

# 第二章 信息的表示和处理 I：位、整数

0.3的IEEE表示，考试重点

# 本章重点

## ■ 整数的表示

- 进制的相互转换
- 有符号数和无符号数的表示以及能够表示的范围
- 有符号数与无符号数之间相互转换

## ■ 整数的扩展和截断

## ■ 整数的加法、乘、移位

- 注意运算过程可能发生溢出

## ■ 经典例题

# 本章目录: 位、字节 和 整型数

- 信息的位表示
- 位级运算
- 整型数
  - 表示: 无符号数和有符号数
  - 无符号数和有符号数的转换
  - 扩展、截断
  - 整数运算: 加、非、乘、移位
  - 总结
- 内存、指针、字符串表示

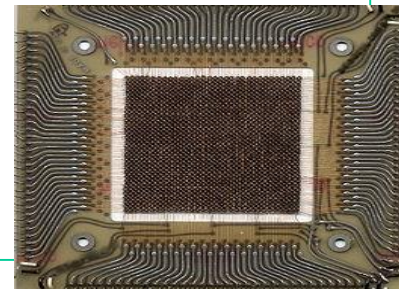
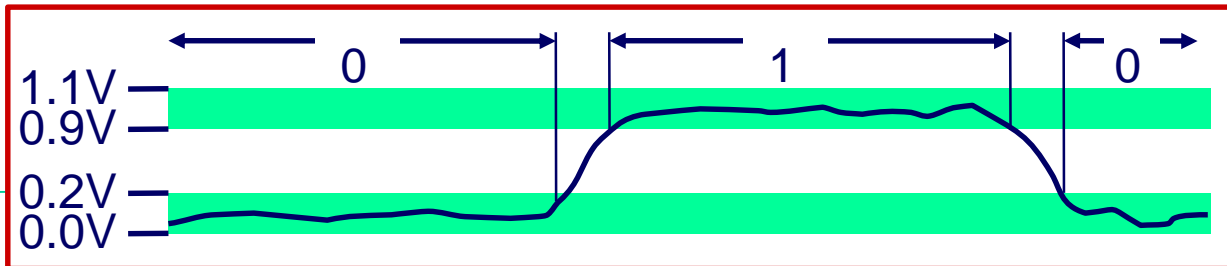
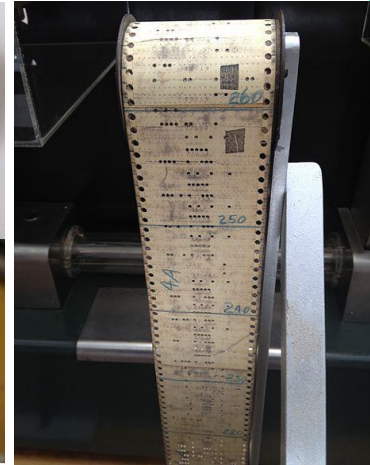
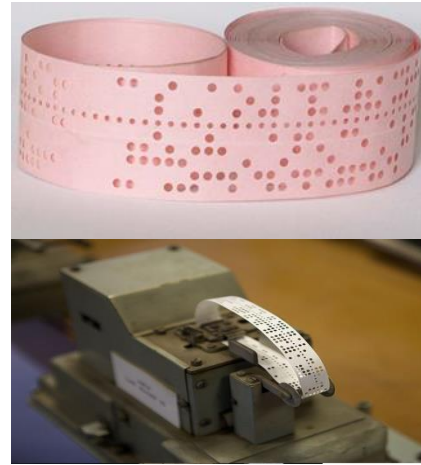
# 为什么用二进制？

## ■ 十进制——适合人类使用

- 有10个手指的人类
- 1000年前源自印度、12世纪发展于阿拉伯、13世纪到西方

## ■ 二进制——更适合机器使用

- 容易表示、存储
  - 打孔纸带上是/否有空
  - 磁场的顺时针/逆时针
- 容易传输
  - 导线上的电压高/低
  - 可以在有噪声、不精确的电路可靠传输



# 位、字节、字

- 计算机存储、处理的信息：二值信号
- “位” 或 “比特”
  - 最底层的二进制数字（数码）称为位（bit，比特），值为0或1
  - 数字革命的基础
- 位组合
  - 把位组合到一起，采用某种规则进行解读
  - 每个位组合都有含义
  - LSB（Least Significant Bit）、MSB（Most Significant Bit）  
一般LSB指的是最低有效位，MSB是最高有效位
- 字节：8-bit块
  - 人物：Dr. Werner Buchholz，1956年7月
  - 事件：IBM Stretch computer的早期设计阶段



维纳·布赫霍尔兹

# 位、字节、字

**字：CPU中ALU的数据位数=CPU中通用寄存器的位数**

**通常计算机是XX位的，是指这台计算机CPU字的长度**

OS是XX位的，是指其CPU的工作模式，这与操作系统各DLL库函数、编译链接环境有关。

1bytes=8bits

8086、286: 1word=2bytes=16bits

80386、486、586: 1word=4bytes=32bits

itanium(merced): 1word=8bytes=64bits

**X86汇编语言/机器语言编程中，一个字指的是16位。**

数据存放时高字节在高地址、低字节在低地址。

位多用于数据通讯中传输率：1200bps, 100Mb

字节用于数据存储和传输中，表示数据的规模：比如10GB

字用于表示计算机cpu中的寄存器。

# 进制

## ■ 数的通用表示

10进制:

$$3721 = 3 \times 10^3 + 7 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$$

$$N = \pm a_n a_{n-1} \dots a_1 a_0 . b_1 b_2 \dots b_m$$

k进制:

$$N = \pm a_n \times k^n + a_{n-1} \times k^{n-1} + \dots + a_1 \times k^1 + a_0 \times k^0 \\ + b_1 \times k^{-1} + b_2 \times k^{-2} + \dots + b_m \times k^{-m}$$

其中 $a_i$ ,  $b_j$ 是 $0 \sim k-1$ 中的一个数码

# 二进制数

- 特点：逢二进一，由0和1两个数码组成，基数为2，各个位权以 $2^i$ 表示

- 二进制数：

$$a_n a_{n-1} \dots a_1 a_0 . b_1 b_2 \dots b_m =$$

$$a_n \times 2^n + a_{n-1} \times 2^{n-1} + \dots + a_1 \times 2^1 + a_0 \times 2^0 \\ + b_1 \times 2^{-1} + b_2 \times 2^{-2} + \dots + b_m \times 2^{-m}$$

其中 $a_i$ ,  $b_j$ 非0即1

便于计算机存储、算术运算简单、支持逻辑运算



# 二进制数

- MSB: 最高有效位 (Most Significant Bit)
- LSB: 最低有效位 (Least Significant Bit)

MSB

LSB

1	0	1	1	0	0	1	0	1	0	0	1	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

15

0

数字串长、书写和阅读不便

# 十六进制数

- 基数16，逢16进位，位权为 $16^i$ ，16个数码：

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

- 十六进制数：

$$a_n a_{n-1} \dots a_1 a_0 . b_1 b_2 \dots b_m =$$

$$\begin{aligned} & a_n \times 16^n + a_{n-1} \times 16^{n-1} + \dots + a_1 \times 16^1 + a_0 \times 16^0 \\ & + b_1 \times 16^{-1} + b_2 \times 16^{-2} + \dots + b_m \times 16^{-m} \end{aligned}$$

其中 $a_i$ ,  $b_j$ 是0 ~ F中的一个数码

# 十六进制数的加减运算

## ■ 十六进制数的加减运算类似十进制

- 逢16进位1，借1当16

$$23D9H + 94BEH = B897H$$

$$A59FH - 62B8H = 42E7H$$

## ■ 二进制和十六进制数之间具有对应关系：

每4个二进制位对应1个十六进制位

$$00111010B = 3AH, F2H = 11110010B$$

与二进制数相互转换简单、阅读书写方便

# 进制转换

## ■ 十进制整数转换为k(2、8或16)进制数

整数转换：用除法—除基取余法

- 十进制数整数部分不断除以基数k(2、8或16)，并记下余数，直到商为0为止
- 由最后一个余数起，逆向取各个余数，则为转换成的二进制和十六进制数

$126 = 01111110$ **B**      二进制数用后缀字母B

$126 = 7E$ **H**                  十六进制数用后缀字母H

# 进制转换

- 十进制**小数**转换为k(2、8或16)进制数...

小数转换：用乘法—乘基取整法

乘以基数k，记录整数部分，直到小数部分为0为止

$$0.8125 = 0.1101\mathbf{B}$$

$$0.8125 = 0.\mathbf{DH}$$

- 小数转换会发生总是无法乘到为0的情况
- 可选取一定位数（精度）
- 将产生无法避免的转换误差

【例 2.2】将十进制数 123.6875 转换成二进制数。

解：

整数部分：

除基	取余	
2   1 2 3	1	最低位
2   6 1	1	
2   3 0	0	
2   1 5	1	
2   7	1	
2   3	1	
2   1	1	最高位
0		

**整数部分结束条件：商为0**

因此整数部分  $123 = (1111011)_2$ 。

乘基取整法（小数部分的转换）：小数部分乘基取整，最先取得的整数为数的最高位，最后取得的整数为数的最低位（即乘基取整，先整为高，后整为低），乘积为 1.0（或满足精度要求）时结束。

**小数部分结束条件：乘积为1或者满足精度就结束**

小数部分：

乘基	取整	
0.6875		
× 2	1	最高位
1.3750		
0.3750		
× 2	0	
0.7500		
× 2	1	
1.5000		
0.5000		
× 2	1	最低位
1.0000		

**最后从高位到低位，写出二进制浮点数的整数和小数部分**

因此小数部分  $0.6875 = (0.1011)_2$ ，所以  $123.6875 = (1111011.1011)_2$ 。

注意：在计算机中，小数和整数不一样，整数可以连续表示，但小数是离散的，所以并不是每个十进制小数都可以准确地用二进制表示。例如 0.3，无论经过多少次乘二取整转换都无法得到精确的结果。但任意一个二进制小数都可以用十进制小数表示，希望读者引起重视。

注意：关于十进制数转换为任意进制数为何采用除基取余法和乘基取整法，以及所取之数放置位置的原理，请结合  $r$  进制数的数值表示公式思考，而不应死记硬背。

# 进制转换

## ■ k进制数转换为十进制数

### 方法：按权展开

#### ■ 二进制数转换为十进制数

0011.1010<sub>B</sub>

$$= 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 3.625$$

#### ■ 十六进制数转换为十进制数

$$1.2_{\text{H}} = 1 \times 16^0 + 2 \times 16^{-1} = 1.125$$

## ■ 2、8、16进制间的转换

4个2进制位对应1个16进制位

3个2进制位对应1个8进制位

# 计算机内的数值表示——编码

## ■ 需要考虑的问题

① 编码的长度

② 数的符号

③ 数的运算



# 字节值编码

## ■ Byte = 8 bits

- 2进制(Binary)  $00000000_2 - 11111111_2$
- 10进制(Decimal):  $0_{10} - 255_{10}$
- 16进制(Hexadecimal):  $00_{16} - FF_{16}$

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

# C数据类型的宽度（与编译器有关）

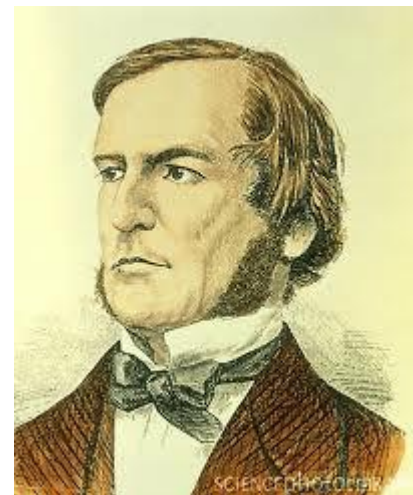
C 数据类型	32位	64位	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	–	–	10/16
pointer	4	8	8

