

第2章 信息的表示和处理

计算机系统基础

任课教师:

龚奕利

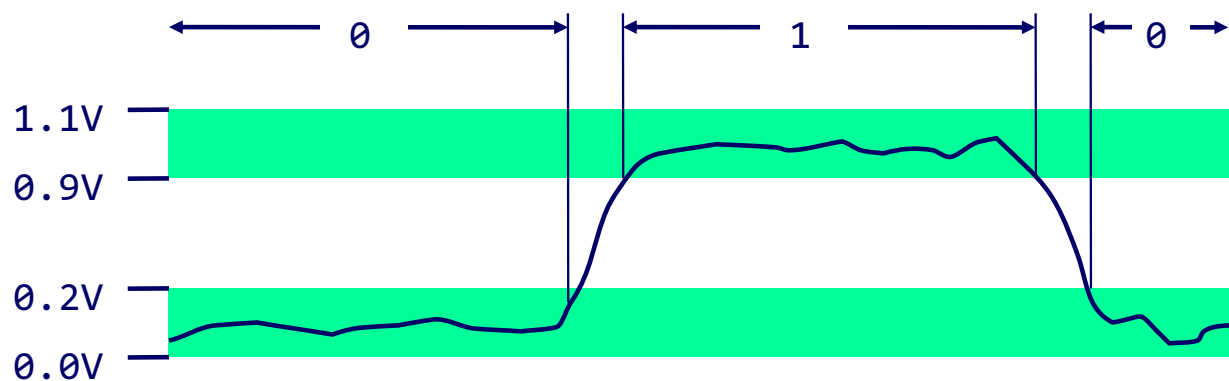
yiligong@whu.edu.cn

本节内容：位 **bit**、字节、整数和浮点数

- 用 **bit** 来表示信息
- 位级操作
- 整数
 - 表示：无符号 **unsigned** 和有符号 **signed**
 - 转换，强制类型转换
 - 扩展，截断
 - 加，取负，乘法，移位
 - 小结
- 在内存中的表示，指针，字符串
- 浮点数

凡事皆是 bit

- 每个 bit 是0或者1
- 以不同的方式编码/解读 bits
 - 计算机确定要做什么（指令）
 - 对数字、集合、字符串等进行表示和处理
- 为什么用 bit 呢？电气特性决定的
 - 存储双状态元素比较容易
 - 在有噪声和误差的线路上也能够可靠地传输



例如，能够以二进制方式计数

■ 基数为2的数的表示

- 430072_{10} 表示为 $110\ 1000\ 1111\ 1111\ 1000_2$
- 1.20_{10} 表示为 $1.0011001100110011[0011]..._2$
- 1.5213×10^4 表示为 $1.1101101101101_2 \times 2^{13}$

对字节值进行编码

■ 1字节 = 8 bits

- 二进制: $00000000_2 \sim 11111111_2$
- 十进制: $0_{10} \sim 255_{10}$
- 十六进制: $00_{16} \sim FF_{16}$
 - 以16为基的数的表示
 - 使用字符‘0’~‘9’, ‘A’~‘F’
 - 在C语言中将 $FA1D37B_{16}$ 写作
 - 0xFA1D37B
 - 0xfa1d37b

十六进制
十进制
二进制

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

数据表示示例

C数据类型	典型的32位系统	典型的64位系统	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
指针	4	8	8

本节内容：位 **bit**、字节、整数和浮点数

- 用 **bit** 来表示信息
- 位级操作
- 整数
 - 表示：无符号 **unsigned** 和有符号 **signed**
 - 转换，强制类型转换
 - 扩展，截断
 - 加，取负，乘法，移位
 - 小结
- 在内存中的表示，指针，字符串
- 浮点数

布尔代数

■ George Boole于19世纪提出来的

■ 逻辑的代数表示

- 将 “True”编码为 1, “False”编码为0

与 And

- $A \& B = 1$ 当 $A=1$ 且 $B=1$

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

非 Not

- $\sim A = 1$ 当 $A=0$

\sim	
0	1
1	0

或 Or

- $A \mid B = 1$ 当 $A=1$ 或 $B=1$

\mid	0	1
0	0	1
1	1	1

异或 Exclusive-Or (Xor)

- $A \wedge B = 1$ 当 $A=1$ 或 $B=1$, 但不同时成立

\wedge	0	1
0	0	1
1	1	0

通用布尔代数

- 对位向量进行操作
 - 操作是按位进行的

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

- 保持布尔代数的所有属性

示例：集合的表示和操作

■ 表示

- 用宽度为 w 的向量来表示 $\{0, \dots, w-1\}$ 的子集

- $a_j = 1$ if $j \in A$

- 01101001 { 0, 3, 5, 6 }

- 76543210

- 01010101 { 0, 2, 4, 6 }

- 76543210

■ 操作

■ &	交	01000001	{ 0, 6 }
■	并	01111101	{ 0, 2, 3, 4, 5, 6 }
■ ^	对称差	00111100	{ 2, 3, 4, 5 }
■ ~	补	10101010	{ 1, 3, 5, 7 }

C 语言中的位级运算

■ C 语言中有运算 $\&$, $|$, \sim , \wedge

- 可应用于任何“整数”数据类型
 - `long`, `int`, `short`, `char`, `unsigned`
- 将参数看作位向量
- 对参数按位处理

■ 示例（`char` 数据类型）

- $\sim 0x41 \rightarrow 0xD6$
 - $\sim 00101001_2 \rightarrow 11010110_2$
- $\sim 0x00 \rightarrow 0xFF$
 - $\sim 00000000_2 \rightarrow 11111111_2$
- $0x69 \& 0x55 \rightarrow 0x41$
 - $01101001_2 \& 01010101_2 \rightarrow 01000001_2$
- $0x69 | 0x55 \rightarrow 0x7D$
 - $01101001_2 | 01010101_2 \rightarrow 01111101_2$

对比：C 语言中对逻辑运算

■ 与逻辑运算符对比

■ &&, ||, !

- 将0看作“False”
- 任何非0的值看作 “True”
- 总是返回0或者1
- 提前终止 (Early termination)

要小心 && vs. & (|| vs. |)...

■ 示例 (char数据类型)

- !0x41 → 0x00
- !0x00 → 0x01
- !!0x41 → 0x01

- 0x69 && 0x55 → 0x01
- 0x69 || 0x55 → 0x01
- p && *p (避免访问空指针)

移位运算

■ 左移: $x \ll y$

- 将位向量 x 左移 y 位
 - 在左边把多余的位丢掉
 - 右边补上0

■ 右移: $x \gg y$

- 将位向量 x 右移 y 位
 - 在右边把多余的位丢掉
- 逻辑右移
 - 在左边补 0
- 算术右移
 - 在左边补最高位的值

■ 未定义行为

- 移位量 < 0 或者 \geq 字长

Argument x	01100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument x	10100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

本节内容：位 **bit**、字节、整数和浮点数

- 用 **bit** 来表示信息
- 位级操作
- 整数
 - 表示：无符号 **unsigned** 和有符号 **signed**
 - 转换，强制类型转换
 - 扩展，截断
 - 加，取负，乘法，移位
 - 小结
- 在内存中的表示，指针，字符串
- 浮点数

整数编码

无符号数 **Unsigned**

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

补码

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

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

符号位
注意：权重为负

- C 语言的short大小为2字节

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

- 符号位

- 对于补码，最高位表示符号
 - 0 代表非负
 - 1 代表为负

补码编码示例

x = 430072: 0110 10001111 11111000
y = -430072: 1001 01110000 00001000

权重	430072		-430072	
1	0	0	0	0
2	0	0	0	0
4	0	0	0	0
8	1	8	1	8
16	1	16	0	0
32	1	32	0	0
64	1	64	0	0
128	1	128	0	0
256	1	256	0	0
512	1	512	0	0
1024	1	1024	0	0
2048	1	2048	0	0
4096	0	0	1	4096
8192	0	0	1	8192
16384	0	0	1	16384
32768	1	32768	0	0
65536	0	0	1	65536
131072	1	131072	0	0
262144	1	262144	0	0
-524288	0	0	1	-524288
和		430072		-430072

数值的范围

■ 无符号数值

■ $UMin = 0$

000...0

■ $UMax = 2^w - 1$

111...1

■ 补码数值

■ $TMin = -2^{w-1}$

100...0

■ $TMax = 2^{w-1} - 1$

011...1

■ 其他特殊值

■ -1

111...1

$w = 16$ 的特殊值

	十进制	十六进制	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

不同字长的特殊值

	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

- $|TMin| = TMax + 1$
 - 非对称的数值范围
- $UMax = 2 * TMax + 1$

■ C 语言编程

- `#include <limits.h>`
- 声明了一些常量，例如
 - `ULONG_MAX`
 - `LONG_MAX`
 - `LONG_MIN`
- 这些值是与平台相关的

无符号数与有符号数的数值

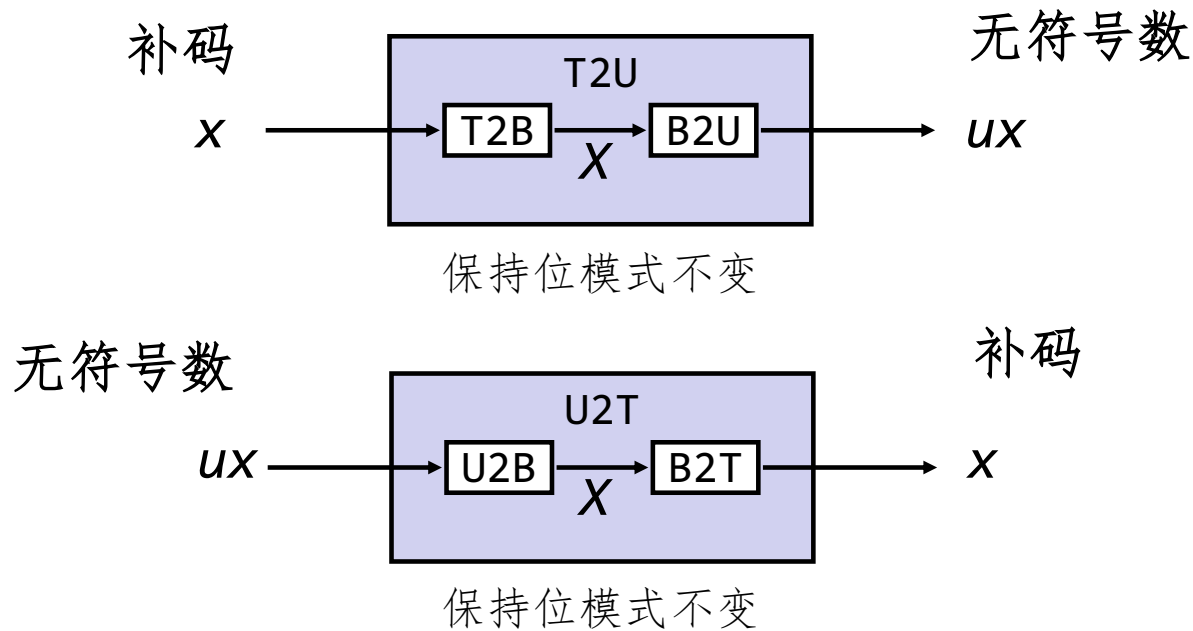
X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

