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：张国标

组员：仝竞奇、周翰亭

报告日期：2021.12.18

目录

1. 工作量及总体设计：

- 张国标 – 负责自制乘除法器、乘除寄存器和内存读写部分 40%

  完成指令:sll、sw、lw、bne、sra、srav、srl、srlv、nor、mflo、mfhi

- 周翰亭 – 负责定向路径，非乘除法的运算指令和部分其他指令的添加 30%

  完成指令:bne、slt、slti、sltiu、sltu、xor、j、mtlo、mthi

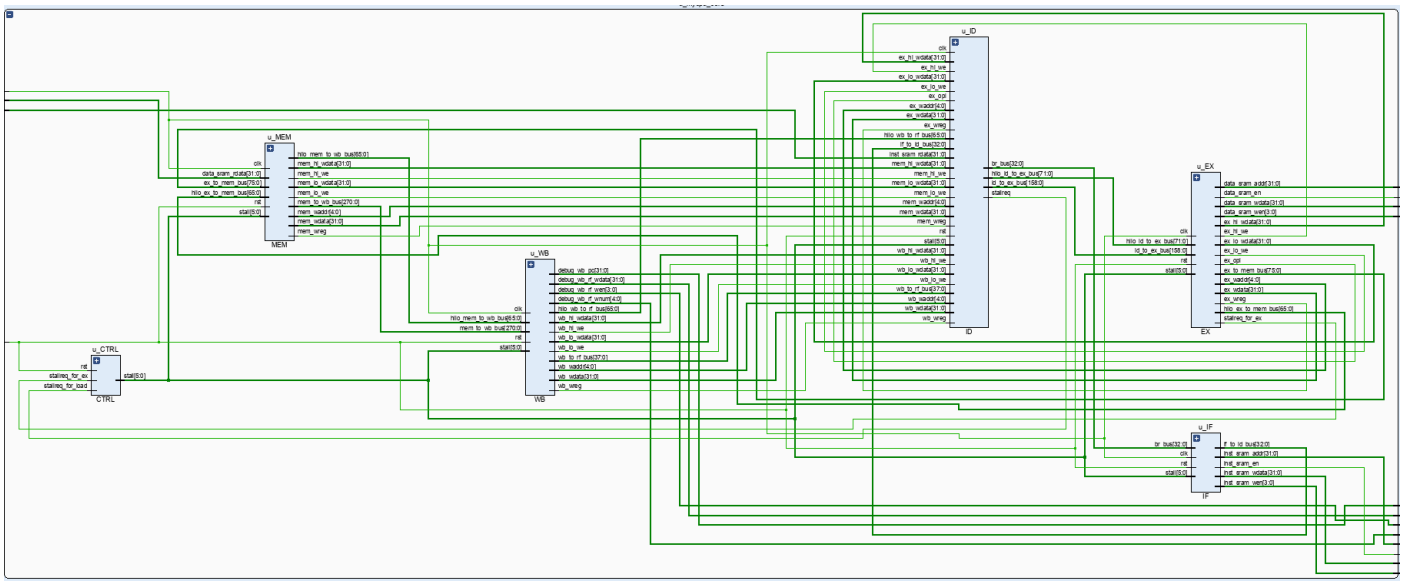
- 仝竟奇 – 分支指令、乘除法具体指令和load store指令的添加 30%

  完成指令:bgez、bgtz、blez、bltz、bltzal、bgezal、jalr、mult、multu、div、divu、lb、lbu、lh、lhu、sb、sh

  总体设计:

  总体采用五段流水的结构,IF段取值、ID段译码、EX段有效地址计算、MEM段访问内存、WB段写回寄存器。同时采用ctrl段进行流水线的暂停控制。

  不同流水段之间得连线图:



2. 环境及开发工具：

使用vivado进行功能仿真作为程序运行环境，通过Vscode编程，使用github远程协作。

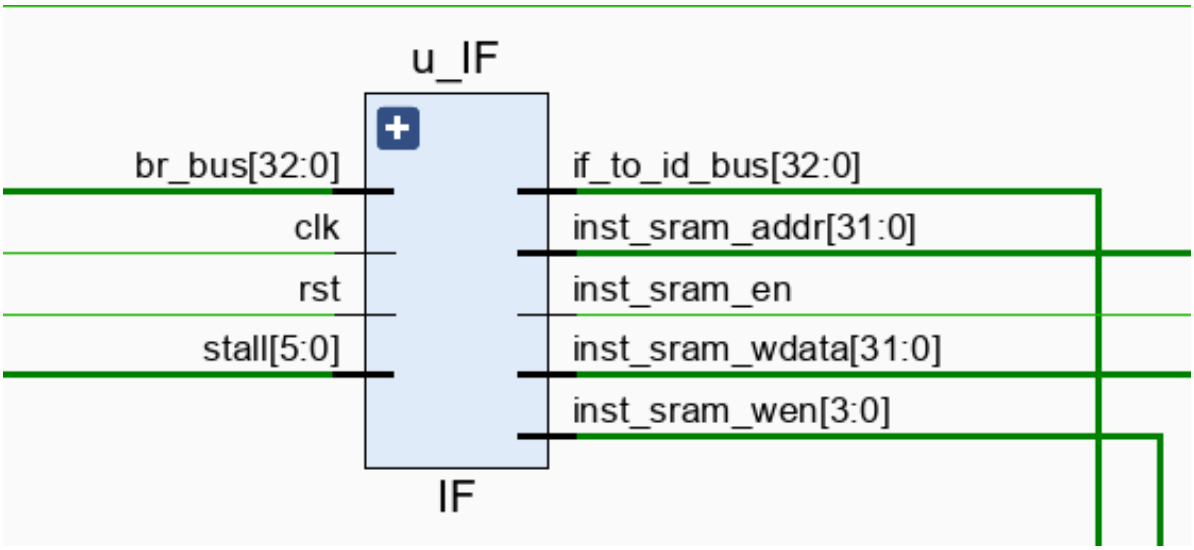
3. 流水段：

3.1. IF段：

3.1.1. 整体功能说明：

IF段主要进行取指操作，根据当前PC寄存器里的PC值从内存中取出指令，将指令传到ID段进行译码操作。

3.1.2. 端口信号介绍：



序号	接口名	宽度（bit）	输入/输出	作用
1	clk	1	输入	时钟信号
2	rst	1	输入	复位信号
3	stall	6	输入	流水线暂停信号，如果某位为1代表该段流水暂停
4	br_bus	33	输入	跳转总线
5	if_to_id_bus	33	输出	IF段传到ID段总线
6	inst_sram_en	1	输出	内存是否可访问（根据CE寄存器的值判断是否可取指令）
7	inst_sram_wen	3	输出	内存是否可写（IF段不写内存，为0）
8	inst_sram_addr	32	输出	内存访问地址（PC值）
9	inst_sram_wdata	32	输出	写内存的值（IF段不写内存，为0）

