组成原理课程第_二_次实报告

学号：_2310420_ 姓名：_王晶_ 班次：_李涛老师_

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。

实验内容说明

（1）复现 32 位乘法器

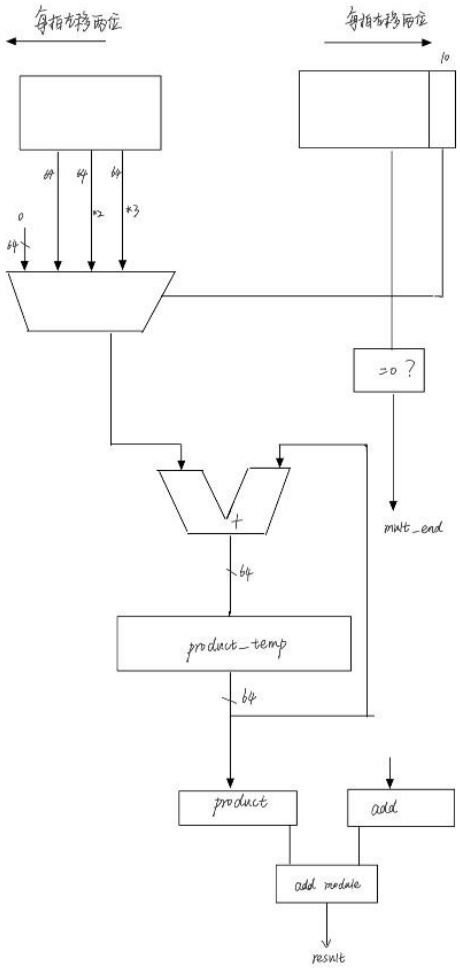
复现指导书中的 32 为乘法器，使用两个 32 为寄存器，将其中的内容相乘，存储在一个 64 为寄存器中，并在实验箱上测试。

（2）改进乘法器

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）。

实验原理图



实验步骤

（1）mac 模块的修改

代码部分

```
verilog
`timescale 1ns / 1ps
//************************************************************************
**
//   > 文件名: mac.v
//   > 描述   ：乘加器模块（MAC），基于迭代乘法器，实现乘法累加运算
//************************************************************************
**
module mac (
    input clk,
```

```
    input mac_begin,
    input [31:0] mult_op1,
    input [31:0] mult_op2,
    input [63:0] accumulator,  // 累加器输入
    output [63:0] product,
    output mac_end
);

    reg mac_valid;
    assign mac_end = mac_valid & ~(|multiplier);

    always @(posedge clk) begin
        if (!mac_begin || mac_end)
            mac_valid <= 1'b0;
        else
            mac_valid <= 1'b1;
    end

    wire op1_sign;
    wire op2_sign;
    wire [31:0] op1_absolute;
    wire [31:0] op2_absolute;
    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 (mac_valid)
            multiplicand <= {multiplicand[61:0], 2'b00};
        else if (mac_begin)
            multiplicand <= {32'd0, op1_absolute};
    end

    reg [31:0] multiplier;
    always @(posedge clk) begin
        if (mac_valid)
            multiplier <= {2'b00, multiplier[31:2]};
        else if (mac_begin)
```

```
            multiplier <= op2_absolute;
    end

    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;

    reg [63:0] product_temp;
    always @(posedge clk) begin
        if (mac_valid)
            product_temp <= product_temp + partial_product1 +
partial_product2;
        else if (mac_begin)
            product_temp <= accumulator;  // 初始值为累加器输入值
    end

    reg product_sign;
    always @(posedge clk) begin
        if (mac_valid)
            product_sign <= op1_sign ^ op2_sign;
    end

    assign product = product_sign ? (~product_temp + 1) : product_temp;

endmodule
```

代码解释：

这一部分代码实现了乘加操作，首先对输入的两个数进行相乘，之后将结果再加上一个数后进行输出。

在乘法部分，要求移动的乘数和输出的结果由一位变成两位，实现时钟周期的缩短。

```
verilog
module mac (
    input clk,
    input mac_begin,
```

```
    input [31:0] mult_op1,
    input [31:0] mult_op2,
    input [63:0] accumulator,  // 累加器输入
    output [63:0] product,
    output mac_end
);
```

