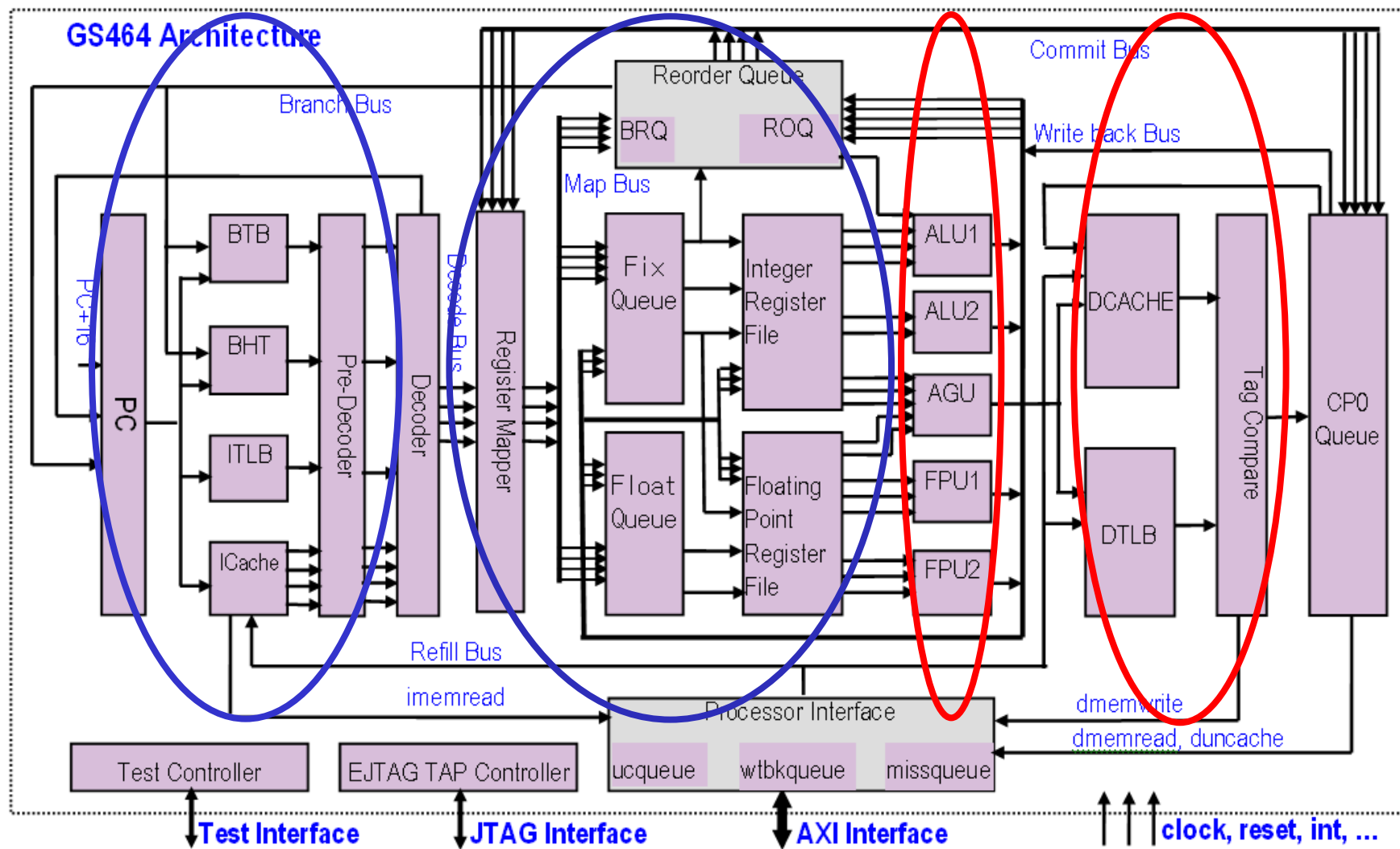


# 计算机体系结构

胡伟武、汪文祥

# 龙芯2号处理器核结构图



# 功能部件

- 定点补码加法器设计
- 定点ALU设计
- 定点补码乘法器的设计

# 定点补码加法器设计

# 先行进位加法器

- 一位全加器
  - 三个输入: A, B, Cin
  - 两个输出: S, Cout

		<sup>^</sup> A		A	
		00	01	11	10
<sup>^</sup> C	0	0	1	0	1
C	1	1	0	1	0
		<sup>^</sup> B	B	<sup>^</sup> B	

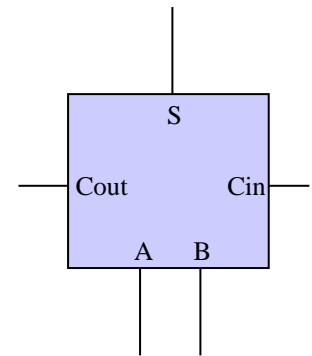
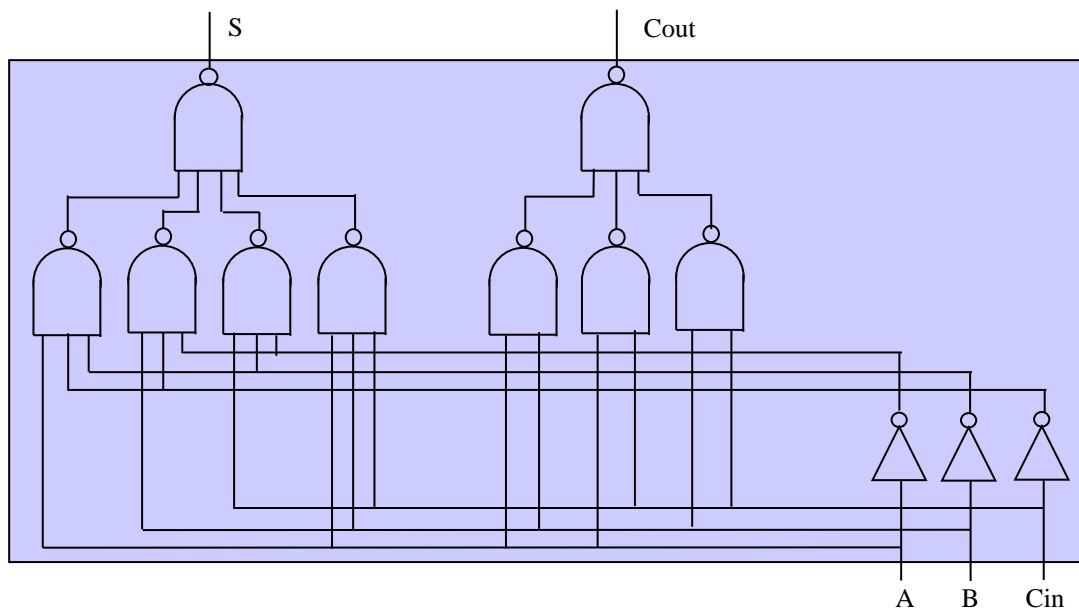
$$S = \textcolor{red}{^A}A * \textcolor{red}{^A}B * \textcolor{red}{Cin} + \textcolor{red}{^A}A * \textcolor{red}{B} * \textcolor{red}{^A}Cin + \textcolor{red}{A} * \textcolor{red}{^B}B * \textcolor{red}{^A}Cin + \textcolor{red}{A} * \textcolor{red}{B} * \textcolor{red}{Cin}$$

		<sup>^</sup> A		A	
		00	01	11	10
<sup>^</sup> C	0	0	0	1	0
C	1	0	1	1	1
		<sup>^</sup> B	B	<sup>^</sup> B	

$$\text{Cout} = \textcolor{teal}{A} * \textcolor{teal}{B} + \textcolor{red}{A} * \textcolor{red}{Cin} + \textcolor{blue}{B} * \textcolor{blue}{Cin}$$

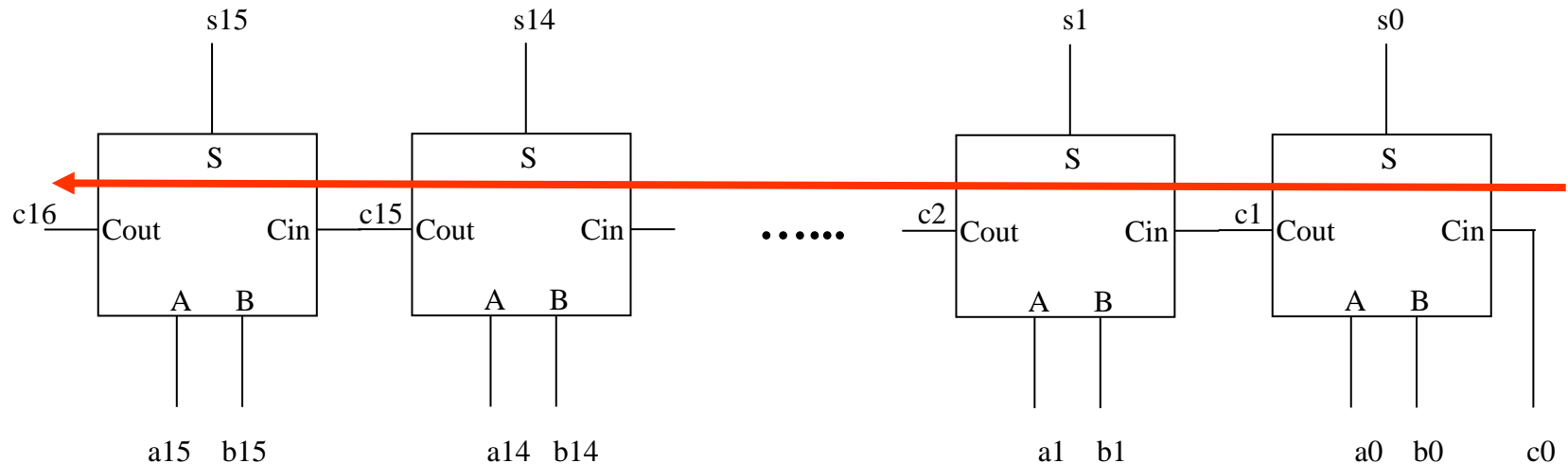
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

# 一位全加器



# 串行进位加法器

- 以16位加法器为例
- 进位从低位到高位传送, 形成c16需要32级门延迟
- 延迟随位数增长线性增长



# 进位的传递

$$\begin{aligned}c_{i+1} &= a_i * b_i + a_i * c_i + b_i * c_i \\&= a_i * b_i + (a_i + b_i) * c_i \\&= g_i + p_i * c_i\end{aligned}$$

- $g_i = a_i * b_i$  称为进位生成因子，只要  $g_i$  为1，就有进位
- $p_i = a_i + b_i$  称为进位传递因子，只要  $p_i$  为1，就把低位进位向前传递
- 四位进位传递为例

$$c_1 = g_0 + (p_0 * c_0)$$

$$c_2 = g_1 + (p_1 * g_0) + (p_1 * p_0 * c_0)$$

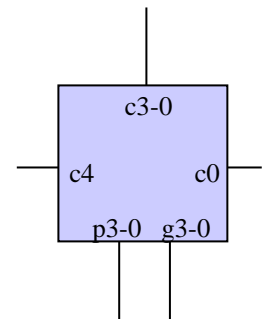
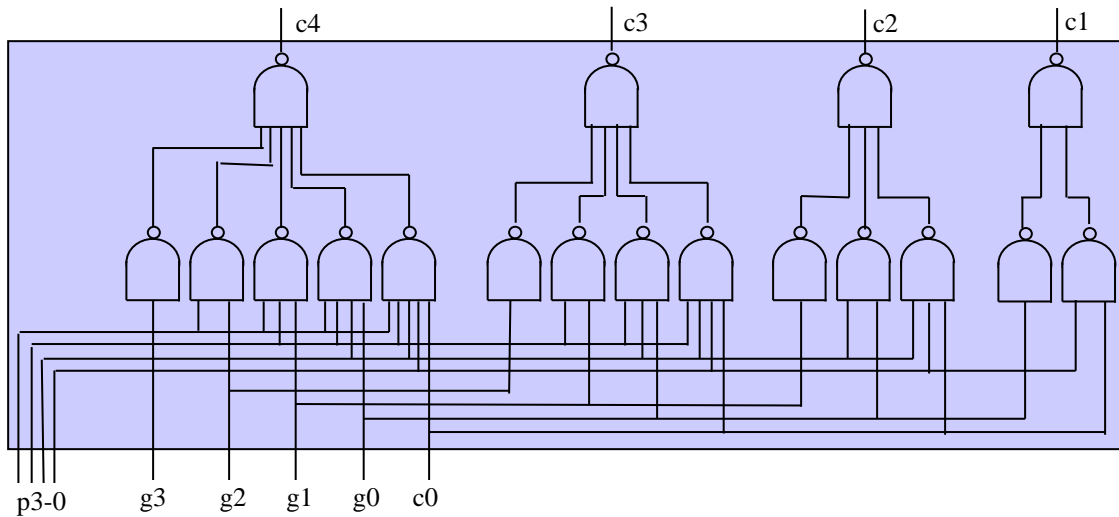
$$c_3 = g_2 + (p_2 * g_1) + (p_2 * p_1 * g_0) + (p_2 * p_1 * p_0 * c_0)$$

$$c_4 = g_3 + (p_3 * g_2) + (p_3 * p_2 * g_1) + (p_3 * p_2 * p_1 * g_0) + (p_3 * p_2 * p_1 * p_0 * c_0)$$

