

组成原理课程第四次实报告

实验名称：ALU 模块的复现和修改

学号：22122266 姓名：张恒硕 班次：0416

一、实验目的

1. 熟悉 MIPS 指令集中的运算指令，学会对这些指令进行归纳分类。
2. 了解 MIPS 指令结构。
3. 熟悉并掌握 ALU 的原理、功能和设计。
4. 进一步加强运用 verilog 语言进行电路设计的能力。
5. 为后续设计 cpu 的实验打下基础。

二、实验内容说明

1. 学习 MIPS 指令集，熟知指令类型，了解指令功能和编码，归纳基础 ALU 运算指令。
2. 实现加减运算，有无符号比较大小，按位与、或非、或、异或逻辑，逻辑左、右移，算术右移，高位加载共 12 种 ALU 运算。
3. 设计实验方案，画出结构框图。设计操作码位数和类型，使用独热码，在 ALU 内部先解码，再确定 ALU 作何种运算。
4. 使用 verilog 编写相应代码。
5. 仿真编写的代码，得到正确的波形图。
6. 将以上设计作为一个单独的模块，设计一个外围模块调用之。外围模块中需调用封装好的 LCD 触摸屏模块，显示 ALU 的两个源操作数、操作码和运算结果，并且需要利用触摸功能输入源操作数。操作码可以用 LCD 触摸屏输入，也可以用拨码开关输入。
7. 将编写的代码进行综合布局布线，并下载到试验箱中的 FPGA 板子上进行演示。
8. 重复上述步骤，通过修改代码，进一步实现与非、同或 2 种 ALU 运算，并改用二进制编码，再仿真、演示。

三、实验原理图

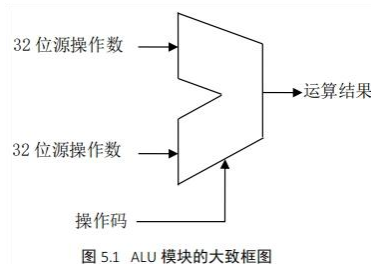


图 5.1 ALU 模块的大致框图

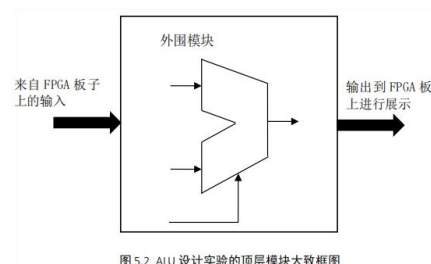
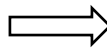


图 5.2 ALU 设计实验的顶层模块大致框图

表 5.1 ALU 的控制信号

控制信号												ALU 操作
11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	0	0	0	0	无
1	0	0	0	0	0	0	0	0	0	0	0	加法
0	1	0	0	0	0	0	0	0	0	0	0	减法
0	0	1	0	0	0	0	0	0	0	0	0	有符号比较, 小于置位
0	0	0	1	0	0	0	0	0	0	0	0	无符号比较, 小于置位
0	0	0	0	1	0	0	0	0	0	0	0	按位与
0	0	0	0	0	1	0	0	0	0	0	0	按位或非
0	0	0	0	0	0	1	0	0	0	0	0	按位或
0	0	0	0	0	0	0	1	0	0	0	0	按位异或
0	0	0	0	0	0	0	0	1	0	0	0	逻辑左移
0	0	0	0	0	0	0	0	0	1	0	0	逻辑右移
0	0	0	0	0	0	0	0	0	0	1	0	算术右移
0	0	0	0	0	0	0	0	0	0	0	1	高位加载

