Verilog 最后的私私细语 ...

Verilog HDL

不同世界的自然

(第一章)

学习Verilog 最难的地方就是
无法释怀视以为常的认知
常以十进制去认识二进制
以顺序语言去思考Verilog
不过浮点数运算可以说清一切

akuei2の学习笔记

黑金动力社区

www.heijin.org

爱心广告：你 1 秒钟真的可以拯救一个人，让爱传满天下

首先声明笔者不是什么 5 元仔，这个广告是笔者的爱心喷发而已。

你知道吗？在非洲北边的某个地区，每一秒都有许许多多的人正在挨饿，每一天至少有一位儿童死于营养不足。你的一次点击就能让某位穷人得到 1.1 杯食物。当然你可以不相信有这样的链接或者是骗点击什么的。事实上这个网站确实是帮穷人得 1.1 杯食物在运行着，只要你点进去按下中间的黄色按钮，就会出来个介绍各种商品的网页（无病毒），同时也会有人因为您的一点而得到 1.1 杯食物，食物是由商家提拱的但爱心却是您献出的。如果你觉得笔者真的是 5 元仔，你不妨可以在网上查一下是真与否。

看到这本笔记的朋友多数都是读电子，或者干电子这一行的，管你穷学生还是穷工人，只要能上网，只要花出一秒种就可以了。人生在世，不然你是 LED 也好，灯泡也好，荧光灯也好，最重要就是要懂得用自身的光去照耀别人，光的强度都是其次。

最后祝福您，也祝您心想事成。

链接：
http://www.thehungersite.com/clickToGive/home.faces?siteId=1&link=ctg_ths_home_from_ths_thankyou_sitenav

目录

目录 03

第 3 章　理想时序的整合 04

1.01　闻风色变的浮点数 04
1.02　浮点数和单精度格式 08
1.03　浮点数加减法 12
1.04　实定和假定小数点的困惑.. 24
1.05　浮点数加减法实例 26
1.06　实验一：浮点数加法器.. 36
1.07　实验二：浮点数减法器 48
1.08　浮点乘法和陷阱 58
1.09　实验三：浮点数乘法器 71
1.10　浮点除法和精度流失 89
1.11　实验三：浮点数除法器 100

总结：112

第一章 不同世界的自然

在整合篇的上半部，中心内容不过与“精密控时”... 具体的经过，包括“整合概念”的基本经过一段长时间的研究，笔者终于迎来整合篇的下半部了，整合篇的上半部由于受到页数的限制，此外笔者也在上半部用光了资料 ... 说实在非常郁闷，收集资料是在编辑笔记中最耗时间和精力的一段过程，期间不仅要大量过滤和理解那些莫名奇妙的资料，而且还要不停的吸收断断续续资料，时不时会发生会消化不良或者大脑短路等现象。好了，牢骚就先发到这里 ... 我们继续 Verilog HDL 语言的学习吧。

接下来的内容，同样也是爬满了刺猬和毛毛虫，问读者有没有勇气和笔者一同走下去了？呵呵，答案是绝对的 ... 那些打开笔记的读者，在某种程度上已经有了相当的觉悟。嗯，好样！深呼吸，集中精神，放松神经，让我们继续走下去。

1.1 闻风色变的浮点数

当我们听到定点立即就会联想到定点整数和定点小数，打个比方（如代码 1.1.1 所示）：

定点整数 7_{10} = (符号位)111_2
定点小数 7.7_{10} = (符号位)111.111_2

代码 1.1.1 定点整数和小数的表示方法

定点整数一般上都简称为“整数”；定点小数一般上不常使用，除了一些特殊的场合。在 c 语言里，整数是正规的“数据格式”，相比之下那些有小数的数字都会用浮点数来表示。“浮点数是什么”估计只有蛋疼的同学才会问，学过 c 语言的朋友都知道，浮点数是（代码 1.1.2 所示）：

```
float pi;          //声明
pi = 3.142;        //赋值
if( pi == 3.142 ) //判断
pi = pi * 2.3;     //运算
```

代码 1.1.2 浮点数在 C 语言（高级语言）中的表示和操作

对呀！关于浮点数的大小姨妈就只有这些东西而已，难道还有其他吗？这个现象如同我们知道某个产生，却不知道某个经过。在高级语言的世界里，我们当然可以一只眼开一只眼关，让这些猥琐的经过都交给编译器去干，自己在旁翘脚等着收钱就是了。呵呵！高级语言和编译器多么便利呀，这也是为什么美国人的 1/3 人口都是肥胖者。

以前的笔者也是如此，今天的读者也是如此，从 C 语言过渡到 Verilog HDL 语言，不知不觉之间养成贪图效率和便利的小胖子了。直到某一天，笔者被某个 Verilog 医生检验出患有“肥胖病症”（懒惰病），笔者晴天霹雳一般，知道自己已经接近踩入不可救药的边缘了。笔者诚心祷告“掌管 Verilog 的神呀，请救救那个病入膏肓的自己吧 ...”，

那一夜，Verilog 之神在梦里对笔者说：

“孩子，在 Verilog 的世界里高级语言是 KFC、是 McDoland、是 PizzaHut、是不则不扣的快餐。‘食’用虽然方便又美味，但是它们都是毒，不但危害健康，而且还会麻痹大脑。孩子你应该放眼在 Verilog 的料理上，Verilog 料理的烹饪方法不仅煎炸一种而已，而是多样百出。此外，Verilog 料理所强调的不只是某个程度的便利和美味而已，除了色香味具备以外，还有营养均衡。”

神明永远不会告诉人们直接的含义，笔者不明白 Verilog 之神这一番对话，然后笔者焦虑的反问道：“神呀！孩子还是不明白 ...”

Verilog 之神抚摸了笔者的头，继续话到：“孩子，你是聪慧的。你既然懂得建模技巧，同样也可以明白这一番话 ... 去研究浮点数吧，孩子！记得用 Verilog 的角度去研究浮点数 ...”

就这样，笔者从梦中惊醒。笔者打从心底明白，这是 Verilog 之神给笔者的一个指引。笔者面对整数乘法器的时候曾经那样想过，笔者只要在文本编辑中输入“*”，那么乘法器——更确切说乘法的组合逻辑会经过编译器而形成，写法如下（如代码 1.1.3 所示）：

```
reg [7:0]rA,rB;
reg [15:0]rData;

always @ ( posedge CLK ... )
    ......
    rData <= rA * rB;
```

代码 1.1.3 在 Verilog 中实现最简单的乘法

在代码 1.1.3 中，rA 和 rB 均为乘法的输入，它们拥有同样的 8 位位宽。更具最简单的整数乘法规则，rData 输出的位宽为 rA 和 rB 的总和。笔者尝试的问自己，“*”是如何实现乘法操作的？那时候的笔者还是菜鸟一只，笔者翻查了许多的资料才知道，“*”乘法操作符，一是用 FPGA 设备自身的乘法硬件来实现；二是用组合逻辑以逻辑门的方法门来实现。前者消耗宝贵的乘法硬件，后者是消耗大量的组合逻辑。

那些有接触过 ASCI 设计的朋友应该知道，随着输入（操作数 a 和 b）的位宽增加，逻辑门的会使用越来越多，而且“组合逻辑的层级”会越来越长，组合逻辑的延迟也会越来越久。相反的，如果读者是用 C 语言来实现整数乘法的话（如代码 1.1.4 所示）...

```
int a,b,data;

data = a * b;
```

代码 1.1.4 C 语言的乘法实现

如同代码 1.1.4 所示那样，就是几段代码而已，至于乘法操作时如何发生，而且如何

设计（硬件的？还是软件的？）使用者是被屏蔽的猪头。在此，笔者尝试站在 Verilog 的角度继续思考：

第一个问题：“如果某个乘法的输入同样是 8 位宽，输出一定要 16 位宽吗？”
第二个问题：“Verilog 的乘法只有一种方法而已吗？（硬件还是软件）”

从中笔者明白到 Verilog 是没有固定性，Verilog 有即时事件和时间点事件之分（这就是创作时序篇的灵感）。笔者越往后想，感觉越是无底层，头脑越是发疼 ... 忽然间笔者明白到一为什么那么多人会喜欢快餐式的高级语言，它是“快餐美味方便”，很可惜 Verilog HDL 语言不是快餐。

如果 Verilog HDL 语言不是快餐的话，那么那些患有快餐上瘾症的使用者该怎么办？（根据资料显示，KFC 和 McDoland 等煎炸和高脂肪的肉类食物会使食用者上瘾）。简单呀，只要调用官方的 ip/lpm 或者消耗更多的逻辑资源，什么乘法都能实现呀。在学习的经过笔者遇到一种现象，原本调用官方 ip/lpm 是一件再普通为过的事情，和食用快食一样偶尔没什么关系，如果长时间养成习惯的话就糟糕了。

我们必须知道，ip/lpm 作为官方产品的一部分一定存在版权的问题，但是官方又不能把所有内容收的死死。一般情况下 ip/lpm 的内容都是“半开半关”，读者尝试打开其中一个 ip/lpm 开看看，读者一定会被内容吓死 ... 理由笔者不用说了，任何人都有类似的经验。所以说，如果读者打算从官方的 ip/lpm 偷学几招盖世功夫还是省了吧，看死不偿命，就算看懂了价值性也是太低 ... 这有点像快食的黑幕，我们明知快食危害健康，却无法抗拒它 ...呵呵，话题有点扯远了。

那些从时序篇过来的朋友，多少都知道要设计一个乘法器不是想象中的简单，也不是想象中的苦难。但是问题就在于，设计短整型乘法器已经够猥琐了，换作浮点数无非不是更猥琐吗？在这里，可能有同学会回答：“用 ip!”... 如果真的觉得这是最好决定，那么就这样作吧！随后可以把这本笔记关闭。

看来，我们已经接近这个章节的重点了，我们先看几段代码（代码 1.1.5）：

```
1.  float FloatMulti( float A, float B ){ ... }; //函数声明和定义
2.
3.  main()
4.  {
5.      float pi;    //浮点数变量声明
6.      float Temp;
7.
8.      pi = 3.142;                      //赋值
9.      Temp = pi * pi;                  //直接运行浮点数的乘法操作
10.     Temp = FloatMulti( pi , pi ); //调用函数实现浮点数的乘法操作
11. }
```

代码 1.1.5 C 语言中实现浮点数运算

代码 1.1.5 是一段很简单的 C 语言代码，第 9 行是直接使用“*”操作符实现乘法的运算，换之第 10 行我们调用函数实现乘法元算。第 8~9 行的运算结果基本上都是一模一样。当我们点击“编译”的一瞬间，一些不为人知的肮脏工作在台面下发生着。我们这些使用者就像笨蛋一样，浮点数是如何执行和运算我们亚丁儿也不知道。

有些同学估计会反驳道：“C 语言就是这样使用的啦！那里有问题？”正确！原本 C 语言的发明就是为了实现代码的可移植性，使用者的工作就是驾驭语言。当然，这些事情都是发生在高级语言的世界里。换作 Verilog HDL 语言，难道我们又妄想编译器帮我们实现一切吗？在扫盲文里，笔者已经说过 Verilog HDL 语言在层次上是低于 C 语言（高级语言），不过这也是大众的一般认识而已。

笔者还说过，Verilog HDL 语言是非常自由的语言，甚至是自由到没有结构，相比 C 语言（高级语言）从出生以来就有固定的结构。如果 Verilog HDL 语言没有自定义的结构，必然是依赖编译器和 ip。反之，如果 Verilog HDL 语言有自定义的结构，我们就可以按着自定义的结构来设计专属的 ip。

继续把问题往后推的话，要实现浮点数的加减乘除运算——这个问题一定会把小屁孩吓得失禁。参考书？不要笑坏笔者的肚子，参考书都有影子没实体；网页？维基百科？博客？这些来源是笔者视为最好的参考资料，但是大多数的内容都是断断续续，内容有限；权威手册？怒笔者会消化不良，不过一些关键的内容还可以看懂。

上述的内容就是笔者要面对的问题 ... 更糟的是，它们都和 Verilog HDL 语言一点关系也没有。如果笔者没有建模技巧，如果笔者更不懂 Verilog HDL 的理想时序，笔者只能说是雪上加霜而已。浮点数不是妖怪，它之所以让人闻风色变是背后的不知，不足和不安。不懂得 Verilog 世界的自然规则是不知（Verilog 的建模技巧和理想时序）；没有足够的资料是不足。最直接的问题还是自身的心境，起伏不定的心境容易产生不安。

读者明白了吧，这就是在 Verilog 的世界里，所看到的“自然现象”。高级语言和 Verilog 的世界各自都有自身的自然规则，反自然者死，自由顺从则小苗亦然茁壮成长，流水也会无孔不穿。当我们在不同世界里互相“呼唤信息”的时候，一定要非常谨慎，不要忘了自己是在那个世界去窥视那个世界，绝对不能打破自身存在在那个世界的自然规则。Verilog 之神为什么把笔者指引到这种窘境？笔者不懂，但是笔者相信神明的指明。

1.2 浮点数和单精度格式

在 1.1 小节里笔者说了一大堆废话的目的只有一个，那就是让读者有思想准备。因为任何人一旦听到“浮点数”这 3 个字，就会莫名的焦虑起来，笔者也是如此。首先我们先了解一个简单的问题：“什么是浮点数？”这个问题看似简单，实际上隐藏着许多信息。

```
float pi;
pi = 3.142;
```

代码 1.2.1 一个实例

如代码 2.1.1 所示，我们先声明一个浮点数的变量 pi，然后将 3.142 赋值给 pi。在此，我们会产生一个错觉，不知不觉的情况下我们会把 3.142 认为浮点数。嗯，这是浮点数最普遍的认知错误。我们还是从数学的角度去了解“什么是浮点数”。

3.142 是以 10 进制表示“拥有小数”的数字：3 是整数；.142 是小数。如果换成浮点数的话：

Number x B^{Exp}（ B:Base 进制 Exp:Exponent 价码）

1. 3.142×10^{0}
2. 0.3142×10^{1}
3. 31.42×10^{-1}

普通的浮点数，数字一定会伴随价码(Exponent),如第一行的 3.142×10^{0}，“3.142”就是数字，“10^{0}”就是价码。随着小数点左移/右移，价码会递增或者递减，无论小数怎么移动“数字的基本值”都不会改变。在计算机里浮点数有特定的表示方式：

```
3.142E0
0.3142E+1 / 0.3142E1
31.42E-1
```

在前面笔者已经说过了，既然我们现在是站在 Verilog 的世界去理解浮点数。所以说，那一套用在高级语言世界里的“自然规则”，当然不能套用在 Verilog 的世界里。假设我们要为某个寄存器赋予浮点数的值，如代码 1.2.2 所示：

```
1.  reg [31:0]rPi
2.  rPi <= 32'b0_10000000_10010010001011010000111
3.  rPi <= 3.142;
```

代码 1.2.2 Verilog HDL 语言里浮点数的表示方法。

第 1 行是 32 寄存器的声明；第 2 行是 3.142 按 ieee 单精度格式赋值；反之第 3 行如同高级语言那样赋值。第 2 行的赋值方法明显是遵守 Verilog 的“自然规则”，至于第 3 行是打破自然规则和强硬的赋值方法。第 3 行的方法在 Quartus II 版本 9.0c 以上（估

计版本 7.0 以上已经可以了）已经支持了，但是唯一不足的是编译器会按 ieee 双精度的规定格式去赋值。

什么是 ieee 的单精度和双精度格式，估计那些学过 C 语言的同学都略懂一二，为了使内容更全面一点，笔者还是简单的介绍一下。IEEE 对于浮点数的规定格式如表 1.2.1 所示。（笔者忘了是哪一个 IEEE 标准）:

表 1.2.1 单精度和双精度浮点数的格式。

浮点数	符号 Sign	价码 Exponent	尾数 Mantissa
单精度 32 位宽	[31]-1 位	[30..23]-8 位	[22..0]-23 位
双精度 64 位宽	[63]-1 位	[62..52]-11 位	[51..0]-52 位

看到这里，估计有些同学已经慌了，已经开始不晓得什么在打什么。事实上，理解单精度和双精度是大一的功课，尤其是“计算机组成原理”哪一环里。不知道这些同学是如何“睡过”大一或者及格考试？笔者真的很好奇 ... 笔者举个简单的例子，就以 3.142 为例：

```
3.142 转换为单精度：

3₁₀ = 0b11
.142₁₀      = 0b001001000101101000011100（实际上是非常的长，因此笔者只摘取前24位）

定点小数的表示方法为  = 0b11.001001000101101000011100      （小数源）
浮点数的表示方法为    = 0b11.001001000101101000011100E+0
                     = 0b1.1001001000101101000011100E+1   （浮点数源）
```

0.142*2=0.284----0（取个位）
0.284*2=0.568----0
0.568*2=1.136----1
0.136*2=0.272----0
0.272*2=0.544----0
0.544*2=1.088----1
...

到目前为止“0b1.1001001000101101000011100E+1”可以称为 2 进制的“浮点数源”，虽然我们已经得到“浮点数源”，但这并不表示我们的工作已经结束。我们还要把“浮点数源”套在单精度的格式里。在此之前，我们先看数学是如何表示“浮点数”。

```
浮点数     = (Sign x -1) Mantissa^Exponent

（中文）   = (符号位 x -1) 尾数^价码

3.142 = 0b (0 x -1) 1.1001001000101101000011100^1
           = 0b1.1001001000101101000011100^1

计算机     = 0b1.1001001000101101000011100E+1
```

SGL = (-1)^SIGN * 1.MANTISSA * 2^(EXPONENT-127)
单精度数=(-1)^符号位 * 1.尾数 * 2^(价码-127)
//2的幂通过小数点的移动实现

单精度格式的变量还没有赋值之前，它有固定的“默认空壳”。

```
[ 0][0111111][00000000000000000000000]   //默认空壳
[31][-30:23-][---------22:0----------]
[符][-价码--][---------尾数----------]
```

这个“默认空壳”除了符号位，和尾数以外，最让人匪夷所思的就是价码。价码的默认值不是 0 而是 0b01111111，亦即 8'd127。假设我们要把 3.142 的“浮点数源”套入这个空壳里边，让我们瞧瞧在套入的过程中会发生什么事？

```
浮点数源 = 0b1.10010010001011010000011100E+1

默认空壳的符号位[31]       = 0
默认空壳的价码[30..23]     = 0b01111111 + (E+1)
                           = 0b01111111 + 1
                           = 0b10000000
默认空壳的尾数[22..0]      = .[10010010001011010000111]00  //小数点后的 23 位

因此 3.142 的单精度表示是 = 32'b0_10000000_10010010001011010000111
```

看到了吧，过程就是这样简单而已 ...“呜呜呜呜”读者别发出这样的声音，笔者知道读者在忧虑什么。读者是不是想问：“为什么空壳的价码是 0b0111111？”和“为什么空壳的尾数只取浮点数源小数点后的 23 位？”。嗯~这是说来话长的故事，首先笔者不是单精度的发明人，当然不知道作者葫芦里卖什么药？

但是从使用者的角度来说，空壳的价码之所以是 0b0111111 是为了简化计算。至于空壳的尾数取“浮点数源”小数点后的 23 位，作者自身称为“隐藏位技术”。因为，任何的“浮点数源”都是“1.**1110**E-10 或者 -1.**0111**E+3”等格式。无论浮点数源是负值还是正直结果都是 1.XXXXE±X。因此小数点前的 0b1 可见是一个常量，在使用者的眼里它的存在是不会被动摇，没有表示出来也不用紧。

说几句讽刺的话，任何“XX”只要在后面加上“技术”二字，“XX”无论是什么简单的原理，加上“技术”二字后，发音都会无比响量。我们常常在文章里都会看见“什么什么产品佩戴 XX 技术”或者“什么什么公司有 XX 技术”... 当我们看到“XX 技术”，在心底里就会无比崇拜和敬畏。哎~现在的人类真是可悲，原因就是我们不知道“XX”到底是什么，才会觉得“XX”很厉害，很伟大。佛家说过一句“无知就是罪过”，笔者深深感受得到。

笔者再举个例子，假设笔者想把 -0.2012 用单精度表示：

```
先取得 -0.2012 的二进制表示：
-0.2012₁₀ = 0b-0.001100111000000111010111110（尾数太长了，仅摘取前小数点后的 27 位）

把 0b-0.001100111000000111010111110 换成浮点数源：
0b-0.001100111000000111010111110 = 0b-1.100111000000111010111110E-3

把浮点数源套入单精度的空壳里：
默认空壳的符号位[31] = 1 (浮点数源为负值)
默认空壳的价码[30:24]      = 0b01111111 + (E-3)
```

```
                    = 0b0111111 - 3
                    = 0b01111100
默认空壳的尾数[22:0]        = .[10011100000011101011111]0 （摘取浮点数源小数点后的23位）

-0.2012 的当精度表示 = 32’b1_01111100_10011100000011101011111
```

好了，单精度的浮点数表示方法就是那么简单而已，笔者总结一下大概的步骤：

（一）把十进制的数字转换为二进制。
（二）把二进制的数字转换为浮点数源。
（三）根据浮点数源的情况，把相关的内容套入单精度-默认空壳的相关地方。

在这里，有些同学可能会上面的任务太过于巨大，如同拯救地球那么困难。确实如此，当我们使用高级语言的时候（如 C 语言），我们都会过度依赖编译器。还记得笔者之前说过的“快食上瘾症”吗？人类对于自身认为困难的东西，潜意识中会累积大量的焦虑。身体智能为了释放这些焦虑，身体智能会影响大脑智能，然后大脑智能会以“纵欲”的方式去释放焦虑，过度的纵欲最后会变成上瘾。快食上瘾症，就是人们为了释放心理焦虑，然后以满足食欲的方式去执行。基本上素食很难上瘾，唯有肉食，尤其是油腻腻的肉食食物 ...

不知不觉又离题了！笔者所要告知读者的是 ... 在 Verilog 的世界里，过度依赖编译器就是一种“上瘾症”的现象。打从那一天我们决定和 Verilog HDL 语言打交道的时候，我们就必须遵守 Verilog 世界的“自然规则”，所以说和二进制和谐相处是注定的事情。如果读者使用编译器，读者就看不到任何和二进制有关的结果。

为了融合在 Verilog 的自然里，我们要常常与二进制相处，相处久了自然而然会发现二进制的可爱之处。当然笔者非要读者在白纸和铅笔上完成转换工作，笔者建议可以使用一些计算工具，来简化转换工作的困难度。就当给自己一个训练，要征服浮点数之前就要先征服二进制。当二进制的存在不再碍眼的时候，再使用一些快捷的工具也不为迟。

```
[ 0][0111111111][0000000000000000000000000000000000000000000000000000]
[63][--62:52---][-------------------------51:0-----------------------]
[符][---价码---][-------------------------尾数-----------------------]
```

双精度的默认空壳

至于双精度和单精度之间是大同小异，双精度比起单精度“精度的表示更为准确”和“数目表示的范围更广阔”以外，基本上任何转换步骤都一样。好了，笔者已经把这小节的重点都已经讨论一番，在此双精度不是这小节的主角，有关单精度读者只要先有个概念就行了，往后会继续深入的。

美食一不到三过多就会厌腻，学习也是一样，重点都是点到即止，不然会学到头昏脑胀。当时机到了，该遇的问题自会相遇，该懂的内容自然会懂。

1.3 浮点数加减法

单精度浮点数的加减法比起当单精度浮点数的乘除法来得更复杂一点。其中涉及到诸多的补码和正码之间的转换工作。我们先简单的回忆一下什么是正码？什么是补码？假设我们有 4 位宽的整数，为了把它区分正与负，我们会在最高位用“符号（Sign）”来表示，如下所示：

$0111_2 = 7_{10}$ // 正码
1000_2 // 取反，反码
$1001_2 = -7_{10}$ // 加 1，补码

读者是否还记得，在我们设计各种乘法器的时候，我们知道有的乘法器可融入补码和正码一起运算，而有些乘法器仅能正码和正码之间一起运算而已。对于单精度表示的浮点数来说，在表示的时候“只显示正码”；加减运算的时候则“正码和补码一起参与”。这是实现补码加减运算的第一问题点。

至于第二个问题点是关于单精度浮点数的运算过程和数学上的浮点数运算过程是非常不同，这句话又何解呢？别急，往后读者就会知晓的。最后一个问题点就是精度的控制，如常常受忽略的 10 进度和 2 进制之间“精度认识”的不同就是最好的一个例子。

```
A = 32'b0_01110100_11101101110000111001010
B = 32'b0_01110100_11101101110000111001001
A == B == 0.0009417
```

在 2 进制的认识上 A 和 B 是不一样的结果，反之在 10 进制的认识上 A 和 B 的结果均为 0.0009417。为什么会发生这样的情形呢？原因很简单，因为单精度的尾数受到位宽限制的关系。除此之外，在浮点数加减乘除的运算过程中“尾数常常会被移动”，很多软件运算会把“失去精度”作为提升运算速度/节省空间的条件。

基本上要注意的内容差不多就是这些问题而已，Verilog HDL 语言在“位操作”上拥有很强的优势，因此在设计浮点数运算的时候，比起其他高级语言“伸缩性”会更好。

Q1. A = 6.751, B = 4.832, A + B = ?

6.751 转换为单精度表示的浮点数：

6.751_{10} = 0b110.110000000010000001100010010011011 1（尾数太长，只摘取一部分）
浮点数源 = 1.10110000000010000001100010010011011 1E+2

默认空壳的符号位[31] = 0
默认空壳的价码[30:23] = 0b0111111 + 2_{10} = 0b0b10000001
默认空壳的尾数[22:0] =.[10110000000010000001100001]00100110111

```
单精度表示的浮点数是 = 32’b0_10000001_10110000000100000110001
4.832 转换为单精度表示的浮点数：

4.832₁₀ = 0b100.01100000111001010110000000100000011 （尾数太长，只摘取一部分）
浮点数源 = 0b1.0001100000111001010110000000100000011E+2

默认空壳的符号位[31]   = 0
默认空壳的价码[30:23]  = 0b0111111 + 2₁₀ = 0b0b10000001
默认空壳的尾数[22:0]   = .[00011000001110010101100 0]00010000011

单精度表示的浮点数是 = 32’b0_10000001_00011000001110010101100 0
```

在这里，我们已经得到 A 和 B 的单精度格式的浮点数，接下来我们要实现加法了。

步骤 1：操作数预处理

“操作数预处理”是任何一个单精度浮点数实现运算之前的热身运动。根据运算目的（加减还是乘除）操作数预处理的动作也不同。事实上，不同的参考书也有不同的内容，但是众多的参考书都是以“优化”为目的，作为理解是非常不适合的。所以读者还是按笔者的方法来学习吧。

隐藏位，2位

什么意思？
23位

```
reg [56:0]rA, rB;

rA <= { A[31], A[30:23], 2’b01, A[22:0], 23’d0 };
rB <= { B[31], B[30:23], 2’b01, B[22:0], 23’d0 };
```

代码 1.3.1 单精度浮点数，加减运算前的预操作。

首先声明位宽为 57 的 rA 和 rB 寄存器，rA 寄存单精度浮点数 A 的预操作结果；rB 则寄存单精度浮点数 B 的预操作结果。

rA 和 rB 各个位定义如表 1.3.1 所示：

符号位[56]	价码[55:48]	隐藏位[47:46]	尾数[45:23]	尾数补偿[22:0]
A[31]	A[30:23]	2'b01	A[22:0]	23'd0

表 1.3.1 寄存器 rA，rB 的相关位定义。

在这里，可能有同学要问：“隐藏位为什么要用 2 位来表示呢？”答案很简单，是用来寄存隐藏位和隐藏位的进位结果，此外也为了简化结果调整的工作（步骤 6）。“那么 23 位宽的尾数补偿是用来干嘛？”在前面笔者已经说过了，在运算的过程中尾数常常会发生移动，尾数补偿就是予防尾数右移的时候浮点数失去原来的精度。

结果 rA 和 rB 的值为：

```
rA = 57'b0_10000001_01_10110000000100000110001_00000000000000000000000
rB = 57'b0_10000001_01_00011000011100101011000_00000000000000000000000
```

步骤 2：价码对齐 即将价码（指数）统一，变为一样的，向较大的数统一

浮点数在实现加减之前，两个浮点数的价码之间需要对齐，对齐的规则是“向大对齐”，这样说有点模糊，笔者还是举个数学上的例子吧：

$A = 5.237 \times 10^{3}$ ， $B = 3.142 \times 10^{-2}$，A+B = ?