- 只要低位有一个进位生成，而且被传递，则进位输出为1<sup>8</sup>

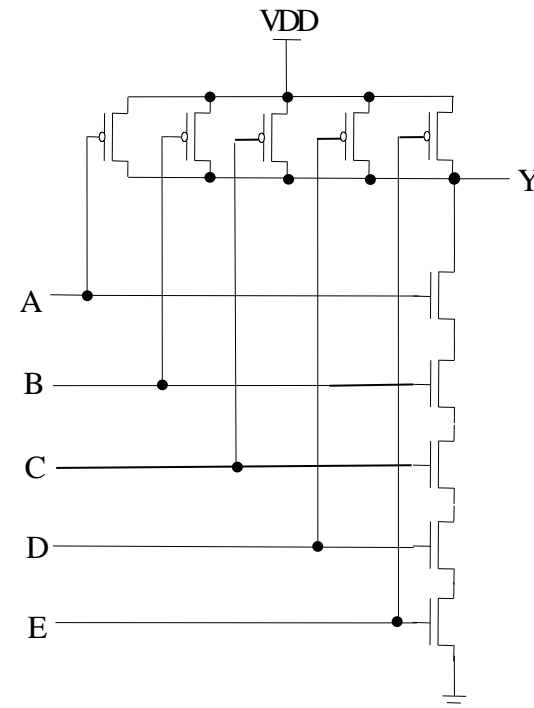


# 4位并行进位逻辑



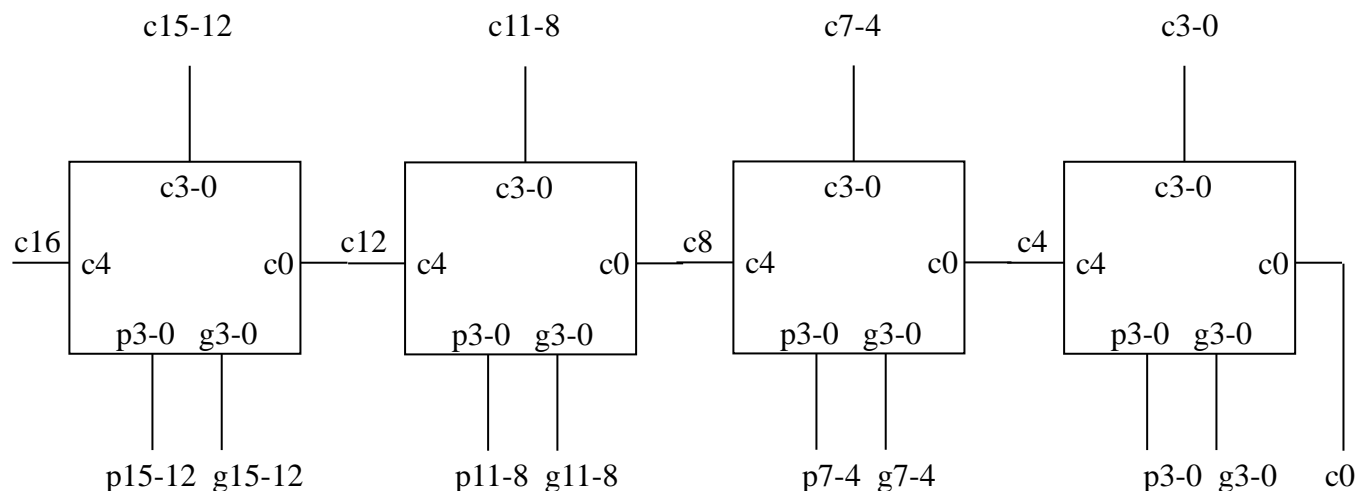
## 五输入与非门不常用

- 五个N管串接电阻大
  - 一般最多串接四个



# 以16位加法器为例

- 输入为 $p_i$ 、 $g_i$ ，输出为 $c_i$
- 每次并行产生4位进位，从 $p_i$ 、 $g_i$ 产生 $c_{16}$ 只要4级传递，8级门延迟（产生运算结果还需要一个异或）。原来从 $a_i$ 、 $b_i$ 产生 $c_{16}$ 需要16级传递，32级门延迟
- 分块，块内并行，块间串行
- 块内并行，块间并行？



# 块内并行, 块间并行

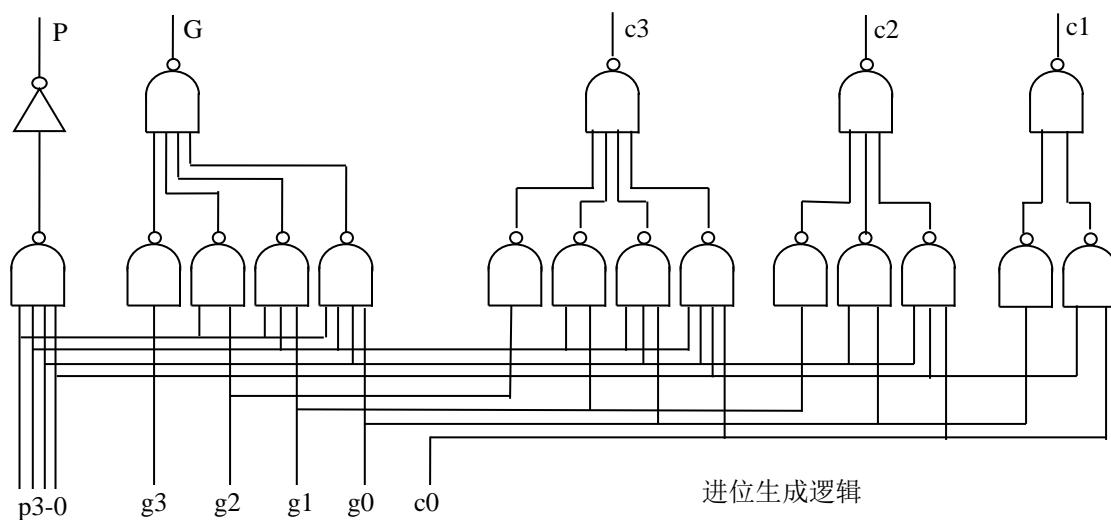
- 老办法：产生每块的进位传递因子和进位产生因子
- 进位传递因子：每一位的传递因子都为1时才能传递

$$P = p_0 * p_1 * p_2 * p_3$$

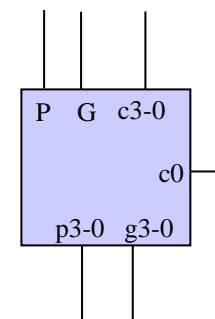
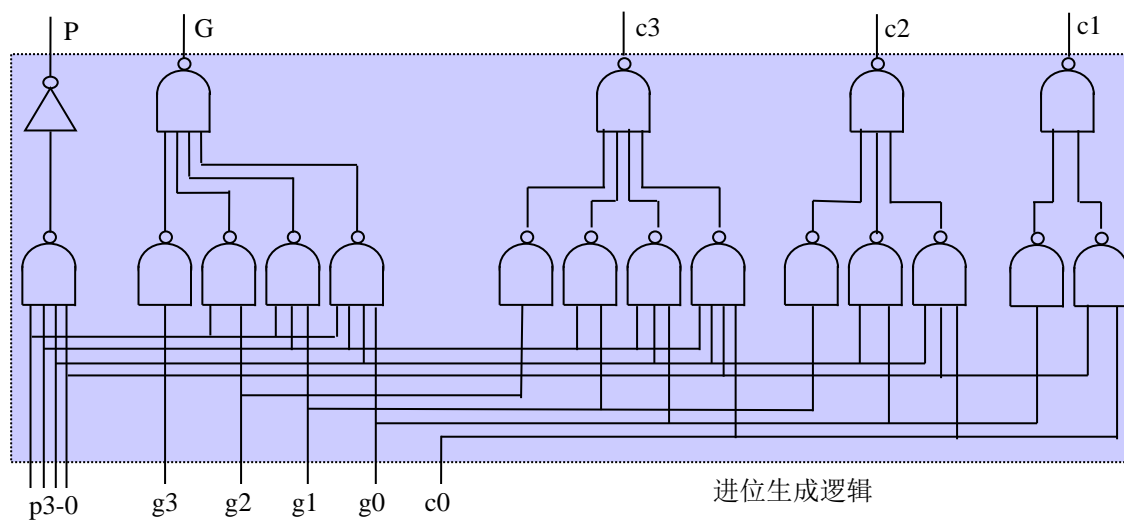
- 进位产生因子：块内产生进位, 不考虑进位输入

$$G = g_3 + (p_3 * g_2) + (p_3 * p_2 * g_1) + (p_3 * p_2 * p_1 * g_0)$$

$$\begin{aligned} c_4 &= g_3 + (p_3 * g_2) + (p_3 * p_2 * g_1) + (p_3 * p_2 * p_1 * g_0) + (p_3 * p_2 * p_1 * p_0 * c_0) \\ &= G + (P * c_0) \end{aligned}$$

