计算机体系结构大作业

- 专业：物联网工程
- 姓名：秦泽斌
- 学号：2212005

一、实验任务

请结合实验指导手册中的《课程设计拓展 优化 CPU 系统》章节实验，完成对五级流水线CPU的优化工作，要求如下：

1. 完成旁路技术和前向传递技术的改进（基本要求40%，必须完成）
2. 完成对目前流水线CPU的指令扩展，要求I型，J型，R型至少各扩展1条指令。（基本要求20%，若无法实现3）中的问题，可做此部分，保证60%的完成度）
3. 按照《CPU设计实战》这本书中的内容，依次完成异常和中断、AXI总线、TLB和Cache功能的添加。（若选择此思路，第2）条可以不做。此部分为提高要求，占比60%，做了尝试或者失败的情况及分析都可以写在报告里面，写了就20%起步）

二、实验准备

本次实验基于计算机体系结构第二次实验，即**静态5级流水线CPU实现**

操作系统	Windows11家庭版
Vivado版本	2017.4
代码资源	计算机体系结构lab2——pipeline
参考资料	《CPU设计实战》(汪文祥, 邢金璋)

三、实验原理

1. 旁路技术和向前传递技术

旁路技术（Bypass or Forwarding）是现代处理器中常用的一种数据通路优化技术，用于解决流水线中的**数据相关性**问题，从而提高指令执行效率，避免不必要的流水线停顿（Pipeline Stall）。当流水线中某条指令需要使用前面指令计算的结果，但结果尚未写回寄存器时，旁路技术可以直接将结果从流水线的某个阶段转发（Forward）到需要该数据的指令，而不必等到写回寄存器再读取。

1.1 实现过程：

1. 检测数据相关性：
    - 流水线控制单元会检测后续指令是否依赖于当前流水线中某条指令的计算结果。
2. 数据转发路径：
    - 在硬件上增加旁路路径，将结果从执行阶段（EX）或存储访问阶段（MEM）直接传递给后续指令。
3. 控制逻辑：
    - 根据相关性判断，旁路控制逻辑决定是否启用旁路路径，将正确的数据转发到对应的流水线阶段。

1.2 旁路技术解决的问题：

以简单的RISC-V流水线为例：

```
add x1, x2, x3   # 指令1：x1 = x2 + x3
sub x4, x1, x5   # 指令2：x4 = x1 - x5
```

在经典五级流水线中，指令1在执行阶段（EX）计算结果，而指令2在解码阶段（ID）需要读取x1的值。如果没有旁路技术，指令2必须等待指令1将结果写回寄存器（WB阶段），这会导致流水线停顿。

引入旁路技术后，指令1的结果可以直接从EX阶段转发给指令2的ID阶段，消除了等待时间，提高了执行效率。

而以上这种旁路技术的实现，我们又可以称之为“向前传递技术”。

---

2. 异常和中断

CPU的例外和中断支持是处理器设计中一个重要部分，用于应对外部事件（中断）和内部异常（例外）。它们能够让处理器在运行中断正常指令流，快速响应特定事件，并在完成处理后恢复执行。

2.1 定义

例外是由CPU内部发生的事件触发的，如指令错误、内存访问异常等。它通常表示程序运行中出现了无法继续执行的问题，需要操作系统或CPU介入处理。

中断是由CPU外部事件触发的，如外设请求、计时器超时等。它通常用于处理异步事件，让处理器能够对外界变化做出反应。

2.2 处理流程

1. 保存当前程序的状态（如PC值）。
2. 当位例外时，转到例外处理程序（操作系统内核中的特定函数）；当位中断时，进入中断向量表中指定的处理程序。
3. 完成处理后返回，恢复原程序的执行（如果可能）。

---

3. AXI总线

3.1 定义

AXI（Advanced eXtensible Interface）是ARM公司设计的一种高性能、低延迟的片上系统（SoC）通信协议。它是AMBA（Advanced Microcontroller Bus Architecture）协议家族的一部分，广泛应用于嵌入式系统、物联网设备以及高性能处理器中。AXI协议定义了主设备（Master）和从设备（Slave）之间的通信规则。
数据传输通过以下五个独立的通道进行：

1. 读地址通道（AR）：
    - 传输主设备请求读取数据的地址和控制信息。
2. 读数据通道（R）：
    - 传输从设备响应的读数据和相关信息。
3. 写地址通道（AW）：
    - 传输主设备请求写入数据的地址和控制信息。

4. 写数据通道（W）：
    - 传输主设备写入的数据。
5. 写响应通道（B）：
    - 传输从设备对写操作的响应。

每条通道都具有独立的握手机制，使得操作可以并行进行，提高总线效率。

3.2 AXI协议的关键特性

- 分离的读写通道：
    - 读写操作互不阻塞，支持全双工通信。
- 突发传输（Burst Transfer）：
    - 主设备可以在一次请求中传输多个数据，减少控制开销。
    - 支持固定地址、递增地址和环形地址模式。
- 握手机制：
    - 基于`VALID`和`READY`信号，确保数据可靠传输。
- 低延迟：
    - 通过流水线操作实现高速通信，减少传输延迟。
- 支持QoS（服务质量）：
    - 允许为不同的事务设置优先级，提高系统资源利用率。

---

4. TLB

TLB（Translation Lookaside Buffer）是一种用于加速虚拟地址到物理地址转换的硬件缓存，广泛应用于现代处理器中。它是内存管理单元（MMU）中的关键组件，通过减少访问页表的次数，提高内存访问效率。

在支持虚拟内存的系统中，CPU执行程序时会生成**虚拟地址**，而实际访问内存需要将虚拟地址转换为**物理地址**。这个转换过程需要查询**页表（Page Table）**，如果每次内存访问都查询页表，会导致性能严重下降。

TLB通过缓存最近使用的虚拟地址到物理地址的映射关系，减少页表查询次数，从而提高效率。

4.1 工作流程

1. 虚拟地址生成：
    - CPU生成一个虚拟地址。
2. 查询TLB：
    - 在TLB中查找虚拟地址对应的物理地址。
    - 如果找到（命中），直接返回物理地址。
    - 如果未找到（未命中），查询页表并将结果存入TLB。
3. 内存访问：
    - 使用物理地址访问主存或高速缓存。

4.2 TLB的组成

- 页表项缓存：

    - TLB中存储的是页表中的部分条目，包括：
        - 虚拟页号（VPN）：虚拟地址的高位部分。
        - 物理页号（PPN）：物理地址的高位部分。
        - 相关标志位：如有效位（Valid）、读写权限（RW）、脏页（Dirty）等。
- 关联映射方式：
    - **全相联映射**：任意条目都可以存储在TLB中的任意位置（效率高，但硬件复杂）。
    - **组相联映射**：结合直接映射和全相联的优点。
    - **直接映射**：简单易实现，但可能增加冲突。

---

5. Cache

Cache（缓存）是计算机中用于提高数据访问速度的一种存储器。它是一种高速度、容量较小的存储设备，位于CPU与主内存之间，用于存储最近或最常访问的数据。通过减少CPU访问主内存的频率，缓存显著提高了计算机系统的性能。

缓存的基本原理是**局部性原理**，即程序在运行时，数据访问存在一定的规律性，主要包括：

- **时间局部性**：如果某个数据被访问过，它很可能会在不久后再次被访问。
- **空间局部性**：如果某个数据被访问，周围的数据也很可能会被访问。

基于局部性原理，Cache将最近访问的数据和指令存储在高速缓存中，以便快速响应未来的访问请求。

5.1 工作流程

1. CPU请求数据：
    - 当CPU需要访问数据时，它首先查询L1 Cache。
2. Cache命中（Hit）：
    - 如果数据在Cache中找到了（命中），则直接从Cache中读取数据。
3. Cache未命中（Miss）：
    - 如果数据没有在Cache中（未命中），则从较低级别的缓存或主内存中加载数据，并将其存入Cache中，以供下次访问使用。

5.2 Cache替换策略

由于Cache容量有限，当Cache满时，必须决定哪些数据被替换掉。常见的替换策略包括：

- LRU（Least Recently Used，最少使用）：
    - 替换最近最少访问的数据。
- FIFO（First In First Out，先进先出）：
    - 按照数据加载的顺序进行替换。
- Random（随机替换）：
    - 随机选择一个缓存条目进行替换。

Cache作为一种高速存储设备，通过减少CPU访问主内存的次数来提高系统性能。它通过多级缓存、替换策略、一致性协议等技术，优化数据访问并加速程序的执行。Cache在RISC-V及其他处理器架构中都起着至关重要的作用，是提升计算机性能的关键组件之一。

四、实验内容

1. 旁路技术与向前传递技术

首先，旁路技术与向前传递技术主要时为了解决可能存在的**数据冒险**，那么大概可能存在集中情况呢，这里我们做了三种假设：

- 相邻指令间存在数据相关，即流水线译码、执行阶段存在数据相关；
- 相隔一条指令的指令间存在数据相关，即流水线译码、访存阶段存在数据相关；
- 相隔两条指令的指令间存在数据相关，即流水线译码、写回阶段存在数据相关。

针对上面的三种情况，接下里我们进行代码编写

1.1 decode.v

在译码阶段，我们需要为上面的三种情况添加对应阶段的写使能、写数据

```
input      wire EXE_write,
input      wire[ 31:0]EXE_wdata,
input      wire MEM_write,
input      wire[ 31:0]MEM_wdata,
input      wire WB_write,
input      wire[ 31:0]WB_wdata
```

