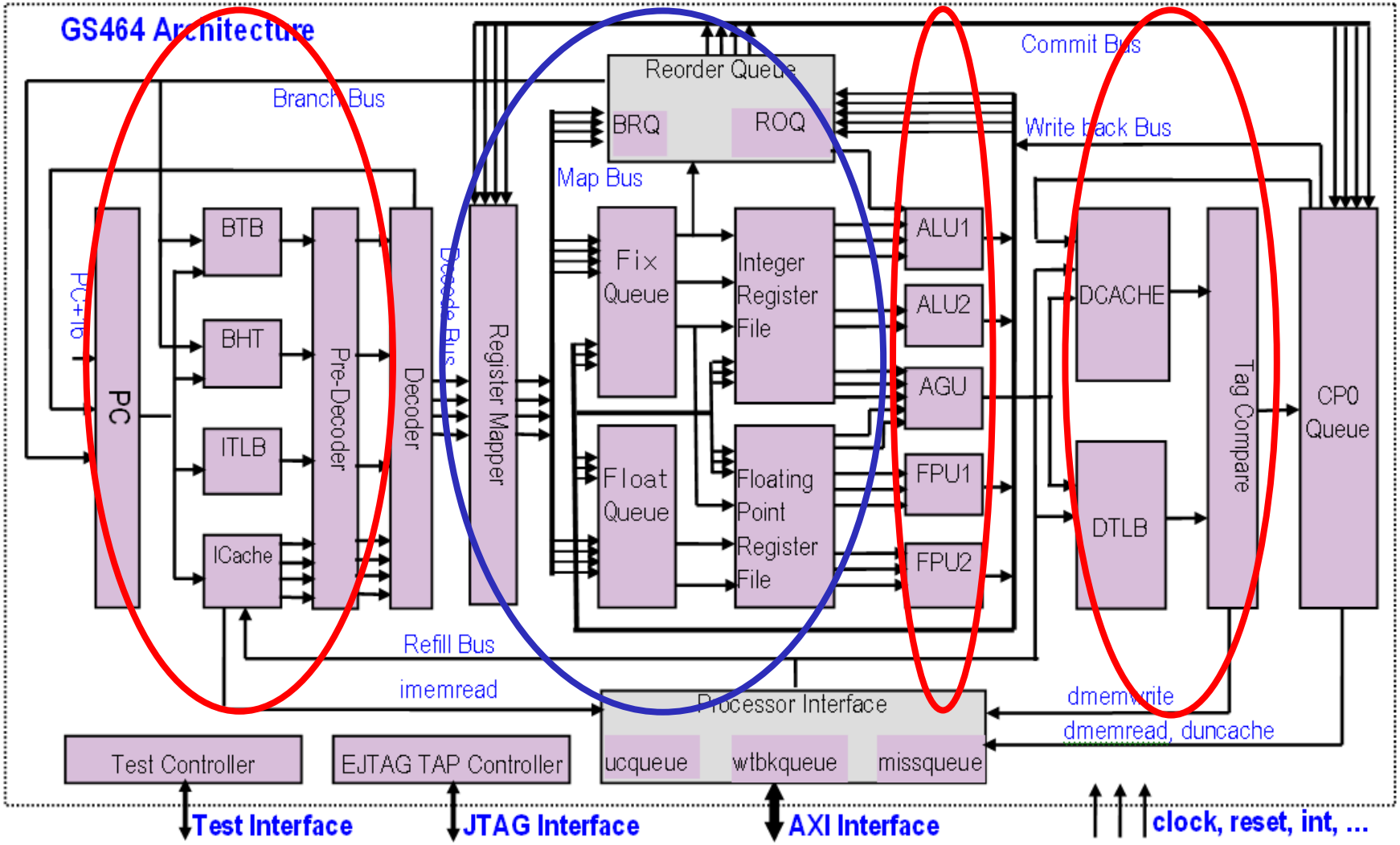
计算机体系结构

胡伟武、汪文祥

龙芯2号处理器核结构图



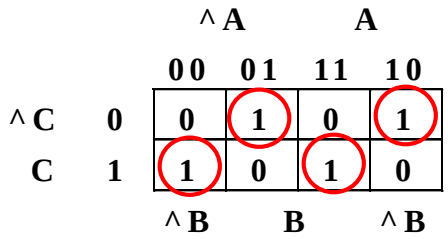
功能部件

- 定点补码加法器设计
- 定点ALU设计
- 定点补码乘法器的设计

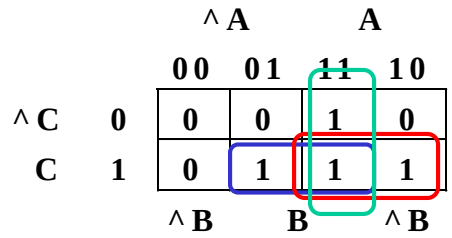
定点补码加法器设计

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**



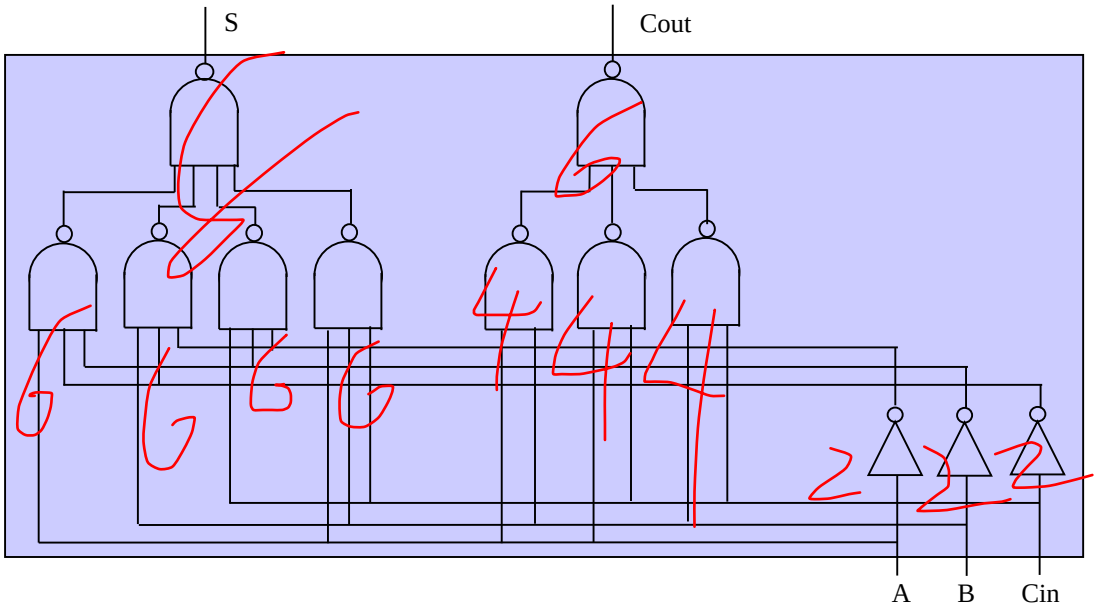
S= ^A*^B*Cin+^A*B*^Cin+A*^B*^Cin+A*B*Cin

