# 5.3 定点运算

# 5.3.4 定点乘法运算——以定点小数为例

例： X = 0.1101　Y = - 0.1011　　X × Y = - 0.10001111

```
          0.1101
        × 0.1011
-------------------
 0.0000 1101
 0.000  1101
 0.00   0000
+0.0    1101
-------------------
 0.10001111
```

笔算乘法过程

**机器实现问题：**

1. 加法器只有两个数据输入端；
2. 加法器与运算数据位数相同。

**解决方案：**

1. 改n输入数相加为两两相加；
2. 改2n位相加过程为n位相加。

# 原码一位乘法

基本公式：

设 被乘数 $[X]_{原} = X_s . X_1X_2 \cdots X_n$

乘　数 $[Y]_{原} = Y_s . Y_1Y_2 \cdots Y_n$

则　积　$[P]_{原} = [X \times Y]_{原} = (X_s \oplus Y_s).(X^* \times Y^*)$

$= P_s.P_1P_2 \cdots P_nP_{n+1} \cdots P_{2n}$

其中，$X^*$ 和$Y^*$分别是X和Y的绝对值，Xs、Ys和Ps分别表示$[X]_{原}$、$[Y]_{原}$和$[P]_{原}$的符号位。

1）算法推导

**乘积绝对值通式：**

$$P^* = X^* \times Y^* = X^* \times (0.Y_1Y_2 \cdots Y_n) = X^* \times \sum_{i=1}^{n} Y_i \times 2^{-i}$$ 展开得

$$= X^* \times (Y_1 2^{-1} + Y_2 2^{-2} + \cdots + Y_n 2^{-n})$$

$$= 2^{-1}(X^* \times Y_1 + 2^{-1}(X^* \times Y_2 + 2^{-1}(\cdots + 2^{-1}(X^* \times Y_{n-1} + 2^{-1}(X^* \times Y_n + 0))\cdots)))$$

# 原码一位乘法（续）

- 令$Z_i$表示第i次的部分积，上述迭代公式可写成递推公式的形式：

$$
\begin{cases}
Z_0 = 0 \\
Z_1 = 2^{-1}(Y_n \times X^* + Z_0) \\
Z_2 = 2^{-1}(Y_{n-1} \times X^* + Z_1) \\
\cdots\cdots \\
Z_i = 2^{-1}(Y_{n-i+1} \times X^* + Z_{i-1}) \\
\cdots\cdots \\
Z_n = 2^{-1}(Y_1 \times X^* + Z_{n-1})
\end{cases}
$$

- 由递推公式可知，$X^* \times Y^*$ 可转化成n次加法和n次右移完成，其中
  - $Z_i$表示第i次运算的部分积；
  - $Z_0$表示部分积的初值（$Z_0$ =0）；
  - $Z_n$则为最后乘积的绝对值，即
  - $P^* = X^* \times Y^* = Z_n$

# 原码一位乘法运算规则

- 参加运算的两个数均为原码，两数符号位不参加运算，取其绝对值做无符号数乘法；
- 乘积的符号位单独处理，由两数符号“异或”产生；
- 设部分积初值为0。为防止运算过程中产生暂时性溢出，部分积采用单符号位参加运算。n位数相乘，部分积连同符号位应取n+1位；
- 运算前先检测0，只要两数有任意一个为0，则乘积为0，不再进行运算；
- 部分积运算：用乘数的最低位作为判断位，若为1，加被乘数；若为0，不加（相当于加0）；运算后部分积逻辑右移一位。
- 当乘数数值部分为n位时，共作n次“加法”和“右移”，最后得到2n位的乘积绝对值。

# 原码一位乘法运算实例

例： X = 0.1101　　Y = - 0.1011

$[X]_{原}$ = 0.1101　　$[Y]_{原}$ = 1.1011

则 X* = 0.1101　　Y* = 0.1011

```
      0.1101
    × 0.1011
    ------
      1101
     1101
    0000
  + 1101
  ------
  0.10001111
```

手工运算过程

| 部分积 | 乘数 |
|---|---|
| 0.0000 | 0.1011 |
| + 0.1101 | |
| 0.1101 | |
| → 0.0110 | 10.101 |
| + 0.1101 | |
| 1.0011 | |
| → 0.1001 | 11 0.10 |
| → 0.0100 | 111 0.1 |
| + 0.1101 | |
| 1.0001 | |
| → 0.1000 | 1111 0. |

$[X \times Y]_{原}$ = 1.1000 1111

X×Y = -0.1000 1111

计算机内运算的描述方法

# 整数原码一位乘法

- 原码一位乘算法规则同样适用于整数，只要把例题中的小数点“.”改为逗号“,”即可。
- 注意整数小数点的隐含位置和定点小数不一样，故乘积最终要向右对齐小数点，这是两种定点数据格式运算时的主要差别。

- 例：X = 1101，Y = - 1011，用原码一位乘法求X×Y=?
- 解：$[X]_{原}$= 0,1101，$[Y]_{原}$=1,1011

  $X^*$= 0,1101，　$Y^*$=1011

  $P_s=X_s \oplus Y_s=0 \oplus 1=1$；$P^*$=0, 1000 1111

  $[P]_{原}=[X \times Y]_{原}$=1, 1000 1111

  X×Y= - (0,1101×1011) = - 1000 1111

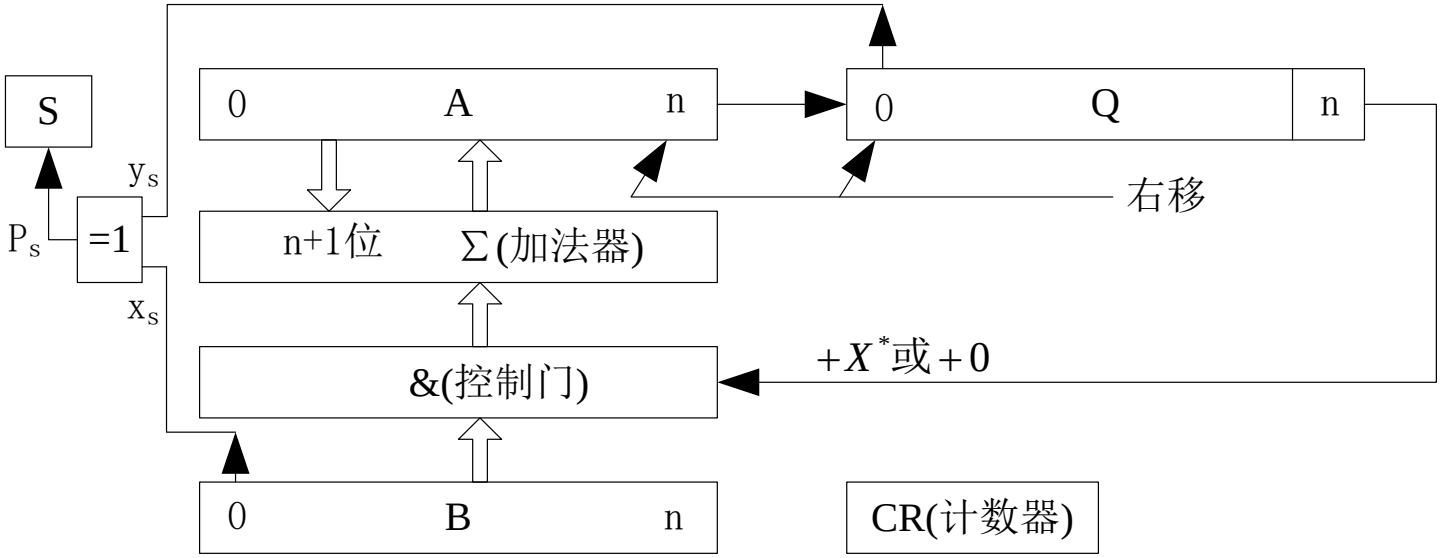
# 整数原码一位乘法运算实例

| 部分积 | 乘数$Y_n$ | 说明 |
|---|---|---|
| 0，0000 | 1 0 1 1 | 初始化 |
| + 0，1101 | | +X* |
| 0，1101 | | 第1步 |
| →1 0，0110 | 1 1 0 1 | |
| + 0，1101 | | +X* |
| 暂时性溢出→ 1，0011 | | 第2步 |
| →1 0，1001 | 1 1 1 0 | |
| + 0，0000 | | +0 |
| 0，1001 | | 第3步 |
| →1 0，0100 | 1 1 1 1 | |
| + 0，1101 | | +X* |
| 1，0001 | | 第4步 |
| →1 0，1000 | 1 1 1 1 | |

小数点位置

# 原码一位乘法运算的硬件实现

## 原码一位乘法运算器框图




- 数值位为n位，运算器的总位数应达到n+1位；
- 原码乘法运算器主要由一个加法器和三个寄存器组成；
- A为累加器，用来存放部分积；
- B为被乘数寄存器，其值保持不变；
- Q为乘数/部分积寄存器，A、Q首尾相接实现联合右移一位。

# 补码一位乘法（比较法）

原码乘法简单易实现，但有**两个问题**:

1. 符号位与数值位分别处理，不方便；
2. 若数据为补码形式，可能需要多于两次的补↔原码转换。

采用补码表示的计算机中往往希望直接用补码完成乘法运算，即从补码操作数开始运算，然后直接得到补码的积。

下面看一看补码乘运算的实现算法。

# 补码一位乘法(比较法)的算法推导

设 被乘数 $[X]_{补} = X_0.\ X_1\ X_2 \cdots X_n$

乘　数 $[Y]_{补} = Y_0.\ Y_1\ Y_2 \cdots Y_n$

先复习两个概念:

① 已知 $[X]_{补} = X_0.\ X_1X_2 \cdots X_n$ 时

$$[X/2]_{补} = X_0.\ X_0X_1X_2 \cdots X_{n-1}$$

② 已知 $[Y]_{补} = Y_0.Y_1Y_2 \cdots Y_n$ 时

$$Y = -Y_0 + \sum_{i=1}^{n} Y_i \times 2^{-i}$$

$$= \sum_{i=0}^{n} (Y_{i+1} - Y_i) \times 2^{-i}$$

# 补码一位乘法(比较法)的算法推导

$[X \times Y]_{补} \stackrel{?}{=} [X]_{补} \times [Y]_{补}$

推导：

$$X \times Y = X \times [-Y_0 + \sum_{i=1}^{n} Y_i \times 2^{-i}]$$

$$= X \times (0.Y_1Y_2 \ldots Y_n) - X \times Y_0$$

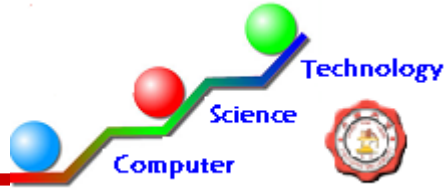
$$[X \times Y]_{补} = [X \times (0.Y_1Y_2 \ldots Y_n) - X \times Y_0]_{补}$$

$$= [X \times (0.Y_1Y_2 \ldots Y_n)]_{补} - [X \times Y_0]_{补}$$

$$\stackrel{?}{=} [X]_{补} \times (0.Y_1Y_2 \ldots Y_n) - Y_0[X]_{补}$$

$$= [X]_{补} \times Y$$

# 补码一位乘法(比较法)的算法推导




求证：$[X\times(0.Y_1Y_2\ldots Y_n)]_{补}=[X]_{补}\times(0.Y_1Y_2\ldots Y_n)$

证明：(1) 当$X\geq 0$时，

$$[X\times(0.Y_1Y_2\ldots Y_n)]_{补}=X\times(0.Y_1Y_2\ldots Y_n)$$

$$=[X]_{补}\times(0.Y_1Y_2\ldots Y_n)$$

(2) 当$X<0$时，

$$[X\times(0.Y_1Y_2\ldots Y_n)]_{补}$$

$$=2+X\times(0.Y_1Y_2\ldots Y_n) \pmod 2$$

$$=2^{n+1}+X\times(0.Y_1Y_2\ldots Y_n) \pmod 2$$

$$=2^{n+1}\times(0.Y_1Y_2\ldots Y_n)+X\times(0.Y_1Y_2\ldots Y_n) \pmod 2$$

$$=(2^{n+1}+X)\times(0.Y_1Y_2\ldots Y_n) \pmod 2$$

$$=(2+X)\times(0.Y_1Y_2\ldots Y_n) \pmod 2$$

$$=[X]_{补}\times(0.Y_1Y_2\ldots Y_n) \pmod 2$$

证毕

# 补码一位乘法(比较法)的算法推导

补码乘法的基本公式：$[X \times Y]_{补} = [X]_{补} \times Y$

$$[Y]_{补} \longrightarrow Y = (-Y_0 + \sum_{i=1}^{n} Y_i \times 2^{-i})$$

展开多项式

$$= -Y_0 \times 2^0 + Y_1 \times 2^{-1} + Y_2 \times 2^{-2} + \ldots + Y_n \times 2^{-n}$$

令：$Y_i \times 2^{-i}$
$= Y_i \times 2^{-(i-1)} - Y_i \times 2^{-i}$

$$= -Y_0 \times 2^0 + (Y_1 \times 2^0 - Y_1 \times 2^{-1}) + (Y_2 \times 2^{-1} - Y_2 \times 2^{-2}) + \ldots +$$

$$= (Y_1 - Y_0) \times 2^0 + (Y_2 - Y_1) \times 2^{-1} + \ldots + (Y_{n+1} - Y_n) \times 2^{-n}$$

合并同次项

最低位后再补1位
令：$Y_{n+1} = 0$

$$= \sum_{i=0}^{n} (Y_{i+1} - Y_i) \times 2^{-i}$$

写成和式

# 补码一位乘法(比较法)的算法推导

## 补码一位乘比较法公式：

$$[P]_{补}=[X\times Y]_{补}=[X]_{补}\times\sum_{i=0}^{n}(Y_{i+1}-Y_i)\times 2^{-i}$$

## 部分积递推公式：

$$[Z_0]_{补}=0$$

$$[Z_1]_{补}=2^{-1}\{[Z_0]_{补}+(Y_{n+1}-Y_n)\times[X]_{补}\}\qquad (令：Y_{n+1}=0)$$

……

$$[Z_i]_{补}=2^{-1}\{[Z_{i-1}]_{补}+(Y_{n-i+2}-Y_{n-i+1})\times[X]_{补}\}$$

……

$$[Z_n]_{补}=2^{-1}\{[Z_{n-1}]_{补}+(Y_2-Y_1)\times[X]_{补}\}$$

$$[Z_{n+1}]_{补}=[Z_n]_{补}+(Y_1-Y_0)\times[X]_{补}=[X\times Y]_{补}=[P]_{补}$$

# 补码一位乘（比较法）算法规则

(1) X、Y均用补码表示，且符号位都参加运算，乘积的符号位由运算过程自动产生；

(2) 设部分积初值为0，部分积采用双符号位运算，但乘数只需要一位符号位参加运算；

(3) 乘数最低位之后增加一位附加位$Y_{n+1}$，且初始令$Y_{n+1}=0$；

(4) 运算前检测0，只要X、Y有任意一个为0，则乘积为0，不再运算；

(5) 用乘数的最低两位$Y_nY_{n+1}$作为判断位，其比较值$(Y_{n+1}-Y_n)$始终决定了每次应执行的操作。

- 部分积运算规则如下：

$Y_nY_{n+1}=00$ 时，$[Z_i]_{补}=[Z_{i-1}]_{补}+0$，算术右移1位；
$Y_nY_{n+1}=01$ 时，$[Z_i]_{补}=[Z_{i-1}]_{补}+[X]_{补}$，算术右移1位；
$Y_nY_{n+1}=10$ 时，$[Z_i]_{补}=[Z_{i-1}]_{补}+[-X]_{补}$，算术右移1位；
$Y_nY_{n+1}=11$ 时，$[Z_i]_{补}=[Z_{i-1}]_{补}+0$，算术右移1位；

(6) 重复n+1次比较和运算，但只右移n次，最后一次只运算不移位。

(7) 运算完成后，为避免误差，乘数的最低两位$Y_nY_{n+1}$清0。

# 补码一位乘（比较法）运算实例

已知：X = 0.1101

Y = - 0.1011

求：X×Y =?

解：$[X]_{补}$ = 0.1101

$[Y]_{补}$ = 1.0101

$[-X]_{补}$ = 1.0011

$[X \times Y]_{补}$ = 11.0111000100

X×Y = -0.10001111

| 部分积 | 乘数 $y_n$ $y_{n+1}$ |
|---|---|
| 0 0. 0 0 0 0 | 1.0 1 0 1 0 |
| +1 1. 0 0 1 1 | |
| 1 1. 0 0 1 1 | |
| → 1 1. 1 0 0 1 | 1 1.0 1 0 1 |
| +0 0. 1 1 0 1 | |
| 0 0. 0 1 1 0 | |
| → 0 0. 0 0 1 1 | 0 1 1.0 1 0 |
| +1 1. 0 0 1 1 | |
| 1 1. 0 1 1 0 | |
| → 1 1. 1 0 1 1 | 0 0 1 1.0 1 |
| +0 0. 1 1 0 1 | |
| 0 0. 1 0 0 0 | |
| → 0 0. 0 1 0 0 | 0 0 0 1 1. 0 |
| +1 1. 0 0 1 1 | |
| 1 1. 0 1 1 1 | 0 0 0 1 0 0 |
| | 清0 |

# 整数补码一位乘（比较法）算法

- 补码一位乘比较法规则同样适用于整数，也需要把例题中的小数点“.”改为逗号“，”，关注的焦点仍是小数点的隐含位置不同于小数。
- 为了硬件实现方便，可将小数点的位置约定在乘数的附加位之后，这意味着乘数在运算前被乘了2。为了保证双倍字长整数乘积的小数点在其最低位（LSB）的右边对齐，最后一步运算后，结果需要算术右移2位。
- 根据补码的符号扩展特性，整数补码乘积高位与符号位取值相同的部分，应该全部看成扩展的符号，而不是有效的数值位。

# 整数补码一位乘（比较法）举例

**X =-1010**
**Y =1101**
**用补码一位乘比较法求X×Y=?**

**解：**
**$[X]_{补}$=11, 0110**
**$[Y]_{补}$=0,1101**
**$[-X]_{补}$=00, 1010**

**$[P]_{补}=[X\times Y]_{补}$**
**=11,1101111 110**
**X×Y=-10000010**

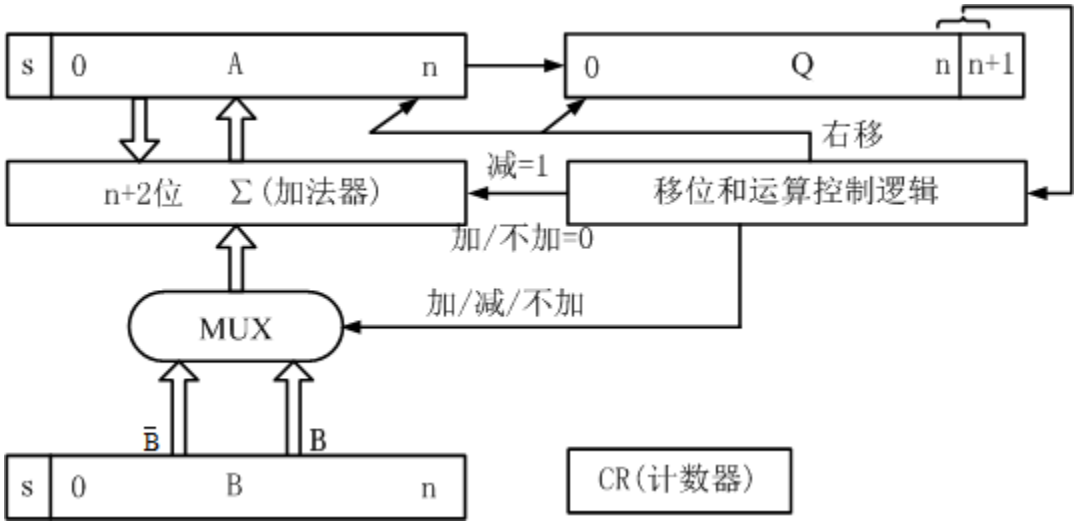
**算法运算步骤：**

| 部分积 | 乘数$y_n$ | 附加值 $y_{n+1}$ | 说明 |
|---|---|---|---|
| 00，0000 | 0，1101 | 0. | 初始化 |
| + 00，1010 | | | $+[-X]_{补}$ |
| 00，1010 | | | 第1步 |
| →1 00，0101 | 00，110 | 1. | |
| + 11，0110 | | | $+[X]_{补}$ |
| 11，1101 | | | 第2步 |
| →1 11，1101 | 100，11 | 0. | |
| + 00，1010 | | | $+[-X]_{补}$ |
| 丢掉→1 00，0111 | | | 第3步 |
| →1 00，0011 | 1100，1 | 1. | |
| + 00，0000 | | | +0 |
| 00，0011 | | | 第4步 |
| →1 00，0001 | 1110 0， | 1. | |
| + 11，0110 | | | $+[X]_{补}$ |
| 11，0111 | 1110 0， | 1. | 第5步 |
| →1 11，1011 | 1111 0 | 0. | |
| →1 11，1101 | 1111 1 | 0 | |

符号延伸　MSB　　小数点位置

# 补码一位乘法运算的实现

## 补码一位乘比较法运算器




特点：与原码一位乘法运算器类似，主要区别为：

- 运算器的位数为n+2位（双符号位运算）；
- 乘数单符号位参加运算，Q寄存器最低位用作附加位$Y_{n+1}$；
- 核心部件为加/减法器， 由n+2个与或控制门电路控制输入；
- Q寄存器的最低两位$Q_nQ_{n+1}$控制部分积作何种运算；

