# 第二章 运算方法和运算器

- 2.1 数据与文字的表示
- 2.2 定点加法、减法运算
- 2.3 定点乘法运算
- 2.4 定点除法运算
- 2.5 定点运算器的组成
- 2.6 浮点运算与浮点运算器



## 2.1数据与文字的表示方法

- 计算机数字和字符的表示方法应有利于数据的存储、加工(处理)、传送；
- 编码：用少量、简单的基本符号，选择合适的规则表示尽量多的信息，同时利于信息处理（速度、方便）
- 计算机中使用的数据可分成两大类：
  - **符号数据**：非数字符号的表示（ASCII、汉字、图形等）
  - **数值数据**：数字数据的表示方式（定点、浮点）



## 2.1.2数的机器码表示

- 机器码：以0/1形式存在于计算机中的数据，可以是带符号或无符号数据。
- 真值：一般书写的数，真实大小

- 无符号整数表示：直接转换为2进制或编码为BCD
  - 例：$(175)_D \rightarrow (10101111)_B$
  - BCD：使用4位二进制表示已位10进制，常用的8421/5421/2421等。
    - 例：$(175)_D \rightarrow (1\ 0111\ 0101)_{8421}$ $(175)_D \rightarrow (1\ 0111\ 1000)_{5421}$

- 带符号整数表示：编码为原码/反码/补码/ 移码
- 原码/反码/补码/ 移码 **编码格式**：
  - **1位符号位 + n位数值位**
  - 编码使用的0/1的位数，没有说明，一般使用8位编码格式



## 2.1.2 数的机器码表示

- 原码：带符号整数原码形式为$X_nX_{n-1}...X_2X_1X_0$,共n+1位
  - $X_n$：最高位,符号位，规定：0表示正，1表示负；其余低位是数值位。

$$[x]_{原}=\begin{cases}0x, 2^n>x\geq 0\\ 2^n-x=2^n+|x|, 0\geq x\geq -2^n\end{cases}$$

  - $[x]_{原}$是机器原码表示； x是真值；
  - 原码求解一般情况：
  - x是正数，$[x]_{原}=0$ x
  - x是负数，$[x]_{原}=1$ |x|



## 2.1.2 数的机器码表示

- 原码：带符号整数原码形式为$X_nX_{n-1}...X_2X_1X_0$,共n+1位
  - **例**：
    - X=+15 |+15|→$(1111)_B$
      $[+15]_{原}=0\ 1111$ $[+15]_{原}=0\ 0001111$
    - X=-15 |-15|→$(1111)_B$
      $[-15]_{原}=1\ 1111$ $[-15]_{原}=1\ 0001111$
  - **例**：
    - X=+10101
      $[+10101]_{原}=0\ \ 10101$
      $[+10101]_{原}=0\ 0010101$
    - X=-10101
      $[-10101]_{原}=1\ \ 10101$
      $[-10101]_{原}=1\ 0010101$



## 2.1.2 数的机器码表示

- 补码：带符号整数补码形式为$X_nX_{n-1}...X_2X_1X_0$,共n+1位
  - $X_n$：最高位,符号位，规定：0表示正，1表示负；其余低位是数值位。

$$[x]_{补}=\begin{cases}0x, 2^n>x\geq 0\\ 2^{n+1}+x=2^{n+1}-|x|, 0\geq x\geq -2^n\end{cases}$$

  - $[x]_{补}$是机器补码表示； x是真值；
  - 补码求解一般情况：
  - x是正数，$[x]_{补}=0$ x
  - x是负数，$[x]_{补}=1$ (~|x|+1)//x按位取反加1

## 2.1.2 数的机器码表示

- 补码：带符号整数补码形式为$X_nX_{n-1}...X_2X_1X_0$,共n+1位
  - 例：
    - X=+15 $|+15|\rightarrow(1111)_B$
      $[+15]_{补}=0\ 1111$ $[+15]_{补}=0\ 0001111$
    - X=-15 $|-15|\rightarrow(1111)_B$
      $[-15]_{补}=1\ 0001$ $[-15]_{补}=1\ 1110001$
  - 例：
    - X=+10101
      $[+10101]_{补}=0\ \ 10101$
      $[+10101]_{补}=0\ 0010101$
    - X=-10101
      $[-10101]_{补}=1\ \ 01011$
      $[-10101]_{补}=1\ 1101011$



## 2.1.2 数的机器码表示

- 反码：
  - 正数的反码与对应正数的原码相同；
  - 负数的反码将对应负数的原码数值位按位取反，最高位/数符位不变；
    - 例：$(+15)_D\rightarrow(0\ 0001111)_B$ $(-15)_D\rightarrow(1\ 1110000)_B$
- 移码：
  - 符号位取值与补码相反，数值位与补码相同。符号位0表示负，1表示正。
  - 一般用于阶码表示。
    - 例：$(+15)_D\rightarrow(1\ 0001111)_B$ $(-15)_D\rightarrow(0\ 1110001)_B$
  - **已知原码，求反码：正数相同，负数符号位不变，数值位按位取反。**
  - **已知原码，补码：正数相同，负数符号位不变，数值位按位取反加1。**
  - **补码→移码**：符号位取反，数值位不变。



## 2.1.2 数的机器码表示

- **例**：8位机器编码表示：X1=+101011 X2=-101011
  - $[X1]_{原}=0\ 0101011$ $[X1]_{反}=0\ 0101011$
  - $[X1]_{补}=0\ 0101011$ $[X1]_{移}=1\ 0101011$
  - $[X2]_{原}=1\ 0101011$ $[X2]_{反}=1\ 1010100$
  - $[X2]_{补}=1\ 1010101$ $[X2]_{移}=0\ 1010101$

- **例**：8位机器编码补码表示：求真值X1和X2 。
  - $[X1]_{补}=0\ 1011011$ $[X2]_{补}=1\ 1011011$
  - $X1=0\times2^7+1\times2^6+0\times2^5+1\times2^4+1\times2^3+0\times2^2+1\times2^1+1\times2^0=+91$
  - $X2=-1\times2^7+1\times2^6+0\times2^5+1\times2^4+1\times2^3+0\times2^2+1\times2^1+1\times2^0=-37$



**[例7]将十进制真值(－127,－1,0,＋1,＋127)列表表示成二进制数及原码、反码、补码、移码值。**

| 真值x（十进制） | 真值x（二进制） | [x]原 | [x]反 | [x]补 | [x]移 |
|---|---|---|---|---|---|
| -127 | -01111111 | 11111111 | 10000000 | 10000001 | 00000001 |
| -1 | -00000001 | 10000001 | 11111110 | 11111111 | 01111111 |
| | | 00000000 | 00000000 | | |
| 0 | 00000000 | | | 00000000 | 10000000 |
| | | 10000000 | 11111111 | | |
| +1 | +00000001 | 00000001 | 00000001 | 00000001 | 10000001 |
| +127 | +01111111 | 01111111 | 01111111 | 01111111 | 11111111 |



## 2.1.2数的机器码表示(例6)

- 表示范围的问题：1符号位，n位数值位；n+1位编码
  - 原码表示范围：
    - 有正0和负0之分 0 000……000/ 1 000……000
    - 纯整数表示范围$-(2^n-1)\sim+(2^n-1)$
    - 纯小数表示范围$-(1-2^{-n})\sim+(1-2^{-n})$
  - 反码表示范围：
    - 有正0和负0之分0 000……000/ 1 111……111
    - 纯整数表示范围$-(2^n-1)\sim+(2^n-1)$
    - 纯小数表示范围$-(1-2^{-n})\sim+(1-2^{-n})$
  - 补码表示范围：
    - 正0和负0表示相同，都为0 000……000
    - 纯整数表示范围$-(2^n)\sim+(2^n-1)$
    - 纯小数表示范围$-1\sim+(1-2^{-n})$
  - 移码表示范围：与补码相同



## 2.1.2 数的机器码表示

- 10进制带符号数据表示：
  - **字符串**形式：8位二进制/1个字节表示1位十进制，使用连续的空间保存一个十进制数据的各位取值。**ASCII码**
    例：+25→ | + | 2 | 5 | -149→ | - | 1 | 4 | 9 |
  - 压缩的**十进制数串**形式： 4位二进制/半个字节表示1位十进制，1个字节表示相邻两位十进制，使用连续的空间保存一个十进制数据的各位取值。**8421BCD**
  - 符号位4位，使用6中无效编码中的取值表示，位置在数串的最后：**C→正号 D→负号**
  - 要求位数（**符号位+数值位**）位偶数，否则最高位补0；

例：+25→ | 0 | 2 | 5 | C | -149→ | 1 | 4 | 9 | D |

## 2.1.1 数据格式

- 带小数点类型的数据如何表示?
- 计算机中数的表示使用两种方法提供小数点的位置信息。
- **定点格式**：小数点的位置固定；
  - **纯小数格式**：小数点的位置固定在数据的最左边（0.XX）
  - **纯整数格式**：小数点的位置固定在数据的最左边（XX.）
- **浮点格式**：小数点的位置不固定；浮点数格式



## 2.1.1 数据格式

- 定点数的表示方法：
  - 在纯小数和纯整数格式中，由于小数点的位置是固定不变的，在计算机中保存定点数时，不用保存小数点的位置信息；
- 表示格式：$x = x_0 x_1 x_2 \cdots x_n$
  - 其中$x_0$是符号位，$x_1 x_2 \cdots x_n$是数值位。
  - 如果是**纯小数**，小数点在$x_0 x_1$之间。数据是$x_0 . x_1 x_2 \cdots x_n$
  - 如果是**纯整数**，小数点在$x_n$之后。数据是$x_0 x_1 x_2 \cdots x_n .$
- 例：使用8位原码格式表示定点数：
  - +1010110. → 0 1010110
  - - 1101001. → 1 1101001
  - +0.11011 → 0 1101100 (后补2个0)
  - - 0.10101 → 1 1010100 (后补2个0)



## 2.1.1 数据格式

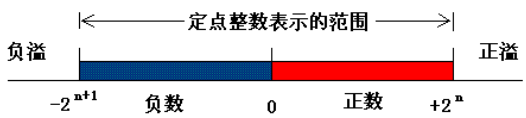
- 特殊的定点数表示举例：原码格式

| | | |
|---|---|---|
| x=0.00...0<br>x=1.00...0 | x=0 | 正0和负0都是0 |
| x=0.11...1 | $x=1-2^{-n}$ | 最大正数 |
| x=0.00...01 | $x=2^{-n}$ | 最接近0的正数 |
| x=1.00...01 | $x=-2^{-n}$ | 最接近0的负数 |
| x=1.11...1 | $x=-(1-2^{-n})$ | 最小负数 |



## 定点表示法的特点

- 目前计算机中多采用定点纯整数表示，因此将定点数表示的运算简称为整数运算。
- 无论是整数或是小数，在机器数的表示中，都不出现小数点“.”,只是约定其位置。
- 定点表示的精度有限：定点数表示数的范围受字长限制，表示数的范围有限；
- 溢出的问题：负溢出与正溢出？






## 定点表示法的特点

- [例8]设机器字长16位,定点表示,尾数15位

**(1)定点原码整数**表示时,最大正数是多少?最小负数是多少?

- 最大正整数：0 111 111 111 111 111
- 最大正整数真值：$(2^{15}-1)_{10}=(+32767)_{10}$
- 最小负整数：1 111 111 111 111 111
- 最小负整数真值：$(1-2^{15})_{10}=-(2^{15}-1)_{10}=(-32767)_{10}$

**(2)定点原码小数**表示，最大正数是多少?最小负数是多少?

- 最大正小数：0 111 111 111 111 111
- 最大正小数真值：$(1-2^{15})_{10}$
- 最小负小数：1 111 111 111 111 111
- 最小负小数真值：$-(1-2^{15})_{10}$



## 2、浮点数的表示方法

- 浮点数表示：$M \times R^E$

例：156.78 $=15.678\times10^1 = 1.5678\times10^2$
$= 0.15678\times10^3 = M\times R^E$

- 基数R：底数，10进制为10,2进制为2;
- E为阶码：指数，表示浮点数的范围;
- 尾数M：表示浮点数的有效数字;
  - M规格化：在定点计算机中，约定：尾数$|M|<1.0$，纯小数。
  - 如：$+156.67=0.15678\times10^3$
- **2进制浮点数尾数规格化：纯小数且小数点后第一位为1。**

例：$+1011.1101 = M\times R^E$
$= +0.10111101\times2^{+4}$
$= 0.10111101\times2^{+100}$

## 2、浮点数的表示方法

- 浮点数真值：$N = \pm R^E \times M$
- 编码/表示格式：n+1+k+1位二进制编码，格式如图：

| $M_s$ | $E_s$ | $E_1$ | … | $E_k$ | $M_1$ | $M_2$ | … | $M_n$ |
|---|---|---|---|---|---|---|---|---|

  - R：底数，隐含约定基数为2，不需要编码。
  - E= $E_s$ $E_1$…$E_K$：阶码，为带符号定点整数，一般是移码表示。需要K+1位；其位数决定数值范围；
  - $E_s$：表示阶码的符号，需要1位；阶符。
  - M = $M_s$ $M_1M_2$…$M_n$：尾数，为带符号定点小数，一般是原码表示。需要n+1位；其位数决定浮点数的精度； $1/2 \leq |M| < 1$，尾数必须规格化。
  - $M_s$：数符位表示浮点数N的正负，需要1位；
- 浮点数的编码是一个三元组：S(符号域),E(阶码域),M(尾数域)。



## 2、浮点数的表示方法

- **IEEE754**标准：规定了浮点数的表示格式，运算规则等；
  - 单精度(32)**float**
  - 双精度(64)**double**
  - 尾数用**原码**，阶码用**移码**







