      assign next_pc = br_e ? br_addr
33                            : pc_reg + 32'h4;
34
35
36      assign inst_sram_en = ce_reg;
37      assign inst_sram_wen = 4'b0;
38      assign inst_sram_addr = pc_reg;
39      assign inst_sram_wdata = 32'b0;
40      assign if_to_id_bus = {
41          ce_reg,
42          pc_reg
43      };
```

结构示意图：



2.2 译码（ID）段

2.2.1 整体功能说明

在此阶段，将对取到的指令进行译码：给出要进行的运算类型，以及参与运算的操作数。

2.2.2 端口介绍

ID 模块接口描述：

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	stall	6	输入	暂停信号
4	if_to_id_bus	33	输入	IF 连接 ID 段总线
5	inst_sram_rdata	32	输入	存储读出数据
6	wb_to_rf_bus	104	输入	WB 连接 ID 总线
7	ex_to_id_bus	146	输入	EX 连接 ID 总线
8	mem_to_id_bus	136	输入	MEM 连接 ID 总线
9	wb_to_id_bus	104	输入	WB 连接 ID 总线
10	stall_en	1	输入	EX 段到 ID 段暂停请求
11	id_to_ex_bus	233	输出	ID 连接 EX 总线
12	br_bus	33	输出	跳转信号总线
13	stallreq	1	输出	ID 段暂停信号

2.2.3 包含功能模块介绍

（1）Regfile 实例化

该模块实现了 32 个 32 位通用整数寄存器，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）译码模块

该模块的作用是对指令进行译码，得到最终运算的类型、子类型、源操作数 1、源操作数 2、要写入的目的寄存器地址等信息，其中运算类型指的是逻辑运算、移位运算、算术运算等，子类型指的是更加详细的运算类型。

（3）数据相关模块

通过 forwarding 技术，解决流水线的数据相关问题。

2.2.4 对应代码与结构示意图

Regfile 实例化

```
regfile u_regfile(
    .clk    (clk    ),
    .raddr1 (rs ),
    .rdata1 (rdata1 ),
    .raddr2 (rt ),
    .rdata2 (rdata2 ),
    .we     (wb_rf_we     ),
    .waddr  (wb_rf_waddr  ),
    .wdata  (wb_rf_wdata  ),
    .hi_we  (wb_hi_we     ),
    .hi_wdata(wb_hi_wdata ),
    .lo_we  (wb_lo_we     ),
    .lo_wdata(wb_lo_wdata ),
    .hi_rdata(hi_rdata    ),
    .lo_rdata(lo_rdata    )
);
```

译码

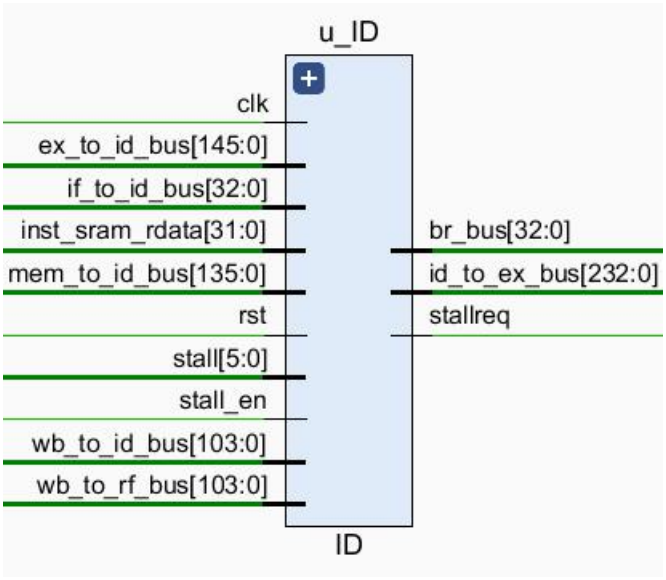
```
decoder_6_64 u0_decoder_6_64(
    .in  (opcode  ),
    .out (op_d )
);

decoder_6_64 u1_decoder_6_64(
    .in  (func  ),
    .out (func_d )
);

decoder_5_32 u0_decoder_5_32(
    .in  (rs  ),
    .out (rs_d )
);

decoder_5_32 u1_decoder_5_32(
    .in  (rt  ),
    .out (rt_d )
);
```

```
    assign inst_ori     = op_d[6'b00_1101];
    assign inst_lui     = op_d[6'b00_1111];
    assign inst_addiu   = op_d[6'b00_1001];
    assign inst_beq     = op_d[6'b00_0100];
    assign inst_subu    = op_d[6'b00_0000] & func_d[6'b10_0011];
    assign inst_addu    = op_d[6'b00_0000] & func_d[6'b10_0001];
    assign inst_jal     = op_d[6'b00_0011];
```

数据相关:

```
    assign selected_rdata1 = (forwarding_ex_rf_we &
(forwarding_ex_rf_waddr == rs)) ? forwarding_ex_result
                           : (forwarding_mem_rf_we &
(forwarding_mem_rf_waddr == rs)) ? forwarding_mem_rf_wdata
                           : (wb_rf_we & (wb_rf_waddr == rs)) ? wb_rf_wdata
                           : rdata1;

    assign selected_rdata2 = (forwarding_ex_rf_we &
(forwarding_ex_rf_waddr == rt)) ? forwarding_ex_result
                           : (forwarding_mem_rf_we &
(forwarding_mem_rf_waddr == rt)) ? forwarding_mem_rf_wdata
                           : (wb_rf_we & (wb_rf_waddr == rt)) ? wb_rf_wdata
                           : rdata2;

    assign selected_hi_rdata = (forwarding_ex_hi_we) ?
forwarding_ex_hi_wdata
                           : (forwarding_mem_hi_we) ?
forwarding_mem_hi_wdata
                           : (wb_hi_we) ? wb_hi_wdata
                           : hi_rdata;

    assign selected_lo_rdata = (forwarding_ex_lo_we) ?
forwarding_ex_lo_wdata
                           : (forwarding_mem_lo_we) ?
forwarding_mem_lo_wdata
                           : (wb_lo_we) ? wb_lo_wdata
                           : lo_rdata;
```

结构示意图:



2.3 执行（EX）段

2.3.1整体功能说明

此阶段将依据译码阶段的结果，对源操作数 1、源操作数 2，进行指定的运算，并判断访存的地址和数据。

2.3.2 端口介绍

（1）EX 段

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	stall	6	输入	暂停信号
4	id_to_ex_bus	233	输入	ID 段连接 EX 段总线
5	ex_to_mem_bus	146	输出	EX 段连接 MEM 段总线
6	data_sram_en	1	输出	访存使能信号
7	data_sram_wen	4	输出	访存写入使能信号
8	data_sram_addr	32	输出	访存地址
9	data_sram_wdata	32	输出	访存写入数据
10	ex_to_id_bus	146	输出	EX 段连接 ID 段总线
11	stallreq_for_ex	1	输出	EX 段暂停信号
12	stall_en	1	输出	EX 段向 ID 段暂停信号

2.3.3包含功能模块介绍

（1） EX 模块

EX 从 ID 得到运算类型、运算子类型、源操作数 src1、源操作数 src2、要写入的目的寄存器地址 waddr。EX 模块依据这些数据进行计算。

（2） ALU 模块

执行算术运算。

（3） MUL 模块

执行乘法运算及暂停。

（4） DIV 模块

执行乘法运算及暂停。

（5） HILO 模块

处理 hilo 寄存器的读写数据。

（6） 访存模块

处理访存相关指令。

2.3.4 对应代码与结构示意图

ALU 及 HILO：

