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: AI 1901 班

组长: w

组员: l

l

报告日期: 2021.12.18

目录

一、项目分工

项目分工：如表 1 所示

	姓名	完成工作	占比
组长	w	1. 添加分支跳转指令、数据移动指令和访存指令； 2. 完成解决三种数据相关在 ID 段、EX 段需要进行的工作； 3. 完成解决 HI 和 LO 寄存器所出现数据相关在 ID 段和 EX 段的需要进行的工作 4. 完成访存读指令、乘法指令和除法指令所需暂停机制的设计； 5. 完成自制乘法器的设计和实现	40%
组员	l	1. 添加算术运算指令和分支跳转指令； 2. 完成解决三种数据相关在 IF 段、MEM 段和 WB 段需要进行的工作； 3. 完成解决 HI 和 LO 寄存器所出现数据相关在 IF 段和 MEM 段需要进行的工作；	32%
组员	l	1. 添加移位指令与逻辑运算指令； 2. 完成解决三种数据相关在 CTRL、CPU_core 和 Top 部分需要的工作； 3. 完成解决 HI 和 LO 寄存器所出现数据相关在 CTRL 部分、寄存器部分和 WB 段需要进行的工作；	28%

表 1

二、开发工具说明

编程语言：Verilog

编辑工具：Visual Studio Code

开发工具：Vivado

工具版本：2021.1

三、总体设计

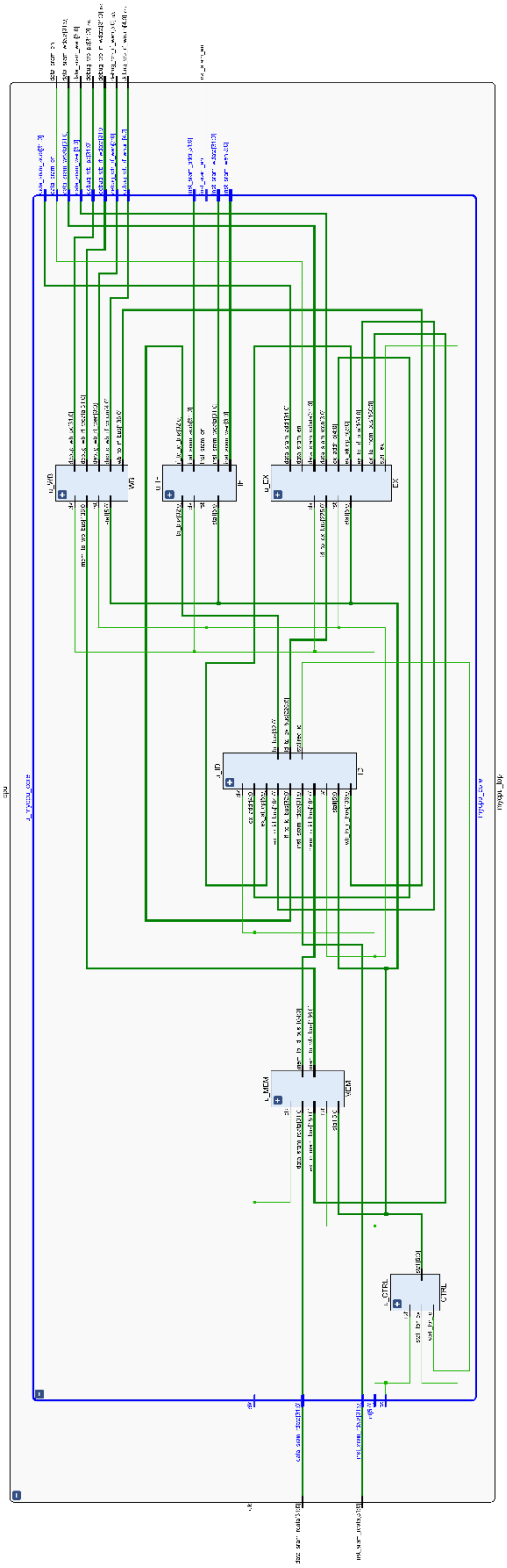

图 1

四、局部设计

4.1 取指（IF）阶段

4.1.1 IF 段结构图

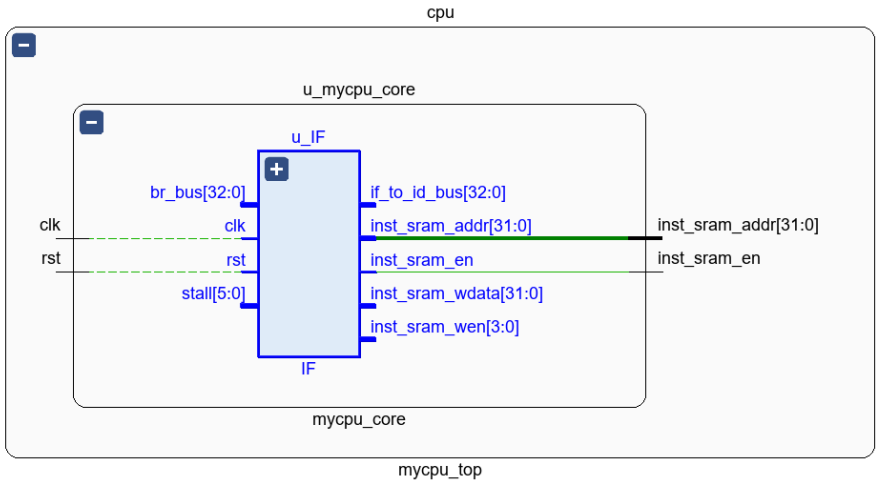

图 2

4.1.2 IF 段整体功能说明

- 根据分支总线传来的信号来更新 PC 值。
- 传递 PC 值到 ID 段。
- 传递 PC 值到指令存储器，使其在下一周期将指令传递到 EX 段。

4.1.3 IF 段端口描述

序号	接口名	宽度(bit)	输入\输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	br_bus	33	输入	传递分支总线的信号
4	stall	6	输入	暂停流水线信号
5	if_to_id_bus	33	输出	PC 使能信号及 PC 值
6	inst_sram_en	1	输出	指令存储器使能信号
7	inst_sram_addr	32	输出	要读取指令的地址
8	inst_sram_wen	4	输出	指令存储器写使能信号

9	inst_sram_wdata	32	输出	要写入的指令

表 1

4.1.4 IF 段信号描述

序号	信号名	宽度(bit)	类型	作用
1	br_e	1	wire	分支总线使能信号
2	br_addr	32	wire	分支总线跳转目标地址
3	ce_reg	1	reg	PC 使能信号
4	pc_reg	32	reg	PC 寄存器
5	next_pc	32	wire	暂时存储下一个 PC 值

表 2

4.1.5 IF 段功能模块说明

1）更新 PC 值的原理图

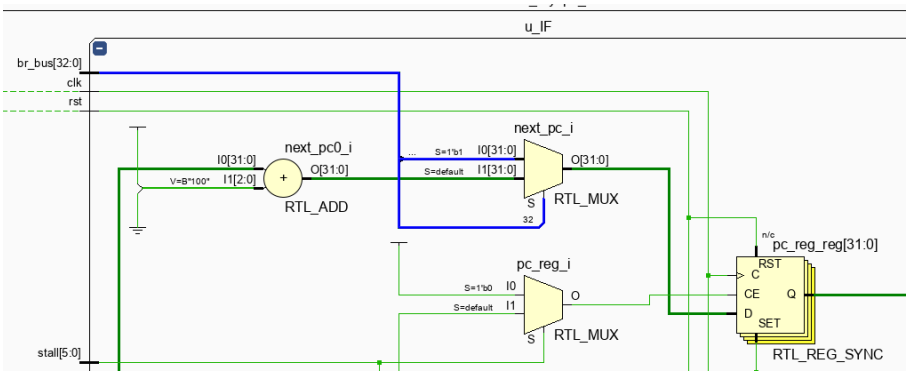

图 3

• 更新 PC 值的代码及代码说明

```
1.  assign next_pc = br_e ? br_addr
2.                       : pc_reg + 32'h4;//只有分支使能时 npc 为总线传来的地址 否则为当前 pc+4 的地址
```

