2 信息的表示和处理

吴海军

南京大学计算机科学与技术系

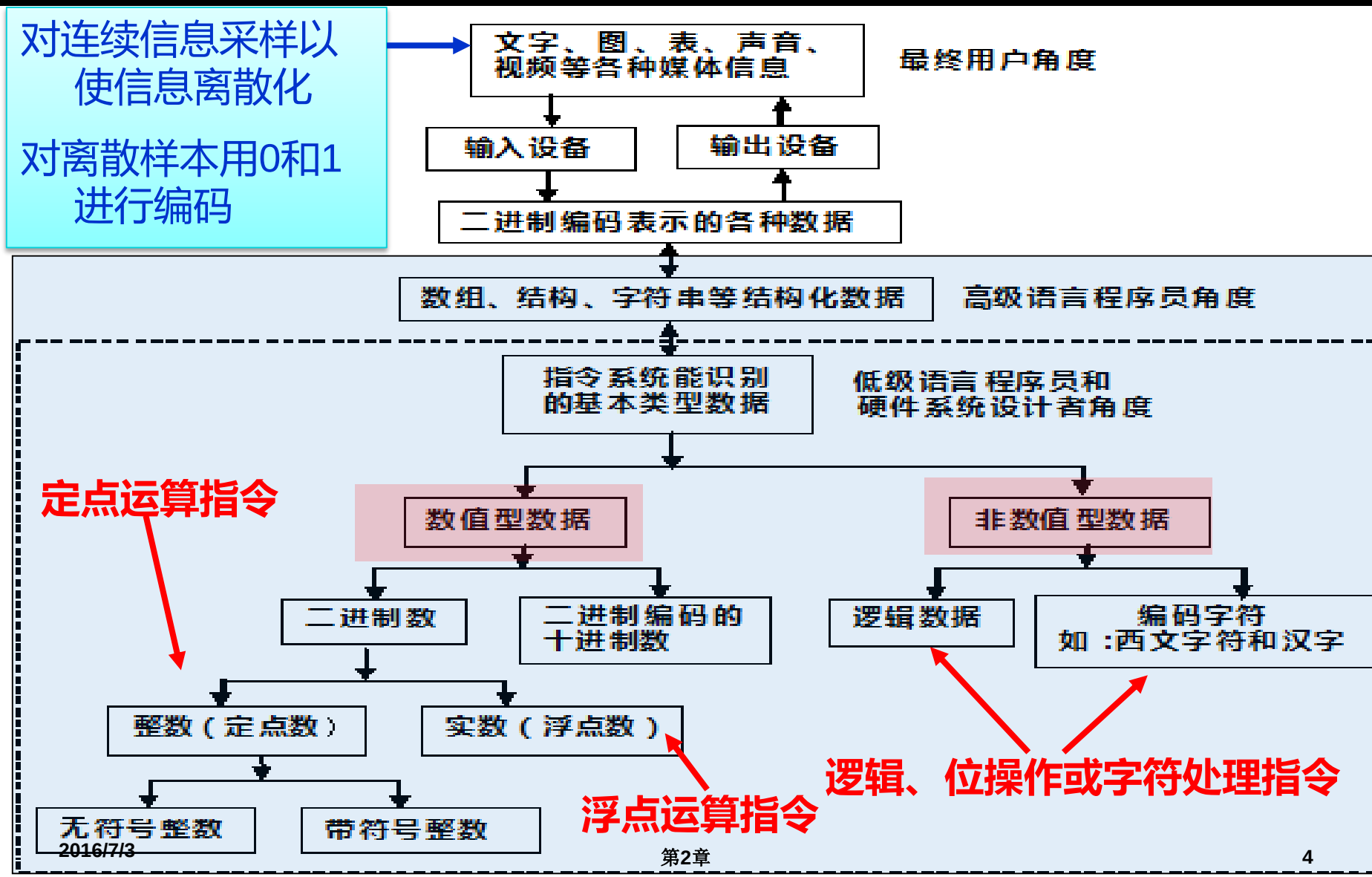
数据的表示和运算

- 信息的表示
  - 数值数据的表示、非数值数据的表示
- 数据存储
  - 数据宽度单位、排列次序、边界对齐
- 位运算
  - 按位运算\逻辑运算\移位运算、位扩展和位截断运算
- 数据运算
  - 无符号和带符号整数的加减运算、乘除运算
  - 变量与常数之间的乘除运算
- 浮点数的表示及运算
  - 浮点数的加减运算

