计 算 机 体 系 结 构 （研 讨 课） 实 验 报 告

实验项目 prj4 小组编号 28 组员姓名 刘景平、张钰堃、付博宇

1 逻辑电路结构与仿真波形的截图及说明

1.1. exp12

1.1.1. 增加 CSR 模块

1. 需要为 CSR 模块单独写一个 verilog 文件，作为专用于 CSR 的寄存器堆，按照教材上的指导加入一系列输入输出信号, 并在头文件中加入每个 CSR 的寄存器号，每个 CSR 各部分的位置等信息：

1. 把所有的控制状态寄存器集中到一个模块中实现；
2. 模块接口分为用于指令访问的接口和与处理器核内部硬件电路逻辑直接交互的控制、状态信号接口两类；
3. 指令访问接口包含读使能（csr_re）[7]、寄存器号（csr_num）、寄存器读返回值（csr_rvalue）、写使能（csr_we）、写掩码（csr_wmask）和写数据（csr_wvalue）；
4. 与硬件电路逻辑直接交互的接口信号视需要各自独立定义，无须再统一编码，如送往预取指（pre-IF）流水级的异常处理入口地址 ex_entry、送往译码流水级的中断有效信号 has_int、来自写回流水级的 ertn 指令执行的有效信号 ertn_flush、来自写回流水级的异常处理触发信号 wb_ex 以及异常类型类型 wb_ecode、wb_esubcode 等。

图 1: 讲义上关于创建 CSR 模块的指导

```
`include "mycpu_head.vh"

module csr_reg(
    input                           clk,
    input                           reset,

    input [`WIDTH_CSR_NUM-1:0]       csr_num,           // Register num

    input                           csr_re,            // Read enable
    output              [31:0]      csr_rvalue,        // Read data
    output              [31:0]      ertn_pc,
    output              [31:0]      ex_entry,

    input                           csr_we,            // Write enable
    input               [31:0]      csr_wmask,         // Write mask
    input               [31:0]      csr_wvalue,        // Write data

    input                           wb_ex,             // Write-back le
    input               [31:0]      wb_pc,             // Exception PC
    input                           ertn_flush,        // ERTN instruct
    input               [5:0]       wb_ecode,          // Exception typ
    input               [8:0]       wb_esubcode,       // Exception typ
    input               [31:0]      wb_vaddr,
    input               [31:0]      coreid_in,

    output                          has_int,
    input               [7:0]       hw_int_in,
    input                           ipi_int_in
);
```

图 2: 创建 CSR 模块

```
`define CSR_ERA 14'h6
`define CSR_BADV 14'h7
`define CSR_EENTRY 14'hc
`define CSR_SAVE0 14'h30
`define CSR_SAVE1 14'h31
`define CSR_SAVE2 14'h32
`define CSR_SAVE3 14'h33
`define CSR_TID 14'h40
`define CSR_TCFG 14'h41
`define CSR_TVAL 14'h42
`define CSR_TICLR 14'h44


//CSR partition

//CSR_CRMD
`define CSR_CRMD_PLV 1:0
`define CSR_CRMD_IE 2:2
`define CSR_CRMD_DA 3:3
`define CSR_CRMD_PG 4:4
`define CSR_CRMD_DATF 6:5
`define CSR_CRMD_DATM 8:7
`define CSR_CRMD_ZERO 31:9

//CSR_PRMD
`define CSR_PRMD_PPLV 1:0
`define CSR_PRMD_PIE 2:2
`define CSR_PRMD_ZERO 31:3
```

图 3: 修改头文件

2. 根据讲义上的例子添加 CRMD、PRMD、ESTAT、ERA、EENTRY、SAVE0 3 八个控制状态寄存器。并按照指令集手册的要求添加关于 read value 的设置

```
reg [1:0] csr_prmd_pplv;
reg csr_prmd_pie;

always @(posedge clk)
    begin
        if(wb_ex)
            begin
                csr_prmd_pplv <= csr_crmd_plv;
                csr_prmd_pie  <= csr_crmd_ie;
            end
        else if(csr_we && csr_num == `CSR_PRMD)
            begin
                csr_prmd_pplv <= csr_wmask[`CSR_PRMD_PPLV] & csr_wvalue[`CSR_PRMD_PPLV]
                               | ~csr_wmask[`CSR_PRMD_PPLV] & csr_prmd_pplv;
                csr_prmd_pie  <= csr_wmask[`CSR_PRMD_PIE] & csr_wvalue[`CSR_PRMD_PIE]
                               | ~csr_wmask[`CSR_PRMD_PIE] & csr_prmd_pie;
            end
    end
```

