**组成原理实验课程第 二 次实验报告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 实验名称 | 乘法器实验 | | | 班级 | 李涛老师 |
| 学生姓名 | 张洋 | 学号 | 2111460 | 指导老师 | 董前琨老师 |
| 实验地点 | 津南实验楼A306 | | 实验时间 | 4月4日 18：30-20：30 | |

1. **实验目的**

（1）理解定点乘法的不同实现算法的原理，掌握基本实现算法。

（2）熟悉并运用verilog语言进行电路设计。

（3）为后续设计cpu的实验打下基础。

1. **实验内容说明**

（1）学习并理解计算机中定点乘法器的多种实现算法的原理，重点掌握迭代乘法的实现算法。

（2）将原有的迭代乘法改进成两位乘法，即每个时钟周期位移2位，从而提高乘法效率。本次实验要求实现的乘法为有符号乘法，因此需要注意计算机存储的有符号数都是补码的形式，设计方案传递进来的数也需是补码。

（3）根据设计的实验方案，使用verilog编写相应代码。

（4）对编写的代码进行仿真，得到正确的波形图。

（5）将以上设计作为一个单独的模块，设计一个外围模块去调用该模块，见下图。外围模块中需调用封装好的LCD触摸屏模块，需要利用触摸功能输入两个乘数显示两个乘数和乘法结果，且数据不在前四格显示。

定点乘法模块

外围模块

来自FPGA板子上的输入

输出到FPGA板上进行展示

定点乘法设计实验的顶层模块大致框图

（6）将编写的代码进行综合布局布线，并下载到实验箱中的FPGA板子上进行演示。

1. **实验原理图**

（1）迭代乘法的算法图

迭代乘法算法的原理图如下：



迭代乘法算法的原理图

图中参与运算的为两个乘数的绝对值，乘法结果也是绝对值，需要单独判断符号位后校正乘积。

迭代乘法是在模拟我们人算乘法的过程，乘数每次右移一位，根据最低位，判断是加被乘数移位后的值还是加0，不停地累加最后就得到乘积了。可以看到迭代乘法是用多次加法完成乘法操作的，故需要多拍时间，其结束标志为乘数移位后为0，故对于32位乘法，最多需要32拍才能完成一次乘法。

（2）两位乘法



两位乘法算法实现的基本原理与迭代乘法算法基本相同，但还有一些需要改进的地方：

* + 1. multiplicand每拍左移两位，multiplier每拍右移两位。
    2. 判断multiplicand最后两位的值更改product\_temp：multiplicand最后两位为00，product\_temp的值为0；multiplicand最后两位为01，product\_temp的值为multiplilier；multiplicand最后两位为10，product\_temp的值为multiplier+multiplier；multiplicand最后两位为11，product\_temp的值为multiplier+multiplier+multiplier。此处不能用乘法因为加法的效率大于乘法。

1. **实验步骤**
2. 理解两位乘法算法的基本原理
3. 改写verilog代码。将原有的迭代乘法改进成两位乘法，即每个时钟周期移位移两位，从而提高乘法效率。
4. 将改进后的乘法器进行仿真验证。
5. 完成调用定点乘法模块的外围模块的设计。
6. 对代码进行综合布局布线下载到实验箱里FPGA板上，进行上板验证。

**改进代码（改进处标红）：**

**1. multiply.v //实现两位乘法算法**

module multiply( // 乘法器

input clk, // 时钟

input mult\_begin, // 乘法开始信号

input [31:0] mult\_op1, // 乘法源操作数1

input [31:0] mult\_op2, // 乘法源操作数2

output [63:0] product, // 乘积

output mult\_end // 乘法结束信号

);

//乘法正在运算信号和结束信号

reg mult\_valid;

assign mult\_end = mult\_valid & ~(|multiplier); //乘法结束信号：乘数全0

always @(posedge clk)

begin

if (!mult\_begin || mult\_end)

begin

mult\_valid <= 1'b0;

end

else

begin

mult\_valid <= 1'b1;

end

end

//两个源操作取绝对值，正数的绝对值为其本身，负数的绝对值为取反加1

wire op1\_sign; //操作数1的符号位

wire op2\_sign; //操作数2的符号位

wire [31:0] op1\_absolute; //操作数1的绝对值

wire [31:0] op2\_absolute; //操作数2的绝对值

assign op1\_sign = mult\_op1[31];

assign op2\_sign = mult\_op2[31];

assign op1\_absolute = op1\_sign ? (~mult\_op1+1) : mult\_op1;

assign op2\_absolute = op2\_sign ? (~mult\_op2+1) : mult\_op2;

