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 3](#_Toc90834108)

[一、概要 3](#_Toc90834109)

[1. 工作量 3](#_Toc90834110)

[2. 指令集 3](#_Toc90834111)

[3. 运行环境及使用工具 3](#_Toc90834112)

[4. 总体设计 4](#_Toc90834113)

[5. 流水段连接图 5](#_Toc90834114)

[二、详细说明 6](#_Toc90834115)

[1. forwarding技术的实现（黄元通） 6](#_Toc90834116)

[（1）理论 6](#_Toc90834117)

[（2）提出问题 6](#_Toc90834118)

[（3）具体实现 7](#_Toc90834119)

[（4）核心代码 8](#_Toc90834120)

[2. ID段流水线暂停机制的实现（于之晟） 9](#_Toc90834121)

[3. load、store访存指令的实现机制（于之晟） 10](#_Toc90834122)

[4. HILO寄存器机制及其forwarding技术的实现（黄元通） 13](#_Toc90834123)

[（1）HILO寄存器机制实现 13](#_Toc90834124)

[（2）HILO寄存器的数据相关问题 14](#_Toc90834125)

[（3）HILO寄存器的forwarding技术实现 15](#_Toc90834126)

[（4）核心代码 17](#_Toc90834127)

[5. 移动指令及其实现机制（黄元通） 17](#_Toc90834128)

[6. 乘法器与除法器接入机制的实现（黄元通） 19](#_Toc90834129)

[（1）接口基础 19](#_Toc90834130)

[（2）乘法器、乘法器的接入 20](#_Toc90834131)

[7. 乘法器暂停机制的实现（于之晟） 21](#_Toc90834132)

[三、感受与建议 23](#_Toc90834133)

[1. 于之晟 23](#_Toc90834134)

[2. 黄元通 23](#_Toc90834135)

[四、参考资料 24](#_Toc90834136)

# CPU搭建

## 一、概要

### 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。

### 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,

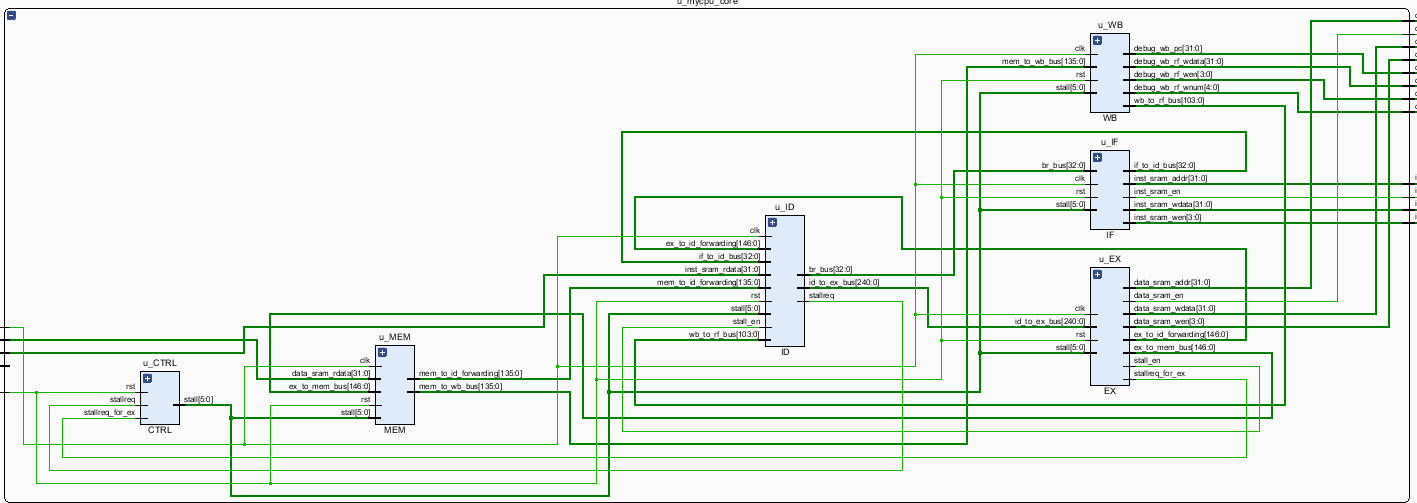
inst\_mflo, inst\_mfhi , inst\_mthi, inst\_mtlo ,

