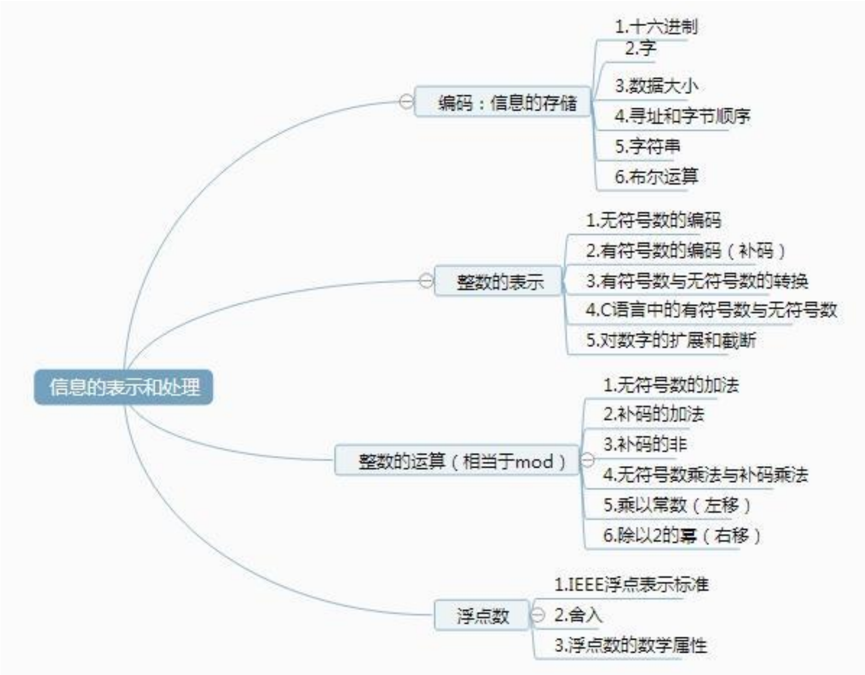
《深入理解计算机系统》| 信息的表示和处理

唐鱼的学习探索 关注



本章目录

[学习信息的存储（编码）和处理有什么用？]

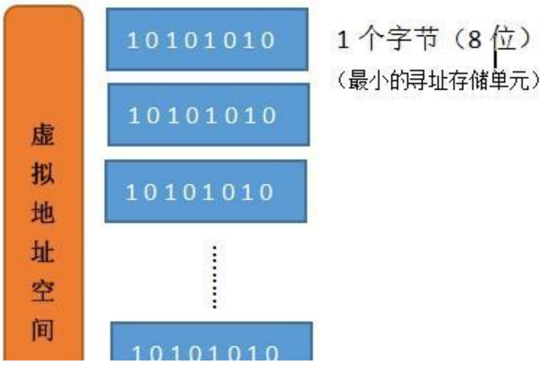
研究数字在计算机中是如何存储的，以及值的范围和算术属性，有助于我们跨越不同的机器、系统以及编译器获得更好的可移植性。了解这些细节非常重要，程序员有责任和义务编写健壮的程序，了解其内部如何工作，其不良行为背后的原因，对于安全领域也有非常高的价值。

[本章是如何展开的？]

本章首先对计算机是如何存储信息（编码）进行了讨论，中间涉及了二进制、十六进制数据的表示、大小以及如何顺序存储、布尔运算等，然后研究了三种重要的编码方式：无符号、补码（有符号）以及浮点数的编码。

[主要笔记]

一、信息的存储：编码



已字节为最小寻址单元进行数据的存储

单个的位没啥用处，当把位组合在一起（字节8个位），再进行某种解释，赋予不同的含义，我们就能表示世间万物了。每个程序都可以简单的视为一个字节块，程序本身就是一个字节序列。

1.十六进制

由于二进制信息太过冗长，于是在描述位模式的时候不是很方便，就发明了16进制。这里没什么好说的了，如下表：

十六进制数字	0	1	2	3	4	5	6	7
十进制值	0	1	2	3	4	5	6	7
二进制值	0000	0001	0010	0011	0100	0101	0110	0111
十六进制数字	8	9	A	B	C	D	E	F
十进制值	8	9	10	11	12	13	14	15
二进制值	1000	1001	1010	1011	1100	1101	1110	1111

16进制表示法

❤️：学习过计算机的同学对这个内容都不陌生，关于各个进制之间的转换作者让我们记住：A C F对应的十进制，然后推出BDE的值：

A	C	F
10	12	15

记住这三个值

还有一个简单的计算诸如：

$$2048 = 2^{11}$$

可以写成幂次方

可以使用公式：n = i + 4j 其中n = 11 ；i的取值范围是[0-3]对应的值为：0对应1,1对应2,2对应4,3对应8相当于2的i次幂，j就代表多少个0。回到上面的例子中，11 = 3 + 4 X 2 就可以写成0X800（j=2两个0，i=3对应8）。算是奇技淫巧吧，了解一下就可以了。

对于16进制和10进制的相互转化就无非是反复乘以或者除以16，也没啥好说的了。

2.字

虚拟地址空间是一个非常大的字节数组

我们前面说过虚拟地址空间可以使得我们很方便的范围到每个字节，但是虚拟地址是以一个字来进行编码的，所以字的长度就决定了我们能范围的最大范围。对于我们使用的32位的计算机而言，程序最多范围2的32次方个数据，也就是我们经常所说的4GB

