实验 11 报告

学号：2017K8009922026，2017K8009922032
姓名：康齐瀚，赵鑫浩
箱子号：63

一、实验任务（10%）

完成 AXI-类 SRAM 转接桥到 CPU 的集成，在生成随机信号种子的情况下通过 lab9 的总共 94 个测试样例以及完成上板测试。

AXI-类 SRAM 转接桥到 CPU 的集成需要将 CPU 在 IF 级的访问 inst-sram 接口改造为类 SRAM 接口，以及 EXE 和 MEM 级的访问 data sram 接口改造为类 SRAM 接口。两个接口作为 AXI-类 SRAM 转接桥的输入信号，并输出 AXI 协议多个通道的信号。最终满足整个 CPU 对外访存通道都是 AXI 协议规定的信号。

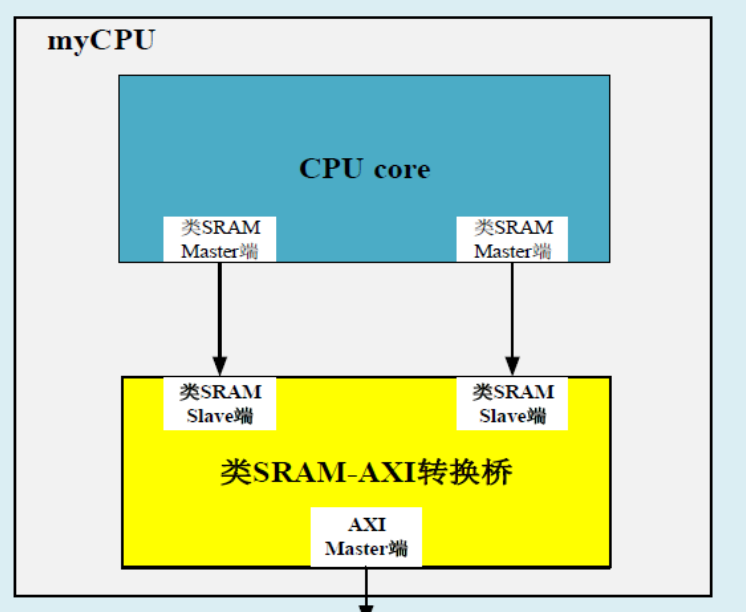
二、实验设计（40%）

（一）总体设计思路

将 IF 级的取指信号改造为类 SRAM 的 inst_req 信号。整个 IF 级可以划分为 Pre-IF 级以及 IF 级，其中 Pre-IF 级表示取指地址被接收，信号是 inst_addr_ok，IF 级的 ready_go 取决于 data_ok 信号或者是 inst_buf 中的数据有效(valid)。

data 的请求需要在 EXE 级与 MEM 级进行，其中 EXE 级负责发出地址，发出写数据等，并负责接收 addr_ok 信号后流入下一个流水级。MEM 级负责接收来自 data-sram 的数据(如果该指令是 load 型指令的话)。

在例外处理上，需要在 Pre-IF 级配置 cancle 信号，使得发生例外时新取出的指令都是无效的：直到例外处理地址 0xbfc00380 被接收为止。



由于本次设计过于复杂，因此设计图暂时使用任务书提供的简化设计图。

（二）重要模块 1 设计：IF 级和 Pre-If 级的接口改造及对于例外的处理。

1、工作原理

因为从 AXI-RAM 中取到的无论是数据还是指令，都不能维持很长时间，在 inst_addr_ok 时，我们会收到 RAM 中取出的数，此时，我们需要寄存器将该地址存入寄存器并从 Pre-If 级传入 if 级，得到真正的 nextpc，配合 nextpc，我们需要相应的改变 fs_ready_go,to_fs_valid 信号，因为替换 RAM 之后，我们需要的指令可能会多拍之后再返回，所以在这期间，流水线需要停下等待指令，直到 inst_addr_ok 拉高，表示指令地址已经收到，而 fs_ready_go 需要等到 inst_data_ok，表示已经收到指令。

2、功能描述

我们首先需要加入存 pc 的寄存器来保证取回的指令不会因为多拍返回而丢失，我们使用信号 buf_valid 来表示当前寄存器是否存着未使用的 pc，如果没有，那么就将 nextpc 存入，但是存在跳转指令和例外处理，我们需要分开考虑这两种情况，处理方法类似。

```
always@(posedge clk) begin
    if(reset) begin
        buf_valid <= 1'b0;
    end
    else if(to_fs_valid && fs_allowin) begin
        buf_valid <= 1'b0;
    end
    else if(!buf_valid) begin
        buf_valid <= 1'b1;
    end
    if(!buf_valid) begin
        if(ex_addr_buf_valid) begin
            buf_npc <= ex_addr_buf;
        end
        else if(branch_buf_valid) begin
            buf_npc <= branch_buf;
        end
        else begin
            buf_npc <= nextpc;
        end
    end
end
```

