



北京邮电大学

BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS



# Computer Organization and Architecture

## Chapter 9

## Computer Arithmetic

School of Computer Science (National Pilot Software Engineering School)

AO XIONG (熊翱)

xiongao@bupt.edu.cn





# 期中考试提醒

---

- 时间：2023年11月18日（周六），上午9：00~11：00
- 地点：教三楼535
- 试卷语言：英语，重点词有注释。中文答题
- 考试题型：
  - 判断题
  - 选择题 (Multiple choices, 单选题)
  - 简答题
  - 问答题
- 考试形式：闭卷
- 考试用具：身份证（学生证，学生卡），签字笔，铅笔，直尺，橡皮等



# Assignment- 1

7.13考虑一个系统,其总线周期为500ns。无论是从处理器到DMA模块,还是从DMA模块到处理器,总线控制的传递都用250ns。一个I/O设备使用DMA方式,其数据传输率是50KB/s。数据每次传送一个字节。

(a) 若使用突发式DMA,即数据块传送之前DMA模块获得总线控制权,并一直维持对总线的控制,直到整个数据块传输完毕。当传送128字节的数据块时,设备占用总线多长的时间?

(b) 若使用周期窃取式DMA,重复上问。



# Assignment- 1

(a) 数据传输需要  $\frac{128B}{50KB/S} = 2.56ms$

传输开始和结束时分别需要250ns进行总线控制的传递

250ns相对于2.56ms可以忽略

因此使用突发式DMA设备占用总线约2.56ms

(b) 使用周期窃取DMA时传输一个字节需要占用的总线是：  $250ns + 250ns + 500ns = 1\mu s$

传送128字节的数据块时设备占用总线约  $128\mu s$



## Assignment- 2

8.6假设处理器当前正在执行的进程的页表如下,所有数据是十进制,用数字表示每个事情均从0开始,所有地址都是存储器的字节地址,一个页的大小为1024B。

(a)准确描述CPU生成的虚拟地址如何转换成主存的物理地址。

(b)虚拟地址:(I)1052,(II)2221,(III)5499对应的物理地址是什么(不考虑页故障)?

Virtual page number	Valid bit	Reference bit	Modify bit	Page frame number
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0



# Assignment- 2

(a) 虚拟地址由虚页号和页内偏移组成，确定虚页号后通过页表将虚页号映射为页帧号，将页帧号与页内偏移组合得到物理地址。

(b) 虚拟地址和物理地址的转换

- I. 因为一个页的大小为1024B,  $1024 < 1052 < 2048$ , 可知1052的虚页号为1, 页内偏移为 $1052 - 1024 = 28$ , 通过页表映射, 页帧号为7, 物理地址为 $1024 * 7 + 28 = 7196$
- II. 同理, 2221的虚页号为2, 查页表可知, 页缺失。
- III. 5499的虚页号为5, 页内偏移为 $5499 - 1024 * 5 = 379$ , 查页表可知映射后的页帧号为0, 物理地址为 $0 * 1024 + 379 = 379$



# Assignment- 2

## 方法2:

I. 1052的二进制: 0001 0000 0111 00

页号: 0001, 页内地址: 0000 0111 00。虚拟页号是0001, 对应页帧是0111。物理地址是: 0111 0000 0111 00

I. 2221的二进制是: 0010 0010 1011 01。虚拟页号是0010, 页表该行无效。页缺失。

II. 5499的二进制是: 0101 0101 1110 11。虚拟页号是0101, 对应页帧是0。物理地址位: 0000 0101 1110 11, 也就是379。



# Assignment- 3

6.3 考虑一个有8个面的磁盘驱动器，每面有512个磁道，每道上有64个扇区，扇区大小为1KB。平均寻道时间是8ms,道间移动时间是1.5ms,磁盘转速为3600rpm。可以读取同一柱面上的连续磁道而磁头不需要移动。

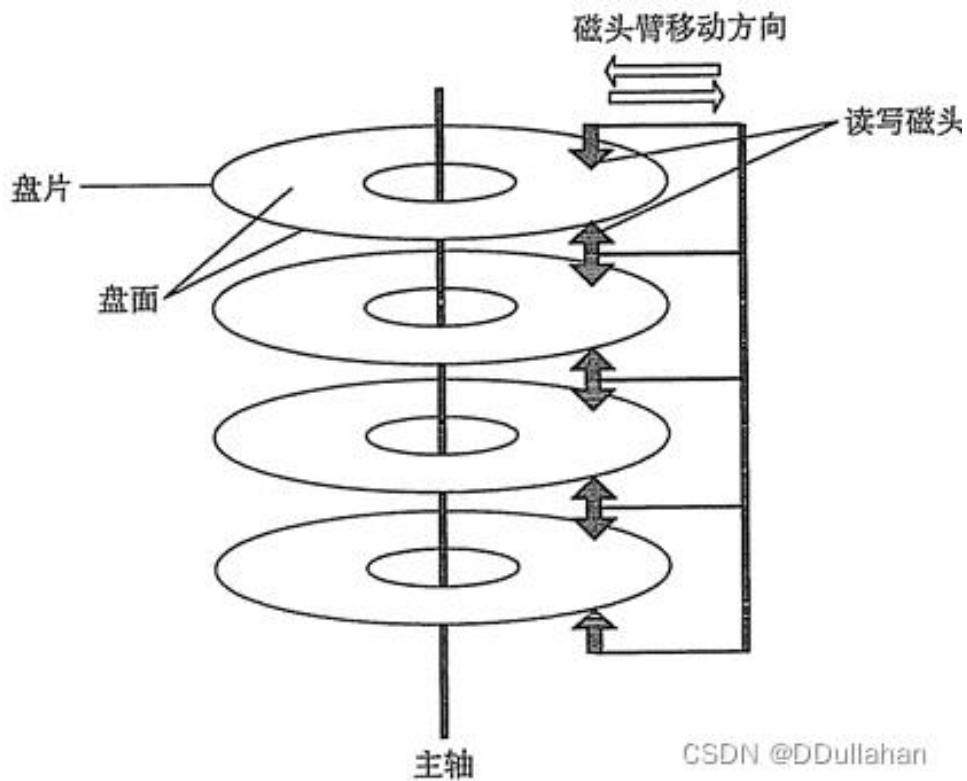
(a)磁盘容量是多少？

(b)平均存取时间是多少？假设某文件被存储在连续柱面的连续扇区和连续磁道上，起始位置为柱面i上第0道的第0号扇区。

(c)估计传送5MB大小的文件所需要的时间。

(b)突发传送率是多少？

# Assignment- 3



- 磁头同步读写对技术要求高
- 每个磁盘独立旋转
- 多盘片的用途主要是提高容量
- 节省了磁头臂的移动时间



# Assignment- 3

(c) 5MB需要 $\frac{5MB}{64KB} = 80$ 个磁道

由于总共有8个盘，所以总共是 $\frac{80}{8} = 10$ 个柱面

传送文件所需时间主要包括寻道时间，旋转延迟，读取时间，道间移动时间

读取时间为每个柱面 $\frac{60}{3600} * 8 = 133.3ms$

传送5MB大小文件所需时间为

$$(8 + (8.3 * 8 + 133.3)) + 9 * (8.3 * 8 + 133.3 + 1.5) = 2018.5ms$$

(d) 突发传送率为

$$\frac{3600}{60} * 64 * 1KB = 3.84MB/S$$

其余柱面

第一个柱面

突发数据传输率(Burst data transfer rate)指的是通过数据总线从硬盘中所读取数据的最高速率



# Preface

## We have learned:

- Basic Concepts and Computer Evolution 基本概念和计算机发展历史
- Performance Issues 性能问题
- Top level view of computer function and interconnection 计算机功能和互联结构顶层视图
- Cache Memory cache存储器
- Internal Memory 内部存储器
- External Memory 外部存储器
- Input& Output 输入输出
- Operating System Support 操作系统支持
  - Operating system overview 操作系统概览
  - Scheduling 调度
  - Memory management 内存管理



# Review

---

- 问题1：采用交换技术之后，为什么需要用虚拟地址？
- 问题2：内存采用分段+分页后，如何得到存储的物理地址？
- 问题3：简单描述一下普通页表和反向页表



# Preface

---

## Next

- Arithmetic and Logic 算术与逻辑
  - Number System 数字系统
  - Computer Arithmetic 计算机算术
  - Digital Logic 数字逻辑



# Preface

**We will focus the following contents today:**

- Computer Arithmetic 计算机算术
  - How is integer stored in computers? 整数如何存储?
  - How to realize integer operation in computer? Especially how to do arithmetic operation quickly? 如何实现整数运算? 特别是如何快速进行运算?
  - How is floating point number stored in computer? 浮点数如何存储?
  - How to realize floating point number operation in computer? 浮点数如何运算?



# Outline

---

- The Arithmetic and Logic Unit (ALU) 算术逻辑单元
- Integer Representation 整数表示
- Integer Arithmetic 整数的算术运算
- Floating-Point Representation 浮点数表示
- Floating-Point Arithmetic 浮点数运算

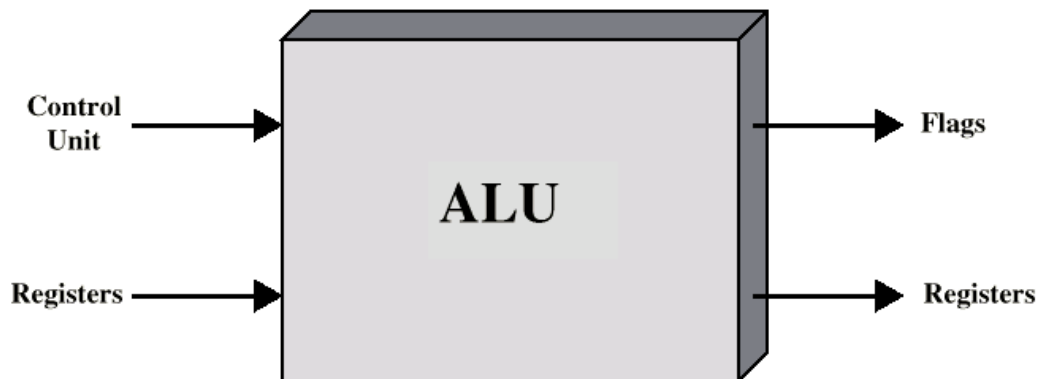


# Arithmetic & logic unit 算术逻辑单元

- Core of computer 计算机的核心部件
- Everything else in the computer is there to service this unit 计算机的所有其他部件均为它服务
- Does arithmetic and logic calculations 完成算术和逻辑计算
  - Handles integers 处理整数
  - May handle floating point (real) numbers 可能会处理浮点数
  - May be separate FPU ( maths co-processor) 可能有独立的浮点数运算单元，数学协处理器
  - May be on chip separate FPU ( 486DX + ) 可能有片上的独立浮点运算单元