```
    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;

    alu u_alu(
        .alu_control (alu_op      ),
        .alu_src1    (alu_src1    ),
        .alu_src2    (alu_src2    ),
        .alu_result  (alu_result  )
    );

    assign ex_result = sel_move_dst[2] ? hi_rdata    //hi --> rd
                     : sel_move_dst[3]  ? lo_rdata    //lo --> rd
                     : alu_result;

    //hilo part start
    assign hi_wdata = sel_move_dst[0] ? rf_rdata1  //rs --> hi
                    : (div_mul_select[0] | div_mul_select[1]) ?
div_result[63:32]    //div
                    : (div_mul_select[2] | div_mul_select[3]) ?
mul_result[63:32]    //mul
                    : hi_rdata;
    assign lo_wdata = sel_move_dst[1] ? rf_rdata1  //rs --> lo
                    : (div_mul_select[0] | div_mul_select[1]) ?
div_result[31:0]    //div
```

```
                    : (div_mul_select[2] | div_mul_select[3]) ?
mul_result[31:0]
                    : lo_rdata;
```

MUL 模块：

```
    wire [63:0] mul_result;
    wire inst_mult, inst_multu;
    wire mul_ready_i;
    reg stallreq_for_mul;

    assign inst_mult = div_mul_select[2];
    assign inst_multu = div_mul_select[3];


    reg [31:0] mul_opdata1_o;
    reg [31:0] mul_opdata2_o;
    reg mul_start_o;
    reg signed_mul_o;

    mymul 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      )
    );
```

DIV 模块：

```
    wire [63:0] div_result;
    wire inst_div, inst_divu;
    wire div_ready_i;
    reg stallreq_for_div;

    assign inst_div = div_mul_select[0];
    assign inst_divu = div_mul_select[1];

    assign stallreq_for_ex = stallreq_for_div | stallreq_for_mul;

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

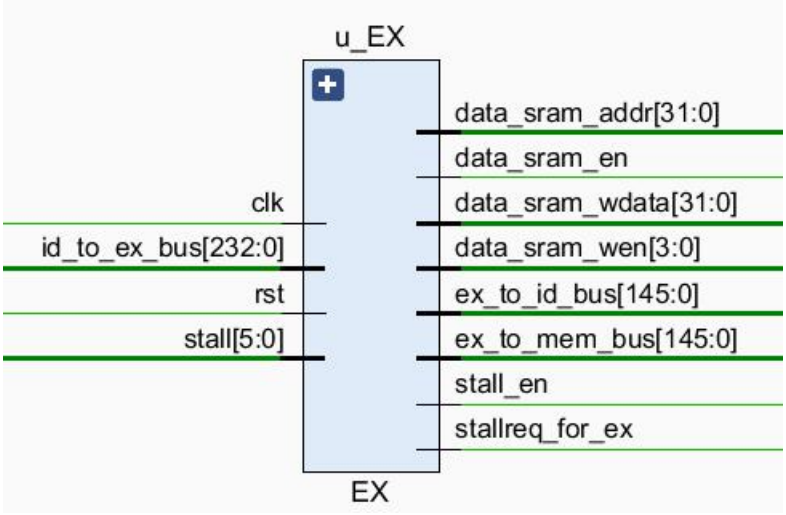
```
    div u_div(
        .rst          (rst          ),
        .clk          (clk          ),
        .signed_div_i (signed_div_o ),
        .opdata1_i    (div_opdata1_o    ),
        .opdata2_i    (div_opdata2_o    ),
        .start_i      (div_start_o      ),
        .annul_i      (1'b0      ),
        .result_o     (div_result     ), // 除法结果 64bit
        .ready_o      (div_ready_i      )
    );
```

访存模块：

```
    assign stall_en = data_ram_en
                    & (data_ram_wen == 4'b0 | data_ram_wen == 4'b0001 |
data_ram_wen == 4'b0010
                        | data_ram_wen == 4'b0100 | data_ram_wen ==
4'b0110);
                    //tell "id" to stall if this instruction is a load
instruction

    assign data_sram_en = data_ram_en;

    assign data_sram_wen = (data_ram_wen == 4'b1101 & alu_result[1:0] ==
2'b00) ? 4'b0011  //sh
                    : (data_ram_wen == 4'b1101 & alu_result[1:0] == 2'b10) ?
4'b1100       //sh
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b00) ?
4'b0001       //sb
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b01) ?
4'b0010       //sb
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b10) ?
4'b0100       //sb
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b11) ?
4'b1000       //sb
                    : (data_ram_wen == 4'b1111) ?
4'b1111                                 //sw
                    :
4'b0;                                                                   //loa
ds


    assign data_sram_addr = alu_result;
    //assign data_sram_wdata = (data_ram_wen == 4'b1111) ? rf_rdata2 : 32'b0;
    assign data_sram_wdata = (data_ram_wen == 4'b1101 & alu_result[1:0] ==
2'b00) ? {16'b0, rf_rdata2[15:0]}
```

```
                    : (data_ram_wen == 4'b1101 & alu_result[1:0] == 2'b10) ?
{rf_rdata2[15:0], 16'b0}
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b00) ?
{24'b0, rf_rdata2[7:0]}
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b01) ?
{16'b0,rf_rdata2[7:0], 8'b0}
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b10) ?
{8'b0, rf_rdata2[7:0], 16'b0}
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b11) ?
{rf_rdata2[7:0], 24'b0}
                    : (data_ram_wen == 4'b1111) ? rf_rdata2
                    : 32'b0;
```

结构示意图：



2.4 访存（MEM）段

2.4.1整体功能说明

根据访存指令相关信息，进行访存操作。

2.4.2 端口介绍

（1）MEM 段

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	stall	6	输入	暂停信号
4	ex_to_mem_bus	146	输入	EX 段连接 MEM 段总线
5	data_sram_rdata	32	输入	访存读出数据