A.Exp - B.Exp = 3 - (-2) = 5 //A 的价码比 B 的价码大 5 个位数

$B = 3.142 \times 10^{-2}$ // B 向 A 对齐之前，B 的尾数必须右移 5 位。
$= .3142 \times 10^{-1}$ // 右移 1 位
$= .03142 \times 10^{-0}$ // 右移 2 位
$= .003142 \times 10^{1}$ // 右移 3 位
$= .0003142 \times 10^{2}$ // 右移 4 位
$= .00003142 \times 10^{3}$ // 右移 5 位

$B' = 0.00003142 \times 10^{3}$ // B' 位 B 向 A 对齐之后的结果。

$A + B' = 5.237 \times 10^{3} + 0.00003142 \times 10^{3} = 5.23703142 \times 10^{3}$

上述的例子是浮点数以 10 进制在数学的运算过程，对于 2 进制的单精度浮点数来说，过程会比较直接一点。接下来，我们看如何用 Verilog HDL 语言来实现。

首先先计算出 A 与 B 的代码差，如代码 1.3.2 所示：

```
reg [9:0]rExp,rExpDiff;

begin
    rExp = A[30:23] - B[30:23];
    if( rExp[8] ) rExpDiff <= ~rExp + 1'b1;   应该是补码
    else rExpDiff <= rExp;
end
```

代码 1.3.2 计算出 A 与 B 的价码差。

寄存器 rExp 和 rExpDiff 均为 10 位，rExp 寄存 A 与 B 的价码差；rExpDiff 则寄存相差位数。寄存器 rExp 之所以要声明为 10 位位宽，是为了判断价码差的符号位。如果 rExp[8]为 1 的话，说明价码差位负值，换句话说 A 的价码小于 B；如果 rExp[8]为 0，说明 A 的价码大于还是等于 B。总结一下：

```
if rExp[8] == 1 then A.Exp < B.Exp;
```

```
if rExp[8] == 0 then A.Exp >= B.Exp;
```

稍微把 A 与 B 的价码差计算一下：

```
rExp   = A[30:23] - B[30:23]
       = 8'b10000001 - 8'b10000001
        = 10'b00_00000000

rExp 的结果为 0 表示 A 和 B 的价码相同。
rExpDiff 为 0。
```

最后，我们根据 rExp[8] 的符号位，将 rExp 的结果赋值与 rExpDiff。rExpDiff 是用来寄存价码差的结果，为了实现移位，理应取得正值。

尾数

统一价码

```
begin
  if( rExp[8] == 1 ) begin rA[47:0] <= rA[47:0] >> rExpDiff; rA[55:48] <= rB[55:48]; end
  else begin rB[47:0] <= rB[47:0] >> rExpDiff; rB[55:48] <= rA[55:48]; end
end
```

代码 1.3.3 价码对齐操作。

代码 1.3.3 是价码对齐的动作，其中 if(rExp[8] == 1) 表示 A 的价码小于 B 的价码，所以 A 必须向 B 对齐，rA 的尾数（包括补偿尾数）即 rA[47..0] 向右移 rExpDiff 位。然后 rA 的价码即，rA[55:48]赋予 rB 的价码。反之 B 必须向 A 对齐，rB 包括补偿尾数移动 rExpDiff 位（rB[47:0] >> rExpDiff），然后 rB 的价码赋予 rA 的价码（rB[55:48] << rA[55:48]）。

在 Q1 中， A 和 B 的价码差位 0，rA 和 rB 既没有变化：

```
rA = 57'b0_10000001_01_10110000000100000110001_000000000000000000000000
rB = 57'b0_10000001_01_00011000011100101011000_000000000000000000000000
```

步骤 3：尾数（包括尾数补偿）运算预处理

什么是运算预处理？未执行加减运算之前，我们还要根据 rA 和 rB 的符号位，把 rA 和 rB 的尾数（包括尾数补偿）转换成补码与否。具体的过程到底是怎么一会事呢？我们来看代码：

运算操作过程中使用的是补码，而之前得到的都是正码

```
reg [48:0]TempA,TempB;

begin
     TempA <= rA[56] ? { rA[56], (~rA[47:0] + 1'b1) } : { rA[56], rA[47:0] };
     TempB <= rB[56] ? { rB[56], (~rB[47:0] + 1'b1) } : { rB[56], rB[47:0] };
end
```

代码 1.3.4 尾数运算前处理。

如代码 1.3.4 所示，TempA 寄存 rA 的尾数（包括隐藏位和尾数补偿），TempB 寄存 rB 的尾数（包括隐藏位和尾数补偿）。TempA 会根据 rA 的符号位，把 rA 的尾数（包括隐藏位和尾数补偿）转换为补码与否。Temp 也有同样的动作。那么 Q1 中的 A 和 B 经过步骤 3 会变成怎么样呢？

```
rA = 57'b0_10000001_01_10110000000100000110001_00000000000000000000000
rB = 57'b0_10000001_01_00011000011100101011000_00000000000000000000000

rA[56] == 0
rB[56] == 0

TempA =49'b0_01_10110000000100000110001_00000000000000000000000
TempB =49'b0_01_00011000011100101011000_00000000000000000000000
```

步骤 4：加减法操作

```
reg [48:0]Temp

begin Temp <= TempA + TempB;   end
```

包括价码哦

代码 1.3.5 加减法操作

代码 1.3.5 是加减法操作，步骤 3 和步骤 4 的存在是为了提高代码的表达能力和修改能力（还记得 IIC 的实验吗？不同速率之间的步骤互换也不会影响实验结果？）。因此，加法和减法之间的操作只是更换其中的运算符而已。

继续计算 Q1 中 A 和 B 的结果：

```
TempA =49'b0_01_10110000000100000110001_00000000000000000000000
TempB =49'b0_01_00011000011100101011000_00000000000000000000000

TempA + TempB = ?

s hh m  // S-Sign   H-Hidden Bit   M-Mantissa
0_01_1011000000010000011000100000000000000000000000
0_01_0001100001110010101100000000000000000000000000
0_10_1100100010000011000100100000000000000000000000

Temp = 49'b0_10_1100100010000011000100100000000000000000000000
```

读者记住注释中的 s h m 一行字，s 表示符号位；一个 h 表示一个隐藏位，两个 h 则是两位隐藏位；m 表示尾数（包括尾数补偿）。这 3 个字的作用可大可小，尤其在手动运

算的时候，会提高不少表达能力。

最后 Temp 的结果为 49'b0_10_1100100010000011000100100000000000000000000000。

? ? ? ? ? ? ? ?

步骤 5：结果预处理

步骤 5 中的“结果预处理”是重要的一环，在前面笔者说过单精度的浮点数是以“正直”表示，在加减运算后如果结果取得赋值的话，我们需要正直化处理。此外还要取得结果的符号位，和更新 rExp 的结果。 负值

```
begin
        isSign <= Temp[48];
        if( Temp[48] == 1'b1) Temp <= ~Temp + 1'b1; // change M be unsigned
        rExp <= {2'b00, rA[55:48]}; // or rB[55:48] , change rExp withbe rA.Exp or rB.Exp
end             之前的价码已统一，结果的价码与之相同
```

代码 1.3.6 结果预处理。

代码 1.3.6 很简单而已，isSign 赋予 Temp[48]，Temp[48]则是加减结果的符号位。然后判断 Temp[48]的符号位，如果为 1 的就正直化结果。最后更新 rExp 为 rA 的价码 (rB 的价码也是等价)。我们继续运算 Q1 的问题看看：

```
Temp = 49'b0_10_1100100010000011000100100000000000000000000000

Temp[48] == 0 then
isSign = 0;
Temp 没有变化。
rExp = 10'b00_10000001
```

步骤 6: 结果调整

单精度格式的浮点数，尤其是尾数，在运行过后结果需要经过调整。那么调整是什么？我们又要怎样调整？嗯，这是一个号问题，我们先看一个简单的例子：

```
0b 01.111111E+2   //第一组尾数
0b 10.111111E+2   //第二组尾数
0b 11.111111E+2   //第三组尾数
0b 00.111111E+2   //第四组尾数
0b 00.011111E+2   //第五组尾数
```

假设我们有 4 组尾数，除了第一组尾数以外，其余 4 组尾数都不适合“浮点数源”的格式，我们知道“浮点数源”的格式都是 0b1.XXXX，所谓的调整就是将结果调整到适合“浮点数源”的格式。

那么其余 3 组尾数的调整过程会是：

```
0b 10.111111E+2 = 0b 1.0111111E+3  // 第二组 尾数右移 1 位，价码加 1
0b 11.111111E+2 = 0b 1.1111111E+3  // 第三组 尾数右移 1 位，价码加 1
0b 00.111111E+2 = 0b 1.11111  E+1  // 第四组 尾数左移 1 位，价码减 1
0b 00.011111E+2 = 0b 1.1111   E+0  // 第五组 尾数左移 2 位，价码减 2
```

比起这个简单的例子，在实际的调整中我们会遇见更多位宽和更大范围的调整。

```
begin
    if( Temp[47:46] == 2'b10 || Temp[47:46] == 2'b11) begin Temp <= Temp >> 1; rExp <= rExp + 1'b1; end
    else if( Temp[47:46] == 2'b00 && Temp[45] ) begin Temp <= Temp << 1; rExp <= rExp - 5'd1; end
    else if( Temp[47:46] == 2'b00 && Temp[44] ) begin Temp <= Temp << 2; rExp <= rExp - 5'd2; end
    else if( Temp[47:46] == 2'b00 && Temp[43] ) begin Temp <= Temp << 3; rExp <= rExp - 5'd3; end
    else if( Temp[47:46] == 2'b00 && Temp[42] ) begin Temp <= Temp << 4; rExp <= rExp - 5'd4; end
    else if( Temp[47:46] == 2'b00 && Temp[41] ) begin Temp <= Temp << 5; rExp <= rExp - 5'd5; end
    else if( Temp[47:46] == 2'b00 && Temp[40] ) begin Temp <= Temp << 6; rExp <= rExp - 5'd6; end
    else if( Temp[47:46] == 2'b00 && Temp[39] ) begin Temp <= Temp << 7; rExp <= rExp - 5'd7; end
    else if( Temp[47:46] == 2'b00 && Temp[38] ) begin Temp <= Temp << 8; rExp <= rExp - 5'd8; end
    else if( Temp[47:46] == 2'b00 && Temp[37] ) begin Temp <= Temp << 9; rExp <= rExp - 5'd9; end
    else if( Temp[47:46] == 2'b00 && Temp[36] ) begin Temp <= Temp << 10; rExp <= rExp - 5'd10; end
    else if( Temp[47:46] == 2'b00 && Temp[35] ) begin Temp <= Temp << 11; rExp <= rExp - 5'd11; end
    else if( Temp[47:46] == 2'b00 && Temp[34] ) begin Temp <= Temp << 12; rExp <= rExp - 5'd12; end
    else if( Temp[47:46] == 2'b00 && Temp[33] ) begin Temp <= Temp << 13; rExp <= rExp - 5'd13; end
    else if( Temp[47:46] == 2'b00 && Temp[32] ) begin Temp <= Temp << 14; rExp <= rExp - 5'd14; end
    else if( Temp[47:46] == 2'b00 && Temp[31] ) begin Temp <= Temp << 15; rExp <= rExp - 5'd15; end
    else if( Temp[47:46] == 2'b00 && Temp[30] ) begin Temp <= Temp << 16; rExp <= rExp - 5'd16; end
    else if( Temp[47:46] == 2'b00 && Temp[29] ) begin Temp <= Temp << 17; rExp <= rExp - 5'd17; end
    else if( Temp[47:46] == 2'b00 && Temp[28] ) begin Temp <= Temp << 18; rExp <= rExp - 5'd18; end
    else if( Temp[47:46] == 2'b00 && Temp[27] ) begin Temp <= Temp << 19; rExp <= rExp - 5'd19; end
    else if( Temp[47:46] == 2'b00 && Temp[26] ) begin Temp <= Temp << 20; rExp <= rExp - 5'd20; end
    else if( Temp[47:46] == 2'b00 && Temp[25] ) begin Temp <= Temp << 21; rExp <= rExp - 5'd21; end
    else if( Temp[47:46] == 2'b00 && Temp[24] ) begin Temp <= Temp << 22; rExp <= rExp - 5'd22; end
    else if( Temp[47:46] == 2'b00 && Temp[23] ) begin Temp <= Temp << 23; rExp <= rExp - 5'd23; end
    // else do nothing if( Temp[47:46] == 2'b01 )
end
```

代码 1.3.7 结果调整，范围 Temp[47] ~ Temp[23]。

我们知道 Temp[48]是符号位，符号位不参与调整。因此 Temp[47:46]~Temp[45:23]之间就成为了调整的平衡点，换个角度说“浮点数源”格式的“假定”小数点就在两者之间，既是“Temp[47:46].Temp[45:23]”。（关于假定小数点的解释在下一个小节）

为了更清楚的表达，我们先假设 4 个情况：

```
  s hh m                s hh m                s hh m                s hh m
49'b0_01_1.....       49'b0_01_0.....       49'b0_00_1.....       49'b0_00_01.....
49'b0_01_1..... +     49'b0_01_0..... +     49'b0_00_1..... +     49'b0_00_01..... +
===========           ===========           ===========           ===========
49'b0_11_0......      49'b0_10_0......      49'b0_01_0......      49'b0_00_10......
```

上面的 4 个情况告诉我们一个事实，当 A 和 B 的尾数作加操作的时候，2 位位宽的隐藏位足够应付隐藏位的进位结果。在实际的操作中结果远远不会那么理想，我们需要执行更大范围的调整工作，如：

```
  s hh m
49'b0_11_xxxxxxxxxxxxxxxxxxxxxxxx x // 隐藏位为 2'b11，尾数需要右移 1 位，价码加 1
49'b0_10_xxxxxxxxxxxxxxxxxxxxxxxx x // 隐藏位为 2'b10，尾数需要右移 1 位，价码加 1

  s hh m
49'b0_00_[1]xxxxxxxxxxxxxxxxxxxxxx x // 隐藏位为 2'b00，[45]位为 1，尾数需要左移 1 位，价码减 1
49'b0_00_0[1]xxxxxxxxxxxxxxxxxxxxx x // 隐藏位为 2'b00，[44]位为 1，尾数需要左移 2 位，价码减 2
49'b0_00_00[1]xxxxxxxxxxxxxxxxxxxx x // 隐藏位为 2'b00，[43]位为 1，尾数需要左移 3 位，价码减 3
......
49'b0_00_0000000000000000000000[1]x x // 隐藏位为 2'b00，[24]位为 1，尾数需要左移 22 位，价码减 22
49'b0_00_0000000000000000000000[1] x // 隐藏位为 2'b00，[23]位为 1，尾数需要左移 23 位，价码减 23

  s  hh   m
49'b0_0[1]_xxxxxxxxxxxxxxxxxxxxxx x // 最终的结果
```

除了结果的隐藏位为 2'b01 以外，其外都需要调整：

如果结果的隐藏位为 2'b11 那么尾数需右移 1 位，然后价码加 1;
如果结果的隐藏位为 2'b10 那么尾数需右移 1 位，然后价码加 1;
如果结果的隐藏位为 2'b00，但是第 45 位为 1，那么需尾数左移 1 位，然后价码减 1;
如果结果的隐藏位为 2'b00，但是第 44 位为 1，那么需尾数左移 2 位，然后价码减 2;
如果结果的隐藏位为 2'b00，但是第 43 位为 1，那么需尾数左移 3 位，然后价码减 3;
....
如果结果的隐藏位为 2'b00，但是第 24 位为 1，那么尾数需左移 22 位，然后价码减 22;
如果结果的隐藏位为 2'b00，但是第 23 位为 1，那么尾数需左移 23 位，然后价码减 23。

经过调整以后，最终的结果都会和"浮点数源"的格式 1.xxxxx 一致。读者需要注意的是，尾数发生左移还是右移的时候价码也会随着改变。此外，可能有同学要问："为什么不调整[23]位之后的尾数呢？"这是一个好问题，一般上只调整范围只要达到尾数的第[23]位已经够用了，一些挑剔的设计可能会把尾数补偿都考虑到调整的范围内。

家下来，我们来看看 Q1 中 A 和 B 的操作结果，经过调整后会变成什么样子？

```
Temp = 49'b0_10_1100100010000011000100100000000000000000000000
```

```
rExp = 10'b00_10000001


   s hh m
49'b0_10_1100100010000011000100100000000000000000000000 // 隐藏位 2'b01, 尾数右移 1 位, 价码加 1

49'b0_10_1100100010000011000100100000000000000000000000 // 之前, rExp = 10'b00_10000001
49'b0_01_0110010001000001100010010000000000000000000000 // 右移 1 位,  rExp + 1
49'b0_01_0110010001000001100010010000000000000000000000 // 之后, rExp = 10'b00_10000010
```

步骤 7：输出和格式化

步骤 7 主要的动作就是把调整后的结果，以单精度浮点数格式化，并且判断价码溢出，或者尾数零值等问题。如果没有任何问题就把格式化以后的结果输出，否则就输出“默认空壳”和相关的错误信息。

我们先来讨论什么是价码溢出：

单精度格式的浮点数，价码的位宽为 8，然后价码从 8'b0111111 之间摆动不定。如果价码为 8'b1000000 那么该浮点数源的价码为 E+1；反之如果价码为 8'b01111110，那么该浮点数源的价码为 E-1。换句话说，正直的价码会从 8'b10000000~8b11111111, 亦即 E+1 ~ E+128；同样负值的价码从 8'b01111110 ~ 8'b00000000，亦即 E-1 ~ E-127。
（对于一些指令格式来说赋值的价码从 8'b01111110 ~ 8'b00000001 而已，其中 8'b00000000 是保留给零值使用）

价码溢出可分为两种，亦即价码上溢和价码下溢。价码上溢就是正直的价码超过 E+128，价码下溢则是负值的价码超过 E-127。价码溢出的问题，最可能发生是在尾数的调整之后，因为尾数在调整的时候，价码也会随着改变。我们为了判定价码是否溢出？上溢还是下溢？因此，我们为 rExp 声明 10 位宽寄存器。

假设 rExp[9:8]为 2'b00，即表示价码没有溢出；rExp[9:8]为 2'b01，即表示价码为上溢；rExp[9:8]为 2'b11，即表示价码为下溢。喜欢抓头的同学可能又要问：“为什么认定 rExp[9:8]等于 2'b01 就是发生价码上溢，其他亦然？”原因很简单，笔者举个简单的演示(表 1.3.2)：

上溢 = 10'b00_11111111 + 10'b00_00000001 = 10'b01_00000000	上溢 = 10'b00_11111111 + 10'b00_00000011 = 10'b01_00000010
无溢出 = 10'b00_00001111 + 10'b00_00100000 = 10'b00_00011111	
下溢 = 10'b00_00000000 - 10'b00_00000001 = 10'b00_00000000 + 10b11_111111111 补 = 10'b11_111111111	下溢 = 10'b00_00000000 - 10'b00_00000111 = 10'b00_00000000 + 10b11_11111001 补 = 10'b11_11111001

表 1.3.2 价码溢出的表示

如果价码正的发生溢出，我们该怎么办才好呢？别慌，我们只要反馈价码上溢，或者下溢的错误信息，然后把输出指定为默认空壳就可以了。

那么，什么是尾数零值呢？

尾数零值仅是一个假想的错误状态，一般上单精度的浮点数无法表示 0 值，0 值在浮点数中表示无穷大。尾数零值还有另一个可能是：当浮点数执行运算过后可能会发生尾数全零的状态，举个例子 1.5E+2 减 1.5E+2。

除此之外，当运算结果使得精度非常非常小的时候，甚至超过单精度可以曾受的范围之内，我们也可以看成是尾数零值。如果发生了尾数零值，我们该怎么办呢？传统中，处理器的指令系统会反馈 zero 或者 infinity 的错误指令。所以笔者有样学样，果真发生尾数零值，反馈尾数零值的错误信息，然后输出默认空壳。

说一点体外话！在笔者的理解中，0 值等价与“精度超过格式可承受的范围”。以此类推，在无限的左移调整中，价码迟早会发生下溢 ... 举个具体的例子：

```
0 = 0.001                                  //单精度有余承受
0 = 0.00000000000000001                    //单精度可以承受
0 = 0.000000000000000000000000001          //只要左移尾数，单精度勉强承受
0 = 0.000000000000000000000000000 xxxxx 1      //已经超过单精度可以承受的范围
```

精度恢复，四舍五入？

如果调整以后的结果，价码没有溢出或者尾数又不是零值。换言之，调整结果是单精度格式可以承受的浮点数结果。在输出之前，我们还要执行四舍五入的操作。对于拥有 10 个手指头的人类来说，四舍五入的概念很简单，换做 2 进制的话又会怎么样呢？

假设小数只要求 3 位数，那么小数第 3 位之后的数目就要执行四舍五入，举个例子：

```
22/7    = 3.142857142857142857l4
        = 3.143
```

对于 2 进制来说四舍五入等价于“0 舍 1 入”，举个例子(同要小数之后的 3 个位数而已)：

```
1.1110111 = 1.111
1.0001111 = 1.001
```

可能有些同学很好奇，为什么尾数输出之前要执行四舍五入呢？原因很简单，就是恢复精度嘛（也可是说是保护精度流失）。如果要执行四舍五入，那么这个小数点以后的“某个定位”又在哪里？单精度浮点数的尾数是 23 位，对应 Temp[45 ... 23]，“某个定位”就是 Temp[22]。用简单的话来说，如果 Temp[22]的值是 1 的话，那么 Temp[45 ... 23]就加 1，反之不然。

```
begin
    if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end          // E Overflow
    else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end   // E Underflow
    else if( Temp[46:23] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
    else if( Temp[22] == 1'b1 ) rResult <= { isSign, rExp[7:0], Temp[45:23] + 1'b1 };   // okay with normalised
    else rResult <= { isSign, rExp[7:0], Temp[45:23] }; // okay without normalise
end
```

代码 1.3.8 结果输出之前的处理。

代码 1.3.8 中有 5 个分支，前两个分支是判断价码是否溢出；第 3 个分支是判断尾数是否零值。如果任何一个分支的条件成立，就拉高相关的错误信息，然后输出默认空壳。第 4 个分支是用来检测 Temp[22]的值是否为一？如果是的话，就往 Temp[45:23]中加 1。然后按单精度的格式输出结果。最后一个分支和第 4 个分支大同小异，只是没有执行四舍五入的动作而已。

好了，我们把 Q1 的题目运算一下：

```
Temp = 49'b0_01_01100100010000011000100[1]0000000000000000000000
rExp = 10'b00_10000010
isSign = 0

Temp[22] is 1 then 23'b01100100010000011000100 + 1 ,equal 23'b01100100010000011000101
输出结果  = 32'b0_10000010_01100100010000011000101
```

我们稍微验证一下：

```
6.751 + 4.382 = 11.133

11.133₁₀      = 0b1011.0010001000001100010010011011110100
浮点数源      = 0b1.0110010001000001100010010011011110100 E+3

默认空壳[31]      = 0
默认空壳[30:23]   = 0b01111111 + 3 = 0b10000010
默认空壳[22:0]    = .[01100100010000011000100]100110111

Real Result       = 32'b0_10000010_01100100010000011000100
```

看！结果是不是已经接近了!? 很好 ... 在前面，笔者已经说过二进制浮点数和十进制浮点数之间会存在不同的“精度认识”。在二进度的认识上，两个结果不一样，但是在 10 进制的认识上，运算结果是正确的。举个例子：

```
32' b0_10000010_01100100010000011000101   Test Result 11.1330003
32' b0_10000010_01100100010000011000100   Real Result 11.1329994
```

```
True Result 11.133
```

Test Result 是两个操作数 6.741 和 4.382 执行一系列运算以后的结果；
Real Result 是预期结果 11.133 直接转换成单精度的结果；
True Result 是 10 进制中预期的结果。

在此 Test Result 和 Real Result 之间有所相差，这是著名的精度问题。但是在 10 进制的认识上，它们都是正确的。

==

最后，笔者稍微总结一下一系列的运算步骤：

（一）操作数预处理：从单精度格式中恢复成浮点数源。
（二）价码对齐：价码向大对齐，取得价码差，然后执行对齐操作。
（三）尾数运算预处理：根据操作数的符号位，处理尾数。
（四）加减操作：尾数执行加减操作。
（五）结果预处理：取得运算结果的符号位，更新 rExp 和正值化运算结果
（六）结果调整：调整运算结果，适合于浮点数源的格式。
（七）输出和格式化：检测价码溢出，尾数零值等错误，此外执行四舍五入。

基本上，单精度格式的浮点数要完成加减运算就是以上 7 个步骤而已。好了，这一小节不多就到这里了。可能读者还觉得例子不够，想吃更多，读者姑且先让大脑休息一会儿。下一个小节，笔者会用更多的实例来消除读者的迷惑。

1.4 实定和假定小数点的困惑

首先笔者插入一个不相干的小节，根据笔者学习的经验，一开始接触浮点数的时候，尤其是二进制的尾数移动那个的环节 ... 刚开始的时候感觉有点“怪怪”的。对于拥有十只脚趾头的人类来说，从小到大的数学教育都是基于 10 进制，即时长大了从事软件相关的行业也很少直接与底层的二进制打交道。

我们先来看看一个例子（例子 1.4.1 所示）：

```
0b 1.1001 E+2
0b 11.001 E+1
0b 110.01 E+0
```

例子 1.4.1 例子一

当我们理解上述的例子，大脑很习惯的看着小数点移动，作为书写的形式，我们可以在白纸或者文本编辑上表示小数点的存在，当我们移动小数点的时候，有“实际”的小数点在发生移动，在此简称“实定移动”。

相反的，我们不能使用 Verilog HDL 语言来表示小数点的存在，我们只能“假定”从某个范围，到某个范围之间的中心点就是小数点的存在，如下代码所示（例子 1.4.2 所示），笔者以 1.3 小节中出现的 Temp[48:0]为例；

```
 符号位      隐藏位      小数点       尾数        尾数补偿
Temp[48]  Temp[47:46]     .      Temp[45:23]  Temp[22:0]
```

例子 1.4.2 例子二

在代码 1.4.2 中，Temp 寄存器是用来寄存运算结果，它的位宽为 49 位。然而 Temp 本身没有牺牲其中的一位表示小数点的存在。在此“小数点”在二进制中是一个“假定”存在，它的存在是介于 Temp[47:46] 和 Temp[45:23]之间。如果小数点不存在，自然不能实现所谓的“小数点移动”，我们最多仅能左移或者右移 Temp 的二进制值，在此简称二进制的“假定位移”。

那么，这种“怪怪”的感觉到底是怎么一回事呢？我们再来看一个例子（例子 1.4.3）：

```
“实定位移”和“假定位移”发生右移的时候：

0b 1.1001 E+2 before Exp = 2
0b 11.001 E+1 小数点右移 1 位
0b 11.001 E+1 after E = 1

0b 0_11_10000000000000000000000()  before  Exp = 0
0b 0_01_11000000000000000000000(0) 二进制值右移 1 位, Exp = Exp + 1
0b 0_01_11000000000000000000000(0) after Exp = 1
```

```
“实定位移”和“假定位移”发生左移的时候：

0b 1.1001  E+2 before Exp = 2
0b 0.11001 E+3 小数点左移 1 位
0b 0.11001 E+3 after Exp = 3

0b 0_00_11100000000000000000000(1)  before  Exp = 1
0b 0_01_11000000000000000000001() 二进制值左移 1 位, Exp = Exp - 1
0b 0_01_11000000000000000000001() after Exp = 0
```

例子 1.4.3 “实定位移”和“假定位移”的区别。

例子 1.4.3 告诉了我们一个事实，“实定位移”和“假定位移”的小数点在发生移动的时候会产生“相反的结果”，这话又何解？如例子 1.4.3 所示，在右移的过程中“实定位移”会使价码递减，反之“假定位移”会使价码递增；在发生左移的时候“实定位移”会使价码递增，“假定位移”则会使价码递减。

哇！这真是一个多么可怕的发现，老人说:“疏忽小细节会毁了一切”，原来这种“怪怪”的感觉就是这么一回事。侥幸笔者福气十足，读者也是 ... 不然的话，心里深处会一直受这种“怪怪”的感觉折磨，有如一只看不见的针，不停的忘心里头扎，直到把我们逼疯。

最后，随便讽刺一下那些所谓的“权威”的参考书，如果小细节都不见得，结果会如老人所说：“疏忽小细节会毁了一切”，再权威也会胯下。

1.5 浮点数加减法实例

在还没有进入实验之前，为了尽可以消除读者对浮点数的迷惑，接下来笔者会举例大量的例子。在此之前，我们稍微复习一下单精度浮点数加减预算的几个步骤：

（一）操作数预处理就是从单精度的格式中恢复浮点数；
（二）价码对齐是浮点数在执行加减运算之前的重要步骤；
（三）尾数运算预处理是根据操作数的符号位，是否对尾数转换为补码形式；
（四）加减处理是尾数的加减运算；
（五）结果预处理，取得结果的符号位，和正值化运算结果；
（六）结果调整就是将尾数调整到适合“浮点数源”的格式。
（七）输出和格式化中除了反馈相关的错误信息以外，还有四舍五入的操作。

Q2：A = 3.65 , B = -7.4 , A + B = ?

```
A = 32'b0_10000000_11010011001100110011010
B = 32'b1_10000001_11011001100110011001101

价码对齐：
Exp - B.Exp = 0b10000000 - 0b10000001
              = 128 - 129
              = -1  （负值表示 B 的价码比 A 的价码大，所以 A 向 B 看齐）

$ A.Exp = B.Exp and A.M >> 1

h m
1 11010011001100110011010     before
0 111010011001100110011001101(0)   >>1 （括号为尾数补偿）
0 111010011001100110011001101(0)   after , A = A’

尾数运算预处理：
$ B.Sign is 1 , while ~B.m + 1（B 的符号位为 1 必须取得补码形式）

s hh m
0 01 11011001100110011001101 before
1 10 00100110011001100110010 ~
1 10 00100110011001100110011 +1
1 10 00100110011001100110011 after, B = B’

加减操作：
e          s hh m
10000001 0 00 111010011001100110011001101(0)   A’
10000001 1 10 00100110011001100110011      B’ +
```

```
10000001 1 11 0001000000000000000000000(0)    sum

结果预处理：
$ isSign = sum.s = 1
$ sum.s is 1, while ~sum.m + 1 （运算结果的符号位为 1，需要正值化）

s hh m
1 11 0001000000000000000000000(0) before
0 00 1110111111111111111111111(1) ~
0 00 1111000000000000000000000(0) +1
0 00 1111000000000000000000000(0) sum'

结果调整：
hh is 2'b00, while sum'.m << 1 and sum'.exp - 1

$ sum'.exp = 8'b10000001 - 1 = 8'b10000000

s hh m
0 00 1111000000000000000000000(0) before
0 01 1110000000000000000000000()  <<1
0 01 1110000000000000000000000    after, sum"

输出和格式化：
32'b1_10000000 11100000000000000000000 Test Tesult -3.75
32'b1_10000000 11100000000000000000000 Real Result -3.75
                                       True Result -3.75
```

上面是 Q2 简化运算（手动）的过程，读者是不是非常不习惯呢？别慌，笔者会一步一步慢慢解释的。首先，笔者直接用工具取得操作数 A 和 B 单精度表示的浮点数，笔者直接省略了转换过程。对于读者来说，只要明白 1.2~1.3 小节，那么用不着重复了（页数有限的关系）。

接下来的动作是价码对齐，如果取得的代码差是负直的话，即表示 A 小于 B。然后 A 的尾数右移一位，A 的价码赋予 B 的价码，A 重新命名为 A'。在执行加减操作之前，操作数会根据自身的符号位来决定是否把尾数转换成补码形式？很不幸，恰好操作数 B 的符号位为 1，因此 B 的尾数必须转换位补码形式。B 重新命名为 B'。