3.数据大小

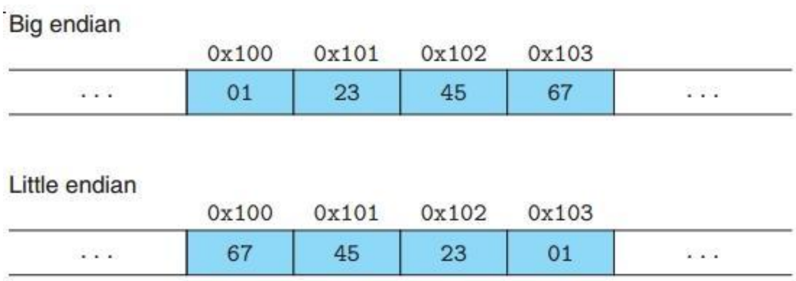
C 声明	32 位机器	64 位机器
char	1	1
short int	2	2
int	4	4
long int	4	8
long long int	8	8
char *	4	8
float	4	4
double	8	8

C语言中的数据类型 字节数

关注数据大小的原因是使得程序对于不同数据类型的大小不敏感，*如果我们用一个int类型（4字节）来存一个指针（64位下可能是8字节）就会带来不小的麻烦。*

4.寻址和字节顺序

对于跨越多个字节的数据、指令和控制信息，我们必须要知道他的地址是什么，以及是按照什么顺序在计算机中存储的。对于同样的一个数字：1234567；有两种存储方法：



大端法和小端法存储

就数据1234567来说，它跨越了4个字节，从0x100开始到0x103结束，我们除了必须要知道开始的地址0x100外，还有一个重要的就是必须要了解是何种顺序在计算机中存储的。就具体的应用来说，至少有下面三个方面：

①通过网络在不同机器以及系统中传递数据时，必须要遵守建立的字节顺序；

②强制类型转换：不会改变真实指针，只是告诉编译器以新的类型来解释数据；

③阅读表示整数的数据类型时（不是很理解）

5.字符串：文本数据比二进制数据有更强的平台独立性

字符串是以null（0）字符结尾的字符数组，在任何系统上面都能看到相识的结果。但二进制机器码就不一样了：

```
int sum(int x, int y) {
    return x + y;
}
```

c语言代码

编译成不同的机器码的结果如下：

Linux 32:	55 89 e5 8b 45 0c 03 45 08 c9 c3
Windows:	55 89 e5 8b 45 0c 03 45 08 5d c3
Sun:	81 c3 e0 08 90 02 00 09
Linux 64:	55 48 89 e5 89 7d fc 89 75 f8 03 45 fc c9 c3

不同系统的生成的二进制码

因此：二进制代码是不兼容的，从机器的角度来看，程序仅仅只是字节序列。

6.布尔运算

关于布尔运算的基本的方法与、或、非、异或就不在叙述了。除了判断逻辑以外讲讲有什么用：

掩码运算：举一个例子，我们要用守蒙住脸蛋防止别人看到我们的脸，但是我们又很想看看对方长啥样子，这时候就会留一个缝隙，让眼睛可以往外边看。这基本上就是掩码的功能了。我们有选择的屏蔽了一些信号，如长相。又如：0xFF（1111 1111）任何一个数与上0xFF，就能将最低的8位保留下来。

位运算与逻辑运算的区别：逻辑运算认为非0就是true，而0表示false；逻辑运算如果第一个表达式能确定结果就不会对第二个求值。

移位运算：

两种移动方式

❤️ 对于无符号的数，右移动必须是逻辑上的。对于有符号的数，可以是任何一种，但常用的是算术右移动

二、整数的表示（存储）

> 本节首先介绍了两种编码方式，一种只能表示非负数（无符号编码），另外一种可以表示负数、0和正数（补码编码），然后讨论了这两组编码的数学属性和机器实现，最后对于一个已知编码的扩展和收缩的方法进行了介绍

C语言支持的数据取值范围：

C 数据类型	最小值	最大值
char	−127	127
unsigned char	0	255
short [int]	−32 767	32 767
unsigned short [int]	0	65 535
int	−32 767	32 767
unsigned [int]	0	65 535
long [int]	−2 147 483 647	2 147 483 647
unsigned long [int]	0	4 294 967 295
long long [int]	−9 223 372 036 854 775 807	9 223 372 036 854 775 807
unsigned long long [int]	0	18 446 744 073 709 551 615

c语言的整型数据保证的取值范围

我们将介绍这些具体的取值范围是如何得来的，以及之间转换所遵循的规则，不知道大家有没有注意到一点：有符号数的范围并不对称，负数的范围比正数大1？我们接下来的内容会告诉大家原因