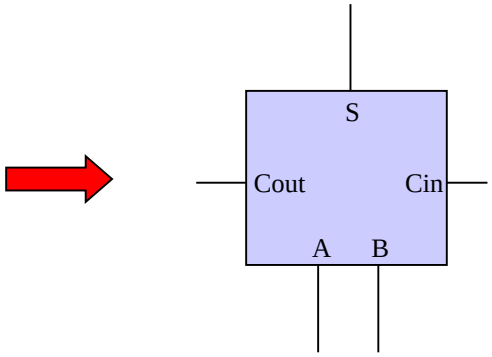


Cout= A*B+A*Cin+B*Cin

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

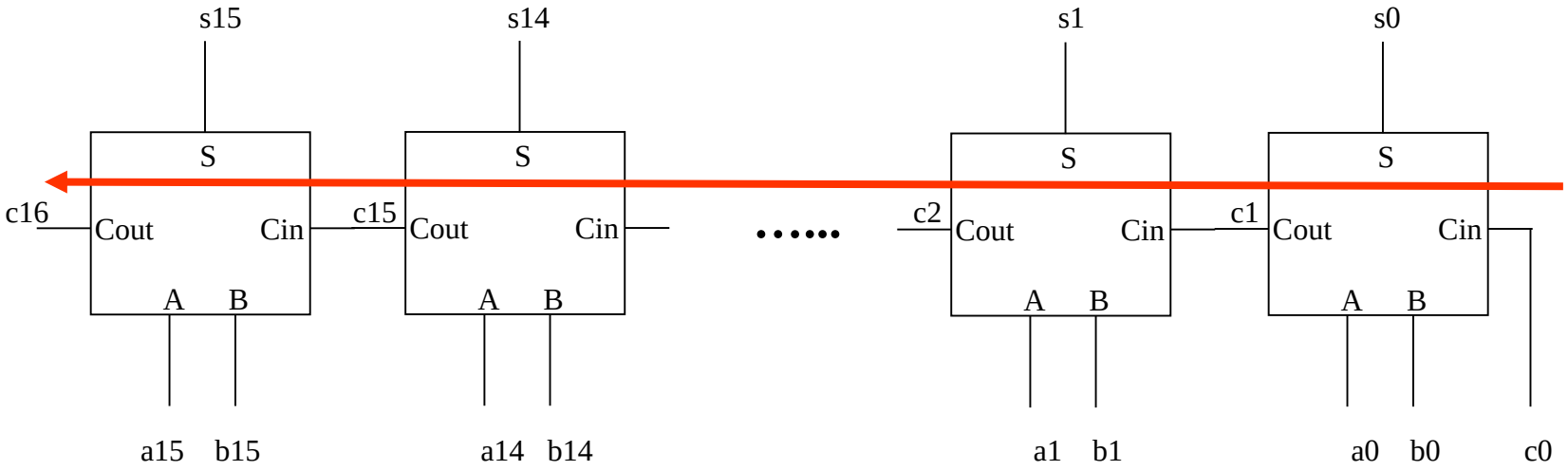
一位全加器





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

$$c_{i+1} = a_i*b_i+a_i*c_i+b_i*c_i$$
$$= a_i*b_i+(a_i+b_i)*c_i$$
$$= g_i+p_i*c_i$$

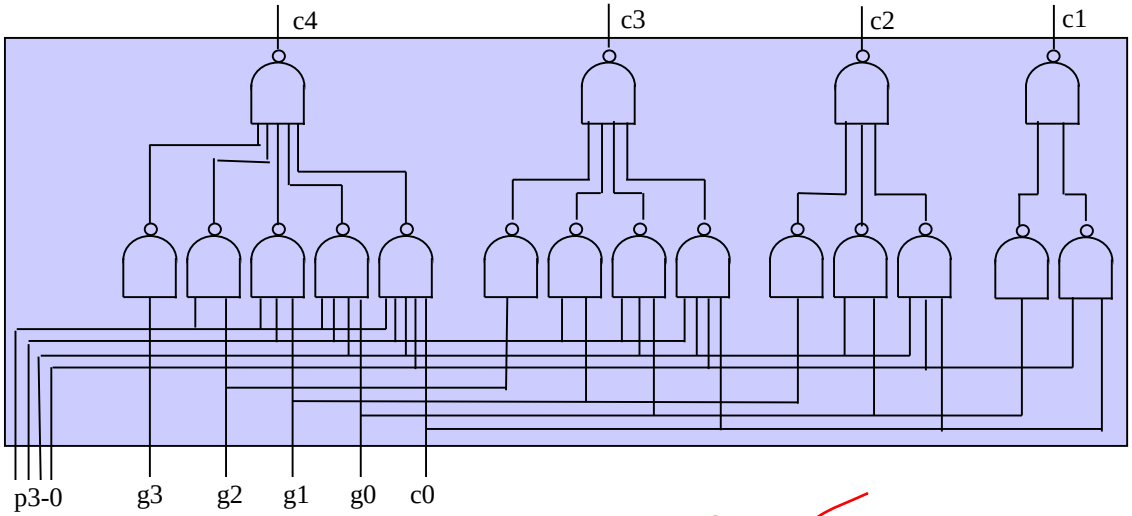
- gi=ai*bi称为进位生成因子，只要gi为1，就有进位
- pi=ai+bi称为进位传递因子，只要pi为1，就把低位进位向前传递
- 四位进位传递为例

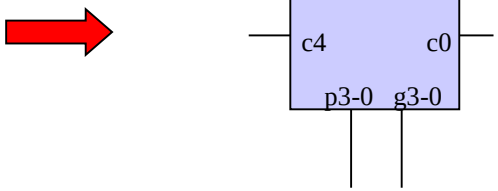
$$c1 = g0+(p0*c0)$$
$$c2 = g1+(p1*g0)+(p1*p0*c0)$$
$$c3 = g2+(p2*g1)+(p2*p1*g0)+(p2*p1*p0*c0)$$
$$c4 = g3+(p3*g2)+(p3*p2*g1)+(p3*p2*p1*g0)+(p3*p2*p1*p0*c0)$$

- 只要低位有一个进位生成，而且被传递，则进位输出为1

4位并行进位逻辑

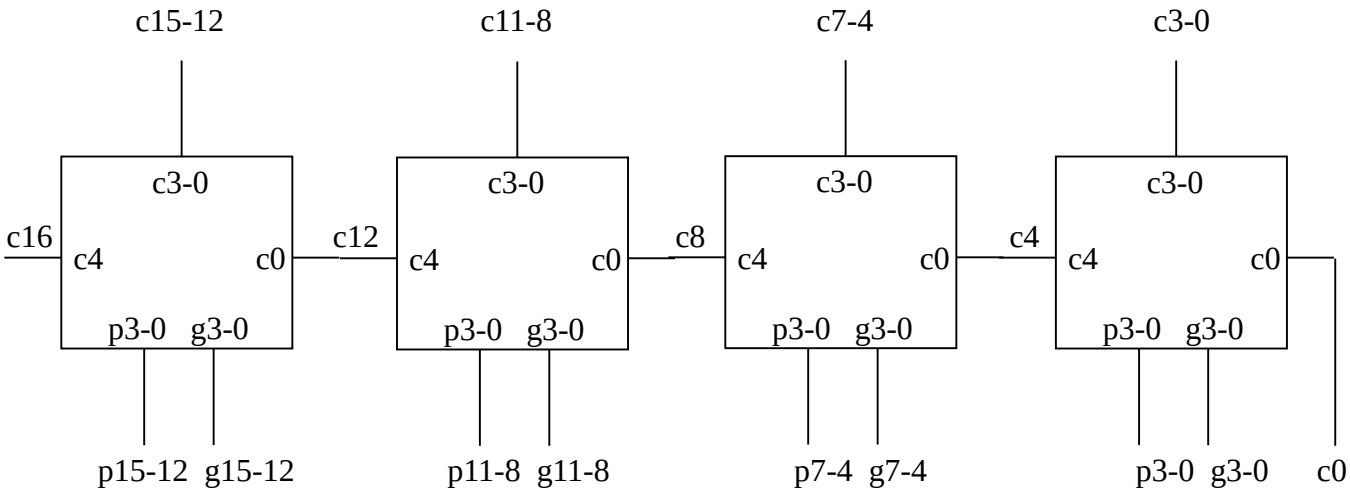
五输入与非门不常用





以16位加法器为例

- 输入为pi、gi，输出为ci
- 每次并行产生4位进位，从pi、gi产生c16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ai、bi产生c16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？



