浙江大学

本科实验报告

课程名称：	计算机组成与设计
姓　　名：	张晋恺
学　　院：	竺可桢学院
系：	所在系
专　　业：	计算机科学与技术
学　　号：	3230102400
指导教师：	刘海风

2024 年 10 月 23 日

浙江大学实验报告

课程名称： 计算机组成与设计 实验类型： 综合

实验项目名称: 乘法器，除法器与浮点加法器

学生姓名： 张晋恺 专业： 计算机科学与技术 学号： 3230102400

同组学生姓名： 指导老师： 刘海风

实验地点： 东 4-512 实验日期： 2024 年 10 月 17 日

一、操作方法与实验步骤

乘法器

建立工程，新增源文件，Verilog 代码如下

```
module multiplier(
  input clk,
  input start,
  input[31:0] A,
  input[31:0] B,
  output reg finish,
  output reg[63:0] res
);

  reg state; // 记录 multiplier 是不是正在进行运算
  reg[31:0] multiplicand; // 保存当前运算中的被乘数

  reg[5:0] cnt; // 记录当前计算已经经历了几个周期（运算与移位）
```

```
14  wire[31:0] absulote_A = (A[31]) ? ~A + 1 : A;
15  wire[31:0] absulote_B = (B[31]) ? ~B + 1 : B;
16
17
18  reg sign = 0;
19
20  initial begin
21    res <= 0;
22    state <= 0;
23    finish <= 0;
24    cnt <= 0;
25    multiplicand <= 0;
26  end
27
28  always @(posedge clk) begin
29    if(~state && start) begin//step 0
30      sign <= A[31] ^ B[31];
31      multiplicand <= absulote_A;
32      res <= {32'b0, absulote_B};
33      state <= 1;
34      finish <= 0;
35      cnt <= 0;
36    end
37
38     else if(state) begin
39      cnt = cnt+1;
40      if(res[0]==1'b1)begin
41        res [63:32]=res[63:32]+multiplicand;
42      end
43
44      res = res >> 1;
45
46    end
47
48    // 填写 cnt 相关的内容，用 cnt 查看当前运算是否结束
49    if(cnt==32) begin
50      // 得到结果
51      cnt <= 0;
52      finish <= 1;
53      state <= 0;
54      res<=sign ? ~res + 1 : res;
55    end
56
57  end
58
```

```
endmodule
```

我们实现的是 32 位有符号乘法器，每一个周期进行一位的运算，一共要进行 32 个周期，对于有符号与无符号的处理；我们采用先判断符号，然后用绝对值进行无符号的运算，最后再根据符号来判断是不是要取补码；需要注意的是，这里有一个比较关键的变量`state`, 用于判断当前是否正在运行乘法运算；当`state`为 0 时，表示乘法运算已经结束，可以进行下一次的运算；**除了初始化之外，计算部分必须要使用阻塞赋值，因为会对下面的结果造成影响，不能使用非阻塞赋值**; 这一点在实验心得会更加详细的说明为什么；

对于算法部分，我使用了课上讲的 Version3, 乘数和结果共用一一个 64 位寄存器，将乘数存在低 32 位，每次判断最低位是否为 1，如果是则在高 32 位加上乘数，然后右移一位；做完第 32 次之后，将 finish 置为 1，表示运算结束，同时通过符号来判断是否要取补码；