此时运行 A' + B' 的操作，得到运行结果后，将 sum 的符号位赋予 isSign（Temp[48]），然后检测 isSign 是否为 1，如果是就正值化 sum 的尾数。sum 重新命名为 sum'。随后我们必须把结果调整致适合“浮点数源”的格式，我们判断 hh 的值（隐藏位）(Temp[47:46])，取得 2'b00，此时小数点后的第一位为 1（Temp[47:46] == 2'b00 && Temp[45]），结果必须左移 1 位，和递减 sum'的价码。sum' 重新命名为 sum"。最后把结果输出。

说实在，简化运算模式习惯以后，在表达浮点数运算全过程的时候是非常方便的，如果读者还不能习惯的话，不要勉强自己 ... 慢慢来。

Q3: A = -3.142 , B = 2.002, A+B = ?

```
A = 32'b1_10000000_10010010001011010000111
B = 32'b0_10000000_00000000010000011000101

价码对齐:
A.exp - B.exp = 0b10000000 - 0b10000000 = 0

尾数运算预处理:
$ A.sign is 1 , while ~A.m + 1'b1

s hh m
0 01 10010010001011010000111 before
1 10 01101101110100101111000 ~
1 10 01101101110100101111001 +1
1 10 01101101110100101111001 after, A = A'

加减操作:
e        s hh m
10000000 1 10 01101101110100101111001 A'
10000000 0 01 00000000010000011000101 B  +
10000000 1 11 01101110000101000111110 sum

结果预处理:
$ isSign = sum.s = 1
$ sum.s is 1, while ~sum.m + 1

s hh
1 11 01101110000101000111110 before
0 00 10010001111010111000001 ~
0 00 10010001111010111000010 +1
0 00 10010001111010111000010 after, sum = sum'

结果调整:
$ sum'.hh is 2'b00, while sum'm << 1 and sum'.exp - 1

sum".exp = 8'b10000000 - 1 = 8'b01111111

hh m
00 10010001111010111000010 before
01 00100011110101110000100 <<1
```

刚开始为什么是0？

```
01 0010001111010111000000100 after, sum’ = sum″

输出和格式化：
32’b1_01111111_00100011110101110000100 Test Result -1.139998
32’b1_01111111_00100011110101110000101 Real Result -1.1399999
                                       True Result -1.14
```

Q4：A = -3.142，B = -11.519，A + B = ?

```
A = 32’b1_10000000_10010010001011010000111
B = 32’b1_10000010_01110000100110111010011

价码对齐：
A.exp - B.exp = 0b1000000 - 10000010 = -2

$ -2 while, A.m >> 2 and A.exp = B.exp

A.exp = B.exp = 0b10000010

hh m
01 10010010001011010000111(00) before
00 01100100100010110100001(11) >> 2
00 01100100100010110100001(11) after, A = A’

运算预处理：
$ A’.sign and B.sign is 1, while ~A.m + 1, ~B.m + 1

s hh
0 00 01100100100010110100001(11) before
1 11 10011011011101001011110(00) ~
1 11 10011011011101001011111(00) +1
1 11 10011011011101001011111(00) after, A’ = A″

s hh
0 01 01110000100110111010011 before
1 10 10001111011001000101100 ~
1 10 10001111011001000101101 +1
1 10 10001111011001000101101 +1 after, B = B’

加减运算:
e        s hh m
10000010 1 11 10011011011101001011111(00) A″
10000010 1 10 10001111011001000101101(00) B’ +
10000010 1 10 00101010110110010001100(00) sum
```

是不变的哦

```
结果预处理：
$ isSign = sum.s = 1
$ sum.s is 1, while ~sum.m + 1'b1

s hh m
1 10 00101010110110010001100(00) before
0 01 11010101001001101110011(11) ~
0 01 11010101001001101110100(11) +1
0 01 11010101001001101110101(00) after, sum = sum'

结果调整：
$ sum'.hh is 2'b01, while do nothing ...

输出和格式化：
32'b1_10000010_11010101001001101110101 Test Result -14.6610002
32'b1_10000010_11010101001001101110101 Real Result -14.6610002
                                        True Result -14.661
```

Q4 的重点就在于 A 和 B 操作数均为负值（符号位），在运算预处理时双方的尾数都需要取得补码的形式。在加减预算过后，尤其是结果预处理的阶段，由于运算结果是负值（符号位），所以结果的尾数必须正值化（因为单精度浮点数是以正直表示）。

Q5：A = 2112.2012，B = -3.1429，A + B = ?

```
A = 32'b0_10001010_00001000000001100111000
B = 32'b1_10000000_10010010010010101000110

价码对齐：
A.exp - B.exp = 0b10001010 - 0b10000000 = 10₁₀
$ 10 mean A.exp > B.exp, while B.m >> 10 and B.exp = A.exp

B.exp = A.exp = 0b10001010

hh M
01 10010010010010101000110()            before
00 00000000011001001001001(0101000110) >> 10
00 00000000011001001001001(0101000110) after, B = B'

运算预处理：
$ B'.sign is 1, while ~B'.m + 1

s hh M
0 00 00000000011001001001001(0101000110) before
```

```
1 11 1111111111001101101101100(1010111001) ~
1 11 1111111111001101101101100(1010111010) +1
1 11 1111111111001101101101100(1010111010) after, B' = B″

加减操作：
e        s hh m
10001010 0 01 00001000000000110011100(0000000000) A'
10001010 1 11 1111111111001101101101100(1010111010) B″ +
10001010 0 01 00000111101000011101110(1010111010) sum

结果预处理：
$ isSign = sum.s = 0

结果调整：
$ sum.hh is 2'b01, while do nothing ...

输出和格式化：
Temp[22] is 1, Temp[45:23] + 1

m
00000111101000011101110(1010111010) before
00000111101000011101111              after

32'b0_10001010_00000111101000011101111 Test Result 2109.0583496
32'b0_10001010_00000111101000011101111 Real Result 2109.0583496
                                        True Result 2109.0583
```

Q5 的重点就在于 A 和 B 操作数之间的价码差很大，因此在价码对齐的步骤里，B 尾数的一部分都填入致尾数补偿。然后，在最后的步骤——输出和格式化，由于尾数补偿最高位（Temp[22]）为 1 所以运算结果必须执行四舍五入。

Q6: A = 33.0451，B = -20.9466，A +B = ?

```
A = 32'b0_10000100_00001000010111000101111
B = 32'b1_10000011_01001111001001010100011

价码对齐：
A.exp - B.exp = 0b10000100 - 0b10000011 = 1₁₀

$ 1 mean A.exp > B.exp, while B.exp >> 1 and B.exp = A.exp

B.exp = A.exp = 0b10000100

hh m
```

```
01 01001111001001010100011() before
10 10100111100100101010001(1) >>1
10 10100111100100101010001(1) after, B = B’

运算预处理:
$ B’.s is 1, while ~B’.m + 1

s hh M
0 00 10100111100100101010001(1) before
1 11 01011000011011010101110(0) ~
1 11 01011000011011010101110(1) +1
1 11 01011000011011010101110(1) after, B’ = B″

加减操作:
e         s hh M
10000100 0 01 00001000010111000101111(0) A
10000100 1 11 01011000011011010101110(1) B″ +
10000100 0 00 01100000110010011011101(1) sum

结果预处理:
$ isSign = sum.s = 0

结果调整:
$ sum.hh is 2’b00, and Temp[44] is 1, while sum.m << 2 and sum.exp - 2

sum’.exp = 0b10000100 - 2 = 0b10000010

00 01100000110010011011101(1) before
01 10000011001001101110110(0) <<2
01 10000011001001101110110(0) after, sum = sum’

输出和格式化:
32’b0_10000010_10000011001001101110110 Test Result 12.0985012
32’b0_10000010_10000011001001101110101 Real Result 12.0985021
                                         True Result 12.0985
```

Q6 没有什么特别之处，除了在结果调整中，隐藏位为 2'b00，恰好尾数补偿的最高第二位（Temp[44]）为 1，所以尾数必须左移 2 位，然后价码 -2。输出结果中 Test Result 和 Real Result 的精度有点微差，但是结果都是正确的范围之内。

Q7：A = 7.751，B = 5.382，A + B = ?

```
A = 32’b0_10000001_11110000000100000110001
B = 32’b0_10000001_01011000011100101011000
```

```
对齐操作:
A.exp - B.exp = 0b10000001 - 0b10000001 = 0

运算预处理:
$ do nothing ...

加减操作:
e          s hh m
10000001 0 01 11110000000100000110001 a
10000001 0 01 01011000011100101011000 b +
10000001 0 11 01001000100000110001001 sum

结果预处理:
$ isSign = sum.s = 0;

结果调整:
$ sum.hh is 2'b10, while sum.m >> 1 and sum.exp + 1

sum'.exp = 0b10000001 + 1 = 0b10000010

hh
11 01001000100000110001001     before
01 10100100010000011000100(1)  >>1
01 10100100010000011000100(1)  after, sum = sum'

输出和格式化:
$ Temp[22] is 1, while Temp[45:23] + 1

m
10100100010000011000100(1) before
10100100010000011000101(0) after, sum' = sum"

32'b0_10000010_10100100010000011000101 Test Result 13.1329994
32'b0_10000010_10100100010000011000101 Real Result 13.1329994
                                       True Result 13.133
```

Q7 和 Q1 一样，没有什么特别。

Q8：A = 1.9999997，B = -1.9999998, A + B = ?

```
A = 32'b0_01111111_11111111111111111111110
B = 32'b1_01111111_11111111111111111111111
```

```
对齐操作：
A.exp - B.exp = 0b01111111 - 0b01111111 = 0

运算预处理：
$ B.s is 1, while ~B.m + 1

s hh m
0 01 11111111111111111111111 before
1 10 00000000000000000000000 ~
1 10 00000000000000000000001 +1
1 10 00000000000000000000001 after, B = B'

加减操作：
e        s hh m
01111111 0 01 11111111111111111111110 A
01111111 1 10 00000000000000000000001 B' +
01111111 1 11 11111111111111111111111 sum

结果预处理：
$ isSign = sum.s = 1
$ sum.s is 1, while ~sum.m + 1

s hh m
1 11 11111111111111111111111 before
0 00 00000000000000000000000 ~
0 00 00000000000000000000001 +1
0 00 00000000000000000000001 after, sum = sum'

结果调整：
$ sum'.hh is 2'b00 and Temp[23] is 1, while sum'.m << 23 and sum'.exp - 23

sum".exp = 0b01111111 - 23 = 0b01101000

hh
00 00000000000000000000001 before
01 00000000000000000000000 << 23
01 00000000000000000000000 after, sum' = sum"

输出和格式化：
32'b1_01101000_00000000000000000000000 Test Result -0.0000001
32'b1_01101000_00000000000000000000000 Real Result -0.0000001
                                           True Result -0.0000001
```

Q7 是一个极度挑剔的问题，然而这个问题的重点就在于“结果调整”这个步骤里，当结

果预处理以后，我们得到的隐藏位为 2'b00，此外小数点以后的第 23 位，亦即 Temp[23] 的值为 1，因此尾数必须左移 23 位，然后价码减去 23。这个现象就涉及“结果调整”即步骤 6 的设计了。如果隐藏位为 2'b00，作为一般的处理我们只是检查致小数点后的最低位而已，并且不涉及继续往后检测之后的尾数补偿。人算不如天算，我们不知道什么时候加减的结果，不但隐藏位为 2'b00,而且小数点以后的 23 位全 0，然而在尾数补偿里巧好存在着某个值为 1 ... 给个具体一点的例子：

```
hh m                          m backup
00_00000000000000000000000(00000000000000000000001)
```

对于这种问题，我们只有两个解决方法而已。第一方法就是反馈”尾数零值“的错误，还记得笔者讲过什么嘛吗？尾数零值基本上有两个可能，值无限大或者超过格式的承载能力。对于单精度浮点数的格式来说，小数点以后 23 位全 0 也表示了“精度超过承载范围”，反馈尾数零值错误也是情有可原。

第二个方法就是把步骤 6 的“结果调整”范围延伸到尾数补偿的最后，如代码 1.5.1 所示。这是一种吃力不讨好的方法，我们为了那么一个小小的结果，而不惜耗费庞大的逻辑资源。

```
else if( Temp[47:46] == 2'b00 && Temp[23] ) begin Temp <= Temp << 23; rExp <= rExp - 5'd23; end
...
else if( Temp[47:46] == 2'b00 && Temp[1] ) begin Temp <= Temp << 45; rExp <= rExp - 6'd45; end
else if( Temp[47:46] == 2'b00 && Temp[0] ) begin Temp <= Temp << 46; rExp <= rExp - 6'd46; end
```

代码 1.5.1 延长结果调整的范围到尾数补偿。

好了，这一小节也差不多了，估计经过这小节的洗礼以后，读者和二进制的认识是不是更上一层楼呢？

1.6 实验一：浮点数加法器

正是久违的实验，我们做了那么多准备就是实现单精度浮点数的加法器。读者知道吗？据说单精度浮点数加/减法器，比起单精度浮点数乘/除更难实现？笔者倒不觉得，这话只是懒人的借口而已 ... 因为只要搞定了单精度的原理，所有功夫都简单了。不过说老实话，单精度浮点数乘发起确实有点难度。好了，我们进入正题吧：

图 1.6.1 浮点数价码模块图形。

图 1.6.1 是这个实验要建立的浮点数加法模块。Start_Sig 和 Done_Sig 是控制信号，作为启动和反馈完成，不过不同的只是 Done_Sig 信号有 4 位位宽，它们分别为 { isOver, isUnder, isZero, isDone }。Done_Sig[3:1] 是用来反馈错误信息，价码上溢，价码下溢，尾数零值等；Done_Sig[0] 才是传统上的完成信号。其他的如 A 和 B 是 32 位宽的操作数输入信号，Result 则是 32 位宽的输出结果。其本上就是这些而已 ...

float_add_module.v

```
1.    module float_add_module
2.    (
3.        input CLK, RSTn,
4.        input [31:0]A,B,
5.        output [31:0]Result,
6.
7.        input Start_Sig,
8.        output [3:0]Done_Sig,
9.
10.       /*****************/
11.       output [56:0]SQ_rA,SQ_rB,
12.       output [48:0]SQ_Temp,SQ_TempA,SQ_TempB,
13.       output [9:0]SQ_rExp,
14.       output [7:0]SQ_rExpDiff
15.
16.   );
```

第 1~16 行是该模块的谁输出声明，第 3~8 行的信号基本上和图 1.6.1 大同小异，至于第 11~14 行是仿真用的信号。

```
17.     /**************************************/
18.
19.         reg [3:0]i;
20.         reg [56:0]rA,rB;          // [56]Sign, [55:48]Exponent, [47:46]Hidden Bit, [45:23]Mantissa [22:0]M'Backup
21.         reg [48:0]Temp;           // [48]M'sign, [47:46]Hidden Bit, [45:23]M, [22:0]M'Backup
22.         reg [48:0]TempA,TempB; //[48]M'sign, [47:46]Hidden Bit, [45:23]M, [22:0]M'Backup
23.         reg [31:0]rResult;
24.         reg [9:0]rExp;            //[9:8] Overflow or underflow check, [7:0] usuall exp.
25.         reg [7:0]rExpDiff;      //Different between A.Exp and B.Exp
26.         reg isSign;
27.         reg isOver;        // exp overflow error feedback
28.         reg isUnder;      // exp underflow error feedback
29.         reg isZero;        // m zero error feedback
30.         reg isDone;
```

第 19~30 行是相关的寄存器声明。rA 和 rB 除了寄存操作数 A 和 B 以外，它还用来恢复隐藏位 [47:46]，并且还添加了尾数补偿的空间 [22:0]。Temp 寄存器是用来寄存 TempA 和 TempB 的运算结果，话说 TempA 和 TempB 操作数运算前（运算预处理）寄存用的寄存器，[48]为符号位，[47:46] 是隐藏位，[45:23]是尾数，[22:0]是尾数补偿。

rResult 是用来驱动 Result 信号；rExp 是用来寄存价码差以外，它还有后期等工作；rExpDiff 是用来寄存价码差的正直，价码对齐的时候被应用；isSign 是用来寄存运算结果（加减操作）的符号位; isOver 是价码上溢的标志，isUnder 是价码下溢的标志，isZero 是尾数零值的表示。

```
31.
32.         always @ ( posedge CLK or negedge RSTn )
33.             if( !RSTn )
34.                 begin
35.                     i <= 4'd0;
36.                     rA <= 57'd0;
37.                     rB <= 57'd0;
38.                     TempA <= 49'd0;
39.                     TempB <= 49'd0;
40.                     Temp <= 49'd0;
41.                     rResult <= 32'd0;
42.                     rExp <= 10'd0;
43.                     rExpDiff <= 8'd0;
44.                     isOver <= 1'b0;
45.                     isUnder <= 1'b0;
46.                     isZero <= 1'b0;
47.                     isDone <= 1'b0;
48.                 end
```

第 35~47 行是相关的复位操作，所有寄存器都清零。

```
49.         else if( Start_Sig )
50.             case( i )
51.
52.                 0: // Initial A,B and other reg.
53.                 begin
54.                     rA <= { A[31], A[30:23], 2'b01, A[22:0], 23'd0 };
55.                     rB <= { B[31], B[30:23], 2'b01, B[22:0], 23'd0 };
56.
57.                     isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0;
58.                     i <= i + 1'b1;
59.                 end
60.
61.                 1: // if rExp[9..8] is 1, mean A.Exp small than B.Exp
62.                    // while rExp[9..8] is 0, mean A.Exp large than B.Exp or same.
63.                 begin
64.                     rExp = A[30:23] - B[30:23];
65.
66.                     if( rExp[8] == 1 ) rExpDiff <= ~rExp[7:0] + 1'b1;
67.                     else rExpDiff <= rExp[7:0];
68.                     i <= i + 1'b1;
69.                 end
```

第 52~59 行是步骤 0，亦即操作数预处理。第 54~55 行是操作 A 和 B 的预处理操作，第 57 行则清零 3 个错误标志寄存器。第 61~69 行是对齐操作中取价码差，第 64 行 rExp 取得 A 价码减去 B 价码的即时结果。第 66~67 行则取得价码差的正直，其中 if(rExp[8] == 1) 则表示，如果价码差是赋值的话 ...

```
70.
71.                 2:  // if A < B; A.M move and A.E = B.E, else opposite act;
72.                 begin
73.                     if( rExp[8] == 1 ) begin rA[47:0] <= rA[47:0] >> rExpDiff; rA[55:48] <= rB[55:48]; end
74.                     else begin rB[47:0] <= rB[47:0] >> rExpDiff; rB[55:48] <= rA[55:48]; end
75.                     i <= i + 1'b1;
76.                 end
77.
78.                 3: // Modify TempA and TempB. with sign
79.                 begin
80.                     TempA <= rA[56] ? { rA[56], (~rA[47:0] + 1'b1) } : { rA[56], rA[47:0] };
81.                     TempB <= rB[56] ? { rB[56], (~rB[47:0] + 1'b1) } : { rB[56], rB[47:0] };
82.                     i <= i + 1'b1;
83.                 end
84.
85.                 4: // Addition
86.                 begin Temp <= TempA + TempB; i <= i + 1'b1; end
```

第 71~76 行价码对齐中“向大看齐”，亦即较小价码的操作数，除了移动尾数以外，还要更新自身的价码。第 73 行表示，如果价码差为负值即是 A.exp < B.exp，A 的尾数位移后则更新自身的价码;第 74 行则相反动作。第 78~83 行是运算预处理操作，该步骤会根据操作数的符号化，是否将尾数转换为补码形式，并且寄存在操作空间 TempA 和 TempB。第 85 行是加减操作。

```
87.
88.         5: // modify result
89.         begin
90.             isSign <= Temp[48];
91.             if( Temp[48] == 1'b1) Temp <= ~Temp + 1'b1; // change M be postive
92.             rExp <= {2'b00, rA[55:48]}; // or rB[55:48] , change rExp withbe rA.Exp or rB.Exp
93.             i <= i + 1'b1;
94.         end
```

第 87~84 行是结果预处理，isSign 寄存操作结果的符号位（第 90 行）；第 91 行的根据符号位，把结果正直化；第 92 行则是更新 rExp。

```
95.
96.         6: // Check M'hidden bit and modify to 2'b01
97.         begin
98.             if( Temp[47:46] == 2'b10 || Temp[47:46] == 2'b11) begin Temp <= Temp >> 1; rExp <= rExp + 1'b1; end
99.             else if( Temp[47:46] == 2'b00 && Temp[45] ) begin Temp <= Temp << 1; rExp <= rExp - 5'd1; end
100.            else if( Temp[47:46] == 2'b00 && Temp[44] ) begin Temp <= Temp << 2; rExp <= rExp - 5'd2; end
101.            else if( Temp[47:46] == 2'b00 && Temp[43] ) begin Temp <= Temp << 3; rExp <= rExp - 5'd3; end
102.            else if( Temp[47:46] == 2'b00 && Temp[42] ) begin Temp <= Temp << 4; rExp <= rExp - 5'd4; end
103.            else if( Temp[47:46] == 2'b00 && Temp[41] ) begin Temp <= Temp << 5; rExp <= rExp - 5'd5; end
104.            else if( Temp[47:46] == 2'b00 && Temp[40] ) begin Temp <= Temp << 6; rExp <= rExp - 5'd6; end
105.            else if( Temp[47:46] == 2'b00 && Temp[39] ) begin Temp <= Temp << 7; rExp <= rExp - 5'd7; end
106.            else if( Temp[47:46] == 2'b00 && Temp[38] ) begin Temp <= Temp << 8; rExp <= rExp - 5'd8; end
107.            else if( Temp[47:46] == 2'b00 && Temp[37] ) begin Temp <= Temp << 9; rExp <= rExp - 5'd9; end
108.            else if( Temp[47:46] == 2'b00 && Temp[36] ) begin Temp <= Temp << 10; rExp <= rExp - 5'd10; end
109.            else if( Temp[47:46] == 2'b00 && Temp[35] ) begin Temp <= Temp << 11; rExp <= rExp - 5'd11; end
110.            else if( Temp[47:46] == 2'b00 && Temp[34] ) begin Temp <= Temp << 12; rExp <= rExp - 5'd12; end
111.            else if( Temp[47:46] == 2'b00 && Temp[33] ) begin Temp <= Temp << 13; rExp <= rExp - 5'd13; end
112.            else if( Temp[47:46] == 2'b00 && Temp[32] ) begin Temp <= Temp << 14; rExp <= rExp - 5'd14; end
113.            else if( Temp[47:46] == 2'b00 && Temp[31] ) begin Temp <= Temp << 15; rExp <= rExp - 5'd15; end
114.            else if( Temp[47:46] == 2'b00 && Temp[30] ) begin Temp <= Temp << 16; rExp <= rExp - 5'd16; end
115.            else if( Temp[47:46] == 2'b00 && Temp[29] ) begin Temp <= Temp << 17; rExp <= rExp - 5'd17; end
116.            else if( Temp[47:46] == 2'b00 && Temp[28] ) begin Temp <= Temp << 18; rExp <= rExp - 5'd18; end
117.            else if( Temp[47:46] == 2'b00 && Temp[27] ) begin Temp <= Temp << 19; rExp <= rExp - 5'd19; end
118.            else if( Temp[47:46] == 2'b00 && Temp[26] ) begin Temp <= Temp << 20; rExp <= rExp - 5'd20; end
119.            else if( Temp[47:46] == 2'b00 && Temp[25] ) begin Temp <= Temp << 21; rExp <= rExp - 5'd21; end
120.            else if( Temp[47:46] == 2'b00 && Temp[24] ) begin Temp <= Temp << 22; rExp <= rExp - 5'd22; end
121.            else if( Temp[47:46] == 2'b00 && Temp[23] ) begin Temp <= Temp << 23; rExp <= rExp - 5'd23; end
```

```
122.                //else do nothing, can extend the hidden bit check area
123.
124.                i <= i + 1'b1;
125.            end
```

第 96~125 行是结果调整，至于要不要继续扩展隐藏位的搜索就自己看着办（第 122 行）。

```
126.
127.            7: //error check and format result in float format
128.            begin
129.                if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end          // E Overflow
130.                else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end    // E Underflow
131.                else if( Temp[46:23] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end   // M Zero
132.                else if( Temp[22] == 1'b1 ) rResult <= { isSign, rExp[7:0], Temp[45:23] + 1'b1 };   // okay with normalised
133.                else rResult <= { isSign, rExp[7:0], Temp[45:23] };                                  // okay without normalise
134.                i <= i + 1'b1;
135.            end
136.
137.            8:
138.            begin isDone <= 1'b1; i <= i + 1'b1; end
139.
140.            9:
141.            begin isDone <= 1'b0; i <= 4'd0; end
142.
143.        endcase
```

第 127~135 行是输出和格式化，第 129 行是检测价码上溢；第 130 行是检测价码下溢；第 131 行是检测尾数零值；第 132 行是四舍五入的操作；第 133 行则是不同的输出。步骤 8~9 是用来产生完成信号。

```
144.
145.    /*************************************/
146.
147.    assign Done_Sig = { isOver, isUnder, isZero, isDone };
148.    assign Result = rResult;
149.
150.    /*************************************/
151.
152.    assign SQ_rA = rA;
153.    assign SQ_rB = rB;
154.    assign SQ_Temp = Temp;
155.    assign SQ_TempA = TempA;
156.    assign SQ_TempB = TempB;
157.    assign SQ_rExp = rExp;
```

```
158.        assign SQ_rExpDiff = rExpDiff;
159.
160.        /****************************************/
161.
162.    endmodule
```

第 147~148 行是相关的输出驱动，第 152~158 行则是仿真用的驱动输出。基本上浮点数加法模块的功能就是这么简单而已。

float_add_module.vt

```
1.    `timescale 1 ps/ 1 ps
2.    module float_add_module_simulation();
3.
4.            reg CLK;
5.            reg RSTn;
6.
7.            reg Start_Sig;
8.            reg [31:0] A;
9.            reg [31:0] B;
10.
11.           wire [3:0]    Done_Sig;
12.           wire [31:0]    Result;
13.
14.           /******************/
15.
16.         wire [56:0]SQ_rA,SQ_rB;
17.         wire [48:0]SQ_Temp,SQ_TempA,SQ_TempB;
18.         wire [9:0]SQ_rExp;
19.         wire [7:0]SQ_rExpDiff;
20.
21.           /*********************************/
22.
23.           float_add_module U1
24.           (
25.                   .CLK( CLK ),
26.                   .RSTn( RSTn ),
27.                   .A( A ),
28.                   .B( B ),
29.                   .Result( Result ),
30.                   .Start_Sig( Start_Sig ),
31.                   .Done_Sig( Done_Sig ),
32.                   .SQ_rA(SQ_rA),
33.                   .SQ_rB(SQ_rB),
```

```
34.             .SQ_Temp( SQ_Temp ),
35.             .SQ_TempA( SQ_TempA ),
36.             .SQ_TempB( SQ_TempB ),
37.             .SQ_rExp( SQ_rExp ),
38.             .SQ_rExpDiff( SQ_rExpDiff )
39.         );
40.
41.         /***********************************/
42.
43.         initial
44.         begin
45.             RSTn = 0; #10 RSTn = 1;
46.             CLK = 0; forever #5 CLK = ~CLK;
47.         end
48.
49.         /***********************************/
50.
51.         reg [3:0]i;
52.
53.         always @ (   posedge CLK or negedge RSTn )
54.             if( !RSTn )
55.                 begin
56.                     A <= 32'd0;
57.                     B <= 32'd0;
58.                     Start_Sig <= 1'b0;
59.                     i <= 4'd0;
60.                 end
61.           else
62.                 case( i )
63.
64.                     0: //A=3.65, B= -7.4, A+B = ?
65.                     if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
66.                 else begin A <= 32'b0_10000000_11010011001100110011010; B <= 32'b1_10000001_11011001100110011001101; Start_Sig <= 1'b1; end
67.
68.                     1: //Exp undeflow check
69.                     if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
70.                 else begin A <= 32'b0_00000000_01010000101101000101101; B <= 32'b1_00000000_00010000101100001000111; Start_Sig <= 1'b1; end
71.
72.                    2: //A=1.9999997, B=-1.9999998 , A+B =?
73.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
74.                 else begin A <= 32'b0_01111111_11111111111111111111110; B <= 32'b1_01111111_11111111111111111111111; Start_Sig <= 1'b1; end
75.
76.                     3: //Exp Overflow
77.                     if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
78.                 else begin A <= 32'b0_11111111_11111111111111111111111; B <= 32'b0_11111111_11111111111111111111111; Start_Sig <= 1'b1; end
```

```
79.
80.                              4: //A= -12.558, B= -7.309 , A+B =?
81.                              if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
82.                      else begin A <= 32'b11000001010010001110110110010001; B <= 32'b11000000111010011110001101010100; Start_Sig <= 1'b1; end
83.
84.                              5: //A= 111.7762, B= 302.4409 , A+B =?
85.                              if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
86.                      else begin A <= 32'b01000010110111111000110101101010; B <= 32'b01000011100101110011100001101111; Start_Sig <= 1'b1; end
87.
88.                              6: //A= 2112.2012, B= -2002.2012 , A+B =?
89.                              if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
90.                      else begin A <= 32'b01000101000001000000001100111000; B <= 32'b11000100111110100100011001110000; Start_Sig <= 1'b1; end
91.
92.                               7:
93.                               i <= i;
94.
95.                         endcase
96.
97.    endmodule
```

以上是浮点数加法模块的激励文件，激励文件的书写风格笔者就不重复了，除了关键字 $display 以外，其他都不陌生。$display 是验证语言，主要是用来输出结果，用法和 C 语言的 printf 类似。顺便补上一句，$display 函数在时序的效果上和组合逻辑一样，都是即时事件。步骤 0，2，4，5，6 是用验证测该模块的通常运作，步骤 1 和 3 则是用来测试价码上溢和下溢的运作。