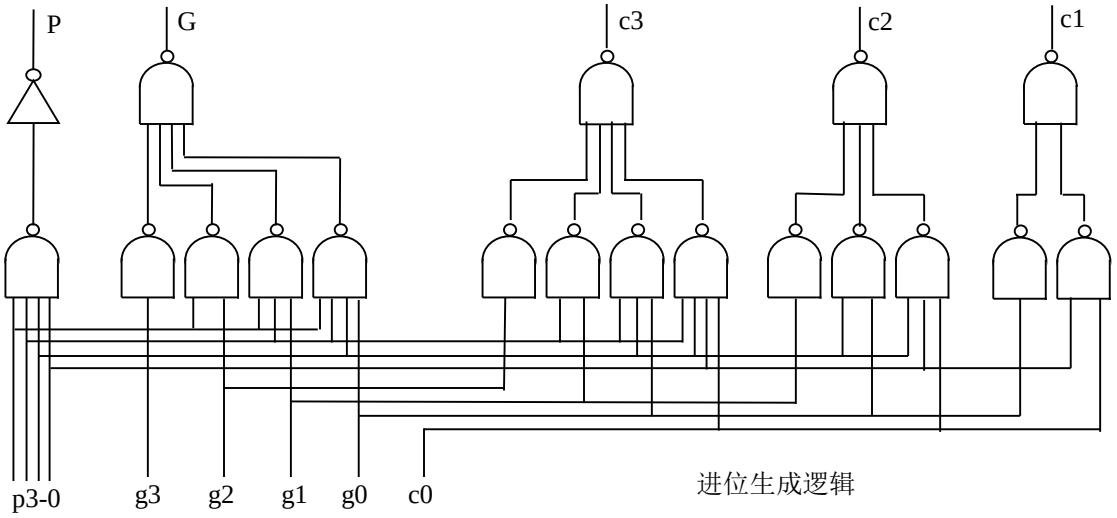
块内并行, 块间并行

- 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

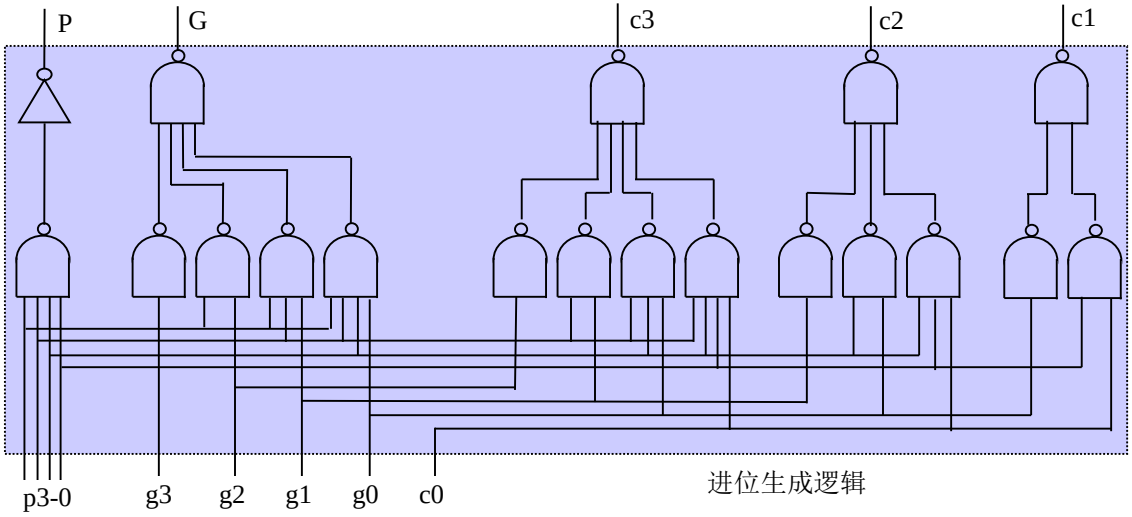
```
P  = p0*p1*p2*p3
```

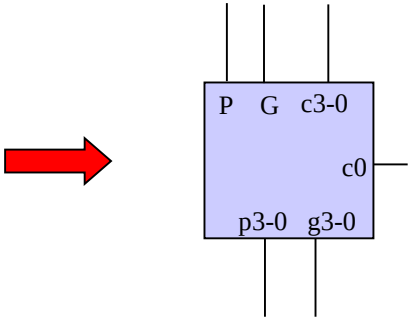
- 进位产生因子：块内产生进位, 不考虑进位输入

```
G  = g3+(p3*g2)+(p3*p2*g1)+(p3*p2*p1*g0)
c4 = g3+(p3*g2)+(p3*p2*g1)+(p3*p2*p1*g0)+(p3*p2*p1*p0*c0)
   = G +(P*c0)
```



