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 班

组长：于之晟

组员：黄元通

报告日期：2021.12.18

目录

CPU 搭建

一、概要

1. 工作量

组长：于之晟

工作量：60%

主要完成工作：

1. 自实现乘法器；
2. ID 段流水线暂停机制；load、store 访存指令的实现机制；乘法器暂停机制的实现；跳转指令机制；
3. 通过 point 1 的指令添加；通过 point 36 的指令添加；通过 point 64 的指令添加。

成员：黄元通

工作量：40%

主要完成工作：

1. 实验报告；
2. forwarding 技术； HILO 寄存器机制；HILO 寄存器 forwarding 技术；移动指令及其实现机制；乘法器与除法器接入机制；
3. 通过 point 43 的指令添加；通过 point 51 的指令添加；通过 point 58 的指令添加。

2. 指令集

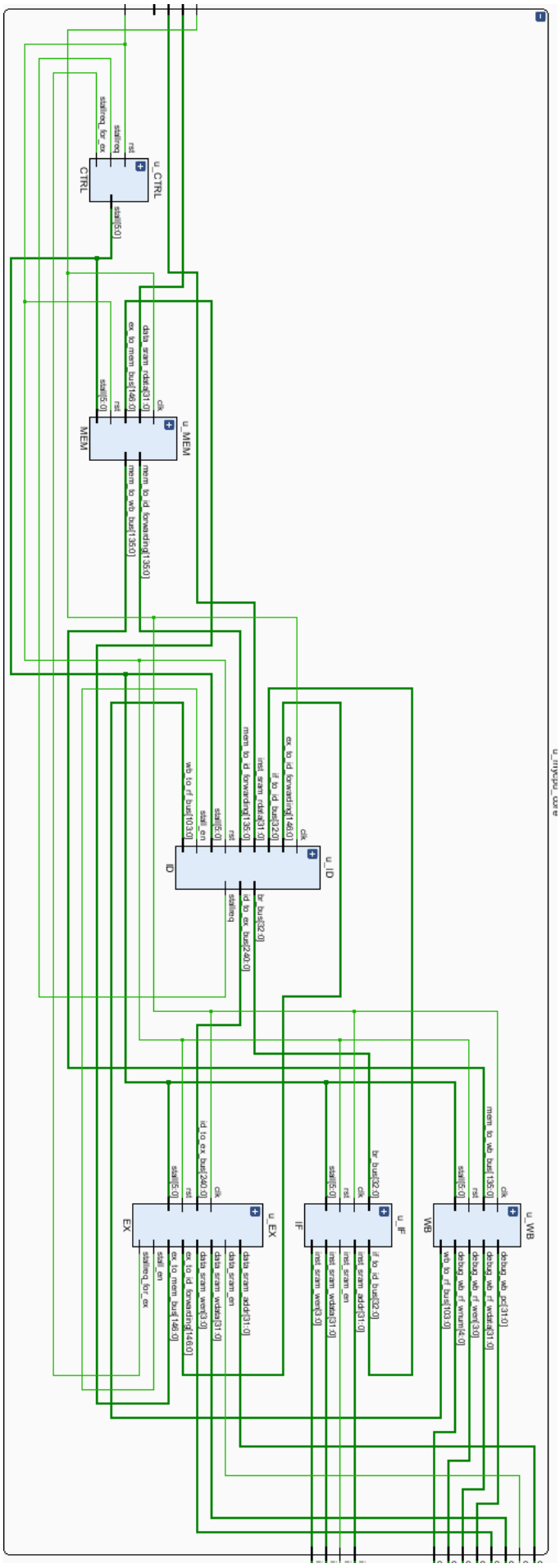
完成了通过功能测试第 64 个测试点的所有指令。

```
inst_ori , inst_lui , inst_addiu, inst_beq , inst_subu,
inst_addu, inst_jal , inst_jr   , inst_sll , inst_or  ,
inst_lw  , inst_xor , inst_sltu , inst_bne , inst_sw  ,
inst_slt , inst_slti, inst_sltiu, inst_j   , inst_add ,
inst_addi, inst_sub , inst_and  , inst_andi, inst_nor ,
inst_xori, inst_sllv, inst_sra  , inst_srav, inst_srl , inst_srlv ,
inst_bgez, inst_bgtz, inst_blez , inst_bltz, inst_bgezal, inst_bltzal, inst_jalr,
inst_div , inst_divu , inst_mult, inst_multu,
inst_mflo, inst_mfhi , inst_mthi, inst_mtlo ,
inst_lb  ,  inst_lbu , inst_lh  , inst_lhu  , inst_sb, inst_sh
```

