

Lab 6 报告

马迪峰 2021K8009929033

饶嘉怡 2021K8009929005

范子墨 2021K8009929006

箱子号 5

一、实验任务（10%）

该实验要理解异常和中断的软硬件协调处理机制。理解精确异常的概念和处理方法，掌握在流水线 CPU 中添加异常和中断支持的方法，在已完成的 CPU 的基础上添加系统调用异常和其他异常与终端支持。

实践 12 需要增加 csr 控制状态寄存器 CRMD、PRMD、ESTAT、ERA、EENTRY 和 SAVE0~3，实现对 csr 的三个读写命令 csrrd、csrwr 和 csrxchg，实现中断返回指令 ertn 和系统调用指令 syscall。

实验 13 要求在实验 12 的基础上增加对取指地址错（ADEF）、地址非对齐（ALE）、断点（BRK）和指令不存在（INE）这几种异常处理的支持，增加对 2 个软中断、8 个硬件中断（实际上并未用到）和定时器中断的支持；在 csr 中增加控制状态寄存器 ECFG、BADV、TID、TCFG、TVAL 和 TICLR。

该实验的正确性同样由基于 trace 比对的调试辅助手段进行验证，通过将 myCPU 中的 WB 阶段的 PC 值与 golden_trace 的 PC 值进行比对，如果不一致则立即报错并且停止仿真，但是并不会检测 csr 模块内的寄存器值，不过可以通过 csr 寄存器访问指令来实现。在仿真正确后进行上板验证。

二、实验设计（40%）

（一）总体设计思路

CPU 发生中断和异常时状态控制信息放置在 csr 寄存器中，因此新增一个 CSR 模块，它包含了所有对 csr 寄存器进行读写的控制电路。该 CSR 模块与 myCPU_top 直接连接，但是通过 WB 模块来传入控制信息和写寄存器值。因此当发生异常或中断后，会将异常信息附着在指令上沿流水线一路携带下去，直至写回级才真正报出异常，此时才会根据携带的信息更新控制状态寄存器。

csrrd、csrwr、csrschg 指令用于软件访问 CSR，需要使用到 rd 寄存器，因此可能会产生数据相关，此时应当修改 ID 阶段阻塞逻辑进行阻塞。

对于 syscall 和 ertn 指令，处理思路类似，在 ID 阶段产生异常信息，并最终流向 WB 模块，WB 模块产生 reflush 信号，清空当前所有流水级的状态，即根据 wb_ex 和 ertn_flush 前递，将每级 valid 信号置 0 并将 nextPC 设

置为异常入口地址。

总的来说检测到异常后的整体处理流程为：拉高异常发生信号，修改 CSR 控制状态寄存器模块，清空流水线并跳转至异常处理程序入口；执行异常处理程序至 ertn 指令退出后，在 ertn 指令运行到 WB 阶段时拉高 wb_ertn 信号，清空流水线，跳转到 ERA 寄存器记录的 PC，并更新 CSR 相关信息（如中断使能等）。

(二) 重要模块 1 设计：csr 模块

1、工作原理

在写回级处理异常，将相关信号传入 CSR 模块。若为 csr 读写指令，则对 csr_num 对应的寄存器进行操作。若异常，则将异常码和出错的地址写入对应寄存器，并在异常处理完成后将异常处理程序的入口传给 pre-IF 级。

2、接口定义

表格 1：csr 模块接口定义

名称	方向	位宽	功能描述
csr_value	OUT	32	csr 寄存器读数据
ex_entry	OUT	32	例外跳转入口地址
ertn_entry	OUT	32	例外处理返回指令
ertn_flush	IN	1	ertn 指令信号
wb_ex	IN	1	例外发生信号
ws2csr_bus	IN	200	包括 csr_re（读使能），csr_we（写使能），csr_num（写地址），csr_wmask（写掩码），csr_wvalue（写数据），wb_pc（例外发生时的 pc），wb_ecode, wb_esubcode, ipi_int_in（核间中断输入），coreid_in（核编号，TID 初值），hw_int_in（硬件中断输入），wb_vaddr 信号
has_int	OUT	1	发生中断信号

3、功能描述

csr 模块包含控制状态寄存器 CRMD、PRMD、ESTAT、ERA、EENTRY、SAVE0~3、ECFG、BADV、TID、TCFG、TVAL、TICLR。每个模块的每个域分别用一个寄存器实现，在读出时再拼接为完整的寄存器，通过多路选择器赋值给 csr_rvalue。每个域在 wmask 中对应的 bit，以及每个寄存器的 csr_num，都用宏在 csr.h 中定义。

CRMD（当前模式信息），用于决定处理器核当前所处的特权等级、全局中断使能和地址翻译模式。其中对于 PLV 域，当触发例外时，该域的值置为 0，以确保陷入后处于最高特权等级。对于 IE 域（当前全局中断使能）。当触发例外时，该域的值置为 0，以确保陷入后屏蔽中断。例外处理程序决定重新开启中断响应时，需显式地将该位置 1。当执行 ERTN 指令从例外处理程序返回时，需要将 PRMD 寄存器中的 PPLV 和 PIE 域分别写入 PLV 和 IE 域。

PRMD（例外前模式信息），当触发例外时，硬件会将此时处理器核的特权等级和全局中断使能位保存至例外前模式信息寄存器中，用于例外返回时恢复处理器核的现场。触发例外时，PRMD 寄存器中的 PPLV 和 PIE 域被赋值成 CRMD 寄存器的 PLV 和 IE 域中的值。

ESTAT（例外状态），该寄存器记录例外的状态信息，包括所触发例外的一二级编码，以及各中断的状态。按照讲义定义对 IS 域、Ecode 域和 EsubCode 域进行赋值即可。

