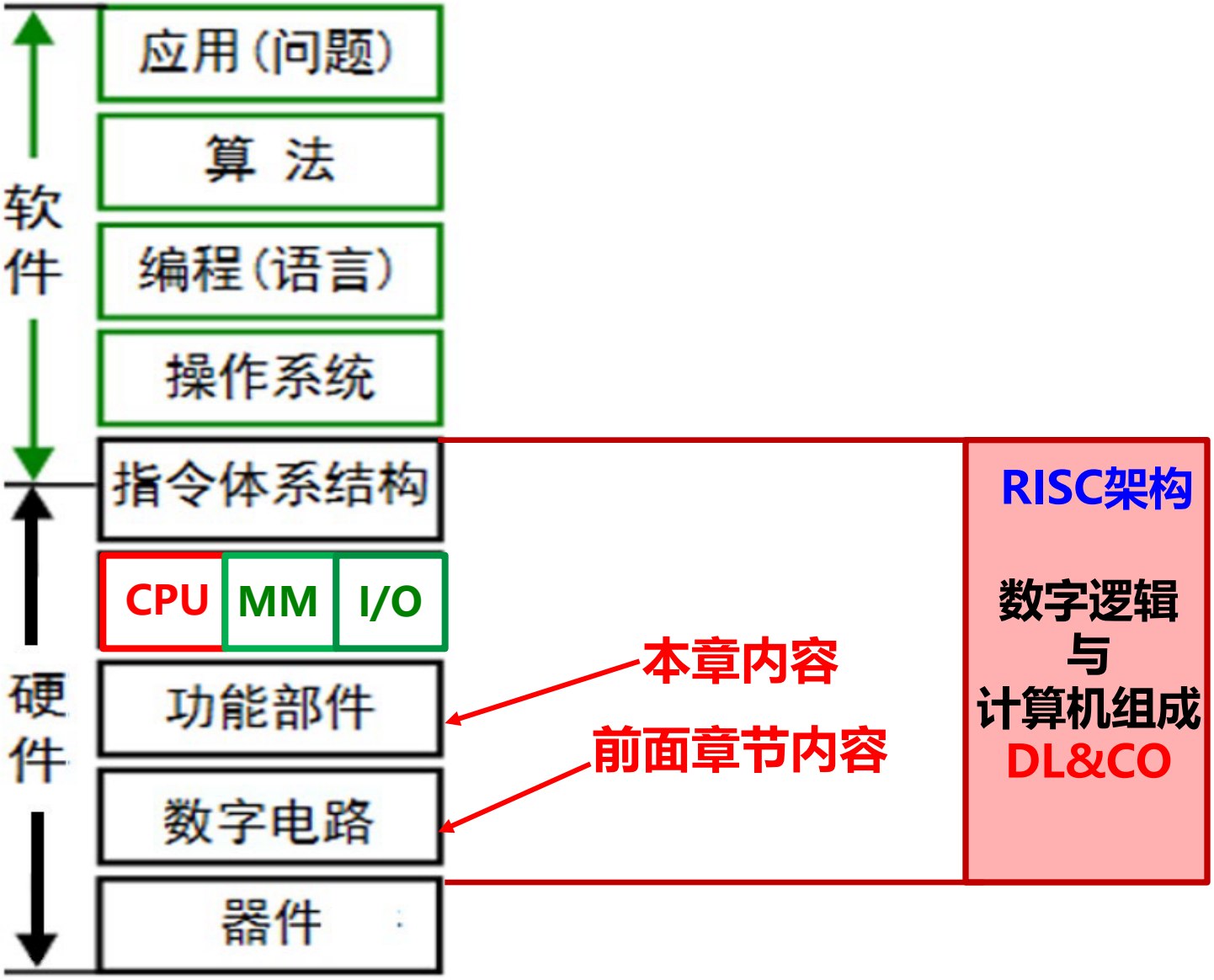
第6章 运算方法和运算部件

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)

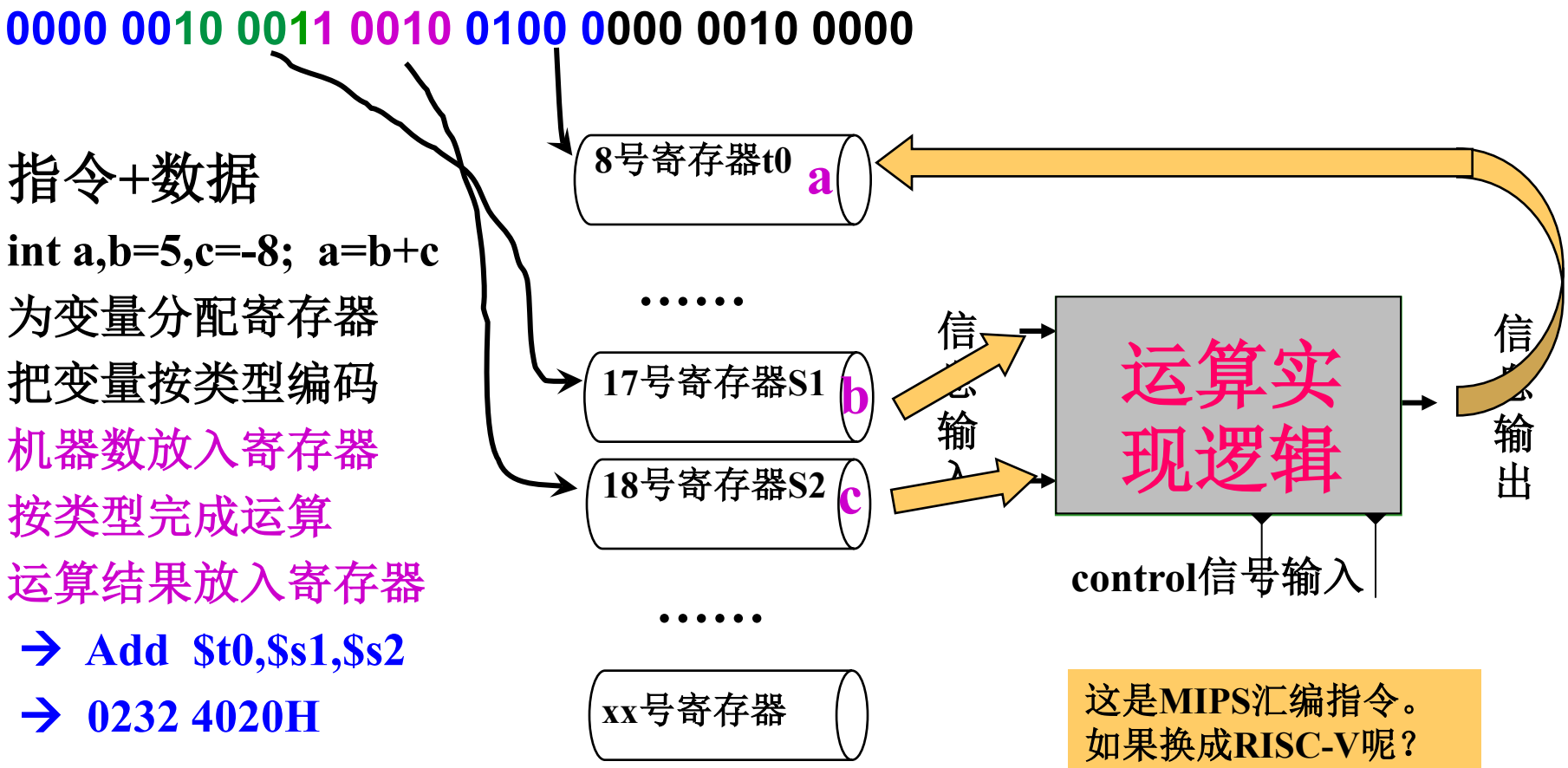
  - 无符号数，带符号整数（补码），浮点数，位串，字符

  - 算术运算 +, −, ×, ÷

  - 按位、逻辑、移位、位扩展和位截断、比较

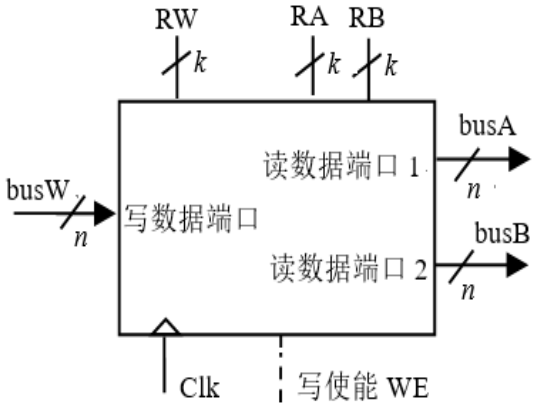
如何实现高级语言源程序中的运算？

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



回顾第12次课

- **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**

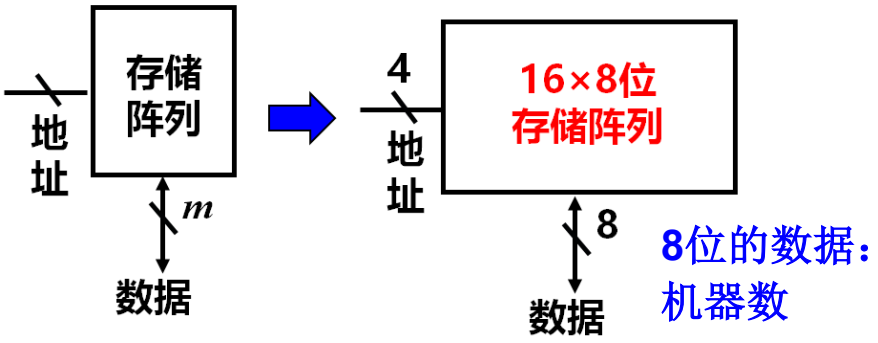


k：k位无符号二进制数，其真值对应一个寄存器编号，且表示此寄存器堆最多只能有2^k个寄存器

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

- **ROM和RAM**
- **存储器**



4位的地址：无符号数

0000
0001
0010
0011
.....
1100
1101
1110
1111

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

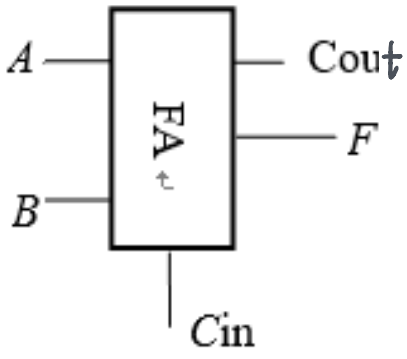
A	B	Cin	F	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

化简后：

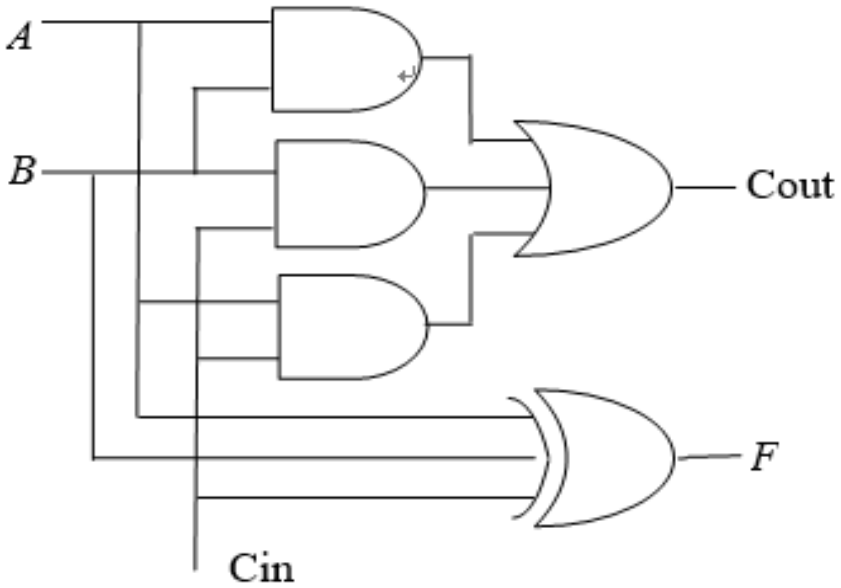
$$F=A\oplus B\oplus Cin$$

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

逻辑符号



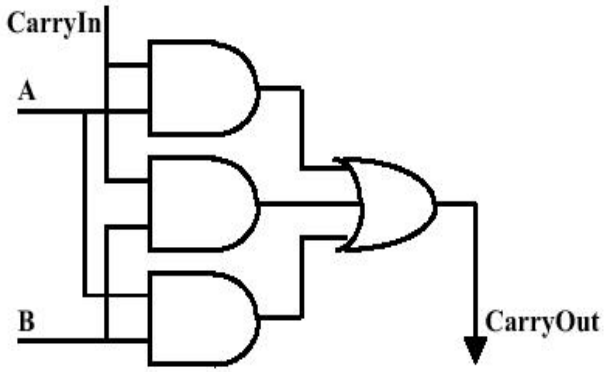
全加器逻辑电路图



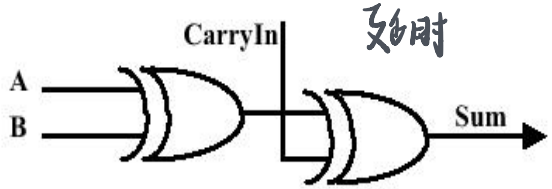
串行进位加法器

CarryOut 和 Sum 的逻辑图

- CarryOut = B & CarryIn | A & CarryIn | A & B



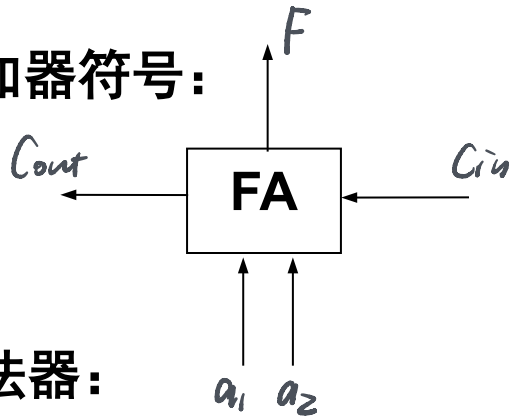
- Sum = A XOR B XOR CarryIn



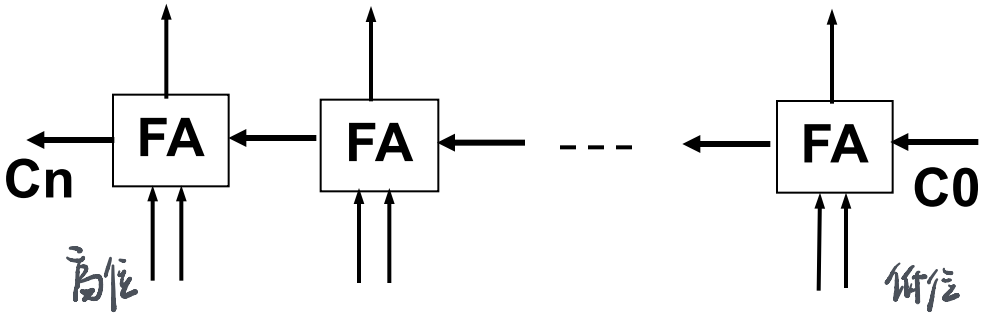
假定与/或门延迟为1，异或门为3，则“和”与“进位”延迟为多少?

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?

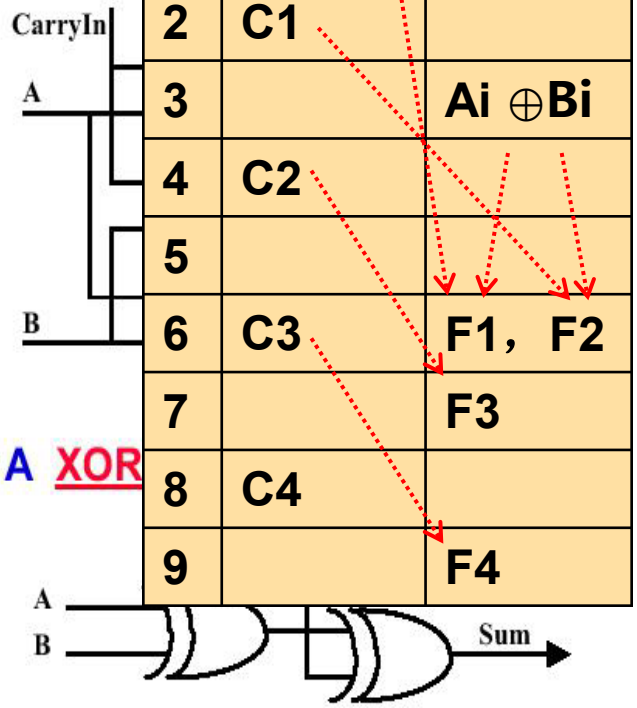
2n+1级门延迟!

(n=1、2的话还是需要6级)

串行进位加法器

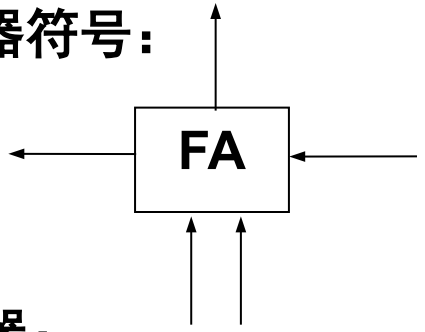
CarryOut 和

- CarryOut = B & A & B

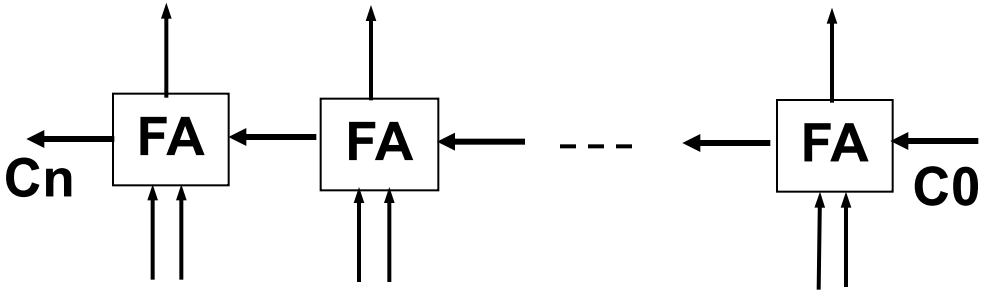


- Sum = A XOR

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！

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。

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？**

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

$$=A\cdot B+(A+B)\cdot Cin$$

$$\triangleq G_i+P_i\cdot Cin$$

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：$C_1=G_1+P_1C_0$

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

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

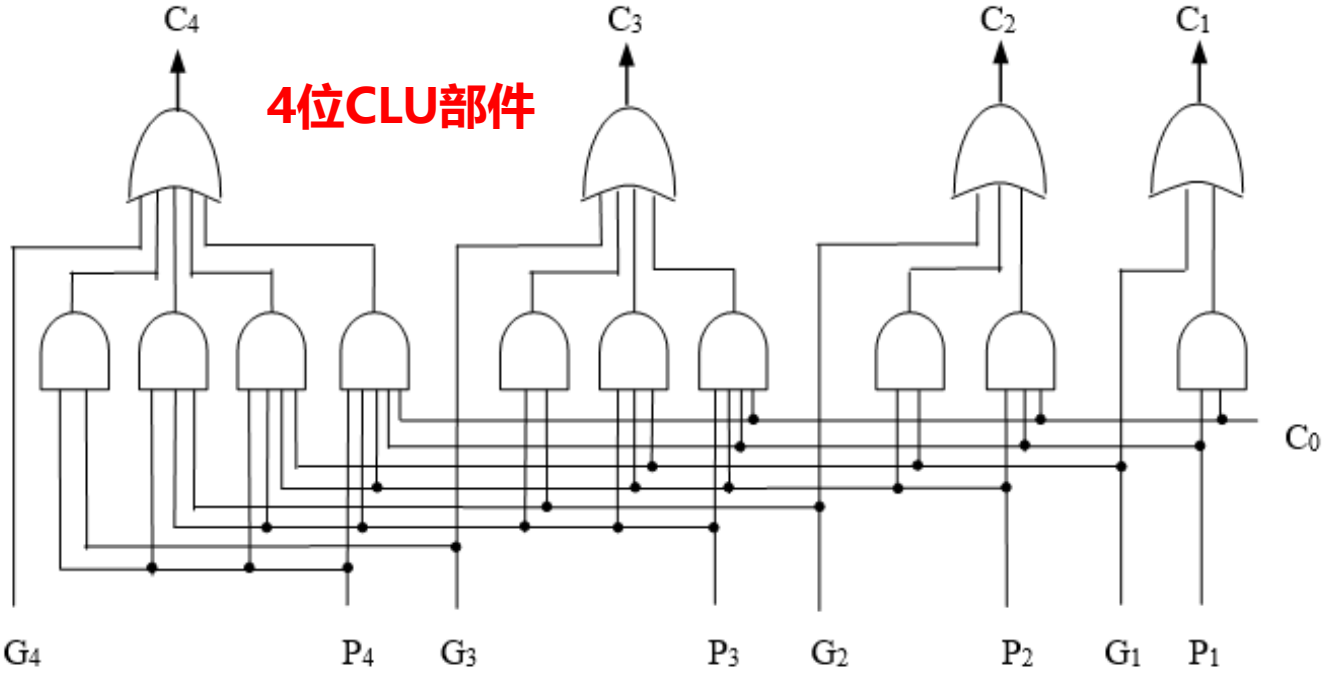
$C_4=G_4+P_4C_3=G_4+P_4G_3+P_4P_3G_2+P_4P_3P_2G_1+P_4P_3P_2P_1C_0$

