组成原理课程第 四 次实报告

实验名称：ALU 设计

学号：2310420 姓名：王晶 班次：周二上午

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

实验内容说明

请根据实验指导手册中完成实验四 ALU 实验的代码功能验证，并做以下改进：

1、原有的操作码为 12 位，请压缩到操作码控制型号位宽为 4 位。

2、在操作码调整到 4 位之后，应该能支持 15 种不同运算，请添加至少三种运算功能（有符号数比较和无符号数比较的大于置位运算、一种未实现的位运算），然后上实验箱验证（可以不用仿真）。

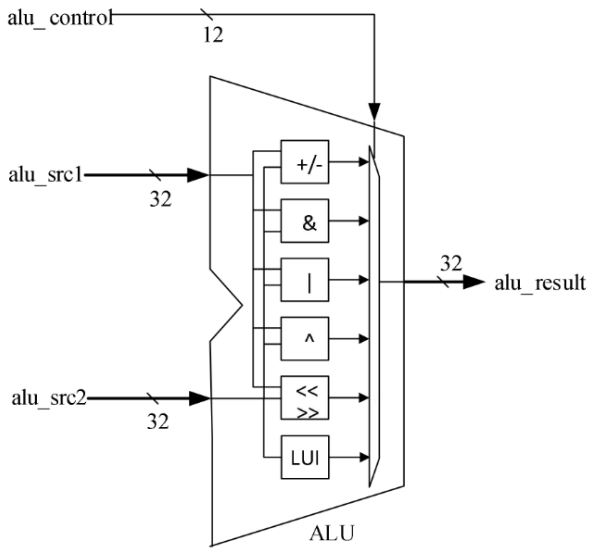
3、实验报告中的原理图就用图 5.3 即可，不再是顶层模块图。

4、添加的三种运算上箱需要有验证照片。

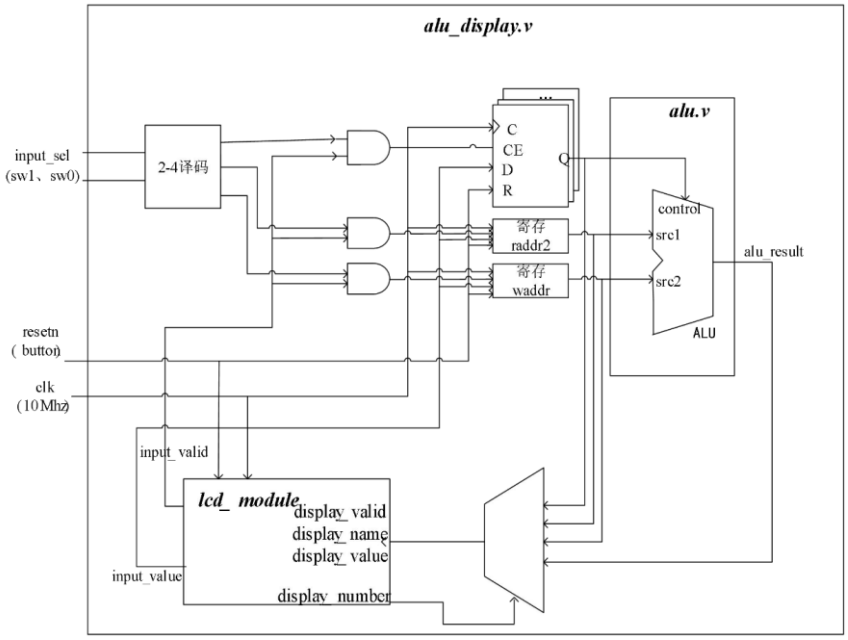
提交时文件名格式为“学号_姓名_组成原理第四次实验.pdf”。注意实验报告中要有介绍分析的内容，针对实验箱照片，要解释图中信息，是否验证成功。

实验原理图

（1）ALU 模块的原理图：



（2）顶层模块原理图



实验原理：

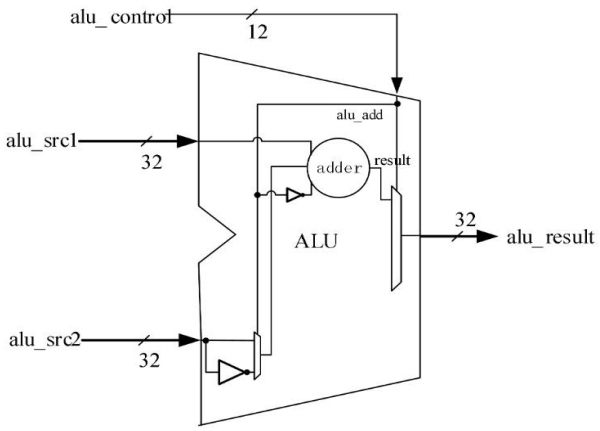
（1）独热编码：

将 ALU control 位数由 16 位改成 4 位，则每个功能编码如下：

编码	ALU 操作
0	无
1	加法

2	减法
3	有符号比较，小于置位
4	无符号比较，小于置位
5	按位与
6	按位或非
7	按位或
8	按位异或
9	逻辑左移
A	逻辑右移
B	算数右移
C	高位加载
D	有符号比较，大于置位
E	无符号比较，大于置位
F	按位同或

