

# 运算方法和运算器

- 数据与文字的表示方法
- 定点加减运算
- 定点乘除运算
- 定点运算器的组成
- 浮点运算方法和浮点运算器

# 真值和机器数

● 真值:

$$(+5)_{10} = (+10101)_2$$

$$(-7)_{10} = (-111)_2$$

符号 (+/-) + 尾数 (绝对值)

用最高位做符号位

● 机器数:

S

↑  
符号位

● 数的符号在计算机中的表示:

原码、反码、补码: 0表示正数, 1表示负数

移码: 0表示负数, 1表示正数

## 1.原码表示法

思考：原码表示绝对值最大（小）的正（负）数分别是什么？

若**定点小数**的真值为 $X = \pm 0.x_1x_2 \dots x_n$ ，则原码表示的定义是：

$$[X]_{\text{原}} = \begin{cases} X & 1 > X \geq 0 \\ 1 - X = 1 + |X| & 0 \geq X > -1 \end{cases}$$

例如， $x = +0.1001$ ，则 $[x]_{\text{原}} = 0.1001$

$x = -0.1001$ ，则 $[x]_{\text{原}} = 1.1001$

对于0，原码有“+0”、“-0”之分，故有两种形式：

$$[+0]_{\text{原}} = 0.000\dots 0 \quad [-0]_{\text{原}} = 1.000\dots 0$$

若**定点整数**的真值为 $X = \pm x_1x_2 \dots x_n$ ，则原码表示的定义是：

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

采用原码表示法简单易懂，但它的缺点是运算复杂。

当两数相加时，如果是同号则数值相加；如果是异号，则要进行减法。而在进行减法时还要比较绝对值的大小，然后大数减去小数，最后还要给结果选择符号。

## 2.反码表示法

所谓反码，就是二进制的各位数码0变为1，1变为0。在一些文献中，这种以2为基数的反码又称为1的补码。

对于正数 $X = +0.x_1x_2 \dots x_n$ ，则反码表示为： $[X]_{\text{反}} = 0.x_1x_2 \dots x_n$

对于负数 $X = -0.x_1x_2 \dots x_n$ ，则反码表示为： $[X]_{\text{反}} = 1.\bar{x}_1\bar{x}_2 \dots \bar{x}_n$

例如， $x = +0.1001$ ，则 $[x]_{\text{反}} = 0.1001$

$x = -0.1001$ ，则 $[x]_{\text{反}} = 1.0110$

若定点小数的真值为 $X = \pm 0.x_1x_2 \dots x_n$ ，则反码表示的定义是：

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

对于0，有 $[+0]_{\text{反}}$ 和 $[-0]_{\text{反}}$ 之分：

$$[+0]_{\text{反}} = 0.000\dots 0$$

$$[-0]_{\text{反}} = 1.111\dots 1$$

若定点整数的真值为 $X = \pm x_1x_2 \dots x_n$ ，则反码表示的定义是：

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

### 3.补码表示法

思考：补码表示绝对值最大（小）的正（负）数分别是什么？

若**定点小数**的真值为 $X = \pm 0.x_1x_2 \dots x_n$ ，则补码表示的定义是：

$$[X]_{\text{补}} = \begin{cases} X & 1 > X \geq 0 \\ 2 + X = 2 - |X| & 0 \geq X \geq -1 \end{cases} \pmod{2}$$

例如， $x = +0.1001$ ，则 $[x]_{\text{补}} = \underline{0.1001}$

$x = -0.1001$ ，则 $[x]_{\text{补}} = 1.0111$

$[+0]_{\text{补}} = [-0]_{\text{补}} = 0.000\dots 0$

$[-1]_{\text{补}} = \underline{1.000\dots 0}$

若**定点整数**的真值为 $X = \pm x_1x_2 \dots x_n$ ，则补码表示的定义是：

$$[X]_{\text{补}} = \begin{cases} X & 2^n > X \geq 0 \\ 2^{n+1} + X = 2^{n+1} - |X| & 0 \geq X \geq -2^n \end{cases} \pmod{2^{n+1}}$$

对于负数，补码等于反码的最低位加1：

$[-2^n]_{\text{补}} = 1000\dots 0$

$[X]_{\text{补}} = [X]_{\text{反}} + 2^{-n}$   $X$  为小数

$[X]_{\text{补}} = [X]_{\text{反}} + 1$   $X$  为整数

#### 4.移码表示法

若**定点整数**的真值为 $X = \pm x_1 x_2 \dots x_n$ ，则移码表示的定义是：

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

例如， $x = +1001$ ，则 $[x]_{\text{移}} = 11001$

$x = -1001$ ，则 $[x]_{\text{移}} = 00111$

四种机器码的比较（以 $n+1$ 位定点整数为例）：

1. 原码、反码和补码的符号位为0正1负，移码为1正0负；
2. 原码和反码的表示范围为 $2^n > x > -2^n$ ，0的原码和反码都有两个；补码和移码的表示范围为 $2^n > x \geq -2^n$ ，0的补码和移码都只有一个；
3. 正数的原码、反码和补码相同，负数则不同；
4. 移码等于补码的符号位取反；移码的大小关系与真值的大小关系相同。

真值	原码	反码	补码	移码
+7	0111	0111	0111	1111
+6	0110	0110	0110	1110
+5	0101	0101	0101	1101
+4	0100	0100	0100	1100
+3	0011	0011	0011	1011
+2	0010	0010	0010	1010
+1	0001	0001	0001	1001
+0	0000	0000	0000	1000
-0	1000	1111	—	—
-1	1001	1110	1111	0111
-2	1010	1101	1110	0110
-3	1011	1100	1101	0101
-4	1100	1011	1100	0100
-5	1101	1010	1011	0011
-6	1110	1001	1010	0010
-7	1111	1000	1001	0001
-8	—	—	1000	0000

# 数据格式

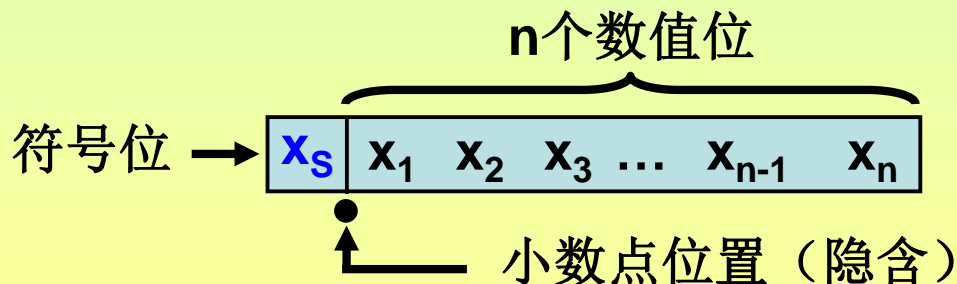
计算机中常用的数据表示格式有两种，一是定点格式，二是浮点格式。定点格式容许的数值范围有限，但要求的处理硬件比较简单；浮点格式容许的数值范围很大，但要求的处理硬件比较复杂。

## 1. 定点数的表示方法

定点数是指小数点的位置固定不变。小数点的位置是事先约定的，不需要占用一个二进制位来表示。通常将数据表示成纯小数或纯整数。

### 定点小数

小数点位于符号位和最高有效位之间。

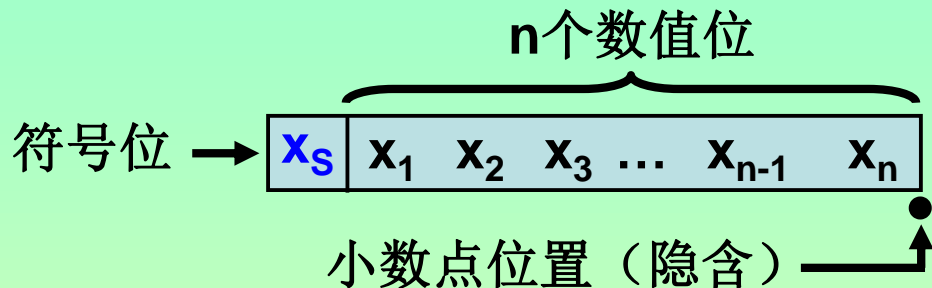


采用原码表示时，所能表示的数的范围是： $0 \leq |X| \leq 1 - 2^{-n}$ 。



## 定点整数

小数点位于最高有效位之后。



采用原码表示时，所能表示的数的范围是： $0 \leq |X| \leq 2^n - 1$ 。

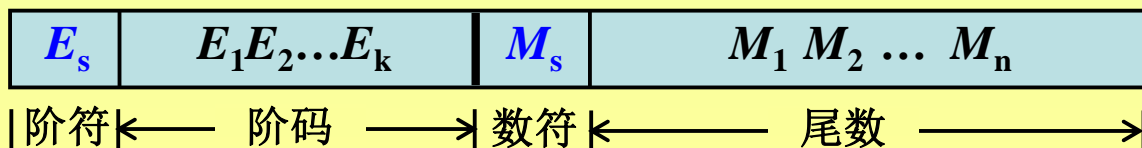
## 2.浮点数的表示方法

一个 $R$ 进制的数 $N$ 可以写成

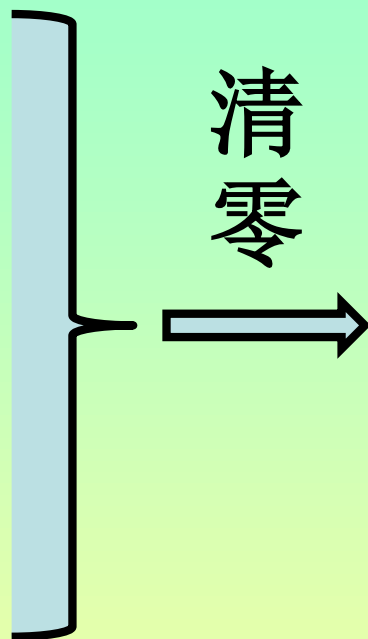
$$N = M \times R^E$$

其中 $M$ 为尾数， $E$ 为指数， $R$ 为基数。在计算机中通常取 $R$ 为2、8或16。

在机器中表示一个浮点数时，一要给出尾数，用定点小数的原码或补码表示；二要给出指数，用整数的补码或移码表示，称为阶码。



$0.0 \times 10^{99}$   
 $0.0 \times 10^{-9}$   
 $-9.9 \times 10^{-100}$   
 $7.8 \times 10^{-100}$



机器零  
 $0.0 \times 10^0$

为了使浮点数的表示形式唯一，当尾数 $M$ 不为0时，若尾数用原码表示，规定  $\frac{1}{2} \leq |M| < 1$ ，即尾数的二进制形式为 $0.1 \times \times \dots \times$ 或 $1.1 \times \times \dots \times$ ；若尾数用补码表示，规定 $\frac{1}{2} \leq M < 1$ 或 $-1 \leq M < -\frac{1}{2}$ ，即尾数的二进制形式为 $0.1 \times \times \dots \times$ 或 $1.0 \times \times \dots \times$ 。这称为浮点数的规格化表示。 $M = -\frac{1}{2}$ 对于原码表示是规格化的，对于补码表示则不是。      **P22的规格化**

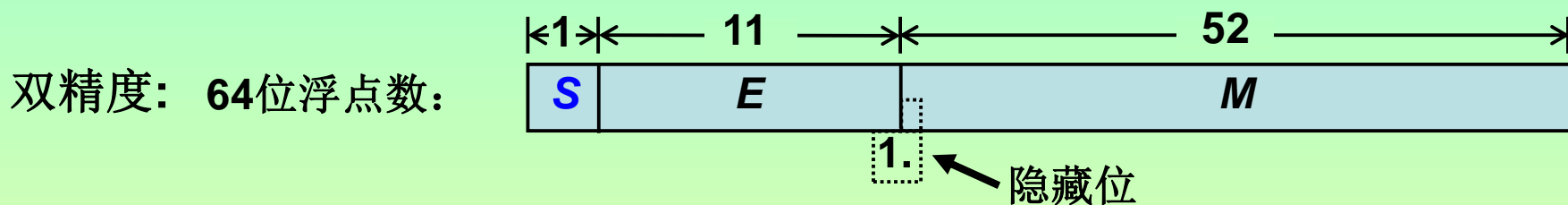
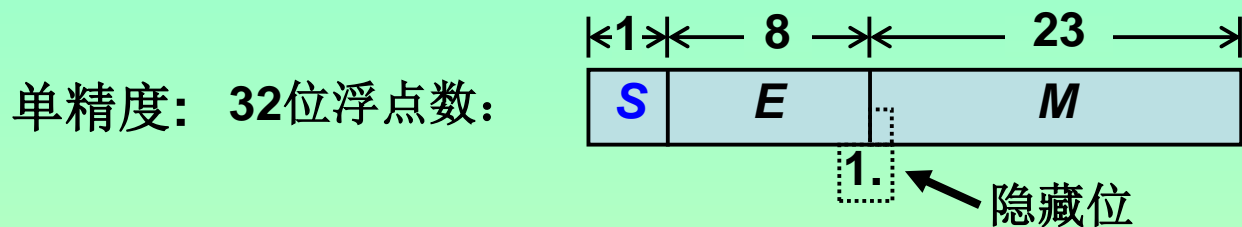
当一个浮点数的尾数 $M=0$ ，不管阶码取何值，或者阶码比它能表示的最小值还小时，不管尾数取何值，都把该浮点当零看待，称为机器零。此时要求把浮点数的阶码和尾数都清为零，保证零这个数表示的唯一性。

**浮点数的表示范围：**设阶码和尾数都用补码表示，阶码共 $k+1$ 位（含一位阶符），尾数共 $n+1$ 位（含一位数符）。浮点数的表示范围如下：

	阶码	尾数	真值
最大正数	0, 11...1	0. 11...1	$(1 - 2^{-n}) \times 2^{2^k - 1}$
绝对值最大负数	0, 11...1	1. 00...0	$(-1) \times 2^{2^k - 1}$
最小正数	1, 00...0	0. 00...1	$(2^{-n}) \times 2^{-2^k}$
规格化最小正数	1, 00...0	0. 10...0	$(2^{-1}) \times 2^{-2^k}$
绝对值最小负数	1, 00...0	1. 11...1	$(-2^{-n}) \times 2^{-2^k}$
规格化绝对值最小负数	1, 00...0	1. 01...1	$-(2^{-1} + 2^{-n}) \times 2^{-2^k}$

## IEEE 754 标准

IEEE 754是目前广泛采用的浮点数编码格式，分为单精度、双精度表示。



32位浮点数中， $S$ 是浮点数的符号位，占1位，安排在最高位，0正1负。符号位后面是阶码 $E$ ，占8位。 $E$ 的计算方法是将浮点数的指数真值加上一个固定的偏移值127（这也叫移码）。 $E$ 的取值范围是1~254，对应的指数真值为-126~+127。 $E$ 取255表示“无穷大”， $E$ 取0表示机器零。 $M$ 是尾数，占23位，放在最低位，用原码表示。尾数的真值为 $1.M$ ，即在 $M$ 前面有一个隐藏的1。这相当于将尾数规格化到1和2之间，而将整数部分的1隐藏不表示。

一个32位浮点数 $X$ 的真值可表示为：

$$12 \quad X = (-1)^S \times (1.M) \times 2^{E-127}$$

64位浮点数中，阶码 $E$ 占11位，计算方法是将浮点数的指数真值加上一个固定的偏移值1023。 $E$ 的取值范围是1~2046，对应的指数真值为-1022~+1023。 $E$ 取2047表示“无穷大”， $E$ 取0表示机器零。尾数 $M$ 占52位，有一个隐藏位1。

一个64位浮点数 $X$ 的真值可表示为：

$$X = (-1)^S \times (1.M) \times 2^{E-1023}$$

*IEEE 754*标准中的一些特殊值（以32位为例）：

	符号	阶码	尾数	值
正零：	0	0	0	+0
负零：	1	0	0	-0
正无穷大：	0	255	0	$+\infty$
负无穷大：	1	255	0	$-\infty$
非数：	0或1	255	$\neq 0$	NaN


注意：

本书中提及的规格化有两种，移码也有两种。

**IEEE754**标准指的移码以及规格化与浮点数表示不同。

	移 码	规 格 化
IEEE754标准	真值加 <b>127</b>	尾数的真值为 <b>1.M</b> 并将整数部分的 <b>1</b> 隐藏不表
浮点数	真值加 <b>2<sup>n</sup></b> 与补码区别符号位	原码： <b>1.1</b> ××或 <b>0.1</b> ×× 补码： <b>1.0</b> ××或 <b>0.1</b> ××

# IEEE 754 标准

- 将-27/64用IEEE754表示
- $-27/64 = -0.011011$
- 规格化: (必须是1.M的形式)  
 $0.011011 = 1.1011 \times 2^{-2}$
- 阶码为  $-2$      $E = e + 127 = 01111101$
- 1 01111101 1011000.....000  


符号 阶码**8**位                      尾数**23**位

用 IEEE754 的单精度表示方法表示 -0.75D

$$\begin{aligned}\text{解: } -0.75 &= 2^0 \times (-0.11000) \\ &= 2^{-1} \times (-1.1000)\end{aligned}$$

IEEE754 标准:

1 01111110 100..... 000000



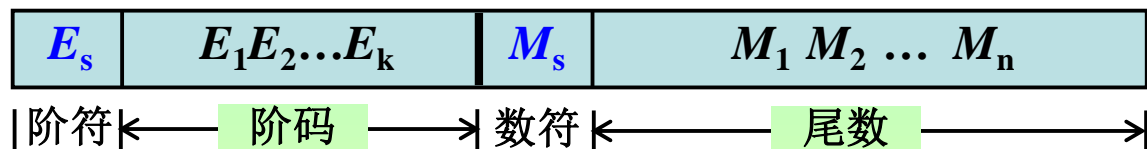
符号 阶码**8**位

尾数**23**位



# 牛刀小试

- 设某机器用32位表示一个实数，阶码部分8位（含1位阶符），用定点整数补码表示；尾数部分24位（含数符1位），用规格化定点小数补码表示，基数为2。



- 1) 求 $X=256.5$ 的浮点表示格式（16进制格式表示）
  - 2) 求 $Y=-256.5$ 的浮点表示格式（16进制格式表示）
- 将十进制数178.125表示成IEEE754单精度浮点数
  - 将下面IEEE754的单精度浮点数表示成十进制真值是多少？  
1011 ,1111,0101,1000,0000,0000,0000,0000

# 牛刀小试

$$X=(256.5)_{10}=+(100000000.1)_2=+(0.1000000001 \times 2^{+9})_2$$

8位阶码为:  $(+9)_{\text{补}}=0000\ 1001$

24位尾数为:  $(+0.10\ 0000\ 0001)_{\text{补}}$   
 $=0.100\ 0000\ 0010\ 0000\ 0000\ 0000$

所求256.5的浮点表示格式为:

0000 1001 0100 0000 0010 0000 0000  
0000

用16进制表示此结果则为:  $(09402000)_{16}$

[返回](#)

# 牛刀小试

$$Y = -(256.5)_{10} = -(100000000.1)_2$$

$$= -0.1000000001 \times 2^{+9}$$

8位阶码为:  $(+9)_{\text{补}} = 0000\ 1001$

24位尾数为:  $(-0.10\ 0000\ 0001)_{\text{补}}$   
 $= 1.011\ 1111\ 1110\ 0000\ 0000\ 0000$

所求-256.5的浮点表示格式为:

0000 1001 **1**011 1111 1110 0000 0000  
0000

用16进制表示此结果则为:  $(09\text{BFE}000)_{16}$

[返回](#)

# 牛刀小试

$$178.125 = 10110010.001B$$

$$= 1.0110010001 \times 2^7$$

$$\text{指数} E = 7 + 127 = 134 = 10000110B$$

127是单精度浮点数应加的指数偏移量，其完整的浮点数形式为：

0 10000110 011 0010 0010 0000 0000  
0000

$$= 43322000H$$

# 牛刀小试

1011 ,1111,0101,1000,0000,0000,0000,0000

数符:  $S=(-1)^1=-1$  (负号)

阶码:  $E=(01111110)_2-127=126-127=-1$

尾数:  $D=(1.1011)_2$

$X=-1.1011 \times 2^{-1} = -(0.11011)_2 = -0.84375$

注 意：

计算的每一个步骤都需认真仔细，不要出错。

考试时候不允许使用计算器

### 3.十进制数串的代表方法

十进制数串在计算机内主要有两种表示形式：

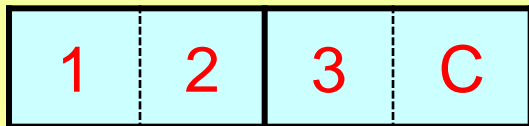
#### 字符串形式

一个字节存放一位十进制数或符号的ASCII码。这种表示方法主要用在非数值计算领域。

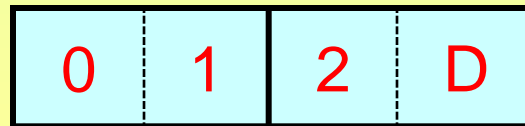
#### 压缩的十进制数串形式

每个字节存放两个十进制数位，每个数位占4位二进制数。通常存放该十进制数位的8421码。符号位也占半个字节，并存放在最低数值位之后，通常用1100表示正号，1101表示负号。在这种表示中，规定数字的个数加符号位之和必须为偶数；当和为奇数时，应在最高数值位之前补0。

+123:



-12:



# 字符和字符串

目前国际上普遍采用的字符系统是用**7**位二进制信息表示的美国国家信息交换标准码，简称**ASCII**码。

**ASCII**码是**128**个字符组成的字符集。其中编码值**0~31**和**127**为控制字符，用于通信中的通信控制或对计算机设备的功能控制。另外**95**个编码对应于**95**个可以从计算机终端输入并且可以显示和打印的有形字符。

一个字符在计算机中占据一个字节，用**8**位二进制数表示。其中最高位**b<sub>7</sub>**在传输过程中可用作奇偶校验位，存储字符时则取**0**。

**ASCII**码编码中有如下两个规律：

(1) 字符**0~9**这**10**个数字字符的高**3**位编码为**011**，低**4**位为**0000~1001**。当去掉高**3**位的值时，低**4**位正好是二进制形式的**0~9**。

(2) 英文字母的编码值满足正常的字母排序关系，且大小写英文字母编码的对应关系相当简便，差别仅表现在**b<sub>5</sub>**一位的值为**0**或**1**，



# ASCII码表

b6 b5 b4 b3 b2 b1 b0	000	001	010	011	100	101	110	111
0000	NUL	DLE	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	BEL	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	\	l	
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

字符串是指连续的一串字符，在主存中通常占用连续的多个字节，每个字节存放一个字符。当主存中一个字由若干个字节构成时，在同一个主存字内，字符串中各个字符既可从低字节向高字节依次存放，也可按从高字节到低字节顺序存放。

例：对于字符串 **IF A>B THEN READ(C)**

假定计算机字长**32**位，每个存储单元分成**4**个字节，每个字节存放一个字符，空格在主存中也占用一个字节位置。有以下两种存放方法：

I	F	空格	A
>	B	空格	T
H	E	N	空格
R	E	A	D
(	C	)	空格

A	空格	F	I
T	空格	B	>
空格	N	E	H
D	A	E	R
空格	)	C	(

# 汉字的表示

## 1.汉字的输入编码

汉字的输入编码主要有三种：数字编码、拼音码和字型码。

数字编码常用的是区位码。根据**GB 2321—80**方案，汉字基本字符集共**6763**个，还有西文字母、数字、图形符号等**682**个，共计**7445**个。所有的国标汉字与符号组成一个**94×94**的矩阵。其中的每一行称为一个“区”，区号为**01~94**；每一列称为一个“位”，位号也为**01~94**。**01~09**区为图形符号，**16~87**区为汉字。一个汉字所在的区号和位号简单地组合在一起就构成了该汉字的区位码。区码和位码各两位十进制数字，因此输入一个汉字需按键四次。例如“啊”位于**16**区**01**位，区位码为**1601**。

如果区号和位号都用**16**进制表示，并且都从**21H**开始编号（**ASCII**码中第一个可显示的字符就是**21H**），得到的编码称为汉字的国标码。例如“啊”的国标码为**3021 H**。国标码和区位码的关系为：

$$\text{国标码} = \text{区位码（16进制）} + 2020 \text{ H}$$

## 2.汉字的内码

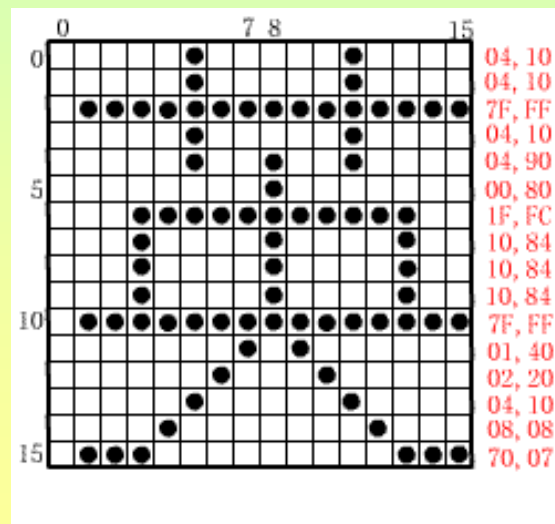
汉字内码是用于汉字信息的存储、交换、检索等操作的机内代码，一般采用两个字节表示。为了和**ASCII**码相互区别，汉字内码的两个字节的最高位均规定为“1”，即将国标码每个字节的最高位变为1：