数据前递意味着参与ALU计算，我们需要增加逻辑判断部分，来判断什么时候我们需要使用数据前递，即我们应该对可能的数据冒险进行检测，至于检测条件，这里我们给出：

- 对于EX冒险，可以得到两对前递条件：

  `EX/MEM.RegisterRd = ID/EX.RegisterRs1`

  `EX/MEM.RegisterRd = ID/EX.RegisterRs2`

  表示ID/EX寄存器堆中的rs1或rs2与EX/MEM寄存器堆中的rd相同。

- 对于MEM冒险，可以得到两对前递条件：

  `MEM/WB.RegisterRd = ID/EX.RegisterRs1`

  `MEM/WB.RegisterRd = ID/EX. RegisterRs2`

  表示ID/EX寄存器堆中的rs1或rs2与MEM/WB寄存器堆中的rd相同。

故我们修改alu中代码如下：

```
assign alu_operand1 =
(~inst_no_rs&&rs!=5'd0&&rs==EXE_wdest&&EXE_write) ? EXE_wdata:

 (~inst_no_rs&&rs!=5'd0&&rs==MEM_wdest&&MEM_write) ? MEM_wdata:

 (~inst_no_rs&&rs!=5'd0&&rs==WB_wdest&&WB_write) ? WB_wdata:
                          inst_j_link ? pc :
                          inst_shf_sa ? {27'd0,sa} :
                          rs_value;
 assign alu_operand2 =     inst_imm_zero ? {16'd0, imm} :
                          inst_imm_sign ?  {{16{imm[15]}}, imm}:

 (~inst_no_rt&&rt!=5'd0&&rt==EXE_wdest&&EXE_write) ? EXE_wdata:

 (~inst_no_rt&&rt!=5'd0&&rt==MEM_wdest&&MEM_write) ? MEM_wdata:
```

```
 (~inst_no_rt&&rt!=5'd0&&rt==WB_wdest&&WB_write) ? WB_wdata:
                          inst_j_link ? 32'd8 :
                          rt_value;
```

这段赋值语句的作用是判断当前译码的指令需要的rs和rt寄存器**是否**就是当前EXE/MEM/WB 阶段中要写入的寄存器，如果是，则将数据直接传递成为alu操作数，否则正常赋值。 同样地，我们也需要修改**SW指令**相关的取值语句：

```
assign store_data =
(~inst_no_rt&&rt!=5'd0&&rt==EXE_wdest&&EXE_write) ? EXE_wdata:

 (~inst_no_rt&&rt!=5'd0&&rt==MEM_wdest&&MEM_write) ? MEM_wdata:

 (~inst_no_rt&&rt!=5'd0&&rt==WB_wdest&&WB_write) ?
WB_wdata:rt_value;
```

最后，我们还应该将原先的延迟等待部分删除，对ID_over信号的赋值语句进行修改：

```
assign ID_over = ID_valid & (~inst_jbr | IF_over);
```

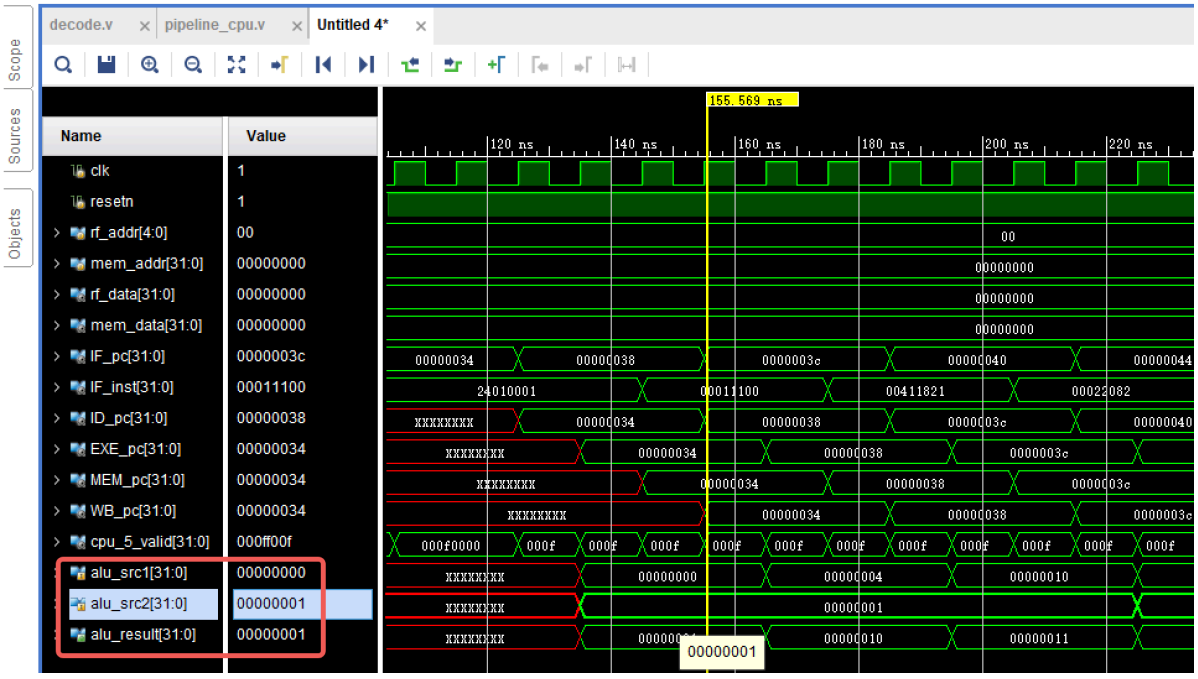
1.2 pipeline_cpu.v

一个另外的必不可缺的工作是在整体模块中为对应的接口进行关联操作，如下：

```
decode ID_module(                // 译码级
       .ID_valid   (ID_valid   ),  // I, 1
       .IF_ID_bus_r(IF_ID_bus_r),  // I, 64
       .rs_value   (rs_value   ),  // I, 32
       .rt_value   (rt_value   ),  // I, 32
       .rs         (rs         ),  // O, 5
       .rt         (rt         ),  // O, 5
       .jbr_bus    (jbr_bus    ),  // O, 33
       //.inst_jbr   (inst_jbr   ),  // O, 1
       .ID_over    (ID_over    ),  // O, 1
       .ID_EXE_bus (ID_EXE_bus ),  // O, 167

       //5级流水新增
       .IF_over     (IF_over     ),// I, 1
       .EXE_wdest   (EXE_wdest   ),// I, 5
       .MEM_wdest   (MEM_wdest   ),// I, 5
       .WB_wdest    (WB_wdest    ),// I, 5

       //展示PC
       .ID_pc       (ID_pc       ), // O, 32

       .EXE_write    (EXE_MEM_bus[116]),
       .EXE_wdata    (EXE_MEM_bus[67:36]),
       .MEM_write    (MEM_WB_bus[0]),
       .MEM_wdata    (MEM_WB_bus[37:6]),
       .WB_write     (rf_wen),
       .WB_wdata     (rf_wdata)
    );
```

1.3 仿真测试

经过仿真模拟测试，模拟的流水线CPU能够正常运行，且通过对对应值的监听可以看到，前递正常实现。



从仿真图中可以看出，在34H指令还未进入到WB阶段时，执行38H指令的ALU模块已经获取到了即将写入1号寄存器的值1，以准备进行后续的计算。下图是测试指令：

34H	addiu $1, $0,#1	[$1] = 0000_0001H	24010001	0010_0100_0000_0001_0000_0000_0000_0001
38H	sll $2, $1,#4	[$2] = 0000_0010H	00011100	0000_0000_0000_0001_0001_0001_0000_0000

至此，一组简单的旁路技术和向前传递技术大致完成了。

2. 异常和中断

从这里开始，我们将开始尝试完成实验的提高部分，主要依据《CPU设计实战》这本书来尝试构建'异常和中断"部分。

首先，我们需要引入**CP0寄存器**的概念，**CP0寄存器**负责管理异常和中断的处理，保存相关的状态信息，并控制异常和中断的行为。其功能涵盖了从处理异常事件到控制中断优先级等多个方面。关于CP0的理论知识这里不再详细介绍，但是对于实现异常和中断来说，CP0寄存器必不可少。

其次，我们要介绍MFC0指令和MTC0指令：

> MIPS指令系统中并没有定义直接操作CP0寄存器的算术逻辑运算类指令，而是定义了 MFC0和MTC0指令用于在通用寄存器和CP0寄存器之间交互数据。举例来说，假如例外处理程序想查询Cause寄存器中的Excode域以确定当前处理的是哪种例外，那么它需要先用 MFC0指令将Cause寄存器的值取到某个通用寄存器中、然后将这个通用寄存器中的第6.2位 提取出来，得到Cause寄存器Excode域的值。再例如，如果想将Stams寄存器的IIE位置0以 屏蔽所有的中断,该如何操作呢?由于此时只是修改Status寄存器的一部分'因此软件需要先 用MFC0将StatUs寄存器的值取到某个通用寄存器，然后通过逻辑位与操作将这个通用寄存器 的第0位置0而保持其他位不变'最后用MTC0将这个通用寄存器的值写回Status寄存器中。

然后，我们开始为CPU添加syscall例外支持，其主要步骤为：

1. 为CPU增加MTC0、MFC0、ERET指令。
2. 为CPU增加CP0寄存器Status、Cause、EPC。

3. 为CPU增加SYSCALL指令,也就是增加syscall例外支持（已在coe文件中存在）

2.1 增加MTC0、MFC0、ERET指令

增加指令的操作与之前实验的步骤基本相同，主要操作文件为ID.v（译码阶段），这里只展示部分代码

```
assign inst_eret    = op_d[6'h10] & rs_d[5'h10] & func_d[6'h18] & rd_d[5'h00] & rt_d[5'h00] & sa_d[5'h00];
assign inst_mfc0    = op_d[6'h10] & rs_d[5'h00] & sa_d[5'h00] & (ds_inst[5:3] == 3'b0);
assign inst_mtc0    = op_d[6'h10] & rs_d[5'h04] & sa_d[5'h00] & (ds_inst[5:3] == 3'b0);
```

