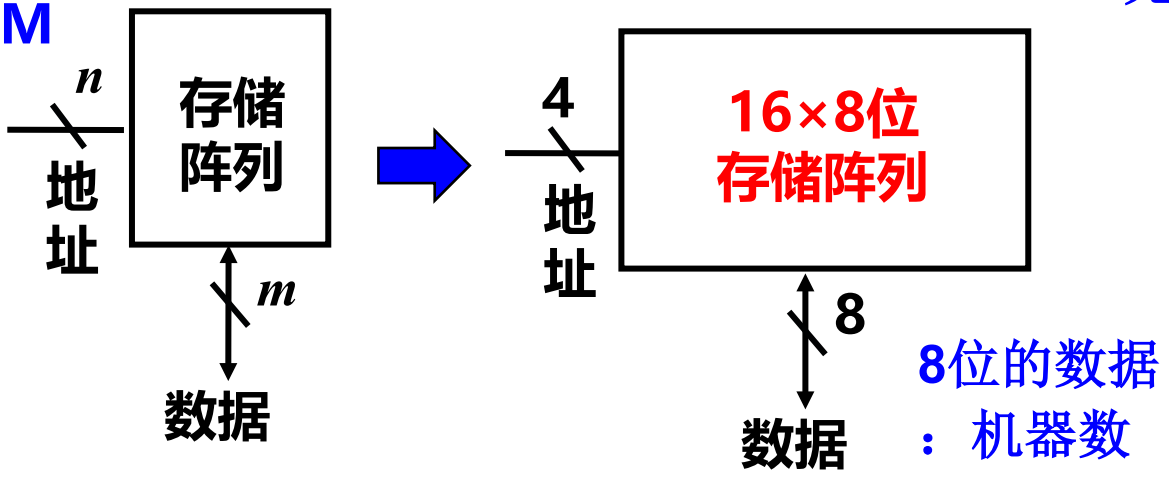
回顾第12次课

- ROM和RAM
- 存储器



4位的地址：无符号数

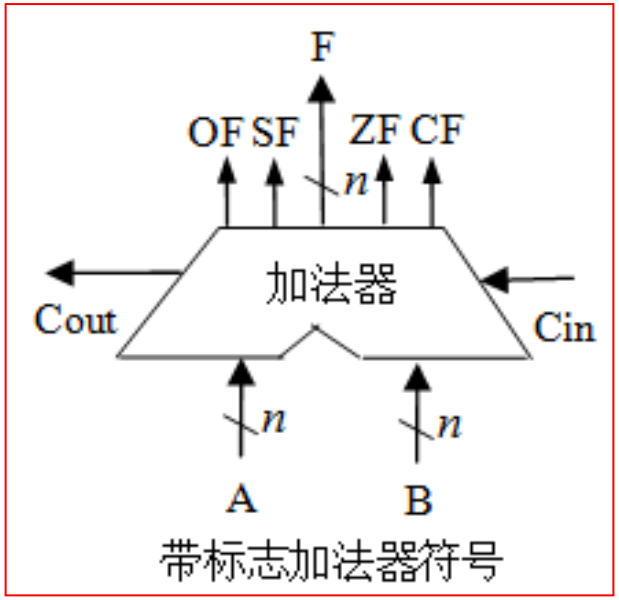
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

8位的数据：机器数

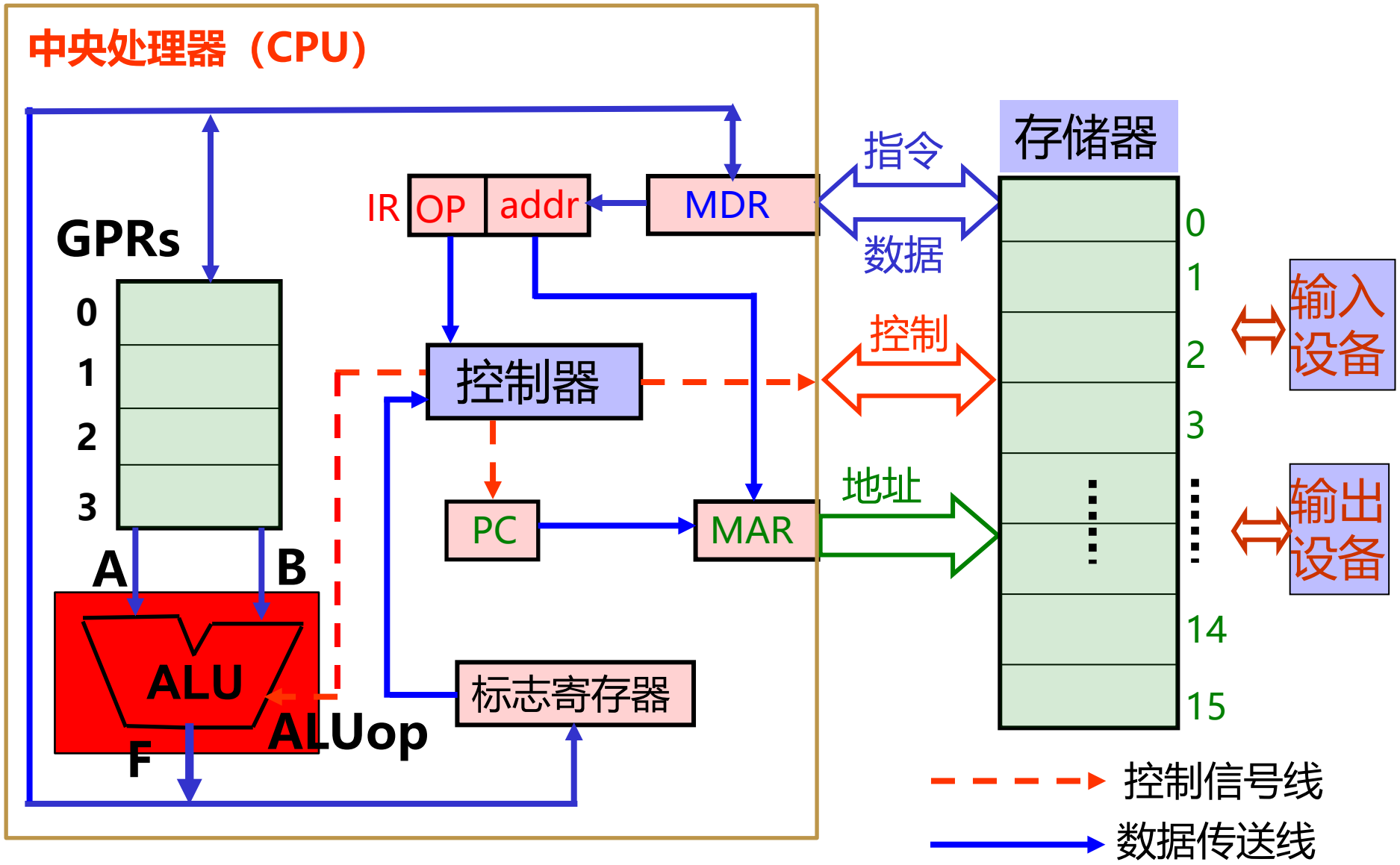
- 高级语言程序里的运算
- → ISA（例如：RISCV指令）
- → 硬件功能部件

