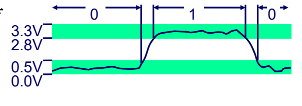
第2章 运算方法和运算器a

2.1 数据与文字的表示方法

数据编码与表示

- 计算机中的数据
  - 整数、浮点数、字符（串）、逻辑值
  - 需要编码进行表示
- 编码原则
  - 使用少量简单的基本符号
  - 一定的规则
  - 表示大量复杂的信息
- 二进制码0、1
  - 符号个数最少，物理上容易实现
  - 与二值逻辑的“真” “假” 两个值对应
  - 用二进制码表示数值数据运算规则简单



无符号数和有符号数

- 无符号数指的是不带符号位的数，例如：
  1个16位二进制数，表示范围为0～65535
  在C语言中，用unsigned short int类型进行声明
- 有符号数指的是带符号位的数，最左边的位用作符号位，用“0”表示正,“1”表示负，例如
  有符号数+0001001 00000000，表示为0x0900
  有符号数-0001001 00000000，表示为0x8900
  在C语言中，用short int类型进行声明

注：在32位机上，int类型通常为32位

计算机中常用的数据表示格式

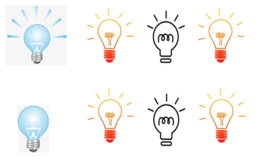
- 定点格式：数值范围有限，处理简单
  - 机器中所有数据的小数点位置固定不变
  - 不使用记号“.”来表示小数点
  - 定点数表示成纯小数或纯整数
- 浮点格式：数值范围很大，处理过程复杂
- 十进制数格式
  - 非压缩BCD
  - 压缩BCD

定点数的表示方法

- 纯小数 $\boxed{x_0}\boxed{x_1\ x_2\ \ldots\ x_{n-1}\ x_n}$
  - x_0表示数的符号，数值0和1分别表示正号和负号，其余为代表数的量值
  - 小数点位于x_0和x_1之间.
  - 表示范围为$0 \le |x| \le 1-2^{-n}$
- 纯整数 $\boxed{x_0}\boxed{x_1\ x_2\ \ldots\ x_{n-1}\ x_n}$
  - x_0表示数的符号，数值0和1分别表示正号和负号，其余为代表数的量值
  - 小数点位于最低位x_n的右边
  - 表示范围为$0 \le |x| \le 2^n-1$

2/20/2023

真值与机器数



$+5$ 0 101

-5 1 101

真值是机器数代表的实际的值

机器数是真值在机器中的表示

数的机器码表示

- 为了方便数的运算操作，在计算机中通常将数的符号位和数值位一起编码
- 小数点：隐含存储
  - 定点数事先约定，浮点数按规则浮动
- 为了区别带有符号表示的数和机器中把符号"数字化"的数，通常将前者称为真值，后者称为机器数或机器码
- 常用的（有符号数）机器码包括
  - 原码、补码、反码、移码

原码表示法（1）

- 定点小数
  - 定义
  - $[x]_{原} = \begin{cases} x & 1 > x \geq 0 \\ 1-x=1+|x| & 0 \geq x > -1 \end{cases}$
  - 举例

$x=+0.1001 \Rightarrow [x]_{原}=0.1001$

$\underbrace{x=-0.1001}_{真值} \Rightarrow \underbrace{[x]_{原}=1.1001}_{机器码}$

原码表示法（2）

- 定点整数 | x_0 | $x_1\ x_2 \ldots x_{n-1}\ x_n$ |
  - 定义
  - $[x]_{原} = \begin{cases} x & 2^n > x \geq 0 \\ 2^n - x = 2^n + |x| & 0 \geq x > -2^n \end{cases}$
  - 举例

$x=+0111001 \Rightarrow [x]_{原}=00111001$

$x=-0111001 \Rightarrow [x]_{原}=10111001$

原码表示法（3）

- 0的原码
  - $[+0]_{原}=00000000$
  - $[-0]_{原}=10000000$
- 正数的原码符号位为0，数值位不变；负数的原码符号位为1，数值位不变

原码的特点

- 简单、直观：采用原码表示法简单明了，易于和真值转换
- 原码用做加法是会出现以下问题

数1	数2	实际操作	结果符号
正	正	加	正（正）
正	负	减	可正可负（负）
负	正	减	可正可负（负）
负	负	加	负（正）

- 基于原码的减法运算
  - 1＋（－1）＝0000 0001＋1000 0001
    ＝1000 0010
    ＝－2 ✕

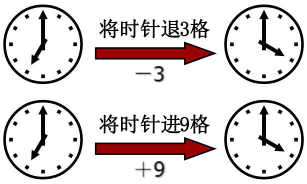
能否只做加法？

如果能找到一个与负数等价的正数，来代替这个负数，就可使减法变为加法

2

补码表示法

将时钟从7点钟调到4点种，有两种调法：



对于时钟来说，—3运算和+9运算是等价的，就是说9是(—3)以12为模的补码

−3=+9 （mod 12）

减法可转换为加法！

补码表示法

- 补数：一个负数加上“模”即得该负数的补数
- 一个正数和一个负数互为补数时，它们的绝对值之和即为模数

1011 -> 0000 ?
（模16）

$$\begin{array}{r} 1011 \\ -1011 \\ \hline 0000 \end{array} \qquad \begin{array}{r} 1011 \\ +0101 \\ \hline 10000 \end{array}$$

−1011可以用+0101代替

即：−1011≡+0101（mod 2^4）

补码表示法（1）

- 定点小数
  - 定义

$$[x]_{补}=\begin{cases} x & 1>x\ge 0 \\ 2+x=2-|x| & 0\ge x\ge -1 \end{cases} \quad (\mathrm{mod}\ 2)$$