- 等价性
 - 非负数值的编码相同
- 唯一性
 - 每个位模式表示不同的整数数值
 - 每个可表示的整数都有唯一的位编码
- \Rightarrow 反射
 - $U2B(x) = B2U^{-1}(x)$
 - 无符号数的位模式
 - $T2B(x) = B2T^{-1}(x)$
 - 补码整数的位模式

本节内容：位 **bit**、字节、整数和浮点数

- 用 **bit** 来表示信息
- 位级操作
- 整数
 - 表示：无符号 **unsigned** 和有符号 **signed**
 - 转换，强制类型转换
 - 扩展，截断
 - 加，取负，乘法，移位
 - 小结
- 在内存中的表示，指针，字符串
- 浮点数

有符号数与无符号数之间的映射



- 无符号数和补码数之间的映射
保持位模式不变，重新解释

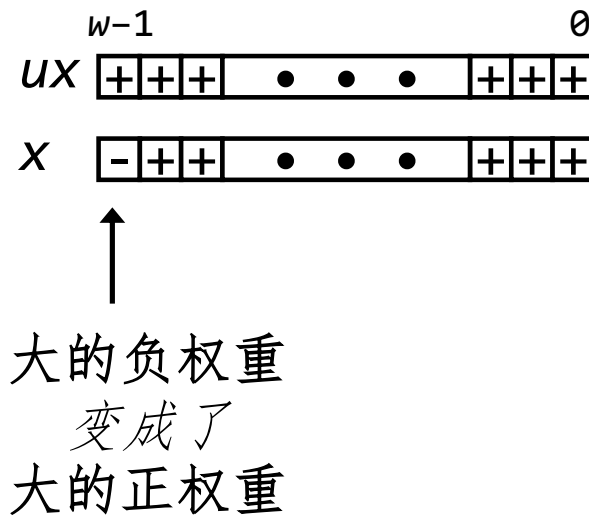
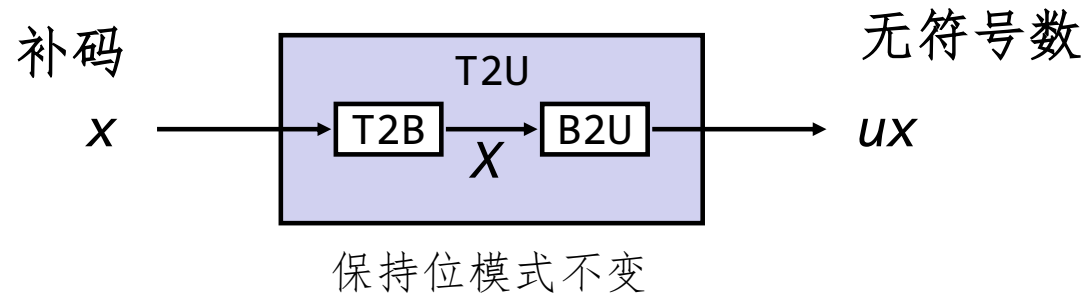
有符号数与无符号数之间的映射

位模式	有符号数		无符号数
0000	0	→ T2U → ← U2T ←	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	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

有符号数与无符号数之间的映射

位模式	有符号数		无符号数
0000	0	\longleftrightarrow =	0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8	\longleftrightarrow +/- 16	8
1001	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

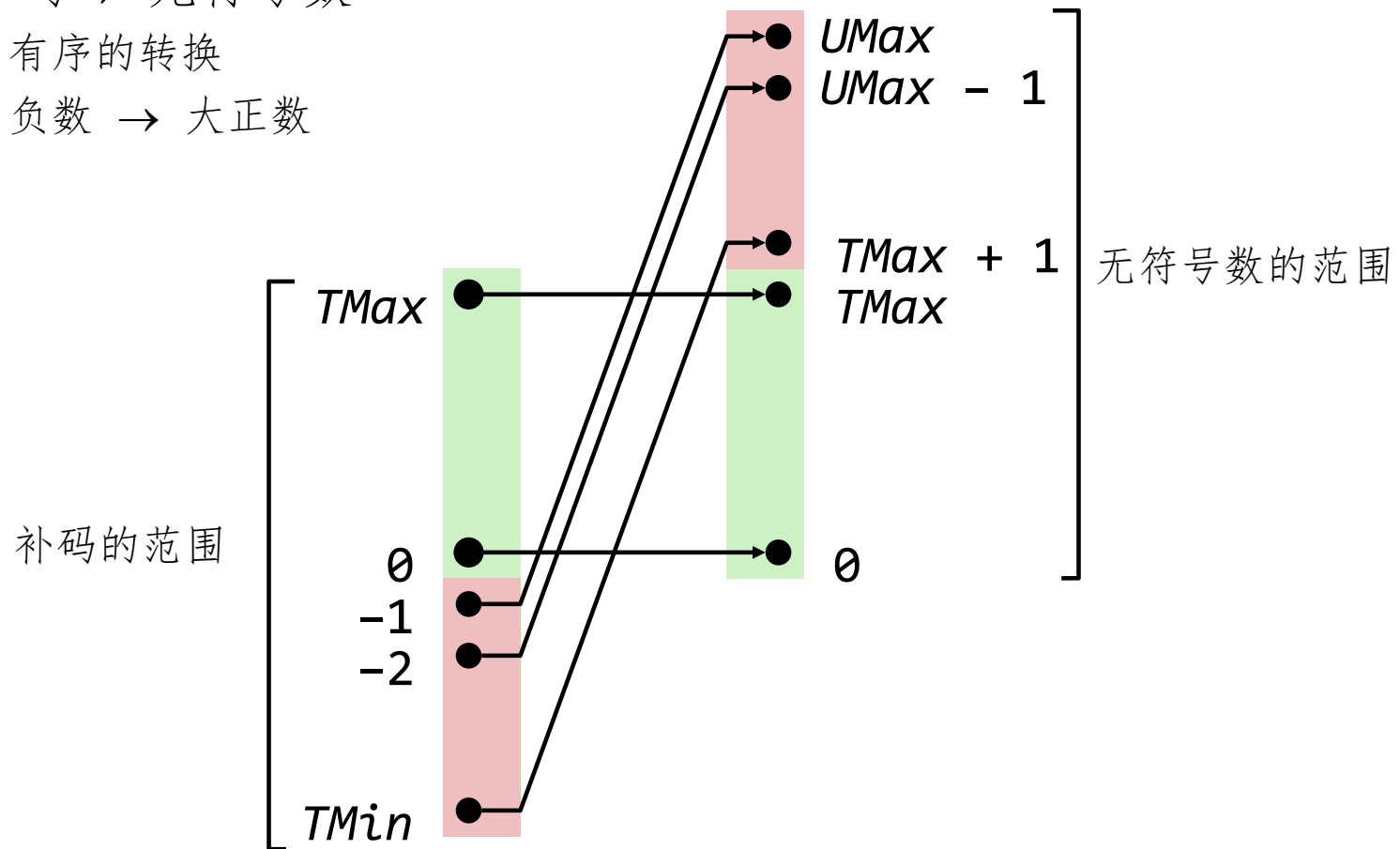
有符号数与无符号数的关系



有符号数与无符号数的转换

■ 补码 → 无符号数

- 有序转换
- 负数 → 大正数



C 语言中的有符号数 vs 无符号数

■ 常量

- 缺省的，被当作有符号整数
- 如果有后缀“U”，就被当作无符号整数

0U, 4294967259U

■ 强制类型转换

- 有符号&无符号数之间的明确的强制类型转换与 U2T 和 T2U 是一样的

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

- 赋值和过程调用中的隐式的强制类型转换

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

强制类型转换中的Surprises!

■ 表达式求值

- 如果在一个表达式中既有无符号数又有有符号数，
有符号数被隐式的强制类型转换为无符号数

- 包括比较操作 <, >, ==, <=, >=

- 对于 $W = 32$: **TMIN = -2,147,483,648** , **TMAX = 2,147,483,647**

■ 常数1	常数2	关系	表达式求值
0	0U	==	无符号数
-1	0	<	有符号数
-1	0U	>	无符号数
2147483647	-2147483647-1	>	有符号数
2147483647U	-2147483647-1	<	无符号数
-1	-2	>	有符号数
(unsigned)-1	-2	>	无符号数
2147483647	2147483648U	<	无符号数
2147483647	(int) 2147483648U	>	有符号数

小结：

有符号数  无符号数之间强制类型转换：基本规则

- 位模式保持不变
- 但是重新解读
- 可能会有意想不到的结果：加/减 2^w
- 表达式中如果同时包含有符号和无符号整数
 - `int` 会被强制类型转换为 `unsigned` !!

本节内容：位**bit**、字节、整数和浮点数

- 用 **bit** 来表示信息
- 位级操作
- 整数
 - 表示：无符号 **unsigned** 和有符号 **signed**
 - 转换，强制类型转换
 - 扩展，截断
 - 加，取负，乘法，移位
 - 小结
- 在内存中的表示，指针，字符串
- 浮点数

符号扩展

■ 任务：

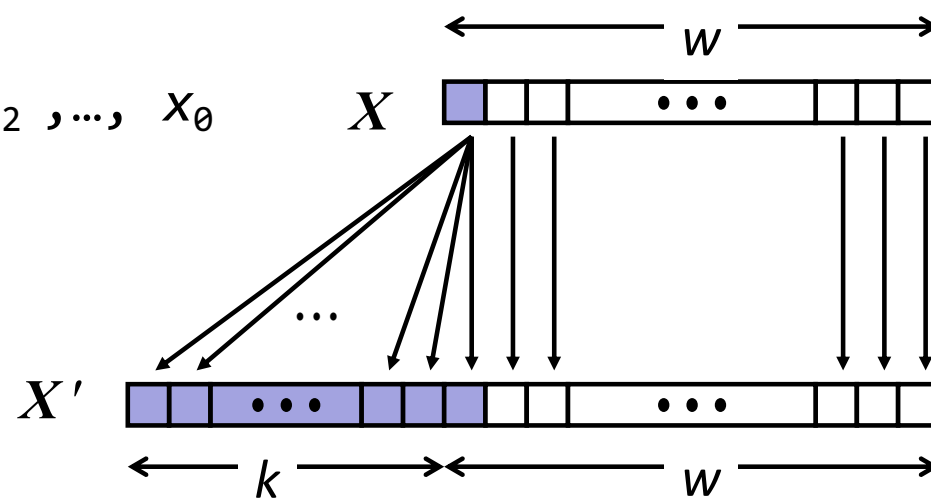
- 给定 w 位有符号整数 x
- 将之转换成数值相等的 $w+k$ 位整数

■ 规则：

- 符号位复制 k 次：

$$X = \underbrace{x_{w-1}, \dots, x_{w-1}}_{\text{最高位复制 } k \text{ 次}}, x_{w-1}, x_{w-2}, \dots, x_0$$

