南開大學
Nankai University

计算机学院

计算机组成原理实验报告

实验四

姓名：林盛森

学号：2312631

专业：计算机科学与技术

2025 年 5 月 7 日

目录

1 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。

2 实验内容说明

针对组成原理第四次的 ALU 实验进行改进，要求：

1、将原有的操作码进行位压缩，调整操作码控制信号位宽为 4 位。

2、操作码调整成 4 位之后，在原有 13 种运算的基础之上，补充 3 种不同类型的运算（要求一种大于置位比较，一种位运算，一种自选），需要上实验箱或仿真验证计算结果。

3、注意改进实验上实验箱验证时，操作码应该已经压缩到 4 位位宽。

3 实验原理图

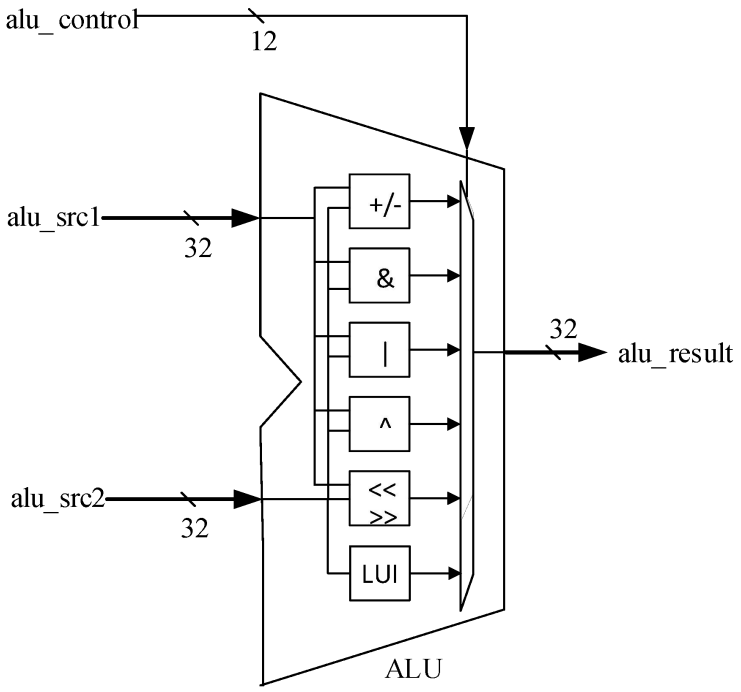

图 3.1: ALU 原理图

4 实验步骤

4.1 实验设计

根据实验要求，我们需要调整操作码控制信号位宽为 4 位，原有的 13 种运算再加上我们添加的 3 种运算一共是 16 种运算，正好可以用 4bits 来控制去选择不同运算。我们可以设计如下的运算，并且

新添了 3 条运算，分别是有符号数大于置位比较，按位取反，低位加载，对应关系如下：

ALU 运算	控制信号
无	0
加法	1
减法	2
有符号比较，小于置位	3
无符号比较，小于置位	4
按位与	5
按位或非	6
按位或	7
按位异或	8
逻辑左移	9
逻辑右移	A
算数右移	B
高位加载	C
有符号比较，大于置位	D
按位取反	E
低位加载	F

表 1: 对应关系

4.2 ALU.v

4.2.1 操作码压缩

首先需要将 ALU 控制信号改为 4 位，共可实现 16 种组合，对应 16 种运算。

```
input  [3:0] alu_control,  // ALU控制信号  改为4位控制
```

新增 3 个 wire 型变量去接收控制信号。

```
    wire alu_sgt;   //有符号比较，大于置位
    wire alu_not;   //按位取反
    wire alu_hui;   //低位加载
```

