# 数据的表示和运算

# 1. 数制&码制

## 1.1. 1/8/10/16进制及抓换

不必多说

## 1.2. 真值和机器数

1. 真值：带有正负的数字
2. 数字化：计算机用0/1代表正负
3. 机器数：结果数字化的真值

## 1.3. BCD码(二进制编码的十进制)

| 十进制数 | 8421码 | 2421码 | 余3码 |
| --- | --- | --- | --- |
| 0 | 0000 | 0000 | 0011 |
| 1 | 0001 | 0001 | 0100 |
| 2 | 0010 | 0010 | 0101 |
| 3 | 0011 | 0011 | 0110 |
| 4 | 0100 | 0100 | 0111 |
| 5 | 0101 | 1011 | 1000 |
| 6 | 0110 | 1100 | 1001 |
| 7 | 0111 | 1101 | 1010 |
| 8 | 1000 | 1110 | 1011 |
| 9 | 1001 | 1111 | 1100 |

### 1.3.1. 8421码(默认BCD码=8421码)

1. 规则
    1. 数的表示：将十进制数的每1位，转化为4位的二进制数
    2. 正负的表示：十六进制的C(1100)表示正，D(1101)表示负，均放数字串的最后
2. 注意事项
    1. 当十进制数是偶数位时，要在高位补4个0
    2. 8421BCD遇1001就要进位
3. 示例

表 2-2　8421 BCD 码示例一

| | | | | |
|---|---|---|---|---|
| +325 | 0011 | 0010 | 0101 | 1100 |
| -325 | 0011 | 0010 | 0101 | 1101 |

表 2-3　8421 BCD 码示例二

| | | | | |
|---|---|---|---|---|
| +56 | 0000 | 0101 | 0110 | 1100 |
| -56 | 0000 | 0101 | 0110 | 1101 |

### 1.3.2. 其余BCD编码

1. 余3码：8421码+0011(二进制3)
2. 当十进制数≥5时四位2421码的最高位为1，<5时，四位2421码的最高位为0，是有权码

## 1.4. 字符串和字符

1. 字符代码：对字符就行编码的二进制代码，最常用的为ASCII码
2. 字符串的大小端存放：
   1. 字符串：内存中占用连续字节，每字节放一个字符
   2. 当字符串占2or4字节时，字符串可以按照从低到高or从高到低字节摆放

## 1.5. 校验码

详见计算机网络，数据链路层笔记

# 2. 定点数的表示和运算

## 2.1. 定点数的表示

1. 无符号数表示：全部二进制位均为数值位，没有符号位，相当于绝对值
2. 有符号数的表示
   1. 计算原则
      - 三种码最高位都是符号位
      - 真值是正数时，三种码形式相同(符号位都为0，数值部分为真值)
      - 真值是负数时，符号位都为1，原码的数值部分仍为真值，反码数值部分为原码取反，补码数值为原码取反+1
   2. 注意事项
      - 0的原码和反码都有两种(+0/-0)，补码只有一种
      - $[x]_{补} \xrightarrow{连通符号位在内每位取反} \xrightarrow{末尾+1} [-x]_{补}$
      - $n$位机器字长的计算机中，补码表示范围为$-2^n \rightarrow 2^n-1$
   3. 补码特点
      - 不论多少位，-1的补码都是111111.....111，0的补码都是00000....0000
      - 最小负数的补码永远是1000.....0000

3️⃣有/无符号数范围区别：有符号数因为有一位被拿去表示正负，所以能表示的最大值砍半，但是范围大小不变

4️⃣移码

1. 补码的缺点：计算机内部会得出$[-21]_{补}=1,01011>[21]_{补}=0,10101$的比较，与事实不符
2. 移码规则：对$n$位的真值加上$2^n$

   $[21]_{移}=10101+100000=110101>[-21]_{移}=-10101+100000=001011$符合事实
3. 表示范围：机器字长为$n$，范围为$0\rightarrow 2^{n+1}-1$
4. 速算：**移码就是补码的符号位取反**

5️⃣小数的表示：$S=2^E(M_0,M_1M_2\dots M_n)$其中$M_0$是符号位

1. $E$为阶码：为顶点整数，用移码/补码表示
2. 一堆$M$为尾数：为定点小数，用原码/补码表示

| 情形 | 含义 | 真值 | 阶码表示 |
| --- | --- | --- | --- |
| $E=0$ | 纯小数 | -0.1111 | 1.1111 |
| $E=n$ | 纯整数 | +1111 | 01111 |
| $E=m<n$ | 浮点数 | +100.100 | $2^3$(0.100100) |

