组成原理实验课程第二次实验报告

实验名称	乘法器的效率提升实验			班级	张金
学生姓名	冯思程	学号	2112213	指导老师	董前琨
实验地点	实验楼 A306		实验时间	4 月 3 日 14：00	

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

1. 学习并理解计算机中定点乘法器的多种实现算法的原理，重点掌握迭代乘法的实现算法。

2. 根据第二次实验原理自行设计本次修改实验的方案，画出结构框图，详细标出输入输出端口。

3. 针对组成原理第二次的乘法法器实验进行改进的具体要求：

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）

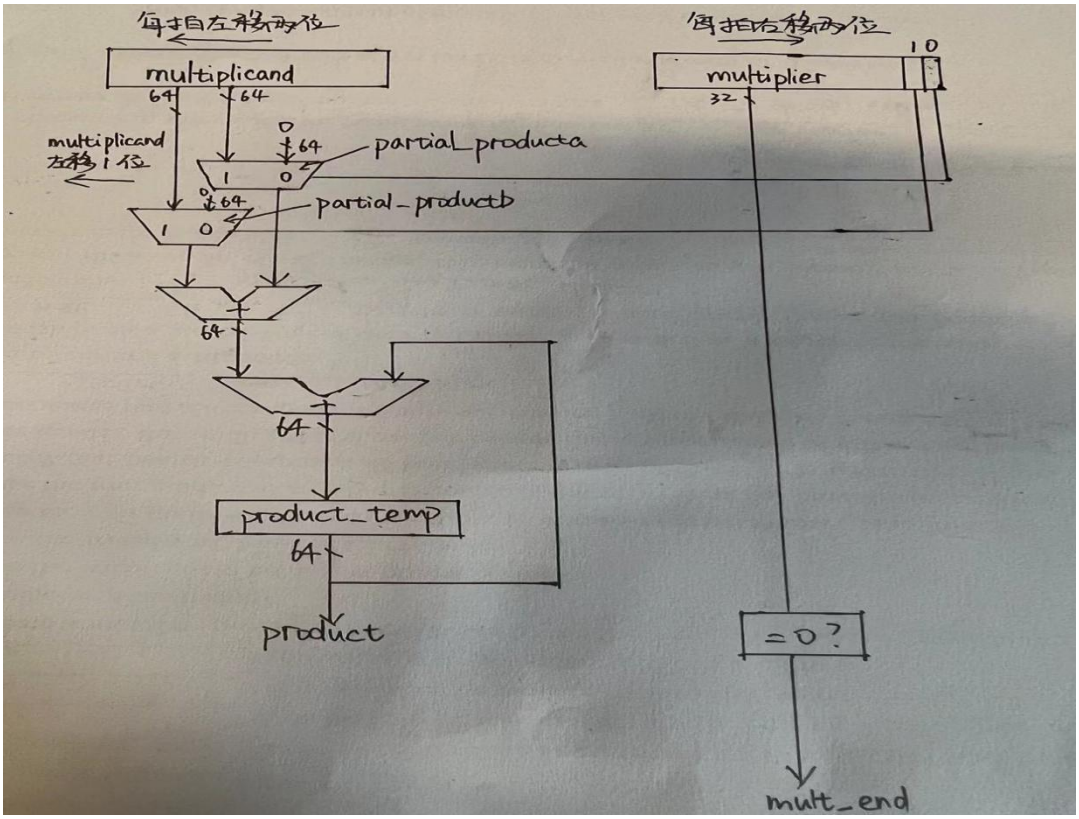
2）将改进后的乘法器进行仿真验证，得到正确的波形图。

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，不再是顶层模块图实验原理图。

3、 实验原理图

对迭代乘法算法原理图进行修改，改为每拍移位两位，原理图如下：



其中 partial_producta,partial_productb 分别代表乘数末位和乘数倒数第二位与被乘数相乘得到的结果，其中当乘数末位为 1 的时候，partial_producta 是乘数本身；当乘数末位为 0，临时变量 a 为 0。当乘数倒数第二位为 1，partial_productb 由被乘数左移一位得到；乘数倒数第二位为 0，partial_productb 为 0。partial_producta,partial_productb 加和作为循环变量迭代。最后当乘数是 0 时候 mult_end 翻转，乘法结束。