## 2、浮点数的表示方法

- 32位表示浮点数编码----**单精度浮点数float**

| 31 | 30 | 29 | … | 23 | 22 | 21 | … | 0 |
|---|---|---|---|---|---|---|---|---|
| $M_s$ | $E_1$ | $E_2$ | … | $E_k$ | $M_1$ | $M_2$ | … | $M_n$ |

  - 数符位$M_s$：使用最高位第31位表示；
  - 阶码**E**：8位（第30~23）移码表示；不需要符号位，全部是数值位。
    - 8 位阶码采用移码方式来表示正负指数。将浮点数的指数真值 $e$ 变成阶码 $E$ 时，应将指数 $e$ 加上一个固定的偏移值127，即 $E=e+127$
  - 尾数**M**：23位（第22~0）原码表示；
    - 采用隐含尾数最高位1的表示方法，实际尾数24位，**尾数真值 = 1 + 尾数**；
  - 编码表示非0浮点数真值为：$(-1)^{M_s} \times 2^{E-127} \times (1 + \text{尾数})$



## 2、浮点数的表示方法

- 32位表示浮点数编码----**单精度浮点数float**

| 63 | 62 | 61 | … | 52 | 51 | 50 | … | 0 |
|---|---|---|---|---|---|---|---|---|
| $M_s$ | $E_1$ | $E_2$ | … | $E_k$ | $M_1$ | $M_2$ | … | $M_n$ |

  - 数符位$M_s$：使用最高位第63位表示；
  - 阶码**E**：11位（第62~52）移码表示；不需要符号位，全部是数值位。
    - 11位阶码采用移码方式来表示正负指数。将浮点数的指数真值 $e$ 变成阶码 $E$ 时，应将指数 $e$ 加上一个固定的偏移值1023，即 $E=e+1023$
  - 尾数**M**：52位（第51~0）原码表示；
    - 采用隐含尾数最高位1的表示方法，实际尾数53位，尾数真值 = 1 + 尾数；
  - 编码表示非0浮点数真值为：$(-1)^{M_s} \times 2^{E-1023} \times (1 + \text{尾数})$



## 2、浮点数的表示方法

- IEEE754几个特殊的编码：
  - **真值零的表示**：当阶码E为全0且尾数M也为全0时的值，符号位S为0或1，表示**正零**或**负零**。**正零**或**负零**的IEEE754编码不相同。
  - **真值为无穷大表示**：当阶码E为全1且尾数M为全0时，符号位S为0或1，表示**+∞**或**-∞**。
  - **NaN的表示**：当阶码E为全1且尾数M为非0时，符号位S为0或1，表示**NaN**→无意义。
- IEEE754表示范围：(表2-1)
  - 在32位浮点数表示中，要除去E用全0和全1（$255_{10}$）表示零、无穷大和的**NaN**特殊情况，对于规格化浮点数，E的范围（**8位数值位0~255**）变为1到254，保证E为正，由于 $E=e+127$，真正的指数值e则为-126到+127，因此32位浮点数表示的绝对值的范围是 $2^{-126} \sim 2^{127} \approx 10^{-38} \sim 10^{38}$。
  - 64位浮点数表示的绝对值的范围是 $2^{-1022} \sim 2^{1023} \approx 10^{-308} \sim 10^{308}$。



## 2、浮点数的表示方法

- 例1 若浮点数x的IEEE754标准存储格式为 $(41360000)_{16}$，求其浮点数的十进制数值。
- 解：将16进制数展开后，可得二制数格式为

**0 100 00010 011 0110 0000 0000 0000 0000**

符号S 阶码E(8位) 尾数M(23位)

- 指数
  - $e=E-127=10000010-01111111=00000011=(3)_{10}$
- 尾数：包括隐藏位1的
  - $1.M=1.011\ 0110\ 0000\ 0000\ 0000\ 0000=1.011011$
- 真值：
  - $(-1)^S \times 1.M \times 2^e = +(1.011011) \times 2^3 = +1011.011 = (11.375)_{10}$

## 2、浮点数的表示方法

- 例2 将数$(20.59375)_{10}$转换成IEEE754标准的32位浮点数的二进制存储格式。
- 解:首先分别将整数和分数部分转换成二进制数:

  **20.59375=10100.10011**
- 按规格化要求移动小数点，使其在第1，2位之间

  **$10100.10011=1.010010011\times2^4$**
- 则:

  **S=0, E=4+127=131, M=010010011**
- M后补零，23位;
- 最后得到32位浮点数的二进制存储格式为:

  0 10000011 01001001100000000000000=$(41A4C000)_{16}$



## 2.1.3 字符和字符串的表示方法

- 字符数据/符号数据：非大小类型的取值。
- 字符数据的机器码可以使用**ASCII/UNICODE**等编码表示。
  - **ASCII码----单字节编码：用一个最高位取值=0的8位二进制数表英文字母、阿拉伯数字、算术运算符、英文标点符号和控制符号，共 128 个字符;**
    - **字符0~9： 30H~39H ;**
    - **字母A~Z： 41H~5AH ;**
    - **字母a~z： 61H~7AH ;**
    - **回车： 0DH ；换行： 0AH ；空格： 20H ;**
  - **UNICODE码----双字节编码：将每个字符和符号赋予一个永久、唯一的16位值，即码点。共65536个码点 ;**
- 字符串：非大小类型的取值。连续的一串字符,保存在连续的一段内存空间中;
  - 例：“**IF A>B THEN READ(C);**”



## 2.1.3 字符和字符串的表示方法

| 低 \ 高 | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 0000 | NUL | DEL | SP | 0 | @ | P | | p |
| 0001 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 0010 | STX | DC2 | “ | 2 | B | R | b | r |
| 0011 | ETX | DC3 | # | 3 | C | S | c | s |
| 0100 | EOT | DC4 | $ | 4 | D | T | d | t |
| 0101 | ENQ | NAK | % | 5 | E | U | e | u |
| 0110 | ACK | SYN | & | 6 | F | V | f | v |
| 0111 | DEL | ETB | | 7 | G | W | g | w |
| 1000 | BS | CAN | ( | 8 | H | X | h | x |
| 1001 | HT | EM | ) | 9 | I | Y | i | y |
| 1010 | LF | SUB | * | : | J | Z | j | z |
| 1011 | VT | ESC | + | ; | K | [ | k | { |
| 1100 | FF | FS | , | < | L | \ | 1 | \| |
| 1101 | CR | GS | - | = | M | ] | m | } |
| 1110 | SO | RS | . | > | N | | n | ~ |
| 1111 | SI | US | / | ? | O | _ | o | DEL |



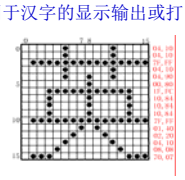
## 2.1.4 汉字的表示方法

- **1. 汉字的输入编码/外码：** 户从键盘上键入汉字时所使用的汉字编码；数字编码/拼音码/字形编码;
  - **数字编码：**常用的是国标区位码，用数字串代表一个汉字输入。区位码是将国家标准局公布的6763个两级汉字分为94个区，每个区分94位，实际上把汉字表示成二维数组，每个汉字在数组中的下标就是区位码。区码和位码各两位十进制数。
  - **拼音码：**拼音码是以汉字拼音为基础的输入方法。
  - **字形编码：**字形编码是用汉字的形状来进行的编码（例：五笔字型）。把汉字的笔划部件用字母或数字进行编码，按笔划的顺序依次输入，就能表示一个汉字。



## 2.1.4 汉字的表示方法

- **汉字交换码：** GB2312-80（6763个）规定一个汉字用两个字节表示，每个字节的最高位值是0 ；与ASCII码一致。
- **2.汉字内码/机器码：**用于汉字信息的存储、交换、检索等操作的机内代码，一般采用**两个字节**表示。
- 编码方式： **GB2312-80编码+8080H**；使汉字交换码中每个字节的最高位值=1，区分汉字机器码和ASCII码;
- **3.汉字字模码/**汉字字形码：为了将汉字在显示器或打印机上输出,把汉字按图形符号设计成点阵图,就得到了相应的点阵代码（字形码）；
  - 字模码构成字库，仅用于汉字的显示输出或打印输出，不能用于机内存储。





## 2.1.5 校验码

- **校验码：** 信息传输和处理过程中受到干扰和故障，容易出错0/1。一般在有效信息中加入一些冗余信息（校验位），使用冗余信息来判断有效信息是否正确。
- **增加校验信息的数据格式：**有效信息+**检验位信息**
  - **奇偶校验：**使用1位检验信息来检查传输的信息中1的个数是奇数个还是偶数个?
    - **奇校验码C的值：**设 $x=(x_0x_1\cdots x_{n-1})$是一个n位有效信息字
    - $C=x_0\oplus x_1\oplus\cdots\oplus x_{n-1}\oplus1$ ，式中⊕代表按位加，当 x 中包含有奇数个1时，C=0。
    - 传输的信息/处理的信息为： $(x_0x_1\cdots x_{n-1}C)$
    - **偶校验码C的值：**设 $x=(x_0x_1\cdots x_{n-1})$是一个n位有效信息字
    - $C=x_0\oplus x_1\oplus\cdots\oplus x_{n-1}$，式中⊕代表按位加，当 x 中包含有偶数个1时，C=0。
    - 传输的信息/处理的信息为： $(x_0x_1\cdots x_{n-1}C)$
- **P10 例10**

## 2.2 定点加法、减法运算

- **机器码编码约定**：如果没有特别申明，运算数据的机器码默认使用补码表示。定点整数。
- **运算约定**：符号位参加运算，运算规律是二进制运算规律。
- **2.2.1 补码加法**：
  - 公式：$[x+y]_{补}=[x]_{补}+[y]_{补}$例11,12
  - [例11] x=+1011，y=+0101，求 x+y=？**注意编码位数！**
  - 解：$[x]_{补}$ = 01001，$[y]_{补}$ = 00101
  - $[x]_{补}$ 0 1 0 0 1
    + $[y]_{补}$ 0 0 1 0 1
    ———————————
    $[x+y]_{补}$ 0 1 1 1 0
    x+y = +1110
  - 如果有进位，进位丢掉**例12**



## 2.2 定点加法、减法运算

- **2.2.2 补码减法**：
  - 公式：$[x-y]_{补}=[x]_{补}-[y]_{补}=[x]_{补}+[-y]_{补}$
  - **变补码**：$[y]_{补}\rightarrow[-y]_{补}$，将$[y]_{补}$连同符号位一起求反加1。
  - 注意区分：已经**y，求$[y]_{补}$和$+[-y]_{补}$。**
  - **例：** y = - 0110，**求$[y]_{补}$和$[-y]_{补}$。**
  - $[y]_{补}$=1.1010 $[-y]_{补}$= 0.0110
  - [例14] x=+1101，y=+0110，求 x-y=？**注意编码位数！**
  - 解：$[x]_{补}$ = 0 1101，$[y]_{补}$ = 0 0110，$[-y]_{补}$ = 1 1010
  - $[x]_{补}$ 0 1 1 0 1
    + $[-y]_{补}$ 1 1 0 1 0
    ———————————
    $[x-y]_{补}$ 1 0 0 1 1 1
    x+y = +0111
  - 如果有进位，进位丢掉



## 2.2 定点加法、减法运算

- **2.2.3 溢出概念和检测方法**：
- **溢出的概念**：两个n位**带符号**2进制表示的数据做ADD/SUB运算，运算的结果也只能使用n位2进制表示。如果运算的结果超过n位2进制的表示范围，运算产生溢出。
  - [例15] x=+1101，y=+1001，求 x+y 。
  - 解：$[x]_{补}$=0 1011，$[y]_{补}$=0 1001
    $[x]_{补}$ 0 1 0 1 1
    + $[y]_{补}$ 0 1 0 0 1
    ———————————
    $[x+y]_{补}$ 1 0 1 0 0
  - 两个正数相加的结果成为负数，一定错误，运算产生溢出，表示正溢。
  - 原因分析？



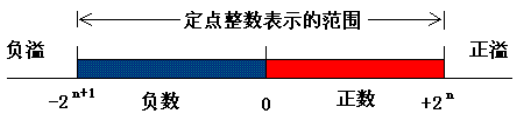
## 2.2 定点加法、减法运算

- **2.2.3 溢出概念和检测方法**：
  - [例16] x=-1101，y=-1011，求 x+y 。
  - 解：$[x]_{补}$=1 0011，$[y]_{补}$=1 0101
    $[x]_{补}$ 1 0 0 1 1
    + $[y]_{补}$ 1 0 1 0 1
    ———————————
    $[x+y]_{补}$ 0 1 0 0 0
  - 两个负数相加的结果成为正数，一定错误，运算产生溢出，表示负溢。
  - 原因分析？



## 2.2 定点加法、减法运算

- **2.2.3 溢出概念和检测方法**：
- **溢出的概念**：在确定了运算的字长和带符号数据的表示方法(编码)后，则能表示的数据范围也就相应决定了。当运算结果超出所能表示的数据范围后，就产生溢出。






## 2.2 定点加法、减法运算

- **2.2.3 溢出概念和检测方法**：
- **可能产生溢出的情况** ：
  - 1）两个相同符号数相加，其运算结果符号与被加数相同，若相反则产生溢出；
  - 2）两个相异符号数相减，其运算结果符号与减数相同，否则产生溢出。
  - 3）相同符号数相减，相异符号数相加不会产生溢出。其他情况没有溢出；
- **溢出的分类**：
  - 两正数加，结果为负数，**正溢**（大于机器所能表示的最大数）
  - 两负数加，结果为正数，**负溢**（小于机器所能表示的最小数）
- **溢出的检测方法**：
  - 双符号位法/变形补码
  - 单符号位法/进位判断法

