

计算机组成原理第四次实验报告

实验名称：ALU模块实现 班级：张金老师 姓名：王众 学号：2313211

指导老师：董前琨 实验地点：A306 实验时间：2025.4.2

一、实验目的

- 熟悉指令集中的运算指令，学会对这些指令进行归纳分类。
- 了解指令结构。
- 熟悉并掌握的原理、功能和设计。
- 进一步加强运用语言进行电路设计的能力。
- 为后续设计的实验打下基础。

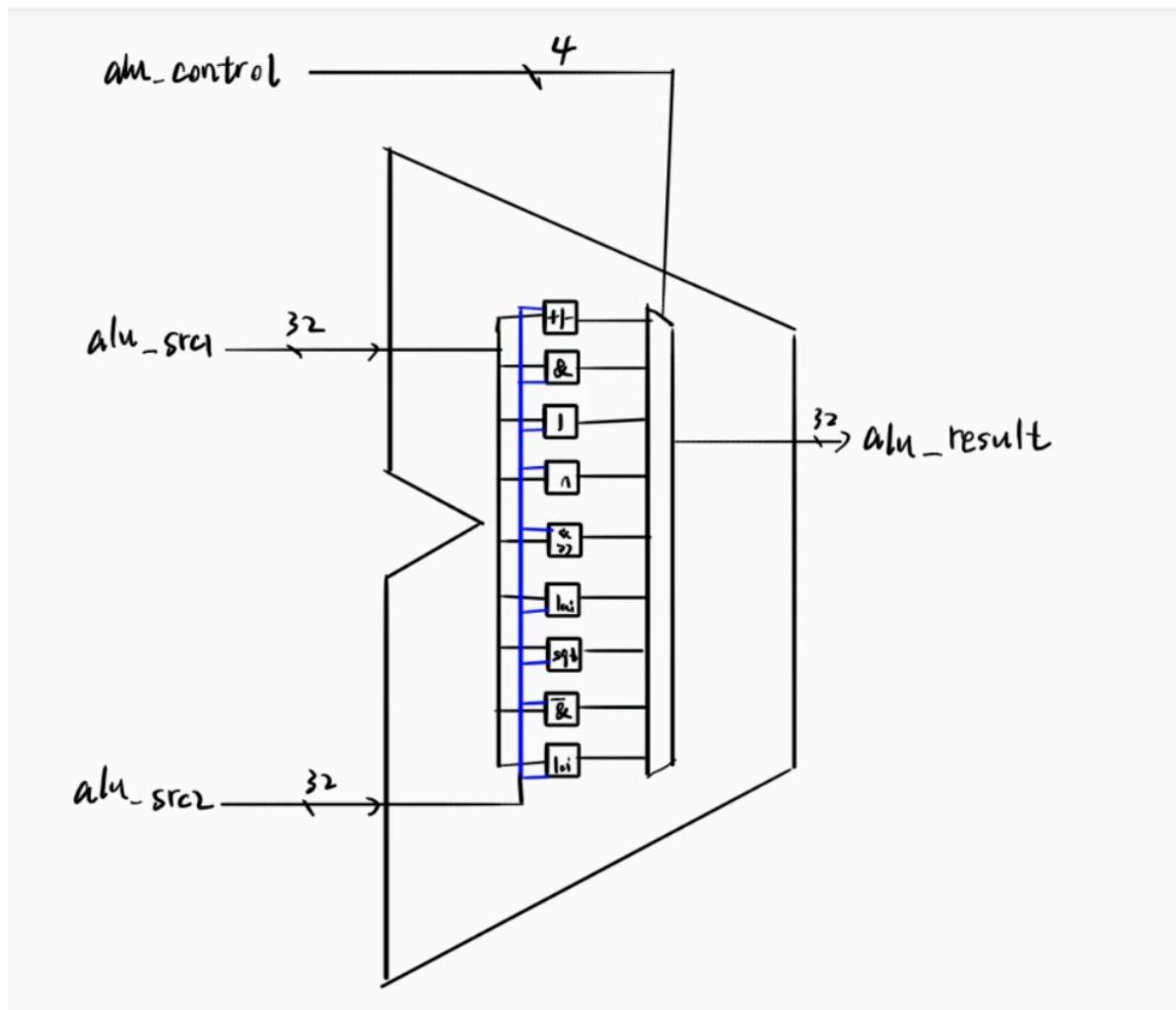
二、实验内容说明

请结合实验指导手册中的实验四（模块实现实验）完成功能改进，实现一个能够完成更多运算的，注意以下几点：

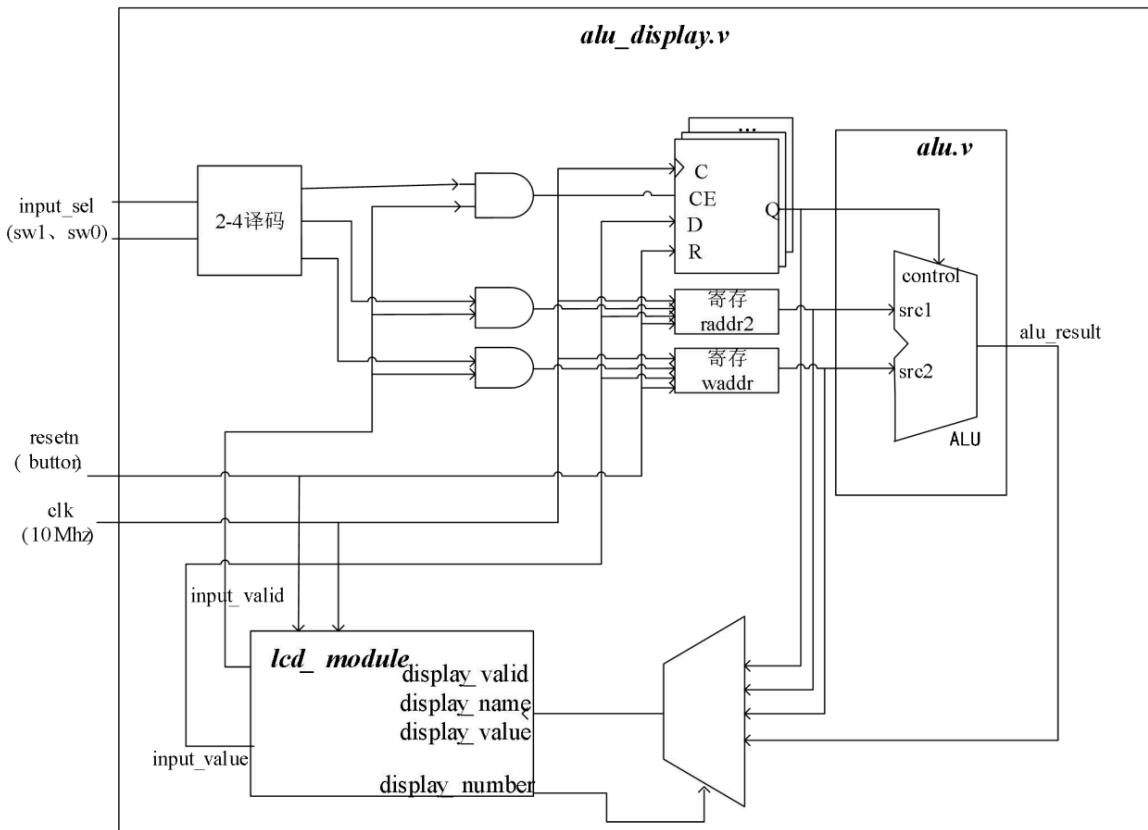
- 原始代码中只有表中的11种运算，请另外补充至少三种不同类型运算（比较运算、位运算、数据加载运算等）。
- 实验报告中原理图参考5.3，只画出补充的运算部分即可，报告中展示的结果应补充运算的计算结果。
- 本次实验报告不需要有仿真结果，但需要有实验箱上箱验证的照片，同样，针对照片中的数据需要解释说明。

三、实验原理图和实验原理

3.1 ALU模块原理图



3.2 顶层模块原理图



3.3 实验原理

3.3.1 ALU运算重排

首先，由于14种控制状态超出了控制板的数量限制，所以我们采用二进制数方式来对运算方式进行标识。从而将15位的输入变为4位的输入控制位。前面的11种运算(包括不计算)，我们不改变其顺序，还是按照0-12排，也就是十六进制的0-C。后面的D,E,F，我们按照顺序依次设置为**有符号比较**，大于置位；按位同或；低位加载，如下表所示

NUMBER	ALU操作
0	不操作
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	按位同或
F	低位加载

3.3.2 原始11种运算方式

3.3.2.1 加法&减法&有符号比较，小于置位&无符号比较，小于置位

对于加减法我们调用实验一中的 `adder.v` 模块就可以完成运算。

对于有符号比较的小于置位，根据pdf的规则提示我们可以推出：

```
assign s1t_result[31:1] = 31'd0;
assign s1t_result[0]    = (alu_src1[31] & ~alu_src2[31]) | (~alu_src1[31]^alu_src2[31]) & adder_result[31];
```

