Lab03
复杂操作实现

-乘法器、除法器、浮点加法

赵莎

College of Computer Science and Technology
Zhejiang University
szhao@zju.edu.cn

2024

Course Outline

- 一、实验目的
- 二、实验环境
- 三、实验目标及任务

实验目的

1. 复习二进制加减、乘除的基本法则
2. 掌握补码的基本原理和作用
3. 了解浮点数的表示方法及加法运算法则
4. 进一步了解计算机系统的复杂运算操作

实验环境

- **实验设备**

  1. 计算机（Intel Core i5以上，4GB内存以上）系统

  2. NEXYS A7开发板

  3. Xilinx VIVADO2017.4及以上开发工具

- **材料**

  无

实验目标及任务

- 目标：熟悉二进制原码补码的概念，了解二进制加减乘除的原理，掌握浮点加法的操作实现
- 任务一：设计实现乘法器
- 任务二：设计实现除法器
- 任务三：设计实现浮点加法器（*选做*）

- 任务一：设计实现乘法器

- （整数）乘法器原理介绍

整数的基本概念

- 整数在IEEE 的规定上有，短整数short integer , 中整数integer 和 长整数long integer， 它们之间的关系如下

整数	字节空间	取值范围
短整数	一个字节	-127 ~ 127
中整数	两个字节	-32767~32767
长整数	和四个字节	-2147483647~2147483647

整数的基本概念

- 短整数的最高位是符号位，符号位的正负表示了该值是“正还是负”。正值的表示方法很简单，反之负值的表示方法是以补码来表示。

+127 亦即8'b0111_1111;

+4 亦即8'b0000_0100;

-127 亦即8'b1000_0001;

-4 亦即8'b1111_1100;

- 符号位0------正
- 符号位1------负

补码在英文又叫2nd implementation ，其本质是“正值的求反又加一”的操作。

8'b0000_0100; // 正值4
8'b1111_1011; // 求反
8'b1111_1100; // 加1， 负值4

乘法的基本概念

- 被乘数x为1000，乘数y为1001，下面的乘法过程是手工运算的一个步骤，而计算机在做乘法时就是模拟手工运算的执行过程。

multiplicand

multiplier

```
      1000
×     1001
----------
      1000
     0000
    0000
   1000
----------
   1001000
```

product

- 运算过程的进一步调整：

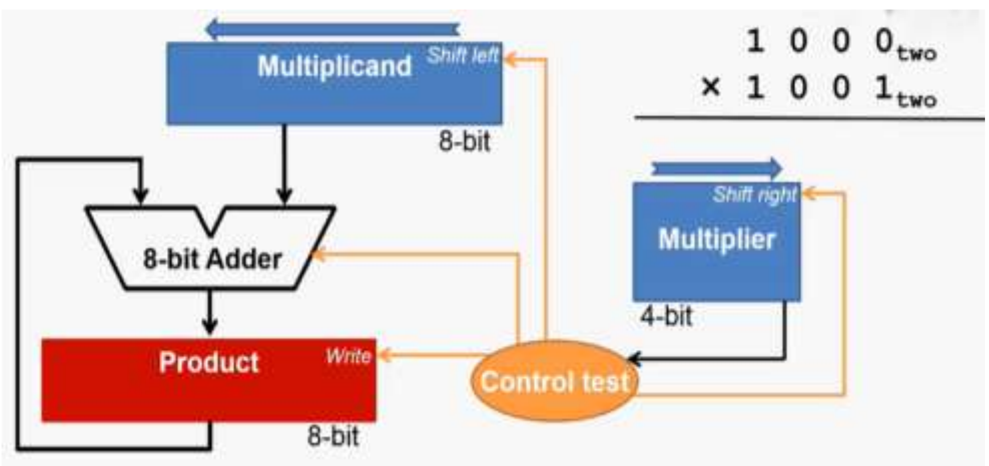
```
      1000
   x  1001
----------
      1000
     0000
    0000
   1000
----------
   1001000
```

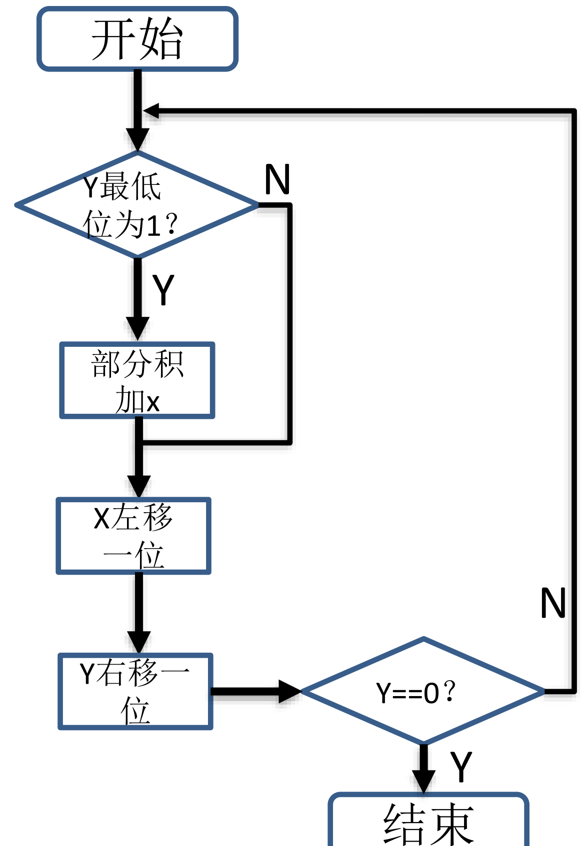
每个中间结果产生后直接与当前的乘积累加

每产生一个中间结果被乘数向左移动一位

乘法的硬件实现

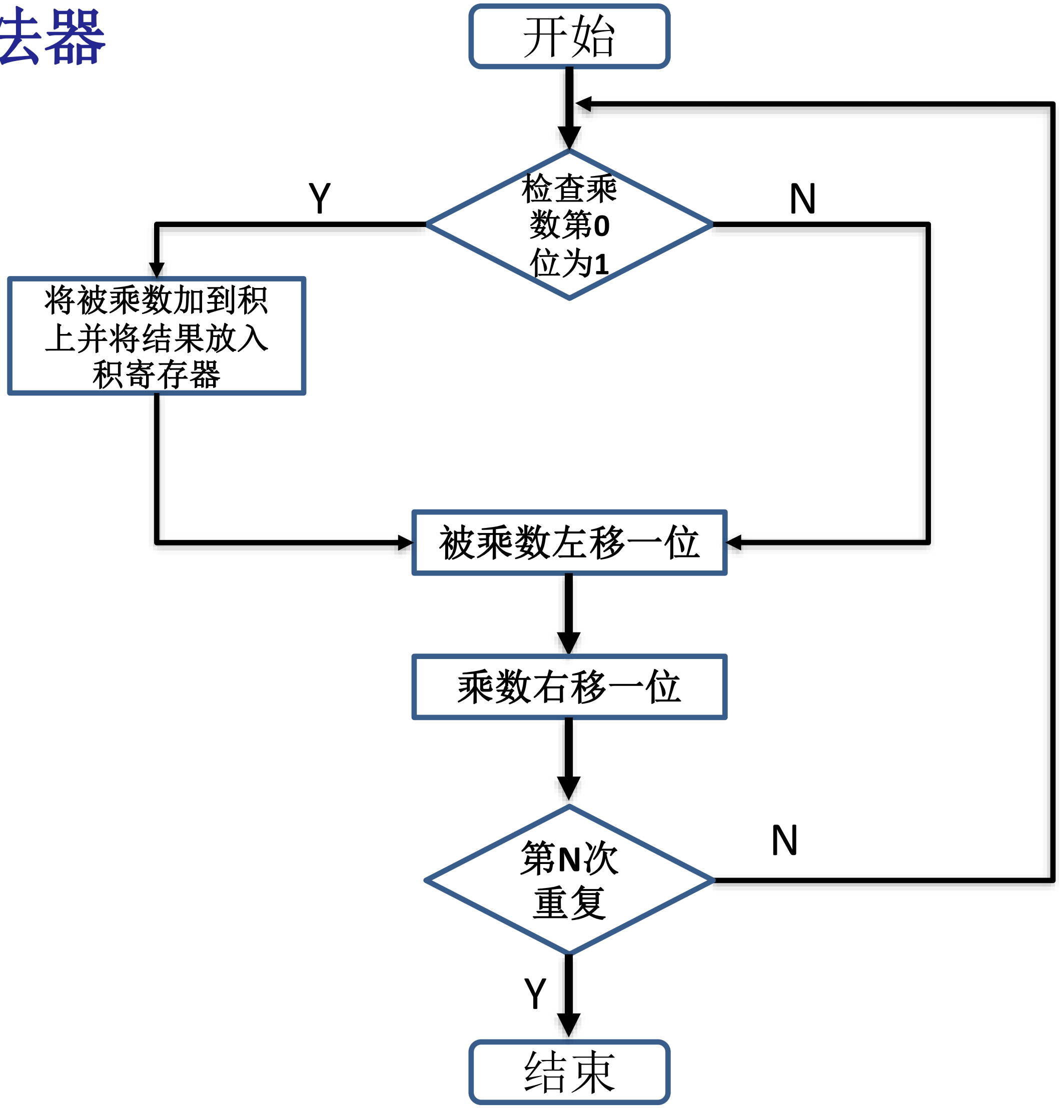
- 因为是两个4位数相乘，所以结果应该是四个数加和得到的。先判断y的最低位是0还是1，如果是1，则需要把x加到部分积上，若为0，则需要把0加到部分积上（实际上加0的这个过程计算机并不执行，因为加0对部分积没有影响），x左移一位，之后再让y右移一位，若y为0，则循环结束，否则继续此循环过程。流程图如下。