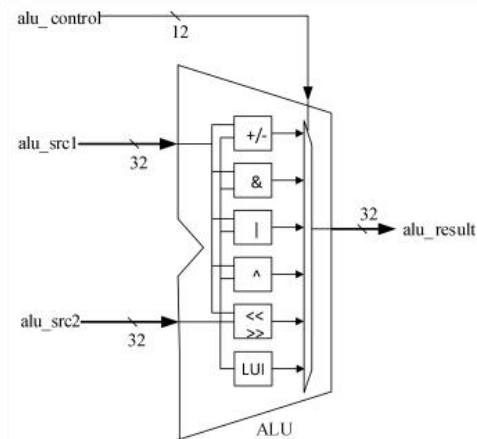


图 5.3 ALU 的原理图

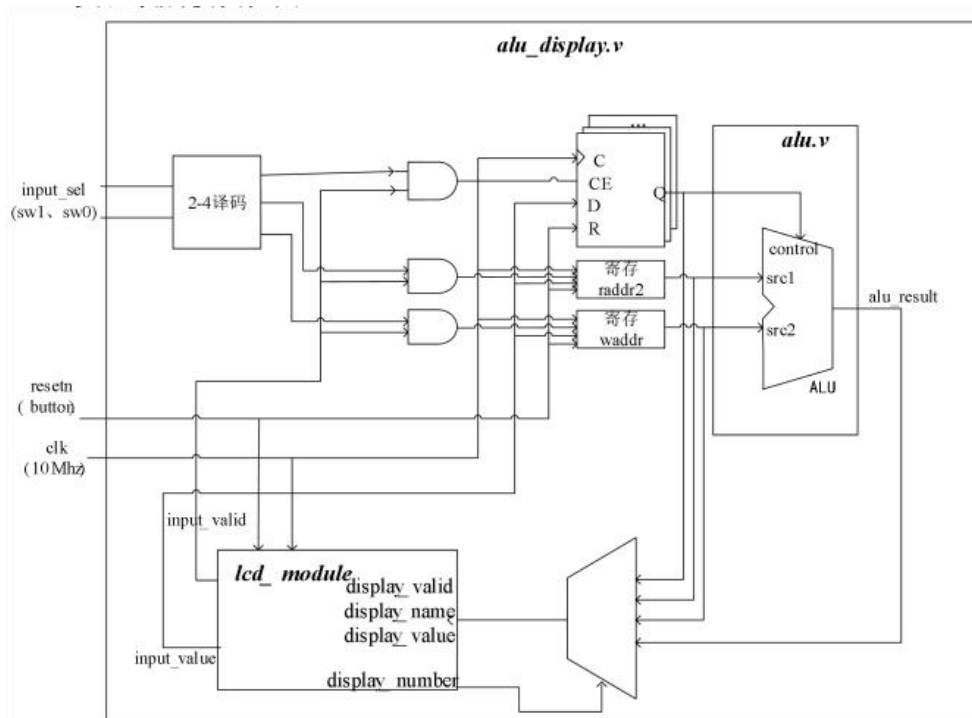


图 5.6 ALU 参考设计的顶层模块框图

以上给出了 ALU 模块的原理。

针对其中的加减运算和小于置位运算，以下给出相应原理：

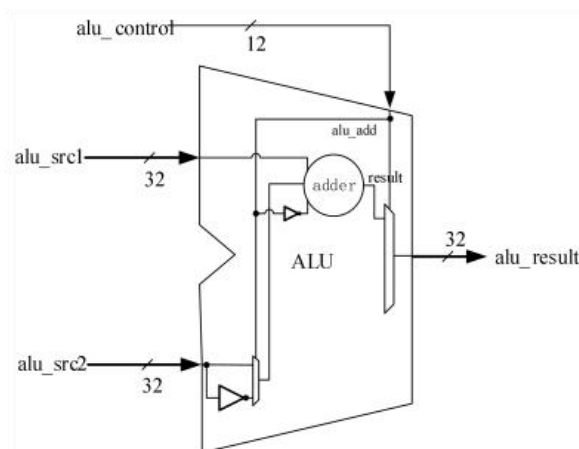


图 5.4 加减法运算的原理图

表 5.2 有符号比较小于置位的真值表

源操作数 1 符号位		源操作数 2 符号位		结果符号位		判断	slt 结果
alu_src1[31]		alu_src2[31]		adder_result[31]			
0	正数	0	正数	0	正数	正>正	0
0	正数	0	正数	1	负数	正<正	1
0	正数	1	负数	X	无关	正>负	0
1	负数	0	正数	X	无关	负<正	1
1	负数	1	负数	0	正数	负>负	0
1	负数	1	负数	1	负数	负<负	1

运算结果表达式：

$$\text{slt_result} = (\text{alu_src1}[31] \ \& \ \sim \text{alu_src2}[31]) \mid (\sim (\text{alu_src1}[31] \ \wedge \ \text{alu_src2}[31]) \ \& \$$

adder_result[31])