对于32位无符号的比较小于置位，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。对比表可知，对于正数的比较，只要减法结果的符号位为1，则表示小于。而位正数相减，其结果的符号位最终可由位加法的 `cout+1'b1` 得到，如图5.5。故无符号位比较小于置位运算结果表达式为：`sltu_result = ~adder_cout`。

```
assign sltu_result = {31'd0, ~adder_cout};
```

3.3.2.2按位与&按位或非&按位或&按位异或&高位加载

```
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
```

我们使用一些简单的逻辑运算就能实现其中四种运算法则，如上所示。

对于高位加载：专门用于设置寄存器中常数的高16位置。将低16位存放到寄存器的高16位，结果的低16位用0填充。

3.3.2.3逻辑左移&逻辑右移&算数左移

```
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
```

由于总体的代码过长且思路相似，所以我们选择逻辑左移来作为例子：主要分为3个步骤：

第一步根据移位量低位，即位，做第一次移位；

第二步在第一次移位基础上根据移位量 位做第二次移位；

第三步在第二次移位基础上根据移位量 位做第三次移位。

3.3.3 新增运算原理

3.3.3.1 有符号比较，大于置位

和小于置位是基本相同的，我们只需要将最后的逻辑取反即可完成任务。但是考虑到跟换之后会出现等于的情况，我们需要新增一个 `adder_zero` 值，用于排除相等的情况，这样就可以处理相等的情况了。

其中，`adder_zero` 是对结果 `adder_result` 进行或缩位运算，当源操作数 = 源操作数时，`adder_zero` 为 1，不置位。因此，大于置位的实现是：`~(小于置位表达式) & adder_zero`，即不满足小于置位且两操作数不相等的就是大于置位。

3.3.3.2 按位同或

将我们前面的按位异或的取反，我们只需要在我们前面的代码上加上一个取反符号就可以了！

```
assign nxor_result = ~xor_result; // 同或结果为异或取反
```

3.3.3.3 低位加载

和高位加载也是一样的，只不过我们选取的位数从高位变成了低位，代码如下所示。

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

四、实验步骤

4.1 复现

首先观察一下我们的程序与硬件对应的部分

```
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)
```