（2）加减法原理图：



（3）有符号比较小于置位的真值表：

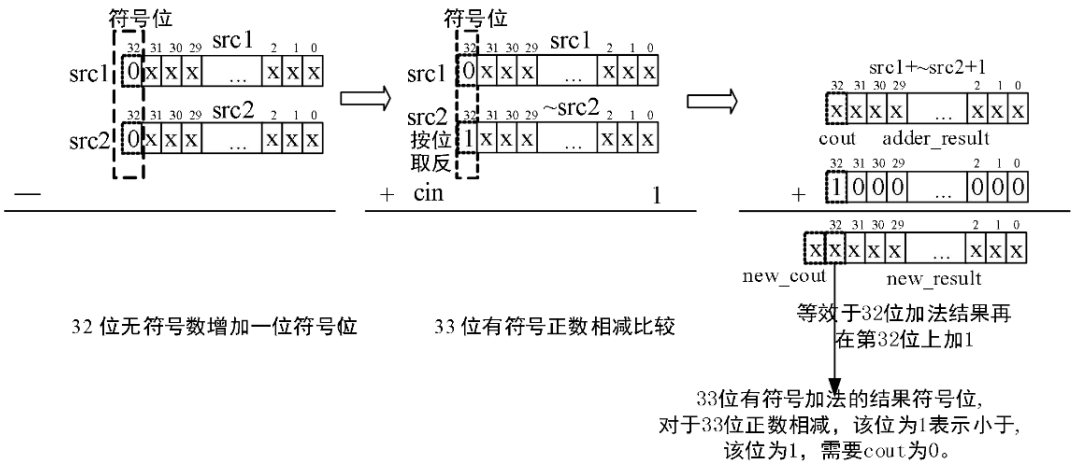
源操作数 1 符号位		源操作数 2 符号位		结果符号位		判断	slt 结果
alu_src1[31]		alu_src2[31]		adder_result[31]			
0	正数	0	正数	0	正数	正>正	0
0	正数	0	正数	1	负数	正<正	1
0	正数	1	负数	X	无关	正>负	0
1	负数	0	正数	X	无关	负<正	1
1	负数	1	负数	0	正数	负>负	0
1	负数	1	负数	1	负数	负<负	1

可以写出代码为：

```
slt_result = (alu_src1[31] & ~alu_src2[31]) |
(~(alu_src1[31]^alu_src2[31]) &
adder_result[31])
```

（4）无符号 32 位数比较的原理图：

对于 32 位无符号比较的小于置位，可在其高位前填 0 组合为 33 位正数的比较，即{1’b0, src1}和{1’b0, src2}的比较，最高位符号位为 0。对比可知，对于正数的比较，只要减法结果的符号位为 1，则表示小于。而 33 位正数相减，其结果的符号位最终可由 32 位加法的 cout+1’b1 得到。故无符号 32 位比较小于置位运算结果表达式为：sltu_result = ~adder_cout.



（5）按位与/按位或非/按位或/按位异或

```
assign and_result = alu_src1 & alu_src2;
assign or_result = alu_src1 | alu_src2;
assign nor_result = ~or_result;
assign xor_result = alu_src1 ^ alu_src2;
```

（6）逻辑左移/逻辑右移/算数左移

```
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
```

（7）高位加载

```
assign lui_result = {alu_src2[15:0], 16'd0};
```

（8）有符号比较，大于置位

```
assign sgt_result[31:1] = 31'd0;
    assign sgt_result[0] = (alu_src1[31] & ~alu_src2[31]) ? 1'b0 :
                        (~alu_src1[31] & alu_src2[31]) ? 1'b1 :
                        ~adder_result[31] & (|adder_result);
```

（9）无符号比较，大于置位

```
assign sgtu_result = {31'd0, (~adder_cout) & (|adder_result)};
```

（10）按位同或

```
assign nxor_result = ~xor_result;
```

实验步骤

代码修改部分如下：

在 alu.v 文件中

（1）修改控制信号的位数：

```
input  [3:0] alu_control,  // ALU 控制信号
```

（2）加上信号定义部分

```
    wire alu_nxor;
    wire alu_sgt;
    wire alu_sgtu;
    assign
alu_add=(~alu_control[3])&(~alu_control[2])&(~alu_control[1])&(alu
_control[0]);//0001
    assign
alu_sub=(~alu_control[3])&(~alu_control[2])&(alu_control[1])&(~alu
_control[0]);//0010
    assign
alu_slt=(~alu_control[3])&(~alu_control[2])&(alu_control[1])&(alu_
control[0]);//0011
    assign
```

```
alu_sltu=(~alu_control[3])&(alu_control[2])&(~alu_control[1])&(~al
u_control[0]);//0100
    assign
alu_and=(~alu_control[3])&(alu_control[2])&(~alu_control[1])&(alu_
control[0]);//0101
    assign
alu_nor=(~alu_control[3])&(alu_control[2])&(alu_control[1])&(~alu_
control[0]);//0110
    assign
alu_or=(~alu_control[3])&(alu_control[2])&(alu_control[1])&(alu_co
ntrol[0]);//0111
    assign
alu_xor=(alu_control[3])&(~alu_control[2])&(~alu_control[1])&(~alu
_control[0]);//1000
    assign
alu_sll=(alu_control[3])&(~alu_control[2])&(~alu_control[1])&(alu_
control[0]);//1001
    assign
alu_srl=(alu_control[3])&(~alu_control[2])&(alu_control[1])&(~alu_
control[0]);//1010
    assign
alu_sra=(alu_control[3])&(~alu_control[2])&(alu_control[1])&(alu_c
ontrol[0]);//1011
    assign
alu_lui=(alu_control[3])&(alu_control[2])&(~alu_control[1])&(~alu_
control[0]);//1100
    assign
alu_sgt=(alu_control[3])&(alu_control[2])&(~alu_control[1])&(alu_c
ontrol[0]);//1101
    assign
alu_sgtu=(alu_control[3])&(alu_control[2])&(alu_control[1])&(~alu_
control[0]);//1110
    assign
alu_nxor=(alu_control[3])&(alu_control[2])&(alu_control[1])&(alu_c
ontrol[0]);//1111
```