  - 举例

$x=+0.1011$ ➡ $[x]_{补}=0.1011$

$x=-0.1011$ ➡

$[x]_{补}=10+x=10-0.1011=1.0101$

补码表示法（2）

- 定点整数 | x_0 | $x_1\ x_2\ \ldots\ x_{n-1}\ x_n$ |
  - 定义

$$[x]_{补}=\begin{cases} x & 2^n>x\ge 0 \\ 2^{n+1}+x=2^{n+1}-|x| & 0\ge x\ge -2^n \end{cases} \quad (\mathrm{mod}\ 2^{n+1})$$

  - 举例

$x=+0111001$ ➡ $[x]_{补}=00111001$

$x=-0111001$ ➡

$[x]_{补}=100000000+x=100000000-0111001=11000111$

补码表示法（3）

- 0的补码

  $[+0]_{补}=[-0]_{补}=00000000$（mod 2）

- 采用补码进行减法运算就比原码方便，减法运算可变成加法运算
- 基于补码的减法运算
  - 1+（−1）＝0000 0001 ＋ 1111 1111

    ＝0000 0000

    ＝0 ✓

反码表示法（1）

- 反码就是二进制数的各位数码0变为1，1变为0。

即：若$x_i=1$，则反码$x_i=\underline{0}$

若$x_i=0$，则反码$x_i=\underline{1}$

反码表示法（2）

- 定点小数 | x_0 | $x_1\ x_2\ \ldots\ x_{n-1}\ x_n$ |
  - 定义

$$[x]_{反}=\begin{cases} x & 1>x\geq 0 \\ (2-2^{-n})+x & 0\geq x>-1 \end{cases}$$

  - 举例

$x=+0.1011 \rightarrow [x]_{反}=0.1011$

$x=-0.1011 \rightarrow [x]_{反}=1.0100$

反码表示法（3）

- 定点小数
  - 负数求补码和反码公式的比较

$$[x]_{反}=(2-2^{-n})+x$$

$$[x]_{补}=2+x=[x]_{反}+2^{-n}$$

  - 结论：对一个负数求补码，其方法是符号位置为1，其余各位求反，然后在最末位上加1。也就是说，负数的补码等于其反码加1。
- 0的反码
  - $[+0]_{反}=00000000$
  - $[-0]_{反}=11111111$

反码表示法（4）

- 定点整数 | x_0 | $x_1\ x_2\ \ldots\ x_{n-1}\ x_n$ |
  - 定义

$$[x]_{反}=\begin{cases} x & 2^n>x\geq 0 \\ (2^{n+1}-1)+x & 0\geq x>-2^n \end{cases}$$

  - 举例

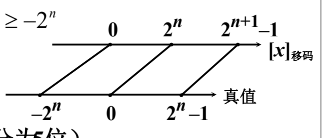
$x=+0111001 \rightarrow [x]_{反}=00111001$

$x=-0111001 \rightarrow [x]_{反}=11000110$

移码表示法

- 移码是在真值X上加一个常数偏移值bias，通常为2^n
  - 移码通常用于表示浮点数的阶码（指数），便于比较大小
- 假设阶码是个n位的整数，定点整数移码的定义为：

$$[x]_{移}=2^n+x \quad 2^n>x\geq -2^n$$



- 举例（假设阶码数字部分为5位）

$x=+10101 \rightarrow [x]_{移}=1,10101$

$x=-10101 \rightarrow [x]_{移}=2^5+x=2^5-10101=0,01011$

移码中用的逗号不是小数点

移码和补码尾数相同，符号为相反

8位无符号数和有符号数的表示

8 位二进制数（其中 1 位为符号位），用其分别代表无符号数、原码、补码和反码，对应的真值范围如下表

二进制代码	无符号数对应的真值	原码对应的真值	补码对应的真值	反码对应的真值
00000000	0	+0	±0	+0
00000001	1	+1	+1	+1
00000010	2	+2	+2	+2
⋮	⋮	⋮	⋮	⋮
01111111	127	+127	+127	+127
10000000	128	-0	-128	-127
10000001	129	-1	-127	-126
⋮	⋮	⋮	⋮	⋮
11111101	253	-125	-3	-2
11111110	254	-126	-2	-1
11111111	255	-127	-1	-0

原码、反码、补码和移码比较（1）

- 四种码制主要是解决有符号数在机器中的表示与运算问题
- 若X为正数，则$[X]_{原}=[X]_{反}=[X]_{补}=X$
- 最高位为符号位：
  - $[X]_{原}$、$[X]_{反}$、$[X]_{补}$用“0”表示正号，用“1”表示负号；
  - $[X]_{移}$用“1”表示正号，用“0”表示负号。

原码、反码、补码和移码比较（2）

- $[0]_{补}$、$[0]_{移}$有唯一编码，
  $[0]_{原}$、$[0]_{反}$ 有两种编码。
- 移码与补码的尾码相同，只是符号位相反。
- 补码、反码和移码的符号位在加减运算时可以当作为数值看待，但原码的符号位必须单独处理

数据的表示

表示
- 无符号数
- 有符号数（真值和机器数）
  - 原码：简单，直观
  - 补码：简化加减运算
  - 反码：原、补码转换的中间过渡
  - 移码：易于比较大小

例1

设机器字长16位，定点表示，尾数15位，数符1位，问：
(1)定点原码整数表示时，最大正数、最小负数各是多少？
(2)定点原码小数表示时，最大正数、最小负数各是多少？