inst\_lb ,  inst\_lbu , inst\_lh  , inst\_lhu  , inst\_sb, inst\_sh

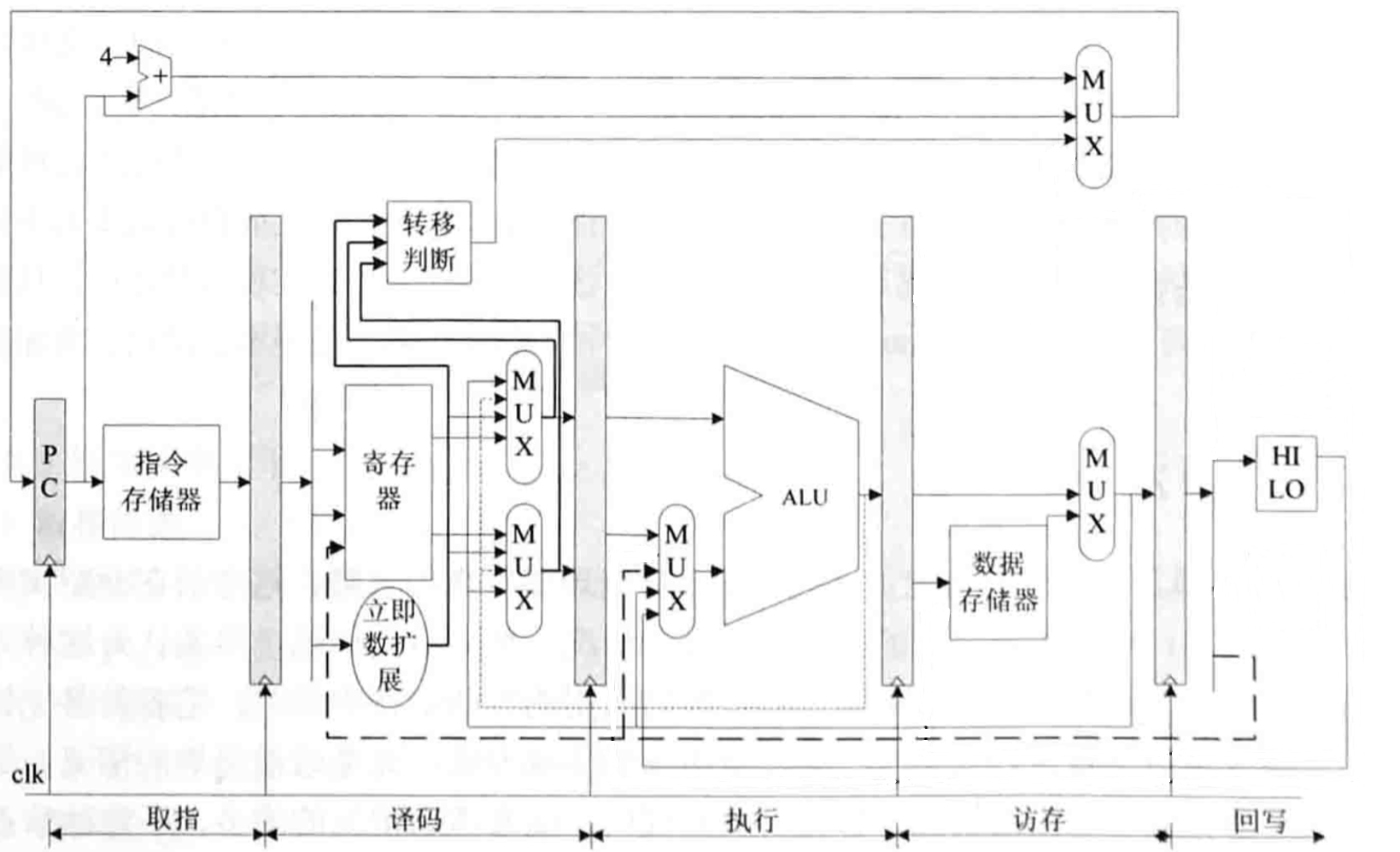
### 运行环境及使用工具

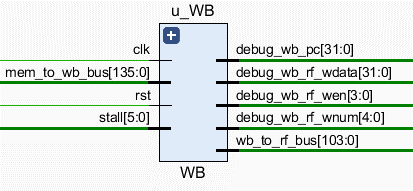
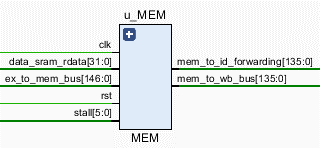
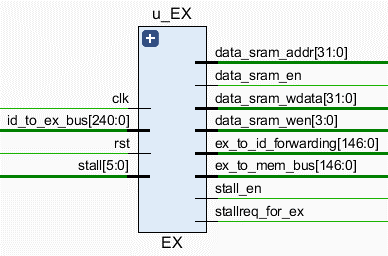
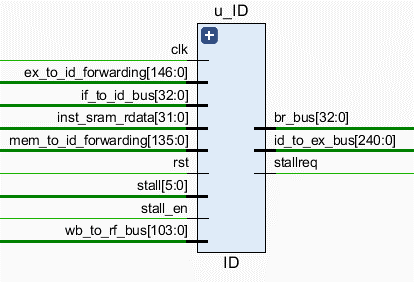
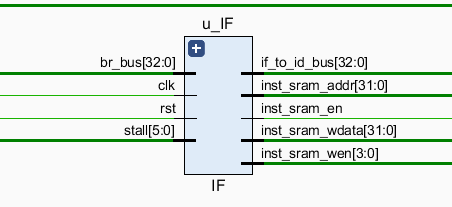
编程语言使用Verilog，实验使用运行在Windows 10系统下的Vivado 2019.2版本软件进行编写和测试运行，以及Microsoft Visual Studio Code 1.63版本软件进行编写。所有文件的编码格式为GB 2312，换行符为CRLF。

### 总体设计



### 流水段连接图





## 二、详细说明

### 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：

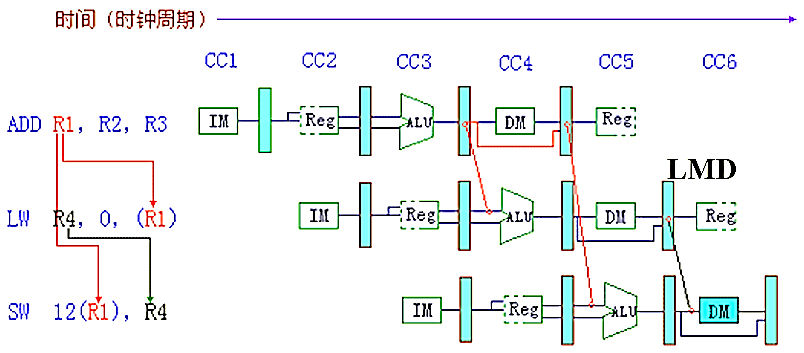




图2-1-1 相邻指令间存在数据相关

#### （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：

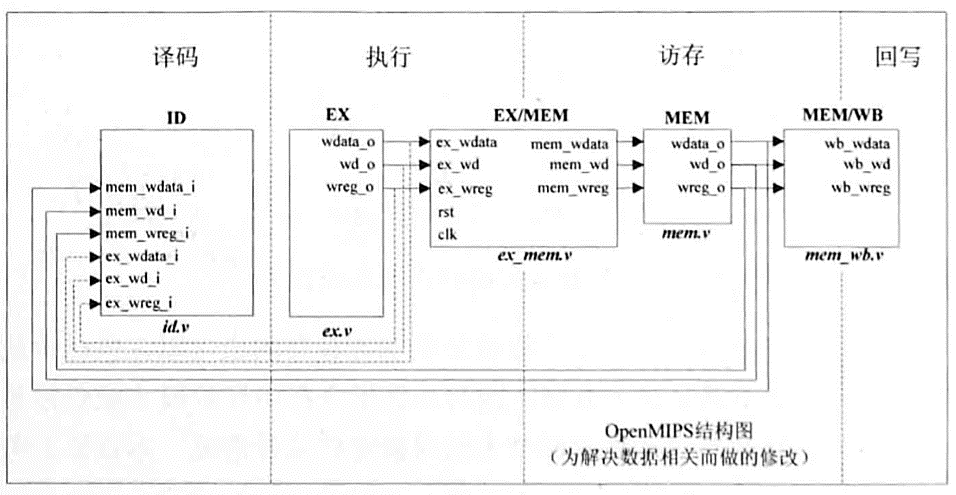


图2-1-2 为实现forwarding对CPU结构做出的修改

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：
   * 1. 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）
     2. 则将selected\_rdata1赋值为相应的ex/mem\_forwarding\_wdata
     3. 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）
     4. 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;

### 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 |

### 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。

### 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）：

    assign id\_to\_ex\_bus = {

        ……

        // hilo\_reg's

        hi\_we,          // 224

        lo\_we,          // 223

        selected\_hi\_rdata,// 191:222

        selected\_lo\_rdata,// 159:190

……

    };

