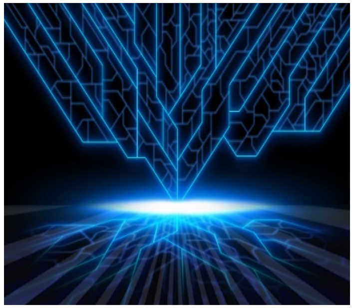
第5讲
整数运算



吴海军

南京大学计算机科学与技术系

主要内容

- 位运算
  - 按位运算
  - 逻辑运算
  - 移位运算
  - 位扩展和位截断运算
- 数据运算
  - 无符号和带符号整数的加减运算、乘除运算
  - 变量与常数之间的乘除运算

逻辑运算-布尔运算

- 布尔代数：1850 英国乔治·布尔发现，采用二进制值0（False）和1（True），能够设计出一种代数的方法来研究逻辑推理的基本原则。
- 1937年香农建立了布尔代数和数字电路之间的联系。
  - 利用布尔代数设计和分析继电器网络。
  - 用开/关来表示为0/1。
- C语言逻辑运算符
  - “‖”表示“或”运算
  - “&&”表示“与”运算
  - “!”表示“非”运算

Examples (char data type)

```
!0x41  =  0x00
!0x00  = 0x01
!!0x41  =  0x01

0x69 && 0x55  =  0x01
0x69 || 0x55  =  0x01
```

- 逻辑表达式的运算结果只有1位，0或1。
- 0看成“False”，任何非0的数看成“True”。

按位运算

- 对位串/位向量中的每一位实现按位运算的操作。
- 操作
  - 按位或： “|”
  - 按位与： “&”
  - 按位取反： “~”
  - 按位异或： “^”

C语言中按位运算适用任何“整数”类型：
long, int, short, char, unsigned
例如： (Char data type)
~0x41 = 0xBE
　~01000001_2 → $=10111110_2$
0x69 & 0x55 =0x41
　01101001_2 & 01010101_2 = 01000001_2
0x69 | 0x55 = 0x7D
　01101001_2 | 01010101_2 → 01111101_2

问题：如何从16位采样数据y中提取高位字节，并使低字节为0?
可用“&”实现位运算操作：y & 0xFF00
例如，当y=0x2C0B时，得到结果为：0x2C00

移位运算

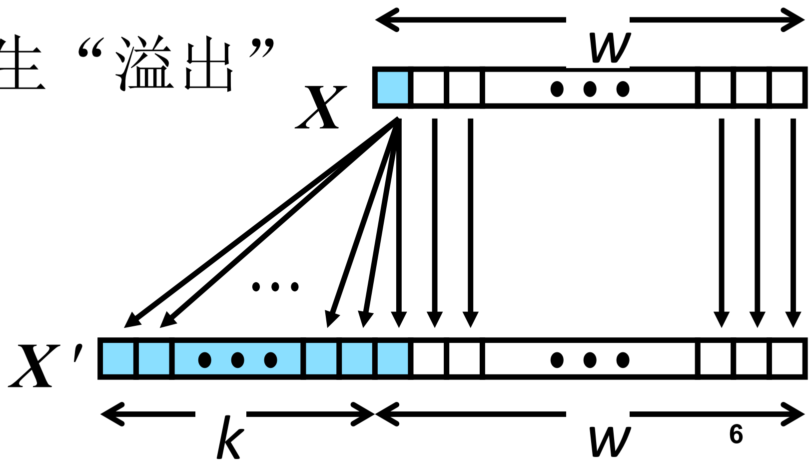
- 用于：提取部分信息，扩大或缩小数值的2、4、8…倍
- 左移： x<<k，表示丢弃最高的k位，并在右端补充k个0，k小于x的位数。
- 右移：x>>k，由x的类型确定补充的数值。
  - 无符号数：表示丢弃最低的k位，并在左端补充k个0
  - 带符号整数：
    - 逻辑右移：表示丢弃最低的k位，并在左端补充k个0
    - 算术右移：表示丢弃最低的k位，并在左端补充k个符号位

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

位扩展和位截断运算

- 用于：类型转换时可能需要数据扩展或截断。
- 操作：没有专门操作运算符，根据类型转换前后数据长短确定是扩展还是截断
  - 扩展：短转长
    - 无符号数：0扩展，高位补0
    - 带符号整数：符号扩展，高位补符号位
  - 截断：长转短
    - 强行将高位丢弃，故可能发生“溢出”



位扩展和位截断实例

```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

例2（截断操作）：i 和 j 是否相等?