- 串行进位
- 先行进位
- 带标志的加法器

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

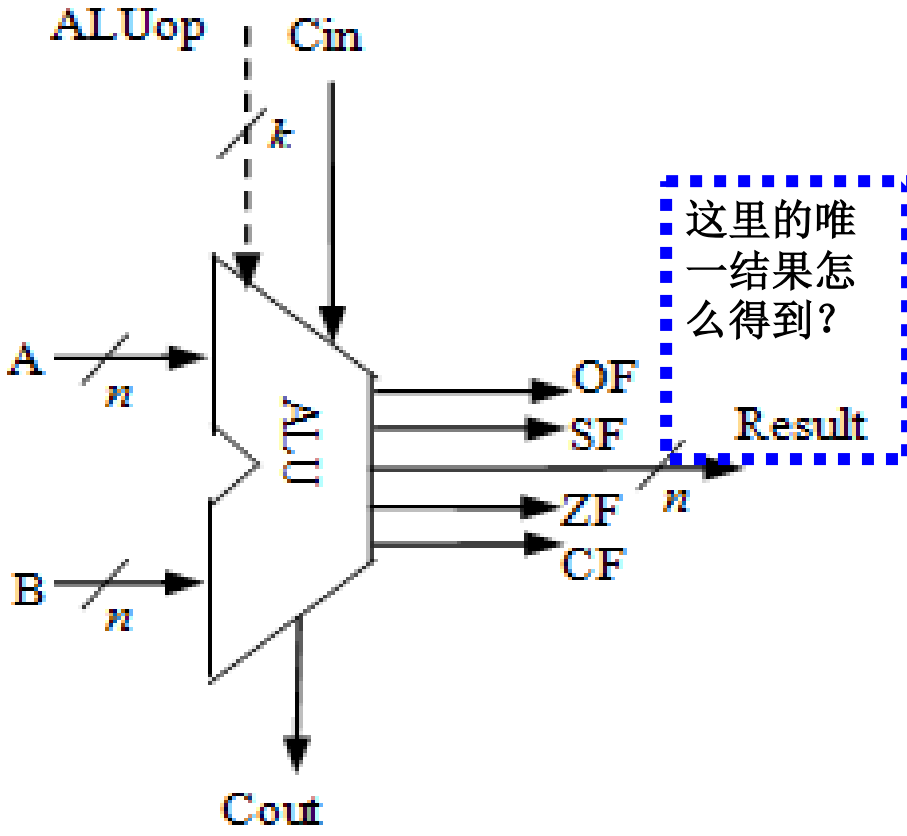


CPU：中央处理器；PC：程序计数器；MAR：存储器地址寄存器
ALU：算术逻辑部件；IR：指令寄存器；MDR：存储器数据寄存器
GPRs：通用寄存器组（由若干通用寄存器组成）



算术逻辑部件（ALU）

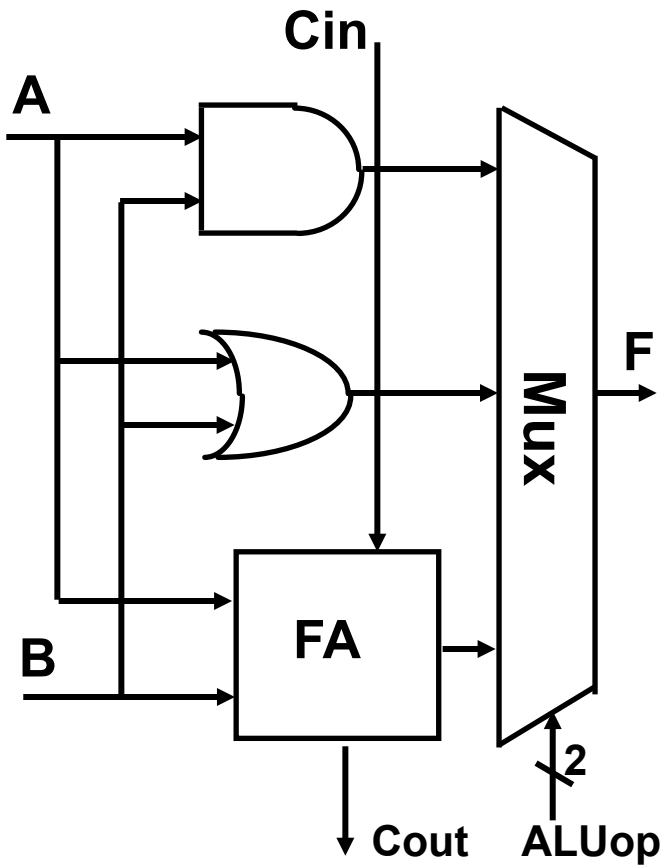
- 有一个**操作控制端**（ALUop），用来决定ALU所执行的处理功能。ALUop的位数k决定了操作的种类例如，当位数k为3时，ALU最多只有2^3=8种操作。
- 进行**基本**算术运算与逻辑运算
  - 无符号整数加、减
  - 带符号整数加、减
  - 与、或、非、异或等逻辑运算
- 核心电路是**整数加/减运算部件**
- 输出除**和/差等**，还有**标志信息**



