

位、整数类型与浮点数

Bits, Integers & Floating Points

课 程 名：计算机系统

第 1 讲（2025年4月23日）

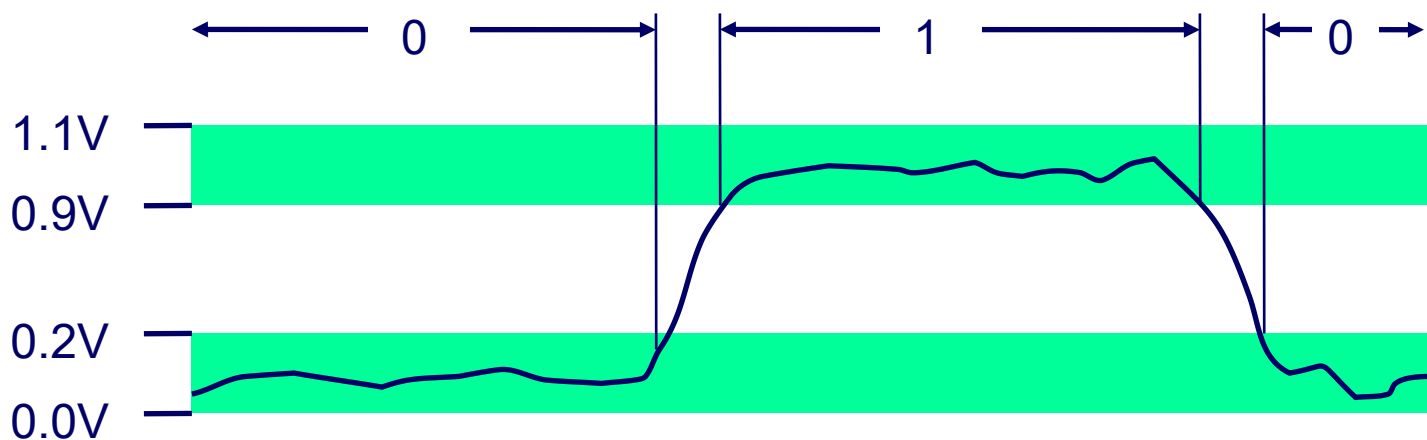
主 讲：华栋

本课内容

- 位操作
- 整型数
 - 表示方法: signed 与 unsigned
 - 类型转换casting, 扩展expanding 与截断truncating
 - 加法、乘法、移位
- 内存中的数据表示, 指针, 字符串
- 浮点数
 - 背景知识: 二进制小数
 - IEEE 浮点数标准: 定义、举例与性质
 - 舍入, 加法, 乘法
 - C 语言的浮点类型

一切都是比特bit

- Bit 就是 0 或 1
- 将一组 bit 按不同的方法进行编码或解析
 - 然后由计算机决定该对其做什么（指令）
 - 实现数值、集合、字符串、... 的表示和运算
- 为何选用 bit？方便用电子技术加以实现
 - 容易用双稳态元件存储
 - 可在充斥噪声的不精确线路上实现稳定传输



举例：利用二进制进行计数

- 二进制数值表示

- 10213_{10} 表示为 $10\ 0111\ 1110\ 0101_2$
- 1.20_{10} 表示为 $1.0011001100110011[0011]..._2$
- 1.0213×10^4 表示为 $1.0011\ 1111\ 00101_2 \times 2^{13}$

对字节的编码

- 1 字节Byte = 8 bits
 - 二进制binary: 00000000_2 to 11111111_2
 - 十进制decimal: 0_{10} to 255_{10}
 - 十六进制hexadecimal: 00_{16} to FF_{16}
 - 以 16 为基数的数制
 - 使用字符 '0' 到 '9' 以及 'A' 到 'F'
 - 数值 $FA1D37B_{16}$ 用 C 语言表示为:
 - `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

数据表示举例

C 语言数据类型	典型 32 位机	典型 64 位机	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
指针类型	4	8	8

- 使用 GCC 的 `-m32` / `-m64` 选项进行编译，查看 `long` 的位宽
 - `gcc -m64 checkLong.c`

本课内容

- 位操作
- 整型数
 - 表示方法: signed 与 unsigned
 - 类型转换casting, 扩展expanding 与截断truncating
 - 加法、乘法、移位
- 内存中的数据表示, 指针, 字符串
- 浮点数
 - 背景知识: 二进制小数
 - IEEE 浮点数标准: 定义、举例与性质
 - 舍入, 加法, 乘法
 - C 语言的浮点类型

布尔代数

- 由 George Boole 于 19 世纪创立
 - 逻辑的代数表示
 - 1 代表 “True”，0 代表 “False”

AND

- $A \& B = 1$ when both $A=1$ and $B=1$

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

NOT

- $\sim A = 1$ when $A=0$

\sim	
0	1
1	0

OR

- $A \mid B = 1$ when either $A=1$ or $B=1$

\mid	0	1
0	0	1
1	1	1

Exclusive-OR (XOR)

- $A \wedge B = 1$ when either $A=1$ or $B=1$, but not both

\wedge	0	1
0	0	1
1	1	0

布尔代数的推广

- 对位向量进行操作
 - 按位进行操作

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

- 布尔代数的所有性质均适用

举例：集合的表示与运算

■ 表示

- 用 w 位宽的向量表示 $\{0, \dots, w-1\}$ 的子集

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

- 01101001 $\{0, 3, 5, 6\}$

- 76543210

- 01010101 $\{0, 2, 4, 6\}$

- 76543210

■ 运算

- | | | |
|--------------------------------|----------|------------------------|
| ■ & 交intersection | 01000001 | $\{0, 6\}$ |
| ■ 并union | 01111101 | $\{0, 2, 3, 4, 5, 6\}$ |
| ■ ^ 对称差symmetric difference | 00111100 | $\{2, 3, 4, 5\}$ |
| ■ ~ 补complement | 10101010 | $\{1, 3, 5, 7\}$ |

C 语言的按位运算

- C 提供 `&`、`|`、`~`、`^` 等位运算符
 - 可用于对任意“整数”类型数据的操作
 - `long`, `int`, `short`, `char`, `unsigned`
 - 将操作数视作位向量
 - 逐位进行运算
- 举例（下列数据均为 `char` 型）
 - $\sim 0x41 \rightarrow 0xBE$
 - $\sim 01000001_2 \rightarrow 10111110_2$
 - $\sim 0x00 \rightarrow 0xFF$
 - $\sim 00000000_2 \rightarrow 11111111_2$
 - $0x69 \ \& \ 0x55 \rightarrow 0x41$
 - $01101001_2 \ \& \ 01010101_2 \rightarrow 01000001_2$
 - $0x69 \ | \ 0x55 \rightarrow 0x7D$
 - $01101001_2 \ | \ 01010101_2 \rightarrow 01111101_2$

对比：C 语言的逻辑运算

■ 逻辑操作符

- `&&`, `||`, `!`
 - 将数值 0 视为 “False”
 - 一切非零值视为 “True”
 - 结果只能是 0 或 1
 - 提前终止 **early termination**（短路式操作）

■ 举例（下列数据均为 char 型）

- `!0x41` \rightarrow `0x00`
- `!0x00` \rightarrow `0x01`
- `!!0x41` \rightarrow `0x01`
- `0x69 && 0x55` \rightarrow `0x01`
- `0x69 || 0x55` \rightarrow `0x01`
- `p && *p++`（避免对空指针解引用）

移位运算

- 左移: $x \ll y$
 - 将位向量 x 左移 y 位, 将左侧多出的位丢弃
 - 右侧空位补 0
- 右移: $x \gg y$
 - 将位向量 x 右移 y 位
 - 将右侧多出的位丢弃
 - 逻辑右移 (unsigned **一定**采用)
 - 左侧空位补 0
 - 算术右移 (signed **大多**采用)
 - 左侧空位补原最高位 (符号位)
- 未定义行为
 - 移位位数 < 0 或 \geq 字长

x	01100010
$\ll 3$	00010000
逻辑 $\gg 2$	00011000
算术 $\gg 2$	00011000

x	10100010
$\ll 3$	00010000
逻辑 $\gg 2$	00101000
算术 $\gg 2$	11101000

本课内容

- 位操作
- 整型数
 - 表示方法: `signed` 与 `unsigned`
 - 类型转换 `casting`, 扩展 `expanding` 与 截断 `truncating`
 - 加法、乘法、移位
- 内存中的数据表示, 指针, 字符串
- 浮点数
 - 背景知识: 二进制小数
 - IEEE 浮点数标准: 定义、举例与性质
 - 舍入, 加法, 乘法
 - C 语言的浮点类型

整型数的编码

无符号数 unsigned

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

补码 Two's Complement

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

```
short int x = 10213;
short int y = -10213;
```

符号位

■ C 的 short 类型占 2 字节

	十进制	十六进制	二进制
x	10213	27 E5	00100111 11100101
y	-10213	D8 1B	11011000 00011011

■ 符号位

- 对于补码数而言，最高位MSB（most significant bit）表示正负
 - 0 为非负
 - 1 为负

补码表示举例（续）

$x =$ 10213: 00100111 11100101
 $y =$ -10213: 11011000 00011011

Weight	10213		-10213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	0	0	1	8
16	0	0	1	16
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	0	0	1	2048
4096	0	0	1	4096
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum	10213		-10213	

表数范围

■ 无符号数

$$\begin{array}{lcl} \blacksquare & UMin & = 0 \\ & 000\dots0 & \end{array}$$

$$\begin{array}{lcl} \blacksquare & UMax & = 2^w - 1 \\ & 111\dots1 & \end{array}$$

■ 补码数

$$\begin{array}{lcl} \blacksquare & TMin & = -2^{w-1} \\ & 100\dots0 & \end{array}$$

$$\begin{array}{lcl} \blacksquare & TMax & = 2^{w-1} - 1 \\ & 011\dots1 & \end{array}$$

■ 其它（特殊值）

$$\begin{array}{lcl} \blacksquare & \text{负 } 1 & \\ & 111\dots1 & \end{array}$$

$W = 16$ 时对应的值

	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

表数范围

	位宽 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

■ 结论

- $|TMin| = TMax + 1$
 - 范围正负不对称
- $UMax = 2 * TMax + 1$

■ C 语言

- `#include <limits.h>`
- 定义边界点的宏，如：
 - `ULONG_MAX`
 - `LONG_MAX`
 - `LONG_MIN`
- 具体值因不同平台而异

