

# 计算机体系结构（研讨课）实验报告

实验项目 prj3 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. exp10

#### 1.1.1. 添加算术指令

1. 添加算术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 `slti` 和 `sltiu` 指令为例：

```
//slti rd, rj, si12
//rd = (signed(rj) < SignExtend(si12, 32)) ? 1 : 0
assign inst_slti = op_31_26_d[6'h00] & op_25_22_d[4'h8];
//sltiu rd, rj, si12
//rd = (unsigned(rj) < SignExtend(si12, 32)) ? 1 : 0
assign inst_sltiu = op_31_26_d[6'h00] & op_25_22_d[4'h9];
```

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`、`aluop` 等，剩余的指令通路与之前的算术指令相同。

3. 注意到 `andi`、`ori`、`xori` 指令的立即数扩展模式是零扩展，与之前的算术指令不同，需要单独处理。

```
assign need_ui12 = inst_andi | inst_ori | inst_xori; //立即数扩展模式
assign imm = src2_is_4 ? 32'h4 :
    need_si20 ? {i20[19:0], 12'b0} :
    need_ui5 ? {27'b0, rk[4:0]} :
    need_si12 ? {{20{i12[11]}}, i12[11:0]} :
    need_ui12 ? {20'b0, i12[11:0]} :
    32'b0 ;
```

#### 1.1.2. 添加乘法指令

1. 添加乘法指令首先需要增加有关指令类型的译码信号，思路同算术指令

```
//mul.w rd, rj, rk
//product = signed(GR[rj]) * signed(GR[rk])
//GR[rd] = product[31:0]
assign inst_mul_w = op_31_26_d[6'h00] & op_25_22_d[4'h0] &
    op_21_20_d[2'h1] & op_19_15_d[5'h18];
//mulh.w rd, rj, rk
```

```

//product = signed(GR[rj]) * signed(GR[rk])
//GR[rd] = product[63:32]
assign inst_mulh_w = op_31_26_d[6'h00] & op_25_22_d[4'h0] &
    op_21_20_d[2'h1] & op_19_15_d[5'h19];
//mulh.wu rd, rj, rk
//product = unsigned(GR[rj]) * unsigned(GR[rk])
//GR[rd] = product[63:32]
assign inst_mulh_wu= op_31_26_d[6'h00] & op_25_22_d[4'h0] &
    op_21_20_d[2'h1] & op_19_15_d[5'h1a];

```

## 2. 更改相关译码信号

3. 实现了一个乘法模块 mul，支持三种操作：普通乘法（mul.w）、高位乘法（mulh.w）和无符号高位乘法（mulh.wu）。根据输入的 mul-op 信号，模块决定如何处理输入的两个 32 位源操作数 mul-src1 和 mul-src2。mul-src1 和 mul-src2 在执行高位乘法时会进行符号扩展，其他情况下则进行零扩展

```

// when executing mulh.w instruction, the operands should be
    sign-extended;
// otherwise, the operands should be zero-extended
assign mul_src1_ext = {op_mulh_w & mul_src1[31], mul_src1};
assign mul_src2_ext = {op_mulh_w & mul_src2[31], mul_src2};

```

乘法结果是 66 位的，取低 32 位作为 mull-result，高 32 位作为 mulh-result（这里并没有挑战自己采用电路级实现乘法功能，而是直接调用 IP）

```

// 66-bit signed multiply
wire signed [65:0] mul_res_66;
assign mul_res_66 = $signed(mul_src1_ext) * $signed(mul_src2_ext);
wire [31:0] mull_result;
wire [31:0] mulh_result;
// wire [31:0] mulh_wu_result;
assign mull_result = mul_res_66[31: 0];
assign mulh_result = mul_res_66[63:32];

```

最后通过多路选择器根据操作类型输出最终结果 mul-result

```

assign mul_result = ({32{op_mul_w      }} & mull_result)
    |
    ({32{op_mulh_w | op_mulh_wu}} & mulh_result);

```

### 1.1.3. 添加除法指令

1. 在 ID 级添加相关译码信号，添加 3 位 div op 信号，分别标识是否为需要 div 的指令，是否为 signed 类指令，是 div 指令（还是 mod 指令），并将此信号传输到 EX 级

```

assign inst_div_w = op_31_26_d[6'h00] & op_25_22_d[4'h0] &
    op_21_20_d[2'h2] & op_19_15_d[5'h00];
assign inst_div_wu = op_31_26_d[6'h00] & op_25_22_d[4'h0] &
    op_21_20_d[2'h2] & op_19_15_d[5'h02];
assign inst_mod_w = op_31_26_d[6'h00] & op_25_22_d[4'h0] &
    op_21_20_d[2'h2] & op_19_15_d[5'h01];
assign inst_mod_wu = op_31_26_d[6'h00] & op_25_22_d[4'h0] &
    op_21_20_d[2'h2] & op_19_15_d[5'h03];

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

assign ds_to_es_bus[155:153] = div_op;

