目　录

第一章　实验目的 1
第二章　实验任务与要求 2
2.1　实验任务 2
2.2　实验要求 2
第三章　实验结果 3
第四章　思考与讨论 7
4.1　课后问题 7
附　　录 11

图 目 录

图 4.1　乘法器波形图 9

表　目　录

第一章 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。

第二章 实验任务与要求

2.1 实验任务

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. 自行设计本次实验的方案，画出结构框图，详细标出输入输出端口，本次实验的乘法器建议采用迭代的方式实现，如果能力有余的，也可以采用其他效率更高的算法实现。本次实验要求实现的乘法为有符号乘法，因此需要注意计算机存储的有符号数都是补码的形式，设计方案传递进来的数也需是补码。
3. 根据设计的实验方案，使用 verilog 编写相应代码。
4. 对编写的代码进行仿真，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该模块。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 板子上进行演示。

2.2 实验要求

1. 做好预习：1）掌握定点乘法的多种实现算法的原理；2）确定定点乘法的输入输出端口设计；3）在课前画好设计框图或实验原理图；4）如果对 FPGA 板了解的话，可确定设计中与 FPGA 板上交互的接口，画出包含外围模块的整体设计框图，即补充完善图 3.1。
2. 实验实施：1）确认定点乘法的设计框图的正确性；2）编写 verilog 代码；3）对该模块进行仿真，得出正确的波形，截图作为实验报告结果一项的材料；4）完成调用定点乘法模块的外围模块的设计，并编写代码；5）对代码进行综合布局布线下载到实验箱里 FPGA 板上，进行上板验证。
3. 实验检查：1）完成上板验证后，让指导老师或助教进行检查，进行现场演示，可对演示结果进行拍照作为实验报告结果一项的材料。
4. 实验报告的撰写：1）实验结束后，需按照规定的格式完成实验报告的撰写。

第三章 实验结果

最终代码:

```
1  `timescale 1ns / 1ps
2  module multiply(                    // 乘法器
3      input           clk,            // 时钟
4      input           mult_begin,     // 乘法开始信号
5      input  [31:0]   mult_op1,       // 乘法源操作数1
6      input  [31:0]   mult_op2,       // 乘法源操作数2
7      output [63:0]   product,        // 乘积
8      output          overflow,       // 溢出
9      output          mult_end        // 乘法结束信号
10 );
11
12     //乘法正在运算信号和结束信号
13     reg mult_valid;
14     assign mult_end = mult_valid & ~(|multiplier); //乘法结束信
           号: 乘数全0
15     always @(posedge clk)
16     begin
17         if (!mult_begin || mult_end) //乘法未开始或者乘法结束
18         begin
19             mult_valid <= 1'b0; //乘法无效
20         end
21         else
22         begin
23             mult_valid <= 1'b1; //乘法有效
24         end
25     end
26
27     //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
28     wire        op1_sign;       //操作数1的符号位
```

```
29      wire          op2_sign;       //操作数2的符号位
30      wire [31:0]   op1_absolute;   //操作数1的绝对值
31      wire [31:0]   op2_absolute;   //操作数2的绝对值
32      assign op1_sign = mult_op1[31];
33      assign op2_sign = mult_op2[31];
34      assign op1_absolute = op1_sign ? (~mult_op1+1) : mult_op1;
35      assign op2_absolute = op2_sign ? (~mult_op2+1) : mult_op2;
36
37      //加载被乘数，运算时每次左移一位
38      reg  [63:0] multiplicand;
39      always @(posedge clk)
40      begin
41          if(mult_valid)
42          begin     // 如果正在进行乘法，则被乘数每时钟左移一位
43              multiplicand <= {multiplicand[62:0],1'b0};
44          end
45          else if(mult_begin)
46          begin    // 乘法开始，加载被乘数，为乘数1的绝对值
47              multiplicand <= {32'd0,op1_absolute};
48          end
49      end
50
51      //加载乘数，运算时每次右移一位
52      reg  [31:0] multiplier;
53      always @(posedge clk)
54      begin
55          if(mult_valid)
56          begin    // 如果正在进行乘法，则乘数每时钟右移一位
57              multiplier <= {1'b0,multiplier[31:1]};
58          end
59          else if(mult_begin)
60          begin    // 乘法开始，加载乘数，为乘数2的绝对值
61              multiplier <= op2_absolute;
62          end
63      end
```

```
64
65      // 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
66      wire [63:0] partial_product;
67      assign partial_product = multiplier[0] ? multiplicand : 64'd0;
68
69      //累加器
70      reg [63:0] product_temp;
71      always @(posedge clk)
72      begin
73          if (mult_valid)
74          begin
75              product_temp <= product_temp + partial_product;
76          end
77          else if (mult_begin)
78          begin
79              product_temp <= 64'd0;  // 乘法开始，乘积清零
80          end
81      end
82
83      //乘法结果的符号位和乘法结果
84      reg product_sign;
85      always @(posedge clk)  // 乘积
86      begin
87          if (mult_valid)
88          begin
89              product_sign <= op1_sign ^ op2_sign;
90          end
91      end
92
93
94      // 若乘法结果为负数，则需要对结果取反+1
95      assign product = product_sign ? (~product_temp+1) : product_temp;
```

```
96
97      // 检测溢出
98      // 当乘法结果大于2^31-1或小于-2^31时，溢出
99      assign overflow = product > 64'h7fffffff || product < -64'
            h80000000;
100
101     endmodule
```

