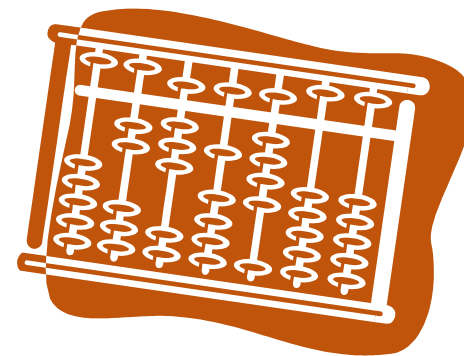




# 计算机组成原理



## 三、运算器



# 本章主要内容

- 3.1 计算机中的运算
- 3.2 定点加减法运算
- 3.3 定点乘法运算
- 3.4 定点除法运算
- 3.5 浮点运算
- 3.6 运算器



# C语言中的位运算

位运算: “&” “|” “~” “^”, 分别对应逻辑与、或、非、异或操作。这些位运算操作符会在编译器的作用下被翻译成与之对应的汇编指令, 如x86中的逻辑与指令 and、逻辑或指令 or、逻辑非指令 not、逻辑异或指令 xor

```
4      int i=1,j=-1;
0x40134e    movl    $0x1,0xc(%esp)
0x401356    movl    $0xffffffff,0x8(%esp)
5      i=i&j;
0x40135e    mov     0x8(%esp),%eax
0x401362    and     %eax,0xc(%esp)
6      i=i|j;
0x401366    mov     0x8(%esp),%eax
0x40136a    or      %eax,0xc(%esp)
7      i=~i;
0x40136e    notl    0xc(%esp)
```

# C语言中的逻辑运算

逻辑运算: “&&” “||” “!”，逻辑运算和位运算的区别在于，它属于非数值运算，操作数只能是“0”和“1”两个值，所有非“0”值都被当作“1”处理，

```
4      int i=2,j=-1;
0x40134e    movl    $0x2,0xc(%esp)
0x401356    movl    $0xffffffff,0x8(%esp)
5      i=i&&j;
0x40135e    cmpl    $0x0,0xc(%esp)
0x401363    je      0x401373 <main+51>
0x401365    cmpl    $0x0,0x8(%esp)
0x40136a    je      0x401373 <main+51>
0x40136c    mov     $0x1,%eax
0x401371    jmp     0x401378 <main+56>
0x401373    mov     $0x0,%eax
0x401378    mov     %eax,0xc(%esp)
```

# C语言中的移位运算

移位运算：“<<” “>>”，分别代表左移和右移。左移运算操作符对应汇编指令中的逻辑左移，而右移运算操作符则根据操作数是无符号还是有符号类型分别对应汇编指令中的逻辑右移和算术右移指令。

```
4      int i=-1;unsigned j=2;
0x40134e    movl    $0xffffffff,0xc(%esp)
0x401356    movl    $0x2,0x8(%esp)
5      i=i<<7;
0x40135e    shll    $0x7,0xc(%esp)
6      i=i>>8;
0x401363    sarl    $0x8,0xc(%esp)
7      j=j>>15;
0x401368    shr    $0xf,0x8(%esp)
8      i=i>>j;
0x40136d    mov     0x8(%esp),%eax
0x401371    mov     %al,%cl
0x401373    sarl    %cl,0xc(%esp)
```

# C语言中的算术运算

算术运算操作符：“+”“-”“\*”“/”4种，分别对应算术运算中的加、减、乘、除。加减运算编译程序通常直接转换成汇编语言中的 add、sub 指令，并不区分符号数据类型。乘除运算则会根据操作数符号类型进行不同的转换，如下面例程中的 mul/imul、div/idiv。

```
4      int i ,j; unsigned char ui,uj=255;
0x40134e    movb    $0xff,0x1f(%esp)
5      float e=1,f;
0x401353    mov     0x404140,%eax
0x401358    mov     %eax,0x18(%esp)
6      i=i+3;
0x40135c    addl    $0x3,0x14(%esp)
7      ui=ui-3;
0x401361    subb    $0x3,0x13(%esp)
8      i=i*j;
0x401366    mov     0x14(%esp),%eax
0x40136a    imul    0xc(%esp),%eax
```

# 本章主要内容

- 3.1 计算机中的运算
- 3.2 定点加减法运算
- 3.3 定点乘法运算
- 3.4 定点除法运算
- 3.5 浮点运算
- 3.6 运算器



## 3.2 定点加减法运算

### ■ 3.2.1 补码加减法运算方法

### ■ 3.2.2 溢出及检测

### ■ 3.2.3 加减法的逻辑实现



# 补码加减法的实现

■ **补码加法：**  $[X + Y]_{\text{补}} = [X]_{\text{补}} + [Y]_{\text{补}}$

□ 和的补码 = 补码的和

■ **补码减法：**  $[X - Y]_{\text{补}} = [X]_{\text{补}} + [-Y]_{\text{补}} = [X]_{\text{补}} - [Y]_{\text{补}}$

□ 差的补码 = 补码的差

□ 减法变加法，关键是求  $[-Y]_{\text{补}}$

■ **求补公式：**  $[-Y]_{\text{补}} = [ [Y]_{\text{补}} ]_{\text{补}}$

□ 对  $[Y]_{\text{补}}$  逐位取反，再在最低位加 1

# 补码加法公式证明

■  $[X + Y]_{\text{补}} = [X]_{\text{补}} + [Y]_{\text{补}}$

1.  $x > 0$   $y > 0$  (无需证明)

2.  $x > 0$   $y < 0$

3.  $x < 0$   $y > 0$  (2/3证明相同)

4.  $x < 0$   $y < 0$

■ 只需证明2/4两种情况即可

## 补码加法公式证明 $x > 0 \ y < 0$

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

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

当  $x + y < 0$  时

$$= [x + y]_{\text{补}} \pmod{2}$$

当  $x + y > 0$  时

$$2 + (x + y) > 2 \quad \text{模舍去}$$

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

$$= [x + y]_{\text{补}} \pmod{2}$$

## 补码加法公式证明 $x < 0 \quad y < 0$

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

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

$$-2 \leq x + y < 0$$

$$\text{故 } 0 \leq 2 + x + y < 2$$

$$\text{故 } 2 + (2 + x + y) \pmod{2} = (2 + x + y)$$

$$= [x + y]_{\text{补}} \pmod{2}$$

## 补码减法公式证明

$$[X-Y]_{\text{补}} = [X]_{\text{补}} - [Y]_{\text{补}} \quad ???$$

$$[X-Y]_{\text{补}} = [X]_{\text{补}} + [-Y]_{\text{补}} \quad (\text{加法公式})$$

$$[-Y]_{\text{补}} = -[Y]_{\text{补}} \quad ?$$

$$[-Y]_{\text{补}} + [Y]_{\text{补}} = [Y + (-Y)]_{\text{补}} = [0]_{\text{补}} = 0$$

$$\text{故} [-Y]_{\text{补}} = -[Y]_{\text{补}} \text{ 成立} \quad [X-Y]_{\text{补}} = [X]_{\text{补}} + [-Y]_{\text{补}} = [X]_{\text{补}} - [Y]_{\text{补}}$$

$$[-Y]_{\text{补}} = [ [Y]_{\text{补}} ]_{\text{补}}$$

## 3.2 定点加减法运算

### ■ 3.2.1 补码加减法运算方法

### ■ 3.2.2 溢出及检测

### ■ 3.2.3 加减法的逻辑实现

# 补码加法的几种情况

正常结果

$$\begin{array}{r} 0.10101 \\ + 0.01000 \\ \hline 0.11101 \end{array}$$

模数舍去  
正常结果

$$\begin{array}{r} 1.10101 \\ + 1.11000 \\ \hline 1.1.01101 \end{array}$$

正正得负  
正溢出

$$\begin{array}{r} 0.10101 \\ + 0.11000 \\ \hline 1.01101 \end{array}$$

负负得正  
负溢出

$$\begin{array}{r} 1.00101 \\ + 1.11000 \\ \hline 1.0.11101 \end{array}$$

计算机如何识别运算结果是否溢出？

## 根据操作数与运算结果的符号位是否一致监测

■ 溢出逻辑： 正正得负 负负得正

■ 设两数符号位为  $f_0 f_1$  , 和运算结果的符号位  $f_s$

■ 溢出检测信号 ***Overflow (OF)***

$$\text{加法: } \textit{Overflow} = \bar{f}_0 \bar{f}_1 f_s + f_0 f_1 \bar{f}_s$$

$$\text{减法: } \textit{Overflow} = f_0 \bar{f}_1 \bar{f}_s + \bar{f}_0 f_1 f_s$$



## 根据最高位数据位的进位与符号位是否一致检测

$C_f=0, C_n=0$

0	.	1	0	1	0	1
+	0	.	0	1	0	0
<hr/>						
0	.	1	1	0	1	

正常结果

$C_f=1, C_n=1$

	1	.	1	0	1	0	1
+	1	.	1	1	0	0	0
<hr/>							
1	1	.	0	1	1	0	1

正常结果

$C_f=0, C_n=1$

0	.	1	0	1	0	1
+	0	.	1	1	0	0
<hr/>						
1	.	0	1	1	0	1

正溢出

$C_f=1, C_n=0$

	1	.	0	0	1	0	1
+	1	.	1	1	0	0	0
<hr/>							
1	0	.	1	1	1	0	1

负溢出

符号位进位位  $C_f$ , 最高位进位位  $C_n$

$$Overflow = C_f \oplus C_n$$

# 双符号溢出检测方法

未溢出

	0	0	.	1	0	1	0	1
+	0	0	.	0	1	0	0	0
<hr/>								
	0	0	.	1	1	1	0	1

未溢出

	1	1	.	1	0	1	0	1
+	1	1	.	1	1	0	0	0
<hr/>								
1	1	1	.	0	1	1	0	1

正溢出

	0	0	.	1	0	1	0	1
+	0	0	.	1	1	0	0	0
<hr/>								
	0	1	.	0	1	1	0	1

负溢出

	1	1	.	0	0	1	0	1
+	1	1	.	1	1	0	0	0
<hr/>								
1	1	0	.	1	1	1	0	1

双符号位最高位永远是正确符号位

$$Overflow = f_1 \oplus f_2$$

## 3.2 定点加减法运算

### ■ 3.2.1 补码加减法运算方法

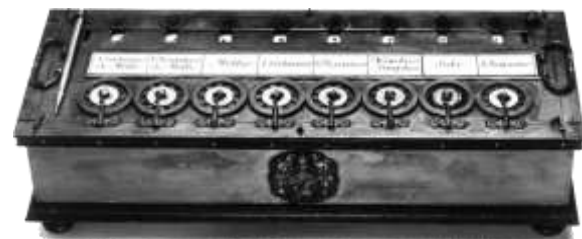
### ■ 3.2.2 溢出及检测

### ■ 3.2.3 加减法的逻辑实现

## 二进制加法运算

- 各位逐位相加，进位从右至左传递
- 首先要考虑一位加法，然后考虑进位链

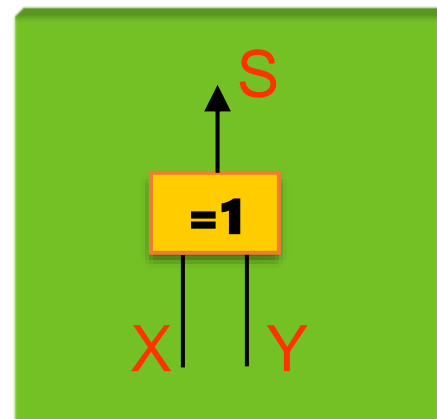
$$\begin{array}{rccccccc} & & X_{n-1} & \cdots & X_2 & X_1 & X_0 \\ + & & Y_{n-1} & \cdots & Y_2 & Y_1 & Y_0 \\ \hline & & ?_{n-1} & \cdots & ?_2 & ?_1 & ?_0 \end{array}$$



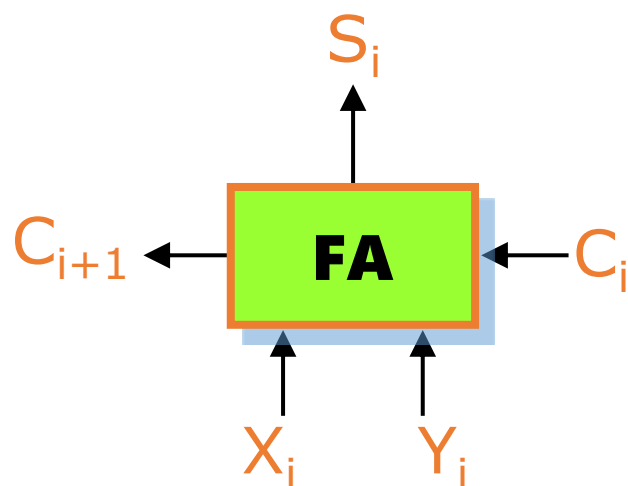
# 一位加法逻辑电路实现

- $0 + 1 = 1$     $1 + 0 = 1$
- $1 + 1 = 0$     $0 + 0 = 0$
- 一个异或门即可实现自动一位加法
- 算术运算变成逻辑电路

$$S = X \oplus Y$$



# 带进位链的一位全加器



一位全加器

加数 $X_i$	加数 $Y_i$	低位进位 $C_i$	和数 $S_i$	进位 $C_{i+1}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S_i = X_i \oplus Y_i \oplus C_i$$

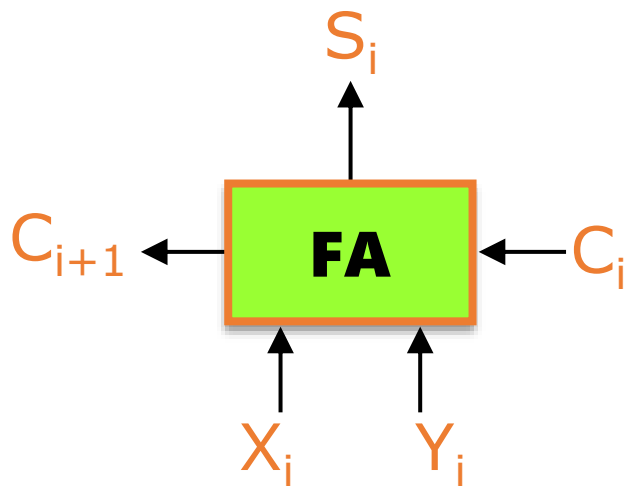
$$C_{i+1} = X_i Y_i + (X_i \oplus Y_i) C_i$$

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

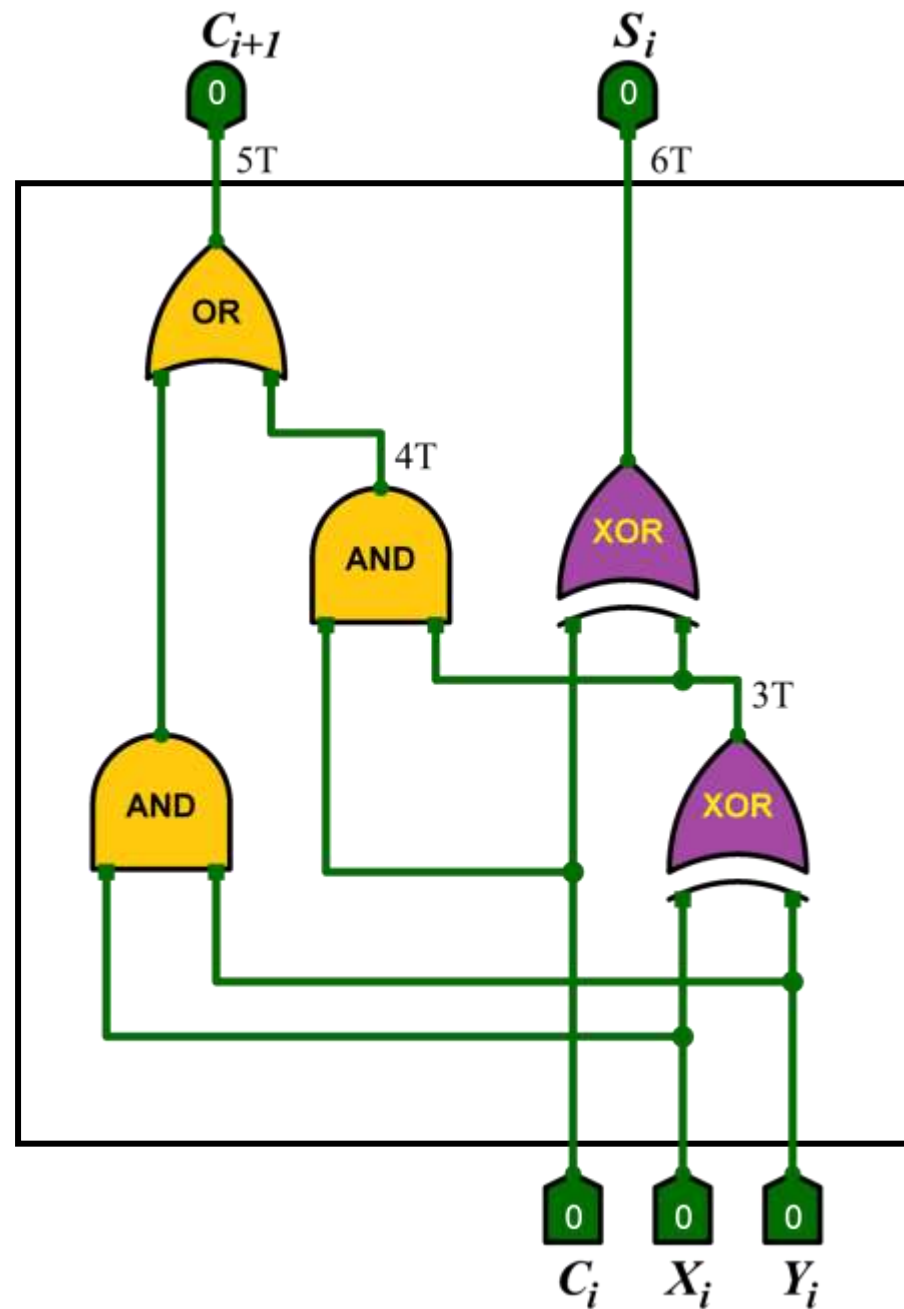
## 一位全加器逻辑实现

$$S_i = X_i \oplus Y_i \oplus C_i$$

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

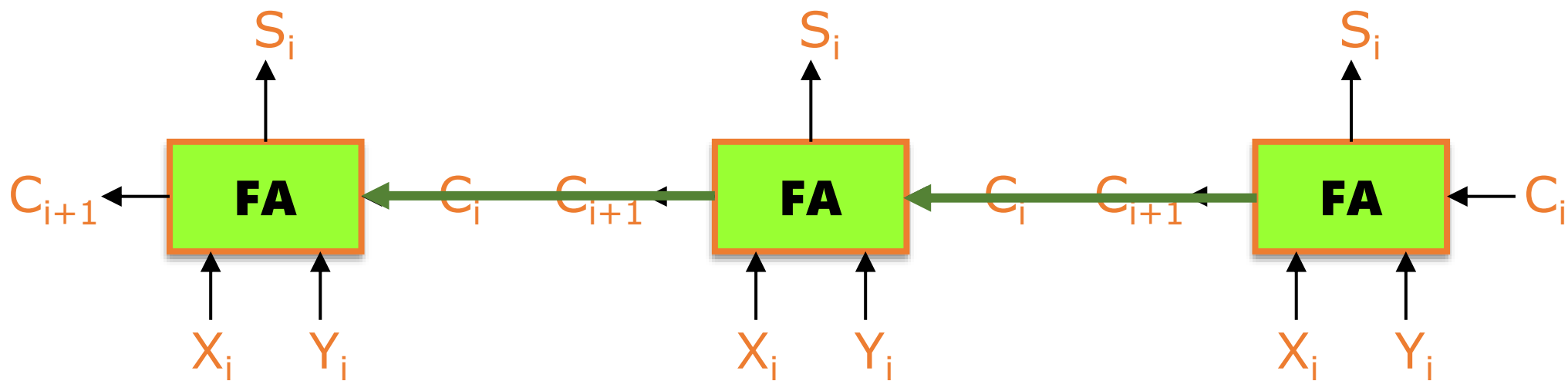


6级门电路延迟 6T



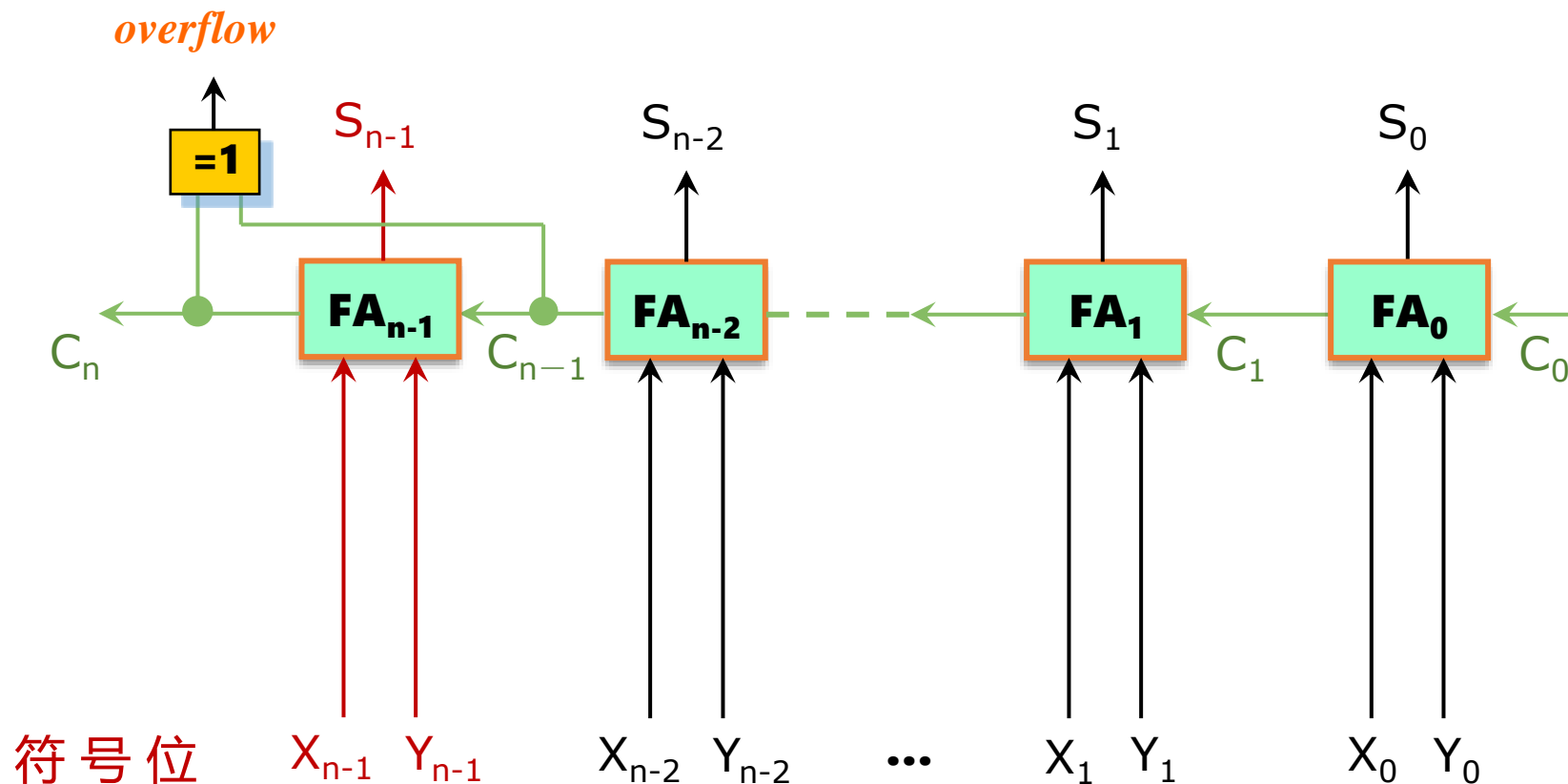
# n位加法器

- n位加法器包含n个全加器
- 将n个一位全加器串联
- 低位进位输出连接到高位进位输入





# 单符号位补码加法器电路(ripple carry adder)



- 对于无符号数的加法运算，溢出检测信号就是C<sub>n</sub>
- 而对于有符号数的溢出检测信号overflow，可以直接利用最高数值位进位和符号位进位异或得到，这两个进位信号都是中间运算结果，所以采用这种方法进行溢出检测最为方便快捷。