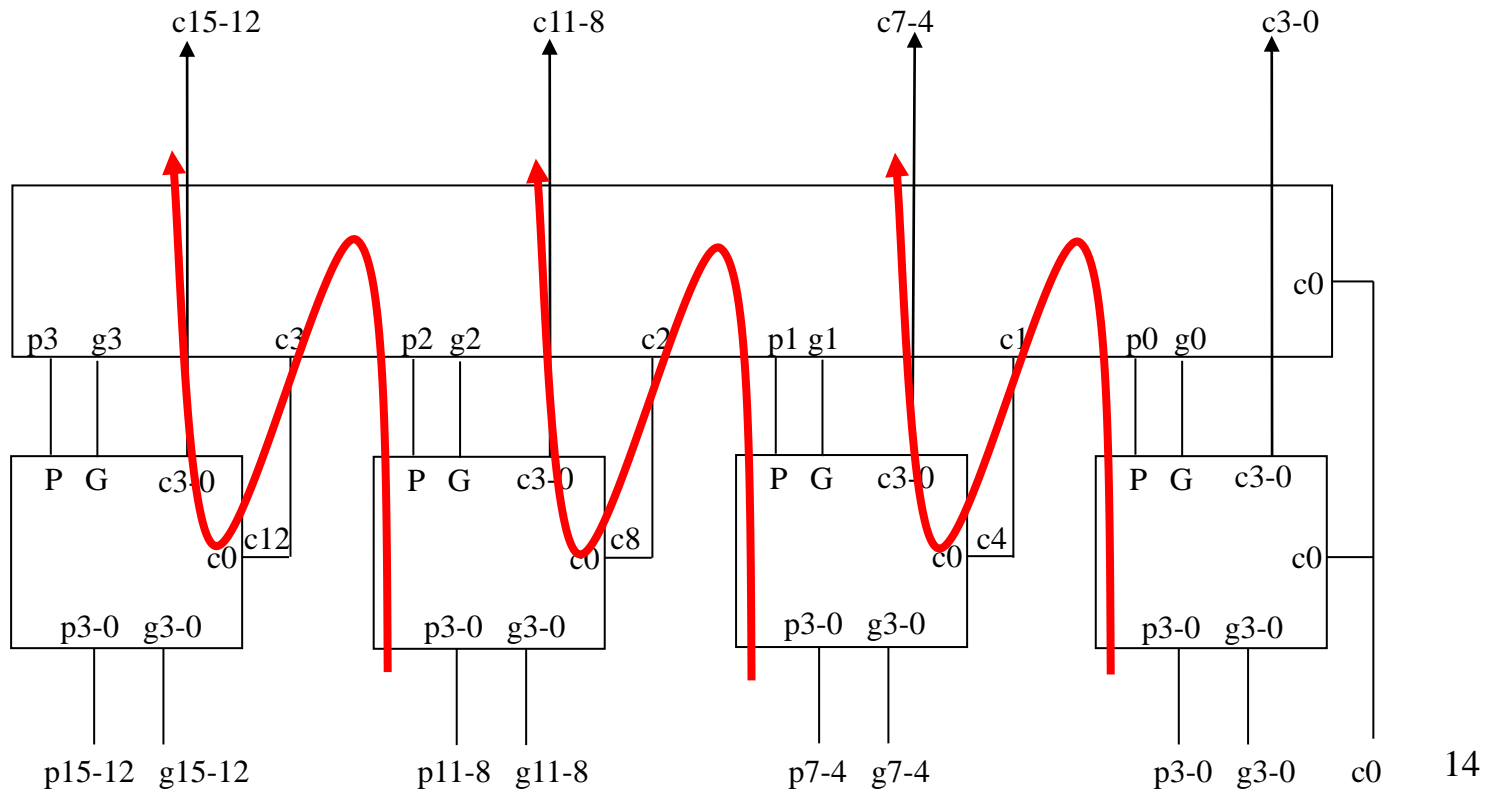


# 进位生成与传递逻辑



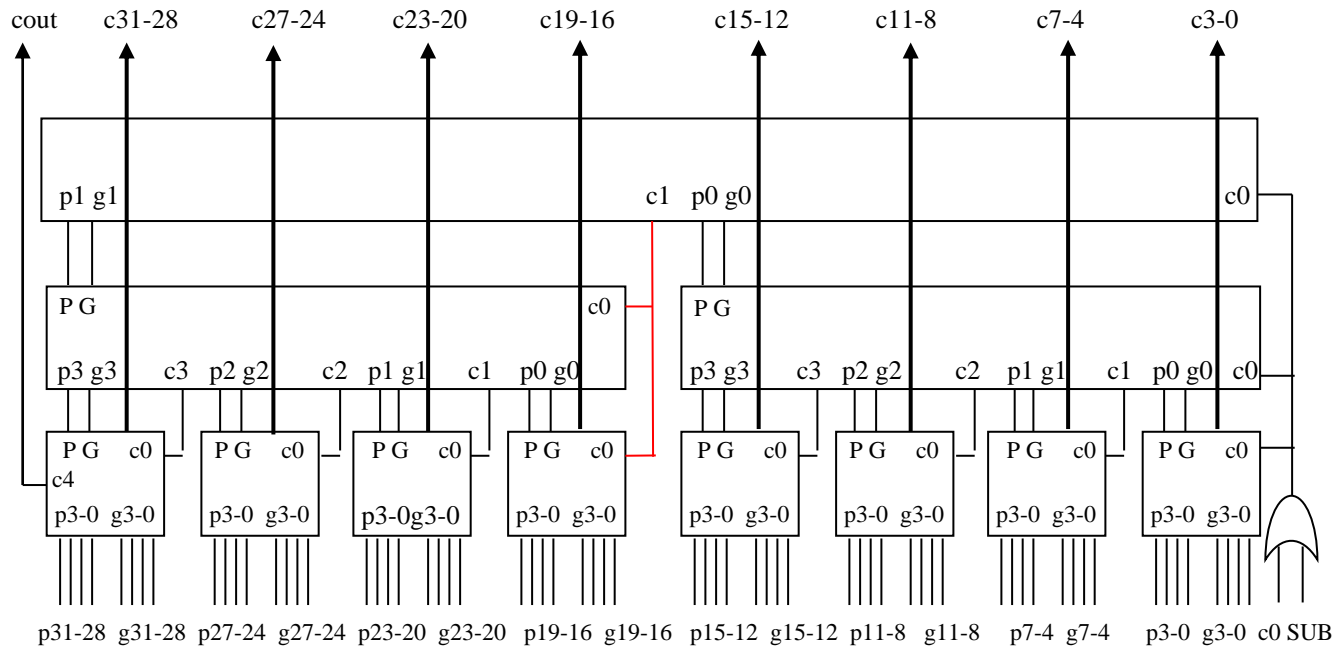
# 块间并行加法器

- 自下而上形成 $p_i g_i$ ，自上而下形成 $c_i$
- 共6级门延迟：
  - 第一层 $p g$ ，第二层 $c(c_4, c_8, c_{12})$ ，第一层 $c(c_1, c_2, c_3, c_5, \dots)$



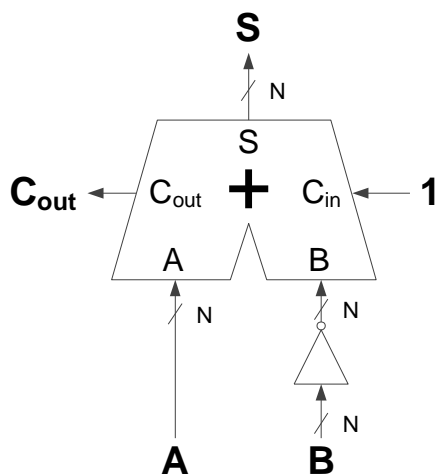
# 32/64位加法器

- 自下而上形成pigi, 自上而下形成ci
- 共10级门延迟:
  - 第一层pg, 第二层pg, 第三层c(c16, c32, ...), 第二层c(c20, c24, ...), 第一层c(c21, c22, c23, c25, ...)

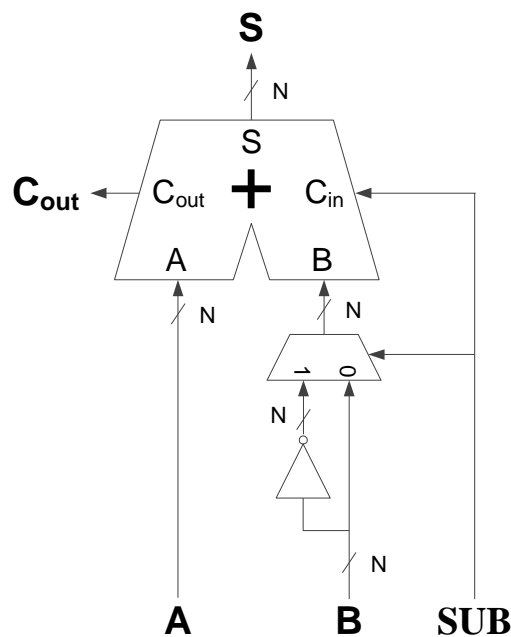


# 补码减法算法

- $[A]_{\text{补}} - [B]_{\text{补}} = [A - B]_{\text{补}} = [A]_{\text{补}} + [-B]_{\text{补}}$ 
  - $[-B]_{\text{补}}$  的计算:  $[B]_{\text{补}}$  “取反加1”
- 只要在B的输入端对B进行取反并置进位为1.



(a)



(b)



# 溢出判断

- 加法: A和B的符号位相同, 但结果的符号位与A和B的符号位不同, 即正数相加得负或负数相加得正

$$ov = s_{31} * ^a_{31} * ^b_{31} + ^s_{31} * a_{31} * b_{31}$$

- 减法: 正数减负数结果为负数或负数减正数结果为正数

$$ov = s_{31} * ^a_{31} * b_{31} + ^s_{31} * a_{31} * ^b_{31}$$

- 因此, 运算器溢出条件为

$$ov = ADD * (s_{31} * ^a_{31} * ^b_{31} + ^s_{31} * a_{31} * b_{31}) + \\ SUB * (s_{31} * ^a_{31} * b_{31} + ^s_{31} * a_{31} * ^b_{31})$$

- 例:

- 1001+0101(-7+5), 0011+0100(3+4)
- 0101+0101(5+5), 1100+1100(-4+(-4))
- 0101-0011(5-3), 0011-0101(3-5)
- 1100-0101(-4-5), 0101-1100(5-(-4))

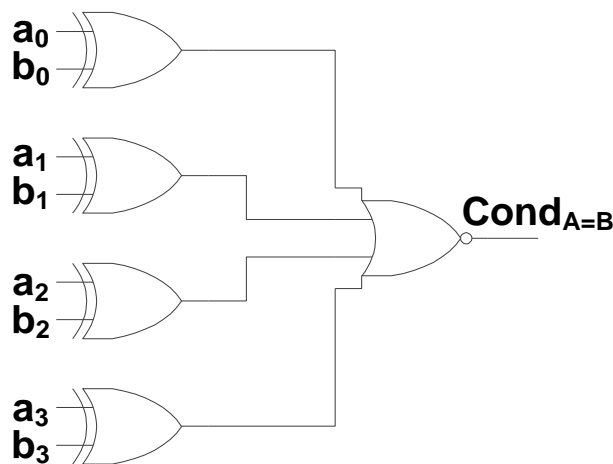
# 定点ALU设计

# ALU的实现

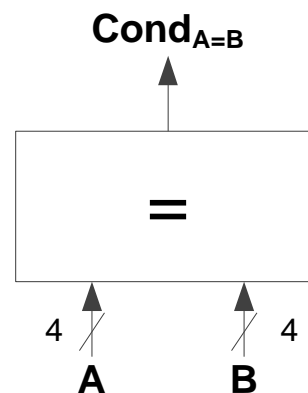
- **ALU表示算术逻辑单元**
  - 实现加减法器
  - 实现逻辑运算 ( $a \& b$ ,  $a | b$ ,  $a \text{ xor } b$ 在加法器中产生)
  - 实现比较器 (相等、大小)
  - 实现移位器
- 最后, 根据操作类型, 从多个结果中选择

# 判断相等

- 判断多bit的A信号和B信号是否相等:  $A_{0-n} == B_{0-n}$ 
  - 使用异或逻辑逐bit的判断( $A_0 \wedge B_0, A_1 \wedge B_1, \dots, A_n \wedge B_n$ )
- 每个bit结果, 有任何一个为1, 则输出为0
  - 多输入或非门, 位数多时需要多级逻辑



(a)



(b)

# 判断大小

- 使用**A-B**来判断大小
  - **A-B > 0** （结果符号位为0）则代表**A**大于**B**
- 小心溢出

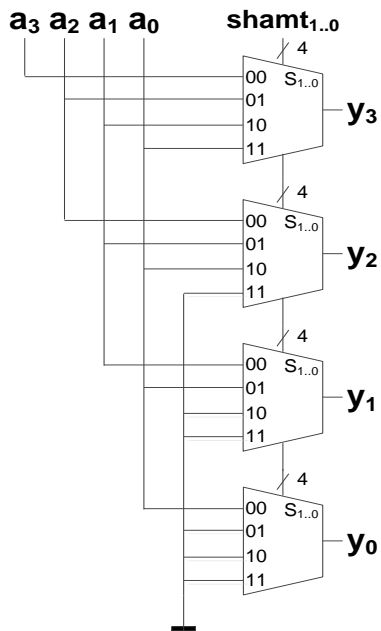
$$\begin{aligned}\text{Cond}_{A<B} &= \sim Ov \ \& \ s_{63} \mid Ov \ \& \ \sim s_{63} \\ &= a_{63} \ \& \ s_{63} \mid \sim b_{63} \ \& \ s_{63} \mid a_{63} \ \& \ \sim b_{63}\end{aligned}$$

