

## 组成原理课程第二次实报告

### 实验名称：定点乘法及移动两位的改进

学号：22122266 姓名：张恒硕 班次：0416

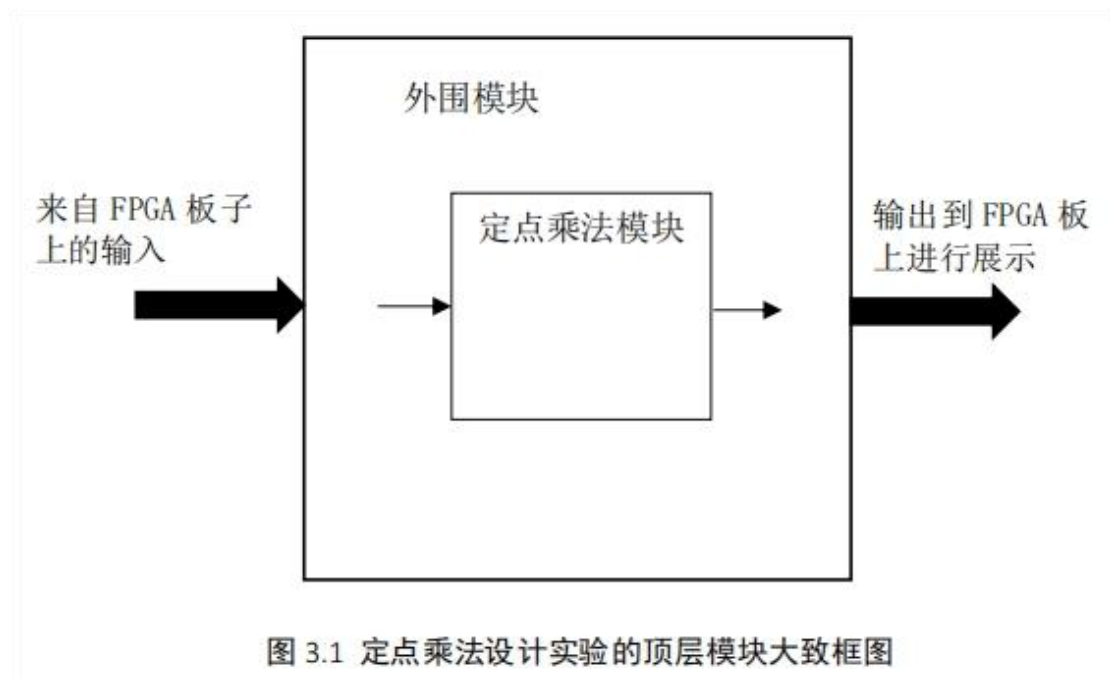
#### 一、实验目的

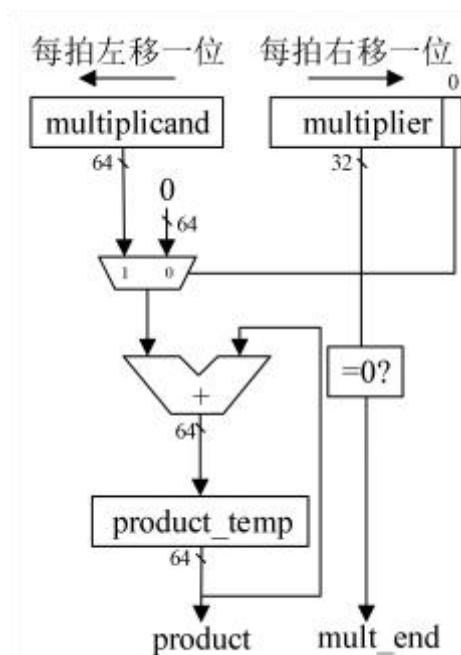
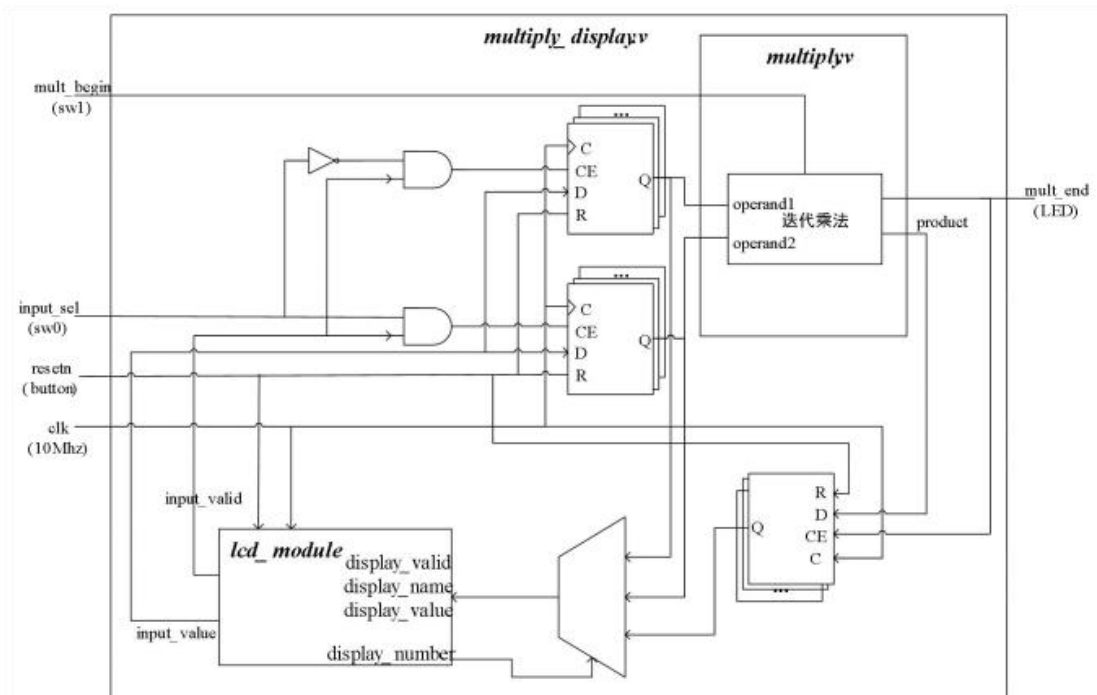
1. 理解不同的定点乘法的实现算法的原理，掌握基本实现算法。
2. 熟悉并运用 verilog 语言进行电路设计。
3. 为后续设计 cpu 的实验打下基础。