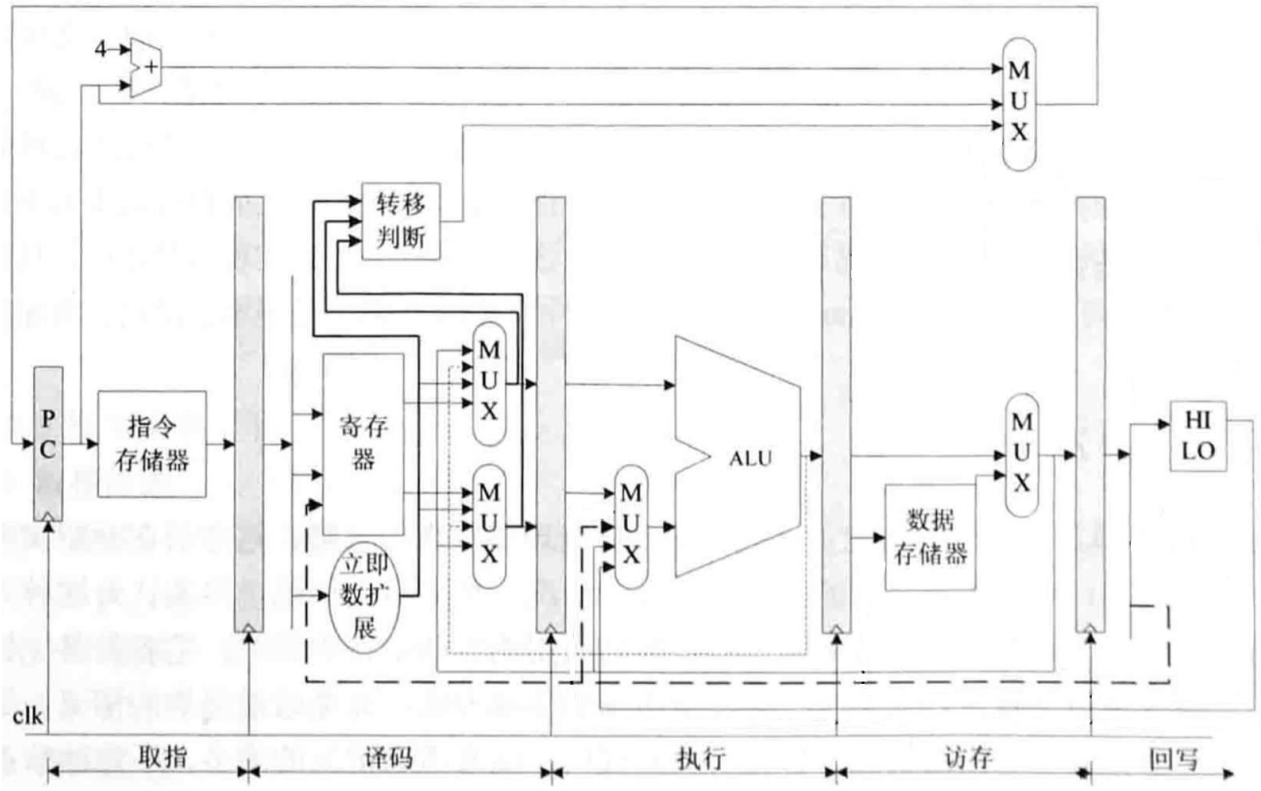
3. 运行环境及使用工具

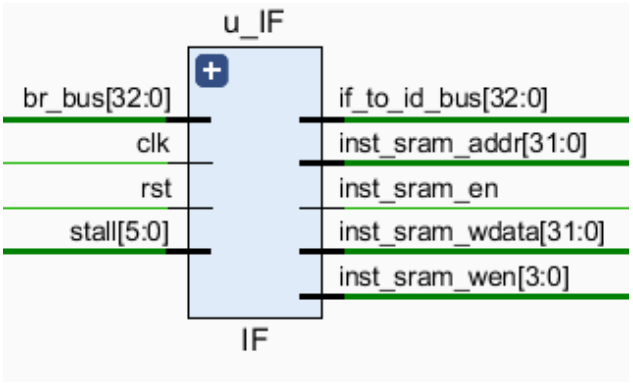
编程语言使用 Verilog，实验使用运行在 Windows 10 系统下的 Vivado 2019.2 版本软件进行编写和测试运行，以及 Microsoft Visual Studio Code 1.63 版本软件进行编写。所有文件的编码格式为 GB 2312，换行符为 CRLF。

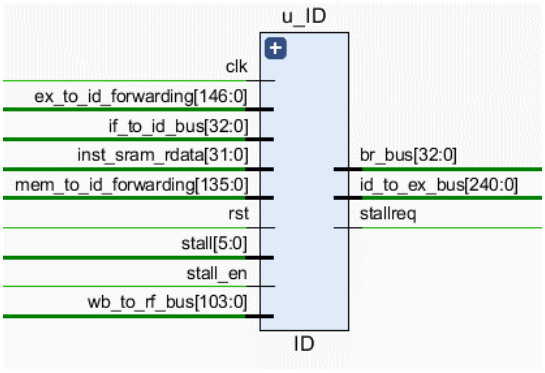
4. 总体设计

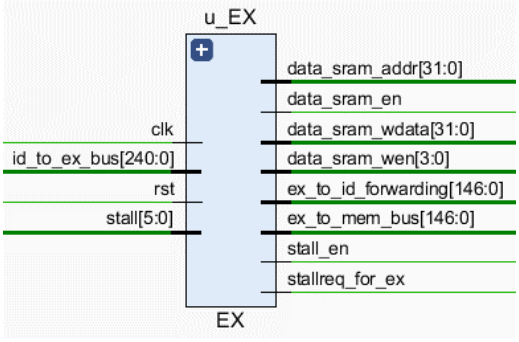


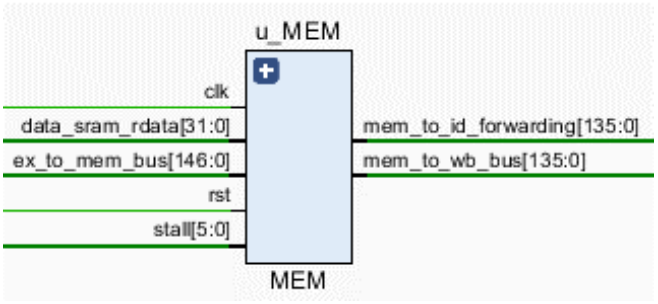
5. 流水段连接图

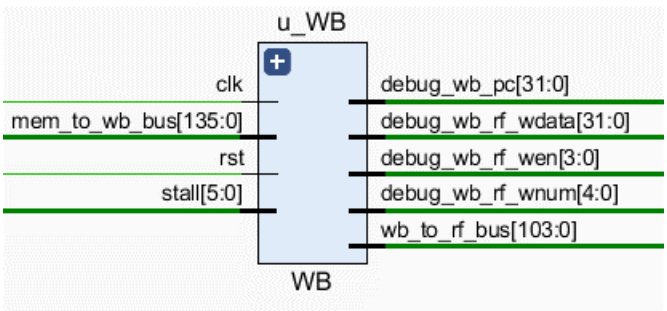

4
+
M
U
X
转移
判断
P
C
指令
存储器
寄存
器
立即
数扩
展
ALU
数据
存储器
HI
LO
clk
取指
译码
执行
访存
回写


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_ID
clk
ex_to_id_forwarding[146:0]
if_to_id_bus[32:0]
inst_sram_rdata[31:0]
mem_to_id_forwarding[135:0]
rst
stall[5:0]
stall_en
wb_to_rf_bus[103:0]
br_bus[32:0]
id_to_ex_bus[240:0]
stallreq
ID


u_EX
clk
id_to_ex_bus[240:0]
rst
stall[5:0]
data_sram_addr[31:0]
data_sram_en
data_sram_wdata[31:0]
data_sram_wen[3:0]
ex_to_id_forwarding[146:0]
ex_to_mem_bus[146:0]
stall_en
stallreq_for_ex
EX


u_MEM
clk
data_sram_rdata[31:0]
ex_to_mem_bus[146:0]
rst
stall[5:0]
mem_to_id_forwarding[135:0]
mem_to_wb_bus[135:0]
MEM


u_WB
clk
mem_to_wb_bus[135:0]
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_rf_bus[103:0]
WB

二、详细说明

1. forwarding 技术的实现（黄元通）

（1）理论

对于本 CPU，只有在 WB 阶段才会写寄存器，因此不存在 WAW 相关。又因为只能在流水线 ID 读寄存器、WB 写寄存器，所有不存在 WAR 相关，所以 CPU 流水线只存在 RAW 相关。

通过定向技术而非暂停流水线以解决数据相关问题。其主要原理为：在发生数据相关时，将计算结果从其产生的地方直接送到需要它的地方，从而解决数据相关问题且避免暂停；当定向硬件检测到前面某条指令的结果寄存器就是当前指令的源寄存器时，控制逻辑会将前面那条指令的结果直接从其产生的地方定向到当前指令所需的位置。

（2）提出问题

例如考虑如下三条指令：

ADD R1, R2, R3
LW R4, 0 , (R1)
SW 12(R1), R4

将 R2、R3 的值求和放入 R1，再将 R1 作为地址，从内存中取出 address(R1)处的值存入 R4，再将 R4 的值存入内存。考察 Lw 指令处于 ID 阶段时（即图中的 CC3 时钟周期），因为第一条指令 ADD 计算出来的值直到 WB 段（即图中的 CC5 时钟周期）才能写入寄存器，而当前的 LW 指令在 CC3 时钟周期已经开始译码。

显然若不做任何调整直接按原来的流水线执行，那么 LW 在 CC3 时钟周期便会向 R1 进行读取操作，而此时 LW 在 CC3 时钟周期在 R1 中读取到的数据并不是 ADD 指令计算出的，因为 ADD 指令直到 CC5 时钟周期才会将其计算出的结果存入 R1 寄存器。所以 LW 指令取到的 R1 中的值是错误的，所以会导致 CPU 的计算结果错误。

为避免这种错误的发生，自然可想到将 ADD 计算结果提前传送给 LW 指令，使 LW 指令在 ADD 按常规顺序在 CC5 写寄存器 R1 前，就能得到 ADD 的计算结果。而 ADD 指令计算结果最早产生是在 CC3 完成后，LW 指令最早用到该计算结果是在 CC4 时钟周期，所以为 forwarding 技术提供了实现条件。