这一部分定义了输入和输出的变量，首先是三位输入，乘法为 32 位，加法为 64 位，输出结果包括两个，product 和 mac_end，product 是最后的结果，而 mac_end 则是指示乘法过程是否结束，如果其值为 1，说明乘法结束。

```
verilog
reg mac_valid;
    assign mac_end = mac_valid & ~(|multiplier);

    always @(posedge clk) begin
        if (!mac_begin || mac_end)
            mac_valid <= 1'b0;
        else
            mac_valid <= 1'b1;
    end
```

这一段代码定义了 mac_end 的值，首先通过一个 always 块控制 mac_valid，其值为 0 时运算不进行，那么当 mac_valid 有效且 multiplier 所有位都是 0 时，mac_end 有效，运算结束。

```
verilog
reg [63:0] multiplicand;
    always @(posedge clk) begin
        if (mac_valid)
            multiplicand <= {multiplicand[61:0], 2'b00};
        else if (mac_begin)
            multiplicand <= {32'd0, op1_absolute};
    end
```

这一段代码用于处理 multiplicand，声明了一个 64 位的寄存器 multiplicand，用于存储被乘数，在时钟上升沿，如果 mac_valid 有效，则将 multiplicand 左移 2 位。

```
verilog
reg product_sign;
    always @(posedge clk) begin
```

```
        if (mac_valid)
            product_sign <= op1_sign ^ op2_sign;
    end

    assign product = product_sign ? (~product_temp + 1) : product_temp;
```

这一段代码用于处理 multiplier，首先声明一个 32 位的寄存器，用于存储乘数，在时钟上升沿，如果 mac_valid 有效，则将 multiplier 右移两位

```
verilog
 reg [63:0] product_temp;
    always @(posedge clk) begin
        if (mac_valid)
            product_temp <= product_temp + partial_product1 +
partial_product2;
        else if (mac_begin)
            product_temp <= accumulator;  // 初始值为累加器输入值
    end
```

这段代码声明了一个 64 位的寄存器 product_temp，用于临时存储乘加运算的中间结果，在时钟上升沿，如果 mac_valid 有效，则将 product_temp 更新为当前值加上两个部分积

```
verilog
 reg product_sign;
    always @(posedge clk) begin
        if (mac_valid)
            product_sign <= op1_sign ^ op2_sign;
    end

    assign product = product_sign ? (~product_temp + 1) : product_temp;
```

这段代码声明了一个 product_sign,用于储存最终结果的符号，product 用于最后结果的输出。

（2）仿真文件的修改

代码部分：

```
verilog
```

```
`timescale 1ns / 1ps

module tb_mac;

    // Inputs
    reg clk;
    reg mac_begin;
    reg [31:0] mult_op1;
    reg [31:0] mult_op2;
    reg [63:0] accumulator;  // 累加器输入

    // Outputs
    wire [63:0] product;
    wire mac_end;

    // Instantiate the Unit Under Test (UUT)
    mac uut (
        .clk(clk),
        .mac_begin(mac_begin),
        .mult_op1(mult_op1),
        .mult_op2(mult_op2),
        .accumulator(accumulator),
        .product(product),
        .mac_end(mac_end)
    );

    initial begin
        // Initialize Inputs
        clk = 0;
        mac_begin = 0;
        mult_op1 = 0;
        mult_op2 = 0;
        accumulator = 64'd0;  // 初始化累加器输入

        // Wait 100 ns for global reset to finish
        #100;

        // Test Case 1: 乘法累加
        mac_begin = 1;
        mult_op1 = 32'H00001111;
```

```
        mult_op2 = 32'H00001111;
        accumulator = 64'H0000000000000010; // 初始累加值为 16
        #40;
        mac_begin = 0;
        #50;

        // Test Case 2: 乘法累加不同的数
        mac_begin = 1;
        mult_op1 = 32'H00001111;
        mult_op2 = 32'H00002222;
        accumulator = 64'H0000000000000020; // 初始累加值为 32
        #400;
        mac_begin = 0;
        #50;

        // Test Case 3: 负数乘法累加
        mac_begin = 1;
        mult_op1 = 32'H00000002;
        mult_op2 = 32'HFFFFFFFF;  // 负数
        accumulator = 64'H0000000000000030; // 初始累加值为 48
        #40;
        mac_begin = 0;
        #50;

        // Test Case 4: 一个正数和一个负数的乘法累加
        mac_begin = 1;
        mult_op1 = 32'H00000002;
        mult_op2 = 32'H80000000;  // 负数
        accumulator = 64'H0000000000000040; // 初始累加值为 64
        #40;
        mac_begin = 0;
        #50;

        // 其他测试用例可在此添加
    end

    // Clock Generation
    always #5 clk = ~clk;

endmodule
```