#### 二、实验内容说明

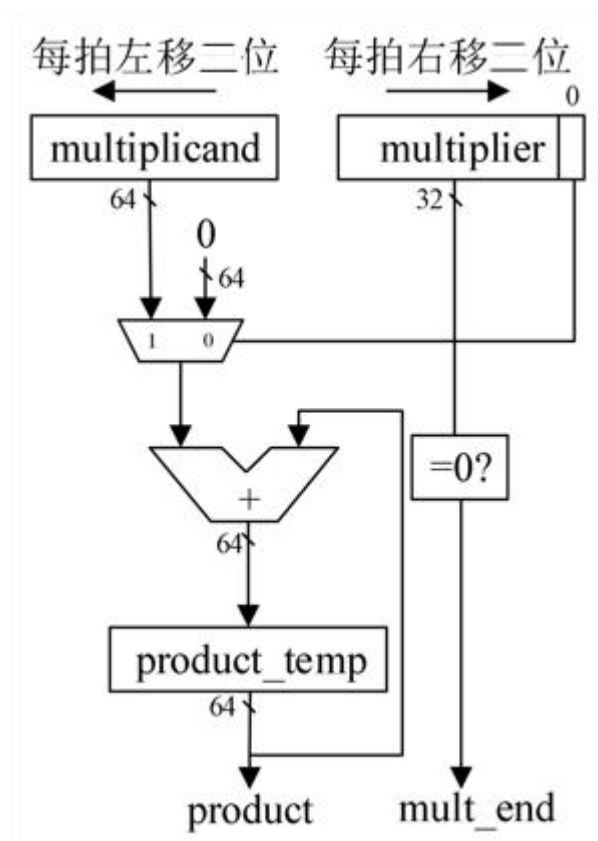
1. 学习并理解计算机多种定点乘法器的实现算法的原理，重点掌握迭代乘法的实现算法。
2. 设计实验方案，画出结构框图。
3. 在 verilog 中搭建乘法模块，编写相应代码。
4. 仿真编写的代码，得到正确的波形图。
5. 将以上设计作为一个单独的模块，设计一个外围模块去调用之。外围模块中需调用封装好的 LCD 触摸屏模块，显示两个乘数和乘法结果，且需要利用触摸功能输入两个乘数。
6. 将编写的代码进行综合布局布线，并下载到实验箱中的 FPGA 板上进行演示。
7. 重复上述步骤，通过修改代码进行改进，并仿真、演示。

