

## 第3章

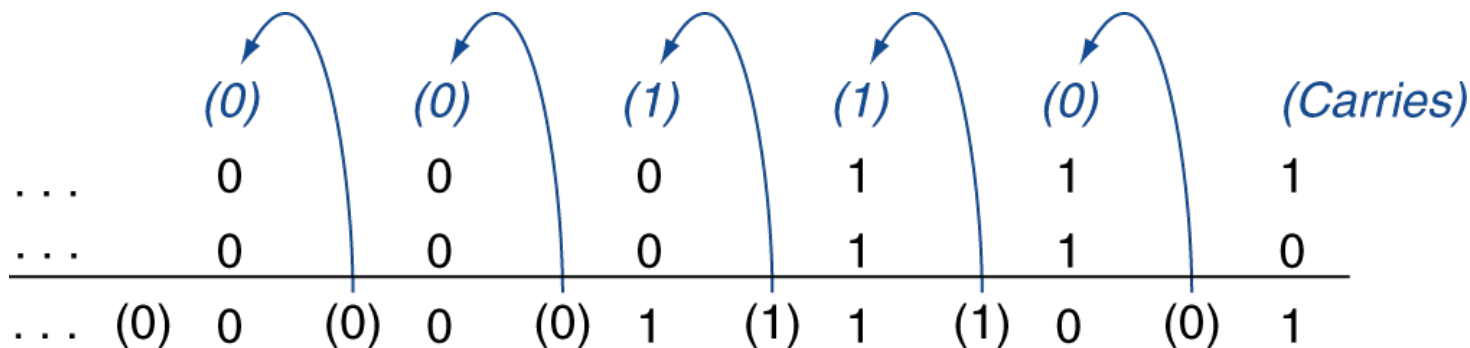
### 计算机的算术运算

# 计算机的算术运算

- 对整数的运算
  - 加法和减法
  - 乘法和除法
  - 处理溢出
- 浮点实数
  - 表示及运算

# 整数加法

■ 例：7 + 6



■ 结果超出范围则溢出

- 一正一负两个操作数相加，无溢出
- 两个正操作数相加
  - 结果的符号为1则溢出
- 两个负操作数相加
  - 结果的符号为0则溢出

# 整数减法

- 加上第二个操作数的相反数

- 例：  $7 - 6 = 7 + (-6)$

+7:	0000 0000 ... 0000 0111
-6:	1111 1111 ... 1111 1010
<hr/>	
+1:	0000 0000 ... 0000 0001

- 结果超出范围则溢出

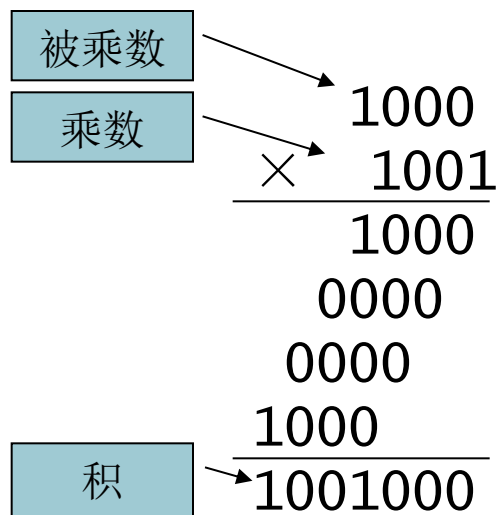
- 两个正操作数或两个负操作数相减，无溢出
- 负操作数减正操作数
  - 结果的符号为0则溢出
- 正操作数减负操作数
  - 结果的符号为1则溢出

# 用于多媒体的算术运算

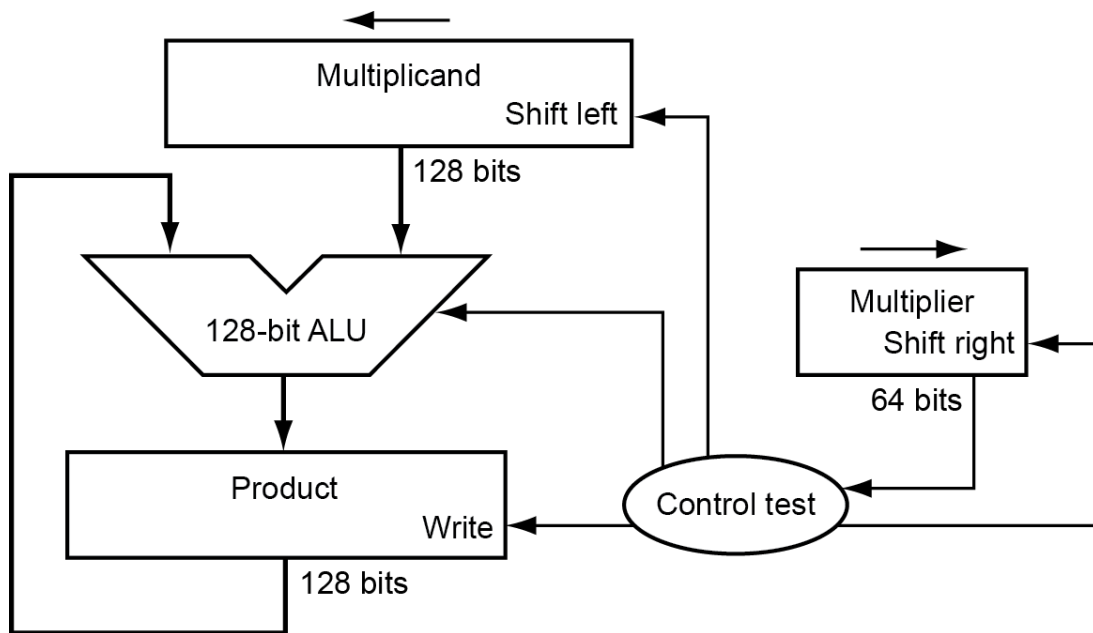
- 图形和媒体处理要对8位和16位的向量数据进行操作
  - 使用具有分段式进位链的64位加法器
    - 对8个8位、4个16位或2个32位向量进行操作
  - SIMD（单指令多数据，single-instruction, multiple-data）
- 饱和操作
  - 溢出时，结果为可表示的最大值
    - 对比二进制补码的取模运算
  - 例如，音频中的限幅、视频中的饱和处理

