1. 表示字符，字符串

字母及符号：ASCI码

共8位，实际只用到7位共128种，32-126可印刷字符，其他是控制和通信字符

32空格

48-57是0-9，前4位0011，后四位0000-1001

65-90/97-122，A-Z/a-z，前三位010/011，后五位00001-11010

中文：两个字节，编码有UTF-8，GB 2312-80等

GB 2312-80：

94个区域（行），94个位置（列），都是0-93，构成区位码

由于计算机是一个一个字节接收，所以要处理与ASCI码的冲突

实际使用国标码：区位码+2020H，防止与控制与通信字符的冲突

存入计算机的码，汉子机内码：国标码+8080H，防止与所有ASCI码的冲突

输入编码：如输入法

字形码：一个矩阵，1表示该像素点亮，否则不亮，即可输出汉字

字符串：从低地址到高地址存储，最后一个字节存\0（00H）

实例：

某计算机按字节编址（每一个字节对应一个地址），从地址为2开始，存入”abc”

则a在2，b在3，c在4，\0（00H）在5

带有多字节数据的，如中文：

大端模式，将多字节数据的最高有效字节（高位）存到低地址

小段模式：将高位存入高地址

1. 数据校验

概念：

跳变：位错误

若干位代码组成的一个字叫码字，码字数量=2^位数

两个码字之间逐位比较，不同位的个数称为两个码字的距离

码距：一种编码方案中，所有合法码字之间的最小距离

编码方案的检错能力：码距为n，<n位的跳变一定会变成非法码字，可以检测错误，>=n位跳变一定会变成另一个合法码字，检测不出错误

码距=1没有检错能力，码距>=2有检错能力，码距>=3在设计合理时，可能会具有纠错能力

1. 奇偶校验：只有检错能力

在一个有效信息位为n的编码的首部或者尾部添加一个校验位

校验位0还是1，根据：

奇检验码：整个校验码（有效信息位+检验位）1的个数为奇数

偶检验码：整个校验码（有效信息位+检验位）1的个数为偶数

一原始编码中，异或所有位，若有偶数/奇数个1，结果为0/1，即可求奇校验位/偶检验位

检错过程：检查接收的的编码1的个数是奇数还是偶数，就可以检错

但是只有奇数位跳变时才有检错能力，偶数位跳变时就检测不出错误

异或收到的检验码所有位判断1的奇偶

1. 海明（汉明）校验码：基于偶检验，有检错，纠错能力

在n位原始编码中，添加k个校验位，分k组，每组偶检验

构建海明码：

（1）确定检验位个数，确定检验位和信息位在海明码中的位置

k的个数：2^k>=n+k+1，构成n+k位海明码，用D1….Dn表示信息位，P1…Pk表示检验位，H1…..Hn+k表示海明码各个位。因为二进制第一位在最右边，所以没有特别说明，H从右至左，左右顺序只影响（1）

校验位的H位置：2\*(i-1)，i=1,2….k信息位H位置从右至左填充

（2）分k组，确定检验位的值，进而确定海明码

列出D所在的H，H的编号写成二进制，最低位为1的位第一组，第二位为1的为第二组，第k位为1的为第k组。

每组按照偶检验规则确定对应的每一位检验位

（3）若有需要，添加一个用偶检验规则的全检验位放在海明码的首部或尾部

对收到的信息进行检验

对第1,2….k组进行偶检验，得到S1,S2…Sk，Sk..S2S1构成的二进制就是出错的位置

海明码有一个比特的纠错能力和两个比特的检错能力，判有几个比特错误要偶检验全检验位

Sk..S2S1==0…00，全检验位==0，没有错误。

Sk..S2S1!=0…00，全检验位==1，有一个比特错误，可以纠错

Sk..S2S1!=0…00，全检验位==0，两个比特错误，无法纠错

1. 循环冗余码（CRC码）

除数用生成多项式或者直接用二进制表示，除数n位，k=n-1

1. 原始编码左移k位，k是生成多项式的最高次幂（二进制除数位数-1），手算则添0
2. 左移后的编码模2除除数（模2除与除法是本质不同的），得到的结果是校验位，就是CRC码的后k位

模2除过程：每次取n位（模2减结果和被除数取下来构成n位），根据这n位编码的最高位是1/0，上商1/0，然后除了最高位，其他模2减（异或）

如图，原始编码101001，生成多项式x3+x2+1即除数为1101，得到校验位001

最终的CRC码为101001001



对收到的信息模2除除数，若结果为0，则没有出错，反之出错。

若出错，校验位为k位，则每2^k-1一个循环，每个循环的余数与出错位置虽无序但固定的，所以当k满足2\*k>=n+k+1时，就可以准确确定出错位置，此时具有纠错能力。

检错能力：可以检测出奇数个比特的错误，或者双比特的错误，或者<=k位的连续错误。

1. 定点数的表示和运算
2. 表示

无符号数：所有位都表示数值

无符号数通常只讨论整数（如C语言没有unsign float），整数范围0到2^n-1，n为位数

有符号数：包括原码，反码，补码，移码，写作[x]原….，最高位表示符号位（0正1负）

有符号数就得讨论整数和小数，定点整数小数点隐含在最低位后面，定点小数小数点隐含在最高位（即符号位）后面，位权根据小数点位置决定，表示一个数时整数部分和小数部分单独保存

尾数是除了符号位的其他位，整数的符号位和尾数用逗号隔开，小数用点隔开

真值是人类容易理解的数值，通常是十进制

正数的原，反，补都等于原，负数则各不相同，以下只讨论负数的反，补，和正负数的移码

原码：

尾数是真值的绝对值,若机器字长为n位，则尾数为n-1位

整数范围：-(2^(n-1)-1)<=x<=2^(n-1)-1 小数范围：-(1-2^(-n+1))<=x<=1-2^(-n+1)