代码说明：pc_reg+32’h4 对应图中的 RTL 加法器，用来计算按正常顺序往下的一条指令的 PC 值，分支总线 br_bus 传来的是要跳转到的指令对应的 PC 值，整个代码为一个选择语句，对应图中的 RTL_MUX 多路选择器，根据 br_e 选择出正确的 PC 值。

```
1. always @ (posedge clk) begin
2.         if (rst) begin
3.             pc_reg <= 32'hbfbf_fffc;//被禁用的时候 PC 为某一定值
4.         end
5.         else if (stall[0]==`NoStop) begin
6.             pc_reg <= next_pc;// PC 寄存器与 NPC 寄存器连线
7.         end
8.     end
```

代码说明：该段代码对应图中的 RTL_REG_SYNC 部分，该部分根据时钟信号、复位信号和暂停信号来决定是否更新传给 ID 段和指令存储器的 PC 值。

4.2 译码（ID）阶段

4.2.1 ID 段结构图

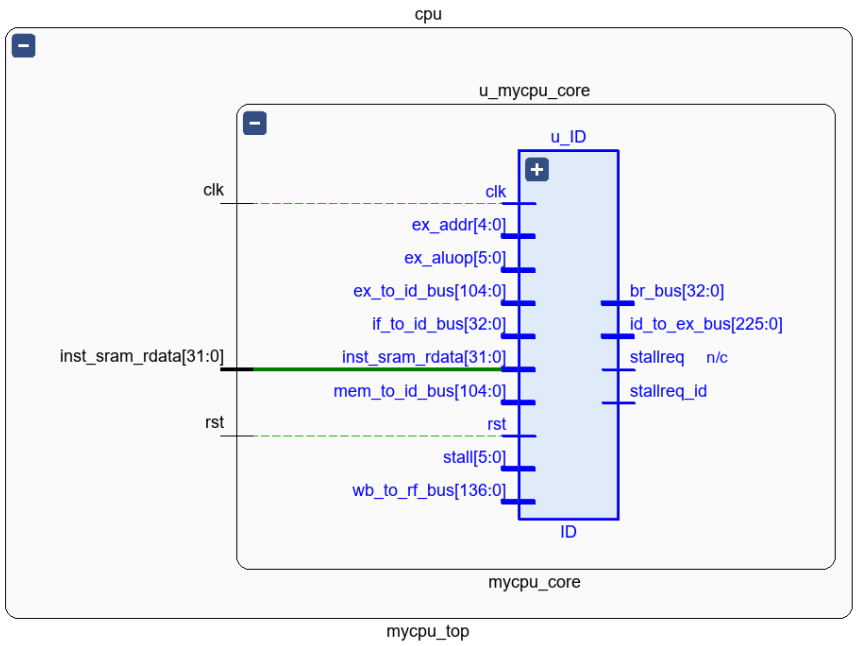

图 4

4.2.2 ID 段整体功能说明

- 确定当前指令的类型。
- 确定两个操作数的来源。
- 确定运算操作的类型。
- 确定运算结果的存储位置。
- 根据数据相关情况，从寄存器、EX 段回传数据、MEM 段回传数据和 WB 段回传数据中选择正确的数据作为操作数。
- 对于跳转指令，确定是否要跳转并计算出跳转的目标地址，将结果通过分支总线回传到 IF 段。
- 对于 EX 段传来的 load 指令信号，向 CTRL 模块发出信号，并由 CRTL 模块修改暂停信号发往流水线各段。
- 向 EX 段传递数据信号。

4.2.3 ID 段端口描述

序号	接口名	宽度(bit)	输入\输出	作用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	br_bus	33	输入	传递分支总线的信号
4	stall	6	输入	暂停流水线信号
5	inst_sram_rdata	32	输入	译码所需指令
6	ex_addr	5	输入	EX 段 load 访存地址
7	ex_aluop	6	输入	EX 段 load 访存类型
8	id_to_ex_bus	226	输出	向 EX 段传递的信号
9	stallreq_id	1	输出	暂停信号
10	if_to_id_bus	33	输入	来自 IF 段的信号
11	ex_to_id_bus	105	输入	EX 段回传的信号
12	mem_to_id_bus	105	输入	MEM 段回传的信号
13	wb_to_id_bus	137	输入	WB 段回传的信号

表 3

4.2.4 ID 段信号描述

序号	信号名	宽度(bit)	类型	作用
1	br_e	1	wire	分支总线使能信号
2	br_addr	32	wire	分支总线跳转目标地址
3	opcode	6	wire	对应指令的操作码段
4	rs	5	wire	对应指令的 rs 寄存器段
5	func	6	wire	对应指令的功能码段
6	op_d	64	wire	操作码对应 64 位独热码
7	rs_d	32	wire	rs 寄存器对应 32 位独热码
8	sel_alu_src1	3	wire	选择 src1 来源信号
9	sel_alu_src2	4	wire	选择 src2 来源信号
10	alu_op	12	wire	运算类型独热码
11	data_ram_en	1	wire	访存使能信号
12	data_ram_wen	4	wire	访存写使能信号
13	rf_we	1	wire	访问普通寄存器使能信号
14	rf_waddr	5	wire	访问普通寄存器地址

15	hl_we	1	wire	访问 HILO 寄存器使能信号
16	hl_waddr	2	wire	访问 HILO 寄存器地址
17	sel_rf_res	1	wire	结果来源使能信号
18	sel_rf_dst	3	wire	结果存储位置
19	data1	32	wire	从普通寄存器中取出的数
20	p_data	64	wire	从 HILO 寄存器中取出的数
21	ex_rf_we	1	wire	EX 段回传的访问寄存器使能
22	ex_rf_waddr	5	wire	EX 段回传的寄存器地址
23	ex_rf_wdata	32	wire	EX 段回传的数据
24	ex_inst_isload	1	wire	EX 段是否在执行 LOAD 指令
25	inst	32	wire	指令
26	id_pc	32	wire	指令的 PC
27	ce	1	wire	使能信号
28	id_stop	1	reg	ID 段暂停信号
29	Inst_ori	1	wire	Ori 指令使能信号
30	Inst_lui	1	wire	Ori 指令使能信号

表 4

说明：由于存在大量的与门、或门等器件导致结构异常复杂同时实现代码所占篇幅过长，接下来将不再展示具有复杂结构图的功能模块并仅给出部分代码，完整项目代码见 github 仓库。

4.2.5 ID 段功能模块说明

1)Load 指令暂停机制

```
reg id_stop;
    always @ (posedge clk) begin
        if (rst) begin
            if_to_id_bus_r <= `IF_TO_ID_WD'b0;
            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;
            id_stop <=1'b1;
        end
        else if (stall[1]==`NoStop) begin
            if_to_id_bus_r <= if_to_id_bus;
```

```
            id_stop <=1'b0;
        end
        else if (stall[1]==`Stop && stall[2]==`Stop) begin
            id_stop <=1'b1;
        end
    end


    assign ex_inst_isload = (ex_aluop==6'b10_0000)|
                            (ex_aluop==6'b10_0100)|
                            (ex_aluop==6'b10_0001)|
                            (ex_aluop==6'b10_0101)|
                            (ex_aluop==6'b10_0011);
    assign stallreq_id=(ex_inst_isload==1'b0)?1'b0:
                        (ex_addr==rs|ex_addr==rt)?1'b1:1'b0;
    assign inst=id_stop?inst:inst_sram_rdata;//PC 对应的指令码
```