## 2.2 定点加法、减法运算

- **2.2.3 溢出概念和检测方法：**
- **双符号位法 / 变形补码**：**带符号**运算数据/运算结果 使用补码编码，2位二进制表示符号位，运算结果为00表示正数，11表示负数。其他情况产生溢出。
  - 运算数据编码：$S_{f1}$ $S_{f2}$ +数值位
    - 正数$S_{f1}$ $S_{f2}$ =00：00 +数值位
    - 负数$S_{f1}$ $S_{f2}$ =11 ： 11+数值位
  - 运算结果的符号位$S_{f1}$ $S_{f2}$ =??:
    - 00表示运算结果为正数，无溢出；
    - 01表示运算结果正溢出；
    - 10表示运算结果负溢出；
    - 11表示运算结果为负数，无溢出；
    - **$S_{f1} \oplus S_{f2} = 1$　　异号，有溢出**
    - **$S_{f1} \oplus S_{f2} = 0$　　同号，无溢出**



## 2.2 定点加法、减法运算

- **2.2.3 溢出概念和检测方法：**
- **双符号位法 / 变形补码：**
  - [例17] x=+01100，y=+01000，求 x+y 。
    解：$[x]_{补}$ = 00 1100，$[y]_{补}$ = 00 1000

| | | |
|---|---|---|
| | $[x]_{补}$ | 0 0 1 1 0 0 |
| + | $[y]_{补}$ | 0 0 1 0 0 0 |
| | $[x+y]_{补}$ | 0 1 0 1 0 0　(表示正溢) |

  - [例18] x=-1100，y=-1000，求 x+y 。
    解：$[x]_{补}$ = 11 0100，$[y]_{补}$ = 11 1000

| | | |
|---|---|---|
| | $[x]_{补}$ | 1 1 0 1 0 0 |
| + | $[y]_{补}$ | 1 1 1 0 0 0 |
| | $[x+y]_{补}$ | 1 0 1 1 0 0　(表示负溢) |



## 2.2 定点加法、减法运算

- **2.2.3 溢出概念和检测方法：**
- **单符号位法 / 进位判断法**：带符号运算数据/运算结果使用补码编码，1位二进制表示符号位。
- 带符号运算数据/结果的编码：**$S_f$ +数值位;正数$S_f$ = 0负数 $S_f$ = 1**
  - 运算过程获取符号位$C_f$ 和$C_0$:
    - $C_f$:符号位运算的进位；
    - $C_0$:数值位最高运算的进位；
  - 操作数$A_{补}$ = **$S_A$数值位 = $S_A$ $S_{A1}$ $S_{A2}$ $S_{A3}$ $S_{A4}$……**
  - 操作数$B_{补}$ = **$S_B$数值位 = $S_B$ $S_{B1}$ $S_{B2}$ $S_{B3}$ $S_{B4}$……**
  - $C_f$ $C_0$
  - $(A \pm B)_{补}$ = $S_f$ $S_1$ $S_2$ $S_3$ $S_4$……
    - $C_f$ $C_0$

| $C_f$ | $C_0$ | |
|---|---|---|
| 0 | 0 | 正确（正数） |
| 0 | 1 | 正溢 |
| 1 | 0 | 负溢 |
| 1 | 1 | 正确（负数） |



## 2.2 定点加法、减法运算

- **2.2.3 溢出概念和检测方法：**
- **单符号位法 / 进位判断法：**
  - [例17] x=+01100，y=+01000，求 x+y 。
    解：$[x]_{补}$ = 0 1100，$[y]_{补}$ = 0 1000

| | | |
|---|---|---|
| | $[x]_{补}$ | 0 1 1 0 0 |
| + | $[y]_{补}$ | 0 1 0 0 0 |
| | $[x+y]_{补}$ | 1 0 1 0 0 |

  - $C_f$=0　　$C_0$=1 **正溢出**
  - [例18] x=-1100，y=-1000，求 x+y 。
    解：$[x]_{补}$ = 1 0100，$[y]_{补}$ =1 1000

| | | |
|---|---|---|
| | $[x]_{补}$ | 1 0 1 0 0 |
| + | $[y]_{补}$ | 1 1 0 0 0 |
| | $[x+y]_{补}$ | 0 1 1 0 0 |

  - $C_f$=1　　$C_0$=0　**负溢出**



## 2.2 定点加法、减法运算

- **溢出问题的进一步讨论：**
- **讨论溢出的问题，有一个限制**：带符号数据的运算。只有带符号数据的运算才会有溢出的情况。
- 无符号数据的运算没有溢出？
  - 只要将运算过程产生的进位作为运算结果最高位的取值，一定没溢出。因为运算结果与运算数据比较，多使用1位二进制表示。
  - [分析/讨论] x=-1100，y=-1000，求 |x|+|y|与x+y 。



## 2.2 定点加法、减法运算

- **2.2.4 基本的二进制加法 / 减法器**
- **真值表→卡诺图→最简表达式→电路图→测试：**
- 一位全加器FA真值表：$(C_{i+1}, S_i) = A_i + B_i + C_i$

| 被加数$A_i$ | 加数$B_i$ | 低位的进位$C_i$ | 和$S_i$ | 进位$C_{i+1}$ |
|---|---|---|---|---|
| 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 |

## 2.2 定点加法、减法运算

- **2.2.4 基本的二进制加法 / 减法器**
- **真值表→卡诺图→最简表达式→电路图→测试：**
- 一位全加器FA表达式→电路图（框图）
  - $S_i = A_i \oplus B_i \oplus C_i$　　$C_{i+1} = A_iB_i + B_iC_i + C_iA_i$


FA（全加器）逻辑电路图　　FA框图



## 2.2 定点加法、减法运算

- **2.2.4 基本的二进制加法 / 减法器**
- **真值表→卡诺图→最简表达式→电路图→测试：**
- n位行波进位加法器（图2-3）
- 思考：如何做n位减法运算?如何实现BCD加减运算？

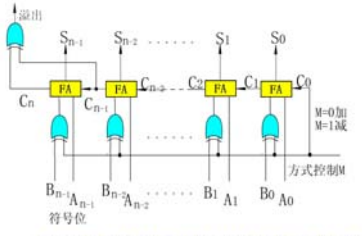



图2.2 行波进位的补码加法减法器



## 2.3 定点乘法运算

- **原码并行乘法运算**：绝对值相乘，符号位单独处理；
  - 先考虑绝对值乘法的实现，在考虑如何计算符号位；
- **补码并行乘法运算**：符号位参加运算；*
- **2.3.1 原码并行乘法**
- **1、人工算法与机器算法的同异性**
  - $[x]_{原} = x_f.x_{n-1}\cdots x_1x_0$　　$[y]_{原} = y_f.y_{n-1}\cdots y_1y_0$
  - $[x.y]_{原} = (x_f \oplus y_f) + (x_{n-1}\cdots x_1x_0).(y_{n-1}\cdots y_1y_0)$



## 2.3 定点乘法运算

- **2.3.1 原码并行乘法**
- **1、人工算法与机器算法的同异性**
  - $A = -0.1101$　　$B = 0.1011$　　$A \times B = -0.10001111$
    - 将n位乘转化为n次“累加与移位”。
    - 从乘数最低位开始，依次求一位乘数对应的部分积，并与原部分积之和作一次累加，然后原部分积之和右移位一次。

```
   0.1101
 × 0.1011
 ------
   1101        ✓ 符号位单独处理，绝对值运算
  1101         ✓ 乘数的一位求一个对应的部分积
 0000          ? 4个部分积依次累加，移位
1101           ✓ 乘积的位数扩大一倍
-------------
0.10001111
```



## 2.3 定点乘法运算

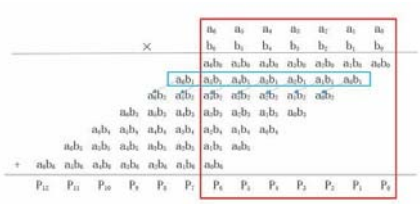
- **2.3.1 原码并行乘法**
- **1、人工算法与机器算法的同异性**
  - 1)机器通常只有n位长，运算器也是n位长，而两个n位二进制数相乘，乘积可能为2n位。
  - 2)加法器每次只能进行两个操作数的相加，难以胜任将n个位积同时相加起来的运算。
  - 解决办法：早期计算机中为了简化硬件结构,采用串行的1位乘法方案,即多次执行“加法—移位”操作来实现。然而串行方法速度太慢。
  - 更好方法：随着大规模集成电路问世以来,出现了各种形式的流水式阵列乘法器,它们属于**并行乘法器**。
  - 理解：求和的方式：
    - 从低位到高位1列1列依次求和。**串行方式**
    - 从上面到下面每行数据依次进行自己的求和运算。**并行方式**



## 2.3 定点乘法运算

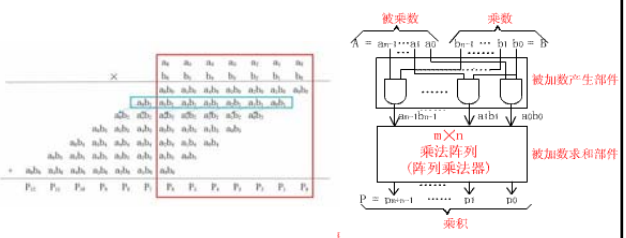
- **2.3.1 原码并行乘法**
- **2、不带符号的阵列乘法器 / / 绝对值乘 / 原码输入输出**
- 设有两个无符号的二进制整数：
  - **(m位)** $A = a_{m-1}\cdots a_1a_0$　　**其数值** $= \sum_{i=0}^{m-1} a_i 2^i$
  - **(n位)** $B = b_{n-1}\cdots b_1b_0$　　**其数值** $= \sum_{j=0}^{n-1} b_j 2^j$
  - **(m+n位)乘积** $P = A \times B = (a_{m-1}\cdots a_1a_0) \times (b_{n-1}\cdots b_1b_0)$

**其数值** $= \left(\sum_{i=0}^{m-1} a_i 2^i\right) \times \left(\sum_{j=0}^{n-1} b_j 2^j\right) = \sum_{k=0}^{m+n-1} p_k 2^k$



## 2.3 定点乘法运算

- **2.3.1 原码并行乘法**
- **2、不带符号的阵列乘法器 / / 绝对值乘 / 原码输入输出**
  - 在$m$位乘$n$位不带符号整数的阵列乘法中,每一个部分乘积项(**位积**)$a_ib_j$叫做一个被加数。这$m\times n$个被加数$\{a_ib_j|0\leqslant i\leqslant m-1$和$0\leqslant j\leqslant n-1\}$可以用$m\times n$个"与"门并行地产生。
  - 实现累加与移位的电路：**阵列乘法器**

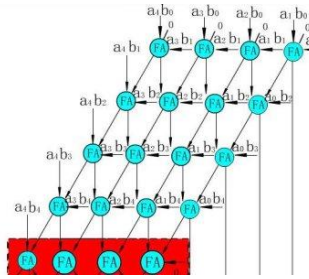







## 2.3 定点乘法运算

- **2.3.1 原码并行乘法**
- **2、不带符号的阵列乘法器 / / 绝对值乘 / 原码输入输出**
  - 5×5的**阵列乘法器**
    - ***n*位×*n*位时,需要$n(n-1)$个"全加器"和$n^2$个"与"门。**



虚线围住的阵列中最后一行构成一个行波进位加法器，其时间延迟为（n-1）*2T。这种乘法器要实现n位*n位时，需要n*（n-1）个全加器和n*n个与门




## 2.3 定点乘法运算

- **2.3.1 原码并行乘法**
- **例**理解**阵列乘法器**实现相加和移位的过程：

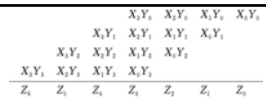
设 $X=X_3X_2X_1X_0$，$Y=Y_3Y_2Y_1Y_0$，计算 $X\times Y$，列式如下：

$$
\begin{array}{cccccccc}
 & & & X_3 & X_2 & X_1 & X_0 & \\
 & & \times & Y_3 & Y_2 & Y_1 & Y_0 & \\
\hline
 & & & X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 & \\
 & & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 & & \\
 & X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 & & & \\
X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3 & & & & \\
\hline
Z_6 & Z_5 & Z_4 & Z_3 & Z_2 & Z_1 & Z_0 &
\end{array}
$$

- 其中 $X_iY_j$ 是与运算，而结果$Z_i$则是与结果的求和。



- **并行加过程分析：6次加**



  - $(Z_0)=X_0Y_0$
  - 第1次加：$Z_1$
    - 第1行每个数据与第2行对应位置数据加，进位输入全为0。共3个加，并行进行：

      $(C_1^1, Z_1)= X_1Y_0+X_0Y_1+0$，　$(C_2^1, Z_2^1)= X_2Y_0+X_1Y_1+0$，

      $(C_3^1, Z_3^1)= X_3Y_0+X_2Y_1+0$
  - 第2次加：$Z_2$
    - 第1次加的2个中间结果及第2行没加的数据与与第3行对应位置数据加，进位输入来自低位上次低位加的进位输出。共3个加，并行进行：

      $(C_2^2, Z_2)= Z_2^1+X_0Y_2+C_1^1$，　$(C_3^2, Z_3^2)= Z_3^1+X_1Y_2+C_2^1$，

      $(C_4^2, Z_4^2)= X_3Y_1+X_2Y_2+C_3^1$
  - 第3次加：$Z_3$
    - 第2次加的2个中间结果及第3行没加的数据与与第4行对应位置数据加，进位输入来自低位上次低位加的进位输出。共3个加，并行进行：

      $(C_3^3, Z_3)= Z_3^2+X_0Y_3+C_2^2$，　$(C_4^3, Z_4^3)= Z_4^2+X_1Y_3+C_3^2$，

      $(C_5^3, Z_5^3)= X_3Y_2+X_2Y_3+C_4^2$
  - 第4次加：3个串行加 $(C_4^4, Z_4)= Z_4^3+0+C_3^3$，$(C_5^4, Z_5)= Z_5^3+C_4^4+C_4^3$，$(C_6^4, Z_6)=X_3Y_3+C_5^4+C_5^3$，$Z_7=C_6^4$



