华北电力大学
North China Electric Power University

计算机组成原理

第二章

华北电力大学计算机系

本章结构

2.1 数值数据的表示
2.2 数据处理与存储
2.3 定点运算
2.4 浮点运算
2.5 运算器部件及进位链结构

2.1 数值数据的表示

- 数值数据表示的三要素
  - 进位计数制
  - 定、浮点表示
  - 如何用二进制编码
- 进位计数制
  - 十进制、二进制、十六进制、八进制数及其相互转换
- 定/浮点表示（解决小数点问题）
  - 定点整数、定点小数
  - 浮点数（可用一个定点小数和一个定点整数来表示）
- 定点数的编码（解决正负号问题）
  - 原码、补码、移码、反码 （很少用）

原码

Decimal	Binary	Decimal	Binary
0	0000	-0	1000
1	0001	-1	1001
2	0010	-2	1010
3	0011	-3	1011
4	0100	-4	1100
5	0101	-5	1101
6	0110	-6	1110
7	0111	-7	1111

“正”号用0表示
“负”号用1表示
数值部分不变!

- 容易理解,但是:
  - ✓ 0 的表示不唯一，故不利于程序员编程
  - ✓ 加、减运算方式不统一
  - ✓ 需额外对符号位进行处理，故不利于硬件设计

从 50年代开始，整数都采用补码来表示，但浮点数的尾数用原码定点小数表示

补码

重要概念：在一个模运算系统中，一个数与它除以“模”后的余数等价

时钟是一种模12系统 现实世界中的模运算系统

假定钟表时针指向10点，要将它拨向6点，则有两种拨法：

① 倒拨4格：10- 4 = 6

② 顺拨8格：10+8 = 18 ≡ 6 (mod 12)

模12系统中： 10- 4 ≡ 10+8 (mod 12)

- 4 ≡ 8 (mod 12)

8是- 4对模12的补码 （即：- 4的模12补码等于8）

同样有 -3 ≡ 9 -5 ≡ 7 （mod 12）

结论1： 一个负数的补码等于模减该负数的绝对值。

结论2： 对于某一确定的模，某数减去小于模的另一数，总可以用该数加上另一负数的补码来代替。

补码（modular运算）：+ 和 – 的统一

补码

补码定义：$[X]_{补} = 2^n + X \quad (-2^n \le X < 2^n, \ \mathrm{mod}\ 2^n)$

- 正数：符号位为0，数值部分不变
- 负数：符号位为1，数值部分“各位取反，末位加1”

变形补码：双符号，用于存放可能溢出的中间结果。

Decimal	补码	变形补码	Decimal	补码	变形补码
0	0000	00000	-0	0000	00000
1	0001	00001	-1	1111	11111
2	0010	00010	-2	1110	11110
3	0011	00011	-3	1101	11101
4	0100	00100	-4	1100	11100
5	0101	00101	-5	1011	11011
6	0110	00110	-6	1010	11010
7	0111	00111	-7	1001	11001
8	1000	01000	-8	1000	11000

值太大，用4位补码无法表示，故“溢出”
但用变形补码可保留符号位和最高数值位

+0和-0表示唯一

移码

- 什么是移码表示？

  将每一个数值加上一个偏置常数（ Excess / bias）

- 当编码位数为n时，bias取 2^{n-1} 或 $2^{n-1}-1$（如 IEEE 754）

  如 n=4: $E_{biased} = E + 2^3$ （bias= 2^3 =1000B）

  -8 (+8) ~ 0000B
  -7 (+8) ~ 0001B
  …
  0 (+8) ~ 1000B
  …
  +7 (+8) ~ 1111B

  0的移码表示唯一

  当bias为 2^{n-1} 时，移码和补码仅第一位不同

  移码用来表示浮点数的阶

- 为什么要用移码来表示阶码?

  便于浮点数加减运算时的对阶操作（比较大小）

例：$1.01 \times 2^{-1} + 1.11 \times 2^3$

补码： 111 < 011 ?
(-1) (3)

简化比较

$1.01 \times 2^{-1+4} + 1.11 \times 2^{3+4}$

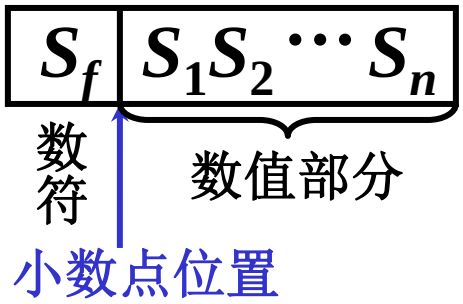
移码： 011 < 111
(3) (7)

定点数与浮点数

定点数的表示

数的小数点固定在同一位置不变。

①带符号的定点小数



约定所有数的小数点的位置，固定在符号位之后。

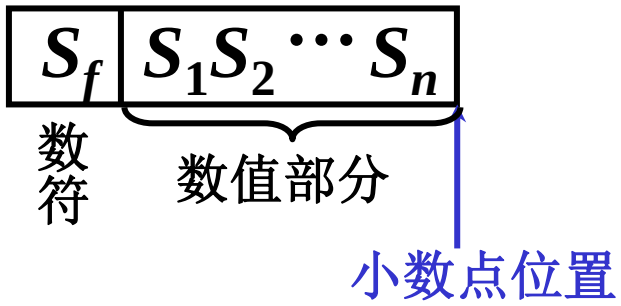
字长 n+1位，则表示范围为： $-(1-2^{-n}) \sim 1-2^{-n}$

定点数与浮点数

定点数的表示

数的小数点固定在同一位置不变。

②带符号的定点整数



小数点的位置固定

在最低数值位之后

字长 n+1位，则表示范围为：$-(2^n-1) \sim 2^n-1$

定点数与浮点数

定点数的表示

③无符号定点整数

小数点的位置固定在最低数值位之后

若代码序列为$X_nX_{n-1}\cdots X_1X_0$，共n+1位，则有：

典型值	真值	代码序列
最大正数	$2^{n+1}-1$	11…1111
最小非零正数	1	00…0001

表示范围为：　0 ~ $(2^{n+1}-1)$

分辨率为：1

定点数与浮点数

字长8位的定点数的表示范围

无符号数： 00000000 ～11111111
0 255

定点整数： $11111111_{原}$～$01111111_{原}$
-127 127

$10000000_{补}$～$01111111_{补}$
-128 127

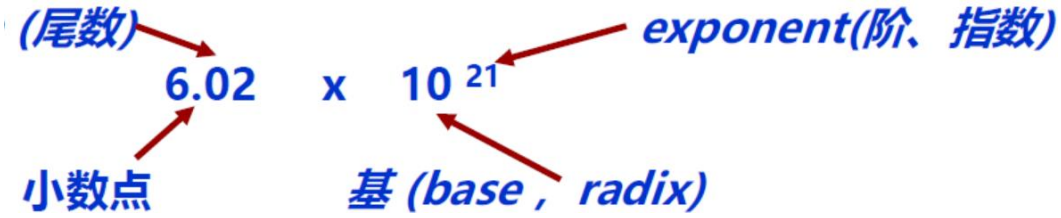
定点小数： $1.1111111_{原}$～$0.1111111_{原}$
$-(1-2^{-7})$ $(1-2^{-7})$

$1.0000000_{补}$～$0.1111111_{补}$
-1 $(1-2^{-7})$

定点数与浮点数

浮点数的表示

对于科学计数法（十进制数）：

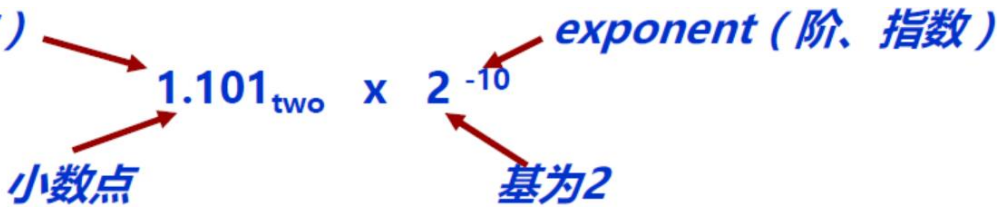


规格化形式: 小数点前只有一位非0数

同一个数有多种表示形式。例：对于数 1/1,000,000,000

- **规格化形式: 1.0×10^{-9} 唯一**
- **非规格化形式: 0.1×10^{-8}, 10.0×10^{-10} 不唯一**

对于二进制数实数

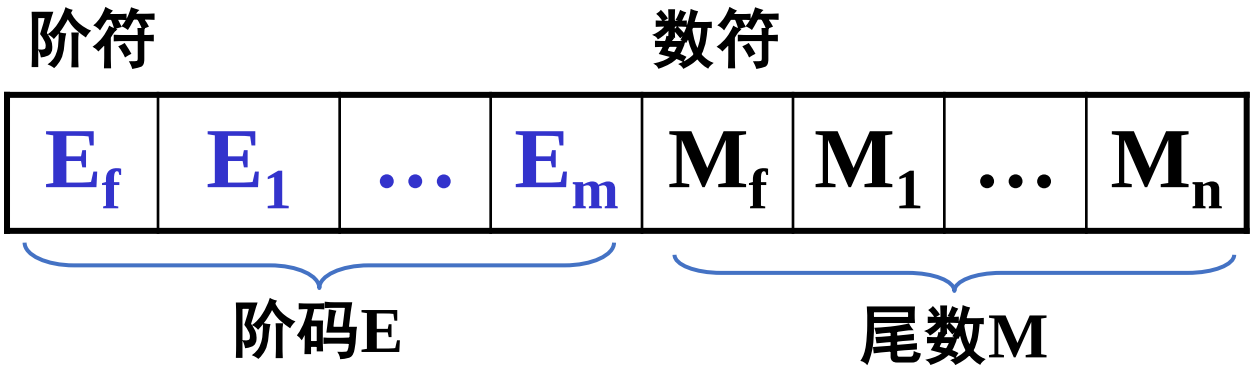


只要对尾数和指数分别编码，就可表示一个浮点数（即：实数）

定点数与浮点数

浮点数的机器（存储）格式

浮点数真值：$N = \pm R^E \times M$



R：阶码的底数，隐含约定为2。

E：阶码，定点整数，补码或移码表示，其位数决定了数值的范围；

M：尾数，为定点小数，原码或补码表示，其位数决定着数的精度；

定点数与浮点数

尾数M的规格化表示

浮点数真值：$N = \pm R^E \times M$

规格化的目的 → 使浮点数的表示代码“唯一”

① 浮点数用原码表示时

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

规格化以后尾数的最高有效位为“1”

$M_{原}$=0.1000，1.1010

② 浮点数用补码表示时

$-1 \le M < -1/2$ 或 $1/2 \le M < 1$

正数，规格化后最高数值位为“1”

负数，规格化后最高数值位为“0”

如0.1010，0.1110

0.625 0.875

如1.0010，1.0000

-0.875 -1

定点数与浮点数

【例1】阶符1位、阶码m位，数符1位、尾数n位，如果表示成规格化补码，分析各真值对应的M、E取值情况。

$$N = M \times 2^E = F(M, E)$$

$$\begin{cases} -1 \le M < -1/2 \text{或} 1/2 \le M < 1 \\ -2^m \le E \le 2^m - 1 \end{cases}$$

最小浮点数	E→	为最大正数：	2^m-1
	M→	为最小负数：	-1
最大浮点数	E→	为最大正数：	2^m-1
	M→	为最大正数：	$1-2^{-n}$
最小浮点正数	E→	为最小负数：	-2^m
	M→	为最小正数：	2^{-1}
最大浮点负数	E→	为最小负数：	-2^m
	M→	为最大负数：	$-(1/2+2^{-n})$

机器零

- 当浮点数 尾数为 0 时，不论其阶码为何值 按机器零处理
- 当浮点数 阶码等于或小于它所表示的最小数 时，不论尾数为何值，按机器零处理

如 $m = 4$ $n = 10$

当阶码和尾数都用补码表示时，机器零为

×, × × × ×； 0. 0 0 …… 0

(阶码 = −16) 1, 0 0 0 0； ×.×× …… ×

当阶码用移码，尾数用补码表示时，机器零为

0, 0 0 0 0； 0. 0 0 …… 0

有利于机器中“ 判 0 ” 电路的实现

IEEE 754 格式的浮点数

直到80年代初，各个机器内部的浮点数表示格式还没有统一 因而相互不兼容，机器之间传送数据时，带来麻烦

1970年代后期, IEEE成立委员会着手制定浮点数标准

1985年完成浮点数标准IEEE 754的制定

现在所有通用计算机都采用IEEE 754来表示浮点数

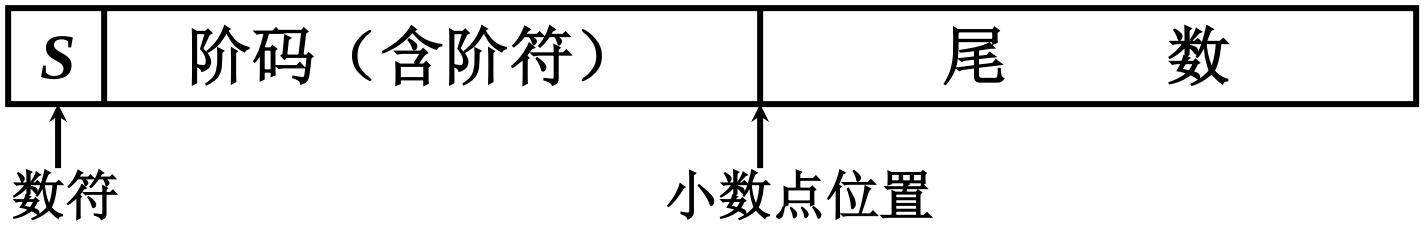
This standard was primarily the work of one person, UC Berkeley math professor William Kahan.

www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

Prof. William Kahan

IEEE 754 格式的浮点数

- IEEE的二进制浮点数算术标准，最广泛使用的浮点数表示和运算标准



	符号位 S	阶码	尾数	总位数
短实数	1	8	23	32
长实数	1	11	52	64
临时实数	1	15	64	80

移码 纯小数

IEEE 754 格式的浮点数

IEEE 754浮点数x的真值表示

$$x = (-1)^S \times (1.M) \times 2^{E-偏移量}$$

- **阶码的移码表示**：隐含阶符，通过移码方法来表示正负阶码值

浮点数的指数真值 e与阶码E的关系为：

$$E = e + 偏移量$$

短实数的偏移量：127 (0111 1111)

长实数的偏移量：1023 (011 1111 1111)

临时实数的偏移量: 16383 (011 1111 1111 1111)

- **尾数形式—1.M**
  - 有别于一般的规格化数形式。尾数域的最高位总为 1，对这一位一般不予存储，而作为隐藏位。
  - 实际表示中，对短实数和长实数，隐藏位的1省略；但对临时实数不采用隐藏位方案。

实数 178.125的几种不同表示

实数表示	数　　值		
原始十进制表示	178.125		
二进制数	10110010.001		
二进制浮点表示	$1.0110010001\times2^{111}$		
短浮点数表示	符号	偏移的阶码	有效值
	0	00000111+01111111 =10000110	01100100010000000000000 1.（隐含的）

注：IEEE754中，单精度浮点数偏移量加127，不是128

作业

1. 将（100.25）$_{10}$转换成短浮点数格式

0；10000101；10010001000000000000000

十六进制代码：42C88000

2. 把短浮点数C1C90000H转换成十进制数(-25.125)

3.Float型数据通常用IEEE754单精度浮点数格式表示。如编译器将float型变量x分配在一个32位浮点寄存器FR1中，且x=-8.25，则FR1的内容是（A）。

A. C104 0000H　B. C242 000H

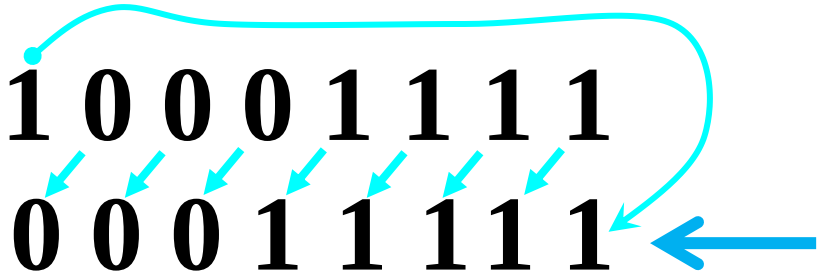
C. C184 000H　C. C1C2 000H

2.2 数据处理与存储

1. 移位操作

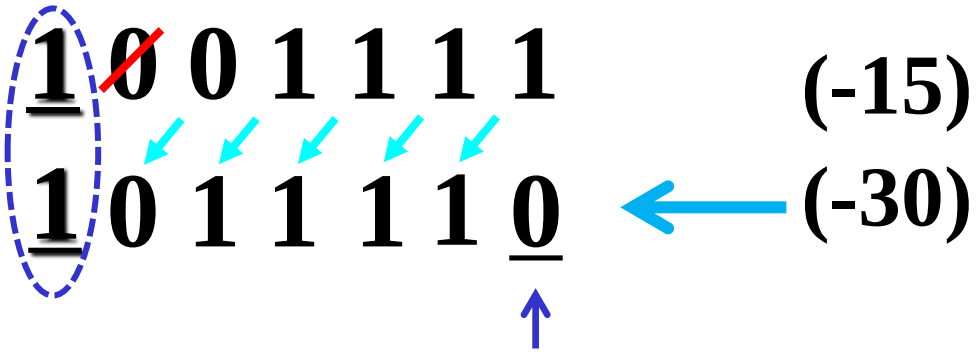
(1) 逻辑移位：数码位置变化。

循环左移：



(2) 算术移位：符号位不变、数码位置变化

算术左移：



数值发生2^n倍变化，n为移动的位数

2.2 数据处理与存储

①正数补码\原码移位规则

※移位规则

数符不变 （单：符号位不变；双：第1符号位不变）

空位补0 （右移时第2符号位移至尾数最高位）

（1）单符号位：

0 0111

左移 0 1110

右移 0 0111

右移 0 0011

（2）双符号位

00 0111

左移 00 1110

左移 01 1100

右移 00 1110

右移 00 0111

2.2 数据处理与存储

②负数补码移位

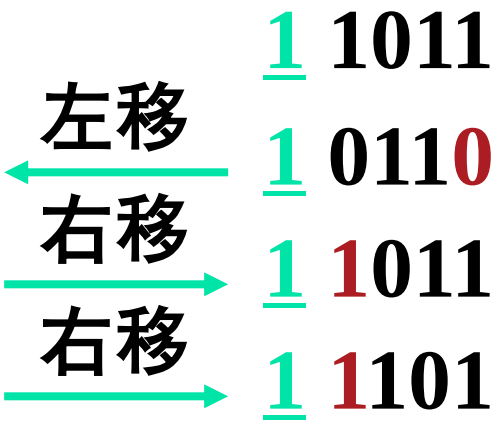
（1）移位规则

数符不变 （单：符号位不变；双：第1符号位不变）

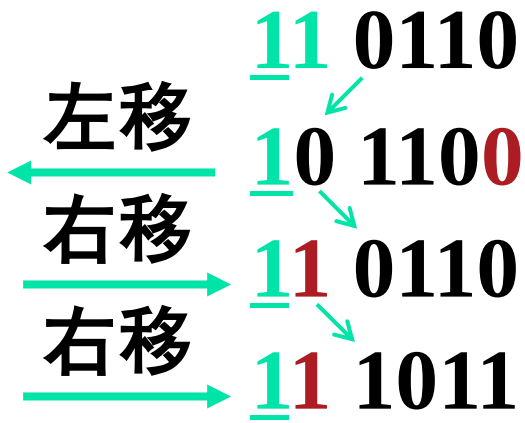
左移空位补0

右移空位补1（第二符号位移至尾数最高位）

（2）单符号位



（3）双符号位



2.2 数据处理与存储

※易出错处：

00 1110

左 ← ~~00 1100~~

正确： 01 1100

01 1100

右 → ~~00 0110~~

正确： 00 1110

11 0110

左 ← ~~11 1100~~

正确： 10 1100

10 1100

右 → ~~11 1110~~

正确： 11 0110

2.2 数据处理与存储

2. 舍入方法

① 0舍1入（原码、补码）

［例］保留4位尾数：

$0\ 0010\underline{0}0_{原} \longrightarrow 0\ 0010_{原}$

$1\ 0010\underline{1}_{原} \longrightarrow 1\ 0011_{原}$

$1\ 1101\underline{1}_{补} \longrightarrow 1\ 1110_{补}$

② 末位恒置1（原码、补码）

［例］保留4位尾数：

$0\ 0010\underline{0}0_{原} \longrightarrow 0\ 0011_{原}$

$1\ 0010\underline{1}_{原} \longrightarrow 1\ 0011_{原}$

$1\ 1101\underline{1}_{补} \longrightarrow 1\ 1101_{补}$

2.2 数据处理与存储

3. 数位扩展与压缩

（1）符号扩展　直接把符号位（0/1）填充到扩展位

000A ⟶ 0000000A

800A ⟶ FFFF800A

（2）0-扩展　高位均全补0（针对无符号数）

002A ⟶ 0000002A

F12C ⟶ 0000F12C

（3）位数压缩　弃高位、留低位

F12B800A ⟶ 800A

02A0F12C ⟶ F12C

2.2 数据处理与存储

4. 数据存储(按字节编址)