可将 ADD 指令在 EX 段 CC3 时钟周期完成后产生的计算结果，通过额外的连线传送到 LW 指令在 CPU 流水线中 EX 段的输入处。同理，还需要增加 CPU 流水线 MEM 段输出向 EX 段输入的 forwarding 连线等，如下图 2-1-1 所示：

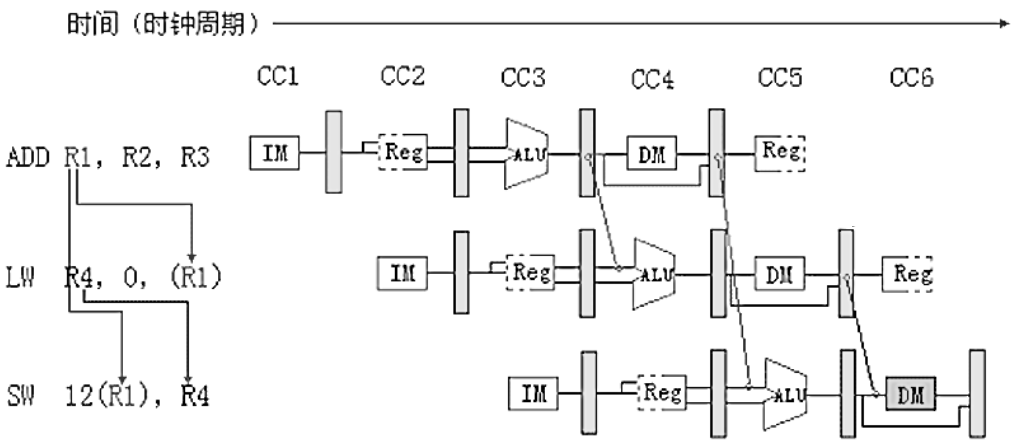


图 2-1-1 相邻指令间存在数据相关

（3）具体实现

在具体实现时，由于在 ID 段即可得到当前该条指令的源寄存器地址，因此，只需将后面的 EX、MEM 段中（分别为前 1 条、前 2 条）指令的目标寄存器地址比较即可：

1. 若 EX、MEM 段中指令需要写入（即为本实验中的 forwarding_ex_rf_we、forwarding_mem_rf_we 信号为 1）；
2. 且 EX、MEM 段中指令的目标寄存器地址（即为本实验中的 forwarding_ex_rf_waddr、forwarding_mem_rf_waddr）与当前该条指令的源寄存器地址相同；

则表示发生了 RAW 数据相关。因此在 ID 段，将当前该条指令的两个源操作数 selected_rdata1、selected_rdata2 改为使用 EX 或 MEM 段 forwarding 传来的最新数值。

即添加的两条数据 forwarding 线路如下图所示，将从 EX、MEM 将向流水线后传递的数据都传向 ID 段：

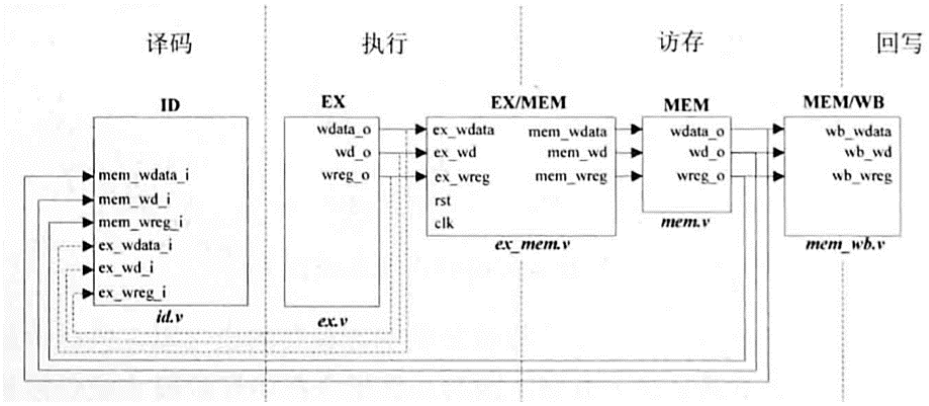


图 2-1-2 为实现 forwarding 对 CPU 结构做出的修改

在 ID 段添加的接口如下表所示：

序号	接口名	宽度（bit）	输入/输出	作用
1	forwarding_ex_rf_we	1	输入	处于访存 EX 阶段的指令是否要写目的寄存器

2	forwarding_ex_rf_waddr	5	输入	处于访存 EX 阶段的指令要写的目的寄存器地址
3	forwarding_ex_result	32	输入	处于访存 EX 阶段的指令要写入目的寄存器的数据
4	forwarding_mem_rf_we	1	输入	处于访存 MEM 阶段的指令是否要写目的寄存器
5	forwarding_mem_rf_waddr	5	输入	处于访存 MEM 阶段的指令要写的目的寄存器地址
6	forwarding_mem_rf_wdata	32	输入	处于访存 MEM 阶段的指令要写入目的寄存器的数据

表 2-1 译码 ID 段增加的接口

具体实现方案如下：

1. 在 EX, MEM 分别 output wire [`EX_TO_MEM_WD-1:0] ex/mem_to_id_forwarding。含有：rf_we, rf_waddr, ex_result 等数据。
2. 在 ID 中用 input wire [`EX_TO_MEM_WD-1:0] ex/mem_to_id_forwarding 接收，并拆为 wire forwarding_ex/mem_rf_we, forwarding_ex/mem_rf_waddr, ex/mem_forwarding_wdata 等
3. 在 ID 中，进行是否发生 forwarding 的判断：
   i. forwarding_ex/mem_rf_we 为 1，且(&)forwarding_ex/mem_rf_waddr 要写入的目标寄存器和当前 ID 段指令的 rs 寄存器为同一个(==)，表示前面指令已将源寄存器修改（rdata1 发生数据相关）
   ii. 则将 selected_rdata1 赋值为相应的 ex/mem_forwarding_wdata
   iii. 若是和 rt 寄存器为同一个（rdata2 发生数据相关），则将 selected_rdata2 赋值为相应的 ex/mem_forwarding_wdata（例如 subu 指令，rt 就是第二个源寄存器）
   iv. 否则 selected_rdata1=rdata1，selected_rdata2=rdata2
4. 在 mycpu_core.v 中加入相关连接

（4）核心代码

核心部分代码（在 ID 段进行判断是否需要使用 forwarding 线路中数值）如下代码框所示：

```
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;
```

2. ID 段流水线暂停机制的实现（于之晟）

当进行 load 类型的指令时,如果出现寄存器冲突，则需要在流水线译码阶段即 ID 段暂停一个周期，使得先完成 load 指令；

(1).首先需要在流水线执行阶段即 EX 段判断当前指令是否是 load 类型的指令：

stall_en=(inst[31:26]==6'b10_0011|inst[31:26]==6'b10_0000|inst[31:26]==6'b10_0100|inst[31:26]==6'b10_0001|inst[31:26]==6'b10_0101)?1'b1:1'b0;

其中 stall_en 作为判断指令 inst 操作码位的使能信号，如果是 load 指令则为 1'b1；反之是 1'b0；

(2).然后将这个信号传递到 ID 段的 stall_en，继续判断寄存器冲突:

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

之前设置的从 EX 段到 ID 段的数据通路中的 forwarding_ex_rf_waddr 传递了 load 指令所要使用的寄存器地址，将他与 rs 和 rt 进行比对，来判断是否存在寄存器冲突；存在则将 stall_reg 赋值为`Stop(1'b1),反之则是`NoStop (1'b0);

