Lab 6 报告

学号 2020K8009908024
姓名 陈卓 史文轩 安金鹏
箱子号 69

一、实验任务（10%）

该实验的目的是在 CPU 中增加异常和中断的支持，包括设计状态控制寄存器，增加各种异常和中断的处理电路，增加 CSR 相关的指令，以及设计相关的阻塞和前递。

二、实验设计（40%）

（一）总体设计思路

该实验中，状态寄存器被单独设计为一个模块（csr.v）。所有读写 csr 模块的行为都被安排在写回级。

预取值模块中，增加了对取指地址错（ADEF）这一例外的判断，并把判断结果随 pc 值传到取值级。

取指模块中，增加了跳转到例外处理地址的功能。当出现异常或中断时，next_pc 的值取 csr_pc 的值，并在下一时钟周期跳转至该地址。

译码模块中，增加了对 csr 相关指令、syscall 指令和 ertn 指令的译码，于实验 13 中又添加了 break 指令、与读计数器有关的指令。由于 csrwr、csrxchg 会写入 rd 寄存器，导产生写后读相关，因此也增加了相关的阻塞部分；同时，对系统调用（SYS）、断点（BRK）、指令不存在（INE）这些例外的表示信号以及从状态寄存器模块传过来的中断信号（INT）都在译码级得到，并不断往后传到写回模块中处理。

执行模块中，首先是增加了对地址非对齐（ALE）这一例外的判断，并不断后传；其次是添加了一个 64bit 的计数器，该计数器用于满足 rdcntvl_w 和 rdcntvh_w 指令的要求，读出计数器中的计数。

写回模块中，由于读写 csr 模块的行为都在这个模块，因此增加了一些向 csr 模块传递的信号；同时，在这个阶段会对例外和中断信号做一些简单的处理和解析，反馈给 csr 模块，比如区分异常的 ecode 和 esubcode 信号。

当执行例外处理地址跳转以及返回异常发生地址时，由于我们将 csr 的读写安排在写回级，因此当异常指令到达协会级时，后面本不应该执行的指令已经进入流水线，因此，我们需要清空整个流水级，使得误取的指令不产生执行效果。

（二）重要模块 1 设计：CSR 寄存器模块

1、工作原理

该模块包含了所有实现的状态寄存器，包括 CRMD、PRMD、ESTAT、ERA、EENTRY、SAVE0~3、ECFG、

BADV、TID、TCFG、TVAL、TICLR。我们没有将各个状态寄存器分散到各流水级，而是集中到一个模块，与其他模块并列，这样不仅设计起来更加简洁，也方便调试。

所有读写该模块的行为都被安排在写回级，这样可以避免产生读写状态寄存器的写后读相关。

在写状态寄存器时，我们以每个状态寄存器的域为单元，对状态寄存器进行修改。

在读状态寄存器时，我们以每个状态寄存器作为一个单元，根据读地址 csr_num 读取相关寄存器的内容。

2、接口定义

信号名称	输入\输出	作用
clk	input	时钟信号
reset	input	重置信号
csr_we	input	状态寄存器写使能信号，在指令为csrwr、csrxchg时有效
csr_num	input	状态寄存器地址
csr_wdata	input	状态寄存器写数据
csr_wmask	input	状态寄存器写掩码
csr_rvalue	output	状态寄存器读数据
wb_ex	input	写回级指令异常信号，在指令出现异常时有效
ertn_flush	input	写回级为 ertn 指令信号
wb_pc	input	写回级 pc，用于传输异常发生地址
csr_pc	output	用于给 nextpc 传输例外入口地址以及返回异常发生地址
wb_ecode	input	例外类型一级编码
wb_esubcode	input	例外类型二级编码
wb_vaddr	input	出现地址相关例外时，出错的虚地址
has_int	output	硬件或软件出现的有效的中断信号