# 2.2. 定点数的运算

## 2.2.0. 计算机为何要采用补码

1️⃣补码能够使$[x]_{补}+[-x]_{补}=0$

2️⃣0的表示只有一种(0元不唯一时运算不可逆)，且可以多表示一个最小负数

3️⃣符号位可以和数值位一起参加运算

4️⃣补码可以将正数+负数，转化为正数+正数，将减法转化为加法方便加法器设计

5️⃣方便扩充，例如8位二进制数扩展为16位时只需在高位补上8个符号位

## 2.2.1. 定点数的移位运算

1️⃣逻辑位移

| 位移类型 | 高位 | 低位 |
| --- | --- | --- |
| 逻辑左移 | 丢弃 | 补0 |
| 逻辑右移 | 补0 | 丢弃 |

2️⃣算数位移：左移补0，右移补符号位

| | 码制 | 添补代码 |
|---|---|---|
| 正数 | 原码、补码、反码 | 0 |
| 负数 | 原码 | 0 |
| | 补码 | 左移添 0 |
| | | 右移添 1 |
| | 反码 | 1 |

这其实基于补码/反码的性质：补码中必定存在一个1，使得这个1右边与原码相同，左边与反码相同

```
原码：11010<1>000
补码：10101<1>000
反码：10101<0>111
```

3️⃣移位对解读的影响

1. 正数原码/反码/补码+负数原码：左移丢掉高位1→直接出错，右移丢掉低位1→影响精度
2. 负数反码：左移丢掉高位0→直接出错，右移丢掉低位0→影响精度
3. 负数补码：左移丢掉高位0→直接出错，右移丢掉低位1→影响精度

## 2.2.2. 原码定点数的加减

对于$[x]_{原}=x_0x_1...x_n$,$[y]_{原}=y_0y_1...y_n$

1️⃣加法

| $x,y$**符号位相同** | $x,y$**符号位不同** |
|---|---|
| 绝对值相加，符号位不变 | 绝对值相减，符号与绝对值大的数相同 |

2️⃣减法：将被减数符号位取反，x-y转化为x+(-y)

## 2.2.3. 补码定点数的加减

1️⃣加法：$[x+y]_{补}=[x]_{补}+[y]_{补}$

2️⃣减法：$[x-y]_{补}=[x]_{补}+[-y]_{补}$

## 2.2.4. 溢出及判定

1️⃣溢出原因：两补码相加大于上界(正溢出)/小于下界(下溢出)，两定点小数相加大于1(上溢)/小于-1(下溢)

2️⃣溢出的后果：数值位跑到符号位来了，结果与预期不同

3️⃣加减法溢出判定法(减法化为加法)

1. 正数+正数=负数，负数+负数=正数时溢出

2. 看符号位&最高有效位

| 是否溢出 | 数值最高位(最高有效位) | 符号位 |
| --- | --- | --- |
| 否 | 有进位 | 有进位 |
| 否 | 无进位 | 无进位 |
| 是 | 无进位 | 有进位 |
| 是 | 有进位 | 无进位 |

左边最高有效位无进位/符号位有进位→溢出，右边最高位有效位&符号位都进位→不溢出

$$[x+y]_{补}=[x]_{补}=1.0101 \quad [x+y]_{补}=[x]_{补}=1.1000$$
$$+[y]_{补}=\underline{1.1001} \qquad +[y]_{补}=\underline{1.1000}$$
$$0.1110 \qquad 1.0000$$

4️⃣两位符号位(变形补码)判定溢出

1. 双符号位运算规则：连通数值部分一起参加运算+高位符号位产生进位直接丢掉
2. 溢出判断原则：结果的高/低两符号位不同时就溢出，01表示正溢出/10表示负溢出，以下为负溢出示例

$$[x+y]_{补}=[x]_{补}=11.0101$$
$$+[y]_{补}=\underline{11.1001} \quad \text{（高符号位的 1 丢掉）}$$
$$110.1110$$

## 2.2.5. 定点数的乘法

### 2.2.5.1. 原码一位乘：以`x=0.1101，y=1.1011`为例

1️⃣对于符号位的处理：负负得正，所以$x,y$的符号位异或(相同为0/不同为1)即得到新的符号位

只要是原码运算，符号位通通单独处理(异或操作)