# 补码两位乘法（比较法）

## 提高乘法运算速度的方法

- 基本乘法算法实质：逐位循环迭代，一次乘转换成n次累加和移位，导致乘法运算比加减慢得多。
- 两条常见的提速思路

(1) 改进算法，或研制更高效的算法

- 常用的改进方法：合并乘法步骤，通过提高算法本身的并行性、减少迭代次数来对乘法加速，如
  - ✧两位乘法——一位乘两步合并一步，速度可提高一倍左右；
  - ✧三位乘法——一位乘三步合并一步，速度可提高两倍左右；
- 无法从根本上改变多次迭代的相乘过程

(2) 用硬件阵列直接实现乘法

- 通过多级加法器并行完成运算，后面专门讨论。

# 补码两位乘比较法

## □算法推导

设 被乘数 $[X]_{补} = X_0 .X_1X_2 \cdots X_n$

乘　数 $[Y]_{补} = Y_0 .Y_1Y_2 \cdots Y_n$

由一位乘算法得：

$$[Z_{i+1}]_{补} = 2^{-1}\{ [Z_i]_{补} + (Y_{n-i+1} - Y_{n-i}) \times [X]_{补} \}$$

$$[Z_{i+2}]_{补} = 2^{-1}\{2^{-1}\{ [Z_i]_{补} + (Y_{n-i+1} - Y_{n-i}) \times [X]_{补} \} + (Y_{n-i} - Y_{n-i-1}) \times [X]_{补} \}$$

$$= 2^{-2}\{ [Z_i]_{补} + (Y_{n-i+1} + Y_{n-i} - 2Y_{n-i-1}) \times [X]_{补} \}$$

## □ 结论

补码两位乘比较法的部分积运算由乘数相邻的三位比较决定，运算后每次右移两位。

# 补码两位乘比较法算法规则

(1) X、Y均用补码表示，且符号位都参加运算，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) 乘数最低位之后增加一位附加位$Y_{n+1}$，初始令$Y_{n+1}=0$；

(4) 运算前检测0：只要X、Y有任意一个为0，则乘积为0，不再进行运算；

(5)设乘数数值部分为n位，

当n为奇数时，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；

当n为偶数时，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。

(6) 运算完成后，为避免误差，乘数的最低三位$Y_{n-1}Y_nY_{n+1}$清0。

# 补码两位乘比较法算法规则

(7)用乘数最低三位作判断位，每求一次部分积右移两位，部分积运算规则如下：

| | | |
|---|---|---|
| $y_{n-1}\ y_n\ y_{n+1}$= 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; |

# 补码两位乘法运算实例

| 部分积 | 乘数 $y_n$ $y_{n+1}$ |
|---|---|
| 0 0.0 0 0 0 | 1.0 1 0 1 0 |
| +1 1.0 0 1 1 | |
| 1 1.0 0 1 1 | |
| 1→1 1.1 0 0 1 | 1 1.0 1 0 1 |
| +0 0.1 1 0 1 | |
| 1 0 0.0 1 1 0 | |
| 1→0 0.0 0 1 1 | 0 1 1.0 1 0 |
| +1 1.0 0 1 1 | |
| 1 1.0 1 1 0 | |
| 1→1 1.1 0 1 1 | 0 0 1 1.0 1 0 |
| +0 0.1 1 0 1 | |
| 1 0 0.1 0 0 0 | |
| 1→0 0.0 1 0 0 | 0 0 0 1 1. 0 |
| +1 1.0 0 1 1 | |
| 1 1.0 1 1 1 | 0 0 0 1 0 0 |
| | 清0 |

| 部分积 | 乘 数 $y_n$ $y_{n+1}$ |
|---|---|
| 0 0 0.0 0 0 0 | 1 1.0 1 0 1 0 |
| + 0 0 0.1 1 0 1 | |
| 0 0 0.1 1 0 1 | |
| 2→0 0 0.0 0 1 1 | 0 1 1 1.0 1 0 |
| + 0 0 0.1 1 0 1 | |
| 0 0 1.0 0 0 0 | |
| 2→0 0 0.0 1 0 0 | 0 0 0 1 1 1. 0 |
| + 1 1 1.0 0 1 1 | |
| 1 1 1.0 1 1 1 | 0 0 0 1 0 0 0 |
| | 清0 |

已知: $[X]_{补} = 0.1101$

$[Y]_{补} = 1.0101$

则 $[-X]_{补} = 1.0011$

$[X \times Y]_{补} = 111.01110001000$

# 比较乘法运算总结

| 乘法类型 | 符号位 | | | 累加次数 | 移位 | | |
|---|---|---|---|---|---|---|---|
| | 参加运算 | 部分积 | 乘数 | | 方向 | 次数 | 每次位数 |
| 补码一位乘法 | 是 | 2 | 1 | $n+1$ | 右 | $n$ | 1 |
| 补码两位乘法 | 是 | 3 | 2(n为偶数) | $\frac{n}{2}+1$ | 右 | $\frac{n}{2}$ | 2 |
| | | | 1(n为奇数) | $\frac{n+1}{2}$ | 右 | $\frac{n+1}{2}$ | 2(最后一次移1位) |

注：n为乘数的数值部分的位数

# 5.3.5 定点除法运算——定点小数为例

例如：X = 0.1011

Y = -0.1101

笔算除法过程：

```
          0.1101
      ┌──────────
0.1101 │ 0.10110
       -0.01101
       ---------
        0.010010
       -0.001101
       ---------
        0.00010100
       -0.00001101
       ---------
        0.00000111
```

机器实现问题：

1. 需单独设计比较器线路；
2. 需2n位的减法器线路。

解决方案：

1. 比较操作改由“试减”实现；
2. 将除数右移改为部分余数左移；
3. 减法由$+[-Y^*]_{补}$转化为加法实现。

# 定点除法运算——定点小数为例

基本公式：

设 被除数 $[X]_{原} = X_s . X_1 X_2 \cdots X_n$

除 数 $[Y]_{原} = Y_s . Y_1 Y_2 \cdots Y_n$

若 $0 < X^* < Y^*$

则 $[Q]_{原} = [X \div Y]_{原} = (X_s \oplus Y_s).(X^* \div Y^*) = Q_s.Q_1Q_2 \cdots Q_n$

其中，$X^*$ 和$Y^*$分别是X和Y的绝对值，$X_s$、$Y_s$和$Q_s$分别表示$[X]_{原}$、$[Y]_{原}$和$[Q]_{原}$的符号位。

# 原码恢复余数除法

例如：

X = -0.1011

Y = -0.1101

求：X÷Y=?

解：

$[X]_{原}$= 1.1011

$[Y]_{原}$= 1.1101

$X^*$ = 0.1011

$Y^*$ = 0.1101

$[-Y^*]_{补}$ = 1.0011

$[X/Y]_{原}$=0.1101

$R^*=0.0111\times2^{-4}$

| 被除数（余数） | 商 | 说　明 |
|---|---|---|
| 0.1011 | 0.0000 | |
| +1.0011 | | +$[-Y^*]_{补}$（减除数） |
| 1.1110 | | 余数< 0, 商上0 |
| +0.1101 | | 恢复余数 +$Y^*$ |
| 0.1011 | | |
| ←1.0110 | 0.0000 | 左移一位 |
| +1.0011 | | +$[-Y^*]_{补}$ |
| 0.1001 | | 余数> 0, 商上1 |
| ←1.0010 | 0.0001 | 左移一位 |
| +1.0011 | | +$[-Y^*]_{补}$ |
| 0.0101 | | 余数> 0, 商上1 |
| ←0.1010 | 0.0011 | 左移一位 |
| +1.0011 | | +$[-Y^*]_{补}$ |
| 1.1101 | | 余数< 0, 商上0 |
| +0.1101 | | 恢复余数 +$Y^*$ |
| 0.1010 | | |
| ←1.0100 | 0.0110 | 左移一位 |
| +1.0011 | | +$[-Y^*]_{补}$ |
| 0.0111 | ←0.1101 | 余数> 0, 商上1 |

# 原码加减交替除法（不恢复余数法）

**算法推导：**

1. 若第 i 次求商，部分余数为 $R_i$，

   若$R_i \geq 0$，则商上1，跳至2；

   若$R_i < 0$，则商0，恢复余数为正且左移得 $2(R_i + Y^*)$，跳至3；

2. 第 i + 1次求商，部分余数为 $R_{i+1} = 2R_i - Y^*$

3. 第 i + 1次求商，部分余数为 $R_{i+1} = 2(R_i + Y^*) - Y^*$

   $= 2R_i + Y^*$

结论：当某步试减操作不够减时，并不需要立即恢复余数，只要下一步试减直接做加法即可。

由于不需要恢复余数，且加减运算交替进行，故称此算法为“原码加减交替除法”或“不恢复余数除法”。

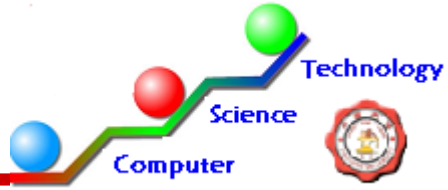
# 原码加减交替除法算法规则

- 参加运算的数均为原码，两数符号位不参加运算，取其绝对值做无符号数除法；
- 商的符号位单独处理，由两数符号“异或”产生；
- 除法运算前检测0：只要X、Y有任意一个为0，不进行实际运算；

  若$X^*$为0，则商为0；若$Y^*$为0，按非法除数处理；
- 运算前应满足条件：$X^* < Y^*$，否则按溢出处理；
- 设商的初值为0（被除数单倍字长）或为被除数低位部分（被除数双倍字长）；

  部分余数的初值为被除数（被除数单倍字长）或为被除数高位部分（被除数双倍字长）。
- 部分余数采用单符号位，且符号位初值为0（取$X^*$）；

# 原码加减交替除法算法规则




## (7) 部分余数运算：

试减 $X^*+[-Y^*]_{补}$

- 余数为正，商上1，余数和商左移一位，减除数；
- 余数为负，商上0，余数和商左移一位，加除数；

重复n步

(8) 第n+1步：若余数为正，商上1，商左移一位，余数不移位，运算结束；

若余数为负，商上0，商左移一位，但当结果需要余数时，需恢复余数（余数不移位直接加除数）。

$R^* = R_n \times 2^{-n}$。

# 原码加减交替除法举例

X = -0.1011

Y = - 0.1101

求： X÷Y=?

解： $[X]_{原}$= 1.1011

$[Y]_{原}$= 1.1101

$X^*$ = 0.1011

$Y^*$ = 0.1101

$[-Y^*]_{补}$ = 1.0011

$[X÷Y]_{原}$= 0.1101

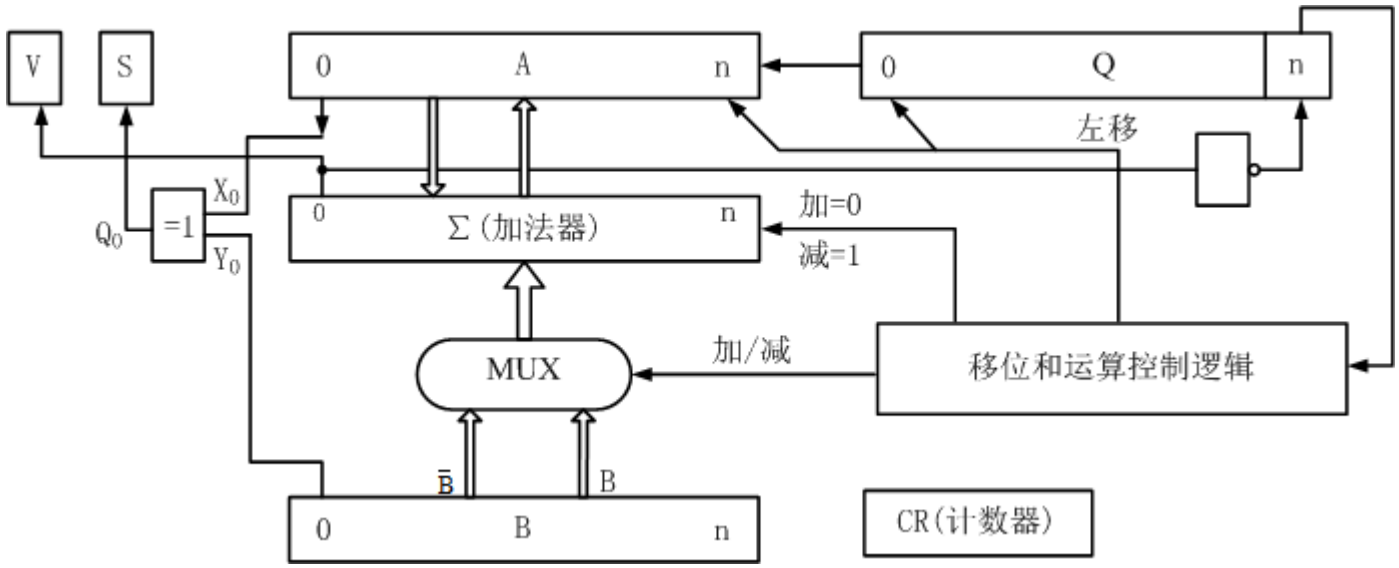
X÷Y= 0.1101

$R^*=0.0111\times2^{-4}$

| 被除数（余数） | 商 | 说　明 |
|---|---|---|
| 0. 1 0 1 1 | 0. 0 0 0 0 | |
| + 1. 0 0 1 1 | | + $[-Y^*]_{补}$ (减除数) |
| 1. 1 0 1 1 | | 余数< 0, 商上0 |
| ←1. 0 1 1 0 | 0. 0 0 0 0 | 左移一位 |
| + 0. 1 1 0 1 | | + $Y^*$ |
| 0. 1 0 0 1 | | 余数> 0, 商上1 |
| ←1. 0 0 1 0 | 0. 0 0 0 1 | 左移一位 |
| + 1. 0 0 1 1 | | + $[-Y^*]_{补}$ |
| 0. 0 1 0 1 | | 余数> 0, 商上1 |
| ←0. 1 0 1 0 | 0. 0 0 1 1 | 左移一位 |
| + 1. 0 0 1 1 | | + $[-Y^*]_{补}$ |
| 1. 1 1 0 1 | | 余数< 0, 商上0 |
| ←1. 1 0 1 0 | 0. 0 1 1 0 | 左移一位 |
| + 0. 1 1 0 1 | | + $Y^*$ |
| 0. 0 1 1 1 | | 余数> 0, 商上1 |
| | ← 0. 1 1 0 1 | 商左移一位 |

# 原码加减交替除法的实现

## 原码加减交替除法运算器




- 原码加减交替除法同样适用于**整数**，但**被除数应采用双倍字长**运算，否则当单字长被除数的绝对值小于除数时，得不到整数商。

# 补码加减交替除法（不恢复余数法）

- 算法基本思想同原码加减交替除法。
- 主要区别：

  被除数和除数都用补码参加运算，直接补码相除；

  求出反码商，再修正为近似的补码商；

  余数也是补码形式，商符通过运算自动产生。

- 需要解决的问题：

  商值的确定；

  求商符；

  获得新余数；

  商的校正。

# 补码加减交替除法算法分析

## (1) 商和新余数的确定

上商的过程实际上是比较被除数（余数）和除数所对应绝对值大小的过程。

**若X与Y同号：做减法，即** $[R_0]_{补}=[X]_{补}+[-Y]_{补}$

够减： $R_0$ **与Y 同号，商上1，左移一位，减除数；**

即 $[R_1]_{补}=2[R_0]_{补}+[-Y]_{补}$

不够减： $R_0$ **与Y 异号，商上0，左移一位，加除数。**

即 $[R_1]_{补}=2[R_0]_{补}+[Y]_{补}$

# 补码加减交替除法算法分析

- 若X与Y异号：做加法，即 $[R_0]_{补}=[X]_{补}+[Y]_{补}$

  够减： $R_0$ 与Y异号，商上0，左移一位，加除数；

  即 $[R_1]_{补}=2[R_0]_{补}+[Y]_{补}$

  不够减： $R_0$ 与Y同号，商上1，左移一位，减除数。

  即 $[R_1]_{补}=2[R_0]_{补}+[-Y]_{补}$

- 分析得出的结论：

  $R_i$ 与Y同号，商上1， $[R_{i+1}]_{补}=2[R_i]_{补}+[-Y]_{补}$

  $R_i$ 与Y异号，商上0， $[R_{i+1}]_{补}=2[R_i]_{补}+[Y]_{补}$

- 注：不能再用 $R_i>0$ （够减）作为上商的依据；

  改用判断 $R_i$ 与Y同/异号来决定上商和下一步加/减。

# 补码加减交替除法算法分析

## (2) 求商符

由于运算前，满足 $0 < |X| < |Y|$

所以第一次试减后的上商结果为：

X与Y同号时：若不够减，$R_0$ 与Y异号，商上0；

X与Y异号时：若不够减，$R_0$ 与Y同号，商上1 。

故商符由第一次试减后自动形成，且与商的数值位上商规律相同。

## (3) 商的校正

**从前面的分析知：当商为负时，算法以反码形式上商，而按补码运算要求，应得商的补码，两者之间相差$2^{-n}$。**

**为了最终得到补码商，通常采用商末位“恒置1”的舍入方法进行处理。采用这种方法产生的误差为：$\pm 2^{-n}$。**

**此方法简单、便于实现，运算后不需对商专门校正。**

# 补码加减交替除法算法规则

- (1) 操作数均为补码表示，两数的符号位参加运算；
  商符由运算自动产生；
- (2) 除法运算前先检测0：
  只要X、Y有任意一个为0，不进行实际运算；
  若X为0，则商为0；
  若Y为0，按非法除数处理；
- (3) 运算前应满足条件：X*<Y*，否则按溢出处理；
- (4) 设商的初值为0（被除数单倍字长）或为被除数低位部分（被除数双倍字长）；
  部分余数的初值为被除数（被除数单倍字长）或为被除数的高位部分（被除数双倍字长）；
- (5) 为防止运算过程中产生暂时性的溢出，部分余数可采用单符号位或双符号位运算，余数也是补码形式；

# 补码加减交替除法算法规则

❒ (6) 部分余数运算：

试减：
- X、Y同号，做减法：$[X]_{补}+[-Y]_{补}$
- X、Y异号，做加法：$[X]_{补}+[Y]_{补}$

上商：
- 余数与Y同号，商上1，余数和商左移一位，做减法；
- 余数与Y异号，商上0，余数和商左移一位，做加法；

重复n步

❒ (7) 第n+1步：余数不移位，商单独左移一位恒置1；

若不需要求余数，除法结束；

若要求求余数：
- 余数与X同号，除法结束。
- 余数与X异号，应恢复余数；

恢复余数方法：
- 若余数与Y同号，做减法。
- 若余数与Y异号，做加法。

# 补码加减交替除法举例

X = - 0.1001

Y = +0.1010

求： X÷Y=?

解： $[X]_{补}$ =1.0111

$[Y]_{补}$ =0.1010

$[-Y]_{补}$=1.0110

$[X \div Y]_{补}$= 1.0001

$[R]_{补}$= $1.1100 \times 2^{-4}$

X÷Y= - 0.1111

| 被除数（余数） | 商 | 说　　明 |
|---|---|---|
| 1 1. 0 1 1 1 | 0. 0 0 0 0 | X与Y异号 |
| + 0 0. 1 0 1 0 | | $+[Y]_{补}$ |
| 0 0. 0 0 0 1 | | 余数与Y同号, 商上1 |
| ← 0 0. 0 0 1 0 | 0. 0 0 0 1 | 左移一位 |
| + 1 1. 0 1 1 0 | | $+[-Y]_{补}$ |
| 1 1. 1 0 0 0 | | 余数与Y异号, 商上0 |
| ← 1 1. 0 0 0 0 | 0. 0 0 1 0 | 左移一位 |
| + 0 0. 1 0 1 0 | | $+[Y]_{补}$ |
| 1 1. 1 0 1 0 | | 余数与Y异号, 商上0 |
| ← 1 1. 0 1 0 0 | 0. 0 1 0 0 | 左移一位 |
| + 0 0. 1 0 1 0 | | $+[Y]_{补}$ |
| 1 1. 1 1 1 0 | | 余数与Y异号, 商上0 |
| ← 1 1. 1 1 0 0 | 0. 1 0 0 0 | 左移一位 |
| + 0 0. 1 0 1 0 | | $+[Y]_{补}$ *(如不求余数可省)* |
| 0 0. 0 1 1 0 | ← 1. 0 0 0 1 | *商左移一位末位恒置1* |
| + 1 1. 0 1 1 0 | | $+[-Y]_{补}$ *(余数与X异号, 需恢复余数)* |
| 1 1. 1 1 0 0 | | |

# 补码加减交替除法的实现

- 在$X^* \geq Y^*$时加减交替除法可正常进行，第一次上商为溢出的数值而不是补码的符号位。此时可改算前判溢出为算后判溢出。方法：
  - 采用双符号位商，真符位可由X、Y的符号位“异或”产生，算后可利用双符号位补码的溢出判断方法判溢出。

- 补码加减交替除法与原码加减交替除法一样也适用于整数。被除数同样需采用双倍字长参加运算，否则不能保证得到的是整数商。

- 补码加减交替除法的运算器框图与原码除法运算器的基本结构相似，只是加减和上商的条件不同，控制逻辑略需修改而已。如果补码采用双符号位运算，相应的加/减法电路和寄存器部件的宽度还需增加到n+2位。此时最左边的一位符号位为真符位，在运算过程中永远保持着部分余数正确的符号。

# 原码、补码加减交替除法规则对照表

| 除法类型 | 符号位参与运算 | 加减次数 | 移位 | | 备　注 |
|---|---|---|---|---|---|
| | | | 方向 | 次数 | |
| 原码加减交替法 | 否 | n+1或n+2 | 左 | n | 若最终余数为负，需恢复余数 |
| 补码加减交替法 | 是 | n+1或n+2 | 左 | n | 末位恒置1<br>若最终余数与被除数异号，需恢复余数 |