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



$C_1=G_1+P_1C_0$

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

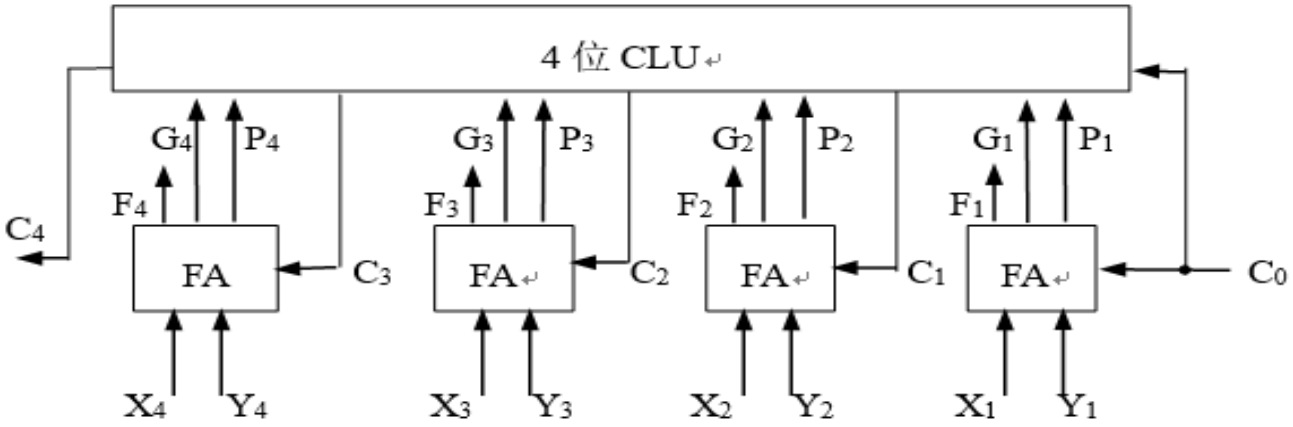
$C_3=G_3+P_3C_2=G_3+P_3G_2+P_3P_2G_1+P_3P_2P_1C_0$

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

$G_i=X_iY_i$

$P_i=X_i+Y_i$ （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

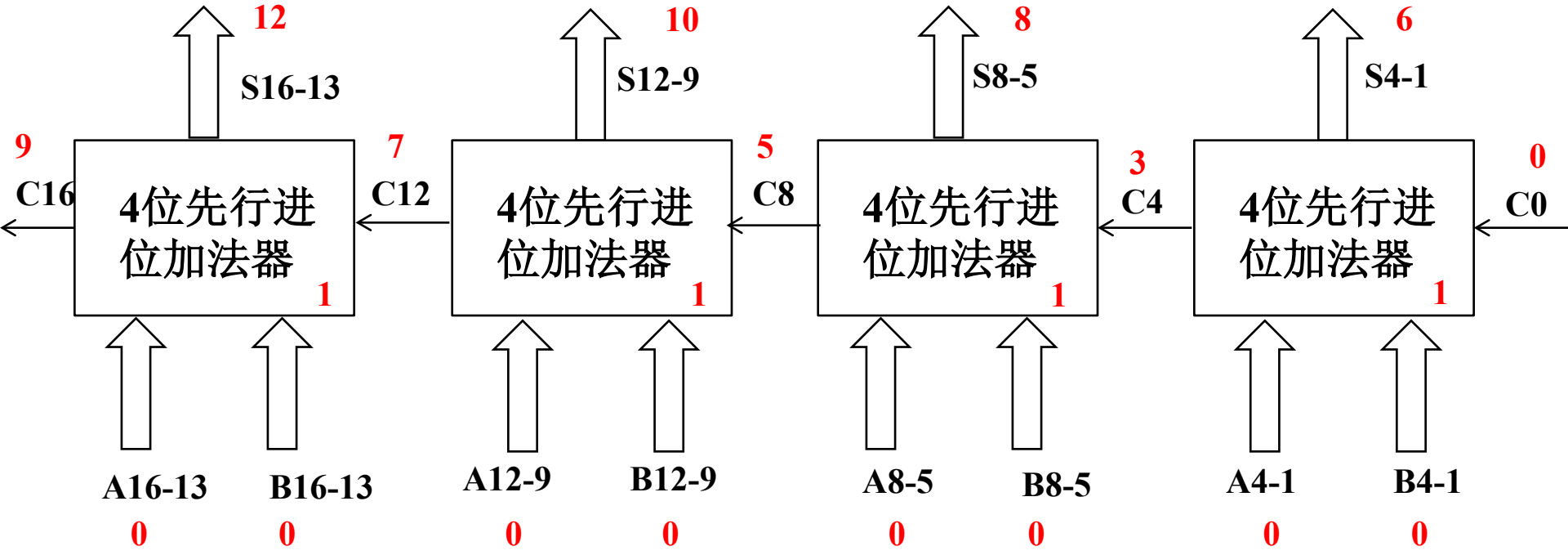
（所有进位独立并同时生成）

(4) 局部（单级）先行进位加法器(不要求！) 折中派

- **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}

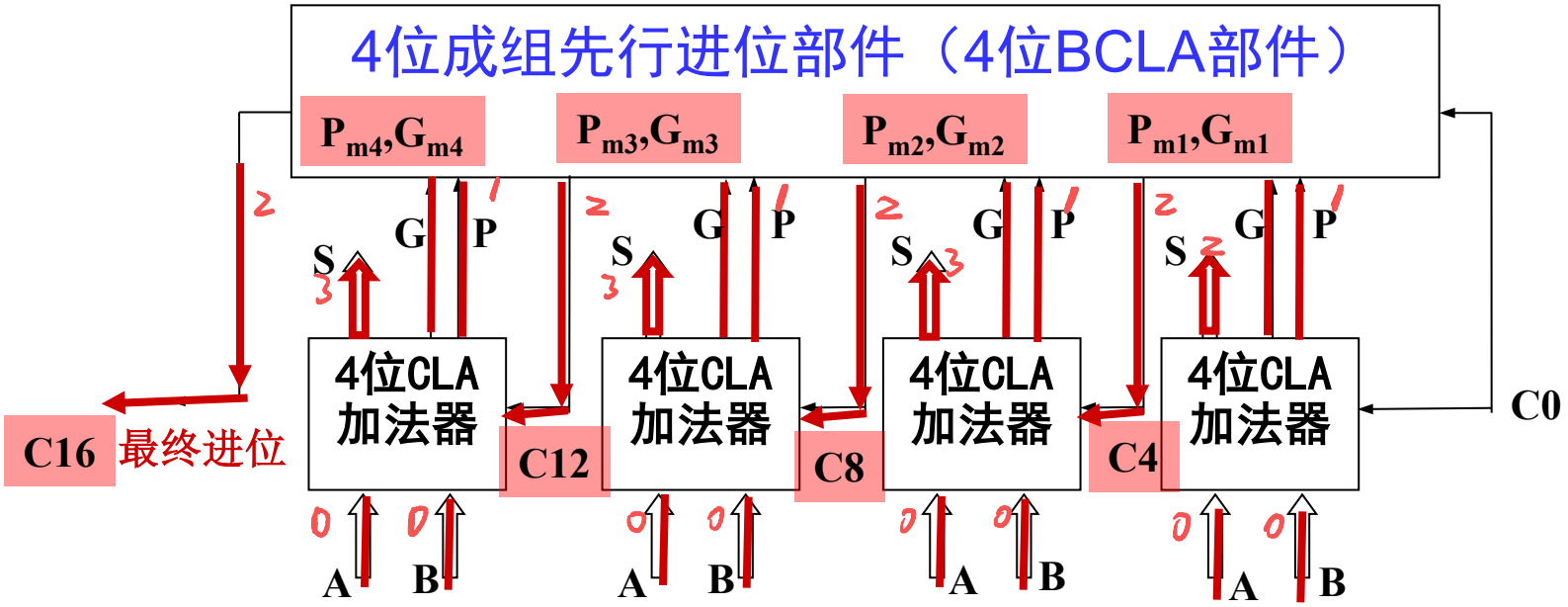
所以$C_4=G_{m1}+P_{m1}C_0$。类似的$C8=G_{m2}+P_{m2}C_4$等。然后与上述展开方法同理，$C_{4,8,12,16}$只与C_0和P_m、G_m有关。实现该逻辑的电路称为4位BCLA部件。

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！）

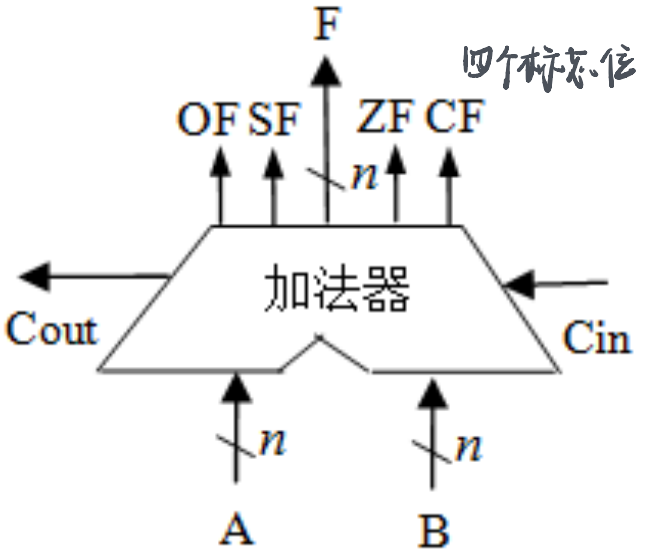
16位两级先行进位加法器



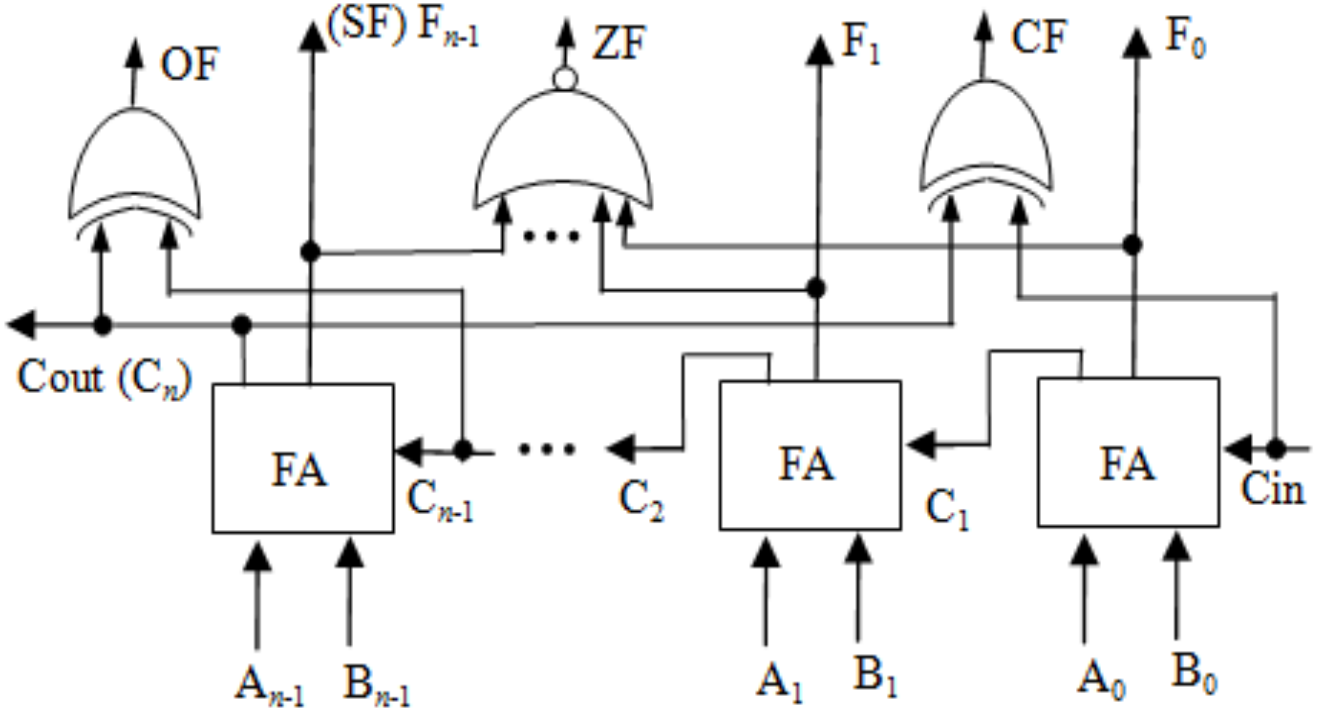
0: A,B,C0
1: Pi,Gi
3: Pmi,Gmi,C1,2,3
5: sum1,2,3,　C4,C8,C12,C16
7: C5,6,7……
10: sum其余位

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：

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

Over flow Flag

符号标志SF：

$SF=F_{n-1}$

Sign Flag

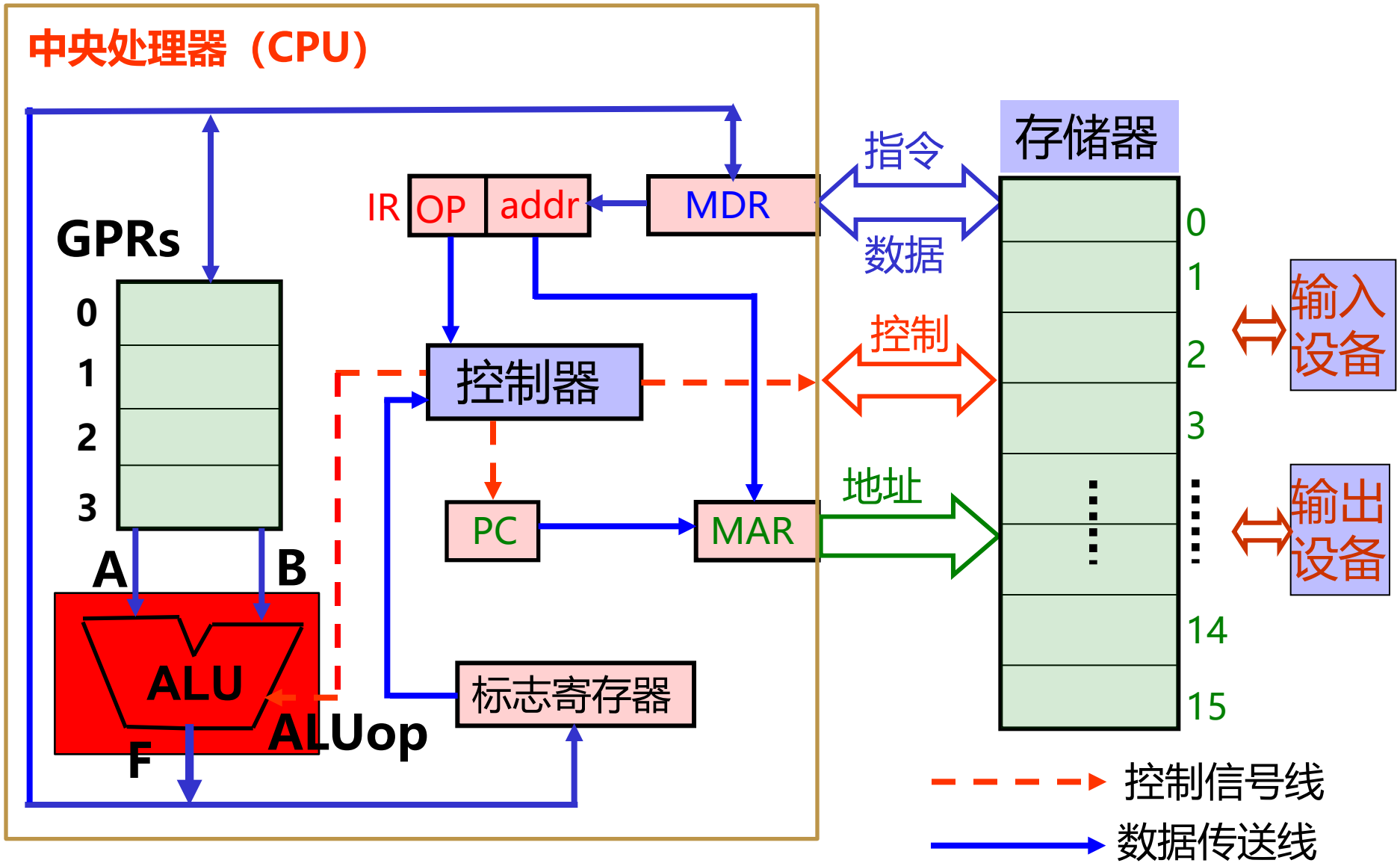
零标志ZF=1当且仅当F=0；

进位/借位标志CF：

$CF=Cout \oplus Cin$

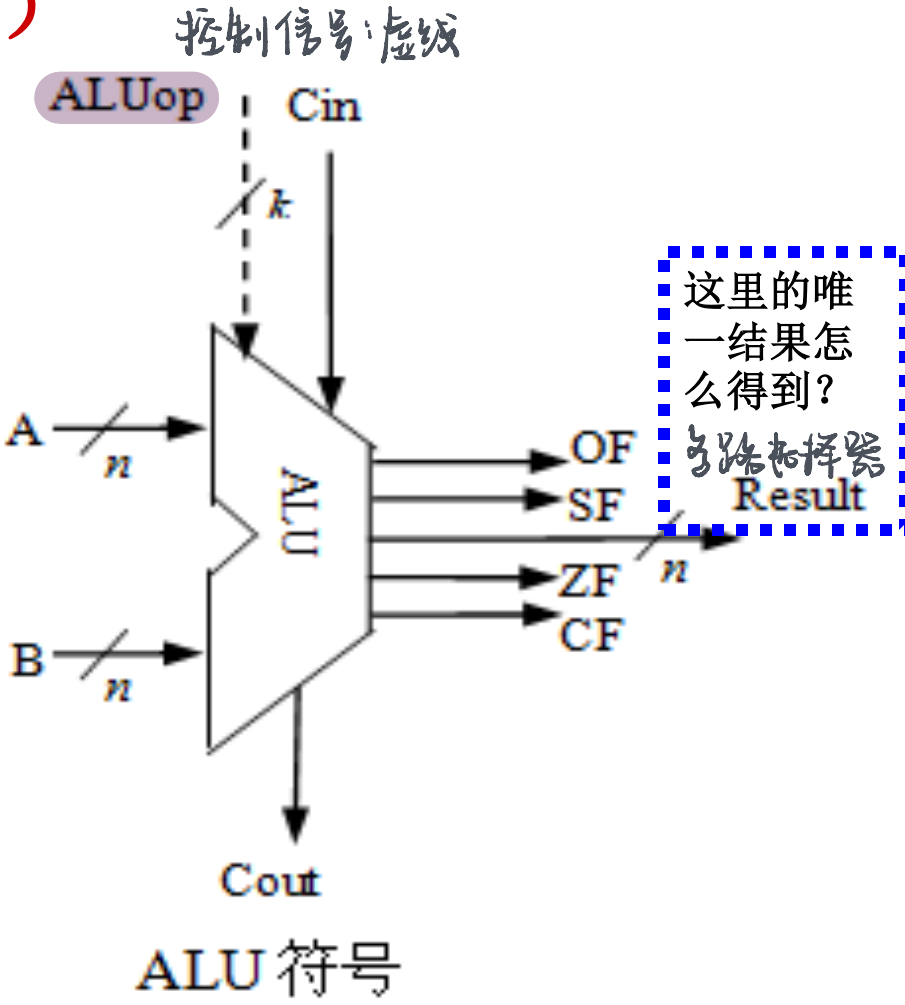
Carry Flag

CPU：中央处理器；PC：程序计数器；MAR：存储器地址寄存器
ALU：算术逻辑部件；IR：指令寄存器；MDR：存储器数据寄存器
GPRs：通用寄存器组（由若干通用寄存器组成）



算术逻辑部件（ALU）

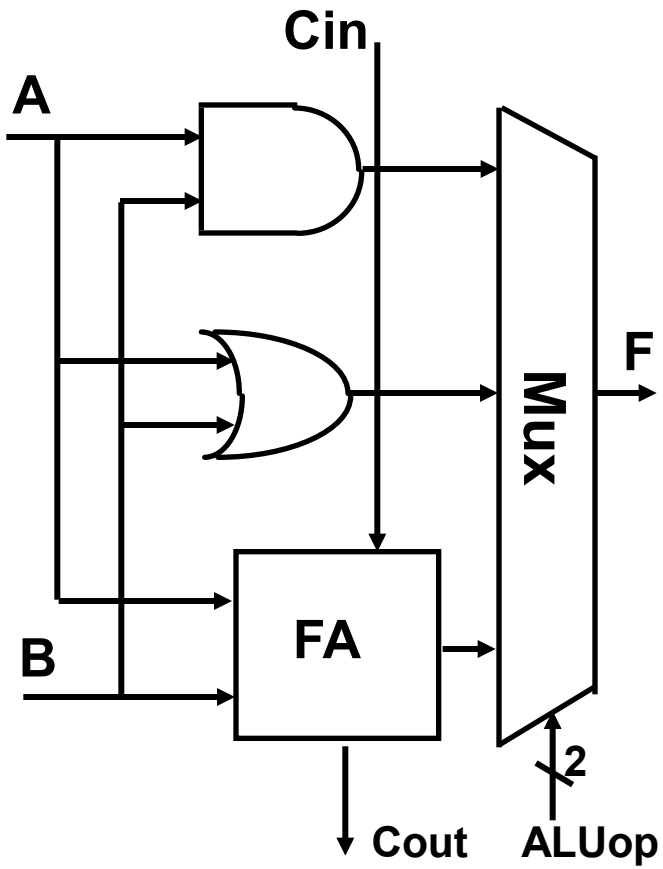
- 有一个操作控制端（ALUop），用来决定ALU所执行的处理功能。ALUop的位数k决定了操作的种类例如，当位数k为3时，ALU最多只有2^3=8种操作。
- 进行基本算术运算与逻辑运算
  - 无符号整数加、减
  - 带符号整数加、减
  - 与、或、非、异或等逻辑运算
- 核心电路是整数加/减运算部件
- 输出除和/差等，还有标志信息