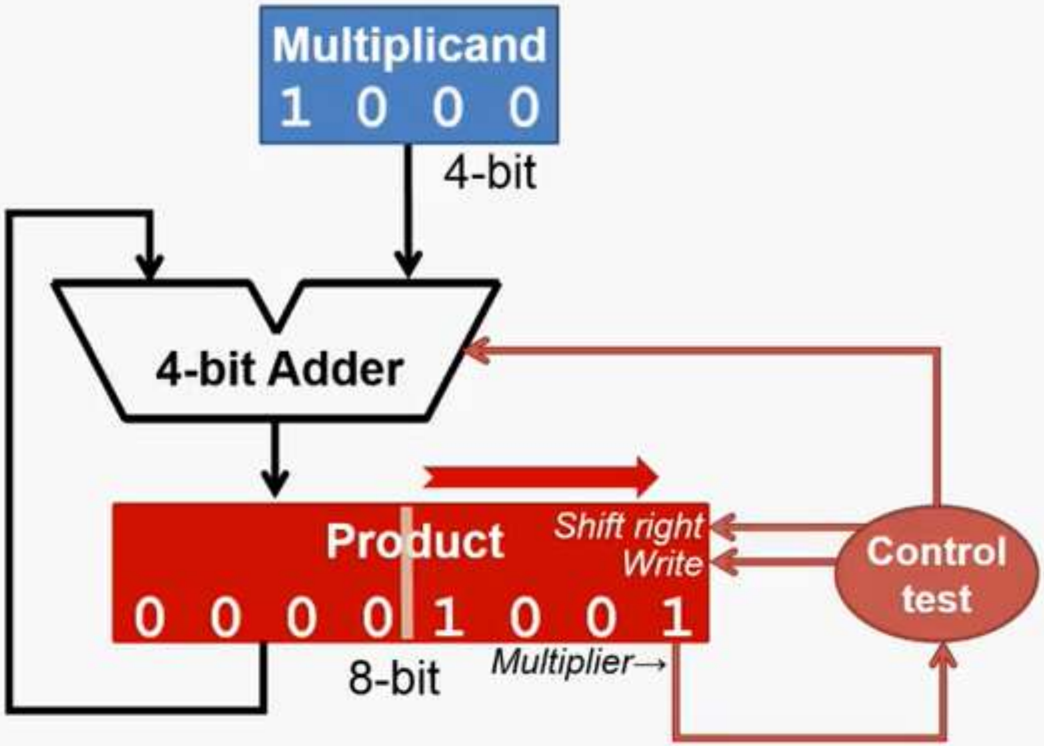


乘法的硬件实现

- N位乘法器



乘法的优化实现-------减少不必要的硬件资源



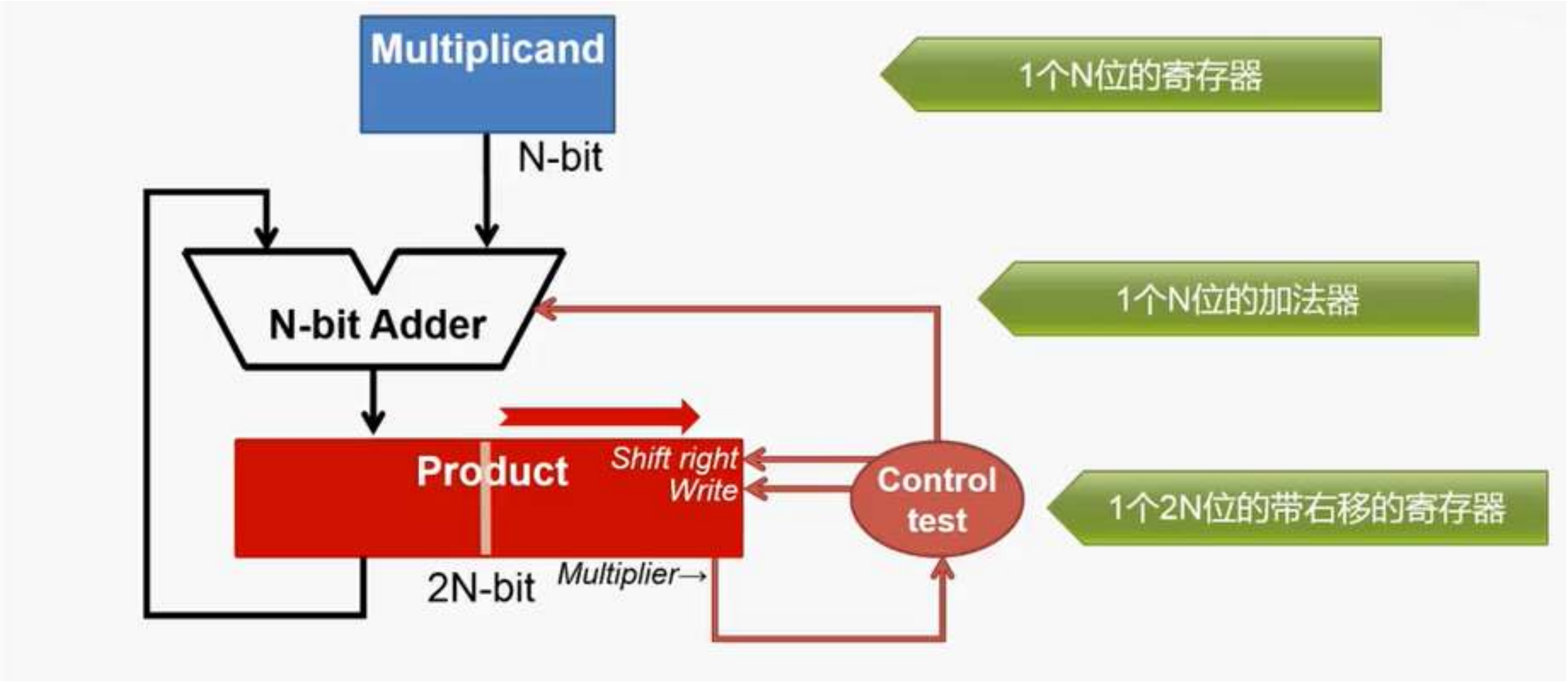
"被乘数寄存器"缩减为4位而且取消左移功能

取消"乘数寄存器"，乘数初始置于"乘积寄存器"低4位

"乘积寄存器"增加右移功能乘积初始值置于其中高4位，随着运算过程不断右移

"加法器"缩减为4位宽，"乘积寄存器"只有高4位参与运算

乘法的优化实现-------N位乘法器的实现结构



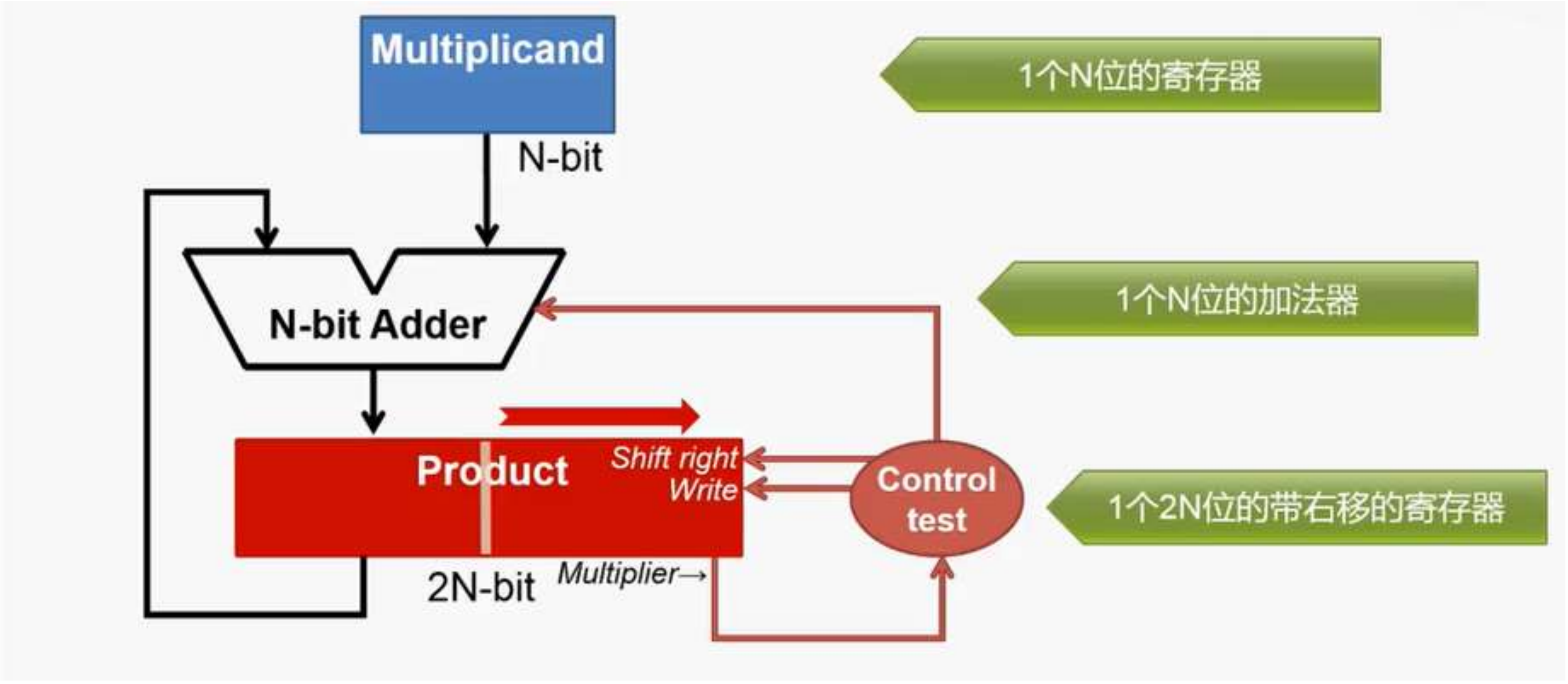
带符号整数乘法

- **z = x * y中，x是被乘数，在Verilog代码中 multiplicand表示，y是乘数，在代码中用multiplier表示。因为x和y都是带符号数，所以应该是用补码乘法，但是如果对x和y求绝对值，让两个绝对值相乘，然后再判断正负，效果和补码乘法是相同的。**

- （整数）乘法器实现

乘法的优化实现

- 采用改良版的硬件算法实现32位无符号数乘法



32位整数乘法：mul32.v

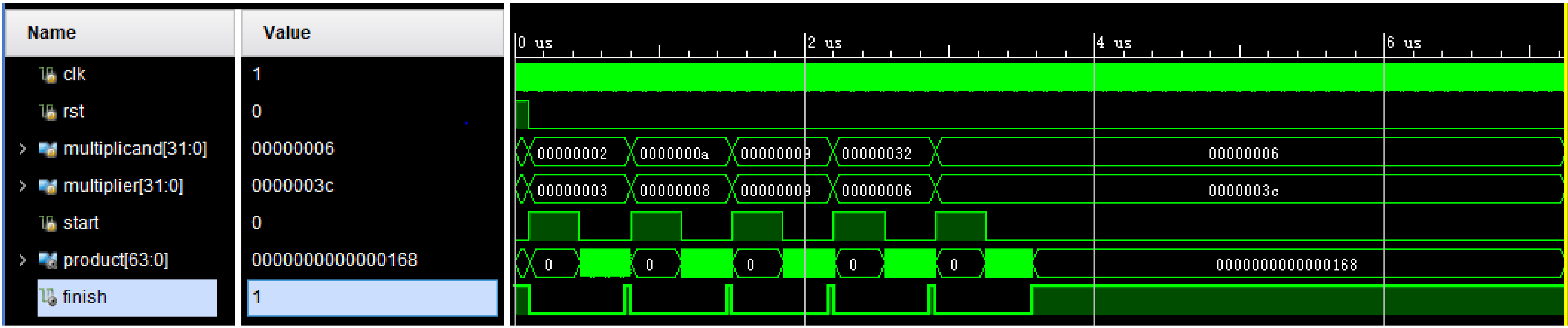
```
module mul32(
  input  clk,
  input rst
  input [31:0] multiplicand,   //被乘数
  input [31:0] multiplier,      //乘数
  input  start,                 //计算开始
  output reg[63:0] product   //积
  output finish                 //计算完成
    );
……………………
```