图 4: PRMD 寄存器的设置

```
assign csr_tid_rvalue = csr_tid_tid;
assign csr_tcfg_rvalue = {csr_tcfg_initval, csr_tcfg_periodic, csr_tcfg_en};
assign csr_tval_rvalue = csr_tval;

assign csr_rvalue = {32{csr_num==`CSR_CRMD}} & csr_crmd_rvalue
                  | {32{csr_num==`CSR_PRMD}} & csr_prmd_rvalue
                  | {32{csr_num==`CSR_ECFG}} & csr_ecfg_rvalue
                  | {32{csr_num==`CSR_ESTAT}} & csr_estat_rvalue
                  | {32{csr_num==`CSR_ERA}} & csr_era_rvalue
                  | {32{csr_num==`CSR_BADV}} & csr_badv_rvalue
                  | {32{csr_num==`CSR_EENTRY}} & csr_eentry_rvalue
                  | {32{csr_num==`CSR_SAVE0}} & csr_save0_rvalue
                  | {32{csr_num==`CSR_SAVE1}} & csr_save1_rvalue
                  | {32{csr_num==`CSR_SAVE2}} & csr_save2_rvalue
                  | {32{csr_num==`CSR_SAVE3}} & csr_save3_rvalue
                  | {32{csr_num==`CSR_TID}} & csr_tid_rvalue
                  | {32{csr_num==`CSR_TCFG}} & csr_tcfg_rvalue
                  | {32{csr_num==`CSR_TVAL}} & csr_tval_rvalue;
```

图 5: 设置 rvalue

1.1.2. 修改五个流水线模块

1. 在 IF 中对 nextpc 进行修改，加入出现中断和例外，以及异常处理返回的情况

```
wire [31:0] next_pc; //nextpc from branch or sequence
assign next_pc = (has_int || wb_ex)? ex_entry : ertn_flush? ertn_pc :
    br_taken? br_target : seq_pc;
```

2. 在 ID 模块中加入对 CSR 读写指令有关的译码信号并增加和 CSR 有关的阻塞;

```
wire csr_crush;
assign csr_crush = (es_csr && (ex_crush1 || ex_crush2)) || (ms_csr &&
    (mem_crush1 || mem_crush2));
```

```
//inst[23:10] -- csr_num
wire [13:0] ds_csr_num;
assign ds_csr_num = inst[23:10];

wire ds_ex_syscall;
assign ds_ex_syscall = inst_syscall;

wire [14:0] ds_code;
assign ds_code = inst[14:0];

wire ds_csr;
assign ds_csr = inst_csrrd | inst_csrwr | inst_csrxchg;

wire ds_csr_write;
assign ds_csr_write = inst_csrwr || inst_csrxchg;

wire [31:0] ds_csr_wmask;
assign ds_csr_wmask = inst_csrxchg ? rj_value : 32'hffffffff;

wire ds_ertn_flush;
assign ds_ertn_flush = inst_ertn;
```

图 6: csr 读写译码

3. 在遇到 ertn 指令或者遇到中断和例外时，需要情况流水级间缓存：

```
reg [`WIDTH_DS_TO_ES_BUS-1:0] ds_to_es_bus_reg;
always @(posedge clk)
    begin
        if(reset)
            ds_to_es_bus_reg <= 0;
        else if(ertn_flush | has_int | wb_ex)
            ds_to_es_bus_reg <= 0;
        else if(ds_to_es_valid && es_allow_in)
            ds_to_es_bus_reg <= ds_to_es_bus;
        else if(ds_to_es_valid)
            ds_to_es_bus_reg <= ds_to_es_bus_reg;
            else
            ds_to_es_bus_reg <= 0;
    end
```

图 7: 清空流水级间缓存

4. 在最后一级 WB 级进行中断与异常的判断：

```
/*--------------------------link csr_reg--------------------*/
assign csr_num = ws_csr_num;
assign csr_re = 1'b1;
//input [31:0] csr_rvalue

assign csr_we = ws_csr_write;
assign csr_wvalue = ws_csr_wvalue;
assign csr_wmask = ws_csr_wmask;
assign ertn_flush = ws_ertn_flush;

assign wb_ex = ws_ex_syscall;    // assign wb_ex = ws_ex_syscall || ws_ex_xxx ||
assign wb_pc = ws_pc;

/*
 *deal with ecode and esubcode according to kind of ex
 *in task12, we just finish syscall
 */
assign wb_ecode = ws_ex_syscall ? 6'hb : 6'h0;          //syscall
assign wb_esubcode = ws_ex_syscall ? 9'h0 : 9'h0;       //syscall
```

