**组成原理课程第 2 次实验报告**

**实验名称：定点乘法与加法**

学号： 2312141 姓名： 张德民 班次： 李涛老师班

一.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，并进行改进。

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。

二.实验内容说明

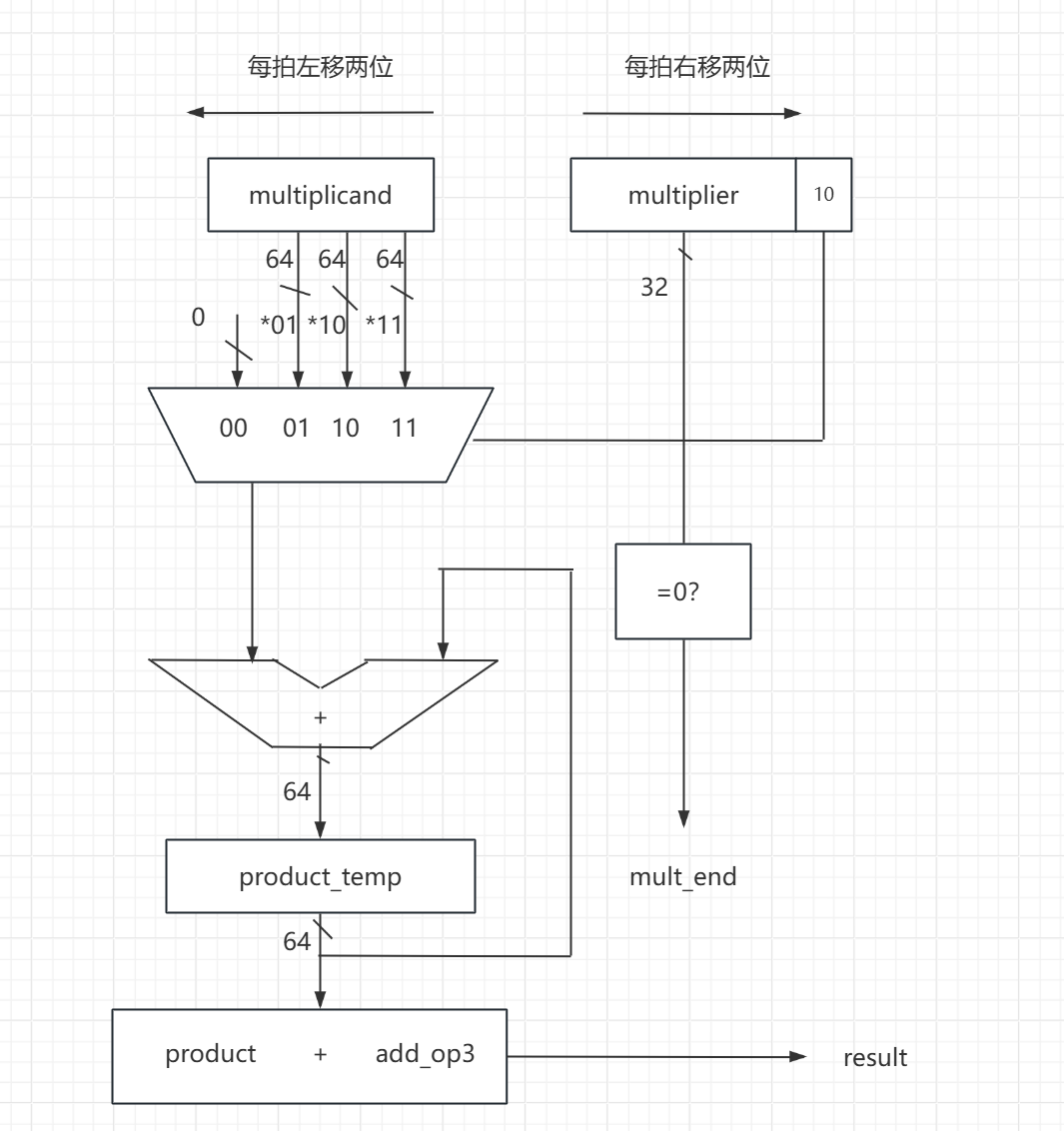
### 将原始的补码一位乘法修改成补码两位乘法，也就是每个时钟周期移两位。

### 将乘法器修改成乘加器，实现A\*B+C的效果，模块至少有三个32位的输入操作数和一个64位的输出操作数（由于有加法，可以考虑添加进位输入和输出，也可以不考虑）。

### 仿真文件原始代码中延迟400、500的时间单位过长，大家可以修改成40/50，这样看波形比较直观，注意波形图上应该是multi\_end为1时的输出才是正确输出。

### 4、上实验箱进行验证时，注意要lcd屏上需要输入三个数据，input\_sel不能再用1位了，此外lcd屏上还需要显示ABC三个数。

三.实验原理图

迭代乘法与加法原理图：

跟原来的相比主要有俩点不同。

首先这里被乘数和乘数每拍要移动两位，同样的，根据乘数的最后两位(00,01,10,11),对于被乘数我们也有不同的处理方发，乘数最后两位为00时，结果为0，为01时，结果为被乘数乘1，也就是它本身，为10时要乘以2，也就是左移一位，为11时要乘以3，也就是左移一位再加上自己本身。这样就实现的乘法，得到乘法结果。