内码 = 国标码 + **8080 H** = 区位码（16进制） + **A0A0 H**

例如“啊”的内码为**B0A1 H**。

## 3.汉字的字模码

构造汉字字型有两种方法，一是用向量法，一是用点阵法。目前普遍使用的汉字字型码是用点阵方式表示的。如一个汉字用**16×16**点阵表示，需要**32**个字节。



# 校验码

对于一种编码，任意两个合法码之间不同位的最小个数称为码距。例如用4位二进制码表示16个状态，则码距为1，这种编码没有检错能力。合理增大码距，可以提高发现错误的能力，但表示一定数量的合法码所需的二进制位数变多，增大了数据存储和传输的开销。

计算机内常遇到的错误有两大类：随机错误和突发错误。前者是孤立出现的一个错误，后者是连续产生的一批错误。本节只讨论前者。

## 1. 奇偶校验码

由数据位和一位校验位组成。对于奇校验，数据位和校验位中1的个数为奇数个；对于偶校验，数据位和校验位中1的个数为偶数个。

若数据位为 $D_{n-1} \dots D_1 D_0$ ，校验位为 $P$ ，则：

对于奇校验： $P = \overline{D_{n-1} \oplus \dots \oplus D_1 \oplus D_0}$

对于偶校验： $P = D_{n-1} \oplus \dots \oplus D_1 \oplus D_0$

奇偶校验码的码距为2，具有发现一位错误的能力。

## 2.海明校验码

海明码也是以奇偶校验为基础的，但校验位是一组而不是一位，它可以检测和定位一位错误。其原理是在 $k$ 个数据位中加入 $r$ 个校验位， $r$ 个校验位有 $2^r$ 个编码，用其中一个表示“没有错误”，其余 $2^r - 1$ 个编码指出错误出现在哪一位（可能是数据位，也可能是校验位），因此 $r$ 应满足：

$$2^r - 1 \geq k + r$$

例如 $r=4$ 则 $k$ 最大为11。

设4个校验位为 $P_1 \sim P_4$ ，11个数据位为 $D_0 \sim D_{10}$ ，形成海明码 $H_1 \sim H_{15}$ 。每个校验位 $P_i$ 被分配在海明码的第 $2^{i-1}$ 的位置上，其余位置为数据位，并按照从低位到高位顺序排列。即

$H_1$	$H_2$	$H_3$	$H_4$	$H_5$	$H_6$	$H_7$	$H_8$	$H_9$	$H_{10}$	$H_{11}$	$H_{12}$	$H_{13}$	$H_{14}$	$H_{15}$
$P_1$	$P_2$	$D_0$	$P_3$	$D_1$	$D_2$	$D_3$	$P_4$	$D_4$	$D_5$	$D_6$	$D_7$	$D_8$	$D_9$	$D_{10}$

每个数据位通过若干个校验位进行校验。例如 $D_8$ 是海明码的第13位 $H_{13}$ ， $13 = 1 + 4 + 8$ ，因此 $D_8$ 通过第1位 $P_1$ 、第4位 $P_3$ 、第8位 $P_4$ 进行校验。

## 海明码的分组

海明码	校验位位置	校验位
$H_1=P_1$	1	$P_1$
$H_2=P_2$	2	$P_2$
$H_3=D_0$	1, 2	$P_1, P_2$
$H_4=P_3$	4	$P_3$
$H_5=D_1$	1, 4	$P_1, P_3$
$H_6=D_2$	2, 4	$P_2, P_3$
$H_7=D_3$	1, 2, 4	$P_1, P_2, P_3$
$H_8=P_4$	8	$P_4$
$H_9=D_4$	1, 8	$P_1, P_4$
$H_{10}=D_5$	2, 8	$P_2, P_4$
$H_{11}=D_6$	1, 2, 8	$P_1, P_2, P_4$
$H_{12}=D_7$	4, 8	$P_3, P_4$
$H_{13}=D_8$	1, 4, 8	$P_1, P_3, P_4$
$H_{14}=D_9$	2, 4, 8	$P_2, P_3, P_4$
$H_{15}=D_{10}$	1, 2, 4, 8	$P_1, P_2, P_3, P_4$

从表中可以找出每个校验位分别校验那些码位，例如 $P_1$ 校验 $D_0$ ， $D_1$ ， $D_3$ ， $D_4$ ， $D_6$ ， $D_8$ 、 $D_{10}$ 和它自己。若采用偶校验，每个校验位就是它所校验的数据位的异或：

$$P_1=D_0\oplus D_1\oplus D_3\oplus D_4\oplus D_6\oplus D_8\oplus D_{10}$$

$$P_2=D_0\oplus D_2\oplus D_3\oplus D_5\oplus D_6\oplus D_9\oplus D_{10}$$

$$P_3=D_1\oplus D_2\oplus D_3\oplus D_7\oplus D_8\oplus D_9\oplus D_{10}$$

$$P_4=D_4\oplus D_5\oplus D_6\oplus D_7\oplus D_8\oplus D_9\oplus D_{10}$$

海明码传到接收方后，将校验位与它校验的数据位异或：

$$G_1=P_1\oplus D_0\oplus D_1\oplus D_3\oplus \dots \oplus D_8\oplus D_{10}$$

$$G_2=P_2\oplus D_0\oplus D_2\oplus D_3\oplus \dots \oplus D_9\oplus D_{10}$$

$$G_3=P_3\oplus D_1\oplus D_2\oplus D_3\oplus \dots \oplus D_9\oplus D_{10}$$

$$G_4=P_4\oplus D_4\oplus D_5\oplus D_6\oplus \dots \oplus D_9\oplus D_{10}$$

若 $G_1\sim G_4$ 全部为0则传送正确；若 $G_1G_2G_3G_4=0011$ 表明被 $P_3$ 和 $P_4$ 校验的位出错，即 $D_7$ 出错。

### 3.循环冗余校验码

在计算机网络、同步通信以及磁介质存储器中广泛使用循环冗余校验码，简称CRC码。CRC码是在 $k$ 位数据位之后拼接 $r$ 位校验位，形成 $n=k+r$ 位编码。编码中用到模2运算。

模2运算以按位模2相加为基础，运算时不考虑进位和借位。

(1) 模2加减：可以异或运算实现，模2加与模2减结果相同。

$$0 \pm 0 = 0, \quad 0 \pm 1 = 1, \quad 1 \pm 0 = 1, \quad 1 \pm 1 = 0.$$

(2) 模2乘：按模2加求部分积之和，不考虑进位。

$$\begin{array}{r} \phantom{00}1 \phantom{00}0 \phantom{00}1 \phantom{00}1 \\ \times \phantom{0000} \phantom{00}1 \phantom{00}1 \\ \hline \phantom{00}1 \phantom{00}0 \phantom{00}1 \phantom{00}1 \\ 1 \phantom{00}0 \phantom{00}1 \phantom{00}1 \\ \hline 1 \phantom{00}1 \phantom{00}1 \phantom{00}0 \phantom{00}1 \end{array}$$



(3) 模2除：按模2减求余数。上商的原则是余数首位为1时商取1，余数首位为0时商取0，余数位数小于除数位数时该余数就是最终的余数。

$$\begin{array}{r}
 \phantom{11} \phantom{11} \phantom{11} \\
 11 \overline{) 1000} \\
 \underline{11} \phantom{00} \\
 10 \phantom{00} \\
 \underline{11} \phantom{00} \\
 10 \phantom{00} \\
 \underline{11} \phantom{00} \\
 1
 \end{array}$$

编码方法：

(1) 待编码的 $k$ 位数据 $C_{k-1}C_{k-2}\dots C_1C_0$ 可表达为一个 $k-1$ 阶多项式：

$$M(x) = C_{k-1}x^{k-1} + C_{k-2}x^{k-2} + \dots C_1x + C_0$$

(2) 将 $k$ 个数据位左移 $r$ 位, 得到 $n=k+r$ 位代码, 对应的多项式为 $M(x)x^r$ :

$$C_{k-1}C_{k-2}\dots C_1C_000\dots0$$

(3) 用 $M(x)x^r$ 除以(模2除)某一个 $(r+1)$ 位的多项式 $G(x)$ (称为生成多项式), 得到商 $Q(x)$ 和 $r$ 位余数 $R(x)$ :

$$M(x)x^r = Q(x)G(x) + R(x)$$

用 $R(x)$ 代替 $M(x)x^r$ 右边的 $r$ 个0, 得到 $M(x)x^r + R(x)$ 就是所要的CRC编码。

### 校验原理:

因为 $M(x)x^r + R(x) = Q(x)G(x)$ (模2加和模2减结果相同), 准确传输的CRC码一定能被生成多项式 $G(x)$ 整除。若余数不为0则表示传输出错, 不同的余数表示不同的位出错。因此CRC码有纠正一位错误的能力。

### 生成多项式:

有三种生成多项式具有极高的检错率, 已经成为标准。它们是:

CRC-12:  $G(x) = x^{12} + x^{11} + x^3 + x^2 + x + 1$

CRC-16:  $G(x) = x^{16} + x^{15} + x^2 + 1$

CRC-CCITT:  $G(x) = x^{16} + x^{12} + x^5 + 1$

## 编码举例：

数据为1101，生成多项式为1011，求CRC编码。

将数据左移3位，得到1101000，再除以生成多项式1011，余数为001：

$$1101000 = 1111 \times 1011 + 001$$

于是CRC编码为1101001。

接收方得到CRC码后，除以生成多项式1011，余数为0表示传输正确。  
余数不为0表示某一位出错。例如余数为101表示最高位出错：

$$0101001 = 0100 \times 1011 + 101$$

余数与出错的位之间有固定的对应关系。

## CRC码的特点：

- (1) CRC码循环右移一位或循环左移一位，得到的新码仍是CRC码，并且校验位仍然在最右边的 $r$ 位；
- (2) 两个CRC码按位异或，得到的新码仍是CRC码。

## 作业

1、P60-61 3 4

2、参照例2.7 以定点小数为例，用数轴形式说明  
 $n+1$ 位（含一位数符）小数原码，补码，反码表示的范围

# 上讲复习

- 数的原码、补码、反码、移码的表示方法，注意边界情况（0的表示，表示范围， $-2^n$ 以及-1）
- 浮点数的表示方法：尾数用定点小数的原码或补码表示；指数用整数的补码或移码表示。数符、阶符分别表示什么含义？正、负、绝对值大于1  
若用补码表示，阶码尾数各有 $k+1$ 位  
规格化最小正数是： $1, 00\dots 0$      $0. 10\dots 0$      $(2^{-1}) \times 2^{-2^k}$
- 数的规格化表示方法
- **IEEE754**表示方法  
尾数用原码表示，规格化为 $1.M$ 的形式，表示时隐藏1  
阶码为真值加127

# 牛刀小试

- 设某机器用32位表示一个实数，阶码部分8位（含1位阶符），用定点整数补码表示；尾数部分24位（含数符1位），用规格化定点小数补码表示，基数为2。求 $X=256.5$ 的浮点表示格式  
 $256.5=+(100000000.1)B$
- 将十进制数178.125表示成IEEE754单精度浮点数  
 $178.125=10110010.001B$

所求256.5的浮点表示格式为：

0000 1001 0100 0000 0010 0000 0000 0000

$$178.125 = 10110010.001B$$

$$= 1.0110010001 \times 2^7$$

$$\text{指数} E = 7 + 127 = 134 = 10000110B$$

127是单精度浮点数应加的指数偏移量，其完整的浮点数形式为：

0 10000110 011 0010 0010 0000 0000  
0000

# 补码加法

计算机内部的数大部分都是以补码表示

例:  $[x]_{\text{补}} = 11111011$

$[y]_{\text{补}} = 00000011$  求  $[x+y]_{\text{补}}$

解: 由  $[x]_{\text{补}} = 11111011$  可知  $x = -5$

由  $[y]_{\text{补}} = 00000011$  可知  $y = 3$

所以  $x + y = -5 + 3 = -2$

$[-2]_{\text{补}} = 11111110$

$[x+y]_{\text{补}} = 11111110$



# 补码加法

用补码表示定点数，加减运算可以统一成加法运算。运算器中只需要一个加法器就可以了，不需要减法器。

补码加法的公式为：

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

证明：设 $x$ 和 $y$ 均为定点小数，且运算结果没有溢出。即

$$|x| < 1, \quad |y| < 1, \quad |x + y| < 1$$

(1)  $x > 0, y > 0$ ，则 $x + y > 0$ 。正数的补码和原码是一样的，可得：

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

(2)  $x > 0, y < 0, x + y \geq 0$ 或 $x + y < 0$ 。

$$[x]_{\text{补}} = x, \quad [y]_{\text{补}} = 2 + y$$

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

当 $x + y \geq 0$ 时， $2 + (x + y) \geq 2$ ，对于模2来说，2被丢掉，于是

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

当 $x + y < 0$ 时， $2 + (x + y) < 2$ ，于是

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

(3)  $x < 0, y > 0$ 。

这种情况和第2种情况证明方法相同。

(4)  $x < 0, y < 0$ , 则  $x + y < 0$ 。

$$[x]_{\text{补}} = 2 + x, \quad [y]_{\text{补}} = 2 + y$$

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

$(2 + x + y)$  小于2而大于1的数, 第一个2作为进位被舍弃。于是

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

至此我们证明了在模2意义下, 两数的补码之和等于这两个数之和的补码。这个结论也适用于定点整数。

例:  $x = +0.1011, y = -0.0101$ , 求  $x + y$ 。

在模2意义下,  
超过2的进位应舍弃

$$\begin{array}{r} 011, \quad [y]_{\text{补}} = 1.1011 \\ [x]_{\text{补}} = 0.1011 \\ \hline [x + y]_{\text{补}} = 10.0110 \end{array}$$

符号位作为数的一部分参加运算

$$x + y = 0.0110$$

例:  $[x]_{\text{补}} = 11111011$

$[y]_{\text{补}} = 00000011$  求  $[x+y]_{\text{补}}$

解一:

由  $[x]_{\text{补}} = 11111011$  可知  $x = -5$

由  $[y]_{\text{补}} = 00000011$  可知  $y = 3$

所以  $x + y = -5 + 3 = -2$

$[-2]_{\text{补}} = 11111110$

$[x+y]_{\text{补}} = 11111110$

解二:

$$[x+y]_{\text{补}} = [x]_{\text{补}} + [y]_{\text{补}} = 11111011 + 00000011 = 11111110$$

$$\begin{array}{r} 11111011 \\ + 00000011 \\ \hline 11111110 \end{array}$$

# 补码减法

数用补码表示时，减法运算可转化为加法运算。公式为：

$$\underline{[x-y]_{\text{补}} = [x+(-y)]_{\text{补}} = [x]_{\text{补}} + [-y]_{\text{补}} \pmod{2}}$$

已知 $[y]_{\text{补}}$ 求 $[-y]_{\text{补}}$ 的方法是：将 $[y]_{\text{补}}$ 的所有位（包括符号位）一起取反，然后在最低位加1（对定点小数相当于加 $2^{-n}$ ）：

$$\underline{[-y]_{\text{补}} = \neg [y]_{\text{补}} + 2^{-n}}$$

证明：

求补运算

不同于求一个数的补码

(1) 若 $y$ 为正数， $1 > y \geq 0$ 。 $[y]_{\text{补}} = 0.y_1y_2 \dots y_n = y$ 。

$$\begin{aligned} [-y]_{\text{补}} &= 2 + (-y) = 2 - 0.y_1y_2 \dots y_n \\ &= 2^{-n} + (1.11 \dots 1 - 0.y_1y_2 \dots y_n) \\ &= \neg 0.y_1y_2 \dots y_n + 2^{-n} \end{aligned}$$

(2) 若 $y$ 为负数， $0 > y > -1$ 。 $[y]_{\text{补}} = 1.y_1y_2 \dots y_n = 2 + y$ 。

$$\begin{aligned} [-y]_{\text{补}} &= -y = 2 - 1.y_1y_2 \dots y_n \\ &= 2^{-n} + (1.11 \dots 1 - 1.y_1y_2 \dots y_n) \\ &= \neg 1.y_1y_2 \dots y_n + 2^{-n} \end{aligned}$$

例：  $x = -1110$ ,  $y = +1101$

$$[x]_{\text{补}} = 10010$$

$$[-x]_{\text{补}} = 01110$$

$$[y]_{\text{补}} = 01101$$

$$[-y]_{\text{补}} = 10011$$

例：  $x = +1101$ ,  $y = +0110$ , 求  $x - y$

$$[x]_{\text{补}} = 01101$$

$$[y]_{\text{补}} = 00110$$

$$[-y]_{\text{补}} = 11010$$

$$x - y = +0111$$

$[x]_{\text{补}}$	01101
$+[-y]_{\text{补}}$	11010
<hr/>	
$[x-y]_{\text{补}}$	100111

例：  $x = -1010$ ,  $y = +1001$  求  $[x]_{\text{补}}$   $[-x]_{\text{补}}$   $[y]_{\text{补}}$   $[-y]_{\text{补}}$   
求  $[x-y]_{\text{补}}$

$$[x]_{\text{补}} = 10110$$

$$[-x]_{\text{补}} = 01010$$

$$[y]_{\text{补}} = 01001$$

$$[-y]_{\text{补}} = 10111$$

$$\begin{array}{r} [x]_{\text{补}} \quad 10110 \\ + [-y]_{\text{补}} \quad 10111 \\ \hline [x-y]_{\text{补}} \quad 101101 \end{array}$$

溢出

## 溢出检测

运算结果超出了机器允许的表示范围称为溢出。两个同符号数相加，如果结果的符号与参加运算的操作数的符号相反，则表明有溢出。两个正数相加产生的溢出称为上溢；两个负数相加产生的溢出称为下溢。

**[例12]**  $x = +0.1011$ ,  $y = +0.1001$ , 求  $x + y$ 。

$$[x]_{\text{补}} = 0.1011 \quad [y]_{\text{补}} = 0.1001$$

$$\begin{array}{r} [x]_{\text{补}} \quad \quad 0.1011 \\ + [y]_{\text{补}} \quad \quad 0.1001 \\ \hline [x+y]_{\text{补}} \quad 1.0100 \end{array}$$

**[例13]**  $x = -0.1101$ ,  $y = -0.1011$ , 求  $x + y$ 。

$$[x]_{\text{补}} = 1.0011 \quad [y]_{\text{补}} = 1.0101$$

$$\begin{array}{r} [x]_{\text{补}} \quad \quad 1.0011 \\ + [y]_{\text{补}} \quad \quad 1.0101 \\ \hline [x+y]_{\text{补}} \quad 0.1000 \end{array}$$

## 1.单符号位法

两个正数相加，最高有效位产生进位而符号位无进位（此时符号位从**0**变成**1**）则产生上溢；两个负数相加，最高有效位无进位而符号位有进位（此时符号位从**1**变成**0**）则产生下溢。

溢出逻辑表达式为：

$$V = C_f \oplus C_0$$

其中 $C_f$ 为符号位产生的进位， $C_0$ 为最高有效位产生的进位。



## 2.双符号位法

双符号位补码也称“变形补码”或“模4补码”。通过采用双符号位，可使所能表示的数的范围比模2补码扩大一倍。其定义为：

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

用 $S_{f1}$ 和 $S_{f2}$ 表示 $[x]_{\text{补}}$ 最高符号位和第二符号位，则：

$$S_{f1} S_{f2} = 00 \quad 0 \leq x < 1$$

$$S_{f1} S_{f2} = 01 \quad 1 \leq x < 2$$

$$S_{f1} S_{f2} = 11 \quad -1 \leq x < 0$$

$$S_{f1} S_{f2} = 10 \quad -2 \leq x < -1$$

模4补码的运算规则为：

$$[x]_{\text{补}} + [y]_{\text{补}} = [x + y]_{\text{补}} \pmod{4}$$

若运算结果中 $S_{f1}$ 和 $S_{f2}$ 相同则无溢出；若 $S_{f1} S_{f2} = 01$ 表示上溢；若 $S_{f1} S_{f2} = 10$ 表示下溢。溢出逻辑表达式为 $S_{f1} \oplus S_{f2}$ 。无论是否溢出，最高符号位 $S_{f1}$ 永远表示运算结果的正确符号。

为了得到两数变形补码之和等于两数之和的变形补码, 同样必须:

1. 两个符号位都看作数码一样参加运算
2. 两数进行以4位模的加法, 即最高符号位上产生的进位要丢掉。

采用变形补码后, 如果两个数相加后, 其结果的符号位出现“01”或“10”两种组合时, 表示发生溢出。这是因为两个绝对值小于1的数相加, 其结果不会大于或等于2, 所以最高符号位永远表示结果的正确符号。

采用变形补码后, 正数的符号为00, 负数的符号为11

[例14]  $x = +0.1100$ ,  $y = +0.1000$ , 求  $x + y$ 。

$[x]_{\text{补}} = 00.1100$ ,  $[y]_{\text{补}} = 00.1000$

$$\begin{array}{r} [x]_{\text{补}} \quad 00.1100 \\ + [y]_{\text{补}} \quad 00.1000 \\ \hline 01.0100 \end{array}$$

两个符号位出现“01”，表示已溢出，即结果大于+1。

[例15]  $x = -0.1100$ ,  $y = -0.1000$ , 求  $x + y$ 。

$[x]_{\text{补}} = 11.0100$ ,  $[y]_{\text{补}} = 11.1000$

$$\begin{array}{r} [x]_{\text{补}} \quad 11.0100 \\ + [y]_{\text{补}} \quad 11.1000 \\ \hline 110.1100 \end{array}$$

两个符号位出现“10”，表示已溢出，即结果小于-1。

用双符号位补码求解

$x = +0.1101$ ,  $y = +0.1010$ , 求  $x + y$

$[x]_{\text{补}} = 00.1101$ ,  $[y]_{\text{补}} = 00.1010$

$$\begin{array}{r} [x]_{\text{补}} \quad \quad \quad 00.1101 \\ + [y]_{\text{补}} \quad \quad \quad 00.1010 \\ \hline \quad \quad \quad \quad \quad 01.0111 \end{array}$$

两个符号位出现“01”，表示已溢出，即结果大于+1。

两个二进制数字 $A_i$ ,  $B_i$ 和一个进位输入 $C_i$ 相加, 产生一个和输出 $S_i$ , 以及一个进位输出 $C_{i+1}$ 。

表2.2 一位全加器真值表

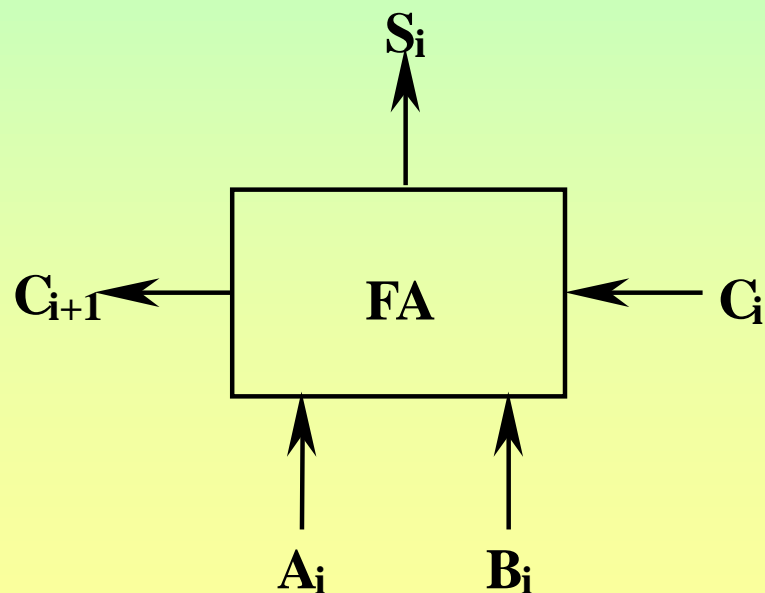
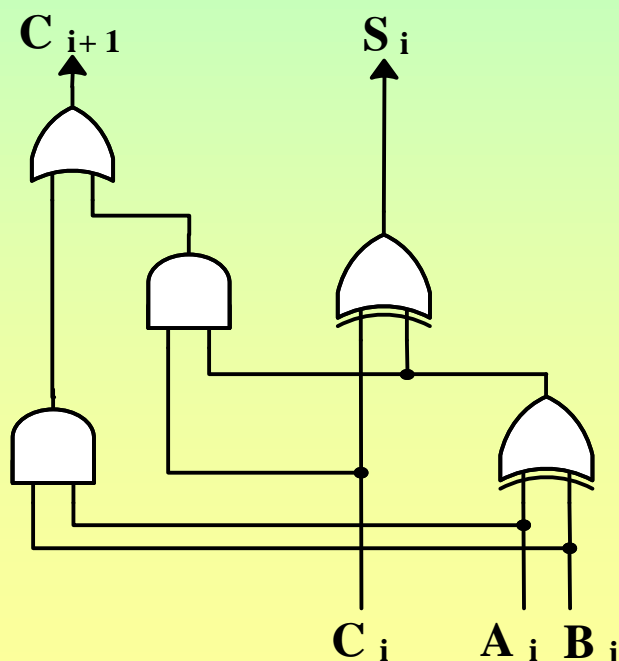
输入			输出	
$A_i$	$B_i$	$C_i$	$S_i$	$C_{i+1}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