# ALU input & output **ALU的输入和输出**



- 一般CPU内部会有一组寄存器，用于临时存放数据
- 控制单元告诉ALU需要做什么操作，同时还控制数据的输入和输出
- 数据由寄存器送给ALU进行运算，运算之后的结果也保存在寄存器中
- ALU在计算后，会设置一些标志，比如溢出标志等，标志也保存在寄存器中



# Outline

---

- The Arithmetic and Logic Unit (ALU) 算术逻辑单元
- Integer Representation 整数表示
- Integer Arithmetic 整数的算术运算
- Floating-Point Representation 浮点数表示
- Floating-Point Arithmetic 浮点数运算



# Integer representation 整数表示

- Electronic components generally have only two basic states 电子元器件一般只有两种基本的状态
  - Whether there is charge, high and low level 有无电荷，高低电平
  - Represents 0 and 1 表示0和1
- Computer use 0 & 1 to represent everything 计算机用0和1来表示所有的东西
  - Positive numbers stored in binary 只能用二进制存储的正数
  - e.g.  $41 = 00101001$
  - No minus sign 没有负数符号
  - No period 没有分隔符
- How to represent negative numbers 如何表示负数
  - Sign-Magnitude 符号幅值表示
  - Two's complement 补码表示



# Decimal and binary 十进制和二进制

- TO convert N from Decimal to Binary 将一个10进制数N转换为2进制
  - Integer: to divide the decimal number by 2 repeatedly 整数: 不停地除以2
  - Fraction: to multiply the decimal number by 2 repeatedly 小数: 不停地乘以2



# Example

---

- $(105.4)_{10} = (? )_2$



# Example

- $(105.4)_{10} = (1101001.011)_2$

Integer :

From 0 to the left

$$105/2 = 52 \text{ .....}1$$

$$52/2 = 26 \text{ .....}0$$

$$26/2 = 13 \text{ .....}0$$

$$13/2 = 6 \text{ .....}1$$

$$6/2 = 3 \text{ .....}0$$

$$3/2 = 1 \text{ .....}1$$

$$1/2 = 0 \text{ .....}1$$

Fraction

From 0 to the right

$$0.4 \times 2 = 0.8 \text{ .....}0$$

$$0.8 \times 2 = 1.6 \text{ .....}1$$

$$0.6 \times 2 = 1.2 \text{ .....}1$$

...



# Decimal and binary 十进制和二进制

- TO convert Binary to Decimal 将二进制转换为十进制

- The decimal value of an n-bit integer with m bit **fraction**  
n位二进制转换为十进制

- Integer: from center to left:

整数部分：从中心往左

$$\sum_{i=0}^{n-1} b_i 2^i$$

- Fraction: from center to right:

小数部分：从中心往右

$$\sum_{i=1}^m b_i 2^{-i}$$



# Example

---

$$(1101001.011)_2 = ( ? )_{10}$$



# Example

$$\begin{aligned}(1101001.011)_2 &= \\&= (2^6 + 2^5 + 2^3 + 2^0) \cdot (2^{-2} + 2^{-3}) \\&= (64 + 32 + 8 + 1) \cdot (0.25 + 0.125) \\&= 105.375 \text{ ----- } \mathbf{(105.4)_{10}}\end{aligned}$$

- 二进制转换成十进制，不会有误差
- 十进制小数转换成二进制，可能会有误差



# Integer representations 整数表示

- unsigned 无符号数
  - Use only non-negative integers 只用非负整数
  - sign bit does not need 不需要符号位
- sign magnitude 符号+幅值
  - Sign +magnitude 符号位+数值
- one 's complement 反码
  - represent the value by use inverse value of Sign +magnitude 用符号-幅值表示法中的数进行取反
- two 's complement 补码
  - Use two 's complement to express integer 用补码来表示整数
- biased 偏移



# Unsigned representations 无符号表示法

- By unsigned, we mean no negative values 因为没有符号，所以只有非负值
  - E.g. 0, 1, 2, ..., 254, 255, 256, 257, .... 65535, 65536, 65537, ...
- The range an N bit number can represent N位值的范围
  - 0 to  $2^n - 1$
- A Byte of 8 bits can store unsigned integers from 8位的字节可以表示的无符号整数范围是
  - 0 to 255 =  $2^8 - 1$ .
  - The largest value is  $1111\ 1111_2 = 255$ , and the smallest is 0 最大是255，最小是0



# Sign-magnitude 符号+幅值

- Left most bit is sign bit 最左边的是符号位
- 0 means positive 0 表示正数
- 1 means negative 1 表示负数
- +18 = 00010010
- -18 = 10010010

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases}$$



# Sign-magnitude 符号+幅值

- Range:  $-(2^{n-1} - 1)$  to  $(2^{n-1} - 1)$  where  $n$  is the number of bits  $n$ 位的  
的符号-幅值表示法的数值的范围是 $-(2^{n-1} - 1)$  到  $(2^{n-1} - 1)$ 
  - 4 bits, -7 to +7
- Problems 问题
  - Need to consider both sign and magnitude in arithmetic 算术计算  
中需要考虑符号和幅值
  - Two representations of zero (+0 and -0) 0有两种表示方法: +0和-0
    - 10000000 -0
    - 00000000 +0
  - Two methods are required to judge 0 判断0需要有两种方法



# One's complement 反码

- Historically important, and we use this representation to get 2's complement integers 历史上很重要，而且用反码来得到补码表示法
- positive integers use the same representation as unsigned 正整数使用和无符号相同的表示
- Negation is done by taking a bitwise complement of the positive representation 负数通过按位取反得到



# One's complement 反码

- -3: 0011  $\rightarrow$  1100
- 11100000: a negative number 负数
  - To find out the value, invert each bit 00011111 is +31 by sight, so  
11100000 = -31 为了得到值，取反所有的位，得到31，所以是-31
- The 1's complement number system using N bits has a range from  $-(2^{N-1} - 1)$  to  $+(2^{N-1} - 1)$  反码表示法中，N位的范围是 $-(2^{N-1} - 1)$  到  $+(2^{N-1} - 1)$



# Example of one's complement 反码举例

- 8-bit examples 8位的例子
  - $1111\ 1110 = -1$
  - $1111\ 1111 = -0$
  - $0000\ 0000 = +0$
  - $0000\ 0001 = +1$
- The two forms of zero are represented by 2种0的表示法
  - $0000\ 0000$  (all zeros) 全0
  - $1111\ 1111$  (all ones) 全1



# Two's complement 补码

- Two's complement is a variation on 1's complement  
补码是反码的一个变种
  - Does NOT have 2 representations for 0 对于0只有1种表示法
- negative: (for example -5) 负数
  - Take the positive value 00101 (+5) 先得到正数
  - Take the 1's complement 11010 (-5 in 1's comp) 得到它的反码
  - Add 1 : 11011 加1

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$



# Benefits 优点

- One representation of zero 0只有1种表示法
- Arithmetic works easily (see later) 算术计算更容易
- Negating is fairly easy 取负数容易得到
- Example: -3

1.  $(3)_{10} = (00000011)_2$

2. Boolean complement gives  $11111100$

3. Add 1 to LSB  $11111101$



# Negation Special Case 1

## 负数的特殊例子1

- $0 =$  00000000
- Bitwise not 11111111
- Add 1 to LSB +1
- Result 1 00000000
- Overflow is ignored, so:
- $-0 = 0$  ✓



## Negation Special Case 2 负数的特殊例子2

- $-128 = 10000000$
- $-(-128)$  is:
  - bitwise not  $01111111$  按位取反
  - Add 1 to LSB  $+1$
  - Result  $10000000$
- So:
  - $-(-128) = -128$  X
  - In two 's complement notation, the range of positive and negative numbers is not completely symmetric 补码表示法中，正数和负数的范围不是完全对称的



# Range of numbers 补码数的范围

- 8 bit 2s compliment 8位补码数
  - $+127 = 01111111 = 2^7 - 1$
  - $-128 = 10000000 = -2^7$
- 16 bit 2s compliment 16位补码数
  - $+32767 = 01111111 11111111 = 2^{15} - 1$
  - $-32768 = 10000000 00000000 = -2^{15}$



## Example 举例

- A binary number 1010 1001 is given. Find out the corresponding decimal value if it is 一个二进制数是10101001，对于各种表示法，它的十进制数是多少？
  - An unsigned number 无符号数
  - A signed number using sign-magnitude representation 符号-幅值法表示的有符号数
  - A signed number using one's complement representation 反码表示法表示的有符号数
  - A signed number using two's complement format 补码表示法表示的有符号数
  - A signed number using Biased representation (B=127) 偏移量表示法表示的有符号数，偏移量为127



## Example 举例

10101001

- Unsigned: 10101001  
 $=128+32+8+1=169$
- Signed sign-magnitude:  $-(32+8+1)=-41$
- Signed one's complement:  $-(64+16+4+2)=-86$
- Signed two's complement:  $-(1010110+1)=- (1010111)$   
 $-(64+16+4+2+1)=-87$
- Signed Biased:  $(10101001-01111111)$   
 $=00101010$   
 $=42$



# Sign extension -1 符号位扩展1

- For sign magnitude number, simply move the sign bit to the new leftmost position and fill in with zeros 对于符号-幅值表示法的有符号整数，简单的在最高位加上符号位，然后用0填充

➤  $+18 =$  00010010

➤  $+18 =$  00000000 00010010

➤  $-18 =$  10010010

➤  $-18 =$  10000000 00010010



## Sign extension -2 符号位扩展2

- It is not going to work for two's complement 对于补码数，这样扩展不对

➤  $+18 = 00010010$

➤  $+18 = 00000000\ 00010010$

➤  $-18 = 10010010$

➤  $-18 = 10000000\ 00010010 = -32658$  X



## Sign extension -3 符号位扩展3

- The rule for two's complement number is 补码的规则这样
- Positive number pack with leading zeros 正数补0
  - $+18 =$  00010010
  - $+18 =$  00000000 00010010
- Negative numbers pack with leading ones 负数补1
  - $-18 =$  10010010
  - $-18 =$  11111111 10010010



## Example 举例

0000 0000 0000 0000 0000 0000 0000 1011

+ 1011 1010

we must sign extend the second number and then  
add 必须将第二个数进行符号位扩展

0000 0000 0000 0000 0000 0000 0000 1011

+ 1111 1111 1111 1111 1111 1111 1011 1010

-----

1 1111 1111 1111 1111 1111 1111 1100 0101



特点	描述
范围	$-2^{n-1} \sim 2^{n-1}-1$
0的表示方法	只有1个0的表示法
计算方法	按位取反后，在最后一位加1，可以得到这个数的相反数的补码
位的扩展	用符号位进行填充
溢出规则	如果2个相同符号的数相加，得到的新数的符号位和原数的符号位不一样，就是溢出
减法规则	取减数的补码，然后和被减数相加



# Outline

---

- The Arithmetic and Logic Unit (ALU) 算术逻辑单元
- Integer Representation 整数表示
- Integer Arithmetic 整数的算术运算
- Floating-Point Representation 浮点数表示
- Floating-Point Arithmetic 浮点数运算