代码解释：

上述仿真文件一共采取四个测试样例，分别为

```
Verilog
mult_op1 = 32'H00001111;
mult_op2 = 32'H00001111;
accumulator = 64'H0000000000000010;
```

```
Verilog
mult_op1 = 32'H00001111;
mult_op2 = 32'H00002222;
accumulator = 64'H0000000000000020;
```

```
Verilog
mult_op1 = 32'H00000002;
mult_op2 = 32'HFFFFFFFF;  // 负数
accumulator = 64'H0000000000000030;
```

```
Verilog
mult_op1 = 32'H00000002;
mult_op2 = 32'H80000000;  // 负数
accumulator = 64'H0000000000000040;
```

（3）display 模块的修改

代码部分：

```
verilog
`timescale 1ns / 1ps
//************************************************************************
**
//   > 文件名: mac_display.v
//   > 描述   ：乘加器显示模块，调用 FPGA 板上的 IO 接口和触摸屏
//   > 作者   : LOONGSON
//   > 日期   : 2016-04-14
```

```
//************************************************************************
**
module mac_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,
    //乘加运算结束信号
    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
);
//-----{调用乘加器模块}begin
    wire        mac_begin;
    reg  [31:0] mac_op1;
    reg  [31:0] mac_op2;    reg  [63:0] accumulator;
    wire [63:0] product;
    wire        mac_end;    assign mac_begin = sw_begin;
    assign led_end = mac_end;
    mac mac_module (
        .clk        (clk        ),
        .mac_begin  (mac_begin  ),
        .mult_op1    (mac_op1    ),
        .mult_op2    (mac_op2    ),
        .accumulator(accumulator),
        .product    (product    ),
```

```
        .mac_end    (mac_end    )
    );
    reg [63:0] product_r;
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            product_r <= 64'd0;
        end
        else if (mac_end)
        begin
            product_r <= product;
        end
    end
//-----{调用乘加器模块}end

//----------------------{调用触摸屏模块}begin---------------------//
    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        ),
        .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_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       )
);

always @(posedge clk)
begin
    if (!resetn)
    begin
        mac_op1 <= 32'd0;
    end
    else if (input_valid && !input_sel)
    begin
        mac_op1 <= input_value;
    end
end
always @(posedge clk)
begin
    if (!resetn)
    begin
        mac_op2 <= 32'd0;
    end
    else if (input_valid && input_sel)
    begin
        mac_op2 <= input_value;
    end
end

always @(posedge clk)
begin
    if (!resetn)
    begin
        accumulator <= 64'd0;
    end
    else if (input_valid)
    begin
        accumulator <= {32'd0, input_value};
    end
end
```

```
always @(posedge clk)
begin
    case(display_number)
        6'd1 :
        begin
            display_valid <= 1'b1;
            display_name  <= "M_OP1";
            display_value <= mac_op1;
        end
        6'd2 :
        begin
            display_valid <= 1'b1;
            display_name  <= "M_OP2";
            display_value <= mac_op2;
        end
        6'd3 :
        begin
            display_valid <= 1'b1;
            display_name  <= "ACCUM";
            display_value <= accumulator[31:0];
        end
        6'd4 :
        begin
            display_valid <= 1'b1;
            display_name  <= "PRO_H";
            display_value <= product_r[63:32];
        end
        6'd5 :
        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
endmodule
```

