浙江大学

本科实验报告

课程名称：	计算机组成与设计
姓　　名：	熊子宇
学　　号：	3200105278
学　　院：	竺可桢学院
专　　业：	混合班
指导教师：	姜晓红/叶大源
报告日期：	2022 年 4 月 6 日

实验三 乘法器、除法器、浮点加法器

一、实验目的

1. 实现乘法器并仿真验证
2. 实现除法器并仿真验证
3. 实现浮点加法器并仿真验证

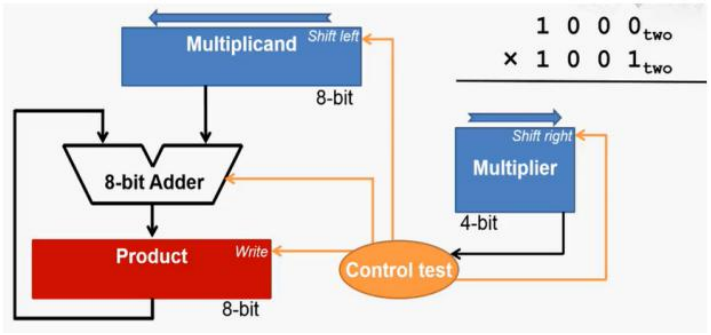
二、实验方法与步骤

1 实现乘法器并仿真验证

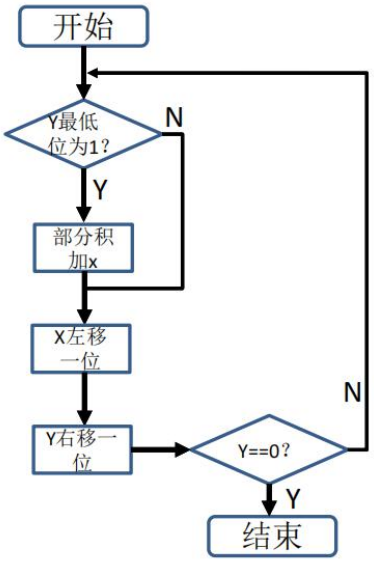
1.1 乘法器原理介绍

理论课上介绍了三种乘法器，其性能不断优化，依次介绍如下：

Version 1：模拟竖式乘法。数据通路与算法设计如下：



V1 乘法器 硬件数据通路

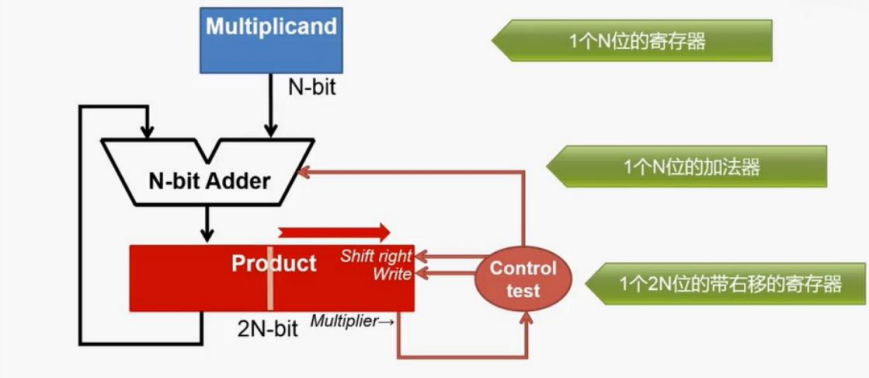


V1 乘法器 算法设计

V1 这种朴素的乘法器的缺陷之一是浪费空间。要实现 n 位乘法器，必须使用 2n 位的 Multiplicand 寄存器、2n 位的 Adder 和 2n 位的 Product 寄存器。最初 Mcand 放在寄存器的低 n 位，而高 n 位闲置未利用。而为了配合 2n 位的 Mcand 和 Product，加法器也必须使用 2n 位。缺陷之二是浪费时间。每一轮循环，都必须要做 shift 两次，而且要做 2n 位的 add。

Version 2：由相对运动原理可知，Multiplicand 左移，Product 不动，相当于 Multiplicand 不动而 Product 右移。这种方法可以将 2n 位 Multiplicand 的空间降低至 n 位，而且 Adder 也可以只使用 n 位。