# Negation 取反

- In sign-magnitude representation 对于符号-幅值表示法  
simple: invert the sign bit. 翻转符号位
- In twos complement notation 对于补码表示法
  - Take the Boolean complement of each bit of the integer (including the sign bit). That is, set each 1 to 0 and each 0 to 1. 先按位取反，包括符号位
  - Treating the result as an unsigned binary integer, add 1 当作无符号整数，加1



# Two's complement operation 补码的取反

- Example

$$\begin{array}{rcl} +18 & = & 00010010 \text{ (twos complement)} \\ \text{bitwise complement} & = & 11101101 \\ & & \underline{+ \quad \quad 1} \\ & & 11101110 = -18 \end{array}$$

$$\begin{array}{rcl} -18 & = & 11101110 \text{ (twos complement)} \\ \text{bitwise complement} & = & 00010001 \\ & & \underline{+ \quad \quad 1} \\ & & 00010010 = +18 \end{array}$$

- The negative of the negative of that number is itself 两次取反后，和原数一致



## Example -1 举例1

- Negation 取反
  - Take the complement of each bit of the integer 按位取反
  - Treating the result as an unsigned binary integer, add 1  
当作无符号数，加1

**-5:**

1. 5 =	0101
2. Complement:	1010
3. Add 1:	1
<hr/>	
4. -5 =	1011

**-7:**

1. 7 =	0111
2. Complement:	1000
3. Add 1:	1
<hr/>	
4. -7 =	1001



## Example -2 举例2

**0:**

1. 0 =	0000
2. Complement:	1111
3. Add 1:	1
<hr/>	
4. 0 =	0000

**-128:**

1. -128 =	10000000
2. Complement:	01111111
3. Add 1:	1
<hr/>	
4. 128 =	10000000

-128的相反数，经过2步，计算得到的数还是10000000，也是-128。所以，128无法表示。  
要注意表示范围的判定



# Addition and Subtraction 加法和减法

- Addition 加法
  - Normal binary addition 正常2进制加法
  - Monitor sign bit for overflow 检查溢出
- Subtraction 减法
  - Take twos compliment of subtrahend and add to minuend 得到减数的补码数，然后加到被减数
  - i.e.  $a - b = a + (-b)$
- So only need addition and complement circuits 只需要加法和取反电路



# Addition 加法

- Straight forward approach consists of adding each bit together from right to left and propagating the carry from one bit to the next 从右到左将每个位相加，并将进位从一位传播到下一位
- Example

$$\begin{array}{r} -7 \quad 1001 \\ 5 \quad 0101 \\ \hline -2 \quad 1110 \end{array}$$

$$\begin{array}{r} 3 \quad 0011 \\ 4 \quad 0100 \\ \hline 7 \quad 0111 \end{array}$$



# Overflow of addition 加法溢出

$$\begin{array}{r} -4 \quad 1100 \\ 4 \quad 0100 \\ \hline 0 \quad (1)0000 \end{array}$$

两个异号的数相加，不会溢出

$$\begin{array}{r} -4 \quad 1100 \\ -1 \quad 1111 \\ \hline -5 \quad (1)1011 \end{array}$$

两个同号的数相加，符号位相同，不溢出

$$\begin{array}{r} -7 \quad 1001 \\ -6 \quad 1010 \\ \hline -13 \quad (1)0011 \end{array}$$

overflow

两个负数相加，符号位为正数，有溢出

$$\begin{array}{r} 5 \quad 0101 \\ 4 \quad 0100 \\ \hline 9 \quad (0)1001 \end{array}$$

overflow

两个正数相加，符号位为负数，有溢出



# Subtraction 减法

- Straight forward approach consists of negating one term and performing integer addition. 对减数求反然后做加法
- 7-6
  - $(7-6)_{10} = (0111-0110)_2$
  - $(6)_{10} = (0110)_2$ ,  $(-6)_{10} = (1001 + 1)_2 = (1010)_2$
  - $(0111 - 0110)_2 = (0111 + 1010)_2 = (0001)_2 = (1)_{10}$
- 6-7
  - $(6-7)_{10} = (0110 - 0111)_2$
  - $(7)_{10} = (0111)_2$ ,  $(-7)_{10} = (1000 + 1)_2 = (1001)_2$
  - $(0110 - 0111)_2 = (0110 + 1001)_2 = (1111)_2 = (-1)_{10}$



# Overflow of subtraction 减法溢出

$$\begin{array}{r} 2 \quad 0010 \\ -7 \quad 1001 \\ \hline -5 \quad 1011 \end{array}$$

$$\begin{array}{r} 5 \quad 0101 \\ -2 \quad 1110 \\ \hline 3 \quad (1)0011 \end{array}$$

$$\begin{array}{r} -5 \quad 1011 \\ -2 \quad 1110 \\ \hline -7 \quad (1)1001 \end{array}$$

$$\begin{array}{r} 5 \quad 0101 \\ -(-2) \quad 0010 \\ \hline 7 \quad 0111 \end{array}$$

$$\begin{array}{r} 7 \quad 0111 \\ -(-7) \quad 0111 \\ \hline (0)1110 \end{array}$$

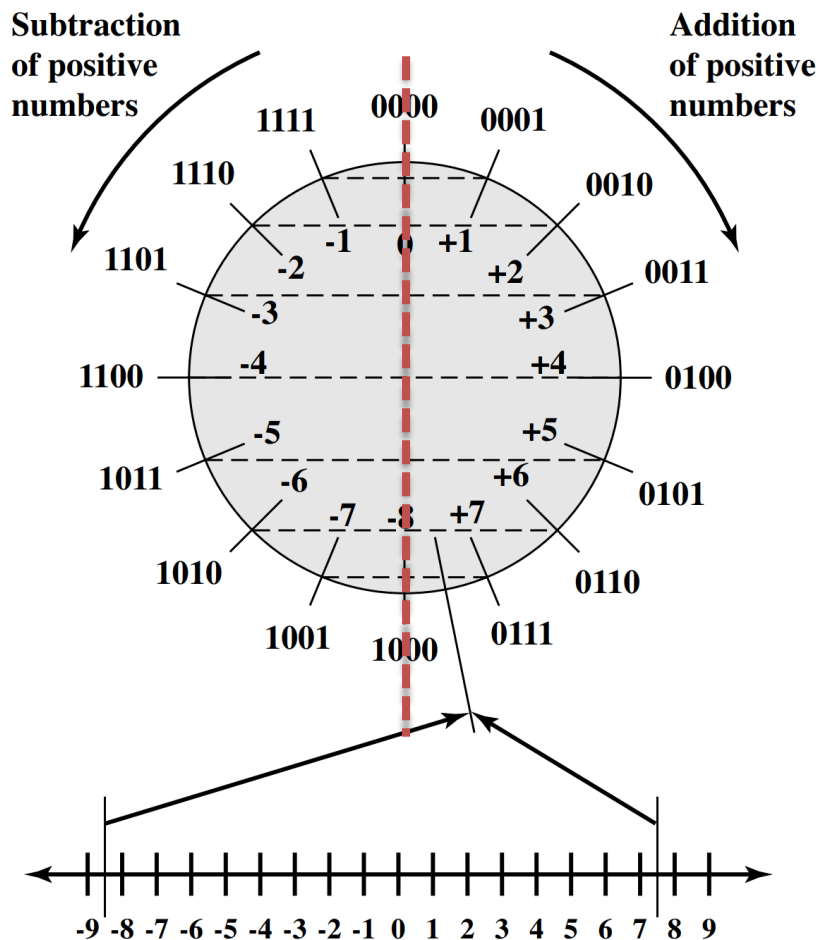
overflow

$$\begin{array}{r} -6 \quad 1010 \\ -4 \quad 1100 \\ \hline (1)0110 \end{array}$$

overflow



# Geometric Depiction of Twos Complement Integers 补码整数的几何表示



- 圆周上列出了4bit的数表示的16种情况，从0000到1111
- 中间有一个对称轴，一个数的相反数就是它针对中间的对称轴的对称点
- 可以看到1000没有对称点，所以只有-8而没有+8的表示
- 对于加法，就是顺时针移动；而对于减法，就是逆时针移动
- 无论是顺时针移动，还是逆时针移动，如果越过了正负的交接点，就表示有溢出



# Judgment of overflow 溢出的判断

- For different representations, the judgment methods of overflow are different 对不同的表示法，溢出的判断方法不一样
- For example:  $01111101 + 01111101 = 11111010$ 
  - For unsigned addition, no overflow 对于无符号加法，没有溢出
  - For signed addition, overflow 对于有符号加法，溢出
- Another example:  $11111101 + 01111101 = (1) 01111010$ 
  - For unsigned addition, overflow 对于无符号加法，溢出
  - For signed addition, overflow 对于有符号加法，没有溢出
- Carry is not a flag to judge overflow 进位不是判断溢出的标志

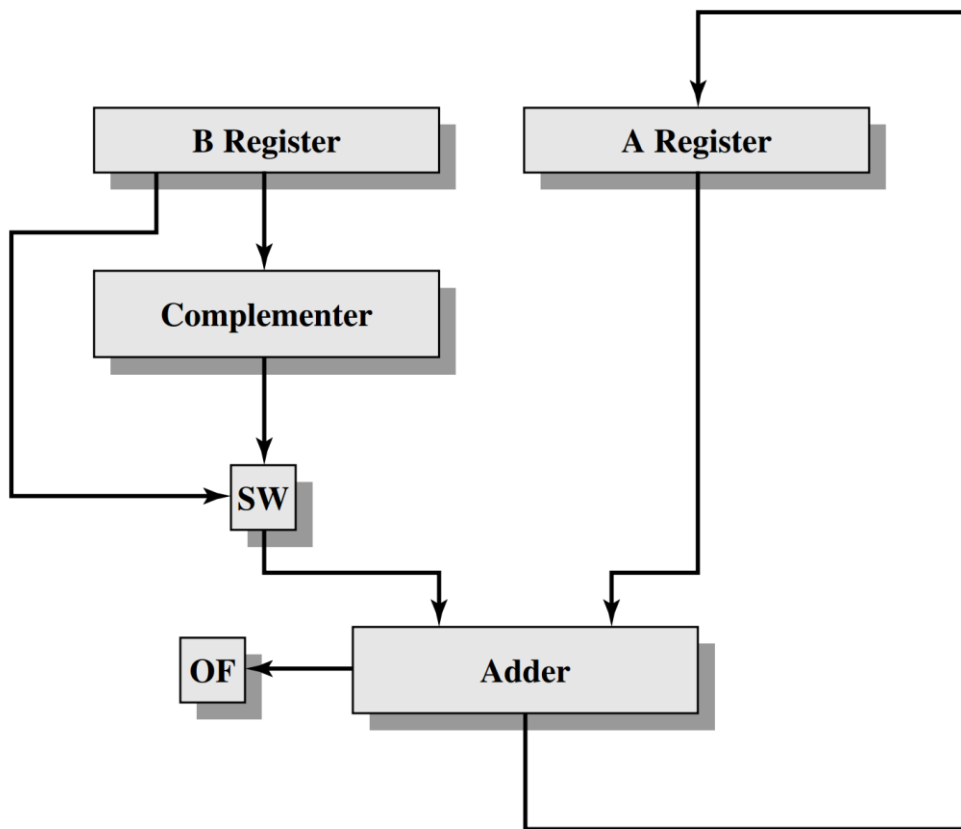


# Judgment of overflow 溢出的判断

- No overflow when adding a positive and a negative number  
一个正数和一个负数相加，不会溢出
- No overflow when signs are the same for subtraction 两个同符号数相减，不会溢出
- Judgment method for two 's complement addition 补码加法溢出的判断方法
  - When two numbers with the same sign are added, the sign bit of the result is opposite, which must be overflow 当两个相同符号的数相加时，结果的符号位相反，一定溢出
  - Subtraction is completed by addition, and the method to judge overflow is the same 减法是通过加法来完成的，判断溢出的方法相同