(3).之后继续将 stall_reg 信号传递到 ctrl.v 文件中，对流水线暂停信号 stall 进行赋值：

```
if(stallreq == `Stop) begin    //处于流水线译码阶段的指令是否请求暂停
stall = `StallBus'b00_0111;
```

(4).当 stall = `StallBus'b00_0111;stall 信号会传递到 ID 段及其他段；

ID 段中

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

根据 stall[0]和 stall[1]来对 if_to_id_bus_r 进行赋值处理，确保 ID 段进行一个周期的暂停；首先对 if_to_id_bus_r 赋初值 0；当 stall[1]==`Stop && stall[2]==`NoStop(流水线译码阶段需要暂停)，if_to_id_bus_r 仍保持值为 0，使得译码阶段获取不到数据；当 stall[1]==`NoStop

时流水线译码阶段正常进行，if_to_id_bus_r 获取从 IF 段得到的值 if_to_id_bus_r，流水线译码阶段正常进行

(5).但是实际上指令也需要对应的延长一个周期，来确保取指地址和指令 inst 对齐；

所以设立了寄存器 id_stop,当 stall[2] == `Stop 时，将他赋值为 1'b1；其他情况赋值为 1'b0；

由于寄存器赋值后会在下一个周期读取到，所以将他作为一个信号，用来使流水线译码阶段的指令信号选择保留上个周期的值，或者是读取新值：

```
assign inst = id_stop ? inst : inst_sram_rdata;
```

这样 ID 段的流水线暂停就完成了

注：ctrl.v:控制流水线暂停，主要负责传递各流水段的暂停信号

```
input wire rst                  //复位信号
input wire stallreq,            //处于流水线译码阶段的指令是否请求暂停
input wire stallreq_for_ex,     //处于流水线执行阶段的指令是否请求暂停
output reg [`StallBus-1:0] stall//向各流水线发送暂停信号
```

Stall	功能
Stall[0]	取指地址 PC 是否保持不变,为 1 就是保持不变
Stall[1]	流水线取指阶段是否暂停，为 1 就是暂停
Stall[2]	流水线译码阶段是否暂停，为 1 就是暂停
Stall[3]	流水线执行阶段是否暂停，为 1 就是暂停
Stall[4]	流水线访存阶段是否暂停，为 1 就是暂停
Stall[5]	流水线回写阶段是否暂停，为 1 就是暂停

3. load、store 访存指令的实现机制（于之晟）

(1)load 指令的实现

load 指令的目标是在流水线访存阶段从存储器中获取值赋值给 rf_wdata，从而存储在对应的寄存器中。

1、获取 load 指令的使能信号：

在流水线译码阶段即 ID 段获取 load 指令的使能信号，并且存储在 op_load 中

op_load	对应指令
op_load[0]	inst_lw
op_load[1]	inst_lb
op_load[2]	inst_lbu
op_load[3]	inst_lh
op_load[4]	inst_lhu

表 2-3-1 op_load 信号各 bit 位功能

op_load 跟随 id_to_ex_bus 传递到流水线执行阶段即 EX 段，再跟随 ex_to_mem_bus 传递到流水线访存阶段即 MEM 段；

其次在流水线译码阶段要获取当前是否为 load 指令的使能信号：

```
// 0 from alu_res ; 1 from ld_res
assign sel_rf_res = inst_lw | inst_lb | inst_lbu | inst_lh | inst_lhu;
```

跟随数据通路传递到流水线访存阶段，用于向寄存器中的值的获取判断，使从存储器中取值。

2、同时在 EX 段设置 alu_result 存储 alu.v 中运算方式结果和运算方式的使能信号；并且跟随流水线执行阶段到流水线访存阶段的数据通路已 ex_result 传递到流水线访存阶段即 MEM 段

3、mem_result 结果判断：

例如：

```
mem_result=op_load[0]?data_sram_rdata//inst_lw
            :(op_load[1]&(ex_result[1:0]==2'b00))?
            {{24{data_sram_rdata[7]}},data_sram_rdata[7:0]}(余下省略)
            : 32'b0
```

其中 op_load 代表当前 load 指令中的各指令的使能信号情况，ex_result 中的最低位则用来表示应该获取 data_sram_rdata 的哪部分字节。

[1].当 op_load[0]为 1'b1 时(当前指令是 lw 指令)，mem_result 获取 data_sram_rdata 的全部字节；

[2].当 op_load[1]为 1'b1 时(当前指令是 lb 指令):此时 ex_result 最低两位分别有四种情况，分别是 00，01，10，11，代表着 data_sram_rdata 从低到高四个字节；此时 mem_result 获取 data_sram_rdata 的对应字节并对余下空位进行符号扩展；

[3].当 op_load[2]为 1'b1 时(当前指令是 lbu 指令):此时 ex_result 最低两位分别有四种情况，分别是 00，01，10，11，代表着 data_sram_rdata 从低到高四个字节；此时 mem_result 获取 data_sram_rdata 的对应字节并对余下空位进行零扩展；

[4].当 op_load[3]为 1'b1 时(当前指令是 lh 指令):此时 ex_result 最低两位分别有两种情况，分别是 00，10，代表着 data_sram_rdata 从低到高四个字节中的各两个(将四个字节分为低位两字节和高位两字节)；此时 mem_result 获取 data_sram_rdata 的对应字节并对余下空位进行符号扩展；

[5].当 op_load[4]为 1'b1 时(当前指令是 lhu 指令):此时 ex_result 最低两位分别有两种情况，分别是 00，10，代表着 data_sram_rdata 从低到高四个字节中的各两个(将四个字节分为低位两字节和高位两字节)；此时 mem_result 获取 data_sram_rdata 的对应字节并对余下空位进行零扩展；

4、向寄存器赋值的获取:

```
rf_wdata = sel_rf_res ? mem_result          //data from memory
                      : ex_result;          //data from alu or hilo
```

对于 rf_wdata 取值判断，实际上当为 load 指令时(sel_rf_res 此时为 1'b1)，应当从存储器中取值，反之从其他寄存器中取值。

(2) store 指令的实现

store 指令的目标是在流水线执行阶段从寄存器中获取值赋值给对应的存储器中。

1、获取 store 指令的使能信号：

在流水线译码阶段即 ID 段获取 store 指令的使能信号，并且存储在 op_store 中

op_store	对应指令
op_store [0]	inst_sw
op_store [1]	inst_sb
op_store [2]	inst_sh

表 2-3-2 op_store 信号各 bit 位功能

op_store 跟随 id_to_ex_bus 传递到流水线执行阶段即 EX 段;

同时流水线译码阶段中使用 data_ram_wen 来存储存储器的写使能信号：

assign data_ram_wen = (inst_sw | inst_sb | inst_sh)? 4'b1111: 4'b0000;

这个信号同样跟随 id_to_ex_bus 传递到流水线执行阶段即 EX 段

2、使能信号的赋值和数据存储：

流水线执行阶段即 EX 段中的 data_sram_en 获取来自流水线译码阶段的 data_ram_en 的值；

data_sram_wen 则受到 op_store 和 alu_result 的控制：

例如：

```
data_sram_wen=op_store[0]?4'b1111                    //inst_sw
                    :(op_store[1]&&(alu_result[1:0]==2'b00)) ? 4'b0001
```

其中 op_store 代表当前 store 指令中的各指令的使能信号情况，alu_result 中的最低位则用来表示应该获取怎样的值。

[1].当 op_store [0]为 1'b1 时(当前指令是 sw 指令)，data_sram_wen 取值为 4'b1111；

[2].当 op_store [1]为 1'b1 时(当前指令是 sb 指令):此时 alu _result 最低两位分别有四种情况，分别是 00，01，10，11，此时 data_sram_wen 分别取 4'b0001，4'b0010，4'b0100，4'b1000；