//加载被乘数，运算时每次左移两位

reg [63:0] multiplicand;

always @ (posedge clk)

begin

if (mult\_valid)

begin // 如果正在进行乘法，则被乘数每时钟左移两位

multiplicand <= {multiplicand[61:0],2'b0};

end

else if (mult\_begin)

begin // 乘法开始，加载被乘数，为乘数1的绝对值

multiplicand <= {32'd0,op1\_absolute};

end

end

//加载乘数，运算时每次右移两位

reg [31:0] multiplier;

always @ (posedge clk)

begin

if (mult\_valid)

begin // 如果正在进行乘法，则乘数每时钟右移两位

multiplier <= {2'b0,multiplier[31:2]};

end

else if (mult\_begin)

begin // 乘法开始，加载乘数，为乘数2的绝对值

multiplier <= op2\_absolute;

end

end

//部分积：

//multiplicand最后两位为00，product\_temp的值为0；

//multiplicand最后两位为01，product\_temp的值为multiplilier；

//multiplicand最后两位为10，product\_temp的值为multiplier+multiplier；

//multiplicand最后两位为11，product\_temp的值为multiplier+multiplier+multiplier；

wire [63:0] partial\_product;

assign partial\_product = multiplier[0] ? ( multiplier[1] ? (multiplicand + multiplicand + multiplicand) : multiplicand ) : ( multiplier[1] ? ( multiplicand + multiplicand ) : 64'd0 );

//累加器

reg [63:0] product\_temp;

always @ (posedge clk)

begin

if (mult\_valid)

begin

product\_temp <= product\_temp + partial\_product;

end

else if (mult\_begin)

begin

product\_temp <= 64'd0; // 乘法开始，乘积清零

end

end

//乘法结果的符号位和乘法结果

reg product\_sign;

always @ (posedge clk) // 乘积

begin

if (mult\_valid)

begin

product\_sign <= op1\_sign ^ op2\_sign;

end

end

//若乘法结果为负数，则需要对结果取反+1

assign product = product\_sign ? (~product\_temp+1) : product\_temp;

endmodule

**2.multiply\_display.v //添加展示外围模块**

module multiply\_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 input\_sel, //0:输入为乘数1;1:输入为乘数2

input sw\_begin,

//乘法结束信号

output led\_end,

//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

);

//-----{调用乘法器模块}begin

wire mult\_begin;

reg [31:0] mult\_op1;

reg [31:0] mult\_op2;

wire [63:0] product;

wire mult\_end;

assign mult\_begin = sw\_begin;

assign led\_end = mult\_end;

multiply multiply\_module (

.clk (clk ),

.mult\_begin(mult\_begin),

.mult\_op1 (mult\_op1 ),

.mult\_op2 (mult\_op2 ),

.product (product ),

.mult\_end (mult\_end )

);

reg [63:0] product\_r;

always @(posedge clk)

begin

if (!resetn)

begin

product\_r <= 64'd0;

end

else if (mult\_end)

begin

product\_r <= product;

end

end

//-----{调用乘法器模块}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0时，表示输入数为乘数1

always @(posedge clk)

begin

if (!resetn)

begin

mult\_op1 <= 32'd0;

end

else if (input\_valid && !input\_sel)

begin

mult\_op1 <= input\_value;

end

end

//当input\_sel为1时，表示输入数为乘数2

always @(posedge clk)

begin

if (!resetn)

begin

mult\_op2 <= 32'd0;

end

else if (input\_valid && input\_sel)

begin

mult\_op2 <= 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'd5 : //第一个乘数显示在lcd屏上的第5块

begin

display\_valid <= 1'b1;

display\_name <= "M\_OP1";

display\_value <= mult\_op1;

end

6'd6 : //第二个乘数显示在lcd屏上的第6块

begin

display\_valid <= 1'b1;