无符号数与带符号数

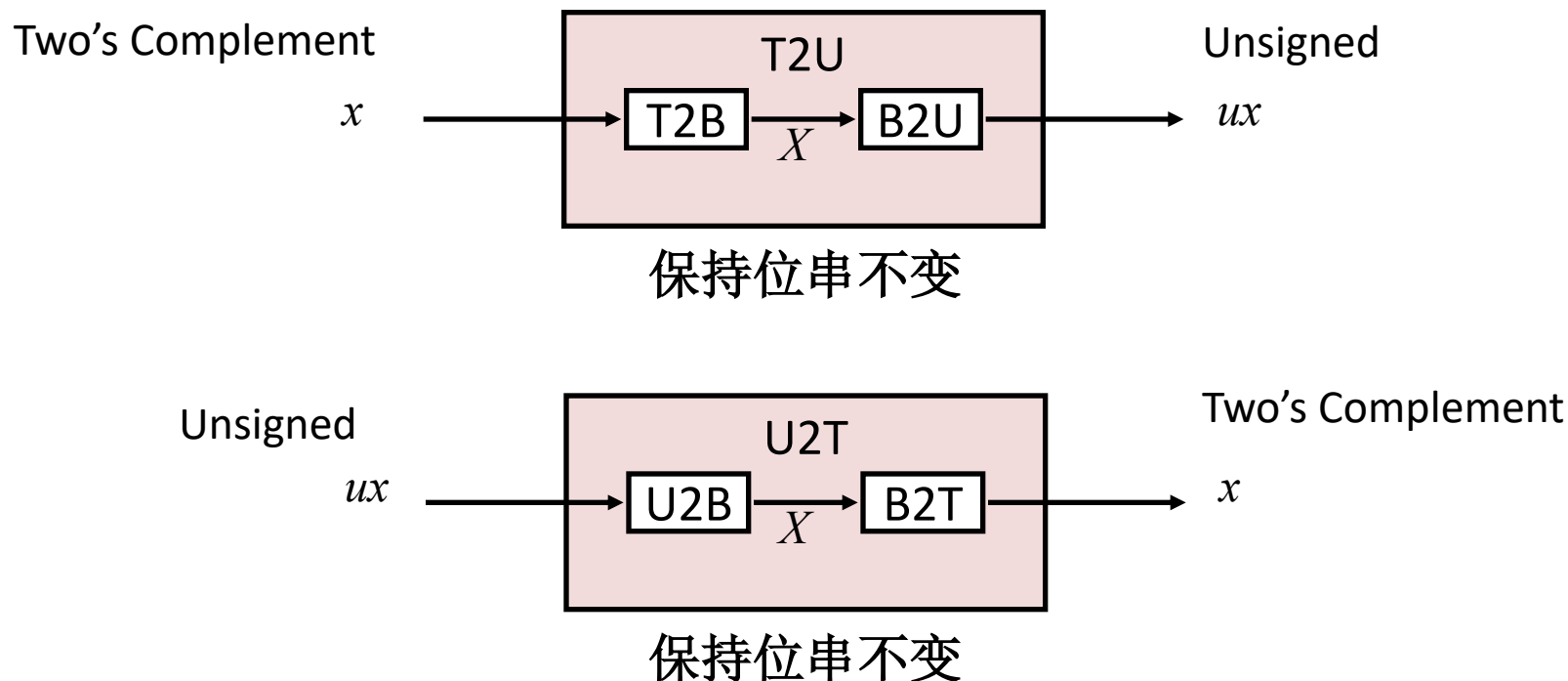
X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

- 相同点
 - 对于非负区间的值，两种编码相同
- 唯一性（位串与真值一一对应）
 - 每个位串表示唯一的整数值
 - 此范围内的每个整数都有唯一的位串表示
- \Rightarrow Can Invert Mappings
 - $U2B(x) = B2U^{-1}(x)$
 - Bit pattern for unsigned integer
 - $T2B(x) = B2T^{-1}(x)$
 - Bit pattern for two's comp integer

本课内容

- 位操作
- **整型数**
 - 表示方法: signed 与 unsigned
 - **类型转换casting, 扩展expanding 与截断truncating**
 - 加法、乘法、移位
- 内存中的数据表示, 指针, 字符串
- 浮点数
 - 背景知识: 二进制小数
 - IEEE 浮点数标准: 定义、举例与性质
 - 舍入, 加法, 乘法
 - C 语言的浮点类型

带符号数与无符号数之间的转换



- 无符号数unsigned和带符号数signed之间的转换
保持位串不变，重新解析得到真值

signed ↔ unsigned

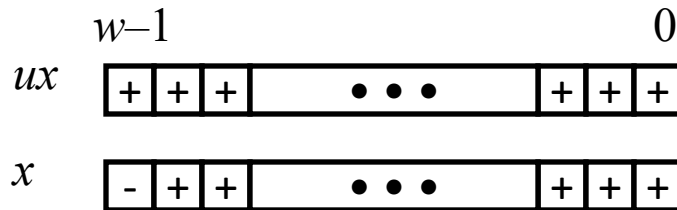
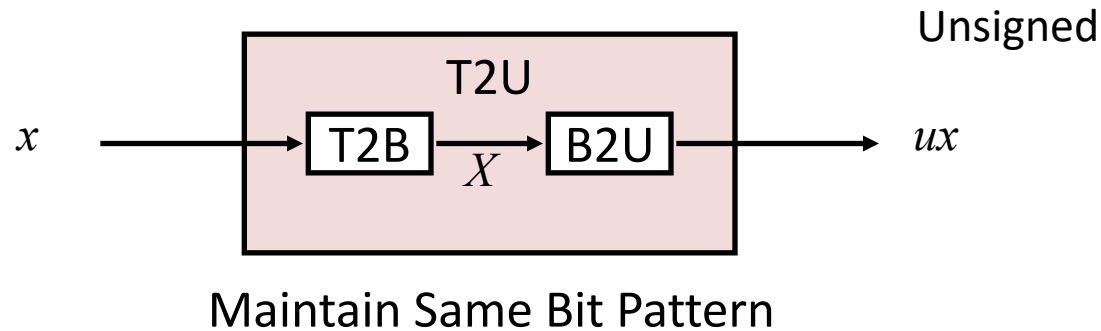
位串	signed		unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8	→ T2U →	8
1001	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1	← U2T ←	15

signed ↔ unsigned

位串	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

signed 和 unsigned 二者间的关系

Two's Complement

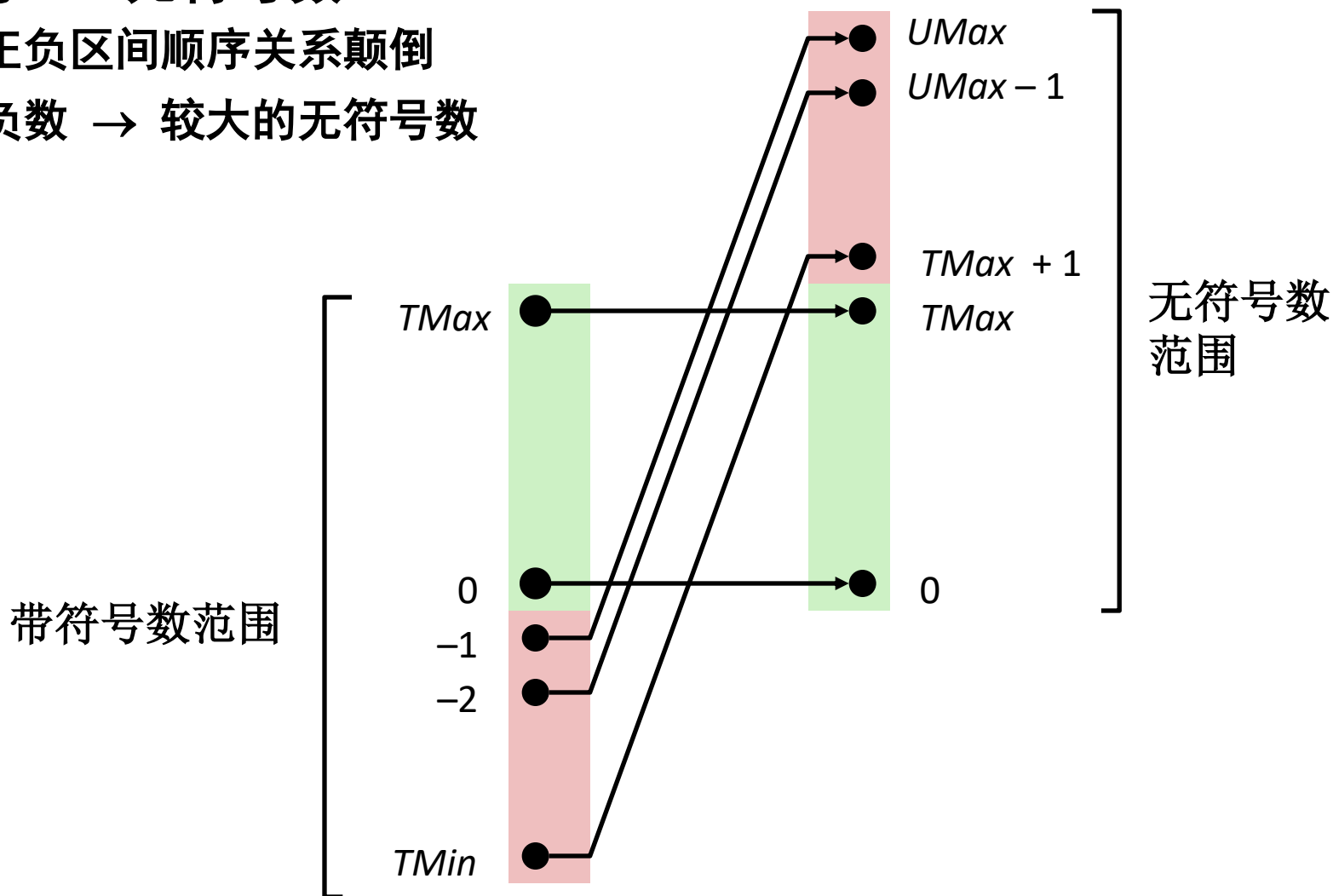


Large negative weight
becomes
 Large positive weight

转换关系可视化

■ 补码 → 无符号数

- 正负区间顺序关系颠倒
- 负数 → 较大的无符号数



C 语言的 signed 与 unsigned 类型

■ 常量

- 默认为带符号数（补码数）
- 若带有 **U** 后缀，则按无符号数处理

0U, 4294967259U

■ 类型转换casting

- 显式类型转换，方法同 U2T 和 T2U

```
int tx, ty;
```

```
unsigned ux, uy;
```

```
tx = (int) ux;
```

```
uy = (unsigned) ty;
```

- 隐式类型转换，在赋值和过程调用时发生

```
tx = ux;
```

```
uy = ty;
```

类型转换之坑

■ 表达式求值

- 如果表达式中既有 signed 又有 unsigned, 则 C 会默认将 signed 转换成 unsigned
- 包括关系运算: <, >, ==, <=, >=
- 以 $W = 32$ 为例: $TMIN = -2,147,483,648$ $TMAX = 2,147,483,647$

■	常量 ₁	常量 ₂	大小关系	类型
	0	0U	相等	unsigned
	-1	0	小于	signed
	-1	0U	大于	unsigned
	2147483647	-2147483647-1	大于	signed
	2147483647U	-2147483647-1	小于	unsigned
	-1	-2	大于	signed
	(unsigned) -1	-2	大于	unsigned
	2147483647	2147483648U	小于	unsigned
	2147483647	(int) 2147483648U	大于	signed

小结:

`signed` \leftrightarrow `unsigned` 转换的基本规则

- 保持位串bit pattern不变
- 但要对其重新解析
- 会产生意外的效果：加上或减去 2^w
- 表达式中同时含有 `int` 和 `unsigned` 时
 - 默认将 `int` 转换成 `unsigned`!

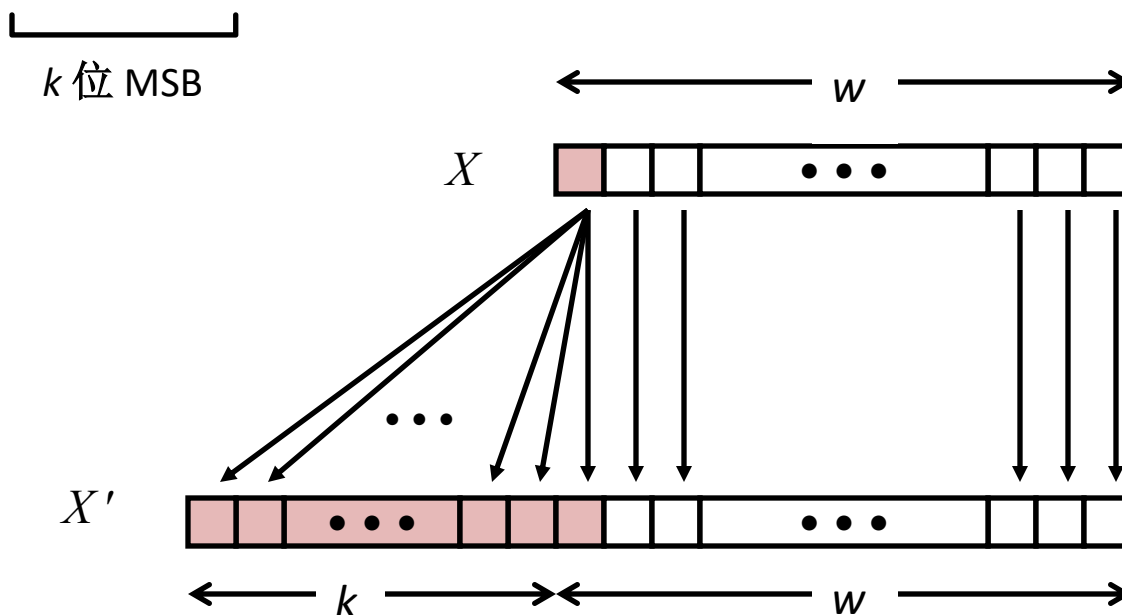
符号扩展运算

■ 任务：

- 给定某 w 位的带符号整数 x
- 将其转换成 $w+k$ 位的整数，要求真值不变

■ 运算规则：

- 将符号位复制 k 份：
- $X' = \underbrace{x_{w-1}, \dots, x_{w-1}}_{k \text{ 位 MSB}}, x_{w-1}, x_{w-2}, \dots, x_0$



