计算机组成原理实验课程第二次实验报告

实验名称：定点乘法器优化

学号：2312966 姓名：林晖鹏 班次：张金老师

一、实验目的

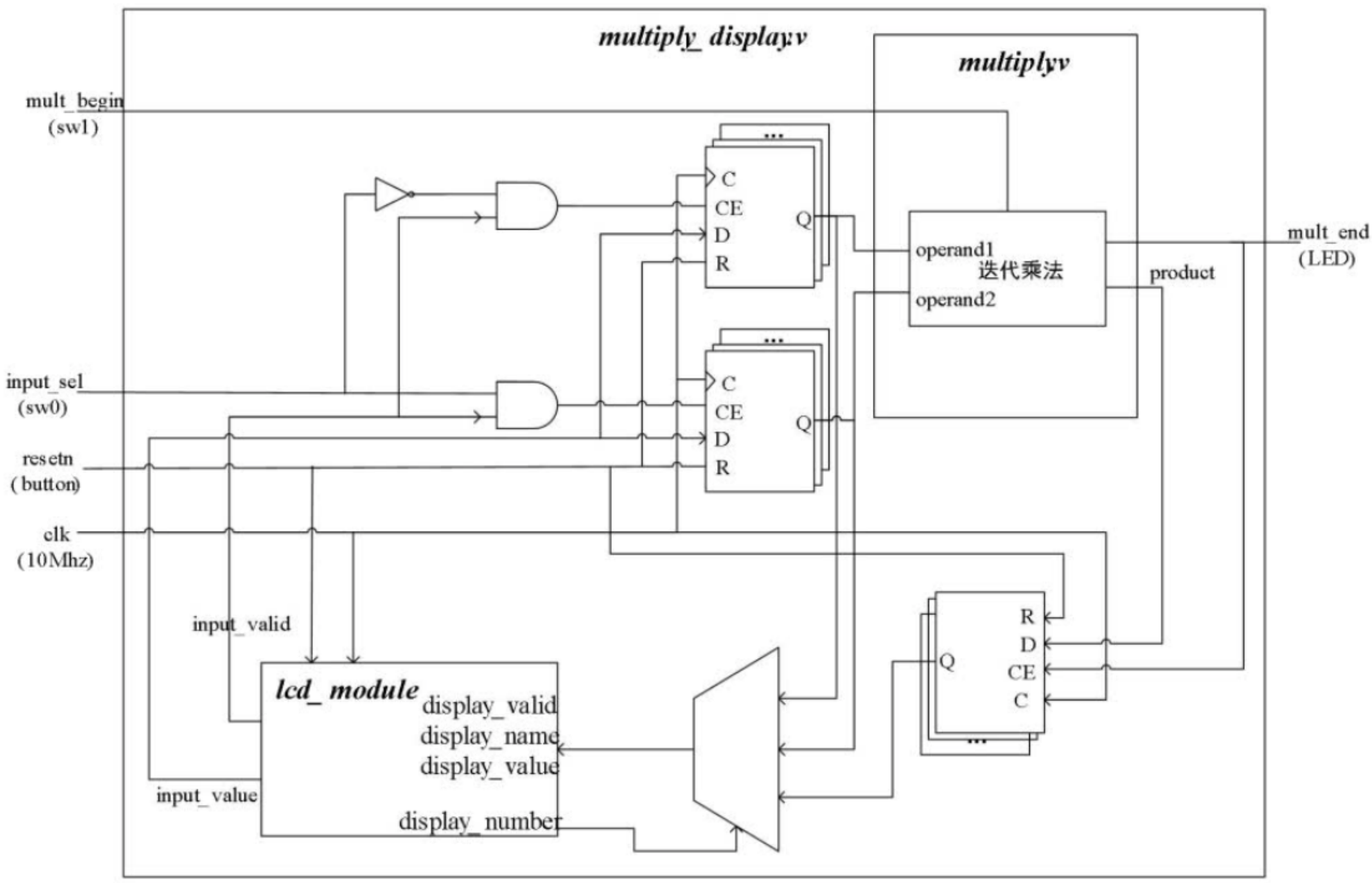
- 理解迭代乘法的实现算法的原理，掌握基本实现算法。
- 了解并尝试对迭代乘法算法进行优化。

