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：

组长：

组员：

报告日期：2021.12.18

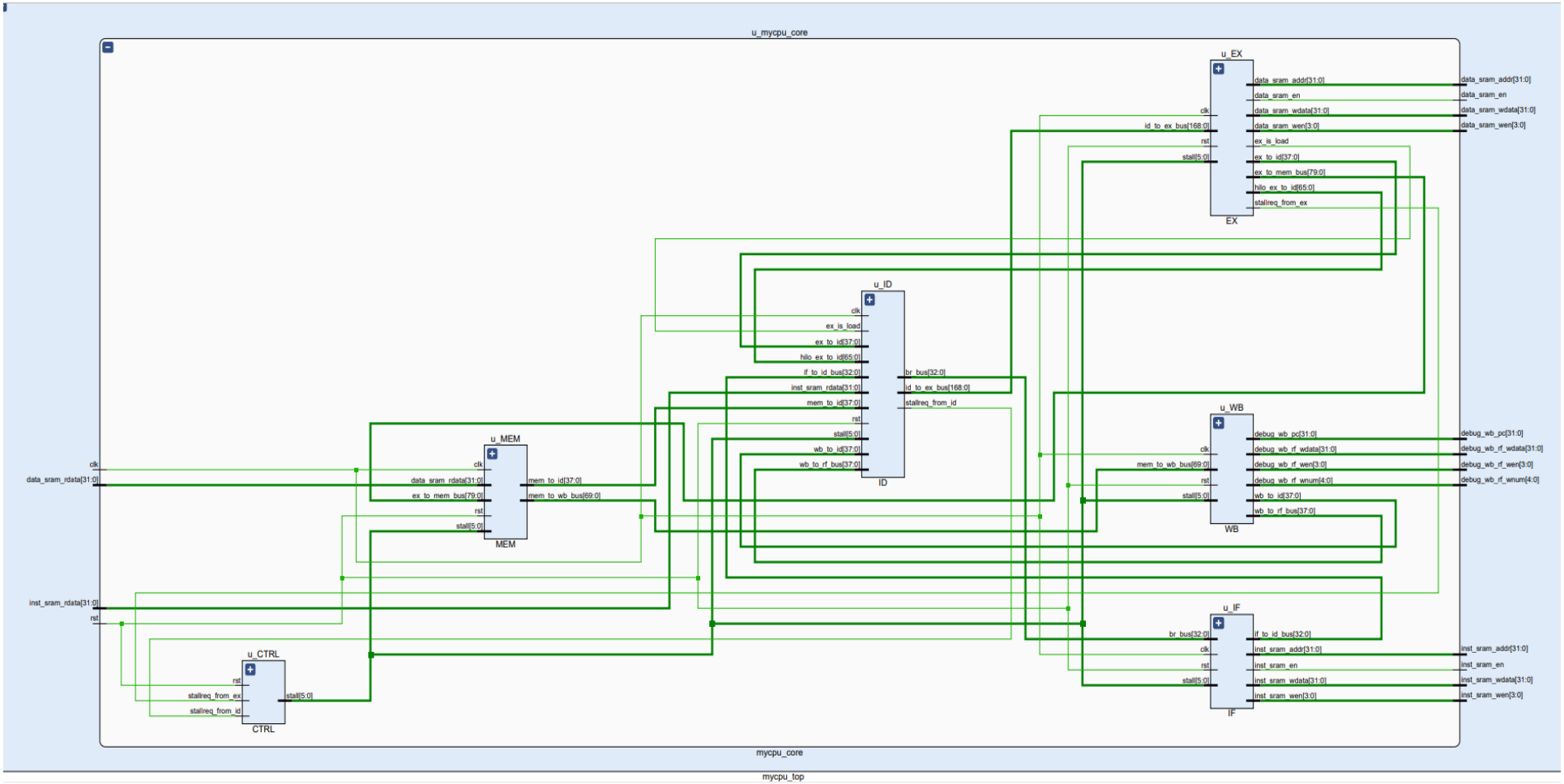
目录

一、1、小组成员工作量划分

姓名	完成任务点	总任务量占比
Szw	暂停的部分实现、乘除法器的连接、一部分指令的添加	50%
Zht	数据相关的解决、暂停的部分实现、一部分指令的添加、自制乘法器的实现	50%

二、总体设计

我们的流水线 CPU 设计主要由七个模块组成，分别为 IF 模块、ID 模块、EX 模块、MEM 模块、WB 模块、CTRL 模块，以及 regfile 模块。总体具有 32 个 32 位的整数寄存器，并且采用大端模式储存，具有 32bit 数据，地址总线宽度。我们的流水线 CPU 可以完成实验课程所要求的 64 个测试点，而且我们设计了自制的 32 周期的 32 位移位乘法器，并且已经通过了上板验证。整体的流水线 cpu 的模块连线图如图所示。


u_mycpu_core
u_EX
data_sram_addr[31:0]
data_sram_en
data_sram_wdata[31:0]
data_sram_wen[3:0]
clk
id_to_ex_bus[168:0]
rst
stall[5:0]
ex_is_load
ex_to_id[37:0]
ex_to_mem_bus[79:0]
hilo_ex_to_id[65:0]
stallreq_from_ex
EX
u_ID
clk
ex_is_load
ex_to_id[37:0]
hilo_ex_to_id[65:0]
if_to_id_bus[32:0]
inst_sram_rdata[31:0]
mem_to_id[37:0]
rst
stall[5:0]
wb_to_id[37:0]
wb_to_rf_bus[37:0]
br_bus[32:0]
id_to_ex_bus[168:0]
stallreq_from_id
ID
u_MEM
clk
data_sram_rdata[31:0]
ex_to_mem_bus[79:0]
rst
stall[5:0]
mem_to_id[37:0]
mem_to_wb_bus[69:0]
MEM
u_WB
clk
mem_to_wb_bus[69:0]
rst
stall[5:0]
debug_wb_pc[31:0]
debug_wb_rf_wdata[31:0]
debug_wb_rf_wen[3:0]
debug_wb_rf_wnum[4:0]
wb_to_id[37:0]
wb_to_rf_bus[37:0]
WB
u_IF
br_bus[32:0]
clk
rst
stall[5:0]
if_to_id_bus[32:0]
inst_sram_addr[31:0]
inst_sram_en
inst_sram_wdata[31:0]
inst_sram_wen[3:0]
IF
u_CTRL
rst
stallreq_from_ex
stallreq_from_id
stall[5:0]
CTRL
clk
data_sram_rdata[31:0]
inst_sram_rdata[31:0]
rst
mycpu_core
mycpu_top

