**Lab 6报告**

学号：2020K8009929034、2020K8009929019、2020K8009929037

姓名：胡康、李子恒、吕星宇

箱子号：05

一、实验任务（10%）

在lab5的基础上，完成以下任务：

1. 添加csr系列指令和CSR系列寄存器，使CPU能够和CSR交互；

2. 增加syscall，ertn指令，实现系统调用处理；

3. 增加三条计时器相关的指令；

4. 增加多种异常（取指地址错、地址非对齐、断点、指令不存在、系统调用）与中断（2个软件中断、8个硬件中断、定时器中断）的处理。

二、实验设计（40%）

（一）总体设计思路

1. CSR单独设计，照任务书上推荐设置在WB阶段，实际分析感觉在EXE阶段较优；这样可以规避store指令的冲突，并且不会造成过多的延时。

2. csr指令和现有指令有部分共享通路。

3. 需要增加额外的flush信号来控制整个流水线上的指令流。

4. 需要新增异常信息缓存信号ex\_cause\_bus，合理设计各位的异常信息，并随流水线进入到WB阶段。

5. 需要增加定时器中断的支持，在EXE阶段实现一个独立计数器stable\_cnt。

6. 对于三条计时器相关的指令的数据相关处理按不同情况分开处理：rdcntvl.w与rdcntvh.w两条指令可以在EXE阶段得到数据，故只需采用前递技术处理；rdcntid指令在WB阶段才能得到数据，故需要让后续冲突指令阻塞1拍或2拍后得到rdcntid在WB阶段前递的数据。

（二）重要模块1设计：CSR模块

处理csr系列指令，能够正确写到CSR寄存器中正确的地址，并能够读取。

1. 工作原理

其工作行为实际上和寄存器堆一致，不过由于其内由多种不同功能的寄存器组成，每一位都有特殊的含义，所以需要详细到每一位上写入。

1. 接口定义

表1：csr信号表

| **名称** | **方向** | **位宽** | **功能描述** |
| --- | --- | --- | --- |
| reset | IN | 1 | 重置信号 |
| clk | IN | 1 | 时钟信号 |
| csr\_num | IN | 14 | csr寄存器传入“序号/地址” |
| csr\_rvalue | OUT | 32 | csr读出值 |
| csr\_we | IN | 1 | csr写使能信号 |
| csr\_wmask | IN | 32 | csr写入值的掩码 |
| csr\_wvalue | IN | 32 | csr写入值 |
| ws\_ex | IN | 1 | ws阶段触发例外 |
| ws\_ecode | IN | 6 | ws阶段传入的异常码 |
| ws\_esubcode | IN | 9 | ws阶段传入的异常码（辅助） |
| ws\_pc | IN | 32 | ws传入的pc（ra等） |
| ws\_vaddr | IN | 32 | load、store访问的错误地址 |
| coreid\_in | IN | 32 | TID寄存器的初始化 |
| has\_int | OUT | 1 | 硬件中断信号 |
| ertn | IN | 1 | syscall返回信号 |
| ex\_entry | OUT | 32 | 例外地址输出信号 |
| era\_entry | OUT | 32 | 返回地址输出值 |
| hw\_int\_in | IN | 8 | 硬件中断使能 |
| ipi\_int\_in | IN | 1 | 核间中断使能 |

1. 功能描述

接受csr系列指令，完成value的读出和写入。模块本身并不需要很多思考，主要是参照手册的标准来完成代码，讲义中也有很多代码可以参考。主要倒是一些控制信号需要思索一番，详见后续错误1。

1. 重要模块2设计：异常信息生成模块ex\_cause\_bus

1、工作原理

设置在ID阶段，随流水线向下传递，并在各阶段附着异常信息。

2、接口定义

表2：ex\_cause\_bus各位对应异常表

| **位** | **对应异常** | **产生阶段** | **判断逻辑** |
| --- | --- | --- | --- |
| 0 | INT | ID | 通过读取ESTAT的IS域，判断是否为11个中断之一。 |
| 1 | SYSCALL | ID | 根据指令译码结果，判断是否为syscall指令。 |
| 2 | ADEF | IF | 若PC值末两位非全0，则指令字不对齐，产生该异常。 |
| 3 | ALE | EXE | load/store类指令末两位不是有效访存地址标志时触发该异常。 |
| 4 | BRK | ID | 根据指令译码结果，判断是否为break指令。 |
| 5 | INE | ID | 若取到指令的指令码不对应任何已实现指令，则产生该异常。 |
| 16:6 | (RESERVED) | **/** | **/** |

