数制与基本运算

（2025秋）

高小鹏

北京航空航天大学计算机学院

目录

数的表示方法

- 在计算机中，任何对象都被表示为一组0/1串
  - 即使是上面这句话，也是由一组0/1串构成的！
  - 这就是所谓的二进制表示
- 如何用二进制表示数呢？
  - 让我们从熟悉的十进制开始介绍

十进制数的表示方法

- 示例：10进制的1234，其值的计算过程可以表示为

$$1234_{10} = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

4个10^0
3个10^1
2个10^2
1个10^3

- 从上例可以看出，每一位数都包含了基和权两部分
  - 基：base；权：weight

1234_{10}的角标10代表十进制。由于十进制是人类的习惯表达方式，因此在很多时候人们会省略这个角标。

数的一般表示方法

- 对于任意一个数，如$d_{n-1}d_{n-2} \ldots d_1d_0$，其值表示为

$$d_{n-1}\times B^{n-1} + d_{n-2}\times B^{n-2} + \ldots + d_1\times B^1 + d_0\times B^0$$

  - 其中d_{n-1}，d_{n-2}，$\ldots d_1$，d_0是该进制的可能取值
  - 例如十进制，可能取值为{0，1，2，3，4，5，6，7，8，9}

二进制的表示方法

- 二进制数的每位也被称为比特（bit）
- 二进制的基能够表示的数字范围只有2个数，即{0,1}
- N位二进制数的各位权从最低位到最高位分别为2^0，2^1，…，2^{N-1}
- 示例：11011_2

$$11011_2 = \underbrace{1\times 2^4}_{1个2^4} + \underbrace{1\times 2^3}_{1个2^3} + \underbrace{0\times 2^2}_{0个2^2} + \underbrace{1\times 2^1}_{1个2^1} + \underbrace{1\times 2^0}_{1个2^0} = 16+8+0+2+1 = 27_{10}$$

常见数制

- 十进制
  - 各位可能取值： 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - 示例：9472_{10} = 9472
- 二进制
  - 各位可能取值：0, 1
  - 示例：101011_2 = 0b101011
- 十六进制
  - 各位可能取值：0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - 示例：$2A5D_{16}$ = 0x2A5D

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

不同数制的示例

- 示例

9472_{10} = 9000 + 400 + 70 + 2

9x1000 + 4x100 + 7x10 + 2x1

$9 \times 10^3 + 4 \times 10^2 + 7 \times 10^1 + 2 \times 10^0$

9472_{10} = $2 \times 16^3 + 5 \times 16^2 + 0 \times 16^1 + 0 \times 16^0$

= **2500_{16}**

0xA15 = **0b 1010 0001 0101**

进制转换1/3

- 十进制转二进制
  - 用十进制数除以2，得到的余数就是相应二进制的最低位
  - 将得到的商继续除以2，得到的余数就是次低位
  - 重复上述过程直至商为0
- 示例：27的二进制计算过程
  - $27_{10} = 11011_2$

步骤	被除数	商	余数	备注
1	27	13	1	最低位
2	13	6	1	
3	6	3	0	
4	3	1	1	
5	1	0	1	最高位

进制转换2/3

- 十六进制与二进制
  - 两者之间转换非常简单，基本方法是：每4位二进制对应1位十六进制
  - 示例：110101101011011_2与$6C5B_{16}$之间的转换

二进制	110	1100	0101	1011
十六进制	6	C	5	B

手工转换时，注意从低位开始转换，以防止高位不足4位导致出错。

进制转换3/3

- 十进制转十六进制
  - 先将十进制数转换为二进制数
  - 再二进制数转换为十六进制数

目录

12

字节、字

- 字节（byte）：由8个二进制位构成的一个元组，是目前计算机数据单位
  - 1个byte的表示范围是从0x00至0xFF，即总共是256个数字
- 字（word）：一个字包含的二进制位数因计算机不同而不同
  - 不同CPU计算能力不同，因此其字的大小也不同
  - 以MIPS为例，CPU能计算的数据大小为32位，故其字长为32位
  - 大量当代计算机，如PC、服务器甚至手机，CPU字长已经发展到64位了
  - 在一些专用领域，CPU的字长甚至达到128位
  - 嵌入式领域中的某些CPU字长可能只有16位甚至8位

