实验 6-2 报告

第 27 小组
姚子薇、陈乐滢

一、实验任务

CPU 顶层修改为 AXI 接口，集成到 SoC_AXI_Lite 系统中。CPU 对外只有一个 AXI 接口，需内部完成取指和数据访问的仲裁。

验证方法：功能测试仿真和上板。

二、实验设计

（一）CPU-AXI 总线接口设计

将 CPU 接口改为类 sram 接口，inst 只读不写，req 和读地址的在 next_pc 发出，取回来的指令送到 fe 级；data 的 req 和读写地址等在 exe 级发出，取回来的数据送到 mem 级。一个要注意的就是读写对整个流水的影响，在读/写操作没有完成，即读的数据没有写回来或写的数据没有写进去时，整个流水线都需要阻塞，阻塞信号要送到五级流水的每一级。因此现在每一级都很长，有很多拍，每一级都一定要做一次取指，是否要取数据或者写数据则要看当前指令，load/store 的 exe 级都是先取指，再读/写数据。

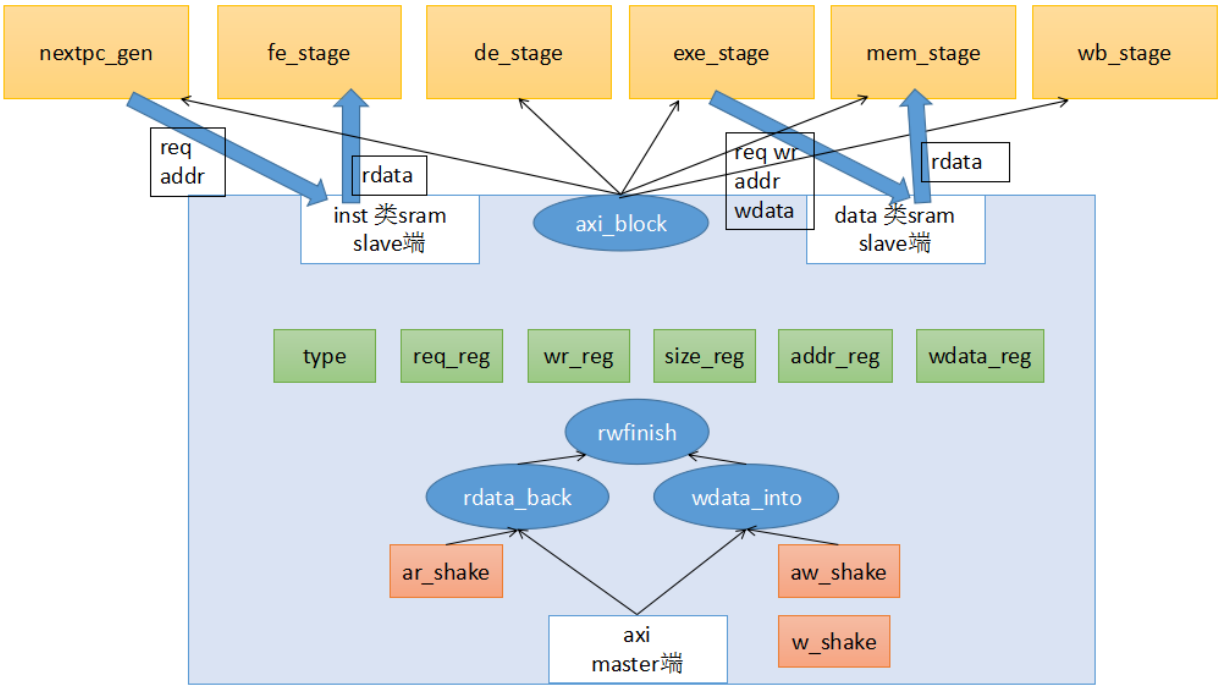

图 1. CPU-转换桥结构图

三、实验过程

（一）实验流水账

1. 姚子薇

2017.12.2，14:00 - 16:00，整体设计。

2017.12.3，10:00 - 12:40，添加模块，修改接口。

2017.12.3，20:00 - 23:00，代码初步完成。

2017.12.4，14:00 - 24:00，仿真调试。

2017.12.5， 8:00 - 14:00，仿真调试，前 43 个测试通过。

2017.12.5，14:30 - 17:30，整理错误，写实验报告。

2017.12.11，21:00 - 24:00，仿真调试。

2017.12.12， 9:30 - 16:30，仿真调试 pass。

2017.12.12，16:30 -17:10，综合调试，最终上板通过。

2017.12.12，17:10 - 17:40，完善实验报告。

2. 陈乐滢

2017.12.3，20:00 - 23:00，代码初步完成。

2017.12.5，10:00 - 17:00，仿真调试，前 45 个测试通过。

2017.12.5，17:30 - 18:00，整理错误，写实验报告。

2017.12.11，18:00 - 24:00，仿真调试。

2017.12.12， 9:30 - 16:30，仿真调试 pass。

2017.12.12，16:30 -17:10，综合调试，最终上板通过。

2017.12.12，17:10 - 17:40，撰写实验报告。

（二）错误记录（错误 10 之后为新的 bug）

1、错误 1

（1）错误现象

bfc00000 取不出指令

（2）分析定位过程

resetn 变 1 的时候 nextpc 直接是 bfc00004，因为取指需要的 req_reg 信号在 resetn 的时候一直是 0，!resetn 的瞬间 nextpc 就变成了 bfc00004，有两个思路，一是在 resetn 期间就把第一条指令取出来，二是!resetn 之后让 bfc00000 多延续一拍。

（3）错误原因

取指地址是 nextpc,而 nextpc 是根据 fe_inst 和 de 级情况的一个组合逻辑，resetn 一起来就会变成 bfc00004，导致第一条指令取不出来。

（4）修正效果

next_pc 原来是在(!resetn)时为 bfc00000，现在将 resetn 设一个 resetn_reg 多沿一拍.

改后的 nextpc 赋值：