符号扩展举例

```
short int x = 10213;
int      ix = (int) x;
short int y = -10213;
int      iy = (int) y;
```

	十进制	十六进制	二进制
x	10213	27 E5	00100111 11100101
ix	10213	00 00 27 E5	00000000 00000000 00100111 11100101
y	-10213	D8 1B	11011000 00011011
iy	-10213	FF FF D8 1B	11111111 11111111 11011000 00011011

- 由位数较少的整型向位数较多的整型转换
- C 自动完成符号扩展

小结:

扩展与截断的基本规则

- 扩展（例如由 `short` 型向 普通整型 转换）
 - `unsigned`: zeros added（零扩展）
 - `signed`: sign extension（符号扩展）
 - 结果都很合理
- 截断（如由 普通整型 向 `short` 型转换）
 - `unsigned/signed`: 多出的位直接丢弃
 - 对结果重新解析
 - `unsigned`: 求余运算`mod`
 - `signed`: 与求余运算类似
 - 对于较小的数，结果符合预期
 - 例: `(short) 32768` 与 `(short) -32768` 各自的结果？

本课内容

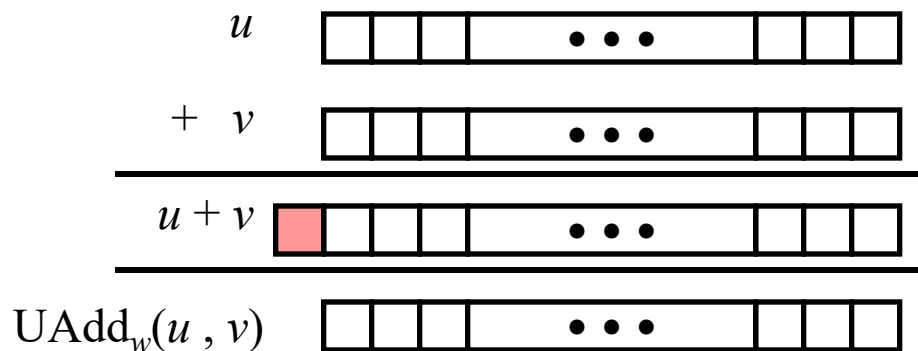
- 位操作
- **整型数**
 - 表示方法: signed 与 unsigned
 - 类型转换casting, 扩展expanding 与截断truncating
 - **加法、乘法、移位**
- 内存中的数据表示, 指针, 字符串
- **浮点数**
 - 背景知识: 二进制小数
 - IEEE 浮点数标准: 定义、举例与性质
 - 舍入, 加法, 乘法
 - C 语言的浮点类型

无符号数加法

操作数: w bits

真正的和: $w+1$ bits

丢弃进位: w bits



- 标准的 C 加法函数
 - 最高位产生的进位直接忽略
- 实行的是模运算

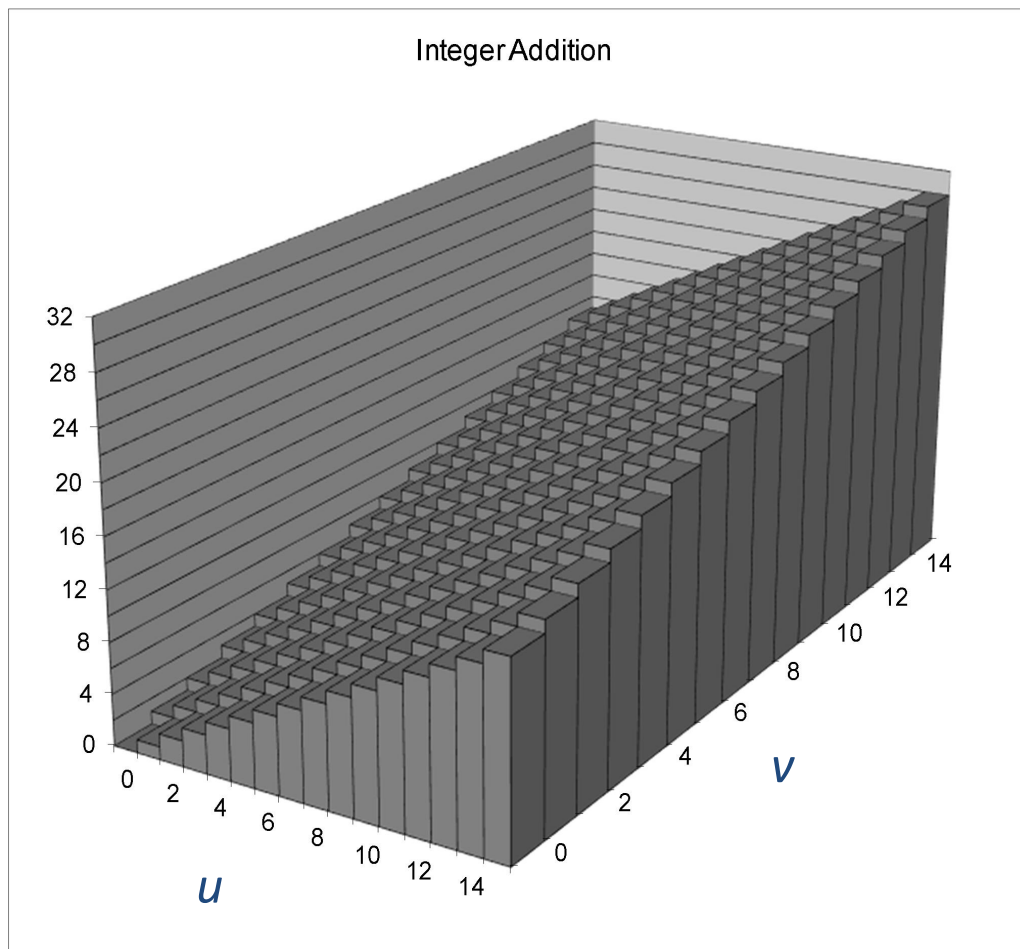
$$s = \text{UAdd}_w(u, v) = (u + v) \bmod 2^w$$

整数加法可视化

■ 整数的加法

- 4-bit integers u, v
- Compute true sum $\text{Add}_4(u, v)$
- Values increase linearly with u and v
- Forms planar surface

$$\text{Add}_4(u, v)$$

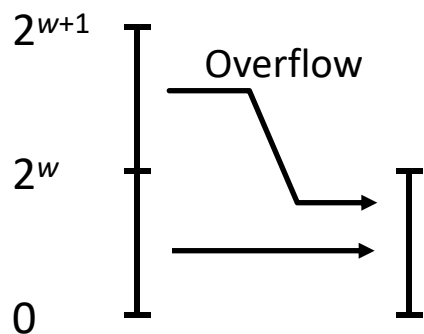


unsigned 型加法可视化

■ 模计数

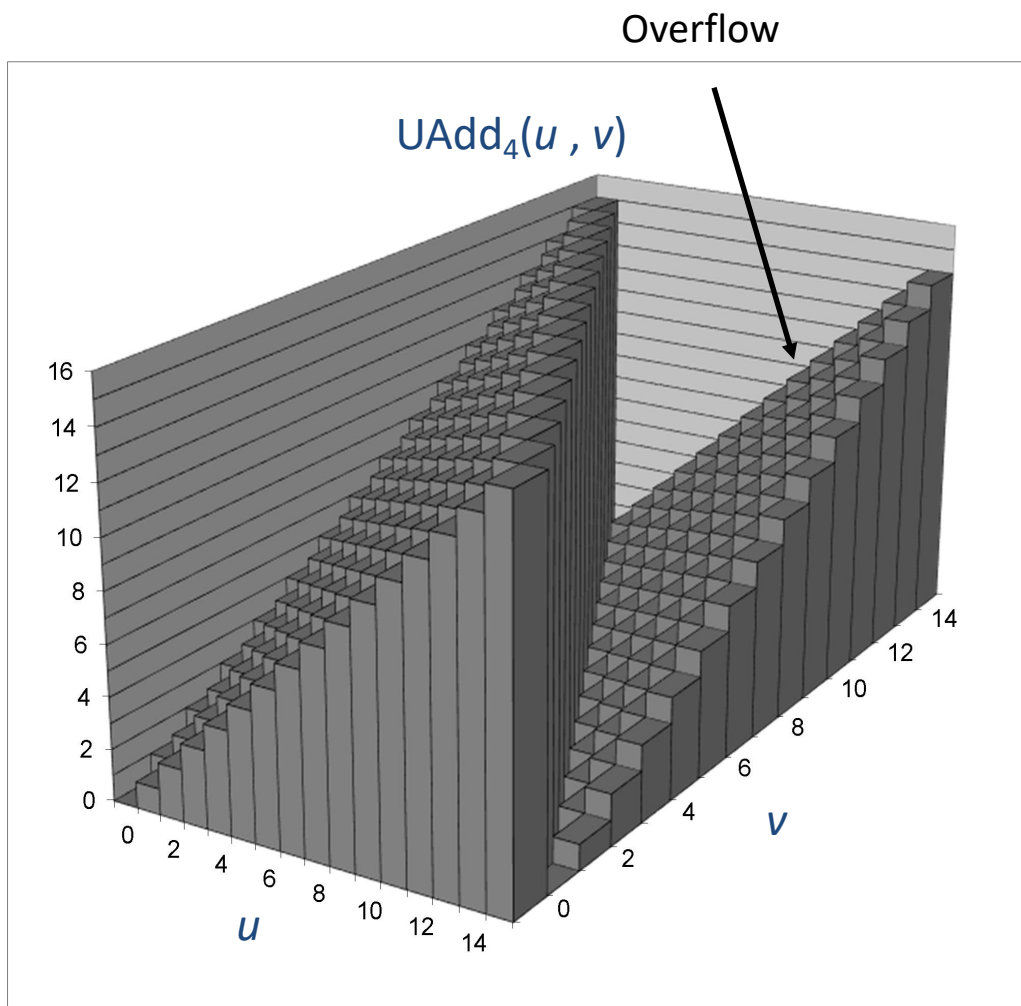
- 若真正的 $\text{sum} \geq 2^w$
- 则发生且最多发生一次环绕wrap around

True Sum



■ 无符号数加法溢出判定

- 充要条件：结果小于任一加数
- 练习题 2.27



补码加法

操作数: w bits

u 

+ v 

真正的和: $w+1$ bits

$u + v$ 

丢弃进位: w bits

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

- TAdd 和 UAdd 在 bit 层面上的运算方法完全一致

- C 的 signed 与 unsigned 加法对比

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

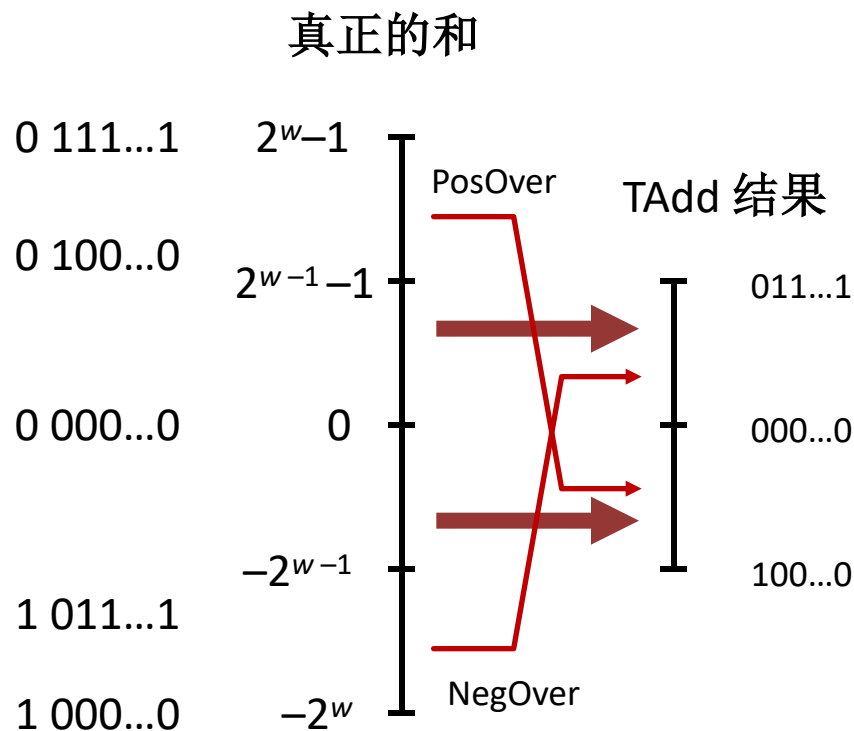
```
s = (int) ((unsigned) u + (unsigned) v);
```

```
t = u + v
```