# 补码减法电路实现

补码减法可以变加法

$$[X]_{\text{补}} - [Y]_{\text{补}} = [X]_{\text{补}} + [-Y]_{\text{补}}$$

关键是求 $[-Y]_{\text{补}}$

方法：将 $Y_{\text{补}}$ 连同符号位一起逐位取反末位加一

$$[-Y]_{\text{补}} = [ [Y]_{\text{补}} ]_{\text{补}}$$

# 加法器的改造

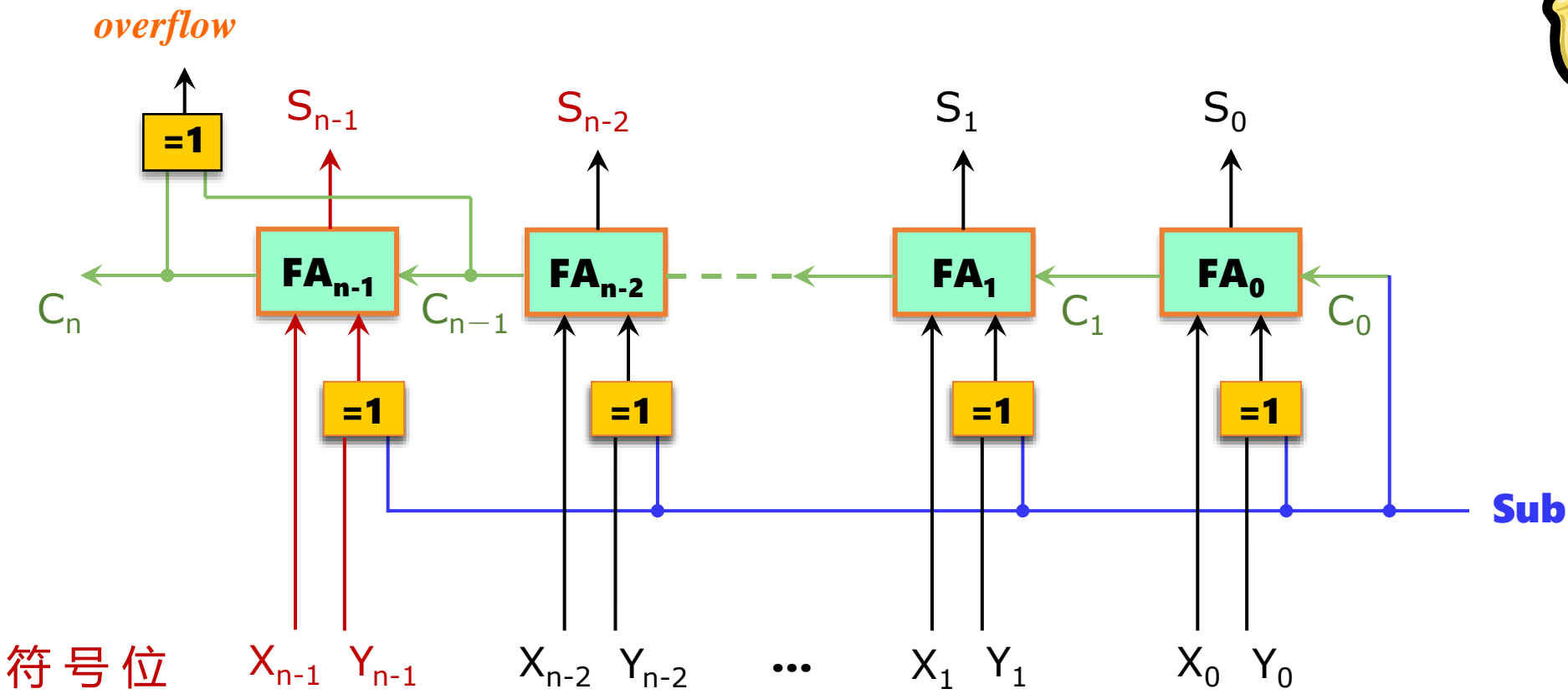
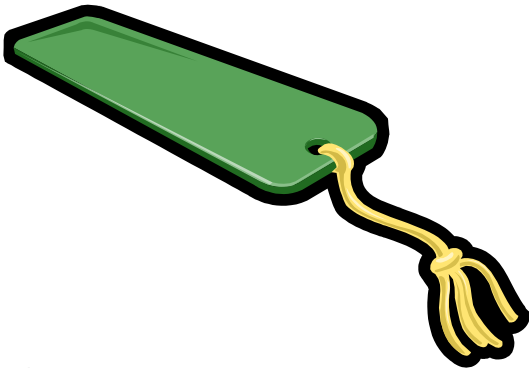
## ■ 引入运算控制位 **Sub**

- Sub=0 时作加法，送入加法器的是 $Y_{\text{补}}$
- Sub=1 时作减法，送入加法器的是 $[-Y]_{\text{补}}$ 
  - ◆ 对  $Y_{\text{补}}$  逐位取反，末位加一

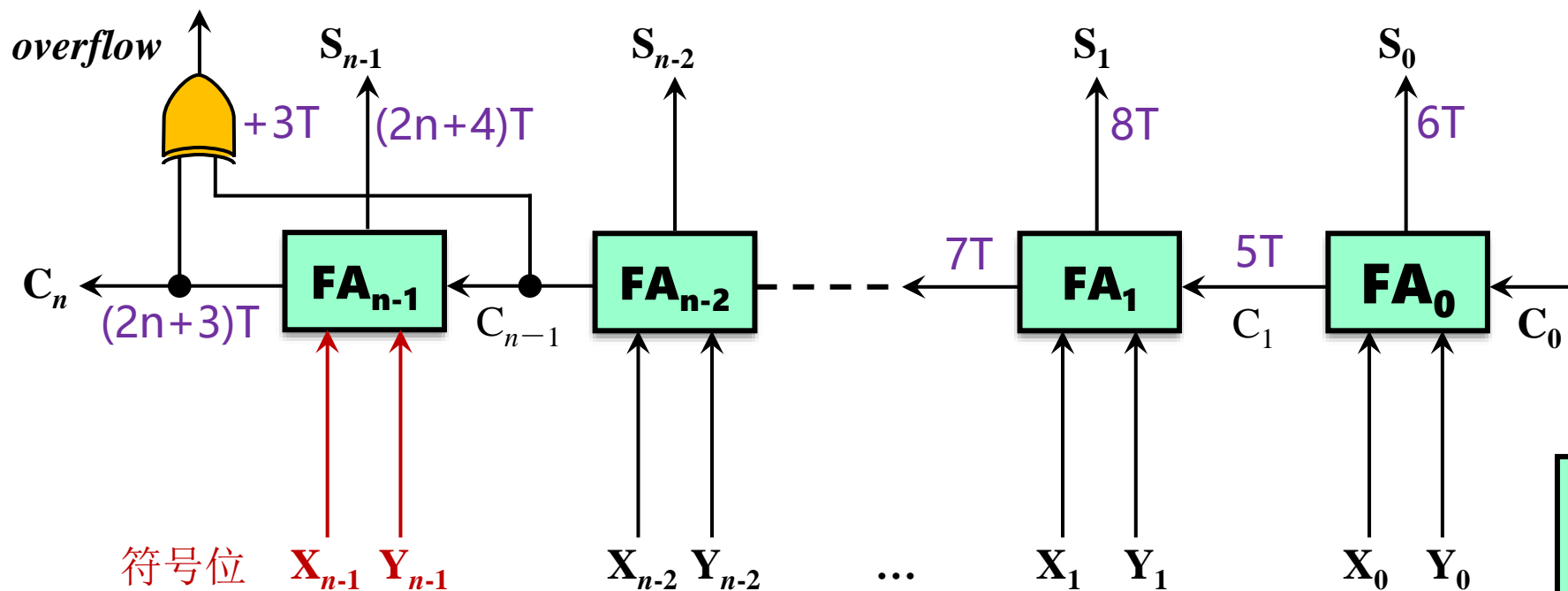
$$[-Y]_{\text{补}} = [ [Y]_{\text{补}} ]_{\text{补}}$$

$Y_i$	Sub	Input
$Y_i$	0	$Y_i$
$Y_i$	1	$\overline{Y_i}$

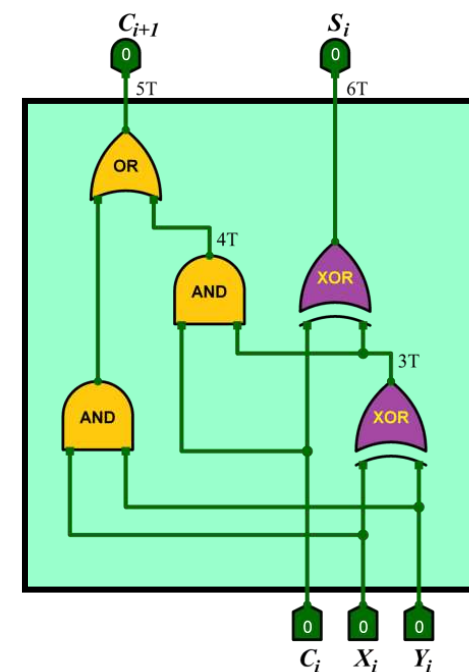
# 可控加减法电路



# 串行加法器时间延迟



- n位串行加法电路中高位的运算依赖低位进位输入 $C_i$ ，所以所有全加器不能并行运行，其时间关键延迟为 $(2n+4)T$ ，与位数呈线性关系，当位宽较大时性能较差。
- 引起这个问题的根源是进位链依赖。



# 快速加法器

- 能否提前产生各位的进位输入
- 使得各位的加法运算能并行起来
- 即可提高多位加法器运算速度



# 并行加法器进位链(carry-lookahead)

$$S_i = X_i \oplus Y_i \oplus C_i$$

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

假设  $G_i = X_i Y_i$  进位生成函数 **Generate**

假设  $P_i = X_i \oplus Y_i$  进位传递函数 **Propagate**

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

$$S_i = P_i \oplus C_i$$

## 并行加法器进位链...

$$C_{n+1} = G_n + P_n C_n$$

$$C_n = G_{n-1} + P_{n-1} C_{n-1}$$

.....

$$C_1 = G_1 + P_1 C_0$$

- 高位运算依赖于低位进位
  - 计算不能并行
- 能否提前得到各位的进位输入??



## 并行加法器进位链...

$$C_1 = \underline{G_0 + P_0 C_0}$$

$$C_2 = G_1 + P_1 \underline{C_1}$$

$$= G_1 + P_1 (G_0 + P_0 C_0) = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2$$

$$= G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_0)$$

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

## 并行加法器进位链...

$$C_n = G_{n-1} + P_{n-1}G_{n-2} + P_{n-1}P_{n-2}G_{n-3} + \dots + P_{n-1}P_{n-2}\dots P_1P_0C_0$$

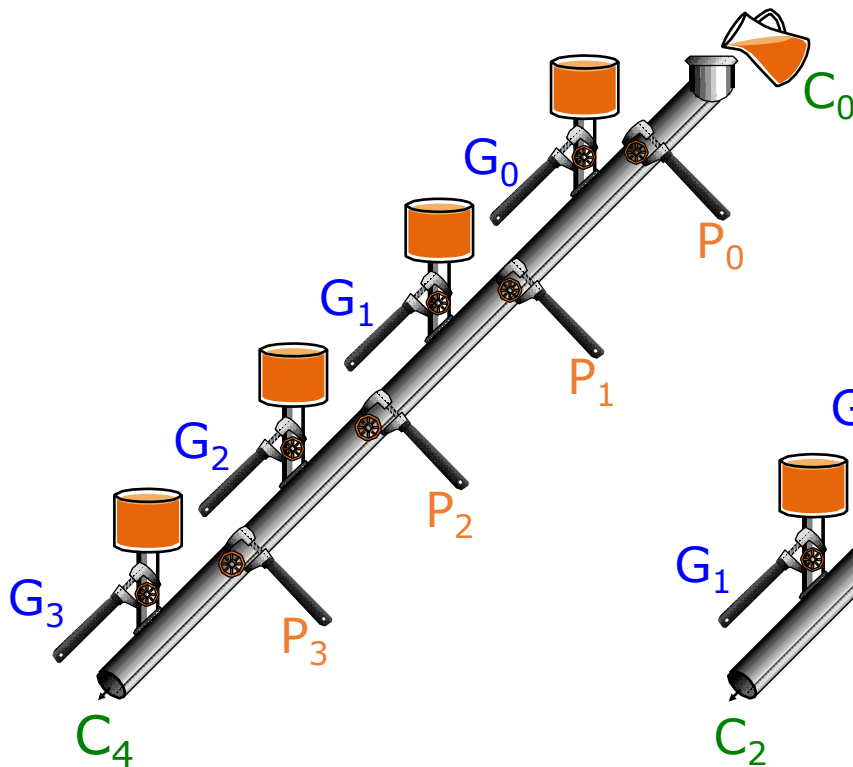
- 进位输出仅与最低位进位输入 $C_0$ 有关
- 利用额外的组合逻辑电路提前产生各位加法运算需要的所有进位输入，再利用 $s_i = P_i \oplus C_i$ ，进行一级异或门运算即可得到最终的和数，这就是先行进位的基本原理。
- 位数越长，进位链电路复杂度越高
- 通常按照**4位一组**进行分组运算

$$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

$$P_i = X_i \oplus Y_i \quad G_i = X_i Y_i \quad S_i = X_i \oplus Y_i \oplus C_i$$