## 2.3 定点乘法运算

- **2.3.1 原码并行乘法**
- **2、不带符号的阵列乘法器 / / 绝对值乘 / 原码输入输出**
  - 例19 **阵列乘法器**参见图2.5，已知不带符号的二进制整数A=11011，B=10101，求每一部分乘积项$a_ib_j$的值与$p_9p_8\ldots p_0$的值。
  - 解：

$$
\begin{array}{rl}
 & 11011 = A\ (27_{10}) \\
\times & 10101 = B\ (21_{10}) \\
\hline
 & 11011 \quad a_4b_0=1,\ a_3b_0=1,\ a_2b_0=0,\ a_1b_0=1,\ a_0b_0=1 \\
 & 00000 \quad a_4b_1=0,\ a_3b_1=0,\ a_2b_1=0,\ a_1b_1=0,\ a_0b_1=0 \\
 & 11011 \quad a_4b_2=1,\ a_3b_2=1,\ a_2b_2=0,\ a_1b_2=1,\ a_0b_2=1 \\
 & 00000 \quad a_4b_3=0,\ a_3b_3=0,\ a_2b_3=0,\ a_1b_3=0,\ a_0b_3=0 \\
+ & 11011 \quad a_4b_4=1,\ a_3b_4=1,\ a_2b_4=0,\ a_1b_4=1,\ a_0b_4=1 \\
\hline
 & 1000110111 = P
\end{array}
$$

$P = p_9p_8p_7p_6p_5p_4p_3p_2p_1p_0 = 1000110111\ (567_{10})$



## 2.3 定点乘法运算

- **2.3.1 原码并行乘法**
- **3、带符号的阵列乘法器 / / 绝对值乘 / 补码输入输出**
  - **原理**：算前求补(**求绝对值**)－乘法器(**绝对值乘**)－算后求补(**还原为带符号数**)
- **求补电路**的原理//**注意，不是求补码**
  - **正数求补算法**：数据给位取值不变，求补前和求补后相同；
  - **负数求补算法**：从数据的最右边开始向左边逐位扫描，找到第一个"1"为止。该"1"的左边各位全部取反；该"1"的右边各位（包括该"1"）保持不变。
  - **例：X=+1110 $[X]_{求补}$=1110　　Y=-0100 $[Y]_{求补}$=1100**

## 2.3 定点乘法运算

- **2.3.1 原码并行乘法**
- **3、带符号的阵列乘法器 / / 绝对值乘 / 补码输入输出**
  - **求补电路**：
    - **E＝0**时，正数求补，输入和输出相等；**E＝1**时，负数求补，则从数最右端往左边扫描，直到第一个1的时候，该位和右边各位保持不变，左边各数值位按位取反；
    - **E的输入**：带符号数据符号位取值；

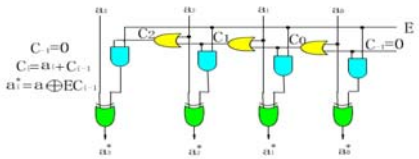



图2.6 对2求补器电路图



## 2.3 定点乘法运算

- **2.3.1 原码并行乘法**
- **3、带符号的阵列乘法器 / / 绝对值乘 / 补码输入输出**
  - **(n+1) ×(n+1)带求补器的无符号阵列乘法器**
    - **实现原码乘法：不用求补器**
    - **实现补码乘法：启用求补器**

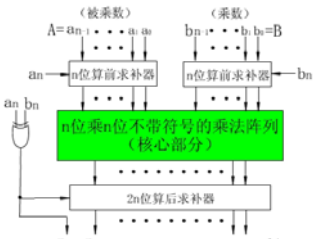



图2.7 （n+1）位乘（n+1）位带补级的阵列乘法器



## 2.3 定点乘法运算

- **2.3.1 原码并行乘法**
- **3、带符号的阵列乘法器 / / 绝对值乘 / 补码输入输出**
  - **实现补码乘法原理：**
    - 两个补码相乘，先用求补器将补码转化为原码/**绝对值**。
    - 然后将数值部分使用不带符号的阵列乘法器求出**乘积的绝对值**；符号位单独处理。
    - 再根据乘积的符号位对乘积的绝对值求补，得出**乘积的补码**。是一种**间接补码乘法！**
    - **2个算前求补器**：将两个操作数*A*和*B*的**数值部分**，在相乘以前，先变成绝对值（**原码**）。
    - **1个算后求补器**：把运算结果变为补码。**（当两个输入操作数A、B的符号不一致时，启动求补器）**



## 2.3 定点乘法运算

- [例20] 设x=+15，y=-13，用带求补器的**原码阵列乘法器**求出乘积x·y=？**这个例题有问题**？
  解：**原码阵列乘法器**：所有运算数据原码表示 / **原码输入，原码输出。**
  - **[x]原=0 1111 , [y]原=1 1101**
  - **算前求补器输出： |x|= 1111 ,|y|= 1101**
  - **绝对值乘：**

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

  - **符号位运算**：0⊕1=1， 输出**[x×y]原=1 11000011**
  - 真值 **x·y =15×（-13）= $(-11000011)_2 = (-195)_{10}$**



## 2.3 定点乘法运算

- [例21]设x=-15，y=-13，用带求补器的**补码阵列乘法器**求出乘积x·y=？对结果进行验证。
  解：**补码阵列乘法器**：运算数据补码表示 / **补码输入，补码输出。**
  - **[x]原=1 1111 , [x]补=1 0001　　[y]原=1 1101, [y]补=1 0011**
  - 输入**[x]补[y]补**，算前求补器输出： **|x|=1111 , |y|=1101**
  - **绝对值乘：**

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

  - **符号位运算**：1⊕1=0 ， 算后求补器输出**11000011**，
  - 输出**[x×y]补=0 11000011**
  - 真值 **x·y = （-15）×（-13）= $(+11000011)_2 = (+195)_{10}$**



## 2.3 定点乘法运算

- **2.3.2 直接补码并行乘法**
- **1、补码与真值的转换公式**
  - **带符号阵列乘法器**：阵列乘法器求**乘积的绝对值**；符号位单独处理。求解过程涉及求补电路，运算速度慢。
  - 考虑设计直接使用补码的乘法器，符号位直接参与运算过程。
  - ***[N]补=$a_n a_{n-1}\cdots\cdots a_2 a_1 a_0$***，其中$a_n$是符号位。
  - 其对应的大小为/**真值*N***；

$$N=\begin{cases} +\sum_{i=0}^{n-1} a_i 2^i, a_n=0, \text{正数} \\ -[1+\sum_{i=0}^{n-1}(1-a_i)2^i], a_n=1, \text{负数} \end{cases}$$

***$a_i$位取反：1-$a_i$***

## 2.3 定点乘法运算

- **2.3.2 直接补码并行乘法**
- **1、补码与真值的转换公式**
  - $[N]_{补}=a_n a_{n-1}\cdots\cdots a_2 a_1 a_0$，其中$a_n$是符号位。
  - 统一转换形式：

$$N=-a_n\times 2^n+\sum_{i=0}^{n-1}a_i 2^i$$

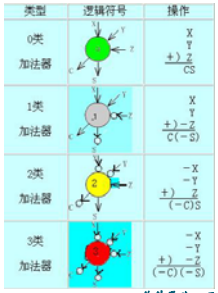
  - 根据 $\bar{a}_i=1-a_i$取反
  - $[-N]_{补}=\bar{a}_n\bar{a}_{n-1}\cdots\cdots\bar{a}_2\bar{a}_1\bar{a}_0+1$，其中$a_n$是符号位。**变补码**

$$-N=-(1-a_n)\times 2^n+\sum_{i=0}^{n-1}(1-a_i)2^i+1$$



## 2.3 定点乘法运算

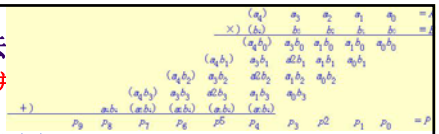
- **2.3.2 直接补码并行乘法**
- 例22、已知： $[N1]_{补}=01101$， $[N2]_{补}=10011$，求真值。
  - $[N1]_{补}=01101=-0\times2^4+1\times2^3+1\times2^2+0\times2^1+1\times2^0=+13$
  - $[N2]_{补}=01101=-1\times2^4+0\times2^3+0\times2^2+1\times2^1+1\times2^0=-13$
- **2、一般化的全加器形式**
- 4类全加器 P37 表2.4

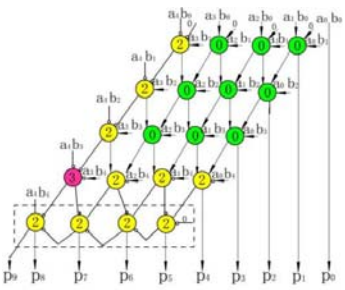





## 2.3 定点乘法

- **2.3.2 直接补码并**
- **3、直接补码阵列乘法器**







## 2.4 定点除法运算

- **2.4.1 原码除法算法原理**
- **基本原理：**运算数据使用原码表示，**商的符号**由两数的符号位"**异或**"得到，**商的数值**由两数的绝对值相除求得。
- 设有$n$位定点小数/定点整数：
  - **被除数x：** $[x]_{原}=x_f.x_{n-1}\cdots x_1x_0$
  - **除数y：** $[y]_{原}=y_f.y_{n-1}\cdots y_1y_0$
  - **商$q=x/y$：**
  - $[q]_{原}=(x_f\oplus y_f)+(0.x_{n-1}\cdots x_1x_0\div 0.y_{n-1}\cdots y_1y_0)$
  - 讨论：一般要求被除数，除数和商机器码使用相同的表示方式，则
    - **两个纯整数**除法：商为纯整数，被除数位数是除数位数的1倍，被除数高一半小于除数。
    - **两个纯小数**除法：被除数位数与除数位数相等或双倍长，被除数高一半小于除数。



## 2.4 定点除法运算

- **2.4.1 原码除法算法原理**
  - **求商：**被除数与除数原码/绝对值，符号位不参与运算，使用2进制运算规律求商。(近似商/余数)
  - 例：被除数$x=0.1001$，除数$y=0.1011$，以**手算**方法求$x\div y$；

| | | （商q）//解释 |
|---|---|---|
| | 0.1 1 0 1 | |
| 0.1 0 1 1 ) | 0.1 0 0 1 0 | x:除数大于被除数,商0，后补0得余数r0 |
| − | 0.0 1 0 1 1 | $2^{-1}y$: 除数右移1位小于r0, r0减除数,商1 |
| | 0.0 0 1 1 1 0 | r1: 减结果后补0得余数r1 |
| − | 0.0 0 1 0 1 1 | $2^{-2}y$: 除数右移2位小于r1, r1减除数,商1 |
| | 0.0 0 0 0 1 1 0 | r2:减结果后补0得余数r2 |
| − | 0.0 0 0 1 0 1 1 | $2^{-3}y$: 除数右移3位大于r2, r2减0,商0 |
| | 0.0 0 0 0 1 1 0 0 | r3:减结果后补0得余数r3 |
| − | 0.0 0 0 0 1 0 1 1 | $2^{-4}y$: 除数右移4位小于r4, r1减除数,商1 |
| | 0.0 0 0 0 0 0 0 1 | r4:减结果 (最后1步不补0)得余数r4 |



## 2.4 定点除法运算

- **2.4.1 原码除法算法原理**
  - **计算机如何商0商1：**人工除法时，人可以比较**被除数（余数）**和**除数**的大小来确定商1（**够减**）或商0（**不够减**）
  - 机器除法时，先做减法运算得到余数，**余数为正表示够减，商1；余数为负表示不够减，商0**；不够减时必须恢复原来余数，才能继续向下运算。这种方法叫**恢复余数法**，控制比较复杂。
- 不恢复余数法（**加减交替法**）：控制简单，有规律。

## 2.4.1 原码除法算法原理

- 例：x=0.101001,y=-0.111，用加减交替法求x ÷ y。
  - 说明：被除数位数一般是除数位数1倍，商位数与除数相同；余数符号与被除数相同；
- 解：求商符号位$x_f$=0，$y_f$=1，$x_f \oplus y_f = 0 \oplus 1 = 1$
- |x|=0.101001，|y|=0.111，$|y|_{补}$=0.111，$-|y|_{补}$=1.001

| | | |
|---|---|---|
| 被除数x | 0.1 0 1 0 0 1 | 第1次被除数减y |
| (加$-|y|_{补}$) | 1.0 0 1 | 差~余数 |
| 余数为负 | 1.1 1 0 0 0 1 ＜0 | 上商$q_0$=0，下次加 |
| $|y|_{补}$右移1位 | 0.0 1 1 1 | |
| (加$|y|_{补}$) | | |
| 余数为正 | 0.0 0 1 1 0 1 ＞0 | 上商$q_1$=1，下次减 |
| $-|y|_{补}$右移2位 | 1.1 1 0 0 1 | |
| (加$-|y|_{补}$) | | |
| 余数为负 | 1.1 1 1 1 1 1 ＜0 | 上商$q_2$=0，下次加 |
| $|y|_{补}$右移3位 | 0.0 0 0 1 1 1 | |
| (加$|y|_{补}$) | | |
| 余数为正 | 0.0 0 0 1 1 0 ＞0 | 上商$q_3$=1 |

商 $|q|=q_0.q_1q_2q_3=0.101$ 余数 r=0.000110 即：x÷y= - 0.101

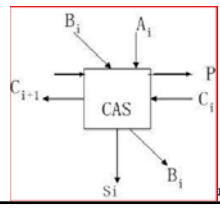


## 2.4 定点除法运算

- 与阵列乘法器相似，**阵列式除法器**也是一种并行运算部件，采用大规模集成电路制造，能提供令人满意的高速运算。
- 阵列除法器有多种多样形式，如**不恢复余数阵列除法器，补码阵列除法器**等。