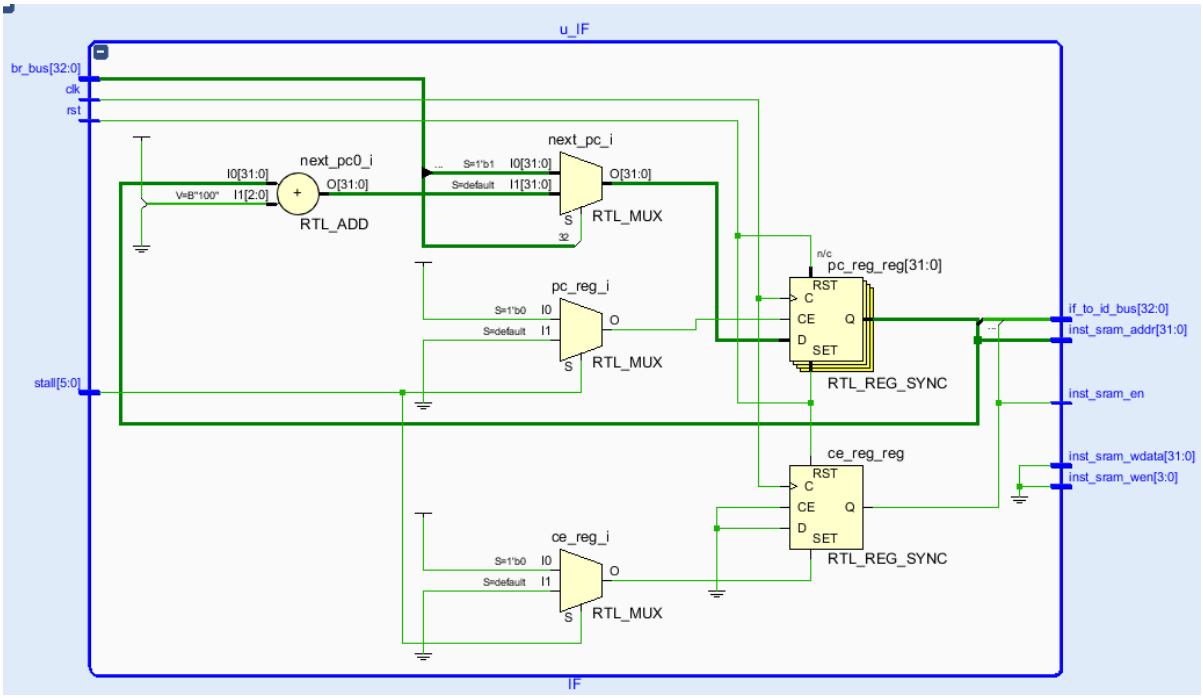
IF 段（取指令阶段）的功能为取指令，并控制指令延迟槽和跳转指令。同时确定下一条指令地址。ID 段（译码阶段）的功能为对指令进行译码、并调用 regfile 模块从通用寄存器读取要使用的通用寄存器的值、以及控制 hilo 寄存器的读取，并且将回写阶段传回的要写入的寄存器地址，将要写入寄存器的数据写入寄存器，如果指令中含有立即数，则设置采用立即数当作操作数的标识，以及立即数扩展方式的标识，并且选择在执行阶段要用到的 ALU，如果是转移指令，判断是否满足转移条件，如果满足转移条件，那么给出转移目标，作为新的指令地址。EX 段（执行阶段）的功能根据译码阶段得到的操作数，以及选择的 ALU 计算方式，计算出运算结果。如果是与 load 与 store 指令相似的指令，那么还会计算出相应的内存地址，并设置与内存数据交互的方式。MEM 段（访存阶段）的功能为判断即将要写回寄存器的值是 ALU 计算的结果还是从内存中读取的结果，并将结果传递到回写阶段。WB 段（回写阶段）的理论功能为为把运算结果储存到 cpu 的 32 个 32 位寄存器中。但是为了提高效率，将要写回寄存器的值以及要写回寄存器的地址发给译码阶段，在译码阶段读取寄存器的值的同时，将值写入对应的寄存器。CTRL 段的功能为控制流水线的暂停、清除等动作。

实验环境：采用龙芯杯指定 XILINX 公司的 Vivado2019.2 为 仿真运行的工具和 FPGA 综合工具，同时使用 VS Code 作为 Vivado2019.2 的默认代码编辑器，使用 git 完成小组内部的协同开发和代码同步。

三、流水线各个阶段的说明：

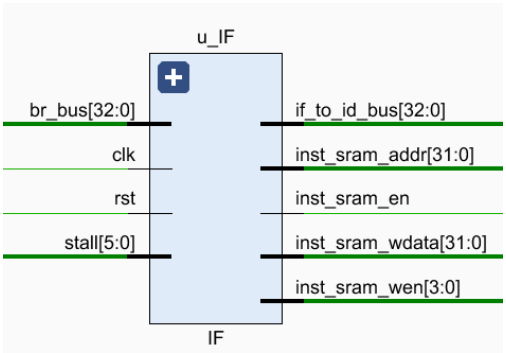
1、IF 模块

IF 段的整体设计图如图所示：



IF 模块功能：取指令，并控制指令延迟槽和跳转指令

IF 模块接口：



序号	接口名	宽度	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	控制暂停信号
4	br_bus	33	输入	ID 段发出的分支跳转指令的信号，控制指令延迟槽是否跳转
5	if_to_id_bus	33	输出	IF 段发给 ID 段的数据
6	inst_sram_en	1	输出	指令寄存器的读写使能信号
7	inst_sram_wen	4	输出	指令寄存器的写使能信

				号
8	inst_sram_addr	32	输出	指令寄存器的地址，用来寻找指令的存放的位置
9	inst_sram_wdata	32	输出	指令寄存器的数据，用来存放数据

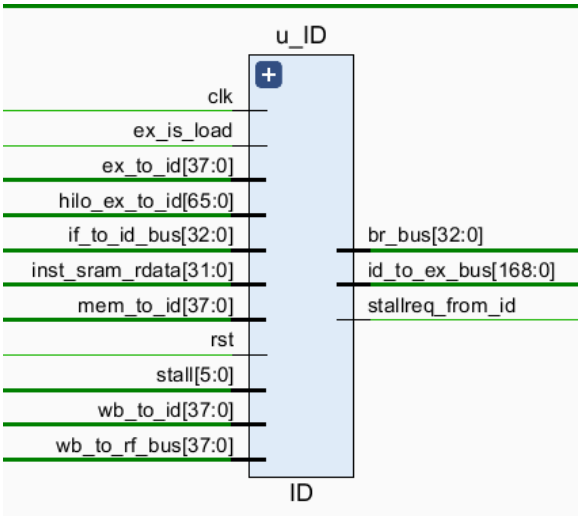
IF 模块细述：

IF 段输入时钟信号、复位信号，如果复位信号为真，就将 pc 的值置为复位的值。输入的 stall 暂停信号是由 CTRL 段发出的，其主要目的就是在流水线暂停时，IF 段通过识别 stall 来暂停指令延迟槽，使得下一条的 pc 值仍等于当前的 PC 值，使得 IF 段暂停一个时钟周期。br_bus 为 ID 段发出的跳转指令的信号，为是否跳转和要跳转的地址的值。在 IF 段若判断到需要跳转，则把 next_pc 值调成为需要跳转的地址值。在没有暂停和跳转发生的正常的情况下，reg_pc 值为当前的 next_pc 的值，同时 next_pc 值加上 4。最后将 reg_pc 的地址发给指令内存，从指令内存中得到相应的 pc 地址对应的值并发给 ID 段。

2、ID 模块

ID 模块功能：对指令进行译码，并将译码结果传给 EX 段，同时与寄存器进行交互，实现寄存器的读写，处理数据相关。

ID 模块接口：



序号	接口名	宽度	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	控制暂停信号
4	ex_is_load	1	输入	判断上一条指令是否是加载指令
5	stallreq	1	输出	控制暂停
6	if_to_id_bus	33	输入	IF 段发给 ID 段的数据
7	inst_sram_rdata	32	输入	当前指令地址中储存的值
8	wb_to_rf_bus	38	输入	WB 段传给 ID 段要写入寄存器的值
9	ex_to_id	38	输入	EX 段传给 ID 段的数据，用来判断数据相关
10	mem_to_id	38	输入	MEM 段传给 ID 段的数据，用来判断数据相关
11	wb_to_id	38	输入	WB 段传给 ID 段的数据，用来判断数据相关
12	hilo_ex_to_id	66	输入	EX 传给 ID 段要写入乘除法寄存器的值
13	id_to_ex_bus	169	输出	ID 段传给 EX 段的数据
14	br_bus	33	输出	ID 段传给 IF 段的数据，用来判断下一条指令的地址
15	stallreq_from_id	1	输出	从 ID 段发出的暂停信号

ID 模块细述：

ID 段分为几段：

第一段：**流水线是否暂停的判断和加气泡的实现。**

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

ID 段会收到来自 CTRL 模块的 stall 值，而 stall 的值就是用来控制流水线的暂停的。当 ID 段判断到 stall 的值的对应 ID 段的部分为`NoStop 时，即意味着没有流水线暂停，则在此部分将 IF 段传给 ID 段的 if_to_id_bus 正常地赋值给 if_to_id_bus_r，然后就可以进行接下来的正常的译码、取操作数的部分，流水线正常运行。但是如果判断到 stall 的值的对应的 ID 段的部分是`Stop，即此刻发生了访存冲突，现在需要读取的寄存器中的值还没有获得，还需要在下一个周期才可以从内存中读取出来，无法通过数据前递解决，则现在需要对流水线的 ID 段进行暂停一个周期，在下个周期获得到需要读取的值后再发给 ID 段。当判断到暂停后就将 if_to_id_bus_r 置为 0，本周期停止，下个周期再恢复正常。

```
always @(posedge clk) begin
    if (stall[1]==`Stop) begin
        q <= 1'b1;
    end
    else begin
        q <= 1'b0;
    end
end
assign inst = (q) ?inst: inst_sram_rdata;
```