仿真结果：

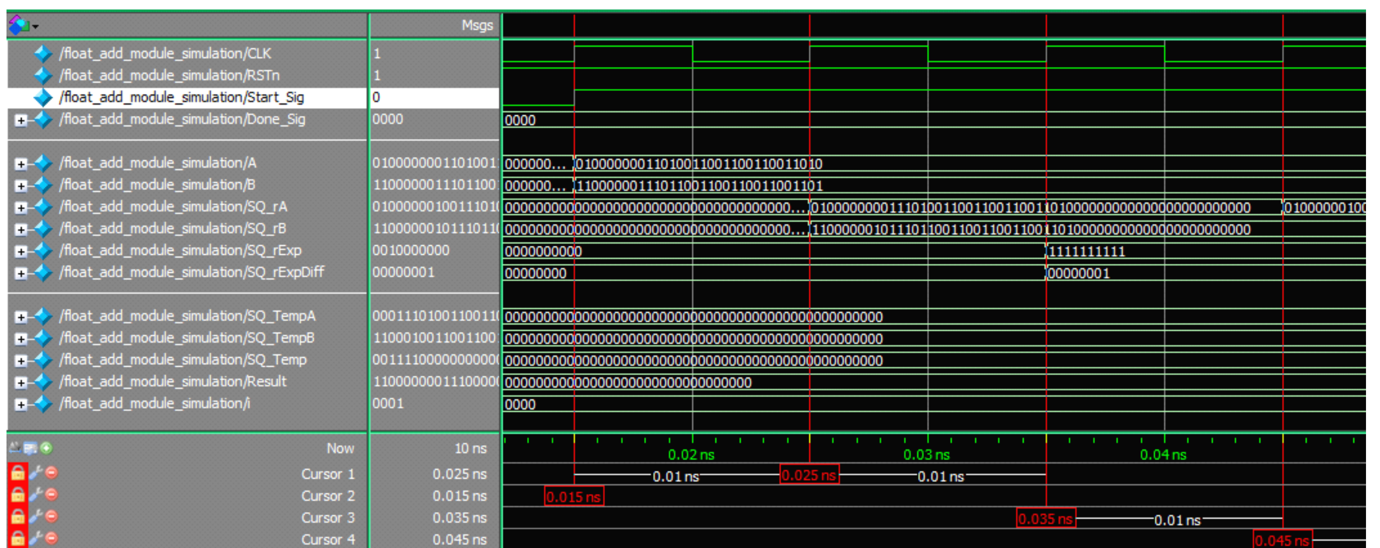

图 1.6.2 仿真流程 1

图 1.6.2 是 3.65 -7.4 的运算过程，C1（以下 Curosr 简称为 C）表示激励文件吧操作数 A 和 B 输

入以后，并且拉高 Start_Sig，在 C1 的未来 32'b0_10000000_11010011001100110011010 和 32'b1_10000001_11011001100110011001101 被输出。在 C2 的时候是操作数预处理，浮点数加法模块读入该过去值，在 C2 的未来输出操作数的预操作结果。在 C3 的时候是取价码差（记得价码差的取得是“即时事件”）结果是 8'b11111111，既是-1 也是 A.exp < B.exp。在 C3 的未来，输出正直化以后的价码差。

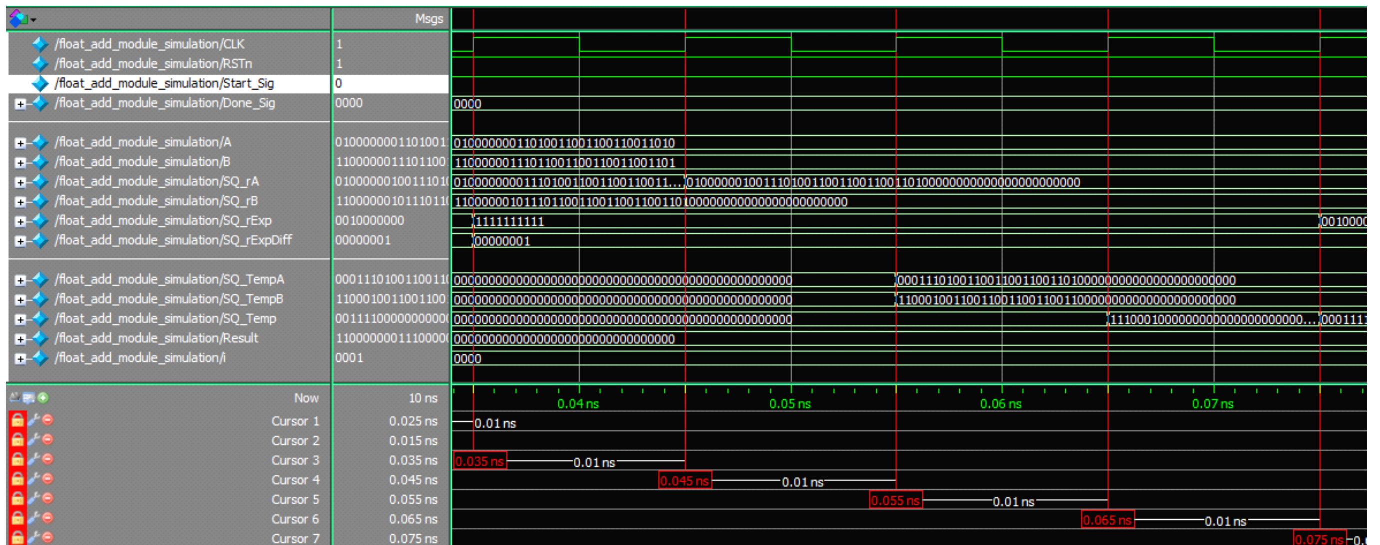

图 1.6.3 仿真流程 2

C4 的时候是“价码向大对齐”的操作，并且更新相关的价码。在 C4 的未来值中，我们知道 rExp[8] 为 1 亦即 A.exp < B.exp，因此决定 A 的尾数右移 rExpDiff 在 C4 的过去值（亦即 1）。所以在 C4 的未来，A 的尾数不但右移 1 位，而且价码的更新结果也和 B 一样。C5 的时候是运算预处理，根据 rA 和 rB 在 C5 中过去的结果，rB 带有符号位，所以 rB 的尾数必须取得补码形式，所以在 C5 的未来，rA 和 rB 的预处理结果会寄存在操作空间 TempA 和 TempB 中。C6 的时候是加减运算，C6 的未来中所输出的值是 TempA 和 TempB 在 C6 过去中相加的结果。

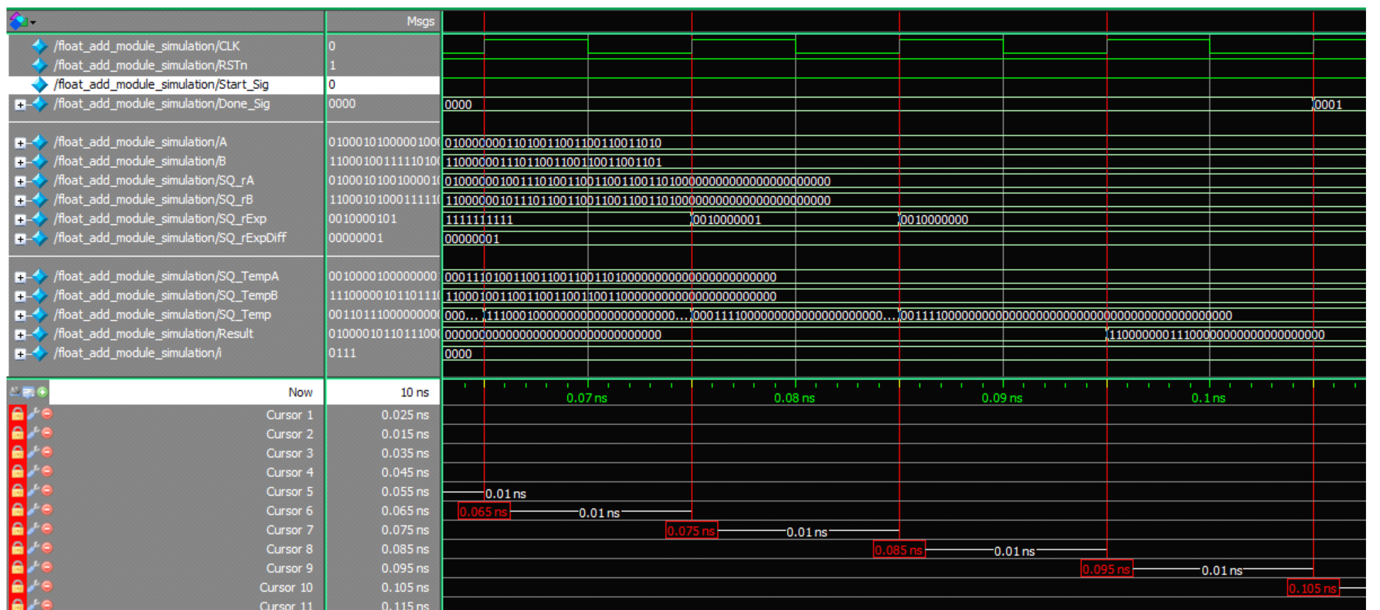

图 1.6.4 仿真流程 3

C7 的时候是结果预处理，isSign 取得在 C7 中 Temp[48]的过去值，亦即是 1。此外，根据过去值 Temp[48]的结果，该值是 1 所以正值化 Temp，在 C7 的未来，被输出的 Temp 为正值化的结果，并且更新 rExp 的结果。C8 的时候是结果调整，Temp[47:46]的过去值为 2'b00（隐藏位），所以在 C8 的未来输出尾数左移一位的结果，其中 rExp 被减去 1。

C9 的时候是输出和个时候，rExp 的过去值为 10'b00_10000000，其中 rExp[9:8] 不是 2'b01 或者 2'b11，所以不发生价码溢出的问题。此外，Temp[22]的过去值也不是 1，所以也不对尾数四舍五入。在 C9 的未来，输出 isSign 为 1，rExp 为 8'b10000000，和尾数为 Temp[45:23]的结果。在这次的加减预算中，没有发生任何错误，所以在 C9 的未来也不输出任何错误反馈。至于 C10~C11 是完成信号的产生经过，怒笔者不解释了。

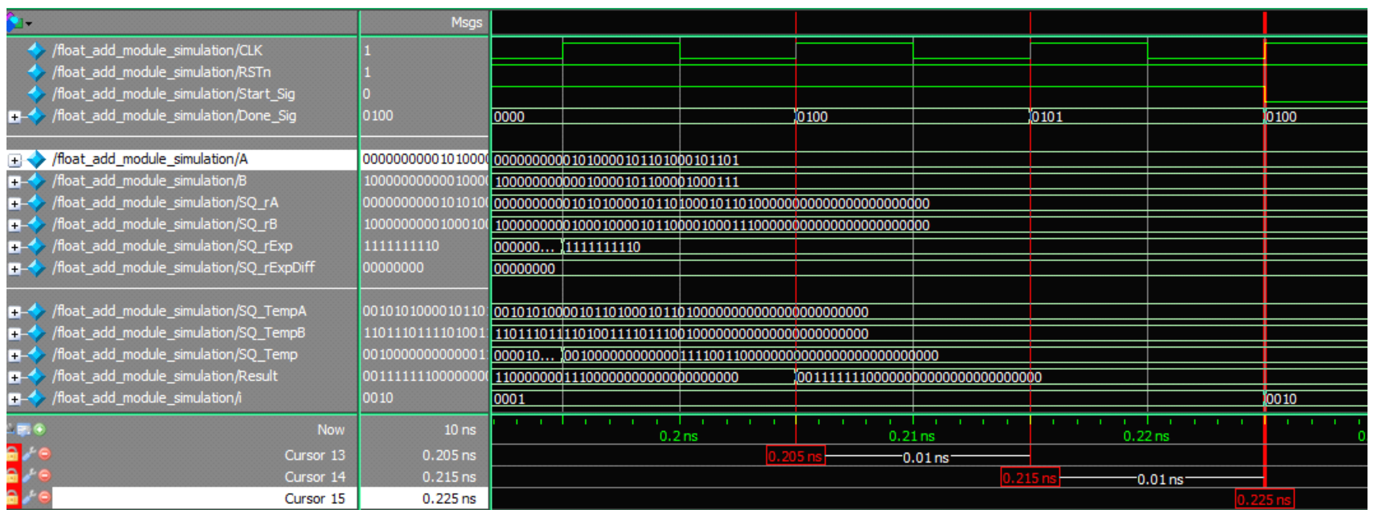

图 1.6.5 价码下溢。

至于图 1.6.5 是激励中故意引起的价码下溢，如 C13 指向的地方，rExp[9:8]的过去值为 2'b11，因此在 C13 的未来判断为价码下溢，拉高 isUnder 的标志（Done_Sig[2]），并且输出默认空壳的值。

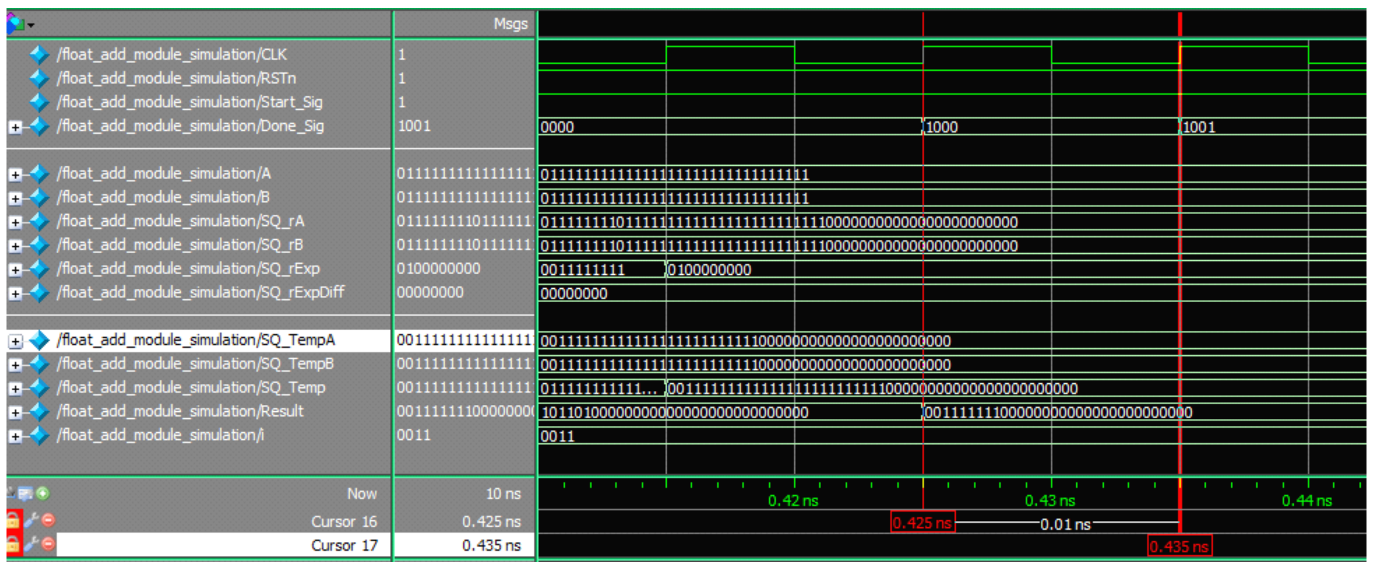

图 1.6.6 价码上溢。

图 1.6.6 同为笔者故意引起的价码上溢，C17 所指向的地方 rExp[9:8]的过去值为 2'b10，因此断定为价码上溢，所以在 C17 的未来，拉高价码上溢的标志（Done_Sig[3]），并且输出默认空壳。关于尾数零值错误信息的产生经过也是大同小异而已。往后还有 -12.558 -7.309，111.7762 + 302.4409 和 2112.2012 -2002.2012 的运算过程也差不多而已，在此笔者就不加说明了，作为补偿笔者就简化运算一下最后一题的过程。

Q9: A= 2112.2012，B = -2002.2012, A + B = ?

```
A = 32'b0_10001010_00001000000001100111000
B = 32'b1_10001001_11110100100011001110000

价码对齐：
A.exp - B.exp = 0b10001010 - 0b10001001 = 1₁₀

$ 1 mean A.exp > B.exp while, B.m >> 1 and B.exp = A.exp;

hh m
01 11110100100011001110000()  before
00 11111010010001100111000(0) >> 1
00 11111010010001100111000(0) after, B = B'

运算前预处理：
$ B'.s is 1, while ~B'.m + 1

s hh m
0 00 11111010010001100111000(0)  before
1 11 00000101101110011000111(1) ~
1 11 00000101101110011001000(0) +1
1 11 00000101101110011001000(0) after, B' = B"

加减运算：
e        s hh m
10001010 0 01 00001000000001100111000     A
10001010 1 11 00000101101110011001000(0)  B" +
10001010 0 00 00001101110000000000000(0) sum

结果预处理：
$ isSign = sum.s = 0

结果调整：
$ sum.hh is 2'b00 and Temp[41] is 1, while sum.m << 5 and sum.exp - 5

sum'.exp = 0b10001010 - 5 = 0b10000101
```

```
hh m
00 00001101110000000000000(0) before
01 10111000000000000000000(0) << 5
01 10111000000000000000000(0) after, sum = sum'

输出和格式化:
32'b0_10000101_10111000000000000000000 Test Result 110
32'b0_10000101_10111000000000000000000 Real Result 110
                                            True Result 110
```

基本上，实验 1 中浮点数加法模块的运算过程就是这样而已，没有什么特别需要强调的地方。只要把最基本的概念都搞懂了，设计单精度浮点数的加法器一点也不困难。好了，这个实验九到此为止把，下一个小节是浮点数减法器的实验，大体上和浮点数加法器大同小异而已，因为笔者在实验一已经买下一些微调的种子，所以只要在相关的步骤动一下手脚就行了。

1.7 实验二：浮点数减法器

图 1.7.1 浮点数减法模块图形。

图 1.7.1 是浮点数减法模块的图形，信号的定义上基本上和浮点数加法模块一模一样，甚至内容也是大同小异，讲多无用我们还是直接来看代码吧。

float_sub_module.v

```
1.   module float_sub_module
2.   (
3.        input CLK, RSTn,
4.        input [31:0]A,B,
5.        output [31:0]Result,
6.
7.        input Start_Sig,
8.        output [3:0]Done_Sig,
9.
10.       /*****************/
11.       output [56:0]SQ_rA,SQ_rB,
12.       output [48:0]SQ_Temp,SQ_TempA,SQ_TempB,
13.       output [9:0]SQ_rExp,
14.       output [7:0]SQ_rExpDiff
15.
16.  );
17.       /**************************************/
18.
19.      reg [3:0]i;
20.      reg [56:0]rA,rB;   // [56]Sign, [55:48]Exponent, [47:46]Hidden Bit, [45:23]Mantissa [22:0]M'resolution
21.      reg [48:0]Temp;     // [48]M'sign, [47:46]Hidden Bit, [45:23]M, [22:0]M'resolution
22.      reg [48:0]TempA,TempB; //[48]M'sign, [47:46]Hidden Bit, [45:23]M, [22:0]M'resolution
23.      reg [31:0]rResult;
24.      reg [9:0]rExp;        //[9:8] Overflow or underflow check, [7:0] usuall exp.
25.      reg [7:0]rExpDiff; //Different between A.Exp and B.Exp
26.      reg isSign;
27.      reg isOver;
28.      reg isUnder;
```

```
29.     reg isZero;
30.     reg isDone;
31.
32.     always @ ( posedge CLK or negedge RSTn )
33.         if( !RSTn )
34.             begin
35.                 i <= 4'd0;
36.                 rA <= 57'd0;
37.                 rB <= 57'd0;
38.                 TempA <= 49'd0;
39.                 TempB <= 49'd0;
40.                 Temp <= 49'd0;
41.                 rResult <= 32'd0;
42.                 rExp <= 10'd0;
43.                 rExpDiff <= 8'd0;
44.                 isOver <= 1'b0;
45.                 isUnder <= 1'b0;
46.                 isZero <= 1'b0;
47.                 isDone <= 1'b0;
48.             end
49.         else if( Start_Sig )
50.             case( i )
51.
52.                 0: // Initial and resulted out Signature
53.                 begin
54.                     rA <= { A[31], A[30:23], 2'b01, A[22:0], 23'd0 };
55.                     rB <= { B[31], B[30:23], 2'b01, B[22:0], 23'd0 };
56.
57.                     isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0;
58.                     i <= i + 1'b1;
59.                 end
60.
61.                 1: // if rExp[9..8] is 1, mean A.Exp small than B.Exp
62.                    // while rExp[9..8] is 0, mean A.Exp large than B.Exp or same.
63.                 begin
64.                     rExp = A[30:23] - B[30:23];
65.
66.                     if( rExp[8] == 1 ) rExpDiff <= ~rExp[7:0] + 1'b1;
67.                     else rExpDiff <= rExp[7:0];
68.                     i <= i + 1'b1;
69.                 end
70.
71.                 2:   // if A < B; A.M move and A.E = B.E, else different act;
72.                 begin
73.                     if( rExp[8] == 1 ) begin rA[47:0] <= rA[47:0] >> rExpDiff; rA[55:48] <= rB[55:48]; end
```

```
74.                 else begin rB[47:0] <= rB[47:0] >> rExpDiff; rB[55:48] <= rA[55:48]; end
75.                 i <= i + 1'b1;
76.             end
77.
78.             3: // Modified TempA and TempB with own signed
79.             begin
80.                 TempA <= rA[56] ? { rA[56], (~rA[47:0] + 1'b1) } : { rA[56], rA[47:0] };
81.                 TempB <= rB[56] ? { rB[56], (~rB[47:0] + 1'b1) } : { rB[56], rB[47:0] };
82.                 i <= i + 1'b1;
83.             end
84.
85.             4: // Subtraction
86.             begin Temp <= TempA - TempB; i <= i + 1'b1; end
87.
88.             5:
89.             begin
90.                 isSign <= Temp[48];
91.                 if( Temp[48] == 1'b1) Temp <= ~Temp + 1'b1; // change M be unsigned
92.                 rExp <= {2'b00, rA[55:48]}; // or rB[55:48] , change rExp withbe rA.Exp or rB.Exp
93.                 i <= i + 1'b1;
94.             end
95.
96.             6: // Check M'hidden bit and modify in 2'b01
97.             begin
98.                 if( Temp[47:46] == 2'b10 || Temp[47:46] == 2'b11) begin Temp <= Temp >> 1; rExp <= rExp + 1'b1; end
99.                 else if( Temp[47:46] == 2'b00 && Temp[45] ) begin Temp <= Temp << 1; rExp <= rExp - 5'd1; end
100.                else if( Temp[47:46] == 2'b00 && Temp[44] ) begin Temp <= Temp << 2; rExp <= rExp - 5'd2; end
101.                else if( Temp[47:46] == 2'b00 && Temp[43] ) begin Temp <= Temp << 3; rExp <= rExp - 5'd3; end
102.                else if( Temp[47:46] == 2'b00 && Temp[42] ) begin Temp <= Temp << 4; rExp <= rExp - 5'd4; end
103.                else if( Temp[47:46] == 2'b00 && Temp[41] ) begin Temp <= Temp << 5; rExp <= rExp - 5'd5; end
104.                else if( Temp[47:46] == 2'b00 && Temp[40] ) begin Temp <= Temp << 6; rExp <= rExp - 5'd6; end
105.                else if( Temp[47:46] == 2'b00 && Temp[39] ) begin Temp <= Temp << 7; rExp <= rExp - 5'd7; end
106.                else if( Temp[47:46] == 2'b00 && Temp[38] ) begin Temp <= Temp << 8; rExp <= rExp - 5'd8; end
107.                else if( Temp[47:46] == 2'b00 && Temp[37] ) begin Temp <= Temp << 9; rExp <= rExp - 5'd9; end
108.                else if( Temp[47:46] == 2'b00 && Temp[36] ) begin Temp <= Temp << 10; rExp <= rExp - 5'd10; end
109.                else if( Temp[47:46] == 2'b00 && Temp[35] ) begin Temp <= Temp << 11; rExp <= rExp - 5'd11; end
110.                else if( Temp[47:46] == 2'b00 && Temp[34] ) begin Temp <= Temp << 12; rExp <= rExp - 5'd12; end
111.                else if( Temp[47:46] == 2'b00 && Temp[33] ) begin Temp <= Temp << 13; rExp <= rExp - 5'd13; end
112.                else if( Temp[47:46] == 2'b00 && Temp[32] ) begin Temp <= Temp << 14; rExp <= rExp - 5'd14; end
113.                else if( Temp[47:46] == 2'b00 && Temp[31] ) begin Temp <= Temp << 15; rExp <= rExp - 5'd15; end
114.                else if( Temp[47:46] == 2'b00 && Temp[30] ) begin Temp <= Temp << 16; rExp <= rExp - 5'd16; end
115.                else if( Temp[47:46] == 2'b00 && Temp[29] ) begin Temp <= Temp << 17; rExp <= rExp - 5'd17; end
116.                else if( Temp[47:46] == 2'b00 && Temp[28] ) begin Temp <= Temp << 18; rExp <= rExp - 5'd18; end
117.                else if( Temp[47:46] == 2'b00 && Temp[27] ) begin Temp <= Temp << 19; rExp <= rExp - 5'd19; end
118.                else if( Temp[47:46] == 2'b00 && Temp[26] ) begin Temp <= Temp << 20; rExp <= rExp - 5'd20; end
```

```
119.                else if( Temp[47:46] == 2'b00 && Temp[25] ) begin Temp <= Temp << 21; rExp <= rExp - 5'd21; end
120.                else if( Temp[47:46] == 2'b00 && Temp[24] ) begin Temp <= Temp << 22; rExp <= rExp - 5'd22; end
121.                else if( Temp[47:46] == 2'b00 && Temp[23] ) begin Temp <= Temp << 23; rExp <= rExp - 5'd23; end
122.                //else do nothing
123.
124.                i <= i + 1'b1;
125.            end
126.
127.            7: //error check and decide final result
128.            begin
129.                if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end      // E Overflow
130.                else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // E Underflow
131.                else if( Temp[46:23] == 23'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
132.                else if( Temp[22] == 1'b1 ) rResult <= { isSign, rExp[7:0], Temp[45:23] + 1'b1 }; // okay with normalised
133.                else rResult <= { isSign, rExp[7:0], Temp[45:23] }; // okay without normalise
134.                i <= i + 1'b1;
135.            end
136.
137.            8:
138.            begin isDone <= 1'b1; i <= i + 1'b1; end
139.
140.            9:
141.            begin isDone <= 1'b0; i <= 4'd0; end
142.
143.        endcase
144.
145.    /***************************************/
146.
147.    assign Done_Sig = { isOver, isUnder, isZero, isDone };
148.    assign Result = rResult;
149.
150.    /****************************************/
151.
152.    assign SQ_rA = rA;
153.    assign SQ_rB = rB;
154.    assign SQ_Temp = Temp;
155.    assign SQ_TempA = TempA;
156.    assign SQ_TempB = TempB;
157.    assign SQ_rExp = rExp;
158.    assign SQ_rExpDiff = rExpDiff;
159.
160.    /*****************************************/
161.
162. endmodule
```

以上是该模块的内容，除了第 85~86 行的加减操作以外，其余的内容都和浮点数加法模块一模一样。至于修改的内容，也是从原本的“+”变成“-”而已。在这里笔者再简单的复习一下 +/- 与整数之间的关系。事实上，编译器在编译 “-”运算符的时候，它会把接续的整数（二进制值）转换为补码形式，也就是我们常作的“取反再加一”。

float_sub_module.vt

```
1.    `timescale 1 ps/ 1 ps
2.    module float_sub_module_simulation();
3.
4.        reg CLK;
5.        reg RSTn;
6.
7.        reg Start_Sig;
8.        reg [31:0] A;
9.        reg [31:0] B;
10.
11.       wire [3:0]   Done_Sig;
12.       wire [31:0]   Result;
13.
14.       /******************/
15.
16.      wire [56:0]SQ_rA,SQ_rB;
17.      wire [48:0]SQ_Temp,SQ_TempA,SQ_TempB;
18.      wire [9:0]SQ_rExp;
19.      wire [7:0]SQ_rExpDiff;
20.
21.       /**********************************/
22.
23.       float_sub_module U1
24.       (
25.            .CLK( CLK ),
26.            .RSTn( RSTn ),
27.            .A( A ),
28.            .B( B ),
29.            .Result( Result ),
30.            .Start_Sig( Start_Sig ),
31.            .Done_Sig( Done_Sig ),
32.            .SQ_rA(SQ_rA),
33.            .SQ_rB(SQ_rB),
34.            .SQ_Temp( SQ_Temp ),
35.            .SQ_TempA( SQ_TempA ),
36.            .SQ_TempB( SQ_TempB ),
37.            .SQ_rExp( SQ_rExp ),
```

```
38.                 .SQ_rExpDiff( SQ_rExpDiff )
39.             );
40.
41.             /*************************************/
42.
43.             initial
44.             begin
45.                 RSTn = 0; #10 RSTn = 1;
46.                 CLK = 0; forever #5 CLK = ~CLK;
47.             end
48.
49.             /*************************************/
50.
51.             reg [4:0]i;
52.
53.             always @ (   posedge CLK or negedge RSTn )
54.                 if( !RSTn )
55.                     begin
56.                         A <= 32'd0;
57.                         B <= 32'd0;
58.                         Start_Sig <= 1'b0;
59.                         i <= 5'd0;
60.                     end
61.               else
62.                     case( i )
63.
64.                             0: //A=3.65, B=-7.4, A-B = ?
65.                             if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
66.           else begin A <= 32'b0_10000000_11010011001100110011010; B <= 32'b1_10000001_11011001100110011001101; Start_Sig <= 1'b1; end
67.
68.                             1: //Exp undeflow check
69.                             if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
70.           else begin A <= 32'b0_00000000_01010000101101000101101; B <= 32'b1_00000000_00010000101100001000111; Start_Sig <= 1'b1; end
71.
72.                            2: //A=1.9999997, B=-1.9999998 , A-B =?
73.                            if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
74.           else begin A <= 32'b0_01111111_11111111111111111111110; B <= 32'b1_01111111_11111111111111111111111; Start_Sig <= 1'b1; end
75.
76.                             3: //Exp Overflow
77.                             if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
78.          else begin A <= 32'b0_11111111_11111111111111111111111; B <= 32'b0_11111111_11111111111111111111111; Start_Sig <= 1'b1; end
79.
80.                            4: //A=78992.118, B=-90116.402, A-B =?
81.                            if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
82.          else begin A <= 32'b01000111100110100100100000001111; B <= 32'b11000111101100000000001000110011; Start_Sig <= 1'b1; end
```

```
83.
84.                     5: //A=6992.330, B=1271.004, A-B =?
85.                     if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
86.         else begin A <= 32'b01000101110110101000001010100100; B <= 32'b01000100100111101110000000100001; Start_Sig <= 1'b1; end
87.
88.                     6: //A=-1.5032, B=-8.3309, A-B =?
89.                     if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
90.         else begin A <= 32'b10111111110000000110100011011100; B <= 32'b11000001000001010100101101011110; Start_Sig <= 1'b1; end
91.
92.                     7: //A=-88.8888, B=-10.9944, A-B =?
93.                     if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
94.         else begin A <= 32'b11000010101100011100011100010001; B <= 32'b11000001001011111110100100010000; Start_Sig <= 1'b1; end
95.
96.                     8: //A=0.001584, B=0.063197, A-B =?
97.                     if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
98.         else begin A <= 32'b00111010110011111001111000111000; B <= 32'b00111101100000010110110101101110; Start_Sig <= 1'b1; end
99.
100.                    9: //A=-613.931, B=27.11402, A-B =?
101.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
102.        else begin A <= 32'b11000100000110010111101110010110; B <= 32'b01000001110110001110100110000011; Start_Sig <= 1'b1; end
103.
104.                    10: //A=44.4444, B=68.3333, A-B =?
105.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; $display("%b",Result); end
106.        else begin A <= 32'b01000010001100011100011100010001; B <= 32'b01000010100010001010101010101010; Start_Sig <= 1'b1; end
107.
108.                    11:
109.                      i <= i;
110.
111.                endcase
112.
113.   endmodule
```

