



第2章 信息的表示与处理

100076202: 计算机系统导论

比特, 字节和整数

Bits, Bytes, and Integers

任课教师:

宿红毅 张艳 黎有琦 李秀星

原作者:

Randal E. Bryant and David R. O'Hallaron



**Carnegie
Mellon
University**

议题: 比特、字节和整数



Today : Bits, Bytes, and Integers

- **用比特表示信息** Representing information as bits
- **比特级操作** Bit-level manipulations
- **整数** Integer
 - 无符号数和有符号数表示 Representation: unsigned and signed
 - 转换和强制类型转换 Conversion, casting
 - 扩展和截断 Expanding, truncating
 - 加、补码非、乘和移位 Addition, negation, multiplication, shifting
 - 小结 Summary
- **内存中的表示、指针和字符串** Representations in memory, pointers, strings

十进制表示 The Decimal Representation

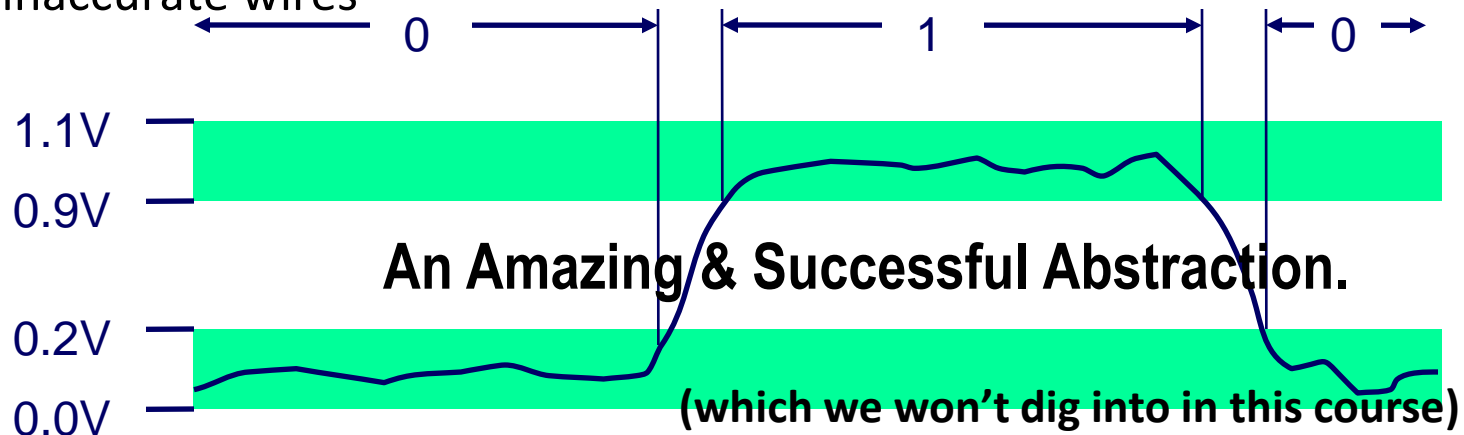


- **基数为10 Base-10**
- **已经使用了超过1000年 Has been in use for over 1000 years**
- **起源于印度 Developed in India**
- **12世纪被阿拉伯数学家改进 Improved by Arab mathematicians in the 12th century**
- **13世纪被带到西方 Brought to the West in the 13th century by**
 - 意大利数学家 the Italian mathematician Leonardo Pisano,
 - 更为大家熟悉的名字是斐波那契 better known as Fibonacci.

一切均是比特位 Everything is bits



- 每个比特是0或1 Each bit is 0 or 1
- 以各种方式编码/解释比特位集合 By encoding / interpreting sets of bits in various ways
 - 计算机确定要做什么（指令） Computers determine what to do (instructions)
 - 。 。 。 以及表示和操作数值、集合、字符串等。 。 。 ... and represent and manipulate numbers, sets, strings, etc...
- 为何是比特？ 电信号实现 Why bits? Electronic Implementation
 - 易于用稳态元件存储 Easy to store with bistable elements
 - 在有噪声和不精确的电缆中可靠传输 Reliably transmitted on noisy and inaccurate wires





举例：用二进制计数

For example, can count in binary

■ 基数为2的数值表示 Base 2 Number Representation

- 0, 1, 10, 11, 100, 101, ...
- 十进制整数表示为二进制整数 Represent 15213_{10} as 11101101101101_2
- 十进制小数表示为二进制小数 Represent 1.20_{10} as $1.0011001100110011[0011]..._2$
- 十进制科学计数法表示为二进制科学计数法 Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

■ 负数表示为。。。？ Represent negative numbers as ...?

二进制数的性质

Binary Number Property



声明/断言 Claim

$$1 + 1 + 2 + 4 + 8 + \dots + 2^{w-1} = 2^w$$

$$1 + \sum_{i=0}^{w-1} 2^i = 2^w$$

■ $w = 0$:

■ $1 = 2^0$

■ 假设对于w-1为真 Assume true for w-1:

■ $1 + 1 + 2 + 4 + 8 + \dots + 2^{w-1} + 2^w = 2^w + 2^w = 2^{w+1}$

$= 2^w$

多个位组成组 Group Bits



- 孤立地看，单个位不是很有用 In isolation, a single bit is not very useful
- 在英语中，它的字母表中有26（或52）个字符。它们单独使用也没有用 In English, there are 26(or 52) characters in its alphabet. They are not useful either in isolation
- 然而，它的词汇表中有很多单词，这是如何实现的？ However, there are plenty of words in its vocabulary. How is this achieved?
- 同样，我们可以使用多个位（而不是单个位）来表示任何有限集的元素 Similarly, we are able to represent the elements of any finite set by using bits (instead of bit)

多个位组成组 Group Bits



维纳·布赫霍尔兹

- **为此，我们** To do this, we
 - 首先把多个比特组合在一起 first group bits together
 - 然后应用某种解释给不同的可能位模式 then apply some *interpretation* to the different possible bit patterns
 - 给每个模式一个意义 that gives meaning to each patterns
- **8位数据块组织成一个字节** 8-bit chunks are organized as a byte
 - 1956年7月维纳博士提出 Dr. Werner Buchholz in July 1956
 - IBM草稿计算机早期设计阶段 during the early design phase for the IBM Stretch computer



多个位的值 Value of Bits

位序列 Bits

01010

值 Value

$$0*2^4+1*2^3+0*2^2+1*2^1+0*2^0 = 10$$

值 Value

102 (1100110)

位序列 Bits

$$102 = 51*2 + 0 \quad (0)$$

$$51 = 25*2 + 1 \quad (1)$$

$$25 = 12*2 + 1 \quad (1)$$

$$12 = 6*2 + 0 \quad (0)$$

$$6 = 3*2 + 0 \quad (0)$$

$$3 = 1*2 + 1 \quad (1)$$

$$1 = 0*2 + 1 \quad (1)$$

编码字节值 Encoding Byte Values



十六进制 Hex

十进制 Decimal

二进制 Binary

■ 一个字节包含8比特位 Byte = 8 bits

- 十进制取值范围 Decimal: 0_{10} to 255_{10}
 - $255 = 2^8 - 1$
- 二进制取值范围 Binary 00000000_2 to 11111111_2
- 十六进制取值范围 Hexadecimal 00_{16} to FF_{16}
 - 基数为16的数值表示Base 16 number representation
 - 字符0-9和A-F Use characters '0' to '9' and 'A' to 'F'
 - C语言中写作前导 '0x' , 以下情况之一
Write in C with leading '0x', either case
 - $0101\ 1010_2 = 0x5a = 0x5A = 0X5a$