根据 EX 段回传的 ex_aluop 和 ex_addr 来判断 EX 段是否正在执行 Load 指令并将结果 stallreq_id 传给 CTRL 模块，模块根据 stallreq_id 修改 Stall 并回传给 ID 段，ID 段根据 stall 的值判断是否暂停，如果需要暂停，则将 id_stop 置 1 并暂停 ID 段，当不需要暂停时，根据 id_stop 的值，通过选择语句恢复因暂停而未执行的指令。

2)解析指令的类型

```
    wire inst_ori, inst_lui, inst_addiu;
    wire inst_sub,inst_slt,inst_sltu,inst_slti,inst_sltiu;
    wire inst_and,inst_nor,inst_or,inst_xor;
    wire inst_sll,inst_srl,inst_sra;
    wire inst_subu;
    wire inst_addu;
    wire inst_add;
    wire inst_addi;
    wire inst_andi;
    wire inst_xori;
    wire inst_sllv;
    wire inst_srav;
    wire inst_srlv;
    //load store
    wire inst_lb,inst_lbu,inst_lh,inst_lhu,inst_lw;
    wire inst_sb,inst_sh,inst_sw;
    wire op_add, op_sub, op_slt, op_sltu;
    wire op_and, op_nor, op_or, op_xor;
    wire op_sll, op_srl, op_sra, op_lui;
    //跳转指令
    wire inst_j,inst_jal,inst_jr,inst_jalr;
    //分支指令
```

```
    wire inst_beq,inst_bne,inst_bgez,inst_bgtz,inst_blez,inst_bltz,inst_bgezal,inst_bltzal;
    //数据移动指令
    wire inst_mfhi,inst_mflo,inst_mthi,inst_mtlo;
    // 乘除指令
    wire inst_mult,inst_multu,inst_div,inst_divu;
    //new
    wire inst_lsa;
    decoder_6_64 u0_decoder_6_64(
     .in  (opcode  ),
        .out (op_d )//运算的独热编码，只有一位为 1 表明该条指令的运算类型为这一种
    );

    decoder_6_64 u1_decoder_6_64(
     .in  (func  ),
        .out (func_d )//func 的独热编码，只有一位为 1 表明该条指令的运算类型为这一种
    );

    decoder_5_32 u0_decoder_5_32(
     .in  (rs  ),
        .out (rs_d )//rs 对应寄存器的独热编码，
    );

    decoder_5_32 u1_decoder_5_32(
     .in  (rt  ),
        .out (rt_d )//rt 对应寄存器的独热编码，
    );

    decoder_5_32 u2_decoder_5_32(
     .in  (rd  ),
        .out (rd_d )//rt 对应寄存器的独热编码，
    );

    decoder_5_32 u3_decoder_5_32(
     .in  (sa  ),
        .out (sa_d )//rt 对应寄存器的独热编码，
    );


    assign inst_ori     = op_d[6'b00_1101];//6'b00_1101 表示索引，取独热码的这一位；
```

```
93.     //同时 00_1101 为 ori 运算的指令码 如果该独热码是 ori 的独热码，那么会取
   出 1 否则 取出 0
94.     assign inst_lui     = op_d[6'b00_1111];
95.     assign inst_addiu   = op_d[6'b00_1001];
96.     //新的运算类型对应的使能信号
97.     assign inst_sub     = op_d[6'b00_0000]&func_d[6'b10_0010];
98.     assign inst_subu    = op_d[6'b00_0000]&func_d[6'b10_0011];
99.     assign inst_slt     = op_d[6'b00_0000]&func_d[6'b10_1010];
100.       assign inst_sltu    = op_d[6'b00_0000]&func_d[6'b10_1011];
101.       assign inst_slti    = op_d[6'b00_1010];
102.       assign inst_sltiu   = op_d[6'b00_1011];
103.       assign inst_and     = op_d[6'b00_0000]&func_d[6'b10_0100];
104.       assign inst_nor     = op_d[6'b00_0000]&func_d[6'b10_0111];
105.       assign inst_or      = op_d[6'b00_0000]&func_d[6'b10_0101];
106.       assign inst_xor     = op_d[6'b00_0000]&func_d[6'b10_0110];
107.       assign inst_sll     = op_d[6'b00_0000]&func_d[6'b00_0000];
108.       assign inst_srl     = op_d[6'b00_0000]&func_d[6'b00_0010];
109.       assign inst_sra     = op_d[6'b00_0000]&func_d[6'b00_0011];
110.       assign inst_addu    = op_d[6'b00_0000]&func_d[6'b10_0001];
111.       assign inst_add     = op_d[6'b00_0000]&func_d[6'b10_0000];
112.       assign inst_addi    = op_d[6'b00_1000];
113.       assign inst_andi    = op_d[6'b00_1100];
114.       assign inst_xori    = op_d[6'b00_1110];
115.       assign inst_sllv    = op_d[6'b00_0000]&func_d[6'b00_0100];
116.       assign inst_srav    = op_d[6'b00_0000]&func_d[6'b00_0111];
117.       assign inst_srlv    = op_d[6'b00_0000]&func_d[6'b00_0110];
118.       //load store 指令
119.       assign inst_lb     =op_d[6'b10_0000];
120.       assign inst_lbu    =op_d[6'b10_0100];
121.       assign inst_lh     =op_d[6'b10_0001];
122.       assign inst_lhu    =op_d[6'b10_0101];
123.       assign inst_lw     =op_d[6'b10_0011];
124.       assign inst_sb     =op_d[6'b10_1000];
125.       assign inst_sh     =op_d[6'b10_1001];
126.       assign inst_sw     =op_d[6'b10_1011];
127.       //跳转指令
128.       assign inst_jr      =op_d[6'b00_0000]&&func_d[6'b00_1000];
129.       assign inst_jalr    =op_d[6'b00_0000]&&func_d[6'b00_1001];
130.       assign inst_jal     =op_d[6'b00_0011];
131.       assign inst_j       =op_d[6'b00_0010];
132.       //分支指令
133.       assign inst_beq     = op_d[6'b00_0100];//==
134.       assign inst_bne     = op_d[6'b00_0101];//! =
135.       assign inst_bgez    = op_d[6'b00_0001]&rt_d[5'b00001];//>=0
```

```
136.      assign inst_bgtz    = op_d[6'b00_0111];//>0
137.      assign inst_blez    = op_d[6'b00_0110];//<=0
138.      assign inst_bltz    = op_d[6'b00_0001]&rt_d[5'b00000];//<0
139.      assign inst_bgezal  = op_d[6'b00_0001]&rt_d[5'b10001];//>=0
140.      assign inst_bltzal  = op_d[6'b00_0001]&rt_d[5'b10000];//<0
141.      //数据移动指令
142.      assign inst_mfhi    = op_d[6'b00_0000]&sa_d[5'b00000]&func_d[
   6'b01_0000];
143.      assign inst_mflo    = op_d[6'b00_0000]&sa_d[5'b00000]&func_d[
   6'b01_0010];
144.      assign inst_mthi    = op_d[6'b00_0000]&sa_d[5'b00000]&func_d[
   6'b01_0001];
145.      assign inst_mtlo    = op_d[6'b00_0000]&sa_d[5'b00000]&func_d[
   6'b01_0011];
146.      //乘除
147.      assign inst_mult    = op_d[6'b00_0000]&sa_d[5'b00000]&func_d[
   6'b01_1000];
148.      assign inst_multu   = op_d[6'b00_0000]&sa_d[5'b00000]&func_d[
   6'b01_1001];
149.      assign inst_div     = op_d[6'b00_0000]&sa_d[5'b00000]&func_d[
   6'b01_1010];
150.      assign inst_divu    = op_d[6'b00_0000]&sa_d[5'b00000]&func_d[
   6'b01_1011];
151.      //new
152.      assign inst_lsa     = op_d[6'b01_1100]&func_d[6'b11_0111];
153.      // rs to reg1
154.      assign sel_alu_src1[0] = inst_ori|inst_addiu|inst_add|inst_ad
   du|inst_addi|inst_sllv|inst_srav|inst_srlv
155.                               |inst_sub|inst_subu|inst_slt|inst_sl
   tu|inst_slti|inst_sltiu
156.                               |inst_and|inst_andi|inst_nor|inst_or
   |inst_xor|inst_xori
157.                               |inst_lw|inst_lb|inst_lbu|inst_lh|in
   st_lhu|inst_sb|inst_sh|inst_sw
158.                               |inst_mthi|inst_mtlo
159.                               |inst_mult|inst_multu|inst_div|inst_
   divu
160.                               |inst_lsa;//rs 的值作 src1
161.      assign op_add = inst_addiu|inst_addu|inst_add|inst_addi
162.                                |inst_lw|inst_lb|inst_lbu|inst_lh|i
   nst_lhu|inst_sb|inst_sh|inst_sw
163.                                |inst_bgezal|inst_bltzal|inst_jal|i
   nst_jalr
164.                                |inst_mthi|inst_mtlo
```

```
                                  |inst_lsa;
      assign op_sub = inst_sub |inst_subu;
      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};//op 独热码