# 乘法

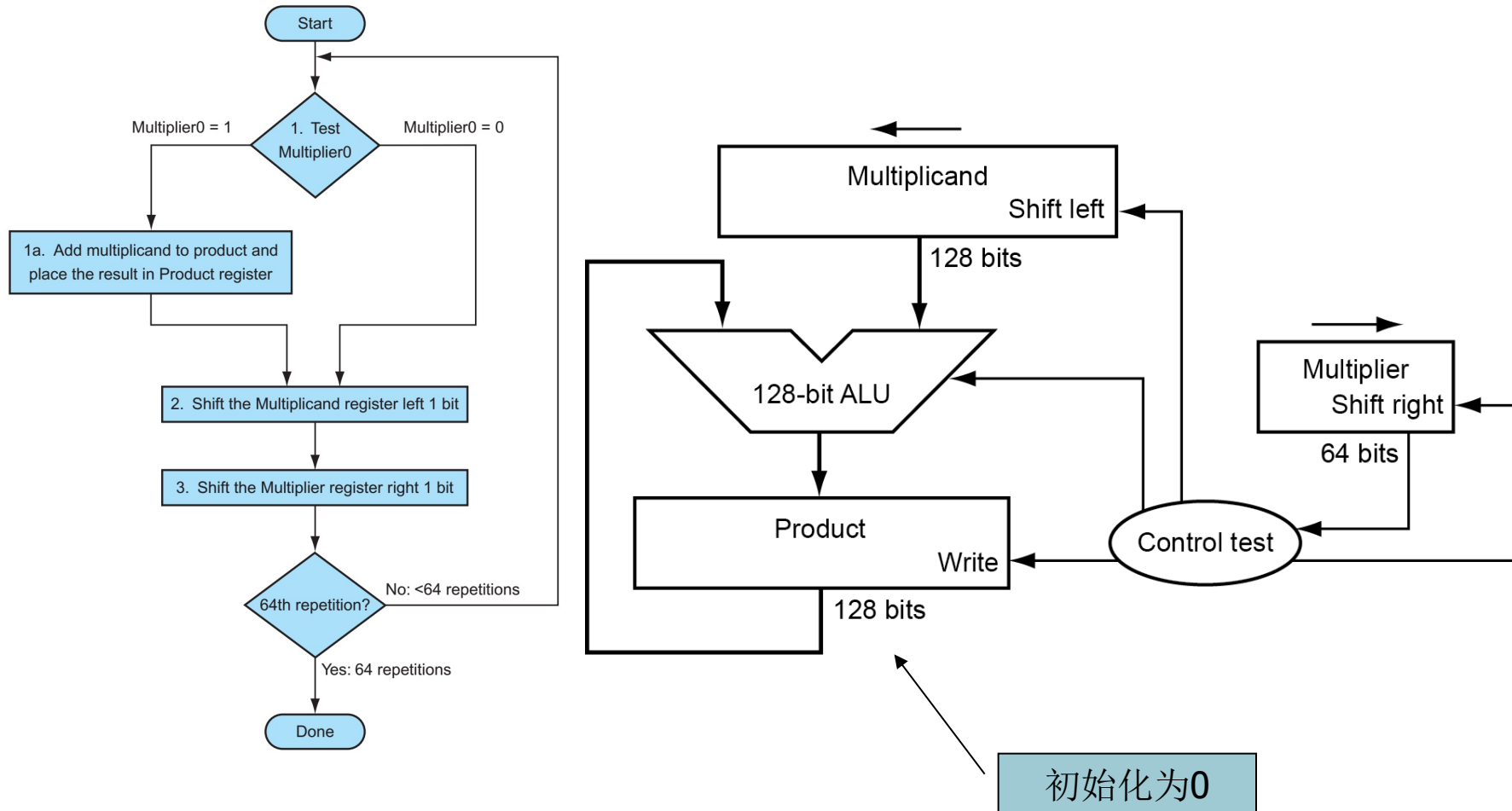
## ■ 从长乘法开始



积的长度是操作数的长度之和

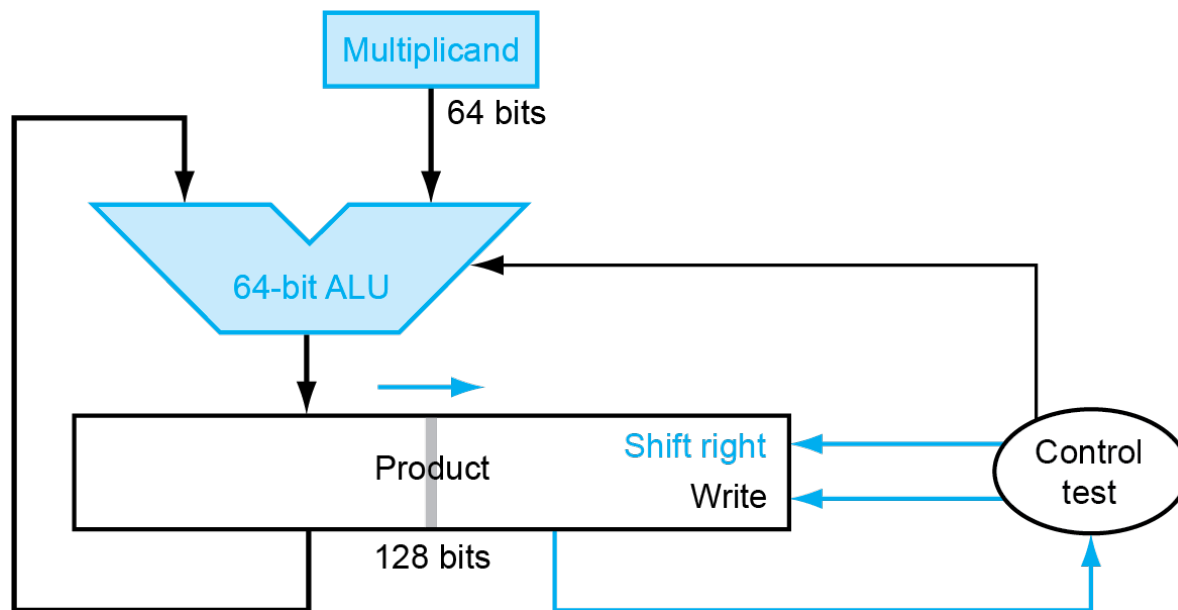


# 乘法的硬件实现



# 优化的乘法器

- 并行地执行各步骤：加法/移位



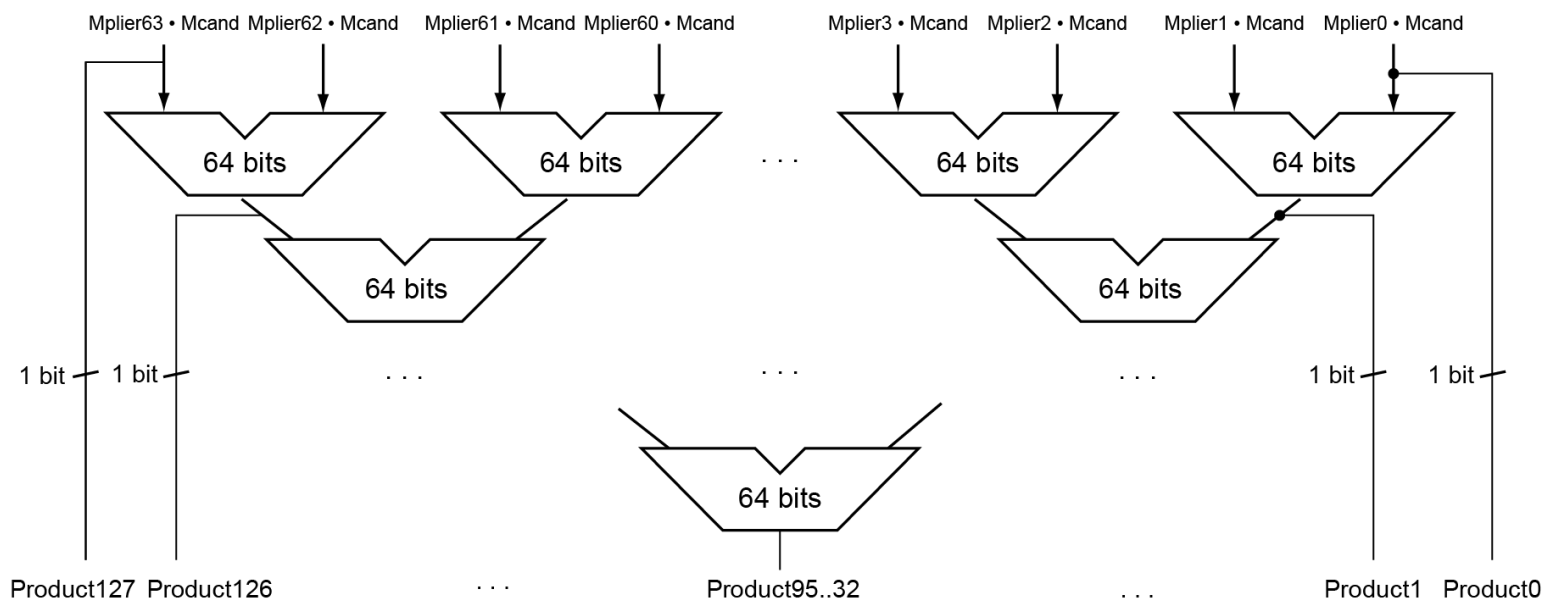
- 每次加上部分积占一个周期
  - 如果不频繁做乘法，这种方法还行



# 更快的乘法器

## ■ 使用多个加法器

### ■ 权衡性价比



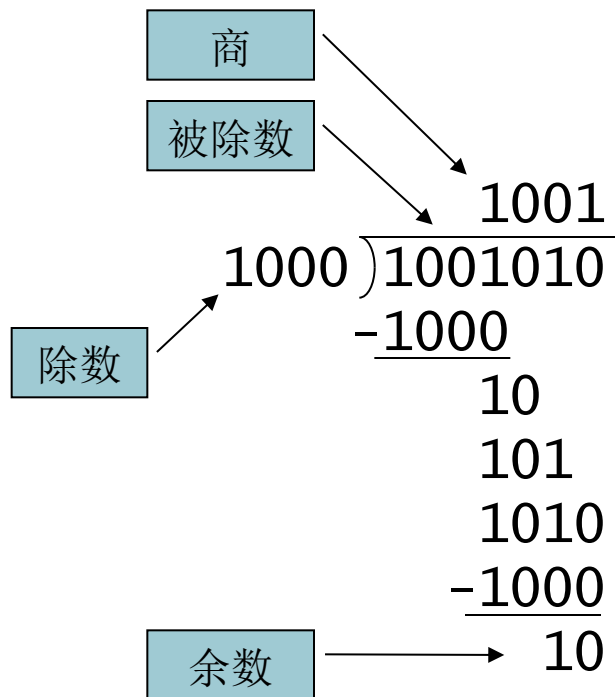
## ■ 可流水线化

### ■ 几个乘法并行执行

# RISC-V乘法

- 4条乘法指令：
  - mul: 乘
    - 给出乘积的低64位
  - mulh: 乘积高位
    - 假设操作数都是有符号数，给出乘积的高64位
  - mulhu: 无符号乘积高位
    - 假设操作数都是无符号数，给出乘积的高64位
  - mulhsu: 有符号/无符号乘积高位
    - 假设一个操作数是有符号数而另一个是无符号数，给出乘积的高64位
- 用mulh/mulhu的结果来检查64位的溢出