Version 3：通过观察发现，Product 寄存器低 n 位最开始闲置，而 Product 和 Multiplier 寄存器均右移，因此将 Multiplier 放在 Product 低 n 位，进一步减少寄存器空间。

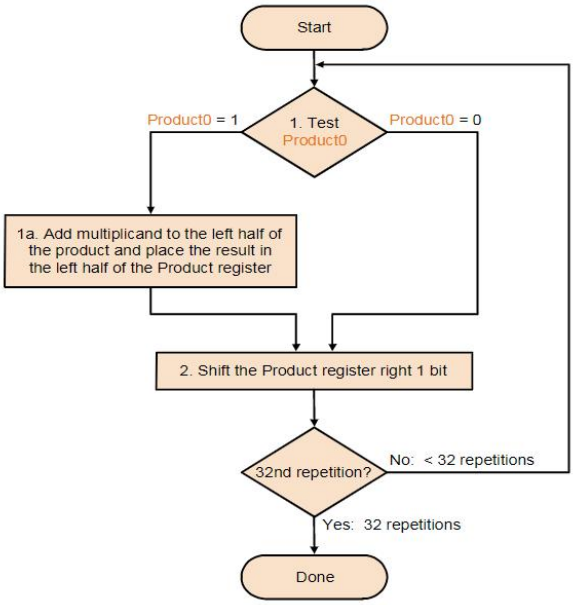


V3 乘法器 硬件数据通路

此外还有性能更好的 Booth 算法，但是我在代码实现时采用的 V3 的方法，因此在这里略过。

1.2 乘法器功能实现

基本思路是结合各种控制流（顺序、循环、条件）来实现下图中的算法。



算法本身没有特别需要说明的。我想要指出我在实现乘法器的过程中遇到的一些问题和解决方法。

第一，控制信号。一共有四个控制信号，clk, rst, start 和 finish。clk 和 rst 表明在马德老师版本的 PPT 中，乘法器是作为一个时序电路的。整个控制流要包裹在 always @(posedge clk or posedge rst) 语句块内。start 和 finish 控制起初对我造成一定问题。随后经过 debug，总结我的逻辑如下：

start 是外界给定的输入信号。start = 0 表示开始计算，start = 1 表示停止计算。

finish 是乘法器的输出信号。finish = 0 表示计算未完成，finish = 1 表示计算完成。

- rst = 1 时，重置 finish = 0。
- 当 start = 0 且 finish = 0 时，表示计算开始且计算未完成，则进入控制流开始计算乘法。

这里我一开始缺少了 finish = 0 的条件，导致算出最终结果后会立刻开始下一轮计算，也即计算不会停止。

- 当 start = 1 时，表示外界命令乘法器停止计算，此时无论有没有计算完成，都应重置 finish = 0。
- 当经过 n 次循环计算完成后，置 finish = 1，表示结束计算。

第二，有符号数的乘法。我代码里采用的方法是，求出 Multiplicand 和 Multiplier 的绝对值，变成正数相乘，最后用异或判断符号，再修改 product 的符号。当我现在写报告的时候发现，事情应该没有这么麻烦，使用补码表示 A 和 B 后进行乘法，结果也是补码，有符号数乘法与无符号数的乘法应当无殊。