最高/最低有效位、最高/最低有效字节

- MSB（Most Significant Bit）：最高有效位，位于最左位
- LSB（Low Significant Bit）：最低有效位，位于最右边位
- MSB（Most Significant Byte）：最高有效字节，位于数据的最左边的字节
- LSB（Least Significant Byte）：最低有效自己，位于数据的最右边的字节

01011011		BCDEF789	
最高位	最低位	最高字节	最低字节

目录

二进制加法

- 二进制加法与十进制加法原理类似
  - 二进制每位数字非0即1，因此两个数字位相加最大值为2，即10_2
  - 显然，此时就应该进位了
- 示例：1011_2与0011_2的计算过程

	b3	b2	b1	b0
	1	0	1	1
	0	0	1	1
+			1	
	1	1	1	0

1)b0-b3位计算(产生进位输出)

	b4	b3	b2	b1	b0
		1	1	1	1
		0	0	0	1
+	1	1	1	1	
	1	0	0	0	0

2)b4位(进位)

溢出

- 人：在上例中，计算结果为10000_2是正常的
  - 人在计算中，通常不考虑位数限制，可以灵活的添加高位

- 计算机：由于受硬件限制，就必须考虑计算结果的位数
  - 假设CPU的字长为4位，则b4位是不存在的，故计算结果就是0000_2，即0。
  - 这种情况就是溢出。产生溢出，意味着计算结果出现错误了
    - CPU在溢出发生时，会通过称为**异常**的机制来报告这个错误

	b4	b3	b2	b1	b0
		1	1	1	1
		0	0	0	1
+	1	1	1	1	
	1	0	0	0	0

溢出～Overflow；异常～Exception

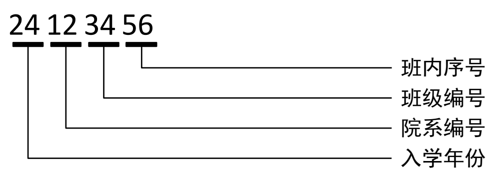
目录

- 常见进制及其转换
- 常见术语
- 二进制加法
- **整数的二进制表示方法**
- 浮点数的二进制表示方法
- 补码的几种常见运算

18

一切皆可用数字表示

- 误区：很多时候，我们认为数字的含义只能是数值
- 示例1：24123456
  - 解读1：如果在银行账户中，那就是存款
  - 解读2：如果在学籍系统中，那就是学号



- 示例2：0和1
  - 0：可以对应逻辑真（F，false）
  - 1：可以对应逻辑真（T，true）

一切皆可用数字表示

- 启示1：
  - 一个数字首先是一个编码，其含义必须结合上下文来解读
- 启示2：
  - 一个数字可以作为整体解读，也可以分为若干部分解读

- 对于一个数字来说，其包含2部分内容
  - 编码：就是数字本身，或者说是一种记号。例如007
  - 语义：编码所代表的概念的含义，是对编码的解释
    - 例如007可以代表邦德，也可以是房间编号

编码本身没有任何意义
编码只有被赋予语义后才有意义

数的编码空间

- 示例：3位十进制
  - 编码空间：000，001，…，999
  - 编码个数：10^3，即1000
- 编码空间：有效编码的集合
- 空间大小：有效编码的总数
  - 对于一个n位B进制数来说，其编码空间大小为B^n
  - 例如
    - 3位二进制，其编码空间大小为2^3=8
    - 3位十六进制，其编码空间为16^3=4096

编码位数

- 对于B进制来说，确定某个编码方案的编码位数
  - 1）首先需要确定需要编码的对象数量N
  - 2）然后根据N计算位数log_B^N（注意：计算结果应向上取整）
- 示例：编码26个英文字母，需要几位二进制编码？
  - 1）需编码的对象总共为52（26个大写字母，26个小写字母）
  - 2）计算$log_2^{52} \approx 5.7$。为此至少需要6位二进制