我们在考虑跳转指令时，由于跳转地址是从 id 级传回，所以我们需要设置一个单独的寄存器来存跳转地址，还需要一个信号控制是否取到跳转地址。

```
always@(posedge clk) begin
    if(reset) begin
        branch_buf_valid <= 1'b0;
    end
    else if(br_taken) begin
        branch_buf_valid <= 1'b1;
    end
    else if(branch_buf_valid && !buf_valid) begin
        branch_buf_valid <= 1'b0;
    end

    if(br_taken) begin
        branch_buf <= br_target;
    end
end
```

在处理例外时，思路和处理跳转指令时基本相同，但需要注意的是，我们从 wb 级得到例外信号传入前面各级，并刷新流水线，但在 AXI-RAM 中，我们需要多拍后才能拿到想要的例外指令，在此期间，我们需要保证流水线等待例外指令，所以我们引入一个新的信号 cancel 来保证正确性，其他部分信号的写法与处理跳转指令类似。

```
always@(posedge clk) begin
    if(reset) begin
        ex_addr_buf_valid <= 1'b0;
    end
    else if(ex_occur || eret) begin
        ex_addr_buf_valid <= 1'b1;
    end
    else if(ex_addr_buf_valid && !buf_valid) begin
        ex_addr_buf_valid <= 1'b0;
    end

    if(ex_occur) begin
        ex_addr_buf <= 32'hbfc00380;
    end
    else if(eret) begin
        ex_addr_buf <= epc_reg;
    end
end
```

```
always@(posedge clk) begin
    if(reset) begin
        cancle <= 1'b0;
    end
    else if(~ex_addr_buf_valid && inst_sram_data_ok) begin
        cancle <= 1'b0;
    end
    else if(ex_occur || eret) begin
        cancle <= 1'b1;
    end
end
```

除了上面这些，我们还需要将 SRAM 的接口进行改造，下面是一些和之前不同的信号。

```
assign inst_sram_wr   = 1'b0         ;
assign inst_sram_size = 2'b10        ;
assign inst_sram_wstrb = 4'h0        ;
assign inst_sram_wdata = 32'b0       ;
```

（三）重要模块 2 设计：load，store 模块

1、工作原理

Load，store 的指令执行在 exe 和 mem 级执行，类似 pre-IF 和 IF 级，我们如果需要从 RAM 在取出数据，那么我们在 exe 级 data_addr_ok 时得到取数的地址以及 size，wstrb 等信号，将这些信号传入 mem 级进行取数，这里需要注意的是，我们存数取数需要多个周期，所以在执行这些指令时，我们需要将前面几级的 ready_go 信号置 0，等到存取数结束后恢复。

2、功能描述

```
always@(posedge clk) begin
    if(reset) begin
        buf_addr_valid <= 1'b0;
    end
    else if(es_to_ms_valid && ms_allowin) begin
        buf_addr_valid <= 1'b0;
    end
    else if(!buf_addr_valid && (is_store || is_lw) && es_valid) begin
        buf_addr_valid <= 1'b1;
    end

    if(!buf_addr_valid && (is_store || is_lw) && es_valid) begin
        buf_addr <= es_alu_result;
    end
end
```

我们在遇到 store 或者 load 指令时，需要使用一个寄存器和他的 valid 信号来保存地址，便于后续操作中使用。

```
always@(posedge clk) begin
    if(reset) begin
        rdata_buf_valid <= 1'b0;
    end
    else if(ms_to_ws_valid && ws_allowin) begin
        rdata_buf_valid <= 1'b0;
    end
    else if(!rdata_buf_valid && !ws_allowin && is_lw) begin
        rdata_buf_valid <= 1'b1;
    end

    if(!rdata_buf_valid && !ws_allowin && is_lw) begin
        rdata_buf <= mem_result;
    end
end
```

还需要注意的一点是，因为我们的存取数需要多个时钟周期才能完成，对于一些需要多周期保存的信号，我们可以与上该级的 valid 信号，这样直到该信号被送到下级流水前不会消失，相应的我们需要修改每级的 ready_go 信号，使得多级延迟存取时保持流水线的阻塞状态。