其次，进行完乘法以后，要再加上一个加数add\_op3,然后得到最后的结果。

## 四.实验步骤

Multiply模块的修改

代码如下：

`timescale 1ns / 1ps

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// > 文件名: multiply.v

// > 描述 ：乘法器模块，低效率的迭代乘法算法，使用两个乘数绝对值参与运算

// > 作者 : LOONGSON

// > 日期 : 2016-04-14

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

input [31:0] add\_op3, //加法操作数

output [63:0] product, // 乘积

output [63:0] ans, //最后答案

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;

reg [63:0] multiplicand;

always @ (posedge clk)

begin

if (mult\_valid)

begin

multiplicand <= {multiplicand[61:0],2'b00};

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，运算时每次右移一位

reg [31:0] multiplier;

always @ (posedge clk)

begin

if (mult\_valid)

begin // 如果正在进行乘法，则乘数每时钟右移一位

multiplier <= {2'b00,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：乘数末位为1，由被乘数左移得到；乘数末位为0，部分积为0

wire [63:0] partial\_product1;

wire [63:0] partial\_product0;

wire [63:0] partial\_product;

assign partial\_product0=multiplier[0] ? multiplicand : 64'd0;

assign partial\_product1=multiplier[1] ? {multiplicand[62:0],1'b0} : 64'd0;

assign partial\_product = partial\_product0+partial\_product1;

//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

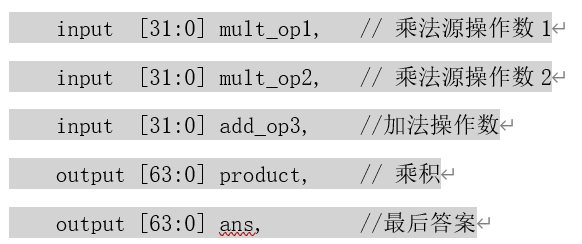
assign product = product\_sign ? (~product\_temp+1) : product\_temp;

assign ans=product+add\_op3;

endmodule

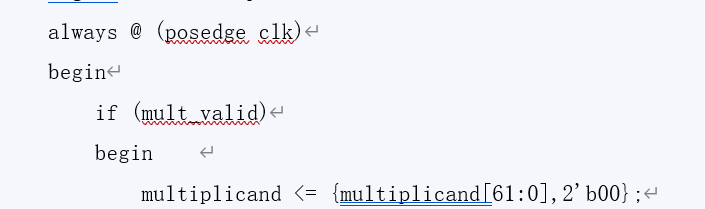
修改点：

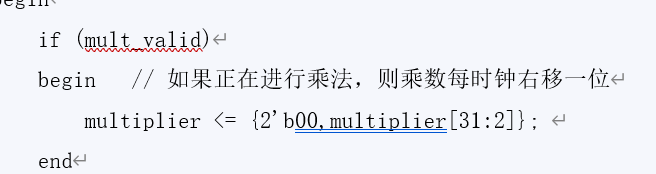
1.



除了原来的两个乘法数之外，我又添加了一个加法操作数add\_op3,并且product并不是最终答案，只是乘法的答案，我用ans来代表实现加法后的答案。

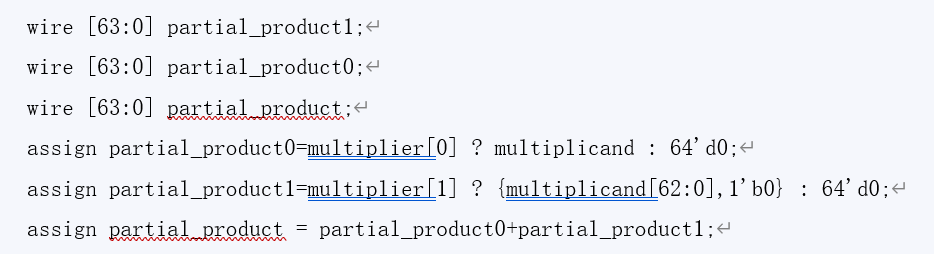
2.





对于移位操作，由原来的每个时钟周期移动一位改为每个周期移动两位。

3.



改为移动两位之后，每个周期得到的结果就由原来的两种可能变成了四种可能，也就是乘以00，01，10，11，即1，2，3，4，那么我增加partial\_product1和partial\_product2两个值用来中转，partial\_product1用来存第一位的结果，即乘数最后一位如果是0，那么partial\_product1就是0，如果是1，那么partial\_product1就是被乘数本身。partial\_product2用来存第二位的结果，即乘数倒数第二位如果是0，那么partial\_product2就是0，如果是1，那么partial\_product2就是被乘数乘以2（左移一位）。最终的partial\_product是二者相加。

4.