ERA（例外返回地址），当触发例外时，触发例外的指令的 PC 将被记录在该寄存器。

EEENTRY（例外入口地址），该寄存器用于配置除 TLB 重填例外之外的例外和中断的入口地址，只有 VA 域会

被 CSR 指令更新。

SAVE0~3（数据保存），提供给特权软件临时存放数据。

ECFG（例外控制），ECFG 的第 0~9 位和第 11~12 位均为局部中断使能位 LIE，每位控制一个中断源，中断源与 ESTAT 的对应位一一对应，其余位为保留位。对于 LIE 域，其读写由 CSR 指令实现，因此只需对此添加相关支持。其余保留位不允许写入，读出的值为 0，因此直接用 0 填充即可。

BADV（出错虚地址），存储触发地址错误相关例外时出错的虚地址。本次实验只涉及到 ADEF 和 ALE。ADEF 将 wb_pc 写入 BADV，ALE 把访存地址传到 wb 级并写入。

TID（定时器编号），软件配置定时器的接口，从低到高依次为定时器使能位、循环模式控制位、初始值、只读位。需要支持 CSR 指令的读写，并用对应位控制定时器 TVAL。

TVAL（定时器值），TVAL 存储定时器的值，初值为 TCFG 中的 InitVal 域，在 TCFG 的 En 域为 1 时递减。减到 0 后，如果 TCFG 的 Periodic 域为 1，则从初始值开始重新倒计时，否则保持不变。

TICLR（定时中断清除），从该寄存器中读出的值始终为 0，通过对该寄存器最低位写 1 的动作来清除时钟中断标记。因此只需要用一个 1bit 信号 csr_ticlr_clr 来实现该寄存器，并始终赋值为 0，如果要读则在高位拼接上 31bit 的 0。至于清除时钟中断，只需要捕捉对寄存器写 1 的动作，不需要真的写入。

（三）重要模块 2 设计：其他重要模块

1、工作原理

简单概述以下其他模块如何支持异常和中断信号的处理：

对 CPU 发生中断和异常时进行状态控制的信息存放在 csr 寄存器中，这部分内容由电路自动完成，为实现这部分电路，代码中新增的 csr 模块（上文已提到），其内包含了所有的 csr 寄存器以及对 csr 寄存器进行读写的控制电路，并例化在 mycpu_top 文件中，与 WB 流水级直接连接。CPU 在 WB 流水级收集到本条指令在所有流水级中发生的异常或中断信息后，对 csr 模块中的控制状态寄存器进行集中读写。

同时为了实现精确异常，首先需要保证异常指令前面所有的指令全部执行完毕，因此在 WB 级之后进行中断和异常处理。此外，还需要保证异常指令后面“错取”的指令不能改变机器的状态，所以如果 WB 级发现本级指令出现异常时，必须将前面各个流水级清空，即通过向前面各流水级传递 flush 信号来把各级的 valid 标志置 0。另外，如果“错取”的指令包含 store 类型指令，那么该指令在清空各流水级之前已经将数据写入到内存当中，导致机器的状态出现了更改。为了解决这一问题，WB 级和 MEM 级都需要把当前所执行的指令是否发生异常的判断信号传递到 EXE 级，如果 EXE 级发现后面两个流水级中存在出现异常的指令，便将数据 ram 的写使能置 0，防止误写数据破坏了机器的状态。

在 IF 模块中：

需要根据 wb 级传回来的 reflush 信号刷新流水线，并根据从 csr 中读出的异常处理程序入口地址或 ertn 返回地址计算 nextpc。

```

assign nextpc = wb_ex? csr_ex_entry:
                ertn_flush? csr_ertn_entry:
                br_taken ? br_target : seq_pc;

```

同时如果发生例外，需要刷掉流水级中的内容，在 fs_reflush 拉高时拉低 fs_to_ds_valid。等 cpu 跳转到新的地址，fs_reflush 拉低，IF 级中的内容已经被更新，旧有的内容被刷新掉，fs_to_ds_valid 才能再次拉高。同时，因为 fs_to_ds_valid 拉低，ID 级中的 ds_valid 也被拉低，ID 级中的内容也被刷新掉。

在 ID 模块中：

ID 需要对从 IF 级传来的指令以及异常处理相关内容进行译码并生成若干控制信号；将译码信号和控制信号以及异常信息传递给 EXE 模块；ID 中的异常处理部分是本实验 exp12 和 exp13 重点需要设计的部分。ID 需要处理与 csr 有关的读写指令、syscall 指令和 ertn 指令。

对于 csr 读写指令，需要生成对应的读写使能信号 csr_re 和 csr_we 和读写寄存器号 csr_num，并将它们继续向下面的流水级传递，直至 WB 级再进行处理。

对于 syscall 指令，此时需要标记指令出现异常，并将异常信号向下面的流水级传递，直至 WB 级从 EENTRY 中读出中断程序入口地址，再交给 IF 级进行跳转。

对于 ertn 指令，需要 ertn_reflush 信号向下面流水级传递，直至 WB 级从 ERA 中读出返回地址，再交给 IF 级进行跳转。

此外，和在 IF 阶段一样，ID 级需要根据从 WB 级传递来的 reflush 信号，将本周期要向下传递的 valid 信号置 0，同时将 ready_go 置 1，从而达到清空流水线的目的。

与此同时，ID 级不仅要接收来自 IF 级的取地址出错信号和错误的取地址，还要处理断点（BRK）、指令不存在（INE）的异常以及外部中断。

这部分控制逻辑如下：

```

assign ds_csr_re    = inst_csrrd | inst_csrwr | inst_csrxchg | inst_rdcntid; //读使能信号
assign ds_csr_we    = inst_csrwr | inst_csrxchg; //写使能信号
assign ds_csr_wmask = {32{inst_csrxchg}} & rj_value | {32{inst_csrwr}};
assign csr_wvalue   = rkd_value;
assign ds_csr_num   = {14{inst_rdcntid}} & `CSR_TID | {14{~inst_rdcntid}} & ds_inst[23:10];

assign ds_ertn_flush = ds_valid & inst_ertn;
assign ds_ex = ds_valid & (inst_syscall | inst_break | ds_ine | ds_has_int | ds_ade);

assign ds_icode = ds_has_int    ? `ECODE_INT
                  : ds_ade      ? `ECODE_ADE
                  : ds_ine      ? `ECODE_INE
                  : inst_break   ? `ECODE_BRK
                  : inst_syscall ? `ECODE_SYS : 6'b0;