整数和小数都有+0和-0，所以总体能表示的整数/小数个数都是2^(n+1)-1

反码：

把原码除了符号位全部取反，反码只是为了方便原码转换成补码

范围与原码一样，也有+0和-0

补码：

反码+1（要考虑进位，包括符号位）

没有-0，因为1,00…..的补码为10,0000…..的1超出机器字长被丢弃。所以整数1,000…表示-2^n，n为机器字长，小数则表示-1

范围比原码整数多出来一个-2^n，比原码小数多一个-1，总体整数/小数个数都为2^n

补码转换成原码：

方法1：除了符号位全部取反后+1或

方法2：补码-1再除了符号位全部取反

方法3：补码最右边的1作为分界，左边全部取反，分界和右边不变

快速从[x]补求[-x]补：[x]补包括符号位全部取反后+1

补码原理和作用：y=kx+r 余数r满足 0<=r<x x为模

(10-3)%12<=>(10+9)%12 在模12情况下，余数相同的称为一类数，-3 9 21 33都是一类数

|-3|+|9|=12 绝对值相加等于mod的称为一对补数，-3和9互为补数

对于a-b⬄a+(-b)，只要找到-b的补数，就能转换成加法

机器字长8位相加只会保留最低八位，自动mod256

补数=256-|-b|，得到的就是补码

补码=反码+1的原理：原码+反码+1=256

由此，加法器就可以实现减法，计算机中存储的都是补码

移码：

补码的符号位取反，只能表示整数

因为负数符号位1变0，正数符号位0变1，所以非常适合用于比较大小

和补码一样没有-0，范围和补码一样

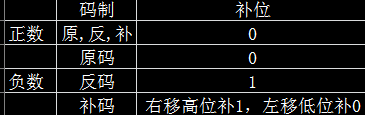
补码符号位取反的操作等同于真值+偏置值(128D，2^(n-1)),注意真值不要添0

