实验 13 报告

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

一、实验任务（10%）

向 CPU 中添加 TLBR， TLBWI，TLBP 指令。在 CP0 寄存器模块中增加 Index, Entryhi, Entrylo0, Entrylo1 寄存器作为 TLB 相关例外处理的协处理器寄存器。在整个 CPU 的外部与实验 12 中实现的 TLB 模块相连接，并设置页面的大小为 4KB，要求整个 CPU 运行过程中取指阶段和访存阶段的地址都必须经过 TLB 以便得到真实的物理地址。最终运行 flb_func 中的测试程序，要求能顺利通过前 6 项测试和顺利上板。

二、实验设计（40%）

（一）总体设计思路

1. 总体设计

在总体设计上，我们需要向 CPU 中增加 TLB 模块。这里的处理方法是在 mycpu_top.v 这个模块中增加与各流水级“平行”的 TLB 模块，其考虑是取值级和访存级都需要经过 TLB，因此不能简单地将 TLB 寄存器在某个流水级中。

在 IF 级，以取值的 PC 值的 VPN2，asid 以及 odd_page 信号作为输出，接到 TLB 的 S0 访问端口。并且将查得 TLB 的 found, valid, pfn,dirty 等信号作为输入信号返回回来。asid 的值需要从 WB 级的 CP0 寄存器的对应域传递过来。

```
// tlb read port0:
output [18:0]       s0_vpn2             ,
output              s0_odd_page         ,
output [7: 0]       s0_asid             ,


// tlb read port 0 input:
input               s0_found            ,
input  [ 3:0]       s0_index            ,
input  [19:0]       s0_pfn              ,
input  [31:0]       cp0_entryhi_reg     ,
```

在这几个信号的产生上，我们只需要使用原先的 AXI 接口的 inst_sram_addr 进行一系列的组合逻辑就可以

```
assign true_phy_pc = (true_npc[31:29] == 3'b100 || true_npc[31:29] == 3'b101) ? {3'b0, true_npc[28:0]}:
                     (s0_found) ? {s0_pfn, true_npc[11:0]} : true_npc;
```

得到了。这里需要注意地址是否在 mapped 段，如果在 unmapped 段，那么是不需要经过 TLB 的。

由于实验 13 中还不涉及 TLB 例外的处理，因此这里暂时省略了相关的例外生成逻辑。

同理在 EXE 级，需要以访存地址的虚拟地址来生成 s1_vpn2, s1_odd_page, s1_asid 等信号。其中 asid 的值也需要来源于 CP0 寄存器的 entryhi 寄存器的 asid 域。

```
assign true_addr = buf_addr_valid ? buf_addr : es_alu_result;
assign true_phy_addr = (true_addr[31:29] == 3'b100 || true_addr[31:29] == 3'b101) ? {3'b0, true_addr[28:0]}:
                       (s1_found) ? {s1_pfn, true_addr[11:0]} : true_addr;
```

2. TLB 相关的协处理器寄存器的实现

TLB 相关的协处理器寄存器有 CP0_ENTRYHI 寄存器，CP0_ENTRYLO0, CP0_ENTRYLO1, CP0_INDEX 寄存器。在设计上，采用的仍然是与之前 CP0 寄存器相同的写法，将其切分为各个域，而不是用作一个 32 位的整体寄存器。

(a). cp0_entryhi 寄存器：

cp0_entryhi_vpn2: 表示虚拟页号。其更新只有两种情况：mtc0 写以及 tlbr 读 TLB 表项是写入：

cp0_entryhi_asid : 用于区分不同进程的页表，更新情况与 vpn2 相同。

```
reg [18:0]  cp0_entryhi_vpn2;
reg [ 7:0]  cp0_entryhi_asid;

always@(posedge clk) begin
    if(reset) begin
        cp0_entryhi_vpn2 <= 19'b0;
        cp0_entryhi_asid <= 8'b0;
    end
    else if(mtc0_we && cp0_waddr == 8'h50) begin
        cp0_entryhi_vpn2 <= cp0_wdata[31:13];
        cp0_entryhi_asid <= cp0_wdata[ 7: 0];
    end
    else if(inst_tlbr && ws_valid)  begin
        cp0_entryhi_vpn2 <= r_vpn2;
        cp0_entryhi_asid <= r_asid;
    end
end
```