# Hardware for Addition and Subtraction 加减法的硬件



OF = Overflow bit

SW = Switch (select addition or subtraction)

- 核心是一个加法器。寄存器A和B是加法器的输入
- 计算的结果保存在寄存器A中，如果有溢出，则溢出标志保存在OF中
- SW开关控制是加法还是减法。如果是加法，B寄存器的数直接输入到加法器中；如果是减法，则B寄存器的数据通过一个补码器生成它的相反数的补码，再输入到加法器中



# Exercise

---

- 补码表示法的计算
  - 十进制数为-24，它的8位补码表示
  - 十进制数为-128，它的8为补码表示
- 加减法
  - $-12+48$
  - $-16-24$
- 溢出的判断
  - $01111110+00101101$
  - $10101101+00101100$



# Exercise

- 补码表示法

- $(24)_{10} = (00011000)_2, (-24)_{10} = (11100111+1)_2 = (11101000)_2$

- $(128)_{10} = (10000000)_2, (-128)_{10} = (01111111+1)_2 = (10000000)_2$

- 加減法

- 12+48

- $(-12)_{10} = (1111\ 0100)_2$

- $(48)_{10} = (0011\ 0000)_2$

- $(-12+48)_{10} = (0010\ 0100)_2 = (36)_{10}$

- 16-24

- $(-16)_{10} = (1111\ 0000)_2$

- $(-24)_{10} = (1110\ 1000)_2$

- $(-16-24)_{10} = (1101\ 1000)_2 = (-40)_{10}$



# Exercise

- 溢出的判断

- $01111110 + 01101101$

$$01111110 + 01101101 = 1110\ 1011$$

两个正数相加，变成负数了，溢出

$01111110$ : 126,  $01101101$ : 109。126+109=235，超过127的最大值

- $10101101 + 00101100$

$$10101101 + 00101100 = 1101\ 1001$$

一个正数，一个负数相加，不会溢出



# Multiplication 乘法

- Much more complex than addition 比加法复杂得多
- Multiply each bit of the multiplier by the multiplicand to get the partial product 乘数的每一位和被乘数相乘，得到部分积
- calculation of partial product should also take into account the position of the multiplying digit 每个部分积的计算要考虑到乘数位所在的位置
- Add all partial products to get the product 所有部分积求和得到乘积
- It 's similar to our manual multiplication 类似我们手工的乘法



# Multiplication 乘法

- Involves the generation of partial products 部分结果的生成方法
  - Equals 0 when the multiplier bit is 0 当乘数位为0，值为0
  - Equals the multiplicand(被乘数) when the multiplier is 1 当乘数位为1，值为被乘数
  - Easier than decimal multiplication 比十进制的乘法简单
- The total product is produced by summing the shifted partial products 最终结果是移位后的部分结果的总和
- Two  $n$  binary number may result in a product of up to  $2n$  bits in length 2个 $n$ 位二进制数相乘，结果最多是 $2n$ 位



# Example

1011      Multiplicand (11 dec)

x 1101      Multiplier      (13 dec)

1011      Partial products

0000

1011

1011

10001111

Product (143 dec)

**Note:** 如果乘数为是0，结果为0。如果乘数位是1，结果为被乘数。

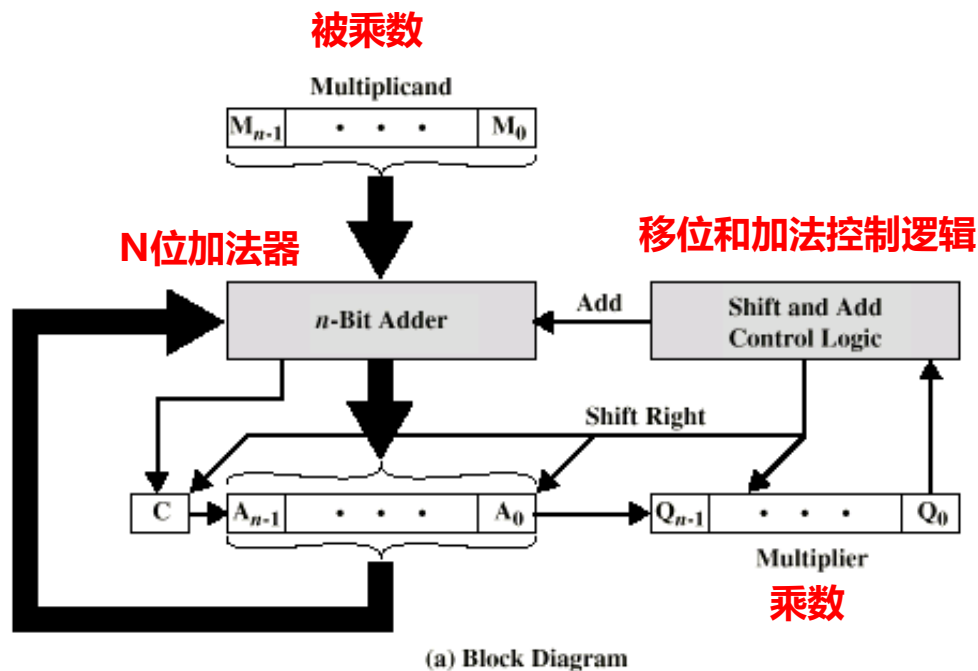
**Note:** 结果需要双倍的比特位



# Shift operation 移位操作

- A shift operation is involved in multiplication 乘法运算中涉及到移位操作
- Any number multiplied by  $2^n$  results the number shifted left by  $n$  bits 任何一个数乘以 $2^n$ ，结果就是这个数向左移位 $n$ 位
- Example
  - $0000\ 0001 \times 00000010 = 0000\ 0010$
  - $0000\ 0001 \times 00000100 = 0000\ 0100$
  - $0000\ 0001 \times 00001000 = 0000\ 1000$

# Unsigned binary multiplication 无符号二进制乘法



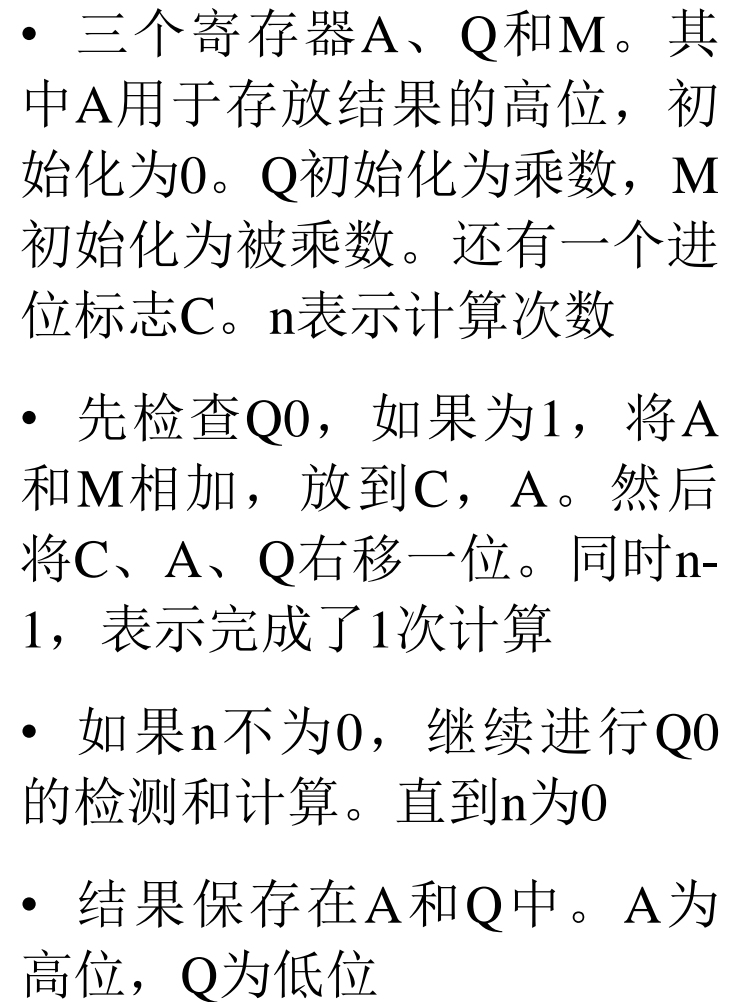
- 核心是一个 $n$ 位加法器
- 先取乘数的最低位 $Q_0$ 。如果是1，则将被乘数送到加法器中。如果是0，则不进行加法。
- 加法器将A和M相加，然后向右移位。有一个移位的寄存器C，用于保存进位。
- 移位后，继续进行 $Q_1$ 的判断，然后相加，移位。这样直到所有的Q都计算完成，这样就可以得到无符号的乘法结果



# Example of multiplication

		乘数	被乘数
		Q: Multiplier	
C	A: Result-H	Result-L	M: Multiplicand
0	0000	1101	1011
0	1011	1101	1011
0	0101	1110	1011
0	0101	1110	1011
0	0010	1111	1011
0	1101	1111	1011
0	0110	1111	1011
1	0001	1111	1011
0	1000	1111	1011

- 初始值C为0，A寄存器为0，乘数为1101，被乘数位1011
- 先取乘数的最低位Q0。如果是1，则将被乘数送到加法器中。如果是0，则不进行加法。
- 第一步，乘数最后一位是1，A和M相加后，得到A为1011。然后移位，A和Q寄存器就变成了：0101 1110。
- 第二步，乘数最后一位是0，A不变，然后移位，A和Q寄存器就变成了：0010 1111。
- 第三步，乘数最后一位是1，将A和M相加，得到1101。然后移位，得到A和Q寄存器变成了：0110 1111。
- 第四步，乘数最后一位是1，A和M相加，得到10001。产生了一个进位。然后移位，进位也右移到A的最高位。得到A和Q寄存器变成了：1000 1111





# Multiplying negative numbers 负数乘法

- Multiplication discussed above is for unsigned integers 讨论的乘法针对的是无符号整数
- Is this method effective for integers represented by two's complement? 针对补码表示法表示的整数，这种方法是否有效呢？
  - $3 * (-4) = -12$
  - 3    0011
  - -4   1100
  - $0011 * 1100 = 00100100 = (32)_{10}$  ✗