- 对于上述代码, $s == t$ 一定为真

TAdd 溢出

- 功能
 - 真正的 sum 需要 $w+1$ 位才能保证容纳
 - 而 MSB 被丢弃
 - 剩余的位视作补码数
- C 语言带符号数加法溢出判定
 - 充要条件：两加数同号，结果与其异号
 - 练习题 2.30



补码加法可视化

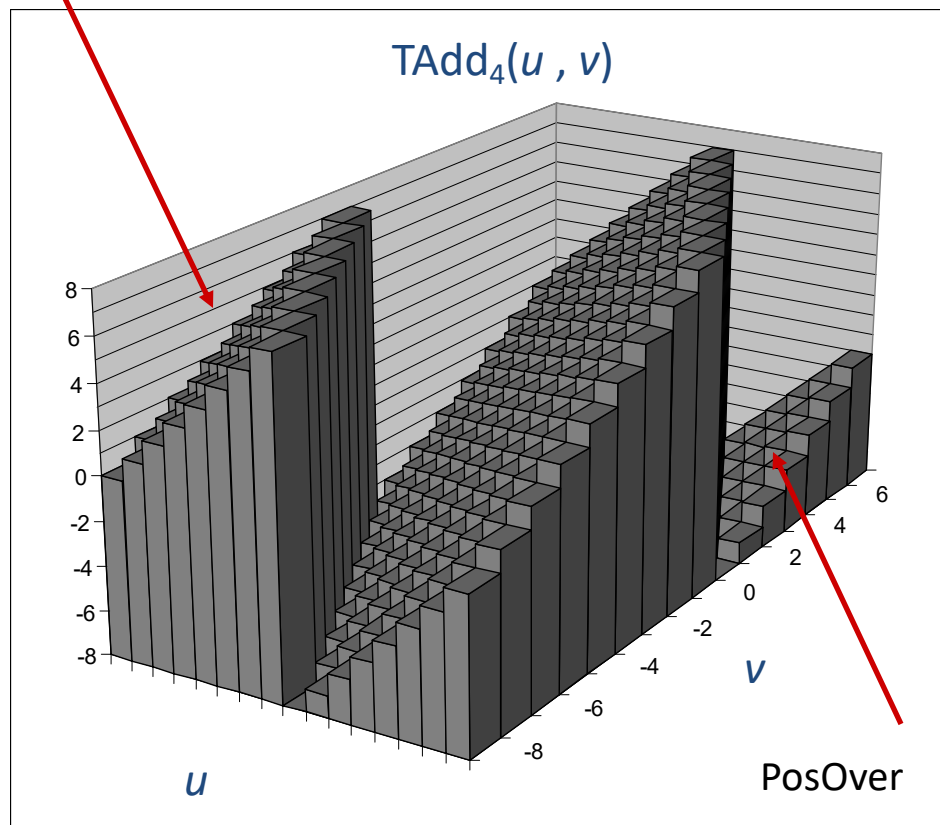
■ 取值

- 4 位补码
- 范围 $-8 \sim +7$

■ 模计数

- 若真正的 $\text{sum} \geq 2^{w-1}$
 - 变为负数（正溢）
 - 最多发生一次
- 若真正的 $\text{sum} < -2^{w-1}$
 - 变成正数（负溢）
 - 最多发生一次

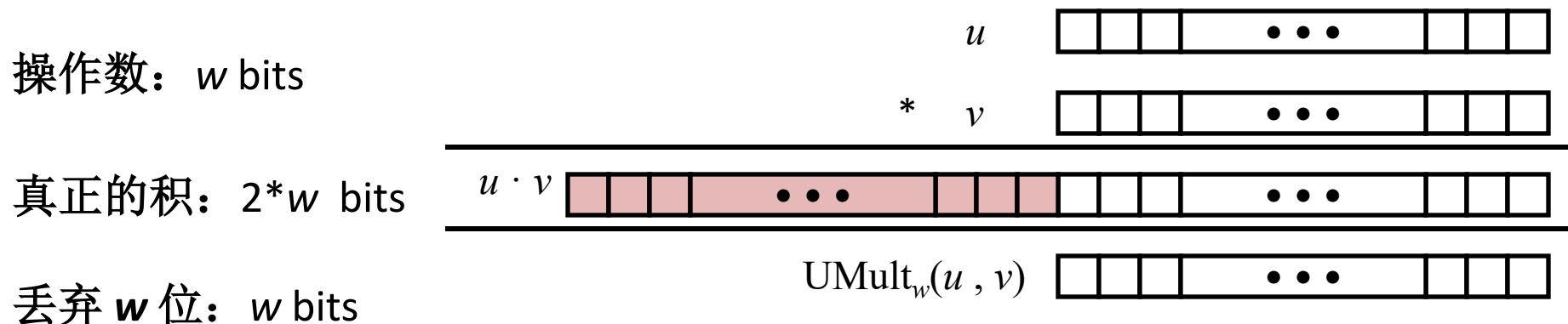
NegOver



乘法

- 目标：计算两个位宽为 w 的数 x 和 y 的积
 - 二者可能为带符号数或无符号数
- 但是，真正的结果有可能超出 w 位的表数范围
 - 无符号数：最多 $2w$ 位可容纳
 - 乘积范围： $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
 - 带符号数最小值（为负）： Up to $2w-1$ bits
 - 乘积范围： $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
 - 带符号数最大值（为正）： Up to $2w$ bits, but only for $(TMin_w)^2$
 - 乘积范围： $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
- 因此，若要保证得到正确的结果：
 - 每次计算乘法都要进行位数扩展
 - 有必要的話，由软件负责完成
 - 例如：使用“任意精度”算术运算包

C 语言的无符号数乘法



- 标准的 C 乘法函数
 - 高 w 位被丢弃（截断）
- 实行的是模运算

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

C 语言的带符号数乘法

操作数: w bits

u

$*$ v

真正的积: $2*w$ bits

$u \cdot v$

丢弃 w 位: w bits

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

- 标准的 C 乘法函数
 - 高 w 位被丢弃（截断）
 - 其中某些位，signed 和 unsigned 乘法是不同的
 - 低 w 位则是一致的

U2B	000	001	010	011	100	101	110	111
000	000	000	000	000	000	000	000	000
001	000	001	010	011	100	101	110	111
010	000	010	100	110	000	010	100	110
011	000	011	110	001	100	111	010	101
100	000	100	000	100	000	100	000	100
101	000	101	010	111	100	001	110	011
110	000	110	100	010	000	110	100	010
111	000	111	110	101	100	011	010	001

■ 以 3 位为例

- 上表为按无符号数的乘法保留低 3 位
- 下表为按带符号数的乘法保留低 3 位
- 教材图 2-27 可见被截断舍弃的高 3 位不尽相同

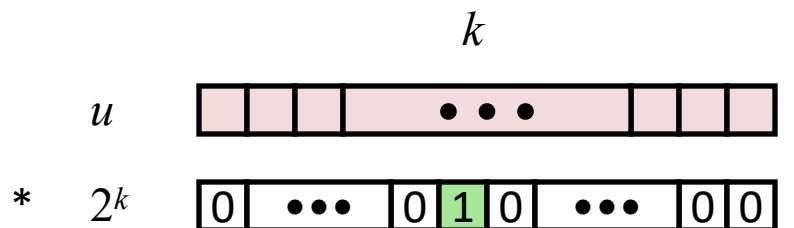
T2B	000	001	010	011	100	101	110	111
000	000	000	000	000	000	000	000	000
001	000	001	010	011	100	101	110	111
010	000	010	100	110	000	010	100	110
011	000	011	110	001	100	111	010	101
100	000	100	000	100	000	100	000	100
101	000	101	010	111	100	001	110	011
110	000	110	100	010	000	110	100	010
111	000	111	110	101	100	011	010	001

用移位实现与 2 的幂的乘积

■ 操作

- $u \ll k$ 得到 $u * 2^k$
- 对 signed 和 unsigned 均适用

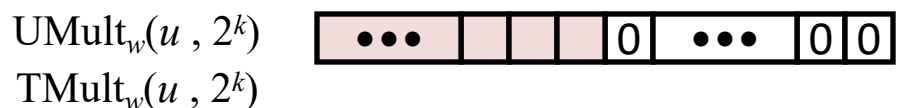
操作数: w bits



真正的积: $w+k$ bits



丢弃 k 位: w bits



■ 举例

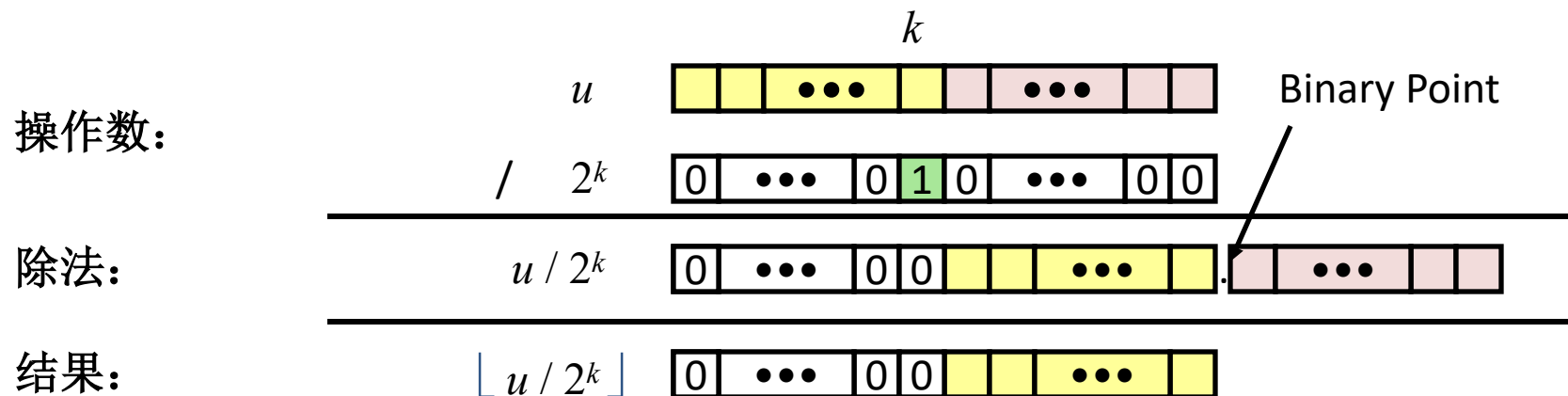
- $u \ll 3 \quad == \quad u * 8$
- $(u \ll 5) - (u \ll 3) \quad == \quad u * 24$
- 在大多数机器上, 移位和加法的速度快于乘法
 - 编译器会自动优化此类代码