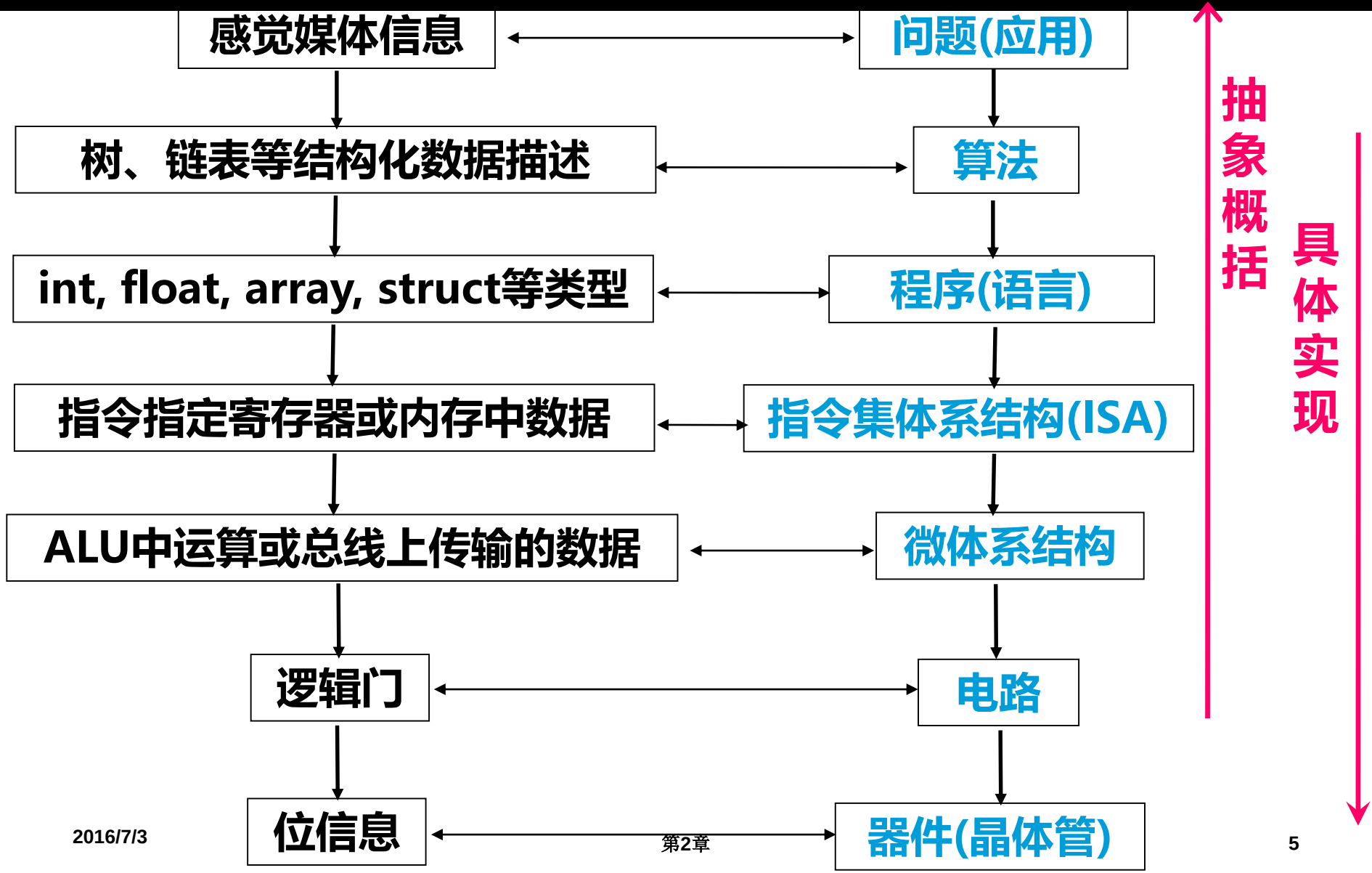
信息的表示

对连续信息采样以使信息离散化

对离散样本用0和1进行编码



数据在不同层次中的表示



信息的表示

- 数值数据的表示
- 三种最重要的数字表示
  - 无符号编码表示：大于等于0的整数，非数值信息
  - 补码表示：带符号的整数
  - 浮点数表示：实数的科学计数法
- 非数值数据的表示

数值数据的表示

- 数值数据表示的三要素：进位记数制、定/浮点表示、编码格式

  要确定一个数值数据的值必须先确定这三个要素。

  例如，机器数 1011001的值是多少？ **答案是：不知道！**

  - **进位记数制**（解决基数问题）
    - 十进制、二进制、十六进制、八进制数及其相互转换
  - **定/浮点数表示**（解决小数点问题）
    - 定点整数、定点小数
    - 浮点数（可用一个定点小数和一个定点整数来表示）
  - **定点数的编码**（解决正负号问题）
    - 原码、补码、反码、移码 （反码很少用）

定点数和浮点数

- 日常生活中所使用的数有**整数**和**实数**之分。
  - 整数的小数点固定在数的最右边，可省略不写。
  - 实数的小数点则不固定。
- **定点数：**小数点位置约定在某一个固定位置的数。
  - 定点整数是**纯整数**，约定的小数点位置在有效数值部分最低位之后。
  - **定点小数**是**纯小数**，约定的小数点位置在有效数值部分最高位之前。
- **浮点数：**小数点位置约定可以浮动的数。

定点/浮点数解决小数点的问题

3、带符号数的表示及运算

- 正负数如何表示？
- 用n位R进制数表示R^n个不同的正负整数值。
- 符号-数值表示法
  - 原码
  - 补码
  - 反码
- 二进制数加减运算

如何用3位二进制数表示8个不同正负整数？

000
001
010
011
100
101
110
111

原码：符号-数值表示法

- 原码：一个数是由表示该数为正或者负的**符号位**和**数值**两部分组成。正数符号位为0，负数符号位为1.
  - 增加1位符号位：
  - [+43]的8位原码为：00101011
  - [-43]的8位原码为：10101011