在此处利用之前定义的运算控制信号去接收我们输入的控制信号。

```
    assign alu_add  = (alu_control == 4'b0001);
    assign alu_sub  = (alu_control == 4'b0010);
    assign alu_slt  = (alu_control == 4'b0011);
    assign alu_sltu = (alu_control == 4'b0100);
    assign alu_and  = (alu_control == 4'b0101);
    assign alu_nor  = (alu_control == 4'b0110);
    assign alu_or   = (alu_control == 4'b0111);
    assign alu_xor  = (alu_control == 4'b1000);
    assign alu_sll  = (alu_control == 4'b1001);
    assign alu_srl  = (alu_control == 4'b1010);
    assign alu_sra  = (alu_control == 4'b1011);
    assign alu_lui  = (alu_control == 4'b1100);
    assign alu_sgt  = (alu_control == 4'b1101);
```

```
    assign alu_not  = (alu_control == 4'b1110);
    assign alu_hui  = (alu_control == 4'b1111);
```

新增 3 个 32 位 wire 型变量去接收运算结果。

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

4.2.2 有符号数大于置位比较

对于有符号数大于置位比较操作来说，类似于小于置位，所以我们对小于置位的逻辑取反即可，但是要注意的是，我们要去判断两个数是否相等，排除相等的情况，assign not_equal = (adder_result != 0) 这句去判断是否相等，并在结果中与上不等。

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

4.2.3 按位取反

按位取反操作只需使用 运算符取反即可。

```
assign not_result = ~alu_src2;                  //按位取反
```

4.2.4 低位加载

低位加载类似于高位加载，我们需要将传进来的 32 位立即数的低 16 位加载到目标操作数的低 16 位上，只需利用传进来的立即数的低 16 位去填充目标操作数的低 16 位，并让目标操作数的高 16 位全填为 0。

```
assign hui_result = {16'd0, alu_src2[15:0]};  //立即数装载结果为立即数移位至低半字节
```

在最后，需要条件控制去选择不同运算的对应结果。

```
    // 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_not          ? not_result :
                        alu_hui          ? hui_result :
                        32'd0;
```

4.3 ALU_display.v

在 display 文件中，我们也要作相应的修改，一处是定义的控制信号应该改为 4 位。

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

另一处把输入的传给控制信号的变量也改为 4 位。

```
    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
```