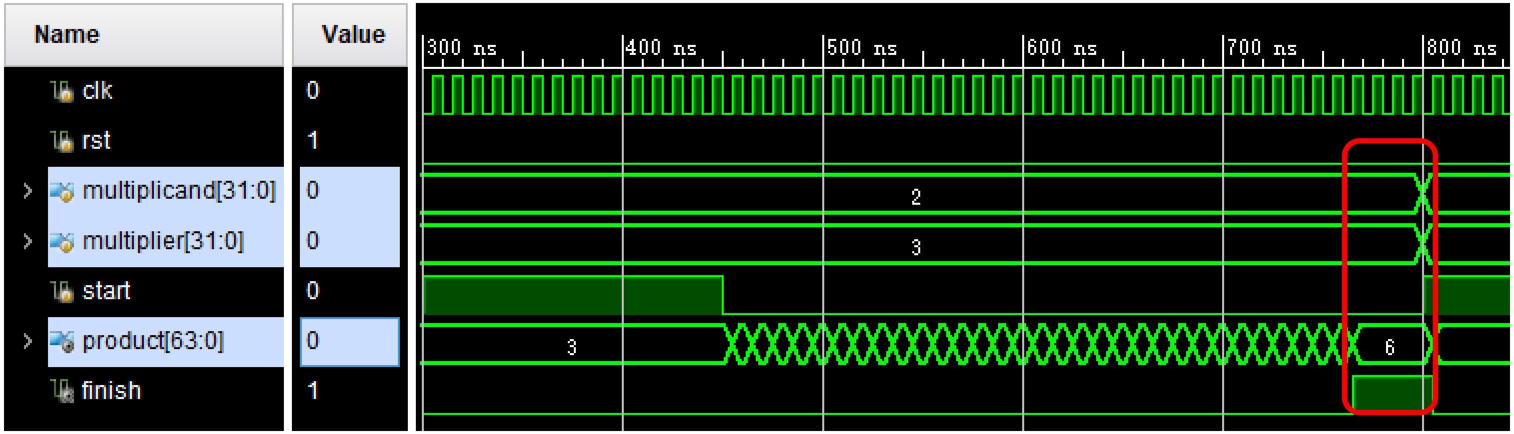
32位整数乘法：mul32.v

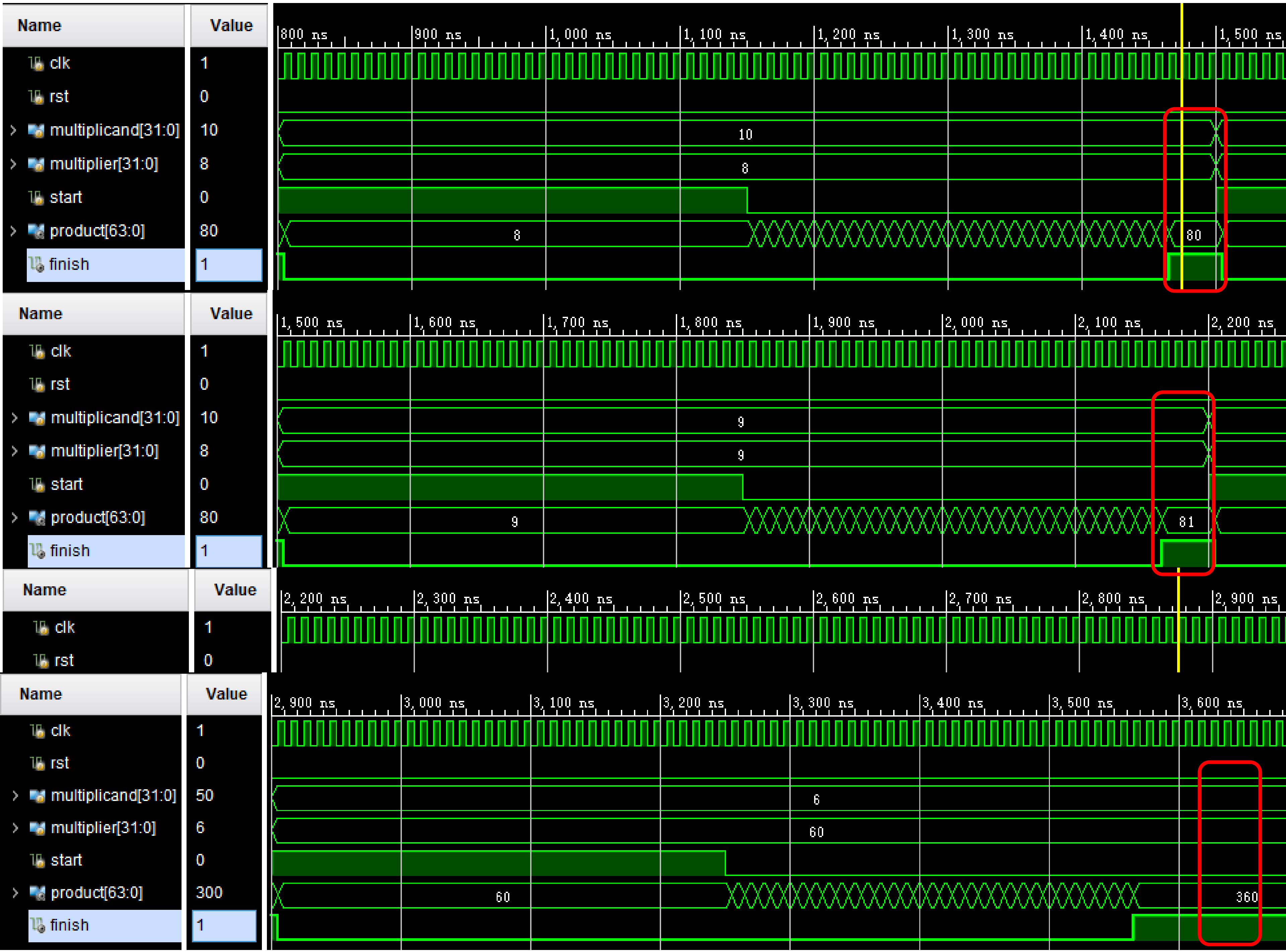
Testbench采用固定输入激励以便观察结果

```
module tb();
    reg clk;
    reg rst;
    reg[31:0] multiplicand;
    reg[31:0] multiplier;
    reg start;
    wire[63:0] product;
    wire finish;

   initial begin
.......................
   multiplicand = 32'd2;
   multiplier     = 32’d3;
   ......
   multiplicand = 32'd10;
   multiplier     = 32'd8;
   ......
   multiplicand = 32'd9;
   multiplier     = 32'd9;
   ......
   multiplicand = 32'd50;
   multiplier     = 32'd6;
   ......
   multiplicand = 32'd6;
   multiplier     = 32'd60;
   #4000 $stop();
   end

   mul32 mul32_u(
.................
);
endmodule
```

32位整数乘法：mul32.v





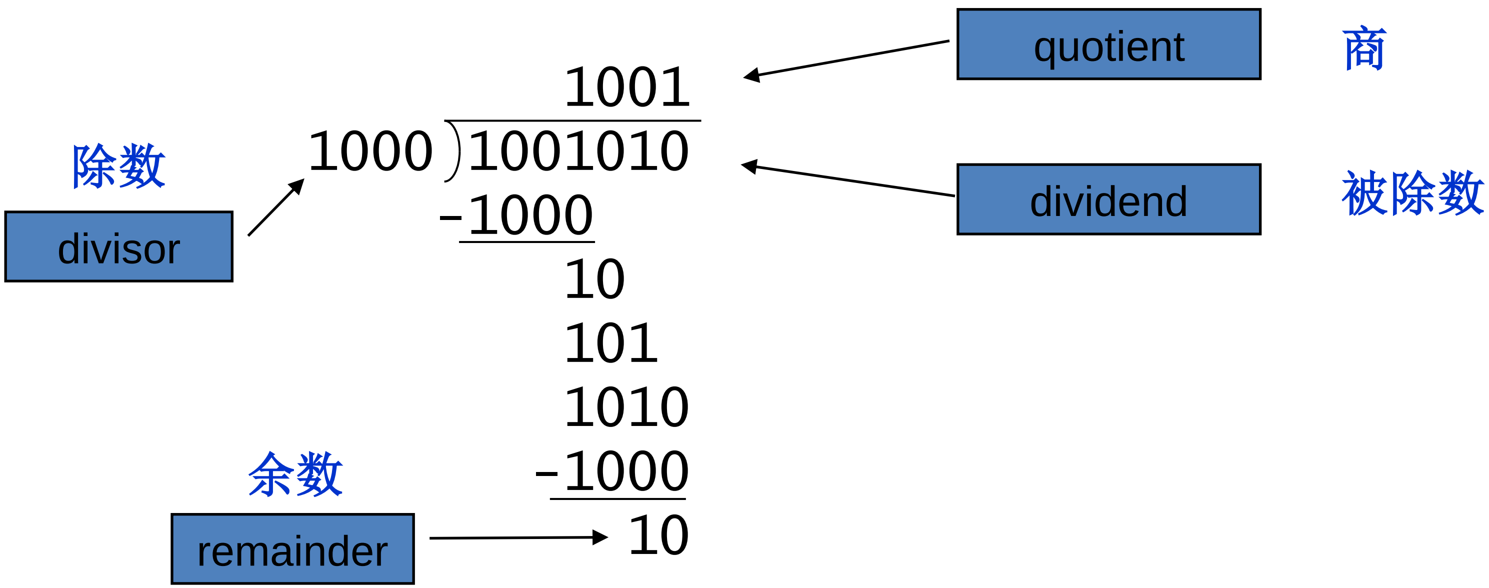

Name
Value
clk 1
rst 0
multiplicand[31:0] 10
multiplier[31:0] 8
start 0
product[63:0] 80
finish 1
800 ns
900 ns
1, 000 ns
1, 100 ns
1, 200 ns
1, 300 ns
1, 400 ns
1, 500 ns
10
8
8
80
Name
Value
clk 1
rst 0
multiplicand[31:0] 10
multiplier[31:0] 8
start 0
product[63:0] 80
finish 1
1, 500 ns
1, 600 ns
1, 700 ns
1, 800 ns
1, 900 ns
2, 000 ns
2, 100 ns
2, 200 ns
9
9
9
81
Name
Value
clk 1
rst 0
2, 200 ns
2, 300 ns
2, 400 ns
2, 500 ns
2, 600 ns
2, 700 ns
2, 800 ns
2, 900 ns
Name
Value
clk 1
rst 0
multiplicand[31:0] 50
multiplier[31:0] 6
start 0
product[63:0] 300
finish 1
2, 900 ns
3, 000 ns
3, 100 ns
3, 200 ns
3, 300 ns
3, 400 ns
3, 500 ns
3, 600 ns
6
60
60
360

- 任务二：设计实现除法器

- （整数）除法器原理介绍

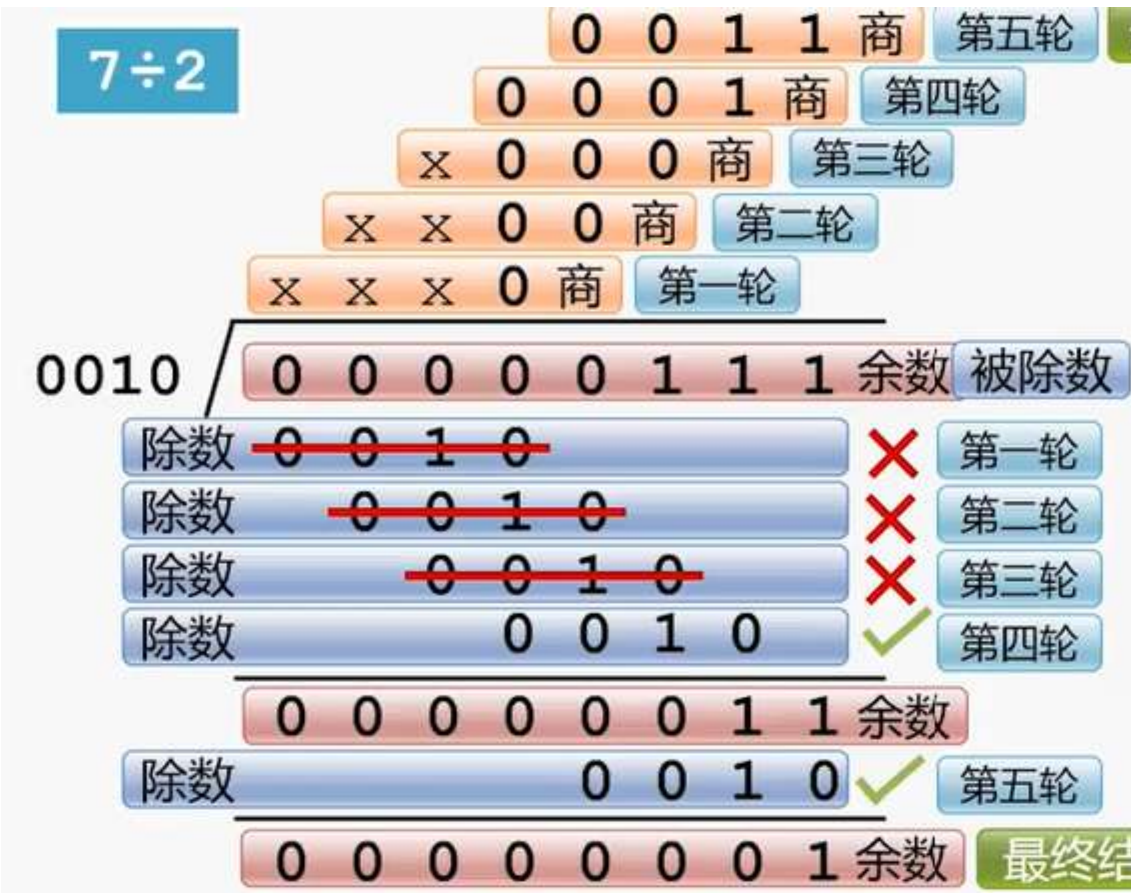
除法的基本概念

- 被除数x为1001010，除数y为1000，下面的除法过程是手工运算的一个步骤，而计算机在做除法时就是模拟手工运算的执行过程。



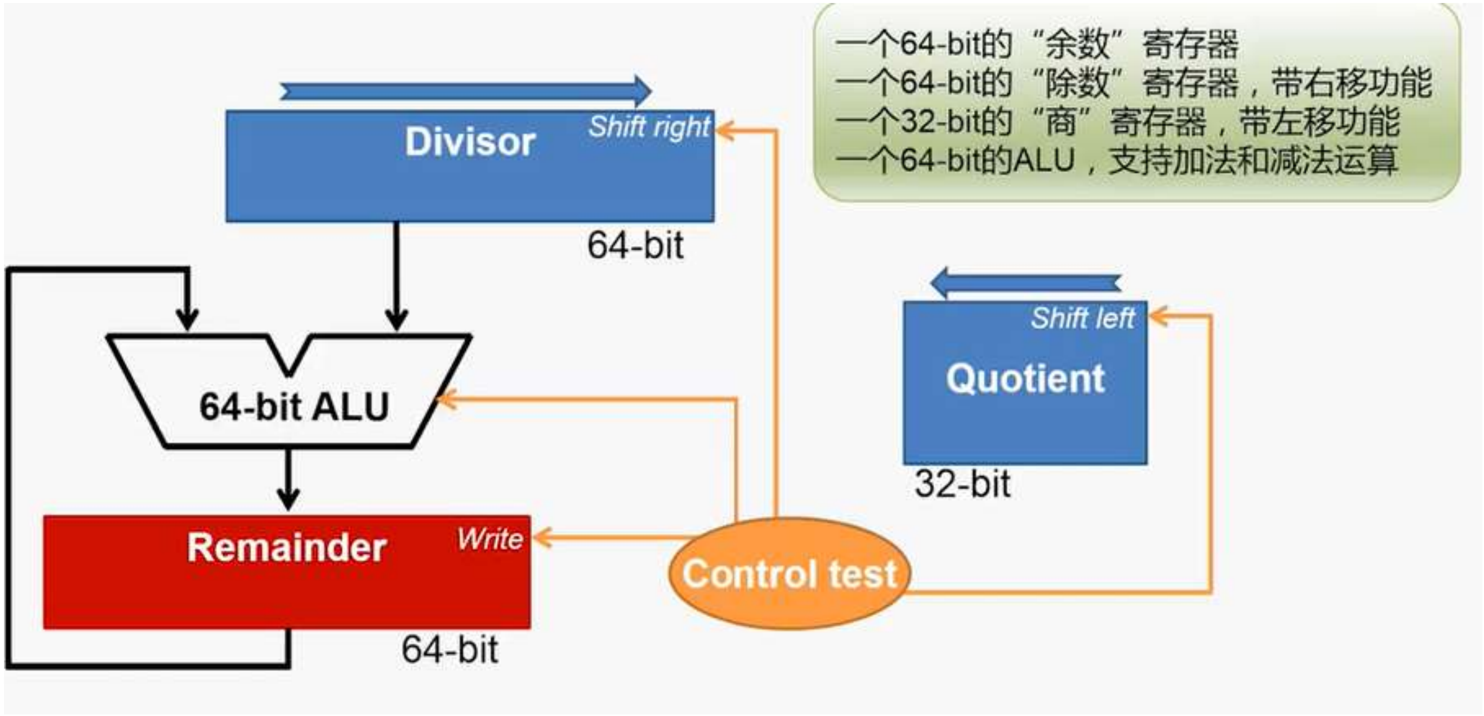
Dividend = Quotient × Divisor + Remainder