[2].当 op_store [1]为 1'b1 时(当前指令是 sh 指令):此时 alu _result 最低两位分别有四种情况，分别是 00，10，此时 data_sram_wen 分别取 4'b0011，4'b1100；

流水线执行阶段的 data_sram_addr 赋值为 alu_result(访问的地址);

data_sram_wdata 则是用来存储要存入存储器的数据，受到 store 指令的控制：

例如：

```
data_sram_wdata = op_store[0]? rf_rdata2                    //inst_sw
                    :op_store[1]?{4{rf_rdata2[7:0]}}        //inst_sb
                    :op_store[2]?{2{rf_rdata2[15:0]}}       //inst_sh
                    :32'b0;
```

其中当指令为 sw 指令时，直接获取 rt 中的数据存入存储器；当当前指令是 sb 指令时，将 rt 中数据的最低位字节四倍化存入存储器中；当当前指令是 sh 指令时，将 rt 中数据的最低两位字节两倍化存入存储器中；

注：在 data_sram_wdata 赋值中之所以不选择符号扩展或者是零扩展，一方面是参考文档中没有具体说明，另一方面这种做法可以有效防止使用时取偏数据。

4. HILO 寄存器机制及其 forwarding 技术的实现（黄元通）

（1）HILO 寄存器机制实现

在实现乘法器、除法器及其数据有关的移动指令前，需要添加实现 HI 寄存器和 LO 寄存器。HI 寄存器存储乘法器结果的后 32 位（乘积的高半部分）、除法器结果的后 32 位（除法的余数）；LO 寄存器存储乘法器结果的前 32 位（乘积的低半部分）、除法器结果的低 32 位（除法的商）。可将 HILO 寄存器均与其他三十二个通用寄存器 regfile 放置于同一个模块中，以增强电路的清晰度与可读性。

参照现有的 regfile 寄存器部分，HILO 寄存器模块设计思路如下：

1、 将 HI 寄存器和 LO 寄存器添加在“regfile.v”文件的 module regfile 模块中：reg [31:0] hi, lo。

2、 并为其各自添加相应的写入使能信号 input wire hi_we、和 input wire lo_we。

3、 当 hi_we 为 1 时，表示需要对 HI 寄存器进行写入，将 HI 寄存器的值直接赋值为模块输入接口的 hi_in，为 0 表示不需要对 HI 寄存器进行写入，其值保持不变；

4、 当 hi_lo 为 1 时，表示需要对 LO 寄存器进行写入，将 LO 寄存器的值直接赋值为模块输入接口的 lo_in，为 0 表示不需要对 LO 寄存器进行写入，其值保持不变。

另一方面，为实现 HI 寄存器和 LO 寄存器接入 CPU 流水线，同样需要将其值和写使能信号在流水线中传递。可通过拓展现有的 ID_TO_EX_WD、EX_TO_MEM_WD、MEM_TO_WB_WD、WB_TO_RF_WD 等各个模块接线（两个寄存器值分别为 32 位，两个写使能信号分别为 1 位，因此都增加 64+2=66 位），完成 hilo 写使能信号及值在流水线中传递。

至此，regfile 模块修改如下表 2-4 所示：

序号	接口名	宽度（bit）	输入/输出	作用
1	rst	1	输入	复位信号
2	hi_we	1	输入	HI 寄存器写使能信号
3	lo_we	1	输入	LO 寄存器写使能信号
4	hi_i	32	输入	要写入 HI 寄存器的值
5	lo_i	32	输入	要写入 LO 寄存器的值
6	hi_o	32	输出	HI 寄存器的输出值
7	lo_o	32	输出	LO 寄存器的输出值

表 2-4　HILO 寄存器在 regfile 模块增加的接口

对 HI 寄存器和 LO 寄存器在 regfile 模块输入输出赋值处理的核心代码如下左下角代码框所示；对模块间总线的拓展示例如右下角代码框所示（ex_to_mem_bus、mem_to_wb_bus、wb_to_rf_bus 总线的拓展也类似）：

```
always @ (posedge clk) begin
    if (rst) begin
        hi <= 32'b0;
    end
    else if (hi_we) begin
        hi <= hi_i;
    end
end
always @ (posedge clk) begin
    if (rst) begin
        lo <= 32'b0;
    end
    else if (lo_we) begin
        lo <= lo_i;
    end
end
assign hi_o = hi;
assign lo_o = lo;
```

```
assign id_to_ex_bus = {
    ......
    // hilo_reg's
    hi_we,          // 224
    lo_we,          // 223
    selected_hi_rdata,// 191:222
    selected_lo_rdata,// 159:190
        ......
};
```

（2）HILO 寄存器的数据相关问题

同其他三十二个通用寄存器一样，HI 寄存器和 LO 寄存器同样会出现 RAW 数据相关现象。例如考虑如下两条指令：

MTHI rs

MFHI rd

将寄存器 rs 中的值放入 HI 寄存器，再将 HI 寄存器中的值放入寄存器 rd。考察 MFHI 指令处于 ID 译码阶段时（即图中的 CC2 时钟周期），因为第一条指令 MTHI 对寄存器 HI 值的修改结果直到 MFHI 指令处于 WB 回写阶段（即图中的 CC5 时钟周期）才能写入 HI 寄存器，而第二条 MFHI 指令在其 ID 译码周期（即图中的 CC3 时钟周期）已经开始译码从 HI 寄存器中读取数据。

显然若不做任何调整直接按原来的流水线执行，那么第二条的 MFHI 指令在 CC3 时钟周期便会向 HI 寄存器进行读取操作，而此时 MFHI 在 CC3 时钟周期在 HI 寄存器中读取到的数据并不是 MTHI 指令修改得到的正确值，因为 MTHI 指令直到 CC5 时钟周期才会将其获得到的结果存入 HI 寄存器。所以 MFHI 指令取到的 HI 寄存器中的值是错误的，所以会导致 CPU 的计算结果错误。

为避免这种错误的发生，自然可想到将 MTHI 计算结果提前传送给 MFHI 指令，使 MFHI 指令在 MTHI 按常规顺序在 CC5 写寄存器 HI 前，就能得到 MTHI 的计算结果，如图 2-4-1 所示。而 MTHI 指令计算结果最早产生是在 CC3 完成后，MFHI 指令最早用到该计算结果是在 CC4 时钟周期，所以为 forwarding 技术提供了实现条件。

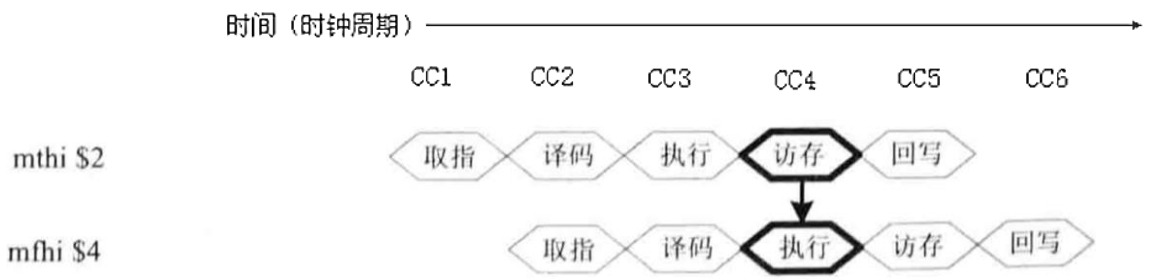


图 2-4-1　相邻指令间存在 HILO 寄存器数据相关

可将 MTHI 指令在 EX 段 CC3 时钟周期完成后获取到的结果，通过额外的连线传送到 MFHI 指令在 CPU 流水线中 EX 段的输入处。同理，还需要增加 CPU 流水线 MEM 段输出向 EX 段输入的 forwarding 连线等。

