实验 14 报告

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

一、实验任务（10%）

在实验 13 的基础上，增加对 TLB 例外的处理，共有 TLB Refill， TLB Invalid， TLB Modified 三种 TLB 例外。

同时取指和访存阶段的虚拟地址需要经过 TLB 获得实际的物理地址，经由 AXI 总线发出的地址应该是物理地址。最终能够完成仿真测试和上板测试。

二、实验设计（40%）

（一）总体设计思路

1. 总体设计思路

在整个 TLB 例外的处理上，和其他例外没有什么太多不一样的区别。都是在每个流水级进行例外的判断和存储。如果发生了例外，就在该级流水级带上例外信号，随着流水线流向接下来的流水级。并且最终在 WB 级根据接收到的例外信号刷新流水线，重新设置 PC 的值，以及更新 CP0 寄存器等等。本次实验一个需要注意的点是 TLB Refill 例外产生时，需要将 PC 的值设置为 0xbfc00200, 这和一般的例外处理时跳转的 PC 不一样。

模块改动：

1. IF 级：

在取指级可能发生的 TLB 例外是 TLB Refill， TLB Invalid 这两种例外。前者是根据 s0_found 的值来设置，后者是根据 s0_found 以及 s0_valid 的值来设置。但需要注意的一点是，产生这两种例外的前提是本身的取指地址是在 mapped 段，才会导致产生这两种例外。

用 mapped 信号来表示该 PC 地址是否处于 mapped 段：

```
assign mapped       = !(true_npc[31:29] == 3'b100 || true_npc[31:29] == 3'b101);
```

在 mapped 的基础上判断 TLB 例外的产生：

```
assign tlb_refill  = mapped & ~s0_found;
assign tlb_invalid = mapped & s0_found & ~s0_v;
```

一旦判断 TLB 例外出现了，就不能向 AXI 总线发起读数据的请求，否则读回来的数据是不确定的。甚至会引


```
assign inst_sram_req = fs_allowin & ~handshake & (~reset) & (~tlb_refill) & (~tlb_invalid);
```

起 AXI 总线出现异常。所以还需要重置 inst_sram_req 信号：

true_npc 最终会到达 fs_pc, 但是由于不需要等待指令返回，因此可以马上进入 ID 级。这里我们使用一个 reg 型变量 tlb_ex, 它表示的是当前处于 fs_pc 中的虚拟地址是否有 tlb 例外。其赋值如下所示：

表示的意思是说如果处于 pre-IF 的这条指令有 TLB 例外，并且 fs_allowin 时，就将它置为 1，从而 tlb_ex 置为

```
always@(posedge clk) begin
    if(reset) begin
        tlb_ex <= 1'b0;
    end
assign fs_ready_go    = (inst_sram_data_ok | rinst_buf_valid | tlb_ex) ;
assign fs_allowin     = !fs_valid || fs_ready_go && ds_allowin;
assign fs_to_ds_valid =  fs_valid && fs_ready_go && ~cancle ;
        tlb_ex <= 1'b0;
    end
end
```

1 的时机实际上是与 true_npc 进入 fs_pc 的时机是同步的。但是对于其置为 0 的时机，我们选择了在刷新流水线时进行。这是因为一旦一条指令发生了 TLB 无效，那么它接下来的大段指令也会发生 TLB 无效，而最开始导致 TLB 无效的那条指令将会在 WB 级刷新流水线，使 tlb_ex 变为 0. 关于 tlb_ex 的用途，体现在下面：

可以看出，如果当前的 fs_pc 中的这条指令是 tlb_ex 的，那么 fs_ready_go 直接置为 1，可以向下传递。这也和前面所说的相符合：发生 TLB 例外的指令，不向 AXI 总线发请求，无需等待指令返回，直接向接下来的流水级传递。

接下来的一个问题是，TLB 例外如何传递，当 true_npc 到达 fs_pc 之后，TLB 的 s0 查找端口返回的不再是 fs_pc 的查找结果了，而是现在的 true_npc 的结果。对此，解决方法是增加新的寄存器，用于保存 tlb 查找的结果。注意，它们的值也需要与 true_pc 进入 fs_pc 时同步更新：

```
assign fs_mapped   =  !(fs_pc[31:29] == 3'b100 || fs_pc[31:29] == 3'b101);
always@(posedge clk) begin
    if(reset) begin
        fs_s0_found_reg <= 1'b0;
        fs_s0_v_reg     <= 1'b0;
    end
    else if(fs_allowin && to_fs_valid) begin
        fs_s0_found_reg <= s0_found;
        fs_s0_v_reg     <= s0_v;
    end
end
```

