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 颜珂

原作者：
Randal E. Bryant and David R. O’Hallaron

上次讲授课程小结
Summary From Last Lecture

- 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

以前讲授内容

  - 加法、补码非、乘法和移位 Addition, negation, multiplication, shifting

今天

- 内存中表示、指针、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

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

符号位
Sign Bit

补码示例 Two's Complement Examples (w = 5)

	-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

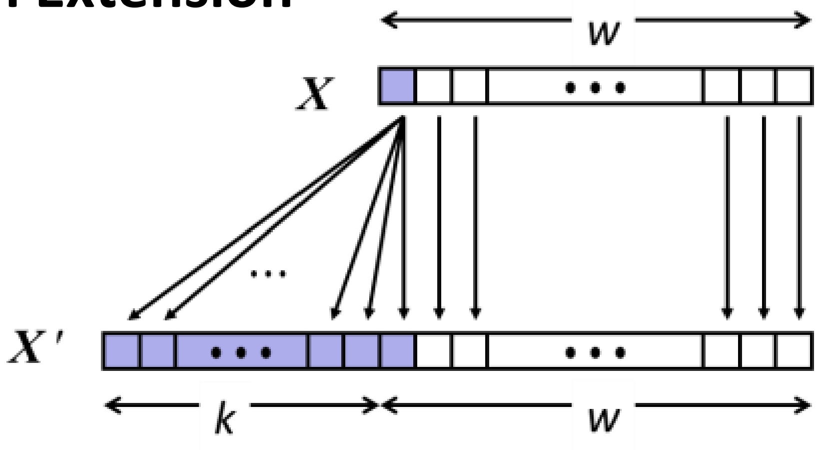
无符号数和有符号数的值
Unsigned & Signed Numeric Values

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

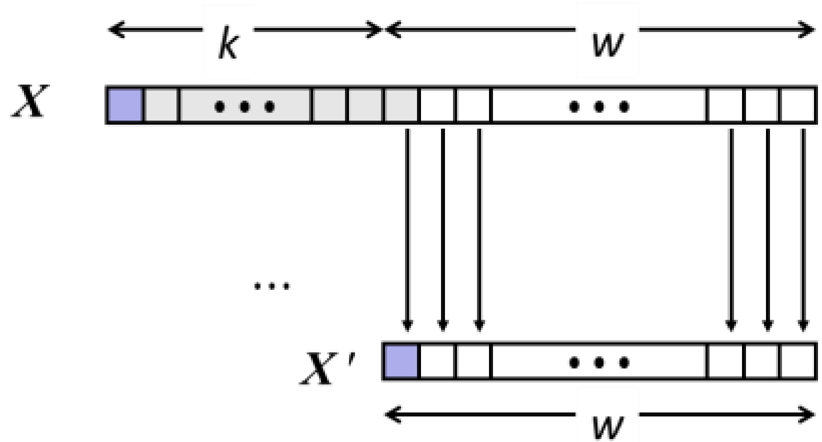
- **等同的 Equivalence**
  - 非负值的编码相同 Same encodings for nonnegative values
- **惟一的 Uniqueness**
  - 每个位模式表示惟一的整数值 Every bit pattern represents unique integer value
  - 每个可表示的整数有惟一的位编码 Each representable integer has unique bit encoding
- **包含有符号和无符号int型的表达式：有符号数强制转换为无符号数 Expression containing signed and unsigned int: `int` is cast to `unsigned`**

符号位扩展和截断
Sign Extension and Truncation

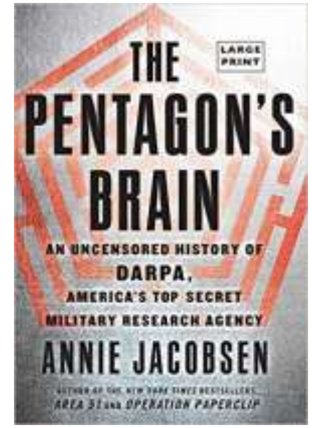
- **符号位扩展 Sign Extension**



- **截断 Truncation**



- **正如我们所知，误解整数可能导致世界末日！ Misunderstanding integers can lead to the end of the world as we know it!**
- 图勒（卡纳克），格陵兰 Thule (Qaanaaq), Greenland
- 美国国防部“Site J”弹道导弹预警系统 US DoD “Site J” Ballistic Missile Early Warning System (BMEWS)
- 10/5/60: world nearly ends世界接近末日
- 导弹雷达回波 Missile radar echo: 1/8s
- BMEWS reports: 75s echo(!)
- 报告了1000多个目标 1000s of objects reported
- NORAD alert level 5:警报5级
  - 立即来袭的核攻击 Immediate incoming nuclear attack!!!!



- 赫鲁晓夫在纽约市10/5/60（不寻常的攻击时间）**Kruschev was in NYC 10/5/60 (weird time to attack)**
  - 有人在卡纳克说“为什么不去外面检查一下？” someone in Qaanaaq said “why not go check outside?”
- “导弹”实际上是在挪威上空升起的月亮 **“Missiles” were actually THE MOON RISING OVER NORWAY**
- 预期最大距离：3000 英里；月球距离：0.25M 英里！ **Expected max distance: 3000 mi; Moon distance: .25M miles!**
- .25M 英里 % sizeof(distance) = 2200mi。**.25M miles % sizeof(distance) = 2200mi.**
- 距离的溢出差点造成核末日 **Overflow of distance nearly caused nuclear apocalypse!!**

代码安全示例 Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}

/* maxlen is negative  */
```

- 在FreeBSD的getpeernname实现中发现类似的代码 Similar to code found in FreeBSD’s implementation of getpeername
- 有很多聪明人尝试发现程序中的漏洞 There are legions of smart people trying to find vulnerabilities in programs

典型的使用方法 Typical Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

恶意使用Malicious Usage

```
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
```

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

议题: 比特、字节和整数
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**
- 内存中的表示、指针和字符串 Representations in memory, pointers, strings
- 小结 Summary

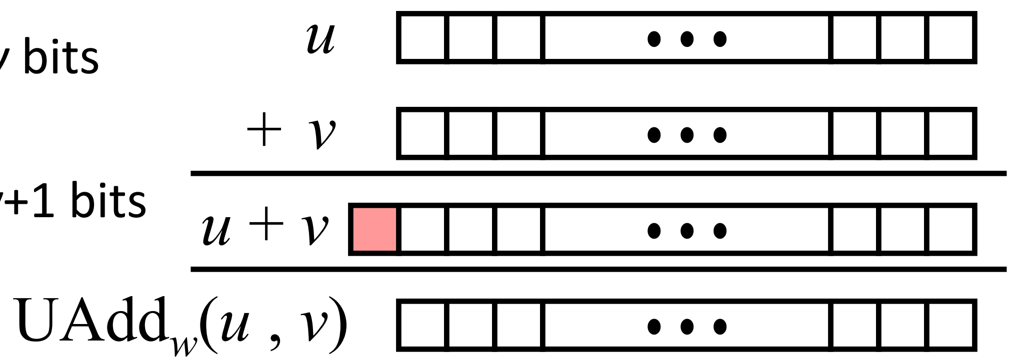
无符号数加法
Unsigned Addition

操作数w位 Operands: *w* bits

真和w+1位 True Sum: *w*+1 bits

丢弃进位后和为w位

Discard Carry: *w* bits



- **标准加法功能 Standard Addition Function**
  - 忽略进位输出 Ignores carry output
- **实现取模运算 Implements Modular Arithmetic**

$s \;=\; \mathrm{UAdd}_w(u\ ,\ v) \;=\; u + v \mod 2^w$

无符号字符

```
unsigned char     1110 1001      E9       233
                + 1101 0101    + D5     + 213
                ___________    ____     _____