代码解释：

```
verilog
module mac_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,
    //乘加运算结束信号
    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
);
```

上述代码时 mac_display 模块定义部分和相关的触摸屏的端口声明

```
verilog
//-----{调用乘加器模块}begin
    wire        mac_begin;
    reg  [31:0] mac_op1;
    reg  [31:0] mac_op2;
    reg  [63:0] accumulator;
    wire [63:0] product;
```

```
    wire        mac_end;
    assign mac_begin = sw_begin;
    assign led_end = mac_end;
    mac mac_module (
        .clk        (clk        ),
        .mac_begin  (mac_begin  ),
        .mult_op1    (mac_op1    ),
        .mult_op2    (mac_op2    ),
        .accumulator(accumulator),
        .product    (product    ),
        .mac_end    (mac_end    )
    );
    reg [63:0] product_r;
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            product_r <= 64'd0;
        end
        else if (mac_end)
        begin
            product_r <= product;
        end
    end
//-----{调用乘加器模块}end
//--------------------{调用触摸屏模块}begin--------------------//
    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       ),
        .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_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       )
    );
```

上述代码分别调用和实例化乘加器模块和触摸屏模块

```
verilog
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            mac_op1 <= 32'd0;
        end
        else if (input_valid && !input_sel)
        begin
            mac_op1 <= input_value;
        end
    end
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            mac_op2 <= 32'd0;
        end
        else if (input_valid && input_sel)
        begin
            mac_op2 <= input_value;
        end
    end
```

```
    always @(posedge clk)
    begin
        if (!resetn)
        begin
            accumulator <= 64'd0;
        end
        else if (input_valid)
        begin
            accumulator <= {32'd0, input_value};
        end
    end
```

这一部分位操作数和累加器的赋值逻辑，第一个 always 块：在时钟上升沿，如果复位信号 resetn 有效，则将乘数 1 mac_op1 清零；如果输入有效信号 input_valid 有效且 input_sel 为 0，则将输入值 input_value 赋值给 mac_op1。第二个 always 块：在时钟上升沿，如果复位信号 resetn 有效，则将乘数 2 mac_op2 清零；如果输入有效信号 input_valid 有效且 input_sel 为 1，则将输入值 input_value 赋值给 mac_op2。第三个 always 块：在时钟上升沿，如果复位信号 resetn 有效，则将累加器 accumulator 清零；如果输入有效信号 input_valid 有效，则将输入值 input_value 扩展为 64 位后赋值给 accumulator。