第三，其他的小错误。比如初始化 product 寄存器时 product = {0,multiplier};忘记把高 n 位清 0，导致错误。

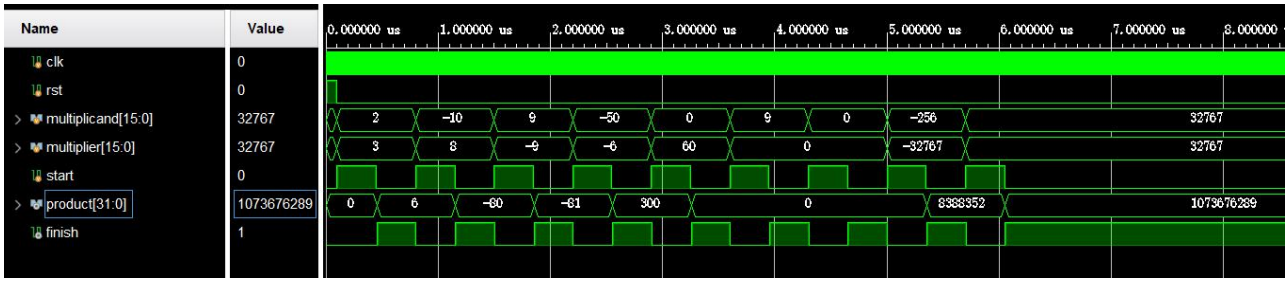
代码如下：

```
 module mul32(
 input clk,
 input rst,
 input [15:0] A,
 input [15:0] B,
 input start,
 output reg [31:0] product,
 output reg finish
  );
  wire signMcand;
  wire signMer;
  wire sign;
  assign signMcand = A[15];
  assign signMer = B[15];
  assign sign = signMcand ^ signMer;
  reg [15:0] multiplicand;
  reg [15:0] multiplier;
  integer i;

  always @(posedge clk or posedge rst) begin
     if (rst) begin
        product = 0;
        finish = 0;
     end
     else if (start == 0 && finish == 0) begin
     //计算开始而且计算未完成，少了 finish 条件循环计算
            multiplicand = signMcand ? ~A+1 : A;
            multiplier = signMer ? ~B+1 : B;//处理负数
            product = {0,multiplier};//忘记把高位清 0，导致错误
            for (i = 0; i < 16; i = i+1) begin
               if (product[0]) begin
            product[31:16] = product[31:16] + multiplicand[15:0];
               end
               product = product >> 1;
            end
            product = sign ? ~product+1 : product;
            finish = 1;
     end
     else if (start) begin finish = 0; end
  end
endmodule
```

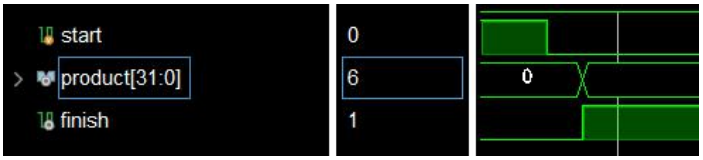
1.3 仿真验证

仿真验证代码均是简单的赋值语句，因此省略，看波形图即可。



首先 rst = 1 重置 product = 0。随后，我设计了正数*正数，正数*负数，负数*正数和负数*负数这四种情况，以及对于乘 0（虽然这并没有意义）的检查，和 0x7FFF（16 位 signed int 最大值）*0x7FFF 的边界情况检验。由仿真波形图可知，各运算结果均正确。