编码与编码对象之间的对应关系是由人来确定的
例如：A可以用000000_2对应，也可以用111000_2对应

二进制无符号数

- 二进制无符号数的编码方案
  - 没有负数；全是自然数

编码空间 ｛ ｝ 整数空间

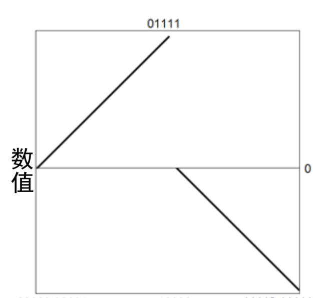
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 0_{10}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 1_{10}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$
... ...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = 2{,}147{,}483{,}645_{10}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = 2{,}147{,}483{,}646_{10}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 2{,}147{,}483{,}647_{10}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 2{,}147{,}483{,}648_{10}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 2{,}147{,}483{,}649_{10}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2{,}147{,}483{,}650_{10}$
... ...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = 4{,}294{,}967{,}293_{10}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = 4{,}294{,}967{,}294_{10}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 4{,}294{,}967{,}295_{10}$

无符号～unsigned

二进制符号数

- 原码方案
  - 用最高位表示符号位（0～正，1～负）
  - 其余位代表绝对值（也就是无符号数）
- 示例：4位原码编码方案
- 原码方案的缺陷
  - 存在2个零，即0000_2和1000_2
    - 有效编码被浪费了
  - 编码数值不连续
    - 希望数值随编码增大而增大
  - 无法直接使用前述的加法运算

$0000_2 = 0_{10}$
$0001_2 = 1_{10}$
$0010_2 = 2_{10}$
$0011_2 = 3_{10}$
$0100_2 = 4_{10}$
$0101_2 = 5_{10}$
$0110_2 = 6_{10}$
$0111_2 = 7_{10}$
$1000_2 = -0_{10}$
$1001_2 = -1_{10}$
$1010_2 = -2_{10}$
$1011_2 = -3_{10}$
$1100_2 = -4_{10}$
$1101_2 = -5_{10}$
$1110_2 = -6_{10}$
$1111_2 = -7_{10}$

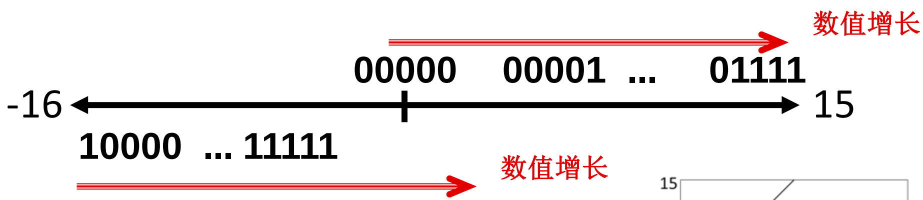


5位原码方案

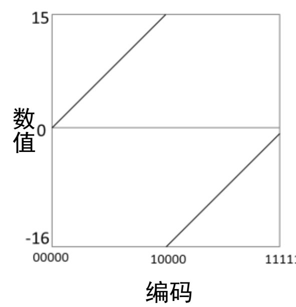
符号～signed

二进制符号数

- 二进制补码方案（以5位二进制补码为例）
  - 最高位代表符号位：0～正，1～负
  - 10000代表最小的负数（-16），11111代表最大的负数（-1）



- 特点：
  - 只有1个零
  - 符号确定后，数值随编码增长而增长



5位补码方案

32位二进制补码

- 表示范围：-2,147,483,648至2,147,483,647

符号位

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{10}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{10}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{10}$

... ...

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{10}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{10}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{10}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{10}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{10}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{10}$

... ...

