实验 14 报告

学号: 2018K8009929043 2018K8009929035
姓名: 曾冕 张翔雨
箱子号： 06

一、实验任务（10%）

本实验主要在 lab13 的基础上将 TLB 模块集成到 lab12 完成的 AXI 总线的 CPU 中。主要涉及 CP0 寄存器的更新以及相关流水级的处理

二、实验设计（40%）

（一）总体设计思路

本次实验的原理并不复杂，把实验所需要的工作分为两部分，一部分是将现有的 tlb 模块接入之前完成的 CPU 中。第二部分是将 tlb 指令添加，这个部分耗费了我较长的时间去进行设计，tlbp 指令的设计比较简单，直接阻塞即可，对于 tlbr 和 tlbwi 指令，并没有采取讲义上设计的重取方式，而是为了尽可能的复用例外处理的逻辑，将写回级报出重取信号的指令定为 tlbwi 和 tlbr 指令本身而不是他们的下一条指令（这两条指令本身一拍就可以执行完成），在这种情况下，只需要在译码级发现当前指令是 tlbwi 或者 tlbr 时向取指级传递一个重取信号，if 阶段用寄存器存储下此时的 nextpc，当 wb 阶段报出 tlb 指令的刷新信号时，复用例外处理的 bus 和 bus 有效信号，重取之前存储的 pc 即可完成重取。

本次实验将 tlb 块作为与五个流水阶段并列的板块，在 sram_cpu 顶层文件中实现例化。其次在 cp0 寄存器中完成了新的 cp0 寄存器的设计。Tlb 与 cpu 的交互部分如下：s0 查找端口用于指令的地址映射，s1 查找端口用于数据的地址映射，主要在 IF，EXE 段进行相关更新。Cp0 寄存器的写入和读取功能在 WB 阶段实现。具体分功能描述如下：

1. IF 模块：利用 s0 查找端口，输出虚地址信息，接受 tlb 返回的查找信息，响应来自 WB 模块的 tlb_reflush 信号，同时保存指令重取情况出现的 pc。
2. EXE 模块：利用 s1 查找端口，输出虚地址信息，接受 tlb 返回的查找信息，同时接受 entryhi 寄存器的信息以实现查找和 tlbp 指令复用 s1 查找端口的设计要求。
3. WB&CP0 模块：cp0 模块中增添了 entryhi，entrylo0，entrylo1，index 等和 tlb 项有关的寄存器，在 WB 模块中完成了例化，同时在 tlbwi 和 tlbr 指令到达时发出 tlb_reflush 信号

4. 在 ID 阶段利用 br_bus 传递 refetch 信号，当重取信号到达时保存当时的 nextpc，以保证下一拍取得的是正确的 PC，还有一些流水向下的数据，模块中均有修改，见后面分模块详细设计部分。