仿真代码如下

```
module multiplier_tb;

  reg clk, start;
  reg[31:0] A;
  reg[31:0] B;

  wire finish;
  wire[63:0] res;

  multiplier m0(.clk(clk), .start(start), .A(A), .B(B), .finish(finish),
   ↪  .res(res));

  initial begin
    $dumpfile("multiplier_signed.vcd");
    $dumpvars(0, multiplier_tb);

    clk = 0;
    start = 0;
    #10;
    A = 32'd1;
    B = 32'd0;
    #10 start = 1;
    #10 start = 0;
    #300;

    A = 32'd10;
```

```
    B = 32'd30;
    #10 start = 1;
    #10 start = 0;
    #300;

    A = 32'd66;
    B = 32'd23;
    #10 start = 1;
    #10 start = 0;
    #300;

    A=-32'd10;
    B=32'd17;
    #10 start = 1;
    #10 start = 0;
    #300;

    A=-32'd10;
    B=-32'd17;
    #10 start = 1;
    #10 start = 0;
    #300;

    A=32'hffffffff;
    B=32'hffffffff;
    #10 start = 1;
    #10 start = 0;
    #300;

    A=32'h80000000;
    B=32'h80000000;
    #10 start = 1;
    #10 start = 0;
    #300;

    A=32'h7fffffff;
    B=32'h7fffffff;
    #10 start = 1;
    #10 start = 0;
    #500;
    $finish();


  end

```

```
  always begin
    #2 clk = ~clk;
  end



endmodule
```

具体的波形分析在第二部分说明；

除法器

建立工程，新增源文件，Verilog 代码如下

```
module divider(
    input   clk,
    input   rst,
    input   start,          // 开始运算
    input[31:0] dividend,   // 被除数
    input[31:0] divisor,    // 除数
    output reg divide_zero,     // 除零异常
    output  reg finish,         // 运算结束信号
    output[31:0] res,       // 商
    output[31:0] rem        // 余数
);

reg [5:0] cnt;
reg [64:0]reminder;
reg quotien;
reg state;

initial begin
    state<=0;
    cnt<=0;
    reminder<=0;
    quotien<=0;
    finish<=0;
    divide_zero<=0;
end

assign res=reminder[31:0];
assign rem=reminder[64:33];
```

```
29
30  always @(posedge clk or posedge rst) begin
31
32      if(rst) begin
33          finish<=0;
34          divide_zero<=0;
35          state<=0;
36      end
37
38      else if(~state&&start) begin//step 0
39          if(divisor==0) begin
40              reminder<=65'bx;
41              divide_zero<=1;
42              finish<=1;
43          end
44          else begin
45              reminder<={32'b0,dividend,1'b0};//sll 1 bit
46              state<=1;
47              cnt<=0;
48              finish<=0;
49              divide_zero<=0;
50          end
51      end
52
53      else if(state) begin
54          cnt=cnt+1;//阻塞赋值
55          if(reminder[63:32]>=divisor) begin
56              reminder[63:32]=reminder[63:32]-divisor;
57              quotien=1;
58          end
59          else begin
60              quotien=0;
61          end
62          reminder={reminder[63:0],quotien};
63      end
64      if(cnt==32) begin
65          cnt<=0;
66          finish<=1;
67          state<=0;
68      end
69
70  end
71
72  endmodule
```

我们实现的的是 32 位无符号，有除零判断的的除法器；我们沿用了乘法器的`state,start`变量，用于判断当前是否正在运行除法运算以及是否开始运算；根据的算法也是课上讲的 Version3，具体步骤如下

1. 读入数据，除数放在 64 位的低 32 位；
2. 读入数据的时候，就把 remaider 寄存器先左移一位；
3. 以下每一个周期进行一次运算，先判断高 32 位是否比被除数大，若是，则减去，并且左移上 1; 否则，左移上 0；一共要进行 32 个周期，在第 32 个周期做了之后, 将 finish 置为 1，state 置 0，表示运算结束，可以进行下一次的运算；
4. 输出结果时，商在低 32 位，余数在高 32 位的左移一位，实际上在设计的时候 remainder 寄存器要设置为 65 位的，因为不能直接丢到移出去的位，它在末尾还需要移回来；