但是还存在的另外一个问题就是 if_to_id_bus 是 IF 段发给 ID 段的内容，而这部分的内容是不包含指令值的，指令的值即 inst 值是在 ID 段时根据上个周期的 IF 段中的 pc 值从内存中读取到的，是直接从内存获取的。在暂停发生后 ID 段虽然对 if_to_id_bus 置为 0 但是 inst 值并没有被置为 0，因此还需要另外的一个操作就是需要判断暂停发生后把当前时刻的 inst 值保存一个周期，下一个周期再使用当前周期的 inst 值，只有这样才可以保证 ID 段和之后所有部分的指令的 pc 值和 inst 值是相互匹配的。

第二段：**正常指令的译码**：实现对指令的译码，依据指令中的特征字段区分指令，同时激活相应的指令对应的 inst_**变量，表示是哪一条指令。根据译码结果，读取通过 regfile 模块读取地址为 rs（inst[25:21]）以及地址为

rt(inst[20:16]的通用寄存器,得到 rdata1 以及 rdata2，并且通过判断是否发生数据相关，从而更改 rdata1 以及 rdata2 的值。同时分析要执行的运算，给对应的 ALU 标识符赋值，0 表示该条指令不采用该 ALU，1 表示该条指令采用该 ALU，同时将所有的 ALU 标识符组合起来成为 alu_op，alu_op 为十二位宽，代表 16 种不同的 ALU，并且作为传入 EX 段的一部分。要写入的目的寄存器。rf_we 代表写使能信号，表示该条指令是否用写入通用寄存器，sel_rf_dst[0]为一位宽，表示该指令要将计算结果写入 rd 通用寄存器，sel_rf_dst[1] 为一位宽， 表示该指令要将计算结果写入 rt 通用寄存器，sel_rf_dst[2] 为一位宽，表示该指令要将计算结果写入 31 号通用寄存器。rf_waddr 表示要该条指令的计算结果要写入的通用寄存器的地址，如果 sel_rf_dst[0]等于 1，那么 rf_waddr 将被赋值为 rd 寄存器的地址，如果 sel_rf_dst[1]等于 1，那么 rf_waddr 将被赋值为 rs 寄存器的地址，如果 sel_rf_dst[2]等于 1，那么 rf_waddr 将被赋值为 31 号寄存器的地址。data_ram_en 为一位宽,表示该条指令是否要与内存中取值或者写入值，如果该条指令要从内存中取值或者写入值，那么它将被赋值为 1’b1,data_ram_wen 为四位宽，表示该条指令是否要写入寄存器，如果该条指令要将计算结果的第几个字节写入寄存器，那么对应位置的值设为 1，就比如 sw 指令要将整个寄存器的值写入内存，那么 data_ram_wen 就设置为 4’b1111。data_ram_readen 为四位宽，表示该条指令是否要从寄存器中读取内容，并设置不同的读取方式。

跳转指令的译码：br_e 为一位宽，表示该条指令是否是跳转指令，如果该指令需要跳转，那么 br_e 被赋值为 1’b1，如果该指令不需要跳转，那么 br_e 被赋值为 1’b0。rs_ge_z 为一位宽，表示是否满足 rdata1 的值大于等于 0，如果 rdata1 的值大于等于 0，那么他被赋值为 1’b1,如果不满足，那么它将被赋值为 1’b0 ；rs_gt_z 为一位宽，表示是否满足 rdata1 的值大于 0，如果 rdata1 的值大于 0，那么他被赋值为 1’b1，如果 rdata1 的值不满足大于 0，那么他被赋值为 1’b0；rs_le_z 为一位宽，表示是否满足 rdata1 的值小于 0，如果 rdata1 的值小于 0，那么他被赋值为 1’b1，如果 rdata1 的值不满足小于 0，那么他被赋值为 1’b0；rs_lt_z 为一位宽，表示是否满足 rdata1 的值小于 0，如果 rdata1 的值小于 0，那么他被赋值为 1’b1，如果 rdata1 的值不满足小于 0，那么他被赋值为 1’b0；rs_eq_rt 为一位宽，表示是否满足 rdata1 是否等于 radta2 的值，如果 rdata1 是否等于 radta2 的值，那么他被赋值为 1’b1，如果 rdata1 的值不满足 rdata1 是否等于 radta2 的值，那么他被赋值为 1’b0。br_addr 为三十二位宽，表示跳转后的地址，如果是 beq 指令，就将该分支指令对应的延迟槽指令的 pc 加上立即数 offset 左移两位并进行有符号扩展的值赋值给 bre_addr。如果是 bne 指令，就将立即数 offset 左

移 2 位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的 PC 计算结果赋值给 bre_addr。如果是 bgez 指令，就将立即数 offset 左移两位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的 PC 计算得到的值赋值给 bre_addr。如果是 bgtz 指令，就将立即数 offset 左移两位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的 PC 计算得到的值赋值给 bre_addr。如果是 blez 指令，就将立即数 offset 左移两位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的 PC 计算得到的值赋值给 bre_addr。如果是 bltz 指令，就将立即数 offset 左移两位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的 PC 计算得到的值赋值给 bre_addr。如果是 bgezal 指令，就将立即数 offset 左移两位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的 PC 计算得到的值赋值给 bre_addr。如果是 bltzal 指令，就将立即数 offset 左移两位并进行有符号扩展的值加上该分支指令对应的延迟槽指令的 PC 计算得到的值赋值给 bre_addr。如果是 j 指令，就将该分支指令对应的延迟槽指令的 PC 的最高四位与立即数 inst_index(inst[26:0])左移两位后的值拼接之后赋值给 bre_addr。如果是 jal 指令，就将该分支指令对应的延迟槽指令的 PC 的最高四位与立即数 inst_index(inst[26:0])左移两位后的值拼接之后赋值给 bre_addr。

第三阶段：**设置操作数来源。**sel_alu_src1 为三位宽，表示对于第一个操作数有三种来源，第一种：第一个操作数的值为 rs 寄存器的值，第二种：当前的 PC 值，第三种：立即数零扩展。sel_alu_src2 为四位宽，表示对于第二个操作数有四种来源，第一种：第二个操作数的值为 rs 寄存器的值，第二种 rs 的值为立即数符号扩展，第三种，将第二个操作数复制为 32'b8，只有个别指令可以用的到，第四种，第二个操作数的值为立即数零扩展。

第四阶段：传值给其他阶段。传值给 ex 阶段，将

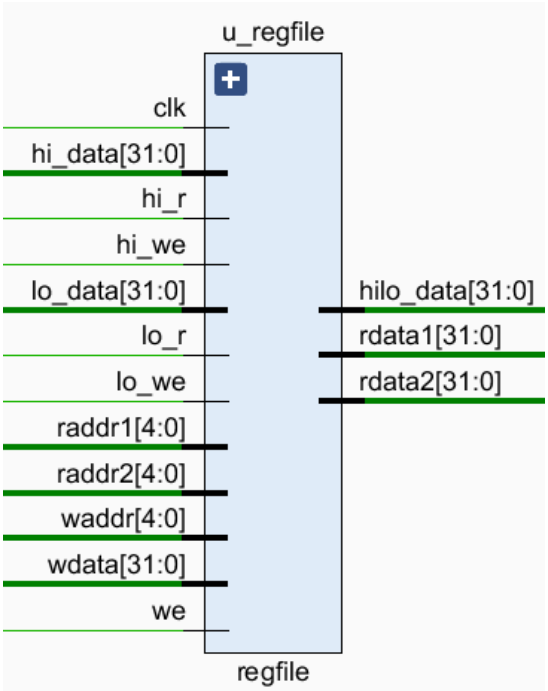
```
assign id_to_ex_bus = {
    data_ram_readen,//168:165
    inst_mthi,      //164
    inst_mtlo,      //163
    inst_multu,     //162
    inst_mult,      //161
    inst_divu,      //160
    inst_div,       //159
    id_pc,          // 158:127
    inst,           // 126:95
    alu_op,         // 94:83
    sel_alu_src1,   // 82:80
    sel_alu_src2,   // 79:76
    data_ram_en,    // 75
    data_ram_wen,   // 74:71
    rf_we,          // 70
    rf_waddr,       // 69:65
    sel_rf_res,     // 64
    rdata11,         // 63:32
    rdata22          // 31:0
};
```