### 2.4.2 并行除法器

- 使用不恢复余数(加减交替)法
- 1、可控加法/减法(CAS)单元
  - CAS用于并行除法流水逻辑阵列中，它有四个输出端和四个输入端。
  - 输入线P＝0时，CAS作加法运算；
  - 输入线P＝1时，CAS作减法运算。

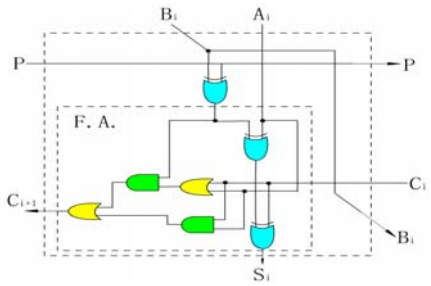





## 2.4 定点除法运算

### 2.4.2 并行除法器

- 1、可控加法/减法(CAS)单元
  - 4个输入端：$A_i$,$B_i$,$C_i$,P
  - 4个输出端：$B_i$,P,$S_i$,$C_{i+1}$






## 2.4 定点除法运算

### 2.4.2 并行除法器

- 1、可控加法/减法(CAS)单元
  - $S_i=A_i \oplus (B_i \oplus P) \oplus C_i$
  - $C_{i+1}=(A_i+C_i)\cdot(B_i \oplus P)+A_iC_i$
  - P＝0，**加法运算**
    - $S_i=A_i \oplus B_i \oplus C_i$
    - $C_{i+1}=A_iB_i+B_iC_i+A_iC_i$
  - P＝1，**减法运算**
    - $B_i^*=B_i \oplus 1$
    - $S_i=A_i \oplus B_i^* \oplus C_i$
    - $C_{i+1}=A_iB_i^* + B_i^* C_i + A_iC_i$



## 2.4 定点除法运算

### 2.4.2 并行除法器

- 2、不恢复余数法/加减交替法阵列除法器
  - **除法规则**：被除数x，除数y
  - 被除数双倍长，绝对值运算|x|÷|y|，余数符号与被除数同号，商的位数与除数相同。

    1)首先做（x-y）运算，即：$x+[-|y|]_{补}$；得差~余数

    2)判断余数符号：
    - 若余数为正（**够减**），则：商上“1”，余数左移一位/除数右移一位，然后做**减除数（+[-y]补）运算→余数**；
    - 若余数为负（**不够减**），则：商上“0”；余数左移一位/除数右移一位，然后做**加除数（+[y]补）运算→余数**。
  - 循环2)，直到整除或求出需要位数的商；
  - 说明：计算机中，小数点是固定的/纯整&纯小;不能简单地采用手算的办法。为便于机器操作，计算机常采用 “余数左移”的方法来替代“除数右移”。（二者等价）



## 2.4.1 原码除法算法原理

- 例23：x=+0.101001,y=+0.111，用加减交替法求x ÷ y。
  - 说明：被除数位数一般是除数位数1倍，商位数与除数相同；余数符号与被除数相同；
- 解：求商符号位$x_f$=0，$y_f$=0，$x_f \oplus y_f = 0 \oplus 0 = 0$
- |x|=0.101001，|y|=0.111，$|y|_{补}$=0.111，$-|y|_{补}$=1.001

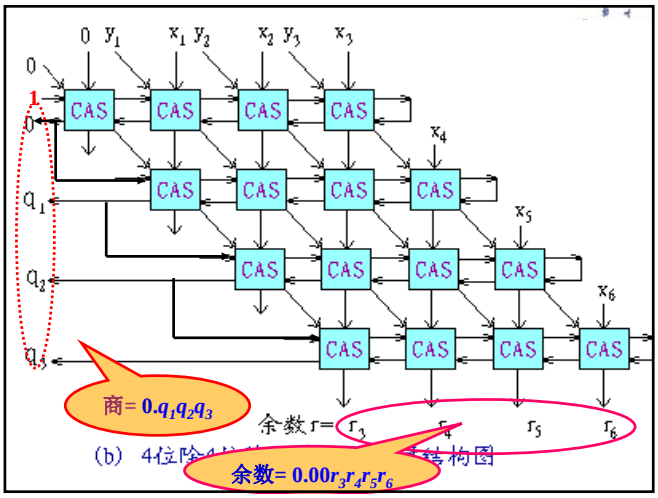
| | | |
|---|---|---|
| 被除数x | 0.1 0 1 0 0 1 | 被除数减y第1次减y |
| (加$-|y|_{补}$) | 1.0 0 1 | |
| 余数为负 | 1.1 1 0 0 0 1 ＜0 | 上商$q_0$=0，下次加y |
| 余数左移 | 1.1 0 0 0 1 | |
| (加$|y|_{补}$) | 0.1 1 1 | |
| 余数为正 | 0.0 1 1 0 1 ＞0 | 上商$q_1$=1，下次减y |
| 余数左移 | 0.1 1 0 1 | |
| (加$-|y|_{补}$) | 1.0 0 1 | |
| 余数为负 | 1.1 1 1 1 ＜0 | 上商$q_2$=0，下次加y |
| 余数左移 | 1.1 1 1 | |
| (加$|y|_{补}$) | 0.1 1 1 | |
| 余数为正 | 0.1 1 0 ＞0 | 上商$q_3$=1 |

商 $|q|=q_0.q_1q_2q_3=0.101$ 余数 r=0.000110 即：x÷y= + 0.101

## 2.4 定点除法运算

- **2.4.2 并行除法器**
- 2、不恢复余数法/加减交替法阵列除法器
  - 被除数 $x=0.x1x2x3x4x5x6$ （双倍长）
  - 除数 $y=0.y1y2y3$
  - 商数 $q=0.q1q2q3$
  - 余数 $r=0.00r3r4r5r6$
  - 字长 n+1=4 **多一位0**
- 则：（4÷4）位阵列除法器逻辑结构见后图:
  - 注意与书上下标的不同。
  - 使用除数右移。
  - 一个(n+1)位除(n+1)位的阵列除法器由$(n+1)^2$个CAS单元组成。






(b) 4位除4位的阵列结构图

## 2.4 定点除法运算

- **2.4.2 并行除法器**
- 2、不恢复余数法/加减交替法阵列除法器
- 输入输出说明：
  - 被除数 $x=0.x_1x_2x_3x_4x_5x_6$ 由顶部一行和最右边的对角线上的垂直输入线来提供的。
  - 除数 $y=0.y_1y_2y_3$ 沿顶部一行对角线方向进入这个阵列。
  - 阵列除法器移位：余数保持固定，而将除数沿对角线右移。
  - 商 $q=0.q_1q_2q_3$ 在阵列的左边产生。
  - 余数 $r=0.00r_3r_4r_5r_6$ 在阵列的最下一行产生。



## 2.4 定点除法运算

- **2.4.2 并行除法器**
- 2、不恢复余数法/加减交替法阵列除法器
- 运算说明：
  - **第一行**所执行的初始操作是减法(P=1)，因此，最上面一行的控制线P固定置成“1”。
  - 减法是+$[-y]_{补}$的运算来实现。这时右端各CAS单元上的反馈线用作**初始的进位输入**。
  - **每一行最左边的单元**的进位输出（**够减为1、不够减为0**）决定着商的数值。将当前的商反馈到下一行，就能确定下一行的操作：**1→减 0→加**。



## 2.5 定点运算器的组成

- **2.5.1 逻辑运算**
- **2.5.2 多功能算术逻辑运算单元**
- **2.5.3 内部总线**
- **2.5.4 定点运算器的基本结构**



## 2.5.1 逻辑运算

- 运算器是数据的加工处理部件，是CPU的重要组成部分。
- 运算器基本结构：
  - 算术运算单元、逻辑运算单元、数据缓冲寄存器、通用寄存器、多路转换器和数据总线等逻辑构件。
- **2.5.1 逻辑运算**
  - 运算数据：**逻辑数**,是指不带符号的二进制数，是非数值型数据，表示的是计算机所要处理的一些状态信息。
  - 基本运算：**逻辑非、逻辑加、逻辑乘、逻辑异**四种。
  - 所有的逻辑运算都是**位运算**，对应位置的二进制数位进行运算。

## 2.5.1 逻辑运算

- **2.5.1 逻辑运算**
  - **逻辑非**：**求反运算**,单目运算符；对某数进行按位求反的操作(**0→1,1→0**),在变量上方加一横线表示逻辑非运算。
    - 例：$x1=01001011 \rightarrow \overline{x1}=10110100$
      $x2=11110000 \rightarrow \overline{x2}=00001111$
  - **逻辑加**：**或运算**,双目运算符；对两数进行按位求或的操作(**0+0=0,1+x=1, 0∨0=0,1∨x=1 x任意**)。
    - 例：$x1=01001011 \quad x2=11110000$
      $x1+x2=01001011+11110000=11111011$
  - **逻辑乘**：**与运算**,双目运算符；对两数进行按位求与的操作(**1 · 1=1,0 · x=0, 1∧1=1,0∧x=0 x任意**)。
    - 例：$x1=01001011 \quad x2=11110000$
      $x1 \cdot x2=01001011 \cdot 11110000=01000000$



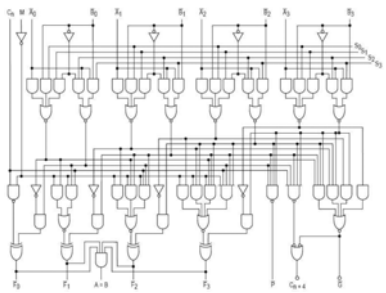
## 2.5.1 逻辑运算

- **2.5.1 逻辑运算**
  - **逻辑异**：**异或运算**,双目运算符；对两数进行按位求异或的操作(**1⊕1=0,0⊕0=0, 1⊕0=1,0⊕1=1 按位加运算**)。
    - 例：$x1=01001011 \quad x2=11110000$
      $x1 \oplus x2=01001011 \oplus 11110000=10111011$
  - **逻辑表达式**：使用逻辑运算实现的表达式。
  - 注意：
    - 运算数据是逻辑数；
    - 逻辑运算都是**本位运算**，不存在进位和借位问题。



## 2.5.2 多功能算术/逻辑运算单元ALU

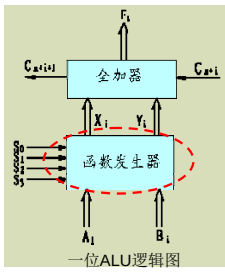
- **使用全加器设计ALU要解决的问题：**
  - **多种运算形式的集成**：实现16种算术运算，16种逻辑运算
  - **运算速度的问题**：采用并行运算操作，**先行进位**的思想





## 2.5.2 多功能算术/逻辑运算单元ALU

- **1.基本思想**
  - 一位全加器FA的逻辑表达式：
    - $F_i=A_i \oplus B_i \oplus C_i$
    - $C_{i+1}=A_iB_i+B_iC_i+C_iA_i$
  - 为了实现多种运算和并行操作，将输入$A_i$、$B_i$进行组合，得出不同的组合函数$X_i$和$Y_i$，作为全加器的新输入，进而实现多种 算术运算和逻辑运算，同时解决先行进位。
  - 一位算术/逻辑运算单元的逻辑表达式为：
    - $F_i=X_i \oplus Y_i \oplus C_{n+i}$
    - $C_{n+i+1}=X_iY_i+Y_iC_{n+i}+C_{n+i}X_i$
    - 下标$n$代表芯片总位号，$i$为芯片上的位号。若$ALU$为4位一片,则：$i=0,1,2,3$。
    - 若干片$ALU$芯片可以组成更大字长的运算器。例如，当4片$ALU$组成16位字长的运算器时,$n=0, 4, 8, 12$。



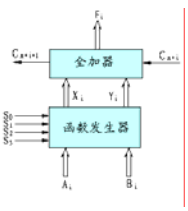
一位ALU逻辑图



## 2.5.2 多功能算术/逻辑运算单元ALU

- **2.逻辑表达式**
  - $Y_i$是受“$S_0S_1$”控制的$A_i$和$B_i$的组合函数，$X_i$是受“$S_2S_3$”控制的$A_i$和$B_i$组合函数，各控制组合关系如下表所示(教材P46)：

| $S_0\ S_1$ | $Y_i$ | $S_2\ S_3$ | $X_i$ |
|---|---|---|---|
| 0 0 | $\overline{A_i}$ | 0 0 | 1 |
| 0 1 | $\overline{A_i}B_i$ | 0 1 | $\overline{A_i}+B_i$ |
| 1 0 | $\overline{A_i}\overline{B_i}$ | 1 0 | $\overline{A_i}+\overline{B_i}$ |
| 1 1 | 0 | 1 1 | $\overline{A_i}$ |

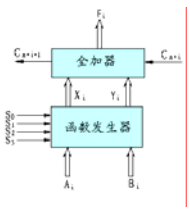



$$Y_i=\overline{S_0}\overline{S_1}\overline{A_i}+\overline{S_0}S_1\overline{A}B_i+S_0\overline{S_1}\overline{A_i}\overline{B_i}$$
$$X_i=\overline{S_2}\overline{S_3}+\overline{S_2}S_3(\overline{A_i}+\overline{B_i})+\overline{S_2}S_3(\overline{A_i}+B_i)+S_2S_3\overline{A_i}$$