# 移位操作

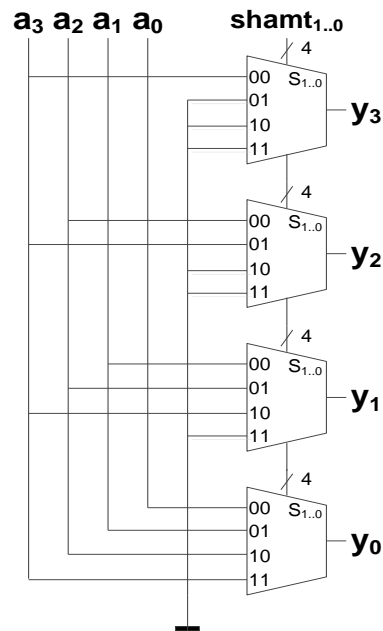
- 移位操作：同时也是乘以/除以 2的幂次的运算
  - 逻辑左移 （低位补0）
  - 逻辑右移 （高位补0）
  - 算数右移 （高位补符号位）
  - 循环右移 （高位补右侧挤掉的数据）
- 例：将32位数 `0xABCD1234` 移8位
  - 逻辑左移 `0xCD123400`
  - 逻辑右移 `0x00ABCD12`
  - 算数右移 `0xFFABCD12`
  - 循环右移 `0x34ABCD12`

# 移位操作（二）

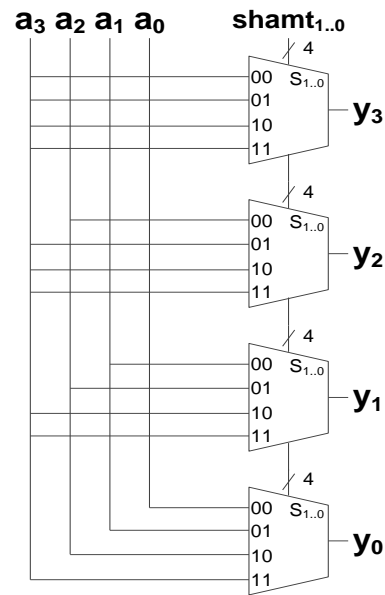
- 对于移N位数的移位操作，使用N选1来实现
  - 根据要移动的位数，从N个输入中选一个
  - 每个输入将输入移动特定位数，不需要延迟和逻辑
- 每种移位结果再根据移位操作类型选择



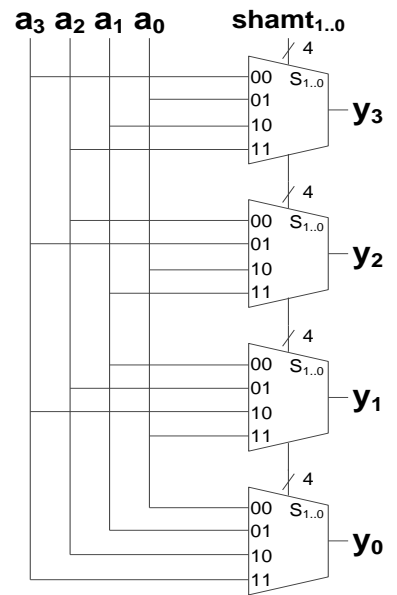
逻辑左移



逻辑右移



算术右移



循环右移

# 补码乘法器设计



# 补码乘法原理

- $[X]_{\text{补}} + [Y]_{\text{补}} = [X+Y]_{\text{补}}$ , 但  $[X]_{\text{补}} * [Y]_{\text{补}} \neq [X*Y]_{\text{补}}$
- 问题: 已知  $[X]_{\text{补}}$  和  $[Y]_{\text{补}}$ , 求  $[X*Y]_{\text{补}}$ .

若  $[Y]_{\text{补}} = y_{31}y_{30}\dots\dots y_1y_0$ ,

则  $Y = -y_{31}*2^{31} + y_{30}*2^{30} + \dots\dots + y_1*2^1 + y_0*2^0$

$$\begin{aligned}
 [X*Y]_{\text{补}} &= [X*(-y_{31}*2^{31} + y_{30}*2^{30} + \dots\dots + y_1*2^1 + y_0*2^0)]_{\text{补}} \\
 &= [-X*y_{31}*2^{31} + X*y_{30}*2^{30} + \dots\dots + X*y_1*2^1 + X*y_0*2^0]_{\text{补}} \\
 &= [-X*y_{31}*2^{31}]_{\text{补}} + [X*y_{30}*2^{30}]_{\text{补}} + \dots\dots + [X*y_1*2^1]_{\text{补}} + [X*y_0*2^0]_{\text{补}} \\
 &= [X]_{\text{补}} * (-y_{31}*2^{31}) + [X]_{\text{补}} * (y_{30}*2^{30}) + \dots\dots + [X]_{\text{补}} * (y_1*2^1) + [X]_{\text{补}} * (y_0*2^0) ??? \\
 &= [X]_{\text{补}} * (-y_{31}*2^{31} + y_{30}*2^{30} + \dots\dots + y_1*2^1 + y_0*2^0)
 \end{aligned}$$

- 在推导中,  $[X]_{\text{补}}$  符号位扩充到64位
- 若  $X = -x_{31}x_{30}\dots\dots x_0$ , 则  $[X*2^k]_{\text{补}}$  为对  $00\dots0x_{31}x_{30}\dots x_0 0_{k-1}\dots0_10_0$  (一共64位) 按位取反后加1, 结果为:

$$\begin{aligned}
 &11\dots1 \wedge x_{31} \wedge x_{30}\dots \wedge x_0 1_{k-1}\dots1_11_0+1 \\
 &= 11\dots1 \wedge x_{31} \wedge x_{30}\dots \wedge x_0 0_{k-1}\dots0_10_0 + 10_{k-1}\dots0_10_0 = (11\dots1 \wedge x_{31} \wedge x_{30}\dots \wedge x_0 + 1)0_{k-1}\dots0_10_0
 \end{aligned}$$

这正是  $[X]_{\text{补}} * 2^k$  的结果.

# 补码乘法算法

- $[X*Y]_{\text{补}} = [X]_{\text{补}} * (-y_{31} * 2^{31} + y_{30} * 2^{30} + \dots + y_1 * 2^1 + y_0 * 2^0)$
- 与普通乘法类似，只是符号位乘项要变加为减
- 符号位的特殊性增加了电路复杂度

1011\*1011 (-5\*-5)

```
      1011
    *1011
    -----
+11111011
+1111011
+000000
-11011
    -----
00011001 (25)
```

1011\*0101 (-5\*5)

```
      1011
    *0101
    -----
+11111011
+0000000
+111011
-00000
    -----
11100111 (-25)
```

# Booth算法

- 对 $(-y_{31} * 2^{31} + y_{30} * 2^{30} + \dots + y_1 * 2^1 + y_0 * 2^0)$ 进行变换

$$(-y_{31} * 2^{31} + y_{30} * 2^{30} + \dots + y_1 * 2^1 + y_0 * 2^0)$$

$$= (y_{30} - y_{31}) * 2^{31} + (y_{29} - y_{30}) * 2^{30} + \dots + (y_0 - y_1) * 2^1 + (y_{-1} - y_0) * 2^0$$

- 每一项都一样, 每次看两位

1011\*1011 (-5\*-5)

```

      1011
    *1011
    -----
  -11111011
  +00000000
  +111011
  -11011
  -----
  +00000101
  +00000000
  +111011
  +00101
  -----
  00011001 (25)

```

$y_i$	$y_{i-1}$	操作
0	0	+0
0	1	$+ [X]_{\text{补}}$
1	0	$- [X]_{\text{补}}$
1	1	+0

1011\*0101 (-5\*5)

```

      1011
    *0101
    -----
  -11111011
  +1111011
  -111011
  +11011
  -----
  +00000101
  +1111011
  +000101
  +11011
  -----
  11100111 (-25)27

```

# Booth二位一乘算法

- 对 $(-y_{31} * 2^{31} + y_{30} * 2^{30} + \dots + y_1 * 2^1 + y_0 * 2^0)$ 进行变换

$$(-y_{31} * 2^{31} + y_{30} * 2^{30} + \dots + y_1 * 2^1 + y_0 * 2^0)$$

$$= (y_{29} + y_{30} - 2 * y_{31}) * 2^{30} + (y_{27} + y_{28} - 2 * y_{29}) * 2^{28} + \dots + (y_1 + y_2 - 2 * y_3) * 2^2 + (y_{-1} + y_0 - 2 * y_1) * 2^0$$

- 每一项都一样，每次看三位，只要16项相加

1011\*1011 (-5\*-5)

```

      1011
    *1011
    -----
  -11111011 (110)
  -111011   (101)
  -----
+00000101
+000101
  -----
    00011001 (25)
  
```

$y_{i+1}$	$y_i$	$y_{i-1}$	操作
0	0	0	+0
0	0	1	$+[X]_{\text{补}}$
0	1	0	$+[X]_{\text{补}}$
0	1	1	$+2[X]_{\text{补}}$
1	0	0	$-2[X]_{\text{补}}$
1	0	1	$-[X]_{\text{补}}$
1	1	0	$-[X]_{\text{补}}$
1	1	1	0

1011\*0101 (-5\*5)

```

      1011
    *0101
    -----
+11111011 (010)
+111011   (010)
  -----
    11100111 (-25)
  
```

# Booth两位乘

1010 × 1001 (-6 × -7)

$$\begin{array}{r}
 \phantom{1010} \phantom{\times} \phantom{1001} \phantom{010} \phantom{100} \\
 \phantom{1010} \phantom{\times} \phantom{1001} 1010 \\
 \times \phantom{1010} \phantom{\times} \phantom{1001} 1001 \\
 \hline
 \phantom{1010} \phantom{\times} \phantom{1001} +11111010 (010) \\
 \phantom{1010} \phantom{\times} \phantom{1001} -11010 \phantom{000} (100) \\
 \hline
 \phantom{1010} \phantom{\times} \phantom{1001} +11111010 \\
 \phantom{1010} \phantom{\times} \phantom{1001} +00110 \\
 \hline
 \phantom{1010} \phantom{\times} \phantom{1001} \underline{100101010} (42)
 \end{array}$$

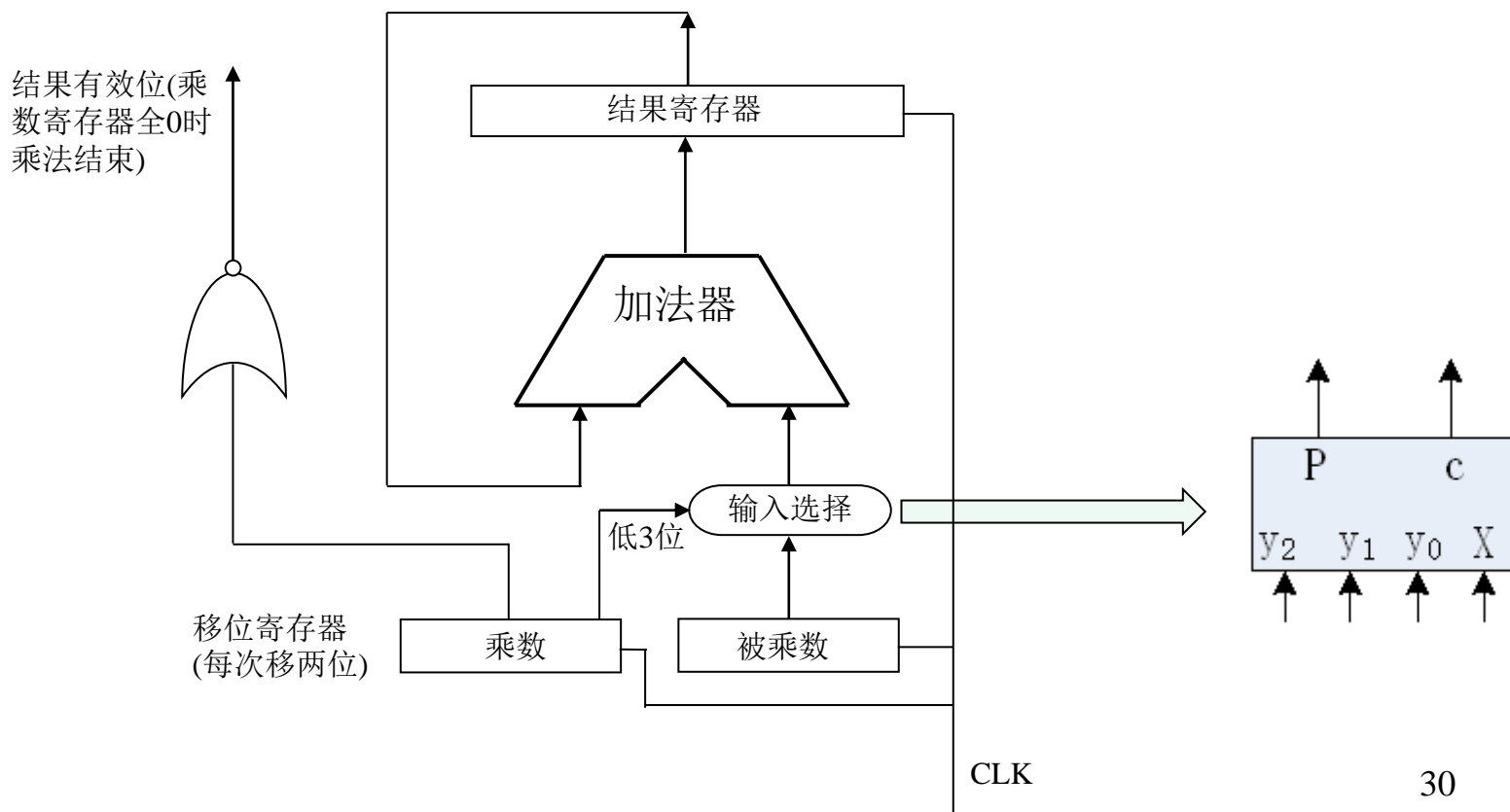
1010 × 0101 (-6 × 5)