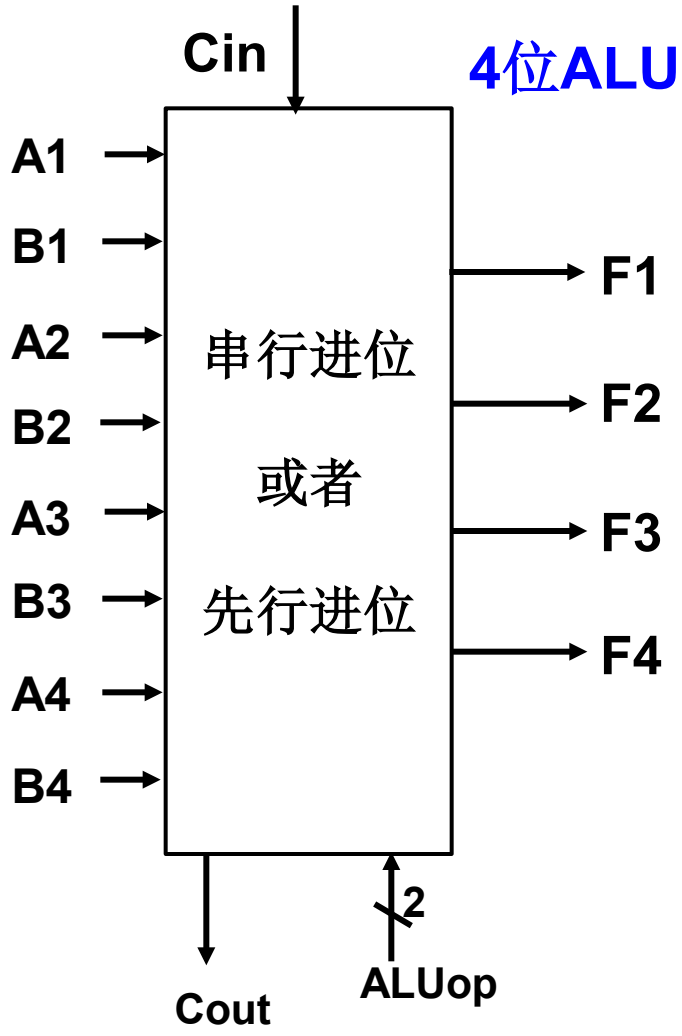
ALU 符号

ALUop	Result	ALUop	Result	ALUop	Result	ALUop	Result
0 0 0	A加B	0 1 0	A与B	1 0 0	A取反	1 1 0	A
0 0 1	A减B	0 1 1	A或B	1 0 1	A⊕B	1 1 1	未用

1-bit ALU和4-bit ALU（简化示意）

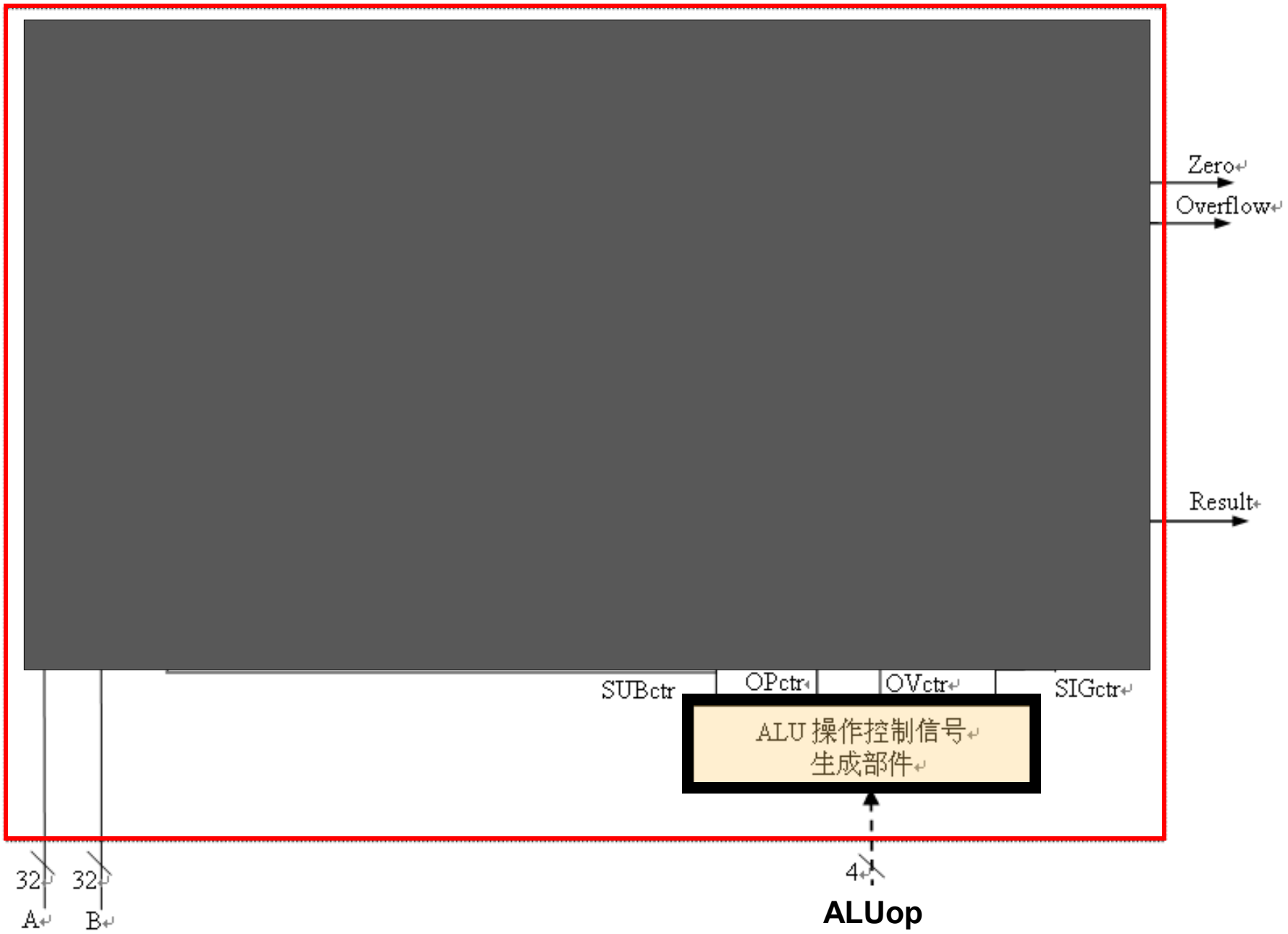


1-bit ALU

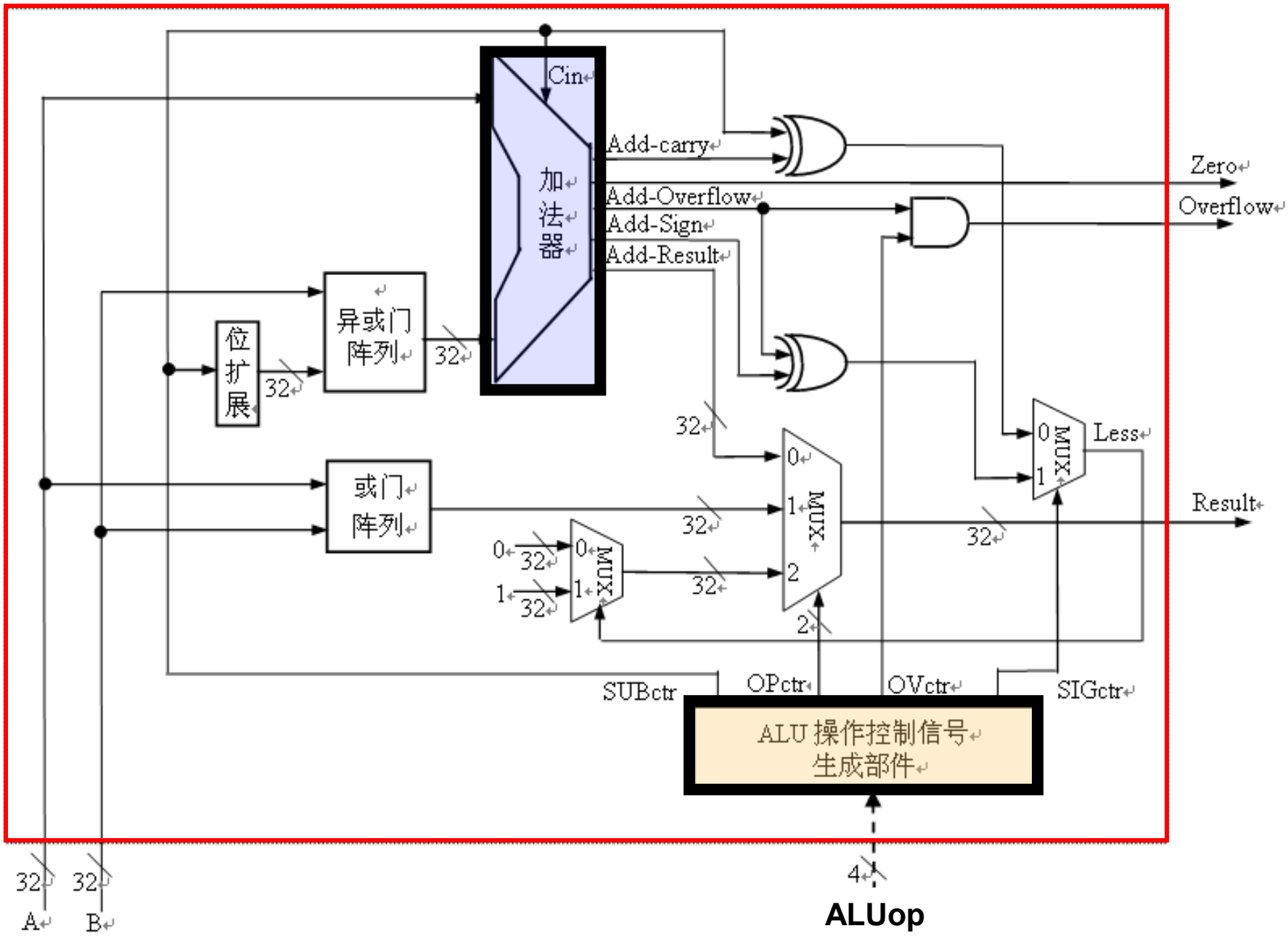


实际的ALU中还包括减法、算术移位、逻辑移位等其他运算功能

例：某ALU


Zero
Overflow
Result
SUBctr
OPctr
OVctr
SIGctr
ALU 操作控制信号
生成部件
32
32
4
A
B
ALUop

例：某ALU


Cin
加法器
Add-carry
Add-Overflow
Add-Sign
Add-Result
Zero
Overflow
异或门阵列
位扩展
32
或门阵列
MUX
Less
Result
SUBctr
OPctr
OVctr
SIGctr
ALU 操作控制信号生成部件
4
ALUop
A
B

第二讲：定点数运算

主　要　内　容

- 定点数加减运算
  - 补码加减运算
  - 原码加减运算
  - 移码加减运算
- 定点数乘法运算
  - 原码乘法运算
  - 补码乘法运算
  - 快速乘法器
- 定点数除法运算
  - 原码除法运算
  - 补码除法运算

n位整数加/减运算器

先看一个C程序段：

```
int x=9, y=-6, z1, z2;
z1=x+y;
z2=x-y;
```

补码的定义　假定补码有n位，则：

$[X]_{补}=2^n+X$

$(-2^{n-1}\le X<2^{n-1},\mod 2^n)$

问题：上述程序段中，x和y的机器数是什么？z1和z2的机器数是 什么？

回答：x的机器数为$[x]_{补}$，　y的机器数为$[y]_{补}$；

z1的机器数为$[x+y]_{补}$；

z2的机器数为$[x-y]_{补}$。

因此，计算机中需要有一个电路，能够实现以下功能：

已知 $[x]_{补}$ 和 $[y]_{补}$，计算$[x+y]_{补}$ 和 $[x-y]_{补}$。

$[-y]_{补}=\overline{[y]_{补}}+1$

根据补码定义，有如下公式：

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

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

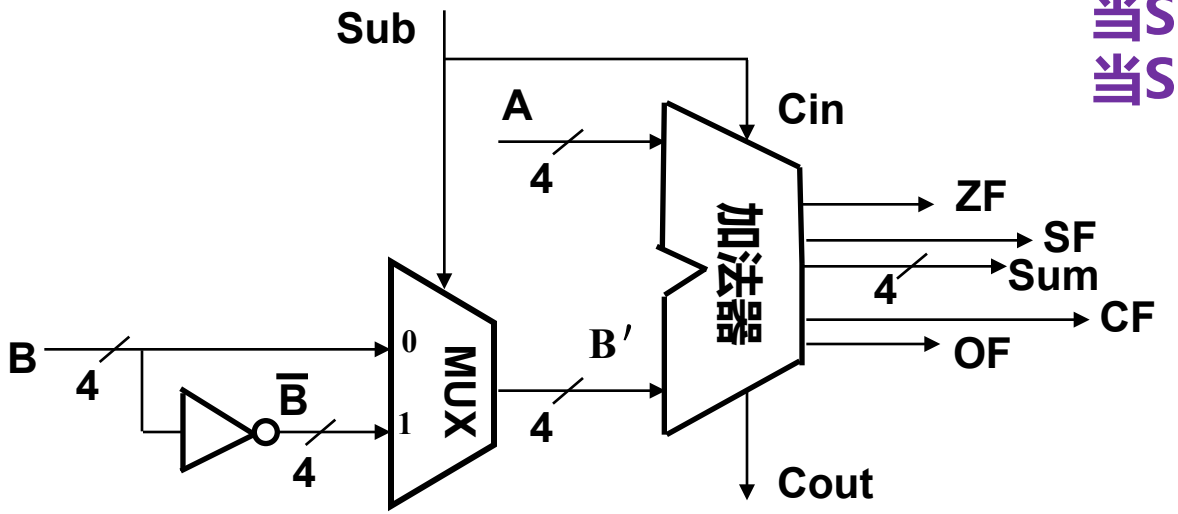
n位整数加/减运算器

- **补码加减运算公式**

$[A+B]_{补} = [A]_{补} + [B]_{补} \quad (\bmod 2^n)$

$[A-B]_{补} = [A]_{补} + [-B]_{补} \quad (\bmod 2^n)$

– **实现减法的主要工作在于：求 $[-B]_{补} = \overline{[B]_{补}} + 1$**



当Sub为1时，做减法
当Sub为0时，做加法

整数加/减运算部件

注意：在整数加/减运算部件基础上，加上寄存器、移位器以及控制逻辑，就可实现ALU、乘/除运算以及浮点运算电路

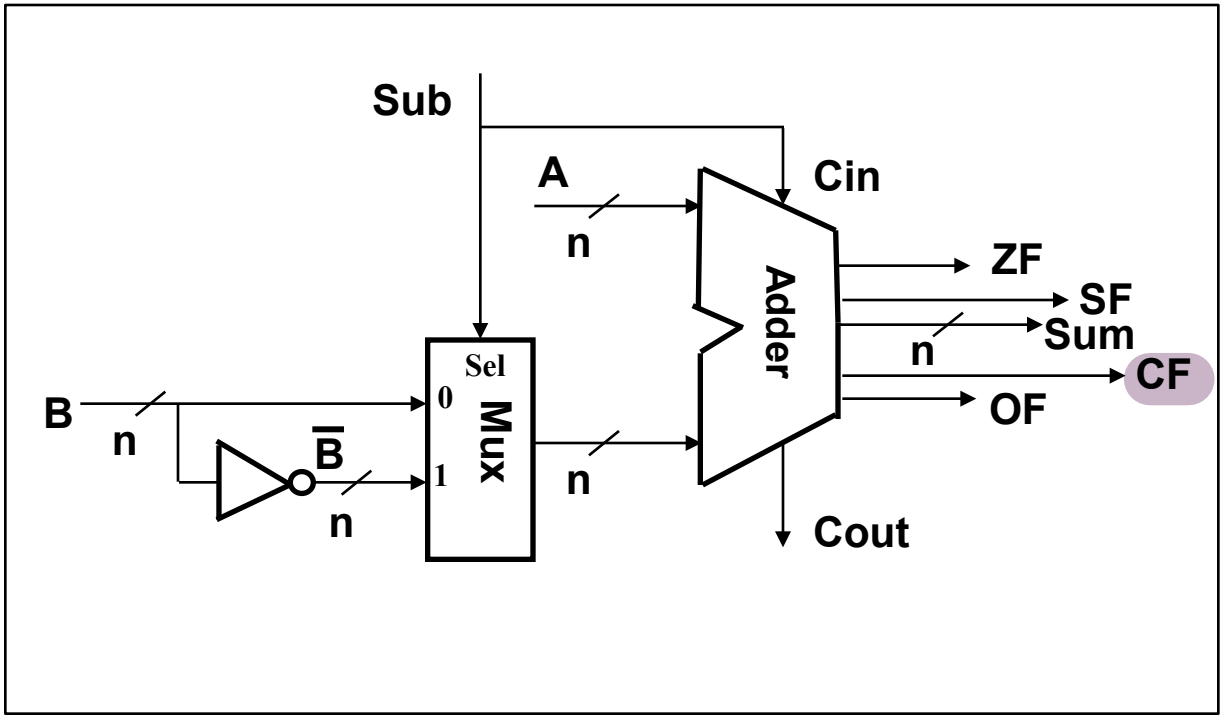
强调：带符号数、无符号数的加减法运算

- 利用带标志加法器，可构造n位整数加/减运算部件，进行以下运算：

  无符号整数加、无符号整数减

  带符号整数加、带符号整数减

无符号数减法也用补码减法实现，
只是结果解释(标志位使用)不同



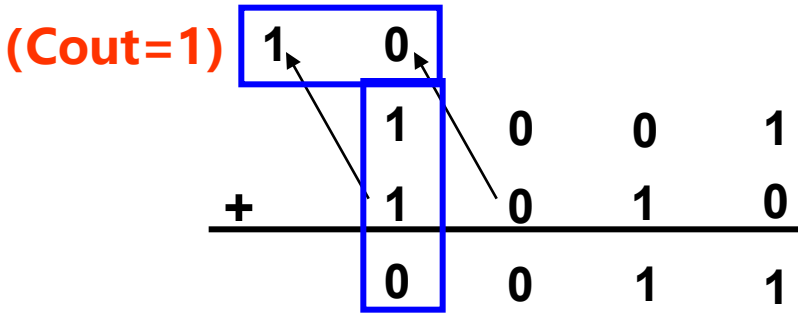
整数减法举例

注意：Cin=sub=1

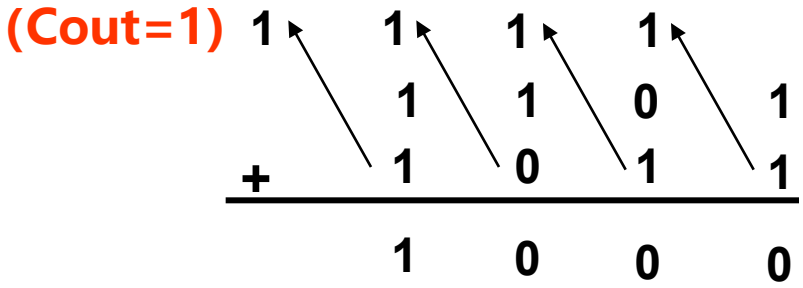
$OF=C_n \oplus C_{n-1}$
$CF=Cout \oplus Cin$

Signed -7- 6 = -7 + (-6) = +3 X
unsigned 9 - 6 = 9 + (-6) = 3 √

-3 - 5 = - 3 + (- 5) = - 8 √
13 - 5 = 13 + (- 5) = 8 √



OF=1、ZF=0
SF=0、借位CF=0



OF=0、ZF=0、
SF=1、借位CF=0

带符号溢出判断：

(1) 最高位和次高位的进位不同 **或者** (2) 和的符号位和加数的符号位不同

做减法以比较大小，规则：
Signed：OF=SF时，大于

验证：-7<6，故OF≠SF
-3<5，故OF≠SF

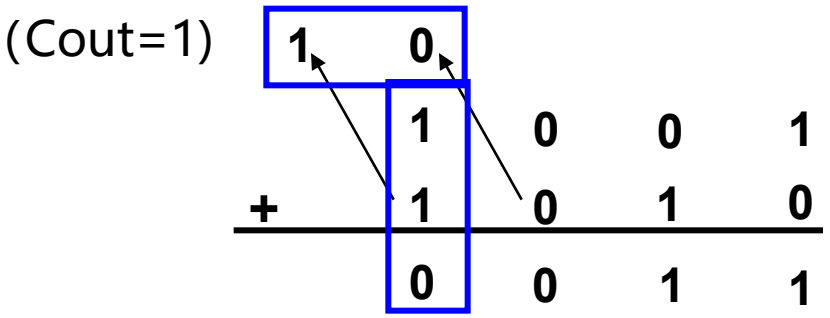
有符号数溢出 $Cn \oplus Cn-1$
比大小 OF? SF

无符号数减法例

注意：Cin=sub=1

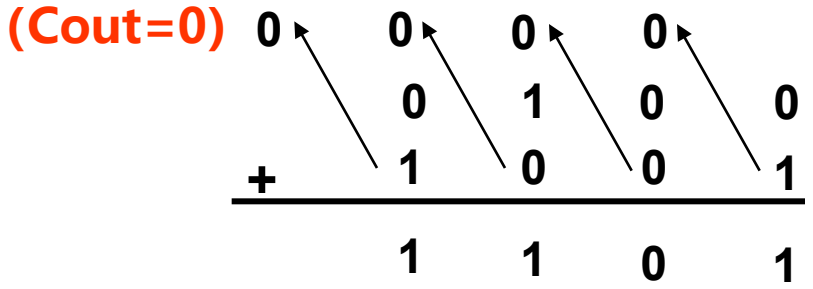
$OF=C_n \oplus C_{n-1}$
$CF=Cout \oplus Cin$