最高位复制 k 次



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	十六进制	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

- 将较小的整数数据类型转换成较大的整数数据类型
- C 语言自动进行符号扩展

小结：扩展，截断：基本规则

- 扩展（例如，`short` 扩展至 `int`）
 - `unsigned`: 在最前面加 0
 - `signed`: 符号扩展
 - 两种数据类型都得到与预期相符的结果
- 截断（例如，`unsigned` 变成 `unsigned short`）
 - `Unsigned/signed`: 高位截断
 - 结果重新解读
 - `Unsigned`: 等价于 `mod` 运算
 - `Signed`: 类似于 `mod` 运算
 - 对于较小的数字得到预期的行为

本节内容：位 **bit**、字节、整数和浮点数

- 用 **bit** 来表示信息
- 位级操作
- 整数
 - 表示：无符号 **unsigned** 和有符号 **signed**
 - 转换，强制类型转换
 - 扩展，截断
 - 加，取负，乘法，移位
 - 小结
- 在内存中的表示，指针，字符串
- 浮点数

无符号数加法

操作数： w 位

u

				•	•	•			
--	--	--	--	---	---	---	--	--	--

$+ v$

				•	•	•			
--	--	--	--	---	---	---	--	--	--

真实的和： $w+1$ 位

$u + v$

				•	•	•			
--	--	--	--	---	---	---	--	--	--

丢弃进位： w 位

$\text{UAdd}_w(u, v)$

				•	•	•			
--	--	--	--	---	---	---	--	--	--

- 标准的加法函数

- 忽略进位输出

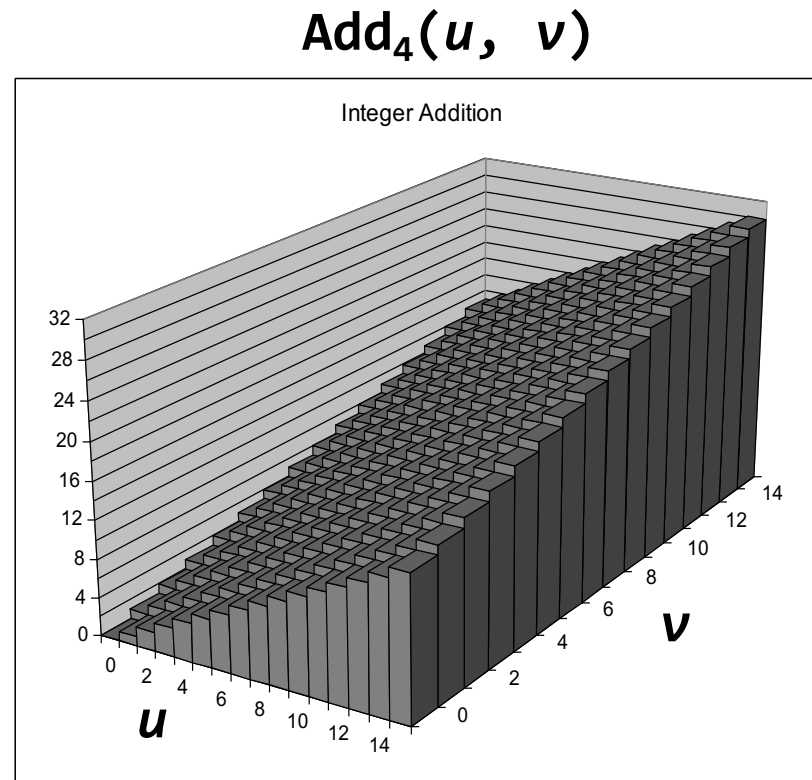
- 实现模运算

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

(数学上)整数加法的可视化表示

■ 整数加法

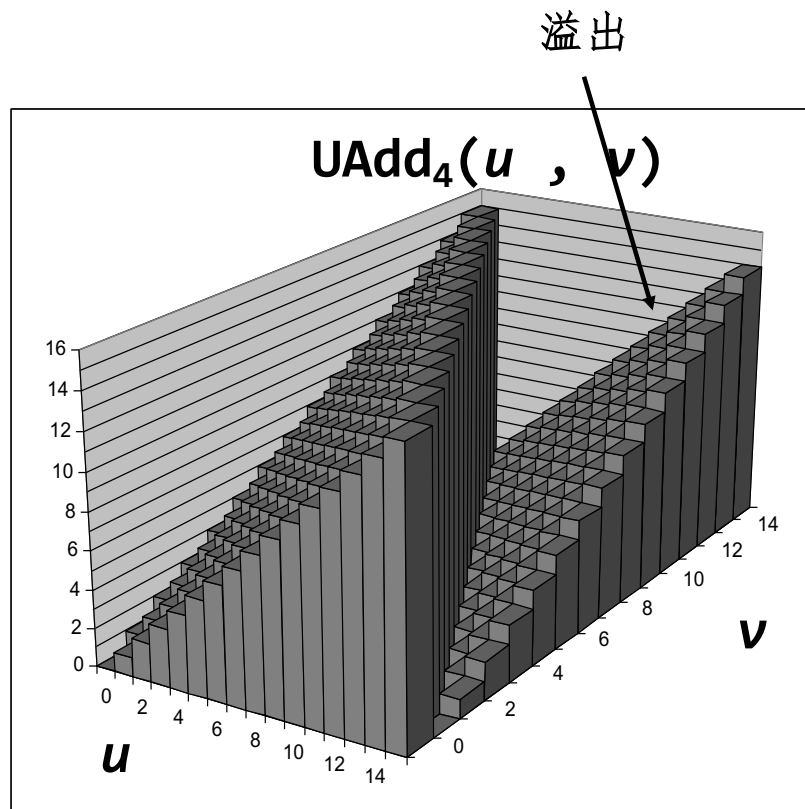
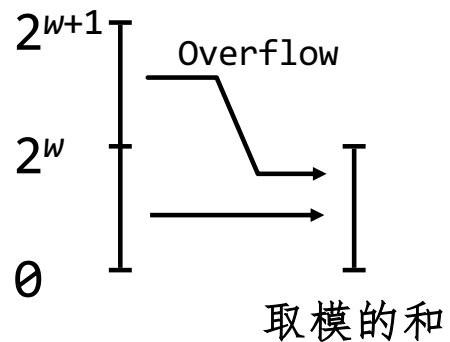
- 4位整数 u , v
- 计算真实的和 $\text{Add}_4(u, v)$
- 值随着 u 和 v 线性的增加
- 行成一个平的平面



无符号数加法的可视化表示

- 环绕回来
 - 如何真实的和 $\geq 2^w$
 - 最多一次

真实的和

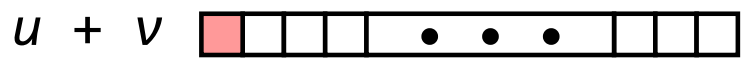


补码加法

操作数： w 位



真实的和： $w+1$ 位



丢弃进位： w 位



■ Tadd 和 Uadd有相同的位级行为

- C 语言中的 signed vs. unsigned 加法

```
int s, t, u, v;
```

```
s = (int) ((unsigned) u + (unsigned) v);
```

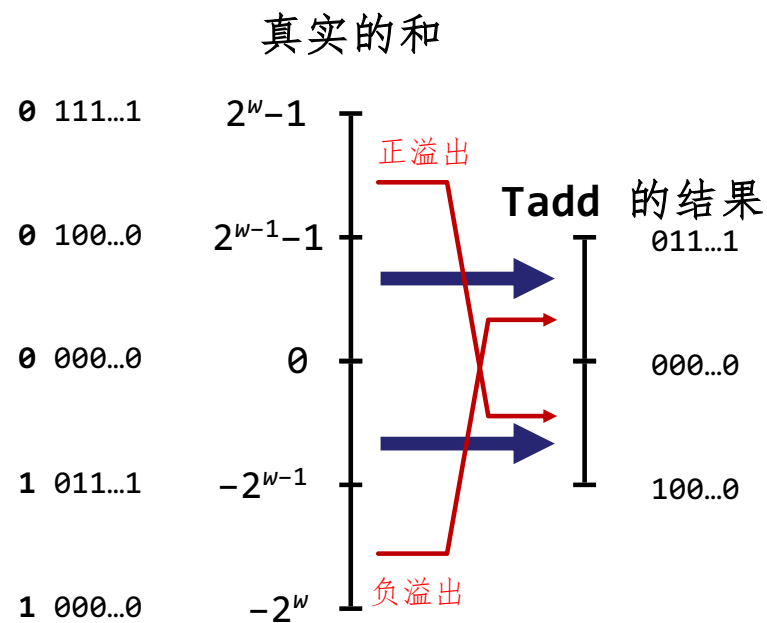
```
t = u + v
```

- 得到： $s == t$

Tadd 的溢出

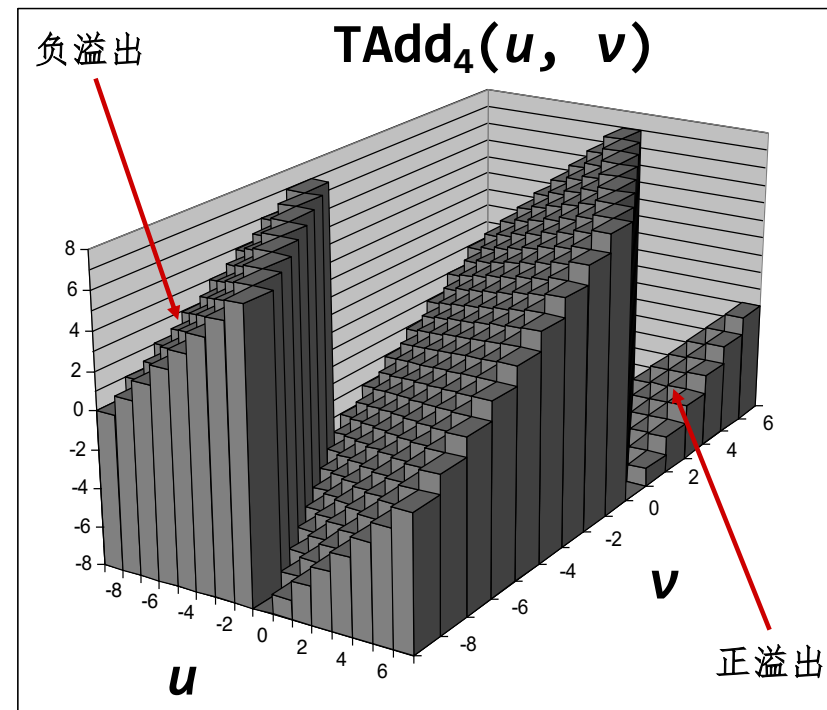
■ 功能

- 真实的和需要 $w+1$ 位
- 丢弃最高位 MSB
- 将剩余的位当作补码表示的整数



补码加法的可视化表示

- 值
 - 4位补码
 - 数值范围 $-8 \sim +7$
- 环绕回来
 - 如果和 $\geq 2^{w-1}$
 - 变成负数
 - 最多一次
 - 如果和 $< -2^{w-1}$
 - 变成正数
 - 最多一次