除此以外，需要在各个文件中为新添加的指令设置端口，配置操作重复性多，这里也不再展示

2.2 增加CP0寄存器Status、Cause、EPC

这里我们需要为CP0新建一个源文件，cp0.v。在本文件中，我们需要实现Status、Cause、EPC这三种CP0寄存器，来完成对中断和异常的控制。下面的三段代码分别为其实现：

```
//CP0寄存器status
reg cp0_status_ie;
always @(posedge clk) begin
    if(rst)
        cp0_status_ie <= 1'b0;
    else if(mtc0_we && cp0_addr == `CP0_STATUS_ADDR)
        cp0_status_ie <= cp0_wdata[0];
end

assign cp0_status =
{
    9'b0,               //31:23
    cp0_status_bev,     //22:22
    6'b0,               //21:16
    cp0_status_im,      //15:8
    6'b0,     //7:2
    cp0_status_exl,     //1:1
    cp0_status_ie       //0:0
};

//CP0_CAUSE
reg cp0_cause_bd;
always @(posedge clk) begin
    if(rst)
        cp0_cause_bd <= 1'b0;
    else if(wb_ex && !cp0_status_exl)
        cp0_cause_bd <= wb_bd;
end
```

```
//CP0寄存器CP0_CAUSE
reg cp0_cause_bd;
always @(posedge clk) begin
    if(rst)
        cp0_cause_bd <= 1'b0;
```

```
    else if(wb_ex && !cp0_status_exl)
        cp0_cause_bd <= wb_bd;
end

reg cp0_cause_ti;
wire count_eq_compare;
assign count_eq_compare = (cp0_count == cp0_compare);

always @(posedge clk) begin
    if(rst)
        cp0_cause_ti <= 1'b0;
    else if(mtc0_we && cp0_addr == `CP0_COMP_ADDR)
        cp0_cause_ti <= 1'b0;
    else if(count_eq_compare)
        cp0_cause_ti <= 1'b1;
end

reg [7:0]cp0_cause_ip;
always @(posedge clk) begin
    if(rst)
        cp0_cause_ip[7:2] <= 6'b0;
    else begin
        cp0_cause_ip[7] <= ext_int_in[5] | cp0_cause_ti;
        cp0_cause_ip[6 : 2] <= ext_int_in[4:0];
    end
end

always @(posedge clk) begin
    if(rst)
        cp0_cause_ip[1:0] <= 2'b0;
    else if(mtc0_we && cp0_addr == `CP0_CAUSE_ADDR)
        cp0_cause_ip[1:0] <= cp0_wdata[9:8];
end

reg [4:0] cp0_cause_excode;
always @(posedge clk) begin
    if(rst)
        cp0_cause_excode <= 5'b0;
    else if(wb_ex)
        cp0_cause_excode <= wb_excode;
end

assign cp0_cause =
{
    cp0_cause_bd,       //31:31
    cp0_cause_ti,       //30:30
    14'b0,              //29:16
    cp0_cause_ip,       //15:8
    1'b0,               //7:7
    cp0_cause_excode,   //6:2
    2'b0        //1:0
};
```

```
//CP0寄存器EPC
reg [31:0] c0_epc;
always @(posedge clk) begin
    if(wb_ex && !cp0_status_exl)
        c0_epc <= wb_bd ? wb_pc - 32'h4 : wb_pc;
    else if(mtc0_we && cp0_addr == `CP0_EPC_ADDR)
        c0_epc <= cp0_wdata;
end

assign cp0_epc = c0_epc;
```

2.3 为CPU项目增加支持syscall例外的支持

在写回阶段WB.v中，增加以下代码，在这些代码中我们处理了异常、中断和控制处理器寄存器（CP0）之间的交互。它通过多个信号与`cp0`模块进行通信，处理指令的执行、异常发生、外部中断输入和数据写入等过程。`ws_valid`表示指令是否有效，`ws_ex`表示是否发生异常，`cp0_we`控制是否写入CP0寄存器等。这些信号和逻辑组合确保了异常和中断的正确处理，并生成相关的调试信息以供分析。

```
wire [5:0]      ext_int_in;
wire [31:0]     cp0_rdata;
wire            cp0_we;
wire [31:0]     cp0_wdata;


wire [31:0]     ws_cp0_epc;
wire [31:0]     ws_cp0_status;
wire [31:0]     ws_cp0_cause;

//valid
assign ws_inst_mfc0_o = ws_valid && ws_inst_mfc0;
assign ws_rf_dest = ws_valid ? ws_dest : 5'b0;

assign ws_ex_o = ws_valid && ws_ex;
// assign cp0_epc = ws_valid && ws_cp0_epc;
assign cp0_epc = {32{ws_valid}} & ws_cp0_epc;
assign cp0_cause = {32{ws_valid}} & ws_cp0_cause;
assign cp0_status = {32{ws_valid}} & ws_cp0_status;


//init
assign ext_int_in = 6'b0;
assign ws_eret = ws_inst_eret && ws_valid;

// TODO
assign rf_we    = {4{ ws_valid & ~ws_ex }} & ws_gr_strb;
assign rf_waddr = ws_dest;
assign rf_wdata = ws_inst_mfc0 ? cp0_rdata :
                  ws_final_result;

// debug info generate
assign debug_wb_pc       = ws_pc;
assign debug_wb_rf_wen   = rf_we;
assign debug_wb_rf_wnum  = ws_dest;
assign debug_wb_rf_wdata = rf_wdata;
```

```
37
38 assign cp0_we = ws_inst_mtc0 && ws_valid && !ws_ex;
39 assign cp0_wdata = ws_final_result;
40
41 cp0 u_cp0(
42     .clk                (clk),
43     .rst                (reset),
44
45     .wb_ex              (ws_ex),
46     .wb_bd              (ws_bd),
47     .wb_excode          (ws_excode),
48     .wb_pc              (ws_pc),
49     .wb_badvaddr        (ws_badvaddr),
50     .ws_eret          (ws_eret),
51     .ext_int_in         (ext_int_in),
52
53     .cp0_addr           (cp0_addr),
54     .cp0_rdata          (cp0_rdata),
55     .mtc0_we            (cp0_we),
56     .cp0_wdata          (cp0_wdata),
57
58     .cp0_epc            (ws_cp0_epc),
59     .cp0_status         (ws_cp0_status),
60     .cp0_cause          (ws_cp0_cause)
61 );
```

至于在译码阶段对于异常的总控，我们在完成对其他例外的支持后统一完成

2.4 增加break、地址错、整数溢出、保留指令例外支持

在译码阶段增加break的实现

```
1 assign inst_break   = op_d[6'h00] & func_d[6'h0d];
```

在取指阶段增加地址错误的实现

```
1 wire addr_error;
2 assign addr_error = (fs_pc[1:0] != 2'b0);
3 assign fs_ex = addr_error && fs_valid;
4 assign fs_bd = ds_is_branch;
5 assign fs_badvaddr = fs_pc;
```

其他例外的支持与上面类似，其中对于整数溢出主要考虑以下内容

> ADD、ADDI和SUB这三条指令如果在计算过程中发生溢出，则发牛辖型溢出例外。附录D关于这三条指令的定义中给出了几种判断溢出例外的方法，不过需要将ALU中加法 器改为33位数据宽。如果要延续之前给出的参考代码中的32位加法器，那么可以根据人判；溢出的思路来生成溢出信号，即对于加法，正数加正数得到负数'或者负数加负数得到正数；对于减法'正数减负数得到负数，或者负数减正数得到正数。

对于保留指令例外来说

> 如果译码时发现指令码不是一条指令系统规范中定义的指令,那么就认为发生了保留指令例外。

2.5 为CPU增加CP0寄存器Count、Compare、BadVAddr

这些CP0寄存器的实现过程与之前的实现过程基本类似：

```
//BADVADDR
reg [31:0] c0_badvaddr;
always @(posedge clk) begin
    if(wb_ex && wb_excode == `CP0_BADV_ADDR)
        c0_badvaddr <= wb_badvaddr;
end

assign cp0_badvaddr = c0_badvaddr;

//COUNT
reg tick;
always @(posedge clk) begin
    if(rst)
        tick <= 1'b0;
    else
        tick <= ~tick;
end

reg [31:0] c0_count;
always @(posedge clk) begin
    if(rst)
        c0_count <= 32'b0;
    else if(mtc0_we && cp0_addr == `CP0_COUNT_ADDR)
        c0_count <= cp0_wdata;
    else if(tick)
        c0_count <= c0_count + 1'b1;
end

assign cp0_count = c0_count;

//COMPARE
reg [31:0]c0_compare;
always @(posedge clk) begin
    if(mtc0_we && cp0_addr == `CP0_COMP_ADDR)
        c0_compare <= cp0_wdata;
end
```

2.6 为异常和中断控制增加总控

在译码阶段ID.v中，增加以下代码：

```
wire other_inst;
assign other_inst = !(inst_addu | inst_subu | inst_slt | inst_sltu |
inst_and | inst_or | inst_xor | inst_nor
| inst_sll | inst_srl | inst_sra | inst_addiu | inst_lui | inst_lw | inst_sw
| inst_beq | inst_bne | inst_jal
| inst_jr | inst_add | inst_addi | inst_sub | inst_slti | inst_sltiu |
inst_andi | inst_ori | inst_xori | inst_sllv
| inst_srlv | inst_srav | inst_mult | inst_multu | inst_div | inst_divu |
inst_mfhi | inst_mflo | inst_mthi | inst_mtlo
```

```
| inst_bgez | inst_bgtz | inst_blez | inst_bltz | inst_j | inst_bltzal | inst_bgezal | inst_jalr | inst_lb | inst_lbu
| inst_lh | inst_lhu | inst_lwl | inst_lwr | inst_sb | inst_sh | inst_swl | inst_swr | inst_syscall | inst_eret | inst_mfc0
| inst_mtc0 | inst_break);

wire interrupt;

assign interrupt = ((cp0_cause[15:8] & cp0_status[15:8]) != 8'b0) && (cp0_status[1:0] == 2'b01);


assign ds_ex = (fs_to_ds_ex | inst_syscall | inst_break | other_inst | interrupt) & ds_valid;

assign ds_excode = (interrupt) ? `EX_INT :
                   (fs_to_ds_ex) ? `EX_ADEL :
                   (other_inst) ? `EX_RI :
                   (inst_syscall) ? `EX_SYS :
                   (inst_break) ? `EX_BP : `EX_NO;
```

至此，我们已经为CPU增加包括系统调用，break，地址错误等等多种异常和例外的支持。

3. AXI总线

接下来我们将尝试进行AXI总线的设计，那么为什么要有总线呢

> 在大多数真实的计算机系统中，CPU通过总线与系统中的内存、外设进行交互。没有总线'，CPU就是个"光杆司令"，什么工作也做不了。总线接口可以自行定义，也可以遵照工业 界的标准。显然，遵照工业界的标准有助于与大量第三方的IP进行集成。

