回顾第11次课

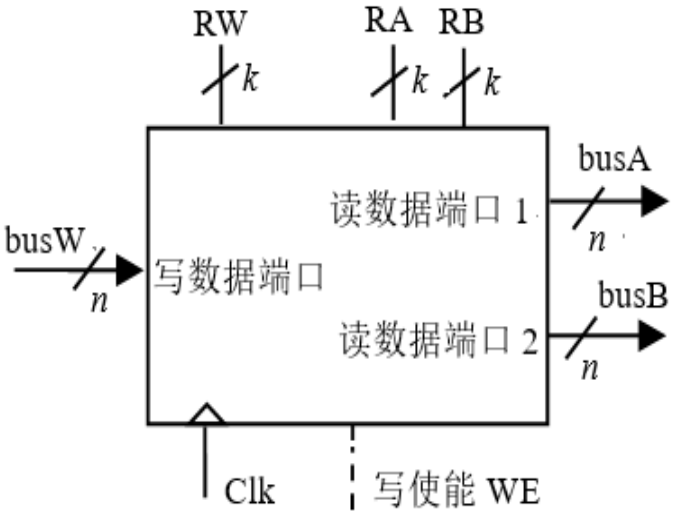
- 未用状态分析(挂起/无法自启动)
- 定时分析(clk-Q时间、时钟周期、setup时间、hold时间)