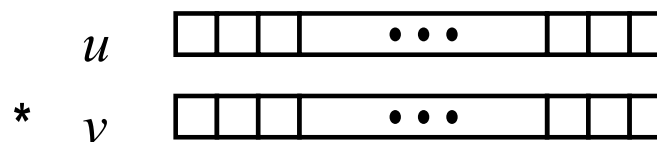


乘法

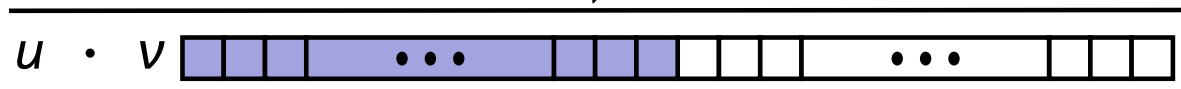
- 目标：计算 w 位数 x 和 y 的乘积
 - 可以是 **signed** 也可以是 **unsigned**
- 但是准确的结果可能会比 w 位更大
 - **Unsigned**: 最高可为 $2w$ 位
 - 结果范围: $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
 - 补码最小值（负数）: 最高可为 $2w-1$ 位
 - 结果范围: $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
 - 补码最大值（正数）: 最高可为 $2w$ 位，只可能是 $(TMin_w)^2$
 - 结果范围: $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
- 所以，要想保持准确的结果...
 - 需要在计算乘积时，扩展字长
 - 如果需要的话，可以通过软件完成
 - 例如，使用“任意精度（arbitrary precision）”数学包

C 语言中的无符号乘法

操作数： w 位



真实的乘积： $2*w$ 位



丢弃 w 位： w 位





- 标准乘法功能
 - 丢弃高 w 位
- 实现取模运算

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

C 语言中的有符号乘法

操作数: w 位


u 

$*$ v 

真实的乘积: $2 * w$ 位

$u \cdot v$ 

丢弃 w 位: w 位

$\text{TMult}_w(u, v)$ 

■ 标准乘法功能

- 丢弃高 w 位
- 有些结果和无符号乘法是不同的
- 低位是相同的

用移位来实现2的幂的乘法

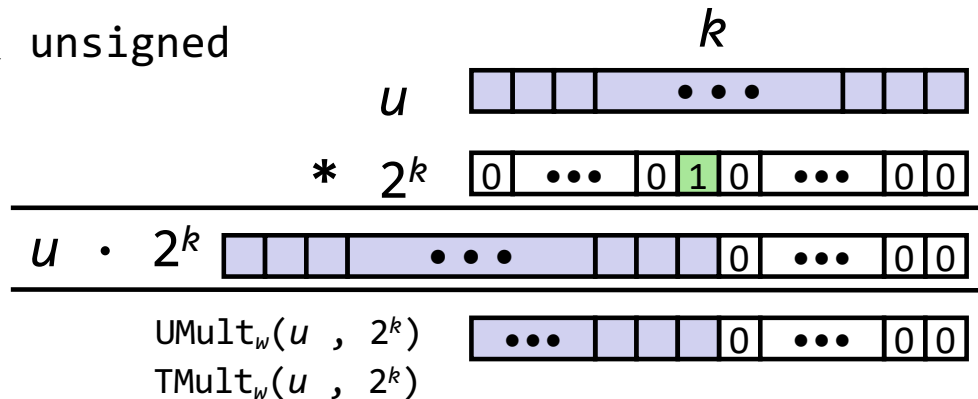
■ 运算

- $u \ll k$ 得到 $u * 2^k$
- u 可以是 signed 也可以是 unsigned

操作数: w 位

真实的乘积: $w + k$ 位

丢弃 k 位: w 位

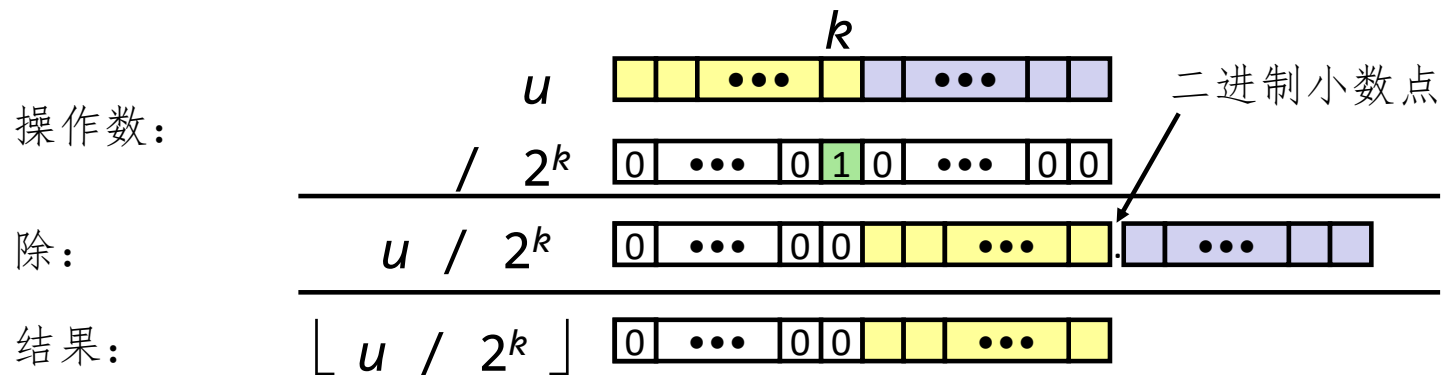


■ 示例

- $u \ll 3 == u * 8$
- $(u \ll 5) - (u \ll 3) == u * 24$
- 大多数机器的移位和加法比乘法要快
 - 编译器会自动生成这样的代码

用移位来实现无符号数除以2的幂

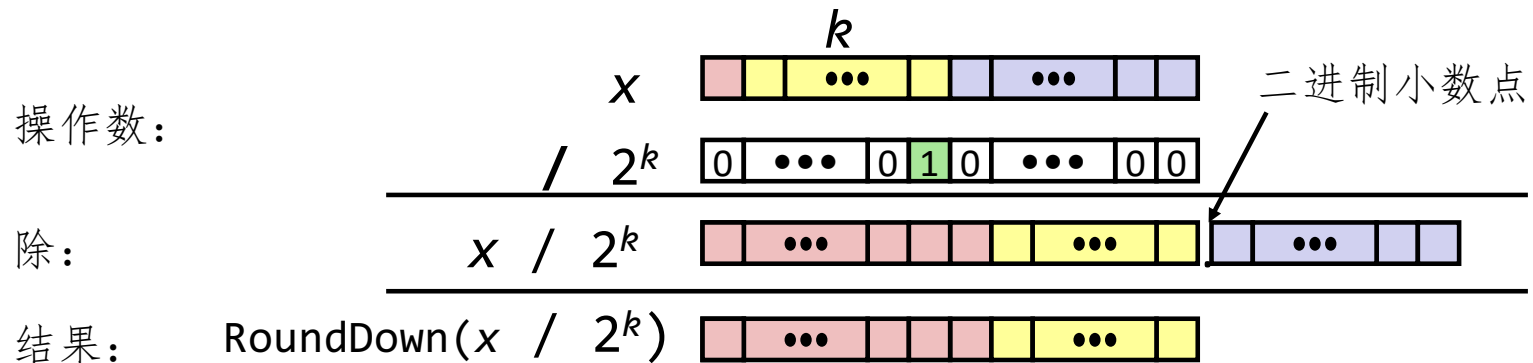
- 无符号数除以2的幂的商
 - $u \gg k$ 得到 $\lfloor u / 2^k \rfloor$
 - 使用逻辑移位



	除	计算结果	十六进制	二进制
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

用移位来实现有符号数除以2的幂

- 有符号数除以2的幂的商
 - $x \gg k$ 得到 $\lfloor x / 2^k \rfloor$
 - 使用算术右移
 - $u < 0$ 时，向错误的方向舍入

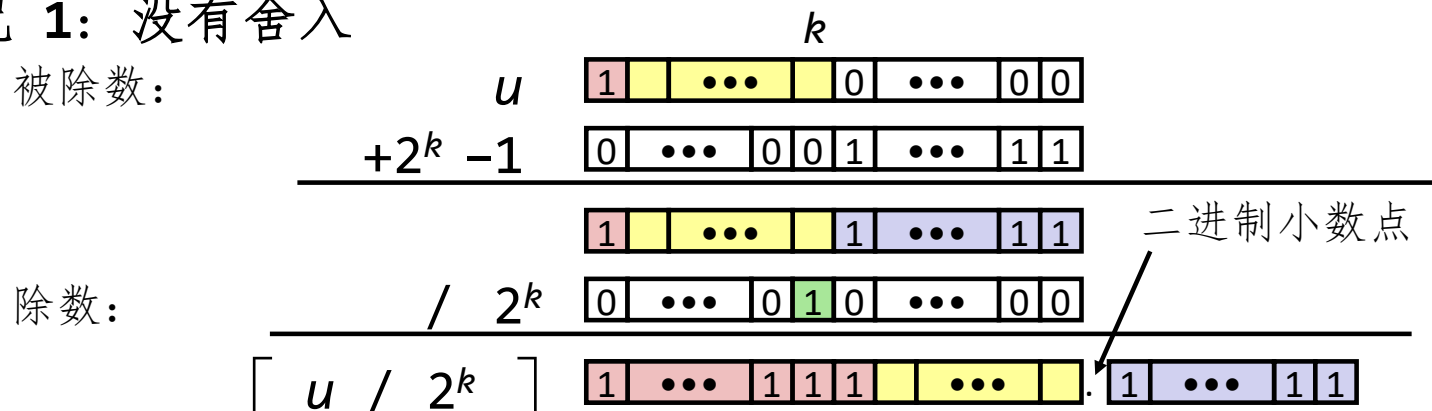


	除	计算结果	十六进制	二进制
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	11100010 01001001
y >> 4	-950.8125	-951	FC 49	11111100 01001001
y >> 8	-59.4257813	-60	FF C4	11111111 11000100

