组成原理实验课程第_2_次实验报告

实验名称	数据运算：定点乘法			班级	李涛
学生姓名	齐明杰	学号	2113997	指导老师	董前琨
实验地点	A306		实验时间	2023.4.4	

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。

2. 实验内容说明

针对组成原理第二次的乘法法器实验(设计出初始的乘法器，能够实现对两个 32 为操作数的乘法运算，并且将结果保存在一个 64 位的寄存器中输出)进行改进，要求：

1、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。（其他形式的优化也建议尝试）

2、将改进后的乘法器进行仿真验证

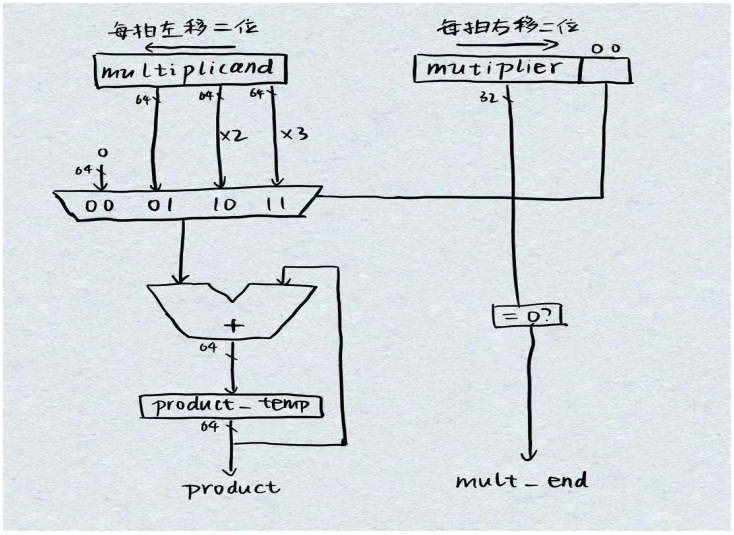
3、将改进后的乘法器进行上实验箱验证，上箱验证时调整数据不在前 4 格显示

4、实验报告中的原理图为迭代乘法的算法图，不再是顶层模块图

注：要求在初始的乘法器的基础上进行改进，让算法能够在 16 个时钟节拍之内完成，提高乘法效率

3. 实验原理图

迭代乘法算法原理图如下:



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。改进后的迭代乘法在模拟乘法的过程中，乘数每次右移两位，根据最低两位，判断是加被乘数的 1 倍，2 倍，3 倍还是加 0，不停地累加得到最终乘积。迭代乘法是用多次加法完成乘法操作的，需要多拍时间，其结束标志为乘数移位后为 0。

4. 实验步骤

项目代码(修改处用红色标记):

(1) multiply.v

```
`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
    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，运算时每次左移二位
    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二位
            multiplicand <= {multiplicand[61:0],2'b00};
```

注：这句代码的意思是把两位 0“拼接到”被乘数的低 62 位(高 2 位由于左移舍去)的右边，再赋值给被乘数，逻辑上相当于左移 2 位。

```
        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，运算时每次右移二位
    reg   [31:0] multiplier;
    always @ (posedge clk)
    begin
        if (mult_valid)
        begin     // 如果正在进行乘法，则乘数每时钟右移二位
            multiplier <= {2'b00,multiplier[31:2]};
```

注：与上面同理，代码将两位 0 拼接到乘数高 30 位的左边，再赋值给乘数，相当于右移两位。

```
        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_product2;
    assign partial_product1 = multiplier[0] ? multiplicand : 64'd0;
    assign partial_product2 = multiplier[1] ? {multiplicand[62:0], 1'b0} : 64'd0;
```

注：原先的二路选择器，由于位移为 2 位而不是 1 位，现在需要四路选择器(即考虑 00，01，10，11)，分别对应部分积要加上全 0，被乘数，2 倍被乘数，3 倍被乘数。

据此我们可以声明两个 64 位变量 partial_product1， partial_product2。(下面简记为 p1,p2)

对于 p1,若乘数最低位为 1，则等于乘数，否则等于 0(代码使用三目运算符表示)

对于 p2,若乘数倒二位为 1，则等于两倍的乘数(采用与前面相同的左移方法表示，即 multiplicand[62:0], 1'b0)，否则等于 0

```
    //累加器
    reg [63:0] product_temp;
    always @ (posedge clk)
    begin
        if (mult_valid)
```