CPU 总体设计图如下图所示：

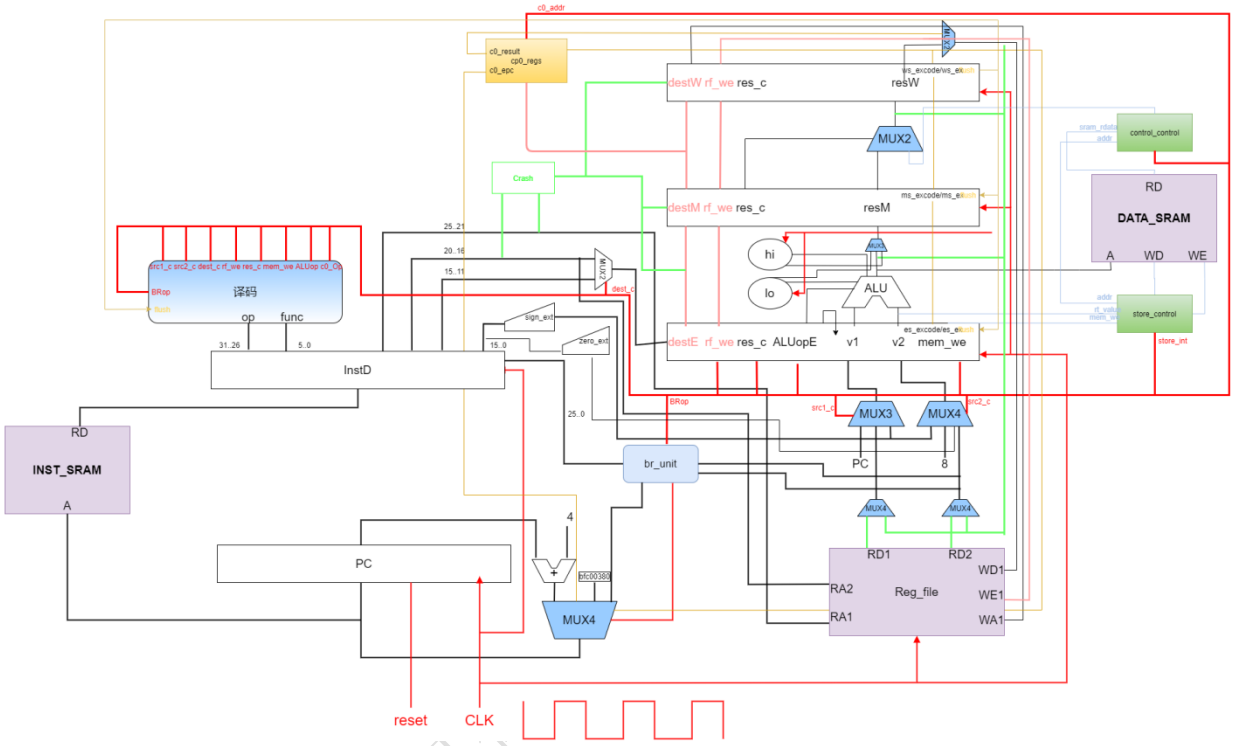

图 01: 整体 CPU 设计

（二）重要模块 1 设计：IF 模块（描述 lab14 更新）

1、工作原理（内容描述）

利用 TLB 表项的 s0 查找端口作为指令的虚实地址查找。 同时根据其余模块来的信号进行流水线刷新/重新取指令等。

2、接口定义（仅展示 lab14 更新）

名称	方向	位宽	功能描述
Tlb_reflush	IN	1	TLBWI 和 TLBR 的冲突消除信号
S0_vpn2	OUT	19	输入虚拟地址
S0_odd_page	OUT	1	是否为奇数页
S0_found	IN	1	输出是否找到
S0_index	IN	4	命中时 TLB 项数
S0_pfn	IN	20	物理页框号
S0_c	IN	3	Cache 属性
S0_d	IN	1	Dirty 位

名称	方向	位宽	功能描述
S0_v	IN	1	Valid(有效)位
Br_bus	IN	35	添加了 refetch 信号的 br 总线

3、功能描述

增添了 refetch_pc 寄存器，当出现需要 refetch 的情况下，利用该寄存器暂存 nextpc 的值，如下图 02 所示：

```
always@(posedge clk)
begin
    if(reset)
        refetch_pc <= 32'd0;
    else if( refetch )
        refetch_pc <= nextpc;
end
```

图 02: 时序 refetch_pc 的设计

当 tlbwi 或者 tlbr 指令到达写回级时刷新流水线的情况下，例外地址寄存器会选用 refetch_pc 作为下一拍跳转地址，如下图 03 所示：

```
always @(posedge clk)
begin
    if(eret_flush)
        ex_bus <= c0_epc;
    else if (ws_ex)
        ex_bus <= 32'hbfc00380;
    else if (tlb_reflush)
        ex_bus <= refetch_pc;
end
```

图 03: 新增例外地址寄存器的逻辑

对于 tlb_reflush 的响应处理与之前的 ws 级例外和 eret_flush 信号同级别处理，当出现即无效化下一拍的取值，由于涉及改动较多，这里仅贴出一个 always 块中的代码设计，表明如何进行无效化的：

```
 else if (ws_ex|eret_flush |tlb_reflush)
begin
    inst_bus_valid <= 1'b0;
end
```

图 04: 刷新掉整个流水的处理示例

相同的处理同样使用于 fs_valid, request_control, cancel， ex_eret, br_bus_valid 等一切和流水线控制有关的寄存器。