在 IF 级以及 EXE 级地址需要经过 TLB，其 asid 信号需要来自于该 cp0_entryhi_asid， 因此需要将其 作为输出信号传送到 IF 和 EXE。

(b). cp0_entrylo0 寄存器，其包括的域有：cp0_entrylo0_pfn, cp0_entrylo0_v, cp0_entrylo0_d, cp0_entrylo0_c, cp0_entrylo0_g。 这些域的更新逻辑也是在 mtc0 写有效或者 tlbr 的时候才会写入。

```
    reg [19:0]  cp0_entrylo0_pfn0   ;
    reg [ 2:0]  cp0_entrylo0_c0     ;
    reg         cp0_entrylo0_d0     ;
    reg         cp0_entrylo0_v0     ;
    reg         cp0_entrylo0_g0     ;

always@(posedge clk) begin
    if(reset) begin
        cp0_entrylo0_pfn0 <= 20'b0  ;
        cp0_entrylo0_c0   <= 3'b0   ;
        cp0_entrylo0_d0   <= 1'b0   ;
        cp0_entrylo0_v0   <= 1'b0   ;
        cp0_entrylo0_g0   <= 1'b0   ;
    end
    else if(mtc0_we && cp0_waddr == 8'h10) begin
        cp0_entrylo0_pfn0 <= cp0_wdata[25:6] ;
        cp0_entrylo0_c0   <= cp0_wdata[ 5:3] ;
        cp0_entrylo0_d0   <= cp0_wdata[2]    ;
        cp0_entrylo0_v0   <= cp0_wdata[1]    ;
        cp0_entrylo0_g0   <= cp0_wdata[0]    ;
    end
    else if(inst_tlbr && ws_valid) begin
        cp0_entrylo0_pfn0 <= r_pfn0 ;
        cp0_entrylo0_c0   <= r_c0   ;
        cp0_entrylo0_d0   <= r_d0   ;
        cp0_entrylo0_v0   <= r_v0   ;
        cp0_entrylo0_g0   <= r_g    ;
    end
```

(c). cp0_entrylo1 寄存器：

cp0_entrylo1 寄存器的各个域与 entrylo0 的各个域均相同，此处不再赘述。

(d). cp0_index 寄存器：

cp0_index 寄存器的有效域是 index 域和最高位 p 域。

cp0_index_Index 的更新条件是：mtc0 写该寄存器或者 tlbp 指令查到的 index 写入该域；

cp0_index_p 的更新条件是：tlbp 查找到了就写 0，否则写 1.

```
always@(posedge clk) begin
    if(reset) begin
        cp0_index_p <= 1'b0;
    end
    else if(mtc0_we  && cp0_waddr == 8'h00) begin
        cp0_index_Index <= cp0_wdata[3:0];
        //cp0_index_p     <= cp0_wdata[31] ;
    end
    else if(inst_tlbp && tlbp_found && ws_valid) begin
        cp0_index_Index <= cp0_windex;
        cp0_index_p     <= 1'b0;
    end
    else if(inst_tlbp && !tlbp_found && ws_valid) begin
        cp0_index_p     <= 1'b1;
    end
end
```

（注：关于 cp0_index_p 的赋值，这里有两种不同的意见。第一种是在 reset 的时候不要复位，使得 p 的值是 X，然后仅在 tlbp 指令时写入 p 位，但是这样的处理貌似会在后面的一处测试时发生错误。另外一种处理方法根据下图实现：

Name	Bit(s)	Description	Read/ Write	Reset State
P	31	Probe Failure. This bit is automatically set when a **TLBP** search of the TLB fails to find a matching entry. The following rules apply when accessing this bit: 1. Root can only set Root.Index.P value to 1 (and not clear it) using the MTC0 instruction. 2, Guest can only set Guest.Index.P value to 1 (and not clear it) using the MTC0 instruction. 3. Root can both set and clear Guest.Index.P value using the MTGC0 instruc-tion.	WO or R/W (See descr)	0