9 - 6 = 3 √



OF=1、ZF=0
SF=0、借位CF=0

4 - 7 = -3 X



OF=0、ZF=0、
SF=1、借位CF=1

无符号溢出判断：CF=1（减法时代表差为负数，即产生了借位）

（加法时Cin=0，所以CF=1代表产生了进位，也就是加法溢出了）

无符号溢出：CF
比大小：CF

做减法以比较大小，规则：
Unsigned: CF=0时，大于

验证：9>6，故CF=0；
13>5，故CF=0（见上页ppt）

验证：4< 7，故CF=1；

带(无)符号整数减法举例续

假定 int为8位

```
unsigned int x=134;
unsigned int y=246;
int m=x;
int n=y;
unsigned int z1=x-y;
unsigned int z2=x+y;
int k1=m-n;
int k2=m+n;
```

无符号减：

$$result=\begin{cases} x-y & (x-y>0) \\ x-y+2^n & (x-y<0) \end{cases}$$

带符号减：

$$result=\begin{cases} x-y-2^n & (2^{n-1}\leqslant x-y) & \text{正溢出} \\ x-y & (-2^{n-1}\leqslant x-y<2^{n-1}) & \text{正常} \\ x-y+2^n & (x-y<-2^{n-1}) & \text{负溢出} \end{cases}$$

但值不一样

x和m的机器数一样：1000 0110 (-122)

y和n的机器数一样：1111 0110 (-10)

```
  1000 0110
  0000 1001
+        1
-------------
  1001 0000
```

Cin=sub=1
Cout=0

z1和k1的机器数一样：1001 0000，标志位也一样CF=1，OF=0，SF=1

无符号z1的真值为144(=134-246+256，x-y<0，CF=1，溢出)

带符号k1的真值为-112 (=-122 – (-10) = -112，OF=0，正常)

带(无)符号整数加法举例续

假定 int为8位

```
unsigned int x=134;
unsigned int y=246;
int m=x;
int n=y;
unsigned int z1=x-y;
unsigned int z2=x+y;
int k1=m-n;
int k2=m+n;
```

无符号加公式：

$$result=\begin{cases} x+y & (x+y<2^n) \\ x+y-2^n & (2^n\leqslant x+y<2^{n+1}) \end{cases}$$

带符号加公式：

$$result=\begin{cases} x+y-2^n & (2^{n-1}\leqslant x+y) \quad \text{正溢出} \\ x+y & (-2^{n-1}\leqslant x+y<2^{n-1}) \quad \text{正常} \\ x+y+2^n & (x+y<-2^{n-1}) \quad \text{负溢出} \end{cases}$$

x和m的机器数一样：1000 0110 (-122)

y和n的机器数一样： 1111 0110 (-10)

```
  1000 0110
+ 1111 0110
-----------
1 0111 1100
```

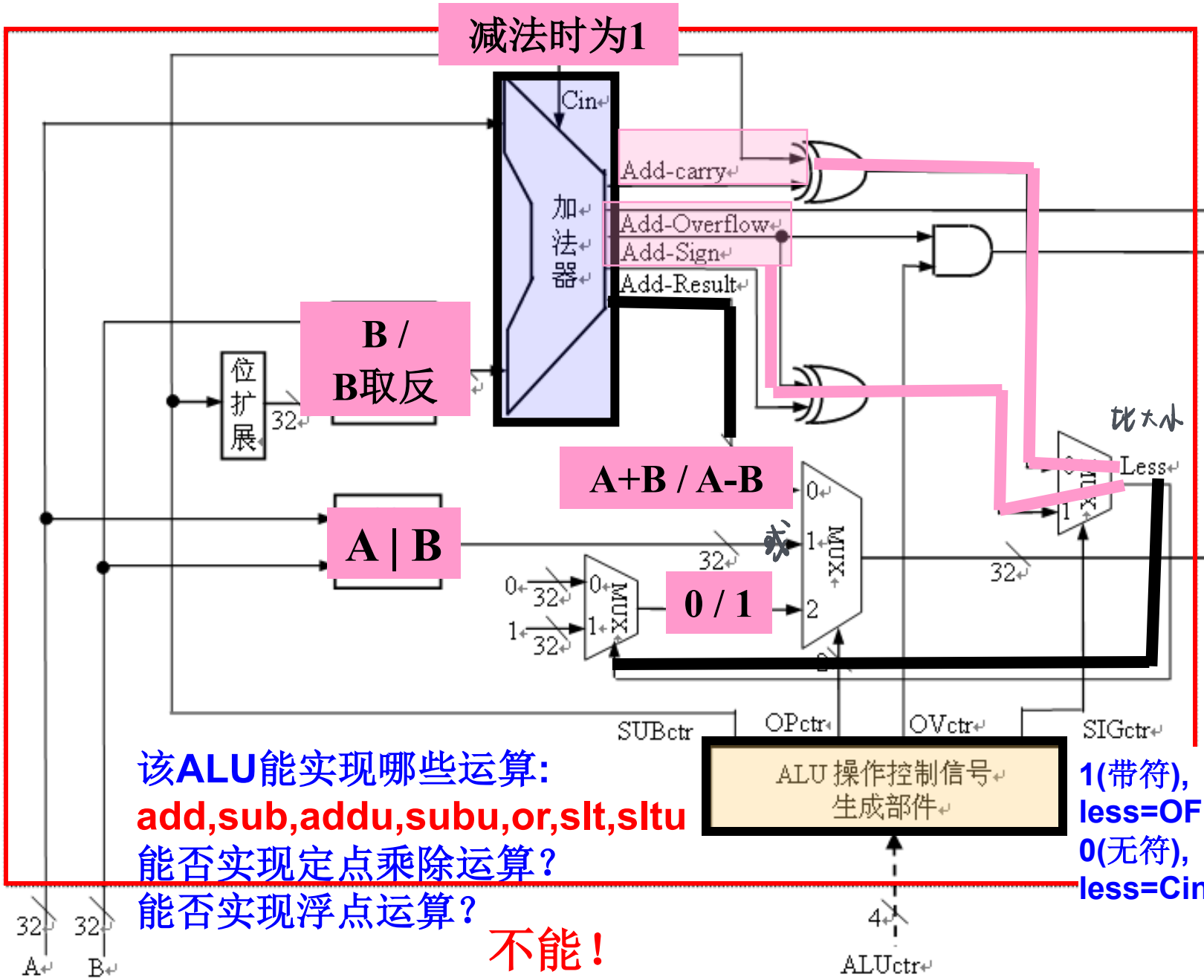
Cin=sub=0
Cout=1

z2和k2的机器数一样：0111 1100，标志位也一样 CF=1，OF=1，SF=0

z2的值为124（=134+246-256，x+y>256， CF=1，溢出）

k2的值为124（=-122+(-10)+256，m+n<-128， OF=1，负溢出）

例：实现部分机器指令功能的ALU



该ALU能实现哪些运算:

add,sub,addu,subu,or,slt,sltu

能否实现定点乘除运算？

能否实现浮点运算？

不能！

1(带符), less=OF⊕ SF

0(无符), less=Cin⊕Cout

无符号数的乘法运算

假定：$[X]_{原}=x_0.x_1\ldots x_n$，$[Y]_{原}=y_0.y_1\ldots y_n$，求$[x\times Y]_{原}$

数值部分 $z_1\ldots z_{2n}=(x_1\ldots x_n)\times(y_1\ldots y_n)$

(小数点位置约定——“定点”，无需区分小数还是整数)

- 手算乘法示例:

被乘数　　1000（X）

乘数　　x　1001（$Y\text{-}y_1y_2y_3y_4$）

1000

0000

0000

1000

积　　01001000

0.1000×0.1001

$=2^{-1}(2^{-1}(2^{-1}(2^{-1}(0.1000\times 1)+1000\times 0)+1000\times 0)+1000\times 1)$

右移后，有效数字丢失了吗？

——没有（预留存放的位置）

——整数和小数同理

两种操作：加法 + 移位

因而，可用ALU和移位器来实现乘法运算

无符号乘法运算的算法推导

- 上述思想可写成如下数学推导过程：

$$X\times Y=\sum_{i=1}^{4}(X\times y_i\times 2^{-i})$$

$$X\times Y = X\times(0.y_1\,y_2\cdots y_n)$$

$$=2^{-1}\underbrace{(2^{-1}(2^{-1}\cdots 2^{-1}}_{n\text{个}2^{-1}}(2^{-1}(0+X\times y_n)+X\times y_{n-1})+\cdots+X\times y_2)+X\times y_1)$$

- 递归！ n次
- 无符号数乘法可归结为：设$P_0=0$，每步的乘积为：

$$P_1 = 2^{-1}(P_0+X\times y_n)$$

$$P_2 = 2^{-1}(P_1+X\times y_{n-1})$$

…… ……

$$P_n = 2^{-1}(P_{n-1}+X\times y_1)$$

- 最终乘积$P_n = X\times Y$ (两个n位数相乘，得到2n位数）

迭代过程从乘数最低位y_n和P_0=0开始，
经n次“判断–加法–右移”循环，直到求出Pn为止。

$y_n=0$?

Example：无符号整数乘法运算

举例说明：

设X=1110　Y=1101

需要哪些存储空间？

应用递推公式：$P_i=2^{-1}(Xy_i+P_{i-1})$

C	乘积P	乘数Y	
0	0000	1101	
	+ 1110		
0	1110	1101	
→ 0	0111	0110	1
→ 0	0011	1011	01
	+ 1110		
1	0001	1011	01
→ 0	1000	1101	101
	+ 1110		
1	0110	1101	101
→ 0	1011	0110	1101

可用一个双倍字长的乘积寄存器；
也可用两个单倍字长的寄存器。

8位

+ X：4位

部分积初始为0。

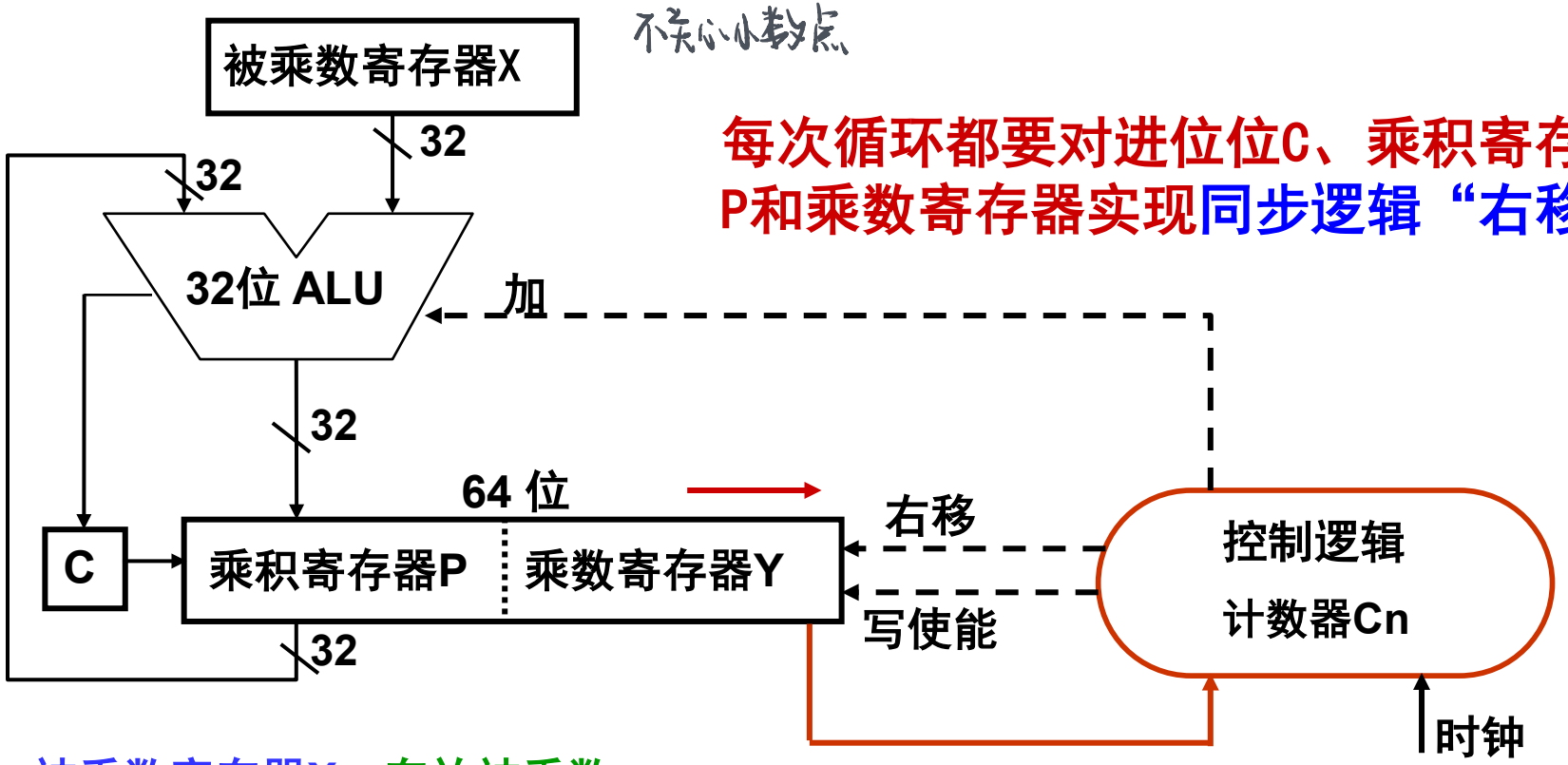
保留进位位。 1位

右移时进位、部分积和剩余乘数一起进行逻辑右移。

验证：X=14, Y=13, XY=182

当乘积取低4位时，结果发生溢出，因为高4位不为全0！

32位无符号乘法运算的硬件实现



每次循环都要对进位位C、乘积寄存器P和乘数寄存器实现同步逻辑“右移”

- 被乘数寄存器X：存放被乘数
- 乘积寄存器P：开始置初始部分积P0 = 0；结束时，存放的是64位乘积的高32位
- 乘数寄存器Y：开始时置乘数；结束时，存放的是64位乘积的低32位
- 进位触发器C：保存加法器的进位信号
- 循环次数计数器Cn：存放循环次数。初值32，每循环一次，Cn减1，Cn=0时结束
- ALU：乘法核心部件。在控制逻辑控制下，对P和X的内容“加”，在“写使能”控制下运算结果被送回P，进位位在C中

原码乘法算法

- 用于浮点数尾数乘运算
- 符号与数值分开处理：积符异或得到，数值用无符号乘法运算

例：设$[x]_{原}=0.1110$，$[y]_{原}=1.1101$，计算$[x\times y]_{原}$

解：数值部分用无符号数乘法算法计算：1110×1101= 1011 0110

符号位：$0\oplus 1=1$，所以：$[x\times y]_{原}=1.10110110$

(*)

一位乘法：每次只取乘数的一位判断，需n次循环，速度慢。

两位乘法：每次取乘数两位判断，只需n/2次循环，快一倍。

- 两位乘法递推公式：

00：$P_{i+1}=2^{-2}P_i$

01：$P_{i+1}=2^{-2}(P_i+X)$

10：$P_{i+1}=2^{-2}(P_i+2X)$

11：$P_{i+1}=2^{-2}(P_i+3X)=2^{-2}(P_i+4X-X)$
$=2^{-2}(Pi-X)+X$

y_{i-1}	y_i	T	操作（最后都要右移两位）	迭 代 公 式
0	0	0	$0\rightarrow T$	$2^{-2}(P_i)$
0	0	1	$+X\quad 0\rightarrow T$	$2^{-2}(P_i+X)$
0	1	0	$+X\quad 0\rightarrow T$	$2^{-2}(P_i+X)$
0	1	1	$+2X\quad 0\rightarrow T$	$2^{-2}(P_i+2X)$
1	0	0	$+2X\quad 0\rightarrow T$	$2^{-2}(P_i+2X)$
1	0	1	$-X\quad 1\rightarrow T$	$2^{-2}(P_i-X)$
1	1	0	$-X\quad 1\rightarrow T$	$2^{-2}(P_i-X)$
1	1	1	$1\rightarrow T$	$2^{-2}(P_i)$

3X时，本次-X，下次+X！

触发器T用来记录下次是否要执行“+X”

“–X”运算用“$+[-X]_{补}$”实现！

(*) 原码两位乘法举例

已知 $[X]_{原}$=0.111001， $[Y]_{原}$= 0.100111，用原码两位乘法计算$[X\times Y]_{原}$

解： 先用无符号数乘法计算111001×100111，原码两位乘法过程如下：

$[|X|]_{补}=000\ 111001$，$[-|X|]_{补}=111\ 000111$

P		Y	T	说明
000	000000	100111	0	开始，P_0=0，T=0
+111	000111			y_5y_6T= 110，–X，T=1
111	000111			P 和 Y 同时右移 2 位
111	110001	11 1001	1	得 P_1
+001	110010			y_3y_4T= 011，+2X，T=0
001	100011			P 和 Y 同时右移 2 位
000	011000	1111 10	0	得 P_2
+001	110010			y_1y_2T= 100，+2X，T=0
010	001010			P 和 Y 同时右移 2 位
000	100010	101111	0	得 P_3

加上符号位，得 $[X\times Y]_{原}$=0. 100010101111

采用补码算术右移，与一位乘法不同?

有加有减，所以要算术移位

为什么用模8补码形式(三位符号位)？

若用模4补码，中间涉及+2X会导致P和Y同时右移2位时，得到的P3是负数，就错了。

若最后T=1，则要+X

速度快，但代价也大

补码乘法运算

用于对什么类型的数据计算？已知什么？求什么？

带符号整数！如C语句：int x=-5，y=-4，z=x*y;

问题：已知$[x]_{补}$和$[y]_{补}$，求$[x*y]_{补}$

因为$[x*y]_{补} \neq [x]_{补}*[y]_{补}$，故不能直接用无符号整数乘法计算。

例如，若x=-5，求x*x=？： $[-5]_{补}$=1011

$[x*x]_{补}$： $[25]_{补}$=0001 1001---正确

$[x]_{补}*[x]_{补}$； $[-5]_{补}*[-5]_{补}$=1111 1001---错误！

思路：根据$[y]_{补}$求y，且$[A+B]_{补}=[A]_{补}+[B]_{补}$，

只要将$[x*y]_{补}$转换为对若干数的和求补即可

(*) 补码乘法运算Booth's Algorithm推导

假定：$[x]_{补}=x_{n-1}x_{n-2}\cdots\cdots x_1x_0$，$[y]_{补}=y_{n-1}y_{n-2}\cdots\cdots y_1y_0$，求：$[x^*y]_{补}=?$

基于补码求真值的公式：

$$y=-y_{n-1}\cdot 2^{n-1}+y_{n-2}\cdot 2^{n-2}+\cdots\cdots y_1\cdot 2^1+y_0\cdot 2^0$$

令：$y_{-1}=0$（不失正确性），则：

当n=4时，$y=-y_3\cdot 2^3+y_2\cdot 2^2+y_1\cdot 2^1+y_0\cdot 2^0+y_{-1}\cdot 2^0$

$$=-y_3\cdot 2^3+(y_2\cdot 2^3-y_2\cdot 2^2)+(y_1\cdot 2^2-y_1\cdot 2^1)+(y_0\cdot 2^1-y_0\cdot 2^0)+y_{-1}\cdot 2^0$$

$$=(y_2-y_3)\cdot 2^3+(y_1-y_2)\cdot 2^2+(y_0-y_1)\cdot 2^1+(y_{-1}-y_0)\cdot 2^0$$

不失正确性——

$$2^{-4}\cdot[x^*y]_{补}=\left[(y_2-y_3)\cdot x\cdot 2^{-1}+(y_1-y_2)\cdot x\cdot 2^{-2}+(y_0-y_1)\cdot x\cdot 2^{-3}+(y_{-1}-y_0)\cdot x\cdot 2^{-4}\right]_{补}$$

$$=\left[2^{-1}(2^{-1}(2^{-1}(2^{-1}(y_{-1}-y_0)\cdot x)+(y_0-y_1)\cdot x)+(y_1-y_2)\cdot x)+(y_2-y_3)\cdot x)\right]_{补}$$

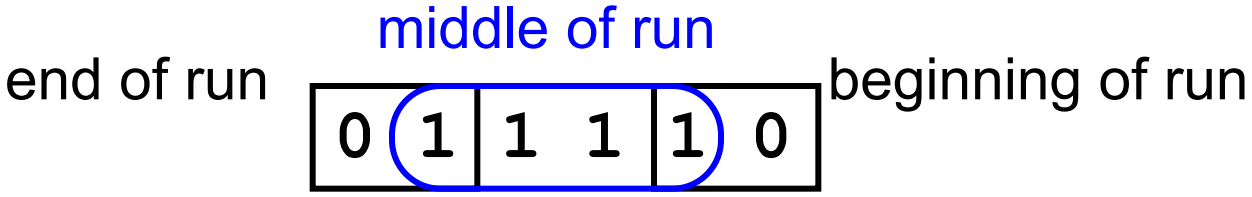
部分积公式：$[P_i]_{补}=[\,2^{-1}([P_{i-1}]_{补}+(y_{i-1}-y_i)\cdot x_{补})\,]_{补}$

注意：这里的yi就是补码中的某一位！

即：$[P_{i-1}]_{补}+[\pm x]_{补}$后右移一位（算术右移）

符号与数值统一处理

Booth's 算法实质



y当前位y_i	y右边位y_{i-1}	操作	Example
1	0	减被乘数x	0001111000
1	1	加0 (不操作)	0001111000
0	1	加被乘数x	0001111000
0	0	加0 (不操作)	0001111000

- 在“1串”中，第一个1时做减法，最后一个1做加法，其余情况只要移位。
- 最初提出这种想法是因为在Booth的机器上移位操作比加法更快！

同前面算法一样，将乘积寄存器右移一位。（这里是算术右移）

右移只是把位置空出来，最终从n位变为2n位空间，
小数点位置依然默认是在最左边的，所以并非是把真值缩小

布斯算法举例

如果X是-8， 那么 $[-X]_{补}$就溢出了?：除了移位实现（快），也可以P前面加补充符号位 （慢）

已知$[X]_{补}$ = 1 101，$[Y]_{补}$ = 0 110，计算$[X \times Y]_{补}$　$[-X]_{补}$ = 0011

X=-3，Y=6，X×Y=-18，$[X \times Y]_{补}$应等于11101110或结果溢出

P	Y	y_{-1}	说明
0 0 0 0	0 1 1 0	0	设 $y_{-1}=0$，$[P_0]_{补}=0$
	→ 1		$y_0\, y_{-1}=00$，P、Y 直接右移一位
0 0 0 0	0 0 1 1	0	得$[P_1]_{补}$
+0 0 1 1			$y_1\, y_0=10$，$+[-X]_{补}$
0 0 1 1	→ 1		P、Y 同时右移一位
0 0 0 1	1 0 0 1	1	得$[P_2]_{补}$
	→ 1		$y_2\, y_1=11$，P、Y 直接右移一位
0 0 0 0	1 1 0 0	1	得$[P_3]_{补}$
+1 1 0 1			$y_3\, y_2=01$，$+[X]_{补}$
1 1 0 1	→ 1		P、Y 同时右移一位
1 1 1 0	1 1 1 0	0	得$[P_4]_{补}$

这里即使产生进位，也是要丢掉的（也就是可以确保不溢出），留下来的4位就是正确的补码加法的结果

算术右移 负数补1

如何判断结果是否溢出?