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

15213: 0011 1011 0110 1101
 3 B 6 D

十六进制与二进制 Hexadecimal vs. Binary



0x173A4C

Hexadecimal	1	7	3	A	4	C
Binary	0001	0111	0011	1010	0100	1100

1111001010110110110011

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

0x3CADB3

十六进制与十进制 Hexadecimal vs. Decimal



Hexadecimal

0xA7

Decimal

$$10 \times 16 + 7 = 167$$

Decimal

$$314156 = 19634 \times 16 + 12 \quad (\text{C})$$

$$19634 = 1227 \times 16 + 2 \quad (2)$$

$$1227 = 76 \times 16 + 11 \quad (\text{B})$$

$$76 = 4 \times 16 + 12 \quad (\text{C})$$

$$4 = 0 \times 16 + 4 \quad (4)$$

Hexadecimal

0x4CB2C

十六进制与二进制 Hexadecimal vs. Binary



- 1100100101111011 -> C97B
- 1001101110011110110101 -> 2 6 E 7 B 5

十进制、十六进制和二进制

Decimal, Hexadecimal, Binary



Decimal	Binary	Hexadecimal
62	00111110	0x3E
$3 \times 16 + 7 = 55$	0011 0111	0x37
$5 \times 16 + 2 = 82$	01010010	0x52



十六进制 Hexadecimal

- $0x503c + 0x8 = 0x5044$
- $0x503c - 0x40 = 0x4ffc$
- $0x503c + 64 = 0x507c$
- $0x50ea - 0x503c = 0xae$

组合字节以创建标量数据类型

Combine bytes to make *scalar data types*



C Data Type	大小(字节数)Size(# of bytes)	
	Typical 32-bit	Typical 64-bit
char	1	1
short	2	2
int	4	4
long	4	8
float	4	4
double	8	8
pointer	4	8

“ILP32”

“LP64”

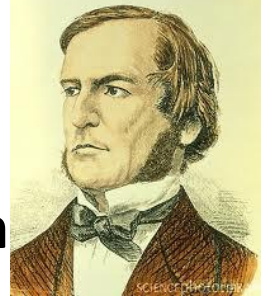


议题: 比特、字节和整数

Today : Bits, Bytes, and Integers

- **用比特表示信息** Representing information as bits
- **比特级操作** Bit-level manipulations
- **整数** Integers
 - 无符号数和有符号数表示 Representation: unsigned and signed
 - 转换和强制类型转换 Conversion, casting
 - 扩展和截断 Expanding, truncating
 - 加、补码非、乘和移位 Addition, negation, multiplication, shifting
 - 小结 Summary
- **内存中的表示、指针和字符串** Representations in memory, pointers, strings

布尔代数 Boolean Algebra



■ 19世纪由布尔开发 Developed by George Boole in 19th Century

- 逻辑的代数表示 Algebraic representation of logic
 - 逻辑值真和假编码为1和0 Encode “True” as 1 and “False” as 0

与 And

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

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

非 Not

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

\sim	
0	1
1	0

或 Or

- $A | B = 1$ when either $A=1$ or $B=1$

$ $	0	1
0	0	1
1	1	1

异或 Exclusive-Or (Xor)

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

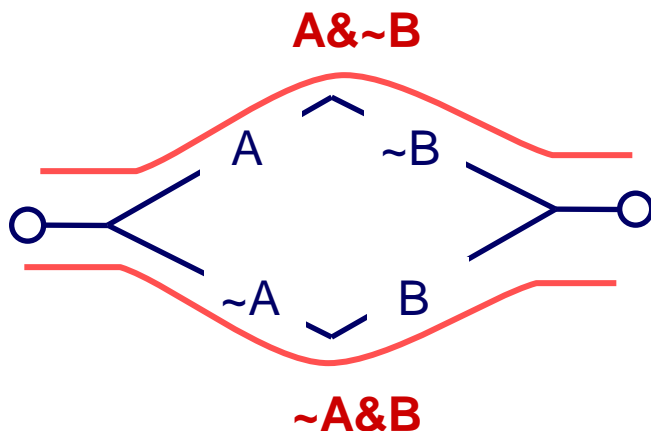
\wedge	0	1
0	0	1
1	1	0

布尔代数的应用

Application of Boolean Algebra



- 由香农应用到数字系统中，信息论的奠基人 Applied to Digital Systems by Claude Shannon, founded the information theory
 - 1937年MIT硕士论文 1937 MIT Master's Thesis
 - 对继电器开关网络进行推理 Reason about networks of relay switches
 - 开关闭合编码为1，开关打开为0 Encode closed switch as 1, open switch as 0



当满足下面条件是连通
Connection when

$$A \& \sim B \mid \sim A \& B \\ = A \wedge B$$

一般布尔代数

General Boolean Algebras



■ 对比特位向量进行运算 Operate on Bit Vectors

- 将运算运用到每个比特位 Operations applied bitwise

01101001	01101001	01101001	
& 01010101	01010101	^ 01010101	~ 01010101
01000001	01111101	00111100	10101010

■ 布尔代数的所有性质都可运用 All of the Properties of Boolean Algebra Apply



举例：小整数集合

Example: Sets of Small Integers

■ 表示 Representation

- 宽度为 w 的比特位向量表示 $\{0, \dots, w-1\}$ 的子集 Width w bit vector represents subsets of $\{0, \dots, w-1\}$
- 当 j 属于 A 集合时, a_j 为1 $a_j = 1$ if $j \in A$

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

- 76543210

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

- 76543210

■ 运算 Operations

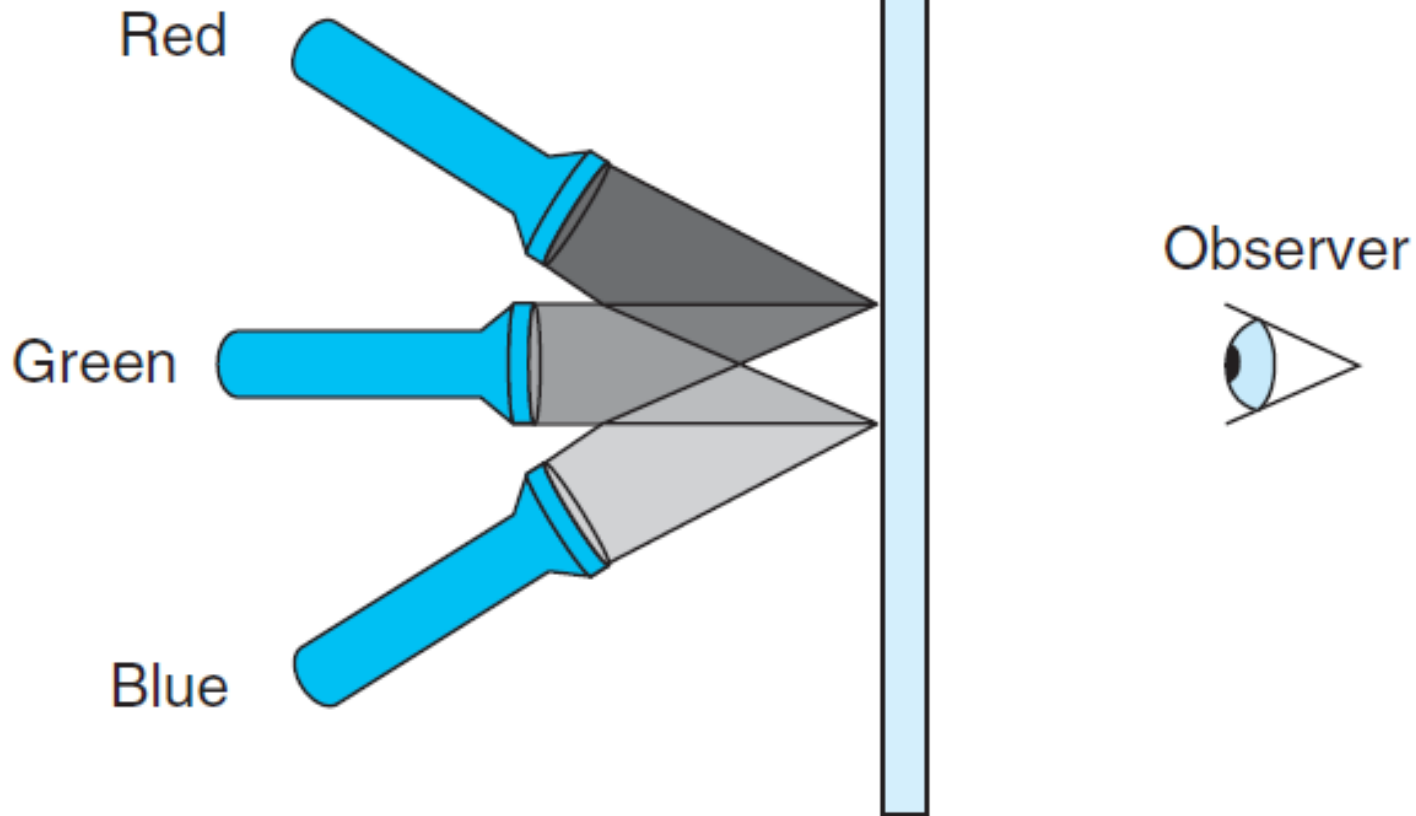
- | | | |
|------------------------------|----------|------------------------|
| ■ & 交 Intersection | 01000001 | $\{0, 6\}$ |
| ■ 并 Union | 01111101 | $\{0, 2, 3, 4, 5, 6\}$ |
| ■ ^ 对称差 Symmetric difference | 00111100 | $\{2, 3, 4, 5\}$ 异或 |
| ■ ~ 补 Complement | 10101010 | $\{1, 3, 5, 7\}$ |