我们发现我们通过修改我们的两个按钮就可以控制我们的写入，我们在约束文件中找到对应的引脚，发现是第二个按钮和第八个按钮，然后我们就可以开始复现原来的项目了

4.2改进

4.2.1代码部分修改

4.2.1.1 alu.v 模块的修改

首先，我们对我们的运算的控制信号位数进行了修改，原先需要4个二进制来确定我们的操作，现在我们只需要3位二进制了，所以我们将位宽修改为[0:3]

```
input [3:0] alu_control,
```

第二步我们便需要新添加我们的三个功能，所以我们需要新增三个wire，用来分别表示我们的三个新运算。

```
wire [31:0] sgt_result;
wire [31:0] nxor_result;
wire [31:0] hui_result;
```

既然修改了控制信号，我们便需要将16中运算方式重新进行绑定：

```
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])&
    (~alu_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_control[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_sr1 = (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_control[0]); //1011
assign alu_lui = (alu_control[3])&(alu_control[2])&(~alu_control[1])&
    (~alu_control[0]); //1100
//补充-----
assign alu_sgt = (alu_control[3])&(alu_control[2])&(~alu_control[1])&
    (alu_control[0]); //1101
assign alu_nxor = (alu_control[3])&(alu_control[2])&(alu_control[1])&
    (~alu_control[0]); //1110
assign alu_hui = (alu_control[3])&(alu_control[2])&(alu_control[1])&
    (alu_control[0]); //1111
```

然后进行的是三个新模块的修改操作，已经说明，不再赘述。

最后一步，我们还需要将三个函数的输出结果存储在我们的 alu_result 中：

```
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 ://有符号数比较，大于置位  
    alu_nxor ? nxor_result ://按位同或  
    alu_hui ? hui_result ://低位加载  
    32'd0;
```

4.2.1.2 alu_display.v

首先将寄存器的位宽从12位修改为4位：

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

然后对应的，我们在输出中也需要对应的进行修改

```
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
```