用移位实现 unsigned 除以 2 的幂

■ unsigned 除以 2 的幂

■ $u \gg k$ 得到 $\lfloor u / 2^k \rfloor$

■ 使用逻辑移位



	真实商值	运算结果	十六进制	二进制
x	10213	10213	27 E5	00100111 11100101
x >> 1	5106.5	5106	13 F2	00010011 11110010
x >> 4	638.3125	638	02 7E	00000010 01111110
x >> 8	39.89453125	39	00 27	00000000 00100111

算术运算：基本规则

■ 加法

- **unsigned/signed**: 先正常完成加法运算，再截断
 - 在 bit 层面上二者完全相同
- **unsigned**: 加法 $\text{mod } 2^w$
 - 数学加法，可能会减去 2^w
- **signed**: **modified addition mod 2^w (result in proper range)**
 - 数学加法，可能会加上或减去 2^w

■ 乘法

- **unsigned/signed**: 先正常完成乘法，再截断
 - 在 bit 层面上二者完全相同
- **unsigned**: 数学乘法 $\text{mod } 2^w$
- **signed**: **modified multiplication mod 2^w (result in proper range)**

何时使用 `unsigned` 类型？

- 在未理解相关的隐含操作之前**不要**用

- 容易出错，例如：

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

- 有可能在细节上出问题，例如：

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

用 unsigned 进行倒数

- 用 unsigned 作循环控制变量的正确方法

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

- 见 Robert Seacord 所著 *Secure Coding in C and C++*

- C 标准保证：无符号数加法按照模运算进行

- $0 - 1 \rightarrow \text{UMax}$

- 更好的写法

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

- `size_t` 类型的定义：unsigned int 型（追一下）
- 即使 `cnt` 取到 `Umax`，代码依旧正确
- 思考：如果 `cnt` 为带符号数且 < 0 会发生什么？

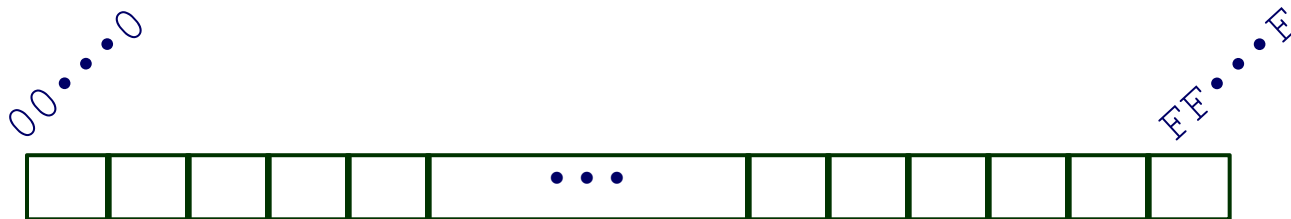
何时使用 `unsigned` 类型？（续）

- 进行代数模运算时使用
 - 多精度数值运算
 - 数论、密码学中的应用
- 当希望用位代表开关状态，而非数值时使用
 - 例：标志位、权限位
- 在用位串表示集合时使用
- 逻辑右移，非采用符号位填充

本课内容

- 位操作
- 整型数
 - 表示方法: signed 与 unsigned
 - 类型转换casting, 扩展expanding 与截断truncating
 - 加法、乘法、移位
- 内存中的数据表示, 指针, 字符串
- 浮点数
 - 背景知识: 二进制小数
 - IEEE 浮点数标准: 定义、举例与性质
 - 舍入, 加法, 乘法
 - C 语言的浮点类型

基于字节的内存组织



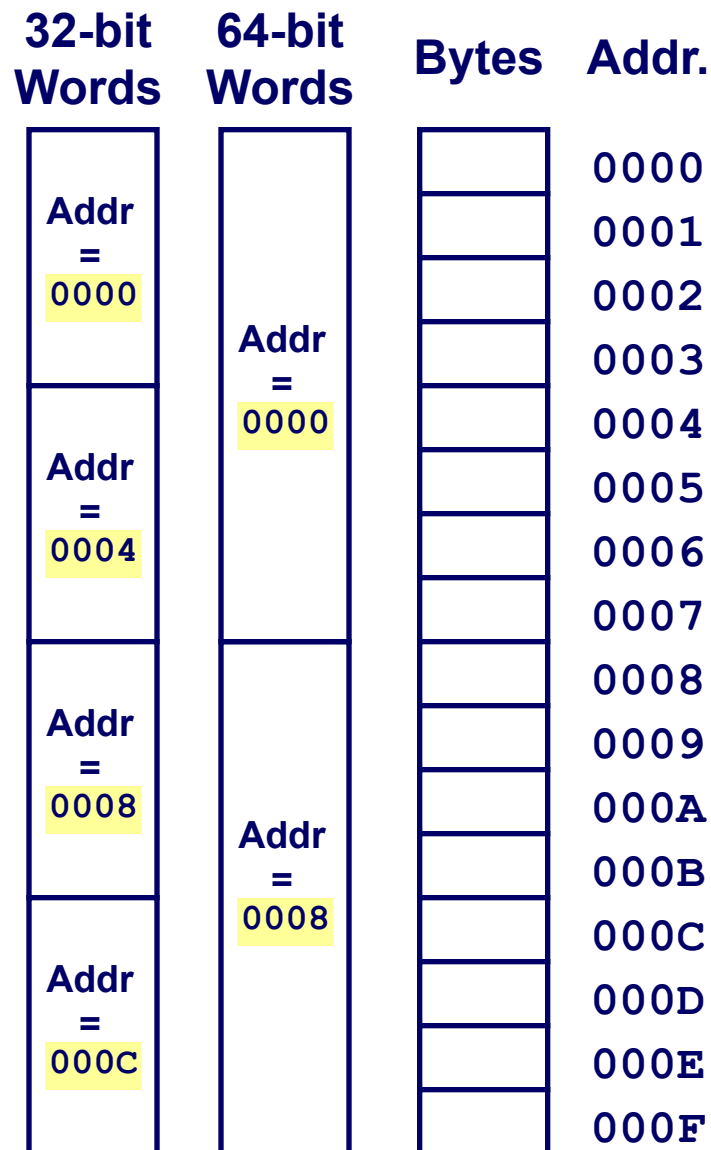
- 程序依据地址访问数据
 - 可将内存视作一个非常大的字节数组
 - 实际上并不是，但仍不妨这样理解
 - 地址即该数组各元素的索引（下标）
 - 每个指针变量存储一个地址
- 注意：系统会给每个进程一个私有的地址空间
 - 进程即运行中的程序
 - 因此，程序仅访问属于自己的数据，而非其它程序的数据

机器字

- 任何一台机器都有其**字长**
 - 由该机的整数处理能力决定
 - 以及地址处理能力（指针的位数）
- 上一代多为 32 位机，即以 32 位为字长（4 字节）
 - 地址范围限定在 4 GB (2^{32} bytes) 以内
- 越来越多的机型采用 64 位字长
 - 理论上可配备 18 EB (exabytes) 的内存
 - 约为 18.4×10^{18} B
- 系统能够支持多种数据格式
 - 本机字长的一部分或若干倍
 - 一定是字节的整数倍
 - 由软硬件共同决定

内存的字地址

- 地址是字节的地址（以字节为单位编地址）
 - “按字编址”是推广的说法
 - 字地址：字内地址最小的字节的地址
 - 相邻字的地址差 4（32位）或差 8（64 位）
 - （CMU 课堂字幕勘误：编译器 **尽最大可能** work hard to 保持数据对齐（不是**难以**保持对齐），以达到更好的性能）



数据表示举例

C 数据类型	典型 32 位机	典型 64 位机	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

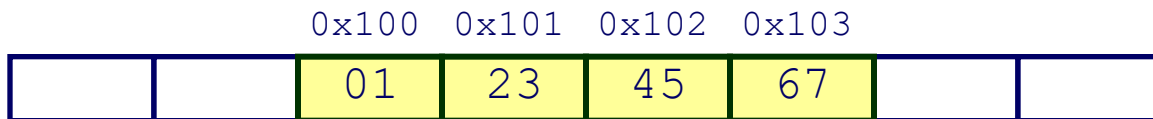
- 字内部各字节如何排序
- 两种约定
 - 大端序Big Endian: Sun、PPC Mac、Internet
 - 低位放在高地址
 - 小端序Little Endian: x86、运行安卓的 ARM 处理器、iOS、Windows
 - 低位放在低地址

字节序举例

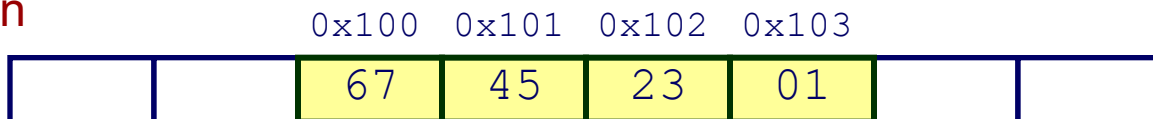
■ Example

- 变量 `x` 占 4 字节，其值为 `0x01234567`
- `&x` 算得的地址为 `0x100`

Big Endian



Little Endian



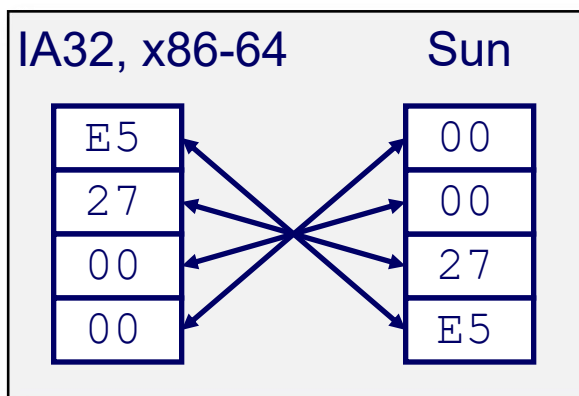
整型数的表示

Decimal: 10213

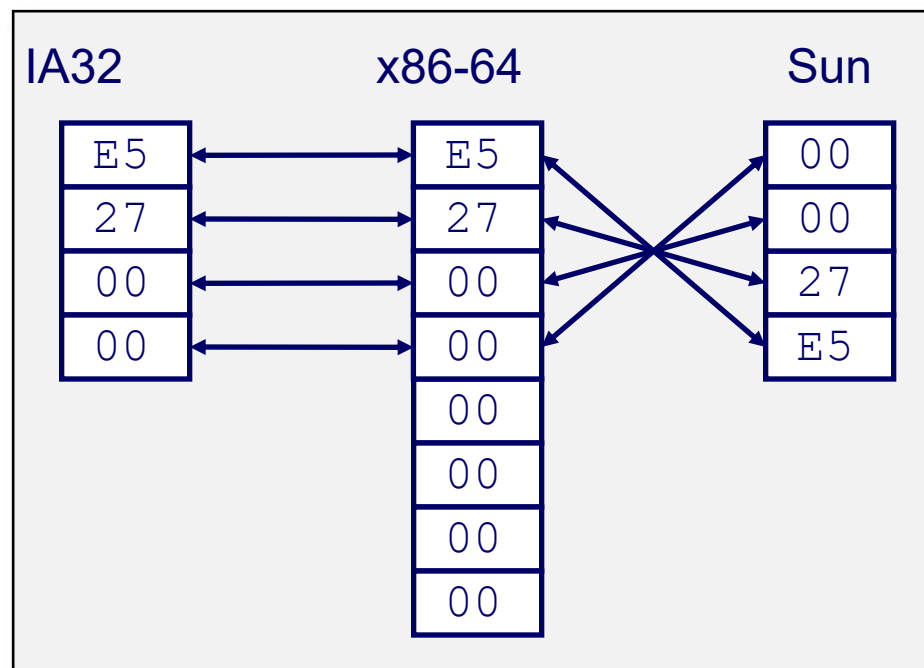
Binary: 0010 0111 1110 0101

Hex: 2 7 E 5

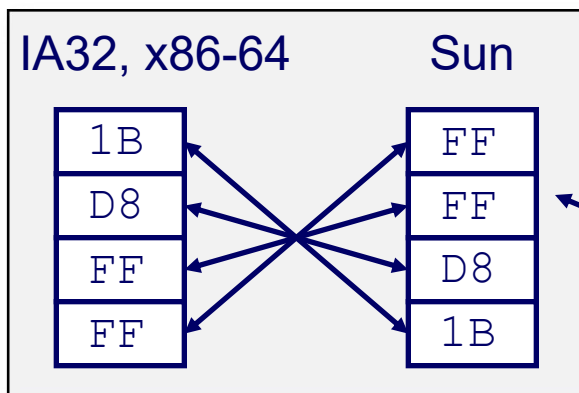
`int A = 10213;`



`long int C = 10213;`



`int B = -10213;`



补码表示

查看本机的数据表示

- 显示数据的字节表示的代码
 - 将其它指针转换为 `unsigned char *` 类型，将该数据按字节数组处理

```
typedef unsigned char *byte_pointer;

void show_bytes(byte_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");
}
```