（1）小端模式/ Little-Endian

小地址单元存储数据的低位(即尾端)

FF	FF	00	01
#103	#102	#101	#100

⟹ FFFF0001

（2）大端模式/ Big-Endian

大地址单元存储数据的低位(即尾端)

FF	FF	00	01
#103	#102	#101	#100

⟹ 0100FFFF

2.3 定点运算

定点数一般用补码表示；

符号位参加运算。

重点：基于补码的加、减、乘、除法

2.3.1 补码加减运算

- **补码加减运算公式**

$$[A+B]_{补} = [A]_{补} + [B]_{补} \ (\text{MOD } 2^n)$$

$$[A-B]_{补} = [A]_{补} + [-B]_{补} \ (\text{MOD } 2^n)$$

$[-B]_{补} = [B_{补}]_{变补}$

B的符号置反后再表示成补码

$B_{补}$连同符号一起变反、末尾+1

- **补码加减运算规则**

（1）参加运算的两个操作数都用补码表示；

（2）符号位作为数的一部分参加运算；

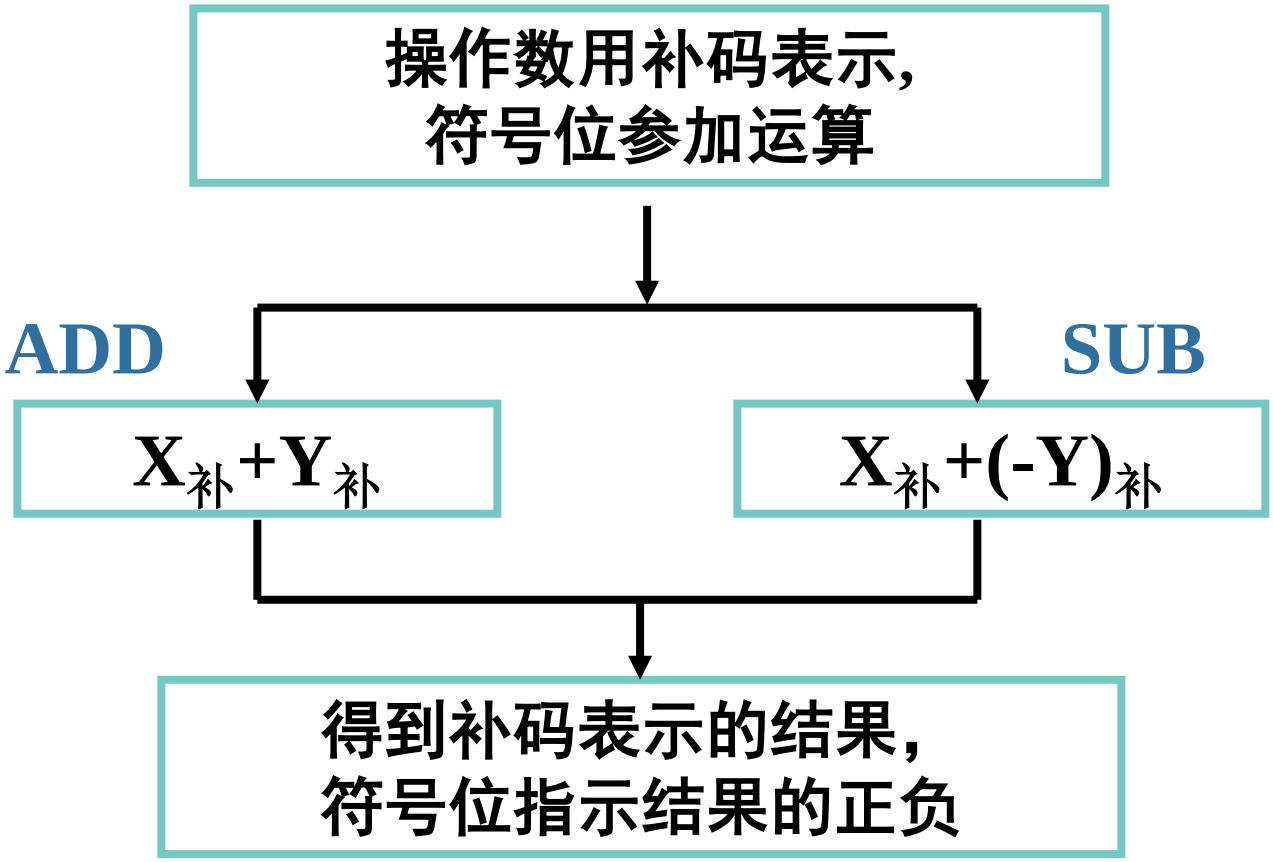
（3）若做加法，则两数直接相加；若做减法，则将减数变补后再与被减数相加；

（4）运算结果仍用补码表示；

（5）符号位的进位为模值，应该去掉。

2.3.1 补码加减运算

补码加减运算流程



2.3.1 补码加减运算

[例 1] 设 $A = 0.1011$，$B = -0.0101$　　求 $[A+B]_{补}$

解：
$$[A]_{补} = 0.1011$$
$$+\ [B]_{补} = 1.1011$$
$$[A]_{补} + [B]_{补} = 10.0110 = [A+B]_{补}$$

$\therefore A + B = 0.0110$

验证
$$\begin{array}{r} 0.1011 \\ -\ 0.0101 \\ \hline 0.0110 \end{array}$$

[例 2] 设机器数字长为 8 位（含 1 位符号位）
且 $A = 15$，$B = 24$，用补码求 $A - B$

解：$A = 15 = 0001111$　　$B = 24 = 0011000$

$[A]_{补} = 0,\ 0001111$　　$[B]_{补} = 0,\ 0011000$

$+\ [-B]_{补} = 1,\ 1101000$

$[A]_{补} + [-B]_{补} = 1,\ 1110111 = [A-B]_{补}$

$\therefore A - B = -1001 = -9$

补码加减运算逻辑实现

#控制信号

加法器输入端：

Sub：控制Mux和Cin

A：输入$A_{补}$

B：输入$B_{补}$

加法器输出端：

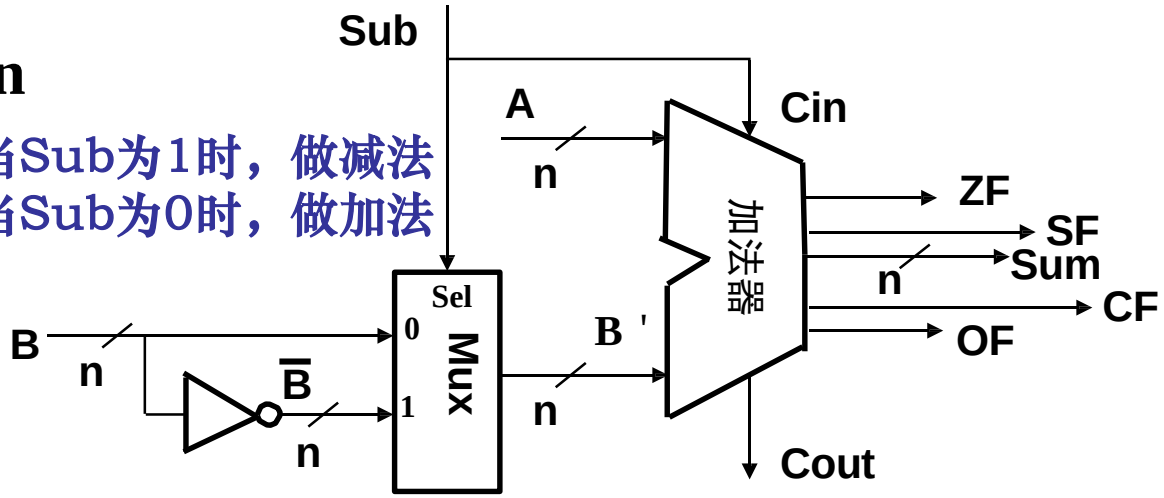
Sum: 加法结果 **Cout**:进位信号

ZF: 0标志位；**SF**：符号标志；

CF:进位/借位标志；**OF**:溢出标志

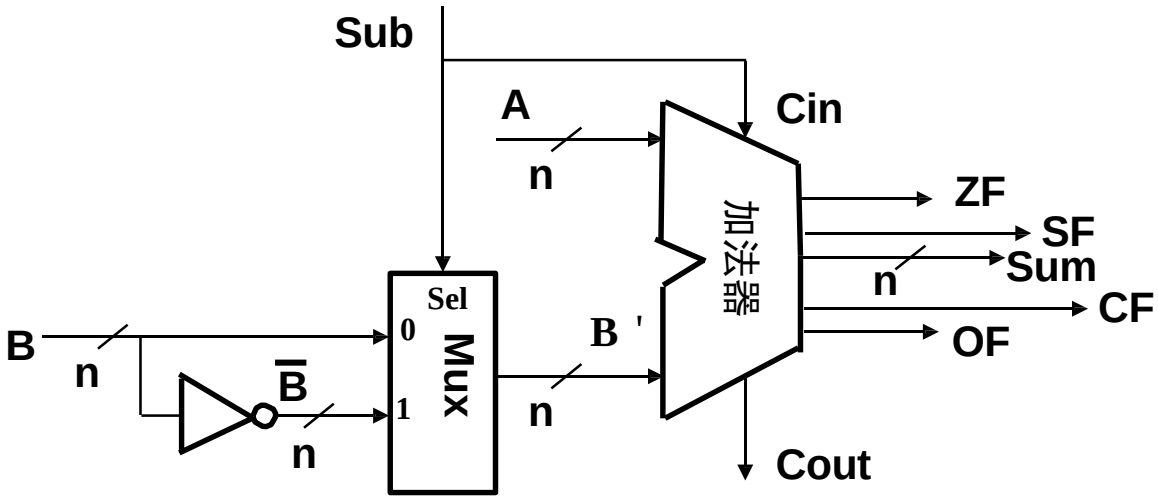
CF=Cout⊕Sub

#补码加减运算器框图



采用补码进行加减法运算，在计算机中只需要一套实现加法运算的电路，从而简化了计算机内部硬件电路的结构。

所有运算电路的核心



整数加/减运算部件

重要认识1： 计算机中所有运算都基于加法器实现！

重要认识2： 加法器不知道所运算的是带符号数还是无符号数。

重要认识3： 加法器不判定对错，总是取低n位作为结果，并生成标志信息。

问题：为什么要生成并保存条件标志？

思考

练习 设机器数字长为 8 位（含 1 位符号位）
且 $A = -97$，$B = +41$，用补码求 $A - B$

$A - B = +1110110 = +118$ **溢出！**

溢出：运算结果超出了机器能表示数的范围。

补码加减的溢出判断

1. 单符号位法

操作数$[A]_{补}$的符号位为S_A，$[B]_{补}$的符号位为S_B，结果的符号位为S_f。则判定溢出的逻辑表达式为：

$$溢出逻辑 = \overline{S}_A\overline{S}_B S_f + S_A S_B \overline{S}_f$$

（1）A=10 B=7

10+7： 0 1010
0 0111 +
正溢 1 0001

（2）A= -10 B= -7

-10+(-7)： 1 0110
1 1001 +
负溢 0 1111

补码加减的溢出判断

2. 进位判断法

最高数值位的进位信号为C，符号位的进位信号为C_f

则判定溢出的逻辑表达式为：

$$溢出逻辑 = C_f \oplus C$$

（2）A=10 B=7

10+7：

C_f=0

C =1

```
  0  1010
  0 10111 +
-------------
  1  0001
```

正溢

（4）A= -10 B= -7

-10+(-7)：

C_f=1

C =0

```
  1  0110
 11  1001 +
-------------
  0  1111
```

负溢

补码加减的溢出判断

3. 两位符号位判溢出

将符号位扩充为两位：S_{s1}和S_{s2}，则溢出表达式为：

$$溢出 = S_{s1} \oplus S_{s2}$$

结果的双符号位 相同　未溢出

结果的双符号位 不同　溢出

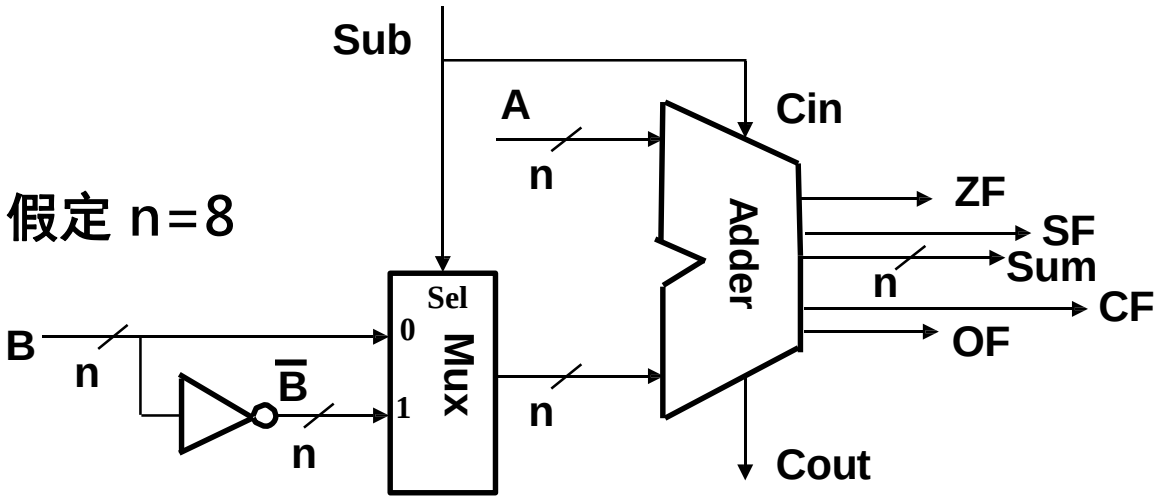
00. ×××××
11. ×××××
00 ×××××
11 ×××××
01. ×××××
10 ×××××
01 ×××××

最高符号位 代表其 真正的符号

整数减法举例

```
unsigned int x=134;
unsigned int y=246;
int m=x;
int n=y;
unsigned int z1=x-y;
unsigned int z2=x+y;
int k1=m-n;
int k2=m+n;
```

无符号和带符号加减运算都用该部件执行



x和m的机器数一样：1000 0110，y和n的机器数一样：1111 0110

z1和k1的机器数一样：1001 0000，CF=1，OF=0，SF=1

z1的值为144（=134-246+256，x-y<0），k1的值为-112。

无符号减公式：

$$result=\begin{cases} x-y & (x-y\geqslant 0) \\ x-y+2^n & (x-y<0) \end{cases}$$

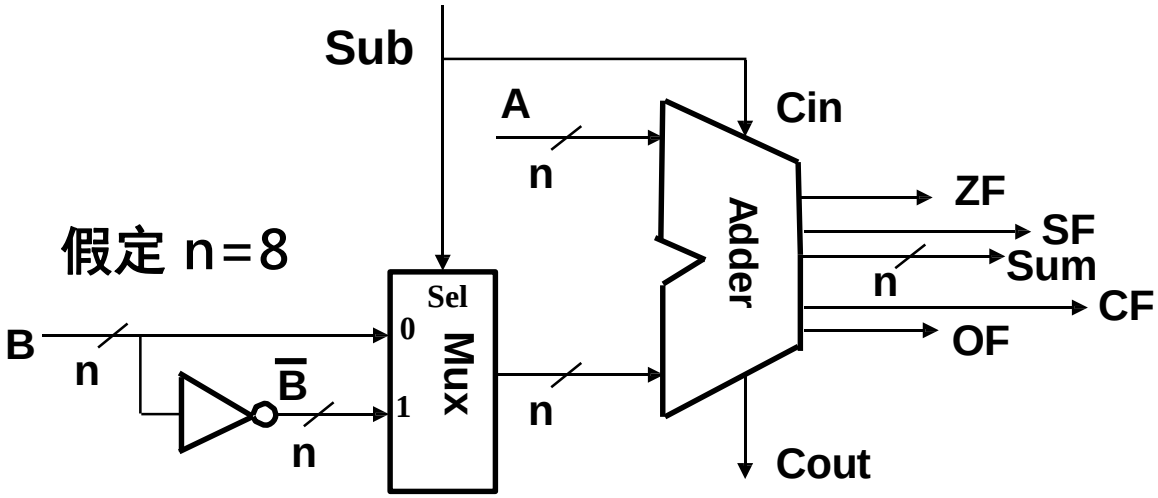
带符号减公式：

$$result=\begin{cases} x-y-2^n & (2^{n-1}\leqslant x-y) \quad \text{正溢出} \\ x-y & (-2^{n-1}\leqslant x-y<2^{n-1}) \quad \text{正常} \\ x-y+2^n & (x-y<-2^{n-1}) \quad \text{负溢出} \end{cases}$$

整数加法举例

```
unsigned int x=134;
unsigned int y=246;
int m=x;
int n=y;
unsigned int z1=x-y;
unsigned int z2=x+y;
int k1=m-n;
int k2=m+n;
```

无符号和带符号加减运算都用该部件执行



x和m的机器数一样：1000 0110，y和n的机器数一样：1111 0110

z2和k2的机器数一样：0111 1100，CF=1，OF=1，SF=0

z2的值为124（=134+246-256，x+y>256）

k2的值为124（=-122+(-10)+256，m+n=-132<-128，即负溢出）

无符号加公式：

$$result=\begin{cases} x+y & (x+y<2^n) \\ x+y-2^n & (2^n\leqslant x+y<2^{n+1}) \end{cases}$$

带符号加公式：

$$result=\begin{cases} x+y-2^n & (2^{n-1}\leqslant x+y) \quad \text{正溢出} \\ x+y & (-2^{n-1}\leqslant x+y<2^{n-1}) \quad \text{正常} \\ x+y+2^n & (x+y<-2^{n-1}) \quad \text{负溢出} \end{cases}$$

2.3.2 定点乘法运算

- √ 原码一位乘
- √ 补码一位乘
- √ 原码两位乘
- √ 补码两位乘

原码一位乘

原码乘法 ⟶ **部分积累加、移位。**

X原　　Y原

↓　　↓

[例] 0.1101×1.1011

乘积 $P = |X| \times |Y|$

符号 $S_P = S_X \oplus S_Y$

原码一位乘

手工运算

$$
\begin{array}{rl}
 & 0.1101 \quad \leftarrow |X| \\
\times & 0.1011 \quad \leftarrow |Y| \\
\hline
 & 1101 \\
 & 1101 \\
 & 0000 \\
+ & 1101 \\
\hline
 & .10001111
\end{array}
$$