解：

(1)定点原码整数表示

最大正数值$=(2^{15}-1)_{10}=(+32767)_{10}$

最小负数值$=-(2^{15}-1)_{10}=(-32767)_{10}$

(2)定点原码小数表示

最大正数值$=(1-2^{-15})_{10}=(+0.111...11)_2$

最小负数值$=-(1-2^{-15})_{10}=(-0.111..11)_2$

32位现代计算机中

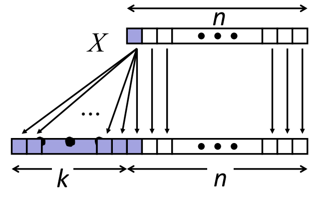
C语言	编译	机器数
int x = 100;		0x00000064
int y = -100		0xFFFFFF9C 补码
unsigned short int x=65534;	编译	0xFFFE

数据表示

C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

符号位扩展

- 将n位的有符号整数x转换为n+k位，并保持值不变




```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

例2

一个C语言程序在一台32位机器上运行。程序中定义了三个变量x、y、z，其中x和z是int型，y为short型。当x=127，y=-9时，执行赋值语句z=x+y后，x、y、z的值分别是

A．x=0000007FH，y=FFF9H，z=00000076H

B．x=0000007FH，y=FFF9H，z=FFFF0076H

C．x=0000007FH，y=FFF7H，z=FFFF0076H

D．x=0000007FH，y=FFF7H，z=00000076H

移位操作

- 机器中的移位操作是一种运算
  - 左移1位 绝对值扩大，即：×2
  - 右移1位 绝对值缩小，即：÷2
- 在计算机中，移位与加减配合，能够实现乘除运算
- 算术移位的法则（有符号数的移位法则）
  - 符号位保持不动
  - 正数：原码、补码、反码均补0
  - 负数
    - 原码：补0
    - 补码：左移补0；右移补1
    - 反码：补1

移位操作举例（1）

- 例：机器字长为8位，x = – 50，对其进行左移位和右移位，结果如下：

$[x]_{\text{原}}$= 10110010

左移：$[x]_{\text{原}}$= 11100100　　– 100

右移：$[x]_{\text{原}}$= 10011001　　– 25

$[x]_{\text{补}}$= 11001110

左移：$[x]_{\text{补}}$= 10011100

　　$[x]_{\text{原}}$= 11100100　　– 100

右移：$[x]_{\text{补}}$= 11100111

　　$[x]_{\text{原}}$= 10011001　　– 25

移位操作举例（2）

$[x]_{\text{反}}$= 11001101

左移：$[x]_{\text{反}}$= 100110 11

　　$[x]_{\text{原}}$= 11100100　　– 100

右移：$[x]_{\text{反}}$= 11100110

　　$[x]_{\text{原}}$= 10011001　　– 25

例子

- 某字长为8位的计算机中，已知整型变量x、y的机器数分别为 $[x]_{\text{补}}$=1 1110100，$[y]_{\text{补}}$=1 0110000。若整型变量 z=2* x+y/2 ，则 z的机器数为

A. 1 100 0000

B. 0 010 0100

C. 1 010 1010

D. 溢出

浮点数的表示方法

- 定点数的局限性

小张的财富 —2786

（2字节定点整数 short类型即可表示）

马云的财富 307446894372

（4字节定点整数 int类型也不能表示）

（只能用8字节 long类型表示）

换一种货币：1人民币 ≈ 10^{10}津巴布韦币

（8字节long型也表示不了）

定点数可表示的数据范围有限，但我们不能无限制的增长数据的长度

如何在位数不变的情况下，增加数据表达范围？

浮点数的表示方法

- 科学计数法

普通计数法：

307446894372

科学计数法：

3.074✖ 10^{11}

+11 +3.074

1人民币 ≈ 10^{10}津巴布韦币

+21 +3.074

阶码

尾数

反映数值大小

反映精度

浮点数的表示方法（1）

- 任意一个十进制数N可以表示成

$$N = M \times 10^E$$

尾数

阶

2.13 x 10^{19}

小数点

基数

- 一个任意进制数N可以表示成

$$N = M \times R^E$$

其中，M称为浮点数的尾数，是一个纯小数；E是比例因子的指数，称为浮点数的指数，是一个整数；R为比例因子的基数，在二进制机器中通常规定R为2，8或16

- 数的小数点位置随比例因子的不同而在一定范围内可以自由浮动，这种表示法称为浮点表示法

浮点数的表示方法（2）

- 机器浮点数的组成
  - 尾数通常为纯小数，常用原码或补码表示。尾数的有效数字的位数决定了浮点数的表示精度
  - 指数为定点整数，称为阶码，常用移码或补码表示，阶码的位数决定了浮点数的表示范围

E_S	$E_1 E_2 \ldots E_m$	M_S	$M_1 M_2 \ldots M_n$
阶符	阶码	数符	尾数

若尾数和阶码均采用原码，非规格化表示方式，则：

最大正数 $(1 - 2^{-n}) \times 2^{2^m-1}$　　最大负数 $-2^{-n} \times 2^{-(2^m-1)}$