其中 data_ram_readen 表示是否需要从内存中读取数据，inst_mthi,inst_mtlo,inst_multu, inst_divu，inst_div，inst_mult,分别表示是否是 mthi,mtlo,multu,mult,divu,div,指令，在 EX 会用到，判断是否调用乘法器以及除法器，alu_op 传到 EX 段，告诉 EX 段将要调用哪一个 ALU，sel_alu_src1 告诉 EX 段操作数一的来源，sel_alu_src2 告诉 EX 段操作数二的来源，data_ram_en，data_ram_wen 告诉 EX 段是否要与内存进行交互，rf_we，rf_waddr 告诉 EX 段是否要将结果写入寄存器，以及要写入寄存器的地址。

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

传给 IF 段，告诉 IF 段目前指令是否为跳转指令，以及要跳转的地址。

regfile 模块：regfile 模块的接口如下：



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	请求读取的第一个数的地址
3	rdata1	32	输出	读取出的第一个数的值
4	raddr2	5	输入	请求读取的第二个数的地址
5	rdata2	32	输出	读取出的第二个数的值
6	we	1	输入	是否要写入寄存器的写使能信号
7	waddr	5	输入	要写入的地址
8	wdata	32	输入	要写入寄存器的值
9	hi_r	1	输入	是否要读取 hi 寄存器的值的读使能信号
10	hi_we	1	输入	是否要写入 hi 寄存器的写使能信号
11	hi_data	32	输入	要写入 hi 寄存器的值
12	lo_r	1	输入	是否要读取 lo 寄存器的值的读使能信号
13	lo_we	1	输入	是否要写入 lo 寄存器的写使能信号

14	lo_data	32	输入	要写入 lo 寄存器的值
15	hilo_data	32	输出	从 hilo 寄存器中读取出来的值

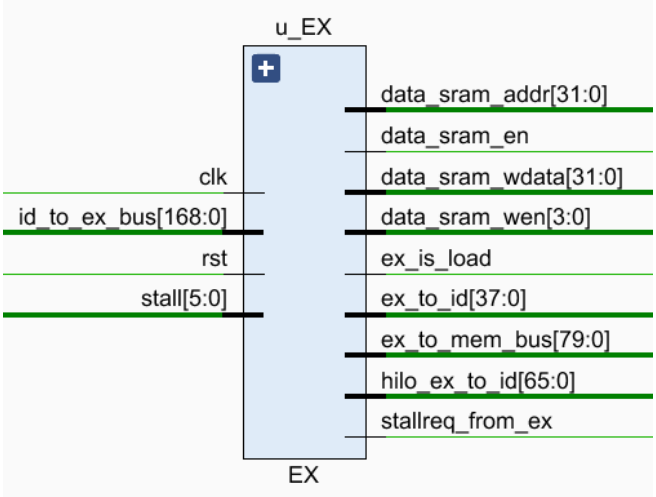
我们的 regfile 模块定义了 32 个 32 位的通用寄存器，以及一个乘除法的高位 hi 寄存器，一个乘除法的低位 lo 寄存器。**确定乘除法寄存器的返回值：**由于所有乘除法高位寄存器以及乘除法低位寄存器在调用时，不会同时调用 hi 寄存器和 lo 寄存器的值，即只会读取其中一个寄存器的值，所以我们的 hilo 寄存器在实现的时候只有一个输出的接口。在读取 hilo 寄存器中的值的时候，先判断 hi_r 是否等于 1’b1，如果等于，输出的 hilo_data 赋值为乘除法高位 hi 寄存器的值；再判断判断 lo_r 是否等于 1’b1，如果是的话那么输出的 hilo_data 赋值为乘除法低位 lo 寄存器的值，如果两个都为 0 即不需要读取 hilo 寄存器的值，输出的 hilo_data 为 0。

确定 rs 寄存器以及 rt 寄存器的值：判断 raddr1 是否为零，如果为零，就把 32’b0 赋值给 rdata1,如果不为零，就把 raddr1 对应的寄存器的值赋值给 rdata1；判断 raddr2 是否为零，如果为零，就把 32’b0 赋值给 rdata2,如果不为零，就把 raddr2 对应的寄存器的值赋值给 rdata2。

3、EX 模块

EX 模块功能：计算 ALU 的结果，根据当前指令，确定即将要写入内存的数据以及地址，或者下一步是否要从内存中读取值。

EX 模块接口：



序号	接口名	宽度	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	控制暂停信号
4	id_to_ex_bus	169	输入	ID 段传给 EX 段的数据
5	ex_to_mem_bus	80	输出	EX 段传给 MEM 短的数据
6	data_sram_en	1	输出	内存数据的读写使能信号
7	data_sram_wen	4	输出	内存数据的写使能信号
8	data_sram_addr	32	输出	内存数据存放的地址
9	ex_to_id	38	输出	EX 段传给 ID 段的数据
10	data_sram_wdata	32	输出	要写入内存的数据
11	stallreq_from_ex	1	输出	EX 发出的是否暂停的信号
12	ex_is_load	1	输出	EX 段发给 ID 段的数据，用来判断上一条指令是否是储存指令
13	hilo_ex_to_id	66	输出	EX 段将乘除法器的结果发给 ID 段的 regfile 模块

EX 模块细述：

定义如下变量 data_ram_readen,inst_mthi,inst_mtlo,inst_multu,inst_mult,inst_divu,inst_div,ex_pc,inst,alu_op,sel_alu_src1,sel_alu_src2,data_ram_en,data_ram_wen,rf_we,rf_waddr,sel_rf_res,rf_rdata1,rf_rdata2 将 ID 段传给 EX 段 id_to_ex_bus_r 复制给相应变量。

判断当前指令是否为 LW 指令，即 inst[31:26] 是否 6'b10_0011，如果是那么就将 ex_is_load 赋值为 1；如果不是，就将 ex_is_load 赋值为 0。**计算 ALU 来个操作数的值**：定义立即数符号扩展的变量并将其赋值为 {{16{inst[15]}},inst[15:0]}，定义立即数零扩展的变量并将其赋值为

{16'b0, inst[15:0]}，定义 s a 零扩展的变量并将其赋值为{27'b0, inst[10:6]}计算参与ＡＬＵ运算的操作数，根据 ID 段，传过来的操作数来源的方式，也就是 sel_alu_src1，sel_alu_src2 中的值，然后给 alu_src1，alu_src2 赋值成对应的值。调用 ALU 模块，将参与 ALU 计算的两个操作数，已经 ALU 计算的方式传给 ALU 的接口，然后从 ALU 模块中得到 ALU 计算的结果，将其赋值给 ex_result。

读写内存：将内存读写使能设置为相应的值。判断当前指令是否是 sb 指令，即 data_ram_readen 是否为 4'b0101，如果是，判断要写入内存的值，即判断要写入内存地址的后两位的值为多少，如果 ex_result[1:0] == 2'b00，那么就将内存写使能赋值为 4'b0001，表示要写入的是第一个字节，如果 ex_result[1:0] == 2'b01，那么就将内存写使能赋值为 4'b0010，表示要写入的是第二个字节，如果 ex_result[1:0] == 2'b10，那么就将内存写使能赋值为 4'b0100，表示要写入的是第三个字节，如果 ex_result[1:0] == 2'b11，那么就将内存写使能赋值为 4'b1000，表示要写入的是第四个字节；如果不是 sb 指令，那么判断是否是 sh 指令，即 data_ram_readen 是否 4'b0111，如果是，判断要写入内存的值，即判断要写入内存地址的后两位的值为多少如果 ex_result[1:0] == 2'b00，那么就将内存写使能赋值为 4'b0011，表示要写入的是第一、第二字节，如果 ex_result[1:0] == 2'b10，那么就将内存写使能赋值为 4'b1100，表示要写入的是第三、第四字节。将写内存的值赋值为当前 ALU 计算的结果。这是由于在当前情况下，写内存的值都是由参与 ALU 计算的两个操作数执行相应运算的结果。