1. 运算

移位运算：

1. 算术移位：左移\*2，右移/2，丢1时失去一定精度

移位时只处理尾数，符号位不处理



应用：如x\*7，等价于x不移位+x左移1位+x左移两位

1. 逻辑移位：无符号数的算术移位，要处理最高位，补位0

应用：拼接字节，如00001111和11110000拼接成0000111111110000

申请两个都是16位的空间a和b，00001111放到a后逻辑左移8位，11110000放到b，然后a+b

1. 循环移位：循环补位

左移移出的高位去补低位，右移移出的低位去补高位

应用：对调高字节和低字节的数据，大端模式与小端模式相互转换

带进位位的循环移位：

左移：最高位到进位位，进位位补低位

右移：进位位变最高位，最低位补进位位

加减运算和处理溢出：只讨论原码和补码

1. 原码

加法：

被加数和加数符号相同：符号位不变，尾数相加

被加数和加数符号不同：绝对值更大的-更小的，符号位与绝对值大的一致

减法：减数符号位取反，转换成等价的加法

1. 补码：由于原码加减法的硬件实现很复杂，所以硬件实现采用补码

a+b：

符号位参与运算直接加减，得到的结果就是加减对应的补码

a-b：

=a+(-b)，求出[-b]补，再补码加法即可

由于这些特性，设计一个基于补码的加法器即可完成加减运算

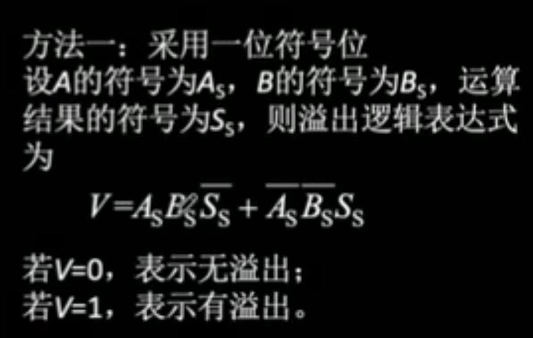
1. 溢出处理：

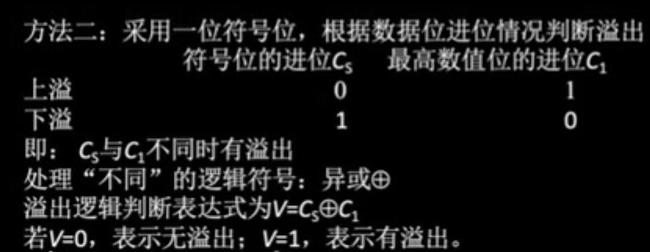
无论是原码还是补码的加减法，只要加法时符号位相同就有可能出现溢出

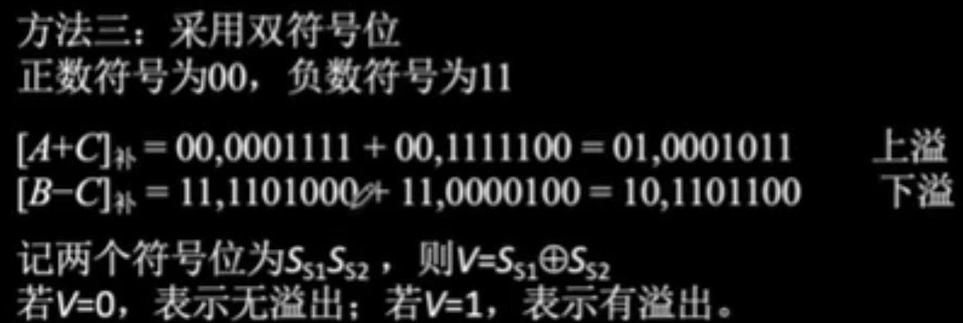
正数+正数超出范围时，发生上溢，变成负数

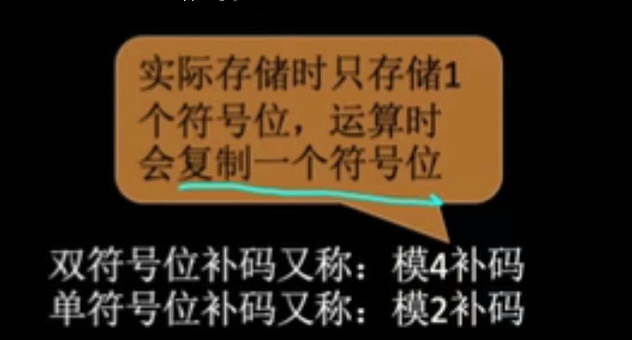
负数+负数超出范围时，发生下溢，变成正数

判断溢出：









处理溢出：短数据扩展为长数据（也叫符号扩展）

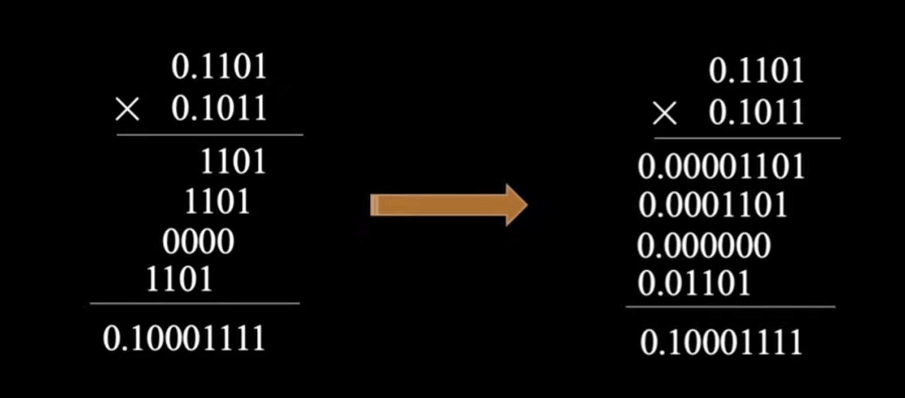
整数在符号位和尾数之间扩充8个位

小数在尾数后面扩充8个位



乘法运算：

原理：



n位乘法：每次有n位参与运算

ACC存放高位部分积，初始化为0，MQ存放低位部分积和乘数，X存放被乘数

原码的一位乘法：

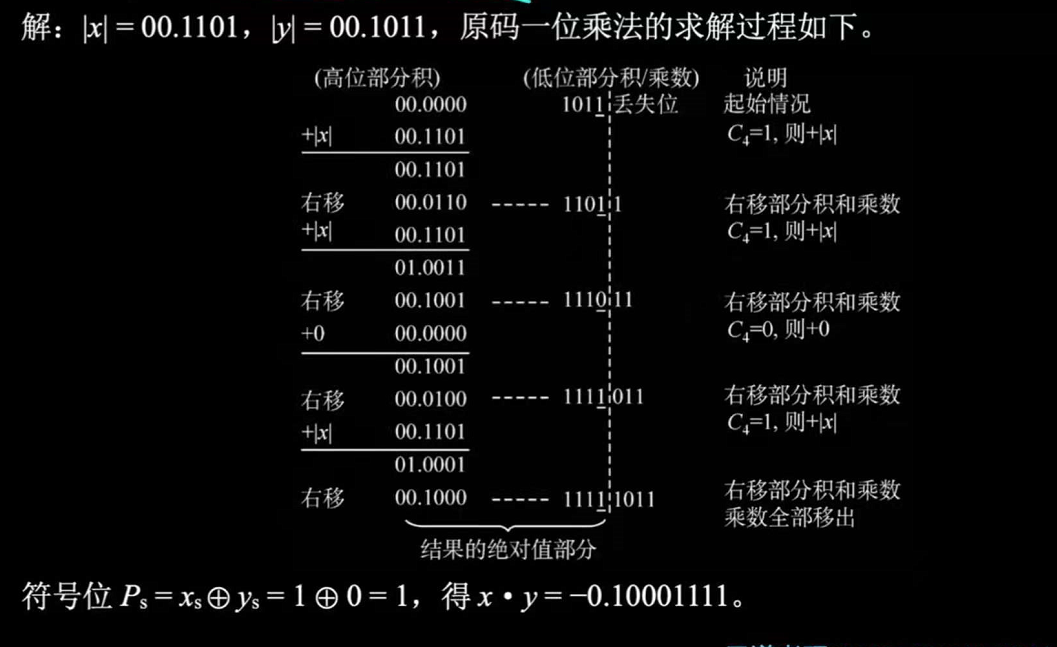
过程：

被乘数绝对值放在X，乘数绝对值放在MQ，X=[|x|]原

乘积符号位：被乘数符号位异或乘数符号位

1. MQ的最低位为1/0，ACC+X/ACC+0
2. ACC和MQ同时逻辑右移（ACC最低位右移到MQ最高位）

若乘数的尾数有n位，则重复（1）（2）n次



这里也可以写成单符号位，不过补码一位乘法必须是双符号位，建议统一

补码的一位乘法：

和原码一位乘法非常类似，不同的是MQ最低位用来作为辅助位。但是为了与原码统一以下提到的MQ最低位是MQ的第二位。由于CPU各寄存器的长度相同且MQ多了一位辅助位，所以ACC和X也多了一位，用来构成双符号位，MQ单符号位