部分积

①加数只为|X|或0

②个数为|Y|的位数

添加符号： 1.10001111

思考：1）加数的个数增多情况

2）加数的位数增多的情况

解决办法：可将1次总加改为分步移位累加

原码一位乘

（1）算法原理

每次将1位乘数所对应的部分积与原部分积的累加和相加，并移位

设置寄存器：

A：存放部分积累加和、乘积高位

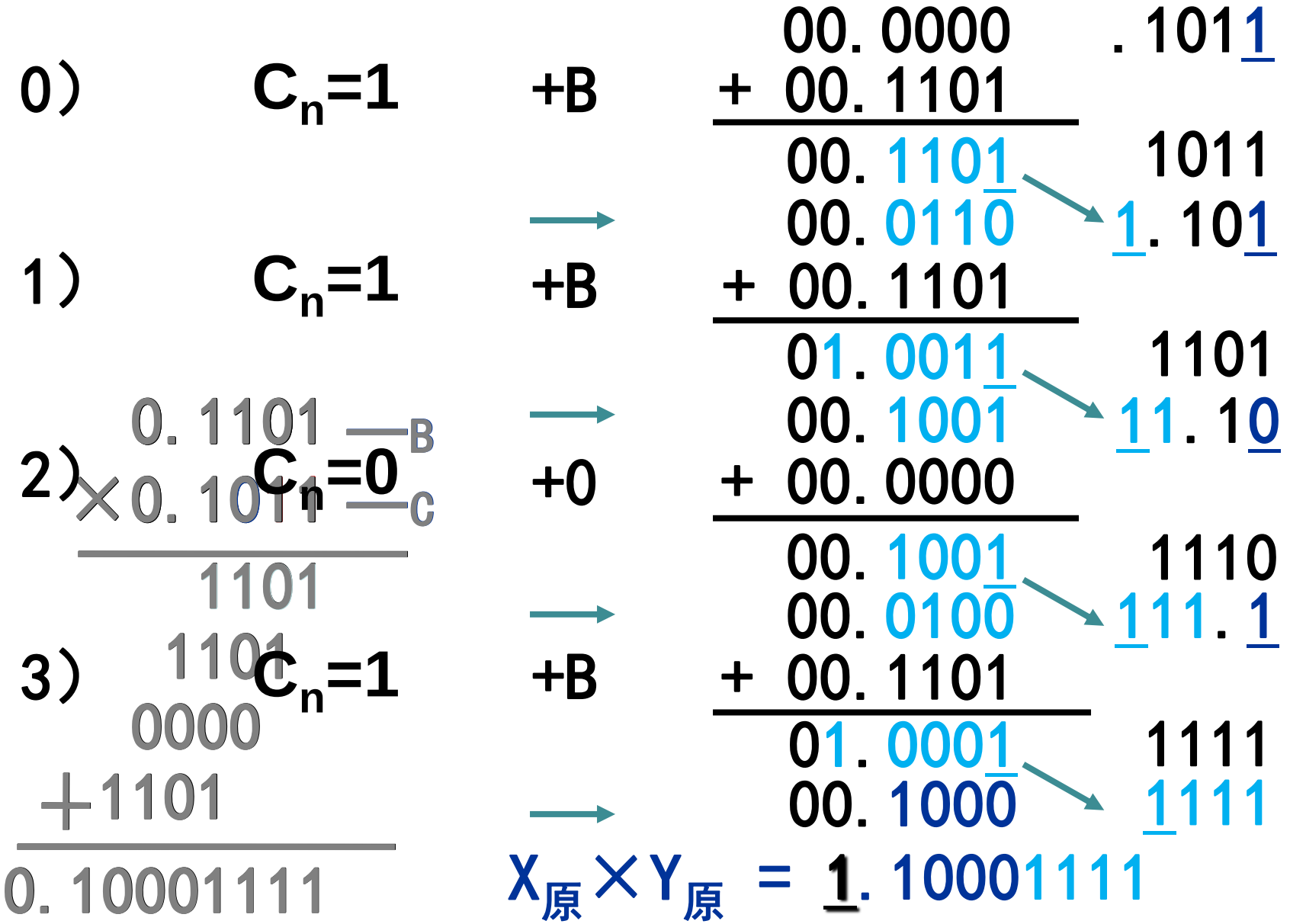
B：存放被乘数

C：存放乘数、乘积低位

设置初值：A = 00.0000

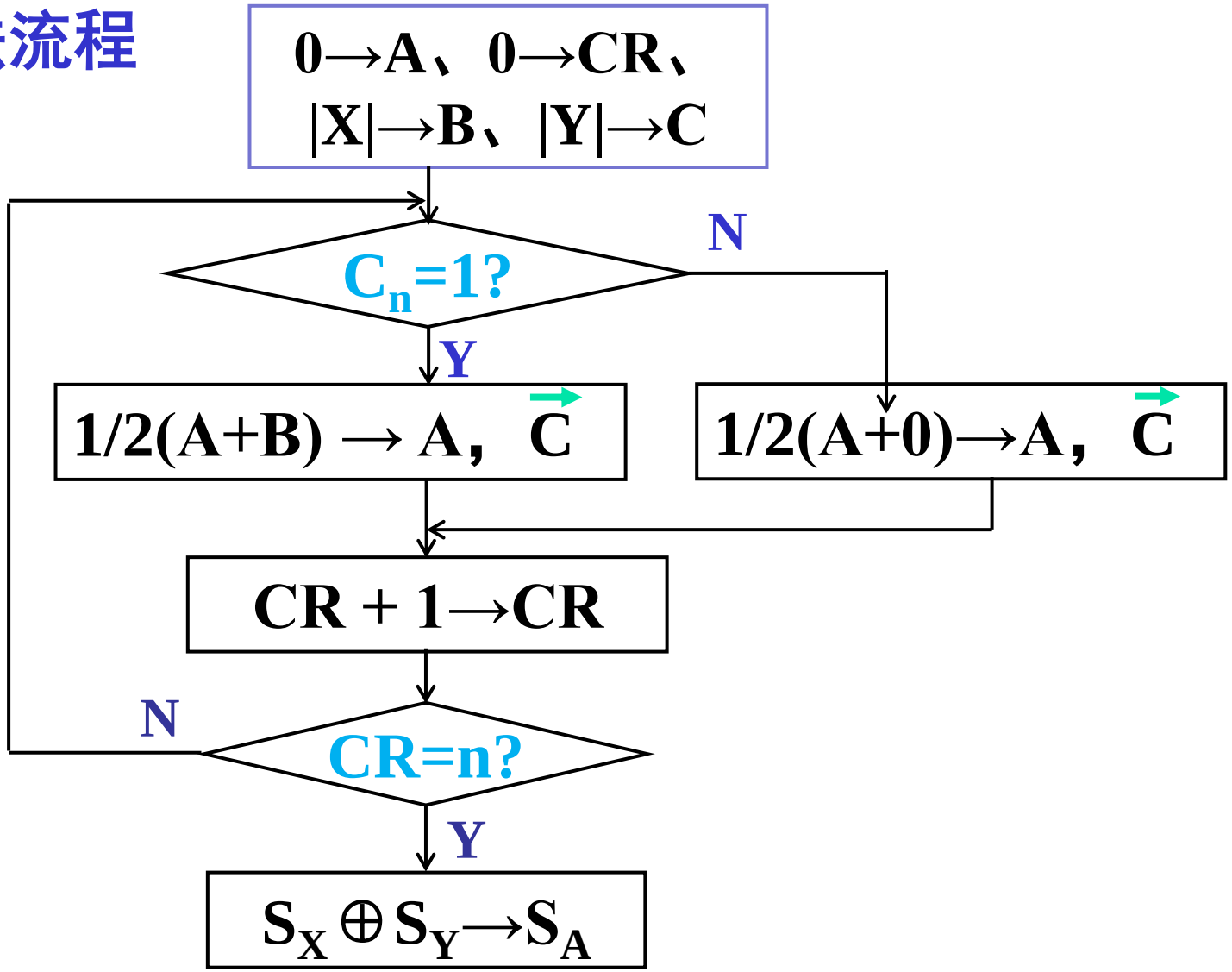
B = |X| = 00.1101

C = |Y| = .1011

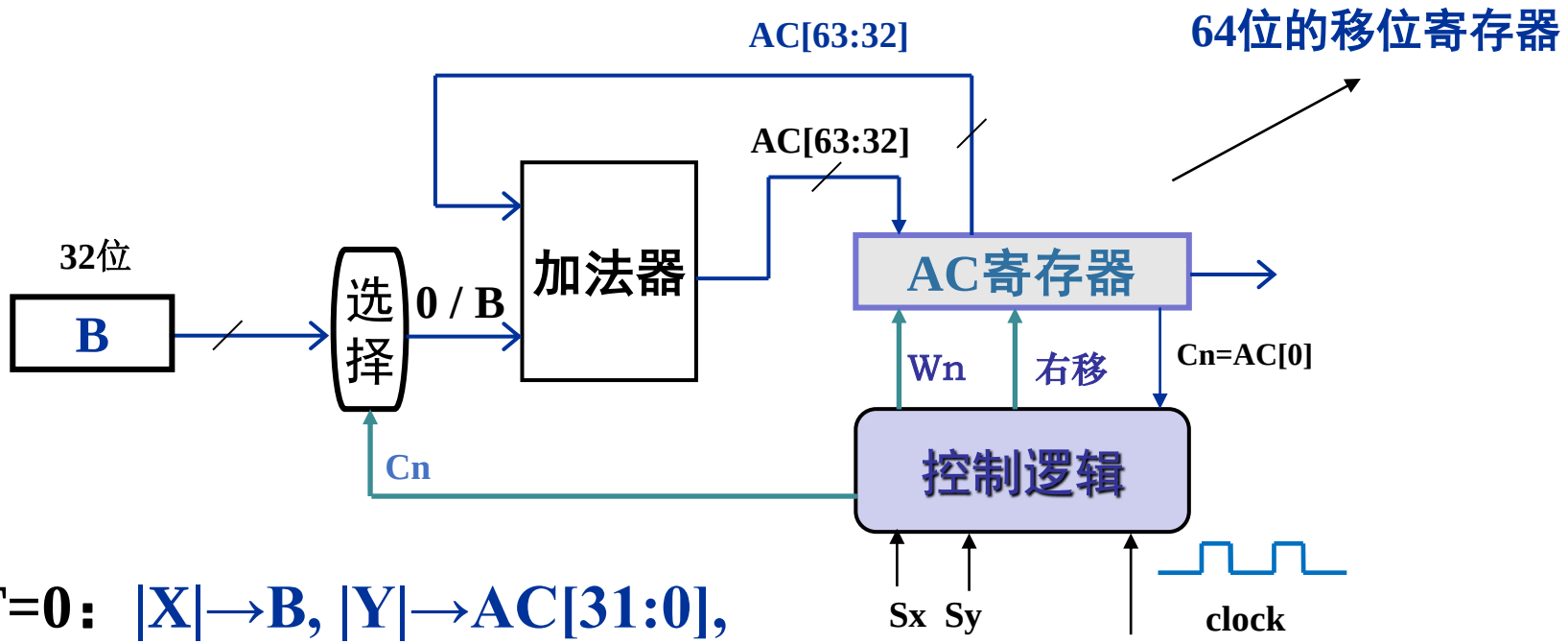

步数CR 条件
操作
A
C
C_n
00. 0000 . 1011
0）
C_n=1
+B
+ 00. 1101
00. 1101 1011
00. 0110 1. 101
1）
C_n=1
+B
+ 00. 1101
01. 0011 1101
00. 1001 11. 10
0. 1101 —B
2）
C_n=0
+0
+ 00. 0000
×0. 1011 —C
00. 1001 1110
1101
00. 0100 111. 1
1101
3）
C_n=1
+B
+ 00. 1101
0000
01. 0001 1111
+1101
00. 1000 1111
0. 10001111
$X_{原} \times Y_{原}$ = 1. 10001111

原码一位乘

（2）算法流程


0→A、0→CR、
|X|→B、|Y|→C
C_n=1?
N
Y
1/2(A+B) → A，C
1/2(A+0)→A，C
CR + 1→CR
CR=n?
N
Y
$S_X \oplus S_Y \rightarrow S_A$

（3）32位硬件逻辑方案



T=0：|X|→B, |Y|→AC[31:0],
0→AC[63:32]

T=1：AC[63:32]+0/B→AC[63:32], AC右移

T=2：

⋮

T=n：同上，然后置符号。

补码一位乘

※Booth（比较法）

$[XY]_{补}=[A_n]_{补}+(Y_1-Y_0)\times[X]_{补}$

Y_n (高位)	Y_{n+1} (低位)	运算操作
0	0	$\frac{1}{2}\times A_{补}$
0	1	$\frac{1}{2}\times (A_{补}+X_{补})$
1	0	$\frac{1}{2}\times (A_{补}- X_{补})$
1	1	$\frac{1}{2}\times A_{补}$

* 乘数尾添加Y_{n+1}，循环判别Y_nY_{n+1}，根据上表计算，再整体右移1位（即$\frac{1}{2}\times$）。

补码一位乘

［例］X= -0.1101，Y= -0.1011，计算$[XY]_{补}$

A=00.0000 $B=X_{补}=11.0011$ $-B=-X_{补}=001101$ $C=Y_{补}=1.0101$

步数	C_nC_{n+1}条件	操作	A	C	C_{n+1}	
			000000	10101	0	
1	1 0	-B	+ 001101			
			001101	10101	0	右移1位→
			000110	11010	1	
2	0 1	+B	+ 110011			
			111001	11010	1	右移1位→
			111100	11101	0	

补码一位乘

步数	C_nC_{n+1}条件	操作	111100 11101 0
3	1 0	-B	+ 001101
			1 001001 11101 0 右移1位→
			000100 11110 1
4	0 1	+B	+ 110011
			110111 11110 1 右移1位→
			111011 11111 0
校正	1 0	-B	+ 001101
			1 001000 11111 0 保持不变!!

$[XY]_{补}$ = 001000 1111

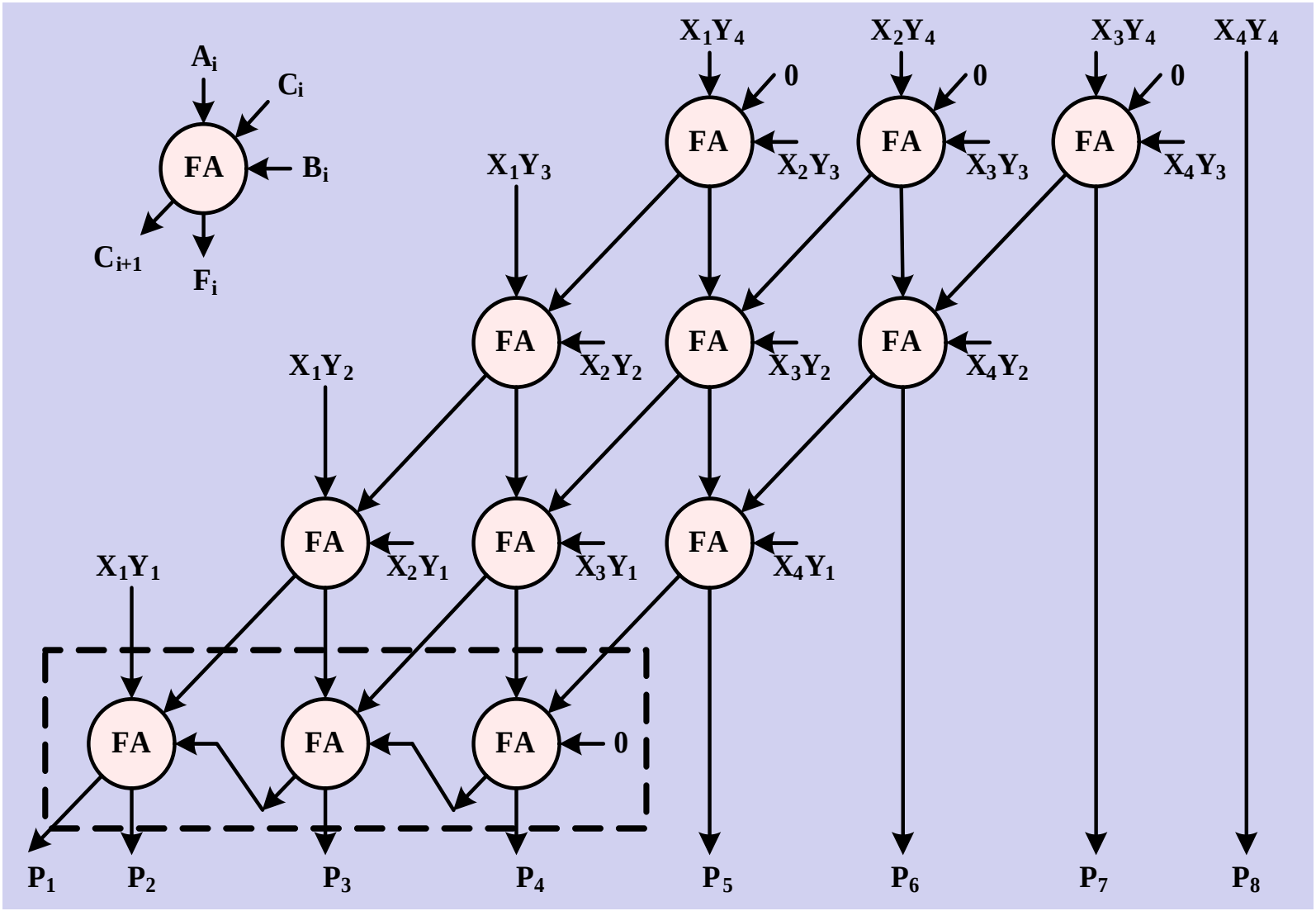
= +0.1000 1111

阵列乘法器

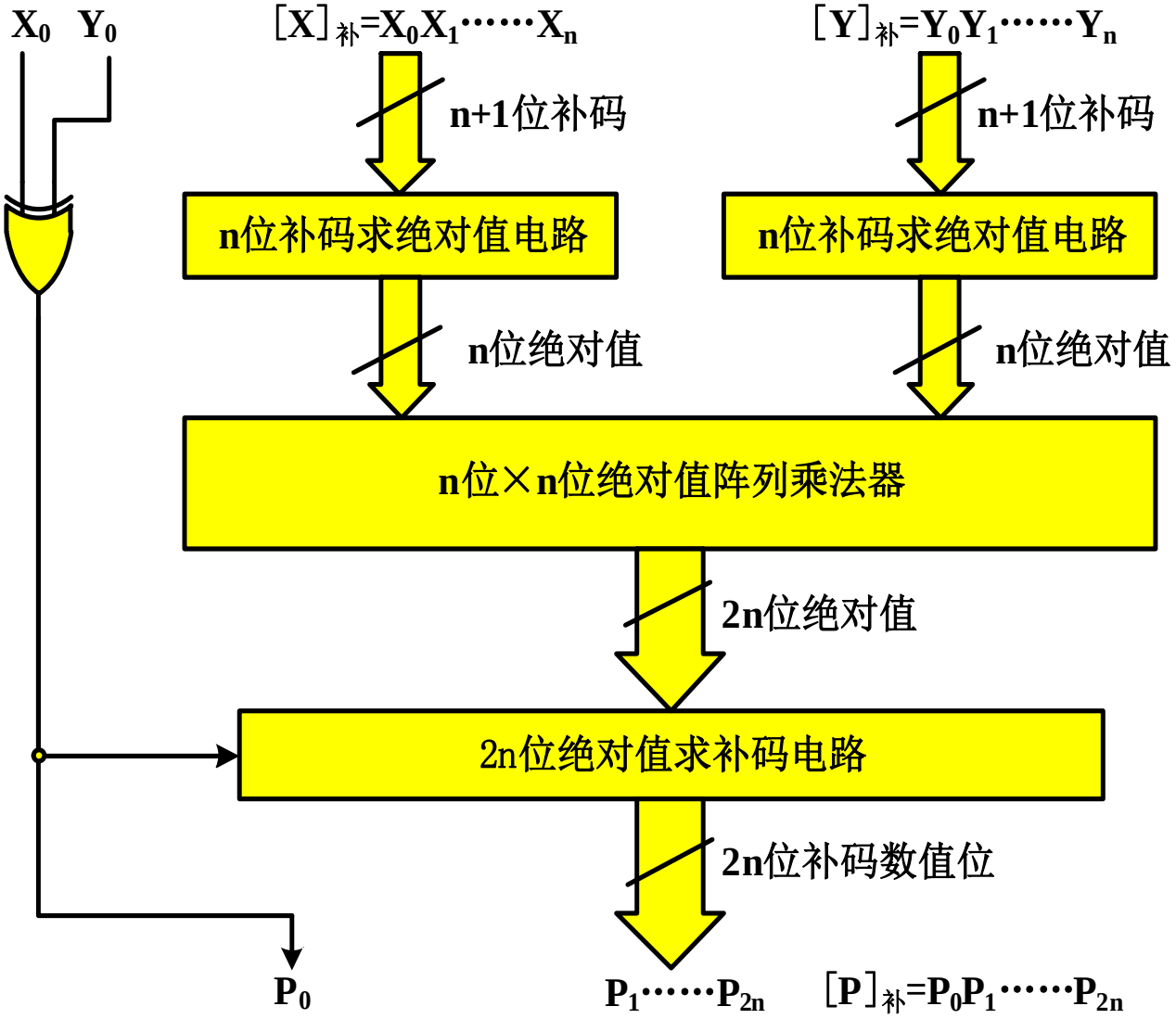
- 原理类似于二进制手工算法
  - 位积的每一位X_iY_j都可以用一个与门实现，而每一位的相加均可以使用一个全加器来实现。

$$
\begin{array}{cccccccc}
 & & & X_1 & X_2 & X_3 & X_4 & \\
 & & \times & Y_1 & Y_2 & Y_3 & Y_4 & \\
\hline
 & & & X_1Y_4 & X_2Y_4 & X_3Y_4 & X_4Y_4 & \\
 & & X_1Y_3 & X_2Y_3 & X_3Y_3 & X_4Y_3 & & \\
 & X_1Y_2 & X_2Y_2 & X_3Y_2 & X_4Y_2 & & & \\
+ & X_1Y_1 & X_2Y_1 & X_3Y_1 & X_4Y_1 & & & \\
\hline
P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 & P_8
\end{array}
$$

绝对值阵列乘法器


A_i
C_i
FA
B_i
C_{i+1}
F_i
X_1Y_4
X_2Y_4
X_3Y_4
X_4Y_4
0
X_1Y_3
X_2Y_3
X_3Y_3
X_4Y_3
X_1Y_2
X_2Y_2
X_3Y_2
X_4Y_2
X_1Y_1
X_2Y_1
X_3Y_1
X_4Y_1
P_1
P_2
P_3
P_4
P_5
P_6
P_7
P_8

补码阵列乘法器


X_0 Y_0
$[X]_{补}=X_0X_1\cdots\cdots X_n$
$[Y]_{补}=Y_0Y_1\cdots\cdots Y_n$
n+1位补码
n+1位补码
n位补码求绝对值电路
n位补码求绝对值电路
n位绝对值
n位绝对值
n位×n位绝对值阵列乘法器
2n位绝对值
2n位绝对值求补码电路
2n位补码数值位
P_0
$P_1\cdots\cdots P_{2n}$
$[P]_{补}=P_0P_1\cdots\cdots P_{2n}$

2.3.3 定点除法运算

- √ 补码不恢复余数除法
- √ 补码恢复余数的除法
- √ 原码恢复/不恢复余数除法

补码不恢复余数除法

[算法思想]（|X|<|Y|）

被除数$X_{补}$、除数$Y_{补}$、余数r_i，i=0、1…

初始化：令$r_0=X_{补}$，比较r_0与$Y_{补}$符号，同号上商1，异号上商0

循环：i=1…n，按下表条件决定每步操作

r_i、$Y_{补}$数符	商	对应操作
同号	1	$r_{i+1}=2\times[r_i]_{补}-Y_{补}$
异号	0	$r_{i+1}=2\times[r_i]_{补}+Y_{补}$

$[-Y_{补}]\leftrightarrow +[Y_{补}]_{变补}$

$\times 2 \leftrightarrow$ 左移1位

商修正：符号位+1，末尾恒置1

余数修正：左移了n次，则余数=$2^{-n}\times r_n$

补码不恢复余数除法

[例]X÷Y=+0.1000÷(-0.1010)=?