4.4 ALU_tb.v

我们添加了仿真文件以更好地去验证代码的正确性。

```
timescale 1ns / 1ps

module alu_tb;
    reg [3:0]  alu_control;
    reg [31:0] alu_src1;
    reg [31:0] alu_src2;
    wire [31:0] alu_result;

    // 实例化被测 ALU
    alu u_alu(
        .alu_control (alu_control),
        .alu_src1    (alu_src1),
        .alu_src2    (alu_src2),
        .alu_result  (alu_result)
    );

    initial begin
        // 初始化波形记录
```

```
        $dumpfile("alu_wave.vcd");
        $dumpvars(0, alu_tb);

        // 测试用例 1: 加法 (ADD)
        alu_control = 4'b0001;
        alu_src1 = 32'h0000_0005;
        alu_src2 = 32'h0000_0003;
        #10;
        $display("ADD: 5 + 3 = %h (Expected 8)", alu_result);

        // 测试用例 2: 减法 (SUB)
        alu_control = 4'b0010;
        alu_src1 = 32'h0000_0008;
        alu_src2 = 32'h0000_0003;
        #10;
        $display("SUB: 8 - 3 = %h (Expected 5)", alu_result);

        // 测试用例 3: 有符号小于置位 (SLT)
        alu_control = 4'b0011;
        alu_src1 = 32'hFFFF_FFFE; // -2 (补码)
        alu_src2 = 32'h0000_0001; // 1
        #10;
        $display("SLT: -2 < 1 ? %h (Expected 1)", alu_result);

        // 测试用例 4: 无符号小于置位 (SLTU)
        alu_control = 4'b0100;
        alu_src1 = 32'h0000_00FF;
        alu_src2 = 32'h0000_FF00;
        #10;
        $display("SLTU: 255 < 65280 ? %h (Expected 1)", alu_result);

        // 测试用例 5: 按位与 (AND)
        alu_control = 4'b0101;
        alu_src1 = 32'hA5A5_A5A5;
        alu_src2 = 32'h0F0F_0F0F;
        #10;
        $display("AND: A5A5A5A5 & 0F0F0F0F = %h (Expected 05050505)", alu_result);

        // 测试用例 6: 按位或非 (NOR)
        alu_control = 4'b0110;
        alu_src1 = 32'hAA00_00AA;
        alu_src2 = 32'h5500_0055;
        #10;
        $display("NOR: ~(AA0000AA | 55000055) = %h (Expected 00FFFF00)", alu_result);

        // 测试用例 7: 按位或 (OR)
        alu_control = 4'b0111;
        alu_src1 = 32'h1234_5678;
        alu_src2 = 32'h0000_FFFF;
```

```
        #10;
        $display("OR: 12345678 | 0000FFFF = %h (Expected 1234FFFF)", alu_result);

        // 测试用例 8: 按位异或 (XOR)
        alu_control = 4'b1000;
        alu_src1 = 32'hDEAD_BEEF;
        alu_src2 = 32'h1234_5678;
        #10;
        $display("XOR: DEADBEEF ^ 12345678 = %h (Expected CC8FE897)", alu_result);

        // 测试用例 9: 逻辑左移 (SLL)
        alu_control = 4'b1001;
        alu_src1 = 4;       // 移位量 (注意: 根据代码实际使用 alu_src1[4:0])
        alu_src2 = 32'h0000_000F; // 被移位数
        #10;
        $display("SLL: F << 4 = %h (Expected 0000000F)", alu_result);

        // 测试用例 10: 逻辑右移 (SRL)
        alu_control = 4'b1010;
        alu_src1 = 4;       // 移位量
        alu_src2 = 32'hF000_0000;
        #10;
        $display("SRL: F0000000 >> 4 = %h (Expected 0F000000)", alu_result);

        // 测试用例 11: 算术右移 (SRA)
        alu_control = 4'b1011;
        alu_src1 = 4;       // 移位量
        alu_src2 = 32'h8000_0000; // 负数 (最高位为1)
        #10;
        $display("SRA: 80000000 >> 4 = %h (Expected F8000000)", alu_result);

        // 测试用例 12: 高位加载 (LUI)
        alu_control = 4'b1100;
        alu_src2 = 32'h0000_1234; // 低16位
        #10;
        $display("LUI: Load Upper 1234 -> %h (Expected 12340000)", alu_result);

        // 测试用例 13: 有符号大于置位 (SGT)
        alu_control = 4'b1101;
        alu_src1 = 32'h0000_0005; // 5
        alu_src2 = 32'hFFFF_FFFE; // -2
        #10;
        $display("SGT: 5 > -2 ? %h (Expected 1)", alu_result);

        // 测试用例 14: 按位取反 (NOT)
        alu_control = 4'b1110;
        alu_src2 = 32'h0000_00FF;
        #10;
        $display("NOT: ~00FF = %h (Expected FFFFFF00)", alu_result);
```

```
117
118         // 测试用例 15: 低位加载 (HUI)
119         alu_control = 4'b1111;
120         alu_src2 = 32'h1234_5678; // 高16位会被忽略
121         #10;
122         $display("HUI: Load Lower 5678 -> %h (Expected 00005678)", alu_result);
123
124         // 结束仿真
125         #10;
126         $finish;
127     end
128 endmodule
```