# 本章目录: 位、字节 和 整型数

- 信息的位表示
- 位级运算
- 整型数
  - 表示: 无符号数和有符号数
  - 无符号数和有符号数的转换
  - 扩展、截断
  - 整数运算: 加、非、乘、移位
  - 总结
- 内存、指针、字符串表示

# 布尔代数(Boolean Algebra)

- **George Boole(1815-1864)**提出逻辑的代数表示
  - 逻辑值 “True(真)” 编码为 1
  - 逻辑值 “False(假)” 编码为 0
- **Claude Shannon(1916–2001)**创立信息论
  - 将布尔代数与数字逻辑关联起来
- 是数字系统设计与分析的重要工具



# 布尔代数(Boolean Algebra)

## 与(And)

- 当A=1 并且 B=1时,  $A \& B = 1$

$\&$	0	1
0	0	0
1	0	1

## 或(Or)

- 当A=1 或 B=1时,  $A | B = 1$

$ $	0	1
0	0	1
1	1	1

## 非(Not)

- 当A=0时,  $\sim A = 1$

$\sim$	
0	1
1	0

## 异或(Exclusive-Or,Xor)

- 当A=1 或 B=1且两者不同时为1,  $A \wedge B = 1$

$\wedge$	0	1
0	0	1
1	1	0

异或原则：相同为0，不同为1

# 一般的布尔代数

## ■ 位向量操作(Operate on Bit Vectors)

### ■ 按位运算

01101001	01101001		01101001
& 01010101	01010101	~ 01010101	^ 01010101
<u>          </u>	<u>          </u>	<u>          </u>	<u>          </u>
01000001	01111101	10101010	00111100

## ■ 布尔代数的全部性质均适用

# 示例:集合的表示与运算

## ■ 表示

- 宽度  $w$  个比特的向量表示集合  $\{0, \dots, w-1\}$  的子集
- 如  $j \in A$ , 则  $a_j = 1$

■ 01101001 { 0, 3, 5, 6 }

76543210

■ 01010101 { 0, 2, 4, 6 }

76543210

## ■ 运算

■ & 交集(Intersection) 01000001 { 0, 6 }

■ | 并集(Union) 01111101 { 0, 2, 3, 4, 5, 6 }

■ ^ 差集(Symmetric difference) 00111100 { 2, 3, 4, 5 }

■ ~ 补集(Complement) 10101010 { 1, 3, 5, 7 }

## 2.1.7 C语言中的位级运算

### ■ C语言中的位运算： &, |, ~, ^

- 适用于任何整型数据类型： long, int, short, char, unsigned
- 将操作数视为位向量
- 将参数按位运算

### ■ 例子(char 类型)

- $\sim 0x41 \rightarrow 0xBE$ 
  - $\sim 01000001_2 \rightarrow 10111110_2$
- $\sim 0x00 \rightarrow 0xFF$ 
  - $\sim 00000000_2 \rightarrow 11111111_2$
- $0x69 \& 0x55 \rightarrow 0x41$ 
  - $01101001_2 \& 01010101_2 \rightarrow 01000001_2$
- $0x69 | 0x55 \rightarrow 0x7D$ 
  - $01101001_2 | 01010101_2 \rightarrow 01111101_2$



# 巧用异或

## ■ 按位异或是一种加的形式

### ■ $A \oplus A = 0$

```
int inplace_swap(int *x, int *y)
{
    *x = *x ^ *y;  /* #1 */
    *y = *x ^ *y;  /* #2 */
    *x = *x ^ *y;  /* #3 */
}
```

Step	*x	*y
Begin	A	B
1	$A \oplus B$	B
2	$A \oplus B$	$(A \oplus B) \oplus B = A \oplus (B \oplus B) = A \oplus 0 = A$
3	$(A \oplus B) \oplus A = (B \oplus A) \oplus A = B \oplus (A \oplus A) = B \oplus 0 = B$	A
End	B	A

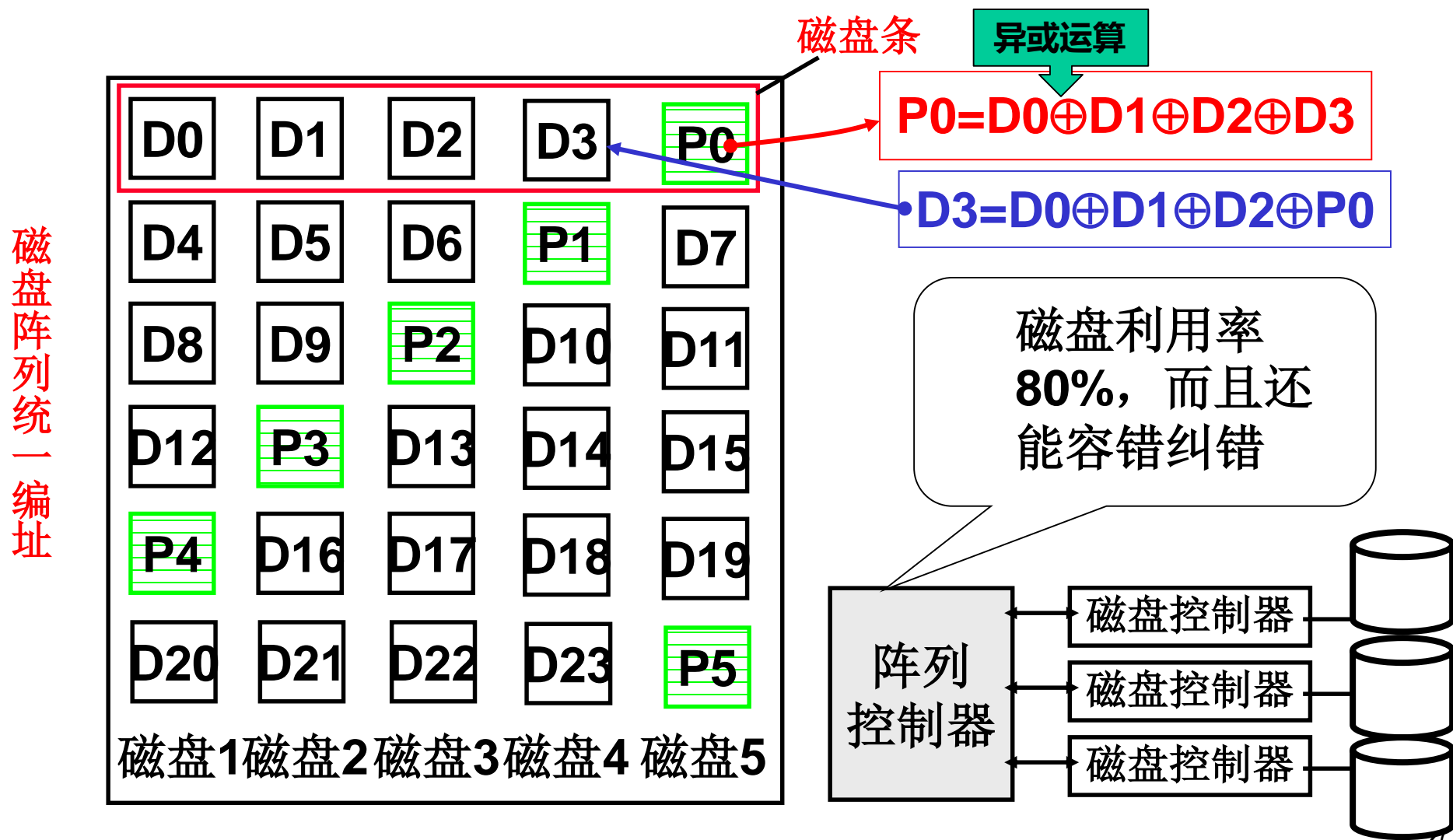
# 巧用异或

//翻转数组

```
1 void reverse_array(int a[], int cnt) {  
2     int first, last;  
3     for (first = 0, last = cnt-1; first <= last; first++, last--)  
4         inplace_swap(&a[first], &a[last]);  
5 }
```

# 巧用异或-RAID5

磁盘阵列场景：校验数据分布在多个盘上，磁盘互相恢复数据



## 2.1.8 对比: C语言的逻辑运算 (非真即假)

### ■ C语言的逻辑运算符: &&, ||, !

- 将0视作逻辑“False(假)”
- 所有非0值视作逻辑“True(真)”
- 计算结果总是0或1
- 提前终止(Early termination)、短路求值(short cut)

### ■ 例子(char 数据类型)

- !0x41 → 0x00
- !0x00 → 0x01
- !!0x41 → 0x01
- 0x69 && 0x55 → 0x01
- 0x69 || 0x55 → 0x01
- p && \*p (避免空指针访问)

## 2.1.9 C语言中的移位运算（无循环移位）

### ■ 左移: $x \ll y$

#### ■ 将位向量x向左移动 y位

- 扔掉左边多出(移出)的位
- 在右边补0

Argument x	01100010
$\ll 3$	00010 <b>000</b>
Log. $\gg 2$	<b>00</b> 011000
Arith. $\gg 2$	<b>00</b> 011000

### ■ 右移: $x \gg y$

#### ■ 将位向量x向右移动 y位

- 扔掉右边多出(移出)的位
- 逻辑(无符号数)右移: 在左边补0
- 算术右移: 复制左边的最高位(y次)

Argument x	10100010
$\ll 3$	00010 <b>000</b>
Log. $\gg 2$	<b>00</b> 101000
Arith. $\gg 2$	<b>11</b> 101000

### ■ 未明确定义

- 移位数量  $y < 0$  或  $y \geq x$  的字长(位数)
- 移位不都是除以/乘以2, 尤其是要区分算术还是逻辑移位  
还要注意可能存在的精度损失

# 本章目录: 位、字节 和 整型数

- 信息的位表示
- 位级运算
- **整型数**
  - 表示: 无符号数和有符号数
  - 无符号数和有符号数的转换
  - 扩展、截断
  - 整数运算: 加、非、乘、移位
  - 总结
- 内存、指针、字符串表示
- 总结

## 2.2 整数编码(Encoding Integers)

无符号数(**U**nsigned)    有符号数——补码(**T**wo's Complement)

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

```
short int x = 15213;
short int y = -15213;
```

符号位

### ■ C short : 2 字节

	10 进制	16 进制	2 进制
<b>x</b>	15213	3B 6D	00111011 01101101
<b>y</b>	-15213	C4 93	11000100 10010011

### ■ 符号位

- 对于补码(2's complement), 最高位表示符号
  - 0 表示非负数 ( != 正数), 1 表示负数

# 例子: B2U(X)和B2T(X) 见教材P45~46

$$\begin{aligned}
 B2U_4([0001]) &= 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 0 + 0 + 1 = 1 \\
 B2U_4([0101]) &= 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 4 + 0 + 1 = 5 \\
 B2U_4([1011]) &= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 0 + 2 + 1 = 11 \\
 B2U_4([1111]) &= 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 2 + 1 = 15
 \end{aligned} \tag{2.2}$$

$$\begin{aligned}
 B2T_4([0001]) &= -0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 0 + 0 + 1 = 1 \\
 B2T_4([0101]) &= -0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 4 + 0 + 1 = 5 \\
 B2T_4([1011]) &= -1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -8 + 0 + 2 + 1 = -5 \\
 B2T_4([1111]) &= -1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -8 + 4 + 2 + 1 = -1
 \end{aligned} \tag{2.4}$$



# 补码示例

x =

15213: 00111011 01101101

y =

-15213: 11000100 10010011

权重	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
总计	15213		-15213	

# 数值范围（共 $w$ 位）

## ■ 无符号数值

$$\begin{aligned} \blacksquare \text{ } UMin &= 0 \\ &000\dots 0 \end{aligned}$$

$$\begin{aligned} \blacksquare \text{ } UMax &= 2^w - 1 \\ &111\dots 1 \end{aligned}$$