# 一位全加器

全加器（Full Adder, FA）将两个二进制数字 $A_i$ 、 $B_i$ 和一个进位输入 $C_i$ 相加，产生一个和输出 $S_i$ ，以及一个进位输出 $C_{i+1}$ 。



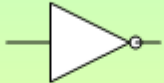




$$S_i = A_i \oplus B_i \oplus C_i$$

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



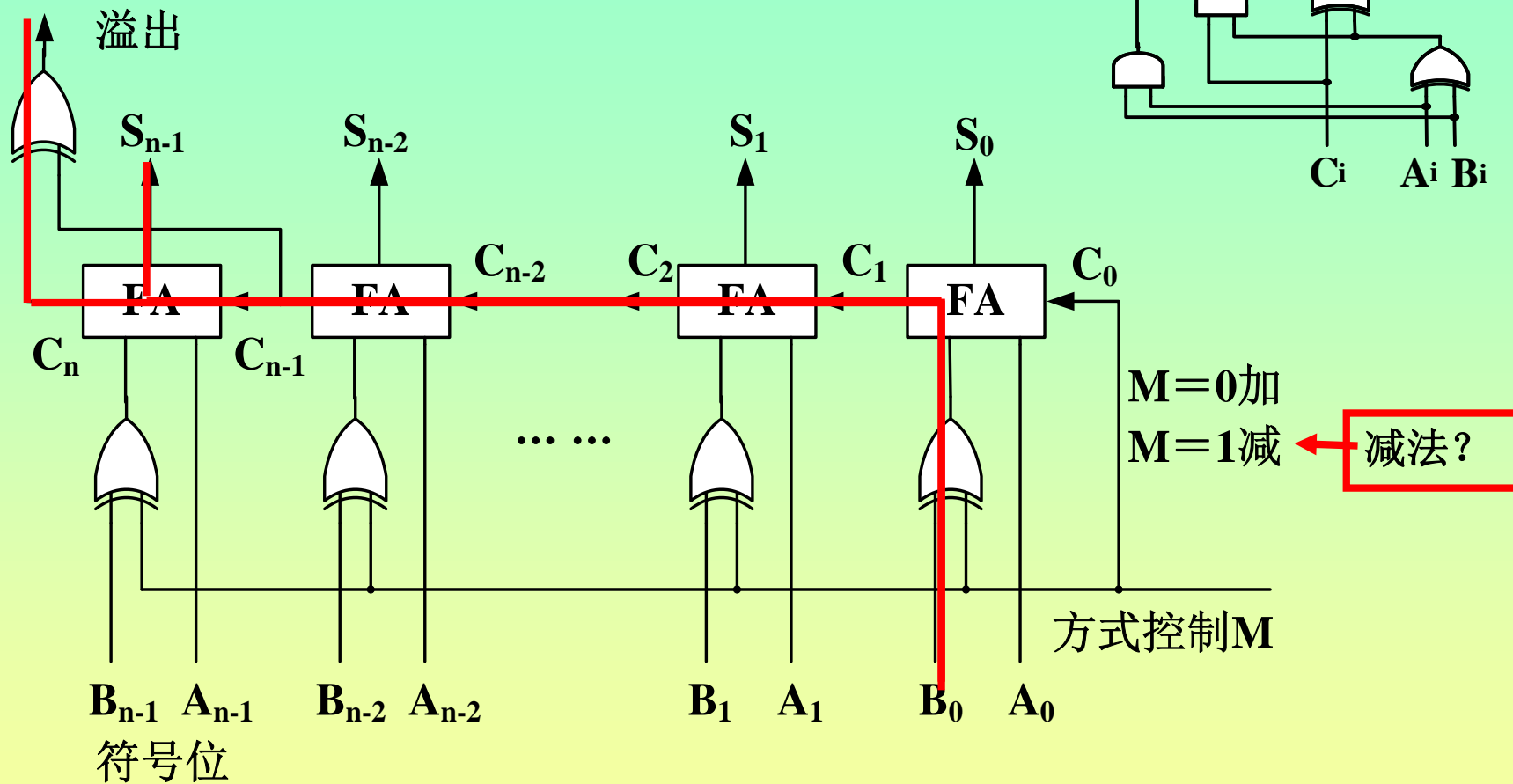
# 门的延迟时间

常用门电路的国外常用符号和延迟时间。 $T$ 表示一个与门或一个或门的延迟时间。

门	逻辑符号	延迟时间
与非		$T$
或非		$T$
非		$T$
与		$T$
或		$T$
异或		$3T$
同或		$3T$

对一位全加器来说， $S_i$ 的时间延迟为 $6T$ ， $C_{i+1}$ 的时间延迟为 $5T$ 。

# 行波进位补码加/减法器



考虑溢出检测时，延迟时间为  $T_a = 3T + 5T + (n-1) \times 2T + 3T = (2n+9)T$

不考虑溢出检测时，延迟时间为  $T_a = 3T + 5T + (n-2) \times 2T + 3T = (2n+7)T$



# 先行进位加法器

行波进位加法器的进位延迟时间和字长成正比。为缩短进位时间，可采用先行进位方式（Carry Look Ahead, CLA）。

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

定义

$$G_i = A_i B_i, \quad P_i = A_i \oplus B_i$$

$G_i$ 称为进位产生函数，含义是两个加数均为1时，必然向高位产生进位； $P_i$ 称为进位传递函数，含义是两个加数中有一个为1时，低位来的进位 $C_i$ 将超越本位向更高位传递。

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

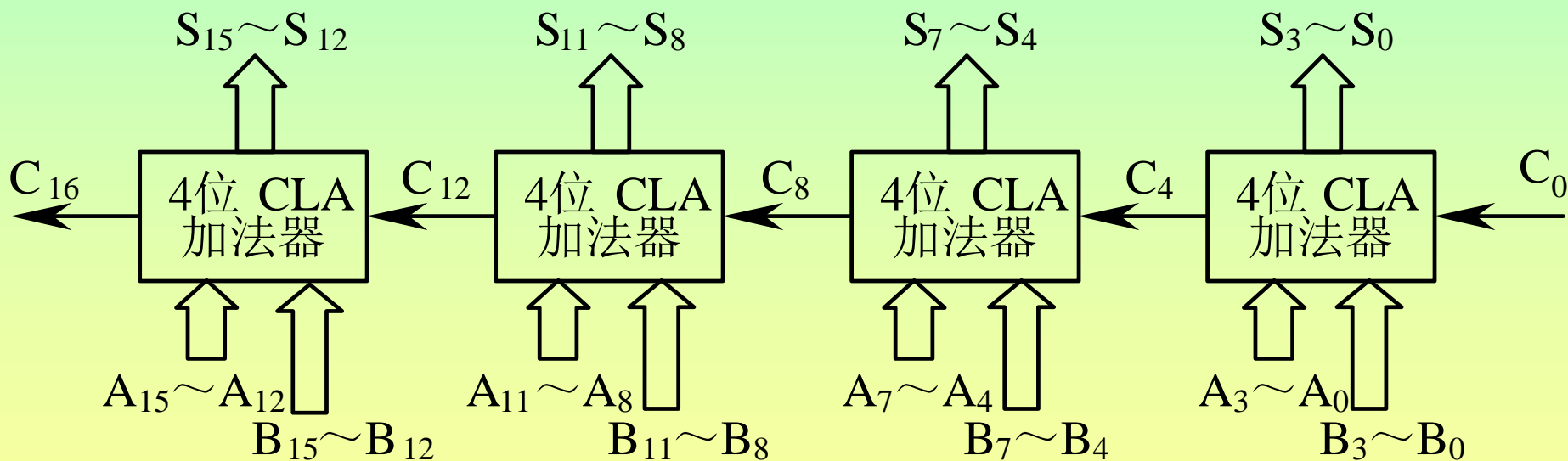
$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$$

各位进位可同时产生，与字长无关。但随着字长的增加， $C_i$ 的表达式越来越复杂，而且受到元器件扇入系数的限制，所以完全采用先行进位是不现实的。

# 单级先行进位

当加法器字长较大时，可采用分组先行进位。以16位加法器为例，可分为4组，每组4位。第一组内的 $C_1$ 、 $C_2$ 、 $C_3$ 和 $C_4$ 通过CLA电路同时产生。 $C_4$ 产生后，作为第二组的进位输入，产生 $C_5$ 、 $C_6$ 、 $C_7$ 和 $C_8$ 。这种组内先行进位，组间串行进位的方式，称为单级先行进位。



单级先行进位电路中，进位延迟时间和组数成正比。组数越多，进位延迟时间越长。当字长较大(>16)时，有必要采用多级先行进位方式。

# 多级先行进位

多级先行进位是指组内和组间都采用先行进位。仍以16位加法器为例，第一小组的最高位进位 $C_4$ 可以写成：

$$\begin{aligned}C_4 &= G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 \\&= G_1^* + P_1^* C_0\end{aligned}$$

$G_1^*$ 称为组进位产生函数， $P_1^*$ 称为组进位传递函数，它们只和 $P_i$ 、 $G_i$ 有关。

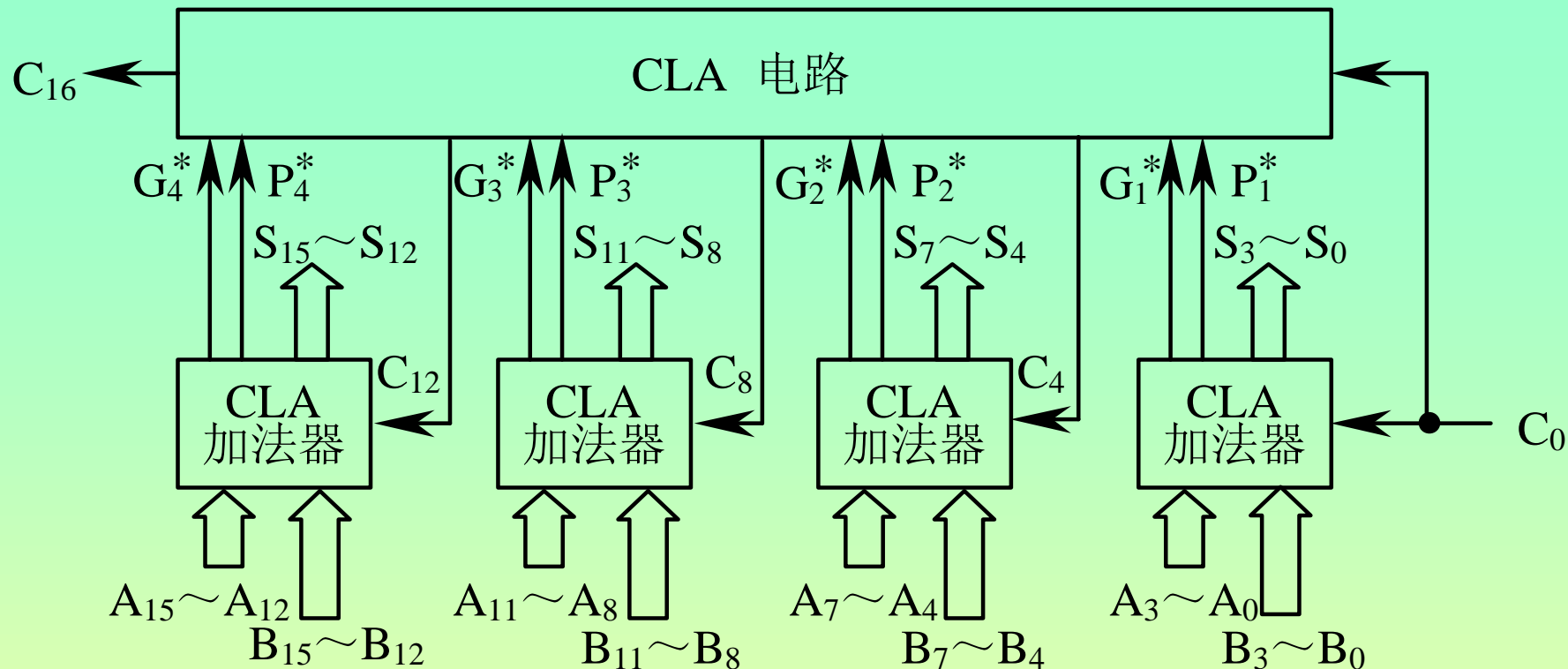
$$C_4 = G_1^* + P_1^* C_0$$

$$C_8 = G_2^* + P_2^* C_4 = G_2^* + P_2^* G_1^* + P_2^* P_1^* C_0$$

$$C_{12} = G_3^* + P_3^* C_8 = G_3^* + P_3^* G_2^* + P_3^* P_2^* G_1^* + P_3^* P_2^* P_1^* C_0$$

$$C_{16} = G_4^* + P_4^* C_{12} = G_4^* + P_4^* G_3^* + P_4^* P_3^* G_2^* + P_4^* P_3^* P_2^* G_1^* + P_4^* P_3^* P_2^* P_1^* C_0$$

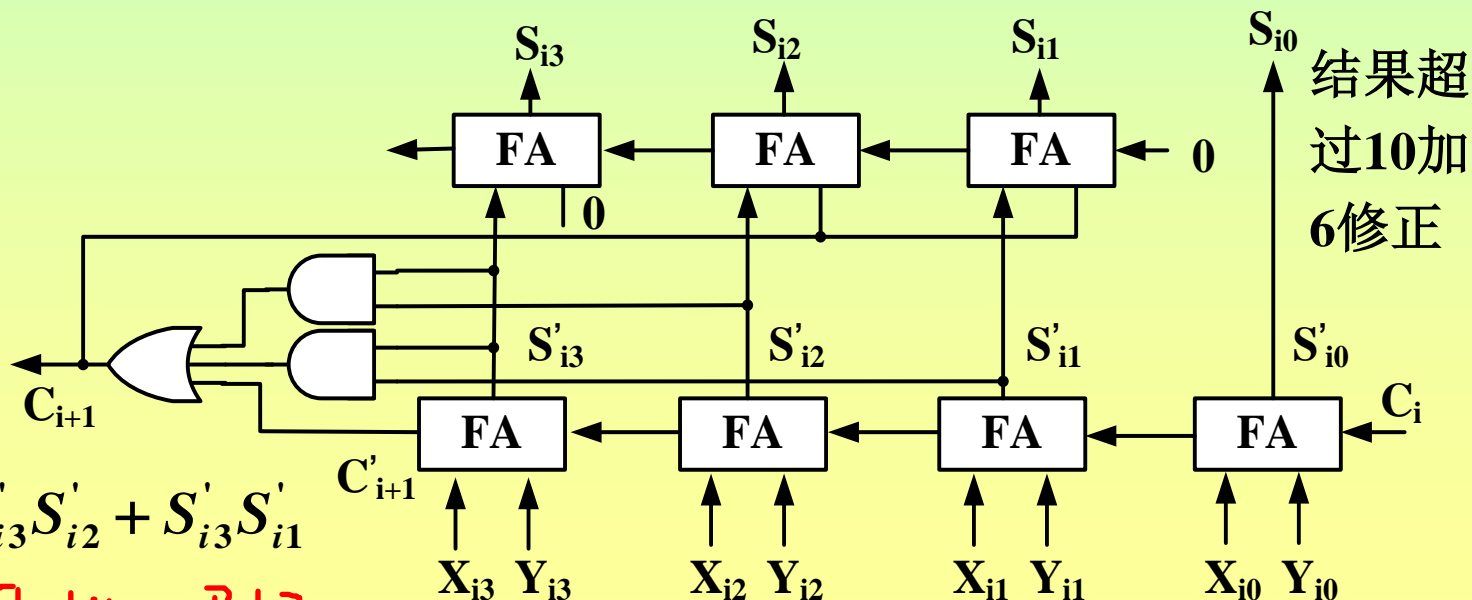
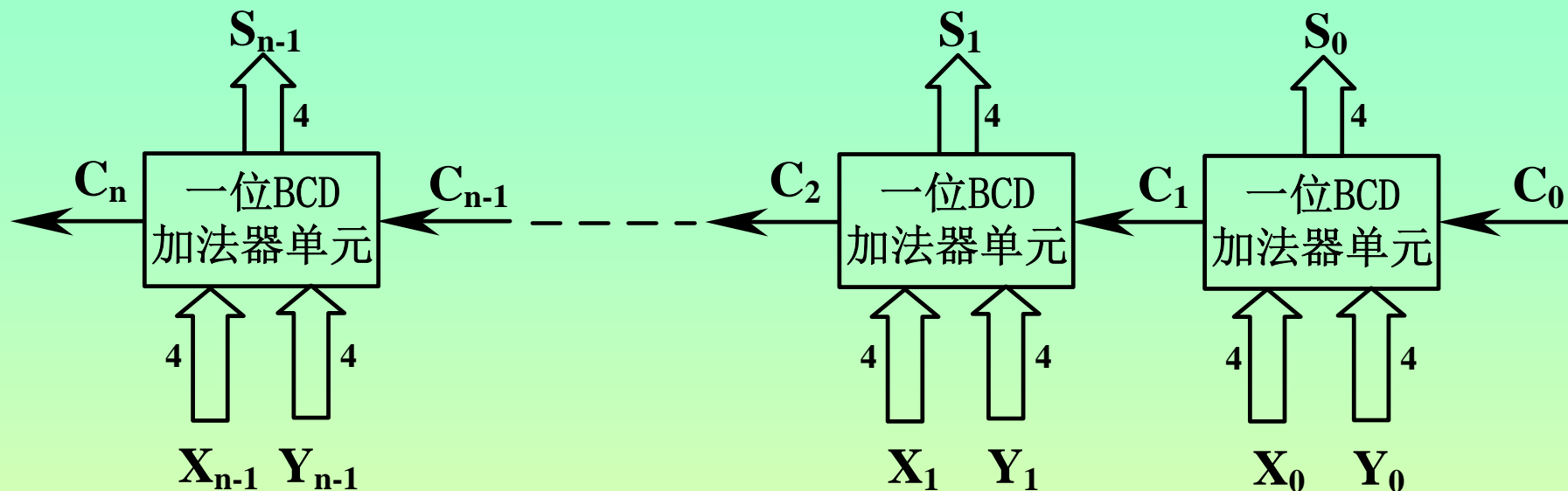
每个4位CLA加法器产生相应的组进位函数 $G_i^*$ 和 $P_i^*$ ，将这些组进位函数再送到CLA电路中，就可以同时产生 $C_4$ 、 $C_8$ 、 $C_{12}$ 和 $C_{16}$ 。



上图是一个两级先行进位的16位加法器。设与门和或门的延迟时间均为 $T_y$ ，该电路的工作过程是：先产生 $G_i$ 和 $P_i$  ( $i=0 \sim 15$ )，经过 $2T_y$ 后产生所有的组进位函数 $G_i^*$ 和 $P_i^*$  ( $i=1 \sim 4$ ) 以及第一组的 $C_1$ 、 $C_2$ 和 $C_3$ ，再经过 $2T_y$ ，由CLA电路产生 $C_4$ 、 $C_8$ 、 $C_{12}$ 和 $C_{16}$ ，再经过 $2T_y$ ，产生第2、3、4组的 $C_5 \sim C_7$ 、 $C_9 \sim C_{11}$ 和 $C_{13} \sim C_{15}$ 。

用同样办法可构造多级先行进位加法器，如用三级先行进位结构设计64位加法器，这种加法器的字长对加法时间影响甚小。

# 十进制加法器



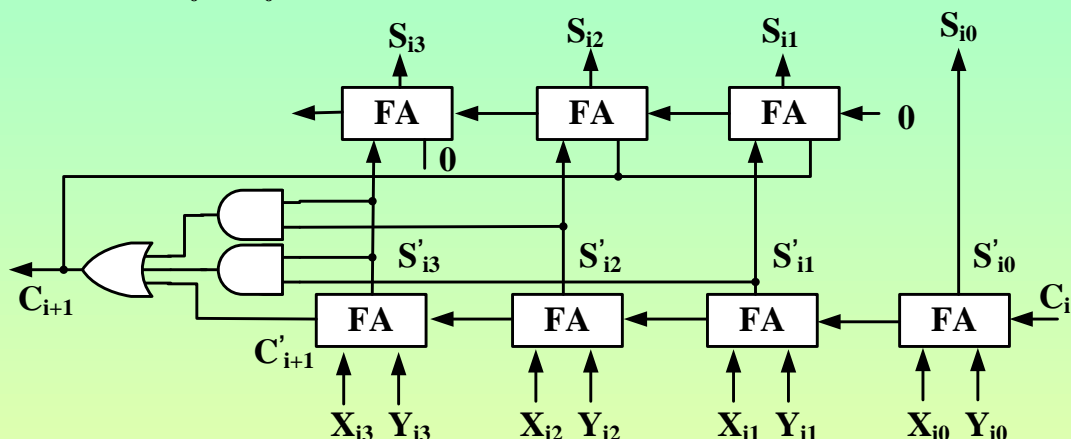
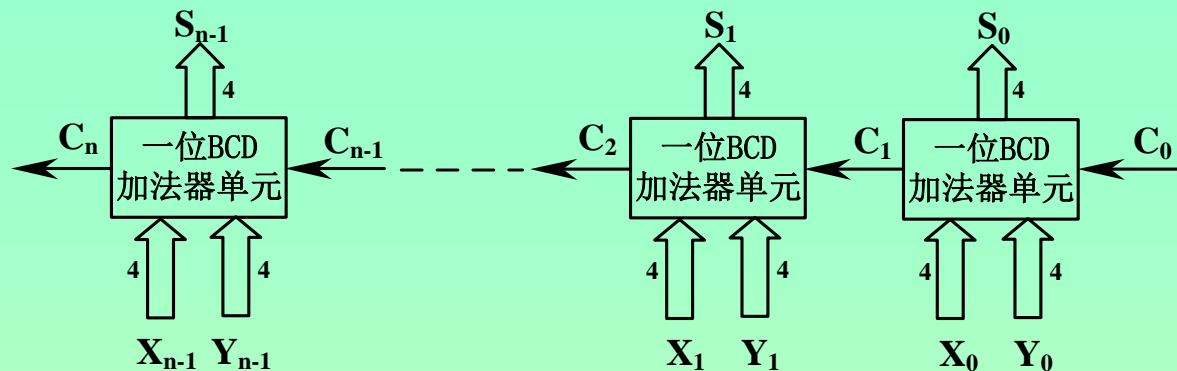
$$C_{i+1} = C'_{i+1} + S'_{i3}S'_{i2} + S'_{i3}S'_{i1}$$

两个一位十进制数相加，并考虑低位的进位，结果范围0~19。

和	加法器输出					修正后输出					和	加法器输出					修正后输出				
	C'	S <sub>3</sub> '	S <sub>2</sub> '	S <sub>1</sub> '	S <sub>0</sub> '	C	S <sub>3</sub>	S <sub>2</sub>	S <sub>1</sub>	S <sub>0</sub>		C'	S <sub>3</sub> '	S <sub>2</sub> '	S <sub>1</sub> '	S <sub>0</sub> '	C	S <sub>3</sub>	S <sub>2</sub>	S <sub>1</sub>	S <sub>0</sub>
0	0	0	0	0	0	0	0	0	0	0	10	0	1	0	1	0	1	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1	11	0	1	0	1	1	1	0	0	0	1
2	0	0	0	1	0	0	0	0	1	0	12	0	1	1	0	0	1	0	0	1	0
3	0	0	0	1	1	0	0	0	1	1	13	0	1	1	0	1	1	0	0	1	1
4	0	0	1	0	0	0	0	1	0	0	14	0	1	1	1	0	1	0	1	0	0
5	0	0	1	0	1	0	0	1	0	1	15	0	1	1	1	1	1	0	1	0	1
6	0	0	1	1	0	0	0	1	1	0	16	1	0	0	0	0	1	0	1	1	0
7	0	0	1	1	1	0	0	1	1	1	17	1	0	0	0	1	1	0	1	1	1
8	0	1	0	0	0	0	1	0	0	0	18	1	0	0	1	0	1	0	0	0	0
9	0	1	0	0	1	0	1	0	0	1	19	1	0	0	1	1	1	0	0	0	1

# 十进制加法器

在十进制运算时,当相加二数之和大于9时,便产生进位。可是用**BCD**码完成十进制数运算时,当和数大于9时,必须对和数进行加6修正。这是因为,采用**BCD**码后,在二数相加的和数小于等于9时,十进制运算的结果是正确的;而当相加的和数大于9时,结果不正确,必须加6修正后才能得出正确的结果。



例如:  $X=15$  (D)     $Y=26$  (D)    计算  $Z = X+Y = 41$ (D)

$X_0=5 = 0101$

$Y_0=6 = 0110$

$X_1=1 = 0001$

$Y_1=2 = 0010$

修正之后应该为:

$S_0=1=0001$

$C_1=1$

$S_1=4=0100$

若不进行修正, 结果为  $S_0=X_0+Y_0=B(H)=1011$

$S_1=X_1+Y_1=3(H)=0011$



作业

P61

5(2) 6(2)(3) 13

5(3) 6(1) 12

# 原码乘法

## 1.人工算法与机器算法的同异性

两个原码表示的数相乘的运算规则是：乘积的符号位由两数的符号位按异或运算得到，而乘积的数值部分则是两个正数相乘之积。

被乘数:  $[x]_{\text{原}} = x_f . x_{n-1} \dots x_1 x_0$

乘数： $[y]_{\text{原}} = y_f \cdot y_{n-1} \cdots y_1 y_0$

乘积:  $[z]_{\text{原}} = (\mathbf{x}_f \oplus \mathbf{y}_f) + (0.\mathbf{x}_{n-1} \dots \mathbf{x}_1 \mathbf{x}_0)(0.\mathbf{y}_{n-1} \dots \mathbf{y}_1 \mathbf{y}_0)$

例:  $x=0.1101$ ,  $y=0.1011$ :

$$\begin{array}{r}
 \begin{array}{cccccc}
 & & & 0. & 1 & 1 & 0 & 1 & (\times) \\
 \times & & & 0. & 1 & 0 & 1 & 1 & (y) \\
 \hline
 & & & & 1 & 1 & 0 & 1 \\
 & & & & & 1 & 1 & 0 & 1 \\
 & & & & & & 0 & 0 & 0 & 0 \\
 + & & 1 & 1 & 1 & 0 & 1 & & & \\
 \hline
 0. & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & (z)
 \end{array}
 \end{array}$$

例对于只有加法器和移位器的计算机，可以通过多次执行“加法—移位”操作来实现乘法。

## 2.不带符号的阵列乘法器

设A和B是m位和n位的不带符号的二进制整数，相乘产生m+n位乘积P:

$$\mathbf{A} = \mathbf{a}_{m-1} \dots \mathbf{a}_1 \mathbf{a}_0, \quad \mathbf{B} = \mathbf{b}_{n-1} \dots \mathbf{b}_1 \mathbf{b}_0$$

$$P = p_{m+n-1} \cdots p_1 p_0$$

则乘积P的数值为:

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

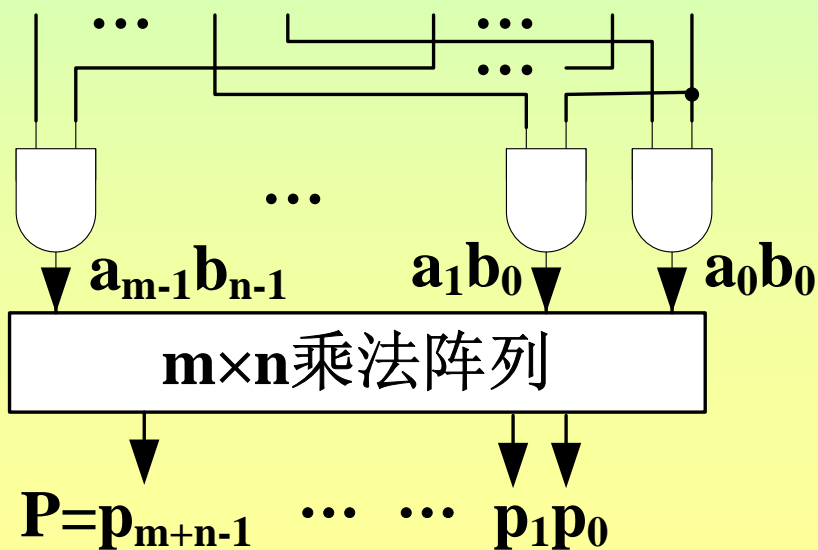
					$\times$	$\mathbf{a}_4$ $\mathbf{b}_4$	$\mathbf{a}_3$ $\mathbf{b}_3$	$\mathbf{a}_2$ $\mathbf{b}_2$	$\mathbf{a}_1$ $\mathbf{b}_1$	$\mathbf{a}_0$ $\mathbf{b}_0$
						$\mathbf{a}_4\mathbf{b}_0$	$\mathbf{a}_3\mathbf{b}_0$	$\mathbf{a}_2\mathbf{b}_0$	$\mathbf{a}_1\mathbf{b}_0$	$\mathbf{a}_0\mathbf{b}_0$
					$\mathbf{a}_4\mathbf{b}_1$	$\mathbf{a}_3\mathbf{b}_1$	$\mathbf{a}_2\mathbf{b}_1$	$\mathbf{a}_1\mathbf{b}_1$	$\mathbf{a}_0\mathbf{b}_1$	
				$\mathbf{a}_4\mathbf{b}_2$	$\mathbf{a}_3\mathbf{b}_2$	$\mathbf{a}_2\mathbf{b}_2$	$\mathbf{a}_1\mathbf{b}_2$	$\mathbf{a}_0\mathbf{b}_2$		
		$\mathbf{a}_4\mathbf{b}_3$	$\mathbf{a}_3\mathbf{b}_3$	$\mathbf{a}_2\mathbf{b}_3$	$\mathbf{a}_1\mathbf{b}_3$	$\mathbf{a}_0\mathbf{b}_3$				
$+$	$\mathbf{a}_4\mathbf{b}_4$	$\mathbf{a}_3\mathbf{b}_4$	$\mathbf{a}_2\mathbf{b}_4$	$\mathbf{a}_1\mathbf{b}_4$	$\mathbf{a}_0\mathbf{b}_4$					
$\mathbf{p}_9$	$\mathbf{p}_8$	$\mathbf{p}_7$	$\mathbf{p}_6$	$\mathbf{p}_5$	$\mathbf{p}_4$	$\mathbf{p}_3$	$\mathbf{p}_2$	$\mathbf{p}_1$	$\mathbf{p}_0$	

$a_i b_j$ 称为位积。 $m \times n$ 个位积可以通过 $m \times n$ 个与门并行产生。

## 2.不带符号的阵列乘法器

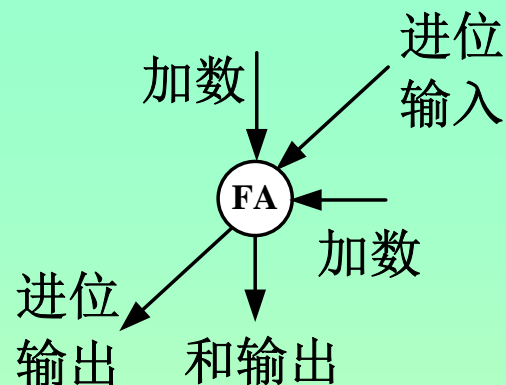
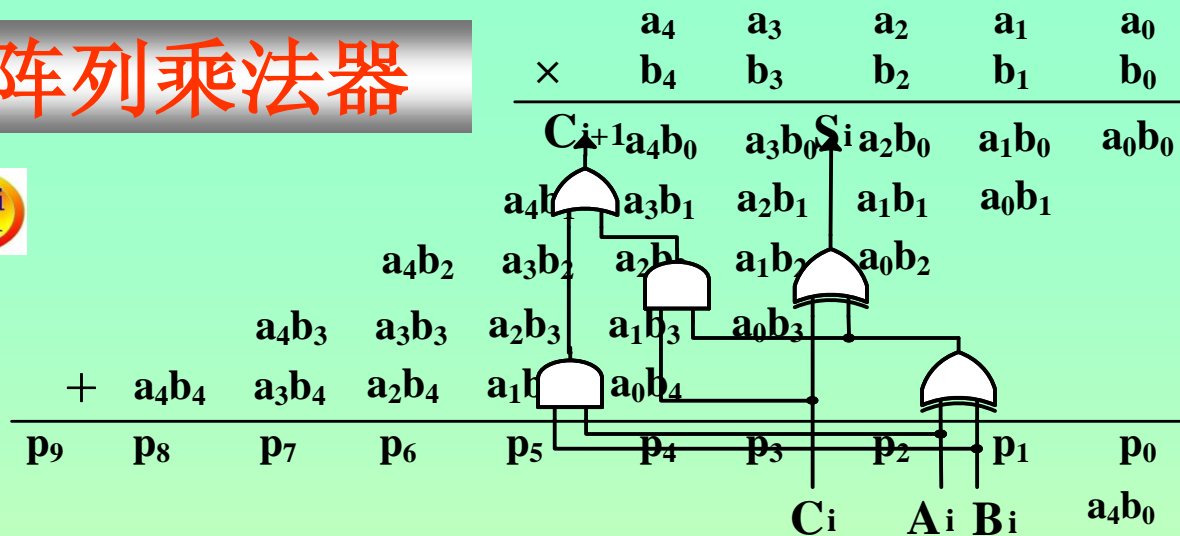
						$a_4$	$a_3$	$a_2$	$a_1$	$a_0$
					$\times$	$b_4$	$b_3$	$b_2$	$b_1$	$b_0$
						$a_4b_0$	$a_3b_0$	$a_2b_0$	$a_1b_0$	$a_0b_0$
				$a_4b_1$	$a_3b_1$	$a_2b_1$	$a_1b_1$	$a_0b_1$		
		$a_4b_2$	$a_3b_2$	$a_2b_2$	$a_1b_2$	$a_0b_2$				
	$a_4b_3$	$a_3b_3$	$a_2b_3$	$a_1b_3$	$a_0b_3$					
$+$	$a_4b_4$	$a_3b_4$	$a_2b_4$	$a_1b_4$	$a_0b_4$					
$p_9$	$p_8$	$p_7$	$p_6$	$p_5$	$p_4$	$p_3$	$p_2$	$p_1$	$p_0$	

$$A = a_{m-1} \dots a_1 a_0 \quad b_{n-1} \dots b_1 b_0 = B$$



$a_i b_j$  称为位积。 $m \times n$  个位积可以通过  $m \times n$  个与门并行产生。

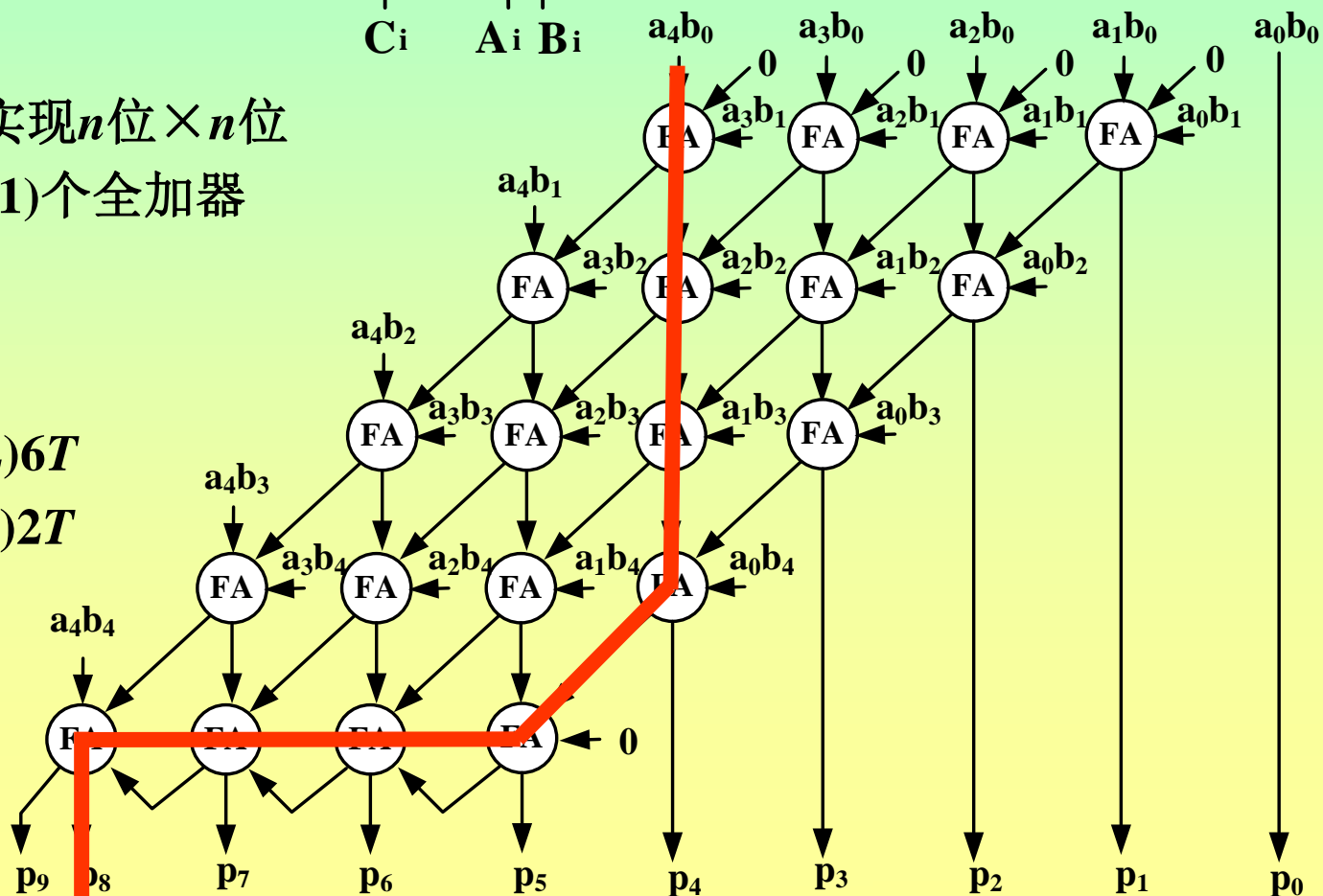
# 阵列乘法器



这种乘法器要实现 $n$ 位 $\times n$ 位乘法时，需要 $n(n-1)$ 个全加器和 $n^2$ 个与门。

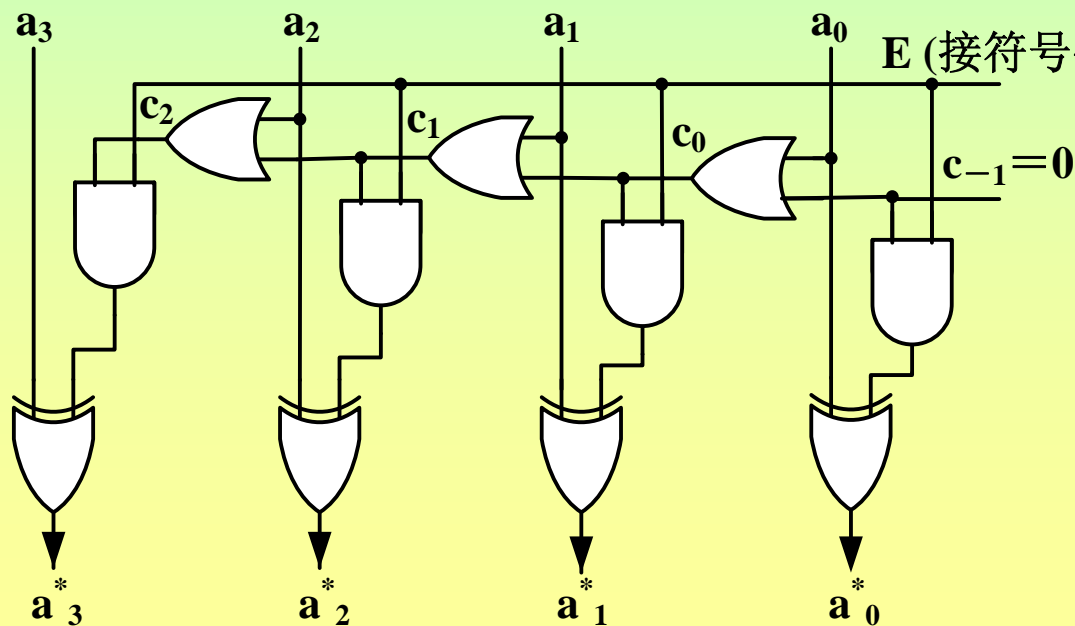
所需时间为：

$$\begin{aligned}
 t_m &= T + (n-2)6T \\
 &\quad + 5T + (n-2)2T \\
 &\quad + 3T \\
 &= (8n-7)T
 \end{aligned}$$



# 对2求补电路

对2求补时，采用自右向左按位扫描技术。设 $A = a_n \dots a_1 a_0$ 是 $(n+1)$ 位带符号数的原码， $a_n$ 为符号位，要求确定它的补码。求补的方法是从数的最右端 $a_0$ 开始，从右向左扫描，直到找出第一个“1”。例如 $a_i = 1$ ， $a_i$ 左侧的每一个数值位都取反， $a_i$ 和 $a_i$ 右侧的位保持不变。当控制信号线E为“1”时，启动对2求补的操作；当控制信号线E为“0”时，输出和输入相等。可以利用符号位来作为控制信号。



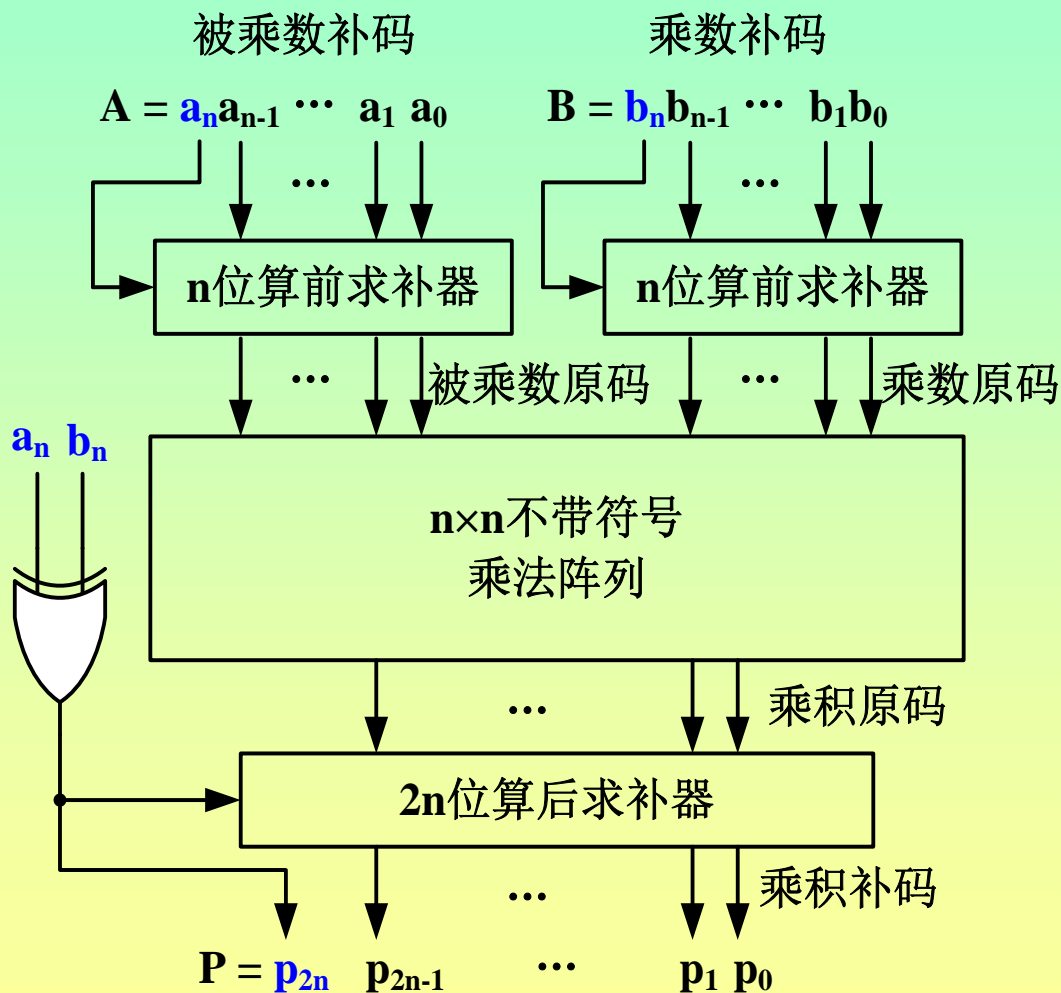
$a_n^*$  的延迟时间为:

$$t_{TC} = n \times T + T + 3T \\ = (n+4)T$$

若输入为带符号数的补码，则输出为该数的原码，即

$$\underline{[ [A]_{\text{补}} ]_{\text{补}} = [A]_{\text{原}}}$$

### 3.带符号的阵列乘法器



算前求补器的作用是将乘数和被乘数补码变成原码；算后求补器的作用是将乘法阵列输出的乘积原码变成补码。

例：  $x = -15$ ，  $y = -13$ ， 用带求补器的阵列乘法器求  $xy$ 。

$$[x]_{\text{补}} = 10001, [y]_{\text{补}} = 10011$$

算前求补器输出为：

$$|x| = 1111, |y| = 1101$$

阵列乘法器输出为：

$$11000011$$

乘积的符号位为0，算后求补器输出为：

$$11000011$$

乘积的补码为：

$$[xy]_{\text{补}} = 011000011$$

# 思考

- 1、“求补”与求一个数补码有何异同之处？
- 2、何时用“求补”运算，何时用“求补码”？



# 补码与真值的关系

考虑一个定点补码整数  $[N]_{\text{补}} = a_n a_{n-1} \dots a_1 a_0$ ，其中  $a_n$  是符号位。  $[N]_{\text{补}}$  和真值  $N$  的关系可以表示成：

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

以上两式可合并成下列形式：

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

因此可以认为符号位  $a_n$  具有负的权值  $(-2^n)$ ，而数值位  $a_i$  具有正的权值  $(+2^i)$ 。

另外， $-N$  的值可以用下式计算：

$$[-N]_{\text{补}} = \overline{a_n} \overline{a_{n-1}} \Lambda \overline{a_1} \overline{a_0} + 1$$

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

# 一般化的全加器

通过把正权或负权加到全加器的输入/输出端，可以归纳出四类加法器。每一类加法

C为1表示“2”，S为1表示“ $-1$ ”，因此结果为“1”

个数命名。  
其真值表为：

类型	逻辑符号	操作
0类 全加器		$\begin{array}{r} X \\ Y \\ +) Z \\ \hline CS \end{array}$
1类 全加器		$\begin{array}{r} X \\ Y \\ +) -Z \\ \hline C(-S) \end{array}$
2类 全加器 (全减器)		$\begin{array}{r} -X \\ -Y \\ +) Z \\ \hline (-C)S \end{array}$
3类 全加器		$\begin{array}{r} -X \\ -Y \\ +) -Z \\ \hline (-C)(-S) \end{array}$

X(+)	Y(+)	X(-)	S(-)	C(+)
0	0		0	0
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	1
1	1	1	1	1

对0类、3类全加器：

$$S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ$$

$$C = XY + YZ + XZ$$

对1类、2类全加器：

$$S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ$$

$$C = XY + Y\bar{Z} + X\bar{Z}$$

# 直接补码阵列乘法器

设被乘数A和乘数B是两个5位的二进制补码数，即

$$A=(a_4)a_3a_2a_1a_0, \quad B=(b_4)b_3b_2b_1b_0$$

它们具有带负权的符号位 $a_4$ 和 $b_4$ （用括号标注）。将符号位一起参加运算，并用括号标注具有负权的位积，则A和B相乘的计算过程如下：

$$\begin{array}{r} \times \quad (a_4) \quad a_3 \quad a_2 \quad a_1 \quad a_0 \\ \quad (b_4) \quad b_3 \quad b_2 \quad b_1 \quad b_0 \end{array}$$

$$\hline (a_4b_0) \quad a_3b_0 \quad a_2b_0 \quad a_1b_0 \quad a_0b_0$$

$$(a_4b_1) \quad a_3b_1 \quad a_2b_1 \quad a_1b_1 \quad a_0b_1$$

$$(a_4b_2) \quad a_3b_2 \quad a_2b_2 \quad a_1b_2 \quad a_0b_2$$

$$(a_4b_3) \quad a_3b_3 \quad a_2b_3$$

$$+ \quad a_4b_4 \quad (a_3b_4) \quad (a_2b_4) \quad (a_1b_4)$$

$$\hline p_9 \quad p_8 \quad p_7 \quad p_6 \quad p_5 \quad p_4 \quad p_3 \quad p_2 \quad p_1 \quad p_0$$