图 8: 在最后一级判断中断与异常

1.2. exp13

1.2.1. 添加转移指令

1. 添加转移指令首先需要增加有关指令类型的译码信号，按照之前的格式，写明该指令的汇编代码和具体操作，以 beq 和 bne 指令为例：

2. 需要根据指令类型译码信号更改其他相关的译码信号，如 gr-we、br-target 等，剩余的指令通路与之前的转移指令相同。

3. 为了同时处理有符号数和无符号数的比较，需要利用加法进行判断，在 ID 模块中加入了一个小加法器。

```
    //imitation calcu slt and sltu in alu
    wire signed_rj_less_rkd;
    wire unsigned_rj_less_rkd;

    wire cin;
    assign cin = 1'b1;
    wire [31:0] adver_rkd_value;
    assign adver_rkd_value = ~rkd_value;
    wire [31:0] rj_rkd_adder_result;
    wire cout;
    assign {cout, rj_rkd_adder_result} = rj_value + adver_rkd_value + cin;

    assign signed_rj_less_rkd = (rj_value[31] & ~rkd_value[31])
                              | ((rj_value[31] ~^ rkd_value[31]) &
                                 rj_rkd_adder_result[31]);
    assign unsigned_rj_less_rkd = ~cout;
```

首先将 rd 中的值取反，然后将 rj 与 rd 相加，并加入一个进位 1，计算出一个 33 位的加法结果，其中最高位 cout 为进位。对于无符号数的比较而言，如果没有进位则说明 rj 小于 rd，即 unsigned_rj_less_rkd 为 1。对于有符号数的比较而言，如果 rj 的符号为 1，而 rd 的符号位为 0，则可直接说明 rj 小于 rd，即 signed_rj_less_rkd 为 1；如果 rj 和 rd 的符号位相同，则需要比较 rj 与 rd 的加法结果的符号位，如果为 1 则说明 rj 小于 rd，即 signed_rj_less_rkd 为 1。

1.2.2. 添加 st 指令

在 LoongArch 指令集中 st.b、st.h、st.w 分别对应 store byte（1 字节）/halfword（2 字节）/word（4 字节），其中.h 和.w 分别要求地址 2 字节和 4 字节对齐。

1. 添加转移指令首先需要增加有关指令类型的译码信号, 不再次举例

2.st.b 的四种偏移 00、01、10、11 分别对应 mem-we 为 0001、0010、0100、1000。st.h 的两种偏移 00、10 分别对应 mem-we 为 0011、1100。如此对 mem-we 进行赋值。32 位的 wdata 将 8 位数据重复 4 次，16 位数据重复 2 次填满即可

3. 在 EX 中对未对齐的地址进行对齐操作，定义了一个写使能信号 w-strb 和真实写数据 real-wdata，根据 st-op 的不同情况进行选择

```
    wire [3:0] w_strb; //depend on st_op
    assign w_strb = es_st_op[0] ? 4'b1111 :
                    es_st_op[1] ? (es_unaligned_addr==2'b00 ? 4'b0001 :
                       es_unaligned_addr==2'b01 ? 4'b0010 :
                              es_unaligned_addr==2'b10 ? 4'b0100 : 4'b1000)
```

```
                                    :
                    es_st_op[2] ? (es_unaligned_addr[1] ? 4'b1100 : 4'b0011) :
                       4'b0000;
    wire [31:0] real_wdata;
    assign real_wdata = es_st_op[0] ? es_rkd_value :
                        es_st_op[1] ? {4{es_rkd_value[7:0]}} :
                        es_st_op[2] ? {2{es_rkd_value[15:0]}} : 32'b0;
```

4. 对 SRAM 相关信号进行修改

```
    assign data_sram_en = 1'b1;
    assign data_sram_wen = (es_mem_we && es_valid) ? w_strb : 4'b0000;
    assign data_sram_addr = (es_mul_op != 0) ? {es_mul_result[31:2],2'b00} :
       {es_alu_result[31:2],2'b00};
    assign data_sram_wdata = real_wdata;
```

