

计算机组成原理第2次实验报告

实验名称：乘法器的改进——实现乘加器

学号：2313546 姓名：蒋枘言 班次：1078

一、实验目的

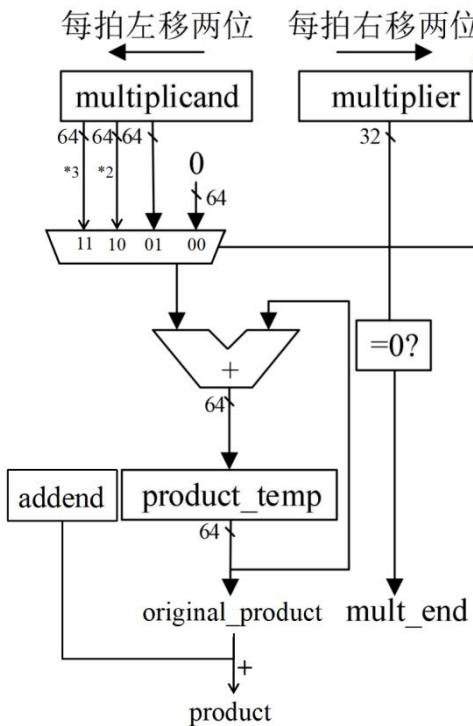
- 1.理解定点乘法的不同实现算法的原理，掌握基本实现算法。
- 2.熟悉并运用 Verilog 语言进行电路设计。
- 3.为后续设计 CPU 的实验打下基础。

二、实验内容说明

请参考实验指导手册上的实验二，把乘法器实验的代码运行验证成功后，完成对乘法器的扩展设计，注意要求如下：

- 1.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。
- 2.将乘法器修改成乘加器，实现 $A \times B + C$ 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）。
- 3.仿真文件原始代码中延迟 400、500 的时间单位过长，大家可以修改成 40、50，这样看波形比较直观，注意波形图上应该是 `multi_end` 为 1 时的输出才是正确输出。
- 4.上实验箱进行验证时，注意要 lcd 屏上需要输入三个数据，`input_sel` 不能再用 1 位了，此外 lcd 屏上还需要显示 A、B、C 三个数。

三、实验原理图



四、实验步骤

创建 multiply.v，输入以下修改后的代码。代码相关的解释见注释。

```
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
    input [31:0] addend,   // 新增的 32 位输入
    output [63:0] product, // 乘积
    output mult_end       // 乘法结束信号
);

reg mult_valid;
assign mult_end = mult_valid & ~(|multiplier); // 乘数全 0 时结束

always @(posedge clk) begin
    if (!mult_begin || mult_end) begin
        mult_valid <= 1'b0;
    end else begin
        mult_valid <= 1'b1;
    end
end

wire op1_sign = mult_op1[31];
wire op2_sign = mult_op2[31];
wire [31:0] op1_absolute = op1_sign ? (~mult_op1 + 1) : mult_op1;
wire [31:0] 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'b0}; // 左移两位
    end else if (mult_begin) begin
        multiplicand <= {32'd0, op1_absolute};
    end
end

/*
 * 被乘数（Multiplicand）左移两位
 * 每周期将被乘数左移两位，相当于乘以 4。
 * 例如，初始被乘数为 A，经过 1 个周期后变为 A<<2（即 4*A），第 2 个周期变为 A<<4（即 16*A）。
 * 在乘法开始时，将被乘数初始化为 32 位绝对值，并放在低 32 位，高 32 位补零（{32'd0, op1_absolute}）。
 */

```

```

reg [31:0] multiplier;
always @(posedge clk) begin
    if (mult_valid) begin
        multiplier <= {2'b0, multiplier[31:2]}; // 右移两位
    end else if (mult_begin) begin
        multiplier <= op2_absolute;
    end
end

```

/* 乘数（Multiplier）右移两位

每周期将乘数右移两位，丢弃已处理的最低位。

例如，初始乘数为 B，第 1 次处理后变为 $B \gg 2$ ，第 2 次变为 $B \gg 4$ 。

当乘数被右移成全 0 时 ($mult_end = mult_valid \& \sim(multiplier)$)，表示所有位已处理完毕。

*/

```

wire [1:0] multiplier_bits = multiplier[1:0];
wire [63:0] partial_product;
assign partial_product =
    (multiplier_bits == 2'b00) ? 64'd0 :
    (multiplier_bits == 2'b01) ? multiplicand :
    (multiplier_bits == 2'b10) ? {multiplicand[62:0], 1'b0} :
    (multiplicand + {multiplicand[62:0], 1'b0});