IF段接入一条总线br_bus，拆解为以下信号

序号	信号名	宽度（bit）	作用
1	br_e	1	是否跳转
2	br_addr	32	跳转到的地址(跳转后的pc值)

3.1.3. 功能模块说明

在每个时钟周期开始时，如果IF段不被暂停，那么PC寄存器将存入上一段的next_pc值，即本段的PC值，CE寄存器将赋为1表示可以取指。

本段的next_pc值将根据是否跳转来决定，跳转则为跳转到的地址，否则为PC+4。

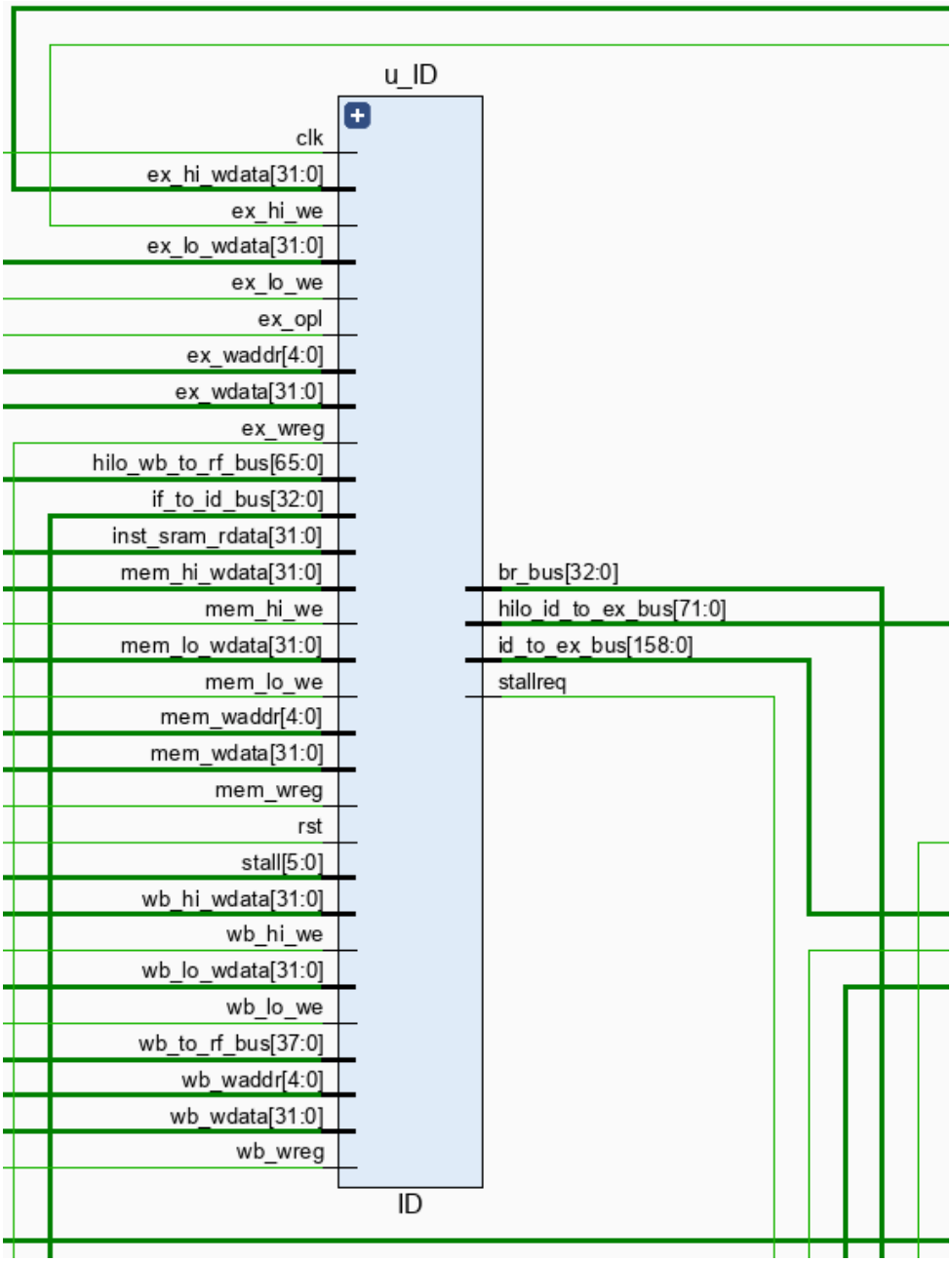
```
assign next_pc = br_e ? br_addr
                      : pc_reg + 32'h4;
```

inst_sram_en赋ce_reg的值表示是否可访问内存取指，inst_sram_addr赋pc_reg的值表示取指地址，注意当IF段暂停时pc_reg内已经是下一条指令的pc值，使用inst_sram_addr在内存中取出的指令会直接送到ID段，所以inst_sram_addr应为ID段的指令值，即为pc_reg内地址的上一条指令。所以在IF段暂停时，应该将pc_reg-4赋给inst_sram_addr。

```
assign inst_sram_en = ce_reg; assign
        inst_sram_wen = 4'b0;
        assign inst_sram_addr = (stall[0] == NoStop) ? pc_reg : pc_reg - 32'h4; assign inst_sram_wdata = 32'b0;
        assign if_to_id_bus = {ce_reg,
                pc_reg
        };
```