6	mem_to_wb_bus	136	输出	MEM 段连接 WB 段总线
7	mem_to_id_bus	136	输出	MEM 段连接 ID 段总线

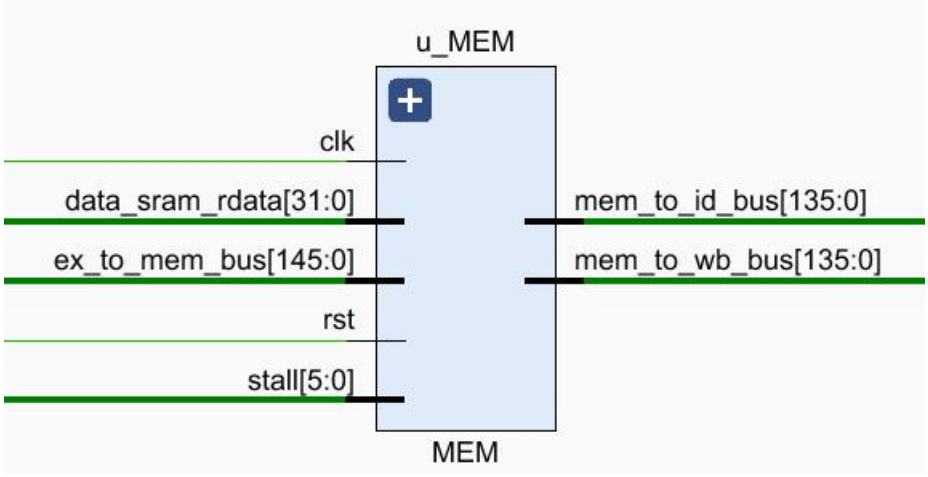
2.4.3 包含功能模块介绍

(1) 处理 load 指令得到的内存数据，将其传给 WB 段进行回写。

2.4.4 相关代码及结构示意图

```
    assign mem_result = (load_judge == 4'b0110 & ex_result[1:0] == 2'b00) ?
{{24{data_sram_rdata[7]}}, data_sram_rdata[7:0]}  //lb
                    : (load_judge == 4'b0110 & ex_result[1:0] == 2'b01) ?
{{24{data_sram_rdata[15]}}, data_sram_rdata[15:8]}  //lb
                        : (load_judge == 4'b0110 & ex_result[1:0] ==
    2'b10) ? {{24{data_sram_rdata[23]}}, data_sram_rdata[23:16]} //lb
                    : (load_judge == 4'b0110 & ex_result[1:0] == 2'b11) ?
{{24{data_sram_rdata[31]}}, data_sram_rdata[31:24]} //lb
                    : (load_judge == 4'b0100 & ex_result[1:0] == 2'b00) ?
{24'b0, data_sram_rdata[7:0]}                    //lbu
                    : (load_judge == 4'b0100 & ex_result[1:0] == 2'b01) ?
{24'b0, data_sram_rdata[15:8]}                   //lbu
                    : (load_judge == 4'b0100 & ex_result[1:0] == 2'b10) ?
{24'b0, data_sram_rdata[23:16]}                  //lbu
                    : (load_judge == 4'b0100 & ex_result[1:0] == 2'b11) ?
{24'b0, data_sram_rdata[31:24]}                  //lbu
                    : (load_judge == 4'b0010 & ex_result[1:0] == 2'b00) ?
{{16{data_sram_rdata[15]}}, data_sram_rdata[15:0]}  //lh
                    : (load_judge == 4'b0010 & ex_result[1:0] == 2'b10) ?
{{16{data_sram_rdata[31]}}, data_sram_rdata[31:16]} //lh
                    : (load_judge == 4'b0001 & ex_result[1:0] == 2'b00) ?
{16'b0, data_sram_rdata[15:0]}                   //lhu
                    : (load_judge == 4'b0001 & ex_result[1:0] == 2'b10) ?
{16'b0, data_sram_rdata[31:16]}                  //lhu
                    :
data_sram_rdata;
                        //lw

  assign rf_wdata = (sel_rf_res) ? mem_result : ex_result;
```

结构示意图：



2.5 回写（WB）段

2.5.1整体功能说明

模块中实现的MEM模块的输出 rf_we、rf_waddr、rf_wdata连接到Regfile模块,分别连接到写使能端口 we、写操作目的寄存器端口 waddr、写入数据端口wdata,所以会将指令的运算结果写入目的寄存器。

2.5.2 端口介绍

OpenMIPS 模块

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	stall	6	输入	暂停信号
4	mem_to_wb_bus	136	输入	MEM 段连接 WB 段总线
5	wb_to_rf_bus	104	输出	WB 段连接 regfile 总线
6	wb_to_id_bus	104	输出	WB 段连接 ID 段总线
7	debug_wb_pc	32	输出	debug 信号
8	debug_wb_rf_wen	4	输出	debug 信号
9	debug_wb_rf_wnum	5	输出	debug 信号
10	degub_wb_rf_wdata	32	输出	debug 信号