（3）为新的三个模块赋值：

```
assign nxor_result =~(alu_src1^slu_src2);
assign sgt_result[31:1] = 31'd0;
assign sgt_result[0] = (alu_src1[31] & ~alu_src2[31]) ? 1'b0 :
                       (~alu_src1[31] & alu_src2[31]) ? 1'b1 :
                       ~adder_result[31] & (|adder_result);

assign sgtu_result = {31'd0, (~adder_cout) & (|adder_result)};
```

（4）修改最后的输出

```
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_sgt           ? sgt_result :
                        alu_sgtu           ? sgtu_result :
                        alu_nxor           ? nxor_result :
                        32'd0;
```

在 alu_display.v 文件中

（1）对控制信号的位数进行修改：

```
reg   [3:0] alu_control;  // ALU 控制信号
```

修改后的文件

alu.v

```
`timescale 1ns / 1ps
//*************************************************************
*********
//   > 文件名: alu.v
//   > 描述   : ALU 模块，可做 12 种操作
//   > 作者   : LOONGSON
//   > 日期   : 2016-04-14
//*************************************************************
*********
module alu(
    input  [3:0] 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
    wire alu_sgt;
    wire alu_nxor;
    wire alu_sgtu;

    assign
alu_add=(~alu_control[3])&(~alu_control[2])&(~alu_control[1])&(alu
_control[0]);//0001
    assign
alu_sub=(~alu_control[3])&(~alu_control[2])&(alu_control[1])&(~alu
_control[0]);//0010
    assign
alu_slt=(~alu_control[3])&(~alu_control[2])&(alu_control[1])&(alu_
control[0]);//0011
    assign
alu_sltu=(~alu_control[3])&(alu_control[2])&(~alu_control[1])&(~al
u_control[0]);//0100
    assign
alu_and=(~alu_control[3])&(alu_control[2])&(~alu_control[1])&(alu_
control[0]);//0101
    assign
alu_nor=(~alu_control[3])&(alu_control[2])&(alu_control[1])&(~alu_
control[0]);//0110
    assign
alu_or=(~alu_control[3])&(alu_control[2])&(alu_control[1])&(alu_co
ntrol[0]);//0111
    assign
alu_xor=(alu_control[3])&(~alu_control[2])&(~alu_control[1])&(~alu
_control[0]);//1000
    assign
```

```
alu_sll=(alu_control[3])&(~alu_control[2])&(~alu_control[1])&(alu_
control[0]);//1001
    assign
alu_srl=(alu_control[3])&(~alu_control[2])&(alu_control[1])&(~alu_
control[0]);//1010
    assign
alu_sra=(alu_control[3])&(~alu_control[2])&(alu_control[1])&(alu_c
ontrol[0]);//1011
    assign
alu_lui=(alu_control[3])&(alu_control[2])&(~alu_control[1])&(~alu_
control[0]);//1100
    assign
alu_sgt=(alu_control[3])&(alu_control[2])&(~alu_control[1])&(alu_c
ontrol[0]);//1101
    assign
alu_sgtu=(alu_control[3])&(alu_control[2])&(alu_control[1])&(~alu_
control[0]);//1110
    assign
alu_nxor=(alu_control[3])&(alu_control[2])&(alu_control[1])&(alu_c
ontrol[0]);//1111




    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;
    wire [31:0] sgt_result;
    wire [31:0] sgtu_result;
    wire [31:0] nxor_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


    assign nxor_result=~xor_result;







//-----{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]);
    assign sltu_result = {31'd0, ~adder_cout};

    wire adder_zero;
    assign adder_zero=|adder_result;
    assign sgt_result[31:1] = 31'd0;
    assign sgt_result[0] = (alu_src1[31] & ~alu_src2[31]) ? 1'b0 :
                      (~alu_src1[31] & alu_src2[31]) ? 1'b1 :
                      ~adder_result[31] & (|adder_result); // 不
小于且不等于

    assign sgtu_result = {31'd0, (~adder_cout) & (|adder_result)};
// 无符号大于的情况

//-----{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 :
                        alu_sgt           ? sgt_result :
```

```
                        alu_sgtu           ? sgtu_result :
                        alu_nxor           ? nxor_result :
                        32'd0;
endmodule
```

alu_display.v

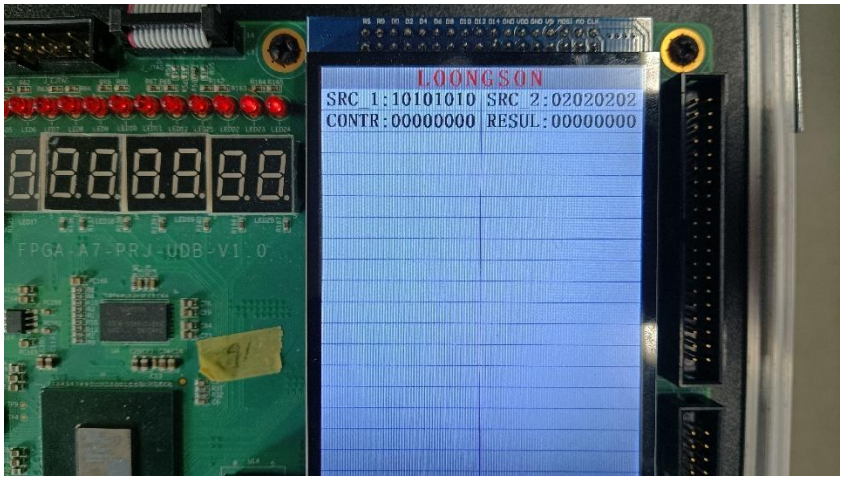
```
//*************************************************************
*********
//   > 文件名: alu_display.v
//   > 描述   : ALU 显示模块，调用 FPGA 板上的 IO 接口和触摸屏
//   > 作者   : LOONGSON
//   > 日期   : 2016-04-14
//*************************************************************
*********
module alu_display(
    //时钟与复位信号
    input clk,
    input resetn,    //后缀"n"代表低电平有效

    //拨码开关，用于选择输入数
    input [1:0] input_sel, //00:输入为控制信号(alu_control)
                           //10:输入为源操作数 1(alu_src1)
                           //11:输入为源操作数 2(alu_src2)

    //触摸屏相关接口，不需要更改
    output lcd_rst,
    output lcd_cs,
    output lcd_rs,
    output lcd_wr,
    output lcd_rd,
    inout[15:0] lcd_data_io,
    output lcd_bl_ctr,
    inout ct_int,
    inout ct_sda,
    output ct_scl,
    output ct_rstn
    );
//-----{调用 ALU 模块}begin
    reg   [3:0] alu_control;  // ALU 控制信号
    reg   [31:0] alu_src1;     // ALU 操作数 1
    reg   [31:0] alu_src2;     // ALU 操作数 2
```