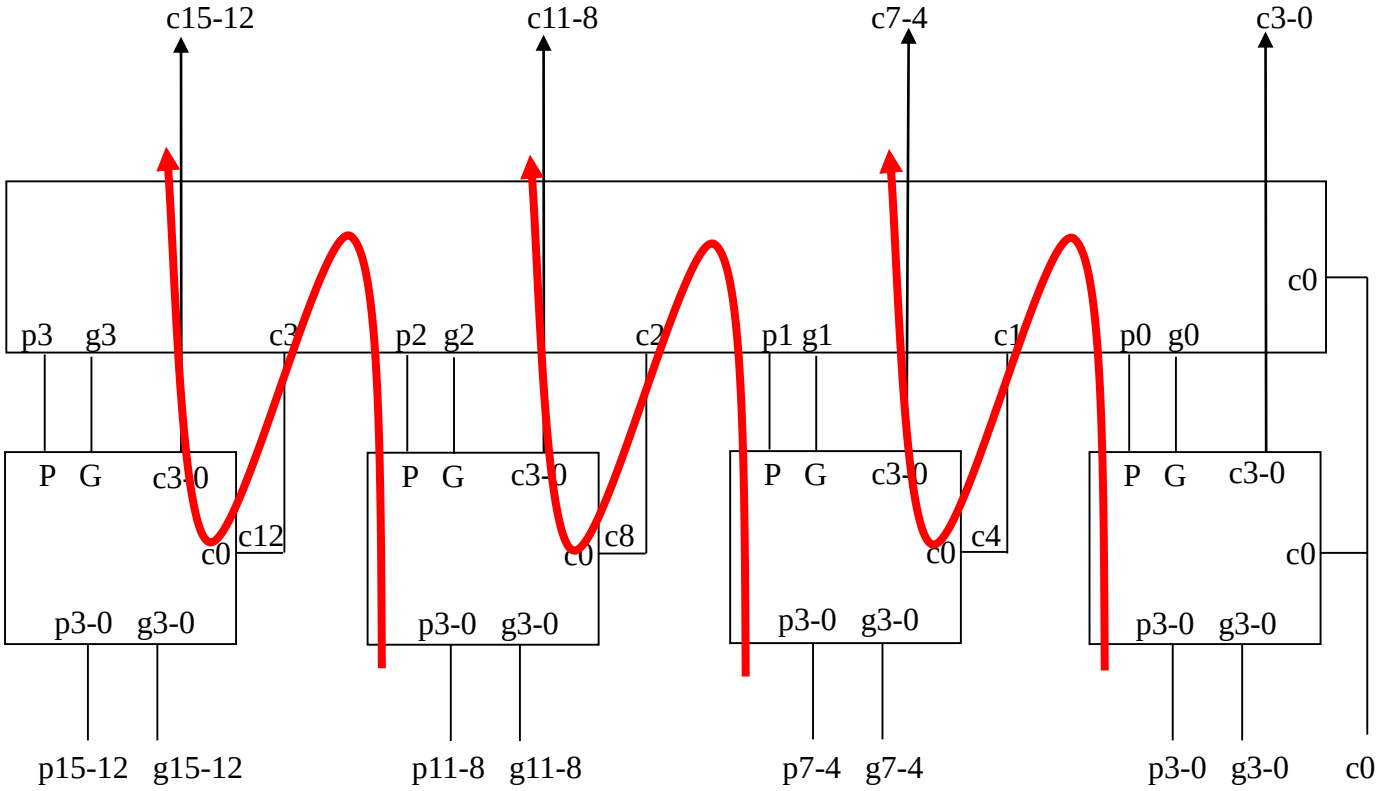
进位生成与传递逻辑





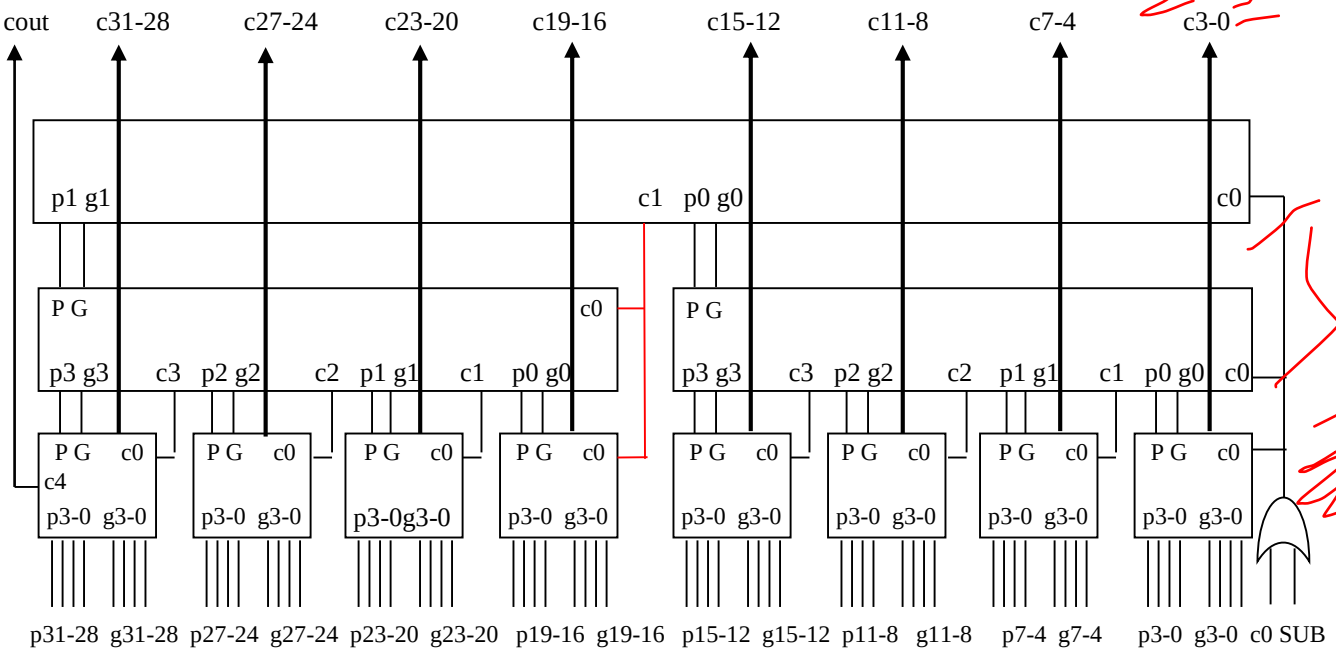
块间并行加法器

- 自下而上形成pigi，自上而下形成ci
- 共6级门延迟：
  - 第一层pg，第二层c，第一层c



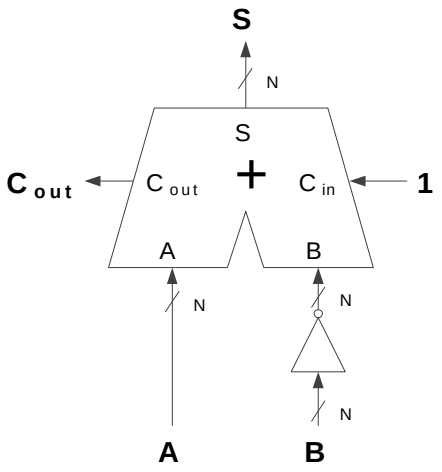
32/64位加法器

- 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**，第二层**c**，第一层**c**

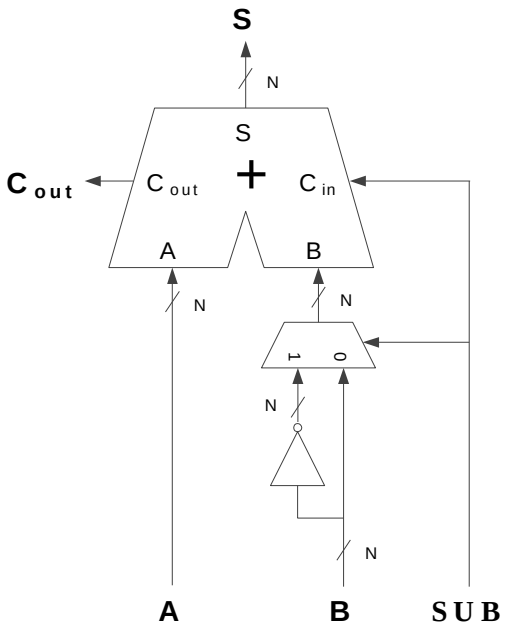


补码减法算法

- $[A]_{补} - [B]_{补} = [A-B]_{补} = [A]_{补} + [-B]_{补}$
  - $[-B]_{补}$ 的计算：$[B]_{补}$ “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=s31*^a31*^b31+^s31*a31*b31

- 减法：正数减负数结果为负数或负数减正数结果为正数

  ov=s31*^a31*b31+^s31*a31*^b31

- 因此，运算器溢出条件为

  ov=ADD*(s31*^a31*^b31+^s31*a31*b31)+

  SUB*(s31*^a31*b31+^s31*a31*^b31)

- 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))**

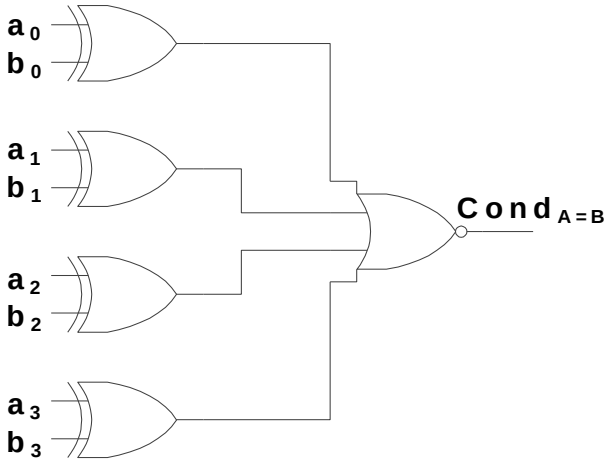
定点ALU设计

ALU的实现

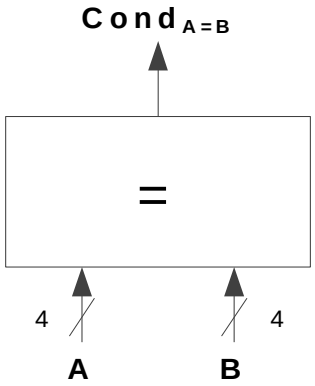
- **ALU表示算术逻辑单元**
  - 实现加减法器
  - 实现逻辑运算（`a&b, a|b, a xor b`在加法器中产生）
  - 实现比较器（相等、大小）
  - 实现移位器
- 最后，根据操作类型，从多个结果中选择

判断相等

- 判断多bit的A信号和B信号是否相等：$A_{0\text{-}n} == B_{0\text{-}n}$
  - 使用异或逻辑逐bit的判断($A_0 \wedge B_0$, $A_1 \wedge B_1$, … , $A_n \wedge B_n$)
- 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

$$Cond_{A<B} = \sim Ov \ \& \ s_{63} \mid Ov \ \& \ \sim s_{63}$$

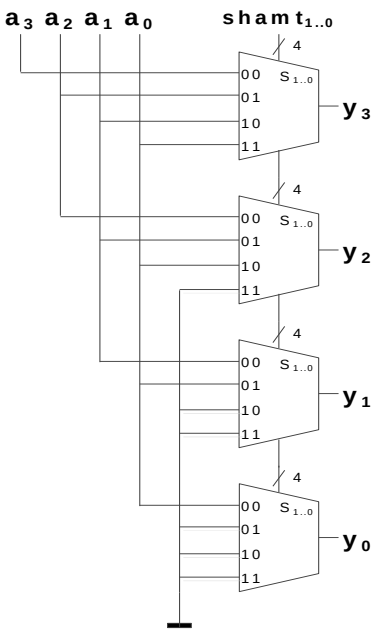
$$= a_{63} \ \& \ s_{63} \mid \sim b_{63} \ \& \ s_{63} \mid a_{63} \ \& \ \sim b_{63}$$