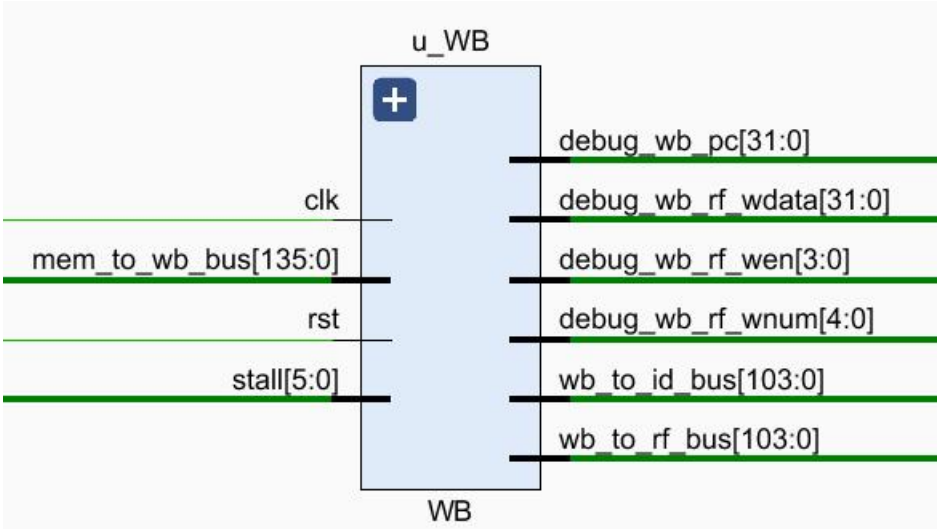
2.5.3 包含功能模块介绍

（1）处理回写数据。

2.5.4 相关代码及结构示意图

```
wire [31:0] wb_pc;
wire rf_we;
    wire [4:0] rf_waddr;
    wire [31:0] rf_wdata;
    wire hi_we, lo_we;
    wire [31:0] hi_wdata, lo_wdata;

    assign {
        wb_pc,
        rf_we,
        rf_waddr,
        rf_wdata,
        hi_we,
        lo_we,
        hi_wdata,
        lo_wdata
    } = mem_to_wb_bus_r;

    // assign wb_to_rf_bus = mem_to_wb_bus_r[`WB_TO_RF_WD-1:0];
    assign wb_to_rf_bus = {
        rf_we,
        rf_waddr,
        rf_wdata,
        hi_we,
        lo_we,
        hi_wdata,
        lo_wdata
    };

    assign wb_to_id_bus = {
        rf_we,
        rf_waddr,
        rf_wdata,
        hi_we,
        lo_we,
        hi_wdata,
        lo_wdata
    };

    assign debug_wb_pc = wb_pc;
    assign debug_wb_rf_wen = {4{rf_we}};
    assign debug_wb_rf_wnum = rf_waddr;
    assign debug_wb_rf_wdata = rf_wdata;
```

结构示意图：



2.6 Regfile 寄存器的实现：

2.6.1 整体功能说明

Regfile 模块实现了普通寄存器阵列（32 个）和 hilo 寄存器，可以完成其读写操作。

2.6.2 端口介绍

在 hilo 模块中实现 hi、lo 寄存器，hilo 模块的接口描述如下所示：

序号	接口名	宽度	输入/输出	作用
1	clk	1	输入	时钟信号
2	raddr1	5	输入	数据 1 地址
3	rdata1	32	输出	读取的数据 1
4	raddr2	5	输入	数据 2 地址
5	rdata2	32	输出	读取的数据 2
6	we	1	输入	寄存器写入使能信号
7	waddr	5	输入	寄存器写入地址
8	wdata	32	输入	寄存器写入数据
9	hi_we	1	输入	hi 寄存器写入使能信号
10	hi_wdata	32	输入	hi 寄存器写入数据
11	lo_we	1	输入	lo 寄存器写入使能信号
12	lo_wdata	32	输入	lo 寄存器写入数据
13	hi_rdata	32	输出	hi 寄存器读出数据

2.6.3 相关代码

```
reg [31:0] reg_array [31:0];

reg [31:0] reg_hi;
reg [31:0] reg_lo;
// write
always @ (posedge clk) begin
    if (we && waddr!=5'b0) begin
        reg_array[waddr] <= wdata;
    end
    if (hi_we) begin
        reg_hi <= hi_wdata;
    end
    if (lo_we) begin
        reg_lo <= lo_wdata;
    end
end

// read out 1
assign rdata1 = (raddr1 == 5'b0) ? 32'b0 : reg_array[raddr1];

// read out2
assign rdata2 = (raddr2 == 5'b0) ? 32'b0 : reg_array[raddr2];

//read hi
assign hi_rdata = reg_hi;
assign lo_rdata = reg_lo;
```

三、流水线暂停机制的实现

3.1 整体设计

因为设计了乘法除法，乘法除法指令在流水线执行阶段占用多个时钟周期，因此需要暂停流水线，以等待这些多周期指令执行完毕，一种直观的实现方法是：要暂停流水线，只需要保持取指令地址 PC 的值不变，同时保持流水线各个阶段的寄存器，也就是 IF/ID、ID/EX、EX/MEM、MEM/WB 模块的输出不变。

OpenMips 采用的是一种改进的方法，加入位于流水线第 n 阶段的指令需要多个时钟周期，进而请求流水线暂停，那么需要保持取指令地址 PC 的值不变，同

时保持流水线第 n 阶段、第 n 阶段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。

为此设计添加了 CTRL 模块，其作用是接收个阶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。

CTRL 模块的输入来自 ID、EX 模块的请求暂停信号 stallreq，其中只有译码、执行阶段可能会有暂停请求，取指、访存阶段都没有暂停请求，因为指令读取、数据存储器的读、写操作都可以在一个时钟周期内完成。

CTRL 模块对暂停请求信号进行判断，然后输出流水线暂停信号 stall。Stall 输出到 PC、IF/ID、ID/EX、EX/MEM、MEM/WB 等模块，从而控制 PC 的值，以及流水线各个阶段的寄存器。

需要暂停的指令有，load 类指令和乘除法指令。流水线中，通常使用 always 块判断是否需要暂停。

3.2 模块接口

CTRL 模块的接口如下所示。

序号	接口名	宽度	输入/输出	作用
1	rst	1	输入	复位信号
2	stallreq	1	输入	处于译码阶段的指令是否请求流水线暂停
3	stallreq_from_ex	1	输入	处于执行阶段的指令是否请求流水线暂停
4	stall	6	输出	暂停流水线控制信号

3.3 模块代码

```
module CTRL(
    input wire rst,
    input wire stallreq,          //from id
    input wire stallreq_for_ex,   //from ex
    //input wire stallreq_for_load,

    // output reg flush,
    // output reg [31:0] new_pc,
    output reg [`StallBus-1:0] stall
);
    always @ (*) begin
        if (rst) begin
            stall = `StallBus'b0;
        end
```

```
        else if(stallreq == `Stop) begin  //from id
            stall = `StallBus'b00_0111;
        end
        else if(stallreq_for_ex == `Stop) begin  //from ex
            stall = `StallBus'b00_1111;
        end
        else begin
            stall = `StallBus'b0;
        end
    end

endmodule
```

暂停代码示例（ID）：

```
    always @ (posedge clk) begin
        if (rst) begin
            if_to_id_bus_r <= `IF_TO_ID_WD'b0;
            if_id_stop <= 1'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;
            //stall_inst <= inst_sram_rdata;
            if_id_stop <= 1'b0;
        end
        else if (stall[1]==`NoStop) begin
            if_to_id_bus_r <= if_to_id_bus;
            if_id_stop <= 1'b0;
        end
        else if(stall[2] == `Stop) begin
            if_id_stop <= 1'b1;
        end
    end

    //assign inst = if_id_stop ? 32'b0 : inst_sram_rdata;
    assign inst = if_id_stop ? inst : inst_sram_rdata;
```

load 暂停信号的发出（详细设计会在访存部分中说明）：

```
assign stallreq = ((stall_en) & ((rs == forwarding_ex_rf_waddr) | (rt == forwarding_ex_rf_waddr))) ? `Stop : `NoStop;
```

四、乘除法器的实现

4.1 整体设计

乘法器采用移位法实验，对于 32 位的乘法，至少需要 32 个时钟周期才能得到乘法结果。最终得到的乘法结果为 64 位，计算过程如下：

1.给定乘数 m，被乘数 n，部分积 s

2. 每个计算周期内，被乘数左移一位，乘数右移一位。判断 m 的最低位是 0 还是 1，如果是 1，将 m 加到部分积 s 上；否则，将 0 加到部分积上

3. 不断重复，直到算出结果。

除法器采用试商法实现除法运算，对于 32 位的除法，至少需要 32 个时钟周期才能得到除法结果。

设被除数是 m，除数是 n，商保存在 s 中，被除数的位数是 k，其计算步骤如下：

1. 取出被除数的最高位 m[k]，使用被除数的最高位减去除数 n，如果结果大于等于 0，则商的 s[k]为 1，反之为 0.