1. 无符号数的编码

我们来探究一个公式，完成二进制到无符号数的编码（Binary to Unsigned）我们编号为（2.1）：

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i \qquad (2.1)$$

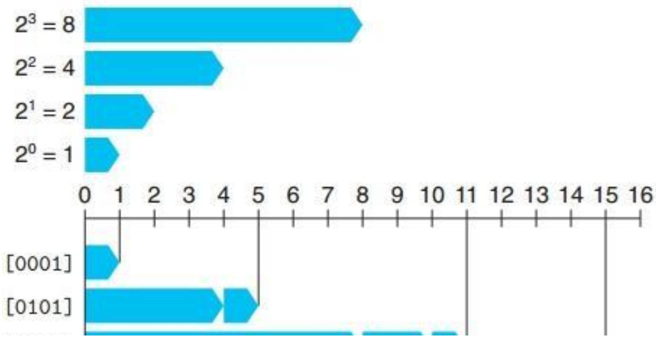
Binary to Unsigned

用几个例子和一幅图帮助大家理解：

$$\begin{aligned}
B2U_4([0001]) &= 0\cdot 2^3+0\cdot 2^2+0\cdot 2^1+1\cdot 2^0 &= 0+0+0+1 &= 1\\
B2U_4([0101]) &= 0\cdot 2^3+1\cdot 2^2+0\cdot 2^1+1\cdot 2^0 &= 0+4+0+1 &= 5\\
B2U_4([1011]) &= 1\cdot 2^3+0\cdot 2^2+1\cdot 2^1+1\cdot 2^0 &= 8+0+2+1 &= 11\\
B2U_4([1111]) &= 1\cdot 2^3+1\cdot 2^2+1\cdot 2^1+1\cdot 2^0 &= 8+4+2+1 &= 15
\end{aligned}$$

各位相加求和

这是对w=4位的几个数字的无符号数的编码，很好理解，就是各个位具体的值和每个位的权值相加，用如下的图来表示就更更清楚了：



用长度为2的i次方表示向量x的长度

在无符号的表示中，统一都用的是向右向量来表示，各个位的权长度不一样，最高的是8最低的是1。我们来看看表示的范围是多少：从最小的无符号数[0000]到最大的无符号数[1111]范围就是0-15（最大值为2的w次方-1）。

2.有符号数补码的编码（Binary to Two's-complement）：

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \tag{2.3}$$

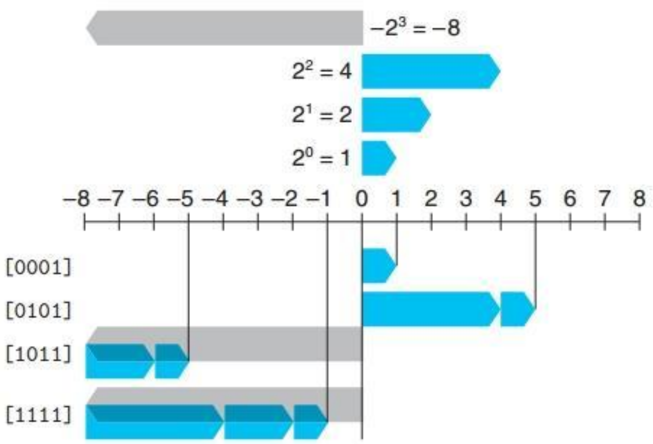
补码编码公式

用实际的例子表示为：

$$\begin{aligned}
B2T_4([0001]) &= -0\cdot 2^3+0\cdot 2^2+0\cdot 2^1+1\cdot 2^0 &= 0+0+0+1 &= 1\\
B2T_4([0101]) &= -0\cdot 2^3+1\cdot 2^2+0\cdot 2^1+1\cdot 2^0 &= 0+4+0+1 &= 5\\
B2T_4([1011]) &= -1\cdot 2^3+0\cdot 2^2+1\cdot 2^1+1\cdot 2^0 &= -8+0+2+1 &= -5\\
B2T_4([1111]) &= -1\cdot 2^3+1\cdot 2^2+1\cdot 2^1+1\cdot 2^0 &= -8+4+2+1 &= -1
\end{aligned}$$

最高位为符号位

最高位为符号位，当为0的时候没什么影响，为1的时候加的就是负权了，如下图：



最高位用灰色表示为符号位，向量向左

我们来看看不对称性的根源？

这种对称性是由于一半的位模式（符号位设置为1）表示负数，而另外一半的位模式（符号位设

置为0）表示非负数，因为0是非负数，也就意味着能表示的正数比负数要少一个。

同样来讨论一下取值范围，最小的负数TMin为[1000]结果为-8，最大的正数TMax[0111]结果为7，我们看得出来|TMin| = |TMax| + 1，最小的负数TMin没有与之对应的+8，这种不对称的特殊性，正是由于0也是非负数，所以-8就没有与之对应了正数值了。

另外，有符号数还有其他两种表示方法：反码和原码，由于运用的非常少，大多数机器都采用的是补码的方式，我们就不再研究了。

3.有符号与无符号数的转换：

> 类型转换的结果是保存位值不变，只是改变了解释这些位的方式

从存数学的角度考虑，我们能想到的规则是：首先对于两者之间的交集，我们保持不变；其次，对于超出范围的值，比如将最大的负数（前文中说的-8无对应的情况）转换成无符号可能会得到0，将无符号的数（太大的部分）转换为有符号的数可能会得到TMax，举个例子：

以w=4位为例：有符号数能表示的范围是：[-8--7] 包含两端的-8和7而无符号的表示范围是：[0--15] 这样，当我们在[0-7]之间的数的进行转换的时候将保持不变；而在进行诸如：无符号数[8-15]转有符号数的时候就只能用TMax表示；有符号数[1111]=-1转到无符号数就变成了[1111]=15。注意看下面这个图：