容易理解，　但是加、减运算方式不统一，需额外对符号位进行处理，不利于硬件设计。

- 特点：
  - 正整数和负整数数值相同，符号位不同，0有两种。
  - n 个二进位的原码可表示的数值范围是：$-2^{n-1}+1 \sim 2^{n-1}-1$

浮点数的尾数用原码定点小数表示

补码

- 基数补码表示法（同余数、模）：
  1. 基数为R的n位数的补码等于从R^n中减去该数。

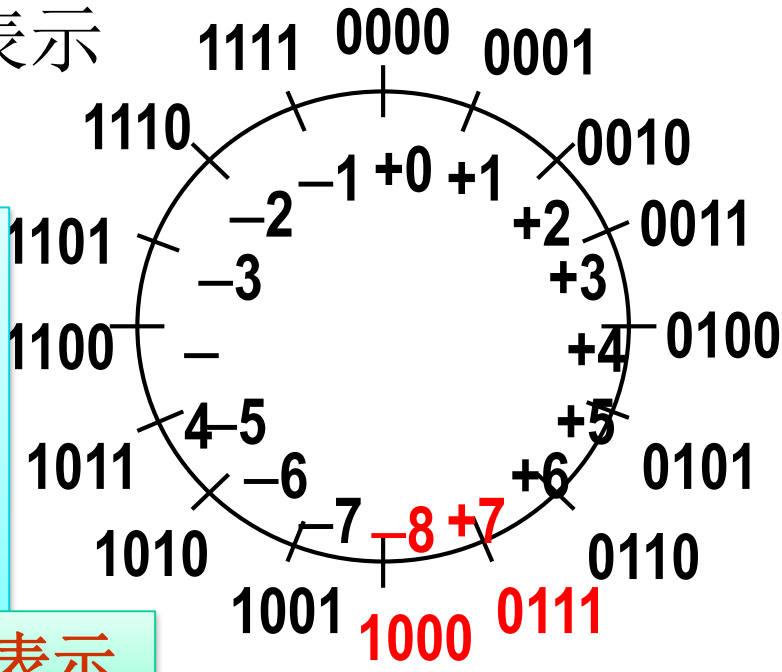
     $D_{补}=R^n-D$
  2. **按位取反加一。**
     - $R^n-D=((R^n-1)-D)+1$; **$(R^n-1)-D$表示反码**，R-1-d/位
- 一个数的补码的补码保持不变。
- 用补码表示带符号的数值，最高位MSB的权$-R^{n-1}$
- $D+D_{补}=R^n$，　　$0 \le D \le R^{n-1}-1$
  - 因为只有n位，最高位溢出，不能保存，则结果为0
  - $D+D_{补}=D-D=0$
  - $D_{补}=-D$

二进制补码表示

- 二进制补码的性质：

$$D_{补} = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

  - 0是唯一表示的(用全0表示数值“0”)
  - 不对称，负整数比正整数多1个（ -2^{n-1} ）
  - n 个二进位的补码可表示的数值范围是：$-2^{n-1} \sim 2^{n-1}-1$
- 正数的补码由符号位0+数值表示
- 负数的补码有两种计算方法：



结论1： 一个负数的补码等于模减该负数的绝对值或反码加1。
结论2： 补码符号位直接参与运算
结论3：补码加减法统一，减去一个数等于加上这个数相反数的补码。

现代计算机系统中整数都采用补码来表示

负二进制数补码的计算方法

第一种，一个N位的负二进制数，其补码 = 2^N-二进制数的绝对值

例如：-99的8位二进制补码表示为： $99=0110\ 0011_2$

$$
\begin{array}{rl}
 & 100000000 \\
- & 01100011 \\
\hline
\text{补码} & 10011101
\end{array}
$$

第二种，按位取反加一

$(-99)_{补}=10011101_2$

二进制数 01100011

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

$$
\begin{array}{rl}
\text{反码} & 10011100 \\
+ & 1 \\
\hline
\text{补码} & 10011101
\end{array}
$$

求特殊数的补码

假定机器数有n位：

① $[-2^{n-1}]_{补} = 2^n - 2^{n-1} = 10\ldots0$（n-1个0） （mod 2^n）

② $[-1]_{补} = 2^n - 0\ldots01 = 11\ldots1$（n个1） （mod 2^n）

③ $[+0]_{补} = [-0]_{补} = 00\ldots0$（n个0）

- 为什么用补码表示带符号整数？
  - 补码运算系统是模运算系统，加、减运算统一
  - 数0的表示唯一，方便使用
  - 比原码和反码多表示一个最小负数

运算器只有有限位，假设为n位，则运算结果只能保留低n位，故可看成是个只有n档的二进制算盘，因此，其模为2^n。

二进制数反码的表示法

- 二进制数的反码　1’s Complements

二进制数 **01101010**

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

反码 **10010101**

逐位取反

- 反码具有对称性，但0有两种表示。
- n 个二进位的补码可表示的数值范围是：$-2^{n-1}+1 \sim 2^{n-1}-1$
- 正数的反码由符号位0+数值表示
- 负数的反码由符号位1+数值位按位取反表示