## ■ 补码数值

$$\begin{aligned} \blacksquare \text{ } TMin &= -2^{w-1} \\ &100\dots 0 \end{aligned}$$

$$\begin{aligned} \blacksquare \text{ } TMax &= 2^{w-1} - 1 \\ &011\dots 1 \end{aligned}$$

$$\begin{aligned} \blacksquare \text{ } -1 & \\ &111\dots 1 \end{aligned}$$

## 位数 $w = 16$ 时的数值

	十进制	16 进制	二进制
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

# 不同字长的数值

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

## ■ 观察

- $|TMin| = TMax + 1$ 
  - 非对称（数轴上）
- $UMax = 2 * TMax + 1$

## ■ C 语言的常量声明

- `#include <limits.h>`
  - `#define INT_MAX 2147483647`
  - `#define INT_MIN (-INT_MAX-1)`
  - `#define UINT_MAX 0xffffffff`
- 平台相关
  - `#define ULONG_MAX`
  - `#define LONG_MAX`
  - `#define LONG_MIN (-LONG_MAX-1)`

# 无符号数与有符号数编码的值

$X$	$B2U(X)$	$B2T(X)$
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

## ■ 相同

- 非负数值的编码相同

## ■ 单值性

- 每个位模式对应一个唯一的整数值
- 每个可描述整数有一个唯一编码

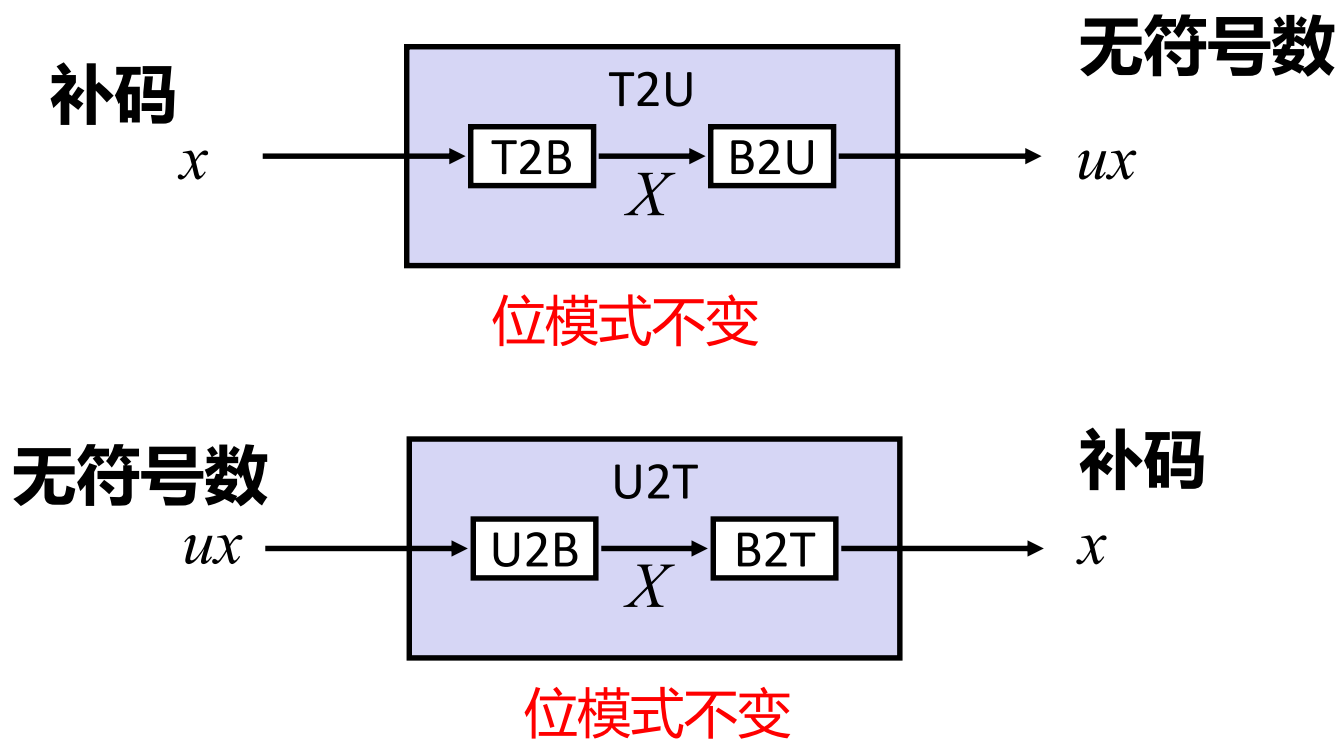
⇒ 可逆映射（一一对应）

- $U2B(x) = B2U^{-1}(x)$ 
  - 无符号整数的位模式
- $T2B(x) = B2T^{-1}(x)$ 
  - 补码的位模式

# 本章目录: 位、字节 和 整型数

- 信息的位表示
- 位级运算
- **整型数**
  - 表示: 无符号数和有符号数
  - **无符号数和有符号数的转换**
  - 扩展、截断
  - 整数运算: 加、非、乘、移位
  - 总结
- 内存、指针、字符串表示

# 有符号/无符号数之间的转换



## ■ 有符号数和无符号数转换规则:

位模式不变、数值可能改变(按不同编码规则重新解读)

# 有符号↔无符号数的转换

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5	→ T2U →	5
0110	6		6
0111	7	← U2T ←	7
1000	-8		8
1001	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

# 有符号↔无符号数的转换

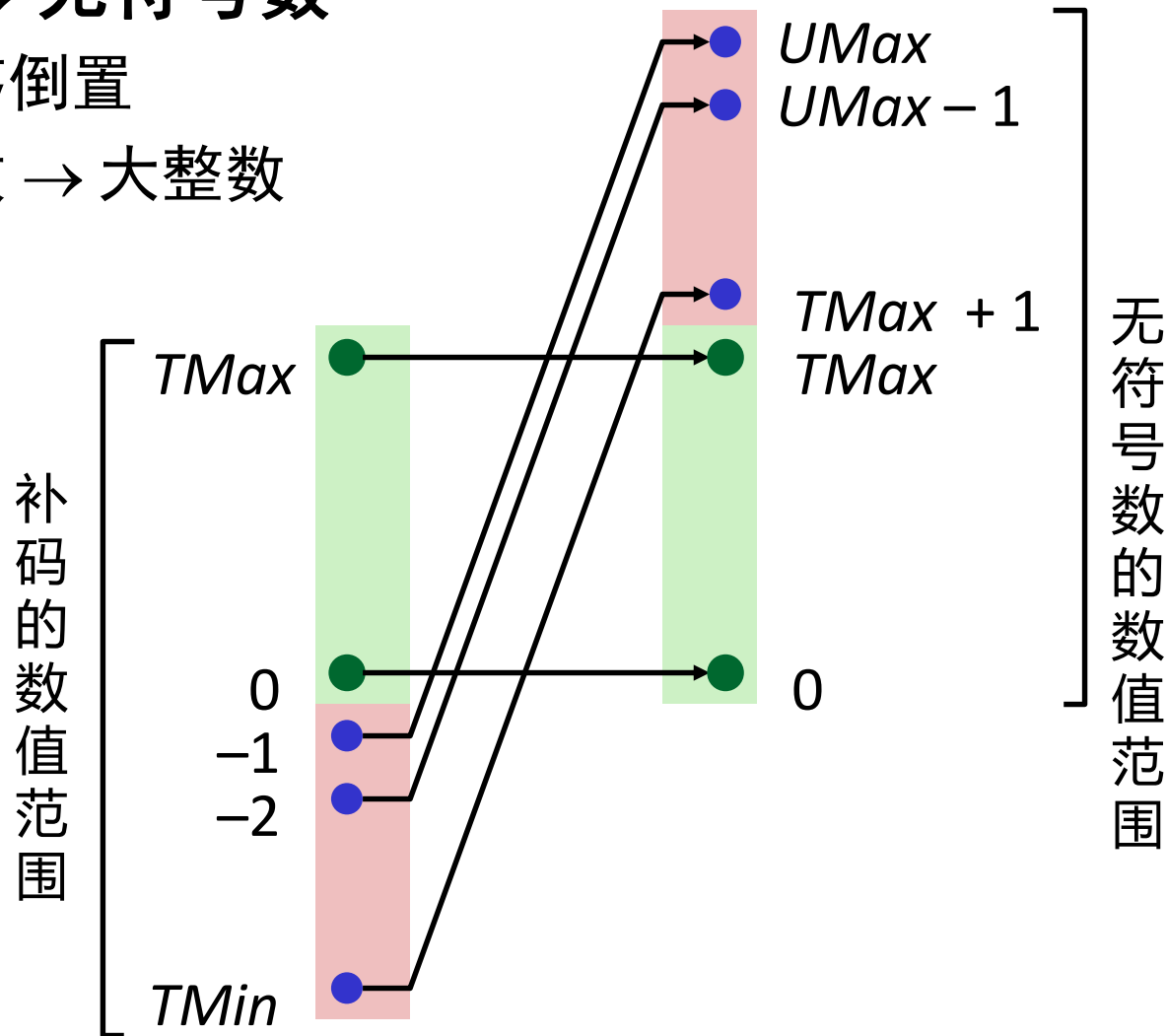
Bits	Signed		Unsigned
0000	0	=	0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8	+/- 16	8
1001	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15



# 转换的可视化

## ■ 补码→无符号数

- 顺序倒置
- 负数 → 大整数



## 2.2.5 C语言中的有符号数和无符号数

### ■ 常量

- 数字默认是有符号数
- 无符号数用后缀“U”  
0U, 4294967259U

### ■ 类型转换

- 显示的强制类型转换

```
int tx, ty;  
unsigned ux, uy;  
tx = (int) ux;  
uy = (unsigned) ty;
```

- 隐式的类型转换（赋值、函数调用等情况下发生）

```
tx = ux;  
uy = ty;
```

# 类型转换的惊喜！

## ■ 表达式计算

- 表达式中有符号和无符号数混用时：

**有符号数隐式转换为无符号数**

- 包括比较运算符  $<$ ,  $>$ ,  $==$ ,  $<=$ ,  $>=$
- 例如  $W = 32$ :

$TMIN = -2,147,483,648$

$TMAX = 2,147,483,647$

# 类型转换的惊喜!

Constant1	Constant2	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483648	>	signed
2147483647U	-2147483648	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

# 有符号数和无符号数转换的基本原则

- **位模式不变**
- **重新解读**（按目标编码类型的规则解读）
- 会有意外副作用：数值被  $\pm 2^w$
- 表达式含无符号数和有符号数时
  - 有符号数被转换成无符号数（如int 转成unsigned int）
  - 当心副作用！！！（即按照无符号数进行运算）

# 本章目录: 位、字节 和 整型数

- 信息的位表示
- 位级运算
- **整型数**
  - 表示: 无符号数和有符号数
  - 无符号数和有符号数的转换
  - **扩展、截断**
  - 整数运算: 加、非、乘、移位
  - 总结
- 内存、指针、字符串表示