此时，判断 TLB 例外应该根据 fs_pc 以及保存的这些 s0 寄存器的值来判断了：

```
assign fs_tlb_refill    = fs_mapped & ~fs_s0_found_reg;
assign fs_tlb_invalid   = fs_mapped &  fs_s0_found_reg & ~fs_s0_v_reg;
```

这些信号是真正应当向下一流水级传递的信号。

而关于向下一流水级传递的 Excode 信号，也需要根据 TLB 例外进行修改：

```
assign if_excode       = (if_ex & invalid_addr_ex) ? 5'h04 :
                         (if_ex & fs_tlb_refill)      ? 5'h02 :
                         (if_ex & fs_tlb_invalid)     ? 5'h02 : 5'h0;
```

2. ID 级：

   ID 级没有什么值得修改的地方。

3. EXE 级：

   EXE 级本质上与 IF 级的处理类似。

   我们用一个 mapped 信号来表示访存地址是否在 mapped 段，如果在，才会根据 s1 的值进行 TLB 例外的判断：

```
assign mapped      = !(true_addr[31:29] == 3'b100 || true_addr[31:29] == 3'b101);
assign true_addr = buf_addr_valid ? buf_addr : es_alu_result;
assign s1_tlb_refill    = mapped & es_valid & (is_store | is_lw) & ~s1_found;
assign s1_tlb_invalid   = mapped & es_valid & (is_store | is_lw) & s1_found & ~s1_v;
assign s1_tlb_modify    = mapped & es_valid & is_store & s1_v & ~s1_d;
```

   同时，EXE 级对 AXI 总线发起访存请求也需要看是否在访存时发生了 TLB 例外：

```
assign data_sram_req  = (is_store | es_load_op) & ms_allowin & ~handshake & es_valid
                      & no_ms_es_ex & no_ms_ws_eret & ~es_ex & ~es_refetch & no_ms_ws_refetch;
```

   虽然这里是根据 es_ex 的值来产生的，但实际上 es_ex 的值也来源于是否发生了 tlb 例外。

如同前面在 IF 级处理 TLB 例外时相同，在 EXE 级如果发生了 TLB 例外，由于不能向 AXI 总线发起访存请求，所以这里 es_ready_go 是可以直接置为 1 了，这一点在 es_ready_go 的产生逻辑中就有所体现：

```
assign es_ready_go = (reflush)? 1'b1 :
                     ((es_load_op | is_store) & ~data_sram_addr_ok & ~es_ex & ~es_refetch) ? 1'b0 :
                     (op_div  & ~m_axis_dout_tvalid_signed & ~es_refetch) ? 1'b0 :
                     (op_divu & ~m_axis_dout_tvalid_unsigned & ~es_refetch) ? 1'b0 :
                     (inst_tlbp && ms_is_valid && is_write_entryhi && !es_refetch) ? 1'b0 : 1'b1;
```

当~es_ex 和~data_sram_addr_ok 时才会发生 EXE 级的阻塞。

一个值得注意的问题是访存经过的 TLB 查询端口和 TLBP 指令需要的 TLB 查询端口都是 s1 端口，因此是否会出现将一个 tlbp 指令返回的 s1_found 误认为是 TLB 例外的情况。仔细思考之后我发现只需要在设置 EXE 级的 excode 和 s1 的查询输入即可：

```
assign s1_vpn2   = (inst_tlbp) ? cp0_entryhi_reg[31:13] : true_addr[31:13];
assign s1_odd_page = (inst_tlbp) ? cp0_entryhi_reg[12] : true_addr[12];
assign s1_asid     = cp0_entryhi_reg[7:0];
```

```
assign exe_stage_excode      = ({5{ex_overflow}}                & 5'h0c)
                             | ({5{ex_invalid_load_addr}}       & 5'h04)
                             | ({5{ex_invalid_store_addr}}      & 5'h05)
                             | ({5{s1_tlb_refill & is_lw}}      & 5'h02)
                             | ({5{s1_tlb_refill & is_store}}   & 5'h03)
                             | ({5{s1_tlb_invalid & is_lw}}     & 5'h02)
                             | ({5{s1_tlb_invalid & is_store}}  & 5'h03)
                             | ({5{s1_tlb_modify}}              & 5'h01);
```

4. MEM 级：

   MEM 级没有什么值得一提的，主要是 ms_ready_go 的设置：

```
assign  ms_ready_go = (is_lw & ~data_sram_data_ok & ~ms_ex & ~ms_refetch) ? 1'b0 : 1'b1;
```

   从这个赋值逻辑就可以看出如果 EXE 级有 TLB 例外，那么传递到 MEM 级的 TLB 例外就会导致 ms_ready_go 变为 1，从而继续向下一个流水级传递下去。