正确的除以2的幂

- 负数除以2的幂的商
 - 想要得到 $\lceil x / 2^k \rceil$ (向0舍入)
 - 通过 $\lfloor (x+2^k-1) / 2^k \rfloor$ 计算得到
 - C语言中: $(x + (1 \ll k) - 1) \gg k$
 - 被除数向 0 加偏移量

情况 1: 没有舍入

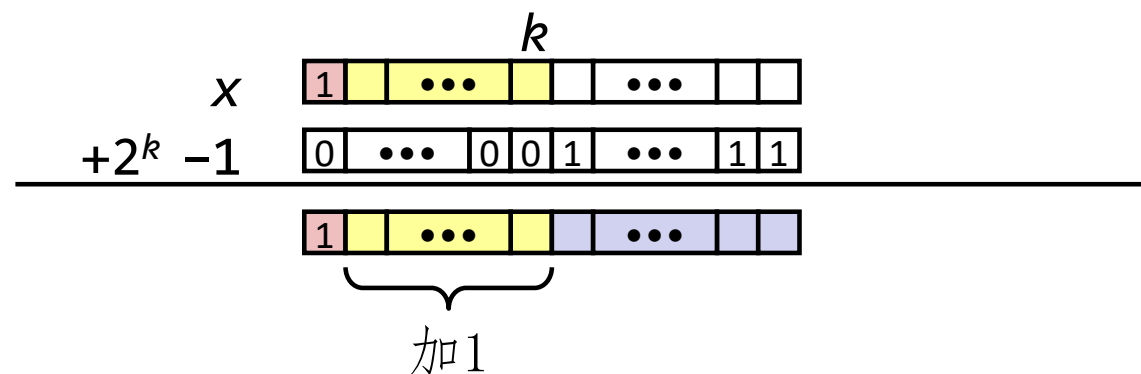


偏置不起效果

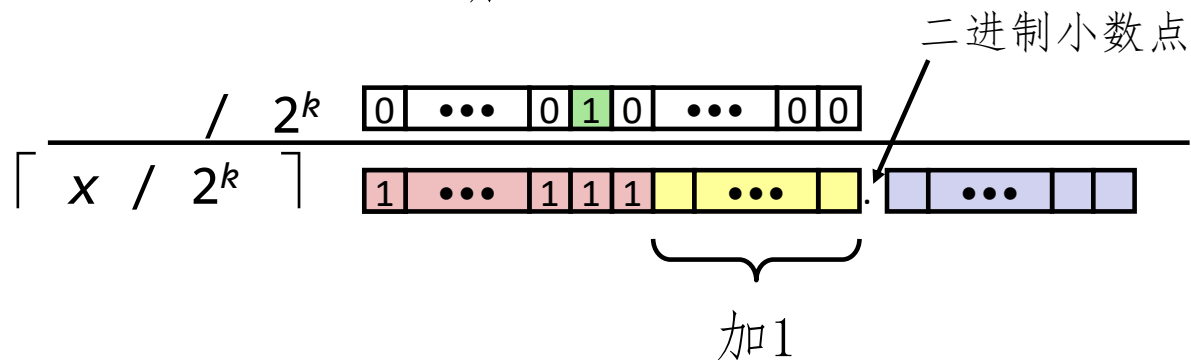
正确的除以2的幂（续）

情况 2：舍入

被除数：



除数：



偏置会在最后的结果上加1

本节内容：位 **bit**、字节、整数和浮点数

- 用 **bit** 来表示信息
- 位级操作
- 整数
 - 表示：无符号 **unsigned** 和有符号 **signed**
 - 转换，强制类型转换
 - 扩展，截断
 - 加，取负，乘法，移位
 - 小结
- 在内存中的表示，指针，字符串
- 浮点数

算数运算：基本规则

■ 加法

- **Unsigned/signed**: 正常做加法，再截断，在位运算层面上操作是一样的
- **Unsigned**: 先加再 $\text{mod } 2^w$
 - 数字相加 + 可能要减去 2^w
- **Signed**: 修改过的加法，再 $\text{mod } 2^w$ (结果会落到正确的范围内)
 - 数字相加 + 可能加也可能减去 2^w

■ 乘法

- **Unsigned/signed**: 正常做乘法，再截断，在位运算层面上操作是一样的
- **Unsigned**: 先乘再 $\text{mod } 2^w$
- **Signed**: 修改过的乘法，再 $\text{mod } 2^w$ (结果会落到正确的范围内)

为什么要使用 **Unsigned**?

- 如果不理解无符号数隐含的含义，就不要使用

- 很容易犯错误

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

- 有些错误可能会很微妙

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

用 **unsigned** 从大到小计数

- 以 **unsigned** 作为循环索引的正确打开方式

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

- 参考 Robert Seacord的 “*Secure Coding in C and C++*”

- C 标准保证 **unsigned** 加法的行为就像是模数运算
 - $0 - 1 \rightarrow UMax$

- 更有甚者

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

- 数据类型 **size_t** 被定义为 **unsigned** 值，长度等于机器字长
- 即使 **cnt** = *Umax* 代码依然正确
- 如果 **cnt** 为 **signed** 并且 < 0 ，会怎样呢？

为什么要使用**Unsigned**? (续)

- 在需要进行模数运算的时候
 - 多精度运算
- 在用位 **bit** 表示集合的时候
 - 逻辑右移, 无符号扩展

整数谜题

初始化

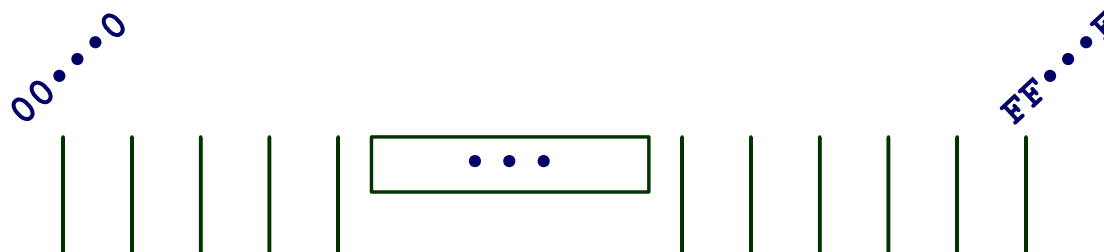
```
int x = foo();  
int y = bar();  
unsigned ux = x;  
unsigned uy = y;
```

1. $x < 0$ $\rightarrow ((x*2) < 0)$
2. $ux \geq 0$
3. $x \& 7 == 7$ $\rightarrow (x \ll 30) < 0$
4. $ux > -1$
5. $x > y$ $\rightarrow -x < -y$
6. $x * x \geq 0$
7. $x > 0 \&\& y > 0$ $\rightarrow x + y > 0$
8. $x \geq 0$ $\rightarrow -x \leq 0$
9. $x \leq 0$ $\rightarrow -x \geq 0$
10. $(x|-x) \gg 31 == -1$
11. $ux \gg 3 == ux/8$
12. $x \gg 3 == x/8$
13. $x \& (x-1) != 0$

本节内容：位 **bit**、字节、整数和浮点数

- 用 **bit** 来表示信息
- 位级操作
- 整数
 - 表示：无符号 **unsigned** 和有符号 **signed**
 - 转换，强制类型转换
 - 扩展，截断
 - 加，取负，乘法，移位
 - 小结
- 在内存中的表示，指针，字符串
- 浮点数

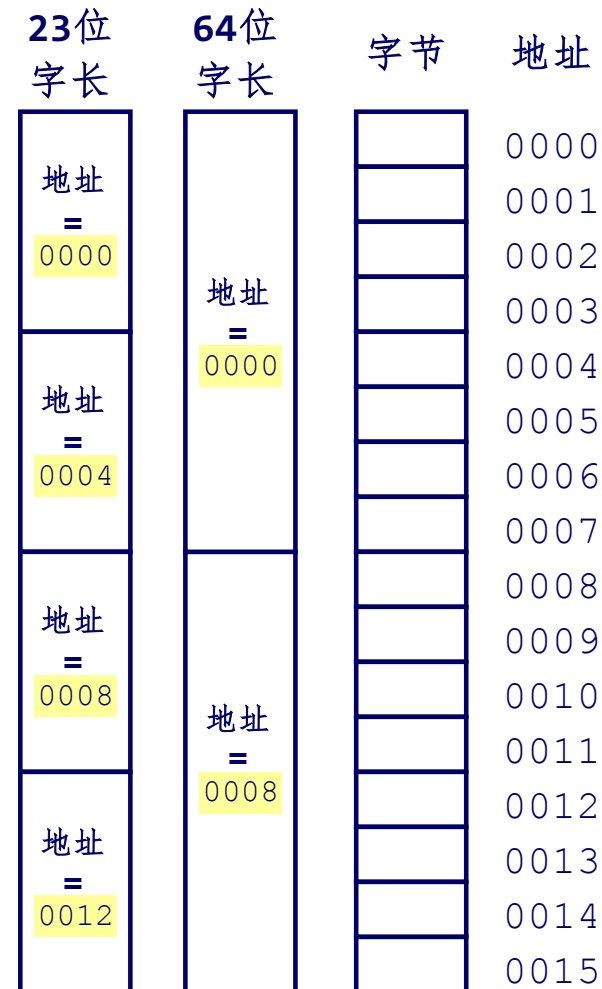
面向字节的内存组织



- 程序利用地址来引用数据
 - 从概念上来说，可以把内存看作一个非常大的字节数组
 - 但是实际上并不是，不过可以这么来看
 - 地址就像是对这个数组的索引
 - 指针变量实际上存储的就是地址
- 注意：系统为每个进程（**process**）提供私有的地址空间
 - 把进程看作一个正在执行的程序
 - 所以，一个程序可以对自己的数据随意处理，但不能碰触其他程序的数据

面向字的内存组织

- 地址是指的字节位置
 - 是字中第一个字节的地址
 - 后续字的地址和这个字的地址相差**4**（**32**位字长）或**8**（**64**位字长）



数据表示示例

C 数据类型	典型的32位系统	典型的64位系统	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

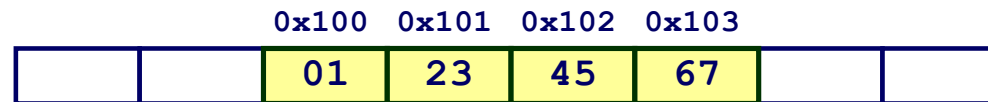
- 那么，在一个多字的字中，字节的顺序是如何的呢？
- 约定
 - 大端法 **Big Endian**: Sun, PPC Mac, Internet
 - 最低位字节在最高地址
 - 小端法 **Little Endian**: x86, 运行Android的ARM处理器, iOS, Windows
 - 最低位字节在最低地址