无符号数、带符号数

编码	无符号	原码	补码	反码	编码	无符号	原码	补码	反码
0000	0	0	0	0	1000	8	-0	-8	-7
0001	1	1	1	1	1001	9	-1	-7	-6
0010	2	2	2	2	1010	A	-2	-6	-5
0011	3	3	3	3	1011	B	-3	-5	-4
0100	4	4	4	4	1100	C	-4	-4	-3
0101	5	5	5	5	1101	D	-5	-3	-2
0110	6	6	6	6	1110	E	-6	-2	-1
0111	7	7	7	7	1111	F	-7	-1	-0

在带符号数编码中最高位为符号位

无符号整数

- 总是大于0的整数，简称为“无符号数”
- 一般在全部是正数运算且不出现负值结果的场合下，可使用无符号数表示。例如：地址运算，编号表示，等等
- 无符号整数的编码中没有符号位
- 能表示的最大值大于位数相同的带符号整数的最大值
  - 例如，8位无符号整数最大是255（1111 1111）

    8位带符号整数最大为127（0111 1111）

C语言程序中的整数

- 无符号数：unsigned int /short / long 常在一个数的后面加一个“u”或“U”表示无符号数
- 若同时有无符号和带符号整数，则C编译器将带符号整数强制转换为无符号数

假定以下关系表达式在32位用补码表示的机器上执行，结果是什么？

关系表达式	类型	结果	说明
0 == 0U	无符号	1	00…0B = 00…0B
-1 < 0	带符号	1	11…1B (-1) < 00…0B (0)
-1 < 0U	无符号	0*	11…1B (2^{32}-1) > 00…0B(0)
2147483647 > -2147483647-1	带符号	1	011…1B (2^{31}-1) > 100…0B (-2^{31})
2147483647U > -2147483647-1	无符号	0*	011…1B (2^{31}-1) < 100…0B(2^{31})
2147483647 > (int) 2147483648U	带符号	1*	011…1B (2^{31}-1) > 100…0B (-2^{31})
-1 > -2	带符号	1	11…1B (-1) > 11…10B (-2)
(unsigned) -1 > -2	无符号	1	11…1B (2^{32}-1) > 11…10B (2^{32}-2)

C语言程序中的整数

1）在有些32位系统上，C表达式-2147483648 < 2147483647的执行结果为false。Why？

2）若定义变量“int i=-2147483648;”，则“i < 2147483647”的执行结果为true。Why？

3）如果将表达式写成“-2147483647-1 < 2147483647”，则结果会怎样呢？Why？

1）在ISO C90标准下，2147483648为unsigned int型，因此“-2147483648 < 2147483647”按无符号数比较，10……0B比01……1大，结果为false。

在ISO C99标准下，2147483648为long long型，因此“-2147483648 < 2147483647”按带符号整数比较，10……0B比01……1小，结果为true。

由C语言中的“Integer Promotion”规则决定的。

2）i < 2147483647 按int型数比较，结果为true。

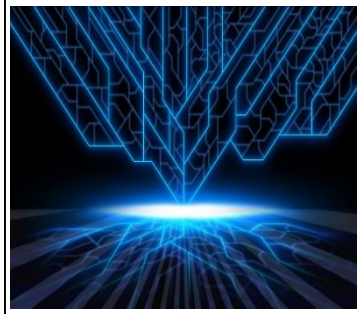
3）-2147483647-1 < 2147483647 按int型比较，结果为true。

非数值数据的表示

逻辑值

字符编码

状态编码



编码

- 用于表示一个数或信息的一组二进制数位的集合，称为二进制**编码**。
  - 用于存储、传输、控制、执行等处理；
  - 和具体应用密切相关，无通用处理逻辑。
- 一个含义确切的特定的二进制数位组合称为**码字**。
  - 码字之间可以有算术关系，也可以没有。
- 编码举例：
  - 逻辑量编码
  - 字符编码：英文、中文、Unicode等
  - 特殊的编码等。

逻辑数据的编码表示

- 表示：用一位二进制数表示。例如，真：1 、假：0
  - N位二进制数可表示N个逻辑数据，或一个位串
- 运算：按位进行
  - 如:按位与 / 按位或 / 逻辑左移 / 逻辑右移 等
- 识别
  - 逻辑数据和数值数据在形式上并无差别，根据指令来识别。
- 位串
  - 用来表示若干个状态位或控制位（OS中使用较多）

例如，x86的标志寄存器含义如下：

				OF	DF	IF	TF	SF	ZF		AF		PF		CF

ASCII字符编码

ASCII（美国标准信息交换码）字符表

低位 \ 高位		0	1	2	3	4	5	6	7
		000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P	、	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	↑	n	~
F	1111	SI	US	/	?	O	←	o	DEL

NJU：01001110 01001010 01010101

汉字编码

- 汉字内码：汉字标准信息交换码，国家先后制定了如**GB2312**（6763 个汉字），**GB18030**（27484个汉字），**GBK，Unicode等标准**
  - 汉字区位码，在编码标准中区位号
  - 汉字国标码，区别ASCII码的控制符编码，=区位码+2020H
  - 汉字机内码，区别ASCII码，每个字节的最高位强制为1，=国标码+8080H
- 外码
  - 汉字的各种输入编码