二、实验内容说明

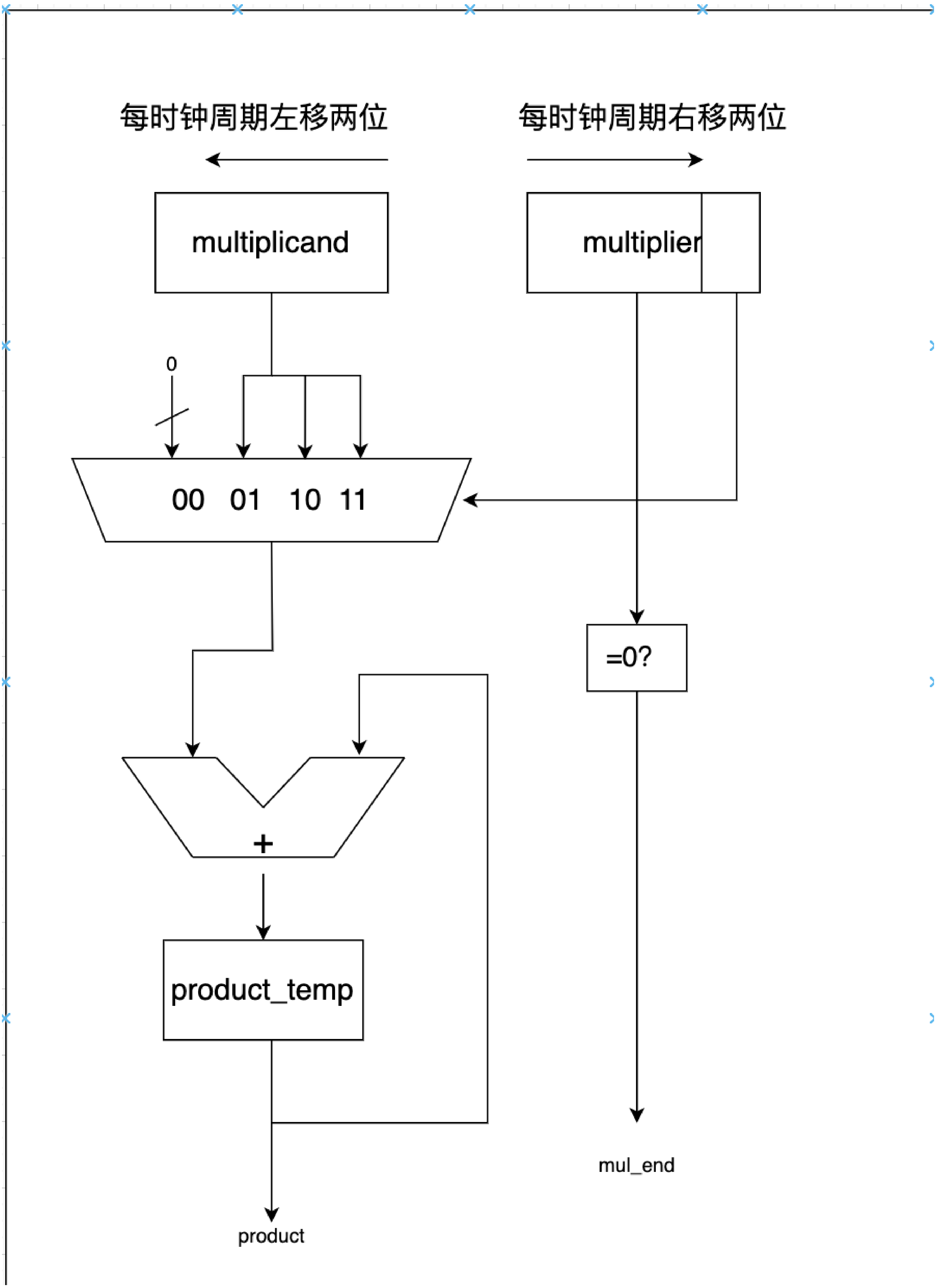
a. 在迭代乘法算法的基础上修改代码，实现二位迭代乘法。
b. 对修改后的代码进行仿真实验
c. 对修改后的代码进行上实验箱测试功能，并且结果显示在触摸屏的5-8位上面。
d. 学习其他优化算法，并尝试用verilog实现，提高算法效率