2️⃣对于数值位：先抛开符号位符号位全部设为0，称$x$为乘数/$y$为被乘数

1. $x,y$为$n$位时，共需$n$次加法和$n$次移位
2. 乘数最后一位为0时执行+0位移一位的操作，最后一位为1时执行+x位移一位的操作，操作完后次低位更新为低位
3. 每次加法仅将原来的部分积高位+$x$
4. 整数原码和小数原码都适用这个规则，小数对应`.`整数对应`,`

| 乘积寄存器 | y寄存器 | 当前所需执行的操作 |
| --- | --- | --- |
| 0.0000 | 1011 | y低位为1，乘积寄存器+x(0.1101) |
| 0.1101 | 1011 | 乘积寄存器右移1位；y寄存器接收这一位，然后也右移1位(舍低位) |
| 0.0110 | 1101 | y低位为1，乘积寄存器+x(0.1101) |

| 乘积寄存器 | y寄存器 | 当前所需执行的操作 |
| --- | --- | --- |
| 1.0011 | 1101 | 乘积寄存器右移1位；y寄存器接收这一位，然后也右移1位(舍低位) |
| 0.1001 | 1110 | y低位为0，乘积寄存器+0(0.0000) |
| 0.1001 | 1110 | 乘积寄存器右移1位；y寄存器接收这一位，然后也右移1位(舍低位) |
| 0.0100 | 1111 | y低位为1，乘积寄存器+x(0.1101) |
| 1.0001 | 1111 | 乘积寄存器右移1位；y寄存器接收这一位，然后也右移1位(舍低位) |
| 0.1000 | 1111 | 乘数所有位数都用完了，当前红色的就是结果 |

原色位初始状态

将每行的红色数字拼接，就是当前的**中间结果**

绿色代表当前的**乘数**

高亮代表乘积寄存器低位的**流向**

3️⃣ 注意事项：

1. 上述所有移位均是逻辑移位操作，即在高位+0
2. 考虑一位溢出，部分积一般用n+1位寄存器

### 2.2.5.2. 补码一位乘

1️⃣ 校正法：对于$x*y$

1. $y$为正数时，按照原码乘来算，但是原有的逻辑移位(高位补0)变为算数移位(高位补符号位)
2. $y$为负数时，按照原码乘来算，移位也是高位补符号位，最后结果还要加上$[-x]_{补}$校正

注意符号位不参与运算，原码一位乘同理

2️⃣ 比较法(booth法)：对于$x*y$

1. $x$与部分积取双符号位，符号位参与运算，采用补码算数位移(但只有次高位参与位移)
2. $y$取单符号位决定最后一步是否加$[-x]_{补}$校正
3. $y$尾部加上附件位$y_{n+1}$初始值设为0，根据$y_ny_{n+1}$判断下一步运算(如下表)

| $y_n y_{n+1}$ | $y_{n+1}-y_n$ | 操　作 |
|---|---|---|
| 00 | 0 | 部分积右移一位 |
| 01 | 1 | 部分积加$[x]_{补}$，再右移一位 |
| 10 | -1 | 部分积加$[-x]_{补}$，再右移一位 |
| 11 | 0 | 部分积右移一位 |

4. 按照上述算法执行$n$步，到$n+1$步时不再位移，只判断是否要加减$[x]_{补}$

3 比较法示例：$[x]_{补}=1.0101$，$[y]_{补}=1.0011$，求$[xy]_{补}$

先全取反末尾+1得$[-x]_{补}=0.1011$，然后需要执行4+1步运算

| 乘积寄存器 | y寄存器 | 当前所需执行的操作 |
|---|---|---|
| 00.0000 | 100110 | $y_n y_{n+1}=10$，部分积+$[-x]_{补}=00.1011$ |
| 00.1011 | 100110 | 执行右移，乘积寄存器低位塞给y寄存器，y寄存器挤掉自己低位 |
| 00.0101 | 110011 | $y_n y_{n+1}=11$，右移一位即可 |
| 00.0010 | 111001 | $y_n y_{n+1}=01$，部分积+$[x]_{补}=11.0101$ |
| 11.0111 | 111001 | 执行右移，乘积寄存器低位塞给y寄存器，y寄存器挤掉自己低位 |
| 11.1011 | 111100 | $y_n y_{n+1}=00$，右移一位即可 |
| 11.1101 | 111110 | $y_n y_{n+1}=10$，部分积+$[-x]_{补}=00.1011$ |
| 00.1000 | 111110 | 最后一步不再右移了，此时的部分积就是$[xy]_{补}$ |

原色位初始状态

将每行的红色数字拼接，就是当前的**中间结果**

绿色代表当前的**乘数**

紫色代表**附加位**

### 2.2.5.3. 补码二位乘

1 XY均用补码表示，符号位都参加运算，乘积的符号位由运算过程自动产生

2 部分积采用三位符号位运算，初值为0

3 设乘数数值部分为$n$位

1. $n$为奇数，乘数设1位符号位，做(n+1)/2次运算和移位，最后一步右移1位
2. $n$为偶数，乘数设2位符号位，做(n/2)+1次运算，n/2次移位，最后一步不移位