步骤 0 与 2 基本和 float_add_module.vt 是一样的操作，但是随着操作符的不同，运算结果页随着不同。

仿真结果：

至于详细的仿真过程，基本上和实验一是换汤不换药，笔者就不重复了，读者自己看着办吧。换之，笔者把运算的结果打印出来给读者作以下比较（说实话以下有几组操作数是非常的挑剔）：

Q1: A = 3.65, B = -7.4, A - B = ?

```
A = 32’b0_10000000_11010011001100110011010
B = 32’b1_10000001_11011001100110011001101
```

```
32'b0_10000010_01100001100110011001101 Test Result 11.0500001
32'b0_10000010_01100001100110011001101 Teal Result 11.0500001
                                          True Result 11.05
```

Q2: A = 1.9999997，B = -1.9999998， A - B = ?

```
A = 32'b0_01111111_11111111111111111111110
B = 32'b1_01111111_11111111111111111111111

32'b0_10000000_11111111111111111111111 Test Result 3.9999997
32'b0_10000000_11111111111111111111111 Real Result 3.9999997
                                          True Result 3.9999997
```

Q3: A = 78992.118， B = -90116.402， A - B = ?

```
A = 32'b0_10001111_00110100100100000001111
B = 32'b1_10001111_01100000000001000110011

32'b0_10010000_01001010010010100100001 Test Result 169108.5156250
32'b0_10010000_01001010010010100100001 Real Result 169108.5156250
                                          True Result 169108.52
```

关于这个问题，读者需要注意的地方就是精度问题了。A - B 的结果显示 Test Result 和 Real Result 已经开始偏远 True Result 了。原因是——单精度格式已经是勉强承载运算过后的结果。用一句傻瓜的话来说，8 个十进制位数（包括小数），单精度浮点数对它的支持已经出现“负荷”的现象了。

Q4: A = 6992.330，B = 1271.004，A - B = ?

```
A = 32'b0_10001011_10110101000001010100100
B = 32'b0_10001001_00111101110000000100001

32'b0_10001011_01100101100101010011100 Test Result 5721.3261718
32'b0_10001011_01100101100101010011100 Real Result 5721.3261718
                                          True Result 5721.326
```

比起 Q3，Q4 的 Test Result 和 Real Result 显得更接近 True Result。用傻瓜的话来说，7 个十进制的位数（包括小数），对于单精度格式来说是最大的承载极限。

Q5: A = -1.5032，B = -8.3309，A - B = ?

```
A = 32'b1_01111111_10000000110100011011100
B = 32'b1_10000010_00001010100101101011110
```

```
32'b0_10000001_10110100111110010000101 Test Result 6.8277001
32'b0_10000001_10110100111110010000101 Real Result 6.8277001
                                       True Result 6.8277
```

Q6: A = -88.8888，B = -10.9944，A - B = ?

```
A = 32'b1_10000101_01100011100011100010001
B = 32'b1_10000010_01011111110100100010000

A = 32'b1_10000101_00110111100100111101111 Test Result -77.8944015
B = 32'b1_10000101_00110111100100111101111 Real Result -77.8944015
                                           True Result -77.8944
```

对于拥有 6 个十进制位数的结果（包括小数），是最佳的状态。

Q7: A = 0.001584，B = 0.063197，A - B = ?

```
A = 32'b0_01110101_10011111001111000111000
B = 32'b0_01111011_00000010110110101101110

32'b1_01111010_11111000101110111101010 Test Result -0.0616130
32'b1_01111010_11111000101110111101010 Real Result -0.0616130
                                       True Result -0.061613
```

即只有小数的情况下，6 个位数的十进制（小数而已）的结果，单精度格式的支持还不错。

Q8: A = -613.931，B = 27.11402，A - B = ?

```
A = 32'b1_10001000_00110010111101110010110
B = 32'b0_10000011_10110001110100110000011

32'b1_10001000_01000000100001011100010 Test Result -641.0450439
32'b1_10001000_01000000100001011100010 Real Result -641.0450439
                                       True Result -641.04502
```

Q8 中 A 和 B 的操作数，10 进制的位数已经不是同侪了，运行以后的结果 ... Test Result 和 Real Result 都偏离 True Result。果然拥有 7 个十进制的位数（包括小数），单精度格式不好支持。

Q9: A = 44.4444，B = 68.3333，A - B = ?

```
A = 32'b0_10000100_01100011100011100010001
B = 32'b0_10000101_00010001010101010101010

32'b1_10000011_01111110001110010000110 Test Result -23.8889274
```

```
32'b1_10000011_01111110001110001111000 Real Result -23.8889007
                                       True Result -23.8889
```

结果再一次的显示，6 个十进制位数（包括小数）是单精度格式在加减操作中，Test Result 和 Real Result 是最接近 True Result 的。不过比较愕然的是，即使 Test Result 中最后几个尾数 和 Real Result 不同，但是它们取得的结果都和 True Result 差不多。

到目前为止我们对单精度浮点数的认识已经有一定的程度了，因此对于实现加减运算是绝对没有问题的。关于单精度格式有一定的承载能力，如果用 10 进制的尾数作为标准的话，我们可以得出一个结论，那就是拥有 6 个十进制位数（包括小数）的操作数，可以取得最接近 True Result 的加减运算结果。

1.8 浮点乘法和陷阱

什么是浮点数的乘法呢？我们先从数学的角度上去认识它：

$$5.0 \times 10^2 \times -2.2 \times 10^3 = (1)5.0 \times 10^2 \times (-1)2.2 \times 10^3$$
$$= (1 \times -1)\ 5.0 \times 2.2 \times 10^{2+3}$$
$$= (-1)11 \times 10^5$$
$$= -11 \times 10^5$$

在传统的数学上，浮点数的乘法可以分成 3 个部分，就是符号位运算（正负运算），数目运算（尾数运算）和价码运算。当我们把同样的概念带入 Verilog 的世界，同样的概念只能发挥一半的功能而已。如果我们一意孤行使用相同的概念，以 2 进制的方式来实现浮点乘法的话，结果必定天珠。这就是著名的浮点乘法陷阱。

在前面，笔者已经说过那一套用在我们认知的常识里，是绝对不适合用在 Verilog 的世界里。我们认识里，数字是以 10 进制为基础，浮点数不存在精度问题的问题。另一个比较明显的例子，就是 1.4 小节里笔者所讨论过的“实定移位”和“假定移位”。至于浮点乘法如何在两个不同的世界里产生违和感，以致掉入它的陷阱里!？接下来，就让我们慢慢去发现 ...

首先让我们先从数据的角度上看二进制运行浮点数乘法的运算？

```
1.110E+2 × -1.011E+3  = (1)1.110E+2 × (-1)1.011E+3
                      = (1)( -1) 1.110 × 1.011 E(2+3)
                      = (-1) 1.0011010 E+5
                      = -1.0011010 E+5
```

符号位 = 1 (-1) = -1	1110 1011 × —————— 1110 11100 000000 1110000 + —————— 尾数 10011010	价码 = 2 + 3 = 5

```
结果 = (符号位)尾数 E(价码)
     = (-1)1.0011010E(5)
     = -1.0011010E+5
```

对于浮点数按二进制方式来执行乘法运算，同样可以用传统数学的方式实现，亦即符号位，尾数和价码分 3 部分来运算。不过不同的是尾数在累加过程中，小数点没有被包含进来。在这里有学同学可能会问“既然尾数在累加过程中不包含小数点，那么在结果中怎样去定义小数点的存在？”嗯!这确实是重点问题。

在前面笔者已经讲过了，在二进制的运算中小数点的存在是“假定”，而不是书写形式上的“实定”。对于“假定”的小数点来说它是被“夹在某个范围之间”，作为前提条件我们需要某个“固定”的位宽。如上面的例子中，操作数 A 和 B 均为 4 位宽，那么这个“固定”的空间就是 A 和 B 的总和。最后“假定”的小数点就是“夹在最高位和最高第二位”之间。具体一点来说，假设 A 和 B 的乘法结果寄存在名为 Temp 的空间里，那么 Temp[7] 和 Temp[6] 之间就是假定的小数点。

读到这里读者估计已经认识到最基本“浮点乘法”在十进制和二进制之间的区别。更深一层的认识，接下来我们要实现单精度格式的浮点数乘法。为了预防一些同学发生失忆，我们在来复习一下单精度的格式：

```
[ 0][0111111][00000000000000000000000]  //默认空壳
[31][-30:23-][---------22:0----------]
[符][-价码--][---------尾数----------]
```

单精度格式可以分为 3 部分，亦即符号位[31]，价码[30~23]和尾数[22:0]。默认空壳中，除了符号位和尾数以外（全零），价码的值是 8'b01111111（在专业的用词中，8'b0111111 称为价码的偏移量）。在尾数中还包含一个“隐藏位”，在单精度格式中“隐藏位”是没有被表示出来的。

普通下，我们都会按照浮点数的乘法执行习惯，把符号位，价码和尾数分为 3 个部分来运算。结合我们对单精度浮点数的加减运算的经验，大致的步骤如下：

（一）操作数预处理：恢复单精度浮点数中的隐藏位；
（二）运算：符号位操作，价码运算操作，尾数运算操作独立运行；
（三）调整结果：将运算结果调整至适合浮点数源的格式；
（四）输出和格式化：从尾数中摘取前半部分，然后检测是否执行四舍五入。

乍看下，乘法运算的过程比起加减运算来得更简单，不过单是这样空口白谈感觉还是很抽象的，笔者几个简单的例子吧：

Q1: A = 2.5，B = 5，A × B = ?

```
A = 32'b0_10000000_01000000000000000000000
B = 32'b0_10000001_01000000000000000000000

操作数预处理：

A' = 32'b0_10000000_101000000000000000000000
B' = 32'b0_10000001_101000000000000000000000

运算：

sum.s     = A'.s ^ B'.s
          = 0 ^ 0
          = 0
```

```
sum.exp    = A'.exp + B'.Exp
           = 8'b10000000 + ( 8b'10000001 - 127 )
           = 128 + ( 129 - 127)
           = 128 + 2
           = 130
           = 0b10000010

sum.m= A'.m * B'.m
           = 24'b101000000000000000000000 * 24'b101000000000000000000000
          = 48'b011001000000000000000000000000000000000000000000

结果调整:
$ sum.m[47] is 0, while sum.m << 1

011001000000000000000000000000000000000000000000 before
110010000000000000000000000000000000000000000000  << 1
110010000000000000000000000000000000000000000000 after, sum = sum'

输出和格式化:
$ Take out sum'.m[46:24], if sum'.m[23] is 1  sum'.m[46:24] + 1

32'b0_10000010_10010000000000000000000 Test Result 12.5
32'b0_10000010_10010000000000000000000 Real Result 12.5
                                            True Result 12.5
```

步骤一：操作数预处理

首先是操作数预处理，乘法运算和加减运算不一样的是，乘法运算没有“价码对齐”这一个步骤，除了不用移动尾数以外，更不需要而外的空间来补偿尾数因位移使失去的部分。此外，还有一点的是，隐藏位的恢复也取得 1 位而已，Verilog HDL 语言的实现如代码 1.8.1 所示：

```
begin
    rA <= { A[31], A[30:23], 1'b1, A[22:0]};
    rB <= { B[31], B[30:23], 1'b1, B[22:0]};
end
```

代码 1.8.1 操作数预处理

步骤二：运算

下来的操作时 3 个部分——符号位，价码和尾数同时运算。符号位的运算可以用亦或取得；价码的运算也可以用简单的加减运算符取得；至于尾数的乘法运算，笔者为了很好说明，笔者先用默认的“*”运算符取得尾数 A 和 B 的相乘结果。Verilog HDL 语言的实现如代码 1.8.2 所示：

```
begin
    isSign <= A[31] ^ B[31];

    BDiff = rB[31:24] - 8'd127; // + (~8'd127 + 1)
    rExp <= rA[31:24] + BDiff; // A.Exp + B.Exp

    Temp <= rA[23:0] * rB[23:0];
end
```

代码 1.8.2 运算

步骤 3：结果调整

运算以后我们要调整结果，这一点和加减运算一样。因为尾数在运算以后，结果不一定是适合“浮点数源”的格式。至于“假定”的小数点是介于“最高位和最高第二位之间”。其外有一点不同的是，乘法运算在结果调整中，调整范围不必加减运算的大，举个例子：

QI:	QII:	QIII:	QIV:
4’b1111	4’b1000	4’b0100	4’b0100
4’b1111 x	4’b1000 x	4’b1111 x	4’b1000 x
8’b11100001 before	8’b01000000 before	8’b00111100 before	8’b00100000 before
8’b11100001 after	8’b10000000 << 1	8’b11110000 << 2	8’b10000000 << 2
	8’b10000000 after	8’b11110000 after, Exp - 1	8’b10000000 after, Exp - 1

我们假设操作数 A 和 B 均为 4 位宽，那么相乘结果最大的可能必然是 8 位宽，至于“假定”小数点必是 [7]~[6]之间。在上面 4 个例子中，笔者取出最有可能得到的 4 大结果。

一、首先是 QI 的结果，其中最高位为 1 所以不用调整；
二、QII 的结果中最高位值为 0，所以必须左移 1 位；
三、QIII 的结果中，[7:6]都为 0，所以必须左移 2 位，然后价码减 1；
四、QIV 的结果中，[7:6]双双还是为 0，所以必须左移 2 位，然后价码减 1。

从上面 4 个结果中，Q1 不用调整，价码理所当然不用更改，至于 Q2 的结果再调整中（左移 1 位），价码同样不用被更改。但是 Q3 和 Q4 的结果再调整中（左移 2 位），价码必须被调整为减 1。在此可能有很多同学会问 WHY！不要问 WHY 了，这是隐藏在乘法运算中的陷阱之一。操作数无论是 4 位宽还是 24 位宽，这个陷阱同样存在，所以读者们要多多醒目！如果要用 Verilog HDL 语言来实现的话，如代码 1.8.3 所示：

这里没看明白

```
begin
    if( Temp[47] == 1'b1 ) begin Temp <= Temp; end // do nothing
    else if( Temp[46] == 1'b1 ) begin Temp <= Temp << 1; end
    else if( Temp[45] == 1'b1 ) begin Temp <= Temp << 2; rExp <= rExp - 1'b1;end
end
```

代码 1.8.3 结果调整。

步骤 4：输出和格式化

在这个步骤中，我们同样要检查价码溢出或者尾数零值等错误信息以外，我们要四舍五入尾数的结果。价码溢出和尾数零值的检测过程，笔者就不多说了，基本上和前两个实验都一样。不过在执行四舍五入的时候，概念会有所不同。在输出之前，我们只能从尾数的运算结果中摘取一部分而已，假设这个运算空间为 48 位宽的 Temp，Temp[47]必定隐藏位，然而 Temp[46:24]就是我们要摘取的部分，如果 Temp[23]的值为 1 的话，那么我们必须为 Temp[46:24]加 1。

Verilog HDL 语言实现的过程如代码 1.8.4 所示：

```
begin
    if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end       // E Overflow
    else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // E Underflow
    else if( Temp[47:24] == 23'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
    else if( Temp[23] == 1'b1 ) rResult <= { isSign, rExp[7:0], Temp[46:24] + 1'b1 }; // okay with normalised
    else rResult <= { isSign, rExp[7:0], Temp[46:24] }; // okay without normalise
end
```

代码 1.8.4 输出和格式化。

==

到目前为止，如果读者只以为浮点乘法只有一个陷阱而已，那么读者就是大错特错了。第二个陷阱更隐蔽，更危险，它也是在许多浮点数乘法器设计上所犯的问题。笔者再举一个例子：

Q2: A = 2，B = 2， A×B = ?

先插入一段废话，尽管 Q2 是一个小学生的问题，但是在此刻它是一个无比的经典，读者要好好的铭记于心。至于这个连小学生都会的问题，到底会出现什么样的陷阱？让我们慢慢去发掘吧！（以下的过程中笔者把操作数预处理简化掉）

```
A = 32'b0_10000000_00000000000000000000000
B = 32'b0_10000000_00000000000000000000000

运算:

sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp = A.exp + B.exp
        = 0b10000000 + ( 0b10000000 - 127 )
        = 0b10000001
```

```
sum.m = A.m * B.m
       = 24'b100000000000000000000000 * 24'b100000000000000000000000
       = 48'b010000000000000000000000000000000000000000000000



结果调整:
$ sum.m[47] is 0, while sum.m << 1

48'b010000000000000000000000000000000000000000000000 before
48'b100000000000000000000000000000000000000000000000 << 1
48'b100000000000000000000000000000000000000000000000 after, sum = sum'

输出和格式化:
32'b0_10000001_00000000000000000000000 Test Result 4
32'b0_10000001_00000000000000000000000 Real Result 4
                                            True Result 4
```

从 Q2 的运算结果中，我们看到 2 * 2 也存在第一个陷阱。在结果调整中，如果尾数必须右移一位才能达到浮点数源的格式，按常理来说我们应该讲价码递减为 1。如果我们真的把价码递减，那么 True Result 就会和 Real Result 不同的结果。

Q3：A = 3.142，B = 3.142，A × B = ?

```
A = 32'b0_10000000_10010010001011010000111
B = 32'b0_10000000_10010010001011010000111

运算:
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp = A.exp + B.exp
         = 0b10000000 + ( 0b10000000 - 127 )
         = 0b10000001

sum.m = A.m * B.m
       = 24'b110010010001011010000111 * 24'b110010010001011010000111
       = 48'b100111011111010001100001111110010111101100110001

结果调整:
$ sum.m[47] is 1, while do nothing

输出和格式化:
$ sum.m[23] is 1, while sum.m[46:24] + 1


h m
```

```
1 0011101111101000110000 1 before
1 00111011111010001100010 +1
1 00111011111010001100010 after, sum = sum'

32'b0_10000001_00111011111010001100010 Test Result 4.9360818
32'b0_10000010_00111011111010001100010 Real Result 9.8721637
                                       True Result 9.872164
```

Q3 我们用和 Q2 一样的运算步骤，可是得出的乘法运算结果 Test Result 却和 Real Result 差十万八千里，此外 Real Result 和 True Result 的结果又差不多，因此我们更能肯定 Test Result 出现错误了。然后我们再比较一下 Test Result 和 Real Result 的二进制值，从中我们可以发现 Test Result 的价码比起 Real Result 的价码少了一位。这又是另一个浮点乘法的阴森陷阱！

为了解决这个问题，我们必须重新审视之前有关浮点乘法的运算步骤。关于之前浮点乘法概念，我们视符号运算，价码运算和尾数运算是 3 方可以独立运行的。这个观点在传统的数学角度上，是绝对没有问题，换之在 Verilog 里，尤其是二进制的世界是绝对有问题的。既然我们已经知道问题是发生在概念或者观点上，但是我们要如何解决问题呢？

问题源于概念切口自然直入概念，所以我们可以这样思考：“符号运算是一个独立的，价码运算和尾数运算又另一个独立的 ... ”因此，乘法运算再也不是 3 方法的，而是两方独立运行。借此我们可以利用浮点运算自身的微妙能力，我们知道浮点数的小数点是可以跳来跳去不是吗？举个例子：

$$5.1 \times 10^3 \times 2.2 \times 10^2 = 5.1 \times 10^3 \times 0.22 \times 10^3$$

1.101E+3 × 1.011 E+2 = 1.101E+3 × 0.1101 E+3

上面的例子中，笔者都是把操作数 B 的价码递增，然后小数点左移 1 位（二进制值右移一位），结果的基本值还是等价的。我们可以利用这个微妙的移位动作来解决 Q3 的陷阱问题。不过不是什么问题组（值操作数 A 和操作数 B）都可以使用此方法，唯有某个“特殊条件”下，才能使用这个方法。所以呢，我们必须建立某个逻辑来判断这个“特殊条件”。最后，我们必须重新整理一下之前的乘法步骤。

（一）操作数预处理：恢复单精度浮点数中的隐藏位；
（二）符号位运算：符号位运算操作独立运行；
（三）移位预处理：根据某个“特殊条件”，把操作数 B 的小数点左移 1 位，然后价码递增 1；
（四）价码和尾数运算：价码运算操作和尾数运算操作独立运行；
（五）调整结果：将运算结果调整至适合浮点数源的格式；
（六）输出和格式化：从尾数中摘取前半部分，然后检测是否执行四舍五入。

这个特殊条件有 2 个，笔者先说其中一个，那就是：“操作数组价码等价，但是双反尾数同不等于 0”。操作组是指操作数 A 和 B，如果它们之间的价码等价，然后尾数又不等于 0，那么移位预处理就可以执行了（注意在判断过程不包括隐藏位，但是在移位过程中却包括隐藏位）。如果用 Verilog HDL 语言来表示的话，代码如 1.8.5 所示：

```
if( rA[31:24] == rB[31:24] && ( rA[22:0] != 23'd0 & rB[22:0] != 23'd0 ) )
```

```
begin rB[31:24] <= rB[31:24] + 1'b1; rB[23:0] <= rB[23:0] >> 1; end
```

代码 1.8.5 移位预处理。

我们用新的运算步骤再一次运算 Q3 的问题看看：

Q3b：A = 3.142，B = 3.142，A × B = ?

```
A = 32'b0_10000000_10010010001011010000111
B = 32'b0_10000000_10010010001011010000111

符号位运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

移位预处理：
$ A.exp == B.exp && A.m and B.m not a zero ( no include hidden bit ),
  while B.m >> 1 and B.exp + 1 (include hidden bit )

B'.exp = 0b10000000 + 1 = 0b10000001

h m
1 10010010001011010000111 before
0 11001001000101101000011 >> 1  //无视尾数补偿
0 11001001000101101000011 after, B = B'

价码和尾数运算：
sum.exp = A.exp + B'.exp
         = 0b10000000 + ( 0b10000001 - 127 )
         = 0b10000010

sum.m = A.m * B.m
      = 24'b110010010001011010000111 * 24'b011001001000101101000011
      = 48'b010011101111110100011000010011000001100100101010101

结果调整：
$ sum.m[47] is 0, while sum.m << 1

h   m
0 10011101111110100011000010011000001100100101010101 before
1 00111011111101000110000100110000011001001010101010 << 1
1 00111011111101000110000100110000011001001010101010 after, sum = sum'

输出和格式化：
$ sum.m[23] is 0, while do nothing

32'b0_10000010_00111011111101000110001 Test Result 9.8721628
```

```
32'b0_10000010_00111011111010001100010 Real Result 9.8721637
                                       True Result 9.872164
```

当我们重新用不同的概念和步骤运行 Q3 的问题以后，最终求得的 Test Result 和 Real Result 差不多。不过操作数 B 在移位预处理的途中，笔者无视尾数的补偿，小小的 1 位尾数而已，用不着那么大惊小怪，有没有补偿它对结果没有多大影响。

在此，可能有些同学要求能不能把“特殊条件”的判断逻辑简化一下？

```
if( rA[31:24] == rB[31:24] && ( rA[22:0] != 23'd0 & rB[22:0] != 23'd0 ) )   // before  正版
if( rA[31:24] == rB[31:24] && rA[22:0] == rB[22:0] )                         // after   简化版
```

那些同学可能因由 Q3b 操作组中“操作数 A 和操作数 B 同样”的关系才会那么想 ... 事实上，那是万万不可的动作！不相信吗？我们再运算一下 Q2 的问题（2*2）就知道结果了。

Q2b：A = 2，B = 2，A × B = ?

```
A = 32'b0_10000000_00000000000000000000000
B = 32'b0_10000000_00000000000000000000000

浮点位运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

移位预处理：
$ A.exp == B.exp && A.m == B.m，while B.m << 1 and B.exp + 1

B'.exp = 0b10000000 + 1
       = 0b10000001

h m
1 00000000000000000000000 before
0 10000000000000000000000 >> 1
0 10000000000000000000000 after, B = B'

价码和尾数运算：
sum.exp = A.exp + B'.exp
        = 0b10000000 + ( 0b10000001 - 127 )
        = 0b10000010

sum.m = A.m * B'.m
      = 24'b100000000000000000000000 * 24'b010000000000000000000000
      = 48'b001000000000000000000000000000000000000000000000

结果调整：
```

```
$ sum.m[47] is 0, while sum.m << 2 and sum.exp - 1

sum'.exp = 0b10000010 - 1 = 0b10000001

h m
0 01000000000000000000000000000000000000000000000 before
1 00000000000000000000000000000000000000000000000 << 2
1 00000000000000000000000000000000000000000000000 after, sum = sum'

32'b0_0b10000001_00000000000000000000000 Test Result 4
32'b0_0b10000001_00000000000000000000000 Real Result 4
                                             True Result 4
```

同学得意的说：“笔者！看吧！特殊条件的判断逻辑简化以后 Q2 的结果还不是一样正确吗？”少得意了这位同学 ... 我们再运算另一个问题看看。

Q4：A = 273.757，B = 483.265，A × B = ?

```
A = 32'b0_10000111_00010001110000011100101
B = 32'b0_10000111_11100011010000111101100

符号位预算：
sum.s = A.s ^ B.s = 0

移位预处理：
$ A.exp == B.exp but A.m != B.m, while do nothing ... //简化判断条件无法成立

价码和尾数运算：=
sum.exp = A.exp + B.exp
         = 0b10000111 + ( 0b10000111 - 127 )
         = 0b10001111

sum.m = A.m * B.m
       = 24'b100010001110000011100101 *  24'b111100011010000111101100
       = 48'b100000010011001001001011001101000101100000011100

结果调整：
$ sum.m[47] is 1, while do nothing ...

输出和格式化：
32'b0_10001111_00000010011001001001011 Test Result 66148.5859375
32'b0_10010000_00000010011001001001011 Real result 132297.1718750
                                           True result 132297.176605
```

从 Q4 的结果看来，我们就可以得知结果 Test Result 已经大量偏移 Real Result 和 True Result 的

值。从 Test Result 和 Real Result 的区别中，我们非常清楚的看到它们之间价码相差 1。因此我们可以断定简化版的判断逻辑（特殊条件用）只能运行部分功能而已，实际上是抬不了场面。接下来，我们用回完整版的判断逻辑再一次运算 Q4 看看。

Q4b：A = 273.757，B = 483.265，A × B = ?

```
A = 32'b0_10000111_00010001110000011100101
B = 32'b0_10000111_11100011010000111101100

符号位预算：
sum.s = A.s ^ B.s = 0

移位预处理：
$ A.exp == B.exp and A.m != 0 and B.m ! = 0 （not include hidden bit）,
  while B.m << 1 ( include hiddent bit ) and B.exp + 1

B'.exp = 0b10000111 + 1 = 0b10001000

h m
1 11100011010000111101100 before
0 11110001101000011110110 >> 1
0 11110001101000011110110 after, B = B'

价码和尾数运算：
sum.exp = A.exp + B'.exp
         = 0b10000111 + ( 0b10001000 - 127 )
         = 0b10010000

sum.m = A.m * B.m
       = 24'b100010001110000011100101 *  24'b011110001101000011110110
       = 48'b010000000100110010010010110011010001011000000011110

结果调整：
$ sum.m[47] is 0, while sum.m << 1

h m
0 1000000100110010010010110011010001011000000011100 before
1 0000001001100100100101100110100010110000000111000 << 1
1 0000001001100100100101100110100010110000000111000 after, sum = sum'

输出和格式化：
32'b0_10010000_00000010011001001001011 Test Result 132297.1718750
32'b0_10010000_00000010011001001001011 Real Result 132297.1718750
                                          True Result 132297.176605
```

当我们用回正版的判断逻辑（特殊条件用）Q4 取得的结果 Test Result 已经和 Real Result 一样了。不过它们距离 True Result 还是有点精度上的差别。结论，我们可以证明简化版的判断逻辑只是鹌鹑而已，用回正版的东西才是王道。

==

在前面笔者声明过“特殊条件”有两个，其中一个是操作数组价码等价，但是双方的尾数都不等于 0 时。另一个“特殊条件”就是“操作组的符号位均不同”，移位预处理同样会被执行。。第二个“特殊条件”用 Verilog HDL 语言来表达更简单，如代码 1.8.5 所示：

```
if( rA[31:24] == rB[31:24] && ( rA[22:0] != 23'd0 & rB[22:0] != 23'd0 ) || rA[31] ^ rB[31] )
    begin rB[31:24] <= rB[31:24] + 1'b1; rB[23:0] <= rB[23:0] >> 1; end
```