                ___________    ____     _____
```

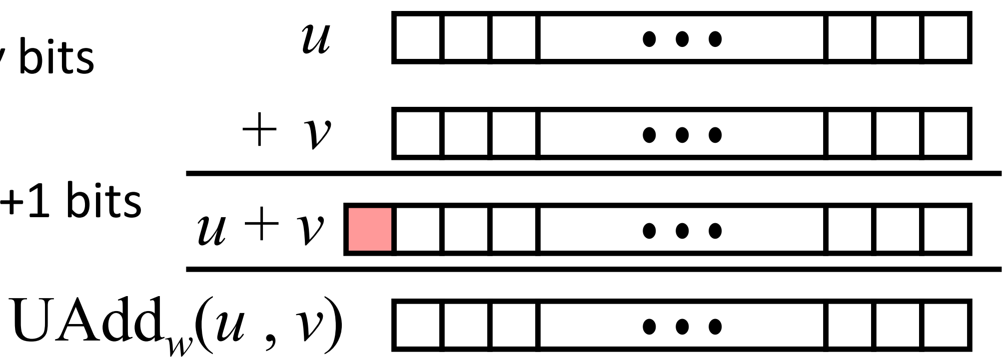
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

无符号数加法
Unsigned Addition

操作数w位 Operands: *w* bits

真和w+1位 True Sum: *w*+1 bits

丢弃进位后和为w位
Discard Carry: *w* bits



- **标准加法功能 Standard Addition Function**
  - 忽略进位输出 Ignores carry output
- **实现取模运算 Implements Modular Arithmetic**

$$s \;=\; \mathrm{UAdd}_w(u\,,v) \;=\; u + v \mod 2^w$$

无符号字符
`unsigned char`

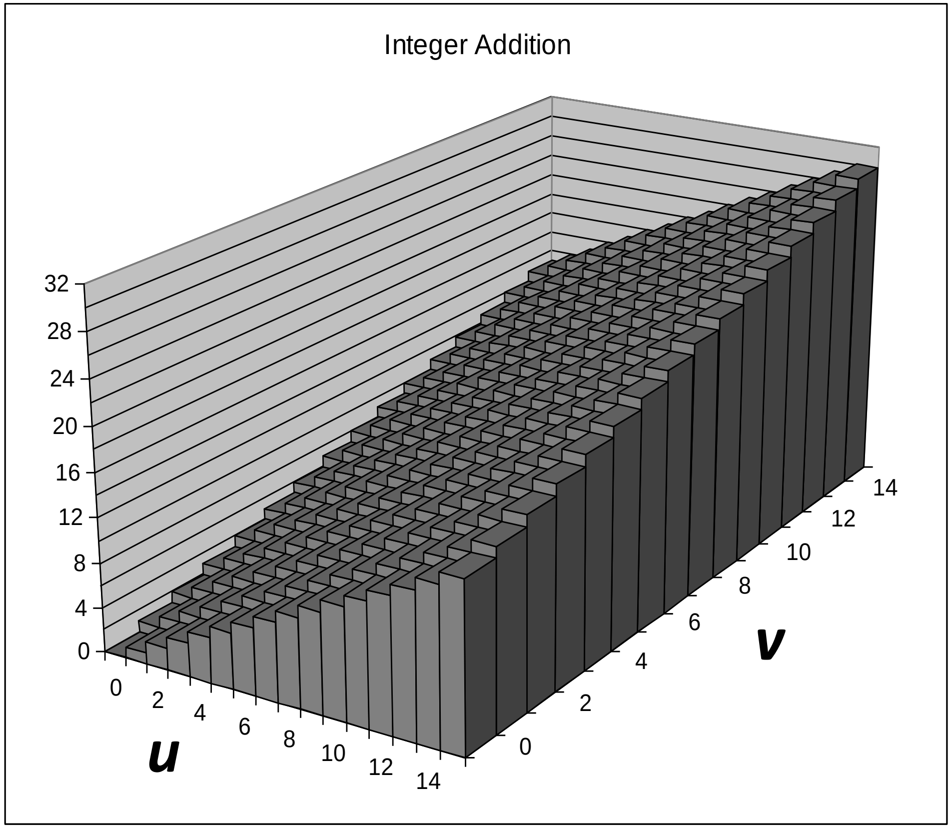
```
    1110 1001      E9      233
+   1101 0101    + D5    + 213
  1 1011 1110     1BE      446
    1011 1110      BE      190
```

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

可视化（数学上）整数加法
Visualizing (Mathematical) Integer Addition

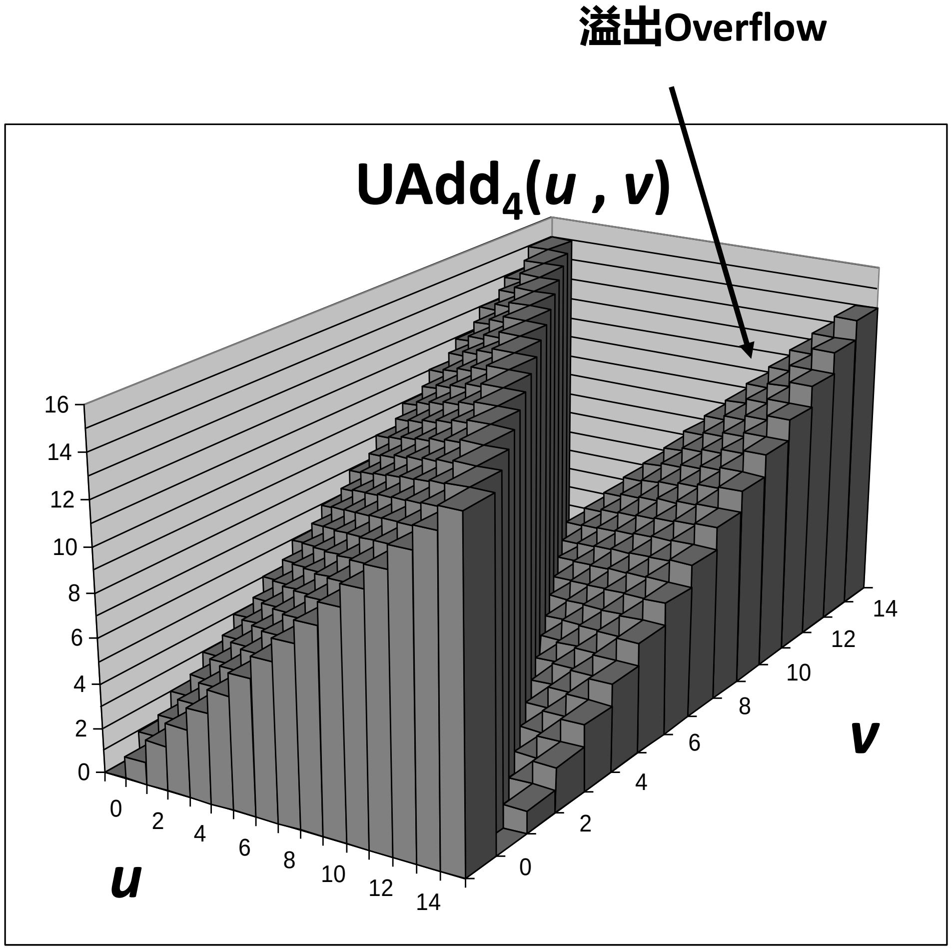
- **整数加法 Integer Addition**
  - 4位整数u和v 4-bit integers *u*, *v*
  - 计算真正的和 Compute true sum $\mathrm{Add}_4(u\ ,\ v)$
  - 值随着u和v线性增加 Values increase linearly with *u* and *v*
  - 形成有坡度的表面 Forms planar surface

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

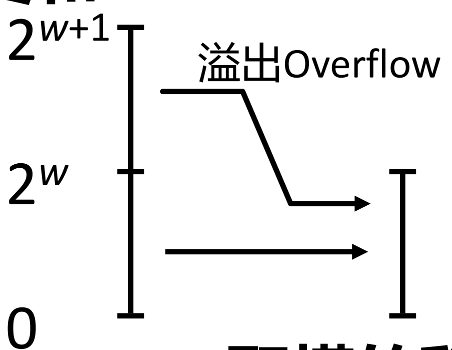


可视化无符号数加法
Visualizing Unsigned Addition

- **绕回 Wraps Around**
  - 如果真正的和大于等于 2^w If true sum ≥ 2^w
  - 最多一次 At most once



真和 True Sum



取模的和
Modular Sum

数学上性质 Mathematical Properties

- **模数加法形成阿贝尔群 Modular Addition Forms an *Abelian Group***
  - **封闭的加法 Closed** under addition

    $0 \leq \mathrm{UAdd}_w(u\,,v) \leq 2^w - 1$
  - **交换性 Commutative**

    $\mathrm{UAdd}_w(u\,,v) = \mathrm{UAdd}_w(v\,,u)$
  - **结合性 Associative**

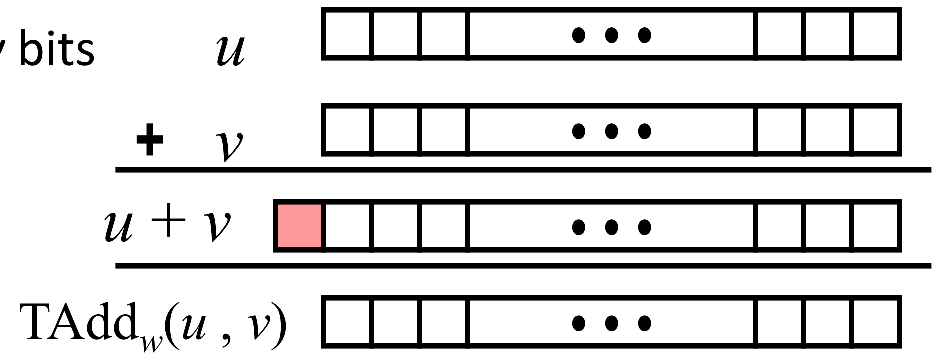
    $\mathrm{UAdd}_w(t, \mathrm{UAdd}_w(u\,,v)) = \mathrm{UAdd}_w(\mathrm{UAdd}_w(t, u\,), v)$
  - **0是加性恒等(单位元) 0** is additive identity

    $\mathrm{UAdd}_w(u\,,0) = u$
  - 每个元素都有加法逆元 Every element has additive **inverse**
    - Let $\mathrm{UComp}_w(u\,) = 2^w - u$

      $\mathrm{UAdd}_w(u\,, \mathrm{UComp}_w(u\,)) = 0$