$$\begin{array}{r}
 \phantom{1010} \phantom{\times} \phantom{0101} \phantom{010} \phantom{010} \\
 \phantom{1010} \phantom{\times} \phantom{0101} 1010 \\
 \times \phantom{1010} \phantom{\times} \phantom{0101} 0101 \\
 \hline
 \phantom{1010} \phantom{\times} \phantom{0101} +11111010 (010) \\
 \phantom{1010} \phantom{\times} \phantom{0101} +111010 \phantom{000} (010) \\
 \hline
 \phantom{1010} \phantom{\times} \phantom{0101} +11111010 \\
 \phantom{1010} \phantom{\times} \phantom{0101} +111010 \\
 \hline
 \phantom{1010} \phantom{\times} \phantom{0101} \underline{111100010} (-30)
 \end{array}$$

- 循环次数降低一倍
- 每次循环算法一样
- $[X]_{\text{补}}$  只有移1位和补码加减运算 (两位一乘但不用乘3)

# Booth算法的串行实现

- 以二位一乘为例，32位定点乘法需要把16个数相加
  - 可以用一个加法器加15次，需要15个时钟周期
- 与普通移位乘法器的输入选择不同：普通无符号移位乘法器的输入选择为32位门



# Booth两位乘电路

- $[X]$ 补表示为 $x_7x_6x_5x_4x_3x_2x_1x_0$
- 部分积表示为 $p_7p_6p_5p_4p_3p_2p_1p_0+c$ 
  - 补码减法 = 按位取反再 + 1
  - 注意左移的处理方式

$$p_i = \begin{cases} \sim x_i, & \text{选择 } -X \\ \sim x_{i-1}, & \text{选择 } -2X \\ x_i, & \text{选择 } X \\ x_{i-1}, & \text{选择 } 2X \\ 0, & \text{选择 } 0 \end{cases}$$

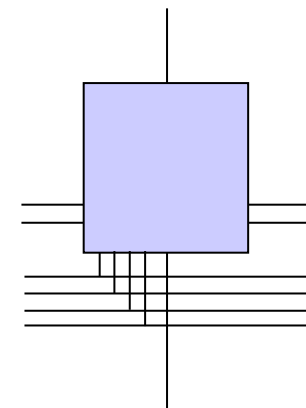
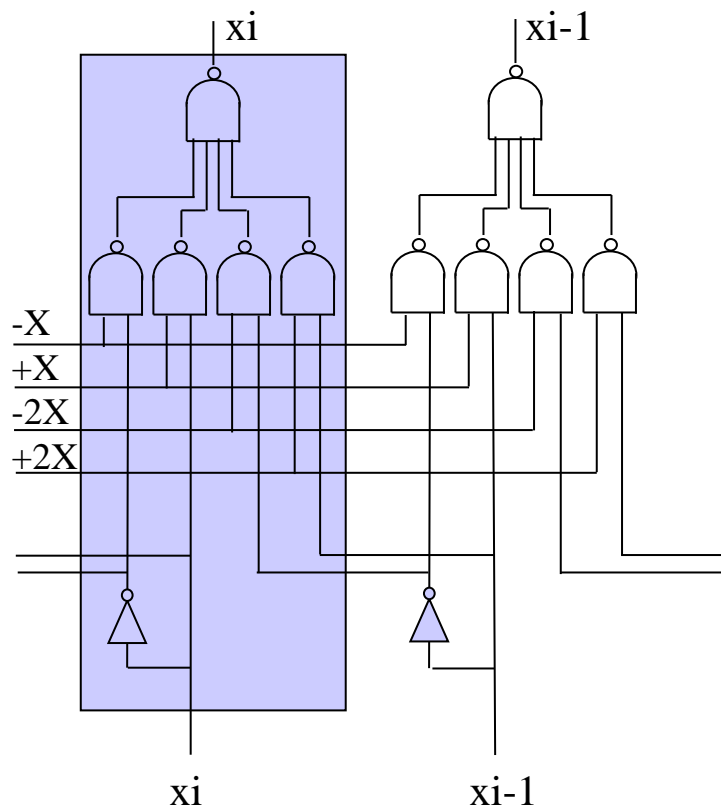
$$c = \begin{cases} 1, & \text{选择 } -X \text{ 或 } -2X \\ 0, & \text{选择 } X \text{ 或 } 2X \text{ 或 } 0 \end{cases}$$

$y_{i+1}$	$y_i$	$y_{i-1}$	操作
0	0	0	+0
0	0	1	$+ [X]_{\text{补}}$
0	1	0	$+ [X]_{\text{补}}$
0	1	1	$+ 2[X]_{\text{补}}$
1	0	0	$- 2[X]_{\text{补}}$
1	0	1	$- [X]_{\text{补}}$
1	1	0	$- [X]_{\text{补}}$
1	1	1	0

# Booth二位乘的输入选择逻辑

- 其中一位

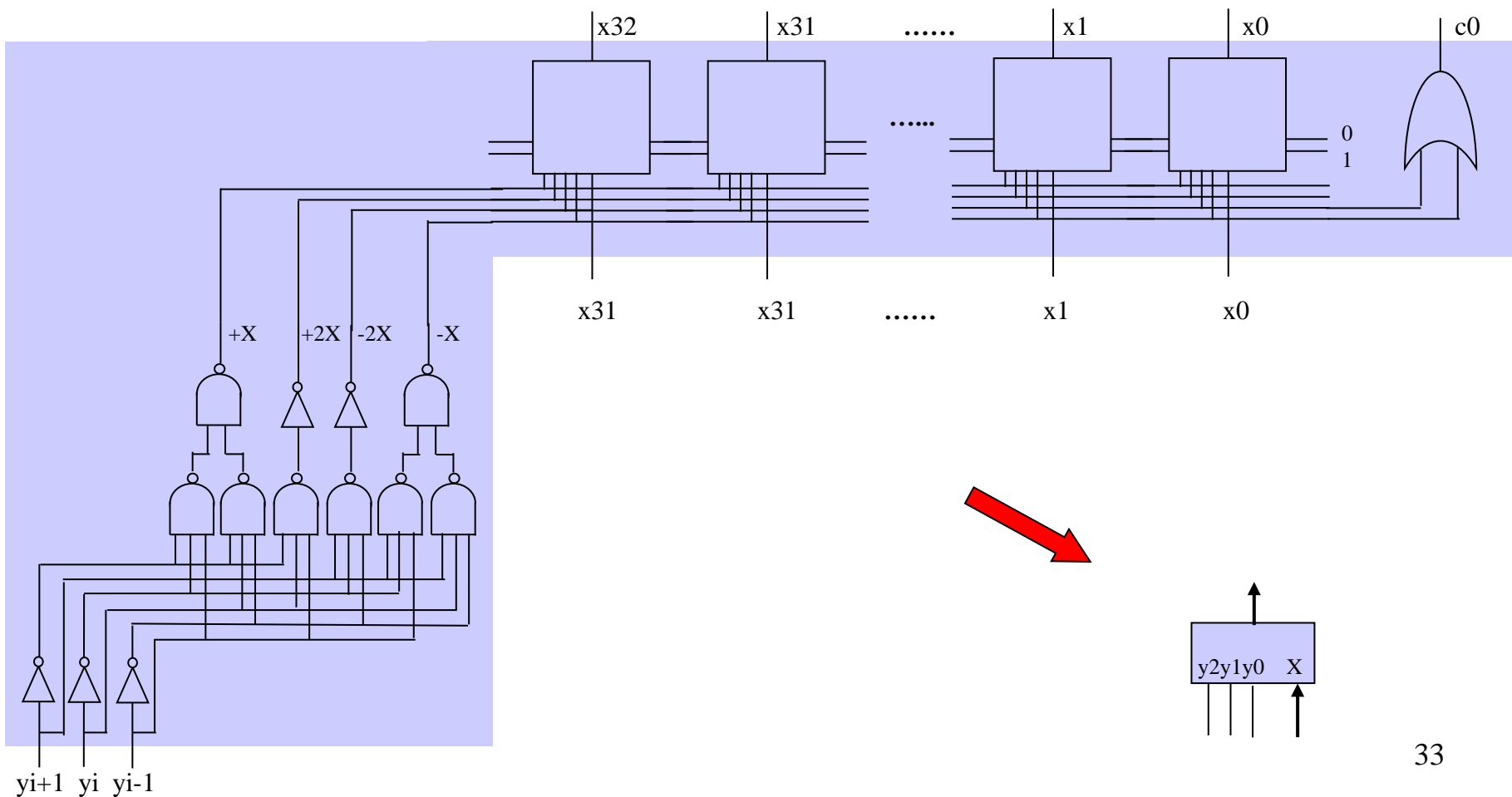
$y_{i+1}$ $y_{i-1}$	$y_i$	操作
0 0	0	$+0$
0 0	1	$+ [X]_{\text{补}}$
0 1	0	$+ [X]_{\text{补}}$
0 1	1	$+ 2[X]_{\text{补}}$
1 0	0	$- 2[X]_{\text{补}}$
1 0	1	$- [X]_{\text{补}}$
1 1	0	$- [X]_{\text{补}}$
1 1	1	$0$





# Booth二位乘的输入选择逻辑

- 一组所有位

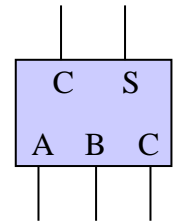
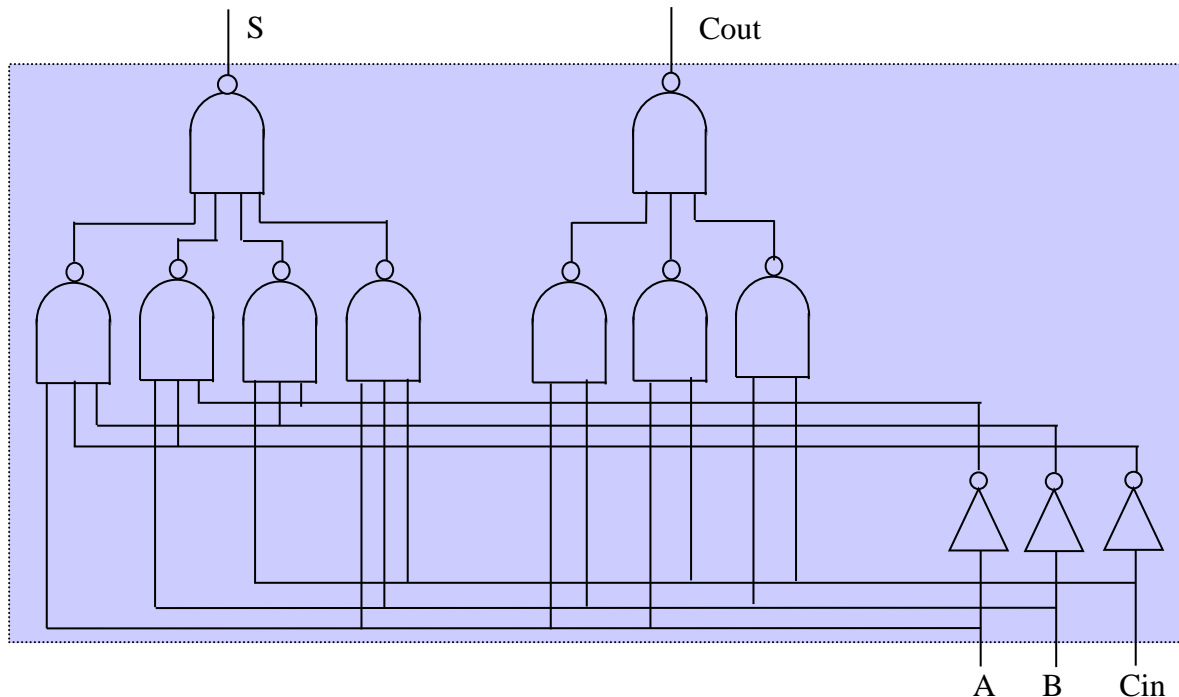