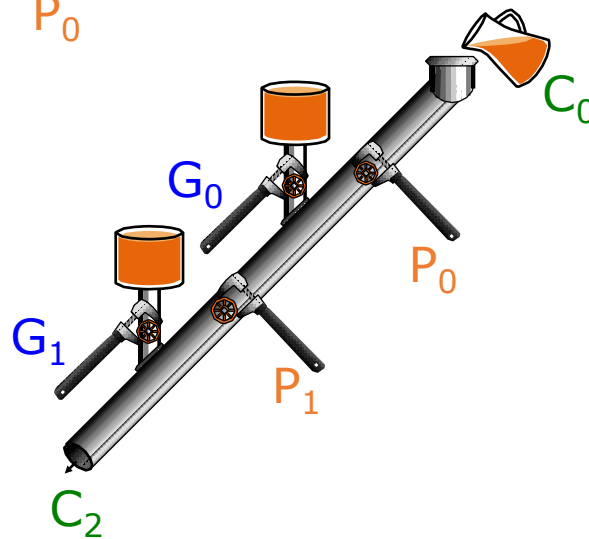
# 生成函数 & 传递函数

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

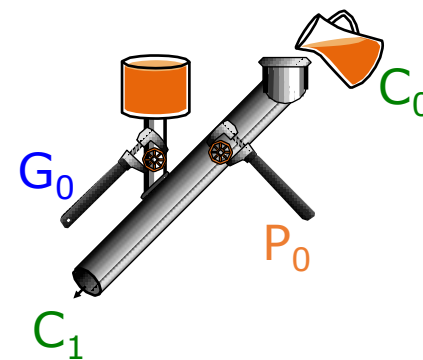


$$G_i = X_i Y_i$$

$$P_i = X_i \oplus Y_i$$

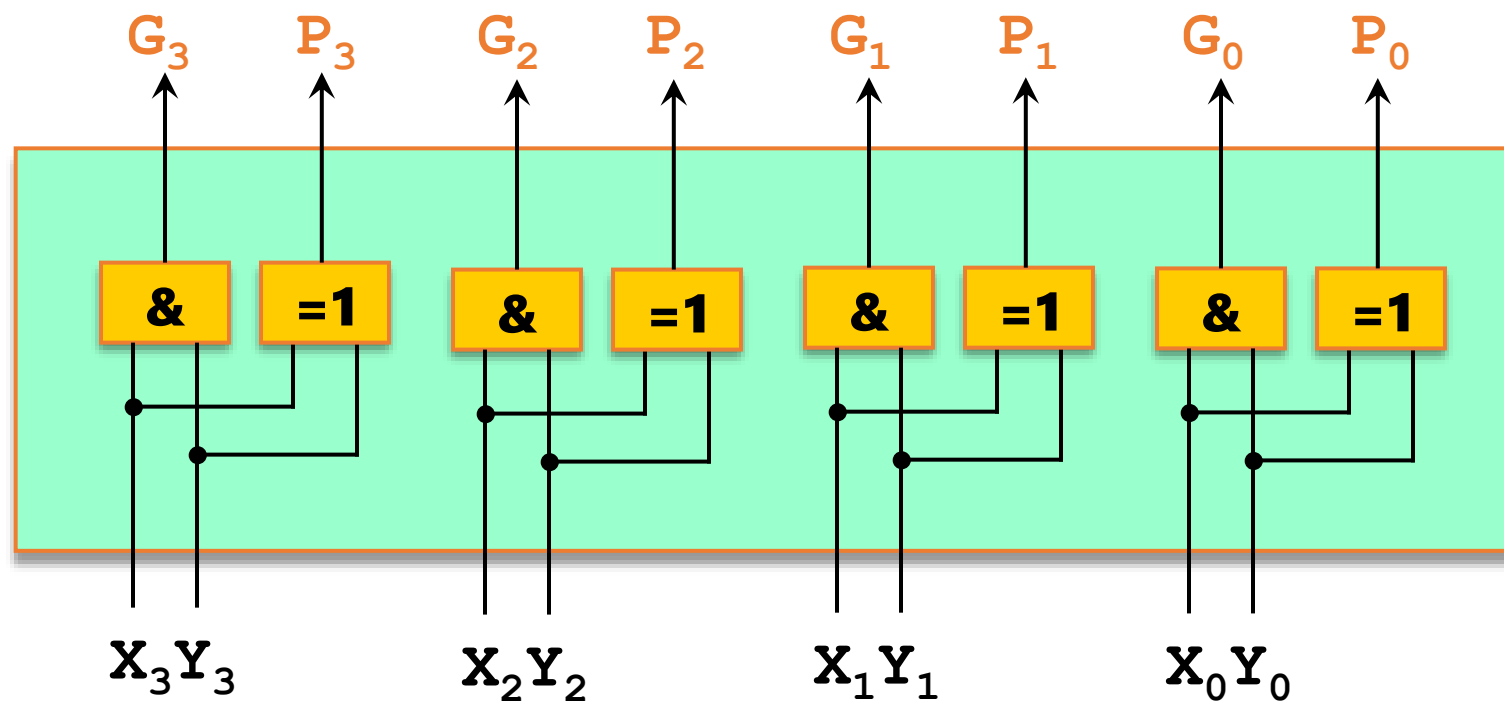


$$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$$



$$C_1 = G_0 + P_0 C_0$$

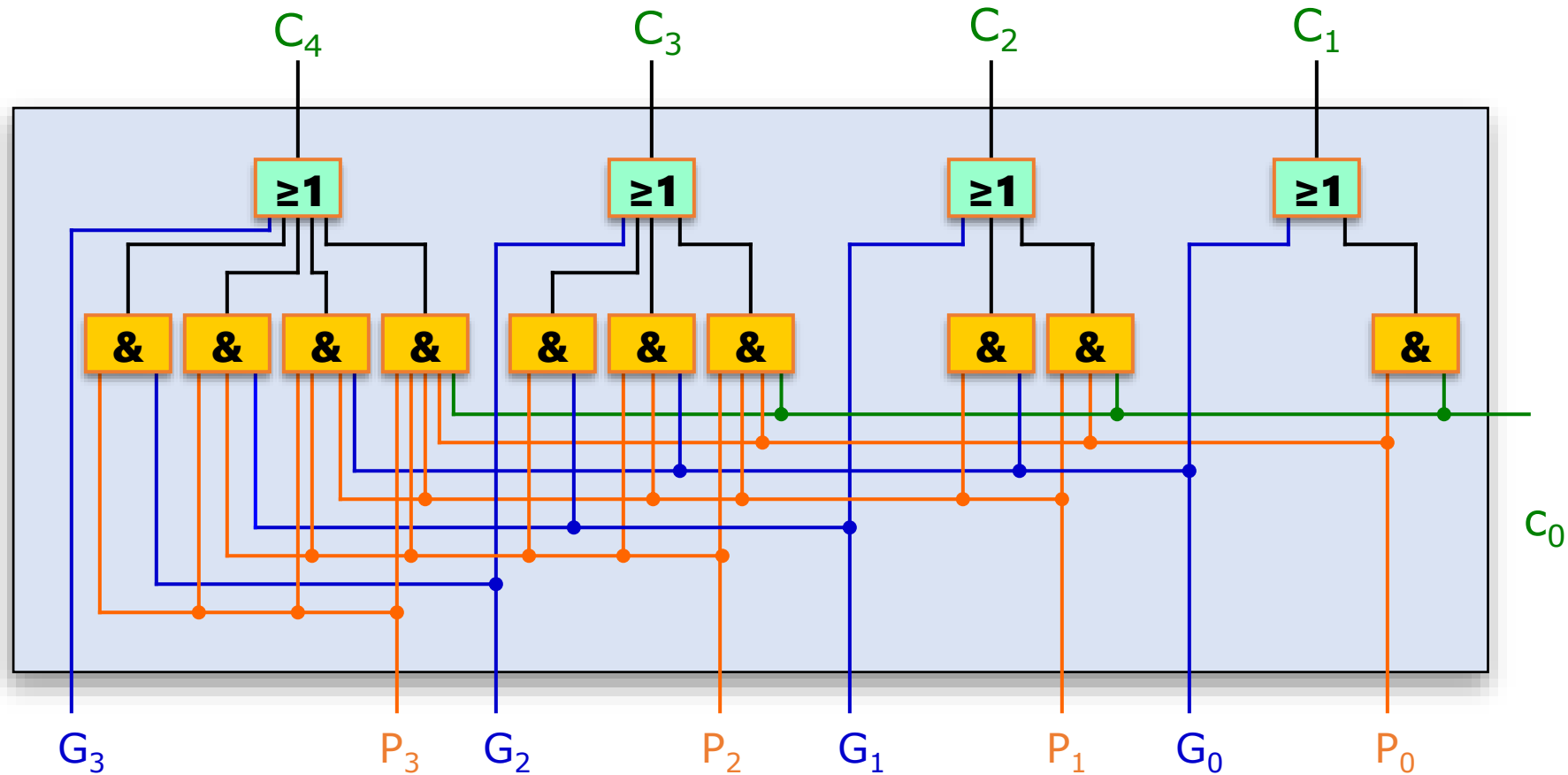
# 与门异或门电路



$$G_i = X_i Y_i$$

$$P_i = X_i \oplus Y_i$$

# 先行进位电路carry lookahead



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

# 成组进位

$$\blacksquare C_4 = \boxed{G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0} + \boxed{P_3 P_2 P_1 P_0} C_0$$

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

□ 成组进位生成函数

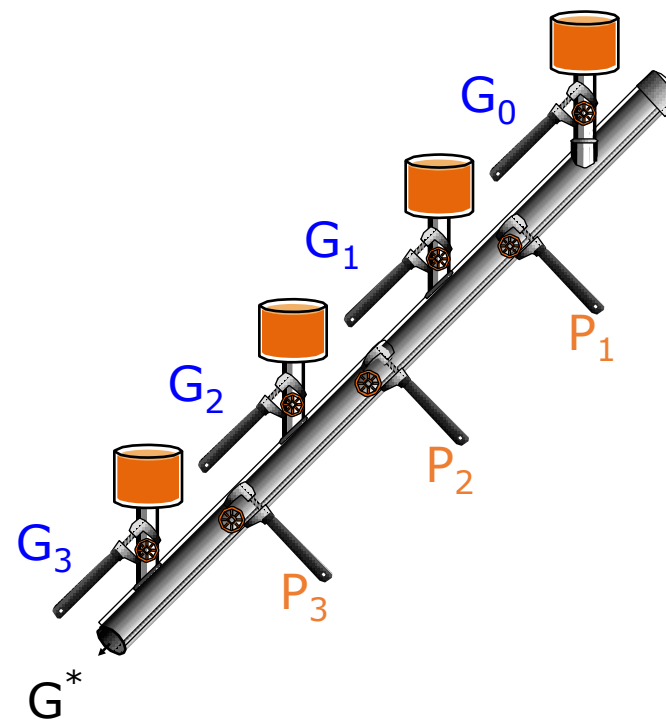
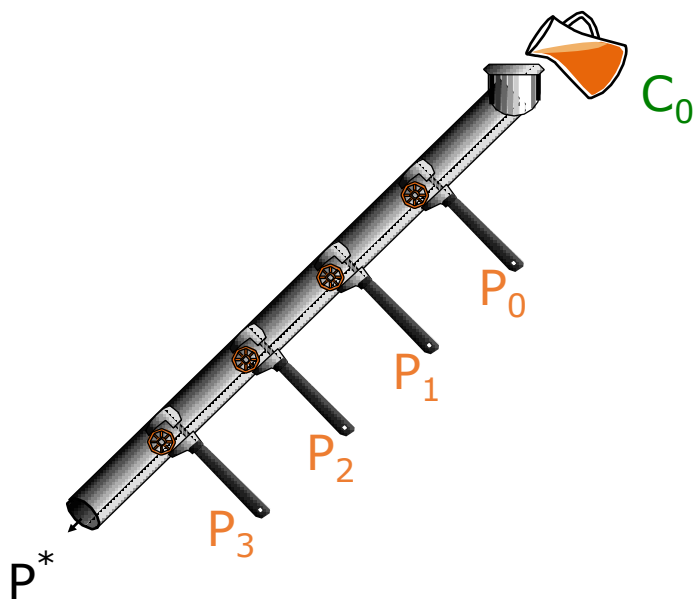
$$\blacksquare P^* = P_3 P_2 P_1 P_0$$

□ 成组进位传递函数

$$\blacksquare C_4 = G^* + P^* C_0$$

$$\blacksquare C_1 = G_0 + P_0 C_0$$

# 成组进位生成函数 & 传递函数

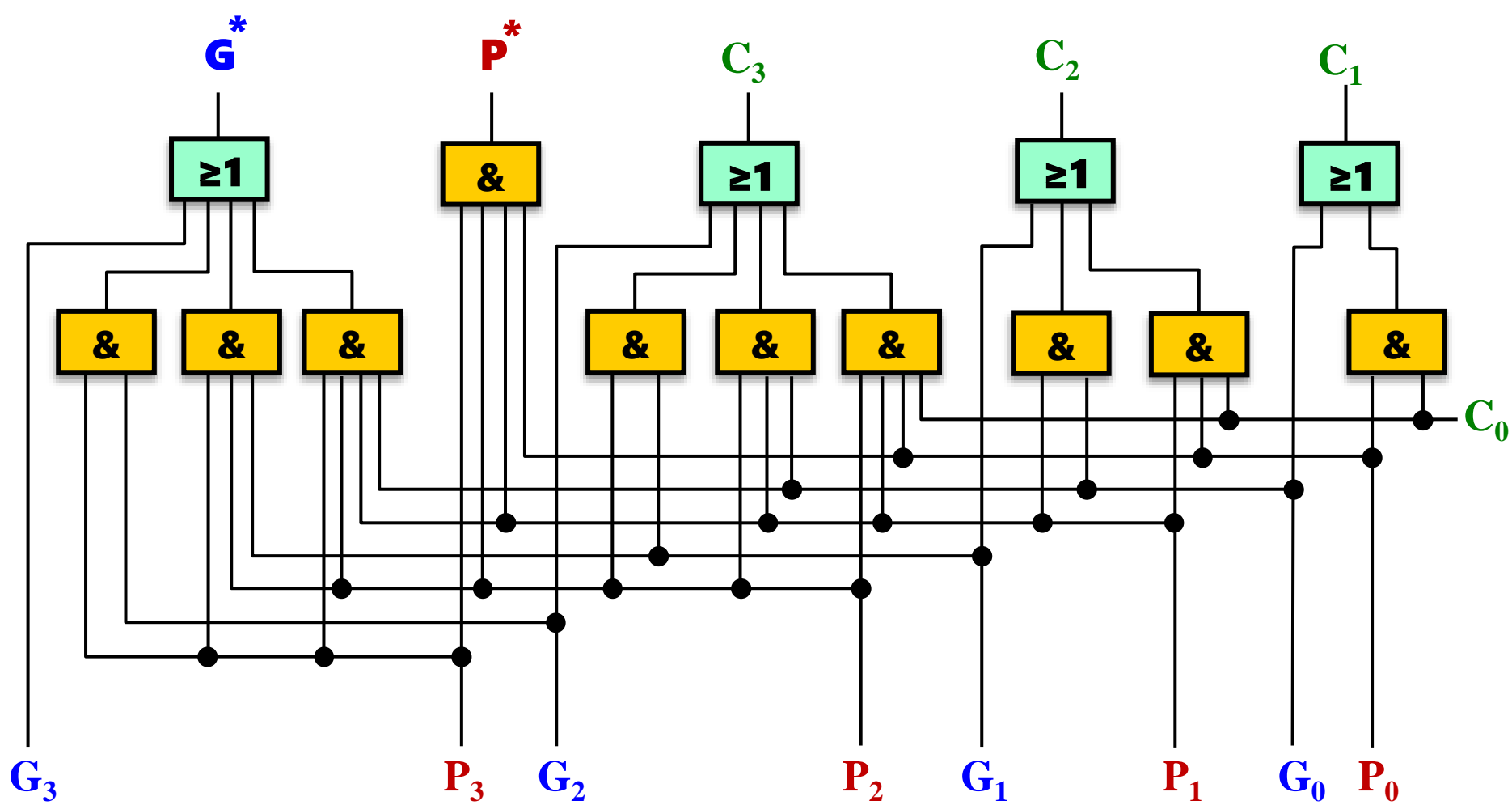


$$P^* = P_3 P_2 P_1 P_0$$

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

$$C_4 = G^* + P^* C_0$$

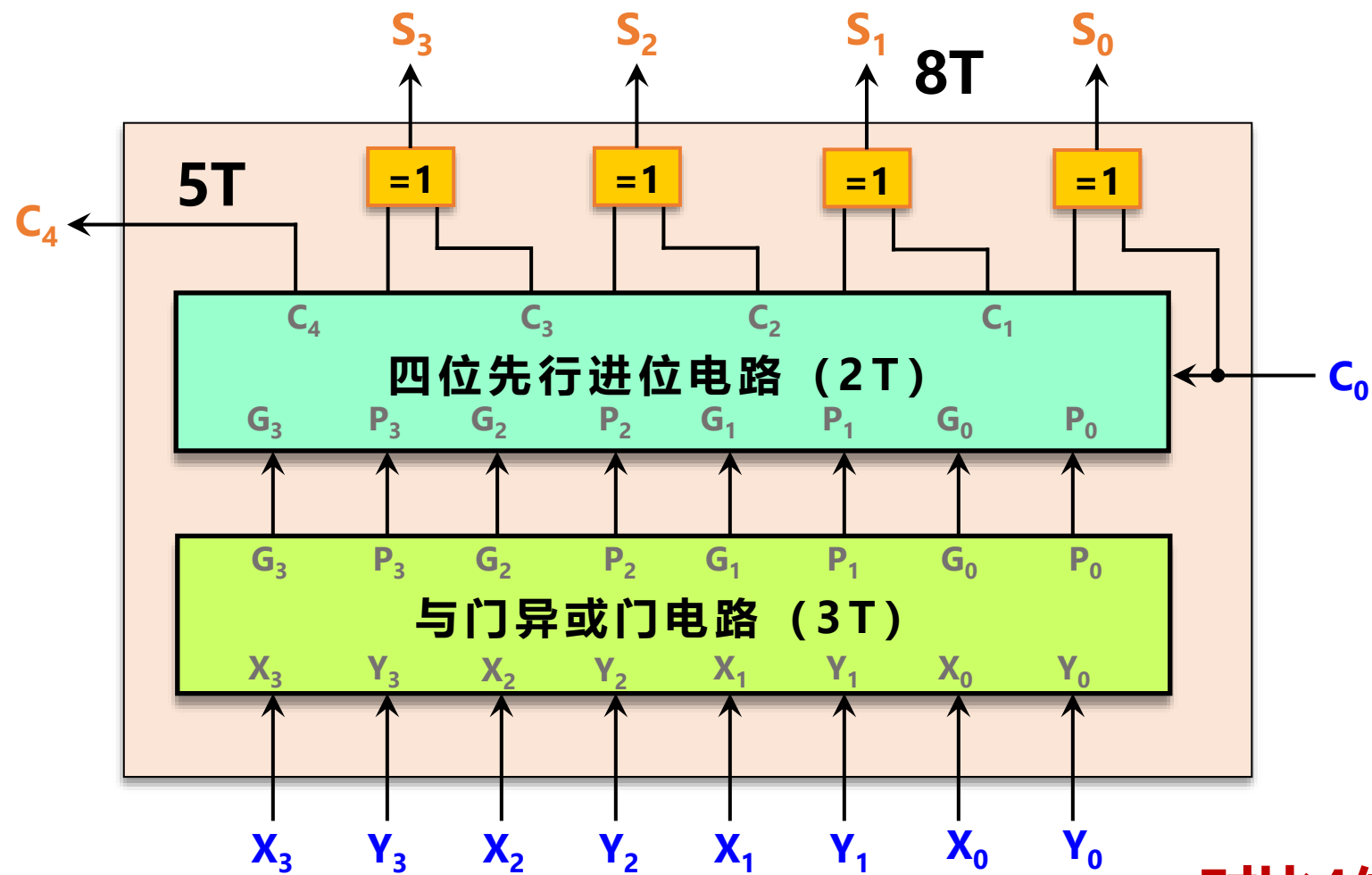
# 两级先行进位电路



2级门电路延迟2T



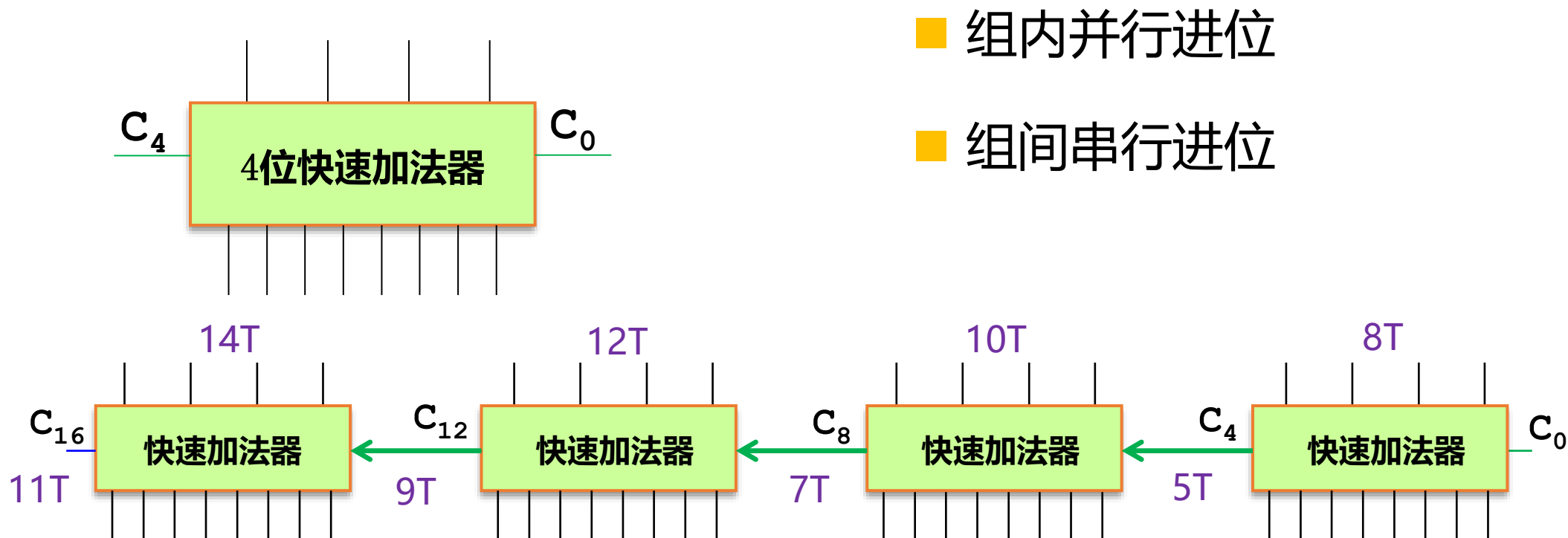
# 四位先行进位快速加法器



8级门电路延迟

对比4位串行加法器  
 $2n+4=12T$ ，节约了  
4T的时间

# 16位加法器



时间延迟为  $14T$ ，相比传统串行的加法器的  $(2n+4)T=36T$ ，其性能提升了约 2.6 倍。

# 先行进位电路 CLA74182

■ 输入:  $P_3G_3 \quad P_2G_2 \quad P_1G_1 \quad P_0G_0 \quad C_0$

■ 输出: 先行进位输出  $C_4 \quad C_3 \quad C_2 \quad C_1$

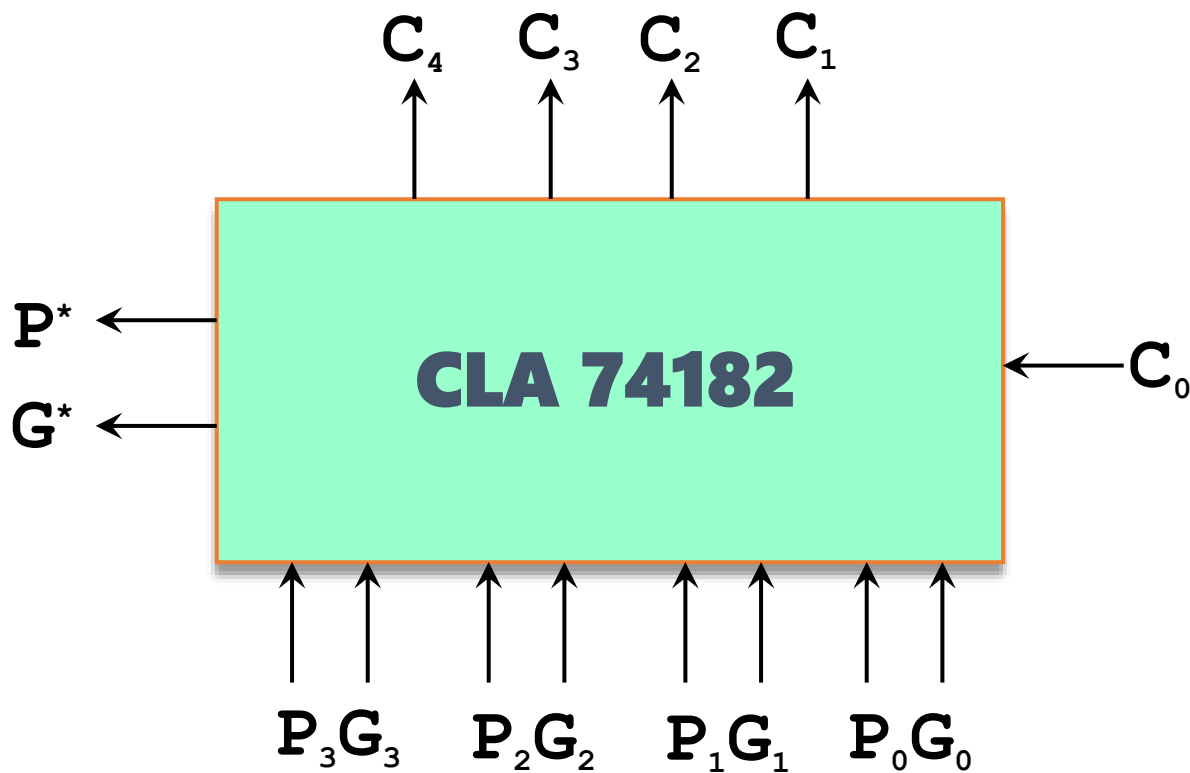
成组进位传送输出  $P^*$

成组进位发生输出  $G^*$

■  $C_n = G_{n-1} + P_{n-1}G_{n-2} + P_{n-1}P_{n-2}G_{n-3} + \dots + P_{n-1}P_{n-2}\dots P_0C_0$

