

中国科学院大学

《计算机体系结构(研讨课)》实验报告

姓名 裴晨皓 竹彦博 纪弘璐 学号 2023K8009916003 2023K8009916001 2023K8009916002
专业 计算机科学与技术 实验项目编号 Project 4 实验名称 异常与中断设计专题实验

1 实验简介

简而言之,这次实验的内容是在已有的流水线处理器中添加异常与中断的支持,以及实现3条CSR指令和3条计时器相关指令。

2 设计方案介绍

下图是完成本次实验后的处理器结构框图:

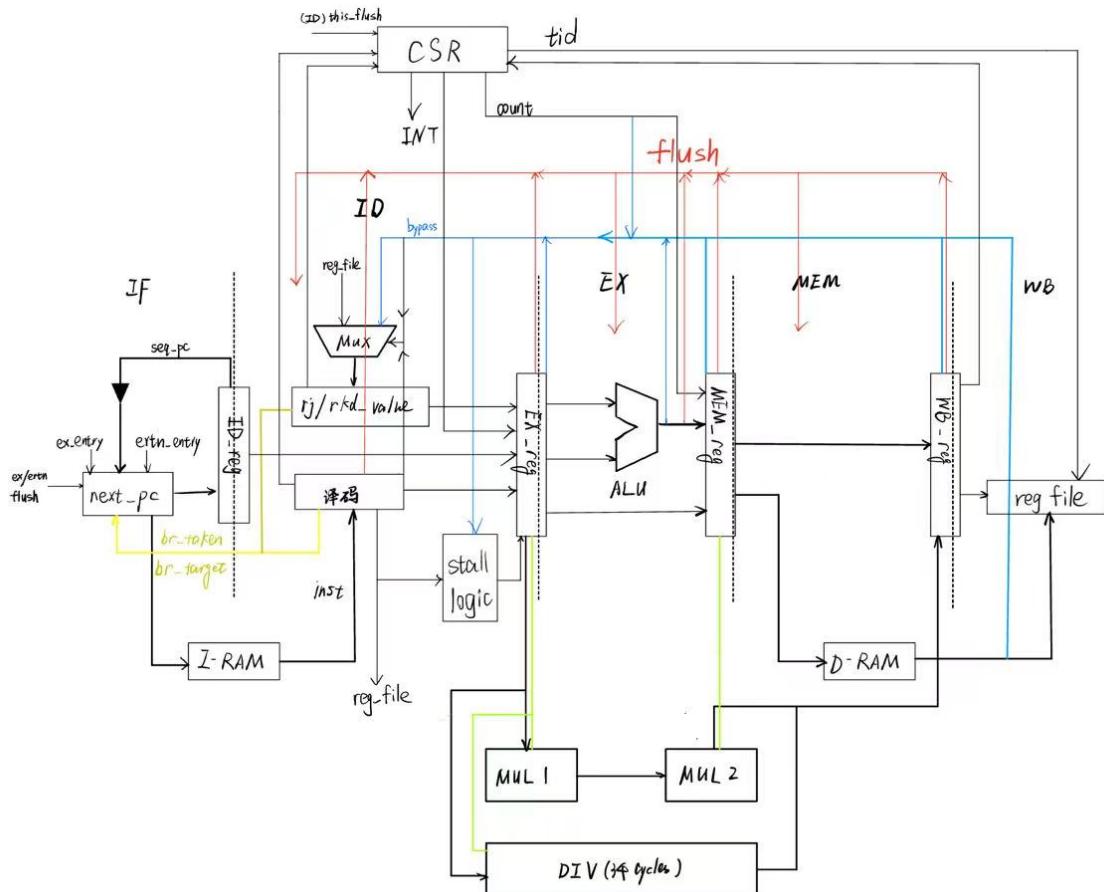


图 1: 处理器结构框图

2.1 总体思路

首先,我们按照教材的提示,阅读指令集手册,完成了CSR模块的设计。然后我们又对已有的流水线处理器进行修改,使之具备检测异常、标记异常、传递异常信息、最终报出异常并进行冲刷的功能,还能够在执行完异常

处理程序后顺利回到异常发生前的位置继续执行——为了达到这一目的并实现精确异常,我们又对处理器的控制信号进行了反复修改。

我们还使处理器具备了完成 csr 指令和计时器相关指令的能力,完成了本次实验。

2.2 CSR 模块设计

2.2.1 接口定义

根据教材的建议,我们把 CSR 实现为一个单独的 module,下面是其接口定义:

名称	方向与类型	位宽	含义
clk	in(wire)	1	时钟信号
rst	in(wire)	1	复位信号
csr_re	in(wire)	1	csr 寄存器写使能
csr_num	in(wire)	14	csr 寄存器地址 (编号)
csr_rvalue	out(wire)	32	csr 寄存器读出值
csr_we	in(wire)	1	csr 寄存器写使能
csr_wmask	in(wire)	32	csr 寄存器写掩码
csr_wvalue	in(wire)	32	csr 寄存器写入值
wb_ex	in(wire)	1	处理器在 WB 阶段是否报出异常
wb_ecode	in(wire)	6	例外类型一级编码
wb_esubcode	in(wire)	9	例外类型二级编码
wb_pc	in(wire)	32	报出异常的指令的 pc
wb_vaddr	in(wire)	32	出现异常的地址
ertn_flush	in(wire)	1	接收处理器执行 ertn 指令时报出的 flush 信号
ex_entry	out(wire)	32	例外处理程序入口地址
has_int	out(wire)	1	向处理器报告有无中断
ertn_entry	out(wire)	32	例外处理完毕后的返回地址
tid	out(wire)	32	专用于 rdcntid 指令读取 tid 寄存器值
count	out(reg)	64	计时器 (不是产生定时器中断时所用的那个倒计时计数器)

表 1: CSR 模块接口定义

这些接口中,除了 clk 和 rst 信号外,可以分为三类:

1. CSR 指令访问的接口 (以 csr_ 为开头命名的信号): 我们把 csr 指令对 csr 寄存器的读写都放在 ID 阶段进行,这些接口专用于“集中实现”本次实验 3 条 csr 指令的这部分需求。
2. 与硬件电路逻辑直接交互的接口 (包括 wb_ 开头的、ertn_ 开头的以及 ex_entry、has_int 信号): 接收处理器 (WB 阶段) 报出的异常、向处理器提供异常处理程序入口 (IF)、报告中断 (ID), 以及接收处理器报出的 ertn(WB) 并提供 ertn 后的返回地址 (IF)——这些功能分散在流水线处理器的不同阶段,一方面为了界面清晰、便于增量式开发,另一方面为了避免与 csr 指令出现结构冲突 (争用 csr 模块接口),我们把用于这些功能的信号单独“分散实现”以更方便地解决需求。
3. (3 条) 计时器相关指令的接口: rdcntvl.w、rdcntvh.w 这两条指令的“读取”任务在流水线的 EX 阶段完成,rdcntid 在 WB 阶段完成,同样为了避免与正常的 csr 指令读取操作结构冲突,单设 tid 和 count 接口服务于这三者。

2.2.2 模块内部实现

我们采用把各个 csr 寄存器的各个字段单独实现的方式进行设计,根据每个字段修改的不同条件对它们进行赋值,以 PRMD 的 PIE 和 PPLV 字段为例:

```
1 always @(posedge clk) begin
2     if (wb_ex) begin
3         csr_prmd_pplv <= csr_crmd_plv;
4         csr_prmd_pie <= csr_crmd_ie;
5     end
6     else if (csr_we && csr_num==`CSR_PRMD) begin
7         csr_prmd_pplv <= csr_wmask[`CSR_PRMD_PPLV] & csr_wvalue[`CSR_PRMD_PPLV]
8             | ~csr_wmask[`CSR_PRMD_PPLV] & csr_prmd_pplv;
9         csr_prmd_pie <= csr_wmask[`CSR_PRMD_PIE] & csr_wvalue[`CSR_PRMD_PIE]
10            | ~csr_wmask[`CSR_PRMD_PIE] & csr_prmd_pie;
11     end
12 end
```

Listing 1: PRMD 的赋值逻辑

当 WB 阶段报出异常时 (wb_ex),把 CRMD 的 PLV 和 IE 字段分别写入 PRMD 的 PPLV 和 PIE 字段。遇到涉及写 PRMD 寄存器的 csr 指令时 (csr_we 拉高、csr_num 对应 PRMD),将掩码与写入值“与”,得到被改写的部分内容;将取反后的掩码与原值“与”,得到未被改写的部分,最后将两部分合(“或”)在一起作为新值赋给相应字段。

把多个字段进行拼接,就能得出一个完整 csr 寄存器的读取数据值:

```
1 assign csr_prmd_rvalue = {29'b0, csr_prmd_pie, csr_prmd_pplv};
```

Listing 2: csr 寄存器读出值 (以 PRMD 为例)

之后,根据输入的 csr_num 在“拼接好”的各 csr 寄存器当中进行选择,就可以得到 csr_rvalue;同样,ex_entry、ertn_entry 和 tid 也利用这些“拼接好”的向量即可。

在形成中断标志时,要注意相应中断状态位 (IS) 和中断使能位 (LIE) 同时为 1,且全局中断使能 (IE) 为 1 时才可报告中断:

```
1 assign has_int = ((csr_estat_is[12:0] & csr_ecfg_lie[12:0]) != 13'b0) && (csr_crmd_ie == 1'b1);
```

Listing 3: 中断标志

此外,计时器 count 只需要在复位时初始为 0,每个上升沿增加 1 即可。

2.3 流水线中异常的检测与异常信息的传递

由于不同类型的异常可能发生在不同的流水级,所以我们在各级模块内都添加了以异常类型命名的异常判定的逻辑:

1. IF:

取指地址错异常 (ADEF),在取指地址 nextpc 不满足 4 字节对齐 (末两位不为 0) 时发生。

```
1 assign ADEF = nextpc[1: 0] != 0;
```

Listing 4: ADEF 的检测

异常的 PC 不应该被用来取指,故这时把指令 sram 的读使能置为 0:

```
1 assign inst_sram_en = !ADEF;
```

Listing 5: ADEF 异常时停止取指

2. ID(由于这部分信号产生逻辑较为直白,故此处不作代码展示):

- (a) 在译码后,根据得到的具体指令(inst_xxx 信号),可以直地判断系统调用异常(SYS)和断点异常(BRK);
- (b) 对于指令不存在异常(INE),我们选择把所有指令类型的信号“或”在一起再取反,即判定当前 inst 是否不属于任何一条已经实现的指令;
- (c) 此外,本阶段还根据 CSR 模块传来的 has_int(has_interrupt) 信号对中断(INT)进行检测。

3. EX:

经过 ALU 的运算,可以得到 load/store 的访存地址(alu_result)。类似 ADEF 的判定,在访存地址不满足对齐要求时,发生地址非对齐异常(ALE)。在访问字节时,不存在这类异常,但在访问半字或全字时,不满足 2 或 4 字节对齐(不满足地址末 1 位或 2 位为 0),就出现这类异常:

```
1 assign ALE = (mem_op[1] || mem_op[4] || mem_op[6]) && alu_result[0] != 1'b0 ||
2 (mem_op[2] || mem_op[7]) && alu_result[1:0] != 2'b00;
```

Listing 6: ALE 的检测(mem_op 是用于区分不同类型访存指令的独热码)