权	12 345		−12 345		53 191	
	位	值	位	值	位	值
1	1	1	1	1	1	1
2	0	0	1	2	1	2
4	0	0	1	4	1	4
8	1	8	0	0	0	0
16	1	16	0	0	0	0
32	1	32	0	0	0	0
64	0	0	1	64	1	64
128	0	0	1	128	1	128
256	0	0	1	256	1	256
512	0	0	1	512	1	512
1 024	0	0	1	1 024	1	1 024
2 048	0	0	1	2 048	1	2 048
4 096	1	4096	0	0	0	0
8 192	1	8192	0	0	0	0
16 384	0	0	1	16 384	1	16 384
±32 768	0	0	1	−32 768	1	32 768
总计	12 345		−12 345		53 191	

12345和-12345的补码表示，以及53191的无符号数表示

-12345同53191有同样的位表示，到这里我们就明白了，在计算机内部的实现方式是：

规则：数值可能会改变，但位的模式不变。

从数学角度来讨论这个规则：

我们定义B2U（）的逆运算为U2B（）；定义B2T（）的逆运算为T2B（）；

那么完成U2T（）的转换就相当于：

等式一：U2T（） = B2T（U2B（））

解释一下就是，先完成无符号到二进制的转换U2B，然后在将二进制转换层有符号数B2T

同样的道理完成T2U（）的转换就相当于：

等式二：T2U（） = B2U（T2B（））

我们先来看看U2T（无符号转有符号）之间的转换公式的推导过程：

假设w=4位

①无符号数B2U计算公式：[1111] = 1 * 8 + 1 * 4 + 1* 2 +1 = 15

②有符号数B2T计算公式：[1111] = -1 * 8 + 1 * 4 + 1 * 2 + 1 = -1

用①-② 也就是B2U（）-B2T（） 后面的内容1 * 4 + 1 * 2 + 1可以抵消掉，其实就是头两个数之间的差值：

B2U（）- B2T（） = 1 * 8 + 1 * 8 = 1 * （8 + 8）= 1*16

❤️ 也就是1 乘以2的4次方，写成公式为：

$$B2U_w(\vec{x}) = x_{w-1}2^w + B2T_w(\vec{x}).$$

有符号数转无符号数：

令x = T2B（x），带入上述公式转换的结果为：

$$\text{B2U（T2B（x））} = \text{B2T(T2B（x））} + X_{W-1} * 2^w$$

由于B2U(T2B) = T2U;B2T(T2B)=x。所以上面的等式可以写成如下形式：

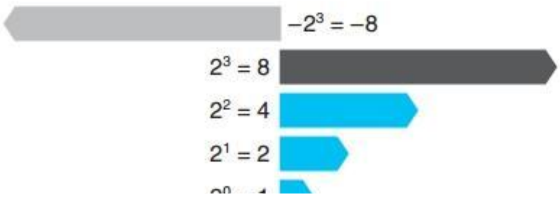
$$B2U_w(T2B_w(x)) = T2U_w(x) = x_{w-1}2^w + x$$

这个公式由于x的最高位w-1位的属性，又决定了以下两种形式，得到：

$$T2U_w(x) = \begin{cases} x+2^w, & x<0 \\ x, & x \geqslant 0 \end{cases}$$

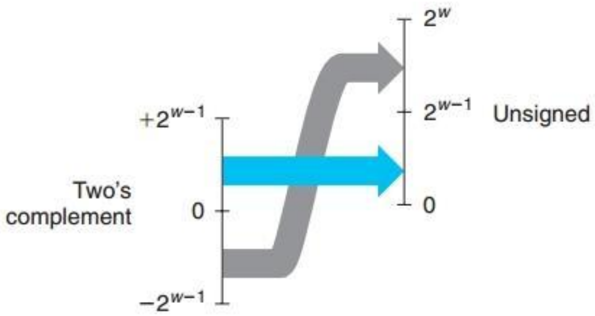
w-1位为0的话，值就是x

进一步解释一下就是，以w=4位为例，[0001]-[0111]之间的有符号数转无符号数其值是不变的。如果无符号数[1001]-[1111]之间的数由于最高位有值，那么结果就会加上2的w次方。比如：将-5 = [1011]转换为无符号数就是：11 （-5+16），而-1就变成了15.如下图看到的



符号位为向左的灰色箭头表示

而T2U的一般行为就是，非负数保持不变，而负数就变成了大的正数



负数就变成大的正数

至此我们探究了T2U的转换内幕，以及其行为，下面我们来看看U2T是如何工作的

无符号数转有符号数：

将x = U2B(x)带入标❤️公式：我们得到

$$B2U(U2B(x)) = B2T(U2B(x)) + X_{W-1} * 2^{w}$$

$$x = U2T + X_{W-1} * 2^{w}$$

$$U2T = -X_{W-1} * 2^{w} + x$$

简单带入变形

我们将上面的等式整合一下，得到了U2T的公式为：

$$U2T_w(u) = -u_{w-1}2^{w} + u$$

综合公式

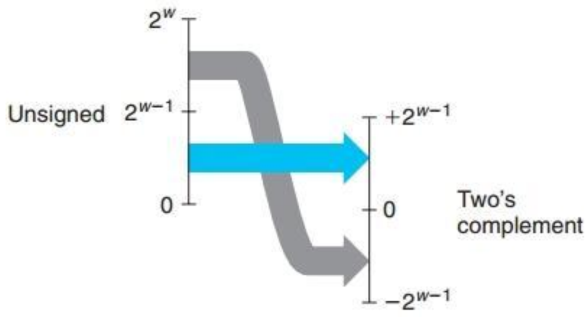
根据无符号位w-1时候大于或者等于2的w-1次方，我们将上个公式可以写成下面这种形式：

$$U2T_w(u)=\begin{cases} u, & x<2^{w-1} \\ u-2^{w}, & u \geqslant 2^{w-1} \end{cases}$$

无符号转有符号数公式

同样的我们以w=4位为例，完成对[0-7]转有符号数的时候，保持本身值不变。而如果是对于≥8的数转换为有符号数，[1001]-[1111]之间的数，由于最高位要解释成符号位，所以结果会减去2的w次方。也就是上述公式所显示的内容，将无符号数[1001]=9转换为有符号数为[1001] =

-7 = 9 - 16.