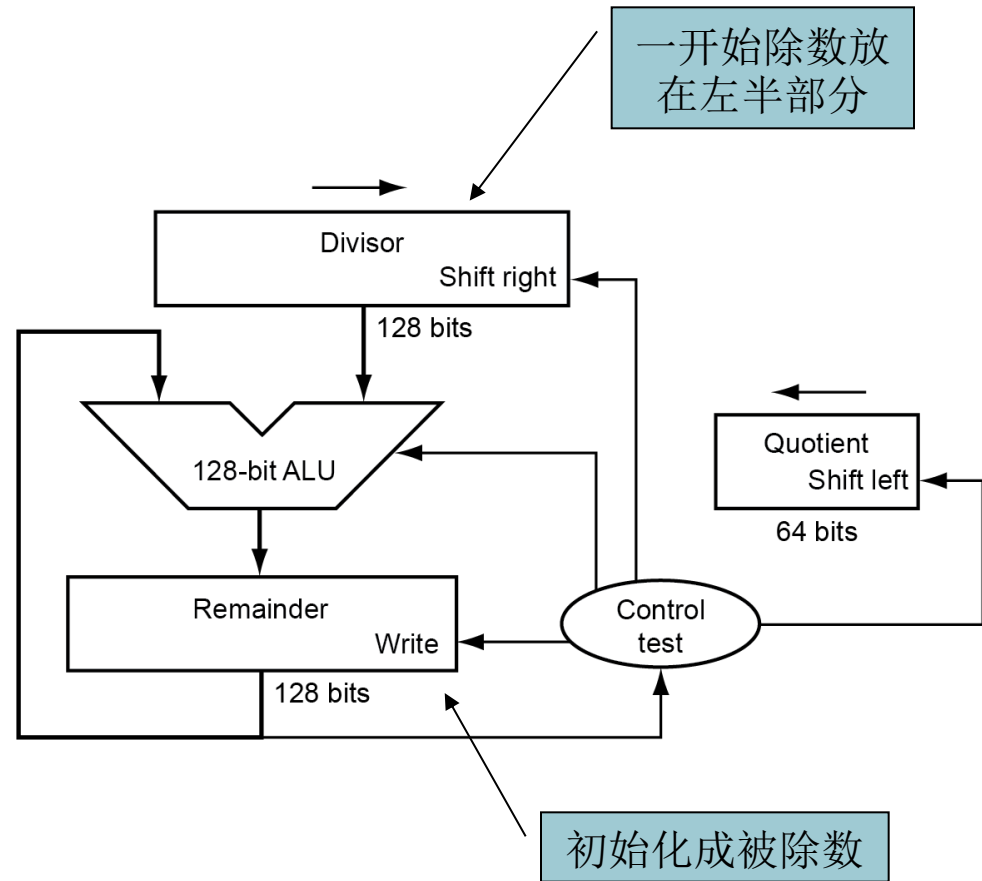
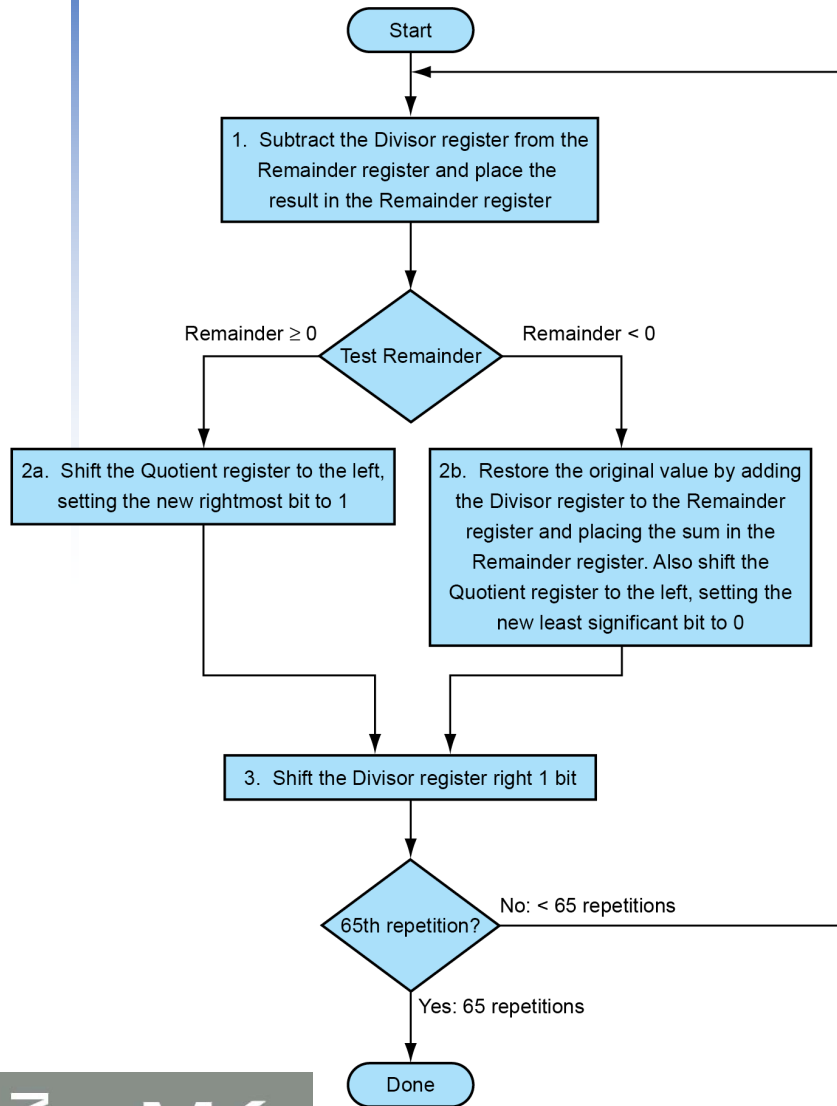
# 除法



