

《计算机系统》 整数

湖南大学

《计算机系统》课程教学组



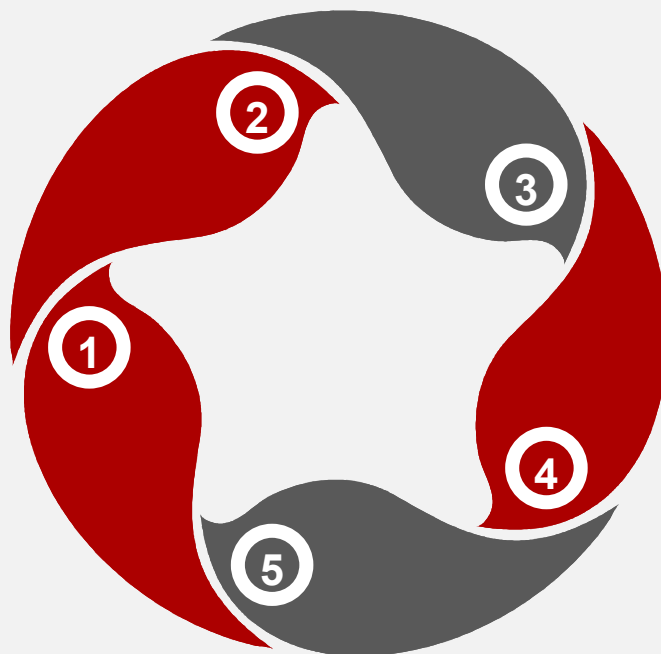
内容提要

转换 & 类型变换

扩展 & 截断

有符号数 & 无符号数

整数的运算操作



小结

整型数据 (32位机)

C数据类型	最小值	最大值
char	-128	127
unsigned char	0	255
short	-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	-2 147 483 648	2 147 483 647
unsigned long	0	4 294 967 295
long long	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	0	18 446 744 073 709 551 615

整型数据 (64位机)

C数据类型	最小值	最大值
char	-128	127
unsigned char	0	255
short	-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	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long	0	18 446 744 073 709 551 605
long long	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	0	18 446 744 073 709 551 615

整数的编码

无符号数

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

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

二进制补码

符号位

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

● C 中短整型为两个字节长度

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

● 符号位

- 二进制补码中，最高位为符号位
 - 0 表示非负数
 - 1 表示负数

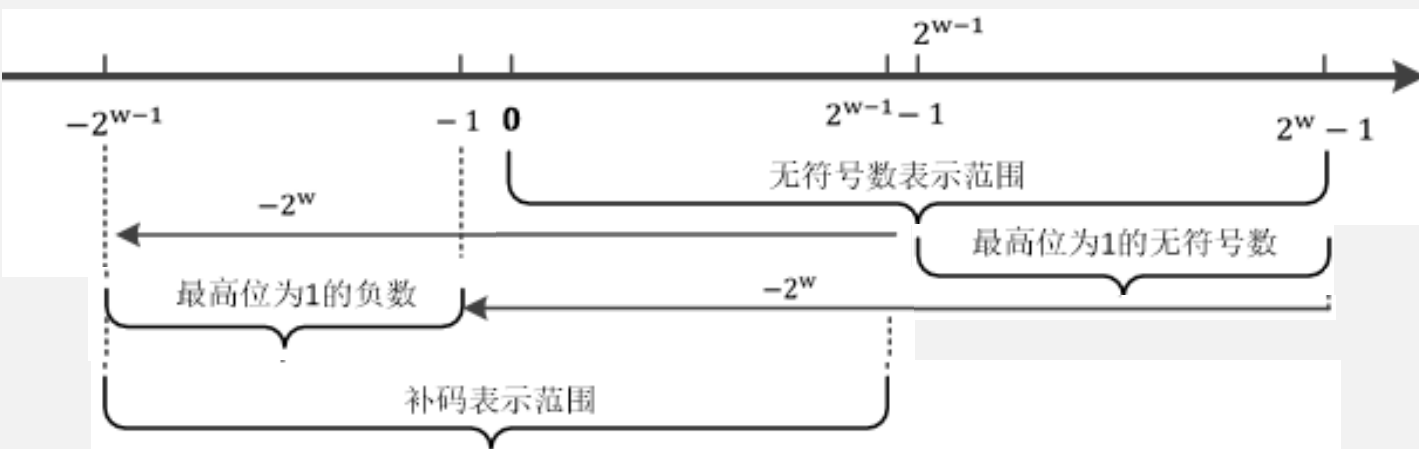
补码的本质

$$B2T_w(x) \equiv -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

用最高位的**负权重**将数值往数轴**负方向**迁移



从9点往前7小时，或往后5小时都指向4点；
在取值**12**的范围内，**+7**和**-5**形成绝对值为**12**的“**互补**”



将无符号数中**最高位为1**的一半
用来表示负数

补码的本质

以 $w = 4$ 为例，可以表示从 0000~1111 共16个数，“表盘”大小就是16。

做无符号数解释：则是从 0~15 这16个非负数；

做有符号数解释：

0000~0111 这八个数依然是0~7不变，所以正数的补码依然是其本身；

1000~1111 这8个数被用来表示负数，这其中每一个数表示的负数就是其对应的正数在16这个范围内有互补关系的负数。

例如：1010原本表示的是+10，在16的范围内对应的互补关系是-6，所以1010就是-6的补码表示。

补码的值

补码与十进制相互转换使用的值盒子及示例

-128	64	32	16	8	4	2	1

8位二进制补码的值盒子

-128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

例1: $-125 = -128 + 2 + 1$

-128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

例2: $-120 = -128 + 8$

补码示例

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

Weight	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	

补码取值范围

	8位机		16位机	
	unsigned	signed	unsigned	signed
原码	0~255	-127~+127	0~65535	-32767~+32767
反码	0~255	-127~+127	0~65535	-32767~+32767
补码	0~255	-128~+127	0~65535	-32768~+32767

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

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

● 观察

- $|TMin| = TMax + 1$
 - 非对称取值范围
- $UMax = 2 * TMax + 1$

● C 语言编程

- `#include <limits.h>`
- 声明常数，例如，
 - `ULONG_MAX`
 - `LONG_MAX`
 - `LONG_MIN`
- 具体取值根据机器平台不同而不同

求补码的 四种方法

对字长为w的负数x求补码

●方法1

- $2^{w-1} - |x| = a$, 则x的补码为符号位1, 接上a的二进制表达。
- 例: -6的4位补码为: 符号位1, 接上 $2^3 - 6 = 2$ 的二进制表达010, 即1010;

●方法2

- $2^w - |x| = a$, 则a的二进制表达即为x的w位补码
- 例: -6的4位补码为 $2^4 - 6 = 10$ 的二进制表达 1010

求补码的 四种方法

对字长为 w 的负数 x 求补码

●方法3

- 写出 x 的 w 位原码;
- 符号位不变, 其余各位取反加1即得。
- 例: -6 的原码为 1110 , 首位符号位不变, 余下三位取反加1 得 1010 .

●方法4

- x 的绝对值的二进制, 所有位取反加1.
- 例: -6 的绝对值 6 的二进制为 0110 , 全部取反加1, 得 1010 .

单选题 1分



$w=8,$

(1) 求 -58 的补码;

(2) $(10010101)_T = (?)_{10}$

- ☐ A 11000101; -104.
- ☐ B 11100101; -110.
- ☐ C 11010011; -105.
- ☒ D 11000110; -107.