```

/* 两位乘数组合

最低两位的可能值为 00、01、10、11，分别对应 0、1、2、3 倍被乘数。

0 倍：直接输出 0。

1 倍：直接使用当前被乘数。

2 倍：被乘数左移 1 位（即 $multiplicand \ll 1$ ）。

3 倍：通过组合逻辑计算 1 倍 + 2 倍（即 $multiplicand + (multiplicand \ll 1)$ ）。

*/

```

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;
    end
end

```

```

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

```

```

    wire [63:0] addend_ext = {{32{addend[31]}}, addend};

/* 符号扩展
将 32 位输入 addend 符号扩展到 64 位。
如果 addend 是负数（最高位为 1），扩展后的高 32 位补 1。
如果 addend 是正数（最高位为 0），扩展后的高 32 位补 0。
*/

    wire [63:0] original_product =
        product_sign ? (~product_temp + 1) : product_temp;
/* 计算原始乘法结果（含符号处理）
原乘法结果 original_product 根据符号位处理：
若乘积为负数，对 product_temp 取反加 1。
若乘积为正数，直接使用 product_temp。
*/

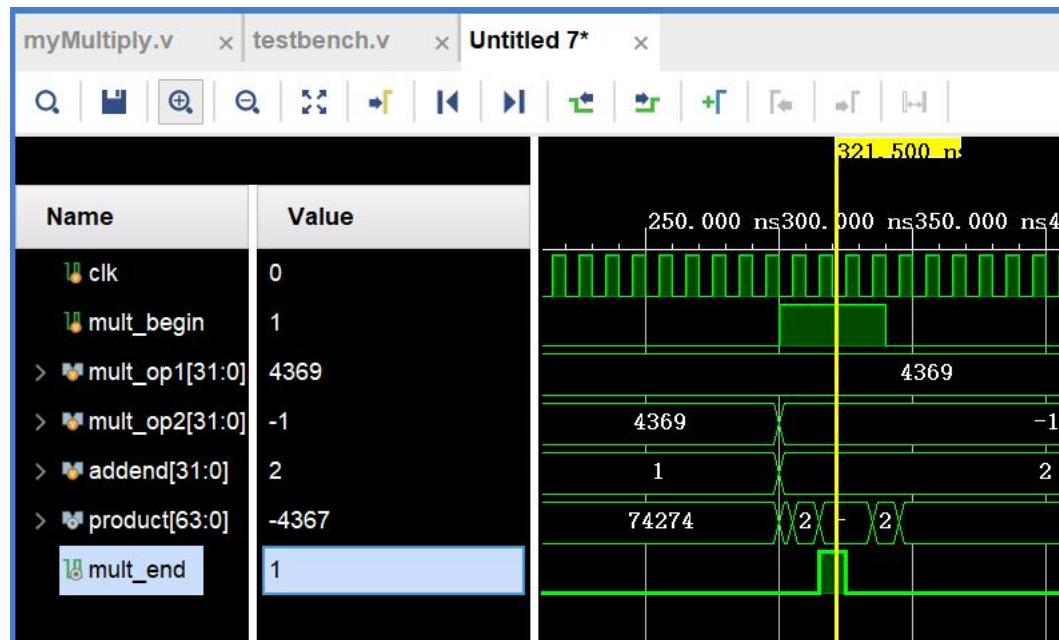
assign product = original_product + addend_ext; // 新输出
assign product = product_sign ? (~product_temp + 1) : product_temp;
// 将符号扩展后的 addend_ext 与原始乘法结果相加，得到最终输出。
endmodule

```

五、实验结果分析

使用两种方式进行验证。

(1) 通过仿真实验进行验证



取 mult_end 为 1 的时刻，此时被乘数与乘数分别为 4369（十进制）和 -1，加数为 2， $4369 \times (-1) + 2 = -4367$ ，与结果相符。

(2) 通过实验箱进行验证

由于此时有 3 个数要输入，`input_sel` 不能再用 1 位了，要改成两位。

首先将 `multiply_display.v` 里的 `input_sel` 扩展为两位：

```
input [1:0] input_sel,
```

实例化触摸屏也做出相应修改：

```
// 当 input_sel 为 0 (二进制 00) 时，输入乘数 1
always @(posedge clk) begin
    if (!resetn) begin
        mult_op1 <= 32'd0;
    end else if (input_valid && (input_sel == 2'b00)) begin
        mult_op1 <= input_value;
    end
end

// 当 input_sel 为 1 (二进制 01) 时，输入乘数 2
always @(posedge clk) begin
    if (!resetn) begin
        mult_op2 <= 32'd0;
    end else if (input_valid && (input_sel == 2'b01)) begin
        mult_op2 <= input_value;
    end
end

// 新增：当 input_sel 为 2 (二进制 10) 时，输入加数
always @(posedge clk) begin
    if (!resetn) begin
        addend <= 32'd0;
    end else if (input_valid && (input_sel == 2'b10)) begin
        addend <= input_value;
    end
end
```