3、功能描述

CRMD 寄存器记录当前特权等级以及当前全局中断使能。

PRMD 寄存器记录发生异常之前 CRMD 寄存器记录的特请点击和全局中断使能。

ESTAT 寄存器的 IS 域记录中断状态，Ecode 和 Esubcode 域用于记录例外类型。

ERA 寄存器记录发生异常时的 PC。

EENTRY 寄存器用于配置除 TLB 重填异常之外的例外处理地址入口和中断入口

SAVE0~3 用于暂存系统软件数据。

ECFG 寄存器记录控制各中断的局部使能位，其 LIE 域有 13 位，每一位控制一个中断源。

BADV 寄存器记录发生 TLB 重填异常和地址错误相关异常时，出错的虚地址。

TID、TCFG、TVAL、TICLR

当发生异常或中断时，相应的标志寄存器会被改动。此外，csrrd、csrwr、csrxchg 指令可以将地址为 csr_num 的状态寄存器的值写入通用寄存器，csrwr、csrxchg 还会将通用寄存器中读出的数据与掩码相与并写入地址为 csr_num 的状态寄存器，实现对状态寄存器的修改。

（三）重要模块 2 设计：阻塞模块

1、工作原理

由于所有读写状态寄存器的行为都在写回级，因此不会产生状态寄存器的写后读相关。

但 csrrd、csrwr、csrxchg 指令直至写回级才会修改通用寄存器，因此仍然会产生通用寄存器的写后读相关。因此，当 csrrd、csrwr、csrxchg 在执行级、访存级、写回级且译码级的指令与之有数据相关时，译码级的指令会被阻塞至上述三条指令离开流水线。

2、接口定义

执行级、访存级、写回级设置信号 inst_csr，判断当前流水级是否为上述三条指令且有效，这三个信号会被传给译码级。传输寄存器写地址 rd 的接口已在前面的实验设置，这里复用就行。

3、功能描述

当执行级、访存级、写回级输入的 inst_csr 有效，且 rd 与 rf_raddr1 或 rf_raddr2 相同时，将 ds_ready_go 信号拉低，使得译码级指令被阻塞。

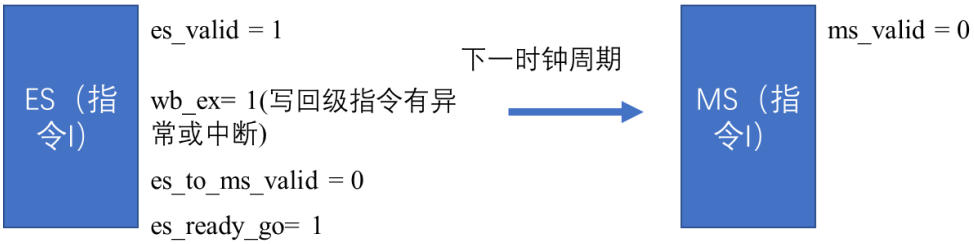
（三）重要模块 3 设计：控制相关的处理

1、工作原理

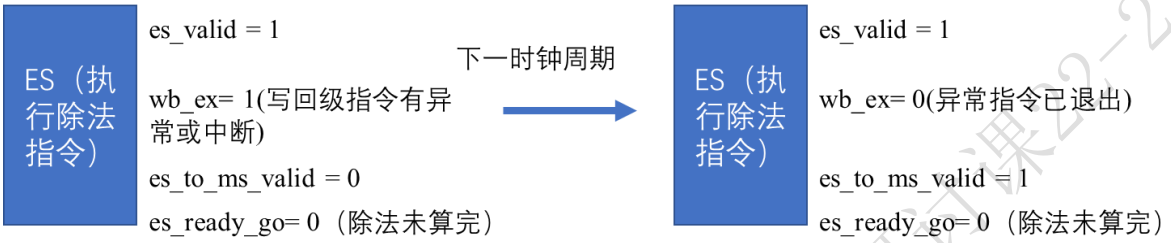
本实验中，在跳转至例外处理入口以及返回异常发生地址时，会发生控制相关。解决方法是清空流水级，使得误取的指令不产生执行效果

2、功能描述

当写回级的指令有异常或中断时，wb_ex 或 ertn_flush 信号拉高。此时取指、译码、执行、访存级的 ready_go 信号被置为 1，xx_to_xx_valid 的信号被置为 0。这样，当进入下一个时钟周期时，译码级、执行级、访存级、写回级的 xx_valid 信号均变为 0，就不会产生执行效果。