```

2. 根据指令要求修改寄存器相关信号，这部分跟 alu 相关计算指令逻辑类似，不在赘述

3. 我选择直接调用 Xilinx IP 来实现除法运算部件。IP 设置跟讲义中所描述的一致，需要注意我例化了两个 IP，分别用来计算符号数除法和无符号数除法，对应设置中 operand sign 的不同选项。

```

div_signed u_div_w(
    .aclk(clk),
    .s_axis_dividend_tdata(es_rj_value),
    .s_axis_dividend_tready(dividend_ready),
    .s_axis_dividend_tvalid(dividend_valid),
    .s_axis_divisor_tdata(es_rkd_value),
    .s_axis_divisor_tready(divisor_ready),
    .s_axis_divisor_tvalid(divisor_valid),
    .m_axis_dout_tdata(div_out),
    .m_axis_dout_tvalid(out_valid)
);
assign {es_div_signed,es_mod_signed} = div_out;

div_unsigned u_div_wu(
    .aclk(clk),
    .s_axis_dividend_tdata(es_rj_value),
    .s_axis_dividend_tready(dividend_u_ready),
    .s_axis_dividend_tvalid(dividend_u_valid),
    .s_axis_divisor_tdata(es_rkd_value),
    .s_axis_divisor_tready(divisor_u_ready),
    .s_axis_divisor_tvalid(divisor_u_valid),
    .m_axis_dout_tdata(div_u_out),

```

```

        .m_axis_dout_tvalid(out_u_valid)
    );
    assign {es_div_unsigned,es_mod_unsigned} = div_u_out;

```

由于这个 IP 采用的是多周期计算，输入输出都需要先握手在传输数据，我选择利用状态机实现多种状态。EXE 对应其他运算以及除法的数据接收（从 ID 级），DIVU WAIT 和 DIV WAIT 分别对应两种 IP 在等待两个输入口的 ready 信号，OUT WAIT 和 UOUT WAIT 对应等待输出口的 valid 信号。当前状态同样会影响 IP 输入口的 valid 信号。

```

always @(*) begin
    case(current_state)
        EXE:
            if(es_div_op[0])begin
                if(es_div_op[1])
                    next_state = DIV_WAIT;
                else
                    next_state = DIVU_WAIT;
            end
        else
            next_state = EXE;
        DIV_WAIT:
            if(dividend_ready & divisor_ready)
                next_state = OUT_WAIT;
            else
                next_state = DIV_WAIT;
        DIVU_WAIT:
            if(dividend_u_ready & divisor_u_ready)
                next_state = UOUT_WAIT;
            else
                next_state = DIVU_WAIT;
        OUT_WAIT:
            if(out_valid)
                next_state = EXE;
            else
                next_state = OUT_WAIT;
        UOUT_WAIT:
            if(out_u_valid)
                next_state = EXE;
            else
                next_state = UOUT_WAIT;
        default:
            next_state = EXE;
    endcase

```

```

end

assign dividend_valid = current_state == DIV_WAIT;
assign divisor_valid = current_state == DIV_WAIT;
assign dividend_u_valid = current_state == DIVU_WAIT ;
assign divisor_u_valid = current_state == DIVU_WAIT;

```

此外,在计算除法指令时,需要将前面的流水级进行阻塞,表现为将 allowin 信号改为 0,readygo 信号改为 0, 逻辑如下:

```

assign es_ready_go = !es_div_op[0] | (current_state==OUT_WAIT &
    out_valid |
    current_state==UOUT_WAIT & out_u_valid) ;
assign es_allow_in = (!es_valid || es_ready_go) && ms_allow_in &&
    (current_state == EXE | current_state==OUT_WAIT & out_valid |
    current_state==UOUT_WAIT & out_u_valid);
assign es_to_ms_valid = es_valid && es_ready_go;

```

4. 在传向 MEM 级的写入数据中加入有关 div, mod 指令的逻辑。

```

assign es_cal_result = es_div_op[0] ? (es_div_op[2] ?
    es_div_result:es_mod_result ):
    ((es_mul_op != 0) ? es_mul_result : es_alu_result);
assign es_div_result = es_div_op[1] ? es_div_signed : es_div_unsigned;
assign es_mod_result = es_div_op[1] ? es_mod_signed : es_mod_unsigned;

```

## 1.2. exp11

### 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 and 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

### 3.1. exp10

张钰堃负责添加算术指令，刘景平负责添加除法指令，付博宇负责添加乘法指令。



### 3.2. exp11

张钰堃负责添加转移指令，刘景平负责添加 load 指令，付博宇负责添加 store 指令。