仿真代码如下

```
`timescale 1ns / 1ps


module divider_tb();
      reg clk;
      reg rst;
      reg [31:0] dividend;
      reg [31:0] divisor;
      reg start;

      wire divide_zero;
      wire [31:0] res;
      wire [31:0] rem;
      wire finish;
      divider   u_div(
         .clk(clk),
         .rst(rst),
         .dividend(dividend),
         .divisor(divisor),
         .start(start),
         .divide_zero(divide_zero),
         .res(res),
         .rem(rem),
         .finish(finish)
      );
```

```
26      always #5 clk = ~clk;
27
28      initial begin
29       clk =0;
30       rst = 1;
31       start = 0;
32       #10
33       rst = 0;
34           dividend = 32'd0;
35           divisor  = 32'd4;
36           start = 1;#10;
37           start = 0;#355;
38
39
40           dividend = 32'd9;
41           divisor  = 32'd5;
42           start = 1;#10;
43           start = 0;#355;
44
45
46           dividend = 32'h80000000;
47           divisor  = 32'd1;
48           start = 1;#10;
49           start = 0;#355;
50
51           dividend = 32'hffffffff;
52           divisor  = 32'h80000000;
53             start = 1;#10;
54             start = 0;#355;
55
56           dividend = 32'd100;
57           divisor  = 32'd0;
58            start = 1;#10;
59            start = 0;#355;
60           #350 $stop();
61
62      end
63  endmodule
64
```

具体的波形分析在第二部分说明；

浮点加法器

新增源文件，Verilog 代码如下

```
`timescale 1ns / 1ps


module FP_adder(
    input clk,
    input rst,
    input start,
    input [31:0] A,
    input [31:0] B,
    output reg [31:0] res,
    output reg finish
    );

    reg [7:0] A_exp, B_exp, res_exp;
    reg [24:0] A_frac, B_frac, res_frac;

    reg working,res_sign;

    reg [3:0] state;

    localparam Check_Denomal=0,
               align=1,
               addition=2,
               normalization=3,
               done=4;

    initial begin
        working <= 0;
        res <= 0;
        finish <= 0;
        state <= Check_Denomal;
    end


    always @(posedge clk or posedge rst) begin
        if(rst) begin
            working <= 0;
            res <= 0;
            finish <= 0;
            state <= Check_Denomal;
```

```
        end
        else if (~working && start) begin
            A_exp <= A[30:23];
            B_exp <= B[30:23];
            A_frac <= {2'b01,A[22:0]};
            B_frac <= {2'b01,B[22:0]};
            state <= Check_Denomal;
            working <= 1;
            finish <= 0;
        end
        else if (working) begin
            case(state)
                Check_Denomal: begin
                    if(A_exp == 8'b00000000 || A_exp== 8'hff) begin
                        res_exp <= A_exp;
                        res_frac <= A_frac;
                        res_sign <= A[31];
                        state <= done;
                    end
                    else if(B_exp == 8'b00000000 || B_exp==8'hff) begin
                        res_exp <= B_exp;
                        res_frac <= B_frac;
                        res_sign <= B[31];
                        state  <= done;
                    end
                    else begin
                        state <= align;
                    end
                end

               align: begin
                if(A_exp == B_exp) begin
                    res_exp <= A_exp;
                    state <= addition;
                end

                else if(A_frac==0 || B_frac ==0)begin
                    res_exp <= A_frac==0 ? B_exp : A_exp;
                    res_frac <= A_frac==0 ? B_frac : A_frac;
                    res_sign <= A_frac==0 ? B[31] : A[31];
                    state <= done;
                end

                else if(A_exp > B_exp) begin
                    B_frac <= {1'b0,B_frac[24:1]};
```

```
                B_exp <= B_exp + 1;
            end

            else if(A_exp < B_exp) begin
                A_frac <= {1'b0,A_frac[24:1]};
                A_exp <= A_exp + 1;
            end

        end

        addition: begin
            if(A[31]^B[31]==0) begin
                res_sign <= A[31];
                res_frac <= A_frac + B_frac;
            end
            else begin
                 res_sign <= A_frac > B_frac ? A[31] : B[31];
                 res_frac <= A_frac > B_frac ? A_frac - B_frac :
                  ↪  B_frac - A_frac;
            end

            state <= normalization;

        end

        normalization: begin
            if (res_frac==0) begin
                res_exp <= 8'b00000000;
                state <= done;
            end

            else if(res_frac[24]==1) begin
                res_frac <= {1'b0,res_frac[24:1]};
                res_exp <= res_exp + 1;
                state <= done;//此时一定是正常化的
            end
            else if(res_frac[23]==0) begin
                res_frac <= {res_frac[23:0],1'b0};
                res_exp <= res_exp - 1;
                state <= normalization;//此时仍然可能还需要继续规格化
            end
            else begin
                state <= done;
            end
        end
```

```
130
131             done: begin
132                 res = {res_sign, res_exp, res_frac[22:0]};
133                 finish <= 1;
134                 working <= 0;
135             end
136
137             endcase
138         end
139     end
140
141 endmodule
```