$R=X_{补}=\underline{00}1000$，$B=Y_{补}=\underline{11}0110$，$-B=001010$，$Q=00000$

步数	条件	操作	被除数/余数		商Q
1	$r_0Y_{补}$异号	上商0	001000	r_0	00000
		$2r_0$/←	010000		0000<u>0</u>
			+ 110110		
			1 000110	r_1	
2	$r_1Y_{补}$异号	上商0	000110		00000
		$2r_1$/←	001100		0000<u>0</u>
			+ 110110		
			1 000010	r_2	

补码不恢复余数除法

$R=X_{补}=\underline{00}1000$，$B=Y_{补}=\underline{11}0110$，$-B=001010$

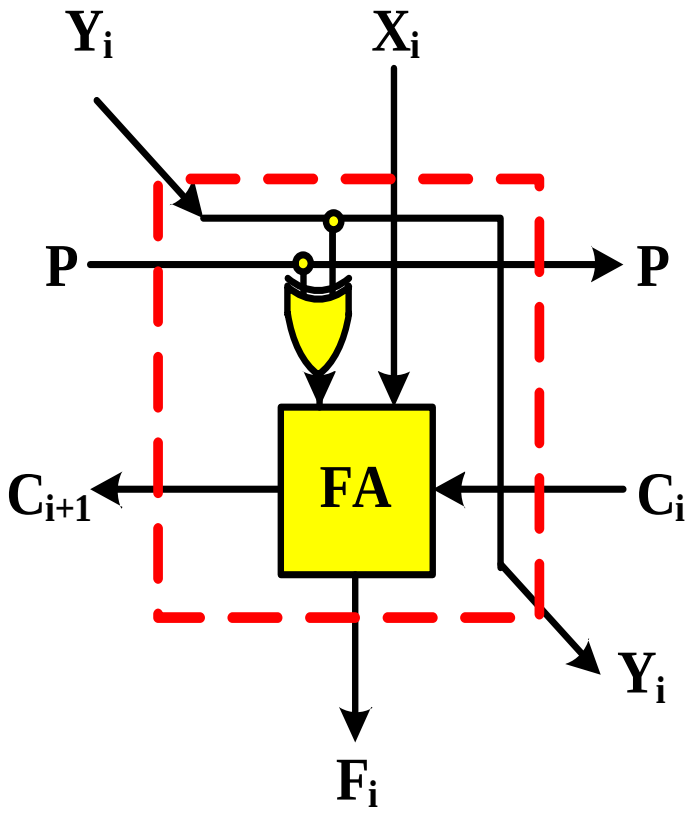
			000010 r_2	00000
3	$r_2Y_{补}$异号	上商0	000010	00000
		$2r_2$/←	000100	00000
			+ 110110	
			111010 r_3	
4	$r_3Y_{补}$同号	上商1	111010	00001
		$2r_3$/←	110100	00010
			+ 001010	+1 置1
			111110 r_4	

商校正：左移了4位 $10011_{补}$

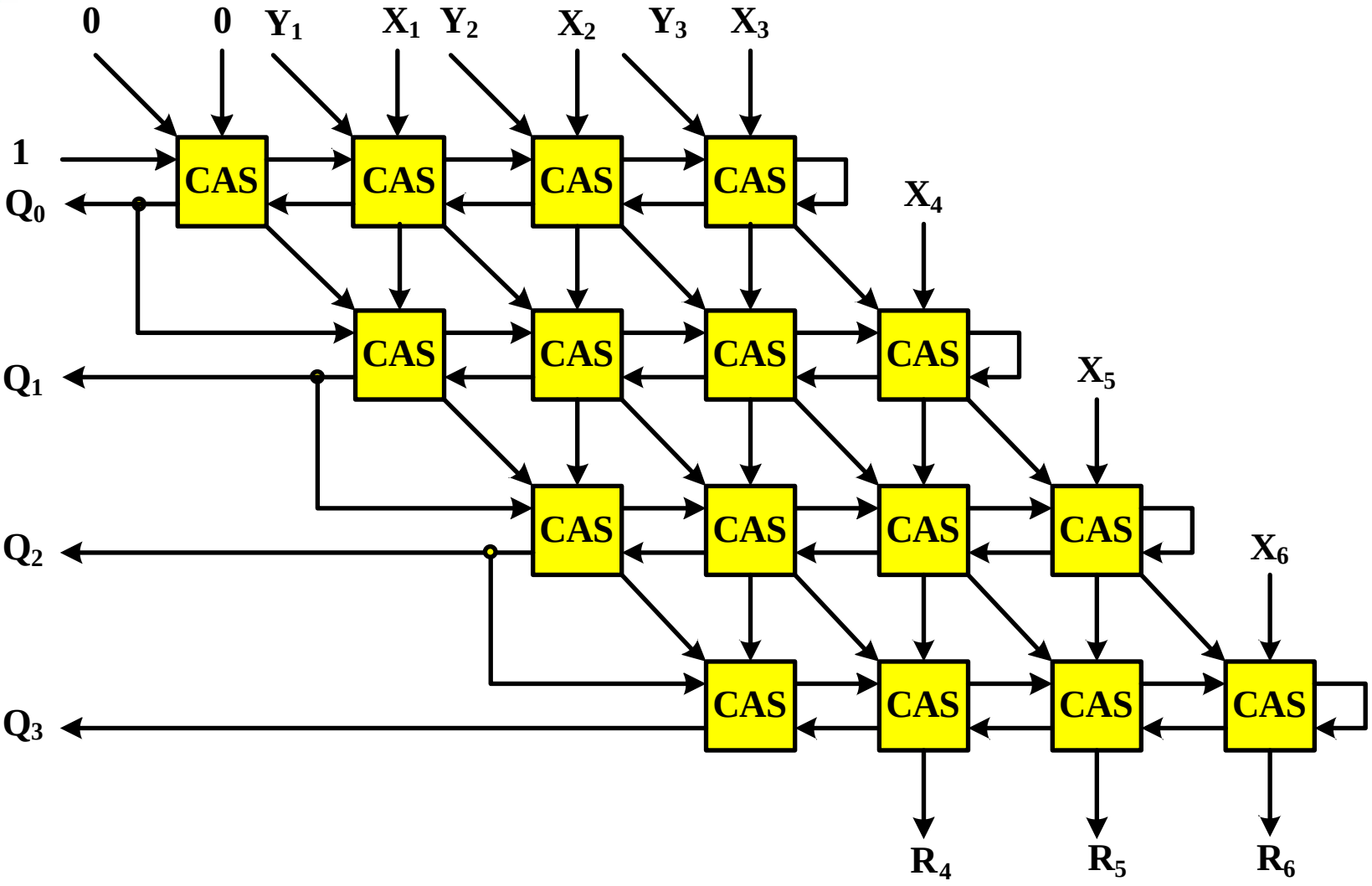
余数：$2^{-4} \times r_4$ 数符+1、末尾恒置1

阵列除法器

- 被除数X=X_1 X_2 X_3 X_4 X_5 X_6，除数Y=Y_1 Y_2 Y_3得到的商Q=$Q_1Q_2Q_3$（Q0=0），R=R_4 R_5 R_6。
- 若为定点小数，则X=0.X_1 X_2 X_3 X_4 X_5 X_6，除数Y=0.Y_1 Y_2 Y_3，得到的商Q=0.$Q_1Q_2Q_3$（Q_0=0），R=0.000 R_4 R_5 R_6
- 构成的基本部件：可控加减单元CAS



阵列除法器



2.3 浮点运算

一、浮点加减运算

$$X = Mx \cdot 2^{Ex} \qquad Y = My \cdot 2^{Ey}$$

(1) 求阶差：$\Delta E = Ex - Ey$

若Ey > Ex，则结果的阶码为Ey

(2) 对阶

将Mx右移ΔE位，尾数变为 $Mx \times 2^{Ex-Ey}$；

(3) 尾数加减

$Mx \times 2^{Ex-Ey} \pm My$

(4) 结果规格化

(5) 溢出判断

2.3 浮点运算

对阶

(1) 求阶差

$$\Delta E = E_x - E_y = \begin{cases} = 0 & E_x = E_y \quad \text{已对齐} \\ > 0 & E_x > E_y \begin{cases} x \text{ 向 } y \text{ 看齐} & M_x \leftarrow 1, \\ y \text{ 向 } x \text{ 看齐} & \checkmark\ M_y \rightarrow 1, \end{cases} \\ < 0 & E_x < E_y \begin{cases} x \text{ 向 } y \text{ 看齐} & \checkmark\ M_x \rightarrow 1, \\ y \text{ 向 } x \text{ 看齐} & M_y \leftarrow 1, \end{cases} \end{cases}$$

(2) 对阶原则

小阶向大阶看齐

2.3 浮点运算

规格化

(1) 规格化数的定义　　$r=2$　　$\frac{1}{2} \le |S| < 1$

(2) 规格化数的判断

$M>0$	规格化形式	$M<0$	规格化形式
真值	0.1×× …×	真值	–0.1×× … ×
原码	0.1×× …×	原码	1.1×× … ×
补码	0.1×× …×	补码	1.0×× … ×

原码　不论正数、负数，第一数位为1

补码　符号位和第 1 数位不同

2.3 浮点运算

规格化

(1) 当尾数最高位为0，需左规

尾数 ←1，阶码减 1

数符和第一数位不同或第一数位为1

(2) 当尾数最高位有进位，需右规

尾数溢出（ >1）时，需 右规

即尾数出现 01. ×× … ×或 10. ×× …×时

尾数 ⟶1，阶码加 1

2.3 浮点运算

舍入

在 对阶 和 右规 过程中，可能出现 尾数末位丢失 引起误差，需考虑舍入

(1) 0舍1入法:

若右移出的是1则在最低位加1

(2) 末尾恒置“1”法:

只要数字位1被移掉，就将最后一位恒置成1

2.3 浮点运算

溢出判断

浮点数的溢出标志：阶码溢出

- **阶码上溢：阶码的符号位为 01，则结果溢出**
- **阶码下溢：阶码的符号位为 10，则结果为0**

尾数为0说明结果应该为0（阶码和尾数为全0）。

2.3 浮点运算

[例 1] $x = (-\frac{5}{8}) \times 2^{-5}$ $y = (\frac{7}{8}) \times 2^{-4}$

求 $x-y$（除阶符、数符外，阶码取 3 位，尾数取 6 位）

解：$x = (-0.101000) \times 2^{-101}$ $y = (0.111000) \times 2^{-100}$

$[x]_{补} = 11,011;\ 11.011000$ $[y]_{补} = 11,100;\ 00.111000$

① 对阶 $[\Delta E]_{补} = [E_x]_{补} - [E_y]_{补}$

$$\begin{array}{r} 11,011 \\ +\ 00,100 \\ \hline 11,111 \end{array}$$

阶差为 –1 ∴ $M_x \longrightarrow 1$, $E_x + 1$

∴ $[x]_{补'} = 11,100;\ 11.101100$

2.3 浮点运算

[例 1] $x = (-\frac{5}{8}) \times 2^{-5}$ $y = (\frac{7}{8}) \times 2^{-4}$

求 $x-y$（除阶符、数符外，阶码取 3 位，尾数取 6 位）

解：② 尾数求和

$$[M_x]_{补'} = 11.101100$$
$$+ \quad [-M_y]_{补} = 11.001000$$
$$\overline{\qquad\qquad\qquad}$$
$$110.110100$$

③ 右规 $[x+y]_{补} = 11,100;\ 10.110100$

右规后 $[x+y]_{补} = 11,101;\ 11.011010$

$$\therefore\ x - y = (-0.100110) \times 2^{-11}$$
$$= (-\frac{19}{32}) \times 2^{-3}$$

浮点数舍入举例

例：将同一实数分别赋值给单精度和双精度类型变量，然后打印输出。

```
#include <stdio.h>
main()
{
    float a;
    double b;
    a = 123456.789e4;
    b = 123456.789e4;
    printf( “%f/n%f/n” ,a,b);
}
```

运行结果如下：

```
1234567936.000000
1234567890.000000
```

为什么float情况下输出的结果会比原来的大？这到底有没有根本性原因还是随机发生的？为什么会出现这样的情况？

float可精确表示7个十进制有效数位，后面的数位是舍入后的结果，舍入后的值可能会更大，也可能更小。

问题：为什么同一个实数赋值给float型变量和double型变量，输出结果会有所不同呢？

C语言中的浮点数类型

- C语言中有float和double类型，分别对应IEEE 754单精度浮点数格式和双精度浮点数格式
- long double类型的长度和格式随编译器和处理器类型的不同而有所不同，IA-32中是80位扩展精度格式
- 从int转换为float时，不会发生溢出，但可能有数据被舍入
- 从int或 float转换为double时，因为double的有效位数更多，故能保留精确值
- 从double转换为float和int时，可能发生溢出，此外，由于有效位数变少，故可能被舍入
- 从float 或double转换为int时，因为int没有小数部分，所以数据可能会向0方向被截断

浮点数比较运算举例

- 对于以下给定的关系表达式，判断是否永真。

```
int x ;
float f ;
double d ;
```

Assume neither d nor f is NaN

x == (int)(float) x 否

x == (int)(double) x 是

f == (float)(double) f 是

d == (float) d 否

f == -(-f); 是

2/3 == 2/3.0 否

d < 0.0 ⇒ ((d*2) < 0.0) 是

d > f ⇒ -f > -d 是

d * d >= 0.0 是

x*x>=0 否

(d+f)-d == f 否

自己写程序
测试一下！

IEEE754 的范围和精度

- 单精度浮点数（float型）的表示范围多大?

  最大的数据: $+1.11\ldots1X\ 2^{127}$ 约 $+3.4 \times 10^{38}$

  双精度浮点数（double型）呢? 约 $+1.8 \times 10^{308}$

- 以下关系表达式是否永真?

```
if ( i == (int) ((float) i) )
      {printf ( "true" );
}
```

Not always true!

How about double? True!

```
if ( f == (float) ((int) f) )
      {printf ( "true" );
}
```

Not always true!

How about double? 同 float

- 浮点数加法结合律是否正确? FALSE!

  $x = -1.5 \times 10^{38},\quad y = 1.5 \times 10^{38},\quad z = 1.0$

  $(x+y)+z = (-1.5\times10^{38}+1.5\times10^{38}) + 1.0 = 1.0$

  $x+(y+z) = -1.5\times10^{38} + (1.5\times10^{38}+1.0) = 0.0$

举例：Ariana火箭爆炸

- 1996年6月4日，Ariana 5火箭初次航行，在发射仅仅37秒钟后，偏离了飞行路线，然后解体爆炸，火箭上载有价值5亿美元的通信卫星。
- 原因是在将一个64位浮点数转换为16位带符号整数时，产生了溢出异常。溢出的值是火箭的水平速率，这比原来的Ariana 4火箭所能达到的速率高出了5倍。在设计Ariana 4火箭软件时，设计者确认水平速率决不会超出一个16位的整数，但在设计Ariana 5时，他们没有重新检查这部分，而是直接使用了原来的设计。
- 在不同数据类型之间转换时，往往隐藏着一些不容易被察觉的错误，这种错误有时会带来重大损失，因此，编程时要非常小心。

举例：Ariana火箭爆炸

- 1991年2月25日，海湾战争中，美国在沙特阿拉伯达摩地区设置的爱国者 导弹拦截伊拉克的飞毛腿导弹失败，致使飞毛腿导弹击中了一个美军军营，杀死了美军28名士兵。其原因是由于爱国者导弹系统时钟内的一个软件错误造成的，引起这个软件错误的原因是浮点数的精度问题。
- 爱国者导弹系统中有一内置时钟，用计数器实现，每隔0.1秒计数一次。程序用0.1的一个24位定点二进制小数x来乘以计数值作为以秒为单位的时间
- 这个x的机器数是多少呢?
- 0.1的二进制表示是一个无限循环序列：0.00011[0011]…，

  x=0.000 1100 1100 1100 1100 1100B。

  显然，x是0.1的近似表示，0.1-x

= 0.000 1100 1100 1100 1100 1100 [1100]… -
  0.000 1100 1100 1100 1100 1100B，即为：
=0.000 0000 0000 0000 0000 0000 1100 [1100]…B

$=2^{-20}\times 0.1 \approx 9.54\times 10^{-8}$ 这就是机器值与真值之间的误差！

举例：Ariana火箭爆炸

已知在爱国者导弹准备拦截飞毛腿导弹之前，已经连续工作了100小时，飞毛腿的速度大约为2000米/秒，则由于时钟计算误差而导致的距离误差是多少?