5 实验结果分析

5.1 仿真验证

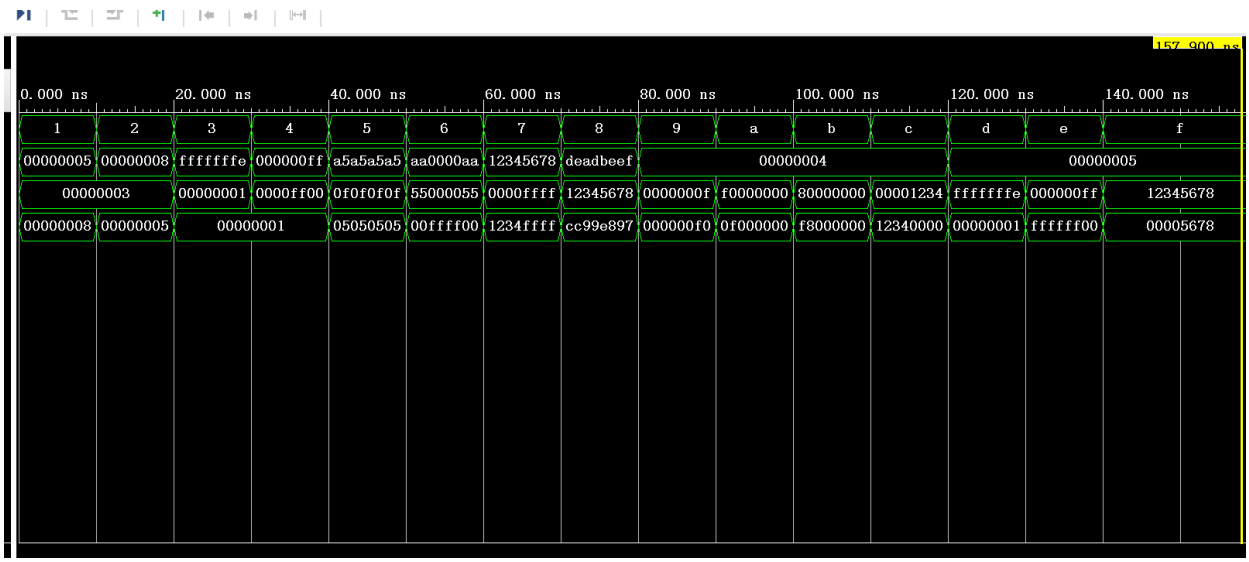

图 5.2: 仿真结果

1：5+3=8 正确

2：8-3=5 正确

3：第一个数是负数，第二个数是正数，小于判断成立，结果为 1，正确

4. 两个无符号数相减，第一个小于第二个，结果为 1，正确

5.a5a5a5a5 于 0f0f0f0f 相与，与 0 相与为 0，与 f 相与为本身，所以得到结果为 05050505，正确

6.aa0000aa 与 55000055 或非运算，a 与 5 或非恰为 0，0 与 0 或非为 1，得到结果为 00ffff00，正确

7. 按位或运算明显正确

8. 按位异或，相同为 0，相异为 1，d 是 1101，1 是 0001，异或结果为 1100，即为 c，另外其他的位运算也相同，得到结果 cc99e897，正确

9. 逻辑左移 1 位，低位补 0，正确

a. 逻辑右移 1 位，高位补 0，正确
b. 算数右移，高位符号位补位，补 1111，即为 f，正确
c. 高位加载，把立即数的低 16 位 1234h 加载到目标操作数的高 16 位，并在低位补 0，正确
d. 有符号大于置位比较，第一个数为正数，第二个数为负数，结果为 1，正确
e. 按位取反明显正确
f. 低位加载，把立即数的低 16 位 5678h 加载到目标操作数的低 16 位，并在高位补 0，正确