2. 如果上一步得出的结果是 0，表示当前的被减数小于除数，则取出被除数剩下的值的最高位 m[k-1]，与当前被减数组合作为下一轮的被减数；如果上一步得出的结果是 1，表示当前的被减数大于除数，则利用上一步中减法的结果与被除数剩下的值的最高位 m[k-1]组合作为下一轮的被减数。然后，设置 k 等于 k-1。

3. 新的被减数减去除数，如果结果大于等于 0，则商的 s[k]为 1，否则 s[k]为 0，后面的步骤重复 2-3，直到 k 等于 1。

计算步骤可如下图所示：

新建一个模块 DIV 和 MUL，在其中实现采用试商法的 32 位除法和乘法运算。当流水线执行阶段的 EX 模块发现当前指令是除法或乘法指令时，首先暂停流水线，然后将被除数（被乘数）、除数（乘数）等信息送到 DIV（MUL）模块，开始除法（乘法）运算。DIV（MUL）模块在除法（乘法）运算结束后，通知 EX 模块，并将除法（乘法）结果送到 EX 模块，后者依据除法结果设置 hi、lo 寄存器的写信息，同时取消暂停流水线。

4.2 模块接口

（1）MUL 模块的接口：

序号	接口名	宽度	输入/输出	作用
1	rst	1	输入	复位信号，高电平有效
2	clk	1	输入	时钟信号
3	signed_mul_i	1	输入	是否有符号乘法，为 1 表示有符号乘法
4	opdata1_i	32	输入	被乘数

5	opdata2_i	32	输入	乘数
6	start_i	1	输入	是否开始乘法运算
7	annul_i	1	输入	是否取消乘法运算，为 1 表示取消乘法运算
8	result_o	64	输出	乘法运算结果
9	ready_o	1	输出	乘法运算是否结束

（2）DIV 模块的接口：

序号	接口名	宽度	输入/输出	作用
1	rst	1	输入	复位信号，高电平有效
2	clk	1	输入	时钟信号
3	signed_div_i	1	输入	是否有符号除法，为 1 表示有符号除法
4	opdata1_i	32	输入	被除数
5	opdata2_i	32	输入	除数
6	start_i	1	输入	是否开始除法运算
7	annul_i	1	输入	是否取消除法运算，为 1 表示取消除法运算
8	result_o	64	输出	除法运算结果
9	ready_o	1	输出	除法运算是否结束

4.3 模块代码

乘法器：

```
module mymul(
    input wire rst,
    input wire clk,
    input wire signed_mul_i,
    input wire[31:0] opdata1_i,              //被乘数
    input wire[31:0] opdata2_i,              //乘数
    input wire start_i,                      //是否开始乘法运算
    input wire annul_i,                      //是否取消乘法运算
    output reg[63:0] result_o,               //乘法运算结果
    output reg ready_o                       //乘法运算是否结束
);
    // four states, mul_free, mul_on, mul_end, mul_by_zero
    reg [31:0] temp_op1;
    reg [31:0] temp_op2;
    reg [63:0] multiplicand;   //被乘数 每次运算左移 1 位
    reg [31:0] multiplier;     //乘数    每次运算右移 1 位
    reg [63:0] product_temp;   //临时结果
```

```
    reg [5:0] cnt;      //if 32, mul stop
    reg [1:0] state;    //共有四个状态

    wire [63:0] partial_product; //部分积

    assign partial_product = multiplier[0] ? multiplicand : {`ZeroWord,
`ZeroWord};

    always @ (posedge clk) begin
        if (rst) begin
            state <= `MulFree;
            ready_o <= `MulResultNotReady;
            result_o <= {`ZeroWord, `ZeroWord};
        end else begin
            case (state)
                `MulFree: begin
                    if (start_i == `MulStart && annul_i == 1'b0) begin
                        if (opdata1_i == `ZeroWord || opdata2_i == `ZeroWord)
begin   //任何操作数为 0，都进入 MUL_BY_ZERO 状态
                            state <= `MulByZero;
                        end else begin
                            state <= `MulOn;
                            cnt <= 6'b000000;
                            if (signed_mul_i == 1'b1 && opdata1_i[31] == 1'b1)
begin    //op1 为负数，取补码
                                temp_op1 = ~opdata1_i + 1;
                            end else begin
                                temp_op1 = opdata1_i;
                            end
                            if (signed_mul_i == 1'b1 && opdata2_i[31] == 1'b1)
begin    //op2 为负数，取补码
                                temp_op2 = ~opdata2_i + 1;
                            end else begin
                                temp_op2 = opdata2_i;
                            end
                            multiplicand <= {32'b0, temp_op1};
                            multiplier <= temp_op2;
                            product_temp <= {`ZeroWord, `ZeroWord};
                        end
                    end else begin
                        ready_o <= `MulResultNotReady;
                        result_o <= {`ZeroWord, `ZeroWord};
                    end
                end
```

```
                `MulByZero: begin
                    product_temp <= {`ZeroWord, `ZeroWord};    //有一个运
算数为0，结果为0
                    state <= `MulEnd;
                end

                `MulOn: begin
                    if (annul_i == 1'b0) begin
                        if (cnt != 6'b100000) begin
                            multiplicand <= {multiplicand[62:0],
1'b0};        //被乘数左移
                            multiplier <= {1'b0,
multiplier[31:1]};           //乘数右移
                            product_temp <= product_temp +
partial_product;    //相加
                            cnt <= cnt + 1;
                        end else begin   //运算结束，如果原来操作数为一正一负，
取补码
                            if ((signed_mul_i == 1'b1) && ((opdata1_i[31] ^
opdata2_i[31]) == 1'b1)) begin
                                product_temp <= ~product_temp + 1;
                            end
                            state <= `MulEnd;
                            cnt <= 6'b000000;
                        end
                    end else begin
                        state <= `MulFree;
                    end
                end

                `MulEnd: begin
                    result_o <= product_temp;
                    ready_o <= `MulResultReady;
                    if (start_i == `MulStop) begin
                        state <= `MulFree;
                        ready_o <= `MulResultNotReady;
                        result_o <= {`ZeroWord, `ZeroWord};
                    end
                end


            endcase
        end
    end
```

```
endmodule
```