(参考资料：MIPS64® P6600 Multiprocessing System Software User's Guide)

图中明确表示 P 位在 reset 时的值为 0，并且可由 mtc0 指令改变。基于此，我们的设计中使用了上面的赋值逻辑。)

3. TLB 指令的实现：

在 ID 级增加三种指令的 inst 信号，并且进行译码和设置相应的控制信号。

a) tlbp：

复用 s1 查找端口，因为同一条指令不可能既访存又通过 s1 端口查 TLB，因此这样是可行的。查找时需要输出 vpn2, asid, odd_page 等域，这些域来自于 cp0_entryhi 寄存器。将返回的 s1_found 和 s1_index 记录下来一路带回到 WB 级写回 cp0 寄存器。

```
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 tlbp_found  = s1_found   ;
assign cp0_windex  = s1_index   ;
```

b) tlbwi:

在 WB 级，将 entryhi, entrylo0, entrylo1,index 等寄存器的值输出该模块，连接到 TLB 模块，并且如果当前指令是 tlbwi 时，就使写使能信号 we 为高，进行写入。

```
assign tlb_we    = inst_tlbwi & ws_valid & ~refetch;
assign w_index   = cp0_index_reg[3:0];
assign r_index   = cp0_index_reg[3:0];
assign w_vpn2    = cp0_entryhi_reg[31:13];
assign w_asid    = cp0_entryhi_reg[ 7: 0];
assign w_g       = cp0_entrylo0_reg[0] & cp0_entrylo1_reg[0];
assign w_pfn0    = cp0_entrylo0_reg[25:6];
assign w_c0      = cp0_entrylo0_reg[5:3];
assign w_d0      = cp0_entrylo0_reg[2];
assign w_v0      = cp0_entrylo0_reg[1];
assign w_pfn1    = cp0_entrylo1_reg[25:6];
assign w_c1      = cp0_entrylo1_reg[5:3];
assign w_d1      = cp0_entrylo1_reg[2];
assign w_v1      = cp0_entrylo1_reg[1];
```

c). tlbr:

tlbr 指令将 index 寄存器的值作为输出，将 TLB 读到的 entryhi 和 entrylo0，entrylo1 的各个域返回回来，并且写入到对应的 CP0 寄存器中。对应的逻辑已经在上面的 CP0 寄存器的各个域的赋值逻辑中体现了，此处不再赘述。

4. 围绕 tlbwi 和 tlbr 产生的问题：

tlbwi 和 tlbr 都会更新 entryhi 寄存器的 asid 域，但是通过 TLB 查找时会看 asid 域，这就导致在 asid 还没写进时进行的取指和访存的 TLB 查找是不对的。需要进行重取操作。对此，实现是：

在 IF 级增加 refetch 寄存器，如果检测到当前 ID 级是 tlbwi 或 tlbr 指令时，就需要将 refetch 置为 1，然后随下一条指令传递到后面的流水级。refetch 寄存器重新置为 0 的条件是收到了下一条回来的指令。因此我们就给下一条指令标上了 refetch，但是并没有给后面所有信号标上 refetch。因为我们给 tlbwi 或 tlbr 的下一条指令标上了 refetch 信号，因此当它到达 wb 级时一定会触发刷新，进行重取，后面的指令也会继续更着重取。这里我们将重取看作是例外的一种，只是不需要设置每一级的 ex 域，从而避免了写 CP0 寄存器。

当 refetch 信号到达 WB 级之后，和普通的报例外一样，向 fs 级发出信号，并且刷新各级流水线。

```
assign refetch         = ws_refetch;
assign refetch_pc      = (ws_valid) ? ws_pc : 32'b0;
assign reflush         = (ws_valid) & (ws_reflush | eret | refetch);
```

另一个需要注意的地方在于，当每一级检测到后面有 refetch 的指令时，就不要让当前的该流水级的指令做出任何可能改变内存，寄存器，或是运算器的动作。这体现在：

1) load, store 指令不要发出 AXI 请求：

```
assign no_ms_es_ex        = ~ms_exc & ~ws_exc;
assign no_ms_ws_eret      = ~ms_eret & ~ws_eret;
assign no_ms_ws_refetch   = ~ms_refetch & ~ws_refetch;
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;
```