```
assign nextpc = (~resetn) ? 32'hbfc00000:
                (~resetn_reg) ? 32'hbfc00000: //新加的
                (exc_handler)? 32'hbfc00380:
                (de_block||inst_block||data_block||axi_block)? fe_pc:
                                                C1 | C2 | C3 | C4 | C5 ;
```

修改后可以正常取指。

2、错误 2

（1）错误现象

[2298 ns] Error!!!

reference: PC = 0xbfc006b8, wb_rf_wnum = 0x10, wb_rf_wdata = 0x00000000

mycpu : PC = 0xbfc006c4, wb_rf_wnum = 0x1f, wb_rf_wdata = 0xbfc006cc

（2）分析定位过程

```
bfc006a8:ac890000       sw    t1,0(a0)
/media/sf_ucas_17-18/lab6/ucas_CDE_v0.3/cpu_func_full/start.S:276
bfc006ac:acaa0000       sw    t2,0(a1)
/media/sf_ucas_17-18/lab6/ucas_CDE_v0.3/cpu_func_full/start.S:277
bfc006b0:       accb0000        sw    t3,0(a2)
/media/sf_ucas_17-18/lab6/ucas_CDE_v0.3/cpu_func_full/start.S:278
bfc006b4:       ae330000        sw    s3,0(s1)
/media/sf_ucas_17-18/lab6/ucas_CDE_v0.3/cpu_func_full/start.S:279
bfc006b8:       3c100000        lui    s0,0x0
```

sw 指令，但是对应的 data_wr 却等于 0。

（3）错误原因

笔误，data_wr=exe_is_store 写成了 exe_is_load。

（4）修正效果

修改后正确运行。

3、错误 3

（1）错误现象

[18913 ns] Error!!!

reference: PC = 0xbfc00cf0, wb_rf_wnum = 0x0a, wb_rf_wdata = 0xffffffff

mycpu : PC = 0xbfc00cf0, wb_rf_wnum = 0x0a, wb_rf_wdata = 0x15400002

（2）分析定位过程

这是 lw 指令，写回地址正确，写回数据错误，应该是由总线输回的数据匹配不当。

lw 的 exe 级，先进行的是一条取指，然后是取数，取指结束后流水就接着走了，此时数还没有取回来。

（3）错误原因

阻塞信号不全面，data_block 发生在取指的过程里，和 inst_block 中间有一拍空拍，导致流水没有被阻塞住。

（4）修正效果

修改 data_block,让它在 data_addr_ok 时就开始阻塞，修改后这个问题解决。

4、错误 4

（1）错误现象

[18938 ns] Error!!!