```
int i = 32768;
short si = (short) i;
int j = si;
```

不相等!

i = 32768   00 00 80 00

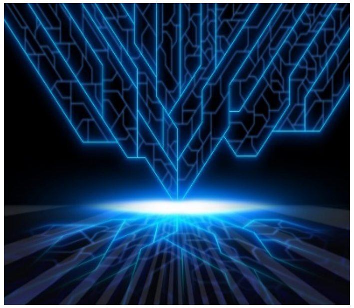
si = -32768   80 00

j = -32768   FF FF 80 00

原因：对i截断时发生了“溢出”，即：32768截断为16位数时，因其超出16位能表示的最大值，故无法截断为正确的16位数!

数据运算

无符号和带符号整数的加减运算
无符号和带符号整数的乘除运算
变量与常数之间的乘除运算



基本运算类型

- C语言程序中的基本数据类型及基本运算类型
  - 基本数据类型
    - 无符号数、带符号整数、浮点数、位串、字符（串）
  - 基本运算类型
    - 算术、按位、逻辑、移位、扩展和截断、匹配
- 计算机如何实现高级语言程序中的运算？
  - 将各类表达式编译（转换）为指令序列
  - 计算机直接执行指令来完成运算

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;
```

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 $[x-y]_{补}$。

根据补码定义，

假定补码有n位，则：

$[X]_{补}=2^n+X \quad (-2^{n-1} \leq X < 2^{n-1}, \bmod 2^n)$

有如下公式：

$[x+y]_{补}=2^n+x+y= 2^n+x+2^n+y= [x]_{补}+[y]_{补} \pmod{2^n}$

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

n位整数加/减运算器

- 补码加减运算公式

  $[A+B]_{补} = [A]_{补} + [B]_{补} \pmod{2^n}$