■  $G_i = X_i Y_i \quad P_i = X_i \oplus Y_i$

# 先行进位芯片 CLA74182



2级门电路延迟

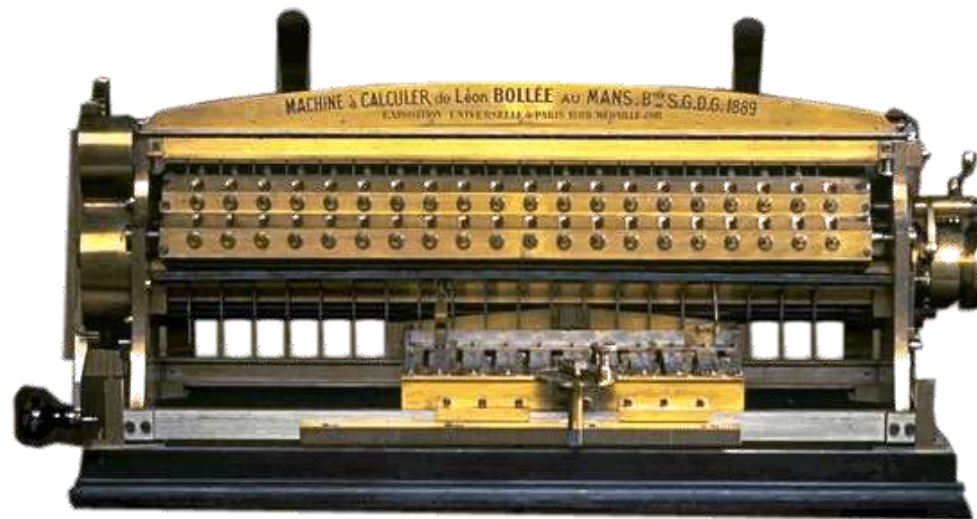
# 本章主要内容

- 3.1 计算机中的运算
- 3.2 定点加/减法运算
- **3.3 定点乘法运算**
- 3.4 定点除法运算
- 3.5 浮点运算
- 3.6 运算器



## 3.3 定点乘法运算

- 3.3.1 原码一位乘法
- 3.3.2 补码一位乘法
- 3.3.3 阵列乘法器
- 3.3.4 补码阵列乘法器
- 3.3.5 乘法器性能优化



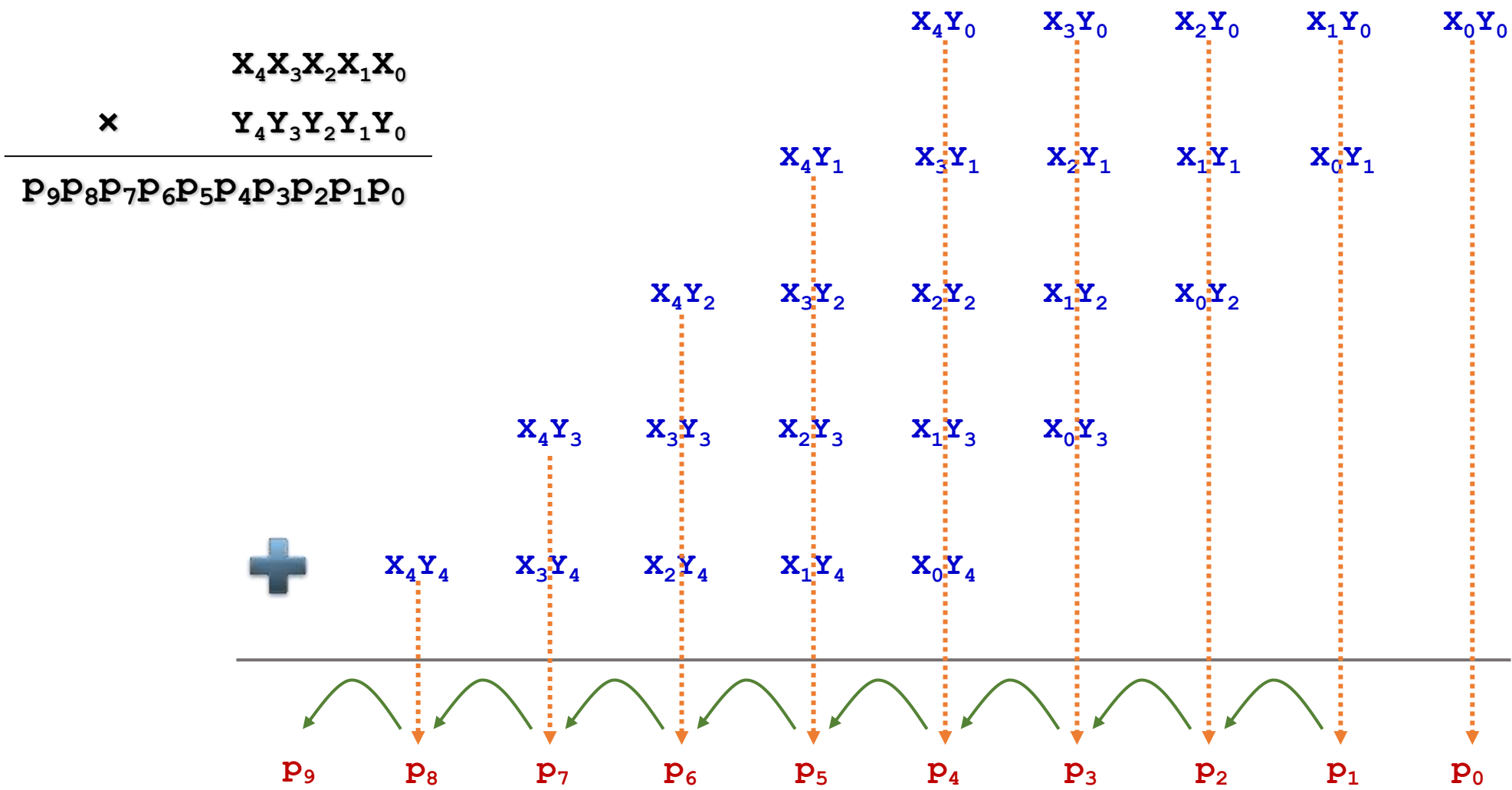
# 乘法运算实现方法

■ 从计算机硬件角度看，实现乘法运算的方法主要有以下两种。

(1)利用多位加法器循环累加实现乘法运算，这种方法硬件开销小，但必须采用时序电路进行控制，需多个时钟周期才能得到运算结果。

(2)采用加法器阵列构成的纯组合逻辑电路实现乘法运算，这种方法只需要一个时钟周期即可完成运算，但硬件开销较大。

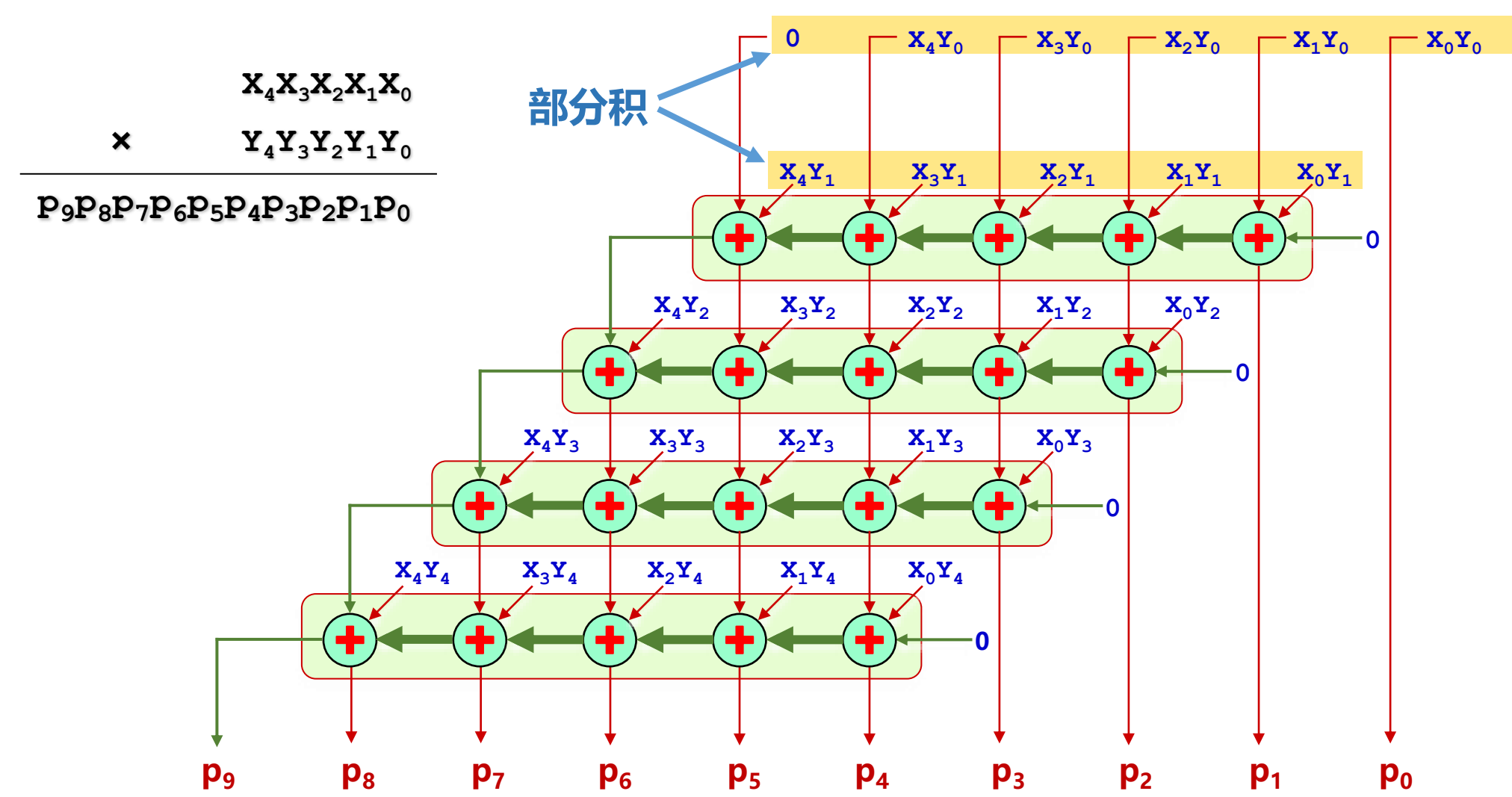
# 二进制手工乘法运算



先计算相加数，然后逐列相加



# 5个部分积相加



5\*5的横向阵列乘法器包括4行全加器，每行5个，需要20个全加器，进位信号横向传递，如图中绿色线缆部分，每一行都是一个5位串行进位加法器，串行进加法器的特点就是性能差，各全加器之间存在着进位依赖，所以这个全加器运算完毕后，这个才能运算，然后是这个，第一行运算完毕后，第二行才能开始运算，所以看上去所有全加器都只能串行工作，整个运算需要20个全加器时延。

## 5个部分积相加

The diagram illustrates a 10-bit ripple-carry multiplier circuit. It consists of four stages of 5-bit adders, each represented by a green rounded rectangle containing five circles with a red plus sign. The adders are connected in a ripple-carry fashion, with the carry-out of one stage becoming the carry-in for the next stage to its left. The multiplicand  $0.10101$  is shifted into the adders from right to left, with the least significant bit  $1$  entering the first adder. The multiplier  $0.x_4x_3x_2x_1x_0$  is applied to the adders from top to bottom, with  $x_0$  entering the first adder,  $x_1$  entering the second,  $x_2$  entering the third, and  $x_3$  entering the fourth. The final product bits  $p_9$  to  $p_0$  are shown at the bottom, with  $p_9$  being the final carry-out of the fourth stage.

## 部分积累加的数学表示

$$\blacksquare \Sigma = XY_n$$

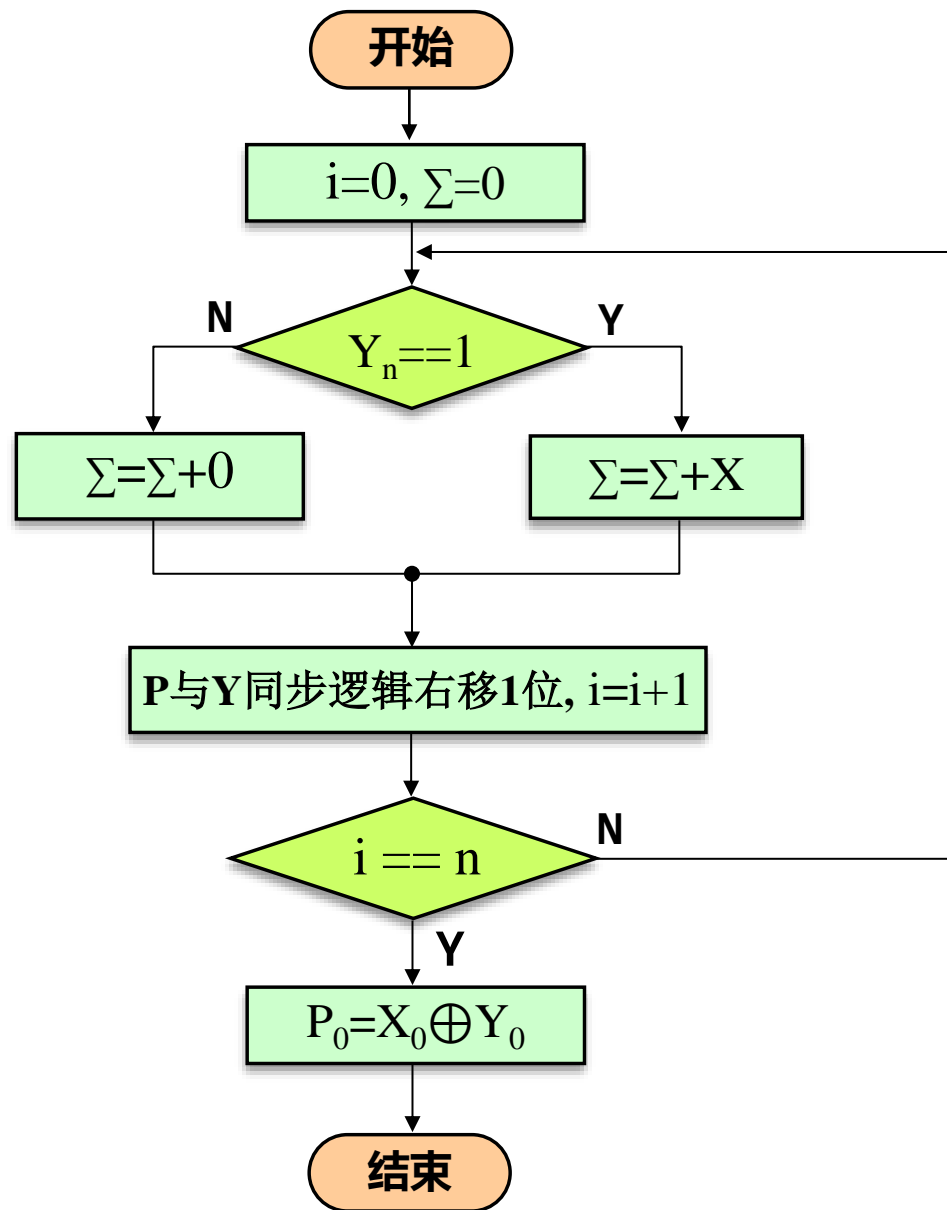
$$\blacksquare \Sigma = XY_{n-1} + XY_n * 2^{-1}$$

$$\begin{aligned}\blacksquare \Sigma &= XY_{n-2} + (XY_{n-1} + XY_n * 2^{-1}) 2^{-1} \\ &= XY_{n-2} + XY_{n-1} 2^{-1} + XY_n * 2^{-2} \\ &= XY_1 * 2^{-1} + XY_2 * 2^{-2} + \dots XY_n * 2^{-n}\end{aligned}$$

$$X * Y = X [Y_1 * 2^{-1} + Y_2 * 2^{-2} + \dots Y_n * 2^{-n}]$$

# 原码乘法算法流程图

- 作完加法，一定移位
- $n$ 次加法
- 符号位单独计算



# 手工计算举例



$X = +0.1101$

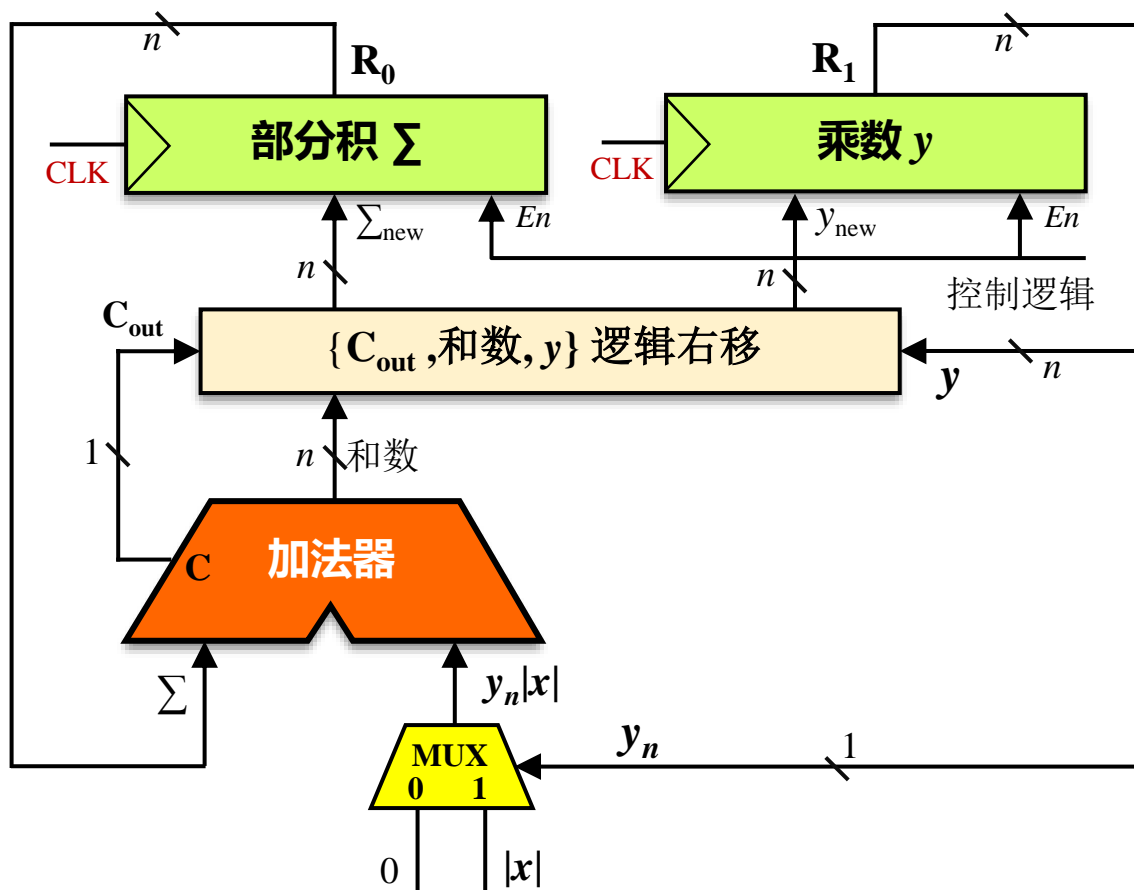
$Y = -0.0011$

求  $[X]_{原} \times [Y]_{原}$

- 符号位和数据位分开运算
- $y_n$  决定累加  $|x|$  是 0
- 连同进位位一起右移
- 可能溢出，移位后正常

$\Sigma=0$	00.0000		乘数判断位 $y_n$
	+ 00.1101		0.001 <u>1</u>
	00.1101		
→	00.0110	1	0.00 <u>1</u>
	+ 00.1101		
	01.0011	1	
→	00.1001	11	0.00 <u>0</u>
	+ 00.0000		
	00.1001	11	
→	00.0100	111	0.0 <u>0</u>
	+ 00.0000		
	00.0100	111	
→	00.0010	0111	← 最终乘积

# 原码一位乘法硬件实现

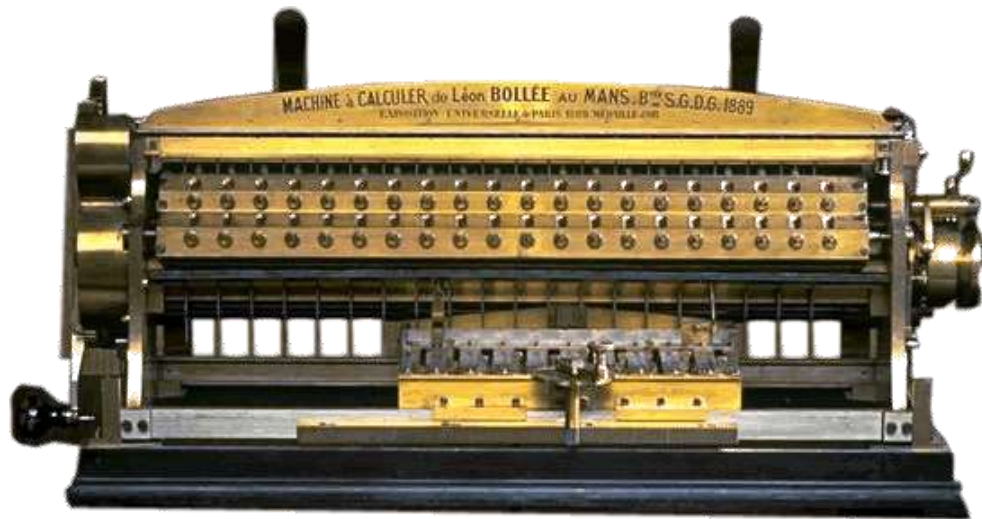


- $R_0$ 存放 $\Sigma$ 高 $n$ 位，初值为0
- $R_1$ 存放 $Y$ 、 $\Sigma$ 其它位（如何载入 $Y$ 初值）
- 运算结果右移后送寄存器输入
- 时钟到来， $R_0$ ， $R_1$ 锁存新值
- 状态机控制使能信号停机
- 停机后**最终乘积**存放在 $R_0$ ， $R_1$ 中

$$\{\Sigma, Y\} = \{\Sigma + Y_n | X|, Y\} / 2 \quad \text{连同进位位右移}$$

## 3.3 定点乘法运算

- 3.3.1 原码一位乘法
- 3.3.2 补码一位乘法
- 3.3.3 阵列乘法器
- 3.3.4 补码阵列乘法器
- 3.3.5 乘法器性能优化



- 计算机中采用补码表示数据，如果用原码乘法计算两个数的乘积，运算前后还需要进行补码和原码之间的转换
- 为减少处理环节，人们提出了补码乘法，该方法由英国人布斯于1950年发明，又称为**布斯(Booth)算法**。