**注：n为除数的数值部分的位数**

# 5.3.6 阵列乘除法器

- 已经讲过的原码或补码一位乘除法运算，采用的是**逐位循环迭代算法**，完成时间直接取决于数据的位数，当位数较多时，用时较长。
- 提高乘除法运算速度的出路之一， 是每次完成多位运算，用的较多的是**两位乘法**，三位乘法、 **跳零跳一除法**等。
- 目前使用较多的是采用**阵列乘法器**、**阵列除法器**实现快速乘除法运算。

# 阵列乘法器

## 基本思路

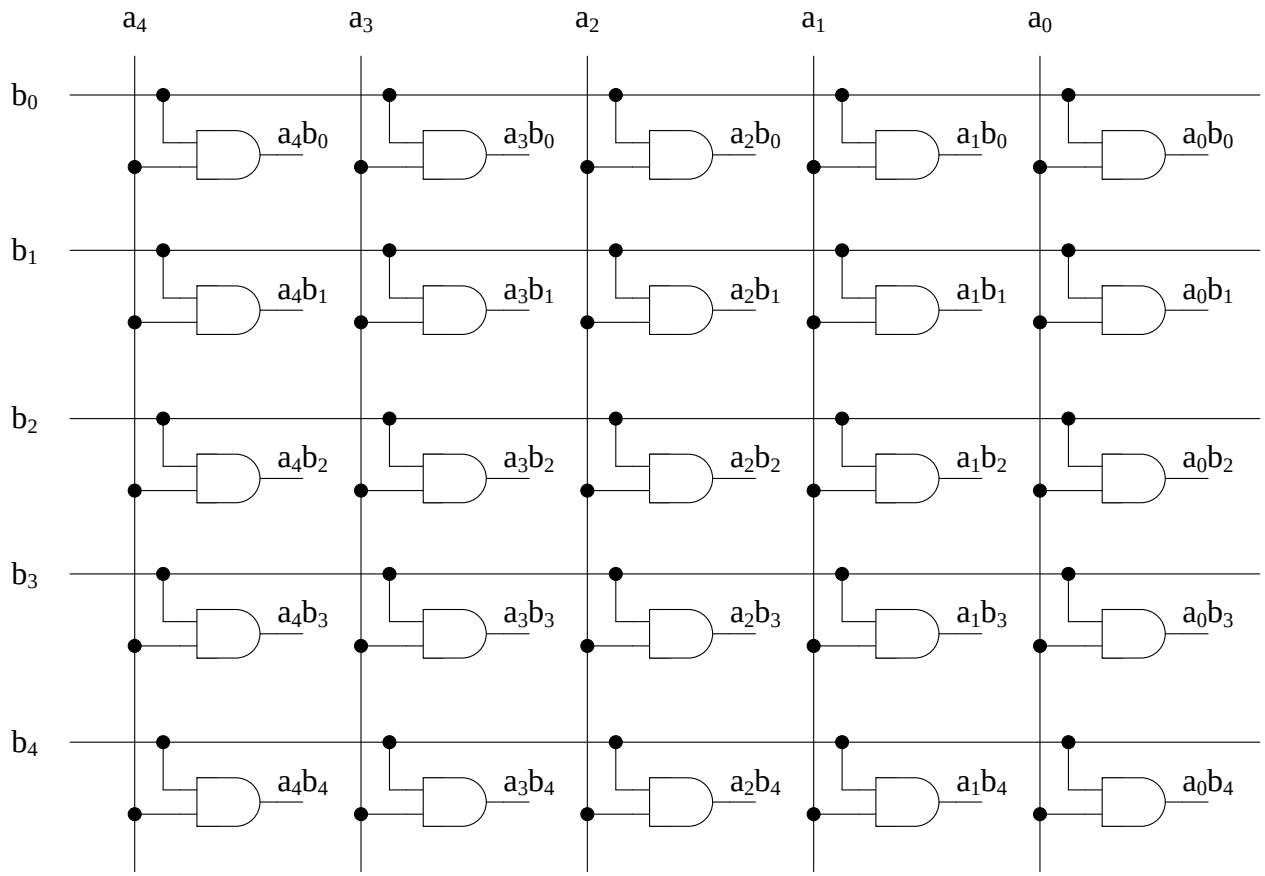
- 利用全加器阵列，完全由硬件直接计算乘法结果
- 以 5 位无符号整数为例

$$
\begin{array}{cccccccccc}
 & & & & a_4 & a_3 & a_2 & a_1 & a_0 & =A \\
 & & & \times & b_4 & b_3 & b_2 & b_1 & b_0 & =B \\
\hline
 & & & & a_4b_0 & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & \\
 & & & a_4b_1 & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & & \\
 & & a_4b_2 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & & \\
 & a_4b_3 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & & \\
+ & a_4b_4 & a_3b_4 & a_2b_4 & a_1b_4 & a_0b_4 & & & & \\
\hline
P_9 & P_8 & P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0 \quad =P
\end{array}
$$

其中 $P_{ij} = a_i \times b_j$ —— 位积

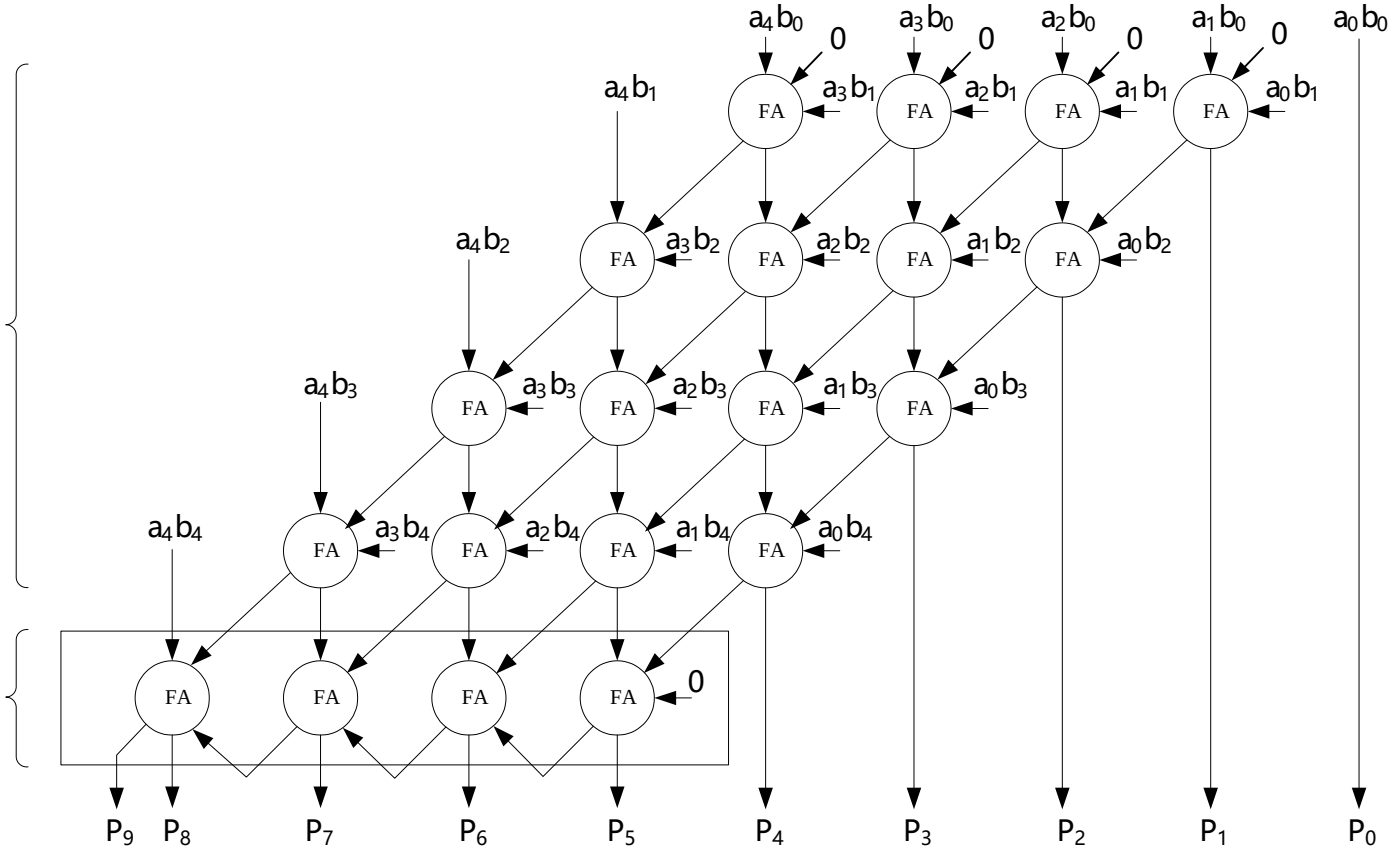
# 阵列乘法器

- 位积产生电路：与门阵列，以5×5为例



# 阵列乘法器

## 5×5位无符号数阵列乘法器



# 阵列乘法器

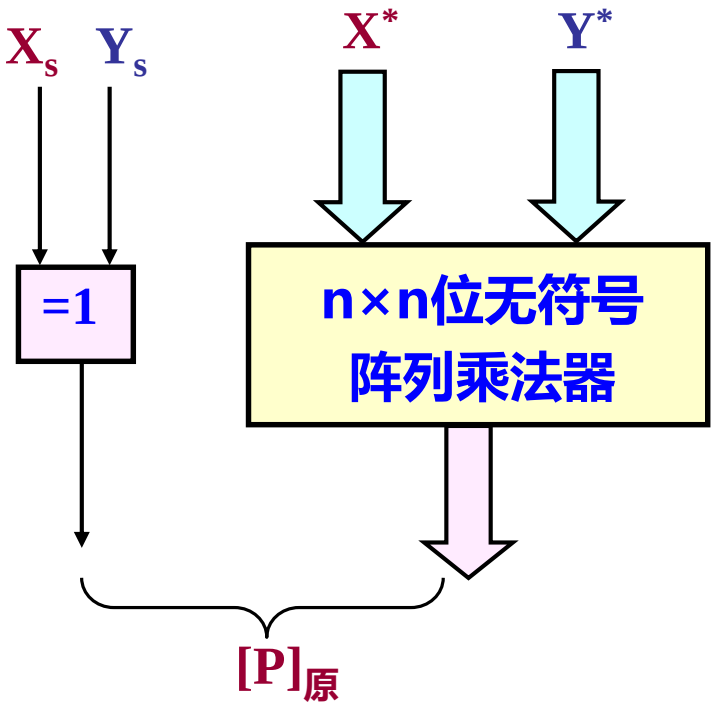
- **保留进位加法：本级的两个数相加，产生的进位信号不马上传递到本级加法器的高一位来参加本级的加法过程，而是暂时予以保留，以便参加下一级加法器高一位相加的加法。**
- **保留进位加法是一种特殊的快速加法，可以减少加法的进位传递时间，阵列乘法器中经常用到。**
- **相应的加法器则称为保留进位加法器（Carry Save Adder，CSA）。CSA与普通加法器的区别仅在于全加器进位输入输出端的连接方法上。**
- **普通的串行进位加法器则缩写为CPA（Carry Propagate Adder）。**

# 原码阵列乘法器

设 被乘数 $[X]_{原} = X_s . X_1X_2 \cdots X_n$

乘 数 $[Y]_{原} = Y_s . Y_1Y_2 \cdots Y_n$

则 积 $[P]_{原}=[X \times Y]_{原} =( X_s \oplus Y_s ) . ( X^* \times Y^* )$



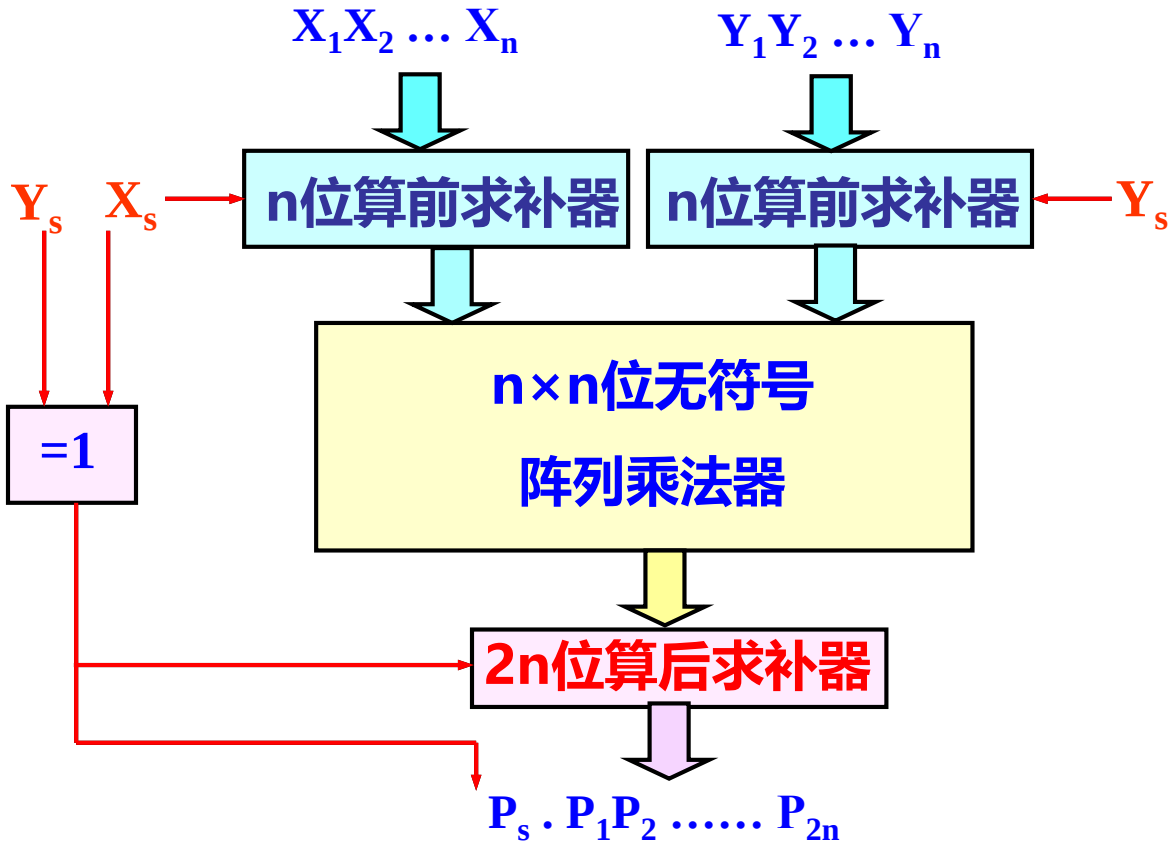
# 带求补器的补码阵列乘法器

图中给出了带求补器的补码阵列乘法器。更好的方法是利用加/减法阵列电路直接实现Booth算法，完成两个补码的比较相乘过程。限于课时不做讨论。

设 被乘数 $[X]_{补} = x_f . x_1 x_2 \cdots x_n$

乘 数 $[Y]_{补} = y_f . y_1 y_2 \cdots y_n$

则 $[P]_{补} = [X \times Y]_{补} = P_s . P_1 P_2 \cdots P_{2n}$



# 阵列除法器

## ❒ 可控加/减法单元CAS

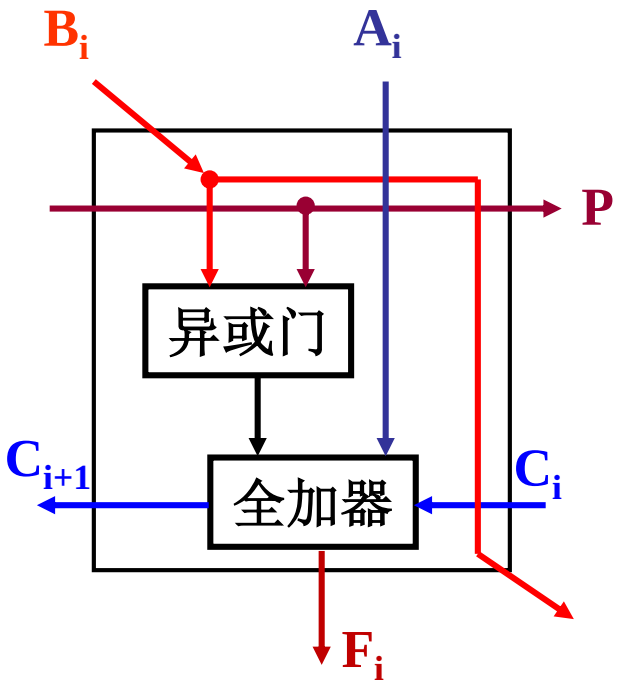
$$\left.\begin{aligned} F_i &= A_i \oplus (B_i \oplus P) \oplus C_i \\ C_{i+1} &= A_i C_i + (A_i + C_i)(B_i \oplus P) \end{aligned}\right\}$$

**□ P = 0 时，输入的操作数不求补，控制做加法**

$$\left.\begin{aligned} F_i &= A_i \oplus B_i \oplus C_i \\ C_{i+1} &= A_i B_i + (A_i + B_i) C_i \end{aligned}\right\}$$

**□ P = 1 时，输入的操作数求补，控制做减法**

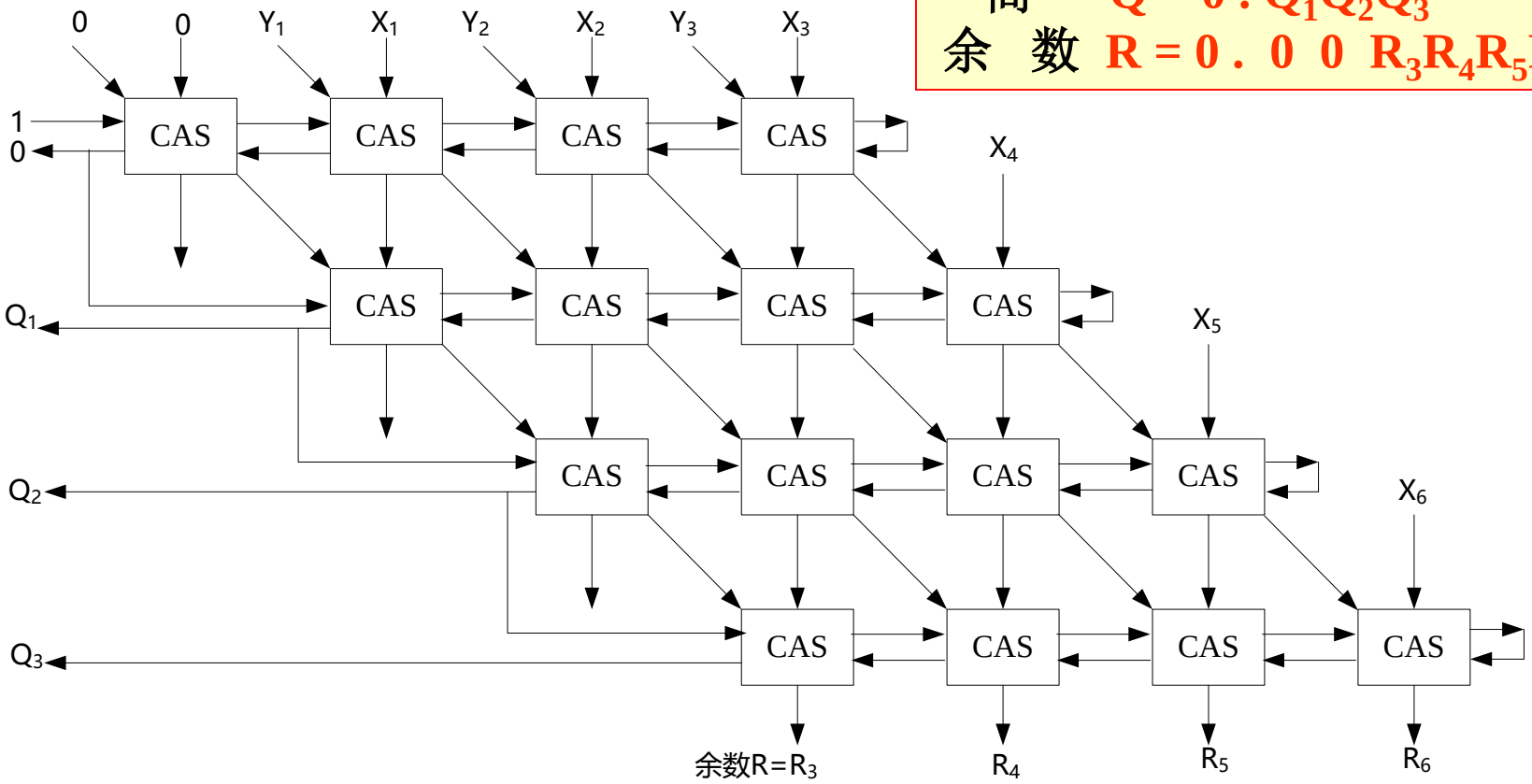
$$\left.\begin{aligned} F_i &= A_i \oplus \overline{B}_i \oplus C_i \\ C_{i+1} &= A_i \overline{B}_i + (A_i + \overline{B}_i) C_i \end{aligned}\right\}$$




可控加减法单元CAS

# 加减交替阵列除法器——3位小数为例

被除数 $X = 0.X_1X_2X_3X_4X_5X_6$
除　数 $Y = 0.Y_1Y_2Y_3$
　商　 $Q = 0.Q_1Q_2Q_3$
余　数 $R = 0.\ 0\ 0\ R_3R_4R_5R_6$