将 ready_go 信号置为 1 是因为在写回级指令有异常或中断时，其前面的流水级可能会出现阻塞的情况，如除法指令，以及译码级的写后读相关指令。若不置为 1，则会导致以下情况（以除法指令为例）：



这样，当除法指令算完，进入访存级时，ms_valid 仍为 1，因此到写回级时，除法指令就会将结果写回寄存器，产生执行效果。因此，需要在发生异常或中断时将 ready_go 信号置为 1。

此外，st 指令在访存级已经产生执行效果。因此，当执行级指令为 st 时，需要检查访存级或写回级的指令是否发生异常或中断。若发生，则 data_sram_we 被置为 0，这样数据就不会被写入内存，产生执行效果。

三、实验过程（50%）

（一）实验流水账

10 月 16 日，进行实验 12 的设计。

10 月 17 日和 18 日，进行实验 12 的 debug。

10 月 18 日，仔细阅读了第七章相关部分的讲义。

10 月 19 日，进行了设计和代码书写并开始 debug。

10 月 20 日，实验 13debug 完成。

（二）错误记录

1、错误 1：ds_ready_go 信号错误

（1）错误现象：

PC 出错

```
[1420827 ns] Error!!!
    reference: PC = 0x1c008010, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000002
    mycpu    : PC = 0x1c072770, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x2f00002e
------------------------------------------------------------
```

（2）分析定位过程

相关的汇编指令如下：

1c072668:　29800199　st.w $r25,$r12,0

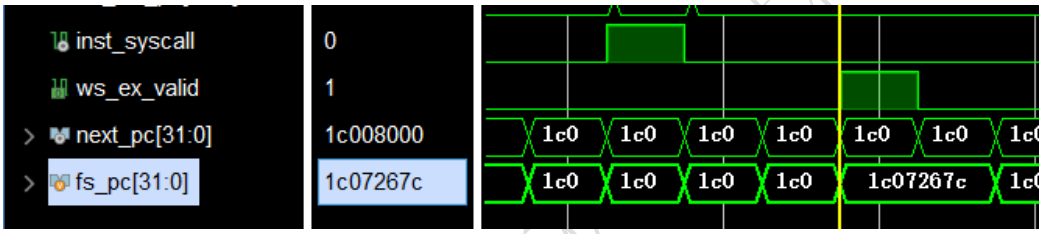
1c07266c:　002b0000　syscall　0x0

1c072670:　0400000c　csrrd　$r12,0x0

1c072674:　03801c0d　ori　$r13,$r0,0x7

1c072678:　0014b58c　and　$r12,$r12,$r13

PC 出错与 syscall 指令有关。



可以看到，syscall 指令到写回级时，next_pc 是正确的地址，但到下一时钟周期时却并未赋给 fs_pc。

fs_pc 赋值如下：

```
always @(posedge clk) begin
    if (reset) begin
        fs_pc <= 32'h_1bfffffc;
    end
    else if (to_fs_valid && fs_allowin) begin
        fs_pc <= next_pc;
    end
```

调出 fs_allowin 信号，发现为 0，因此一开始以为是 allowin 信号出错，做了如下修改：

```
    else if (to_fs_valid && (fs_allowin || wb_ex || wb_ertn_flush)) begin
```

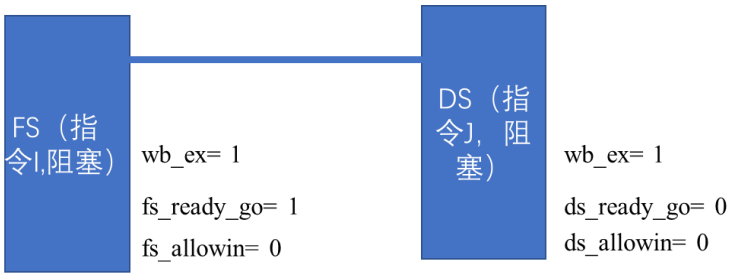
但测试仍未通过。

后来查看汇编代码，发现 csrrd 和 and 有数据相关，csrrd 在访存级，and 在译码级，产生了阻塞。

由 fs_pc 的赋值可知，此时 ds_allowin 为 0，导致 fs_allowin 为 0。因此，next_pc 的值并未传给 fs_pc，导致 pc 错误。