除法的基本概念



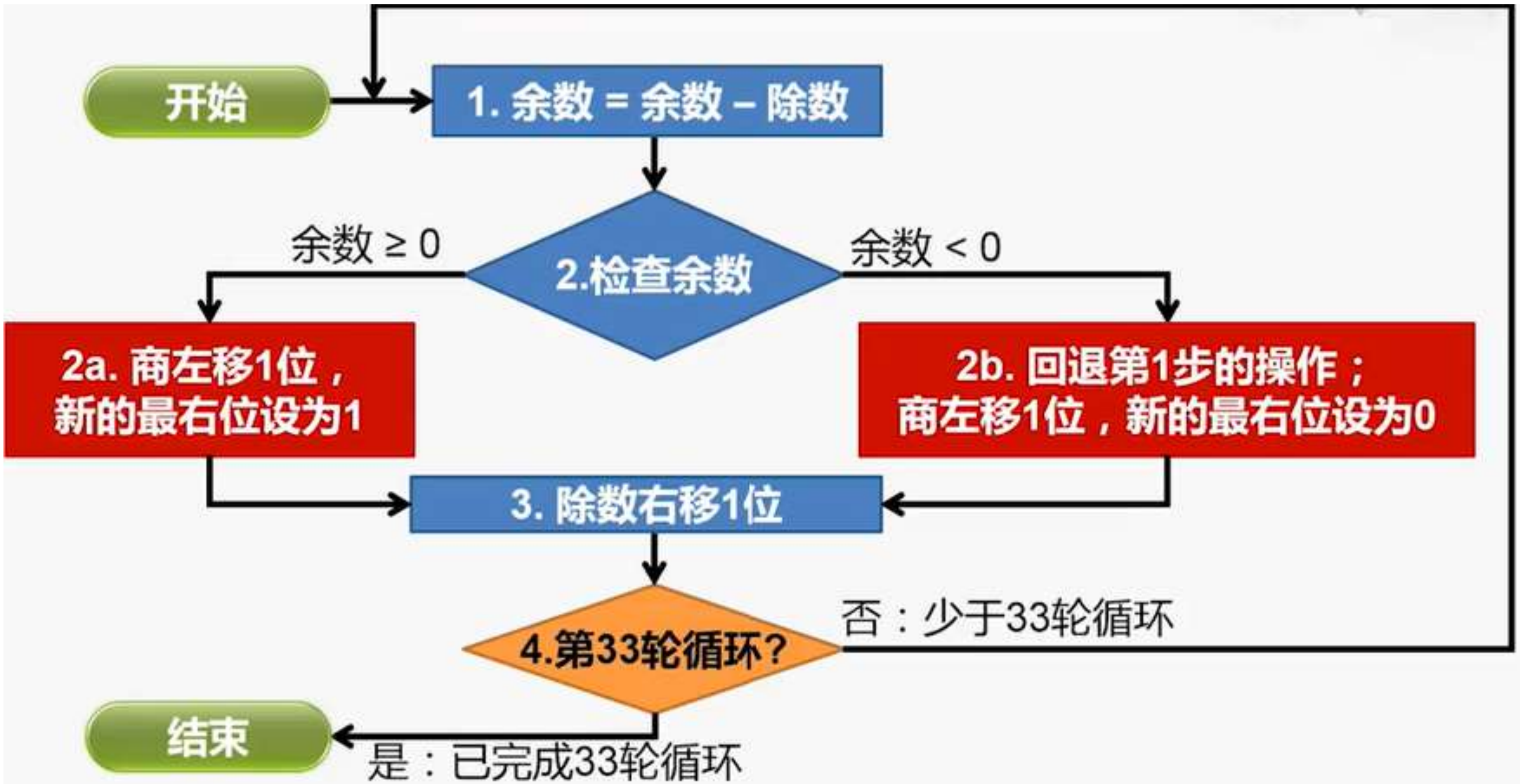
	十进制	二进制	位宽
被除数	7	00000111	8-bit
注：可以视为初始时的余数			
除数	2	0010	4-bit
注：可视为不断右移，并和被除数相减			
余数	1	00000001	8-bit
注：可与被除数共享一个寄存器			
商	3	0011	4-bit
注：每个bit依次生成，可视为不断左移			

除法的硬件实现



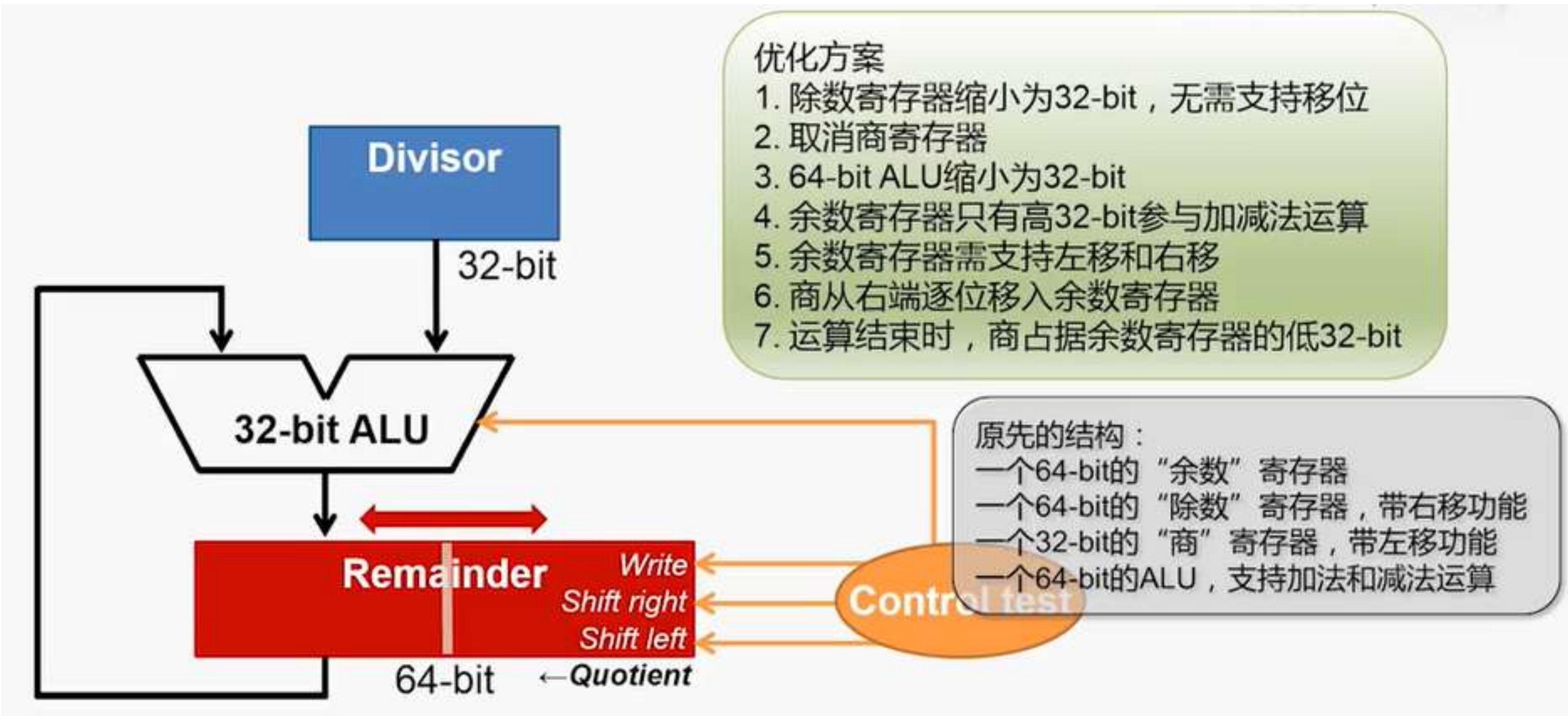
一个64-bit的“余数”寄存器
一个64-bit的“除数”寄存器，带右移功能
一个32-bit的“商”寄存器，带左移功能
一个64-bit的ALU，支持加法和减法运算
Divisor
Shift right
64-bit
64-bit ALU
Remainder
Write
64-bit
Shift left
Quotient
32-bit
Control test

除法的硬件实现

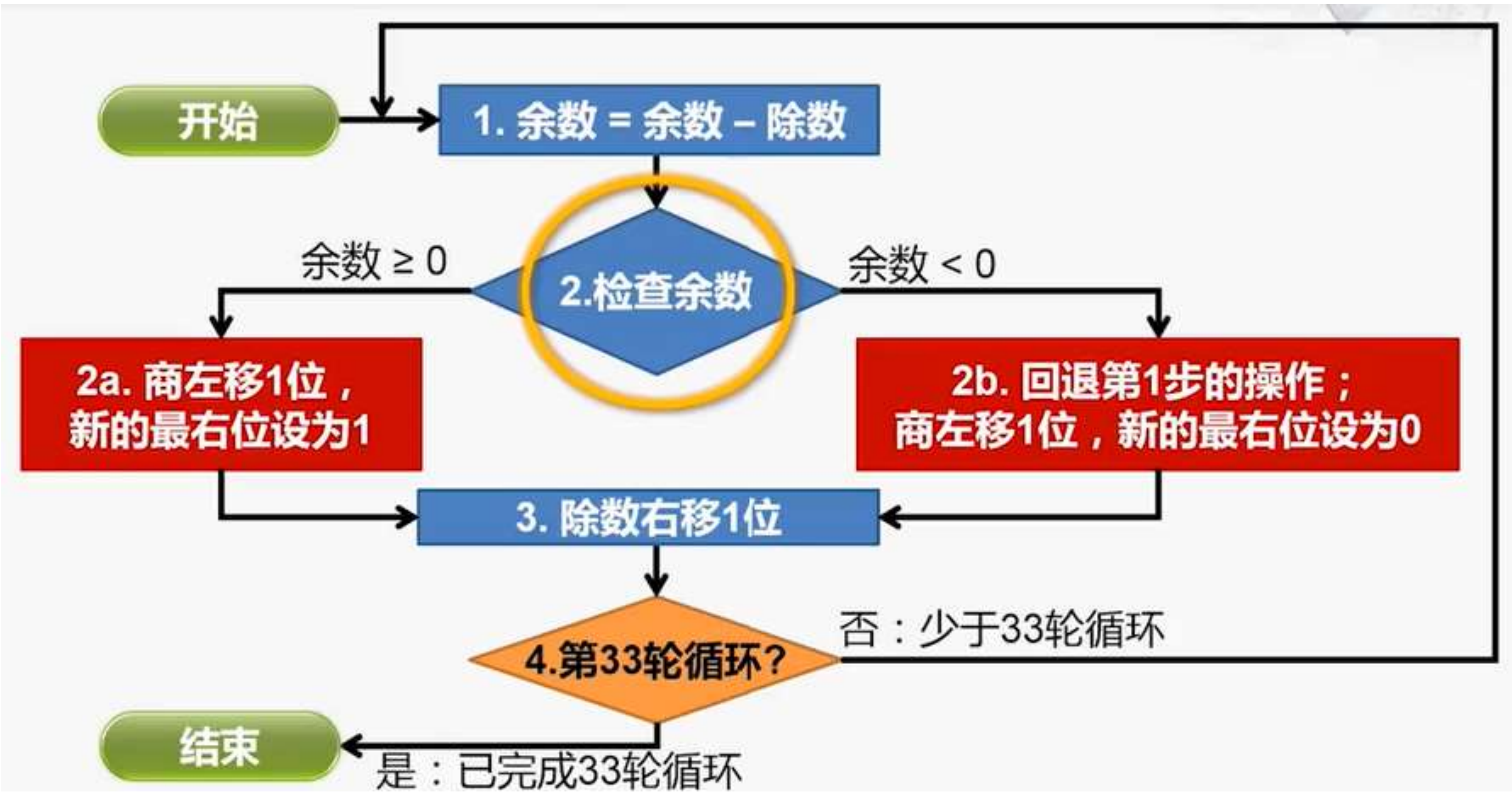


开始
1. 余数 = 余数 – 除数
余数 ≥ 0
2.检查余数
余数 < 0
2a. 商左移1位，
新的最右位设为1
2b. 回退第1步的操作；
商左移1位，新的最右位设为0
3. 除数右移1位
否：少于33轮循环
4.第33轮循环？
结束
是：已完成33轮循环