# How?

- Solution 1 方法1
  - Convert to positive if required 转成正数乘法
  - Multiply as above 按照上述相乘
  - If signs were different, negate answer 如果两个符号不同，取反
  - Convert to two's complement 转换成补码表示法
  - It's the same way we do multiplication. Regardless of the sign bit, multiply, and then determine the sign of the result 跟平常做乘法的方法一样。先不考虑符号位相乘，最后再确定结果的符号
- Solution 2 方法2
  - Booth's algorithm 布斯乘法
  - Not only solves the problem, but also improves the efficiency 解决了补码表示法的乘法问题，同时提高乘法效率



# Why does Booth's algorithm work? 原理1

- In particular, consider a positive multiplier consisting of one block of 1s surrounded by 0s (for example, 00011110) 一般来说，一个正数的乘数包括若干个被0包围的一组1

$$\begin{aligned} M \times (00011110) &= M \times (2^4 + 2^3 + 2^2 + 2^1) \\ &= M \times (16 + 8 + 4 + 2) \\ &= M \times 30 \end{aligned}$$

$$2^n + 2^{n-1} + \dots + 2^{n-K} = 2^{n+1} - 2^{n-K}$$

$$\begin{aligned} M \times (00011110) &= M \times (2^5 - 2^1) \\ &= M \times (32 - 2) \\ &= M \times 30 \end{aligned}$$

$$2^{n+1} - 2^{n-k}$$



# Why does Booth's algorithm work? 原理2

- For a binary number, we cannot require all its 1s to be connected together. What should we do? 一个二进制数，我们不能要求它所有的1都连在一起，那怎么办
- Use the idea of segmentation. Make the connected 1 into a block 用分块的思想。将连在一起的1组成一个块
- If single 1 is surrounded by 0, a single 1 is also a block 如果单个1被0包围，单个的1也是一个块

$$\begin{aligned} M \times (01111010) &= M \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= M \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$



# Why does Booth's algorithm work? 原理3

- Booth's algorithm makes use of this rule. For continuous 1, it need not calculate, but only at the beginning and end 布斯运算利用这个规律。对于连续的1，在中间不计算，只在头尾进行计算
- When 10 is encountered, subtraction is performed 遇到10做减法
- When 01 is encountered, addition is performed 遇到01做加法

$$\begin{aligned} M \times (01111010) &= M \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= M \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$

01, 对应+2<sup>2</sup>

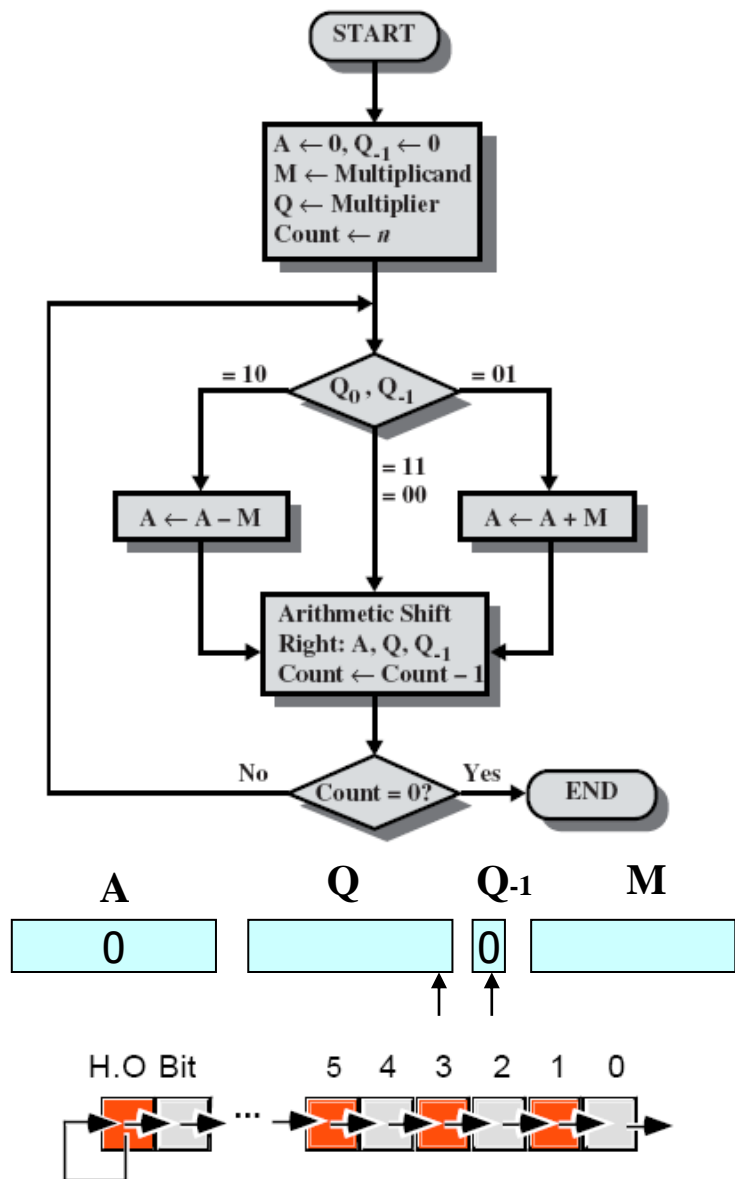
10, 对应-2<sup>1</sup>



# Why does Booth's algorithm work? 原理4

- For continuous 1, it only needs to be calculated at the beginning and end to improve the calculation efficiency 对于连续的1，只在头尾进行计算，提高效率
  - $M^*(01111110) = M^*(2^7 - 2^1)$
- The worst case is that 0 and 1 alternate, so each time you need to calculate 最差的情况是0和1交替，这样每次都要计算
  - $M^*(0101\ 0101) = M^*(2^7 - 2^6 + 2^5 - 2^4 + 2^3 - 2^2 + 2^1 - 2^0)$

# Diagram of Booth's algorithm 布斯算法图示



- A寄存器保存结果的高位，Q寄存器保存乘数和结果的低位，M寄存器保存被乘数。Q-1寄存器保存上一个乘数数字。初始化时A和Q-1均为0
- 开始计算，如果Q0和Q-1是11或00，A和Q以及Q-1不处理，直接移位。
- 如果Q0和Q-1是10，减去M，然后进行移位。
- 如果Q0和Q-1是01，加上M，然后进行移位。
- A、Q和Q-1右移的时候，A的最高位往下移位时，**最高位不是补0，而是将原来的最高位保留，也就是保留符号位**
- 移位之后，继续判断Q0和Q-1，直到所有的位数都处理完毕
- 最后得到的A和Q寄存器中的值就是乘积



# Example-1

$$Q = 2^2 - 2^0$$

A	Q	Q <sub>-1</sub>	M	Initial Values		
0000	0011	0	0111			
1001	0011	0	0111	A	<u>A - M</u>	First Cycle
1100	1001	1	0111	Shift		
1110	0100	1	0111	Shift		Second Cycle
0101	0100	1	0111	A	<u>A + M</u>	
0010	1010	0	0111	Shift		Third Cycle
0001	0101	0	0111	Shift		
						Fourth Cycle

第一次运算，减去M

第二次运算，加上M



## Example-2

- The previous question is how to quickly multiply negative numbers 前面的问题是如何快速完成负数（补码）的乘法
- Can the Booth' s algorithm just discussed solve the multiplication problem of negative numbers? 刚才讨论的布斯算法能否解决负数的乘法问题呢？
- Example
  - Multiplier is -6 乘数是-6
  - Multiplicand is 3 被乘数是3
  - two' s complement of -6 is 1010 -6的补码是1010



## Example-2

A	Q	Q <sub>-1</sub>	M	初始化
0000	1010	0	0011	Initial value
0000	0101	0	0011	First Cycle
1101	0101	0	0011	Second Cycle
1110	1010	1	0011	
0001	1010	1	0011	Third Cycle
0000	1101	0	0011	
1101	1101	0	0011	Fourth Cycle
1110	1110	0	0011	

The value of 11101110 is -18

- 第一步， $Q_0Q_{-1}$ 为00，只需要移位。

- 第二步， $Q_0Q_{-1}$ 为10，减法，再移位。

- 第三步， $Q_0Q_{-1}$ 为01，加法，再移位。

- 第四步， $Q_0Q_{-1}$ 为10，减法，再移位。

- 得到A和Q是1110 1110。为结果的补码数

- 关键点：符号位也进行了移位

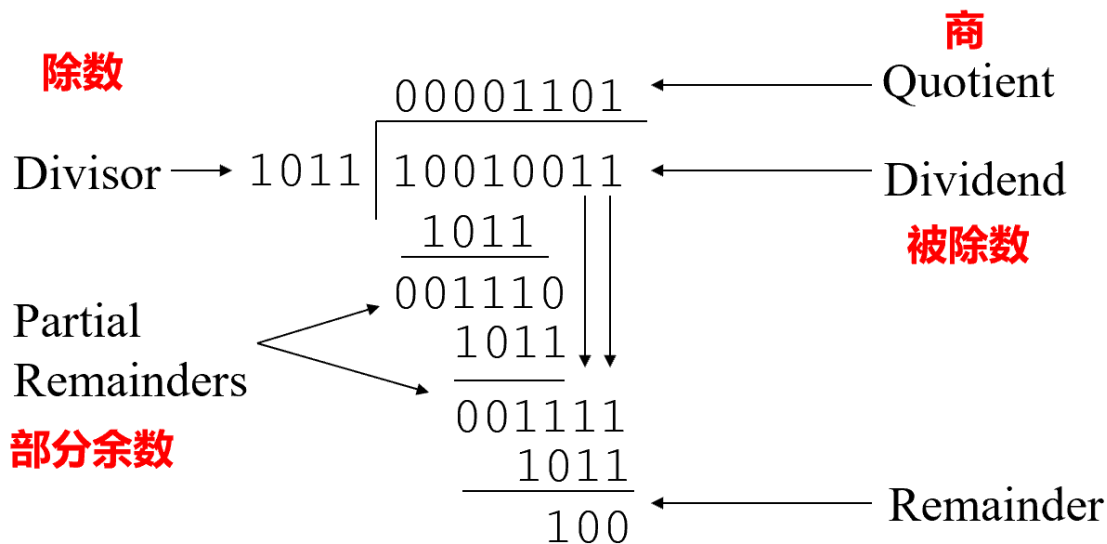


# Division 除法

- Based on long division, similar to decimal division 基于长除法，跟十进制的除法类似
- Much more complex than multiplication 比乘法更复杂
  - It is necessary to determine the size of part bits and divisor in the dividend 需要判断被除数中部分位和除数的大小
  - Determine whether the quotient is 0 or 1 after judgment 判断后决定商位是0还是1
- It 's a little simpler than decimal division 比十进制的除法稍许简单一些
- Negative numbers are really bad! 负数非常麻烦