```

以 sub 指令为例，sub 指令为 R 类指令，需要通过操作码和功能码来进行识别，因此将指令操作码的独热码与功能码的独热码进行逻辑与运算，如果该指令为 sub 指令，inst_sub 将置 1，op_sub 也将置 1，op_sub 在 alu_op 中对应的位置也会置 1，表明该指令为 sub 指令。

3)解析指令的操作数来源

```
// pc to reg1
    assign sel_alu_src1[1] = 1'b0|inst_bgezal|inst_bltzal|inst_jal|inst_jalr;//src1 为pc 值

    // sa_zero_extend to reg1
    assign sel_alu_src1[2] = 1'b0|inst_sll|inst_srl|inst_sra;//无符号扩展数做src1

    // rt to reg2
    assign sel_alu_src2[0] = 1'b0|inst_sub|inst_subu|inst_slt|inst_sltu|inst_and|inst_srav|inst_srlv
                                  |inst_nor|inst_or|inst_xor|inst_sll|inst_srl|inst_sra|inst_addu|inst_add|inst_sllv
                                  |inst_mult|inst_multu|inst_div|inst_divu
                                  |inst_lsa;//rt 作src2

    // imm_sign_extend to reg2
    assign sel_alu_src2[1] = 1'b0|inst_lui|inst_addiu|inst_lw|inst_lb|inst_lbu|inst_lh|inst_lhu|inst_sb|inst_sh|inst_sw|inst_slti|inst_sltiu|inst_addi;//有符号扩展数作src2

```

```
    // 32'b8 to reg2
    assign sel_alu_src2[2] = 1'b0|inst_bgezal|inst_bltzal|inst_jal|inst_jalr;//32 位的整数 8 作为 src2

    // imm_zero_extend to reg2
    assign sel_alu_src2[3] = 1'b0|inst_ori|inst_andi|inst_xori;//立即数的无符号扩展作 src2
```

src1 有 3 种情况，src2 有 4 种情况；通过分析每种指令两个操作数的来源分别将其指令使能信号赋给对应位置的 sel_alu_src 以解析操作数来源。例如：sub 指令的两个操作数分别来自 rs 与 rt 寄存器，则将 inst_sub 仅放在 sel_alu_src1[0]和 sel_alu_src2[0]后，当指令为 sub 指令时，sel_alu_src1 为 3’b001,sel_alu_src2 为 4’b0001，表明两个操作数分别来源于 rs 和 rt 寄存器。

4)解析运算结果的存储位置

```
// load and store enable
    assign data_ram_en = inst_lw|inst_lb|inst_lbu|inst_lh|inst_lhu|inst_sb|inst_sh|inst_sw;

    // write enable
    assign data_ram_wen = inst_sb?4'b0001:
                          inst_sh?4'b0011:
                          inst_sw?4'b1111:
                          inst_lb|inst_lbu|inst_lh|inst_lhu|inst_lw?4'b0000:4'b1110;
    //一个字节 还是 半字 还是字

    // regfile sotre enable
    assign rf_we = inst_ori |inst_lui|inst_sub|inst_subu|inst_srav|inst_srlv
                          |inst_and|inst_andi|inst_nor|inst_or|inst_xor|inst_xori
                          |inst_add|inst_addi|inst_addiu|inst_addu|inst_sllv
                          |inst_slt|inst_slti|inst_sltiu|inst_sltu|inst_sll|inst_srl|inst_sra
                          |inst_bgezal|inst_bltzal|inst_jal|inst_jalr
                          |inst_lw|inst_lb|inst_lbu|inst_lh|inst_lhu
                          |inst_mfhi|inst_mflo
                          |inst_lsa;

    assign hl_we = inst_mthi|inst_mtlo
```

```
                    |inst_mult|inst_multu|inst_div|inst_divu;//结果是不是在hilo 寄存器上写

    // store in [rd]
    assign sel_rf_dst[0] = 1'b0|inst_sub|inst_subu|inst_addu|inst_and|inst_sllv|inst_srav|inst_srlv
                                |inst_nor|inst_or|inst_xor|inst_slt|inst_sltu|inst_sll|inst_srl|inst_sra|inst_add|inst_jalr
                                |inst_mfhi|inst_mflo
                                |inst_lsa;
    // store in [rt]
    assign sel_rf_dst[1] = inst_ori|inst_lui|inst_addiu|inst_lw|inst_lb|inst_lbu|inst_lh|inst_lhu|inst_slti|inst_sltiu|inst_sltiu|inst_addi|inst_andi|inst_xori;
    // store in [31]
    assign sel_rf_dst[2] = 1'b0|inst_bgezal|inst_bltzal|inst_jal;

    // sel for regfile address
    assign rf_waddr = {5{sel_rf_dst[0]}} & rd
                    | {5{sel_rf_dst[1]}} & rt
                    | {5{sel_rf_dst[2]}} & 32'd31;//选择结果存储的寄存器

    assign hl_waddr[0] = inst_mult|inst_multu|inst_div|inst_divu;//乘除法 hi 和lo 寄存器都要使用 数据移动指令仅使用hi 或 lo
    assign hl_waddr[1] = inst_mtlo ? 1'b1:1'b0;//判断结果写在lo 还是 hi
    // 0 from alu_res ; 1 from ld_res
    assign sel_rf_res = 1'b0|inst_lw|inst_lb|inst_lbu|inst_lh|inst_lhu; //选择存储寄存器中的值的来源
```

说明：指令最后要么访存，要么访问普通寄存器，要么访问 HILO 寄存器。因此需要设置不同的变量来区分这 3 种访问方式：

• 对于访存的指令：以 lw 和 sw 指令为例：当指令为 lw 或 sw 指令时，data_ram_en 置 1，data_ram_wen 也要根据访存指令的不同进行修改，访存读指令如 lw，会在 MEM 段判断结果长度，访存写指令如 sw 会在 EX 段判断结果长度。同时 lw 指令还需要将 rf_we 和 sel_rf_dst[1]置 1 表明访存结果要存到 rt 寄存器中，将 sel_rf_res 置 1 在告诉 MEM 段结果来源于访存。

• 对于访问普通寄存器的指令：以 sub 指令为例，由于 sub 指令的结果需要存储到 rd 寄存器中，则将 rf_we 置 1，sel_rf_dst[0]置 1，sel_rf_res 置 0 表明结果为 EX 段的计算值。

• 对于访问 HILO 寄存器的指令：以 mult、mtlo 和 mflo 指令为例。这三个指令均会使用到 HILO 寄存器因此将 hl_we 置 1，对于 mult 指令，它最后要同时使用 HI 寄存器和 LO 寄存器以存储结果，因此需要将 hl_waddr[0]置 1 表明指令需