```

```
assign ds_esubcode = ds_adeft ? `ESUBCODE_ADEF : 9'b0;
```

在 EXE 模块中:

EXE 模块中需要根据 wb 级传回来的 reflush 信号刷新流水线, 以及根据从 ms 传回的 ex_from_ms 拉低 es_mem_we, 以避免在异常或 etrn 指令后的指令向内存中写入数据。此外, 还需要检测 EXE 是否产生了 ALE 异常。

EXE 级中比较重要的是需要完成 load 和 store 指令的访存地址异常检测。如果读写对象是 word, 则地址后两位必须为全 0; 如果读写对象是 half word, 则地址末尾必须是 0。如果不是 0, 则拉高 ld_ale 或 st_ale 信号, 表示存在异常。检测到异常后会将 ex 拉高, 将其和出错的访存地址一并传入后续流水级。

```
assign ld_ale = es_load_op[1] & es_alu_result[0] // inst_ld_h
              | es_load_op[2] & (es_alu_result[1] | es_alu_result[0]) // inst_ld_w
              | es_load_op[4] & es_alu_result[0]; // inst_ld_hu
assign st_ale = es_store_op[1] & es_alu_result[0] // inst_st_h
              | es_store_op[2] & (es_alu_result[1] | es_alu_result[0]); // inst_st_w
assign es_ale = ld_ale | st_ale;
```

在 MEM 模块中:

MEM 模块在目前的设计中和 WB 模块一样并不会产生新的中断, 因此实现比较简单, 只需要根据 EXE 模块传递来的数据执行相应的访存操作。并将访存指令结果、写回控制信号、PC 等信息在下一个时钟上升沿传递给 WB 模块。但是 MEM 级需要利用从 EXE 级传来的 exc_data, 生成异常信号 ms_to_es_ex 和包含所有异常信息的 exc_data, 并分别传递到 EXE 级和 WB 级。其中 ms_to_es_ex 的作用是告知 EXE 级前面有指令出现异常, 防止 EXE 级的 store 指令将数据写入数据 ram 中, 从而实现精确异常。此外, MEM 级会根据从 WB 级传来的 reflush 信号进行流水线的清空。

```
assign ms_ex_to_es = (mem_ertn_flush | mem_ex) & ms_valid;
```

在 WB 模块中:

WB 模块比较重要, 它需要将寄存器堆写信号和数据传回 ID 阶段执行写寄存器的操作, 同时还需要处理前面流水级传来的异常和中断信号, 对 CSR 控制寄存器进行读写, 然后 CSR 模块将中断处理程序入口地址或返回地址传回 IF 级, WB 模块发出清空流水线的信号。

WB 实现的主要是实现 CSR 的控制逻辑, 之后 CSR 模块会将它的输出传送给各个模块。

WB 模块从 CSR 模块读取的数据进行写回操作, 即将有关信息传递到 ID 模块中的寄存器堆中。

CSR 直接将中断信号 has_int 传递到 ID 模块, 从而对译码级指令进行中断标记。

```
wire ws_csr_re;
wire ws_csr_we;
wire [13:0] ws_csr_num;
wire [31:0] ws_csr_wmask;
```



```

wire [31:0] ws_csr_wvalue;
wire [ 5:0] ws_ecode;
wire [ 8:0] ws_esubcode;
wire [31: 0]ws_wrong_addr;
wire [31: 0]ws_rkd_value;
// exp13
wire [ 7:0] ws_hw_int_in  = 8'b0 ;
wire          ws_ipi_int_in = 1'b0 ;
wire [31:0] ws_coreid_in  = 32'b0;

assign {ws_wrong_addr,
        ws_csr_we,
        ws_csr_wmask,
        ws_csr_num,
        ws_ertn_flush,
        ws_ex,
        ws_esubcode,
        ws_ecode,
        ws_csr_re
        } = ws_exc_data & {`MS_EXC_DATA_WD {ws_valid}}; // wb_ex=inst_syscall,
ertn_flush=inst_ertn

```

三、实验过程（50%）

（一）实验流水账

exp12 实验流水账

- 2023/10/24 22:00 exp12 实验完成，开始 debug
- 2023/10/25 20:00 exp12 仿真通过，完成 exp12
- 2023/10/26 17:00 exp12 上板验证出现问题，检测错误原因
- 2023/10/26 21:00 找出 exp12 上板错误原因并修正，输入输出端口反向连接导致错误，上板测试通过

exp13 实验流水账

- 2023/11/1 15:00 exp13 实验开始
- 2023/11/1 22:00 exp13 代码部分结束，开始调试
- 2023/11/3 12:00 exp13 仿真上板通过，exp13 结束，开始撰写实验报告
- 2023/11/6 21:00 exp13 实验报告撰写完毕，准备开始 exp14 实验

（二）错误记录

1、错误 1：第二个操作数从 rk 和 rd 里选择错误

(1) 错误现象

[1357407 ns] Error!!!

reference: PC = 0x1c010288, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000000

mycpu : PC = 0x1c010288, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xbfaff080