`%p`: 输出指针值
`%x`: 输出十六进制值

show_bytes 执行举例

```
int a = 10213;  
printf("int a = 10213;\n");  
show_bytes((byte_pointer) &a, sizeof(int));
```

运行结果（openEuler x86-64）：

```
int a = 10213;  
0x7ffc338256ec 0xe5  
0x7ffc338256ed 0x27  
0x7ffc338256ee 0x00  
0x7ffc338256ef 0x00
```

指针的表示

```
int B = -10213;  
int *P = &B;
```

Sun	IA32	x86-64
EF	AC	3C
FF	28	1B
FB	F5	FE
2C	FF	82
		FD
		7F
		00
		00

不同编译器-处理器组合为对象分配的地址不尽相同
即使在同一台机器上，程序每次运行时的地址也可能不同

字符串的表示

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

- C 语言的字符串

- 表示为字符数组

- 每个字符以 ASCII 编码表示

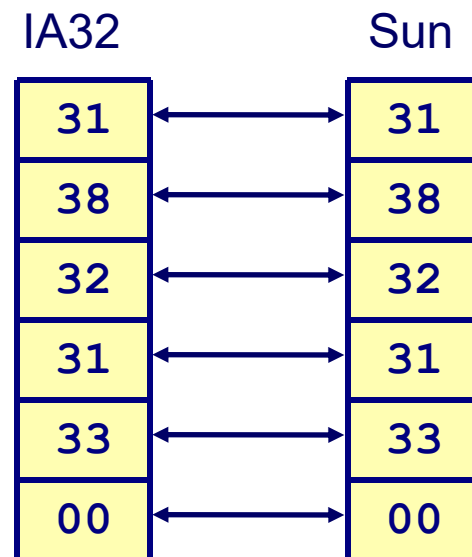
- Standard 7-bit encoding of character set
 - Character '0' has code 0x30
 - Digit '*i*' has code 0x30 + *i*

- 字符串必以空字符结束

- 最后一个字符值为 0 ('\\0')

- 兼容性














- 字节序不影响字符串的存储



练习：C 整型数谜题

初始化

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

- $x < 0 \rightarrow ((x*2) < 0)$ 
- $ux \geq 0$ 
- $x \& 7 == 7 \rightarrow (x \ll 30) < 0$ 
- $ux > -1$ 
- $x > y \rightarrow -x < -y$ 
- $x * x \geq 0$ 
- $x > 0 \&\& y > 0 \rightarrow (x+y) > 0$ 
- $x \geq 0 \rightarrow -x \leq 0$ 
- $x \leq 0 \rightarrow -x \geq 0$ 
- $(x | -x) \gg 31 == -1$ 
- $ux \gg 3 == ux / 8$ 
- $x \gg 3 == x / 8$ 
- $x \& (x-1) != 0$ 

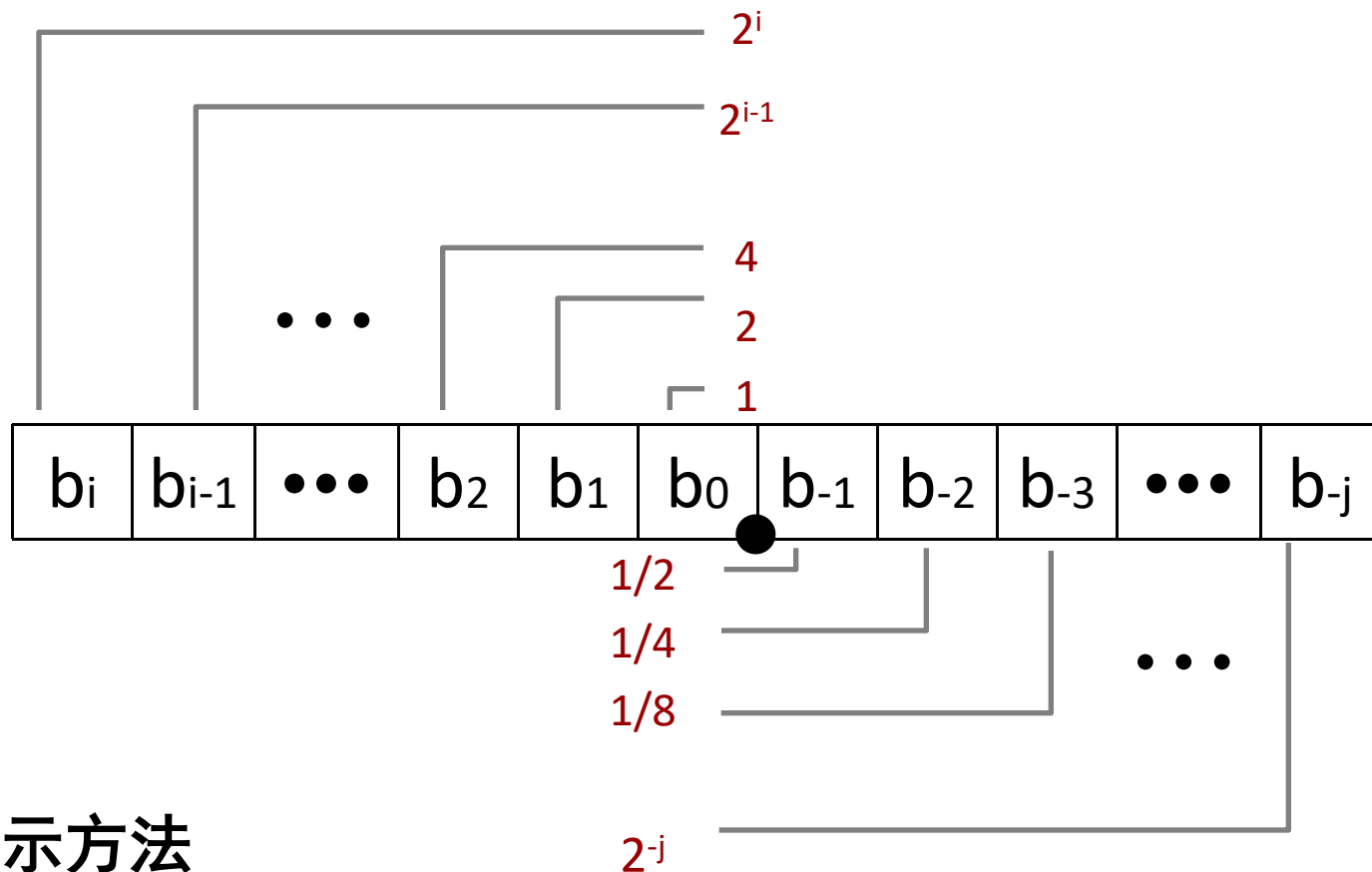
本课内容

- 位操作
- 整型数
 - 表示方法: signed 与 unsigned
 - 类型转换casting, 扩展expanding 与截断truncating
 - 加法、乘法、移位
- 内存中的数据表示, 指针, 字符串
- 浮点数
 - 背景知识: 二进制小数
 - IEEE 浮点数标准: 定义、举例与性质
 - 舍入, 加法, 乘法
 - C 语言的浮点类型

二进制小数

- What is 1011.101_2 ?

二进制小数



■ 表示方法

- Bits to right of “binary point” represent fractional powers of 2

- Represents rational number:
$$\sum_{k=-j}^i b_k \times 2^k$$

二进制小数举例

■ 真值

5 3/4

2 7/8

1 7/16

二进制表示

101.11₂

10.111₂

1.0111₂

■ 规律

- Divide by 2 by shifting right (unsigned)
- Multiply by 2 by shifting left
- Numbers of form 0.111111...₂ are just below 1.0
 - $1/2 + 1/4 + 1/8 + \dots + 1/2^i + \dots \rightarrow 1.0$
 - Use notation $1.0 - \epsilon$

可表示的数

- 局限 #1

- 只能精确表示形如 $x/2^k$ 的有理数

- Other rational numbers have repeating bit representations

- 真值 二进制表示

- $1/3$ $0.0101010101[01]..._2$
- $1/5$ $0.001100110011[0011]..._2$
- $1/10$ $0.0001100110011[0011]..._2$

- 局限 #2

- Just one setting of binary point within the w bits

- Limited range of numbers (very small values? very large?)

本课内容

- 信息的二进制表示
- 位操作
- 整型数
 - 表示方法: signed 与 unsigned
 - 类型转换casting, 扩展expanding 与截断truncating
 - 加法、乘法、移位
- 内存中的数据表示, 指针, 字符串
- 浮点数
 - 背景知识: 二进制小数
 - IEEE 浮点数标准: 定义、举例与性质
 - 舍入, 加法, 乘法
 - C 语言的浮点类型

IEEE 浮点数标准

- IEEE 754 标准
 - Established in 1985 as uniform standard for floating point arithmetic
 - Before that, many idiosyncratic formats
 - Supported by all major CPUs
- 由数值计算需求驱动
 - Nice standards for rounding, overflow, underflow
 - Hard to make fast in hardware
 - Numerical analysts predominated over hardware designers in defining standard

浮点表示

- 数值形式:

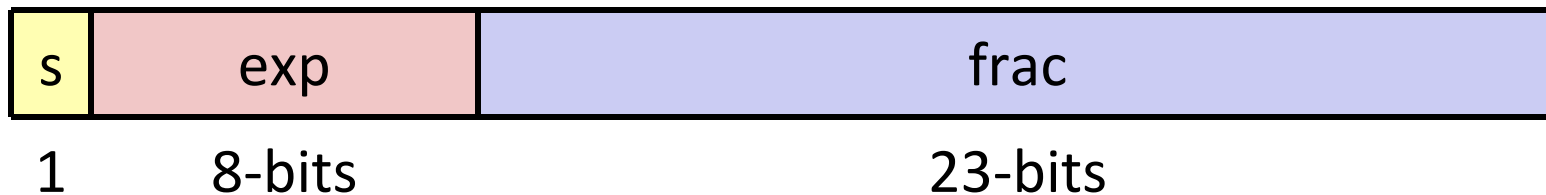
$$(-1)^s M 2^E$$

- 符号 Sign s determines whether number is neg or pos
- 尾数 Significand M normally a fractional value in range $[1.0, 2.0)$
- 指数 Exponent E weights value by power of two
- 二进制编码
 - 符号位 s is sign s
 - 阶码 exp field encodes E (but is not equal to E)
 - 尾数 frac field encodes M (but is not equal to M)