将大于8的数转化为负值

4.C语言中的有符号数与无符号数

C当然没有指定有符号数用什么编码方式，但是各个机器基本上都是使用补码的形式表示。所以，如果要创建一个无符号数常量，那么就必须要申明加入后缀‘U’或者'u'的形式。

特别是类型转换中的隐形转换，笔录表达式赋值给另外的一个变量，就容易被忽略，很难发现错误之处。其实一个为了避免出错，最好的一个做法就是**尽量不使用无符号数。**

（另外提一下，**无符号数有什么用处：**当我们想把字仅仅当做位的集合来看，没有任何数字意义的时候，无符号数还是非常有用的。例如：往字中放入描述各种布尔条件的标记时，就形成了地址，而地址当然是能访问的越多越好）

C语言中的转换规则，如果执行一个运算，它的一个运算符是有符号另外一个无符号，那么C就会隐式的件有符号的参数强制类型转换为无符号数，并假设这两个都是非负数来进行计算。这里特别需要**注意的就是像＞或者＜结果可能出错。**

如比较：-1 ＜0U的值时，先将-1转无符号数：4294967295 ＜0U 就有问题了。

我们来说说为什么-1转有符号的数会是一个这么大的数，

同样的以w=4位为例：-1 = [1111] 转换为无符号数，由于保持位值不变，只是改变解释那么无符号数[1111] = 15 也就是能表示的最大值，4294957595是采用补码的32位机器所表示的最大无符号数。

5.对数字的位进行扩展与截断的技术

①扩展一个数字的位：

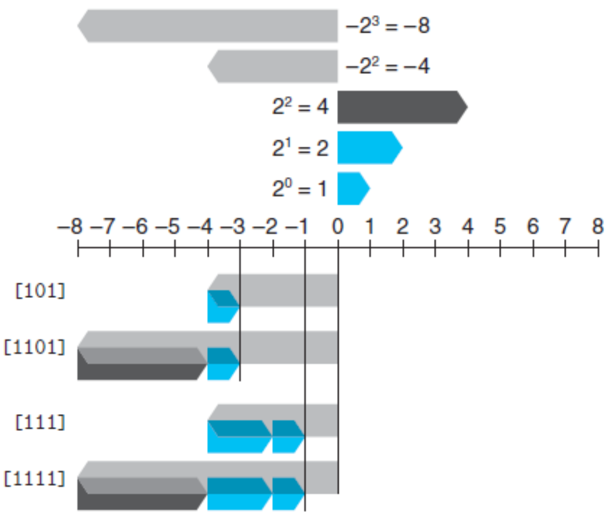
这里在一个程序中应用的特别多，比如将一个short类型的数据转换为int，或者将unsigned short 转为unsigned int 这种数字长度的增加，到底是依据如何的规则进行的转换，大致说来有两种扩展方式：零扩展和符号扩展：

1>（零扩展）无符号转更大数据：在数据的开头位添加0即可

2>（符号扩展）有符号转更大数据：在数据的开头添加最高位的副本

零扩展不用说会保持数值不变，我们来研究一下符号扩展：假设字长从3为变成了4，有补码表示的数[101] = -3 将变成[1101] = -3 即使增加了1位结果任然是一样的，是什么原因保证了这

种变化保存了数值的不变呢？我们来探究一下：



从3位增加到4位

这里有一个关键属性就是：就是上图所示的，当我们做最开始的两位的加法的时候，其结果与上一等式的第一个数的值相同。

$$[101] = -1\times2^2 + 0\times2^1 + 1\times2^0$$

$$[1101] = -1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$$

从3位扩展到4位

$$-1\times2^3 + 1\times2^2 = -1\times2^2$$

前两位的折返与原数相同

用更普遍的观点来看，关键的属性就是：

$$2^w - 2^{w-1} = 2^{w-1}.$$

结果会保留原始的值

另外要提一下，将一个数据大小改变，和无符号有符号之间这两组转换的相对顺序，会影响一个程序的行为。如果将short转为unsigned时，**就会先改变大小，然后再完成有符号到无符号的转变**。如：short sx = -12345 转为 unsigend的uy时，先将0X CF C7扩展成0X FF FF CF C7然后再进行到无符号数的解释，结果就得到了4294954951这样的一个数字。

②截断一个数字的位：

对于无符号的数：[1111] = 15 截断1位，其实是将15 mod 8（2的3次方） = 7 [111]

对于有符号数：[1101] = -3 的截断，是先将[1101]转为无符号的13然后来mod 8 = 5无符号表示就是[101]，然后再把[101]解释成有符号数[101] = -3