#### 三、实验原理图





multiplcand 为被乘数, 记为 MD, multiplier 为乘数, 记为 ME, 二者下标从 0 开始, 由右向左增加。当  $ME_i=0$  时, 当前位结果为 0; 当  $ME_i=1$  时, 当前位结果为 MD 向左移  $i$  位, 用 0 补位。在遍历完 ME 后, 各位结果相加, 得到最终的结果。该结果的十六位数, 高八位和低八位分别显示。该算法需要等候所有位相乘的结果后, 才能进行加和, 消耗时间较长。



末两位	计算
00	product_temp + 0
01	product_temp + 被乘数
10	product_temp + 左移一位后的被乘数
11	product_temp + 左移一位的被乘数+被乘数

在改进的算法中，MB 每次左移两位并用 0 补位，ME 选取两位，进行相乘，将各步的结果相加，得到最终的结果，并通过高位、低位分别显示。在各步中，MB 与  $ME_{2i+1} ME_{2i}$  相乘，其结果根据后者有四种情况，上表已经给出。本算法将各位相乘与相加流水线化，大大节省了计算时间。选取移动两位，是因为两位只有四种情况，在各位乘法的计算中耗时较短。

#### 四、实验步骤

以下给出了定点乘法的代码。

```

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，运算时每次左移一位
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一位
        multiplicand <= {multiplicand[62:0],1'b0};
    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 <= {1'b0,multiplier[31:1]};
    end
end

```

```

        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_product;
    assign partial_product = multiplier[0] ? multiplicand : 64'd0;

    //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;
endmodule

```

代码先声明了时钟信号 clk、乘法开始信号 mult\_begin、乘法源操作数 mult\_op1 和 mult\_op2、乘积 product、乘法结束信号 mult\_end。在运算前，取绝对值，对尾数和符号位分开计算。被乘数在运算时每次左移一位，乘数则每次右移一位，最后累加部分积，得到结果，并加上符号位。

在上述代码的基础上，可以进行改进，实现移两位的定点乘法。

```

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;

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

```

```

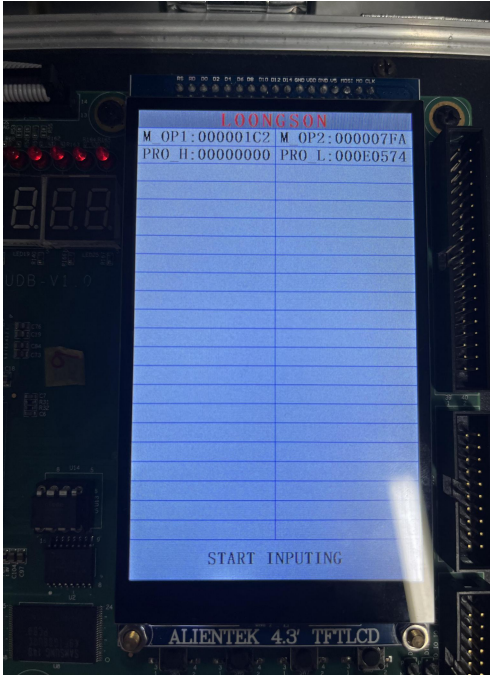

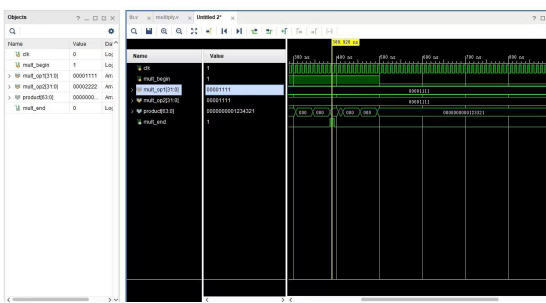
reg [31:0] multiplier;
always @ (posedge clk)
begin
    if (mult_valid)
    begin
        multiplier <= {2'b00,multiplier[31:2]};
    end
    else if (mult_begin)
    begin
        multiplier <= op2_absolute;
    end
end
endmodule