除法器：

```
module div(
    input wire rst,                         //复位
    input wire clk,                         //时钟
    input wire signed_div_i,                        //是否为有符号除法运算，1 位有符号
    input wire[31:0] opdata1_i,             //被除数
    input wire[31:0] opdata2_i,             //除数
    input wire start_i,                     //是否开始除法运算
    input wire annul_i,                     //是否取消除法运算，1 位取消
    output reg[63:0] result_o,              //除法运算结果
    output reg ready_o                      //除法运算是否结束

);

    wire [32:0] div_temp;
    reg [5:0] cnt;                          //记录试商法进行了几轮
    reg[64:0] dividend;                     //低 32 位保存除数、中间结果，第 k 次迭代结束的时候 dividend[k:0]保存的就是当前得到的中间结果，
                                            //dividend[31:k+1]保存的是被除数没有参与运算的部分，dividend[63:32]是每次迭代时的被减数
    reg [1:0] state;                        //除法器处于的状态
    reg[31:0] divisor;
    reg[31:0] temp_op1;
    reg[31:0] temp_op2;

    assign div_temp = {1'b0, dividend[63: 32]} - {1'b0, divisor};


    always @ (posedge clk) begin
        if (rst) begin
            state <= `DivFree;
            result_o <= {`ZeroWord,`ZeroWord};
            ready_o <= `DivResultNotReady;
        end else begin
            case(state)

                `DivFree: begin         //除法器空闲
                    if (start_i == `DivStart && annul_i == 1'b0) begin
                        if(opdata2_i == `ZeroWord) begin            //如果除数为 0
```

```
                        state <= `DivByZero;
                    end else begin
                        state <= `DivOn;                    //除数不为0
                        cnt <= 6'b000000;
                        if(signed_div_i == 1'b1 && opdata1_i[31] == 1'b1)
begin          //被除数为负数
                            temp_op1 = ~opdata1_i + 1;
                        end else begin
                            temp_op1 = opdata1_i;
                        end
                        if (signed_div_i == 1'b1 && opdata2_i[31] ==
1'b1 ) begin            //除数为负数
                            temp_op2 = ~opdata2_i + 1;
                        end else begin
                            temp_op2 = opdata2_i;
                        end
                        dividend <= {`ZeroWord, `ZeroWord};
                        dividend[32: 1] <= temp_op1;
                        divisor <= temp_op2;
                    end
                end else begin
                    ready_o <= `DivResultNotReady;
                    result_o <= {`ZeroWord, `ZeroWord};
                end
            end

            `DivByZero: begin           //除数为0
                dividend <= {`ZeroWord, `ZeroWord};
                state <= `DivEnd;
            end

            `DivOn: begin               //除数不为0
                if(annul_i == 1'b0) begin           //进行除法运算
                    if(cnt != 6'b100000) begin
                        if (div_temp[32] == 1'b1) begin
                            dividend <= {dividend[63:0],1'b0};
                        end else begin
                            dividend <= {div_temp[31:0],dividend[31:0],
1'b1};
                        end
                        cnt <= cnt +1;      //除法运算次数
                    end else begin
                        if ((signed_div_i == 1'b1) && ((opdata1_i[31] ^
opdata2_i[31]) == 1'b1)) begin
```

```
                            dividend[31:0] <= (~dividend[31:0] + 1);
                        end
                        if ((signed_div_i == 1'b1) && ((opdata1_i[31] ^
dividend[64]) == 1'b1)) begin
                            dividend[64:33] <= (~dividend[64:33] + 1);
                        end
                        state <= `DivEnd;
                        cnt <= 6'b000000;
                    end
                end else begin
                    state <= `DivFree;
                end
            end

            `DivEnd: begin          //除法结束
                result_o <= {dividend[64:33], dividend[31:0]};
                ready_o <= `DivResultReady;
                if (start_i == `DivStop) begin
                    state <= `DivFree;
                    ready_o <= `DivResultNotReady;
                    result_o <= {`ZeroWord, `ZeroWord};
                end
            end

        endcase
    end
end


endmodule
```

五、转移与分支指令实现

5.1 整体设计

分支指令实现的方法即为更改下一条指令的地址，从而实现跳转执行目标指令。

跳转指令(j)：有限的 32 位指令长度对于大型程序的分支跳转支持确实是个难题。MIPS 指令中最小的操作码域占 6 位，剩下的 26 位用于跳转目标的编址。由于所有指令在内存中都是 4 字节对齐的，因此最低的 2 个比特位是无需存储的，这样实际可供寻址范围为 2^28=256MB。分支跳转地址被当做一个 256MB 的段内

绝对地址，而非 PC 相对寻址。这对于地址范围超过 256MB 的跳转程序而言是无能为力的，所幸目前也很少遇到这么大的远程跳转需求。

段外分支跳转可以使用寄存器跳转指令实现，它可以跳转到任意（有效的）32 位地址。

条件分支跳转指令(b)编码域的后 16 位 broffset 是相对 PC 的有符号偏移量，由于指令是 4 字节对齐的，因此可支持的跳转范围实际上是 2^18=256KB（相对 PC 的-128KB~+128KB）。如果确定跳转目标地址在分支指令前后的 128KB 范围内，编译器就可以编码只生成一条简单的条件分支指令。

转移指令，即为将某个寄存器的值取出，搬运至另一个寄存器。本实验中实现的转移（数据搬移）指令一共有四条，分别为 mflo，mfhi，mthi，mtlo，涉及 hilo 寄存器的读取操作。

5.2 代码实现

跳转指令：

```
    wire br_e;
    wire [31:0] br_addr;
    wire rs_eq_rt;
    wire rs_neq_rt;
    wire rs_ge_z;
    wire rs_gt_z;
    wire rs_le_z;
    wire rs_lt_z;
    wire [31:0] pc_plus_4;
    assign pc_plus_4 = id_pc + 32'h4;

    assign rs_eq_rt = (selected_rdata1 == selected_rdata2);
    assign rs_neq_rt = (selected_rdata1 != selected_rdata2);
    assign rs_ge_z = (~selected_rdata1[31]);
    assign rs_gt_z = ($signed(selected_rdata1) > 0);
    assign rs_le_z = (selected_rdata1[31] == 1'b1 || selected_rdata1 ==
32'b0);
    assign rs_lt_z = (selected_rdata1[31] == 1'b1);


    assign br_e = inst_beq & rs_eq_rt | inst_jal | inst_jr | inst_bne &
rs_neq_rt | inst_j
                    | inst_bgez & rs_ge_z | inst_bgtz & rs_gt_z | inst_blez
& rs_le_z
                    | inst_bltz & rs_lt_z | inst_bltzal & rs_lt_z |
inst_bgezal & rs_ge_z
```

```
                    | inst_jalr;
    assign br_addr = inst_beq ? (pc_plus_4 + {{14{inst[15]}}, inst[15:0],
2'b0})
                    : inst_jal ? ({pc_plus_4[31:28], inst[25:0], 2'b0})
                    : inst_jr ? selected_rdata1
                    : inst_bne ? (pc_plus_4 + {{14{inst[15]}}, inst[15:0],
2'b0})
                    : inst_j ? ({pc_plus_4[31:28], inst[25:0], 2'b0})
                    : inst_bgez ? (pc_plus_4 + {{14{inst[15]}}, inst[15:0],
2'b0})
                    : inst_bgtz ? (pc_plus_4 + {{14{inst[15]}}, inst[15:0],
2'b0})
                    : inst_blez ? (pc_plus_4 + {{14{inst[15]}}, inst[15:0],
2'b0})
                    : inst_bltz ? (pc_plus_4 + {{14{inst[15]}}, inst[15:0],
2'b0})
                    : inst_bltzal ? (pc_plus_4 + {{14{inst[15]}}, inst[15:0],
2'b0})
                    : inst_bgezal ? (pc_plus_4 + {{14{inst[15]}}, inst[15:0],
2'b0})
                    : inst_jalr ? selected_rdata1
                    : 32'b0;

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

数据搬移指令：