6.注意事项：

1> 尽量避免使用无符号数

2> 特别留意隐藏的强制类型转换行为

三、整型的运算（相当于mod运算）

我们首先要来理解一下“字节膨胀”的概念：

比如我们以w＝4位为例，进行无符号数[1111]＝15和无1.符号数［1010］＝10的加法运算，结果为25=［11001］需要5位来表示结果，依次类推我们如果要完整的表示运算结果，就不能对字长做任何限制。大部分编程语言都选择了固定精度的加减乘除运算，会对结果进行一定的处理。也就与我们数学上的运算有所不同，这一节我们就来学习这些处理方法。

首先来看看加法运算：我们会接触到［无符号的加法］和［补码的加法］。这两组加法运算使用的是相同的机器指令

1.无符号数的加法：

$$x +_w^{\mathrm{u}} y = \begin{cases} x + y, & x + y < 2^w \\ x + y - 2^w, & 2^w \leq x + y < 2^{w+1} \end{cases} \tag{2.11}$$

相当于截断高位

溢出的真正含义就是：完整的结果不能放入到固定精度的字长中去，于是最高位就被丢弃掉了。减去2的w次方，相当于结果mod（2的w次方）。

⚠️ 如何判断是否溢出?

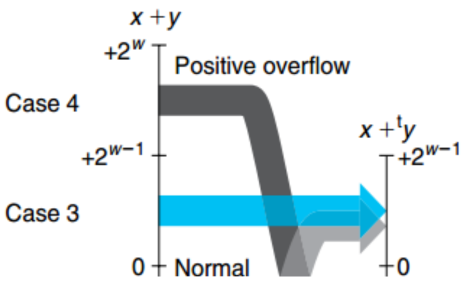
运行c程序的时候，溢出并不认为是一种错误。那么我们自己如何判断是否发生了溢出呢？我们可以设s＝x＋y，当结果s < x或者s < y的时候发生了溢出。（证明如下：前面我们2.11的第二种情况下有溢出，我们确定的是y<2的w次方，那么y－2dw就<0。两边加上x，结果就是x＋y－2dw < x发生了溢出）

2.补码的加法：

$$x +_w^{\mathrm{t}} y = \begin{cases} x + y - 2^w, & 2^{w-1} \leq x + y & \text{Positive overflow} \\ x + y, & -2^{w-1} \leq x + y < 2^{w-1} & \text{Normal} \\ x + y + 2^w, & x + y < -2^{w-1} & \text{Negative overflow} \end{cases} \tag{2.14}$$

正溢出产生负数，负溢出产生正数

为什么会有“**正溢出产生负数，负溢出产生正数**”？



正溢出减去了2的w次方，负溢出加上了2的w次方

举例说明：

正溢出［0101］＋［0101］＝5＋5＝10＝［01010］截断最高位0结果为[1010]=-6;

负溢出［1000］＋［1011］＝−8＋−5＝−13＝［10011］截断高位1结果为[0011]=3.

主要的原因还是我们使用的是固定精度的运算，由于结果不能被完整的保存，我们就需要使用截断高位保存低位的方法。这样做由于正溢出是两个大正数相加，完整的结果仍然数正数，截断最高位相当于减少了2的w次方；而负溢出数两个大负数相加，完整的结果仍然数负数，截断高位1以后相当于加上了2的w次方。

3.补码的非

$$-^{t}_{w}\, x = \begin{cases} -2^{w-1}, & x = -2^{w-1} \\ -x, & x > -2^{w-1} \end{cases} \qquad (2.15)$$

设置不能表示的数的不骂为它本身

以w=4位为例，补码的表示范围在[-8，7]之间，也就是说[-7，7]内的数可以表示为-x，但是对于最小的TMin=-8的情况怎么办呢？C语音中求解补码的方法是：每位求反，结果加1.

就如[0101] = 5 每位求反为 [1010] 再加上1为：[1011]补码表示为-5。那么同样的方法计算[1000] = -8的求反[0111]再加上1的结果还是[1000] = -8我们就认为的定义了：-8的非就是-8，也就是上个算式中显示的内容了。

4.无符号乘法和补码的乘法（使用相同的机器指令）

无符号的乘法：我们知道如果是w位的两个数相乘，结果最大可能是2w位，C语音中仍然使用的是固定精度的运算，这就导致了结果只能截断到w位，只保留真实值的低w位。计算公式为：

$$x *^{u}_{w} y = (x \cdot y) \bmod 2^{w} \qquad (2.16)$$

截断高w位相当于mod（2w次方）

补码的乘法：使用的是同无符号数相同的机器指令，并且在低位是相同的

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	5	[101]	3	[011]	15	[001111]	7	[111]
Two's comp.	−3	[101]	3	[011]	−9	[110111]	−1	[111]

相同位表示的不同数，其乘法运算结果的低w位是一样的

公式表示为:

$$x *^{t}_{w} y = U2T_w((x \cdot y) \bmod 2^w) \qquad (2.17)$$

乘积截断高4位后转补码表示

关于低w位相同的证明其实很简单:

$$\begin{aligned}(x' \cdot y') \bmod 2^w &= [(x + x_{w-1}2^w) \cdot (y + y_{w-1}2^w)] \bmod 2^w \qquad (2.18)\\ &= [x \cdot y + (x_{w-1}y + y_{w-1}x)2^w + x_{w-1}y_{w-1}2^{2w}] \bmod 2^w \\ &= (x \cdot y) \bmod 2^w\end{aligned}$$