提交

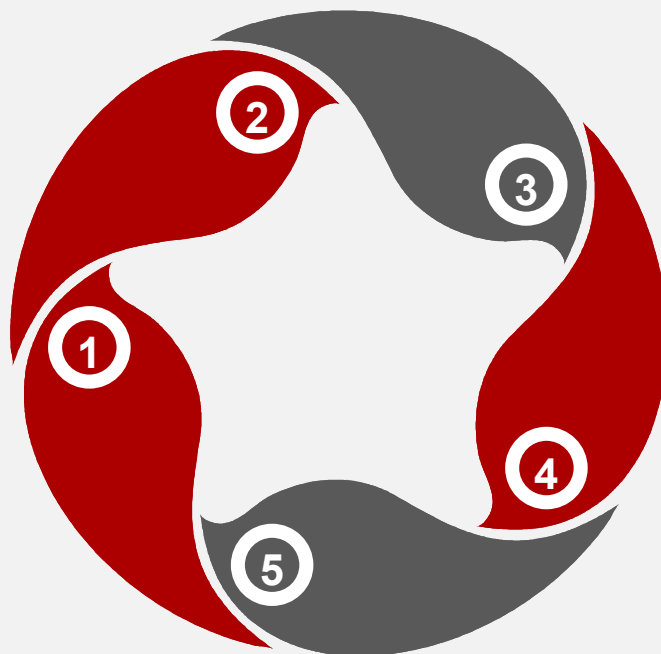
内容提要

转换 & 类型变换

扩展 & 截断

有符号数 & 无符号数

整数的运算操作



小结

数值比较

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

- **等价性**

- 非负数的补码就是其本身

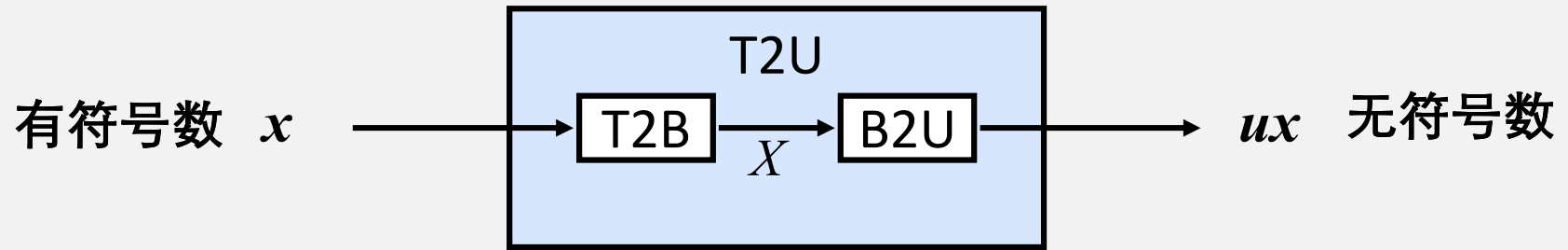
- **唯一性**

- 每一个“位模式”表达唯一的一个整数值
- 每一个整数有唯一的二进制“位模式”

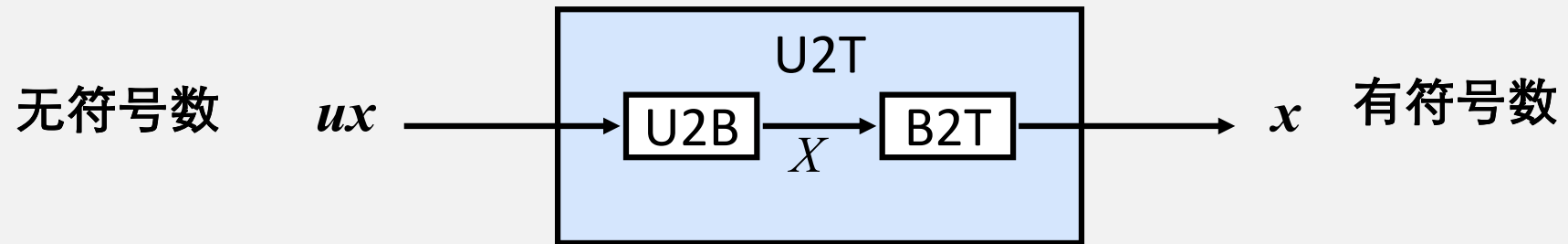
- **⇒ 可以逆向映射（双射）**

- $U2B(x) = B2U^{-1}(x)$
 - 无符号数的位模式
- $T2B(x) = B2T^{-1}(x)$
 - 补码的位模式

相互映射



Maintain Same Bit Pattern



Maintain Same Bit Pattern

有符号数与无符号数之间的映射就是：位模式不变，重新（按照转换后的类型）解释

相互映射

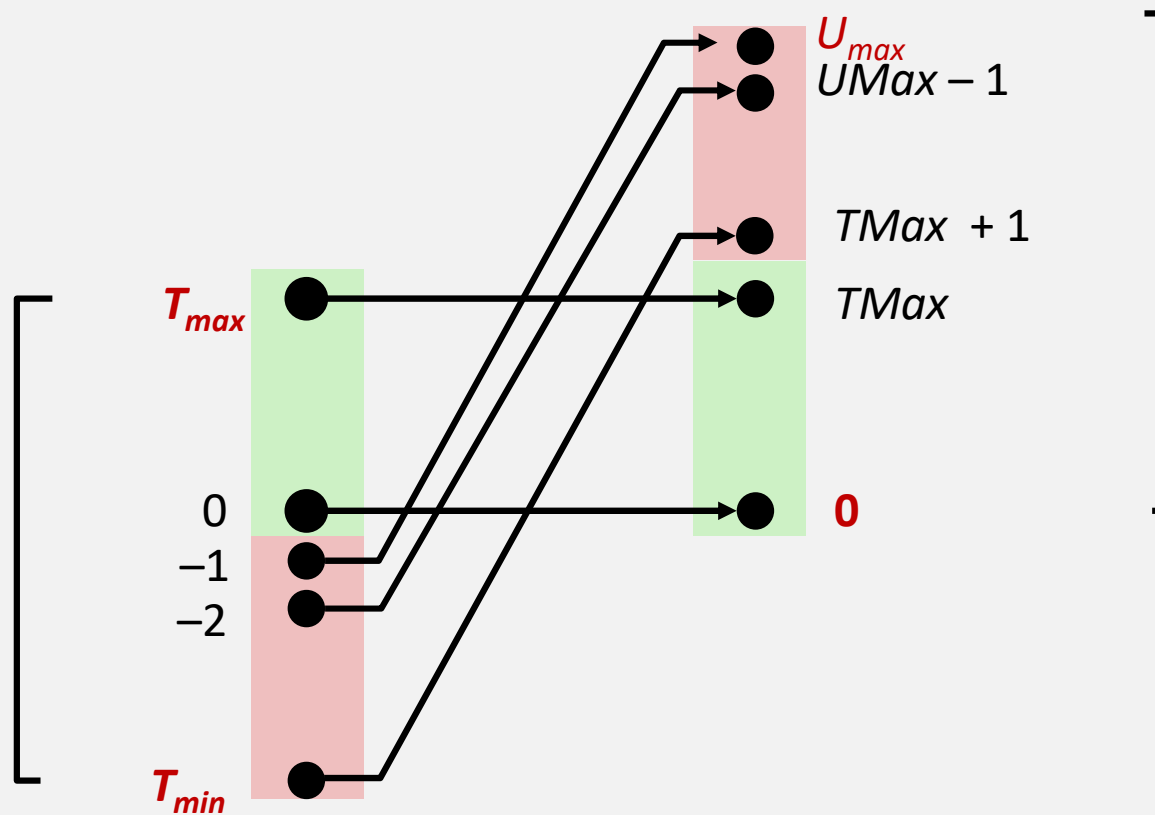
Bits	Signed		Unsigned
0000	0	→ T2U ← U2T ←	0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