（四）重要模块 3 设计：对于顶层接口的改造

我们在该实验中需要将之前的类 sram 接口改造成 AXI 接口，我们需要在各级流水线中涉及 ram 的信号修改，并添加握手信号，顶层模块中我们也需要增加 AXI 的 5 组握手信号，并与之前写的转接桥相连，确保功能的正常使用。

三、实验过程（50%）

（一）实验流水账

2019/11/21 11：30 ~ 21：30 完成接口改造，顺利开始仿真，不过一直卡在第一个测试点…

2019/11/22 9：30 ~ 21：30 顺利通过随机种子的仿真测试，不过综合时报错

2019/11/23 8：30 ~ 9：30 发现综合报错原因是由于一个组合环，改掉组合环，成功综合并上板

（二）错误记录

1、错误 1：PC 值错误

（1）错误现象

第一个测试样例中，PC 值对比出错，从而导致整条指令执行出错。如下图所示：

```
[   2307 ns] Error!!!
    reference: PC = 0xbfc006b8, wb_rf_wnum = 0x04, wb_rf_wdata = 0xbfb00000
    mycpu    : PC = 0xbfc00010, wb_rf_wnum = 0x08, wb_rf_wdata = 0x80000000
```

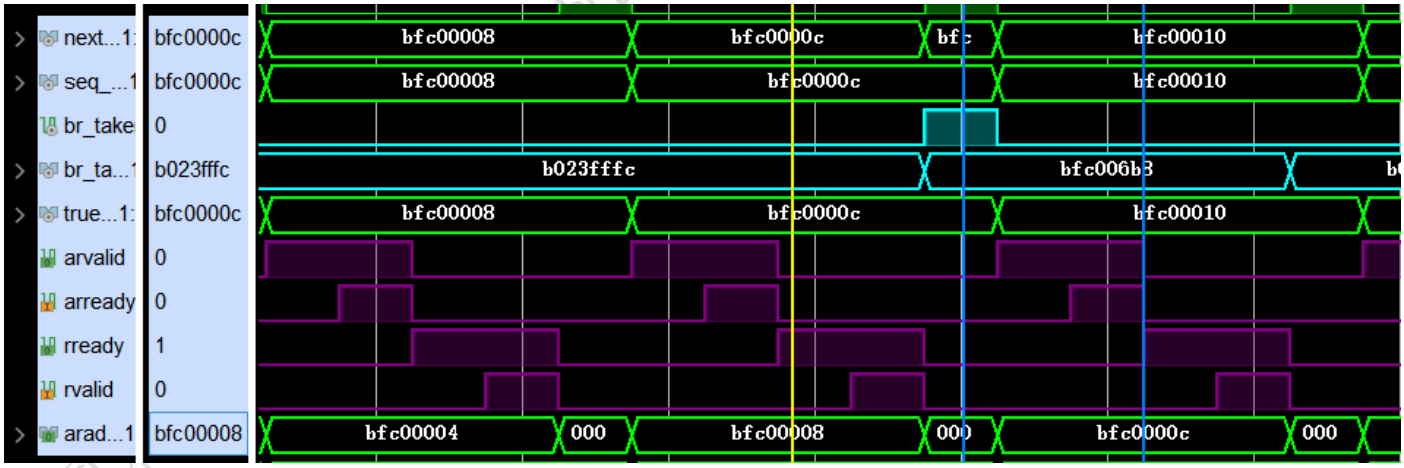
（2）分析定位过程

0xbfc00008 是一条跳转指令，如下图所示：

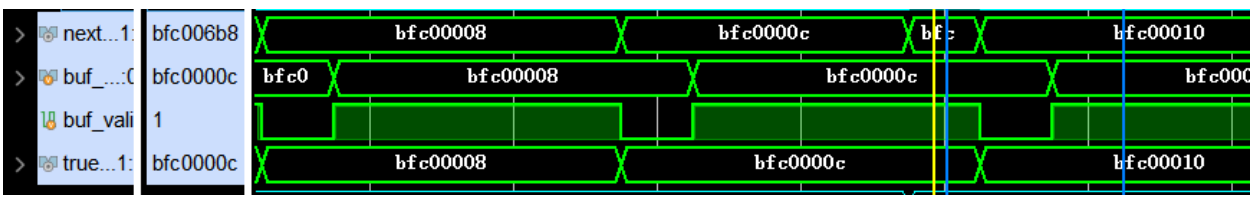
```
/media/sf_xjz/class/ucas/ucas19_20/jiafan_20191014/lab8-9/func_lab9/start.S:26
bfc00008:   100001ab    b   bfc006b8 <locate>
/media/sf_xjz/class/ucas/ucas19_20/jiafan_20191014/lab8-9/func_lab9/start.S:27
bfc0000c:   00000000    nop
/media/sf_xjz/class/ucas/ucas19_20/jiafan_20191014/lab8-9/func_lab9/start.S:30
bfc00010:   3c088000    lui t0,0x8000
```