4、 实验步骤

（1）对 multiply.v 的修改：（红色表示进行修改的地方）

第一部分，对加载被乘数的修改，原来是运算时每次左移 1 位，现在改为两位。代码如下：

```
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两位
        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
```

第二部分，对加载乘数的修改，原来是运算时每次右移 1 位，改为两位。代码如下：

```
reg   [31:0] multiplier;
always @ (posedge clk)
begin
    if (mult_valid)
    begin     // 如果正在进行乘法，则乘数每时钟右两位
        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
```

第三部分，对部分积的修改，当乘数末位为 1，临时变量 a 是乘数本身；当乘数末位为 0，临时变量 a 为 0。乘数倒数第二位为 1，临时变量 b 由被乘数左移一位得到；乘数倒数第二位为 0，临时变量 b 为 0。通过上述分别处理乘数的末位和倒数第二位，我可以实现对每次移位两次的处理，代码如下：

```
wire [63:0] partial_producta;
wire [63:0] partial_productb;
assign partial_producta = multiplier[0] ? multiplicand : 64'd0;
assign partial_productb = multiplier[1]?{multiplicand[62:0],1'b0} : 64'd0;
```

第四部分，对后面累加器的表达式进行微小修改，将原来的一个 partial_换成了上面部

分积中的两个临时变量。代码如下：

```
reg [63:0] product_temp;
    always @ (posedge clk)
    begin
        if (mult_valid)
        begin
            product_temp <= product_temp + partial_producta + partial_productb;
        end
        else if (mult_begin)
        begin
            product_temp <= 64'd0;   // 乘法开始，乘积清零
        end
    end
```

（2）对 display 部分的修改：要求不能从显示屏的前四格显示，于是下面定义显示区域的语句进行修改，将 1-4 显示区域改成 5-8 显示区域。（红色标注修改部分）

```
    always @(posedge clk)
        begin
            case(display_number)
                6'd5 :
                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
        endcase
End
```

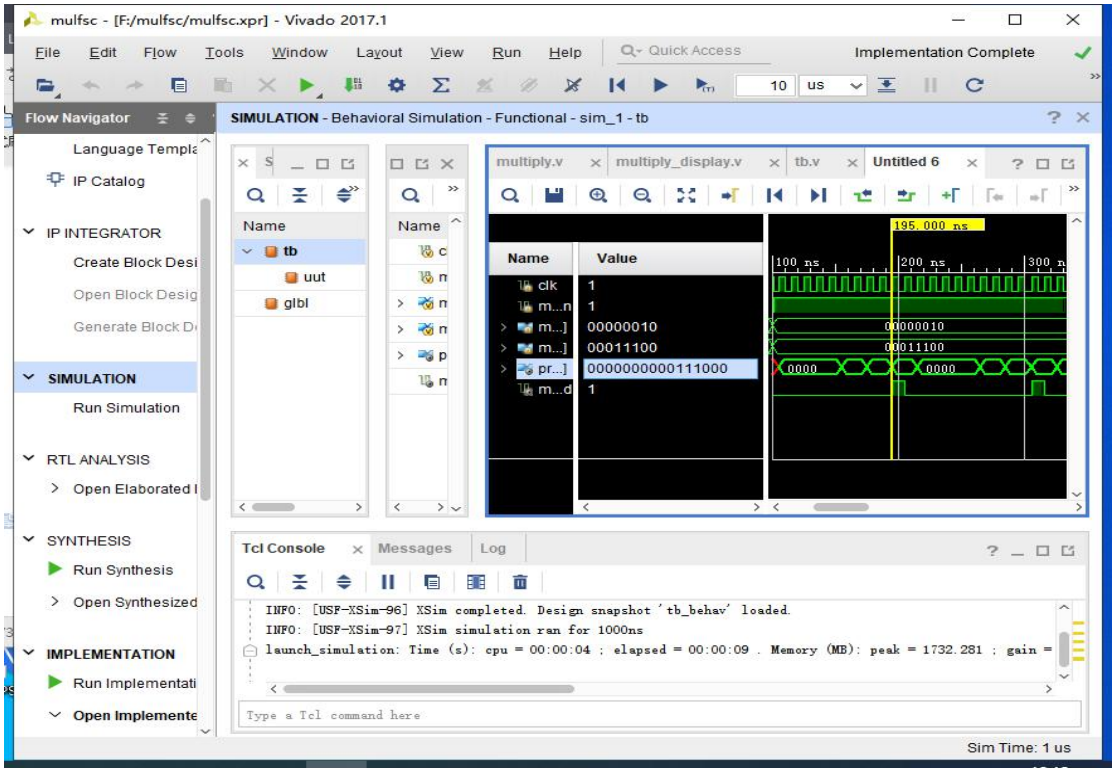
（3）对仿真文件的示例数据进行一定修改（修改成一组很容易验证正误的数据），然后跑仿真并验证仿真结果(分别 run synthesis 和 run implementation,然后在 run behavioral simulation)，代码如下：（红色标注修改部分）

```
#100;
mult_begin = 1;
mult_op1 = 32'H00000010;
mult_op2 = 32'H00011100;
```