（3）错误原因

当写回级指令为异常或中断时，ds_ready_go 在阻塞的情况下为 0，导致以下情况：



fs_allowin 为 0 导致 fs_pc 不更新，进而导致 pc 出错。

（4）修正效果

在写回级指令异常或中断时，将 ds_ready_go 拉高

修改前：

```
assign ds_ready_go    = !(
    es_blk_valid  &&(es_rf_dest == rf_raddr1 || es_rf_dest == rf_raddr2) ||
    es_csr_blk_valid &&(es_rf_dest == rf_raddr1 || es_rf_dest == rf_raddr2) ||
    ms_csr_blk_valid &&(ms_rf_dest == rf_raddr1 || ms_rf_dest == rf_raddr2) ||
    ws_csr_blk_valid &&(rf_waddr == rf_raddr1 || rf_waddr == rf_raddr2)) ;
```

修改后：

```
    assign ds_ready_go    = !(
    es_blk_valid  && !wb_ertn_flush && !wb_ex && (es_rf_dest == rf_raddr1 || es_rf_dest ==
rf_raddr2) ||
    es_csr_blk_valid && !wb_ertn_flush && !wb_ex && (es_rf_dest == rf_raddr1 || es_rf_dest
== rf_raddr2) ||
    ms_csr_blk_valid && !wb_ertn_flush && !wb_ex && (ms_rf_dest == rf_raddr1 || ms_rf_dest
== rf_raddr2) ||
    ws_csr_blk_valid && !wb_ertn_flush && !wb_ex && (rf_waddr == rf_raddr1 || rf_waddr ==
rf_raddr2)) ;
```

2、错误 2：es_ready_go 信号错误

（1）错误现象：

PC 出错

```
[1423577 ns] Error!!!
    reference: PC = 0x1c008000, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x001d0000
    mycpu    : PC = 0x1c0726f8, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000000
```

（2）分析定位过程

相关的汇编指令如下：

1c0726f4:　002b0000　syscall　0x0

1c0726f8:　0020418c　div.w　$r12,$r12,$r16

1c0726fc:5c00733e　bne　$r25,$r30,112(0x70) # 1c07276c <inst_error>

PC 出错与 syscall 指令有关。

根据上个错误的经验，猜测又是 ready_go 的问题。

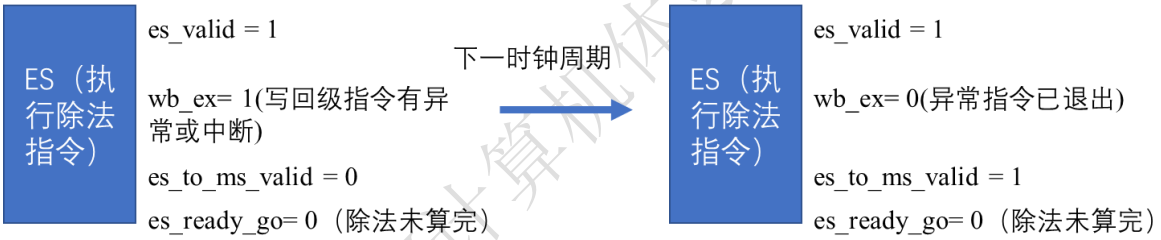
原来的代码如下：

```
assign es_ready_go    =
    (es_inst_div_w|es_inst_mod_w)     ? signed_dout_tvalid   :
    (es_inst_div_wu|es_inst_mod_wu)  ? unsigned_dout_tvalid :
    1'b1;
```

而汇编指令中的除法指令存在阻塞的现象，导致出现错误。

（3）错误原因

当写回级指令为异常或中断时，es_ready_go 在阻塞的情况下为 0，导致以下情况：



此时 fs_allowin 为 0，进而导致 next_pc 的值没有赋给 fs_pc。

（4）修正效果

在写回级指令异常或中断时，将 es_ready_go 拉高

修改后：

```
assign es_ready_go    =
    (es_inst_div_w|es_inst_mod_w) && !wb_ex && !wb_ertn_flush     ? signed_dout_tvalid   :
    (es_inst_div_wu|es_inst_mod_wu) && !wb_ex && !wb_ertn_flush  ? unsigned_dout_tvalid :
    1'b1;
```