100小时相当于计数了$100\times60\times60\times10=36\times10^5$次，因而导弹的时钟已经偏差了$9.54\times10^{-8}\times36\times10^5$ 0.343秒

因此，距离误差是2000×0.343秒 687米

小故事：实际上，以色列方面已经发现了这个问题并于1991年2月11日知会了美国 陆军及爱国者计划办公室（软件制造商）。以色列方面建议重新启动爱国者系统的 电脑作为暂时解决方案，可是美国陆军方面却不知道每次需要间隔多少时间重新启 动系统一次。1991年2月16日，制造商向美国陆军提供了更新软件，但这个软件最 终却在飞毛腿导弹击中军营后的一天才运抵部队。

举例：Ariana火箭爆炸

- 若x用float型表示，则x的机器数是什么？0.1与x的偏差是多少？系统运行100小时后的时钟偏差是多少？在飞毛腿速度为2000米/秒的情况下，预测的距离偏差为多少？
  - 0.1= 0.0 0011[0011]B=+1.1 0011 0011 0011 0011 0011 00B×2^{-4}，x的机器数为0 011 1101 1 100 1100 1100 1100 1100 1100
  - Float型仅24位有效位数，后面的有效位全被截断，故x与0.1之间的误差为：|x–0.1|=0.000 0000 0000 0000 0000 0000 0000 1100 [1100]...B。这个值等于2^{-24}×0.1 ≈ 5.96×10^{-9}。100小时后时钟偏差5.96×10^{-9}×36×10^{5}≈0.0215秒。距离偏差0.0215×2000≈43米。比爱国者导弹系统精确约16倍。
- 若用32位二进制定点小数x=0.000 1100 1100 1100 1100 1100 1100 1101 B表示0.1，则误差比用float表示误差更大还是更小？
  - 当x=0.000 1100 1100 1100 1100 1100 1100 1101 B时，与0.1之间的误差约为：|x–0.1|=0.000 0000 0000 0000 0000 0000 0000 0000 00 1100 [1100]...B。这个值等于2^{-30}×0.1 ≈ 9.31×10^{-11}。
    100小时后时钟偏差 9.31×10^{-11}×36×10^{5}≈0.000335秒。
    预测的距离偏差仅为0.000335×2000≈0.67米。

举例：浮点数运算的精度问题

- 从上述结果可以看出：
  - 用32位定点小数表示0.1 ，比采用float精度高64倍
  - 用float表示在计算速度上更慢，必须先把计数值转换为IEEE 754格式浮点数，然后再对两个IEEE 754格式的数相乘，故采用float比直接将两个二进制数相乘要慢
- Ariana 5火箭和爱国者导弹的例子带来的启示

✓ 程序员应对底层机器级数据的表示和运算有深刻理解

✓ 计算机世界里，经常是“差之毫厘，失之千里”，需要细心再细心，精确再精确

✓ 不能遇到小数就用浮点数表示，有些情况下（如需要将一个整数变量乘以一个确定的小数常量），可先用一个确定的定点整数常量与整数变量相乘，然后再通过移位运算来确定小数点

2.3 浮点运算

二、浮点乘除运算

$$x = \mathrm{M}_x \cdot 2^{E_x} \quad y = \mathrm{M}_y \cdot 2^{E_y}$$

1. 乘法 $x \cdot y = (\mathrm{M}_x \cdot \mathrm{M}_y) \times 2^{E_x + E_y}$

2. 除法 $\dfrac{x}{y} = \dfrac{\mathrm{M}_x}{\mathrm{M}_y} \times 2^{E_x - E_y}$

3. 步骤

(1) 阶码采用定点加（乘法）减（除法）运算

(2) 尾数采用定点乘除运算

(3) 规格化

4. 浮点运算部件

阶码运算部件，尾数运算部件

2.3 浮点运算

浮点数运算：由多个ALU + 移位器实现

- 加/减运算
  - 对阶 、尾数相加减、规格化处理、舍入、判断溢出
- 乘/除运算
  - 尾数用定点乘/除运算实现，阶码用定点加/减运算实现
- 溢出判断
  - 当结果发生阶码上溢时，结果发生溢出；发生阶码下溢时，结果为0
- 精确表示运算结果
  - 中间结果增设保护位、舍入位，等
  - 最终结果舍入

2.3 浮点运算

三、浮点运算的实现

低档微机，通过子程序

中档微机，通过浮点处理器（协处理器）

高档微机，通过专门的浮点运算部件

思考题

1. 假定有4个整数用8位补码分别表示

r1=FEH，r2=F2H，r3=90H，r4=F8H，

若将运算结果存放在一个8位寄存器中，则下列运算会发生溢出的是(B)

A. r1 × r2　　B. r2 × r3

C. r1 × r4　　D. r2 × r4

思考题

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

A．X=0000007FH，y=FFF9H，z=00000076H

B．X=0000007FH，y=FFF9H，z=FFFF0076H

C．X=0000007FH，y=FFF7H，z=FFFF0076H

D．X=0000007FH，y=FFF7H，z=00000076H

思考题

3.浮点数加减运算过程一般包括对阶、尾数运算、规格化、舍入和判溢出 等步骤。设浮点数的阶码和尾数均采用补码表示，且位数分别为 5 位和 7 位（均 含 2 位符号位）。若有两个数 $X=2^7\times29/32$，$Y=2^5\times5/8$，则用浮点加法计算 X+Y 的最终结果是(D)

A．00111 1100010　　B.00111 0100010

C．01000 0010001　　D.发生溢出

解题思路：

- X= $2^{00111}\times0.11101$； Y = $2^{00101}\times0.101$ ；对阶后大的阶码为00111
- 位数相加后的结果为：01.00010，
- 尾数需右移规格化，同时阶码加1后变成 01 000

2.4 运算器部件及进位链结构

需解决的关键问题：

如何以加法器为基础，实现各种类型的算术逻辑运算处理。

解决思路：

复杂运算 → 四则运算 → 加法运算

解决方法：

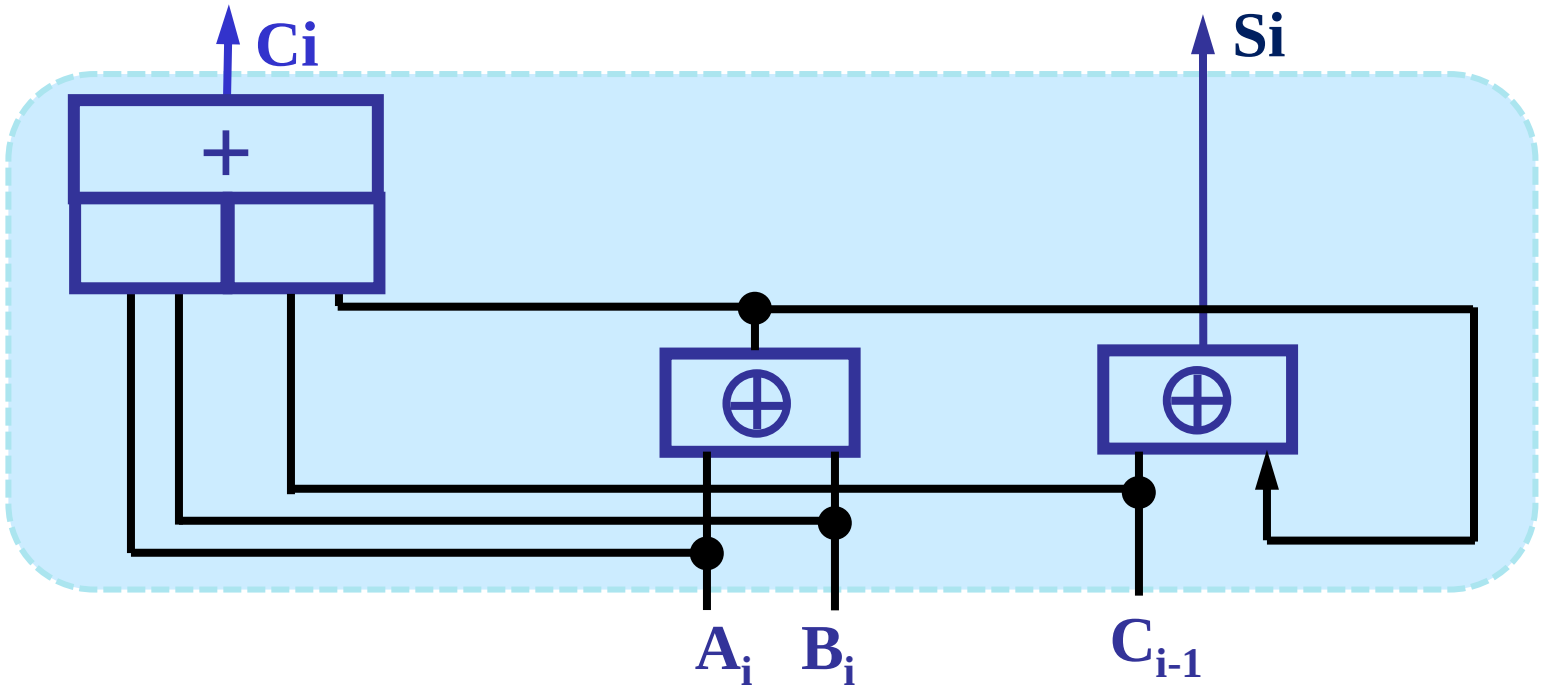
在加法器的基础上，增加移位传送功能，并且输入运算控制条件。

2.4.1 加法单元

全加器

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$

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



2.4.2 加法器与进位链逻辑

1. 进位信号的基本逻辑

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

设 $G_i = A_iB_i$ 为进位产生函数。

$P_i = A_i \oplus B_i$为进位传递函数

P_iC_{i-1}为传送进位。

$$C_i = G_i + P_i C_{i-1}$$

并行加法器的进位链

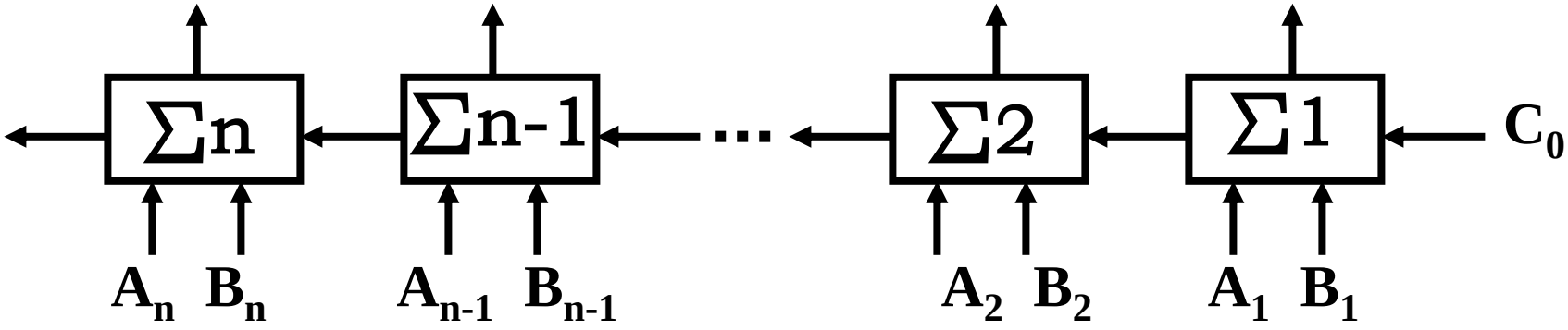
2. 串行进位 低位向高位依次传递进位信号

$$C_1=A_1B_1+(A_1\oplus B_1)C_0=G_1+P_1C_0$$

$$C_2=A_2B_2+(A_2\oplus B_2)C_1=G_2+P_2C_1$$

$$\vdots$$

$$C_n=A_nB_n+(A_n\oplus B_n)C_{n-1}=G_n+P_nC_{n-1}$$



影响运算速度的主要因素：进位信号的传递

并行加法器的进位链

3. 并行进位

$C_1 = G_1 + P_1C0$

$C_2 = G_2 + P_2C1$ （代换C1）

$= G_2 + P_2G_1 + P_2P_1C_0$

⋮

$C_n = G_n + P_nC_{n-1}$ （代换Cn-1）

$= G_n + P_nG_{n-1} + \ldots + P_nP_{n-1}\ldots P_2P_1C_0$

n + 1 项

[特点] 各位进位信号同时形成

并行加法器的进位链

4. 分组并行进位

主要思想：将n位全加器分成若干小组，组内各位之间并行进位，组间可以并行进位，也可以串行进位。

(1)组内并行，组间串行的进位链(单重分组跳跃进位)

以16位加法器为例

第一小组组内各位的进位逻辑表达式为：

$C_1=G_1+P_1C_0$

$C_2=G_2+P_2C_1=G_2+P_2G_2+P_2P_1C_0$

$C_3=G_3+P_3C_2=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1C_0$

$C_4=G_4+P_4C_3=G_4+P_4G_3+P_4P_3G_2+P_4P_3P_2G_1+P_4P_3P_2P_1C_0$

并行加法器的进位链

$$C_4=G_4+P_4C_3=\underbrace{G_4+P_4G_3+P_4P_3G_2+P_4P_3P_2G_1}_{G_I}+\underbrace{P_4P_3P_2P_1}_{P_I}C_0$$

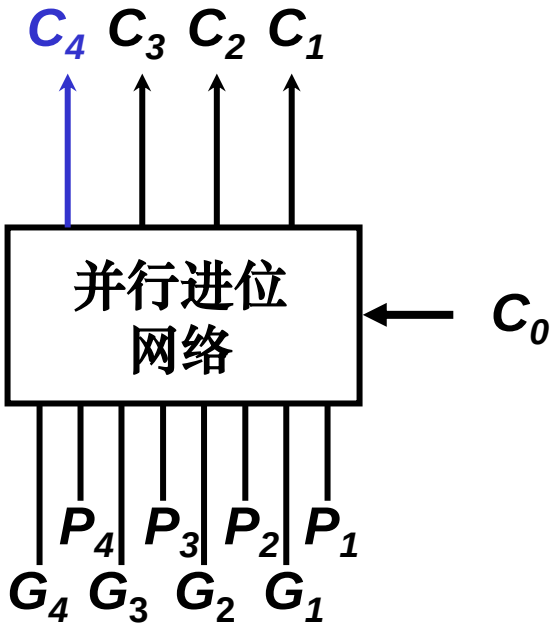
$$C_I=C_4=G_I+P_IC_0$$

同理，

$$C_{II}=C_8=G_{II}+P_{II}C_I$$

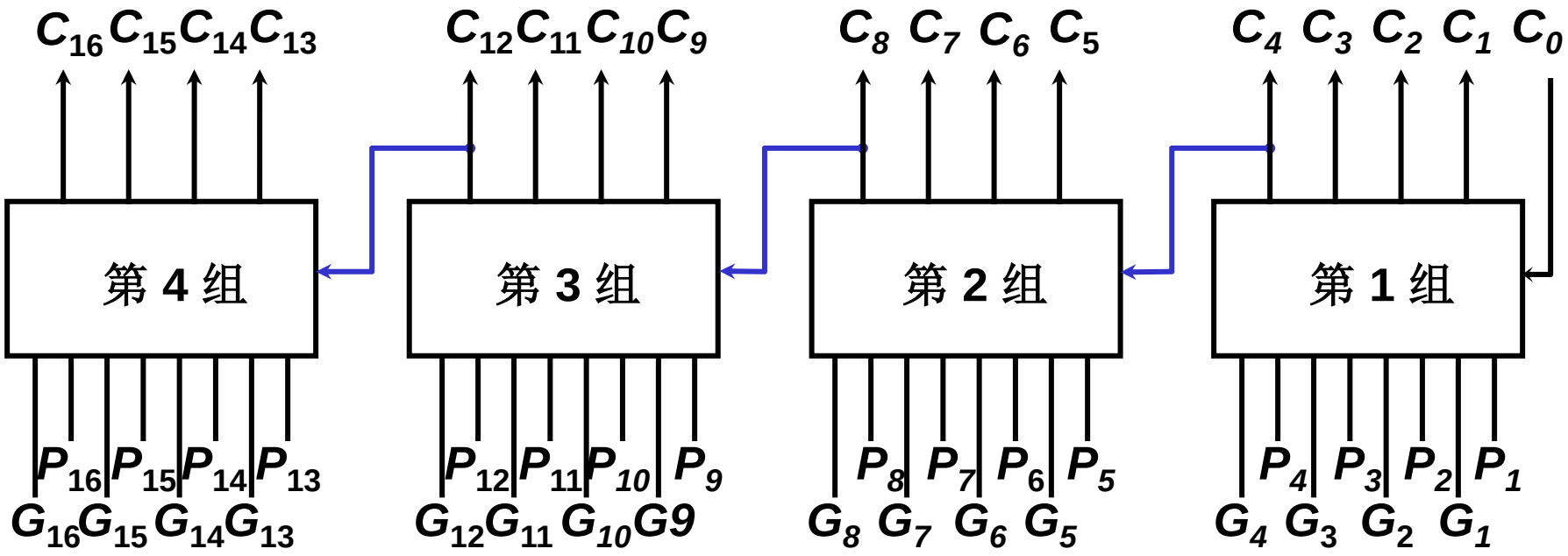
$$C_{III}=C_{12}=G_{III}+P_{III}C_{II}$$

$$C_{IV}=C_{16}=G_{IV}+P_{IV}C_{III}$$



并行加法器的进位链

16位组内并行组间串行进位加法器



并行加法器的进位链

(2) 组内并行，组间并行的进位链(多重分组跳跃进位)

$$C_I=C_4=G_I+P_IC_0$$

$$C_{II}=C_8=G_{II}+P_{II}C_I$$

$$= G_{II}+P_{II}G_I+P_{II}P_IC_0$$

$$C_{III}= C_{12}=G_{III}+P_{III}C_{II}$$

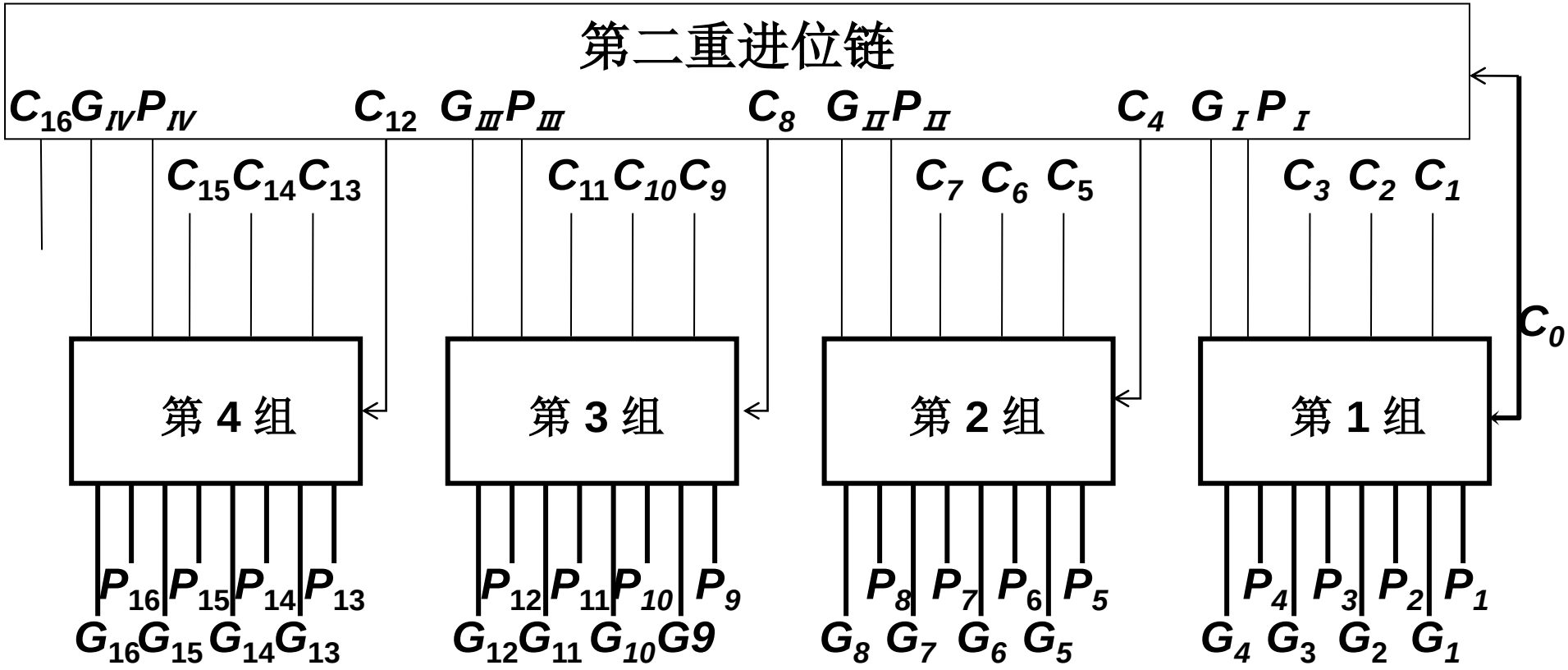
$$= G_{III}+P_{III}G_{II}+P_{III}P_{II}\ G_I+\ P_{III}P_{II}P_IC_0$$