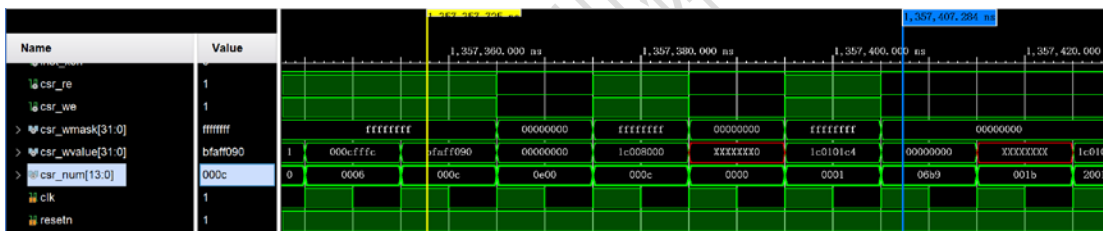
写回寄存器的数据错误。

(2) 分析定位过程

经查找，出错的指令是 `csrwr`，该指令是将指定的 CSR 寄存器的旧值写回 `rd` 寄存器。在排除了指令译码出错的可能性后，就只可能是在上一个 `SCR` 写指令时，将错误的值写入了这个 CSR 寄存器（或者将正确的值写入了错误的其他的 CSR 寄存器，导致这个 CSR 寄存器里的值没有更新从而出错）。

```
1c010278: 04000420  csrwr  $r0,0x1
1c01027c: 04001820  csrwr  $r0,0x6
1c010280: 04003020  csrwr  $r0,0xc
1c010284: 1438010c  lu12i.w $r12,114696(0x1c008)
1c010288: 0400302c  csrwr  $r12,0xc
1c01028c: 4c000020  jirl   $r0,$r1,0
```

查看反汇编代码，发现 1c010288 的上一个 CSR 写指令是 1c010280。从汇编指令可以看出，它的 CSR 寄存器字段与 1c010288 相同，都是 0xc，也就是 EENTRY 寄存器。



从波形中也可以印证这一点，黄色光标是上一个 `csrwr` 的 ID，往后两拍是这个 `csrwr` 的 ID，二者的 CSR 地址一样，都是 0xc。

由于之前出现过类似的情况，因此很容易就想到，CSR 读写指令的第二个操作数是 `rd` 中取来的，但没有更新 `src_reg_is_rd`。于是在 `src_reg_is_rd` 中添加了 `csr_re` 的情况，也就是 CSR 读写指令。

```
assign src_reg_is_rd = inst_beq | inst_bne | inst_blt | inst_bge | inst_bltu | inst_bgeu | (store_op) | csr_re;
```

(3) 错误原因

CSR 读写指令的第二个操作数是 `rd` 中取来的，但没有在 `src_reg_is_rd` 添加 CSR 读写指令的情况。导致在出错指令的上一个指令执行时，错误地把 `rk` 而不是 `rd` 寄存器里的数据写入了 EENTRY 寄存器。该指令执行时，就从 EENTRY 寄存器里读回了错误的值。

(4) 修正效果

在 `src_reg_is_rd` 中添加 CSR 读写指令的情况后，不再出错。

(5) 归纳总结（可选）

在新增指令时，除了新增功能之外，也一定要考虑到是否需要修改之前的逻辑。

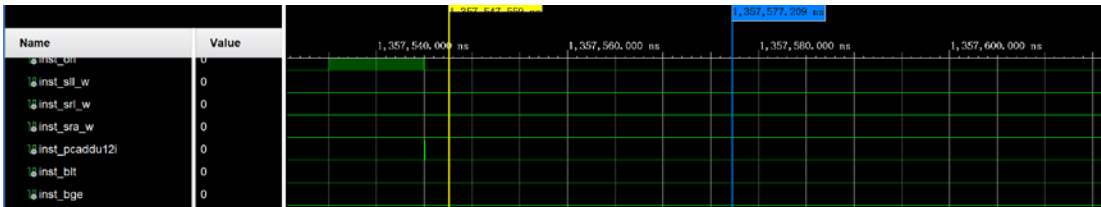
2、错误 2：历史遗留错误之 LIU12I.W 指令

(1) 错误现象

```
[1357577 ns] Error!!!  
reference: PC = 0x1c071cd8, wb_rf_wnum = 0x1b, wb_rf_wdata = 0x1c071cd8  
mycpu      : PC = 0x1c071cd8, wb_rf_wnum = 0x1b, wb_rf_wdata = 0x00000000
```

写回寄存器的数据错误。

(2) 分析定位过程



查看反汇编代码，得知出错的指令是 LIU12.W，但在波形上回到第二拍，却发现没有任何一个指令拉高，inst_lu12i_w 也没有拉高。翻看 inst_lu12i_w 的代码，发现它使用的是 inst 而不是 inst_reg，也就是错误地使用了此时出于 IF 阶段的指令去译码。将 inst 改成 inst_reg 修改如下。

```
assign inst_lu12i_w= op_31_26_d[6'h05] & ~inst_reg[25];
```

反思一下之前为什么没出错，猜测是因为测试文件里所有的当拍的 inst[25]恰好和 inst_reg[25]一样，导致这个错误一直隐藏。

(3) 错误原因

inst_lu12i_w 使用的是 inst 而不是 inst_reg，也就是错误地使用了此时出于 IF 阶段的指令去译码。这导致这一拍没有任何指令执行，所有的控制信号拉低，因此与或逻辑的 final_result 为零。最后选择写回的数据时使用的是三目运算符，同样由于控制信号拉低，最后选中了全零的 final_result_reg，导致写回的数据全零。

(4) 修正效果

将 inst 改成 inst_reg 后，不再出错。

3、错误 3：冲突时的阻塞逻辑考虑不全

(1) 错误现象

```
[1357747 ns] Error!!!  
reference: PC = 0x1c008090, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x000b0000  
mycpu      : PC = 0x1c00f060, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x1c071ce0
```

PC 错误，根据经验一般是跳转指令出错。

(2) 分析定位过程



蓝色光标为报错的位置，报错原因是从 PC 开始就和金标准不相符。可以看到蓝色光标的上一拍，debug_wb_rf_we 为 f，是拉高的，也就是说这一拍会和金标准做对比。这一拍的 PC 是 1c00f050，金标准这一拍的 PC 是 1c008088，发现在这一拍就已经出错了，不知道出于什么原因没有报错。