这里的最后结果ans是乘法结果product与加数之和。

Testbench模块的修改

代码如下：

module testbench;

// Inputs

reg clk;

reg mult\_begin;

reg [31:0] mult\_op1;

reg [31:0] mult\_op2;

reg [31:0] add\_op3;

// Outputs

wire [63:0] product;

wire [63:0] ans;

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),

.add\_op3(add\_op3),

.product(product),

.mult\_end(mult\_end),

.ans(ans)

);

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 = 32'H00001111;

mult\_op2 = 32'H00001111;

add\_op3=32'H00001111;

#80;

mult\_begin = 0;

#40;

mult\_begin = 1;

mult\_op1 = 32'H00001111;

mult\_op2 = 32'H00002222;

add\_op3=32'H00001111;

#80;

mult\_begin = 0;

#40;

mult\_begin = 1;

mult\_op1 = 32'H00000002;

mult\_op2 = 32'HFFFFFFFF;

add\_op3=32'H00001111;

#80;

mult\_begin = 0;

#40;

mult\_begin = 1;

mult\_op1 = 32'H00000002;

mult\_op2 = 32'H80000000;

add\_op3=32'H00001111;

#80;

mult\_begin = 0;

// Add stimulus here

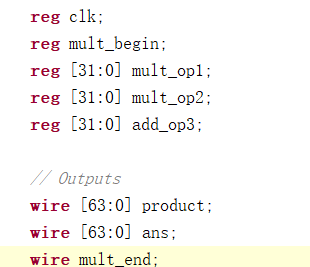
end

always #5 clk = ~clk;

endmodule

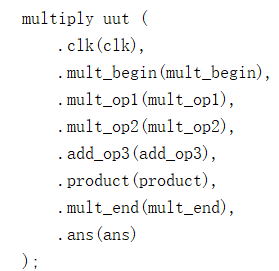
修改点：

1.



增加了我需要的新的参数add\_op3和最终结果ans.

2.



调用模块时增加了新的参数。

3.

修改了运行时的等候时间，方便最后的观察。

Multiply\_display模块的修改

代码如下：

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\_sel1,

input input\_sel2,

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;

reg [31:0] add\_op3;

wire [63:0] product;

wire [63:0] ans;

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 ),

.add\_op3 (add\_op3),

.product (product ),

.ans (ans),

.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 <= ans;

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\_sel1为0,sel2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\_sel1&&!input\_sel2)

begin

mult\_op1 <= input\_value;

end

end

//当input\_sel1为1,sel2为0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\_sel1&&!input\_sel2)

begin

mult\_op2 <= input\_value;

end

end

//当input\_sel1为0,sel2为1时，表示输入数为加数

always @(posedge clk)

begin

if (!resetn)

begin

add\_op3 <= 32'd0;

end

else if (input\_valid && !input\_sel1&&input\_sel2)

begin

add\_op3 <= 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 <= "M\_OP1";

display\_value <= mult\_op1;

end

6'd2 :

begin

display\_valid <= 1'b1;

display\_name <= "M\_OP2";

display\_value <= mult\_op2;

end

6'd3 :

begin

display\_valid <= 1'b1;

display\_name <= "A\_OP3";

display\_value <= add\_op3;

end

6'd5 :

begin

display\_valid <= 1'b1;

display\_name <= "PRO\_H";

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

end

6'd6 :

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

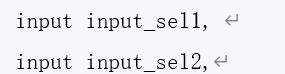
//-----{输出到触摸屏显示}end

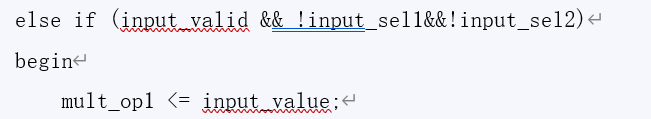
//----------------------{调用触摸屏模块}end---------------------//

endmodule

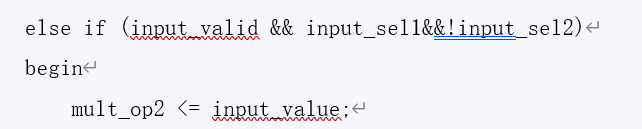
修改点：

1.

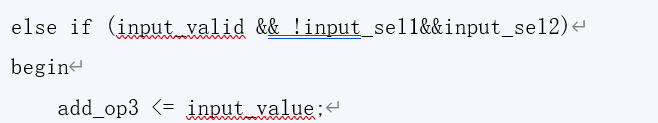


拨码开关由原来的1个改为2个，因为我要输入的是3个数，一个开关无法满足需求。

当input\_sel1和2都是0时，输入被乘数op1.



当input\_sel2是0, input\_sel1是1时，输入乘数op2.



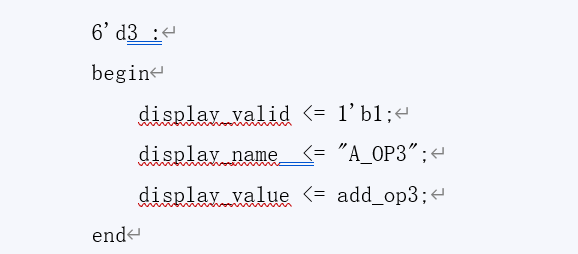
当input\_sel2是1, input\_sel2是1时，输入乘数op2.

2.



这部分修改了调用的参数，增加add\_op3和ans。

3.