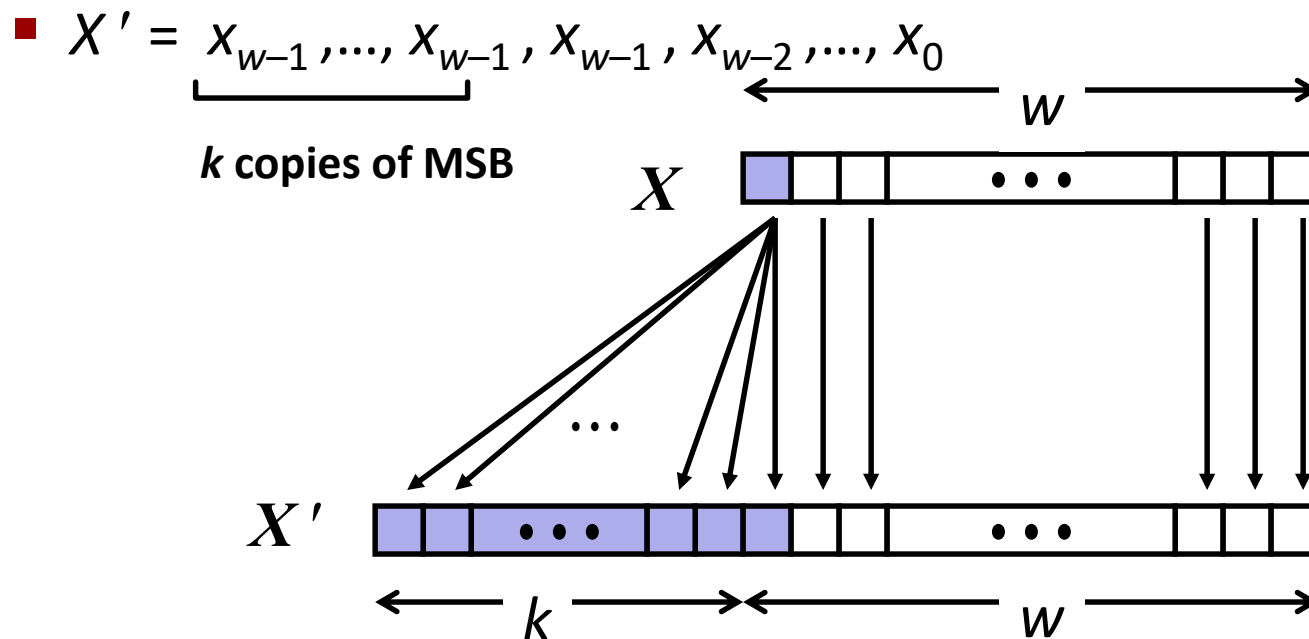
# 符号扩展 ( $w$ 位 $\rightarrow w+k$ 位)

## ■ 任务:

- 给定 $w$ 位的有符号整型数 $x$
- 将其转换为 $w+k$ 位的相同数值的整型数

## ■ 规则:

- 将最高有效位(符号位) $x_{w-1}$ 复制  $k$ 份:



# 符号扩展示例

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

	十进制	16进制	二进制
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

- 从短整数类型向长整数类型转换时，C自动进行符号扩展



# 截断 ( $w$ 位 $\rightarrow k$ 位, 并且 $k < w$ )

## ■ 无符号数的截断 ( $w$ 位 $\rightarrow k$ 位)

- 将无符号数对 $2^k$ 取模
- $X' = B2U([x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k$

## ■ 有符号数的截断 ( $w$ 位 $\rightarrow k$ 位)

- 与无符号数截断类似, 但要将余数按照补码进行解读
- $X' = U2T(B2U([x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k)$

# 总结:扩展、截断的基本规则

## ■ 扩展 (例如从short int 到int的转换)

- 无符号数: 填充0
- 有符号数: 符号扩展
- 结果都是明确的预期值

## ■ 截断 (例如从unsigned 到unsigned short的转换)

- 无论有/无符号数: 多出的位 (高位) 均被截断
- 结果重新解读
- 无符号数: 相当于求模运算
- 有符号数: 与求模运算相似
- 对于小整数, 结果是明确 (正确) 的预期值

# 本章目录: 位、字节 和 整型数

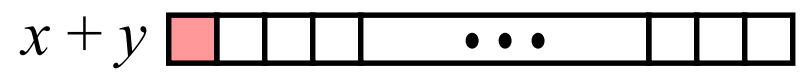
- 信息的位表示
- 位级运算
- **整型数**
  - 表示: 无符号数和有符号数
  - 无符号数和有符号数的转换
  - 扩展、截断
  - **整数运算: 加、非、乘、移位**
- 内存、指针、字符串表示
- 总结

# 无符号数加法

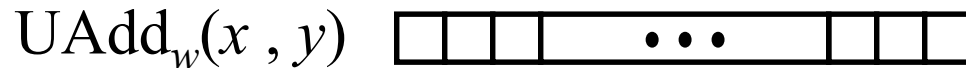
操作数:  $w$  位



真实和:  $w+1$  位



丢弃进位:  $w$  位



## ■ 标准加法功能

- 忽略进位输出

## ■ 模数加法：相当于增加一个模运算

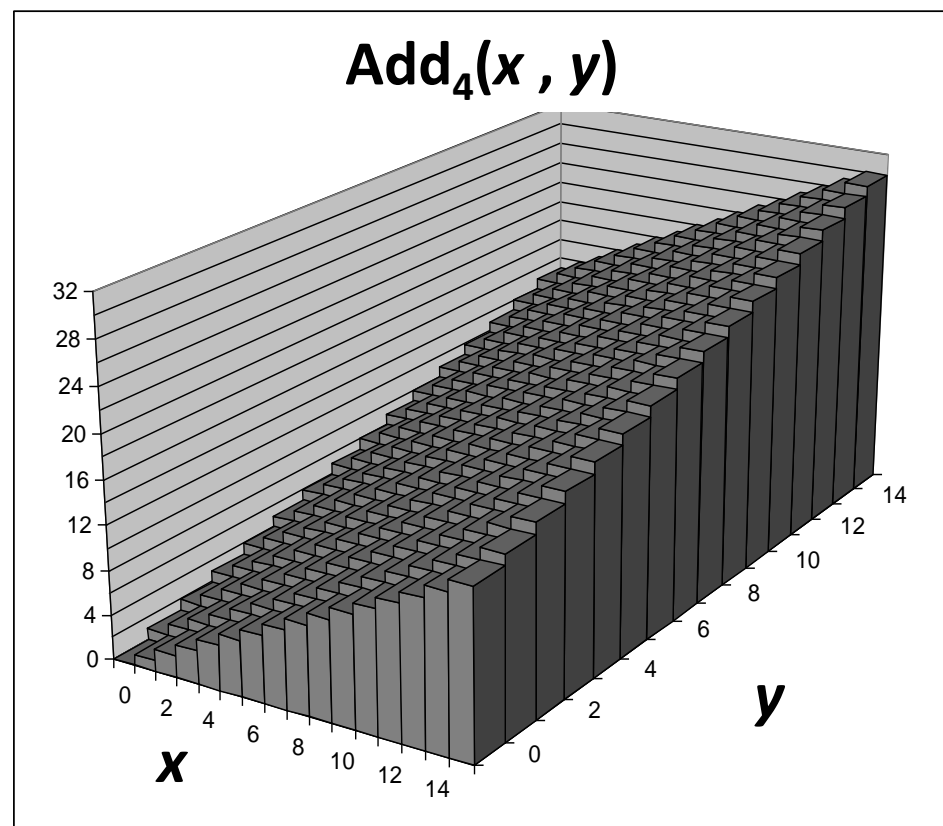
$$s = \text{UAdd}_w(x, y) = (x + y) \bmod 2^w$$

$$\text{UAdd}_w(x, y) = \begin{cases} x + y & x + y < 2^w \\ x + y - 2^w & x + y \geq 2^w \end{cases}$$

# 整数加法可视化示意图（正常的整数加法）

## ■ 整数加法

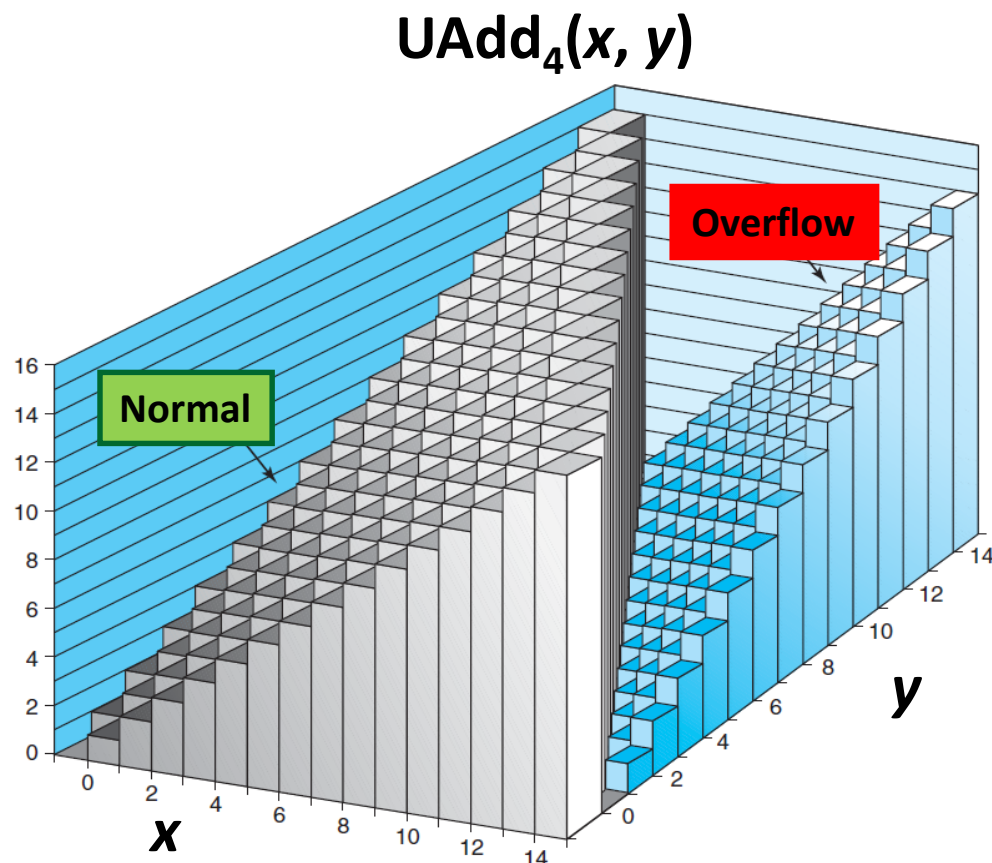
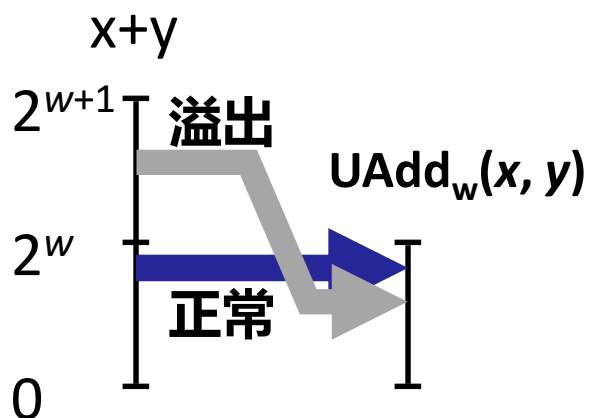
- 4-bit 整型数  $x, y$
- 计算真实值  $\text{Add}_4(x, y)$
- 和随  $x$  和  $y$  线性增加
- 表面为斜面形
- 正常加法可以有五位



# 无符号数加法可视化示意图

## ■ 数值面有弯折（非饱和运算—不单调）

- 当真实和  $\geq 2^w$  时溢出
- 最多溢出一次

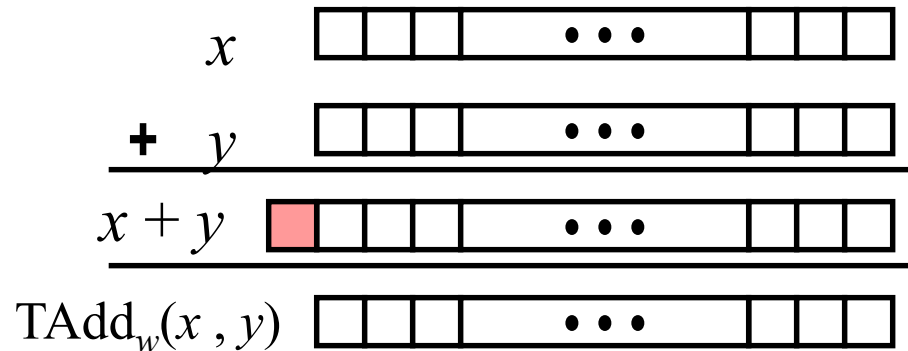


# 补码加法（其实CPU不知道数是有/无符号）

操作数:  $w$  位

真实和:  $w+1$  位

丢弃进位:  $w$  位



## ■ TAdd 和 UAdd 具有完全相同的位级表现

- C语言中有符号数(补码)与无符号数加法:

```
int s, t, x, y;
```

```
s = (int) ((unsigned) x + (unsigned) y);
```

```
t = x + y
```