三基色模型 RGB Color Model

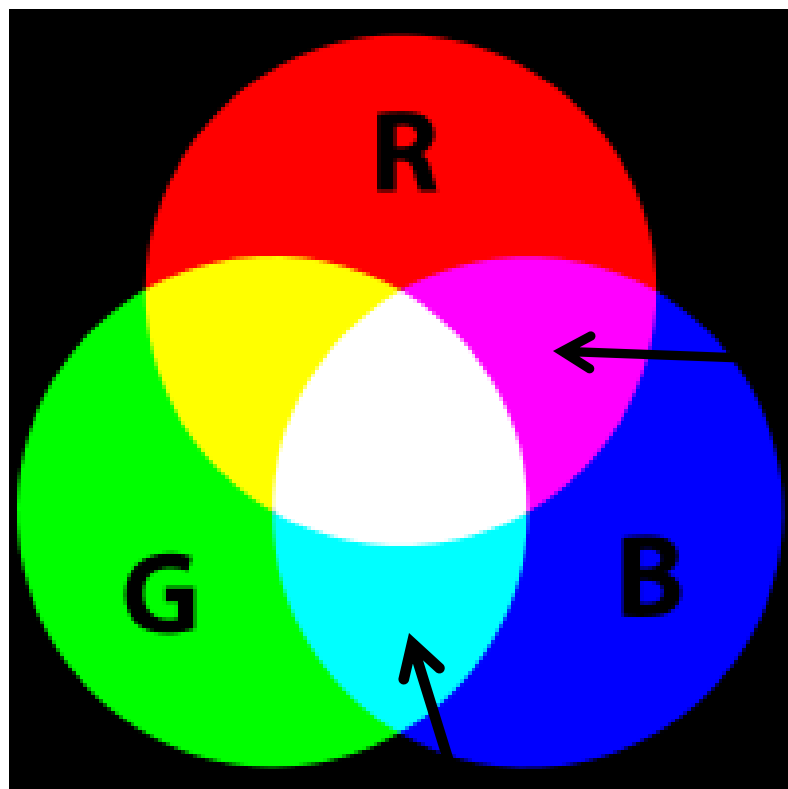
Light sources

Glass screen





三基色模型 RGB Color Model



Cyan 青色

品红 Magenta

<i>R</i>	<i>G</i>	<i>B</i>	Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White



C语言中的比特级运算

Bit-Level Operations in C

- **C中可用运算 Operations &, |, ~, ^ Available in C**
 - 运用到任何整数类数据类型 Apply to any “integral” data type
 - long, int, short, char, unsigned
 - 参数视为比特位向量 View arguments as bit vectors
 - 参数运用到每个比特位 Arguments applied bit-wise
- **举例（字符数据类型） Examples (Char data type)**
 - $\sim 0x41 \rightarrow$
 - $\sim 0x00 \rightarrow$
 - $0x69 \& 0x55 \rightarrow$
 - $0x69 | 0x55 \rightarrow$



C语言中的比特级运算

Bit-Level Operations in C

- **C中可用运算 Operations &, |, ~, ^ Available in C**
 - 运用到任何整数类数据类型 Apply to any “integral” data type
 - long, int, short, char, unsigned
 - 参数视为比特位向量 View arguments as bit vectors
 - 参数运用到每个比特位 Arguments applied bit-wise
- **举例（字符数据类型） Examples (Char data type)**
 - $\sim 0x41 \rightarrow 0xBE$
 - $\sim 0100\ 0001_2 \rightarrow 1011\ 1110_2$
 - $\sim 0x00 \rightarrow 0xFF$
 - $\sim 0000\ 0000_2 \rightarrow 1111\ 1111_2$
 - $0x69 \& 0x55 \rightarrow 0x41$
 - $0110\ 1001_2 \& 0101\ 0101_2 \rightarrow 0100\ 0001_2$
 - $0x69 | 0x55 \rightarrow 0x7D$
 - $0110\ 1001_2 | 0101\ 0101_2 \rightarrow 0111\ 1101_2$

用异或进行很酷的操作

Cool Stuff with Xor



- **比特位级异或是一种加法** Bitwise Xor is form of addition
- **具有额外的性质，即每个值都是其自身的加法逆元** With extra property that every value is its own additive inverse
 - $A \wedge A = 0$

用异或进行很酷的操作

Cool Stuff with Xor



```
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

用异或进行很酷的操作

Cool Stuff with Xor