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

注：接上文，我们不难发现乘数的低 2 位决定了 p1 和 p2 各自的取值，在此将其相加，便完成了所需的四路选择器：

① **00：此时 p1=p2=0，部分积不变**
② **01：此时 p1=乘数，p2=0，部分积加上 1 倍乘数**
③ **10：此时 p1=0，p2=2 倍乘数，部分积加上 2 倍乘数**
④ **11：此时 p1=乘数，p2=2 倍乘数，部分积加上 3 倍乘数**

```
            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

```
        //************************************************************************
        //    > 文件名: multiply_display.v
        //    > 描述   : 乘法器显示模块，调用 FPGA 板上的 IO 接口和触摸屏
        //    > 作者   : LOONGSON
        //    > 日期   : 2016-04-14
        //************************************************************************
        module 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'd7 :
            begin
                display_valid <= 1'b1;
                display_name  <= "M_OP1";
                display_value <= mult_op1;
            end
            6'd8 :
            begin
                display_valid <= 1'b1;
                display_name  <= "M_OP2";
                display_value <= mult_op2;
            end
```

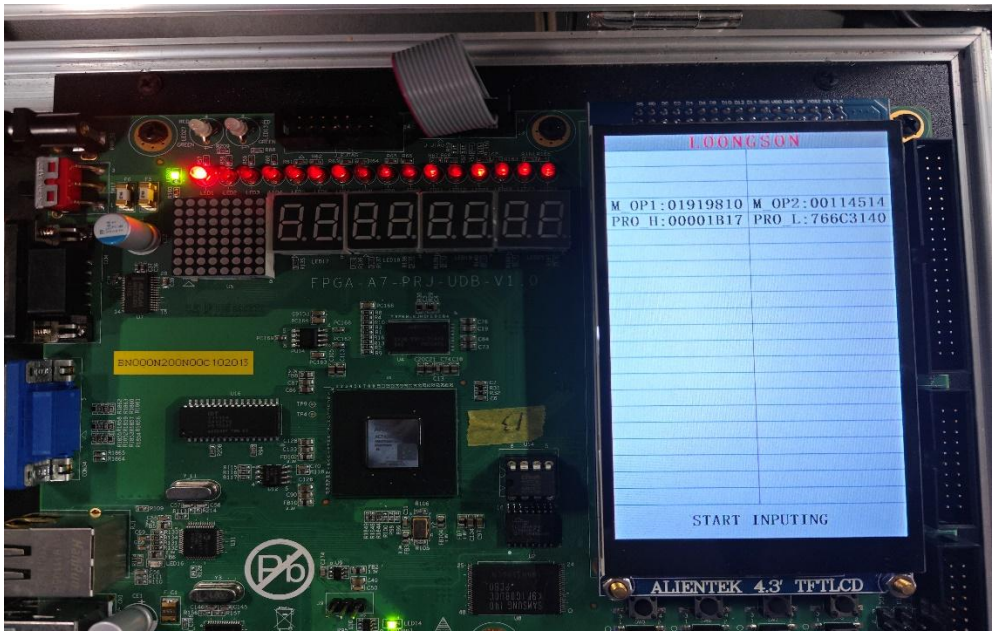
```