精度种类

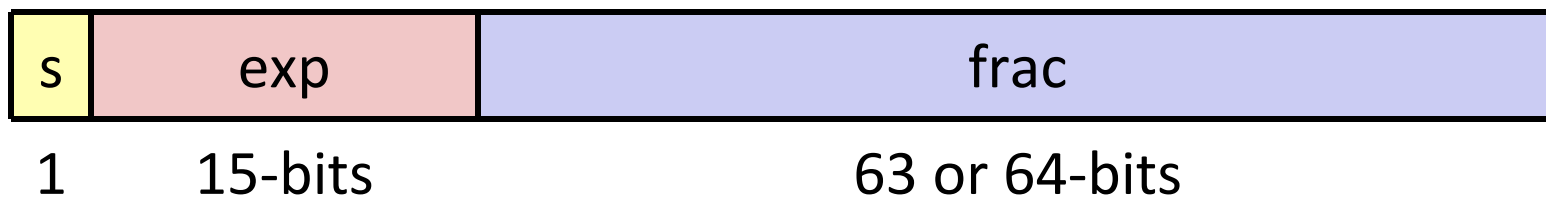
- **单精度：32 bits**



- **双精度：64 bits**



- **扩展精度：80 bits（仅限 Intel）**



规格化部Normalized Values

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

- $\text{exp} \neq 000\dots 0$ 且 $\text{exp} \neq 111\dots 1$ 的部分
- 指数的编码：满足 $E = \text{exp} - \text{bias}$
 - exp : unsigned value of exp field
 - $\text{bias} = 2^{k-1} - 1$, where k is number of exponent bits
 - Single precision: 127 (exp : 1 ... 254, E : -126 ... 127)
 - Double precision: 1023 (exp : 1 ... 2046, E : -1022 ... 1023)
- 尾数的编码（**隐含前置 1**）： $M = 1.\text{xxx}\dots\text{x}_2$
 - $\text{xxx}\dots\text{x}$: bits of frac field
 - Minimum when $\text{frac} = 000\dots 0$ ($M = 1.0$)
 - Maximum when $\text{frac} = 111\dots 1$ ($M = 2.0 - \epsilon$)
 - Get extra leading bit for “free”

规格化部数据编码实例

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

$$E = \text{exp} - \text{bias}$$

■ 真值: float $F = 10213.0;$

$$\begin{aligned} 10213_{10} &= 10011111100101_2 \\ &= 1.0011111100101_2 \times 2^{13} \end{aligned}$$

■ 尾数

$$M = 1.\underline{0011111100101}_2$$

$$\text{frac} = \underline{0011111100101}0000000000_2$$

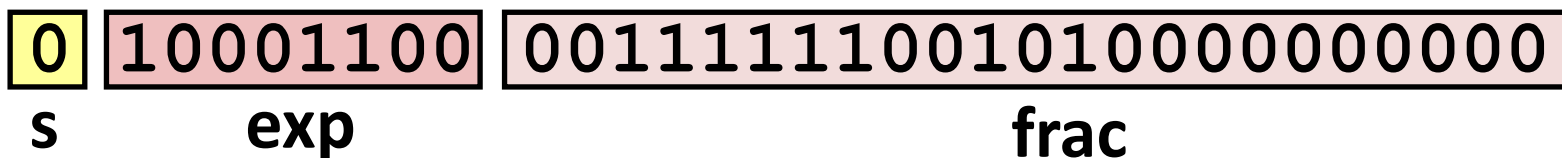
■ 阶码

$$E = 13$$

$$\text{bias} = 127$$

$$\text{exp} = 140 = 10001100_2$$

■ 结果:



非规格化部 Denormalized Values

- 条件: $\text{exp} = 000\dots 0$
- 指数 $E = 1 - \text{bias}$ (instead of $E = 0 - \text{bias}$)
- 尾数无隐含前置 1, Significand coded with implied leading 0:
 $M = 0.\text{xxx}\dots\text{x}_2$
 - $\text{xxx}\dots\text{x}$: bits of frac
- 实例
 - $\text{exp} = 000\dots 0, \text{frac} = 000\dots 0$
 - Represents zero value
 - Note distinct values: +0 and -0 (why?)
 - $\text{exp} = 000\dots 0, \text{frac} \neq 000\dots 0$
 - 最接近零的一批数
 - 在实数轴上离散、均匀分布Equispaced

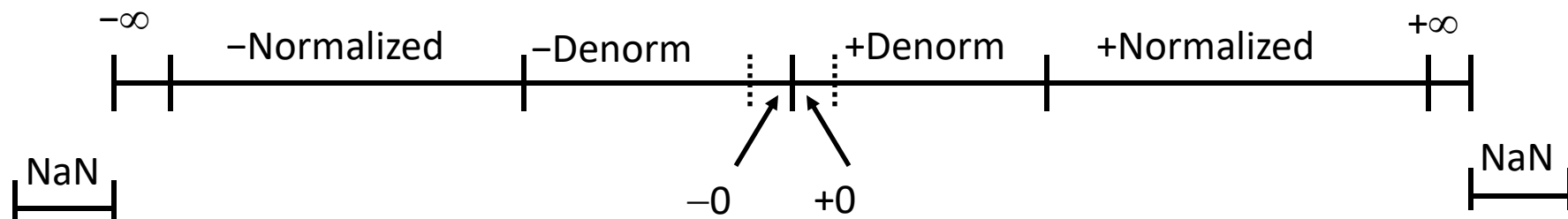
$$v = (-1)^s M 2^E$$

$$E = 1 - \text{bias}$$

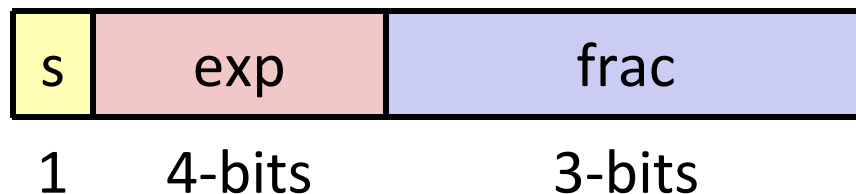
特殊数值部Special Values

- 条件: $\text{exp} = 111\dots 1$
- 实例: $\text{exp} = 111\dots 1, \text{frac} = 000\dots 0$
 - Represents value ∞ (infinity)
 - Operation that overflows
 - Both positive and negative
 - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty, 1.0/-0.0 = -\infty$
- 实例: $\text{exp} = 111\dots 1, \text{frac} \neq 000\dots 0$
 - Not-a-Number (NaN)
 - Represents case when no numeric value can be determined
 - E.g., $\text{sqrt}(-1), \infty - \infty, \infty \times 0$

可视化：浮点数编码



举例：缩小版浮点数模型



- **8-bit 浮点数表示**
 - the sign bit is in the MSB (most significant bit)
 - the next four bits are the exponent, with bias = 7
 - the last three bits are the frac
- **与 IEEE 格式类似，仅位数不同**
 - normalized, denormalized
 - representation of 0, NaN, infinity

表数枚举（仅正数区间）

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

规: $E = 1 - \text{bias}$

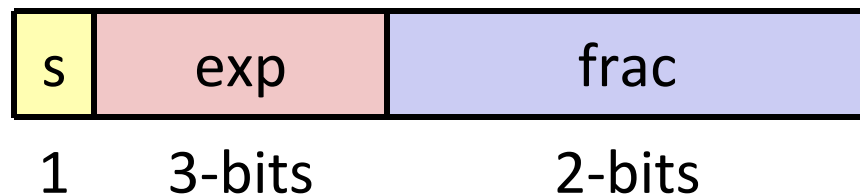
非: $E = \text{exp} - \text{bias}$

	s	exp	frac	指数	真值	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	$1/8 * 1/64 = 1/512$	closest to zero
	0	0000	010	-6	$2/8 * 1/64 = 2/512$	
	
	0	0000	110	-6	$6/8 * 1/64 = 6/512$	
	0	0000	111	-6	$7/8 * 1/64 = 7/512$	largest denorm
Normalized numbers	0	0001	000	-6	$8/8 * 1/64 = 8/512$	smallest norm
	0	0001	001	-6	$9/8 * 1/64 = 9/512$	
	
	0	0110	110	-1	$14/8 * 1/2 = 14/16$	
	0	0110	111	-1	$15/8 * 1/2 = 15/16$	closest to 1 below
	0	0111	000	0	$8/8 * 1 = 1$	
	0	0111	001	0	$9/8 * 1 = 9/8$	closest to 1 above
	0	0111	010	0	$10/8 * 1 = 10/8$	
	
	0	1110	110	7	$14/8 * 128 = 224$	
	0	1110	111	7	$15/8 * 128 = 240$	largest norm
	0	1111	000	n/a	inf	

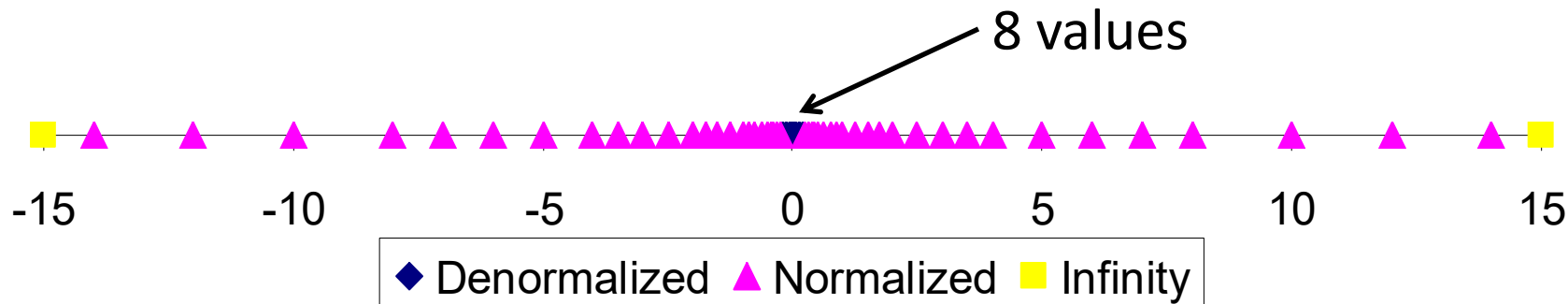
数值分布

■ 6-bit 类 IEEE 格式

- exp = 3 exponent bits
- frac = 2 fraction bits
- bias is $2^{3-1}-1 = 3$



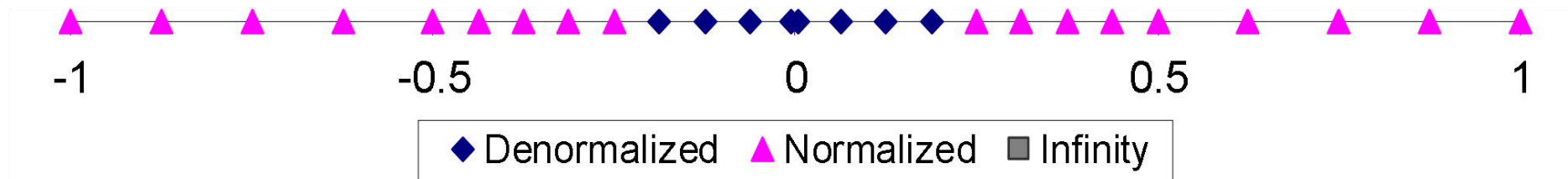
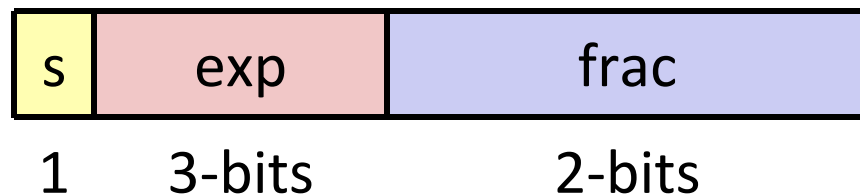
■ 注意观察：越是趋近 0，数值分布越是密集



数值分布（放大观察）

■ 6-bit 类 IEEE 格式

- exp = 3 exponent bits
- frac = 2 fraction bits
- bias is $2^{3-1}-1 = 3$



IEEE 浮点格式的特殊性质

- 浮点数 0 同整数 0
 - 所有位均为 0
- 方便比较：几乎可以照搬无符号整型数比较的结果
 - 必须首先比较符号位，且注意 $-0 = 0$
 - NaNs 是例外
 - Will be greater than any other values
 - What should comparison yield?
- 其它数（规格化、非规格化、 ∞ ）均适用
 - 正数区间：与相同位串的无符号数大小关系一致
 - 负数区间：与相同位串的无符号数大小关系相反
 - 一正一负： $s = 1$ 的 $\leq s = 0$ 的（相等发生在二者均为 0）

本课内容

- 信息的二进制表示
- 位操作
- 整型数
 - 表示方法: signed 与 unsigned
 - 类型转换casting, 扩展expanding 与截断truncating
 - 加法、乘法、移位
- 内存中的数据表示, 指针, 字符串
- 浮点数
 - 背景知识: 二进制小数
 - IEEE 浮点数标准: 定义、举例与性质
 - 舍入, 加法, 乘法
 - C 语言的浮点类型