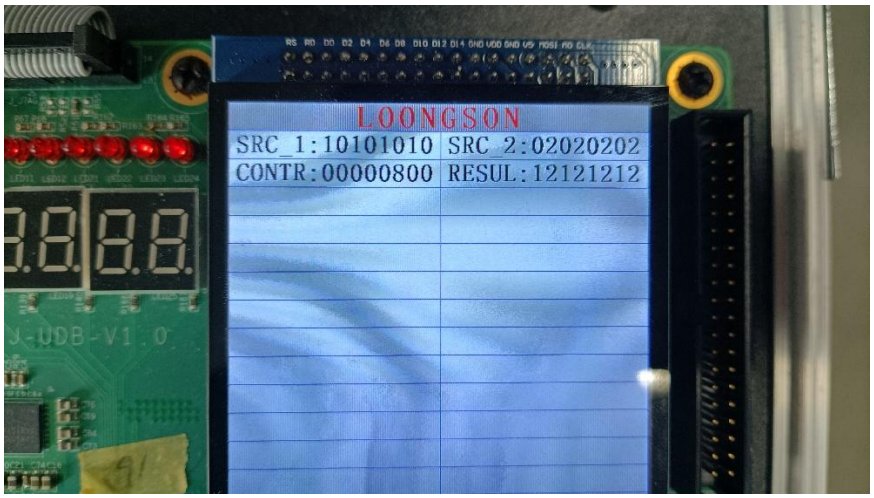
```
    wire  [31:0] alu_result;   // ALU 结果
    alu alu_module(
        .alu_control(alu_control),
        .alu_src1   (alu_src1   ),
        .alu_src2   (alu_src2   ),
        .alu_result (alu_result )
    );
//-----{调用 ALU 模块}end

//--------------------{调用触摸屏模块}begin--------------------//
//-----{实例化触摸屏}begin
//此小节不需要更改
    reg         display_valid;
    reg  [39:0] display_name;
    reg  [31:0] display_value;
    wire [5 :0] display_number;
    wire        input_valid;
    wire [31:0] input_value;

    lcd_module lcd_module(
        .clk            (clk           ),   //10Mhz
        .resetn         (resetn        ),

        //调用触摸屏的接口
        .display_valid  (display_valid ),
        .display_name   (display_name  ),
        .display_value  (display_value ),
        .display_number (display_number),
        .input_valid    (input_valid   ),
        .input_value    (input_value   ),

        //lcd 触摸屏相关接口，不需要更改
        .lcd_rst        (lcd_rst       ),
        .lcd_cs         (lcd_cs        ),
        .lcd_rs         (lcd_rs        ),
        .lcd_wr         (lcd_wr        ),
        .lcd_rd         (lcd_rd        ),
        .lcd_data_io    (lcd_data_io   ),
        .lcd_bl_ctr     (lcd_bl_ctr    ),
        .ct_int         (ct_int        ),
        .ct_sda         (ct_sda        ),
        .ct_scl         (ct_scl        ),
        .ct_rstn        (ct_rstn       )
    );
```

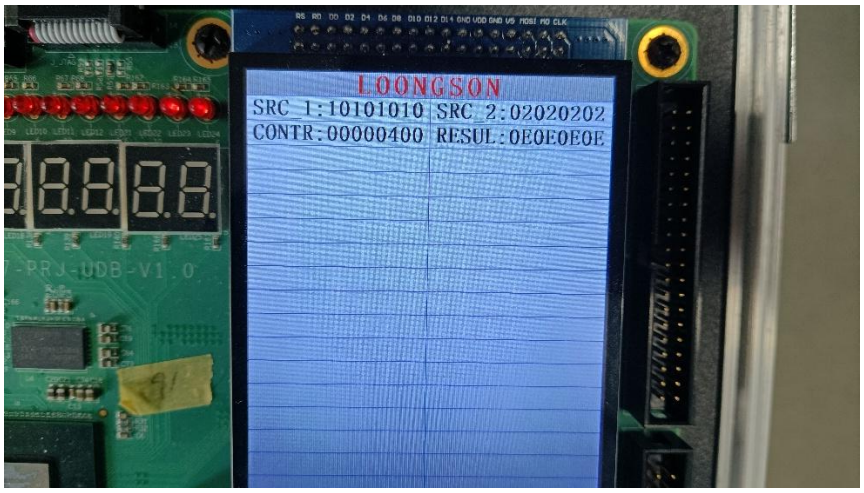
```
//-----{实例化触摸屏}end

//-----{从触摸屏获取输入}begin
//根据实际需要输入的数修改此小节，
//建议对每一个数的输入，编写单独一个 always 块
    //当 input_sel 为 00 时，表示输入数控制信号，即 alu_control
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            alu_control <= 12'd0;
        end
        else if (input_valid && input_sel==2'b00)
        begin
            alu_control <= input_value[3:0];
        end
    end

    //当 input_sel 为 10 时，表示输入数为源操作数 1，即 alu_src1
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            alu_src1 <= 32'd0;
        end
        else if (input_valid && input_sel==2'b10)
        begin
            alu_src1 <= input_value;
        end
    end

    //当 input_sel 为 11 时，表示输入数为源操作数 2，即 alu_src2
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            alu_src2 <= 32'd0;
        end
        else if (input_valid && input_sel==2'b11)
        begin
            alu_src2 <= input_value;
        end
    end
//-----{从触摸屏获取输入}end
```