以下加入两种 ALU 运算，分别是与非、同或，其逻辑表达式如下：

与非： $\sim(A*B)$

同或： $(\sim A)*(\sim B)+A*B$

采用二进制编码，操作码对应如下：

高位 加载	算 术 右 移	逻 辑 右 移	逻 辑 左 移	按 位 异 或	按 位 或	按 位 或 非	按 位 与	无 符 号 比 较	有 符 号 比 较	减 法	加 法	按 位 同 或	按 位 与 非
lui	sra	srl	sll	xor	or	nor	and	sltu	slt	sub	add	xnor	and
0	1	2	3	4	5	6	7	8	9	10	11	12	13
000 0	000 1	001 0	001 1	010 0	010 1	011 0	011 1	100 0	100 1	101 0	101 1	110 0	110 1

四、实验步骤

以下给出了 ALU 模块的代码。

```
module alu(  
    input  [11:0] alu_control, // ALU 控制信号  
    input  [31:0] alu_src1,    // ALU 操作数 1，为补码  
    input  [31:0] alu_src2,    // ALU 操作数 2，为补码  
    output [31:0] alu_result   // ALU 结果  
);  
  
    // ALU 控制信号，独热码  
    wire alu_add;    //加法操作  
    wire alu_sub;    //减法操作  
    wire alu_slt;    //有符号比较，小于置位，复用加法器做减法  
    wire alu_sltu;   //无符号比较，小于置位，复用加法器做减法  
    wire alu_and;    //按位与  
    wire alu_nor;    //按位或非  
    wire alu_or;     //按位或  
    wire alu_xor;    //按位异或  
    wire alu_sll;    //逻辑左移  
    wire alu_srl;    //逻辑右移  
    wire alu_sra;    //算术右移  
    wire alu_lui;    //高位加载  
  
    assign alu_add  = alu_control[11];  
    assign alu_sub  = alu_control[10];  
    assign alu_slt  = alu_control[ 9];  
    assign alu_sltu = alu_control[ 8];  
    assign alu_and  = alu_control[ 7];  
    assign alu_nor  = alu_control[ 6];
```

```

assign alu_or    = alu_control[ 5];
assign alu_xor   = alu_control[ 4];
assign alu_sll   = alu_control[ 3];
assign alu_srl   = alu_control[ 2];
assign alu_sra   = alu_control[ 1];
assign alu_lui   = alu_control[ 0];

wire [31:0] add_sub_result;
wire [31:0] slt_result;
wire [31:0] sltu_result;
wire [31:0] and_result;
wire [31:0] nor_result;
wire [31:0] or_result;
wire [31:0] xor_result;
wire [31:0] sll_result;
wire [31:0] srl_result;
wire [31:0] sra_result;
wire [31:0] lui_result;

assign and_result = alu_src1 & alu_src2;      // 与结果为两数按位与
assign or_result  = alu_src1 | alu_src2;      // 或结果为两数按位或
assign nor_result = ~or_result;              // 或非结果为或结果按位取反
assign xor_result = alu_src1 ^ alu_src2;      // 异或结果为两数按位异或
assign lui_result = {alu_src2[15:0], 16'd0};  // 立即数装载结果为立即数移位
至高半字节

//-----{加法器}begin
//add, sub, slt, sltu 均使用该模块
wire [31:0] adder_operand1;
wire [31:0] adder_operand2;
wire        adder_cin      ;
wire [31:0] adder_result  ;
wire        adder_cout     ;
assign adder_operand1 = alu_src1;
assign adder_operand2 = alu_add ? alu_src2 : ~alu_src2;
assign adder_cin       = ~alu_add; //减法需要 cin
adder adder_module(
    .operand1(adder_operand1),
    .operand2(adder_operand2),
    .cin      (adder_cin      ),
    .result   (adder_result  ),
    .cout     (adder_cout     )
);