具有负权的进位  
(实际上就是借位)

$$[xy]_{\text{补}} = (1)00111101$$

$$xy = -195$$

例:  $x=15, y=-13$

$$[x]_{\text{补}} = 0 \ 1111$$

$$[y]_{\text{补}} = (1)0011$$

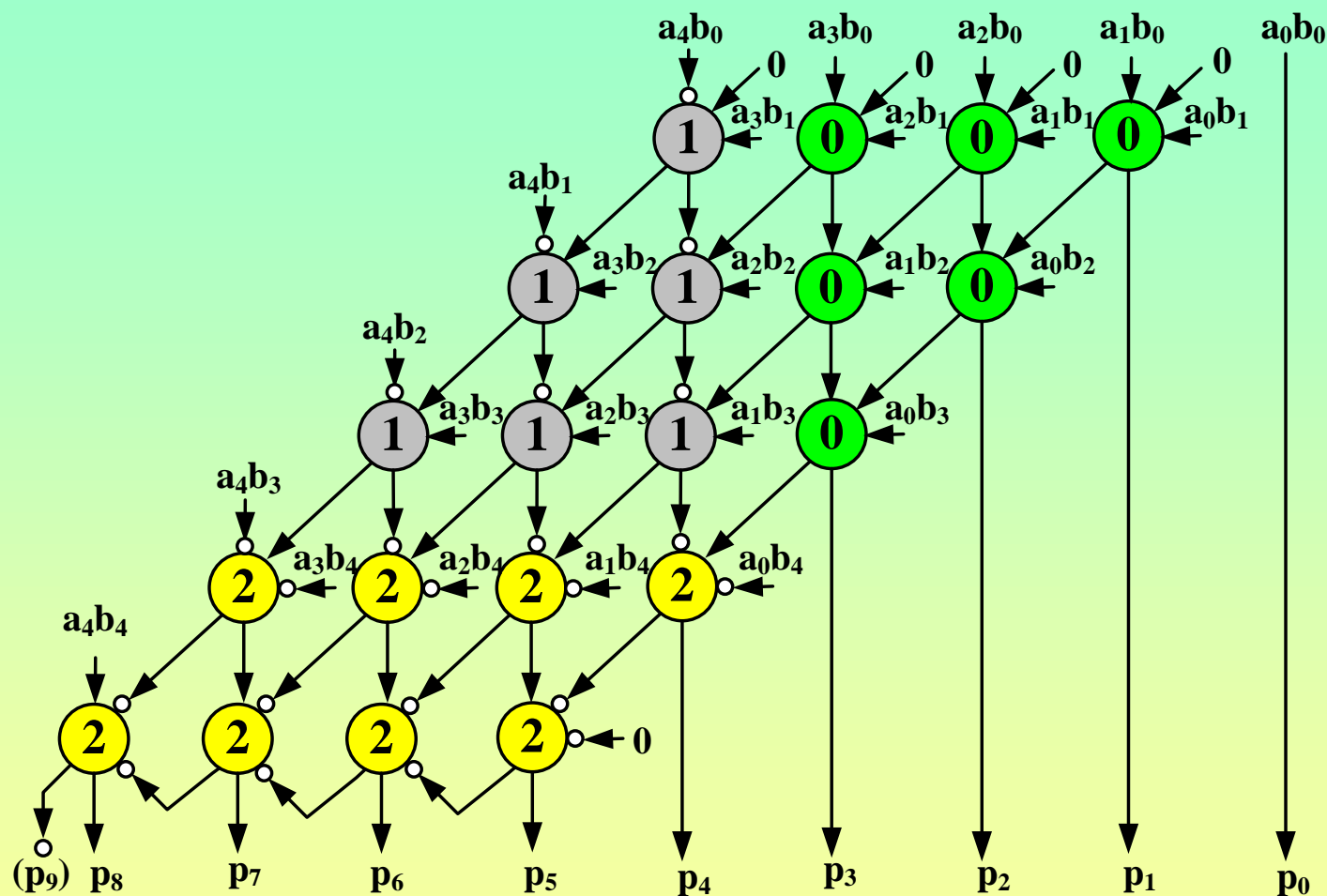
$$\begin{array}{r} \quad \quad \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \\ \times \quad (1) \quad 0 \quad 0 \quad 1 \quad 1 \\ \hline \quad \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \end{array}$$

$$\begin{array}{r} 0 \quad 1 \quad 1 \quad 1 \quad 1 \\ 0 \quad 0 \quad 0 \quad 0 \quad 0 \end{array}$$

$$\begin{array}{r} + \quad 0 \quad (1) \quad (1) \quad (1) \quad (1) \\ \quad \quad (1) \quad (1) \quad (1) \quad 1 \quad 1 \quad 1 \\ \hline (1) \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \end{array}$$

进位

利用一般化的全加器可以构成直接补码阵列乘法器：



符号位

右上角的三角形用0类全加器，左上角的三角形用1类全加器，最后两行用2类全加器。结果的最高位 $p_9$ 具有负权，所以结果为补码。

作业

P61 7 (1)

## 原码除法

$$[\mathbf{x}]_{\text{原}} = \mathbf{x}_f \cdot \mathbf{x}_{n-1} \cdots \mathbf{x}_1 \mathbf{x}_0, \quad [\mathbf{y}]_{\text{原}} = \mathbf{y}_f \cdot \mathbf{y}_{n-1} \cdots \mathbf{y}_1 \mathbf{y}_0$$

则商 $q = x / y$ 的原码为:

$$[\mathbf{q}]_{\text{原}} = (\mathbf{x}_f \oplus \mathbf{y}_f) + (0.\mathbf{x}_{n-1} \dots \mathbf{x}_1 \mathbf{x}_0 / 0.\mathbf{y}_{n-1} \dots \mathbf{y}_1 \mathbf{y}_0)$$

例：被除数 $x=0.1001$ ，除数 $y=0.1011$ 。手工计算 $x / y$ 的过程如下：

[illegible]

**$x < y$ , 商0**

得余数 $r_0$  (即 $x$ )

除数右移一位，余数减除数，商1

得余数 $r_1$

除数右移一位，余数减除数，商1

得余数 $r_2$

除数右移一位，余数不减除数，商0

得余数 $r_3$

除数右移一位，余数减除数，商1

得余数 $r_4$

# 恢复余数法

被除数 $x=0.1001$ ，除数 $y=0.1011$   $[-y]_{\text{补}}=1.0101$

为保证余数左移时符号位不变，应采用双符号位

对于上算法修改如下

(1) 将心算比较余数和除数大小改为减法比较，并且减 $y$ 改为加 $[-y]_{\text{补}}$ 。余数减除数大于等于0则商1，小于0则商0。

(2) 将除数右移改为余数左移。

如果余数减除数的差小于0，应将差加上除数，恢复原来的余数。这就是恢复余数法。

	0 0.	1 0 0 1	商
$[-y]_{\text{补}}$	1 1.	0 1 0 1	
	1 1.	1 1 1 0	0
	+ 0 0.	1 0 1 1	
	0 0.	1 0 0 1	
	← 0 1.	0 0 1 0	
$+ [-y]_{\text{补}}$	1 1.	0 1 0 1	
	0 0.	0 1 1 1	0.1
	← 0 0.	1 1 1 0	
$+ [-y]_{\text{补}}$	1 1.	0 1 0 1	
	0 0.	0 0 1 1	0.11
	← 0 0.	0 1 1 0	
$+ [-y]_{\text{补}}$	1 1.	0 1 0 1	
	1 1.	1 0 1 1	0.110
	+ 0 0.	1 0 1 1	
	0 0.	0 1 1 0	
	← 0 0.	1 1 0 0	
$+ [-y]_{\text{补}}$	1 1.	0 1 0 1	
	0 0.	0 0 0 1	0.1101

说明

$x-y$

余数 $r_0 < 0$ ，商0  
加 $y$ 恢复余数

余数左移一位

减 $y$ 比较

余数 $r_1 > 0$ ，商1

余数左移一位

减 $y$ 比较

余数 $r_2 > 0$ ，商1

余数左移一位

减 $y$ 比较

余数 $r_3 < 0$ ，商0

加 $y$ 恢复余数

余数左移一位

减 $y$ 比较

余数 $r_4 > 0$ ，商1

# 加减交替法

恢复余数法中，设某次余数为 $r_i$ ，要继续进行下面的求商运算，需要将 $r_i$ 左移一位，然后减去除数，进行比较：

$$2r_i - y$$

结果小于0时商上0，并加 $y$ 恢复余数：

$$(2r_i - y) + y = 2r_i$$

继续下面的求商，又要将它左移一位，再减去除数

$$2(2r_i - y) - y = 4r_i - y$$

当 $(2r_i - y)$ 小于0时，商仍上0，但不进行加 $y$ 恢复余数的操作，而是将 $(2r_i - y)$ 左移一位，然后加上除数

$$2(2r_i - y) + y = 4r_i - y$$

也得到同样的余数 $(4r_i - y)$ 。所以，当比较结果小于0时，仍将结果左移一位，然后加上除数 $y$ 。这就是不恢复余数法，也称加减交替法。

加减交替法的运算规则是：

余数为正时，商上1，余数左移一位，再减去除数，得到新的余数；

余数为负时，商上0，余数左移一位，再加上除数，得到新的余数。



	0 0. 1 0 0 1	商	说明
$+[-y]_{\text{补}}$	1 1. 0 1 0 1		$x-y$
	1 1. 1 1 1 0	0	余数 $r_0 < 0$ , 商0
	1 1. 1 1 0 0		左移一位
$+y$	0 0. 1 0 1 1		余数为负, 加y
	0 0. 0 1 1 1	0.1	余数 $r_1 > 0$ , 商1
	0 0. 1 1 1 0		左移一位
$+[-y]_{\text{补}}$	1 1. 0 1 0 1		余数为正, 减y
	0 0. 0 0 1 1	0.11	余数 $r_2 > 0$ , 商1
	0 0. 0 1 1 0		左移一位
$+[-y]_{\text{补}}$	1 1. 0 1 0 1		余数为正, 减y
	1 1. 1 0 1 1	0.110	余数 $r_3 < 0$ , 商0
	1 1. 0 1 1 0		左移一位
$+y$	0 0. 1 0 1 1		余数为负, 加y
	0 0. 0 0 0 1	0.1101	余数 $r_4 > 0$ , 商1

余数:

**注意:** 得到的余数是经过左移4次后的结果  
故余数为  $0.0001 \times 2^{-4}$

## 恢复余数法的运算规则是：

余数为正时，商上1，余数左移一位，再减去除数，得到新的余数；  
余数为负时，商上0，再加上除数（恢复余数），余数左移一位，再  
减去除数，得到新的余数。

## 加减交替法的运算规则是：

余数为正时，商上1，余数左移一位，再减去除数，得到新的余数；  
余数为负时，商上0，余数左移一位，再加上除数，得到新的余数。

证明：  $2(R+y) - y = 2R + y$

特点：

- 1、恢复余数除法，最后的操作为减法
- 2、都需要余数左移一位后再进行加或减

为保证余数左移时符号位不变，应采用双符号位

双符号位最高符号位永远表示结果的正确符号；

若用单符号位，余数左移后将可能改变符号位的值

余数永远不会溢出？

原因何在？

设某次运算的余数为R，  
不论是何种运算方法，

若R为正，下次的余数为  
 $2R - y = R + R - y$

若R为负，下次的余数为  
 $2R + y$  都不会溢出

余数的绝对值小于除数的绝对值

		0 0. 1 0 0 1	商
$+[-y]_{补}$		1 1. 0 1 0 1	
		1. 1 1 1 0	0
+		0. 1 0 1 1	
		0. 1 0 0 1	
		1. 0 0 1 0	
$+[-y]_{补}$		1. 0 1 0 1	
		0. 0 1 1 1	0. 1
		0. 1 1 1 0	
$+[-y]_{补}$		1. 0 1 0 1	
		0. 0 0 1 1	0. 1 1
		0. 0 1 1 0	
$+[-y]_{补}$		1. 0 1 0 1	
		1. 1 0 1 1	0. 1 1 0
+		0. 1 0 1 1	
		0. 0 1 1 0	
		0. 1 1 0 0	
$+[-y]_{补}$		1. 0 1 0 1	
		0 0. 0 0 0 1	0. 1 1 0 1

说明

$x - y$

余数 $r_0 < 0$ ，商0  
加y恢复余数

余数左移一位  
减y比较

余数 $r_1 > 0$ ，商1  
余数左移一位  
减y比较

余数 $r_2 > 0$ ，商1  
余数左移一位  
减y比较

余数 $r_3 < 0$ ，商0  
加y恢复余数

余数左移一位  
减y比较

余数 $r_4 > 0$ ，商1

# 可控加法/减法单元

可控加法/减法单元（CAS）由一个全加器和一个异或门构成，有4个输入端和4个输出端。当控制端P为0时，实现全加器：

$$S_i = A_i \oplus B_i \oplus C_i$$

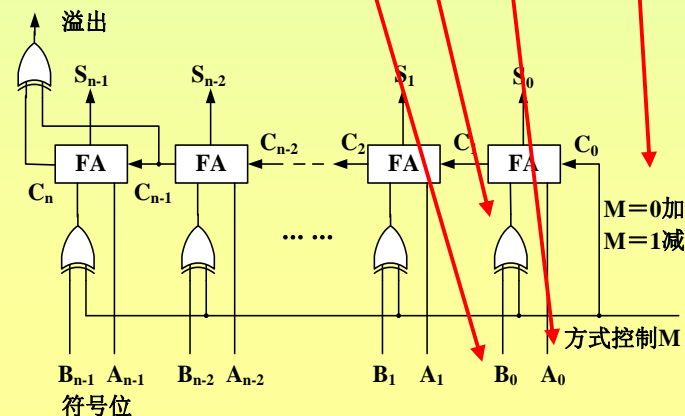
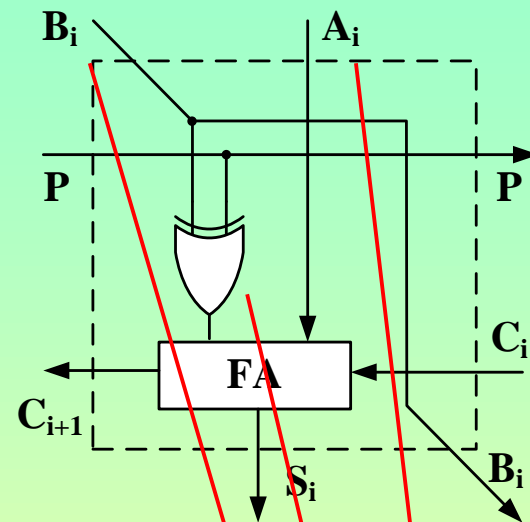
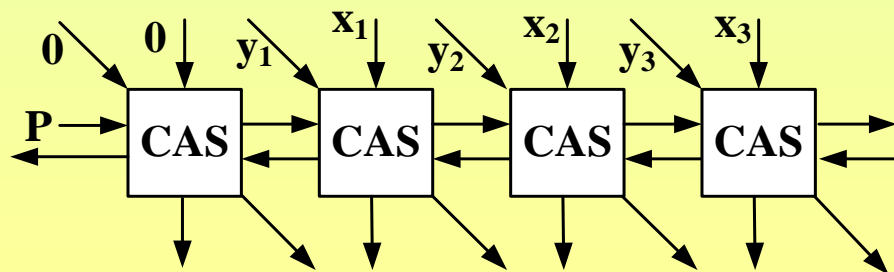
$$C_{i+1} = A_i B_i + B_i C_i + A_i C_i$$

控制端为1时，用于实现补码减法：

$$S_i = A_i \oplus \bar{B}_i \oplus C_i$$

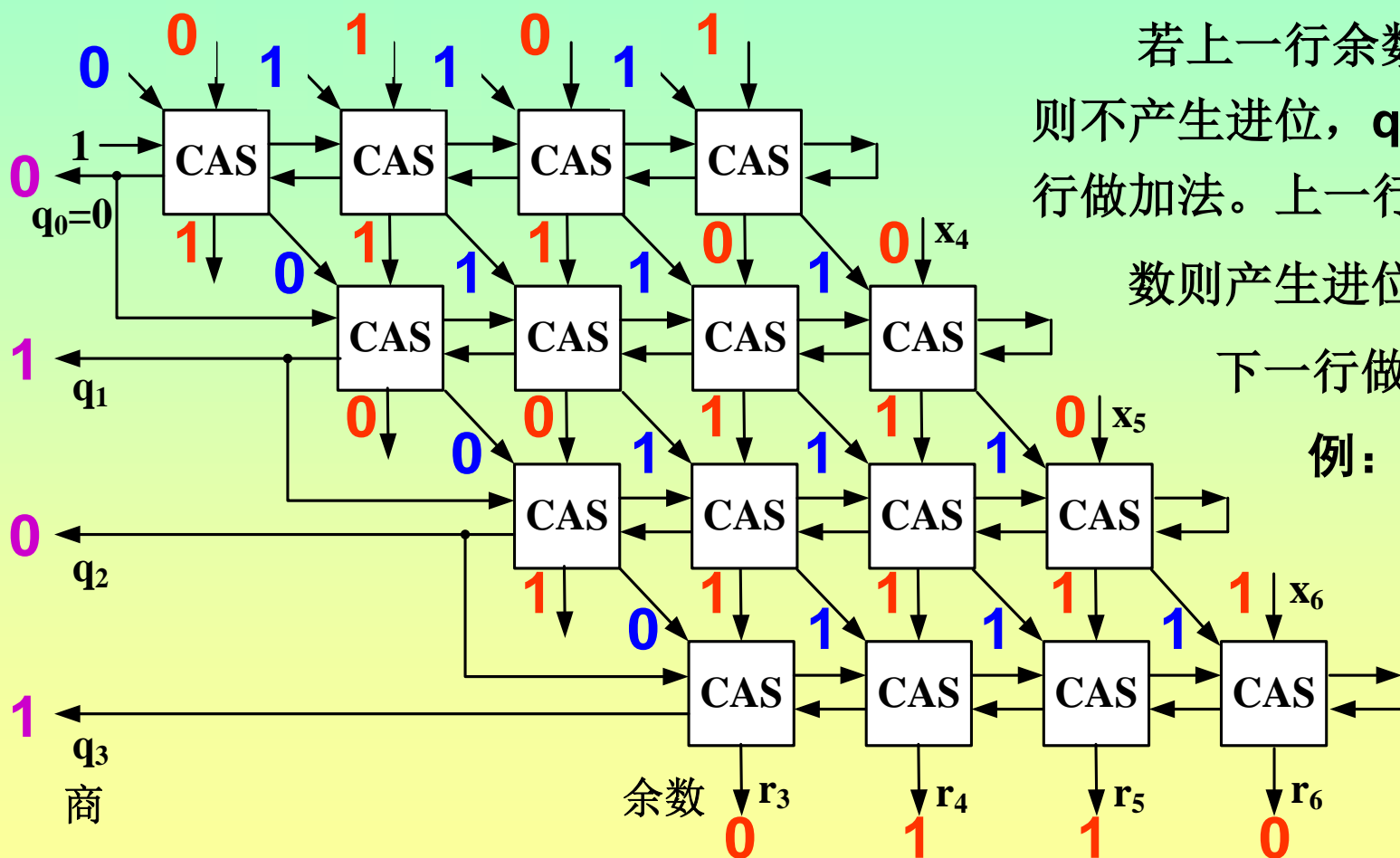
$$C_{i+1} = A_i \bar{B}_i + \bar{B}_i C_i + A_i C_i$$

设 $[x]_{\text{补}} = 0.x_1x_2x_3$ ， $[y]_{\text{补}} = 0.y_1y_2y_3$ 。下图电路当 $P=0$ 时实现 $x+y$ ， $P=1$ 时实现 $x-y$ 。减法是通过加上 $[-y]_{\text{补}}$ 实现的。



# 阵列除法器

利用CAS构成的不恢复余数阵列除法器如下。被除数 $x=0.x_1x_2x_3x_4x_5x_6$ ，除数 $y=0.y_1y_2y_3$ ，均为正数，且 $x < y$ （否则商大于1，溢出）。



若上一行余数小于除数  
则不产生进位， $q_i=0$ ，下一  
行做加法。上一行余数大于除  
数则产生进位， $q_i=1$ ，  
下一行做减法。

例：  $x=0.101001$   
 $y=0.111$

2n位除以n位的阵列除法器，有单元 $(n+1)^2$ 个，延迟时间为 $t_d=(n+1)^2 T_{CAS}$ 。

# 阵列除法器

上一行余数小于除数则不产生进位， $q_i=0$ ，下一行做加法。

“不够减”，不产生进位，例  $3 - 5$ ，转化为  $3 + (-5)$ 补

上一行余数大于除数则产生进位， $q_i=1$ ，下一行做减法。

“够减”，产生进位，例  $5 - 3$ ，转化为  $5 + (-3)$ 补

作业

P61 8 (2)

用恢复余数与加减交替法分别计算



# 逻辑运算

## 1. 逻辑非运算

设

$$x = x_0 x_1 x_2 \dots x_n$$

$$z = \overline{x} = z_0 z_1 z_2 \dots z_n$$

则

$$z_i = \overline{x_i} \quad (i=0,1,2,\dots,n)$$

## 2. 逻辑加运算

设

$$x = x_0 x_1 x_2 \dots x_n$$

$$y = y_0 y_1 y_2 \dots y_n$$

$$z = x \vee y = z_0 z_1 z_2 \dots z_n$$

则

$$z_i = x_i \vee y_i \quad (i=0,1,2,\dots,n) \quad \text{用符号} + \text{或者} \vee \text{来表示}$$

## 3. 逻辑乘运算

设

$$x = x_0 x_1 x_2 \dots x_n$$

$$y = y_0 y_1 y_2 \dots y_n$$

$$z = x \wedge y = z_0 z_1 z_2 \dots z_n$$

则

$$z_i = x_i \wedge y_i \quad (i=0,1,2,\dots,n) \quad \text{用符号} + \text{或者} \cdot \text{来表示}$$

## 4. 逻辑异运算

设

$$x = x_0 x_1 x_2 \dots x_n$$

$$y = y_0 y_1 y_2 \dots y_n$$

$$z = x \oplus y = z_0 z_1 z_2 \dots z_n$$

则

$$z_i = x_i \oplus y_i \quad (i=0,1,2,\dots,n) \quad \text{用符号} \oplus \text{来表示}$$

# 多功能算术逻辑单元

算术逻辑运算单元(Arithmetic Logic Unit, ALU)不仅具有多种算术运算和逻辑运算的功能，而且具有先行进位逻辑，从而能实现高速运算。

全加器的逻辑表达式为：

$$F_i = A_i \oplus B_i \oplus C_i$$

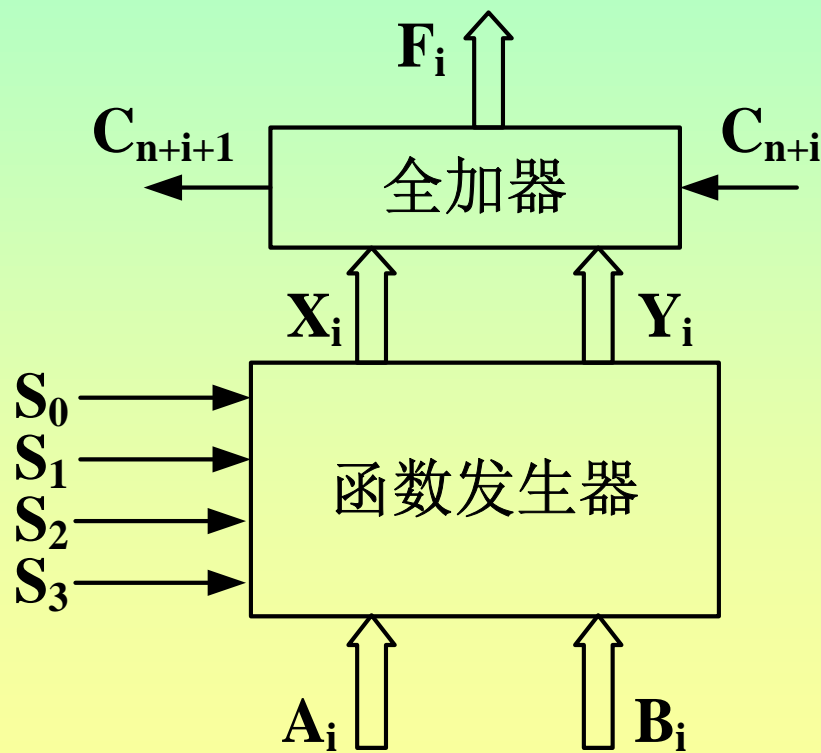
$$C_{i+1} = A_i B_i + B_i C_i + A_i C_i$$

为了对全加器进行功能扩展以完成多种算术和逻辑运算，将 $A_i$ 和 $B_i$ 送到由 $S_0, S_1, S_2, S_3$ 控制的函数发生器内，产生组合函数 $X_i$ 和 $Y_i$ ，再送到全加器内进行全加：