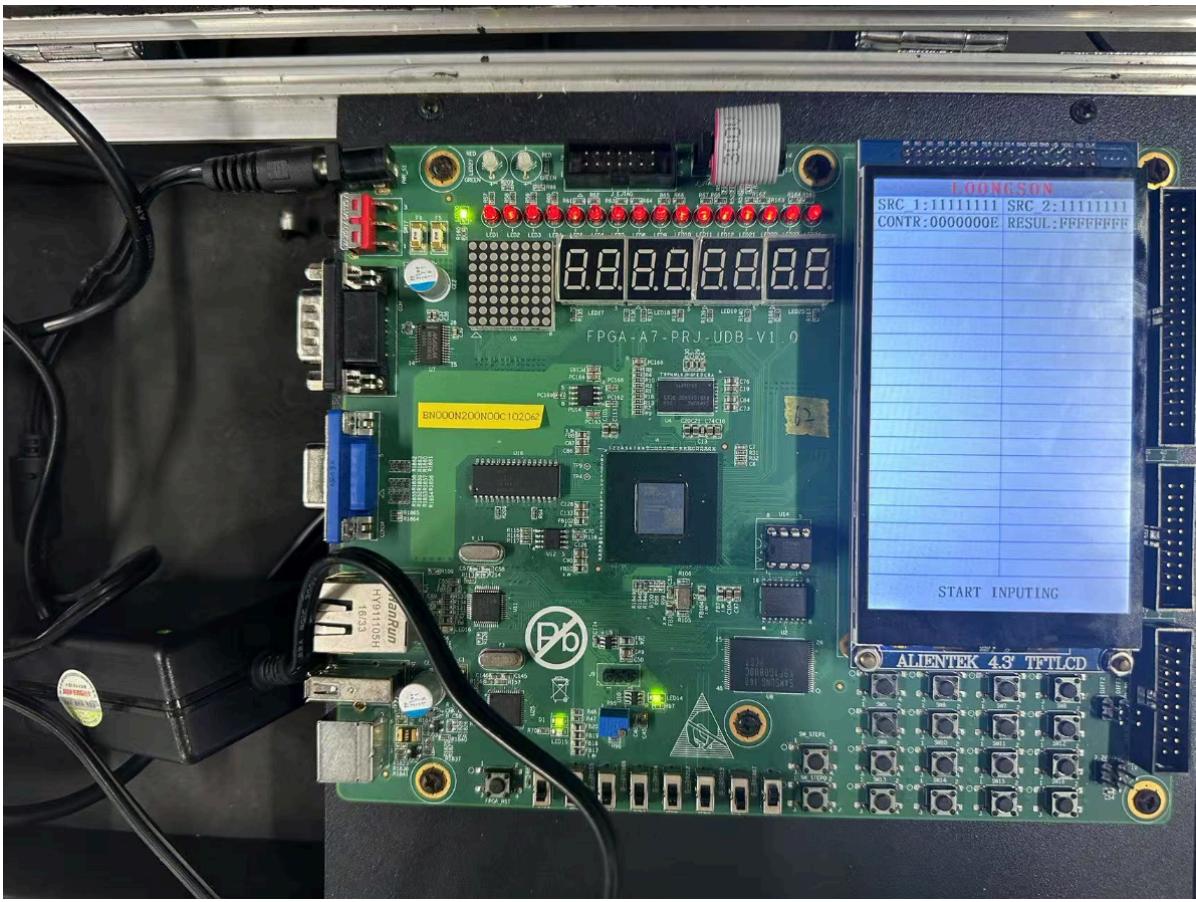
五、实验结果分析

5.1 复现

CONTR就是置位选择运算类型

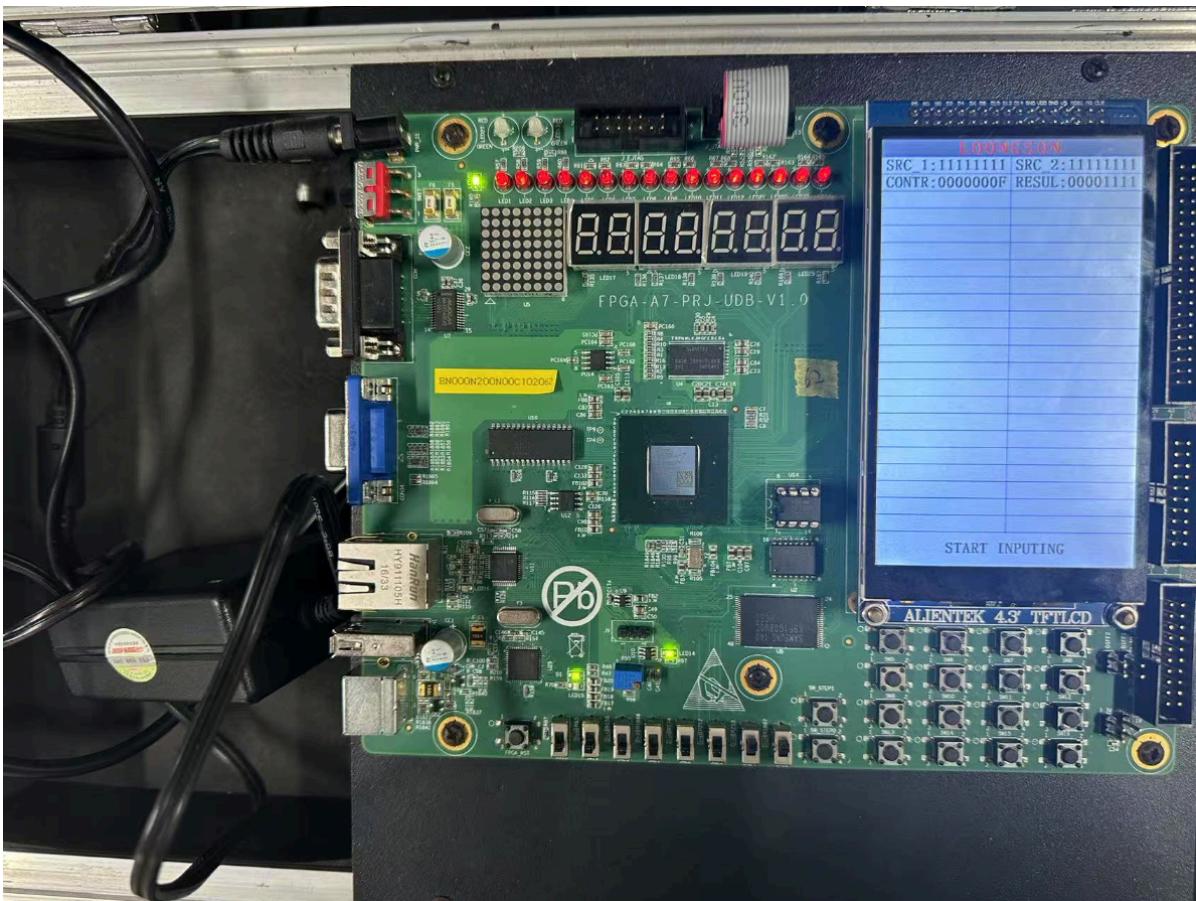
我们首先先验证一下我们原来的那些运算的正确性，此处我们不改变两个输入数的值，只改变了中间的CONTR值，也就是将我们的位数做了改变。