值得注意的是，我实现的乘法器是时序部件，当放大波形图局部时，可见从开始计算（start = 0）到完成计算（finish = 1）的确存在一定的时延。

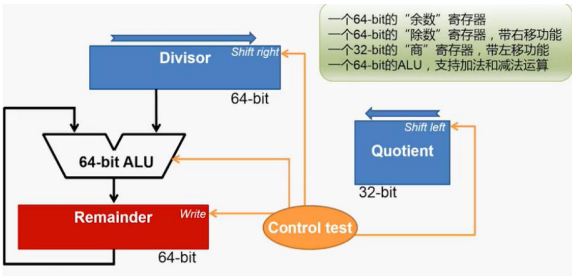


2 实现除法器并仿真验证

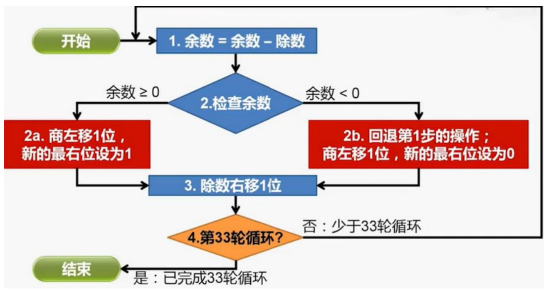
2.1 除法器功能介绍

理论课上介绍了三种除法器。本实验采用 V3 的方法，但将三种方法都介绍如下：

Version 1：模拟竖式除法。数据通路与算法设计如下：



V1 除法器 硬件数据通路

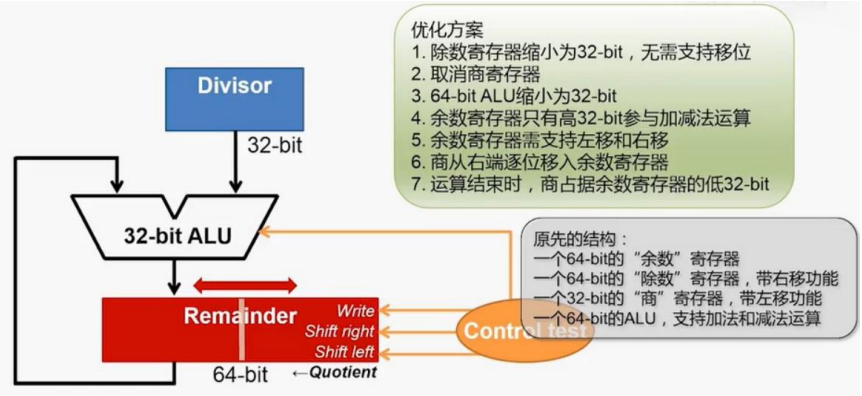


V1 除法器 算法设计

和竖式乘法一样，V1 这种朴素的除法器的缺陷之一是浪费空间。必须使用 2n 位的 Divisor 寄存器、2n 位的 ALU 和 2n 位的 Remainder 寄存器。最初 Divisor 放在高 n 位，Remainder 放在低 n 位，Remainder 寄存器的高 n 位始终闲置。缺陷之二是浪费时间。每一轮循环，都必须要做 shift 两次，而且要做 2n 位的 sub，如果余数<0 还要 add 回来。

Version 2：由相对运动原理可知，Divisor 右移 Remainder 不动，相当于 Divisor 不动而 Remainder 左移。这种方法可以将 2n 位 Divisor 的空间降低至 n 位，而且 ALU 也可以只使用 n 位。

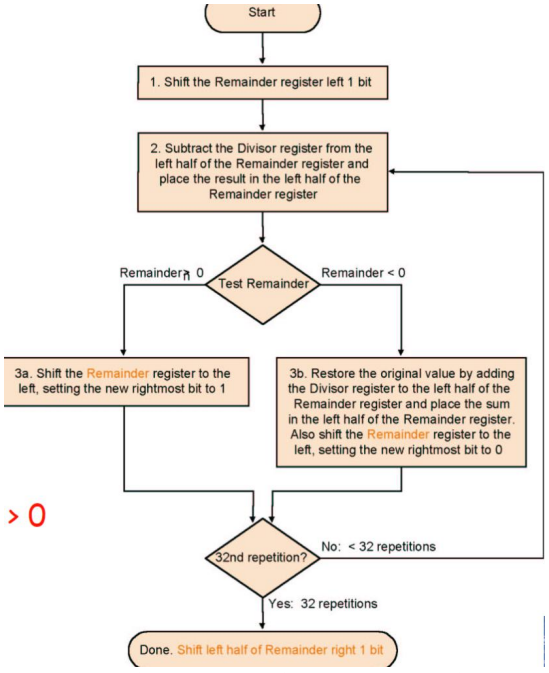
Version 3：同乘法器一样，可以将 Quotient 直接和 Remainder 合并，在 Remainder 左移的同时直接将 Quotient 的结果放在 Remainder 寄存器的最右边。



V3 除法器 硬件数据通路

2.2 除法器功能实现

基本思路是结合各种控制流（顺序、循环、条件）来实现下图中的算法。



算法本身需要注意的小细节：在循环体开始之前，首先要把 Remainder 寄存器左移一位，使得余数和商之间隔开一位；当循环体结束以后，要再把 Remainder 寄存器右移一位，使得余数复位。这点在代码中有所体现：

```
temp = {0,dividend} << 1;
remainder = signR ? ~(temp[31:16] >> 1)+1 : temp[31:16] >> 1;
```

其他需要在实现中注意的是：

第一，除数为 0 的特殊情况。马德老师 PPT 中并没有这个报错信号，我结合了另一个 PPT 进行扩充。引入 `divide_zero` 输出信号。如果 divisor = 0，则直接置其为 1，且置 finish = 1 结束除法。