移位操作

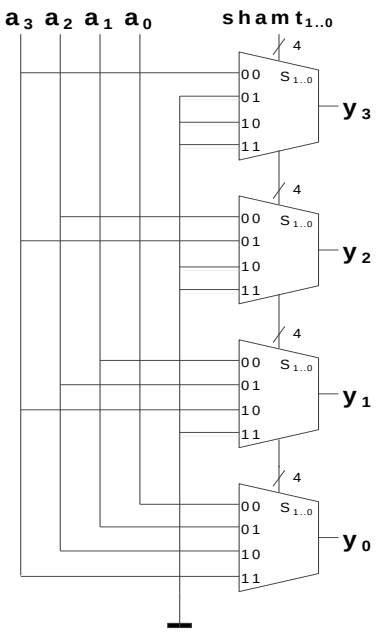
- 移位操作：同时也是乘以/除以 2的幂次的运算
  - 逻辑左移 （低位补0）
  - 逻辑右移 （高位补0）
  - 算数右移 （高位补符号位）
  - 循环右移 （高位补右侧挤掉的数据）
- 例：将32位数 0xABCD1234 移8位
  - 逻辑左移 0xCD123400
  - 逻辑右移 0x00ABCD12
  - 算数右移 0xFFABCD12
  - 循环右移 0x34ABCD12

移位操作（二）

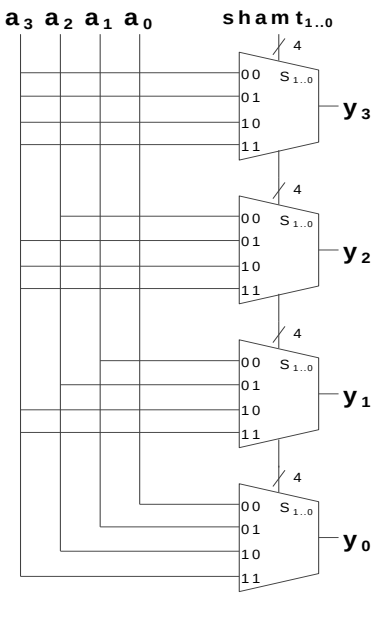
- 对于移N位数的移位操作，使用N选1来实现
  - 根据要移动的位数，从N个输入中选一个
  - 每个输入将输入移动特定位数，不需要延迟和逻辑
- 每种移位结果再根据移位操作类型选择



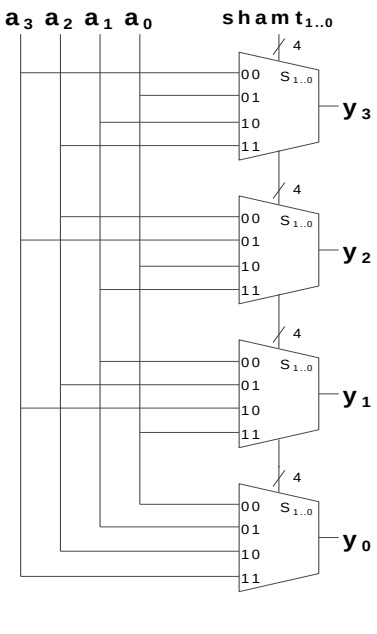
逻辑左移



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

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

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

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

$[X*Y]_{补}= [X*(-y_{31}*2^{31}+y_{30}*2^{30}+\cdots\cdots+y_1*2^1+y_0*2^0)]_{补}$

$=[-X*y_{31}*2^{31}+X*y_{30}*2^{30}+\cdots\cdots+X*y_1*2^1+X*y_0*2^0]_{补}$

$=[-X*y_{31}*2^{31}]_{补}+[X*y_{30}*2^{30}]_{补}+\cdots\cdots+[X*y_1*2^1]_{补}+[X*y_0*2^0]_{补}$

$=[X]_{补}*(-y_{31}*2^{31})+[X]_{补}*(y_{30}*2^{30})+\cdots\cdots+[X]_{补}*(y_1*2^1)+[X]_{补}*(y_0*2^0)$???

$=[X]_{补}*(-y_{31}*2^{31}+y_{30}*2^{30}+\cdots\cdots+y_1*2^1+y_0*2^0)$

- 在推导中, $[X]_{补}$符号位扩充到64位
- 若$X= -x_{31}x_{30}\cdots\cdots x_0$, 则$[X*2^k]_{补}$为对$00\ldots0x_{31}x_{30}\ldots x_0\,0_{k-1}\ldots0_10_0$（一共64位）按位取反后加1, 结果为:

$11\ldots1\ \hat{}x_{31}\hat{}x_{30}\ldots\hat{}x_0\,1_{k-1}\ldots1_11_0+1$

$=11\ldots1\ \hat{}x_{31}\hat{}x_{30}\ldots\hat{}x_0\,0_{k-1}\ldots0_10_0+10_{k-1}\ldots0_10_0=(11\ldots1\ \hat{}x_{31}\hat{}x_{30}\ldots\hat{}x_0+1)0_{k-1}\ldots0_10_0$

这正是$[X]_{补}*2^k$的结果.

补码乘法算法

- $[X*Y]_{补}=[X]_{补}*(-y_{31}*2^{31}+y_{30}*2^{30}+\ldots\ldots+y_1*2^1+y_0*2^0)$
- 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$(-y_{31}*2^{31}+y_{30}*2^{30}+\cdots\cdots+y_1*2^1+y_0*2^0)$进行变换

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

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

- 每一项都一样, 每次看两位

```
1011*1011 (-5*-5)

    1011
   *1011
---------
-11111011
+0000000
+111011
-11011
---------
+00000101
+0000000
+111011
+00101
---------
 00011001(25)
```

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

```
1011*0101 (-5*5)

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

Booth二位一乘算法

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

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

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

- 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	$+[X]_{补}$
0	1	0	$+[X]_{补}$
0	1	1	$+2[X]_{补}$
1	0	0	$-2[X]_{补}$
1	0	1	$-[X]_{补}$
1	1	0	$-[X]_{补}$
1	1	1	0

```
1011*0101 (-5*5)

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

Booth两位乘

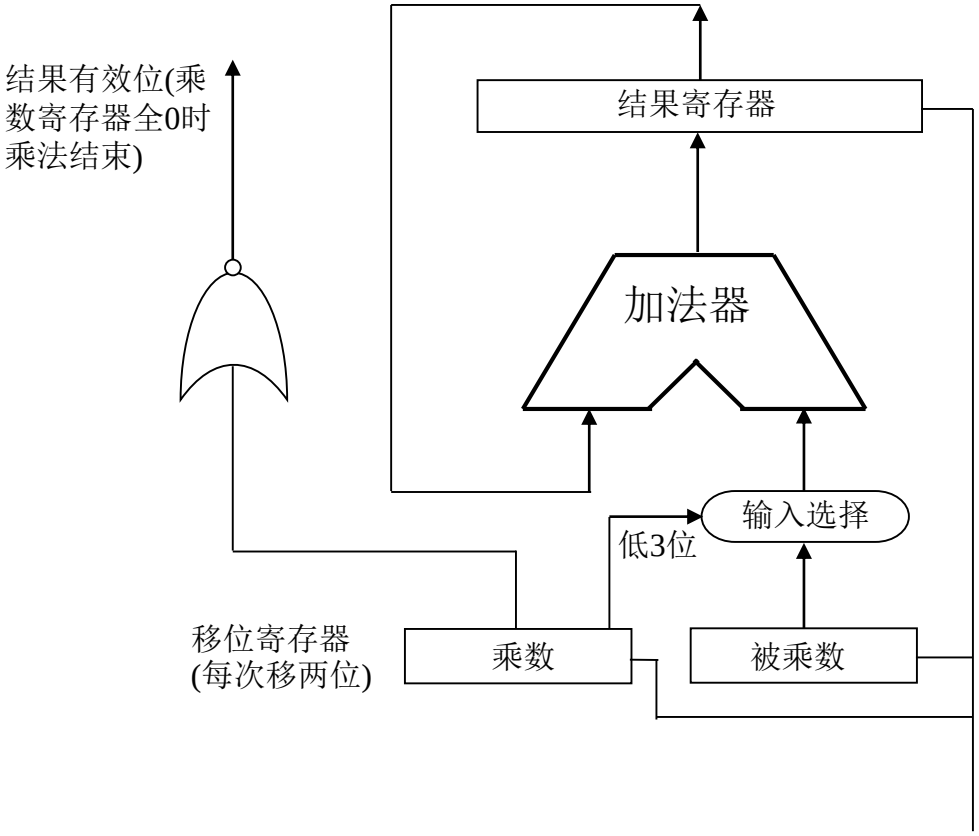
```
1010×1001(-6×-7)
          1010
   ×      1001
-----------------
      +11111010(010)
      -11010  (100)
-----------------
      +11111010
      +00110
-----------------
      100101010(42)
```

```
1010×0101 (-6×5)
          1010
   ×      0101
-----------------
      +11111010(010)
      +111010(010)
-----------------
      +11111010
      +111010
-----------------
      111100010(-30)
```