字节序示例

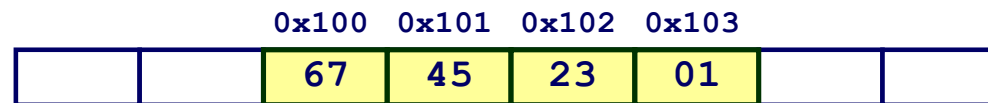
■ 示例

- 变量 `x` 占4个字节，值为 `0x01234567`
- `&x` 得到的地址为 `0x100`

大端法



小端法



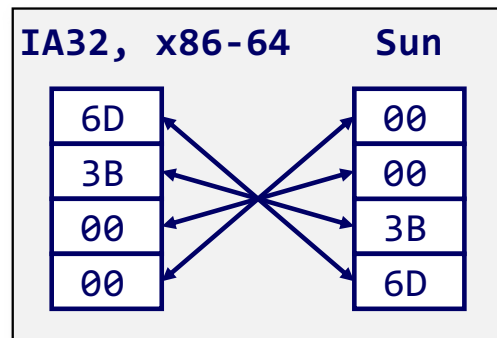
表示整数

十进制： 15213

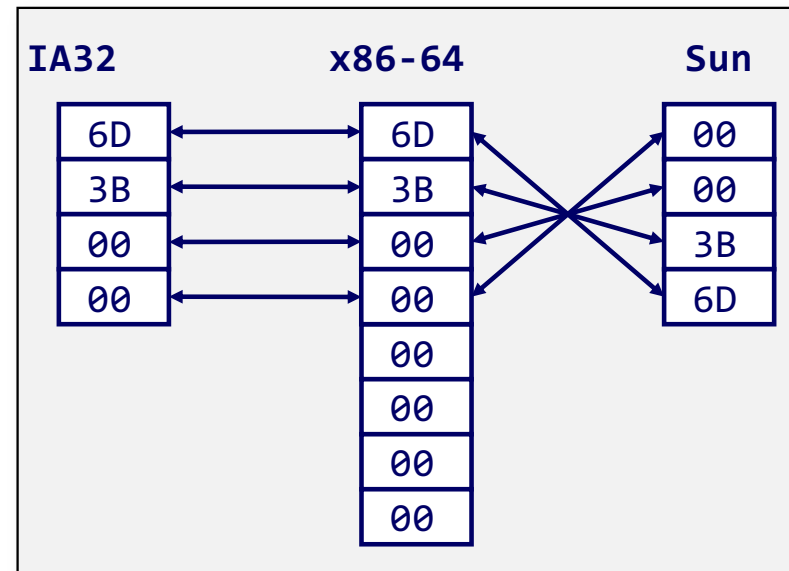
二进制： 0011 1011 0110 1101

十六进制： 3 B 6 D

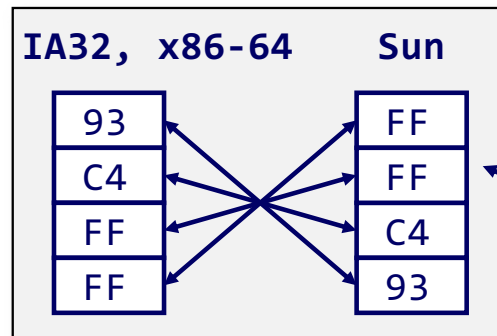
int A = 15213;



long int C = 15213;



int B = -15213;



补码表示

审视数据的表示

■ 打印数据的字节表示的代码

- 将指针强制类型转换为 `unsigned char *` 使得可以把内存当作字节数组来看待和处理

```
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 指示符:

`%p:` 打印指针

`%x:` 打印十六进制

show_bytes 执行示例

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

结果 (Linux x86-64):

