

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, eretn 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。

2、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

3、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。

（三）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

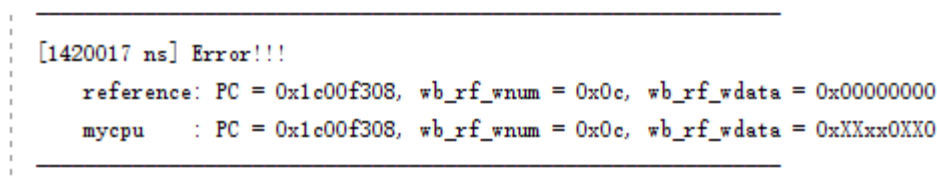
2022.10.22	19: 30——23: 30	阅读讲义并完成 exp13 的设计
2022.10.23	19: 30——23: 40	调试 exp13, 仿真通过
2022.10.24	8: 00 ——11: 30	完成 exp13 部分实验报告的书写

(二) 错误记录

1、错误 1：信号逻辑+位宽错误合集

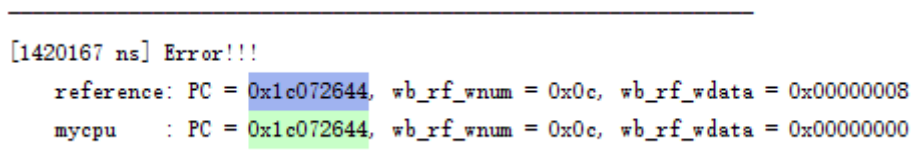
(1) 错误现象

wb_rf_wdata 出错:



```
[1420017 ns] Error!!!
reference: PC = 0x1c00f308, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000000
mycpu : PC = 0x1c00f308, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xXXxx0XX0
```

图 1：错误 1 中出错情况截图



```
[1420167 ns] Error!!!
reference: PC = 0x1c072644, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000008
mycpu : PC = 0x1c072644, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000000
```

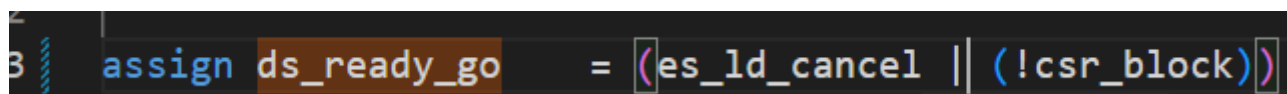
图 2：错误 1 中出错情况截图

(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:



```
assign ds_ready_go = (es_ld_cancel || (!csr_block));
```

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:

```

[1420847 ns] Error!!!
reference: PC = 0x1c008000, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x001d0000
mycpu      : PC = 0x1c07267c, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x00000000

```

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。

```

1c07266c: 002b0000 syscall 0x0
1c072670: 0400000c csrrd $r12,0x0
1c072674: 03801c0d ori $r13,$r0,0x7
1c072678: 0014b58c and $r12,$r12,$r13
1c07267c: 0015000d move $r13,$r0
1c072680: 5c00edac bne $r13,$r12,236(0xec)