3.2. ID段：

3.2.1. 端口信号介绍：



序号	接口名	输入I/输出O	宽度	作用
1	clk	I	1	时钟信号
2	rst	I	1	复位信号
3	stall	I	6	输入的延迟信号
4	stallreq	O	1	输出的延迟信号
5	if_to_id_bus	I	33	IF->ID总线
6	inst_sram_rdata	I	32	传入的指令机器码
7	wb_to_rf_bus	I	38	WB->寄存器总线
8	hilo_wb_to_rf_bus	I	66	HILO寄存器的WB线
9	id_to_ex_bus	O	159	ID->EX总线
10	br_bus	O	33	地址线
11	hilo_id_to_ex_bus	O	72	HILO寄存器的ID->EX线
12	ex_wreg	I	1	EX段是否有传入寄存器
13	ex_waddr	I	5	EX段传入寄存器地址
14	ex_wdata	I	32	EX段新的寄存器数值
15	ex_opl	I	1	EX段是否为load指令
16	ex_hi_we	I	1	EX段是否传入HI寄存器
17	ex_lo_we	I	1	EX段是否传入LO寄存器
18	ex_hi_wdata	I	32	EX段HI寄存器的值
19	ex_lo_wdata	I	32	EX段LO寄存器的值
20	mem_wreg	I	1	MEM段是否有传入寄存器
21	mem_waddr	I	5	MEM段传入寄存器地址
22	mem_wdata	I	32	MEM段新的寄存器数值
23	mem_hi_we	I	1	MEM段是否传入HI寄存器
24	mem_lo_we	I	1	MEM段是否传入LO寄存器
25	mem_hi_wdata	I	32	MEM段HI寄存器的值
26	mem_lo_wdata	I	32	MEM段LO寄存器的值
27	wb_wreg	I	1	WB段是否有传入寄存器
28	wb_waddr	I	5	WB段传入寄存器地址
29	wb_wdata	I	32	WB段新的寄存器数值
30	wb_hi_we	I	1	WB段是否传入HI寄存器
31	WB_LO_WE	I	1	WB段是否传入LO寄存器
32	WB_HI_WDATA	I	32	WB段HI寄存器的值
33	wb_lo_wdata	I	32	WB段LO寄存器的值

3.2.2. 功能模块说明：

运行机制：

ID段接入四条总线：inst_sram_rdata、wb_to_rf_bus、if_to_id_bus和 hilo_wb_to_rf_bus， 输出三条：id_to_ex_bus、br_bus和 hilo_id_to_ex_bus。

1. inst_sram_rdata是按时序逻辑依次读入的指令机器码。inst_sram_rdata会被按位拆解为数个不同部分，如最高六位为 op_code，经译码器解析后归类为一每个指令独有的导线单位， 开头为 inst_：

```
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_bne          =op_d[6'b00_0101];
```

有些指令还需要最低六位 func经过译码器解析后辅助判定其种类：

```
assign inst_sra              =op_d[6'b00_0000] &func_d[6'b00_0011];
assign  inst_srav  =  op_d[6'b00_0000]  &  func_d[6'b00_0111];  assign  inst_srl  =
op_d[6'b00_0000]  & func_d[6'b00_0010]; assign inst_srlv      = op_d[6'b00_0000]  &
func_d[6'b00_0110];
assign inst_nor    =op_d[6'b00_0000] &func_d[6'b10_0111];
```

然后接下来一切大多都与其属于何种指令（哪个 inst_导通）有着密不可分的关系。

比如，确定EX段alu要执行哪个装置。

```
assign op_add=inst_addiu|inst_jal|inst_lw|inst_addu|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_sltiu|inst_sltu; 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};
```

再确定两个操作数，由 SEL_ALU_SRC1和 SEL_ALU_SRC2判断。这两个导线组采用one-hot式编码：SEL_ALU_SRC1由低到高三位分别代表第一个操作数取为 rs段号的寄存器，取pc值和取 sa段数值的无符号拓展；SEL_ALU_SRC2则有四位，代表第二操作数的取法：由高到低四位分别是取 rt段操作数的值，取立即数区的符号拓展，取正整数8和取立即数区的无符号扩展。相似的，还有 sel_rf_dst这一负责指示结果储存寄存器的导线组，从低到高分别表示 rd段号寄存器，rt段号寄存器以及31号寄存器，然后将运算出的内容写入 rf_waddr中：

```
// 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;
```

另外在读写方面，inst_X还决定了 data_ram_en和data_ram_wen，两者分别是「能否读写内存」和「读写内存使能信号」，负责寄存器和内存的数据交换管理：

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

// write enable bianhao
assign data_ram_wen = inst_sw ? 4'b1111 :
                      inst_lw ? 4'b0000 : inst_lb ? 4'b1110 :
                      inst_lbu ? 4'b1101 : inst_lh ? 4'b1100 :
                      inst_lhu ? 4'b0110 : inst_sb ? 4'b0001 :
                      inst_sh ? 4'b0011 : 4'b0;
```

2. 而由上可知，作为操作数的寄存器只可能取 rs段和 rt段，所以这里采用每次运行都调用一次 regfile读出 rs段和 rt段寄存器的内容，无论其是否被用到，读出的内容被放在 RDATA1和RDATA2中。同时，这个 regfile模块也负责读入 hi和 lo这两个特殊寄存器的内容存入hi/lo_rdata，并将从MEM、EX和WB三段传来的数组信号调入 regfile，从而实现局部回路功能。

   ex、mem、wb打头的就是用于局部回路的组线。其中，结尾为 wreg和 we的提示是否存在回路的回调数据，结尾为 waddr的是传回寄存器的号数，而 wdata结尾的则是传回的数据本 身。如果 waddr正好和要调用的 rs或 rt相同，同时 wreg处于1状态（有回调数据），则用 wdata取代自寄存器组件中读出的数据。对于 hi和 lo两个寄存器同理，只不过起到存在性作用的换成 hi/lo_we结尾的导线，而传回的数据则是 hi/lo_wdata结尾的导线组。

3. sel_rf_res段标记EX段alu的计算结果是要存到目标的直接数据还是某个要读取的内存地址。rf_we代表指令是否需要读写普通寄存器。

lo/hi_e代表指令是否读 hi和 lo两个寄存器。lo/hi_rf_we

代表指令是否会写入 hi或 lo两个寄存器。

以上1.2.3.这些信号涉及 hi和 lo的会被并入 hilo_id_to_ex_bus总线，其余的均会被并入 id_to_ex_bus总线，两者均传到EX段。

4. if_to_id_bus是由IF段传入的总线，携带有当前的 pc值信息，供ID段某些操作调用并传导到EX段。

5. wb_to_rf_bus和hilo_wb_to_rf_bus是两条从上一个WB段传来的总线，这两条总线做如下拆分：

