



2023秋季

计算机系统概论

Introduction to Computer Systems

整 数

⊗ Zhang, Youhui (张悠慧)

✉ zyh02@tsinghua.edu.cn



计算机体系结构

》 C程序在硬件层面的表示

数据

- 整数 / 浮点数 (第二讲)
- 数组 / 结构 (第五讲)

代码

•

```
int array[4] = {1, 2, 3, 4};
static int brray[4] = {1, 2, 3, 4};    main.c

static int intra_sum (int x[4], int y)
{
    return x[y-1];
}

int main()
{
    int val = intra_sum(array, 3) + inter_sum(brray, 3);
    return val;
}
```

编译

链接

```
intra_sum:
    movslq    %esi, %rsi
    movl      -4(%rdi,%rsi,4), %eax
    ret

main:
    pushq     %rbx
    movl      $3, %esi
    movl      @array, %edi
    callq     intra_sum
    movl      %eax, %ebx// val -> ebx
    movl      $3, %esi
    movl      @brray, %edi
    movl      $0, %eax
    callq     inter_sum
    addl      %ebx, %eax// val -> eax
    popq      %rbx
    ret

brray:
    .long     1
    .long     2
    .long     3
    .long     4

array:
    .long     1
    .long     2
    .long     3
    .long     4

main.o
```

CPU

P
C

Registers

Condition Codes

运行

```
0000000004004de <main>:
4004de: 53                push    %rbx
4004df: be 03 00 00 00    mov     $0x3,%esi
4004e4: bf 40 10 60 00    mov     $0x601040,%edi
4004e9: e8 e8 ff ff ff    callq   4004d6 <intra_sum>
4004ee: 89 c3             mov     %eax,%ebx
4004f0: be 03 00 00 00    mov     $0x3,%esi
4004f5: bf 30 10 60 00    mov     $0x601030,%edi
4004fa: b8 06 00 00 00    mov     $0x0,%eax
4004ff: e8 04 00 00 00    callq   400508 <inter_sum>
400504: 01 d8            add     %ebx,%eax
400506: 5b               pop     %rbx
400507: c3               retq
```

内存地址

仅给出部分内容

地址/Addresses

数据/Data

指令/Instructions

Memory

数据段

代码段

程序/数据在机器层面的表示与运行



目录

CONTENTS

01

数的机器表示（初步）



02

整数的表示



预备知识

- **1K** = 2^{10} = 1024 (kilo)
- **1G** = 1024M = 2^{30} (Giga)
- **1P** = 1024T = 2^{50} (Peta)
- **1M** = 1024K = 2^{20} (Mega)
- **1T** = 1024G = 2^{40} (Tera)
- **1E** = 1024P = 2^{60} (Exa)

- 1个二进制位: **bit** (比特)
- 8个二进制位: **Byte** (字节) 1Byte = 8bit
- 2个字节: **Word** (字) 1Word = 2Byte = 16bit*

* X86架构下如此, RISC-V的话1-word = 32-bit

• 机器字(machine word)长

» 一般指计算机进行一次整数运算所能处理的二进制数据的位数

▸ 通常也指数据地址长度

» 32位字长

▸ 地址的表示空间是4GB

▸ 对很多内存需求量大的应用而言，非常有限

» 64位字长

▸ 地址的表示空间约是 1.8×10^{19} bytes

▸ 目前的X86-64 机型实际支持 48位宽的地址: 256 TB

机器字在内存中的组织

地址按照字节(byte)来定位

- ▶ 机器字中**第一个字节**的地址
- ▶ **相邻机器字的地址**相差4 (32位字) 或者8 (64位字)

Addr
= 0000

Addr
= 0004

Addr
= 0008

Addr
= 0012

32位字编址

Addr
= 0000

Addr
= 0008

64位字编址

	0000
	0001
	0002
	0003
	0004
	0005
	0006
	0007
	0008
	0009
	0010
	0011
	0012
	0013
	0014
	0015

字节编址 地址

字节序 (Byte Ordering)

一个机器字内的各个字节如何排列?

- ▶ **大端** (Big Endian) : Sun, PowerPC, Internet, Java
低位字节(Least significant byte, LSB) 占据高地址
- ▶ **小端** (Little Endian) : X86, RISC-V, ARM(默认)
与LSB相反

数值是0x01234567, 地址是0x100

Big Endian

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

Little Endian

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



Gulliver's Travels

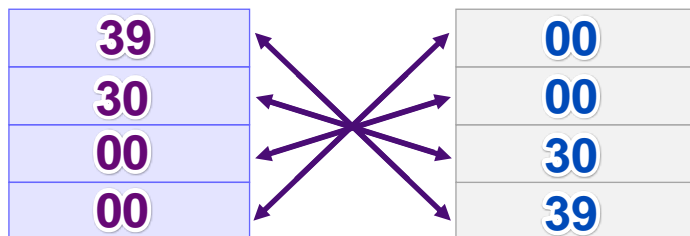
字节序 (Byte Ordering)

```
int A = 12345 ;  
int B = -12345 ;  
long int C = 12345 ;
```