相互映射

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

直观转换图

补码取值范围



无符号数
取值范围

C语言中的整数

- 常数

- 默认为有符号数
- 使用U作为无符号数的后缀: 0U, 4294967259U

- 转换

- 有符号数和无符号数的转换与 U2T 以及 T2U完全一样

```
int tx, ty;
```

```
unsigned ux, uy;
```

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

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

- 也可以通过赋值语句和过程调用实现

```
tx = ux;
```

```
uy = ty;
```

特殊情况

表达式

- 如果一个表达式中既有无符号数又有有符号数，*有符号数直接转换成无符号数*
- 上述表达式包括比较运算：< , > , == , <= , >=
- 例如 $w = 32$: $T_{\text{MIN}} = -2,147,483,648$, $T_{\text{MAX}} = 2,147,483,647$

常数1	常数2	关系	类型	结果
0	0U	==	unsigned	1
-1	0	<	signed	1
-1	0U	<	unsigned	0
2147483647	-2147483647-1	>	signed	1
2147483647U	-2147483647-1	>	unsigned	0
-1	-2	>	signed	1
(unsigned) -1	-2	>	unsigned	1
2147483647	2147483648U	<	unsigned	1
2147483647	(int) 2147483648U	<	signed	0

代码运行示例

例: 某32位机器, 考虑以下C代码:

```
1      int x = -1;
2      unsigned u = 2147483648;
3
4      printf ( "x = %u = %d\n" , x, x);
5      printf ( "u = %u = %d\n" , u, u);
```

在32位机器上运行上述代码时, 它的输出结果是什么? 为什么?

$x = 4294967295 = -1$

$u = 2147483648 = -2147483648$

- ♦ 因为-1的补码整数表示为“11...1”, 作为32位无符号数解释时, 其值为 $2^{32}-1=4\ 294\ 967\ 296-1=4\ 294\ 967\ 295$ 。
- ♦ 2^{31} 的无符号数表示为“100...0”, 被解释为32位带符号整数时, 其值为最小负数: $-2^{32-1} = -2^{31} = -2\ 147\ 483\ 648$ 。

有符号数小结

- 位模式不变
- 本质是用最高位为1的数去表示有互补关系的负数
- 重新解释
- 当加上或者减去 2^w 时会有意外情况
- 表达式中同时存在有符号数和无符号数时,
 - `int` 默认转换为 `unsigned`!!