此条指令跳转的目标是 0xbfc006b8，而提供的 ref 也是 0xbfc006b8。按照正常的执行流程是 0xbfc00008, 0xbfc0000c, 0xbfc006b8。因此发生此次错误的原因是没有成功跳转。

观察波形，如下图所示：



可以看出，在第一个 marker 的位置，实际上 br_taken 是被拉高了的，说明该指令判断要发生跳转，计算的地址也是正确的，是 0xbfc006b8, 不过该信号仅仅保持了一个周期后就消失了。就保持的一个周期的这个时间来看，还不足以使得 tru_npc 发送出正确的地址信号。因此发生了这样的错误。



再观察 buf_npc 的值可以发现，0xbfc00380 这个位置根本没有被存进 buf_npc 这个寄存器中。而根据老师提供的代码的赋值逻辑：

```
always@(posedge clk) begin
    if(reset) begin
        buf_valid <= 1'b0;
    end
    else if(to_fs_valid && fs_allowin) begin
        buf_valid <= 1'b0;
    end
    else if(!buf_valid) begin
        buf_valid <= 1'b1;
    end
    if(!buf_valid) begin
        buf_npc <= nextpc;
    end
end
```

```
assign true_npc = buf_valid ? buf_npc : nextpc;
```

当 nextpc 恰好为 0xbfc00380 的时候，buf_valid 恰好不是 0，因此 true_npc 永远不会是 0xbfc00380,导致了错误。

（3）错误原因

跳转指令算出的跳转地址没有被保存下来，导致跳转位置的 PC 无法发出取指请求，从而导致了错误。

（4）修正效果

需要用一个新的 buf 保存下来跳转指令的目标地址，为它配置对应的 valid 标志。当现在 buf_npc 的 valid 无效并且跳转地址缓冲区中的 valid 有效的时候，将跳转地址缓冲区中的数据存入 buf_npc 中即可。具体实现如下：

```
always@(posedge clk) begin
    if(reset) begin
        branch_buf_valid <= 1'b0;
    end
    else if(br_taken) begin
        branch_buf_valid <= 1'b1;
    end
    else if(branch_buf_valid && !buf_valid) begin
        branch_buf_valid <= 1'b0;
    end

    if(br_taken) begin
        branch_buf <= br_target;
    end
end
```

该 branch_buf_valid 在确实发生跳转，即 br_taken 有效的时候被保存下来，并置相应的 valid 位置。当目前 branch_buf_valid 有效且 buf_valid 无效的时候，需要将现在的缓冲区有效位设置为 0，因为数据已经被写入到了 buf_npc 中了。

在进行了这样的处理之后，能成功进行跳转了。

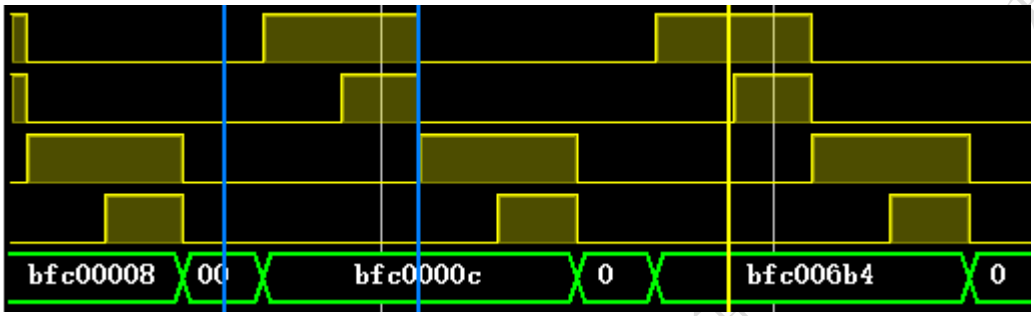
（5）归纳总结（可选）

写代码之前没有考虑清楚。

2、错误 2：PC 数值出错

（1）错误现象

该错误并不是由 test bench 提供的参考功能发现的。是在观察波形图的运行时 PC 值时发现的错误。承接上一个 bug，虽然在增加了 branch_buf 之后能完成跳转，但跳转位置错误了。