# 加减交替阵列除法器

阵列除法过程：

**第一步**：试减，P=1，实现 $X^*+[-Y^*]_{补}$ 。

因为X*<Y*, 所以一定不够减，则最高位进位$C_{i+1}$ =0，

利用此进位输出产生商和下一步的P（= $C_{i+1}$）。

**第二步**：P=0，做 $X^*+Y^*$ 。

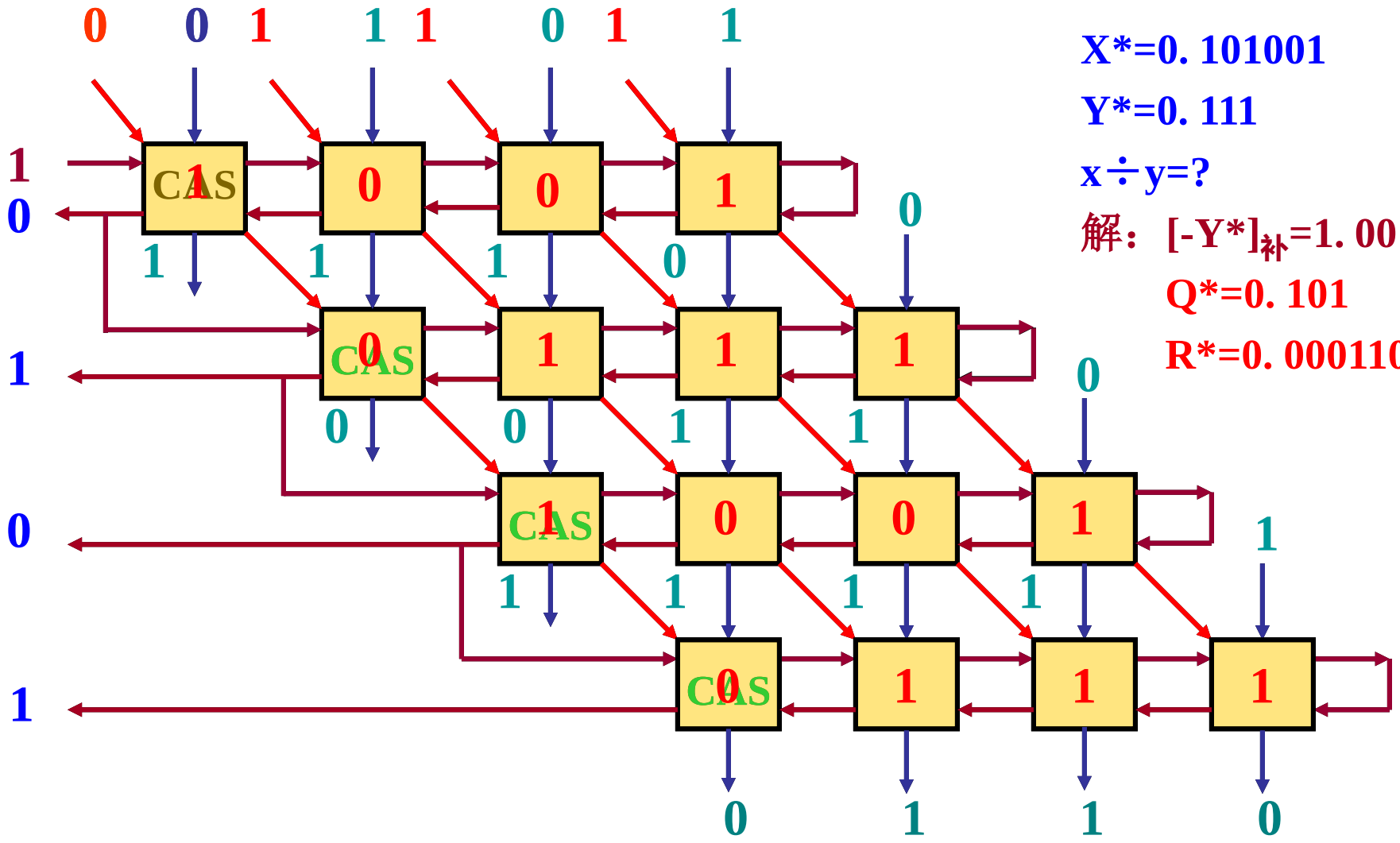
当最高位进位$C_{i+1}$ =1时，表示够减，则 $q_1$ = 1, P =1;

当最高位进位$C_{i+1}$ =0时，表示不够减，则 $q_1$ = 0, P =0。

**第三步和第四步**：P=0时,做 $X^*+Y^*$；P=1时,做 $X^*+[-Y^*]_{补}$ 。

上商和P值产生的规则与第二步相同。

# 加减交替阵列除法器举例




X*=0. 101001

Y*=0. 111

x÷y=?

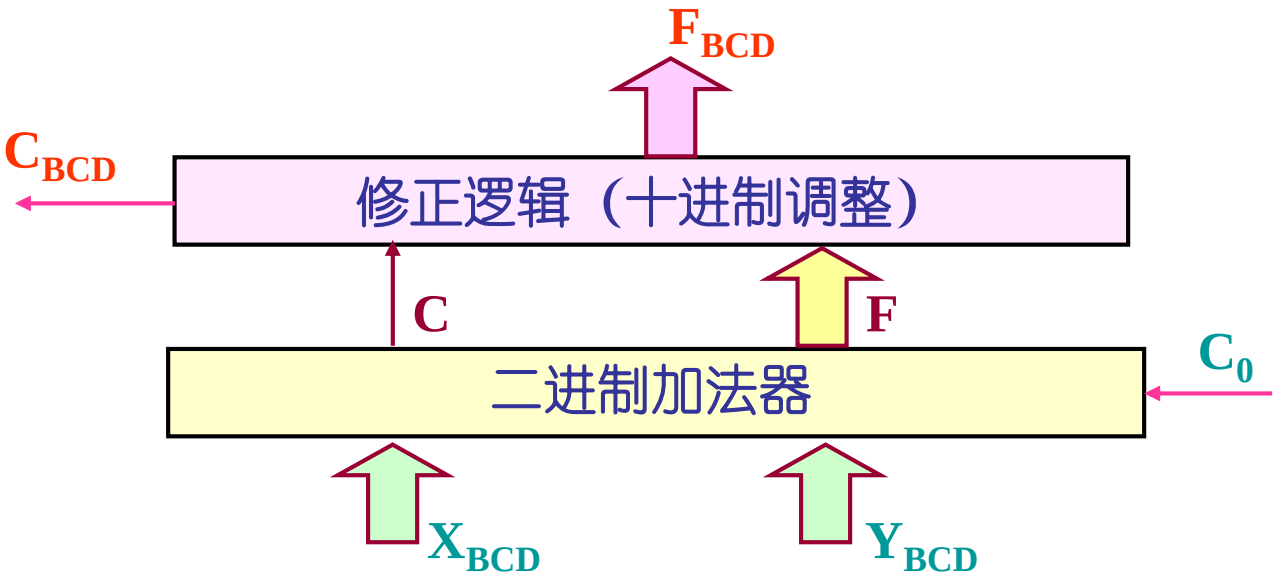
解：$[-Y^*]_{补}$=1. 001

Q*=0. 101

R*=0. 000110

# 5.3.7 十进制运算

- 十进制运算——直接对BCD码进行的运算。
- 十进制加法实现思想
  - 将BCD码看成二进制数，进行二进制加法；
  - 对二进制和、进位进行修正，转换成BCD码。
  - 一位十进制加法单元组成框图如下：



# 一位8421码加法

- **8421码加法修正规则：**
  - C=0，且F = 0000~1001，结果不必修正；
  - C=0，且F = 1010~1111(非码)，加6修正；

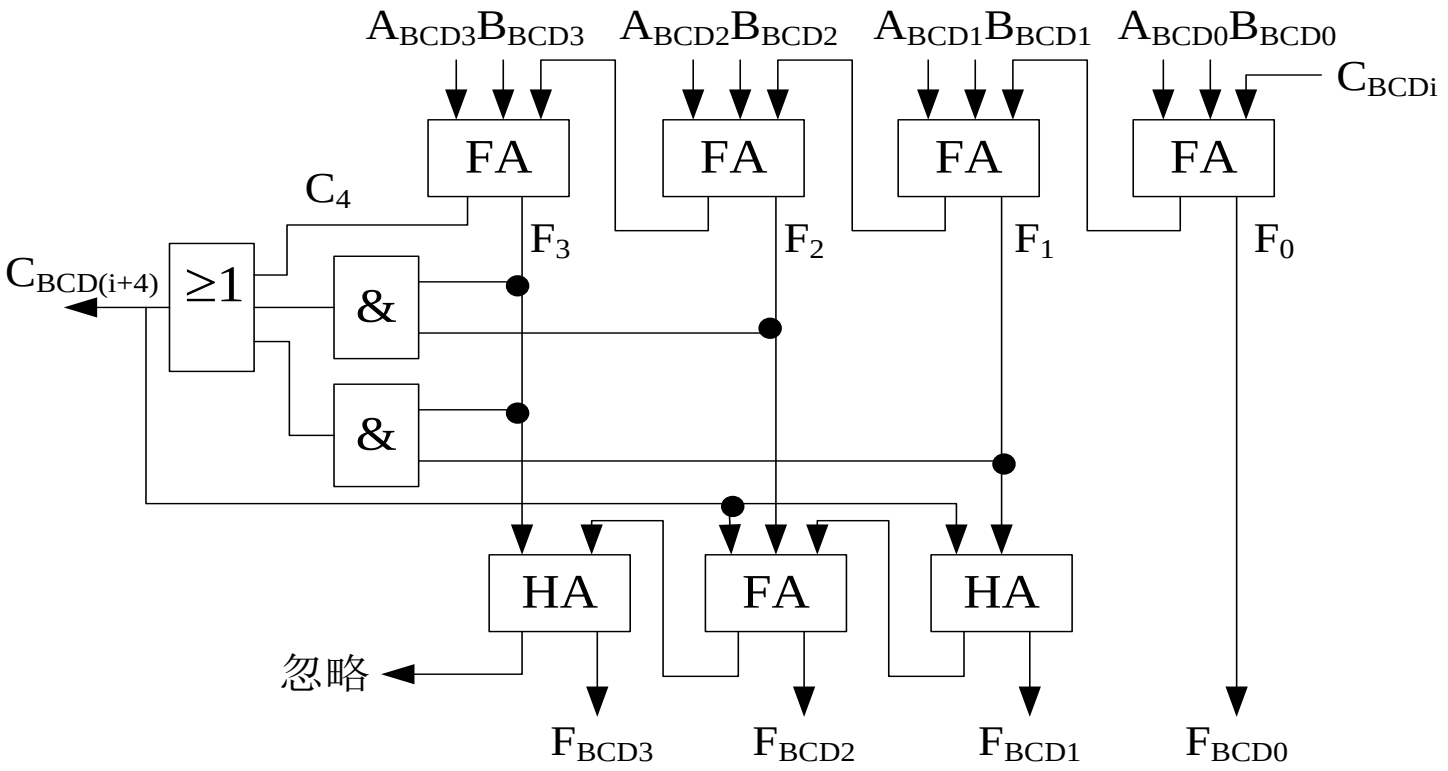
    即 $F_{BCD} = F + 0110$，$C_{BCD} = 1$
  - C=1，加6修正。即 $F_{BCD} = F + 0110$，$C_{BCD} = C$
- **8421码和的修正关系表：**见下页。
- **8421码的进位生成逻辑**（由关系表得）：

  $C_{BCD}=C+F_3F_2+F_3F_1$
- **一位8421码加法单元逻辑电路：**见下下页。

# 一位8421码和的修正关系表

| 十进制数 | 8421码和<br>$C_{BCD}\ F_{BCD3}\ F_{BCD2}\ F_{BCD1}\ F_{BCD0}$ | 校正前的和<br>$C_4\ F_3\ F_2\ F_1\ F_0$ | 校正关系 |
|---|---|---|---|
| 0~9 | 0 0 0 0 0<br>……<br>0 1 0 0 1 | 0 0 0 0 0<br>……<br>0 1 0 0 1 | 不校正 |
| 10 | 1 0 0 0 0 | 0 1 0 1 0 | +6校正 |
| 11 | 1 0 0 0 1 | 0 1 0 1 1 | |
| 12 | 1 0 0 1 0 | 0 1 1 0 0 | |
| 13 | 1 0 0 1 1 | 0 1 1 0 1 | |
| 14 | 1 0 1 0 0 | 0 1 1 1 0 | |
| 15 | 1 0 1 0 1 | 0 1 1 1 1 | |
| 16 | 1 0 1 1 0 | 1 0 0 0 0 | |
| 17 | 1 0 1 1 1 | 1 0 0 0 1 | |
| 18 | 1 1 0 0 0 | 1 0 0 1 0 | |
| 19 | 1 1 0 0 1 | 1 0 0 1 1 | |

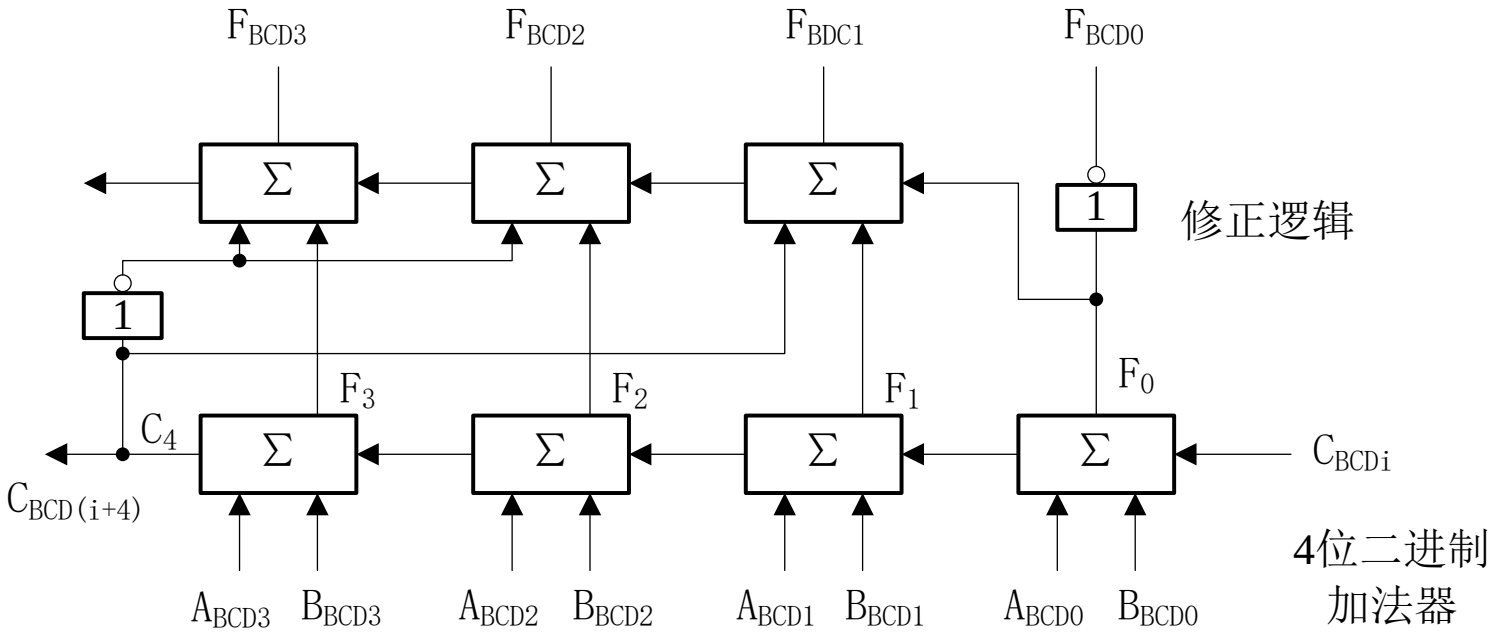
# 一位8421码加法单元




**结构特点：由一级四位二进制加法器、一级简化的四位二进制+6加法器和进位产生逻辑三部分组成。**

# 一位余3码加法

- **余3码加法修正规则：**
  - **C=0，减3修正。即 $F_{BCD} = F + 1101$, $C_{BCD} = C$**
  - **C=1，加3修正；即 $F_{BCD} = F + 0011$, $C_{BCD} = C$**
  - **一位余3码加法单元逻辑图：**



# 一位余3码和的修正关系表

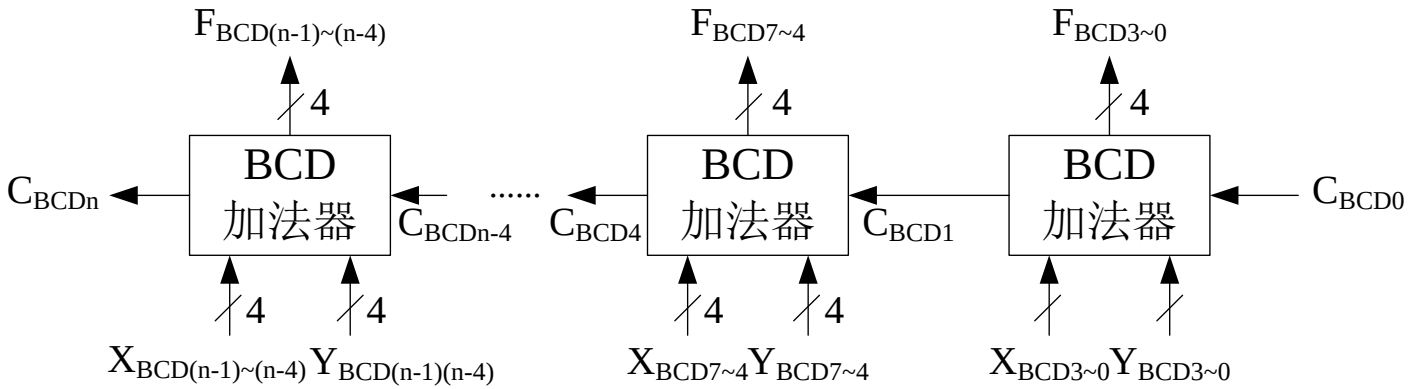
| 十进制数 | 余3码和 $C_{BCD}$ | $F_{BCD3}$ | $F_{BCD2}$ | $F_{BCD1}$ | $F_{BCD0}$ | 校正前的和 $C_4$ | $F_3$ | $F_2$ | $F_1$ | $F_0$ | 校正关系 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | -3校正 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | |
| 2 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | |
| 3 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | |
| 4 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | |
| 5 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | |
| 6 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | |
| 7 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | |
| 8 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | |
| 9 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | |
| 10 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | +3校正 |
| 11 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | |
| 12 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | |
| 13 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | |
| 14 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | |
| 15 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | |
| 16 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | |
| 17 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | |
| 18 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | |
| 19 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | |

# n位十进制加法器

- n位的十进制加法以一位十进制加法为基础实现。
- n位加法规则

  (1) BCD码位内的相加，按一位BCD码加法及修正规则进行；

  (2) n位BCD码并行相加，位与位之间的十进制进位信号串行传递（最简进位方式）。
- n位串行进位的十进制并行加法器示意图：



# 8421码加法举例

- 【例6.51】用8421码加法求57+48=?
- 解：$(57)_{BCD}$ = 0101，0111；$(48)_{BCD}$ = 0100，1000

  为清楚起见，这里将各位BCD码之间用逗号隔开。

```
    0101，0111  ——57
+   0100，1000  —— 48
------------------------
    1001，1111  ——低位无进位
+       0110    —— +6修正
------------------------
      1，0101   —— 产生出低位进位
    1010，0101  —— 低位进到高位，高位无进位
+   0110        —— +6修正
------------------------
0001，0000，0101 —— 105
```

# 余3码加法举例

- 【例6.52】用余3码求316+258=?
- 解：$(316)_{E3}$ = 0110，0100，1001；

  $(258)_{E3}$ = 0101，1000，1011

| | |
|---|---|
| 0110，0100，1001 | —— 316 |
| + 0101，1000，1011 | —— +258 |
| 1011，1101，0100 | |
| + 1101，1101，0011 | —— 有进位+3（+0011），无进位-3（+1101） |
| 1000，1010，0111 | —— 574 |

# 5.3.8 基本的逻辑运算

- 特点：操作数和运算结果均为逻辑数，即无数值及位权关系的数；运算按位进行，位间不存在进位、借位等关系。

- 四种基本运算：大多数机器均具备四种基本逻辑运算，即逻辑非、逻辑加、逻辑乘和逻辑异。其他运算可以通过基本运算组合实现。

# 逻辑非（求反）

- 操作：按位求反。
- 运算符：“–”或“¬”，一元运算。
- 一位逻辑非运算规则：

| $X_i$ | $F_i$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

例：　　X= 10110100

则：Z= ¬X= 01001011

- 逻辑描述：

设 $X = X_0X_1\cdots X_n$

若 $Z = \overline{X} = Z_0Z_1\cdots Z_n$

则 $Z_i = \overline{X_i}$

# 逻辑加（逻辑或）

- 操作：按位“或”。
- 运算符：“∨”或“+”，二元运算。
- 一位二进制数的逻辑加运算规则为：

| $X_i$ ∨ $Y_i$ | $Z_i$ |
|---|---|
| 0 0 | 0 |
| 0 1 | 1 |
| 1 0 | 1 |
| 1 1 | 1 |

例：　X=10110100
　　Y=00101101
则：X+Y=10111101

- 逻辑描述：

设 $X = X_0X_1\cdots X_n$，$Y = Y_0Y_1\cdots Y_n$

若 $Z = X \vee Y = Z_0Z_1\cdots Z_n$

则 $Z_i = X_i \vee Y_i$

# 逻辑乘（逻辑与）

- 操作：按位“与”。
- 运算符：“∧”或“•”，二元运算。
- 一位二进制数的逻辑乘运算规则为：

| $X_i$ ∧ $Y_i$ | $Z_i$ |
|---|---|
| 0 0 | 0 |
| 0 1 | 0 |
| 1 0 | 0 |
| 1 1 | 1 |

例：X=10110100

Y=00101101

则：X∧Y=00100100

- 逻辑描述：

设 $X = X_0X_1\cdots X_n$，$Y = Y_0Y_1\cdots Y_n$

若 $Z = X \wedge Y = Z_0 Z_1\cdots Z_n$

则 $Z_i = X_i \wedge Y_i$

# 逻辑异（按位加）

- 操作：按位“加”、按位“异或”。
- 运算符：“⊕”，二元运算。
- 一位二进制数的逻辑异运算规则为：

| $X_i$ ⊕ $Y_i$ | $Z_i$ |
|---|---|
| 0 0 | 0 |
| 0 1 | 1 |
| 1 0 | 1 |
| 1 1 | 0 |