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{10}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{10}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{10}$

二进制补码的一般性表示

- 对于一个$x_{N-1}x_{N-2}\ldots x_1x_0$的$N$位二进制补码数，其值A为：

$$A = (x_{N-1} \times -2^{N-1}) + (x_{N-2} \times 2^{N-2}) + \cdots + (x_1 \times 2^1) + (x_0 \times 2^0)$$
$$= x_{N-1} \times -2^{N-1} + \sum_{i=0}^{n-2} x_i \times 2^i$$

二进制补码小结

- 现代计算机普遍采用
- 使用最高位作为符号位
  - 0为正，1为负
- 正数区间与负数区间大致相同
  - 负数区间比正数区间多一个数
- 相反数计算方法：各位取反，然后加1
  - 示例：已知+7=0111_2，则-7=1000_2+1_2=1001_2

课堂练习

- 假设CPU字长为4位。以下哪个范围可以用二进制补码表示？

a) -15 至 +15　空间大小为31，需要至少5位

b) 0 至+15　空间大小为16（合理），但没有负数区间

c) -8 至 +7　正负区间都有，负区间多1个，空间大小16

d) -16 至 +15　正负区间都有，负区间多1个，但空间大小为32

目录

浮点数概述

- 计算机除了处理整数外，很多时候也需要处理浮点数
  - 例如圆周率3.1415926，就无法用前面讲的二进制整数编码方案表示
  - 在工程计算中，就大量涉及浮点数
- 浮点数具有表示范围和精度都较高的特点
  - 它解决了整数和小数位长度固定的限制
  - 允许表示一个很大的数或者很想的数

WilliamM.Kahan（威廉•凯亨），1933年6月
领导开发了Intel的8087浮点协处理器
制定了IEEE754和854标准
浮点数标准之父；1989年图灵奖获得者
For his fundamental contributions to numerical analysis. One of the foremost experts on floating-point computations. Kahan has dedicated himself to "making the world safe for numerical computations"!

浮点数格式

- 科学计数法回顾：5230可以表示5.23×10^3，包含3部分
  - 尾数（mantissa，M）：5.23
  - 基数（base，B）：10
  - 指数（exponent，E）：3
- 浮点数的方法与科学记数法非常相似，一个浮点数可以表示为：

$$\pm \mathrm{M} \times B^E$$

  - 4部分信息：符号（sign，S）、尾数、基数、指数

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	指数								尾数																						

  - 符号：1位，0～正，1～负
  - 指数：8位，用于表示范围
  - 尾数：23位，用于表示精度
  - 基数：由于基数固定为2，因此就被省略了

指数和尾数，增加任何一方位数都会减少另一方位数

目前方案是在精度与表示范围之间反复权衡后的结果

浮点数格式（优化前）

- 示例：用二进制浮点数格式表示5.23×10^3
  - $5.23\times10^3=5230_{10}=1010001101110=1.01000110111\times2^{12}$
  - 符号为0，指数为12（1100_2），尾数为1.01000110111

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	指数								尾数																						
0	0	0	0	0	1	1	0	0	1	0	1	0	0	0	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0

浮点数格式（优化后）

- 科学记数法通常不会让0出现在尾数的第1位（即小数点左边那位）。类似，浮点数表示方法同样不会让0出现在尾数的第1位
  - 由于尾数的第1位只能为1，那么就没有必要再占用实际存储位
  - 这使尾数多了1位有效存储位
  - 注意：在计算过程中，需要将该位自动补回
- 格式优化：5.23×10^3的尾数1.01000110111，优化为01000110111

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	指数								尾数																						
0	0	0	0	0	1	1	0	0	1	0	1	0	0	0	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0

优化前的浮点数格式

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	指数								尾数																						
0	0	0	0	0	1	1	0	0	0	1	0	0	0	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0

优化后的浮点数格式

表示绝对值小于0的浮点数

- 绝对值小于0的浮点数，其指数是负的
  - 示例：0.5和0.25，其对应的就是$1.0x2^{-1}$与$1.0x2^{-2}$
- 思路：指数采用二进制补码
- 缺点：负指数的浮点数虽较小，但其二进制却似乎是个较大的数
  - 示例：0.5与2.0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	指数								尾数																						
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0.5（1.0×2^{-1}）的浮点数格式为：0x7F800000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	指数								尾数																						
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