其中x”代表的无符号的值

由于结果中带有2的w次方的同mod2的w次方会丢弃掉。因此我们看到了，mod2的w次方留下的就是低w位，结果同无符号数是一样的。

5.乘以常数（左移）

乘法运算太慢了，需要10个时钟周期，于是编译器就使用移位和加法指令来替代乘以常数。

如：x * 14： 其中 14被分解为：2d3 + 2d2 + 2d1 （其中d代表次方）编译器的写为：(x<<3)+(x<<2)+(x<<1)一些聪明的编译器甚至改写成：(2<<4) - (2 << 1)这时候只需要两个移位指令和一个减法指令了。

6.除以2的幂（右移）

除法指令比乘法更慢，相当于30多个时钟周期，所以当除以2的幂的时候经常用右移来代替。

无符号数：逻辑右移动（左边空出来的位补0）

补码：算数右移（左边补出来的数加最高位的值）

舍入的方法：整数的除法是舍入到0的，其中对于负数的除法结果是向下舍入，如-7/2不是-3而是-4，这样做其实是使用的一种偏置值的方法。[X/Y] = [(X+Y-1)/Y]也就是原本是-30/4]=7.5却变成了[-27/4]=6.75向下舍入到-7

7.整数运算总结

整数运算不论是加减乘除，其本身来说就是一种mod运算。由于结果的固定精度，大的就可能会溢出。补码和无符号数使用的是相同的机器运算指令，有相同的位级表示。特别是无符号数的一些意向不到的行为，程序员特别需要注意。

四、浮点数

1.IEEE浮点表示标准：

$$V=(-1)^s \times M \times 2^E$$

s符号，m尾数，e阶码

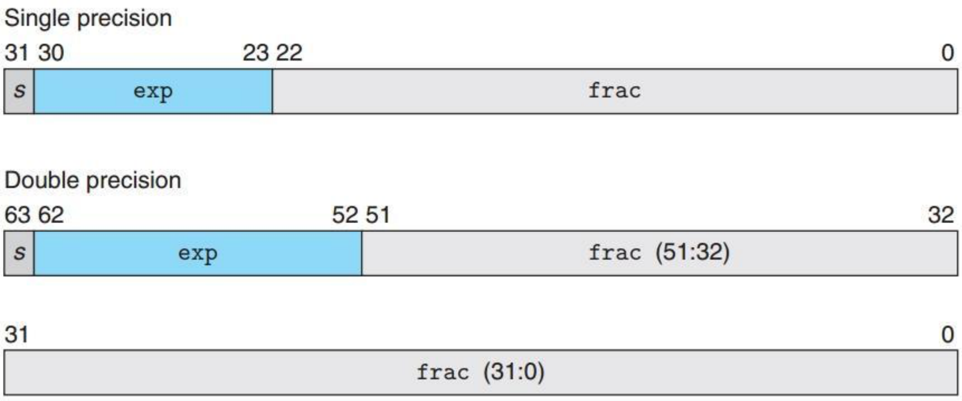
说明：

符号（S）：当s=1为负数，当s=0为正数；

尾数（M）：表示从（1~2）或者（0~1）之间的数；

阶码（E）：可以是负数

单精度和双精度的表示：



M为23或者52位，E为8或者11位

① 规格化值 E ≠ 0 或者 E ≠ 255 （E不全为0或者1）

阶码段被解释成有偏置值形式的有符号整数：E = e - Bias （其中Bias = 2的k-1次方 - 1；以8位为例就是127，11位为例为2047）。

M尾数定义为1+f 也就是隐含了已1开头，多表示了1位。

② 非规格化值：（E全为0时）

上一个方法中如何来表示0呢，非规格化就是在表示非常接近于0和0的数

M = f

E = 1 - Bias

③ 特殊值（E全为1时）

a.当M=0时得到无穷大s=0是正无穷大，s=1时是负无穷大；

b.当M≠0时得到了Not a Number（NAN）

应用举例：以w=6位为例

s1位，E有3位，M有2位

我们尝试来表示最大的规格化数字：（E不全为0或者1）

合成6位的表示就是[0] [110] [11]也就是：011011；

解释一下：

S = 0 结果为 +   偏置值为 3

那么E不全为1，能表示的最大值就是[110]表示为：E = 6-3 = 3

M最大为[11]也就是3/4由于M = 1 + f 所以结果为 7 / 4

那么最后的计算就是：+1 * 8 * （7/4）= 14

如果以8位为例，我们看图就很好理解了】

描述	位表示	指数			小数		值		
		e	E	2^E	f	M	$2^E \times M$	V	十进制
0	0 0000 000	0	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{0}{8}$	$\frac{0}{512}$	0	0.0
最小的非规格化数	0 0000 001	0	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$	$\frac{1}{512}$	0.001953
	0 0000 010	0	−6	$\frac{1}{64}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$	$\frac{1}{256}$	0.003906
	0 0000 011	0	−6	$\frac{1}{64}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$	$\frac{3}{512}$	0.005859
	⋮								
最大的非规格化数	0 0000 111	0	−6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0.013672
最小的规格化数	0 0001 000	1	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0.015625
	0 0001 001	1	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$	$\frac{9}{512}$	0.017578
	⋮								
	0 0110 110	6	−1	$\frac{1}{2}$	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$	$\frac{7}{8}$	0.875
	0 0110 111	6	−1	$\frac{1}{2}$	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$	$\frac{15}{16}$	0.9375
1	0 0111 000	7	0	1	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	1	1.0
	0 0111 001	7	0	1	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	1.125
	0 0111 010	7	0	1	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$	$\frac{5}{4}$	1.25
	⋮								
	0 1110 110	14	7	128	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{1792}{8}$	224	224.0
最大的规格化数	0 1110 111	14	7	128	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240.0
无穷大	0 1111 000	—	—	—	—	—	—	∞	—