5.2 上箱验证

5.2.1 有符号数大于置位验证

首先验证 11111111 和 11111111，相等，所以应该返回 0，可以看到 result 为 0，正确

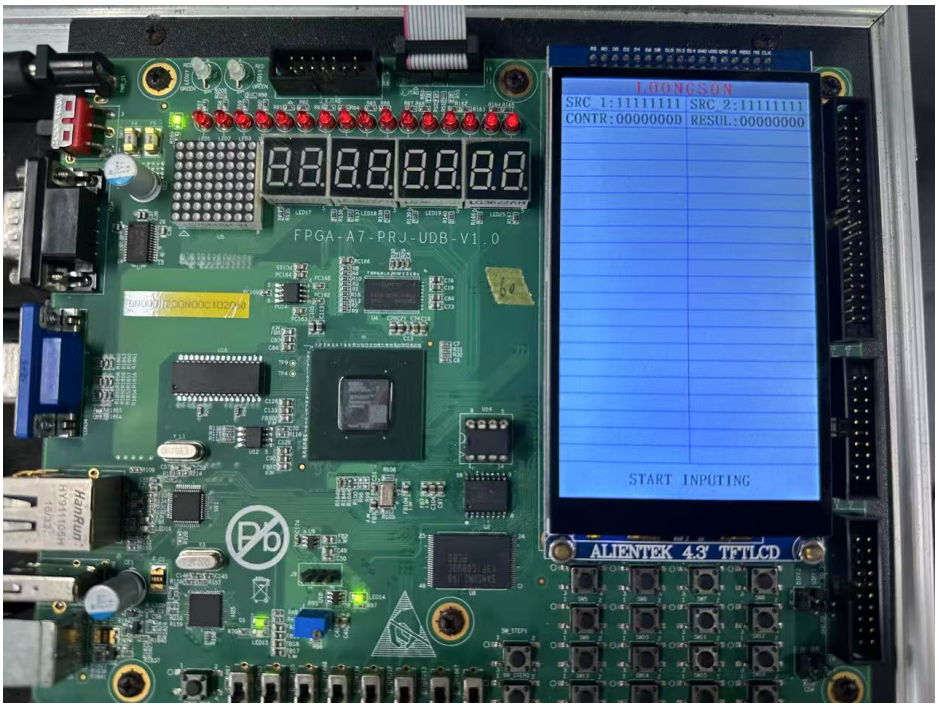

图 5.3: 同号比较

11111111 大于 1，所以 result 为 1，正确

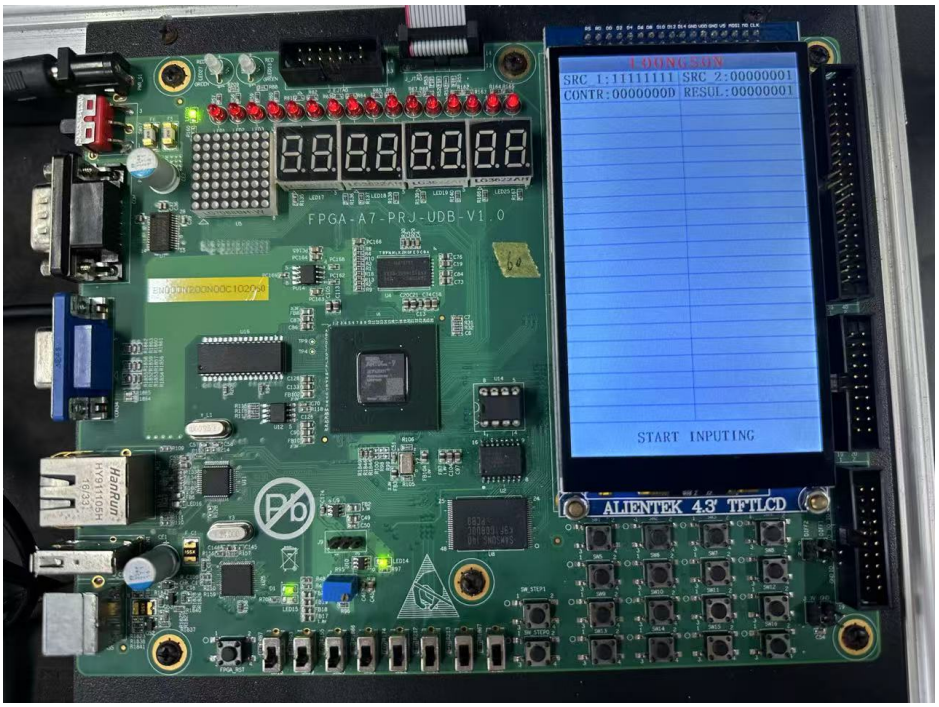

图 5.4: 同号比较