在之前的实验中，CPU一直采用的是SRAM接口，为了改造成AXI总线接口，我们将本节目标主要分为以下三个阶段：

1. 阶段一:将原有CPU访问SRAM的接口调整为类SRAM总线接口。类SRAM总线只是在SRAM接口的基础上增加了握手信号，可以降低直接实现AXI总线的设计复杂度。
2. 阶段二:实现一个AXI总线接口信号，准备与上一阶段的类SRAM总线接口对齐。
3. 阶段二:设计实现一个"类SRAM-AXI"的转接桥，拼接上阶段一完成的CPU。

3.1 类SRAM总线接口的设计

什么是类SRAM总线呢？用一句话来说就是，添加了握手机制的SRAM接口。下图介绍了类SRAM总线各接口信号

表 8-1　类 SRAM 总线接口信号

信号	位宽	方向	功能
clk	1	input	时钟
req	1	master → slave	请求信号，为 1 时有读写请求，为 0 时无读写请求
wr	1	master → slave	为 1 表示该次是写请求，为 0 表示该次是读请求
size	[1:0]	master → slave	该次请求传输的字节数，0: 1byte；1: 2bytes；2: 4bytes
addr	[31:0]	master → slave	该次请求的地址
wstrb	[3:0]	master → slave	该次写请求的字节写使能
wdata	[31:0]	master → slave	该次写请求的写数据
addr_ok	1	slave → master	该次请求的地址传输 OK，读：地址被接收；写：地址和数据被接收
data_ok	1	slave → master	该次请求的数据传输 OK，读：数据返回；写：数据写入完成
rdata	[31:0]	slave → master	该次请求返回的读数据

我们需要将原有的访问SRAM的接口调整为类SRAM总线接口，其中包括`inst_sram`和`data_sram`，除此以外另一个需要注意的内容是握手机制的实现。握手机制的实现主要由size、addr_ok、data_ok三个信号完成，对于size信号，因为AXI总线协议上有arsize和awsize信号，所以需要把这个信号通 过类SRAM接口传送给AXI接口。至于addr_ok、data_ok这两个信号，就复杂得多。

addr_ok信号用于和req信号一起完成读写请求的握手。只有在clk的上升沿同时看到 req和addr_ok为1的时候才是一次成功的请求握手。data _ok信号有双重身份。对应读事务的时候它是数据返回的有效信号;对应写事务的时候,它是写人完成的有效信号。无论dataok表达的是对读事务的问应还是对写事务的问应，统称为数据响应。在类SRAM接口中，Master对于数据响应总是可以接收，所以不再设置Master接收dataok的握手信号。也就是说，如果存在未返回数据响应的请求，则在 clk的上升沿看到dataok为1就可以认为是一次成功的数据响应握手。

在CPU的顶层模块中，我们实现了类SRAM总线的设计

```
    // inst sram interface
    wire        inst_sram_req;
    wire        inst_sram_wr;
    wire [ 1:0] inst_sram_size;
    wire [31:0] inst_sram_addr;
    wire [ 3:0] inst_sram_wstrb;
    wire [31:0] inst_sram_wdata;
    wire        inst_sram_addr_ok;
    wire        inst_sram_data_ok;
    wire [31:0] inst_sram_rdata;
    // data sram interface
    wire        data_sram_req;
    wire        data_sram_wr;
    wire [ 1:0] data_sram_size;
    wire [31:0] data_sram_addr;
    wire [ 3:0] data_sram_wstrb;
    wire [31:0] data_sram_wdata;
    wire        data_sram_addr_ok;
    wire        data_sram_data_ok;
    wire [31:0] data_sram_rdata;
```

当然，我们还需要针对各个源文件中的接口进行类似更改，例如在exe.v（执行阶段）中，接口修改如下:

```
    // data sram interface
    // output        data_sram_en   ,
    // output [ 3:0] data_sram_wen  ,
    // output [31:0] data_sram_addr ,
    // output [31:0] data_sram_wdata,
    output          data_sram_req,
    output          data_sram_wr,
    output  [ 1:0]  data_sram_size,
    output  [31:0]  data_sram_wdata,
    output  [ 3:0]  data_sram_wstrb,
    output  [31:0]  data_sram_addr,
    input           data_sram_addr_ok,
    input   [31:0]  data_sram_rdata,
    input           data_sram_data_ok,
    output          es_data_waiting,
```

其他文件中的修改这里不再一一展示。

3.2 实现AXI总线接口信号

以下是部分AXI总线接口信号，我们需要实现具体的AXI接口信号。

表 8-4　32 位 AXI 接口信号一览

信号	位宽	方向	功能	备注
AXI 时钟与复位信号				
aclk	1	input	AXI 时钟	
aresetn	1	input	AXI 复位，低电平有效	
读请求通道，(以 ar 开头)				
arid	[3:0]	master → slave	读请求的 ID 号	取指置为 0；取数置为 1
araddr	[31:0]	master → slave	读请求的地址	
arlen	[7:0]	master → slave	读请求控制信号，请求传输的长度（数据传输拍数）	固定为 0
arsize	[2:0]	master → slave	读请求控制信号，请求传输的大小（数据传输每拍的字节数）	
arburst	[1:0]	master → slave	读请求控制信号，传输类型	固定为 0b01
arlock	[1:0]	master → slave	读请求控制信号，原子锁	固定为 0
arcache	[3:0]	master → slave	读请求控制信号，Cache 属性	固定为 0
arprot	[2:0]	master → slave	读请求控制信号，保护属性	固定为 0

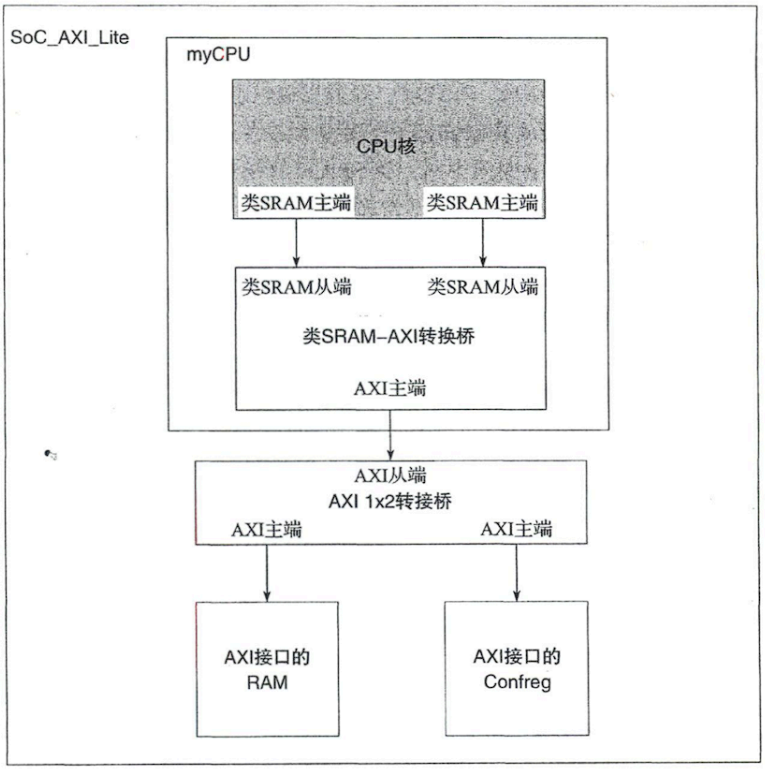
AXI总线协议非常复杂也细节居多，这里直接给出在顶层模块中对AXI接口信号的实现

```
    input   [ 5:0]      int,
    input               aclk,
    input               aresetn,
    //axi interface
    //read request
    output  [ 3:0]      arid,
    output  [31:0]      araddr,
    output  [ 7:0]      arlen,
    output  [ 2:0]      arsize,
    output  [ 1:0]      arburst,
    output  [ 1:0]      arlock,
    output  [ 3:0]      arcache,
    output  [ 2:0]      arprot,
    output              arvalid,
    input               arready,

    //read response
    input   [ 3:0]      rid,
    input   [31:0]      rdata,
    input   [ 1:0]      rresp,
    input               rlast,
    input               rvalid,
    output              rready,

    //write request
    output  [ 3:0]      awid,
    output  [31:0]      awaddr,
    output  [ 7:0]      awlen,
    output  [ 2:0]      awsize,
    output  [ 1:0]      awburst,
```

```
    output  [ 1:0]      awlock,
    output  [ 3:0]      awcache,
    output  [ 2:0]      awprot,
    output              awvalid,
    input               awready,

    //write data
    output  [ 3:0]      wid,
    output  [31:0]      wdata,
    output  [ 3:0]      wstrb,
    output              wlast,
    output              wvalid,
    input               wready,

    //write response
    input   [ 3:0]      bid,
    input   [ 1:0]      bresp,
    input               bvalid,
    output              bready,

    // trace debug interface
    output [31:0] debug_wb_pc,
    output [ 3:0] debug_wb_rf_wen,
    output [ 4:0] debug_wb_rf_wnum,
    output [31:0] debug_wb_rf_wdata
```

3.3 实现“类SRAM-AXI”的转接桥，拼接上阶段一完成的CPU

目前我们设计的CPU有一个指令段的类SRAM接口和一个数据段的类SRAM接口。我们设计的转接桥是供CPU使用的，所以它要有两个类SRAM接口。作为一个简单的CPU，对外通常有一个总线接口，所以我们设计的转接桥要有一个AXI接口。整个转接桥与CPU 其余部分，以及与SoC中的其他部分的关系如下图所示。