2) 当前处于 WS 级的 refetch 指令不要写回寄存器堆。

```
assign rf_we    = (ws_valid & ~ws_ex & ~refetch) & (ws_gr_we | mfc0_read);
```

3) 当前指令如果是乘法或除法指令，EXE 级的除法器, 乘法器(如果有的话)的握手信号不要拉高，以防止进入相应的运算。虽然即使进入运算之后，返回的结果可能也能通过一定的逻辑丢弃掉，但是这实在过于冒险。

以有符号除法的握手信号为例：其判别条件需要加上 no_ms_ws_refetch:

```
else if(op_div && !is_dividing_signed && no_ms_es_ex && es_valid && no_ms_ws_eret && no_ms_ws_refetch) begin
    s_axis_dividend_tvalid_signed <= 1'b1;
end
```

关于报 refetch 后，fs 级到底如何处理，根本上与例外处理相同：用一个 ex_addr_buf 来存，直到等到 true_npc 来接收该地址。

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

always@(posedge clk) begin
    if(reset) begin
        refetch_reg <= 1'b0;
    end
    else if(refetch_reg && inst_sram_data_ok && !cancle) begin
        refetch_reg <= 1'b0;
    end
    else if(ds_tlbr_tlbwi) begin
        refetch_reg <= 1'b1;
    end
end
```

这里的设计有一定的问题，主要是 refetch 的设置，如果普遍认为指令的返回 inst_sram_data_ok 都是有延迟的，那么这里的处理毫无问题。因为当 tlbwi 或 tlbr 指令在 ID 级时，它的下一条 PC 一定在 fs 级等待指令返回，或者在等待地址握手信号。如果该过程有延迟，那么无论如何下一条指令的数据都不会和 tlbwi 或 tlbr 同步在下一周期移动到下一流水级，因此可以给 refetch_reg 寄存器一个周期的时间来让它被置为 1，从而当下一条指令到达的时候，refetch_reg 可以和指令一起进入 DS 级。但是如果指令的返回没有延迟，那么当 ID 级检测到 tlbwi 或 tlbr 时，fs 级的 refetch 还未置起，这也就导致了 tlbwi 和 tlbr 的下一条指令到达 ID 级时相应的 refetch 信号为 0，产生错误。

关于这一问题，需要适当的考虑当前的 inst_sram_data_ok, 使它和 inst 同步到达下一流水级。也就是说，如果 rinst_buf_valid 无效并且 inst_sram_data_ok 并且 ds_allowin 时，就说明这一条指令不会进入 rinst_buf 了，也就是说该条指令将会直接进入 ds 级，此时，直接将 fs_refetch 信号设置为来自 ID 级的指示当前 ID 级的是否是 tlbwi 或是 tlbr 的信号 ds_tlbr_tlbwi 信号即可，否则需要用 refetch_reg 信号。

正确的赋值逻辑应当如下图所示：