  $[A-B]_{补} = [A]_{补} + [-B]_{补} \pmod{2^n}$

  – 实现减法的主要工作在于：求$[-B]_{补}$

问题：如何求$[-B]_{补}$？

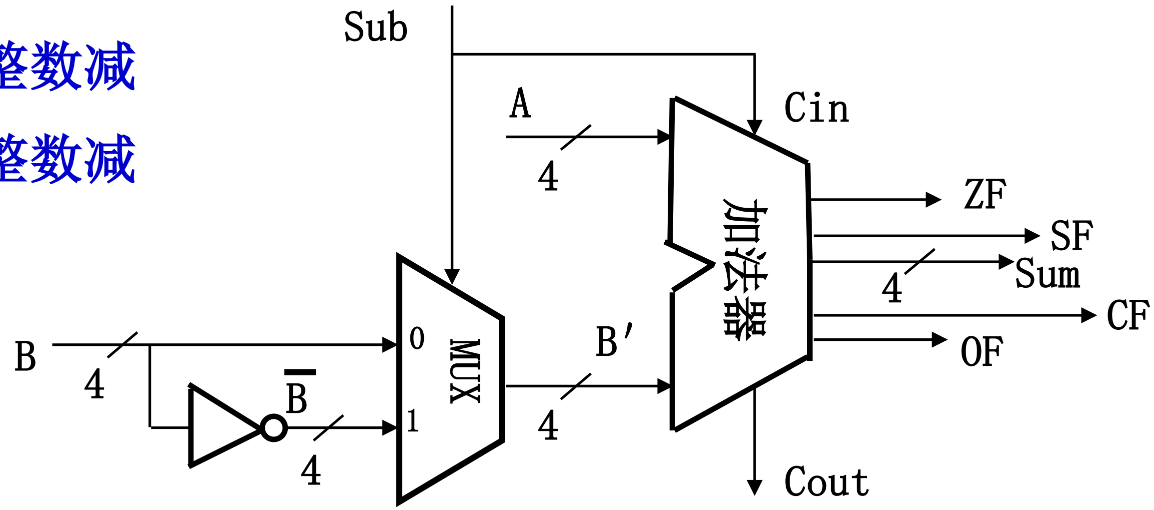
$[-B]_{补}=[-B]_{反}+1$

- 利用带标志加法器，可构造整数加/减运算器，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

当Sub为1时，做减法
当Sub为0时，做加法

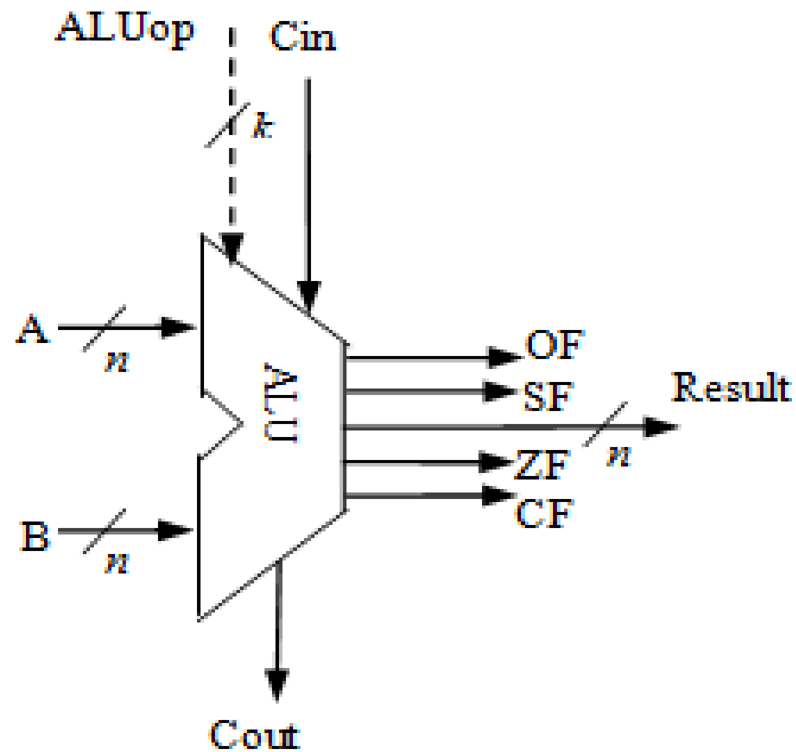


整数加/减运算部件

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

算术逻辑部件（ALU）

- 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
- 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。

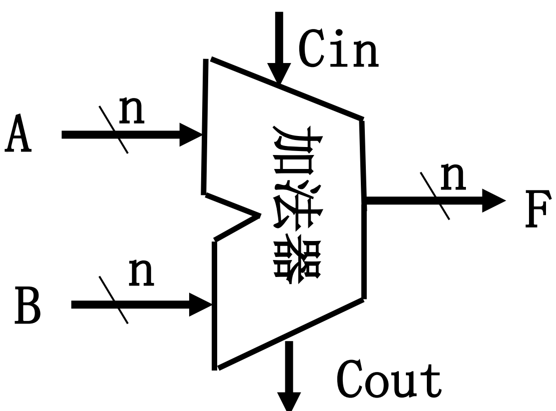


ALU 符号

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

整数加、减运算

- C语言程序中的整数有
  - 带符号整数，如char、short、int、long型等
  - 无符号整数，如unsigned char、unsigned short、unsigned等
- 指针、地址等通常被说明为无符号整数，因而在进行指针或地址运算时，需要进行无符号整数的加、减运算
- 无符号整数和带符号整数的加、减运算电路完全一样，这个运算电路称为整数加减运算部件，基于带标志加法器实现
- 最基本的加法器，因为只有n位，所以是一种模2^n运算系统！