reference: PC = 0xbfc00d00, wb_rf_wnum = 0x08, wb_rf_wdata = 0x00000001

mycpu : PC = 0xbfc00cf0, wb_rf_wnum = 0x0a, wb_rf_wdata = 0xffffffff

（2）分析定位过程

data 读完了可以选择新任务，应该选择取指令，但仍然选择了取数据，导致取指错误触发保留指令例外

选择的原因是 data_req 和 inst_req 同时为 1，且都是读，那么 data 的优先级更高，就会一直取数据不取指令，流水卡死在这。

（3）错误原因

data_req 没有在开始执行取数后清零。

（4）修正效果

data_req 改为 reg 类型，在 load/store 的时候为 1，在 data_addr_ok 时降为 0，其他情况保持原值。

```
always @(posedge clk) begin
    data_req <= (!resetn) ? 1'b0 :
            (exe_pc != exe_pc_reg) &&(exe_is_load || exe_is_store) ? 1'b1 :
            data_addr_ok ? 1'b0 : data_req;
end
```

修改后……错误提前了，又回到了四个 sw 连着的那个位置，而且发现之前对其实是侥幸过了,实际并没有写进去，而且怎么都改不对。

转换桥中各个信号之间互相关联太多，为了一个目的修改其中一个值会导致其他的不对，我一开始用的是自己新写的优化桥，读写是分离的，sw 的 exe 级需要取指和写数据都完成后再继续流水，我试图修改 block 信号，可是要不中间有一个空拍又死回了上一个问题，要不然 block 太久流水卡住了，尤其是读写同时的时候更难弄。

感觉用老版的一次只能读/写一件事的桥会更好管理一些，就换成了那个转换桥，可是根本问题没解决，四个

sw 还是过不去。

只改 block 信号这个思路走进了死胡同，后来想出来了一个用一个 wait 信号判断这一级要做几件事，取指令一定要做，可能会有一个写数据或者取数据。我在这一级到来的时候将 wait 信号置成要完成的操作数，每完成一次 wait 减 1,在 wait 减到 0 之前都是阻塞状态，这种写法总真正地通过了四个 sw。

```
always @(posedge clk) begin
    if (~resetn) begin
        wait_axi <= 2'b0;
    end
    else if (exe_pc != exe_pc_reg && (exe_is_load || exe_is_store)) begin
        wait_axi <= 2'b10;
    end
    else if (exe_pc != exe_pc_reg) begin
        wait_axi <= 2'b01;
    end
    else if ((data_data_ok||inst_data_ok)&&(wait_axi!=2'b0))begin
        wait_axi <= wait_axi - 1;
    end
end
 assign axi_block = (inst_block)||(wait_axi!=2'b0);
```

5、错误 5

（1）错误现象

```
[  18898 ns] Error!!!
  reference: PC = 0xbfc00cf0, wb_rf_wnum = 0x0a, wb_rf_wdata = 0xffffffff
  mycpu    : PC = 0xbfc00cf0, wb_rf_wnum = 0x0a, wb_rf_wdata = 0x25080001
```

（2）分析定位过程

lw 指令要取的数据在 exe 级就已经读到了，但是在 mem 级才输入输入，中间历经很多拍但是数据没有保存导致丢失。mem 级取到的数是之后新读的指令。

（3）错误原因

现在每一级都有很多拍，lw 取的数据在 exe 级就已经读出，但是没有保存，导致在 mem 级时该数据已经丢失。

（4）修正效果

在 mem 级加一个寄存器，将读到的数据存住，直到下一拍来临才会更新。修改后这个问题解决。

6、错误 6

（1）错误现象

[19028 ns] Error!!!

reference: PC = 0xbfc00d00, wb_rf_wnum = 0x08, wb_rf_wdata = 0x00000001

mycpu : PC = 0xbfc00380, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00000000

（2）分析定位过程

触发例外，估计是跟取值相关，于是看总线里的 rdata 信号，发现十分混乱，取的指令和数据混杂，取出的数据是 0xffffff 被误认为时指令，触发保留指令例外。