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

$$C_{n+i+1} = X_i Y_i + X_i C_{n+i} + Y_i C_{n+i}$$

进位下标用 $n+i$ 代替 $i$ ， $n$ 表示多片级联时每片的进位输入。



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

$X_i$ 和 $Y_i$ 的逻辑表达式为:

$$X_i = \overline{S_2}\overline{S_3} + \overline{S_2}S_3(\overline{A_i} + \overline{B_i}) + S_2\overline{S_3}(\overline{A_i} + B_i) + S_2S_3\overline{A_i}$$

$$Y_i = \overline{S_0}\overline{S_1}\overline{A_i} + \overline{S_0}S_1\overline{A_i}B_i + S_0S_1\overline{A_i}\overline{B_i}$$

$$X_i = \overline{S_3A_iB_i + S_2A_i\overline{B_i}}$$

$$Y_i = \overline{A_i + S_0B_i + S_1\overline{B_i}}$$

另外还有:

$$X_i + Y_i = X_i$$

$$X_i Y_i = Y_i$$

因此全加器的进位信号可以写成:

$$C_{n+i+1} = X_i Y_i + (X_i + Y_i)C_{n+i} = Y_i + X_i C_{n+i}$$

4位之间采用先行进位逻辑，每一位的进位公式递推如下：

$$C_{n+1} = Y_0 + X_0 C_n$$

$$C_{n+2} = Y_1 + X_1 C_{n+1} = Y_1 + X_1 Y_0 + X_1 X_0 C_n$$

$$C_{n+3} = Y_2 + X_2 C_{n+2} = Y_2 + X_2 Y_1 + X_2 X_1 Y_0 + X_2 X_1 X_0 C_n$$

$$C_{n+4} = Y_3 + X_3 C_{n+3} = Y_3 + X_3 Y_2 + X_3 X_2 Y_1 + X_3 X_2 X_1 Y_0 + X_3 X_2 X_1 X_0 C_n$$

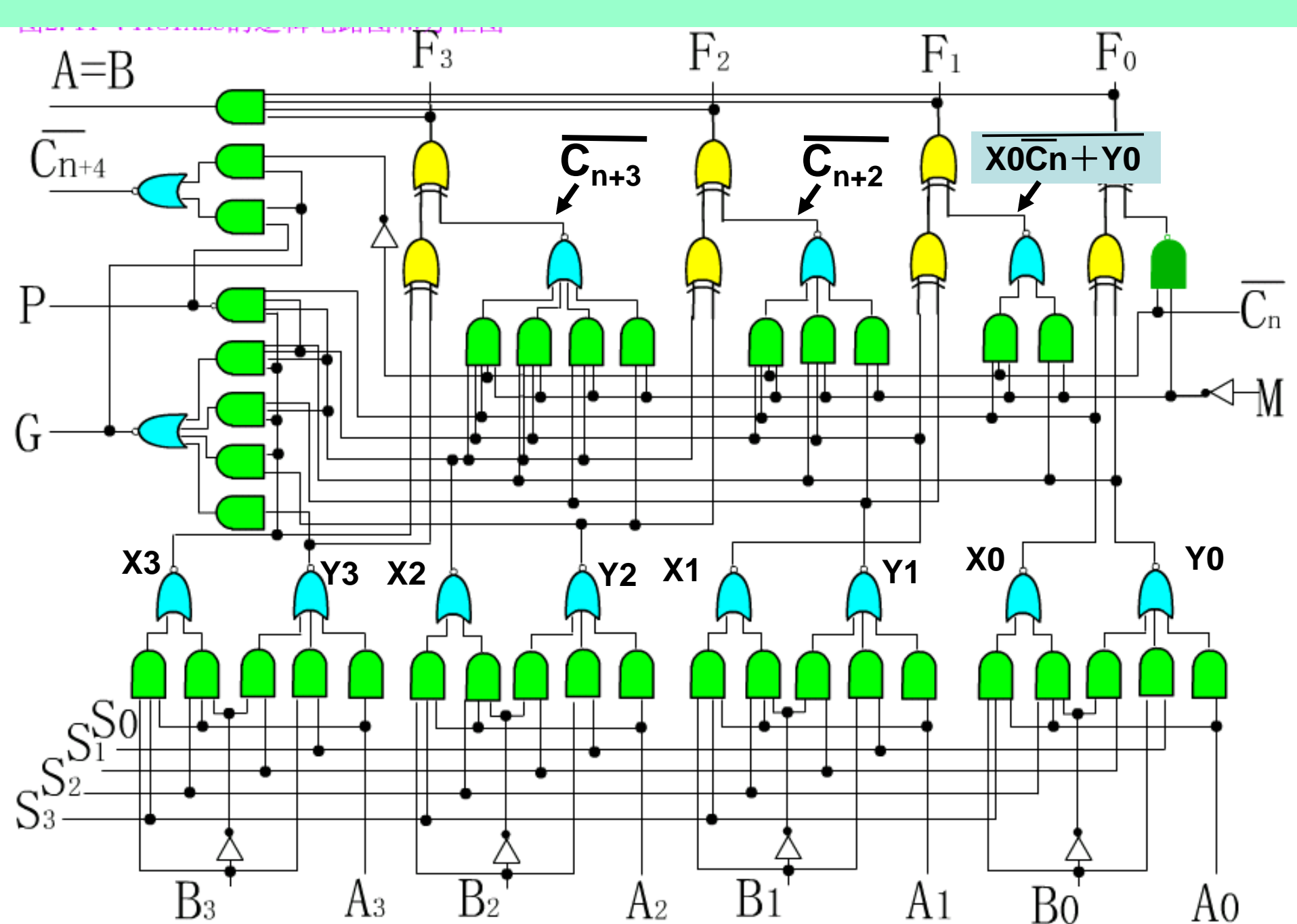
定义

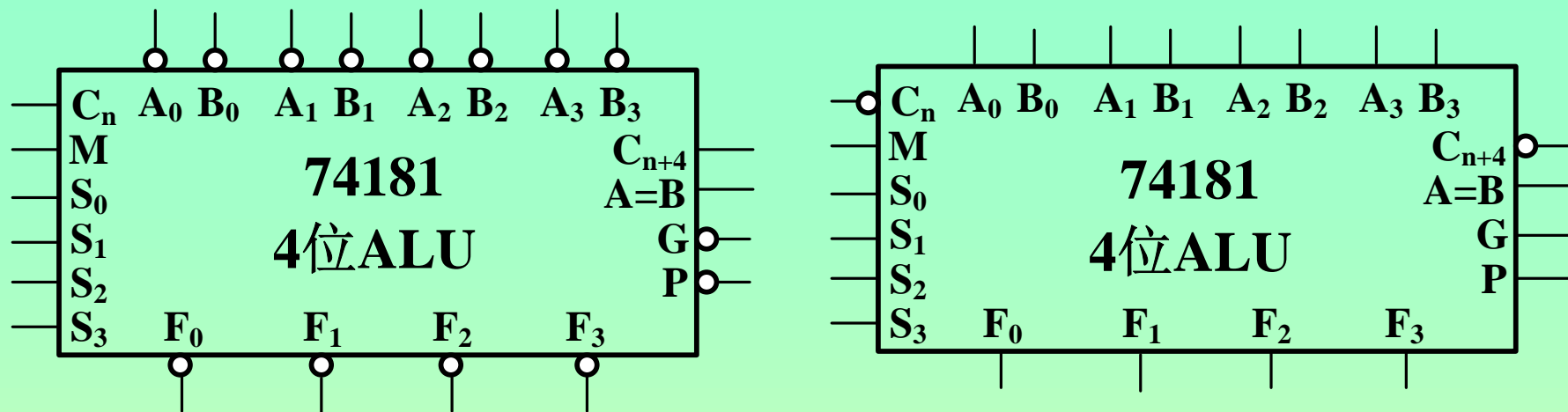
$$G = Y_3 + X_3 Y_2 + X_3 X_2 Y_1 + X_3 X_2 X_1 Y_0$$

$$P = X_3 X_2 X_1 X_0$$

**G**称为进位产生输出，**P**称为进位传递输出，它们和**CLA**电路配合使用，可以实现多级先行进位。

**74181**是4位算术逻辑单元。它除了有**S<sub>0</sub>~S<sub>3</sub>**四个控制端外，还有一个控制端**M**，用来控制**74181**是进行算术运算还是进行逻辑运算。**74181**既适用于正逻辑数的运算，也适用于负逻辑数的运算。还有一个输出端“**A=B**”可用于检测两数相等。





负逻辑和正逻辑方式下74181的方框图

以正逻辑为例。当 $M=H$ （高电平）时，各位的输出表达式为：

$$F_i = X_i \oplus Y_i \oplus 1 \quad i = 0, 1, 2, 3$$

此时 $F_i$ 只和 $X_i$ 及 $Y_i$ 有关，与进位信号无关，实现逻辑运算。

当 $M=L$ （低电平）时，各位的输出表达式为：

$$F_i = X_i \oplus Y_i \oplus \overline{C_{n+i}}$$

$$= \overline{X_i} \oplus \overline{Y_i} \oplus \overline{C_{n+i}} \quad i = 0, 1, 2, 3$$

而 $\overline{X_i}$ 、 $\overline{Y_i}$ 和 $\overline{C_{n+i}}$ 相加产生的进位刚好为 $\overline{C_{n+i+1}}$ ，此时74181实现算术运算：

$C_n=H$ 无进位输入，74181实现  $\overline{X_3}\overline{X_2}\overline{X_1}\overline{X_0}$ 加  $\overline{Y_3}\overline{Y_2}\overline{Y_1}\overline{Y_0}$ ；

$C_n=L$ 有进位输入，74181实现  $\overline{X_3}\overline{X_2}\overline{X_1}\overline{X_0}$ 加  $\overline{Y_3}\overline{Y_2}\overline{Y_1}\overline{Y_0}$ 加1。

以正逻辑为例。当 $M=H$ （高电平）时，各位的输出表达式为：

$$F_i = X_i \oplus Y_i \oplus 1 \quad i = 0,1,2,3$$


当 $M=L$ （低电平）时，各位的输出表达式为：

**74181**实现算术运算：

$C_n=H$ 无进位输入，**74181**实现  $\overline{X_3}\overline{X_2}\overline{X_1}\overline{X_0}$ 加  $\overline{Y_3}\overline{Y_2}\overline{Y_1}\overline{Y_0}$ ；

$C_n=L$ 有进位输入，**74181**实现  $\overline{X_3}\overline{X_2}\overline{X_1}\overline{X_0}$ 加  $\overline{Y_3}\overline{Y_2}\overline{Y_1}\overline{Y_0}$ 加1。

$$F_i = X_i \oplus Y_i \oplus \overline{C_{n+i}}$$


$$= \overline{X_i} \oplus \overline{Y_i} \oplus \overline{C_{n+i}}$$

$$i = 0, 1, 2, 3$$



全加器的逻辑表达式为：

$$F_i = A_i \oplus B_i \oplus C_i$$

$$C_{i+1} = A_i B_i + B_i C_i + A_i C_i$$

$$F_0 = A_0 \oplus B_0 \oplus C_0$$

$$F_1 = A_1 \oplus B_1 \oplus C_1$$

$$F_2 = A_2 \oplus B_2 \oplus C_2$$

$$F_3 = A_3 \oplus B_3 \oplus C_3$$

$$C_1 = A_0 B_0 + B_0 C_0 + A_0 C_0$$

$$C_2 = A_1 B_1 + B_1 C_1 + A_1 C_1$$

$$C_3 = A_2 B_2 + B_2 C_2 + A_2 C_2$$

每一级需要的进位信号为上一级的加数两两相乘之和

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

$$F_0 = X_0 \oplus Y_0 \oplus C_n$$

$$F_1 = X_1 \oplus Y_1 \oplus C_{n+1}$$

$$F_2 = X_2 \oplus Y_2 \oplus C_{n+2}$$

$$F_3 = X_3 \oplus Y_3 \oplus C_{n+3}$$

$$\begin{aligned} C_{n+1} &= X_0 Y_0 + Y_0 C_n + X_0 C_n \\ C_{n+2} &= X_1 Y_1 + Y_1 C_{n+1} + X_1 C_{n+1} \\ C_{n+3} &= X_2 Y_2 + Y_2 C_{n+2} + X_2 C_{n+2} \end{aligned}$$

$$F_i = \bar{X}_i \oplus \bar{Y}_i \oplus C_{n+i}$$

$$F_0 = \bar{X}_0 \oplus \bar{Y}_0 \oplus C_n$$

$$F_1 = \bar{X}_1 \oplus \bar{Y}_1 \oplus C_{n+1}$$

$$F_2 = \bar{X}_2 \oplus \bar{Y}_2 \oplus C_{n+2}$$

$$F_3 = \bar{X}_3 \oplus \bar{Y}_3 \oplus C_{n+3}$$

$$\begin{aligned} C_{n+1} &= \bar{X}_0 \bar{Y}_0 + Y_0 C_n + \bar{X}_0 C_n \\ C_{n+2} &= \bar{X}_1 \bar{Y}_1 + Y_1 C_{n+1} + \bar{X}_1 C_{n+1} \\ C_{n+3} &= \bar{X}_2 \bar{Y}_2 + Y_2 C_{n+2} + \bar{X}_2 C_{n+2} \end{aligned}$$

$\overline{X}_i$ 、 $\overline{Y}_i$ 和 $\overline{C_{n+i}}$ 相加产生的进位为：

$$\overline{X}_i \overline{Y}_i + \overline{X}_i \overline{C_{n+i}} + \overline{Y}_i \overline{C_{n+i}} =$$

$$\overline{X_i + Y_i + X_i C_{n+i} + Y_i C_{n+i}} =$$

$$\overline{X_i} + \overline{X_i C_{n+i}} + \overline{Y_i C_{n+i}} = \overline{X_i} + \overline{Y_i C_{n+i}}$$

$$\begin{aligned} \text{而 } \overline{C_{n+i+1}} &= \overline{Y_i + X_i C_{n+i}} \\ &= \overline{Y_i} \cdot \overline{X_i C_{n+i}} \\ &= \overline{Y_i} \cdot (\overline{X_i} + \overline{C_{n+i}}) \\ &= \overline{Y_i} \overline{X_i} + \overline{Y_i} \overline{C_{n+i}} \\ &= \overline{Y_i + X_i} + \overline{Y_i C_{n+i}} \\ &= \overline{X_i} + \overline{Y_i C_{n+i}} \end{aligned}$$

故 $\overline{X}_i$ 、 $\overline{Y}_i$ 和 $\overline{C_{n+i}}$ 相加产生的进位刚好为 $\overline{C_{n+i+1}}$

例：正逻辑方式下， $S_3S_2S_1S_0=LHLL$ ， $M=H$ ，74181实现何种运算？

$M=H$ 时，输出表达式为：

$$F_i = X_i \oplus Y_i \oplus 1$$

$S_3S_2S_1S_0=LHLL$ 时， $X_i = \overline{A_i} + B_i$ ， $Y_i = \overline{A_i}$ ，所以

$$F_i = (\overline{A_i} + B_i) \oplus \overline{A_i} \oplus 1 = \overline{A_i} \overline{B_i}$$

故此时74181实现逻辑运算 $\overline{AB}$ 。如果把输出表达式中“与”“或”互换、“同或”“异或”互换、“0”“1”互换（即求对偶），得到的就是负逻辑方式下电路实现的逻辑运算，所以此时电路在负逻辑方式下实现的逻辑运算为 $A+B$ 。

例：正逻辑方式下， $S_3S_2S_1S_0=HLLL$ ， $M=L$ ，74181实现何种运算？

$M=L$ 时，电路实现算术运算 $\overline{X_i}$ 加 $\overline{Y_i}$ 。

$S_3S_2S_1S_0=HLLL$ 时， $X_i = \overline{A_i} + \overline{B_i}$ ， $Y_i = \overline{A_i}$ ，所以

$C_n=H$ 时，电路实现 $AB$ 加 $A$ ； $C_n=L$ 时，电路实现 $AB$ 加 $A$ 加1。

正逻辑下的全加运算在负逻辑下仍是全加运算。负逻辑方式下的功能为：

$C_n=H$ 时实现 $(A+B)$ 加 $A$ 加1，相当于有进位；

100  $C_n=L$ 时实现 $(A+B)$ 加 $A$ ，相当于无进位。

工作方式 选择输入 $S_3S_2S_1S_0$	负逻辑		正逻辑	
	逻辑运算 $M=H$	算术运算 $M=L, C_n=L$ (无进位)	逻辑运算 $M=H$	算术运算 $M=L, C_n=H$ (无进位)
LLLL	$\overline{A}$	A减1	$\overline{A}$	A
LLLH	$\overline{AB}$	AB减1	$\overline{A+B}$	$A+B$
LLHL	$\overline{A}+B$	$\overline{AB}$ 减1	$\overline{A}B$	$A+\overline{B}$
LLHH	逻辑1	减1	逻辑0	减1 (即1111)
LHLL	$\overline{A+B}$	A加( $A+\overline{B}$ )	$\overline{AB}$	A加 $\overline{AB}$
LHLH	$\overline{B}$	AB加( $A+\overline{B}$ )	$\overline{B}$	( $A+B$ )加 $\overline{AB}$
LHHL	$\overline{A\oplus B}$	A减B减1	$A\oplus B$	A减B减1 (即A加 $\overline{B}$ )
LHHH	$A+\overline{B}$	$A+\overline{B}$	$\overline{AB}$	$\overline{AB}$ 减1
HLLL	$\overline{AB}$	A加( $A+B$ )	$\overline{A}+B$	A加AB
HLLH	$A\oplus B$	A加B	$\overline{A\oplus B}$	A加B
HLHL	B	$\overline{AB}$ 加( $A+B$ )	B	( $A+\overline{B}$ )加AB
HLHH	$A+B$	$A+B$	AB	AB减1
HHLL	逻辑0	A加 $A^*$	逻辑1	A加 $A^*$
HHLH	$\overline{AB}$	AB加A	$A+\overline{B}$	( $A+B$ )加A
HHHL	AB	$\overline{AB}$ 加A	$A+B$	( $A+\overline{B}$ )加A
HHHH	A	A	A	A减1

- 注意区分：

＋ 加  $\oplus$

三个符号分别表示的含义



# 运算器逻辑功能表

运算类型	S3 S2 S1 S0	CN	功 能
逻辑运算	0000	X	$F=A$ (直通)
	0001	X	$F=B$ (直通)
	0010	X	$F=AB$ (FZ)
	0011	X	$F=A+B$ (FZ)
	0100	X	$F=\neg A$ (FZ)
移位运算	0101	X	$F=A$ 不带进位循环右移 B (取低 3 位) 位 (FZ)
	0110	0	$F=A$ 逻辑右移一位 (FZ)
		1	$F=A$ 带进位循环右移一位 (FC, FZ)
	0111	0	$F=A$ 逻辑左移一位 (FZ)
		1	$F=A$ 带进位循环左移一位 (FC, FZ)
算术运算	1000	X	置 $FC=CN$ (FC)
	1001	X	$F=A+B$ (FC, FZ)
	1010	X	$F=A+B+FC$ (FC, FZ)
	1011	X	$F=A-B$ (FC, FZ)
	1100	X	$F=A-1$ (FC, FZ)
	1101	X	$F=A+1$ (FC, FZ)
	1110	X	(保留)
	1111	X	(保留)

表中“X”为任意态



# 两级先行进位ALU

**74182**是一个先行进位形成电路，和**74181**的进位产生输出**G**和进位传递输出**P**配合使用，可以实现两级先行进位。

设4片**74181**的先行进位输出依次为**P<sub>0</sub>, G<sub>0</sub>, P<sub>1</sub>, G<sub>1</sub>, P<sub>2</sub>, G<sub>2</sub>, P<sub>3</sub>, G<sub>3</sub>**，**74182**所提供的进位逻辑关系为：

$$C_{n+x} = G_0 + P_0 C_n$$

$$C_{n+y} = G_1 + P_1 C_{n+x} = G_1 + P_1 G_0 + P_1 P_0 C_n$$

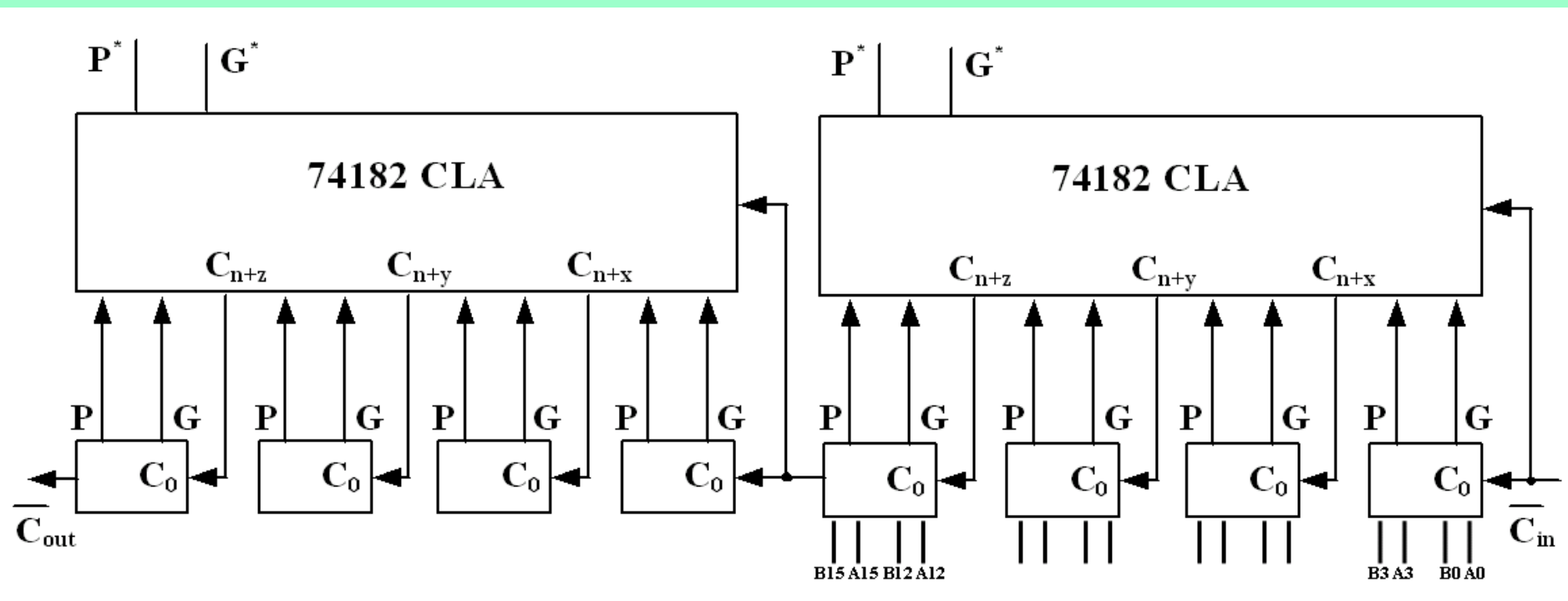
$$C_{n+z} = G_2 + P_2 C_{n+y} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_n$$

此外，**74182**还提供组进位产生输出**G\***和组进位传递输出**P\***，用于实现三级先行进位：

$$P^* = P_3 P_2 P_1 P_0$$

$$G^* = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

$$\begin{aligned} G^* + P^* C_n &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_n \\ &= G_3 + P_3 C_{n+z} \end{aligned}$$

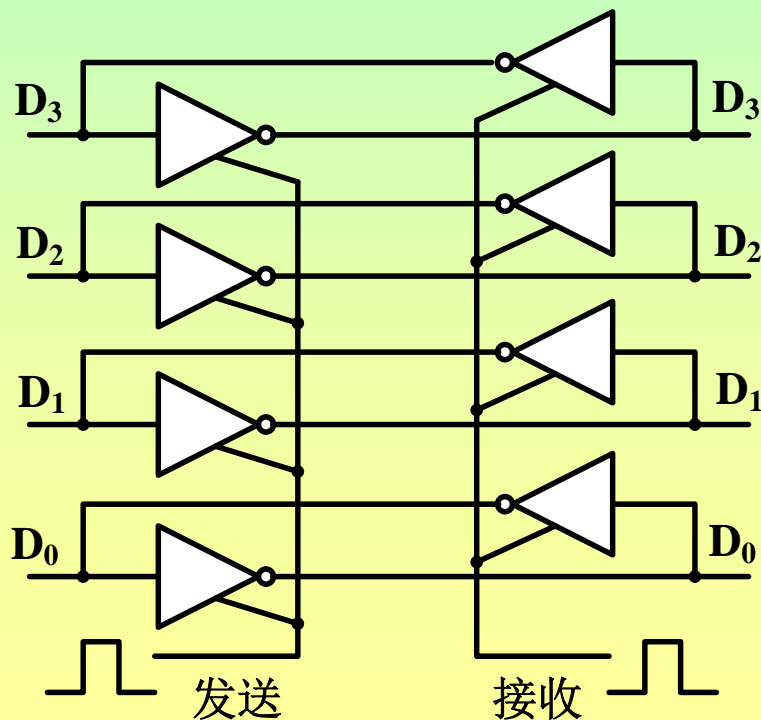


上图利用八片**74181**和两片**74182**构成两级先行进位的**32位ALU**。一片**74182**产生 $C_4$ 、 $C_8$ 、 $C_{12}$ ，另一片**74182**产生 $C_{20}$ 、 $C_{24}$ 、 $C_{28}$ 。