要写入内存的数据：判断要写的数据是什么，即判断 data_sram_wen 为多少，如果为 4'b1111，表示要将写入数据来源的四个字节全部写入内存，则把 data_sram_wdata 赋值为 rf_rdata2；如果为 4'b1111，表示要将写入数据来源的四个字节全部写入内存，则把 data_sram_wdata 赋值为 rf_rdata2；如果为 4'b0001，暗示是 sb 指令，而 sb 指令是只写入最低字节，只是放置位置不同，表示要将写入数据来源的第一个字节放到第一个字节的位置，其他位填充 0 写入内存，则把 data_sram_wdata 赋值为{24'b0,rf_rdata2[7:0]}；如果为 4'b0010，暗示是 sb 指令，表示要将写入数据来源的第一个字节放到第二个字节的位置，其他位填充 0 写入内存，则把 data_sram_wdata 赋值为 {16'b0,rf_rdata2[7:0],8'b0}；如果为 4'b0100，暗示是 sb 指令，表示要将写入数据来源的第一个字节放到第三个字节的位置，其他位填充 0 写入内存，则把 data_sram_wdata 赋值为{8'b0,rf_rdata2[7:0],16'b0}；如果为 4'b1000，暗示是 sb 指令，表示要将写入数据来源的第一个字节放到第二个字节的位置，其他位填充 0 写入内存，则把 data_sram_wdata 赋值为

rf_rdata2；如果为4'b0011,，暗示该条指令是sh指令，而sh指令是值写入最低两个字节，只是最低来个字节放置的位置不同，暗示是sb指令，表示要将写入数据来源的第一、第二字节放到第一、第二个字节的位置，其他位填充0写入内存，则把data_sram_wdata赋值为{16'b0,rf_rdata2[15:0]}，如果为4'b1100,，暗示该条指令是sh指令，暗示是sb指令，表示要将写入数据来源的第一、第二字节放到第三、第四个字节的位置，其他位填充0写入内存，则把data_sram_wdata赋值为{rf_rdata2[15:0],16'b0}。调用乘除法器，并且处理乘除法器返回的值：通过

读内存：将data_sram_en赋值为对应的值后，1表示要可以读内存，0表示不可以读取内存，此外将data_sram_addr赋值为要读内存的地址，在EX会获取到想要读取内存地址的数据。

发送EX段结果给其他段：**发给WB段**：将内存的读使能信号，当前的Pc值，内存的读写使能，以及内存写使能信号，以及寄存器的写使能信号，以及寄存器要写的地址与数据。**发给ID段**：寄存器的写使能信号，以及寄存器要写的地址与数据，用来让ID段判断是否会出现相关的情况发生。还有乘除法器高位寄存器以及低位寄存器的写使能信号，以及乘除法器高位和低位要写入的数据，让ID段在调用regfile的同时，将乘除法器高位和低位的值也一并写入寄存器中，提高了CPU效率。

乘除法指令的实现：在ex段我们加入了MUL和DIV模块，借此完成乘法和除法的指令。在收到来自ID段的id_to_ex_bus后，可以判别是否为乘数法，并调用相应的乘法或除法模块进行运算。

如果是乘法，则需要声明几个变量，分别为：stallreq_for_mul、mul_ready_i、signed_mul_o、mul_opdata1_o、mul_opdata2_o、mul_start_o。stallreq_for_mul表明是否因为多周期的mul进行流水线的暂停，mul_ready_i表明乘法是否已经结束，signed_mul_o表示是否为有符号的乘法，mul_opdata1_o、mul_opdata2_o是ID段传过来的两个操作数，mul_start_o表明是否开始乘法运算。具体的乘法器模块的解释说明在下文中给出，这里需要说明的是对传入乘法器mymul的相应的值的赋值。

```
case ({inst_mult,inst_multu})
    2'b10:begin
        if (mul_ready_i == `MulResultNotReady) begin
            mul_opdata1_o = rf_rdata1;
            mul_opdata2_o = rf_rdata2;
            mul_start_o = `MulStart;
            signed_mul_o = 1'b1;
            stallreq_for_mul = `Stop;
        end
        else if (mul_ready_i == `MulResultReady) begin
            mul_opdata1_o = rf_rdata1;
            mul_opdata2_o = rf_rdata2;
            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
```

当检测到乘法的指令 inst_mult 或者 inst_multu 乘法器就会开始运作，开始前，会先判断当前乘法器的状态，若当前乘法器为空闲的状态，则将操作数传入乘法器开始运算，同时因为该乘法器是 32 周期的，因此需要对流水线进行暂停，将 stallreq_for_mul 的值改为`Stop 并传给 CTRL 模块进行流水线暂停。当乘法指令运行结束后，stallreq_for_mul 的值改为`NoStop，流水线继续运行。乘法的运行结果存放到了 mul_result 中。

除法指令的实现：

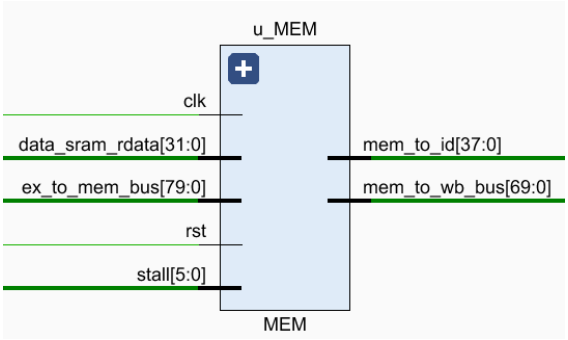
除法指令的实现与乘法类似，都是先判断是否为有符号除法或无符号除法，然后判断除法器的状态是否为空闲，除法器可用时，就把被除数和除数传入除法器，并把 stallreq_for_div 值改为`Stop 来暂停流水线。在除法结束后结果放在 div_result 中并恢复流水线继续运行。

在乘除法结束后，需要将结果存到 hilo 寄存器中。首先把 result 中的高 32 位的值和低 32 位的值放在对应的 hi、lo 的值中，然后把要写入的值和写使能信号直接发送给 ID 段。因为在 MEM 和 WB 段没有涉及到 hilo 寄存器的读写的问题，因此我们直接把 hilo 的写入的线发给了 ID 段，并没有像通用寄存器一样向后传到 WB 段再发给 ID 段。至此，乘法和除法的指令已经可以正常运行。

4、MEM 模块

MEM 模块功能：读取内存中相应地址的值（其实是在 EX 到 MEM 的过程中读取的），根据当前指令，确定要写入寄存器的值。

MEM 模块接口：



序号	接口名	宽度	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	控制暂停信号
4	ex_to_mem_bus	80	输入	EX 传给 MEM 段的数据
5	data_sram_rdata	32	输入	从内存中读出来要写入寄存器的值
6	mem_to_id	38	输出	MEM 传给 ID 段的数据
7	mem_to_wb_bus	70	输出	MEM 传给 WB 段的数据

MEM 模块细述：

```
assign {
    data_ram_readen,//79:76
    mem_pc,          // 75:44
    data_ram_en,     // 43
    data_ram_wen,    // 42:39
    sel_rf_res,      // 38
    rf_we,           // 37
    rf_waddr,        // 36:32
    ex_result        // 31:0
} =  ex_to_mem_bus_r;
```

接收从执行段传过来的值。其中 data_ram_readen 以及 data_ram_en 告诉访存段是否要进行从内存读取值的分析。mem_pc 是当前指令地址。rf_we，rf_waddr，ex_result 告诉访存段该指令是否要写入寄存器，以及要写入寄存器的地址，以及要写入寄存器的数据。

确定要写入寄存器的值：由于要写入寄存器的值不止由 ALU 计算的结果，也可能是在执行段到访存段访问内存之后得到的值。所以在访存阶段要进行分析。此处可能需要插入图片。判断是否是 **lw 指令**(lw 指令是将从内存中的值全部写入相应寄存器)，如果是，就将要写入寄存器的值，更新为从内存中读出来