# 补码一位乘法

1) 被乘数X符号任意，乘数Y为正

$$[X]_{\text{补}} = X_0.X_1X_2\dots X_n \quad [Y]_{\text{补}} = 0.Y_1Y_2\dots Y_n$$

$$[X]_{\text{补}} \times [Y]_{\text{补}} = (2 + X) \times Y = (2^{n+1} + X) \times Y$$

$$= 2^{n+1}Y + XY$$

$$= 2 \times 2^n \times 0.Y_1Y_2\dots Y_n + XY$$

$$= 2(Y_1Y_2\dots Y_n) + XY$$

$$= 2 + XY$$

$$= [X \times Y]_{\text{补}}$$

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

## 补码一位乘法

2) 被乘数[X]符号任意, 乘数[Y]为负数

$$[X]_{\text{补}} = X_0.X_1X_2\dots X_n \quad [Y]_{\text{补}} = 1.Y_1Y_2\dots Y_n$$

$$[Y]_{\text{补}} = 2 + Y \quad Y = [Y]_{\text{补}} - 2 = 0.Y_1Y_2\dots Y_n - 1$$

$$\begin{aligned} [X \times Y]_{\text{补}} &= [X \times (0.Y_1Y_2\dots Y_n - 1)]_{\text{补}} \\ &= [X \times 0.Y_1Y_2\dots Y_n - X]_{\text{补}} \\ &= [X \times 0.Y_1Y_2\dots Y_n]_{\text{补}} - [X]_{\text{补}} \\ &= [X]_{\text{补}} \times 0.Y_1Y_2\dots Y_n - [X]_{\text{补}} \\ &= [X]_{\text{补}} \times 0.Y_1Y_2\dots Y_n - Y_0[X]_{\text{补}} \end{aligned}$$

## 补码一位乘法

$$[X \times Y]_{\text{补}} = [X]_{\text{补}} \times 0.Y_1Y_2\dots Y_n - Y_0[X]_{\text{补}}$$

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

$$= [X]_{\text{补}} \times (-Y_0 + Y_12^{-1} + Y_22^{-2} + \dots Y_n2^{-n})$$

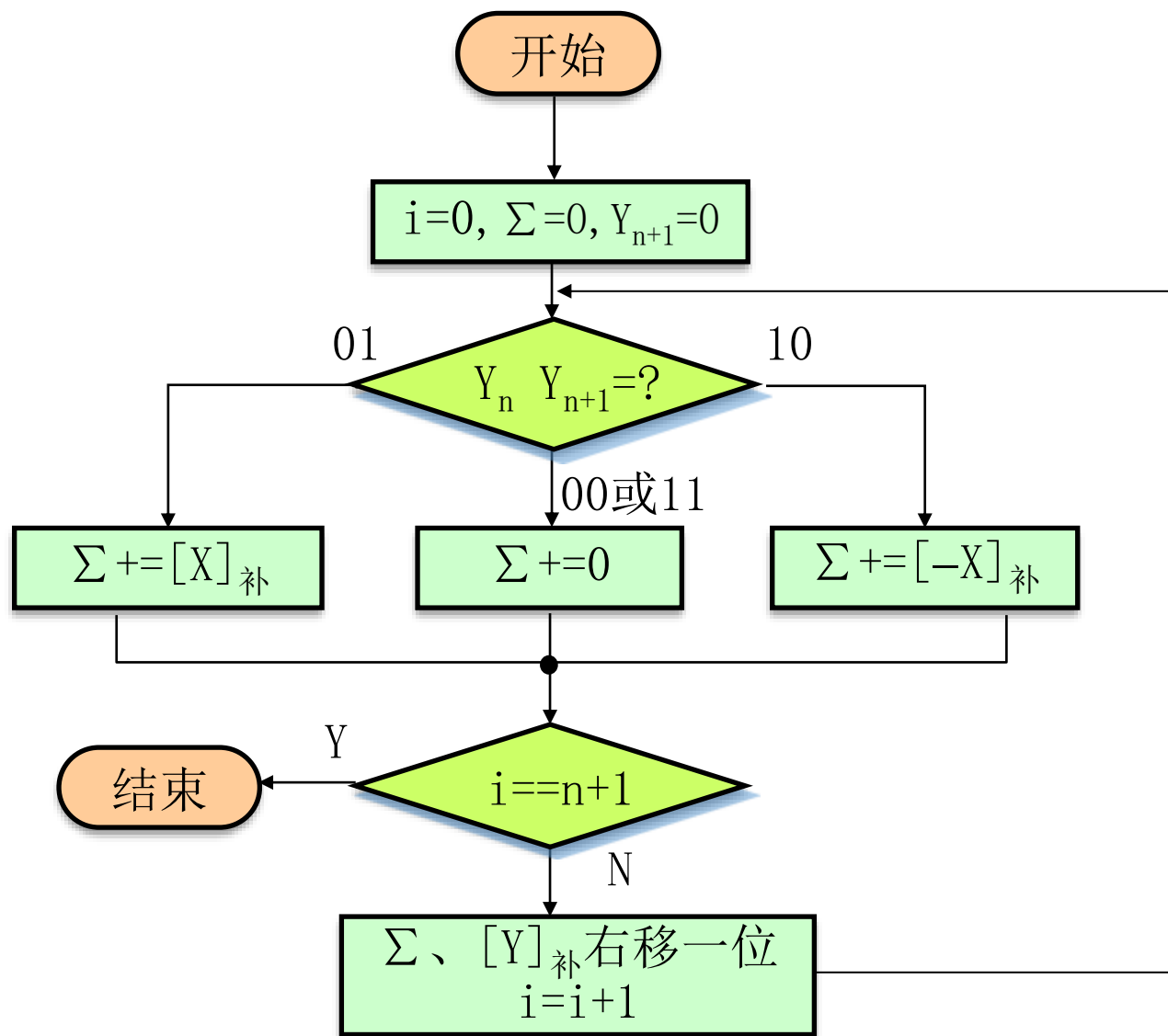
$$= [X]_{\text{补}} \times [-Y_0 + (Y_1 - Y_12^{-1}) + (Y_22^{-1} - Y_22^{-2}) + \dots (Y_n2^{-n+1} - Y_n2^{-n})]$$

$$= [X]_{\text{补}} \times [Y_1 - Y_0 + (Y_2 - Y_1)2^{-1} + (Y_3 - Y_2)2^{-2} + \dots (0 - Y_n)2^{-n}]$$

■  $XY = X \times [Y_12^{-1} + Y_22^{-2} + \dots Y_n2^{-n}]$

原码乘法

# 补码一位乘法流程图 (booth一位乘法)



- $n+1$ 次加法
- $n$ 次移位
- 符号位参与运算
- 不需单独计算符号位

# 补码乘法举例



$[X]_{补} = 0.1101$

$[Y]_{补} = 1.1101$

求  $[X]_{补} \times [Y]_{补}$

■  $[-X]_{补} = 1.0011$

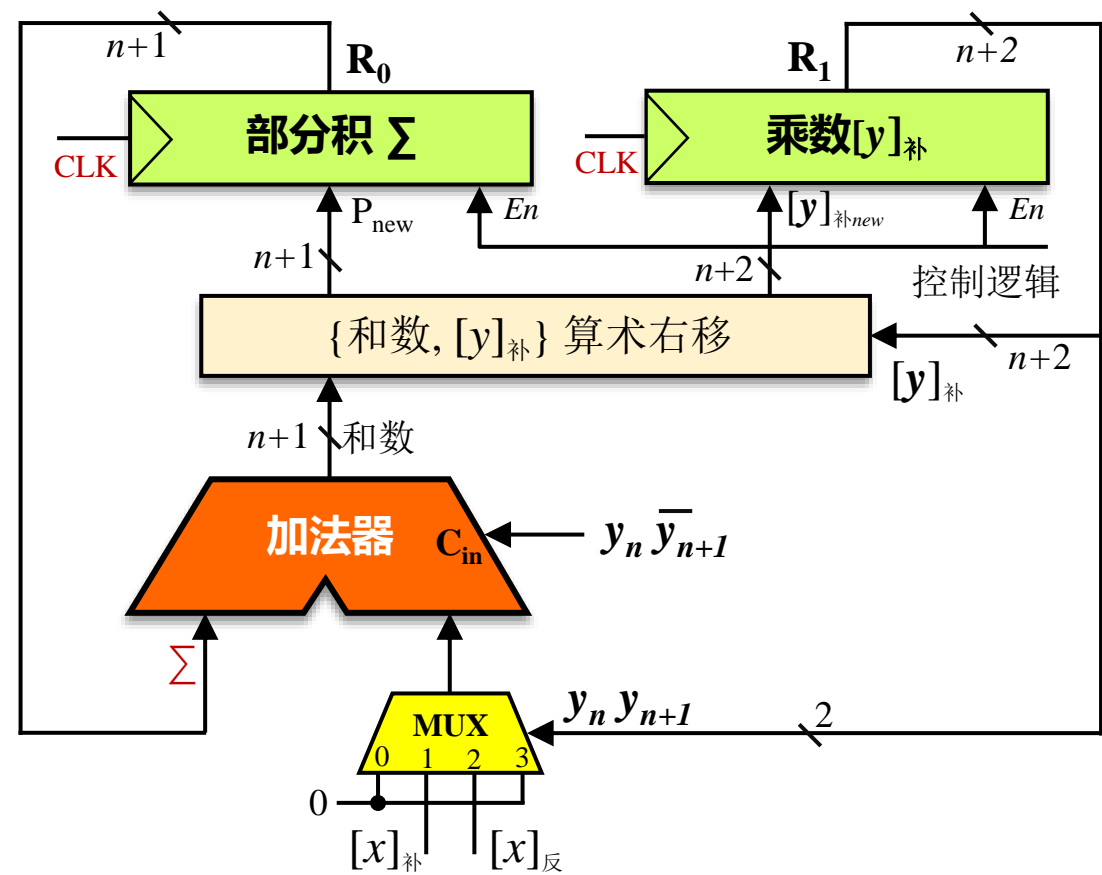
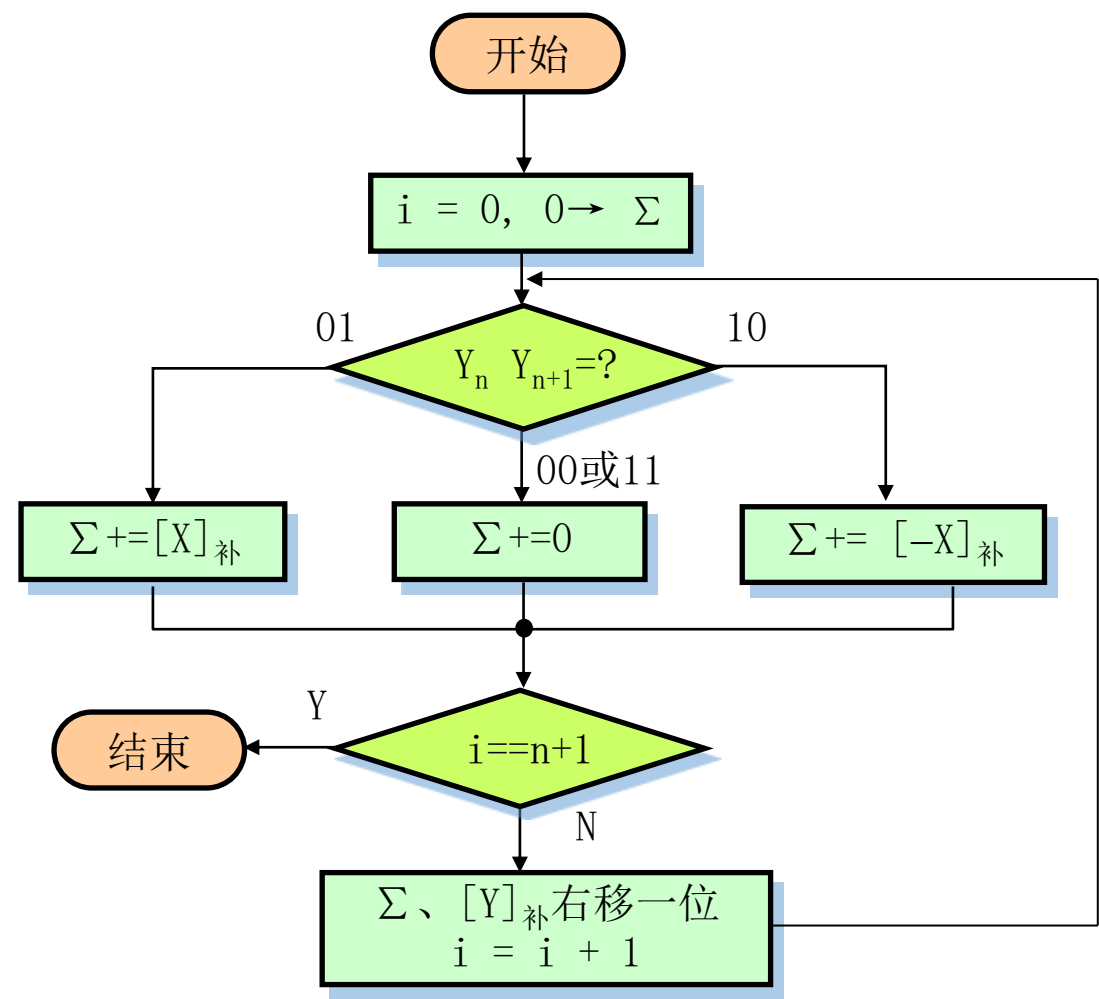
■  $Y_{n+1}Y_n$  决定累加值

□  $\Sigma+ = (Y_{n+1}-Y_n)[X]_{补}$

■ 算术右移

$\Sigma=0$	00.0000		乘数判断位 $Y_nY_{n+1}$
	+ 11.0011		1.110 <u>10</u>
	11.0011		
→	11.1001	1	.111 <u>01</u>
	+ 00.1101		
	00.0110	1	
→	00.0011	01	.11 <u>10</u>
	+ 11.0011		
	11.0110	01	
→	11.1011	001	.1 <u>11</u>
	+ 00.0000		
	11.1011	001	
→	11.1101	1001	. <u>11</u>
	+ 00.0000		
	11.1101	1001	← 最终乘积

# 补码一位乘法流程与硬件逻辑



$\{\Sigma, Y\} = \{\Sigma + (Y_{n+1} - Y_n)[X]_{\text{补}}, Y\} / 2$  算术右移

## 习题

$X=0.1001$ ,  $Y=-0.1011$ ,  $XY$ 的原码一位乘法?

$X=0.1001$ ,  $Y=-0.1011$ ,  $XY$ 的补码一位乘法?

# 习题

$X=0.1001$ ,  $Y=-0.1011$ ,  $XY$ 的原码一位乘法?

部分积	乘数	说明
0 0.0 0 0 0	1 0 1 1	初始值
0 0.1 0 0 1		+X
0 0.1 0 0 1		<u>1</u> → (右移 1 位)
0 0.0 1 0 0	1 1 0 1	
0 0.1 0 0 1		+X
0 0.1 1 0 1		<u>1</u> →
0 0.0 1 1 0	1 1 1 0	
0 0.0 0 0 0		+0
0 0.0 1 1 0		<u>1</u> →
0 0.0 0 1 1	0 1 1 1	
0 0.1 0 0 1		+X
0 0.1 1 0 0		<u>1</u> →
0 0.0 1 1 0	0 0 1 1	
$ X \cdot Y  = 0.1001 \times 0.1011 = 0.01100011$ $X \cdot Y = -0.01100011$		



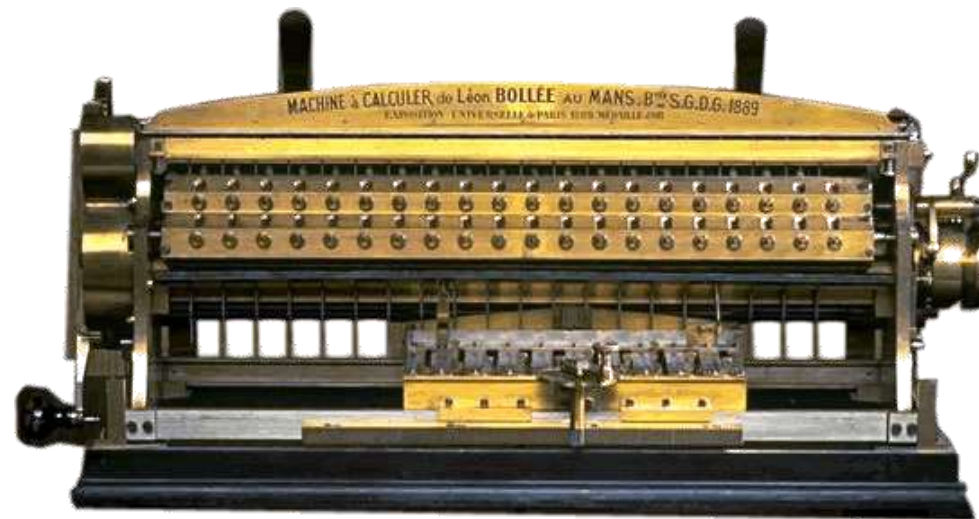
# 习题

$X=0.1001$ ,  $Y=-0.1011$ ,  $XY$ 的补码一位乘法?

部分积	乘数	说明
0 0.0 0 0 0	1 0 1 0 <u>1</u> 0	初始值, 乘数最后位补一个 0
1 1.0 1 1 1		$+[-X]_{\text{补}}$
1 1.0 1 1 1		<u>1</u> $\rightarrow$ (右移 1 位)
1 1.1 0 1 1	1 1 0 1 <u>0</u> 1	
0 0.1 0 0 1		$+ [X]_{\text{补}}$
0 0.0 1 0 0		<u>1</u> $\rightarrow$
0 0.0 0 1 0	0 1 1 0 <u>1</u> 0	
1 1.0 1 1 1		$+ [-X]_{\text{补}}$
1 1.1 0 0 1		<u>1</u> $\rightarrow$
1 1.1 1 0 0	1 0 1 1 <u>0</u> 1	
0 0.1 0 0 1		$+ [X]_{\text{补}}$
0 0.0 1 0 1		<u>1</u> $\rightarrow$
0 0.0 0 1 0	1 1 0 1 <u>1</u> 0	
1 1.0 1 1 1		$+ [-X]_{\text{补}}$
1 1.1 0 0 1	1 1 0 1	