十进制: **12345**
二进制: **0011 0000 0011 1001**
十六进制: **3 0 3 9**

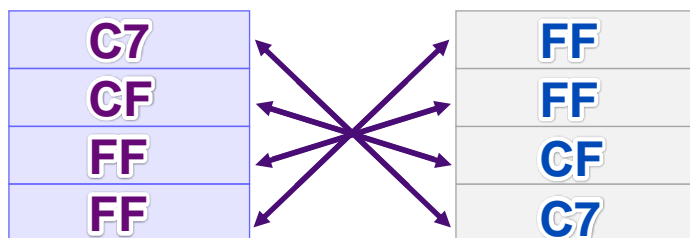
IA32, x86-64

Sparc



IA32, x86-64

Sparc

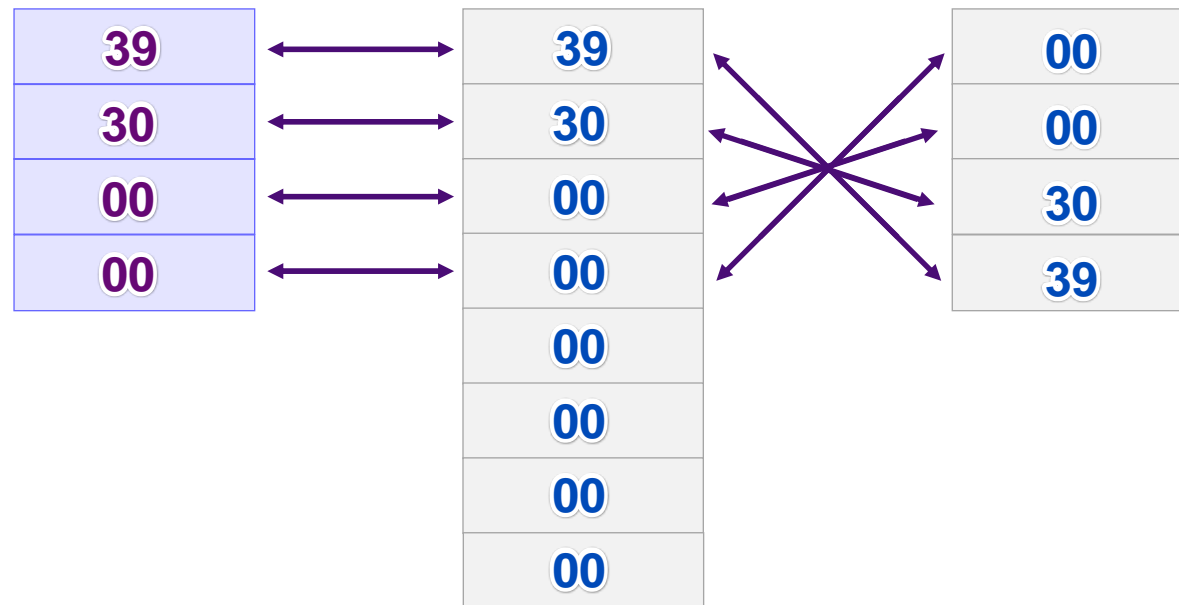


补码表示

IA32

x86-64

Sparc



Show Bytes示例

```
typedef unsigned char *byte_pointer;
```

```
void show_bytes(byte_pointer start, size_t len)
{
    for (int i = 0; i < len; i++)
    {
        printf("%.2x\t", start[i]);
    }
}
```

```
void show_int(int x)
{
    show_bytes((byte_pointer)&x, sizeof(int));
}
```

```
void show_double(double x)
...
```

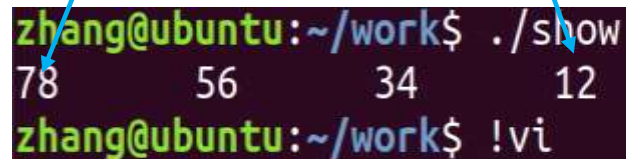
```
void show_float(float x)
...
```

```
int main(void)
{
    int x = 0x12345678;
    show_int(x);
}
```

start[0]		start[1]		start[2]		start[3]	
byte0	byte1	byte2	byte3	byte4	byte5	byte6	byte7
8	7	6	5	4	3	2	1

LSB

MSB (most significant bit)



```
zhang@ubuntu:~/work$ ./show
78      56      34      12
zhang@ubuntu:~/work$ !vi
```

C语言中基本数据类型的大小 (单位为字节)

C Data Type	32-bit	x86-32	x86-64	
char	1	1	1	
short	2	2	2	
int	4	4	4	
long	4	4	8	(Linux)
long long	8	8	8	
float	4	4	4	
double	8	8	8	
long double	8	10/12	10/16	
char * or any other pointer	4	4	8	
	/	/	/	


计算机中整数的二进制编码方式 (w表示字长)

无符号数 (原码表示)

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

带符号数 (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

short int x = 12345 ;
short int y = - 12345 ;

	Decimal	Hex	Binary	
x	12345	30 39	00110000	00111001
y	-12345	CF C7	11001111	11000111