要同时使用 HI 寄存器和 LO 寄存器；对于 mtlo 指令，仅用到 LO 寄存器，因此将 hl_waddr[0]置 0，hl_waddr[1]置为 Lo 寄存器的地址即可；对于 mflo 指令的最后结果存储到 rd 寄存器中，因此需要将 rf_we 和 sel_rf_dst[0]置 1。最后由于三个指令不涉及访存，因此需要将 sel_rf_res 置 0。

5)跳转指令解析

```
assign br_e = inst_j|inst_jr|inst_jal|inst_jalr
                        |(inst_beq&rs_eq_rt)|(inst_bne&rs_neq_rt)
                        |(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)
;
    assign br_addr =(inst_j|inst_jal)?{pc_plus_4[31:28],instr_index,
2'b0}:
                    ((inst_jr|inst_jalr)?data1:
                    (inst_beq|inst_bne|inst_bgez|inst_bgtz|inst_blez
|inst_bltz|inst_bgezal|inst_bltzal)?(pc_plus_4 + {{14{offset[15]}},o
ffset,2'b0}):32'b0 );
```

对于跳转指令，先解析出跳转的类型，然后结合跳转类型和跳转条件对跳转使能信号 br_e 进行赋值，并跟据跳转类型利用选择语句对目标地址 br_addr 进行计算并赋值。

6)数据相关处理

```
 assign data1=(ex_rf_we==1'b1)&&(ex_rf_waddr==rs)? ex_rf_wdata:
               (mem_rf_we==1'b1)&&(mem_rf_waddr==rs)? mem_rf_wdata:
             (wb_rf_we==1'b1)&&(wb_rf_waddr==rs)? wb_rf_wdata:rdata1;

assign data2=(ex_rf_we==1'b1)&&(ex_rf_waddr==rt)?ex_rf_wdata:
              (mem_rf_we==1'b1)&&(mem_rf_waddr==rt)?mem_rf_wdata:
              (wb_rf_we==1'b1)&&(wb_rf_waddr==rt)?wb_rf_wdata:rdata2;
```

注意：本项目将 HILO 寄存器与 regfile 寄存器进行共用，即 HILO 寄存器的实现也在 regfile.v 文件中实现，因此 HILO 寄存器与普通寄存器的数据相关处理方法相同，在此仅对解决普通寄存器对应的数据相关问题的代码进行说明：

根据 EX 段、MEM 段和 WB 段回传的信号进行判断，如果使能信号为 1，并且地址与 ID 段所需要的地址相同，则无需访问寄存器直接将数据传递给 ID 段。对于数据来源于 EX 还是 MEM 亦或是 WB，通过选择语句即可实现。

4.3 执行（EX）阶段

4.3.1 EX 段结构图

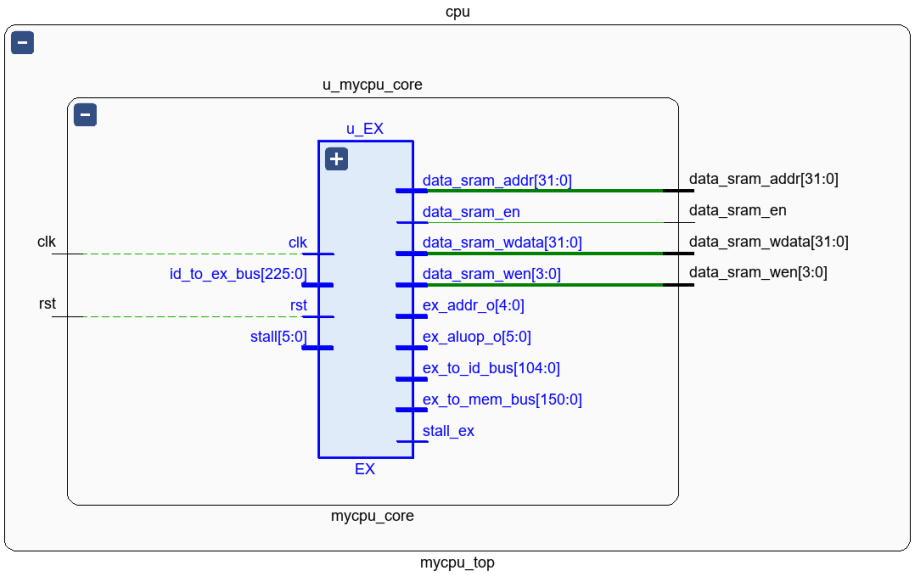

图 5

4.3.2 EX 段整体功能说明

• 进行算术、移位和逻辑运算。
• 对于乘法和除法指令，向 CTRL 模块发出暂停信号，并由 CTRL 发出流水线暂停信号。
• 对于 Load 访存指令，向 ID 段发出信号，表明 EX 段正在执行 load 访存指令。
• 对于访存指令更新访存信号、访存地址和要写的数据，发送给数据存储器。
• 完成 EX 段的功能后，将数据回传到 ID 段以解决数据相关问题。
• 向 MEM 段传递数据信号。

4.3.3 EX 段端口描述

序号	接口名	宽度(bit)	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	暂停流水线信号
4	id_to_ex_bus	226	输出	向 EX 段传递的信号
5	ex_addr	5	输出	EX 段 load 访存地址

6	ex_aluop	6	输出	EX 段 load 访存类型
7	stall_ex	1	输出	暂停信号
8	data_sram_en	1	输出	访存使能信号
9	data_sram_wen	4	输出	访存写使能信号
10	data_sram_addr	32	输出	访存地址
11	data_sram_wdata	32	输出	访存写数据
12	ex_to_id_bus	105	输出	回传 ID 段的信号
13	ex_to_mem_bus	151	输出	传递给 MEM 段的信号

表 5

4.3.4 EX 段信号描述

序号	信号名	宽度(bit)	类型	作用
1	ex_stop	1	reg	EX 段暂停信号
2	alu_src1	32	wire	操作数 1
3	alu_result	32	wire	ALU 计算结果
4	ex_result	32	wire	EX 段最终结果
5	ld_type_o	6	wire	LOAD 指令的类型
6	addr_ram_o	2	wire	访存地址的低前两位
7	muldiv_mt_hl	64	wire	要访问 HILO 寄存器的数据
8	ex_to_iden	1	wire	回传 ID 段使能信号
9	ex_to_iregaddr	5	wire	回传 ID 段访问寄存器地址
10	ex_to_idata	32	wire	回传 ID 段数据

表 6

4.3.5 EX 段功能模块说明

1)乘除法暂停机制

```
9.      reg ex_stop;
10.     always @ (posedge clk) begin
11.         if (rst) begin
12.             id_to_ex_bus_r <= `ID_TO_EX_WD'b0;
13.         end
14.         else if (stall[2]==`Stop && stall[3]==`NoStop) begin
15.             id_to_ex_bus_r <= `ID_TO_EX_WD'b0;
16.             ex_stop=1'b1;
17.         end
18.         else if (stall[2]==`NoStop) begin
```

```
            id_to_ex_bus_r <= id_to_ex_bus;
            ex_stop=1'b0;
        end
        else if(stall[2]==`Stop && stall[3]==`Stop) begin
            ex_stop=1'b1;
        end
    end
    assign id_to_ex_bus_rr=id_to_ex_bus_r;

    wire [31:0] ex_pc, inst;//指令地址以及指令
    wire [11:0] alu_op;//运算类型
    wire [2:0] sel_alu_src1;//操作数 1 来源
    wire [3:0] sel_alu_src2;//操作数 2 来源
    wire data_ram_en;
    wire [3:0] data_ram_wen;//访存信号
    wire rf_we;//读寄存器使能信号
    wire [4:0] rf_waddr;//寄存器的位置
    wire hl_we;//hl we
    wire [1:0]hl_waddr;// hl address
    wire sel_rf_res;//选择的结果
    wire [31:0] rf_rdata1, rf_rdata2;//从寄存器中读入的数据
    wire [63:0] hl_rdata;//hl data
    assign ex_aluop_o=inst[31:26];
    assign ex_addr_o =rf_waddr;
    assign {
        hl_rdata,       //225:162
        hl_waddr,       //161:160
        hl_we,          //159
        ex_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
        rf_rdata1,      // 63:32
        rf_rdata2       // 31:0
    } = ex_stop?id_to_ex_bus_rr:id_to_ex_bus_r;
```