触摸屏上，选用第三块来显示加数，其余的需要也依次进行调整。

限制文件的修改

修改点：



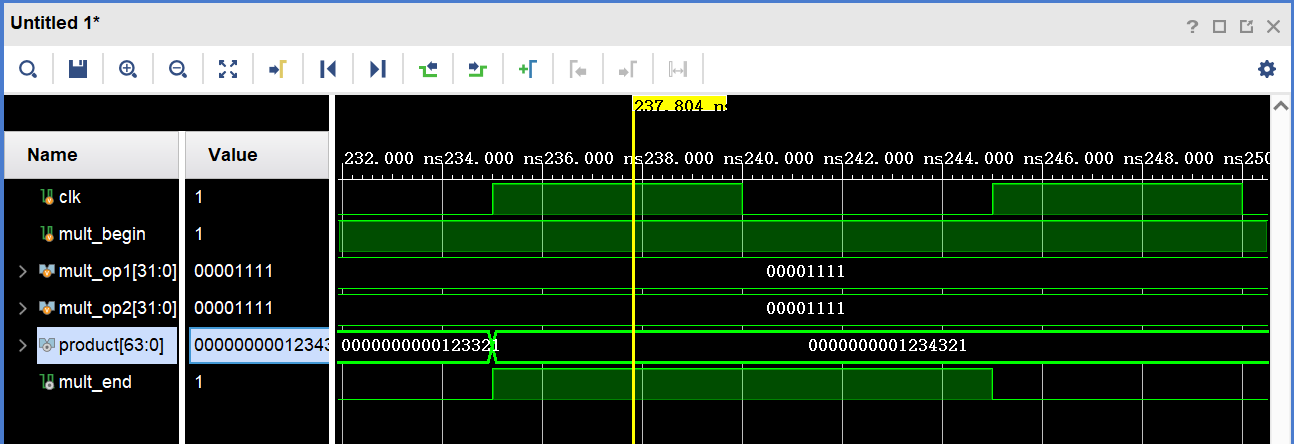
原先的AC21开关作为input\_sel1, AC22作为input\_sel2.