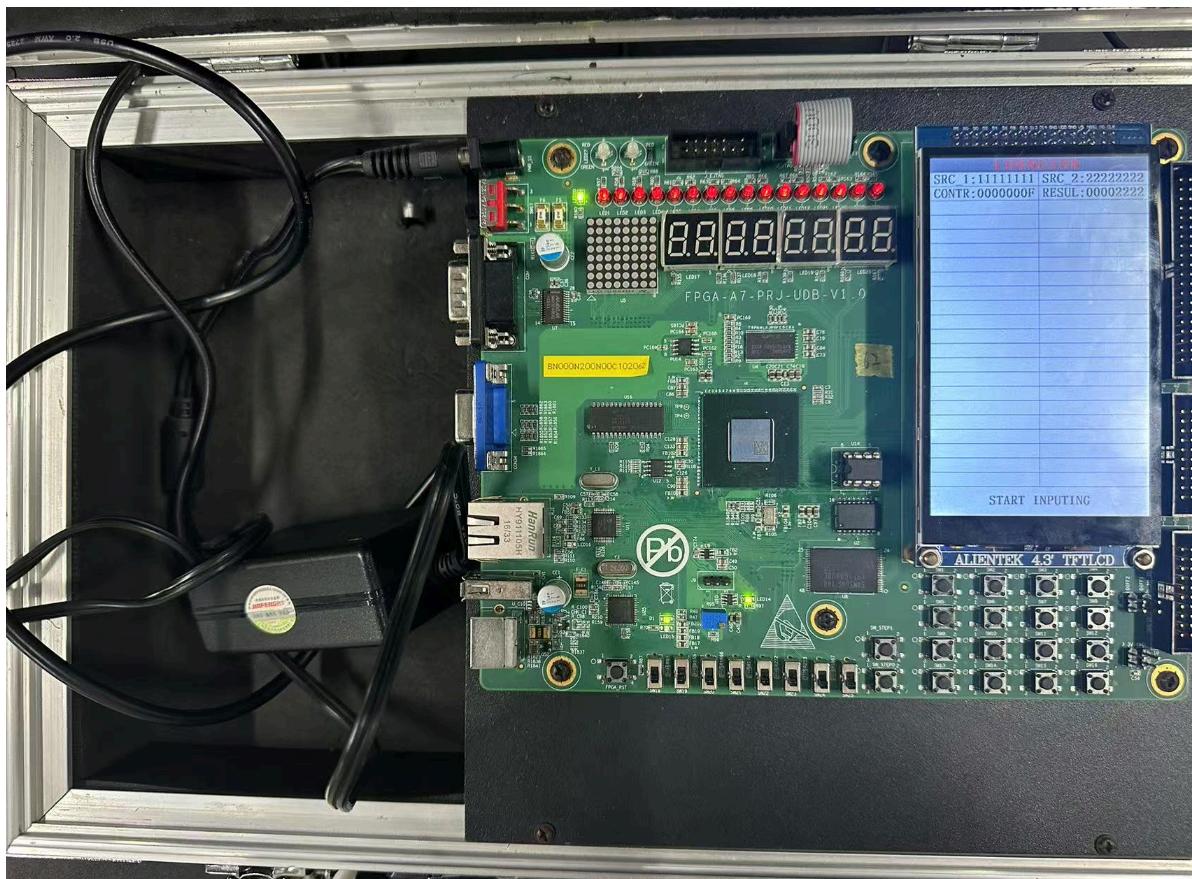
(1) 首先是同或运算



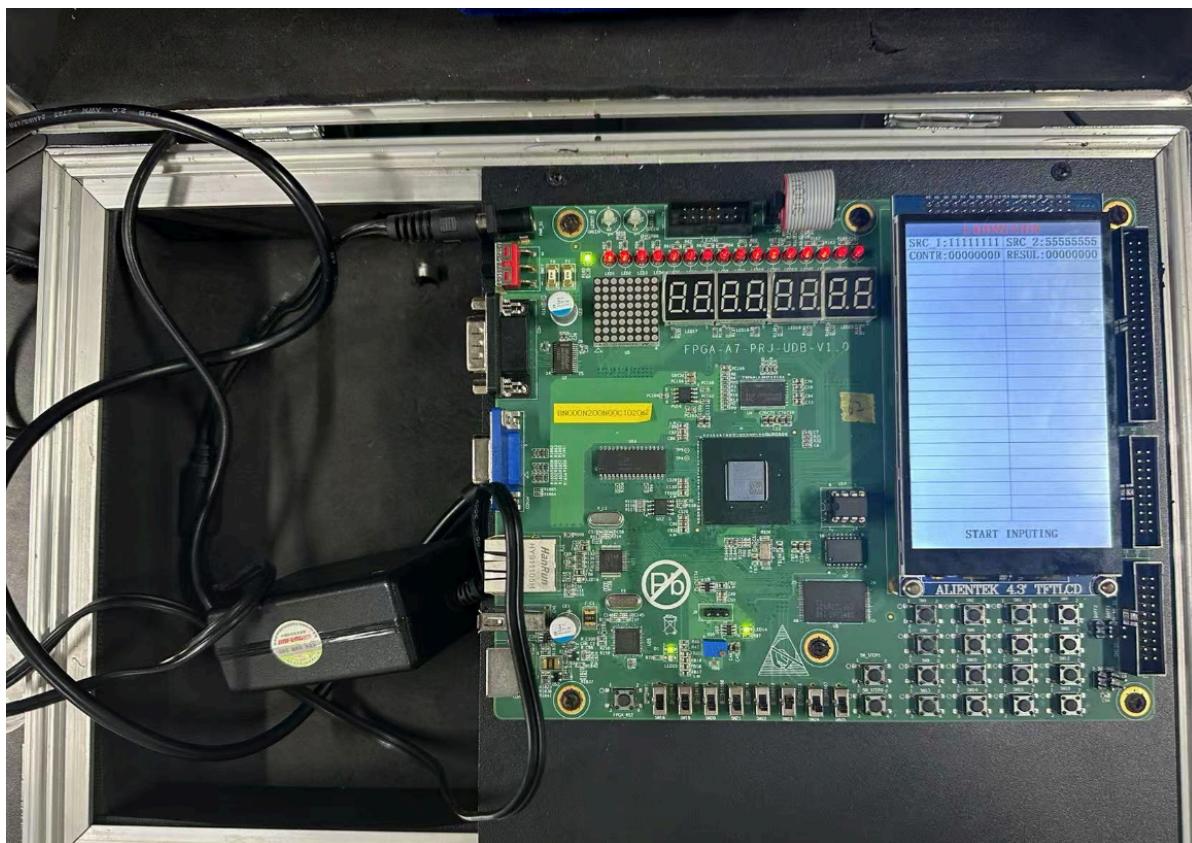
11111111 和 11111111 同或的结果为 FFFFFFFF。符合我们的预期