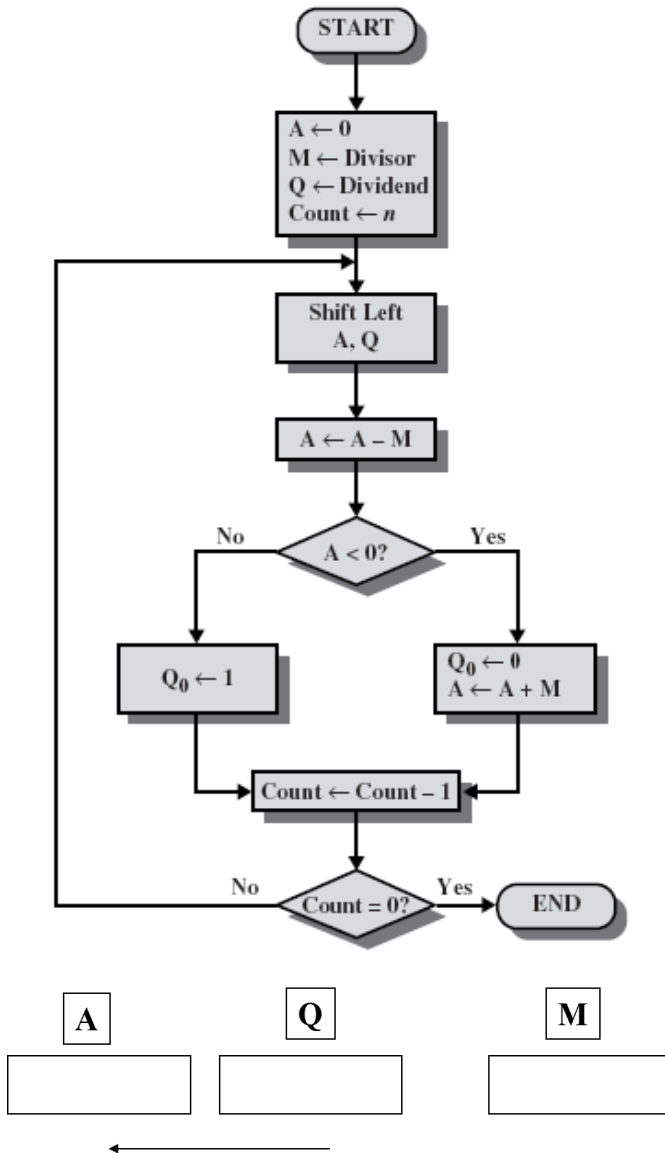
# Division of unsigned binary integers 无符号除法



- 被除数的高4位都比除数小。到第五位，被除数是10010，比除数大，所以商位为1，然后减去除数，得到部分余数是111，继续下一位。直到被除数的最后一位。
- 二进制的长除相对于10进制的长除简单一些。10进制的时候，因为商的每一位可能是0~9，所以我们还需要挨个儿去试，看商的每位是多大。
- 对于二进制，商的每一位只能是0或者1，所以只需要对比被除数和除数的大小
- 结果包括商和余数



# Flowchart for Unsigned Binary Division 无符号除法流程



- 除数在M寄存器中，被除数放在Q寄存器中。 $A$ 的初始值为0，保存被除数的部分余数
- 每一次，先将A和Q进行移位，然后检查A和M的大小，用无符号减法来比较。如果A大于M，当前的余数可以减去M，得到一个商位为1，放在 $Q_0$ 。如果A比M小，则 $Q_0$ 为0，同时A恢复成减去M之前的值
- 然后继续移位并比较。一直到被除数的所有位都用完
- 最后得到的商在Q寄存器中，余数保存在A寄存器中



# Example

A 0000	Q 0111	M 0011	(7)/(3) Initial value
0000 1101 0000	1110  1110	0011	Shift left Subtract Restore
0001 1110 0001	1100  1100	0011	Shift left Subtract Restore
0011 0000 0000	1000  1001	0011	Shift left Subtract Set Q0=1
0001 1110 0001	0010  0010	0011	Shift left Subtract Restore

- $7/3=2$ ，余1
- 初始化时，A为0000，Q为被除数0111，M是除数0011
- 第一次移位后，不够减，恢复
- 第二次移位，还是不够
- 第三次移位，够减了，设置此时的Q0=1
- 第四次，还是不够减，恢复
- 此时已经处理完毕。得到Q=0010，A=0001。所以，商是2，余数是1



# Outline

---

- The Arithmetic and Logic Unit (ALU) 算术逻辑单元
- Integer Representation 整数表示
- Integer Arithmetic 整数的算术运算
- Floating-Point Representation 浮点数表示
- Floating-Point Arithmetic 浮点数运算



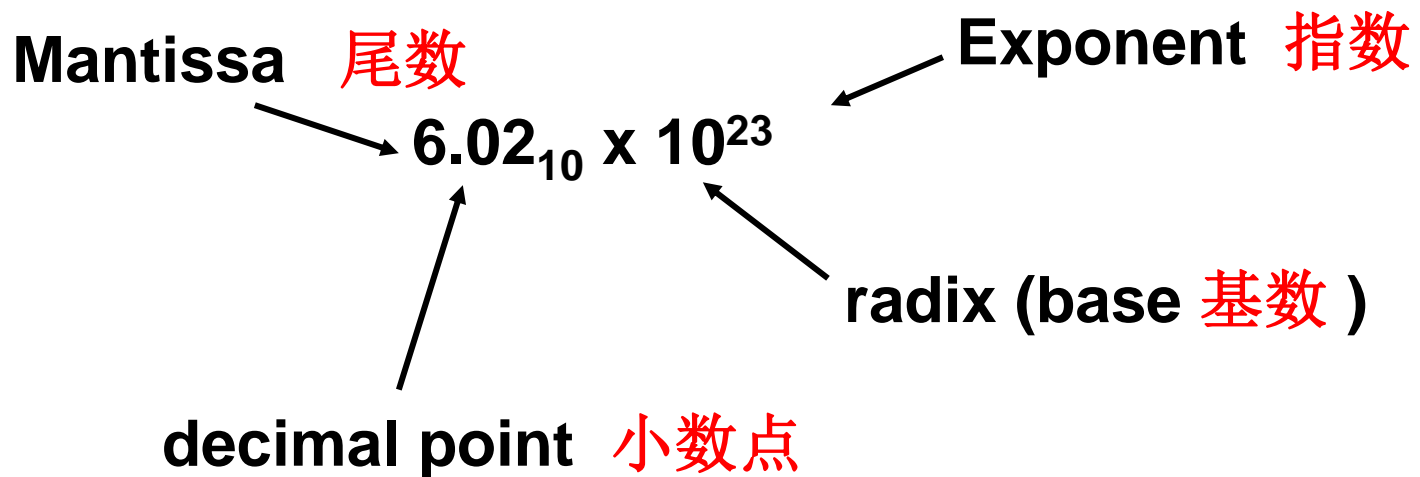
# Real numbers 实数

- Real number: number with fractions 实数：带有小数的数
- Could be represent in pure binary 能够用纯二进制数表示
  - $1001.1010 = 2^4 + 2^0 + 2^{-1} + 2^{-3} = 9.625$
- Problem: Where is the binary point? 二进制小数点放哪呢？
  - Fixed? 固定位置？
    - Very limited 受限
  - Moving? 移动位置
    - How do you show where it is? 它在哪个地方？



# Scientific Notation 科学计数法

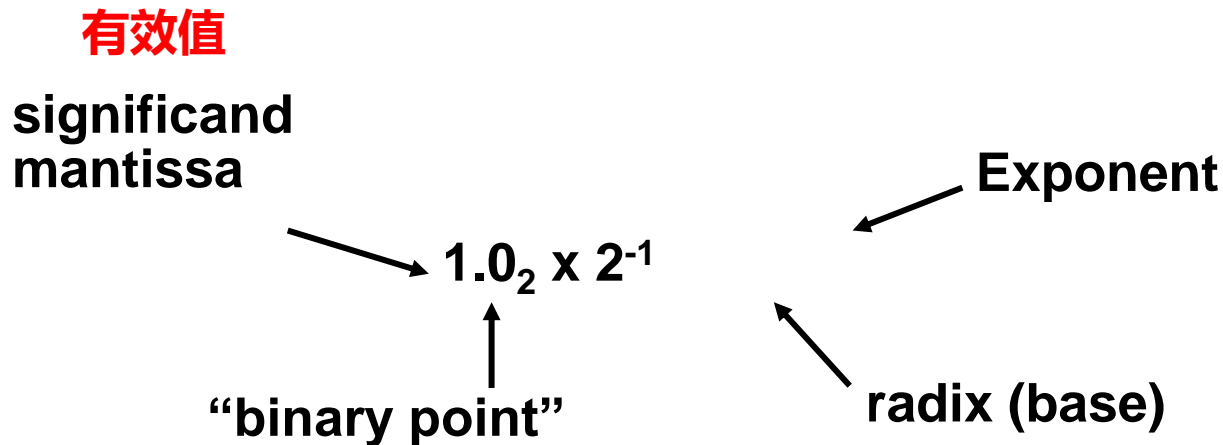
- In decimal, large numbers are represented by scientific notation  
十进制中，用科学计数法表示大数
- Use a mantissa to multiply by the power of 10 用一个尾数，乘以10的多少次幂





# Scientific Notation 科学计数法

- The same in binary 二进制也同样
- Represent a large binary number by multiplying the mantissa and the power of 2 用尾数乘以2的幂次来表示大数
- Representation consists of three parts: sign bit, significant value and exponent 表示法包括三个部分：符号位，有效值，指数





# Floating point 浮点表示法



(a) Format

- Floating point representation 浮点表示法
  - Symbol represented by 0 or 1 0或1表示的符号位
  - Exponent of biased notation 移码表示的指数
  - Valid value, that is, the mantissa of the number 有效值，也就是这个数的尾数部分
- Binary point is the first place on the right of the highest significant value 小数点在有效值最高位的右边第一位



# Exponent 指数

- Why exponent is notated in biased ? 指数为什么用移码?
  - Exponent can be positive or negative 指数可能是正或者负
  - For two's complement representation, it is not very easy to compare 对于补码表示法，数的大小比较不方便
- How to use biased notation? 如何用移码?
  - For a  $k$  bit exponent, the bias is  $2^{k-1} - 1$  对于 $k$ 位的指数，偏移量是 $2^{k-1} - 1$
  - actual exponent range is:  $-(2^{k-1} - 1) \sim 2^{k-1} - 1$  实际指数范围为 $-(2^{k-1} - 1) \sim 2^{k-1} - 1$
  - The bias is added to the actual exponent to get the stored exponent 偏移量加到实际的指数上得到存储的指数



# Example

- For an 8 bit exponent field, the bias is 127 8位指数域，偏移量是127
- Range of stored exponent is: 0-255 存储的指数范围为0~255
- Range of actual exponent is: -127~128 实际的指数范围为-127~128
- Example:
  - Actual exponent is -20
  - Stored exponent is: 107



# Examples 举例



- $-1.1010001 \times 2^{10100}$  :
- Sign: 1
- Exponent:  $0111\ 1111 + 10100 = 1001\ 0011$
- Significand:  $101\ 0001\ 0000\ 0000\ 0000\ 0000$
- The floating point representation for above number:
  - $1\ 10010011\ 101000100000000000000000$
- Problem: Where is the first 1? 尾数中的第一个1去哪了?