$$C_{IV}= C_{16}=G_{IV}+P_{IV}C_{III}$$

$$= G_{IV}+P_{IV}G_{III}+\ P_{IV}P_{III}G_{II}+\ P_{IV}P_{III}P_{II}G_I+\ P_{IV}\ P_{III}P_{II}P_IC_0$$

并行加法器的进位链

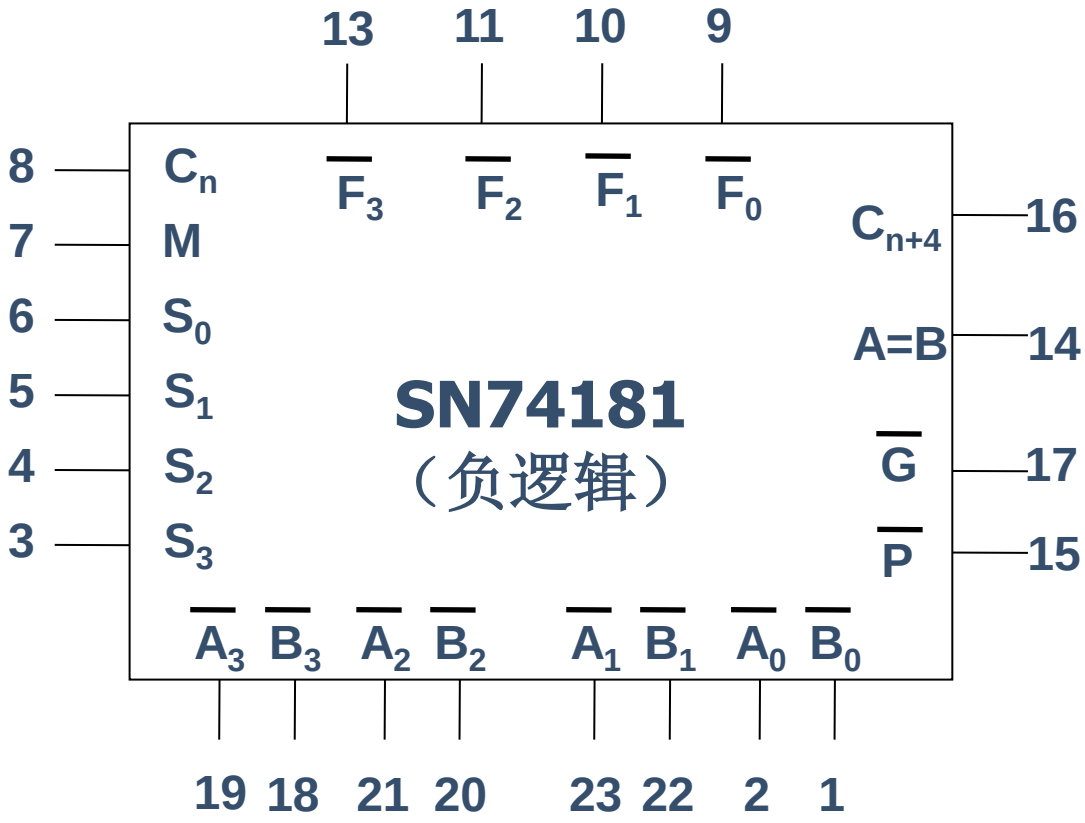
16位组内并行组间并行进位加法器



2.4.3 多位ALU部件

一、4位ALU芯片

SN74181芯片



2.4.3 多位ALU部件

74181的算术逻辑运算功能表

工作方式选择 $S_3S_2S_1S_0$	负逻辑			正逻辑		
	逻辑运算 (M=1)	算术运算 (M=0) C_n=0 无进位	算术运算 (M=0) C_n=1 有进位	逻辑运算 (M=1)	算术运算 (M=0) C_n=0 无进位	算术运算 (M=0) C_n=1 有进位
0000	$\overline{A}$	A减1	A	$\overline{A}$	A	A加1
0001	$\overline{AB}$	AB减1	AB	$\overline{A+B}$	A+B	(A+B) 加1
0010	$\overline{A}+B$	$\overline{AB}$减1	$A\overline{B}$	$\overline{A}B$	$A+\overline{B}$	(A+$\overline{B}$) 加1
0011	1	减1	0	0	减1	0
0100	$\overline{A+B}$	A加 (A+$\overline{B}$)	A加 (A+$\overline{B}$) 加1	$\overline{AB}$	A加$A\overline{B}$	A加$A\overline{B}$加1
0101	$\overline{B}$	AB加 (A+$\overline{B}$)	AB加 (A+$\overline{B}$) 加1	$\overline{B}$	(A+B) 加$A\overline{B}$	(A+B) 加$A\overline{B}$加1
0110	$\overline{A\oplus B}$	A减B减1	A减B	$A\oplus B$	A减B加1	A减B

2.4.3 多位ALU部件

工作方式选择 $S_3S_2S_1S_0$	负逻辑			正逻辑		
	逻辑运算 (M=1)	算术运算 (M=0) C_n=0 无进位	算术运算 (M=0) C_n=1 有进位	逻辑运算 (M=1)	算术运算 (M=0) C_n=0 无进位	算术运算 (M=0) C_n=1 有进位
0111	$\overline{A+B}$	$\overline{A+B}$	$\overline{(A+B)}$ 加1	$\overline{AB}$	$\overline{AB}$减1	$\overline{AB}$
1000	$\overline{AB}$	A加 (A+B)	A加 (A+B) 加1	$\overline{A}$+B	A加AB	A加AB加1
1001	A⊕B	A加B	A加B加1	$\overline{A\oplus B}$	A加B	A加B加1
1010	B	$\overline{AB}$加 (A+B)	$\overline{AB}$加 (A+B) 加1	B	(A+$\overline{B}$) 加AB	(A+$\overline{B}$) 加AB加1
1011	A+B	A+B	(A+B) 加1	AB	AB减1	AB
1100	0	A加A*	A加A加1	1	A加A*	A加A加1
1101	A $\overline{B}$	AB加A	AB加A加1	$\overline{A+B}$	(A+B) 加A	(A+B) 加A加1
1110	AB	$\overline{AB}$加A	$\overline{AB}$加A加1	A+B	(A+$\overline{B}$) 加A	(A+$\overline{B}$) 加A加1
1111	A	A	A加1	A	A减1	A

*：A加A=2A，算术左移一位

2.4.3 多位ALU部件

二、多位ALU部件的设计

74181ALU为4位并行加法器，

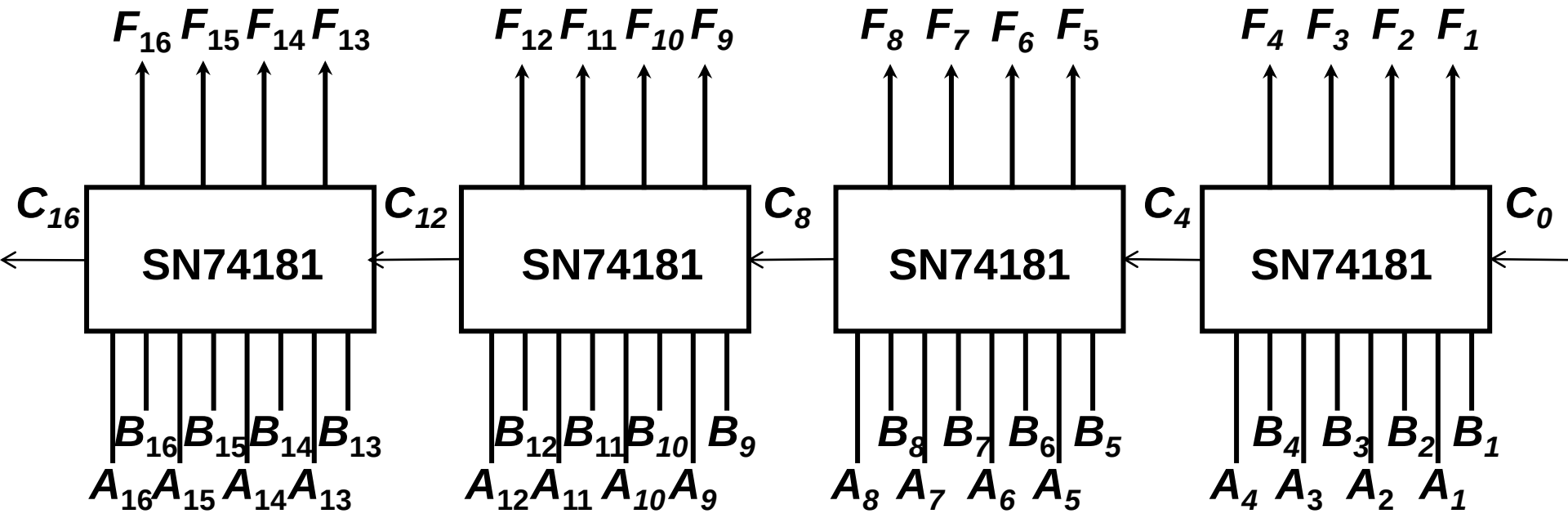
组成16位的并行加法器——怎么办?

4片(组)74181连接 ——怎样连?

- **组与组之间串行连接**
- **组与组之间并行连接**

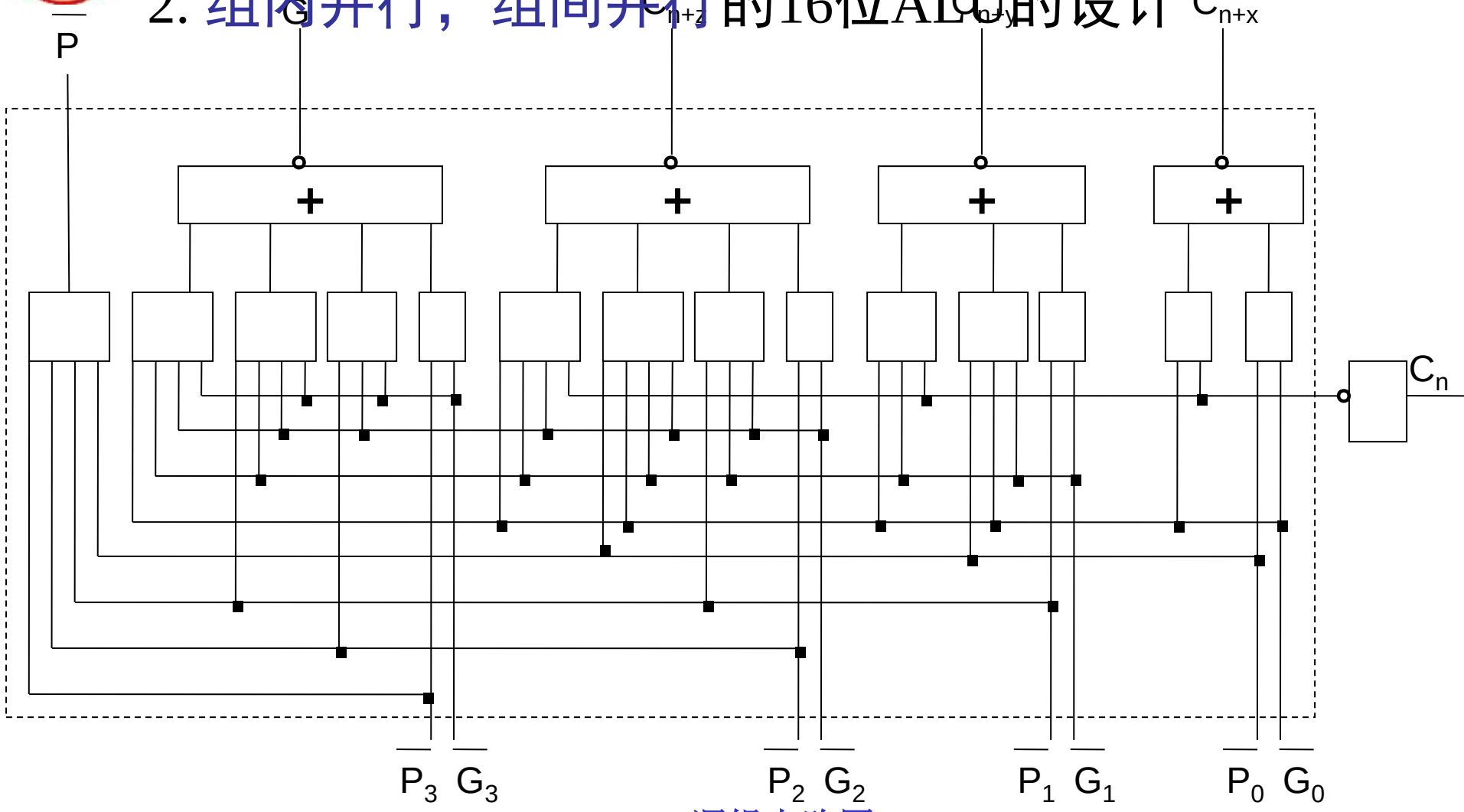
2.4.3 多位ALU部件

1. 组间串行，组内并行进位的16位ALU的设计



2.4.3 多位ALU部件

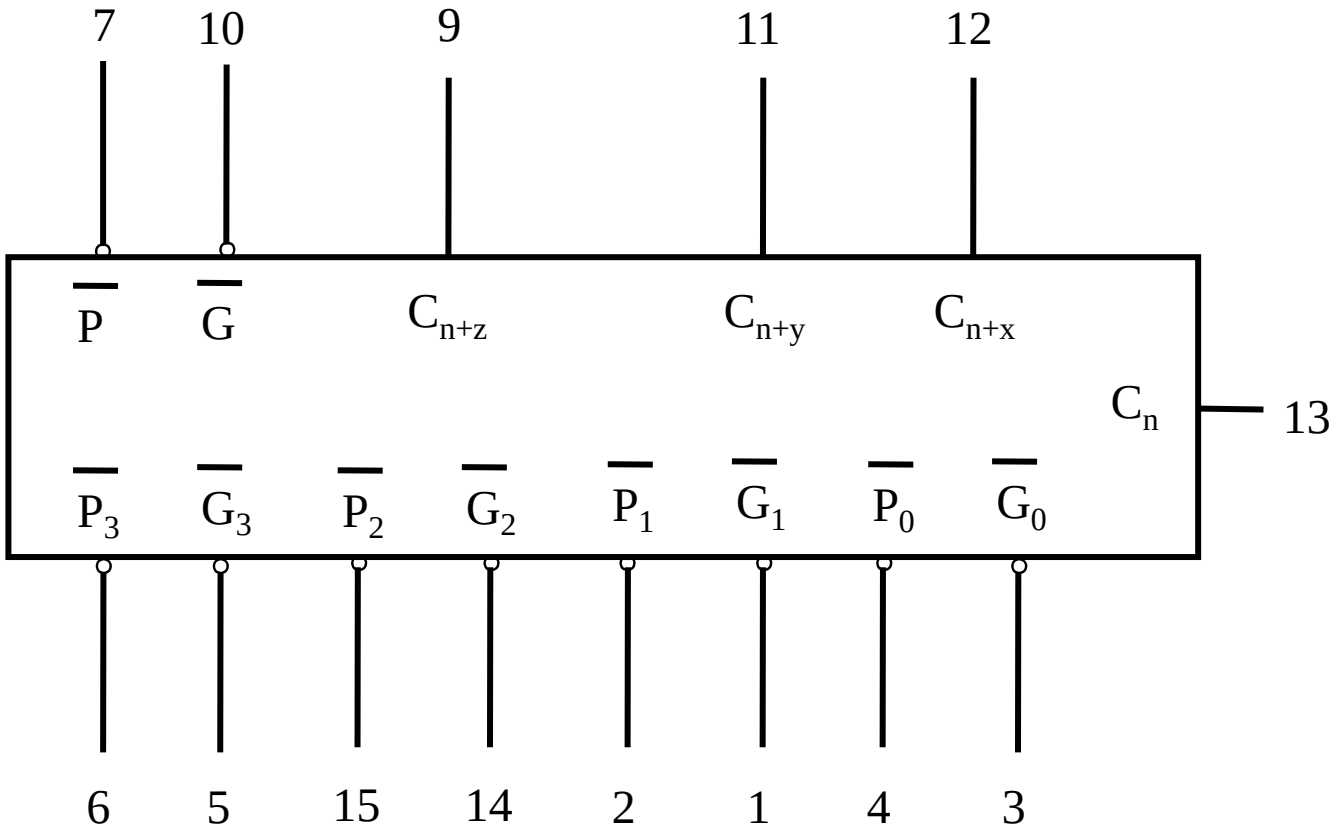
2. 组内并行，组间并行的16位ALU的设计



SN74182逻辑电路图

2.4.3 多位ALU部件

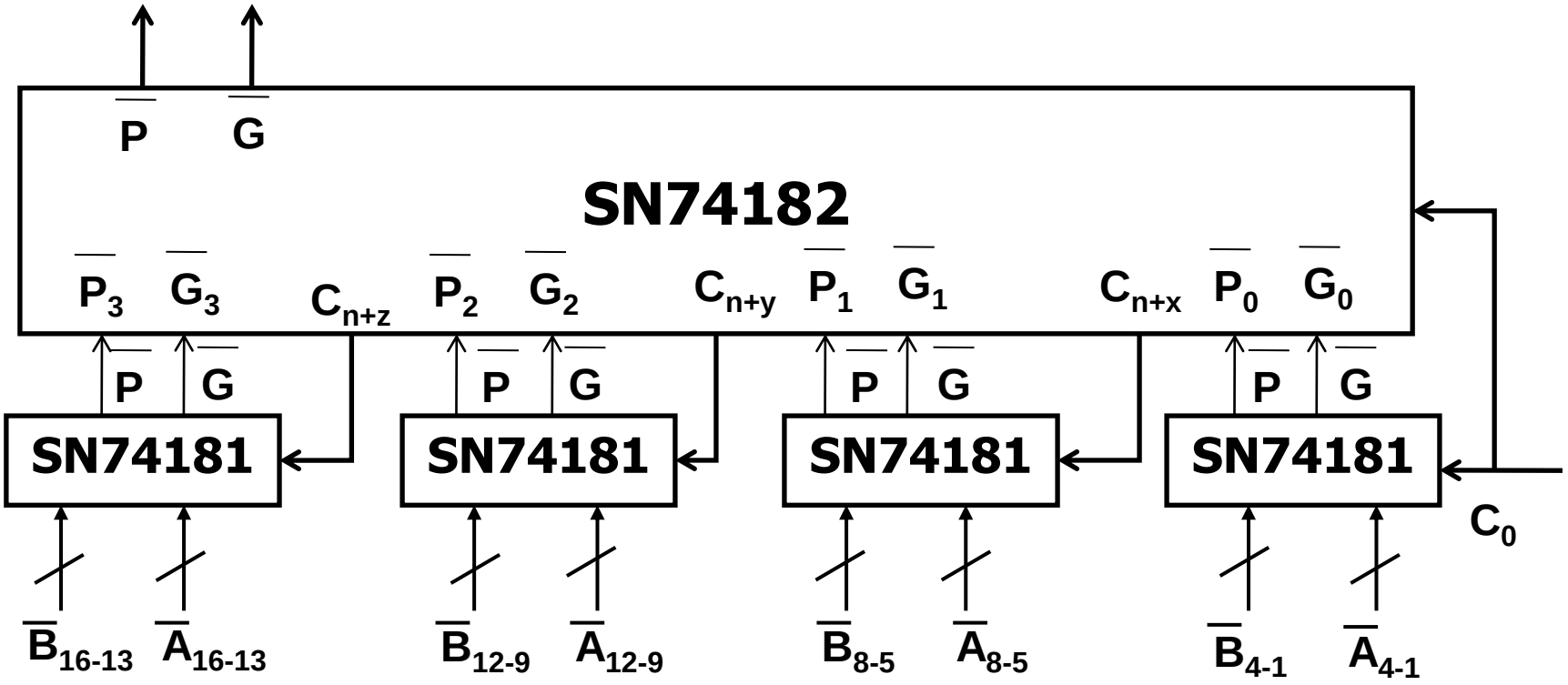
并行进位部件SN74182的芯片



2.4.3 多位ALU部件

16位ALU(组内并行，组间并行)