我们实现的是 32 位浮点加法器，
将其划分为以下几个时序部分

1. **初始化**: 在时钟的上升沿或复位信号为高时，初始化工作状态、结果和完成信号。工作状态 (working) 设为 0，结果 (res) 设为 0，完成 (finish) 设为 0，状态 (state) 设为 Check_Denomal。

2. **开始加法**: 如果工作状态为低且开始信号 (start) 为高，提取输入 A 和 B 的指数和尾数，并转到 Check_Denomal 状态。

3. **检查非规范数**: 在 Check_Denomal 状态，检查 A 和 B 的指数是否为零或全一。如果是，结果的指数、尾数和符号直接取自相应的输入，并转到 done。否则，转到 align。因为全零和全一的指数都有特殊的含义，所以不需要进行对齐和加法，直接输出即可；

4. **对齐尾数**: 在 align 状态，若指数相同，转到 addition。如果任一尾数为零，结果取另一操作数的指数和尾数，并转到 done。否则，逐步右移较小指数的尾数，并增加相应的指数，直到对齐。

5. **执行加法**: 在 addition 状态，检查符号位。如果相同，结果符号为输入符号，结果尾数为尾数相加。如果不同，结果符号为较大的尾数的符号，结果尾数为两者的差。转到 normalization。

6. **规格化结果**: 在 normalization 状态，检查结果尾数是否为零。如果是，指数设为零，转到 done。如果尾数的最高位 (溢出保留位) 为 1，右移尾数并增加指数，此时已经有前置 1 了，我们可以直接转到 done。如果最高位为 0，左移尾数并减小指数，继续规格化，直到前置 1 出现；

7. **完成**: 在 done 状态，组合结果的符号、指数和尾数，设置完成信号为 1，返回工作状态为 0。

需要注意的是，我们移位的时候并没有保留，这是因为向 0 舍入；在 frac 部分，我们需要添加两位，一位用于补全前导 1，另一位用于溢出保留；

仿真代码如下

```
`timescale 1ns / 1ps
module floatadder32_tb();

reg clk;
reg start;
reg rst;
reg [31:0] A;
reg [31:0] B;
wire finish;
wire [31:0] res;

FP_adder uut(
    .clk(clk),
    .start(start),
    .rst(rst),
    .A(A),
    .B(B),
    .finish(finish),
    .res(res)
);

always #5 clk = ~clk;

initial begin
    clk = 0;
    start = 0;
    rst = 0;

    // 测试非规格化数
    A = 32'h0000_0000;
    B = 32'h4040_0000;
    #10 start = 1;
    #10 start = 0;
    #355;
    //res = 32'h0000_0000;    直接输出 0.0

    A = 32'h40A0_0000;  // 5.0
    B = 32'h4040_0000;  // 3.0
```

```
    #10 start = 1;
    #10 start = 0;
    #200;
    // res = 32'h4100_0000;   5.0 + 3.0 = 8.0

    A = 32'hC0A0_0000;  // -5.0
    B = 32'h4040_0000;  // 3.0
    #10 start = 1;
    #10 start = 0;
    #355;
    // res = 32'hC000_0000;   -5.0 + 3.0 = -2.0

    A = 32'hC088_0000;  // -4.25
    B = 32'h40D0_0000;  // 6.5
    #10 start = 1;
    #10 start = 0;
    #355;
    // res = 32'h4010_0000;   -4.25 + 6.5 = 2.25

    A = 32'hC104_0000;  // -8.25
    B = 32'hC040_0000;  // -3.0
    #10 start = 1;
    #10 start = 0;
    #355;
    // res = 32'hC134_0000;   -8.25 + (-3.0) = -11.25

    A = 32'hC0A0_0000;  // -5.0
    B = 32'h40A0_0000;   //5.0
    #10 start = 1;
    #10 start = 0;



    #355;

    A = 32'h0;  // 0
    B = 32'h40A0_0000;   //5.0
    #10 start = 1;
    #10 start = 0;
    #355;


    rst = 1;
    #200;
    $finish();
```

```
84 end
85 endmodule
```