根据乘法或者除法发出的暂停信号,将stall_ex进行赋值并传递到CTRL段，CTRL 段修改 Stall 信号并传输到流水线各段。EX 段根据 stall 的值判断是否暂停，如果需要暂停，则将 ex_stop 置 1 并暂停 EX 段，当不需要暂停时，根据

ex_stop 的值，通过选择语句恢复因暂停而未使用的指令和数据。

2)执行访存指令

```
1. assign data_sram_en=data_ram_en;
2. assign data_sram_wen=(data_ram_wen==4'b0001)&&(data_sram_addr[1:0]==
   2'b00)?4'b0001:
3.                          (data_ram_wen==4'b0001)&&(data_sram_addr[1
   :0]==2'b01)?4'b0010:
4.                          (data_ram_wen==4'b0001)&&(data_sram_addr[1
   :0]==2'b10)?4'b0100:
5.                          (data_ram_wen==4'b0001)&&(data_sram_addr[1
   :0]==2'b11)?4'b1000:
6.                          (data_ram_wen==4'b0011)&&(data_sram_addr[1
   :0]==2'b00)?4'b0011:
7.                          (data_ram_wen==4'b0011)&&(data_sram_addr[1
   :0]==2'b10)?4'b1100:
8.                          (data_ram_wen==4'b1111)?4'b1111:4'b0000;
9. assign data_sram_addr=data_ram_en?ex_result:32'd0;
10.assign data_sram_wdata=(data_ram_wen==4'b0001)&&(data_sram_addr[1:0]
   ==2'b00)?{24'b0,rf_rdata2[7:0]}:
11.                           (data_ram_wen==4'b0001)&&(data_sram_addr
   [1:0]==2'b01)?{16'b0,rf_rdata2[7:0],8'b0}:
12.                           (data_ram_wen==4'b0001)&&(data_sram_addr
   [1:0]==2'b10)?{8'b0,rf_rdata2[7:0],16'b0}:
13.                           (data_ram_wen==4'b0001)&&(data_sram_addr
   [1:0]==2'b11)?{rf_rdata2[7:0],24'b0}:
14.                           (data_ram_wen==4'b0011)&&(data_sram_addr
   [1:0]==2'b00)?{16'b0,rf_rdata2[15:0]}:
15.                           (data_ram_wen==4'b0011)&&(data_sram_addr
   [1:0]==2'b10)?{rf_rdata2[15:0],16'b0}:
16.                           (data_ram_wen==4'b1111)?rf_rdata2:32'd0;
```

若是访存指令，需将访存使能信号 data_sram_en 置 1，同时通过选择语句并根据访存类型—要访问一个字节、半个字还是一个字对 data_sram_wen 和 data_sram_wdata 进行赋值。确保写入主存或从主存中读出的数据是正确的。

4.4 访存（MEM）阶段

4.4.1 MEM 段结构图

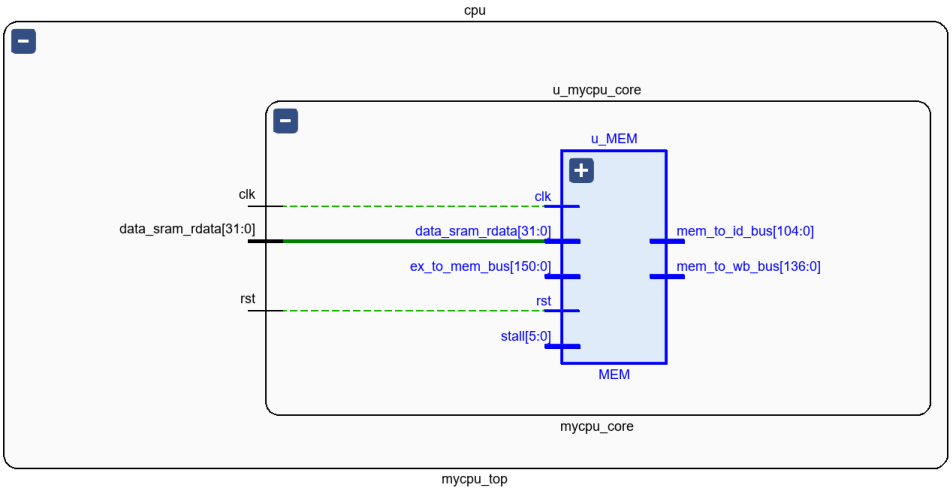

图 6

4.4.2 MEM 段整体功能说明

• 接受访存回传的数据信号，并根据指令类型在 EX 段传来的数据与访存数据间选择正确的结果。
• 回传信号到 ID 段以解决数据相关问题。
• 传递信号到 WB 段

4.4.3 MEM 段端口描述

序号	接口名	宽度(bit)	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	暂停流水线信号
4	data_sram_rdata	32	输入	数据寄存器传递的数据
5	ex_to_mem_bus	151	输入	EX 段传递的信号
6	mem_to_id_bus	105	输出	回传到 ID 段的信号
7	mem_to_wb_bus	137	输出	回传到 WB 段的信号

表 7

4.4.4 MEM 段信号描述

序号	接口名	宽度(bit)	类型	作用
1	mem_to_iden	1	wire	回传 ID 段使能信号
2	mem_to_idregaddr	5	wire	回传 ID 段寄存器地址
3	mem_to_idata	32	wire	回传 ID 段数据
4	mem_result	32	wire	访存结果
5	ld_type_i	6	wire	LOAD 指令类型
6	addr_ram_i	2	wire	访存地址低前两位

表 8

4.4.5 MEM 段功能模块说明

1)访存指令的结果处理

```
1. assign mem_result=(ld_type_i==6'b10_0000)&&(addr_ram_i==2'b00)?{{24{
   data_sram_rdata[7]}},data_sram_rdata[7:0]}:
2.                          (ld_type_i==6'b10_0000)&&(addr_ram_i==2'b01)?
   {{24{data_sram_rdata[15]}},data_sram_rdata[15:8]}:
3.                          (ld_type_i==6'b10_0000)&&(addr_ram_i==2'b10)?
   {{24{data_sram_rdata[23]}},data_sram_rdata[23:16]}:
4.                          (ld_type_i==6'b10_0000)&&(addr_ram_i==2'b11)?
   {{24{data_sram_rdata[31]}},data_sram_rdata[31:24]}:
5.                          (ld_type_i==6'b10_0100)&&(addr_ram_i==2'b00)?
   {24'b0,data_sram_rdata[7:0]}:
6.                          (ld_type_i==6'b10_0100)&&(addr_ram_i==2'b01)?
   {24'b0,data_sram_rdata[15:8]}:
7.                          (ld_type_i==6'b10_0100)&&(addr_ram_i==2'b10)?
   {24'b0,data_sram_rdata[23:16]}:
8.                          (ld_type_i==6'b10_0100)&&(addr_ram_i==2'b11)?
   {24'b0,data_sram_rdata[31:24]}:
9.                          (ld_type_i==6'b10_0001)&&(addr_ram_i==2'b00)?
   {{16{data_sram_rdata[15]}},data_sram_rdata[15:0]}:
10.                         (ld_type_i==6'b10_0001)&&(addr_ram_i==2'b10)?
   {{16{data_sram_rdata[31]}},data_sram_rdata[31:16]}:
11.                         (ld_type_i==6'b10_0101)&&(addr_ram_i==2'b00)?
   {16'b0,data_sram_rdata[15:0]}:
12.                         (ld_type_i==6'b10_0101)&&(addr_ram_i==2'b10)?
   {16'b0,data_sram_rdata[31:16]}:
13.                         (ld_type_i==6'b10_0011)?data_sram_rdata[31:0]
   :32'b0;
```

ld_type_i 为 EX 段传递而来的指令的操作码，addr_ram 为 EX 段传递而来的

访存地址的低 2 位，data_sram_rdata 为主存提供的数据。通过选择语句结合 ld_type_i 和 addr_ram 可以确定访存结果的长度，进而从 data_sram_rdata 中取出正确的结果。最后通过 sel_rf_res 判断传递给 WB 段的结果来源于访存结果还是 EX 段的计算结果。