```

```

//加减结果
assign add_sub_result = adder_result;

//slt 结果
//adder_src1[31] adder_src2[31] adder_result[31]
//      0      1      X(0 或 1)      "正-负", 显然小于不成立
//      0      0      1      相减为负, 说明小于
//      0      0      0      相减为正, 说明不小于
//      1      1      1      相减为负, 说明小于
//      1      1      0      相减为正, 说明不小于
//      1      0      X(0 或 1)      "负-正", 显然小于成立
assign slt_result[31:1] = 31'd0;
assign slt_result[0] = (alu_src1[31] & ~alu_src2[31]) |
(~(alu_src1[31]^alu_src2[31]) & adder_result[31]);

//sltu 结果
//对于 32 位无符号数比较, 相当于 33 位有符号数 ({1'b0, src1} 和 {1'b0, src2}) 的比
//较, 最高位 0 为符号位
//故, 可以用 33 位加法器来比较大小, 需要对 {1'b0, src2} 取反, 即需要
{1'b0, src1}+{1'b1, ~src2}+cin
//但此处用的为 32 位加法器, 只做了运算:
src1 +
~src2 +cin
//32 位加法的结果为 {adder_cout, adder_result}, 则 33 位加法结果应该为
{adder_cout+1'b1, adder_result}
//对比 slt 结果注释, 知道, 此时判断大小属于第二三种情况, 即源操作数 1 符号位为
0, 源操作数 2 符号位为 0
//结果的符号位为 1, 说明小于, 即 adder_cout+1'b1 为 2'b01, 即 adder_cout 为 0
assign sltu_result = {31'd0, ~adder_cout};
//-----{加法器}end

//-----{移位器}begin
// 移位分三步进行,
// 第一步根据移位量低 2 位即 [1:0] 位做第一次移位,
// 第二步在第一次移位基础上根据移位量 [3:2] 位做第二次移位,
// 第三步在第二次移位基础上根据移位量 [4] 位做第三次移位。
wire [4:0] shf;
assign shf = alu_src1[4:0];
wire [1:0] shf_1_0;
wire [1:0] shf_3_2;
assign shf_1_0 = shf[1:0];
assign shf_3_2 = shf[3:2];

// 逻辑左移
wire [31:0] sll_step1;

```

```

    wire [31:0] sll_step2;
    assign sll_step1 = {32{shf_1_0 == 2'b00}} & alu_src2 // 若
shf[1:0]="00",不移位
                    | {32{shf_1_0 == 2'b01}} & {alu_src2[30:0], 1'd0} // 若
shf[1:0]="01",左移 1 位
                    | {32{shf_1_0 == 2'b10}} & {alu_src2[29:0], 2'd0} // 若
shf[1:0]="10",左移 2 位
                    | {32{shf_1_0 == 2'b11}} & {alu_src2[28:0], 3'd0}; // 若
shf[1:0]="11",左移 3 位
    assign sll_step2 = {32{shf_3_2 == 2'b00}} & sll_step1 // 若
shf[3:2]="00",不移位
                    | {32{shf_3_2 == 2'b01}} & {sll_step1[27:0], 4'd0} // 若
shf[3:2]="01",第一次移位结果左移 4 位
                    | {32{shf_3_2 == 2'b10}} & {sll_step1[23:0], 8'd0} // 若
shf[3:2]="10",第一次移位结果左移 8 位
                    | {32{shf_3_2 == 2'b11}} & {sll_step1[19:0], 12'd0}; // 若
shf[3:2]="11",第一次移位结果左移 12 位
    assign sll_result = shf[4] ? {sll_step2[15:0], 16'd0} : sll_step2; // 若
shf[4]="1",第二次移位结果左移 16 位

// 逻辑右移
wire [31:0] srl_step1;
wire [31:0] srl_step2;
    assign srl_step1 = {32{shf_1_0 == 2'b00}} & alu_src2 // 若
shf[1:0]="00",不移位
                    | {32{shf_1_0 == 2'b01}} & {1'd0, alu_src2[31:1]} // 若
shf[1:0]="01",右移 1 位,高位补 0
                    | {32{shf_1_0 == 2'b10}} & {2'd0, alu_src2[31:2]} // 若
shf[1:0]="10",右移 2 位,高位补 0
                    | {32{shf_1_0 == 2'b11}} & {3'd0, alu_src2[31:3]}; // 若
shf[1:0]="11",右移 3 位,高位补 0
    assign srl_step2 = {32{shf_3_2 == 2'b00}} & srl_step1 // 若
shf[3:2]="00",不移位
                    | {32{shf_3_2 == 2'b01}} & {4'd0, srl_step1[31:4]} // 若
shf[3:2]="01",第一次移位结果右移 4 位,高位补 0
                    | {32{shf_3_2 == 2'b10}} & {8'd0, srl_step1[31:8]} // 若
shf[3:2]="10",第一次移位结果右移 8 位,高位补 0
                    | {32{shf_3_2 == 2'b11}} & {12'd0, srl_step1[31:12]}; // 若
shf[3:2]="11",第一次移位结果右移 12 位,高位补 0
    assign srl_result = shf[4] ? {16'd0, srl_step2[31:16]} : srl_step2; // 若
shf[4]="1",第二次移位结果右移 16 位,高位补 0

// 算术右移
wire [31:0] sra_step1;

