Prj 4 报告

组员：
武弋 2022K8009929002
王泽黎 2022K8009929011
张家玮 2022K8009929010
箱子号：16

一、实验任务

Exp12：

1. 为 CPU 增加 csrrd、csrwr、csrxchg 和 ertn 指令。

2. 为 CPU 增加控制状态寄存器 CRMD、PRMD、ESTAT、ERA、EENTRY、SAVE0~3。

3. 为 CPU 增加 syscall 指令，实现系统调用异常支持。

4. 运行 exp12 对应的 func，要求成功通过仿真和上板验证。

Exp13：

1. 为 CPU 增加取指地址错（ADEF）、地址非对齐（ALE）、断点（BRK）和指令不存在（INE） 异常的支持。

2. 为 CPU 增加中断的支持，包括 2 个软件中断、8 个硬件中断和定时器中断。

3. 为 CPU 增加控制状态寄存器 ECFG、BADV、TID、TCFG、TVAL、TICLR。

4. 为 CPU 增加 rdcntvl.w、rdcntvh.w 和 rdcntid 指令。

5. 运行 exp13 对应的 func，要求成功通过仿真和上板验证。

二、实验设计

（一）Exp12 设计：

1. 为 CPU 增加 csrrd、csrwr、csrxchg 和 ertn 指令。

csrrd、csrwr、csrxchg 指令用于控制状态寄存器的读写，ertn 指令用于从异常或中断处理程序返回。

添加上述四条新指令，首先要参照指令集手册，对指令码译码得到这些指令的控制信号，代码如下：

```
wire        inst_csrrd;
wire        inst_csrwr;
wire        inst_csrxchg;
wire        inst_ertn;
```

得到译码信号之后，要用这些信号来组成数据选择信号，控制数据通路，以读寄存器堆选择信号 2 为例，代码如下：

```
assign rf_using2 = inst_beq
                || inst_bne
                || inst_blt
                || inst_bge
                || inst_bltu
                || inst_bgeu // exp11: add branch instructions
                || inst_jirl // branch instructions without b/bl
                // || inst_ld_w
                // || inst_st_w // load/store instructions
                || inst_add_w
                || inst_sub_w
                || inst_slt
                || inst_sltu
                || inst_nor
                || inst_and
                || inst_or
                || inst_xor // ALU without imm instructions
                || mul_inst // multiplication instructions
                || div_inst // division instructions
                || inst_sll_w
                || inst_srl_w
                || inst_sra_w // shift instructions without imm instructions
                // || inst_slli_w
                // || inst_srli_w
                // || inst_srai_w
                // || inst_addi_w // imm instructions
                || inst_csrxchg
                || inst_csrrd
                || inst_csrwr
;
```

其给出了需要使用寄存器堆读出数据 2 的情况，其他控制信号的组合逻辑与此类似，都是根据具体指令的实现方法，用译码信号通过或逻辑得到。

三条 CSR 读写指令用于在通用寄存器和 CSR 寄存器之间交互数据，其在 WB 阶段代码如下：

```
assign final_result = res_from_mem_WB ? mem_result_WB :
                      mul_inst_WB     ? mul_result_WB :
                      res_from_csr_WB ? csr_value_WB :
                      alu_result_WB;
assign csr_waddr = csr_dest_WB;
assign csr_wdata = rkd_value_WB;
assign csr_we    = {32{inst_csrwr_WB}} | {32{inst_csrxchg_WB}} & rj_value_WB;
assign csr_wbex  = ex_WB;
```

ERTN 指令一方面将 ERA 寄存器中存放的指针值作为目标地址跳转过去，同时将 PRMD 控制状态寄存器中的 PPLV 和 PIE 域的值分别写入到 CRMD 控制状态寄存器的 PLV 和 IE 域，将 ERTN 译码信号接入 CSR 寄存器代码

如下：

```
assign csr_raddr = inst_ertn ? 14'b110 :
                   inst_syscall ? 14'b1100 : imm[13:0];
assign csr_ertn  = inst_ertn;
```