第二，有符号数的除法。在除法中，负数必须要特殊处理，否则无法输出正常结果。约定 sign of Quotient = x sign ⊕ y sign，sign of remainder = x sign。我采用的方法是，将 Divisor

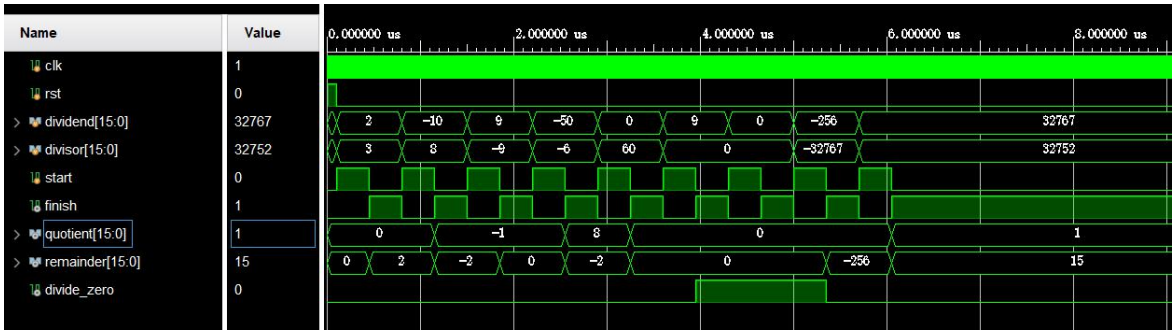
和 Dividend 都转化成正数作除法，最后根据约定修改 Quotient 和 Remainder 的符号。

除法主体代码如下：

```
    ...
    assign signDend = A[15];
    assign signDor = B[15];
    assign signQ = signDend ^ signDor;
    assign signR = signDend;
                ...
                dividend = signDend ? ~A+1 : A;
                divisor = signDor ? ~B+1 : B;//处理负数
                divide_zero = 0;
                if (divisor == 0) begin
                    divide_zero = 1;
                    finish = 1;
                end
                else begin
                    temp = {0,dividend} << 1;
                    for (i = 0; i < 16; i = i+1) begin
                        temp[31:16] = temp[31:16] - divisor;
                        if (temp[31]) begin
                            temp[31:16] = temp[31:16] + divisor;
                            temp = temp << 1;
                        end
                        else begin
                            temp = (temp << 1) + 1;
                        end
                    end
                    quotient = signQ ? ~temp[15:0]+1 : temp[15:0];
remainder = signR ? ~(temp[31:16] >> 1)+1 : temp[31:16] >> 1;
```

2.3 仿真验证

仿真验证代码均是简单的赋值语句，因此省略，看波形图即可。



首先 rst = 1 重置 quotient 和 remainder = 0。随后，我设计了正数/正数、正数/负数、负数/正数、负数/负数、0/某数、某数/0 这六种情况，0x7FFF(16 位 signed int 最大值）/0x7FF0 的边界情况检验。由仿真波形图可知，各运算结果均正确。尤其是当除数为 0 时，会将 divide_zero 置 1。

3 实现浮点加法器并仿真验证

3.1 浮点加法器功能介绍

（1）IEEE754 单精度浮点数的编码方式

在计算机内存中，IEEE754 单精度浮点数编码如下：

符号位	阶码	尾数	表示
0/1	255	非零1xxxx	*NaN* Not a Number
0/1	255	非零0xxxx	*sNaN* Signaling NaN
0	255	0	$+\infty$
1	255	0	$-\infty$
0/1	1~254	f	$(-1)^S \times (1.f) \times 2^{(e-127)}$
0/1	0	f(非零)	$(-1)^S \times (0.f) \times 2^{(-126)}$
0/1	0	0	+0/-0

（2）浮点数加法运算过程

1）0 操作数检查。对 A = 0, B = 0, A = -B 这三种情况进行剪枝处理。

2）处理尾数，获得双符号位补码表示的尾数。双符号位是指用最高位和次高位表示符号位。从左到右的第三位是原来小数点前的一位，第四位到最后一位是小数点后的尾数。整体采用补码表示。这样处理是为了方便后面对结果规格化。