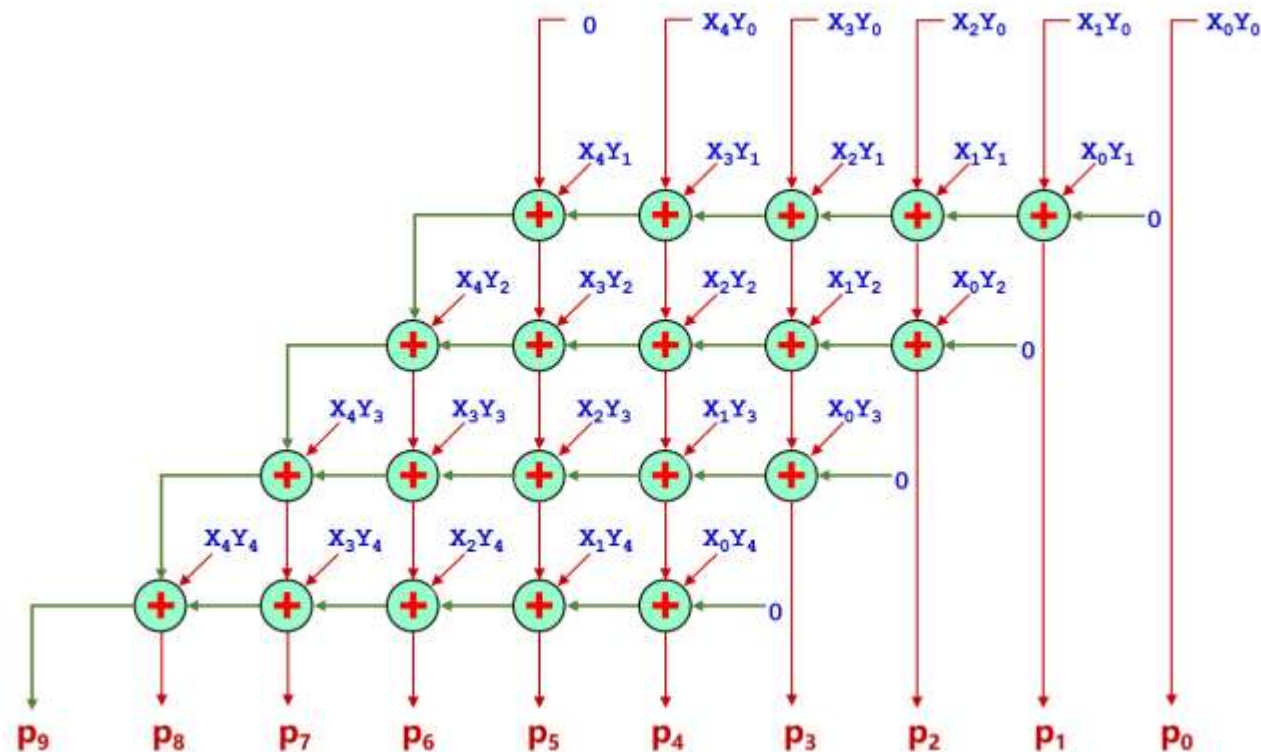
## 3.3 定点乘法运算

- 3.3.1 原码一位乘法
- 3.3.2 补码一位乘法
- **3.3.3 阵列乘法器**
- 3.3.4 补码阵列乘法器
- 3.3.5 乘法器性能优化

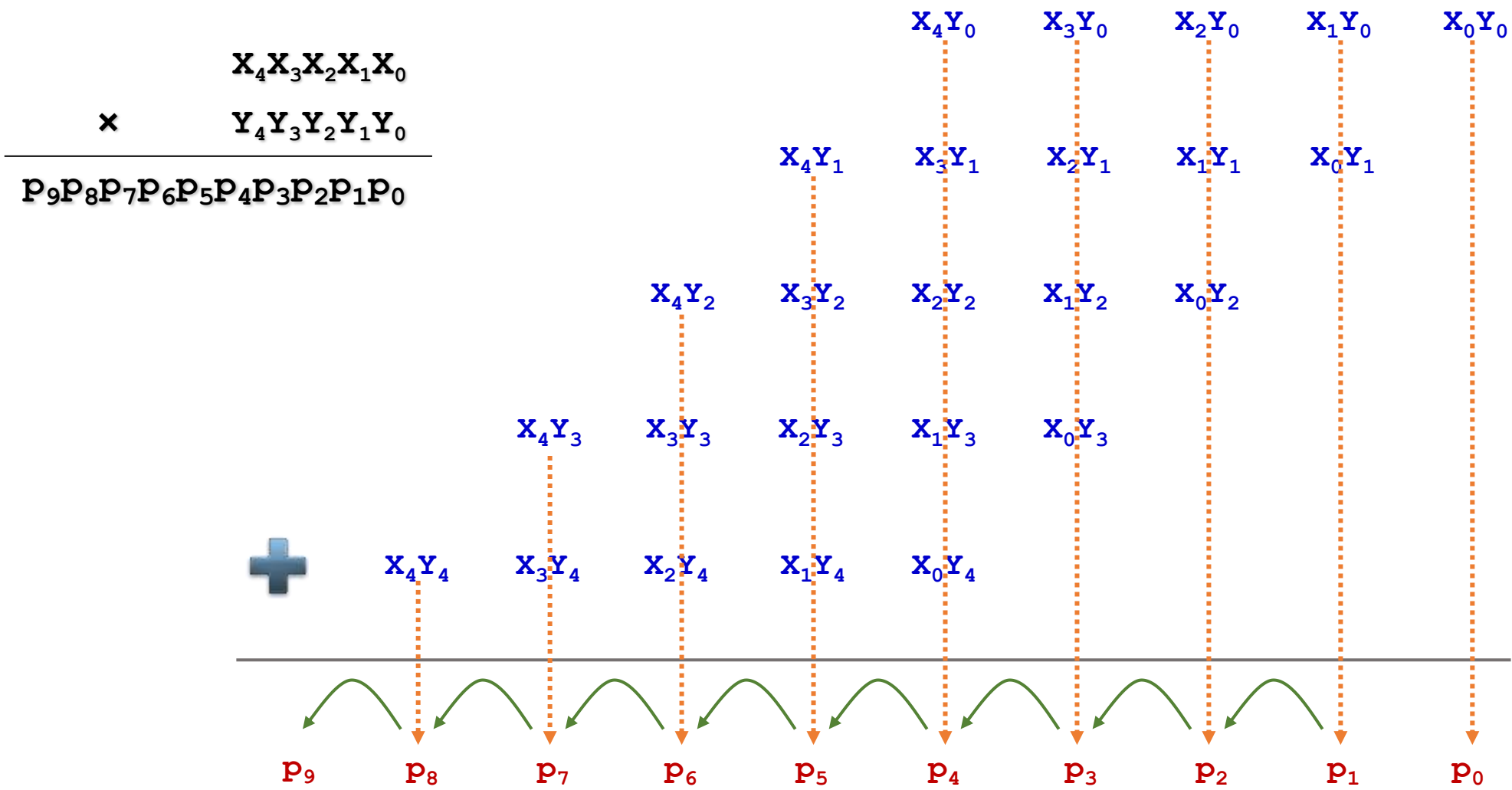


# 阵列乘法器

- 原码、补码一位乘法主要是通过加法器的循环累加计算多个位积和求解乘积的，速度较慢。
- 为提高多个位积求和的速度，可以采用**硬件的方式**实现阵列乘法器。其基本思想是采用类似**手动乘法运算**的方法，用大量**与门阵列**同时产生手动乘法中的各乘积项,同时将**大量一位全加器**按照手动乘法运算的需要构成全加器阵列。



# 二进制手工乘法运算



先计算相加数，然后逐列相加

# 一位乘法逻辑实现

■  $R = X * Y$

□  $1 \times 1 = 1$

□  $1 \times 0 = 0$

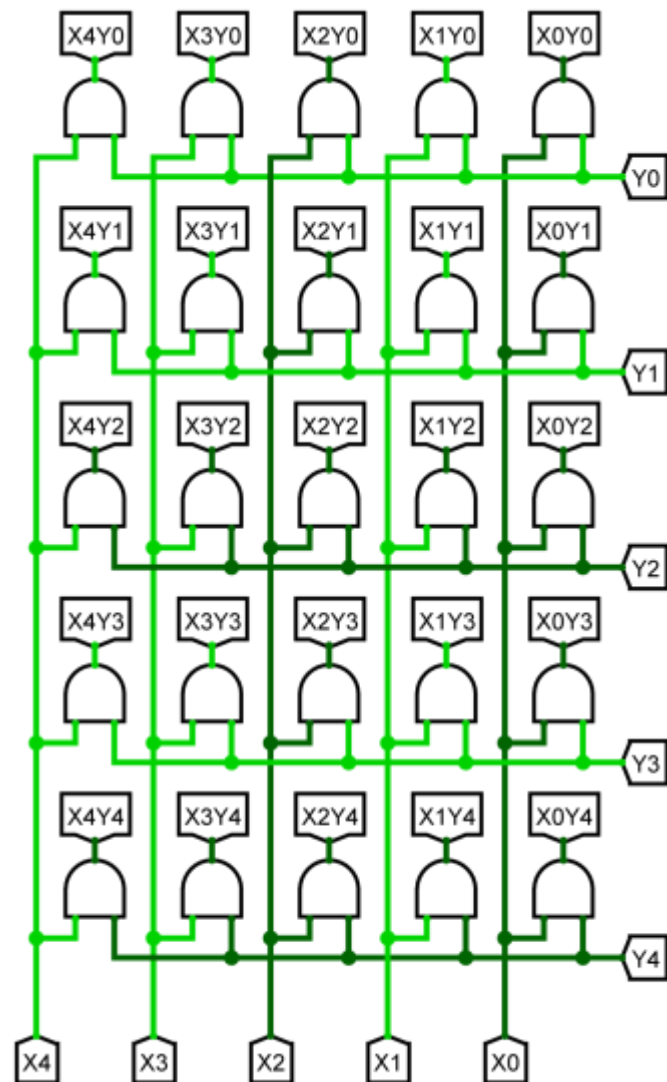
□  $0 \times 1 = 0$

□  $0 \times 0 = 0$

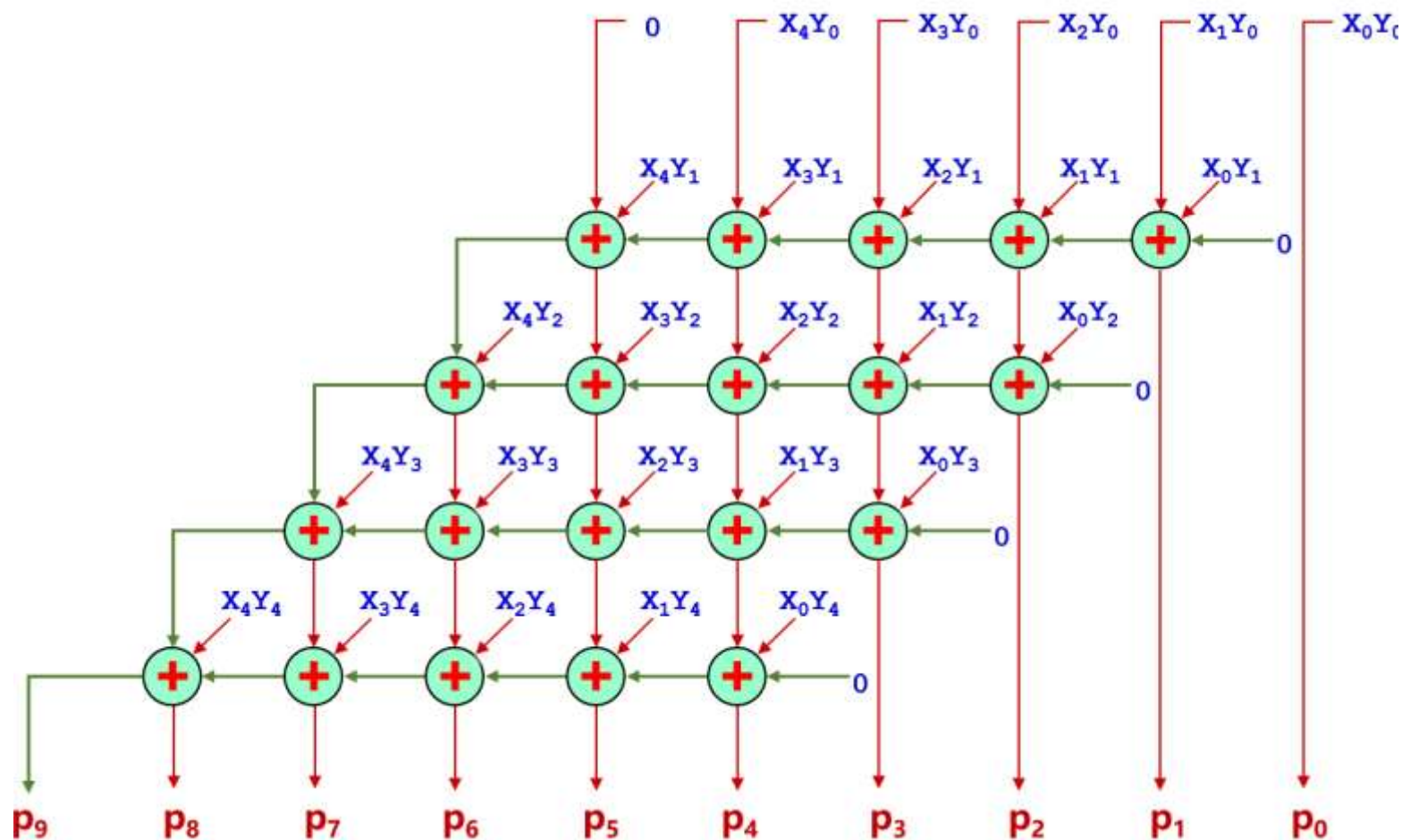
■ 与门实现一位乘法

■ 25个与门并发

■ 一级门延迟，生成所有相加数

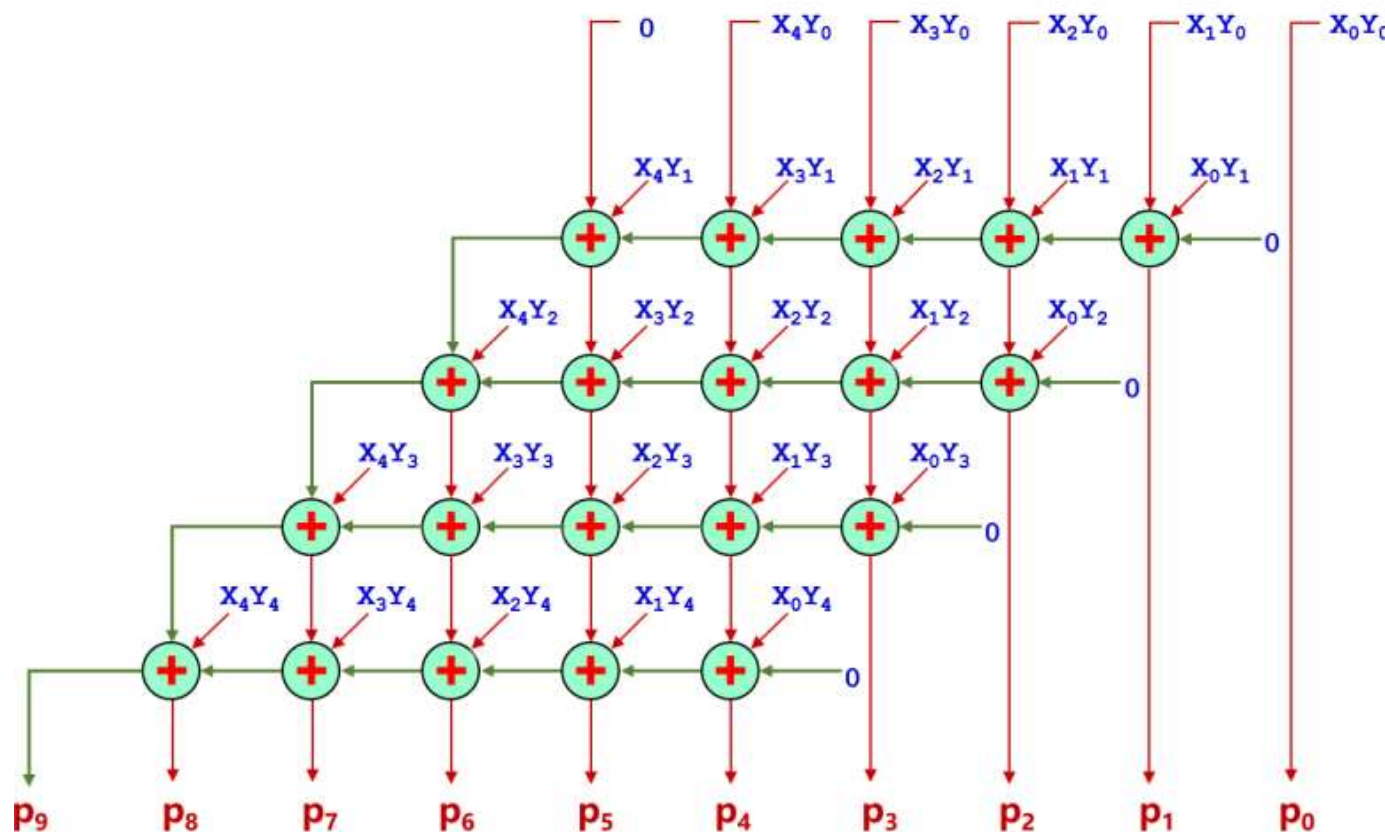


# 横向进位无符号阵列乘法器



包括20个全加器FA，全加器从上到下分为5行，前4行内的全加器没有进位依赖关系，行内全加器可并行，时间延迟为全加器的时间延迟6T。

# 横向进位无符号阵列乘法器

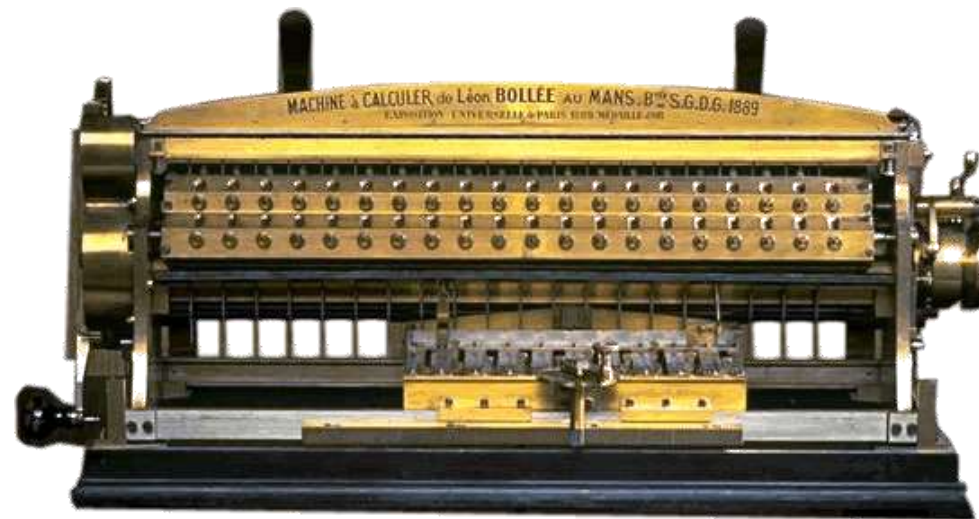


- 行与行之间有进位依赖关系, 最下面一行的全加器是 $n-1=4$ 位串行加法器, 串行加法器的时间延迟是 $[2(n-1)+4]T=12T$ , 所以总时间延迟为  $1T+(n-1) \times 6T + [2(n-1)+4]T = (8n-3)T = 37T$ 。
- 相比  $n$ 位串行加法器的 $(2n+4)T$ , 阵列乘法器的时间延迟并没有到 $n$ 量级, 有4倍左右的速度差异。



## 3.3 定点乘法运算

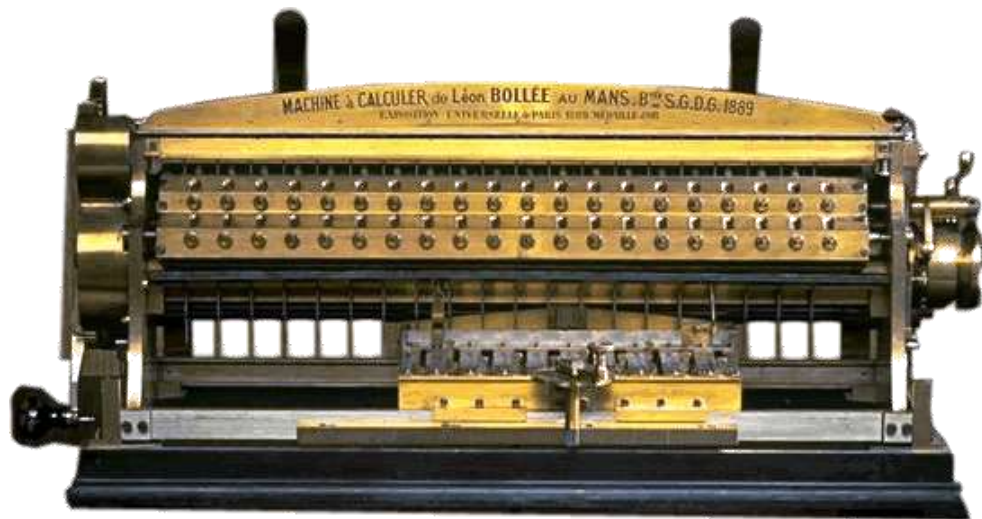
- 3.3.1 原码一位乘法
- 3.3.2 补码一位乘法
- 3.3.3 阵列乘法器
- **3.3.4 补码阵列乘法器**
- 3.3.5 乘法器性能优化





## 3.3 定点乘法运算

- 3.3.1 原码一位乘法
- 3.3.2 补码一位乘法
- 3.3.3 阵列乘法器
- 3.3.4 补码阵列乘法器
- 3.3.5 乘法器性能优化



# 本章主要内容

- 3.1 计算机中的运算
- 3.2 定点加/减法运算
- 3.3 定点乘法运算
- 3.4 定点除法运算
- 3.5 浮点运算
- 3.6 运算器



# 除法手工运算

关键是判断是否够减？

0.1 1 0 1 ← 商

0.1011 ) 0.1001001

0.01011

0.001110

0.001011

0.00001110

0.00001011

余数 0.00000011

- 不够减，商上0，
- 够减，商上1，求余数
- 除数右移一位
- 继续按规则上商直至所需位数

## 恢复余数除法（绝对值）

- 如何判断是否够减：在原码恢复余数法中，比较被除数(余数)与除数的大小是用减法实现的，相减结果为正(符号位为0)说明够减，商上1;相减结果为负(符号位为1)说明不够减，商上0。
- 当商上1时，减法得到的差值就是余数，可以进行后续的除法操作。
- **恢复余数法：**但商上0时表明不够减，减法得到的余数是负数，因此需要将余数加上除数，即将余数恢复成比较操作之前的数值。

# 恢复余数法



$X=0.1001$

$Y=0.1011$

求  $X \div Y$

$[-Y]_{补}=1.0101$

- 恢复余数法的运算过程:比较→上商(商为0时还需要恢复余数)→左移→比较,直到商达到规定的位数为止。
- 一般商的位数与除数的位数相同。

被除数/余数R	上商位 $Q_n$	说明
$00.1001$		
$[-Y] + 11.0101$		
$11.1110$	0	$R < 0, Q_n = 0, +Y$
$+ 00.1011$		
$00.1001$		
$\leftarrow 01.0010$		$R = 2R - Y$
$[-Y] + 11.0101$		
$00.0111$	0.1	$R > 0, Q_n = 1$
$\leftarrow 00.1110$		$R = 2R - Y$
$[-Y] + 11.0101$		
$00.0011$	0.11	$R > 0, Q_n = 1$
$\leftarrow 00.0110$		$R = 2R - Y$
$[-Y] + 11.0101$		
$11.1011$	0.110	$R < 0, Q_n = 0, +Y$
$+ 00.1011$		
$00.0110$		
$\leftarrow 00.1100$		$R = 2R - Y$
$[-Y] + 11.0101$		
$00.0001$	0.1101	$R > 0, Q_n = 1$

# 恢复余数乘法问题

- 需要进行恢复余数的操作
  - 余数是负数，必须恢复余数
  - 绝对值运算，余数不可能是负数
- 恢复余数的操作次数不确定
  - 运算时间不固定
  - 最慢除法（每次都不够除），拖慢除法速度
- 实际应用通常采用**不恢复余数除法**