    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;

#### （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。

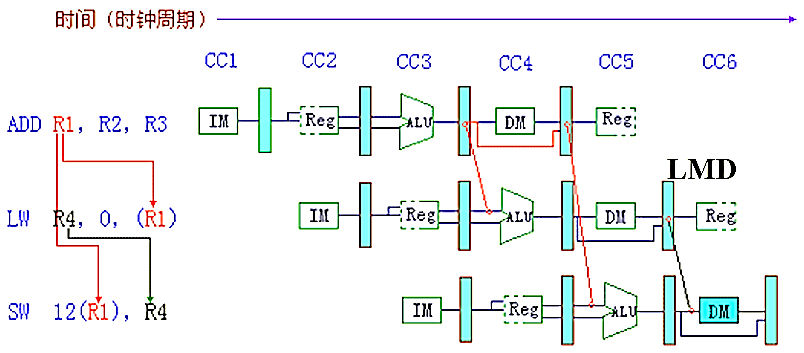




图2-4-1 相邻指令间存在HILO寄存器数据相关

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](#_（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：

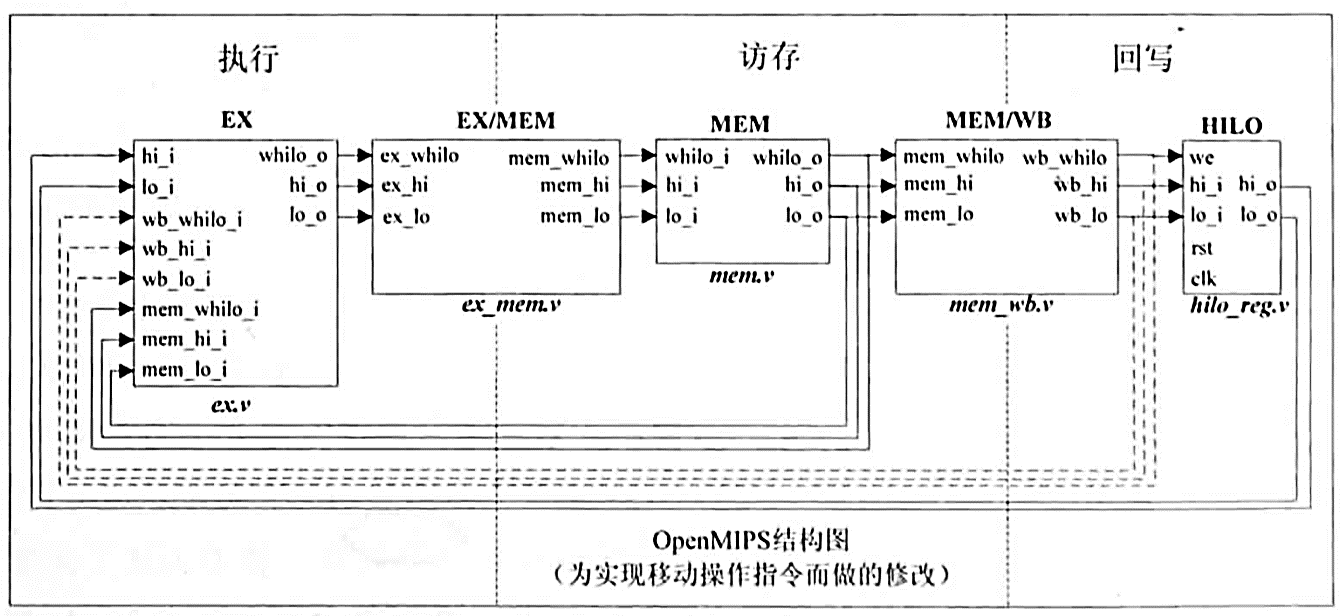


图2-4-2 为实现HILO寄存器的forwarding对CPU结构做出的修改

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：
   * 1. 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）
     2. 则将selected\_hi\_rdata赋值为相应的forwarding\_ex\_hi\_result或是forwarding\_mem\_hi\_result
     3. 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）
     4. 则将selected\_lo\_rdata赋值为相应的forwarding\_ex\_lo\_result或是forwarding\_mem\_lo\_result
     5. 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;

### 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
   1. 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；
   2. 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；
   3. 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；
   4. 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；
   5. 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；
   6. 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](#_（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 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;

// 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;

### 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](#_（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 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;

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;

#### （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。

### 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

### 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。

### 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.