于是，我们新建一个名为`transfer_bridge.v`的新源文件。在这个文件中，有很多设计要求，包括但不限于：

1. aresem有效期间，AXIMaster端的所有valid类输出必须为0'所有ready类输出不能为X值。
2. AXIMaster端的所有valid输出信号的置l逻辑中，一定不能允许组合逻辑来自同一通道的ready输入信号（时序逻辑来自ready是可以的）。
3. 对于AXIMaster端的所有ready输出信号'一定不能允许组合逻辑来自同一通道的valid输入信号（时序逻辑来自valid是可以的）。
4. 无论读、写请求、类SRAM接口上的事务要和AXI接口上的事务严格一一对应。特 别要注意'既不要将类SRAM接口上的一个事务在AXI总线上重复发送多次，也不要将类 SRAM发来的多个地址连续的读事务或写事务合并成一个AXI总线事务。
5. 在AXIMaster端的读请求、写请求、写数据通道上'如果Master输出的valjd置为 l的时候对应的ready是0,那么在ready信号变为1之前'不允许Master变更该通道的所 有输出信号。这一点与类SRAM总线不同,类SRAM总线允许中途更改请求。

故文件部分核心代码如下：

```
/************ define ************/
/* AXI read request */
reg         axi_ar_busy;
reg  [ 3:0] axi_ar_id;
reg  [31:0] axi_ar_addr;
reg  [ 2:0] axi_ar_size;

/* AXI read response */
wire        axi_r_data_ok;
wire        axi_r_inst_ok;
wire [31:0] axi_r_data;

/* AXI write request */
reg         axi_aw_busy;
reg  [31:0] axi_aw_addr;
reg  [ 2:0] axi_aw_size;
reg         axi_w_busy;
reg  [31:0] axi_w_data;
reg  [ 3:0] axi_w_strb;

/* AXI write response */
wire        axi_b_ok;

/* middle read request */
wire        read_req_sel_data;
wire        read_req_sel_inst;

wire        read_req_valid;
wire [ 3:0] read_req_id;
wire [31:0] read_req_addr;
wire [ 2:0] read_req_size;

wire        read_data_req_ok;
wire        read_inst_req_ok;

/* middle read inst response */
```

```
wire        read_inst_resp_wen;
wire        read_inst_resp_ren;
wire        read_inst_resp_empty;
wire        read_inst_resp_full;
wire [31:0] read_inst_resp_input;
wire [31:0] read_inst_resp_output;
fifo_buffer #(
    .DATA_WIDTH     (32),
    .BUFF_DEPTH     (6),
    .ADDR_WIDTH     (3)
) read_inst_resp_buff (
    .clk            (aclk),
    .resetn         (aresetn),
    .wen            (read_inst_resp_wen),
    .ren            (read_inst_resp_ren),
    .empty          (read_inst_resp_empty),
    .full           (read_inst_resp_full),
    .input_data     (read_inst_resp_input),
    .output_data    (read_inst_resp_output)
);

/* middle read data response */
wire        read_data_resp_wen;
wire        read_data_resp_ren;
wire        read_data_resp_empty;
wire        read_data_resp_full;
wire [31:0] read_data_resp_input;
wire [31:0] read_data_resp_output;
fifo_buffer #(
    .DATA_WIDTH     (32),
    .BUFF_DEPTH     (6),
    .ADDR_WIDTH     (3)
) read_data_resp_buff (
    .clk            (aclk),
    .resetn         (aresetn),
    .wen            (read_data_resp_wen),
    .ren            (read_data_resp_ren),
    .empty          (read_data_resp_empty),
    .full           (read_data_resp_full),
    .input_data     (read_data_resp_input),
    .output_data    (read_data_resp_output)
);

/* middle write request */
wire        write_req_valid;
wire [31:0] write_req_addr;
wire [ 2:0] write_req_size;
wire [31:0] write_req_data;
wire [ 3:0] write_req_strb;
wire        write_data_req_ok;

/* middle write response */
wire        write_data_resp_wen;
wire        write_data_resp_ren;
wire        write_data_resp_empty;
wire        write_data_resp_full;
```

```
fifo_count #(
    .BUFF_DEPTH     (6),
    .ADDR_WIDTH     (3)
) write_data_resp_count (
    .clk            (aclk),
    .resetn         (aresetn),
    .wen            (write_data_resp_wen),
    .ren            (write_data_resp_ren),
    .empty          (write_data_resp_empty),
    .full           (write_data_resp_full)
);

/* SRAM inst request */
wire        inst_read_valid;
wire        inst_related;  // TODO

/* SRAM inst response */
wire        inst_read_ready;

/* SRAM data request */
wire        data_read_valid;
wire        data_write_valid;
wire        data_related;

// request record
wire        data_req_record_wen;
wire        data_req_record_ren;
wire        data_req_record_empty;
wire        data_req_record_full;
wire        data_req_record_related_1;
wire [32:0] data_req_record_input;      // {wr, addr}
wire [32:0] data_req_record_output;     // {wr, addr}
fifo_buffer_valid #(
    .DATA_WIDTH     (33),
    .BUFF_DEPTH     (6),
    .ADDR_WIDTH     (3),
    .RLAT_WIDTH     (32)
) data_req_record (
    .clk            (aclk),
    .resetn         (aresetn),
    .wen            (data_req_record_wen),
    .ren            (data_req_record_ren),
    .empty          (data_req_record_empty),
    .full           (data_req_record_full),
    .related_1      (data_req_record_related_1),
    .input_data     (data_req_record_input),
    .output_data    (data_req_record_output),
    .related_data_1 (data_sram_addr)
);

/* SRAM data response */
wire        data_read_ready;
wire        data_write_ready;


/************ assign ************/
```

```
/* AXI read request */
always @ (posedge aclk) begin
    if (!aresetn) begin
        axi_ar_busy <= 1'b0;
        axi_ar_id   <= 4'h0;
        axi_ar_addr <= 32'h0;
        axi_ar_size <= 3'h0;
    end else if (!axi_ar_busy && read_req_valid) begin
        axi_ar_busy <= 1'b1;
        axi_ar_id   <= read_req_id;
        axi_ar_addr <= read_req_addr;
        axi_ar_size <= read_req_size;
    end else if (axi_ar_busy && arvalid && arready) begin
        axi_ar_busy <= 1'b0;
        axi_ar_id   <= 4'h0;
        axi_ar_addr <= 32'h0;
        axi_ar_size <= 3'h0;
    end
end
assign arvalid  = axi_ar_busy;
assign arid     = axi_ar_id;
assign araddr   = axi_ar_addr;
assign arsize   = axi_ar_size;

/* AXI read response */
assign axi_r_data_ok = rvalid && rready && rid == DATA_ID;
assign axi_r_inst_ok = rvalid && rready && rid == INST_ID;
assign axi_r_data    = rdata;

assign rready = !read_inst_resp_full && !read_data_resp_full;

/* AXI write request */
always @ (posedge aclk) begin
    if (!aresetn) begin
        axi_aw_busy <= 1'b0;
        axi_aw_addr <= 32'h0;
        axi_aw_size <= 3'h0;
    end else if (!axi_aw_busy && !axi_w_busy && write_req_valid) begin
        axi_aw_busy <= 1'b1;
        axi_aw_addr <= write_req_addr;
        axi_aw_size <= write_req_size;
    end else if (axi_aw_busy && awvalid && awready) begin
        axi_aw_busy <= 1'b0;
        axi_aw_addr <= 32'h0;
        axi_aw_size <= 3'h0;
    end
    if (!aresetn) begin
        axi_w_busy  <= 1'b0;
        axi_w_data  <= 32'h0;
        axi_w_strb  <= 4'h0;
    end else if (!axi_aw_busy && !axi_w_busy && write_req_valid) begin
        axi_w_busy  <= 1'b1;
        axi_w_data  <= write_req_data;
        axi_w_strb  <= write_req_strb;
    end else if (axi_w_busy && wvalid && wready) begin
        axi_w_busy  <= 1'b0;
```

```
205         axi_w_data  <= 32'h0;
206         axi_w_strb  <= 4'h0;
207     end
208 end
209 assign awvalid  = axi_aw_busy;
210 assign wvalid   = axi_w_busy;
211 assign awaddr   = axi_aw_addr;
212 assign awsize   = axi_aw_size;
213 assign wdata    = axi_w_data;
214 assign wstrb    = axi_w_strb;
215
216 /* AXI write response */
217 assign axi_b_ok = bvalid && bready;
218 assign bready = !write_data_resp_full;
219
220 /* middle read request */
221 assign read_req_sel_data    = data_read_valid;
222 assign read_req_sel_inst    = !data_read_valid && inst_read_valid;;
223
224 // to axi
225 assign read_req_valid       = inst_read_valid || data_read_valid;
226 assign read_req_id          = read_req_sel_data ? DATA_ID : INST_ID;
227 assign read_req_addr        = read_req_sel_data ? data_sram_addr :
    inst_sram_addr;
228 assign read_req_size        = read_req_sel_data ? data_sram_size :
    inst_sram_size;
229
230 // to sram
231 assign read_data_req_ok = read_req_sel_data && !axi_ar_busy;
232 assign read_inst_req_ok = read_req_sel_inst && !axi_ar_busy;
233
234 /* middle read inst response */
235 assign read_inst_resp_ren = inst_read_ready;
236 assign read_inst_resp_wen = axi_r_inst_ok;
237 assign read_inst_resp_input = axi_r_data;
238
239 /* middle read data response */
240 assign read_data_resp_ren = data_read_ready;
241 assign read_data_resp_wen = axi_r_data_ok;
242 assign read_data_resp_input = axi_r_data;
243
244 /* middle write request */
245 // to axi
246 assign write_req_valid      = data_write_valid;
247 assign write_req_addr       = data_sram_addr;
248 assign write_req_size       = data_sram_size;
249 assign write_req_data       = data_sram_wdata;
250 assign write_req_strb       = data_sram_wstrb;
251
252 // to sram
253 assign write_data_req_ok = data_write_valid && !axi_aw_busy && !axi_w_busy;
254
255 /* middle write response */
256 assign write_data_resp_ren = data_write_ready;
257 assign write_data_resp_wen = axi_b_ok;
258
```

```
/* SRAM inst request */
assign inst_read_valid = inst_sram_req && !inst_sram_wr && !inst_related;
assign inst_sram_addr_ok = read_inst_req_ok;
assign inst_related = 0;

/* SRAM inst response */
assign inst_read_ready = 1;
assign inst_sram_data_ok = !read_inst_resp_empty;
assign inst_sram_rdata = read_inst_resp_output;

/* SRAM data request */
assign data_related      = data_req_record_related_1;
assign data_read_valid   = data_sram_req && !data_sram_wr && !data_related;
assign data_write_valid  = data_sram_req && data_sram_wr && !data_related;
assign data_sram_addr_ok = read_data_req_ok || write_data_req_ok;

// request record
assign data_req_record_ren = data_sram_data_ok;
assign data_req_record_wen = data_sram_req && data_sram_addr_ok;
assign data_req_record_input = {data_sram_wr, data_sram_addr};

/* SRAM data response */
assign data_sram_rdata  = read_data_resp_output;
assign data_read_ready  = !data_req_record_empty &&
!data_req_record_output[32];
assign data_write_ready = !data_req_record_empty &&
data_req_record_output[32];

assign data_sram_data_ok =
    (data_read_ready && !read_data_resp_empty) ||
    (data_write_ready && !write_data_resp_empty);

endmodule
```