```
assign {
    wb_rf_we, wb_rf_waddr,
    wb_rf_wdata
} = wb_to_rf_bus;

assign {
    wb_hi_rf_wdata,
    wb_lo_rf_wdata,
    wb_rf_hi_we, wb_rf_lo_we
} = hilo_wb_to_rf_bus;
```

然后均接入 regfile模组，用于进行 regfile模组中各寄存器内容的更新，实现写寄存器操作。其中，和局部回路部分类似，we结尾的代表对应的单元是否有变动（ wb_rf_we代表标准寄存器，wb_rf_hi/lo_we代表 hi和 lo寄存器），而 wb_rf_waddr代表要写入的地址，wdata结尾的则是要写入各对应寄存器的具体数值。

6. br_bus线传到下一个指令的IF段，代表是否发生跳转。bf_bus的构造为：

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

其中，br_e为是否进行跳转，br_addr为跳转到的目标指令地址。

br_e的判断由多种要素共同决定。首先，根据指令种类（ inst_X），给出几个作为判定容器的导线：

```
assign rs_eq_rt = (rdata1 == rdata2);
assign rs_geq_ze = (rdata1[31] == 1'b0); //rs>=0
assign rs_gt_ze = (rdata1[31] == 1'b0 && rdata1 != 0); //rs>0 assign rs_leq_ze = (rdata1[31] == 1'b1 ||
rdata1 == 0); // rs<=0 assign rs_lt_ze = (rdata1[31] == 1'b1); //rs<0
```

这几个是传输判断 RDATA1和 RDATA2是否满足某种条件的导线。它们与对应的指令导线求与即是 br_e的值：

```
    assign br_e = (inst_beq & rs_eq_rt) |(inst_bne & ~rs_eq_rt) |(inst_bgez & rs_geq_ze) |inst_jal
                  |inst_jr |inst_j |(inst_bgtz & rs_gt_ze) |(inst_blez & rs_leq_ze) |(inst_bltz & rs_lt_ze)
                  |(inst_bltzal & rs_lt_ze) |(inst_bgezal & rs_geq_ze) |
inst_jalr;
```

可以注意到，有一些指令是不需要这类 rs导线限制的，因为它们是无条件跳转指令。还有一些如 inst_bne，其实现是由某个的 rs类导线求反得到的。

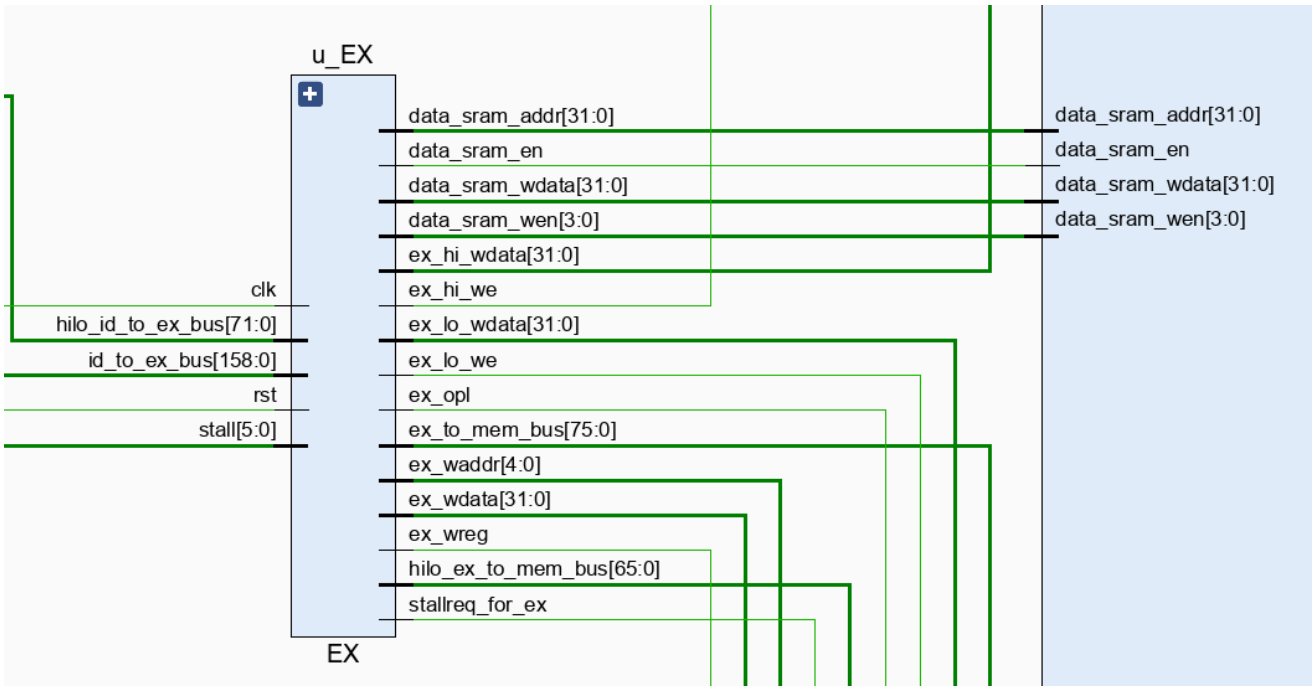
而 br_addr为跳转的目标地址，其实现第一取决于跳转指令，不同指令目标地址的给出格式不同。第二取决于操作数：第一部分为 pc_plus_4 = id_pc + 32'H4，第二个大多数情况下是指令的立即数部分的扩展。通过移位和拼接运算，可以简单的算出 br_addr。这样，就完整的实现了整个跳转机制。

3.3. EX段：

3.3.1. 整体功能说明：

EX段主要进行有效地址计算。包含的功能有，对算数运算指令计算出其结果；对store指令计算出存入的地址，将要存入内存的值和地址传给mycore；对于load指令计算出要取得内存地址传给mycore；对于乘除指令调用mul或div模块计算出结果传给hilo总线。当需要发出暂停时stallreq_for_ex置1并传给ctrl，同时将ex段的计算结果传给ID段构成数据通路，避免数据相关。

3.3.2. 端口信号介绍：



序号	接口名	宽度(bit)	输入/输出	作用
1	clk	1	输入	时钟信号
2	rst	1	输入	复位信号
3	stall	6	输入	流水线暂停信号
4	id_to_ex_bus	159	输入	自ID段传入EX的总线
5	hilo_id_to_ex_bus	72	输入	自ID段传入EX的乘除总线
6	ex_to_mem_bus	76	输出	EX段传到MEM段的总线
7	hilo_ex_to_mem_bus	66	输出	EX段传到MEM段的乘除总线
8	data_sram_en	1	输出	内存是否可访问
9	data_sram_wen	4	输出	内存写使能信号
10	data_sram_addr	32	输出	内存访问地址
11	data_sram_wdata	32	输出	写内存的值
12	ex_wreg	1	输出	EX段写使能信号
13	ex_waddr	5	输出	EX段写寄存器地址
14	ex_wdata	32	输出	EX段写寄存器值
15	ex_opl	1	输出	EX段指令是否是load指令
16	ex_hi_we	1	输出	EX段指令是否写hi寄存器
17	ex_lo_we	1	输出	EX段指令是否写lo寄存器
18	ex_hi_wdata	32	输出	EX段指令写lo寄存器值
19	ex_lo_wdata	32	输出	EX段指令写lo寄存器值
20	stallreq_for_ex	1	输出	EX段是否发出暂停信号

EX段接入两条总线id_to_ex_bus 和 hilo_id_to_ex_bus。

1、id_to_ex_bus在每个时钟上升沿在不暂停的情况下存入寄存器，并被被拆解为以下信号：

序号	信号名	宽度（bit）	作用
1	ex_pc	32	EX段PC值
2	inst	32	EX段指令码
3	alu_op	2	算数指令操作码
4	SEL_ALU_SRC1	3	判定操作数1选择信号
5	SEL_ALU_SRC2	4	判定操作数2选择信号
6	data_ram_en	1	内存是否可访问
7	data_ram_wen	4	只是load和store指令的编号
8	rf_we	1	寄存器是否可写
9	rf_waddr	5	写寄存器地址
10	sel_rf_res	1	选择写入寄存器的值（内存值或EX段计算值）
11	RF_RDATA1	32	rs寄存器的值
12	RF_RDATA2	32	rt寄存器的值

2、hilo_id_to_ex_bus在每个时钟上升沿在不暂停的情况下存入寄存器，并被被拆解为以下信号：

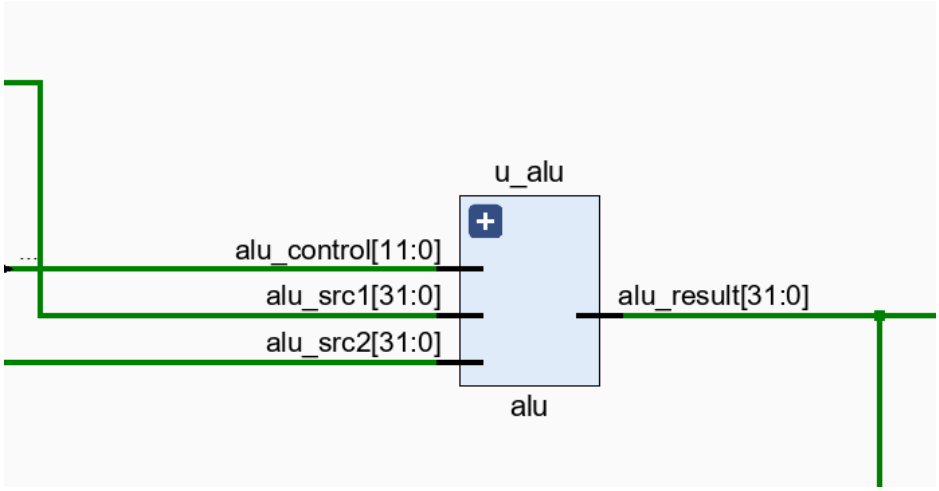
序号	信号名	宽度（bit）	作用
1	hilo_inst	4	hilo相关指令编号
2	hi_rdata	32	hi寄存器的值
3	lo_rdata	32	lo寄存器的值
4	hi_rf_we	1	hi寄存器是否可写
5	lo_rf_we	1	lo寄存器是否可写
6	hi_e	1	hi寄存器是否可读
7	lo_e	1	lo寄存器是否可读

3.3.3. 功能模块说明：

提前计算出立即数扩展、立即数零扩展和sa零扩展的值