（3）错误原因

没有区分取出的 rdata 是给 data 的还是给 ram 的，原代码如下：

```
assign inst_rdata = rdata;

assign data_rdata = rdata;
```

（4）修正效果

将 inst_rdata 和 data_rdata 都改成 reg 类型，在得到取的指令/数据时存住，直到下一次读出指令/数据再更新，修改后终于把 bfc00cf0 过去了。

```
always @(posedge clk) begin
    inst_rdata <= (!resetn) ? 32'h0 :
                  (inst_data_ok) ? rdata : inst_rdata;
    data_rdata <= (!resetn) ? 32'h0 :
                  (data_data_ok) ? rdata : data_rdata;
end
```

7、错误 7

（1）错误现象

[145104 ns] Error!!!

reference: PC = 0xbfc847bc, wb_rf_wnum = 0x04, wb_rf_wdata = 0x800d34c4

mycpu : PC = 0xbfc847bc, wb_rf_wnum = 0x04, wb_rf_wdata = 0x00000000

（2）分析定位过程

```
bfc847b8:       8c8566a8        lw    a1,26280(a0)

bfc847bc:8ca466a8       lw    a0,26280(a1)
```

连续两条 lw 是数据相关，原来的 de_block 阻塞是让第二个 lw 等待到第一个 lw 的 mem 级数据回来再继续，但是现在 exe 级的时候数据就回来了，而且每一级都有很多拍，数据前递取回来的数是错的。

（3）错误原因

原 CPU 的 lw 的数据在 mem 级回来，现在 exe 级就回来了，数据前递也要相应更改。

（4）修正效果

lw 相关时的让操作数 1 直接取 data_rdata 的值，修改后此问题解决。

8、错误 8

（1）错误现象

[471524 ns] Error!!!

reference: PC = 0xbfc778f4, wb_rf_wnum = 0x17, wb_rf_wdata = 0x800c8123

mycpu : PC = 0xbfc77d34, wb_rf_wnum = 0x09, wb_rf_wdata = 0x10000000

（2）分析定位过程

lw 后跟一条 bne。两个操作数都是之前的 lw 的写回寄存器，第一个操作数取值正确了，bneq 使用 fwd_data2 和第一个操作数进行比较进行比较，fwd_data2 没有处理过这种数据相关导致本应相等的现在不等，跳转错误。

（3）错误原因

第二个操作数也出现类似错误 7 的数据相关。

（4）修正效果

如错误 7 在数据相关时直接取 data_rdata 的值即可。

9、错误 9

（1）错误现象

----[1368594 ns] Number 8'd43 Functional Test Point PASS!!!

[1372000 ns] Test is running, debug_wb_pc = 0xbfc7ab88

[1382000 ns] Test is running, debug_wb_pc = 0xbfc7ab88

[1392000 ns] Test is running, debug_wb_pc = 0xbfc7ab88

（2）分析定位过程

在除法处卡死，除法运算结束后还在 block，原因是在除法的 block 阶段一直在取指，取完一次又取一次，导使 inst_block 信号反复跳起，恰好补上了除法运算结束时 de_block 的缺,导致流水卡死。

而不停取指的原因是 inst_req 一直是 1，有 req 就代表有操作要做，但实际上在除法阻塞的过程中，取指一次就够了。

（3）错误原因

inst_req 一直赋值为 1，导致取指结束但是除法没结束的阻塞阶段一直在取指，block 一直是 1，流水无法继续。

（4）修正效果

一开始将 inst_req 信号写的跟 data_req 差不多，将它改成 reg 类型，在每一级刚开始的时候赋值成 1，取指结束后赋值成 0，其他情况保持原值。inst_req 是在 nextpc_gen 模块里赋值的，所以需要给这个模块添加时钟信号。之后发现取指的任务要在这一级 fe_pc 前一拍发出，否则就会出现 block 耦合不上的问题。

always @(posedge clk) begin

```
inst_keep <= (!resetn) ? 1'b1 :
        (!resetn_reg) ? 1'b1 :
        (nextpc != fe_pc)  ? 1'b1 :
        inst_data_ok ? 1'b0 : inst_keep;
end
assign inst_req = (nextpc != fe_pc) || inst_keep;
```