# Wallace加法树

- 串行把16个数相加, 需要15次加法时间
  - 64位加法需要 $2+10+3=15$ 级们延迟
- 用15个加法器组织成树状, 需要4次加法时间, 又浪费硬件
- **Wallace树基本思想**
  - $n$ 个全加器每次把三个 $n$ 位的数相加转换成两个数相加
  - 因此,  $n$ 个全加器每次可以把 $m$ 个 $n$ 位的数相加转换成 $2m/3$ 个数相加, 再用一层全加器转换成 $4m/9$ 个数相加, 直到转换成2个数; 再用加法器把最后两个数相加

# 全加器

- 三个输入，两个输出
- 进位输出在下一级相加时连到下一位
- 两级门延迟



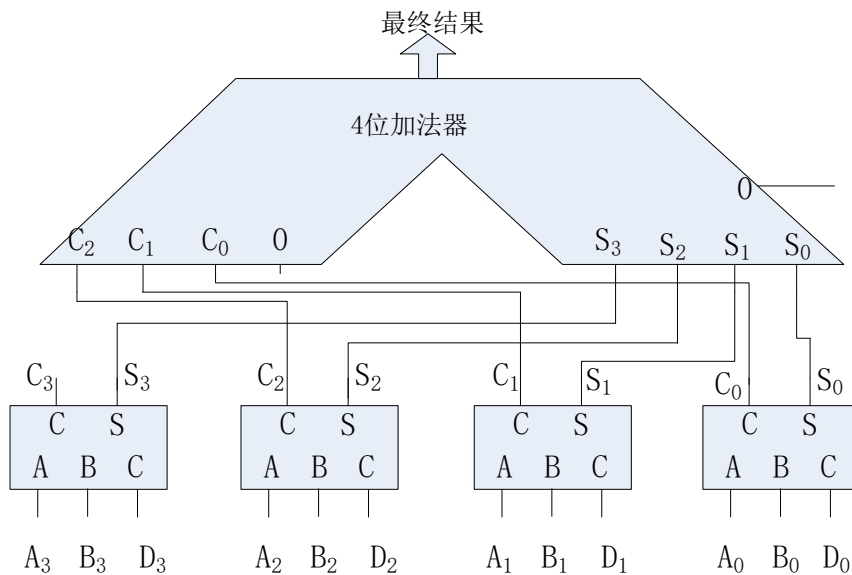
# 全加器把三个加数变成两个加数

x31 x31 x31 x31 x31 x31 x30 x29	.....	x5 x4 x3 x2 x1 x0
x31 x31 x31 x31 x30 x29 x28 x27	.....	x3 x2 x1 x0
x31 x31 x30 x29 x28 x27 x26 x25	.....	x1 x0
-----		
s36 s35 s34 s33 s32 s31 s30 s29	.....	s5 s4 s3 s2 s1 s0
c35 c34 c33 c32 c31 c30 c29 c18	.....	c4 c3 c2 c1 c0



## 例：3个4位数相加

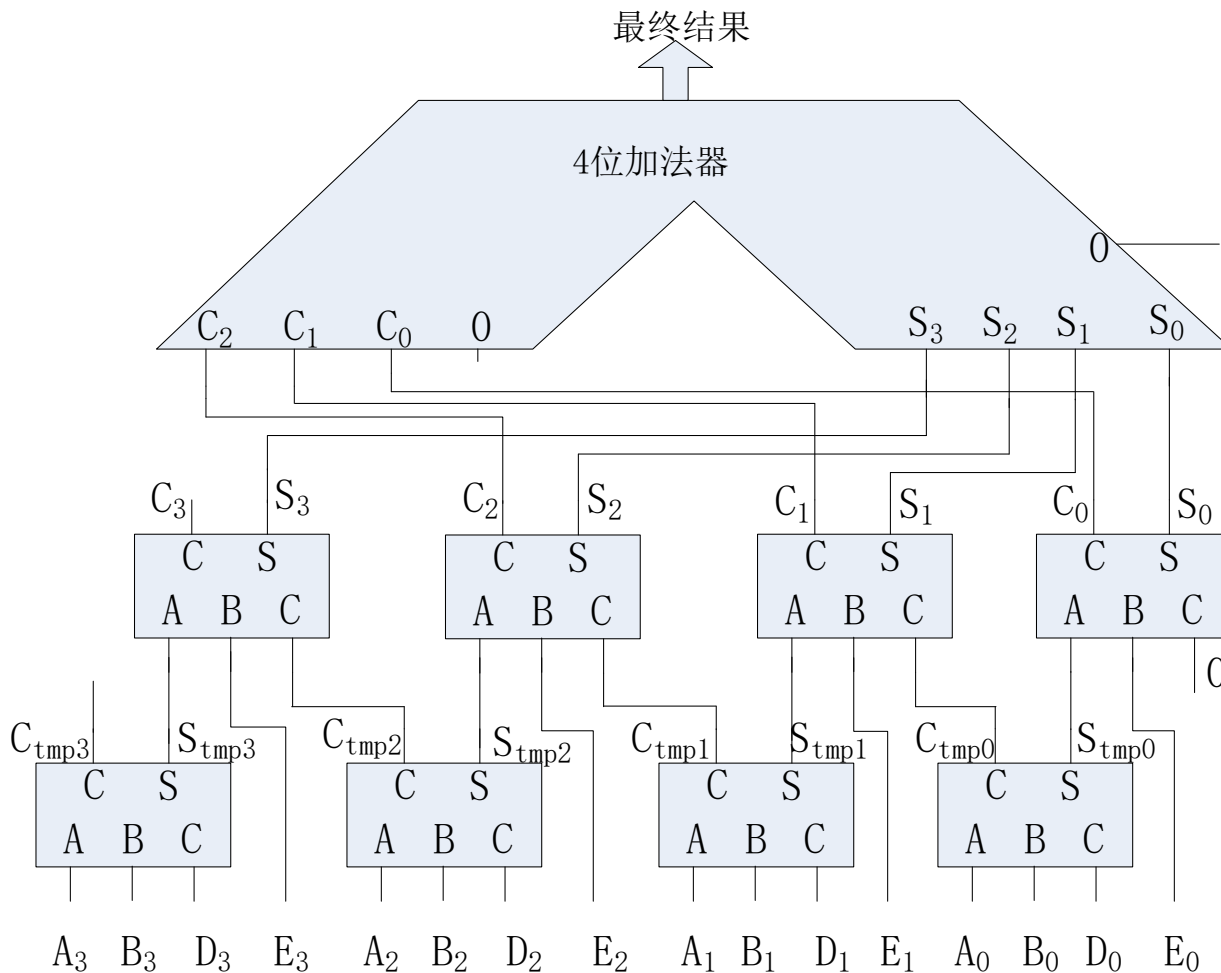
- 3个四位数相加
  - 3个（ABD）4bit数的加法，转换为2个数的加法
  - $A + B + D = S + \{C, 1'b0\}$  （请着重注意连线的方式）
  - 3+15级门延迟 （vs. 15+15级门延迟）



注：

回顾乘法算法，两个N位数相加，只保留N位的结果，高位可以丢弃

# 例：4个4位数相加



注意：

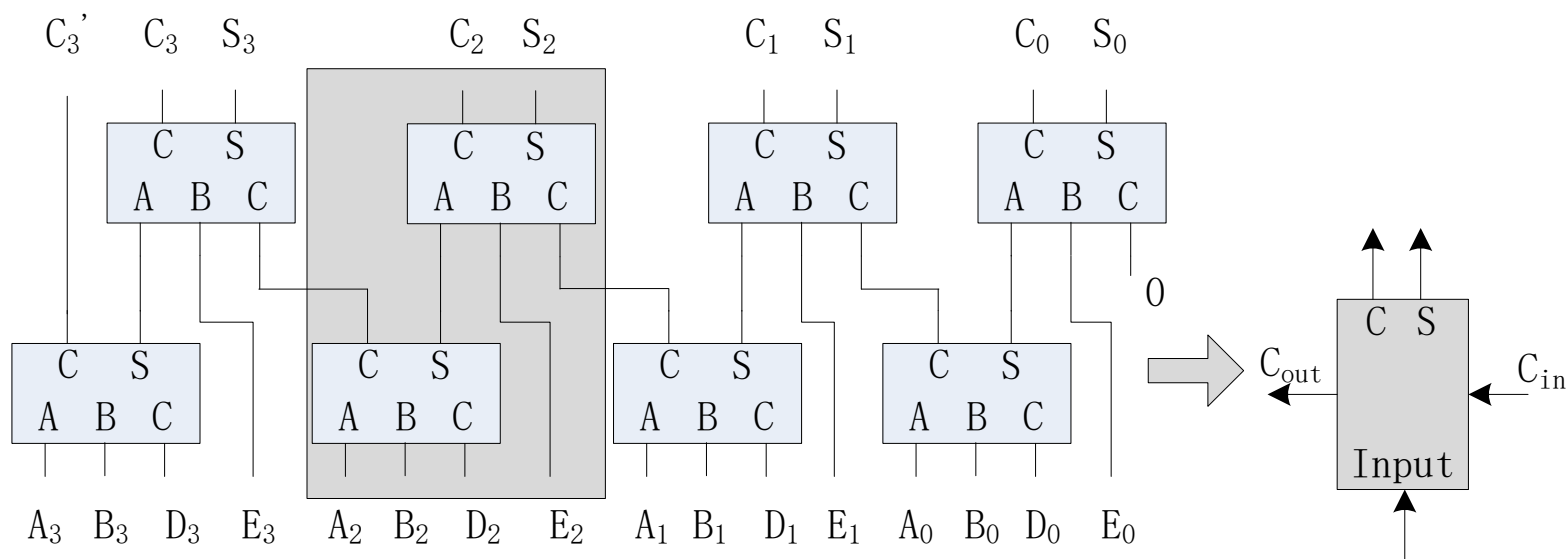
根据公式，

$C_{tmp}$  在参与第二层运算时，已经左移1位

因此， $C_{tmp0}$  要与  $S_{tmp1}$  和  $E_1$  参与同一个全加器

## 其中一位

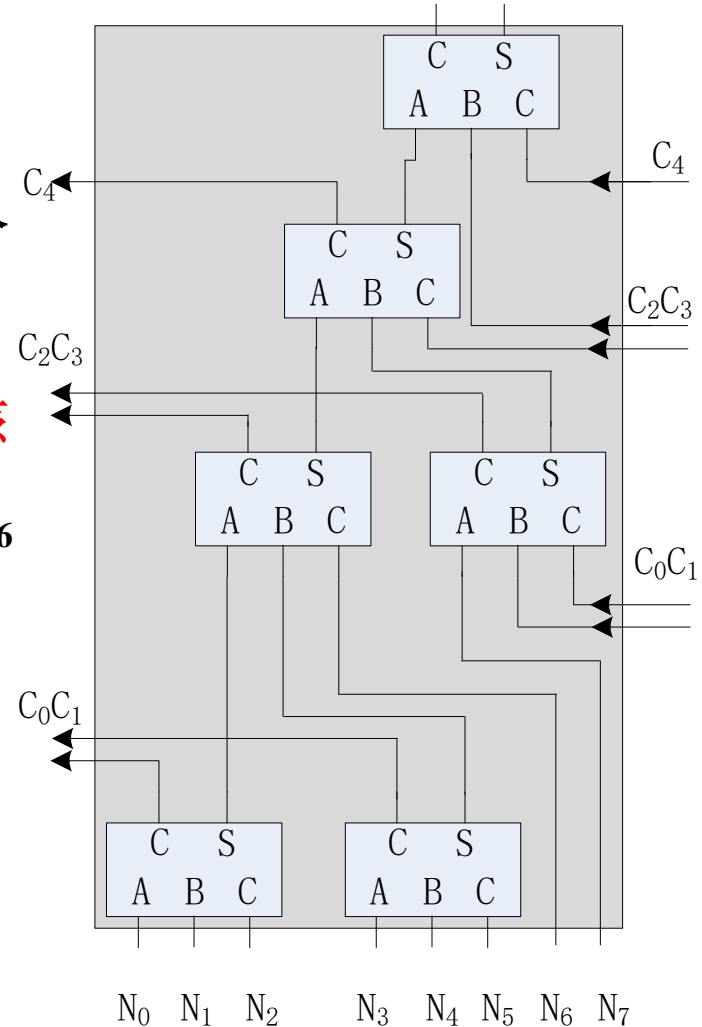
- 把4个数变成2个数相加需要两级全加器
  - 请关注 $C_2$ 和 $S_2$ 的生成逻辑，抽象出来，即为1bit华莱士树