```
1 void reverse_array(int a[], int cnt) {  
2     int first, last;  
3     for (first = 0, last = cnt-1;  
4         first <= last;  
5         first++,last--)  
6         inplace_swap(&a[first], &a[last]);  
7 }
```



掩码操作 Mask Operations

■ 位模式 Bit pattern

- 0xFF
 - 最低8个有效位为1 Having 1s for the least significant eight bits
 - 指明一个字的最低字节 Indicates the lower-order byte of a word

■ 掩码操作 Mask Operation

- $X = 0x89ABCDEF$
- $X \& 0xFF = ?$

■ 位模式~0 Bit Pattern ~0

- 为何不是0xFFFFFFFF? Why not 0xFFFFFFFF?
- 无论字长是多少 No matter how word size



掩码操作 Mask Operations

- 写C表达式使其对任何字长 w 大于等于8的数都能适用 Write C expressions that work for any word size $w \geq 8$
- For $x = 0x87654321$, with $w = 32$
- x 的最低有效字节, 其它位都设为0 The least significant byte of x , with all other bits set to 0
 - $[0x00000021]$
 - $x \& 0xFF$



掩码操作 Mask Operations

- **x的最低有效字节不变，所有其它位变反（取补）** All but the least significant byte of complemented, with the least significant byte left unchanged
 - $[0x789ABC21] \quad x \wedge \sim 0xFF$
- **x的最低有效字节设置为全1，所有其它字节保持不变** The least significant byte set to all 1s, and all other bytes of x left unchanged.
 - $[0x876543FF]. \quad x | 0xFF$

位设置和位清除 Bis & Bic



- **设置结果z为x并进行修改** Set result z to x and modify it
 - X data, m mask
- **$z = \text{bis}(\text{int } x, \text{int } m)$ (位设置 bit set) = $x \mid m$**
 - 在m为1的每个位置上, 将结果z的对应位设置为1 Set result z to 1 at each bit position where m is 1
- **$z = \text{bic}(\text{int } x, \text{int } m)$ (位清除 bit clear) = $x \& \sim m$**
 - 在m为1的每个位置上, 将结果z的对应位设置为0 set result z to 0 at each bit position where m is 1
- **DEC公司的VAX机** The Digital Equipment VAX
- **使用位设置和位清除来实现** Use bis and bic to implement
 - $\text{Or}(\text{int } x, \text{int } y)$ $\text{bis}(x, y)$
 - $\text{Xor}(\text{int } x, \text{int } y)$ $\text{bis}(\text{bic}(x, y), \text{bic}(y, x))$
 - $(x \& \sim y) \mid (y \& \sim x)$

对比：C语言中的逻辑运算

Contrast: Logic Operations in C



■ 对比特位级的操作符 Contrast to Bit-Level Operators

■ 逻辑运算 Logic Operations: &&, ||, !

- 视0为假 View 0 as “False”
- 任何非零视为真 Anything nonzero is true
- 总是返回0或1 Always return 0 or 1
- 提前终止 Early termination

■ 举例（字符数据类型） Example

- !0x41 → 0x00
- !0x00 → 0x01
- !!0x41 → 0x01
- 0x69 && 0x55 → 0x01
- 0x69 || 0x55 → 0x01
- p && *p (avoids null pointer access)避免访问空指针

注意&& vs. &(以及|| vs. |)...
超级常见的C编程错误!

Watch out for && vs. & (and || vs. |)...
Super common C programming pitfall!



逻辑运算的快捷方式

Short Cut in Logical Operations

■ `a && 5/a`

- 如果a为零，不会计算5/a If a is zero, the evaluation of 5/a is stopped
- 避免了被零除 avoid division by zero

■ `p && *p`

- 不会导致间接引用空指针 Never cause the dereferencing of a null pointer

■ 仅使用位级和逻辑操作 Using only bit-level and logical operations

- 实现`x==y` Implement `x == y`
- x和y相等时返回1，否则返回0 it returns 1 when x and y are equal, and 0 otherwise
- $!(x \wedge y)$



移位运算 Shift Operations

■ 左移 Left Shift: $x \ll y$

- 位向量x左移y位 Shift bit-vector x left y positions
 - 丢弃左边多余比特 Throw away extra bits on left
 - 右边填0 Fill with 0's on right

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

■ 右移 Right Shift: $x \gg y$

- 位向量x右移y位 Shift bit-vector x right y positions
 - 丢弃右边多余的比特 Throw away extra bits on right
- 逻辑移位 Logical shift
 - 左边填0 Fill with 0's on left
- 算术移位 Arithmetic shift
 - 左边复制最高有效位 Replicate most significant bit on left

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

■ 未定义行为 Undefined Behavior

- 移位量小于零或大于等于字长 Shift amount < 0



C语言中的移位操作

Shift Operations in C

■ 会发生什么情况? What happens ?

- `int lval = 0xFEDCBA98 << 32;`
- `int aval = 0xFEDCBA98 >> 36;`
- `unsigned uval = 0xFEDCBA98u >> 40;`

■ 可能的情况 It may be $k \bmod w$

- `lval` `0xFEDCBA98` (0)
- `aval` `0xFFEDCBA9` (4)
- `uval` `0x00FEDCBA` (8)

■ 要小心 Be careful about

- `1<<2 + 3<<4` means `1<<(2 + 3)<<4`
- 512
- Not 52



位计数 bitCount

- 返回字中1的个数 Returns number of 1's a in word
- 例如: Examples: `bitCount(5) = 2`, `bitCount(7) = 3`
- 合法的运算符: Legal ops: `! ~ & ^ | + << >>`
- 最大运算符: 40 Max ops: 40



求和4位一组共8组

Sum 8 groups of 4 bits each

```
int bitCount(int x) {  
    int m1 = 0x11 | (0x11 << 8);  
    int mask = m1 | (m1 << 16);  
    int s = x & mask;  
    s += x>>1 & mask;  
    s += x>>2 & mask;  
    s += x>>3 & mask;  
}
```



将和组合在一起 Combine the sums

```
/* Now combine high and low order sums */
```

```
s = s + (s >> 16);
```

```
/* Low order 16 bits now consists of 4 sums.
```

```
Split into two groups and sum */
```

```
mask = 0xF | (0xF << 8);
```

```
s = (s & mask) + ((s >> 4) & mask);
```

```
return (s + (s>>8)) & 0x3F;
```

```
}
```

议题: 比特、字节和整数



Today : Bits, Bytes, and Integers

- **用比特表示信息** Representing information as bits
- **比特级操作** Bit-level manipulations
- **整数** Integers
 - **无符号数和有符号数表示** Representation: unsigned and signed
 - 转换和强制类型转换 Conversion, casting
 - 扩展和截断 Expanding, truncating
 - 加、补码非、乘和移位 Addition, negation, multiplication, shifting
 - 小结 Summary
- **内存中的表示、指针和字符串** Representations in memory, pointers, strings
- **小结** Summary

编码整数 Encoding Integers



无符号 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 = 15213;  
short int y = -15213;
```

符号位
Sign
Bit

- **C语言并不强制使用二进制补码** C does not mandate using two's complement
 - 然而大部分机器一般会用二进制补码进行运算, 我们也如此假设
But, most machines do, and we will assume so
- **C语言中short为2字节长** C short 2 bytes long