例：X=10110100

Y=00101101

则：X⊕Y=10011001

- 逻辑描述：

设 $X = X_0X_1\cdots X_n$，$Y = Y_0Y_1\cdots Y_n$

若 $Z = X \oplus Y = Z_0Z_1\cdots Z_n$

则 $Z_i = X_i \oplus Y_i$

# 组合逻辑运算举例

- 设 A=0011 1100，B=1100 1100
  C=0101 0101，D=1010 1010
- 计算 F= ¬ [(A⊕B∨C)D]=?
- 解：运算步骤如下：

| 第1步： | 00111100 | 第2步： | 11110000 | 第3步： | 11110101 | 第4步： | |
|---|---|---|---|---|---|---|---|
| ⊕ | 11001100 | ∨ | 01010101 | ∧ | 10101010 | ¬ | 10100000 |
| | 11110000 | | 11110101 | | 10100000 | | 01011111 |

F= ¬ [ （A⊕B∨C） D]

= ¬[(0011 1100⊕1100 1100∨0101 0101)∧1010 1010]

= 0101 1111

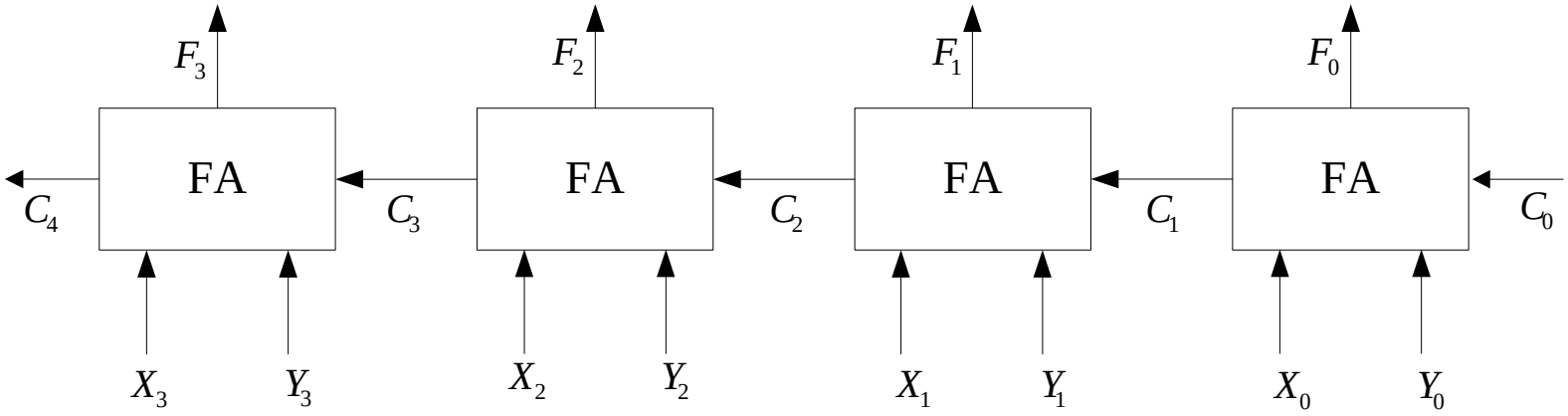
- 这个运算实例在计算机中实现时，通常需要由软、硬件配合完成。首先使用XOR、OR、AND、NOT等指令编程，然后在运行这段汇编语言程序时，再分别由各条指令去控制完成相应的硬件功能。

# 5.4 定点运算的实现

# 5.4.1 加法器的进位技术

## 1. 行波进位加法器

- 行波进位加法器——串行进位的并行加法器，最简单的并行加法器。n位行波进位加法器可由n个全加器通过进位输入输出端首尾相连组成，其链式的串行进位结构也叫作“串行进位链”。
- 四位行波进位加法器逻辑图



# 行波进位原理

- 行波进位逻辑

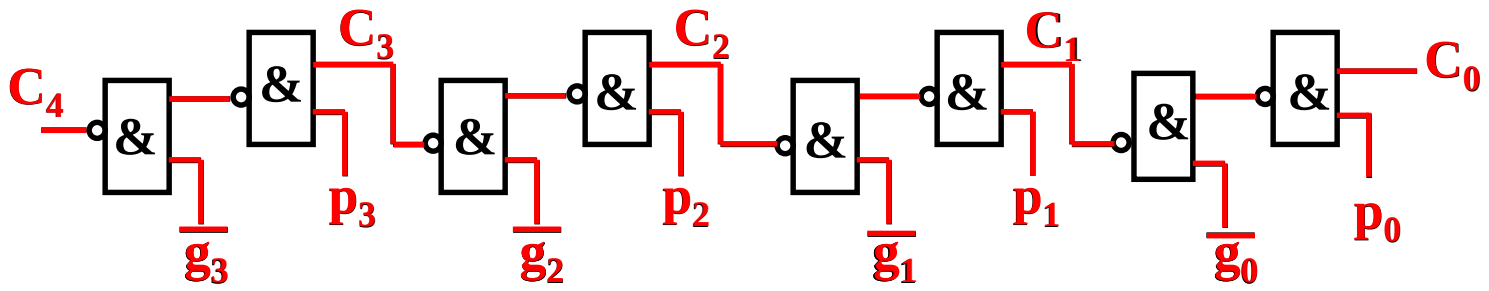
$$C_{i+1} = X_i Y_i + (X_i \oplus Y_i) C_i$$

- 为了简化全加器的进位逻辑，定义两个进位函数
  - 进位生成函数（carry generate）：$g_i = X_i Y_i$
  - 进位传递函数（carry propagate）：$p_i = X_i \oplus Y_i$
- 代入上式得：$C_{i+1} = g_i + p_i C_i$
- 四位行波进位加法器各位的进位逻辑

$$\left.\begin{aligned} C_1 &= g_0 + p_0 C_0 \\ C_2 &= g_1 + p_1 C_1 \\ C_3 &= g_2 + p_2 C_2 \\ C_4 &= g_3 + p_3 C_3 \end{aligned}\right\}$$

# 串行进位链(行波进位线路)

- 若采用与非逻辑电路，其进位链结构如下：




- **速度分析：**若设与非门的延迟时间为$t_y$，4位加法器进位时延则为**8ty**，n位加法器最长进位时间为$2nt_y$。n=16时，最长进位时间=$32t_y$。

- **特点：**进位延迟时间长，影响加法器速度。

# 先行进位原理

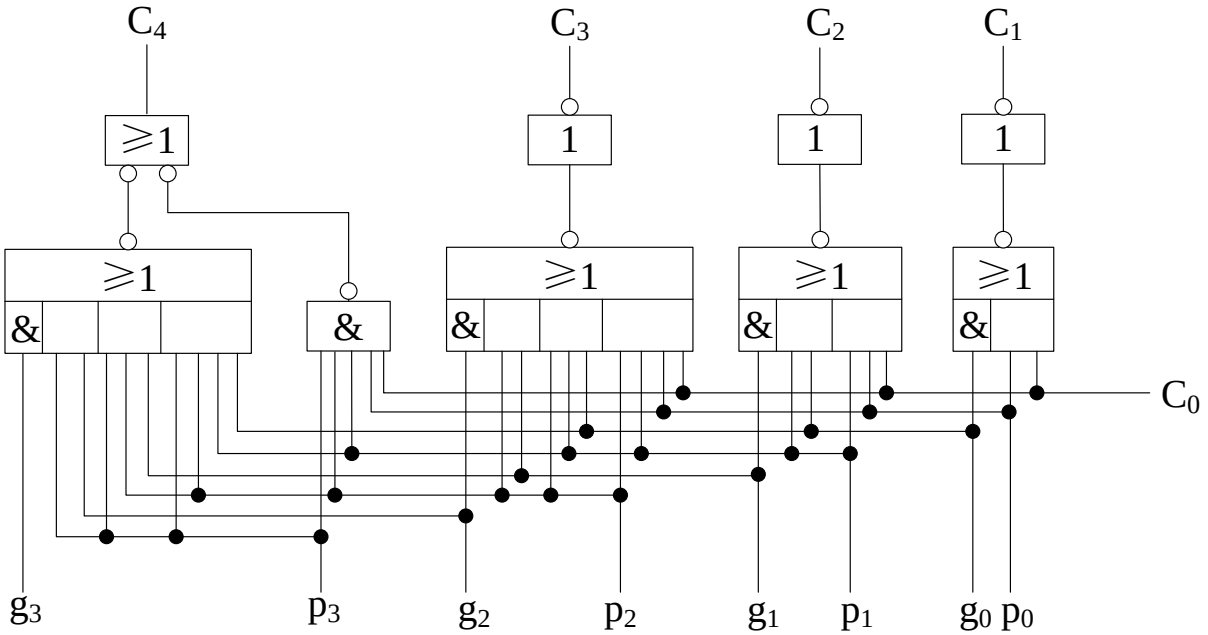
- **串行进位的特征：**每位进位输出都依赖于低位进位输入。
- **先行进位原理：**打破进位间依赖关系，直接由并行输入的加数产生进位信号，并行得到各位的进位。
- 仍以四位加法器为例，将前一位进位信号表达式直接代入本位，得到一组新的进位表达式：

$$
\left.
\begin{aligned}
C_1 &= g_0 + p_0 C_0 \\
C_2 &= g_1 + p_1 g_0 + p_1 p_0 C_0 \\
C_3 &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 C_0 \\
C_4 &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 C_0
\end{aligned}
\right\}
$$

- 这组表达式中各位进位信号直接由进位函数和最低位进位输入产生，而进位函数是直接由加数生成的，与低一位进位信号无关。

# 先行进位线路（CLA）

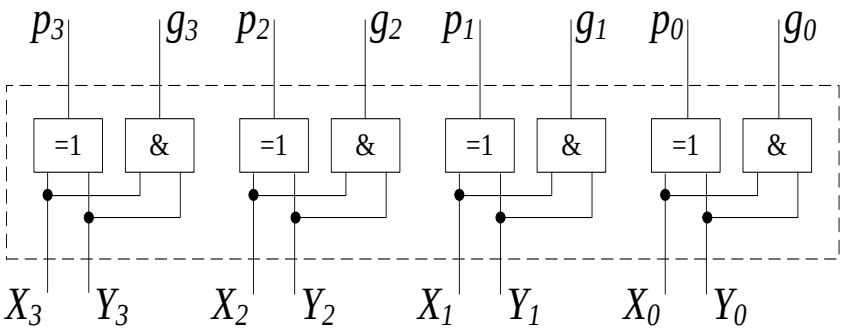
- 直接由原始操作数产生进位信号的技术，称作先行进位（Carry Look Ahead，CLA）或并行进位、同时进位、超前进位等。
- 由与或非-与非逻辑实现的四位先行进位线路（CLA部件）如图所示：



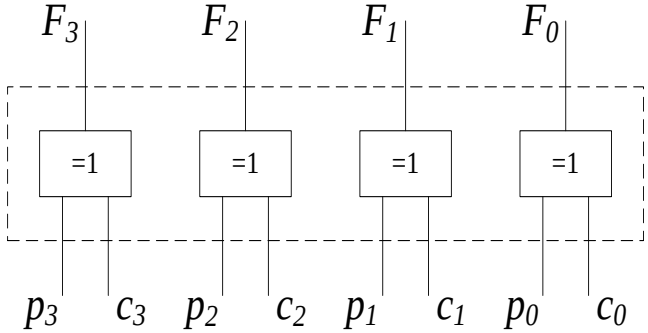

速度分析：假设与或非门时延为1.5ty，与非门时延仍为1ty，忽略产生进位函数的时延，则产生进位输出共需两级门2.5ty左右的时延。这个延迟时间是常数，各位进位都一样，与加法器的位数无关。

# 先行进位加法器

- 先行进位线路逻辑结构复杂，无法再直接使用全加器来搭建先行进位加法器。
- 一个完整的先行进位加法器可看成是由进位函数产生线路（进位生成/传递部件）、求和线路（求和部件）及先行进位线路（CLA部件）三部分构成。
- **四位先行进位加法器结构框图**

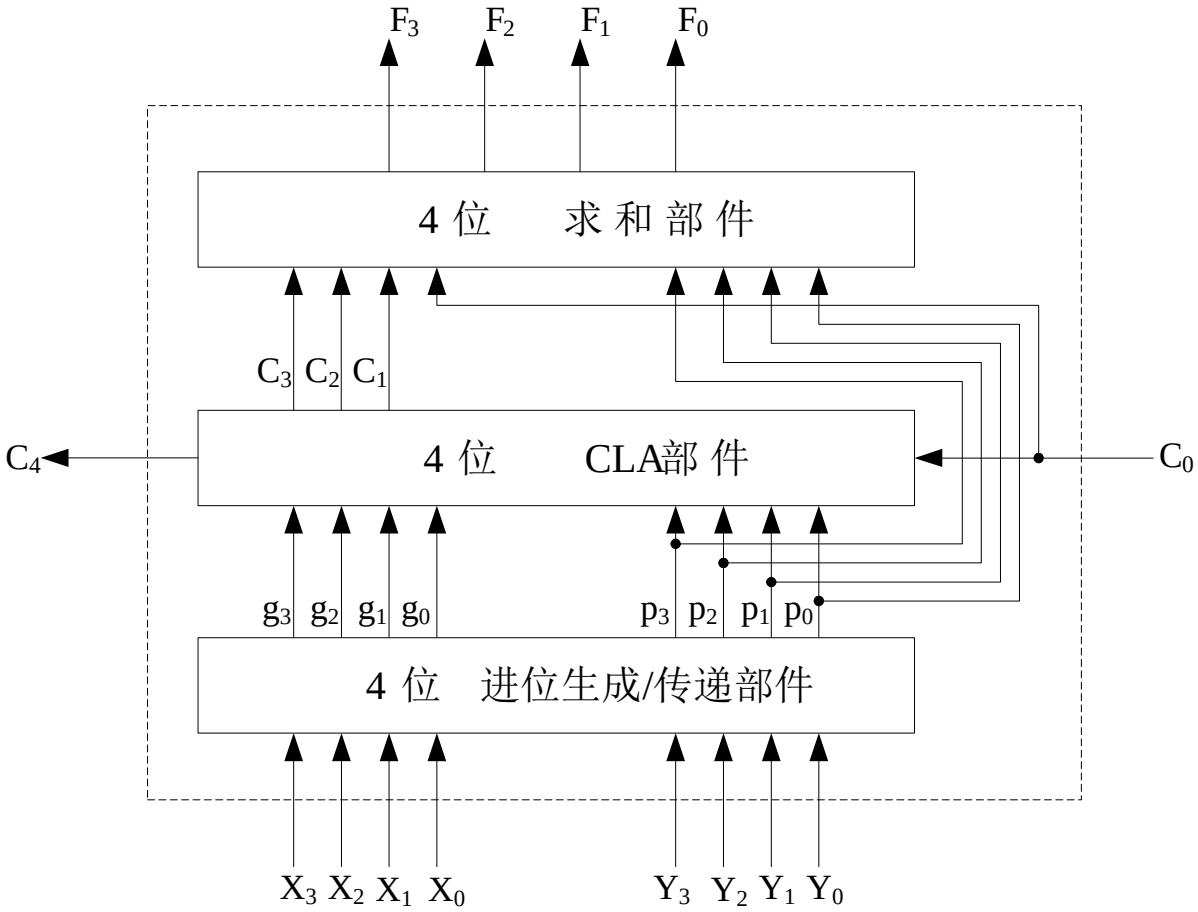



四位进位生成/传递部件




四位求和部件

# 四位先行进位加法器举例



# 多级先行进位技术

- 先行进位技术受到的限制：由于门电路扇入系数的影响，先行进位加法器的规模一般最多只能到8位左右。
- 改良方案
  1）成组先行-级联进位
    - 将n位并行加法器分成若干组，每组若干位。组内采用先行进位方式，组间采用串行进位（级联方式）。
  2）多级先行进位
    - 在分组先行进位的基础上，如果不仅组内先行进位，组间也采用先行进位，就构成二级先行进位加法器。
    - 加法器位数更多时还可采用三级先行进位，但当进位级数过多时，其加速作用会受到进位传递时延的抵消，实际应用时多级先行进位线路的级数不宜过多。

# 成组先行-级联进位

- **16位成组先行-级联进位加法器举例：**每4位为一组，共分4组。各组先行进位的通用表达式如下：

$$
\begin{aligned}
C_{n+1} &= g_n + p_n C_n \\
C_{n+2} &= g_{n+1} + p_{n+1} g_n + p_{n+1} p_n C_n \\
C_{n+3} &= g_{n+2} + p_{n+2} g_{n+1} + p_{n+2} p_{n+1} g_n + p_{n+2} p_{n+1} p_n C_n \\
C_{n+4} &= g_{n+3} + p_{n+3} g_{n+2} + p_{n+3} p_{n+2} g_{n+1} + p_{n+3} p_{n+2} p_{n+1} g_n + p_{n+3} p_{n+2} p_{n+1} p_n C_n
\end{aligned}
$$

- 式中n表示各组下标的增量，在4分组的情况下，n=0，4，8，12。$C_{n+1}$、$C_{n+2}$、$C_{n+3}$为小组内进位，$C_{n+4}$为小组间进位信号，在组间串行传递。

# 成组先行-级联进位逻辑

- 将n=0，4，8，12代入上述进位通式可得各组先行进位逻辑
  - 第0组

$$
\begin{aligned}
C_1 &= g_0 + p_0C_0 \\
C_2 &= g_1 + p_1g_0 + p_1p_0C_0 \\
C_3 &= g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0C_0 \\
C_4 &= g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0C_0
\end{aligned}
$$

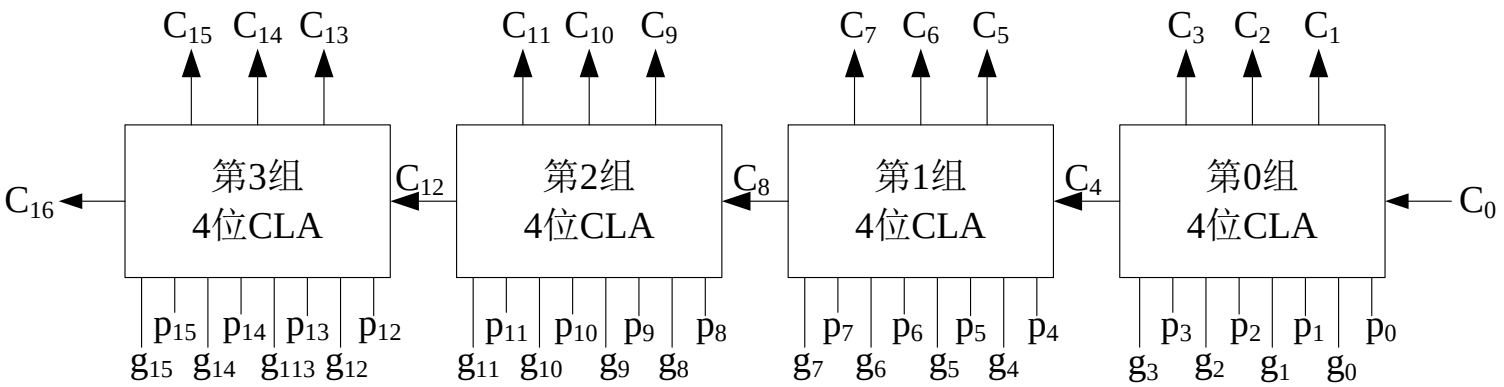
  - 第1组

$$
\begin{aligned}
C_5 &= g_4 + p_4C_4 \\
C_6 &= g_5 + p_5g_4 + p_5p_4C_4 \\
C_7 &= g_6 + p_6g_5 + p_6p_5g_4 + p_6p_5p_4C_4 \\
C_8 &= g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4 + p_7p_6p_5p_4C_4
\end{aligned}
$$

  - 第2组 ……

# 成组先行-级联进位线路

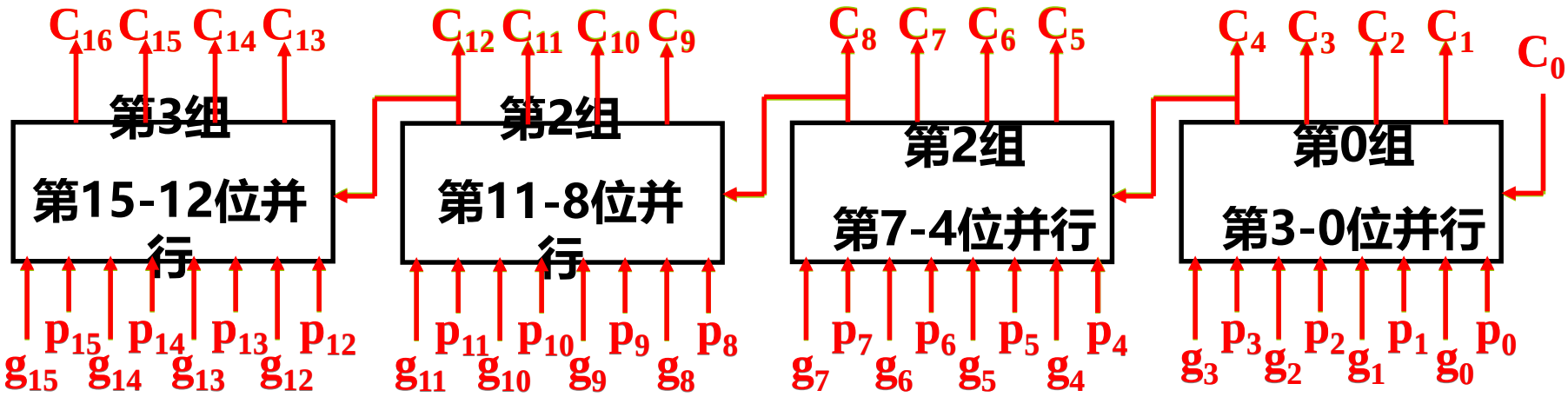
## ❒ 16位成组先行—级联进位框图




- ❒ **特点：二级进位结构，组内为第一级，用4位CLA部件实现；组间构成第二级，用串行进位方式实现。**
- ❒ **速度分析：组内先行进位线路时延为2.5ty，16位4分组的最长进位时延达2.5ty×4=10ty。比16位行波进位最长时延32ty快了2倍多，是一种较好的进位加速折衷方案。**