（4）将源码给出的约束文件导入和同时导入 lcd 屏模块，然后再进行上箱实验验证。其中约束文件可以直接使用，无需再进行修改。lcd 屏模块也在源码文件（source code）中给出，直接导入。然后分别依次跑综合、增强后确认无误，将电脑连接到实验箱后生成流文件到实验箱上，然后在实验箱上进行调试观察，验证是否完整的实现了目标功能。

5、 实验结果分析

（1）仿真结果：（先调整波形图的横坐标尺度，找到 mult_end 翻转的地方，然后去观察乘法器输出的结果）

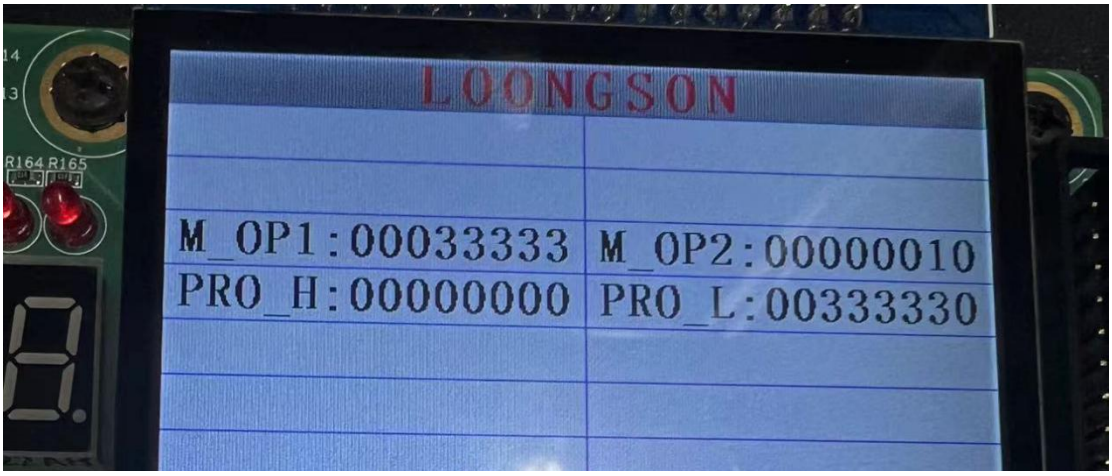


分析：结果正确，发现对于这组数据一共经过 9 个时钟周期得出结果，对照我们的改进乘法器原理，每次右移位 2 位，在乘数为 0 时停止乘法，乘数是 00011100，发现两两