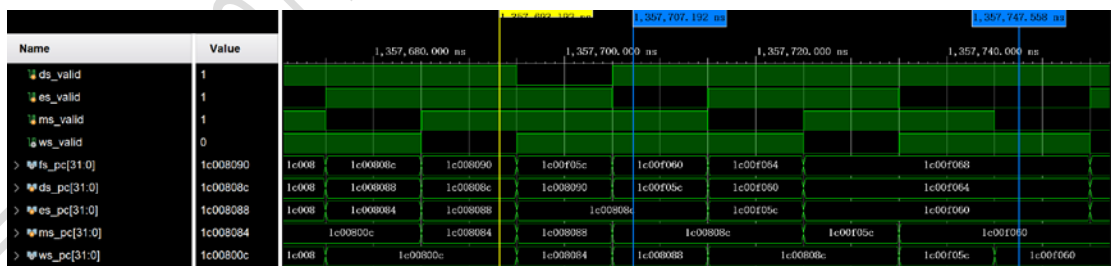
再往前追溯。

再上拍 1c008088 是 CSR RD，紧跟在跳转指令后面被取消了。

再上拍 1c00808c 是 BNE，看来是跳转错了。这里从波形来看，br_taken 拉高，实际执行了跳转，且经计算跳转地址看样子是对的。那么有可能是这里实际上不该执行跳转，判断条件错误，但通过检查 rj 和 rd 寄存器数值的大小，也发现没有错误。

```
1 1c071cdc 1b 1c071ce0
1 1c008000 0d 001d0000
1 1c008004 0d 00000001
1 1c008008 0c 00000001
1 1c008084 19 00000000
1 1c008088 0c 1c071ce0
1 1c008090 0c 000b0000
1 1c008094 0c 0000000b
```

查看金标准文件，按照金标准的意思不应该跳转，那么一定是判断条件错误。



再查看波形，黄色光标时 ID 的 BNE 跳转指令和 EXE 的 CSR RD 指令冲突了，理论上是要阻塞 BNE，但没阻塞，而是直接前递了。

```

assign ds_ready_go  = ~(((exe_rf_we && (es_block | es_csr_re) &&
    (exe_dest == rf_raddr1 && rf_raddr1 && (~src1_is_pc & ~inst_lu12i_w & (alu_op) | (mul_op)) || //
    exe_dest == rf_raddr2 && rf_raddr2 && (~src2_is_imm & (alu_op) | (mul_op)))) ||
    mem_rf_we && (ms_csr_re) &&
    (mem_dest == rf_raddr1 && rf_raddr1 && (~src1_is_pc & ~inst_lu12i_w & (alu_op) | (mul_op)) || //
    mem_dest == rf_raddr2 && rf_raddr2 && (~src2_is_imm & (alu_op) | (mul_op))
    ));

```

查看代码中的阻塞逻辑，发现是由于与上了(alu_op)，因此没考虑到 BNE 等六条条件跳转指令。

```

wire ds_ready_go;
wire exe_conflict_rj;
wire exe_conflict_rkd;
wire mem_conflict_rj;
wire mem_conflict_rkd;
wire branch;

assign branch = inst_beq | inst_bne | inst_blt | inst_bge | inst_bltu | inst_bgeu;
assign exe_conflict_rj = exe_dest == rf_raddr1 && rf_raddr1 && (~src1_is_pc & ~inst_lu12i_w & (alu_op) | (mul_op) | csr_re | branch);
assign exe_conflict_rkd = exe_dest == rf_raddr2 && rf_raddr2 && (~src2_is_imm & (alu_op) | (mul_op) | csr_re | branch);
assign mem_conflict_rj = mem_dest == rf_raddr1 && rf_raddr1 && (~src1_is_pc & ~inst_lu12i_w & (alu_op) | (mul_op) | csr_re | branch);
assign mem_conflict_rkd = mem_dest == rf_raddr2 && rf_raddr2 && (~src2_is_imm & (alu_op) | (mul_op) | csr_re | branch);

assign ds_ready_go  = ~(
    exe_rf_we && (es_block | es_csr_re) && (exe_conflict_rj || exe_conflict_rkd) ||
    mem_rf_we && (ms_csr_re) && (mem_conflict_rj || mem_conflict_rkd)
);

```

于是修改代码，定义变量 branch，表示那六条条件跳转指令，并将其加入阻塞逻辑。

同时还注意到，阻塞逻辑中没有考虑到 CSR 读写指令，于是将 csr_re 一并加入阻塞逻辑。

之前之所以没出错，猜测是因为之前的测试文件里全都没有 ld 指令后面紧跟 branch 的情况。现在新出现了 CSR 读写指令后面紧跟 branch 的情况，暴露了之前的问题。

(3) 错误原因

阻塞逻辑(ds_ready_go)中没考虑到 BNE 等六条条件跳转指令，导致 BNE 跳转指令和 EXE 的 csrrd 指令冲突时，理论上是要阻塞 BNE，但实际上没阻塞，而是直接前递了。

(4) 修正效果

将 branch 加入阻塞逻辑之后，不再出错。并最终通过 exp12 的仿真。

```

=====
Test end!
---PASS!!!
$finish called at time : 1363385 ns : File "D:/CA_local_depository/exp12/mycpu_env/soc_verify/soc_bram/testbench/mycpu_tb.v" Line 270
run: Time (s): cpu = 00:11:51 ; elapsed = 00:12:15 . Memory (MB): peak = 2105.645 ; gain = 46.566