有关指令虚实映射发出的部分较为简单，就是使用 nextpc 作为地址查询，这部分设计将在 Lab15 中更改，以使得 CPU 能正确进行虚实地址转换，如下图 05 所示:

```
assign s0_vpn2 = nextpc[31:13];
assign s0_odd_page = nextpc[12];
```

图 05: 取值阶段指令虚地址转换发出信号

（三）重要模块 2 设计：ID 模块

1、工作原理

（lab14 新增）更新 br_bus，多了一位信号用于判断是否可能发生 tlb 冲突。向下一级流水的信号增添了 tlb 的三个指令

2、接口定义（lab14 更新）

名称	方向	位宽	功能描述
Br_bus	OUT	35	添加了 refetch 信号的 br 总线
ds_to_es_bus	OUT	213	添加了 3 个 tlb 指令的流水总线

3、功能描述

这部分设计较为简单，主要是巧妙的利用了 br_bus 来传输一位判断信号，减小了设计开销，译码和 refetch 信号如下所示：

```
assign inst_tlbp         = op_d[6'h10] & ds_inst[25] & (ds_inst[24:6] == 0) & func_d[6'h08];
assign inst_tlbr         = op_d[6'h10] & ds_inst[25] & (ds_inst[24:6] == 0) & func_d[6'h01];
assign inst_tlbwi        = op_d[6'h10] & ds_inst[25] & (ds_inst[24:6] == 0) & func_d[6'h02];
wire   refetch;
assign refetch = (inst_tlbwi | inst_tlbr) & ds_valid;
```

图 06:ID 阶段主要更新

部其余例行更新的 inst_valid，gr_we 信号等在此略去（注意三个 tlb 指令都不需要修改寄存器，所以 gr_we 需要额外取非），包括需要将新指令添加至保留指令中。

（四）重要模块 3 设计：EXE&MEM 模块（仅描述 lab14 更新）

1、工作原理

Lab14 中，exe 完成数据部分的虚实转换，同时该阶段执行 tlbp 操作。以及接受前面两级流水线传递的 mtc0 有效信号和 reflush 信号等。

MEM 模块仅多添加了 ms_mtc0 和 ms_tlb_reflesh 两个信号，较为简单，所以并在一起描述（也不在接口中重复叙述了）

2、接口定义（略去了简单的总线流水位宽信号改变）

名称	方向	位宽	功能描述
S1_vpn2	IN	19	输入虚拟地址
S1_odd_page	IN	1	是否为奇数页
S1_asid	IN	8	进程地址空间标志位
S1_found	OUT	1	输出是否找到
S1_index	OUT	4	命中时 TLB 项数
S1_pfn	OUT	20	物理页框号
S1_c	OUT	3	Cache 属性
S1_d	OUT	1	Dirty 位

名称	方向	位宽	功能描述
S1_v	OUT	1	Valid(有效)位
S1_vpn2	IN	19	输入虚拟地址
Es_tlbp	OUT	1	TLBP 使能信号
Ms_mtc0	IN	1	上次流水 ms 级
Ws_mtc0	IN	1	上次流水 ws 级 mtc0 信号
Ms_tlb_reflesh	IN	1	上次流水 ms 级 tlb 标志刷新信号
Tlb_reflesh	IN	1	上次流水 ws 级 mtc0 信号

3、功能描述

首先是 es 级的 we 信号需要在满足下两级流水的 tlb_reflesh 信号没报出的情况，这里也是相当于将 tlb 指令对流水级的刷新并入了例外处理中，即 exe 阶段能够写入内存的条件必须是 mem 和 wb 阶段不发生 tlb_reflush。

其次 es_ready_go 信号需要添加额外的阻塞判断，保证 mtc0 写入 entryhi 寄存器的情况和 tlbp 同时发生的情况下被阻塞在 es 阶段不向下一阶段流水，设计代码如下图所示：

```
assign es_ready_go    = flush ? 1'b1 :
                        (inst_tlbp & (ms_mtc0 | ws_mtc0)) ? 1'b0:
                        ((es_load_op | es_store) & (~data_sram_addr_ok
                        (es_alu_op[14] ? div_dout_tvalid:
                        es_alu_op[15] ? divu_dout_tvalid:
                        1'b1);//Update Lab6
```