因此，得到时序约束关系：

(1) $t_{clk} > t_{ffpd(max)} + t_{comb(max)} + t_{setup}$

时钟周期不能小于这个值，但也不需要大过很多

- 典型时序逻辑部件：计数器、寄存器/通用寄存器组、移位寄存器



k：k位无符号二进制数，其真值对应一个寄存器编号，且表示此寄存器堆最多只能有2^k个寄存器

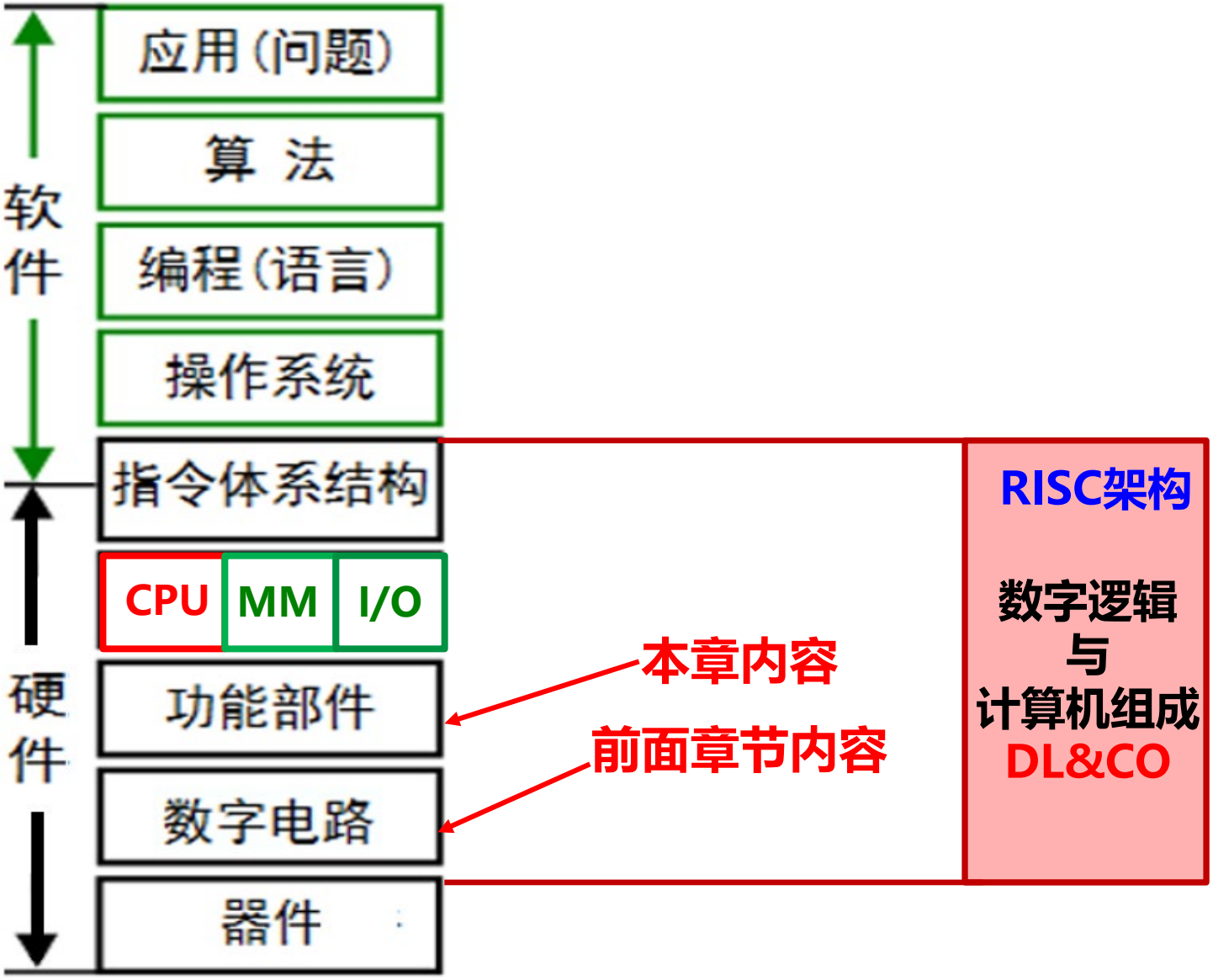
n：代表每个寄存器内存放n位二进制机器数（可以是无符号数、补码、浮点数等等）

两个读口一个写口是最常用配置

第6章 运算方法和运算部件

第一讲 基本运算部件
第二讲 定点数运算
第三讲 浮点数运算

回顾：计算机系统层次结构



第一讲：基本运算部件

主　要　内　容

- 高级语言程序中涉及的运算（以C语言为例）
  - 整数算术运算、浮点数算术运算
  - 按位、逻辑、移位、位扩展和位截断
- 串行进位加法器
- 并行进位加法器
  - 全先行进位加法器
  - 两级/多级先行进位加法器
- 带标志加法器
- 算术逻辑部件（ALU）

高级语言中的运算

- 高级语言程序中涉及的数据类型和运算

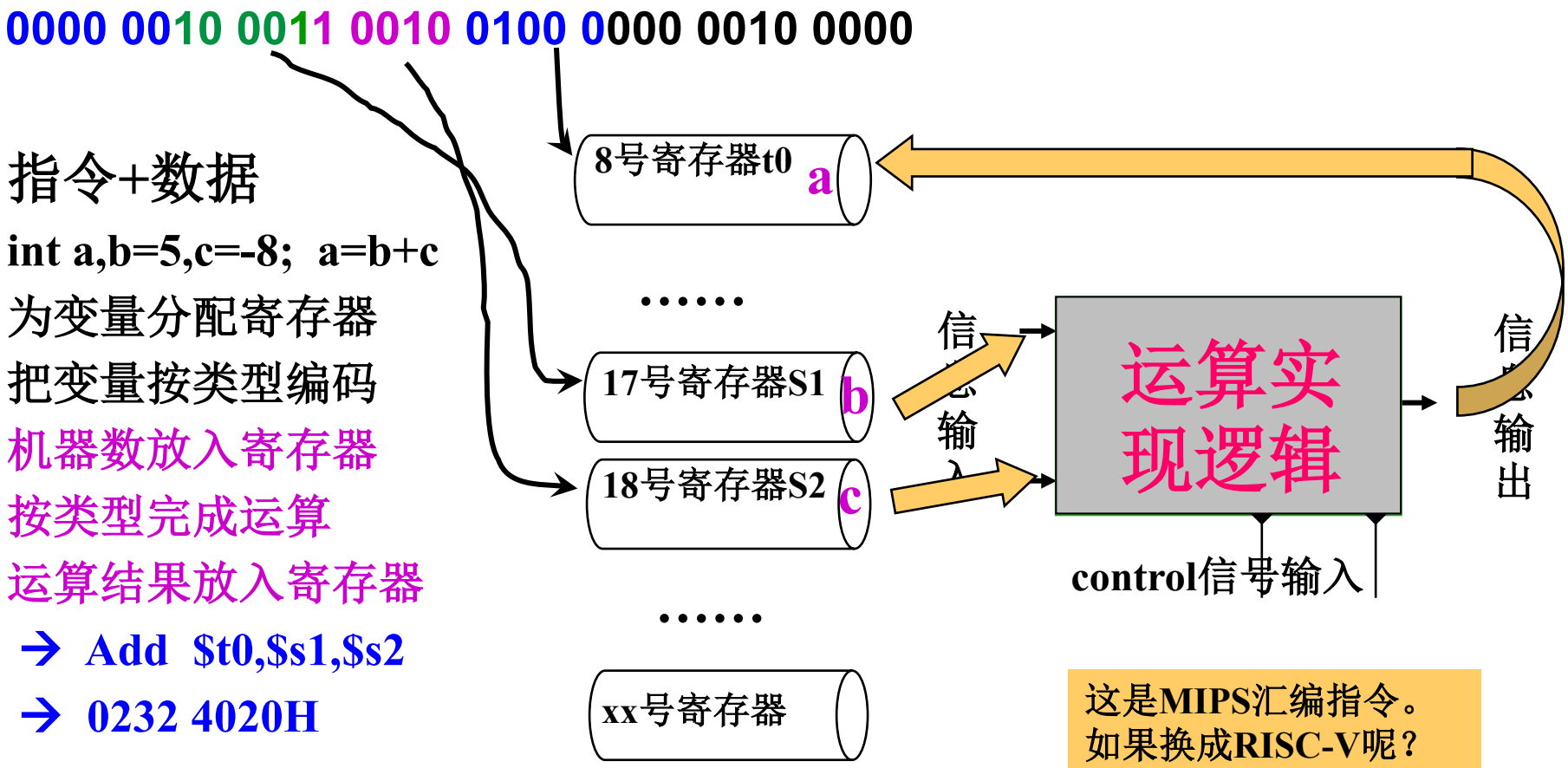
  (以C语言为例)