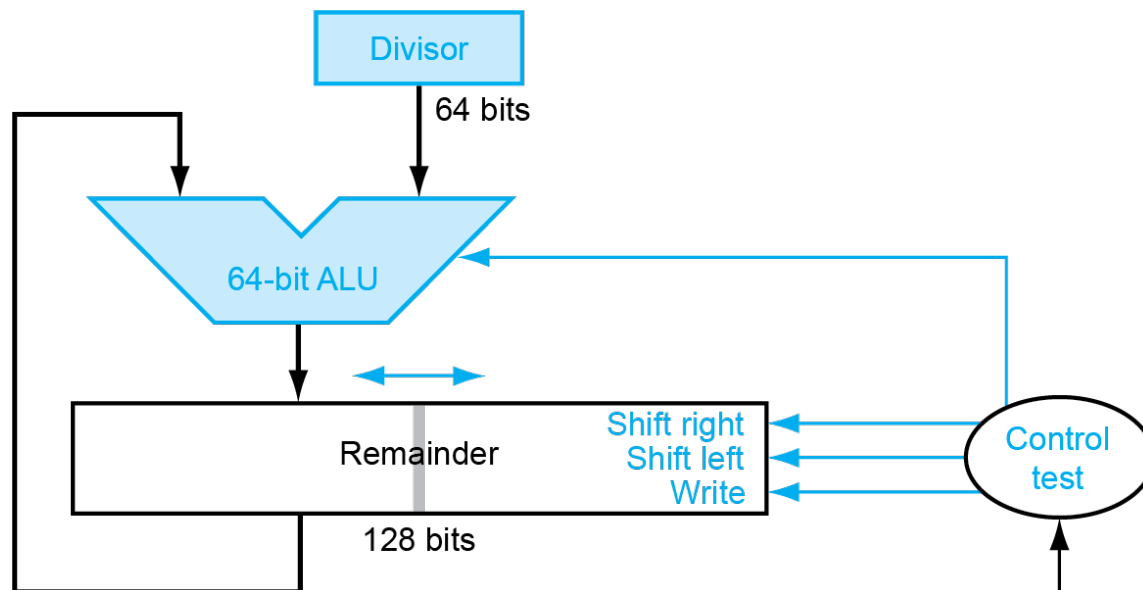
$n$ 位操作数生成  
 $n$ 位商和余数

- 检查0除数
- 长除法的方法
  - 如果除数 $\leq$ 被除数的当前位
    - 商的当前位置1，减去除数
  - 否则
    - 商的当前位置0，被除数当前位传递给被除数下一位
- 除法恢复
  - 先做减法，余数小于0则把除数加回去
- 有符号除法
  - 用绝对值相除
  - 按要求调整商和余数的符号

# 除法的硬件实现



# 优化的除法器



- 每次减去部分余数占一个周期
- 看起来很像乘法器！
  - 二者可以用相同的硬件

# 更快的除法

- 不能像乘法器那样使用并行硬件
  - 减法取决于余数的符号
- 更快的除法器（例如**SRT**除法）每步产生多个商位
  - 仍需多个步骤

# RISC-V除法

- 4条指令：
  - div, rem: 有符号数除法、求余数
  - divu, remu: 无符号数除法、求余数
- 溢出和除0不产生错误
  - 只返回预先定义的结果
  - 对于通常无错误的情况执行更快

# 浮点运算

- 表示非整数数字
  - 包括很小或很大的数
- 类似科学记数法
  - $-2.34 \times 10^{56}$  ← 规格化
  - $+0.002 \times 10^{-4}$  ← 非规格化
  - $+987.02 \times 10^9$  ← 非规格化
- 用二进制表示
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- C语言中的float和double型



# 浮点标准

- 由IEEE 754-1985标准定义
  - 后来有754-2008版本和754-2019修订版
- 因表达方式有差异而制定
  - 科学计算代码的可移植性问题
- 现在几乎全球通用
- 两种表述
  - 单精度（32位）
  - 双精度（64位）

# IEEE浮点格式

单精度：8位

双精度：11位

单精度：23位

双精度：52位

S	指数	尾数
---	----	----

$$x = (-1)^S \times (1 + \text{尾数}) \times 2^{(\text{指数} - \text{偏阶})}$$

- S：符号位（0  $\Rightarrow$  非负数，1  $\Rightarrow$  负数）
- 规格化的有效数字： $1.0 \leq |\text{有效数字}| < 2.0$ 
  - 小数点前总有一位1，因而不必显式地表达（隐含位）
  - 有效数字就是恢复了“1.”的尾数
- 指数：偏置表述——实际指数+偏阶
  - 确保指数为无符号数
  - 单精度：偏阶=127；双精度：偏阶=1023

# 单精度范围

- 指数00000000和11111111为保留值
- 最小值
  - 指数: 00000001  
 $\Rightarrow$  实际指数 =  $1 - 127 = -126$
  - 尾数: 000...00  $\Rightarrow$  有效数字 = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- 最大值
  - 指数: 11111110  
 $\Rightarrow$  实际指数 =  $254 - 127 = +127$
  - 尾数: 111...11  $\Rightarrow$  有效数字  $\approx 2.0$
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

# 双精度范围

- 指数0000...00和1111...11为保留值
- 最小值
  - 指数: 00000000001  
 $\Rightarrow$  实际指数 =  $1 - 1023 = -1022$
  - 尾数: 000...00  $\Rightarrow$  有效数字 = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- 最大值
  - 指数: 111111111110  
 $\Rightarrow$  真实指数 =  $2046 - 1023 = +1023$
  - 尾数: 111...11  $\Rightarrow$  有效数  $\approx 2.0$
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