（3）HILO 寄存器的 forwarding 技术实现

在具体实现时，由于在 ID 段即可得到当前该条指令的源寄存器地址，因此，只需将后面的 EX、MEM 段中（分别为前 1 条、前 2 条）指令的目标寄存器地址比较即可。因为 hilo 在 hi_we、lo_we 为 1 时即为需要写操作无需外加地址判断，所以采用在“1. forwarding 技术的实现”的（3）具体实现中提出的 forwarding 线路基础上，同步拓宽 ex_to_id_forwarding、mem_to_id_forwarding，同样是都增加 64+2=66 位（两个寄存器值分别为 32 位，两个写使能信号分别为 1 位）。

判断：

若 EX、MEM 段中指令需要对 HI 寄存器或是 LO 寄存器进行写入（即为本实验中 EX 段向 ID 段前传的信号中的 forwarding_ex_hi_we、forwarding_ex_lo_we 信号为 1），MEM 段也同样类型）。

则表示可能发生 HI 寄存器或是 LO 寄存器的 RAW 数据相关。因此在 ID 段，将当前该条指令的 HI 寄存器和 LO 寄存器的值 selected_hi_rdata、selected_lo_rdata 改为使用 EX 或 MEM 段 forwarding 传来的最新数值。

即添加的两条数据 forwarding 线路如下图 2-4-2 所示，将从 EX、MEM 将向流水线后传递的有关 HI 寄存器和 LO 寄存器的数据都传向 ID 段：

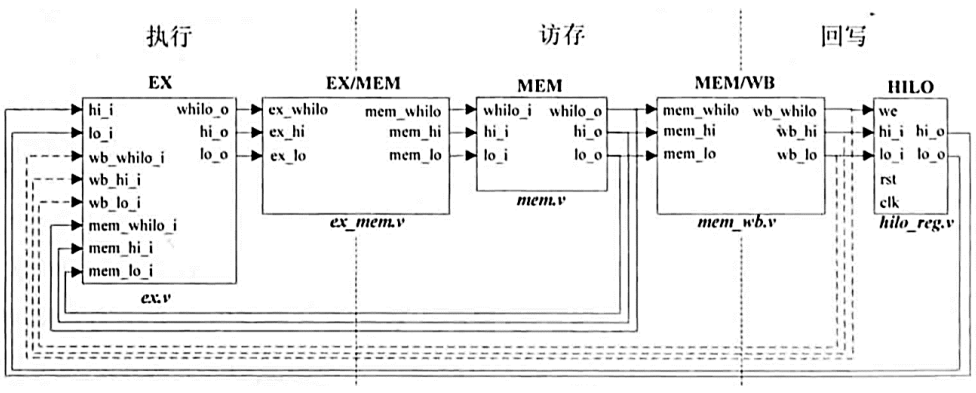


图 2-4-2　为实现 HILO 寄存器的 forwarding 对 CPU 结构做出的修改

在 ID 段添加的接口如下表所示：

序号	接口名	宽度（bit）	输入/输出	作用
1	forwarding_ex_hi_we	1	输入	处于执行 EX 阶段的指令是否要写 HI 寄存器
2	forwarding_ex_lo_we	1	输入	处于执行 EX 阶段的指令是否要写 LO 寄存器
3	forwarding_ex_hi_result	32	输入	处于执行 EX 阶段的指令要写入 HI 寄存器的数据
4	forwarding_ex_lo_result	32	输入	处于执行 EX 阶段的指令要写入 LO 寄存器的数据
5	forwarding_mem_hi_we	1	输入	处于访存 MEM 阶段的指令是否要写 HI 寄存器
6	forwarding_mem_lo_we	1	输入	处于访存 MEM 阶段的指令是否要写 LO 寄存器
7	forwarding_mem_hi_result	32	输入	处于访存 MEM 阶段的指令要写入 HI 寄存器的数据
8	forwarding_mem_lo_result	32	输入	处于访存 MEM 阶段的指令要写入 LO 寄存器的数据

表 2-4　HILO 寄存器 forwarding 在 ID 段增加的接口

具体实现方案如下：

1. 在 EX, MEM 在拓宽的 forwarding 总线 output wire [`EX_TO_MEM_WD-1:0] ex/mem_to_id_forwarding 中，添加：hi_we、lo_we、hi_result、lo_result 等数据。
2. 在 ID 中在拓宽的 forwarding 总线 input wire [`EX_TO_MEM_WD-1:0] ex/mem_to_id_forwarding 接收，并拆为 forwarding_ex_hi_we、forwarding_ex_lo_we、forwarding_ex_hi_result、forwarding_ex_lo_result、forwarding_mem_hi_we、forwarding_mem_lo_we、forwarding_mem_hi_result、forwarding_mem_lo_result 表格中的 8 个接口数据等
3. 在 ID 中，进行是否发生 forwarding 的判断：
   i. forwarding_ex_hi_we 或 forwarding_mem_hi_we 为 1，表示前面指令已将 HI 寄存器修改（hi_rdata 发生数据相关）
   ii. 则将 selected_hi_rdata 赋值为相应的 forwarding_ex_hi_result 或是 forwarding_mem_hi_result
   iii. forwarding_ex_lo_we 或 forwarding_mem_lo_we 为 1，表示前面指令已将 LO 寄存器修改（lo_rdata 发生数据相关）
   iv. 则将 selected_lo_rdata 赋值为相应的 forwarding_ex_lo_result 或是 forwarding_mem_lo_result
   v. 否则 selected_hi_rdata=hi_rdata，selected_lo_rdata=lo_rdata

（4）核心代码

核心部分代码（在 ID 段进行判断是否需要使用 forwarding 线路中数值）如下代码框所示：

```
assign selected_hi_rdata = forwarding_ex_hi_we ? forwarding_ex_hi_result
                         : forwarding_mem_hi_we ? forwarding_mem_hi_result
                         : wb_hi_we ? wb_hi_wdata
                         : hi_rdata;

assign selected_lo_rdata = forwarding_ex_lo_we ? forwarding_ex_lo_result
                         : forwarding_mem_lo_we ? forwarding_mem_lo_result
                         : wb_lo_we ? wb_lo_wdata
                         : lo_rdata;
```

5. 移动指令及其实现机制（黄元通）

四条移动指令 inst_mfhi、inst_mflo、inst_mthi、inst_mtlo 功能分别为如下表 2-5-1 所示：

指令	功能
inst_mfhi	将 HI 寄存器的值写入到寄存器 rd 中
inst_mflo	将 LO 寄存器的值写入到寄存器 rd 中
inst_mthi	将寄存器 rs 的值写入到 HI 寄存器中
inst_mtlo	将寄存器 rs 的值写入到 LO 寄存器中