- 循环次数降低一倍
- 每次循环算法一样
- $[X]_{补}$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

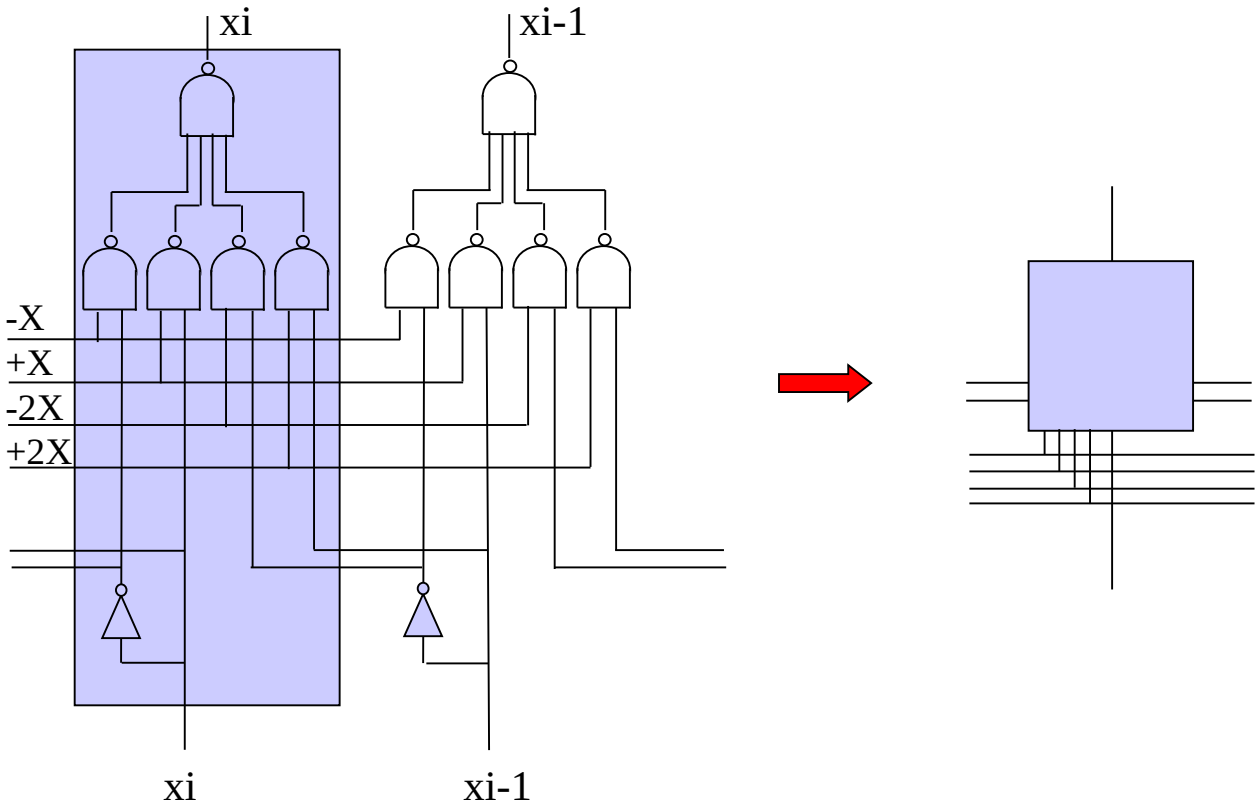
- **以二位一乘为例，32位定点乘法需要把16个数相加**
- **可以用一个加法器加15次，需要15个时钟周期**