- 将一定会有  $s == t$

# 补码加法(Tadd)

## ■ 功能

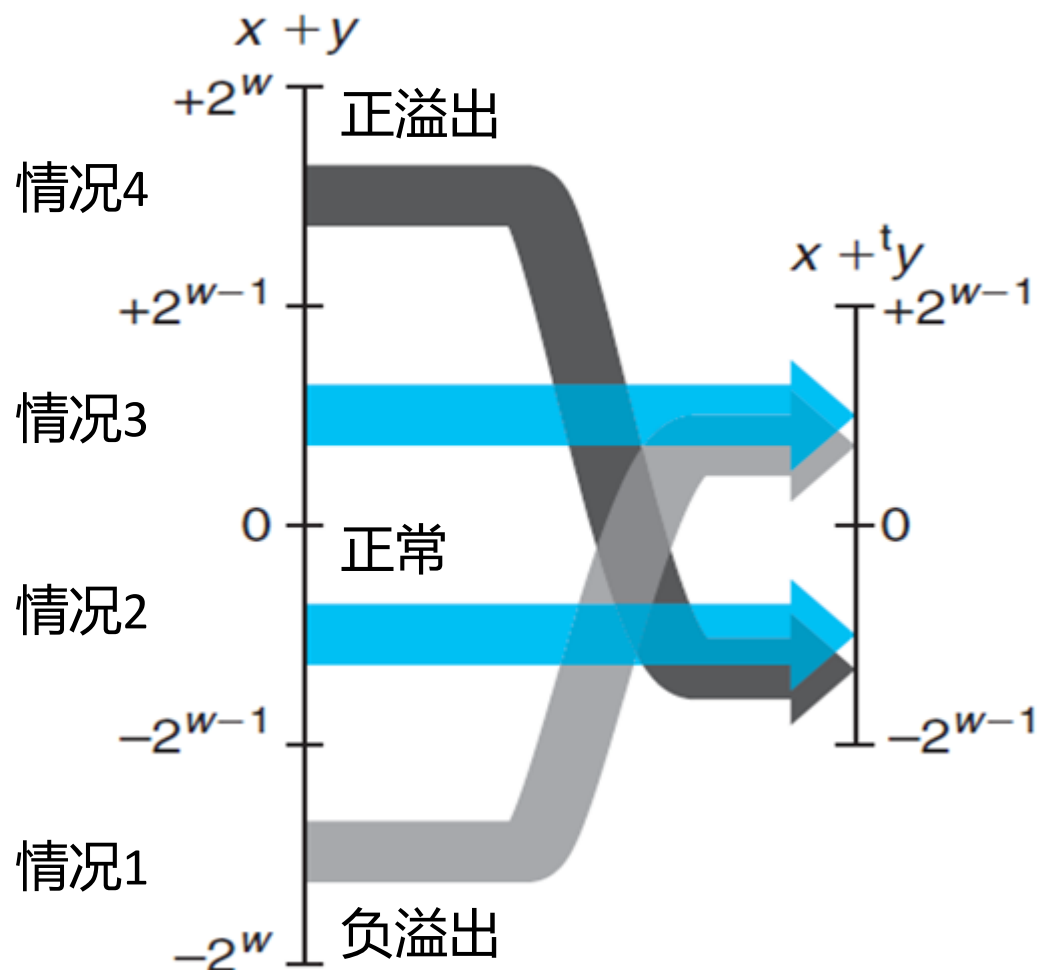
- 真实和需要 $w+1$ 位
- 丢弃最高有效位 (MSB)
- 将剩余的位视作补码 (整数)

$$TAdd(x, y) = \begin{cases} x + y - 2^w, & TMax_w < x + y & \text{正溢出} \\ x + y, & TMin_w \leq x + y \leq TMax_w & \text{正常} \\ x + y + 2^w, & x + y < TMin_w & \text{负溢出} \end{cases}$$

- 判断补码是否超范围 (Xh表示x的符号位) :
  - $Xh \neq Yh$  或者  $Xh == Yh == Zh$  则正常 ( $Z = X + Y$ )
  - $Xh == Yh \neq Zh$  则溢出



# 补码加法(Tadd)的溢出问题



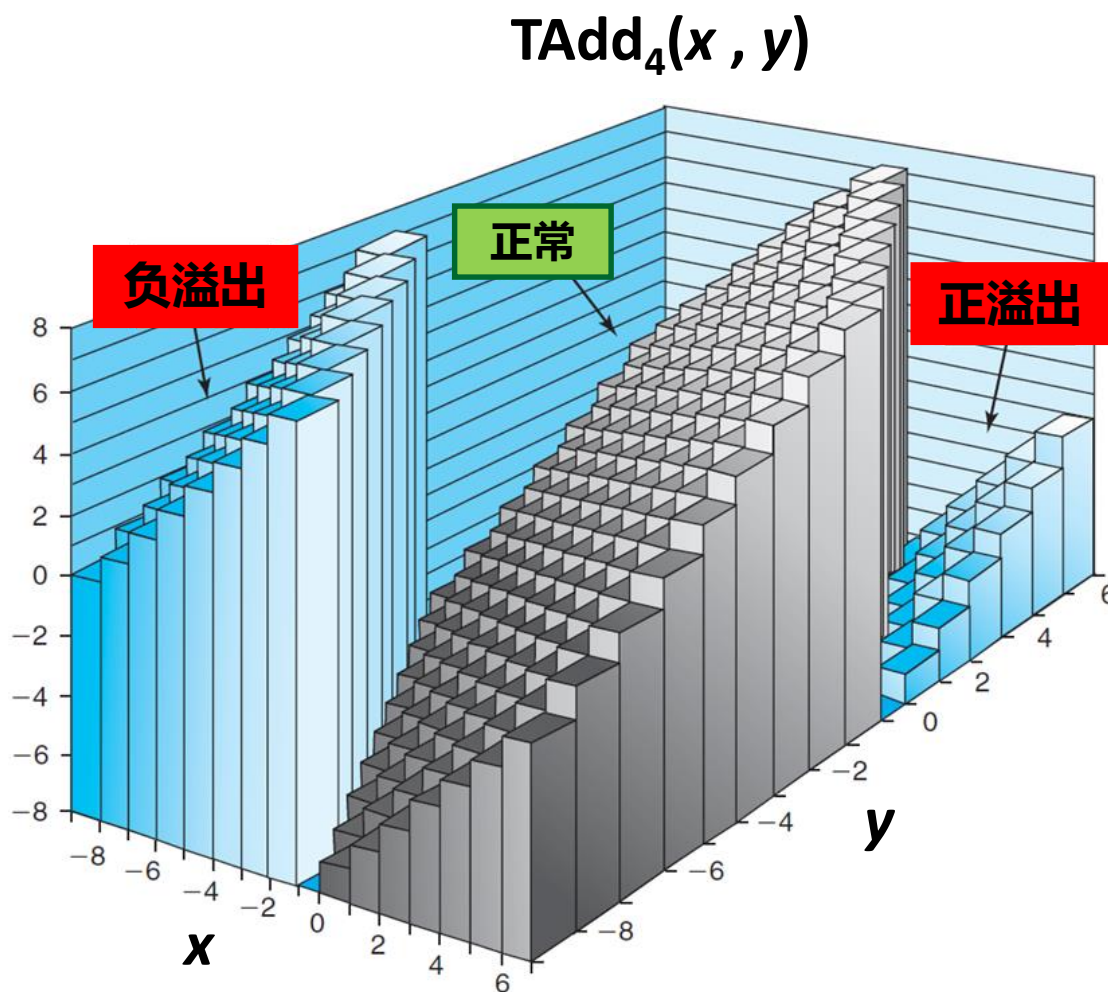
# 补码加法可视化示意图

## ■ 数值

- 4位补码
- 数值范围 $-8 \sim +7$

## ■ 弯折——溢出

- $x+y \geq 2^{w-1}$  时
  - 变成负数
- $x+y < -2^{w-1}$ 
  - 变成正数



# 乘法

- 目标: 计算 $w$ 位的两个数 $x$ 和 $y$ 的乘积
  - 有符号数或者无符号数
- 乘积的精确结果最多可能**超过  $w$  位**
  - 乘积的无符号数最多可达  $2w$  位
    - 结果范围:  $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - 补码的最小值 (负数)最多需要 $2w-1$  位
    - 结果范围:  $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
  - 补码最大值(正数)最多需要 $2w$  位——值为  $(TMin_w)^2$ 
    - 结果范围:  $x * y \leq (2^{w-1})^2 = 2^{2w-2}$
- 为获得精确结果可扩展乘积的字长
  - 在需要用软件方法完成, 例如: 算术程序包“arbitrary precision”

备注: 1)  $2^{2w}$ 需要 $2w+1$ , 由于后面有减法, 所以不超过 $2w$ , 另一方面:  $2^{2w} - 2^{w+1} + 1 - 2^w (w-1) > 0$ , 所以至少有 $2w$ 位。

2)  $-2^{2w-2}$  需要 $2w-1$ 位, 这是补码的极端最小负数情况。

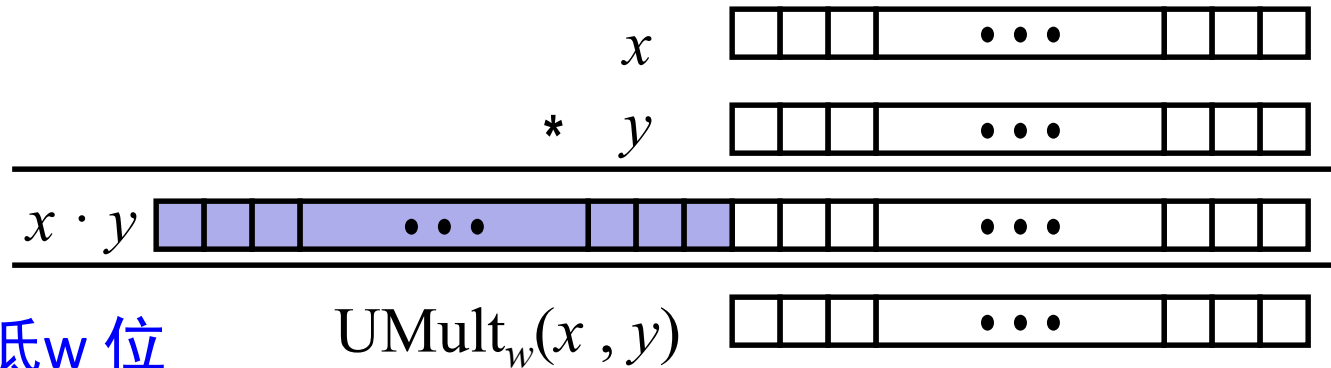
3) 负数相乘是正数,  $2^{2w-2}$  是1个1后面 $2w-2$ 个0, 同时由于正数, 所以第一位必须是0。

# C 语言的无符号数乘法

操作数:  $w$  位

真实乘积:  $2 * w$  位

丢弃  $w$  位: 只保留低  $w$  位



## ■ 标准乘法功能

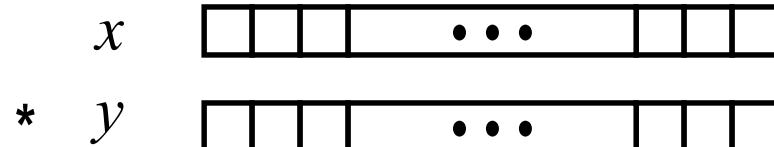
- 忽略高  $w$  位

## ■ 相当于对乘积执行了模运算

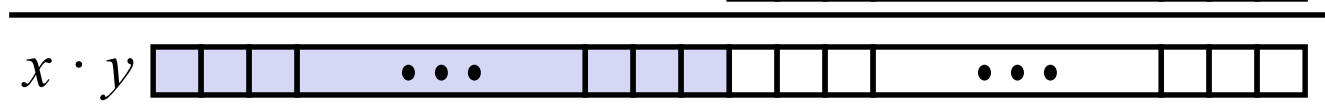
$$\text{UMult}_w(x, y) = (x \cdot y) \bmod 2^w$$