2.0（$1.0\times2^{+1}$）的浮点数格式：0x00800000

浮点数：0.5<2.0
二进制：0x7F800000 > 0x00800000

表示绝对值小于0的浮点数：偏阶计数法

- 目标： 00000000_2对应最小的负指数，而11111111_2对应最大的正指数
- 思路：编码用的指数=真实指数+127；该方法被称为偏阶记数法
- 示例：0.5与2.0
  - 0.5的偏阶编码指数：$11111111_2+01111111_2=01111110_2$
  - 2.0的偏阶编码指数：$00000001_2+01111111_2=10000000_2$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	指数								尾数																						
0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0.5（1.0×2^{-1}）的偏阶浮点数格式为：0x3F000000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	指数								尾数																						
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

2.0（$1.0\times2^{+1}$）的偏阶浮点数格式：0x40000000

IEEE754浮点标准就采用了偏阶记数法

单精度浮点表示范围

- 单精度浮点：占据32位
  - 最大正数：$+3.402824 \times 10^{38}$
    - 符号位：0，正数
    - 指数：127（编码指数为254）
    - 尾数：11111111111111111111111111111111（23个1）

最大正数计算方法

$$-1^{0} \times 1.M \times 2^{E} = +1.11111111111111111111111 \times 2^{127} \approx 3.402823 \times 10^{38}$$

  - 最小正数：$+1.175494 \times 10^{-38}$
  - 最大负数：$-1.175494 \times 10^{-38}$
  - 最小负数：-3.402824×10^{38}
- 单精度表示区间
  - 正数区间：$+1.175494 \times 10^{-38}$至$+3.402824 \times 10^{38}$
  - 负数区间：-3.402824×10^{38}至$-1.175494 \times 10^{-38}$

双精度浮点表示范围

- 为了提供更大的取值范围和更高的精度，IEEE754标准还定义了双精度浮点
- 双精度浮点：占据64位

格式	总位数	符号位	指数位	尾数位
单精度浮点	32	1	8	23
双精度浮点	64	1	11	52

- 双精度表示范围
  - 正数区间：$+2.22507385850720 \times 10^{-308}$至$+1.79769313486232 \times 10^{308}$
  - 负数区间：$-1.79769313486232 \times 10^{308}$至$-2.22507385850720 \times 10^{-308}$

特殊情况

- IEEE754标准用指数和尾数的某些特殊编码表示一些特殊情况
  - 例如如：0、±∞、非法数（或者不存在的数，如$\sqrt{-1}$）

表示的数	单精度浮点数			双精度浮点数		
	符号	指数	尾数	符号	指数	尾数
0	X	0	0	X	0	0
+∞	0	255	0	0	2047	0
-∞	1	255	0	1	2047	0
NaN（非数）	X	255	非0	X	2047	非0
正的浮点数	0	1~254	任意	0	1~2046	任意
负的浮点数	1	1~254	任意	1	1~2046	任意

注意：IEEE754编码方案中有2个0

上溢和下溢

- 以单精度浮点数为例，其区间为

$$[-3.402824\times10^{38}，\ -1.175494\times10^{-38}]，[+1.175494\times10^{-38}，\ +3.402824\times10^{38}]$$

- 当结果A：A>3.402824×10^{38} 或 A<-3.402824×10^{38}
  - 这种情况被称为上溢
  - A被向上舍入为±∞
- 当结果A：-1.175494×10^{-38}<A<0 或 0<A<$+1.175494\times10^{-38}$
  - 这种情况被称为下溢
  - A被向下舍入为0