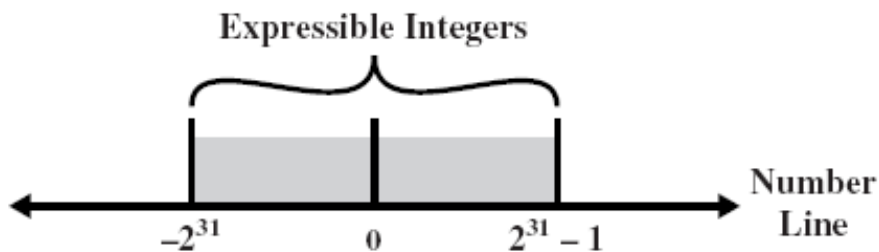
# Normalization 规格化

- To simplify operations on floating-point numbers, it is required to normalize the number 为了简化浮点数的操作，需要进行规格化
- How to normalize? 如何规格化？
  - The leftmost bit of the significand is 1 尾数的最高位是1
    - e.g.  $1.XXXXX \times 2^k$
  - Because the most significand bit is always one, it is unnecessary to store this bit – implicit 因为最高位为1，所以没有必要写出来，隐含就可以了
  - 23-bit field is used to store a 24-bit significant 用23位表示了24位的有效值



# Range of number 数的表示范围

- What is the range of floating point numbers? 浮点数表示的范围多大?
- Two values that affect the range of floating point numbers: significand and exponent 影响浮点数范围的2个值：有效值，指数
- For integer, if it use two's complement representation, the range of 32-bit integer is:  $-2^{31} \sim 2^{31}-1$  对于整数，如果用补码表示法，32位数的表示范围：  $-2^{31} \sim 2^{31}-1$





# Range of floating point 浮点数的范围



- The smallest significand: 1.0 最小的尾数
- The biggest significand =  $1.11111\dots = 2 - 2^{-23}$  最大的尾数
- The smallest exponent = -127 最小的指数（偏移量为127）
- The biggest exponent = 128 最大的指数（偏移量为127）
- Negative numbers between:  $-(2 - 2^{-23}) \times 2^{128}$  and  $-2^{-127}$  负数
- Positive numbers between:  $2^{-127}$  and  $(2 - 2^{-23}) \times 2^{128}$  正数

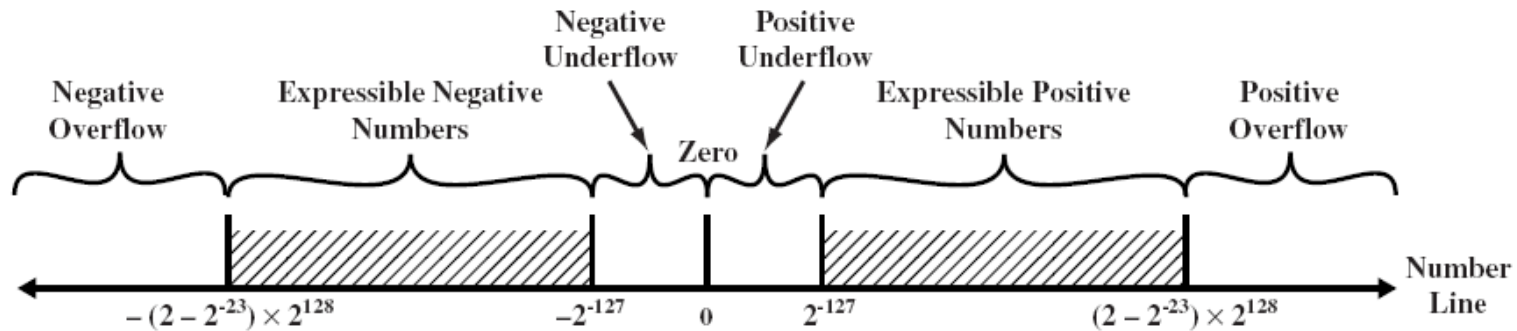


# Floating point range and accuracy 范围和精度

- For a 32 bit number 对于32位的浮点数
  - 8 bit exponent, up to 128 8位指数, 最大为128
  - $\pm 2^{128} \approx 1.0 \times 10^{39}$
- Accuracy 准确性
  - The effect of changing lsb of mantissa 尾数最低位的效果
  - 23 bit mantissa  $2^{-23} \approx 1.2 \times 10^{-7}$
  - About 6 decimal places 大约10进制的小数点后6位
- Because total bits of floating point numbers is fixed, the range and accuracy are contradictory 浮点数总位数固定, 范围和精度矛盾



# Floating point overflow 浮点数的溢出

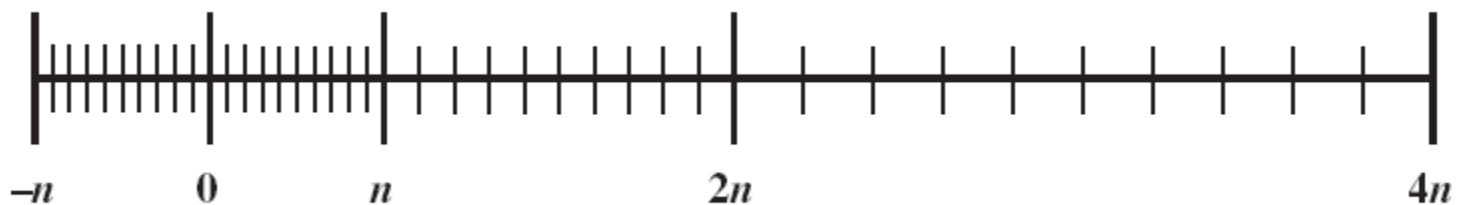


- Because the number of floating point numbers is limited, some numbers cannot be represented, which is called overflow or underflow 浮点数的位数有限，有些数无法表示，称为上溢出或下溢出
- Overflow: When  $> (2 - 2^{-23}) \times 2^{128}$  or  $< -(2 - 2^{-23}) \times 2^{128}$  上溢出
- Underflow: When value  $> 0$ ,  $< (2 - 2^{-23}) \times 2^{-127}$  or  $< 0$ ,  $> (2 - 2^{-23}) \times 2^{-127}$  下溢出



# Density of floating point numbers 浮点数的密度

- For fixed-point numbers, the numbers are spaced evenly 对于定点数，在空间上是均匀的
- For float-point notation, the numbers are not evenly spaced 浮点数在空间上是不均匀的
  - The length of exponent is fixed 指数的长度是固定的
  - As the number is getting bigger, the space between two numbers is bigger 数越来越大，数之间的距离越来越大





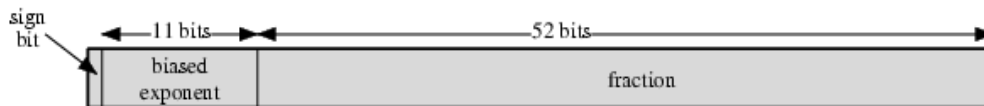
# IEEE 754 754标准

- Standard for floating point storage 浮点数存储标准
- 32 and 64 bit standards 32位和64位标准
- 32 bits standard
  - 8 bits exponent , 23 bits significand 8位指数, 23位有效值
- 64 bits standard
  - 11 bits exponent , 52 bits significand 11位指数, 52位有效值
- Extended formats (both mantissa and exponent) for intermediate results 针对中间结果, 规定了扩展格式, 包括尾数和指数都进行了规定



(a) Single format

单精度格式



(b) Double format

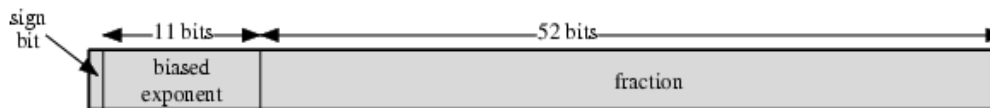
双精度格式

- 思考：
  - 双精度格式的数据，数据表示的范围？
  - 11位指数
  - 52位有效值



(a) Single format

单精度格式



(b) Double format

双精度格式

- 分析:

- 11位指数，采用移位表示法。指数范围是 $-(2^{10}-1) \sim 2^{10}$ ， $-1023 \sim 1024$
- 52位有效值，范围为： $1 \sim (2-2^{-52})$
- 正数： $2^{-1023} \sim (2-2^{-52}) * 2^{1024}$
- 负数： $-(2-2^{-52}) * 2^{1024}$ ， $\sim -2^{-1023}$



# Special about IEEE 754 formats 特殊情况

- Extreme value of the exponent to define a particular value 指数的极值表示特殊的值
- Exponent after bias is 2047, that is, all 1, indicating infinity 如果移位后的指数是2047，也就是全1，表示的是无穷大
- The actual range is smaller than the ideal range 实际的范围比理想的范围要小
- For negative, the range is  $-(2-2^{-52}) * 2^{1023} \sim -2^{1022}$  负数的范围
- For positive, the range is  $2^{-1022} \sim (2-2^{-52}) * 2^{1023}$  正数的范围



# Outline

---

- The Arithmetic and Logic Unit (ALU) 算术逻辑单元
- Integer Representation 整数表示
- Integer Arithmetic 整数的算术运算
- Floating-Point Representation 浮点数表示
- Floating-Point Arithmetic 浮点数运算



# Floating point arithmetic +/- 浮点数的加减

---

- 问题:
  - 浮点数的加减法和整数的加减法差别在哪里?
  - 需要做什么处理?