4 根据$y_{n-1}y_n y_{n+1}$的值来决定运算，如下

$y_{n-1}\ y_n\ y_{n+1}$= 000 时，$[Z_{i+2}]_{补}=[Z_i]_{补}+0$，算术右移2位;
$y_{n-1}\ y_n\ y_{n+1}$= 001 时，$[Z_{i+2}]_{补}=[Z_i]_{补}+[X]_{补}$，算术右移2位;
$y_{n-1}\ y_n\ y_{n+1}$= 010 时，$[Z_{i+2}]_{补}=[Z_i]_{补}+[X]_{补}$，算术右移2位;
$y_{n-1}\ y_n\ y_{n+1}$= 011 时，$[Z_{i+2}]_{补}=[Z_i]_{补}+2[X]_{补}$，算术右移2位;
$y_{n-1}\ y_n\ y_{n+1}$= 100 时，$[Z_{i+2}]_{补}=[Z_i]_{补}+2[-X]_{补}$，算术右移2位;
$y_{n-1}\ y_n\ y_{n+1}$= 101 时，$[Z_{i+2}]_{补}=[Z_i]_{补}+[-X]_{补}$，算术右移2位;
$y_{n-1}\ y_n\ y_{n+1}$= 110 时，$[Z_{i+2}]_{补}=[Z_i]_{补}+[-X]_{补}$，算术右移2位;
$y_{n-1}\ y_n\ y_{n+1}$= 111 时，$[Z_{i+2}]_{补}=[Z_i]_{补}+0$，算术右移2位;

5 运算完成后$y_{n-1}y_ny_{n+1}$全部清0

## 2.2.6. 定点数的除法运算

### 2.2.6.1. 原码恢复余数法：$[x/y]_{原}$为例

1 运算前的处理：

1. 符号位单独处理，取$x, y$绝对值运算
2. 判断是否满足$0 < |x| < |y|$，不满足的话必定溢出

2 运算规则：被除数$x$也可以看成余数

1. 第一次：先将$x$减去被除数即$x = x + [-y]_{补}$

| 第1次运算结果 | 商低位塞? | 需要恢复余数? | 下一步操作(得到第二次结果) |
|---|---|---|---|
| $x > 0$ | 0 | 不需要 | $x$左移1位，<br>$x = x + [-y]_{补}$ |
| $x < 0$ | 0 | 恢复<br>$x = x + [y]_{补}$ | $x$左移1位，<br>$x = x + [-y]_{补}$ |

2. 以后若干次，参考第二次结果

| 第2次运算结果 | 商低位塞? | 需要恢复余数? | 下一步操作 |
|---|---|---|---|
| $x > 0$ | 1 | 不需要 | $x$左移1位，<br>$x = x + [-y]_{补}$ |
| $x < 0$ | 0 | 恢复<br>$x = x + [y]_{补}$ | $x$左移1位，<br>$x = x + [-y]_{补}$ |

3. 重复以上步骤$n$次，设置一个计数器来控制次数
4. 完成最后一次的运算，如果$x < 0$则恢复余数$x = x + [y]_{补}$，否则不需要

3 运算示例：$x = -0.10110, y = 0.11111$，计算$[x/y]_{原}$

先求出$[y]_{补} = 00.11111$，$[-y]_{补} = 11.00001$

| 步数(conut) | 被除数/余数/x | 商 | 当前所需执行的操作 |
| --- | --- | --- | --- |
| 初始状态 | 00.10110 | xxxxxx | 执行$x = x + [-y]_{补}$，进入第一步 |
| 第一步 (count=5) | 11.10111 < 0 | xxxxxx | 商低位一律塞0，恢复 $x = x + [y]_{补}$ |
| 第一步 (count=5) | 00.10110 | 0xxxxx | $x$左移1位(上商) |
| 第一步 (count=5) | 01.01100 | 0xxxxx | 执行$x = x + [-y]_{补}$，进入下一步 |
| 第二步 (count=4) | 00.01101 > 0 | 0xxxxx | 商低位塞1，无需恢复余数 |
| 第二步 (count=4) | 00.01101 | 01xxxx | $x$左移1位(上商) |
| 第二步 (count=4) | 00.11010 | 01xxxx | 执行$x = x + [-y]_{补}$，进入下一步 |
| 第三步 (count=3) | 11.11011 < 0 | 01xxxx | 商低位塞0，恢复$x = x + [y]_{补}$ |
| 第三步 (count=3) | 00.11010 | 010xxx | $x$左移1位(上商) |
| 第三步 (count=3) | 01.10100 | 010xxx | 执行$x = x + [-y]_{补}$，进入下一步 |
| 第四步 (count=2) | 00.10101 > 0 | 010xxx | 商低位塞1，无需恢复余数 |
| 第四步 (count=2) | 00.10101 | 0101xx | $x$左移1位(上商) |
| 第四步 (count=2) | 01.01010 | 0101xx | 执行$x = x + [-y]_{补}$，进入下一步 |
| 第五步 (count=1) | 00.01011 > 0 | 0101xx | 商低位塞1，无需恢复余数 |
| 第五步 (count=1) | 00.01011 | 01011x | $x$左移1位(上商) |
| 第五步 (count=1) | 00.10110 | 01011x | 执行$x = x + [-y]_{补}$，进入下一步 |
| 第六步 (count=0) | 11.10111 < 0 | 01011x | 商低位塞0，恢复$x = x + [y]_{补}$ |
| 第六步 (count=0) | 00.10110 | 010110 | count=0终止 |