Unicode编码

- Unicode 采用两个字节编码体系，可以表示65536个字符
  - 前128个Unicode字符是标准的ASCII字符
  - 接下来 的128个扩展的ASCII字符，
  - 内容包括字母和符号10236个、汉字 27786个、韩文拼音11172个、造字区6400个、保留20249个、控制符65个。
- UNICODE 最显著特点在于：UNICODE是两字节的全编码
- 简化了汉字的处理过程。

信息存储

信息处理的单位

- 比特（bit）是计算机中处理信息的最小单位。
  - 字节(Byte)， 1B=8bits
  - 字（Word)，1W=2B=16bits
  - 千字节(KB)，1KB=2^{10}字节=1024B
  - 兆字节(MB)，1MB=2^{20}字节=1024KB
  - 吉字节 (GB)，1GB=2^{30}字节=1024MB
  - 太字节 (TB)，1TB=2^{40}字节=1024GB
  - 拍字节 (PB)，1PB=2^{50}字节=1024TB
- 存储二进制信息时的度量单位要比字节或字大得多

信息存储

- 二进制信息存储的计量单位是“字节”(Byte)，也称“位组”
  - 现代计算机中，存储器按字节编址
  - 字节是最小可寻址单位 (*addressable* unit)
- 所有的可能寻址的集合称为寻址空间。
- 系统中最大可寻址空间取决于**地址总线**的宽度。
- IA-32最大可寻址空间2^{32}B=4GB。

数据的宽度

- “字长”的含义：指**数据总线**的宽度。
  - “字长”等于CPU内部总线的宽度、运算器的位数、通用寄存器的宽度等。
  - **字长取决于系统的体系结构。**X86-16/IA-32/X86-64各不相同。
  - 数据总线指CPU内部数据流经的路径以及路径上的部件，主要是CPU内部进行数据运算、存储和传送的部件，这些部件的数据宽度要一致，才能相互匹配。

C语言中数据类型的宽度

C语言中数值数据类型的宽度 (单位：字节)

C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

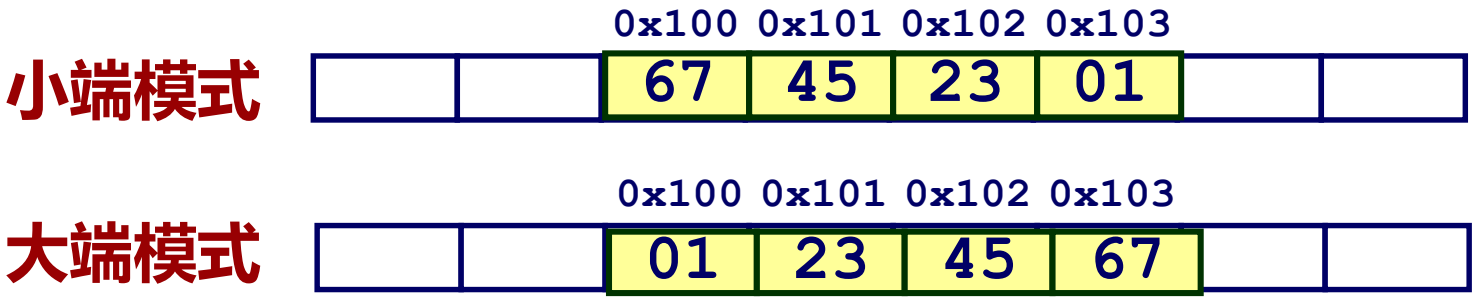
- C语言中char类型的宽度为1个字节，可表示一个字符（非数值数据），也可表示一个8位的整数（数值数据）
- 不同体系架构计算机表示的同一种类型的数据宽度可能不相同
- 分配的字节数随机器字长和编译器的不同而不同。

必须确定相应的机器级数据表示方式和相应的处理指令

数据的存储和排列顺序

- 多字节数据在存储器中如何**按序存取**？

例如：给定 x= 0x01234567，存放地址从 0x100开始，则数据存放格式有：



- 小端模式Little **Endian**：最低有效字节**LSB**存放在最小地址位的方式。如：**Intel 80x86, DEC VAX**
- 大端模式**Big Endian**：最高有效字节**MSB**存放在最小地址位的方式。 如：**IBM 360/370, SUN, MIPS, Sparc, HP PA,Internet**

小端模式示例

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

Result (Linux):

```
int a = 15213;
0x11ffffcb8 0x6d
0x11ffffcb9 0x3b
0x11ffffcba 0x00
0x11ffffcbb 0x00
```

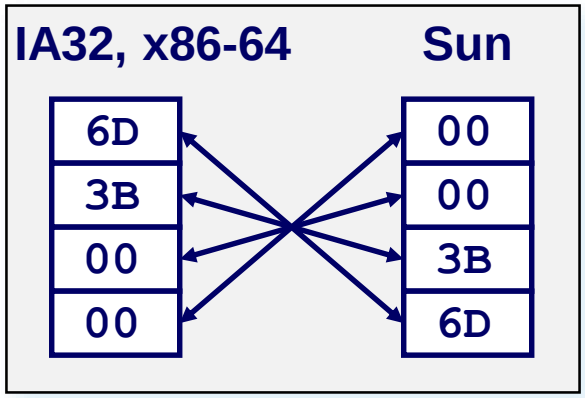
Decimal: 15213

Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

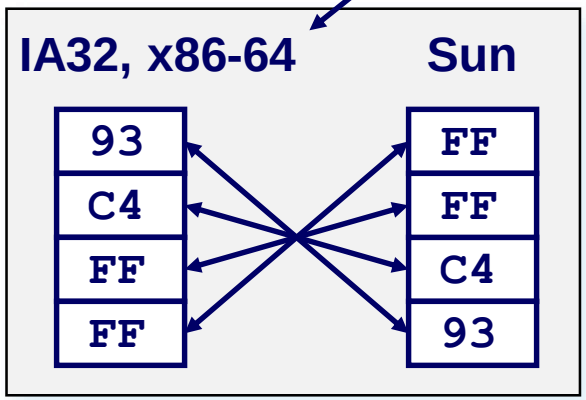
大端、小端转换

int A = 15213;