符号位 (sign bit)

- 对于补码表示, MSB (Most Significant Bit) 表示整数的符号

▶ 1 for negative ▶ 0 for nonnegative

- 非负数: 补码 = 原码 (反码是原码各位取反)
- 负数: 补码 = 绝对值的反码 + 1 = 2^w + 该负数
- 一个数的补码: 它的相反数的补码的按位取反加一

取值范围

无符号数

- ▶ $U_{\min} = 0$ 000...0
- ▶ $U_{\max} = 2^w - 1$ 111...1

带符号数

- ▶ $T_{\min} = -2^{w-1}$ 100...0
- ▶ $T_{\max} = 2^{w-1} - 1$ 011...1
- ▶ 负1 = 111...1

假设字长为16(w=16)

	Decimal	Hex	Binary
U_{\max}	65535	FF FF	11111111 11111111
T_{\max}	32767	7F FF	01111111 11111111
T_{\min}	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

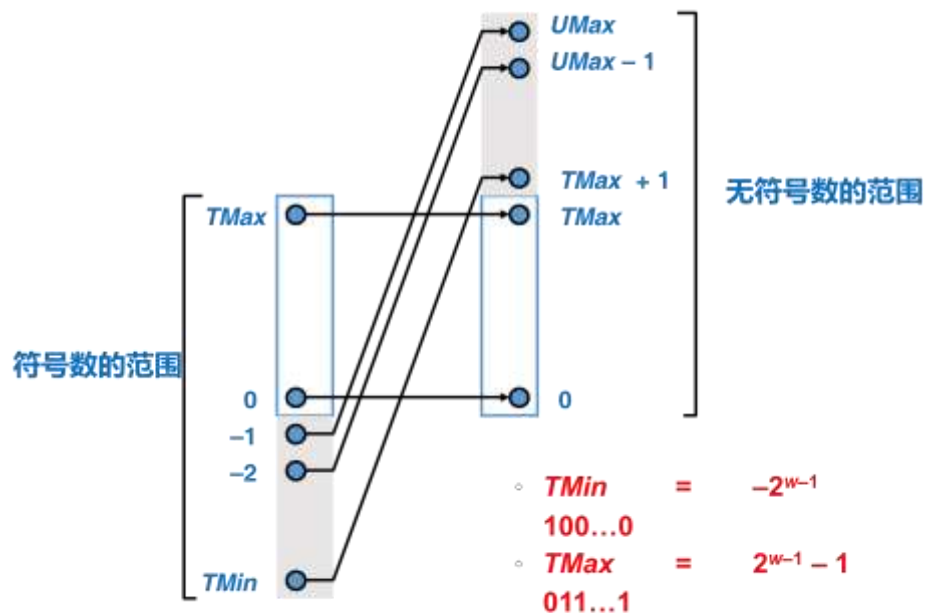
无符号数与带符号数

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

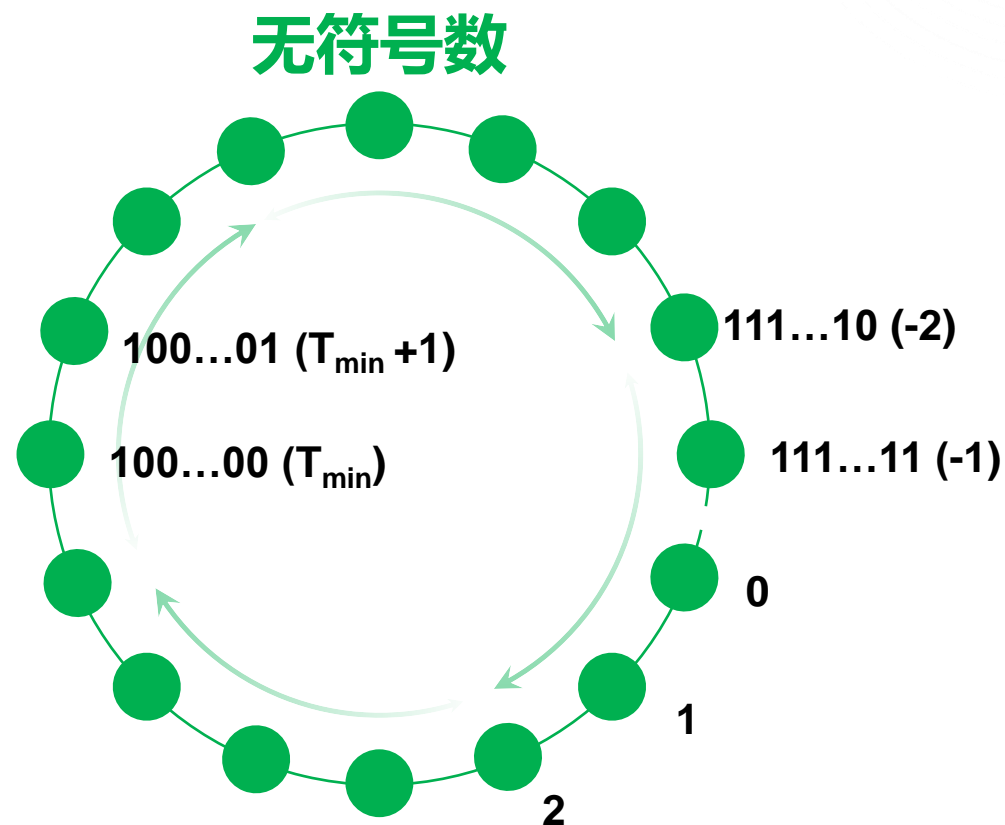
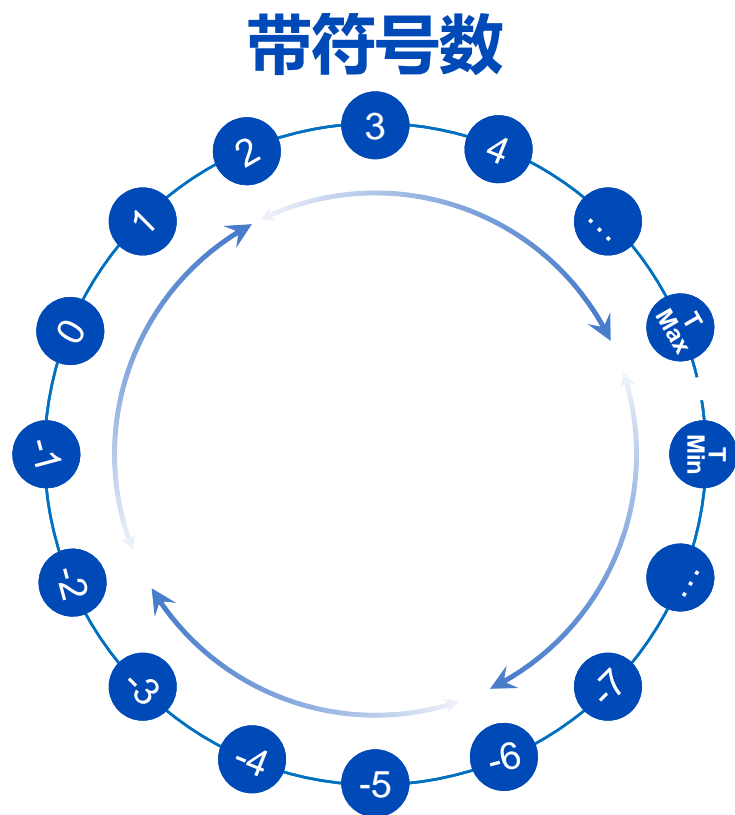
》 无符号数与带符号数之间的转换

- ## 二进制位串表示是不变的

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

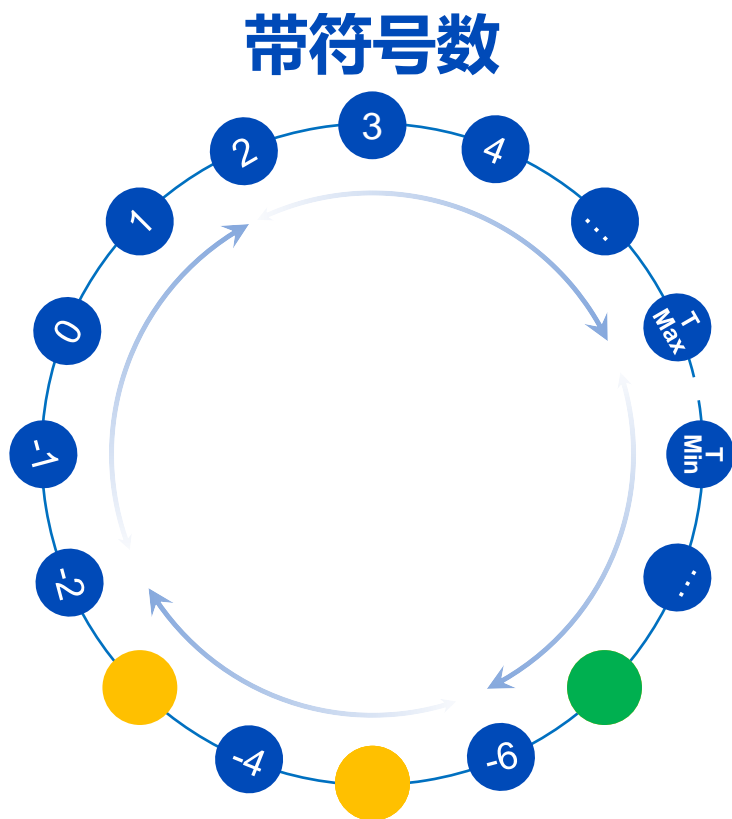


带符号/无符号整数的加减操作



带符号数的补码系统的核心： 把负整数映射到无符号数的高数值范围

带符号/无符号整数的加减操作



x: 圆环上的任意一个位子 | **y**: 正数 ($y > 0$)