  - 无符号数，带符号整数， 浮点数，位串，字符

  - 算术运算
  - 按位、逻辑、移位、位扩展和位截断、比较

如何实现高级语言源程序中的运算？

将各类表达式转换成指令序列
计算机执行指令来完成运算



数据的运算

- **指令集中涉及的运算（如RISC-V指令系统提供的运算类指令）**
  - **涉及的定点数运算**
    - **算术运算**
      - **带符号整数：取负 / 符号扩展 / 加 / 减 / 乘 / 除 / 算术移位**
      - **无符号整数：0扩展 / 加 / 减 / 乘 / 除**
    - **逻辑运算**
      - **逻辑操作：与 / 或 / 非 / …**
      - **移位操作：逻辑左移 / 逻辑右移**
  - **涉及的浮点数运算：加、减、乘、除**

完全能够支持高级语言对运算的所有需求

逻辑运算、移位、扩展和截断等指令实现较容易，算术运算指令实现较难！

所有运算都可由ALU或加法器+移位器+多路选择器+控制逻辑实现！

以下介绍基本运算部件：加法器（串行→并行 ）→ 带标志加法器 → ALU

回顾： 半加器和全加器

- **全加器（Full Adder，简称FA）**

输入为加数、被加数和低位进位Cin，输出为和F、进位Cout

$$F=\overline{A}\cdot\overline{B}\cdot Cin+\overline{A}\cdot B\cdot\overline{Cin}+A\cdot\overline{B}\cdot\overline{Cin}+A\cdot B\cdot Cin$$

$$Cout=\overline{A}\cdot B\cdot Cin+A\cdot\overline{B}\cdot Cin+A\cdot B\cdot\overline{Cin}+A\cdot B\cdot Cin$$

真值表

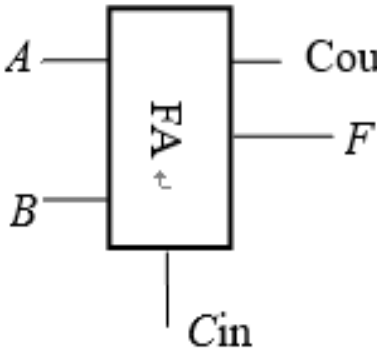
A	B	Cin	F	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