采用双符号位的补码：

对正数：S=00.1×××…×

对负数：S=11.0×××…×

3）比较阶码大小并完成对阶。阶码小的数向大的移位。

4）结果规格化。非规格化数一共有四种模式：

规格化方法

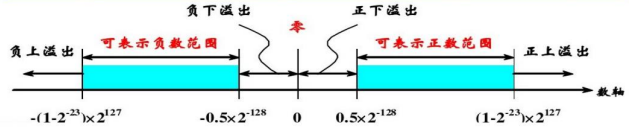
- 00.0XXXX --→ 00.1XXX0 左规
- 11.1XXXX --→ 11.0XXX0 左规
- 01.XXXXX --→ 00.1XXXX 右规
- 10.XXXXX --→ 11.0XXXX 右规

左归表示向左移位，同时阶码减 1；右归表示向右移位，同时阶码加 1。

5）舍入处理。有多种舍入方法，包括就近舍入、朝 0 舍入、向正无穷舍入和向负无穷舍入。由于朝 0 舍入（即简单的截尾）容易积累误差，所以本实验中增加了一位 guard bit，并采用向正无穷舍入，也即对正数来说，只要 guard bit 为 1 则进位 1，对负数则简单截尾。

6）溢出判断。

格式	最小值	最大值
单精度	$E_{min}=1$, M=0, $1.0\times2^{1-127}=2^{-126}$	$E_{max}=254$, f=1.1111…, $1.111\ldots1\times2^{254-127}=2^{127}\times(2-2^{-23})$
双精度	$E_{min}=1$, M=0, $1.0\times2^{1-1023}=2^{-1022}$	$E_{max}=2046$, f=1.1111…,$1.111\ldots1\times2^{2046-1023}=2^{1023}\times(2-2^{-52})$



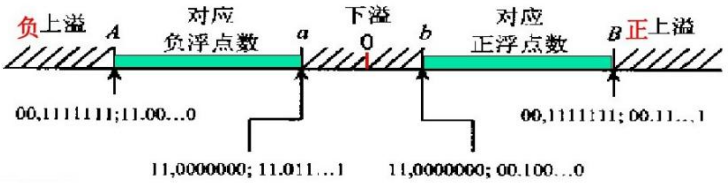
浮点数表示有一定范围。当计算结果的绝对值小于能表示的最小值时，称为下溢出，此时一般不会中断计算，而是返回机器 0 作为结果。

当计算结果的绝对值大于能表示的最大值时，称为上溢出。此时会中断计算，报错上溢出。

溢出判断取决于规格化以后的阶码。阶码同样用双符号位的补码表示。当规格化以后的阶码出现以下两种情形时，分别代表上下溢出，要分别处理。

阶码$[j]_{补}$=01,×××××为上溢，机器停止运算，做中断处理；

阶码$[j]_{补}$=10,×××××为下溢，按机器零处理。



3.2 浮点加法器功能实现

借由浮点数运算的六个步骤，分别实现相关算法。

（0）准备工作，定义临时变量。

```
wire signA;
reg [7:0] expA;
wire [22:0] fracA;
/*省略 B*/
reg expBsubA;
assign signA = A[31];
assign fracA = A[22:0];
assign signB = B[31];
assign fracB = B[22:0];

reg [26:0] pA; //2 符号位 + 原 23 位尾数 + 1guard bit
reg [26:0] pB;
reg [7:0] resExp; //存放结果的阶码，这里使用的是 unsigned 8bit
reg [26:0] tempRes; //存放尾数相加后的临时结果
```

（1）0 操作数处理及剪枝。

对于三种简单情况，直接获得 result。

```
if (A == 0) result = B;
            else if (B == 0) result = A;
            else if (A[30:0] == B[30:0]) result = 0;//剪枝，A = -B
```