# C 语言的有符号数乘法

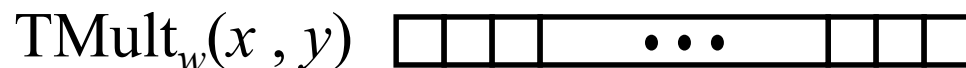
操作数:  $w$  位



真实乘积:  $2*w$ 位



丢弃  $w$  位: 保留低  $w$  位



## ■ 标准乘法功能

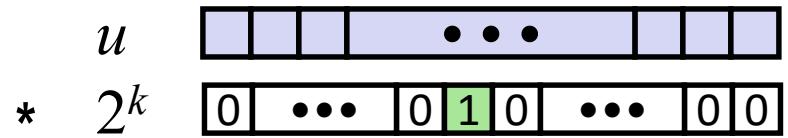
- 忽略高  $w$  位
- 有符号数乘、无符号数乘有不同之处
  - 乘积的符号扩展

# 用移位实现“乘以2的幂”

- 无论有符号数还是无符号数：

$u \ll k$  可得到  $u * 2^k$

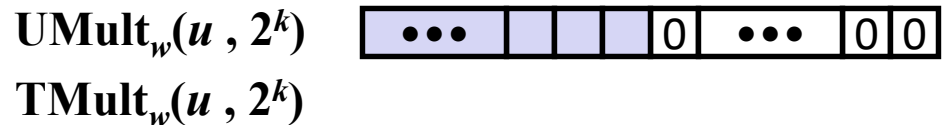
操作数:  $w$  位



真实乘积:  $w+k$  位



丢弃高  $k$  位: 保留低  $w$  位



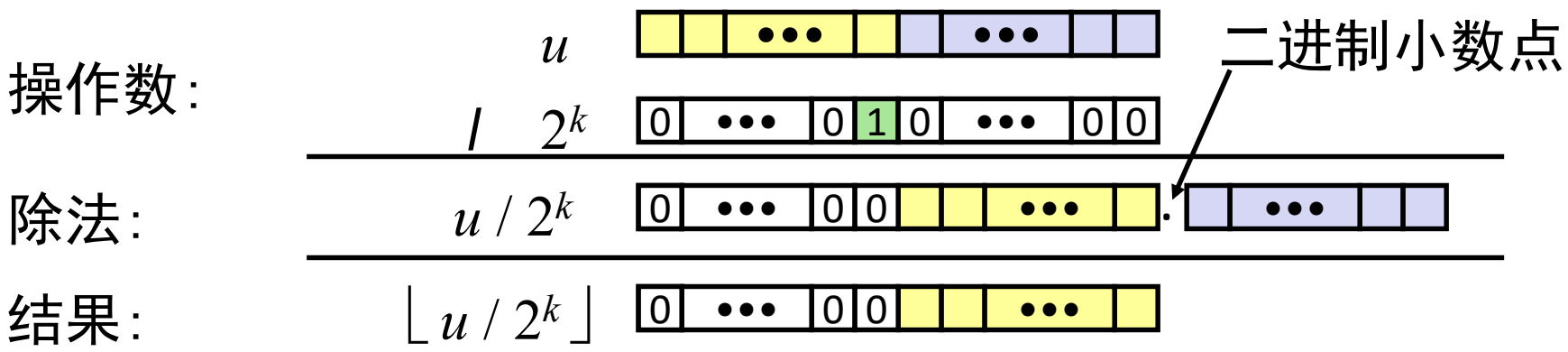
## ■ 示例

- $u \ll 3 \quad == \quad u * 8$
- $(u \ll 5) - (u \ll 3) \quad == \quad u * 24$
- 绝大多数机器，移位比乘法快
- 编译器自动生成基于移位的乘法代码

# 用移位实现无/有符号数 “除以2的幂”

## ■ 无符号数 “除以2的幂” 的商

- $u \gg k$  得到  $\lfloor u / 2^k \rfloor$
- 使用逻辑右移 **没有四舍五入**

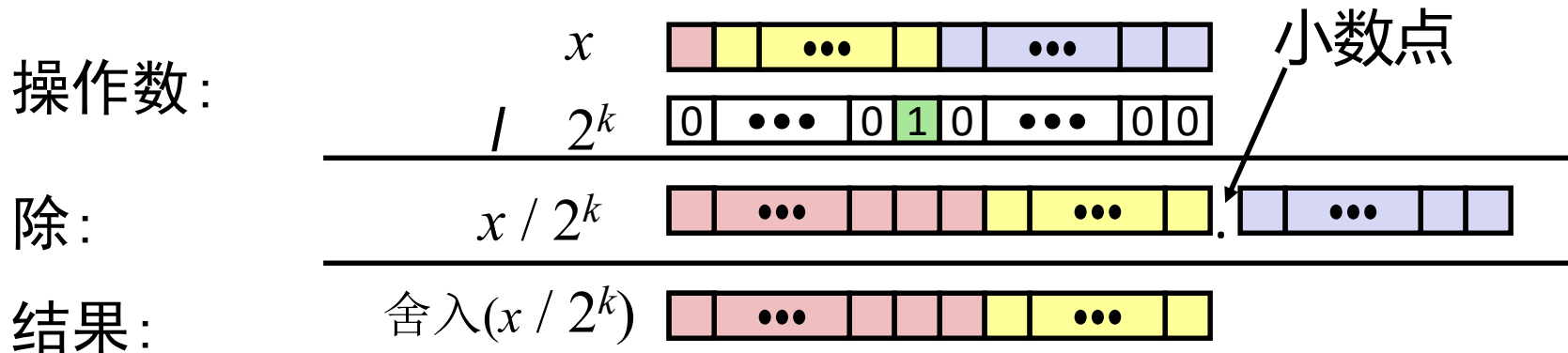


	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

# 用移位实现有符号数“除以2的幂”

## ■ 有符号数“除以2的幂”的商

- $x \gg k$  得到  $\lfloor x / 2^k \rfloor$
- 当  $x < 0$  时，舍入方向出错，**应该统一向零舍入**
- 注意使用算术右移，可以保留符号



	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	11100010 01001001
y >> 4	-950.8125	-951	FC 49	11111100 01001001
y >> 8	-59.4257813	-60	FF C4	11111111 11000100

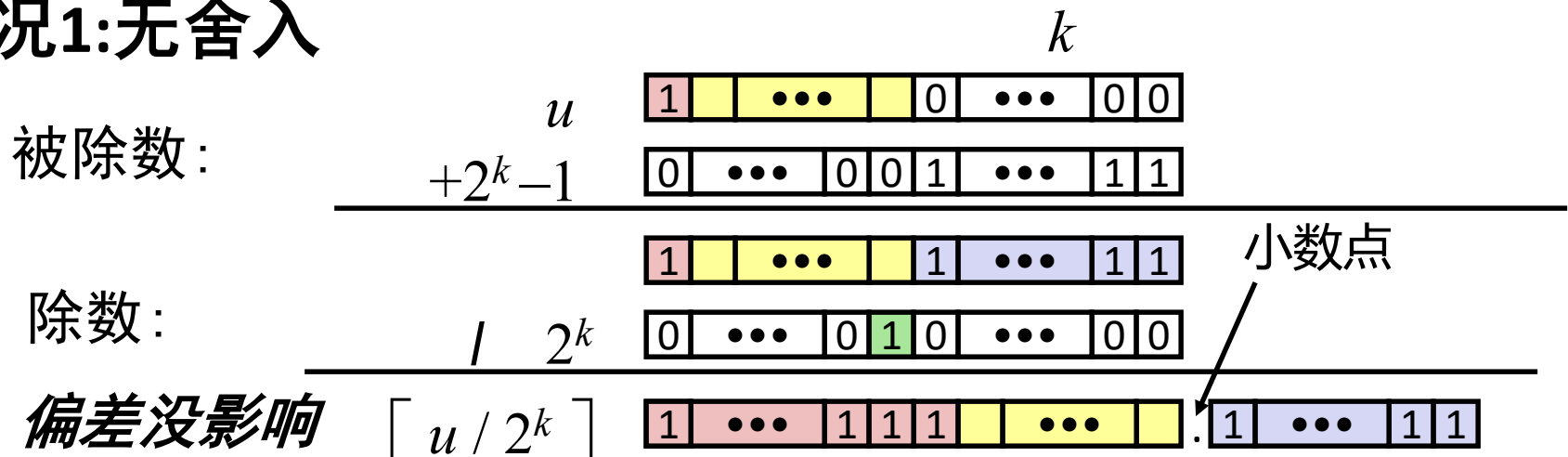


# 修正 2 的整数幂 除法

## ■ 负数除以2的整数幂的商

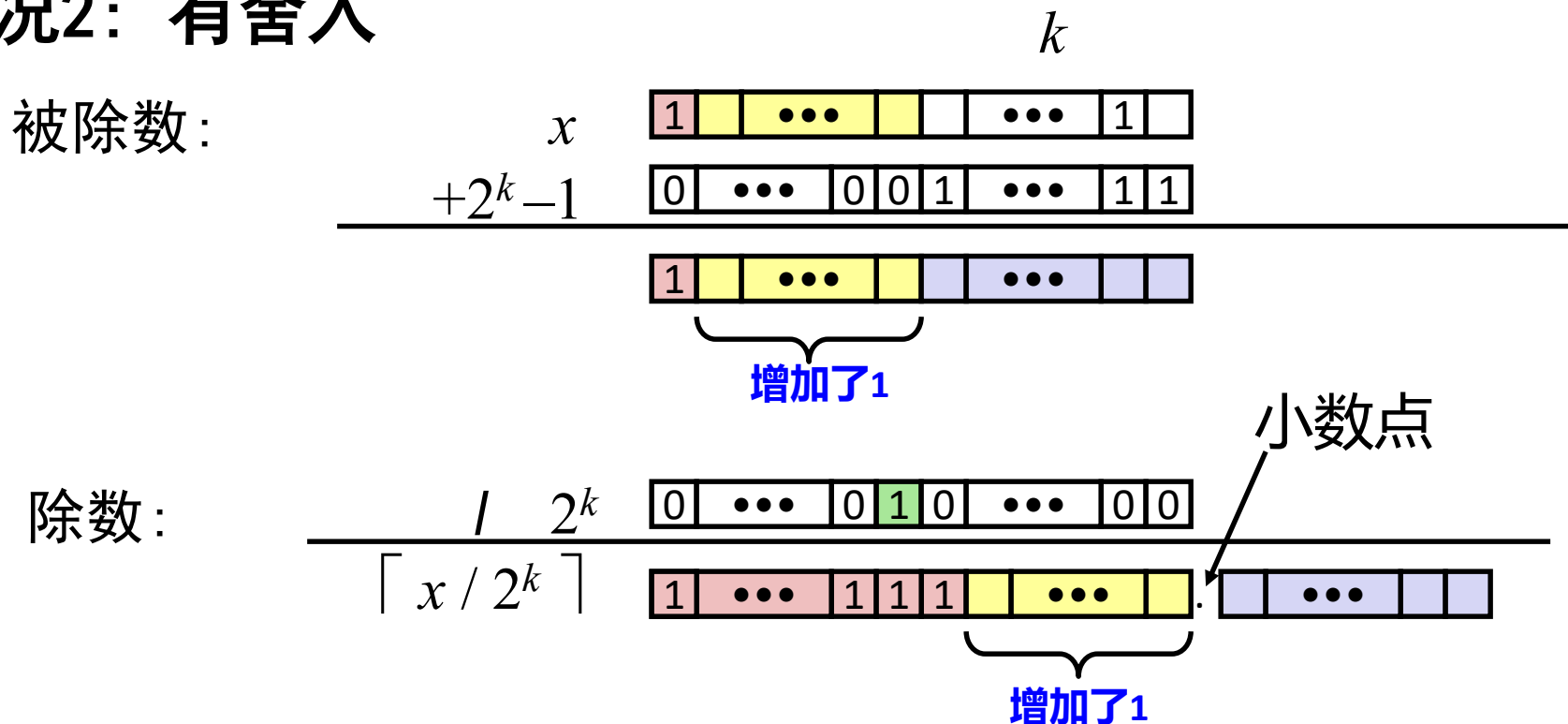
- 欲计算  $\lceil x / 2^k \rceil$  (向0舍入)
- 按  $\lfloor (x + 2^k - 1) / 2^k \rfloor$  计算
  - C表达式:  $(x + (1 \ll k) - 1) \gg k$
  - 被除数偏差趋向0