三、实验原理图

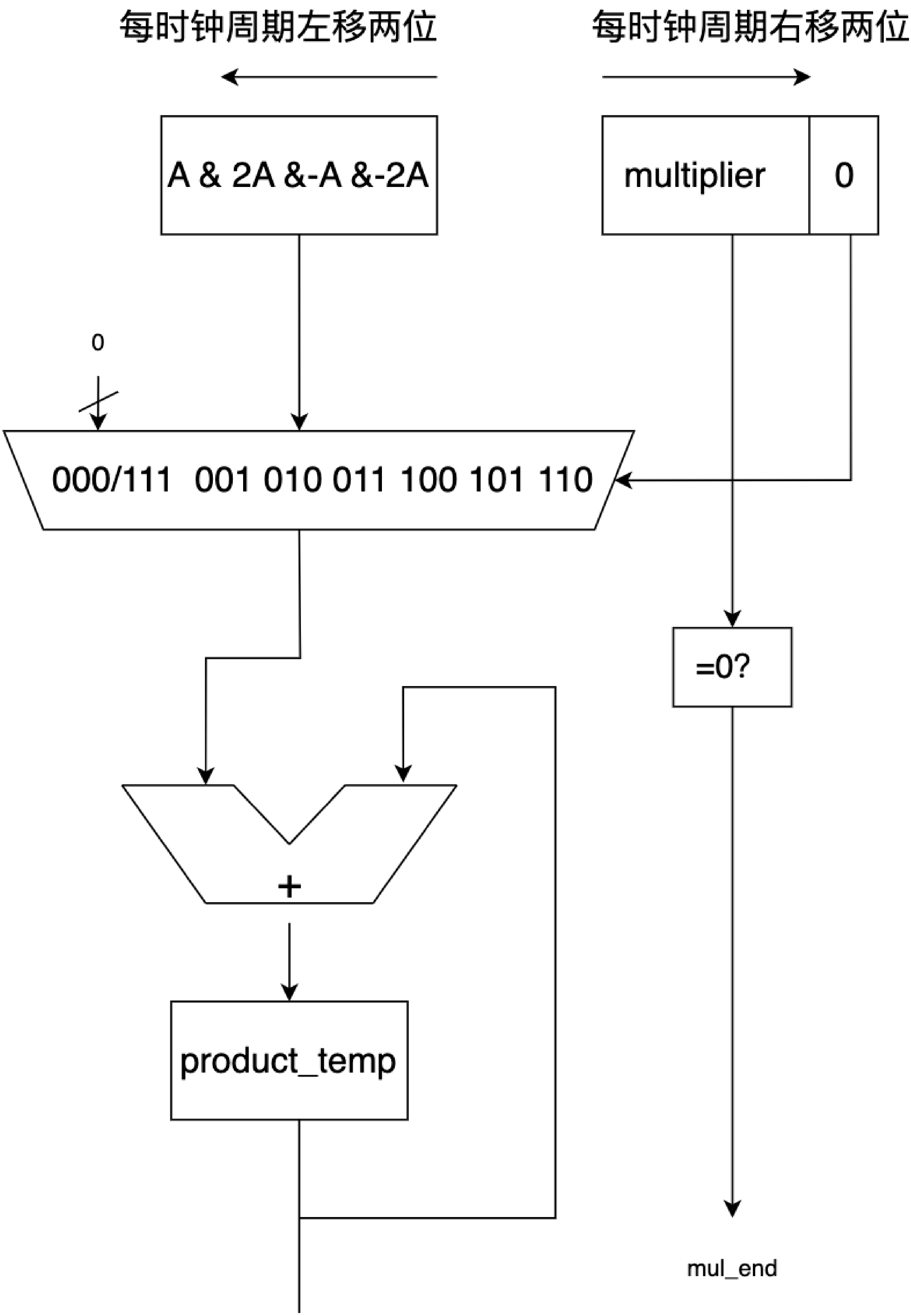
顶层模块和源代码相同，主要在迭代乘法处进行修改。



下面是二位迭代乘法的原理图：



下面是booth算法的原理图：


每时钟周期左移两位
每时钟周期右移两位
A & 2A &-A &-2A
multiplier
0
0
000/111 001 010 011 100 101 110
=0?
+
product_temp
mul_end