display\_name <= "M\_OP2";

display\_value <= mult\_op2;

end

6'd7 : //结果的前32位显示在lcd屏上的第7块

begin

display\_valid <= 1'b1;

display\_name <= "PRO\_H";

display\_value <= product\_r[63:32];

end

6'd8 : //结果的后32位显示在lcd屏上的第8块

begin

display\_valid <= 1'b1;

display\_name <= "PRO\_L";

display\_value <= product\_r[31: 0];

end

default :

begin

display\_valid <= 1'b0;

display\_name <= 48'd0;

display\_value <= 32'd0;

end

endcase

end

**3.testbench.v //在进行功能仿真时，需要先建立一个testbench(测试平台)。产生输入激励，送入到要测试的功能模块里，然后读出功能模块的执行结果，与预期的结果进行比较，以此验证功能模块的正确性。**

module tb;

// Inputs

reg clk;

reg mult\_begin;

reg [31:0] mult\_op1;

reg [31:0] mult\_op2;

// Outputs

wire [63:0] product;

wire mult\_end;

// Instantiate the Unit Under Test (UUT)

multiply uut (

.clk(clk),

.mult\_begin(mult\_begin),

.mult\_op1(mult\_op1),

.mult\_op2(mult\_op2),

.product(product),

.mult\_end(mult\_end)

);

initial begin

// Initialize Inputs

clk = 0;

mult\_begin = 0;

mult\_op1 = 0;

mult\_op2 = 0;

// Wait 100 ns for global reset to finish

#100;

mult\_begin = 1;

mult\_op1 = $random; //更改为随机数

mult\_op2 = $random;

#400;

mult\_begin = 0;

#100;

mult\_begin = 1;

mult\_op1 = $random;

mult\_op2 = $random;

#400;

mult\_begin = 0;

#100;

mult\_begin = 1;

mult\_op1 = $random;

mult\_op2 = $random;

#400;

mult\_begin = 0;

#100;

mult\_begin = 1;

mult\_op1 = $random;

mult\_op2 = $random;

#400;

mult\_begin = 0;

// Add stimulus here

end

always #5 clk = ~clk;