## 情况1:无舍入



# 修正 2 的整数幂 除法

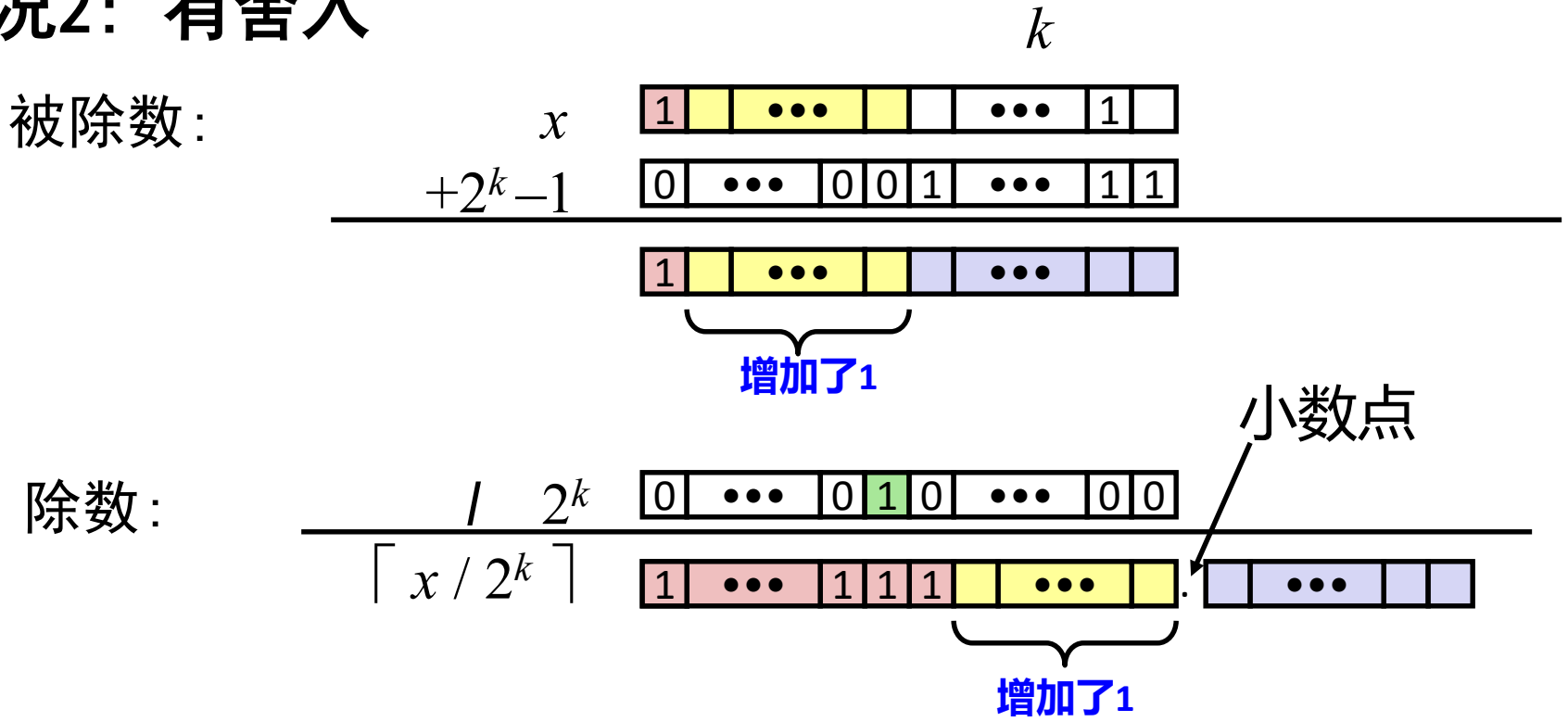
## 情况2：有舍入



偏差导致最终结果增加了 1

# 修正 2 的整数幂 除法

## 情况2：有舍入



最终效果:

	Division	Computed	Binary
y	-15213	-15213	11000100 10010011
y >> 1	-7606.5	-7606	11100010 01001010
y >> 4	-950.8125	-950	11111100 01001010
y >> 8	-59.4257813	-59	11111111 11000101

# 编译生成的有符号数除代码

## C 函数

```
long idiv8(long x)
{
    return x/8;
}
```

分两种情况：① 如果x是负数，先用test汇编指令进行判断，如果是负数，js指令会决定跳转到L4。对于负数除以 $2^k$ 的计算，在**向零舍入**时需先进行 $x+(2^k-1)$ ，再进行移k位操作，本例题中 $k=3$ ；② 如果x是正数，直接移位并返回，即执行L3。

## 编译生成的结果

```
testq %rax, %rax
js    L4
L3:
    sarq $3, %rax #sarq: 算术移位右移
    ret
L4:
    addq $7, %rax
    jmp  L3
```

注意：汇编代码需要阅读教材P136~P139

## 解释

```
if x < 0
    x += 7;
# Arithmetic shift
return x >> 3;
```

- 算术右移3位等价于除以8
- 无论x是正还是负，都实现了向零舍入

# 本章目录: 位、字节 和 整型数

- 信息的位表示
- 位级运算
- 整型数
  - 表示: 无符号数和有符号数
  - 无符号数和有符号数的转换
  - 扩展、截断
  - 整数运算: 加、非、乘、移位
  - 总结
- 内存、指针、字符串表示

# 算术运算: 基本规则

## ■ 加法:

**备注: 位级的运算就是指按位运算**

- 无/有符号数的加法: 正常加法后再截断, 位级的运算相同
- 无符号数: 加后对  $2^w$  求模
  - 数学加法 + 可能减去  $2^w$
- 有符号数: 加后对  $2^w$  求模, 使结果在合适范围
  - 数学加法 + 可能减去或加上  $2^w$

## ■ 乘法:

- 无/有符号数的乘法: 正常乘法后加截断操作, 位级运算相同
- 无符号数: 乘后对  $2^w$  求模
- 有符号数: 修改的乘后对  $2^w$  求模, 使结果在合适范围内

# 为何用无符号数？

## ■ 一定要知道隐含的转换规则， 否则不要用

### ■ 常见错误

```
unsigned i;  
for (i = cnt-2; i >= 0; i--)  
    a[i] += a[i+1];
```

### ■ 不易察觉的问题

```
#define DELTA sizeof(int)  
int i;  
for (i = CNT; i - DELTA >= 0; i -= DELTA)  
    ...
```

注意：无符号数和有符号数运算，都是按照无符号进行运算

# 巧用无符号数：向下计数

## ■ 使用无符号类型循环变量的适当方法

```
unsigned i;
for (i = cnt-2; i < cnt; i--) //循环执行cnt-1次
    a[i] += a[i+1]; //将数组从后往前累加
```

## ■ 参考Robert Seacord著《Secure Coding in C and C++》

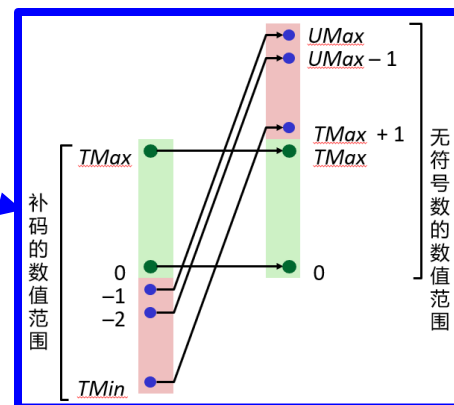
- C语言标准确保无符号数加法的行为与模运算类似

- $0 - 1 \rightarrow UMax$

## ■ 好方法

```
size_t i;
for (i = cnt-2; i < cnt; i--)
    a[i] += a[i+1];
```

- `size_t` 定义为32位或64位的无符号数，一般和机器字长相关
- 即便 `cnt = Umax` 也能很好工作
- 若 `cnt` 是有符号数，且值小于0，会如何？





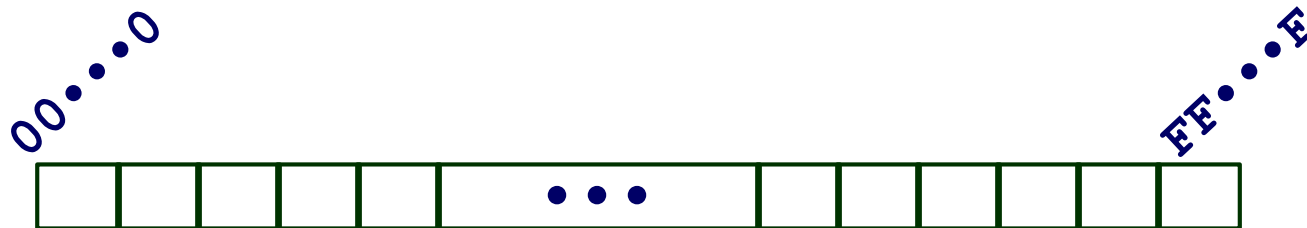
# 为何用无符号数？

- 需要进行模运算的时候，就用无符号数
  - 多精度的算术运算
- 用二进制位表示集合时，就用无符号数
  - 逻辑右移、无符号扩展

# 主要内容: 位、字节 和 整型数

- 信息的位表示
- 位级运算
- 整型数
  - 表示: 无符号数和有符号数
  - 无符号数和有符号数的转换
  - 扩展、截断
  - 整数运算: 加、非、乘、移位
  - 总结
- 内存、指针、字符串表示

# 面向字节的内存组织管理



## ■ 程序用地址来引用内存中的数据

- 内存可看做巨大的“数组”
  - 实际上不是这样，但不妨这样联想
- 地址就像这个“数组”的索引
  - 指针变量可保存地址数值

## ■ 注意:

- 操作系统为每个进程提供私有的地址空间
- 每个进程可访问自己地址空间中的内存数据，彼此不干扰。

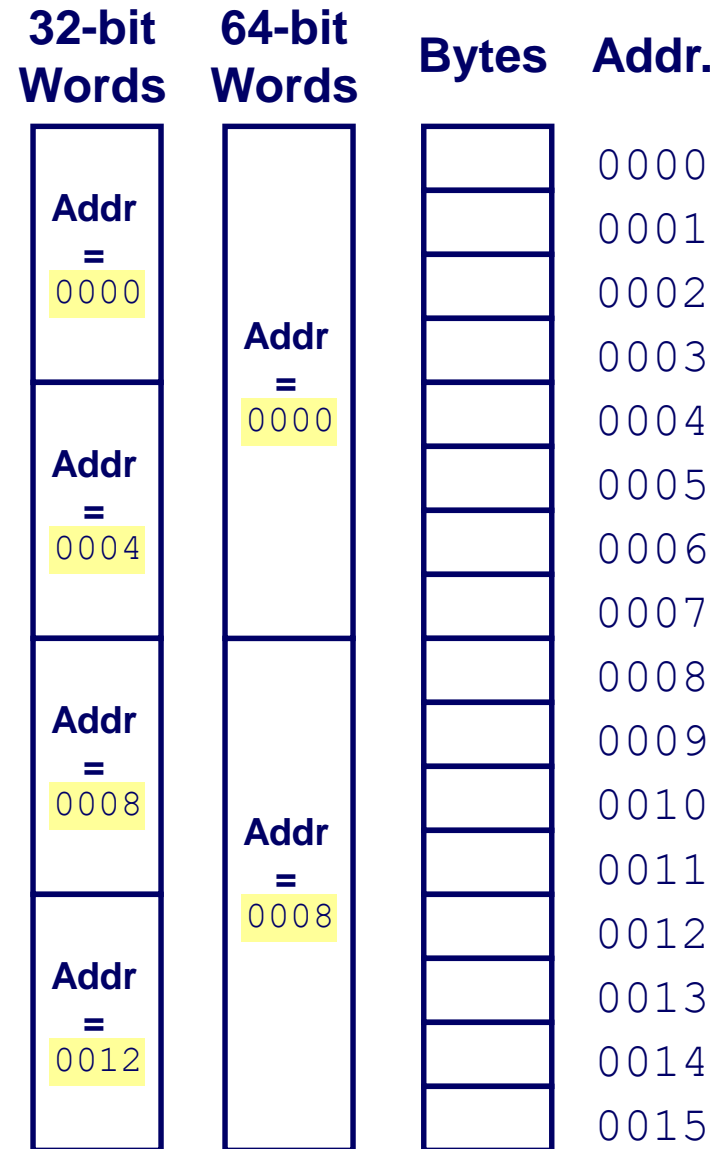
# 机器字

## ■ 任何机器都有一个“字长”

- 整型值数据的名义长度
  - 地址的名义长度
- 1985年intel 386 CPU开始,大多数机器使用32位 (4字节) 字长
  - 地址空间最大4GB ( $2^{32}$  bytes)
- 目前, 64位字长的机器是主流
  - 潜在地, 可以有18 EB (Exabytes) 的可寻址内存
  - 约 $18.4 \times 10^{18}$ 字节
- 机器依然支持多种数据格式
  - 字长的一部分或几倍长度
  - 始终是整数个字节