```
assign fs_refetch = (inst_sram_data_ok & ~rinst_buf_valid & ds_allowin) ? ds_tlbr_tlbwi : refetch_reg;
```

三、实验过程（50%）

（一）实验流水账

2019/12/4	19：00 ~ 22：00	写代码， 仿真
2019/12/5	10：00 ~ 11：30	通过仿真，上板失败
2019/12/6	19：30 ~ 20：30	找上板错的地方，正准备使用逻辑分析仪时发现有一根模块的线没连。修正后上板通过。

（二）错误记录

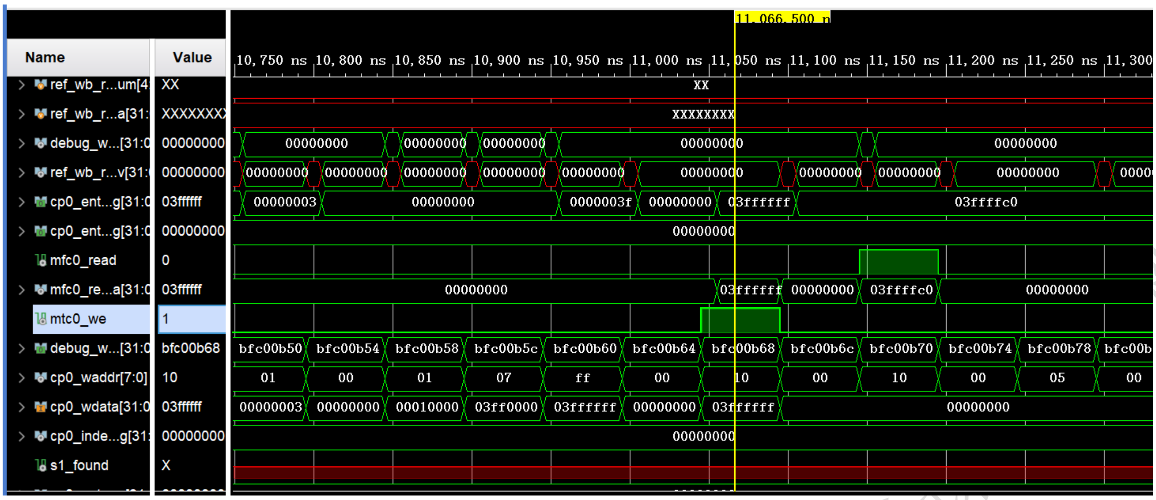
1、错误 1：cp0 寄存器赋值错误

（1）错误现象

```
============================================================
Test begin!
----[   6485 ns] Number 8'd01 Functional Test Point PASS!!!
----[   9675 ns] Number 8'd02 Functional Test Point PASS!!!
------------------------------------------------------------
[  11515 ns] Error(  0)!!! Occurred in number 8'd03 Functional Test Point!
```

在进行第三个测试点时发生错误。

（2）分析定位过程



查看反汇编指令，是一条 mfc0 读 entrylo0 寄存器的值出错。对指令解析后，发现 entrylo0 寄存器在前面的 mtc0 指令时已经得到了正确的值，但是却在下一个时钟上升沿变化了，而没有其他 cp0 指令，他的值不应该变化，之后我检查 entrylo0 寄存器的赋值逻辑发现，在将 cp0 寄存器的各个部分统一赋值后，没有加 begin，end 导致除了 pfn0，其他部分都会在时钟上升沿发生变化，导致错误。

（3）错误原因

在将 cp0 寄存器的各个部分统一赋值后，没有加 begin，end 导致除了 pfn0，其他部分都会在时钟上升沿发生变化。

```
else if(wb_tlbr) begin
    cp0_entrylo0_pfn0 <= r_pfn0;
    cp0_entrylo0_c0 <= r_c0;
    cp0_entrylo0_d0 <= r_d0;
    cp0_entrylo0_v0 <= r_v0;
    cp0_entrylo0_g0 <= r_g;
end
else if(mtc0_we && cp0_waddr == 8'b00010000)
    cp0_entrylo0_pfn0 <= cp0_wdata[25:6];
    cp0_entrylo0_c0 <= cp0_wdata[5:3];
    cp0_entrylo0_d0 <= cp0_wdata[2];
    cp0_entrylo0_v0 <= cp0_wdata[1];
    cp0_entrylo0_g0 <= cp0_wdata[0];
```

（4）修正效果

在加上 begin end 后通过该测试点。

（5）归纳总结（可选）

该错误是将一个寄存器的多个域一起赋值而考虑不周导致的，以后应该更加注意细节。

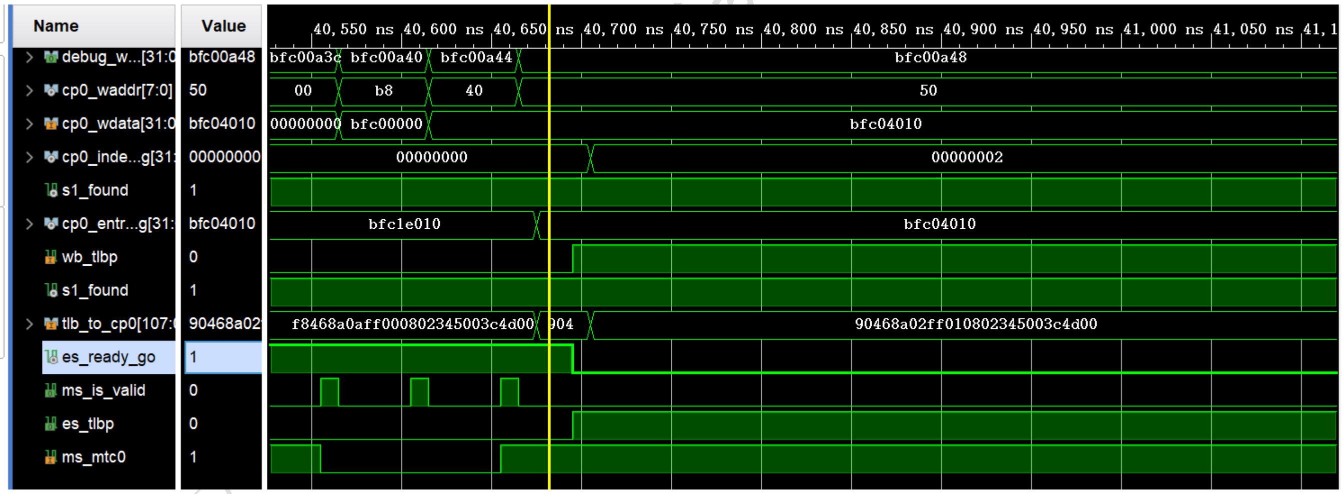
2、错误 2：ready_go 信号

（1）错误现象