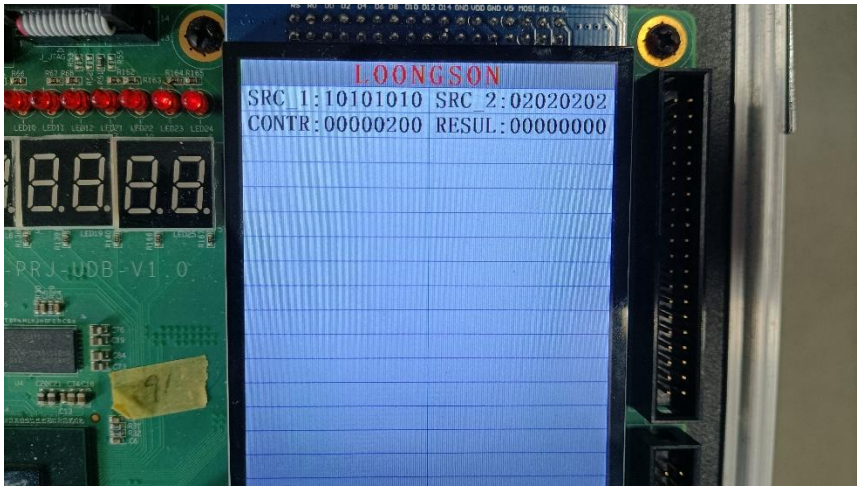
```
//-----{输出到触摸屏显示}begin
//根据需要显示的数修改此小节，
//触摸屏上共有 44 块显示区域，可显示 44 组 32 位数据
//44 块显示区域从 1 开始编号，编号为 1~44，
    always @(posedge clk)
    begin
        case(display_number)
            6'd1 :
            begin
                display_valid <= 1'b1;
                display_name  <= "SRC_1";
                display_value <= alu_src1;
            end
            6'd2 :
            begin
                display_valid <= 1'b1;
                display_name  <= "SRC_2";
                display_value <= alu_src2;
            end
            6'd3 :
            begin
                display_valid <= 1'b1;
                display_name  <= "CONTR";
                display_value <={20'd0, alu_control};
            end
            6'd4 :
            begin
                display_valid <= 1'b1;
                display_name  <= "RESUL";
                display_value <= alu_result;
            end
            default :
            begin
                display_valid <= 1'b0;
                display_name  <= 40'd0;
                display_value <= 32'd0;
            end
        endcase
    end
//-----{输出到触摸屏显示}end
//----------------------{调用触摸屏模块}end--------------------//
endmodule
```