# 面向字的内存组织管理

- **地址：指定字节的位置**
  - 字中第一个字节的地址
  - 相邻字的地址相差 4 (32-bit) 或 8 (64-bit)



# C数据类型的典型大小(字节数)

C 数据类型	32位	64位	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
long double	–	–	10/16
pointer	4	8	8

# 字节序

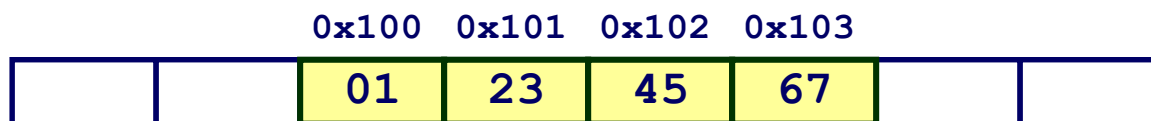
- 有多个字节的“字” (word), 其各个字节在内存中的排列
- 惯例
  - 大端序、大尾序 (Big Endian) : Sun, PPC Mac, Internet
    - 最低有效位字节的地址最高
  - 小端序、小尾序 (Little Endian) : x86、运行Android 的ARM 处理器、 iOS和Windows
    - 最低有效位字节的地址最低
- 双端序(Bi-Endian)
  - 机器可以配置成大端序或小端序
  - 很多新近的处理器的支持双端序

# 字节序示例

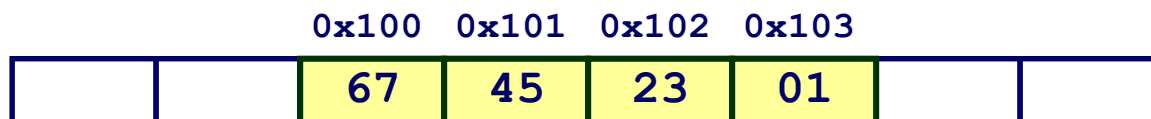
## ■ 示例

- 变量x 有4字节数值0x01234567
- 假定x的地址为 0x100

### 大端序



### 小端序





# 整型数的表示

十进制: 15213

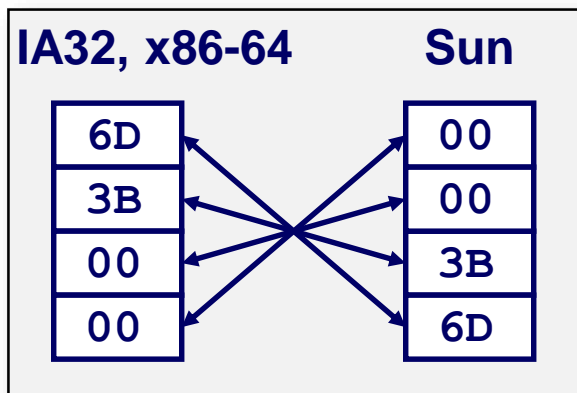
二进制: 0011 1011 0110 1101

16进制: 3 B 6 D

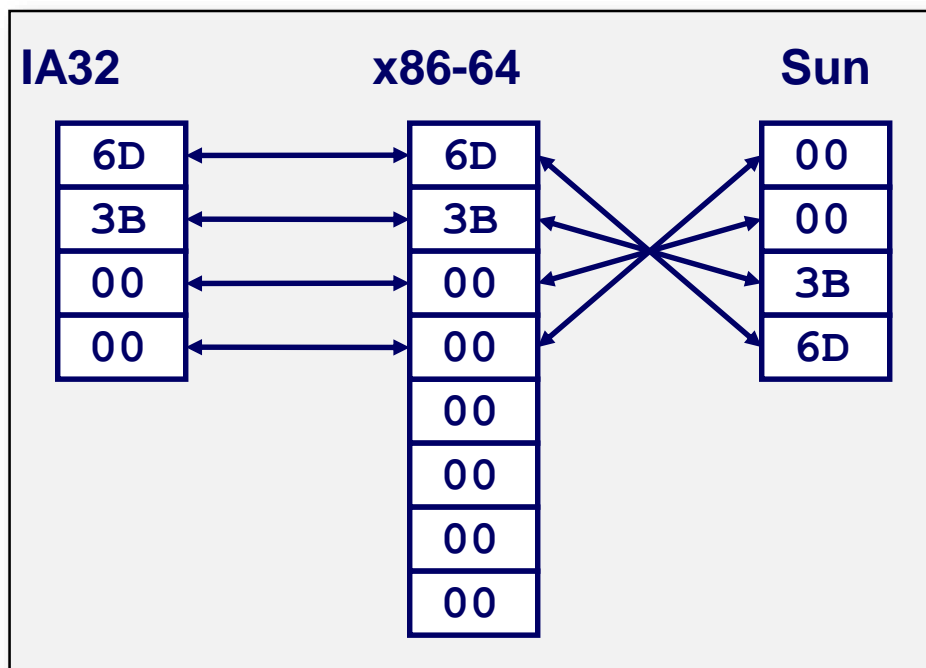
`int A = 15213;`

低

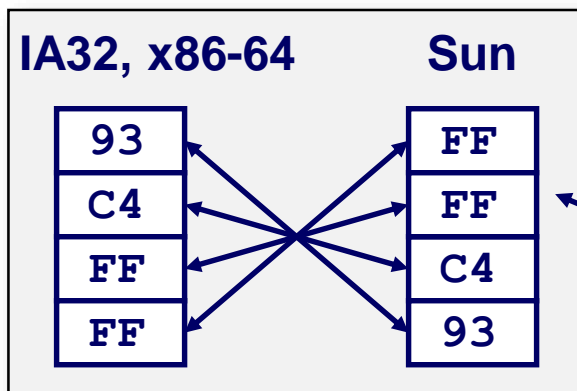
高



`long int C = 15213;`



`int B = -15213;`



补码表示

# 验证数的表示

## ■ 打印数据字节表示的程序代码

- 将指针转换成unsigned char \* 类型，从而按字节数组处理

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
    size_t i;
    for(i = 0; i < len; i++){
        printf("%p\t0x%.2x\n", start+i, start[i]);
        printf("\n");
    }
}
```

**printf 指令:**

**%p: 打印指针**

**%x: 16进制格式打印**

# show\_bytes 的执行实例

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

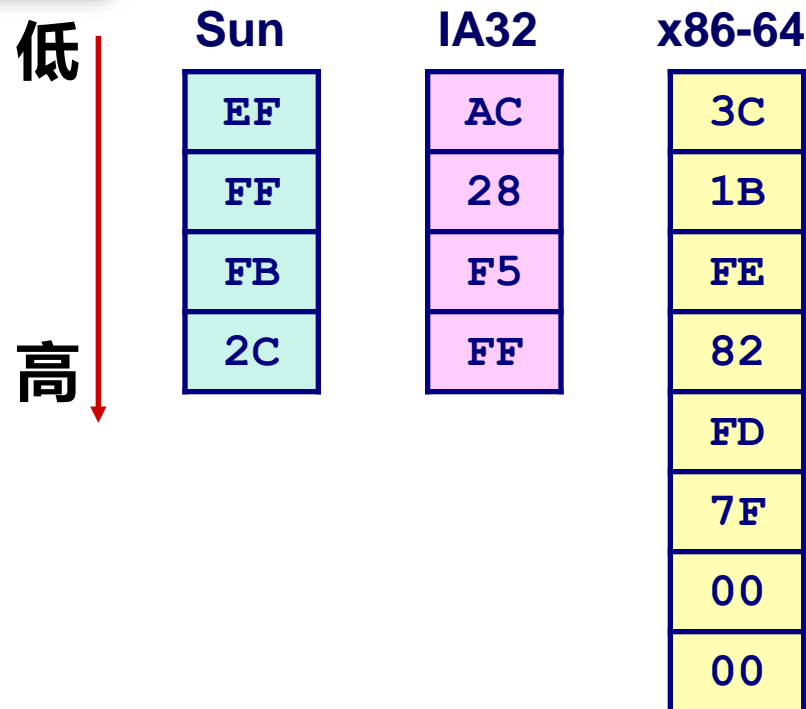
## Result (Linux x86-64):

```
int a = 15213;
0x7fffb7f71dbc    6d
0x7fffb7f71dbd    3b
0x7fffb7f71dbe    00
0x7fffb7f71dbf    00
```

<b>十进制:</b>	15213
<b>二进制:</b>	0011 1011 0110 1101
<b>16进制:</b>	3 B 6 D

# 指针的表示（打印p，即打印B的地址）

```
int B = -15213;  
int *P = &B;
```



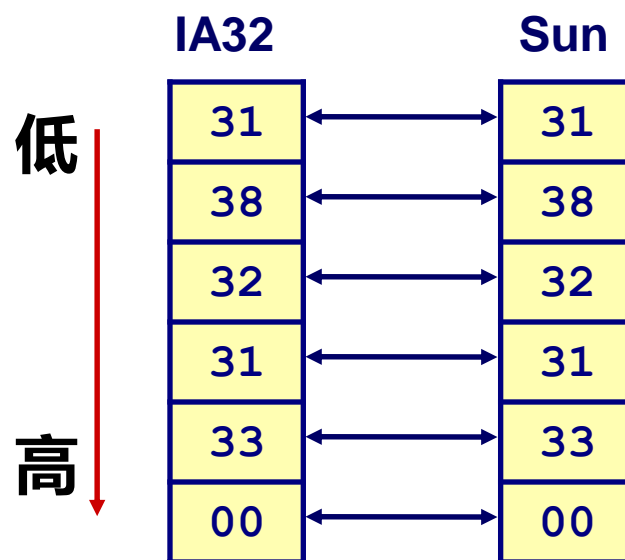
不同的编译器、机器会有不同的运行结果。  
甚至程序的每次运行结果都不同

# 字符串的表示

## ■ C字符串

- 用字符数组表示
- 每个字符都是ASCII格式编码
  - 字符集合的标准7位编码
  - 字符'0'的编码是 0x30
    - 数码  $i$  的编码是  $0x30+i$
- 字符串以\0结尾
  - 最后的字符 = 0
- 兼容性
  - 字节序不是个事!
  - 任何系统上都得到相同结果，与字节顺序和字大小规则无关

```
char S[6] = "18213";
```



***Enjoy!***