代码有点长，很多都是对于接口的定义。在这个文件中，我们完成了从类SRAM总线到AXI总线的转换。至此，AXI总线设计基本结束。

4. TLB

从这一节开始，我们又将进人一个新的设计阶段，即在现有的CPU中添加TLB MMU 的支持。这一部分的设计难点在于涉及的技术细节较多，不容易分出主次。因此，我们将整个设计分为三个阶段：

1. 第一阶段：我们专注于TLB模块自身的设计。
2. 第二阶段：我们将TLB模块集成至上一节改造完的CPU中，将维护TLB涉及的指令和CP0寄存器等要素在CPU中予以实现。
3. 第三阶段：我们将TLB相关例外的支持添加完毕。

4.1 TLB模块的设计

存储管理是现代操作系统的重要功能。基于页表的页式存储管理是最为常见的存储管理方式。TLB是内存中的页表在CPU内部的高速缓存。TLB MMU是CPU内部基于TLB实现 的内存管理逻辑。显然，这套MMU至少具备两项功能：一是完成虚实地址转换，二是能够被操作系统访问管理。 TLB模块是TLBMMU中的核心模块，那么它也要支持这两项主要功能。

通过梳理TLB模块相关知识,并结合CPU流水线的结构特点,我们对TLB模块的设计分析如下:

1. TLB模块内部的主体应是一个二维组织结构的查找表。查找表的每一项分为两个部分,第一部分存储的信息既参与读写又参与查找比较,包括VPN2、ASID、G;第二部分仅参与读写,包括PFNO、C0、D0、V0、PFN1、C1、D1、V1。查找表的项数由实现者自行定义。
2. TLB模块要支持取指和访存两个部分的虚实地址转换需求,即都需要对TLB模块进行查找。如果不考虑取指部分,只有读操作的差异,则两部分对应的查找功能一致。查找时,需要向TLB模块输入s_vpn2、s_odd_page和s_asid信息,1TLB模块输出的信息包含sfound、s_pfn、s_c、s_d、s_v。其中输入的s_vpn2来自访存虚地址的31..13位,s_odd_page来自访存虚地址的第12位,sasid来自CPOEntryHi寄存器的ASID域。TLB输出的s_pfin用于产生最终的物理地址,sfound的结果用于判定是否产生TLB重填例外,s_found和s_v结果用于判定是否产生TLB无效例外,s_found、s_v和s_d结果用于:判定是否产生TLB修改例外。
3. 为了使流水线能够满负荷运转不断流,TLB模块要能够支持取指和访存同时进行查找,这意味着上面的查找端口应该有两套。
4. TLB模块需要支持TLBP指令的查找操作。我们倾向于复用访存查找的端口。输入复用s_vpn2和s_asid。输出除了复用已有的s_found外还需要一个额外的s_index输出,用于记录命中在第几项。
5. TLB模块需要支持TLBWI指令的写操作。我们倾向于为此设置独立的端口。此时需要向TLB模块输入写地址w_index,以及其他写入信息w_vpin2、w_asid、w_g、w_pfn0、w_c0、w_d0、w_v0、w_pfnl、w_cl、w_dl、w_vl。因为是写操作,所以必须有一个写使能输入信号we。
6. TLB模块需要支持TLBR指令的读操作。我们倾向于为此设计独立的端口。此时需要向TLB模块输入读地址r_index。TLB模块需要输出读的结果有r_vpn2、r_asid、r_g、r_g、r_pfn0、r_c0、r_d0、r_v0、r_pfnl、r_cl、r_dl、r_vl。

通过上述分析,我们得到TLB模块的接口与内部主要信号的定义如下:(在文件tlb_mmu.v中)

```
module tlb
#(
    parameter TLBNUM = 16
)
(
    input           clk,
    input           reset,

    // search port 0
    input  [              18:0] s0_vpn2,
    input                       s0_odd_page,
    input  [               7:0] s0_asid,
    output                      s0_found,
    output [$clog2(TLBNUM)-1:0] s0_index,
    output [              19:0] s0_pfn,
    output [               2:0] s0_c,
    output                      s0_d,
    output                      s0_v,

    // search port 1
    input  [              18:0] s1_vpn2,
    input                       s1_odd_page,
    input  [               7:0] s1_asid,
    output                      s1_found,
    output [$clog2(TLBNUM)-1:0] s1_index,
    output [              19:0] s1_pfn,
    output [               2:0] s1_c,
    output                      s1_d,
    output                      s1_v,
```

```
30
31      // write port
32      input                          we,
33      input  [$clog2(TLBNUM)-1:0] w_index,
34      input  [              18:0] w_vpn2,
35      input  [               7:0] w_asid,
36      input                          w_g,
37      input  [              19:0] w_pfn0,
38      input  [               2:0] w_c0,
39      input                          w_d0,
40      input                          w_v0,
41      input  [              19:0] w_pfn1,
42      input  [               2:0] w_c1,
43      input                          w_d1,
44      input                          w_v1,
45
46      // read port
47      input  [$clog2(TLBNUM)-1:0] r_index,
48      output [              18:0] r_vpn2,
49      output [               7:0] r_asid,
50      output                         r_g,
51      output [              19:0] r_pfn0,
52      output [               2:0] r_c0,
53      output                         r_d0,
54      output                         r_v0,
55      output [              19:0] r_pfn1,
56      output [               2:0] r_c1,
57      output                         r_d1,
58      output                         r_v1
59  );
60
61  reg  [      18:0] tlb_vpn2     [TLBNUM-1:0];
62  reg  [       7:0] tlb_asid     [TLBNUM-1:0];
63  reg               tlb_g        [TLBNUM-1:0];
64
65  reg  [      19:0] tlb_pfn0     [TLBNUM-1:0];
66  reg  [       2:0] tlb_c0       [TLBNUM-1:0];
67  reg               tlb_d0       [TLBNUM-1:0];
68  reg               tlb_v0       [TLBNUM-1:0];
69
70  reg  [      19:0] tlb_pfn1     [TLBNUM-1:0];
71  reg  [       2:0] tlb_c1       [TLBNUM-1:0];
72  reg               tlb_d1       [TLBNUM-1:0];
73  reg               tlb_v1       [TLBNUM-1:0];
74
75  wire [TLBNUM -1:0] match0;
76  wire [TLBNUM -1:0] match1;
77
78  wire [$clog2(TLBNUM)-1:0] s0_index_arr [TLBNUM -1:0];
79  wire [$clog2(TLBNUM)-1:0] s1_index_arr [TLBNUM -1:0];
80
81  //Search
82  assign s0_found = (|match0) ? 1'b1 : 1'b0;
83  assign s1_found = (|match1) ? 1'b1 : 1'b0;
84
85  assign s0_index = s0_index_arr[TLBNUM -1];
```

```
86  assign s1_index = s1_index_arr[TLBNUM -1];
87
88  assign s0_pfn = s0_odd_page ? tlb_pfn1[s0_index] : tlb_pfn0[s0_index];
89  assign s0_c = s0_odd_page ? tlb_c1[s0_index] : tlb_c0[s0_index];
90  assign s0_d = s0_odd_page ? tlb_d1[s0_index] : tlb_d0[s0_index];
91  assign s0_v = s0_odd_page ? tlb_v1[s0_index] : tlb_v0[s0_index];
92
93  assign s1_pfn = s1_odd_page ? tlb_pfn1[s1_index] : tlb_pfn0[s1_index];
94  assign s1_c = s1_odd_page ? tlb_c1[s1_index] : tlb_c0[s1_index];
95  assign s1_d = s1_odd_page ? tlb_d1[s1_index] : tlb_d0[s1_index];
96  assign s1_v = s1_odd_page ? tlb_v1[s1_index] : tlb_v0[s1_index];
97
98  genvar tlb_i;
99  generate for (tlb_i = 0; tlb_i < TLBNUM; tlb_i = tlb_i + 1) begin:gen_tlb
100
101     assign match0[tlb_i] = (s0_vpn2 == tlb_vpn2[tlb_i]) && ((s0_asid ==
    tlb_asid[tlb_i]) || tlb_g[tlb_i]);
102     assign match1[tlb_i] = (s1_vpn2 == tlb_vpn2[tlb_i]) && ((s1_asid ==
    tlb_asid[tlb_i]) || tlb_g[tlb_i]);
103
104     if (tlb_i == 0) begin
105         assign s0_index_arr[tlb_i] = {$clog2(TLBNUM){match0[tlb_i]}} &
    tlb_i;
106         assign s1_index_arr[tlb_i] = {$clog2(TLBNUM){match1[tlb_i]}} &
    tlb_i;
107     end else begin
108         assign s0_index_arr[tlb_i] = s0_index_arr[tlb_i - 1] |
    ({$clog2(TLBNUM){match0[tlb_i]}} & tlb_i);
109         assign s1_index_arr[tlb_i] = s1_index_arr[tlb_i - 1] |
    ({$clog2(TLBNUM){match1[tlb_i]}} & tlb_i);
110     end
111
112 //Write
113     always @(posedge clk) begin
114         if (reset) begin
115             tlb_vpn2[tlb_i] <= 0;
116             tlb_asid[tlb_i] <= 0;
117             tlb_g   [tlb_i] <= 0;
118
119             tlb_pfn0[tlb_i] <= 0;
120             tlb_c0  [tlb_i] <= 0;
121             tlb_d0  [tlb_i] <= 0;
122             tlb_v0  [tlb_i] <= 0;
123
124             tlb_pfn1[tlb_i] <= 0;
125             tlb_c1  [tlb_i] <= 0;
126             tlb_d1  [tlb_i] <= 0;
127             tlb_v1  [tlb_i] <= 0;
128         end else if (we && w_index == tlb_i) begin
129             tlb_vpn2[tlb_i] <= w_vpn2;
130             tlb_asid[tlb_i] <= w_asid;
131             tlb_g   [tlb_i] <= w_g;
132
133             tlb_pfn0[tlb_i] <= w_pfn0;
134             tlb_c0  [tlb_i] <= w_c0;
135             tlb_d0  [tlb_i] <= w_d0;
```

```
            tlb_v0  [tlb_i] <= w_v0;

            tlb_pfn1[tlb_i] <= w_pfn1;
            tlb_c1  [tlb_i] <= w_c1;
            tlb_d1  [tlb_i] <= w_d1;
            tlb_v1  [tlb_i] <= w_v1;
        end
    end

end endgenerate

//Read
assign r_vpn2 = tlb_vpn2[r_index];
assign r_asid = tlb_asid[r_index];
assign r_g    = tlb_g[r_index];

assign r_pfn0 = tlb_pfn0[r_index];
assign r_c0   = tlb_c0[r_index];
assign r_d0   = tlb_d0[r_index];
assign r_v0   = tlb_v0[r_index];

assign r_pfn1 = tlb_pfn1[r_index];
assign r_c1   = tlb_c1[r_index];
assign r_d1   = tlb_d1[r_index];
assign r_v1   = tlb_v1[r_index];

endmodule

module vpaddr_transfer (
    input   [31:0]  vaddr,
    output  [31:0]  paddr,
    output          tlb_refill,
    output          tlb_invalid,
    output          tlb_modified,

    input           inst_tlbp,
    input   [31:0]  cp0_entryhi,

    output  [18:0]  tlb_vpn2,
    output          tlb_odd_page,
    output  [ 7:0]  tlb_asid,
    input           tlb_found,
    input   [19:0]  tlb_pfn,
    input   [ 2:0]  tlb_c,
    input           tlb_d,
    input           tlb_v
);

wire unmapped;
assign unmapped = vaddr[31] & !vaddr[30];
// assign unmapped = 1'b1;

assign tlb_vpn2 = (inst_tlbp)? cp0_entryhi[31:13] : vaddr[31:13];
assign tlb_odd_page = vaddr[12];
assign tlb_asid = cp0_entryhi[7:0];