化简后：

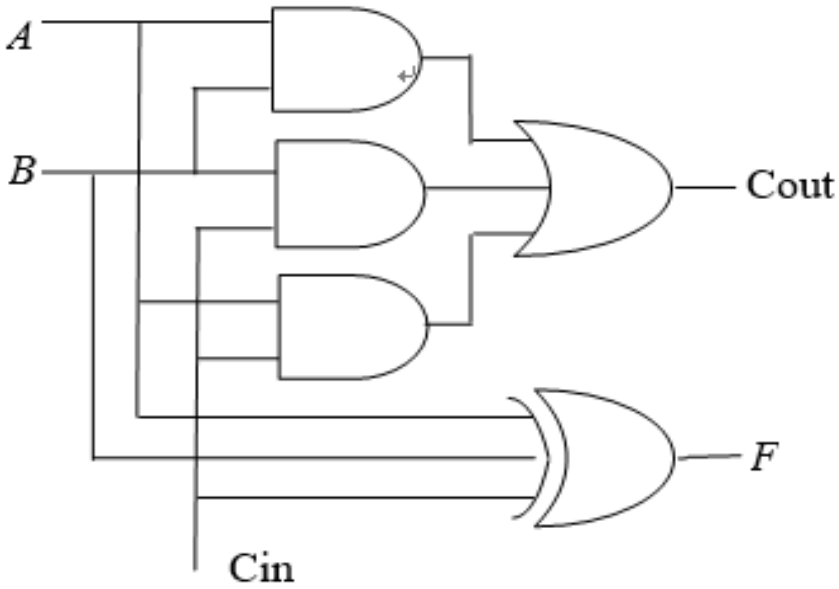
$$F=A\oplus B\oplus Cin$$

$$Cout=A\cdot B+A\cdot Cin+B\cdot Cin$$

逻辑符号



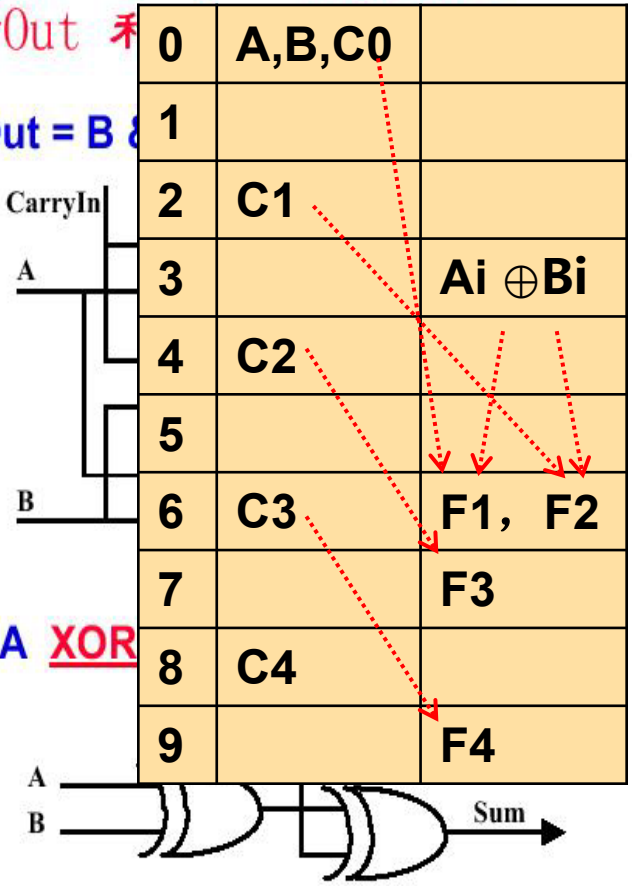
全加器逻辑电路图



串行进位加法器

CarryOut 和

- CarryOut = B & A & B

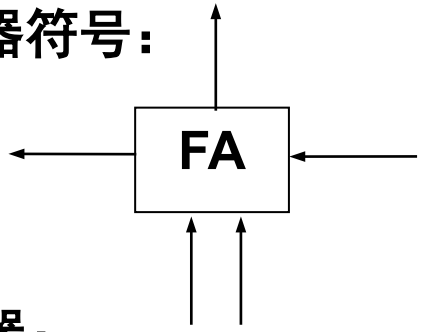


- Sum = A XOR

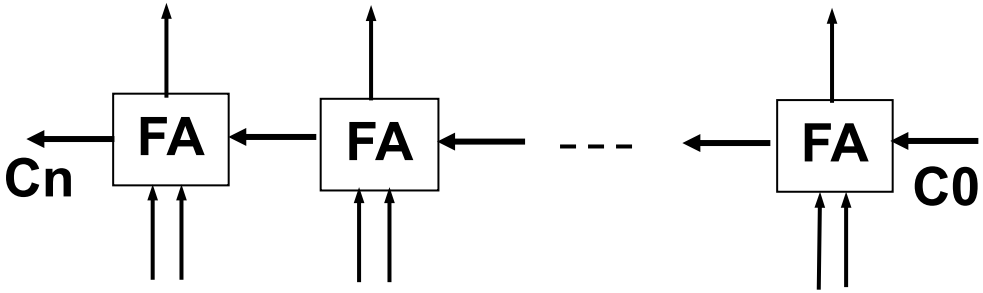
假定与/或门延迟为1，异或门为3，则“和”与“进位”延迟为多少?

Sum延迟为6；Carryout延迟为2。

全加器符号：



n位串行(行波)加法器：



串行加法器的缺点：

进位按串行方式传递，速度慢！

问题：n位串行加法器从C0到Cn的延迟时间为多少？ 2n级门延迟！

最后一位和数的延迟时间为多少?