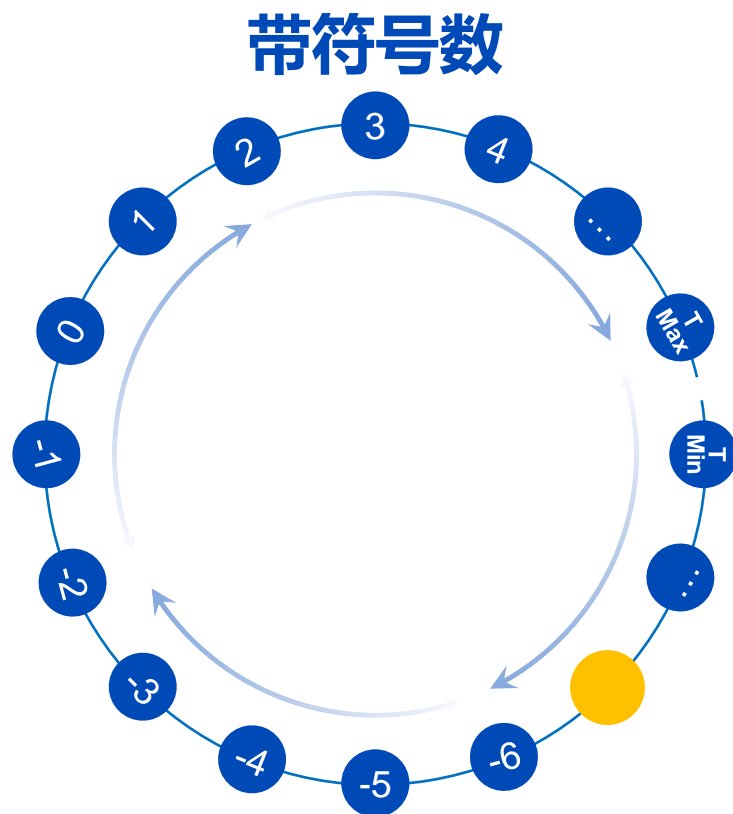
$x+y$: x顺时针移动y个位子

$x-y$: x逆时针移动y个位子

x逆时针移动y个位子 = x顺时针移动 $(R-y)$ 个位子

w位整数: $R=2^w$

带符号/无符号整数的加减操作



减法可以用取反和加法代替
实现了加法和减法在电路层面的统一

公式 1

$$\begin{aligned}x - y &= x \text{ 逆时针移动 } y \text{ 个位子} \\ &= \text{顺时针移动 } [R - y] \text{ 个位子}\end{aligned}$$

公式 2

$$\begin{aligned}R - 1 &= y + (\sim y) = \text{全1的}w\text{位二进制串} \\ R - y &= (\sim y) + 1\end{aligned}$$

公式 3

$$\begin{aligned}x - y &= x + (R - y) \\ -y &= R - y \\ -y &= (\sim y) + 1 \\ x - y &= x + (\sim y) + 1\end{aligned}$$

补码加法公式*



$$\gg [x]_{\text{补}} + [y]_{\text{补}} \equiv [x + y]_{\text{补}} \pmod{2^w}$$

意义:

负整数用补码表示后, 可以和正整数一样来处理, 这样 (处理器的) 运算器只需要加法器

证明:

$$[y]_{\text{补}} = 2^w + y \quad (y < 0)$$

- $x > 0; y > 0$ 由于正数的补码和原码一致, $x + y > 0$, 所以在这种情况下 $[x]_{\text{补}} + [y]_{\text{补}} \equiv [x + y]_{\text{补}}$
- $x > 0; y < 0$ 且 $x + y > 0$ 我们有如下的等式: $[x]_{\text{补}} = x$ $[y]_{\text{补}} = 2^w + y$, 所以, $[x]_{\text{补}} + [y]_{\text{补}} = x + y + 2^w \equiv x + y = [x + y]_{\text{补}}$:
- $x > 0; y < 0$ 且 $x + y < 0$, 可以发现以下等式: $[x]_{\text{补}} = x$ $[y]_{\text{补}} = 2^w + y$, 所以, $[x]_{\text{补}} + [y]_{\text{补}} = x + y + 2^w = [x + y]_{\text{补}}$:

C语言中的无符号数与带符号数

》 常数 (Constants)

- 默认是带符号数
- 如果有“U” 作为后缀则是无符号数，如 0U, 4294967259U

》 无符号数与带符号数混合使用

- 带符号数默认转换为无符号数
- 包括比较操作符 实例 (w=32)

Constant ₁	Constant ₂	比较大小
0	0U	
-1	0	
-1	0U	
2147483647	-2147483647-1	
2147483647U	-2147483647-1	
-1	-2	
(unsigned)-1	-2	
2147483647	2147483648U	
2147483647	(int)2147483648U	

C语言中的无符号数与带符号数

Constant ₁	Constant ₂	比较大小	带符号数或无符号数处理
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