(2) 低位加载

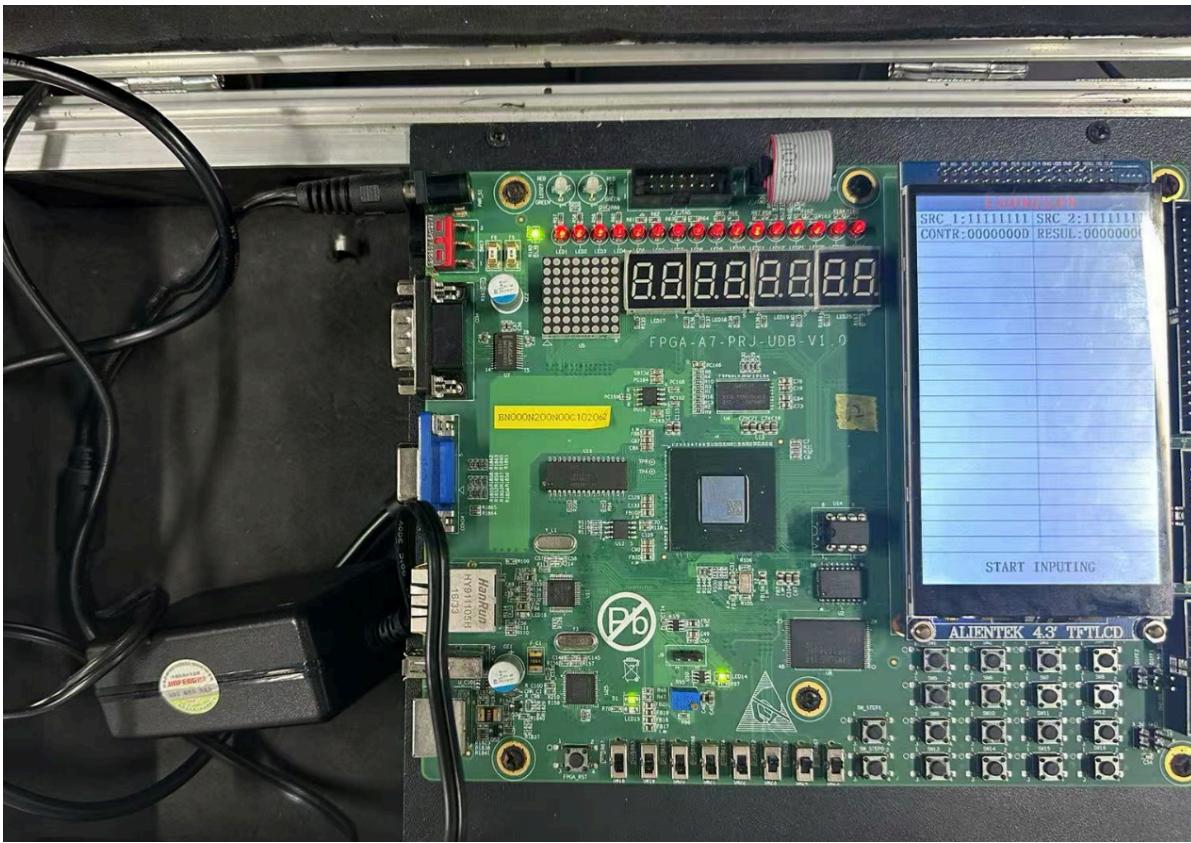




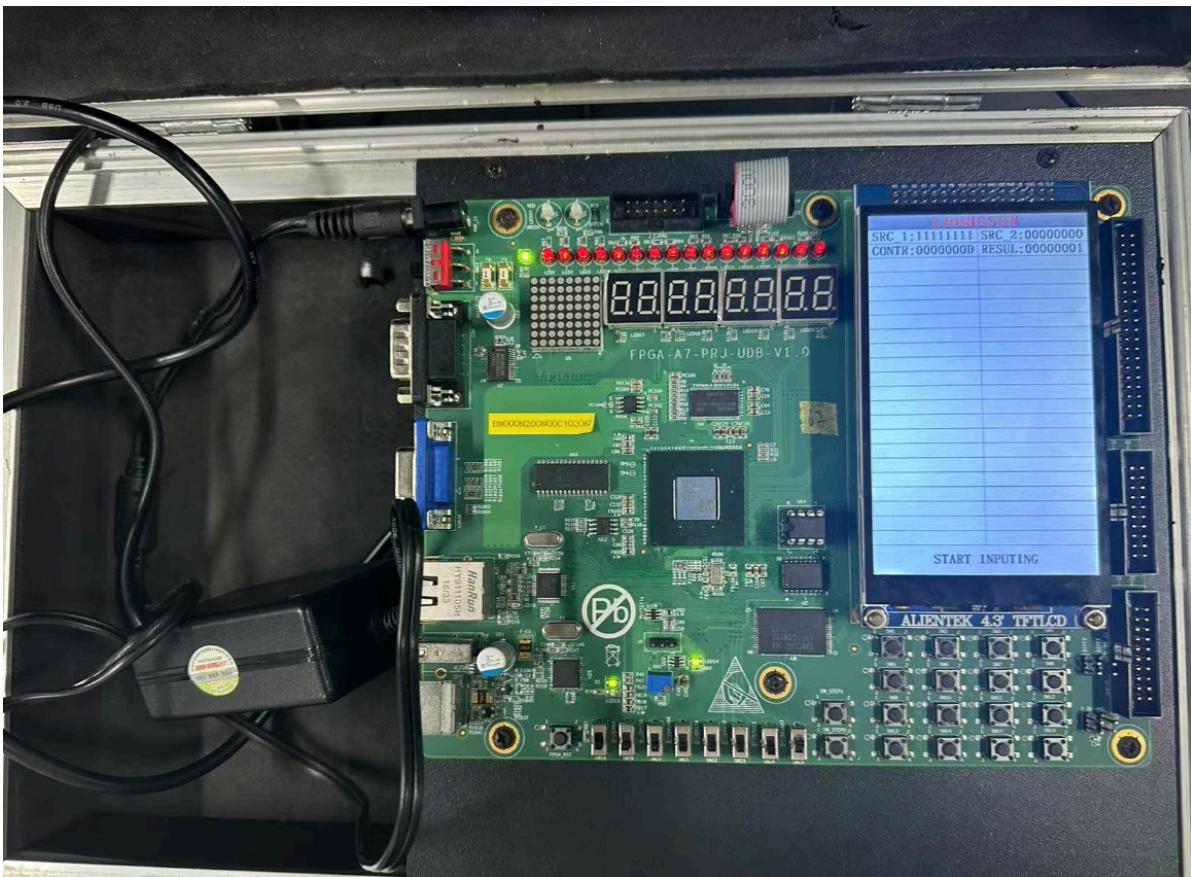
(3) 大于零则置位



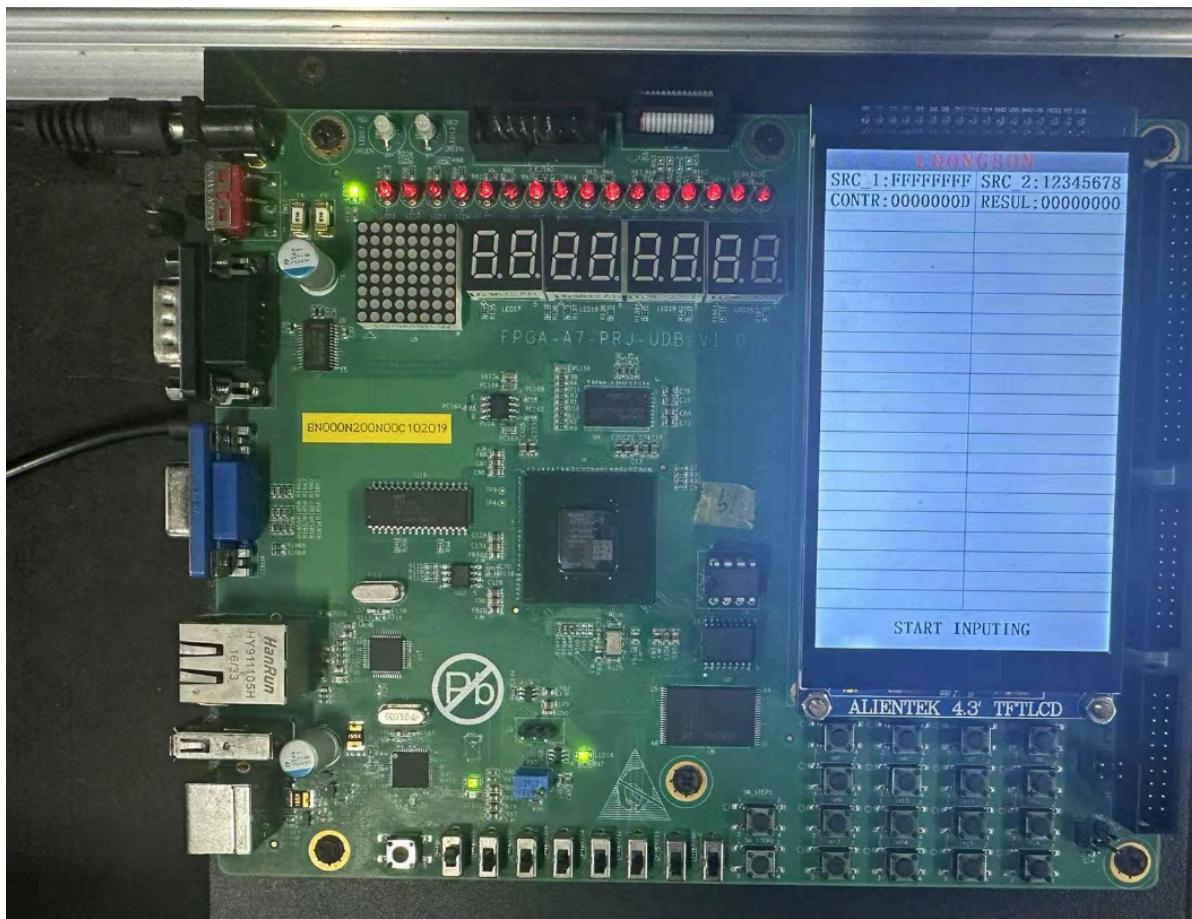
11111111<55555555则不置位