# 浮点精度

- 相对精度
  - 所有尾数位都有权重
  - 单精度：约 $2^{-23}$ 
    - 相当于 $23 \times \log_{10} 2 \approx 23 \times 0.3 \approx 6$ 位十进制数的精度
  - 双精度：约 $2^{-52}$ 
    - 相当于 $52 \times \log_{10} 2 \approx 52 \times 0.3 \approx 16$ 位十进制数的精度

# 浮点的例子

- 表示-0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - $S = 1$
  - 尾数 =  $1000...00_2$
  - 指数 =  $-1 + \text{偏阶}$ 
    - 单精度:  $-1 + 127 = 126 = 01111110_2$
    - 双精度:  $-1 + 1023 = 1022 = 01111111110_2$
- 单精度:  $10111111101000...00$
- 双精度:  $101111111111101000...00$

# 浮点的例子

- 哪个数字的单精度浮点表示是  
 $11000000101000...00$ 
  - $S = 1$
  - 尾数 =  $01000...00_2$
  - 指数 =  $10000001_2 = 129$
- $$\begin{aligned}x &= (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)} \\&= (-1) \times 1.25 \times 2^2 \\&= -5.0\end{aligned}$$

# 浮点加法

- 考虑一个4位十进制的例子
  - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- 1. 对齐十进制小数点
  - 右移指数小的数
  - $9.999 \times 10^1 + 0.016 \times 10^1$
- 2. 将有效数字相加
  - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
- 3. 规格化结果并检查上溢/下溢
  - $1.0015 \times 10^2$
- 4. 舍入，如有需要再次规格化
  - $1.002 \times 10^2$



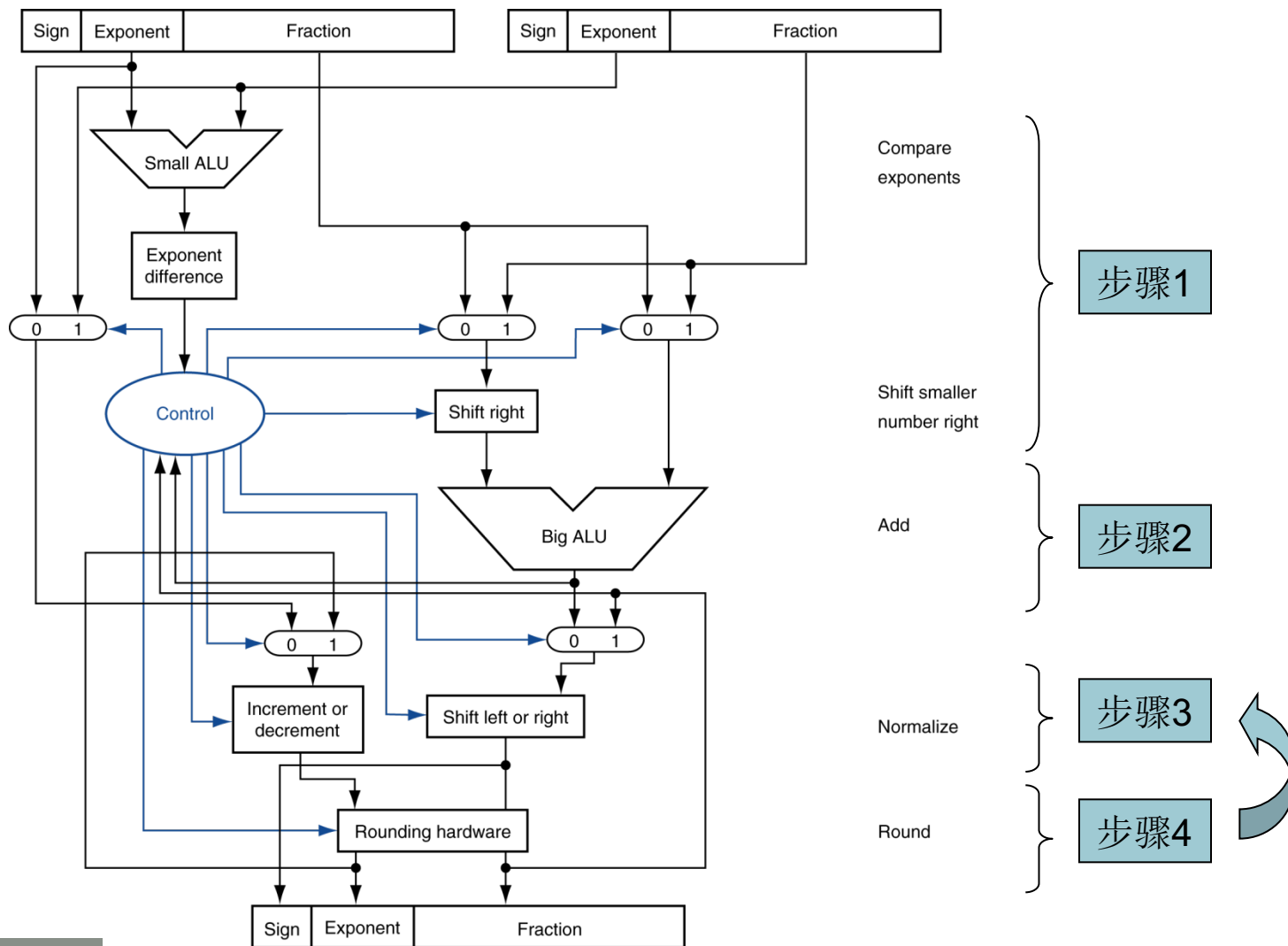
# 浮点加法

- 现在考虑一个4位二进制的例子
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$  ( $0.5 + -0.4375$ )
- 1. 对齐二进制小数点
  - 右移指数小的数
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. 将有效数字相加
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. 规格化结果并检查上溢/下溢
  - $1.000_2 \times 2^{-4}$ , 无上/下溢
- 4. 舍入, 如有需要再次规格化
  - $1.000_2 \times 2^{-4}$  (不变) = 0.0625