实验结果：

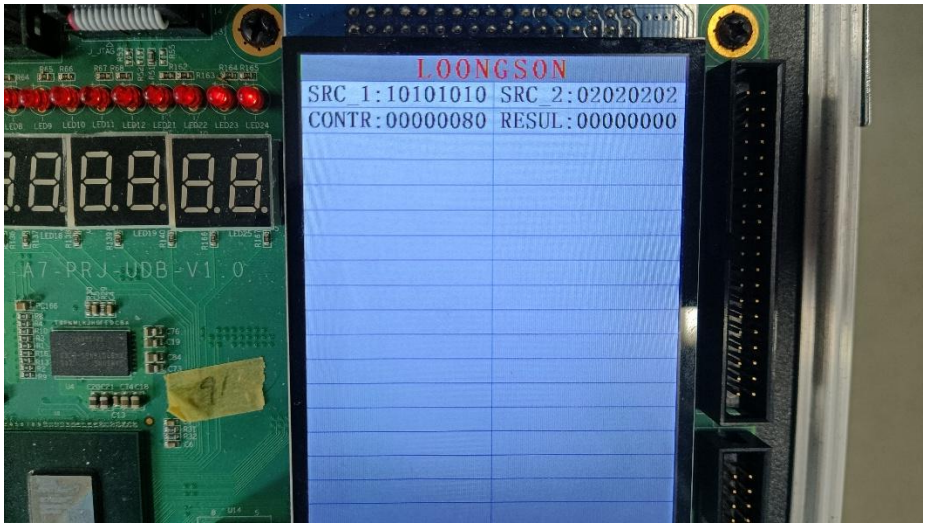

LOONGSON
SRC_1:10101010 SRC_2:02020202
CONTR:00000000 RESUL:00000000


LOONGSON
SRC_1:10101010 SRC_2:02020202
CONTR:00000800 RESUL:12121212

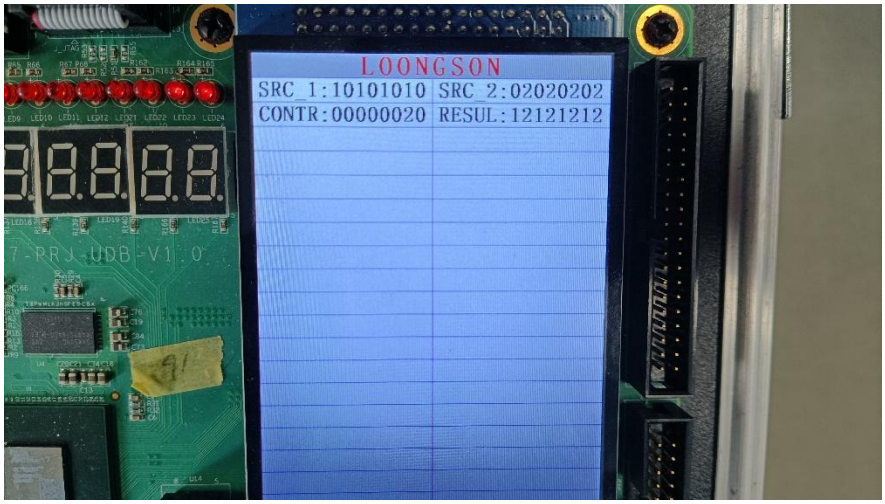

LOONGSON
SRC_1:10101010 SRC_2:02020202
CONTR:00000400 RESUL:0E0E0E0E

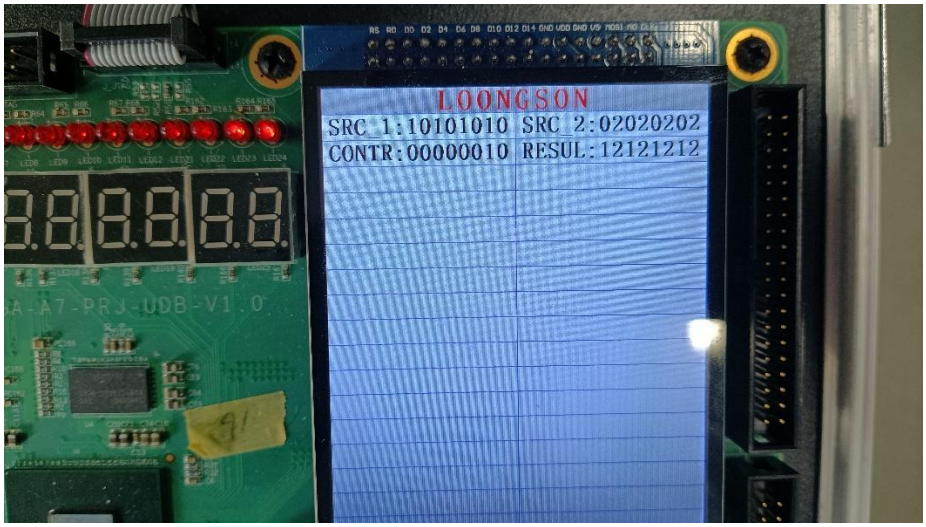

LOONGSON
SRC_1:10101010 SRC_2:02020202
CONTR:00000200 RESUL:00000000

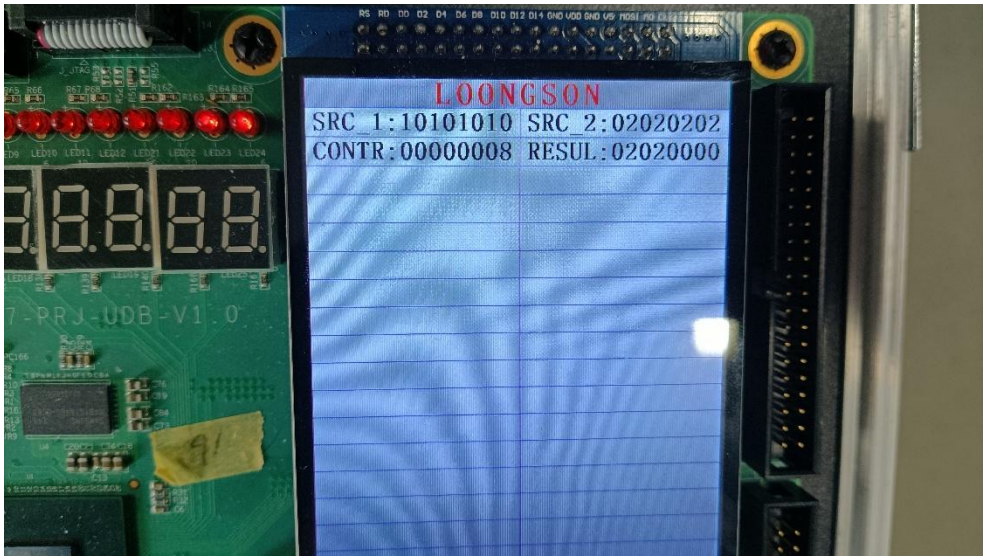

LOONGSON
SRC_1:10101010 SRC_2:02020202
CONTR:00000100 RESUL:00000000

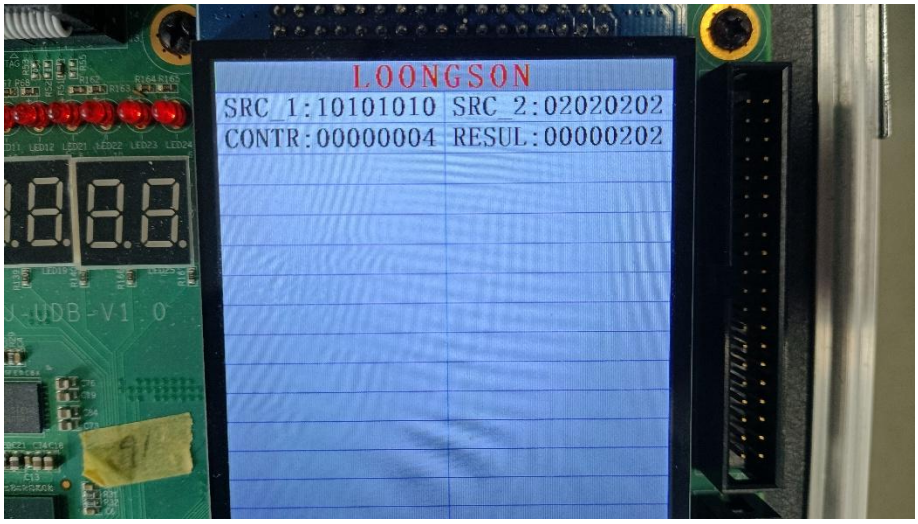

LOONGSON
SRC_1:10101010 SRC_2:02020202
CONTR:00000080 RESUL:00000000

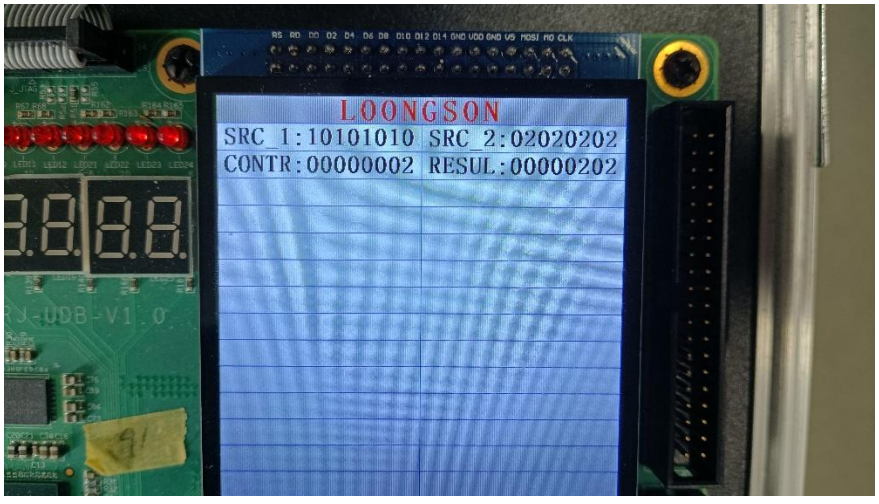

LOONGSON
SRC 1:10101010 SRC 2:02020202
CONTR:00000040 RESUL:EDEDEDED

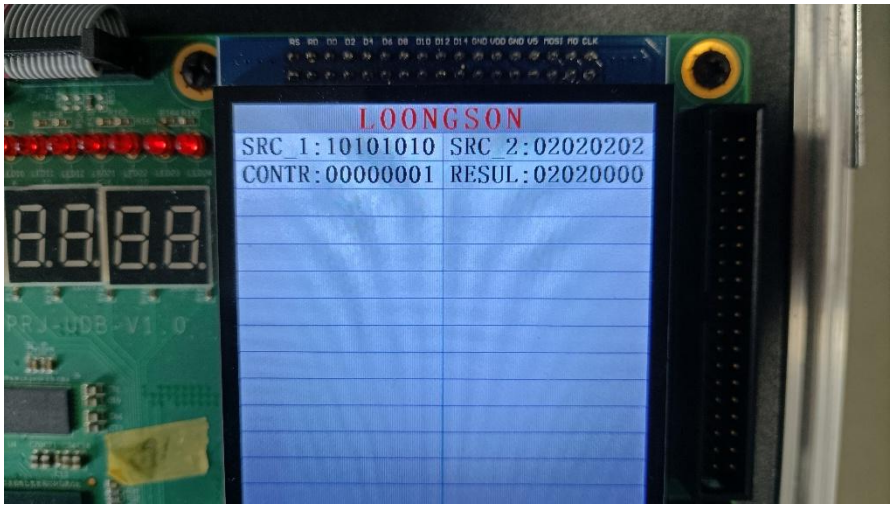

LOONGSON
SRC 1:10101010 SRC 2:02020202
CONTR:00000020 RESUL:12121212