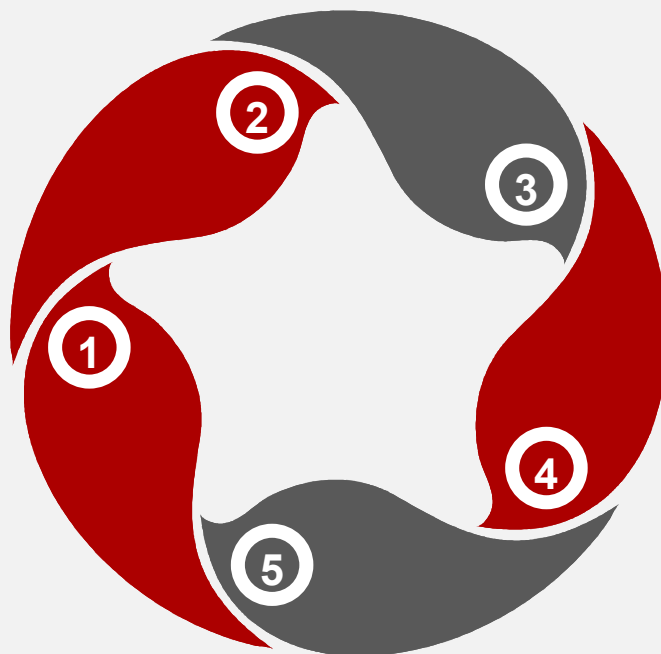
内容提要

转换 & 类型变换

扩展 & 截断

有符号数 & 无符号数

整数的运算操作



小结

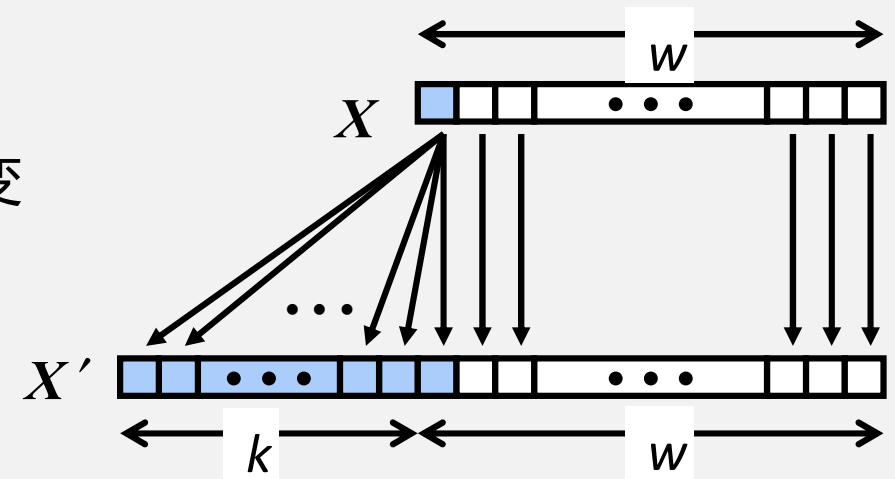
符号扩展

- **任务:**

- 给定一个 w 位的有符号数 x
- 将其转换为 $(w+k)$ -bit 的整数, 保持值不变

- **规则:**

- 将 X 符号位复制 k 个:
- $X = x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0$



符号扩展示例

```
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

将较小整型数据转换为较大整型数据

C语言中此类符号扩展自动完成（当发生数据类型变化时）

例如：-6 的补码为 1010 (w=4)，当 w=8, -6 的补码变为：

$2^8 - 6 = 250 = (\mathbf{1111}1010)_2$ ，仅发生了符号扩展。

符号扩展

- 为什么补码的扩展只需要做符号扩展，值保持不变？

- w 位二进制扩展成 $(w+k)$ 位，补码“表盘”范围变成了 2^{w+k} ，此时对负数 x 求补码变为： $2^{w+k} - |x|$
- 之前的 w 位补码为： $2^w - |x|$
- 增加了： $(2^{w+k} - |x|) - (2^w - |x|) = 2^{w+k} - 2^w = 2^w(2^k - 1)$

- $2^k - 1$ 的二进制就是 k 个1，乘以 2^w 就是左移 w 位，变成：

$$\underbrace{11\cdots\cdots 1}_{k\uparrow 1} \underbrace{00\cdots\cdots 0}_{w\uparrow 0}$$

符号扩展后
增加的值

- 把原本 w 位的 x 补码加上去，刚好填充 w 个0，变成 x 补码的符号扩展形式。

截断

- 截断的规则（例如从“无符号整型”变成“无符号短整型”）
 - 无符号/有符号：二进制位被截取
 - 截断结果意义与之前不同，需重新解释
 - 对无符号数来说就是取模运算 mod
 - 对有符号数来说类似取模运算
 - 较小数值的截断结果比较符合预期

截断示例

教材P52, 练习题2.24 假设将一个4位数值截断到3位数值。填写下表, 根据位模式的无符号和补码解释, 说明这些截断对某些情况的结果。

十六进制		无符号		补码	
原始值	截断值	原始值	截断值	原始值	截断值
0	0	0	0	0	0