1. 加上符号商结果为**-0.10110**
2. 全过程逻辑左移了$n=5$次，所以余数为$x*2^{-n}$为**0.0000010110**

4️⃣总结：

1. $n$位尾数的合法除法，需要逻辑移位$n$次，上商$n+1$次
2. 缺点在于不知道要恢复多少次余数，会使电路设计复杂

### 2.2.6.2. 原码不恢复余数法(加减交替法)：$[x/y]_{原}$为例

1️⃣步骤：第一步先执行$x=x+[-y]_{补}$，然后循环以下步骤

| 运算结果 | 商低位塞? | 移位 | 下一步操作 |
| --- | --- | --- | --- |
| $x>0$ | 1 | 左移1位 | $x=x+[-y]_{补}$ |
| $x<0$ | 0 | 左移1位 | $x=x+[y]_{补}$ |

最后一步：$x<0$时再$x=x+[y]_{补}$恢复

2️⃣示例：$x=-0.10110, y=0.11111$，计算$[x/y]_{原}$

先求出$[y]_{补}=00.11111$，$[-y]_{补}=11.00001$

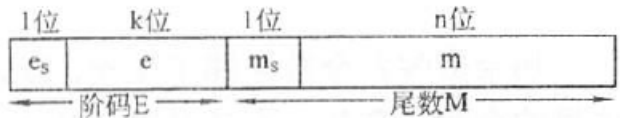
| 步数(conut) | 被除数/余数/x | 商 | 当前所需执行的操作 |
| --- | --- | --- | --- |
| 初始状态 | 00.10110 | xxxxxx | 执行$x=x+[-y]_{补}$，进入第一步 |
| 第一步 (count=5) | 11.10111 < 0 | xxxxxx | 商低位塞0，左移一位 |
| 第一步 (count=5) | 11.01110 | 0xxxxx | $x=x+[y]_{补}$ |
| 第二步 (count=4) | 00.01101 > 0 | 0xxxxx | 商低位塞1，左移一位 |
| 第二步 (count=4) | 00.11010 | 01xxxx | $x=x+[-y]_{补}$ |
| 第三步 (count=3) | 11.11011 < 0 | 01xxxx | 商低位塞0，左移一位 |
| 第三步 (count=3) | 11.10110 | 010xxx | $x=x+[y]_{补}$ |
| 第四步 (count=2) | 00.10101 > 0 | 010xxx | 商低位塞1，左移一位 |
| 第四步 (count=2) | 01.01010 | 0101xx | $x=x+[-y]_{补}$ |
| 第五步 (count=1) | 00.01011 > 0 | 0101xx | 商低位塞1，左移一位 |

| 步数(conut) | 被除数/余数/x | 商 | 当前所需执行的操作 |
| --- | --- | --- | --- |
| 第五步 (count=1) | 00.10110 | 01011x | $x = x + [-y]_{补}$ |
| 第六步 (count=0) | 11.10111 < 0 | 01011x | 商低位塞0，但由于count=0所以不左移 |
| 第六步 (count=0) | 11.10111 | 010110 | x<0所以还要恢复一次，$x = x + [y]_{补}$ |
| 第六步 (count=0) | 00.10110 | 010110 | 这便是结果 |