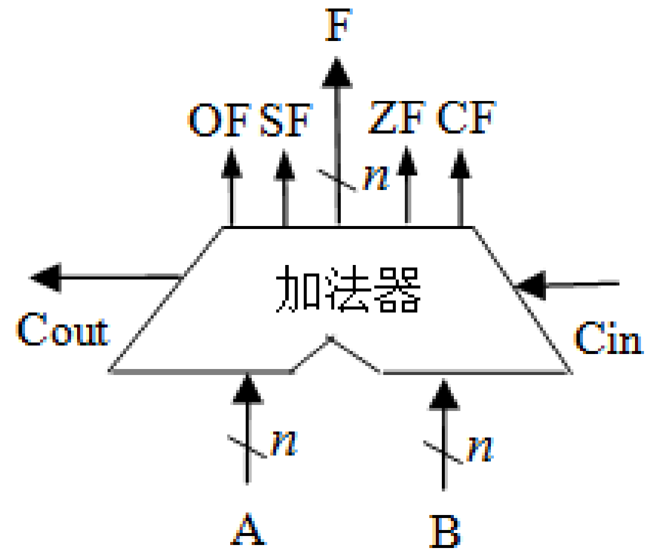
例：n=4，A=1001，B=1100

则：F=0101，Cout=1

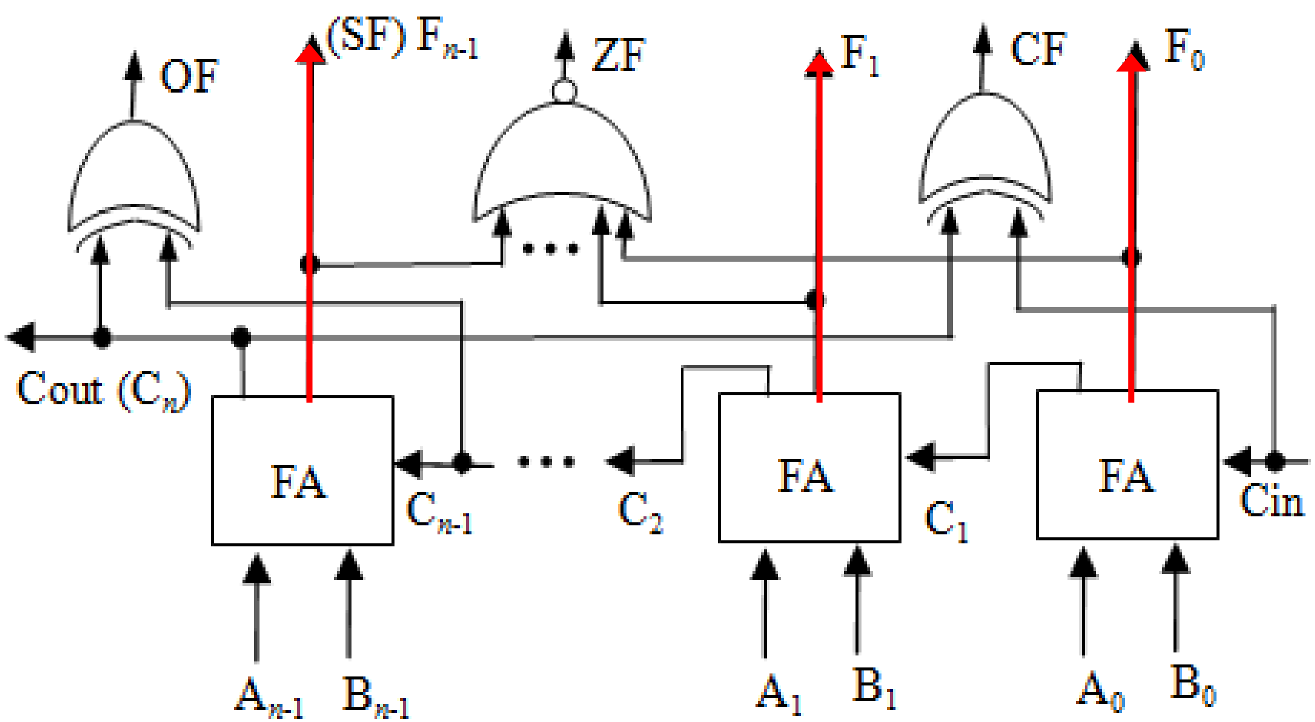
还记得这个加法器是如何实现的？

n位带标志加法器

- 程序中经常需要比较数值大小，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



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：

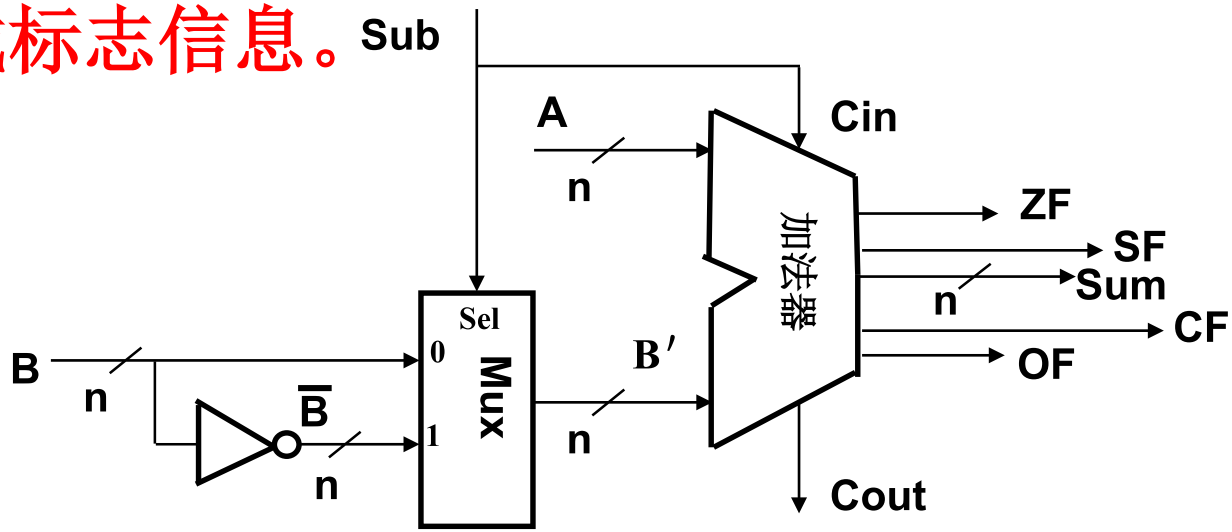
$CF=Cout \oplus Cin$

ALU的核心

重要认识1：计算机中所有算术运算都基于加法器实现！

重要认识2：加法器不知道所运算的是带符号数还是无符号数。

重要认识3：加法器不判定对错，总是取低n位作为结果，并生成标志信息。



ZF零标志
SF符号标志
CF进/借位标志
OF溢出标志

整数加/减运算部件

条件标志位（条件码CC）

- 条件标志（Flag）在运算电路中产生，存放到专门的状态寄存器EFLAGS中。用于异常处理、大小比较、条件转移等。
- 条件标志：零标志ZF、溢出标志OF、进/借位标志CF、符号标志SF。