2. 为 CPU 增加控制状态寄存器 CRMD、PRMD、ESTAT、ERA、EENTRY、SAVE0~3。

本次任务所需的控制状态寄存器有 CRMD、PRMD、ESTAT、ERA、EENTRY、SAVE0~3，根据 LoongArch 指令手册第 7 章控制状态寄存器相关内容进行实现，控制状态寄存器模块代码如下：

```
module csr_regfile(
    input  wire        clk,
    input  wire        rst,
    // READ PORT 1
    input  wire [13:0] raddr,
    output wire [31:0] rdata,
    // WRITE PORT
    input  wire [31:0] we,       //write enable, HIGH valid
    input  wire [13:0] waddr,
    input  wire [31:0] wdata,
    input  wire        ertn,
    input  wire        wb_ex,
    input  wire [31:0] wb_pc,
    input  wire [ 5:0] ecode //例外号
);
`define CSR_CRMD waddr==14'b0
`define CSR_PRMD waddr==14'b1
`define CSR_ESTATUS waddr==14'b101
`define CSR_ERA waddr==14'b110
`define CSR_EENTRY waddr==14'b1100
`define CSR_SAVE (waddr==14'b110000 | waddr==14'b110001 | waddr==14'b110010 |
waddr==14'b110011)
reg [31:0] csr[51:0];

wire [31:0] crmd;
wire [31:0] prmd;
wire [31:0] estat;
wire [31:0] era;
wire [31:0] eentry;
wire [31:0] save0;
wire [31:0] save1;
wire [31:0] save2;
wire [31:0] save3;

assign crmd = csr[0];
assign prmd = csr[1];
```

```
assign estat = csr[5];
assign era = csr[6];
assign eentry = csr[12];
assign save0 = csr[48];
assign save1 = csr[49];
assign save2 = csr[50];
assign save3 = csr[51];

wire csr_find_w;
wire csr_find_r;

//WRITE
always @(posedge clk) begin
    if (rst) begin
        csr[0] <= 32'b1000;
        csr[5] <= 32'b0;
    end
    else if (wb_ex) begin
        csr[1][2:0] <= csr[0][2:0];
        csr[0][2:0] <= 3'b0;
        csr[5][21:16] <= ecode;
        csr[6] <= wb_pc;
    end
    else if (ertn) begin
        csr[0][2:0] <= csr[1][2:0];
    end
    else if (csr_find_w) begin
        csr[waddr] <= ~we & csr[waddr] | we & wdata;
    end
end

assign csr_find_w = |we & ((waddr == 14'b0) |
                    (waddr == 14'b1) |
                    (waddr == 14'b101) |
                    (waddr == 14'b110) |
                    (waddr == 14'b1100) |
                    (waddr == 14'b110000) |
                    (waddr == 14'b110001) |
                    (waddr == 14'b110010) |
                    (waddr == 14'b110011));

assign csr_find_r = (raddr == 14'b0) |
                    (raddr == 14'b1) |
                    (raddr == 14'b101) |
                    (raddr == 14'b110) |
```

```
                    (raddr == 14'b1100) |
                    (raddr == 14'b110000) |
                    (raddr == 14'b110001) |
                    (raddr == 14'b110010) |
                    (raddr == 14'b110011);

//READ OUT 1
assign rdata = (~csr_find_r) ? 32'b0 : csr[ertn ? 32'b110 :raddr];

endmodule
```

3. 为 CPU 增加 syscall 指令，实现系统调用异常支持。

执行 SYSCALL 指令将立即无条件的触发系统调用例外。

添加 syscall 指令，首先要参照指令集手册，对指令码译码得到这些指令的控制信号，代码如下：

```
assign inst_syscall    = (inst_ID[31:15] == 17'b1010110);
```

syscall 相关控制信号译码，代码如下：

```
assign exception = inst_syscall;
assign csr_dest      = inst_syscall ? 14'h6 : imm[13:0];
```

syscall 指令实现功能需要将译码信号接入控制状态寄存器模块，代码如下：

```
assign csr_raddr = inst_ertn ? 14'b110 :
                   inst_syscall ? 14'b1100 : imm[13:0];
assign csr_ecode = inst_syscall ? 6'hb : 6'h0;
```

（二）Exp13 设计：

1. 为 CPU 增加取指地址错（ADEF）、地址非对齐（ALE）、断点（BRK）和指令不存在（INE） 异常的支持。

这四类异常的功能实现流程主要参考教材与指令手册，以取值地址错（ADEF）为例。

LoongArch 指令系统要求所有的指令的 PC 都是字对齐（地址最低两比特为全 0），当违反这一规定时，将触发取指地址错误异常。因此加入必要的控制信号和异常条件判断，代码如下：

```
wire pc_unalign;
assign pc_unalign = pc[1:0] != 2'b00;
assign ex_IF = pc_unalign;
assign csr_ecode = pc_unalign ? 6'h8 : 6'h0;
```