（5）约束文件的修改

代码部分：

```
verilog
set_property PACKAGE_PIN AC19 [get_ports clk]
set_property PACKAGE_PIN H7   [get_ports led_end]
set_property PACKAGE_PIN Y3   [get_ports resetn]
set_property PACKAGE_PIN AC21 [get_ports input_sel]
set_property PACKAGE_PIN AD24 [get_ports sw_begin]

set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports led_end]
set_property IOSTANDARD LVCMOS33 [get_ports resetn]
set_property IOSTANDARD LVCMOS33 [get_ports input_sel]
set_property IOSTANDARD LVCMOS33 [get_ports sw_begin]
```

```
# 触摸屏引脚连接
set_property PACKAGE_PIN J25 [get_ports lcd_rst]
set_property PACKAGE_PIN H18 [get_ports lcd_cs]
set_property PACKAGE_PIN K16 [get_ports lcd_rs]
set_property PACKAGE_PIN L8 [get_ports lcd_wr]
set_property PACKAGE_PIN K8 [get_ports lcd_rd]
set_property PACKAGE_PIN J15 [get_ports lcd_bl_ctr]
set_property PACKAGE_PIN H9 [get_ports {lcd_data_io[0]}]
set_property PACKAGE_PIN K17 [get_ports {lcd_data_io[1]}]
set_property PACKAGE_PIN J20 [get_ports {lcd_data_io[2]}]
set_property PACKAGE_PIN M17 [get_ports {lcd_data_io[3]}]
set_property PACKAGE_PIN L17 [get_ports {lcd_data_io[4]}]
set_property PACKAGE_PIN L18 [get_ports {lcd_data_io[5]}]
set_property PACKAGE_PIN L15 [get_ports {lcd_data_io[6]}]
set_property PACKAGE_PIN M15 [get_ports {lcd_data_io[7]}]
set_property PACKAGE_PIN M16 [get_ports {lcd_data_io[8]}]
set_property PACKAGE_PIN L14 [get_ports {lcd_data_io[9]}]
set_property PACKAGE_PIN M14 [get_ports {lcd_data_io[10]}]
set_property PACKAGE_PIN F22 [get_ports {lcd_data_io[11]}]
set_property PACKAGE_PIN G22 [get_ports {lcd_data_io[12]}]
set_property PACKAGE_PIN G21 [get_ports {lcd_data_io[13]}]
set_property PACKAGE_PIN H24 [get_ports {lcd_data_io[14]}]
set_property PACKAGE_PIN J16 [get_ports {lcd_data_io[15]}]
set_property PACKAGE_PIN L19 [get_ports ct_int]
set_property PACKAGE_PIN J24 [get_ports ct_sda]
set_property PACKAGE_PIN H21 [get_ports ct_scl]
set_property PACKAGE_PIN G24 [get_ports ct_rstn]

set_property IOSTANDARD LVCMOS33 [get_ports lcd_rst]
set_property IOSTANDARD LVCMOS33 [get_ports lcd_cs]
set_property IOSTANDARD LVCMOS33 [get_ports lcd_rs]
set_property IOSTANDARD LVCMOS33 [get_ports lcd_wr]
set_property IOSTANDARD LVCMOS33 [get_ports lcd_rd]
set_property IOSTANDARD LVCMOS33 [get_ports lcd_bl_ctr]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[4]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {lcd_data_io[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports ct_int]
set_property IOSTANDARD LVCMOS33 [get_ports ct_sda]
set_property IOSTANDARD LVCMOS33 [get_ports ct_scl]
set_property IOSTANDARD LVCMOS33 [get_ports ct_rstn]
```

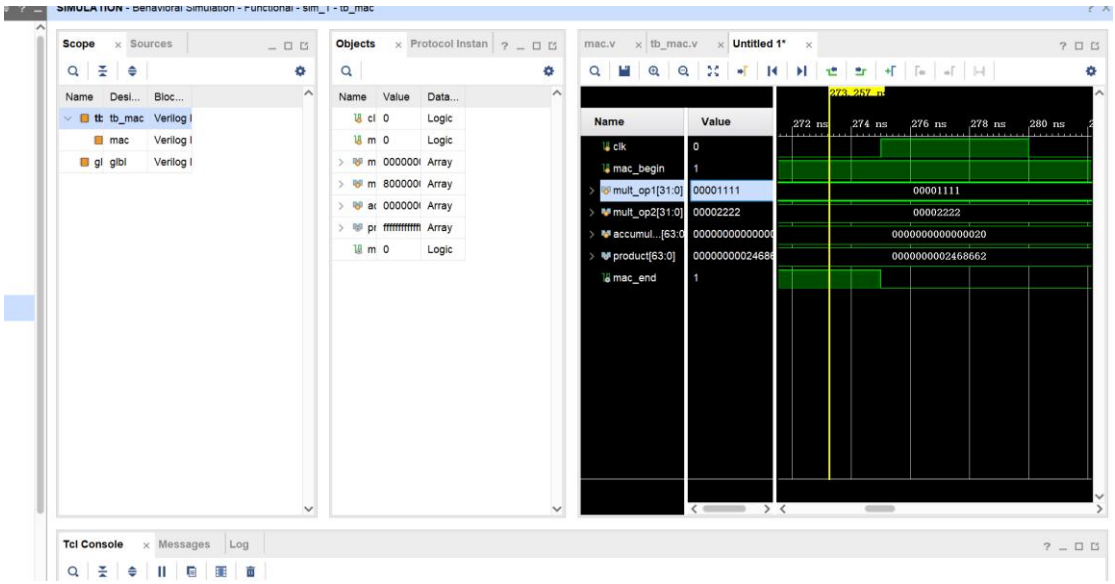
代码解释：

```
verilog
set_property PACKAGE_PIN AC19 [get_ports clk]
set_property PACKAGE_PIN H7   [get_ports led_end]
set_property PACKAGE_PIN Y3   [get_ports resetn]
set_property PACKAGE_PIN AC21 [get_ports input_sel]
set_property PACKAGE_PIN AD24 [get_ports sw_begin]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports led_end]
set_property IOSTANDARD LVCMOS33 [get_ports resetn]
set_property IOSTANDARD LVCMOS33 [get_ports input_sel]
set_property IOSTANDARD LVCMOS33 [get_ports sw_begin]
```