除法的优化实现


Divisor
32-bit
32-bit ALU
Remainder
Write
Shift right
Shift left
64-bit
←Quotient
Control test
优化方案
1. 除数寄存器缩小为32-bit，无需支持移位
2. 取消商寄存器
3. 64-bit ALU缩小为32-bit
4. 余数寄存器只有高32-bit参与加减法运算
5. 余数寄存器需支持左移和右移
6. 商从右端逐位移入余数寄存器
7. 运算结束时，商占据余数寄存器的低32-bit
原先的结构：
一个64-bit的“余数”寄存器
一个64-bit的“除数”寄存器，带右移功能
一个32-bit的“商”寄存器，带左移功能
一个64-bit的ALU，支持加法和减法运算

除法的优化实现



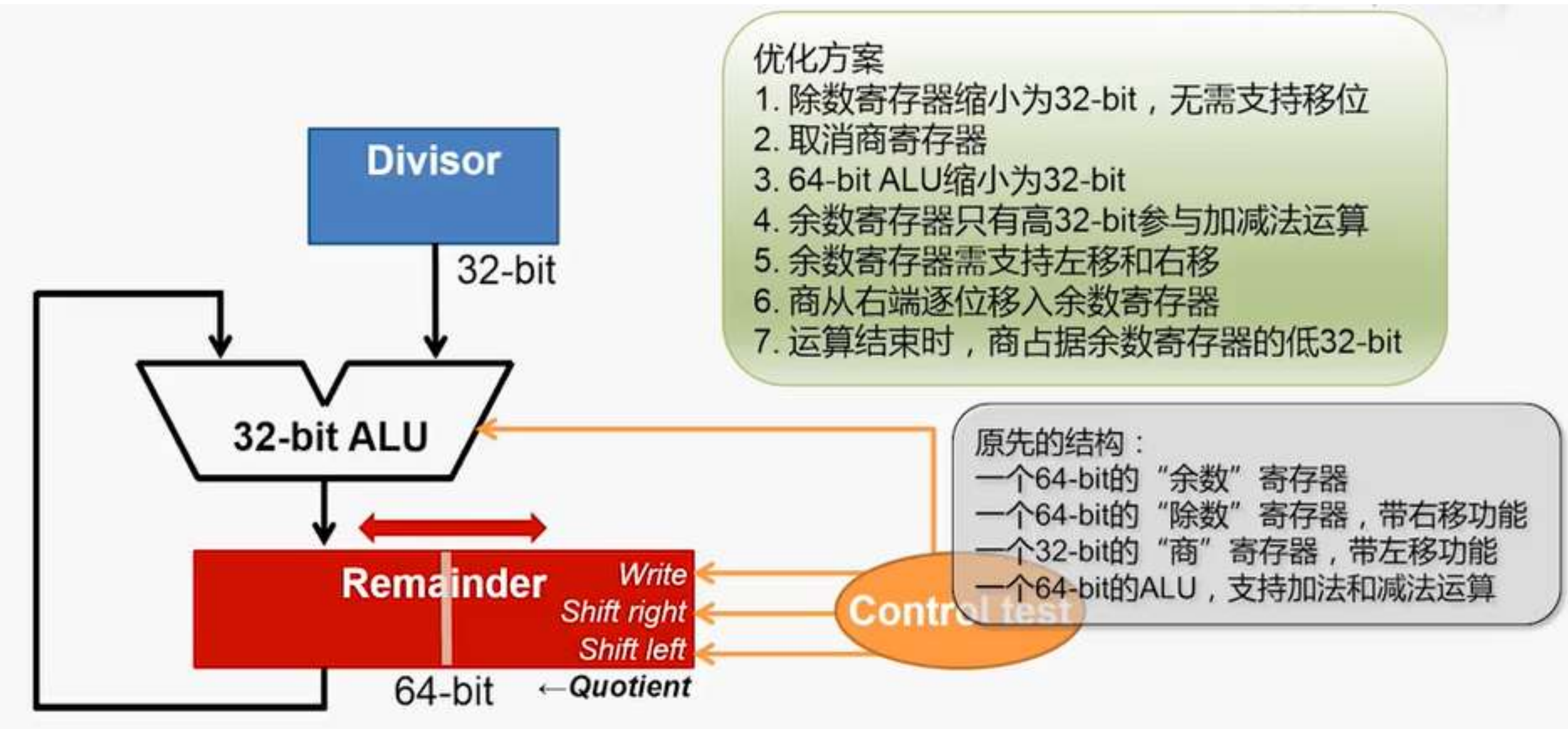
开始
1. 余数 = 余数 – 除数
2.检查余数
余数 ≥ 0
余数 < 0
2a. 商左移1位，新的最右位设为1
2b. 回退第1步的操作；商左移1位，新的最右位设为0
3. 除数右移1位
4.第33轮循环？
否：少于33轮循环
是：已完成33轮循环
结束

带符号定点整数除法

- z = x /y中，x是被除数，y是除数。因为x和y都是带符号数，所以应该是用补码除法，但是如果对x和y求绝对值，让两个绝对值相除，然后再判断正负，效果和补码乘法是相同。

- 除法器实现（整数）

除法的实现



优化方案
1. 除数寄存器缩小为32-bit，无需支持移位
2. 取消商寄存器
3. 64-bit ALU缩小为32-bit
4. 余数寄存器只有高32-bit参与加减法运算
5. 余数寄存器需支持左移和右移
6. 商从右端逐位移入余数寄存器
7. 运算结束时，商占据余数寄存器的低32-bit
Divisor
32-bit
32-bit ALU
Remainder
Write
Shift right
Shift left
64-bit
←Quotient
Control test
原先的结构：
一个64-bit的“余数”寄存器
一个64-bit的“除数”寄存器，带右移功能
一个32-bit的“商”寄存器，带左移功能
一个64-bit的ALU，支持加法和减法运算

32位整数除法：div32.v

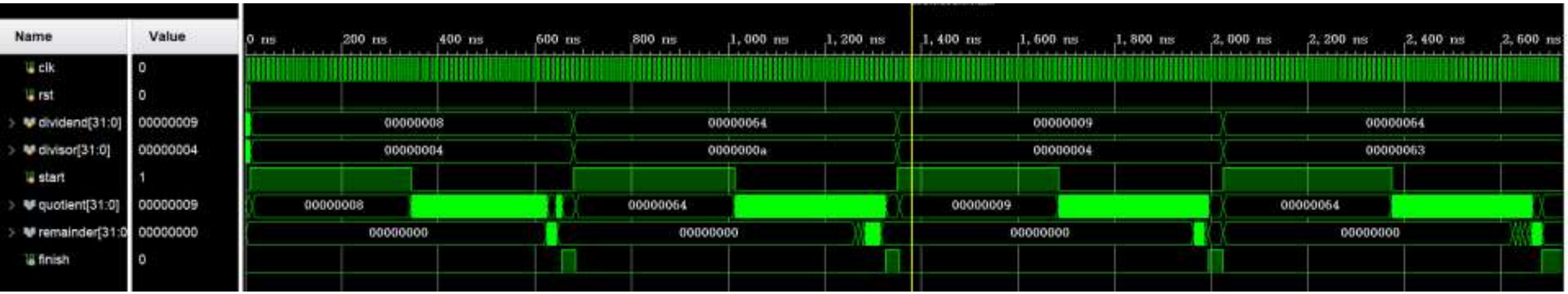
```
module div32(
 input   clk,
 input  rst,
 input start,
 input[31:0] dividend,                    //被除数
 input[31:0] divisor,                     //除数
 output  [31:0] quotient,                 //商
 output  [31:0] remainder                 //余数
 output reg finish
);
……………………
```

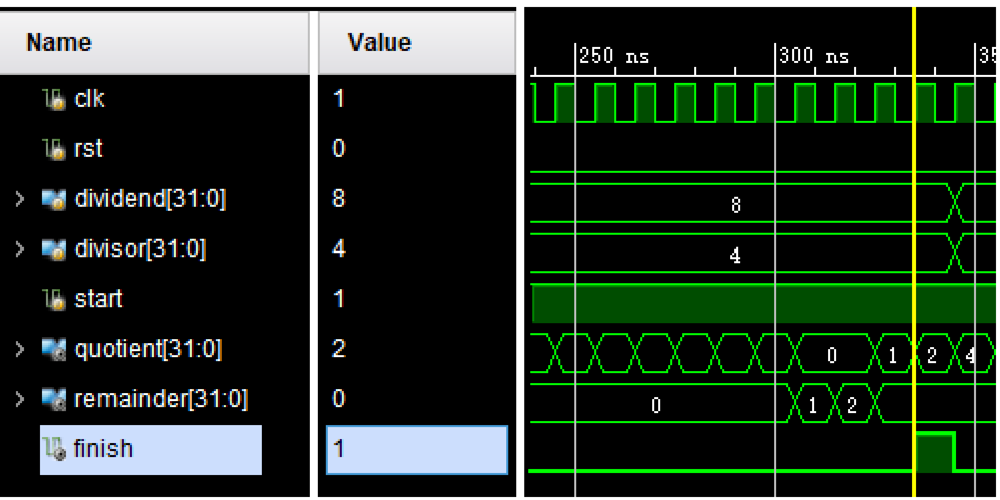
32位整数除法：div32.v

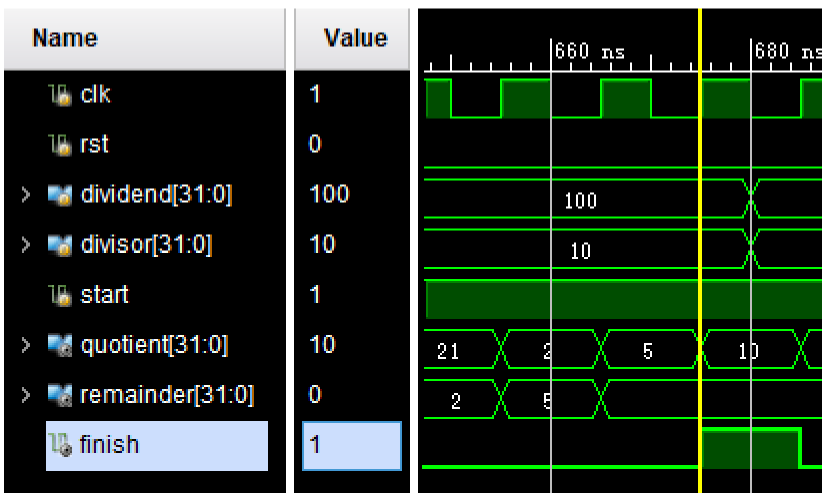
Testbench采用固定输入激励以便观察结果

```
module div32_tb( );
   reg clk;
   reg rst;
   reg [31:0] divident;
   reg [31:0] divisor;
   wire [31:0] quotient;
   wire [31:0] remainder;
…………
   initial begin
   divident = 32'd8;
   divisor    = 32'd4;
   #335
   divident = 32'd100;
   divisor    = 32'd10;
   #335
   divident = 32'd9;
   divisor   = 32'd4;
   #335
   ………………

div32 u_div32
   (
   .divident (divident),
   .divisor  (divisor ),
    ………………
   .quotient  (quotient),
   .remainder (remainder)
   );
endmodule
```