补码加法 Two's Complement Addition

操作数w位 Operands: *w* bits



真正和w+1位
True Sum: *w*+1 bits

丢弃进位
Discard Carry: *w* bits

- **有符号和无符号数加法有同样的比特位级行为 TAdd and UAdd have Identical Bit-Level Behavior**
  - C语言中带符号和无符号数加法 Signed vs. unsigned addition in C:

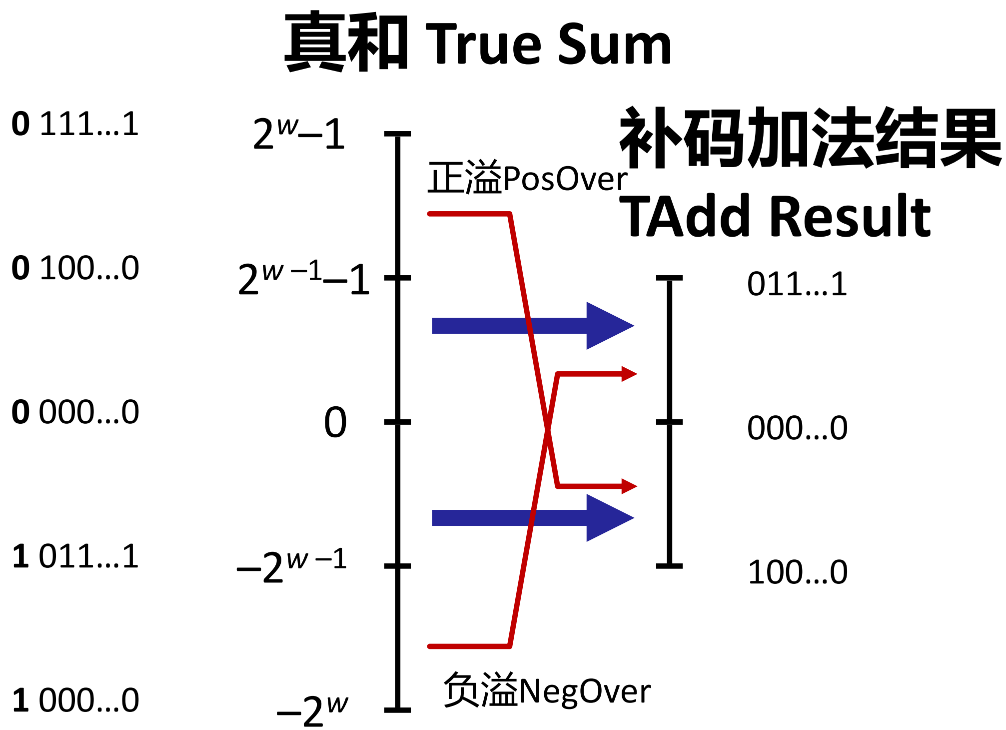
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - 结果s和t相同 Will give `s == t`

1110 1001	E9	-23
+ 1101 0101	+ D5	+ -43
1 1011 1110	1BE	-66
1011 1110	BE	-66

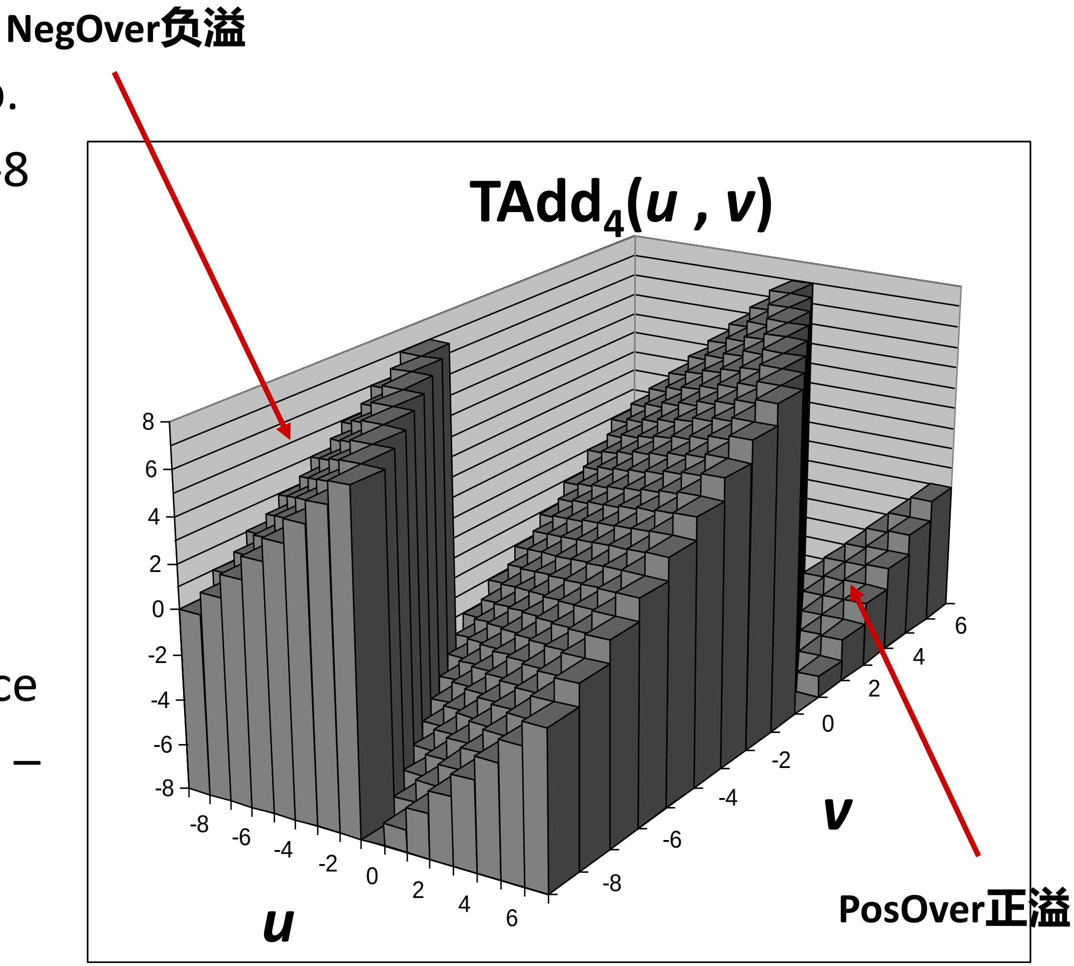
有符号数加法溢出 TAdd Overflow

- **功能 Functionality**
  - 真和需要w+1位 True sum requires *w*+1 bits
  - 丢弃最高有效位 Drop off MSB
  - 剩余位作为补码整数对待 Treat remaining bits as 2’s comp. integer



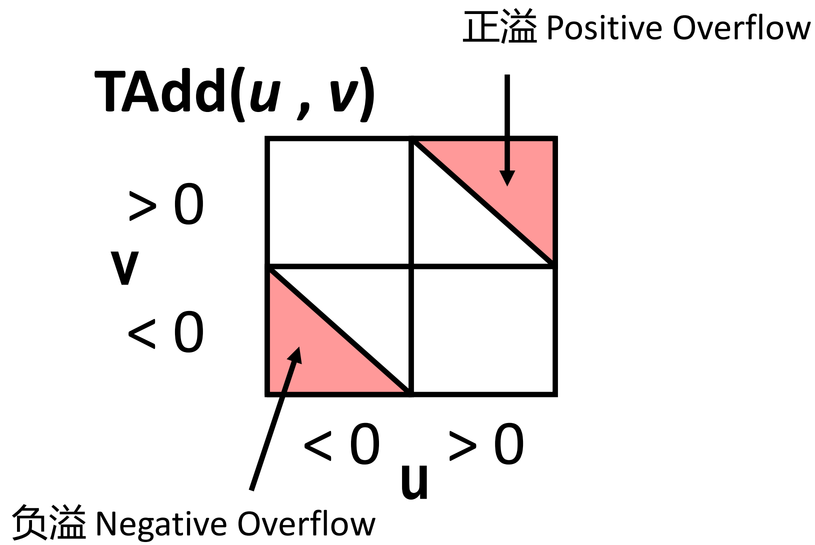
可视化补码加法
Visualizing 2's Complement Addition

- 值 Values
  - 4位补码 4-bit two's comp.
  - 值域-8到+7 Range from -8 to +7
- 绕回 Wraps Around
  - 如果和大于等于2^{w-1} If sum $\geq 2^{w-1}$
    - 变成负数 Becomes negative
    - 最多一次 At most once
  - 如果和小于-2^{w-1} If sum $< -2^{w-1}$
    - 变成正数Becomes positive
    - 最多一次At most once



有符号数加法特征
Characterizing TAdd

- **功能 Functionality**
  - 真正的和需要w+1位 True sum requires w+1 bits
  - 丢弃最高有效位 Drop off MSB
  - 剩余位看成补码整数 Treat remaining bits as 2's comp. integer



$$TAdd_w(u,v) = \begin{cases} u+v+2^w & u+v < TMin_w \quad \textbf{(负溢 NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^w & TMax_w < u+v \quad \textbf{(正溢 PosOver)} \end{cases}$$

TAdd数学上的性质
Mathematical Properties of TAdd

- **与无符号数的Uadd是同构群 Isomorphic Group to unsigneds with UAdd**
  - $\text{TAdd}_w(u, v) = \text{U2T}(\text{UAdd}_w(\text{T2U}(u), \text{T2U}(v)))$
    - 因为都有同样的比特位模式 Since both have identical bit patterns

- **TAdd下补码形成一个群 Two's Complement Under TAdd Forms a Group**
  - 封闭性、交换性、结合性、0具有加性恒等性（单位元） Closed, Commutative, Associative, 0 is additive identity
  - 每个元素都有加法逆元 Every element has additive inverse

$$TComp_w(u) = \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}$$