何时采用无符号数

- 模运算
- 按位运算
- 建议：不能仅仅因为取值范围是非负而使用

» 示例一

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

» 示例二

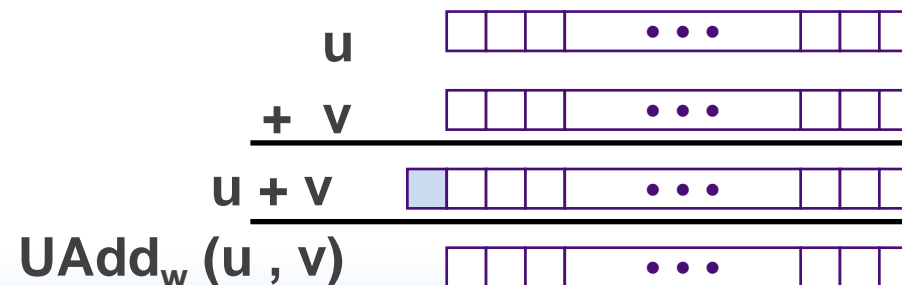
```
#define DELTA sizeof(int)  
int i;  
for (i = CNT; i-DELTA >= 0; i-= DELTA)  
    ...
```

无符号数加法

操作数位宽: w bits

真实结果位宽: $w + 1$ bits

放弃进位: w bits



$$s = UAdd_w(u, v) = (u + v) \bmod 2^w$$



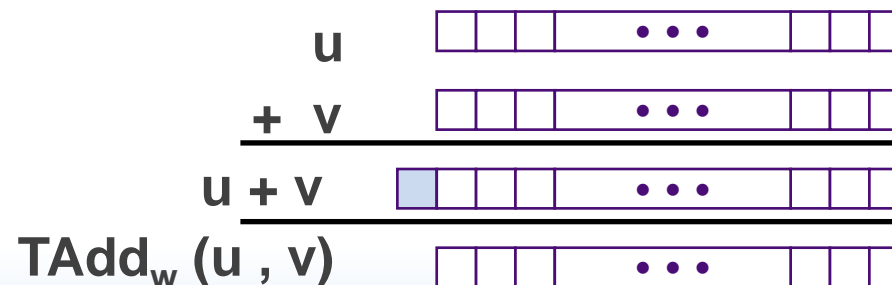
$$UAdd_w(u, v) = \begin{cases} u + v & u + v < 2^w \\ u + v - 2^w & u + v \geq 2^w \quad \text{溢出} \end{cases}$$

