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

1、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。

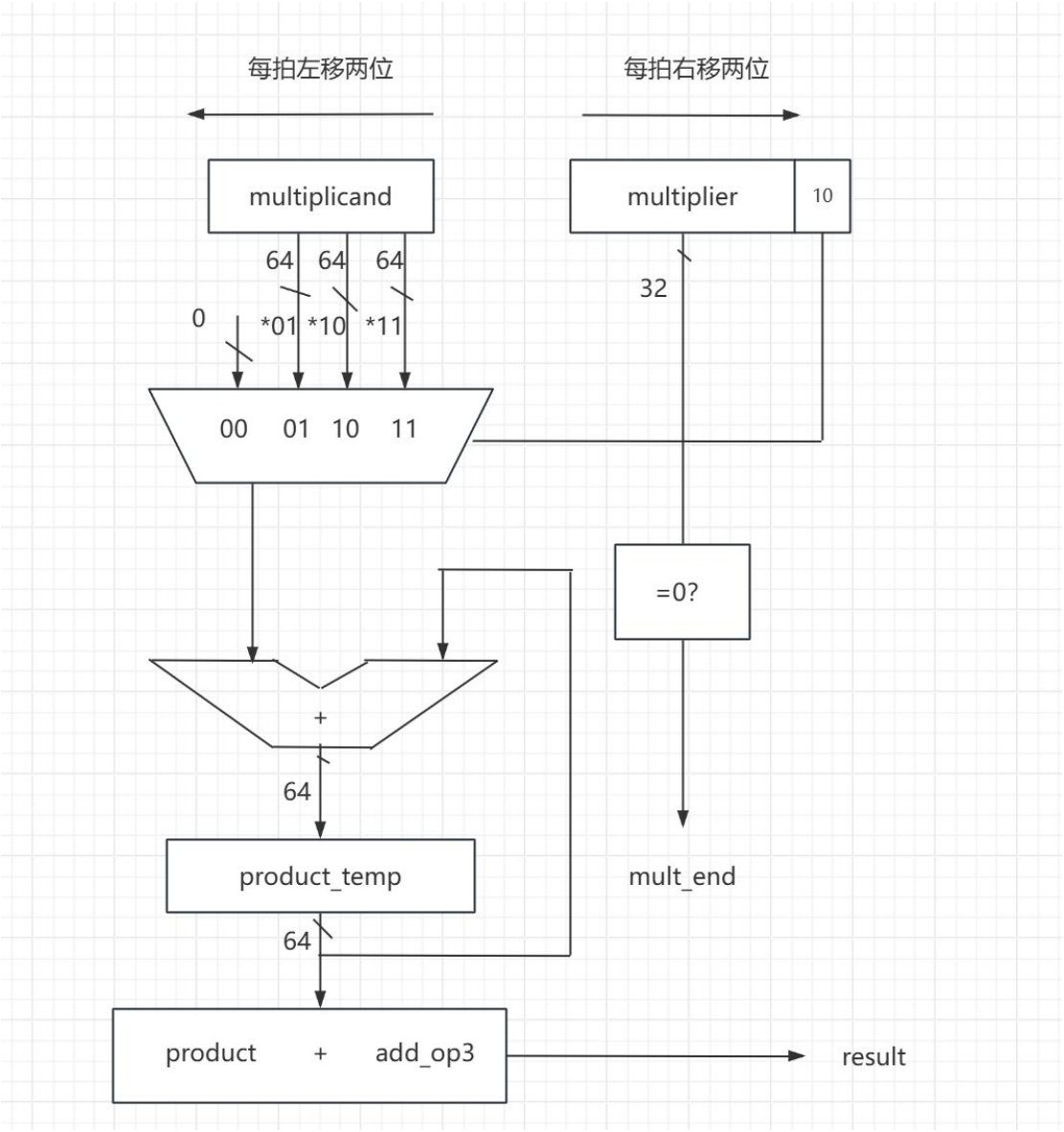
2、将乘法器修改成乘加器，实现 A*B+C 的效果，模块至少有三个 32 位的输入操作数和一个 64 位的输出操作数（由于有加法，可以考虑添加进位输入和输出，也可以不考虑）。

3、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.

```
    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
```

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.

```
always @ (posedge clk)
begin
    if (mult_valid)
    begin
        multiplicand <= {multiplicand[61:0],2'b00};
```

```
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
```

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.

```
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;
```

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.

```
assign ans=product+add_op3;
```

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.

```
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;
```

增加了我需要的新的参数 add_op3 和最终结果 ans.