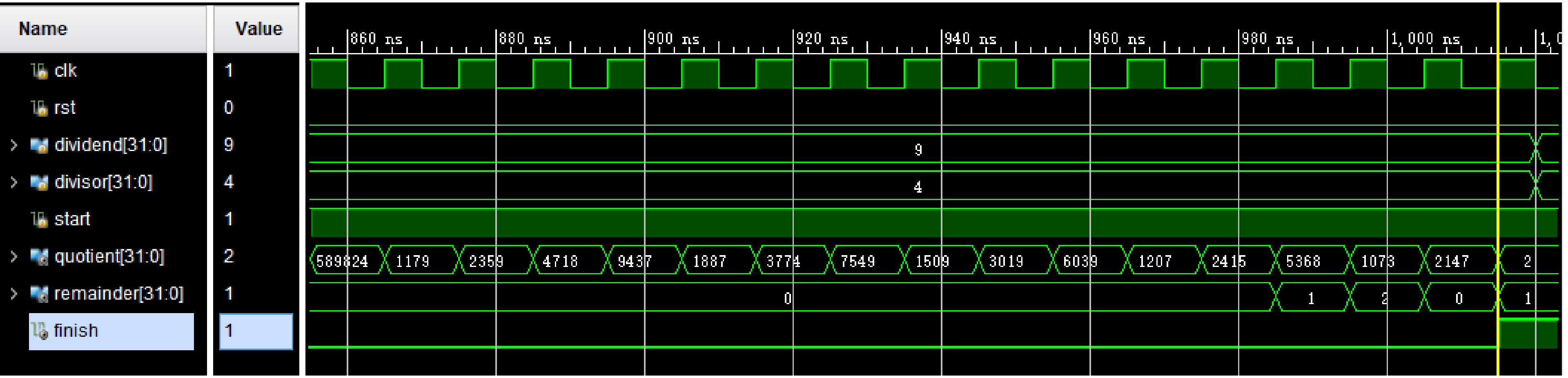
32位整数除法：div32.v

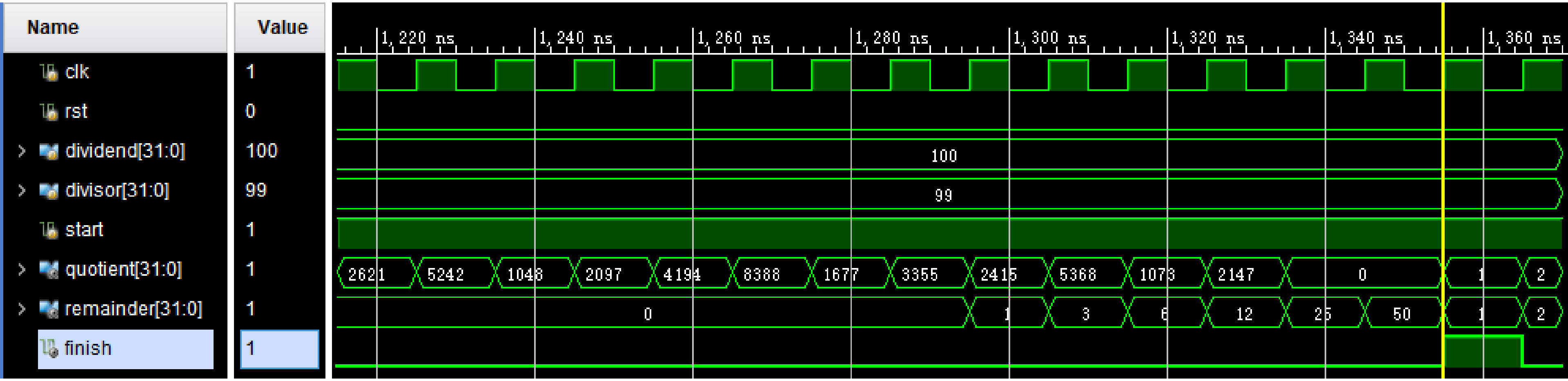






32位整数除法：div32.v





- 任务三：设计实现浮点加法器（*选做*）

- 浮点加法器原理介绍

浮点数的基本概念

一、浮点数的表示

$9\times10^{-28}=0.9\times10^{-27}$

$2\times10^{33}=0.2\times10^{34}$

任意一个十进制数 N 可以写成

$N=10^{E}\times M$ （十进制表示）

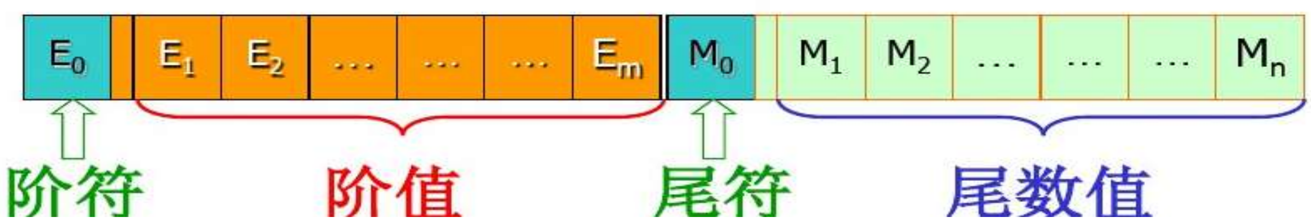
计算机中一个任意进制数 N 可以写成

$$N=R^{e}\times m=2^{E}\times M=2^{\pm e}\times(\pm m)$$

m ： 尾数，是一个纯小数。

e ： 浮点的指数，是一个整数。

R ： 基数，对于二进计数值的机器是一个常数，一般规定 R 为2，8或16



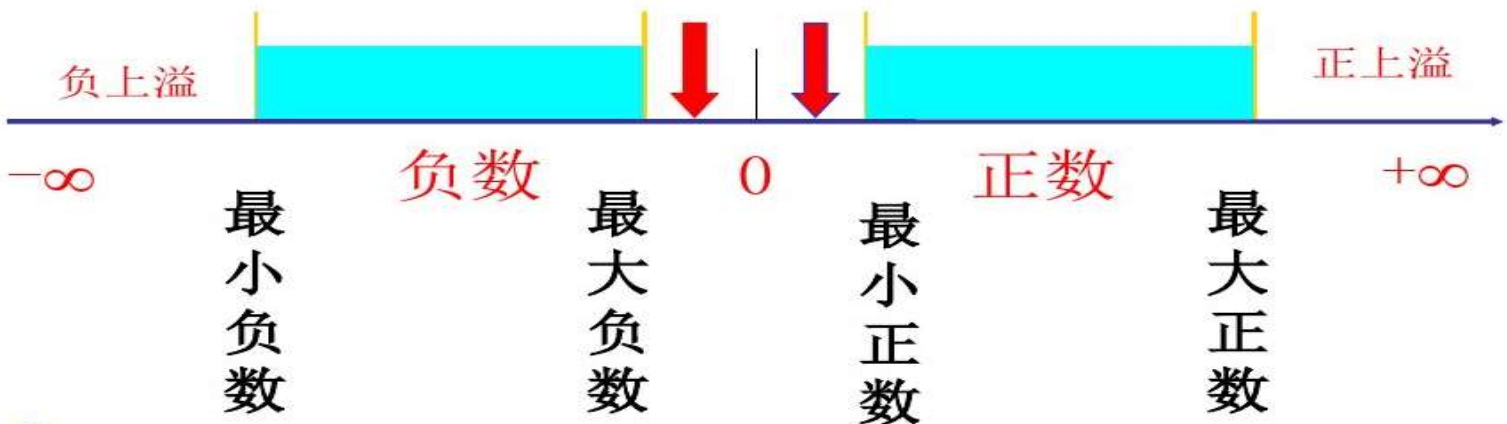
浮点数的基本概念

一个机器浮点数由阶码和尾数及其符号位组成：

尾数：用定点小数表示，给出有效数字的位数，决定了浮点数的表示精度

阶码：用定点整数形式表示，指明小数点在数据中的位置，决定了浮点数的表示范围。

浮点数的表示范围



- $N=2^E\times M$
- $|N|\to\infty$ 产生正上溢或者负上溢
- $|N|\to 0$ 产生正下溢或者负下溢

浮点数的基本概念

- 机器字长一定时，阶码越长，表示范围越大，精度越低
- 浮点数表示范围比定点数大，精度高
- 8位定点小数可表示的范围
  - 0.0000001 --- 0.1111111
  - 1/128 --- 127/128
- 设阶码2位，尾数4位
  - 可表示2^{-11}*0.0001 --- 2^{11}*0.1111
  - 0.0000001 --- 111.1
- 设阶码3位，尾数3位
  - 可表示2^{-111}*0.001 --- 2^{111}*0.111
  - 0.0000000001 --- 1110000

浮点数的基本概念

二、浮点数规格化

浮点数是数学中实数的子集合，由一个纯小数乘上一个指数值来组成。

一个浮点数有不同的表示：

0.5；　0.05×10^{1}；　0.005×10^{2}；　50×10^{-2}

为提高数据的表示精度，需做规格化处理。

在计算机内，其纯小数部分被称为浮点数的尾数，对非 0 值的浮点数，要求尾数的绝对值必须 >= 1/2，即尾数域的最高有效位应为1，称满足这种表示要求的浮点数为规格化表示：

0.1000101010

把不满足这一表示要求的尾数，变成满足这一要求的尾数的操作过程，叫作浮点数的规格化处理，通过尾数移位和修改阶码实现。

浮点数的基本概念

规格化目的：

为了提高数据的表示精度

为了数据表示的唯一性

尾数为R进制的规格化：

绝对值大于或等于1/R

二进制原码的规格化数的表现形式：

正数 0.1xxxxxx

负数 1.1xxxxxx

补码尾数的规格化的表现形式：尾数的最高位与符号位相反。

正数 0.1xxxxxx

负数 1.0xxxxxx

浮点数的基本概念

例： 对数据123_{10}作规格化浮点数的编码，假定1位符号位，基数为2，阶码5位，采用移码，尾数10位，采用补码。

解： $123_{10}=1111011_2=0.1111011000_2\times2^7$

$[7]_{移}=10000+00111=10111$

$[0.1111011000]_{补}=0.1111011000$

$[123]_{浮}=1011\ 1\ 0\ 11\ 1101\ 1000=\text{BBD8H}$

浮点数的基本概念

三 、浮点数的标准格式IEEE754

为便于软件移植，使用 IEEE（电气和电子工程师协会）标准IEEE754 标准：尾数用原码；阶码用“移码”；基为2。

32位

31	30　　23	22　　0
S	E	M

64位

63	62　　52	51　　0
S	E	M

S——尾数符号，0正1负；

M——尾数，纯小数表示，小数点放在尾数域的最前面。采用原码表示。

E——阶码，采用“移码”表示（移码可表示阶符）；

阶符采用隐含方式，即采用移码方法来表示正负指数。

浮点数的基本概念

规格化浮点数的真值

32位浮点数格式：

31	30　　23	22　　　　0
S	E	M

一个规格化的32位浮点数 x 的真值为：

$$x = (-1)^s \times (1.M) \times 2^{E-127} \qquad e = E - 127$$

一个规格化的64位浮点数 x 的真值为：

$$x = (-1)^s \times (1.M) \times 2^{E-1023}$$

这里e是真值， E 是机器数

1.隐藏位技术

原码非0值浮点数的尾数数值最高位必定为 1，则在保存浮点数到内存前，通过尾数左移， 强行把该位去掉，用同样多的位数能多存一位二进制数，有利于提高数据表示精度，称这种处理方案使用了隐藏位技术。