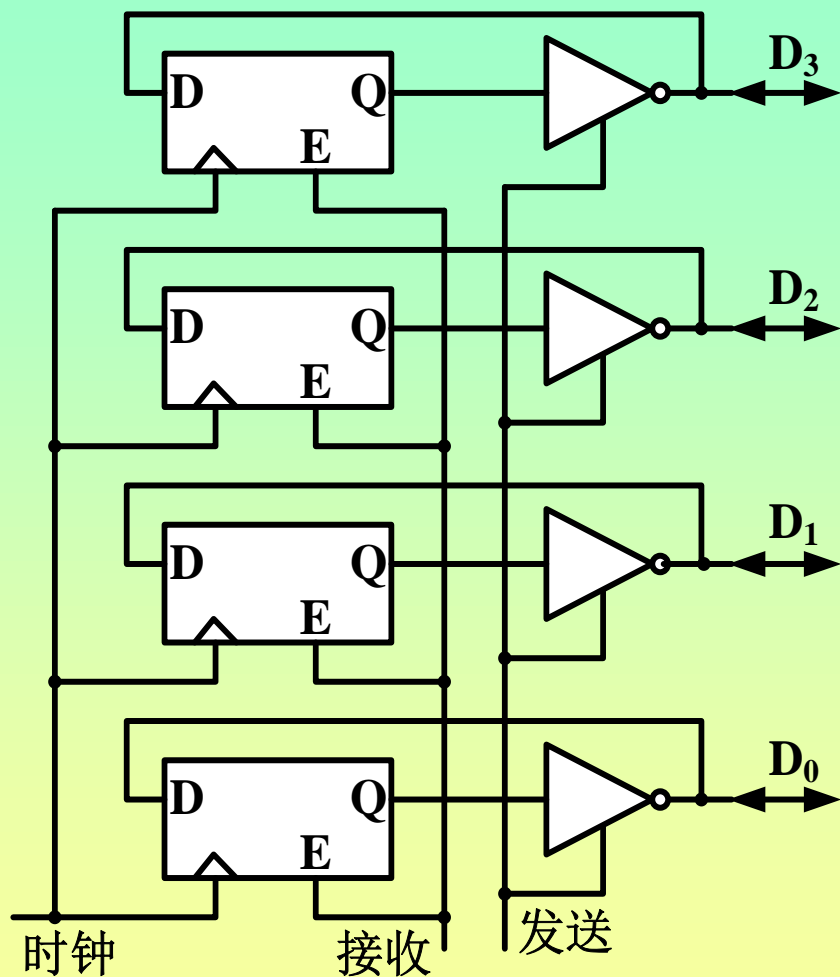
# 内部总线

根据总线所在位置，总线分为内部总线和外部总线两类。内部总线是指CPU内各部件的连线，而外部总线是指系统总线，即CPU与存储器、I/O系统之间的连线。本节讨论内部总线。

按总线的逻辑结构来说，总线可分为单向传送总线和双向传送总线。所谓单向总线，就是信息只能向一个方向传送。所谓双向总线，就是信息可以分两个方向传送，既可以发送数据，也可以接收数据。



左图是带有缓冲器的4位双向数据总线，其中所用的基本电路就是三态门。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。

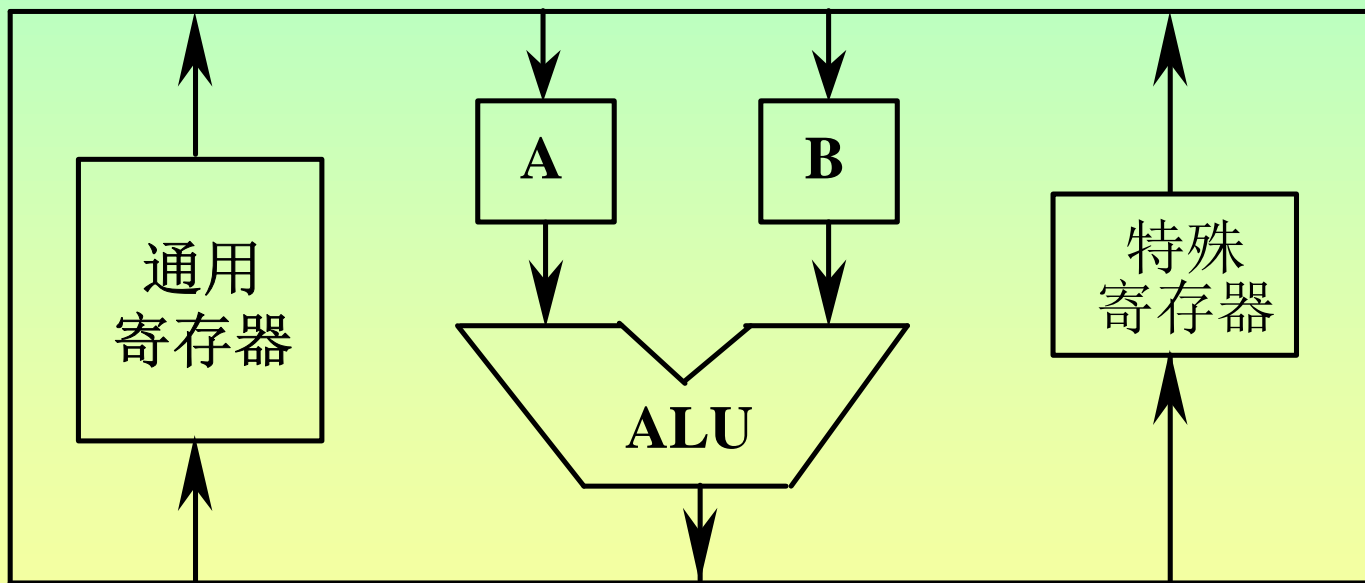


左图是带有锁存器的4位双向数据总线，由DE触发器（带有允许端E的D触发器）和三态门组成。E=1时，总线上的数据被接收到锁存器，此时发送信号无效将三态门禁止；E=0时，触发器的锁存功能无效，发送信号有效时将触发器输出端Q的值送到数据总线上。

# 定点运算器的基本结构

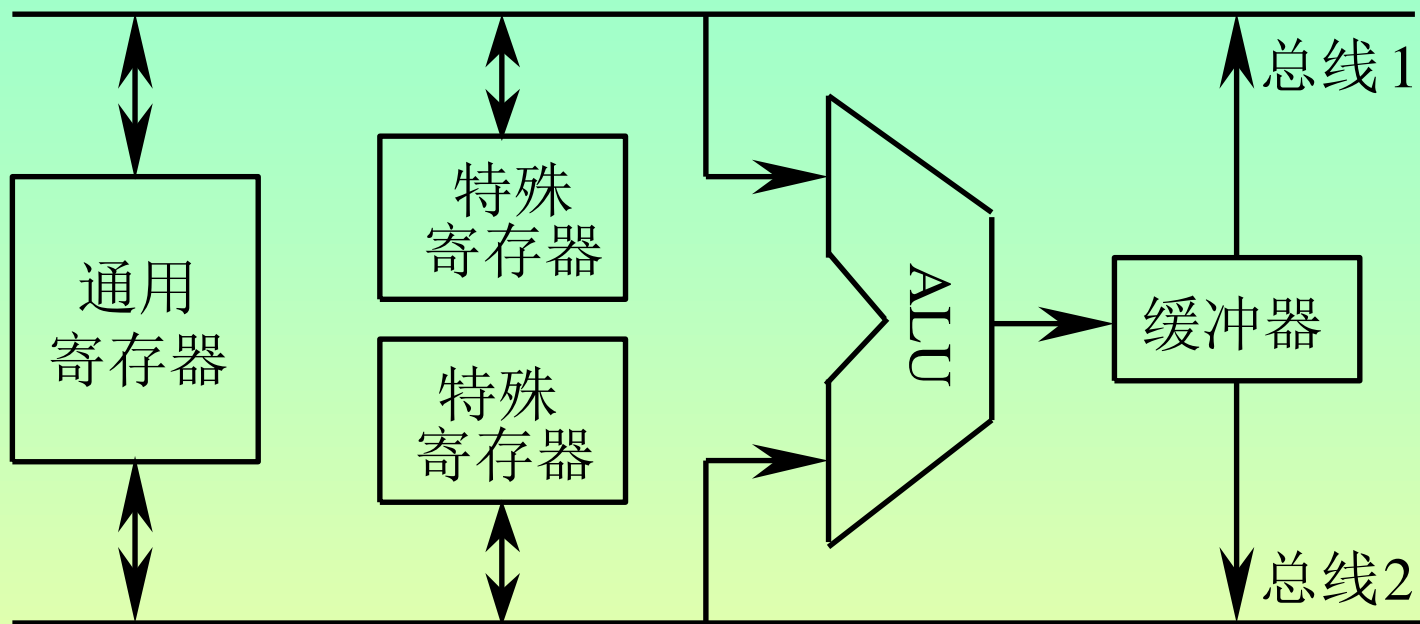
运算器包括**ALU**、乘法阵列、寄存器、多路开关、三态缓冲器、数据总线等逻辑部件。运算器的设计，主要围绕**ALU**、寄存器和数据总线之间如何传送操作数和运算结果进行。运算器大体有以下三种结构：

## 1.单总线结构的运算器



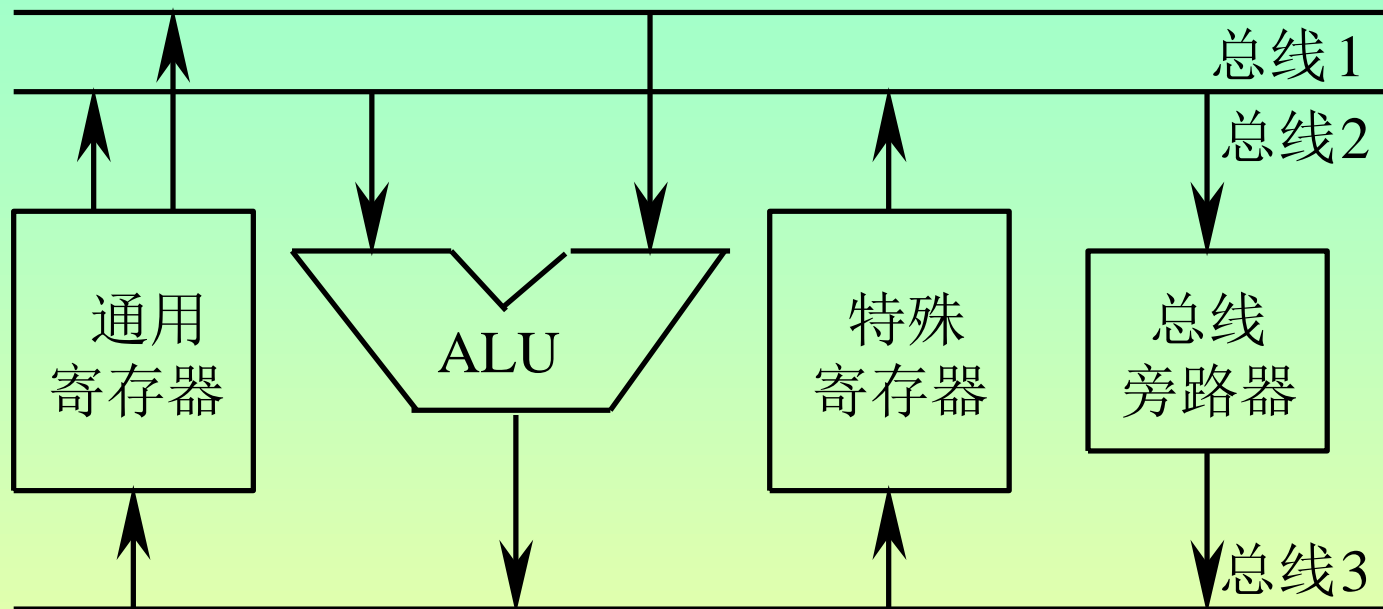
在同一时间内，只能有一个操作数放在单总线上。为了把两个操作数输入到**ALU**，需要分两次来做，而且还需要**A**、**B**两个缓冲寄存器。这种结构的主要缺点是操作速度较慢，但控制电路简单。

## 2.双总线结构的运算器



两个操作数可以分别通过总线1和总线2同时送到ALU去进行运算，并且立即可以得到运算的结果。但是ALU的输出不能直接送到总线上，这是因为此时两条总线都被操作数所占据着，所以必须在ALU的输出端设置一个缓冲器，先将运算结果送入缓冲器，第二步再把结果送至目的寄存器。显然，它的执行速度比单总线要快，每次操作比单总线少一步。

### 3.三总线结构的运算器



ALU的两个输入端分别由两条总线供给，输出与第三条总线相连，这样算术逻辑操作可以在一步控制之内完成。如果某一个数不需要运算和修改，而需要直接由总线2传到总线3，可通过总线旁路器把数据送出，而不必借助于ALU。三总线结构的特点是操作速度快，但控制较前两种复杂。

作业:

写出 74181ALU

**S3~S0:0011**

**S3~S0:1001**

两种状态下的正逻辑的输出结果（含推导过程）



# 浮点加减运算

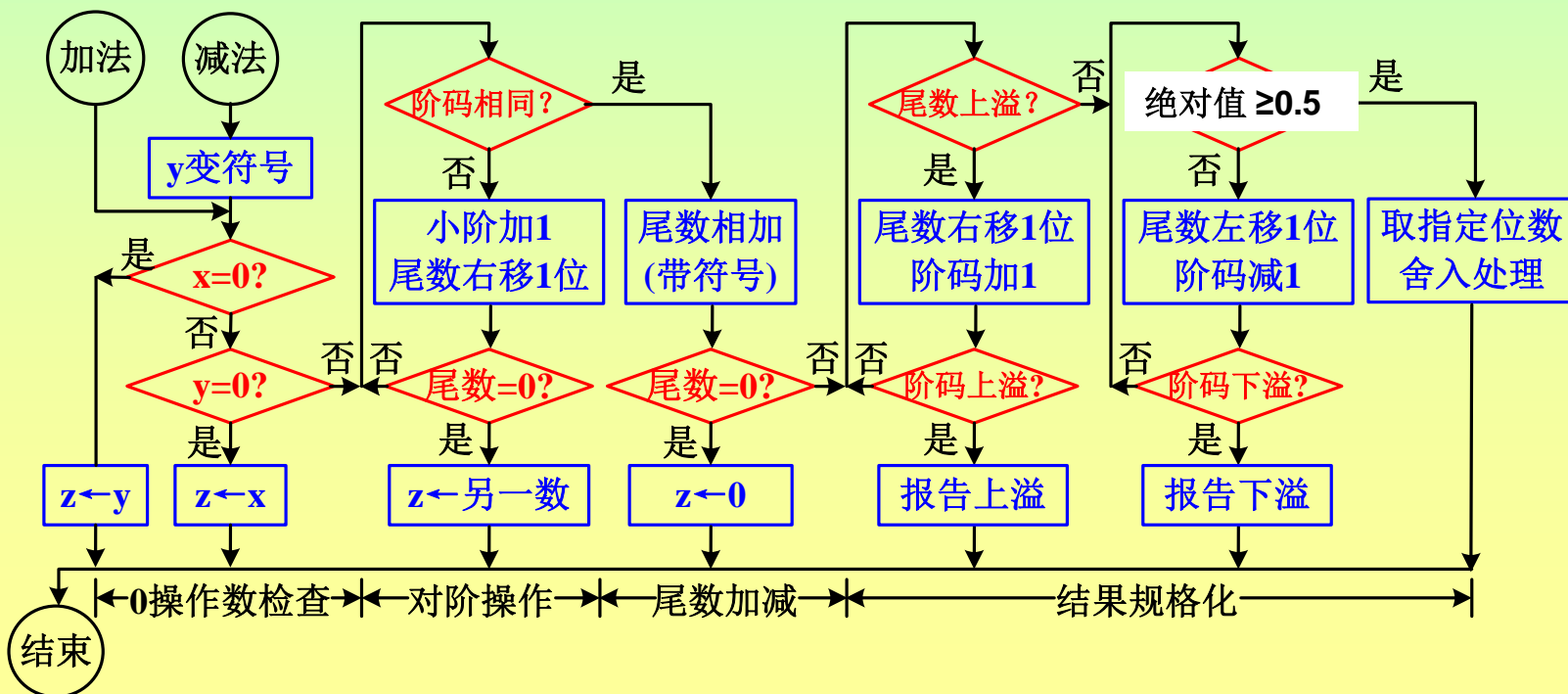
设有两个浮点数 $x$ 和 $y$ ，它们分别为

$$x = 2^{E_x} \cdot M_x \quad y = 2^{E_y} \cdot M_y$$

其中 $E_x$ 和 $E_y$ 分别为数 $x$ 和 $y$ 的阶码， $M_x$ 和 $M_y$ 为数 $x$ 和 $y$ 的尾数。

两浮点数进行加法和减法的运算规则是：

$$x \pm y = \begin{cases} (M_x \cdot 2^{E_x - E_y} \pm M_y) \cdot 2^{E_y} & E_x \leq E_y \\ (M_x \pm M_y \cdot 2^{E_y - E_x}) \cdot 2^{E_x} & E_x > E_y \end{cases}$$



以十进制为例，思考，如何计算

$$7.5 \times 10^5 + 8.25 \times 10^7$$

## (1) 0 操作数检查

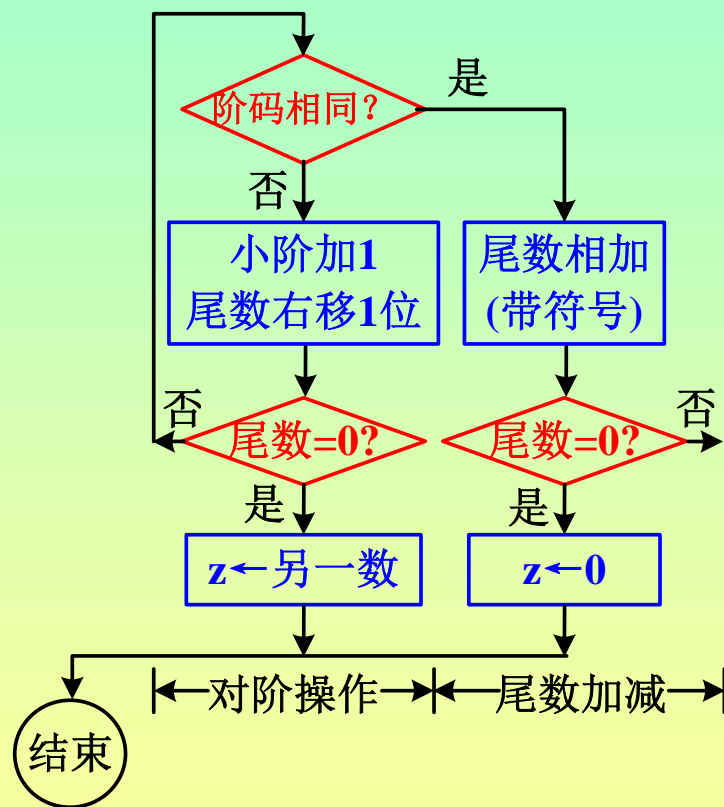
如果判知两个操作数 $x$ 或 $y$ 中有一个数为0，即可得知运算结果而没有必要再进行后续的一系列操作以节省运算时间。

## (2) 比较阶码大小并完成对阶

若二数阶码相同，表示小数点是对齐的，可以进行尾数的加减运算；若二数阶码不同，表示小数点位置没有对齐，此时必须使二数阶码相同，这个过程叫作对阶。由于浮点数是规格化的，尾数左移会引起最高有效位的丢失，造成很大误差；尾数右移虽引起最低有效位的丢失，但造成误差较小。因此，对阶操作规定使尾数右移，阶码相应增加。对阶时，总是使小阶向大阶看齐，即小阶的尾数向右移位，每右移一位，其阶码加1，直到两数阶码相等。按“小阶向大阶看齐”的原则。

## (3) 尾数求和

对阶结束后，即可进行尾数的求和运算。其方法与定点加减法运算一样。



#### (4) 结果规格化

尾数加减运算后得到的结果可能不是规格化数，此时要对结果进行规格化，具体做法取决于浮点数格式。设尾数M用补码表示，规格化的要求是  $\frac{1}{2} \leq M < 1$  或  $-1 \leq M < -\frac{1}{2}$ 。假定运算中采用双符号位，运算结果可能有6种情况：

① 00.1 × × ... ×

② 11.0 × × ... ×

③ 00.0 × × ... ×

④ 11.1 × × ... ×

⑤ 01.× × × ... ×

⑥ 10.× × × ... ×

第①种和第②种已经是规格化数。

第③种和第④种需要使尾数左移以实现规格化，称为左规。尾数每左移一次，阶码相应减1。左规可以进行多次。

第⑤种和第⑥种表示运算结果的绝对值大于1，在定点运算中称为溢出，但对浮点数可以通过尾数右移实现规格化，称为右规。尾数每右移一位，阶码相应加1。右规只需进行一次。

## (5) 舍入处理

对阶或右规时，尾数要向右移位，被右移的尾数的低位部分会被丢掉，此时要进行舍入处理。简单的舍入方法有两种：一种是“0舍1入”法，即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。另一种是“恒置1”法（又称冯·诺依曼舍入），即只要有数位被移掉，就将尾数的末位置1。

### P63

*IEEE 754*标准中，舍入处理提供了四种可选方法：

**就近舍入：**其实质就是“四舍五入”。例如，尾数超出规定的23位的多余数字是10010，则最低有效位应加1；若多余的5位是01111，则简单的截尾。对多余的5位是10000这种特殊情况，规定若最低有效位现为0，则截尾；若最低有效位现为1，则向上进一位使其变为0。

**朝0舍入：**朝数轴原点方向舍入，就是简单的截尾。无论尾数是正数还是负数，截尾都使结果的绝对值比原值的绝对值小。

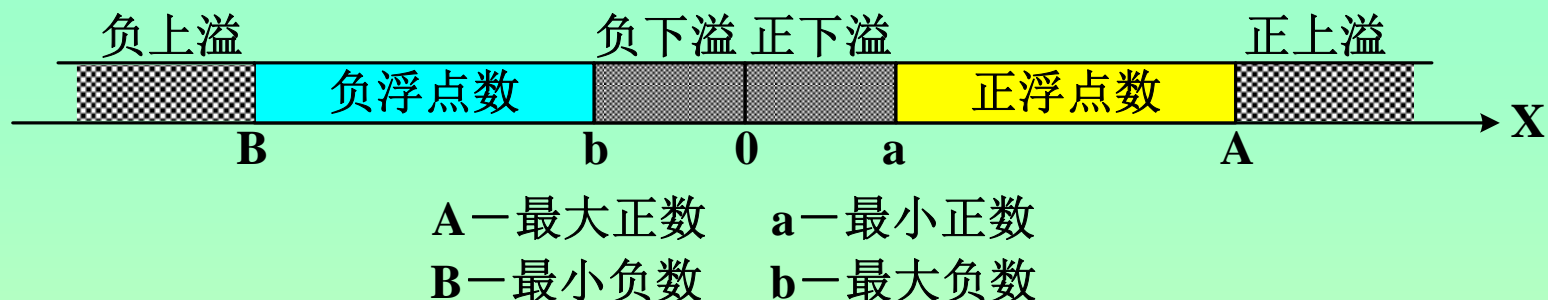
**朝 $+\infty$ 舍入：**对正数来说，只要多余位不全为0则向最低有效位进1；对负数来说则是简单的截尾。舍入后的值一定比原值大。

**朝 $-\infty$ 舍入：**对正数来说，是简单截尾；对负数来说，只要多余位不全为0则向最低有效位进1。舍入后的值一定比原值小。

**尾数用补码表示时**，其舍入的效果应当和用原码表示时的舍入效果完全相同。如果尾数用原码表示时采用的是“0舍1入”法，则尾数用补码表示且尾数为负数时其舍入规则为：

- ① 被丢掉的各位均为0时，舍去；
- ② 被丢掉数位的最高位为0，以下各位不全为0，则舍去；
- ③ 被丢掉数位的最高位为1，且以下各位全为0，则舍去；
- ④ 被丢掉数位的最高位为1，以下各位不全为0，则将尾数的末位加1。

## (6) 溢出处理



机器浮点数值大于最大正数A值，或小于最小负数B值时，称为上溢，这两种情况意味着阶码运算值超出了它所表示的范围，机器必须做中断处理。

当机器浮点数值小于最小正数a值，或大于最大负数b值时，称为下溢。下溢通常看作为机器零。

浮点数的溢出是以其阶码溢出表现出来的。若阶码溢出，则要进行相应处理。另外对尾数的溢出也需要处理。

**阶码上溢：**正指数值超过了阶码可能表示的最大值，一般将其认为是 $+\infty$ 和 $-\infty$ 。

**阶码下溢：**负指数值超过了阶码可能表示的最小值，一般将其认为是0。

**尾数上溢：**两个同符号尾数相加，结果的绝对值超过1，需进行右规。

**尾数下溢：**尾数的有效位从右端移出了，需进行舍入处理。

[例] 设  $x = 2^2 \times 0.11011011$

$y = 2^{-4} \times (-0.10101100)$ , 求  $x + y$

$[x]_{\text{浮}} = 00\ 010, \quad 0.11011011$

$[y]_{\text{浮}} = 00\ 100, \quad 1.01010100$

<1> 0 操作数检查

<2> 求阶差并对阶

$$\Delta E = E_x - E_y = [E_x]_{\text{补}} + [-E_y]_{\text{补}} = 00\ 010 + 11\ 100 = 11\ 110$$

即  $\Delta E$  为 -2,  $x$  的阶码小, 应使  $M_x$  右移两位,  $E_x$  加 2,

$[x]_{\text{浮}} = 00\ 100, 0.00110110(11)$

其中(11)表示  $M_x$  右移 2 位后移出的最低两位数。舍入处理

$[x]_{\text{浮}} = 00\ 100, 0.00110111$

<3> 尾数求和 (注意, 需要用双符号位来判断溢出)

00.00110111

+ 11.01010100

11.10001011      符号位为 11, 不溢出

<4> 规格化处理

尾数运算结果的符号位与最高数值位同值, 应执行左规处理, 结果为  
1.00010110, 阶码为 00 011

<5> 舍入处理

<6> 阶码判溢出: 阶码符号位为 00, 不溢出

120  $x + y = 2^{011} \times (-0.11101010)$



例：已知 $x = -0.875 \times 2^1$ ， $y = 0.625 \times 2^2$ ，设阶码2位，阶符1位，数符1位，尾数3位。通过补码运算规则求 $z = x - y$ 的二进制浮点规格化的结果。