# 浮点加法器的硬件实现

- 比整数加法器复杂得多
- 在一个时钟周期内完成太过耗时
  - 比整数运算的时间长得多
  - 更慢的时钟会拖累所有指令
- 浮点加法器通常占几个周期
  - 能流水线化

# 浮点加法器的硬件



# 浮点算术的硬件实现

- 浮点乘法器与浮点加法器的复杂程度近似
  - 但用一个乘法器而非加法器处理有效数字
- 浮点算术的硬件通常用于进行
  - 加法、减法、乘法、除法、倒数、平方根运算
  - 浮点 $\leftrightarrow$ 整数转换
- 运算通常占几个周期
  - 能流水线化

# RISC-V的浮点指令

- 独立的浮点寄存器：f0, ..., f31
  - 双精度
  - 单精度数值保存在低32位中
- 浮点指令只对浮点寄存器进行运算
  - 程序一般不对浮点数据做整数操作，反之亦然
  - 寄存器的增多对代码大小略有影响
- 浮点取数和存数指令
  - flw, fld
  - fsw, fsd

# RISC-V的浮点指令

- 单精度算术运算
  - `fadd.s`, `fsub.s`, `fmul.s`, `fdiv.s`, `fsqrt.s`
    - 例如, `fadd.s f2, f4, f6`
- 双精度算术运算
  - `fadd.d`, `fsub.d`, `fmul.d`, `fdiv.d`, `fsqrt.d`
    - 例如, `fadd.d f2, f4, f6`
- 单/双精度的比较大小
  - `feq.s`, `flt.s`, `fle.s`
  - `feq.d`, `flt.d`, `fle.d`
  - 比较结果为0或1, 保存在整数目的寄存器中
    - 例如, `feq.s x5, f0, f1`
    - 使用`beq`, `bne`根据比较结果进行跳转
- ~~根据浮点条件码的true或false进行跳转~~
  - ~~`B.cond`~~

# 浮点的例子：将°F转换为°C

- C代码：

```
float f2c (float fahr) {  
    return ((5.0/9.0)*(fahr - 32.0));  
}
```

- `fahr`保存在f10中，结果保存在f10中，数字保存在全局存储空间中

- 编译后的RISC-V代码：

f2c:

```
f1w    f0,const5(x3)    # f0 = 5.0f  
f1w    f1,const9(x3)    # f1 = 9.0f  
fdiv.s f0, f0, f1       # f0 = 5.0f / 9.0f  
f1w    f1,const32(x3)   # f1 = 32.0f  
fsub.s f10,f10,f1       # f10 = fahr - 32.0  
fmul.s f10,f0,f10       # f10 = (5.0f/9.0f)*(fahr-32.0f)  
jalr   x0,0(x1)         # return
```

# 浮点的例子：数组乘法

- $C = C + A \times B$

- 均为 $32 \times 32$ 矩阵，64位双精度矩阵元素

- C代码：

```
void mm (double c[][],  
         double a[][], double b[][]) {  
    size_t i, j, k;  
    for (i = 0; i < 32; i = i + 1)  
        for (j = 0; j < 32; j = j + 1)  
            for (k = 0; k < 32; k = k + 1)  
                c[i][j] = c[i][j]  
                    + a[i][k] * b[k][j];  
}
```

- c, a, b的地址保存在x10, x11, x12中，i, j, k保存在x5, x6, x7中



# 浮点的例子：数组乘法

RISC-V代码：

mm:...

```
        li    x28,32      # x28 = 32 (row size/loop end)
        li    x5,0        # i = 0; initialize 1st for loop
L1:     li    x6,0        # j = 0; initialize 2nd for loop
L2:     li    x7,0        # k = 0; initialize 3rd for loop
        slli  x30,x5,5     # x30 = i * 2**5 (size of row of c)
        add   x30,x30,x6   # x30 = i * size(row) + j
        slli  x30,x30,3    # x30 = byte offset of [i][j]
        add   x30,x10,x30  # x30 = byte address of c[i][j]
        fld   f0,0(x30)    # f0 = c[i][j]
L3:     slli  x29,x7,5     # x29 = k * 2**5 (size of row of b)
        add   x29,x29,x6   # x29 = k * size(row) + j
        slli  x29,x29,3    # x29 = byte offset of [k][j]
        add   x29,x12,x29  # x29 = byte address of b[k][j]
        fld   f1,0(x29)    # f1 = b[k][j]
```

# 浮点的例子：数组乘法

...

```
slli    x29,x5,5      # x29 = i * 2**5 (size of row of a)
add     x29,x29,x7     # x29 = i * size(row) + k
slli    x29,x29,3      # x29 = byte offset of [i][k]
add     x29,x11,x29    # x29 = byte address of a[i][k]
fld     f2,0(x29)      # f2 = a[i][k]
fmul.d  f1, f2, f1      # f1 = a[i][k] * b[k][j]
fadd.d  f0, f0, f1      # f0 = c[i][j] + a[i][k] * b[k][j]
# fmadd.d f0, f2, f1, f0可替换之前的fmul.d、fadd.d两条指令
addi    x7,x7,1        # k = k + 1
bltu    x7,x28,L3      # if (k < 32) go to L3
fsd     f0,0(x30)      # c[i][j] = f0
addi    x6,x6,1        # j = j + 1
bltu    x6,x28,L2      # if (j < 32) go to L2
addi    x5,x5,1        # i = i + 1
bltu    x5,x28,L1      # if (i < 32) go to L1
```