代码 1.8.5 第二会引起移位预处理的特殊条件。

“哦 ... 纳尼！？ (⊙o⊙)”，读者别用这样的表情看笔者，笔者所说的话都是事实。不信吗？那好，我们再运算其他问题看看。

Q5 A = 6.3565，B = -0.0063，A × B = ?

```
A = 32'b0_10000001_10010110110100001110011
B = 32'b1_01110111_10011100111000000111011

价码运算：
sum.s = A.s ^ B.s= 0 ^ 1 = 1

移位预处理：
$ sum.s is 1,while B.m << 1 and B.exp + 1

B'.exp = 0b01110111 + 1 = 0b01111000


h m
1 10011100111000000111011 before
0 11001110011100000011101 >> 1
0 11001110011100000011101 after, B = B'

价码和尾数运算：
sum.exp = A.exp + B'.exp
         = 0b10000001 + ( 0b01111000 - 127 )
         = 129 + ( 120 - 127 )
         = 122
         = 0b01111010


sum.m = A.m * B'm
       = 24'b110010110110100001110011 * 24'b011001110011100000011101
```

```
      = 48’b0101001000000001110011100001010001111110100000111

结果调整:
$ sum.m[47] is 0, while sum.m << 1

h m
0 101001000000001110011100001010001111110100000111 before
1 010010000000011100111000010100011111101000001110 << 1
1 010010000000011100111000010100011111101000001110 after, sum = sum’

输出和格式化:
32’b1_01111010_01001000000011100111000 Test Result -0.0400459
32’b1_01111010_01001000000011100111001 Real Result -0.0400459
                                       True Result -0.04004595
```

读者看到了吧，当操作数 A 和 B 之前的符号位异或运算以后，如果是 1 那么移位预处理就要执行了。如 Q5 所示 A 和 B 的符号位预算结果是 1，经过移位预处理以后，最终求得的 Test Result 和 Real Result 差不多。

在 Verilog 的世界里，尤其是二进制实现乘法运算，这里那里到处都充满陷阱。在这个小节里，我们知道乘法运算又 2 个陷阱，第一是在结果调整中，另一个是移位预处理中。这两个陷阱是非常隐蔽的，这也使得浮点数乘法器设计的难度大大提升。笔者还记得在网上冲浪寻找资源的时候，笔者发现很多网站的内容都无视这个问题，此外相关的论文也显得更疲软了，笑~

笔者不知道著名的有关 Pentium 在浮点数运算的 BUG 是不是和这些陷阱有关系？但是笔者非常可定的是，要发现陷阱就要勇敢去踩陷阱，作为发现陷阱的带价就是付出一大半的 HP（勇者斗恶魔经典的精神）。

1.9 实验三：浮点数乘法器

在前一个小节里，笔者不但详细讨论过在浮点乘法会出现的隐蔽陷阱，此外还讨论浮点乘法的基本概念和过程。总结下，浮点乘法的大概步骤如下：

（一）操作数预处理：恢复单精度浮点数中的隐藏位；
（二）符号位运算：符号位运算操作独立运行；
（三）移位预处理：根据某个“特殊条件”，把操作数 B 的小数点左移 1 位，然后价码递增 1；
（四）价码和尾数运算：价码运算操作和尾数运算操作独立运行；
（五）调整结果：将运算结果调整至适合浮点数源的格式；
（六）输出和格式化：从尾数中摘取前半部分，然后检测是否执行四舍五入。

在实验三里我们就是按照这些步骤去设计一个浮点数乘法模块。

图 1.9.1 浮点数乘法模块图形。

图 1.9.1 是浮点数乘法模块的图形，至于信号之间的定义，基本上和之前的几个实验都一样。在此笔者就不罗嗦了，我们直接来看代码吧。

float_multi_module.v

```
1.    module float_multi_module
2.    (
3.        input CLK, RSTn,
4.        input [31:0]A,B,
5.        output [31:0]Result,
6.
7.        input Start_Sig,
8.        output [3:0]Done_Sig,
9.
10.       /****************/
11.
12.       output [32:0]SQ_rA,SQ_rB,
13.       output [47:0]SQ_Temp,
14.       output [9:0]SQ_rExp,SQ_BDiff
15.
16.   );
```

第 1~9 行是该模块的相关输入输出声明，至于第 12~14 行是仿真用的输出。

```
17.   /**************************************/
18.
19.       reg [3:0]i;
20.       reg [32:0]rA,rB;   // [32]Sign, [31:24]Exponent, [23]Hidden Bit, [22:0]Mantissa
21.       reg [47:0]Temp;     // [48]M'sign, [47:46]Hidden Bit, [45:23]M, [22:0]M'resolution
22.       reg [31:0]rResult;
23.       reg [9:0]rExp,BDiff;        //[9:8] Overflow or underflow check, [7:0] usuall exp.
24.       reg isSign;
25.       reg isOver;
26.       reg isUnder;
27.       reg isZero;
28.       reg isDone;
29.
30.       always @ ( posedge CLK or negedge RSTn )
31.           if( !RSTn )
32.               begin
33.                   i <= 4'd0;
34.                   rA <= 33'd0;
35.                   rB <= 33'd0;
36.                   Temp <= 48'd0;
37.                   rResult <= 32'd0;
38.                   rExp <= 10'd0;
39.                   BDiff <= 10'd0;
40.                   isOver <= 1'b0;
41.                   isUnder <= 1'b0;
42.                   isZero <= 1'b0;
43.                   isDone <= 1'b0;
44.               end
```

第 19~28 行是相关的寄存器声明，rA 和 rB 是用来寄存操作数 A 和 B 的隐藏位恢复结果，因此位宽声明为 32；Temp 是用来寄存相乘的结果；至于其他的寄存器，大意上和之前的几个实验都一样。

```
45.           else if( Start_Sig )
46.               case( i )
47.
48.                   0: // Initial and resulted out Signature
49.                   begin
50.                       rA <= { A[31], A[30:23], 1'b1, A[22:0]};
51.                       rB <= { B[31], B[30:23], 1'b1, B[22:0]};
52.                       isSign <= A[31] ^ B[31];
53.
```

```
54.                 isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0;
55.                 i <= i + 1'b1;
56.             end
57.
58.             1:
59.             begin
60.                if( rA[31:24] == rB[31:24] && ( rA[22:0] != 23'd0 & rB[22:0] != 23'd0 ) || rA[31] ^ rB[31] )
61.                    begin rB[31:24] <= rB[31:24] + 1'b1; rB[23:0] <= rB[23:0] >> 1; end
62.                i <= i + 1'b1;
63.             end
```

第 48~56 行是操作数预处理，除了从操作数 A 和 B 恢复隐藏位以外，还有清零相关的错误标志寄存器（第 54 行）。第 58~63 行是移位预处理，注意第 60 行是移位预处理的判断条件。在此笔者再强调一下，在判断图中隐藏位是不参与，反之在判断条件成立后，隐藏位在尾数 B 的移位却参与。

```
64.
65.             2: // if rExp[9..8] is 1, mean A.Exp small than B.Exp
66.                // while rExp[9..8] is 0, mean A.Exp large than B.Exp or same.
67.             begin
68.                 BDiff = rB[31:24] - 8'd127; // + (~8'd127 + 1)
69.                 rExp <= rA[31:24] + BDiff; // A.Exp + B.Exp
70.                 i <= i + 1'b1;
71.             end
72.
73.             3:
74.             begin
75.                 Temp <= rA[23:0] * rB[23:0];
76.                 i <= i + 1'b1;
77.             end
78.
79.             4: // Check M'hidden bit
80.             begin
81.                 if( Temp[47] == 1'b1 ) begin Temp <= Temp; end // do nothing
82.                 else if( Temp[46] == 1'b1 ) begin Temp <= Temp << 1; end
83.                 else if( Temp[45] == 1'b1 ) begin Temp <= Temp << 2; rExp <= rExp - 1'b1;end
84.
85.                 i <= i + 1'b1;
86.             end
```

第 65~71 行是价码运算，第 73~77 行是尾数运算，至于运算过程在 1.8 小节里已经解释过了，在此读者自己看着办。第 79~86 行是结果调整，比起加减操作，乘法操作在该步骤显得更简单，不过读者不要大意，这里是某个陷阱点。

```
87.
```

```
88.                     5: //error check and decide final result
89.                     begin
90.                         if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end       // E Overflow
91.                         else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // E Underflow
92.                         else if( Temp[47:24] == 23'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
93.                         else if( Temp[23] == 1'b1 ) rResult <= { isSign, rExp[7:0], Temp[46:24] + 1'b1 }; // okay with normalised
94.                         else rResult <= { isSign, rExp[7:0], Temp[46:24] }; // okay without normalise
95.                         i <= i + 1'b1;
96.                     end
97.
98.                     6:
99.                     begin isDone <= 1'b1; i <= i + 1'b1; end
100.
101.                    7:
102.                    begin isDone <= 1'b0; i <= 4'd0; end
103.
104.                endcase
105.
106.        /****************************************/
```

第 88~96 行是输出和格式化，内容基本上和之前的实验大同小异，除了第 92 行的“假定”小数点的位置不同以外，还有就是第 93~94 行之间尾数的位置。用简单的话来说，假定小数点夹在 Temp[47]~Temp[46]之间，然而输出用的尾数是 Temp[46 .. 24]，最后四舍五入的判断位置是 Temp[23]。

```
107.
108.        assign Done_Sig = { isOver, isUnder, isZero, isDone };
109.        assign Result = rResult;
110.
111.        /****************************************/
112.
113.          assign SQ_rA = rA;
114.          assign SQ_rB = rB;
115.          assign SQ_rExp = rExp;
116.          assign SQ_BDiff = BDiff;
117.          assign SQ_Temp = Temp;
118.
119.        /*****************************************/
120.
121.   endmodule
```

第 108~109 行是相关的输出驱动，第 113~117 行是仿真用的相关输出驱动。

float_multi_module.vt

```
1.    `timescale 1 ps/ 1 ps
2.    module float_multi_module_simulation();
3.
4.        reg CLK;
5.        reg RSTn;
6.
7.        reg [31:0] A;
8.        reg [31:0] B;
9.        reg Start_Sig;
10.
11.       wire [2:0]   Done_Sig;
12.       wire [31:0]   Result;
13.
14.       /**************************/
15.
16.       wire [9:0]   SQ_BDiff;
17.       wire [47:0]   SQ_Temp;
18.       wire [32:0]   SQ_rA;
19.       wire [32:0]   SQ_rB;
20.       wire [9:0]   SQ_rExp;
21.
22.       /**************************/
23.
24.       float_multi_module U1
25.        (
26.           .CLK(CLK),
27.             .RSTn(RSTn),
28.             .A(A),
29.             .B(B),
30.             .Result(Result),
31.             .Start_Sig(Start_Sig),
32.             .Done_Sig(Done_Sig),
33.             /************************/
34.             .SQ_BDiff(SQ_BDiff),
35.             .SQ_Temp(SQ_Temp),
36.             .SQ_rA(SQ_rA),
37.             .SQ_rB(SQ_rB),
38.             .SQ_rExp(SQ_rExp)
39.       );
40.
41.     /*****************************/
42.
43.     initial
```

```
44.     begin
45.         RSTn = 0; #10 RSTn = 1;
46.         CLK = 0; forever #5 CLK = ~CLK;
47.     end
48.
49.     /*******************************/
50.
51.     reg [5:0]i;
52.
53.     always @ (  posedge CLK or negedge RSTn )
54.         if( !RSTn )
55.             begin
56.               A <= 32'd0;
57.               B <= 32'd0;
58.               Start_Sig <= 1'b0;
59.               i <= 6'd0;
60.           end
61.       else
62.           case( i )
63.
64.                   0: //A=2.5  , B=7.4, A+B = ?
65.                   if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
66.                   else begin A <= 32'b01000000001000000000000000000000; B <= 32'b01000000101000000000000000000000; Start_Sig <= 1'b1; end
67.
68.                 1:
69.                 begin
70.                     $display("A = %b and rA = %b", A,SQ_rA);
71.                     $display("B = %b and rB = %b", B,SQ_rB);
72.                     $display("A.m * B.m = %b", SQ_Temp);
73.                     $display("Result = %b", Result);
74.                     i <= i + 1'b1;
75.                 end
76.
77.                 2: //3.142 * 3.142
78.                 if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
79.                 else begin A <= 32'b01000000010010010001011010000111; B <= 32'b01000000010010010001011010000111; Start_Sig <= 1'b1; end
80.
81.                 3:
82.                 begin
83.                     $display("A = %b and rA = %b", A,SQ_rA);
84.                     $display("B = %b and rB = %b", B,SQ_rB);
85.                     $display("A.m * B.m = %b", SQ_Temp);
86.                     $display("Result = %b", Result);
87.                     i <= i + 1'b1;
88.                 end
```

```
89.
90.     4: //12.662 * 4.903
91.     if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
92.     else begin A <= 32'b01000001010010101001011110001101; B <= 32'b01000000100111001110010101100000; Start_Sig <= 1'b1; end
93.
94.     5:
95.     begin
96.         $display("A = %b and rA = %b", A,SQ_rA);
97.         $display("B = %b and rB = %b", B,SQ_rB);
98.         $display("A.m * B.m = %b", SQ_Temp);
99.         $display("Result = %b", Result);
100.        i <= i + 1'b1;
101.    end
102.
103.    6: // 2 * 2
104.    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
105.    else begin A <= 32'b01000000000000000000000000000000; B <= 32'b01000000000000000000000000000000; Start_Sig <= 1'b1; end
106.
107.    7:
108.    begin
109.        $display("A = %b and rA = %b", A,SQ_rA);
110.        $display("B = %b and rB = %b", B,SQ_rB);
111.        $display("A.m * B.m = %b", SQ_Temp);
112.        $display("Result = %b", Result);
113.        i <= i + 1'b1;
114.    end
115.
116.    8: // 3.142 * 2
117.    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
118.    else begin A <= 32'b01000000010010010001011010000111; B <= 32'b01000000000000000000000000000000; Start_Sig <= 1'b1; end
119.
120.    9:
121.    begin
122.        $display("A = %b and rA = %b", A,SQ_rA);
123.        $display("B = %b and rB = %b", B,SQ_rB);
124.        $display("A.m * B.m = %b", SQ_Temp);
125.        $display("Result = %b", Result);
126.        i <= i + 1'b1;
127.    end
128.
129.    10: // 73.767 * 83.266
130.    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
131.    else begin A <= 32'b01000010100100111000100010110100; B <= 32'b01000010101001101000100000110001; Start_Sig <= 1'b1; end
132.
133.    11:
```

```
134.            begin
135.                $display("A = %b and rA = %b", A,SQ_rA);
136.                $display("B = %b and rB = %b", B,SQ_rB);
137.                $display("A.m * B.m = %b", SQ_Temp);
138.                $display("Result = %b", Result);
139.                i <= i + 1'b1;
140.            end
141.
142.            12: // 83.266 * 83.266
143.            if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
144.            else begin A <= 32'b01000010101001101000100000110001; B <= 32'b01000010101001101000100000110001; Start_Sig <= 1'b1; end
145.
146.            13:
147.            begin
148.                $display("A = %b and rA = %b", A,SQ_rA);
149.                $display("B = %b and rB = %b", B,SQ_rB);
150.                $display("A.m * B.m = %b", SQ_Temp);
151.                $display("Result = %b", Result);
152.                i <= i + 1'b1;
153.            end
154.
155.            14: // 1024 * 256
156.            if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
157.            else begin A <= 32'b01000100100000000000000000000000; B <= 32'b01000011100000000000000000000000; Start_Sig <= 1'b1; end
158.
159.            15:
160.            begin
161.                $display("A = %b and rA = %b", A,SQ_rA);
162.                $display("B = %b and rB = %b", B,SQ_rB);
163.                $display("A.m * B.m = %b", SQ_Temp);
164.                $display("Result = %b", Result);
165.                i <= i + 1'b1;
166.            end
167.
168.            16: // 33.116 * 16.558
169.            if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
170.            else begin A <= 32'b01000010000001000111011011001001; B <= 32'b01000001100001000111011011001001; Start_Sig <= 1'b1; end
171.
172.            17:
173.            begin
174.                $display("A = %b and rA = %b", A,SQ_rA);
175.                $display("B = %b and rB = %b", B,SQ_rB);
176.                $display("A.m * B.m = %b", SQ_Temp);
177.                $display("Result = %b", Result);
178.                i <= i + 1'b1;
```

```
179.                    end
180.
181.                    18: // 0.00416 * 0.00014
182.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
183.                    else begin A <= 32'b00111011100010000101000010011100; B <= 32'b00111001000100101100110011110111; Start_Sig <= 1'b1; end
184.
185.                    19:
186.                    begin
187.                        $display("A = %b and rA = %b", A,SQ_rA);
188.                        $display("B = %b and rB = %b", B,SQ_rB);
189.                        $display("A.m * B.m = %b", SQ_Temp);
190.                        $display("Result = %b", Result);
191.                        i <= i + 1'b1;
192.                    end
193.
194.                    20: // 0.125 * 0.125
195.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
196.                    else begin A <= 32'b00111110000000000000000000000000; B <= 32'b00111110000000000000000000000000; Start_Sig <= 1'b1; end
197.
198.                    21:
199.                    begin
200.                        $display("A = %b and rA = %b", A,SQ_rA);
201.                        $display("B = %b and rB = %b", B,SQ_rB);
202.                        $display("A.m * B.m = %b", SQ_Temp);
203.                        $display("Result = %b", Result);
204.                        i <= i + 1'b1;
205.                    end
206.
207.                    22: // 0.0868 * 0.0868
208.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
209.                    else begin A <= 32'b00111101101100011100010000110011; B <= 32'b00111101101100011100010000110011; Start_Sig <= 1'b1; end
210.
211.                    23:
212.                    begin
213.                        $display("A = %b and rA = %b", A,SQ_rA);
214.                        $display("B = %b and rB = %b", B,SQ_rB);
215.                        $display("A.m * B.m = %b", SQ_Temp);
216.                        $display("Result = %b", Result);
217.                        i <= i + 1'b1;
218.                    end
219.
220.                    24: // 0.0868 * 0.01085
221.                    if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
222.                    else begin A <= 32'b00111101101100011100010000110011; B <= 32'b00111100000110001110001000011001 1; Start_Sig <= 1'b1; end
223.
```

```
224.                25:
225.                begin
226.                    $display("A = %b and rA = %b", A,SQ_rA);
227.                    $display("B = %b and rB = %b", B,SQ_rB);
228.                    $display("A.m * B.m = %b", SQ_Temp);
229.                    $display("Result = %b", Result);
230.                    i <= i + 1'b1;
231.                end
232.
233.                26: // 0.0078125 * 0.0057488
234.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
235.                else begin A <= 32'b00111100000000000000000000000000; B <= 32'b00111011101111000110000011010001; Start_Sig <= 1'b1; end
236.
237.                27:
238.                begin
239.                    $display("A = %b and rA = %b", A,SQ_rA);
240.                    $display("B = %b and rB = %b", B,SQ_rB);
241.                    $display("A.m * B.m = %b", SQ_Temp);
242.                    $display("Result = %b", Result);
243.                    i <= i + 1'b1;
244.                end
245.
246.                  28: // 3.1828 * -0.0063
247.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
248.                else begin A <= 32'b01000000010010111011001011111111; B <= 32'b10111011110011100111000000111011; Start_Sig <= 1'b1; end
249.
250.                29:
251.                begin
252.                    $display("A = %b and rA = %b", A,SQ_rA);
253.                    $display("B = %b and rB = %b", B,SQ_rB);
254.                    $display("A.m * B.m = %b", SQ_Temp);
255.                    $display("Result = %b", Result);
256.                    i <= i + 1'b1;
257.                end
258.
259.                30: // 6.631 * 0.25
260.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
261.                else begin A <= 32'b01000000110101000011000100100111; B <= 32'b00111110100000000000000000000000; Start_Sig <= 1'b1; end
262.
263.                31:
264.                begin
265.                    $display("A = %b and rA = %b", A,SQ_rA);
266.                    $display("B = %b and rB = %b", B,SQ_rB);
267.                    $display("A.m * B.m = %b", SQ_Temp);
268.                    $display("Result = %b", Result);
```

```
269.                    i <= i + 1'b1;
270.                end
271.
272.                32: // 6.3565 * -0.0063
273.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
274.                else begin A <= 32'b01000000110010110110100001110011; B <= 32'b10111011110011100111000000111011; Start_Sig <= 1'b1; end
275.
276.                33:
277.                begin
278.                    $display("A = %b and rA = %b", A,SQ_rA);
279.                    $display("B = %b and rB = %b", B,SQ_rB);
280.                    $display("A.m * B.m = %b", SQ_Temp);
281.                    $display("Result = %b", Result);
282.                    i <= i + 1'b1;
283.                end
284.
285.                34: // 273.757 * 483.265
286.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
287.                else begin A <= 32'b01000011100010001110000011100101; B <= 32'b01000011111100011010000111101100; Start_Sig <= 1'b1; end
288.
289.                35:
290.                begin
291.                    $display("A = %b and rA = %b", A,SQ_rA);
292.                    $display("B = %b and rB = %b", B,SQ_rB);
293.                    $display("A.m * B.m = %b", SQ_Temp);
294.                    $display("Result = %b", Result);
295.                    i <= i + 1'b1;
296.                end
297.
298.                36: // 3.1828 * 2
299.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
300.                else begin A <= 32'b01000000010010111011001011111111; B <= 32'b01000000000000000000000000000000; Start_Sig <= 1'b1; end
301.
302.                37:
303.                begin
304.                    $display("A = %b and rA = %b", A,SQ_rA);
305.                    $display("B = %b and rB = %b", B,SQ_rB);
306.                    $display("A.m * B.m = %b", SQ_Temp);
307.                    $display("Result = %b", Result);
308.                    i <= i + 1'b1;
309.                end
310.
311.                48:
312.                i <= i;
313.
```

```
314.                    endcase
315.
316.  endmodule
```

读者千万不要被 float_multi_module.vt 激励文本的长度被吓到，笔者在验证浮点乘法模块的时候，不知觉中就写到那么长了，撤了有觉得可惜，所以就保留了下来。激励文本的风格没有多大变化，不过类似第 69-75 行的步骤会常常出现，主要是打印恢复以后的操作数 rA 和 rB，随之打印相乘的结果，最后打印单精度格式的结果。

仿真结果：

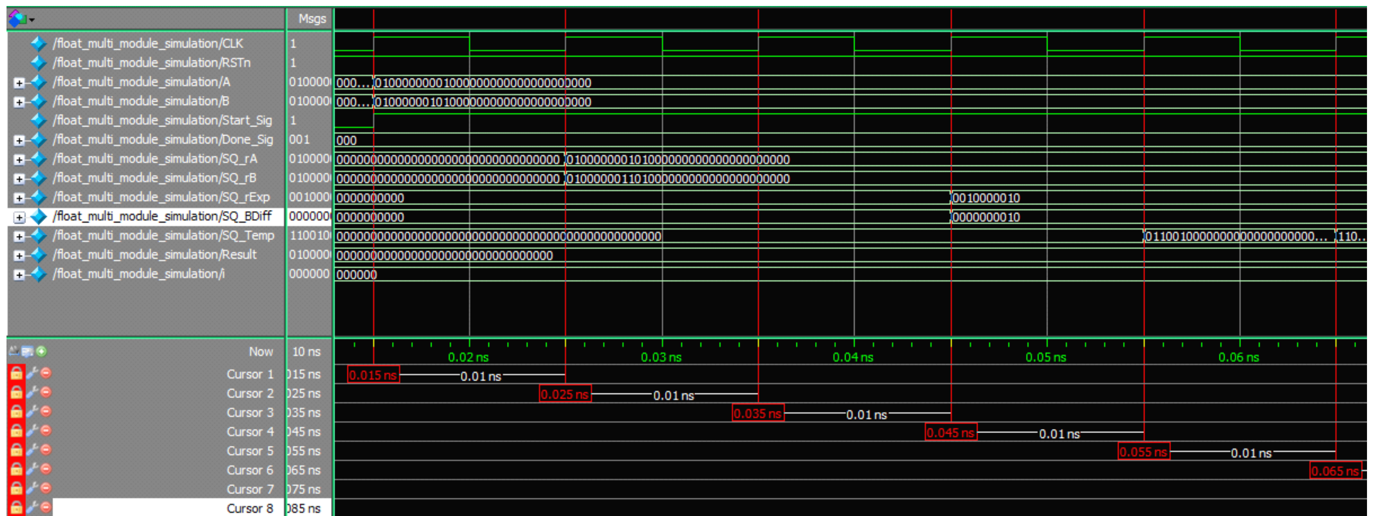

图 1.9.2 仿真过程一。

图 1.9.2 中 C1 是指向乘法模块的启动开始。在 C2 的时候是操作数预处理和符号位运算，该模块寄存并且恢复操作数 A 和 B，所以在 C2 的未来输出恢复以后的 rA 和 rB，与此同时也取得符号位结果。在 C3 的时候是移位预处理，此时特殊条件不成立，因此操作数 rB 没有发生移位预处理。在 C4 的时候是价码运算，所以在 C4 的未来输出价码结果。在 C5 的时候是尾数运算，rA 和 rB 的尾数决定相乘，所以在 C5 的未来输出相乘结果。

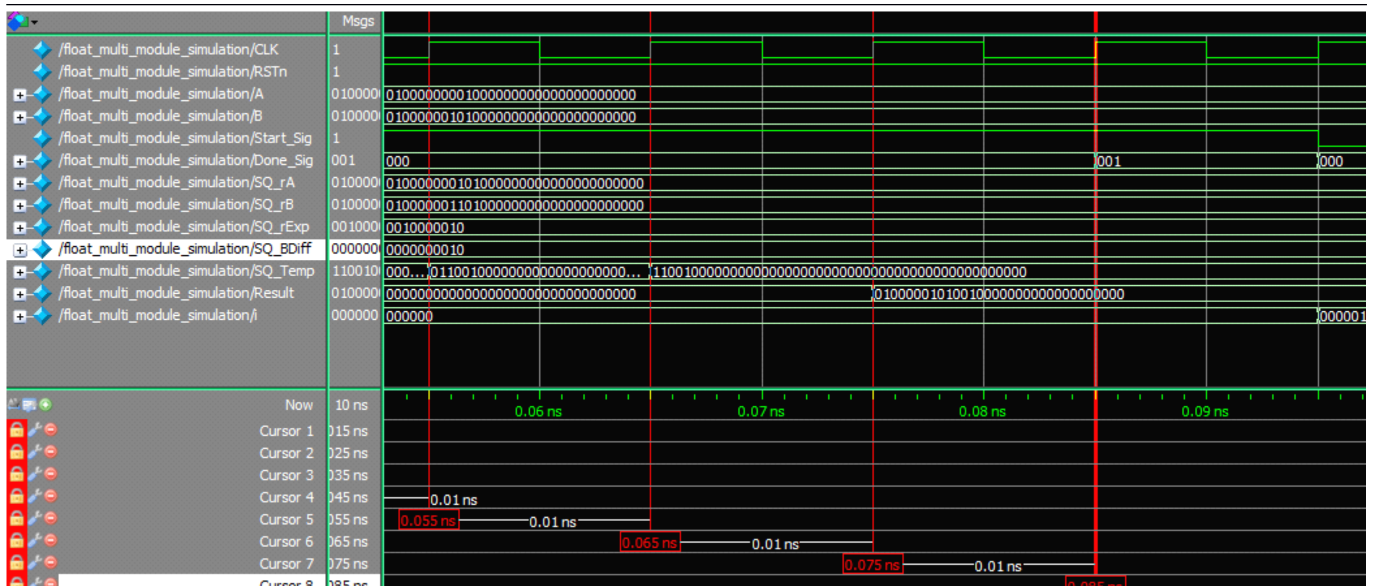

图 1.9.3 仿真过程 2。

在 C6 的时候是结果调整，此时 Temp[47]的过去值为 0，所以在 C6 的未来输出 Temp 右移一位以后的结果。在 C7 的时候是输出和格式化，各种错误反馈没有发生，最后在 C7 的未来输出单精度格式的结果。以上的仿真过程是没有处罚特殊条件的情形，接下来我们来看看相反情形的仿真过程。

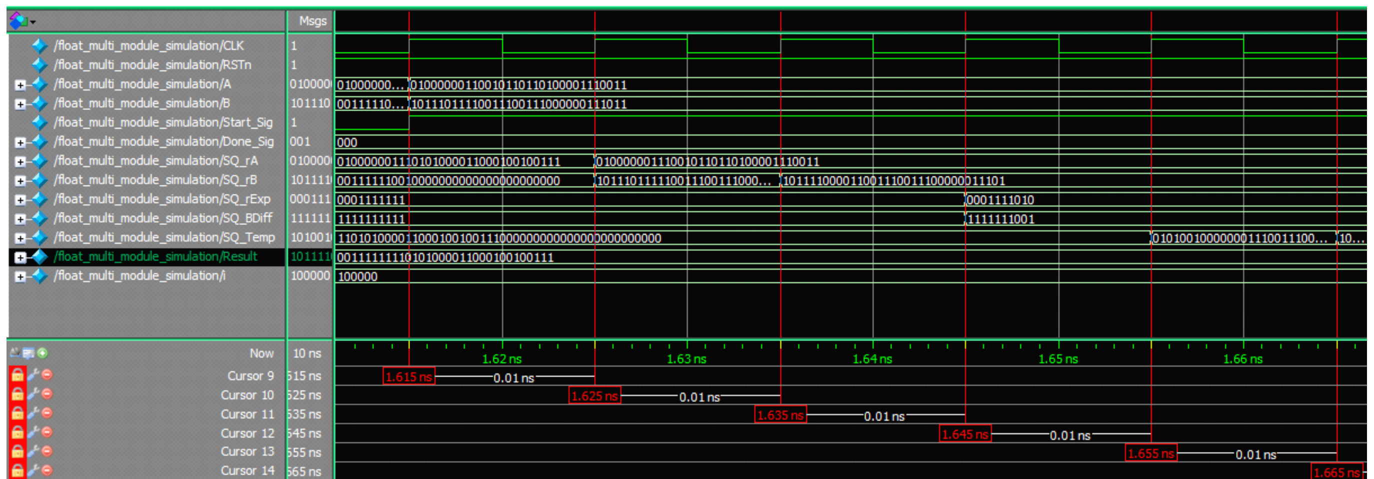

图 1.9.4 仿真过程 3。

图 1.9.4 是测试操作数组符号位为不一样的情况，亦即特殊条件之一的情况。C9 指向该模块使能的开始。C10 是操作数预处理和符号位运算，所以在 C10 的未来输出操作数预处理以后的结果，同时也取得符号位运算的结果。C11 是移位预处理，这时候特殊条件成立了，所以在 C11 的未来输出右移一位和价码加 1——操作数 B 的结果。