最小正数 $2^{-n} \times 2^{-(2^m-1)}$　　最小负数 $-(1 - 2^{-n}) \times 2^{2^m-1}$

浮点数的表示方法（3）

- 浮点数的规格化形式
  - 为了充分利用尾数的有效位数、提高运算精度
    - 1.01×2^3可以写成0.101×2^4、0.0101×2^5
  - 对任何一个浮点数其规格化形式是唯一的
  - 对于阶码基数为R的规格化浮点数，定义其尾数的绝对值范围：

$$1/R \le |M| < 1$$

  - 如果R=2，则：

$$1/2 \le |M| < 1$$

当尾数用原码表示时，尾数的最高位总等于1

因此，最小正数 $2^{-n} \times 2^{-(2^m-1)}$ 规格化形式为 $2^{-1} \times 2^{-(2^m-1)}$

最大负数 $-2^{-n} \times 2^{-(2^m-1)}$ 规格化形式为 $-2^{-1} \times 2^{-(2^m-1)}$

例2

某浮点数字长32位，其中阶码8位，以2为底，补码表示；尾数为纯小数，24位（含1位数符），补码表示。现有一浮点数为C18F1234，求它所表示的二进制真值是多少？

阶码（8位）	数符	尾数（23位）

解：展开C18F1234，得：

1100 0001 1000 1111 0001 0010 0011 0100

阶码原码：1011 1111，十进制：-63

尾数原码：1.111 0000 1110 1101 1100 1100

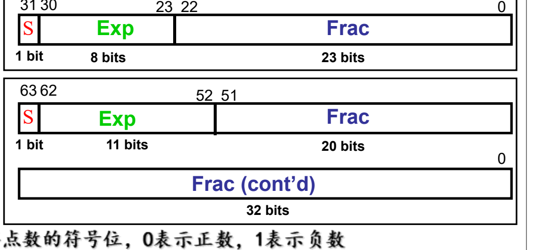
二进制真值:$-0.11100001110110111001100 \times 2^{-63}$

IEEE 754浮点数标准

$(-1)^S * M * 2^{Exp-Bias}$　　其中，$1 \le M < 2$ 或 $0 \le M < 1$

单精度
float x;

双精度
double y;



S：浮点数的符号位，0表示正数，1表示负数

Frac：小数字段，用原码表示，小数点放在尾数域的最前面。规格化数中：M=1+Frac，即：隐含最高位1

Exp：阶码，用移码来表示

Bias：偏移值 $\begin{cases} 127 & \text{（单精度）} \\ 1023 & \text{（双精度）} \end{cases}$

阶的移码表示

- IEEE 754中，浮点数的阶E采用移码表示
- 移码：规格化数，在真正的阶e上加一个规定的值
  - 对单精度浮点数：E = e + Bias = e + 127
  - 对双精度浮点数：E = e + Bias = e + 1023
- 最小的阶：00000001_2　　1_{10}
- 最大的阶：11111110_2　　254_{10}　（单精度）
- 例如:$1.0 * 2^{-1}$ 在机器中的表示如下：

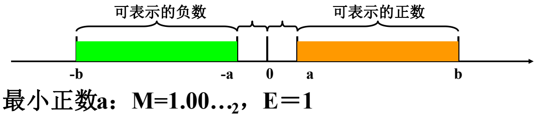
0	0111 1110	0000 0000 0000 0000 0000 000

真值计算：$(-1)^S * (1 + Frac) * 2^{(E - Bias)}$

小数字段Frac

- 尾数用原码表示
- 规格化数表示，隐含最高位1
  - M = 1 + Frac，1≤M<2
- 非规格化数表示
  - M = Frac，0≤M<1
- 单精度为：1 + 23 位，双精度为 1 + 52 位
- 含义：
  - 十进制：$1.6732 = (1x10^0) + (6x10^{-1}) + (7x10^{-2}) + (3x10^{-3}) + (2x10^{-4})$
  - 二进制：$1.1001 = (1x2^0) + (1x2^{-1}) + (0x2^{-2}) + (0x2^{-3}) + (1x2^{-4})$

规格化数表示的数值范围（单精度）



- 最小正数a：$M=1.00\ldots_2$，E＝1
  - $a = 1.0\ldots_2 * 2^{1-127} = 2^{-126}$
- 最大正数b：$M=1.1\ldots11_2$，E＝254
  - $b = 1.1 \ldots 11_2 * 2^{254-127}$
    $= 0.11\ldots11_2 * 2^{128}$
    $= \underbrace{(1-2^{-24})}_{24个1} * 2^{128}$

浮点数的二→十进制转换

若32位浮点数x在计算机中存储的数为$(41360000)_{16}$，求其十进制值

解：首先将十六进制数展开，得到二进制格式

十六进制	4	1	3	6	0	0	0	0
二进制	0100	0001	0011	0110	0000	0000	0000	0000

S　阶码E　尾数Frac

$x=(-1)^S \times 1.Frac \times 2^{E-127}$

$x=(-1)^0 \times (1.011011) \times 2^{10000010-01111111}$

$x=+(1.011011) \times 2^3=+1011.011$

$=11+0 \times 2^{-1}+1 \times 2^{-2}+1 \times 2^{-3}$

$= 11.375$

浮点数十→二进制转换（1）

- 把纯小数化为分数后，如果分母是2的整数次方，则转换结果是准确的；否则转换结果是近似的。
- 如：-0.75的二进制
  - $-0.75 = -3/4 \longrightarrow -11_2/100_2 = -0.11_2$
  - 规格化为：$-1.1_2 \times 2^{-1}$