## 不恢复余数

- 不恢复余数法：不够减时不需要恢复余数，而根据**余数符号**进行不同的运算处理。
- 运算步数固定，控制简单，有效提高了除法运算速度。

## 不恢复余数法

- 设某次余数为 $R_i$ ，求下位商需将 $R_i$ 左移一位，再减去除数 $Y$ 进行比较，此过程可表示为

$$2R_i - Y$$

- 余数 $R_i$ 小于0时商上0，需要恢复余数，左移一位，再减除数 $Y$ 比较

$$(2R_i + Y) - Y = 2R_i + Y$$

- **不恢复余数法**：当余数为正时，商上1，余数左移一位，减去除数。当余数为负时，商上0，余数左移一位，加上除数。



# 不恢复余数法



$X=0.1001$

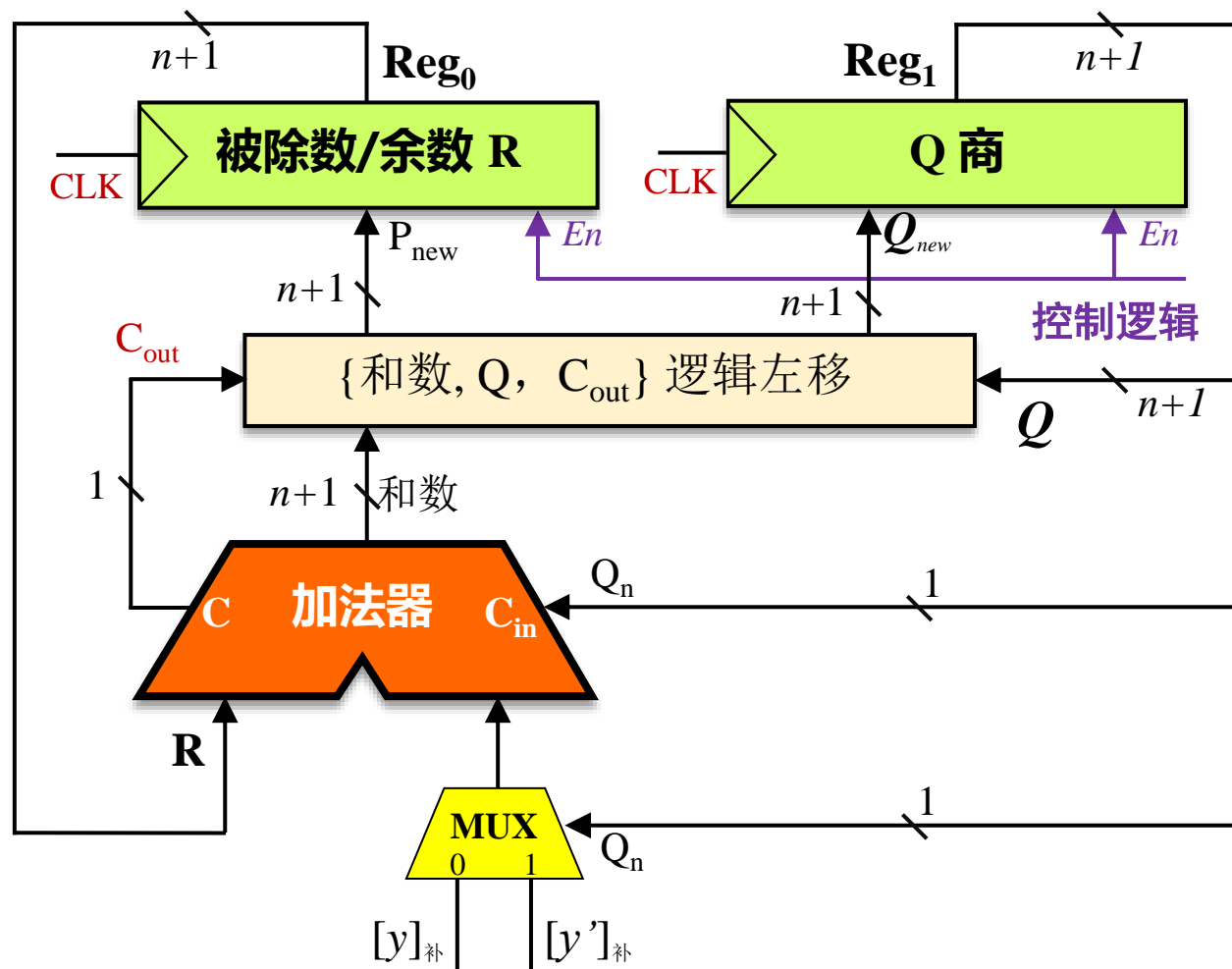
$Y=0.1011$

求  $X \div Y$

- $[-Y]_{补}=1.0101$
- 当余数为正时，商上1，余数左移一位，减去除数。
- 当余数为负时，商上0，余数左移一位，加上除数。

被除数/余数R	上商位 $Q_n$	说明
$00.1001$		$-Y$
$[-Y] + 11.0101$		
$11.1110$	0	$R < 0, Q_n = 0$ $R = 2R + Y$
$\leftarrow 11.1100$		
$[Y] + 00.1011$		
$00.0111$	0.1	$R > 0, Q_n = 1$ $R = 2R - Y$
$\leftarrow 00.1110$		
$[-Y] + 11.0101$		
$00.0011$	0.11	$R > 0, Q_n = 1$ $R = 2R - Y$
$\leftarrow 00.0110$		
$[-Y] + 11.0101$		
$11.1011$	0.110	$R < 0, Q_n = 0$ $R = 2R + Y$
$\leftarrow 11.0110$		
$[Y] + 00.1011$		
$00.0001$	0.1101	$R > 0, Q_n = 1$

# 不恢复余数法逻辑实现



## 习题

$X=0.1011, Y=0.1101$ , 用恢复余数法求  $X/Y$

X=0.1011,Y=0.1101,

[Y]<sub>补</sub>=00.1101,  
[-Y]<sub>补</sub>=11.0011,

	被除数(余数R)	商	操作说明
	00 1011	00000	开始情形
+)	11 0011		+[-Y] <sub>补</sub>
	11 1110	0000 0	不够减,商上0
+)	00 1101		+Y, 恢复余数
	00 1011		
	01 0110	000 00	左移一位
+)	11 0011		+[-Y] <sub>补</sub>
	00 1001	000 01	够减, 商上1
	01 0010	00 010	左移一位
+)	11 0011		+[-Y] <sub>补</sub>
	00 0101	00 011	
	00 1010	0 0110	左移
+)	11 0011		+[-Y] <sub>补</sub>
	11 1101	0 0110	
+)	00 1101		+Y, 恢复余数
	00 1010		

被除数(余数R)	(被除数)(商)	操作说明
00 1010	0   0 1 1 0	
01 0100	0   1 1 0 0	左移
11 0011 <sup>+) )</sup>		+[-Y] <sub>补</sub>
00 0111	0 1 1 0 1	

$X/Y=0.1101$ , 余数 $=0.0111 \times 2^{-4}$

# 本章主要内容

- 3.1 计算机中的运算
- 3.2 定点加/减法运算
- 3.3 定点乘法运算
- 3.4 定点除法运算
- 3.5 浮点运算
- 3.6 运算器



# 浮点运算

- 浮点数比定点数表示的范围大，有效精度也更高，更适合于工程计算。
- 数据运算处理过程比较复杂，硬件成本高，运算速度也慢一些。
- 浮点数常采用规格化数进行运算。

# 浮点运算

- 阶码和尾数均采用补码

$$\begin{array}{rcl} X = 2^{E_x} M_x & Y = 2^{E_y} M_y & + Y = 2^{E_y} M_y \\ & & \hline & & ??????? \end{array}$$

- 当 $E_x=E_y$ 时，尾数部分直接运算即可得到浮点形式的运算结果。
- 但当 $E_x \neq E_y$ 时，必须先设法让两个阶码相等后才能进行尾数部分的运算。
- 使阶码相等的过程称为**对阶**，对阶完成后即可进行尾数的加减法运算



# 浮点数加减法五步骤

- 对阶
- 尾数求和
- 规格化
- 溢出判断
- 舍入

# 对阶

- 对阶 (使得尾数可以直接相加)
- 对阶的原则: 是**小的阶码向大的阶码看齐**, 这是因为小阶码数值增大时, 尾数部分会右移, 舍去的是尾数的低位部分, 只有很小的精度影响。而如果让大阶码向小阶码看齐, 则尾数部分需进行左移, 将会丢失尾数的高位部分, 会严重影响运算精度和结果的正确性。
- 例子  $2^8 * (0.11000) + 2^6 * (0.00111)$

大对小 or 小对大 ?

大阶对小阶  $2^8 * (0.11000) \rightarrow 2^6 * (11.000)$  溢出

小阶对大阶  $2^6 * (0.00111) \rightarrow 2^8 * (0.00001\underline{11})$  ?

# 对阶

对阶的两个步骤。

①求阶差:

②阶码的调整与尾数的移位。尾数右移时通常将最低位的移出位暂时保留，称为**保留附加位**。保留附加位参与中间运算以提高运算精度，尾数运算结束，结果规格化后再进行舍入。

## 尾数运算

对阶完成后可按照定点数的补码加减运算法则执行尾数加减操作。

## 运算结果规格化

- 结果规格化就是使运算结果成为规格化数。
- 让尾数的符号位扩展为**双符号位**，当尾数运算结果**不是11.0.或00.1**的形式时，应进行相应的规格化处理。
- 当尾数符号位为01 或10时，运算结果上溢，需要**向右规格化**，且只需将尾数右移一位，同时将结果的阶码值加1。
- 当尾数运算结果为 11.1或00.0. .时需要**向左规格化**，而且左移次数不固定，与运算结果的形式有关。向左规格化时尾数连同符号位一起左移，直到尾数部分出现11.0...或00.1...的形式为止。向左规格化时阶码做减法，左移多少位就减多少。

# 舍入处理

0	1	1	0	0	0	1	1	
0	0	1	1	0	0	0	1	1

- 右规后低位部分丢失了一位，产生误差
- 舍入方法：**末位恒置1法**、**0舍1入**
- **末位恒置1法**:只要因移位而丢失的位中有一位是1，就把运算结果的最低位置1，而不管最低位原来是0还是1。
- **0舍1入法**:当丢失位数的最高位是1时将尾数的末位加1。注意舍入操作可能会破坏规格化结果，所以舍入操作后还需要再次进行规格化处理。

## 溢出处理

- 由于浮点数中阶码的位数决定数的表示范围，因此对浮点运算而言，  
**当阶码出现溢出时才表示运算结果溢出**，即当阶码的符号位为01和10时才表示运算结果溢出。当为01时，置溢出标志，反之置运算结构机器零

## 浮点运算举例

- 例1 两浮点数  $x = 2^{101} \times 0.11011011$ ,  $y = 2^{111} \times (-0.10101100)$ 。假设尾数在计算机中以补码表示, 可存储10位尾数, 2位符号位, 阶码以补码表示, 双符号位, 求  $x + y$ 。

解: 将 $x, y$ 转换成浮点格式

$$[x]_{\text{浮}} = 00101, 00.11011011$$

$$[y]_{\text{浮}} = 00111, 11.01010100$$

步骤1: 对阶, 阶差为  $E_x - E_y = [E_x]_{\text{补}} + [-E_y]_{\text{补}}$

$$[-E_y]_{\text{补}} = 11001 \quad E_x - E_y = 00101 + 11001 = 11110 = -2 < 0$$

小阶对大阶, X阶码加2, 尾数右移2位

$$[x]_{\text{浮}} = 00111, 00.00110110\underline{11} \quad \text{保留位}$$



## 浮点运算举例

$$[X]_{\text{浮}} = 00111, 00.00110110\underline{11} \quad \text{保留位}$$

$$[Y]_{\text{浮}} = 00111, 11.01010100$$

### 步骤2: 尾数求和

$$[X+Y]_{\text{浮}} = 00111, 11.10001010\underline{11} \quad \text{保留位参与运算}$$

### 步骤3: 结果规格化

$$[X+Y]_{\text{浮}} = 00110, 11.00010101\underline{1} \quad \text{非规数, 左归1位, 阶码减1, 保留位?}$$

### 步骤4: 舍入处理

$$[X+Y]_{\text{浮}} = 00110, 11.00010110 \quad (\text{0舍1如法})$$

### 步骤5: 溢出判断

$$[X+Y]_{\text{浮}} = 2^{110} \times (-0.11101011) \quad \text{无溢出}$$

## IEEE 754 溢出例子

- IEEE754 浮点数的阶码采用移码表示，而尾数采用原码表示，且尾数的高位隐藏，因此，IEEE754 浮点数的加减运算会有如下不同。

- (1)对阶和规格化过程中，阶码的运算采用**移码**的加减运算规则。
- (2)尾数的运算采用**原码**运算规则，且**隐藏位要参与尾数运算**。
- (3)**隐藏位参与尾数规格化**判断及尾数规格化过程。

# IEEE 754 溢出例子

(4)舍入处理，IEEE754中主要有以下4种舍入方式。

- **就近舍入**，舍入为最近可表示的数，如果数据正好处于两个可表示的中间则向偶数舍入。
- **朝正 $\infty$ 方向舍入**，总是取右侧最近的可表示的数。
- **朝负 $\infty$ 方向舍入**，总是取左侧最近的可表示的数。
- **朝0方向舍入**，直接丢弃多余位，也称为截去法。

(5)溢出判断。浮点运算的溢出可通过阶码的溢出来判断。对IEEE754单精度浮点数而言，向右规格化使阶码为全1(即11111111，真值为128)时发生规格化上溢。向左规格化使阶码为全0时发生规格化下溢。

# IEEE 754 溢出例子

$X = 00C0\ 0000$      $Y = 0080\ 0000$     求  $X - Y$ ?

X	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	--

■ 阶码相同， $E=1$ ，尾数还原隐藏位后运算

□  $1.1 - 1.0 = 0.1$

■ 非规数，尾数左移一位，阶码减一

■ 阶码为0，非规格化数，运算下溢

# 浮点数乘法运算

如:  $X=2^m \times M_x$      $Y=2^n \times M_y$

$$X \times Y = (2^m \times M_x) \times (2^n \times M_y) = 2^{m+n} \times (M_x \times M_y)$$

## 1. 阶码相加

阶码相加可能产生溢出, 要进行溢出判断, 如溢出计算机要进行处理

## 2. 尾数相乘

尾数相乘可得积的尾数, 可按定点乘法运算方法运算

## 3. 结果规格化

可按浮点加/减法运算规格化方式处理, 舍入方式也相同

# 浮点数除法运算

如:  $X=2^m \times M_x$      $Y=2^n \times M_y$

$$X \div Y = (2^m \times M_x) \div (2^n \times M_y) = 2^{m-n} \times (M_x \div M_y)$$

## 1. 尾数调整

如被除数尾数大于除数尾数 (绝对值), 则将被除数尾数右移一位, 阶码+1

## 2. 阶码求差

商的阶码等于被除数的阶码减去除数的阶码

## 3. 尾数相除

以被除数的尾数除以除数的尾数以获得商的尾数, 尾数相除与定点除法运算相同

## 习题

$X=2^{010}\cdot 0.11011011$ ,  $Y=2^{100}\cdot (-0.10101100)$ , 求 $X+Y$

## 习题

$$X=2^{010}\cdot 0.11011011, \quad Y=2^{100}\cdot (-0.10101100)$$

计算过程:

①对阶操作: 阶差 $\Delta E=[E_x]_{\text{补}}+[-E_y]_{\text{补}}=00010+11100=11110$

X阶码小,  $M_x$ 右移2位, 保留阶码 $E=00100$

$[M_x]_{\text{补}}=00\ 00110110\ \underline{11}$

②尾数相加:  $[M_x]_{\text{补}}+[M_y]_{\text{补}}=00\ 00110110\ \underline{11}+11\ 01010100$

$=11\ 10001010\ \underline{11}$

③规格化操作: 左规, 移一位, 结果 $=11\ 00010101\ \underline{10}$

阶码减1,  $E=00011$

④舍入: 附加位最高位为1, 在结果的最低位+1,

得新结果 $[M]_{\text{补}}=11\ 00010110, M=-0.11101010$

⑤判溢出: 阶符为00, 不溢出, 最终结果为

$$X+Y=2^{011}\cdot (-0.11101010)$$



# 本章主要内容

- 3.1 计算机中的运算
- 3.2 定点加/减法运算
- 3.3 定点乘法运算
- 3.4 定点除法运算
- 3.5 浮点运算
- 3.6 运算器



- 计算机中的各类算术运算都可以由最基本的**定点加法**和**移位运算**迭代实现
- 采用数字逻辑电路自动实现，将逻辑运算、移位运算、各种算术运算的逻辑实现集成在一起就可以构成CPU中的运算器。运算器是对数据进行加工处理的部件，它是CPU 中的重要组成部分，具体可分为**定点运算部件**和**浮点运算部件**。

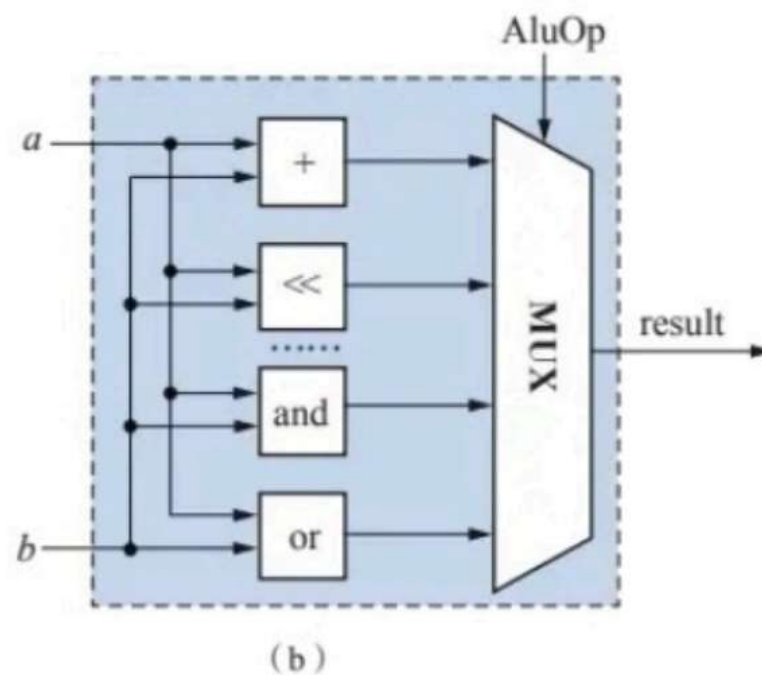
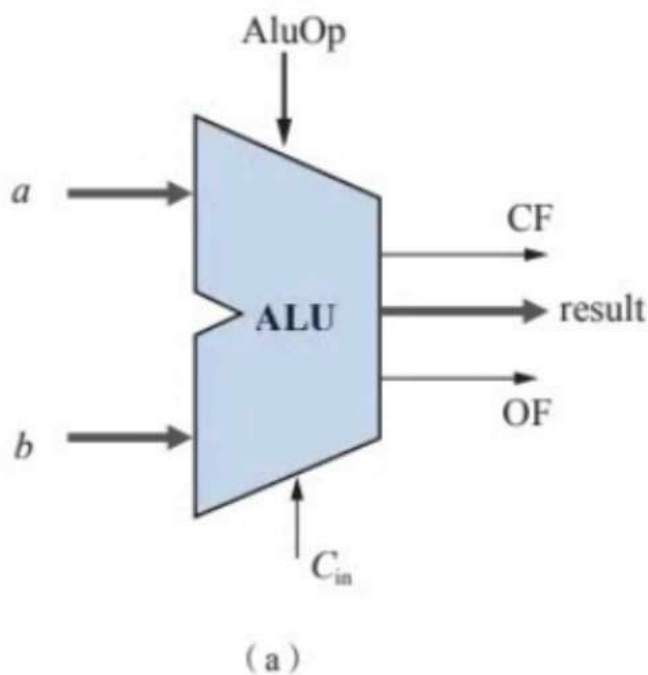
- **定点运算部件：** 算术逻辑运算单元 (Arithmetic Logic Unit, ALU)，可以进行定点数据的逻辑、移位、算术运算。
- **浮点运算部件(Float Point Unit, FPU)：** 浮点数的算术运算。

# 定点运算器

- 一般包含如下几个基本部分：算术逻辑运算单元、通用寄存器组、输入数据选择电路、输出数据控制电路等

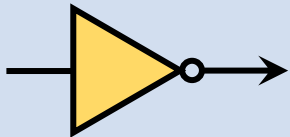
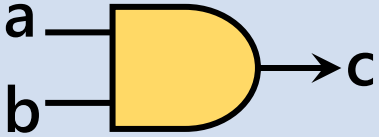
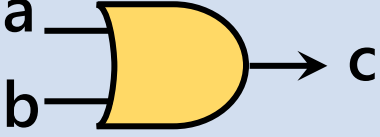
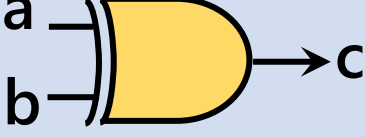
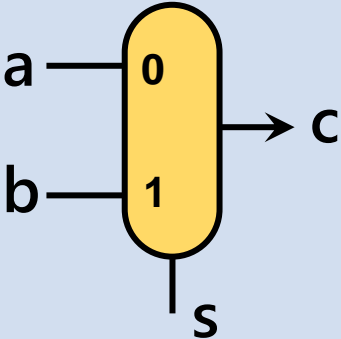
# ALU