乘法 Multiplication

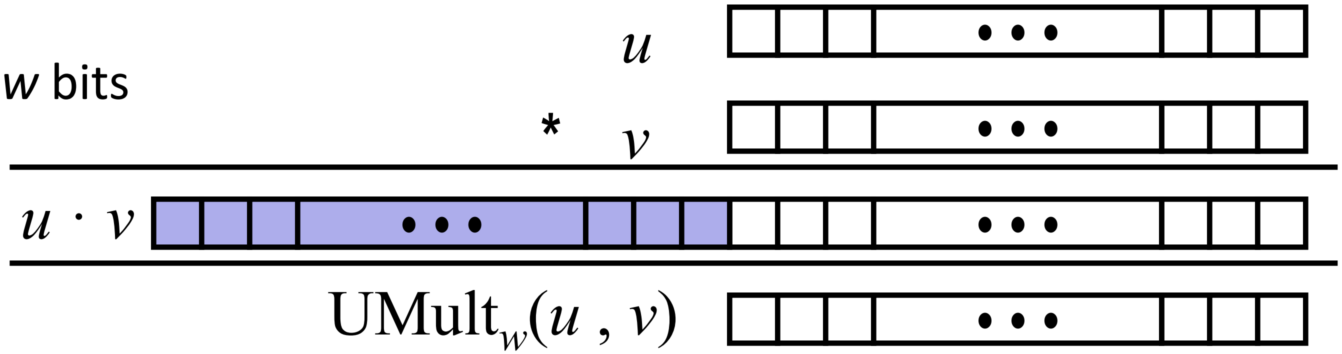
- **目标：计算w位的数x和y的乘积 Goal: Computing Product of *w*-bit numbers *x*, *y***
  - 要么是有符号的，要么是无符号的 Either signed or unsigned
- **精确的结果比w位大得多 exact results can be bigger than *w* bits**
  - 无符号数：到2w位 Unsigned: up to 2*w* bits
    - 结果范围：Result range: $0 \le x * y \le (2^{w} - 1)^{2} = 2^{2w} - 2^{w+1} + 1$
  - 补码最小（负数）：到2w-1位 Two's complement min (negative): Up to 2*w*-1 bits
    - 结果范围：Result range: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - 补码最大（正数）：到2w位，但仅限于$(TMin_w)^2$ Two's complement max (positive): Up to 2*w* bits, but only for $(TMin_w)^2$
    - 结果范围：Result range: $x * y \le (-2^{w-1})^{2} = 2^{2w-2}$
- **所以，保持精确的结果。。。So, maintaining exact results…**
  - 需要在计算每个乘积时不断扩大乘积结果表示的字长 would need to keep expanding word size with each product computed
  - 如果需要由软件完成 is done in software, if needed
    - 例如，任意精度算术软件包 e.g., by "arbitrary precision" arithmetic packages

C语言中的无符号数乘法
Unsigned Multiplication in C

操作数w位 Operands: *w* bits

真乘积2w位
True Product: 2**w* bits

丢弃w位 Discard *w* bits: *w* bits



- **标准乘法功能 Standard Multiplication Function**
  - 忽略高w位 Ignores high order *w* bits
- **实现取模运算 Implements Modular Arithmetic**

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

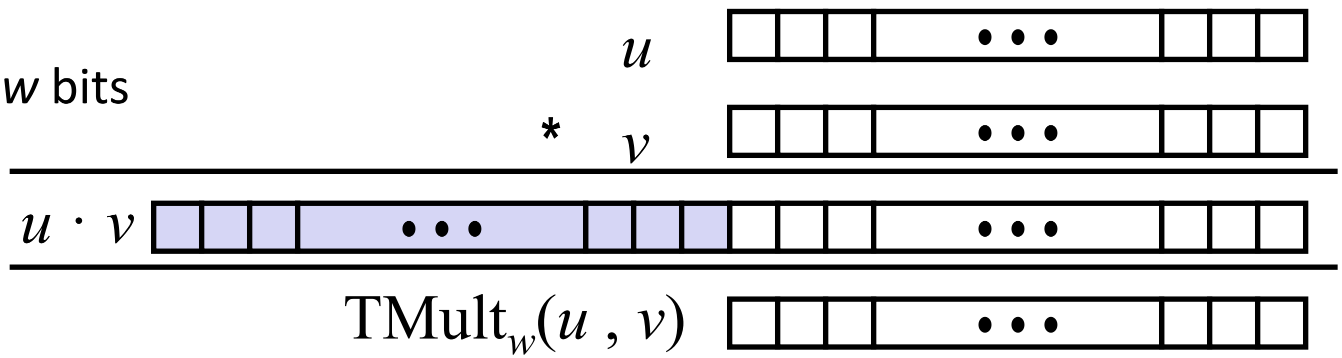
```
            1110 1001       E9        233
*           1101 0101   *   D5   *    213
 1100 0001 1101 1101      C1DD      49629
            1101 1101       DD        221
```

C语言中的有符号数乘法
Signed Multiplication in C

操作数w位 Operands: *w* bits

真乘积2w位
True Product: 2**w* bits

丢弃w位 Discard *w* bits: *w* bits



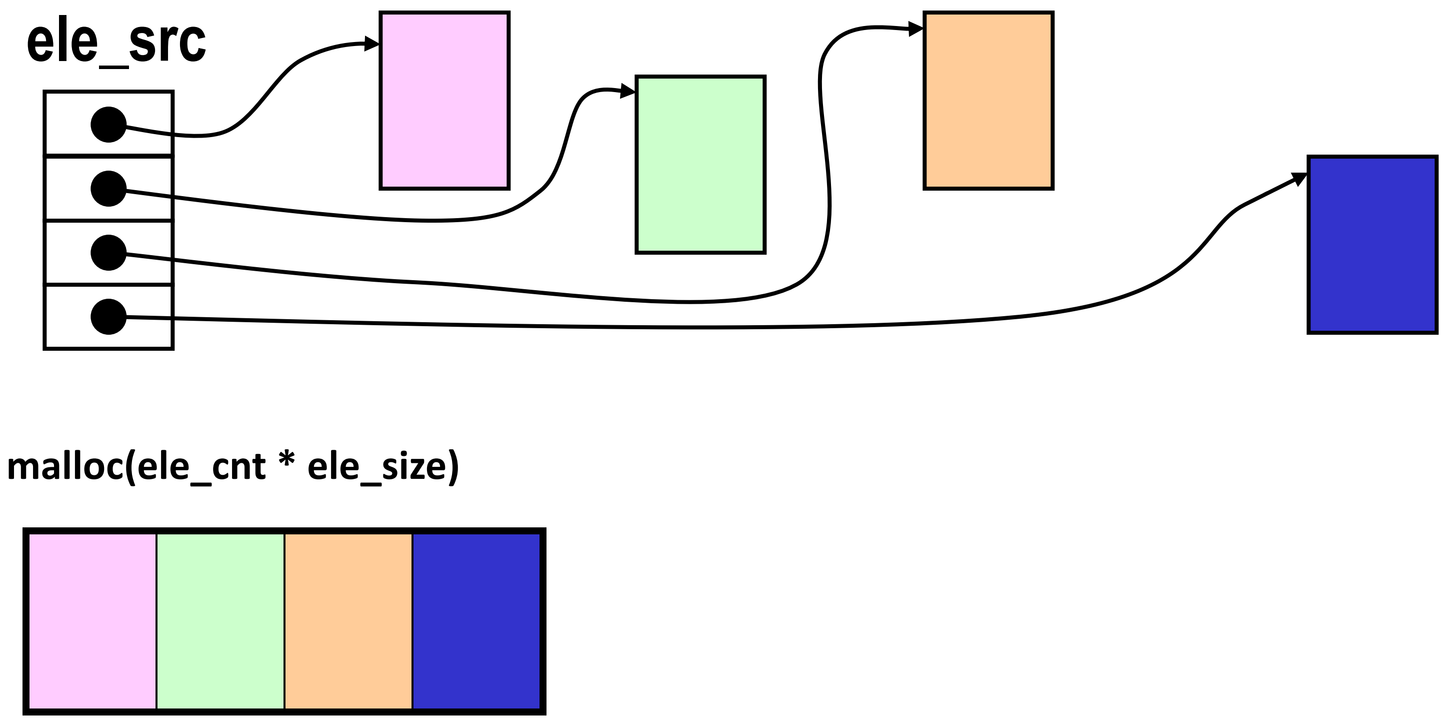
- **标准乘法功能 Standard Multiplication Function**
  - 忽略高w位 Ignores high order *w* bits
  - 有符号数和无符号数乘法有些不同 Some of which are different for signed vs. unsigned multiplication
  - 低位是相同的 Lower bits are the same