当然，在取回这样的浮点数到运算器执行运算时，必须先恢复该隐藏位。

2.阶码用“移码”偏移值127而不是128

$E_{min}=1$， $E_{max}=254/2046$

浮点数的基本概念

例： 若浮点数 x 的二进制存储格式为$(41360000)_{16}$，求其32位浮点数的十进制值。

解： 0100,0001,0011,0110,0000,0000,0000,0000

数符：0

阶码：1000,0010

尾数：011,0110,0000,0000,0000,0000

指数e＝阶码－127＝10000010－01111111 ＝00000011=$(3)_{10}$

包括隐藏位1的尾数：

$1.M = 1.011\ 0110\ 0000\ 0000\ 0000\ 0000 = 1.011011$

于是有 $x = (-1)^s \times 1.M \times 2^e$

$$= +(1.011011) \times 2^3 = +1011.011 = (11.375)_{10}$$

浮点数的基本概念

例: 将十进制数20.59375转换成32位浮点数的二进制格式来存储。

解: 首先分别将整数和分数部分转换成二进制数:

$$20.59375 = 10100.10011$$

然后移动小数点，使其在第1，2位之间

$$10100.10011 = 1.010010011 \times 2^4$$

$$e = 4$$

于是得到: $e = E - 127$

$S = 0$，$E = 4 + 127 = 131 = 1000,0011$，$M = 010010011$

最后得到32位浮点数的二进制存储格式为

0100 0001 1010 0100 1100 0000 0000 0000 $= (41A4C000)_{16}$

浮点数的基本概念

例：将十进制数-0.75表示成单精度的IEEE 754标准代码。

解：$-0.75 = -3/4 = -0.11_2 = -1.1 \times 2^{-1}$

$= (-1)^1 \times (1 + 0.1000\ 0000\ 0000\ 0000\ 0000\ 000) \times 2^{-1}$

$= (-1)^1 \times (1 + 0.1000\ 0000\ 0000\ 0000\ 0000\ 000) \times 2^{126-127}$

$s=1$，$E=126_{10} = 01111110_2$，$F=1000 \ldots 000$。

1 011, 1111, 0 100, 0000, 0000, 0000, 0000, 0000
B F 4 0 0 0 0 0 H

浮点数的基本概念

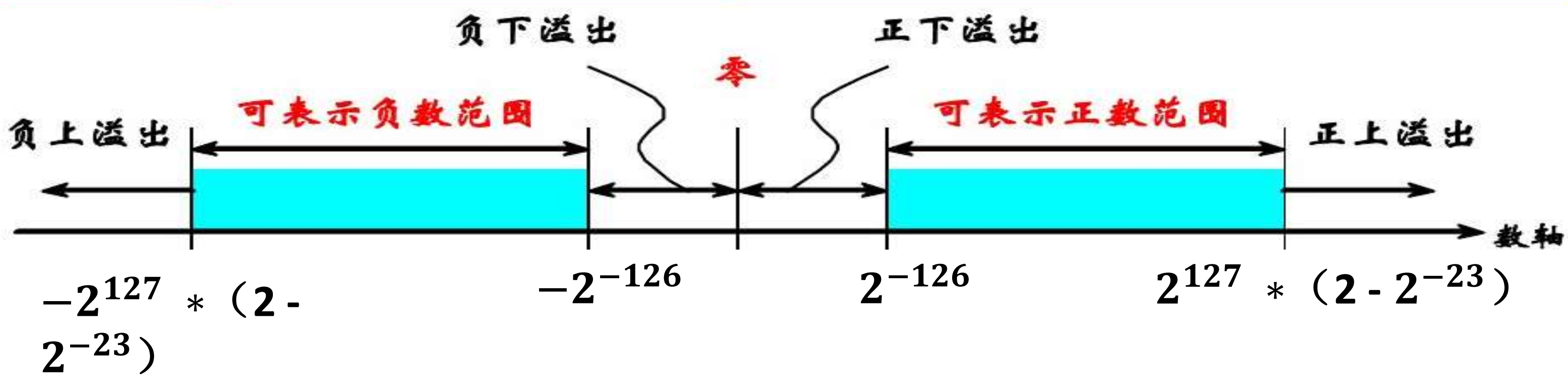
单精度浮点数编码格式

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	$+0/-0$

浮点数的基本概念

IEEE754 规格化浮点数表示范围

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



浮点数的加减法

浮点加减法运算

设有两个浮点数 x 和 y，它们分别为:

$$x = 2^{Ex} \cdot M_x$$

$$y = 2^{Ey} \cdot M_y$$

其中 E_x 和 E_y 分别为数 x 和 y 的阶码，

M_x 和 M_y 为数 x 和 y 的尾数。

两浮点数进行加法和减法的运算规则是:

$$x \pm y = (M_x 2^{Ex-Ey} \pm M_y) 2^{Ey} \qquad E_x <= E_y$$

浮点数的加减法

完成浮点加减运算的操作过程大体分为：

（1） 0 操作数的检查；

（2） 比较阶码大小并完成对阶；

（3） 尾数进行加或减运算；

（4） 结果规格化。

（5） 舍入处理。

（6） 溢出处理。

浮点数的加减法

(1) 0 操作数检查

(2) 对阶

使二数阶码相同（即小数点位置对齐），这个过程叫作对阶。

- 先求两数阶码 E_x 和 E_y之差，即 $\triangle E = E_x - E_y$

  若 $\triangle E = 0$，表示 $E_x=E_y$

  若 $\triangle E > 0$, $E_x>E_y$

  若 $\triangle E < 0$, $E_x<E_y$

  通过尾数的移动来改变E_x或E_y，使其相等。

- 对阶原则

  阶码小的数向阶码大的数对齐；

  对阶过程小阶的尾数右移，每右移一位，其阶码加1（右规）。

$2^{10}*(0.11000)+2^{8}*(0.00110)$

大阶对小阶 $2^{10}*(0.11000)\text{--}\rightarrow 2^{8}*(11.000)$

$11.000+0.00110$?????????

小阶对大阶 $2^{8}*(0.00110)\text{--}\rightarrow 2^{10}*(0.00001)$

$0.00001+0.11000=0.11001$

浮点数的加减法

例：$x=2^{01}\times 0.1101$，$y=2^{11}\times(-0.1010)$， 求x+y=?

解：为便于直观了解，两数均以**补码**表示，阶码、尾数均采用双符号位。

$[x]_{补}=00\ 01$， 00.1101　　　$[y]_{补}=00\ 11$， 11.0110

$[\triangle E]_{补}=[E_x]_{补}-[E_y]_{补}=00\ 01+11\ 01=11\ 10$

$\triangle E=-2$， 表示E_x比E_y小2， 因此将x的尾数右移两位.

右移一位， 得 $[x]_{补}=00\ 10$， 00.0110

再右移一位， 得 $[x]_{补}=00\ 11$， 00.0011

至此，$\triangle E=0$， 对阶完毕.

浮点数的加法减法

(3) 尾数求和运算

尾数求和方法与定点加减法运算完全一样。

对阶完毕可得： $[x]_{补}=00\ 11,\ 00.0011$

$[y]_{补}=00\ 11,\ 11.0110$

对尾数求和：

$$
\begin{array}{r}
00.0011 \\
+\ 11.0110 \\
\hline
11.1001
\end{array}
$$

即得： $[x+y]_{补}=00\ 11,\ 11.1001$

浮点数的加减法

(4) 结果规格化

求和之后得到的数可能不是规格化了的数，为了增加有效数字的位数，提高运算精度，必须将求和的结果规格化。

①规格化的定义： $\frac{1}{2} \le |S| < 1$ （二进制）

采用原码：

正数： S=0.1 ×××…×

负数： S=1.1 ×××…×

采用双符号位的补码：

对正数： S=00. 1×××…×

对负数： S=11. 0×××…×

浮点数的加减法

规格化规则

- 运算结果产生溢出时，必须进行右归
  - 如变形补码结果出现 10.XX 或者 01.XXX
- 如运算结果出现 0.0XXX或 1.1XX 必须左归
- 左归时最低数据有效位补0
- 右归时连同符号位进位位一起右移
- 左归时，阶码作减法，右归时，阶码作加法

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

浮点数的加减法

例：两浮点数 $x=0.1101\times2^{10}$, $y=(0.1011)\times2^{01}$，求x+y。

解： $[x]_{补}=00\ 10，\ 00.1101$　　$[y]_{补}=00\ 01，\ 00.1011$

对阶： $[\triangle E]_{补}=[E_x]_{补}-[E_y]_{补}=00\ 10+11\ 11=00\ 01$

y向x对齐，将y的尾数右移一位，阶码加1。

$[y]_{补}=00\ 10，\ 00.0101$

求和：

$$\begin{array}{r} 00.1101 \\ +\ 00.0101 \\ \hline 01.0010 \end{array}$$

$[x+y]_{补}=00\ 10，\ 01.0010$

右归：运算结果两符号位不同，其绝对值大于1，右归。

$[x+y]_{补}=00\ 11，\ 00.1001$

浮点数的加减法

(5) 舍入处理

在对阶或向右规格化时，尾数要向右移位，这样，被右移的尾数的低位部分会被丢掉，从而造成一定误差,因此要进行舍入处理。

- 简单的舍入方法有两种：

① “0舍1入”法

即如果右移时被丢掉数位的最高位为0则舍去，反之则将尾数的末位加“1”。

② “恒置1”法

即只要数位被移掉，就在尾数的末位恒置“1”。从概率上来说，丢掉的0和1各为1/2。

浮点数的加减法

在IEEE754标准中,舍入处理提供了四种可选方法:

就近舍入 其实质就是通常所说的"四舍五入"。例如,尾数超出规定的23位的多余位数字是10010,多余位的值超过规定的最低有效位值的一半,故最低有效位应增1。若多余的5位　是01111,则简单的截尾即可。对多余的5位10000这种特殊情况:若最低有效位现为0,则截尾;若最低有效位现为1,则向上进一位使其变为0。

朝0舍入 即朝数轴原点方向舍入,就是简单的截尾。无论尾数是正数还是负数,截尾都使取值的绝对值比原值的绝对值小。这种方法容易导致误差积累。

朝+∞舍入 对正数来说,只要多余位不全为0则向最低有效位进1对负数来说则是简单的截尾。

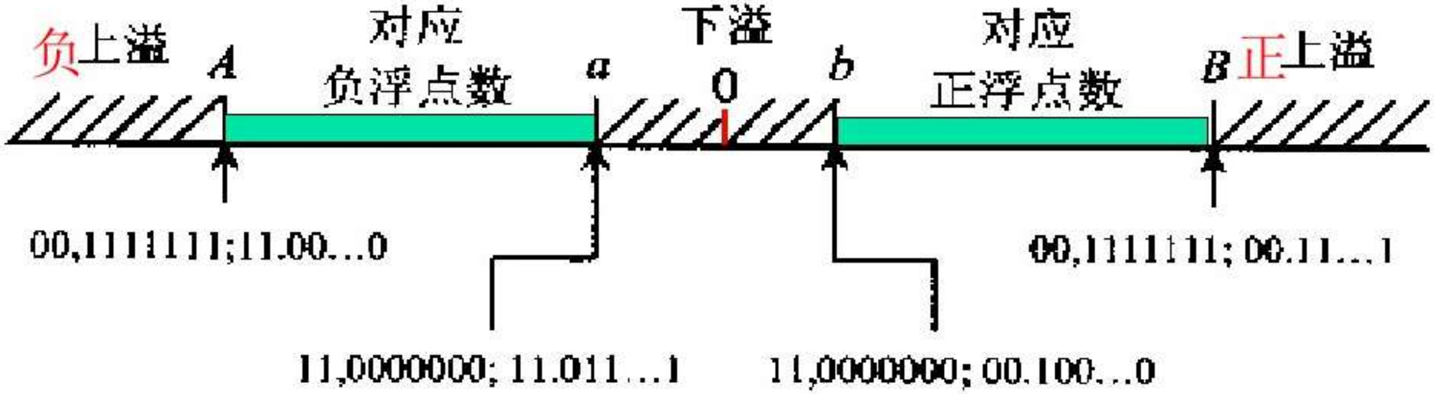
朝−∞舍入 处理方法正好与 朝+∞舍入情况相反。对正数来说,只要多余位不全为0则简单截尾;对负数来说,向最低有效位进1。

浮点数的加减法

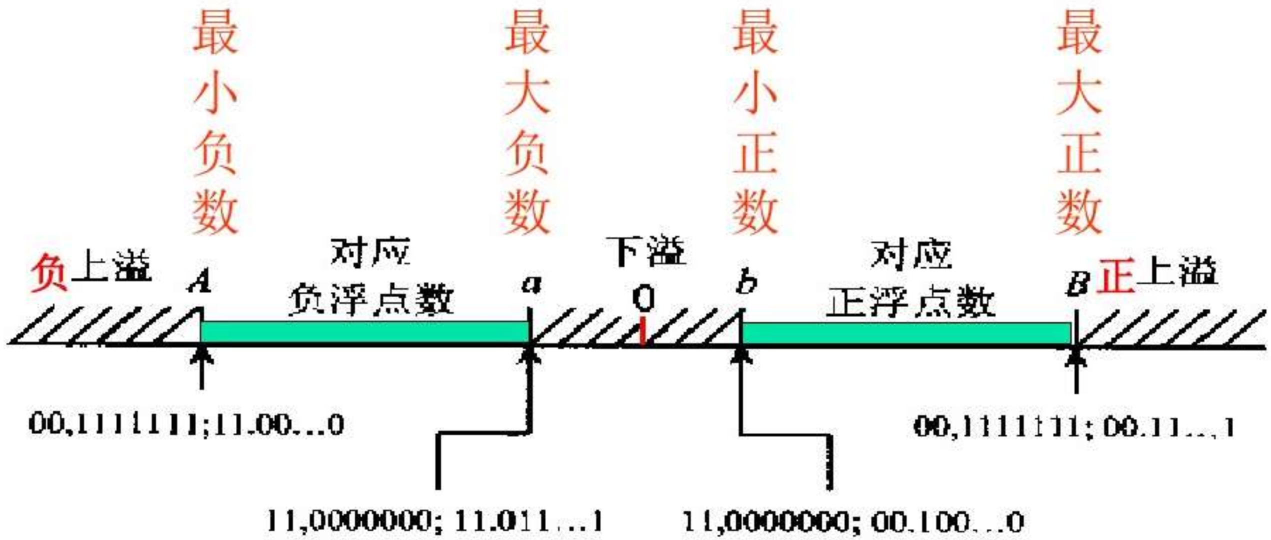
(6) 溢出处理

与定点加减法一样，浮点加减运算最后一步也需判溢出。在浮点规格化中已指出，当尾数之和(差)出现01. ××…×或10. ××…×时，并不表示溢出，只有将此数右规后，再根据阶码来判断浮点运算结果是否溢出。