# 成组先行-级联进位线路

- 16位成组先行-级联进位加法器进位过程




- 注意：16位成组先行-级联进位加法器最长进位延迟时间为$10t_y$。但32位时则达到$20t_y$，位数较多时加速作用明显下降。

# 多级先行进位

- 16位多级先行进位加法器举例：仍采用4-4-4-4分组方案，小组进位信号$C_{n+4}$不再由组内的先行进位线路产生，而是由组间先行进位线路产生。组间进位信号的通式为：

$$C_{n+4}= g_{n+3}+p_{n+3}g_{n+2}+p_{n+3}p_{n+2}g_{n+1}+p_{n+3}p_{n+2}p_{n+1}g_n+p_{n+3}p_{n+2}p_{n+1}p_nC_n$$

- 式中前4项为本组产生进位的条件，第5项则反映低一组进位信号通过本组传递到高一组的条件。可以定义两个小组进位函数：

$$G_i = g_{n+3}+p_{n+3}g_{n+2}+p_{n+3}p_{n+2}g_{n+1}+p_{n+3}p_{n+2}p_{n+1}g_n$$

小组进位生成函数

$$P_i = p_{n+3}p_{n+2}p_{n+1}p_n$$

小组进位传递函数

- 函数的下标 i 表示组号，在16位加法器中，i=0（n=0），1（n=4），2（n=8），3（n=12）。

# 二级先行进位逻辑

- 将小组进位函数代入小组进位信号表达式中可得：

$$
\left.
\begin{aligned}
C_4 &= G_0 + P_0 C_0 \\
C_8 &= G_1 + P_1 G_0 + P_1 P_0 C_0 \\
C_{12} &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \\
C_{16} &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0
\end{aligned}
\right\}
$$

- 由这组表达式可以看出，**组间先行进位的逻辑结构与组内先行进位线路完全相同**，其输入都依赖于进位函数和最低位进位，输出均为进位信号。因此，我们**可以使用通用的先行进位部件CLA来构成组间的先行进位线路。**
- 通常，最低级进位函数$p_i$、$g_i$称为**一级**进位函数；小组进位函数$P_i$、$G_i$称为**二级**进位函数；**三级**进位函数在本书中通常用$P^*$、$G^*$表示；……

# 二级先行进位逻辑

- **成组先行进位部件（Block Carry Look Ahead，BCLA）**

  **不直接输出小组间进位信号，而是输出一对组间进位函数P、G。**

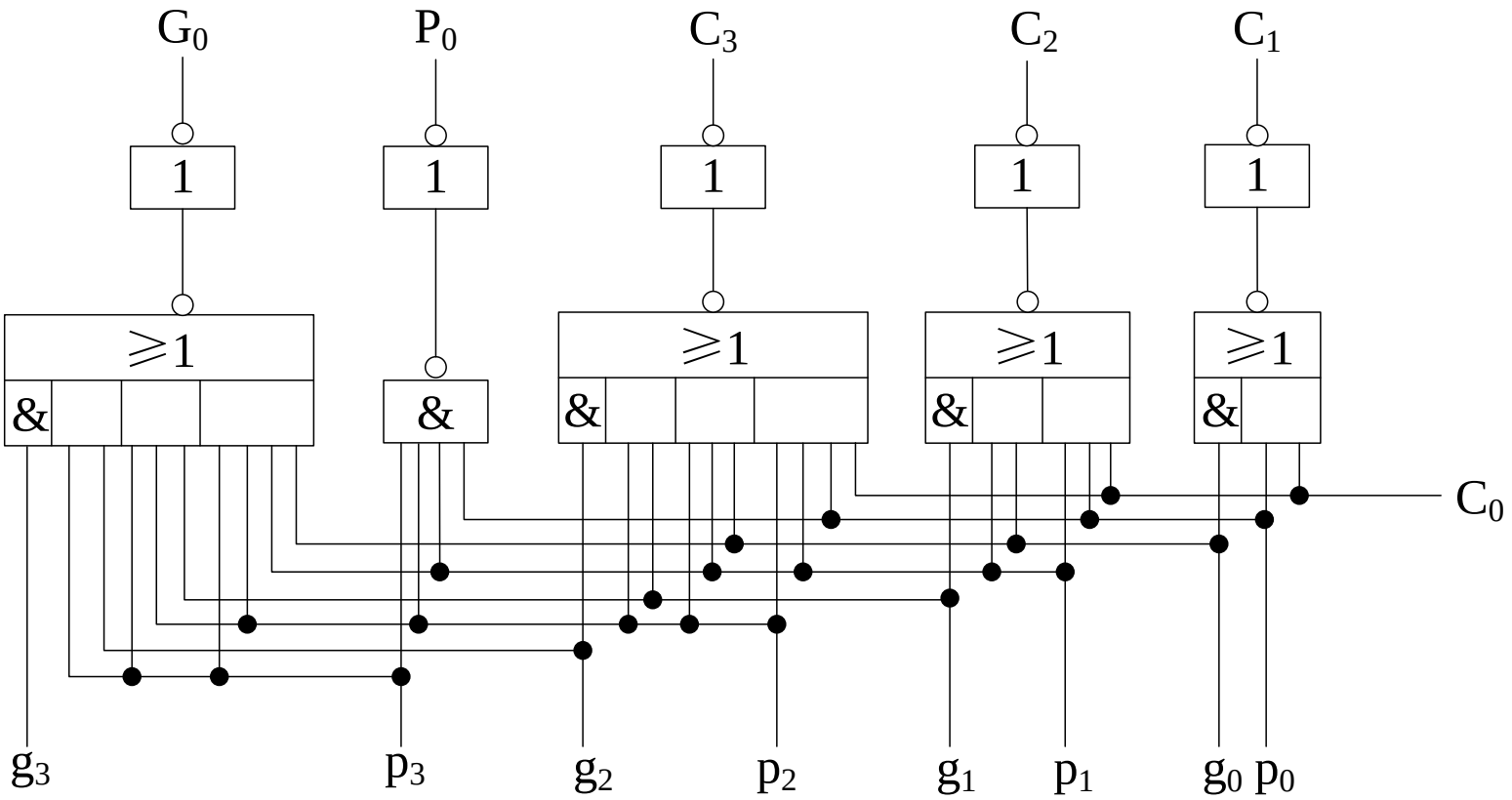
- **4位BCLA部件的输入输出逻辑**

$$
\begin{aligned}
C_{n+1} &= g_n + p_n C_n \\
C_{n+2} &= g_{n+1} + p_{n+1} g_n + p_{n+1} p_n C_n \\
C_{n+3} &= g_{n+2} + p_{n+2} g_{n+1} + p_{n+2} p_{n+1} g_n + p_{n+2} p_{n+1} p_n C_n \\
G_n &= g_{n+3} + p_{n+3} g_{n+2} + p_{n+3} p_{n+2} g_{n+1} + p_{n+3} p_{n+2} p_{n+1} g_n \\
P_n &= p_{n+3} p_{n+2} p_{n+1} p_n
\end{aligned}
$$

- **式中n仍然表示不同组的下标增量，$C_{n+1}$~$C_{n+3}$为组内进位输出；$p_n$~$p_{n+3}$、$g_n$~$g_{n+3}$为组内的进位函数输入；$P_n$、$G_n$为组间进位函数输出。**

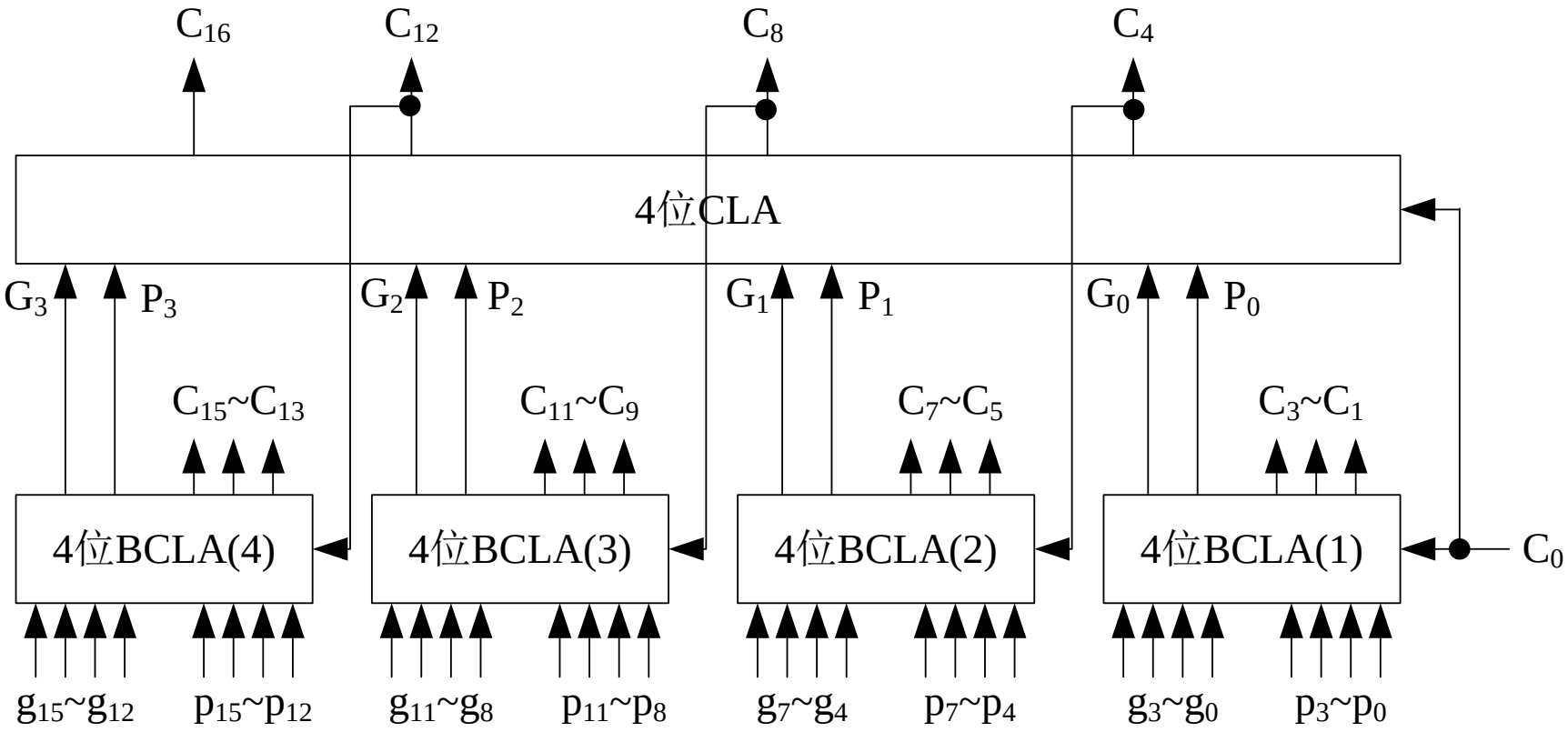
# 二级先行进位线路

## ❒ 4位成组先行进位部件BCLA逻辑图



# 二级先行进位逻辑

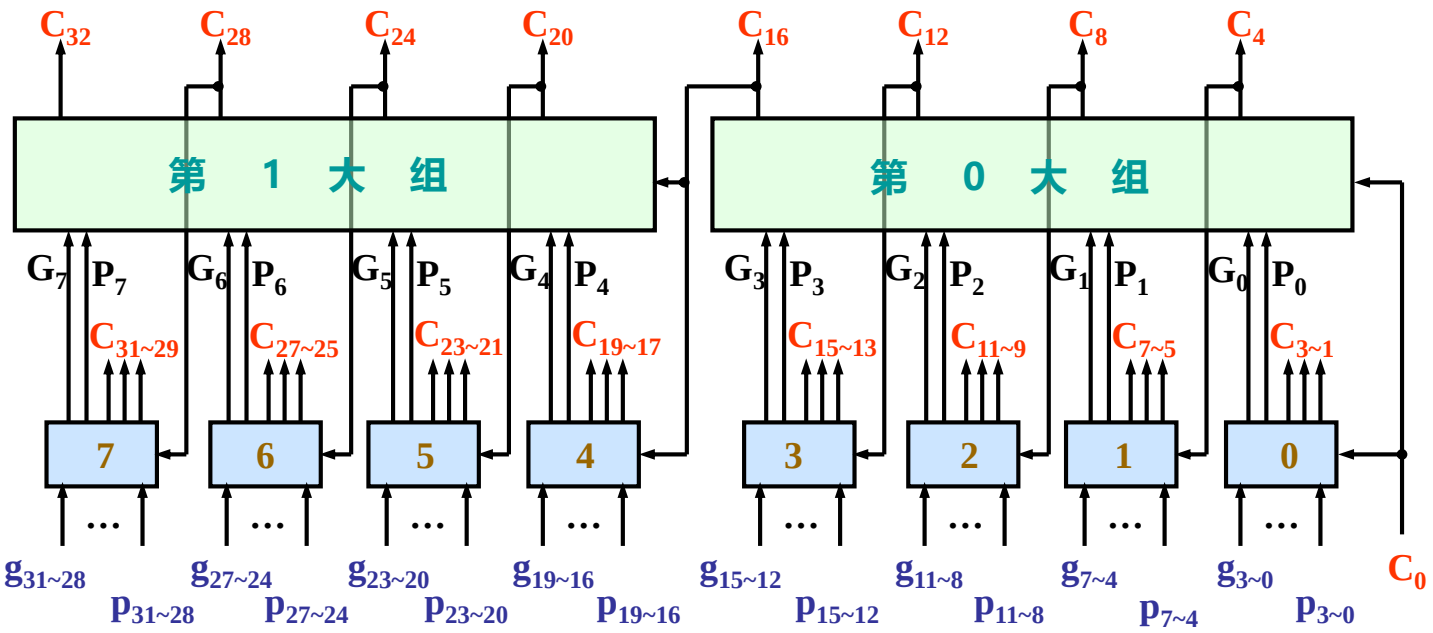
## ❒ 4-4-4-4分组的16位二级先行进位加法器逻辑框图



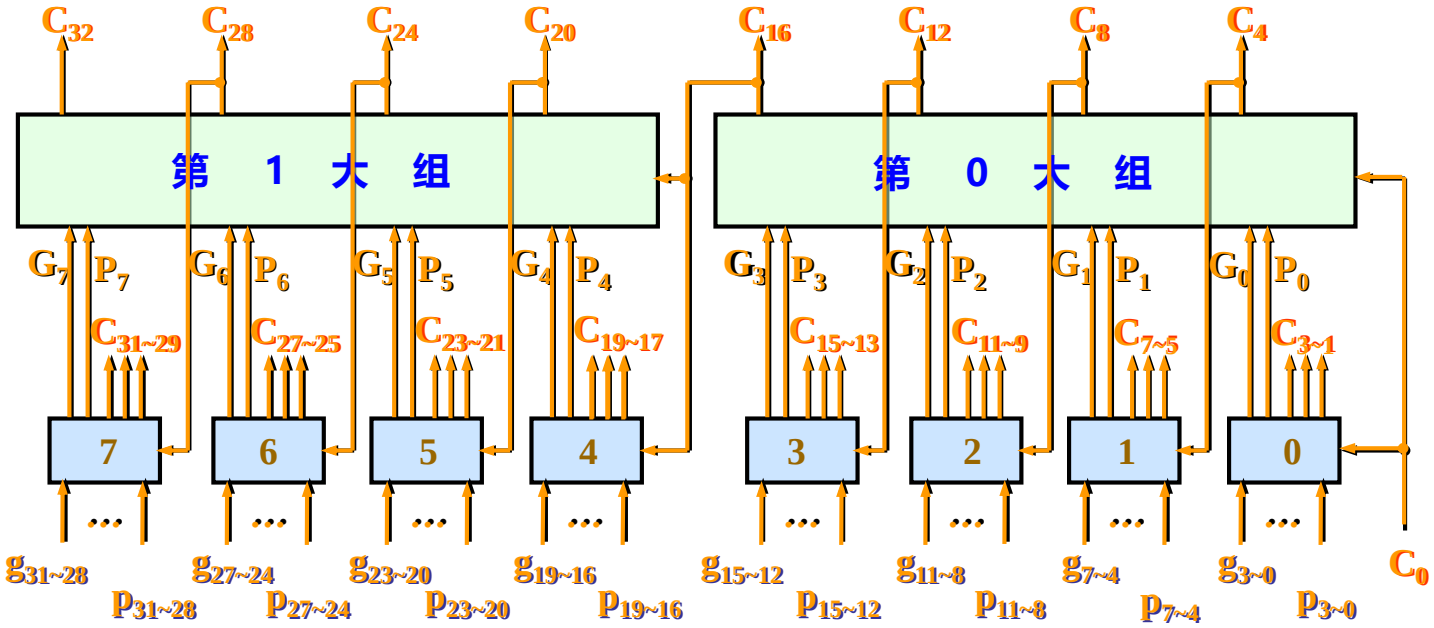
# 二级先行—级联进位逻辑

*思想:* 将n位并行加法器分成几个*大组*，每个大组又分成几个*小组*，大组和小组内的进位同时产生，大组间串行进位。

## 32位并行加法器二级先行-级联进位链框图



# 二级先行-级联进位过程




$g_i$、$p_i$、$C_0$输入并稳定;

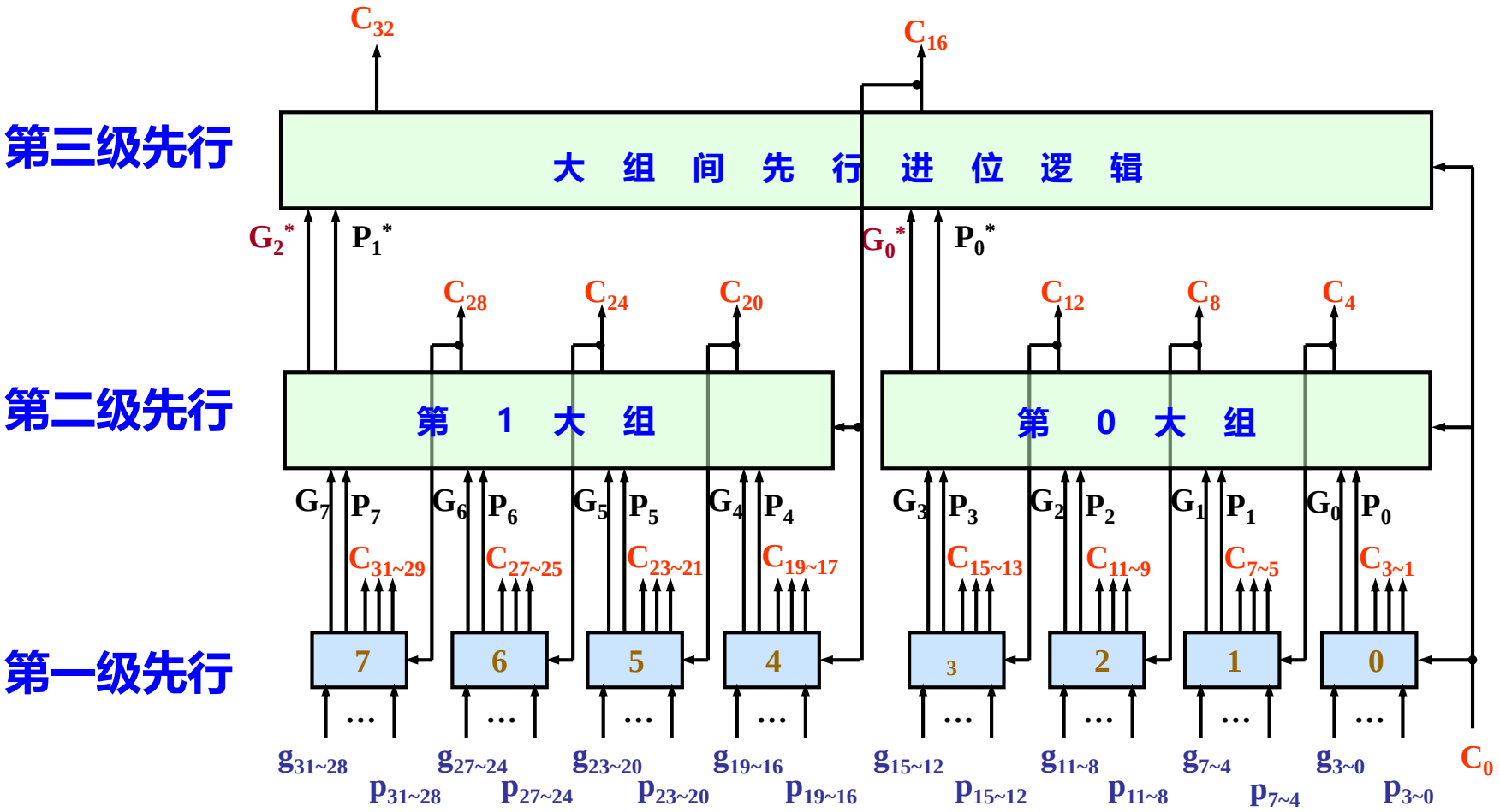
经2.5 $t_y$后产生$C_3$-$C_1$及$G_7$-$G_0$、$P_7$-$P_0$;

再经2.5 $t_y$后产生$C_{16}$、$C_{12}$、$C_8$、$C_4$;