过程：

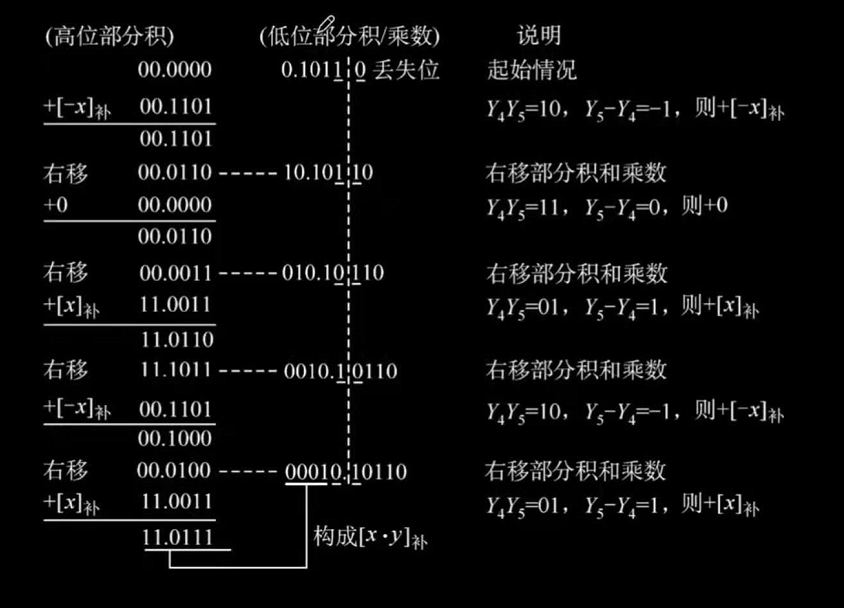
被乘数的补码放在X，乘数补码放在MQ，X=[x]补

符号位直接参与运算，运算结束符号位自动确定

会有专门的电路实现[x]补 转换成[-x]补

1. 辅助位-最低位=1/0/-1，ACC+[x]补/ ACC+ 0/ ACC+[-x]补
2. ACC和MQ同时算术右移

若乘数的尾数有n位，则重复（1）（2）n次，最后再执行一次ACC+[x]补



除法运算：

注意：定点小数的除法，被除数必须小于除数，否则商大于1，定点小数无法表示

ACC存放被除数，每一步的余数，MQ存放商，X存放除数

原码：

恢复余数法：

过程：

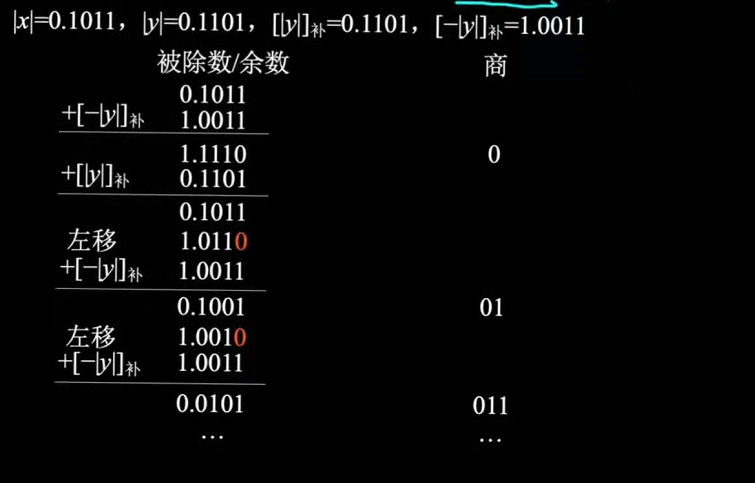
被除数的绝对值放在ACC中，除数的绝对值放在X中，X=[|x|]原，MQ初始化为0

商符号位=被除数符号位 异或 除数符号位，ACC和X符号位参与运算

1. MQ最低位上商1,ACC-X=ACC+(-X)=ACC+[-[x]]补。
2. 若此时ACC的符号位=0则直接跳过（2），若此时是定点小数除法的第一步说明被除数大于除数，直接退出除法运算；如果是符号位=1则MQ最低位改为0，ACC恢复上一步余数即ACC+X补码=ACC+[|x|]补（第（2）步相当于比较ACC和X的大小）
3. ACC和MQ同时逻辑左移（MQ最高位到ACC最低位）

若除数的尾数有n位，则重复（1）（2）（3）n次，最后再执行一次（1）（2），定点小数最终得到的余数要除2^-n（因为小数点隐含在ACC最高位后面）

部分过程：



加减交替法（又叫不恢复余数法）：

优化了恢复余数法，当商1余数为负时，（2）（3）和下一步的（1）合为一个步骤2a+b，a为商1后得到的余数（不管正负），b为X=[|x|]原，若a是正数则不需要恢复余数，2a-b

a是负数 //（1）得到的余数，上商改为0

a+b恢复余数 //（2）

左移2(a+b)=2a+2b //（3）

ACC-MQ=2a+2b-b=2a+b //下一步的（1），根据2a+b的正负看是否需要商改0

////

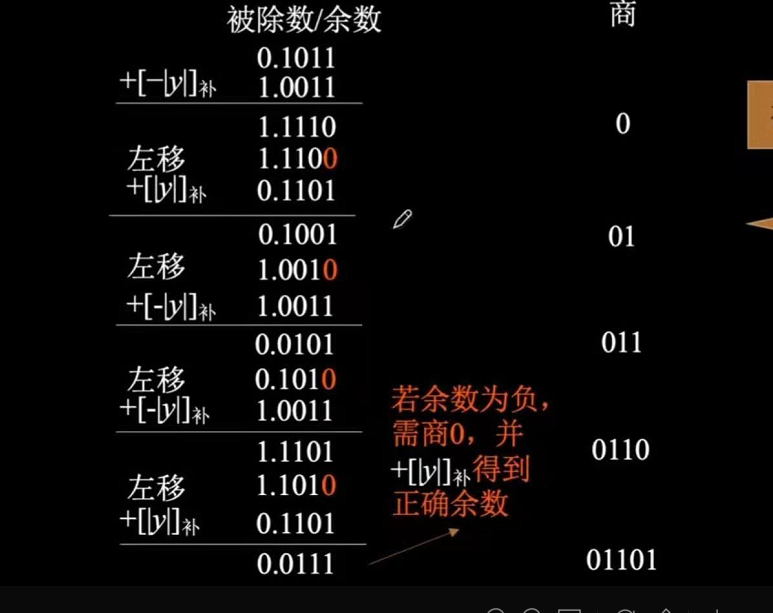
a是正数 //（1）得到的余数，商不变

不回复余数 //（2）跳过

左移 2a //（3）

ACC-MQ 2a-b //下一步的（1），根据2a-b的正负看是否需要商改0

2a+b和2a-b构成加减交替，其他步骤与恢复余数法相同



补码：

加减交替法：

过程：

被除数的补码放在ACC中，除数的补码放在X中，X=[x]补，MQ初始化为0

ACC，X采用双符号位，MQ单符号位，符号位参与运算

1. ACC+X（被除数补码+除数补码）
2. 新余数与X（除数补码）同号/异号，MQ最低位商1/0（同或）
3. 同号/异号，余数逻辑左移-X（即+[-x]补，2a-b）/余数逻辑左移+X（+除数补码，2a+b），MQ同时逻辑左移
4. 最后一步，无论如何都商1（误差不会超过2^-n）