5. WB 级：

   WB 级检测到 TLB 例外发生后，会像一般的例外一样，传递到 IF 级并且发出刷新流水线信号。值得注意的是由于 TLB Refill 例外的例外处理入口是 0xbfc00200，所以我们还需要将 TLB Refill 例外作为一个专门的例外信号传递到 IF 级：

```
if(ex_occur && !refill_ex) begin
    ex_addr_buf <= 32'hbfc00380;
end
else if(ex_occur && refill_ex) begin
    ex_addr_buf <= 32'hbfc00200;
end
```

三、实验过程（50%）

（一）实验流水账

2019/12/8　　13:00 ~ 18:00　　写完代码并完成仿真测试，成功上板

（二）错误记录

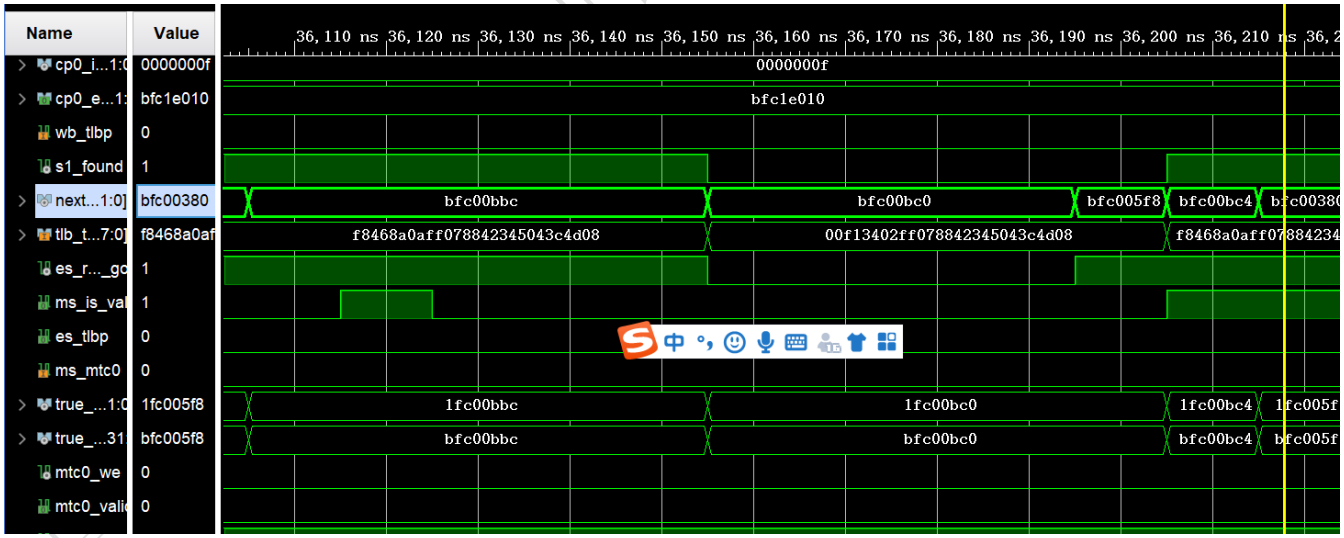
1、错误 1：例外跳转地址错误。

（1）错误现象

```
Test begin!
----[   6435 ns] Number 8'd01 Functional Test Point PASS!!!
----[   9475 ns] Number 8'd02 Functional Test Point PASS!!!
----[  12565 ns] Number 8'd03 Functional Test Point PASS!!!
----[  15655 ns] Number 8'd04 Functional Test Point PASS!!!
      [  22000 ns] Test is running, debug_wb_pc = 0xbfc00a08
      [  32000 ns] Test is running, debug_wb_pc = 0xbfc00a68
      [  42000 ns] Test is running, debug_wb_pc = 0xbfc005f8
      [  52000 ns] Test is running, debug_wb_pc = 0xbfc003b4
      [  62000 ns] Test is running, debug_wb_pc = 0xbfc00398
```

在测试例外时发生错误，自从进入例外后一直在 0xbfc00380 附近循环。

（2）分析定位过程



查看反汇编指令，是一条 sw 指令触发的 tlb invalid 例外，但他跳转到例外地址 0xbfc00380,查看讲义发现应该跳转到 0xbfc00200，在查看代码后发现之前在处理例外时在 wb 通过之前传入的 excode 来判断例外类型，判断后传入 if 级跳转，查看讲义发现 tlb invalid 和 tlb refill 的 excode 相同但是例外的跳转的地址却不同，光用 excode 判断不够妥当，我们还是按最原始的做法将例外信号顺着流水线重新传入 wb 级再进行例外处理，并区分这两种例外使其跳转到不同的地址执行例外处理。

（3）错误原因

我们仅仅从上一级的 excode 无法判断出准确的例外类型，还是需要将例外信号顺着流水线传入 wb 级作重新处理，这样才能避免跳转地址的错误。

```
assign ws_ex          = (ws_valid) ? ms_ex : 1'b0      ;
assign ws_excode      = (ws_valid) ? ms_excode : 5'b0 ;
assign ws_reflush     = (ws_valid == 1'b0) ? 1'b0 : ms_ex;
```