具体的波形分析在第二部分说明.

二、实验结果与分析

乘法器仿真

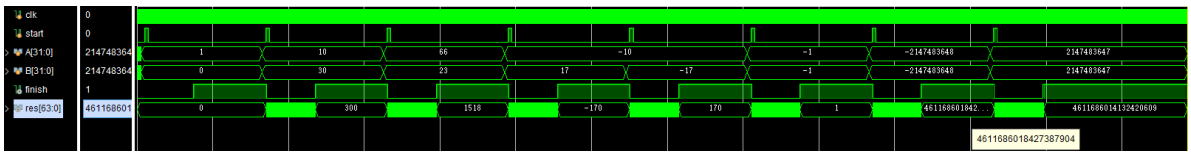
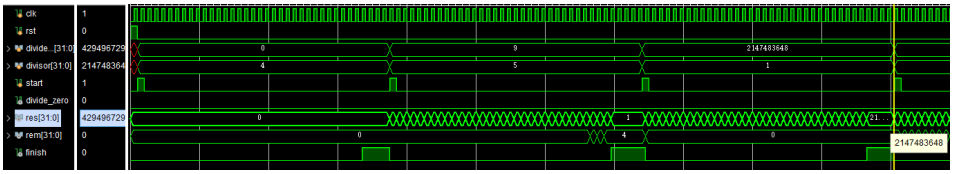


图 1: 乘法器仿真波形

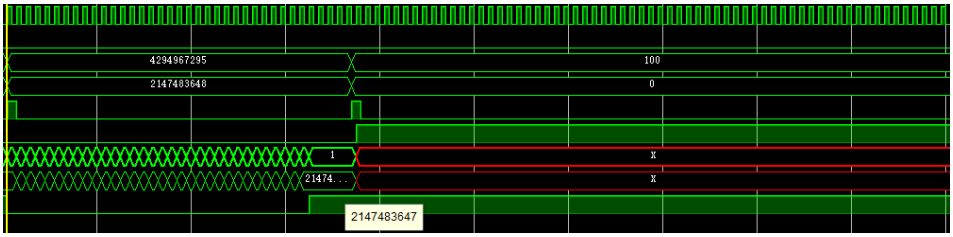
仿真结果分析

- 第一个测试 $1 \times 0 = 0$ 结果正确
- 第二第三个测试普通数据的相乘 $10 \times 30 = 300$ 与 $66 \times 23 = 1518$ 也正确，这说明我们的乘法器具备处理一般数据的能力
- 接下来测试正数与负数相乘的结果，比较简单，结果也是正确的
- 最后我们测试大数据相乘，32 位有符号数据的表示范围是 $-2,147,483,648$ 到 $2,147,483,647$，我们可以测试一些边界情况，比如 $2,147,483,647 \times 2,147,483,647$ 和 $-2,147,483,648 \times -2,147,483,648$。结果也正确，说明我们的乘法器具备处理二进制大数据的能力