```

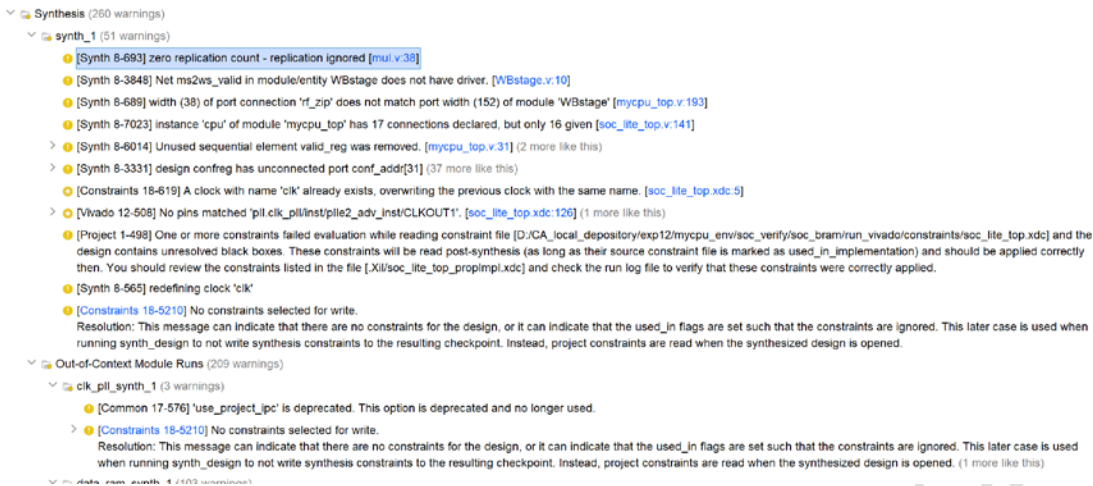
4、错误 4：仿真通过，上板不过

(1) 错误现象



红灯全亮。

(2) 分析定位过程



打开综合报告，逐一排查错误。再次生成比特流后，上板通过。虽然修改了若干个错误，但怀疑导致上板失败的是因为其中的一个无驱动错误。

❗ [Synth 8-3848] Net ms2ws_valid in module/entity WBstage does not have driver. [WBstage.v:10]

报错显示 WB 模块里的 ms2ws_valid 信号无驱动。

```
input wire ms_allowin,  
output wire ws_allowin,  
input wire es2ms_valid,  
output wire ms2ws_valid,
```

发现是因为误将 input 的信号 ms2ws_valid 写成了 output。同时还有一些其他错误，将这一段修改如下。

```
output wire ws_allowin,  
input wire ms2ws_valid,
```

(3) 错误原因

猜测是因为误将 WB 模块中的 input 的信号 ms2ws_valid 写成了 output。

(4) 修正效果

修改过后，重新生成比特流，上板通过。

以上为 exp12 出现的 bug,下图是为 exp12 综合实现的时序结果。相比于上次实验而言，奇怪的是，它的时序居然变得很好了。Vivado 真奇怪(> ~ <)

5、错误 5：赋值位数错误

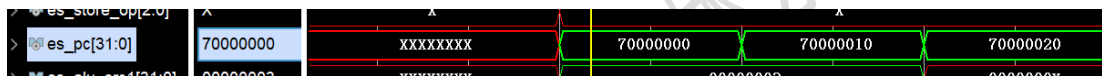
Test begin!

```
[ 2057 ns] Error!!!
reference: PC = 0x1c000000, wb_rf_wnum = 0x0c, wb_rf_wdata = 0xffffffff
mycpu    : PC = 0x00000000, wb_rf_wnum = 0x10, wb_rf_wdata = 0x0000000X
```

Nextpc 出错，检查第一条指令的数据通路，发现是 ds2es bus 的长度出现错误，导致正常数据未被传输过去。



修改后仍然错误，继续往后查看 EXE 阶段，es pc 出现奇怪的值，说明赋值出现错误



```
es_alu_op, es_mul_op, es_load_op, es_store_op, es_rkd_value, es_gr_we, es_dest, ds_exc_data = ds2es_bus_reg;
```

检查后发现 EXE 阶段漏给 es time op 赋值, 遗漏了两位。

```
assign {es_pc,es_res_from_mul, es_alu_src1, es_alu_src2, es_alu_op,
es_mul_op, es_load_op, es_store_op, es_rkd_value, es_gr_we, es_dest,
ds_exc_data,es_time_op} = ds2es_bus_reg;
```

修正后该时间仿真测试通过。

6、错误 6：各模块之间的 bus 位宽长度出错

(1) 错误现象

```
[1542000 ns] Test is running, debug_wb_pc = 0x00000000
[1552000 ns] Test is running, debug_wb_pc = 0x00000000
[1562000 ns] Test is running, debug_wb_pc = 0x00000000
[1572000 ns] Test is running, debug_wb_pc = 0x00000000
[1582000 ns] Test is running, debug_wb_pc = 0x00000000
[1592000 ns] Test is running, debug_wb_pc = 0x00000000
[1602000 ns] Test is running, debug_wb_pc = 0x00000000
```

(2) 分析定位过程

ID 模块中的 ds_ready_go 信号此时未拉高，而它理应拉高，因为此时是例外产生信号，应当把 bus 中的数据刷掉，但这里并未刷掉。

(4) 修正效果

```
ds_ready_go = ~(exe_rf_we && (es_block | es_csr_re) && (exe_conflict_rj || exe_conflict_rkd) ||  
                mem_rf_we && (ms_csr_re) && (mem_conflict_rj || mem_conflict_rkd) ) || ds_reflush;
```

应当在 ds_ready_go 信号加上 ds_reflush 信号。修改后该位置仿真不报错

8、错误 8：寄存器 0 地址不需要赋值

(1) 错误现象

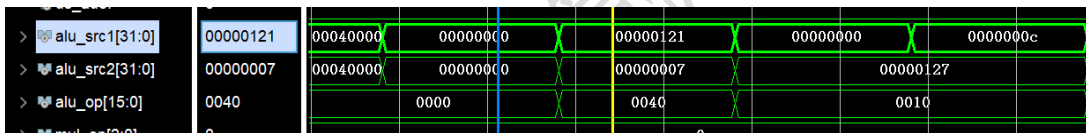
```
[1373157 ns] Error!!!  
reference: PC = 0x1c07202c, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x00000007  
mycpu      : PC = 0x1c07202c, wb_rf_wnum = 0x0d, wb_rf_wdata = 0x00000127
```