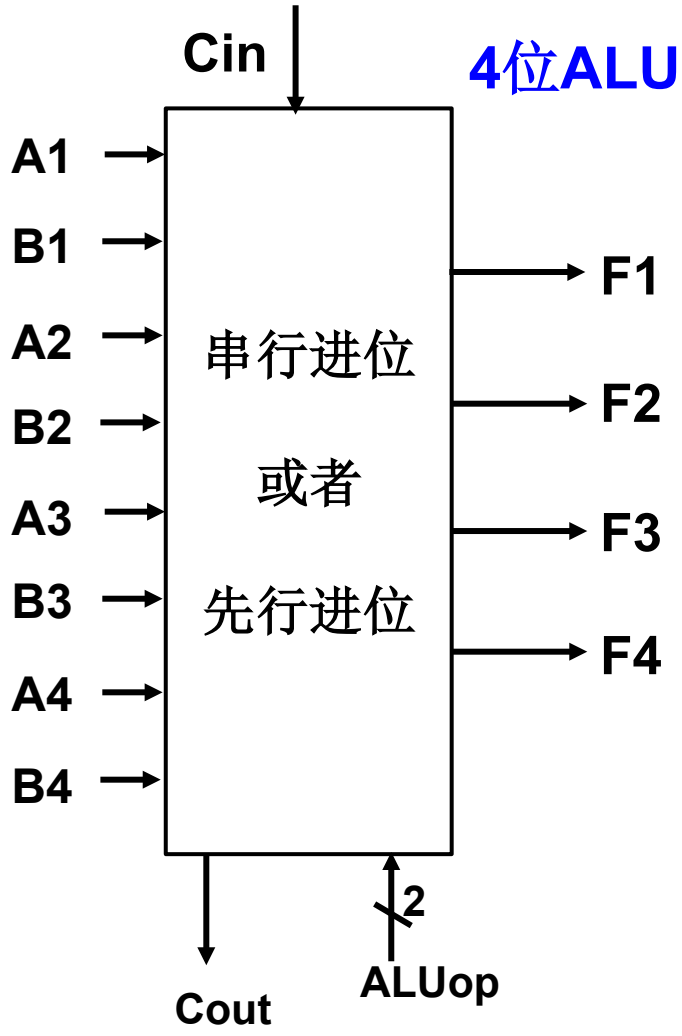
ALU 符号

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）

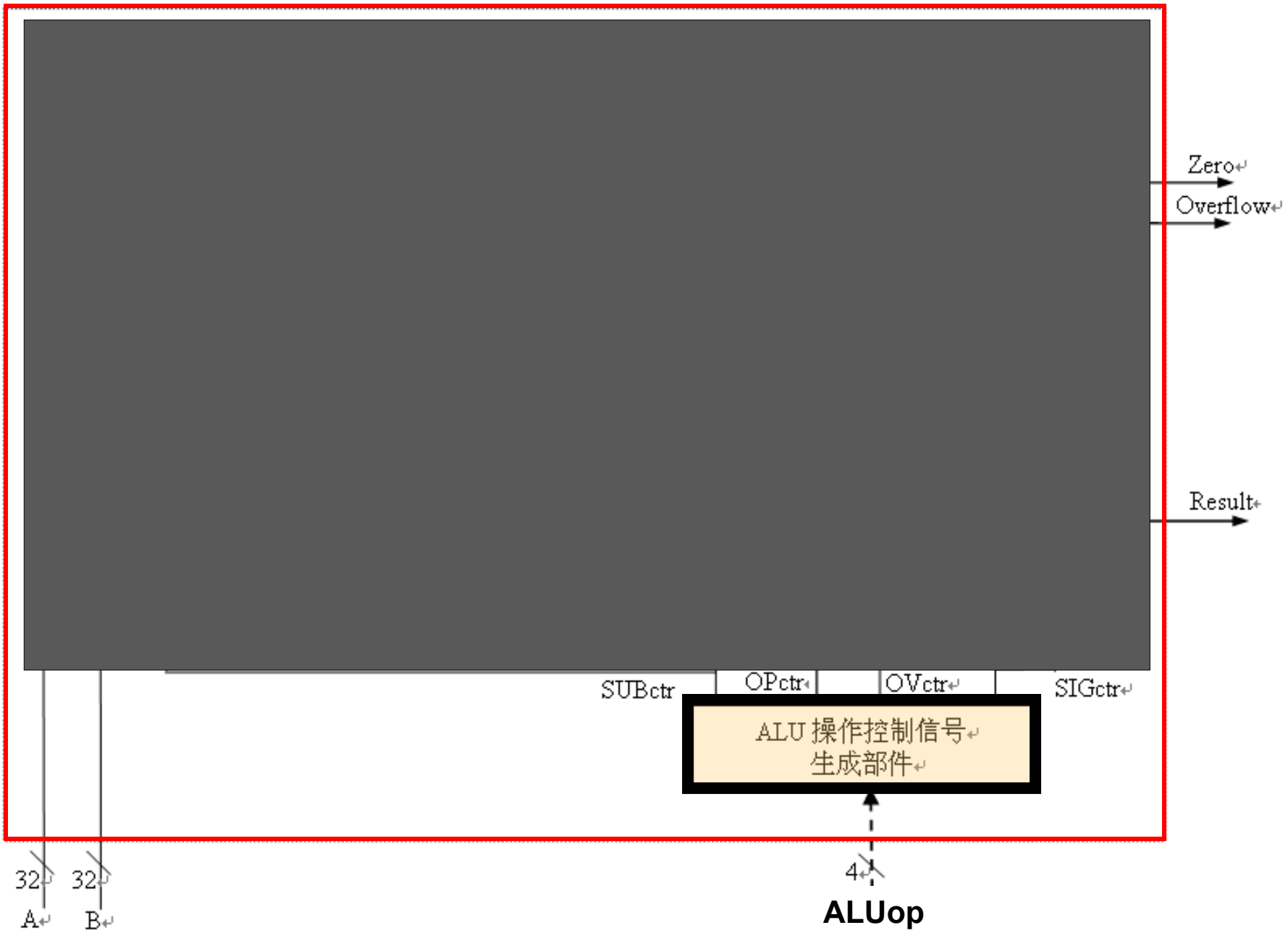


1-bit ALU



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

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：

$[X]_{补}=2^n+X \quad (-2^n \le X < 2^n, \mathrm{mod}\ 2^n)$

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：

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

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

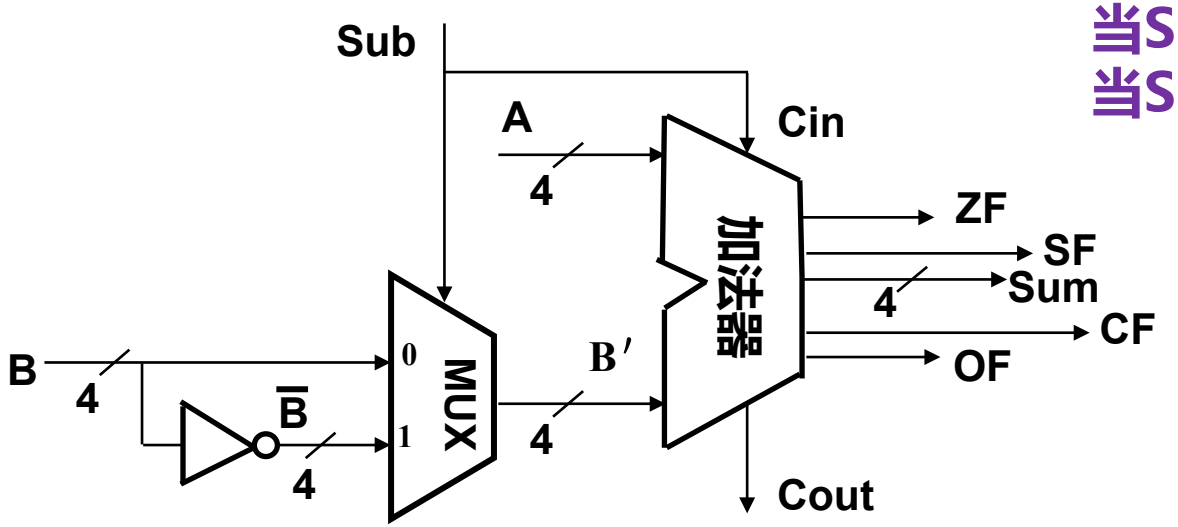