```
            1110 1001        E9        -23
*           1101 0101    *   D5   *    -43
----------------------   ------   --------
  0000 0011 1101 1101      03DD        989
----------------------   ------   --------
            1101 1101        DD        -35
```

代码安全示例2
Code Security Example #2

- **SUN的XDR库 SUN XDR library**
  - 广泛用于机器之间传输数据的库 Widely used library for transferring data between machines

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);
```



XDR代码 XDR Code

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each of ele_size bytes
     * and copy from locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL)
        /* malloc failed */
        return NULL;
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```

XDR漏洞 XDR Vulnerability

malloc(ele_cnt * ele_size)

- **如果出现下列情况会怎样 What if:**
  - `ele_cnt` = $2^{20} + 1$
  - `ele_size` = 4096 = 2^{12}
  - 分配多少空间? Allocation = ??

- **如何才能使该函数安全? How can I make this function secure?**

用移位实现2的整数次幂乘法

Power-of-2 Multiply with Shift

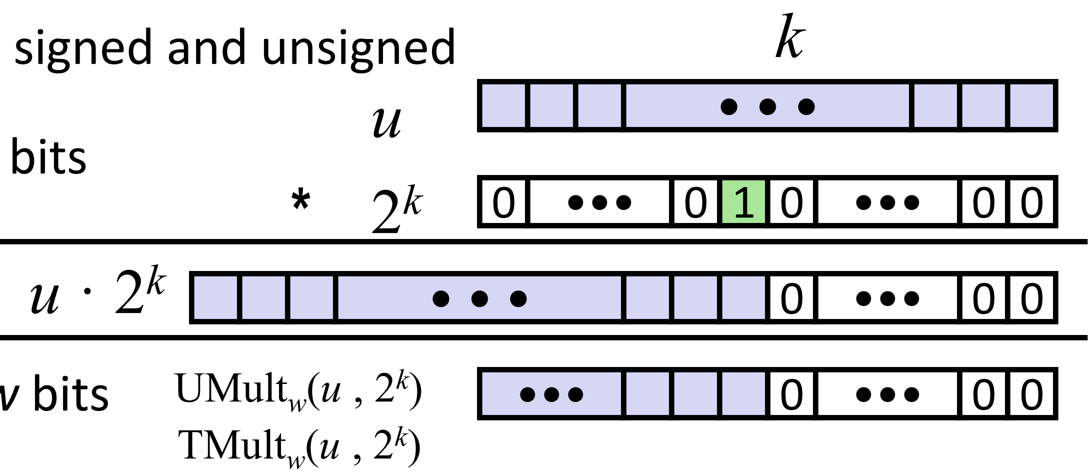
- **运算 Operation**
  - **左移k位等于乘以2^k** `u << k` gives `u * 2^k`
  - 有/无符号数均如此 Both signed and unsigned

操作数w位 Operands: *w* bits

真乘积w+k位

True Product: *w+k* bits

丢弃k位 Discard *k* bits: *w* bits

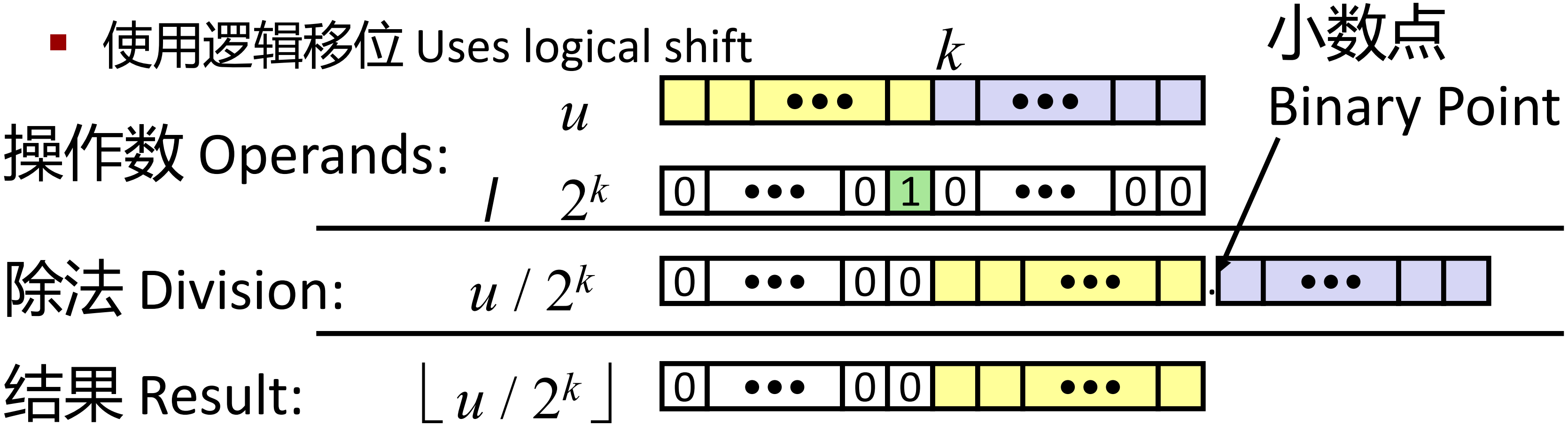


- **举例 Examples**
  - `u << 3 == u * 8`
  - `(u << 5) - (u << 3)== u * 24`
  - 大多数机器移位和加法比乘法更快 M
    faster than multiply
    - 编译器自动生成这种代码 Compi
      automatically

重要教训： Important Lesson: 信任编译器 Trust Your Compiler!

用移位实现无符号数2的整数次幂除法
Unsigned Power-of-2 Divide with Shift

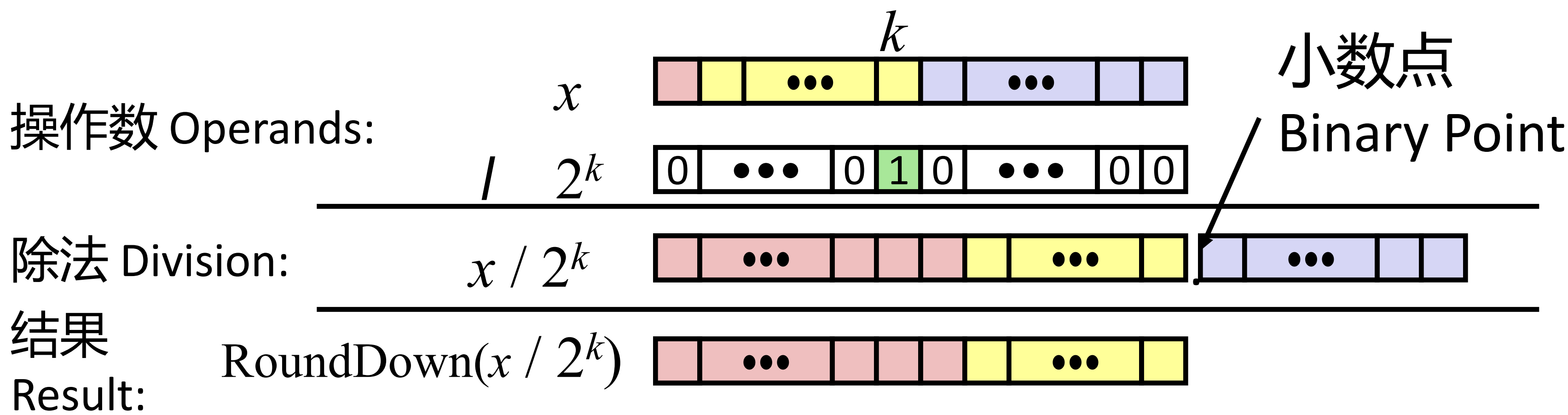
- **无符号数除以2的整数次幂的商 Quotient of Unsigned by Power of 2**
  - **右移k位等于整除2^k** `u >> k` gives $\lfloor u / 2^k \rfloor$
  - 使用逻辑移位 Uses logical shift