的值；如果不是 lw 指令，判断是否是 **lb 指令**（lb 指令是根据要写入寄存器的地址的最后两位判断要将从内存中读出来的数据的哪一个字节写入寄存器的最低一个字节，其他字节采用符号扩展的方式），如果是，判断要写入寄存器的地址的后两位的值，如果为 2’b00,就将从内存中读出来的值的第一个字节写入寄存器的最低一个字节其他部分用第 7bit 的值进行符号扩展，如果为 2’b01,就将从内存中读出来的值的第二个字节写入寄存器的最低一个字节其他部分用第 15bit 的值进行符号扩展，如果为 2’b10,就将从内存中读出来的值的第三个字节写入寄存器的最低一个字节其他部分用第 23bit 的值进行符号扩展，如果为 2’b11,就将从内存中读出来的值的第四个字节写入寄存器的最低一个字节其他部分用第 31bit 的值进行符号扩展；如果也不是 lb 指令，判断是否是 **lbu 指令**（lbu 指令是根据要写入寄存器的地址的最后两位判断要将从内存中读出来的数据的哪一个字节写入寄存器的最低一个字节，其他字节采用零扩展的方式），如果是，判断要写入寄存器的地址的后两位的值，如果为 2’b00,就将从内存中读出来的值的第一个字节写入寄存器的最低一个字节其他部分进行零扩展，如果为 2’b01,就将从内存中读出来的值的第二个字节写入寄存器的最低一个字节其他部分进行零扩展，如果为 2’b10,就将从内存中读出来的值的第三个字节写入寄存器的最低一个字节其他部分进行零扩展，如果为 2’b11,就将从内存中读出来的值的第四个字节写入寄存器的最低一个字节其他部分进行零扩展；如果也不是 lbu 指令，判断是否是 **lh 指令**（lh 指令是根据要写入寄存器的地址的最后两位判断要将从内存中读出来的数据的哪两个字节写入寄存器的最低两个字节，其他字节采用符号扩展的方式），如果是，判断要写入寄存器的地址的后两位的值，如果为 2’b00,就将从内存中读出来的值的第一，第二字节写入写入寄存器的最低两个字节其他部分用第 15bit 的值进行符号扩展，如果为 2’b00,就将从内存中读出来的值的第三，第四字节写入写入寄存器的最低两个字节其他部分用第 15bit 的值进行符号扩展；如果也不是 lh 指令，判断是否是 **lhu 指令**（lhu 指令是根据要写入寄存器的地址的最后两位判断要将从内存中读出来的数据的哪两个字节写入寄存器的最低两个字节，其他字节采用零扩展的方式），如果是，判断要写入寄存器的地址的后两位的值，如果为 2’b00,就将从内存中读出来的值的第一，第二字节写入写入寄存器的最低两个字节其他部分进行零扩展，如果为 2’b10,就将从内存中读出来的值的第三，第四字节写入写入寄存器的最低两个字节其他部分进行零扩展，如果都不是，那么将要写入寄存器的值赋值为指令阶段计算的结果。

传值给 WB 阶段：

```
assign mem_to_wb_bus = {
    mem_pc,      // 41:38
    rf_we,       // 37
    rf_waddr,    // 36:32
    rf_wdata     // 31:0
};
```

将当前的 PC 值，以及是否要写入寄存器，以及要写入寄存器的值还有数据发给回写段。

传值给 ID 段：

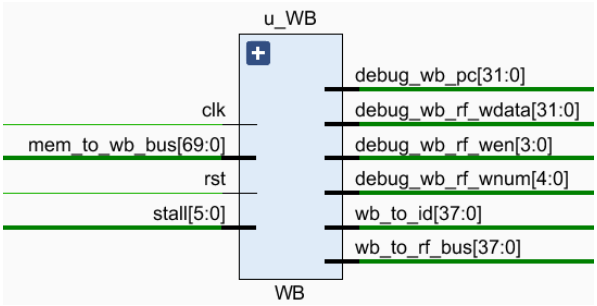
```
assign  mem_to_id =
{   rf_we,       // 37
    rf_waddr,    // 36:32
    rf_wdata     // 31:0
};
```

将当前的 PC 值，以及是否要写入寄存器，以及要写入寄存器的值还有数据发给译码段，用来判断是否发生数据相关。

5、WB 模块

WB 模块功能：将结果写入寄存器，这一段实际是在 ID 段调用 regfile 模块实现的。

WB 模块接口：



序号	接口名	宽度	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	控制暂停信号
4	mem_to_wb_bus	70	输入	MEM 传给 WB 的数据
5	wb_to_rf_bus	38	输出	WB 传给 rf 的数据
6	wb_to_id	38	输出	WB 传给 ID 的数据

7	debug_wb_pc	32	输出	用来 debug 的 pc 值
8	debug_wb_rf_wen	4	输出	用来 debug 的写使能信号
9	debug_wb_rf_wnum	5	输出	用来 debug 的写寄存器地址
10	debug_wb_rf_wdata	32	输出	用来 debug 的写寄存器数据

WB 模块细述：

接受数据：

```
assign {
    wb_pc,
    rf_we,
    rf_waddr,
    rf_wdata
} = mem_to_wb_bus_r;
```

接受从 MEM 传过来的 PC 值，以及是否要写入寄存器，以及要写入寄存器的值还有数据。

传值给 rf 寄存器：

```
assign wb_to_rf_bus = {
    rf_we,
    rf_waddr,
    rf_wdata
};
```

将当前的 PC 值，以及是否要写入寄存器，以及要写入寄存器的值还有数据发给 rf 寄存器模块。

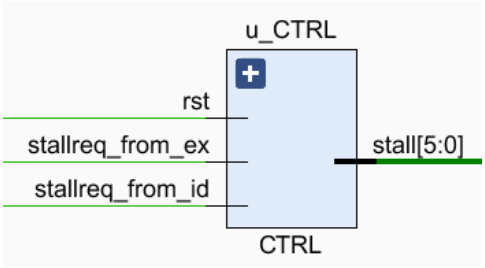
6、CTRL 模块:

CTRL 模块功能： 接收各段传递过来的流水线请求信号，从而控制流水线各阶段的运行。（在本实验中目前为止只有俩个会发送请求信号，译码阶段和执行阶段，取值和访存阶段没有暂停请求，应为这列个阶段的操作都可以在一个周期内完成）

CTRL 模块原理：

假设位于流水线第 n 阶段的指令需要多个周期，进而请求流水线暂停，那么需要保持取指令地址 PC 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保持不变，而第 n 阶段后面的指令继续运行。比如：流水线执行阶段请求暂停流水线，那么保持 PC 不变，同时保持取值，译码，执行阶段的寄存器不变，但是可以允许访存、回写的指令继续运行。

CTRL 模块接口：



序号	接口名	宽度	输入/输出	作用
1	rst	1	输入	复位信号
2	stallreq_from_ex	1	输入	处于译码阶段的指令是否请求流水线暂停
3	stallreq_from_id	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

输出暂停信号：

stall[0]为 1 表示没有暂停

stall[1]为 1 if 段暂停

stall[2]为 1 id 段暂停

stall[3]为 1 ex 段暂停

stall[4]为 1 mem 段暂停

stall[5]为 1 wb 段暂停

判断处于译码阶段的指令是否请求流水线暂停，如果请求暂停，那么就把流水线暂停控制信号赋值为 6'b001111，表示取值阶段暂停，译码阶段暂停，执行阶段暂停，访存阶段不暂停，回写阶段不暂停。