后来出现问题，因为现在取指完成的时间不一定，有可能会出现取指比除法完成得还慢的情况，此时就需要在除法结束下一级到来之前后 de_block 持续变成 0，处理方法是将 complete 信号用一个 reg 类型保存下来，在下一级的时候该 reg 信号降成 0。

10、错误 10

（1）错误现象

在除法测试中，所有的 mfhi 和 mflo 都是紧挨着 div 指令之后进行的，用的全都是前递的数据，具体现象没有记录下来，但除法结果是有误的。

（2）分析定位过程

除法器的设计是 complete 当拍除法器结果正确，但由于随机延迟，现在的每一级时间都较长，可能不能在 complete 当拍就存下除法结果，需要考虑结果的延续，所以添加若干个寄存器以完成结果延续的目的。

（3）错误原因

除法的结果前递设计有误。

（4）修正效果

将除法与紧接着的 mfhi 或 mflo 的数据前递设计为如下：

```
(de_is_mfhi && complete) ? div_result[31: 0]:
(de_is_mflo && complete) ? div_result[63:32]:
(de_is_mfhi && complete_keep && exe_is_div) ? div_result_reg[31: 0]:
(de_is_mflo && complete_keep && exe_is_div) ? div_result_reg[63:32]:
```

前两项是 complete 当拍的数据前递，complete_keep 一直持续到 exe 级的除法结束，由于 div_result 是 wire 类型的变量，用 div_result_reg 将 div_result 延续下来，在 exe 级的剩余时间都前递该变量。

将除法与间隔一条的 mfhi 或 mflo 的数据前递设计为如下：

```
(de_is_mfhi && (mem_is_div||mem_is_mul)) ? mem_value:
(de_is_mflo && (mem_is_div||mem_is_mul)) ? mem_value:
```

即在 mem 级是除法时直接将 mem_value 前递。

而 mem_value 是如下设计的：

```
assign mem_value = (mem_is_load) ? unaligned_wdata :
```

```
(mem_is_div && de_is_mfhi) ? mem_hi_data:
(mem_is_div && de_is_mflo) ? mem_lo_data:
(mem_is_mul && de_is_mfhi) ? mem_hi_data:
(mem_is_mul && de_is_mflo) ? mem_lo_data:mem_exe_value;
```

mem_hi_data 和 mem_lo_data 的设计分别如下：

```
always @(posedge clk) begin
    mem_hi_data <= (~resetn)? 32'b0:
        (complete) ? div_result[31:0]:
        (mem_is_mul && ~wb_is_mul) ? mul_result[63:32]: mem_hi_data;
end

always @(posedge clk) begin
    mem_lo_data <= (~resetn)? 32'b0:
        (complete) ? div_result[63:32]:
        (mem_is_mul && ~wb_is_mul) ? mul_result[31:0]: mem_lo_data;
end
```

即在除法 complete 的时候更新寄存器内的值，其余时候保持。全部更改后除法测试通过。

11、错误 11

（1）错误现象

同除法测试，在乘法测试中，所有的 mfhi 和 mflo 都是紧挨着乘法之后进行的，所以用的全都是前递的数据，具体现象没有记录下来，但乘法结果是有误的。

（2）分析定位过程

乘法的流水级是 2，但这里的 2 是指两个时钟拍而非两个流水级，所以在 exe 级把数据送给乘法器后两拍乘法的结果就出来了，我们选择把它延续到 mem 级后两拍使用。

（3）错误原因

乘法结果出来的时机从 wb 级变成了 mem 级，所以一系列前递和阻塞信号都需要更改。

（4）修正效果

在 de_block 中添加了乘法相关的情况如下：

```
((exe_inst[31:26]==6'b000000) && (exe_inst[5:1] == 5'b01100) && (de_is_mfhi || de_is_mflo) && ~(mem_is_mul_reg))
```

即 exe 级是乘法，de 级是 mfhi 或者 mflo，且乘法结果已经在 mem 级存在两拍前均阻塞，这样就只存在如除法中 mem_value 的前递，相应代码已在除法中列出。全部更改后乘法测试通过。