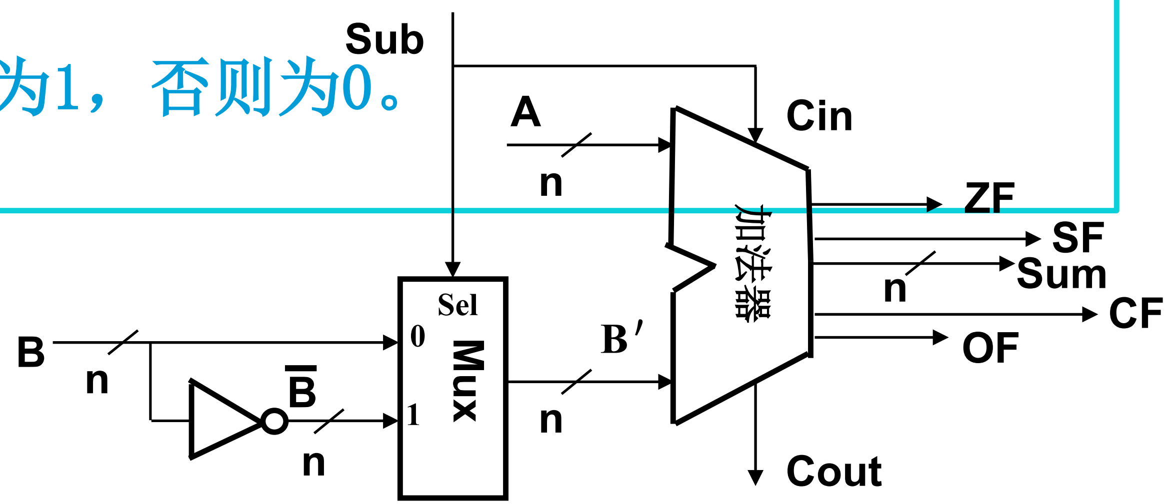
如何得到各个标志位

OF：若A与B同符号但与Sum不同符号，则为1；否则为0。

SF：sum符号。

ZF：如Sum为0，则为1，否则为0。

CF：Cout ⊕ Cin



整数加法举例

做加法时，主要判断是否溢出

无符号加法溢出条件：CF=1

带符号加法溢出条件：OF=1

若n=8，计算107+46=?

$$
\begin{array}{r}
107_{10}=\ 0110\ 1011_2 \\
46_{10}=\ 0010\ 1110_2 \\
\hline
\boxed{0}\ 1001\ 1001
\end{array}
$$

进位是真正的符号：+153

计算标志位：

$OF=C_n \oplus C_{n-1}$，$OF=0 \oplus 1=1$

$CF=Cout \oplus Cin$，$CF=0 \oplus 0=0$

溢出标志OF=1、零标志ZF=0、符号标志SF=1、进位标志CF=0

无符号数相加：sum=153，因为CF=0，未溢出，结果正确！

带符号数相加：sum= -103，因为OF=1，发生溢出，结果错误！

整数减法举例

做减法时，判断是否溢出

无符号减法溢出条件：CF=1

带符号减法溢出条件：OF=1

若n=8，计算46–107=?

$$
\begin{array}{rr}
46_{10} = & 0010\ 1110_2 \\
+(-107)_{10} = & 1001\ 0101_2 \\
\hline
 & 1100\ 0011
\end{array}
$$

计算标志位：

$OF=C_n \oplus C_{n-1}$，$OF=0\oplus0=0$

$CF=Cout \oplus Cin$，$CF=0\oplus1=1$

溢出标志OF=0、零标志ZF=0、符号标志SF=1、借位标志CF=1

无符号数相减：差=195，因为借位CF=1，溢出，结果错误！

带符号数相减：差= -61，因为OF=0，未溢出，结果正确！

整数减法举例

-7- 6 = -7 + (-6) = +3 x

9 - 6 = 3 √

$$
\begin{array}{cccccc}
 & \boxed{1} & \boxed{0} & & & \\
 & & 1 & 0 & 0 & 1 \\
+ & & 1 & 0 & 1 & 0 \\
\hline
 & & 0 & 0 & 1 & 1
\end{array}
$$

OF=1、ZF=0

SF=0、借位CF=0

-3 - 5 = - 3 + (- 5) = - 8 √

13 - 5 = 8 √

$$
\begin{array}{cccccc}
 & 1 & 1 & 1 & 1 & \\
 & & 1 & 1 & 0 & 1 \\
+ & & 1 & 0 & 1 & 1 \\
\hline
 & & 1 & 0 & 0 & 0
\end{array}
$$

OF=0、ZF=0、

SF=1、借位CF=0

带符号溢出：（1）最高位和次高位的进位不同

（2）和的符号位和加数的符号位不同

无符号减溢出：差为负数，即借位CF=1

做减法可以比较大小，规则：

无符号：CF=0时，大于

带符号：OF=SF时，大于

验证：9>6，故CF=0；13>5，故CF=0

验证：−7<6，故OF≠SF

−3<5，故OF≠SF

无符号整数加法溢出判断程序

如何用程序判断一个无符号数相加没有发生溢出

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

发生溢出时，一定满足 result<x and result<y

否则，若$x+y-2^n \geq x$，则 $y \geq 2^n$，这是不可能的！

```
/* Determine whether arguments can be added without
overflow */
int uadd_ok(unsigned x, unsigned y)
{
      unsigned sum = x+y;
      return sum >= x;
}
```

带符号整数加法溢出判断程序

Add指令需要用以下公式：

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

CF=?, ZF=?, OF=?, SF=?