Booth二位乘的输入选择逻辑

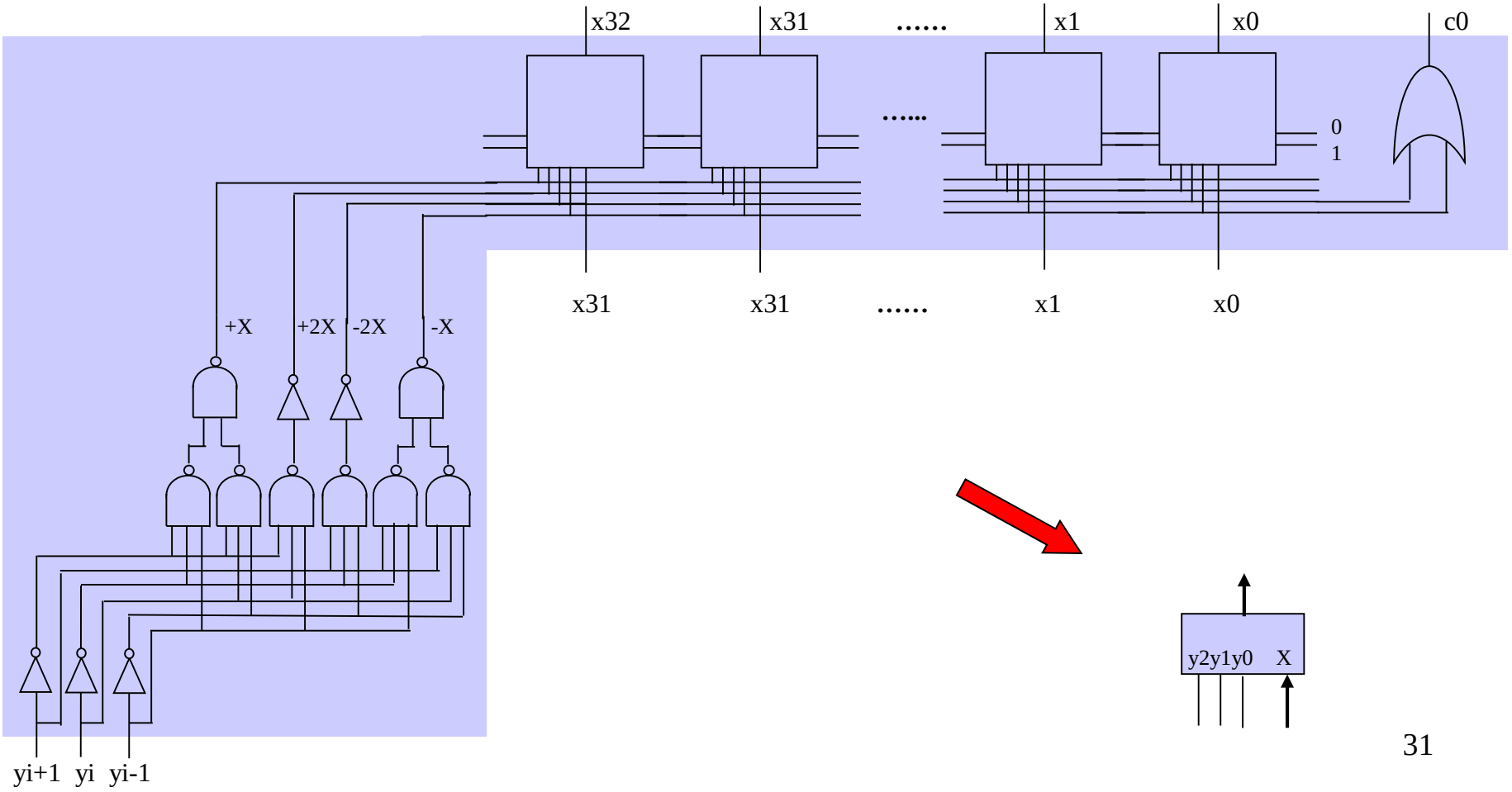
- 其中一位

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



Booth二位乘的输入选择逻辑

- 一组所有位

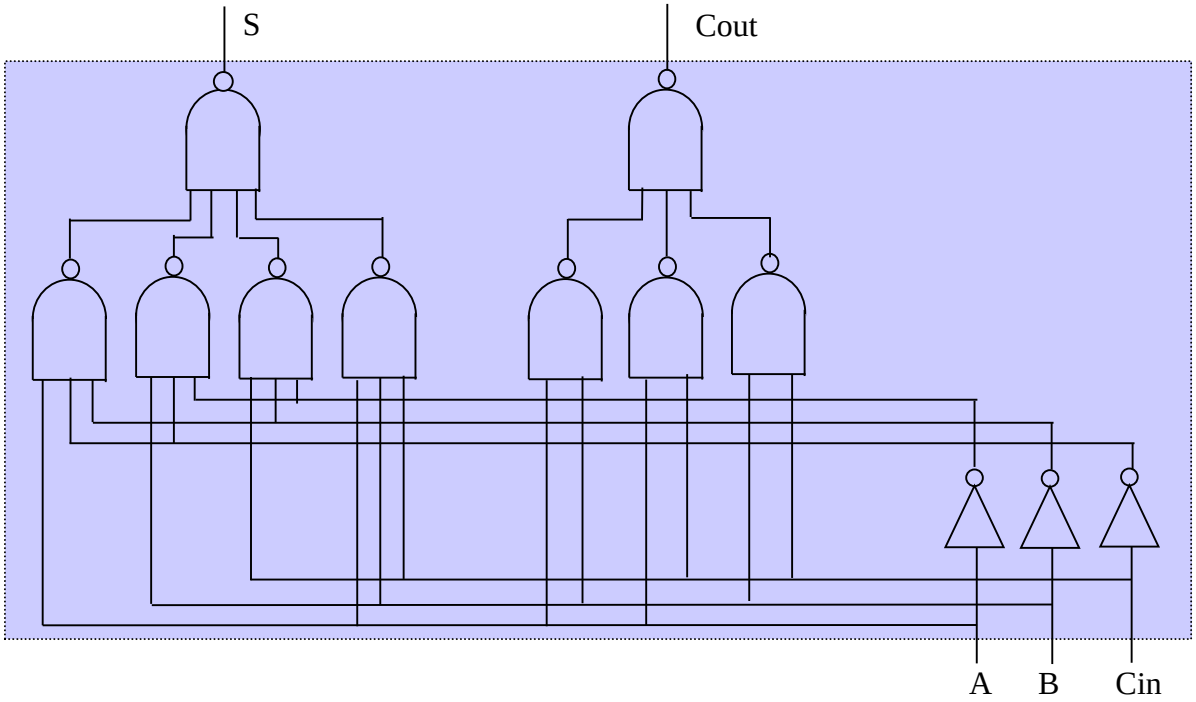


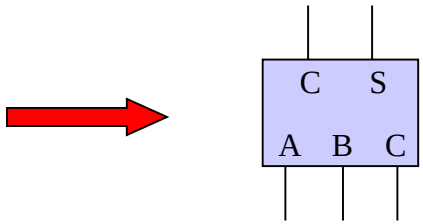
Wallace加法树

- 串行把16个数相加, 需要15次加法时间
- 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

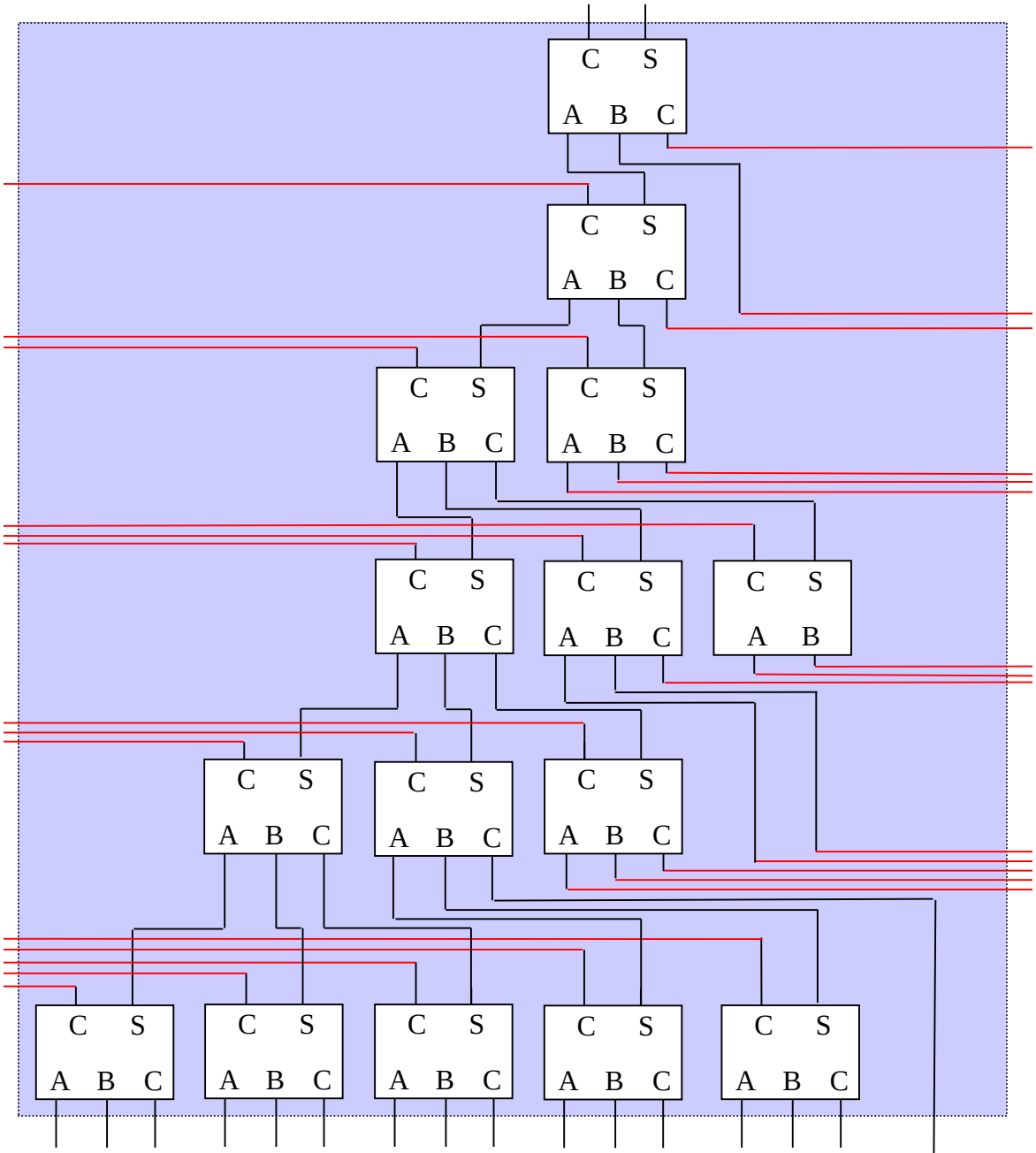


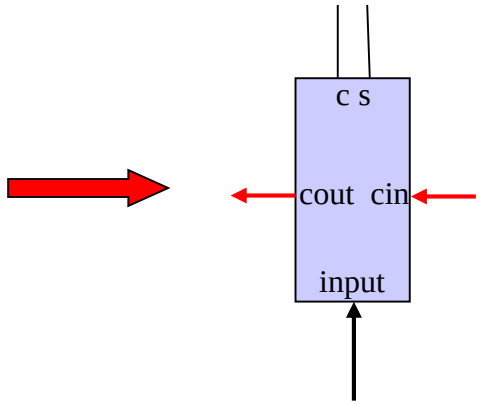


全加器把三个加数变成两个加数

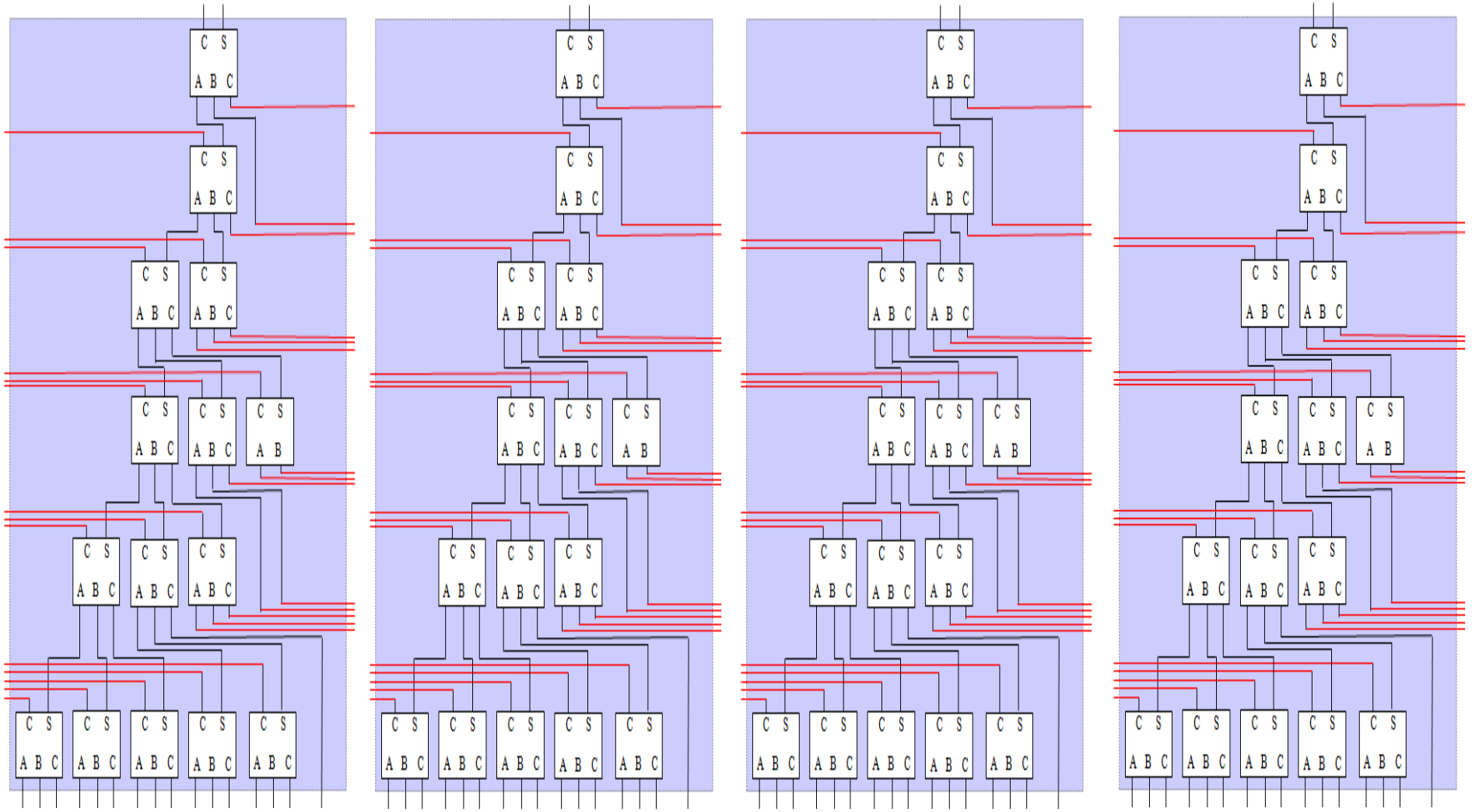
```
x31 x31 x31 x31 x31 x31 x30 x29        .....        x5 x4 x3 x2 x1 x0
x31 x31 x31 x31 x30 x29 x28 x27        .....        x3 x2 x1 x0