目录

负数的二进制补码的计算方法

- 相反数：符号相反但绝对值相同的一对整数
  - 假设a与b是2个整数，如果a+b=0，则a与b是相反数
- 显然a与-a就是相反数
  - 示例：+7与-7是相反数，-100与+100是相反数
- 相反数计算方法
  - 方法：X的二进制补码表示为$x_{N-1}\ldots x_1 x_0$，则-X为X的各位取反然后加1
  - 原理：
    - 由于$x_{N-1}\ldots x_1 x_0$与$\bar{x}_{N-1}\ldots \bar{x}_1 \bar{x}_0$各对应位的2位为0、1互斥，因此相加后$N$位计算结果必然是全1，即：

$$x_{N-1}\ldots x_1 x_0 + \bar{x}_{N-1}\ldots \bar{x}_1 \bar{x}_0 = 1\ldots 11$$

    - 二进制补码中的全1代表-1，故：$x + \bar{x} = -1$
    - 将上式变形，得：$-x = \bar{x} + 1$

负数的二进制补码的计算方法

- 示例1：用8位二进制补码表示-14
  - 1）14的二进制补码表示为00001110_2
  - 2）将该数各位取反可得　11110001_2
  - 3）再计算　$11110001_2+1 = 11110010_2$
  - -14的8位二进制补码为　11110010_2

- 示例2：二进制补码为11110010_2，请给出其10进制数值
  - 分析：最高位为1说明这是一个负数，那么计算出其对应的相反数即可
  - 1）将11110010_2各位取反可得00001101_2
  - 2）再计算$00001101_2+1=00001110_2$
  - 00001110_2的10进制为$8+4+2=14$

减法

- 思路：为了计算*x*-*y*，可以通过数学变形改为计算*x*+(-*y*)
- 示例：计算28-14（假设字长为8位）
  - 1）28的8位二进制补码为：00011100_2
  - 2）-14的二进制补码为：
    - 14的二进制补码为：00001110_2
    - 各位取反为：11110001_2
    - 加1，得：11110010_2
  - 3）计算：　$00011100_2+11110010_2=00001110_2=14$

$$
\begin{array}{ccccccccc}
 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
+ & 1 & 1 & 1 & & & & & \\
\hline
\boxed{1} & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0
\end{array}
$$

补码体系的优势：只需要设计加法运算的硬件，减法可以转换为加法运算，故不需为减法设计相应硬件

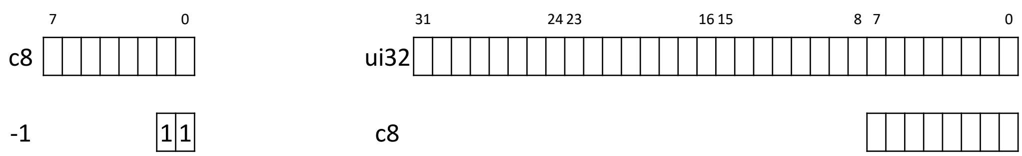
位扩展

- 程序中存在不同类型间的变量赋值

```
1 char           c8  ;
2 int            i32  ;
3 unsigned int   ui32  ;
4
5 c8       = -1  ;
6 i32      = c8  ;
7 ui32     = (unsigned char)c8  ;
```

8位符号数
32位符号数
32位无符号数

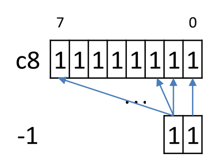
- 不同类型的变量往往位数不同，这就涉及扩展问题
  - Line5：-1最少用2位二进制补码表示，即11_2，但c8需要8位
  - Line7：ui32是32位，c8是8位



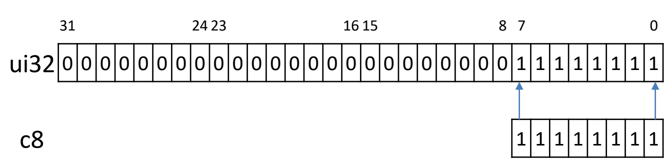
位扩展

- 符号扩展：对于Line5，由于c8是符号数，因此需要将-1的符号位复制到c8的高位部分。
  - 计算完成后，c8=11111111_2，即8位补码的-1

```
1 char           c8  ;
2 int            i32  ;
3 unsigned int   ui32  ;
4
5 c8       = -1  ;
6 i32      = c8  ;
7 ui32     = (unsigned char)c8  ;
```