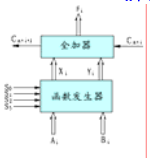
## 2.5.2 多功能算术/逻辑运算单元ALU

- **2.逻辑表达式**
  - **ALU**的第i位逻辑表达式**(2.31)**：
  - $X_i=S_3A_iB_i+S_2A_i\overline{B_i}$
  - $Y_i=A_i+S_0B_i+S_1\overline{B_i}$
  - $F_i=X_i \oplus Y_i \oplus C_{n+i}$
  - $C_{n+i+1}=Y_i+X_iC_{n+i}$



## 2.5.2 多功能算术/逻辑运算单元ALU

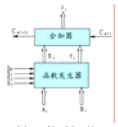
- **2.逻辑表达式**
  - **ALU**的**先行进位**设计(4位FA,$C_n$是向第0位（末位）的进位）：
  - 第0位向第1位的进位公式为：（i=0）
    - $C_{n+1}=Y_0+X_0C_n$
  - 第1位向第2位的进位公式为：（i=1）
    - $C_{n+2}=Y_1+X_1C_{n+1}=Y_1+Y_0X_1+X_0X_1C_n$
  - 第2位向第3位的进位公式为：（i=2）
    - $C_{n+3}=Y_2+X_2C_{n+2}=Y_2+Y_1X_1+Y_0X_1X_2+X_0X_1X_2C_n$
  - 第3位的进位输出（即整个4位运算进位输出）公式为：（i=3）
    - $C_{n+4}=Y_3+X_3C_{n+3}$
      $=Y_3+Y_2X_3+Y_1X_2X_3+Y_0X_1X_2X_3+X_0X_1X_2X_3C_n$






## 2.5.2 多功能算术/逻辑运算单元ALU

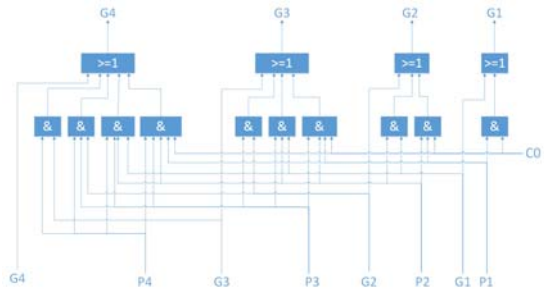
- **2.逻辑表达式**
  - **ALU**的**先行进位**设计(4位FA,$C_n$是向第0位（末位）的进位）：
  - **进位发生输出**G：
    - $G=Y_3+Y_2X_3+Y_1X_2X_3+Y_0X_1X_2X_3$
  - **进位传送输出**P：
    - $P=X_0X_1X_2X_3$
  - 则：$C_{n+4}=G+PC_n$
  - $C_{n+4}$是本片(组)的最高进位输出。逻辑表达式表明： 第0位的进位输入$C_n$可以直接传送到最高位上去，这是一个先行进位逻辑，因而可以实现**高速运算**。
  - 增加P和G的目的在于实现多片（组）ALU之间的先行进位，需要配合电路，称为先行进位发生器（CLA）






## 2.5.2 多功能算术/逻辑运算单元ALU

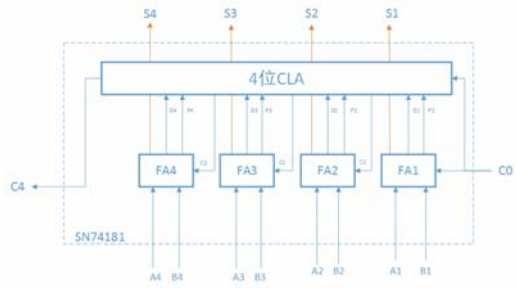
- **2.逻辑表达式**
  - **ALU**的**先行进位**设计(4位FA,$C_n$是向第0位（末位）的进位）：






## 2.5.2 多功能算术/逻辑运算单元ALU

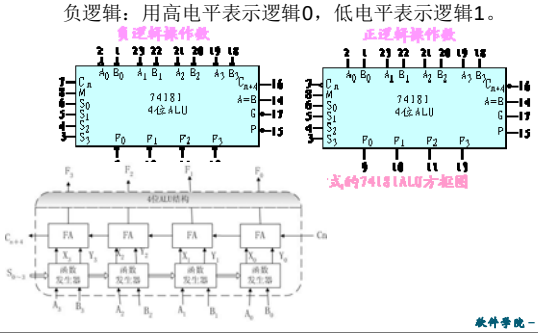
- **3.算术逻辑运算的实现**
  - **74181ALU**(4位算术/逻辑运算器)：先行进位






## 2.5.2 多功能算术/逻辑运算单元ALU

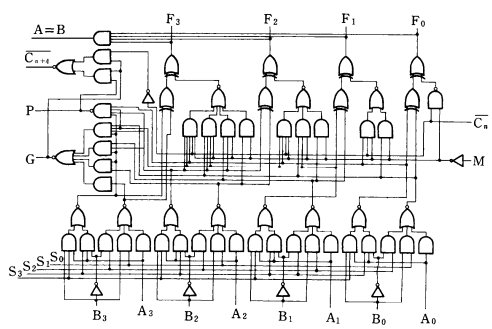
- **3.算术逻辑运算的实现**
  - **74181ALU**(4位算术/逻辑运算器)：
  - 正逻辑：用高电平表示逻辑1，低电平表示逻辑0。
    负逻辑：用高电平表示逻辑0，低电平表示逻辑1。






## 2.5.2 多功能算术/逻辑运算单元ALU

- **3.算术逻辑运算的实现**
  - **74181ALU**(4位算术/逻辑运算器) **正逻辑**：



## 2.5.2 多功能算术/逻辑运算单元ALU

- **3.算术逻辑运算的实现**
  - **74181ALU**(4位算术/逻辑运算器)**正逻辑**：
  - 控制端M的作用：
  - ***M*=0/L**时，进行**算术操作**。***M*=1/H**时，进行**逻辑操作**。

| 工作方式选择输出 $S_3S_2S_1S_0$ | 负逻辑输入与输出 逻辑 ($M$=H) | 负逻辑输入与输出 算术运算 ($M$=L)($C_n$=L) | 正逻辑输入与输出 逻辑 ($M$=H) | 正逻辑输入与输出 算术运算 ($M$=L)($C_n$=H) |
|---|---|---|---|---|
| L L L L | $\overline{A}$ | A减1 | $\overline{A}$ | A |
| L L L H | $\overline{AB}$ | AB减1 | $\overline{A+B}$ | A+B |
| L L H L | $\overline{A}+B$ | $A\overline{B}$减1 | $\overline{A}B$ | A+$\overline{B}$ |
| L L H H | 逻辑1 | 减1 | 逻辑0 | 减1 |
| L H L L | $\overline{A+B}$ | A加$(A+\overline{B})$ | $A\overline{B}$ | A加$A\overline{B}$ |
| L H L H | $\overline{B}$ | AB加$(A+\overline{B})$ | $\overline{B}$ | $(A+B)$加$A\overline{B}$ |
| L H H L | $\overline{A\oplus B}$ | A减B减1 | $A\oplus B$ | A减B减1 |
| L H H H | $A+\overline{B}$ | $A+\overline{B}$ | $A\overline{B}$ | $A\overline{B}$减1 |
| H L L L | $\overline{AB}$ | A加$(A+B)$ | $\overline{A}+B$ | A加AB |
| H L L H | $A\oplus B$ | A加B | $\overline{A\oplus B}$ | A加B |
| H L H L | B | $A\overline{B}$加$(A+B)$ | B | $(A+\overline{B})$加AB |
| H L H H | A+B | A+B | AB | AB减1 |
| H H L L | 逻辑0 | A加A* | 逻辑1 | A加A* |
| H H L H | $A\overline{B}$ | AB加A | $A+\overline{B}$ | $(A+\overline{B})$加A |
| H H H L | AB | $A\overline{B}$加A | A+B | $(A+\overline{B})$加A |
| H H H H | A | A | A | A减1 |



## 2.5.2 多功能算术/逻辑运算单元ALU

- **3.两级先行进位的ALU----ALU扩展**
  - **74181ALU**(4位算术/逻辑运算器)：
  - 如何实现两个8位2进制数据($a_7a_6a_5a_4a_3a_2a_1a_0$与$b_7b_6b_5b_4b_3b_2b_1b_0$)的运算？
    - 1片**74181ALU**实现两个4位数据运算，2片**74181ALU**。可以实现两个8位运算，依次类推。
  - 考虑片间连接的问题：
    - 如果两片**74181ALU**直接没有任何关系，则实现2组4位数据的运算。两片之间如何连接，才能实现2个8位数据的运算？
    - 使用**74182**先行进位部件**CLA**，将低4位运算的进位取值($a_3a_2a_1a_0$与$b_3b_2b_1b_0$)提供给高4位运算($a_7a_6a_5a_4$与$b_7b_6b_5b_4$)。



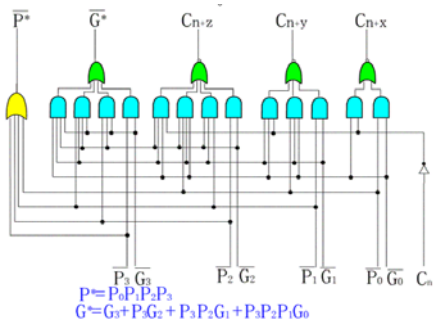
## 2.5.2 多功能算术/逻辑运算单元ALU

- **3.两级先行进位的ALU----ALU扩展**
  - **74182CLA**(先行进位部件)：
  - 假设4片/组74181的先行进位输出依次为:$P_0,G_0$; $P_1$, $G_1$; $P_2,G_2$; $P_3,G_3$, 先行进位部件**74182CLA**所提供的进位逻辑关系如下：
    - $C_{n+x}=G_0+P_0C_n$
    - $C_{n+y}=G_1+P_1C_{n+x}=G_1+G_0P_1+P_0P_1C_n$
    - $C_{n+z}=G_2+P_2C_{n+y}=G_2+G_1P_2+G_0P_1P_2+P_0P_1P_2C_n$
    - $C_{n+4}=G_3+P_3C_{n+z}=G_3+G_2P_3+G_1P_2P_3+G_0P_1P_2P_3+P_0P_1P_2P_3C_n$
      $=G^*+P^*C_n$
  - 其中：$P^*=P_0P_1P_2P_3$ $G^*=G_3+G_2P_3+G_1P_2P_3+G_0P_1P_2P_3$
  - $G^*$为成组先行进位发生输出
  - $P^*$为成组先行进位传送输出



## 2.5.2 多功能算术/逻辑运算单元ALU

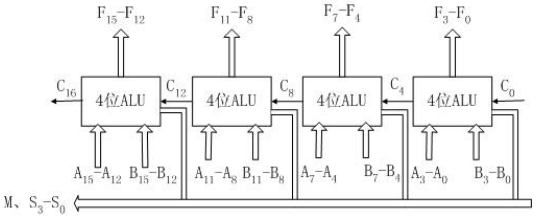
- **3.两级先行进位的ALU----ALU扩展**
  - **74182CLA**(先行进位部件)：






## 2.5.2 多功能算术/逻辑运算单元ALU

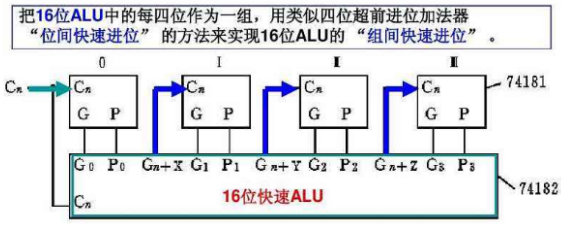
- **3.两级先行进位的ALU----ALU扩展**
  - **16位的ALU**(使用4片4位的74181 ALU组成)：**组内先行进位，组间串行进位**。
  - 速度慢





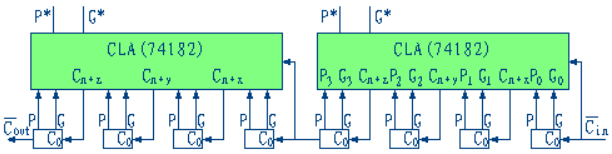

## 2.5.2 多功能算术/逻辑运算单元ALU

- **3.两级先行进位的ALU----ALU扩展**
  - **16位的ALU**(使用4片4位的74181 ALU和1片74182 CLA组成)：**组内先行进位，组间先行进位**。
  - 速度快



## 2.5.2 多功能算术/逻辑运算单元ALU

- **3.两级先行进位的ALU----ALU扩展**
  - **32位的ALU**(使用8片4位的74181 ALU和2片74182 CLA组成)：**片内先行进位，片间串行进位**。
  - 速度快





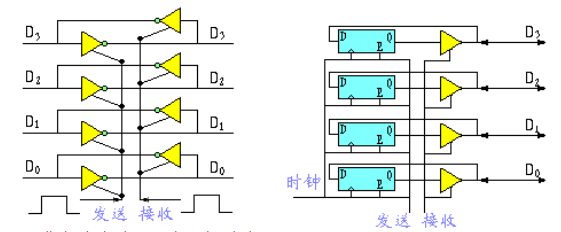

## 2.5.3 内部总线

- **内部总线**
  - 机器内部各部份数据传送频繁,可以把寄存器间的数据传送通路加以归并,**使不同来源的信息在此传输线上分时传送,**组成总线结构。
  - **内部总线是计算机CPU内各部件之间传送信息的公用的数据通道。**
  - 分类
    - 所处位置
      - 内部总线（连接CPU内部各部件）
      - 外部总线（系统总线，连接cpu与存储器，I/O设备）
    - 逻辑结构
      - 单向传送总线
      - 双向传送总线