进而在后续过程中，错误的 PC 值将被硬件记录在 BADV 控制状态寄存器。

2. 为 CPU 增加中断的支持，包括 2 个软件中断、8 个硬件中断和定时器中断。

硬件中断的源头来自于处理器核外部，LoongArch 处理器核内部 ESTAT 控制状态寄存器 IS（Interrupt State）域的 9..2 这八位对 8 个中断输入引脚接入的中断信号进行采样。软件中断顾名思义是由软件来设置的，通过 CSR 写指令对 ESTAT 状态控制寄存器 IS 域的 1..0 这两位写 1 或写 0 就可以完成两个软件中断的置起和撤销。定时器

中断经常用于操作系统的调度和计时功能的实现。该中断源来自于定时器，通常分为核外和核内两种实现方式。LoongArch 指令系统采用了核内实现定时器中断源的方式。

接下来我们主要分析定时器中断的实现。

首先在控制状态寄存器模块中加入新信号，代码如下：

```
reg  [31:0] timer_cnt;
```

在开启定时功能后每个时钟周期减 1，当减到 0 值即可触发一次定时器中断，代码如下：

```
always @(posedge clk) begin
    if (reset)
        timer_cnt <= 32'hffffffff;
    else if (csr_we && csr_waddr==`CSR_TCFG && tcfg_next_value[`CSR_TCFG_EN])
        timer_cnt <= {tcfg_next_value[`CSR_TCFG_INITV], 2'b0};
    else if (csr_tcfg_en && timer_cnt!=32'hffffffff) begin
        if (timer_cnt[31:0]==32'b0 && csr_tcfg_periodic)
            timer_cnt <= {csr_tcfg_initval, 2'b0};
        else
            timer_cnt <= timer_cnt - 1'b1;
    end
end
```

当定时器倒 计时到 0 时硬件将 ESTAT 控制状态寄存器 IS 域的第 11 位置 1，软件通过对 TICLR 控制寄存器的 CLR 位写 1 将 ESTAT 控制状态寄存器 IS 域的第 11 位清 0，代码如下：

```
    if (timer_cnt[31:0]==32'b0)
        csr_estat_is[11] <= 1'b1;
    else if (csr_we && csr_waddr==`CSR_TICLR && csr_wmask[`CSR_TICLR_CLR]
    && csr_wdata[`CSR_TICLR_CLR])
        csr_estat_is[11] <= 1'b0;
```

has_int 识别到这个变化，拉高，传到 WB 阶段之后触发中断，代码如下：

```
assign has_int = ((csr_estat_is[12:0] & csr_ecfg_lie[12:0]) != 13'b0)
             && (csr_crmd_ie == 1'b1);
```

```
always @(posedge clk) begin
    if (reset) begin
        csr_wbex_rst <= 1'b1;
    end
    else if (ex_WB || has_int_WB) begin
        csr_wbex_rst <= 1'b0;
    end
    else if (~ex_WB && ~has_int_WB) begin
        csr_wbex_rst <= 1'b1;
    end
end
```

进而通过 wb_ex 信号重置控制状态寄存器中 estat 的第 11 位，中断结束，代码如下：

```
assign csr_wbex  = (ex_WB || has_int_WB) && csr_wbex_rst;
```

```
always @(posedge clk) begin
    if (wb_ex) begin
        csr_estat_ecode <= wb_ecode;
        csr_estat_esubcode <= wb_esubcode;
    end
end
```

3. 为 CPU 增加控制状态寄存器 ECFG、BADV、TID、TCFG、TVAL、TICLR。

主要在控制状态寄存器模块添加新寄存器，以 ECFG 举例，代码如下：

```
`define CSR_ECFG 14'h4
`define CSR_ECFG_LIE 12:0
reg [12:0] csr_ecfg_lie;
always @(posedge clk) begin
    if (reset)
        csr_ecfg_lie <= 13'b0;
    else if (csr_we && csr_waddr==`CSR_ECFG)
        csr_ecfg_lie <= csr_wmask[`CSR_ECFG_LIE] & 13'h1bff & csr_wdata[`CSR_ECFG_LIE]
                      | ~csr_wmask[`CSR_ECFG_LIE] & 13'h1bff & csr_ecfg_lie;
end
wire [31:0] csr_ecfg_value = {19'b0, csr_ecfg_lie};
assign csr_rdata = {32{csr_raddr==`CSR_CRMD}} & csr_crmd_value
                 | {32{csr_raddr==`CSR_PRMD}} & csr_prmd_value
                 | {32{csr_raddr==`CSR_ECFG}} & csr_ecfg_value
                 | {32{csr_raddr==`CSR_ESTAT}} & csr_estat_value
                 | {32{csr_raddr==`CSR_ERA}} & csr_era_value
                 | {32{csr_raddr==`CSR_BADV}} & csr_badv_value
                 | {32{csr_raddr==`CSR_EENTRY}} & csr_eentry_value
                 | {32{csr_raddr==`CSR_SAVE0}} & csr_save0_value
                 | {32{csr_raddr==`CSR_SAVE1}} & csr_save1_value
                 | {32{csr_raddr==`CSR_SAVE2}} & csr_save2_value
                 | {32{csr_raddr==`CSR_SAVE3}} & csr_save3_value
                 | {32{csr_raddr==`CSR_TID}} & csr_tid_value
                 | {32{csr_raddr==`CSR_TCFG}} & csr_tcfg_value
                 | {32{csr_raddr==`CSR_TVAL}} & csr_tval_value
                 | {32{csr_raddr==`CSR_TICLR}} & csr_ticlr_value;
assign has_int = ((csr_estat_is[12:0] & csr_ecfg_lie[12:0]) != 13'b0)
              && (csr_crmd_ie == 1'b1);
```