2n+1级门延迟！

(n=1、2的话还是需要6级)

并行进位加法器（CLA）

- 为什么用先行进位方式？

  串行进位加法器采用串行逐级传递进位，电路延迟与位数成正比关系。

  因此，现代计算机采用一种先行进位(Carry look ahead)方式。

- 如何产生先行进位？

$$Cout=A\cdot B+A\cdot Cin+B\cdot Cin$$

定义辅助函数：$G_i=A_iB_i$　　进位生成函数

$P_i=A_i+B_i$　　进位传递函数

通常把实现上述逻辑的电路称为进位生成/传递部件

全加逻辑方程：$F_i=A_i\oplus B_i\oplus C_{i-1}$　$C_i=G_i+P_iC_{i-1}$ (i=1,…n)

设n=4,则：$C_1=G_1+P_1C_0$

$C_2=G_2+P_2C_1=G_2+P_2G_1+P_2P_1C_0$

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

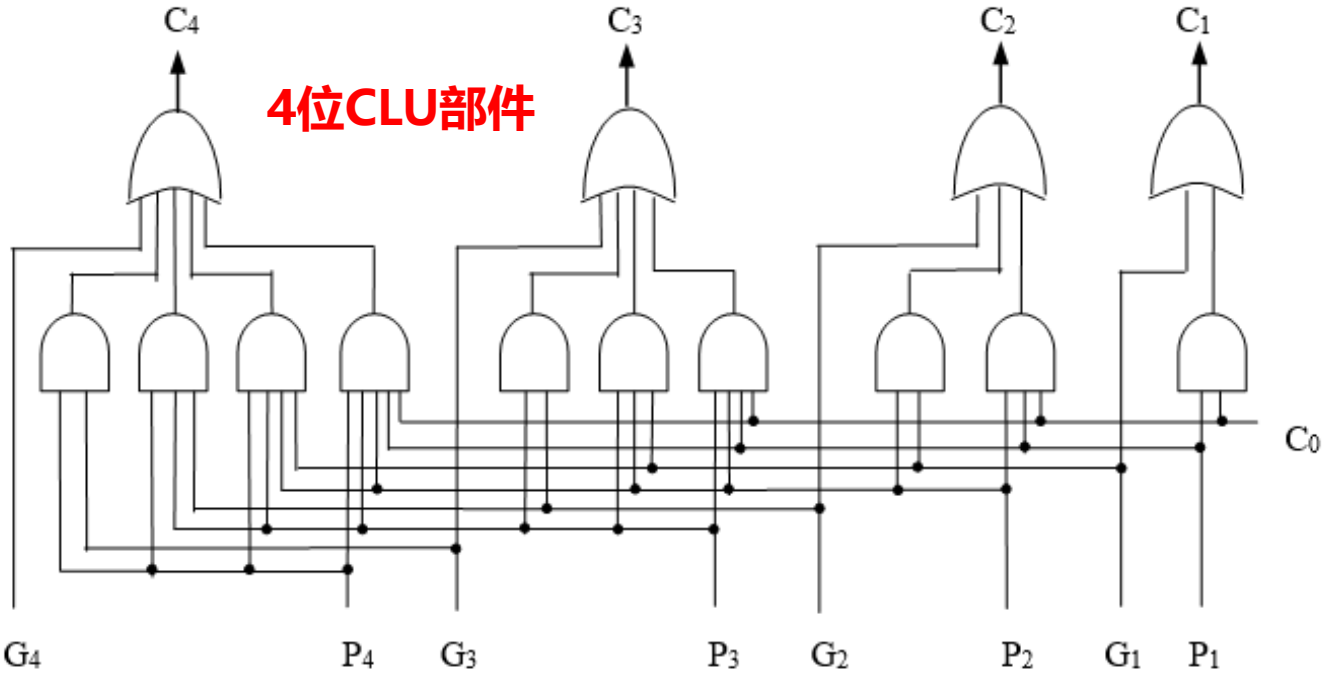
$C_4=G_4+P_4C_3=G_4+P_4G_3+P_4P_3G_2+P_4P_3P_2G_1+P_4P_3P_2P_1C_0$