在我们的设计中,发生异常的指令进入 WB,才会报出异常,因此要修改流水线的数据通路,使异常信息随指令一起流动。我们在各阶段定义了“has_exception_out”信号,用于标明当前的指令从进入 IF 到当前阶段完成是否发生异常(每一级的 has_exception_out 是下一级的 has_exception):

```
1 always @(posedge clk) begin
2   if (rst) begin
3     has_exception_out <= 1'b0;
4   end
5   else if (in_valid && ready_go && out_ready) begin
6     has_exception_out <= has_exception || SYSCALL || BRK || INE || INT;
7   end
8 end
```

Listing 7: has_exception_out 信号(以 ID 阶段为例)

流水线流动时,每一级需要把本级可能产生的异常标志,合并(逻辑“或”)到从上一级“继承”来的异常标记 has_exception 中,作为 has_exception_out 传递下去,成为下一级的 has_exception。

此外,例外类型的编码(ecode 和 esubcode)也需要随指令流动,在 WB 报出异常的同时传给 csr。同样以 ID 阶段为例:

```
1 always @(posedge clk) begin
2   if (rst) begin
3     ecode_out <= 6'b0;
4   end
5   else if (in_valid && ready_go && out_ready) begin
6     if(INT) begin
7       ecode_out <= 6'h0;
8     end
9     else if (!has_exception) begin
10       ecode_out <= {6{SYSCALL}} & 6'hb | {6{BRK}} & 6'hc | {6{INE}} & 6'hd;
```

```

11      end
12  else begin
13      ecode_out <= ecode;
14  end
15 end
16

```

Listing 8: ecode 信号 (以 ID 阶段为例)

在流入下一级时,把中断作为最高优先级,如果有中断,就置 INT 对应的 ecode,否则看当前指令在前面所有阶段是否已经发生异常 (has_exception 为 1 还是 0):

1. 未在前面发生异常,则根据当前阶段的各种异常标志给 ecode_out 赋值,传给下一级。
2. 前面阶段已发生异常,即 IF 发生取指地址错异常,既然取指地址出现问题,自然也不会取出指令,所以更不必谈指令本身是否存在异常,于是不用顾及本阶段的异常标志,直接把上一级 IF 传来的 ecode 向下传递。

EX 到 MEM 的 ecode 传递也类似,如果 EX 阶段指令在 EX 前发生异常,则一定不是一条正常取出的访存指令,也就必不可能出现 ALE 异常,所以直接传递上一级的 ecode 是正确的;MEM 由于没有新的异常检测逻辑,所以直接把 ecode 传给 WB 即可。

```

1 always @(posedge clk) begin
2     if (rst) begin
3         esubcode_out <= 9'b0;
4     end
5     else if (in_valid && ready_go && out_ready) begin
6         if(INT) begin
7             esubcode_out <= 9'b0;
8         end
9         else if (!has_exception) begin
10            esubcode_out <= 9'b0;
11        end
12        else begin
13            esubcode_out <= esubcode;
14        end
15    end
16

```

Listing 9: esubcode 信号 (以 ID 阶段为例)

esubcode 与 ecode 类似,依旧是中断优先级最高,其他情况依旧按照进入当前流水级之前是否发生异常进行选择。

除了例外标志和例外类型的编码,还要传递 ADEF 异常的错误 PC、ALE 异常的错误访存地址(exception_maddr),在 WB 报出异常时一起传给 CSR 模块。

2.4 处理器控制信号的修改 (精确异常/异常返回的实现)

1. 为了实现精确异常,需要让报出异常的指令之后的指令全都“好像未执行过”一样,在完成异常处理后才去执行它们。
2. 由于 ertn 指令在 WB 阶段才将下一条指令的地址改为异常处理完毕后的返回地址,所以在给出返回地址前,ertn 指令“之前”的流水级中的指令都不应该被执行 (ertn 在流水线中流动时,后续还会有顺序取指得到的指令进入流水线,但这些指令是“误入”流水线的,甚至可能是根本没有实现的指令)。

其实，上面提到的两种情况都可以理解为有指令“误入”流水线，它们不应该被执行，不能影响体系状态——我们采取冲刷的方式以应对之，即把异常指令/ertn 指令以及前面流水级的指令都冲刷掉。(实际上，ertn 并非被冲刷，而是从流水线中流出，但为了信号方面便于复用，也可将其视为(流出的同时)冲刷——这与流出是等效的)

2.4.1 阻止“误入”指令执行的控制信号 this_flush

如果误入流水线的指令对整个体系的状态产生影响(比如修改了 CSR 寄存器、内存等)，那么我们冲刷时还需要把产生的这些影响还原，产生额外的开销。所以我们在每一流水级内引入控制信号 this_flush，用于标记当前流水级的指令是否在接下来的某一刻被冲刷，阻止这些会被冲刷的指令改变体系状态：

```
1 assign this_flush = in_valid && (has_exception || next_flush || ALE || ertn);
```

Listing 10: this_flush 信号(以 EX 阶段为例)

以 EX 阶段为例，首先要求 EX 阶段有效，否则也不必谈及阻止指令行为或冲刷。在这一基础上，如果当前指令在前面流水级出现异常(has_exception)或在当前流水级出现异常(ALE)，亦或是当前指令是 ertn，那么当前指令就不能被执行，会被冲刷。此外，next_flush 是下一级 MEM 的 this_flush 信号，由于异常/ertn 指令所在流水级之前的连续多个流水级的指令都需要被取消掉，即“下一级冲刷(被取消、不能执行)，当前级指令也一定要冲刷(被取消、不能执行)”，所以要把 next_flush 也纳入 this_flush 的逻辑。

this_flush 在多处使用，例如只有当 this_flush 为 0，当前阶段的指令不是“误入”指令时，才可以与乘除法器模块握手：

```
1 assign to_mul_req_valid = in_valid && res_from_mul && !this_flush;
2 assign to_div_req_valid = in_valid && res_from_div && !this_flush;
```