1 小于 11111111，所以 result 为 0，正确

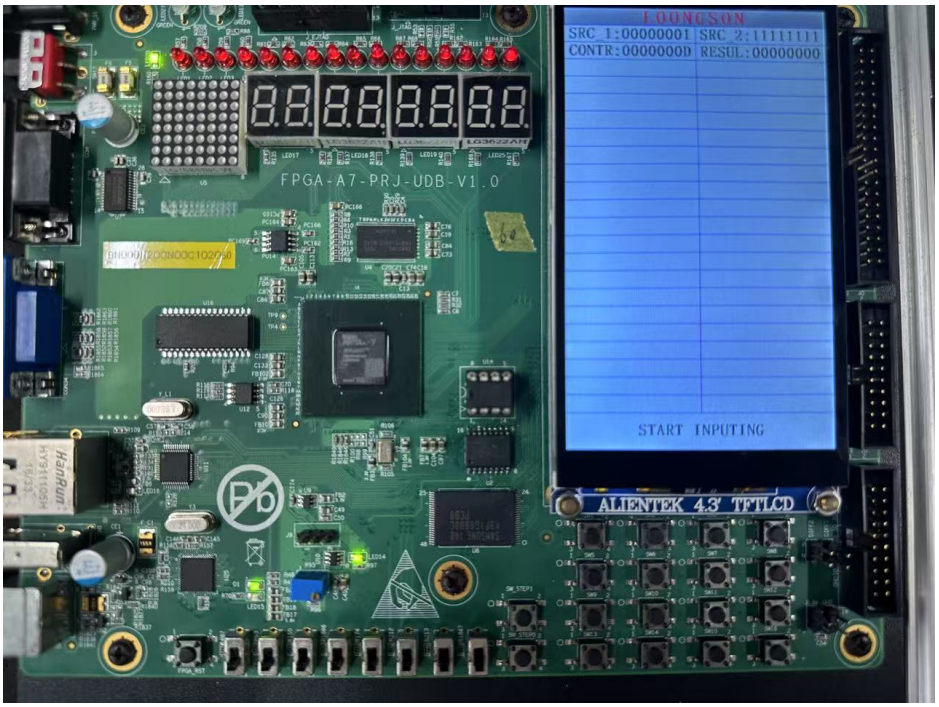

图 5.5: 同号比较

11111111 为正数，ffffffff 为负数，结果为大于，result 为 1，正确

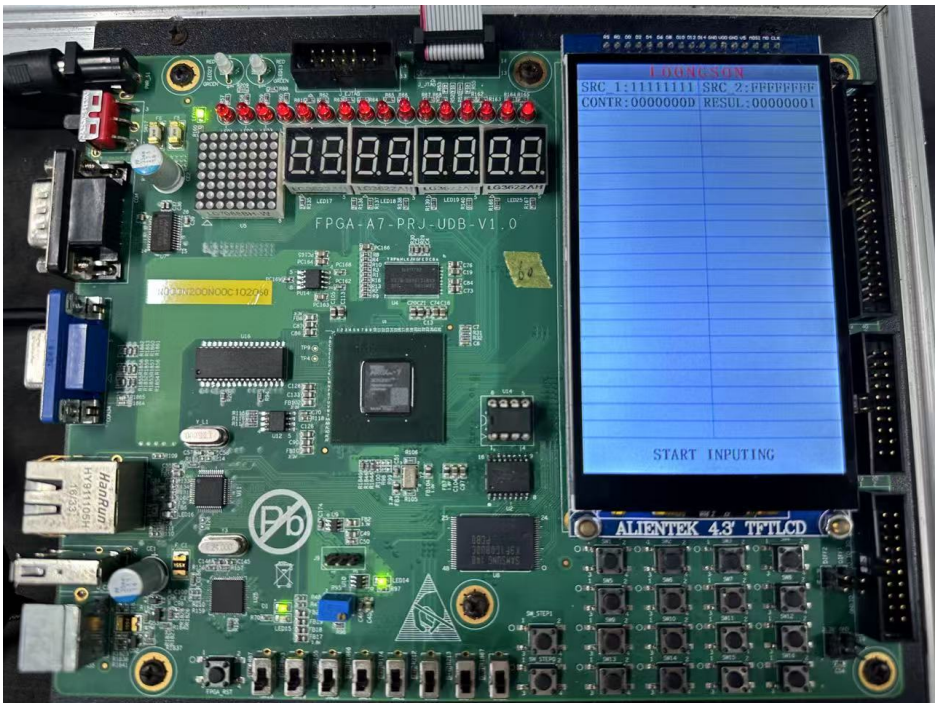

图 5.6: 正减负

ffffffff 为负数，11111111 为正确，结果为小于，result 为 0，正确