跳转位置应该是 0xbfc006b8, 但是却是 0xbfc006b4。

（2）分析定位过程

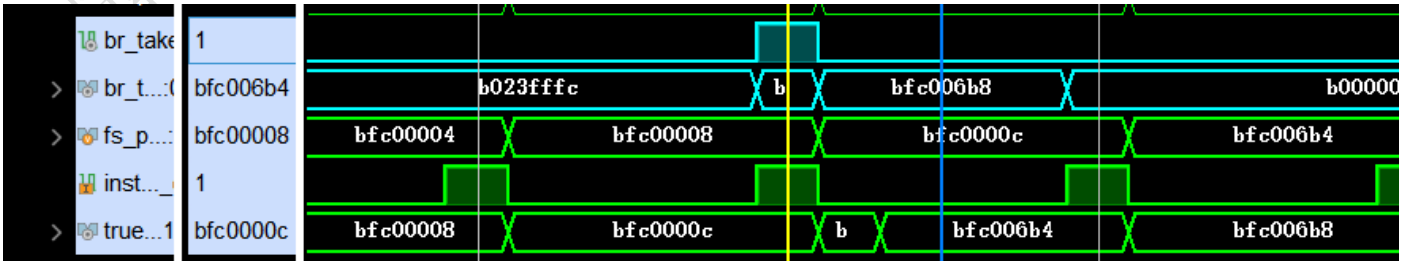
观察 br_target 的值，如下图所示：

该值为 0xbfc006b4, 因此发生了错误。需要找到 br_target 生成的地方：

```
assign br_target = (inst_beq || inst_bne || inst_bgez || inst_bgtz || inst_blez || inst_bltz || inst_bltzal || inst_bgezal)
                                            (fs_pc + {{14{imm[15]}}, imm[15:0], 2'b0}) :
                   (inst_jr   || inst_jalr)              ? rs_value :
                   /*inst_jal*/              {fs_pc[31:28], jidx[25:0], 2'b0};
```

br_target 的值来源于 fs_pc, 而 fs_pc 的值来源于 fs_to_ds_bus 的最低 32 位，从 IF 级的生成关系来看，fs_to_ds_bus 的最低 32 位是 IF 级的 fs_pc 寄存器的值。

在 CPU 设计中，fs_pc 是指当前正在 FS 级等待读回指令的 PC 值，在未加入 AXI 总线设计之前，当一条指令进入 ds 级之后，它的下一条指令 PC 也进入了 fs 级等待指令返回。但是在加入了 AXI 总线设计之后，这不再一定正确了，因为可能一条指令进入 ds 级之后它的下一条指令仍然在等待 addr_ok 返回而未进入 fs 级。如下图所示：



可以看出 0xbfc0000c 并没有还没有进入 fs_pc，所以 ID 级将 fs_pc 认为是延迟槽中的 PC 来计算 br_target，显然是错误的。

（3）错误原因

跳转指令的跳转地址计算使用的 fs_pc 不正确，应该使用当前指令延迟槽中的指令 fs_pc, 但是当延迟槽指令还未收到 addr_ok 时该地址尚未进入 FS 级，因此导致了错误。

（4）修正效果

一种做法是设置一个信号 true_fs_pc, 如果当前 fs_valid 有效，那么就应该取 fs 级的指令 PC，否则就应该取现在正在等待 addr_ok 的指令 PC。然后传递到下一级：

```
assign true_fs_pc      = (fs_valid) ? fs_pc : true_npc
```

采用这种做法会出现组合环， 因为 true_npc 来源于 nextpc，nextpc 来源于 br_target, br_target 来源于 true_fs_pc，因此会出现一个在 fs 和 ds 级之间的组合环。这个组合环在仿真时并没有产生任何的错误，但是在综合时却产生了时序违约，导致了综合失败。

第二个解决方法较为直接，直接在 ds 级将当前的 ds_pc 加 4 作为了延迟槽指令的 PC 值进行计算：

```
assign bd_pc = ds_pc + 32'd4;

assign br_target = (inst_beq || inst_bne || inst_bgez || inst_bgtz || inst_blez || inst_bltz || inst_bltzal || inst_bgezal) ?
                                            (bd_pc + {{14{imm[15]}}, imm[15:0], 2'b0}) :
                   (inst_jr   || inst_jalr)              ? rs_value :
                  /*inst_jal*/              {bd_pc[31:28], jidx[25:0], 2'b0};
```