Listing 11: EX 阶段与乘法器的握手请求信号

再比如，只有 this_flush 为 0 时，才允许拉高数据内存的写使能：

```
1 assign data_sram_we = {4{mem_we && valid && in_valid && !this_flush}} & (
2     ({4{mem_op[5]}} & (4'b0001 << result[1: 0])) | // SB
3     ({4{mem_op[6]}} & (4'b0011 << result[1: 0])) | // SH
4     ({4{mem_op[7]}} & 4'b1111) // SW;
5 );
```

Listing 12: 数据内存写使能信号

2.4.2 “冲刷”行为的进行

```
1 assign exception_submit = in_valid && has_exception; //exception_submit 即 ex_flush
```

Listing 13: WB 报告异常(报告 ertn 的逻辑类似)

当 ertn 指令或是异常指令在 WB 报出时，会给出 ertn_flush 或是 ex_flush 信号，通知各流水级进行冲刷，各流水级把本级内的有效信息变成“空泡”(给 out_valid 赋值为 0 作为下级的 in_valid)，流入下一级，完成冲刷。

```
1 always @(posedge clk) begin
2     if (rst) begin
3         out_valid <= 1'b0;
4     end
5     else if (out_ready) begin
6         out_valid <= in_valid && ready_go && !ex_flush && !ertn_flush;
7     end
8 end
```

Listing 14: 冲刷时对 valid 的赋值逻辑(以 MEM 级为例)

在具体设计中,我们运用 this_flush 信号,使每一级在冲刷时的 ready_go 都为 1,进而使得每一级的 out_ready 都为 1,从而使得冲刷过程可以 1 拍完成(且由于需要被冲刷的指令功能受到抑制,也不会因自身功能未完成而被阻塞,故对 ready_go 如此处理是合理的):

```

1 assign ready_go = !in_valid ||  
2     this_flush ||  
3     !(res_from_mul && !(to_mul_resp_ready && from_mul_resp_valid)) &&  
4     !(res_from_div && !(to_div_resp_ready && from_div_resp_valid));

```

Listing 15: ready_go 信号(以 MEM 级为例)

```

1 assign in_ready = ~rst & (~in_valid | ready_go & out_ready);

```

Listing 16: in_ready 信号(以 MEM 级为例(MEM 的 in_ready 即为 EX 的 out_ready))

2.5 处理器中 CSR 指令的实现

为了减少不必要的阻塞,我们把对 CSR 寄存器的读写操作全部放在 ID 完成,下面是 CSR 指令与 CSR 寄存器交互的信号实现:

```

1 assign csr_num = inst[23: 10];  
2 assign csr_we = in_valid && (inst_csrwr || inst_csrwchg) && ready_go && out_ready && !this_flush  
    ;  
3 assign csr_wmask = {32{inst_csrwr}} | {32{inst_csrwchg}} & rj_value;  
4 assign csr_wvalue = rkd_value;

```

Listing 17: CSR 指令与 CSR 寄存器的交互信号

1. csr_num: 根据指令码的 10 到 23 位,选择要访问的 CSR 寄存器。
2. csr_we: 当前 ID 阶段是有效的改写 CSR 寄存器的 CSR 指令,且当前阶段工作完成并能进入下一级(ready_go 和 out_ready 同时为 1),并且不是“误入”流水线的指令(this_flush 为 0),则可以拉高写使能。这里要尤其注意,csr_we 必须在 ID 与 EX 握手成功(ready_go 和 out_ready 同时为 1)时才能拉高,即进入 EX 前的那一拍拉高。否则:如果 ID 为被阻塞的 csrxchg 指令(需要向目标 csr 写入新值,并读取其旧值),那么它在被阻塞期间的第一个上升沿就会把新值写入目标 csr,从而使得读出值变成写入后的newValue,出现错误。
3. csr_mask: 遇到改写 csr 寄存器的两条指令时,写掩码置为 rj 寄存器的值。
4. csr_wvalue: 遇到改写 csr 寄存器的两条指令时,写入值为 rd 寄存器的值。

涉及读取 csr 寄存器值的 csr 指令,要把读出值逐级传递到 WB,写入目的通用寄存器。在我们的实现中,如果 csr 指令遇到关于通用寄存器的写后读相关,仍使用前递的方式解决。

2.6 处理器中计时器相关指令的实现

为了减少不必要的阻塞,rdcntvl.w 和 rdcntvh.w 这两条指令在 EX 阶段读取计时器的值(分别读低 32 位和高 32 位),把读取的结果逐级传递到 WB,写入目的通用寄存器。

按照教材的提示,rdcntid 指令读取的 tid 寄存器会被 CSR 指令修改,故不能在 ID 阶段读取,为简单起见,将其放在写回级读取。由于 CSR 是异步读,所以读取后可以马上写入目标通用寄存器。

与 load-use 情况类似,rdcntid 遇到“写后读”相关时,“写”值在 WB 阶段才被确定,所以采取与 load-use 情况类似的方式把发生相关的“读者”阻塞在 ID,我们等 rdcntid 完成 WB 再对被阻塞者予以放行。具体实现方法与阻塞 load-use 情况、阻塞与乘除法指令数据相关的指令的方式类似,不再占用篇幅展示代码。

此外,由于我们把计时器实现在了 csr 模块里,为了保证设计的一致性,rdcntvl.w 和 rdcntvh.w 读出的数据向下一流水级传递时复用 csr 读取值传递的数据通路:

```

1  always @(posedge clk) begin
2      if (rst) begin
3          csr_result_out <= 32'b0;
4      end
5      else if (in_valid && ready_go && out_ready) begin
6          csr_result_out <= rdcntvl_w ? count[31:0] :
7              rdcntvh_w ? count[63:32] :
8                  csr_result;
9      end
10 end

```

Listing 18: rdcntvl.w 和 rdcntvh.w 向下级传递时复用 csr 读取值的数据通路

在前递时这两条指令则与其他读取 csr 的指令以及 alu_result 指令共用数据通路。

3 Debug 记录

下面展示本组同学在 Debug 过程中遇到的一些印象深刻的 bug。

(由于 debug 时我们对部分设计进行了调整,部分信号名称难免与最终版本有所出入)