负数采用补码表示

int B = -15213;



存放方式不同的机器间程序移植或数据通信时，数据需要进行字节交换。

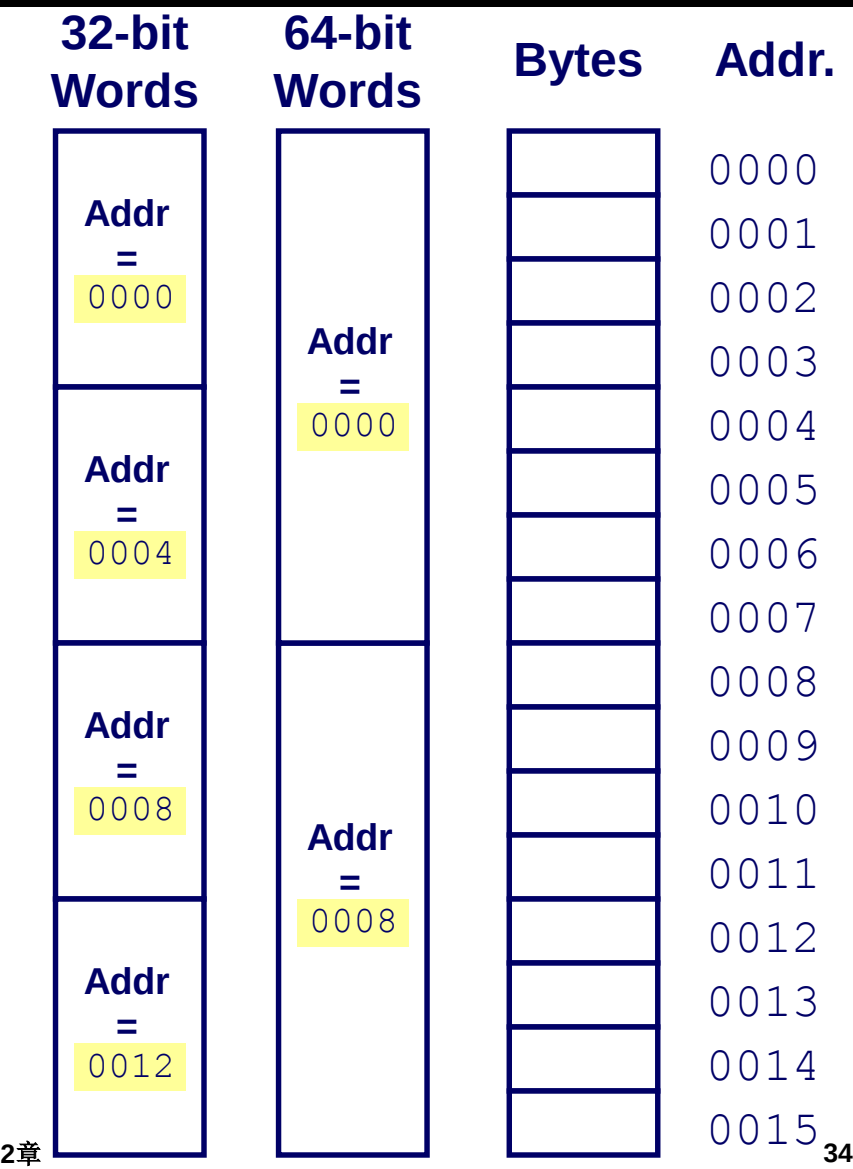
- 每个系统内部次序是一致的，但在系统间通信时可能会发生问题！
- 因为顺序不同，需要进行顺序转换
- 音、视频和图像等文件格式或处理程序都涉及到字节顺序问题

