兰州大学信息科学与工程学院
计算机组成原理课程设计实验报告

一、 实验目的

1. 理解定点乘法的不同实现算法原理，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、 实验器材与设备

2.1 装有 Xilinx vivado 的计算机一台

2.2 LS-CPU-EXB-002 教学系统实验箱一套

三、 实验分析与设计

3.1 实验原理

通俗来讲，迭代乘法的原理就是模拟人手工列竖式计算两数相乘：

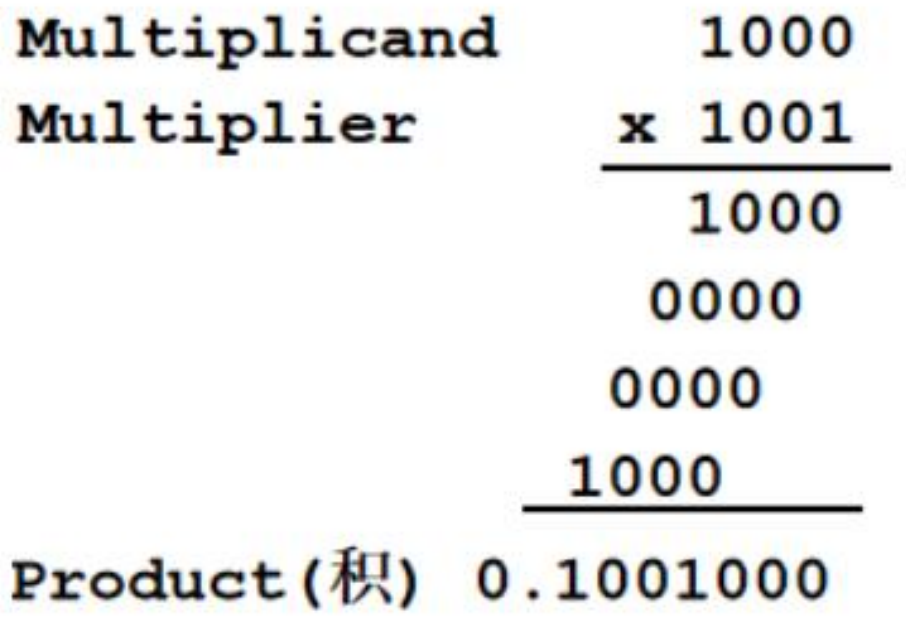


图 1 乘法实现原理的具体例子

设计的乘法部件主要使用的是迭代乘法算法，其原理图如下：

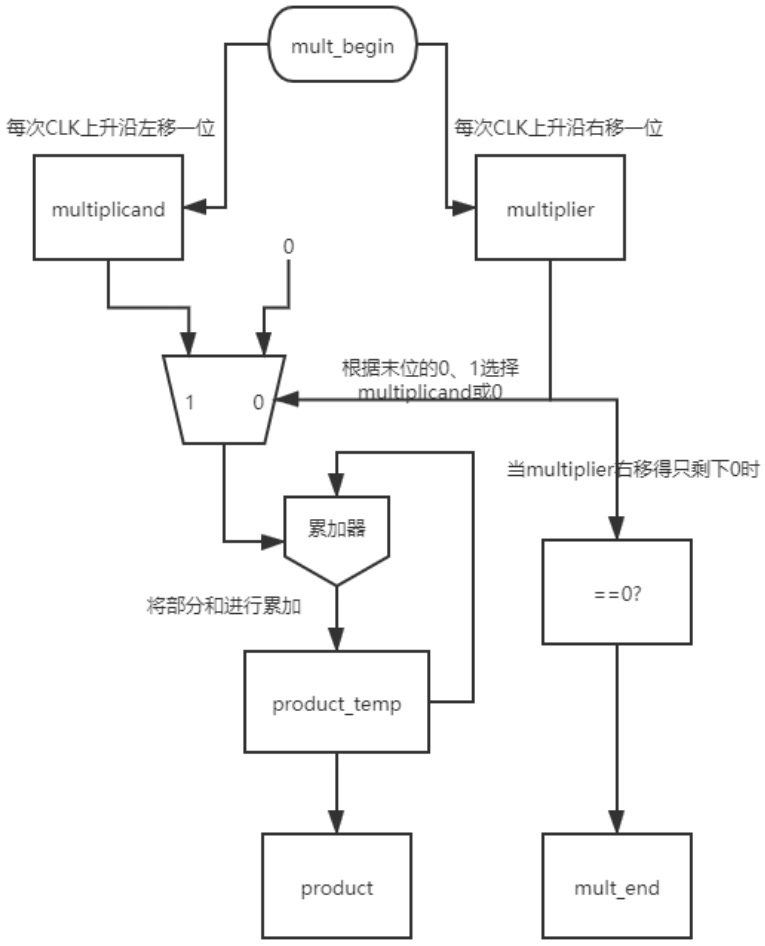


图 1 迭代乘法器算法原理图

具体的算法步骤如下：

3.2 端口设计

M_OP1：第一个 32 位乘数
M_OP2：第二个 32 位乘数
PRO_H：结果的高 32 位
PRO_L：结果的低 32 位

同样只用到了前两个开关 SW18 和 SW19

3. 3 设计框图

1. 顶层的设计框图

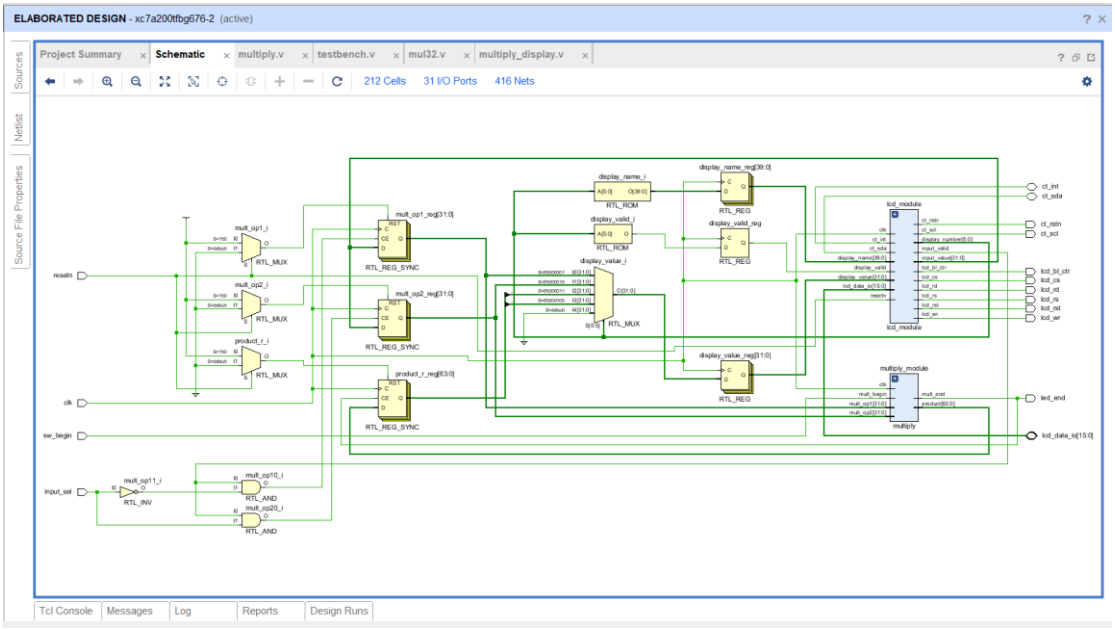


图 3 顶层的设计框图

2. 乘法器的设计框图

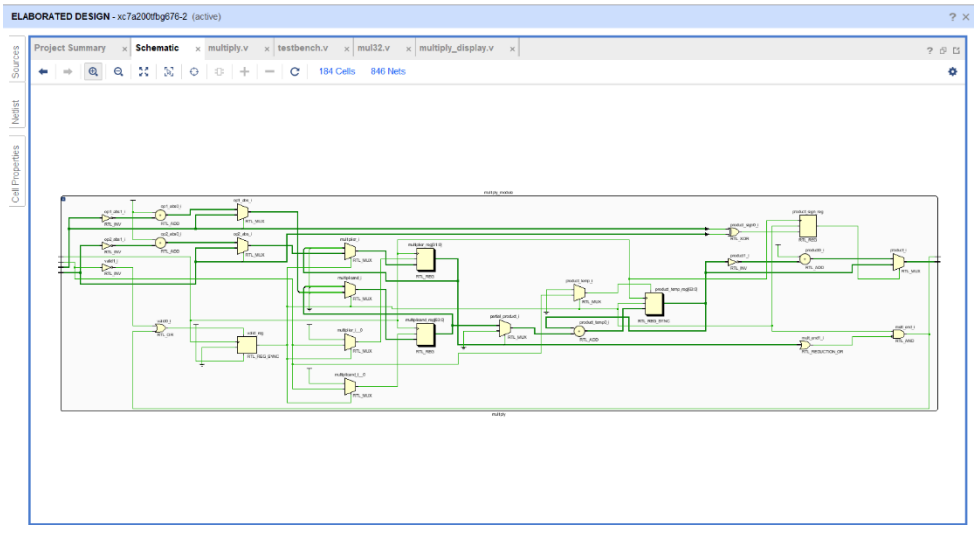


图 4 乘法器的设计框图

四、 实验步骤

4. 1 代码实现

1. 首先是外围模块 multiply_display.v 的实现
   我们直接使用模板进行改造即可