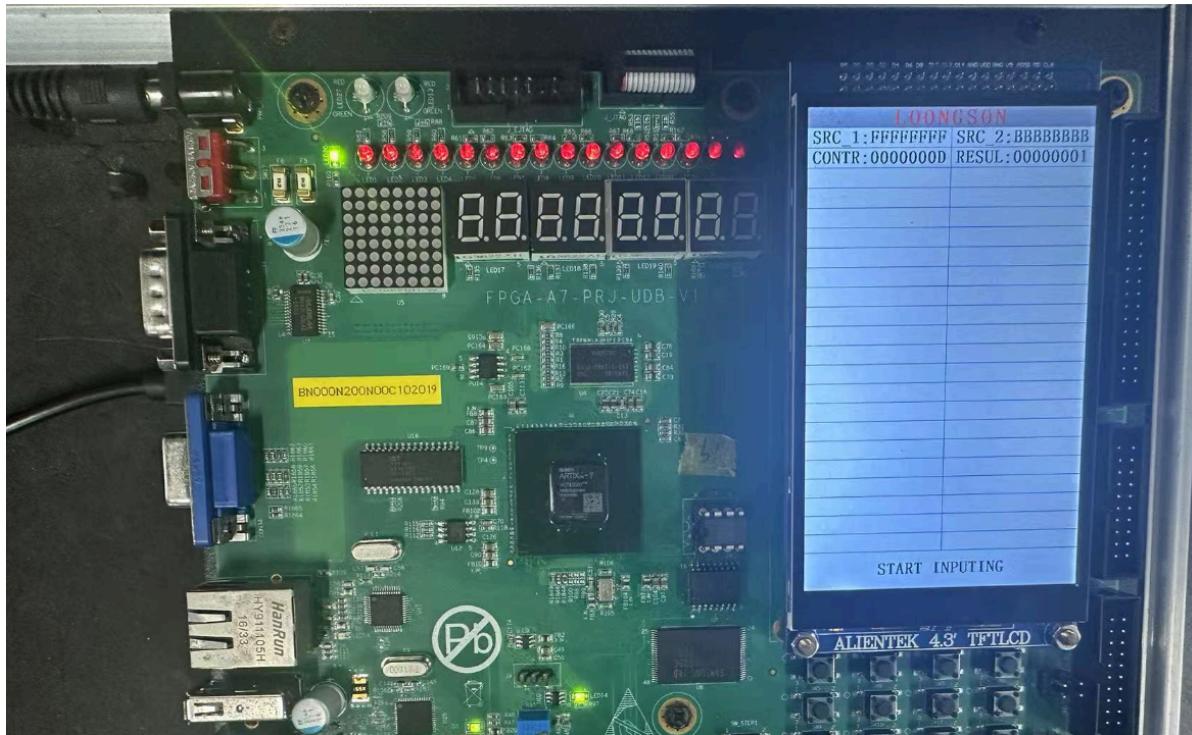
11111111=11111111等于的时候也不置位



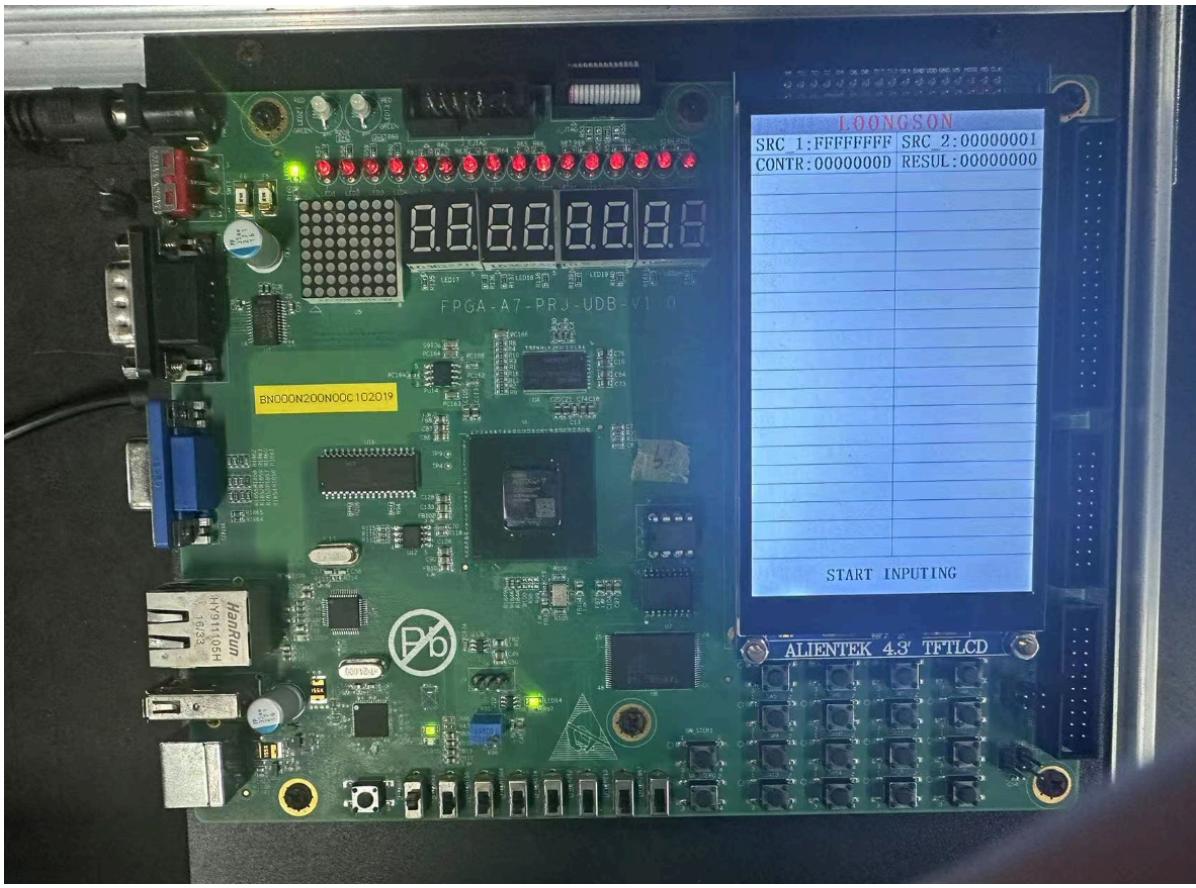
11111111>00000000大于的时候置位



-1<12345678小于则不置位



负数之间的比较，大于置位



-1<1不置位

我们可以看到小于等于的时候不置位，只有在大于的时候才变为1。

六、总结感想

本次实验，复现难度不大，就是把代码放到一起跑一下而已，但是对于代码的修改与增加功能，需要深入了解此次实验的代码构成以及前面的实验课的内容，都需要融会贯通，才能很快地完成本次实验。通过本次实验，更加理解了整个程序的流程以及实验箱的结构，进一步熟悉了代码的编写；掌握了模块的实现方式，以及不同运算的实现过程，为之后设计 打好了基础。