  - $(-1)^1 \times (1 + .100\ 0000 \ldots 0000) \times 2^{-1}$

1	0111 1110	1000 0000 0000 0000 0000 000

浮点数十→二进制转换（2）

- 将小数化为分数，若分母不是2的整数倍，转换方法为：
  - 求出足够多的有效位
  - 根据精度要求截断多余的位。
  - 按标准要求给出符号位、阶和尾数。
- 如：求-2.15的二进制

0.15	0.30	0.60	0.20	0.40	0.80	…
× 2	× 2	× 2	× 2	× 2	× 2	
0 .30	0 .60	1 .20	0 .40	0 .80	1 .60	

⟶　- 10.00100110011001100110011001001

- 规格化　- 1.00010011001100110011001100 1001… × 2^1
- 阶　$1＋127＝128＝10000000_2$

1	10000000	0001 0011 0011 0011 0011 001

特殊的浮点数值

- IEEE浮点标准中，阶码0、阶码（2^8 - 1）或（2^{11} - 1）被保留，用作特殊用途

特殊值	阶	小数（尾数）
+/- 0	0000…0000	0
非规格化数	0000…0000	非0
NaN（Not a Number）	1111…1111	非0
+/- ∞	1111…1111	0

非规格化小数：$\mp(0.xxx)_2 \times 2^{-126}$

隐含最高位变为0　　阶码真值固定视为−126

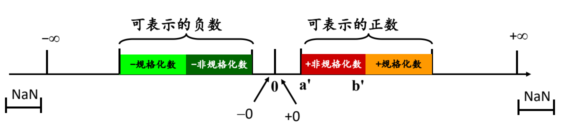
如：0/0，∞ − ∞等非法运算的结果就是NaN

非规格化数表示范围（单精度）

非规格化数 ⇒ E=0

- M=Frac，没有隐含的前导1
- e = 1 - Bias = 1 – 127 = -126，注：不是-127，保证了非规格化值到规格化值的平滑过渡
- 最小的正数：M=0.00…1_2
  - a' = 0.0…1_2 * 2^{1-127}
    = 2^{-23} * 2^{-126}
    = 2^{-149}
- 最大的正数：M=0.11…1_2
  - b'= 0.11…1_2 * 2^{1-127} = 0.11…1_2 * 2^{-126}
    = (1 - 2^{-23}) * 2^{-126}

浮点数表示的数值范围



IEEE 754 32位浮点数 小结

对32位浮点数N：

- 若0 < E < 255，则
  N = $(-1)^s$ ×1.M × 2^{E-127}，规格化数表示
- 若E = 0且M = 0，则
  N = $(-1)^s$ 0，机器+0、-0表示
- 若E = 0且M ≠ 0，则
  N = $(-1)^s$ × 0.M × 2^{1-127}，非规格化数表示
- 若E= 255且M = 0，则
  N=$(-1)^s$∞（正无穷大，负无穷大）
- 若E = 255且M ≠ 0，则
  N = NaN，非数NaN（Not a Number）

例3（1）

- 假设由S，E和M三个域组成的一个32位二进制数（E与M分别为8位和23位），所表示的非零规格化浮点数 x，真值表示为（注：不是IEEE754格式）：
  $x = (-1)^s \times (1.M) \times 2^{E-128}$
  问：它所表示的规格化的最大正数、最小正数、最大负数、最小负数是多少？

(1)最大正数

0 1111 1111 111 1111 1111 1111 1111 1111

$x = [1 + (1 - 2^{-23})] \times 2^{127}$

例3（2）

(2)最小正数

000 000 000000 000 000 000 000 000 000 00

$x = 1.0 \times 2^{-128}$

(3)最小负数

111 111 111111 111 111 111 111 111 111 11

$x = -[1 + (1 - 2^{-23})] \times 2^{127}$

(4)最大负数

100 000 000000 000 000 000 000 000 000 00

$x = -1.0 \times 2^{-128}$

例4

float型整数据常用IEEE754单精度浮点格式表示，假设两个float型变量x和y分别在32位寄存器f1和f2中，若（f1）=CC900000H，（f2）=B0C00000H，则x和y之间的关系为：

A x<y且符号相同

B x<y且符号不同

C x>y且符号相同

D x>y且符号不同

十进制数表示方法

- 字符串形式：一个字节存放一个十进制的数位或符号位
- 压缩的十进制数：一个字节存放两个十进制的数位
  - 用四位二进制表示一位十进制，16个编码状态选用其中的10个编码
  - 有多种BCD方案
    - 8421码
    - 余 3 码
    - 循环码

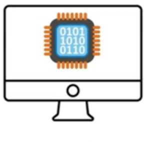
	8421	余3码	循环码
0	0000	0011	0000
1	0001	0100	0001
2	0010	0101	0011
3	0011	0110	0010
4	0100	0111	0110
5	0101	1000	1110
6	0110	1001	1010
7	0111	1010	1000
8	1000	1011	1100
9	1001	1100	0100

有权码

无权码

2.1.3 字符的表示方法

字符型数据的表示（ASCII码）



数字
字母 ——共128个字符——7位二进制编码—— ASCII码
符号
（控制符）

$2^7 = 128$