CF=0, ZF=0, OF=1, SF=1（正溢出）

CF=1, ZF=0, OF=1, SF=0（负溢出）

如何用程序判断一个带符号整数相加没有发生溢出

- /* Determine whether arguments can be added without overflow */

```
int tadd_ok(int x, int y) {
    int sum = x+y;
    int neg_over = x < 0 && y < 0 && sum >= 0;
    int pos_over = x >= 0 && y >= 0 && sum < 0;
    return !neg_over && !pos_over;
}
```

整数的乘运算

- 通常高级语言中两个n位整数相乘得到的结果通常也是一个n位整数，即结果只取2n位乘积中的低n位。
  - 例如，在C语言中，参加运算的两个操作数的类型和结果的类型必须一致，如果不一致则会先转换为一致的数据类型再进行计算。

```
int mul(int x, int y)
{
    int z=x*y;
    return z;
}
```

x*y 被转换为乘法指令，在乘法运算电路中得到的乘积是64位，但是，只取其低32位赋给z。

整数的乘运算

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，$5^2=-7<0$ ！

$$
\begin{array}{r}
0101 \\
\times \quad 0101 \\
\hline
0101 \\
+\quad 0101 \\
\hline
0001\underline{1001}
\end{array}
$$

结果溢出

只取低4位，值为-111B=-7

```
int mul(int x, int y)
{
    int z=x*y;
    return z;
}
```

高级语言程序如何判断z是正确值？

当 !x || z/x==y 为真时

编译器如何判断？

当 $-2^{n-1} \leqslant x*y < 2^{n-1}$ （不溢出）时

即：乘积的高n位为全0或全1，并等于低n位的最高位！

即：乘积的高n+1位为全0或全1

若x、y和z都改成unsigned类型，则判断方式为

乘积的高n位为全0，则不溢出

整数的乘运算

结论：假定两个n位无符号整数x_u和y_u对应的机器数为X_u和Y_u，$p_u=x_u\times y_u$，p_u为n位无符号整数且对应的机器数为P_u；

两个n位带符号整数x_s和y_s对应的机器数为X_s和Y_s，$p_s=x_s\times y_s$，p_s为n位带符号整数且对应的机器数为P_s。

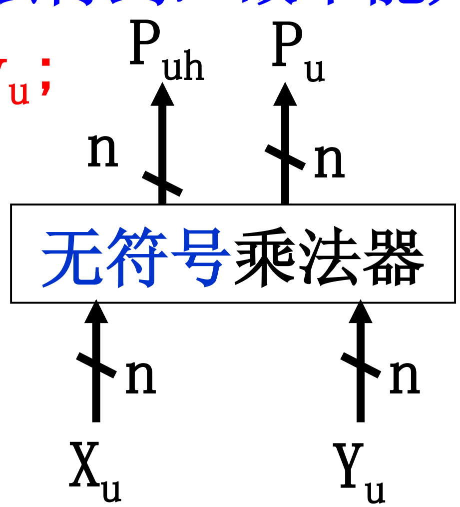
若$X_u=X_s$且$Y_u=Y_s$，则$P_u=P_s$。

可用无符号乘来实现带符号乘，但高n位无法得到，故不能判断溢出。

无符号：若$P_{uh}=0$，则不溢出

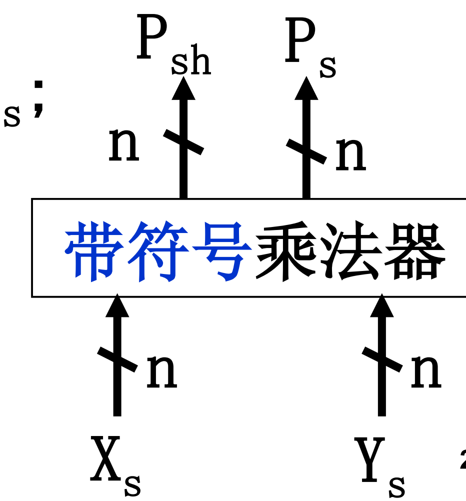
带符号：若P_{sh}每位都等于P_s的最高位，则不溢出

$p_u=x_u*y_u$;



$P_{uh} \neq P_{sh}$

$p_s=x_s*y_s$;



整数的乘运算

- 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

- 硬件保留2n位乘积，故指令的乘积可达2n位，可供编译器使用。
- 指令：分无符号数乘指令、带符号整数乘指令
- 乘法指令的操作数长度为n, 而乘积长度为2n
- 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。