(2) 分析定位过程

查看汇编文件，发现这是一条 ori 指令，

```
101156 1c072018: 0401042c csrwr $r12,0x41  
101157 1c07201c: 50000000 b 0 # 1c07201c <n49_ti_ex_test+0x5c>  
101158 1c072020: 04010420 csrwr $r0,0x41  
101159 1c072024: 5c04773e bne $r25,$r30,1140(0x474) # 1c072498 <inst_error>  
101160 1c072028: 0400000c csrwr $r12,0x0  
101161 1c07202c: 03801c0d ori $r13,$r0,0x7
```

查看对应的译码阶段，发现是操作数选取有错。继续追踪发现是 rj_value 的赋值是 WB 阶段的前递结果，而事实上并不需要前递。虽然出现写后读，但是这是一条写地址为 0 的指令，所以并不需要前递。



(3) 错误原因

零地址不需要前递。

```
assign rj_value =  
    (exe_rf_we && exe_dest == rf_raddr1)? alu_result :  
    (mem_rf_we && mem_dest == rf_raddr1)? final_result :  
    (rf_we && rf_waddr == rf_raddr1)? rf_wdata :  
    rf_rdata1;  
assign rkd_value =  
    (exe_rf_we && exe_dest == rf_raddr2)? alu_result :  
    (mem_rf_we && mem_dest == rf_raddr2)? final_result :  
    (rf_we && rf_waddr == rf_raddr2)? rf_wdata :  
    rf_rdata2;
```

(4) 修正效果

通过增加判断读地址是否为 0 号位来控制需不需要前递数据。

```
assign rj_value =  
    (exe_rf_we && exe_dest == rf_raddr1 && |rf_raddr1)? alu_result :  
    (mem_rf_we && mem_dest == rf_raddr1 && |rf_raddr1)? final_result :  
    (rf_we && rf_waddr == rf_raddr1 && |rf_raddr1)? rf_wdata :
```


10、错误 10：位宽错误

(1) 错误现象

```
[1926397 ns] Error!!!
reference: PC = 0x1c0081fc, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00080000
mycpu      : PC = 0x1c0081fc, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x000d0000
```

(2) 分析定位过程

查看汇编文件，这是一条 csrrd 指令，其读出的值有错，说明可能之前写错或者和上个 bug 一样考虑到了特殊位。

```
8 1c0081f4: 0400183c csrrw $r28,0x6
9 1c0081f8: 5c6e677c bne $r27,$r28,28260(0x6e64) # 1c00f05c <ex_finish>
0 1c0081fc: 0400140c csrrd $r12,0x5
1 1c008200: 0044c18c srli.w $r12,$r12,0x10
```

不同的位是 ESTAT 寄存器的 Ecode 域，参考值位 8，但实现的 CPU 是 d，分别对应 ADE 错误和 INE 错误。

21:16	Ecode	R	例外类型一级编码。触发例外时，硬件会根据例外类型将表 7-7 中 Ecode 栏定义的数值写入该域。
-------	-------	---	--

找到 CPU 中上次给 ECODE 写入 d 的情况，发现对应的指令是一个取值错误 pc，理应是 ine 而不是 ade。

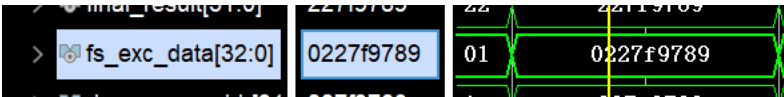


这是在 EXE 产生新异常的时候未考虑这可能是错误 pc 导致的异常，因此 ecode 应为取指地址错误的情况而非访存地址错误的情况。

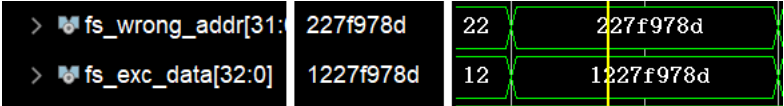
```
wire fs_exc_data;
wire [31:0] ds_inst;
wire [31:0] ds_pc;
wire [31:0] ds_wrong_addr;
wire ds_ade;
assign {ds_ade, ds_wrong_addr} = fs_exc_data;
```

原因还是因为源代码中，传给 ID 阶段的 fs_exc_data 位宽不对，而因为之前并没有取指地址错误的情况，所以并没有报错。但修改后继续仿真发现还是在同样的位置报错，检查后发现从 IF 传过来的例外信息与预期不符。

ID 中：



IF 中：



立刻想到例外信息的位宽应该是 97 位，而不是 96 位，所以最高位被切掉出现错误。

(3) 修正效果

```
define BR_BUS 33
define FS2DS_BUS_LEN 96 //fs_exc_data, fs_pc, inst
```

重新修改后进行仿真，该测试点通过。

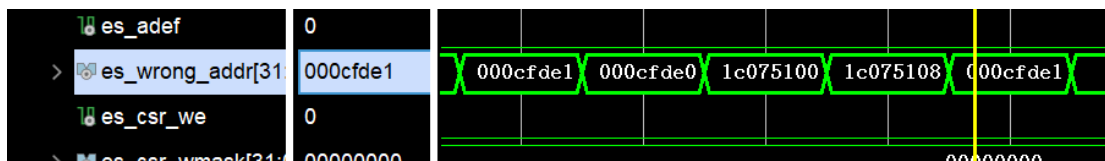
11、错误 11：魔数导致的 bug

(1) 错误现象

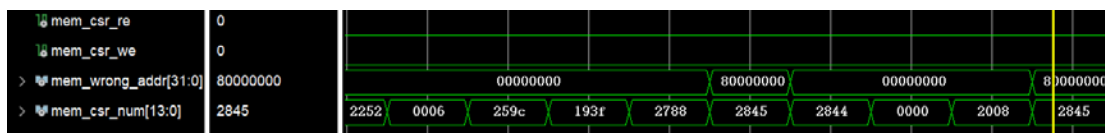
```
[1935217 ns] Error!!!  
reference: PC = 0x1c075124, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x000cfde1  
mycpu      : PC = 0x1c075124, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x0000fde1
```