	十进制 Decimal	十六进制 Hex	二进制 Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

编码整数 Encoding Integers



无符号 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 = 15213;  
short int y = -15213;
```

符号位
Sign
Bit

■ 符号位 Sign Bit

- 对于补码最高有效位是符号位 For 2's complement, most significant bit indicates sign
 - 0表示非负 0 for nonnegative
 - 1表示负 1 for negative

补码：简单示例



Two-complement: Simple Example

	-16	8	4	2	1	
10 =	0	1	0	1	0	$8+2 = 10$

	-16	8	4	2	1	
-10 =	1	0	1	1	0	$-16+4+2 = -10$

补码编码举例 (续)



Two-complement Encoding Example (Cont.)

$x =$ 15213: 00111011 01101101
 $y =$ -15213: 11000100 10010011

Weight 权	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
Sum	15213		-15213	

无符号数表示 Unsigned Representation



- **二进制（物理上） Binary (physical)**
 - 位向量 Bit vector $[x_{w-1}, x_{w-2}, x_{w-3}, \dots, x_0]$
- **二进制到无符号数（逻辑上） Binary to Unsigned (logical)**

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



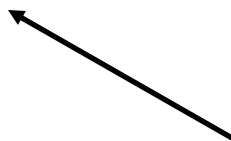
补码 Two's Complement

- 二进制（物理上） Binary (physical)
 - 位向量 Bit vector $[x_{w-1}, x_{w-2}, x_{w-3}, \dots, x_0]$
- 二进制到有符号数（逻辑上） Binary to Signed (logical)

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

- 补码 2's complement

符号位
Sign
Bit





从补码到二进制

From Two's Complement to Binary

- 如果是非负数 If nonnegative
 - 不需要变化 Nothing changes
- 如果是负数, 其二进制表示为 If negative, its binary representation is
 - $1x_{w-2}\dots x_1x_0$
 - 其值为 x Its value is x
 - 假设其绝对值为 $y = -x$ Assume its absolute value is $y = -x$
- y 的二进制表示为 The binary representation of y is
 - $0y_{w-2}\dots y_1y_0$

从补码到二进制(x是负数)



From Two's Complement to Binary

$$X = -2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i = -Y = -\sum_{i=0}^{w-2} y_i 2^i$$

$$\sum_{i=0}^{w-2} x_i 2^i = 2^{w-1} - \sum_{i=0}^{w-2} y_i 2^i = \sum_{i=0}^{w-2} (1 - y_i) 2^i + 1$$

$$2^{w-1} = \sum_{i=0}^{w-2} 2^i + 1 \quad x_{w-1} = 1 \quad y_{w-1} = 0$$



补码 Two's Complement

$$\sum_{i=0}^{w-1} x_i 2^i = \sum_{i=0}^{w-1} (1 - y_i) 2^i + 1$$

■ 这个公式意味着什么呢？ What does it mean?

- 计算x的绝对值成w位的二进制数 Computing the negation of x into binary with w-bits
- 对结果变反（求补） Complementing the result
- 加一 Adding 1

从十进制数变换成补码

From a Number to Two's Complement



■ -5

- 5
- 0101 (5的二进制表示 binary for 5)
- 1010 (变反 after complement)
- 1011 (加一 add 1)

补码编码示例



Two's Complement Encoding Examples

二进制/十六进制表示 Binary/Hexadecimal Representation for
-12345

二进制 Binary: 0011 0000 0011 1001 (12345)

十六进制 Hex: 3 0 3 9

二进制 Binary: 1100 1111 1100 0110 (变反后 after
complement)

十六进制 Hex: C F C 6

二进制 Binary: 1100 1111 1100 0111 (加一 add 1)

十六进制 Hex: C F C 7



数值范围 Numeric Ranges

■ 无符号值 Unsigned Values

- $UMin = 0$
000...0
- $UMax = 2^w - 1$
111...1

■ 补码值 Two's Complement Values

- $TMin = -2^{w-1}$
100...0
- $TMax = 2^{w-1} - 1$
011...1

■ 其它值 Other Values

- 负一 (-1) Minus 1
111...1

Values for $W = 16$

	Decimal	Hex	Binary
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



不同字长的取值范围

Values for Different Word Sizes

	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 Observations

- $|TMin| = TMax + 1$
 - 非对称范围 Asymmetric range
- $UMax = 2 * TMax + 1$
- Question: $abs(TMin)$?

■ C语言编程 C Programming

- `#include <limits.h>`
- 声明常量 Declares constants, e.g.,
 - `ULONG_MAX`
 - `LONG_MAX`
 - `LONG_MIN`
- 值随平台而定 Values platform specific

无符号和有符号数的数值

Unsigned & Signed Numeric Values



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

■ 等同的 Equivalence

- 非负值同样的编码 Same encodings for nonnegative values

■ 惟一性 Uniqueness

- 每个比特模式表示惟一的整数值 Every bit pattern represents unique integer value
- 每个可表示的整数有惟一的比特位编码 Each representable integer has unique bit encoding

■ ⇒ 能够逆映射 Can Invert Mappings

- $U2B(x) = B2U^{-1}(x)$
 - 无符号整数比特模式 Bit pattern for unsigned integer
- $T2B(x) = B2T^{-1}(x)$
 - 补码整数比特模式 Bit pattern for

其它有符号数表示



Alternative representations of signed numbers

- **反码：与补码相同，除了最高位权值为 $-(2^{w-1} - 1)$ 而不是 -2^{w-1}**
Ones' complement : The same as two's complement, except that the most significant bit has weight $-(2^{w-1} - 1)$ rather than -2^{w-1}

$$\text{B2O}_w(\vec{x}) = -x_{w-1}(2^{w-1} - 1) + \sum_{i=0}^{w-2} x_i 2^i$$

- 过去有使用反码的机器，现在都用补码 use in the past, Now use two's complement
- +0 : 00000000 -0 : 11111111

- **原码：最高有效位是符号位，决定剩余位是负权还是正权**
Sign magnitude : The most significant bit is a sign bit that determines whether the remaining bits should be given negative or positive weight

$$\text{B2S}_w(\vec{x}) = (-1)^{x_{w-1}} \cdot \left(\sum_{i=0}^{w-2} x_i 2^i \right)$$

- 在浮点数表示中还在使用 Used with floating-point numbers
- +0 : 00000000 -0 : 10000000



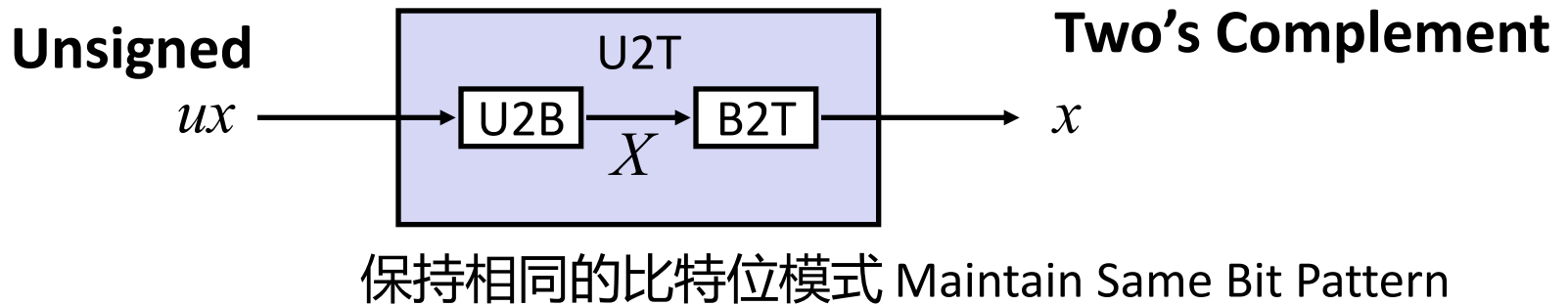
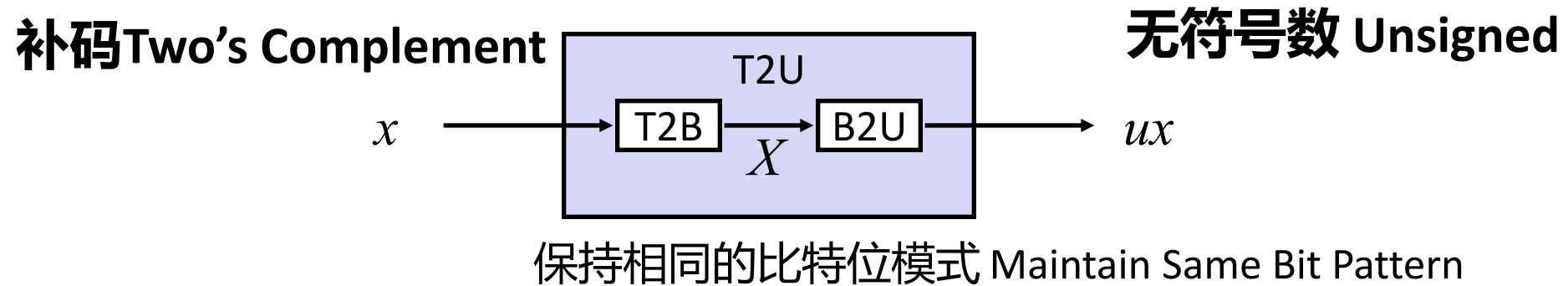
议题: 比特、字节和整数