乘法指令可生成溢出标志，编译器也可使用2n位乘积来判断是否溢出！

整数乘法溢出漏洞

以下程序存在什么漏洞，引起该漏洞的原因是什么。

```
/* 复制数组到堆中，count为数组元素个数 */
int copy_array(int *array, int count) {
    int i;
   /* 在堆区申请一块内存 */
    int *myarray = (int *) malloc(count*sizeof(int));
   if (myarray == NULL)
      return -1;
   for (i = 0; i < count; i++)
      myarray[i] = array[i];
   return count;
}
```

2002年，Sun Microsystems公司的RPC XDR库带的xdr_array函数发生整数溢出漏洞，攻击者可利用该漏洞从远程或本地获取root权限。

攻击者可构造特殊参数来触发整数溢出，以一段预设信息覆盖一个已分配的堆缓冲区，造成远程服务器崩溃或者改变内存数据并执行任意代码。

当参数count很大时，则count*sizeof(int)会溢出。如count=2^{30}+1时，count*sizeof(int)=4。

堆（heap）中大量数据被破坏！

变量与常数之间的乘运算

- 整数乘法运算比移位和加法等运算所用时间长，乘法运算需要多个时钟周期，而一次移位、加法和减法等运算只要一个或更少的时钟周期
- 编译器在处理变量与常数相乘时，往往以移位、加法和减法的组合运算来代替乘法运算。

例如，对于表达式x*20，编译器可以利用$20=16+4=2^4+2^2$，将x*20转换为(x<<4)+(x<<2)，这样，一次乘法转换成了两次移位和一次加法。

- 不管是无符号数还是带符号整数的乘法，即使乘积溢出时，利用移位和加减运算组合的方式得到的结果都是和采用直接相乘的结果是一样的。

整数的除运算

- 对于带符号整数来说，n位整数除以n位整数，除$-2^{n-1}/-1= 2^{n-1}$会发生溢出外，其余情况都不会发生溢出。Why?

  因为商的绝对值不可能比被除数的绝对值更大，因而不会发生溢出，也就不会像整数乘法运算那样发生整数溢出漏洞。

- 因为整数除法，其商也是整数，所以，在不能整除时需要进行舍入，通常按照朝0方向舍入，即正数商取比自身小的最接近整数（Floor，地板），负数商取比自身大的最接近整数（Ceiling，天板）。

  例如，7/2=?，　-7/2=?

  7/2=3，　-7/2=-3

整数的除运算

- 整数除0的结果可以用什么机器数表示？

  整数除0的结果无法用一个机器数表示！

- 整数除法时，除数不能为0，否则会发生“异常”，此时，需要调出操作系统中的异常处理程序来处理。

整数的除运算

代码段一：

```
int a = 0x80000000;
int b = a / -1;
printf("%d\n", b);
```

运行结果为-2147483648

用objdump看代码段一的反汇编代码，得知除以 -1 被优化成取负指令neg，故未发生除法溢出

代码段二：

```
int a = 0x80000000;
int b = -1;
int c = a / b;
printf("%d\n", c);
```

为什么显示是“浮点异常”呢?

做实验看看，分析两者反汇编代码的异同！

运行结果为“Floating point exception”，显然CPU检测到了异常

为什么两者结果不同！

变量与常数之间的除运算

- 由于计算机中除法运算比较复杂，而且不能用流水线方式实现，所以一次除法运算大致需要几十个或更多个时钟周期，**比乘法指令的时间还要长**！
- 为了缩短除法运算的时间，编译器在处理一个变量与一个2的幂次形式的整数相除时，常采用右移运算来实现。
  - 无符号整数：逻辑右移；带符号整数：算术右移
- 结果取整数
  - 能整除时，直接右移得到结果，移出的为全0

    例如，12/4=3：0000 1100>>2=0000 0011

    -12/4=-3：1111 0100 >>2=1111 1101
  - 不能整除时，右移移出的位中有非0，需要进行相应处理

变量与常数之间的除运算

- 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（取地板）：**移出的低位直接丢弃**
  - 带符号负整数（取天板）：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≠-3

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 : (x+31))>>5

但题目要求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;
}
```