四、实验步骤（两位乘法）

算法思路：

> 👍 在迭代乘法的基础上，我们每个周期读取乘数的低两位，并计算此时的部分积，加到结果里面。这样总周期数将会减少一半。

1. multiply.v模块修改

模块功能解释：

本模块输入为两个32位的有符号数，输出一个64位的有符号数，用迭代乘法的算法实现一个32位乘法器。

代码修改：

我们这里只是把算法的计算过程方法修改，希望减少时钟周期花费的时间，所以本模块的输入输出设置无需修改。

在迭代算法中，每算完一个周期，被乘数左移一位，乘数右移一位，本质上是那乘数的每一位去乘被乘数，被乘数左移是因为乘数位的位高逐渐变高。而在我们修改的两位乘法中，每个周期我们移动两位而不是一位，来减少周期次数。这样就是拿乘数的两位去乘被乘数。

所以在这里，我们要修改每个周期乘数和被乘数移动的位数，如下面修改后的部分代码：

```
      multiplicand <= { multiplicand[61:0], 2'b00 };
      multiplier <= {2'b00,multiplier[31:2]};
```

在迭代乘法中，我们用一个部分积去计算乘数末尾乘上被乘数的结果，在两位乘法中，我们每个周期用两位去乘被乘数，所以这里我们仿照设计了两个部分积，并且高位的部分积要左移一位。如下面代码：

```
  // 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_product2;
```

```
    assign partial_product1 = multiplier[0] ? multiplicand : 64'd0;
    assign partial_product2 = multiplier[1] ? { multiplicand [62:0],1'b0 }:
64'd0;
```

在累加器处，我们由加一个部分积变成加两个部分积：

```
product_temp <= product_temp + partial_product1 + partial_product2;
```

2. multiply_display.v模块修改

这个模块是顶层模块，涉及将其他模块组合在一起实现功能。

应实验要求，我们要在触摸屏的5-8位显示结果而不在前四位，这里我们修改display_number中的数字来实现功能

```
    always @(posedge clk)
    begin
        case(display_number)
            6'd5 ://这里1改成5，下面修改地方相同
            begin
                display_valid <= 1'b1;
                display_name  <= "M_OP1";
                display_value <= mult_op1;
            end
            6'd6 :
            begin
                display_valid <= 1'b1;
                display_name  <= "M_OP2";
                display_value <= mult_op2;
            end
            6'd7 :
            begin
                display_valid <= 1'b1;
                display_name  <= "PRO_H";
                display_value <= product_r[63:32];
            end
            6'd8 :
            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
```

五、实验步骤（booth算法）

Radix 4-booth算法：

我们观察到，二位迭代虽然周期次数减少了，但是这里每个周期的加法器次数却是 2 ，其实加法器的调用并没有减少，下面我们介绍booth算法，希望来减少加法器的使用：

我们对B的补码展开，通过代数变换之后得到新的表达式，此时项数个数减少了一半。

$$B = -B_{n-1}2^{n-1} + (\sum_{i=0}^{n-2} B_i * 2^i)$$

$$= -B_{n-1}2^{n-1} + B_{n-2}2^{n-2} + B_{n-3}2^{n-3} + B_{n-4}2^{n-4} + \ldots\ldots +$$

$$B_3 2^3 + B_2 2^2 + B_1 2^1 + B_0 2^0 + B_{-1}$$

$$= (-2B_{n-1} + B_{n-2} + B_{n-3})2^{n-1} + (-2B_{n-3} + B_{n-4} + B_{n-5})2^{n-4} + \ldots\ldots +$$

$$(-2B_5 + B_4 + B_3)2^4 + (-2B_3 + B_2 + B_1)2^2 + (-2B_1 + B_0 + B_{-1})2^0$$

并且每一项的系数由B的补码里面的三位决定，决定的规则如下：

Bi+1	Bi	Bi-1	-2*(Bi+1)+Bi+Bi-1	部分积操作
0	0	0	+0	0
0	0	1	+1	1*A
0	1	0	+1	1*A
0	1	1	+2	2*A
1	0	0	-2	-2*A
1	0	1	-1	-1*A
1	1	0	-1	-1*A
1	1	1	-0	0