n位整数加/减运算器

- **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 $[-B]_{补} = \overline{[B]_{补}} + 1$



整数加/减运算部件

当Sub为1时，做减法
当Sub为0时，做加法

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

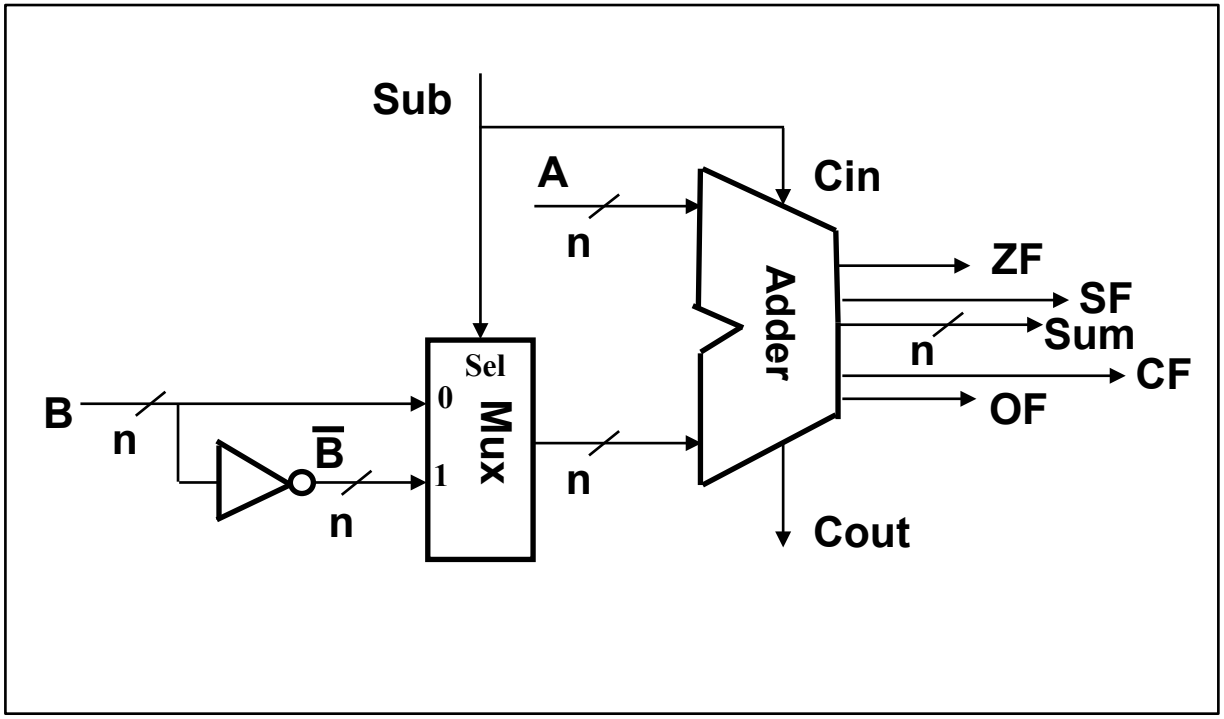
强调：带符号数、无符号数的加减法运算

- 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　　-3 - 5 = - 3 + (- 5) = - 8 √

unsigned　9 - 6 = 9 + (-6) = 3 √　　13 - 5 = 13 + (- 5) = 8 √

(Cout=1)

```
  1   0
      1   0   0   1
+     1   0   1   0
  ---------------------
      0   0   1   1
```