2	2	2	2	2	2
9	1	9	1	-7	1
B	3	11	3	-5	3
F	7	15	7	-1	-1

无符号数截断, P52式(2-9): $B2U_k([x_{k-1}, x_{k-2}, \dots, x_0]) = B2U_w([x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k$

有符号数截断, P52式(2-10): $B2T_k([x_{k-1}, x_{k-2}, \dots, x_0]) = U2T_k(B2U_w([x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k)$

w=4, 那么13截断一位变成?

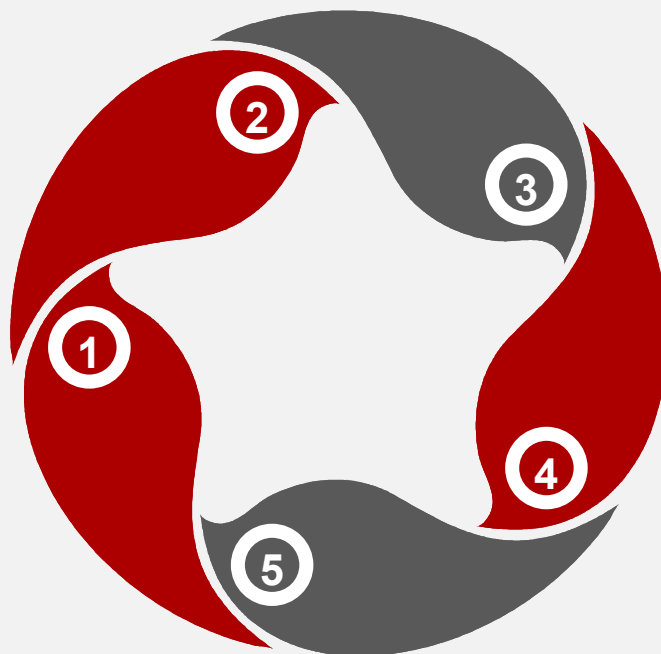
内容提要

转换 & 类型变换

扩展 & 截断

有符号数 & 无符号数

整数的运算操作



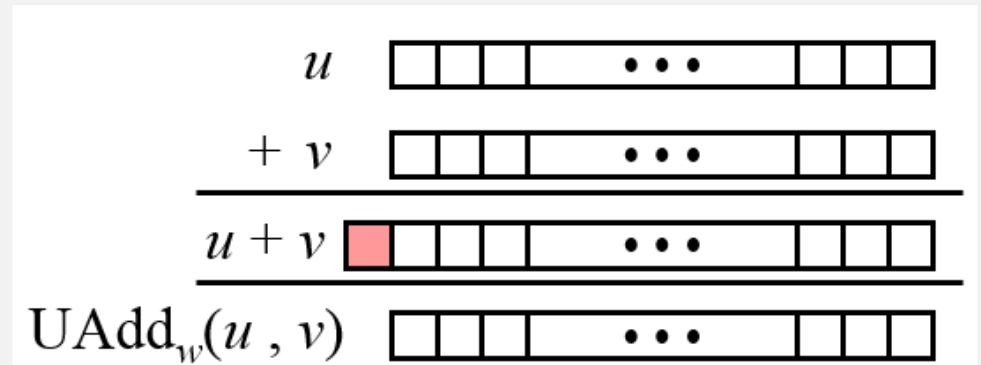
小结

无符号数加法 UAdd

操作数: w bits

实际结果: $w+1$ bits

舍弃最高进位: w bits



- 标准加法函数n

忽略最高进位

- 通过取模运算来实现

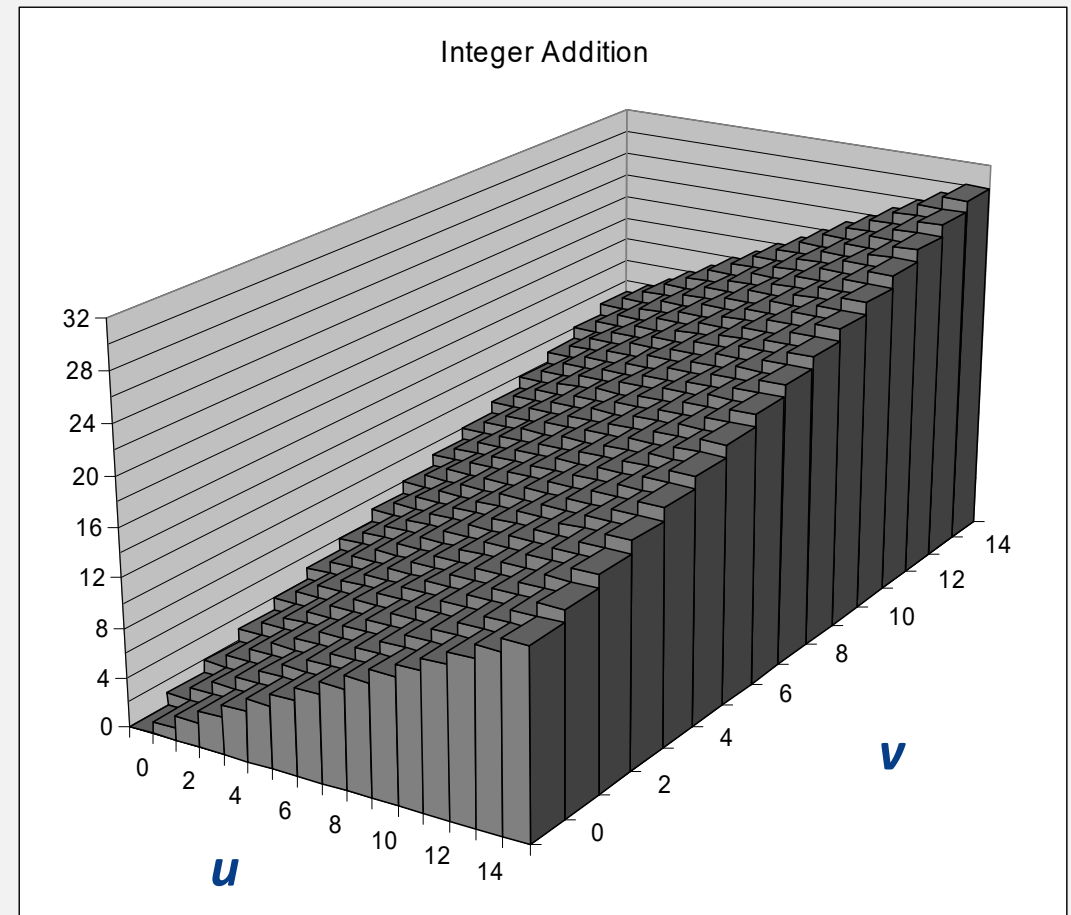
$$s = \text{UAdd}_w(u, v) = (u + v) \bmod 2^w$$



加法的可视化 UAdd

$Add_4(u, v)$

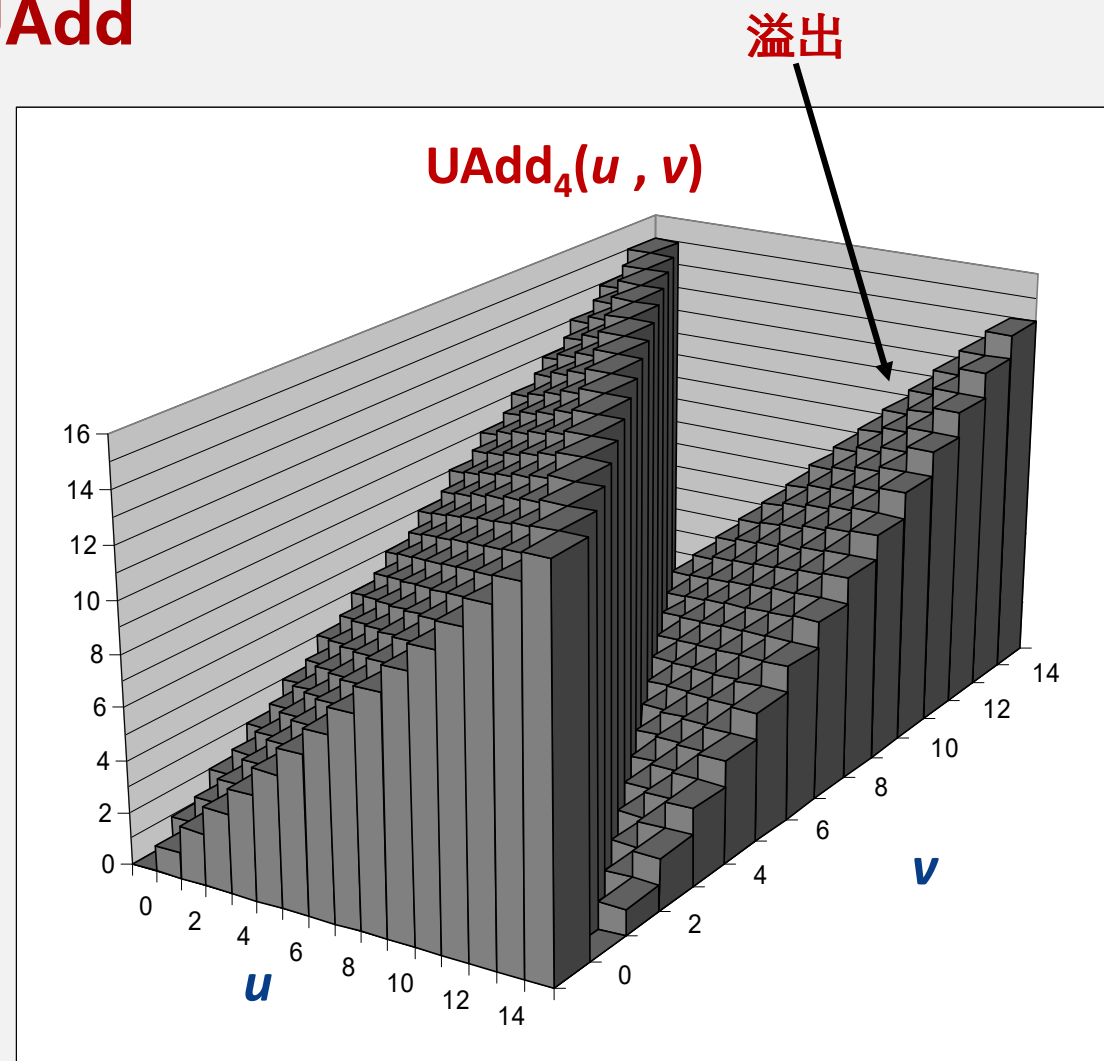
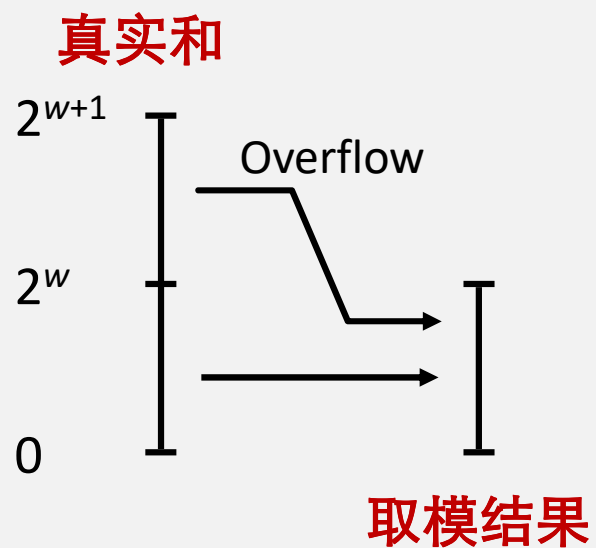
- 整数加法的真实和
 - 4-bit 整数 u, v
 - 计算真实和 $Add_4(u, v)$
 - 和随着 u 与 v 值增加而线性增加
 - 形成一个平坦表面



加法的可视化 UAdd

● 功能实现

- 若真实和 $\geq 2^w$
- 取模运算

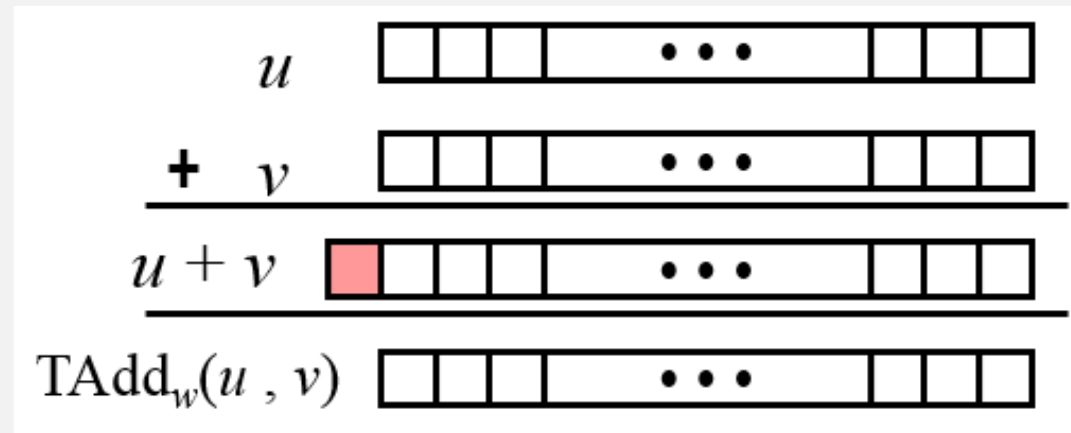


补码加法 TAdd

操作数: w bits

真实和: $w+1$ bits

舍弃进位: w bits



- UAdd与TAdd在“位”的操作上完全一致

- C 语言中的 UADD 与 TADD:

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

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