```

代码先声明了同样的变量，依旧在运算前取绝对值，只不过在运算时，每次移动两位。

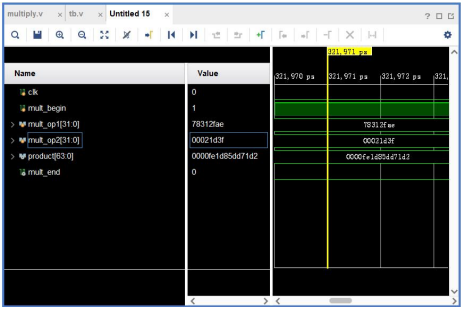
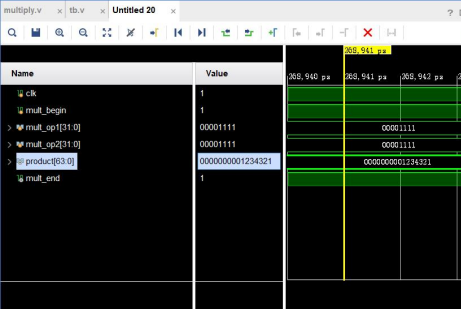
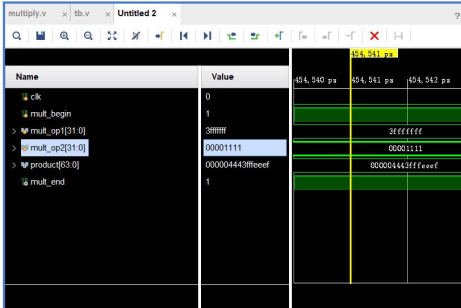
### 五、实验结果分析

被乘数	乘数	理论结果	实际结果 (高位/ 低位)	实验截图
00000221	00001123	247B83	00000000 / 00247B83	

000001C2	000007FA	E0574	00000000 / 000E0574	
000657EF	0004169F	19EF3B27 71	00000019 / EF3B2771	
00001111	00001111	1234321	00000000 / 01234321	

注：本实验中，选取了四组不同大小的输入，获得了不一样的结果，对乘法运算的准确性和高位、低位输出关系进行了验证。



被乘数	乘数	理论结果	实际结果 (高位/ 低位)	实验截图
78312FAE	00021D3F	FE1D 85DD71D2	0000FE1D / 85DD71D2	
00001111	00001111	1234321	00000000 / 01234321	
3FFFFFFF	00001111	444 3FFFEDEF	00000444 / 3FFFEDEF	
注：本实验中，选取了三组不同大小的输入，再次验证了乘法运算的准确性和高位、低位输出关系。				

然而，在对代码进行测试时，我们发现，当被乘数最高位大于 8 时，两种乘法器都会报错，如两个 FFFFFFFF 相乘时，会出现错误的结果。

## 六、总结感想

本次实验相较于上次实验，在算法逻辑和思想难度上有较大提升，促进了对 verilog 语言的学习，并加强了使用 LS-CPU-EXB-002 实验箱和软件平台的熟练性。本次实验实现了两种不同的定点乘法算法，二者在逻辑上存在递进关系，这帮助我加强了对乘法器的理解，为后续的学习和实验打下了基础。