这个方案能解决这个问题，仿真和上板过程中未出现任何错误。

3、错误 3：PC 数值出错

（1）错误现象

myCPU 运行的指令 PC 值与 ref_PC 的值不同，如下图所示：

```
[8089077 ns] Error!!!
    reference: PC = 0xbfc00380, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00004000
    mycpu    : PC = 0xbfc1e878, wb_rf_wnum = 0x12, wb_rf_wdata = 0x00000001
```

（2）分析定位过程

根据提供的 reference 以及测试情况来看，0xbfc00380 是例外的入口地址，因此可能是在发生了例外的情况下导致了 PC 改变，但是指令流仍然保持原先的执行顺寻导致的。

查看 test.s:

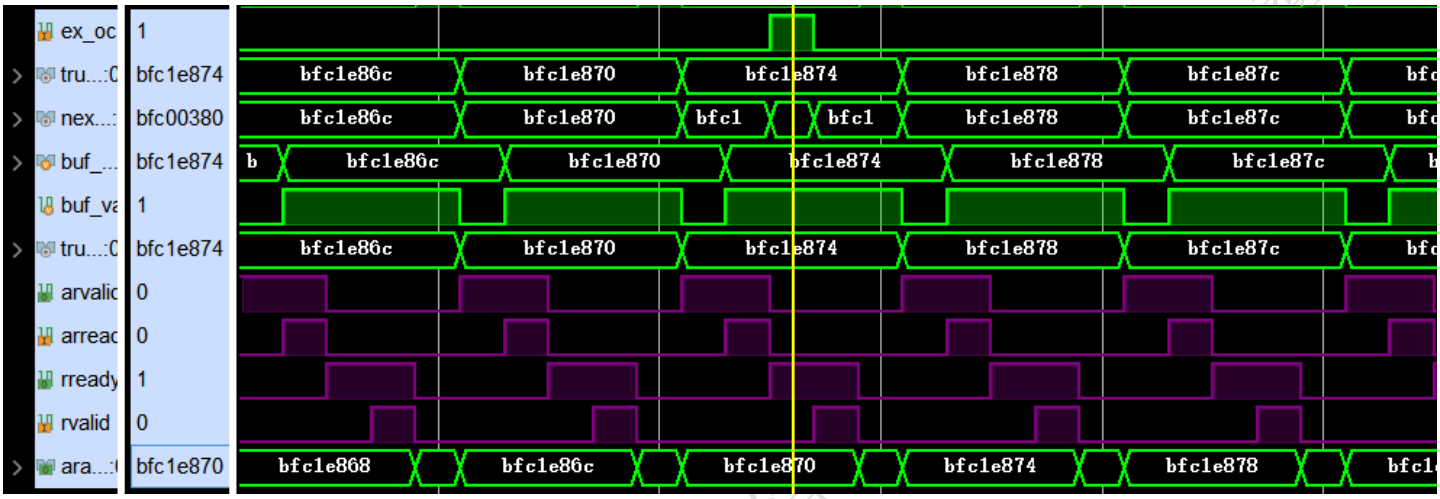
```
bfc1e86c <syscall_pc1>:
/media/sf_xjz/class/ucas/ucas19_20/jiafan_20191014/lab8-9/func_lab9/inst/n69_syscall_ex.S:21
bfc1e86c:   0000000c    syscall
/media/sf_xjz/class/ucas/ucas19_20/jiafan_20191014/lab8-9/func_lab9/inst/n69_syscall_ex.S:22
bfc1e870:   1657003c    bne s2,s7,bfc1e964 <inst_error>
/media/sf_xjz/class/ucas/ucas19_20/jiafan_20191014/lab8-9/func_lab9/inst/n69_syscall_ex.S:23
bfc1e874:   00000000    nop
/media/sf_xjz/class/ucas/ucas19_20/jiafan_20191014/lab8-9/func_lab9/inst/n69_syscall_ex.S:25
bfc1e878:   24120001    li  s2,1
```

可以看出 0xbfc1e86c 是一条 syscall 指令，会触发例外。但是 myCPU 的指令一直执行到了 0xbfc1e878。

从波形可以看出，在这个时钟周期，nextpc 保持了一小段时间的 0xbfc00380，但是很快就消失不见了，所以接下来的访问指令 sram 的地址都没有 0xbfc00380，导致没有拿到例外处理的第一条指令。