3、注意事项

没有到WB阶段时，对应bus上某位为Z值表示该异常信息尚未附着上，并非出错。

三、实验过程（50%）

（一）实验流水账

2022.10.12 12：30——22：16 学习csr设计，并完成代码部分

2022.10.13 10：00——11：30 调试exp12，仿真通过

2022.10.14 14：00——20：00 完成exp12部分实验报告的书写

2022.10.22 19：30——23：30 阅读讲义并完成exp13的设计

2022.10.23 19：30——23：40 调试exp13，仿真通过

2022.10.24 8：00 ——11：30 完成exp13部分实验报告的书写

（二）错误记录

1、错误1：信号逻辑+位宽错误合集

（1）错误现象

wb\_rf\_wdata出错：

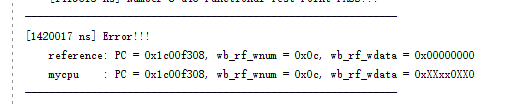


图1：错误1中出错情况截图

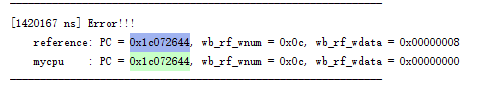


图2：错误1中出错情况截图

（2）分析定位过程

先从pc找到当前指令的行为是什么，然后手动推导其行为，最后自后向前搜查波形问题，找到对与错变化的信号，再从源代码中寻找错误。

（3）错误原因

比如ds\_ready\_go的逻辑错误：



图3：错误1中出错代码截图

应该要对两者取与，而不是或，是两者都不发生时，才是ready\_go，否则就是该阶段被阻塞。

还有一些位宽问题，在csr.v文件中，crmd\_rvalue忘记声明，直接使用assign了，结果只有一位。检查之后estat\_rvalue的位宽也没有声明。

（4）修正效果

位宽修改合适，逻辑重调即可，基本上不需要动脑子的活。

2、错误2：syscall-ertn操作中PC跳转出错

（1）错误现象

Pc跳转出错：

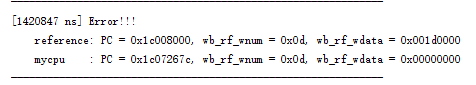


图4：错误2中出错情况截图

（2）分析定位过程

pc出错，只能是syscall了，但是奇怪的是之前有一个syscall却成功了，查看上下代码并结合波形发现其出错原因。

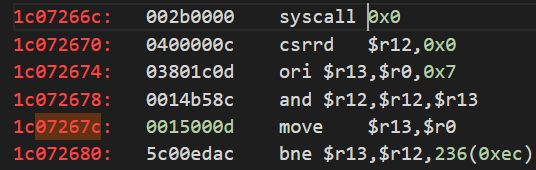


图5：错误2中出错位置指令序列截图

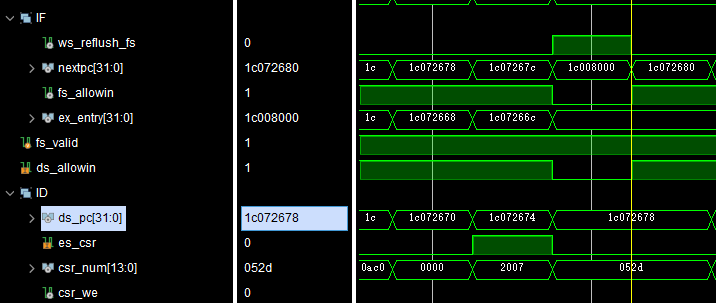


图6：错误2中出错波形截图

（3）错误原因

很明显，在黄线之前的两个周期。es\_csr拉高，导致IF阶段的pc没有传递给ID阶段。其原因是，后面的syscall恰好碰见一个csr指令，结果产生了阻塞导致ds\_ready\_go为0，从而使得IF阶段的pc没能正确地进入ID阶段。在wb提交syscall指令后，中断信号丢失，pc组合地恢复到pc+4的情况，这就造成了错误。