4片SN74181和1片SN74182



2.4.3 多位ALU部件

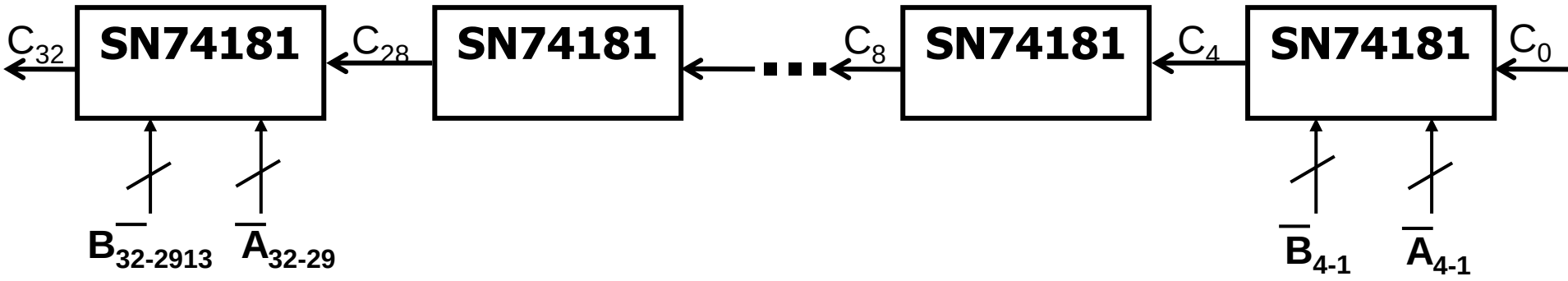
[例1]用SN74181和SN74182设计32位ALU

（1）行波进位方式

（2）两重进位方式

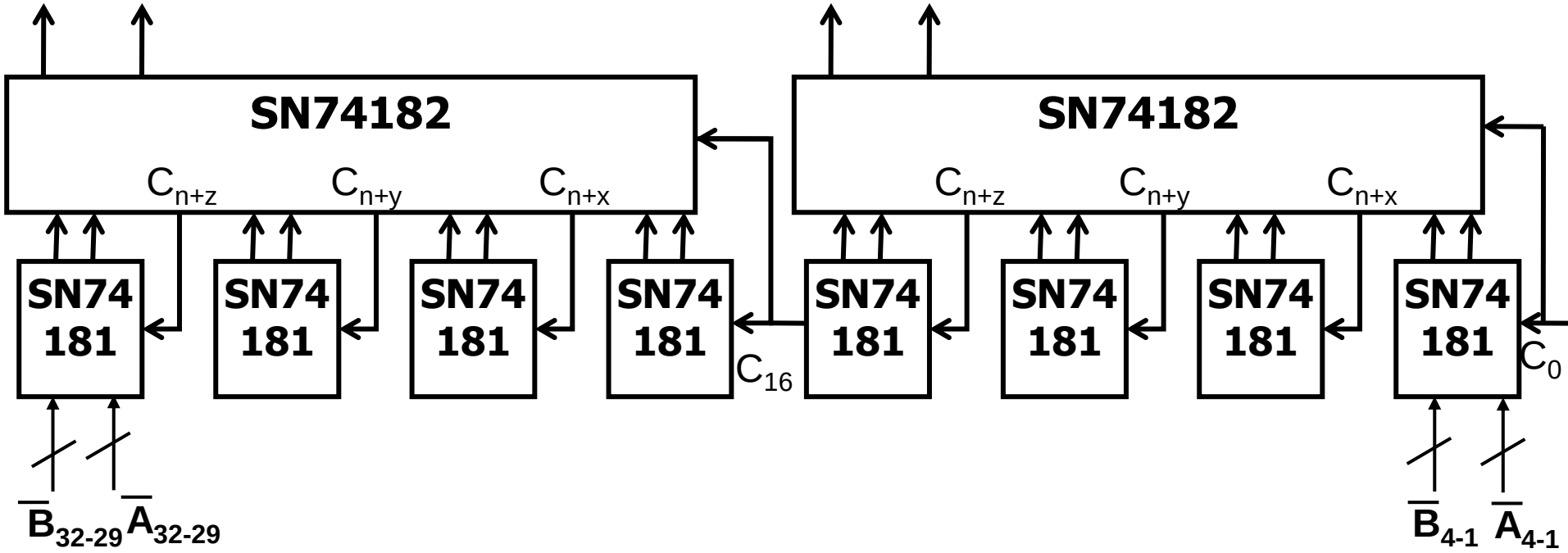
（3）三重进位方式

解：（1）行波进位方式的32位ALU



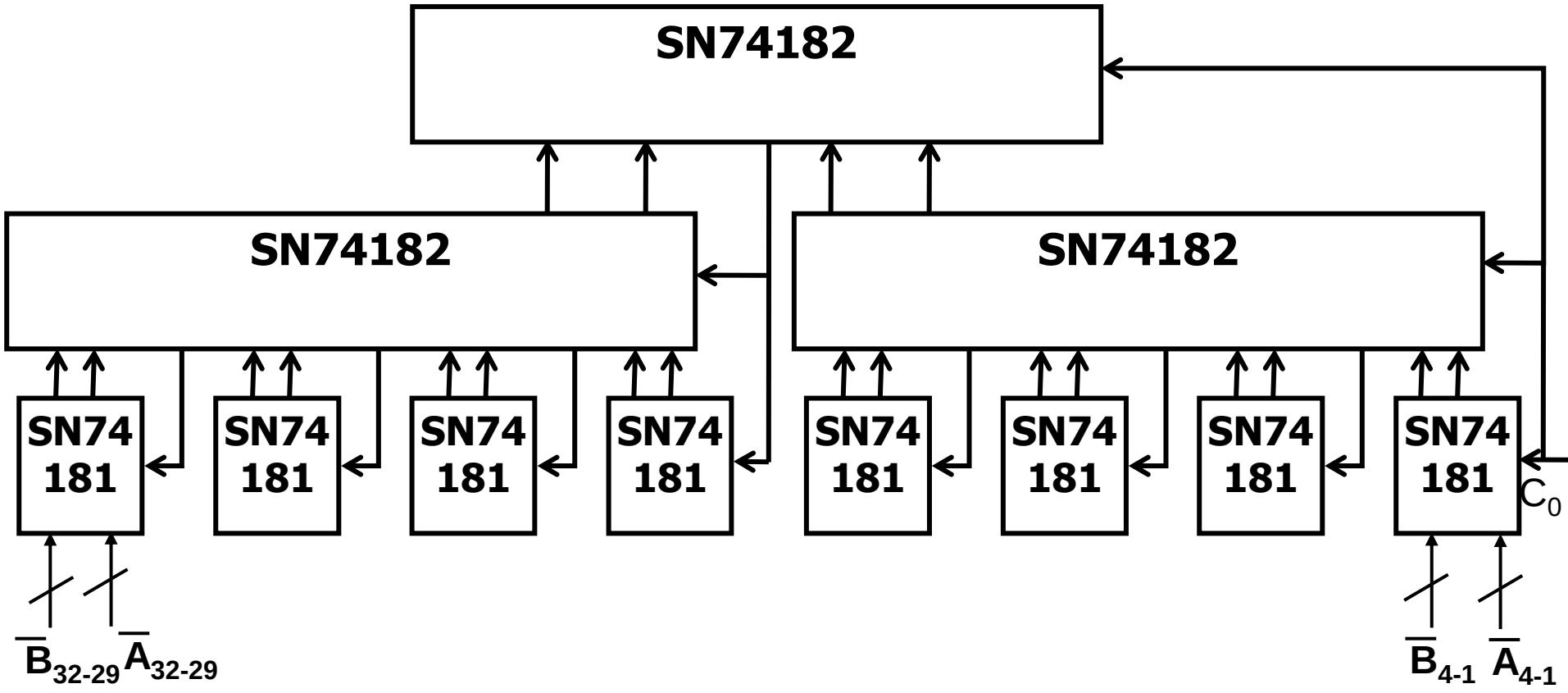
2.4.3 多位ALU部件

（2）两重进位方式的32位ALU



2.4.3 多位ALU部件

（3）三重进位方式的32位ALU



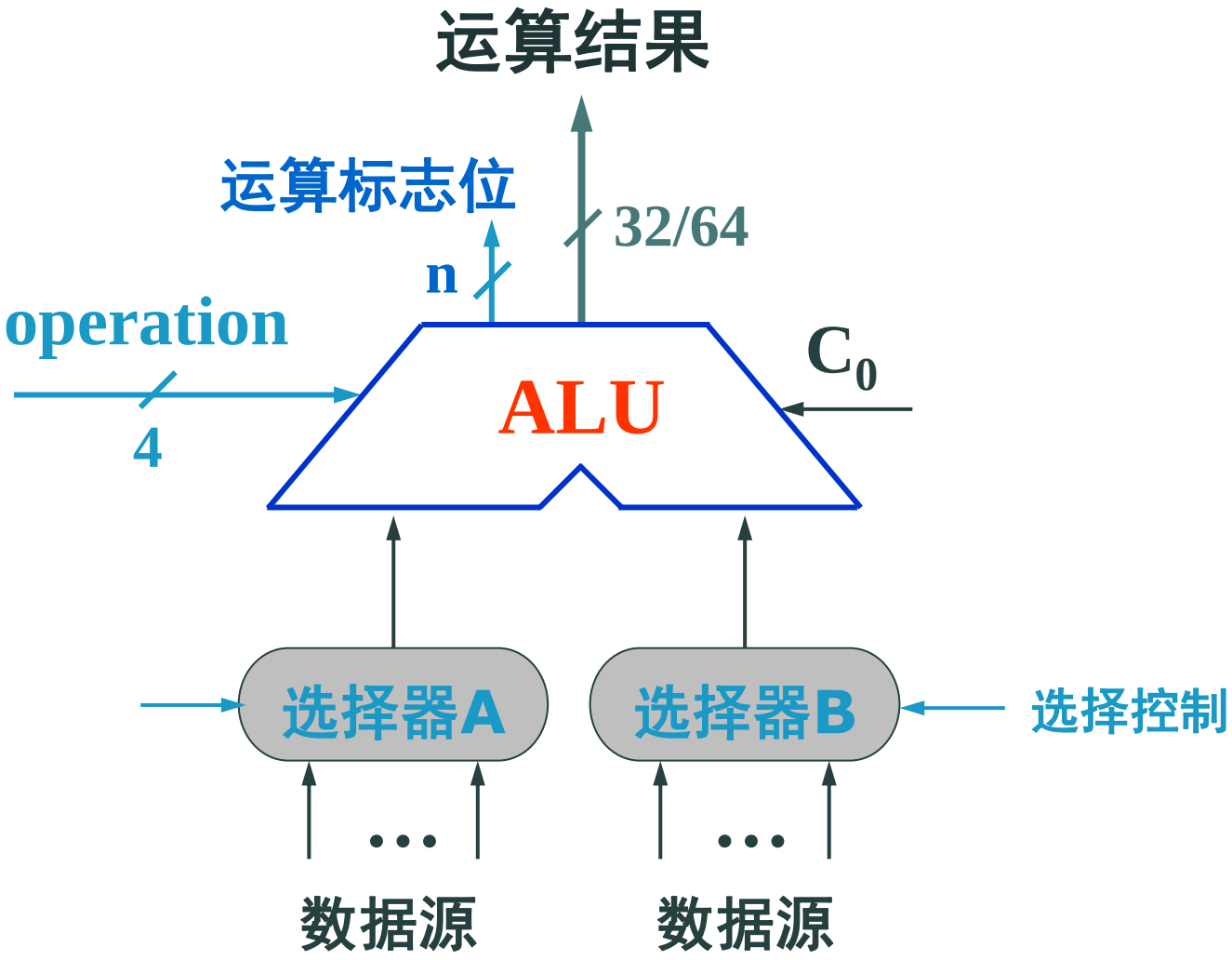
2.4.4 定点运算器

- 基本组成包括：
  - **算术逻辑运算单元ALU：核心部件**
  - **暂存器：用来存放参与计算的数据及运算结果，它只对硬件设计者可见，即只被控制器硬件逻辑控制或微程序所访问**
  - **通用寄存器堆：用于存放程序中用到的数据，它可以被软件设计者所访问。**
  - **内部总线：用于连接各个部件的信息通道。**
  - **标志寄存器：用来保存ALU操作结果的某些状态**
- 设计定点运算器，如何确定各部件的功能和组织方式是关键，这取决于以下几个方面：
  - **指令系统**
  - **机器字长**
  - **机器数及其运算原理**
  - **体系结构**

2.4.4 定点运算器

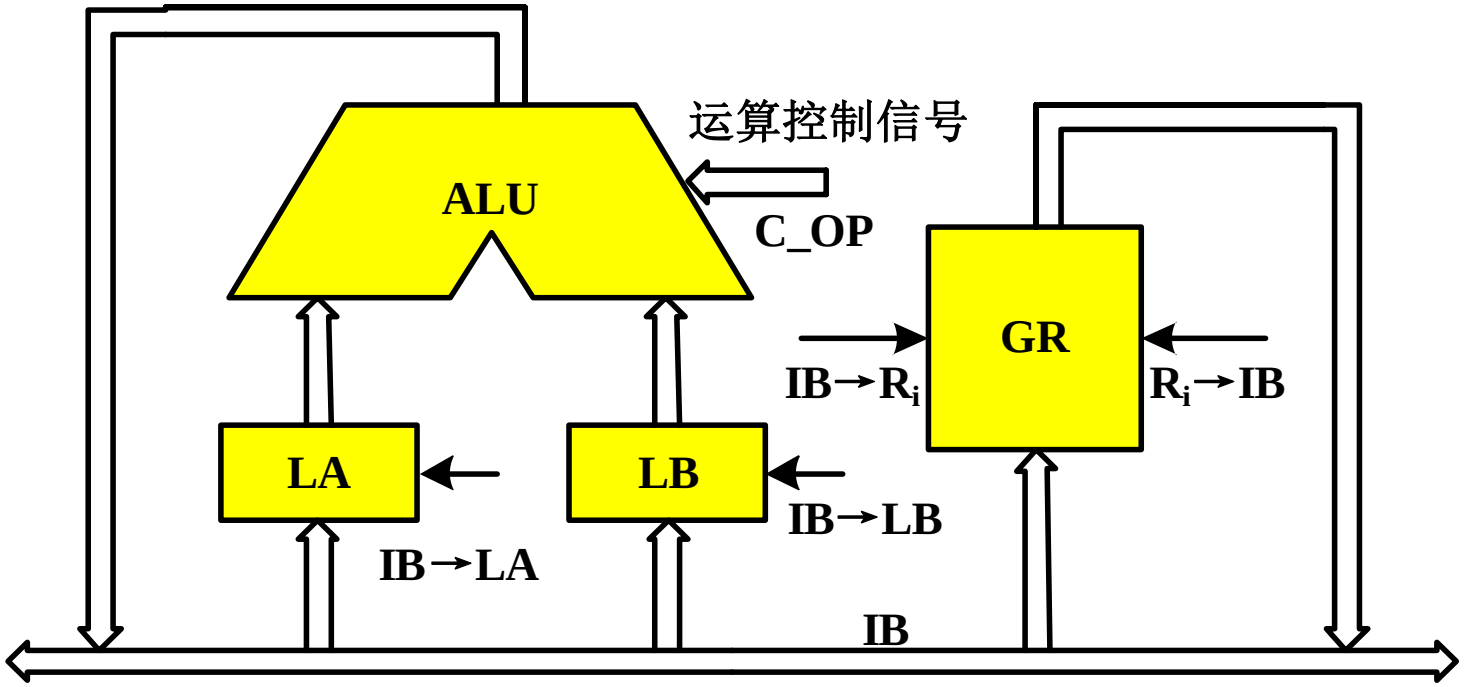
- 标志寄存器用来保存ALU操作结果的某些状态，这种状态可作为外界对操作结果进行分析的一个依据，也可以用于判断程序是否要转移的条件，该寄存器通常也称为状态寄存器。
- 依据功能上的差别，不同的CPU，其标志寄存器中包含的标志也不尽相同。
- 一般标志寄存器中包含了最基本的5种运算结果标志：
  - ZF 结果为零标志（zero flag bit）
  - CF 进位/借位标志位（carry flag bit）
  - OF 溢出标志（overflow flag bit）
  - SF 符号标志（sign flag bit)
  - PF 奇偶标志（parity flag bit)