若尾数除数n位，执行一次（1））后，循环执行n次（2）（3），然后执行一次（4）

最后要注意，余数在ACC是双符号位前两位决定符号，还要/2^-n,MQ是单符号位，最高位决定符号



强制类型转换：

计算机中都是存储补码，有符号数最高位存符号位，unsign全部存数值

1. 有符号数转成相同字节的无符号数

二进制不变，所以符号位会被解析成数值，所以负数-x转换后!=x，正数则无影响

1. 有符号长数据转成有符号短数据

直接取低字节，丢弃高字节，需要注意的是，转换后最高位变成符号位，这就会造成转换前后符号有可能不同的情况

1. 短数据转换成长数据（符号扩展）

扩充字节，与（1）（2）不同的是，会对符号位进行处理放到长数据的最高位（整数才需要，小数因为在尾数后边扩充，不需要处理），所以转换前后真值不变

总之，只有（3）不用考虑符号位，真值不变，（1）会把符号位解析成数值位，（2）把原本的数值位解析成符号位

数据的存储和排列：

一串二进制数，左高位右低位，最高/最低有效字节缩写MSB/LSB，数据是连续存储的

计算机通常是按字节编址（每个字节对应一个地址），可按字，按半字，按字节寻址（一个字就是一个存储单元，字=存储字长，半字=存储字长/2），每次访存只能读写一个字

设字=8\*n，则按字寻址时，字的首字节地址= 字编号逻辑左移logn位（即\*n）

半字的首字节地址=半字编号逻辑左移logn-1位（即\*(n/2)）

排列方式：解析数据是从低地址到高地址哦

大端模式：高字节存放到低地址，适合于读取数据，更符合人类阅读习惯

小端模式：高字节存放到高地址，适合于需要优先取得低字节数据的情况

由于计算机每次访存只能读写一个字：

边界不对齐：每个字的全部空间都存储数据，会有一个数据的字节在不同字的情况

边界对齐：一个字只存储一个数据的所有字节，剩余的空间用0填充。访问一个数据只需要访问一个字，而边界不对齐可能会出现访问一个数据会访问两个字的情况，边界对齐是一种空间换时间的策略。

1. 浮点数的表示和运算：

表示：

浮点数解决了定点数范围的局限性，能表示更大的数据

a^n，a是尾数，体现精度，n是阶码，体现范围和小数点的实际位置，都有符号，阶码的符号叫阶符

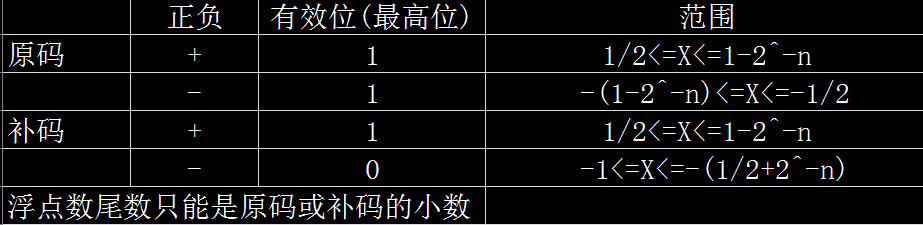
二进制浮点数，小数点可以在任意位置，小数点前面是阶码（补码或移码表示的定点整数），小数点后是尾数（原码或补码表示的定点小数），真值=尾数\*r^阶码（r可以取2,4,8,16……，通常是2），r^阶码是算术移位