```

```

    wire [31:0] sra_step2;
    assign sra_step1 = {32{shf_1_0 == 2'b00}} & alu_src2
// 若 shf[1:0]="00",不移位
        | {32{shf_1_0 == 2'b01}} & {alu_src2[31], alu_src2[31:1]}
// 若 shf[1:0]="01",右移 1 位,高位补符号位
        | {32{shf_1_0 == 2'b10}} & {{2{alu_src2[31]}}, alu_src2[31:2]}
// 若 shf[1:0]="10",右移 2 位,高位补符号位
        | {32{shf_1_0 == 2'b11}} & {{3{alu_src2[31]}},
alu_src2[31:3]}; // 若 shf[1:0]="11",右移 3 位,高位补符号位
    assign sra_step2 = {32{shf_3_2 == 2'b00}} & sra_step1
// 若 shf[3:2]="00",不移位
        | {32{shf_3_2 == 2'b01}} & {{4{sra_step1[31]}},
sra_step1[31:4]} // 若 shf[3:2]="01",第一次移位结果右移 4 位,高位补符号位
        | {32{shf_3_2 == 2'b10}} & {{8{sra_step1[31]}},
sra_step1[31:8]} // 若 shf[3:2]="10",第一次移位结果右移 8 位,高位补符号位
        | {32{shf_3_2 == 2'b11}} & {{12{sra_step1[31]}},
sra_step1[31:12]}; // 若 shf[3:2]="11",第一次移位结果右移 12 位,高位补符号位
    assign sra_result = shf[4] ? {{16{sra_step2[31]}}, sra_step2[31:16]} :
sra_step2; // 若 shf[4]="1",第二次移位结果右移 16 位,高位补符号位
//-----{移位器}end

// 选择相应结果输出
assign alu_result = (alu_add|alu_sub) ? add_sub_result[31:0] :
                    alu_slt           ? slt_result :
                    alu_sltu          ? sltu_result :
                    alu_and            ? and_result :
                    alu_nor            ? nor_result :
                    alu_or             ? or_result :
                    alu_xor            ? xor_result :
                    alu_sll            ? sll_result :
                    alu_srl            ? srl_result :
                    alu_sra            ? sra_result :
                    alu_lui            ? lui_result :
                    32'd0;
endmodule

```

代码先声明了 ALU 控制信号 `alu_control`、ALU 操作数 `alu_src1` 和 `alu_src2`、ALU 结果 `alu_result`。通过 ALU 控制信号，利用独热码的方式，设定了 12 种 ALU 运算。每种运算都有对应的运算逻辑。

以下给出改进代码，只给出有改动的部分，改动部分已标红：

```

module alu(
// input  [11:0] alu_control, // ALU 控制信号，独热编码
input  [3:0] alu_control,    // ALU 控制信号，二进制编码
input  [31:0] alu_src1,      // ALU 操作数 1,为补码
input  [31:0] alu_src2,      // ALU 操作数 2,为补码

```



```

output [31:0] alu_result    // ALU 结
);

// ALU 控制信号, 独热码
wire alu_add;    //加法操作
wire alu_sub;    //减法操作
wire alu_slt;    //有符号比较, 小于置位, 复用加法器做减法
wire alu_sltu;   //无符号比较, 小于置位, 复用加法器做减法
wire alu_and;    //按位与
wire alu_nand;   //按位与非
wire alu_nor;    //按位或非
wire alu_or;     //按位或
wire alu_xor;    //按位异或
wire alu_xnor;   //按位同或
wire alu_sll;    //逻辑左移
wire alu_srl;    //逻辑右移
wire alu_sra;    //算术右移
wire alu_lui;    //高位加载

    assign alu_nand = alu_control[ 3] & alu_control[ 2] & ~alu_control[ 1] &
alu_control[ 0];
    assign alu_xnor = alu_control[ 3] & alu_control[ 2] & ~alu_control[ 1] &
~alu_control[ 0];
    assign alu_add  = alu_control[ 3] & ~alu_control[ 2] & alu_control[ 1] &
alu_control[ 0];
    assign alu_sub  = alu_control[ 3] & ~alu_control[ 2] & alu_control[ 1] &
~alu_control[ 0];
    assign alu_slt  = alu_control[ 3] & ~alu_control[ 2] & ~alu_control[ 1] &
alu_control[ 0];
    assign alu_sltu = alu_control[ 3] & ~alu_control[ 2] & ~alu_control[ 1] &
~alu_control[ 0];
    assign alu_and  = ~alu_control[ 3] & alu_control[ 2] & alu_control[ 1] &
alu_control[ 0];
    assign alu_nor  = ~alu_control[ 3] & alu_control[ 2] & alu_control[ 1] &
~alu_control[ 0];
    assign alu_or   = ~alu_control[ 3] & alu_control[ 2] & ~alu_control[ 1] &
alu_control[ 0];
    assign alu_xor  = ~alu_control[ 3] & alu_control[ 2] & ~alu_control[ 1] &
~alu_control[ 0];
    assign alu_sll  = ~alu_control[ 3] & ~alu_control[ 2] & alu_control[ 1] &
alu_control[ 0];
    assign alu_srl  = ~alu_control[ 3] & ~alu_control[ 2] & alu_control[ 1] &
~alu_control[ 0];
    assign alu_sra  = ~alu_control[ 3] & ~alu_control[ 2] & ~alu_control[ 1] &