高4位是否全为符号位!

验证：当X×Y取8位时，结果 -0010010B=-18；取低4位时，结果溢出

(*) 补码两位乘法

- 补码两位乘可用布斯算法推导如下：
  - $[P_{i+1}]_{补} = 2^{-1}\,([P_i]_{补} + (y_{i-1} - y_i)\,[X]_{补})$
  - $[P_{i+2}]_{补} = 2^{-1}\,([P_{i+1}]_{补} + (y_i - y_{i+1})\,[X]_{补})$

    $= 2^{-1}\,(2^{-1}\,([P_i]_{补} + (y_{i-1} - y_i)\,[X]_{补}) + (y_i - y_{i+1})\,[X]_{补})$

    $= 2^{-2}\,([P_i]_{补} + (y_{i-1} + y_i - 2y_{i+1})\,[X]_{补})$

- 开始置附加位y_{-1}为0，乘积寄存器最高位前面添加一位附加符号位0。
- 最终的乘积高位部分在乘积寄存器P中，低位部分在乘数寄存器Y中。
- 因为字长总是8的倍数，所以补码的位数n应该是偶数，因此，总循环次数为n/2。

y_{i+1}	y_i	y_{i-1}	操作(都要右移两位)	迭代公式
0	0	0	0	$2^{-2}[P_i]_{补}$
0	0	1	$+[X]_{补}$	$2^{-2}\{[P_i]_{补}+[X]_{补}\}$
0	1	0	$+[X]_{补}$	$2^{-2}\{[P_i]_{补}+[X]_{补}\}$
0	1	1	$+2[X]_{补}$	$2^{-2}\{[P_i]_{补}+2[X]_{补}\}$
1	0	0	$+2[-X]_{补}$	$2^{-2}\{[P_i]_{补}+2[-X]_{补}\}$
1	0	1	$+[-X]_{补}$	
1	1	0	$+[-X]_{补}$	$2^{-2}\{[P_i]_{补}+[-X]_{补}\}$
1	1	1	0	$2^{-2}\{[P_i]_{补}+[-X]_{补}\}$
				$2^{-2}[P_i]_{补}$

补码两位乘法举例

- 已知 $[X]_{补}$ = 1 101， $[Y]_{补}$ = 0 110，用补码两位乘法计算$[X\times Y]_{补}$。
- 解：$[-X]_{补}$= 0 011，用补码二位乘法计算$[X\times Y]_{补}$的过程如下。

P_n	P	Y	y_{-1}	说明
0	0 0 0 0	0 1 1 0	0	开始，设$y_{-1}=0$，$[P_0]_{补}=0$
+ 0	0 1 1 0			$y_1y_0y_{-1}$ =100，+2$[-X]_{补}$
0	0 1 1 0		→ 2	P和Y同时右移二位
0	0 0 0 1	1 0 0 1	1	得$[P_2]_{补}$
+ 1	1 0 1 0			$y_3y_2y_1$ = 011，+2$[X]_{补}$
1	1 0 1 1		→ 2	P和Y同时右移二位
1	1 1 1 0	1 1 1 0		得$[P_4]_{补}$

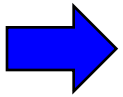
因此 $[X\times Y]_{补}$=1110 1110，与一位补码乘法（布斯乘法）所得结果相同，但循环次数减少了一半。

验证：-3×6=-18 （-10010B）

快速乘法器（不要求）

- 前面介绍的乘法部件的特点
  - 通过一个ALU多次做“加/减+右移”来实现
    - 一位乘法：约n次“加+右移”
    - 两位乘法：约n/2次“加+右移”

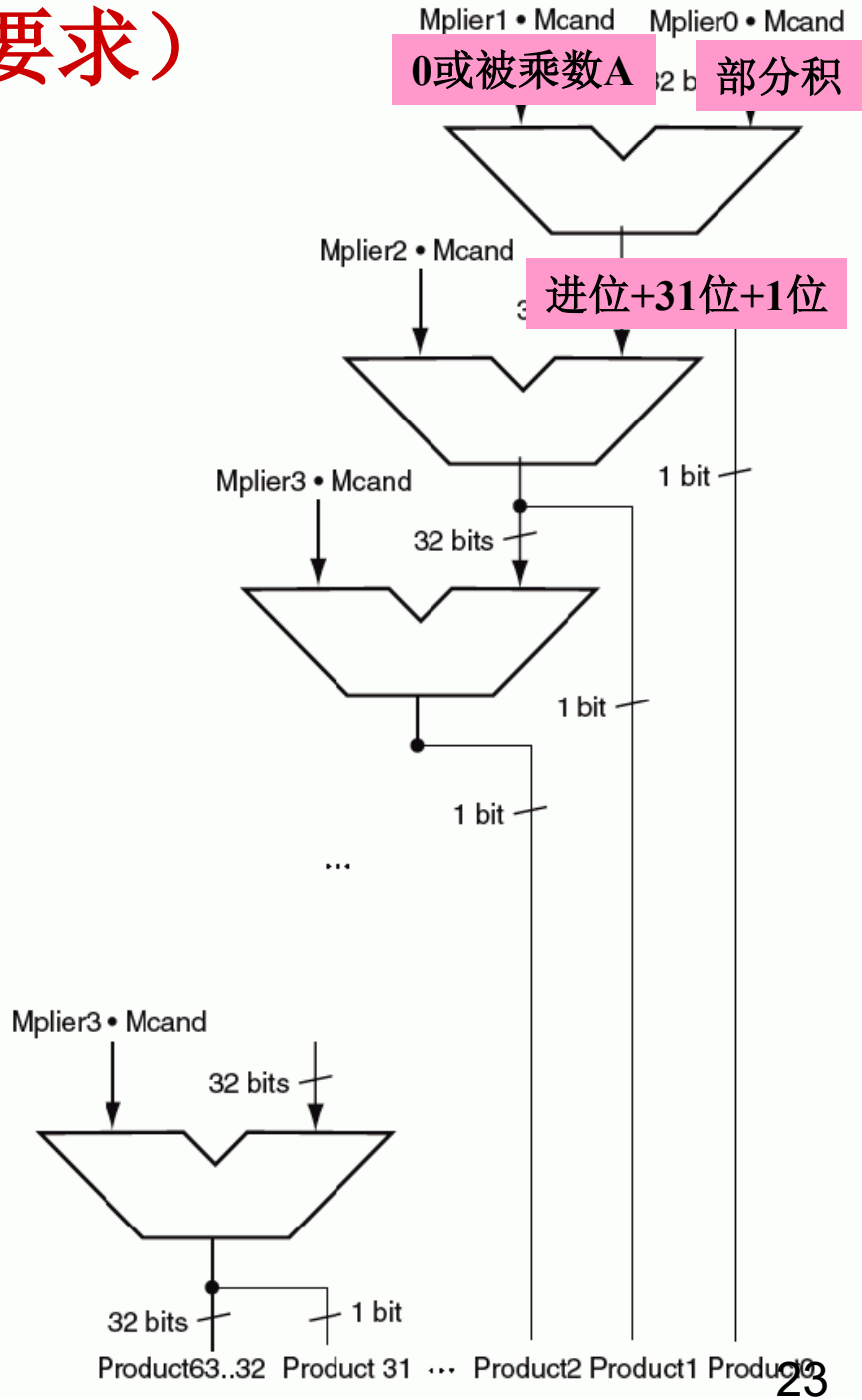
  所需时间随位数增多而加长，由时钟和控制电路控制
- 设计快速乘法部件的必要性
  - 大约1/3是乘法运算
- 快速乘法器的实现（由特定功能的组合逻辑单元构成）
  - 流水线方式
  - 硬件叠加方式（如：阵列乘法器）



流水线快速乘法器（不要求）

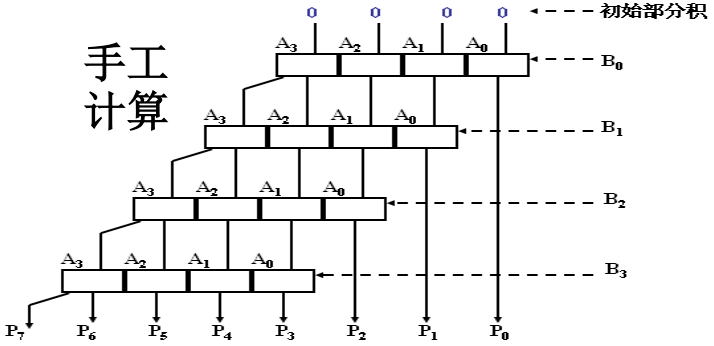
- 为乘数的每位提供一个n位加法器
- 每个加法器的两个输入端分别是：
  - 本次乘数对应的位与被乘数相与的结果（即：0或被乘数）
  - 上次部分积
- 每个加法器的输出分为两部分：
  - 和的最低有效位(LSB)作为本位乘积
  - 进位和高31位的和数组成一个32位数作为本次部分积

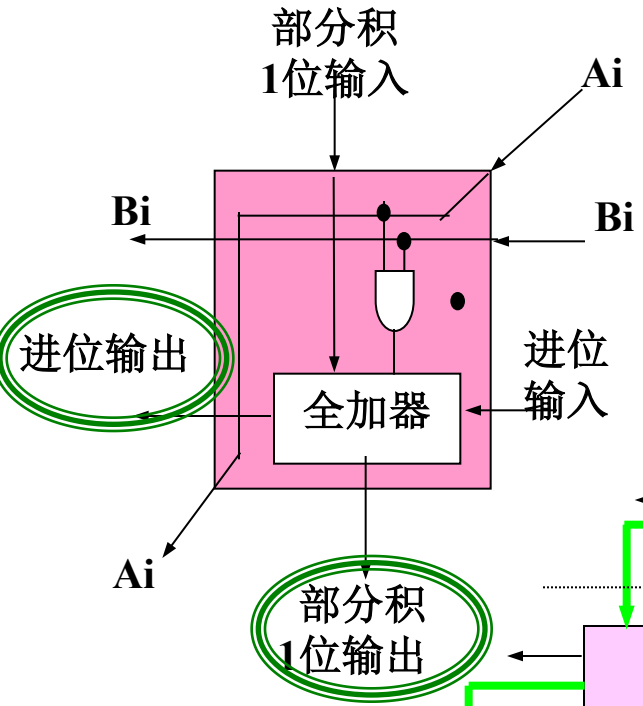
像流水一样，完全是串行，浪费加法器资源——但是，组合逻辑电路！无需控制器控制

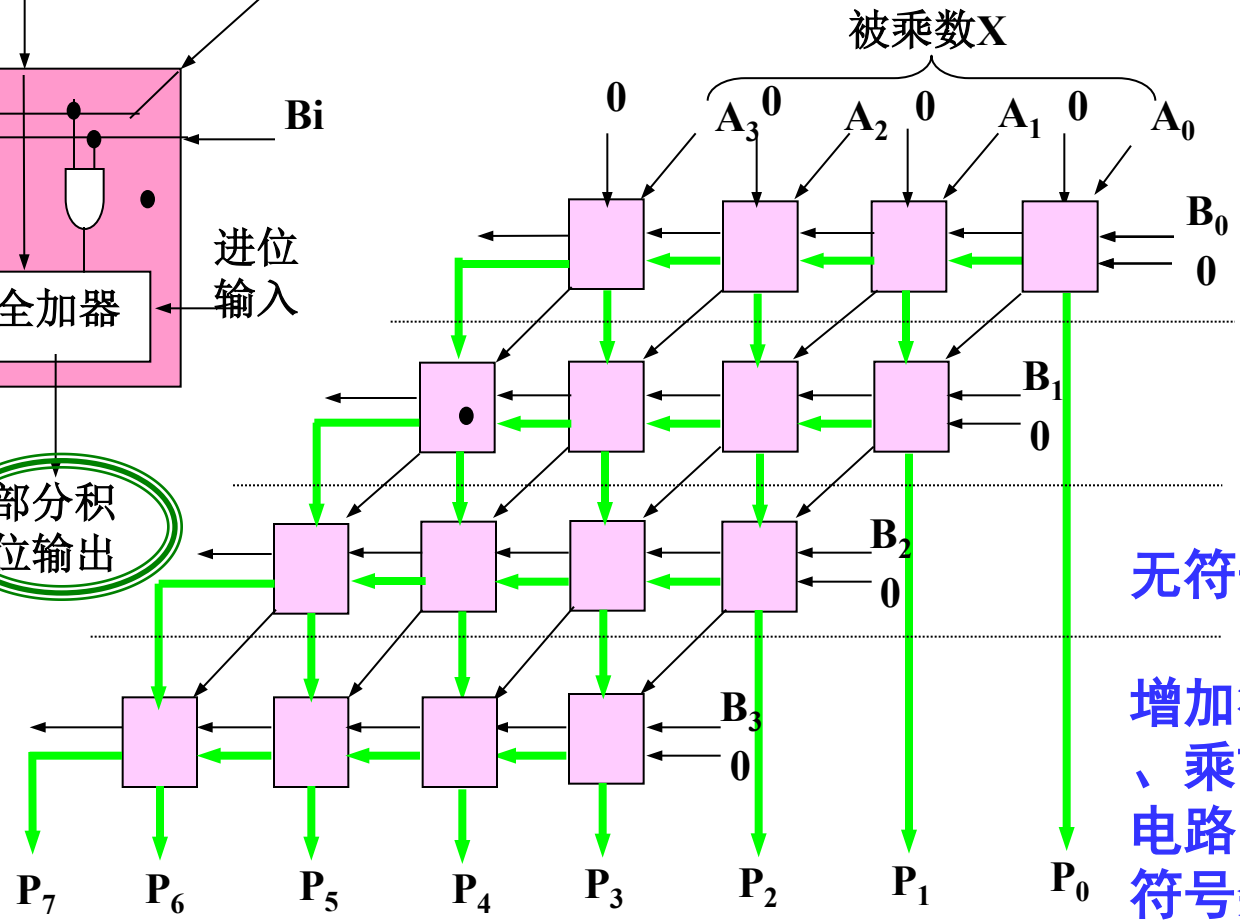


CRA阵列乘法器（不要求）

- 阵列乘法器：“细胞”模块的阵列







速度仅取决于逻辑门和加法器的传输延迟

无符号阵列乘法器

增加符号处理电路、乘前及乘后求补电路，即可实现带符号数乘法器。

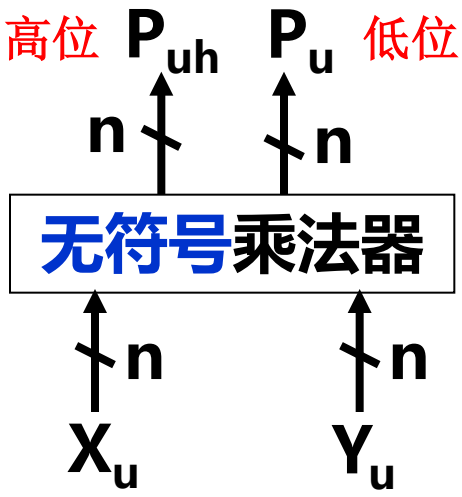
还可采用树形结构（如华莱士树）进行部分积求和，以加快速度

归纳：整数的乘运算

可用无符号乘来实现带符号乘。

n位 x n位，结果机器数可获得高n位和低n位。

高n位可用来判断溢出，也可直接作为乘积的高位（肯定不溢出）。



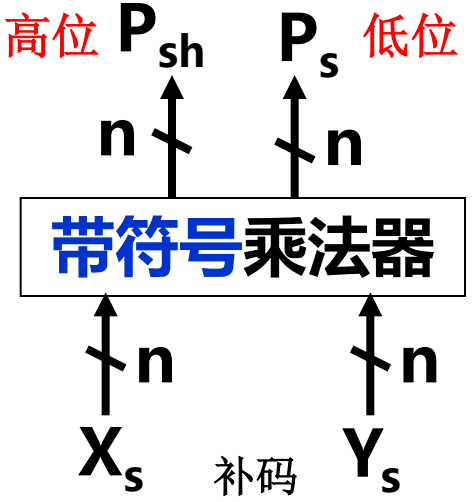
如果：

$X_u = X_s$

$Y_u = Y_s$

则：$P_u = P_s$

$P_{uh} \neq P_{sh}$ 不一定等于



小写字母都是真值(下页ppt)，大写字母都是机器数

u代表unsigned，s代表signed

整数的乘运算（溢出判断）

- **如果结果仅保留低n位，X*Y的高n位可以用来判断溢出，规则如下：**
  - **无符号：若高n位全0，则不溢出，否则溢出**
  - **带符号：若高n位全0或全1且等于低n位的最高位，则不溢出。**

运算	x	X	y	Y	x×y	X×Y	p	P	溢出否
无符号乘	6	0110	10	1010	60	0011 1100	12	1100	溢出
带符号乘	6	0110	-6	1010	-36	1101 1100	-4	1100	溢出
无符号乘	8	1000	2	0010	16	0001 0000	0	0000	溢出
带符号乘	-8	1000	2	0010	-16	1111 0000	0	0000	溢出
无符号乘	13	1101	14	1110	182	1011 0110	6	0110	溢出
带符号乘	-3	1101	-2	1110	6	0000 0110	6	0110	不溢出
无符号乘	2	0010	12	1100	24	0001 1000	8	1000	溢出
带符号乘	2	0010	-4	1100	-8	1111 1000	-8	1000	不溢出

整数的乘运算（机器级语言层面）

- **机器指令：分无符号数乘指令、带符号整数乘指令**
- **硬件可保留2n位乘积，故有些指令的乘积为2n位，可供软件使用**
- **乘法指令的操作数长度为n，而乘积长度为2n，例如：**
  - IA-32中，若指令只给出一个操作数SRC，则另一个源操作数隐含在累加器AL/AX/EAX中，将SRC和累加器内容相乘，结果存放在AX（16位时）或DX-AX（32位时）或EDX-EAX（64位时）中。
  - MIPS中，mult会将两个32位带符号整数相乘，得到的64位乘积置于两个32位内部寄存器Hi和Lo中，因此，可以根据Hi寄存器中的每一位是否等于Lo寄存器中的第一位来进行溢出判断。
  - RISC-V中，用“mul rd, rs1, rs2”获得低32位乘积并存入结果寄存器rd中；mulh、mulhu指令分别将两个乘数同时按带符号整数、同时按无符号整数相乘后，得到的高32位乘积存入rd中

乘法指令可生成溢出标志，编译器可使用2n位乘积来判断是否溢出！

高级语言程序也可以增加防止溢出的代码。（如果都不做，可能出严重错误）

整数的乘运算（机器级语言层面）

- **机器指令：分无符号数乘指令、带符号整数乘指令**
- **硬件可保留2n位乘积，故有些指令的乘积为2n位，可供软件使用**
- **乘法指令的操作数长度为n，而乘积长度为2n，例如：**
  - IA-32中，[redacted]个源操作数隐含在累加器AL[redacted]，结果存放在AX（16位时）[redacted]位时）中。
  - MIPS中，[redacted]得到的64位乘积置于两个32[redacted]据Hi寄存器中的每一位是否[redacted]断。
  - RISC-V中，用“mul rd, rs1, rs2”获得低32位乘积并存入结果寄存器rd中；mulh、mulhu指令分别将两个乘数同时按带符号整数、同时按无符号整数相乘后，得到的高32位乘积存入rd中

——乘法指令的硬件实现时就进行溢出判断和标志生成

——编译后生成的指令序列：
指令1：mul r1, rs1, rs2
指令2：mulh r2, rs1, rs2
指令3。。。：判断r1和r2的内容情况
指令x：如果溢出就跳转。。

乘法指令可生成溢出标志，编译器可使用2n位乘积来判断是否溢出！

高级语言程序也可以增加防止溢出的代码。（如果都不做，可能出严重错误）

整数的乘运算（高级语言程序层面）

在计算机内部，一定有$x^2 \geq 0$吗?

若x是带符号整数，则不一定!

如x是浮点数，则一定!

例如，当 n=4 时, $5^2=-7<0$!

```
    0101
  × 0101
-------
    0101
+ 0101
-------
 00011001
```

结果溢出

只取低4位，值为-111B=-7

```
int imul_overflow(int x, int y)
{//判断是否溢出
    return x*y/y !=x
}
```

多进行一次除法运算，程序变慢!

注意：这里是针对【n位 与 n位相乘，结果保留n位】的情况。

思考（自学）

在字长为32位的计算机上，某C函数原型声明为：

int imul_overflow(int x, int y);

该函数用于对两个int型变量x和y的乘积（也是int类型）判断是否溢出，若溢出则返回非0，否则返回0。请完成下列任务或回答下列问题。

（1）两个n位无符号数（带符号整数）相乘的溢出判断规则各是什么?