12、错误 12

（1）错误现象

在随后的某个延迟槽测试中，出现了乘法之后很久才 mflo 或 mfhi 的情况，此时取出的数据有误。

（2）分析定位过程

观察 hi 或 lo 寄存器，发现其被写入了多次，于是决定将写使能改为 1 拍，并且观察应将乘法结果延续到哪一拍才能正确写入。

（3）错误原因

没有对 hi 和 lo 寄存器正确写入。

（4）修正效果

由于 hi 或者 lo 在 wb 级的第一拍就已经成功写入了，所以设定一个 wb_inst_change 信号如下：

```
assign wb_inst_change = (mem_inst_reg2!=mem_inst_reg);
```

并将 wen 信号与 wb_inst_change 相与，从而设定 hi_wdata 和 lo_wdata 的数据如下：

```
assign HI_wen   = (~exc_handler_reg3 && ~exc_handler_reg4)&&(is_mthi | is_mul & wb_inst_change | (is_div & complete));

assign LO_wen   = (~exc_handler_reg3 && ~exc_handler_reg4)&&(is_mtlo | is_mul & wb_inst_change | (is_div & complete));

assign HI_wdata = ({32{is_mthi}} & mem_value_reg) | ({32{is_mul}} & mul_div_result_reg2[63:32]) | ({32{is_div}} & mul_div_result[31: 0]);

assign LO_wdata = ({32{is_mtlo}} & mem_value_reg) | ({32{is_mul}} & mul_div_result_reg2[31: 0]) | ({32{is_div}} & mul_div_result[63:32]);
```

乘法用的是存到 wb 级的数据。全部更改后该延迟槽测试通过。

13、错误 13

（1）错误现象

```
[1949171 ns] Error!!!

    reference: PC = 0xbfc00380, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00004000

    mycpu    : PC = 0xbfc00384, wb_rf_wnum = 0x1b, wb_rf_wdata = 0x00000000
```

（2）分析定位过程

经过一番辗转，我们发现在 380 的 de 级取出来 inst 是 0，其原因是在 next_pc 中一看到 exc_handler 就将 pc 跳到了 380，没有等到取到 380 的指令后再继续执行，pc=bfc00380 处的指令没有执行，自然不能抓到写回级的数据，所以发生错误。

（3）错误原因

在 next_pc 中把 exc_handler 的条件放到了阻塞条件之前，导致取不出 380 处的指令。

（4）修正效果

调整两条指令的顺序后可以成功取指。

14、错误 14

（1）错误现象

[1952844 ns] Error!!!

reference: PC = 0xbfc00380, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00000022

mycpu : PC = 0xbfc00380, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00000000

（2）分析定位过程

观察 test.s 发现 syscall 后面跟着一条乘法，由于 de 级中添加了乘法的数据相关，有 mem_is_mul 信号，使得 de_vsrc1 由于被取消的乘法发生了错误的数据相关，用的是 mem_value 的值而非从 hi 寄存器中读出的值。

（3）错误原因

发的空操作没有清空 inst，而 de 级是根据 inst 做相关判定的，所以出错。

（4）修正效果

将空操作的 inst 设置为 32'b0 后运行正确。

15、错误 15

（1）错误现象

----[931144 ns] Number 8'd80 Functional Test Point PASS!!!

[932000 ns] Test is running, debug_wb_pc = 0xbfc66408

[942000 ns] Test is running, debug_wb_pc = 0xbfc66408

[952000 ns] Test is running, debug_wb_pc = 0xbfc66408

（2）分析定位过程

观察波形发现 count 和 compare 相等时 cpu 并没有处理时钟中断，这是由于在原来的设计中 int_sig 只有一拍，导致没有等到 axi_block 下来中断信号就结束了，所以没有进入中断处理程序。所以我们考虑将中断信号延续到 axi_block 下来一拍以后再降下来。

（3）错误原因

int_sig 不够长，导致 exc_handler 不够长，无法进入时钟中断处理。

（4）修正效果

添加 int_sig_keep 信号如下：

```
always @(posedge clk) begin
    if (~resetn) begin
        int_sig_keep <= 1'b0;
    end else if(int_sig) begin
        int_sig_keep <= 1'b1;
    end else if(axi_block == 1'b0) begin
        int_sig_keep <= 1'b0;
    end
```

end

同时将 exc_handler 或上 int_sig_keep 信号如下：

assign exc_handler = int_sig || int_sig_keep || exc_sig;