```
t = u + v
```

结果是 `s == t`

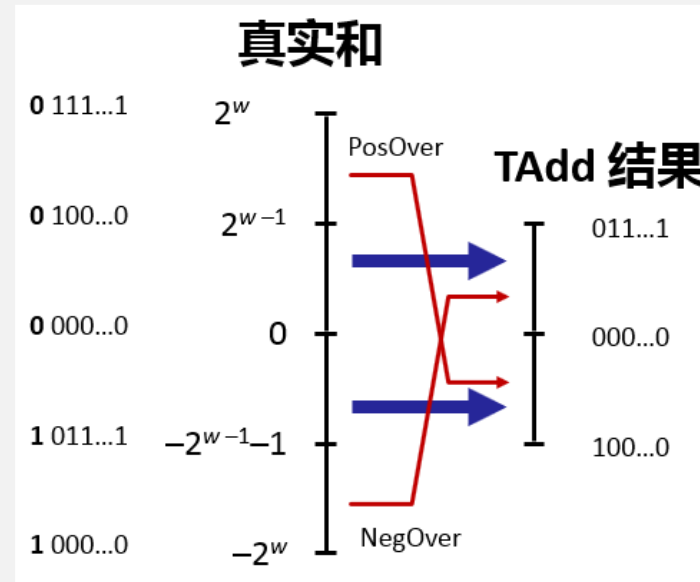
TAdd溢出

- 功能实现

- 真实和需要 $w+1$ bits
- 舍弃 MSB
- 将余下的位看做补码结果

- $x +_w^t y =$

$$U2T_w[(x + y) \bmod 2^w]$$

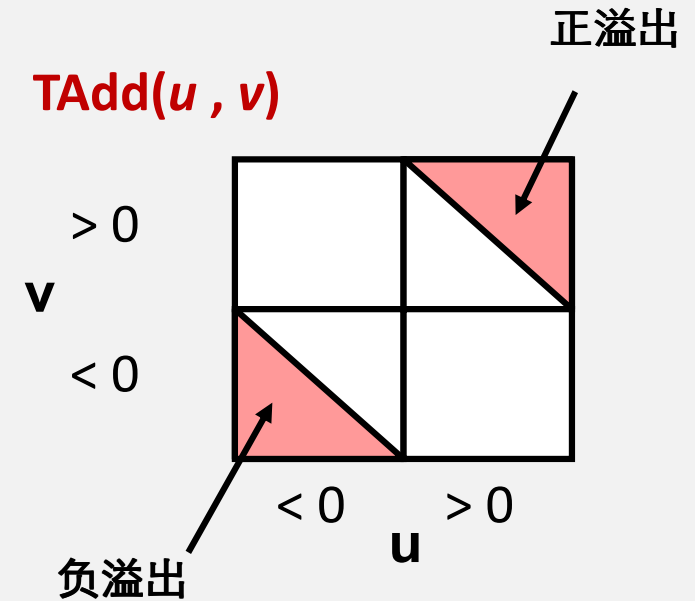


TAdd溢出

● 功能实现

- 真实和需要 $w+1$ bits
- 舍弃 MSB
- 将余下的位看做补码结果

$$TAdd_w(u, v) = \begin{cases} u + v + 2^w & u + v < TMin_w \\ u + v & TMin_w \leq u + v \leq TMax_w \\ u + v - 2^w & TMax_w < u + v \end{cases}$$



Tadd可视化

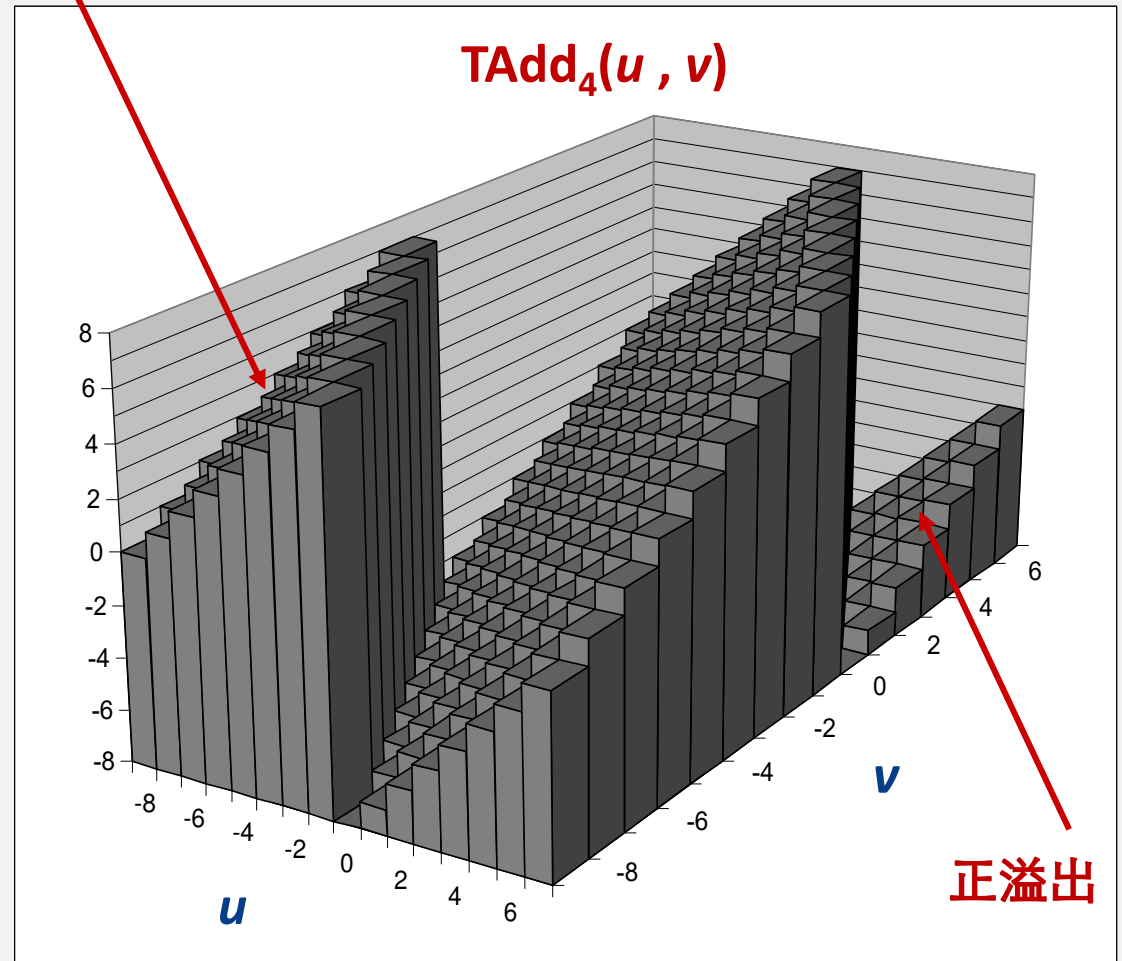
●取值

- 4-bit 补码.
- 取值从 -8 到 +7

●功能实现

- If $\text{sum} \geq 2^{w-1}$, 补码结果为负数
为**正溢出**, 导致结果减少16
例如: $0110(6)+0111(7)=1101(-3)$
最多一次
- If $\text{sum} < -2^{w-1}$, 补码结果为正数
为**负溢出**, 导致结果增加16
例如: $1010(-6)+1100(-4)=0110(6)$
最多一次

负溢出



有符号数减法

- 写出下列过程 ($w = 4$) :

- $5-3=2$
- 5 的补码是 0101
- -3的补码是 1101
- $5-3=5+(-3)=0101+1101=10010$, 实际结果为 **0010** (舍弃最高位)
- 0010 是 2 的补码, 因此结果为 2。

- 那么 $-5-1=?$

- -5的 补码为 1011; -1 的补码是 1111;
- $1011 + 1111 = 1010$ (舍弃最高位), 这是 -6 的补码, 因此结果为 -6。

乘法

计算 w 位的两个数 x 和 y 的乘积（有无符号均可）

- 注意：结果可能会多于 w bits

- 无符号数：最大可以达到 $2w$ bits: $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
- 补码负的乘积最大可以到 $2w-1$ bits: $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
- 补码正的乘积最大可以到 $2w$ bits, 仅对 $(TMin_w)^2$ 成立: $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$

- 所以为了得到完整的结果...

- 需要随着乘积的表达扩展字长
- 如有必要，可以使用软件完成（例如，使用 “arbitrary precision” 计算功能）

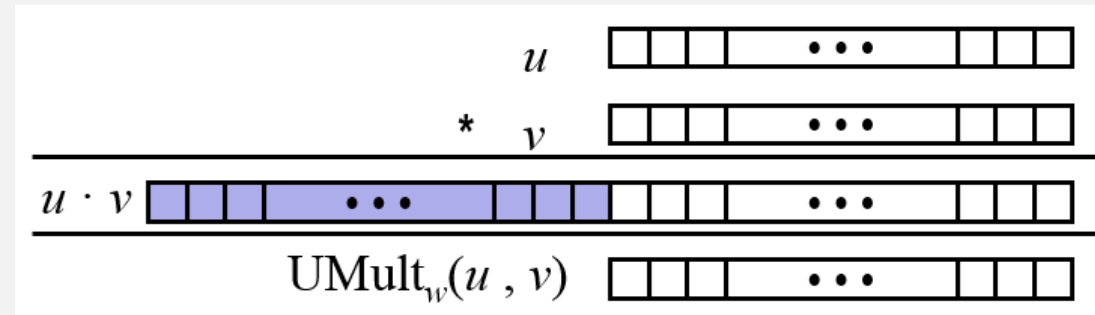
C语言的乘法

无符号数乘法

操作数: w bits

真实乘积: $2 * w$ bits

舍弃 w bits, 得到 w bits



- 标准乘法函数

- 忽略乘积的高 w 位

- 采用取模运算

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

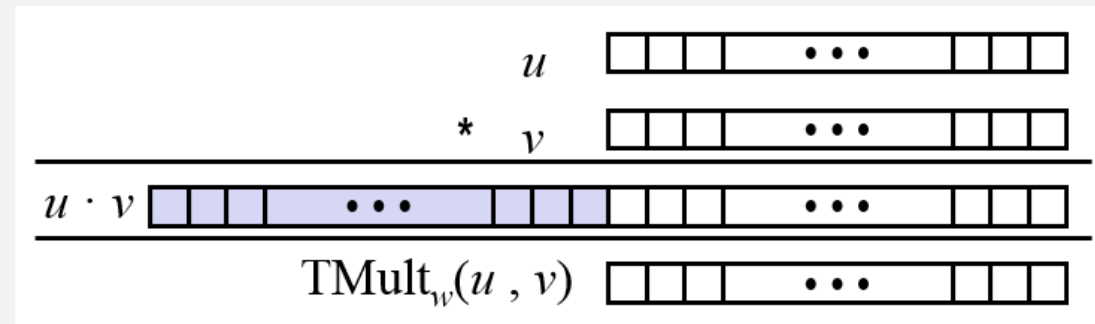
C语言的乘法

有符号数乘法

操作数: w bits

真实乘积: $2 * w$ bits

舍弃 w bits, 得到 w bits



- 标准乘法函数

- 忽略乘积的高 w 位

- 采用取模运算

- $\text{TMult}_w(u, v) = U2T_w[(u \cdot v) \bmod 2^w]$

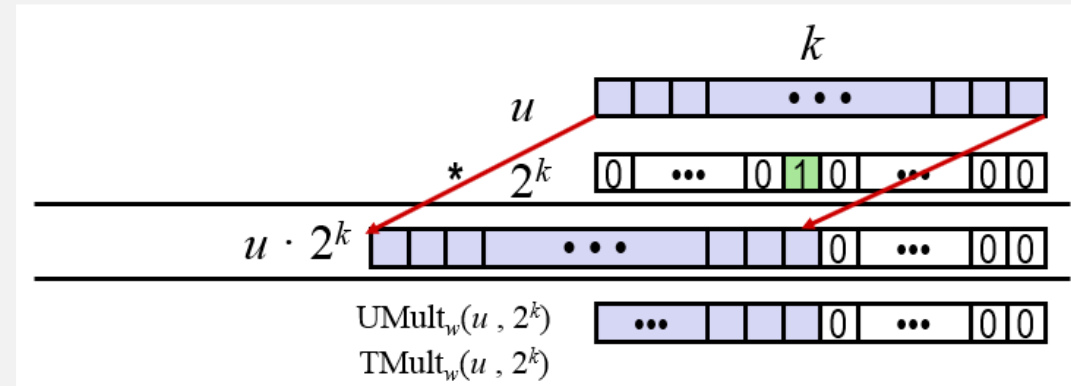
2的幂次

- 操作

- $u \ll k$ 得到 $u * 2^k$
- 对有/无符号数均适用

- 举例

- $u \ll 3 \quad == \quad u * 8$
- $u \ll 5 - u \ll 3 \quad == \quad u * 24$
- 在大部分机器中移位和加法运算比乘法快得多
- 因此编译器总是自动将符合移位操作要求的乘法变成移位运算



编译后的 乘法指令

C Function

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

Compiled Arithmetic Operations