   a. 拨码开关(对应了上面的 SW18 SW19)

```
    //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,  //0表示乘法未开始 1表示开始乘法运算
```

b. 调用编写好的乘法器

```
//-----{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
```

c. 显示屏的输出

```
always @(posedge clk)
begin
    case(display_number)
        6'd1 :
        begin
            display_valid <= 1'b1;
            display_name  <= "M_OP1";
            display_value <= mult_op1;
        end
        6'd2 :
        begin
            display_valid <= 1'b1;
            display_name  <= "M_OP2";
            display_value <= mult_op2;
        end
        6'd3 :
        begin
            display_valid <= 1'b1;
            display_name  <= "PRO_H";
            display_value <= product_r[63:32];
        end
        6'd4 :
        begin
            display_valid <= 1'b1;
            display_name  <= "PRO_L";
            display_value <= product_r[31: 0];
        end
        default :
        begin
```

2. 然后是 multiply.v 的实现

```
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
);
    //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_abs;  //操作数1的绝对值
    wire [31:0] op2_abs;  //操作数2的绝对值
    assign op1_sign = mult_op1[31];//取符号位
    assign op2_sign = mult_op2[31];//取符号位
    assign op1_abs = op1_sign ? (~mult_op1+1) : mult_op1;//根据符号位的01得到其绝对值
    assign op2_abs = op2_sign ? (~mult_op2+1) : mult_op2;
    //assign {op1_abs, op2_abs} = op1_abs>op2_abs ? {op1_abs, op2_abs} : {op2_abs,op1_abs};//将较大的数作为被乘数，可以减少加法的次数
```