- 无符号扩展：对于Line7来说，①强制转换要求将c8视为无符号数，②为确保扩展后的结果正确，ui32的高位部分应该都为0。
  - 计算完成后，ui32=$00000000000000000000000011111111_2$



比较

- C语言定义了多个比较操作，其运算结果是逻辑的真假值
  - =、≠、>、<、≥、≤
- < 和=：必须实现
- ≠、>、≥ 、≤：以< 和=为基础，结合NOT逻辑运算，即可实现

原运算	$A \neq B$	$A > B$	$A \geq B$	$A \leq B$
等价运算	$\overline{A = B}$	$B < A$	$\overline{A < B}$	$\overline{B < A}$

比较：小于运算(符号数)1/3

- 如果A和B是符号数：A<B小于运算可以转换为A-B<0
  - 1）首先执行减法，即C←A-B
  - 2）判断C的符号位：
    - 如果为1，则C<0，即A<B为真
    - 如果为0，则C≥0，即A<B为假
- 示例：A、B为4位二进制补码的0000_2与1000_2，计算“A<B”的真假值
  - 理论分析：0000_2和1000_2分别为0和-8，显然0大于-8，故“A<B”为假
  - 实际计算：$A - B = A + \bar{B} + 1 = 0000 + 0111 + 0001 = 1000$
  - 计算的结论：结果的符号位b3为1，即A-B<0，意味着“A<B”为真

比较：小于运算(符号数)2/3

- 正确的理论计算
  - 从数学上可知，0减-8的结果应该是+8
  - 如果用补码表示+8，则至少需要5位二进制补码，即01000_2
    - 由于b4的存在，故b3的“1”含义是数值，而不是符号
- 错误的实际计算
  - 前述计算采用4位二进制补码计算，结果为1000_2
  - 由于b4的缺失，故b3的“1”被错误的解读为符号
- 错误的原因：由于位数不足，导致计算结果+8超出了4位补码的表示范围，发生了溢出

比较：小于运算(符号数)3/3

- 解决方案：扩展符号位后再计算
  - 2个N位二进制补码的操作数均扩展1位符号位
  - 进行N+1位二进制补码的减法计算
- 示例：A、B为4位二进制补码的0000_2与1000_2，计算“A<B”的真假值
  - 1）将A、B符号位扩展1位，分别为00000_2和11000_2
  - 2）计算：$A-B=A+\bar{B}+1=00000+00111+00001=01000$
  - 3）根据符号位b4为0可知$A-B\geq 0$，故“A<B”为假
  - 与理论分析一致！

比较：小于运算(无符号数)

- 解决方案：0扩展后再按符号数计算
  - 1）如果比较的两个数均为无符号数，将其视为符号数
  - 2）为了能够用符号数正确表达其原值，需要将两个数分别扩展1位符号位，且符号位为0
  - 3）之后，就可以采用前述的方案了

相等

- 方案1：采用XOR运算，然后判断结果是否全0
  - 1）执行XOR运算，即C←A⊕B
  - 2）将C的各位OR起来，得到1位结果
    - 如果为0，即“A=B”为真
    - 如果为1，即“A=B”为假
- 方案2：采用减法运算，然后判断结果是否全0
  - 1）执行减法运算，即C←A-B
  - 2）将C的各位OR起来，得到1位结果
    - 如果为0，即“A=B”为真
    - 如果为1，即“A=B”为假

乘法：无符号数

- 二进制乘法的基本计算过程与十进制乘法基本类似
- 示例：计算$0101_2 \times 1011_2$
  - 理论分析结果：$0101_2 \times 1011_2 = 5 \times 11 = 55$
  - 实际计算结果：$0110111_2 = 55$

```
          0 1 0 1
        × 1 0 1 1
        ---------
          0 1 0 1
        0 1 0 1
      0 0 0 0
  + 0 1 0 1
  ---------------
    0 1 1 0 1 1 1
```

- 基本原理：循环累加左移后的被乘数
  - 假设被乘数A为N位，乘数B为M位
  - 结果C为N+M-1位

```
C = 0
for i = 0 to M-1
  if B[i] then
    C = C + A<<i
```