```

图 5：错误 2 中出错位置指令序列截图

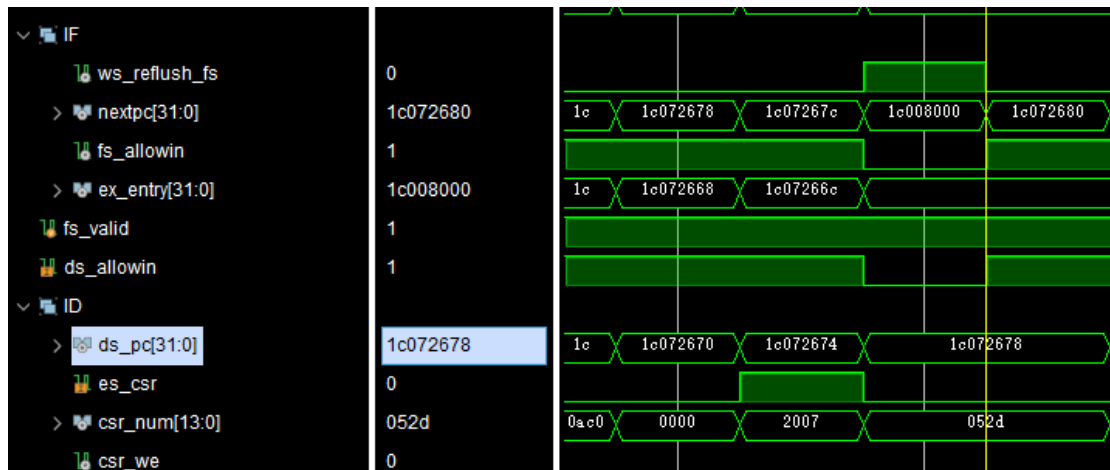


图 6：错误 2 中出错波形截图

（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

```

assign ds_ready_go = (es_ld_cancel & (!csr_block)) | ws_reflush_ds ;

```

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：

```
[1421557 ns] Error!!!  
reference: PC = 0x1c008004, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x00000001  
mycpu    : PC = 0x1c008004, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x1c0726a8
```

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：

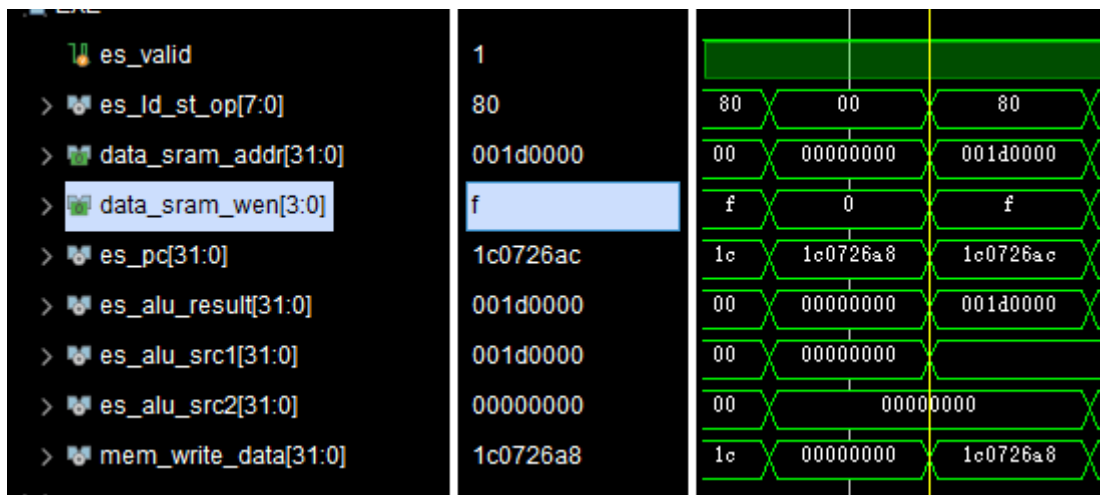


图 9：错误 3 中出错波形截图

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：

```
102497 1c0726a8 <syscall_pc2>:  
102498 1c0726a8: 002b0000 syscall 0x0  
102499 1c0726ac: 2980027b st.w $r27,$r19,0  
102500 1c0726b0: 2880126d ld.w $r13,$r19,4(0x4)
```

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

```
assign data_sram_wen = (es_mem_we && es_valid && !ms_int) ? mem_write_strb : 4'h0;
```

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。

```
[1974387 ns] Error!!!  
reference: PC = 0x1c008000, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x001d0000  
mycpu    : PC = 0x1c07546c, wb_rf_wnum = 0x1f, wb_rf_wdata = 0x00000000
```