LOONGSON
SRC 1:10101010 SRC 2:02020202
CONTR:00000010 RESUL:12121212


LOONGSON
SRC_1:10101010 SRC_2:02020202
CONTR:00000008 RESUL:02020000


LOONGSON
SRC_1:10101010 SRC_2:02020202
CONTR:00000004 RESUL:00000202


LOONGSON
SRC_1:10101010 SRC_2:02020202
CONTR:00000002 RESUL:00000202

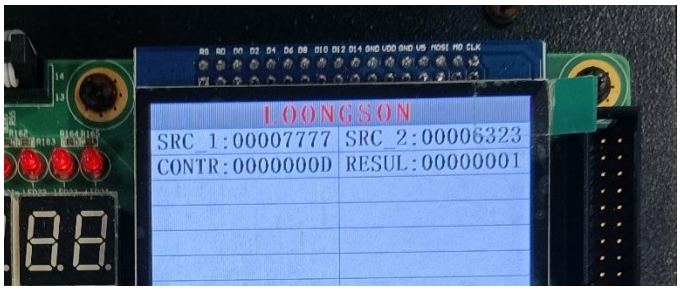


上述为前 13 种运算的输出结果

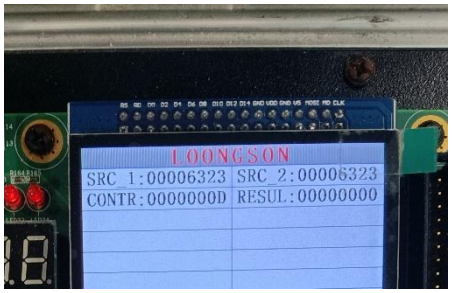
下面为新增加的运算的输出结果

首先是有符号运算，大于置位

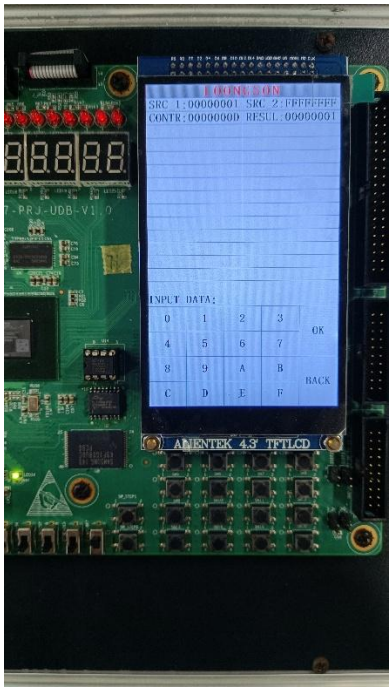
下面为两个正数的比较,由 SRC_1>SRC_2,则 result 为 1



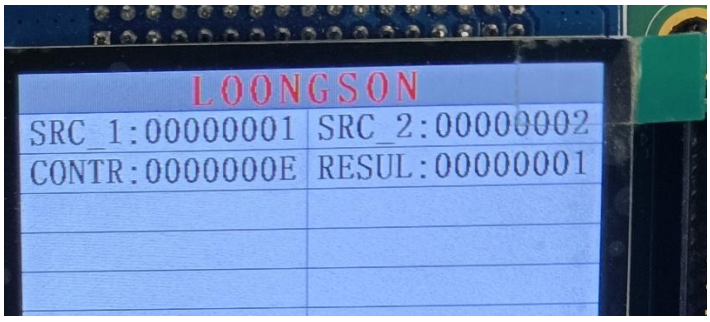
下面为相等的两个数的比较，由 SRC_1=SRC_2,则 result 为 0



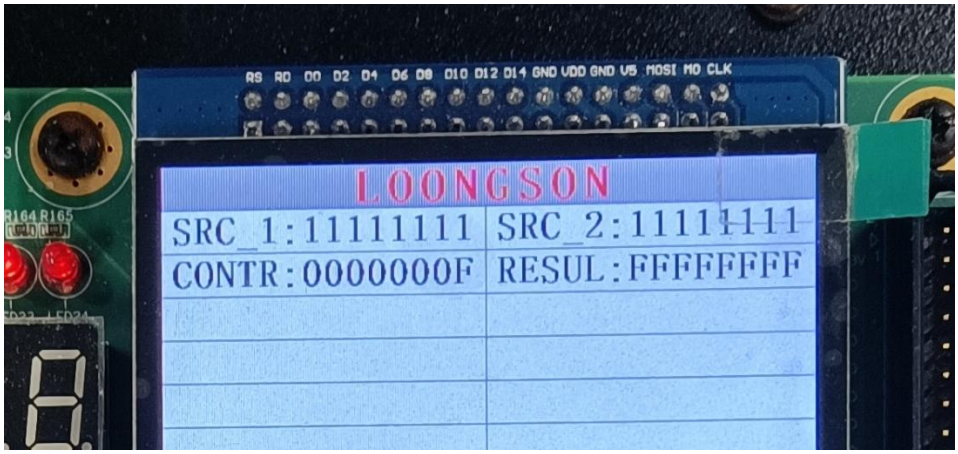
下面为两个负数的比较：由 SRC_1>SRC_2,则 result 为 1



下面为无符号数的比较：由 SRC_1<SRC_2,则 result 为 1



下面为按位同或的结果：SRC_1 与 SRC_2 按位同或，结果如图


RS RD DO D2 D4 D6 D8 D10 D12 D14 GND VDD GND V5 MOSI MO CLK
LOONGSON
SRC_1:11111111
SRC_2:11111111
CONTR:0000000F