2.

```
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)
);
```

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.

```
input input_sel1,
input input_sel2,
```

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。

```
else if (input_valid && !input_sel1&&!input_sel2)
begin
    mult_op1 <= input_value;
```

当 input_sel1 和 2 都是 0 时，输入被乘数 op1.

```
 else if (input_valid && input_sel1&&!input_sel2)
 begin
     mult_op2 <= input_value;
```

当 input_sel2 是 0, input_sel1 是 1 时，输入乘数 op2.

```
else if (input_valid && !input_sel1&&input_sel2)
begin
    add_op3 <= input_value;
```

当 input_sel2 是 1, input_sel2 是 1 时，输入乘数 op2.

2.

```
//-----{调用乘法器模块}begin
```

这部分修改了调用的参数，增加 add_op3 和 ans。

3.

```
        6'd3 :
        begin
            display_valid <= 1'b1;
            display_name  <= "A_OP3";
            display_value <= add_op3;
        end
```

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：

```
set_property PACKAGE_PIN AC21 [get_ports input_sel1]
set_property PACKAGE_PIN AC22 [get_ports input_sel2]
```

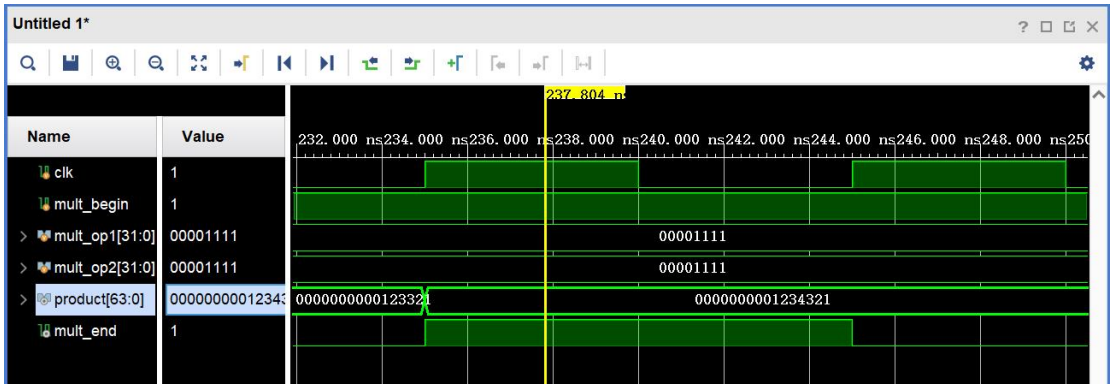
原先的 AC21 开关作为 input_sel1, AC22 作为 input_sel2.

```
set_property IOSTANDARD LVCMOS33 [get_ports input_sel1]
set_property IOSTANDARD LVCMOS33 [get_ports 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：

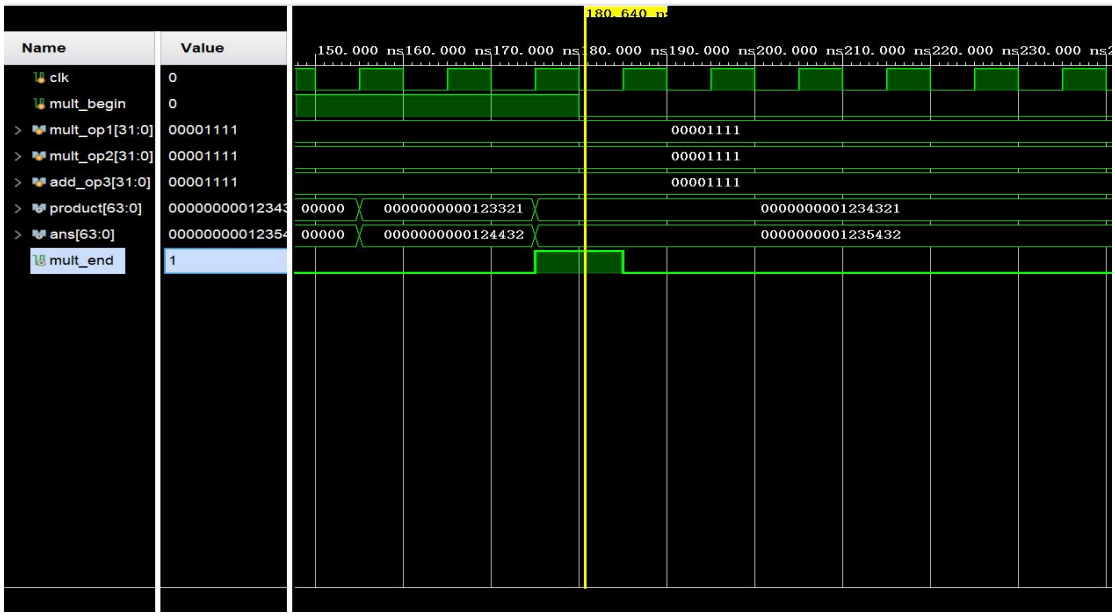


如上图，原乘法器使用 1111h 乘 1111h，结果是 1234321h，答案正确，复现成功。

波形图仿真：

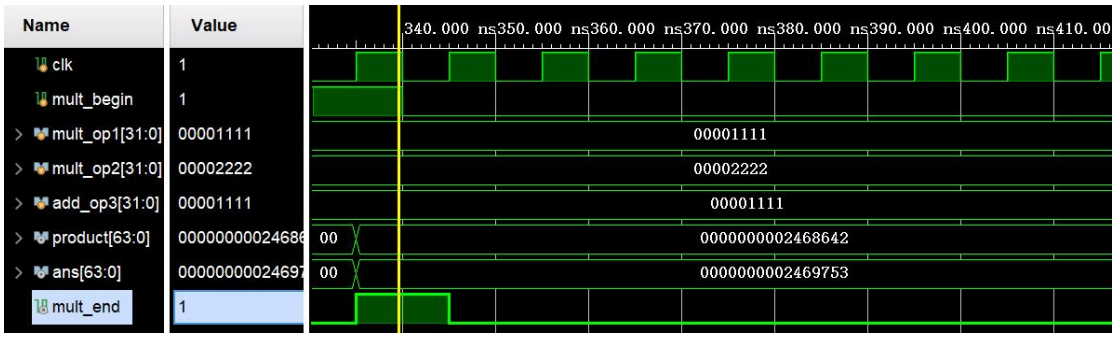
这里的验证我使用的是原代码提供的四个运算，并且每个运算里的加数我都设置为 1111，这是为了方便观察。

1.



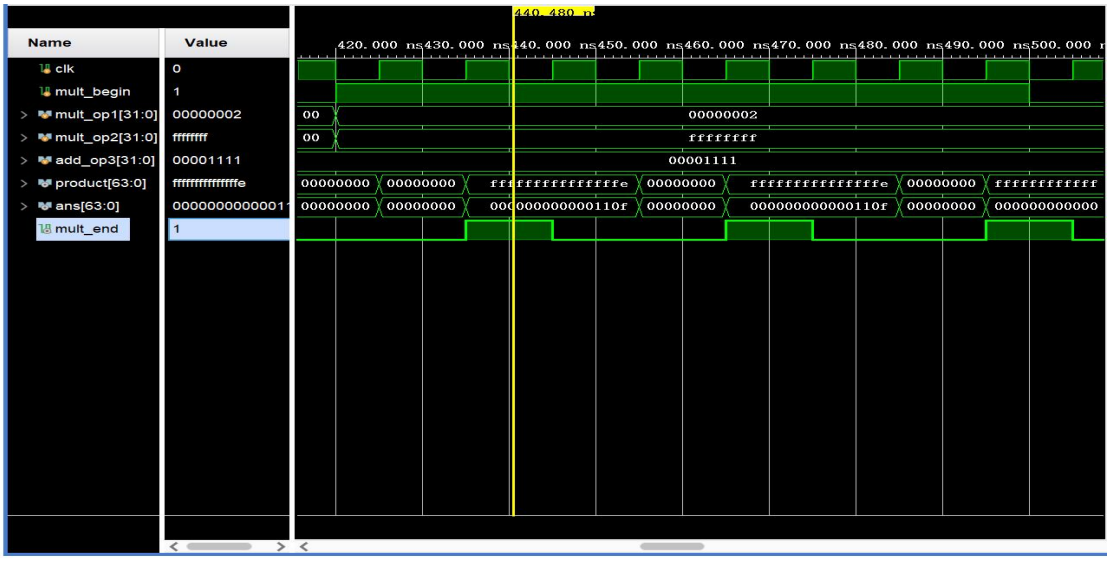
如图，1111*1111+1111，换为 10 进制就是 4369*4369+4369，答案是 19092539，验证结果正确。