（4）修正效果

在加上例外判断信号后通过该测试点。

（5）归纳总结（可选）

该错误是对例外处理的 excode 不是特别熟悉，以后应该更加注意细节。

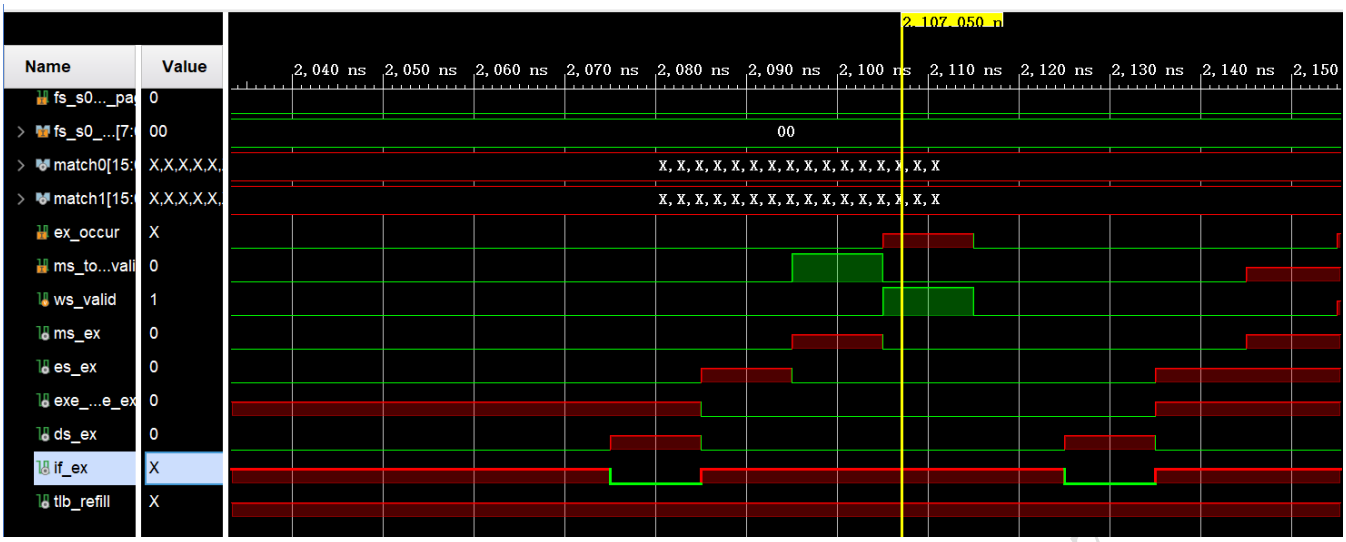
2、错误 2：tlb 例外信号

（1）错误现象

```
Test begin!
        [  22000 ns] Test is running, debug_wb_pc = 0xbfc00000
        [  32000 ns] Test is running, debug_wb_pc = 0xbfc00000
        [  42000 ns] Test is running, debug_wb_pc = 0xbfc00000
        [  52000 ns] Test is running, debug_wb_pc = 0xbfc00000
        [  62000 ns] Test is running, debug_wb_pc = 0xbfc00000
```

仿真发现 pc 一直停在 0xbfc00000 不动。

（2）分析定位过程



查看波形，我们可以清楚地发现，if_ex 信号在大部分情况为 X，观察赋值后发现是 s0_found 为 X 的原因，究其原因，我们的 tlb 项没有初始化，所以查找的时候也为 X。

```
assign tlb_refill = (s0_found == 1'b0) ;
```

为了避免这种情况的发生，我引入了 1 个 16 位信号 tlb_valid 来判断 tlb, reset 时，将 tlb_valid 信号置 0,如果 tlb 被 tlbwi 指令写入，则将 tlb_valid 信号对应位置 1，这样如果未被写入，我们直接将例外信号置 0 防止例外的发生。之后通过 tlb_valid 信号判断例外信号。

```
assign tlb_refill = (tlb_valid!=16'b0)?  (s0_found==1'b0)  : 1'b0 ;
```

tlb_valid 信号的赋值如下：

```
always@(posedge clk)
begin
    if(reset) begin
        tlb_valid_reg <= 16'b0;
    end
    else if(we) begin
        tlb_valid_reg[w_index] <= 1'b1;
    end
end
```

（3）错误原因

Tlb_refill 信号的赋值没有考虑周全，没有考虑信号为 X 的情况。

（4）修正效果

在之前赋值的基础上多增加 tlb_valid 的信号，流水线能够成功运行。

四、实验总结（可选）