若机器数为补码，尾数为规格化形式，并假设阶符取2位，阶码取7位、数符取2位，尾数取n位，则它们能表示的补码在数轴上的表示范围如图所示。



浮点数的加减法



图中A，B，a，b分别对应最小负数、最大正数、最大负数和最小正数。它们所对应的真值分别是：

A最小负数 $2^{+127}\times(-1)$

B最大正数 $2^{+127}\times(1-2^{-n})$

a最大负数 $2^{-128}\times(-2^{-1}-2^{-n})$

b最小正数 $2^{-128}\times 2^{-1}$

浮点数的加减法

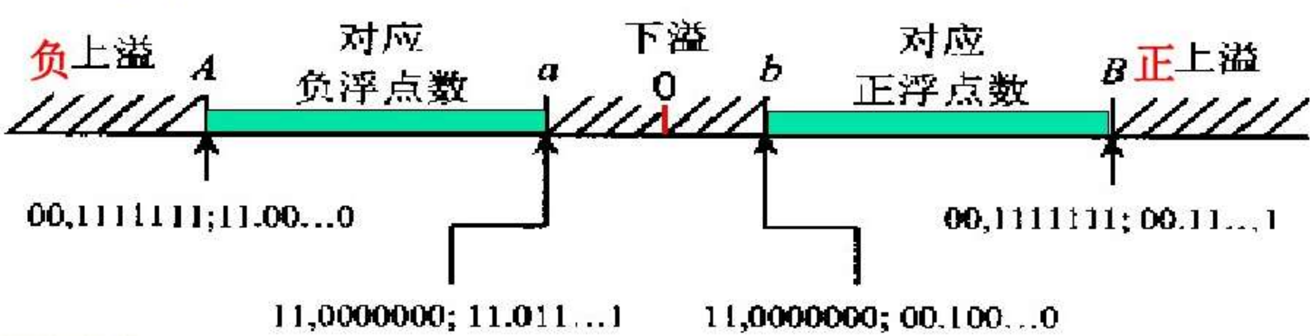
图中a,b之间的**阴影部分**，对应阶码小于128的情况，叫做浮点数的下溢。下溢时．浮点数值趋于零，故机器不做溢出处理，仅把它作为机器零。

图中的A、B两侧阴影部分，对应阶码大于127的情况，叫做浮点数的上溢。此刻，浮点数真正溢出，机器需停止运算，作溢出中断处理。**一般说浮点溢出，均是指上溢。**

可见，浮点机的溢出与否可由阶码的符号决定：

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。



浮点数的加减法

例：若某次加法操作的结果为 $[X+Y]_{补}$=00.111, 10.1011100111

则应对其进行向右规格化操作：

尾数为：11.0101110011，　阶码加1：　01.000

阶码超出了它所能表示的最大正数（+7），表明本次浮点运算产生了溢出。

例：若某次加法操作的结果为 $[X+Y]_{补}$=11.010, 00.0000110111

则应对其进行向左规格化操作：

尾数为：00.1101110000，　阶码减4：

$$\begin{array}{r} 11.010 \\ +\ 11.100 \\ \hline 10.110 \end{array} \quad [-4]_{补}$$

阶码超出了它所能表示的最小负数（-8），表明本次浮点运算产生了溢出。

浮点数的加减法

例：两浮点数 $x = 2^{101} \times 0.11011011$，$y = 2^{111} \times (-0.10101100)$。假设尾数在计算机中以补码表示，可存储10位尾数，2位符号位，阶码以补码表示，双符号位，求 $x+y$。

解：将x,y转换成浮点数据格式

$[x]_{浮}$ = 00 101, 00.11011011

$[Y]_{浮}$ = 00 111, 11.01010011+1

00 111, 11.01010100

步骤1：对阶， 阶差为Ex-Ey=[Ex]$_{补}$+[-Ey]$_{补}$

[-Ey]$_{补}$=11000＋1＝11001

Ex-Ey＝00101＋11001＝11110

＝－（00001＋1）＝－00010＝－2 < 0

Ex-Ey<0　　Ex<Ey 小阶对大阶，

X阶码加2　　X尾数右移2位

浮点数的加减法

解：将x,y转换成浮点数据格式

$[x]_{浮}$ = 00 101, 00.11011011

$[Y]_{浮}$ = 00 111, 11.01010011+1

00 111, 11.01010100

步骤1：对阶，阶差为$Ex-Ey=[Ex]_{补}+[-Ey]_{补}$

$Ex-Ey=-2<0$　　$Ex-Ey<0$　　$Ex<Ey$ 小阶对大阶，

X阶码加2　　X尾数右移2位

$[x]_{浮}$ = 00 111, 00.00110110(11)

步骤2：尾数求和

$[X+Y]_{浮}$ = 00 111, 00.00110110(11)

+ 00 111, 11.01010100

= 00 111, 11.10001010(11)

浮点数的加减法

步骤2：尾数求和

$[X+Y]_{浮}$ = 00 111, 00.00110110(11)

\+ 00 111, 11.01010100

= 00 111, 11.10001010(11)

步骤3：计算结果规格化

$[X+Y]_{浮}$ 为非规格化数，左归一位, 阶码减一，

00110, 11.00010101(1)

步骤4：舍入处理

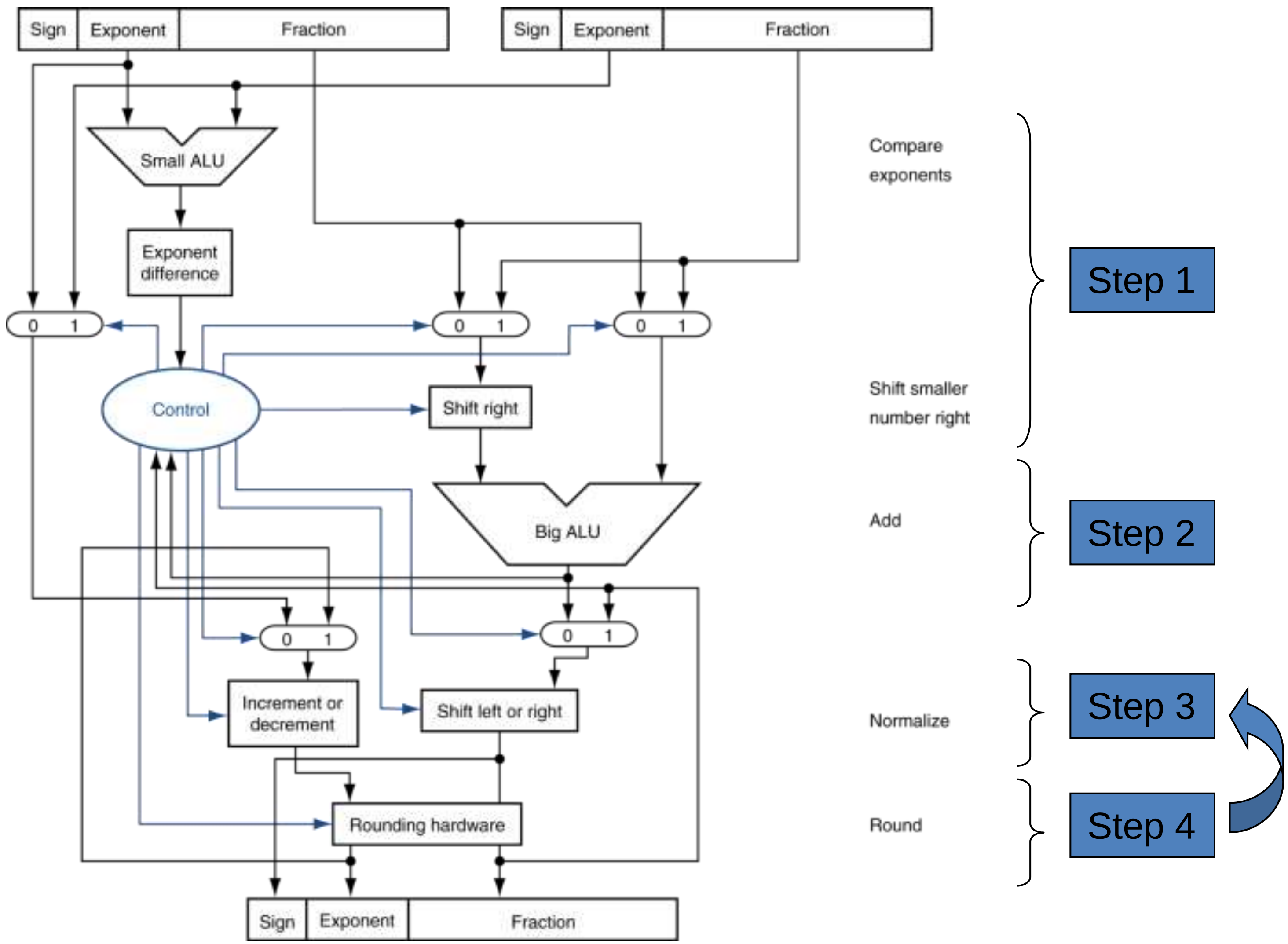
$[X+Y]_{浮}$ = 00 110, 11.00010110 (0舍1如法)

$[X+Y]_{浮}$ = 00 110, 11.00010101 (截去法)

步骤5：溢出判断

无溢出 $[X+Y]_{浮} = 2^{110} \times (-00.11101011)$

浮点数的加法硬件实现



Sign
Exponent
Fraction
Sign
Exponent
Fraction
Small ALU
Exponent difference
Control
Shift right
Big ALU
Increment or decrement
Shift left or right
Rounding hardware
Sign
Exponent
Fraction
Compare exponents
Shift smaller number right
Add
Normalize
Round
Step 1
Step 2
Step 3
Step 4

- 浮点加法器实现

32位浮点加法：floatadd.v

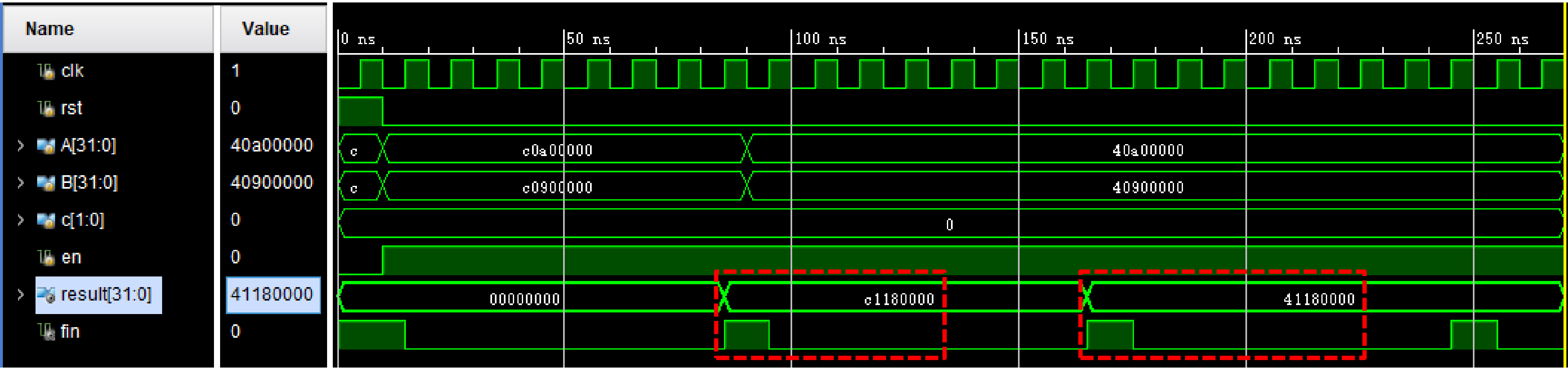
```
module float_add32(
 input   clk,
 input  rst,
 input en,                                  //开始标志
 input[31:0] A,                             //被加数
 input[31:0] B,                              //加数
output  [31:0] result,                       //商
output reg fin                              //结束标志
);
……………………
```

32位浮点加法：floatadd.v

Testbench采用固定输入激励以便观察结果

```
initial begin
rst = 1’b1;
clk = 1’b0;
en = 1’b0;
A = 32'b0;
B = 32'b0;
#10
rst = 1’b0;
en = 1;
A=32’hc0a00000; //-5.0
B=32’hc0900000; //-4.5
                //c1180000(-9.5)
#80
A=32’h40a00000; //+5.0
B=32’h40900000; //+4.5
#1000 $stop();   //41180000(+9.5)
```

32位浮点加法：floatadd.v



◉END