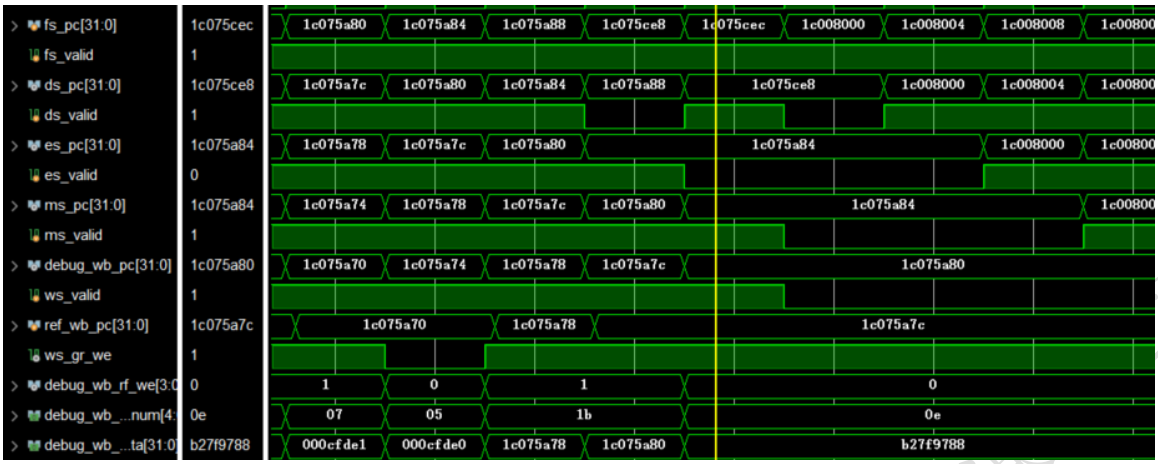
3、错误 3：写回的数据出错

（1）错误现象：

PC 出错，ref 的 PC 信号为 0x1c075a7c，debug_wb_pc 信号为 0x1c075a80.

（2）分析定位过程

我们截取错误的波形如下图：



相关的汇编指令如下图：

```
1c075a6c:   039e20a5    ori $r5,$r5,0x788
1c075a70:   02a11487    addi.w  $r7,$r4,-1979(0x845)
1c075a74:   29a11085    st.w    $r5,$r4,-1980(0x844)
1c075a78:   1c00001b    pcaddu12i   $r27,0
1c075a7c:   0280237b    addi.w  $r27,$r27,8(0x8)
1c075a80:   28a1148e    ld.w    $r14,$r4,-1979(0x845)
1c075a84:   5c02673e    bne $r25,$r30,612(0x264) # 1c075ce8 <inst_error>
```

我们可以看到出错的指令是一条 ld.w 指令，而且由于 ref 的指令在该 ld 指令之前都还是与 debug 信号相同的，我们只能考虑 ld 指令出错的可能性。再在此基础上合理怀疑 ld 在本阶段的写回状态：经查看，我发现 debug_wb_rf_we 信号在 ld 指令处于写回阶段时是显示为 0 的，然而，ws_gr_we 信号却显示为 1，这显然不合理。因此，我查看了这两个信号的赋值代码如下：

```
assign debug_wb_rf_we    = !(ws_ex_valid) & rf_we;
```

```
assign rf_we    = {4{ ws_gr_we && ws_valid }};
```

可以发现错误应该就是在这里，由于我们在实际写回寄存器的时候，译码级接受的写回级的前递信号是 rf_we，因而实际上是写回了，但是该指令本来是不该写回的（由于这里的测试是在 ALE 异常处理的过程中，该条指令被标记了异常），而导致了错误。