（3）错误原因

发生例外时的例外入口地址 0xbfc00380 没有被保存下来，导致后续执行并没有取到这个地址的指令，从而导



致了错误。

（4）修正效果

修正方法与跳转指令的实现方式相同，用一个寄存器 ex_addr_buf 在有例外发生时来保存例外入口地址，并且配置一个用于标记该 buf 是否有效的 valid 位，大致处理流程与 branch target 的处理类似：

```
always@(posedge clk) begin
    if(reset) begin
        ex_addr_buf_valid <= 1'b0;
    end
    else if(ex_occur || eret) begin
        ex_addr_buf_valid <= 1'b1;
    end
    else if(ex_addr_buf_valid && !buf_valid) begin
        ex_addr_buf_valid <= 1'b0;
    end

    if(ex_occur) begin
        ex_addr_buf <= 32'hbfc00380;
    end
    else if(eret) begin
        ex_addr_buf <= epc_reg;
    end
end
```

```
if(!buf_valid) begin
    if(ex_addr_buf_valid) begin
        buf_npc <= ex_addr_buf;
    end
    else if(branch_buf_valid) begin
        buf_npc <= branch_buf;
    end
    else begin
        buf_npc <= nextpc;
    end
end
```

ex_addr_buf 中保存的不仅仅是例外的入口地址，如果当前指令是 eret 指令的话，也是需要保存下来的。这里需要注意是否会发生 ex_occur 和 eret 指令同时发生的情况，通常来说不会。因为 eret 和 ex_occur 都是在写回级才报出的。

但是这里在给 buf_npc 赋值时就需要注意优先级顺序了，如果现在 ex_addr_buf_valid 有效，那么就把 ex_addr_buf 中的值赋值给 buf_npc, 否则才是 branch_buf 中的分支地址。

（5）归纳总结（可选）

和之前在处理跳转地址时犯的错误一样，都是在写代码之前没有考虑清楚。

4、错误 4：PC 数值出错

给出的跳转 PC 的值与 referrence 提供的跳转 PC 值不同，二者相差 4：

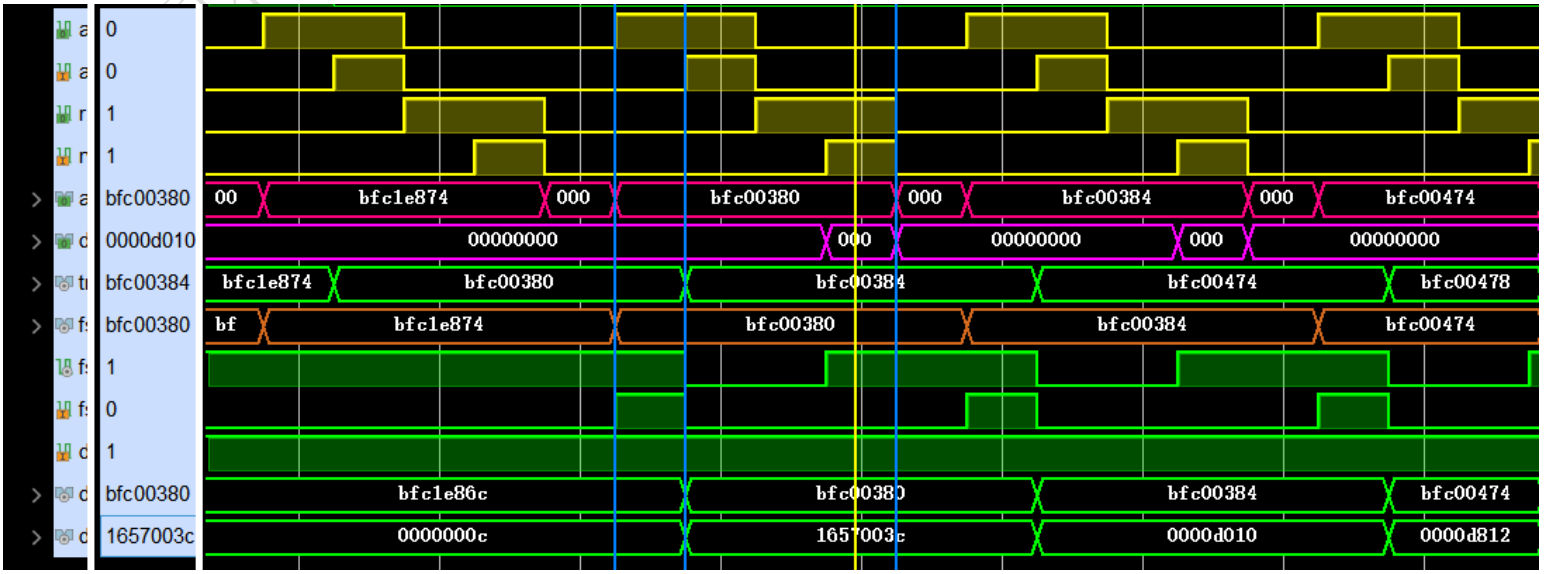
```
[8089097 ns] Error!!!
    reference: PC = 0xbfc00380, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00004000
    mycpu    : PC = 0xbfc00384, wb_rf_wnum = 0x1a, wb_rf_wdata = 0x00004000
```