3.1 csr_we 过早拉高导致 csrxchg 指令的 csr 读错误

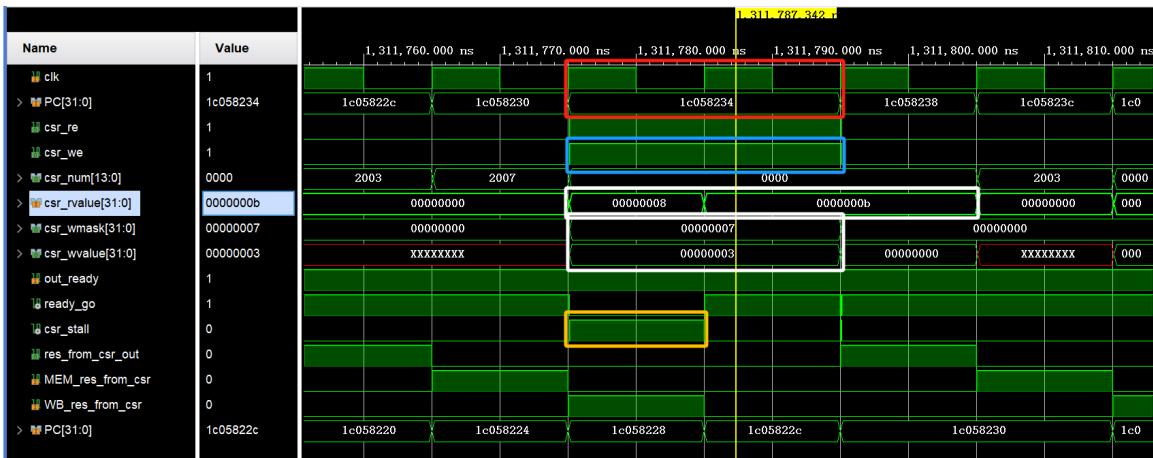


图 2: csr_we 过早拉高导致 csrxchg 指令的 csr 读错误示意图

在最初的实现中,我们把 csr 指令关于通用寄存器的写后读冲突全都采取阻塞处理。如上图所示:红框圈出的两个周期内,一条 csrxchg 指令因为写后读冲突 (csr_stall, 图中橙色方框所示) 被阻塞在 ID 阶段,但 csr_we 信号持续处于拉高状态 (图中蓝色方框所示),导致在第一个周期结束的那个上升沿,目标 csr 寄存器的末三位被置为了 0b011,导致目标 csr 的读出值从 0x8 变成了 0xb,导致这条指令无法把 csr 的旧值随流水传递下去,而是错误地把写入后的newValue传递下去。

修改方式是,在 ID 与 EX 握手成功的那一拍,即 ID 进入 EX 前的最后一拍拉高 csr_we,在 ID 流入 EX 的那个上升沿使得 csr 寄存器的旧值传入 ID 与 EX 间的流水寄存器,并且完成newValue写入。代码在前文中已展示,此处不在重复。

3.2 顶层模块对各子模块的连线出错

在设计过程中,由于各模块有较多信号需要输入或输出,而这些输入输出线与其他模块的连接都要依靠顶层模块 mycpu_top。如此多的接口难免连线混乱,出现一些错误,比如:

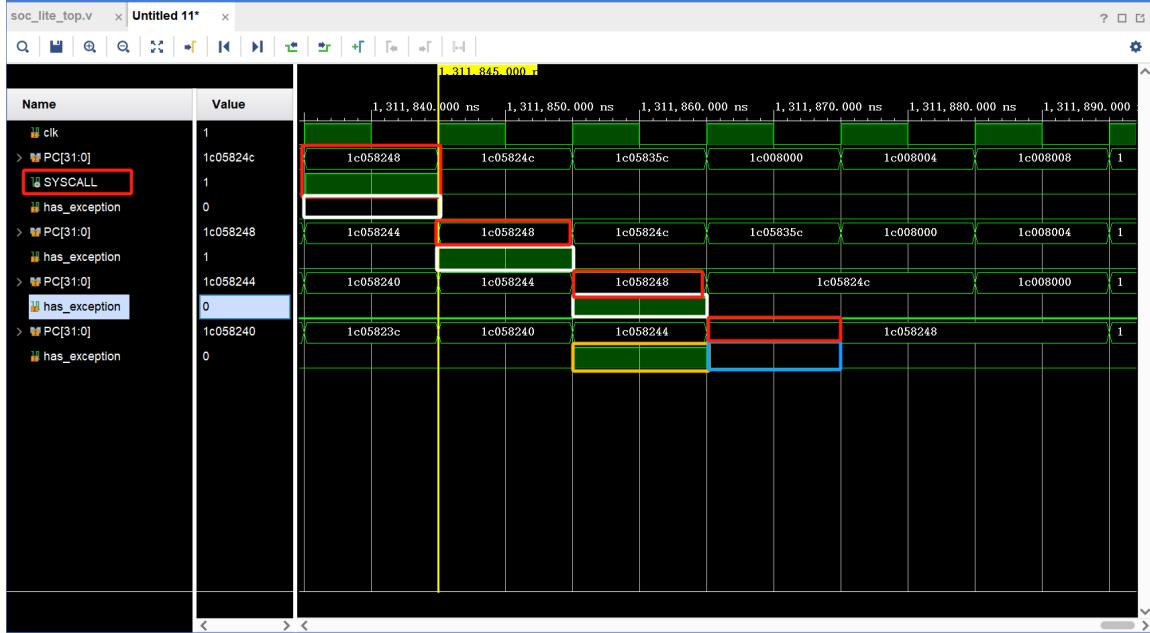


图 3: 顶层模块接线错误示例 (WB_has_exception 信号接错)