设置俩拨码开关的输入输出标准。

## 五.实验结果分析

原代码复现：

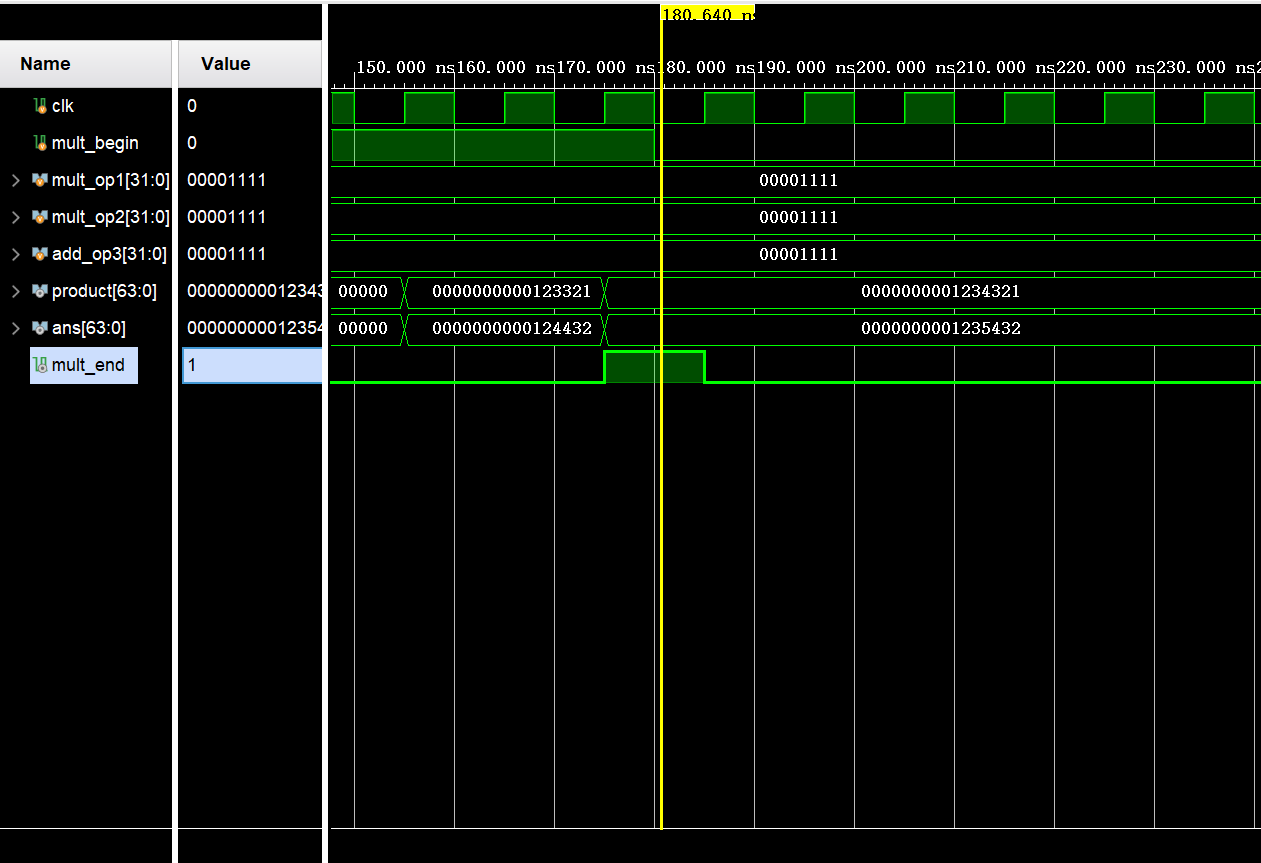


如上图，原乘法器使用1111h乘1111h，结果是1234321h，答案正确，复现成功。

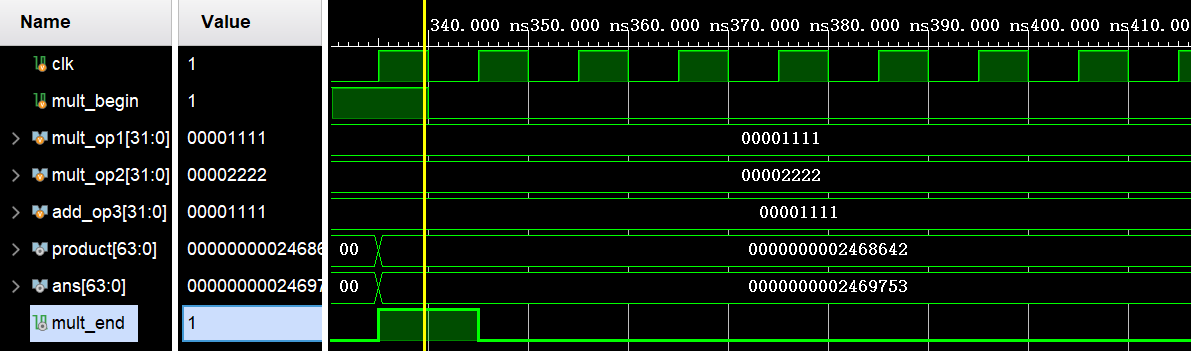
波形图仿真：

这里的验证我使用的是原代码提供的四个运算，并且每个运算里的加数我都设置为1111，这是为了方便观察。

1.

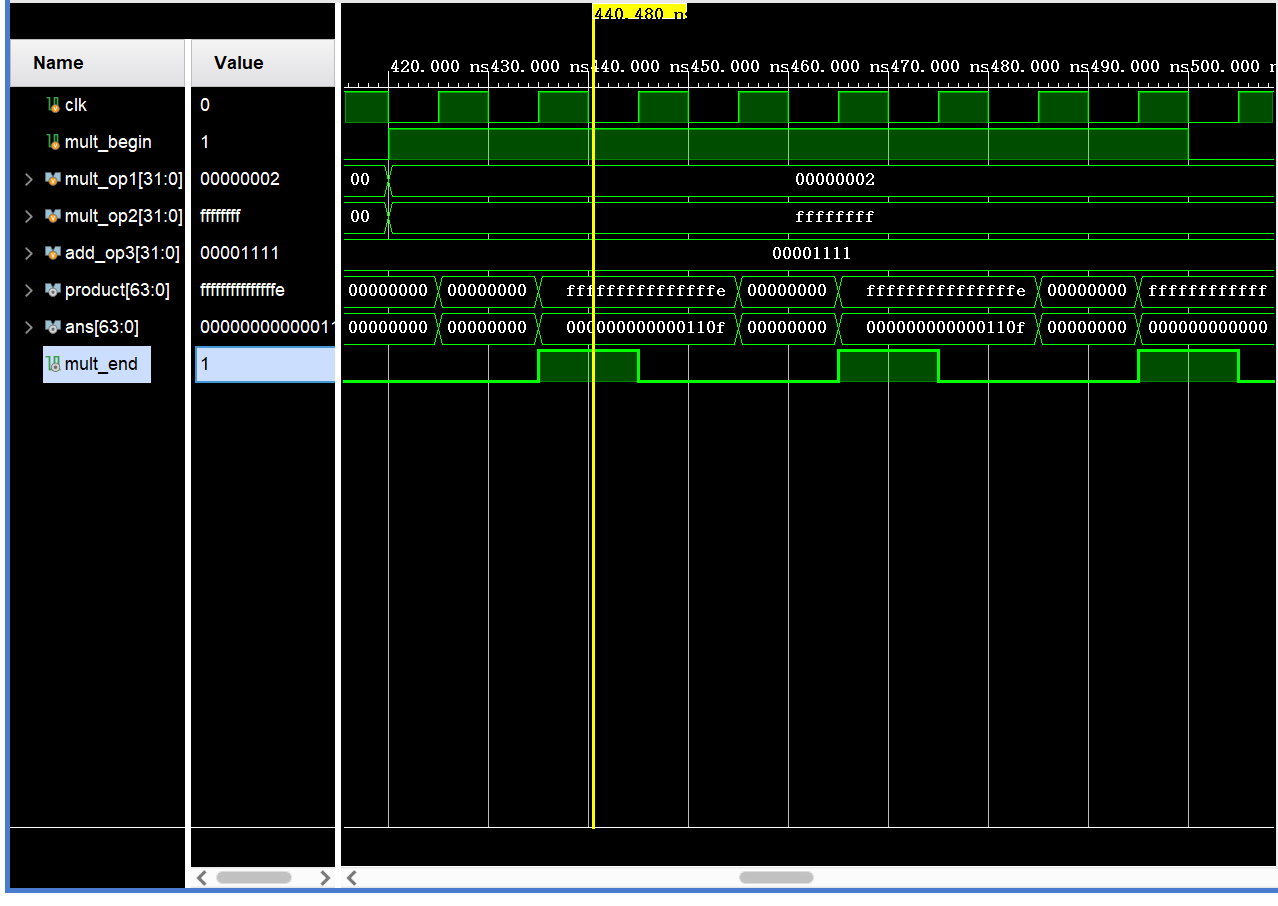
如图，1111\*1111+1111，换为10进制就是4369\*4369+4369，答案是19092539，验证结果正确。

2.