再经2.5 $t_y$后产生$C_{19}$-$C_{17}$、$C_{15}$-$C_{13}$、$C_{11}$-$C_9$、$C_7$-$C_5$、及$C_{32}$、$C_{28}$、$C_{24}$、$C_{20}$;

最后再经2.5 $t_y$产生$C_{31}$-$C_{29}$、$C_{27}$-$C_{25}$、$C_{23}$-$C_{21}$;

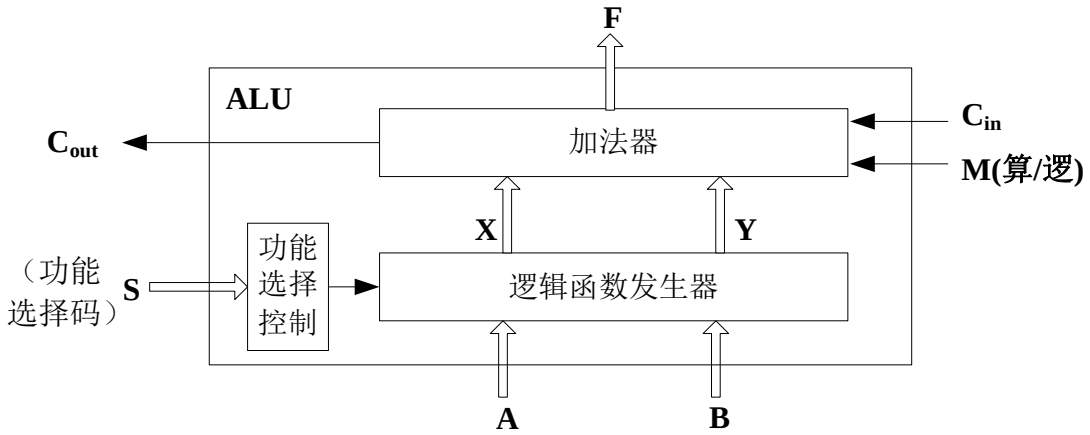
# 32位并行加法器三级先行进位链框图



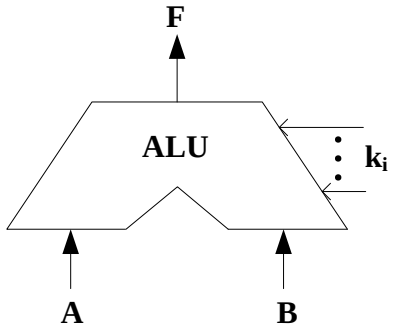
# 5.4.2算术逻辑单元（ALU）

## ❒ ALU的基本结构

- ❍同时兼有算术、逻辑运算功能并具备简单的选择控制能力的运算电路称为算术逻辑单元，简称ALU（Arithmetic Logic Unit）。
- ❍ALU是运算器核心部件，基本结构由加法器、逻辑函数发生器、功能选择电路等三部分组成。




（a）内部结构

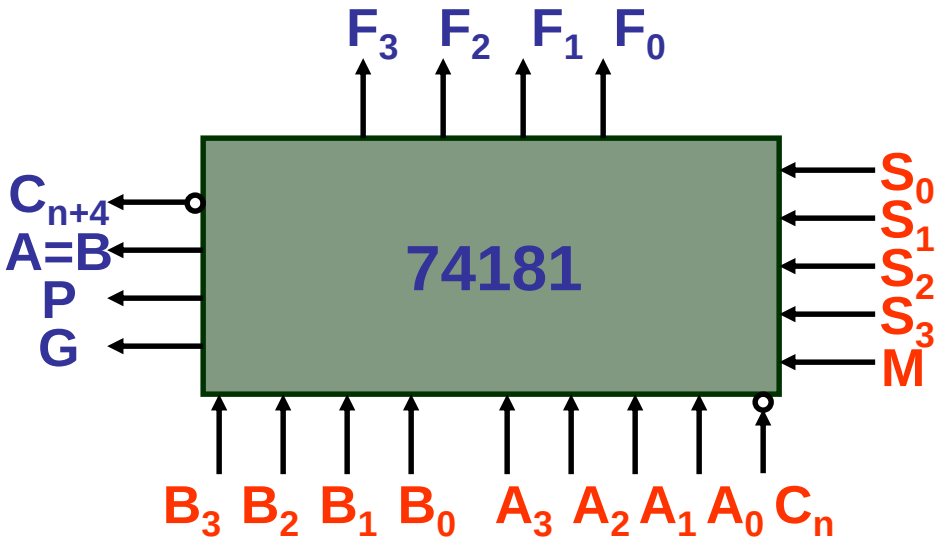



（b）逻辑符号

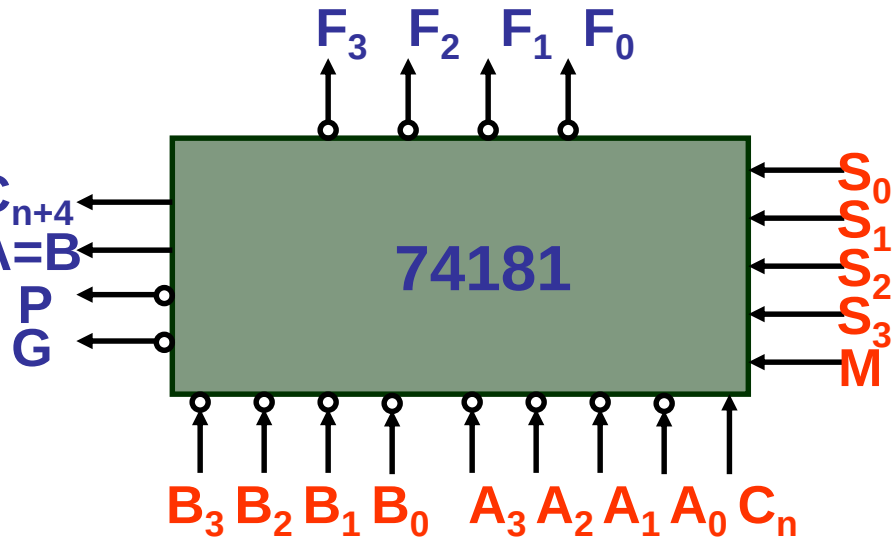
# 典型ALU芯片实例

- 多功能算术逻辑运算单元芯片——74181

74181是4位二进制多功能ALU部件，具有16种可选算/逻运算功能，支持正、负两种逻辑电平输入/输出，经常用来构成多位ALU电路。逻辑符号如下：




正逻辑



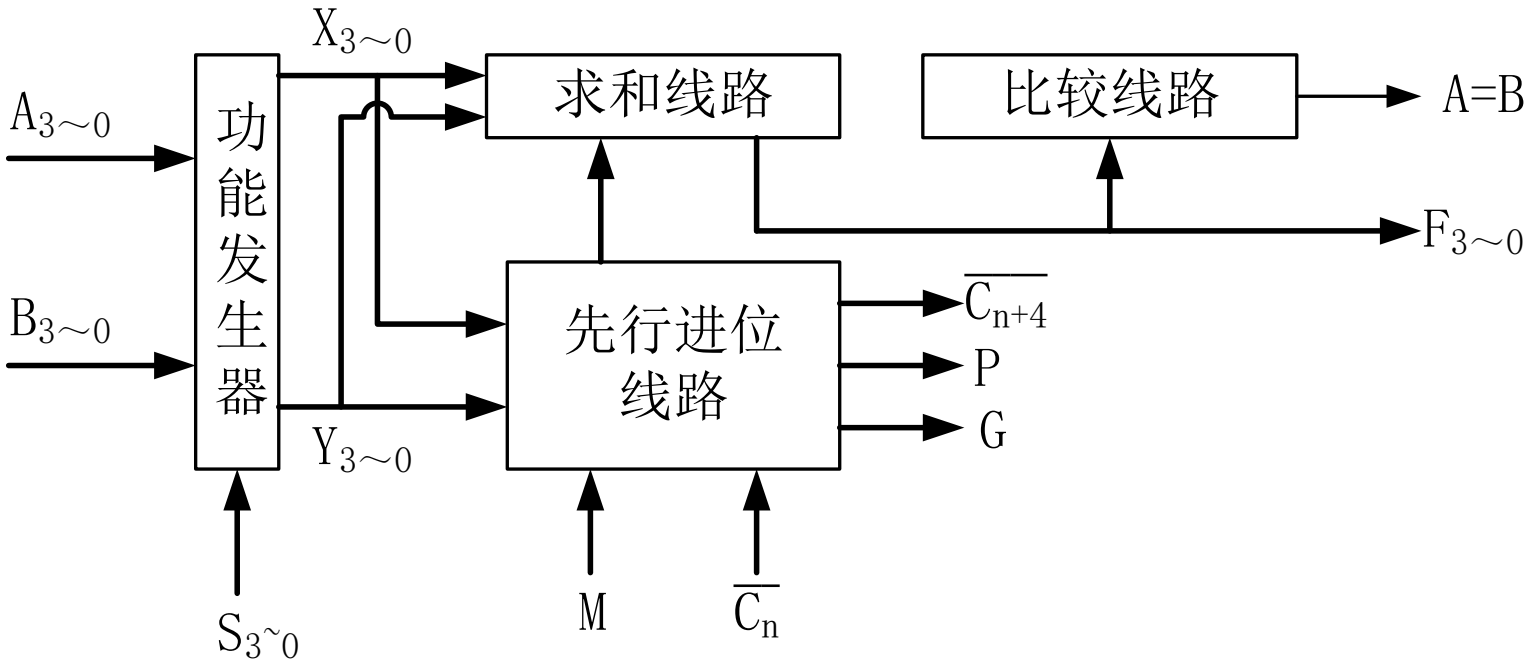

负逻辑

# 74181 ALU算术/逻辑运算功能表

| 工作方式选择输入 $S_3\ S_3\ S_1\ S_0$ | 负逻辑输入与输出 | | 正逻辑输入与输出 | |
|---|---|---|---|---|
| | 逻辑 (M=H) | 算术(M=L,$C_n$=L) | 逻辑 (M=H) | 算术(M=L ,$C_n$=H) |
| L L L L | /A | A减1 | /A | A |
| L L L H | /(AB) | AB减1 | /(A+B) | A+B |
| L L H L | /A+B | A(/B)减1 | /AB | A+/B |
| L L H H | 逻辑1 | 减1 | 逻辑0 | 减1 |
| L H L L | /(A+B) | A加(A+/B) | /(AB) | A加A ( /B) |
| L H L H | /B | AB加(A+/B) | /B | (A+/B)加A(/B) |
| L H H L | /(A⊕B) | A减B减1 | A⊕B | A减B减1 |
| L H H H | A+/B | A+/B | A(/B) | A(/B)减1 |
| H L L L | /AB | A加(A+B) | /A+B | A加AB |
| H L L H | A⊕B | A加B | /(A⊕B) | A加B |
| H L H L | B | A(/B)加(A+B) | B | (A+/B)加A(/B) |
| 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(/B) | AB加A | A+/B | (A+B )加 A |
| H H H L | AB | A(/B)加A | A+B | (A+/B)加A |
| H H H H | A | A | A | A减1 |

# 74181芯片内部结构

- 74181芯片内部的逻辑结构由逻辑函数发生器（功能发生器）、求和线路、先行进位线路、比较线路四部分组成：



# 74181芯片内部结构（续）

- **逻辑函数发生器。对原始输入数据$A_3$~$A_0$、$B_3$~$B_0$进行逻辑组合，产生出$X_3$~$X_0$、$Y_3$~$Y_0$两路逻辑函数输出。**
  - 第 i 位函数发生器的逻辑表达式

$$X_i = \overline{S_3 A_i B_i + S_2 \overline{A_i} B_i}$$

$$Y_i = \overline{A_i + S_0 B_i + S_1 \overline{B_i}}$$

（正逻辑）

  - 设计特点：当进行加法运算时，$X_i$、$Y_i$既是加数，又是进位函数$g_i$、$p_i$，即：

$$g_i = X_i \wedge Y_i = Y_i$$

$$p_i = X_i \vee Y_i = X_i$$

# 74181芯片内部结构（续）

- **求和电路**

  两级异或门组成，以正逻辑为例其等效逻辑表达式为：

  $$F_i=X_i \oplus Y_i \oplus (C_{n+i}+M)$$

  当M=H时，选择逻辑运算：$F_i= X_i \oplus Y_i \oplus 1=X_i \odot Y_i$

  当M=L时，选择算术运算：$F_i= X_i \oplus Y_i \oplus C_{n+i}$

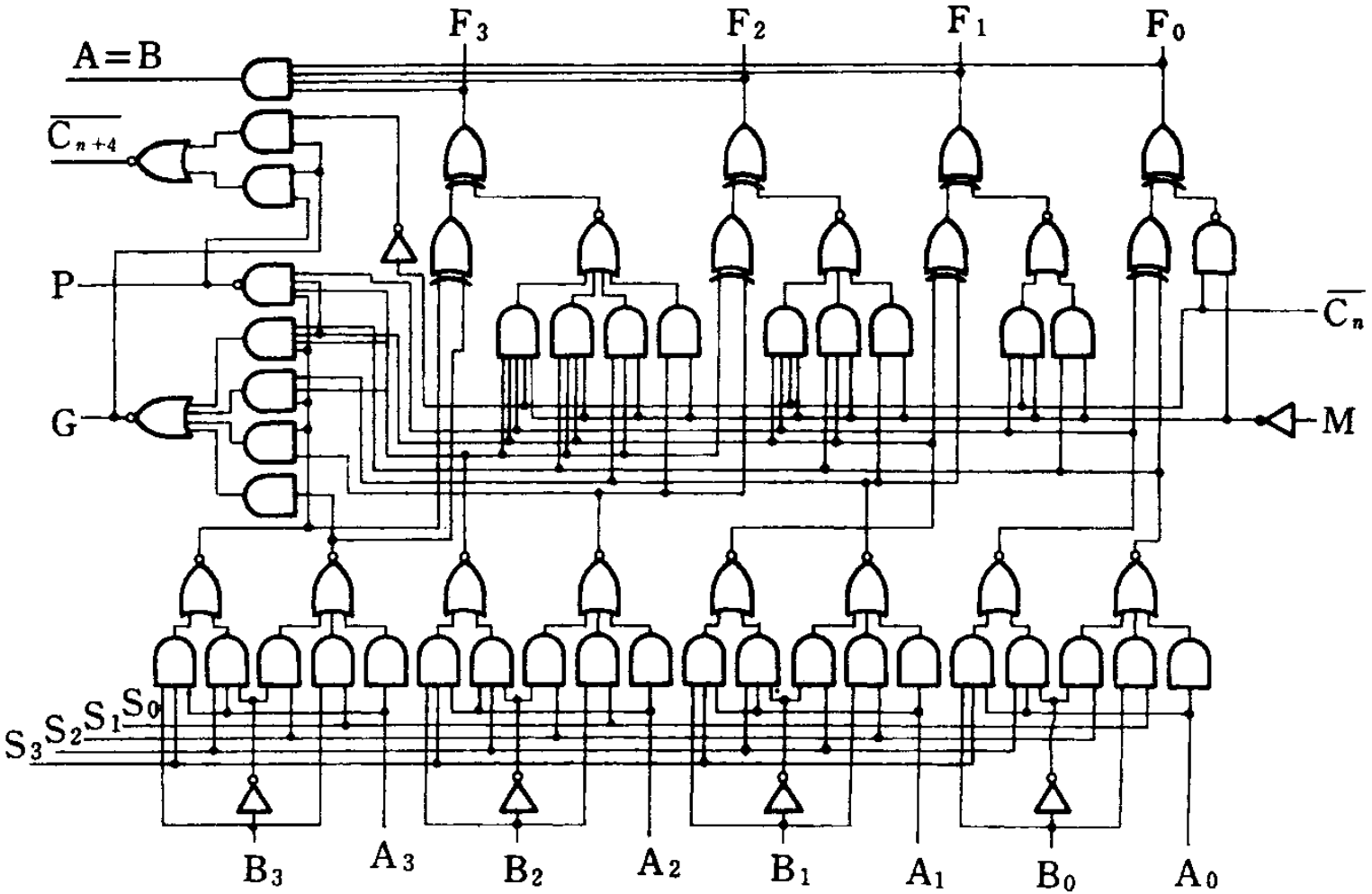
- **先行进位电路**

  兼有CLA、BCLA两种先行进位功能，四位CLA电路产生$C_{n+4}$~$C_{n+1}$四个进位信号，$C_{n+3}$~$C_{n+1}$为组内进位不输出，小组间进位信号$C_{n+4}$通过相应的引脚输出。74181还可输出小组进位函数P、G，供外接的BCLA芯片使用。

- **比较电路**

  74181内部设置了一个比较门，当$F_3$~$F_0$为全0时，其输出端A=B输出有效的高电平比较信号。

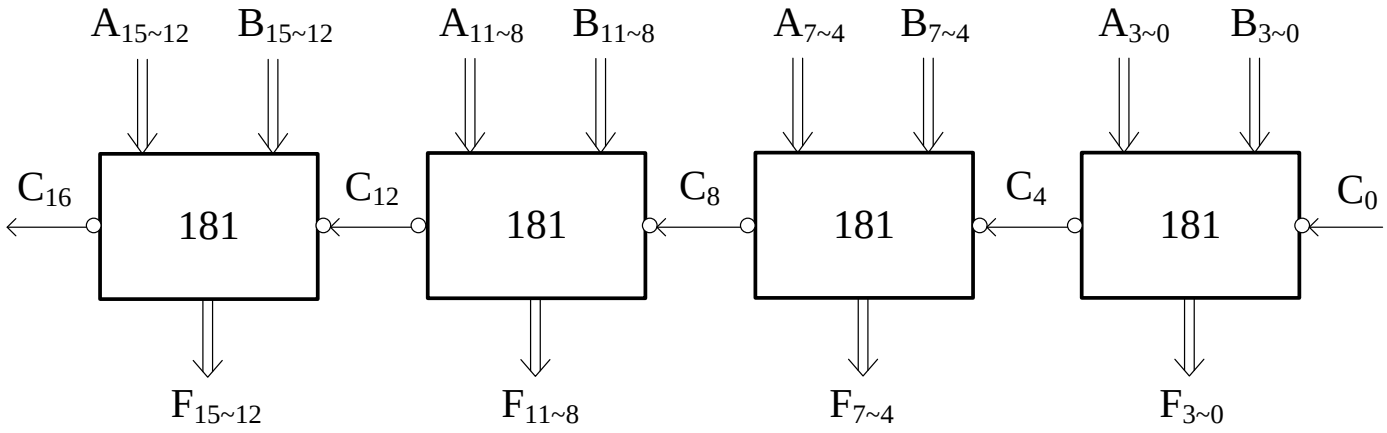
# 正逻辑的74181ALU内部逻辑线路



# 多位ALU线路实现举例

- 多位成组先行-级联进位的ALU

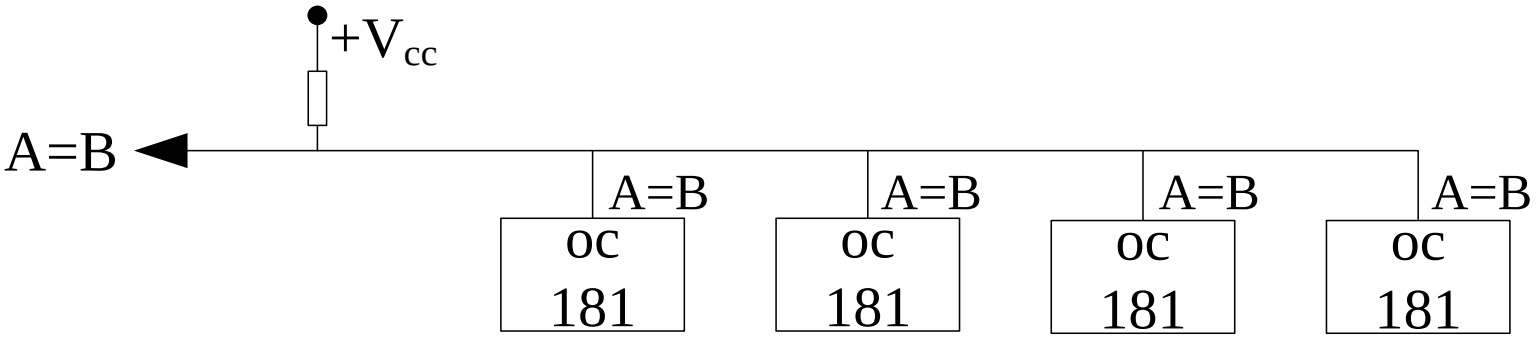
  多片74181芯片的进位输入/输出端串联可以构成多位的成组先行-级联进位ALU。以正逻辑为例，一个用4片74181构成的16位成组先行—级联进位ALU逻辑如下：