```
int a = 15213;  
0x7ffffb7f71dbc      6d  
0x7ffffb7f71dbd      3b  
0x7ffffb7f71dbe      00  
0x7ffffb7f71dbf      00
```

阅读指令代码

■ 反汇编

- 二进制机器码的文本表示
- 由读入机器码的程序生成

■ 片段示例

地址	指令代码	汇编表示
8048365:	5b	pop %ebx
8048366:	81 c3 ab 12 00 00	add \$0x12ab,%ebx
804836c:	83 bb 28 00 00 00 00	cmpl \$0x0,0x28(%ebx)

■ 数字值编码:

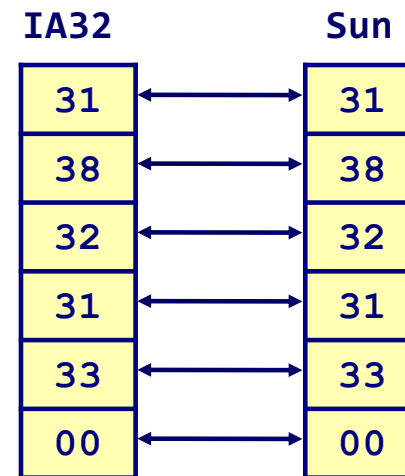
- 填充至32位:
- 按字节划分:
- 反过来:

0x12ab
0x000012ab
00 00 12 ab
ab 12 00 00

表示字符串

- C 语言中的字符串
 - 表示为字符数组
 - 每个字符以 **ASCII** 格式编码
 - 字符集的7位标准编码
 - 字符 “0” 的编码为 $0x30$
 - 数字 i 的编码为 $0x30+i$
 - 字符串应以 **null** 结尾
 - 最后一个字符 = 0
- 兼容性
 - 字节序不是问题

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



本节内容：位 **bit**、字节、整数和浮点数

- 用 **bit** 来表示信息
- 位级操作
- 整数
 - 表示：无符号 **unsigned** 和有符号 **signed**
 - 转换，强制类型转换
 - 扩展，截断
 - 加，取负，乘法，移位
 - 小结
- 在内存中的表示，指针，字符串
- 浮点数

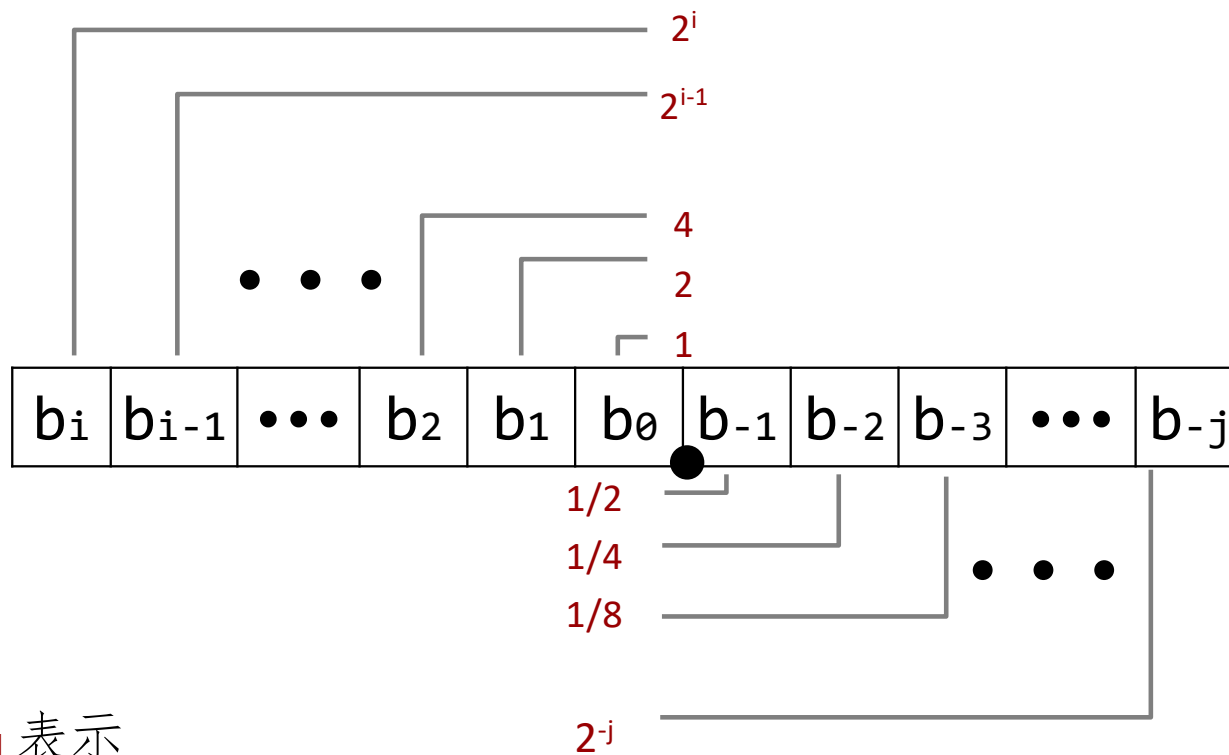
浮点数

- 背景：二进制小数
- IEEE 浮点数标准：定义
- 示例与性质
- 舍入、加法、乘法
- C 语言中的浮点数
- 小结

二进制小数

■ 1011.101_2 是什么？

二进制小数



■ 表示

- 小数点右边的位表示小数的2的幂

- 表示有理数：
$$\sum_{k=-j}^i b_k \times 2^k$$

二进制小数：示例

■ 值	表示
5 3/4	101.11_2
2 7/8	10.111_2
1 7/16	1.0111_2

■ 观察

- 右移可以实现除以2 (unsigned)
- 左移可以实现乘以2
- 形如 $0.111111..._2$ 的数是最接近 1.0 又比 1.0 小的数
 - $1/2 + 1/4 + 1/8 + \dots + 1/2^i + \dots \rightarrow 1.0$
 - 记为 $1.0 - \varepsilon$

可表示的数

■ 局限 #1

- 只能表示型如 $x/2^k$ 的数
 - 其他的有理数有循环的位表示
- 值 表示
 - $1/3$ $0.0101010101[01]..._2$
 - $1/5$ $0.001100110011[0011]..._2$
 - $1/10$ $0.0001100110011[0011]..._2$

■ 局限 #2

- 对于 w 位数，只有一个小数点位置的设定
 - 表数范围有限（对于特别小或特别大的值该怎么办呢？）

浮点数

- 背景：二进制小数
- IEEE 浮点数标准：定义
- 示例与性质
- 舍入、加法、乘法
- C 语言中的浮点数
- 小结

IEEE 浮点数

■ IEEE 标准754

- 1985年提出，作为浮点数运算对统一标准
 - 在那之前，有多种各异的格式
- 现在已为所有主流 CPU 支持

■ 设计的出发点是数值计算

- 该标准在舍入、上溢和下溢方面设计都很好
- 在硬件设计上很难实现得很快
 - 在定义标准时，数值分析师的意见压倒了硬件设计者的意见

浮点数表示

■ 数值形式:

$$(-1)^s M 2^E$$

- 符号位 s 决定这个数是负还是正
- 尾数 M 通常是一个范围在 $[1.0, 2.0)$ 内的小数
- 指数 E 是2的幂, 表明权重

■ 编码

- 最高位MSB s 就是符号位 s
- exp 字段是 E 的编码 (但不等于 E)
- frac 字段是 M 的编码 (但不等于 M)



精度选择

■ 单精度：32位



■ 双精度：64位



■ 扩展精度：80位 (Intel only)



规格化的值

$$v = (-1)^s M 2^E$$

- 当 $\text{exp} \neq 000\dots 0$ 且 $\text{exp} \neq 111\dots 1$ 时
- 指数编码为一个偏置的值: $E = \text{Exp} - \text{Bias}$
 - Exp : exp 字段对应的 unsigned 值
 - $\text{Bias} = 2^{k-1} - 1$, 此处 k 是指数的位数数值
 - 单精度: 127 (Exp : 1...254, E : -126...127)
 - 双精度: 1023 (Exp : 1...2046, E : -1022...1023)
- 尾数编码, 有一个隐含的最高位1: $M = 1.\text{xxx}\dots\text{x}_2$
 - $\text{xxx}\dots\text{x}$: frac 字段对应的位
 - $\text{frac}=000\dots 0$ 为最小值 ($M = 1.0$)
 - $\text{frac}=111\dots 1$ 为最大值 ($M = 2.0 - \epsilon$)
 - 最高位是“免费”多得来的

规格化的编码示例

$$v = (-1)^s M 2^E$$
$$E = \text{Exp} - \text{Bias}$$

■ 值: float F = 15213.0;

$$\begin{aligned} \blacksquare 15213_{10} &= 11101101101101_2 \\ &= 1.1101101101101_2 \times 2^{13} \end{aligned}$$

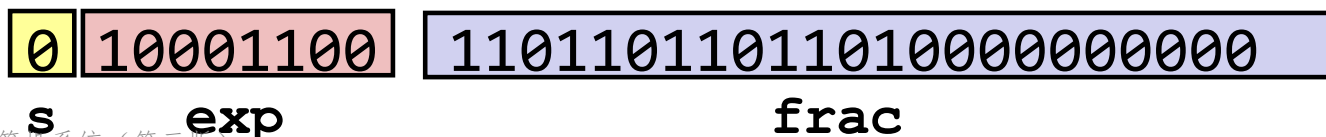
■ 尾数

$$\begin{aligned} M &= 1.\underline{1101101101101}_2 \\ \text{frac} &= \underline{1101101101101}0000000000_2 \end{aligned}$$

■ Exponent

$$\begin{aligned} E &= 13 \\ \text{Bias} &= 127 \\ \text{Exp} &= 140 = 10001100_2 \end{aligned}$$

■ Result:



非规格化的值

$$\begin{aligned} v &= (-1)^s M 2^E \\ E &= 1 - \text{Bias} \end{aligned}$$

- 条件: $\text{exp} = 000\dots 0$
- 指数值: $E = 1 - \text{Bias}$ (而不是 $E = 0 - \text{Bias}$)
- 尾数编码, 有一个隐含的最高位0: $M = 0.\text{xxx}\dots\text{x}_2$
 - $\text{xxx}\dots\text{x}$: frac 的位
- 情况
 - $\text{exp} = 000\dots 0, \text{frac} = 000\dots 0$
 - 表示值0
 - 注意不同的值: $+0$ and -0 (为什么会这样?)
 - $\text{exp} = 000\dots 0, \text{frac} \neq 000\dots 0$
 - 接近于 0.0 的值
 - 等间距的

特殊值

■ 条件: **exp** = 111...1

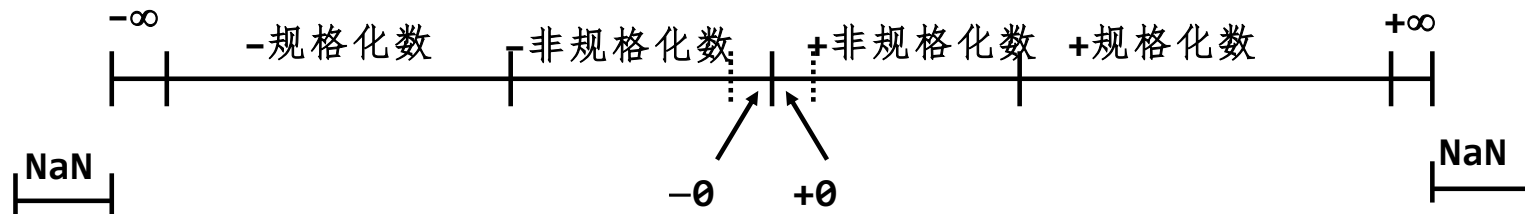
■ 情况: **exp** = 111...1, **frac** = 000...0

- 表示值 ∞ (无穷大)
- 运算溢出
- 有正值也有负值
- E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

■ 情况: **exp** = 111...1, **frac** \neq 000...0

- 非数 Not-a-Number (NaN)
- 表示不能确定数值的情况
- E.g., $\text{sqrt}(-1)$, $\infty - \infty$, $\infty \times 0$

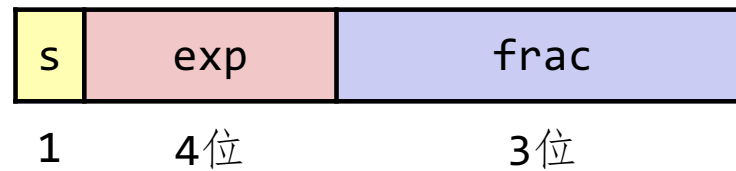
浮点数编码的可视化表示



浮点数

- 背景：二进制小数
- IEEE 浮点数标准：定义
- 示例与属性
- 舍入、加法、乘法
- C 语言中的浮点数
- 小结

一个小小浮点数示例



■ 8位浮点数表示

- 最高位为符号位
- 接下来4位是阶码 **exp**，偏置为7
- 最后3位是 **frac**

■ 与 IEEE 格式一致

- 规格化、非规格化
- 0、NaN和无穷大的表示

表数范围（仅正数）

$$v = (-1)^s M 2^E$$

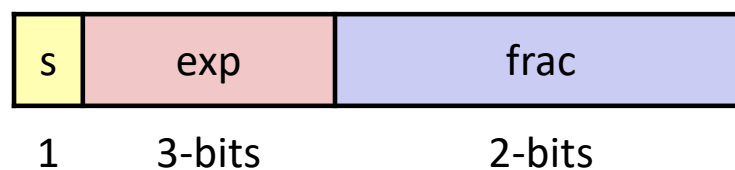
n: $E = \text{Exp} - \text{Bias}$
d: $E = 1 - \text{Bias}$

	s	exp	frac	E	值	
非规格化数	0	0000	000	-6	0	
	0	0000	001	-6	$1/8 * 1/64 = 1/512$	接近0
	0	0000	010	-6	$2/8 * 1/64 = 2/512$	
	...					
	0	0000	110	-6	$6/8 * 1/64 = 6/512$	
	0	0000	111	-6	$7/8 * 1/64 = 7/512$	最大的非规格化数
规格化数	0	0001	000	-6	$8/8 * 1/64 = 8/512$	最小的规格化数
	0	0001	001	-6	$9/8 * 1/64 = 9/512$	
	...					
	0	0110	110	-1	$14/8 * 1/2 = 14/16$	
	0	0110	111	-1	$15/8 * 1/2 = 15/16$	小于1但接近1
	0	0111	000	0	$8/8 * 1 = 1$	
	0	0111	001	0	$9/8 * 1 = 9/8$	大于1但接近1
	0	0111	010	0	$10/8 * 1 = 10/8$	
	...					
	0	1110	110	7	$14/8 * 128 = 224$	
	0	1110	111	7	$15/8 * 128 = 240$	最大的规格化数
	0	1111	000	n/a	inf	

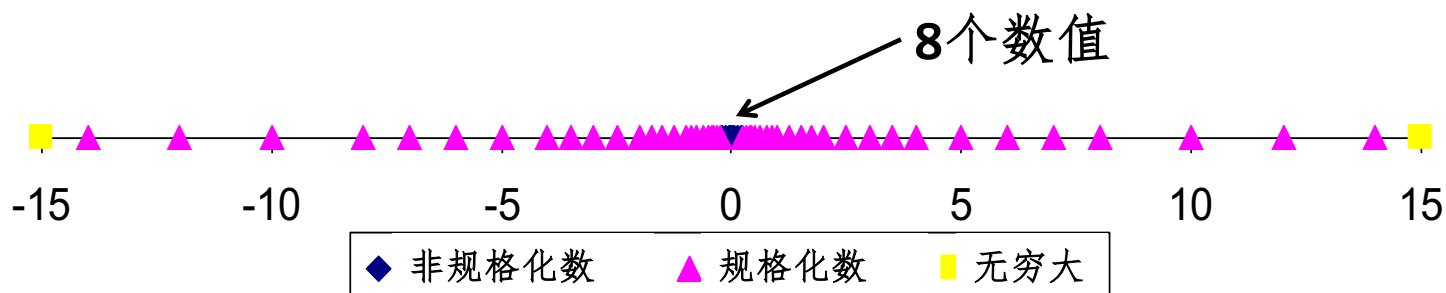
数值的分布

■ 6位类 IEEE 格式

- $e = 3$ exp位
- $f = 2$ frac位
- 偏置为 $2^{3-1}-1 = 3$



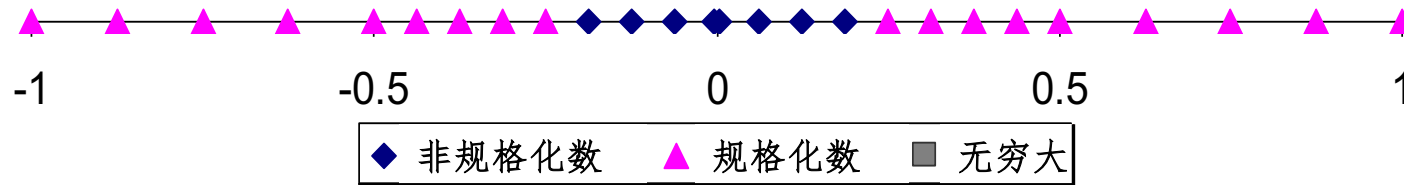
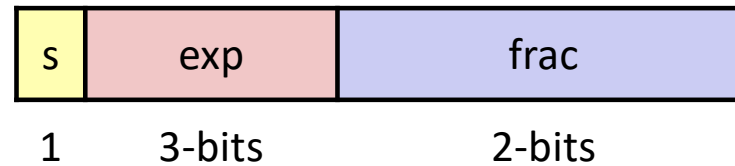
■ 注意数值的分布越接近0越密集



数值的分布(拉近一点儿看)

■ 6位类 IEEE 格式

- $e = 3$ exp位
- $f = 2$ frac位
- 偏置为 3



IEEE 编码的特殊属性

- 浮点数的0与整数0的表示相同
 - 所有的位都 = 0
- (几乎) 可以使用无符号整数的比较
 - 先比较符号位
 - 要考虑 $-0 = 0$
 - NaN 有点儿麻烦
 - 大于任何其他数值
 - 该如何产生比较结果呢?
 - 其他的都没问题
 - 非规格化数 vs. 规格化数
 - 规格化数 vs. 无穷大

浮点数

- 背景：二进制小数
- IEEE 浮点数标准：定义
- 示例与性质
- 舍入、加法、乘法
- C 语言中的浮点数
- 小结

浮点数运算：基本思想

$$\blacksquare \mathbf{x} +_f \mathbf{y} = \text{Round}(\mathbf{x} + \mathbf{y})$$

$$\blacksquare \mathbf{x} \times_f \mathbf{y} = \text{Round}(\mathbf{x} \times \mathbf{y})$$

■ 基本思想

- 首先计算准确的结果
- 使之适配到期望的精度
 - 如果指数太大可能会溢出
 - 可能需要舍入以适配 `frac`

舍入

■ 舍入模式

	1.40	1.60	1.50	2.50	-1.50
向0舍入	1	1	1	2	-1
向下舍入 ($-\infty$)	1	1	1	2	-2
向上舍入 ($+\infty$)	2	2	2	3	-1
向偶数舍入 (缺省的)	1	2	2	2	-2

细看向偶数舍入

■ 缺省的舍入模式

- 不使用汇编很难换而使用其他模式
- 所有其他的模式都是统计学上有偏差的
 - 所有正数全部都被高估或者低估

■ 十进制数的向偶数舍入

- 当需要舍入的值刚好在两个合法值中间时
 - 向最低位为偶数的方向舍入
- E.g., 舍入到最近的1/100

7.8949999 7.89 (不到一半)

7.8950001 7.90 (大于一半)

7.8950000 7.90 (刚好在中间, 向上舍入)

7.8850000 7.88 (刚好在中间, 向下舍入)

二进制数的舍入

■ 二进制小数

- “偶数”是指最低位为0
- “刚好在中间”是指待舍入位置点的右边的位 = 100...₂

■ 示例

- 舍入到最近的 1/4 (小数点后2位)

值	二进制	舍入后	动作	舍入后的值
2 3/32	10.00011 ₂	10.00 ₂	(<1/2-向下)	2
2 3/16	10.00110 ₂	10.01 ₂	(>1/2-向上)	2 1/4
2 7/8	10.11100 ₂	11.00 ₂	(1/2-向上)	3
2 5/8	10.10100 ₂	10.10 ₂	(1/2-向下)	2 1/2

浮点数乘法

- $(-1)^{s1} M1 2^{E1} \times (-1)^{s2} M2 2^{E2}$
- 准确的结果: $(-1)^s M 2^E$
 - 符号位 s : $s1 \wedge s2$
 - 尾数 M : $M1 \times M2$
 - 指数 E : $E1 + E2$
- 修正
 - 如果 $M \geq 2$, M 右移, E 增加
 - 如果 E 超出范围, 溢出
 - 对 M 进行舍入使之符合 frac 的精度
- 实现
 - 最大的工作量是尾数相乘

浮点数加法

■ $(-1)^{s1} M1 2^{E1} + (-1)^{s2} M2 2^{E2}$

■ 假设 $E1 > E2$

■ 准确的结果: $(-1)^s M 2^E$

■ 符号 s , 尾数 M :

■ 有符号数对齐 & 相加的结果

■ 指数 E : $E1$

■ 修正

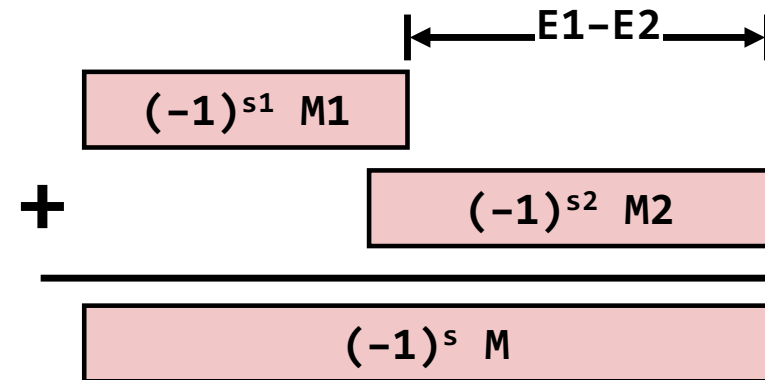
■ 如果 $M \geq 2$, M 右移, E 增加

■ 如果 $M < 1$, M 左移 k 位, E 减去 k

■ 如果 E 超出范围, 溢出

■ 对 M 进行舍入使之符合 frac 的精度

二进制小数点要对齐



浮点数加法的数学性质

■ 可交换吗？

- Yes

■ 可结合吗？

- No

- 可能溢出，不准确的舍入

- $(3.14+1e10)-1e10 = 0$, $3.14+(1e10-1e10) = 3.14$

■ 保证单调性吗？

- $a \geq b \Rightarrow a+c \geq b+c$?

- 除了对于无穷大和 NaN 以外，都成立

浮点数乘法的数学性质

■ 可交换吗？

- Yes

■ 可结合吗？

- No

- 可能溢出，不准确的舍入

- 例： $(1e20 * 1e20) * 1e-20 = \text{inf}$, $1e20 * (1e20 * 1e-20) = 1e20$

■ 乘法在加法上可分配吗？

- No

- 可能溢出，不准确的舍入

- $1e20 * (1e20 - 1e20) = 0.0$, $1e20 * 1e20 - 1e20 * 1e20 = \text{NaN}$

■ 保证单调性吗？

- $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c$?

- 除了对于无穷大和 NaN 以外，都成立

浮点数

- 背景：二进制小数
- IEEE 浮点数标准：定义
- 示例与性质
- 舍入、加法、乘法
- C 语言中的浮点数
- 小结

C 语言中的浮点数

■ C 语言中有两种浮点数

- **float** 单精度
- **double** 双精度

■ 转换/强制类型转换

- **int, float 和 double** 之间强制类型转换会改变位表示
- **double/float → int**
 - 截断小数部分
 - 类似于向0舍入
 - 超出表数范围或 NaN 时，行为未定义：通常会设置为 TMin
- **int → double**
 - 准确的转换，只要 $\text{int} \leq 53 \text{ bit}$
- **int → float**
 - 可能会进行舍入

浮点数谜题

- 对下列 C 语言表达式，判断是下列两种情况中的哪一种：
 - 它对所有的参数值都正确
 - 解释什么情况不正确