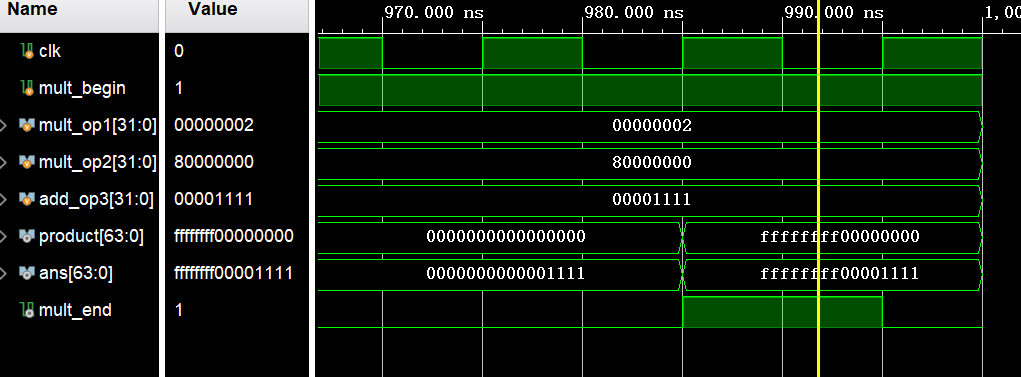
1111\*2222+1111，换为10进制就是4369\*8738+4369，答案是38180691，验证答案正确。

3.



2\*ffffffff(-1)+1111, 换为10进制就是2\*(-1)+4369，答案是4367,验证答案是正确的。

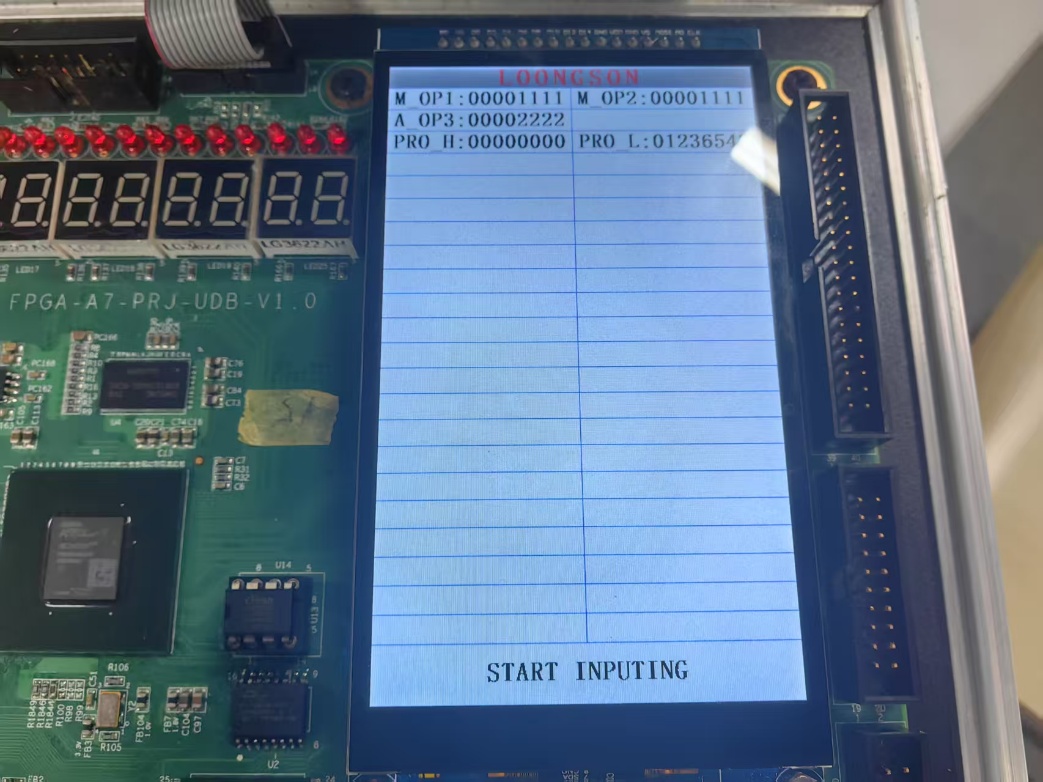
4.