(2) 分析定位过程

这同样是一条 csrrd 指令，读出 BADV 寄存器中的值，因此立刻联想到访存地址或取指地址出错产生的异常，查看上一次给 BADV 寄存器写入 0x0000fde1 的情况，看到其 EXE 阶段给出的地址正确，但最后写入的地址却是错误的



想破脑袋都想不明白，为什么 mem_wrong_addr 会变成 0x80000000。肯定不会是位宽的问题，因为如果位宽错了，早应该在之前某个地方就报错了，我本来是这么想的，但是因为在 MEM 的译码只是为了生成一些简单的控制逻辑，所以这里不应该有影响才对，mem 的异常信息是直接传给了 WB 阶段。而这里之所以会是 0x80000000 是因为未声明 wmask 信号，导致其隐式为 1 出现 wrong_addr 变为奇怪的值。



但是这并不能解释为什么直接传给 WB 阶段的异常信息的译码会出现两位位的差别。

最终检查 WB 阶段的译码时发现了问题，之前该位置是一个魔数，源于 exp12 中对其设置的魔数，其只有固定的 83 位，但是加入新的内容后，当前应有 97 位，导致 ws_wrong_addr 的一部分被抹去，从而出现了错误。因此引入头文件中定义的宏定义从而避免魔数的问题。

(3) 修正效果

```
assign {ws_wrong_addr,  
        ws_csr_we,  
        ws_csr_wmask,  
        ws_csr_num,  
        ws_ertn_flush,  
        ws_ex,  
        ws_esubcode,  
        ws_ecode,  
        ws_csr_re  
        } = ws_exc_data & {`MS_EXC_DATA_WD {ws_valid}};
```

修改后该时间点测试通过。

12、错误 12：除法器历史遗留问题

(1) 错误现象

```
[1938497 ns] Error!!!  
reference: PC = 0x1c075258, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00000000  
mycpu      : PC = 0x1c075258, wb_rf_wnum = 0x0c, wb_rf_wdata = 0x00004094
```

(2) 分析定位过程

该位置是一条除法指令，写数据与预期不符，很奇怪，因为之前除法器的仿真并未出错，但是这个错误确实就是除法器导致的。

检查后发现，这居然是因为当前指令是 0 操作数除以一个非 0 值，但是之前的除法器未考虑到被除数为 0，除数非 0 的情况，最终导致照常恢复余数产生错误的结果。

(3) 修正效果

对最后的 `result` 进行选取即可：

```
assign result = ({divisor} ? {quotient,remainder} : {32'b0,remainder});
```

修改后，仿真通过。

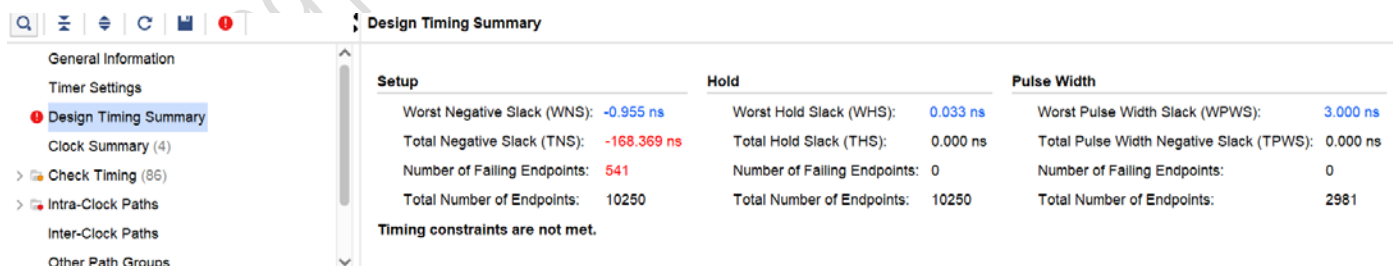
四、实验总结（可选）

本次实验中比较难的部分在于捋清每个异常指令是如何运行的，比如 `ertn` 和 `syscall` 指令应该放在哪部分处理。决定放 WB 阶段后，哪些指令对应的数据会变成无效的以及如何置为无效。由于老师已经将大部分 csr 寄存器部分的代码给出，因此本次实验主要是对控制通路进行修改。

在这个实验中，深切感受到了头文件和宏定义的重要性，不然新增位宽时，调整每个地方的信号位宽将是一个十分痛苦和折磨的事情。同时处理器框架的设计很重要，随着需要加的模块越来越多，信号越来越多，在每次实验开始时，需要调整和兼容的信号也变得比较复杂。因此利用 bus 信号线来传输数据确实是一个非常明智的选择，但是其位宽的变化也成为了一个非常容易出错的问题，需要着重考虑。实验中出现的大部分 bug 也很多都是因为位宽出错导致的。

随着代码增加 debug 的难度也增加了不少，由于不同同学写的代码不一样，风格不尽相同，每次都需要重新理解其他同学写的代码，debug 花费的时间反而是最多的。因此组内的分工反而也变成了一个很有意思的问题，在之后还是需要考虑考虑。

最后附上最终上板通过版本的时序情况：



Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -0.955 ns	Worst Hold Slack (WHS): 0.033 ns	Worst Pulse Width Slack (WPWS): 3.000 ns
Total Negative Slack (TNS): -168.369 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 541	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 10250	Total Number of Endpoints: 10250	Total Number of Endpoints: 2981

Timing constraints are not met.

时序勉强合格，最大的违约路径依然是由于前递数据通路导致的，但这似乎无可避免。不过目前来说，时序并未产生毁灭性的错误，所以姑且先暂时不考虑优化时序，毕竟完全不清楚 vivado 的综合实现的机制，很大可能出现反向优化的情况。其上板依然能正常通过。