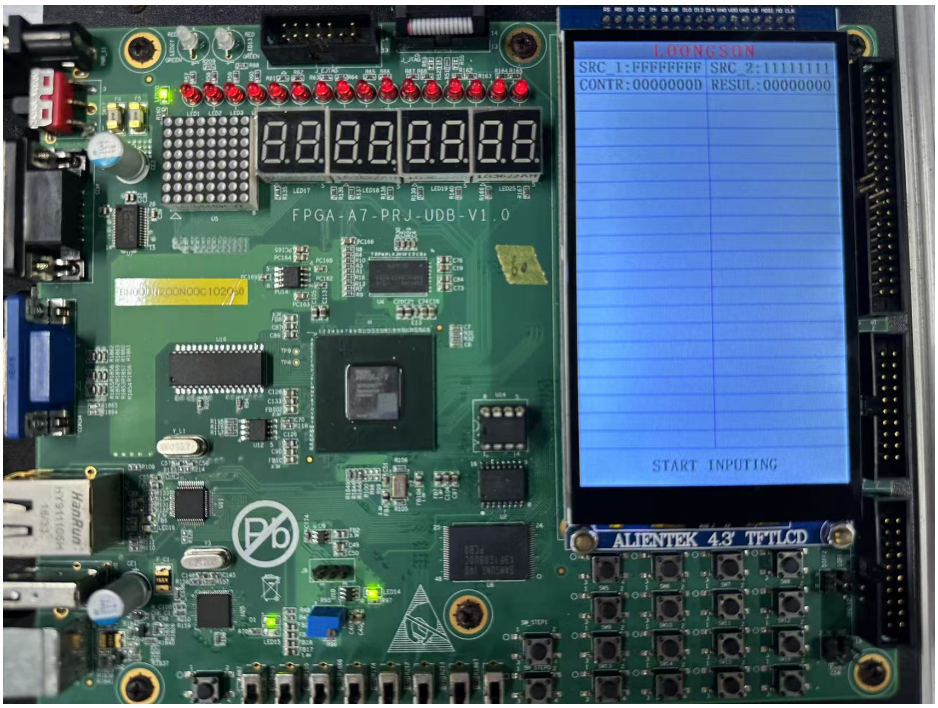

图 5.7: 负减正

5.2.2 按位取反验证

我们输入第二个操作数为 11111111，按位取反结果即为 eeeeeeee，正确

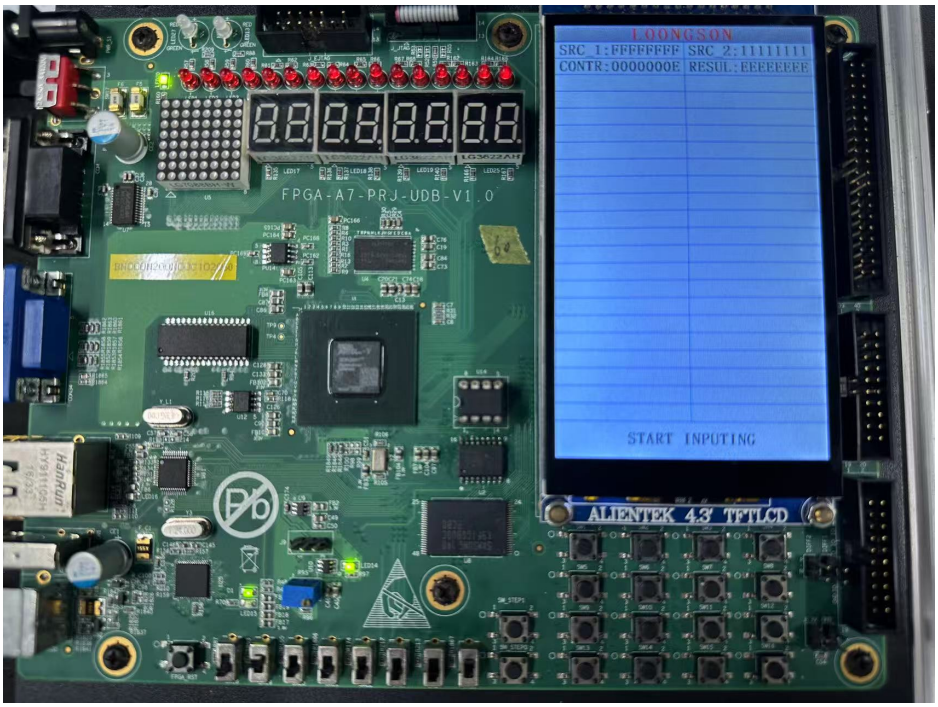

图 5.8: 按位取反

5.2.3 低位加载验证

输入第二个操作数为 12345678，低位加载即为 5678，高 16 位为 0，即为 00005678，正确