C12 是价码运算，所以在 C12 的未来输出运算以后的价码结果。C13 是尾数运算，所以在 C13 的未来输出相乘结果。

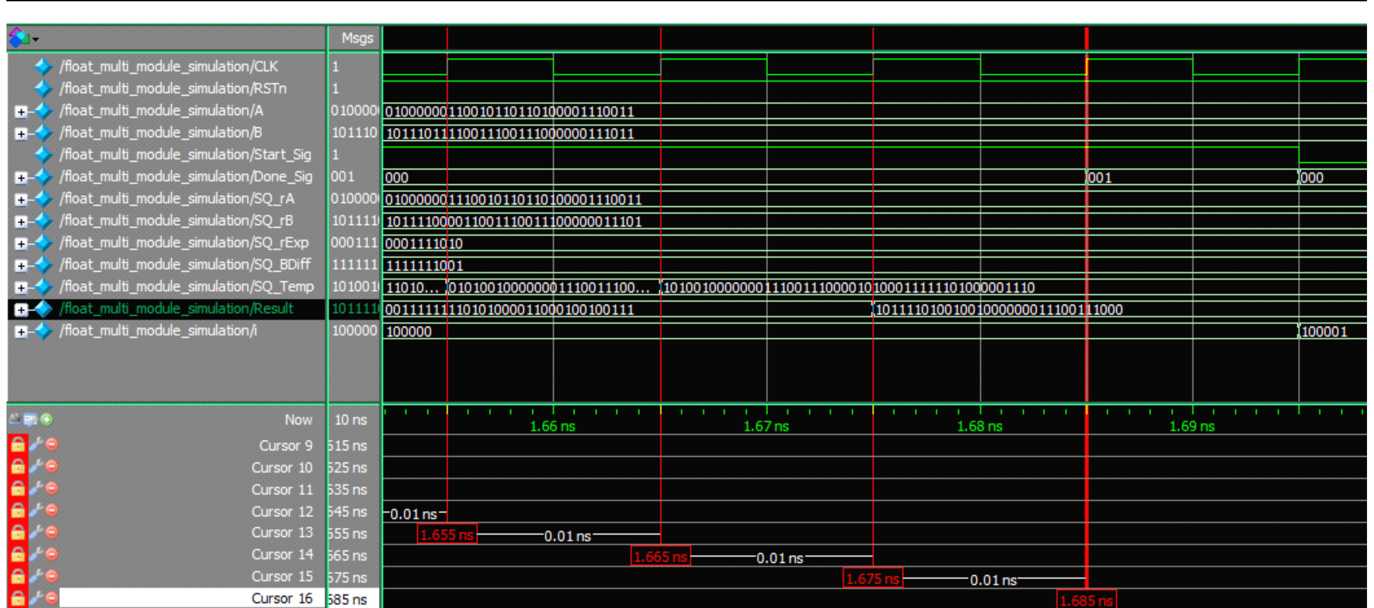

图 1.9.5 仿真过程 4。

在 C14 的时候是结果调整，根据 Temp 的过去值 Temp[47]为 0，所以在 C14 的未来输出右移一位后的 Temp。C15 是输出和格式化，任何错误反馈没有发生，所以在 C15 的未来输出单精度格式的结果。有发生移位预处理和无发生移位预处理是根据特殊状况 /条件而定，不过在运算过程中变化不是很大。

这个实验的激励过程好长好长，所以笔者就不将仿真过程一一贴出来，读者自己看着办吧，作为补偿笔者很乐意将所以验证结果贴处理。

Q1: A = 2.5，B = 5，A×B = ?

```
A = 32'b0_10000000_01000000000000000000000
B = 32'b0_10000001_01000000000000000000000

32'b0_10000010_10010000000000000000000 Test Result 12.5
32'b0_10000010_10010000000000000000000 Real Result 12.5
                                       True Result 12.5
```

Q1 这题很简单，没有什么好说的。

Q2: A = 3.142，B = 3.142，A×B = ?

```
A = 32'b0_10000000_10010010001011010000111
B = 32'b0_10000000_10010010001011010000111

32'b0_10000001_00111011111010001100010 Test Result 9.8721637
32'b0_10000001_00111011111010001100010 Real Result 9.8721637
                                       True Result 9.872164
```

对于十进制操作组的在 4 位以上的时候（包括小数），单精度浮点数的相乘结果显得有点勉强了。此外，Q2 是特殊条件之一。

Q3：A = 12.662，B = 4.903，A×B = ?

```
A = 32’b0_10000010_10010101001011110001101
B = 32’b0_10000001_00111001110010101100000


32’b0_10000100_11110000101001110111111 Test Result 62.0817832
32’b0_10000100_11110000101001111000000 Real Result 62.0817871
                                      True Result 62.081786
```

十进制操作组位数不一致，一个是 5 位（包含小数），另一位是 4 位（包含小数）得到的单精度结果还是可以接受。

Q4：A = 2，B = 2，A×B = ?

```
A = 32’b0_10000000_00000000000000000000000
B = 32’b0_10000000_00000000000000000000000


32’b0_10000001_00000000000000000000000 Test Result 4
32’b0_10000001_00000000000000000000000 Real Result 4
                                      True Result 4
```

Q5：A = 73.676，B = 83.266，A×B = ?

```
A = 32’b0_10000101_00100111000100010110100
B = 32’b0_10000101_01001101000100000110001


32’b0_10001011_01111111111001001000011 Test Result 6142.2827148
32’b0_10001011_01111111111001001000100 Real Result 6142.2832031
                                      True Result 6142.283022
```

对于十进制的操作数拥有 5 位以上（包括小数），单精度格式在精度方面已经显得有点承受不了了。此外，Q5 是特殊条件之一。

Q6：A = 83.266，B = 83.266，A×B = ?

```
A = 32’b0_10000101_01001101000100000110001
B = 32’b0_10000101_01001101000100000110001


32’b0_10001011_10110001010100111001111 Test Result 6933.2260742
32’b0_10001011_10110001010100111010000 Real Result 6933.2265625
```

```
                                            True Result 6933.226756
```

Q6 也是特殊条件之一。

Q7：A = 1024，B = 256，A×B = ?

```
A = 32'b0_10001001_00000000000000000000000
B = 32'b0_10000111_00000000000000000000000

32'b0_10010001_00000000000000000000000 Test Result 262144
32'b0_10010001_00000000000000000000000 Real Result 262144
                                            True Result 262144
```

Q8：A = 33.116，B = 16.558，A×B = ?

```
A = 32'b0_10000100_00001000111011011001001
B = 32'b0_10000011_00001000111011011001001

32'b0_10001000_00010010001010101101101 Test Result 548.3347778
32'b0_10001000_00010010001010101101100 Real Result 548.3347167
                                            True Result 548.334728
```

Q8 也是特殊条件之一。

Q9：A = 0.00416，B = 0.00014，A×B = ?

```
A = 32'b0_01110111_00010000101000010011100
B = 32'b0_01110010_00100101100110011110111

32'b0_01101010_00111000101011000111001 Test Result 0.0000005000
32'b0_01101010_00111000101011000111001 Real Result 0.0000005000
                                            True Result 0.0000005824
```

对于有 5 位十进制的小数来说，得到的单精度结果，精度已经越来越偏离了，或者说已经超过单精度格式可以承受的程度。

Q10：A = 0.125，B = 0.125，A×B = ?

```
A = 32'b0_01111100_00000000000000000000000
B = 32'b0_01111100_00000000000000000000000

32'b0_01111001_00000000000000000000000 Test Result 0.015625
32'b0_01111001_00000000000000000000000 Real Result 0.015625
                                            True Result 0.015625
```

Q11: A = 0.0868，B = 0.0868，A×B = ?

```
A = 32'b0_01111011_01100011100010000110011
B = 32'b0_01111011_01100011100010000110011

32'b0_01110111_11101101110000111001000 Test Result 0.0009417
32'b0_01110111_11101101110000111001001 Real Result 0.0009417
                                       True Result 0.00094178
```

仅有 4 位的十进制小数，得到的结果还可以接受。Q11 也是特殊条件之一。

Q12: A = 0.0078125，B = 0.0057488，A×B = ?

```
A = 32'b0_01111000_00000000000000000000000
B = 32'b0_01110111_01111000110000011010001

32'b0_01110000_01111000110000011010001 Test Result 0.0000449
32'b0_01110000_01111000110000001101110 Real Result 0.0000449
                                       True Result 0.0000449125
```

Q12 是非常挑剔的操作组，得到的结果 无论是 Real Result 还是 Test Result，它们都已经是最大极限了。

Q13： A = 3.1828，B = -0.0063，A×B = ?

```
A = 32'b0_10000000_10010111011001011111111
B = 32'b1_01110111_10011100111000000111011

32'b1_01111001_01001000100001101010110 Test Result -0.0200516
32'b1_01111001_01001000100001101010110 Real Result -0.0200516
                                       True Result -0.02005164
```

Q13 是特殊条件之一。

Q14: A = 6.631，B = 0.25，A×B = ?

```
A = 32'b0_10000001_10101000011000100100111
B = 32'b0_01111101_00000000000000000000000

32'b0_01111111_10101000011000100100111 Test Result 1.6577500
32'b0_01111111_10101000011000100100111 Real Result 1.6577500
                                       True Result 1.65775
```

Q15: A = 6.3565，B = -0.0063，A×B = ?

```
A = 32'b0_10000001_10010110110100001110011
B = 32'b1_01110111_10011100111000000111011


32'b1_01111010_01001000000011100111000 Test Result -0.0400459
32'b1_01111010_01001000000011100111001 Real Result -0.0400459
                                       True Result -0.04004595
```

Q16: A = 273.757，B = 483.265，A×B = ?

```
A = 32'b0_10000111_00010001110000011100101
B = 32'b0_10000111_11100011010000111101100


32'b0_10010000_00000010011001001001011 Test Result 132297.1718750
32'b0_01110000_01111000110000001101110 Real Result 132297.1718750
                                       True Result 132297.176605
```

Q16 同样也是挑剔的操作数，操作组都拥有 6 位十进制（包括小数），得到的结果，精度偏离的程度只能用掉眼泪来形容。Q16 是特殊条件之一。

Q17: A = 3.1828，B = 2，A×B = ?

```
A = 32'b0_10000000_10010111011001011111111
B = 32'b0_10000000_00000000000000000000000


32'b0_10000001_10010111011001011111111 Test Result 6.3656001
32'b0_10000001_10010111011001011111111 Real Result 6.3656001
                                       True Result 6.3656
```

Q17 没有什么特别，操作组价码同是，但是其中一个操作数的尾数（不包含隐藏位）是全零，所以不引起移位预处理。最后得到的 Test Result 不但接近 Real Result，而且也和 True Result 的偏度不大。

==

呼，先松一口气 ... 在 17 组的操作组中，我们都可以晓得所有 Test Result 都非常接近 Real Result 和 True Result，不过根据操作数的十进制位数，精度也会有所不同。在没有包括小数的情况下，十进制位数多少都不会严重影响结果，反之在包含小数的情况下，唯有 3 位十进制小数才是单精度格式可以发挥最佳的状态。

最后我们可以简单的总结一下，浮点数乘法器在理论或者概念上，它显得非常容易设，这是许多人最容易大意的现象。实际上浮点数乘法器存在 2 个隐蔽的陷阱，稍微不注意问题就大条了。因此，隐蔽的陷阱结果提升了设计的难度。最后说几句讽刺的话，很多参考书都不把这些问题（浮点乘法的陷阱）考虑进去，可见参考书的态度是非常随便。

1.10 浮点除法和精度流失

浮点数除法运算比起，加减和相乘运算可以说是最简单的一个。浮点除法在数学的形式上和浮点乘法的运算过程差不多，不过是反着方向而已。笔者举个简单的例子：

$12\ x10^4 / -3\ x10^2 = (1)12\ x10^4\ \ /\ (-1)3\ x10^2$
$= (1)(-1)\ 12/3\ x10^{4-2}$
$= (-1)\ 4\ x10^2$
$= -4\ x10^2$

浮点除法可以把 符号运算，价码运算和尾数运算分开 3 个独立运算来执行。在此有些同学可能会担心浮点除法和浮点乘法那样存在陷阱 ... 嗯，实际上浮点除法也存在陷阱，不过比起浮点乘法的陷阱，阴险度倒是低了不少。那么浮点除法的陷阱是什么，笔者先卖个关子，在往后的讨论中，那时候笔者再说。

不过在执行除法运算的时候，我们常常会碰上典型的现象？我们来看这样一个例子：

$A = 24\ x10^3,\ B = 30\ x10^4$

$24\ x10^3 / 30\ x10^4 = 24\ x10^3 / 3\ x10^5$
$= 24\ /\ 3\ x10^{3-5}$
$= 8\ x10^{-2}$
$= 0.08$

$24\ x10^3 / 30\ x10^4 = 24\ x10^3 / 0.3\ x10^6$
$= 24\ /\ 0.3\ x10^{3-6}$
$= 80\ x10^{-3}$
$= 0.08$

$24\ x10^3 / 30\ x10^4 = 24\ x10^3 / 0.03\ x10^7$
$= 24\ /\ 0.03\ x10^{3-7}$
$= 800\ x10^{-4}$
$= 0.08$

从上面的例子中，如果操作数 A 与 B 要相除，其中操作数 B 的尾数大于操作数 A 的尾数，如果真的发生相除的话，那么结果一定会产生余数。对于浮点除法来说，相除以后的余数如果越大，那么商数的精度就会下降。因此在浮点除法里，操作数 A 的尾数永远必须必操作数 B 的尾数大。所以，我们会利用小数点的移位，将操作数 A 的尾数变大。

那么问题来了，在 A 的尾数变大其中，我们没有一个规定尾数要变多大？浮点数左移 1 位，2 位还是 3 位呢？。对于在书写形式上的十进制来说，操作数 A 和 B 是没有固定的尾数，因此小数点位移的范围是非常的大。不过相反的，在二进制的世界里，浮点数可以套入在某个格式中，就以单精度格式为话题。

我们知道单精度格式的尾数，包括隐藏位一共是 24 位，因此只要把操作数 A 的尾数左移 24 位，无论是什么样的操作组，我们都可以断定操作数 A 的尾数一定大过操作数 B 的尾数。笔者举个具体一点的例子：

```
A = 24'b100000000000000000000001
B = 24'b100000000000000000000000

A/B = Quotient = 1'b1, Reminder = 1'b1
```

在上述的例子中，操作数 A 比起操作数 B 大，在 A 和 B 相除以后，我们会得到商数为 1，余数为 1 的结果。对于浮点数来说，余数的存在某个程度上就是精度流失的现象。因此我们必须把 A 变大，预防余数的产生。

```
A = 48'b100000000000000000000001000000000000000000000000E-24
B = 48'b000000000000000000000000100000000000000000000000E+0

A/B = Quotient = 25'b1000000000000000000000010, Reminder  = 0
```

途中，我们把 A 左移 24 位然后和 B 相除，相除以后取得商数为 25'b1000000000000000000000010 余数为 0 的结果。在此我们的问题不是要怎样处理 A 移位以后的价码，而我们主要的任务就是竟可能消除余数而已。

接下来我们来看一个更具体的例子：

```
Q1: A = 10, B = 3, A / B = ?
```

```
A = 32'b0_10000010_01000000000000000000000
B = 32'b0_10000000_10000000000000000000000

操作数预处理：
A = 33'b0_10000010_101000000000000000000000
B = 32'b0_10000000_110000000000000000000000

运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp = A.exp - B.exp
         = 0b10000010 - ( 0b10000000 - 127 )
         = 0b10000010 - 1
         = 0b10000001

sum.m = A.m / B.m
       = 24'b101000000000000000000000 / 24'b110000000000000000000000
```

```
      = 0, Reminder is 1101010101010101 ...
输出和格式化：
$ 由于求得的商值为 0，所以结果反馈尾数零值的错误。
```

在 Q1 的运算中，由于操作数 A 的尾数小于操作数 B 的尾数，所以相除之后所求的商数为 0，因此也造成尾数零值的错误发生，为此我们需要把操作数的尾数加大，我们把 A 的尾数加大一点点再来运算 Q1 看看：

Q1b: A = 10，B = 3，A / B = ?

```
A = 32'b0_10000010_01000000000000000000000
B = 32'b0_10000000_10000000000000000000000

操作数预处理：
A = 33'b0_10000010_101000000000000000000000
B = 32'b0_10000000_110000000000000000000000

运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp = A.exp - B.exp
        = 0b10000010 - ( 0b10000000 - 127 )
        = 0b10000010 - 1
        = 0b10000001

sum.m = A.m / B.m
      = { A.m, 4'd0 } / { 4'd0, B.m }
      = 28'b1010000000000000000000000000 / 28'b0000110000000000000000000000
      = 28'b0000000000000000000000001101, reminder is 0101010101...

结果调整：
$ sum.m[3] is 1, while sum.m << 24 and sum.exp -1

sum'.exp = 0b10000001 - 1 = 0b10000000

h m
0 0000000000000000000000001101 before
1 1010000000000000000000000000 << 24
1 1010000000000000000000000000 after, sum = sum's

输出和格式化：
32'b0_10000000_10100000000000000000000 Test Result 3.2500000
32'b0_10000000_10101010101010101010101 Real Result 3.3333332
                                       True Result 3.333...
```

Q1b 中的 Test Result 距离 Real Result 虽然已经接近了，但是精度的流失极为严重。不管了，我们把马力加到最大，尽可以把操作数 A 的尾数加到最大，然后在一次运算 Q1 看看。

Q1C: A = 10，B = 3，A / B = ?

```
A = 32’b0_10000010_01000000000000000000000
B = 32’b0_10000000_10000000000000000000000

操作数预处理：
A = 33’b0_10000010_101000000000000000000000
B = 32’b0_10000000_110000000000000000000000

运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp = A.exp - B.exp
        = 0b10000010 - ( 0b10000000 - 127 )
        = 0b10000010 - 1
        = 0b10000001

sum.m = A.m / B.m
      = { A.m, 24’d0 } / { 24’d0, B.m }
      = 48’b101000000000000000000000000000000000000000000000 / 48’b000000000000000000000000110000000000000000000000
      = 48’b000000000000000000000000110101010101010101010101, reminder is 0101010101...

结果调整：
$ sum.m[23] is 1, while sum.m << 24 and sum.exp -1

sum’.exp = 0b10000001 - 1 = 0b10000000

h m
0 000000000000000000000000110101010101010101010101 before
1 101010101010101010101010000000000000000000000000 << 24
1 101010101010101010101010000000000000000000000000 after, sum = sum’s

输出和格式化：
32’b0_10000000_10101010101010101010101 Test Result 3.3333332
32’b0_10000000_10101010101010101010101 Real Result 3.3333332
                                       True Result 3.333...
```

Q1c 的预算结果中，我们得到的 Test Result 和 Real Result 一模一样，所以结果是正确的。在这里，估计许多同学都看到头冒金星，完全看不懂运算细节。同学们别着急，向把运算的细节放在一边去。为此笔者只是想证明一件事就是，尽可以减少余数的产生，虽然 10/3 可能有算不完的余数，但是

我们要尽最大的努力和最大的可能避免余数过多而导致精度流失的问题。

如 Q1b 的结果，操作数 A 的马力只开到 4 位档而已，所以造成 Test Result 严重的精度流失。反之在 Q1c 的运算结果中，操作数 A 的马力开到最大，亦即 24 位档，最终 Test Result 的结果已经接近 Real Result。这并不是所谓精度的流失没有发生，而是压缩到最低的程度而已。事实上 24 位大的马力不是最大档，而马力是没有上限的，马力越大精度的流失可以压缩越低，作为代价逻辑资源的消耗就会越高。不过对于单精度可是来说，24 位档的马力已经足够了，多了就是浪费。

==

根据 Q1c 的运算过程，浮点除法的大概步骤如下所示：

（一）操作数预处理：恢复操作数 A 和 B；
（二）运算：符号位运算，价码运算和尾数运算独立运行，尾数运算中可以考虑马力的档位；
（三）结果调整：调整运算结果适合浮点数源的格式；
（四）输出和格式化：检测任何错误信息，执行四舍五入操作和输出结果。

步骤一：操作数预处理

浮点除法和浮点乘法的操作数预处理步骤，内容没有变化。该步骤的目的就是恢复隐藏在操作数中的符号位。Verilog HDL 语言的实现如代码 1.10.1 所示：

```
begin
    rA <= { A[31], A[30:23], 1'b1, A[22:0] };
    rB <= { B[31], B[30:23], 1'b1, B[22:0] };
end
```

代码 1.10.1 操作数预处理。

步骤二：运算

浮点除法比起浮点乘法所存在的陷阱，由于阴险程度一点也比不上，所以符号位，价码和尾数可以独立运算。至于 Verilog HDL 语言的实现如代码 1.10.2 所示。在代码 1.10. 2 所示中操作数 A 的尾数左移 24 位，对于单精度格式的来说这种程度的马力已经足够满足运算。

```
begin
     isSign <= A[31] ^ B[31];

     BDiff = rB[31:24] - 8'd127;
     rExp <= rA[31:24] - BDiff;

     Temp <= { rA[23:0] , 24'd0} / { 24'd0 , rB[23:0] };
end
```

代码 1.10.2 运算。

步骤三：结果调整

还记得在浮点乘法中出现的陷阱之一吗？假设“假定”小数点介于 Temp[47]~Temp[46]之间，在结果调整中，如果 Temp[47] 为 1 结果调整没有发生；再如果 Temp[46]为 1，结果就要左移 1 位，可是价码没有变化；再再如果 Temp[45]为 1，结果就要左移 2 位，此时价码就要 -1。同样的陷阱也出现在浮点除法之中，不过只是换了一件衣服而已。笔者举个例子：

假设操作数 A 和 B 均为 4 位，操作数 A 最佳的马力档位是 4，那么相除结果一定是 8 位。然后假设“假定”小数点介于结果的 [7]~[6]之间。

QI：	QII：	QIII：	QIV：
A = 4’b1111	A = 4’b1111	A = 4’b1000	A = 4’b1000
B = 4’b1000	B = 4’b1111	B = 4’b1000	B = 4’b1111
8’b11110000	8’b11110000	8’b10000000	8’b10000000
8’b00001000 ÷	8’b00001111 ÷	8’b00001000 ÷	8’b00001111 ÷
8’b00011110 before	8’b00010000 before	8’b00010000 before	8’b00001001 before
8’b11110000 << 3	8’b10000000 << 3	8’b10000000 << 3	8’b10010000 << 4，exp -1
8’b11110000 after	8’b10000000 after	8’b10000000 after	8’b10010000 after

笔者列出四个最大的可能会出现的状况，在 QI 的操作组中，A 左移 4 位，然后在运算之后，得到的答案为 8'b00011110，亦即[4] 为 1。在结果调整中，左移 3 位价码不变；在 QII 的操作组中，执行同样的动作，得到的运算结果是 8'b00010000，亦即[4] 为 1。在结果调整中，左移 3 位价码不变；在 QIII 的运算结果中，取得 8'b00010000，亦即[4] 为 1，结果左移 3 位，价码不变；在最后的 QIV 运算中，求得结果为 8'b00001001，亦即[3]为 1，结果左移 4 位，然后价码递减 1。

如果操作数从 4 位变成 24 位，马力为 24 档位，然后根据 QI~QIV 之间的情况。那么，QI~QIII 的运算结果中，[24]一定为 1，然后结果左移 23 位。至于 QIV 的运算结果，[23]一定为 1，然后结果左移 24 位，价码随之减一。为了给结果调整买份保险，我们可以在 [24 ... 23] 的前后加入调整的范围。Verilog HDL 语言的实现如代码 1.10.3 所示：

```
begin
        if( Temp[25] == 1'b1 ) begin Temp <= Temp << 22; end
        else if( Temp[24] == 1'b1 ) begin Temp <= Temp << 23; end
        else if( Temp[23] == 1'b1 ) begin Temp <= Temp << 24; rExp <= rExp - 1'b1; end
end
```

代码 1.10.3 结果调整。

在代码 1.10.3 中，if(Temp[25] == 1'b1) 是一个保险措施，虽然笔者知道在运算之后，调整范围一定会发生在 Temp[24] ~ Temp[23] 之间。不过为了安心，笔者还是买下这份绝对安心保险（意思是不可能得到赔偿的保险）。

步骤四：输出和格式化

```
begin
        if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end
        else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end
          else if( Temp[47:24] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end
          else rResult <= { isSign, rExp[7:0], Temp[46:24] };
 end
```

代码 1.10.4 输出和格式化。

在输出和格式化这个步骤中，浮点乘法和浮点除法的内容基本上是大同小异，因为浮点除法和浮点乘法都一样，我们“假定”小数点介于 Temp[47]~Temp[46] 之间。不过有一点读者会吓到的是，浮点除法少了四舍五入这个操作，为什么呢？有两种原因，第一个原因是经过结果调整以后，Temp[23]~Temp[0] 很大可能是全零。另一个原因是，在尾数运算途中保护精度的措施已经最大发挥了，所以四舍五入操作显得有点鹌鹑，因此就被节了。

==

到目前为止，读者的心是否还是忐忑？头脑是否迷糊中？这是理所当然的事情，因为浮点除法运算比起其他的运算，它虽然容易但是显得有点特殊。不过不要紧，只要我们简单的手动运算一下，有个感知的认识以后，这些“讨厌”的感觉自然而然会消失。

（注意在接下来的手动运算中，笔者会省略掉操作数预处理这个操作）

Q2：A = 6.32，B = 0.26，A / B = ?

```
A = 32'b0_10000001_10010100011110101110001
B = 32'b0_01111101_00001010001111010111000

运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp    = A.exp - B.exp
           = 0b10000001 - ( 0b01111101 - 127 )
           = 0b10000001 -  ( -2 )
           = 0b10000011

sum.m      = A.m / B.m
           = { A.m,  24'd0 } / { 24'd0,  B.m }
           = 48'b110010100011110101110001 ... 0 /  48'b0 ... 100001010001111010111000
           = 48'b0000000000000000000000001100001001110110001010000

结果调整：
$ sum.m[24] is 1, while sum.m << 23
```

```
h m
0 000000000000000000000001100001001110110001010000 before
1 100001001110110001010000000000000000000000000000 << 23
1 100001001110110001010000000000000000000000000000 after, sum = sum'

输出和格式化：
32'b0_10000011_10000100111011000101000 Test Result 24.3076934
32'b0_10000011_10000100111011000100111 Real Result 24.3076915
                                          True Result 24.307692
```

Q3：A = 3.48，B = 10，A / B = ?

```
A = 32'b0_10000000_10111101011100001010010
B = 32'b0_10000010_01000000000000000000000

运算：
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp    = A.exp - B.exp
          = 0b10000000 - ( 0b10000010 - 127 )
          = 0b10000000 - （ 3 ）
          = 0b01111101

sum.m      = A.m / B.m
          = { A.m, 24'd0 } / { 24'd0, B.m }
          = 48'b101111010111000010100100 ... 0 /  48'b0 ... 101000000000000000000000
          = 48'b000000000000000000000000101100100010110100001110 0

结果调整：
$ sum.m[24] is 1, while sum.m << 23

h m
0 000000000000000000000001011001000101101000011100 before
1 011001000101101000011100000000000000000000000000 << 23
1 011001000101101000011100000000000000000000000000 after, sum = sum'

输出和格式化：
32'b0_01111101_01100100010110100001110 Test Result 0.3479999
32'b0_01111101_01100100010110100001110 Real Result 0.3479999
                                          True Result 0.348
```

Q4：A = 1.442，B = 11.515，A / B = ?

```
A = 32'b0_01111111_01110001001001101110101
B = 32'b0_10000010_01110000011110101110001

运算:
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp    = A.exp - B.exp
           = 0b01111111 - ( 0b10000010 - 127 )
           = 0b01111111 - ( 3 )
           = 0b01111100

sum.m      = A.m / B.m
           = { A.m, 24'd0 } / { 24'd0, B.m }
           = 48'b101110001001001101110101 ... 0 /  48'b0 ... 101110000011110101110001
           = 48'b000000000000000000000001000000000111011110000100

结果调整:
$ sum.m[24] is 1, while sum.m << 23

h m
0 000000000000000000000001000000000111011110000100 before
1 000000000111011110000100000000000000000000000000 << 23
1 000000000111011110000100000000000000000000000000 after, sum = sum'

输出和格式化:
32'b0_01111100_00000000011101111000010 Test Result 0.1252279
32'b0_01111100_00000000011101111000010 Real Result 0.1252279
                                           True Result 0.12522796352...
```

Q5：A = 1000，B = 0.5，A / B = ?

```
A = 32'b0_10001000_11110100000000000000000
B = 32'b0_01111110_00000000000000000000000

运算:
sum.s = A.s ^ B.s = 0 ^ 0 = 0

sum.exp    = A.exp - B.exp
           = 0b10001000 - ( 0b01111110 - 127 )
           = 0b10001000 - ( -1 )
           = 0b10001001

sum.m      = A.m / B.m
           = { A.m, 24'd0 } / { 24'd0, B.m }
```

```
          = 48'b111110100000000000000000 ... 0 /  48'b0 ... 100000000000000000000000
          = 48'b000000000000000000000001111101000000000000000000

结果调整:
$ sum.m[24] is 1, while sum.m << 23

h m
0 00000000000000000000001111101000000000000000000 before
1 11111010000000000000000000000000000000000000000 << 23
1 11111010000000000000000000000000000000000000000 after, sum = sum'

输出和格式化:
32'b0_10001001_11110100000000000000000 Test Result 2000
32'b0_10001001_11110100000000000000000 Real Result 2000
                                          True Result 2000
```