2.



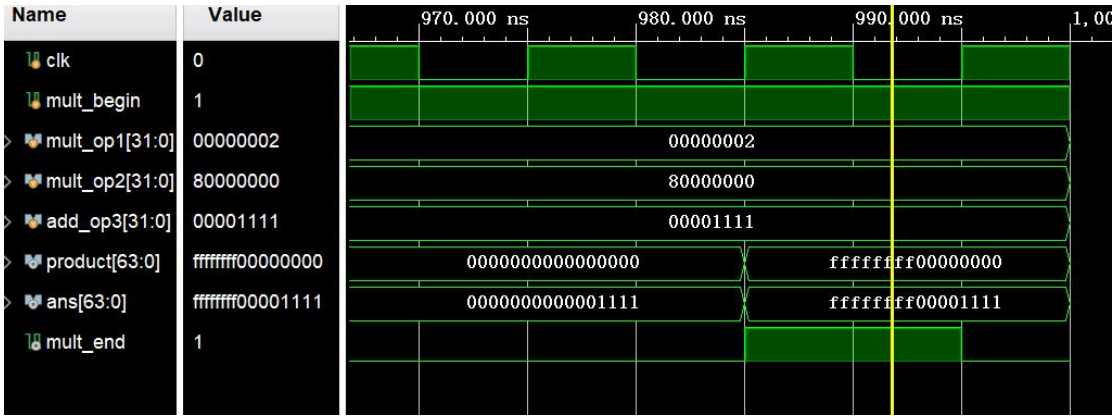
1111*2222+1111，换为 10 进制就是 4369*8738+4369，答案是 38180691，验证答案正确。

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。

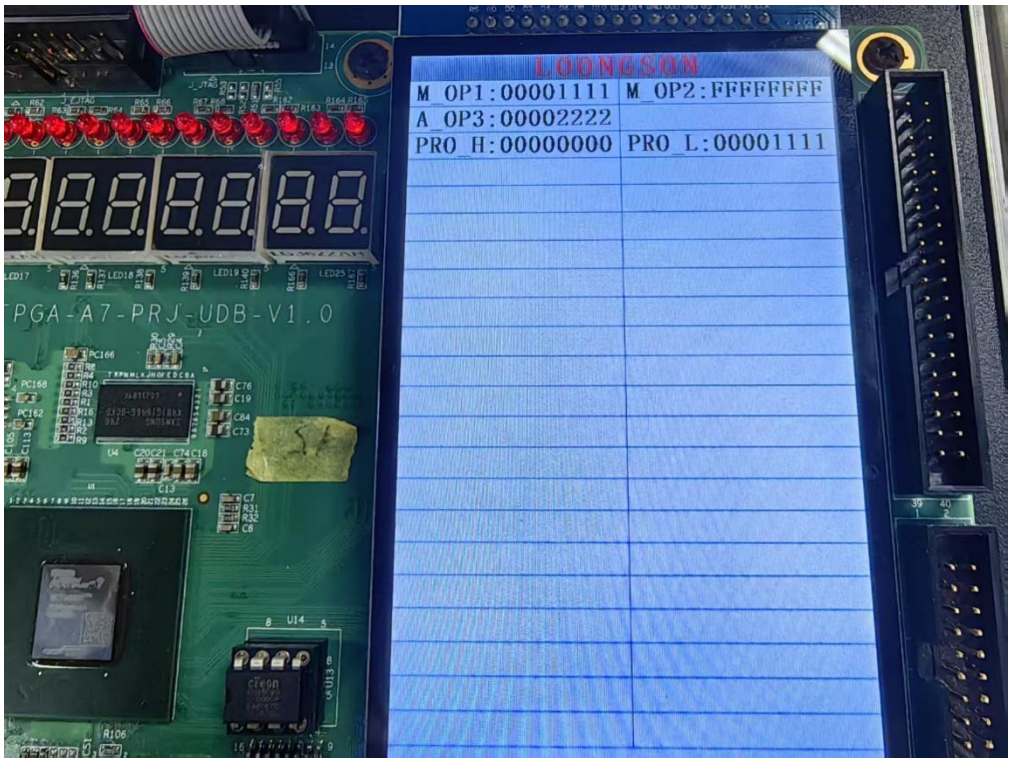
实验箱验证：

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

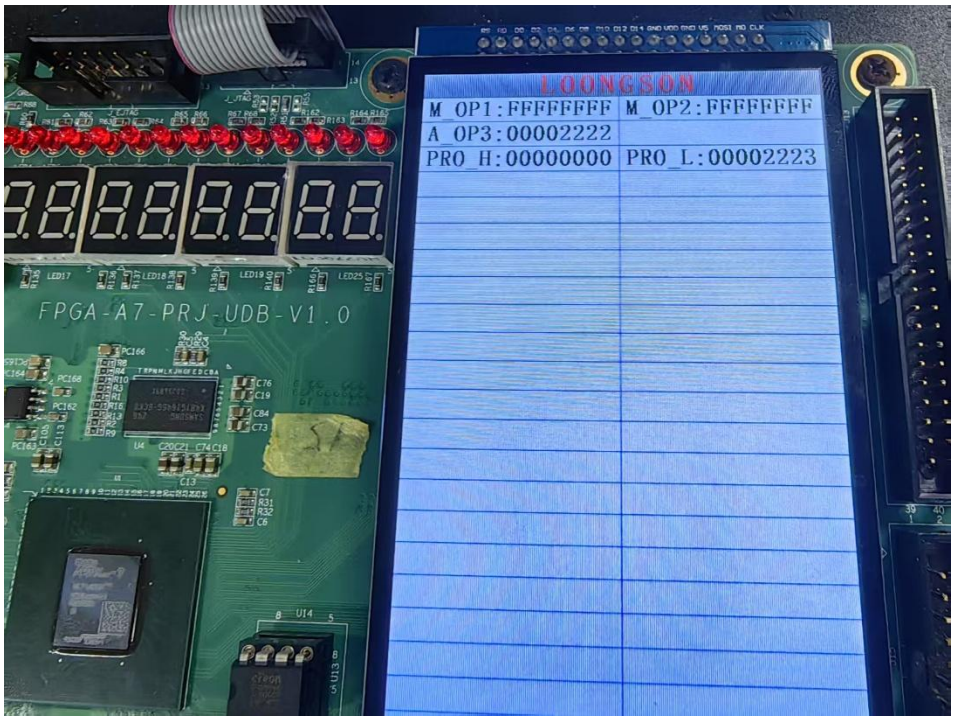
如图，1111*1111+2222，和波形图得到的答案是一样的，正确。

2. 正数乘负数



如图，1111*(-1)+2222,最后答案显然是 1111，正确。

3. 负数乘负数



-1*(-1)+2222,最后答案是 2223，显然是正确的。

4. 超 32 位

5. 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。