# 3. 浮点数的表示和运算

## 3.1. 浮点数的表示

### 3.1.1. 浮点数的一般表示$N = r^E * M$




1. 阶码

    1. 为定点整数，常用补码/移码表示
    2. $e_s$为其符号位
    3. $k$反应了浮点数表示范围+小数点位置

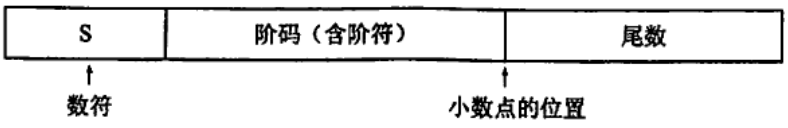
2. 尾数

    1. 为定点小数(一般是纯小数)，常用原码/补码表示
    2. $m_s$为其符号位(决定了整个浮点数的符号)
    3. $n$反应了浮点数的精度

3. 底$r$省略

### 3.1.2. IEEE754标准表示

1. 基本结构&常用浮点数




| | 符号位 S | 阶码 | 尾数 | 总位数 | 最大指数 | 最小指数 | 指数偏移值 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 短实数 | 1 | 8 | 23 | 32 | +127 | -126 | +127 |
| 长实数 | 1 | 11 | 52 | 64 | +1023 | -1022 | +1023 |
| 临时实数 | 1 | 15 | 64 | 80 | +16383 | -16382 | +16383 |

    1. $S$数符表示了浮点数的正负

2. 尾数使用原码表示，阶码使用移码表示

2️⃣ 示例：$x$的IEEE754表示是`41360000H`，求$x$的十进制表示

1. 展开`41360000H = 0100 0001 0011 0110 0000 0000 0000 0000`
2. 按照短实数格式排列为`0 10000010 01101100000000000000000`
   - 数符：为`0`表示正数
   - 指数：`e = 阶码10000010 - 127 = 3`，即为$2^3$
   - 尾数：加上隐藏的1，为`1.01101100000000000000000 = 1.011011`
3. 综上所述：`x = +1011.011 = 11.375`

## 3.2. 浮点数的加减运算

### 3.2.1. 浮点数的规格化

1️⃣ 什么是规范化数：基数为2，尾数$W$满足$\frac{1}{2} \leq |W| < 1$时，其所代表浮点数就是规范化的

2️⃣ 特殊情况：

1. 原码表示尾数/补码表示尾数(正数)：形式必定为`0.1xxx...x`或者`1.1xxx...x`
2. 补码表示尾数(负数)：形式必定为`0.0xxx...x`或者`1.0xxx...x`

   采用双符号位时为：`00.0xxx...x`或者`11.0xxx...x`

3️⃣ 特殊情况：`-1/2`不是规范化数，`-1`特别规定为规范化数

### 3.2.2. 浮点数加减的步骤

1️⃣ 对阶：小数点对齐，一定是低阶向高阶对齐。以1100100+1010为例

```
2^7 * 00.1100100 (尾数采用补码表示)
2^4 * 00.1010000 (尾数采用补码表示)
向高阶对齐
2^7 * 00.1100100
2^7 * 00.0001010
```

2️⃣ 尾数求和：

```
2^7 * 00.1101110
```

3️⃣ 规格化：以下所说的移位都是补符号位

1. 左规：尾数求和后，出现`00.0xxx...x`或者`11.1xxx...x`，则一直执行左移直到满足补码规格化
2. 右规：尾数求和后，出现`01.xxxx...x`或者`10.xxxx...x`，右移一次即可

4️⃣ 舍入：为提高精度，要考虑右移时丢弃的数值位

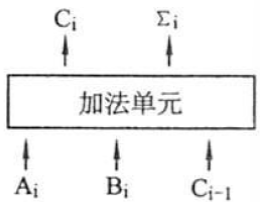
1. 0舍1入：尾数右移时，末位为0则直接移除，末位为1时则移除后再加回1
2. 恒置1法：不论右移移除的是0还是1，都移除后强行加1

5️⃣ 检查溢出：假设用补码+双符号位判断溢出，则01为上溢/10为下溢/其余为正常

# 4. 算数逻辑单元

## 4.1. 串行/并行加法器

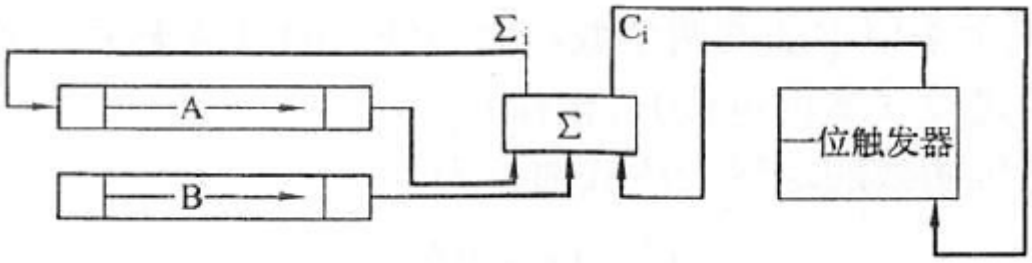
### 4.1.1. 全加器：求和单元+进位链




1. $\sum_i$是三者相加的结果：$A_iB_iC_{i-1}$=100/010/001/111时$\sum_i = 1$
2. $C_i$存放进位：$A_iB_iC_{i-1}$=110/101/011/111是$C_i = 1$