无符号整数相乘：若乘积的高n位为非0，则溢出。

带符号整数相乘：若乘积高n位的每一位都相同，且都等于乘积低n位的符号，则不溢出，否则溢出。

（2）已知入口参数x、y分别在寄存器a0、a1中，返回值在a0中，写出实现imul_overflow函数功能的RISC-V汇编指令序列，并给出注解。（编译器中判断溢出的代码，学完第7章再做）

（3）使用64位整型（long long）变量来编写imul_overflow函数的C代码或描述实现思想。

思考（自学）

（2）RISC-V汇编指令序列

实现该功能的汇编指令序列不唯一。

某实现方案下的汇编指令序列如下：

```
mul     t0, a0, a1          # x*y的低32位在t0中
mulh    a0, a0, a1          # x*y的高32位在a0中
srai    t0, t0, 31          # 乘积的低32位算术右移31位
xor     a0, a0, t0          # 按位异或，若结果为0，表示不溢出
```

思考（自学）

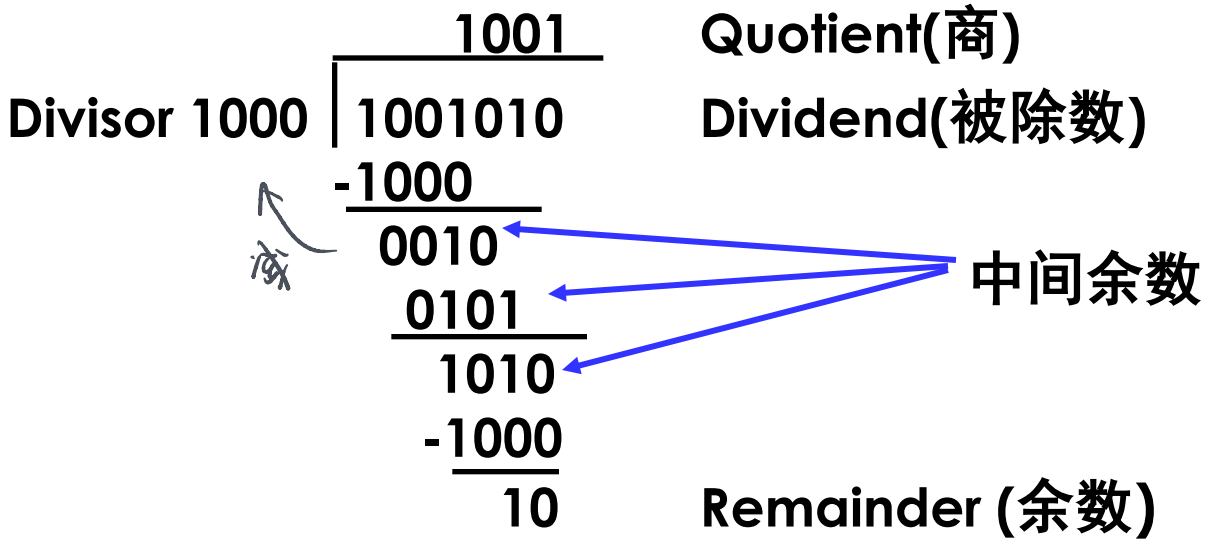
（3）采用long long型变量实现的C程序

将x*y的结果保存在long long型变量中，得到64位乘积，然后把64位乘积强制转换为32位，再符号扩展成64位，和原来真正的64位乘积相比，若不相等则溢出。

```
int imul_overflow(int x, int y)
{
    long long prod_64= (long long) x*y;
    return prod_64 != (int) prod_64;
}
```

例如：x=-4,y=6, 位数n=4
则prod_8=1110 1000
截断后为1000
重新扩展为1111 1000

除法Divide: Paper & Pencil



- 手算除法的基本要点

（1）被除数减去除数（以除数的位数为准对齐）

够减则上商为1；不够减则上商为0。得到的差为中间余数

（2）除数右移一位，然后用中间余数减去除数或0

够减则上商1；否则上商0。得到的差仍为中间余数，

重复执行本步骤。

（3）直到求得的商的位数足够为止。

定点除法运算

- 除前预处理

什么时候做？谁在做？　　软件or硬件

①若被除数=0且除数≠0，或定点整数除法|被除数|<|除数|，则商为0，不再继续

②若被除数≠0、除数=0，则发生“除数为0”异常

③若被除数和除数都为0，则有些机器产生一个不发信号的NaN，即“quiet NaN”

当被除数和除数都≠ 0，且商≠ 0时，才进一步进行除法运算。

- 计算机内部无符号数除法运算
  - 与手算一样，通过被除数（中间余数）减除数来得到每一位商够减上商1；不够减上商0
  - 基本操作为减法（用加法实现）和移位，与乘法用同一套硬件

减法计算完之后发现不够减怎么办？

说明这次本不该减！要恢复余数！

定点除法运算（两个n位正数相除的情况）

与乘法的不同

(1) 定点正整数（即无符号数 ）相除：在被除数的高位添n个0

(2) 定点正小数（即原码小数）相除：在被除数的低位添加n个0

这样，就将所有情况都统一为：一个2n位数除以一个n位数

Divisor 1.0100

0.1000 | 0.10100000

1000

0100

1000

—1000

0

Divisor 00001 Quotient(商)

1000 | 00001010 Dividend(被除数)

0001

0010

0101

1010

— 1000

10 Remainder (余数)

中间余数

n位除法需 n+1步

手算中的“除数右移”改为“被除数（中间余数）左移”

第一次试商为1时的情况

问题：第一次试商为1，说明什么?

商有n+1位数，因而溢出!

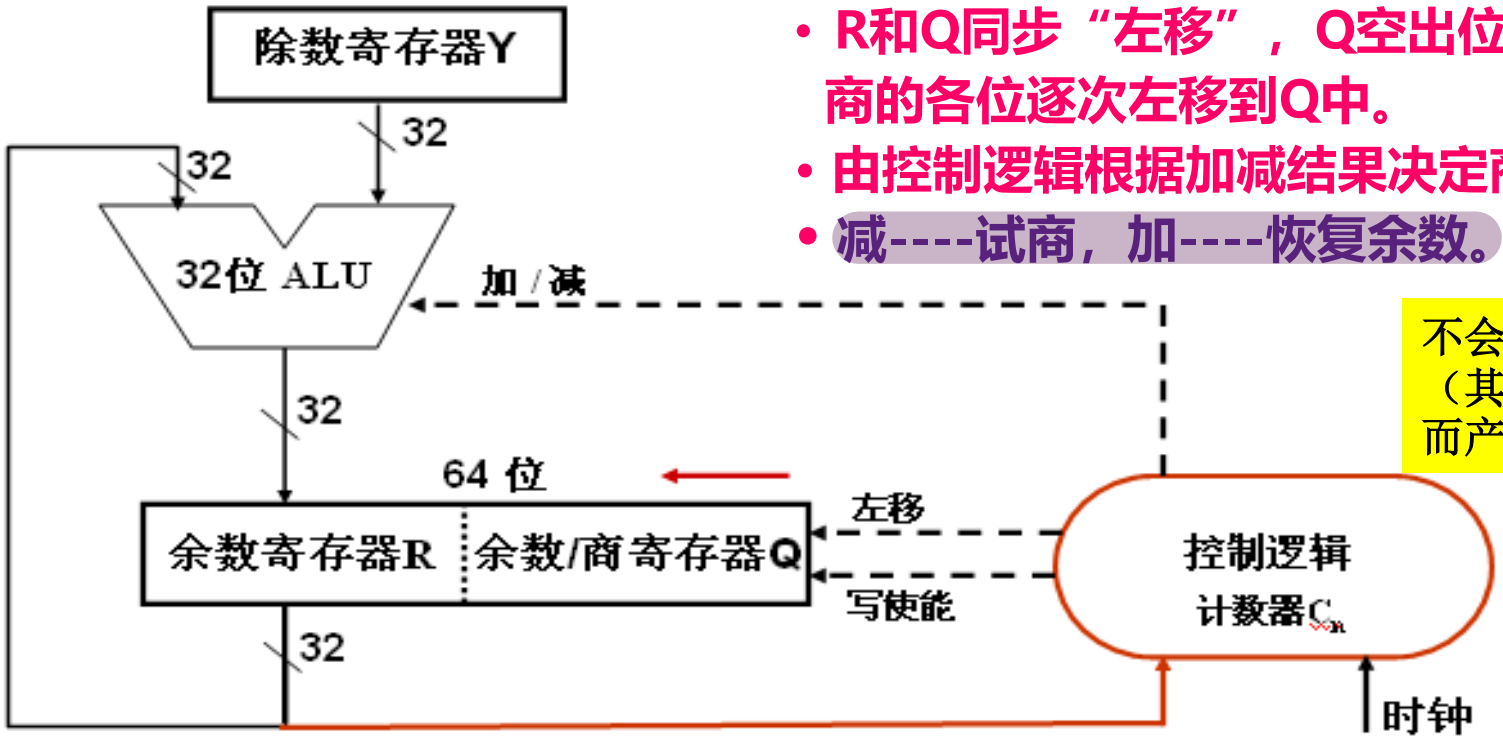
若是2n位除以n位的无符号整数运算，则说明将会得到多于n+1位的商，因而结果“溢出”（即：无法用n位表示商）。

例：1111 1111/1111 = 1 0001

若是两个n位数相除，则第一位商为0，且肯定不会溢出，为什么?

最大商为: 0000 1111/0001=1111

无符号数除法算法的硬件实现



- R和Q同步“左移”，Q空出位上“商”，商的各位逐次左移到Q中。
- 由控制逻辑根据加减结果决定商为0还是1
- 减----试商，加----恢复余数。

不会因为加法（其实是减法）而产生最高进位C

- 除数寄存器Y：存放除数。
- 余数寄存器R：初始时高位部分为高32位被除数；结束时是余数。
- 余数/商寄存器Q：初始时为低32位被除数；结束时是32位商。
- 循环次数计数器Cn：存放循环次数。初值是32（不包括第一次试商），每循环（移位）一次，Cn减1，当Cn=0时，除法运算结束。
- ALU：除法核心部件。在控制逻辑控制下，对于寄存器R和Y的内容进行“加/减”运算，在“写使能”控制下运算结果被送回寄存器R。

无符号数除法例子

验证：7 / 2 = 3 余 1

+D = 0010
–D = 1110

R：被除数（中间余数）；D：除数

4位无符号数（但数值范围只能是1-7，因为用补码实现减法，最高位需留做符号位），最后商是4位。

整数，所以R初始在高位扩展0

这里是两个n位无符号数相除，肯定不会溢出，故余数先左移而省略判断溢出过程。

	D: 0010	R: 0000 0111
Shl R	D: 0010	R: 0000 1110
R = R–D	D: 0010	R: 1110 1110 试商
+D, sl R, 0	D: 0010	R: 0001 1100 (恢复余数，左移，上商 0)
R = R–D	D: 0010	R: 1111 1100
+D, sl R, 0	D: 0010	R: 0011 1000
R = R–D	D: 0010	R: 0001 1000
sl R, 1	D: 0010	R: 0011 0001
R = R–D	D: 0010	R: 0001 0001
sl R, 1	D: 0010	R: 0010 0011
Shr R(rh)	D: 0010	R: 0001 0011

从例子可看出：

每次上商为0时，需做加法以“恢复余数”。所以，称为“恢复余数法

最后为了上商，把余数也左移了一位（共移了5次），故最后余数需向右移一位

不恢复余数除法(加减交替法)

恢复余数法可进一步简化为“加减交替法”

根据恢复余数法(设D为除数，$R_i=2R_{i-1}-D$为第i次中间余数)，有：

- 若$R_i<0$,则商上“0”，做加法恢复余数，即：
- $R_{i+1}=2(R_i+D)-D=2R_i+D$ （“负，左移，上商0，加”）
- 若$R_i>=0$,则商上“1”，不需恢复余数，即：
- $R_{i+1}=2R_i-D$ （“正，左移，上商1，减”）

省去了恢复余数的过程

- 注意：最后一次上商为“0”的话，需要“纠余”处理，即把试商时被减掉的除数加回去，恢复真正的余数。
- 不恢复余数法也称为加减交替法

Divide Algorithm example

验证：7 / 2 = 3 余 1　　–D = 1110

R：被除数（中间余数）；D：除数

R: 0000 0111

R = R–D　　1110

1110

sl R, 0　　R: 1100 1110

R = R+D　　0010

1110

sl R, 0　　R: 1101 1100

R = R+D　　0010

1111

sl R, 0　　R: 1111 1000

R = R+D　　0010

0001

sl R, 0　　R: 0011 0001

R = R-D　　1110

0001 0011

第1次上商为“试商”

不恢复余数法、加减交替法

负，0，加

正，1，减

第1位商为0，表示结果不溢出，最后（第5次）左移出去，并加上最后一位商

且最后一次上商1，余数无需恢复就是正确的。

这里的最后 一步，余数保持不变，没有和商一起左移，所以也不用右移了

带符号数除法

- 原码除法
  - 商符和商值分开处理
    - 商的数值部分由无符号数除法求得
    - 商符由被除数和除数的符号确定：同号为0，异号为 1
  - 余数的符号同被除数的符号
- 补码除法
  - 方法1：同原码除法一样，先转换为正数，先用无符号数除法，然后修正商和余数。
  - 方法2：直接用补码除法，符号和数值一起进行运算，商符直接在运算中产生。

若是两个n位补码整数除法运算，则被除数进行符号扩展。

若被除数为2n位，除数为n位，则被除数无需扩展。

原码除法举例（小数）

已知 $[X]_{原}$ = 0.1011

$[Y]_{原}$ = 1.1101

用恢复余数法计算$[X/Y]_{原}$

解：$[|X|]_{补}$ = 0.1011

$[|Y|]_{补}$ = 0.1101

$[-|Y|]_{补}$ = 1.0011

商的符号位：0 ⊕ 1 = 1

减法操作用补码加法实现，是否够减通过中间余数的符号来判断，所以中间余数要加一位符号位。

小数在低位扩展0

余数寄存器R	余数/商寄存器Q	说　明
✗ 01011	0000□	开始 $R_0=X$
+10011		$R_1=X-Y$
11110	00000	$R_1<0$，则 $q_4=0$
+01101		恢复余数：$R_1=R_1+Y$
01011		得 R_1
10110	0000□	$2R_1$（R和Q同时左移，空出一位商）
+10011		$R_2=2R_1-Y$
01001	00001	$R_2>0$，则 $q_3=1$
10010	0001□	$2R_2$ （R和Q同时左移，空出一位商）
+10011		$R_3=2R_2-Y$
00101	00011	$R_3>0$，则 $q_2=1$
01010	0011□	$2R_3$ （R和Q同时左移，空出一位商）
+10011		$R_3=2R_2-Y$
11101	00110	$R_4<0$，则 $q_1=0$
+01101		恢复余数：$R_4=R_4+Y$
01010	00110	得 R_4
10100	0110□	$2R_4$ （R和Q同时左移，空出一位商）
+10011		$R_5=2R_4-Y$
00111	01101	$R_5>0$，则 $q_0=1$

商的最高位为0，说明没有溢出，商的数值部分为 1101

所以，$[X/Y]_{原}$=1.1101 (最高位为符号位)，余数为 0.0111×2^{-4}

用于判断是否溢出

若求$[Y/X]_{原}$结果溢出

思考：若实现无符号数相除，即1011除以1101，则有何不同？结果是什么？

被除数高位补0，1011除以1101，结果等于0

原码除法举例

已知 $[X]_{原} = 0.1011$

$[Y]_{原} = 1.1101$

用加减交替法计算$[X/Y]_{原}$

解：$[|X|]_{补} = 0.1011$

$[|Y|]_{补} = 0.1101$

$[-|Y|]_{补} = 1.0011$

余数寄存器R	余数/商寄存器Q	说　明
01011	0000□	开始 $R_0=X$
+10011		$R_1=X-Y$
11110	00000	$R_1<0$，则 $q_4=0$，没有溢出
11100	0000□	$2R_1$（R和Q同时左移，空出一位商）
+01101		$R_2=2R_1+Y$
01001	00001	$R_2>0$，则 $q_3=1$
10010	0001□	$2R_2$（R和Q同时左移，空出一位商）
+10011		$R_3=2R_2-Y$
00101	00011	$R_3>0$，则 $q_2=1$
01010	0011□	$2R_3$（R和Q同时左移，空出一位商）
+10011		$R_3=2R_2-Y$
11101	00110	$R_4<0$，则 $q_1=0$
11010	0110□	$2R_4$（R和Q同时左移，空出一位商）
+01101		$R_5=2R_4+Y$
00111	01101	$R_5>0$，则 $q_0=1$

“加减交替法”的要点：

负、0、加

正、1、减

得到的结果与恢复余数法一样！

用被除数（中间余数）减除数试商时，怎样确定是否“够减”？

中间余数的符号！（正数-够减）

补码除法能否这样来判断呢？

补码除法

- 补码除法判断是否“够减”的规则

（1）当被除数（或当前余数）与除数同号时，做减法，得到新余数

（2）当被除数（或当前余数）与除数异号时，做加法，得到新余数

若新余数的符号与当前余数符号一致表示够减，否则为不够减；

当前余数R的符号	除数Y的符号	同号：新中间余数=R–Y（同号为正商）		异号：新中间余数=R+Y（异号为负商）	
		0	1	0	1
0	0	够减	不够减		
0	1			够减	不够减
1	0			不够减	够减
1	1	不够减	够减		

即：余数不变号够减、变号不够减

经过分析归纳（过程略），得到接下来的不恢复余数法

补码（n位，包括1位符号位）不恢复余数法

- **算法要点：**

判断是否同号（决定加or减、上商）不是新老余数之间！而是余数和除数Y之间

(1) 操作数的预置：

除数装入除数寄存器Y，被除数经符号扩展后装入余数寄存器R和余数/商寄存器Q。

(2) 根据以下规则求第一位商q_n

若被除数X与Y同号，则R1=X–Y；否则R1 =X+Y，并按以下规则确定商值q_n：

① 若R1与Y同号，则q_n置1，转下一步；

② 若R1与Y异号，则q_n置0，转下一步；

q_n用来判断是否溢出，而不是真正的商。以下情况下会发生溢出：

若X与Y同号且上商q_n=1，或者，若X与Y异号且上商q_n = 0。

(3) 对于i =1到n+1 ，按以下规则求出q_n和接下来的n位商（i=n+1时，只需置商）：

① 若Ri与Y同号，则q_{n+1-i}置1，Ri+1 = 2Ri –[Y]补，i = i +1；　同、1、减

② 若Ri与Y异号，则q_{n+1-i}置0，Ri+1 =2Ri+[Y]补，i = i +1；　异、0、加

(4) 商的修正：最后一次Q寄存器左移一位，将最高位q_n移出，最低位置上商q_0。若X与Y同号， Q中就是真正的商；否则，将Q中商的末位加1。　商已经是“反码”

(5) 余数的修正：若余数符号同X符号，则不需修正，余数在R中；否则，按下列规则进行修正：当X和Y符号相同时，最后余数加Y；否则，最后余数减Y。

其运算过程也呈加/减交替方式，因此也称为“加减交替法”。

举例：-9/2

将X=-9和Y=2分别表示成5位补码形式为：

[X]补 = 1 0111

[Y]补 = 0 0010

被除数符号扩展为：

[X]补=11111 10111

[–Y] 补 = 1 1110

同、1、减

异、0、加

X/Y= – 0100B = – 4

余数为 –0001B = –1

将各数代入公式：

“除数×商+余数= 被除数”进行验证，得

2×(–4) +(–1) = –9

除以2^k的快速处理——右移k位

- 无符号整数：逻辑右移，高位补0，低位丢弃
- 带符号整数：算术右移，高位补符，低位丢弃

举例：

unsigned 16/4=4 ： 0001 0000>>2 = 0000 0100

signed -16/4=-4 ： 1111 0000>>2 = 1111 1100

提醒：
N位被除数扩充为2N位，在高还是低位补？补0还是补符？
列竖式的时候N到底是几位（几位数值位，几位符号位）？

整数除法的近似处理（除以2^k）

- 不能整除时，采用朝零舍入，即截断方式
  - 无符号数、带符号正整数（地板）：移出的低位直接丢弃
  - 带符号负整数（天板）：加偏移量(2^k-1)，然后再右移k位，低位截断（这里K是右移位数）

举例：

无符号数　14/4=3：0000 1110>>2=0000 0011

带符号负整数　-14/4=-3

若直接截断，则 1111 0010 >>2=1111 1100=-4（错）

应先纠偏，再右移: k=2, 故$(-14+2^2-1)/4=-3$

即：　1111 0010+0000 0011=1111 0101

1111 0101>>2=1111 1101=-3

变量与常数之间的除运算—举例

- 假设*x*为一个int型变量，请给出一个用来计算*x*/32的值的函数div32。要求不能使用除法、乘法、模运算、比较运算、循环语句和条件语句，可以使用右移、加法以及任何按位运算。