浮点运算：基本思想

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

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

- 基本思想

- 首先**计算精确结果**
- 再作**精度适配（修正）**：将此结果在所要求的精度框架之内表示
 - Possibly overflow if exponent too large
 - Possibly **round to fit into** frac

舍入rounding

■ 舍入方式（以 ¥ 舍入作比）

	¥1.40	¥1.60	¥1.50	¥2.50	- ¥1.50
■ 向 0 舍入	¥1	¥1	¥1	¥2	- ¥1
■ 向下 ($-\infty$) 舍入	¥1	¥1	¥1	¥2	- ¥2
■ 向上 ($+\infty$) 舍入	¥2	¥2	¥2	¥3	- ¥1
■ 向最近偶数舍入 (默认)	¥1	¥2	¥2	¥2	- ¥2

向偶数舍入round-to-even

- 默认采取的舍入方式
 - Hard to get any other kind without dropping into assembly
 - All others are statistically biased
 - Sum of set of positive numbers will consistently be over- or under-estimated
- 同样适用于其它十（二）进制位
 - When exactly halfway between two possible values
 - Round so that least significant digit is even
 - E.g., round to nearest hundredth

7.8949999	7.89	(Less than half way)
7.8950001	7.90	(Greater than half way)
7.8950000	7.90	(Half way—round up)
7.8850000	7.88	(Half way—round down)

二进制数的舍入

- 二进制小数
 - “Even” when least significant bit is 0
 - “Half way” when bits to right of rounding position = 100...₂
- 例：向最近的整 1/4 舍入（小数点右第 2 位）

Value	Binary	Rounded	Action	Rounded Value
$2 \frac{3}{32}$	10.00 011 ₂	10.00 ₂	(<1/2—down)	2
$2 \frac{3}{16}$	10.00 110 ₂	10.01 ₂	(>1/2—up)	$2 \frac{1}{4}$
$2 \frac{7}{8}$	10.11 100 ₂	11.00 ₂	(1/2—up)	3
$2 \frac{5}{8}$	10.10 100 ₂	10.10 ₂	(1/2—down)	$2 \frac{1}{2}$

浮点数乘法

- $(-1)^{s1} M1 2^{E1} \times (-1)^{s2} M2 2^{E2}$
- 精确结果: $(-1)^s M 2^E$
 - Sign s : $s1 \wedge s2$
 - Significand M : $M1 \times M2$
 - Exponent E : $E1 + E2$
- 精度适配
 - If $M \geq 2$, shift M right, increment E
 - If E out of range, overflow
 - Round M to fit frac precision
- 实现
 - Biggest chore is multiplying significands

浮点数加法

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

▪ Assume $E1 > E2$

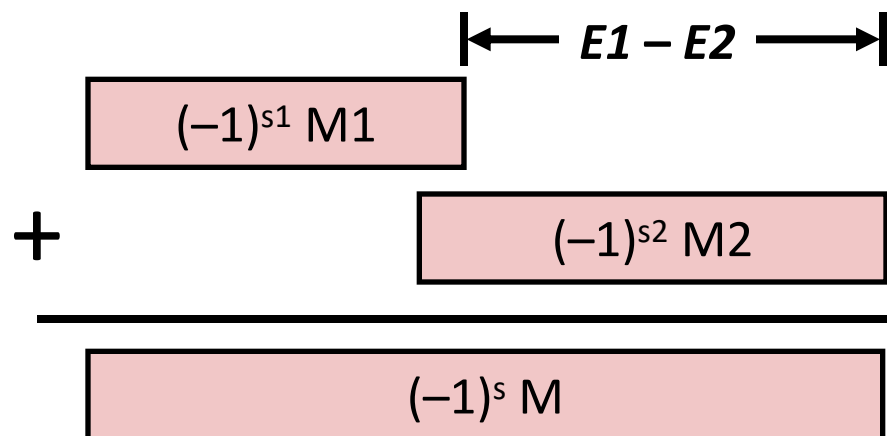
对齐小数点（对阶）

▪ 精确结果: $(-1)^s M 2^E$

▪ Sign s , significand M :

▪ Result of signed align & add

▪ Exponent E : $E1$



▪ If $M \geq 2$, shift M right, increment E

▪ If $M < 1$, shift M left k positions, decrement E by k

▪ Overflow if E out of range

▪ Round M to fit frac precision

浮点加法的数学性质

■ 相比阿贝尔群Abelian Group

■ 封闭性closed under addition?

Yes

- But may generate infinity or NaN

■ 交换律commutative?

Yes

■ 结合律associative?

No

- Overflow and inexactness of rounding

$$\text{■ } (3.14 + 1e10) - 1e10 = 0 \quad 3.14 + (1e10 - 1e10) = 3.14$$

■ 以 0 为单位元additive identity?

Yes

■ 每个元素都有逆元additive inverse?

Almost

- Yes, except for infinities & NaNs

■ 单调性

Almost

- $a \geq b \Rightarrow a+c \geq b+c$? (Yes, except for infinities & NaNs)

浮点乘法的数学性质

■ 相比交换环commutative ring

- 封闭性closed under multiplication? Yes
 - But may generate infinity or NaN
- 交换律commutative? Yes
- 结合律associative? No
 - Possibility of overflow, inexactness of rounding
 - E.g.: $(1e20 * 1e20) * 1e-20 = \text{inf}$, $1e20 * (1e20 * 1e-20) = 1e20$
- 乘法以 1.0 为单位元multiplicative identity? Yes
- 对加法分配律distribute over addition? No
 - Possibility of overflow, inexactness of rounding
 - $1e20 * (1e20 - 1e20) = 0.0$, $1e20 * 1e20 - 1e20 * 1e20 = \text{NaN}$

■ 单调性

- $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c$? Almost
 - Except for infinities & NaNs

本课内容

- 信息的二进制表示
- 位操作
- 整型数
 - 表示方法: signed 与 unsigned
 - 类型转换casting, 扩展expanding 与截断truncating
 - 加法、乘法、移位
- 内存中的数据表示, 指针, 字符串
- 浮点数
 - 背景知识: 二进制小数
 - IEEE 浮点数标准: 定义、举例与性质
 - 舍入, 加法, 乘法
 - C 语言的浮点类型

C 语言的浮点类型











- C 提供两种浮点类型: `float` 和 `double`
 - 但 C 不规定必须使用 IEEE 754
 - 支持 754 的机器, `float` 对应单精度, `double` 为双精度
- 类型转换
 - Casting between `int`, `float`, and `double` changes bit representation
 - `double/float` \rightarrow `int`
 - Truncates fractional part
 - Like rounding toward zero
 - Not defined when out of range or NaN: Generally sets to *TMin*
 - `int` \rightarrow `double`
 - Exact conversion, as long as `int` has ≤ 53 bit word size
 - `int` \rightarrow `float`
 - Will round according to rounding mode

浮点数谜题

- 以下 C 表达式是否一定为真？若否请解释（尽量给出反例）
 - Argue that it is true for all argument values
 - Explain why not true

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

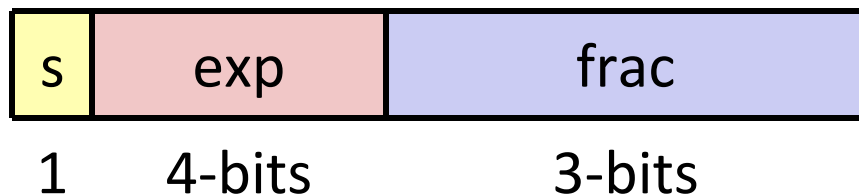
假定 d 和 f 均不为 NaN

- | | |
|--|---|
| • <code>x == (int)(float) x</code> |  |
| • <code>x == (int)(double) x</code> |  |
| • <code>f == (float)(double) f</code> |  |
| • <code>d == (double)(float) d</code> |  |
| • <code>f == -(-f);</code> |  |
| • <code>2/3 == 2/3.0</code> |  |
| • <code>d < 0.0 ⇒ ((d*2) < 0.0)</code> |  |
| • <code>d > f ⇒ -f > -d</code> |  |
| • <code>d * d >= 0.0</code> |  |
| • <code>(d + f) - d == f</code> |  |

小结

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form $M \times 2^E$
- One can reason about operations independent of implementation
 - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
 - Violates associativity/distributivity
 - Makes life difficult for compilers & serious numerical applications programmers

对浮点数编码



■ 步骤

1. Normalize to have leading 1
2. Round to fit within fraction
3. Postnormalize to deal with effects of rounding

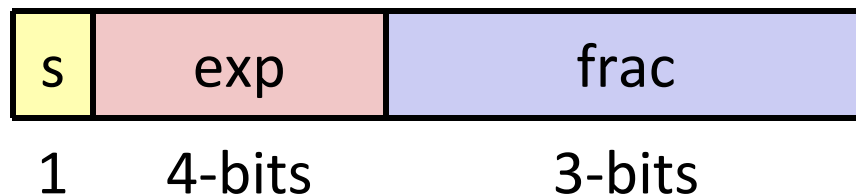
■ 实例

- Convert 8-bit unsigned numbers to tiny floating point format

Example Numbers

128	10000000
15	00001101
33	00010001
35	00010011
138	10001010
63	00111111

规格化



■ 要求

- Set binary point so that numbers of form 1.xxxxx
- Adjust all to have leading one
 - Decrement exponent as shift left

Value	Binary	Fraction	Exponent
128	10000000	1.0000000	7
15	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

舍入

1 . BBG**RXXX**

Guard bit: LSB of result

Round bit: 1st bit removed

Sticky bit: OR of remaining bits

■ Round up conditions

- Round = 1, Sticky = 1 → > 0.5
- Guard = 1, Round = 1, Sticky = 0 → Round to even

Value	Fraction	GRS	Incr?	Rounded
128	1.000 0000	000	N	1.000
15	1.101 0000	100	N	1.101
17	1.000 1000	010	N	1.000
19	1.001 1000	110	Y	1.010
138	1.000 1010	011	Y	1.001
63	1.111 1100	111	Y	10.000

后规格化

■ 问题

- Rounding may have caused overflow
- Handle by shifting right **once** & incrementing exponent

Value	Rounded	Exp	Adjusted	Result
128	1.000	7		128
15	1.101	3		15
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	1.000/6	64

- 试用 GDB 检查 (int) (float) T_{max} 的值并思考原因

有意思的数

{single, double}

	exp	frac	Numeric Value
■ 0	00...00	00...00	0.0
■ Smallest Pos. Denorm.	00...00	00...01	$2^{-\{23,52\}} \times 2^{-\{126, 1022\}}$
■ Single $\approx 1.4 \times 10^{-45}$			
■ Double $\approx 4.9 \times 10^{-324}$			
■ Largest Denormalized	00...00	11...11	$(1.0 - \epsilon) \times 2^{-\{126, 1022\}}$
■ Single $\approx 1.18 \times 10^{-38}$			
■ Double $\approx 2.2 \times 10^{-308}$			
■ Smallest Pos. Normalized	00...01	00...00	$1.0 \times 2^{-\{126, 1022\}}$
■ Just larger than largest denormalized			
■ 1	01...11	00...00	1.0
■ Largest Normalized	11...10	11...11	$(2.0 - \epsilon) \times 2^{\{127, 1023\}}$
■ Single $\approx 3.4 \times 10^{38}$			
■ Double $\approx 1.8 \times 10^{308}$			

有意思的题目

- 测试 C 语言计算 $1/0$ 和 $1.0/0.0$ 会产生什么结果
- `float/double` 能够精确表示的最大非整数是？
其对应的二进制编码为？
- 练习题 2.46，美军爱国者导弹拦截伊军飞毛腿导弹
失败原因分析
- 练习题 2.84，借用无符号数比较浮点数
- 设 $DMax$ 、 $Dmin$ 分别为双精度浮点数规格化部中的
最大、最小数
 - 写出 $DMax$ 和 $DMin$ 的值
 - 在区间 $[DMax, DMin]$ 之内是否有双精度格式无法
精确表示的整数？如果有，请举一例