由上式可知:各进位之间无等待，相互独立并同时产生。

通常把实现上述逻辑的电路称为4位先行进位部件（4位CLU）

CLA加法器

0: X,Y,C0
1: Pi,Gi
3: C1,2,3,4,Xi ⊕Yi
6: F1,2,3,4



$C_1=G_1+P_1C_0$

$C_2=G_2+P_2C_1=G_2+P_2G_1+P_2P_1C_0$

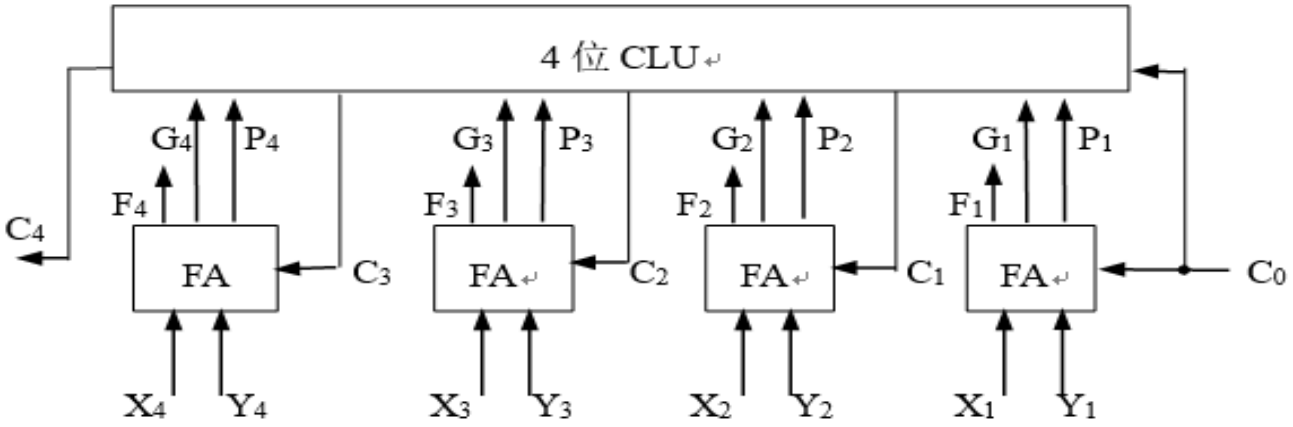
$C_3=G_3+P_3C_2=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1C_0$

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

$G_i=X_iY_i$

$P_i=X_i+Y_i$ （或 $P_i=X_i\oplus Y_i$ ）

$F_i=X_i\oplus Y_i\oplus C_{i-1}$



4位全先行进位加法器CLA

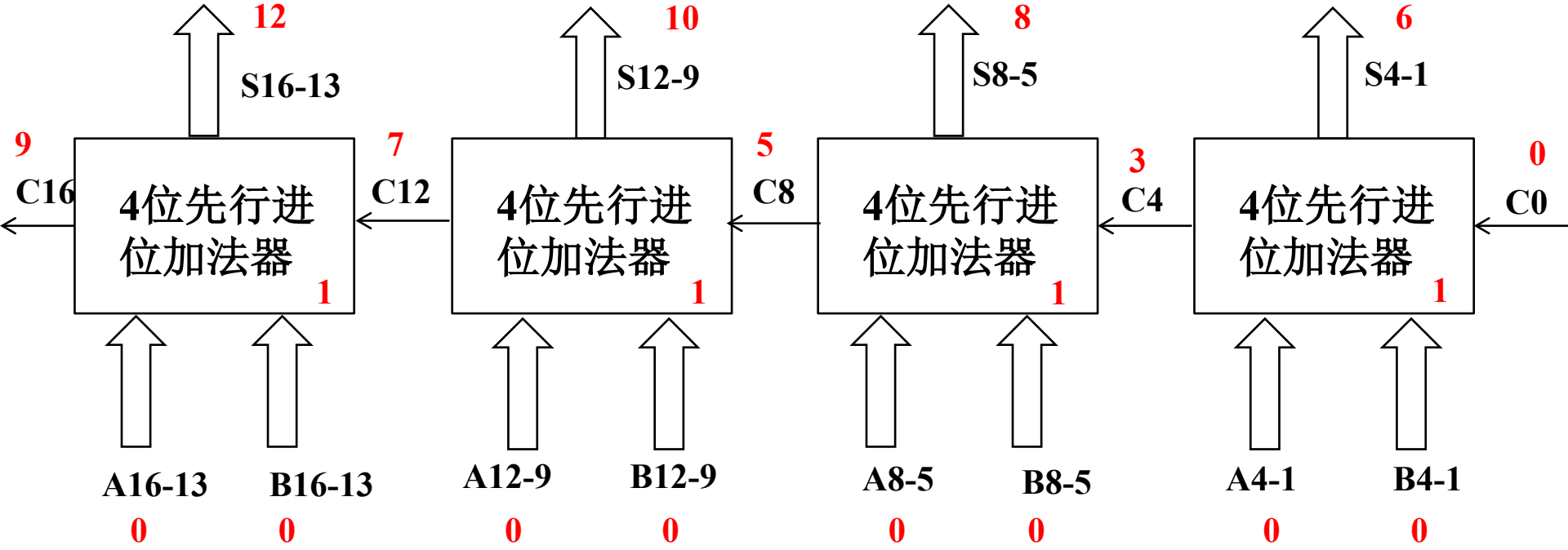
（所有进位独立并同时生成）

局部（单级）先行进位加法器(不要求！)

- **Partial Carry Lookahead Adder**
  - 实现全先行进位加法器的成本太高
  - 位数多了，逻辑方程太长，电路面积大

“组内并行组间串行”