RESUL:FFFFFFFF

总结感想：

（1）算术逻辑单元（ALU）是计算机的核心模块之一，用于执行算术运算（如加减法）和逻辑运算（如与、或、非等）。ALU 是处理器（CPU）中的一个基本组成部分，几乎所有的计算操作最终都会通过 ALU 实现。

（2）ALU 是一种组合逻辑电路，可以执行两类操作：

算术运算：包括加法、减法、乘法、除法、加减法结合的移位操作等。

逻辑运算：包括按位的与（AND）、或（OR）、异或（XOR）、非（NOT）、比较（如是否相等、大于、小于）等操作

（3）ALU 的输入：操作数（通常是两个）。控制信号（决定执行的运算类型）。

ALU 的输出：运算结果。状态标志（如进位标志、溢出标志、零标志、符号标志等）。

（4）ALU 是一种组合逻辑电路，其核心组件包括以下部分

算术运算器：加法器，由全加器（Full Adder）组成，用于执行加法；减法器，通过将一个数的补码与另一个数相加实现减法。

逻辑运算器：包含一组简单的逻辑门（如与门、或门、异或门、非门）来执行按位逻辑运算。

多路选择器（Multiplexer）：用于根据控制信号选择算术运算或逻辑运算的结果。

（4）状态寄存器

存储 ALU 运算后的状态标志位，包括：

进位标志（Carry Flag, CF）：加法或减法运算中是否产生进位或借位。

零标志（Zero Flag, ZF）：结果是否为零。

溢出标志（Overflow Flag, OF）：运算结果是否超出范围。

符号标志（Sign Flag, SF）：结果是否为负。

（5）ALU 的性能主要由以下指标决定：

运算速度：ALU 的速度受限于逻辑门的延迟。高速加法器（如超前进位加法器）可以显著提高 ALU 的性能。

位宽：位宽决定了 ALU 一次可以处理的操作数长度。常见的位宽包括 8 位、16 位、32 位和 64 位。

功能多样性：ALU 支持的运算种类越多，其复杂性也越高。

功耗：特别是在嵌入式系统中，低功耗 ALU 非常重要。