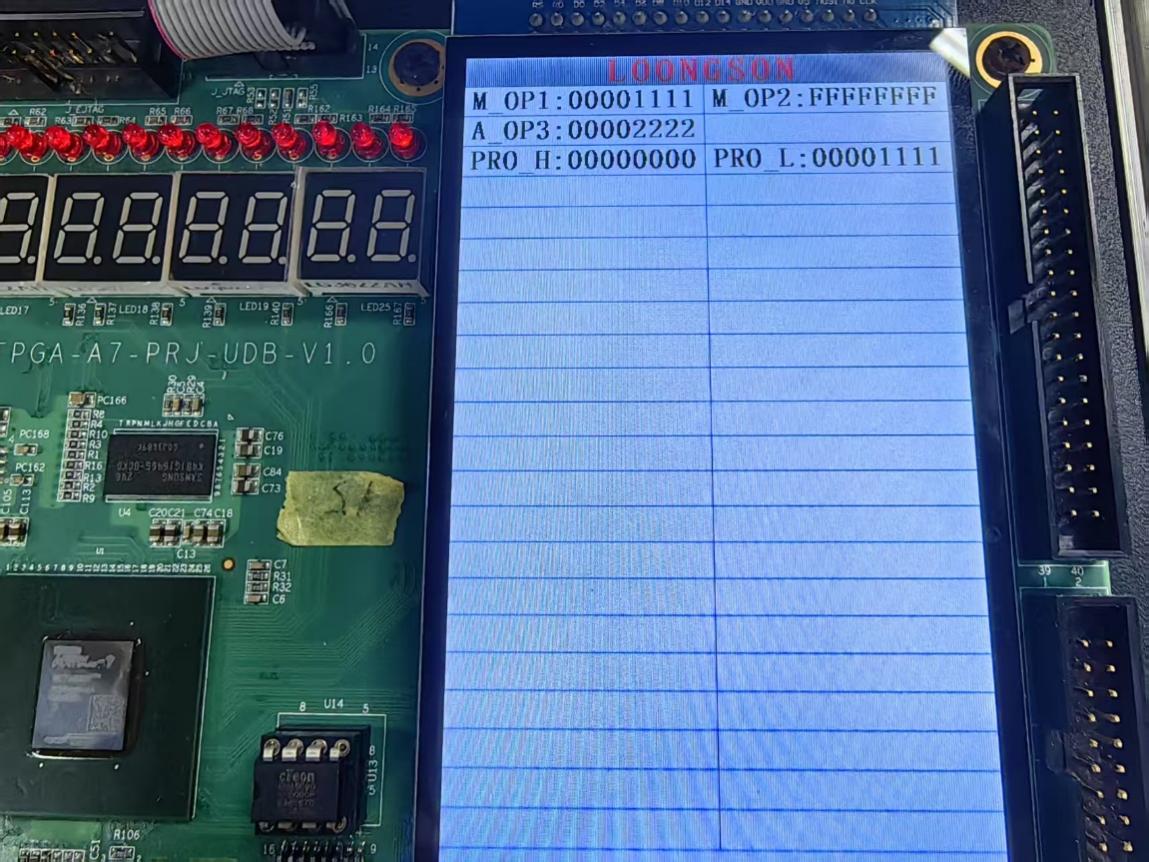
2\*80000000+1111，要注意到，80000000就是-2^32，换为10进制就是2\*-2^32+4369，答案是ffffffff00001111，也就是-4294962927，验证得答案正确。