（2）处理尾数，获得双符号位补码表示的尾数。以 A 为例：

```
//初始化 pA
   pA = {fracA, 4'h0};
//暂存 expA
   expA = A[30:23];
//把隐藏的小数点前的 1 右移,pA 形如 0.1XXXX（X 是原来的 23 位尾数）
   if (expA >=1 && expA <=254) begin
      pA = pA >> 2;
      sliceA = pA[24:0];
      pA = {1'b0, 1'b1, sliceA};//这里语法有问题，拼接符害人不浅
      expA = expA - 1;
   end
```

```
    //pA 本来是 0.XXXX,补一个 0 进来。变成 0.0XXXX
    else if (expA == 0) begin
        pA = pA >> 1;
    end

//把符号位放进来，如果 A>0 则形如 00.1XXXX 或者 00.XXXX
//如果 A<0 则形如 11.0XXXX 或者 11.XXXX(先取反，再把符号位移进来）
    if (signA) begin
        pA = ~pA+1;
        pA = $signed(pA) >>> 1;
    end
    else pA = pA >> 1;
```

（3）比较阶码大小并完成对阶。

```
expBsubA = expB - expA;//错误 3：没计算 expBsubA
if (expBsubA >= 0) begin // B >= A, A 向右移对齐
    pA = $signed(pA) >>> expBsubA;
    resExp = expB;
    end
else if (expBsubA < 0) begin // B < A, B 向右移对齐 A
    pB = $signed(pB) >>> (~expBsubA+1);
    //expBsubA == 0 则不需要移位操作。一开始没判断出现溢出错误
    resExp = expA; end
```

（4）结果规格化。

```
    tempRes = pA + pB;
    //规格化
    flag = 0;
    //01.XXX -> 00.1XXX, 10.XXX -> 11.0XXX, 右归，指数++
    if (tempRes[26:25] == 2'b10 || tempRes[26:25] == 2'b01)
     begin tempRes = $signed(tempRes) >>> 1; resExp = resExp + 1; end

    //00.0XXX -> 00.1XX,左归，指数--
    else if (tempRes[26:24] == 3'b000) begin
         while (tempRes[26:24] != 3'b001) begin
               tempRes = tempRes << 1;
               if (resExp == 0) flag = 1;
               resExp = resExp - 1;
         end
    end

    //11.1XXX -> 11.0XX, 左归，指数--
    else if (tempRes[26:24] == 3'b111) begin
         while (tempRes[26:24] != 3'b110) begin
               tempRes = tempRes << 1;
               if (resExp == 0) flag = 1;
               resExp = resExp - 1;
         end
    end
```

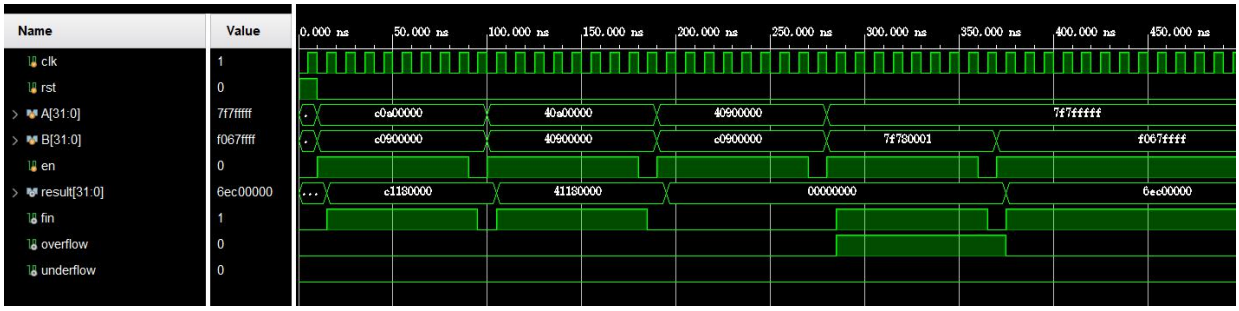
（5）溢出和舍入处理。

```
    //detect underflow or overflow
    if (flag == 1 && tempRes != 0) begin
       underflow = 1;
       result = 0;
    end
    else if (resExp >= 254) begin
```

```
        overflow = 1;
        result = 0;
    end
    else begin
        resExp = resExp + 1;
        result[31] = tempRes[26];//sign bit
        result[30:23] = resExp;
        //截取 tempRes 的[23:1]作为结果的 fraction 部分。负数取补码
        result[22:0] = (result[31]) ? ~tempRes[23:1]+1 : tempRes[23:1];
        if (result[31] == 0 && tempRes[0] == 1) result = result+1;
        //以 tempRes[0]作为 rounding 的依据。向正无穷进位
    end
```