解：若x为正数，则将x右移k位得到商；若x为负数，则x需要加一个偏移量(2^k-1)后再右移k位得到商。因为$32=2^5$，所以 k=5。

即结果为: (x>=0 ? (x + 0) : (x+31))>>5

但不能用比较和条件语句，因此要找一个计算偏移量b的方式

这里，x为正时b=0，x为负时b=31. 因此，可以从x的符号得到b

x>>31 得到的是32位符号，取出最低5位，就是偏移量b。

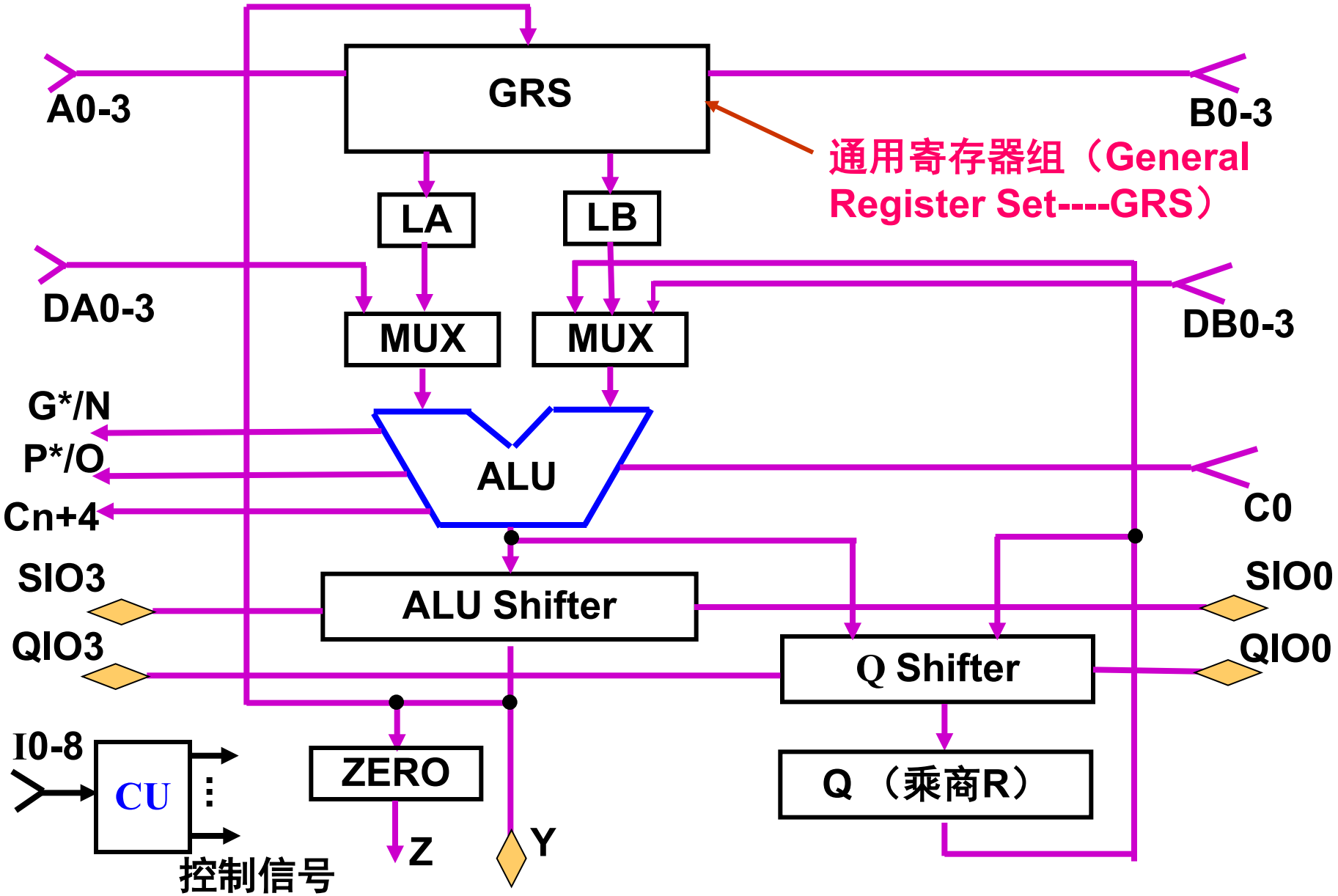
```
int div32(int x)
{  /* 根据x的符号得到偏移量b */
    int b=(x>>31) & 0x1F;
    return (x+b)>>5;
}
```

定点运算部件

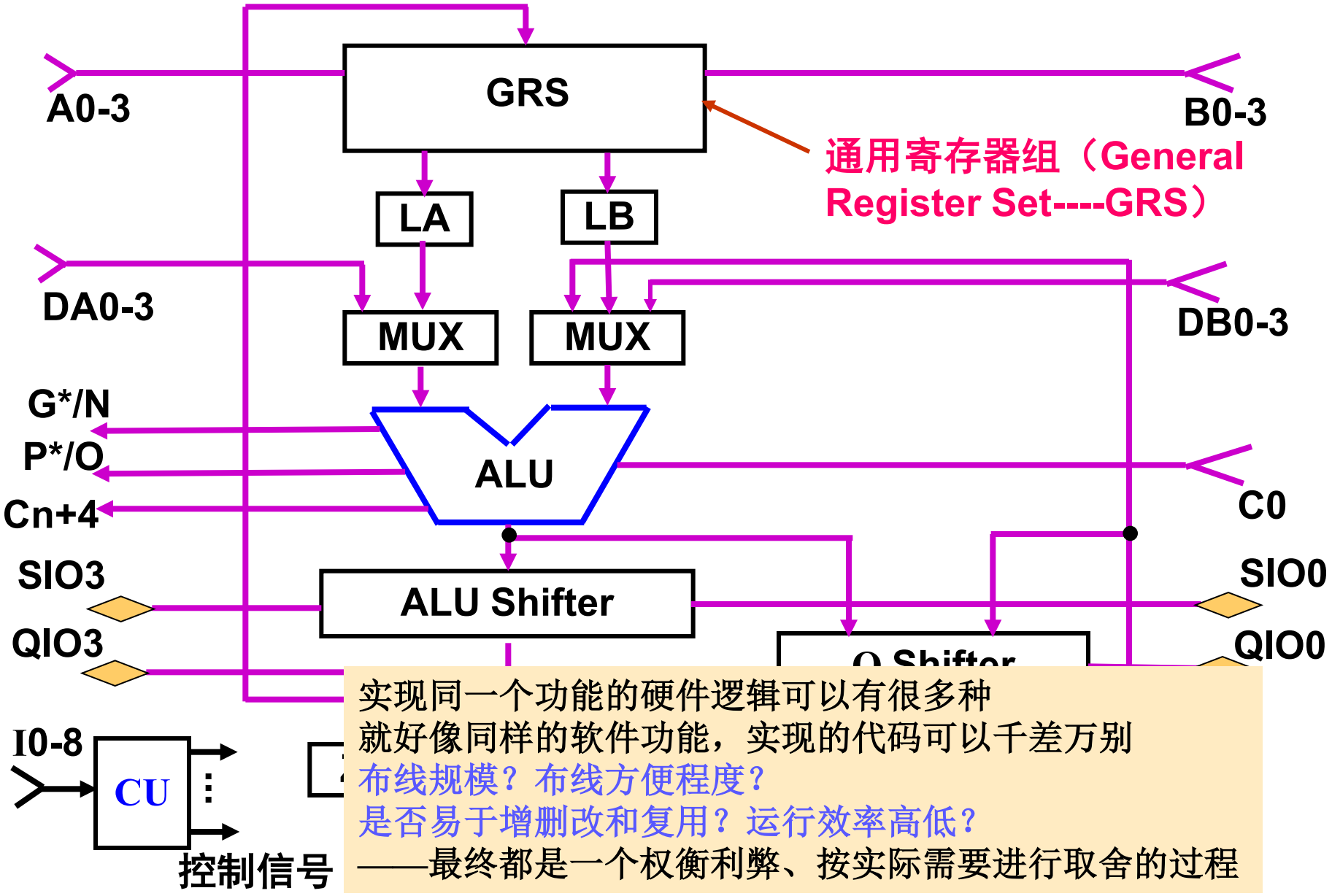
- **综合考虑各类定点运算算法后，发现：**
  - **所有运算都可通过“加”和“移位”操作实现**
- **以一个或多个ALU（或加法器）为核心，加上移位器、存放中间临时结果的寄存器组，在相应控制逻辑的控制下，通过多路选择器和实现数据传送的总线等，即可以实现各种运算——也就是构成了一个运算数据通路。**
  - 可用专门运算器芯片实现（如：4位运算器芯片AM2901）
  - 可用若干芯片级联实现（如4个AM2901构成16位运算器）
  - **现代计算机把运算数据通路和控制器都做在CPU中，为实现高级流水线，CPU中有多个运算部件，通常称为“功能部件”或“执行部件”。**

“运算器（Operate Unit）”、“运算部件（Operate Unit）”、“功能部件（Function Unit）”、“执行部件（Execution Unit）”和“数据通路（DataPath）”的含义基本上一样，只是强调的侧重不同

定点运算器芯片举例-AM2901A（不要求）



定点运算器芯片举例-AM2901A（不要求）



实现同一个功能的硬件逻辑可以有很多种
就好像同样的软件功能，实现的代码可以千差万别
布线规模？布线方便程度？
是否易于增删改和复用？运行效率高低？
——最终都是一个权衡利弊、按实际需要进行取舍的过程

RISC-V中整数的乘、除运算处理

ISA要素：指令+数据类型+寄存器设计等

- 乘法指令: mul，mulh，mulhu，mulhsu
  - mul rd, rs1, rs2：将低32位乘积存入结果寄存器rd
  - mulh、mulhu：将两个乘数同时按带符号整数（mulh）、同时按无符号整数（mulhu）相乘，高32位乘积存入rd中
  - mulhsu：将两个乘数分别作为带符号整数和无符号整数相乘后得到的高32位乘积存入rd中
  - 得到64位乘积需要两条连续的指令，其中一定有一条是mul指令，实际执行时只有一条指令
  - 两种乘法指令都不检测溢出，而是直接把结果写入结果寄存器。由软件根据结果寄存器的值自行判断和处理溢出
- 除法指令: div ，divu，rem，remu
  - div / rem：按带符号整数做除法，得到商 / 余数
  - divu / remu：按无符号整数做除法，得到商 / 余数

第二讲小结

逻辑运算、移位运算、扩展运算等电路简单

主要考虑算术运算

- **定点运算涉及的对象**

  无符号数；带符号整数(补码)；原码小数；移码整数

- **定点运算：(ALU实现基本算术和逻辑运算，ALU+移位器实现其他运算)**

  补码加/减：符号位和数值位一起运算，减法用加法实现。同号相加时可能溢出

  原码加/减：符号位和数值位分开运算，用于浮点数尾数加/减运算 （等到浮点数运算时介绍）

  移码加减：移码的和、差等于和、差的补码，用于浮点数阶码加/减运算（等到浮点数运算时介绍）

第二讲小结

乘法运算：

无符号数乘法：“加”+“右移”

原码（一位/两位）乘法：符号和数值分开运算，数值部分用无符号数乘法实现，用于浮点数尾数乘法运算。

补码（一位/两位）乘法：符号和数值一起运算，采用Booth算法。

快速乘法器：流水化乘法器、阵列乘法器

除法运算：

无符号数除法：用“加/减”+“左移”，有恢复余数和不恢复余数两种。

原码除法：符号和数值分开，数值部分用无符号数除法实现，用于浮点数尾数除法运算。

补码除法：符号位和数值位一起。有恢复余数和不恢复余数两种。

- **定点运算部件**
- 作业：习题3、4、5、6、7、11（4）（11月2号晚上24:00之前交）

(*) 第三讲：浮点数运算

主　要　内　容

- 指令集中与浮点运算相关的指令
  - 涉及到的操作数
    - 单精度浮点数
    - 双精度浮点数
  - 涉及到的运算
    - 算术运算： 加 / 减 / 乘 / 除
- 浮点数加减运算
- 浮点数乘除运算
- 浮点数运算的精度问题

有关Floating-point number的问题

实现一套浮点数运算指令，要解决的问题有：

- 编码表示：

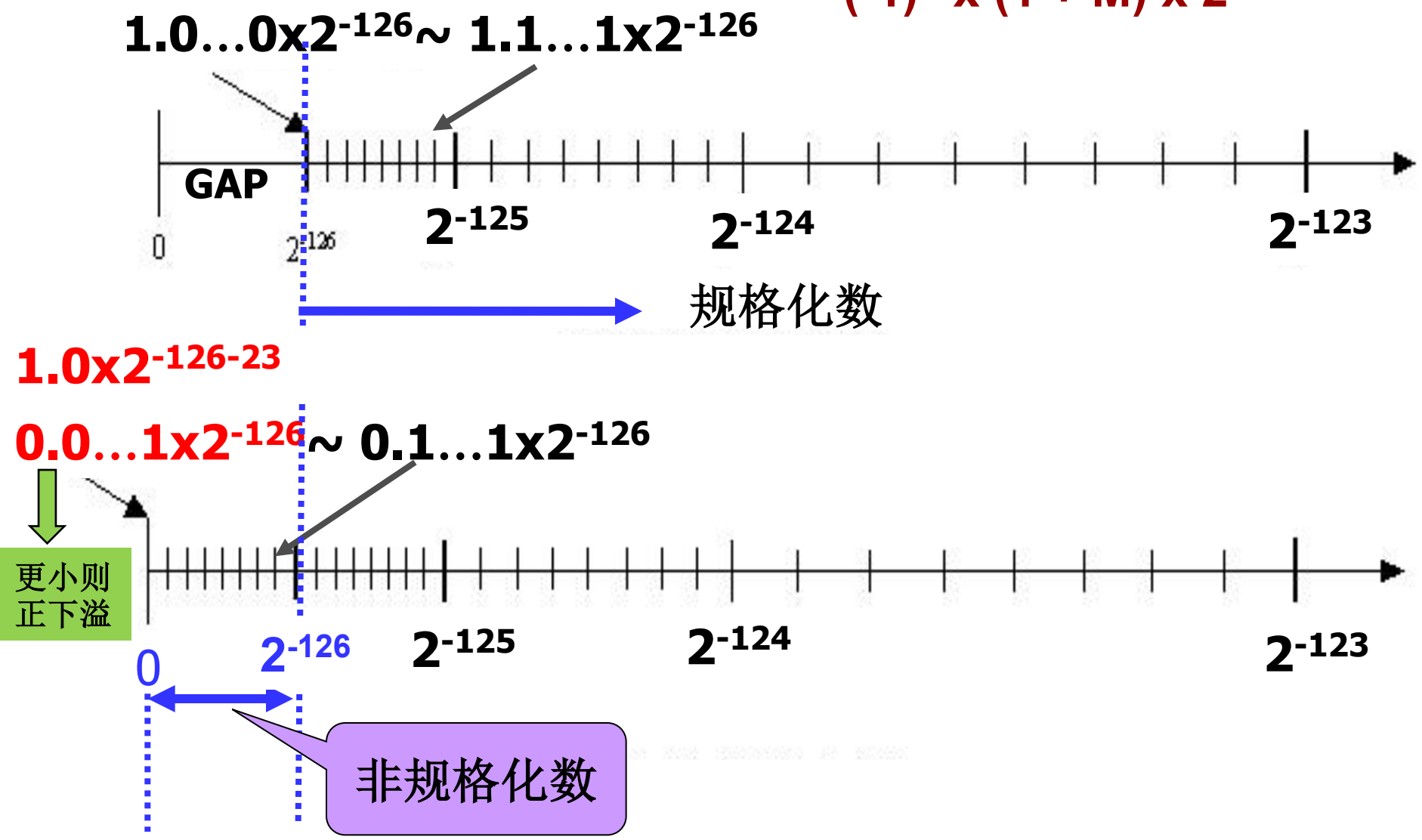
  Normalized form (规格化形式) 和 Denormalized form

  单精度格式 和 双精度格式

- 表数范围和精度
- 算术运算(+, -, *, /)
- 舍入 (Rounding)
- 异常处理： (Exceptions，如除数为0，上溢，下溢等)
- 误差与精度控制

回顾:单精度浮点数的表示

$(-1)^S \times (1 + M) \times 2^{(E-127)}$



浮点数运算及结果

设两个规格化浮点数分别为 $A=M_a \cdot 2^{Ea}$　$B=M_b \cdot 2^{Eb}$,则:

$A \pm B = (M_a \pm M_b \cdot 2^{-(Ea-Eb)}) \cdot 2^{Ea}$　(假设Ea>=Eb)

$A*B = (M_a * M_b) \cdot 2^{Ea+Eb}$

$A/B = (M_a / M_b) \cdot 2^{Ea-Eb}$

1.01+1.10=10.11

1.10-1.01=0.01

上述运算结果可能出现以下几种情况:

SP最大指数为多少? 127

阶码上溢: 一个正指数超过了最大允许值 =〉+∞/-∞/溢出

阶码下溢: 一个负指数比最小允许值还小 =〉+0/-0

SP最小指数呢?

-126-23

尾数溢出: 最高有效位有进位 =〉右规

非规格化尾数: 数值部分高位为0 =〉左规

右规或对阶时，右段有效位丢失 =〉尾数舍入

尾数溢出，结果不一定溢出

运算过程中添加保护位(附加位)

IEEE754标准规定的五种异常情况

① 无效运算（无意义）

- 运算时有一个数是非有限数，如：

  加 / 减∞、0 x ∞、 ∞/∞等

- 结果无效，如：

  源操作数是NaN、0/0、x REM 0、 ∞ REM y 等

② 除以0（即：无穷大）

③ 数太大（阶码上溢）：对于SP，阶码 E >1111 1110 (指数大于127)

④ 数太小（阶码下溢）：对于SP，阶码 E < 0000 0001(指数小于-126-23)

⑤ 结果不精确（舍入时引起），例如1/3，1/10等不能精确表示成浮点数

上述情况硬件可以捕捉到，因此这些异常可设定让硬件处理（硬件陷阱），也可设定让软件处理（调用特定的异常处理程序）。

浮点数加/减运算

- 十进制科学计数法的加法例子

$0.123 \times 10^5 + 0.560 \times 10^2$

其计算过程为：

$$0.123 \times 10^5 + 0.560 \times 10^2 = 0.123 \times 10^5 + 0.000560 \times 10^5$$
$$= (0.123 + 0.00056) \times 10^5 = 0.12356 \times 10^5$$
$$= 0.124 \times 10^5$$

（1）进行尾数加减运算前，必须“对阶”！
（2）最后还要考虑舍入
计算机内部的二进制运算也一样

- “对阶”操作：目的是使两数阶码相等
  - 小阶向大阶看齐，阶小的那个数的尾数右移，右移位数等于两个阶码差的绝对值（保证不会在对阶时溢出，但可能丢失数据位）
  - IEEE 754尾数右移时，要将隐含的“1”移到小数部分，高位补0，移出的低位保留到特定的“附加位”上

浮点数加/减运算–对阶

问题：如何对阶?

通过计算$[\Delta E]_{补}$来判断两数的阶差：

$[\Delta E]_{补} = [Ex-Ey]_{补} = [Ex]_{移} + [-[Ey]_{移}]_{补} \pmod{2^n}$

$[\Delta E]_{补} = 256+Ex-Ey = 256+127+Ex-(127+Ey)$

$= 256 + [Ex]_{移} - [Ey]_{移} = [Ex]_{移} + [-[Ey]_{移}]_{补} \pmod{256}$

问题：IEEE754 SP格式的偏置常数是127，这会不会影响阶码运算电路的复杂度? 对计算$[Ex-Ey]_{补} \pmod{2^n}$ 没有影响

但$[Ex+Ey]_{移}$和 $[Ex-Ey]_{移}$的计算会变复杂！ 浮点乘除运算涉及之。

问题：在ΔE为何值时无法根据$[\Delta E]_{补}$来判断阶差? **溢出时!**

例：4位移码，Ex=7，Ey=-7，则$[\Delta E]_{补}$=1111+1111=1110，ΔE<0,错

问题：对IEEE754 SP格式来说，|ΔE|大于多少时，结果就等于阶大的那个数（即小数被大数吃掉）？ **24!**

1.xx…x → 0.00…01xx…x(右移24位后，尾数变为0)

求阶码的和、差（浮点数乘除法中使用）

设E1和E2分别是两个操作数的阶码（移码），Ex和Ey是对应指数

Eb是结果的阶码（移码），则：

- 阶码加法公式为： $Eb \leftarrow E1+E2+129 \ (\mathrm{mod}\ 2^8)$

$$[Ex+Ey]_{移} = 127 + Ex + Ey = 127 + Ex + 127 + Ey - 127$$

$$= [Ex]_{移} + [Ey]_{移} - 127$$

$$= [Ex]_{移} + [Ey]_{移} + [-127]_{补}$$

$$= [Ex]_{移} + [Ey]_{移} + 10000001B \ (\mathrm{mod}\ 2^8)$$

- 阶码减法公式为： $Eb \leftarrow E1+[-E2]_{补}+127 \ (\mathrm{mod}\ 2^8)$

$$[Ex-Ey]_{移} = 127 + Ex - Ey = 127+Ex-(127+Ey)+127$$

$$= [Ex]_{移}-[Ey]_{移} + 127$$

$$= [Ex]_{移}+\boxed{[-[Ey]_{移}]_{补}}+01111111B \ (\mathrm{mod}\ 2^8)$$

把E2按位取反，末尾加1

浮点数加减法基本要点（1）
（规格化数）

如果参与运算的有规格化数和非规格化数，对阶计算还需修改

Xm、Ym分别是X和Y的尾数， Xe和Ye 分别是X和Y的阶码

(1)求阶差：Δe=Ye – Xe（假设Ye > Xe)

求$[\Delta E]_{补}$ 此时，和(差)的初始阶码为Ye

(2) 对阶：将Xm右移Δe位，尾数变为 $Xm*2^{Xe-Ye}$

隐含位右移到数值部分，高位补0
保留移出低位部分（附加位）

(3) 尾数加减： $Xm*2^{Xe-Ye} \pm Ym$

隐藏位还原后，按原码进行加减运算，附加位一起运算

浮点数加减法基本要点（2）
（规格化数）

(4) 规格化：

± 0.0…01x…x 形式

当尾数高位为0，则需左规：尾数左移一次，阶码减1，直到MSB为1或阶码为00000000（-126，非规格化数）

每次阶码减1后要判断阶码是否下溢（比最小可表示的阶码还要小）

±1x .xx……x 形式

当尾数最高位有进位，需右规：尾数右移一次，阶码加1，直到MSB为1

每次阶码加1后要判断阶码是否上溢（比最大可表示的阶码还要大）

注意：IEEE 754 加减运算右规时最多只需一次

—— 即使是两个最大的尾数相加，得到的和的尾数最多是11.xxx

浮点数加减法基本要点（3）
（规格化数）

(5) 如果尾数比规定位数长（有附加位），则需考虑舍入（有多种舍入方式）如果“入”导致需要右规，则也要在右规前后判断阶码是否上溢。

(6) 若运算结果尾数是0，则需要将阶码也置0。为什么?