```
wire [31:0] imm_sign_extend, imm_zero_extend, sa_zero_extend; assign imm_sign_extend =
{{16{inst[15]}},inst[15:0]};
assign imm_zero_extend ={16'b0, inst[15:0]}; assign sa_zero_extend =
{27'b0,inst[10:6]};
```

算数运算模块:



调用alu模块，通过alu_control的控制和算数操作数ALU_SRC1和ALU_SRC2计算出结果alu_result。

通过ID段传来的sel_alu_src选择对应的操作数。操作数1在pc值、sa0扩展和rs寄存器的值之间选择，操作数2在符号扩展立即数、8、零扩展立即数和rt寄存器的值中选择，按照对应位是否为1进行判断。

```
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 : // bgtzal... sel_alu_src2[3] ?
                  imm_zero_extend : rf_rdata2;
```

内存访问模块：

inst_X等值通过当前EX段的inst值可以得出,不在过多解释。根据store指令和写地址的后两位设置选择器，选择对应的写使能信号，1表示可写的字节。而从ID段传来的data_ram_wen并无实际用途，本次实验把他用作编号以得出inst_X的值。

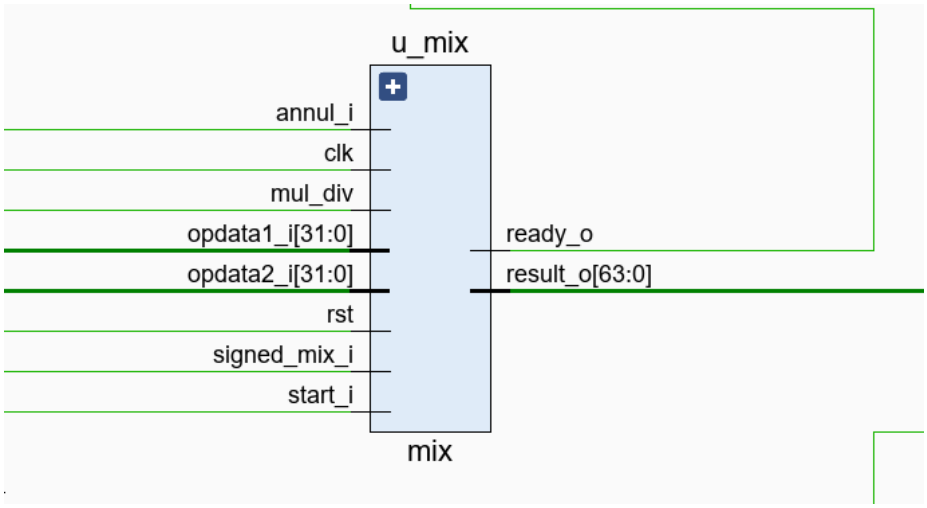
```
    assign data_sram_wen = (inst_sb & ex_result[1:0] == 2'b00) ? 4'b0001 :
                           (inst_sb & ex_result[1:0] == 2'b01) ? 4'b0010 : (inst_sb & ex_result[1:0]
                           == 2'b10) ? 4'b0100 : (inst_sb & ex_result[1:0] == 2'b11) ? 4'b1000 :
                           (inst_sh & ex_result[1:0] == 2'b00) ? 4'b0011 : (inst_sh & ex_result[1:0]
                           == 2'b10) ? 4'b1100 : inst_sw ? 4'b1111 : 4'b0; //memwen
```

根据写使能信号data_sram_wen，将对应字节的值写入data_sram_wdata，传入到mycore 模块。根据A03指令集的描述，都是传入最低字节或半字的值，所以sb指令中的赋给data_sram_wdata每个字节都是最低字节的值，sh指令内同理，每个半字都是最低半字的值。

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

乘除模块：

将操作数和是否有符号乘法（除法）作为输入调用乘除模块，操作数的选择同算数运算模块。



对于32周期的乘法和除法，需要使用时序逻辑在每个周期开始时传入操作数并且接受ready_o判断是否完成乘除操作，并且需要在ready_o为0即未完成运算时将stallreq_for_ex置为1暂停流水线。

```
    assign stallreq_for_ex =stallreq_for_mix ;
```

最后根据指令将结果赋给hi/lo_rf_wdata，传到MEM段，最后由WB段传给ID段写回到hilo 寄存器。

```
    assign hi_rf_wdata =inst_mthi ? rf_rdata1:
                                inst_mix |inst_mixu ?mix_result[63:32] : 32'b0; assign lo_rf_wdata =inst_mtlo ?
rf_rdata1:
                                inst_mix |inst_mixu ?mix_result[31:0] : 32'b0;
```

定向路径：

将寄存器是否可写和写寄存器的地址和值输出，传入ID段，在ID段判断是否需要用到该寄存器的值以解决RAW数据相关。

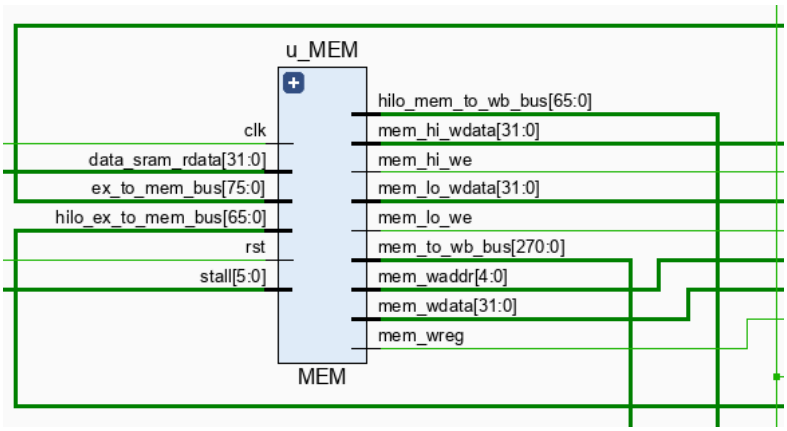
```
assign ex_wreg=rf_we; assign
ex_waddr=rf_waddr; assign
ex_wdata=ex_result; assign
ex_hi_we=hi_rf_we; assign
ex_lo_we=lo_rf_we;
assign ex_hi_wdata=hi_rf_wdata; assign
ex_lo_wdata=lo_rf_wdata;
```

3.4. MEM段

3.4.1. 整体功能说明：

MEM段主要功能就是对的内存访问，将load指令对应的内存地址所需的字节值从内存中取出出来传到WB段。需要定向路径解决数据相关。

3.4.2. 端口信号介绍：



序号	接口名	输入I/输出O	宽度	作用
1	clk	I	1	时钟信号
2	rst	I	1	复位信号
3	stall	I	6	延迟信号
4	ex_to_mem_bus	I	76	EX->MEM段总线
5	hilo_ex_to_mem_bus	I	66	HILO寄存器EX->MEM
6	data_sram_rdata	I	32	读取出的内存数据
7	mem_to_wb_bus	O	271	MEM->WB段总线
8	hilo_mem_to_wb_bus	O	66	HILO寄存器EX->WB
9	mem_wreg	O	1	MEM段局部回路开关
10	mem_waddr	O	5	MEM段传回寄存器号
11	mem_wdata	O	32	MEM段传回数据
12	mem_hi_we	O	1	MEM段是否传回HI
13	mem_lo_we	O	1	MEM段是否传回LO
14	mem_hi_wdata	O	32	MEM段传回HI值
15	mem_lo_wdata	O	32	MEM段传回LO值

3.4.3. 功能模块说明：

MEM段最主要的操作是读内存。首先，根据EX段总线 ex_to_mem_bus中从ID段传来的data_ram_wen判断原操作是哪种操作：

```
wire inst_lw, inst_lb, inst_lh, inst_lbu, inst_lhu; assign inst_lw       =
data_ram_wen == 4'b0000 ? 1 : 0; assign inst_lb     = data_ram_wen == 4'b1110 ?
1 : 0; assign inst_lh       = data_ram_wen == 4'b1100 ? 1 : 0; assign inst_lbu
                              = data_ram_wen == 4'b1101 ? 1 : 0; assign inst_lhu
                              = data_ram_wen == 4'b0110 ? 1 : 0;
```

在确认了操作种类后，可以解析得到读取内存情况。