为了存入计算机，通常在最高位补0，凑足1字节

Unicode或 ISO 10646
统一码，表示多种语言文字

字符型数据的表示（ASCII码）

0	NUL	16	DLE	32	SPC	48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

可印刷字符：32~126，
其余为控制、通信字符

数字：48（0011 0000）~ 57（0011 1001）
大写字母：65（0100 0001）~90（0101 1010）
小写字母：97（0110 0001）~122（0111 1010）

汉字编码

计算机处理汉字信息的过程：

1.汉字输入到计算机－汉字输入码

2.计算机内部的表示和存储－汉字内码

3.计算机向外部显示和打印－汉字字形（字模）码

汉字的表示方法（1）

- 汉字输入码（外码）
  - 为了通过键盘字符把汉字输入计算机而设计的一种编码。汉字输入方案有很多，大致可分为以下4种类型
    - 音码：如全拼、双拼、微软拼音等
    - 形码：如五笔字型、郑码、表形码等
    - 音形码：如智能ABC、自然码等
    - 数字码：如区位码、电报码等

汉字的表示方法(2)

- 国标码GB 2312-1980
  - 共7445个字符（6763个汉字＋682个图形字符）
  - 区位码　94个区　94个位置

阿：16　02

↓　↓

10H 02H

用所在的区和位来对汉字进行编码，称为区位码。这个码是唯一的，不会有重码字

区位码 $\xrightarrow[+20H]{?}$ 国标码（保留ASCII控制符）

汉字的表示方法（3）

- 汉字机内码（内码）
  - 用于在计算机内部存储和传输汉字而设计的一种2字节编码。
  - 是否可以用国标码作为机器内码？
  - 为了避免ASCII码和国标码一起使用时产生二义性问题，把国标码每个字节的最高位置1，用以区别7位ASCII码，称为汉字机内码

汉字机内码、国标码和区位码三者之间的关系为：
区位码（十六进制）的两个字节分别加20H（32）得到对应的国标码；
汉字交换码（国标码）的两个字节分别加80H（128）得到对应的机内码；

阿：16　02

↓　↓

直观，方便理解　区位码　10H 02H

+20H↓　↓+20H　防止与国际字符、控制通信字符冲突

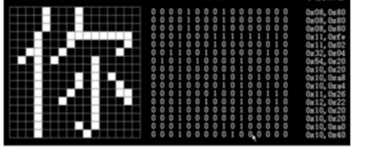
国标码　30H 22H

+80H↓　↓+80H　防止与ASCII冲突

汉字内码　B0H A2H

汉字的表示方法（4）

- 汉字字模码（输出码）
  - 用于把机内码转成能显示和打印的一种汉字编码，即用点阵表示的汉字代码



校验码

- 在数据存储、传输过程中，附加在数据中的可以用来检查和纠正因元件故障、噪声干扰等因素导致数据错误的编码
- 数据校验原理

信息位（k位）	校验码（r位）

- 分类
  - 检错码，如奇偶校验码、海明码、循环冗余校验码
  - 纠错码，如海明码、BCH码、RS码

奇偶校验码

设 X=（$x_0x_1…x_{n-1}$）

- 奇校验
  - 加上检验位后，传输的位组中，“1”的个数一定为奇数；否则就发生了错误
  - 奇校验位$\overline{C}$= $x_0 \oplus x_1 \oplus … \oplus x_{n-1}$
- 偶校验
  - 加上检验位后，传输的位组中，“1”的个数一定为偶数；否则就发生了错误
  - 偶校验位C= $x_0 \oplus x_1 \oplus … \oplus x_{n-1}$
- 奇/偶校验
  - $x_0' \oplus x_1' \oplus … \oplus x_{n-1}' \oplus C'$，结果为1/0说明出错/没出错

实例

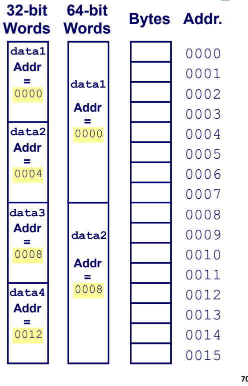
数据	偶校验编码	奇校验编码
10101010	10101010 0	10101010 1
01010100	01010100 1	01010100 0
00000000	00000000 0	00000000 1
11111111	11111111 0	11111111 1

偶校验		校验结果	
10101010 0	→ 10101011 0	1	✗
	→ 10101001 0	0	✓

奇偶校验能检测奇数个错误，无法检测偶数个错误。

数据在存储器中的表示

- 按字节寻址的存储器可视为一个很大的字节数组，数组下标为存储器地址，数组元素的值即为存储器存储的内容
- 连续存放的32位或64位的字数据的地址差值为4或8
- 多字节的字数据在存储器中是如何存放的？



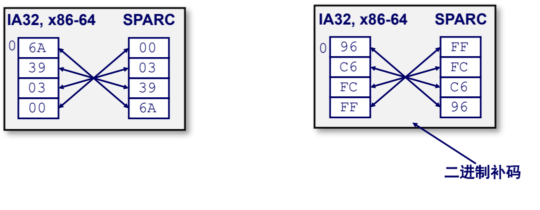
存储器中的字节序

- 小端方式Little Endian
  - 低字节存放在小地址处，即低字节在前高字节后
  - x86处理器
- 大端方式Big Endian
  - 低字节存放在大地址处，即高字节在前低字节后
  - SPARC处理器，IBM Power处理器

例 不同处理器存储整数的字节序

十进制:	211306				
二进制:	0011	0011	1001	0110	1010
十六进制:	3	3	9	6	A

int A = 211306;　　int B = -211306;



2.2 定点加法、减法运算

加法与减法运算

- 补码加法公式

整数 $[x]_补+[y]_补=[x+y]_补$ （mod 2^{n+1}）

小数 $[x]_补+[y]_补=[x+y]_补$ （mod 2）

- 补码减法公式

整数 $[x]_补-[y]_补=[x]_补+[-y]_补=[x-y]_补$ （mod 2^{n+1}）