修改后可以成功进入时钟中断。

16、错误 16

（1）错误现象

[937011 ns] Error!!!

reference: PC = 0xbfc005d0, wb_rf_wnum = 0x1a, wb_rf_wdata = 0xbfc665b4

mycpu : PC = 0xbfc005d0, wb_rf_wnum = 0x1a, wb_rf_wdata = 0xbfc665b0

（2）分析定位过程

由于 int_sig 置一后的下一拍 EXL 会马上置一，EXL 置一后就不能再更新 EPC，wt_epc 是由 exe 级给出的信号，EXL 置一前只有一拍可以写，此时的 wt_epc 不一定是正确的。所以要等到下一次切换指令之前的最后再更改 epc，这个时刻的标志就是 int_sig_keep&&~axi_block，所以在原来的~status_EXL 条件上或一个该条件，让指令在该情况下也可以改写 epc。

（3）错误原因

时钟中断写 epc 的时间没有延长到这一级末尾，导致写 epc 的值早了一拍。

（4）修正效果

将 epc 改为如下赋值：

```
else if((~status_EXL|| int_sig_keep&&~axi_block )&&(exc_sig || int_sig ||int_sig_keep))
    cp0_epc_reg <= wt_epc;
```

之后该时钟中断的过程正确。

17、错误 17

（1）错误现象

[937584 ns] Error!!!

reference: PC = 0xbfc005d0, wb_rf_wnum = 0x1a, wb_rf_wdata = 0xbfc66604

mycpu : PC = 0xbfc005d0, wb_rf_wnum = 0x1a, wb_rf_wdata = 0xbfc665fc

（2）分析定位过程

观察 test.s 发现最后一次时钟中断竟然给 compare 和 count 直接写了一样的值，而我们的 cpu 在 de 级的第一拍就把 count 写进去了，第二拍 TI 置 1，第三拍 IP7 置一，int_sig 置一，此时对应的 exe 级还是 mtc0 上一条的 nop。而实际上 mtc0 count 应该已经被执行完，epc 中存储的应该是 mtc0 下一条的 b 指令。

（3）错误原因

根本的错误原因是 cp0 内部的变量是拍驱动而非指令驱动的，TI 置一得过早，int_sig 置一得过早，导致 epc 写了错误的值。

（4）修正效果

将 TI 在非 axi_block 时置零，int_sig 在指令更改时再置一，更改后时钟中断测试通过。

18、错误 18

（1）错误现象

[938384 ns] Error!!!

reference: PC = 0xbfc005d0, wb_rf_wnum = 0x1a, wb_rf_wdata = 0xbfc10410

mycpu : PC = 0xbfc005d0, wb_rf_wnum = 0x1a, wb_rf_wdata = 0xbfc1040c

（2）分析定位过程

此时是软件中断测试。

写 epc 寄存器对应的代码为

```
else if(~status_EXL&&(cause_IP1&&status_IM1||cause_IP0&&status_IM0))
    cp0_epc_reg <= wt_epc+4;
```

与时钟中断写 epc 的错误类似，软件中断也需要将写的时间延长到这一级末尾。int_sig 是对应所有中断处理的，而软件中断只有 IP1 和 IP0 位，因此对应添加 soft_int_sig 和 soft_int_keep 信号作为判断条件。

（3）错误原因

软件中断写 epc 的时间没有延长到这一级末尾，导致写的 epc 的值早了一拍。

（4）修正效果

添加 soft_int_sig 和 soft_int_keep 信号：

```
assign soft_int_sig = status_IE && (~status_EXL) && exe_pc_change
            && (cause_IP1&&status_IM1||cause_IP0&&status_IM0);
if (~resetn) begin
  soft_int_keep <= 1'b0;
end else if(soft_int_sig) begin
  soft_int_keep <= 1'b1;
end else if(axi_block == 1'b0) begin
  soft_int_keep <= 1'b0;
end
```

修改软件中断 epc 的写回条件为：

```
else if((~status_EXL|| soft_int_keep&&~axi_block)&&(soft_int_sig||soft_int_keep))
    cp0_epc_reg <= wt_epc+4;
```

修改后软件中断测试通过！

19、错误 19

（1）错误现象

----[1024396 ns] Number 8'd92 Functional Test Point PASS!!!