算法实现：

实现思路：

> 😀 仿照原来的二位迭代想法，我们每次左移被乘数，但这里我们不再计算A，而是A、-A、2A以及-2A，我们可以初始化的时候计算好这些数，然后每个周期左移这些数。
>
> 由于第一位系数由最低位和倒数第二位决定，我们可以在B的低位处扩展一个0，这样每次右移两位，检测结束则检测扩展后B的高32位（不考虑扩展的那位）。
>
> 我们设计一个 bit3 模块，根据B的低三位决定此时迭代乘法的系数。

1. bit3模块设计

模块功能：

由B的低三位来决定部分积为哪种情况，并输出部分积。

输入为：此时的A、-A、2A、-2A（随周期数这些数会逐渐左移）、B的低三位bit3。

输出为：部分积。

```
module bit3(
    input [2:0] bit3,
    input [63:0] inversed_A,
    input [63:0] inversed_2A,
    input [63:0] A,
    input [63:0] tow_A,
    output reg [63:0] partial_product
    );
    wire [2:0]b;
    assign b = bit3;

    always @(*) begin
       if (b[2] == 0) begin
           if (b[1:0] == 2'b00) begin
               partial_product = 64'd0;
           end else if (b[1:0] == 2'b10 || b[1:0] == 2'b01) begin
               partial_product = A;
           end else if (b[1:0] == 2'b11) begin
               partial_product = tow_A;
           end
       end
       else begin
           if (b[1:0] == 2'b00) begin
               partial_product = inversed_2A;
           end else if (b[1:0] == 2'b10 || b[1:0] == 2'b01) begin
```

```
                    partial_product = inversed_A;
                end else if (b[1:0] == 2'b11) begin
                    partial_product = 64'd0;
                end
            end
        end

endmodule
```

2. multiply模块修改

2.1 这里我们初始化A、-A、2A、-2A，并且每次周期结束都左移两位。

```
    //加载被乘数，运算时每次左移2位
    //reg  [63:0] multiplicand;
    reg [63:0] inversed_A;
    reg [63:0] inversed_2A;
    reg [63:0] A;
    reg [63:0] two_A;
    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2位
            //multiplicand <= { multiplicand[61:0], 2'b00 };
            A <= { A[61:0], 2'b00 };
            two_A <= { two_A[61:0], 2'b00 };
            inversed_A <= { inversed_A[61:0], 2'b00 };
            inversed_2A <= { inversed_2A[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
                A          <= {32'd0, op1_absolute};
                two_A      <= {31'd0, op1_absolute, 1'b0};
                inversed_A <= {32'hffffffff,~{op1_absolute}+1};
                inversed_2A <= {32'hffffffff,~{op1_absolute,1'b0}+1};
        end
    end

```

2.2 扩展乘数的低位

```
    //加载乘数，运算时每次右移2位
```

```
    reg  [32:0] 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[32:2]};
        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,1'b0}; //扩展低位
        end
    end
```

2.3 调用bit3模块计算部分积

```
        wire [63:0] partial_product;
        wire [2:0] bit3;
        assign bit3 = multiplier[2:0];
        bit3 bit(bit3,inversed_A,inversed_2A,A,two_A,partial_product);
```

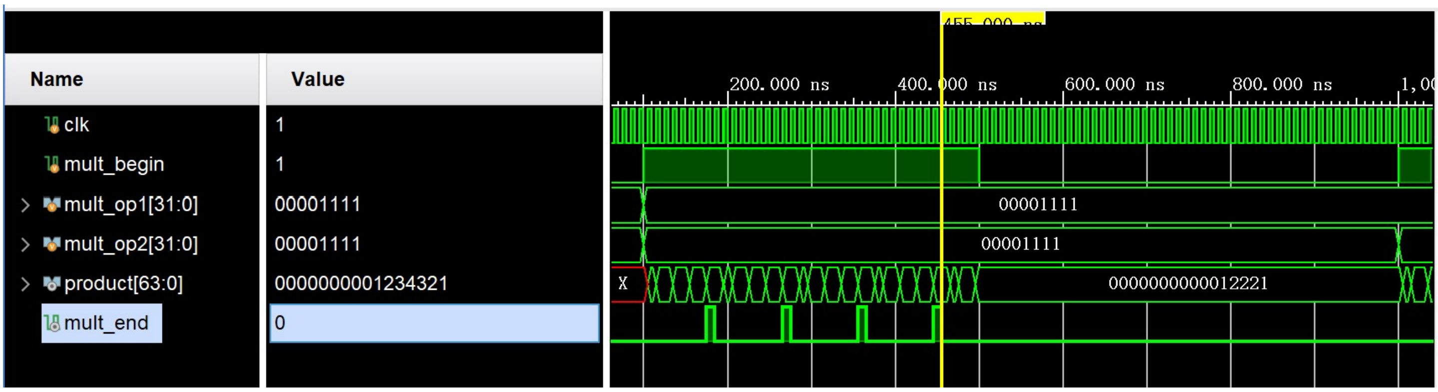
2.4 累加器每周期只需要执行一次加法

```
product_temp <= product_temp + partial_product;
```