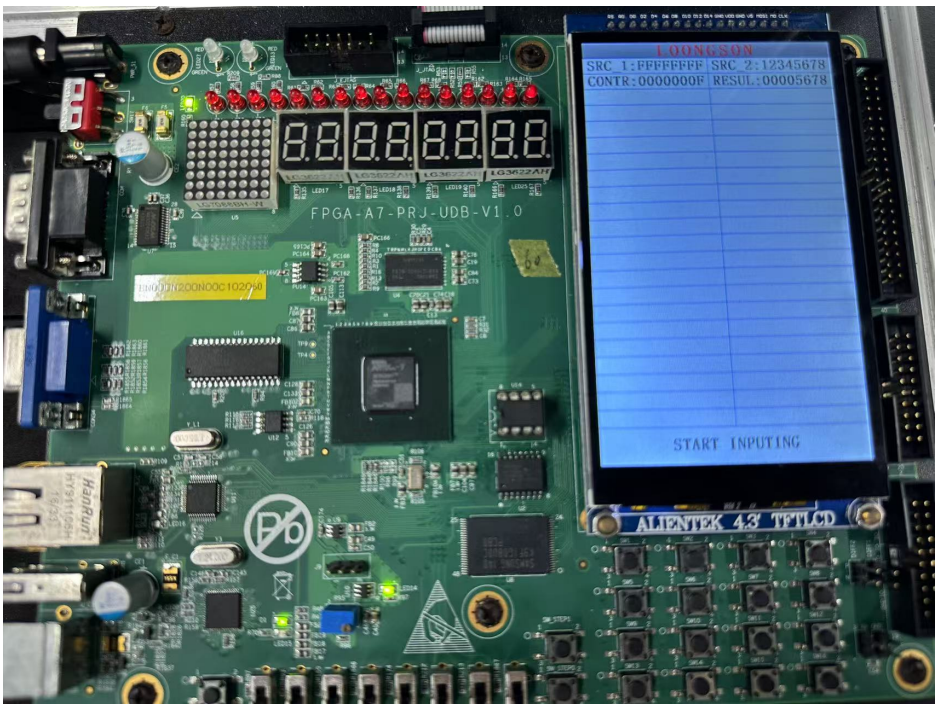

图 5.9: 低位加载

6 总结感想

1. 对计算机中减法操作如何实现有了更好的理解。另外比较操作都是利用这个减法的原理去实现的。

2. 能够把之前的独热编码控制逻辑改为更少的位数去接受，减少了线数与接口数，提高了资源利用率，为如何设计 cpu 打下基础。

3. 参照所给运算，自主设计了另外 3 种运算并得到实现，经得住验证。

4. 可以自主写出仿真文件去验证代码的正确性，为上箱节省了时间。