ex. Little endian: GIF, PC Paintbrush, Microsoft RTF,etc

Big endian: Adobe Photoshop, JPEG, MacPaint, etc

存储空间的组织

- 地址从初始的字节开始。
- 根据字长（8位/16位/32位/64位）来确定连续的地址编码。



存储边界对齐

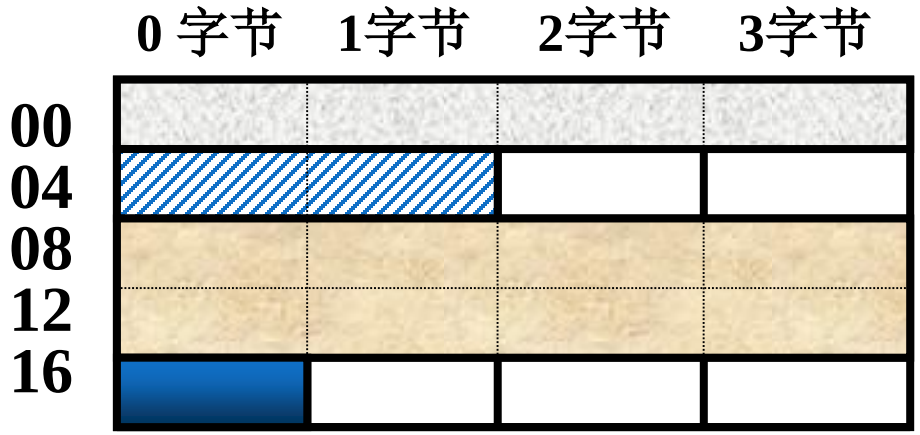
- 目前机器字长一般为32位或64位，而存储器地址按字节编址
- 指令系统支持对字节、半字、字及双字的运算
- 各种不同长度的数据存放时，有两种处理方式：
  - 按边界对齐 （假定字长的宽度为32位，按字节编址）
    - 双字地址：8的倍数(低四位为0)
    - 字地址：4的倍数(低两位为0)
    - 半字地址：2的倍数(低位为0)
    - 字节地址：任意
  - 不按边界对齐

    坏处：可能会增加访存次数！

存储边界对齐

存储器按字节编址，32位字长，每次只能读写某个字地址开始的4个单元中连续的1个、2个、3个或4个字节

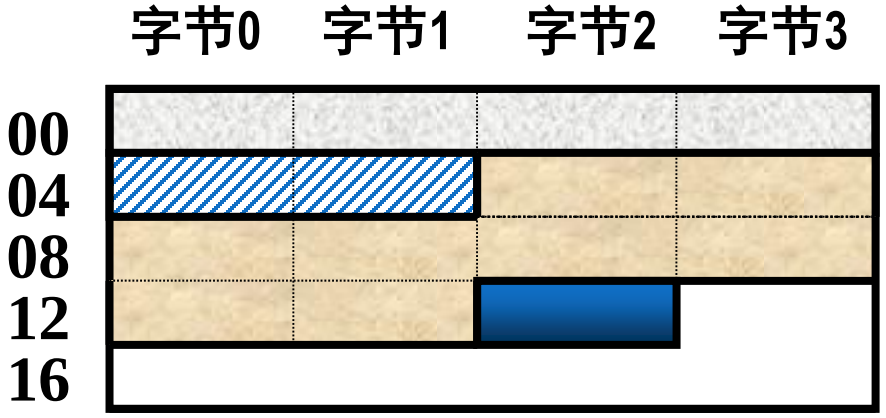
- 如：int i, short k, double x, char c



按边界对齐

则： &i=0; &k=4; &x=8; &c=16;

x的读取周期？



边界不对齐

则： &i=0; &k=4; &x=6; &c=14;

空间和时间的权衡！

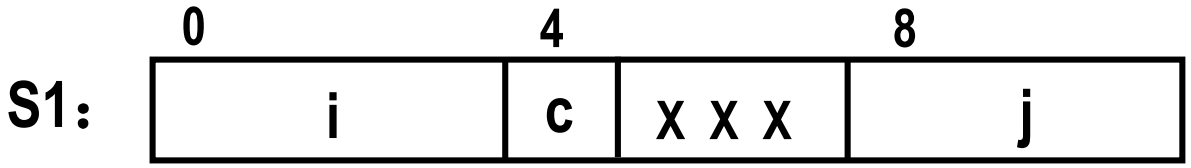
存储边界对齐

```
struct  S1 {
        int     i;
        char    c;
        int     j;
};
```

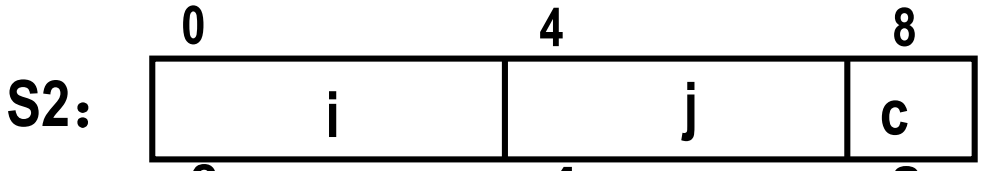
```
struct  S2 {
        int     i ;
        int     j ;
        char    c ;
} ;
```