### 4.1.2. 串行加法器

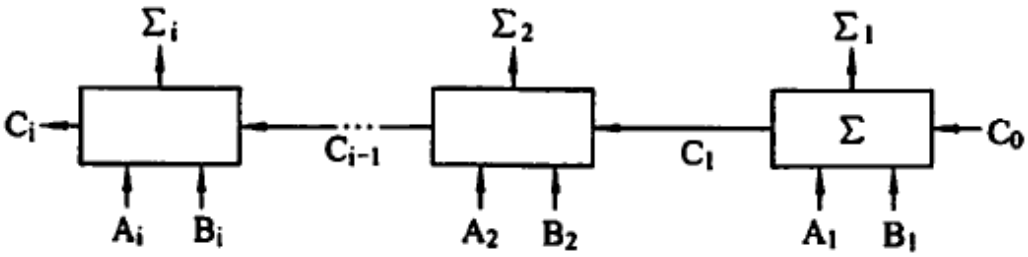
1. 结构




1. 只含有一个加法器
2. 有两个移位寄存器，通过移位将操作数压入加法器，其中一个存储最终计算结果
3. 用一位触发器来纪录进位信息

2. 缺点：$n$位操作数就需要$n$次串行计算，太慢

### 4.1.3. 并行加法器：串行进位链

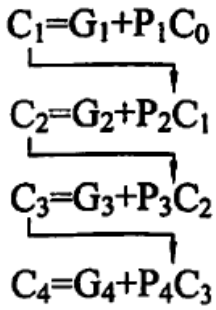



1. 改进的点：一位一位的加→所有位同时加
2. 缺点：进位是低位向高位传递的，还是太慢

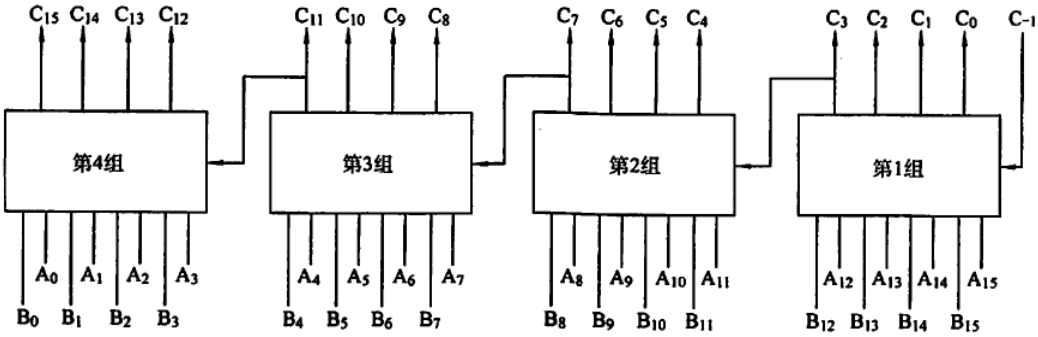
## 4.1.4. 并行加法器：并行进位链

1 进位逻辑：$C_i = G_i + P_i C_{i-1}$其中⊕表示异或

1. 本地进位$G_i = A_i B_i$：记录本地的进位，与低位无关
2. 进位条件$P_i = A_i \oplus B_i$：只有当$P_i = 1$时低位的进位才能向上传递
3. 串行进位的逻辑



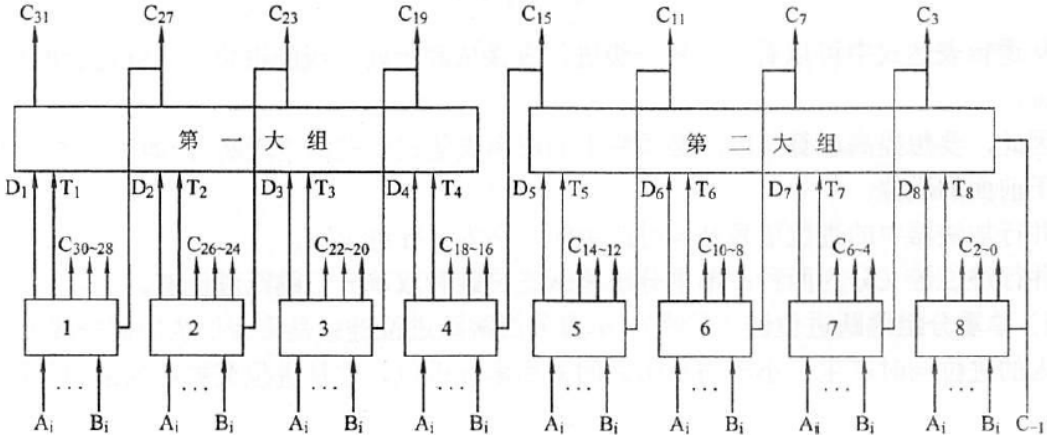

2 单重分组跳跃进位链：组内并行




1. 结构：所有全加器分组，每组同时产生进位，组与组间串行进位
2. 代入法解析：以第一组为例，只要$C_0$生成后，$C_1, C_2, C_3, C_4$便可以同时生成