实验箱验证：

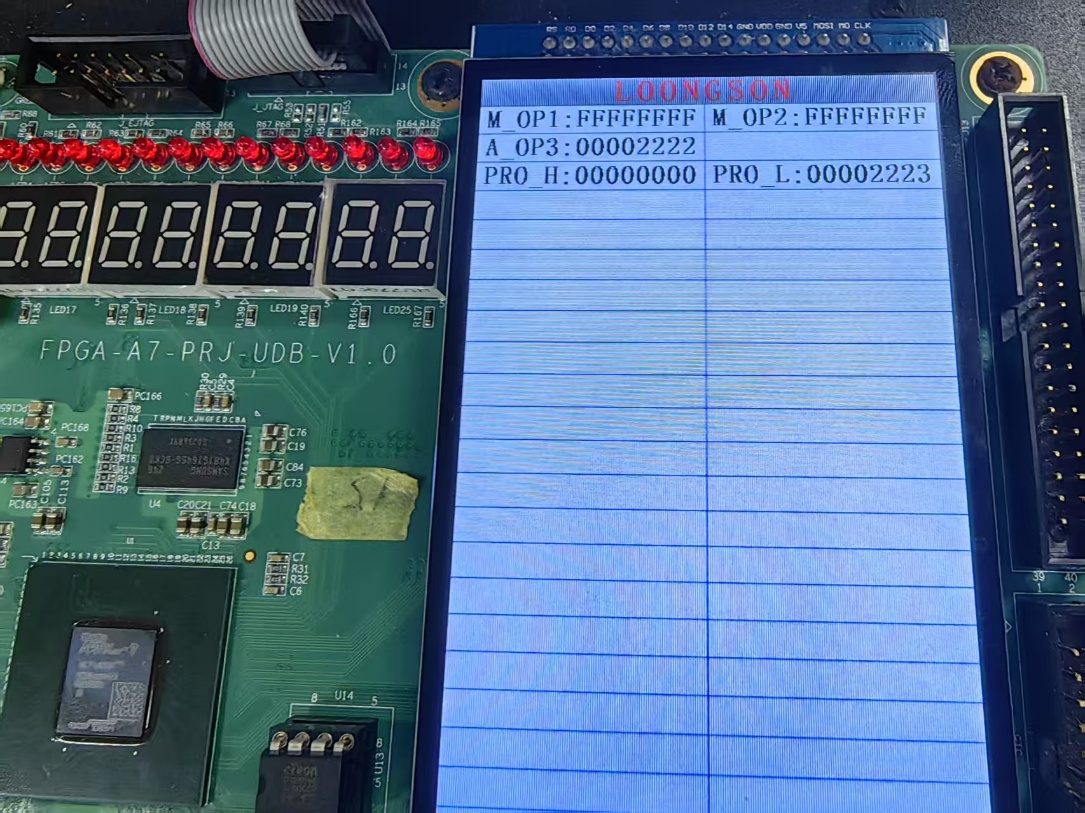
实验箱上我也用了四组不同的数，验证不同的情况

1. 正数乘正数  
   

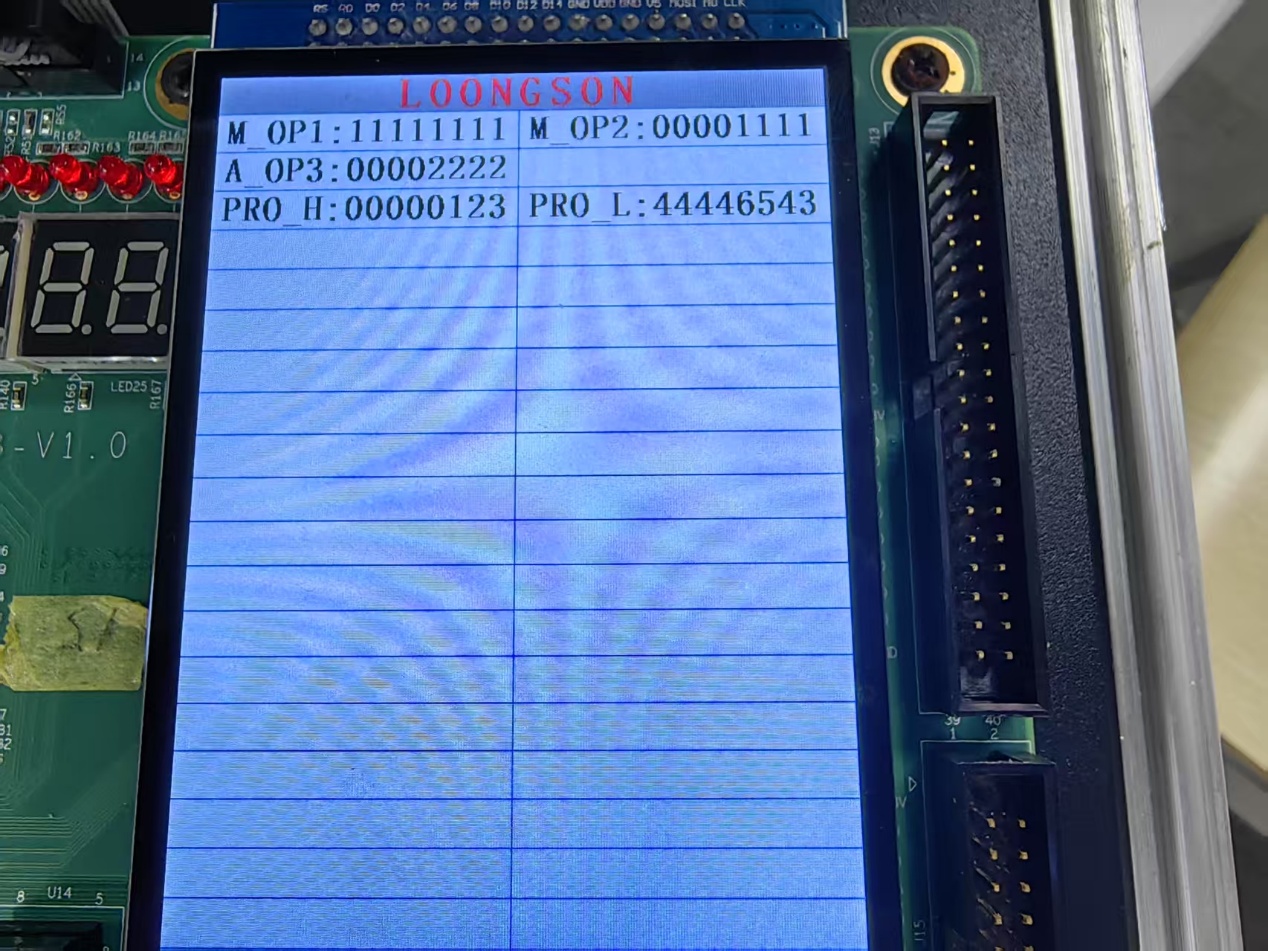
如图，1111\*1111+2222，和波形图得到的答案是一样的，正确。

1. 正数乘负数  
   

如图，1111\*(-1)+2222,最后答案显然是1111，正确。

1. 负数乘负数  
   

-1\*(-1)+2222,最后答案是2223，显然是正确的。

1. 超32位  
   
2. 用11111111\*1111+2222，这里主要是为了验证一下使用两块触摸屏来表示64位数是正确的，经过验证这个答案也是正确的。

## 六.总结感想

1.本次实验在原先移动一位的乘法器基础上进行了改进，将其修改为每一次移动两位，在这个改进的过程里，我对于迭代乘法的原理有了更深刻的认识，还有对于verilog语言的语句是并行执行的，与其他编程语言按顺序执行不同这一点有了更加深刻的认识。对于verilog的always等语句也更加清楚了。对于cpu执行的原理也有更多的认识，对于如何提高cpu计算速度（16周期改为8个）有个一定的认识。

2.在修改展示文件以及限制文件时，我查阅了各个引脚对应图，这令我对于实验箱仿真的各个流程更加熟悉了。