图 07:es_ready_go 信号的阻塞逻辑

最后是数据部分的查找虚实地址转换，如下图所示：

```
//Lab 14
assign s1_vpn2     = (es_tlbp)? c0_entryhi[31:13] : es_alu_result[31:13];
assign s1_odd_page = (es_tlbp)? c0_entryhi[12]   : es_alu_result[12];
```

图 08:数据利用 s1 端口进行虚实地址转换

这里当出现 tlbp 指令（且本级流水有效）的情况下会复用查找端口，但是利用的是 c0_entryhi 寄存器的值

Mem 级涉及的改动很少，主要就是输出两个信号便于 EXE 级的阻塞，同时 wb 阶段也会输出类似的两个信号，逻辑如下图所示：

```
wire inst_tlbr;
wire inst_tlbwi;
assign ms_mtc0_valid = ms_valid & ms_mtc0 & (c0_addr == `CR_ENTRYHI);
assign ms_tlb_reflush = ms_valid & (inst_tlbr | inst_tlbwi);
```

图 09:MEM 级需要向外传递的两个信号

（五）重要模块 4 设计：cp0 模块（仅描述 lab14 更新）

1、工作原理

Lab14 中对 cp0 模块的改动较大，且和 tlb 的实现有较大关联，所以单独列出作为一个重要模块介绍。

2、接口定义

名称	方向	位宽	功能描述
Tlbp	IN	4	TLBP 指令
Tlbp_found	IN	19	数据寻找到的标志（s1）
tlbr	IN	8	Tlbr 指令
r_g	IN	1	读全局标志位
r_index	IN	4	读 TLB 项
r_vpn2	IN	19	读虚拟地址
r_asid	IN	8	读进程号
r_g	IN	1	读全局标志位
r_c0	IN	3	偶数 Cache 属性
r_d0	IN	1	偶数 Dirty 位
r_v0	IN	1	偶数 Valid(有效)位
r_c1	IN	3	奇数 Cache 属性
r_d1	IN	1	奇数 Dirty 位
r_v1	IN	1	奇数 Valid(有效)位
r_pfn0	IN	20	奇数页虚拟地址
r_pfn1	IN	20	偶数页虚拟地址
C0_entryhi	OUT	32	ENTRYhi 寄存器内容
C0_entrylo0	OUT	32	Lo0 寄存器内容
C0_entrylo1	OUT	32	Lo1 寄存器内容
C0_index	OUT	32	Index 寄存器内容

3、功能描述

首先是 extryhi 寄存器的处理,主要分为两个域，然后拼接后处理：

```
always @(posedge clk)
begin
    if(reset)
        c0_entryhi_vpn2 <= 19'd0;
    else if(mtc0_we && c0_addr == `CR_ENTRYHI)
        c0_entryhi_vpn2 <= c0_wdata[31:13];
    else if(tlbr)
        c0_entryhi_vpn2 <= r_vpn2;
    else if((wb_excode==5'h01||wb_excode==5'h02||wb_excode==5'h03)&&wb_ex)
        c0_entryhi_vpn2<= wb_badvaddr[31:13];
end

always @(posedge clk)
begin
    if(reset)
        c0_entryhi_asid <= 8'd0;
    else if(mtc0_we && c0_addr == `CR_ENTRYHI)
        c0_entryhi_asid <= c0_wdata[7:0];
    else if(tlbr)
        c0_entryhi_asid <= r_asid;
end
assign c0_entryhi = {
                    c0_entryhi_vpn2 , //31:13
                    5'b0            , //12:8
                    c0_entryhi_asid   //7:0
                    };
```

图 10:entry_hi 寄存器

对于 vpn 域，当指令为 mtc0 的时候写入 wdata 的 31~13 位，当为读 tlb 指令的时候将对应的虚拟地址写入，当为 wb 阶段发生例外的时候将对应例外虚地址写入。