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ P_2 & 1 & 0 & 0 \\ P_3P_2 & P_3 & 1 & 0 \\ P_4P_3P_2 & P_4P_3 & P_4 & 1 \end{pmatrix} \begin{pmatrix} G_1 \\ G_2 \\ G_3 \\ G_4 \end{pmatrix} + \begin{pmatrix} P_1 \\ P_2P_1 \\ P_3P_2P_1 \\ P_4P_3P_2P_1 \end{pmatrix} C_0 : three :$$

3 双重分组跳跃进位链：组内/组间都并行，所有加法器分为大组/大组又分为小组




1. 每个小组会产生两种进位，最高位进位和其他位进位，但是最高位进位会有一定延迟(虽然并行)
2. 大组内各个小组的最高位进位同时产生
3. 大组与大组之间串行进位

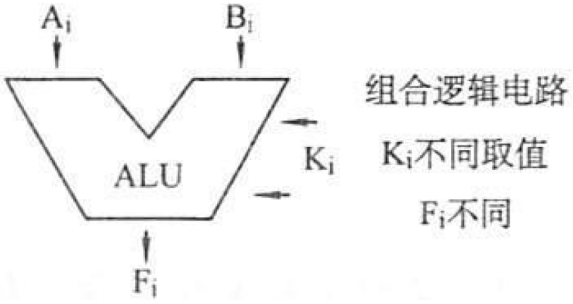
# 4.2. ALU

## 4.2.1. 组合/时序逻辑电路

1. 组合逻辑电路：无记忆，此时的输出只取决于此时的输入，结果要立马送寄存器，**比如ALU**

2. 时序逻辑电路：有记忆，由此时输入+电路原来状态共同决定输出，**比如触发器/CPU**

## 4.2.2. ALU概述




1. 功能：执行+-*/算术运算，执行与或非异或逻辑运算，执行并行进位

2. 锁存器：其实就是存储多位的触发器，临时存放数据，ALU的AB端都必须连接内容不变的锁存器

3. 电路框架：$A_i, B_i \xrightarrow[\text{运算后输出}]{K_i\text{控制信号}} F_i$

## 4.2.3. 算术逻辑运算74181芯片

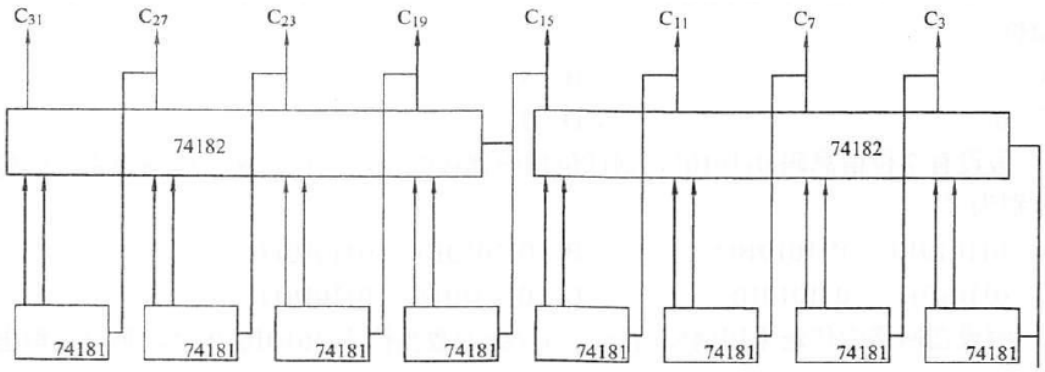
1. 结构：实质上是4位的ALU电路

2. 组合：4片74181芯片拼在一起，构成16位全加器(按4位一组的单重分组跳跃进位链)

3. 功能：

    1. 4片74181，组内并行组间串行，可完成16种算数/16种逻辑运算
    2. 将组件串行变为组间并行：有请74182芯片

## 4.2.4. 先行进位74812芯片




1. 结构：

    1. 74181可以看作双重分组跳跃进位链的小组，74182可以看作大组
    2. 2片74182芯片+8片74181芯片就可组成32位的双重分组跳跃进位链

2. 位数：74181的位数*4