# 1位华莱士树

- 连线规则:

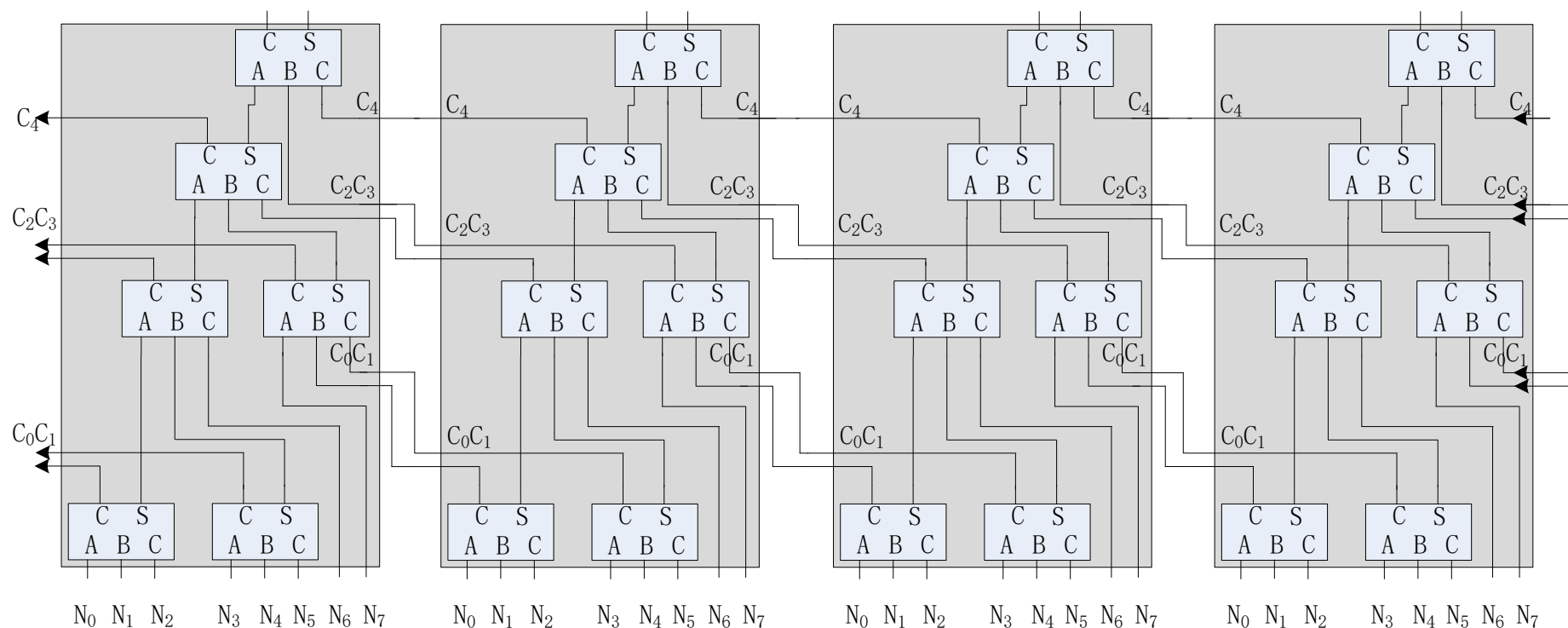
- 每一层可以把 $3n$ 个加数变成 $2n$ 个加数
- 除最高层全加器外，进位输入和进位输出个数相等
- 接收的进位输入信号，必须接到进位输出该信号层次的上一层或更高（例如第一层的 $N_6$ 和 $N_7$ ，以及第3层的 $C_3$ ）





# 多位华莱士树

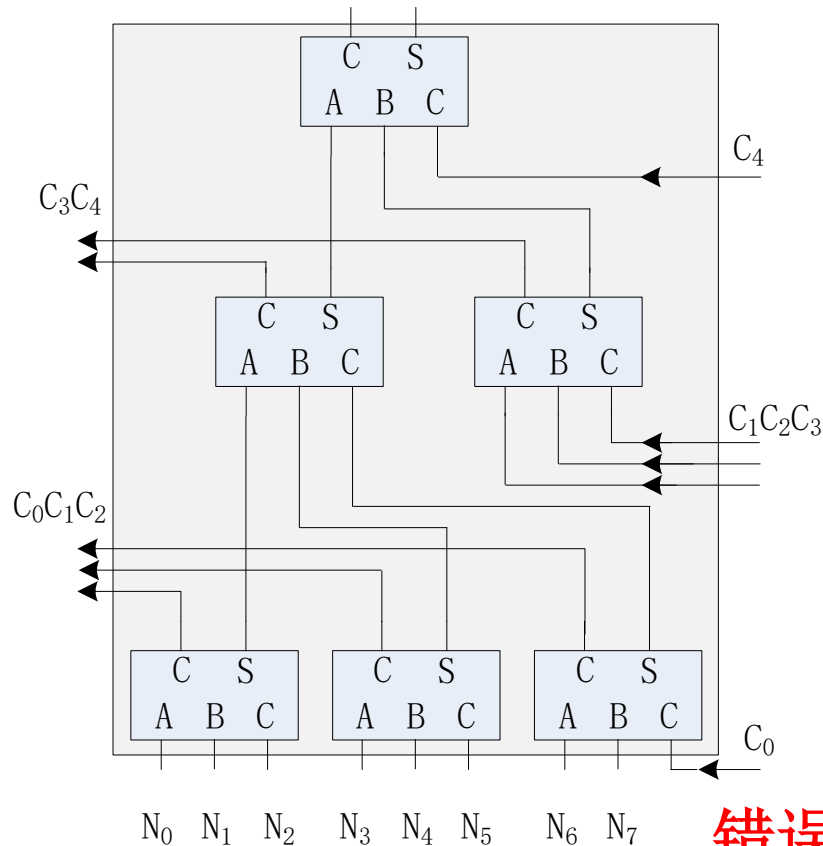
将多个1bit华莱士树连接，将多个多位数相加转为2个多位数相加



可以看出：任意S和C结果，最多需要4级全加器的延迟

# 华莱士树进位连接要求

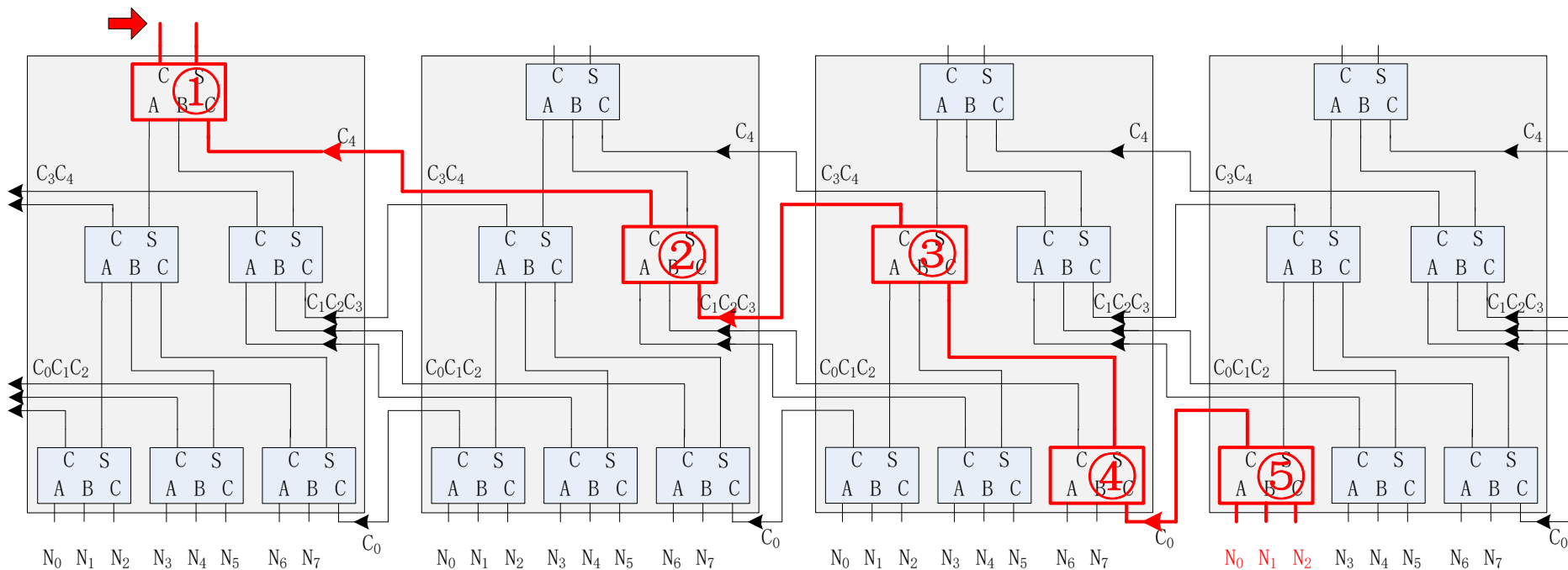
- 进位输入信号必须接到进位输出该信号层次的上一层或更高
  - 下图也使用全加器，将8个数变为2个数：