除法器仿真



(a) a



(b) b

图 2: 除法器仿真波形

仿真结果分析

- 第一个测试 $0 \div 4 = 0$ 结果正确
- 第二个测试 $9 \div 5 = 1$ 余数为 4，结果正确，这是一般数据的除法
- 第三个测试 $2{,}147{,}483{,}648 \div 1 = 2{,}147{,}483{,}648$ 余数为 0，这是商比较大的情况，也是正确的
- 接下来在下面的波形图中，我们测试了 $4294967295 \div 2147483648 = 1$ 余数为 2147483647，这是余数比较大的情况，也是正确的
- 最后我们测试除零的情况，$100 \div 0$，这时候我们的除法器会输出未定义的值，同时 divide_zero 信号升高，说明除零发生；这也是正确的

浮点加法器仿真

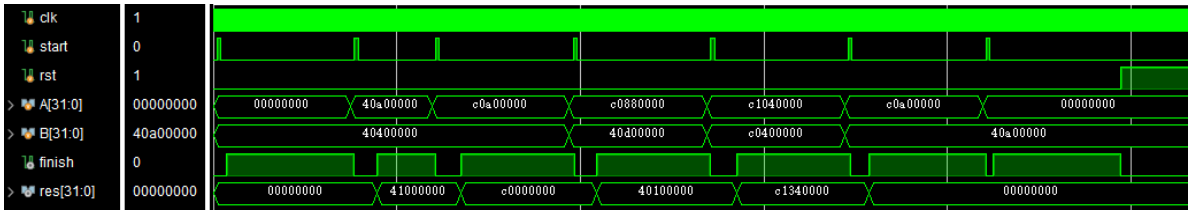


图 3: 浮点加法器仿真波形

仿真结果分析

这一部分的仿真样例设计比较麻烦；

- 第一个测试 A 是非规格化数，输出了非规格化数全 0；这是正确的
- 测试对应的浮点数为 $5.0 + 3.0 = 8.0$，而结果 32'h4100_0000 按照 IEEE754 标准表示为 8.0，结果正确
- 测试 $-5.0 + 3.0 = -2.0$ 结果为 32'hC000_0000 表示为浮点数-2.0，结果正确
- 测试 $-8.25 + (-3.0) = -11.25$ 结果为 32'hC134_0000 表示为浮点数-11.25，结果正确
- 测试 $-5.0 + 5.0 = 0.0$ 正负相消，结果为 0.0，结果正确
- 最后我们测试复位信号，$rst = 1$，这时候我们的浮点加法器对应的寄存器都会被重新初始化，这也是正确的

其中还有一些测试样例，在仿真代码中已经注释好其结果，这里就不再赘述。

三、讨论与实验心得

实验心得

本次实验给我最大的体会就是让我真真切切感受到了阻塞赋值和非阻塞赋值的区别：

当实现乘法器的时候，我的 cnt 一开始使用的是阻塞赋值，这导致我的 cnt 在第 32 次的时候，32 并没有立即赋值给 cnt ，而是在 if 语句将 cnt 当作 31 判断之后在赋值，所以会多做一个周期；

而在实现除法器的时候，我一开始对我的 quotient 变量使用的也是非阻塞赋值，这导致在这一周期中，remainder 寄存器的值将会上上一个周期的 quotient 的值，而不是这一周期的值，这导致了我的除法器的结果是错误的；

Verilog 阻塞和非阻塞赋值的区别

在 Verilog 中，阻塞（blocking）赋值和非阻塞（non-blocking）赋值是两种给寄存器型变量（`reg`）赋值的方式，它们的主要区别在于赋值的执行顺序和行为，尤其是在时序逻辑中。

1. 阻塞赋值 (`=`)

- **符号**: `=`
- **行为**: 阻塞赋值是**顺序执行**的。当执行阻塞赋值时，当前语句必须完成，后续的语句才能执行。它类似于传统的顺序编程方式。
- **应用场景**: 阻塞赋值一般用于**组合逻辑**的建模，例如 `always @*` 块中。它常用于描述在同一时钟周期内多个赋值按顺序完成的逻辑。

示例

```
always @(*) begin
    a = b;  // 阻塞赋值
```

```
    c = a;  // 当 a = b 完成后，才会执行 c = a
end
```