```
leal    (%eax,%eax,2), %eax
sall    $2, %eax
```

Explanation

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

- 与常数相乘时，C 编译器总是**自动生成移位和加法指令**以简化机器的运算。

除法

- 除数为**无符号数**中 2 的幂次时，商的计算如下：

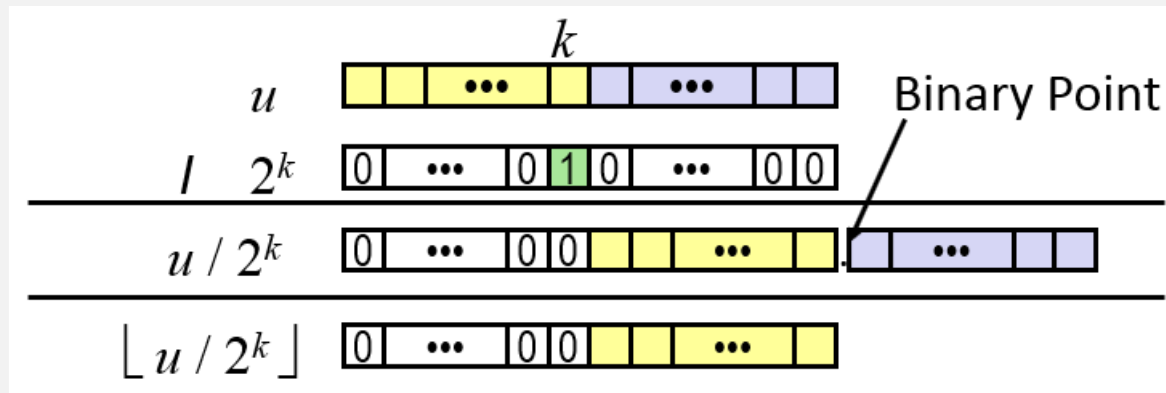
- $u \gg k$ gives $\lfloor u / 2^k \rfloor$

- 使用**逻辑右移**

操作数:

除法:

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

除法

- 除数为**有符号数**中 2 的幂次时，商的计算如下：

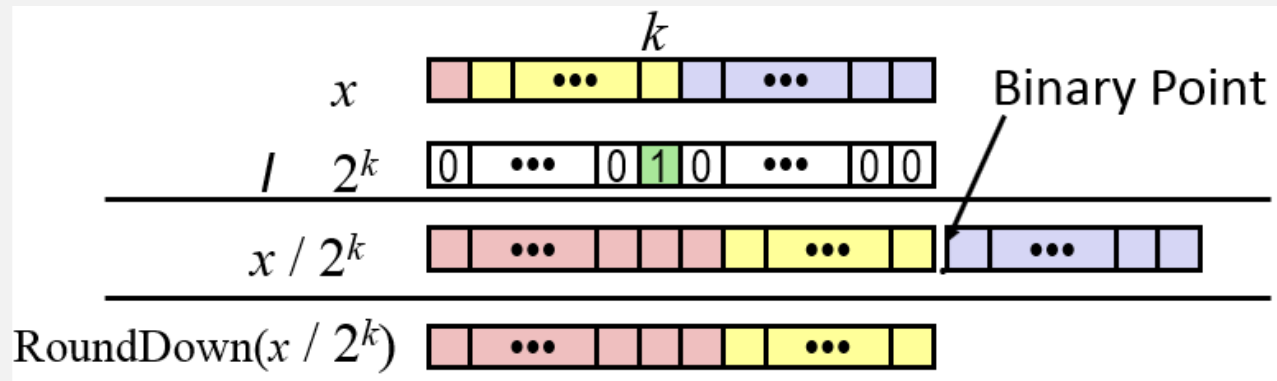
- $x \gg k$ gives $\lfloor x / 2^k \rfloor$

- 使用**算术右移**

操作数:

除法:

结果:



	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

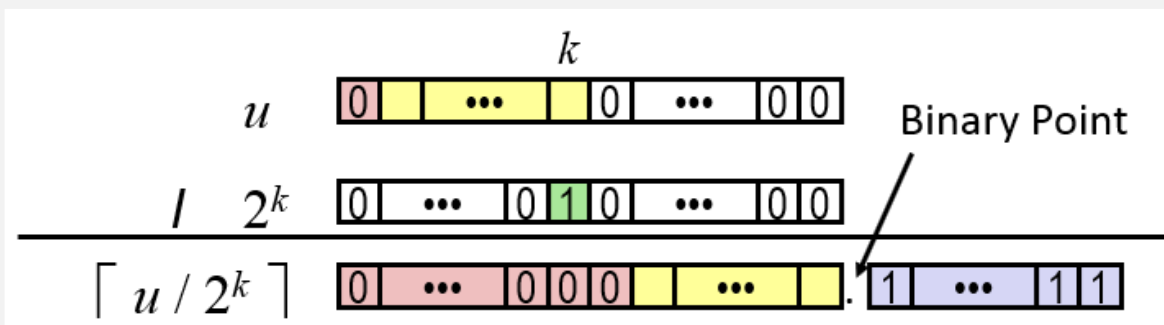
除法的矫正

- **被除数为负，除数为 2 的幂次**
 - 需要 $\lceil x / 2^k \rceil$ (向 0 取整)
 - 计算: $\lfloor (x+2^k-1) / 2^k \rfloor$
 - 在 C 中: $(x + (1 \ll k) - 1) \gg k$
 - 将被除数向 0 偏置

情况 1: 无需近似

被除数

除数

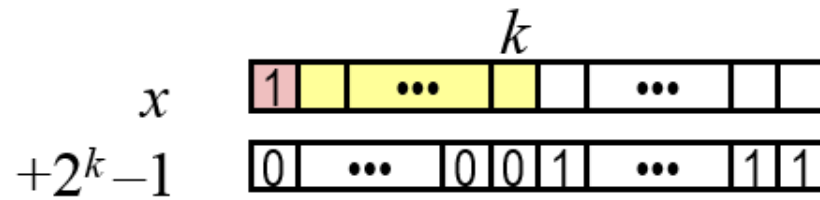


偏置无效果

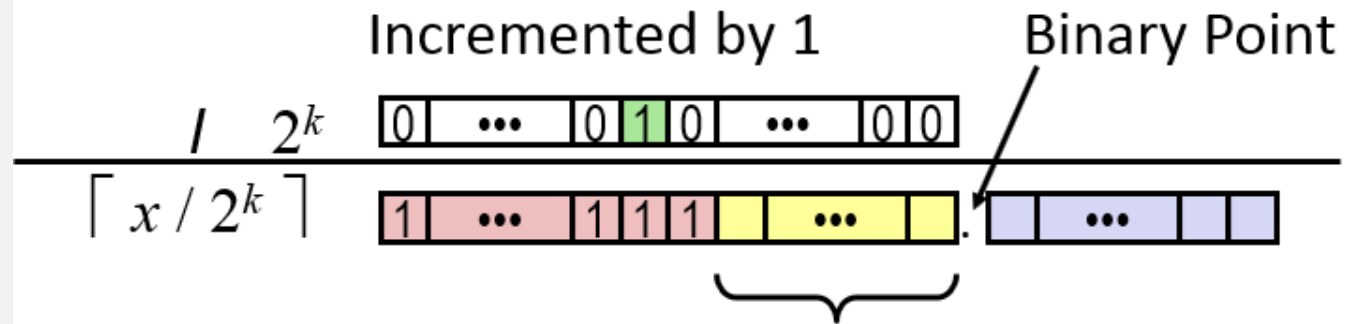
除法的矫正

情况 2: 需要近似

被除数



除数



因为偏置结果加一

除法指令

C Function

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

汇编代码

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

- 对整数采用算术移位操作
- 对 Java 用户，算术移位写作：>>

含义

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

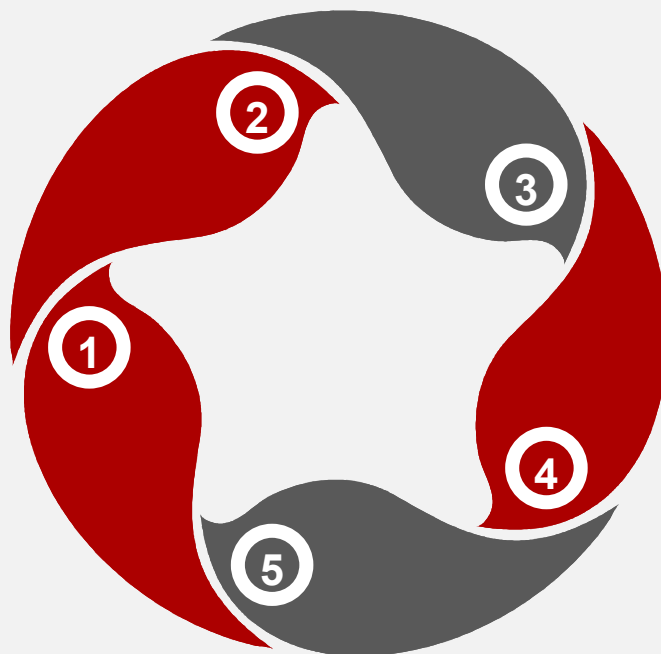
内容提要

转换 & 类型变换

扩展 & 截断

有符号数 & 无符号数

整数的运算操作



小结

运算规则

● 加法

- 有/无符号数: 正常相加后截断（舍弃最高位），两种加法在位操作上是完全一样
- 无符号数: 和对 2^w 模 ($\text{mod } 2^w$)
 - 算术加法+ 对 2^w 的减法（如果需要）
- 有符号数: 修正的和对 2^w 取模（确保结果在正确范围）
 - 算术加法+对 2^w 的加减法（如果需要）

● 乘法

- 有/无符号数: 正常相乘后截断（舍弃高 w 位），两种乘法在位操作上是完全一样
- 无符号数: 乘积 $\text{mod } 2^w$
- 有符号数: 修正后乘积 $\text{mod } 2^w$ (确保结果在正确范围)

何时使用 无符号数?

- 不要因为仅仅因为需要非负数而使用

- 极易犯错

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

- 会非常微妙敏感

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

- 取模运算时使用

- 多精度运算

- 使用位来表示集合时

- 逻辑右移, 没有符号扩展

随处可见的坑