对于 asid 域，逻辑更简单，不需要考虑发生例外写入例外虚地址情况，只是当为 tlbr 指令的时候写入的是 r_asid 的内容。

其次是 entrylo0 和 entrylo1 两个寄存器，由于除了奇偶以外都相同，这里仅展示 lo0 的涉及代码，如下图所示：

```
always @(posedge clk)
begin
    if(reset)
        c0_entrylo0_pfn2 <= 20'd0;
    else if(mtc0_we && c0_addr == `CR_ENTRYLO0)
        c0_entrylo0_pfn2 <= c0_wdata[25:6];
    else if(tlbr)
        c0_entrylo0_pfn2 <= r_pfn0;

end
always @(posedge clk)
begin
    if(reset)
        c0_entrylo0_C_D_V_G <= 6'd0;
    else if(mtc0_we && c0_addr == `CR_ENTRYLO0)
        c0_entrylo0_C_D_V_G <= c0_wdata[5:0];
    else if(tlbr)
        c0_entrylo0_C_D_V_G <= {r_c0,r_d0,r_v0,r_g};
end
assign c0_entrylo0 = {
                    6'b0                ,  //31:26
                    c0_entrylo0_pfn2    ,  //25:6
                    c0_entrylo0_C_D_V_G    //5:0
                    };
```

图 11:entry_lo0 寄存器

基本设计思路相同，当为 mtc0 且写寄存器号匹配的时候写入 wdata，tlbr 的时候利用 tlb 读出信息更新寄存器，再将两个域拼接起来。

最后是 index 寄存器，如下图 12 所示：

（节省页面空间直接在图上面进行叙述）按照设计要求，当数据找到的情况下会将 index 的高位 p 位置 0，否则会置 1，剩余域的处理方式同上两个寄存器，mtc0 写，tlbr 的时候进行更新，最后域拼接。

```
always@(posedge clk)
begin
    if(reset)
        c0_index_p <= 1'b0;
    else if(tlbp & ~tlbp_found)
        c0_index_p <= 1'b1;
    else if(tlbp & tlbp_found)
        c0_index_p <= 1'b0;
end

always@(posedge clk)
begin
    if(reset)
        c0_index_index <= 4'h0;
    else if(mtc0_we && c0_addr == `CR_INDEX  )
        c0_index_index <= c0_wdata[3:0];
    else if(tlbp & tlbp_found)
        c0_index_index <= index;
end

assign c0_index = {
                   c0_index_p     ,//31
                   27'b0          ,//30:4
                   c0_index_index //3:0
                   };
```

（六）重要模块 5 设计：WB 模块（仅描述 lab14 更新）

3、工作原理

Lab14 中 WB 主要完成了例化新 cp0，完成 tlbp 和 cp0 的更新，向外发出 tlb 标志刷新信号等功能

4、接口定义

名称	方向	位宽	功能描述
r_g	IN	1	读全局标志位
r_index	IN	4	读 TLB 项
r_vpn2	IN	19	读虚拟地址
r_asid	IN	8	读进程号
r_g	IN	1	读全局标志位
r_c0	IN	3	偶数 Cache 属性
r_d0	IN	1	偶数 Dirty 位
r_v0	IN	1	偶数 Valid(有效)位
r_c1	IN	3	奇数 Cache 属性
r_d1	IN	1	奇数 Dirty 位
r_v1	IN	1	奇数 Valid(有效)位
r_pfn0	IN	20	奇数页虚拟地址
r_pfn1	IN	20	偶数页虚拟地址

名称	方向	位宽	功能描述
Es_tlbp	IN	1	Tlbp 使能信号（建立在 exe 阶段流水有效情况下）
Tlbp_found	IN	1	实际为 s1_found 信号
C0_entryhi	OUT	32	ENTRYhi 寄存器内容
C0_entrylo0	OUT	32	Lo0 寄存器内容
C0_entrylo1	OUT	32	Lo1 寄存器内容
C0_index	OUT	32	Index 寄存器内容
Tlb_reflush	OUT	1	标志刷新信号
Ws_mtc0	OUT	1	传至 exe 阻塞相关信号
Inst_tlbwi	OUT	1	传至 tlb 的 we 信号

3、功能描述

向外传递的 flush 信号需要加入是否为标志刷新的判断，代码略去。

更新的两个向外传递的信号逻辑如下图所示：

```
assign tlb_reflush = ws_valid ? (inst_tlbr|inst_tlbwi):1'b0;
assign ws_mtc0     = mtc0_we & (c0_addr == `CR_ENTRYHI);
```