# Floating point arithmetic +/- 浮点数的加减

- Addition and subtraction of floating point numbers must first ensure that the exponent are the same 浮点数的加减法首先需要保证阶值相同
- How to do this? Move the binary point position of the valid value of the operand 移动小数点的位置
- Steps for addition and subtraction floating point numbers 浮点数加减法的步骤
  - Check for zeros 检查0
  - Align significands (adjusting exponents) 通过调整指数，对齐有效位
  - Add or subtract significands 加或减尾数
  - Normalize result 规格化结果



# Example 举例

指数不一样，  
不能直接计算

$$1.011 * 2^5 + 1.001 * 2^3$$

- Check for zeros: no zero 检查0
- Align significands 对齐尾数

$$1.001 * 2^3 = 0.0101 * 2^5$$

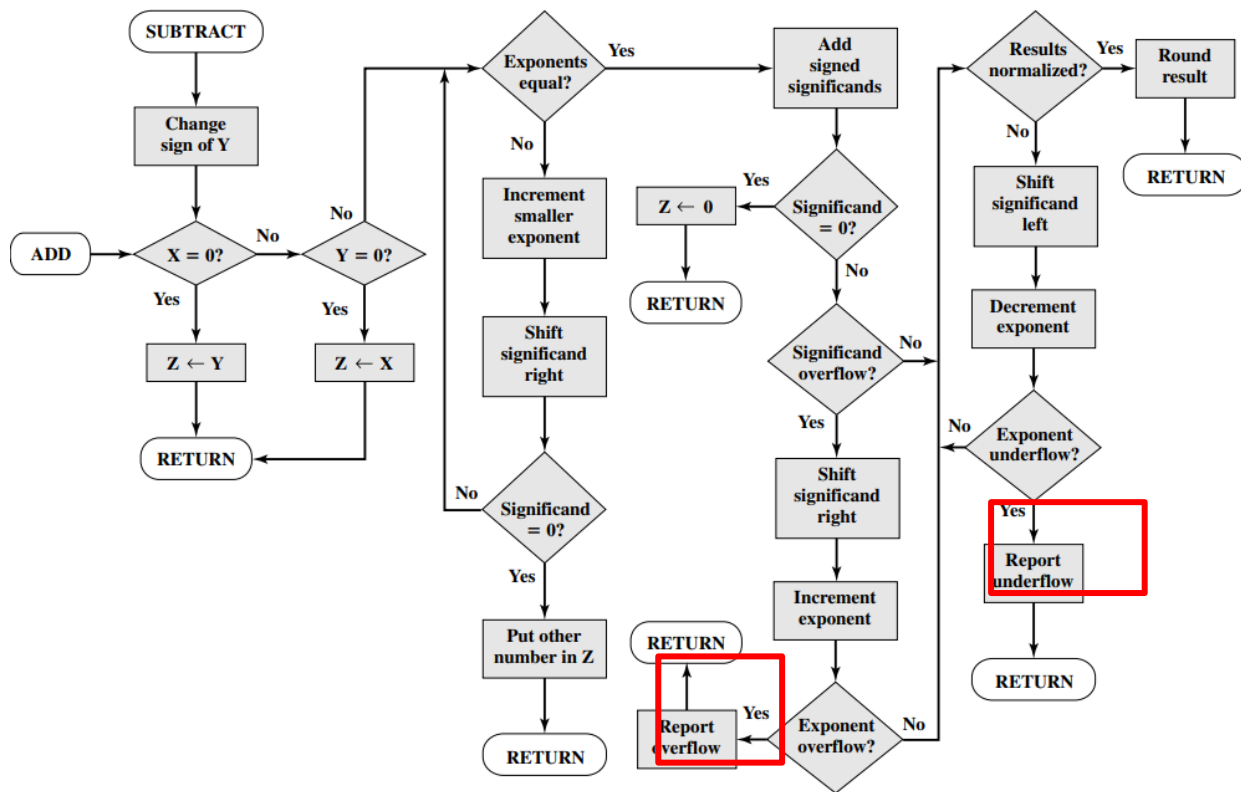
- Add or subtract significands 加减尾数

$$1.011 + 0.0101 = 1.1011$$

- Normalize result 规格化

$$\text{Result} = 1.1011 * 2^5$$

# FP Addition & Subtraction Flowchart 浮点加减流程



- 减法用加法来实现
- 判断上是否有0，如果有0，另一个数就是结果
- 调整尾数，使得指数相同
- 尾数相加
- 判断溢出
- 规格化



# Floating point arithmetic $\times/\div$ 浮点数的乘除

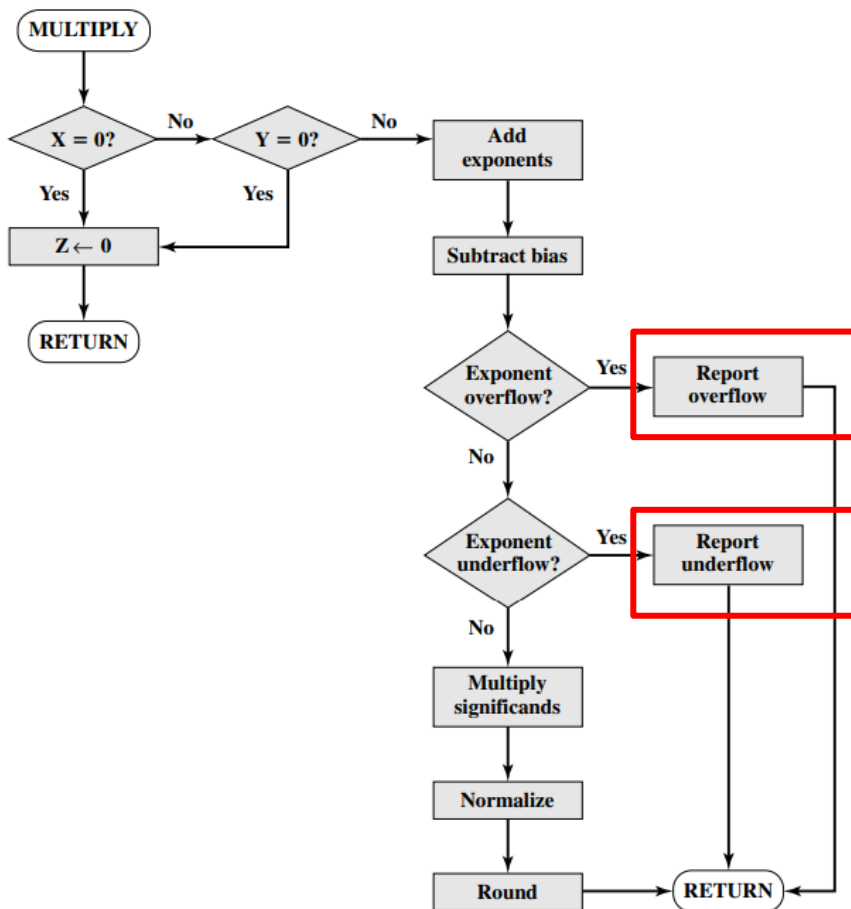
- Multiplication and division are simpler than addition and subtraction 乘除相对加减法简单一些
- Steps:
  - Check for zero 检查是否为0
  - Add/subtract exponents 加/减指数
  - Multiply/divide significands (watch sign) 尾数相乘，考虑符号
  - Normalize 规格化
  - Round 四舍五入
- All intermediate results should be in double length storage 中间结果需要双倍长度
- Is that right? 还有问题吗？



# Floating point arithmetic +/- 浮点数的加减

- 指数问题：
  - 指数都是移码表示
  - 做乘法时，指数相加，得到的结果中是有两个偏移量。
  - 需要从指数的和上减去一个偏移量。同时应考虑指数的上溢问题。那我们是先求和再减去偏移量，还是先把两个指数减去偏移量求和，再加上偏移量呢？大家考虑一下。
  - 做除法时，指数相减，两个偏移量已经减掉了，所以在结果中应加一个偏移量。

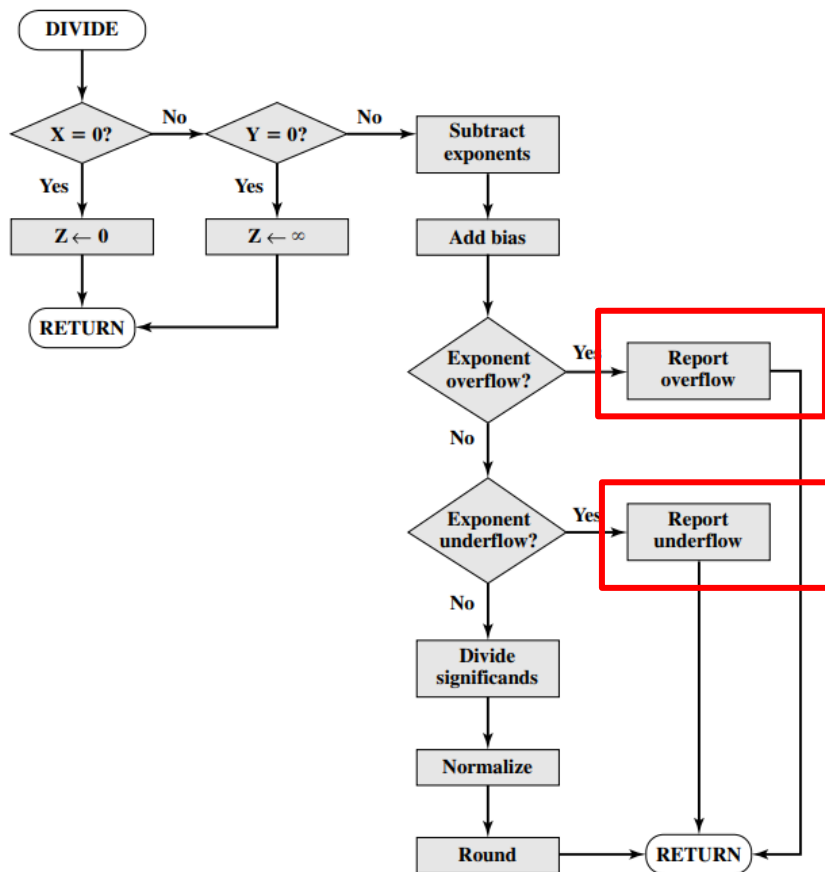
# Floating point multiplication 浮点数乘法



- 判断X或Y是否为0。如果有一个为0，那么结果就是0。
- 指数先减去偏移量，然后相加。
- 如果指数溢出，就报告溢出，包括上溢出和下溢出。
- 尾数相乘。
- 规格化和四舍五入，得到最终结果



# Floating point division 浮点数除法



- X或Y是否为0。如果X是0，结果是0。X不是0，Y是0，那么结果是无穷大
- 对指数进行相减，然后加上一个偏移量
- 如果指数溢出，就报告溢出，包括上溢出和下溢出。
- 尾数相除。
- 规格化和四舍五入，得到最终结果



## Example 举例

$$(1.011 * 2^5) * (1.001 * 2^3)$$

- Check for zeros: no zero 检查是否为0
- Add/subtract exponents 加/减指数

$$5+3=8=(1000)_2$$

- Multiply/divide significands 尾数相乘/相除

$$1.011 * 1.001 = 1.100011$$

- Normalize result 规格化结果

$$\text{Result} = 1.100011 * 2^8$$

- Round 四舍五入

$$1.100011 * 2^8 \approx 1.100 * 2^8$$



# Key Terms

ALU	Exponent overflow	Negative underflow	Positive underflow	Significand underflow
Arithmetic shift	Exponent underflow	Normalized number	Radix point	Sign-magnitude representation
Biased representation	Fixed-point representation	Ones complement representation	Sign bit	Two complement representation
Denormalized number	Floating-point representation	overflow	significand	
exponent	Negative overflow	Positive overflow	Significand overflow	



# Summary and Question

---

- 小结
  - 这节课我们对计算机算术的方法进行了分析，包括整数的表示方法，整数的计算，浮点数的表示方法和浮点数的计算。
- 问题
  - 问题1：一个8位补码数表示的范围，用十进制表示是多少？
  - 问题2：浮点数的加减法中为什么要调整有效值？



# Assignments

---

- Review Questions
  - 10.4, 10.8, 10.11
- Review Questions
  - 10.10, 10.23, 10.24, 10.38, 10.39, 10.40



**谢谢大家!**