```

```
assign paddr = (unmapped)? {3'b0, vaddr[28:0]} : {tlb_pfn, vaddr[11:0]};
// assign paddr = vaddr;

assign tlb_refill   = !unmapped && !tlb_found;
assign tlb_invalid  = !unmapped && tlb_found && !tlb_v;
assign tlb_modified = !unmapped && tlb_found && tlb_v && !tlb_d;

endmodule
```

这段代码实现了一个简单的**TLB（Translation Lookaside Buffer）**模块和相关的虚拟地址到物理地址转换模块。TLB是虚拟内存系统中的关键组件，用于加速虚拟地址到物理地址的转换过程。整个设计可以分为两部分：TLB模块和地址转换模块。

首先，TLB模块通过两个**搜索**端口、一个写入端口和一个读取端口实现了多功能的地址映射和管理功能。TLB存储了多个条目，每个条目包含虚拟页号（VPN）、地址空间标识（ASID）、全局标志（G）、以及用于物理地址转换的页帧号（PFN）、缓存属性（C）、修改位（D）、和有效位（V）。每次搜索操作会根据输入的虚拟页号和地址空间标识在TLB中匹配条目，返回对应的物理页帧号以及相关属性。匹配逻辑通过`genvar`生成硬件，逐条检查是否满足搜索条件，并输出匹配结果的索引和对应的属性。写入操作通过`w_index`指定更新的条目，同时对TLB中对应位置的内容进行更新。读取操作允许用户根据`r_index`获取指定条目的所有字段。

其次，`vpaddr_transfer`模块实现了**虚拟地址到物理地址**的转换逻辑。该模块接收一个32位的虚拟地址，结合TLB查询的结果，决定生成的物理地址。如果地址属于非映射区域（由高位标志判定），直接生成未经过TLB处理的物理地址；否则，根据TLB查询的页帧号组合生成最终物理地址。同时，模块还生成TLB异常标志，包括TLB未命中（refill）、无效条目（invalid）、和修改位检查（modified），用以支持处理器对地址异常情况的处理。此外，模块还支持`inst_tlbp`操作，允许处理器查询特定虚拟地址对应的TLB条目。

4.2 添加TLB相关指令和CP0寄存器

我们需要实现的TLB相关的CP0寄存器有EntryHi、EntryLo0、EntryLol和Index。这 几个CP0寄存器均可被MFC0读取，其中的大多数域均能被MTC0更新。因此，我们倾向于将其和已实现的CP0寄存器放在一个模块中维护，这样MFC0和MTC0指令访问这几个Cp0寄存器的数据通路就可以复用现有的设计。不过，这几个CP0寄存器还会被TLB指令更新，并被TLB指令和硬件逻辑读取。这就势必引人其他围绕这些CP0寄存器的相关关系，需要我们通过设计来确保这些相关关系不会引发冲突。

以下是CP0寄存器的添加：

- EntryHi

```
//ENTRYHI
reg [18:0] entry_hi_vpn2;
always @(posedge clk) begin
    if(wb_ex && excode_tlb)
        entry_hi_vpn2 <= wb_badvaddr[31:13];
    else if(mtc0_we && cp0_addr == `CP0_ENTRYHI_ADDR)
        entry_hi_vpn2 <= cp0_wdata[31:13];
    else if(tlbr)
        entry_hi_vpn2 <= r_vpn2;
end

reg [7:0] entry_hi_asid;
always @(posedge clk) begin
    if (rst) begin
        entry_hi_asid <= 8'b0;
```

```
    end else if(mtc0_we && cp0_addr == `CP0_ENTRYHI_ADDR)
        entry_hi_asid <= cp0_wdata[7:0];
    else if(tlbr)
        entry_hi_asid <= r_asid;
end

assign cp0_entryhi =
{
    entry_hi_vpn2,       //31:13
    5'b0,                //12:8
    entry_hi_asid        //7:0
};
```

- EntryLo0

```
//ENTRYLO0
reg [19:0] entrylo0_pfn;
always @(posedge clk) begin
    if(mtc0_we && cp0_addr == `CP0_ENTRYLO0_ADDR)
        entrylo0_pfn <= cp0_wdata[25:6];
    else if(tlbr)
        entrylo0_pfn <= r_pfn0;
end

reg [2:0] entrylo0_c;
always @(posedge clk) begin
    if(mtc0_we && cp0_addr == `CP0_ENTRYLO0_ADDR)
        entrylo0_c <= cp0_wdata[5:3];
    else if(tlbr)
        entrylo0_c <= r_c0;
end

reg entrylo0_d;
always @(posedge clk) begin
    if(mtc0_we && cp0_addr == `CP0_ENTRYLO0_ADDR)
        entrylo0_d <= cp0_wdata[2];
    else if(tlbr)
        entrylo0_d <= r_d0;
end

reg entrylo0_v;
always @(posedge clk) begin
    if(mtc0_we && cp0_addr == `CP0_ENTRYLO0_ADDR)
        entrylo0_v <= cp0_wdata[1];
    else if(tlbr)
        entrylo0_v <= r_v0;
end

reg entrylo0_g;
always @(posedge clk) begin
    if(mtc0_we && cp0_addr == `CP0_ENTRYLO0_ADDR)
        entrylo0_g <= cp0_wdata[0];
    else if(tlbr)
        entrylo0_g <= r_g;
end

```

```
assign cp0_entrylo0 =
{
    6'b0,               //31:26
    entrylo0_pfn,       //25:6
    entrylo0_c,         //5:3
    entrylo0_d,         //2:2
    entrylo0_v,         //1:1
    entrylo0_g          //0:0
};

```

- EntryLol

```
//ENTRYLO1
reg [19:0] entrylo1_pfn;
always @(posedge clk) begin
    if(mtc0_we && cp0_addr == `CP0_ENTRYLO1_ADDR)
        entrylo1_pfn <= cp0_wdata[25:6];
    else if(tlbr)
        entrylo1_pfn <= r_pfn1;
end

reg [2:0] entrylo1_c;
always @(posedge clk) begin
    if(mtc0_we && cp0_addr == `CP0_ENTRYLO1_ADDR)
        entrylo1_c <= cp0_wdata[5:3];
    else if(tlbr)
        entrylo1_c <= r_c1;
end

reg entrylo1_d;
always @(posedge clk) begin
    if(mtc0_we && cp0_addr == `CP0_ENTRYLO1_ADDR)
        entrylo1_d <= cp0_wdata[2];
    else if(tlbr)
        entrylo1_d <= r_d1;
end

reg entrylo1_v;
always @(posedge clk) begin
    if(mtc0_we && cp0_addr == `CP0_ENTRYLO1_ADDR)
        entrylo1_v <= cp0_wdata[1];
    else if(tlbr)
        entrylo1_v <= r_v1;
end

reg entrylo1_g;
always @(posedge clk) begin
    if(mtc0_we && cp0_addr == `CP0_ENTRYLO1_ADDR)
        entrylo1_g <= cp0_wdata[0];
    else if(tlbr)
        entrylo1_g <= r_g;
end

assign cp0_entrylo1 =
{
```

```
    6'b0,               //31:26
    entrylo1_pfn,       //25:6
    entrylo1_c,         //5:3
    entrylo1_d,         //2:2
    entrylo1_v,         //1:1
    entrylo1_g          //0:0
};
```