4. 为 CPU 增加 rdcntvl.w、rdcntvh.w 和 rdcntid 指令。

rdcntvl.w、rdcntvh.w 和 rdcntid 是三条计时器相关指令，以 rdcntvl.w 实现为例，代码如下：

首先加入 64 位宽计时器

```
reg [63:0] cnt;
always @(posedge clk) begin
    if (reset) begin
        cnt <= 64'h0;
    end
    else begin
        cnt <= cnt + 1;
    end
end
```

加入指令译码信号

```
wire        inst_rdcntvl_w;
assign inst_rdcntvl_w = (inst_ID[31:15] == 17'b0 && inst_ID[14:10] == 5'b11000 &&
inst_ID[9:5] == 5'b0);
```

修改相关数据通路，以最终写入结果 final_result 为例

```
assign final_result = res_from_mem_WB ? mem_result_WB :
                      mul_inst_WB     ? mul_result_WB :
                      res_from_csr_WB ? csr_value_WB :
                      inst_rdcntid_w_WB ? csr_value :
                      inst_rdcntvh_w_WB || inst_rdcntvl_w_WB ? cnt_result_WB :
                      alu_result_WB;
```

实现读取功能

```
assign cnt_result = inst_rdcntvl_w_EX ? cnt[31:0] : cnt[63:32];
```

三、实验过程

（一）错误记录

1、**错误 1**：Exp12：分支跳转类指令后接除法类指令导致阻塞异常

（1）错误现象

当一条分支跳转类指令后接除法类指令时，处理器将永久阻塞。

（2）分析定位过程

通过观察流水线控制信号波形，发现是由于保证除法器 IP 核完成除法所用的控制信号导致处理器阻塞异常。

（3）错误原因

在添加除法类型指令时，忽略了除法器 IP 核需要多周期完成除法指令，进而会因分支跳转指令导致除法无法正常完成，因此引发永久阻塞。

（4）修正效果

在除法指令信号传入执行阶段的时序逻辑中，加入分支跳转指令信号进行控制，进而保证当下一条指令正常分支跳转时不会启动除法 IP 核。

2、**错误 2**：Exp13：CSR 寄存器写入异常

（1）错误现象

某些 CSR 寄存器被写入多次。

（2）错误原因

csr_wbex/flush 信号可能会持续一个周期以上，然后导致某些 CSR 寄存器被写入不止一次。

（3）修正效果

加入 csr_wbex_rst/flush_rst 信号，重置 csr_wbex/flush 信号，保证不会多周期重复写入。

3、错误 3：Exp13：指令冲突

（1）错误现象

加入计时器相关指令后出现指令冲突。

（2）错误原因

由于指令 rdcntid.w 在 WB 阶段取值，因此可能与其他指令发生冲突，引发错误。

（3）修正效果

设置当指令 rdcntid.w 与其他指令冲突时，阻塞流水线三个周期，以确保 rdcntid.w 之后的指令能够从寄存器文件中获取正确的值。

四、实验总结

本次实验的主要内容是添加对异常和中断的认识，这是此前计算机组成原理研讨课未接触的部分。通过本次实验的学习与讨论，小组同学在顺利完成实验的同时，也联系理论课内容，加深了对异常和中断的理解。此外，经过本次实验，小组成员合作交流的经验也更加丰富，通过合理运用 Git 等工具，提高了工作效率。

五、实验分工

exp12 代码编写： 武弋

exp13 代码编写：张家玮

prj4 实验报告撰写：王泽黎

除完成各自的任务外，小组成员之间在遇到问题时相互讨论，共同解决了实验中遇到的问题。