表 2-5-1　各移动指令功能

可观察到移动指令主要由两部分决定：源寄存器和目标寄存器，为实现这两部分的标识，通过以下机制实现。

1. 为实现移动指令源寄存器的确定，通过在 ID 中添加 4 比特位的信号 move_sourse 实现，0~3 比特位依次表示移动指令的源寄存器为 rs、rt、hi、lo
2. 且在 ID 段需将相应目标寄存器的使能信号相应接口置为 1
3. ID_TO_EX_WD 再次加 4，传递至 EX，如果 move_sourse 不为 0，则用相应值更新 EX 段的 hi_result、lo_result 或 ex_result
   a) move_sourse[0]为 1 且 HI 寄存器的写使能信号 hi_we 为 1，表示 rs 寄存器为源寄存器，HI 寄存器为目标寄存器，因此在 EX 段将 HI 的值 hi_result 置为 rs 的值即 rf_rdata1；
   b) move_sourse[0]为 1 且 LO 寄存器的写使能信号 lo_we 为 1，表示 rs 寄存器为源寄存器，LO 寄存器为目标寄存器，因此在 EX 段将 LO 的值 lo_result 置为 rs 的值即 rf_rdata1；
   c) move_sourse[1]为 1 且 HI 寄存器的写使能信号 hi_we 为 1，表示 rt 寄存器为源寄存器，HI 寄存器为目标寄存器，因此在 EX 段将 HI 的值 hi_result 置为 rt 的值即

rf_rdata2；

d) move_sourse[1]为 1 且 LO 寄存器的写使能信号 lo_we 为 1，表示 rt 寄存器为源寄存器，LO 寄存器为目标寄存器，因此在 EX 段将 LO 的值 lo_result 置为 rt 的值即 rf_rdata2；

e) move_sourse[2]为 1，则表示 HI 寄存器为源寄存器，因此在 EX 段将 ex_result 的值置为 HI 的值即 hi_rdata，在 MEM 段由之前已完成的程序完成 ex_result 值赋值为相应的 rs 寄存器、rt 寄存器或 rd 寄存器；

f) move_sourse[3]为 1，则表示 LO 寄存器为源寄存器，因此在 EX 段将 ex_result 的值置为 LO 的值即 lo_rdata，在 MEM 段由之前已完成的程序完成 ex_result 值赋值为相应的 rs 寄存器、rt 寄存器或 rd 寄存器。

信号	功能
move_sourse[0]	为 1 时，表示源寄存器为 rs 寄存器
move_sourse[1]	为 1 时，表示源寄存器为 rt 寄存器
move_sourse[2]	为 1 时，表示源寄存器为 HI 寄存器
move_sourse[3]	为 1 时，表示源寄存器为 LO 寄存器

表 2-5-2 move_sourse 信号各比特位含义

通过此源寄存器区分机制，配合上原有的 MEM 段保存位置机制，加上在（1）HILO 寄存器机制实现 部分加上的 HILO 寄存器实现和保存机制，即可完成运所有移动指令的功能需求。

加上 inst_mfhi、inst_mflo、inst_mthi、inst_mtlo4 条指令后，主要在 ID 段和 EX 段增加了代码。增加的主要功能代码如下：

ID 段

```
// Move
wire [3:0] move_sourse;
// rs to move sourse
assign move_sourse[0] = inst_mthi | inst_mtlo;
// rt to move sourse
assign move_sourse[1] = 1'b0;
// hi to move sourse
assign move_sourse[2] = inst_mfhi;
// lo to move sourse
assign move_sourse[3] = inst_mflo;
……
// store in hi
assign hi_we = inst_mthi;
// store in lo
assign lo_we = inst_mtlo;
```

EX 段

```
assign ex_result =    move_sourse[2] ? hi_rdata
                    : move_sourse[3] ? lo_rdata
                    : alu_result;
assign hi_result =    (move_sourse[0] & hi_we) ? rf_rdata1
                    : (move_sourse[1] & hi_we) ? rf_rdata2
                    : (op_mul_and_div[0]|op_mul_and_div[1]) ?
                    : hi_rdata;

assign lo_result =    (move_sourse[0] & lo_we) ? rf_rdata1
                    : (move_sourse[1] & lo_we) ? rf_rdata2
                    : (op_mul_and_div[0]|op_mul_and_div[1]) ?
                    : hi_rdata;
```

6. 乘法器与除法器接入机制的实现（黄元通）

（1）接口基础

四条移乘法除法指令 inst_div、inst_divu、inst_mult、inst_multu 功能分别为如下表 2-6-1 所示：

指令	功能
inst_div	有符号除法，寄存器 rs 的值除以寄存器 rt 的值，商写入 LO 寄存器中，余数写入 HI 寄存器中
inst_divu	无符号除法，寄存器 rs 的值除以寄存器 rt 的值，商写入 LO 寄存器中，余数写入 HI 寄存器中
inst_mult	有符号乘法，寄存器 rs 的值乘以寄存器 rt 的值，乘积的低半部分和高半部分分别写入 LO 寄存器和 HI 寄存器
inst_multu	无符号乘法，寄存器 rs 的值乘以寄存器 rt 的值，乘积的低半部分和高半部分分别写入 LO 寄存器和 HI 寄存器

表 2-6-1　各移乘法除法指令功能

可观察到乘法除法指令源寄存器都是 rs 寄存器和 rt 寄存器，保存的目标寄存器都是 HI 寄存器和 LO 寄存器，为了实现区别他们 4 条指令，同样可类似 move 移动指令的实现机制：

1. 为实现乘法除法指令类型的确定，通过在 ID 中添加 4 比特位的信号 op_mul_and_div 实现，0~3 比特位依次表示该指令的操作为 inst_mult、inst_multu、inst_div、inst_divu；
2. 且在 ID 段需将相应 HI 寄存器和 LO 寄存器的使能信号 hi_we、lo_we 相应接口置为 1；
3. ID_TO_EX_WD 再次加 4，传递至 EX，如果 op_mul_and_div 不为 0，则表示为相应的乘法或除法功能，相应操作后的 div_result 或 mul_result 相应分段更新 EX 段的 hi_result 和 lo_result。

信号	功能
op_mul_and_div[0]	为 1 时，表示操作为有符号乘法
op_mul_and_div[1]	为 1 时，表示操作为无符号乘法
op_mul_and_div[2]	为 1 时，表示操作为有符号除法
op_mul_and_div[3]	为 1 时，表示操作为无符号除法

表 2-6-2　op_mul_and_div 信号各比特位含义

通过此乘法除法指令区分机制，配合上在 （1）HILO 寄存器机制实现 部分加上的 HILO 寄存器实现和保存机制，即可完成运所有乘法除法指令的接入接口需求。

增加的主要代码如下：

ID 段

```
wire [3:0] op_mul_and_div;
// mult signal
assign op_mul_and_div[0] = inst_mult ;
// multu signal
assign op_mul_and_div[1] = inst_multu;
// div signal
assign op_mul_and_div[2] = inst_div   ;
// divu signal
assign op_mul_and_div[3] = inst_divu ;
……
// store in hi
assign hi_we = inst_div | inst_divu |
inst_mult | inst_multu | inst_mthi;
// store in lo
assign lo_we = inst_div | inst_divu |
inst_mult | inst_multu | inst_mtlo;
```

EX 段

```
assign ex_result =     move_sourse[2] ? hi_rdata
                     : move_sourse[3] ? lo_rdata
                     : alu_result;
assign hi_result =     (move_sourse[0] & hi_we) ? rf_rdata1
                     : (move_sourse[1] & hi_we) ? rf_rdata2
                     : (op_mul_and_div[0]|op_mul_and_div[1]) ?
mul_result[63:32]     // mul's high 32
                     : (op_mul_and_div[2]|op_mul_and_div[3]) ?
div_result[63:32]     // div's remain
                     : hi_rdata;
assign lo_result =     (move_sourse[0] & lo_we) ? rf_rdata1
                     : (move_sourse[1] & lo_we) ? rf_rdata2
                     : (op_mul_and_div[0]|op_mul_and_div[1]) ?
mul_result[31:0]     // mul's low 32
                     : (op_mul_and_div[2]|op_mul_and_div[3]) ?
div_result[31:0]     // div's quotient
                     : hi_rdata;
```