书写时，用分号分开阶码和尾数，阶码和尾数各自用点或逗号分开符号和数值

浮点数的规格化：

左规：浮点数尾数的树脂部分最高位必须是一个有效位，否则尾数算术左移n位，阶码-n

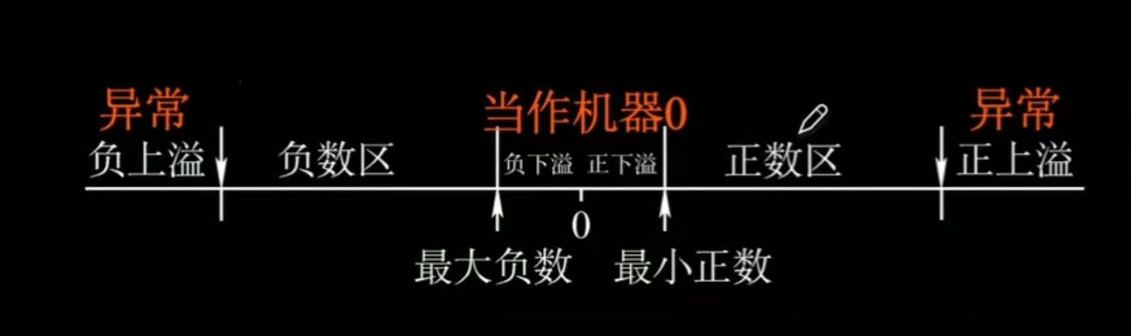
有效位以及规格化的尾数范围：



右规：双符号位发生假溢出时，浮点数尾数逻辑右移1位（补位与符号位一致），阶码+1

浮点数的表示范围：结合规格化的尾数范围和阶码范围

当发生下溢时，看成0，上溢时则异常



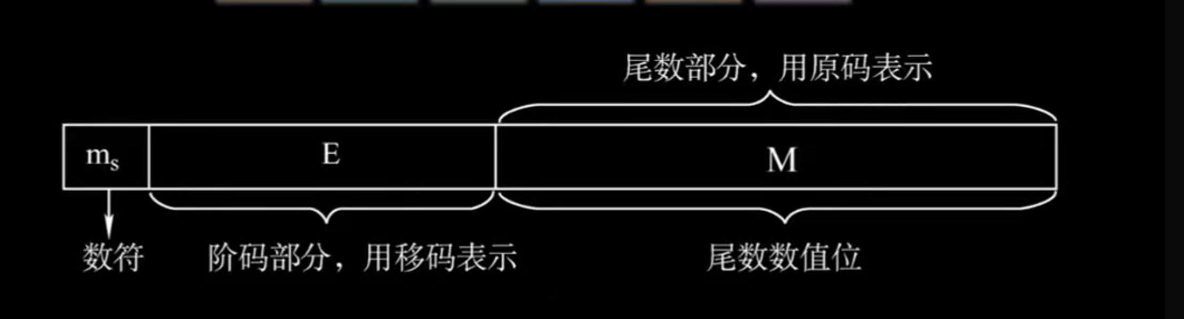
IEEE 754标准：

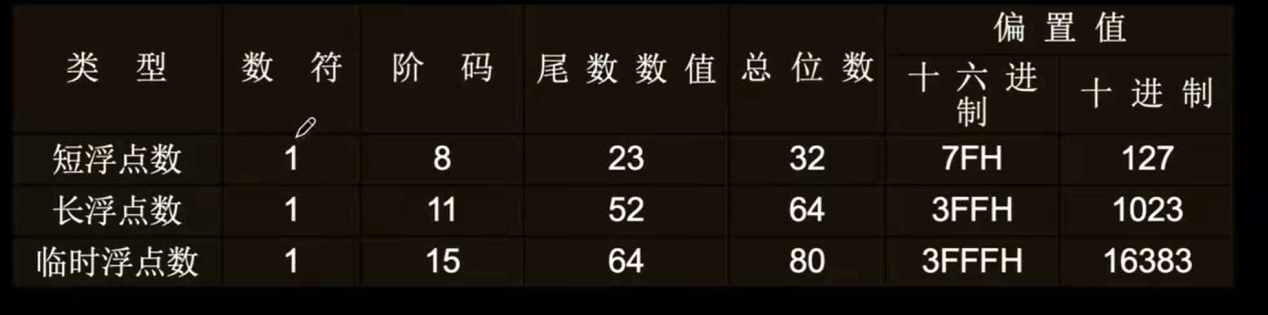
标准化前，阶码是移码或补码的整数，尾数是原码或补码的小数，各占多少位不确定

数符：在最前面，表示尾数符号

阶码：包括阶符，阶码。用移码表示（移码只能表示整数与阶码契合），这里的移码的偏置值是2^(n-1)-1（原来是2^(n-1)），且表示范围少了两个最小值（如8位范围变成-126到127，-128和-127的移码分别是全1和全0，用作特殊用途）

尾数：用原码表示，小数点在尾数最高位前，且在小数点前隐含一个有效位1，所以尾数应结束为1.M，多了一个隐含位





以下均以float举例，其他只要改变偏置值即可

真值转换成浮点数：

-0.75 二进制-0.11

数符：1

尾数：1.1隐含掉1=0.1

阶码：0.11=1.1\*2^-1，原码=1.0000001，移码=-1+127=126=1.1111110

所以浮点数为：1 11111110 10000000000000000000000

最小/最大绝对值：

最小：阶码真值=-126，尾数=1.000…..

最大：阶码真值=+127，尾数=1.111……

全0和全1的特殊处理：

阶码真值=-128/-127,时，移码全1/全0

全0：

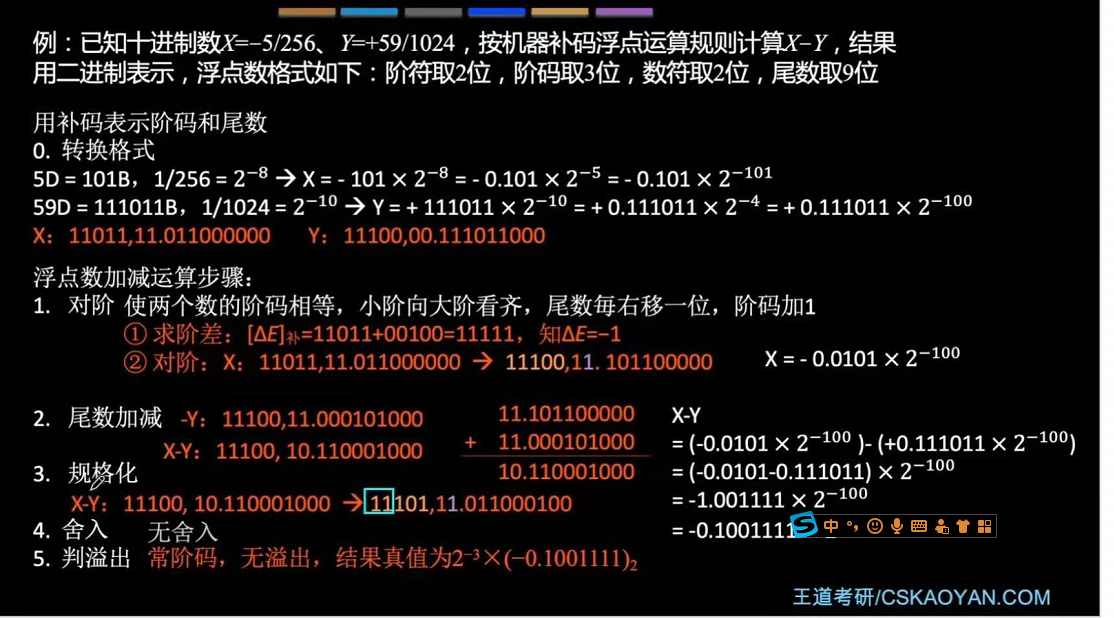
1. 当要表示小于最小绝对值时使用，此时尾数隐含位变为0，阶码真值看做-126，如1.001\*2^-129=0.001001\*2^-126，阶码存全0，尾数存001001000…..（小数点前隐含0）
2. 表示+0或-0时使用，阶码尾数全为0

全1：

1. 阶码全1，尾数全0，表示正无穷或负无穷
2. 阶码全1，尾数不全为0，表示NaN，出错

运算：

1. 十进制真值->二进制真值->原码机器数->补码机器数（阶码尾数都是补码）
2. 对阶（小对大，算术移位），，阶码相减得到的补码转换的原码的真值反映两者大小以及对阶数
3. 尾数加减
4. 规格化（左规右规）
5. 舍入，如需要舍入，方法1：0舍1入，舍去了0/1，不处理/结果+1，舍入后可能会需要继续右规舍入，方法2：恒置1，最低位置1
6. 溢出判断：下溢变0，上溢异常需处理