- 为了突出进位结构，图中与进位无关的引脚连线省略。

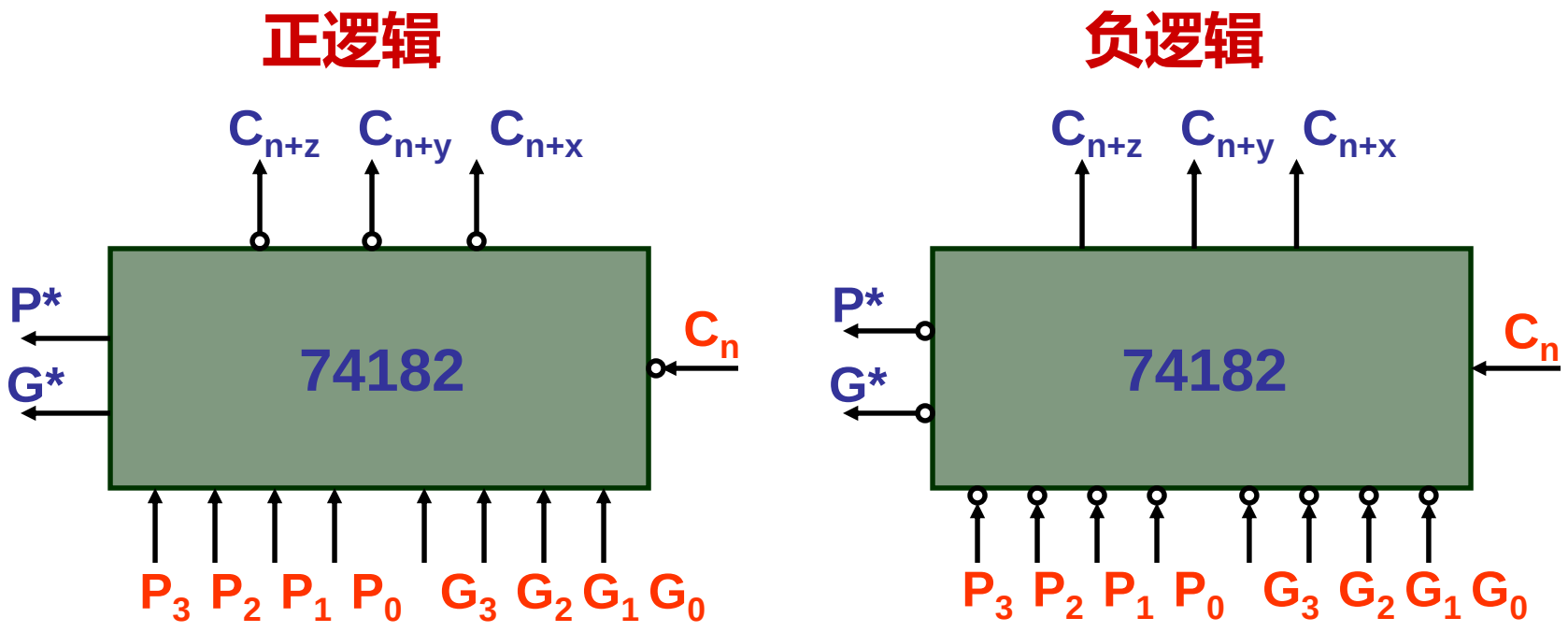
# 多位ALU线路实现举例

- 将4片74181芯片的A=B输出端**并联**，还可实现**16位的比较输出线路**，如下图：



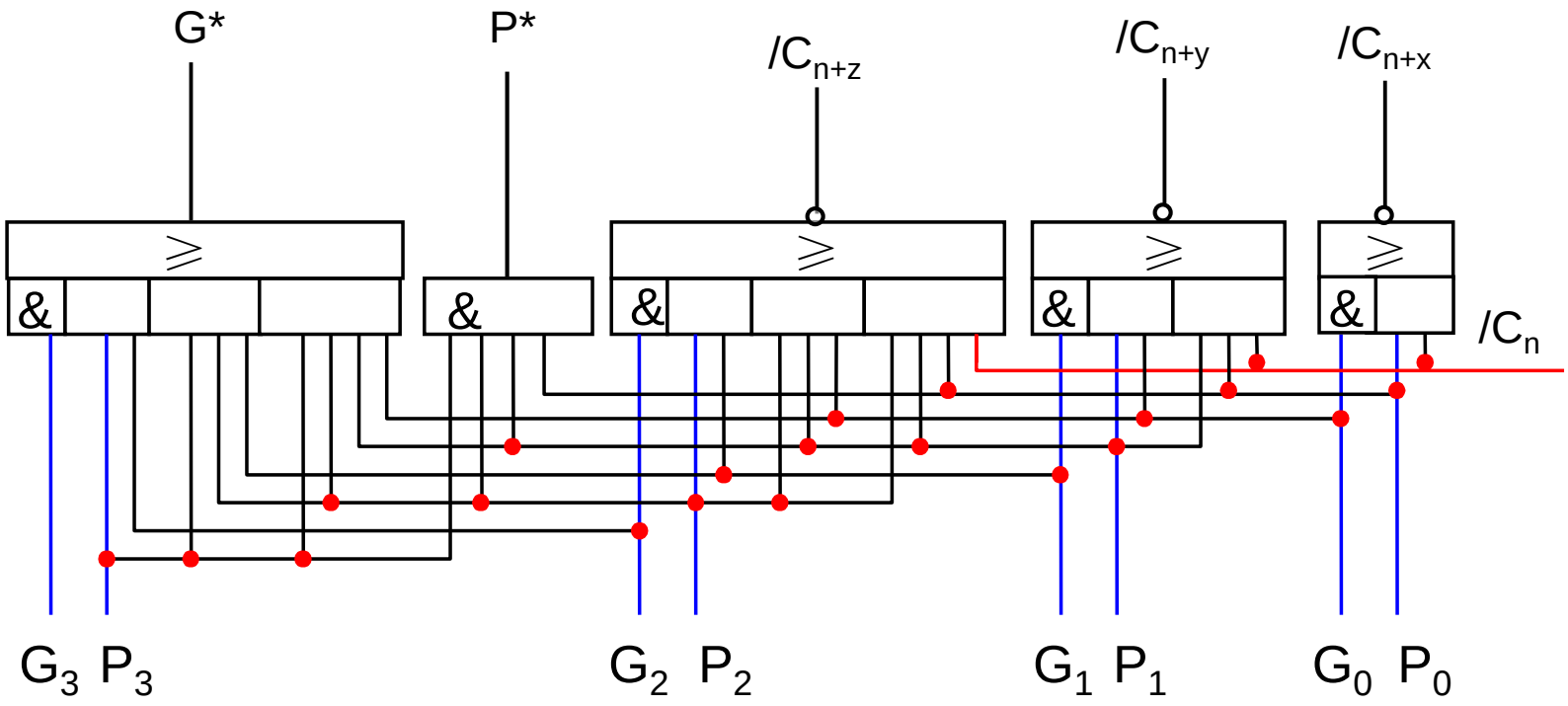
# 74182 BCLA芯片

- 74182是与74181配套的通用4位成组先行进位部件，74181和74182配合可以构成多位的多级先行进位ALU。74182的逻辑符号如下：



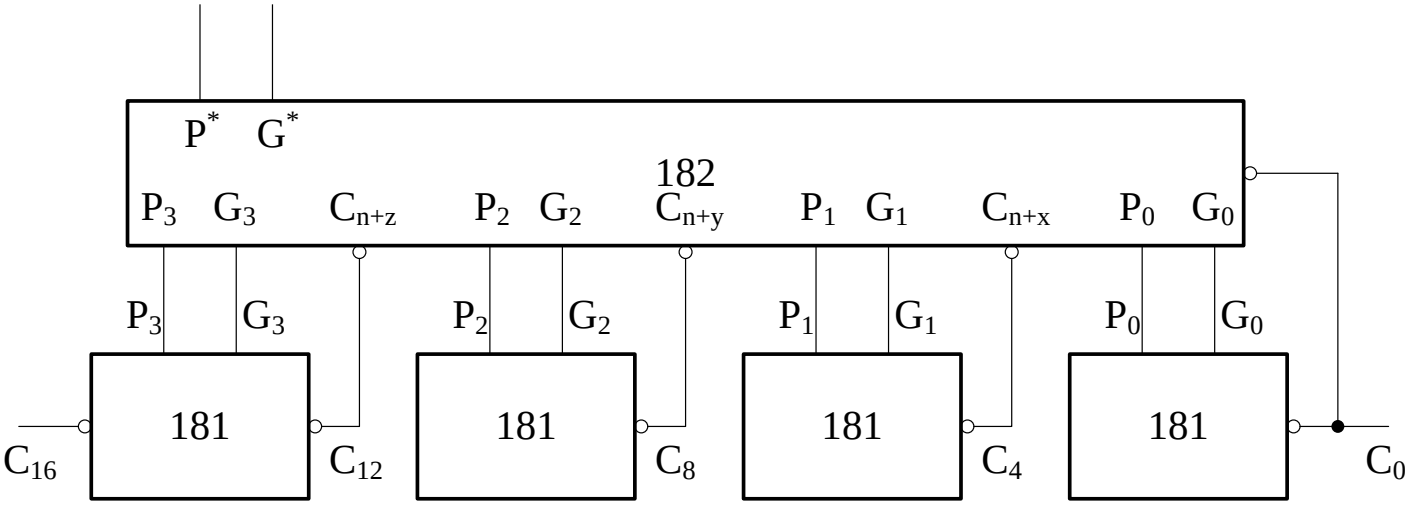
# 74182内部逻辑结构原理图（正逻辑）

- **特点：为了减少延迟时间，进位信号输出采用了与进位函数输入反相的输出方式。**



# 二级先行进位ALU举例

- 4片74181与1片74182联合可构成16位二级先行进位的ALU，如下图（正逻辑为例）：

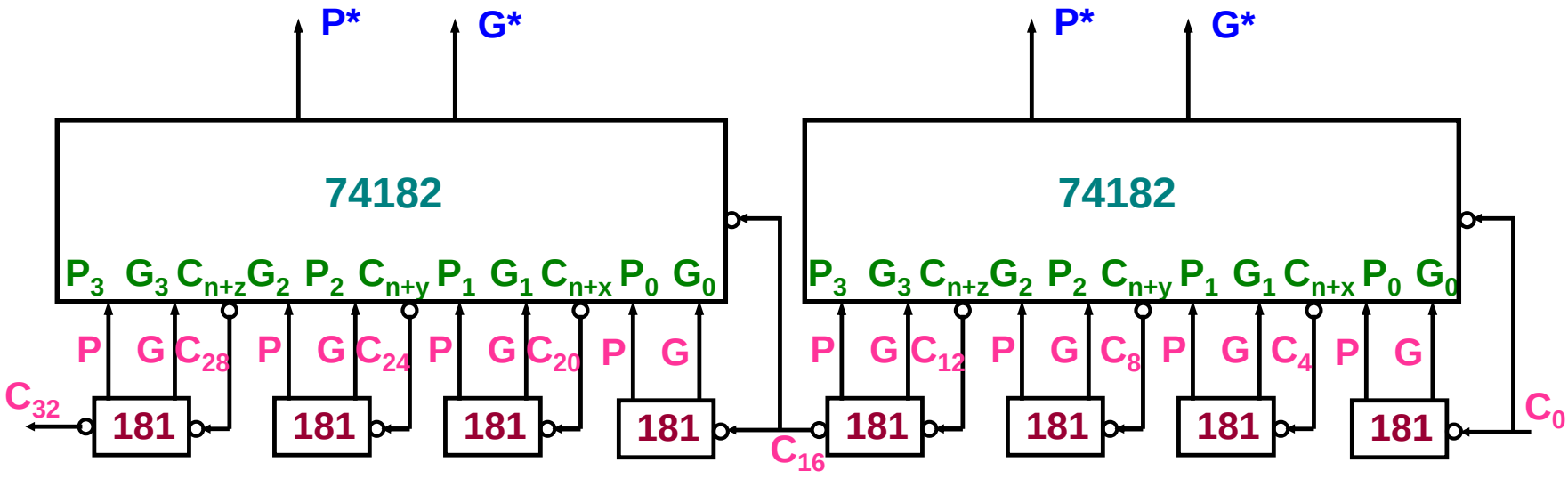



- 特点

16位ALU仍分4组，组内先行进位由74181内部实现，但组间不再使用74181的进位输出端$C_{n+4}$，而是将74181输出的小组进位函数P、G信号送往74182，再通过74182产生组间先行进位信号。

# 二级先行-级联进位的ALU举例

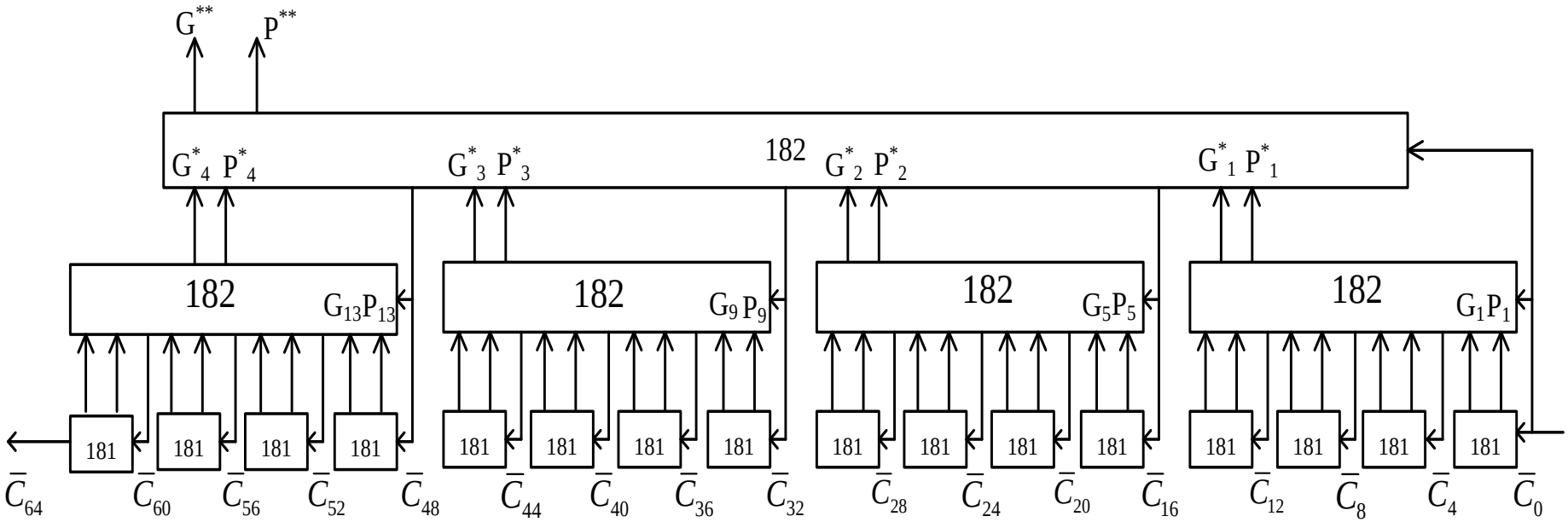
- 由2片74182和8片74181构成的32位二级先行-级联进位的ALU逻辑框图（正逻辑为例）：




- 注：图中芯片省略了与进位无关的引脚。

# 三级全先行进位的ALU举例

- 64位的ALU，4位一小组、16位一大组分组，共分16个小组，用16片74181构成；4个大组，大组内部的二级先行进位线路用4片74182构成；4个大组间的先行进位再用1片74182实现。如下图（正逻辑）：

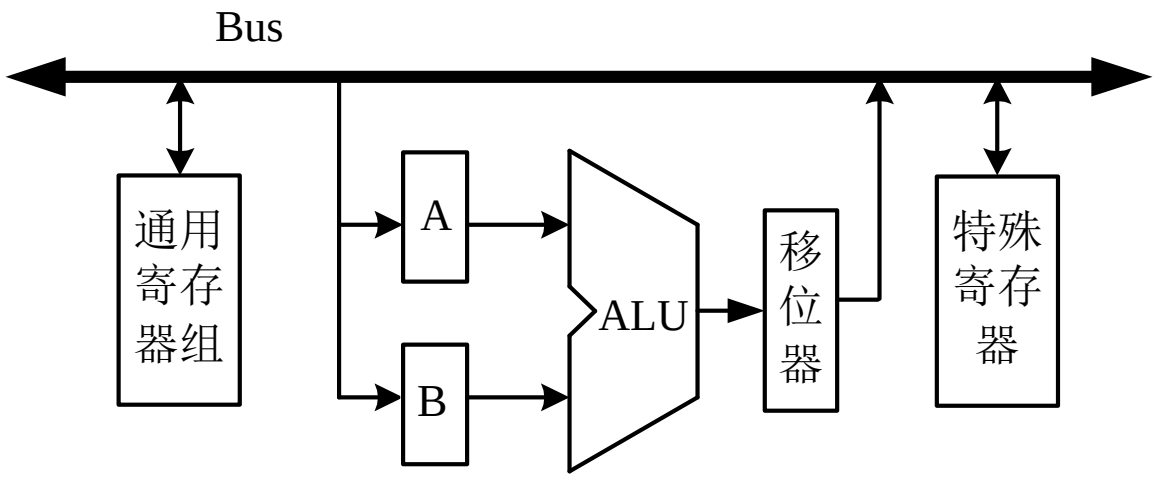


# 5.4.3定点运算器的基本结构

- 运算器的基本功能
  - 完成对数据的**算术/逻辑运算**，由算逻运算部件ALU承担，它在给出运算结果的同时，还给出结果的某些特征（溢出、进位、为零、为负等）。
  - **暂存参加运算的原始数据和中间结果**，通过运算器内部的一组寄存器完成（通用寄存器组）。
  - **设置乘商寄存器（专用寄存器）及移位器部件**，支持传统乘除运算所需移位操作及移位指令的执行。
  - ALU的输入/输出与通用寄存器组等部件间的相互连接通常通过几组**多路选择器**电路实现，运算器内部的数据传送通过其**内部总线**完成。
  - 运算器作为CPU内部传送数据的主要通路，要考虑与计算机中其他功能部件连接和协同运行，故运算器也经常称为CPU的**数据通路**。
- 运算器的组成是计算机设计时的一项重点工作。本章仅围绕运算器组织基本功能的逻辑实现讨论。

# 定点运算器的基本结构

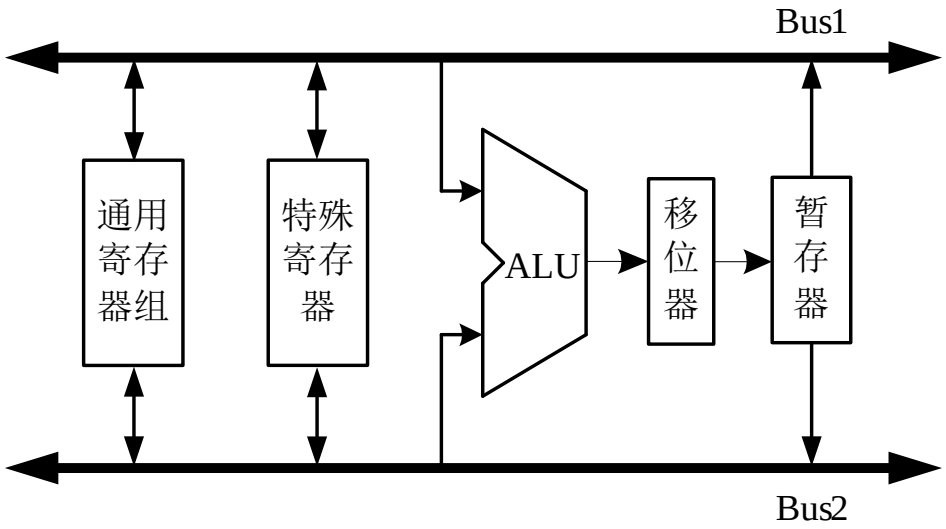
- 分散互连结构的运算器
- 单总线结构的运算器




- 特点
  - 需要设置2个暂存器协调ALU输入/输出端同时传送问题；
  - 完成一次加法运算需要3个CPU时钟周期的时间；
  - 结构简单、规整，运算速度慢。

# 定点运算器的基本结构

## ❒ 双总线结构的运算器

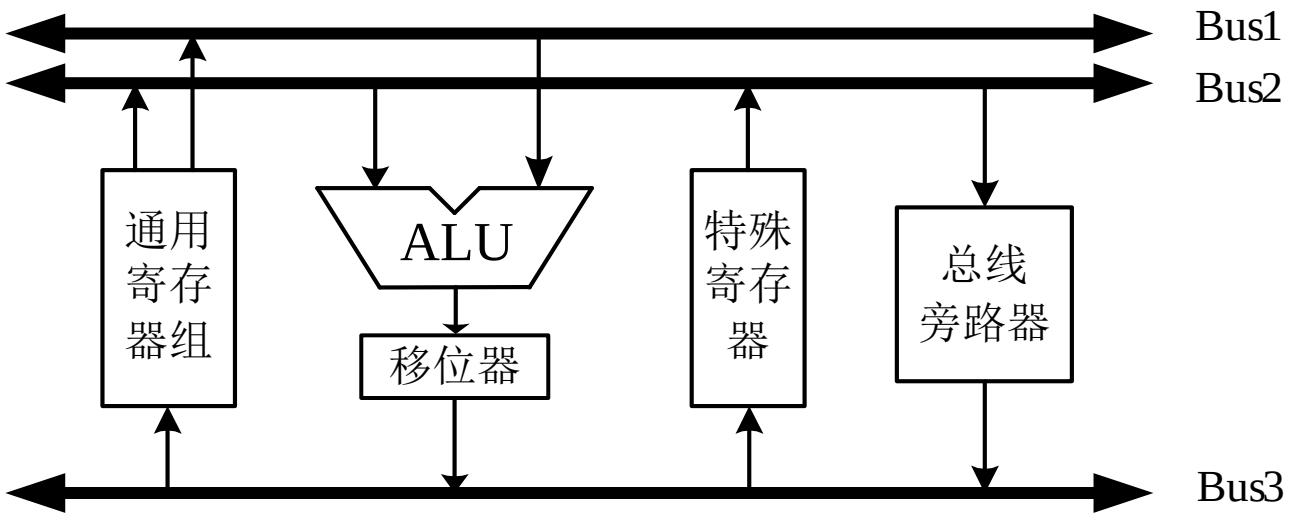



## ❒ 特点

- ❍ ALU的输入/输出端只需要设一个暂存器。
- ❍ 运算器完成一次加法运算需要2个CPU时钟周期的时间。
- ❍ 两条总线之间的传输可通过ALU进行，或通过设置专用的总线连接器实现。

# 定点运算器的基本结构

## ❒ 三总线结构的运算器




## ❒ 特点

- ❍ 运算器完成一次加法运算只需要1个CPU时钟周期的时间，运算速度不再受总线分时使用的限制，在此总线已不是影响运算器性能的主要因素。

# 本章第二次作业（总第9次作业）

- 5.18、5.19、5.28