OF=1、ZF=0
SF=0、借位CF=0

(Cout=1)

```
  1   1   1   1
      1   1   0   1
+     1   0   1   1
  ---------------------
  1   0   0   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

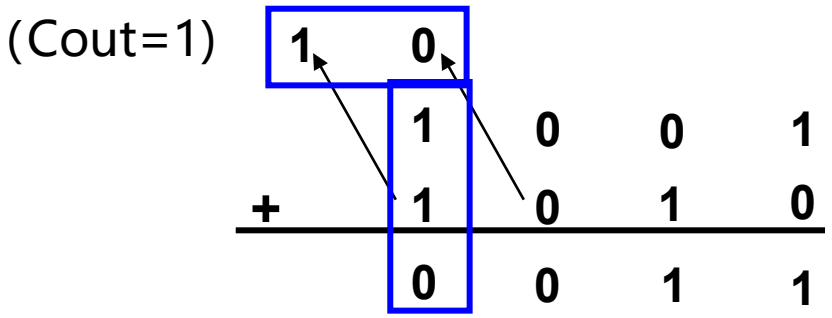
验证：-7<6，故OF≠SF
　　-3<5，故OF≠SF

无符号数减法例

注意：Cin=sub=1

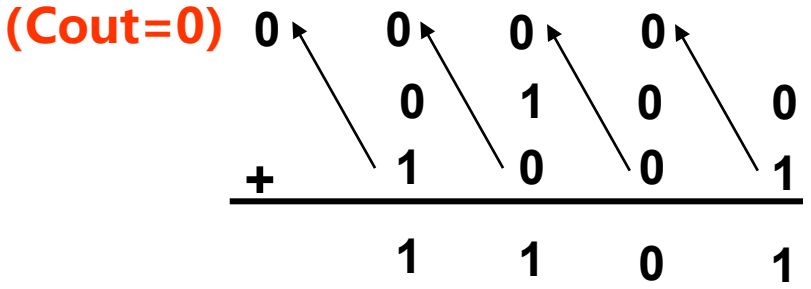
$OF=C_n \oplus C_{n-1}$
$CF=Cout \oplus Cin$

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）

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：

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

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：

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

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
```

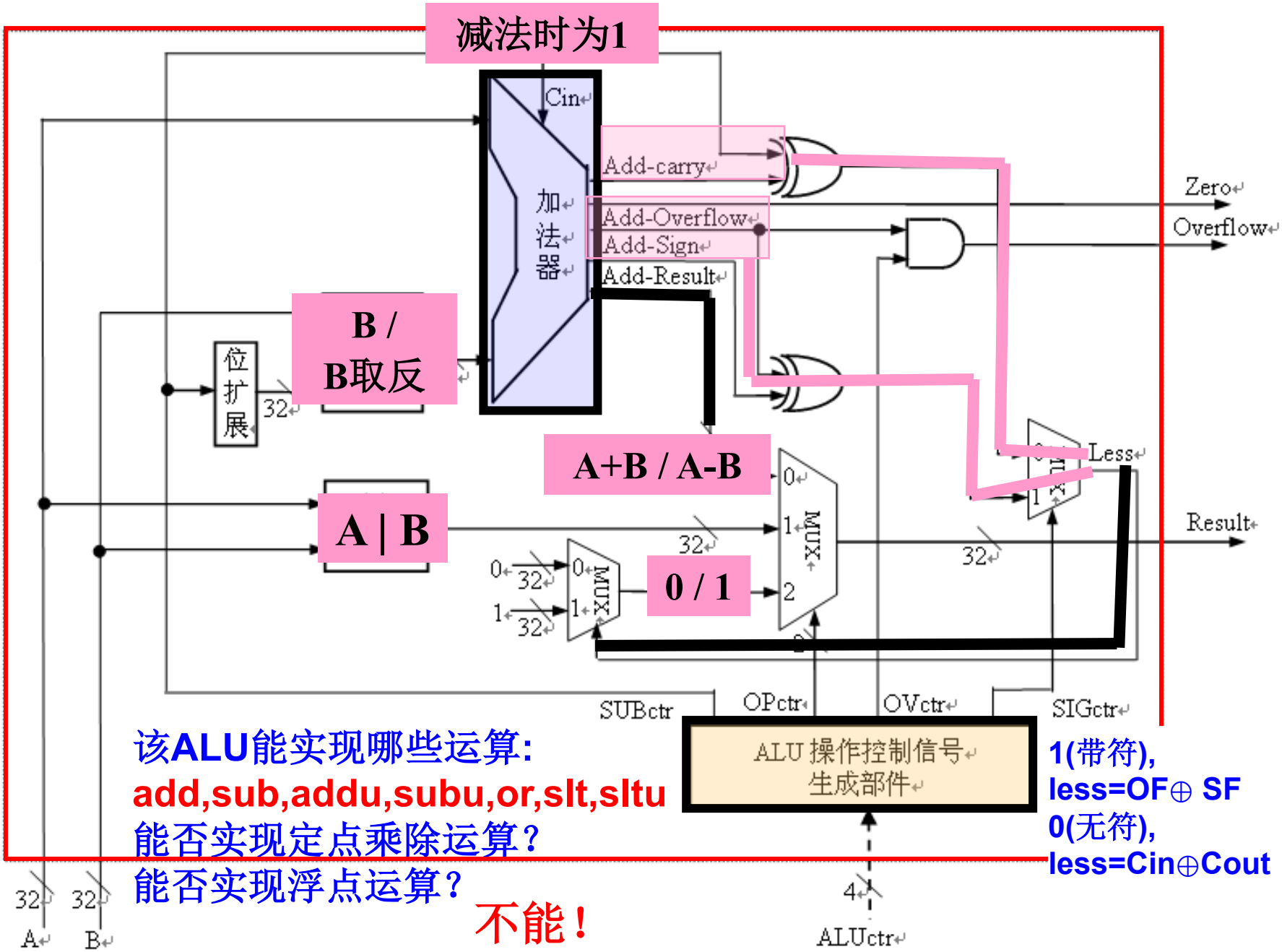
Cin=sub=0
Cout=1

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MIPS指令的ALU



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，不区分小数还是整数)