- 看似3级延迟
- 也是6个全加器
- 进位输入和输出等宽
- 为何不使用这个更快的逻辑？

错误范例请勿模仿

# 偷鸡不成蚀把米



- 把多个树连起来，算算延迟：

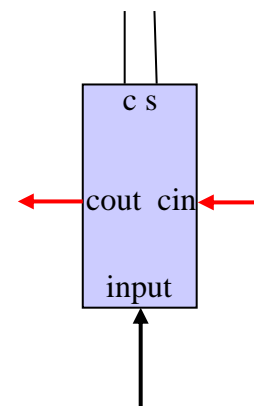
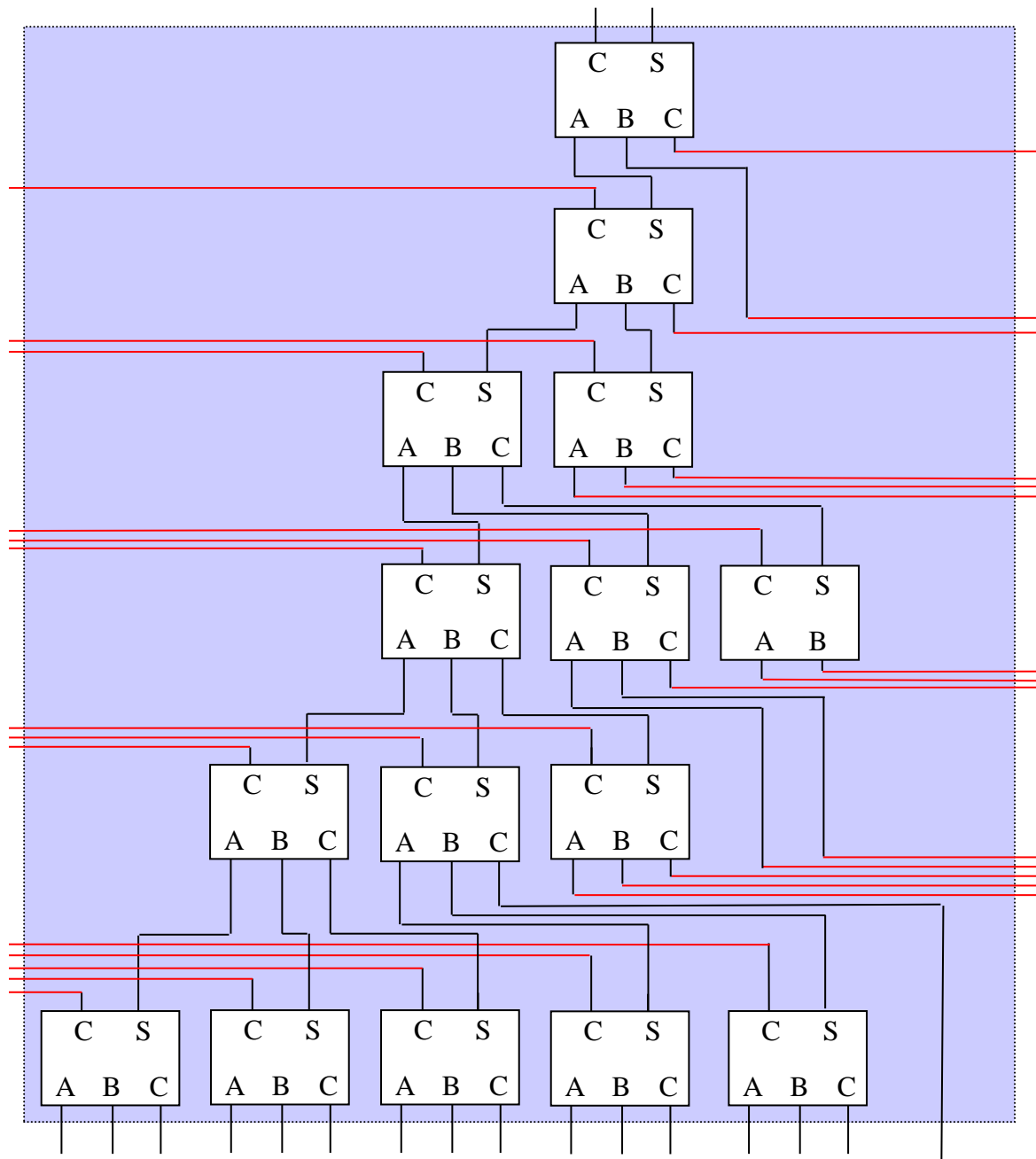
1. 输出依赖于①号全加器
2. ① 依赖上个树进位的 $C_4$ ，由②号全加器计算
3. ② 依赖于上个树的 $C_3$ ，由③号全加器计算
4. ③ 依赖于当前树中的④号全加器
5. ④ 依赖于上个树的 $C_0$ ，由⑤号全加器计算

- 完成一次进位传递需要5级全加器

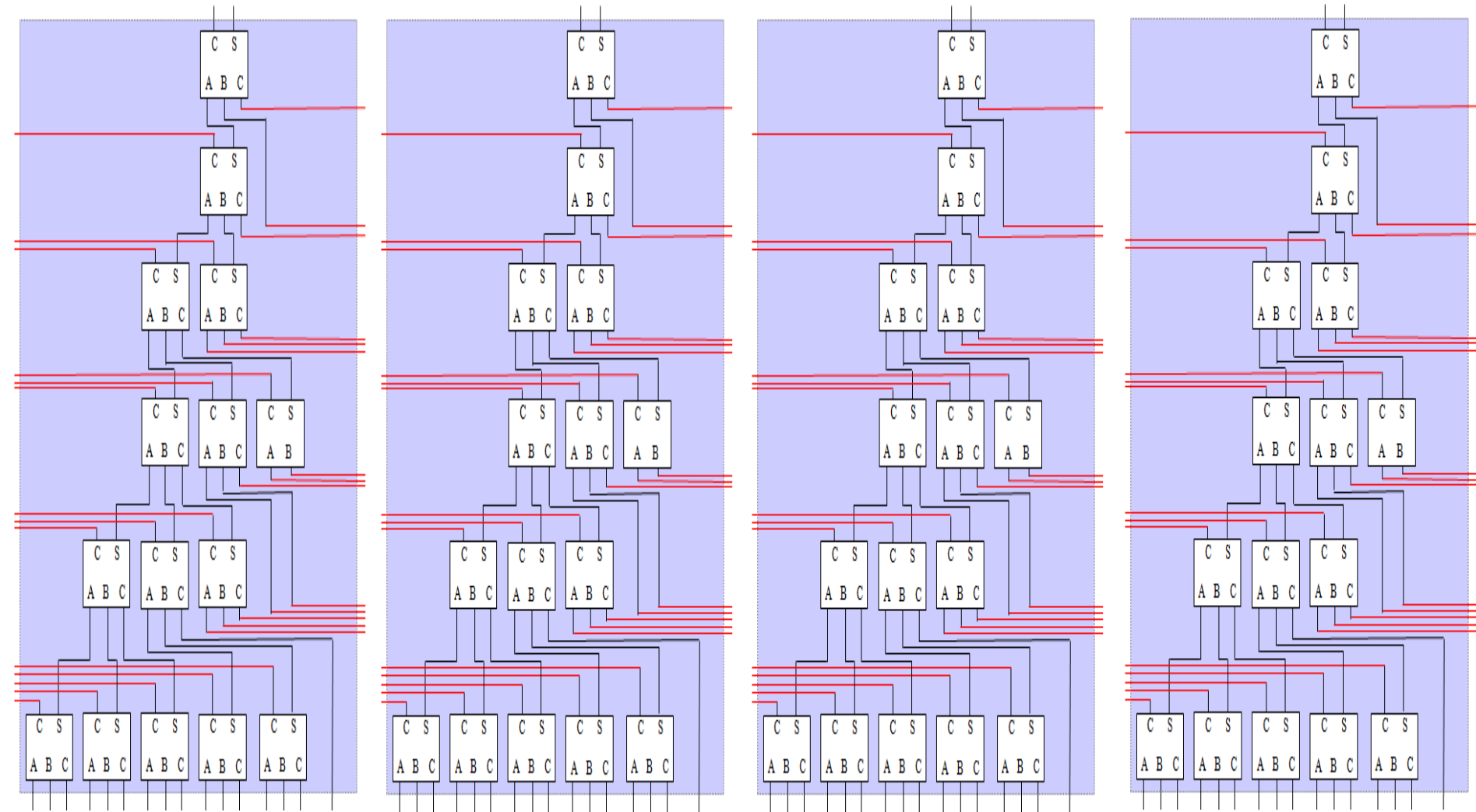
- 偷鸡不成，反蚀把米！

- 问题在于：同一层运算使用了同一层的进位

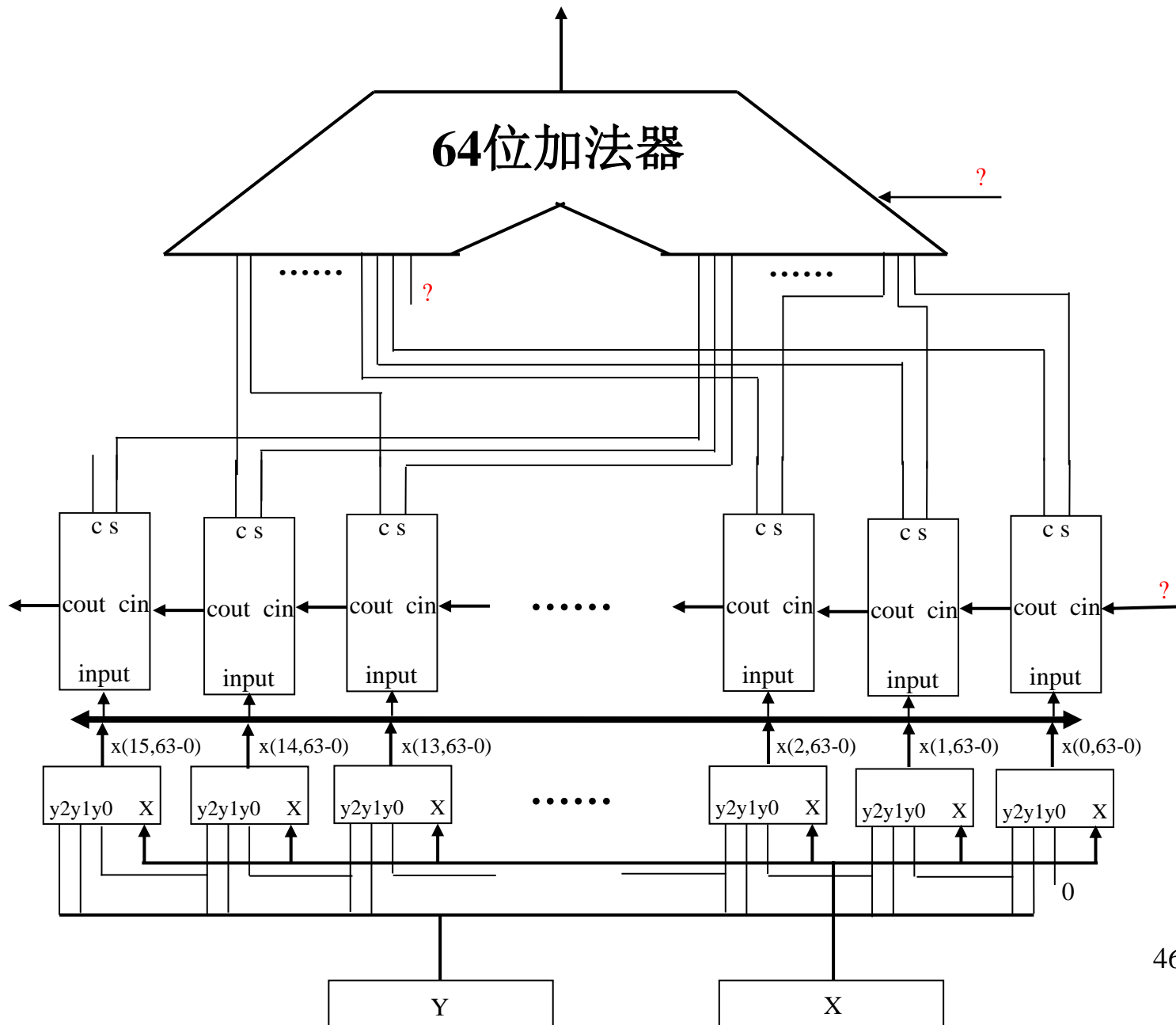
# 16个数相加的1位加法树



# 16个数相加的四位华莱士树（左右进位相连）



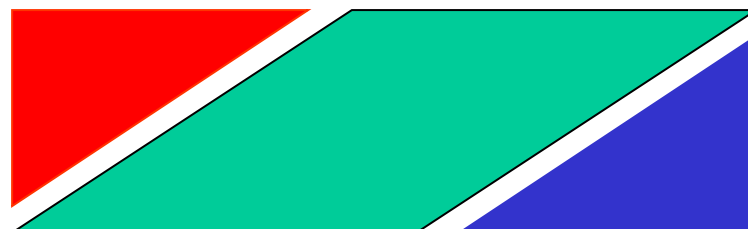
# 32\*32补码乘法器框图





# 说明

- 32位的 $[X]_{\text{补}}$ ,  $[-X]_{\text{补}}$ ,  $[2X]_{\text{补}}$ ,  $[-2X]_{\text{补}}$  如何扩展到64位
  - $[X]_{\text{补}}$ : 左边符号位扩充, 右边补0
  - $[-X]_{\text{补}}$ : 对 $[X]_{\text{补}}$ 按位求反, 左边符号位扩充, 右边补1, 末位加1.
- 末位的进位问题
  - 由 $-2[X]_{\text{补}}$ ,  $-[X]_{\text{补}}$ 引起
  - 加法树中的全加器/半加器个数至少是 (相加数的个数-1)
- 硬件优化
  - 低位“0”不用加
  - 高位符号位扩充位可以优化
  - 请查阅相关资料





# 作 业