六、实验结果分析

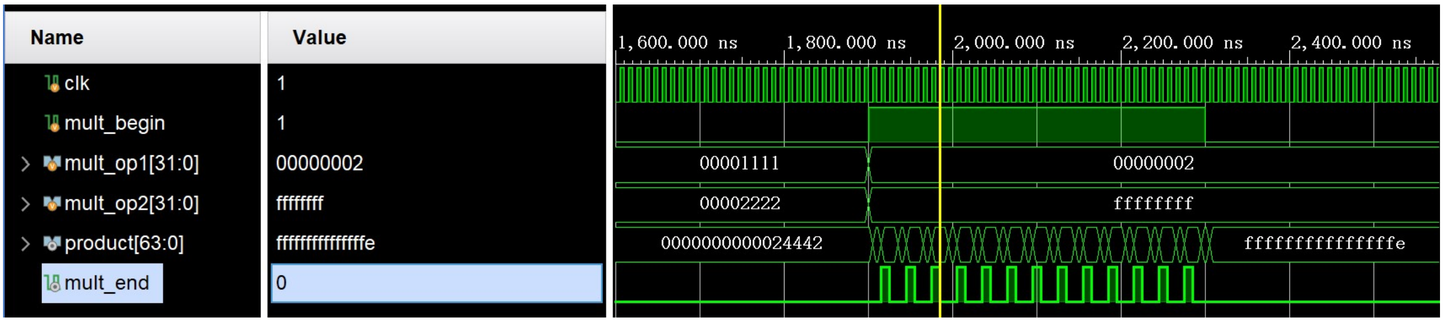
1. 二位迭代算法仿真结果分析

我们下面给出两个有代表性的例子：



二位迭代算法-例子一

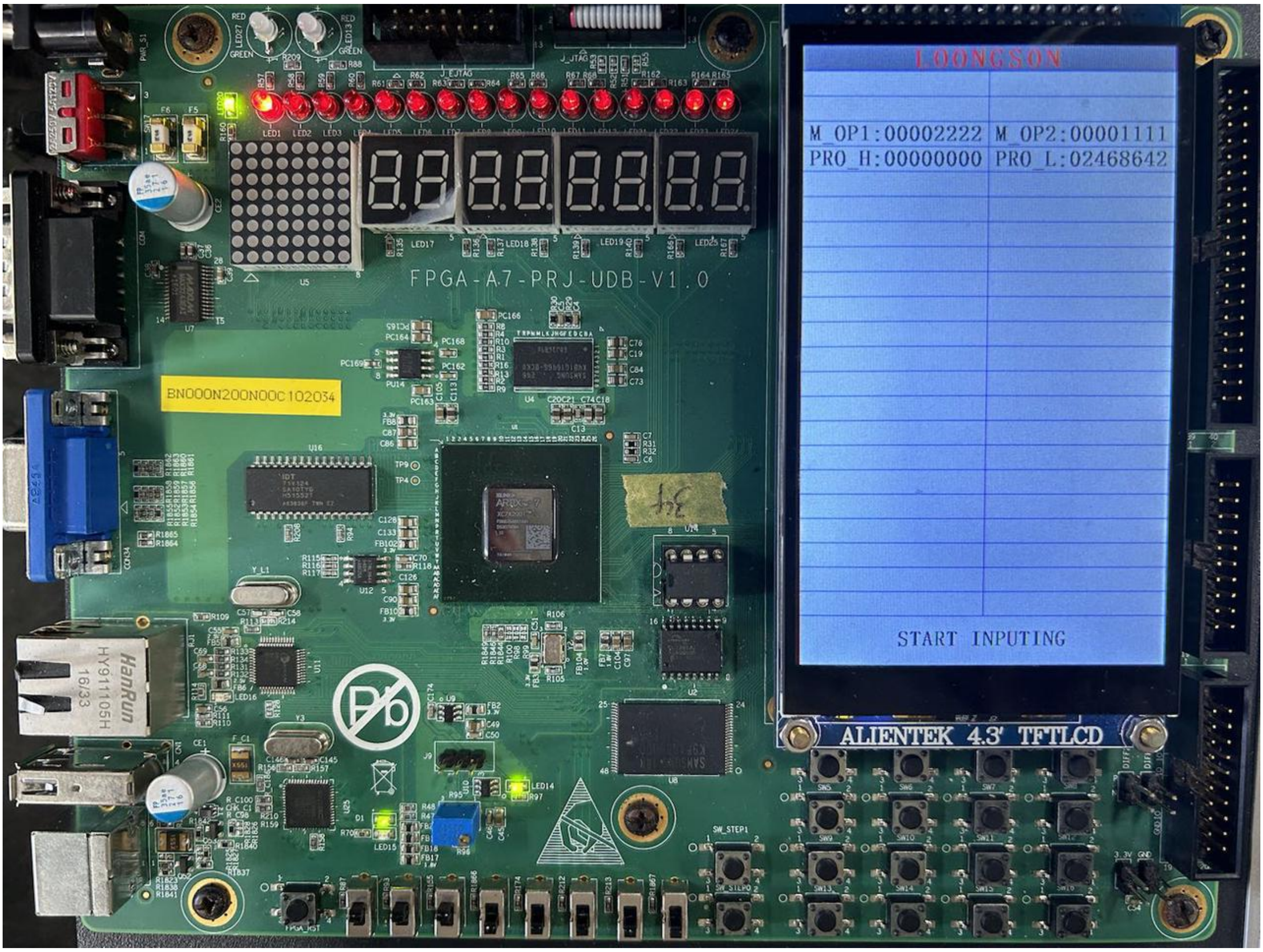