4.5 回写（WB）阶段

4.5.1 WB 段结构图

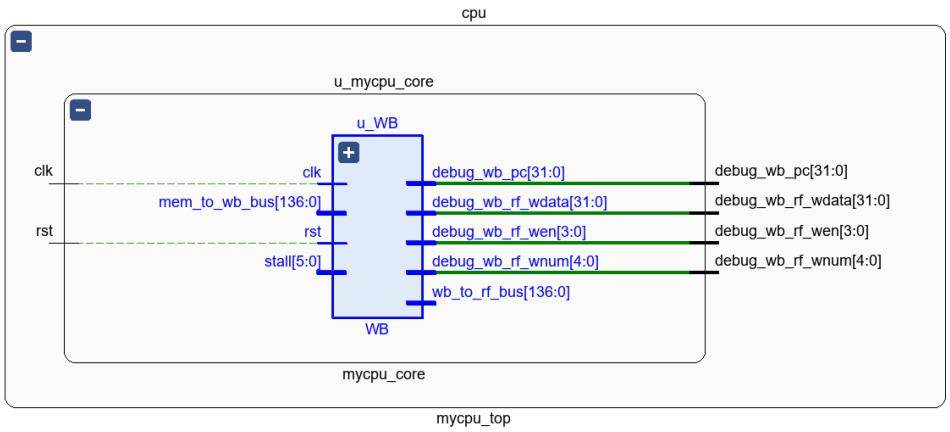

图 7

4.5.2 WB 段整体功能说明

- 接受来 MEM 段传递的信号。
- 回传信号给 ID 段，解决数据相关，并写入寄存器。

4.5.3 WB 段端口描述

序号	接口名	宽度(bit)	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	暂停流水线信号
4	mem_to_wb_bus	137	输入	MEM 段传递的信号
5	wb_to_rf_bus	137	输出	回写到 ID 段的信号
6	debug_wb_pc	32	输出	程序输出 PC 值
7	debug_wb_rf_wdata	32	输出	程序输出的回写数据
8	debug_wb_rf_wen	4	输出	程序输出的写使能

9	debug_wb_rf_wnum	5	输出	程序输出的回写地址

表 9

4.5.4 WB 段信号描述

序号	接口名	宽度(bit)	类型	作用
1	rf_we	1	wire	回写寄存器使能信号
2	rf_waddr	5	wire	回写寄存器的地址
3	rf_wdata	32	wire	回写寄存器的数据
4	wb_pc	32	wire	WB 段指令的 PC

表 10

由于 WB 段只是将 MEM 段传递的数据回写到寄存器中，因此该段不需要另外添加功能，所以省略了对 WB 段功能模块的描述。

4.6 寄存器模块与 CRTL 模块介绍

寄存器模块结构图

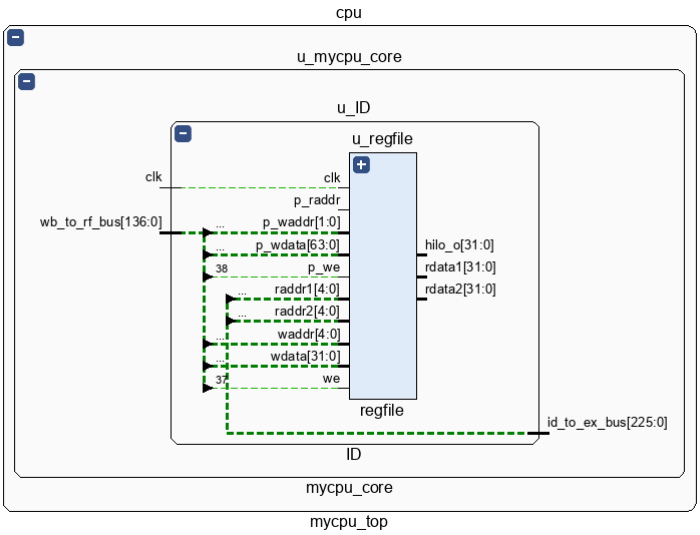

图 8

端口介绍

序号	接口名	宽度(bit)	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	raddr1	5	输入	要读的寄存器地址 1
4	rdata1	32	输出	raddr1 中数据
5	raddr2	6	输入	要读的寄存器地址 2
6	rdata2	32	输出	raddr2 中数据

7	we	1	输入	写使能信号
8	waddr	5	输入	写地址
9	wdata	32	输入	写入的数据
10	p_raddr	1	输入	要读 HILO 寄存器地址
11	hilo_o	32	输出	p_raddr 的数据
12	p_we	1	输入	HILO 写使能信号
13	p_waddr	2	输入	HILO 写地址
14	p_wdata	64	输入	HILO 要写入的数据

表 11

寄存器工作原理

普通寄存器与 HILO 寄存器相同都是根据地址从中读取数据，根据使能信号和地址将数据写入寄存器

CTRL 模块结构图

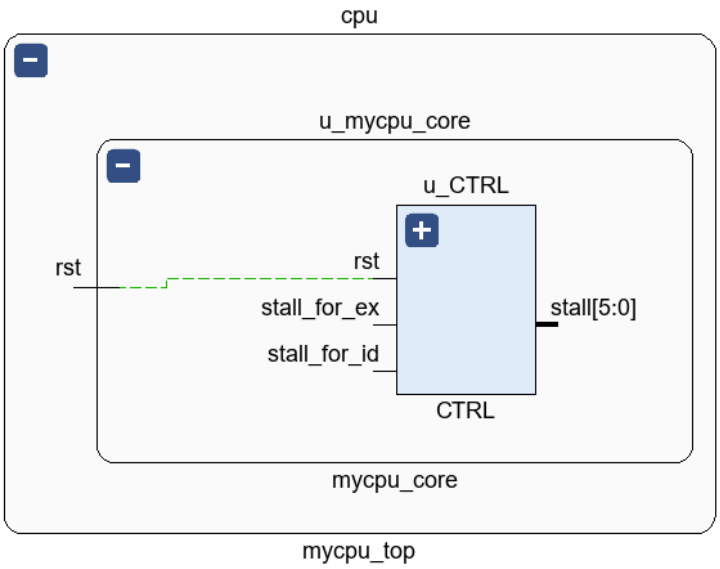

图 9

CTRL 模块端口介绍

序号	接口名	宽度(bit)	输入\输出	作用
1	rst	1	输入	复位信号
2	stall_for_id	1	输入	ID 段暂停信号
3	stall_for_ex	1	输入	EX 段暂停信号
4	stall	6	输出	暂停信号

CTRL 模块工作原理

当 stall_for_id 为 1 时，stall=000111,表示 PC 自增、取指、译码暂停其余不受影响；当 stall_for_ex 为 1 时，stall=001111 表示 PC 自增、取指、译码和执行暂停，其余不受影响。