```

```

alu_control[ 0];
    assign alu_lui  = ~alu_control[ 3] & ~alu_control[ 2] & ~alu_control[ 1] &
~alu_control[ 0];

    wire [31:0] nand_result;
    wire [31:0] xnor_result;
    wire [31:0] add_sub_result;
    wire [31:0] slt_result;
    wire [31:0] sltu_result;
    wire [31:0] and_result;
    wire [31:0] nor_result;
    wire [31:0] or_result;
    wire [31:0] xor_result;
    wire [31:0] sll_result;
    wire [31:0] srl_result;
    wire [31:0] sra_result;
    wire [31:0] lui_result;

    assign and_result = alu_src1 & alu_src2;          // 与结果为两数按位与
    assign nand_result= ~and_result;                 // 与非结果为与结果按位取反
    assign or_result  = alu_src1 | alu_src2;          // 或结果为两数按位或
    assign nor_result = ~or_result;                   // 或非结果为或结果按位取反
    assign xor_result = alu_src1 ^ alu_src2;          // 异或结果为两数按位异或
    assign xnor_result = ~or_result | and_result;     // 同或结果为两数按位同或
    assign lui_result = {alu_src2[15:0], 16'd0};      // 立即数装载结果为立即数移位
至高半字节

    // 选择相应结果输出
    assign alu_result = (alu_add|alu_sub) ? add_sub_result[31:0] :
        alu_slt      ? slt_result :
        alu_sltu     ? sltu_result :
        alu_and      ? and_result :
        alu_nor      ? nor_result :
        alu_or       ? or_result  :
        alu_xor      ? xor_result :
        alu_sll      ? sll_result :
        alu_srl      ? srl_result :
        alu_sra      ? sra_result :
        alu_lui      ? lui_result :
        alu_nand     ? nand_result :
        alu_xnor     ? xnor_result :
        32'd0;


endmodule

```

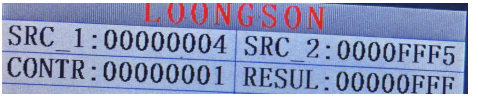
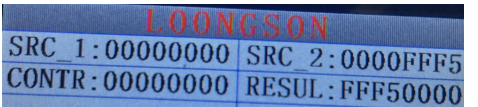