```
1.  #include <stdio.h>
2.
3.  int main()
4.  {
5.      if(sizeof(int) - sizeof(double) < 0)
6.      {
7.          printf("<\n");
8.      }
9.      else
10.     {
11.         printf(">=\n");
12.     }
13.
14.     return 0;
15. }
```

求以上程序的输出结果，我们知道`sizeof(int) = 4`，`sizeof(double) = 8`，则 $4 - 8 = -4$ ，则`if(sizeof(int) - sizeof(double) < 0)`的条件为真，所以，输出应该是打印出“<”，程序运行的结果如下：



点击[此处](#)折叠或打开

1. >=

发现和预期的结果相反，究其原因，原来`sizeof()`的返回值是一个无符号整型十进制数值，即：`unsigned int`型，所以两个`unsigned int`型的数据进行运算后其结果只能是`unsigned int`型的，即大于等于0的一个整数值，所以，`if()`条件不满足，执行下一条语句。这个错误比较隐蔽，很容易出错，今天遇到了在此做一个记录，以便以后的复习。

又一个坑!

预测一下这道题的结果，注意unsigned与signed

```
[cpp]    
1. #include<iostream>  
2. #include<stdio.h>  
3. using namespace std;  
4. int arr[]={1,2,3,4,5};  
5. int main(){  
6.     for(int i=-1;i<sizeof(arr)/sizeof(arr[0]) - 1;i++)  
7.         cout<<arr[i+1]<<endl;  
8. }
```

sizeof()返回的值是无符号数，有符号数遇到无符号数变成无符号数，所以在`i<sizeof(arr)/sizeof(arr[0])`运算中`i=-1`变成无符号数，那么`i`将变成一个非常大的数，所以这个程序没有输出

以后编程千万小心，给自己挖的坑少点、再少点!

下一节：浮点数

湖南大学

《计算机系统》课程教学组