第四章 思考与讨论

4.1 课后问题

1. 以 4 位二进制数 1010 和 0110 为源操作数 1 和源操作数 2，手工完成计算过程

$$
\begin{array}{rcccccccc}
 & & & & 1 & 0 & 1 & 0 \\
\times & & & & 0 & 1 & 1 & 0 \\
\hline
 & & & & 0 & 0 & 0 & 0 \\
 & & & 1 & 0 & 1 & 0 & \\
 & & 1 & 0 & 1 & 0 & & \\
+ & 0 & 0 & 0 & 0 & & & \\
\hline
 & 0 & 1 & 1 & 1 & 1 & 0 & 0
\end{array}
$$

2. 判断乘法器的有效输入为有符号数还是无符号数？尝试更改为另一种有效输入方案

乘法器的有效输入为有符号数，因为乘法器的输入是有符号数，输出也是有符号数

更改为无符号数:

```
module multiply(
    input           clk,
    input           mult_begin,
    input  [31:0]   mult_op1,    // 改为无符号数
    input  [31:0]   mult_op2,    // 改为无符号数
    output [63:0]   product,
    output          mult_end
);

    // 乘法正在运算信号和结束信号
    reg mult_valid;
    assign mult_end = mult_valid & ~(|multiplier);
    always @(posedge clk) begin
        if (!mult_begin || mult_end)
            mult_valid <= 1'b0;
        else
            mult_valid <= 1'b1;
```

```
18      end
19
20      // 加载被乘数，运算时每次左移一位
21      reg  [63:0] multiplicand;
22      always @(posedge clk) begin
23          if (mult_valid)
24              multiplicand <= {multiplicand[62:0], 1'b0};
25          else if (mult_begin)
26              multiplicand <= {32'd0, mult_op1}; // 无需绝对值
                处理
27      end
28
29      // 加载乘数，运算时每次右移一位
30      reg  [31:0] multiplier;
31      always @(posedge clk) begin
32          if (mult_valid)
33              multiplier <= {1'b0, multiplier[31:1]};
34          else if (mult_begin)
35              multiplier <= mult_op2; // 无需绝对值处理
36      end
37
38      // 部分积
39      wire [63:0] partial_product;
40      assign partial_product = multiplier[0] ? multiplicand :
          64'd0;
41
42      // 累加器
43      reg [63:0] product_temp;
44      always @(posedge clk) begin
45          if (mult_valid)
46              product_temp <= product_temp + partial_product;
47          else if (mult_begin)
48              product_temp <= 64'd0;
49      end
50
```

```
51      // 乘法结果
52      assign product = product_temp;
53
54  endmodule
```

3. 对仿真结果（波形图）进行注释

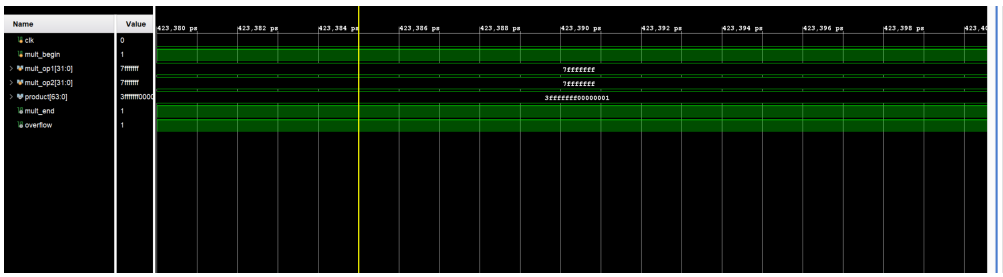

图 4.1 乘法器波形图

(a) mult_begin 信号在第一个时钟上升沿变为 1，表示乘法开始

(b) mult_end 信号在第四个时钟上升沿变为 1，表示乘法结束

(c) mult_op1 和 mult_op2 在第一个时钟上升沿变为 7fffffff, 7fffffff

(d) product 在第四个时钟上升沿变为 3fffffff00000001

(e) overflow 为 1，表示溢出

4. 说明 mult_begin 信号最终由什么（实验箱）确定

根据代码中的逻辑，当 mult_begin 为高电平时，表示乘法操作应该开始。而 mult_begin 信号的状态受到外部时钟信号 clk 的影响，并且还受到其他输入信号 mult_op1 和 mult_op2 的状态的影响。只有当 mult_begin 为高电平且其他条件满足时，乘法操作才会开始。

5. 给出乘法器溢出标志位（按有符号数计算处理），提交代码

```
1   module multiply(                // 乘法器
2   input           clk,            // 时钟
3   input           mult_begin,     // 乘法开始信号
4   input   [31:0]  mult_op1,       // 乘法源操作数1
5   input   [31:0]  mult_op2,       // 乘法源操作数2
6   output  [63:0]  product,        // 乘积
7   output          overflow,       // 溢出
8   output          mult_end        // 乘法结束信号
9   );
```

```
10
11      //......
12
13      // 当乘法结果大于2^31-1或小于-2^31时，溢出
14      assign overflow = product > 64'h7fffffff || product <
            -64'h80000000;
```

附　　录

参考链接：https://github.com/zehua0417/ComputerOrganizationAndArchitecture_exp