8位浮点表示的非负数值

主要解释一下过渡阶段：

最大的非规格化数	0 0000 111	0	−6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0.013672
最小的规格化数	0 0001 000	1	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0.015625

平滑过渡

非规格化偏置被设置成1-Bias，而不是-Bias。我们其实是补偿非规格化的尾数没有隐含的1开头这一事实，这样7/512和8/512就实现了平滑的过渡。

我们来练习一下将整数12345二进制表示为[11000000111001]转化为单精度浮点表示，将二

进制左移13位表示为1.1000000111001 X 2的13次方。为了适应IEEE规格化表示，我们表示尾数M时丢弃掉最高位1，并在末尾增加10个0.M = [1000000111001 0000000000]。为了构造阶码字段我们用13+127 = 140 二进制表示为[10001100]再加上一个符号位0我们的最后结果就是[01000110010000001110010000000000]我们观察12345（0X3039）同浮点数12345.0（0X4640e400）的位级表示：

```
     0   0   0   0   3   0   3   9
00000000000000000011000000111001
                   *************
          4   6   4   0   E   4   0   0
          01000110010000001110010000000000
```

我们看到尾数M同0x3039正好是相差最高位1

《深入理解计算机系统》| 信息的表示和处理

相同点。

2. 舍入

方 式	1.40	1.60	1.50	2.50	−1.50
向偶数舍入	1	2	2	2	−2
向零舍入	1	1	1	2	−1
向下舍入	1	1	1	2	−2
向上舍入	2	2	2	3	−1

以美元为例的4种不同的舍入方法

计算机使用的是找偶数舍入的方法，为了避免统计上的误差，在一半的时间向下舍入，另一半的时间向上舍入。

3.浮点数的数学属性

由于舍入而产生的丢失精度，浮点数的运算中不具有结合性

C语言中的浮点数使用注意事项：

- 从 int 转换成 float，数字不会溢出，但是可能被舍入。
- 从 int 或 float 转换成 double，因为 double 有更大的范围（也就是可表示值的范围），也有更高的精度（也就是有效位数），所以能够保留精确的数值。
- 从 double 转换成 float，因为范围要小一些，所以值可能溢出成为 $+\infty$ 或 $-\infty$。另外，由于精确度较小，它还可能被舍入。
- 从 float 或者 double 转换成 int，值将会向零舍入。例如，1.999 将被转换成 1，而 −1.999 将被转换成 −1。进一步来说，值可能会溢出。C 语言标准没有对这种情况指定固定的结果。与 Intel 兼容的微处理器指定位模式 [10...00]（字长为 w 时的 $TMin_w$）为**整数不确定**（integer indefinite）值。一个从浮点数到整数的转换，如果不能为该浮点数找到一个合理的整数近似值，就会产生这样一个值。因此，表达式 (int)+1e10 会得到 −21483648，即从一个正值变成了一个负值。

【词汇】

程序对象：学习过汇编的同学应该不难理解，有点儿像程序的数据段、代码段，即是：程序数

据、指令和控制信息；

10人点赞 > 《深入理解计算机系统》 ···

"小礼物走一走，来简书关注我"

赞赏支持

还没有人赞赏，支持一下

写下你的评论... 评论4 赞10 ···

总资产29 (约2.83元) 共写了10.4W字 获得530个赞 共463个粉丝 关注

写下你的评论...

全部评论 4 只看作者

按时间倒序 按时间正序

sunyInTheSky
4楼 2018.03.18 15:13

😳好想不头晕

赞 回复

fxj0057
3楼 2018.01.11 11:34

无敌了，老铁

赞 回复

唐鱼的学习探索 作者
2018.01.11 22:39

@fxj0057 谢谢支持

回复

添加新评论

无处容身
2楼 2017.03.14 10:44

nice

赞 回复

被以下专题收入，发现更多相似内容

资料 《深入理解计算... 深入理解计算机... 系统

推荐阅读

更多精彩内容 >

第二章 信息的表示和处理

本章我们来研究三种重要的数字表示 无符号是基于传统二进制表示法，表示大于或等于0的数字 补码是表示有符号整数的最常...

程序员必修课 阅读 238 评论 2 赞 2

Charpter Two 信息的表示和处理

2.1 信息存储2.1.4 表示字符串独立性(文本数据/二进制数据)文本数据比二进制数据具有更强的平台独立性原因:...

1nfinity 阅读 33 评论 0 赞 0

深入理解机器码（原码，反码，补码）和算术溢出

如果你是一个计算机专业的本科生，那么你可能大一时就在《数字逻辑》（或《数字电路》）这本书里面学习了机器码。可能当时...

航航大魔王 阅读 9,332 评论 6 赞 23

计算机运算基础荟萃

二进制数的运算方法 电子计算机具有强大的运算能力，它可以进行两种运算：算术运算和逻辑运算。 1．二进制数的算术运算...

阿星Plus 阅读 292 评论 0 赞 0

唉

小俊俊的她 阅读 12 评论 0 赞 0