```
    assign data_sram_wen2 = ((inst_lb|inst_lbu) &ex_result[1:0] == 2'b00) ? 4'b0001
:
                            ((inst_lb|inst_lbu) &ex_result[1:0] == 2'b01) ? 4'b0010
:
                            ((inst_lb|inst_lbu) &ex_result[1:0] == 2'b10) ? 4'b0100
:
                            ((inst_lb|inst_lbu) &ex_result[1:0] == 2'b11) ? 4'b1000
:
                            ((inst_lh|inst_lhu) &ex_result[1:0] == 2'b00) ? 4'b0011
:
                            ((inst_lh|inst_lhu) &ex_result[1:0] == 2'b10) ? 4'b1100
:0;
```

其中，ex_result为自 ex_to_mem_bus传来的EX段ALU运算结果，如指令为读内存操作，则此运行结果即为目标内存地址。由于读取机制一次读取四个字节，因此应通过其最后两位和指令类型判定取该字节组的某几个字节。（如果指令为 lw取一整个四个字节，则无需此步骤）作为结果，DATA_SRAM_WEN2代表要读取四个字节中的哪几个字节，按自高到低的顺序排列，为1的位读取，为0位的不读取。然后，根据预先读取到的 ex_result处内存片段 data_sram_rdata，按指令要求和DATA_SRAM_WEN2取结果：

```
    assign mem_result =         inst_lw? data_sram_rdata :
                                data_sram_wen2 == 4'b1000 ? {(inst_lbu ? 24'b0 :
{24{data_sram_rdata[31]}}), data_sram_rdata[31:24]} :
                                data_sram_wen2 == 4'b0100 ? {(inst_lbu ? 24'b0 :
{24{data_sram_rdata[23]}}), data_sram_rdata[23:16]} :
                                data_sram_wen2 == 4'b0010 ? {(inst_lbu ? 24'b0 :
{24{data_sram_rdata[15]}}), data_sram_rdata[15:8]} :
                                data_sram_wen2 == 4'b0001 ? {(inst_lbu ? 24'b0 :
{24{data_sram_rdata[7]}}), data_sram_rdata[7:0]} :
                                data_sram_wen2 == 4'b1100 ? {(inst_lhu ? 16'b0 :
{16{data_sram_rdata[31]}}), data_sram_rdata[31:16]} :
                                data_sram_wen2 == 4'b0011 ? {(inst_lhu ? 16'b0 :
{16{data_sram_rdata[15]}}), data_sram_rdata[15:0]} : 32'b0;
```

其中 lbu和 lhu代表读取出的数据片段做无符号扩展，lb和 lh代表做有符号扩展。

最后，还要通过 ex_to_mem_bus提供的导线 sel_rf_res判断该指令是否涉及读内存运算，如果涉及读内存，便将 mem_result作为最终的输出结果写入 rf_wdata，然后将 rf_wdata送入mem_to_wb_bus总线输出到WB段，再到下一ID段写入寄存器

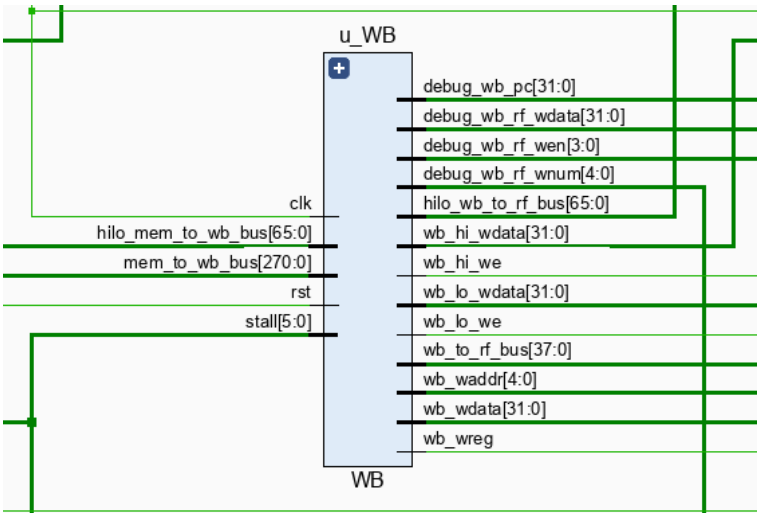
对于 hi和 lo两个特殊寄存器，其读写处理已经在EX被处理完毕，MEM段要做的只是传到WB段。

3.5. WB段

3.5.1. 整体功能介绍

WB段主要功能是写回，将写寄存器的值传到ID段，在ID段访问regfile写入寄存器，同样需要定向路径解决数据相关。

3.5.2. 端口信号介绍：



序号	接口名	输入I/输出O	宽度	作用
1	clk	I	1	时钟信号
2	rst	I	1	复位信号
3	stall	I	6	延迟信号