强制类型转换：

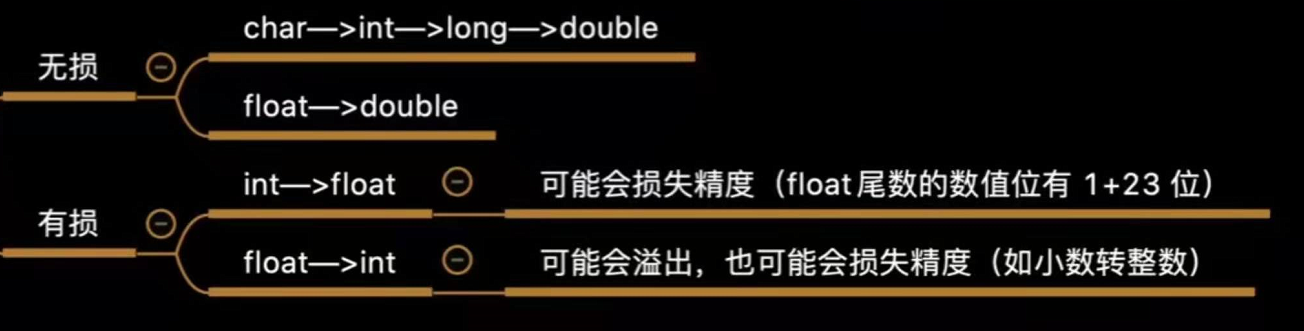
不同机器字长各数据类型字节数；



32位机器字长下，long32位，double尾数53位，精度无损

64位机器字长下：long64位，double尾数53位，精度有损

32位下：



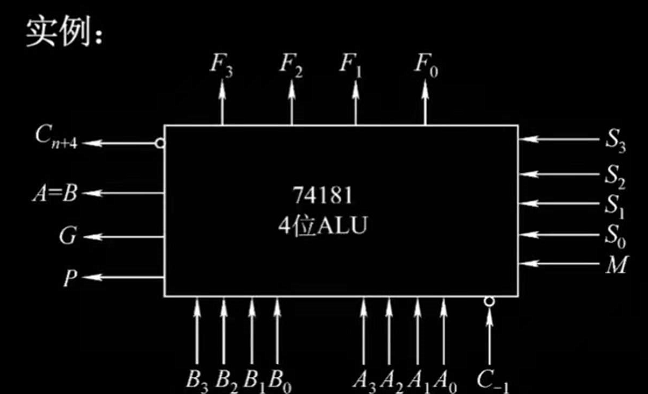
因为float范围比int大，所以int->float不会溢出

1. ALU和加法器：

ALU功能：算术运算，逻辑运算，移位运算，求补，[x]补=[-x]补

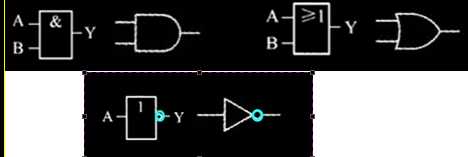
机器字长=ALU位数=寄存器位数

如图一个4位ALU（机器字长4位），A0-A3,B0-B3分别是输入两个四位二进制数，M和S0-S3是控制信号（M决定算术还是逻辑运算，S支持决定16种运算，共支持32种运算），F0-F3输出运算结果