endmodule

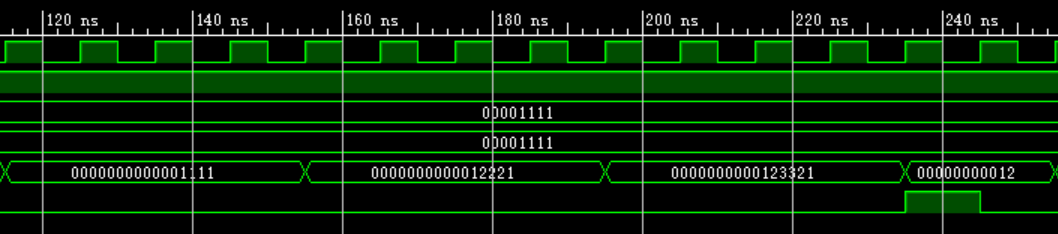
**4.multiply.xdc** **//引脚绑定的约束文件。**

（相较于源代码没有更改故没有粘贴到实验报告中）

1. **实验结果分析**

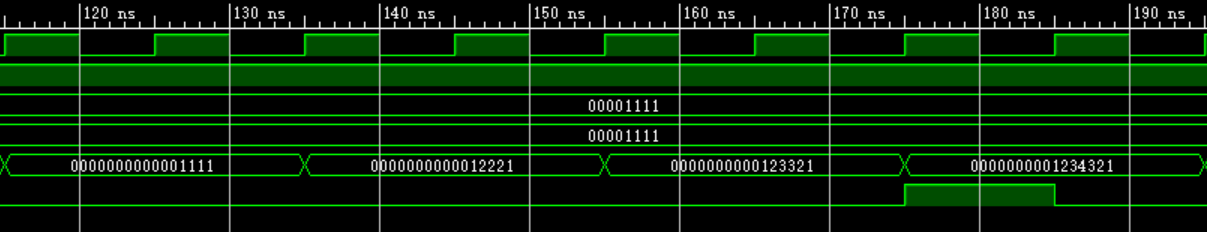
**仿真验证：**

1. 更改代码前（迭代乘法）：



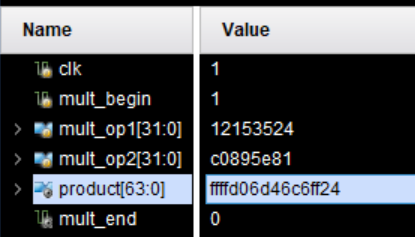
可以看到经过12个时钟周期得出结果。

（2） 更改代码后（两位乘法）：



可以看到经过6个时钟周期得出结果，因此**两位乘法比迭代乘法效率更高。**

（3） 验证结果

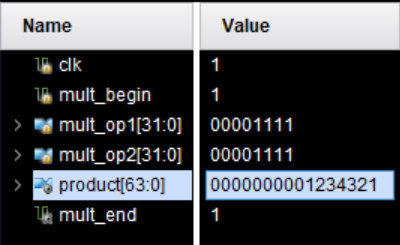
 12153524\*c0895e81=ffffd06d46c6ff24

输入：mult\_op1 = 12153524，mult\_op2 = c0895e81，mult\_begin = 1

输出：product = ffffd06d46c6ff24

为了方便检查结果把数据写成简单的十六进制数进行检验

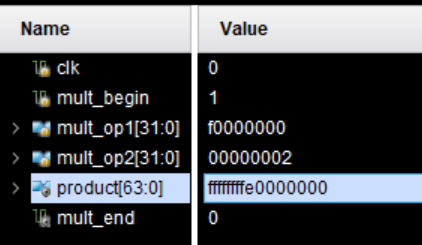
* + 1. 正数\*正数

 00001111\*00001111=0000000001234321

输入：mult\_op1 = 00001111，mult\_op2 = 00001111，mult\_begin = 1

输出：product = 0000000001234321

* + 1. 正数\*负数

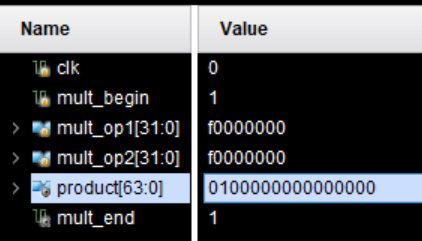
 f0000000\*00000002=ffffffffe0000000

输入：mult\_op1 = f0000000，mult\_op2 = 00000002，mult\_begin = 1

输出：product = ffffffffe0000000

算法的实现步骤：

1. 符号位为负，mult\_op1取绝对值后为10000000
2. 取绝对值后的mult\_op1和mult\_op2相乘得到0000000200000000
3. 加上符号位，上一步得到的结果取补码为ffffffffe0000000，即为最终结果
   * 1. 负数\*负数