```
int x = ...;  
float f = ...;  
double d = ...;
```

假设 **d** 和 **f** 都不是NaN

1. `x == (int)(float) x`
2. `x == (int)(double) x`
3. `f == (float)(double) f`
4. `d == (double)(float) d`
5. `f == -(-f);`
6. `2/3 == 2/3.0`
7. `d < 0.0` \Rightarrow `((d*2) < 0.0)`
8. `d > f` \Rightarrow `-f > -d`
9. `d * d >= 0.0`
10. `(d+f)-d == f`

浮点数除0的问题

这是网上的一个帖子

```
#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;
```

```
}
```

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???

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。
- 原因是将一个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的一个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是0.1的近似表示， $0.1-x$

$$= 0.000\ 1100\ 1100\ 1100\ 1100\ 1100\ [1100]... - 0.000\ 1100\ 1100\ 1100\ 1100\ 1100B$$

即为：

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

$$= 2^{-20} \times 0.1 \approx 9.54 \times 10^{-8} \quad \text{这就是机器值与真值之间的误差!}$$

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，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多少？

100小时相当于计数了 $100 \times 60 \times 60 \times 10 = 36 \times 10^5$ 次，因而导弹的时钟已经偏差了 $9.54 \times 10^{-8} \times 36 \times 10^5 \approx 0.343$ 秒

因此，距离误差是 $2000 \times 0.343 \text{秒} \approx 687$ 米

举例：爱国者导弹定位错误

- 若 x 用 **float** 型表示，则 x 的机器数是什么？ 0.1 与 x 的偏差是多少？系统运行 **100** 小时后的时钟偏差是多少？在飞毛腿速度为 **2000** 米/秒的情况下，预测的距离偏差为多少？
- $0.1 = 0.0\ 0011[0011]_B = +1.1\ 0011\ 0011\ 0011\ 0011\ 0011\ 00B \times 2^{-4}$ ，故 x 的机器数为 $0\ 011\ 1101\ 1\ 100\ 1100\ 1100\ 1100\ 1100\ 1100$
- **Float** 型仅 **24** 位有效位数，后面的有效位全被截断，故 x 与 0.1 之间的误差为：
 $|x - 0.1| = 0.000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1100\ [1100]..._B$ 。这个值等于 $2^{-24} \times 0.1 \approx 5.96 \times 10^{-9}$ 。100 小时后时钟偏差 $5.96 \times 10^{-9} \times 36 \times 10^5 \approx 0.0215$ 秒。距离偏差 $0.0215 \times 2000 \approx 43$ 米。比爱国者导弹系统精确约 **16** 倍。

举例：爱国者导弹定位错误

- 若用32位二进制定点小数 $x=0.000\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1101\ 1101\ B$ 表示0.1，则误差比用float表示误差更大还是更小？
- 当 $x=0.000\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1101\ B$ 时，与0.1之间的误差约为： $|x-0.1|=0.000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 00\ 1100\ [1100]...B$ 。这个值等于 $2^{-30} \times 0.1 \approx 9.31 \times 10^{-11}$ 。100小时后时钟偏差 $9.31 \times 10^{-11} \times 36 \times 10^5 \approx 0.000335$ 秒。预测的距离偏差仅为 $0.000335 \times 2000 \approx 0.67$ 米。

举例：浮点数运算的精度问题

■ 从上述结果可以看出：

- 用32位定点小数表示0.1，比采用float精度高64倍
- 用float表示在计算速度上更慢，必须先把计数值转换为IEEE 754格式浮点数，然后再对两个IEEE 754格式的数相乘，故采用float比直接将两个二进制数相乘要慢

■ Ariana 5火箭和爱国者导弹的例子带来的启示

- ✓ 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

浮点数小结

- IEEE 浮点数有明确的数学性质
- 表示形如 $M \times 2^E$ 的数字
- 可以依据规则推测运算的结果，与实现无关
 - 好像是按照完美精度进行计算，再进行舍入
- 与真实的算术计算是不同的
 - 没有结合律和分配律
 - 对编译器和严肃的数值应用程序猿来说，增加了很多困难