小数 $[x]_补-[y]_补=[x]_补+[-y]_补=[x-y]_补$ （mod 2）

举例

- x ＝0.1001， y ＝0.0101，求 x ＋ y

解：$[x]_补=0.1001$, $[y]_补=0.0101$

$$\begin{array}{rl} [x]_补 & 0.1001 \\ +[y]_补 & 0.0101 \\ \hline [x+y]_补 & 0.1110 \end{array}$$

所以 x ＋ y ＝＋0.1110

举例

- x ＝＋0.1011， y ＝－0.0101， 求 x ＋ y

解：$[x]_补=0.1011$, $[y]_补=1.1011$

$$\begin{array}{rl} [x]_补 & 0.1011 \\ +[y]_补 & 1.1011 \\ \hline [x+y]_补 & 10.0110 \end{array}$$

所以 x ＋ y ＝＋0.0110

补码加法：符号位要作为数的一部分参与运算，进行模2的加法

证明$[x]_补-[y]_补=[x-y]_补$

补码减法的公式

$[x]_补-[y]_补\overset{?}{=}[x-y]_补=[x]_补+[-y]_补$ (mod 2)

证明上述公式成立的关键是$[-y]_补=-[y]_补$

证明$[-y]_补=-[y]_补$

证：

令 x=–y，于是

$[x+y]_补=[x]_补+[y]_补=[-y]_补+[y]_补$

又

$[x+y]_补=[-y+y]_补=[0]_补=0$

故

$[-y]_补+[y]_补=0$

也即

$[-y]_补=-[y]_补$

2/20/2023

$[-y]_{补}$求补法则

- 补码减法

  $[x]_{补}-[y]_{补}=[x]_{补}+[-y]_{补}=[x-y]_{补}$

- 在进行补码减法前，需从$[y]_{补}$求出$[-y]_{补}$。最简单的方法是：对$[y]_{补}$各位（包括符号位）“求反且末位加1”，即可得到$[-y]_{补}$
- 写成运算表达式，则为$[-y]_{补}=\neg[y]_{补}+2^{-n}$

  符号“$\neg$”表示对$[-y]_{补}$作包括符号位在内的求反操作

举例

$x_1=-0.1110$， $x_2=0.1101$，求$[x_1]_{补}$， $[-x_1]_{补}$，$[x_2]_{补}$，$[-x_2]_{补}$，

【解:】$[x_1]_{补}=1.0001+0.0001=1.0010$

$[-x_1]_{补}=\neg[x_1]_{补}+2^{-4}$

$=0.1101+0.0001$

$=0.1110$

$[x_2]_{补}=0.1101$

$[-x_2]_{补}=\neg[x_2]_{补}+2^{-4}$

$=1.0010+0.0001$

$=1.0011$

举例

$x=+0.1101$，$y=+0.0110$，求 $x-y$

解：$[x]_{补}=0.1101$

$[y]_{补}=0.0110$，$[-y]_{补}=1.1010$

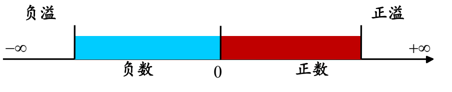
$$\begin{array}{rr} [x]_{补} & 0.1101 \\ +[-y]_{补} & 1.1010 \\ \hline [x-y]_{补} & 10.0111 \end{array}$$

所以 $x-y=+0.0111$

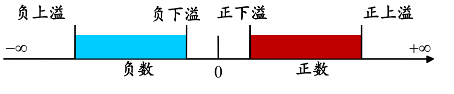
在模2的意义下相加，即超过2的进位要丢掉

溢出概念

- 无论是定点小数还是定点整数，在运算过程中会出现超出机器表示范围的现象，称之为“溢出”
- 正溢：运算结果为正，且超出机器所能表示的范围
- 负溢：运算结果为负，且超出机器所能表示的范围



- 上溢：结果的绝对值大于机器所能表示的最大绝对值（+∞，−∞）（overflow）
- 下溢：结果的绝对值小于机器所能表示的最小绝对值（机器零）（underflow）



实例

$x=+0.1011$，$y=+0.1001$，求 $x+y$？

解：$[x]_{补}=0.1011$，$[y]_{补}=0.1001$

$$\begin{array}{rr} [x]_{补} & 0.1011 \\ +[y]_{补} & 0.1001 \\ \hline [x+y]_{补} & 1.0100 \end{array}$$

两个正数相加的结果成为负数？

$x=-0.1101$，$y=-0.1011$，求 $x+y$？

解：$[x]_{补}=1.0011$，$[y]_{补}=1.0101$

$$\begin{array}{rr} [x]_{补} & 1.0011 \\ +[y]_{补} & 1.0101 \\ \hline [x+y]_{补} & 10.1000 \end{array}$$

两个负数相加的结果成为正数？

检测方法

1. 采用双符号位法，即“变形补码”或“模4补码”
2. 采用单符号位法

1. 双符号位法（1）

- 变形补码的定义

  小数：$[x]_{补}=\begin{cases} x & 2>x\geq 0 \\ 4+x & 0>x\geq -2 \end{cases}$

  整数：$[x]_{补}=2^{n+2}+x$ （mod 2^{n+2}）

  - 下式也同样成立

    $[x]_{补}+[y]_{补}=[x+y]_{补}$ (mod 2)

  - 注意事项
    - 两个符号位均参加运算
    - 最高符号位上产生的进位要丢掉

1. 双符号位法（2）

- 溢出检测规则
  - 两数相加后，结果的符号位出现“01”或“10”两种情况时，表示发生溢出