输入为{00001111}和{00001111}，输出为{0000000001234321}，结果正确。



二位迭代算法-例子二

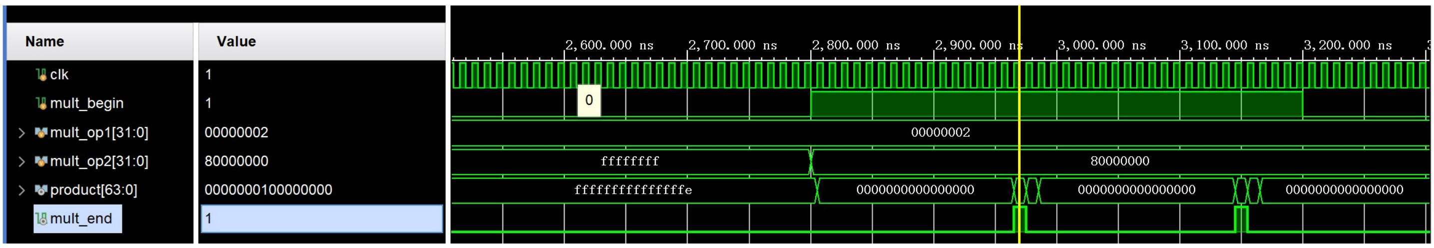
输入为{00000002}和{ffffffff}（实际上就是2*-1），得到结果为{fffffffffffffe}（也就是-2），结果正确。