（4）修正效果

图7：错误2中代码修正截图

或上flush信号即可。

3、错误3：syscall-ertn操作中syscall与st指令冲突出错

（1）错误现象

ld结果出错：

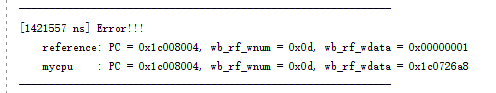


图8：错误3中出错情况截图

（2）分析定位过程

ld指令出错，目前唯一的可能就是之前的store指令出了问题，找到写入对应值的st.w指令：

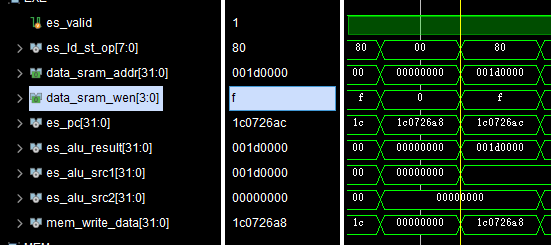


图9：错误3中出错波形截图

这不是这个阶段会出现的bug，所以只能是和syscall之间的问题。查指令发现：

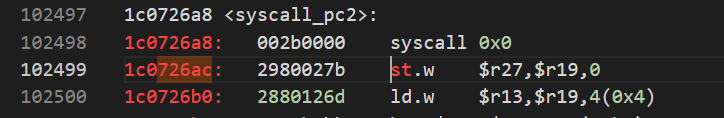


图10：错误3中出错位置指令序列截图

（3）错误原因

只增加了访存信号结果计算模块，而没有相应地修改ID阶段的各个控制信号。

Syscall生效时会flush之前阶段的信号，但是st指令会在es阶段完成操作，这就需要在es和ms之间加一条信号流，从而取消这种相关。

（4）修正效果



图11：错误3中代码修正截图

这样就可以避免这种情况。

4、错误4： 没有禁止异常指令写回

（1）错误现象

如图所示，出错时刻对应一条全为f的指令（0xffffffff），发生指令不存在异常。

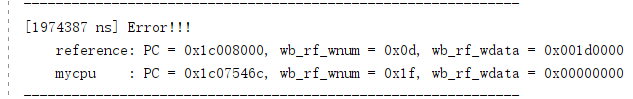


图12：错误4中出错情况截图

（2）分析定位过程

发生指令不存在异常，应当在下一拍跳转到异常处理函数入口地址。从波形上看到的确如此，但在检查波形的同时发现，本条指令意外地置写使能有效。由此确定，异常发生时没有屏蔽写使能信号，造成了写使能有效，向寄存器中存入了错误信息。

（3）错误原因

如图，此时异常处理信号均为正确值，但ws\_gr\_we信号被拉高，导致该条错误指令向寄存器中写入了异常值：

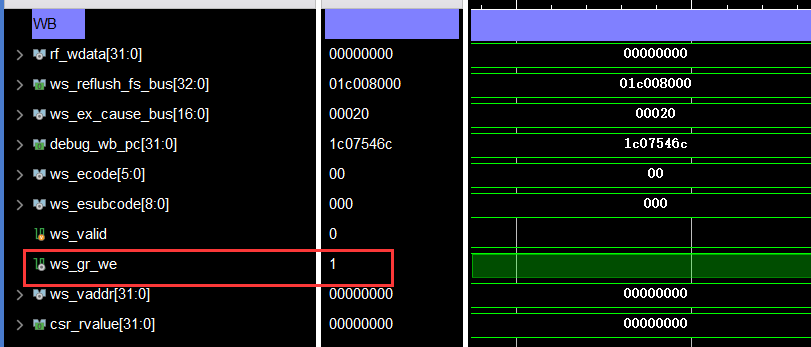


图13：错误4中出错波形截图

（4）修正效果

在EXE阶段，ex\_cause\_bus\_r上已经搭载了所有的异常信息，因此只需要判断ex\_cause\_bus\_r是否存有异常即可，可使用如下代码，在EXE阶段更新gr\_we：

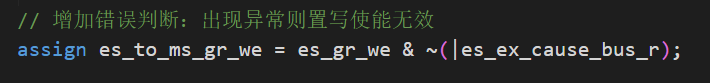


图14：错误4代码修正截图

5、错误5：发生异常时没有禁止store指令向内存中写入数据

（1）错误现象

如图所示，出错时刻对应一条load指令，说明从内存中取出的数据错误。

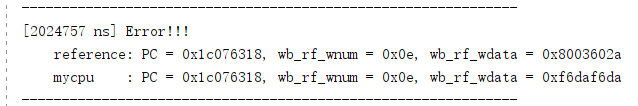


图15：错误5中出错情况截图

（2）分析定位过程

首先确认该条load指令没有发生地址非对齐异常，由比对信号可以看出，向寄存器中写的数据出错了。而该数据来源于内存，因此向前追溯该地址上的读写情况。发现之前有一条store指令向该地址上写过数据。检查发现，该条store指令发生了地址非对齐异常，因此不能在EXE阶段发出写信号。但是它发出了写信号，把一个错误的数据（即图15中的0xf6daf6da）写回到了该地址上，造成后续读出数据错误。