注：完整代码放在附录中

4. 2 仿真与综合

1. 首先是 Run Simulation，我们可以看到波形图：

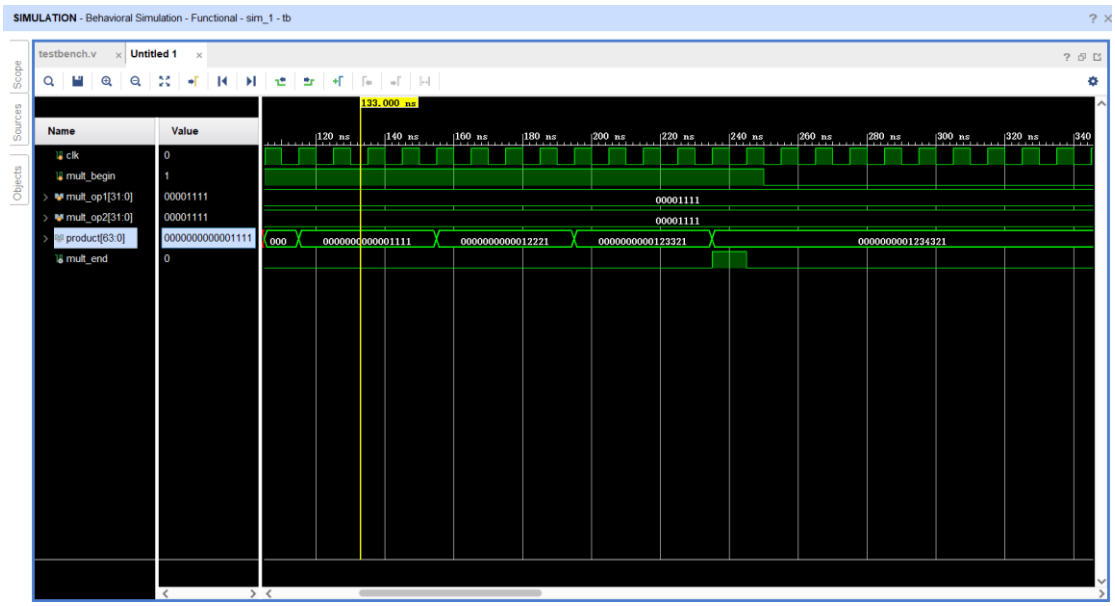
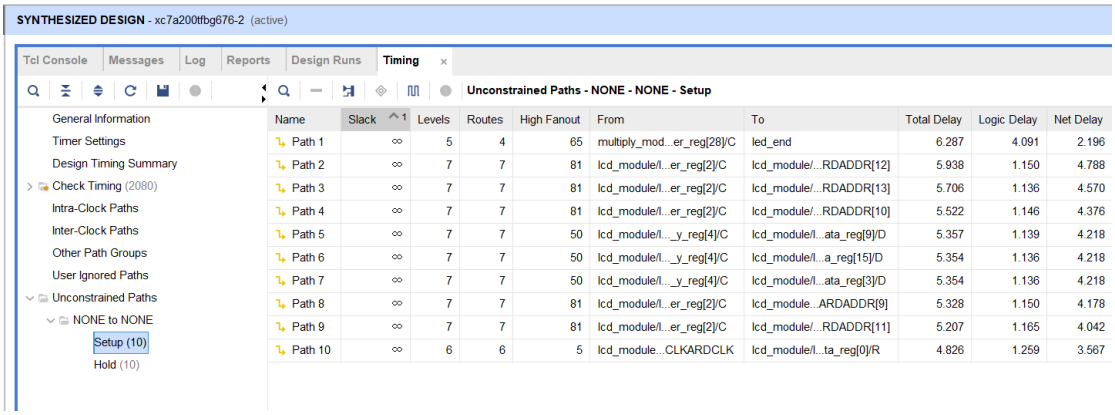


图 5 波形图

我们从波形图中的结果 product 的变化也可以看出迭代法的计算过程——就是部分积的累加，从 1111 到 12221 到 12321 到 1234321 就是部分积累加和位移所致的每个时刻的部分和，最后得到的就是正确的结果。