- 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）

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：

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

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

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

- 递归!
- 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。

Example：无符号整数乘法运算

举例说明：

需要哪些存储空间？

设X=1110　Y=1101

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。

部分积初始为0。

保留进位位。

右移时进位、部分积和剩余乘数一起进行逻辑右移。

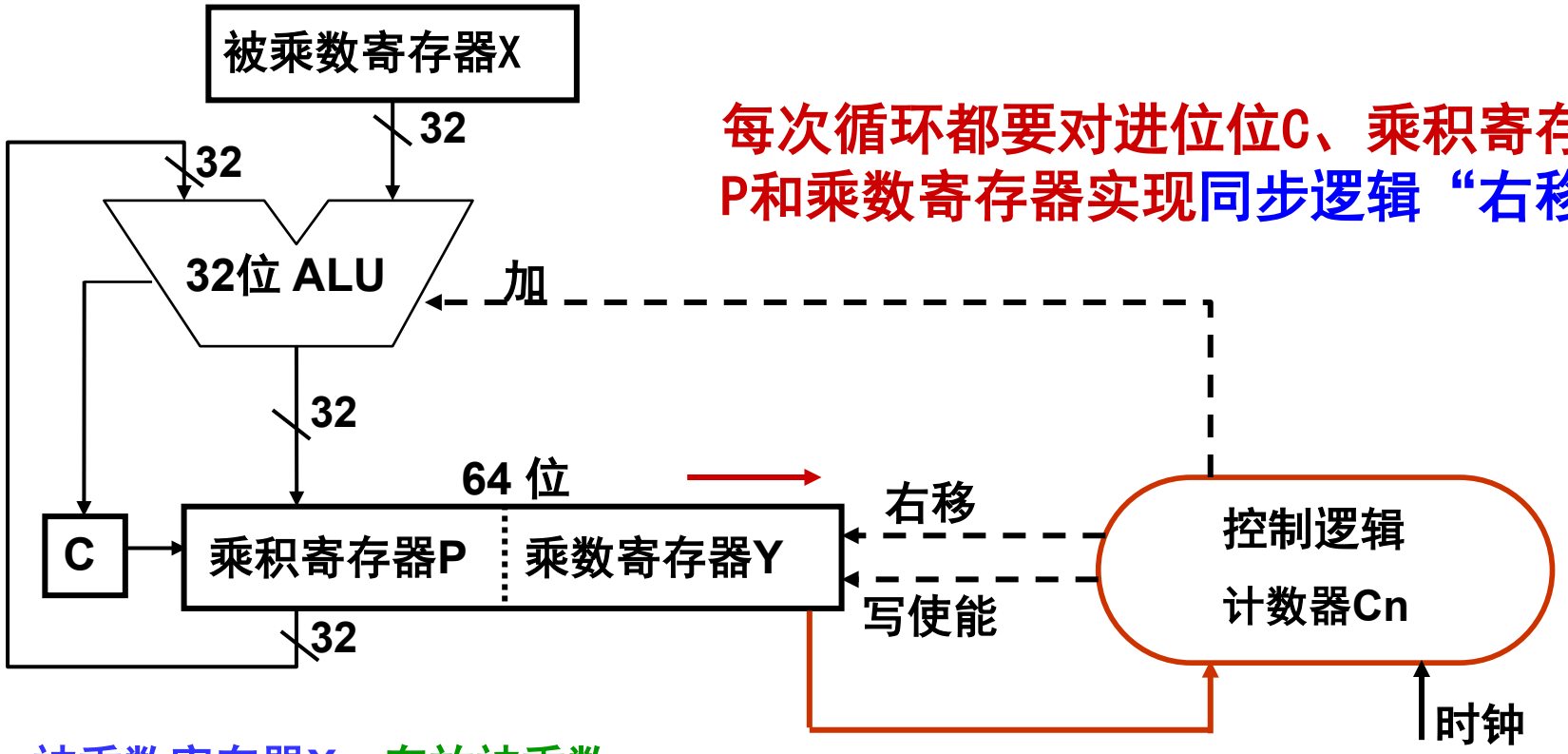
验证：X=14, Y=13, XY=182

当乘积取低4位时，结果发生溢出，因为高4位不为全0！

回顾第13次课

- 算术逻辑部件（ALU）——核心是加法器，用ALUop指定运算类型
- 有（无）符号数的加减运算（溢出判断，大小比较）
- 无符号数乘法：“判断”、“加”+“右移”（每次逻辑右移1位）

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

3X时，本次-X，下次+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 → T	$2^{-2}(P_i)$
0	0	1	+X 0 → T	$2^{-2}(P_i+X)$
0	1	0	+X 0 → T	$2^{-2}(P_i+X)$
0	1	1	+2X 0 → T	$2^{-2}(P_i+2X)$
1	0	0	+2X 0 → T	$2^{-2}(P_i+2X)$
1	0	1	–X 1 → T	$2^{-2}(P_i-X)$
1	1	0	–X 1 → T	$2^{-2}(P_i-X)$
1	1	1	1 → T	$2^{-2}(P_i)$

T触发器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?

为什么用模8补码形式(三位符号位)？

有加有减，所以要算术移位

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——

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

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

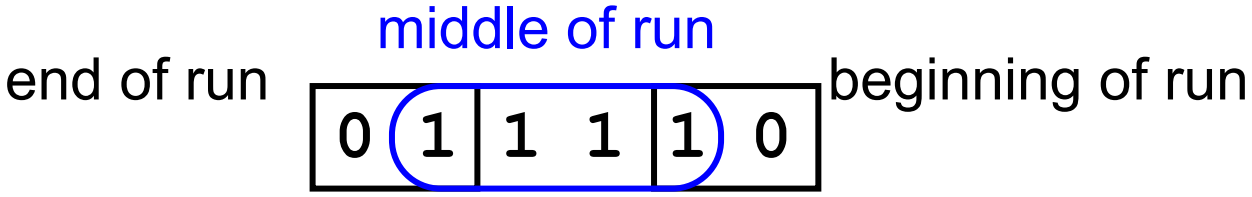
部分积公式：$[P_i]_{补}=[\,2^{-1}([P_{i-1}]_{补}+(y_{i-1}-y_i)\cdot x)\,]_{补}$

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$[P_4]_{补}$

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$[P_4]_{补}$

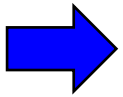
因此 $[X\times Y]_{补}$=1110 1110，与一位补码乘法（布斯乘法）所得结果相同，但循环次数减少了一半。

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”