图 13:向外传递信号

其中 mtc0 冲突只会发生在是对 entryhi 寄存器修改的情况，所以需要额外判断。

例化 cp0 部分略去，同时对于 mfc0 的 result 进行更新如下所示：

```
assign c0_result = (c0_addr == `CR_STATUS)   ? c0_status   :
                   (c0_addr == `CR_CAUSE)    ? c0_cause    :
                   (c0_addr == `CR_EPC)      ? c0_epc      :
                   (c0_addr == `CR_COMPARE)  ? c0_compare  :
                   (c0_addr == `CR_COUNT)    ? c0_count    :
                   (c0_addr == `CR_BADVADDR) ? c0_badvaddr :
                   (c0_addr == `CR_ENTRYHI)  ? c0_entryhi  :
                   (c0_addr == `CR_ENTRYLO0) ? c0_entrylo0 :
                   (c0_addr == `CR_ENTRYLO1) ? c0_entrylo1 :
                   (c0_addr == `CR_INDEX)    ? c0_index    :
                   32'b0;
```

图 14:c0_result 的额外选择器

三、实验过程（50%）

（一）实验流水账

12.3 19:00-22:00 12.4 16:00-16:30 构思设计进行代码编写并 debug 成功

12.14 开始写实验报告，零零碎碎大约花费 4 小时完成。

（二）错误记录

1、错误 1：通过前 4 个测试点，后两个错误

（1） 错误现象

通过前四个测试，但无法通过后两个测试点：

```
Test begin!
----[  14155 ns] Number 8'd01 Functional Test Point PASS!!!
       [  22000 ns] Test is running, debug_wb_pc = 0xbfc00d58
----[  23445 ns] Number 8'd02 Functional Test Point PASS!!!
       [  32000 ns] Test is running, debug_wb_pc = 0xbfc00b7c
----[  34485 ns] Number 8'd03 Functional Test Point PASS!!!
       [  42000 ns] Test is running, debug_wb_pc = 0xbfc00ca4
----[  43635 ns] Number 8'd04 Functional Test Point PASS!!!
       [  52000 ns] Test is running, debug_wb_pc = 0xbfc0084c
------------------------------------------------------------
[  52875 ns] Error(  0)!!! Occurred in number 8'd05 Functional Test Point!
------------------------------------------------------------
------------------------------------------------------------
[  58645 ns] Error(  1)!!! Occurred in number 8'd06 Functional Test Point!
------------------------------------------------------------
       [  62000 ns] Test is running, debug_wb_pc = 0xbfc006c4
============================================================
Test end!
Fail!!!Total   2 errors!
```

图 15:错误 1 的报错图

（2）分析定位过程

前四个测试点只测了新加的 cp0 寄存器，没有测试新指令，猜测是新指令出错，查找波形发现，应该存的重取的 pc 多变化了几次，根据波形判断本应该只拉高一拍的译码级重取信号拉高了很多拍，如下图所示。

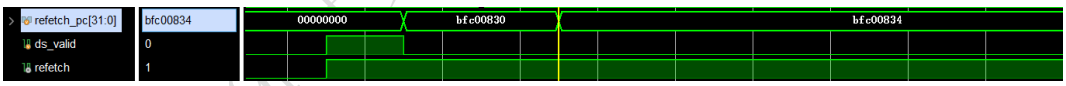

图 16:refetch 拉高无法降低的波形图

（3）错误原因

refetch 信号没有与 ID 阶段的有效信号相与，导致流水级有效信号拉低的的情况下还会重取错误的 pc

（4）修正效果，

修改后代码见图 06，修改后成功通过测试点。

四、实验总结（可选）

这次实验并不困难，第一部分代码我认为是体力活，需要大量重复的接口连接，是非常细致琐碎的工作，也是在我代码编写中花费时间最多的一部分，却没有什么理解上的难度。第二部分对于讲义设计的更改则是基于我偷懒和提高处理器性能的想法而实现的，代码构思的时间大部分花在这上面。在和助教交流这种设计思路的时候，也知道了讲义上设计的重取标志信号在指令系统特权态设计上的意义，而不是仅仅服务于这一次实验，由于本学

期实验还不涉及到特权态的区分，因此我还是继续采用了这样可以提高性能的设计。