五、自制 32 周期移位乘法器

5.1 乘法器结构图

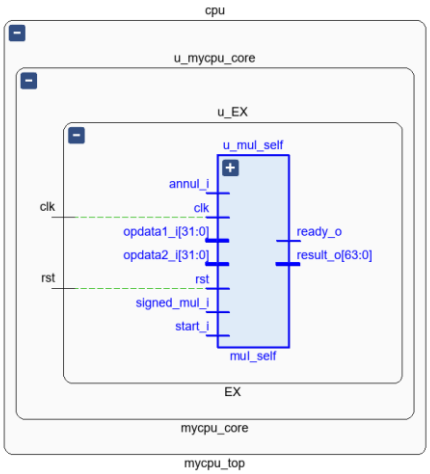

图 10

5.2 乘法器原理图

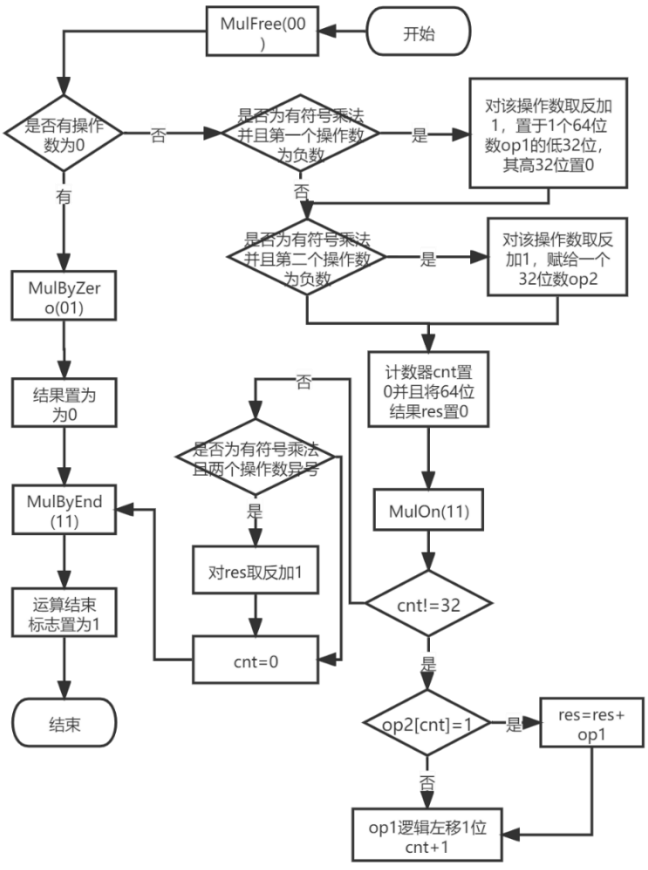

图 11

5.3 乘法器端口描述

序号	接口名	宽度(bit)	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	annul_i	1	输入	是否取消乘法运算
4	opdata1_i	32	输入	乘数 1
5	opdata2_i	32	输入	乘数 2
6	signed_mul_i	1	输入	是否为有符号乘法
7	start_i	1	输入	是否开始乘法运算
8	ready_o	1	输出	乘法运算是否结束
9	result_o	64	输出	乘法运算结果

表 11

5.4 使用方法

需声明的变量：

序号	变量	类型	长度	作用
1	mul_result	wire	64	乘法结果
2	inst_mult	wire	1	有符号乘
3	inst_multu	wire	1	无符号乘
4	mul_ready_i	wire	1	乘法运算结束
5	stallreq_for_mul	reg	1	乘法暂停信号
6	mul_opdata1_o	reg	32	乘数 1
7	mul_opdata2_o	reg	32	乘数 2
8	mul_start_o	reg	1	乘法是否开始
9	signed_mul_o	reg	1	是否为有符号乘法

表 11

乘法器模块调用方法：

调用分为两部分：完成模块的接口和加上组合逻辑控制语句。两部分都需要放在 EX.v 文件中。

第一部分：模块接口

```
1.  mul_self u_mul_self(
2.          .rst            (rst                ),
3.          .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       ),
        .ready_o      (mul_ready_i      )
    );
```

第二部分：组合逻辑控制语句

```
always @ (*) begin
        if (rst) begin
            stallreq_for_mul = `NoStop;
            mul_opdata1_o = `ZeroWord;
            mul_opdata2_o = `ZeroWord;
            mul_start_o = `MulStop;
            signed_mul_o = 1'b0;
        end
        else begin
            stallreq_for_mul = `NoStop;
            mul_opdata1_o = `ZeroWord;
            mul_opdata2_o = `ZeroWord;
            mul_start_o = `MulStop;
            signed_mul_o = 1'b0;
            case ({inst_mult,inst_multu})
                2'b10:begin
                    if (mul_ready_i == `MulResultNotReady) begin
                        mul_opdata1_o = alu_src1;
                        mul_opdata2_o = alu_src2;
                        mul_start_o = `MulStart;
                        signed_mul_o = 1'b1;
                        stallreq_for_mul = `Stop;
                    end
                    else if (mul_ready_i == `MulResultReady) begin
                        mul_opdata1_o = alu_src1;
                        mul_opdata2_o = alu_src2;
                        mul_start_o = `MulStop;
                        signed_mul_o = 1'b1;
                        stallreq_for_mul = `NoStop;
                    end
                    else begin
                        mul_opdata1_o = `ZeroWord;
                        mul_opdata2_o = `ZeroWord;
                        mul_start_o = `MulStop;
                        signed_mul_o = 1'b0;
```

```
                        stallreq_for_mul = `NoStop;
                    end
                end
                2'b01:begin
                    if (mul_ready_i == `MulResultNotReady) begin
                        mul_opdata1_o = alu_src1;
                        mul_opdata2_o = alu_src2;
                        mul_start_o = `MulStart;
                        signed_mul_o = 1'b0;
                        stallreq_for_mul = `Stop;
                    end
                    else if (mul_ready_i == `MulResultReady) begin
                        mul_opdata1_o = alu_src1;
                        mul_opdata2_o = alu_src2;
                        mul_start_o = `MulStop;
                        signed_mul_o = 1'b0;
                        stallreq_for_mul = `NoStop;
                    end
                    else begin
                        mul_opdata1_o = `ZeroWord;
                        mul_opdata2_o = `ZeroWord;
                        mul_start_o = `MulStop;
                        signed_mul_o = 1'b0;
                        stallreq_for_mul = `NoStop;
                    end
                end
                default:begin
                end
            endcase
        end
    end
```

六、完成指令

运算指令	ADD	ADDI	ADDU	ADDIU
	SUB	SUBU	SLT	SLTI
	SLTU	SLTIU	DIV	DIVU
	MULT	MULTU		
逻辑运算指令	AND	ANDI	LUI	NOR
	OR	ORI	XOR	XORI
移位指令	SLL	SLLV	SRA	SRAV

	SRL	SRLV		
跳转指令	BEQ	BNE	BGEZ	BGTZ
	BLEZ	BLTZ	BLTZAL	BGEZAL
	J	JAL	JR	JALR
数据移动指令	MFHI	MFLO	MTHI	MTLO
访存指令	LB	LBU	LH	LHU
	LW	SB	SH	SW

表 12

七、实验心得及改进意见

w：在这次实验中，我通过实践深入了解了流水线的思想及其在 CPU 设计中的重要作用。建立数据旁路以解决数据相关问题加深了我对课堂上数据相关知识的理解，自制乘法器让我明白了在实际的 CPU 中乘法的运行原理。同时对计算机底层知识的学习也让我明白在往后的编程如何编写高效的代码。

1：这次实验使得我对计算机底层原理的认识更深刻，也明白了在平时编程时所编写的程序及所调用的库函数和框架的底层实现原理。同时实验也有助于我理论知识的学习，在实践中体会理论知识比单一看书获得知识更加有趣。

1：通过实践我解决了在课堂上学习理论知识时的一些困惑，理解了理论学习和工程实践时对 CPU 设计的不同要求和针对不同问题所采取的不同措施。同时在实践过程中，我学会了使用 GitHub 的各项功能，提升自己在科研方面的基础技能。

八、参考资料

[1]张晨曦等.《计算机体系结构（第二版）》.高等教育出版社.2005 年
[2]雷思磊.《自己动手做 CPU》.电子工业出版社.2014 年