Today : Bits, Bytes, and Integers

- 用比特表示信息 Representing information as bits
- 比特级操作 Bit-level manipulations
- **整数 Integers**
 - 无符号数和有符号数表示 Representation: unsigned and signed
 - **转换和强制类型转换 Conversion, casting**
 - 扩展和截断 Expanding, truncating
 - 加、补码非、乘和移位 Addition, negation, multiplication, shifting
 - 小结 Summary
- 内存中的表示、指针和字符串 Representations in memory, pointers, strings

有符号数和无符号数之间进行映射

Mapping Between Signed & Unsigned

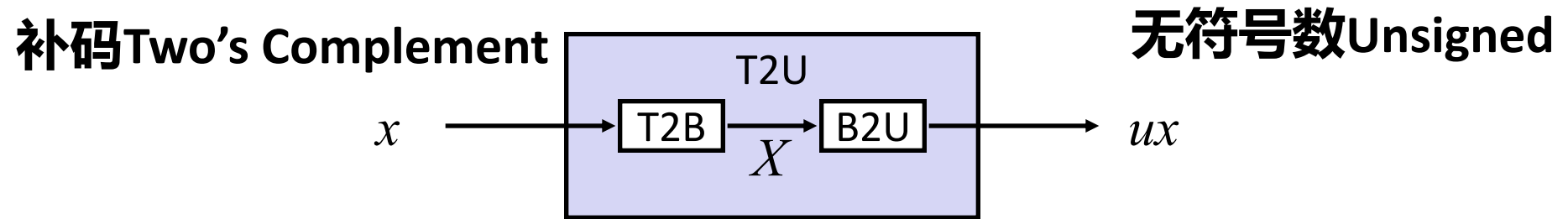


- 无符号数和补码之间进行映射 Mappings between unsigned and two's complement numbers:

保持位表示并重新解释 Keep bit representations and reinterpret

有符号数和无符号数之间进行映射

Mapping Between Signed & Unsigned



保持相同的比特位模式 Maintain Same Bit Pattern

$$\begin{array}{r}
 \begin{array}{c} w-1 \qquad \qquad \qquad 0 \\
 ux \quad \boxed{+|+|+} \cdot \cdot \cdot \boxed{+|+|+} \\
 -x \quad \boxed{-|+|+} \cdot \cdot \cdot \boxed{+|+|+} \end{array} \\
 \hline
 +2^{w-1} - -2^{w-1} = 2 * 2^{w-1} = 2^w
 \end{array}$$

$$ux = \begin{cases} x & x \geq 0 \\ x + 2^w & x < 0 \end{cases}$$

有符号和无符号数之间映射

Mapping Signed \leftrightarrow Unsigned



Bits	Signed		Unsigned
0000	0	\longrightarrow T2U \longrightarrow \longleftarrow U2T \longleftarrow	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

有符号和无符号数之间映射

Mapping Signed \leftrightarrow Unsigned



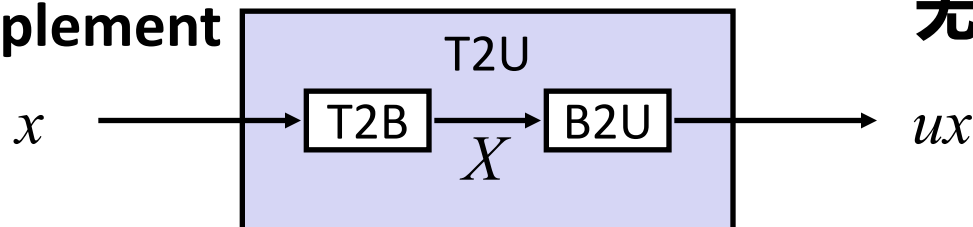
Bits	Signed		Unsigned
0000	0	\longleftrightarrow =	0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8	\longleftrightarrow +/- 16	8
1001	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

有符号数和无符号数之间的关系

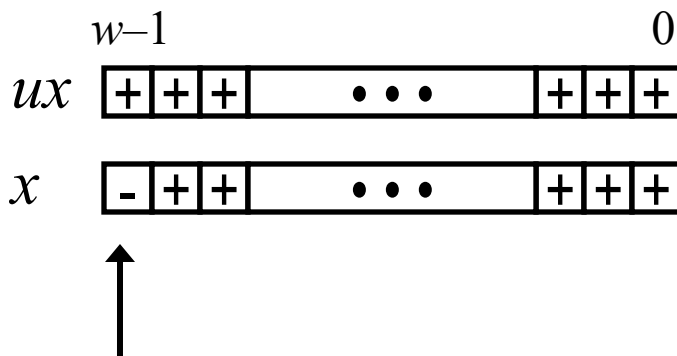
Relation between Signed & Unsigned



补码 Two's Complement



无符号 Unsigned



大的负权 Large negative weight

变成 becomes

大的正权 Large positive weight

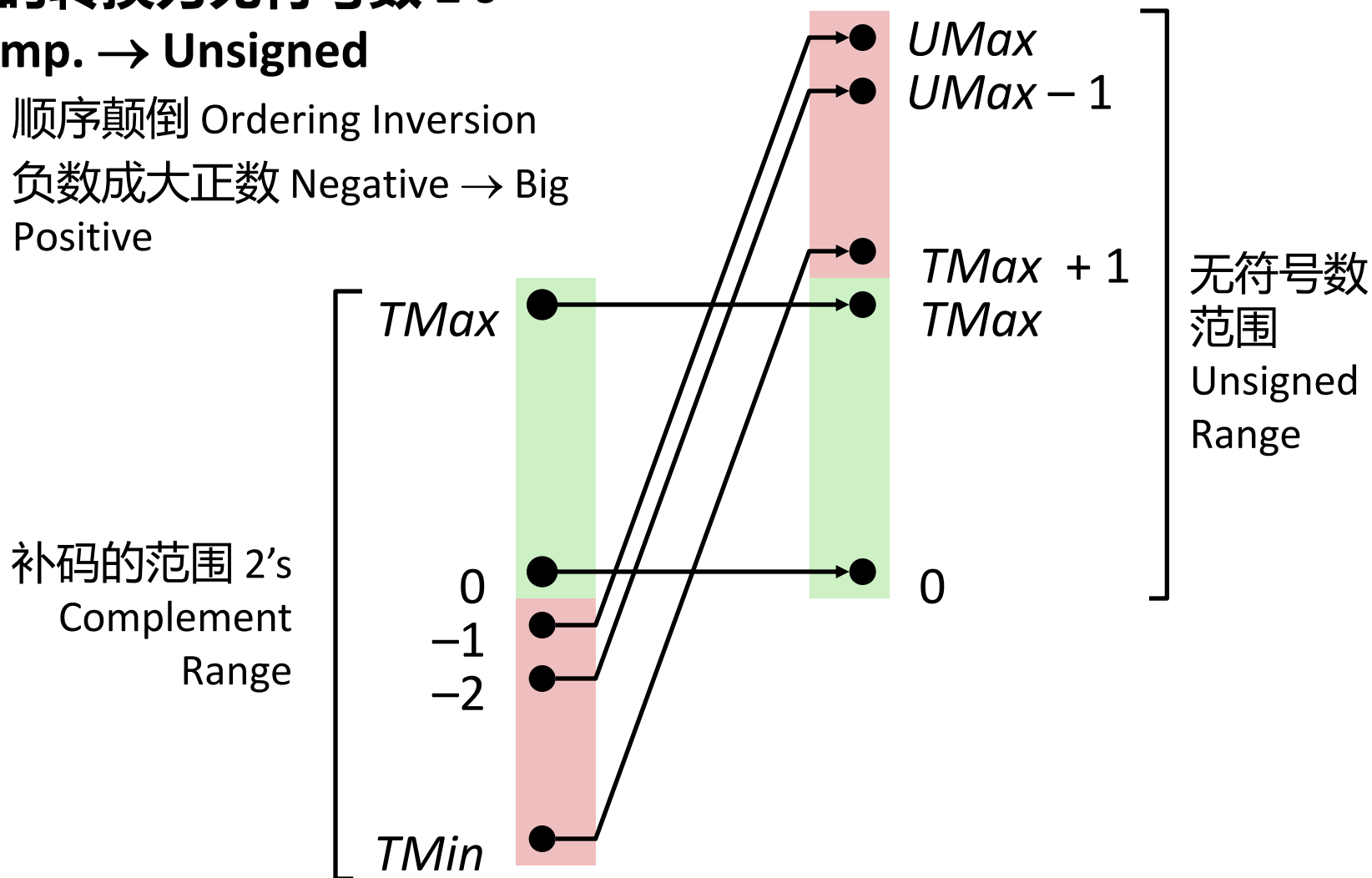
转换可视化 Conversion Visualized



■ 补码转换为无符号数 2's

Comp. \rightarrow Unsigned

- 顺序颠倒 Ordering Inversion
- 负数成大正数 Negative \rightarrow Big Positive





C语言中的整数数据类型

Integral data type in C

- **有符号类型 Signed type (整型数 for integer numbers)**
 - char, short [int], int, long [int]
- **无符号类型 Unsigned type (非负数 for nonnegative numbers)**
 - unsigned char, unsigned short [int], unsigned [int], unsigned long [int]
- **Java没有无符号数据类型 Java has no unsigned data type**
 - 使用字节代替字符 Using byte to replace the char

C语言整数数据类型典型取值范围-32位

Typical Ranges for C integral data types 32



C语言声明 C declaration	典型32位 Typical 32-bit	
	最小 minimum	最大 maximum
char	-128	127
unsigned char	0	255
short [int]	-32,768	32,767
unsigned short	0	65,535
int	-2,147,483,648	2,147,483,647
unsigned [int]	0	4,294,967,295
long [int]	-2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,295
int32_t	-2,147,483,648	2,147,483,647
uint32_t	0	4,294,967,295
int64_t	-9,223,372,036,854,775,800	9,223,372,036,854,775,800
uint64_t	0	18,446,744,073,709,551,615

C语言整数数据类型典型取值范围-64位

Typical Ranges for C integral data types 64



C语言声明 C declaration	典型64位 Typical 64-bit	
	最小 minimum	最大 maximum
char	-128	127
unsigned char	0	255
short [int]	-32,768	32,767
unsigned short	0	65,535
int	-2,147,483,648	2,147,483,647
unsigned [int]	0	4,294,967,295
long [int]	-9,223,372,036,854,775,800	9,223,372,036,854,775,800
unsigned long	0	18,446,744,073,709,551,615
int32_t	-2,147,483,648	2,147,483,647
uint32_t	0	4,294,967,295
int64_t	-9,223,372,036,854,775,800	9,223,372,036,854,775,800
uint64_t	0	18,446,744,073,709,551,615

C语言整数数据类型确保取值范围

Guaranteed Ranges for C integral data types



C语言声明 C declaration	典型32位 Typical 32-bit	
	最小 minimum	最大 maximum
char	-127	127
unsigned char	0	255
short [int]	-32,767	32,767
unsigned short	0	65,535
int	-32,767	32,767
unsigned [int]	0	65,535
long [int]	-2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,295
int32_t	-2,147,483,648	2,147,483,647
uint32_t	0	4,294,967,295
int64_t	-9,223,372,036,854,775,800	9,223,372,036,854,775,800
uint64_t	0	18,446,744,073,709,551,615

C语言有/无符号数之间强制类型转换



Casting among Signed and Unsigned in C

■ C语言允许一种类型的变量解释为另一种数据类型

C Allows a variable of one type to be interpreted as other data type

- 类型转换（隐式） Type conversion (implicitly)
- 强制类型转换（显式） Type casting (explicitly)

C语言中的有符号数和无符号数

Signed vs. Unsigned in C



■ 常量 Constants

- 默认为有符号整数 By default are considered to be signed integers
- 有U做后缀表示无符号数 Unsigned if have “U” as suffix
0U, 4294967259U

■ 强制类型转换 Casting

- 显示强制类型转换有/无符号数等同于U2T/T2U Explicit casting between signed & unsigned same as U2T and T2U

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

- 隐式强制类型转换通过赋值和过程调用也会发生 Implicit casting also occurs via assignments and procedure calls

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

强制从有符号数转换成无符号数

Casting from Signed to Unsigned