判断处于译码阶段的指令是否请求流水线暂停，如果请求暂停，那么就把

流水线暂停控制信号赋值为 6'b000111，表示取值阶段暂停，译码阶段暂停，执行阶段暂停，访存阶段不暂停，回写阶段不暂停。

如果都不暂停将流水线控制信号赋值为 6'b000000，表示不暂停。

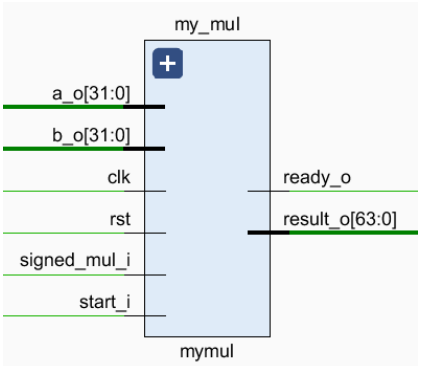
在对应的 IF、ID、EX、MEM、WB 段中都有对应的读取 stall 值控制对应模块是否暂停的部分。

7、MUL 模块:

MUL 模块功能：实现两个 32 位有符号或者无符号数的乘法，并返回一个 64 位的结果。

MUL 模块原理：先把被乘数扩展成 64 位，再左移 32 位，每一次左移之前都要判断乘数最低位是否为 1，为 1 则把那一步的被乘数加到结果里（result），为 0 则不做处理，于此同时将乘数右移一位。

MUL 模块接口：



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	是否为有符号乘法，为 1 表示有符号乘法
4	a_o	32	输入	被乘数
5	b_o	32	输入	乘数
6	start_i	1	输入	是否开始乘法运算
7	result_o	64	输出	乘法运算结果
8	ready_o	1	输出	乘法运算是否结束

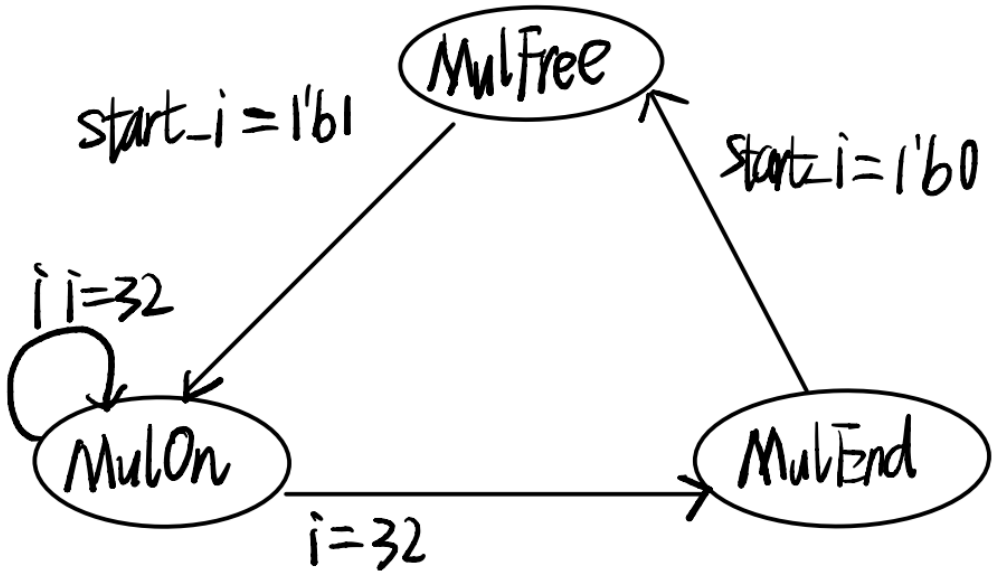
MUL 模块总述：

MUL 模块的主要部分是一个状态机，由三个状态。

MulFree：乘法模块空闲

MulOn：乘法运算进行中

MulEnd：乘法运算结束



复位的时候，MUL 模块处于 MULFREE 状态，当输入信号 start_i 为 `MulStart 时，表示乘法操作开始。

进入 MulOn 状态，使用移位乘法，经过 32 个周期，得到乘法结果，然后进入 MulEnd 模块，并通知 EX 模块得到乘法运算结果。

```
reg [5:0] i;//进行到第几位
```

表示被乘数以及乘数

```
reg [31:0] temp_opa,temp_opb;
```

MulFree：

```
`MulFree: begin         //乘法器空闲
    if (start_i== `MulStart) begin
        state <= `MulOn;
        i <= 6'b00_0000;
        if(signed_mul_i == 1'b1 && a_o[31] == 1'b1) begin
            temp_opa = ~a_o + 1;
        end else begin
            temp_opa = a_o;
        end
        if(signed_mul_i == 1'b1 && b_o[31] == 1'b1 ) begin
                temp_opb = ~b_o + 1;
        end else begin
            temp_opb = b_o;
        end
        ap <= {32'b0,temp_opa};
        ready_o <= `MulResultNotReady;
        result_o <= {`ZeroWord, `ZeroWord};
        pv <= 64'b0;
    end
end
```

开始乘法运算，如果是有符号乘法，且被除数或者乘数为负数，那么对被除数或者乘数为负数取补码，被乘数保存到 ap 的低 32 位。同时将乘法器结果置为 0,乘法结果中间值 pv 置为 0。

MulOn：

```
`MulOn: begin               //乘法运算
        if(i != 6'b100000) begin
            if(temp_opb[0]==1'b1) begin
                pv <= pv + ap;
                ap <= {ap[62:0],1'b0};
                temp_opb <= {1'b0,temp_opb[31:1]};
            end
            else begin
                ap <= {ap[62:0],1'b0};
                temp_opb <= {1'b0,temp_opb[31:1]};
            end
            i <= i + 1;
        end
        else begin
            if ((signed_mul_i == 1'b1) && ((a_o[31] ^ b_o[31])
                pv <= ~pv + 1;
            end
            state <= `MulEnd;
            i <= 6'b00_0000;
        end

    end
```

乘法运行状态：如果 i！=32，如果乘数的最低位位 1，那么将 pv 加上一个 ap，同时将 ap 左移一位，乘数右移一位；如果乘数最低为为零，将 ap 左移一位，乘数右移一位。然后将 i+1。如果 i==32，表示移位乘法结束。如果似乎有符号乘法，且乘数与被乘数一正一负，将 pv 取补码。

MulEnd：

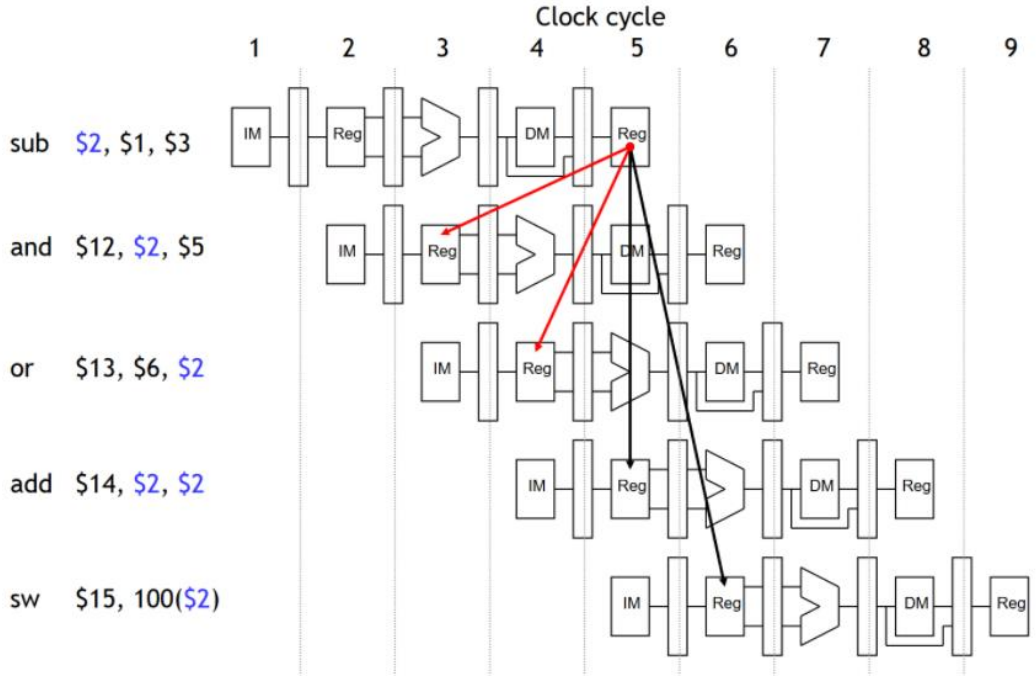
```
`MulEnd: begin          //乘法结束
    result_o <= pv;
    ready_o <= `MulResultReady;
    if (start_i == `MulStop) begin
        state <= `MulFree;
        ready_o <= `MulResultNotReady;
        result_o <= {`ZeroWord, `ZeroWord};
    end
end
```

乘法运算结束,将 pv 赋值给乘法器结果，结果设置位可获取状态。同时将乘法器设置为空闲状态，并赋为初始值。

数据相关:

问题描述：

在流水线中运行指令的过程中，难免会出现以下的问题。



可以看到，SUB 指令的结果是 AND 指令的输入，也是 OR 指令的输入；而 ADD 指令与 SW 指令的输入同样依赖 SUB 指令。其中，红线由前指向后，因此从时间流动的角度，红线就代表一处「数据冲突」，即 Data Hazard.