 f0000000\*f0000000=0100000000000000

输入：mult\_op1 = f0000000，mult\_op2 = f0000000，mult\_begin = 1

输出：product = 0100000000000000

算法的实现步骤：

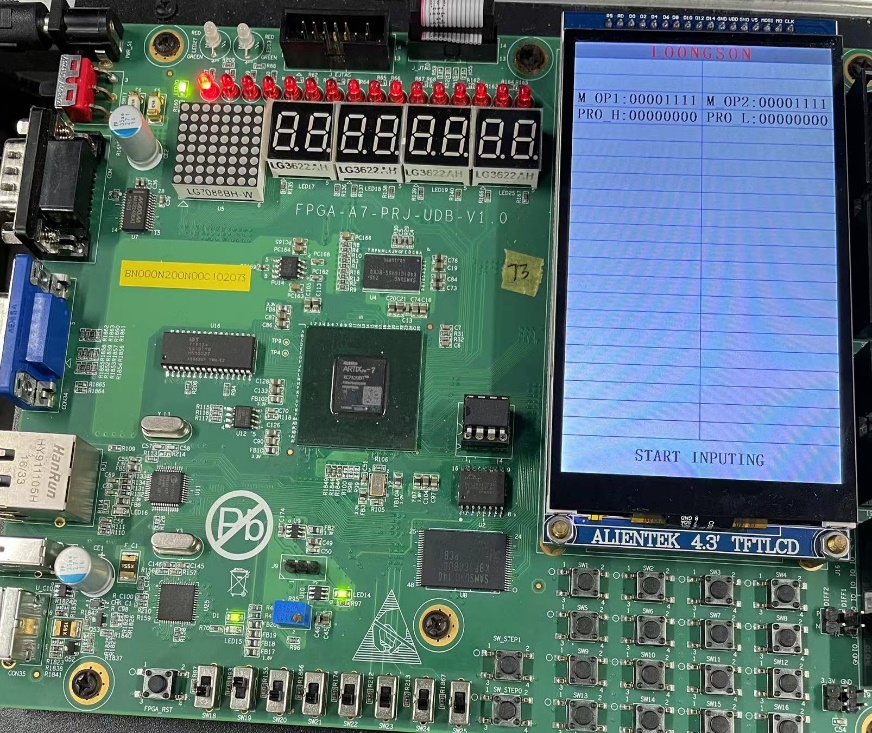
（1）multi\_op1和mult\_op2的绝对值为10000000

（2）符号位为正，计算两个绝对值相乘即为最终结果：10000000\*10000000=0100000000000000

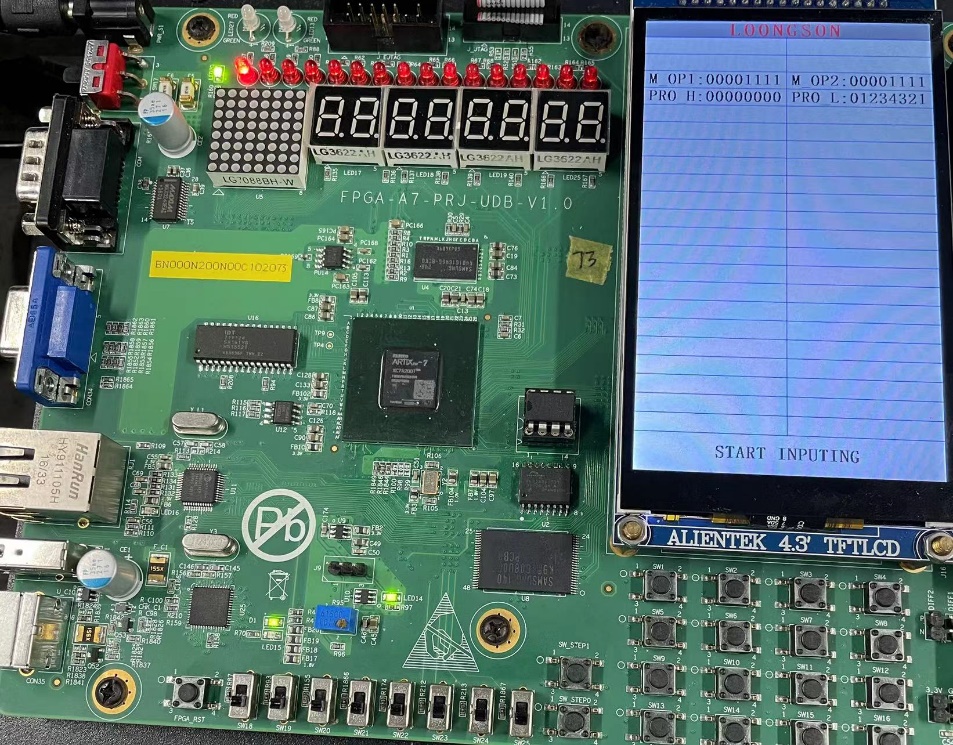
**在实验箱上验证：**

i. 正数\*正数