（3）错误原因

如图所示，此时发生了地址非对齐异常，但对应的内存写使能信号仍然被拉高了，导致该条错误指令向内存中写入了异常值：

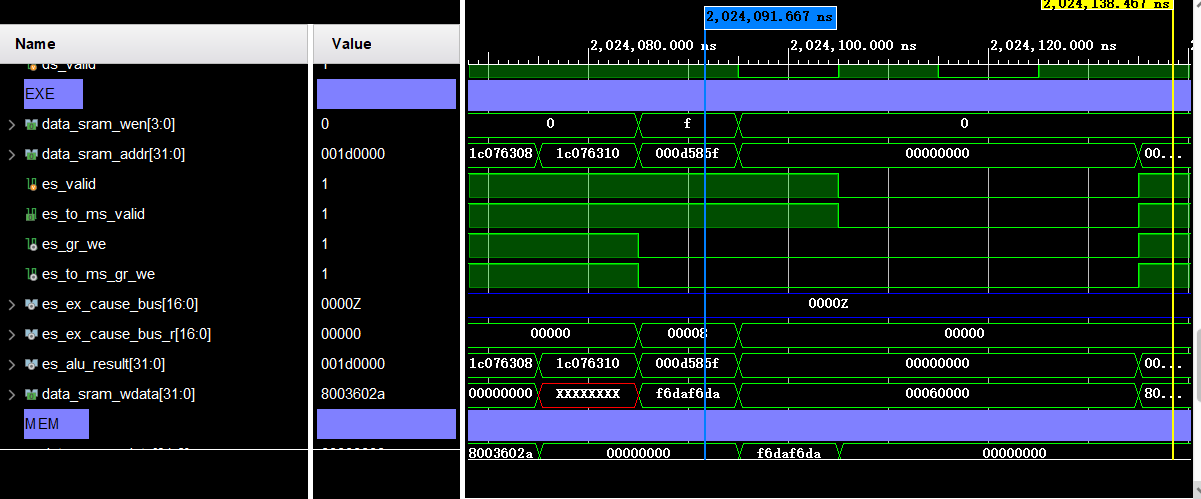


图16：错误5中出错波形截图

（4）修正效果

在EXE阶段，之前添加过一个ms\_int信号，它表示后续的异常要禁止store指令在此处置写使能为有效。可以仿照这个思路，加上一个es\_int信号，它表示本阶段产生了地址非对齐异常：



图17：错误5代码修正截图1

还要更新data\_sram\_wen内存写使能信号：



图18：错误5代码修正截图2

6、错误6：rdcntid指令的写后读冲突

（1）错误现象

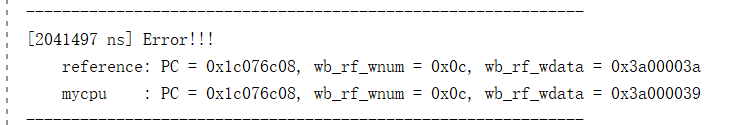
如图19，出错时刻对应一条or指令，or指令的计算结果出现错误。

图19：错误6中出错情况截图

（2）分析定位过程

排除or指令本身的逻辑错误的可能后，可以确定问题根源在于or指令读的某个寄存器之前被写入了错误的数据，因此需要向前找写入这两个读寄存器的指令。向前查看寄存器写入的情况后确定，问题根源在于rdcntid指令与后面的bne指令的写后读冲突。

（3）错误原因

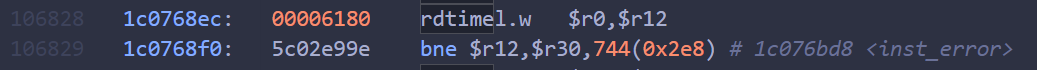
如图20，rdtimel.w（在LoongArch精简版中对应rdcntid指令）指令需要在bne指令前将数据写入12号寄存器，但由于存在写后读冲突，bne指令会先读取12号寄存器原本的错误的数据，进而判断出错误的跳转条件，导致后续执行的指令流发生错误。

图20：错误6中出错位置指令序列截图

（4）修正效果

要修正这一错误，只需要正确实现rdcntid指令的写后读冲突。由于rdcntid指令的读写操作均在WB阶段完成，因此在这里无法通过将数据从EXE阶段递给紧邻的bne指令，这里便不得不采用阻塞+前递的方式（类似load指令阻塞一拍后前递），让后面发生冲突的指令阻塞一拍或两拍。具体方法是修改ds\_ready\_go的赋值逻辑——当当前读指令检测到EXE、MEM阶段的指令有数据相关时便赋值为0。修改完成后，可以看到没有发生错误的跳转，最后一个bug也被解决了！

四、实验总结（可选）

小组合作要搞好分工，另外还要理解好小组使用的代码。

Syscall-ertn系列指令流中，很容易出现由指令顺序造成的bug，属于开始写时只考虑功能，未考虑实际中所有出问题的可能性，这一部分应该由小组成员详细讨论决定。