4	mem_to_wb_bus	I	271	MEM->WB段总线
5	hilo_mem_to_wb_bus	I	66	HILO寄存器MEM->WB
6	wb_to_rf_bus	O	38	WB->寄存器总线
7	hilo_wb_to_rf_bus	O	66	HILO寄存器的WB线
8	debug_wb_pc	O	32	调试程序读取pc的接口
9	debug_wb_rf_wen	O	4	调试程序读取是否写寄存器
10	debug_wb_rf_wnum	O	5	调试程序读取写寄存器号
11	debug_wb_rf_wdata	O	32	调试程序读取写寄存器值
12	WB_WREG	O	1	WB段局部回路开关
13	wb_waddr	O	5	WB段传回寄存器号
14	wb_wdata	O	32	WB段传回寄存器值
15	wb_hi_we	O	1	WB段是否传回HI
16	wb_lo_we	O	1	WB段是否传回LO
17	wb_hi_wdata	O	32	WB段传回HI的值
18	wb_lo_wdata	O	32	WB段传回LO的值

3.5.3. WB段工作流程：

WB段有两个内容：

1. 向平台的调试程序传送必要信号。

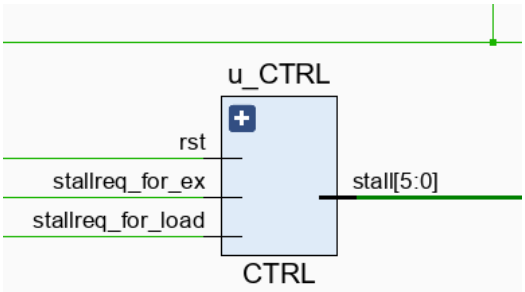
2. 向下一个ID段的regfile组件传入一条wb_to_rf_bus和一条hilo_wb_to_rf_bus，更新regfile组件的寄存器数据，起到写入寄存器的作用。

3.6. CTRL段

3.6.1. 整体功能说明：

CTRL段进行流水线的暂停控制，接受ID段和EX段发出的暂停请求信号，然后向各段流水发出相应的暂停信号，完成流水线的暂停控制。

3.6.2. 端口信号介绍：



序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号
2	stallreq_for_ex	1	输入	EX段是否发出暂停
3	stallreq_for_load	1	输入	ID段是否发出load相关的暂停
4	stall	6	输出	暂停信号

3.6.3. 功能模块说明：

在收到stallreq信号时发出相应暂停信号，5位二进制数，每一位代表着该流水段是否暂停。当收到的暂停请求为stallreq_for_load时，是load相关，则只暂停IF、ID段流水，之后流水段继续执行load指令，暂停一个周期就恢复。当收到的暂停请求为stallreq_for_ex时，为乘除运算引起的暂停，需要暂停整个流水线直到mul或div模块计算出结果为止，共32各周期。

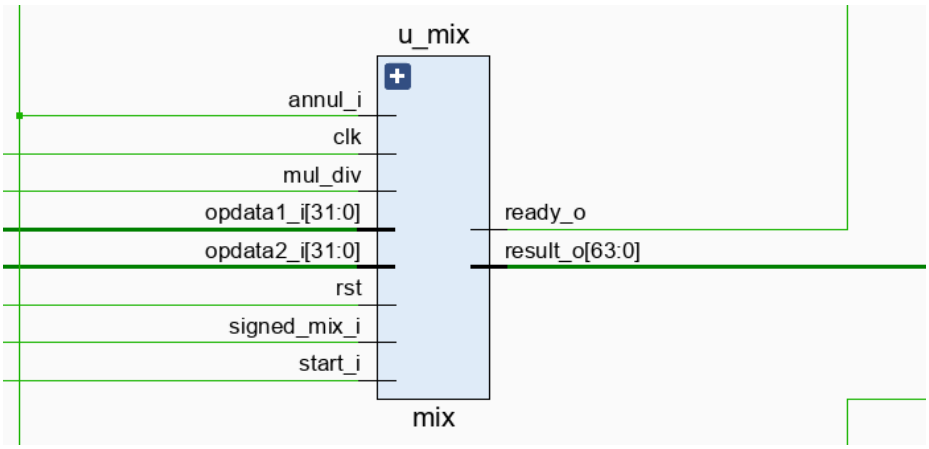
```
always @(*) begin if (rst)
        begin
                stall = StallBus'b0;
        end
        else if(stallreq_for_load==1) begin stall = 5'b00111;
        end
        else if(stallreq_for_ex==1) begin stall = 5'b11111;
        end
        else begin
                stall = StallBus'b0;
        end
end
```

3.7. 自制整合乘除法器mix说明

3.7.1. 整体功能说明：

跟据传入信号做有符号或无符号的32周期移位乘除法，在32周期后得出乘除法结果。

3.7.2. 端口信号介绍：



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	mul_div	1	输入	乘法还是除法
4	signed_mix_i	1	输入	是否有符号乘除法
5	OPDATA1_I	32	输入	操作数1
6	OPDATA2_I	32	输入	操作数2
7	start_i	1	输入	是否开始乘法运算
8	annul_i	1	输入	是否取消乘法运算，1位取消
9	result_o	64	输出	乘法运算结果
10	ready_o	1	输出	乘法运算是否结束

3.7.3. 功能模块说明：

时序控制仿照原来32位除法器实现。模拟除法过程自制移位除法器并将其与自制乘法器合并。模拟二进制除法过程，将32位操作数1（被除数）每次左移一位进temp（中间变量）中，将其与（操作数2）除数比较（无符号），若比除数大则商最高位置1，temp减去除数，若比除数小则商最高位置1，temp不进行任何操作，最后temp为余数；模拟二进制乘法过程，将32位操作数1左移保存到temp中（中间变量），每次根据操作数2对应位是否为1判断是否累加temp，最后得出结 果。

有符号除法先把操作数转为无符号整数，在对最后结果转为有符号整数。只在迭代过程中的移位操作中通过mul_div变量做选择,其他主要逻辑全部复用。