（2）乘法器、乘法器的接入

对于现有的乘法器，还有以下 4 条线路需要连接，即可完成乘法器的接入：

1. 符号标志：mul_signed = op_mul_and_div[0]（inst_mult 指令）
2. 乘法操作数 ina 为：rf_rdata1
3. 乘法操作数 inb 为：rf_rdata2
4. 添加暂停信号：stallreq_for_mul

对于现有的除法器，还有以下 2 条线路需要连接，即可完成乘法除法的接入：

1. assign inst_div = op_mul_and_div[2];
2. assign inst_divu = op_mul_and_div[3];

乘法器和除法器的暂停机制实现在下小节中描述。

7. 乘法器暂停机制的实现（于之晟）

乘法器和除法器的暂停机制实际上是实现流水线执行阶段的暂停机制。

(1).首先需要判断流水线执行阶段即 EX 段当前执行指令是否是乘除法指令；

我们采用的是在流水线译码阶段即 ID 段获取指令信号，如果是乘除法则存到 op_mul_and_div 中，跟随 ID 段到 EX 段的数据通路 id_to_ex_bus 向下传输。

op_mul_and_div	对应指令
op_mul_and_div[0]	inst_mult
op_mul_and_div[1]	inst_multu
op_mul_and_div[2]	inst_div
op_mul_and_div[3]	inst_divu

(2)在流水线执行阶段即 EX 段分别设置线路获取乘法暂停，除法暂停，以及 EX 段暂停信号

stallreq_for_ex = stallreq_for_div | stallreq_for_mul;// 处于流水线执行阶段的指令是否请求暂停

```
stallreq_for_div                    //除法暂停
stallreq_for_mul                    //乘法暂停
```

(3)乘法暂停:

对于乘法暂停，需要做到乘法执行时，EX 段暂停；之后流水线需要正常运行，所以乘法暂停只能执行一个周期。一个周期后需要将乘法暂停信号变为 0。

所以设置了寄存器信号 cnt 和 next_cnt 用来判断乘法暂停的周期长度。

具体细节如下：

周期信号赋值：

```
always @ (posedge clk) begin
        if (rst) begin
            cnt <= 1'b0;
        end
        else begin
            cnt <= next_cnt;
        end
    end
```

暂停信号赋值举例(初值赋为 1'b0)：

```
if((op_mul_and_div[0]|op_mul_and_div[1])&~cnt) begin
            stallreq_for_mul <= 1'b1;
            next_cnt <= 1'b1;
```

首先对于各信号初值设置为 1'b0，当当前指令为乘法指令时，即(op_mul_and_div[0] |

op_mul_and_div[1])成立，且 cnt==1'b0(当前不是暂停周期，上一条指令不是乘法)，需要对 stallreq_for_mul 和 next_cnt 赋值为 1'b1，同时 cnt 获得 next_cnt 的值；
当 cnt==1'b1 时说明执行的指令时乘法指令，之后需要结束暂停，所以需要对 stallreq_for_mul 和 next_cnt 赋值为 1'b0，同时 cnt 获得 next_cnt 的值。
(4)除法暂停:

对于除法暂停，需要做到除法执行时，EX 段暂停；之后流水线需要正常运行，所以除法暂停只能执行 32 个周期。32 个周期后需要将乘法暂停信号变为 0。
(5)流水线执行阶段暂停信号：

```
stallreq_for_ex = stallreq_for_div | stallreq_for_mul;// 处于流水线执行阶段的指令是否请求暂停
```

这个信号会传递到 ctrl.v 中，并作出如下判断：

```
if(stallreq_for_ex == `Stop) begin    //from ex
        stall = `StallBus'b00_1111;
```

(6)stall 信号传递到流水线执行阶段(EX 段):
执行如下操作(主要)：

```
if (stall[2]==`Stop && stall[3]==`NoStop) begin
        id_to_ex_bus_r <= `ID_TO_EX_WD'b0;
    end
    else if (stall[2]==`NoStop) begin
        id_to_ex_bus_r <= id_to_ex_bus;
    end
end
```

首先对 id_to_ex_bus_r 赋初值 0；当 stall[2]==`Stop && stall[3]==`NoStop(即流水线执行阶段需要暂停)，id_to_ex_bus_r 的值仍为 0，流水线无法继续进行；反之当 stall[2]==`NoStop(流水线执行阶段不需要暂停)，将从流水线译码阶段传到流水线执行阶段的值 id_to_ex_bus 赋给 id_to_ex_bus_r，使得流水线正常运行。

三、感受与建议

1. 于之晟

在这次实验中虽然困难很多，但是也是通过各种各样的方法得到了解决，比如询问他人，自己查找参考资料和代码等。在和队友的共同努力下，最终也是完成了一个简易的五级流水，实现了如加减法指令、跳转指令、乘除法指令等多种功能，实现了 forwarding 支路技术、流水线暂停技术、乘法除法机制等众多复杂的模块设计，成功完成了五级流水的不断完善与实现。在实验进行中《动手写 CPU》以及网络上类似的程序给了我很多灵感和启发，例如流水线暂停，访存指令；同时也是通过这样一种方式锻炼了自己的能力。

平心而论，这次的实验难度上手困难，但是熟悉之后还是比较正常的难度水准。也是通过这样一个实验，自己对于五层流水有了一个更加深刻的认识，之前一些只是由文字组成的知识点也成功地在自己的手上实现了出来，比如说流水线暂停等其他功能的实现。我希望之后如果有类似的实验，可以先提供一些简易的上手实验，便于进行过渡，不然对大家的积极性打击挺大的。

2. 黄元通

在和之晟的共同的努力下，我们两人完成了一个基本五级 CPU 的设计与实现，实现了如加减法指令、跳转指令、移动指令、分支指令、移动指令、乘除法指令等多种功能，实现了 forwarding 支路技术、流水线暂停技术、HILO 寄存器机制、移动指令机制、乘法除法机制等众多复杂的模块设计，成功完成了译码阶段、运行阶段、访存阶段、回写阶段和寄存器模块的不断完善与实现，完成了这次感觉是有史以来最难的一次实验课，取得了成功。

在设计时体会到了查阅资料的重要性。因为和之前的知识面涉及差距太大，相关方面的知识储备很少，需要自己学习的新知识太多所有会觉得这次实验的难度很大。比如参考书籍《自己动手写 CPU》就提供了很大帮助，通过对书籍中流水线 RAW 数据相关问题的解决的示例的学习，我们学以致用，加上大家的互帮互助、互相学习，才能顺利从完全无从下手，到敢于尝试并连接好 forwarding 通路。就目前而言个人还是比较满意的，课设完成后也有一定的成就感。

另外，这次是真真正正地领悟到了队员分工合作的极度重要性，这学期 5 门实验课，门门赶着这几天要交实验报告，学校还十分奇怪地要求提前两周完成教学任务，所以这真的是大学目前为止最忙的一学期，如果没有小组成员的分工合作的话真的不知会如何是好，实在万分感谢于之晟组长的付出的超多努力，才能让这次实验如期完成。

四、参考资料

1、张晨曦.《计算机体系结构》（第二版）. 北京：高等教育出版社.2000.

2、雷思磊.《自己动手写 CPU》.2014.