1.2.3. 添加 ld 指令

1. 添加转移指令首先需要增加有关指令类型的译码信号, 与之前添加各项指令译码。

```
  assign inst_ld_b = op_31_26_d[6'h0a] & op_25_22_d[4'h0];
  assign inst_ld_h = op_31_26_d[6'h0a] & op_25_22_d[4'h1];
  assign inst_ld_bu = op_31_26_d[6'h0a] & op_25_22_d[4'h8];
  assign inst_ld_hu = op_31_26_d[6'h0a] & op_25_22_d[4'h9];
```

此外添加 3 位 ld op，搭配 res from mem 信号标志当前执行的是哪个 load 指令，其中 0 位表示 load byte，1 位表示 load half word，2 位表示为有符号扩展。

```
    wire [2:0] ld_op;
    assign ld_op[0] = inst_ld_b | inst_ld_bu;//load byte
    assign ld_op[1] = inst_ld_h | inst_ld_hu;//load half word
    assign ld_op[2] = inst_ld_h | inst_ld_b;//is signed
```

大部分译码部分相关信号只用把原先只有 ld.w 信号的部分扩展成各类 ld 指令的信号，不再赘述。

2. 在 EX 级需要注意，由于载入的不再是按字节 load，地址的后两位可能非零，这一数据需要传递到 MEM 级进行处理。此外，ld op 也需要传递到 MEM 级来处理读取的数据。

```
    wire [1:0] es_unaligned_addr;
    assign es_unaligned_addr = (es_mul_op != 0) ? es_mul_result[1:0] :
       es_alu_result[1:0];
    assign es_to_ms_bus[72:71] = es_unaligned_addr;
    assign es_to_ms_bus[75:73] = es_ld_op;
```

3. 在 MEM 级根据 ld op 对读取的数据进行选择和扩充。

```
    wire [31:0] load_b_res,load_h_res;
    assign load_b_res = (unaligned_addr == 2'h0) ?
        {{ms_ld_op[2]?{24{data_sram_rdata[7]}}:24'b0} ,data_sram_rdata[7:0]}
            :(unaligned_addr == 2'h1) ?
                {{ms_ld_op[2]?{24{data_sram_rdata[15]}}:24'b0},data_sram_rdata[15:8]}
            :(unaligned_addr == 2'h2) ?
                {{ms_ld_op[2]?{24{data_sram_rdata[23]}}:24'b0},data_sram_rdata[23:16]}
            :(unaligned_addr == 2'h3) ?
                {{ms_ld_op[2]?{24{data_sram_rdata[31]}}:24'b0},data_sram_rdata[31:24]}
                : 32'b0;
    assign load_h_res = (unaligned_addr[1]) ?
        {{ms_ld_op[2]?{16{data_sram_rdata[31]}}:16'b0}
        ,data_sram_rdata[31:16]}
            :{{ms_ld_op[2]?{16{data_sram_rdata[15]}}:16'b0}
                ,data_sram_rdata[15:0]};
    assign mem_result = ms_ld_op[0] ? load_b_res
            : ms_ld_op[1] ? load_h_res
            : data_sram_rdata;
```

2 实验过程中遇到的问题、对问题的思考过程及解决方法

2.1. exp10

1. 在添加乘法指令时，需要注意传递给乘法器的数据只可能是寄存器传来的数据，所以这里传给乘法器的数据为 es rj value，es rkd value，而非 cal src。

2.2. exp11

1. 在添加有关 ld.b,ld.h 等指令有关扩展至 32 位的逻辑时，发现先判断后复制的逻辑会导致只有最后一位填充，而先扩充成 24 位在判断选择就能正常执行，具体指令如下：

```
    assign load_h_res = (unaligned_addr[1]) ?
        {{24{ms_ld_op[2]?{data_sram_rdata[31]}:0}} ,data_sram_rdata[31:16]}
:{{ms_ld_op[2]?{16{data_sram_rdata[15]}}:16'b0} ,data_sram_rdata[15:0]};
// 0x8XXX -> 0x00018XXX

    assign load_h_res = (unaligned_addr[1]) ?
        {{ms_ld_op[2]?{16{data_sram_rdata[31]}}:16'b0}
        ,data_sram_rdata[31:16]}
:{{ms_ld_op[2]?{16{data_sram_rdata[15]}}:16'b0} ,data_sram_rdata[15:0]};
// 0x8XXX -> 0xffff8XXX
```

猜测是由于前者会把扩展后的 24 位整体当作一位添加在原本的 16 位之前，然后自动填充 15 位 0，导致实验结果与预期不同。修改成后者便能正常运行。

3 实验分工

张钰堃负责完成 exp12，刘景平负责完成 exp13。