（3）错误原因

由于未很好地区分不该写回的地方写回 debug_wb_rf_we 信号和 ws_gr_we 信号，导致不该写回的地方发生了写回。

（4）修正效果

将 rf_we 信号更正为出现异常时为 0（异常不写回）。

```
assign rf_we    = {4{ ws_gr_we && ws_valid && !ws_ex_valid}};
```

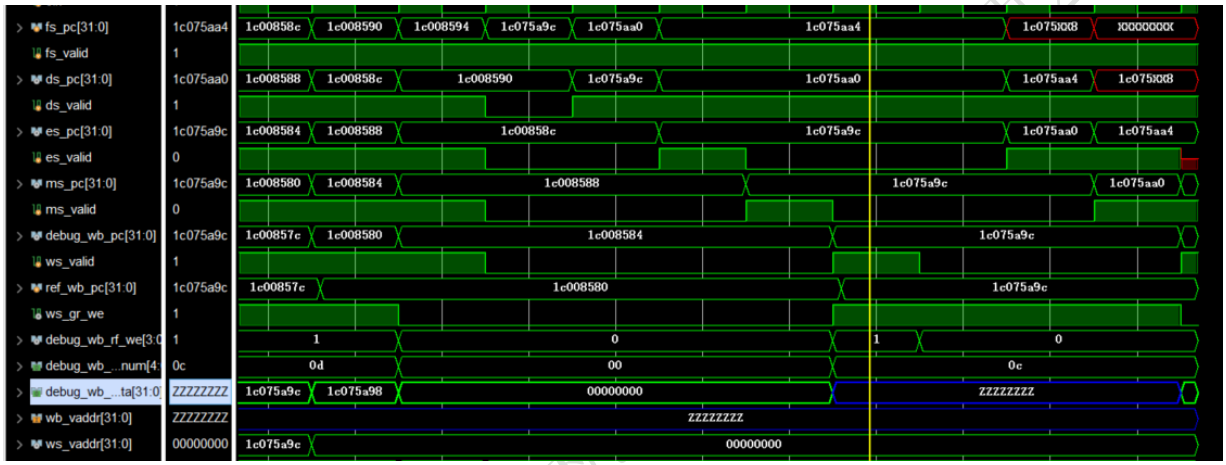
4、错误 4：顶层信号没对应好

（1）错误现象：

写回数据出现 ZZZZZZZZ 的情况。

（2）分析定位过程

我们截取错误的波形如下图：



相关的汇编指令如下图：

```
1c075a8c:   14003a0c    lu12i.w $r12,464(0x1d0)
1c075a90:   03803c19    ori $r25,$r0,0xf
1c075a94:   29800199    st.w    $r25,$r12,0
1c075a98:   002b0000    syscall 0x0
1c075a9c:   04001c0c    csrrd   $r12,0x7
```

我们可以看到出错的指令是一条 csrrd 指令，此时写回的数据应该是从 csr 模块中的某个控制状态寄存器中读取出来的。但是无论如何读取，都不该是 ZZZZZZZZ 这种数据。一般来说， 出现 Z 信号都是由于模块间线没连好造成的。检查后发现是在顶层模块中实例化写回级模块和 csr 模块时变量名字没有对上，从而导致了错误。

（3）错误原因

顶层模块中实例化写回级模块和 csr 模块时变量名字没有对上，使得在写回级接受来自 csr 模块的寄存器数据时实际上没有连到一起，进而出现了高阻信号 Z

（4）修正效果

将模块中的相关部分都改成一个变量名。

四、实验总结（可选）

本次实现了中断和例外等异常的处理，虽然有关控制状态寄存器的大部分代码都在讲义中给出了，但是本次实验由于需要消化理解的内容比较多，实现的细节也比较多，因而还是给我们组带来了不小的麻烦。本次实验的第一部分由陈卓同学完成，第二部分由史文轩同学完成，在我们都仔细阅读讲义之后，陈卓同学给我们组剩余二人仔细讲解了自己实现的内容以及一些细节，总体上来说交接的比较成功，在第二个同学实现自己的部分的时候，并没有因为陈卓同学的代码出现什么 bug。在完成整个实验后，我们组的三位同学又进行了深入的交流，使得组内的每个成员都对实现的内容有比较充分的了解。