```
    wire hi_we, lo_we;
    wire [3:0] sel_move_dst;

    //rs move to hi
    assign sel_move_dst[0] = inst_mthi;
    //rs move to lo
    assign sel_move_dst[1] = inst_mtlo;
    //hi move to rd
    assign sel_move_dst[2] = inst_mfhi;
    //lo move to rd
    assign sel_move_dst[3] = inst_mflo;


    assign hi_we = inst_mthi | inst_div | inst_divu | inst_mult | inst_multu;
    assign lo_we = inst_mtlo | inst_div | inst_divu | inst_mult | inst_multu;
```

六、加载与存储指令实现

6.1 总体设计

加载存储指令（load/store）涉及对内存的操作，需要在 ID 段得到访存相关信息，在 EX 段计算访存地址和写入数据，在 MEM 段计算得到读出数据，并送入 WB 段写入目的寄存器。Load 指令涉及数据相关问题，需要在执行 load 指令之后暂停流水线。

6.2 详细设计与相关代码

ID 段：给出需要 ALU 计算的相关信息（略），给出访存相关信息

```
    // load and store enable
    assign data_ram_en = inst_lw | inst_sw | inst_lb | inst_lbu | inst_lh
| inst_lhu | inst_sb | inst_sh;   //if load or store: 1'b1

    // write enable
    //assign data_ram_wen = 1'b0;
    // to simplify the problem, we use data_ram_wen to judge which l/s
instruction is
    assign data_ram_wen = inst_sw  ? 4'b1111
                        : inst_sb  ? 4'b1110
                        : inst_sh  ? 4'b1101
                        : inst_lb  ? 4'b0110
                        : inst_lbu ? 4'b0100
                        : inst_lh  ? 4'b0010
                        : inst_lhu ? 4'b0001
                        : 4'b0;                    //4'b0: lw
```

EX 段：计算写入数据

```
    //load and store instructions
    //assign load_judge = data_ram_wen;   // transfer to mem to judge which
kind of load is

    assign stall_en = data_ram_en
                    & (data_ram_wen == 4'b0 | data_ram_wen == 4'b0001 |
data_ram_wen == 4'b0010
                        | data_ram_wen == 4'b0100 | data_ram_wen ==
4'b0110);
                    //tell "id" to stall if this instruction is a load
instruction
```

```
    assign data_sram_en = data_ram_en;

    assign data_sram_wen = (data_ram_wen == 4'b1101 & alu_result[1:0] ==
2'b00) ? 4'b0011  //sh
                    : (data_ram_wen == 4'b1101 & alu_result[1:0] == 2'b10) ?
4'b1100       //sh
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b00) ?
4'b0001       //sb
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b01) ?
4'b0010       //sb
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b10) ?
4'b0100       //sb
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b11) ?
4'b1000       //sb
                    : (data_ram_wen == 4'b1111) ?
4'b1111                                 //sw
                    :
4'b0;                                                                   //loa
ds


    assign data_sram_addr = alu_result;
    //assign data_sram_wdata = (data_ram_wen == 4'b1111) ? rf_rdata2 : 32'b0;
    assign data_sram_wdata = (data_ram_wen == 4'b1101 & alu_result[1:0] ==
2'b00) ? {16'b0, rf_rdata2[15:0]}
                    : (data_ram_wen == 4'b1101 & alu_result[1:0] == 2'b10) ?
{rf_rdata2[15:0], 16'b0}
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b00) ?
{24'b0, rf_rdata2[7:0]}
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b01) ?
{16'b0,rf_rdata2[7:0], 8'b0}
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b10) ?
{8'b0, rf_rdata2[7:0], 16'b0}
                    : (data_ram_wen == 4'b1110 & alu_result[1:0] == 2'b11) ?
{rf_rdata2[7:0], 24'b0}
                    : (data_ram_wen == 4'b1111) ? rf_rdata2
                    : 32'b0;

    //load and store instructions end
```
MEM 段：计算读出数据传入 WB
```
    assign mem_result = (load_judge == 4'b0110 & ex_result[1:0] == 2'b00) ?
{{24{data_sram_rdata[7]}}, data_sram_rdata[7:0]}  //lb
                    : (load_judge == 4'b0110 & ex_result[1:0] == 2'b01) ?
{{24{data_sram_rdata[15]}}, data_sram_rdata[15:8]}  //lb
```

```
                    : (load_judge == 4'b0110 & ex_result[1:0] == 2'b10) ?
{{24{data_sram_rdata[23]}}, data_sram_rdata[23:16]} //lb
                    : (load_judge == 4'b0110 & ex_result[1:0] == 2'b11) ?
{{24{data_sram_rdata[31]}}, data_sram_rdata[31:24]} //lb
                    : (load_judge == 4'b0100 & ex_result[1:0] == 2'b00) ?
{24'b0, data_sram_rdata[7:0]}                    //lbu
                    : (load_judge == 4'b0100 & ex_result[1:0] == 2'b01) ?
{24'b0, data_sram_rdata[15:8]}                   //lbu
                    : (load_judge == 4'b0100 & ex_result[1:0] == 2'b10) ?
{24'b0, data_sram_rdata[23:16]}                  //lbu
                    : (load_judge == 4'b0100 & ex_result[1:0] == 2'b11) ?
{24'b0, data_sram_rdata[31:24]}                  //lbu
                    : (load_judge == 4'b0010 & ex_result[1:0] == 2'b00) ?
{{16{data_sram_rdata[15]}}, data_sram_rdata[15:0]}  //lh
                    : (load_judge == 4'b0010 & ex_result[1:0] == 2'b10) ?
{{16{data_sram_rdata[31]}}, data_sram_rdata[31:16]} //lh
                    : (load_judge == 4'b0001 & ex_result[1:0] == 2'b00) ?
{16'b0, data_sram_rdata[15:0]}                   //lhu
                    : (load_judge == 4'b0001 & ex_result[1:0] == 2'b10) ?
{16'b0, data_sram_rdata[31:16]}                  //lhu
                    :
data_sram_rdata;
                        //lw

    assign rf_wdata = (sel_rf_res) ? mem_result : ex_result;
```

七、附加模块说明

ALU：算术运算单元，可以执行 add，sub 等十二种基本运算，在 ID 段得到执行相关信息，相关代码如下：