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

用移位实现有符号数2的整数次幂除法
Signed Power-of-2 Divide with Shift

- **有符号数除以2的整数次幂的商 Quotient of Signed by Power of 2**
  - **x右移k位等于整除2^k 向下舍入** `x >> k` gives $\lfloor$ `x / ` $2^k \rfloor$
  - 使用算术移位 Uses arithmetic shift
  - 当x<0时向错误的方向舍入 Rounds wrong direction when `x < 0`



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

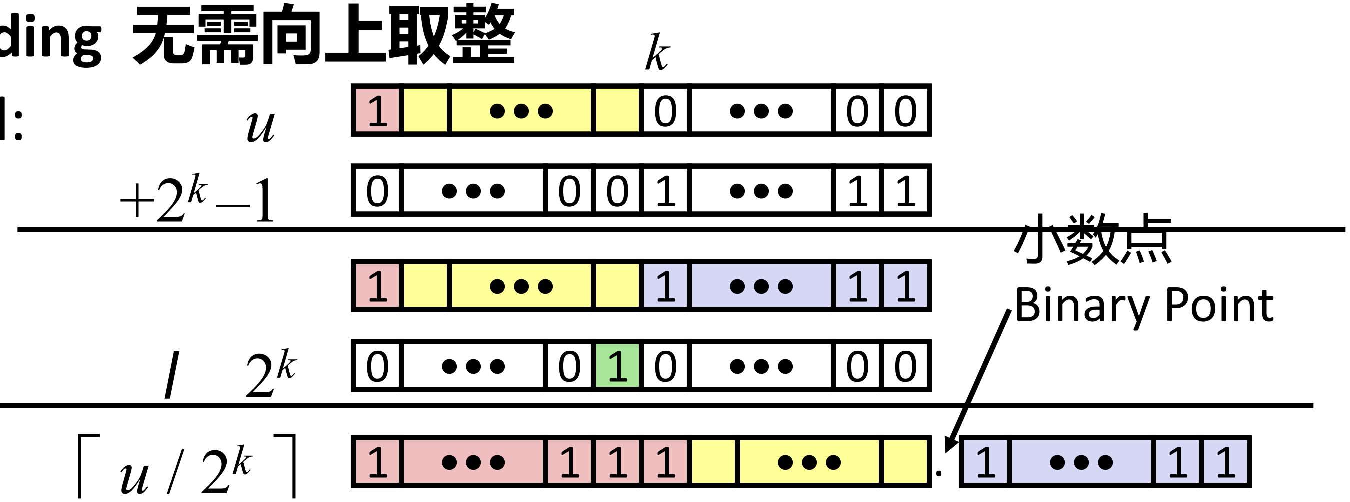
修正2的整数次幂除法
Correct Power-of-2 Divide

- **负数的2的整数次幂除法的商 Quotient of Negative Number by Power of 2**
  - 想要整除向上舍入（向0舍入）Want ⌈ **x / 2**^{k} ⌉ (Round Toward 0)
  - 计算 Compute as ⌊ **(x+2**^{k}**-1) / 2**^{k} ⌋
    - C语言中 In C: `(x + (1<<k)-1) >> k`
    - 偏置被除数向0方向 Biases dividend toward 0

Case 1: No rounding 无需向上取整

被除数 Dividend:

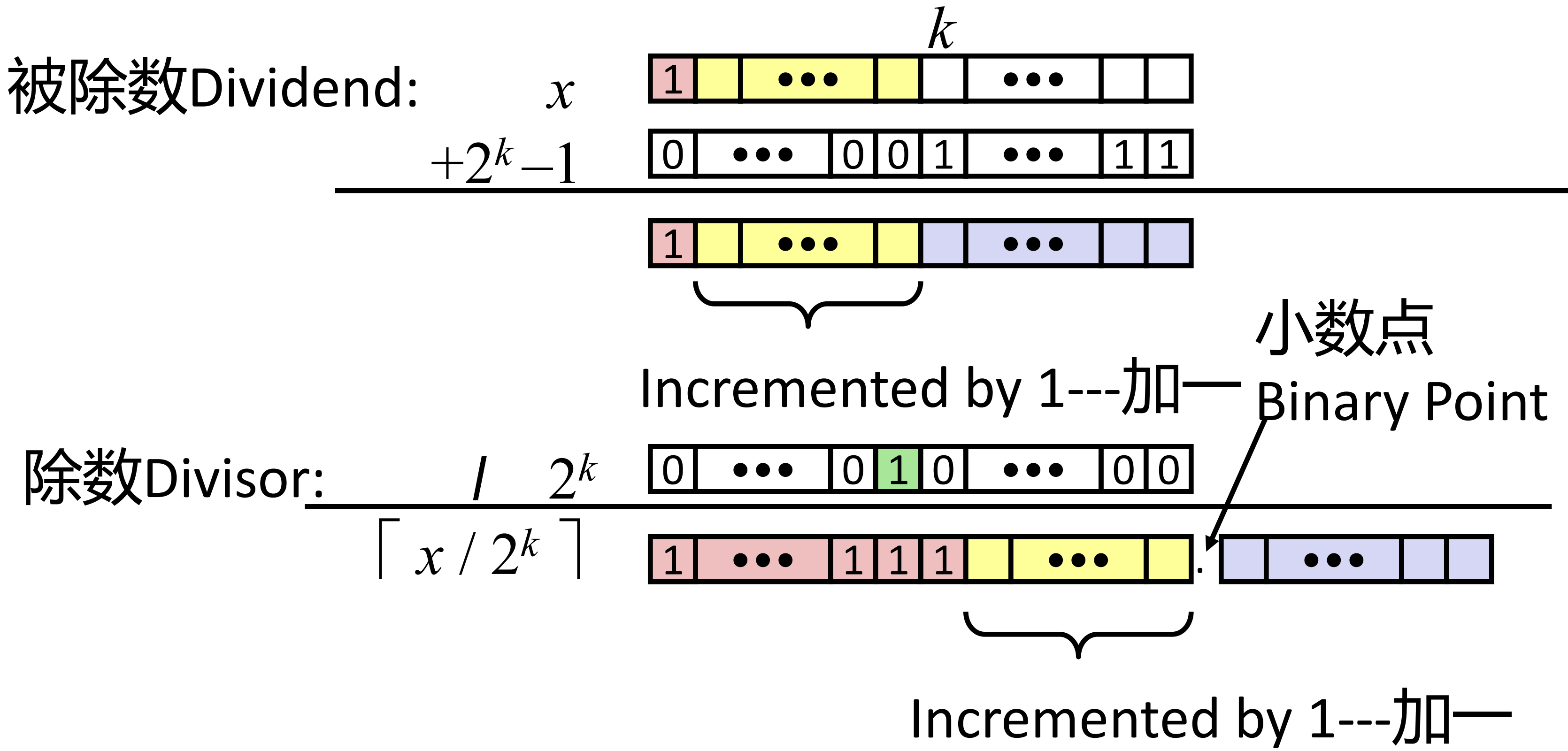
除数 Divisor:



偏置没有影响 Biasing has no effect

修正2的整数次幂除法（续）
Correct Power-of-2 Divide (Cont.)