2. Synthesis——Report Timing Summary



Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 1	∞	5	4	65	multiply_mod...er_reg[28]/C	led_end	6.287	4.091	2.196
Path 2	∞	7	7	81	lcd_module/l...er_reg[2]/C	lcd_module/...RDADDR[12]	5.938	1.150	4.788
Path 3	∞	7	7	81	lcd_module/l...er_reg[2]/C	lcd_module/...RDADDR[13]	5.706	1.136	4.570
Path 4	∞	7	7	81	lcd_module/l...er_reg[2]/C	lcd_module/...RDADDR[10]	5.522	1.146	4.376
Path 5	∞	7	7	50	lcd_module/l..._y_reg[4]/C	lcd_module/l...ata_reg[9]/D	5.357	1.139	4.218
Path 6	∞	7	7	50	lcd_module/l..._y_reg[4]/C	lcd_module/l...a_reg[15]/D	5.354	1.136	4.218
Path 7	∞	7	7	50	lcd_module/l..._y_reg[4]/C	lcd_module/l...ata_reg[3]/D	5.354	1.136	4.218
Path 8	∞	7	7	81	lcd_module/l...er_reg[2]/C	lcd_module...ARDADDR[9]	5.328	1.150	4.178
Path 9	∞	7	7	81	lcd_module/l...er_reg[2]/C	lcd_module/...RDADDR[11]	5.207	1.165	4.042
Path 10	∞	6	6	5	lcd_module...CLKARDCLK	lcd_module/l...ta_reg[0]/R	4.826	1.259	3.567