（1）输入两个乘数00001111\*00001111

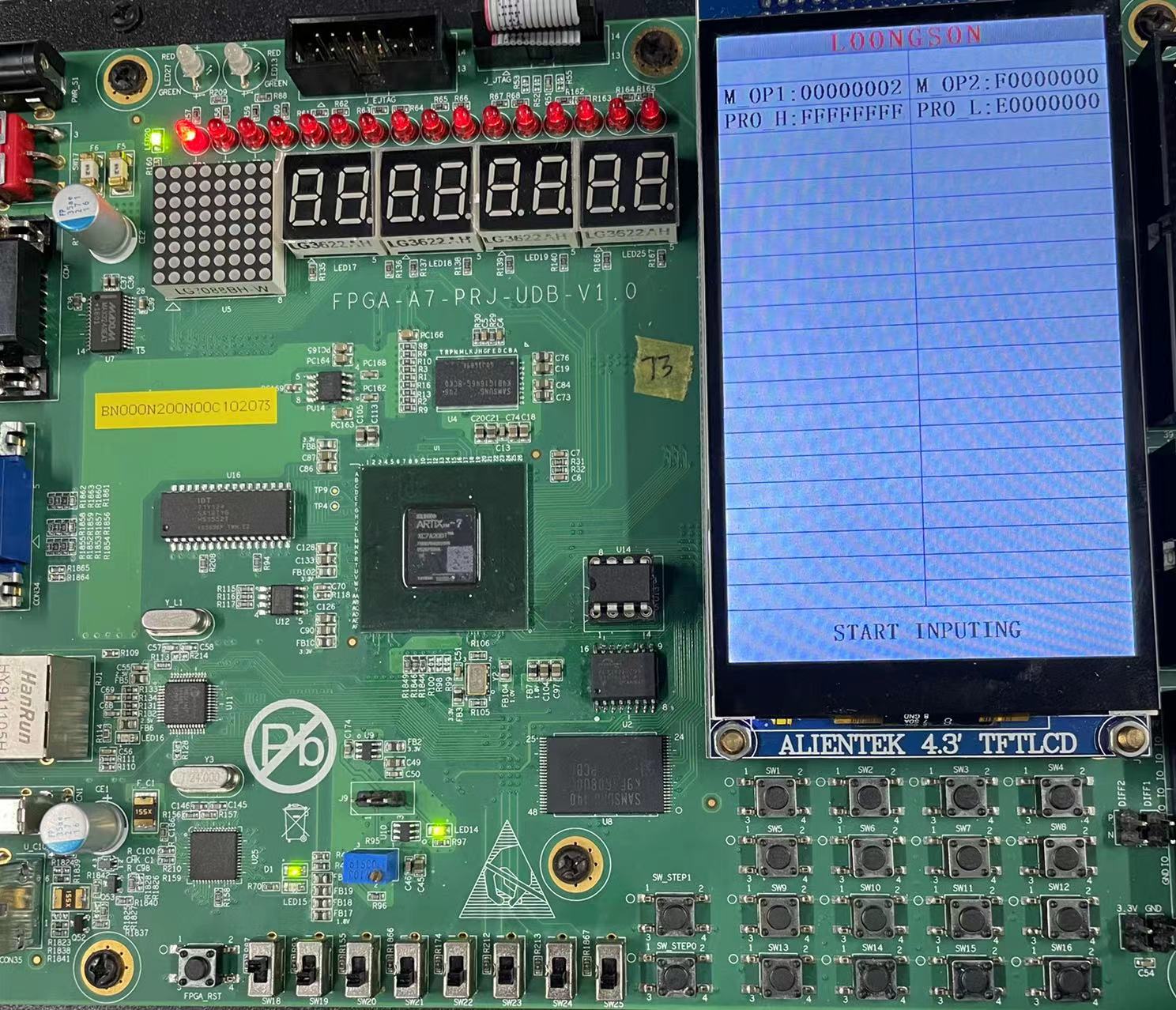


1. 拨动拨码开关SW19得到结果0000000001234321



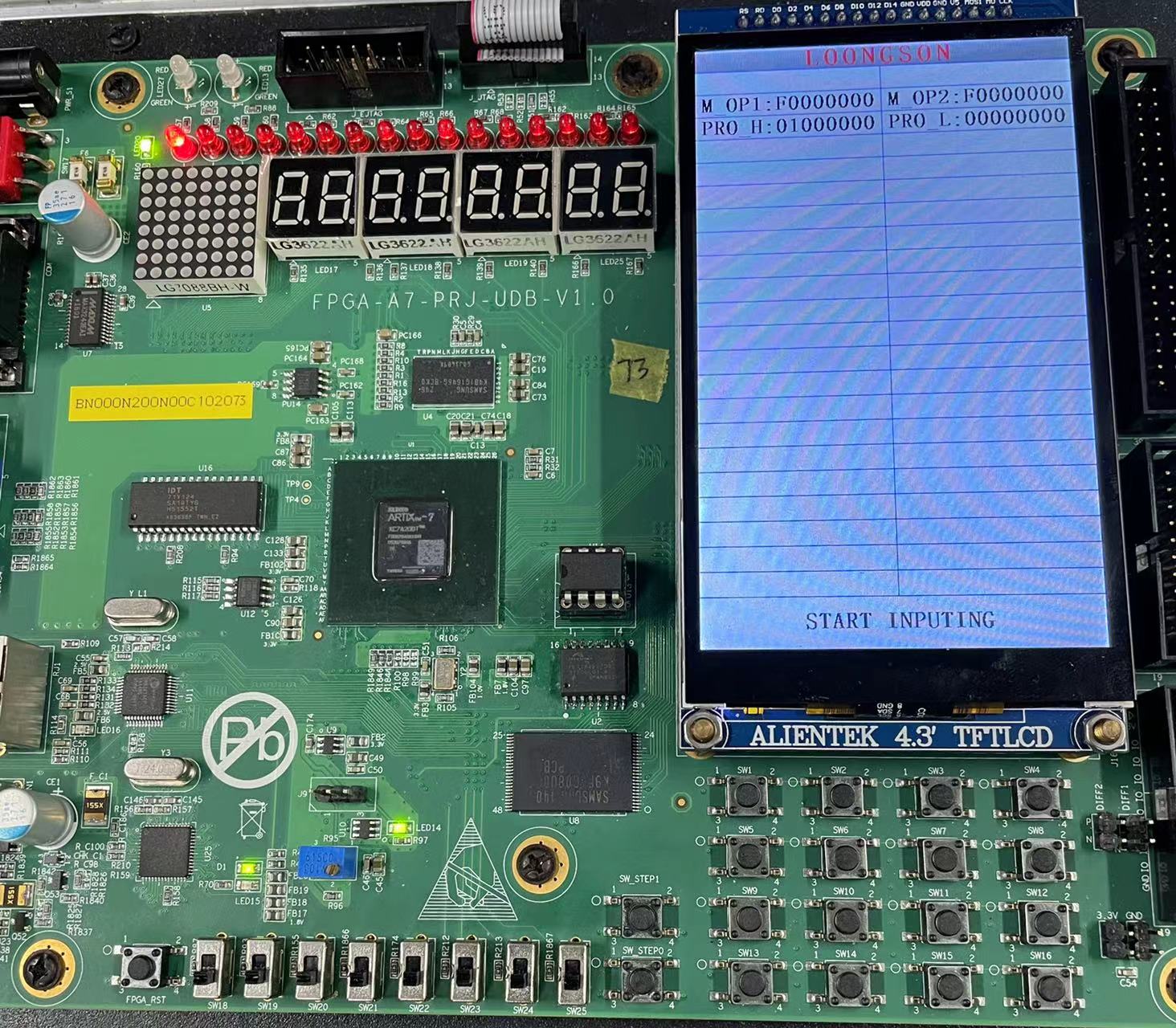
00001111\*00001111 = 0000000001234321，结果正确。

ii. 正数\*负数



00000002\*f0000000 = ffffffffe0000000，结果正确。

iii. 负数\*负数



F0000000\*F0000000 = 0100000000000000，结果正确。

1. **总结感想**

在将迭代乘法更改为两位乘法的实验中，我学习了如何使用Verilog语言来实现数字电路设计。此外，我还深入了解了乘法器的工作原理和实现方法。

通过这个实验，我发现两位乘法器相对于迭代乘法器具有更高的效率和更快的速度，因为它可以同时计算多个位。与迭代乘法器相比，两位乘法器需要更少的时钟周期和更少的硬件资源来完成相同的操作。

在编写Verilog代码的过程中，我学习了如何使用模块化编程和层次式设计来实现复杂的数字电路。这种方法可以使代码更易于维护和修改，并且可以提高代码的可重用性。此外，我还学会了如何使用仿真工具来验证我的设计，并且可以对其进行调试和优化。

总之，这个实验为我提供了一个深入理解两位甚至多位乘法器设计的机会，并且使我更加熟练地掌握了Verilog编程技巧。我相信这些知识和技能将对我的未来学习和职业发展有所裨益。