在这个例子中，`c` 的值取决于 `a`，而 `a` 的值又取决于 `b`。由于使用的是阻塞赋值，`c` 最终会得到 `b` 的值。

2. 非阻塞赋值 (`<=`)

- **符号**: `<=`
- **行为**: 非阻塞赋值是**并行执行**的。当执行非阻塞赋值时，所有语句的右侧表达式在同一个时钟周期内同时求值，赋值的更新会在时钟周期结束时发生，而不是立即完成。
- **应用场景**: 非阻塞赋值一般用于**时序逻辑**的建模，例如在 `always @(posedge clk)` 块中。它确保在时钟边沿事件中，并行计算的信号正确存储。

示例

```
always @(posedge clk) begin
    a <= b;  // 非阻塞赋值
    c <= a;  // a 和 c 都在时钟沿捕获，但 a 的更新不会立即影响 c
end
```

在这个例子中，`a <= b` 和 `c <= a` 同时在时钟沿发生。在这个时钟周期中，`c` 将不会获得 `a` 的新值，而是它之前的值。因此，`c` 在当前时钟沿仍然保留旧的 `a` 值，`a` 将在下一个时钟沿更新。

3. 主要区别总结

特性	阻塞赋值 (`=`)	非阻塞赋值 (`<=`)
执行顺序	顺序执行	并行执行
更新时机	立即更新，赋值立即生效	在时钟周期结束时生效
常见应用	组合逻辑 (`always @(*)`)	时序逻辑 (`always @(posedge clk)`)
对时序的影响	可能导致错误的时序逻辑	保持时序逻辑一致性

- 在 **组合逻辑**中，应使用**阻塞赋值** (`=`) 来确保逻辑按顺序执行。
- 在 **时序逻辑**中，应使用**非阻塞赋值** (`<=`) 以避免竞争条件，并确保信号在下一个时钟沿时更新。

其它

在实验网站中给出的除法器仿真代码中的 start 一直是 1，这是不对的，因为如果在上一次的运算结束后，start 还是 1，而除数和被除数没有及时改变的话，会将上一次的数据直接作为这一次的输入，这是不对的；

不过也是最后一次计组了，问题不大

思考题

1

双精度浮点数 x,y,z，若 $x=-1.5\times10^{38},y=1.5\times10^{38},z=1.0$，则 $(x+y)+z=$?；$x+(y+z)=$?；两者有区别吗？请解释你的回答。

$x+y=0.0$，所以

$$(x+y)+z=0.0+1.0=1.0$$

;

而对于第二个

$$x+(y+z)=0.0$$

, 因为首先进行的是 $y+z$，在进行浮点数的加法时, 首先要进行阶码对齐,-1.5×10^{38}; 双精度的 $frac$ 部分有 52 位, 而

$$2^{52}-1=4503599627370495<10^{38}$$

这也就是说 1.0 与 y 在做小对大的阶码对齐的时候,1 会一直右移 52 次以上, 直到精度损失把 1 忽略, 所以 $y+z$ 的结果就是 y, 最后再于相反数 x 相加，结果就是 0.0

2

```
float x = SOME_VALUE_0;
float sum = 0.0f;
for(int i = 0; i < SOME_VALUE_1; ++i) sum += x;
printf("%f\n", sum - 100.0f);
```

- 如果 SOME_VALUE_0 := 0.1, SOME_VALUE_1 := 1000，你将得到什么结果？

  会得到-0.000954; 对于这种情况, 我们的预期输出是 0.0, 因为 $0.1\times1000=100.0$, 但是实际输出是-0.000954, 这是因为 0.1 并不能精确表示为浮点数, 在运算时会存在精度损失

- 如果 SOME_VALUE_0 := 0.125, SOME_VALUE_1 := 800，你将得到什么结果？

  会得到 0.000000

  因为 0.125 可以精确表示为浮点数, 所以运算时不会存在精度损失, 最后得到的结果就是 0.0