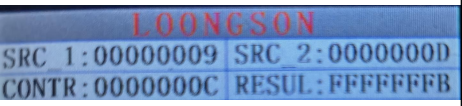
五、实验结果分析

ALU 模块

操作	Case1	Case2
加法	<div>LOONGSON</div> <div>SRC_1:0000152A SRC_2:000926EF</div> <div>CONTR:00000800 RESUL:00093C19</div>	<div>LOONGSON</div> <div>SRC_1:00000006 SRC_2:00000002</div> <div>CONTR:00000800 RESUL:00000008</div>
减法	<div>LOONGSON</div> <div>SRC_1:000267AF SRC_2:000133AA</div> <div>CONTR:00000400 RESUL:00013405</div>	<div>LOONGSON</div> <div>SRC_1:000267AF SRC_2:00030000</div> <div>CONTR:00000400 RESUL:FFFF67AF</div>
有符号比较, 小于置位	<div>LOONGSON</div> <div>SRC_1:0002A999 SRC_2:00030000</div> <div>CONTR:00000200 RESUL:00000001</div>	<div>LOONGSON</div> <div>SRC_1:00000020 SRC_2:00000005</div> <div>CONTR:00000200 RESUL:00000000</div>
	<div>LOONGSON</div> <div>SRC_1:00000005 SRC_2:00000005</div> <div>CONTR:00000200 RESUL:00000000</div>	<div>LOONGSON</div> <div>SRC_1:80000001 SRC_2:00000005</div> <div>CONTR:00000200 RESUL:00000001</div>
无符号比较, 小于置位	<div>LOONGSON</div> <div>SRC_1:A0000000 SRC_2:80000002</div> <div>CONTR:00000100 RESUL:00000000</div>	<div>LOONGSON</div> <div>SRC_1:80000001 SRC_2:80000002</div> <div>CONTR:00000100 RESUL:00000001</div>
按位与	<div>LOONGSON</div> <div>SRC_1:8421F731 SRC_2:FFFFFFFF</div> <div>CONTR:00000080 RESUL:8421F731</div>	<div>LOONGSON</div> <div>SRC_1:8421F731 SRC_2:000000BA</div> <div>CONTR:00000080 RESUL:00000030</div>
按位或非	<div>LOONGSON</div> <div>SRC_1:8421F731 SRC_2:000000BA</div> <div>CONTR:00000040 RESUL:7BDE0844</div>	<div>LOONGSON</div> <div>SRC_1:8421F731 SRC_2:FFFFFFFF</div> <div>CONTR:00000040 RESUL:00000000</div>
按位或	<div>LOONGSON</div> <div>SRC_1:8421F731 SRC_2:FFFFFFFF</div> <div>CONTR:00000020 RESUL:FFFFFFFF</div>	<div>LOONGSON</div> <div>SRC_1:FFF0FF0E SRC_2:000700F1</div> <div>CONTR:00000020 RESUL:FFF7FFFF</div>
按位异或	<div>LOONGSON</div> <div>SRC_1:BA549358 SRC_2:BA549358</div> <div>CONTR:00000010 RESUL:00000000</div>	<div>LOONGSON</div> <div>SRC_1:BA549358 SRC_2:BA669008</div> <div>CONTR:00000010 RESUL:00320350</div>
逻辑左移	<div>LOONGSON</div> <div>SRC_1:00000004 SRC_2:0FFFFFFFFF</div> <div>CONTR:00000008 RESUL:FFFFFFFF0</div>	<div>LOONGSON</div> <div>SRC_1:00000001 SRC_2:0FFFFFFFFF</div> <div>CONTR:00000008 RESUL:1FFFFFFFFE</div>
逻辑右移	<div>LOONGSON</div> <div>SRC_1:00000001 SRC_2:0FFFFFFFFF</div> <div>CONTR:00000004 RESUL:07FFFFFFF</div>	<div>LOONGSON</div> <div>SRC_1:00000004 SRC_2:0FFFFFFFFF</div> <div>CONTR:00000004 RESUL:00FFFFFFF</div>
算术右移	<div>LOONGSON</div> <div>SRC_1:00000001 SRC_2:800000FF</div> <div>CONTR:00000002 RESUL:C000007F</div>	<div>LOONGSON</div> <div>SRC_1:00000004 SRC_2:800FFFFFFF</div> <div>CONTR:00000002 RESUL:F800FFFFFF</div>
高位加载	<div>LOONGSON</div> <div>SRC_1:00000000 SRC_2:00000FFF</div> <div>CONTR:00000001 RESUL:0FFF0000</div>	<div>LOONGSON</div> <div>SRC_1:00000000 SRC_2:000032FF</div> <div>CONTR:00000001 RESUL:32FF0000</div>

无		
---	---	--

改动后的 ALU 模块，主要呈现改变的编码方式和新添加的两种运算，对之前已有的运算仅选择个例展示

操作	Case1	Case2
逻辑右移		
高位加载		
与非		
同或		

以上展示了不同编码方式的实际操作成果，代码成功实现了预期的功能。

六、总结感想

本次实验实现了 ALU 模块这一内容，并对其进行了改动。在本次实验中，对 ALU 这一算术逻辑单元的实现逻辑和功能有了更深入的理解，并学习了独热码的编码方式。更进一步地改动加强了对操作数进行二进制编码的理解。通过本次实验，向着实现 CPU 的最后目标又迈进了一步。