Case 2: Rounding 向上取整



偏置给最终结果加一 Biasing adds 1 to final result

编译生成的乘法代码
Compiled Multiplication Code

C语言函数 C Function

```
long mul12(long x)
{
  return x*12;
}
```

编译生成的算术运算

Compiled Arithmetic Operations

```
leaq (%rax,%rax,2), %rax
salq $2, %rax
```

解释Explanation

```
t <- x+x*2
return t << 2;
```

- **当乘以常量时，C语言编译器自动生成移位/加法代码**
  C compiler automatically generates shift/add code when multiplying by constant

编译生成无符号数除法代码
Compiled Unsigned Division Code

C语言函数 C Function

```
unsigned long udiv8
      (unsigned long x)
{
  return x/8;
}
```

编译生成的算术运算
Compiled Arithmetic Operations

```
  shrq $3, %rax
```

解释 Explanation

```
  # Logical shift
  return x >> 3;
```

- **对于无符号数使用逻辑移位 Uses logical shift for unsigned**
- **对于Java用户 For Java Users**
  - 逻辑移位记为>>> Logical shift written as >>>

编译生成的有符号数除法代码
Compiled Signed Division Code

C语言函数 C Function

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

编译生成的算术运算

Compiled Arithmetic Operations

```
  testq %rax, %rax
  js    L4
L3:
  sarq $3, %rax
  ret
L4:
  addq $7, %rax
  jmp  L3
```

解释 Explanation

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

- **对于int使用算术移位 Uses arithmetic shift for int**
- **对于Java用户 For Java Users**
  - 算术移位记为>> Arith. shift written as >>>

补码非：求补和递增
Negation: Complement & Increment

- 通过求补和加一得到补码非 Negate through complement and increase

  `~x + 1 == -x`

- 示例 Example
  - Observation: `~x + x == 1111…111 == -1`

```
      x    1 0 0 1 1 1 0 1
+    ~x    0 1 1 0 0 0 1 0
--------------------------
     -1    1 1 1 1 1 1 1 1
```

x = 15213

	Decimal	Hex	Binary
`x`	15213	`3B 6D`	`00111011 01101101`
`~x`	-15214	`C4 92`	`11000100 10010010`
`~x+1`	-15213	`C4 93`	`11000100 10010011`
`y`	-15213	`C4 93`	`11000100 10010011`

求补和递增示例
Complement & Increment Examples

x = 0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

x = TMin

	Decimal	Hex	Binary
x	-32768	80 00	10000000 00000000
~x	32767	7F FF	01111111 11111111
~x+1	-32768	80 00	10000000 00000000

规范的反例 Canonical counter example

议题: 比特、字节和整数

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

算数运算：基本规则
Arithmetic: Basic Rules

- **加法 Addition:**
  - 无/有符号数：正常加法然后截断，比特位级运算是相同的 Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - 无符号数：加法再取模数运算 Unsigned: addition mod 2^w
    - 数学上加法+可能减去模 Mathematical addition + possible subtraction of 2^w
  - 有符号数：修正的模加法（结果在正确的范围）Signed: modified addition mod 2^w (result in proper range)
    - 数学上加法+可能加或减模 Mathematical addition + possible addition or subtraction of 2^w

算数运算：基本规则
Arithmetic: Basic Rules

- **乘法 Multiplication:**
  - 无/有符号数：正常的乘法然后截断，比特位级运算是相同的 Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - 无符号数：乘法取模 Unsigned: multiplication mod 2^w
  - 有符号数：修正的乘法取模（结果在正确范围）Signed: modified multiplication mod 2^w (result in proper range)

运算：基本规则 Arithmetic: Basic Rules

- **无符号数、补码都是同构环：同构=强制类型转换 Unsigned ints, 2’s complement ints are isomorphic rings: isomorphism = casting**

- **左移 Left shift**
  - 无/有符号数：乘以2的整数次幂 Unsigned/signed: multiplication by 2^k
  - 总是逻辑移位 Always logical shift

- **右移 Right shift**
  - 无符号数：逻辑移位，除以2的整数次幂（除法+向0舍入）Unsigned: logical shift, div (division + round to zero) by 2^k
  - 有符号数：算术移位 Signed: arithmetic shift
    - 正数：除以2^k（除法+向0舍入）Positive numbers: div (division + round to zero) by 2^k
    - 负数：除以2^k（除法+远离0舍入），使用偏置修正 Negative numbers: div (division + round away from zero) by 2^k Use biasing to fix

无符号数运算的性质
Properties of Unsigned Arithmetic

- **用加法的无符号数乘法形成交换环 Unsigned Multiplication with Addition Forms Commutative Ring**
  - 加法是具有交换性的群组 Addition is commutative group
  - 封闭的乘法 Closed under multiplication

    $0 \leq \mathrm{UMult}_w(u\ ,\ v) \leq 2^w - 1$
  - 乘法具有交换性 Multiplication Commutative

    $\mathrm{UMult}_w(u\ ,\ v) = \mathrm{UMult}_w(v\ ,\ u)$
  - 乘法具有结合性 Multiplication is Associative

    $\mathrm{UMult}_w(t, \mathrm{UMult}_w(u\ ,\ v)) = \mathrm{UMult}_w(\mathrm{UMult}_w(t, u\ ), v)$
  - 1是乘性恒等的（单位元） 1 is multiplicative identity

    $\mathrm{UMult}_w(u\ ,\ 1) = u$
  - 乘法对加法具有分配性 Multiplication distributes over addtion

    $\mathrm{UMult}_w(t, \mathrm{UAdd}_w(u\ ,\ v)) = \mathrm{UAdd}_w(\mathrm{UMult}_w(t, u\ ), \mathrm{UMult}_w(t, v))$

补码运算的属性
Properties of Two's Comp. Arithmetic

- **同构的代数 Isomorphic Algebras**
  - 无符号数乘法和加法 Unsigned multiplication and addition
    - 截断到w位 Truncating to *w* bits
  - 补码乘法和加法 Two's complement multiplication and addition
    - 截断到w位 Truncating to *w* bits
- **都形成闭环 Both Form Rings**
  - 同构于整数模2^w的闭环 Isomorphic to ring of integers mod 2^w
- **数学整数运算比较Comparison to(Mathematical)Integer Arithmetic**
  - 都是闭环 Both are rings
  - 整数遵循按序属性 Integers obey ordering properties, e.g.,

$$u > 0 \quad \Rightarrow \quad u + v > v$$

$$u > 0, v > 0 \quad \Rightarrow \quad u \cdot v > 0$$

  - 补码运算不遵循的属性These properties not obeyed by two's comp.arithmetic

```
TMax + 1 == TMin
15213 * 30426 == -10030
```

(16-bit words)

为什么应该使用无符号数? Why Should I Use Unsigned?

- **在没有理解实现方法的情况下*不要使用* *Don't* use without understanding implications**
  - 容易犯错误 Easy to make mistakes

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

  - 可能非常微妙 Can be very subtle

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

用无符号数倒计数
Counting Down with Unsigned

- **使用无符号数作为循环索引的正确方法 Proper way to use unsigned as loop index**

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

- **参考材料 See Robert Seacord, *Secure Coding in C and C++***
  - C语言标准确保无符号加法行为类似于取模运算 C Standard guarantees that unsigned addition will behave like modular arithmetic
    - 0 – 1 → *UMax*
- **更好的办法 Even better**

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

  - 数据类型size_t定义为长度为字长的无符号值 Data type `size_t` defined as unsigned value with length = word size
  - 即使在cnt为Umax时仍然正常运行 Code will work even if `cnt` = *UMax*
  - 如果cnt是有符号数且小于零又怎样呢？ What if `cnt` is signed and < 0?

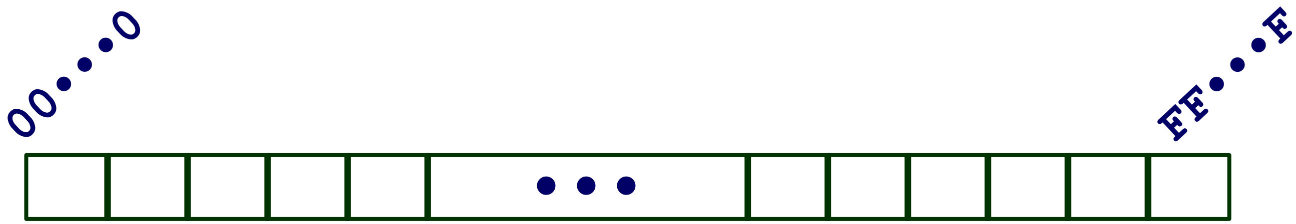
为什么应该使用无符号数?
Why Should I Use Unsigned? (cont.)

- **当执行模运算时*使用无符号数* *Do* Use When Performing Modular Arithmetic**
  - 多精度运算 Multiprecision arithmetic
- **当使用比特位表示集合时*使用无符号数* *Do* Use When Using Bits to Represent Sets**
  - 逻辑右移，无需符号扩展 Logical right shift, no sign extension
- **在系统编程时*使用无符号数* *Do* Use In System Programming**
  - 位掩码、设备命令。。。 Bit masks, device commands,...