回顾前面的描述: has_exception 这一寄存器信号描述了当前阶段的指令在前面的所有流水级中是否出现异常。

红色方框标注的部分展示了 PC 为 0x1c058248 的指令在 ID 阶段译码,被判定为一条 syscall 指令,它随着流水线逐级流动,应当在 WB 报出异常。但如白色方框所示,实际行为却是:这条 syscall 进入 EX 后,has_exception 的确被拉高,但接下来流入 MEM 后,MEM 和 WB 的 has_exception 同时拉高,导致前面一条非异常指令 (PC 为 0x1c058244) 在 WB 错误地拿到异常信息 (如黄色方框所示)——导致错误地发生冲刷行为,使 syscall 没能进入 WB 报出异常 (本应在蓝色方框所示位置报异常)。蓝色方框上方的红色方框处虽然 PC 对应这条 syscall 指令,但实际上它已经被冲刷,通过蓝框所标注周期的 ID 阶段 PC(clk 下方的那个 PC 信号)可以看出,在这一周期时,ID 阶段已经是异常处理程序的入口指令。

通过波形也不难猜到,这一问题是由于把 EX 阶段传出的 has_exception 错误地同时接到了 MEM 和 WB,而非只接入 MEM。在修复这一 bug 的同时,我们重新检查了顶层的所有接线逻辑,并完善了顶层线路命名的规范。

3.3 nextpc 选择逻辑错误 (优先级考虑不周)

next_pc 有 4 种情况,顺序取指、分支跳转、异常入口、异常返回,最后两者应当优先级最高,其次是分支跳转目标,再次是顺序取指。

前面的实验中只有顺序取指和分支目标两种选择,不涉及优先级的问题,而加入新的两种情况后,我们忘记考虑优先级问题,把它们随意放在了优先级低于分支跳转的位置:

```
1 assign nextpc = out_ready && br_taken ? br_target : ex_flush ? ex_entry : ertn_flush ?
      ertn_entry : seq_pc;
```

Listing 19: 错误的 nextpc 选择逻辑

导致了如下情况：

```
1c058298 <syscall_pc2>:
1c058298:    002b0000      syscall 0x0
1c05829c:    2980027b      st.w   $r27,$r19,0
1c0582a0:    2880126d      ld.w   $r13,$r19,4(0x4)
1c0582a4:    5c00b9bb      bne    $r13,$r27,184(0xb8) # 1c05835c <inst_error>
1c0582a8:    5c00b73e      bne    $r25,$r30,180(0xb4) # 1c05835c <inst_error>
1c0582ac:    03800419      ori    $r25,$r0,0x1
1c0582b0:    29800279      st.w   $r25,$r19,0
```

图 4: 某条异常 (syscall) 指令及其后的指令序列

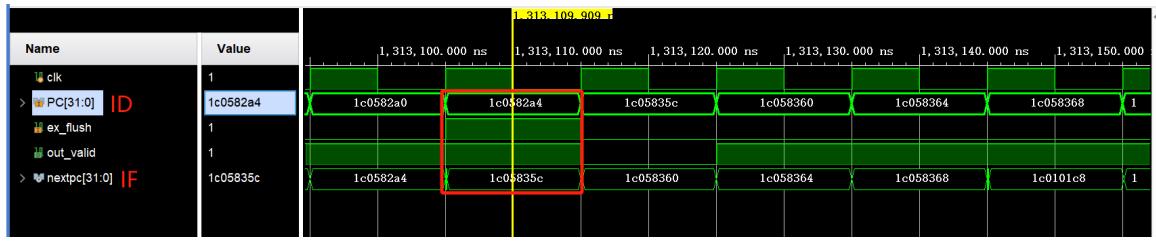


图 5: 图 4 指令序列执行时因 nextpc 选择逻辑错误导致的错误情况波形

红框圈出的位置，ID 阶段是 PC 为 1c0582a4 的分支指令 bne，它报出了分支跳转信号 br_taken，而这时 PC 为 1c058298 的 syscall 指令位于 WB，报出了异常冲刷信号 ex_flush。由于分支跳转优先级高，所以这时的 nextpc 被设置成了跳转目标 1c05835c，也就是 inst_error 的位置，而非异常处理程序入口。

改成正确的 nextpc 选择逻辑，代码变为：

```
1 assign nextpc = ex_flush ? ex_entry : ertn_flush ? ertn_entry : out_ready && br_taken ?
br_target : seq_pc;
```

Listing 20: 正确的 nextpc 选择逻辑

3.4 WB 执行 ertn 指令时忘记冲刷流水线

由于 ertn 指令在 WB 阶段才把 PC 设为异常返回地址，在此之前可能会有“意外”的 PC 因顺序取指而“误入”流水线，这些 PC 甚至可能没有指令，它们进入流水线后会报出指令不存在异常，并向下传递。

我们最初只把 ertn_flush 信号交给了 csr 模块，忘记利用这一信号对流水线进行冲刷：

下图包含 exp12 的 test.s 文件中异常返回指令 ertn，其 PC 为 0x1c00f078：

```
1c00f05c <ex_finish>:
ex_finish():
1c00f05c:    00100000      add.w  $r0,$r0,$r0
1c00f060:    0400180d      csrrd  $r13,0x6
1c00f064:    028011ad      addi.w $r13,$r13,4(0x4)
1c00f068:    0400182d      csrwr  $r13,0x6
1c00f06c:    0280032d      addi.w $r13,$r25,0
1c00f070:    5c000b20      bne    $r25,$r0,8(0x8) # 1c00f078 <ex_ret>
1c00f074:    141ffff9      lu12i.w $r25,65535(0xffff)

1c00f078 <ex_ret>:
ex_ret():
1c00f078:    06483800      ertn
...
```

图 6: 异常返回指令-反汇编

做 ertn 时, 忘记冲刷导致如下波形:

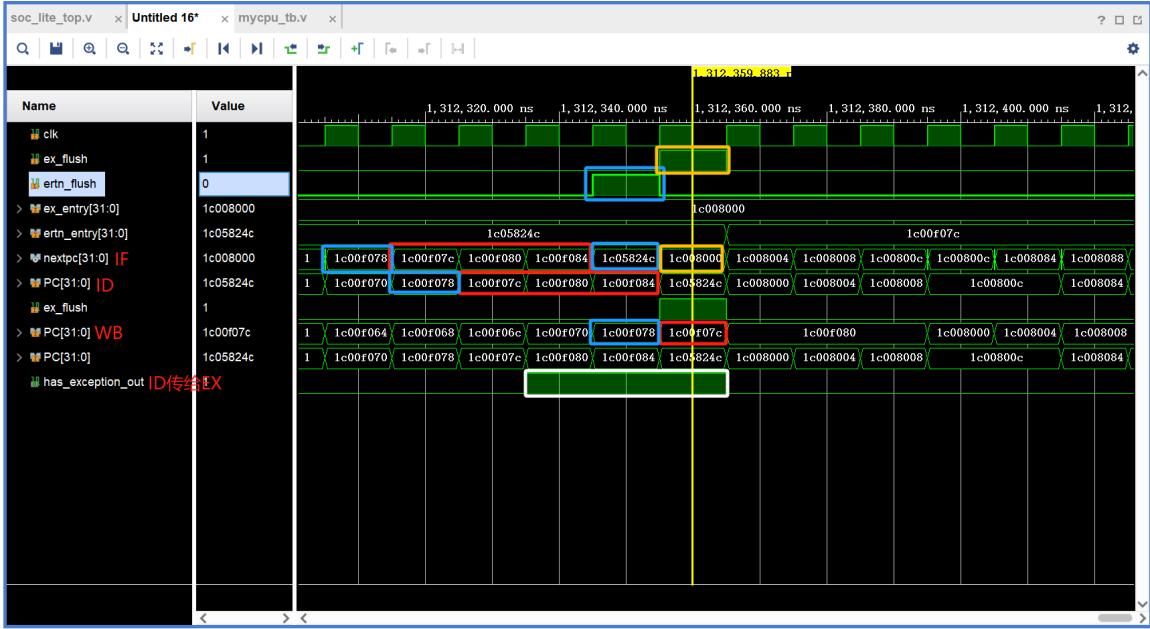


图 7: 做 ertn 指令时忘记冲刷前面的流水级

如蓝框所示, ertn 异常返回指令进入 WB, 报出 ertn_flush, 同时 nextpc 置为异常返回地址。由于 ertn 进入 WB 前, 没有跳转和异常/异常返回, 所以 PC 顺序取指, 遂在 PC 为 0x1c00f07c 以及后续地址尝试取指, 并随流水线一直流动至 WB(如红色方框所示)。因为这些 PC 的指令不存在, 所以 ID 阶段判定指令不存在后传给 EX 的 has_exception 为 1, 使这条不存在的指令带上异常标记(如白色框所示)。不存在的指令进入 WB 便报出异常 ex_flush, 使得取指 next_pc 又被设置成异常处理入口(黄色方框所示), 刚离开异常处理程序就又回去, 进入死循环。

把 ertn_flush 纳入各模块有效信号(valid)的赋值逻辑, 实现 ertn 在 WB 时对流水线的冲刷后, 由于不存在的指令被冲掉(无效), 自然也就不再会报出异常。具体代码见前文“‘冲刷’行为的进行”部分。

3.5 冲刷后各流水级未清除异常标志

最初我们忘记给 this_exception(后来把 ertn 合并其中并改名为 this_flush) 信号加上 valid 的限制, 使得被冲刷后的流水级仍保持着 this_exception。又由于前文提到 this_exception 为 1 时, ready_go 一定为 1, 就会导致一些本该被阻塞的情况因 ready_go 为 1 而被错误地放行。

比如前文中的图 4 所示的指令序列, PC 为 1c0582a0 和 1c0582a4 的两条相邻指令存在写后读数据相关, 读者需要被阻塞, 但是因为错误的 this_exception 而被放行:

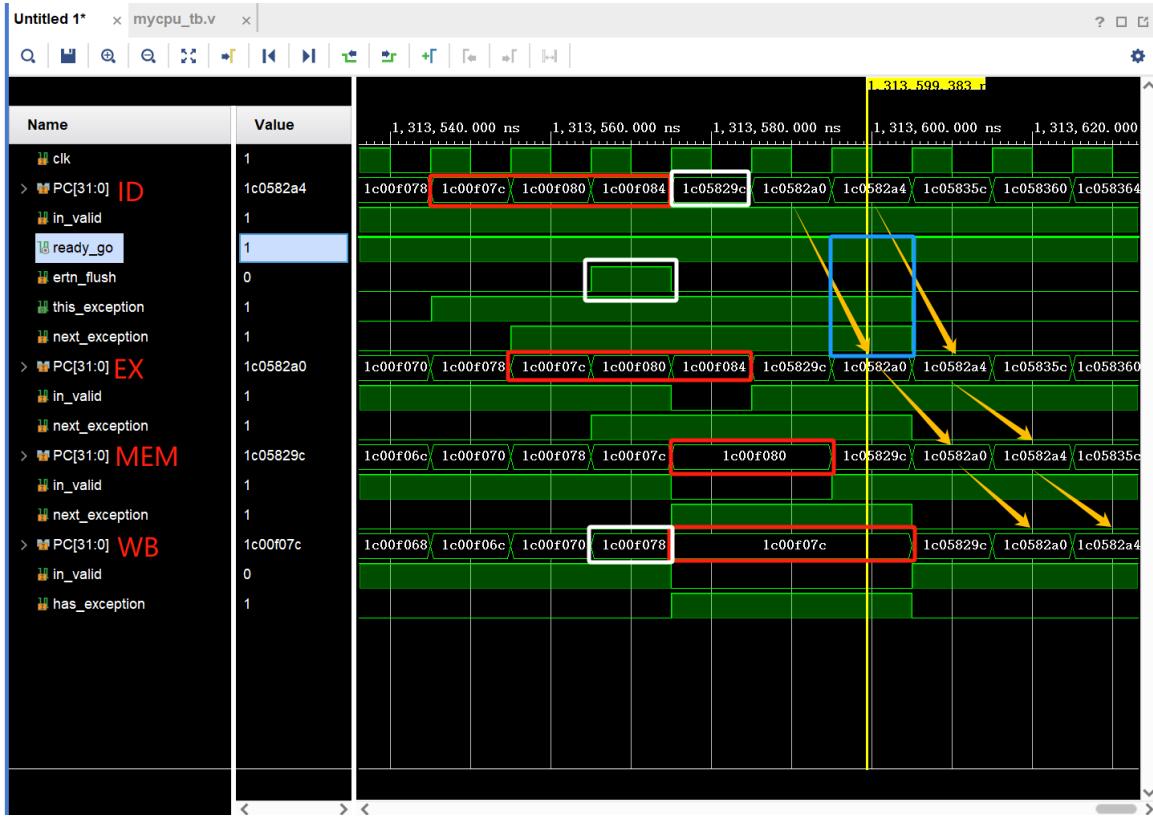


图 8: 冲刷后各流水级未清除异常标志导致的错误阻塞示例

上图红框内的 PC 为 1c00f078 后面的顺序 PC(相应的指令不存在), 会报出异常。如白色方框所示, ertn 在 WB 报出冲刷并开始取异常返回地址的指令, 下一拍这条指令就进入 ID。而冲刷后虽然 EX、MEM、WB 的 valid 都变成 0, this_exception 却为 1, 后续虽然有新的无异常指令进入 ID、EX 和 MEM, 使得本级的 has_exception 为 0, 但由于 WB 仍然无效, 仍有 has_exception(也是 WB 的 this_exception) 为 1, 它再通过 next_exception 传递给前面各级, 使得各级的 this_exception 都为 1。如蓝色框所示, 由于 ID 的 this_exception 为 1, 所以 ready_go 为 1, 所以本应阻塞在 ID 的 0x1c0582a4 指令错误进入了 EX 并继续流动(图中靠右的三段黄色箭头)。

所以我们采用 valid 信号对本级的 this_exception 进行限制, 使无效流水级的异常标记不会传递到前面的流水级(具体代码引入 valid 的方式参见前文 this_flush 信号)。

3.6 未阻止 ertn 后面误入流水线的指令发挥作用

上面一个 bug 描述中的 this/next_exception 后来被我们改造成了 this/next_flush, 就是因为我们发现, ertn 后面的顺序取指不一定会取出不存在的指令, 也有可能取出当前 CPU 支持的指令, 如下图反汇编文件的片段所示:

1c07ea04:	0280239c	addi.w \$r28,\$r28,8(0x8)
1c07ea08:	06483800	ertn
1c07ea0c:	0400182c	csrwr \$r12,0x6

图 9: ertn 指令后的顺序 PC 存在已实现的指令

图中的这条 csrwr 指令会在 ertn 进入流水线后被顺序取出, 由于它不会导致异常, 且后续流水级也没有异常, 所以功能正常进行, 在(下图所示的光标位置处)ID 阶段对 csr 进行写入(csr_we 被拉高), 写入了 b088f329:

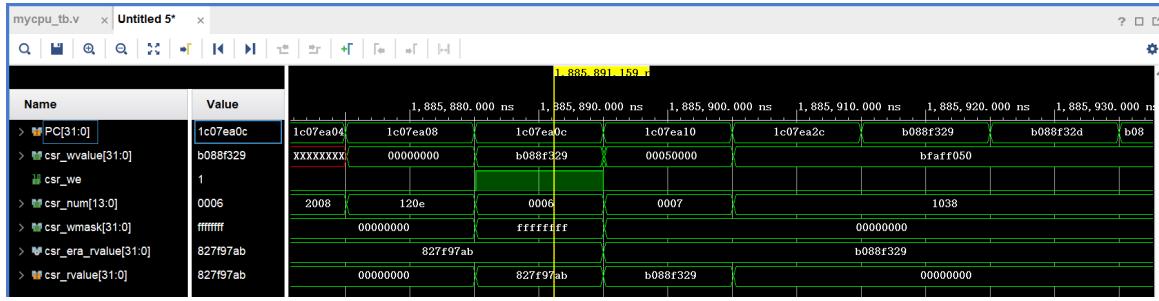


图 10: ertn 后的顺序指令 csrwr 改写了 CSR 寄存器值

在后来真正需要读取 csr 寄存器的值时,也就顺理成章地读出了被误写入的值,进而在写入目标通用寄存器时,与 golden_trace 不一致,报错,暴露 bug:

```
[1886127 ns] Error!!!
reference: PC = 0x1c0081f4, wb_rf_wnum = 0x1c, wb_rf_wdata = 0x827f97ab
mycpu    : PC = 0x1c0081f4, wb_rf_wnum = 0x1c, wb_rf_wdata = 0xb088f329
```

图 11: 因 ertn 后误入流水线的指令发挥作用 (改写 csr 寄存器) 导致的 csr 读错误,进而导致的报错

所以我们把 ertn 的情况也合并到了 this_exception 里,形成了前文所说的 this_flush,用来限制 ertn/异常指令前面流水级的指令发挥作用。

后续我们试图简化设计逻辑,想把 ertn 和异常的逻辑基本完全合并,但很快意识到这是不可行的:只有冲刷本身的逻辑两者可以共用,但冲刷后的 PC 设置以及 csr 寄存器的修改逻辑完全不同。

4 合作说明

本实验由本组成员共同合作完成,组内同学同等贡献。