```
    // rs to reg1
    assign sel_alu_src1[0] = inst_ori   | inst_addiu | inst_subu  |
inst_addu
                           | inst_or    | inst_lw    | inst_xor   |
inst_sltu
                           | inst_sw    | inst_slt   | inst_slti  |
inst_sltiu
                           | inst_add   | inst_addi  | inst_sub   |
inst_and
                           | inst_andi  | inst_nor   | inst_xori  |
inst_sllv
                           | inst_srav | inst_srlv  | inst_lb    | inst_lbu
```

```
                          | inst_lh   | inst_lhu   | inst_sb    |
inst_sh;

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

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


    // rt to reg2
    assign sel_alu_src2[0] = inst_subu | inst_addu | inst_sll | inst_or
                          | inst_xor | inst_sltu | inst_slt | inst_add
                          | inst_sub | inst_and  | inst_nor | inst_sllv
                          | inst_sra | inst_srav | inst_srl | inst_srlv;

    // imm_sign_extend to reg2
    assign sel_alu_src2[1] = inst_lui   | inst_addiu | inst_lw   | inst_sw
                          | inst_slti | inst_sltiu | inst_addi | inst_lb
                          | inst_lbu  | inst_lh    | inst_lhu  | inst_sb
                          | inst_sh;

    // 32'b8 to reg2
    assign sel_alu_src2[2] = inst_jal | inst_bltzal | inst_bgezal |
inst_jalr;

    // imm_zero_extend to reg2
    assign sel_alu_src2[3] = inst_ori | inst_xori | inst_andi;



    assign op_add = inst_addiu | inst_addu | inst_jal | inst_lw | inst_sw
| inst_add | inst_addi
                  | inst_bltzal | inst_bgezal | inst_jalr | inst_lb |
inst_lbu | inst_lh
                  | inst_lhu | inst_sb | inst_sh;
    assign op_sub = inst_subu | inst_sub;
    assign op_slt = inst_slt | inst_slti;
    assign op_sltu = inst_sltu | inst_sltiu;
    assign op_and = inst_and | inst_andi;
    assign op_nor = inst_nor;
    assign op_or = inst_ori | inst_or;
    assign op_xor = inst_xor | inst_xori;
    assign op_sll = inst_sll | inst_sllv;
```

```
assign op_srl = inst_srl | inst_srlv;
assign op_sra = inst_sra | inst_srav;
assign op_lui = inst_lui;

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};
```

Decoder：译码器，将指令信息转化为 64 位 one-hot 编码便于操作。

八、组员实验感受

穆奕博：做为我们小组的队长，我深知自己的责任，在写单周期的过程中，问题还是很多的，明白为什么要这么设计，但是通过查阅 MIPS 手册，对整个代码进行一个全局的了解，慢慢的很多地方也是可以搞定的。再不能搞懂的地方就是去问学校好的同学，然后同学就会给解释清楚。我在此期间还参考了《自己动手写 CPU》这本书，当时还不知道这本书是用五级流水线的架构来写的（当时也不知道流水线是什么），然后就参照他这样分为五个阶段，取值译码，执行，访存，回写，把这五个阶段整合到单周期的设计中，后来逐渐了解了流水线的过程，完成了数据相关，气泡暂停的一系列操作，由于之前只是在做计算机上层应用，并没有向下延申和探讨，知识也是从无到有的慢慢增长，在 lw 指令添加时，我们整个小组的步伐都被这个指令停住了，当时我们都不知道如何添加这个指令，学习探索了很久，也问了很多人，其他两个队友的信心受挫，我作为队长一定要攻克这个难题，让队友们重拾信心，最终我们完成了这条指令，之后虽然也遇到了一些困难，但是我们迎难而上，最终不但完成了基础部分还额外完成了附加部分，并且通过了上板测试，我要感谢于老师给了我们这个了解底层的机会，也让我知道人工智能不仅仅是上层应用还有底层加速，同时也要感谢助教老师的帮助和指导。

高天：学习 Verilog 语法，学习 MIPS 架构。从一个单周期 CPU 开始搭建，逐步增加指令，R 型，I 型，J 型指令选择有代表性的一些指令都实现一下。说实话，对于一个 Verilog 都不怎么会的人来说，上手一个单周期也是及其困难的，而且当时计组、体系结构什么的都没有学，像大小端这种比较简单的概念可能都难以区分。这个实验也确实能让人学到了很多，在 lw 指令添加时，我们整个小组的步伐都被这个指令停住了，当时我们都不知道如何添加这个指令，学习探索了很久，也问了很多人，从最开始 vivado 都不会装到后来能写出一个 MIPS CPU，但是在队友和其他同学的鼓励和帮助下，最终完成了实验的主题内容，最终不但完成了基础部分还额外完成了附加部分，并且通过了上板测试同时也收获满满，了解了计算机的底层，丰富了知识，最后要感谢队长的鼓励和助教老师的帮助和指导。

朱梓铭：通过写一个简单的单周期 CPU 来渐渐学习 MIPS 架构、Verilog 语法，了解一些比较常见的名词概念。通过单周期，可以知道条指令的执行需要经

过哪些阶段，若干个模块如何协调工作。对整个流水线的架构有一定的认识与了解。当时还不知道这本书是用五级流水线的架构来写的（当时也不知道流水线是什么），然后就参照他这样分为五个阶段，取值译码，执行，访存，回写，把这五个阶段整合到单周期的设计中，后来逐渐了解了流水线的过程，代码写好之后 debug 的过程是比较漫长的。在 lw 指令添加时，我们整个小组的步伐都被这个指令停住了，当时我们都不知道如何添加这个指令，学习探索了很久，也问了很多人最终在队长的带领下突破了这个难关，最后要感谢队长的鼓励和其他同学的帮助。

九、参考资料

[1] 实验课程仓库 https://github.com/fluctlight001/SampleCPU
[2] 雷思磊“自己动手写 CPU”
[3] 唐硕飞“计算机组成原理 第三版”
[4] 大赛相关文档