这一部分是时钟和控制信号的引脚分配

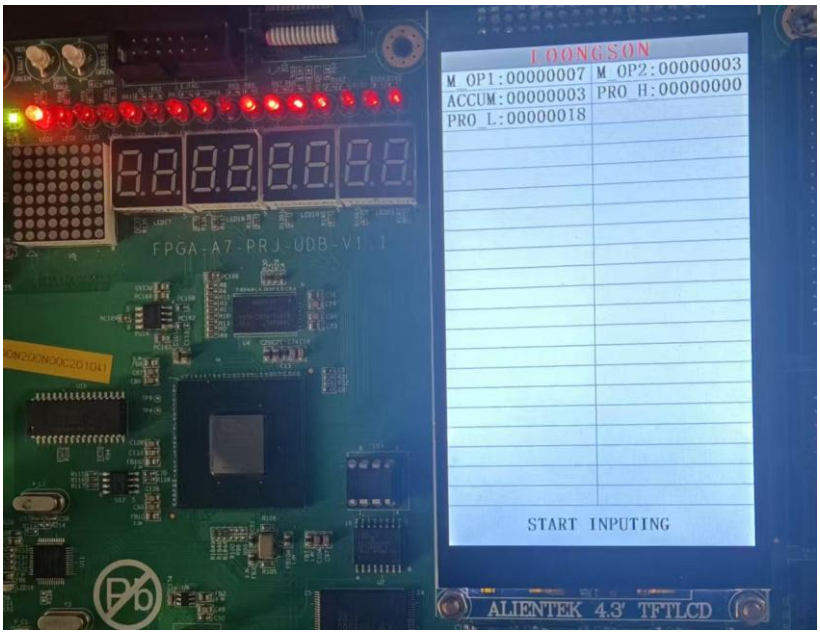
实验结果分析

仿真结果如下：



观察到乘数 1 为 00001111，乘数 2 为 00002222，加数为 0000000000000020，结果为 0000000002468662

上箱验证：



乘数 1 为 00000007，乘数 2 为 00000003，加数为 0000003，最后结果为 00000018，符合要求

总结感想

（1）通过本次实验，我进一步学习理解了计算机中定点加法器的多种实现算法的原理，掌握了迭代乘法的实现算法。

（2）通过本次实验，我进一步对 Verilog 语句有了深入的理解，对如何编写仿真文件和约束文件，对每一行代码的含义都进行掌握。对实验箱的用法进一步了解。

（3）本次实验中，采用两次移位，达到了乘法计算更快，效率更高的效果，本次实验要求实现的乘法为有符号的乘法，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要是补码的形式。

（4）本次实验中，我认识到对于底层电路设计而言，需要先构思出大致的顶层模块图与大致的实现方法，然后再开始编写代码。