- 折中做法：
  - 连接几个N位先行进位加法器，形成一个大加法器
  - 例如：4个4位构成一个16位



多级先行进位加法器（不要求！）

C4 = Gm1 + Pm1 * C0
C8 = Gm2 + Pm2 * Gm1 + Pm2 * Pm1 *C0
C12 = Gm3 + Pm3 * Gm2 + Pm3 * Pm2 *Gm1 + Pm3* Pm2 * Pm1 * C0
C16 = Gm4 + Pm4 * Gm3 + Pm4 * Pm3*Gm2 + Pm4 * Pm3 * Pm2 * Gm1 + Pm4 * Pm3* Pm2 * Pm1 *C0

“组内并行、组间并行”进位方式

设n=4,则：$C_1=G_0+P_0C_0$

$C_2=G_1+P_1C_1=G_1+P_1G_0+P_1P_0C_0$

$C_3=G_2+P_2C_2=G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0C_0$

$C_4=G_3+P_3C_3=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1G_0+P_3P_2P_1P_0C_0$

G_{m1}　　P_{m1}

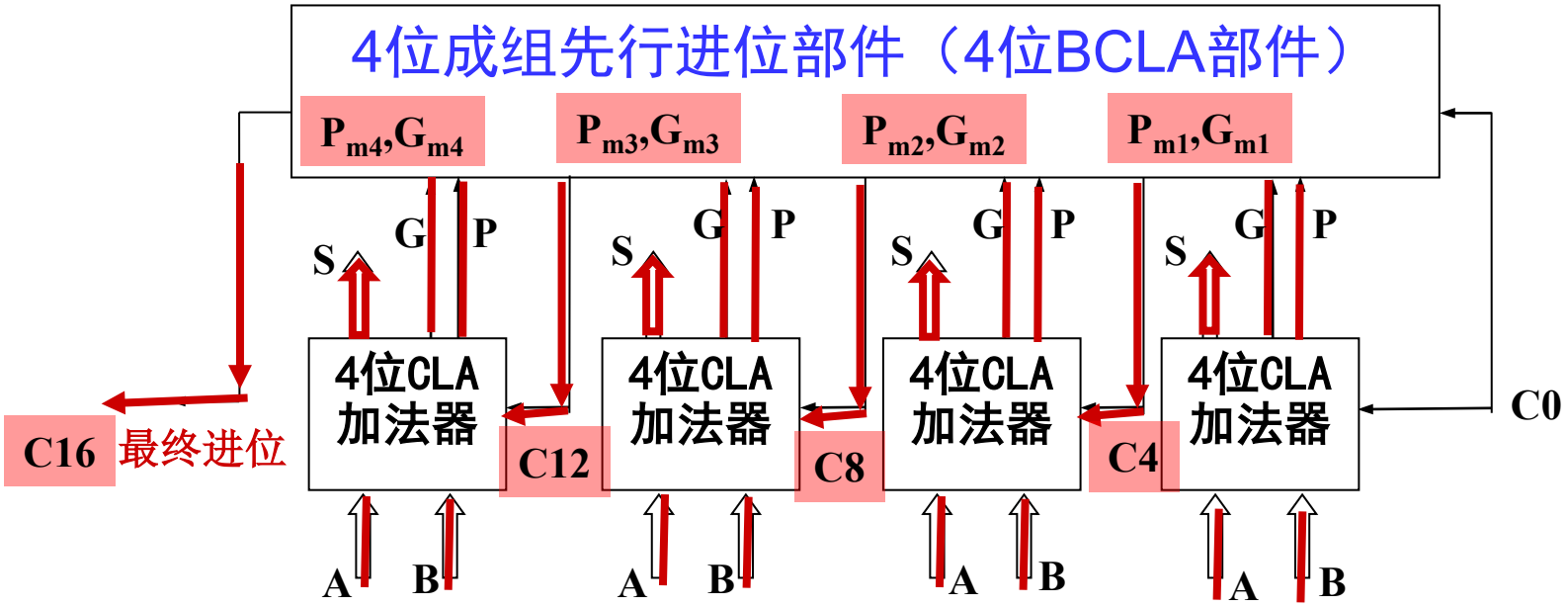
所以$C_4=G_{m1}+P_{m1}C_0$。类似的$C8=G_{m2}+P_{m2}C_4$等。然后与上述展开方法同理，$C_{4,8,12,16}$只与C_0和P_m、G_m有关。实现该逻辑的电路称为4位BCLA部件。