（2）分析定位过程

```
/media/sf_xjz/class/ucas/ucas19_20/jiafan_20191014/lab8-9/func_lab9/start.S:63
bfc00380:   0000d010    mfhi    k0
/media/sf_xjz/class/ucas/ucas19_20/jiafan_20191014/lab8-9/func_lab9/start.S:64
bfc00384:   0000d812    mflo    k1
```

0xbfc00380 位置应该是 0x0000d010 指令，但是从波形图上来看却不是这样：

可以看到在 ds_pc 为 0xbfc00380 的时候，指令却是 0x1657003c, 而当 ds_pc 为 0xbfc00384 的时候，指令才成为了 0x0000d010.



可以看出在读出了 inst 的数据为 0x0000d010 之后，fs_to_ds_valid 一直没变，导致了读出的数据还没有在 0xbfc00380 有效的时候写入到下一级里面。反而是把上一条指令，即 0xbfc1e86c 地址的指令写到了 ds 级。但是这样做实际上是不对的，因为在发生了例外之后，会刷新流水线，让 fs_valid 变为无效。在以前的处理中，刷新一个时钟周期就行了，但是现在读回指令有延迟，因此需要将刷新信号持续很久，直到例外处理入口的指令返回了之后再降下刷新信号。

（3）错误原因

在发生了例外的时候，未能正确地刷新流水线，导致一些非法的指令也进入到了 ds 级。

（4）修正效果

配上 cancel 信号，该信号当在发生例外的时候拉起，在例外入口的指令被读回了之后降下来，在 cancel 有效期间，所有的取得的指令都将被标记为无效，从而即使向后面送也不会产生实际的运行效果。

```
always@(posedge clk) begin
    if(reset) begin
        cancle <= 1'b0;
    end
    else if(~ex_addr_buf_valid && inst_sram_data_ok) begin
        cancle <= 1'b0;
    end
    else if(ex_occur || eret) begin
        cancle <= 1'b1;
    end
end
```

第一步是设置 cancle 信号，当例外地址被接收(ex_addr_buf_valid 为 0)并且收回到的指令正好是例外入口的那条指令时(inst_sram_data_ok)时，才会降为 0。

```
always@(posedge clk) begin
    if(reset) begin
        rinst_buf_valid <= 1'b0;
    end
    else if(fs_to_ds_valid && ds_allowin) begin
        rinst_buf_valid <= 1'b0;
    end

    else if(!rinst_buf_valid && inst_sram_data_ok && !cancle) begin
        rinst_buf_valid <= 1'b1;
    end

    if(!rinst_buf_valid && inst_sram_data_ok && !cancle) begin
        rinst_buf <= inst_sram_rdata;
    end
end
```

在 cancle 信号有效期间，说明收回到的指令应该被取消，所以只有在 cancle 无效的时候才会给 rinst_buf_valid 赋值为 1。才会给 rinst_buf 中赋值。

在做了这样的处理后，能够顺利地进行跳转了。

（5）归纳总结（可选）

这个错误仍然是因为对整个运行流程理解不清晰导致的。在以前写例外处理时需要刷新 fs_valid，但是那时即使刷新也只持续了一个时钟周期，现在延迟增加的太大了，只持续一拍的 fs 无效导致了电路行为错误。

5、错误 5：inst_req 信号出错

仿真开始后，inst_addr_ok 一直为 x，不拉高，导致 cpu 暂停工作，无法取值。

（2）分析定位过程

我们直接查找 inst_req 的赋值可以发现，该信号含有 inst_addr_ok，而在上个星期写转换桥时我使用了 inst_req 来赋值 inst_addr_ok，这样形成了一个组合环导致信号错误。

（3）错误原因

在给信号赋值时产生了组合环。

（4）修正效果

我在 if 级流水线设置了一个寄存器 inst_req_r 来赋值给 inst_req 信号，而 inst_req_r 由其他握手信号驱动，这样就避免了组合环，使 CPU 能够正常工作。