3.3 仿真验证

仿真验证代码均是简单的赋值语句，因此省略，看波形图即可。



我设计了五组仿真。分别是：

第一组，(-5.0)+(-4.5) = -9.0

第二组，5.0 + 4.5 = 9.5

第三组，4.5 + (-4.5) = 0

第四组，上溢出情况。此时置 overflow = 1。

A = 32'b0_11111110_1111_1111_1111_1111_1111_111

B = 32'b0_11111110_1111_0000_0000_0000_0000_001

第五组，有较多位对阶操作。

A = 32'b0_11111110_1111_1111_1111_1111_1111_111

B = 32'b1_11100000_1100_1111_1111_1111_1111_111

由仿真图可见，这五组仿真结果均正确。

三、实验心得

这一部分主要叙述我做 Lab3 时遇到的 Bug 或困难。

问题一：乘法器和除法器上板验证失败。我本打算结合两个版本的 PPT 实现上板显示，但是首先遇到一个 bug，无法通过 implementation，报错 ALUT。在互评时经周杨叶同学指出，该错误出现在助教给出的这部分代码里没有将 div_res, div_rem 和 div_by_0 补齐为 32 位。

```
VGA_muldiv vga_muldiv(
          .dbg_A({16'h0, A}),
          .dbg_B({16'h0, B}),
          .dbg_mul_res(mul_res),
```

```
.dbg_div_res({16'h0,div_res}),//本来没有补齐{16’h0}
.dbg_div_rem({16'h0,div_rem}),
.dbg_div_by_0({31'h0,div_by_0}),
...
```

修改该部分代码后，可以生成 bit 流，但是上板以后所有变量都是 0。由于时间紧迫，我没有继续 debug。我怀疑可能的原因是潘老师 PPT 中乘除法器都是组合逻辑，但是我结合马老师版本后将乘除法器改为了时序逻辑电路（即引入了 clk），在调用时钟的时候可能出现了频率不匹配的问题。

问题二：有符号数和无符号数的除法。对于无符号数的除法，必须要仔细考虑是否存在溢出的可能性。事实上无符号数可能会溢出，有符号数则不会。除法算法 V3 在循环开始前将 Dividend 向左移一位，然后循环 n 次。这个过程中可能会将无符号数的最高位移出寄存器，导致溢出。而有符号数取绝对值后一定能保证最高位为 0，因此不会把有效位移出寄存器，从而不会溢出。我最初没有考虑过这个问题，是在互评过程中与周杨叶和单卓同学交流中才意识到的。

问题三：浮点加法器中，阶码和尾数要采用双符号位补码表示。双符号位补码的尾数有助于规格化，而双符号位补码的阶码有助于判断溢出情况。但是由于最开始写代码的时候思路不够清晰，我只对尾数使用了双符号位补码，而阶码仍然采用的是 8 位的 unsigned int，导致我的代码对于上下溢出的判断较为麻烦（可能还存在某些 bug）。

问题四：语言上的 Bug 和逻辑上的 Bug。在设计浮点加法器时，我遇到至少 3 个比较关键的 Bug。比如说 Verilog 语言中的拼接符。`pA = {1'b0, 1'b1, pA};`不会报错，但是会给原来的 `pA` 左边补 `2` 个 `0`。必须要使得`{}`内的位数和等号左侧的位数相等，才能成功赋值。所以一种可行的写法如下：

```
sliceA = pA[24:0];
pA = {1'b0, 1'b1, sliceA};
```

问题五：`Debug` 过程中的中间变量显示。我发现使用 `Vivado debug` 较为麻烦，只有设置为 `output` 的变量才会在波形仿真图上显示。但往往最开始测试时不会一次成功，需要知道中间变量的变化。因此要将中间变量设置为 `output`，并且在 `tb.v` 中引入形参。