议题: 比特、字节和整数
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**

面向字节的内存组织
Byte-Oriented Memory Organization



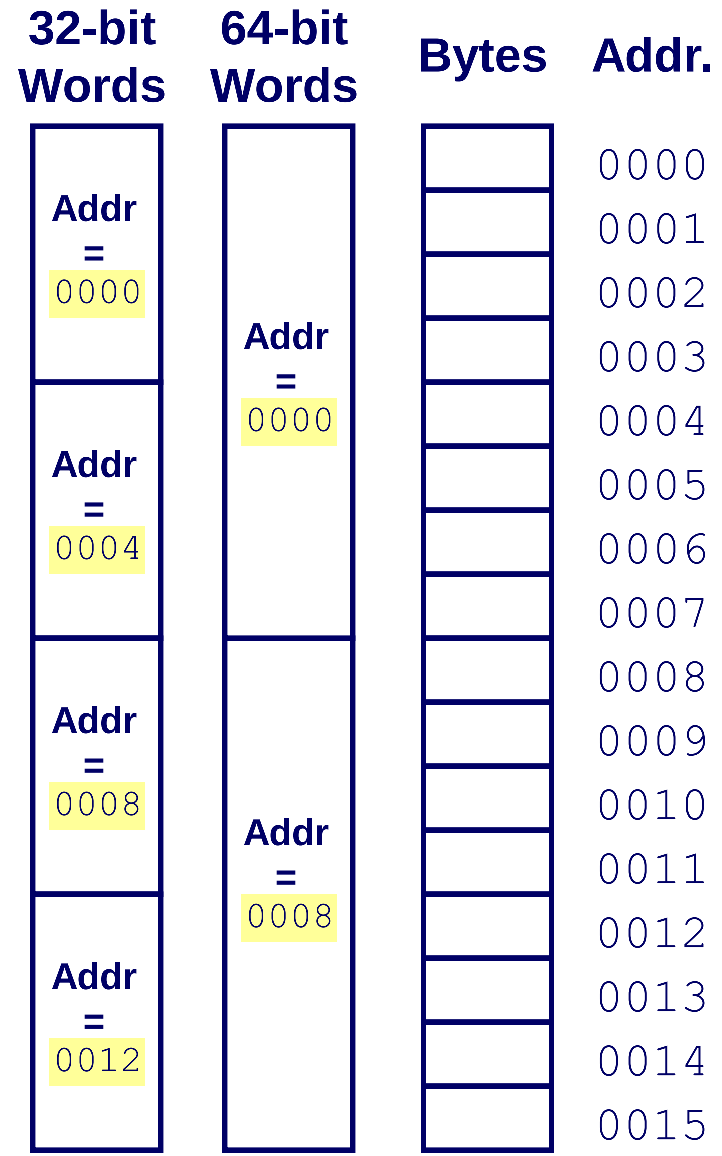
- **程序按照地址引用数据 Programs refer to data by address**
  - 概念上，将其想象成一个非常大的字节数组 Conceptually, envision it as a very large array of bytes
    - 事实上，并非如此，但是可以这样看待 In reality, it’s not, but can think of it that way
  - 地址就像是数组的索引 An address is like an index into that array
    - 而且指针变量存储地址 and, a pointer variable stores an address
- **注意：系统给每个进程提供私有地址空间 Note: system provides private address spaces to each “process”**
  - 进程看成执行中的程序 Think of a process as a program being executed
  - 因此，程序可以任意处理自己的数据，但是不能处理其他程序数据 So, a program can clobber its own data, but not that of others

机器字 Machine Words

- **任何特定的计算机都有“字长” Any given computer has a “Word Size”**
  - 整数值数据的标称大小 Nominal size of integer-valued data
    - 以及地址 And of addresses
  - 直到最近，大多数机器使用32位（4字节）作为字长 Until recently, most machines used 32 bits (4 bytes) as word size
    - 地址局限到4GB Limits addresses to 4GB (2^{32} bytes)
  - 机器字长增长为64位 Increasingly, machines have 64-bit word size
    - 潜在地，可以有18EB地址空间 Potentially, could have 18 EB (exabytes) of addressable memory
    - 即 That’s 18.4 X 10^{18}
  - 机器还支持多种数据格式 Machines still support multiple data formats
    - 字长的部分或倍数 Fractions or multiples of word size
    - 总是字节的整数倍 Always integral number of bytes

面向”字”的内存组织 Word-Oriented Memory Organization

- **地址指定字节位置 Addresses Specify Byte Locations**
  - 字中第一个字节的地址 Address of first byte in word
  - 后继字地址相差4字节（32位）或8字节（64位） Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



数据表示的示例
Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	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

字节顺序 Byte Ordering

- **因此，字中的多个字节在内存里如何排序？So, how are the bytes within a multi-byte word ordered in memory?**
- **约定 Conventions**
  - 大端法：Big Endian: Sun, PPC Mac, Internet
    - 最低有效字节有最高的地址 Least significant byte has highest address
  - 小端法：Little Endian: x86, ARM processors running Android, iOS, and Windows
    - 最低有效字节有最低地址 Least significant byte has lowest address

字节顺序示例 Byte Ordering Example

- **示例 Example**
  - 变量x有4字节值 Variable x has 4-byte value of 0x01234567
  - x的地址为0x100 Address given by &x is 0x100

大端法 Big Endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		

小端法 Little Endian

		0x100	0x101	0x102	0x103		
		67	45	23	01		

表示整数
Representing Integers

Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D

`int A = 15213;`

IA32, x86-64	Sun
6D	00
3B	00
00	3B
00	6D

`int B = -15213;`

IA32, x86-64	Sun
93	FF
C4	FF
FF	C4
FF	93

补码表示 Two’s complement representation

`long int C = 15213;`

IA32	x86-64	Sun
6D	6D	00
3B	3B	00
00	00	3B
00	00	6D
	00	
	00	
	00	
	00	

检查数据表示

Examining Data Representations

- **打印数据的字节表示的代码 Code to Print Byte Representation of Data**
  - 强制类型转换无符号字符指针允许作为字节数组对待 Casting pointer to unsigned char * allows treatment as a byte array

```
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");
}
```

格式指示符 Printf directives:

%p: 打印指针 Print pointer

%x: 打印十六进制 Print Hexadecimal

show_bytes执行示例
show_bytes Execution Example

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

结果 Result (Linux x86-64):

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

表示指针 Representing Pointers

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

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

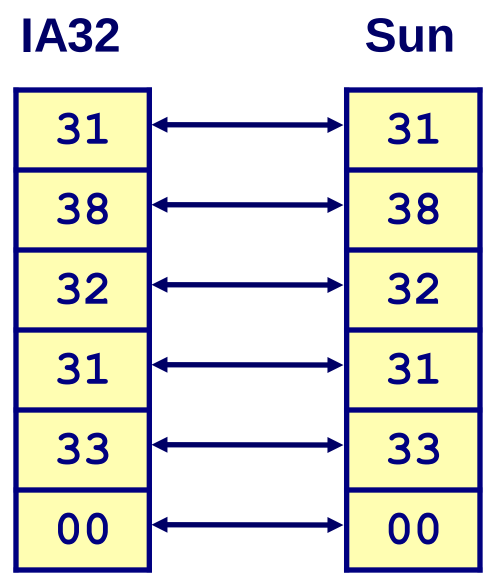
不同的编译器和机器分配不同的位置给对象 Different compilers & machines assign different locations to objects

甚至每次运行程序会得到不同的结果 Even get different results each time run program

表示字符串 Representing Strings

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

- **C语言中的字符串 Strings in C**
  - 用字符数组来代表 Represented by array of characters
  - 每个字符编码成ASCII格式 Each character encoded in ASCII format
    - 标准7位字符集编码 Standard 7-bit encoding of character set
    - 字符‘0’编码为0x30 Character “0” has code 0x30
      - 数字i代码为0x30+i Digit *i* has code 0x30+*i*
  - 字符串应该以空作为结尾 String should be null-terminated
    - 最后的字符为0 Final character = 0
- **兼容性 Compatibility**
  - 字节顺序不存在问题 Byte ordering not an issue



阅读逆序字节列表

Reading Byte-Reversed Listings

- **反汇编 Disassembly**
  - 二进制机器代码的文本表示 Text representation of binary machine code
  - 由读取机器代码的程序生成 Generated by program that reads the machine code
- **示例片段 Example Fragment** **汇编表示**

Address	Instruction Code	Assembly Rendition
`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)`

- **解密数值 Deciphering Numbers**
  - 值：Value: 0x12ab
  - 填充到32位：Pad to 32 bits: 0x000012ab
  - 分成字节 Split into bytes: 00 00 12 ab
  - 逆序 Reverse: ab 12 00 00

C语言整数难题 Integer C Puzzles

- `x < 0` ⇒ `((x*2) < 0)` ✗
- `ux >= 0` ✓
- `x & 7 == 7` ⇒ `(x<<30) < 0` ✓
- `ux > -1` ✗
- `x > y` ⇒ `-x < -y` ✗
- `x * x >= 0` ✗
- `x > 0 && y > 0` ⇒ `x + y > 0` ✗
- `x >= 0` ⇒ `-x <= 0` ✓
- `x <= 0` ⇒ `-x >= 0` ✗
- `(x|-x)>>31 == -1` ✗
- `ux >> 3 == ux/8` ✓
- `x >> 3 == x/8` ✗
- `x & (x-1) != 0` ✗

初始时 Initialization

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