- 算术逻辑运算单元(ALU)是对**定点数据进行加工处理**的纯组合逻辑电路
- 包括**算术运算**和**逻辑运算**，也常作为数据传送的通路。

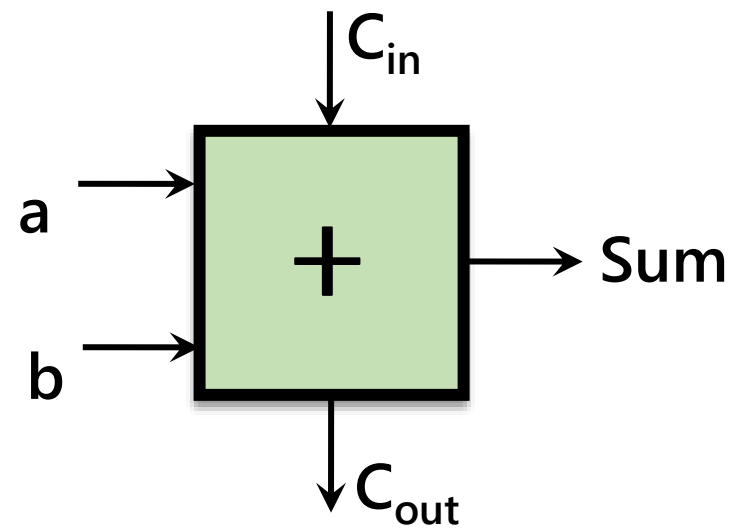
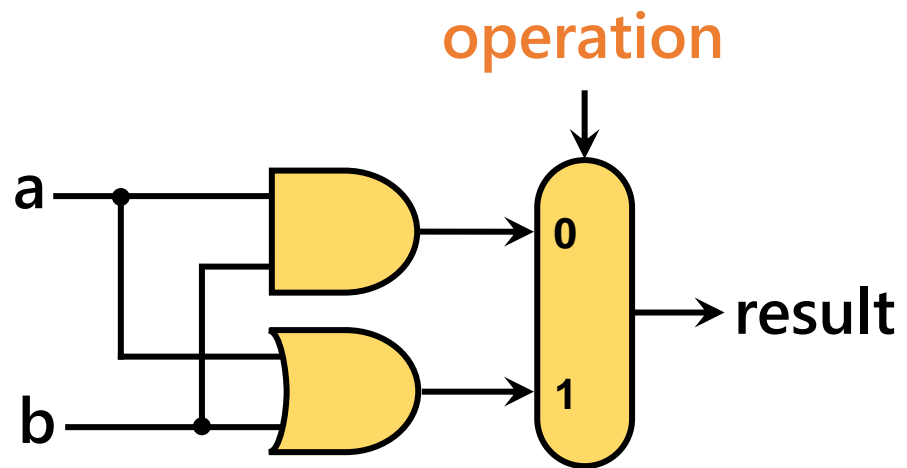


# ALU设计

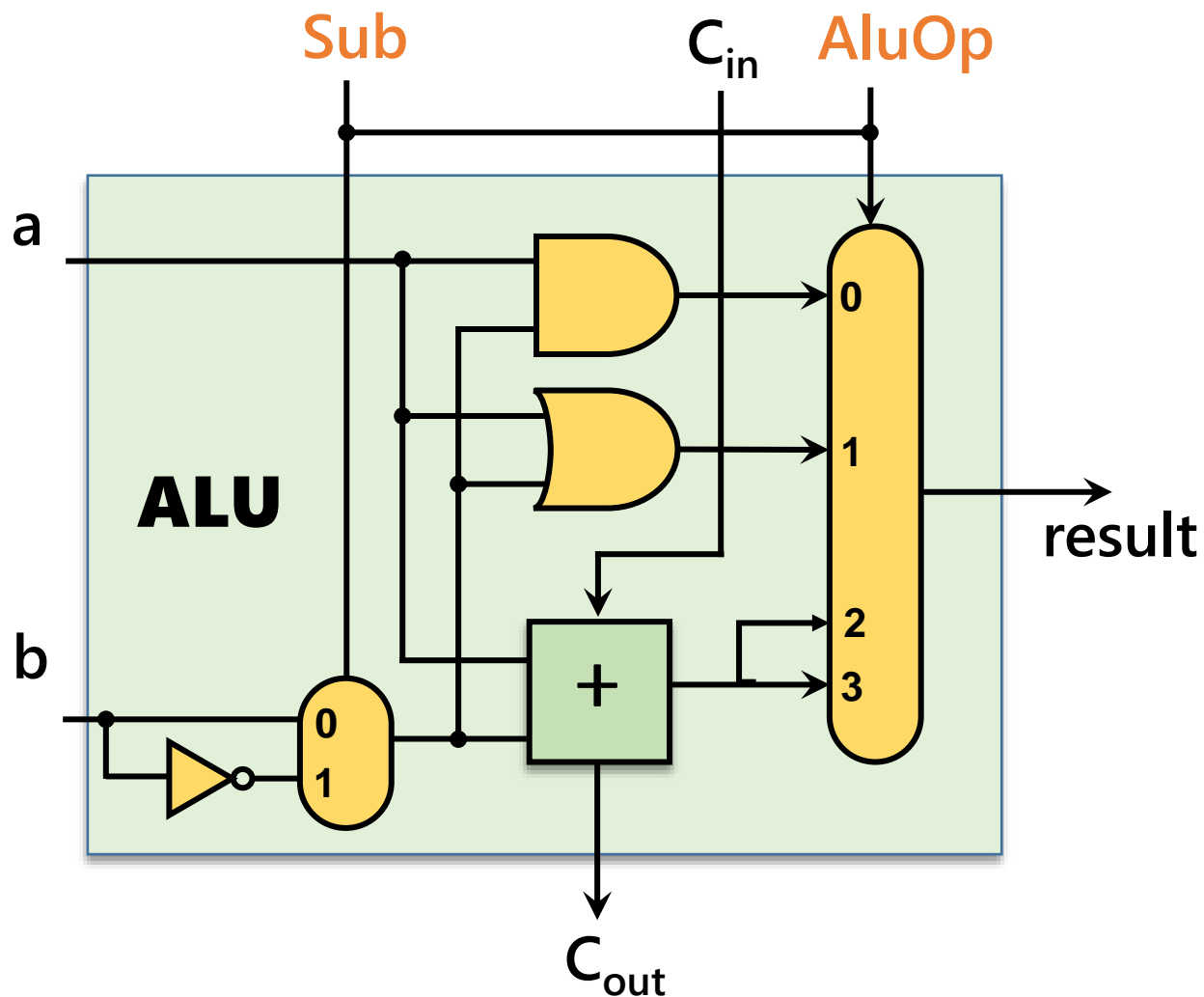
ALU是计算机的核心部件，能实现的基本功能包括加、减等算术运算和与、或、非等逻辑运算。实现上述算术运算和逻辑运算功能的部件就是构造ALU的基本单元。

非	与	或	异或	多路选择
				
$c = \sim a$	$c = a \& b$	$c = a   b$	$c = a \wedge b$	$c = (s == 0)?a,b$

# 简单运算



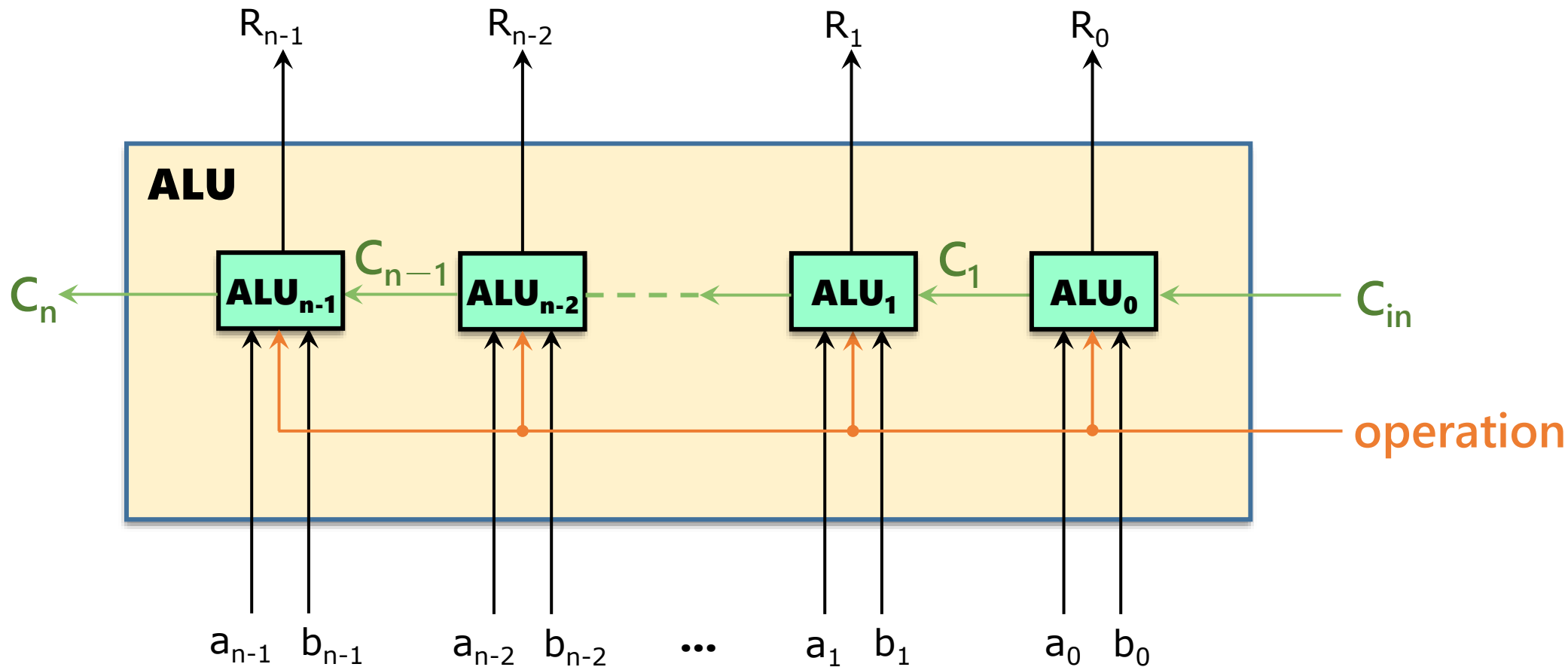
# 1位ALU



- *AluOp*为2位**运算选择码**,
- 当 $AluOp=0$ 时, ALU 完成逻辑与运算;
- 当 $AluOp=1$ 时, 完成逻辑或运算;
- 当 $AluOp=2$ 时, *Sub* 信号应通过相关逻辑译码为 0。通过二路选择器选择 *b* 进入全加器, 运算结果为  $a+b$ ,
- 当  $AluOp=3$  时, ALU完成减法运算。



# 多位ALU

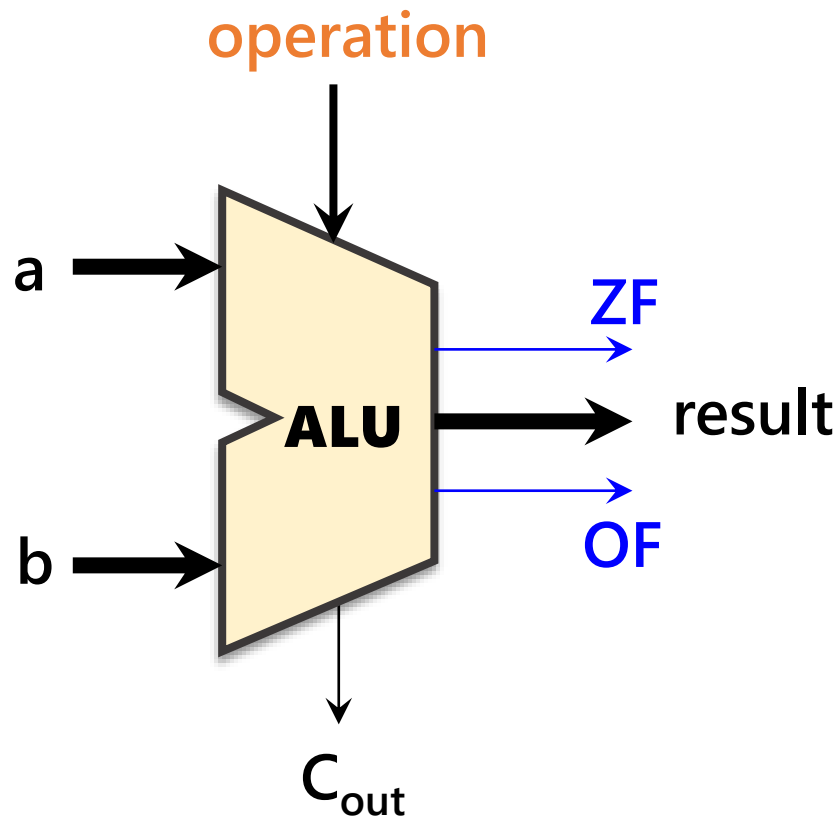


- 多个1位ALU按进位链串联即可得到n位的ALU
- 进行多位减法时，最低位进位位应该置1，也就是要将Sub信号连接到低位ALU的进位输入端，实现末位加1的功能。

# 算术逻辑运算单元ALU组成...

■ ALU的输出除运算结果result 外，还包括若干**状态标志位**，常见的状态标志位如下。

- ZF (结果为零)
- SF (结果为负数)
- CF (进位/借位)
- OF (有符号溢出)



大萝卜喊你去做实验



## 通用寄存器组

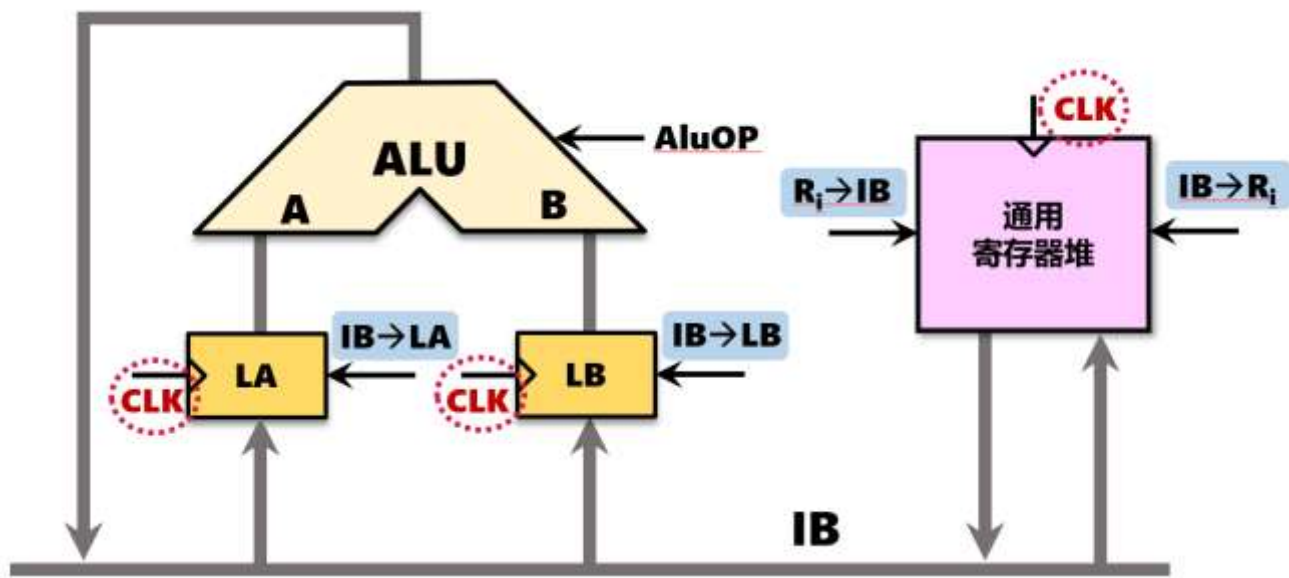
- **暂时存放参加运算的数据和运算结果**，尽量减少指令执行过程中访问主存的次数，以提高运算速度。
- **作为状态寄存器**，保存运算过程中设置的状态，如进位、溢出、结果为负等。这些状态可用于程序执行流程的控制。
- **可作为变址寄存器、堆栈指示器使用**。不同的计算机对这组寄存器的使用情况和设置个数不相同。

# 运算器的结构

- 运算器的基本结构与运算器中的总线结构以及运算器各部件与总线的连接方式紧密相关，不同的连接构成不同的数据通路，形成不同结构的运算器。
  - 单总线
  - 双总线
  - 三总线

# 运算器与总线结构

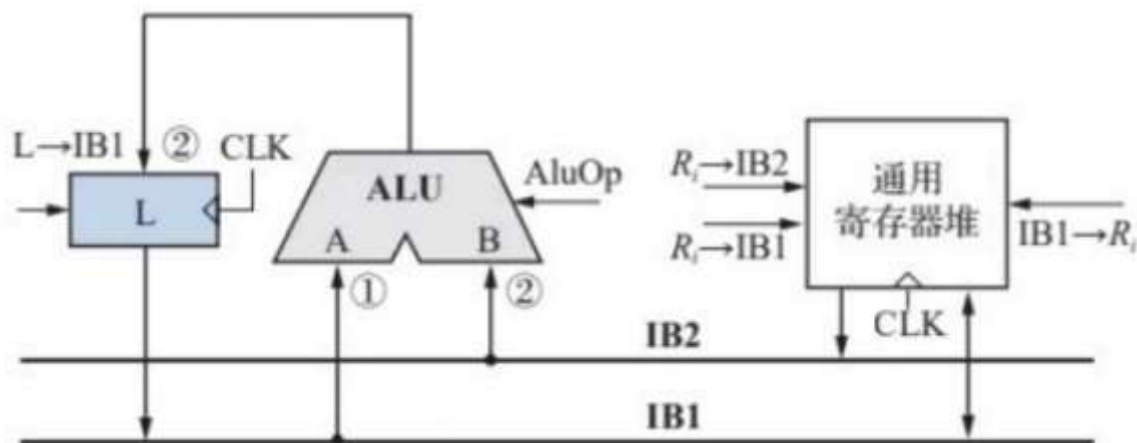
## ■ 单总线结构中的数据通路



- 所有部件都与**内部总线IB**(Internal Bus)连接。
- 同一时刻总线上只能传输一个数据，
- 在ALU 输入端设置LA、LB 两个**缓冲寄存器**。通过总线**分时**将两个寄存器操作数分别送入LA、LB缓冲器，只有**两个操作数同时**出现在ALU的输入端，ALU才能正确执行相应运算。
- 运算结果可通过总线存入通用寄存器或缓冲器LA或LB 中。
- 单总线结构需要两个缓冲器

# 运算器与总线结构

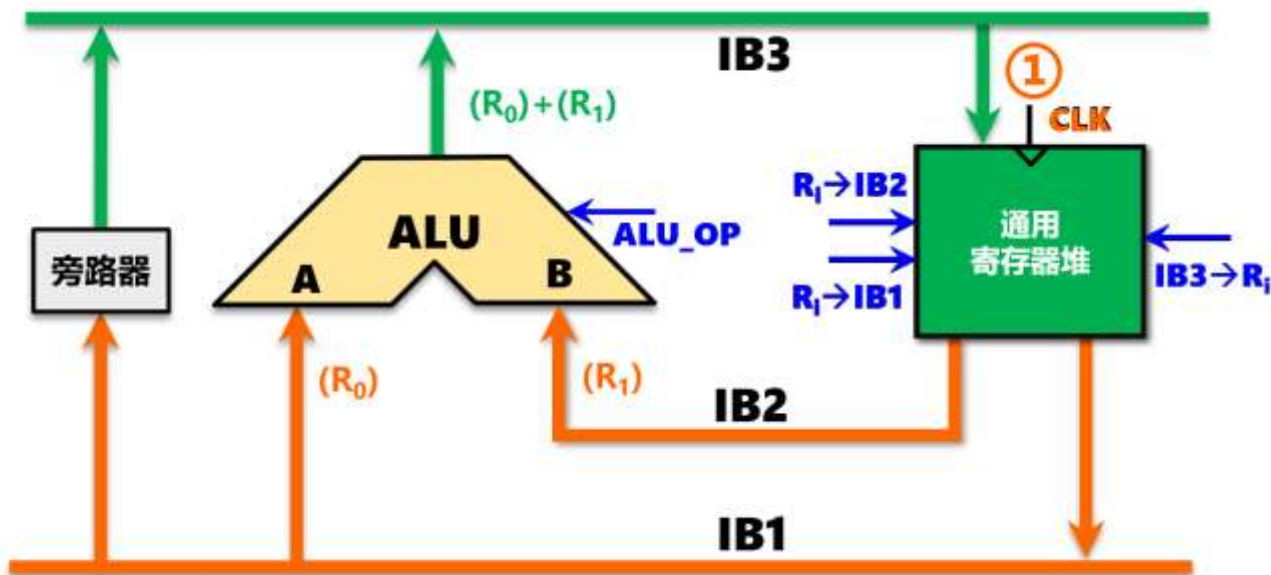
## ■ 双总线结构与运算通路1



- ALU与通用寄存器堆都连接在**总线IB1 和IB2**上，通用寄存器堆有两个输出端口，可以通过两组总线分别将两个寄存器操作数**同时加载到 ALU 的两个输入端**。
- 双总线结构运算器完成运算需要两个时钟周期：
  - 第1个时钟周期给出 $R_i \rightarrow IB1$ 、 $R_i \rightarrow IB2$ 信号来分别输出两个寄存器操作数，同时给出 ALU运算控制信号 $AluOp$ 来控制数据进行正确的运算，时钟到来时运算结果会自动写入缓冲寄存器L中；
  - 第2个时钟周期将L中的数据送入IB1总线，在时钟信号的配合下将数据写回通用寄存器中。

# 运算器与总线结构

## ■ 三总线结构与运算通路



- 操作部件连接在**3组总线**上，可同时通过3组总线传输数据(包括两个寄存器操作数和一个运算结果)。
- 在时钟周期配合下完成运算，整个运算只需要一个时钟周期，速度是3种结构中最快的，且不需要缓冲寄存器。
- 通用寄存器堆需要提供两个读端口、一个写端口。

## 运算器与总线结构

- 单总线，2个锁存器，3个时钟周期
- 双总线，1个锁存器，2个时钟周期
- 三总线，0个锁存器，1个时钟周期
- **总线越多，性能越好**