- Index

```
//INDEX
reg index_p;
always @(posedge clk) begin
    if(rst)
        index_p <= 1'b0;
    else if (tlbp)
        index_p <= !(s1_found);
end

reg [3:0] index_index;
always @(posedge clk) begin
    if(rst)
        index_index <= 4'b0;
    else if(mtc0_we && cp0_addr == `CP0_INDEX_ADDR)
        index_index <= cp0_wdata[3:0];
    else if(tlbp) begin
        index_index <= s1_index;
    end
end

assign cp0_index =
{
    index_p,        //31:31
    27'b0,          //30:4
    index_index     //3:0
};
```

我们先来看TLBP指令。这个指令需要对TLB进行查找，这意味着它执行时需要利用 TLB模块中的查找逻辑。TLB模块中只有两套查找逻辑,一套用于取指，另一套用于访存， 在不增加逻辑资源的情况下，TLBP指令的执行势必要复用其中一套查找逻辑。而且，我们希望最好能确保复用的时候不阻塞其他指令访问TLB模块，因此复用访存的TLB查找端口并在EX级发起查找请求是最合适的。但是，这时候就面临一个问题: TLBP指令的待查找内容是来自EntryHi寄存器的，而这个寄存器会被写回级的MTC0指令修改。如果TLBP指令在EX级的时候，有一条修改EntryHi的MTC0指令恰好在MEM级，那么直接用CP0寄存器模块中EntryHi寄存器的值就会出现问题。这种情况没有什么好的解决办法，要么采用阻塞，要么采用前递。我们认为,TLBP指令的执行频率非常低，而前面提到的冲突情况的出现概率就更低了，因此设计一套前递的逻辑来处理这种冲突的投人产出比太低，我们还是选择用阻塞的方式

在译码阶段，我们增加TLBP的实现：（具体的添加方式与之前添加新指令的步骤基本一致）

```
wire        inst_tlbp;
assign inst_tlbp    = op_d[6'h10] & (ds_inst[25] == 1'b1) & (ds_inst[24:6] == 19'b0) & func_d[6'h08];
```

再来看 **TLBWI 指令**。由于 TLB 模块设计了一组独立的写入接口，因此 TLBWI 指令何时写入 TLB，与读取该指令源操作数的时间有关。考虑到 EntryHi、EntryLo0 和 EntryLo1 寄存器会被写回级的 MTC0 指令更新，让 TLBWI 指令在写回级写入 TLB 是合适的，因为此时相关 CP0 寄存器的值都是正确的。然而，实现 TLBWI 指令的复杂性在于其他方面。由于所有指令的取指和访存指令的访存操作可能需要读取 TLB 的内容用于虚实地址转换，而 TLBWI 指令会更新 TLB 的内容，这两者之间形成了围绕 TLB 的写后读相关。

既然 TLBWI 的写入发生在写回级，那么这种写后读相关会引发冲突。如何解决冲突呢？有些读者可能会提出阻塞 TLBWI 后续指令的思路，但这种方法无法彻底解决问题。因为在 ID 级才能知道一条指令是否为 TLBWI 指令，而此时 IF 级可能已经根据当前 TLB 的虚实映射关系取回了指令。这种情况下，这条指令需要被取消并重取。既然取消操作不可避免，那么可以单纯采用取消机制来解决冲突。一个可行的方案是：只要发现流水线中存在 TLBWI 指令，就为该指令后的所有指令设置一个重取标志。带有重取标志的指令如同被标记了异常一般，不会产生任何执行效果，同时阻止其后的指令产生执行效果。当带有重取标志的指令到达写回级后，将像处理异常那样清空流水线。然而，它并非真正的异常，因此不会修改任何 CP0 寄存器，也不会提升处理器的特权等级。此外，pre-IF 级更新的 nextPC 应为带有重取标志指令的 PC，而非异常的入口地址。

在译码阶段，我们增加TLBWI的实现：（具体的添加方式与之前添加新指令的步骤基本一致）

```
wire        inst_tlbwi;
assign inst_tlbwi    = op_d[6'h10] & (ds_inst[25] == 1'b1) & (ds_inst[24:6] ==
19'b0) & func_d[6'h02];
```

再来看 **TLBR 指令**。由于 TLB 模块设计了一组独立的读出接口，因此 TLBR 指令的读操作时间不受限制。重点需要考虑的是与 TLBR 指令相关的 CP0 寄存器可能对其执行产生的影响。首先，TLBR 指令需要读取 Index 寄存器作为读地址，而 Index 寄存器可能被写回级的 MTC0 指令更新。其次，TLBR 指令需要更新 EntryHi、EntryLo0 和 EntryLo1 寄存器，而这些寄存器可能被写回级的 MFC0 指令读取。因此，TLBR 指令在写回级读取 TLB，然后更新相关 CP0 寄存器是最合适的。

然而，工作并未结束。由于 TLBR 指令会更新 EntryHi 寄存器的 ASID 域，而所有指令的取指和访存操作可能需要读取 EntryHi 的 ASID 域以查找 TLB，因此 TLBR 指令需要像 TLBWI 指令一样解决 CP0 冲突问题。解决方法也相同，采用为后续指令标记重取标志的方式来解决冲突。

在译码阶段，我们增加TLBR的实现：（具体的添加方式与之前添加新指令的步骤基本一致）

```
wire        inst_tlbr;
assign inst_tlbr     = op_d[6'h10] & (ds_inst[25] == 1'b1) & (ds_inst[24:6] ==
19'b0) & func_d[6'h01];
```

```
wire other_inst;
assign other_inst = !(inst_addu | inst_subu | inst_slt | inst_sltu | inst_and
| inst_or | inst_xor | inst_nor
| inst_sll | inst_srl | inst_sra | inst_addiu | inst_lui | inst_lw | inst_sw
| inst_beq | inst_bne | inst_jal
| inst_jr | inst_add | inst_addi | inst_sub | inst_slti | inst_sltiu |
inst_andi | inst_ori | inst_xori | inst_sllv
| inst_srlv | inst_srav | inst_mult | inst_multu | inst_div | inst_divu |
inst_mfhi | inst_mflo | inst_mthi | inst_mtlo
| inst_bgez | inst_bgtz | inst_blez | inst_bltz | inst_j | inst_bltzal |
inst_bgezal | inst_jalr | inst_lb | inst_lbu
| inst_lh | inst_lhu | inst_lwl | inst_lwr | inst_sb | inst_sh | inst_swl |
inst_swr | inst_syscall | inst_eret | inst_mfc0
| inst_mtc0 | inst_break | inst_tlbp | inst_tlbr | inst_tlbwi);
```

4.3 TLB相关的例外实现

如何报出例外？与已实现的非TLB相关例外一样：例外信息沿流水线逐级传递，当指令到达写回级的时候报出例外，修改相关的CP0寄存器（域）同时跳转到例外入口地址。具体实现如下：

```
wire interrupt;

assign interrupt = ((cp0_cause[15:8] & cp0_status[15:8]) != 8'b0) &&
(cp0_status[1:0] == 2'b01);


assign ds_ex = ds_valid && (
    fs_to_ds_ex || inst_syscall || inst_break || other_inst || interrupt ||
    ds_after_tlb || inst_eret
);
assign ds_excode =
    interrupt       ? `EX_INT :
    fs_to_ds_ex     ? fs_to_ds_excode :
    other_inst      ? `EX_RI :
    inst_syscall    ? `EX_SYS :
    inst_break      ? `EX_BP :
    `EX_NO;
assign ds_badvaddr = fs_to_ds_badvaddr;

//TLB例外
reg  ds_after_tlb_r;
assign ds_after_tlb = ds_after_tlb_r;

always @ (posedge clk) begin
    if (reset) begin
        ds_after_tlb_r <= 1'b0;
    end else if (do_flush) begin
        ds_after_tlb_r <= 1'b0;
    end else if (affect_tlb && es_allowin && ds_to_es_valid) begin
        ds_after_tlb_r <= 1'b1;
    end
end
```

至此，TLB模块的集成工作已经完成了。

5. Cache

我们的最后一站是：Cache！作为最有难度的一集，由于截止时间将近且任务困难复杂，这里只做简单讨论，不再尝试实现。

根据《CPU设计实战》书中所言，Cache的添加可以主要分为四个阶段：

1. 阶段一：设计Cache模块。
2. 阶段二：将Cache模块作为ICaChe（指令Cache）集成到CPU中，完成与CPU取指的配合、调整，并完成总线接口模块的设计调整
3. 阶段三：将CaChe模块作为DCache（指令Cache）集成到CPU中，完成与CPU访存的配合、调整，并完成总线接口模块的设计调整。
4. 阶段四：实现对Cache指令的支持。

五、实验总结

呼，讲真的不开玩笑，本次大作业可以算是最有难度的一次，尤其是在大三下期末复习压力的加成下，简直是地狱难度，“龙芯杯”的部分内容被拿来作为课程大作业，打比赛的同学们需要花费数月时间才能完成的任务，想要在期末被压缩的时间中部分实现似乎不太现实。好在老师的评分要求应该是在鼓励尝试（~~写了就有分~~）。

借着《CPU设计实战》这本书的指导，以及github上或多或少的关于“龙芯杯”的源码，本人也算是系统的了解并学习了一遍从中断和例外到TLB设计的知识。但是众所周知，项目是debug出来的，不是敲出来就行了，即便完成了大部分的步骤要求，但是面对如此复杂的项目，想要成功运行每一模块还是太有难度了。加之计网、OS等等课程压力，耗费巨量时间在Debug上也不现实。故本报告的提高部分旨在尽量完整的完成必要步骤与整体方向，至于跑通代码……（笑），还是能力不足，害。当然本人也尝试了很久的debug，只不过bug是一个接着一个跳出来，属实无能为力了。

当然啦，这次实验还是收获颇丰的，不仅仅是我对于计算机硬件功能的动手能力以及相关知识，我觉得更多的是视野的拓展，我们当然知道计算机是一个无比精妙的仪器，并且知道这是人类做出来的，但是当我们亲身其感受如何具体的设计一个CPU时，还是被震撼到了。另外，我也看到了很多关于龙芯杯的信息，我们南开也有三等奖（但是我为什么找不到他们的源码?!），不禁感叹，打比赛真不容易啊~

好了，瞎感叹了一堆。。。