在生成所有的P和G之后，需要2级门延迟可计算出所有的P_{m*}和G_{m*}

然后还需要2级门延迟计算出C4,8,12,16

多级先行进位加法器（不要求！）

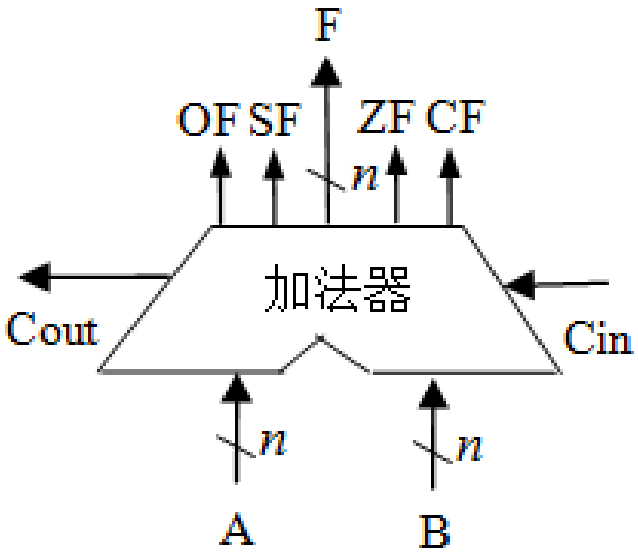
16位两级先行进位加法器



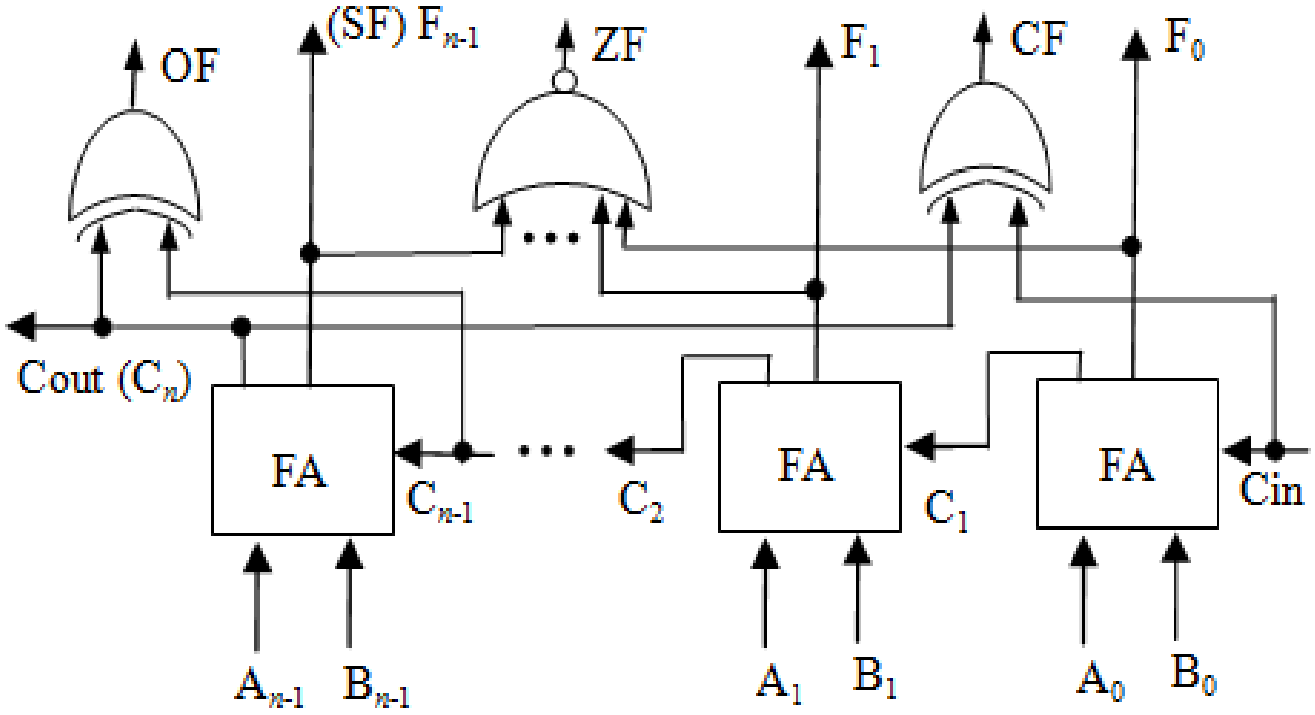
0: A,B,C0
1: Pi,Gi
3: Pmi,Gmi,C1,2,3
5: sum1,2,3,　C4,C8,C12,C16
7: C5,6,7……
10: sum其余位

n位带标志加法器

- n位加法器无法用于两个n位带符号整数（补码）相加，无法判断是否溢出
- 程序中经常需要比较大小，通过（在加法器中）做减法得到的标志信息来判断



带标志加法器符号



（这里是串行进位）

带标志加法器的逻辑电路

溢出标志OF：

$OF=C_n \oplus C_{n-1}$

符号标志SF：

$SF=F_{n-1}$

零标志ZF=1当且仅当F=0；

进位/借位标志CF：

CF=Cout⊕Cin