                6'd9 :
                begin
                    display_valid <= 1'b1;
                    display_name   <= "PRO_H";
                    display_value <= product_r[63:32];
                end
                6'd10 :
                begin
                    display_valid <= 1'b1;
                    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
```

注：主要修改了显示屏的显示位置，从 1，2，3，4 修改成了 7，8，9，10

5. 实验结果分析

（一） 上箱验证

1. 正数乘正数

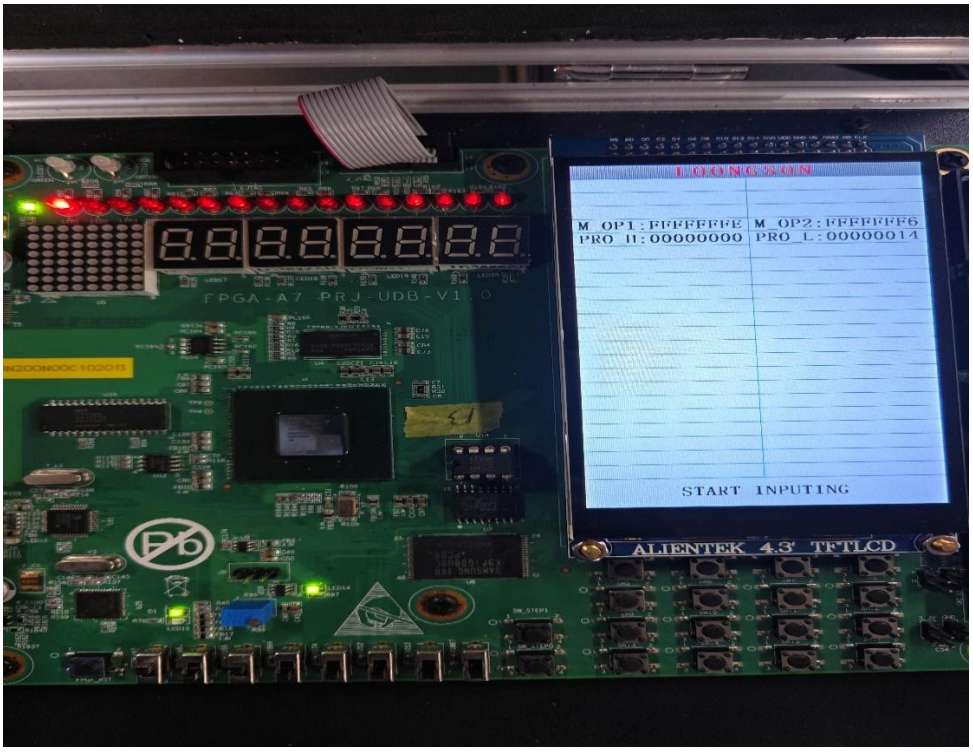


解释： op1 = 0x1919810, op2 = 0x114514，op1*op2 = 0x1B17766C3140，结果正确。

2. 正数乘负数

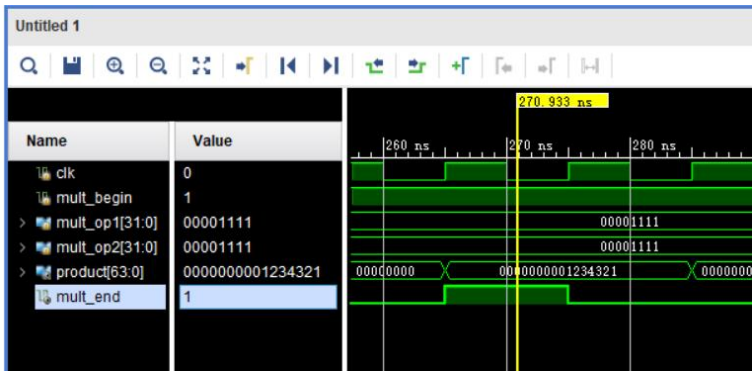
解释：op1 = 0x4 = 4, op2 = 0xFFFFFFF6 = -10，op1*op2 = 4*(-10) = -40 = 0xFFFFFFFFFFFFFFD8，结果正确。

3. 负数乘负数

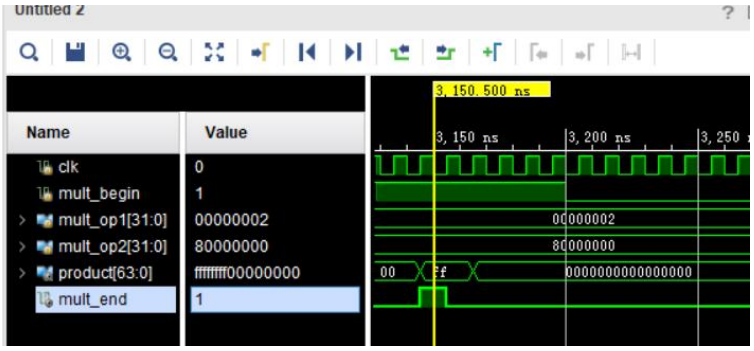


解释： op1 = 0xFFFFFFFE = -2 , op2 = 0xFFFFFFF6 = -10， op1*op2 = (-2)*(-10) = 20 = 0x14，结果正确。

（二） 仿真验证



解释： op1 = 0x1111, op2 = 0x1111, op1*op2 = 0x1234321(mult_end 为 1 表示迭代结束)



解释： op1=0x2，op2=0x80000000=-2147483648,op1*op2=-4294967296=0xFFFFFFFF00000000

6. 总结感想

通过本次实验，我对 vivado 的使用更加熟悉了，并且更加熟练对 Verilog 语句的编写，对数据的移位操作有了更加深入的了解，知道了在什么场景下应用这些移位操作，同时对乘法器的原理以及优化方法有了一定的了解。