图 6 synthesis 时延

3. Implement Design

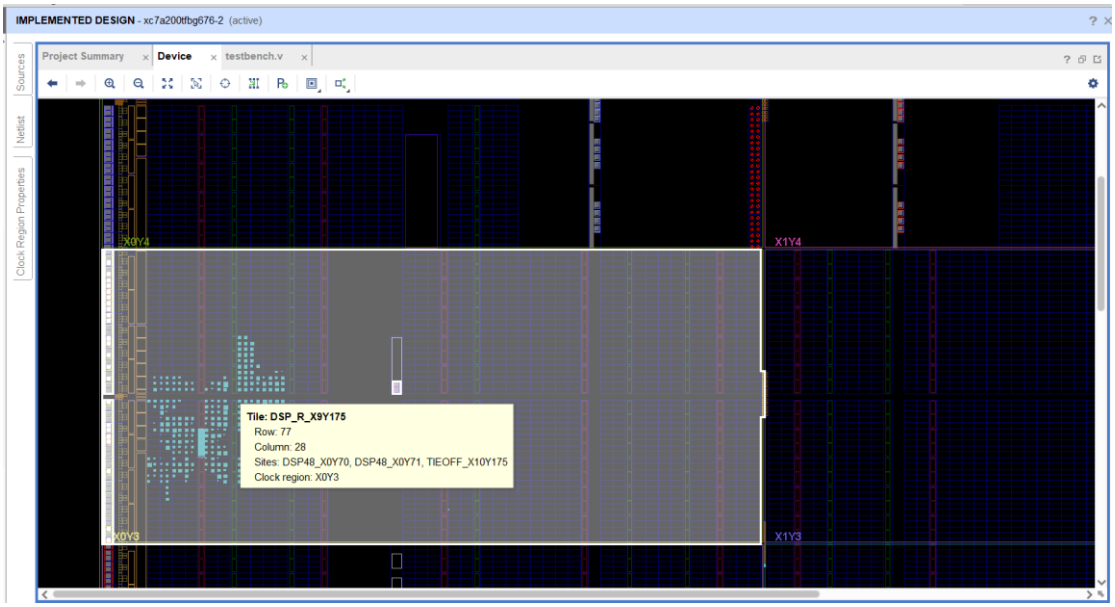


图 7 implement device

4.3 上板验证与结果展示

最后就是把写好的程序烧入到 LS-CPU-EXB 中，使用显示屏中的输入进行乘法器的验证

1. 低位乘法的验证：

1111*1111 = 1234321 答案正确

2. 高位乘法的验证：

1111*22222222=246888888642 结果正确

3. 负数*正数

4. 负数*负数

手工演算结果验算正确

五、 问题回答

5.1 基础问题

1. **为什么迭代乘法需要多次节拍才能实现一次乘法过程，对于 32 位乘法，最多需要多少拍才能完成一次乘法？**

2. **请详细说明迭代乘法算法原理以及流程，以第一步、第二步这样的写法把步骤写清楚**

答：迭代乘法算法的原理：部分积的移位累加。从被乘数的最低位开始，为 1 则乘数左移后与上一次的和相加；为 0 则左移全零相加，直到被乘数的最高位。

步骤：

a) 每一次 CLK 的上升沿触发被乘数左移一位，乘数右移一位
b) 根据乘数的末位，是 1 则本次部分和为刚刚左移的被乘数，0 则本次部分和为 0
c) 将部分和累加到最终结果上，完成本次运算
d) 将 abc 步骤进行数次直到乘数右移变为 0 为止

3. **Multiply_display.v 的结构是怎样的？怎么调用显示屏进行输入和输出？如何理解整个乘法器项目中各个文件的作用？**

5.2 思考题 基于 booth 算法的 32 位有符号乘法器，结合代码和仿真结果逐步讲解其原理和实现

原理讲解：**首先基于事实：任何二进制中连续的 1 可以被分解为两个二进制数之差**，所以我们可以使用更简单的运算来替换原数中连续 1 的乘法，通过加上乘数，对部分积进行位移运算，最后再将其从乘数中减去——本质上就是减少部分和的累加次数从而达到运算速度的提高。

六、 收获与体会

1. **学习了 32 位迭代乘法器的设计**
2. **再次巩固了从代码编写到上板验证的一套流程**
3. **理解了整个文件的模块功能与划分**
4. **为以后更复杂的设计打下良好的基础**
5. **初步掌握了 booth 算法的原理和 Verilog 语言的实现**

附录：迭代乘法器的实现代码：

```
`timescale 1ns / 1ps

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
```

```
);
    //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_abs;   //操作数 1 的绝对值
    wire [31:0] op2_abs;   //操作数 2 的绝对值
    assign op1_sign = mult_op1[31];//取符号位
    assign op2_sign = mult_op2[31];//取符号位
    assign op1_abs = op1_sign ? (~mult_op1+1) : mult_op1;//根据符号位的 01 得到其绝对值
    assign op2_abs = op2_sign ? (~mult_op2+1) : mult_op2;
    //assign {op1_abs, op2_abs} = op1_abs>op2_abs ? {op1_abs, op2_abs} : {op2_abs,op1_abs};//将较大的数作为被乘数，可以减少加法的次数

     //乘法运行的有效标志
    reg valid;
    always @(posedge clk)
    begin
        if (!mult_begin || mult_end)      //如果没有开始或者已经结束了
        begin
            valid <= 1'b0;        //mult_valid  赋值成 0，说明现在没有进行有效的乘法运算
        end
        else
        begin
            valid <= 1'b1;   //进行乘法运算
        end
    end

    //加载被乘数，运算时每次左移一位
    reg   [63:0] multiplicand;
    always @ (posedge clk)
    begin
        if (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};   //被乘数 op1 每次左移一位。
        end
        else if (mult_begin)
        begin     //  乘法开始，初始化被乘数，为 op1 的绝对值
            multiplicand <= {32'd0,op1_abs};
        end
    end
```

```
//op2
reg   [31:0] multiplier;
always @ (posedge clk)
begin
if(valid)
begin           //如果正在进行乘法，乘数 op2 右移一位
      multiplier <= {1'b0,multiplier[31:1]};
end
else if(mult_begin)
begin     //乘法开始，初始化乘数，为 op2 的绝对值
     multiplier <= op2_abs;
     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
reg [63:0] product_temp;
always @ (posedge clk)
begin
     if (valid)
     begin
          //进行每行结果的累加
          product_temp <= product_temp + partial_product;
     end
     else if (mult_begin)
     begin
           product_temp <= 64'd0;
     end
 end

//乘法结果的符号位和乘法结果
reg product_sign;//乘积结果的符号
always @ (posedge clk)
begin
     if (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;
    assign mult_end = valid & ~(|multiplier); //乘法结束信号：乘数全 0
endmodule
```