# 算术精确性

- IEEE 754标准规定了附加的舍入控制
  - 为精度多保留的位（保护位guard、舍入位round、粘性位sticky）
  - 舍入模式的选项（754-2008定义了5种）
    - 舍入到最近的偶数，舍入到最近的最大幅值
    - 向0舍入，向 $-\infty$ 舍入，向 $+\infty$ 舍入
  - 允许程序员精调计算中的数值行为
- 不是每个浮点单元都能实现全部选项
  - 多数编程语言和浮点库仅使用默认选项
- 在硬件复杂度、性能和市场需求间权衡

# 子字并行

- 图像和音频应用程序能够利用对短向量的同时运算
  - 例：128位加法器：
    - 16个8位加
    - 8个16位加
    - 4个32位加
- 亦称数据级并行、向量并行或SIMD (Single Instruction, Multiple Data, 单指令多数据)

# x86浮点体系结构

- 起初基于8087浮点协处理器
  - 8个80位的扩展精度寄存器
  - 用作一个向下压入的栈
  - 寄存器从栈顶开始编号：ST(0), ST(1), ...
- 浮点数值是内存中的32位或64位
  - 在存取内存操作数时进行转换
  - 整型操作数也可在存取时转换
- 很难生成和优化代码
  - 后果：浮点性能差劲

# x86浮点指令

数据传输	算术运算	比较	超越函数
FILD mem/ST(i) FISTP mem/ST(i) FLDPI FLD1 FLDZ	FIADDP mem/ST(i) FISUBRP mem/ST(i) FIMULP mem/ST(i) FIDIVRP mem/ST(i) FSQRT FABS FRNDINT	FICOMP FIUCOMP FSTSW AX/mem	FPATAN F2XMI FCOS FPTAN FPREM FPSIN FYL2X

- 可选变体
  - I: 整型操作数
  - P: 从栈弹出操作数
  - R: 反转操作数顺序
  - 不过有的组合不允许使用

## 第2代流式SIMD扩展（SSE2）

- 增加了4个128位寄存器
  - 在AMD64/EM64T中扩展到8个寄存器
- 可用于多个浮点操作数
  - 2个64位双精度
  - 4个32位单精度
  - 指令同时计算这些操作数
    - Single-Instruction Multiple-Data

# 矩阵乘法

## ■ 未经优化的代码：

```
1. void dgemm (int n, double* A, double* B, double* C)
2. {
3.   for (int i = 0; i < n; ++i)
4.     for (int j = 0; j < n; ++j)
5.       {
6.         double cij = C[i+j*n]; /* cij = C[i][j] */
7.         for(int k = 0; k < n; k++ )
8.           cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
9.         C[i+j*n] = cij; /* C[i][j] = cij */
10.      }
11. }
```



# 矩阵乘法

## ■ x86汇编代码:

```
1. vmovsd (%r10),%xmm0    # Load 1 element of C into %xmm0
2. mov %rsi,%rcx           # register %rcx = %rsi
3. xor %eax,%eax          # register %eax = 0
4. vmovsd (%rcx),%xmm1    # Load 1 element of B into %xmm1
5. add %r9,%rcx           # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1,
   element of A
7. add $0x1,%rax          # register %rax = %rax + 1
8. cmp %eax,%edi          # compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
10. jg 30 <dgemm+0x30>    # jump if %eax > %edi
11. add $0x1,%r11d        # register %r11 = %r11 + 1
12. vmovsd %xmm0, (%r10)  # Store %xmm0 into C element
```

# 矩阵乘法

## ■ 优化后的C代码:

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.     for ( int i = 0; i < n; i+=4 )
5.         for ( int j = 0; j < n; j++ ) {
6.             __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j]
               */
7.             for( int k = 0; k < n; k++ )
8.                 c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.                                     _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.                                                    _mm256_broadcast_sd(B+k+j*n)));
11.             _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.         }
13. }
```

# 矩阵乘法

## ■ 优化后的x86汇编代码:

```
1. vmovapd (%r11),%ymm0      # Load 4 elements of C into %ymm0
2. mov %rbx,%rcx              # register %rcx = %rbx
3. xor %eax,%eax              # register %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
5. add $0x8,%rax              # register %rax = %rax + 8
6. vmulpd (%rcx),%ymm1,%ymm1 # Parallel mul %ymm1, 4 A elements
7. add %r9,%rcx               # register %rcx = %rcx + %r9
8. cmp %r10,%rax              # compare %r10 to %rax
9. vaddpd %ymm1,%ymm0,%ymm0   # Parallel add %ymm1, %ymm0
10. jne 50 <dgemm+0x50>        # jump if not %r10 != %rax
11. add $0x1,%esi             # register % esi = % esi + 1
12. vmovapd %ymm0, (%r11)     # Store %ymm0 into 4 C elements
```

# 右移与除法

- 左移 $i$ 位相当于乘以常数 $2^i$
- 右移就是除以 $2^i$ 吗？
  - 仅对无符号整数成立
- 对于有符号整数
  - 算术右移：复制符号位
  - 例如， $-5 / 4$ 
    - $11111011_2 \gg 2 = 11111110_2 = -2$
    - 向 $-\infty$ 方向舍入
  - 对比 $11111011_2 \ggg 2 = 00111110_2 = +62$

# 结合律

- 并行程序可能按不可预料的顺序进行多个操作
  - 结合律可能失效

		$(x+y)+z$	$x+(y+z)$
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- 需要在变化的并行度下验证并行程序

# 谁关心浮点精度？

- 精度对科学计算程序很重要
  - 但对消费者的日常使用呢？
    - “我的银行余额少了0.0002¢！” 😞
- Intel Pentium FDIV漏洞
  - 消费市场也同样关心精度
  - 参见Colwell, *The Pentium Chronicles*

# 本章小结

- 数字位没有固定的内在含义
  - 如何诠释取决于使用的指令
- 数字的计算机表述
  - 精度、范围有限
  - 在程序中需要考虑这一点

# 本章小结

- ISA支持算术运算
  - 有符号和无符号整数
  - 对实数的浮点近似
- 有限的范围和精度
  - 运算可能上溢或下溢