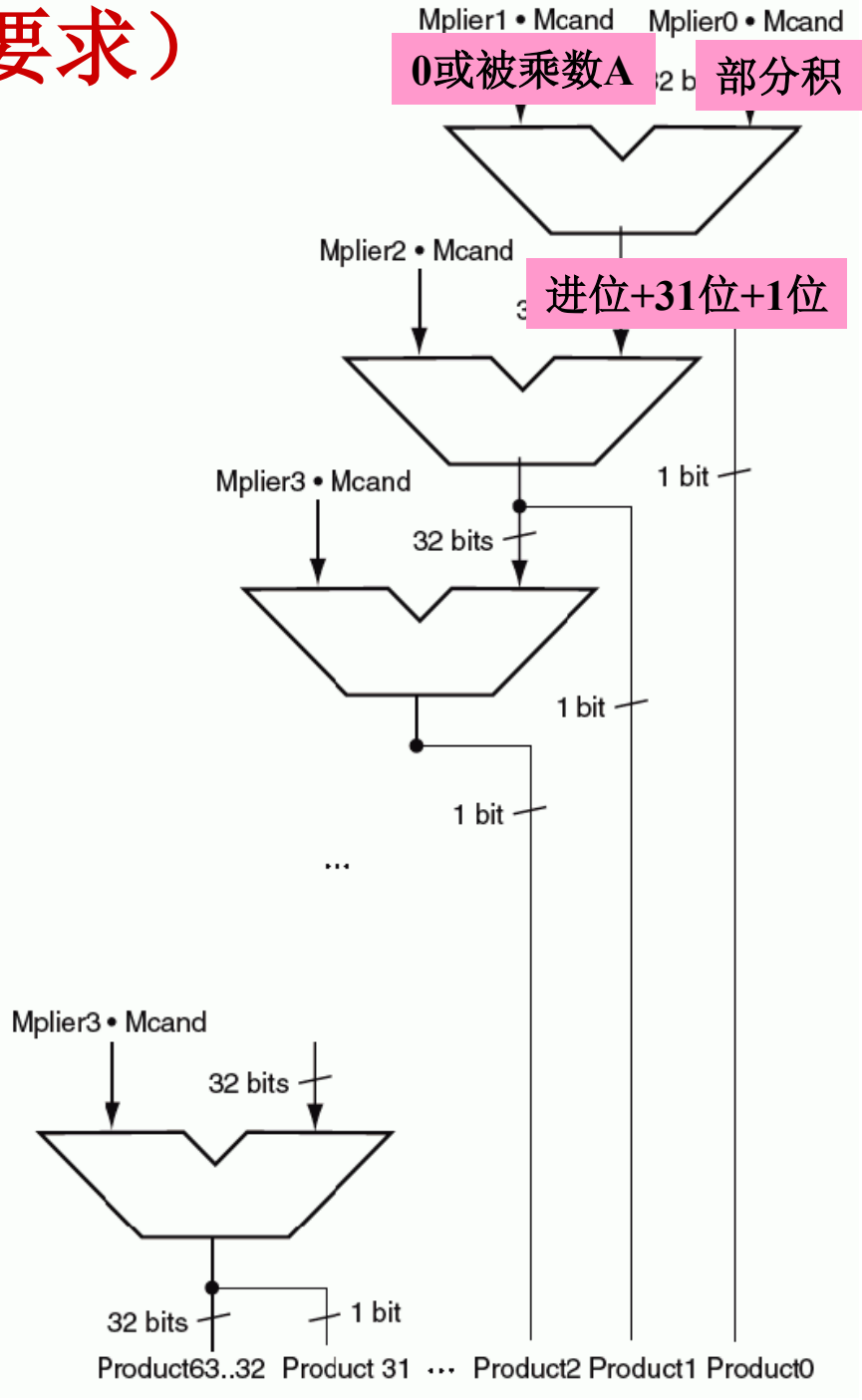
  所需时间随位数增多而加长，由时钟和控制电路控制
- 设计快速乘法部件的必要性
  - 大约1/3是乘法运算
- 快速乘法器的实现（由特定功能的组合逻辑单元构成）
  - 流水线方式
  - 硬件叠加方式（如：阵列乘法器）



流水线快速乘法器（不要求）

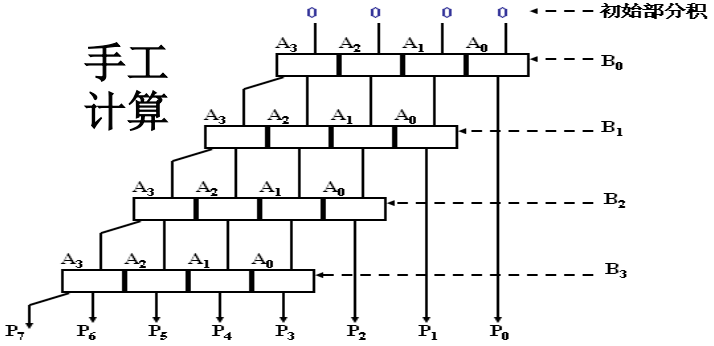
- 为乘数的每位提供一个n位加法器
- 每个加法器的两个输入端分别是：
  - 本次乘数对应的位与被乘数相与的结果（即：0或被乘数）
  - 上次部分积
- 每个加法器的输出分为两部分：
  - 和的最低有效位(LSB)作为本位乘积
  - 进位和高31位的和数组成一个32位数作为本次部分积