2. 二位迭代算法上箱验证



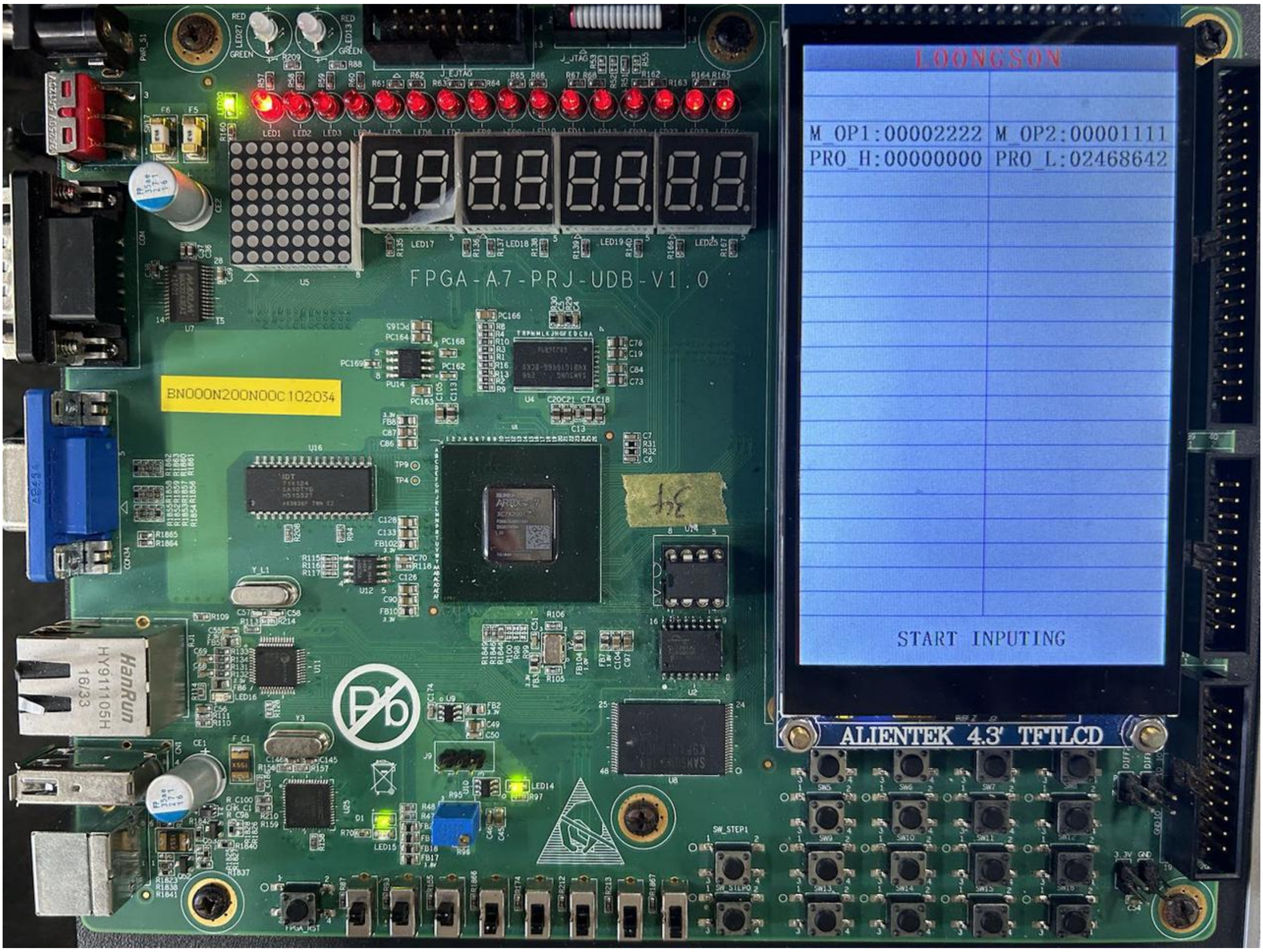
输入为{00002222}和{00001111}，结果为{0000000002468642}，结果正确。

3. booth算法仿真测试



这里输入为{00000002}和{800000000}，输出为{0000000100000000}，结果正确。

4. booth算法上箱验证



这里输入为{00001111}和{00002222}，输出为{0000000002468642}，结果正确。

七、总结感想

1. 本次实验流程比较久，主要是booth算法实现比较绕，因为我们这里代码的基础是先取绝对值，结果在取符号。和网上的做法有所区别，考察了二进制的绝对值、补码的知识。
2. 其实迭代算法本质上复杂度没有变化，只是减少了运行时间，如果进一步做基4、基8，还能进一步减少周期数，但是要保证在周期内能执行完计算。
3. 但是如果我们用booth算法，进一步做基8位，能在减少周期数的同时，减少加法器的调用，从而实现算法优化。但我个人感觉我在计算bit3模块的时候，写的不够底层，导致实际上那部分的复杂度并不比二位迭代的算法低，这里可以再考虑一下bit3模块的设计，尽量少使用器件。