  - 最高符号位永远表示结果的正确符号
- 溢出的逻辑表达式为

  $V=S_{f1}\oplus S_{f2}$

举例

x ＝＋1100

y ＝＋1000，求 x ＋ y

解:[x]补 ＝001100

[y]补＝001000

[x]补 001100

＋[y]补 001000

[x ＋ y]补 010100

两个符号位为“01”，表示已经溢出

最高有效位产生进位而符号位无进位

x ＝－1100

y ＝－1000，求 x ＋ y

解:[x]补 ＝110100

[y]补＝111000

[x]补 110100

＋[y]补 111000

[x ＋ y]补 1101100

两个符号位为“10”，表示已经溢出

最高有效位无进位而符号位有进位

2. 单符号位法

- 溢出的逻辑表达式为

  $V=C_f\oplus C_0$

  C_f为符号位产生的进位，C_0为最高有效位产生的进位

例子

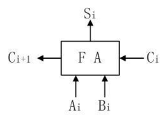
- 若x＝103，y=-25，采用8位定点补码运算时，会发生溢出的是：
- A x＋y
- B -x＋y
- C x-y
- D –x-y

2.2.4 基本的二进制加法／减法器

1位全加器FA（1）

$S_i = \overline{A}_i \overline{B}_i C_i + \overline{A}_i B_i \overline{C}_i$
$+ A_i \overline{B}_i \overline{C}_i + A_i B_i C_i$
$= A_i \oplus B_i \oplus C_i$

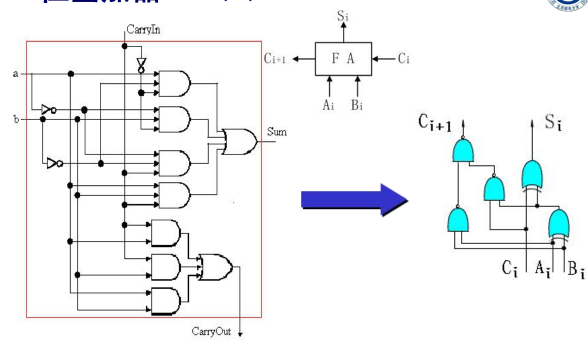
$C_{i+1} = \overline{A}_i B_i C_i + A_i \overline{B}_i C_i$
$+ A_i B_i \overline{C}_i + A_i B_i C_i$
$= A_iB_i + B_iC_i + C_iA_i$
$= A_iB_i + (A_i \oplus B_i)C_i$



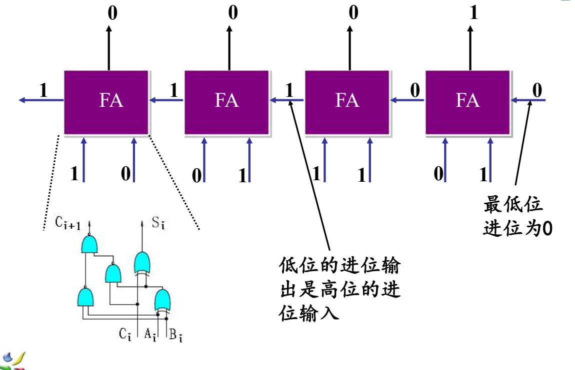
一位全加器真值表

输入			输出	
A_i	B_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

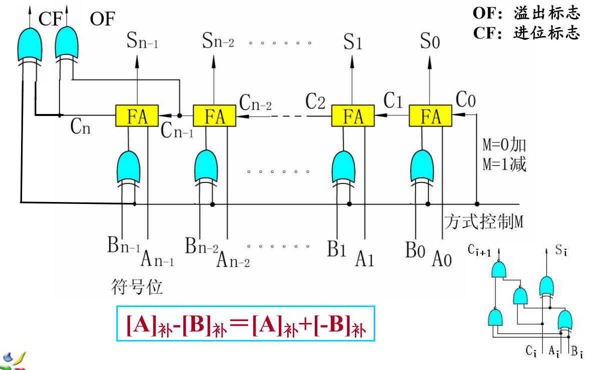
1位全加器FA（2）



4位加法器设计



n位行波进位的补码加法器



$[A]_补-[B]_补=[A]_补+[-B]_补$

n位行波进位加法器的时间延迟

- 考虑溢出检测，n位的行波进位加法器的时间延迟

  n = 2时

  $t_a = 3T + 3T + 2T + 2T + 3T$

  $= 6T + 2 \cdot 2T + 3T$

  n位时：$t_a = 3T + 3T + n \cdot 2T + 3T = (2n+9)T$

  其中T为单级逻辑电路的单位门延迟，每级异或门延迟3T

- 不考虑溢出检测时，n位的行波进位加法器的时间延迟

  $t_a = 3T + 3T + 2T + 3T$ (n=2时)

  $t_a = 3T + 3T + (n-1)2T + 3T = 2(n-1)T + 9T$ (n位时)

改进 分析进位链逻辑

- 加法器进位链的基本逻辑关系

  $C_{i+1} = A_iB_i +（A_i \oplus B_i）C_i$

- 令$G_i = A_iB_i$为进位产生函数（本地进位）

  $P_i = A_i \oplus B_i$为进位传递函数（进位条件）

  即：P_iC_i为传送进位或条件进位

  于是，$C_{i+1} = G_i + P_iC_i$

- 只有当$A_i = B_i = 1$时，本位才向高位进位
- 当$A_i \neq B_i$时，低一位的进位将向更高位传送
- 传送进位和本地进位不可能同时为1

进位产生和传递函数

$C_{i+1} = G_i + P_iC_i = G_i + P_i(G_{i-1} + P_{i-1}C_{i-1})$

$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

…………

- 依此公式递推，所有进位均可从最低进位C_0直接求得而无需等待低一位进位
- 对任意i，均可得到只含G_k、P_k（k=0~i）以及C_0的C_{i+1}的表达式
- 先行进位逻辑的总时间延迟为5T =（3T+2T）

每级与门或或门的延迟为T，每级异或门延迟3T

4位先行进位部件CLA

C1 = G0 + P0C0

C2 = G1 + P1G0 + P1P0C0

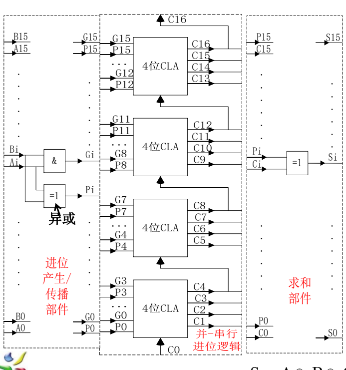
C3 = G2 + P2G1 + P2P1G0 + P2P1P0C0

C4 = G3 + P3G2 + P3P2G1 + P3P2P1G0 + P3P2P1P0C0

延迟时间：2T（不含求P_i的时间）

CLA：Carry Look Ahead

16位单级分组先行进位并行加法器



分成4个进位组，组内并行进位，组间串行进位。延迟时间：

- 先行进位部件：2T×4=8T
- 进位产生/传播部件：3T
- 求和部件：3T
- 加法器的总延迟时间：

t= 3T+4×2T+3T = 14T

比较：

- 串行进位16位加法器的总延迟时间：

t= 9T+2(n-1)T

= 9T+2×15T = 39T

$S_i = A_i \oplus B_i \oplus C_i$