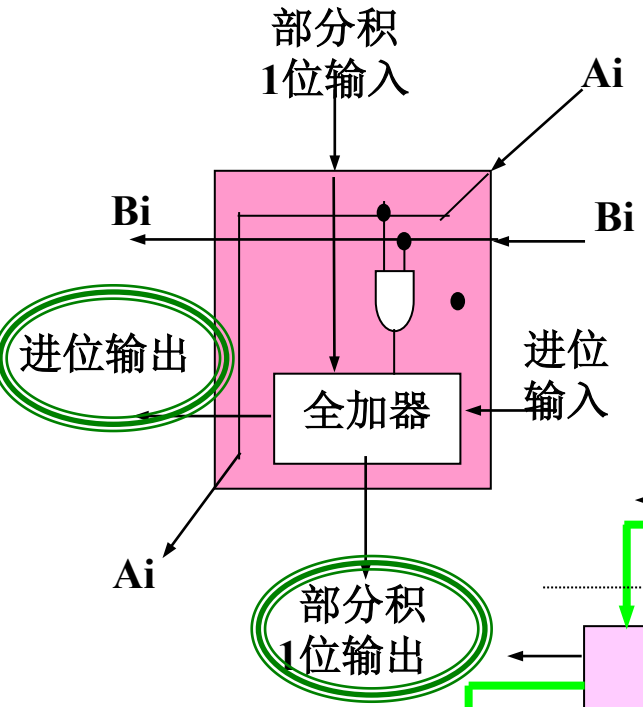
像流水一样，完全是串行，浪费加法器资源——但是，组合逻辑电路！无需控制器控制

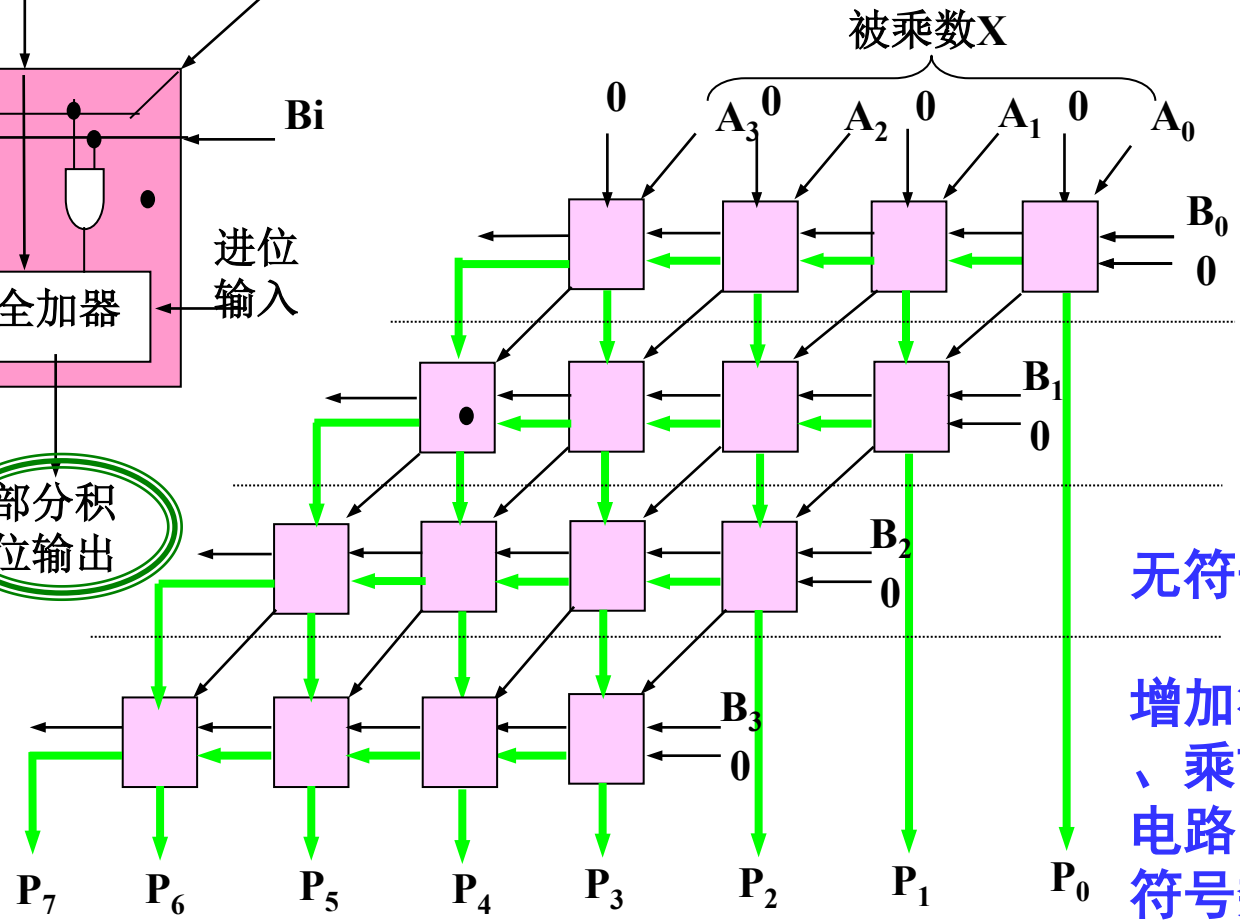


CRA阵列乘法器（不要求）

- 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。

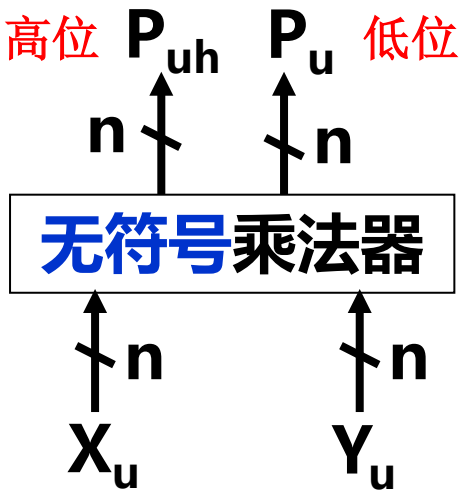
还可采用树形结构（如华莱士树）进行部分积求和，以加快速度

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）。



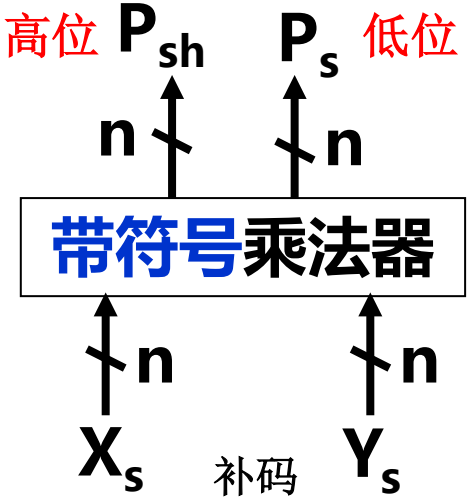
如果：

$X_u = X_s$

$Y_u = Y_s$

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，个源操作数隐含在累加器AL，结果存放在AX（16位时）立时）中。
  - MIPS中，得到的64位乘积置于两个32据Hi寄存器中的每一位是否断。
  - 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