$S1 = (-0.111)_2$  [S1]补 = 1.001

$S2 = (+0.101)_2$  [S2]补 = 0.101      $[-S2]$ 补 = 1.011

## 0 操作数检查

对阶后：

x 010,1100

-y 010,1011

尾数1.001右移一位，阶码加1  
1.1001，尾数4位，  
进行舍入，舍入后为 1.100

尾数（采用双符号位）：

```

      11.100
+   11.011
-----

```

110.111

符号位为10，溢出 小数点左移，阶码加1

阶码为3

121 结果：011,1011

尾数1.011  $-0.625 \times 2^3$

例：已知 $x = 0.125 \times 2^1$ ， $y = 0.625 \times 2^3$ ，设阶码2位，阶符1位，数符1位，尾数3位。通过补码运算规则求 $z = x - y$ 的二进制浮点规格化的结果。

$$S1 = (+0.001)_2 \quad [S1]_{\text{补}} = 0.001$$

$$S2 = (+0.101)_2 \quad [S2]_{\text{补}} = 0.101 \quad [-S2]_{\text{补}} = 1.011$$

## 0 操作数检查

对阶后：

**x 011,0000**

**-y 011,1011**

尾数为零：

$$z = -y$$

结果：

**011, 1011**

避免将一个很大的数与很小的数相加减，  
小数会被大数“吃掉”

# 浮点乘除运算

设有两个浮点数x和y:

$$x = 2^{E_x} \cdot M_x$$

$$y = 2^{E_y} \cdot M_y$$

浮点乘法运算的规则是:

$$x \times y = 2^{(E_x + E_y)} \cdot (M_x \times M_y)$$

即乘积的尾数是两数的尾数之积，乘积的阶码是两数的阶码之和。

浮点除法运算的规则是：

$$x \div y = 2^{(E_x - E_y)} \cdot (M_x \div M_y)$$

即商的尾数是两数的尾数之商，商的阶码是两数的阶码之差。

浮点乘除运算可分为六步：**0**操作数检查；阶码加/减操作；尾数乘/除操作；结果规格化和舍入处理，确定积的符号。

## (1) 阶码运算

浮点数的阶码通常用补码或移码表示。

下面介绍移码的运算规则[补充内容]。

移码的定义为

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

按此定义，则有

$$\begin{aligned} [x]_{\text{移}} + [y]_{\text{移}} &= 2^n + x + 2^n + y = 2^n + (2^n + (x + y)) \\ &= 2^n + [x + y]_{\text{移}} \end{aligned}$$

即直接用移码对阶码求和时，结果的最高位多加了个1。要得到正确的移码结果，必须对结果的符号位再执行一次求反。

求阶码之和可以采用另一种方式实现。 $[y]_{\text{补}}$ 的定义为：

$$[y]_{\text{补}} = 2^{n+1} + y \quad (\text{mod } 2^{n+1})$$

于是

$$\begin{aligned} [x]_{\text{移}} + [y]_{\text{补}} &= 2^n + x + 2^{n+1} + y = 2^{n+1} + (2^n + (x + y)) \\ &= [x + y]_{\text{移}} \quad (\text{mod } 2^{n+1}) \end{aligned}$$

同理：

$$[x - y]_{\text{移}} = [x]_{\text{移}} + [-y]_{\text{补}} \quad (\text{mod } 2^{n+1})$$

若阶码的运算结果溢出，上述公式不成立。为进行溢出检测，运算中采用双符号位，并规定**移码的高符号位为0**，则结果的高符号位为**1**表示溢出。结果的符号位为**10**表示上溢，为**11**表示下溢，**01**表示正，**00**表示负。

## (2) 尾数处理

浮点加减法对结果的规格化和舍入处理也适用于浮点乘法。

尾数用原码表示时，简单的舍入方法有两种：一种是“恒置1”法，即只要有数位被移掉，就将尾数的末位置1。另一种是“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。

**尾数用补码表示时**，其舍入的效果应当和用原码表示时的舍入效果完全相同。如果尾数用原码表示时采用的是“0舍1入”法，则尾数用补码表示且尾数为负数时其舍入规则为：

- ① 被丢掉的各位均为0时，舍去；
- ② 被丢掉数位的最高位为0，以下各位不全为0，则舍去；
- ③ 被丢掉数位的最高位为1，且以下各位全为0，则舍去；
- ④ 被丢掉数位的最高位为1，以下各位不全为0，则将尾数的末位加1。

**[例26]**  $x = +011$ ,  $y = +110$ , 求  $[x + y]_{\text{移}}$  和  $[x - y]_{\text{移}}$ , 并判断是否溢出。

**[解:]**

$[x]_{\text{移}} = 01\ 011$ ,  $[y]_{\text{补}} = 00\ 110$ ,  $[-y]_{\text{补}} = 11\ 010$

$[x + y]_{\text{移}} = [x]_{\text{移}} + [y]_{\text{补}} = 10\ 001$ , 结果上溢。

$[x - y]_{\text{移}} = [x]_{\text{移}} + [-y]_{\text{补}} = 00\ 101$ , 结果正确, 为  $-3$ 。

**[例]**  $x = -011$ ,  $y = +110$ , 求  $[x + y]_{\text{移}}$  和  $[x - y]_{\text{移}}$ , 并判断是否溢出。

**[解:]**

$[x]_{\text{移}} = 00\ 101$ ,  $[y]_{\text{补}} = 00\ 110$ ,  $[-y]_{\text{补}} = 11\ 010$

$[x + y]_{\text{移}} = [x]_{\text{移}} + [y]_{\text{补}} = 01\ 011$ , 结果+3。

$[x - y]_{\text{移}} = [x]_{\text{移}} + [-y]_{\text{补}} = 11\ 111$ , 结果下溢

**[例27]** 设  $[x_1]_{\text{补}} = 11.01100000$ ,  
           $[x_2]_{\text{补}} = 11.01100001$ ,  
           $[x_3]_{\text{补}} = 11.01101000$ ,  
           $[x_4]_{\text{补}} = 11.01111001$ ,  
求执行只保留小数点后4位有效数字的舍入操作值。

**[解:]**

执行舍入操作后,其结果值分别为

$[x_1]_{\text{补}} = 11.0110$  (不舍不入)  
 $[x_2]_{\text{补}} = 11.0110$  (舍)  
 $[x_3]_{\text{补}} = 11.0110$  (舍)  
 $[x_4]_{\text{补}} = 11.1000$  (入)



**【例30】** 设有浮点数  $x = 2^{-5} \times 0.0110011$ ,  
 $y = 2^3 \times (-0.1110010)$ , 阶码用4位补码表示, 尾数 (含符号位)  
用8位原码表示。求  $[x \times y]$  浮。要求用原码完成尾数乘法运  
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。

移码采用双符号位, 尾数补码采用单符号位, 则有

$$[M_x]_{\text{原}} = 0.0110011, [M_y]_{\text{原}} = 1.1110010,$$

$$[E_x]_{\text{补}} = 11\ 011, [E_y]_{\text{补}} = 00\ 011,$$

$$[x]_{\text{浮}} = 11\ 011, 0.0110011, [y]_{\text{浮}} = 00\ 011, 1.1110010$$

## 1) 求阶码和

$$\begin{aligned}[E_x + E_y]_{\text{补}} &= [E_x]_{\text{补}} + [E_y]_{\text{补}} \\ &= 11\ 011 + 00\ 011 = 11\ 110, \text{值为补码形式}-2.\end{aligned}$$

## 2) 尾数乘法运算可采用原码阵列乘法器实现, 即有

$$\begin{aligned}[M_x]_{\text{原}} \times [M_y]_{\text{原}} &= [0.0110011]_{\text{原}} \times [1.1110010]_{\text{原}} \\ &= [1.0101101, 0110110]_{\text{原}}\end{aligned}$$

[1.0101101,0110110]原

**(3) 规格化处理**

乘积的尾数符号位与最高数值位符号相同,不是规格化的数,需要左规,阶码变为11 101(-3),

尾数变为 1.1011010,1101100。

**(4) 舍入处理**

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1.1011011。

最终相乘结果为

$[x \times y]_{\text{浮}} = 11\ 101, 1.1011011$

其真值为

$x \times y = 2^{-3} \times (-0.1011011)$

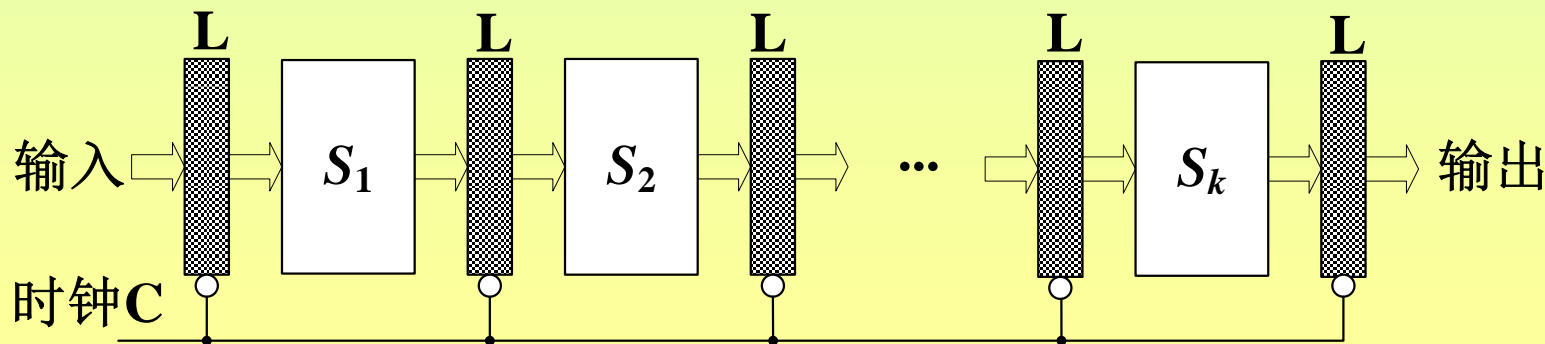
# 浮点运算流水线

把执行一条指令称为一个任务，一个任务可以分割为一系列子任务。每个子任务的处理过程称为过程段。不同任务的子任务可以在流水线的各个过程段并发执行，从而实现子任务级的并行。

流水线中，原则上要求各个过程段的处理时间相同。设任务 $T$ 被分成 $k$ 个子任务，可表达为

$$T = \{ T_1, T_2, \dots, T_k \}$$

各子任务之间有一定的优先关系：若 $i < j$ ，则必须 $T_i$ 完成之后， $T_j$ 才能开始工作。具有这种优先关系的流水线称为线性流水线，由一系列串联的过程段组成，各个过程段之间设有高速缓存，以暂时保存上一个过程段处理的结果。在统一的时钟 $C$ 的控制下，数据从一个过程段流向下一个过程段。



设过程段  $S_i$  所需的时间为  $\tau_i$ , 缓冲器的延时为  $\tau_l$ , 线性流水线的时钟周期定义为

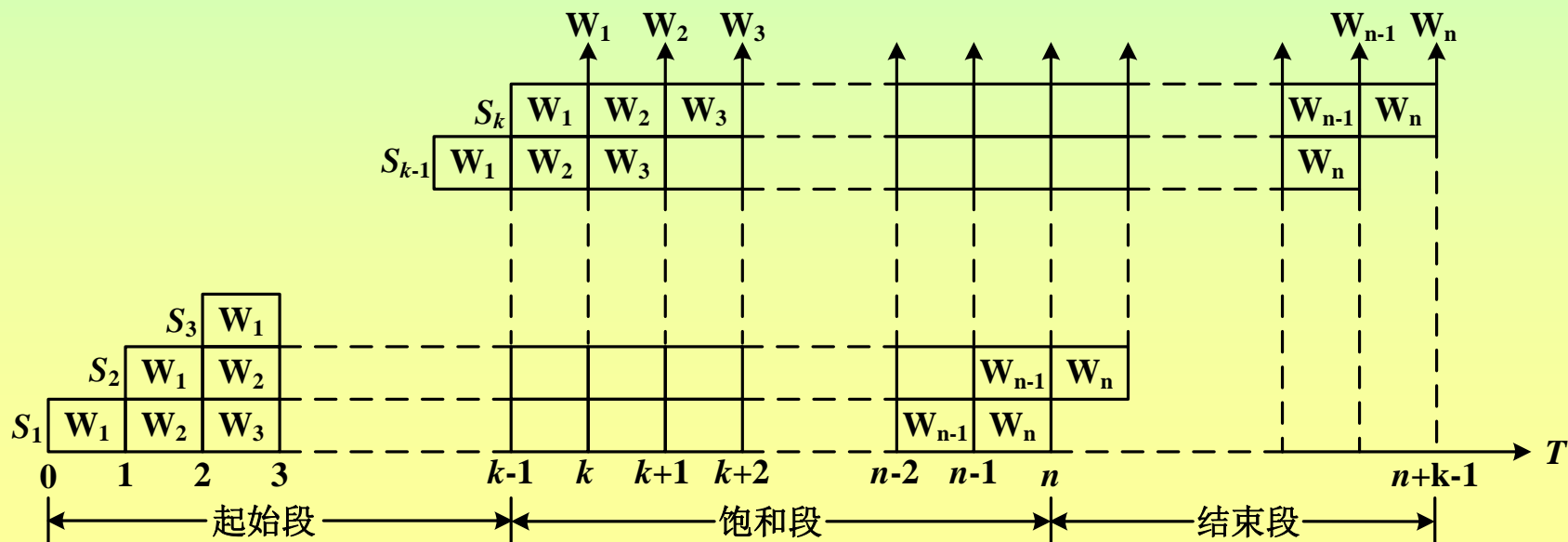
$$\tau = \max\{\tau_i\} + \tau_l = \tau_m + \tau_l$$

流水线处理的频率为  $f = 1/\tau$ 。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

$$T_k = k + (n - 1)$$

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, 则所需时钟周期数为

$$T_L = n \cdot k$$



将 $T_L$ 和 $T_k$ 的比值定义为 $k$ 级线性流水线的加速比：

$$C_k = \frac{T_L}{T_k} = \frac{n \cdot k}{k + (n - 1)}$$

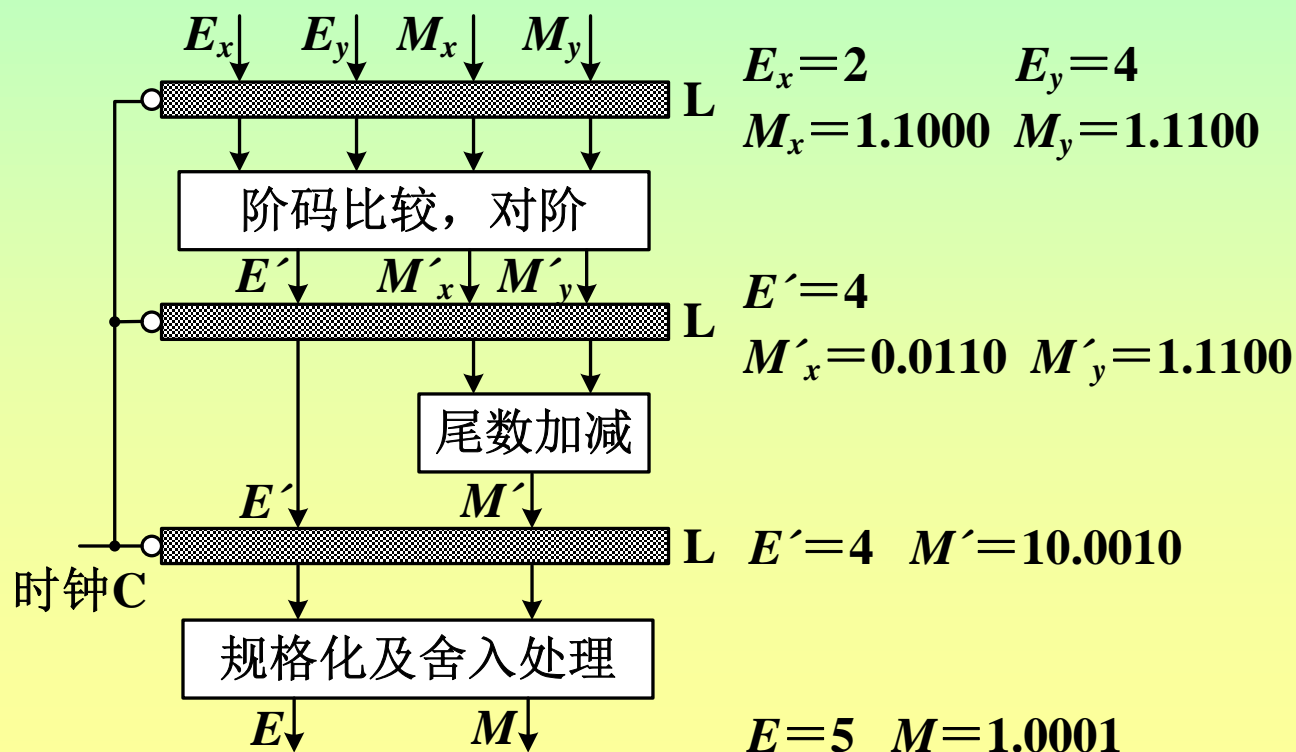
当 $n \gg k$ 时， $C_k \rightarrow k$ 。这就是说，理论上 $k$ 级线性流水线处理几乎可以提高 $k$ 倍速度。但实际上这个理想的加速比不一定能达到。

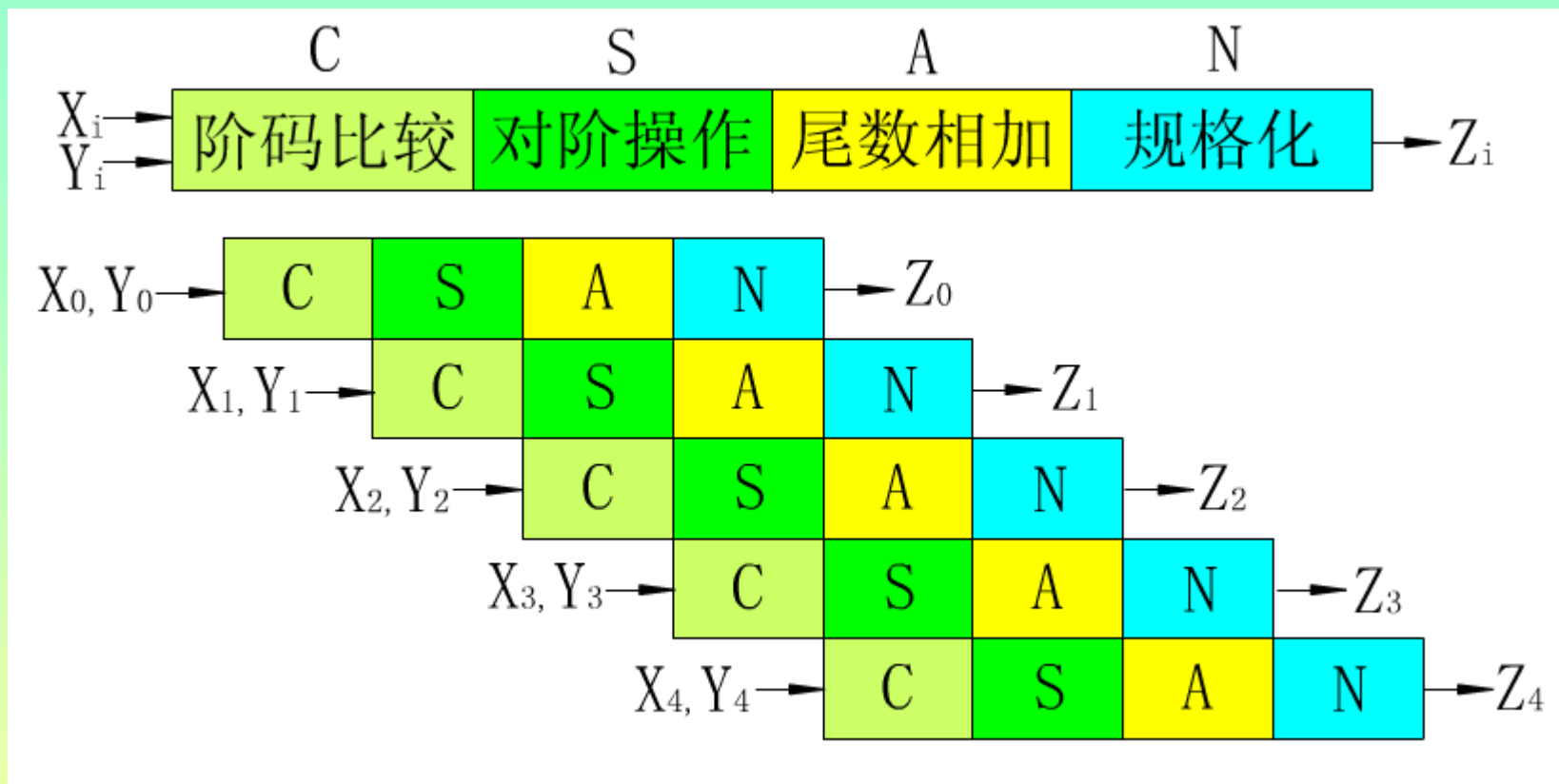
右图是3段浮点加法器的框图。设两个IEEE 754标准的规格化浮点数为：

$$X = 1.1000 \times 2^2$$

$$Y = 1.1100 \times 2^4$$

缓冲器中保存的流水运算结果如图。





向量加法计算的流水时空图

- 关于规格化的两种形式
  - 1、IEEE754的要求为1.M的形式
  - 2、设尾数M用补码表示，规格化的要求是 $\frac{1}{2} \leq M < 1$ 或 $-1 \leq M < -\frac{1}{2}$
- 关于移码的两种形式
  - 1、IEEE754的要求为e+127的形式
  - 2、其余的为 $2^n+x$

若不指明为IEEE754标准，所指的规格化、移码均为第2种



## 本节要点

- 1、浮点数加、减运算过程
- 2、浮点数乘、除运算过程

# 作业

1、分析在正逻辑的情况下，当

**$S_3S_2S_1S_0=LHLH$ ,  $M=H$ ,  $M=L$ 时，74181实现 何种运算？给出具体推导过程**

2、P64 9






3、 **$X=2^{+010} \times (+0.1101)$ ,  $Y=2^{+100} \times (-0.1010)$ , 按照浮点数加减法的计算方法求 $X-Y$**

练习：

**P64 8（2）** 用加减交替法来实现

知识点： 原码阵列除法

习题8 (2)  $[x]_{\text{补}}=0.01011$   $[y]_{\text{补}}=0.11001$   $[-y]_{\text{补}}=1.00111$

	0. 0 1 0 1 1	商
$+[-y]_{\text{补}}$	1. 0 0 1 1 1	
	<span style="border: 1px dashed black;">0</span> 1. 1 0 0 1 0	0
	1. 0 0 1 0 0	
$+ [y]_{\text{补}}$	0. 1 1 0 0 1	
	<span style="border: 1px dashed black;">0</span> 1. 1 1 1 0 1	0.0
	1. 1 1 0 1 0	
$+ [y]_{\text{补}}$	0. 1 1 0 0 1	
	<span style="border: 1px dashed black;">1</span> 0. 1 0 0 1 1	0.0 1
	1. 0 0 1 1 0	
$+ [-y]_{\text{补}}$	1. 0 0 1 1 1	
	<span style="border: 1px dashed black;">1</span> 0. 0 1 1 0 1	0.0 1 1
	0. 1 1 0 1 0	
$+ [-y]_{\text{补}}$	1. 0 0 1 1 1	
	<span style="border: 1px dashed black;">1</span> 0. 0 0 0 0 1	0.0 1 1 1
	0. 0 0 0 1 0	
$+ [-y]_{\text{补}}$	1. 0 0 1 1 1	
	<span style="border: 1px dashed black;">0</span> 1. 0 1 0 0 1	0.0 1 1 1 0

$|x| \div |y|$ :

$$q = 0.01110$$

$$r = 0.00001 \times 2^{-4}$$

$x \div y$ :

$$q = -0.01110$$

$$r = -0.00001 \times 2^{-4}$$

## 第二章 复习

- 数的原码、补码、反码、移码的表示方法，注意边界情况（0的表示，表示范围， $-2^n$ ）
- 浮点数的表示方法：尾数用定点小数的原码或补码表示；指数用整数的补码或移码表示。数符、阶符分别表示什么含义？正、负、绝对值大于1  
规格化最小正数是： $1.00\dots0$      $0.10\dots0$
- 数的规格化表示方法
- **IEEE754**表示方法  
尾数用原码表示，规格化为 $1.M$ 的形式，表示时隐藏1；阶码为真值加127
- 指定格式的浮点数表示方法
- 补码加法的公式，补码减法的公式
- 溢出的检测，单符号位，双符号位
- 全加器的公式以及时间延迟
- 行波进位补码加/减法器的电路分析
- 原码阵列乘法器

## 第二章 复习

- 对2求补电路
- 间接补码阵列乘法器
- 直接补码阵列乘法器
- 原码除法的两种方法：恢复余数法，加减交替法
- 阵列除法器
- **ALU**的电路基本概念
- 先行进位的基本原理
- 浮点加减运算
- 浮点乘除运算
- 流水线的基本概念以及优点

作业：

P61

8（1）用加减交替法来实现