尾数为0说明结果应该为0（阶码和尾数为全0）。

阶码上溢（全1 ），则结果溢出（无穷大）

阶码下溢到无法用非规格化数表示，则结果为0

若尾数为全0，则下溢，结果为0

（左规、右规、舍入时都需要判断溢出与否）

IEEE 754 浮点数加法运算举例

已知x=0.5, y=-0.4375, 求x+y=? (用IEEE754标准单精度格式计算)

解: $x=0.5=1/2=(0.100...0)_2=(1.00...0)_2x2^{-1}$

$[x]_{浮}$=0 01111110, 000…0

$y=-0.4325=(-0.01110...0)_2=(-1.110..0)_2x2^{-2}$

$[y]_{浮}$=1 01111101, 110…0

对阶: $\Delta E=[x]_{阶}-[y]_{阶}=1$，结果的初始指数为-1（较大的那个）

故对y进行对阶：$[y]_{尾}$=0.1110…0 00(隐藏位右移,最后增加附加位)

$[y]_{阶}$ 从 0111 1101→0111 1110

尾数相加：01.0000...0 (原码加法，最左边一位为符号位，符号位分开处理)

+ 10.1110...0 00 = 00.00100…0 00

左规三次： $+(0.00100…0\ 00)_2x2^{-1}=+(1.00…000)_2x2^{-4}$ (阶码减3，实际上是加了三次11111111)

$[x+y]_{浮}$=0 01111011(-4+127) 00…0(去除隐藏位)

$x+y=(1.0)_2x2^{-4}=1/16=0.0625$

问题：尾数加法器应该是多少位?

考虑符号位、隐藏位、附加位，以及23位尾数

Extra Bits(附加位)

加多少附加位才合适? 无法给出准确的答案!

IEEE754规定: 中间结果须在右边加2个附加位 (guard & round)

Guard (保护位): 在尾数最右边的位

Round (舍入位): 在保护位右边的位

附加位的作用:

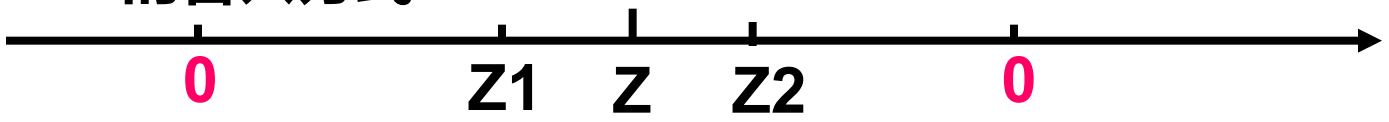
用以保护对阶时右移的位或运算的中间结果。

附加位的处理:

①左规时被移到尾数中; ② 作为舍入的依据。

IEEE 754的舍入方式的说明

IEEE 754的舍入方式



（Z1和Z2分别是结果Z的最近的可表示的左、右两个数）

(1) 就近舍入：舍入为最近可表示的数

非中间值：0舍1入；

中间值：强迫结果为偶数-慢

例如：附加位为
01：舍
11：入
10：(强迫结果为偶数)

例：1.1101**11** → 1.1110;　1.1101**01** → 1.1101;
1.1101**10** → 1.1110;　1.1111**10** → 10.0000;

(2) 朝+∞方向舍入:舍入为Z2(正向舍入)

(3) 朝-∞方向舍入:舍入为Z1(负向舍入)

(4) 朝0方向舍入：截去。正数：取Z1；负数：取Z2

舍入位和粘位的作用

IEEE 754通过在舍入位后再引入“粘位sticky bit”增强精度

加减运算对阶过程中，若阶码较小的数的尾数右移时，舍入位之后有非0数，则可设置sticky bit。

举例：

$1.110 \times 2^5 + 1.010 \times 2^1$ 分别采用二位、三位附加位时，结果各是多少？（就近舍入到偶数）

尾数精确结果为1.110101, 所以分别为：

1.110，1.111（误差较小）

原码加/减运算

- 用于浮点数尾数运算
- 符号位和数值部分分开处理
- 仅对数值部分进行加减运算，符号位起判断和控制作用
- 规则如下：
  - 比较两数符号，对加法实行“同号求和，异号求差”，对减法实行“异号求和，同号求差”。
  - 求和：数值位相加，和的符号取被加数（被减数）的符号。若最高位产生进位，则结果溢出。
  - 求差：被加数（被减数）加上加数（减数）的补码。

    a) 最高数值位产生进位表明加法结果为正，所得数值位正确。

    b) 最高数值位没产生进位表明加法结果为负，得到的是数值位的补码形式，需对结果求补，还原为绝对值形式的数值位。
  - 差的符号位：a)情况下，符号位取被加数（被减数）的符号；

    b)情况下，符号位为被加数（被减数）的符号取反。

原码加/减运算

例1：已知 $[X]_{原} = 1.0011$，$[Y]_{原} = 1.1010$，要求计算$[X+Y]_{原}$

解：由原码加减运算规则知：同号相加，则求和，和的符号同被加数符号。

和的数值位为：0011 + 1010 = 1101　（ALU中无符号数相加）

和的符号位为：1

$[X+Y]_{原} = 1.1101$

例2：已知 $[X]_{原} = 1.0011$，$[Y]_{原} = 1.1010$，要求计算$[X-Y]_{原}$

解：由原码加减运算规则知：同号相减，则求差（补码减法）

差的数值位为：$0011+(1010)_{求补} = 0011 + 0110 = 1001$

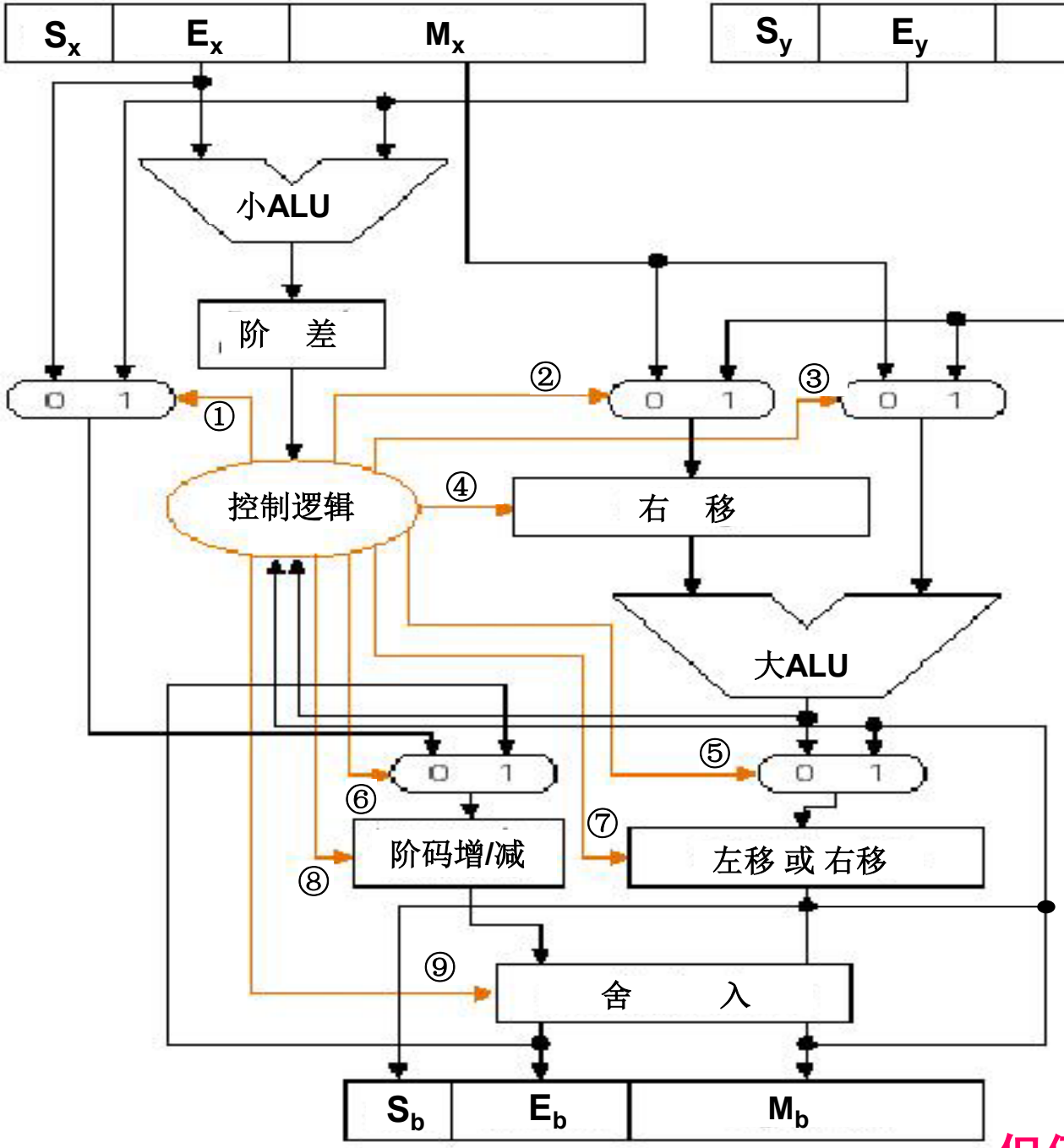
最高数值位没有产生进位，表明加法结果为负，需对1001求补，还原为绝对值形式的数值位。即：$(1001)_{补} = 0111$

差的符号位为$[X]_{原}$的符号位取反，即：0

$[X-Y]_{原} = 0.0111$

求差：加补码，不会溢出，符号分情况

思考：如何设计一个基于加法器的原码加/减法器?



阶码相减

阶小的数的尾数右移

尾数加/减

规格化

舍入

可用流水线方式实现！

但仍然比整数运算更耗时！

浮点数乘/除法基本要点

- 浮点数乘法：$A*B=(M_a * M_b)\cdot 2^{Ea+Eb}$
- 浮点数除法：$A/B=(M_a / M_b)\cdot 2^{Ea-Eb}$

X的平方不会是负数！

浮点数乘 / 除法步骤

（Xm、Ym分别是X和Y尾数原码， Xe和Ye 分别是X和Y阶移码 ）

(1) 求阶： Xe ± Ye ∓ 127

(2) 尾数相乘除： Xm */Ym （两个形为1.xxx的数相乘/除） 原码乘/除

(3) 两数符号相同，结果为正；两数符号相异，结果为负；

(4) 当尾数高位为0，需左规；当尾数最高位有进位，需右规。

(5) 如果尾数比规定的长，则需考虑舍入。

(6) 若尾数是0，则需要将阶码也置0。

(7) 阶码溢出判断

如果是其它浮点数编码，可能需要右规：0.11110000/0.1000=+1.1110

问题1：IEEE754浮点数乘法运算结果最多左规几次？最多右规几次？ 不需左规！最多右规1次！

问题2：除法呢？ 左规次数不定！不需右规！

浮点数乘除法结果溢出判断

以下情况下，可能会导致阶码溢出（续）

- 乘法运算求阶码的和时
  - 若Ex和Ey最高位皆1，而Eb最高位是0或Eb为全1，则阶码上溢
  - 若Ex和Ey最高位皆0，而Eb最高位是1或Eb为全0，则阶码下溢
- 除法运算求阶码的差时
  - 若Ex的最高位是1，Ey的最高位是0，Eb的最高位是0或Eb为全1，则阶码上溢。
  - 若Ex的最高位是0，Ey的最高位是1，Eb的最高位是1或Eb为全0，则阶码下溢。

溢出判断（续）

以下情况也要判断阶码是否溢出

- **左规（阶码 - 1）时**
  - **左规（- 1）时：先判断阶码是否为全0，若是，则直接置阶码下溢；否则，阶码减1后判断阶码是否为全0，若是，则阶码下溢。**
- **右规（阶码 +1）时（包括：舍入的“入”导致需要右规）**
  - **右规（+ 1）时，先判断阶码是否为全1，若是，则直接置阶码上溢；否则，阶码加1后判断阶码是否为全1，若是，则阶码上溢。**

问题：机器内部如何减1？ $+[-1]_{补} = +11\ldots1$

比整数运算更耗时！

浮点数除0的问题

```
#include <conio.h>
#include <stdio.h>
int    main()
{
        int    a=1,b=0;
        printf( "Division    by    zero:%d\n ",a/b);
        getchar();
        return    0;
}
int    main()
{
        double    x=1.0,y=-1.0,z=0.0;
        printf( "division    by    zero:%f    %f\n ",x/z,y/z);
        getchar();
        return    0;
}
```

这是网上的一个帖子

为什么整数除0会发生异常?

为什么浮点数除0不会出现异常?

浮点运算中，一个有限数除以0，结果为正无穷大（负无穷大）

问题一:为什么整除int型会产生错误? 是什么错误?

二:用double型的时候结果为1.#INF00和-1.#INF00，作何解释???

C语言中的浮点数类型

- C语言中有float和double类型，分别对应IEEE 754单精度浮点数格式和双精度浮点数格式
- long double类型的长度和格式随编译器和处理器类型的不同而有所不同，IA-32中是80位扩展精度格式
- 从int转换为float时，不会发生溢出，但可能有数据被舍入
- 从int或 float转换为double时，因为double的有效位数更多，故能保留精确值
- 从double转换为float和int时，可能发生溢出，此外，由于有效位数变少，故可能被舍入
- 从float 或double转换为int时，因为int没有小数部分，所以数据可能会向0方向被截断

回顾：IEEE 754表示的一些问题

- 表数范围?

  单精度可表示最大正数: +1.11…1x 2^{127}　　约 +3.4 x 10^{38}

  双精度呢?　　约 +1.8 x 10^{308}

- 数据转换时可能发生的问题？i是32位补码，f是float，d是double

  i 和 (int) ((float) i))　　不一定相等

  i 和 (int) ((double) i))　　相等

  f 和 (float) ((int) f))　　不一定相等

  d 和 (double) ((int) d))　不一定相等

- FP参与加法时的不同计算顺序可能带来的问题？

  x = – 1.5 x 10^{38},　y = 1.5 x 10^{38},　z = 1.0

  (x+y)+z = (–1.5x10^{38}+1.5x10^{38}) +1.0 = 1.0

  x+(y+z) = –1.5x10^{38}+ (1.5x10^{38}+1.0) = 0.0

举例：Ariana火箭爆炸

- 1996年6月4日，Ariana 5火箭初次航行，在发射仅仅37秒钟后，偏离了飞行路线，然后解体爆炸，火箭上载有价值5亿美元的通信卫星。 最大约 $+1.8 \times 10^{308}$ 最大32767
- 原因是**在将一个64位浮点数转换为16位带符号整数时，产生了溢出异常。**溢出的值是火箭的水平速率，这比原来的Ariana 4火箭所能达到的速率高出了5倍。在设计Ariana 4火箭软件时，设计者确认水平速率决不会超出一个16位的整数，但在设计Ariana 5时，他们没有重新检查这部分，而是直接使用了原来的设计。
- **在不同数据类型之间转换时，往往隐藏着一些不容易被察觉的错误**，这种错误有时会带来重大损失，因此，编程时要非常小心。

举例：爱国者导弹定位错误

- 1991年2月25日，海湾战争中，美国在沙特阿拉伯达摩地区设置的爱国者导弹拦截伊拉克的飞毛腿导弹失败，致使飞毛腿导弹击中了一个美军军营，杀死了美军28名士兵。其原因是由于爱国者导弹系统时钟内的一个软件错误造成的，引起这个软件错误的原因是浮点数的精度问题。
- 爱国者导弹系统中有一内置时钟，用计数器实现，每隔0.1秒计数一次。程序用0.1(表示为24位定点二进制小数x）来乘以计数值作为以秒为单位的时间
- 这个x的机器数是多少呢?
- 0.1的二进制表示是一个无限循环序列：0.00011[0011]...，x=0.000 1100 1100 1100 1100 1100B。显然，x取24位后，和0.1之间的误差为：

  = 0.000 1100 1100 1100 1100 1100 [1100]... -

  0.000 1100 1100 1100 1100 1100B，即为：

  =0.000 0000 0000 0000 0000 0000 1100 [1100]...B

  $=2^{-20}\times 0.1 \approx 9.54\times 10^{-8}$

  这就是机器值与真值之间的误差!

举例：爱国者导弹定位错误

已知在爱国者导弹准备拦截飞毛腿导弹之前，已经连续工作了100小时，相当于计数了$100\times60\times60\times10=36\times10^5$次，

因而导弹的时钟已经偏差了$9.54\times10^{-8}\times36\times10^5\approx0.343$秒

飞毛腿的速度大约为2000米/秒，则由于时钟计算误差而导致的距离误差是2000×0.343秒≈687米

小故事：实际上，以色列方面已经发现了这个问题并于1991年2月11日知会了美国陆军及爱国者计划办公室（软件制造商）。**以色列方面建议重新启动爱国者系统的电脑作为暂时解决方案，可是美国陆军方面却不知道每次需要间隔多少时间重新启动系统一次。**1991年2月16日，制造商向美国陆军提供了更新软件，但这个软件最终却在飞毛腿导弹击中军营后的一天才运抵部队。

举例：爱国者导弹定位错误

- 若x用float型表示，0.1= 0.0 0011[0011]B=+1.1 0011 0011 0011 0011 0011 00B×2^{-4}，（比前面多保留了一组“1100”，因为规格化带来的好处）
  - 故x的机器数为0 011 1101 1 100 1100 1100 1100 1100 1100
- 0.1与x的偏差是|x–0.1|= 0.000 0000 0000 0000 0000 0000 0000 1100 [1100]...B= 2^{-24}×0.1 ≈ 5.96×10^{-9} 。
- 系统运行100小时后的时钟偏差是$5.96\times10^{-9}\times36\times10^{5}$ ≈ 0.0215秒
- 在飞毛腿速度为2000米/秒的情况下，预测的距离偏差为0.0215×2000≈43米。比爱国者导弹系统精确约16倍。
- 如果用double呢?

总结：浮点数运算的精度问题

- ✓ 程序员应对底层机器级数据的表示和运算有深刻理解
- ✓ 计算机世界里，经常是“差之毫厘，失之千里”，需要细心再细心，精确再精确
- ✓ 不能遇到小数就用浮点数表示，有些情况下（如需要将一个整数变量乘以一个确定的小数常量），可先用一个确定的定点整数与整数变量相乘，然后再通过移位运算来确定小数点

第三讲小结

- 浮点数的表示（IEEE754标准）
  - 单精度SP（float）和双精度DP（double）
    - 规格化数(SP)：阶码1~254，尾数最高位隐含为1
    - 0(阶为全0，尾为全0)
    - ∞(阶为全1，尾为全0)
    - NaN(阶为全0，尾为非0)
    - 非规数(阶为全1，尾为非0)
- 浮点数加减运算
  - 对阶、尾数加减、规格化（上溢/下溢处理）、舍入
- 浮点数乘除运算
  - 求阶、尾数乘除、规格化（上溢/下溢处理）、舍入
- 浮点数的精度问题
  - 中间结果加保护位、舍入位（和粘位）
  - 最终进行舍入（有四种舍入方式）
    - 就近（中间值强迫为偶数）、+∞方向、-∞方向、0方向
    - 默认为“就近”舍入方式

本章总结（1）

定点数运算：由ALU + 移位器实现各种定点运算

- 移位运算
  - 逻辑移位：对无符号数进行，左（右）边补0，低（高）位移出
  - 算术移位：对带符号整数进行，移位前后符号位不变，编码不同，方式不同。
  - 循环移位：最左（右）边位移到最低（高）位，其他位左（右）移一位。
- 扩展运算
  - 零扩展：对无符号整数进行高位补0
  - 符号扩展：对补码整数在高位直接补符
- 加减运算
  - 补码加/减运算：用于整数加/减运算。符号位和数值位一起运算，减法用加法实现。同号相加时，若结果的符号不同于加数的符号，则会发生溢出。
  - 原码加/减运算：用于浮点数尾数加/减运算。符号位和数值位分开运算，同号相加，异号相减；加法直接加；减法用加负数补码实现。
- 乘法运算：用加法和右移实现。
  - 补码乘法：用于整数乘法运算。符号位和数值位一起运算。采用Booth算法。
  - 原码乘法：用于浮点数尾数乘法运算。符号位和数值位分开运算。数值部分用无符号数乘法实现。
- 除法运算：用加/减法和左移实现。
  - 补码除法：用于整数除法运算。符号位和数值位一起运算。
  - 原码除法：用于浮点数尾数除法运算。符号位和数值位分开运算。数值部分用无符号数除法实现。

本章总结（2）

- 浮点数运算：由多个ALU + 移位器实现
  - 加减运算
    - 对阶 、尾数相加减、规格化处理、舍入、判断溢出
  - 乘除运算
    - 尾数用定点原码乘/除运算实现，阶码用定点数加/减运算实现。
  - 溢出判断
    - 当结果发生阶码上溢时，结果发生溢出，发生阶码下溢时，结果为0。
  - 精确表示运算结果
    - 中间结果增设保护位、舍入位、粘位
    - 最终结果舍入方式：就近舍入 / 正向舍入 / 负向舍入 / 截去四种方式。
- ALU的实现
  - 算术逻辑单元ALU：实现基本的加减运算和逻辑运算。
  - 加法运算是所有定点和浮点运算（加/减/乘/除）的基础，加法速度至关重要
  - 进位方式是影响加法速度的重要因素
  - 并行进位方式能加快加法速度
  - 通过“进位生成”和“进位传递”函数来使各进位独立、并行产生
- 作业：习题3、4、5、6、7、11（4）(11月2号晚上24:00之前交)