```
short int          x = 12345;
unsigned short int ux = (unsigned short) x;
short int          y = -12345;
unsigned short int uy = (unsigned short) y;
```

■ 结果值 Resulting Value

- 位表示没有变化 No change in bit representation
- 非负数值没有变化 Nonnegative values unchanged
 - $ux = 12345$
- 负值变成大的正值 Negative values change into (large) positive values
 - $uy = 53191$



Weight	12,345		−12,345		53,191	
	Bit	Value	Bit	Value	Bit	Value
1	1	1	1	1	1	1
2	0	0	1	2	1	2
4	0	0	1	4	1	4
8	1	8	0	0	0	0
16	1	16	0	0	0	0
32	1	32	0	0	0	0
64	0	0	1	64	1	64
128	0	0	1	128	1	128
256	0	0	1	256	1	256
512	0	0	1	512	1	512
1,024	0	0	1	1,024	1	1,024
2,048	0	0	1	2,048	1	2,048
4,096	1	4096	0	0	0	0
8,192	1	8192	0	0	0	0
16,384	0	0	1	16,384	1	16,384
±32,768	0	0	1	−32,768	1	32,768
Total		12,345		−12,345		53,191

强制类型转换奇怪之处 Casting Surprises



■ 表达式求值 Expression Evaluation

- 如果单一表达式中混合了无符号数和有符号数 If there is a mix of unsigned and signed in single expression,
有符号值隐含强制转换成无符号数 *signed values implicitly cast to unsigned*
- 包括比较运算 Including comparison operations $<$, $>$, $==$, $<=$, $>=$
- Examples for $W = 32$: **TMIN = -2,147,483,648**, **TMAX = 2,147,483,647**

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



小结 Summary

有/无符号数强制转换：基本规则

Casting Signed \leftrightarrow Unsigned: Basic Rules

- 比特位模式保持不变 Bit pattern is maintained
- 但是需要重新解释 But reinterpreted
- 可能有不期望的效果：加或减 2^w Can have unexpected effects: adding or subtracting 2^w
- 表达式包含有符号和无符号整数时 Expression containing signed and unsigned int
 - 有符号数强制类型转换为无符号数 `int` is cast to `unsigned`!!



议题: 比特、字节和整数

Today : Bits, Bytes, and Integers

- **用比特表示信息** Representing information as bits
- **比特级操作** Bit-level manipulations
- **整数** Integers
 - 无符号数和有符号数表示 Representation: unsigned and signed
 - 转换和强制类型转换 Conversion, casting
 - **扩展和截断** Expanding, truncating
 - 加、补码非、乘和移位 Addition, negation, multiplication, shifting
 - 小结 Summary
- **内存中的表示、指针和字符串** Representations in memory, pointers, strings



从短到长的转换 From short to long