非零对恰好为 9 组，与结果对应。证明了原理的正确性和代码的准确性。同时证明了这种经过修改后的乘法器的效率提升了，运行周期减少一半。

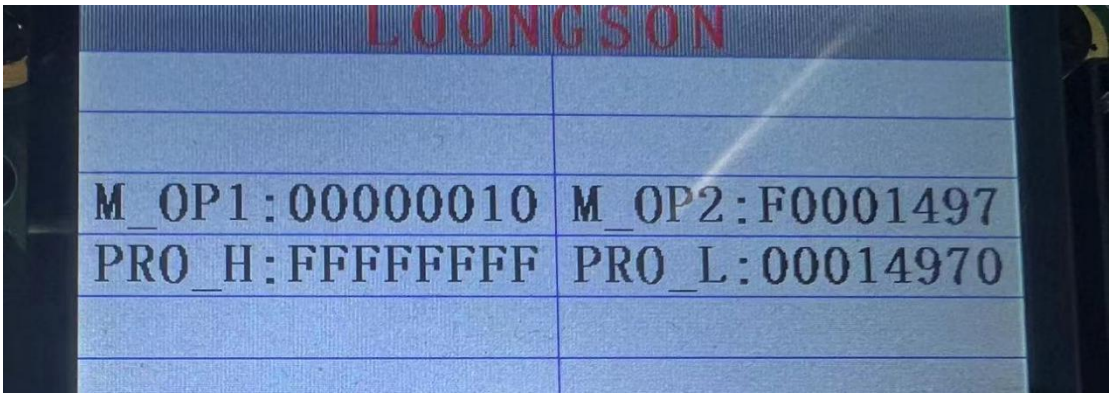
（2）上箱结果验证：（要对按钮进行说明，下面一排按钮最左边按钮控制选择输入乘数还是被乘数，在这排按钮的左边有一个按键是复位键）

正数与正数相乘：



结果正确，16 进制数乘一个 0x10 就是非零数位左移一位，成功验证了乘法器的正确功能。

正数与负数相乘：



其中 0xF0001497 是负数，0x00000010 是正数。正数的补码还是本身，负数的补码是取反后+1，即为 0x0FFFEB69，将补码相乘，结果是 0x00000000FFFEB690,最后判断正负，结果是负数，对补码结果取反+1，得到 0xFFFFFFFF00014970,经验证上述结果依然正确。

综上可以说明代码成功的实现了本次实验的要求。完成了实验任务。

6、 总结感想

在本次实验中，我成功地实现了一个 32 位移位乘法器，并将其烧录到实验箱上进行了测试。通过这个实验，我更加熟练地使用 Verilog 语言描述数字电路的行为和结构，掌握了移位乘法的原理和实现方法。

在实验过程中，我遇到了一些问题，如 Verilog 语法错误和测试不通过等。但通过仔细分析和调试，最终解决了这些问题，并且成功地完成了实验。

此外，在本次实验中，我也更加熟练地使用 Vivado 软件进行 FPGA 开发，让我更好地理解数字电路和硬件描述语言。

当我在实现 32 位移位乘法器的过程中，我发现使用每次移动两位来计算乘法比每次移动一位要快得多。这是因为移位操作是一种基本的位运算，比乘法操作要快得多。我深刻地体会到了效率的重要性。

因为效率对于未来整体 cpu 的设计至关重要。所以我想在了解一些乘法器效率优化上的技巧或者方法，以下是我学到的一些简单易懂的方法：

除了使用移位来计算乘法以外，还有一些其他的方法可以进一步优化乘法器的效率，如下：

1.Wallace 树：Wallace 树是一种将多个部分积相加的加法器结构。通过 Wallace 树，可以将乘法器的延迟和面积都大大减少。它的思想我认为类似分治，将整个事件进行拆分，实现效率的大幅度提升。

2.Karatsuba 算法：Karatsuba 算法是一种通过分治算法来实现乘法的方法。通过 Karatsuba 算法，可以将乘法的复杂度从 O(n^2)降低到 O(n^log2(3))，从而提高乘法器的效率。

综上所述，乘法器的效率可以通过多种方法进行优化。在实际设计中，我们需要根据具体的需求和资源限制，选择合适的方法来优化乘法器的效率。