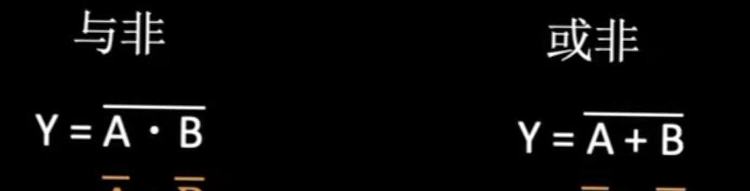


门电路：

与或非门的分别两种画法：



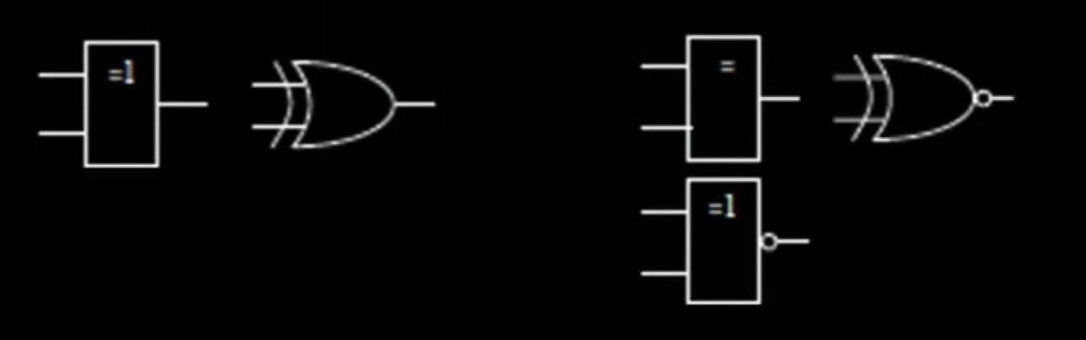
与非，或非的书写两种画法：





异或，同或的书写和两种/三种画法：





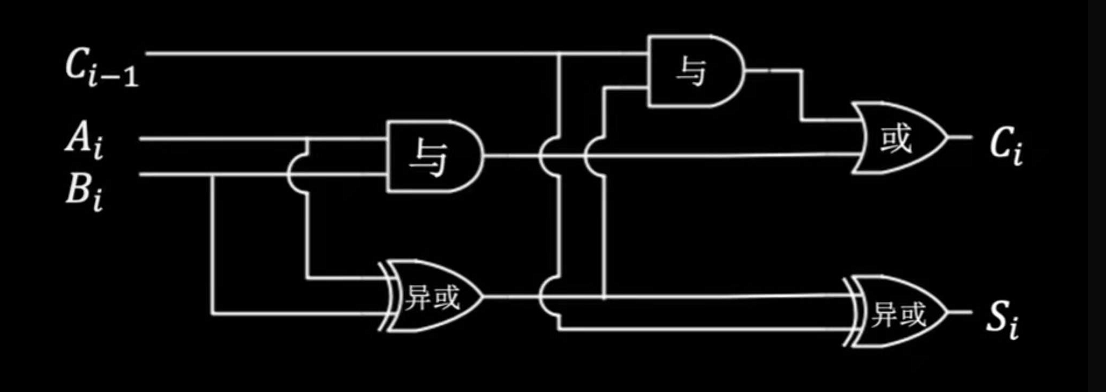
一位全加器（FA）：

逻辑表达式：Ai,Bi是本位值，Si为结果值，Ci是本位进位，Ci-1是上一位进位



至少两个1，Ci才为1

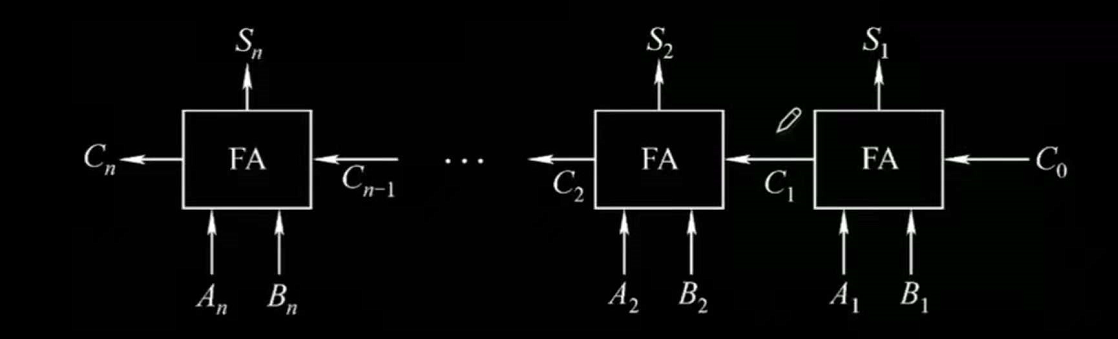
电路图：



n位全加器：

串行加法器：串联n个FA，用一个进位触发器存储进位值，缺点是速度慢，只有完成上一个FA才能进行下一个FA

串行进位的并行加法器（又称形波进位）：并行输入A,B，并行完成A异或B，A与B，但是进位C是串行的

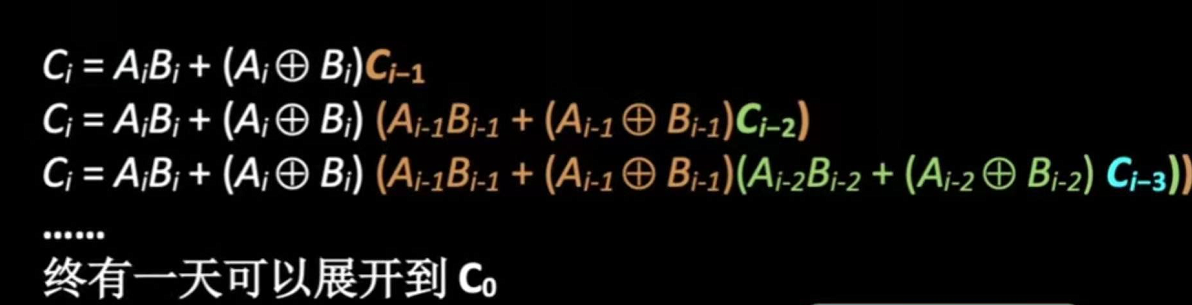


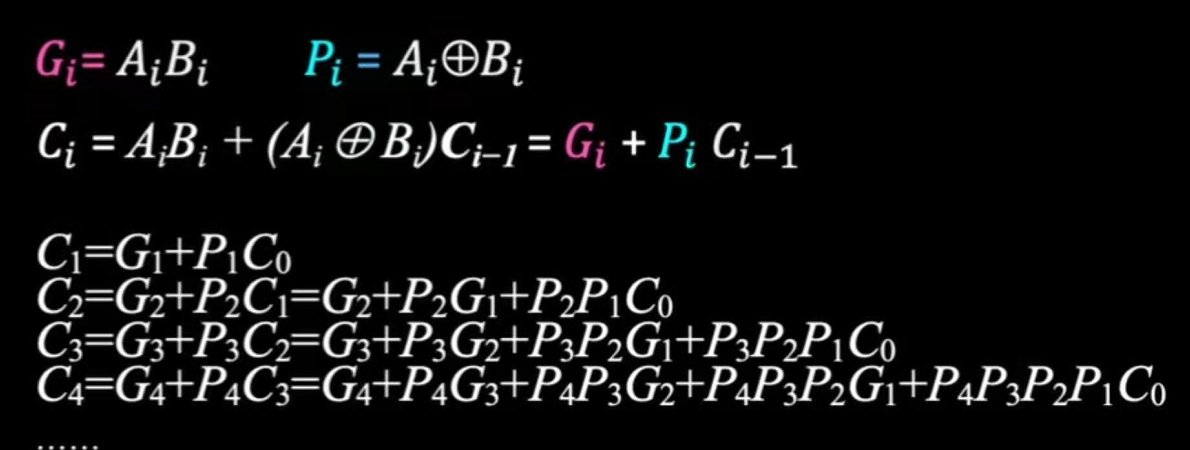
并行进位的并行加法器的（又称先行进位或并行进位）：

Ci类似于递归，每一个FA需要的进位其实可以由A0-A1,B0-B1，C0计算的出，而A,B是并行计算的，整体就是并行的。

但是位数越多，最后一位的逻辑表达式就越复杂，一般并行计算4位。

原理：





Gi叫进位产生函数，因为Gi=1一定有进位

Pi叫进位传递函数，因为Pi=1，是否有进位就只与上一位进位有关

三、存储器