```
short int x = 12345;
```

```
int ix = (int) x;
```

```
short int y = -12345;
```

```
int iy = (int) y;
```

- 我们需要扩展数据长度 We need to expand the data size
- 无符号类型之间强制类型转换正常 Casting among unsigned types is normal
- 有符号类型之间强制类型转换需要技巧 Casting among signed types is trick



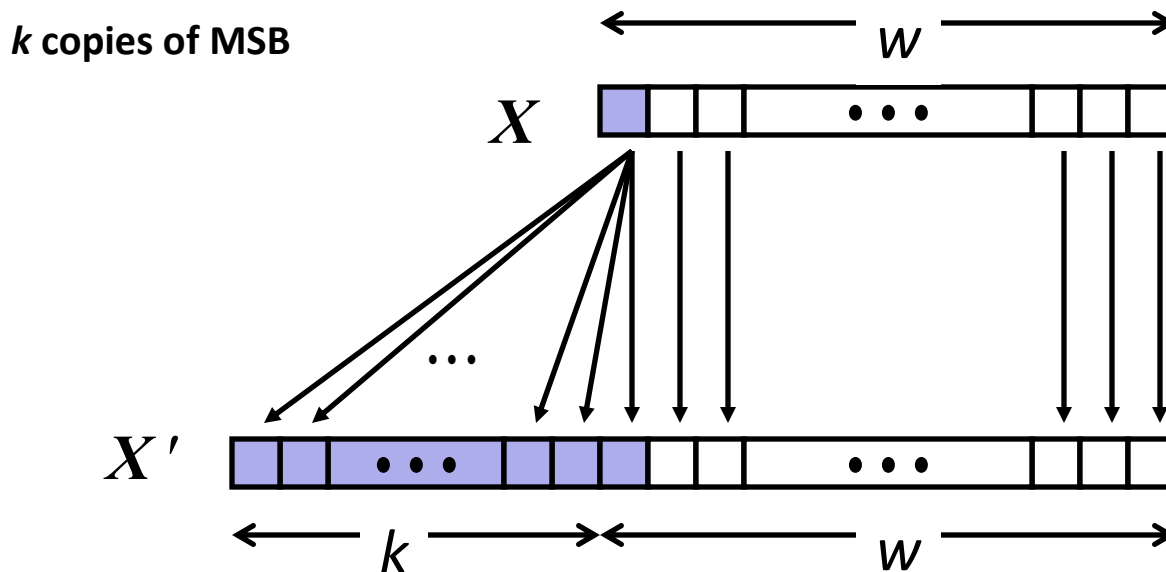
符号扩展 Sign Extension

■ 任务 Task:

- 给定 w 位的带符号整数 x Given w -bit signed integer x
- 转换成数值相同的 $w+k$ 位整数 Convert it to $w+k$ -bit integer with same value

■ 规则 Rule: MSB-最高有效位

- 把符号位复制 k 位 Make k copies of sign bit:
- $X' = \underbrace{x_{w-1}, \dots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \dots, x_0$





符号位扩展：简单示例

Sign Extension: Simple Example

正数 Positive number

		-16	8	4	2	1
10 =		0	1	0	1	0
	-32	16	8	4	2	1
10 =	0	0	1	0	1	0

Diagram illustrating sign extension for a positive number (10). The number 10 is represented in a 5-bit binary format (01010). The sign bit (0) is extended to the 6th bit (-32) in the 6-bit format (001010), as indicated by the red arrows.

负数 Negative number

		-16	8	4	2	1
-10 =		1	0	1	1	0
	-32	16	8	4	2	1
-10 =	1	1	0	1	1	0

Diagram illustrating sign extension for a negative number (-10). The number -10 is represented in a 5-bit binary format (10110). The sign bit (1) is extended to the 6th bit (-32) in the 6-bit format (110110), as indicated by the red arrows.

更大型符号位扩展举例

Larger Sign Extension Example



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

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

- 从小整数数据类型转换到大整数数据类型 Converting from smaller to larger integer data type
- C语言自动执行符号扩展 C automatically performs sign extension



从短到长的扩展 From short to long

```
short int  sx = 12345; : 0x0309
```

```
int  x = (int)  sx;      : 0x00000309
```

```
short int  sy =-12345; : 0xcfc7
```

```
int  y = (int)  sy;      : 0xffffcfc7
```

```
usinged ux = sy;        : 0xffffcfc7
```



从短到长的扩展 From short to long

```
int fun1(unsigned word) {  
    return (int) ((word << 24) >> 24);  
}  
int fun2(unsigned word) {  
    return ((int) word << 24) >> 24;  
}
```

w	fun1(w)	fun2(w)
0x00000076	00000076	00000076
0x87654321	00000021	00000021
0x000000C9	000000C9	FFFFFFC9
0xEDCBA987	00000087	FFFFFF87

描述每个函数执行时字中的有用计算 Describe in words the useful computation each of these functions performs.



从长到短转换 From long to short

```
int          x  = 53191;  
short int    sx = x;  
int          y  = -12345;  
Short int    sy = y;
```

- 我们需要截断数据大小 We need to truncate the data size
- 强制类型转换从长到短需要技巧 Casting from long to short is trick

截断 Truncation

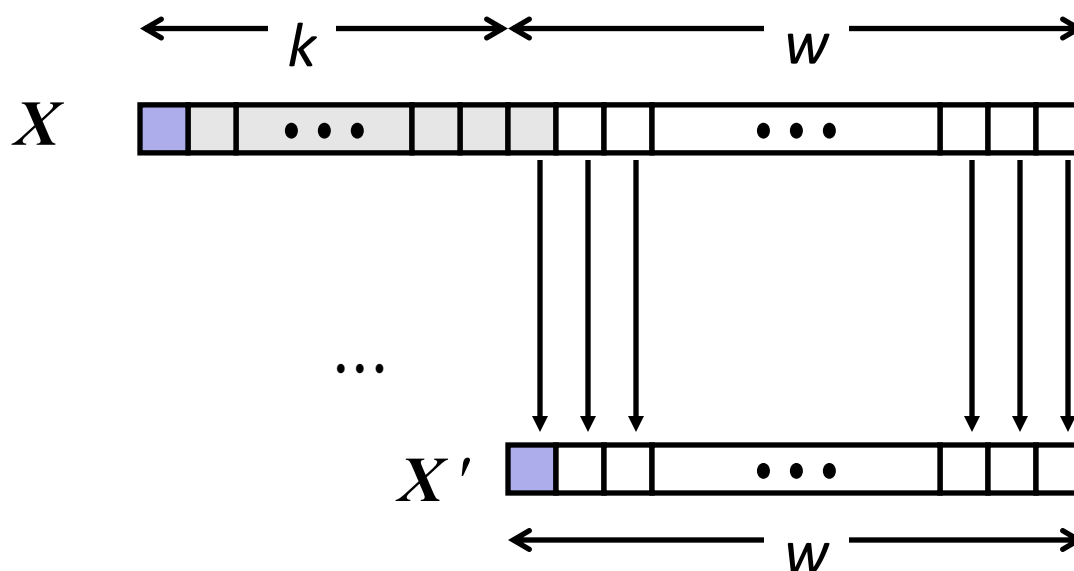


■ 任务 Task:

- 给定 $k+w$ 位有符号或无符号整数 X Given $k+w$ -bit signed or unsigned integer X
- 转换成 w 位整数 X' ，对于“足够小的” X 具有同样的值 Convert it to w -bit integer X' with same value for “small enough” X

■ 规则 Rule:

- 丢弃头 k 位 Drop top k bits:
- $X' = x_{w-1}, x_{w-2}, \dots, x_0$



截断：简单示例 Truncation: Simple Example



符号位没变 No sign change

	-16	8	4	2	1
2 =	0	0	0	1	0

	-8	4	2	1
2 =	0	0	1	0

$$2 \bmod 16 = 2$$

	-16	8	4	2	1
-6 =	1	1	0	1	0

	-8	4	2	1
-6 =	1	0	1	0

$$-6 \bmod 16 = 26U \bmod 16 = 10U = -6$$

符号位变化 Sign change

	-16	8	4	2	1
10 =	0	1	0	1	0

	-8	4	2	1
-6 =	1	0	1	0

$$10 \bmod 16 = 10U \bmod 16 = 10U = -6$$

	-16	8	4	2	1
-10 =	1	0	1	1	0

	-8	4	2	1
6 =	0	1	1	0

$$-10 \bmod 16 = 22U \bmod 16 = 6U = 6$$



截断数值 Truncating Numbers

■ Unsigned Truncating

$$\begin{aligned} B2U_w([x_w, x_{w-1}, \dots, x_0]) \bmod 2^k \\ = B2U_k([x_k, x_{k-1}, \dots, x_0]) \end{aligned}$$

• Signed Truncating

$$\begin{aligned} B2T_k([x_k, x_{k-1}, \dots, x_0]) \\ = B2T_k(B2U_w([x_w, x_{w-1}, \dots, x_0]) \bmod 2^k) \end{aligned}$$



小结 Summary:

扩展和截断：基本规则

Expanding, Truncating: Basic Rules

- **扩展（例如short扩展成int） Expanding (e.g., short int to int)**
 - 无符号数：添加零 Unsigned: zeros added
 - 有符号数：符号位扩展 Signed: sign extension
 - 都还会产生期望的结果 Both yield expected result
- **截断（例如无符号int截断成无符号short） Truncating (e.g., unsigned to unsigned short)**
 - 无/有符号数：比特位截断 Unsigned/signed: bits are truncated
 - 结果重新解释 Result reinterpreted
 - 无符号数：模取余运算 Unsigned: mod operation
 - 有符号数：类似模取余 Signed: similar to mod
 - 对于小的数值还会产生期望的行为 For small numbers yields expected behavior