[1032000 ns] Test is running, debug_wb_pc = 0x800d0000

[1042000 ns] Test is running, debug_wb_pc = 0x800d0000

[1052000 ns] Test is running, debug_wb_pc = 0x800d0000

（2）分析定位过程

此处在中断例外的 jr 延迟槽,在出现问题之前相关代码如下：

bfc41254:　　3c08800d　　lui　t0,0x800d

.....

bfc41298:　　01000008　　jr　　t0

bfc4129c:0000000c　　syscall

jr 跳转到的地址会取出一个非法指令，造成保留指令例外。而它的延迟槽里是一条 syscall 例外。它卡在那的原因是在 syscall 例外的处理信号报出后又报了一个保留指令例外，但是按道理这一条保留指令例外是应该被取消的。一开始我们误以为是应取消它的取指结果，修改后后面有更多的问题，后来意识到应该是取消这个保留指令例外的信号，它的操作本身已经被取消了。

（3）错误原因

syscall 后只取消了下一条指令的具体操作，没有它的例外信号，导致例外多报。

（4）修正效果

修改为：如果当前是例外发生的的后两条指令，就将其他例外信号的寄存器赋值为 0。

修改后无随机种子的仿真通过。

20、错误 20

（1）错误现象

[1252984 ns] Error!!!

reference: PC = 0xbfc371e4, wb_rf_wnum = 0x16, wb_rf_wdata = 0xffff0000

mycpu　: PC = 0xbfc371e4, wb_rf_wnum = 0x16, wb_rf_wdata = 0x80000000

（2）分析定位过程

相关指令如下：

bfc371dc:00800013　　mtlo a0

bfc371e0:0109001a　　div　zero,t0,t1

bfc371e4:0000b012　　mflo s6

除法结果计算出来后就是 mflo，此时应该将除法的结果前递，但是同时它也和出发之前的 mtlo 发生数据相关。原有代码的数据相关在在 exe 和 mem 都有相关的情况下，选取了 mtlo 的前递值，但是应该选用 mem 级的数据前递。

（3）错误原因

在除法结果的 mem 级数据前递和 mtlo 的 exe 级数据前递中，本应选最近的 mem 级数据，但是由于选择器的顺序写反了，导致取错了前递数据。

（4）修正效果

调整选择顺序，优先询问是否满足 mem 级前递的条件。修改后有默认随机种子的仿真通过。

21、错误 21

（1）错误现象

综合出现 critical warning，报了一个 timing loop 组合环。

报的位置在 nextpc_gen.v 文件的 46 行:

```
assign nojump = ~(de_br_taken | de_br_is_j |de_br_is_jr | de_is_eret | exc_handler);
```

（2）分析定位过程

检查了它报的这一行，貌似没有发现组合环，然后采用群里的同学分享的方法重新综合报出组合环的具体路径，果然找到了一条。

nextpc:

```
assign nextpc = (~resetn) ? 32'hbfc00000:
                (~resetn_reg) ? 32'hbfc00000:
                (de_block||inst_block||data_block||axi_block)? fe_pc:
                (exc_handler)? 32'hbfc00380:
                        C1 | C2 | C3 | C4 | C5 ;
assign inst_req = (nextpc != fe_pc) || inst_keep;
```

interface:

```
assign inst_block = !type && inst_req && req_reg;
```

上述代码中有一个 nextpc->inst_req->inst_block->nextpc 的组合环。

inst_block 中的 inst_req 本就是一个不必要的条件，直接删去即可破坏掉环。

（3）错误原因

assign 时一个不必要的条件导致了一个组合环。

（4）修正效果

将 inst_block 修改为：

```
assign inst_block = !type && req_reg;
```

综合后没有 critical warning 了，且上板通过。

四、实验总结

（一）组员：姚子薇

哈哈哈哈哈哈哈哈哈哈哈啊哈哈哈哈哈哈哈哈啊哈哈哈哈哈哈哈哈哈哈哈哈哈啊啊啊终于过啦！！！

（二）组员：陈乐滢

哈哈哈哈哈哈哈哈哈哈哈啊哈啊哈哈哈哈哈哈哈哈哈啊哈哈哈哈哈哈哈哈哈上板过啦！！！！！