```
Test begin!
----[   6485 ns] Number 8'd01 Functional Test Point PASS!!!
----[   9675 ns] Number 8'd02 Functional Test Point PASS!!!
----[  12915 ns] Number 8'd03 Functional Test Point PASS!!!
----[  16155 ns] Number 8'd04 Functional Test Point PASS!!!
        [  22000 ns] Test is running, debug_wb_pc = 0xbfc00840
        [  32000 ns] Test is running, debug_wb_pc = 0xbfc00840
----[  39295 ns] Number 8'd05 Functional Test Point PASS!!!
        [  42000 ns] Test is running, debug_wb_pc = 0xbfc00a48
        [  52000 ns] Test is running, debug_wb_pc = 0xbfc00a48
        [  62000 ns] Test is running, debug_wb_pc = 0xbfc00a48
        [  72000 ns] Test is running, debug_wb_pc = 0xbfc00a48
```

发现程序在通过第 5 个测试点之后，一直停在了某条指令。

（2）分析定位过程



查看反汇编指令，是一条 tlbp 指令，观察 es_ready_go 信号发现，在执行 tlbp 时一直为 0，导致整个流水线阻塞，我在写 tlbp 指令时根据讲义中的写法修改了 es 级的 ready_go 信号，当 es 级有 tlbp 指令而 mem 级有一条修改 EntryHi 的 MTC0 指令时，我们需要阻塞，但波形上一直阻塞。后来我想到了原因，在换用 AXI 接口后，ms 信号因为阻塞原因只存在 1 拍，而我们也只需要将流水线阻塞 1 拍即可，所以我修改了 ready_go 的逻辑使得 ms 级为 mtc0 指令，且 ms 流水级有效时阻塞。

（3）错误原因

Ready_go 信号的赋值没有考虑周全，导致流水线一直阻塞。

（4）修正效果

在之前赋值的基础上多与上 ms_is_valid，通过仿真。

实验总结：

本次实验较为简单，本质上只需要在 CPU 送往 AXI 桥的两个访存地址上通过组合逻辑形成正确的物理地址即可。也不涉及任何的例外(暂时)。唯一涉及到时序逻辑的地方就是在 refetc_reg 这个信号的处理上，以及在报出refetch之后如何改变PC的值重新取指。采用的解决方法是将refetch本身作为一种特殊的例外来处理。在refetch_reg的处理上，需要注意到指令数据返回延迟的问题， 使得 refetch_reg 信号能够同步与指令进入下一流水级。这个问题的解决也许能为以后 cache 模块的集成提供一定的思路。