解决办法：数据前递 Data Forwarding

事实上，对 SUB 指令来说，其结果的产生是在其第三流水阶段：EX，也就是整个流水线 CPU 的第三个时钟周期。而对 SUB 之后的 AND 以及 OR 来说，寄存器 $2 的值是在它们的第三阶段 EX 需要的，也就是流水线 CPU 的第 4-5 个

时钟周期。纵观整个流水过程，事实上 $2 的值是在第 3 个时钟周期产生，并在第 4-5 个时钟周期需要，因此我们只需要越过流水线五个阶段中 WB（WriteBack）的过程，让第 3 个时钟周期里面计算产生的 $2 寄存器值 直接赋值 给第 4-5 个时钟周期指令 ADD 以及 OR 指令即可。

实现：

在 EX、MEM、WB（其实 WB 段可以没有）中将当前时钟周期内得到的要写入寄存器的地址以及要写入的数据，发给 ID 段，在 ID 判断当前操作数的地址是否与其将要写入寄存器的地址相等，如果相等的话，就直接将要写入寄存器的值拿过来用即可。

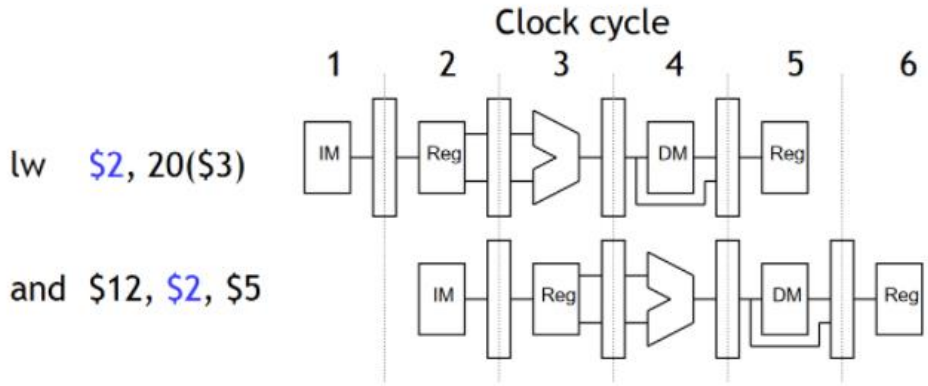
另外，在 hilo 寄存器中也会有数据相关的问题。不过和通用的寄存器不太相同的地方是，我们的 hilo 寄存器在 EX 段得到要写入的值后就会直接发送给 ID 段，在下一个周期的 ID 段完成写入，因此在 hilo 寄存器中的数据相关只存在于两条相邻的指令中，而 MEM 和 WB 段不会存在 hilo 寄存器的数据相关的问题。第一条指令需要完成 hilo 寄存器的写入，而第二条指令请求读取 hilo 寄存器的值。

解决方法同样是使用了数据前递，因为上一个周期需要写入 hilo 寄存器的值已经发送给了 ID 段，只不过还没有写入到 hilo 寄存器中。因此如果发生了 hilo 寄存器的数据相关，在该 ID 段就不需要从 hilo 寄存器中读取，直接将上一个时钟周期的 EX 段发过来的要写入 hilo 寄存器的值拿过来用就可以了。

访存冲突

问题描述：

虽然前面介绍的方法规避了 ALU 算术运算的数据冲突，但是对于需要访问数据存储器的指令（比如 LW）来说，我们并不能规避类似下面的指令带来的数据冲突：



也就是要用的寄存器的值在用的时候并不是最新的值，最新的值还没有写进去。

解决方法：

在 LW 指令的 MEM 阶段，我们就获得了相应的数据，那么，对于下一条 AND 指令，我们只需要在其 EX 阶段前将流水线 Stall 住一个时钟周期，即可将 Data Memory 的数据前递至正确的地方。这种解决方法也叫向流水线中引入一个 bubble。

实现：

在 EX 段将是否是 load 指令、和将要写入的寄存器的地址发送给 ID 段，在 ID 段进行判断，如果 ID 段请求读取数据的寄存器的地址和 EX 段将要写入的寄存器的地址想同，并且 EX 段当前是 load 类指令，则可以判断发生了访存冲突。需要在 ID 段发送给 CTRL 段请求说明 ID 段有访存冲突，然后 CTRL 段就会发送给流水线各个部位 stall 值，流水线的各个部位再根据 stall 的值进行相对应的暂停处理。等到 EX 段获得了 ID 段需要读取的数据后，再将该数据直接发送到 ID 段，然后流水线恢复正常运行。

四、组员感受以及改进意见

Szw:

在这次实验中，我们把在课堂上学到的东西运用到了实践中，让我对流水线的整体运行有了更加深刻的理解。

在编写流水线 CPU 的过程了，因为接触较少且很多地方需要自行设计，所以许多看起来较为简单的问题，因为思维不够缜密，在执行的时候也出现了很多的

困难。例如数据相关的解决，流水线暂停的实现，内存的读写的实现等问题，为此我通过查阅了资料和与队友交流，解决了这个问题。在进行流水线暂停的设计时，一开始我并没有了解它真正的原理，于是设计的出来有 bug 无法正常运行，后期通过在网上查阅资料，了解到了是指令值没有处理好，设计的程序会出现错误，我也逐一进行分析与修改，使我积累了很多的经验。

这次的实验也使我的对流水线的理解有了很大提高，而且是我更加深刻的理解了团队合作以及总体架构的重要性。在较短的时间内想要完成较多的任务，只有通过团队合作的方式。而要想使团队合作能够顺利进行，就需要使用模块化的的设计方法，合理安排软件的结构。总体上说，这次课程设计加深了我对理论的理解，增强了我的编码能力，让我积累了更多的有关于团队合作的经验，使我受益匪浅。

意见：在实验刚开始的阶段，因为我们的有关 verilog 的编程经验很少，在拿到项目的代码后面对多个模块有点迷茫不知道该从何开始。如果在实验前进行过更多的有关 verilog 的编程或者在实验开始时有更详细的说明文档就更容易上手实验部分了。

Zht：

在这次实验中，我深刻体会到“罗马非一日建成”这句话，外表看起来巨大、庞杂的罗马，也是通过人们一步一步、一天一天、一点一点建成的，处理器也是如此，我们不应该被处理器的神秘吓到，从最简单的地方入手，逐步增加功能，逐步增加指令，完善设计，一行代码一行代码地去书写，一步一步，脚踏实地，等有一天回头的时候，会发现你已经走了很远，实现了很多的指令，克服了很多困难，慢慢的去攻克这 64 个点，去完成数据相关等等。总的来说，这一次计算机系统的实验。让我对于 CPU 有了一个更加深入的了解和认识，收获很多。

意见:我希望学长能够在明年的实验中先讲一下 verilog 的基础部分，以及其中的时序逻辑以及组合逻辑，这样的话就不会像今年这样大家上手这们慢了。

五、参考资料

1、张晨曦 著《计算机体系结构》（第二版） 高等教育出版社

2、雷思磊 著《自己动手写 CPU》 电子工业出版社

3、（美）DavidA.Patterson、John L.Hennessy 著 《计算机组成与设计：硬件、软件接口（原书第 4 版）》

4、Yale N.Patt 著 《计算机系统概论（原书第 2 版）》

4、助教给出的龙芯杯官方的参考文档