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：

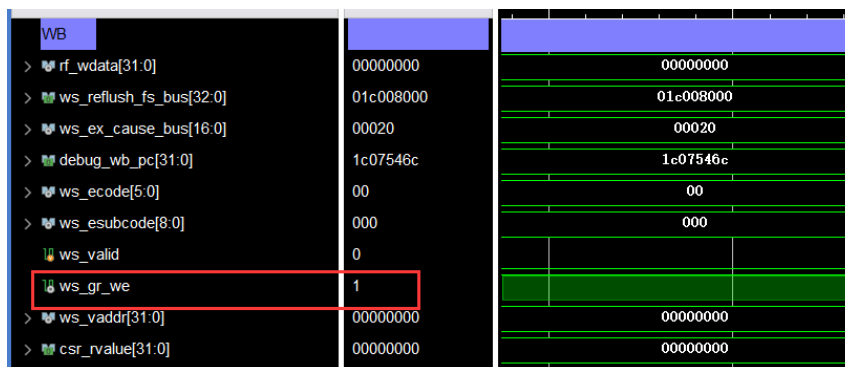


图 13：错误 4 中出错波形截图

(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：

```
// 增加错误判断：出现异常则置写使能无效  
assign es_to_ms_gr_we = es_gr_we & ~(|es_ex_cause_bus_r);
```

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。

```
[2024757 ns] Error!!!  
reference: PC = 0x1c076318, wb_rf_wnum = 0x0e, wb_rf_wdata = 0x8003602a  
mycpu    : PC = 0x1c076318, wb_rf_wnum = 0x0e, wb_rf_wdata = 0xf6daf6da
```

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

(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：

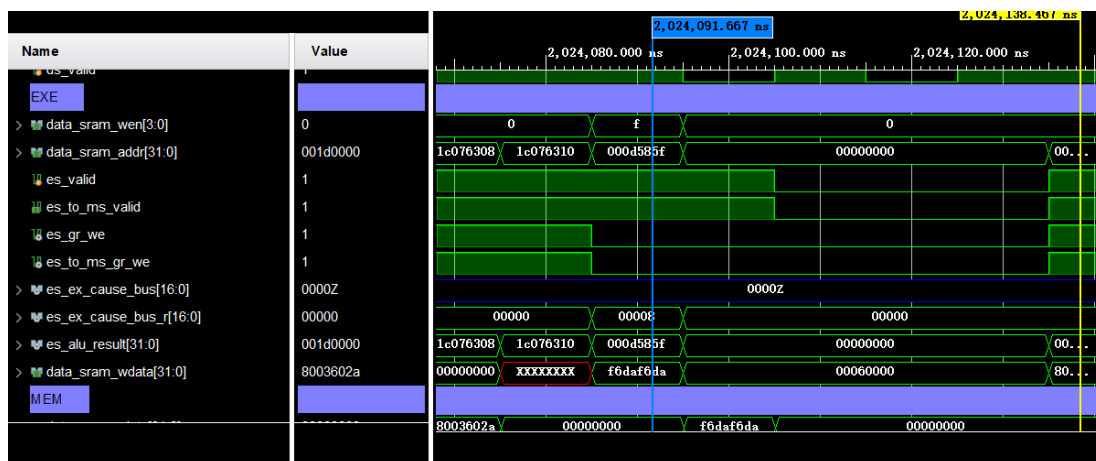


图 16：错误 5 中出错波形截图

(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：

```
assign es_int = es_ex_cause_bus_r[6'h3];
```

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：

```
assign data_sram_wen = (es_mem_we && es_valid && !ms_int && !es_int) ? mem_write_strb : 4'h0;
```

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。

```
-----
[2041497 ns] Error!!!
reference: PC = 0x1c076c08, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x3a00003a
mycpu    : PC = 0x1c076c08, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x3a000039
-----
```

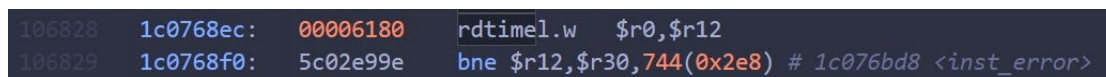
图 19：错误 6 中出错情况截图

（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，rdtime1.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。



```
1c0768ec: 00006180 rdtimel.w $r0,$r12
1c0768f0: 5c02e99e bne $r12,$r30,744(0x2e8) # 1c076bd8 <inst_error>
```

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。