## 2.5.3 内部总线

- **双向总线举例**
  - **当“发送”信号有效时,数据从左向右传送。**
  - **当“接收”信号有效时,数据从右向左传送。**




(a) 带有缓冲器的双向数据总线 (b) 带有锁存器的4位双向数据总线



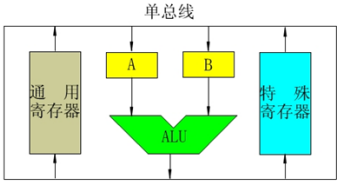
## 2.5.4 定点运算器的基本结构

- **运算器**：
  - 组成:
    - ALU、阵列乘/除法器、寄存器组、多路开关、三态缓冲器、数据总线等逻辑部件。
  - 运算器的设计，主要是围绕ALU和寄存器组与数据总线之间如何传送操作数和运算结果的。
- 三种常用结构形式：
  - 单总线、双总线、三总线。



## 2.5.4 定点运算器的基本结构

- **单总线结构**：
  - 由于所有部件都接到同一总线上，所以数据可以在任何两个寄存器之间，或者在任一个寄存器和ALU之间传送。如果具有阵列乘法器或除法器，那么它们所处的位置应与ALU相当。
  - **工作特点**：在同一时间内，只能有一个操作数放在单总线上。为了把两个操作数输入到ALU，需要分两次来送，而且还需要A、B两个缓冲寄存器。

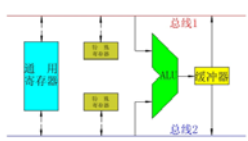



（a）单总线结构的运算器



## 2.5.4 定点运算器的基本结构

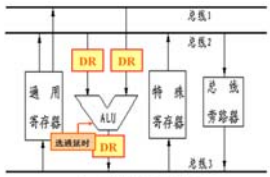
- **双总线结构**：
  - **结构特点**：
    - 两个操作数可以同时加到ALU进行运算，只需一次操作控制，而且马上就可以得到运算结果。图中，两条总线各自把其数据送至ALU的输入端。
    - 特殊寄存器分为两组，它们分别与一条总线交换数据。这样，通用寄存器中的数就可进入到任一组特殊寄存器中去，从而使数据传送更为灵活。
  - 在双总线结构的运算器中，操作的控制要分两步完成:
    - 1、 在ALU的两个输入端输入操作数，形成结果并送入输出缓冲寄存器;
    - 2、把结果送入目的寄存器。




（b）双总线结构的运算器

## 2.5.4 定点运算器的基本结构

- **三总线结构**：
  - 在三总线结构的运算器结构中，ALU的两个输入端分别由两条总线供给，而ALU的输出则与第三条总线相连。算术逻辑操作就可以在一步的控制之内完成。
  - 由于ALU本身有时间延迟，所以发出输出结果的选通脉冲必须考虑到包括这个延迟。另外，设置了一个总线旁路器。如果一个操作数不需要修改，而直接从总线2传送到总线3，那么可以通过控制总线旁路器把数据传出；如果一个操作数传送时需要修改，那么就借助于ALU。很显然，三总线结构的运算器的最大特点是操作时间快。（通常仍然带缓冲器DR）




(c)三总线结构的运算器



## 2.6 浮点运算方法和浮点运算器

- 2.6.1 浮点加法、减法运算
- 2.6.2 浮点乘法、除法运算
- 2.6.3 浮点运算流水线
- 2.6.4 浮点运算器实例



## 2.6.1 浮点加法、减法运算

- 两个浮点数$x$和$y$： $x=2^{E_x}\cdot M_x$ $y=2^{E_x}\cdot M_y$
  - $E_x$和$E_y$是数$x$和$x$的阶码，$M_x$和$M_Y$是数$x$和$y$的尾数。
- 加法和减法的运算规则是：
  - $x\pm y=2^{E_x}\cdot M_x\pm M_y 2^{E_y}=(M_x 2^{E_x-E_y}\pm M_y)2^{E_y}$，$E_x\leqslant E_y$
- 浮点加减运算步骤如下：
  - 1) 0操作数检查；
  - 2) 比较阶码并完成对阶（小阶向大阶对齐：$E_x$- $E_y$）；
  - 3) 尾数求和运算；
  - 4) 结果规格化；
  - 5) 舍入处理。



## 2.6.1 浮点加法、减法运算

- **0操作数检查**：
  - 浮点加减运算过程比定点运算过程复杂。如果判知两个操作数 $x$ 或 $y$ 中有一个数为0,即可得知运算结果而没有必要再进行后续的一系列操作以节省运算时间。0操作数检查步骤则用来完成这一功能。
- **对阶**：见后面
- **尾数求和运算**：
  - 对阶结束后,即可进行尾数的运算。不论加法运算还是减法运算,都按加法进行操作,其方法与定点加减法运算完全一样。



## 2.6.1 浮点加法、减法运算

- **对阶**：
  - 比较阶码并完成对阶（小阶向大阶对齐：$E_x$- $E_y$）：
    - 两浮点数进行加减,首先要看两数的阶码是否相同,即小数点位置是否对齐。让两个运算数据阶码相同,这个过程叫作**对阶**。
  - 设：两数阶码$E_x$和$E_y$之差为：$\triangle E = E_x - E_y$
    - 若$\triangle E=0$，表示两数阶码相等，即$E_x=E_y$；
    - 若$\triangle E>0$，表示$E_y<E_x$；若$\triangle E<0$，表示$E_x>E_y$。
    - 当$E_x\neq E_y$时，通过尾数的移动来改变$E_x$或$E_y$,使之相等。
  - 尾数**左移**引起**最高有效位**的丢失,造成**大误差**。
  - 尾数**右移**引起**最低有效位**的丢失,造成**小误差**。
  - 对阶操作：**小阶向大阶看齐**。
    - 小阶数的尾数每右移一位,其阶码应加1,直到两数的阶码相等为止，即：阶差$\triangle E=0$。



## 2.6.1 浮点加法、减法运算

- **结果规格化**：
  - 结果规格化：**0.5≤|M|<1.0**
  - **右规 / 向右规格化**：小数点位置不变，尾数右移1位,阶码加1。
  - **|M|>1.0**，即运算结果为： **01.φ…φ**或**10.φ…φ**
    - 在浮点加减运算时，尾数求和的结果**溢出时**,即两符号位不等，这在定点加减法运算中要产生"溢出中断"的。但在浮点运算中,可以通过将运算结果右移,即可得到正确结果,并实现规格化处理,称为尾数的向右规格化。
  - **左规 / 向左规格化**：小数点位置不变，尾数左移1位,阶码减1 。
  - **|M|<0.5**。

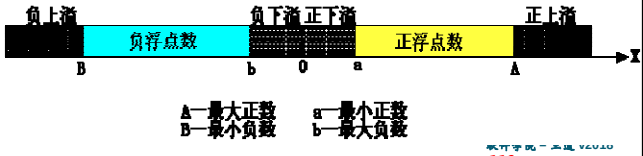
## 2.6.1 浮点加法、减法运算

- **舍入处理**：
  - 在对阶或向右规格化时,尾数要向右移位,被右移的尾数的低位部分会被丢掉,从而造成一定误差,因此要进行舍入处理。
    - “**0舍1入**”法,即如果右移时被丢掉数位的最高位为0则舍去，为1则将尾数的末位加“1”。
    - “**就近舍入**”法，即只要数位被移掉，就在尾数的末尾恒置“1”。
    - “**朝0舍入**”法
    - “**朝+ ∞**”法
    - “**朝- ∞**”法



## 2.6.1 浮点加法、减法运算

- **浮点数溢出**：P54图2-17
  - 当机器浮点数值大于最大正数值(+∞)，或小于最小负数值(-∞)时，称为上溢，这两种情况的特点：阶码运算值“上溢”，机器必须做中断处理。
  - 当机器浮点数值小于最小正数值/**正无穷小**，或大于最大负数值/**负无穷大**时，称为下溢。
  - 浮点数的溢出是以其阶码溢出表现出来的。在运算过程中，只要检查阶码是否溢出即可；若阶码溢出,则要进行相应的处理。另外，对尾数的溢出也需要处理。

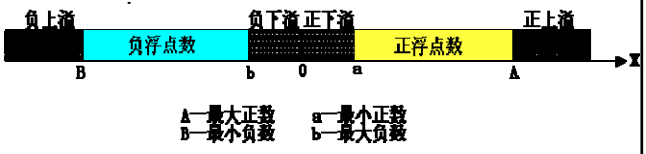



A—最大正数　a—最小正数
B—最小负数　b—最大负数



## 2.6.1 浮点加法、减法运算

- **浮点数溢出**：
  - **阶码上溢** 阶码超过了可能表示的最大值的正指数值，一般认为该浮点数是＋∞和－∞。
  - **阶码下溢** 阶码超过了可能表示的最小值的负指数值，一般认为该浮点数=0。
  - **尾数溢出** 将尾数右移，阶码增1即可得到正确结果。




A—最大正数　a—最小正数
B—最小负数　b—最大负数



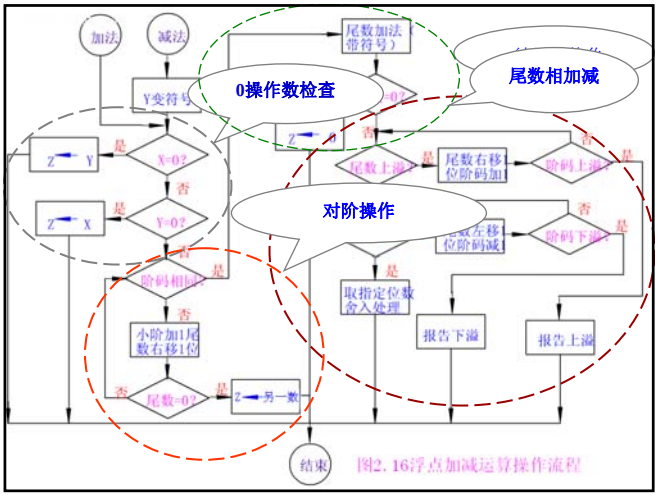



图2.16浮点加减运算操作流程

## 2.6.1 浮点加法、减法运算

- 例：设浮点数均以补码表示，阶码采用双符号位，尾数采用单符号位,求$x+y$。已知：
  - $x=2^{+010}\times0.11011011, y=2^{+100}\times(-0.10101100)$,
  - [解]：浮点编码表示为(阶码 数符 尾数,尾数、阶码都用补码)：
  - $[x]_{浮}$=00 010， 0.11011011 $[y]_{浮}$=00 100， 1.01010100
  - 1、0操作数检查（非0）
  - 2、对阶：
    - $\triangle E=E_x-E_y=[E_x]_{补}+[-E_y]_{补}$=00 010+11 100=11 110
    - 即$\triangle E$为−2，$x$的阶码小，应使$M_x$右移两位，$E_x$加2
    - $[x]_{浮}$=00 100, 0.00110110(11)
    - 其中(11)表示$M_x$右移2位后移出的最低两位数。
  - 3、尾数加：

$$\begin{array}{rr} & 0.0011010\,(11) \\ + & 1.0101010\,0 \\ \hline & 1.0001010\,(11) \end{array}$$



## 2.6.1 浮点加法、减法运算

- 即：**尾数：M = 1.1000101 0 (11)**
  - 4、结果规格化
  - 尾数运算结果的符号位与最高数值位同值为1(负)，应执行左规处理，结果为M=1.00010101(10)，阶码减1，即：
  - 00 100-1= 00 011。
  - [x+y]浮 = 00 011， 1.00010101(10)
  - 5、舍入处理：采用“0舍1入法”处理,则有：

$$\begin{array}{rr} & 1.0001010\,1\,(\cancel{1\,0}) \\ + & \qquad\qquad\quad 1 \\ \hline & 1.0001011\,0 \end{array}$$

  - 6、溢出判断：阶码为：00 011，符号位为00，∴ 没有溢出。故得最终结果为：
  - $x+y=2^{+011}\times(-0.11101010)$

## 2.6.2 浮点乘法、除法运算

- 运算规则：设 $x=M_x\cdot 2^{E_x}$，$y=M_y\cdot 2^{E_y}$
  - 1、乘法
    $X\times Y=(M_x\times M_y)\cdot 2^{E_x+E_y}$
  - 2、除法
    $X\div Y=(M_x\div M_y)\cdot 2^{E_x-E_y}$
- 运算步骤：
  - 1) 0 操作数检查；
  - 2) 阶码加/减操作；（**定点加/减法器完成**）
  - 3) 尾数乘/除操作；（**阵列乘/除法器完成**）
  - 4) 结果规格化/ 舍入处理；
  - 5) 确定积/商符号；



## 2.6.2 浮点乘法、除法运算

- 2) 阶码加/减运算：设 $E_x\pm E_y$
  - 4种阶码运算：＋＋ 1、－－ 1、两阶码求和、两阶码求差，检查结果是否溢出。
  - 阶码表示：补码或移码。
- 移码的运算规则和判定溢出的方法
  - $[x+y]_{移}=[x]_{移}+[y]_{补}$　$[x-y]_{移}=[x]_{移}+[-y]_{补}$
  - 移码双符号位，最高位为0；01为正，00为负；
  - 运算结果符号为取值：
    - **00**：不溢出，结果为负；
    - **01**：不溢出，结果为正；
    - **10**：上溢出；
    - **11**：下溢出；



## 2.6.2 浮点乘法、除法运算

- 2) 阶码加/减运算：设 $E_x\pm E_y$
- 例：$x=+011$，$y=+110$，求$[x+y]_{移}$ 和 $[x-y]_{移}$， 并判断是否溢出。
  - 解：　$[x]_{移}=01\ 011$，
    $[y]_{补}=00\ 110$，　$[-y]_{补}=11\ 010$
  - $[x+y]_{移}=[x]_{移}+[y]_{补}=10\ 001$ 上溢出；
  - $[x-y]_{移}=[x]_{移}+[-y]_{补}=00\ 101$ 不溢出，结果为负；
  - 真值：$x-y=-3$