Q6：A = 3.142857，B = -0.723093，A / B = ?

```
A = 32'b0_10000000_10010010010010010010010
B = 32'b1_01111110_01110010001110010011111

运算:
sum.s = A.s ^ B.s = 0 ^ 1 = 1

sum.exp    = A.exp - B.exp
          = 0b10000000 - ( 0b01111110 - 127 )
          = 0b10000000 -  ( -1 )
          = 0b10000001

sum.m      = A.m / B.m
          = { A.m, 24'd0 } / { 24'd0, B.m }
          = 48'b110010010010010010010010 ... 0 /  48'b0 ... 101110010001110010011111
          = 48'b000000000000000000000001000101100010101110001100

结果调整:
$ sum.m[24] is 1, while sum.m << 23

h m
0 00000000000000000000001000101100010101110001100 before
1 00010110001010111000110000000000000000000000000 << 23
1 00010110001010111000110000000000000000000000000 after, sum = sum'

输出和格式化:
32'b1_10000001_00010110001010111000110 Test Result -4.3464078
```

```
32'b0_10001001_00010110001010111000110 Real Result -4.3464078
                                       True Result -4.3464077
```

经过 Q2~Q6 这五题的运算以后，读者有什么感想呢？从中我们可以看到从 Q2~Q6，操作数组内容越来越挑剔，不过取得的 Test Result 都非常接近 Real Result。基本上，浮点除法的运算过程就是这样而已，只要把基本概念搞清楚以后就没有什么好怕的。

1.11 实验四：浮点数除法模块

图 1.11.1 浮点数除法模块图形。

图 1.11.1 是实验四里要建立的浮点数除法模块。至于信号之间的定义，基本上和前几个实验一样，没有什么号重复的。我们还是直接来看代码吧：

float_divide_module.v

```
1.   module float_divide_module
2.   (
3.       input CLK, RSTn,
4.       input [31:0]A,B,
5.       output [31:0]Result,
6.
7.       input Start_Sig,
8.       output [3:0]Done_Sig,
9.
10.      /*****************/
11.
12.      output [32:0]SQ_rA,SQ_rB,
13.      output [47:0]SQ_Temp,
14.      output [9:0]SQ_rExp,SQ_BDiff
15.
16.  );
```

第 3~8 行是该模块的输入输出声明，反之第 12~14 行是仿真用的输出。

```
17.     /**************************************/
18.
19.     reg [3:0]i;
20.     reg [32:0]rA,rB;   // [32]Sign, [31:24]Exponent, [23]Hidden Bit, [22:0]Mantissa
21.     reg [47:0]Temp;     // [47] Hidden-A [46:24] Mantissa-A , [23] Hidden-B [22:0]Mantissa-B
22.     reg [31:0]rResult;
23.     reg [9:0]rExp,BDiff;        //[9:8] Overflow or underflow check, [7:0] usuall exp.
24.     reg isSign;
```

```
25.     reg isOver;
26.     reg isUnder;
27.     reg isZero;
28.     reg isDone;
29.
30.     always @ ( posedge CLK or negedge RSTn )
31.         if( !RSTn )
32.             begin
33.                 i <= 4'd0;
34.                 rA <= 33'd0;
35.                 rB <= 33'd0;
36.                 Temp <= 24'd0;
37.                 rResult <= 32'd0;
38.                 rExp <= 10'd0;
39.                 BDiff <= 10'd0;
40.                 isOver <= 1'b0;
41.                 isUnder <= 1'b0;
42.                 isZero <= 1'b0;
43.                 isDone <= 1'b0;
44.             end
```

第 19~28 行是相关的寄存器声明，功能上和浮点数乘法模块大同小异，不过 Temp 是寄存相除以后的结果。第 33~43 行是复位动作，所有寄存器都清零。

```
45.         else if( Start_Sig )
46.             case( i )
47.
48.                 0: // Initial and resulted out Signature
49.                 begin
50.                     rA <= { A[31], A[30:23], 1'b1, A[22:0] };
51.                     rB <= { B[31], B[30:23], 1'b1, B[22:0] };
52.
53.                     isSign <= A[31] ^ B[31];
54.                     isOver <= 1'b0; isUnder <= 1'b0; isZero <= 1'b0;
55.                     i <= i + 1'b1;
56.                 end
57.
58.                 1: // if rExp[9..8] is 1, mean A.Exp small than B.Exp
59.                    // while rExp[9..8] is 0, mean A.Exp large than B.Exp or same.
60.                 begin
61.                     BDiff = rB[31:24] - 8'd127; // + (~8'd127 + 1)
62.                     rExp <= rA[31:24] - BDiff; // A.Exp - B.Exp
63.                     i <= i + 1'b1;
64.                 end
```

```
65.
66.                 2: // Divide and A.m << 24
67.                 begin
68.                     Temp <= { rA[23:0] , 24'd0} / { 24'd0 , rB[23:0] };
69.                     i <= i + 1'b1;
70.                 end
```

第 49~56 行是操作数预处理和符号位运算；第 58~64 行是价码运算；第 66~70 行是尾数运算。虽然说，符号位运算，价码运算和尾数运算等操作都可以压缩在同一个步骤里运行，但是为了统一和前面几个实验一样的风格，因此笔者免了。上述的操作基本上和 1.10 小节里解释过的一样，在此笔者就不重复了。

```
71.
72.                 3: // Check M'hidden bit
73.                 begin
74.                     if( Temp[25] == 1'b1 ) begin Temp <= Temp << 22; end
75.                     else if( Temp[24] == 1'b1 ) begin Temp <= Temp << 23; end
76.                     else if( Temp[23] == 1'b1 ) begin Temp <= Temp << 24; rExp <= rExp - 1'b1; end
77.
78.                     i <= i + 1'b1;
79.                 end
80.
81.                 4: //error check and decide final result
82.                 begin
83.                     if( rExp[9:8] == 2'b01 ) begin isOver <= 1'b1; rResult <= {1'b0,8'd127, 23'd0}; end      // E Overflow
84.                     else if( rExp[9:8] == 2'b11 ) begin isUnder <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // E Underflow
85.                     else if( Temp[47:24] == 24'd0 ) begin isZero <= 1'b1; rResult <= {1'b0, 8'd127, 23'd0}; end // M Zero
86.                     else rResult <= { isSign, rExp[7:0], Temp[46:24] }; // okay without normalise
87.                     i <= i + 1'b1;
88.                 end
```

第 72~79 行是结果调整，然而第 81~88 行是输出和格式化。同样，笔者也不重复解释了。

```
89.
90.                 5:
91.                 begin isDone <= 1'b1; i <= i + 1'b1; end
92.
93.                 6:
94.                 begin isDone <= 1'b0; i <= 4'd0; end
95.
96.             endcase
97.
98.     /**************************************/
99.
```

```
100.      assign Done_Sig = { isOver, isUnder, isZero, isDone };
101.      assign Result = rResult;
102.
103.      /****************************************/
104.
105.        assign SQ_rA = rA;
106.        assign SQ_rB = rB;
107.        assign SQ_rExp = rExp;
108.        assign SQ_BDiff = BDiff;
109.        assign SQ_Temp = Temp;
110.
111.      /*****************************************/
112.
113.  endmodule
```

余下的步骤 5~6 是产生完成信号，至于第 100~101 行是相关的输出驱动。最后第 105~109 行是仿真用的相关输出驱动。

float_divide_module.vt

```
1.    `timescale 1 ps/ 1 ps
2.    module float_divide_module_simulation();
3.
4.        reg CLK;
5.        reg RSTn;
6.
7.        reg [31:0] A;
8.        reg [31:0] B;
9.        reg Start_Sig;
10.
11.       wire [3:0]   Done_Sig;
12.       wire [31:0]   Result;
13.
14.       /**************************/
15.
16.       wire [9:0]   SQ_BDiff;
17.       wire [47:0]   SQ_Temp;
18.       wire [32:0]   SQ_rA;
19.       wire [32:0]   SQ_rB;
20.       wire [9:0]   SQ_rExp;
21.
22.       /**************************/
23.
24.       float_divide_module U1
```

```
25.        (
26.            .CLK(CLK),
27.             .RSTn(RSTn),
28.             .A(A),
29.             .B(B),
30.             .Result(Result),
31.             .Start_Sig(Start_Sig),
32.             .Done_Sig(Done_Sig),
33.             /************************/
34.             .SQ_BDiff(SQ_BDiff),
35.             .SQ_Temp(SQ_Temp),
36.             .SQ_rA(SQ_rA),
37.             .SQ_rB(SQ_rB),
38.             .SQ_rExp(SQ_rExp)
39.       );
40.
41.     /*****************************/
42.
43.     initial
44.       begin
45.             RSTn = 0; #10 RSTn = 1;
46.             CLK = 0; forever #5 CLK = ~CLK;
47.       end
48.
49.     /******************************/
50.
51.     reg [5:0]i;
52.
53.       always @ (   posedge CLK or negedge RSTn )
54.             if( !RSTn )
55.                  begin
56.                    A <= 32'd0;
57.                    B <= 32'd0;
58.                    Start_Sig <= 1'b0;
59.                    i <= 6'd0;
60.               end
61.          else
62.               case( i )
63.
64.                         0: // 3.142 / 3.142
65.                         if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
66.                       else begin A <= 32'b01000000010010010001011010000111; B <= 32'b01000000010010010001011010000111; Start_Sig <= 1'b1; end
67.
68.                         1:
69.                         begin
```

```
70.                 $display("A = %b and rA = %b", A,SQ_rA);
71.                 $display("B = %b and rB = %b", B,SQ_rB);
72.                 $display("A.m / B.m = %b", SQ_Temp);
73.                 $display("Result = %b", Result);
74.                 i <= i + 1'b1;
75.             end
76.
77.             2: // 10 / 3
78.             if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
79.             else begin A <= 32'b01000001001000000000000000000000; B <= 32'b01000000010000000000000000000000; Start_Sig <= 1'b1; end
80.
81.             3:
82.             begin
83.                 $display("A = %b and rA = %b", A,SQ_rA);
84.                 $display("B = %b and rB = %b", B,SQ_rB);
85.                 $display("A.m / B.m = %b", SQ_Temp);
86.                 $display("Result = %b", Result);
87.                 i <= i + 1'b1;
88.             end
89.
90.             4: // 6.32 / 0.26
91.             if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
92.             else begin A <= 32'b01000000110010100011110101110001; B <= 32'b00111110100001010001111010111000; Start_Sig <= 1'b1; end
93.
94.             5:
95.             begin
96.                 $display("A = %b and rA = %b", A,SQ_rA);
97.                 $display("B = %b and rB = %b", B,SQ_rB);
98.                 $display("A.m / B.m = %b", SQ_Temp);
99.                 $display("Result = %b", Result);
100.                i <= i + 1'b1;
101.            end
102.
103.            6: // 3.48 / 10
104.            if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
105.            else begin A <= 32'b01000000010111101011100001010010; B <= 32'b01000001001000000000000000000000; Start_Sig <= 1'b1; end
106.
107.            7:
108.            begin
109.                $display("A = %b and rA = %b", A,SQ_rA);
110.                $display("B = %b and rB = %b", B,SQ_rB);
111.                $display("A.m / B.m = %b", SQ_Temp);
112.                $display("Result = %b", Result);
113.                i <= i + 1'b1;
114.            end
```

```
115.
116.                8: // 1.442 / 11.515
117.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
118.                else begin A <= 32'b00111111101110001001001101110101; B <= 32'b01000001001110000011110101110001; Start_Sig <= 1'b1; end
119.
120.                9:
121.                begin
122.                    $display("A = %b and rA = %b", A,SQ_rA);
123.                    $display("B = %b and rB = %b", B,SQ_rB);
124.                    $display("A.m / B.m = %b", SQ_Temp);
125.                    $display("Result = %b", Result);
126.                    i <= i + 1'b1;
127.                end
128.
129.                10: // 1000 / 0.5
130.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
131.                else begin A <= 32'b01000100011110100000000000000000; B <= 32'b0_01111110_00000000000000000000000; Start_Sig <= 1'b1; end
132.
133.                11:
134.                begin
135.                    $display("A = %b and rA = %b", A,SQ_rA);
136.                    $display("B = %b and rB = %b", B,SQ_rB);
137.                    $display("A.m /B.m = %b", SQ_Temp);
138.                    $display("Result = %b", Result);
139.                    i <= i + 1'b1;
140.                end
141.
142.                12: // 1000 / 4.5
143.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
144.                else begin A <= 32'b01000100011110100000000000000000; B <= 32'b01000000100100000000000000000000; Start_Sig <= 1'b1; end
145.
146.                13:
147.                begin
148.                    $display("A = %b and rA = %b", A,SQ_rA);
149.                    $display("B = %b and rB = %b", B,SQ_rB);
150.                    $display("A.m /B.m = %b", SQ_Temp);
151.                    $display("Result = %b", Result);
152.                    i <= i + 1'b1;
153.                end
154.
155.                14: // 176.9218 / 1146.3369
156.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
157.                else begin A <= 32'b01000011001100001110101111111011; B <= 32'b01000100100011110100101011001000; Start_Sig <= 1'b1; end
158.
159.                15:
```

```
160.                begin
161.                    $display("A = %b and rA = %b", A,SQ_rA);
162.                    $display("B = %b and rB = %b", B,SQ_rB);
163.                    $display("A.m /B.m = %b", SQ_Temp);
164.                    $display("Result = %b", Result);
165.                    i <= i + 1'b1;
166.                end
167.
168.                16: // 3.142857 / -0.723093
169.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
170.                else begin A <= 32'b01000000010010010010010010010010; B <= 32'b10111111001110010001110010011111; Start_Sig <= 1'b1; end
171.
172.                17:
173.                begin
174.                    $display("A = %b and rA = %b", A,SQ_rA);
175.                    $display("B = %b and rB = %b", B,SQ_rB);
176.                    $display("A.m / B.m = %b", SQ_Temp);
177.                    $display("Result = %b", Result);
178.                    i <= i + 1'b1;
179.                end
180.
181.                18: // 0.0625 / 0.25
182.                if( Done_Sig[0] ) begin Start_Sig <= 1'b0; i <= i + 1'b1; end
183.                else begin A <= 32'b00111101100000000000000000000000; B <= 32'b00111110100000000000000000000000; Start_Sig <= 1'b1; end
184.
185.                19:
186.                begin
187.                    $display("A = %b and rA = %b", A,SQ_rA);
188.                    $display("B = %b and rB = %b", B,SQ_rB);
189.                    $display("A.m /B.m = %b", SQ_Temp);
190.                    $display("Result = %b", Result);
191.                    i <= i + 1'b1;
192.                end
193.
194.                20:
195.                i <= i;
196.
197.            endcase
198.
199. endmodule
```

激励文本的书写风格基本上没有什么改变，怒笔者懒得解释了。我们还是直接来看仿真结果吧。

仿真结果：

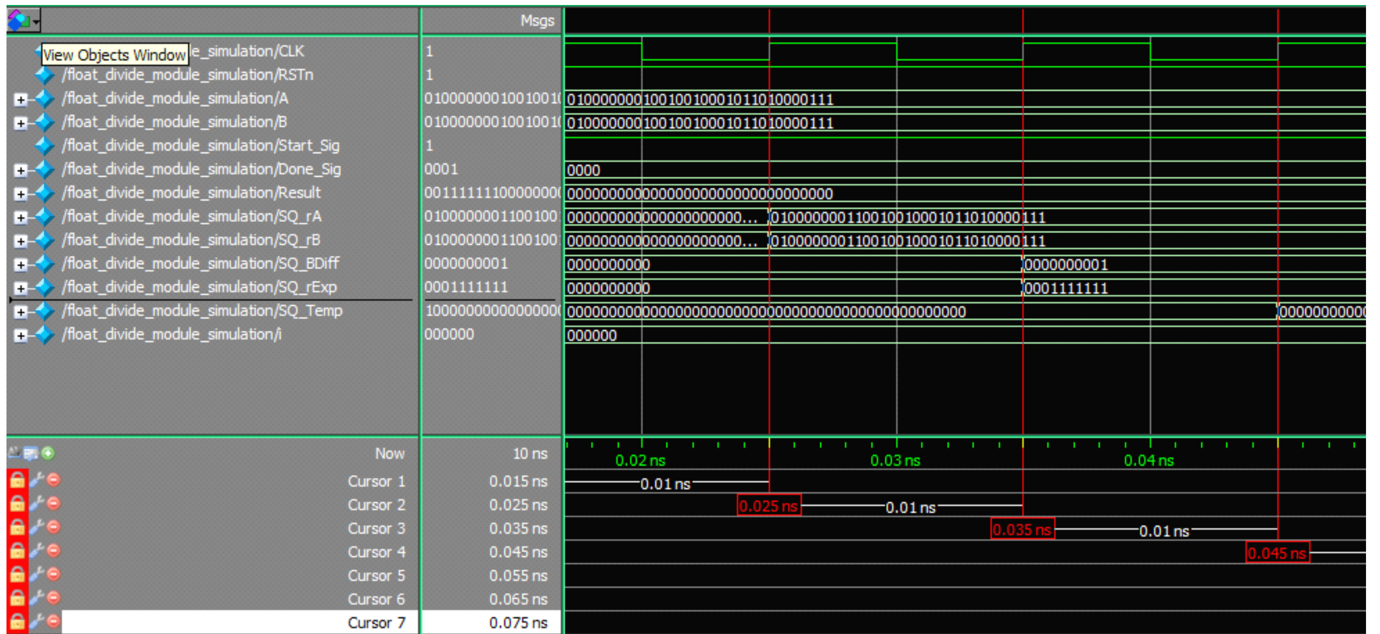

图 1.11.2 仿真过程 1。

如图 1.11.2 所示，在 C1 的时候该模块被使能。C2 的时候是操作数预处理和符号位运算，所以在 C2 的未来输出被回复的 rA 和 rB，此外也取得符号运算结果。在 C3 的时候是价码操作，所以在 C3 的未来输出价码运算的结果。

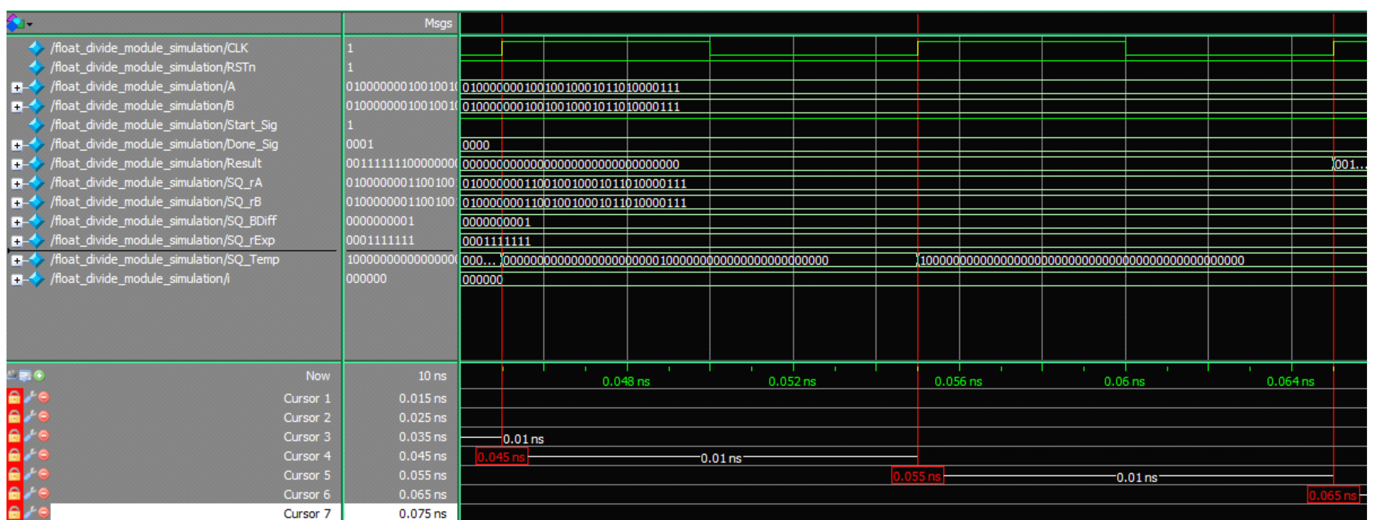

图 1.11.3 仿真过程 2。

在 C4 的时候是尾数运算，A 的尾数被拉大 24 档马力然后和 B 运算，所以在 C4 的未来输出运算的结果。在 C5 的时候是结果调整，Temp[24]的过去值为 1，所以在 C5 的未来输出被左移 24 位的 Temp。

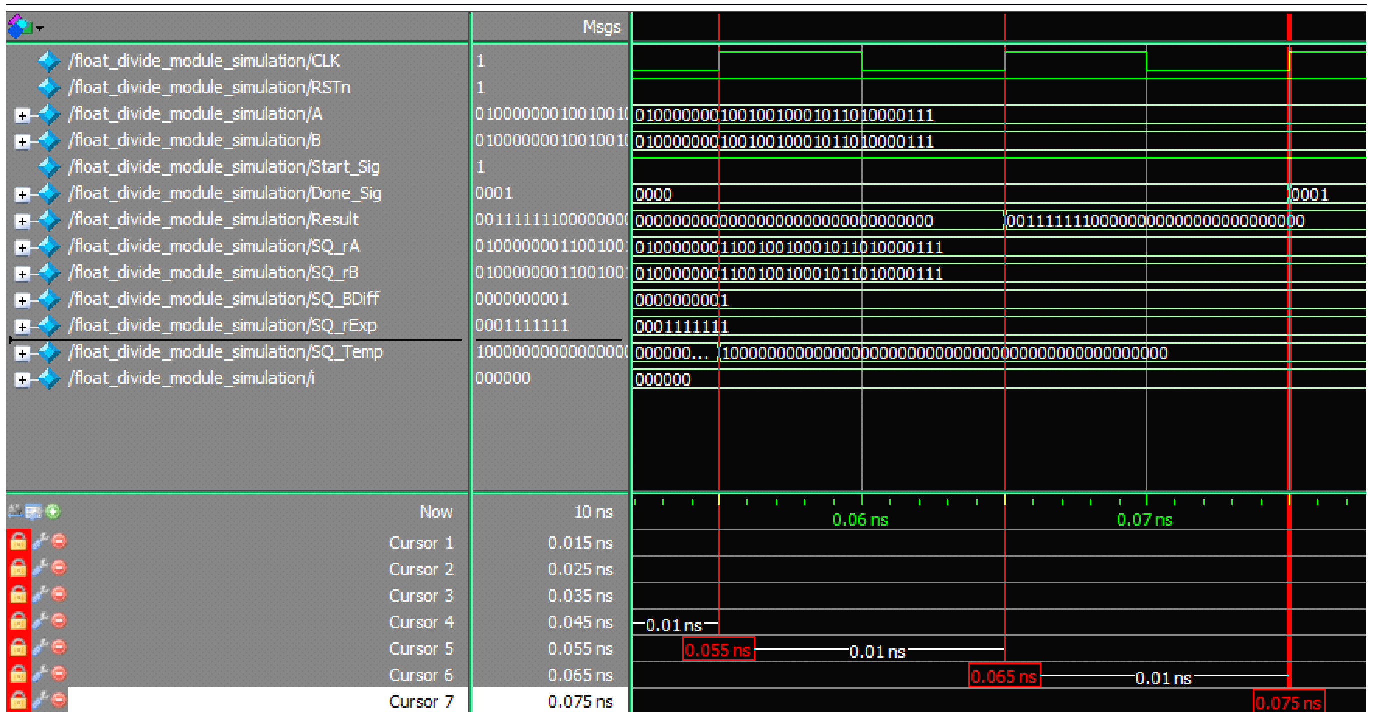

图 1.11.4 仿真过程 3。

在 C6 的时候是输出和格式化，所以在 C6 的未来输出相关的结果。基本上，浮点除法运算是总多运算之中最简单，步骤也是最单调的。接下来还是老习惯，笔者不解释所有仿真过程，作为补偿笔者给出所有运算的验证结果。

Q1: A = 3.142, B = 3.142, A / B = ?

```
A = 32'b0_10000000_10010010001011010000111
B = 32'b0_10000000_10010010001011010000111

32'b0_01111111_00000000000000000000000 Test Result 1
32'b0_01111111_00000000000000000000000 Real Result 1
                                        True Result 1
```

Q2: A = 10, B = 3, A / B = ?

```
A = 32'b0_10000010_01000000000000000000000
B = 32'b0_10000000_10000000000000000000000

32'b0_10000000_10101010101010101010101 Test Result 3.3333332
32'b0_10000000_10101010101010101010101 Real Result 3.3333332
                                        True Result 3.3333333...
```

Q3: A = 6.32, B = 0.26, A / B = ?

```
A = 32'b0_10000001_10010100011110101110001
```

```
B = 32'b0_01111101_00001010001111010111000

32'b0_10000011_10000100111011000101000 Test Result 24.3076934
32'b0_10000011_10000100111011000100111 Real Result 24.3076915
                                       True Result 24.30769
```

Q4: A = 3.48，B = 10，A / B = ?

```
A = 32'b0_10000000_10111101011100001010010
B = 32'b0_10000010_01000000000000000000000

32'b0_01111101_01100100010110100001110 Test Result 0.3479999
32'b0_01111101_01100100010110100001110 Real Result 0.3479999
                                       True Result 0.348
```

Q5: A = 1.442，B = 11.515，A / B = ?

```
A = 32'b0_01111111_01110001001001101110101
B = 32'b0_10000010_01110000011110101110001

32'b0_01111100_00000000011101111000010 Test Result 0.1252279
32'b0_01111100_00000000011101111000010 Real Result 0.1252279
                                       True Result 0.12522796352...
```

Q6: A = 1000，B = 0.5，A / B = ?

```
A = 32'b0_10001000_11110100000000000000000
B = 32'b0_01111110_00000000000000000000000

32'b0_10001001_11110100000000000000000 Test Result 2000
32'b0_10001001_11110100000000000000000 Real Result 2000
                                       True Result 2000
```

Q7: A = 1000，B = 4.5，A / B = ?

```
A = 32'b0_10001000_11110100000000000000000
B = 32'b0_10000001_00100000000000000000000

32'b0_10000110_10111100011100011100011 Test Result 222.2222137
32'b0_10000110_10111100011100011100100 Real Result 222.2222290
                                       True Result 222.2222222....
```

Q8: A = 176.9218，B = 1146.3369，A / B = ?

```
A = 32’b0_10000110_01100001110101111111011
B = 32’b0_10001001_00011110100101011001000

32’b0_01111100_00111100000101001101101 Test Result 0.1543366
32’b0_01111100_00111100000101001101101 Real Result 0.1543366
                                       True Result 0.154336652
```

Q9: A = 3.142857，B = -0.723093，A / B = ?

```
A = 32’b0_10000000_10010010010010010010010
B = 32’b1_01111110_01110010001110010011111

32’b1_10000001_00010110001010111000110 Test Result -4.3464078
32’b1_10000001_00010110001010111000110 Real Result -4.3464078
                                       True Result -4.3464077
```

Q10: A = 0.0625，B = 0.25，A / B = ?

```
A = 32’b0_01111011_00000000000000000000000
B = 32’b0_01111101_00000000000000000000000

32’b0_01111101_00000000000000000000000 Test Result 0.25
32’b0_01111101_00000000000000000000000 Real Result 0.25
                                       True Result 0.25
```

Q1~Q10 基本上也没有什么好说的，因为每一题的 Test Result 都非常接近 Real Result，无论操作组拥有几位十进制的尾数都不会使精度偏离得很厉害。答案很简单，因为浮点除法所采取保护精度的方法和其他的浮点运算不同。实验再一次证明，对于单精度格式来说，执行除法运算的时候，“24 位档”是最理想的马力程度。

总结：

哇~这一章是终于结束了！从收集资料到编辑笔记，笔者真的做了很长的准备功夫。在这里有些同学可能很遗憾，因为本章没有出现任何和精密控时的内容。读者千万不要怀疑，本章的内容也是围绕着整合的思想。不过比较不同的是，本章的目的是在思想和思路上做好准备。那些从顺序语言过渡而来的朋友，在许多情况下（都是不知不觉）他们都无法抛弃一些顺序语言在头脑里扎根已久的运行模式。笔者希望透过本章可以彻底并且狠狠地为顽固的读者敲出一条可以活动的裂缝。

笔者和读者都一样，在接触 Verilog HDL 语言一段时间后，一直无法摆脱我们在这个世界的认知或者常识。我们的现实世界是基于十个手指头，同样高级语言也是基于十个手指头。相反的，主宰 Verilog 世界里的自然规则是二进制。说实话，要完全的适应 2 进制是一个非常漫长的过程，因为在生理上大脑会抗拒。

就在那个神奇的夜晚 Verilog 之神给笔者指点，神明为笔者指出一条有效并能适应二进制的捷径，那就是透过学习浮点数的运算。从本章的内容中，我们可以知晓浮点数运算在书写形式上，或者十进制的模式上，都和二进制有很大的区别，就以浮点乘法为例，那些被隐藏的陷阱就是我们不熟悉的一面。

当然神明的指引不只是要笔者适应二进制，更贴切说是适应 Verilog 世界的自然规则。神明要笔者和 Verilog 的自然融合成为和谐的一致，然后按着自然在这个世界不停的创造。此外，本章还是为后续的几个章节做好准备。笔者常说实验的目的是其次，最重要的是思路 ... 读者还记得时序篇里的整数乘法和整数除法吗？它们在接下来的章节中，不是帮助读者更能理解 Verilog HDL 语言的理想时序吗？

好了，差不多要结束本章的内容了。在结束之前，笔者希望读者可以认真去理解浮点数四中不同的操作，因为浮点数在 Verilog HDL 语言是另一个很重要的大门。想想看有多少人被摘住在这个大门之外？很可能是我们使用高级语言太多了，有一种理论说过“如果人类过度依赖某个方便的技术，那么人类某个天然的功能就会退化”。高级语言也是同样的道理，因为很多运算在使用者不知道的情况下进行。

说白了就是“在高级语言中晓得浮点数，实际上一点也不浮点数是虾米”。高级语言给人的感觉是越用回脑经越懒得思考，反之 Verilog HDL 语言越用脑经越好使。此外，笔者经过这一次完成的笔记，非常惊讶到 ... 原来有关浮点数的资料真的少得可怜，不仅而已很多资料根本不沾边。不说了，说了都使人忧心。