2.4.4 定点运算器



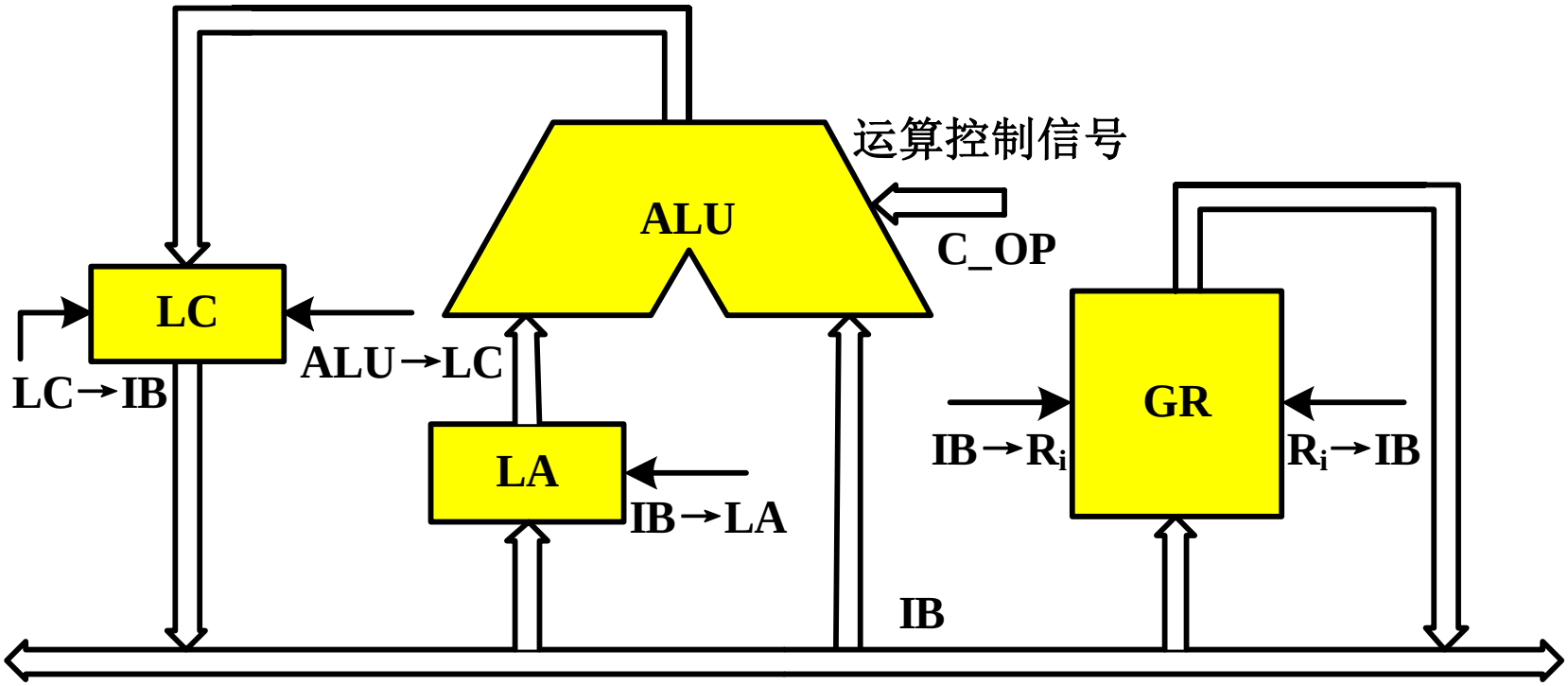
定点运算器的总线结构

- 1、单总线结构
  - 单总线运算器的结构形式1



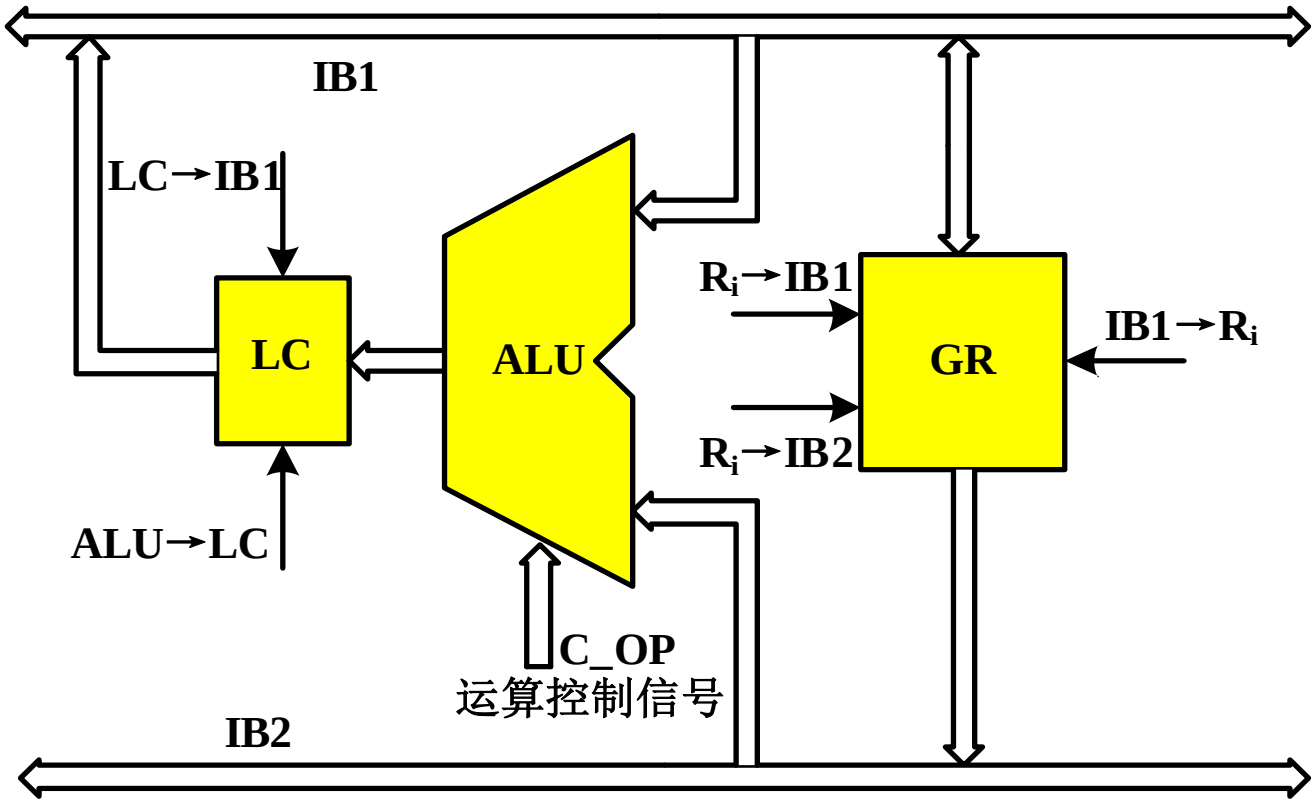
定点运算器的总线结构

- 单总线运算器的结构形式2



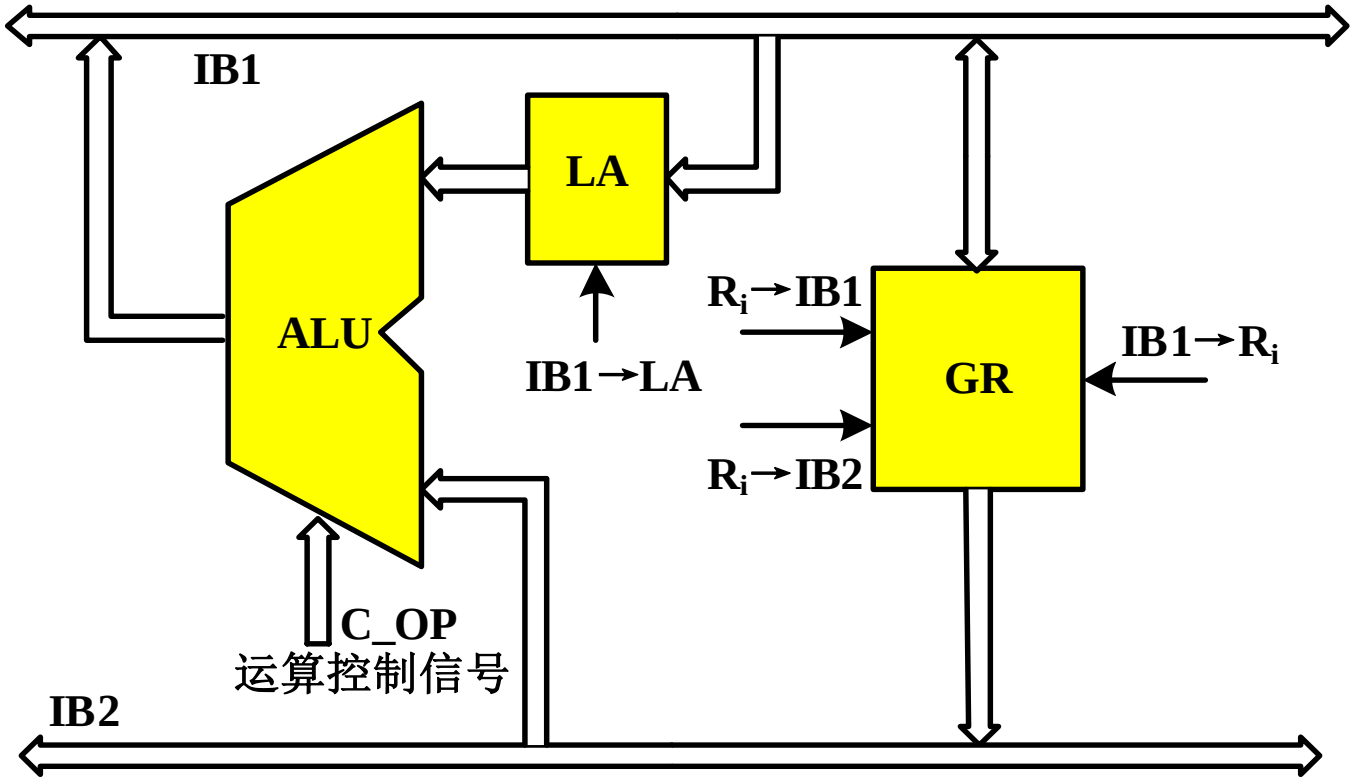
定点运算器的总线结构

- 2、双总线结构
  - 双总线运算器的结构形式1



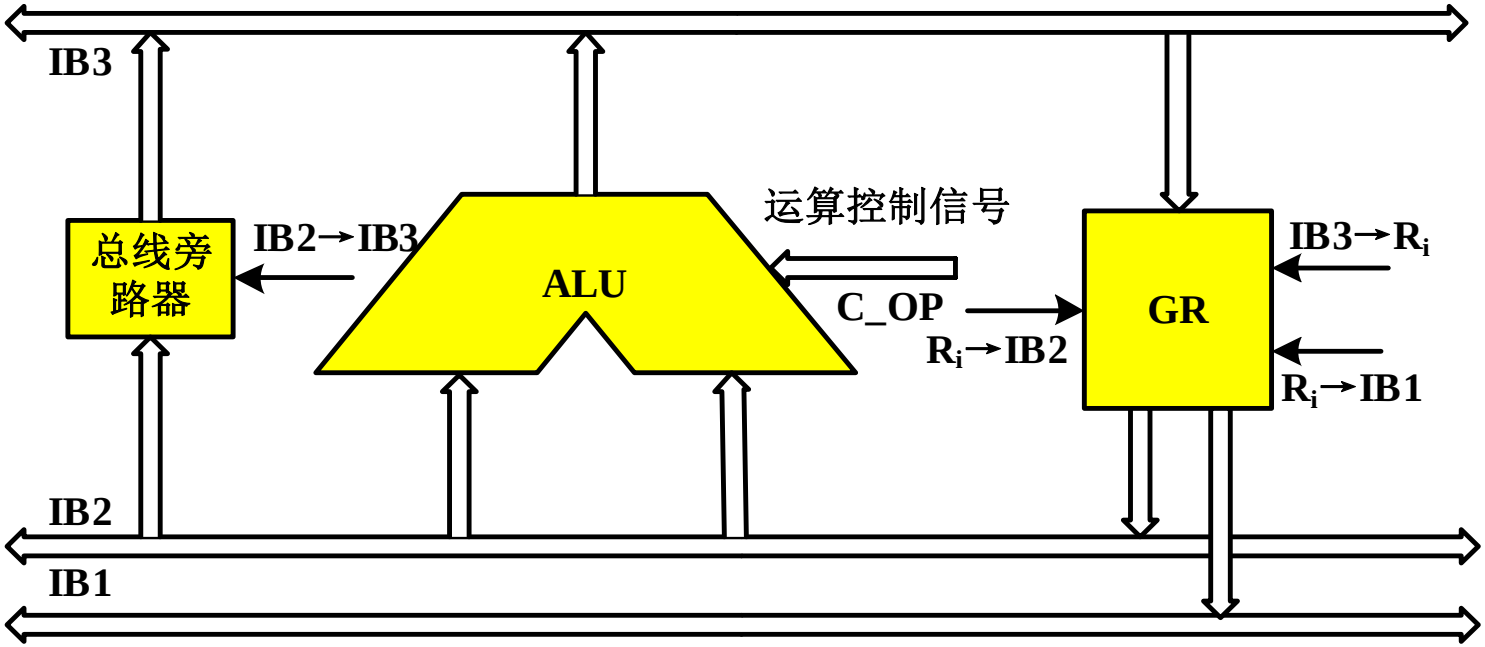
定点运算器的总线结构

- 双总线运算器的结构形式2



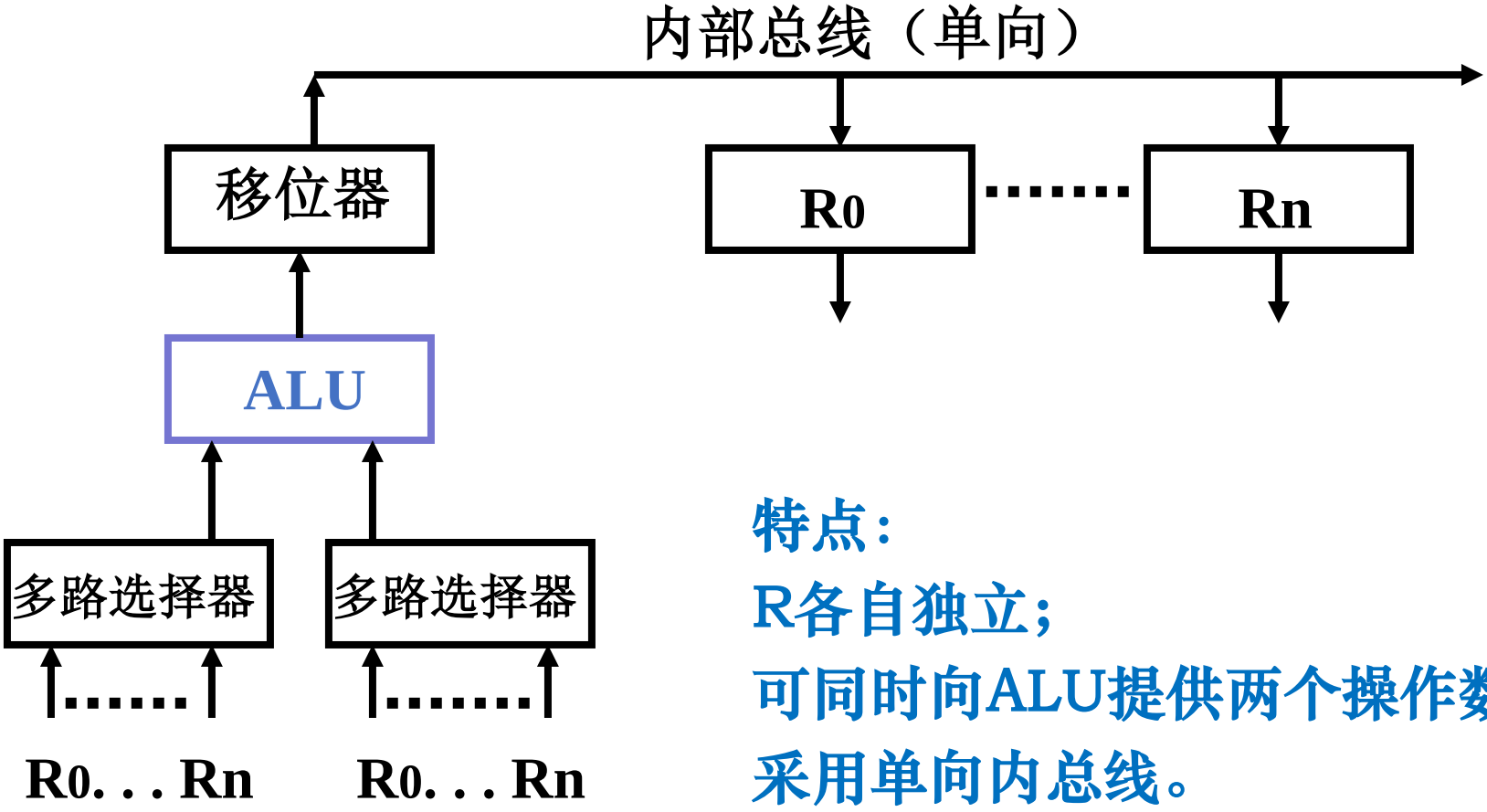
定点运算器的总线结构

- 3、三总线结构



基本的原则：在一个CPU周期（一步）内，**某条总线上的数据必须是唯一的**，且不能保留（至下一个CPU周期）。

带多路选择器的运算器



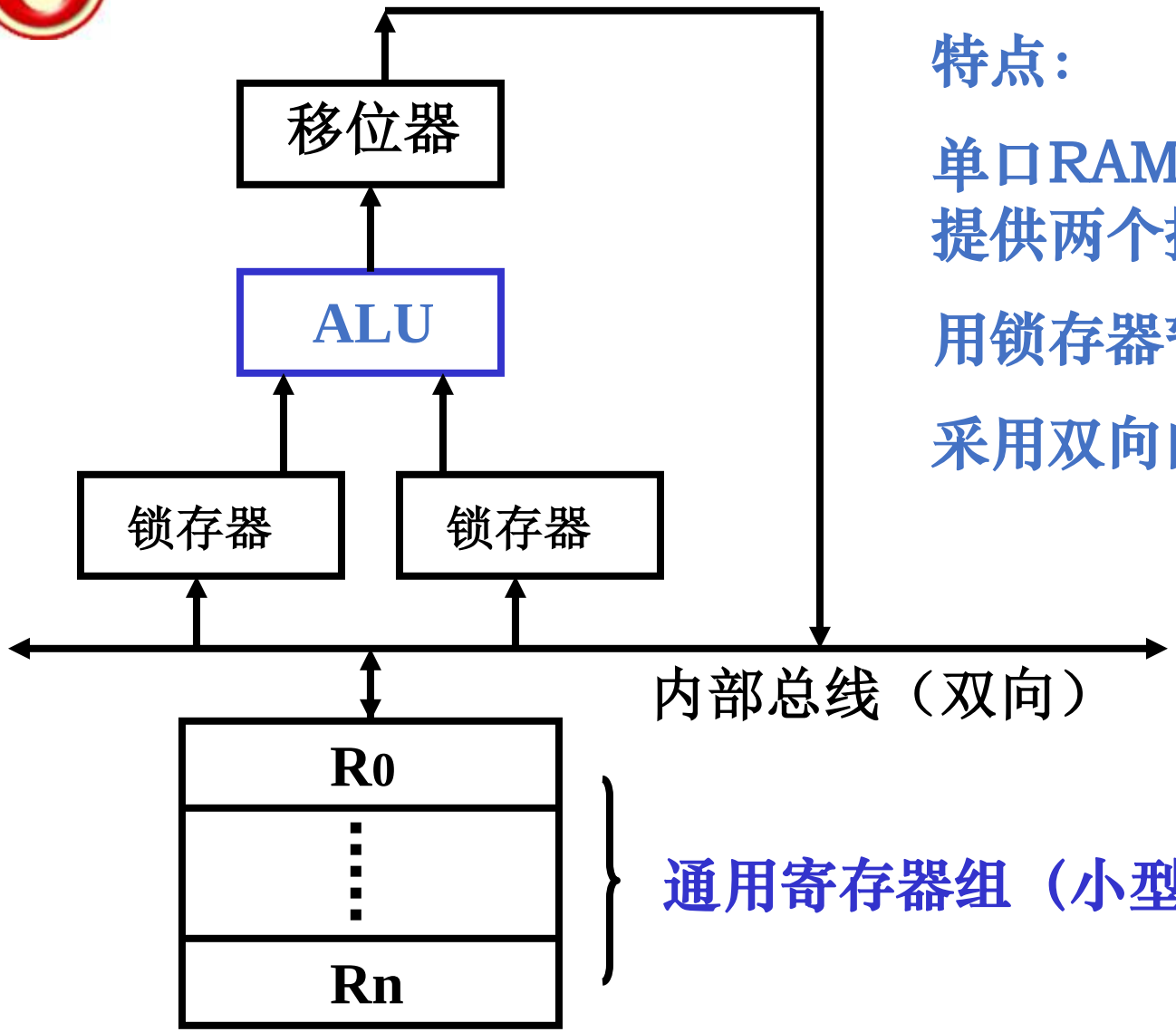
特点：

R各自独立；

可同时向ALU提供两个操作数；

采用单向内总线。

带输入锁存器的运算器



特点：

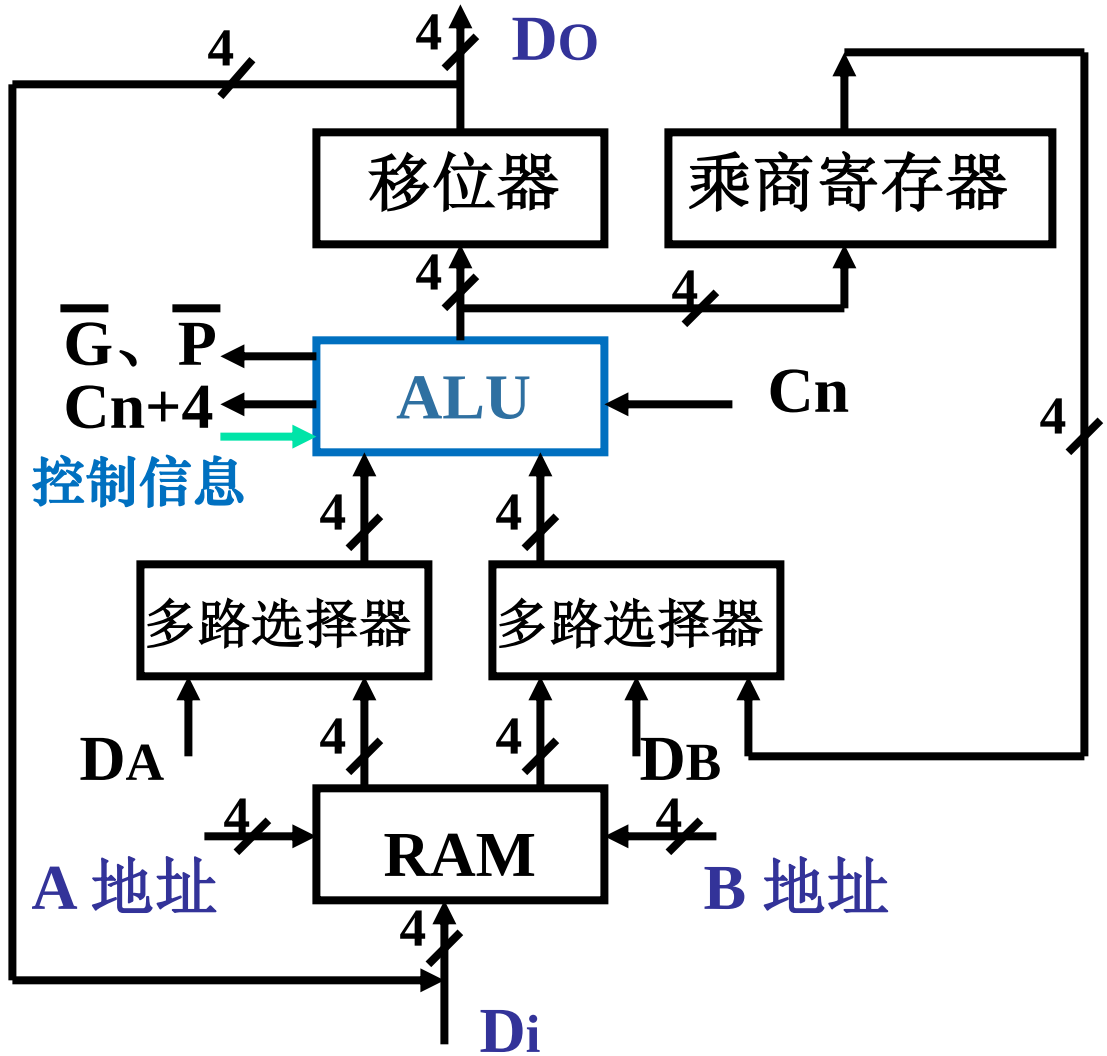
单口RAM不能同时向ALU提供两个操作数；

用锁存器暂存操作数；

采用双向内总线。

位片式运算器

4位片运算器粗框



特点：

用双口RAM（两地址端、两数据端）作通用寄存器组，可同时提供数据；

用多路选择器作输入逻辑，不需暂存操作数；

ALU增加乘、除功能，用乘商寄存器存放乘数、乘积或商。