补码加法

操作数位宽: w bits

真实结果位宽: $w + 1$ bits

放弃进位: w bits



》 与无符号数的一致

- C语言中的带符号数 / 无符号数加法

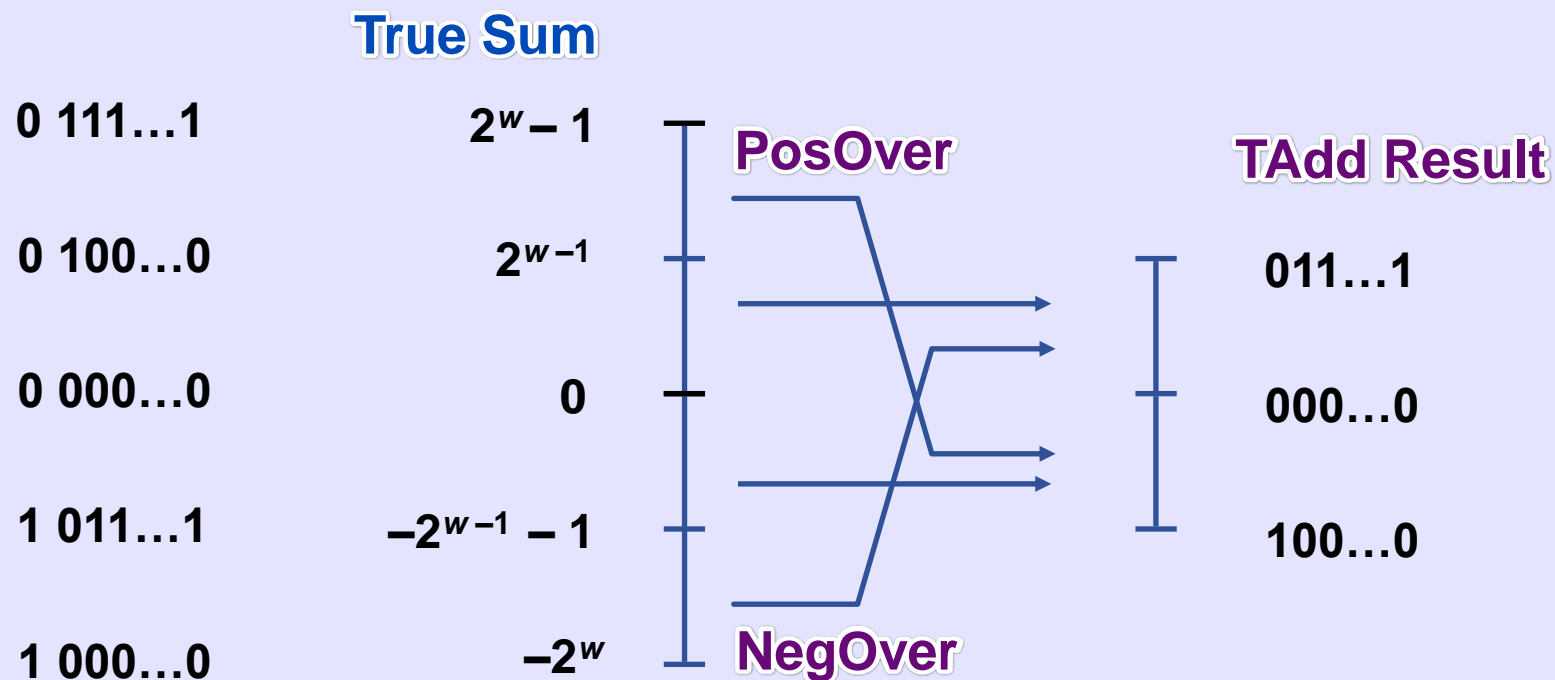
```
int s, t, u, v;
```

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

```
t = u + v
```

- $s == t$

补码加法的溢出



$$TAdd_w(u,v) = \begin{cases} u + v + 2^w & u + v < TMin_w \quad \text{NegOver / 向下溢出} \\ u + v & TMin_w \leq u + v \leq TMax_w \\ u + v - 2^w & TMax_w < u + v \quad \text{PosOver / 向上溢出} \end{cases}$$

填空题

101: 0x00000067

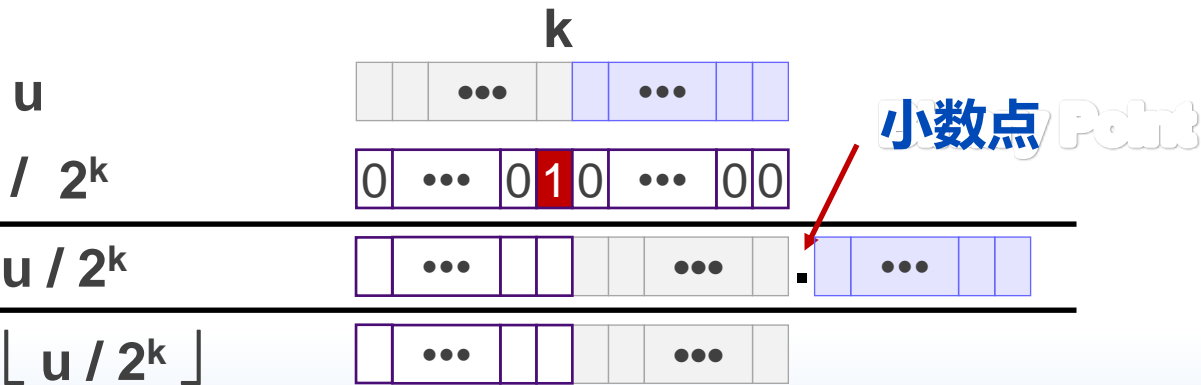
- 已知某32位整数X，其值为-101（十进制）
则其16进制补码为 [填空1] 0x FFFFFFF8 + 1
- 另一32位整数Y的补码为0xFFFFF6A
则 X + Y 的16进制补码（32位）为 [填空2]
X - Y 的16进制补码为 [填空3]

无符号整数除以2的k次幂

$u \gg k$ 等价于 $\lfloor u / 2^k \rfloor$

采用逻辑右移

操作数:



结果:

	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

无符号整数除以2的k次幂

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

等价的汇编代码

```
shrl    $3, %eax
```

C代码的解释

```
# 逻辑右移
return x >> 3;
```

带符号整数除以2的k次幂

$x \gg k$ 等价于 $\lfloor x / 2^k \rfloor$

采用算术右移

但是 $x < 0$ 时, 舍入错误

操作数:

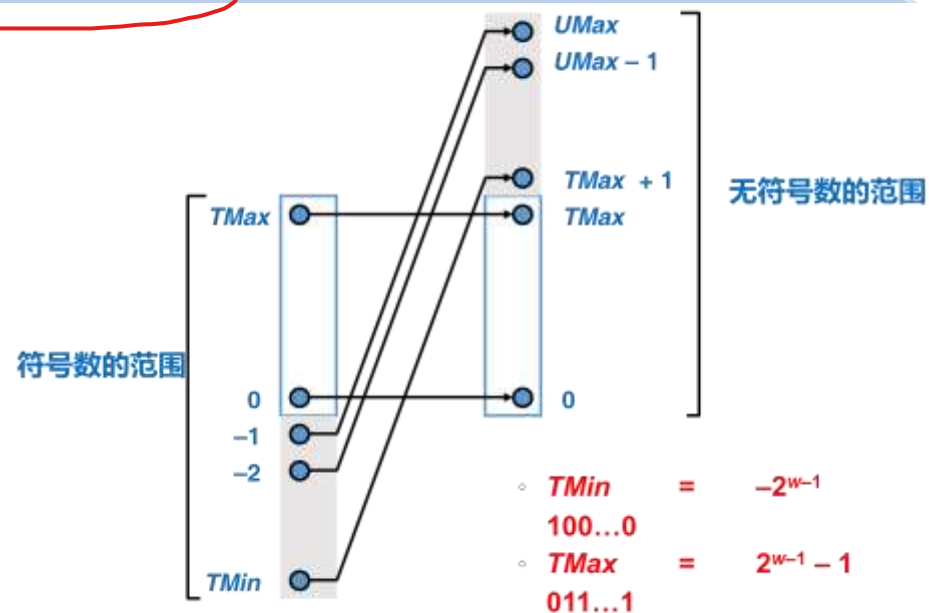
u

$/ 2^k$

$u / 2^k$

$\lfloor u / 2^k \rfloor$

结果:



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

带符号整数除以2的k次幂

期望的结果是 $\lceil x / 2^k \rceil$ (需要向0舍入, 而不是向"下"舍入)

→ Bias / 偏置量

》 所以引入偏置量 $\lfloor (x+2^{k-1}) / 2^k \rfloor$

► **C语言:** $(x + (1 \ll k) - 1) \gg k$

情况一：能够被 2^k 整除

被除数:

Diagram illustrating the division of a number u by 2^k to find the integer part of $u/2^k$.

The number u is represented as a binary string: $1 \dots 0 \dots 00$.

The divisor is 2^k , represented as a binary string: $0 \dots 001 \dots 11$.

The division result is shown as a binary string with a decimal point (小数点): $1 \dots 1 \dots 11$.

The integer part of the result is $\lfloor u / 2^k \rfloor$, represented as a binary string: $1 \dots 111 \dots$.

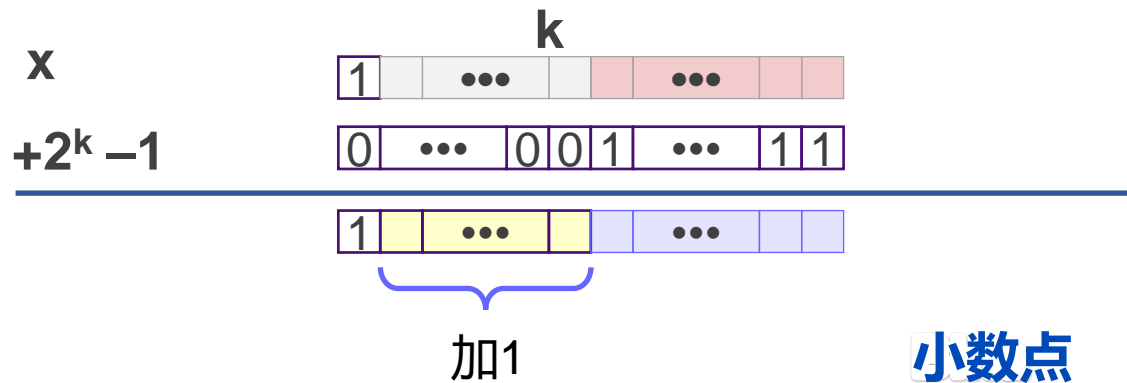
The remainder is $u \bmod 2^k$, represented as a binary string: $0 \dots 010 \dots 00$.

此时偏置量不起作用

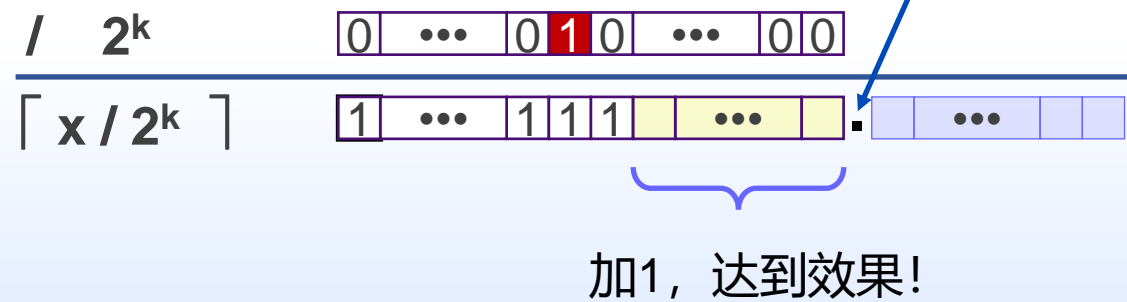
带符号整数除以2的k次幂

情况二

被除数:



除数:



带符号整数除以2的k次幂

C函数

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

等价的汇编代码

```
testl    %eax, %eax
js       L4
L3:      sarl    $3, %eax
ret
L4:      addl    $7, %eax
jmp      L3
```

C代码的解释

```
if x < 0
    x += 7;
# 算术右移
return x >> 3;
```

整数运算的一些可能极端情况

判断以下的推断或者关系式是否成立 (不成立则给出示例)

- x, y 为32位带符号整数
- ux, uy 为与 x, y 具有相同二进制表示的32位无符号整数

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

$x=0$

以上仅从带符号整数、无符号数的定义出发，不涉及C语言的编译器具体实现，有兴趣同学可以深入参考C语言的“未定义行为”