x31 x31 x30 x29 x28 x27 x26 x25        .....        x1 x0
-------------------------------------------------------------------
s36 s35 s34 s33 s32 s31 s30 s29     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

16个数相加的1位加法树

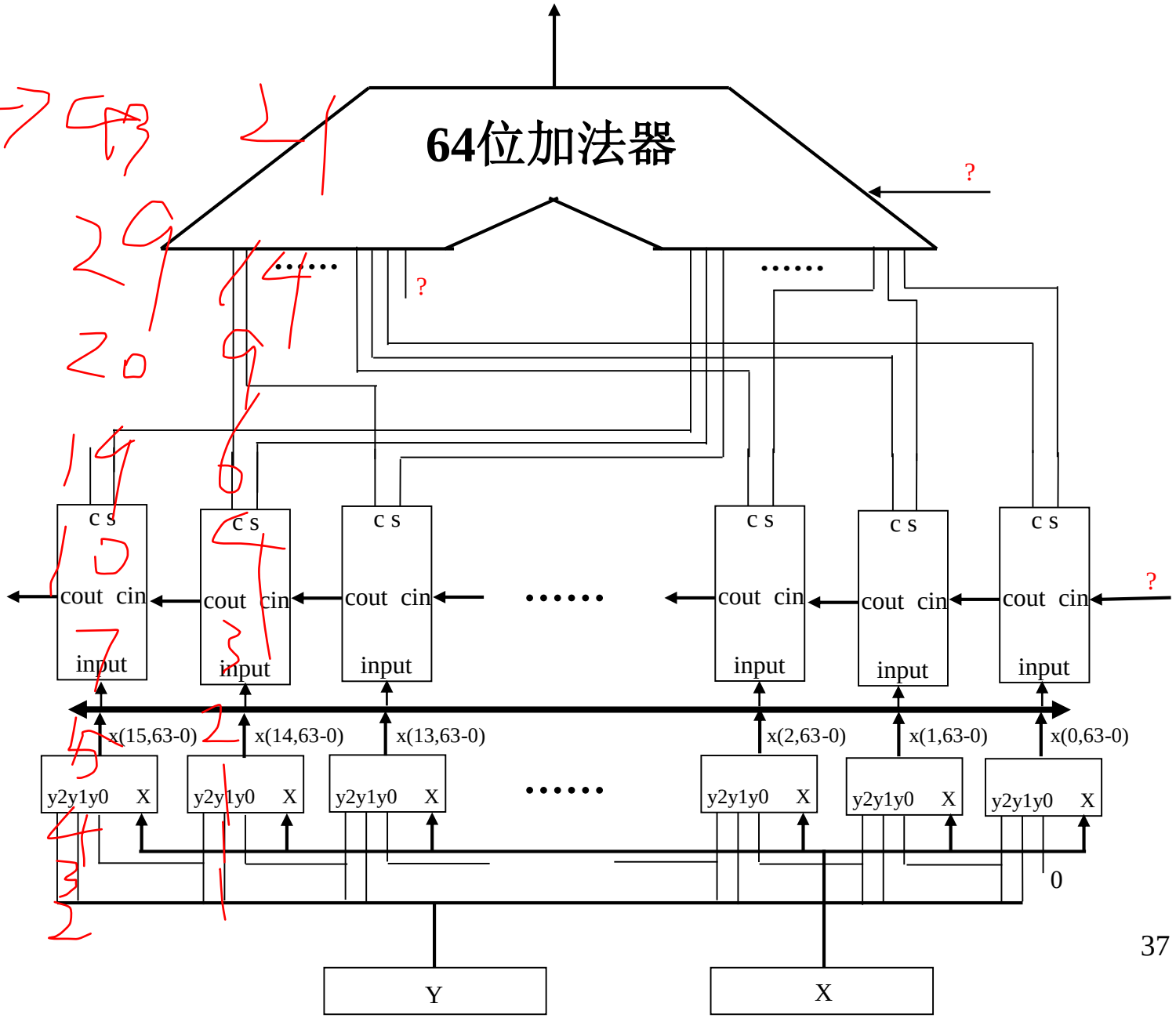




16个数相加的四位华莱士树（左右进位相连）

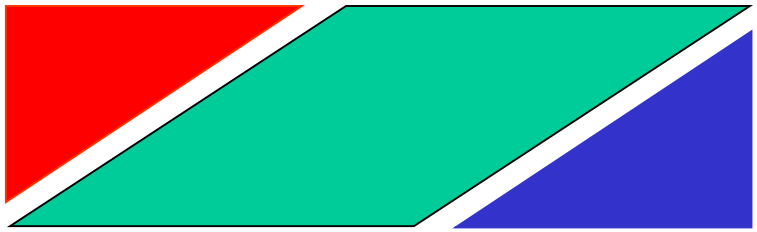


32*32补码乘法器框图



说明

- 32位的$[X]_{补}$, $[-X]_{补}$, $[2X]_{补}$,$[-2X]_{补}$ 如何扩展到64位
  - $[X]_{补}$: 左边符号位扩充, 右边补0
  - $[-X]_{补}$: 对$[X]_{补}$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$-2[X]_{补}$, $-[X]_{补}$引起
  - 加法树中的全加器个数至少是（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