然后修改约束文件 `multiply.xdc` 里的相关代码：

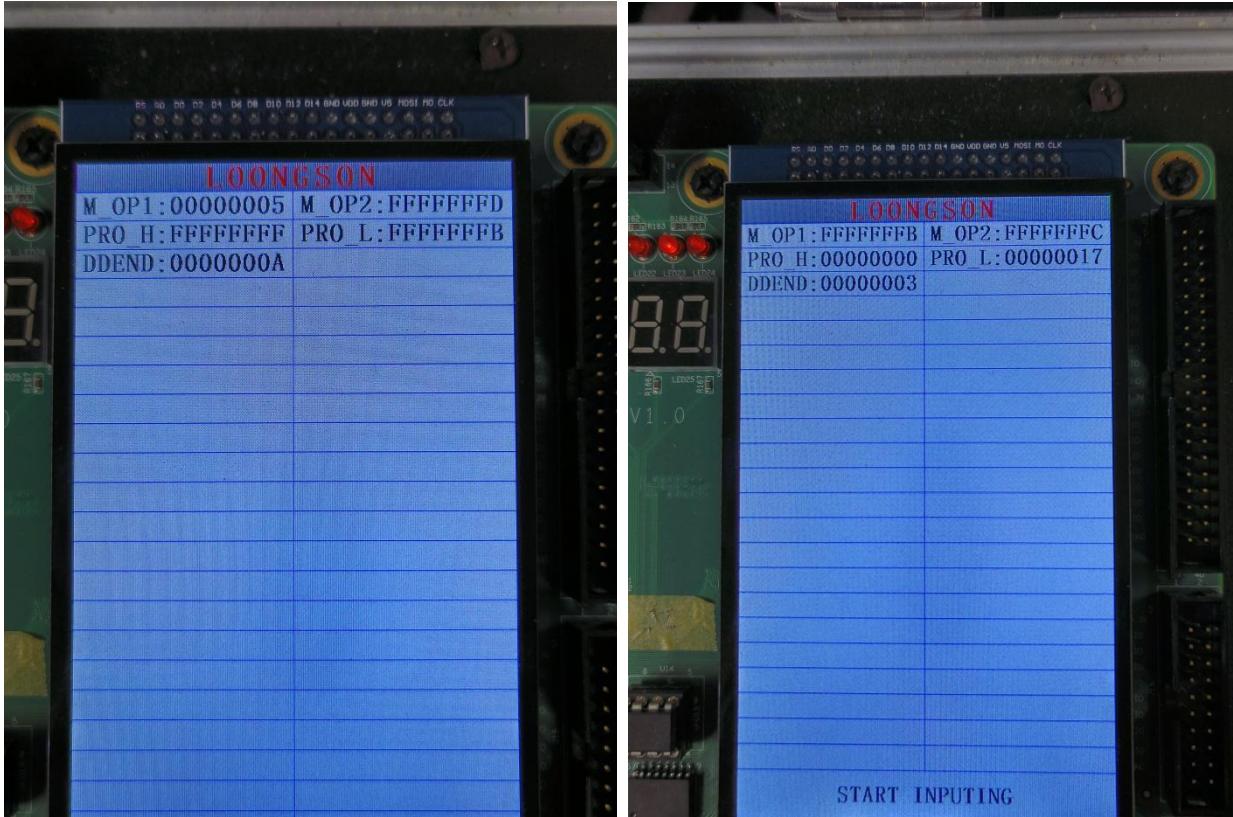
将

```
set_property PACKAGE_PIN AC21 [get_ports input_sel]
```

修改为

```
# 绑定 input_sel[1] 到 SW0 (AC21)，对应于左起第一个拨码开关
set_property PACKAGE_PIN AC21 [get_ports {input_sel[1]}]
# 绑定 input_sel[0] 到 SW2 (AC22)，对应于左起第三个拨码开关
set_property PACKAGE_PIN AC22 [get_ports {input_sel[0]}]
```

这样就可以通过左起第一个和第三个拨码开关控制输入了。



输入:

mult_op1 = 5 (0x00000005)
 mult_op2 = -3 (补码 0xFFFFFFFF)
 addend = 10 (0x0000000A)

预期输出:

product = $5 * (-3) + 10 = -5$
 PRO_H: 0xFFFFFFFF
 PRO_L: 0xFFFFFFFF

输入:

mult_op1 = -5 (补码 0xFFFFFFFb)
 mult_op2 = -4 (补码 0xFFFFFFFc)
 addend = 3 (0x00000003)

预期输出:

product = $(-5) * (-4) + 3 = 17$
 PRO_H: 0x00000000
 PRO_L: 0x00000017

结果完全正确！

六、总结感想

通过此次实验，在实现乘加器（ $A \times B + C$ ）的过程中，我进一步熟悉了 Verilog 语言的应用和 Vivado 的使用（虽然还不算特别熟悉）。

实验中需要输入三个操作数，这要求将原有的 1 位 `input_sel` 扩展为 2 位。通过修改约束文件，将两个拨码开关分别绑定到 `input_sel[1]` 和 `input_sel[0]`，最终实现了输入模式的灵活切换。这一点困扰了我很久，自行查找了资料最终才知道如何正确修改约束文件，才解决了这个问题。自行解决问题之后还是比较有成就感。