乘法：符号数

- 基本原理：借助无符号数乘法
  - 假设A、B均为32位符号数，C为64位计算结果
  - 1）得到A、B的绝对值A′和B′
    - 由于A和B均为符号数，故A′和B′均为31位
  - 2）借助无符号乘法，计算乘法结果的绝对值C′＝A′×B′
    - 从计算的角度，C′的位数为61位（31+31-1=61）
  - 3）根据A和B的符号位决定结果的正或负
    - 同号为正，异号为负
  - 4）根据结果计算C
    - 如果结果为正：$C = 000||C'$
    - 如果结果为负：$C = \overline{000||C'} + 1$

A[31]	B[31]	结果
0	0	正
0	1	负
1	0	负
1	1	正

除法的复杂性

- 运算结果种类
  - 乘法：运算结果只有一个，即积
  - 除法：有2个运算结果，即商和余数
- 结果正负性质
  - 数学公式：被除数＝除数×商＋余数
  - 示例：-7÷2。如果仅按数学公式，可以有2种结果
    - 结果1：-7＝2×(-3)＋(-1)
    - 结果2：-7＝2×(-4)＋1
- 中间计算过程
  - 乘法：以加法为核心
  - 除法：以减法为核心。减法比加法要复杂些

除法：无符号数1/3

- 示例：计算1001010_2除以1000_2
  - 理论分析：1001010_2为74，1000_2为8，故74÷8的商为9，余数为2
  - 计算过程：与十进制除法类似，核心是“试商”
- 试商：在二进制除法中，商的各位非0即1，“试商”更为简单
  - 计算商的每位时，仅需做一次减法
  - 然后根据减法结果的符号位判断是否够减

```
位序              6 5 4 3 2 1 0
                        1 0 0 1   商
除数  1 0 0 0  | 1 0 0 1 0 1 0   被除数
               - 1 0 0 0
                        1 0 1 0
                      - 1 0 0 0
                        0 0 1 0   余数
```

注意
在示例的计算过程中，b2、b1的试商过程被省略了。
计算机是很“傻”的，因此在实际计算中必须给出完整的计算过程。

除法：无符号数2/3

- 中间余数：假设除数为N位，则中间余数为N+1位
- 计算方法：每一次迭代计算商的一位
  - 每次从被除数中取出1位，构成中间余数的末位
  - 试商：用中间余数减除数，根据减法结果决定商的各位取值

位序	6 5 4 3 2 1 0	
	0 0 0 1 0 0 1	商
除数 1 0 0 0	0 0 0 0 1 0 0 1 0 1 0	被除数。初始中间余数为00001_2。
	- 0 1 0 0 0	
	0 0 0 1 0	差值<0：商位6为0；中间余数=中间余数左移1位 ‖ 被除数位5
	- 0 1 0 0 0	
	0 0 1 0 0	差值<0：商位5为0；中间余数=中间余数左移1位 ‖ 被除数位4
	- 0 1 0 0 0	
	0 1 0 0 1	差值<0：商位4为0；中间余数=中间余数左移1位 ‖ 被除数位3
	- 0 1 0 0 0	
	0 0 0 1 0	差值≥0：商位3为1；中间余数=差值左移1位 ‖ 被除数位2
	- 0 1 0 0 0	
	0 0 1 0 1	差值<0：商位2为0；中间余数=中间余数左移1位 ‖ 被除数位1
	- 0 1 0 0 0	
	0 1 0 1 0	差值<0：商位1为0；中间余数=中间余数左移1位 ‖ 被除数位0
	- 0 1 0 0 0	
	0 0 0 1 0	差值≥0：商位0为1；计算到最后一位，停止计算；中间余数=差值

除法：符号数3/3

- 符号数除法的基本思路
  - 1）获得被除数、除数的绝对值
  - 2）执行两者绝对值的除法，得到商和余数
  - 3）商的符号：由被除数与除数的符号决定，即同号为正、异号为负
  - 4）余数的符号：与被除数的符号相同