```
if(cnt != 6'b100000) begin if(mul_div == 1'b0) begin
            mix_res <= mix_res + (temp_op2[cnt] == 1'b1 ? temp_num : 0); temp_num <= (temp_num << 1);
      end else begin
            temp_num[31:0] = {mix_res[62:32],temp_op1[31- cnt]}; if(temp_num[31:0] >= temp_op2) begin
                  mix_res[63:32] <= temp_num[31:0]-   temp_op2; mix_res[31- cnt]
                  <= 1;
            end else begin
                  mix_res[63:32] <= temp_num[31:0]; mix_res[31-
                  cnt] <= 0;
            end
      end
      cnt <= cnt +1;                  //乘除法运算次数
end
```

3.8. 张国标实验感想：

本次实验给我带来的收获颇丰，通过本次实验我首先了解了vivado的使用和verilog语言的一些语法。其次通过自己添加乘除寄存器，以及定向路径，更深刻地了解了整个五级流水的工作过程和流水线相关问题的解决方案。对stall、定向路径、延迟槽等概念有了更加深刻的理解，对整个CPU的结构有了更好的认知。此外，通过对波形图的调试，与队友的分工合作，让我的工程能力也得到了极大的锻炼，学会了github等平台的使用。

3.9. 周翰亭实验感想：

这次实验通过调试结构的各种问题，有效的锻炼的我应对突发情况排除问题的能力，深化了我对课程内容的理解。同时，本次实验有效的提高了我的团队协作和沟通能力，考验了我的耐心和细心，让我获得了难以忘怀的回忆。

在这次实验中，我增强了我对git和github使用方法的掌握能力，同时也强化了我的表达能力和沟通能力，也强化了我对verilog和vivado的理解。

3.10. 仝竞奇实验感想：

这次实验首先让我感受到了理论和实践上存在的种种差异，很多课堂上的知识到工程上的转换非常具有挑战性，让我对于流水线的各种操作有了更加深刻的理解。添加指令的过程遇到了各种各样的问题，需要通过不断调试和阅读代码才能找到原因，通过本次实验增强了我解决实际问题的能力。

同时，团队合作让我更加感受到分工合作的重要性，好的分工可以让工程并行进行，从而提高效率。

4. 参考文献：

1. 《自己动手做CPU》，雷思磊
2. 《计算机体系结构》，张晨曦