## 2.6.2 浮点乘法、除法运算

- 4)结果规格化/舍入处理：
  - 浮点加减法对结果的规格化及舍入处理也适用于浮点乘除法。
  - **截断处理**：无条件地丢掉正常尾数最低位之后的全部数值。会影响结果的精度。
  - **舍入处理**：运算过程中保留右移中移出的若干低位的值,最后再按某种规则用这些位上的值修正尾数。
  - **尾数原码表示**：
    - **恒置1**：舍去尾数中有为1的数值位,让有效尾数最低位为1。
    - **0舍1入法**：舍去尾数最高位为1，有效尾数加1，否则加0。
  - **尾数补码表示**：
    - **不舍不如**：舍去尾数为0时,有效尾数加0；
    - **舍**：舍去尾数最高位为0 时,其余各位不全为0时；或舍去尾数最高位为1,其余各位均为0时,有效尾数加0；
    - **入**：舍去尾数最高位为1,其余各位不全为0 时，有效尾数加1；



## 2.6.2 浮点乘法、除法运算

- 4)结果规格化/ 舍入处理：
  - 例：求执行只保留小数点后4位有效数字的舍入操作值。
    - $[x_1]_{补}=11.01100000$
    - $[x_2]_{补}=11.01100001$
    - $[x_3]_{补}=11.01101000$
    - $[x_4]_{补}=11.01111001$
  - 执行舍入操作后,其结果值分别为：
    - $[x_1]_{补}=11.0110$　(不舍不入)
    - $[x_2]_{补}=11.0110$　(舍)
    - $[x_3]_{补}=11.0110$　(舍)
    - $[x_4]_{补}=11.1000$　(入)



## 2.6.2 浮点乘法、除法运算

- 例：浮点数$x=2^{-5}\times 0.0110011$，$y=2^{3}\times(-0.1110010)$，阶码用4位移码表示，尾数(含符号位)用 8位补码表示。求：$[x\times y]_{浮}$。要求用补码完成尾数乘法运算,运算结果尾数保留高8位(含符号位),并用尾数低位字长值处理舍入操作。
- 解：移码采用双符号位,尾数补码采用单符号位
  - $[M_x]_{补}=0.0110011$, $[M_y]_{补}=1.0001110$,
  - $[E_y]_{移}=01\ 011$, $[E_y]_{补}=00\ 011$, $[E_x]_{移}=00\ 011$,
  - $[x]_{浮}=00\ 011,\ 0.0110011$, $[y]_{浮}=01\ 011,\ 1.0001110$
  - 1、0操作数检查（非0）
  - 2、阶码加
    - $[E_x+E_y]_{移}=[E_x]_{移}+[E_y]_{补}=00\ 011+00\ 011=00\ 110$
    - 真值=-2
  - 3、尾数乘
    - $[M_x\times M_y]_{补}=[M_x]_{补}+[M_y]_{补}$
    - $=[0.0110011]_{补}\times[1.0001110]_{补}=[1.1010010,\ldots]_{补}$

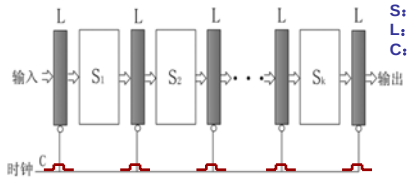
## 2.6.2 浮点乘法、除法运算

- 例：浮点数$x=2^{-5}\times0.0110011$，$y=2^{3}\times(-0.1110010)$，阶码用4位移码表示，尾数(含符号位)用 8位补码表示。求：$[x\times y]_{浮}$。要求用补码完成尾数乘法运算,运算结果尾数保留高8位(含符号位),并用尾数低位字长值处理舍入操作。
- 解：移码采用双符号位,尾数补码采用单符号位
  - 4、规格化/舍入处理
    - $[M_x\times M_y]_{补}=$ **[1.1010010,1001010]补**
    - 尾数符号位与最高数值位符号相同,不是规格化的数, 左规
    - 阶码=**00 101**(-3)，尾数=**[1.0100101,0010100]补**
    - 尾数为负数,取尾数高位字长，按舍入规则,舍去低位字长，故尾数为1.0100101 。
    - 最终相乘结果为：$[x\times y]_{浮}$=00 101,1.0100101
    - 其真值为：$x\times y=2^{-3}\times(-0.1011011)$



## 2.6.3 浮点运算流水线

- 提高并行操作：
  - **空间并行性**：增加冗余部件，如增加多操作部件处理机和超标量处理机
  - **时间并行性**：改善操作流程如：流水线技术
- **1.流水线原理**
  - 计算机的流水处理过程同工厂中的流水装配线类似。为了实现流水,首先必须把输入的任务分割为一系列的子任务,使各子任务能在流水线的各个阶段并发地执行。将任务连续不断地输入流水线,从而实现了子任务的并行。

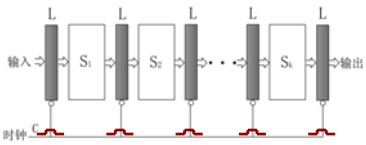



**S：处理一个子任务的过程**
**L：高速缓冲寄存器**
**C：统一的时钟**



## 2.6.3 浮点运算流水线

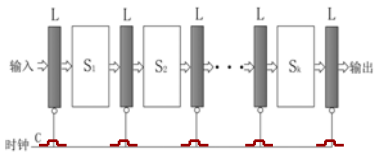
- **1.流水线原理**：
  - 在流水线中必须是连续的任务，只有不断的提供任务才能充分发挥流水线的效率
  - 把一个任务分解为几个有联系的子任务。每个子任务由一个专门的功能部件实现
  - 在流水线中的每个功能部件之后都要有一个缓冲寄存器，或称为锁存器
  - 流水线中各段的时间应该尽量相等，否则将会引起“堵塞”和“断流”的现象
  - 流水线需要有装入时间和排空时间，只有当流水线完全充满时，才能充分发挥效率






## 2.6.3 浮点运算流水线

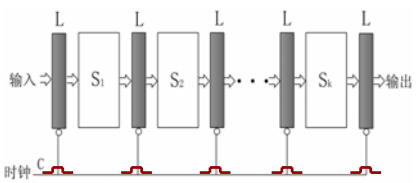
- **1.流水线原理**：如图，
  - **过程段**($S_i$)：处理一个子任务的过程。线性流水线由一系列串联的过程段组成。
  - **缓冲寄存器**(L)：各个过程之间设有高速的以暂时保存上一过程子任务处理的结果。
  - **统一的时钟**(C)：控制数据从一个过程段流向相邻的过程段。
  - **时钟周期** $\tau$：设过程段$S_i$所需的时间为$\tau_i$,缓冲寄存器的延时为$\tau_l$,线性流水线的定义为：
    - $\tau=\max\{\tau_i\}+\tau_l=\tau_m+\tau_l$
  - 流水线处理的频率f：**f=1/τ**






## 2.6.3 浮点运算流水线

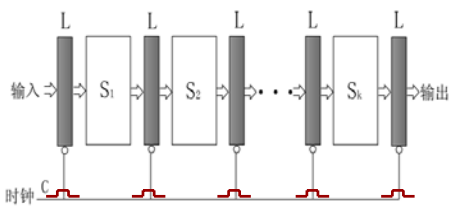
- **1.流水线原理**：
  - 在流水线处理中,当任务饱满时,任务源源不断的输入流水线,不论有多少级过程段,每隔一个时钟周期都能完成并输出一个任务。从理论上说,一个具有k 级过程段的流水线处理n 个任务需要的**时钟周期**数为：
    - $T_k=k+(n-1)$　其中：k个时钟周期用于处理第一个任务。
  - 如果用非流水线的硬件来处理这n个任务，时间上只能串行进行，无缓冲，则所需**时钟周期**数为：
    - $T_L=n\cdot k$






## 2.6.3 浮点运算流水线

- **1.流水线原理**：
  - 将$T_L$和$T_k$的比值定义为k级线性流水线的**加速比**：$c_k=\dfrac{T_L}{T_k}=\dfrac{nk}{k+(n-1)}$
  - 当 **n>>k** 时，**$C_k$ → k** 。理论上k级线性流水线处理几乎可以提高处理速度k 倍。实际上：由于存储器冲突、数据相关等因素的制约，这个理想的加速比不一定能达到。



## 2.6.3 浮点运算流水线

- **2.流水线浮点加法器：**
  - 浮点数加减法由0操作数检查、对阶操作、尾数操作、结果规格化及舍入处理实现完成，因此流水线浮点加法器可由**4个过程段**组成。
  - (1) 求阶差 (2) 对阶 (3) 相加 (4) 规格化
  - **$A=a\times 2^{p}$ ，$B=b\times 2^{q}$** 在4级流水线加法器中实现上述浮点加法时,分为以下操作：

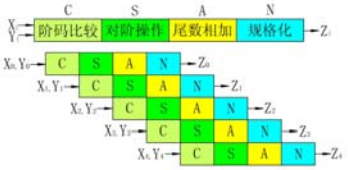



图2.21 向量加法计算的流水时空图



## 2.6.3 浮点运算流水线

- **2.流水线浮点加法器：**

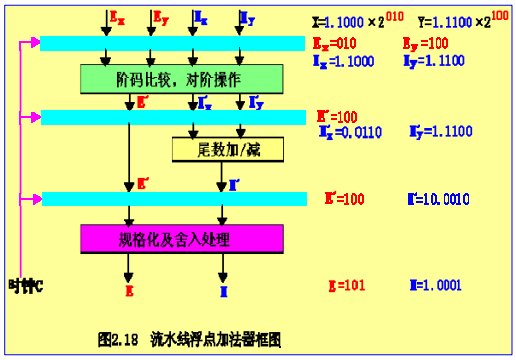



图2.18 流水线浮点加法器框图



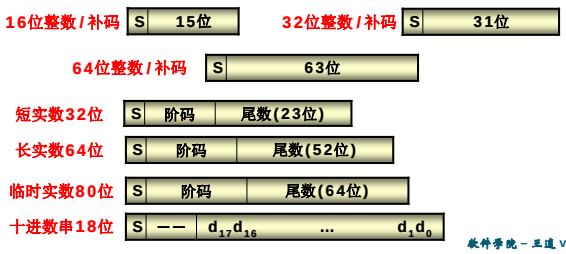
## 2.6.3 浮点运算流水线

- **2.流水线浮点加法器：**
  - 例：假设流水线加法器每个过程段所需的时间为：
    **求阶差 $\tau_1=70ns$，对阶 $\tau_2=60ns$，相加 $\tau_3=90ns$，规格化 $\tau_4=80ns$，缓冲寄存器L的延时为 $t_l=10ns$。**
    - 求：(1) 4 级流水线加法器的加速比为多少？
    - (2)如果每个过程段的时间相同，即都为75ns，(包括缓冲寄存器时间)，加速比是多少？
  - 解：
    - (1) 加法器的流水线时钟周期至少为:
      - $\tau=90ns+10ns=100ns$
    - 采用同样的逻辑电路，但不是流水线方式，则浮点加法所需的时间为:**不需要缓冲**
      - $\tau 1+\tau 2+\tau 3+\tau 4=300ns$
    - 4级流水线加法器的加速比为:
      - $C_k=300/100=3$
    - (2) 当每个过程段的时间（包括缓冲延迟）都是75ns时,加速比为:
      - $C_k=300/75=4$



## 2.6.4 浮点运算器实例

- **1.CPU之外的浮点运算器：**
  - 80×87是美国Intel公司为处理浮点数等数据的算术运算和多种函数计算而设计生产的专用算术运算处理器。由于它们的算术运算是配合80×86CPU进行的，所以又称为协处理器。
  - 可处理包括二进制浮点数、二进制整数、和压缩十进制数串三大类7种数据，其中浮点数的格式符合IEEE754标准。





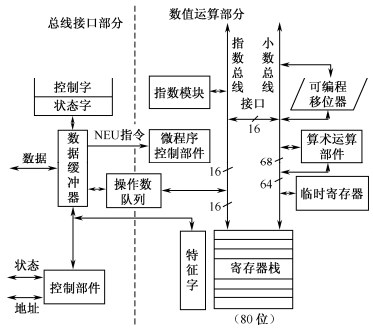

## 2.6.4 浮点运算器实例

- **1.CPU之外的浮点运算器：**
  - 80×87数据格式：
    - **S符号位**，0代表正，1代表负。
    - **阶码**基值为2，用移码表示。
    - **尾数**用原码表示。
  - 浮点数有32位、64位、80位三种。
    - 80×87从存储器取数和向存储器写数时，均用80位的临时实数和其他6种数据类型执行自动转换。全部数据在80×87 中均以80位临时数据的形式表示。因此80×87具有80位字长的内部结构，并有八个80位字长以“先进后出”方式管理的寄存器组，又称**寄存器堆栈**。



## 2.6.4 浮点运算器实例

- **1.CPU之外的浮点运算器：**
  - 80×87浮点运算器逻辑图：



## 2.6.4 浮点运算器实例

- **2.CPU之内的浮点运算器：**
  - 奔腾CPU将浮点运算器包含在芯片内。浮点运算部件采用流水线设计。指令执行过程分为8段流水线。
  - 前4 段为指令预取(DF)、指令译码($D_1$)、地址生成($D_2$)、取操作数(EX)，在U、V流水线中完成；
  - 后4段为执行1($X_1$)、执行2($X_2$)、结果写回寄存器堆(WF)、错误报告(ER)，在浮点运算器中完成。
  - 一般情况下，由U流水线完成一条浮点数操作指令。浮点部件内有浮点专用加法器、乘法器和除法器，有8个80位寄存器组成的寄存器堆，内部的数据总线为80位宽。因此浮点部件可支持IEEE754标准的单精度和双精度格式的浮点数。另外还使用一种称为临时实数的80位浮点数。对于浮点数的取数、加法、乘法等操作，采用了新的算法，其执行速度是80486的10倍多。