在要求对齐的情况下，哪种结构声明更好？

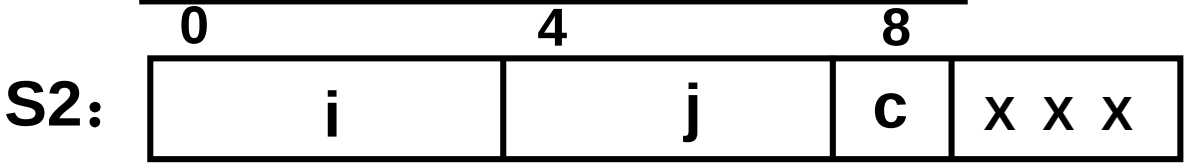
S2比S1好



需要12个字节

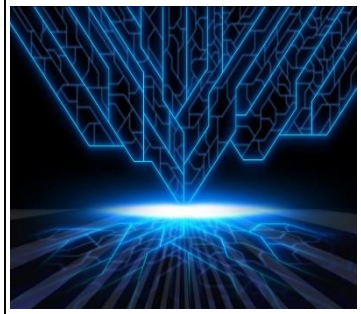


只需要9个字节，通常最后加3字节



对于“struct S2 d[4]”，只分配9个字节能否满足对齐要求？

位运算



按位运算

逻辑运算

移位运算

位扩展

位截断运算

基于位的运算

- 布尔代数：1850 英国乔治·布尔发现，采用二进制值0（ False ）和1（True），能够设计出一种代数的方法研究逻辑推理的基本原则。
  - 布尔运算：
    - 取反 /Not/ ~： ~A = 1 when A=0
    - 或/ Or/ |： A|B = 1 when either A=1 or B=1
    - 与 /And /&：A&B = 1 when both A=1 and B=1
    - 异或/ Xor/ ^ ： A^B = 1 when A≠B
- 1937年香农建立了布尔代数和数字电路之间的联系。
  - 利用布尔代数设计和分析继电器网络。
  - “关”表示为1，“开”表示为0.

逻辑运算

- 用于逻辑表达式的运算，总是返回0或1。0看成“False”，任何非0的数看成“True”。
- 操作
  - “||”表示“或”运算
  - “&&”表示“与”运算
  - “!”表示“非”运算

Examples (char data type)

!0x41 = 0x00
!0x00 = 0x01
!!0x41 = 0x01

0x69 && 0x55 = 0x01
0x69 || 0x55 = 0x01

- 与按位运算的差别
  - 符号表示不同：& ： && ； |： || ； ~： !
  - 运算过程不同：按位 ： 整体
  - 结果类型不同：位串 ： 逻辑值

按位运算

- 对位串（位向量）实现“掩码”（mask）操作或相应的其他处理
- 操作
  - 按位或：“|”
  - 按位与：“&”
  - 按位取反：“~”
  - 按位异或：“^”

C语言中按位运算适用任何“整数”类型：
long, int, short, char, unsigned
例如： (Char data type)
~0x41 = 0xBE
~01000001_2 →=10111110_2
0x69 & 0x55 =0x41
01101001_2 & 01010101_2 = 01000001_2
0x69 | 0x55 = 0x7D
01101001_2 | 01010101_2 → 01111101_2

问题：如何从16位采样数据y中提取高位字节，并使低字节为0？
可用“&”实现“掩码”操作：y & 0xFF00
例如，当y=0x2C0B时，得到结果为：0x2C00

移位运算

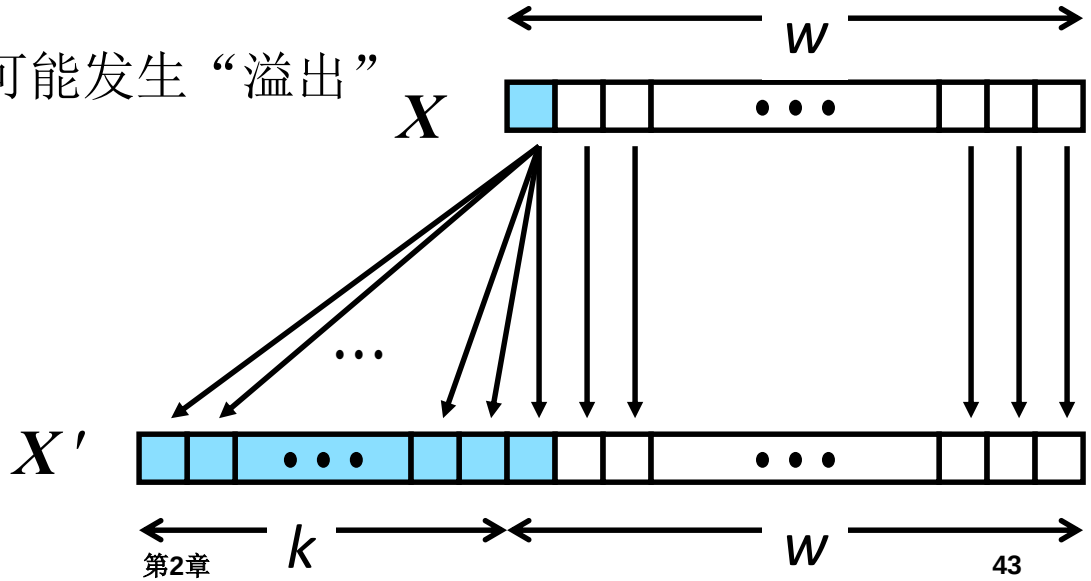
- 左移： x<<k，表示丢弃最高的k位，并在右端补充k个0，k小于x的位数。
- 右移：x>>k，由x的类型确定补充的数值。
  - 无符号数：表示丢弃最低的k位，并在左端补充k个0
  - 带符号整数：
    - 逻辑右移：表示丢弃最低的k位，并在左端补充k个0
    - 算术右移：表示丢弃最低的k位，并在左端补充k个符号位

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

位扩展和位截断运算

- 用途：类型转换时可能需要数据扩展或截断。
- 操作：没有专门操作运算符，根据类型转换前后数据长短确定是扩展还是截断
  - 扩展：短转长
    - 无符号数：0扩展，前面补0
    - 带符号整数：符号扩展，前面补符号位
  - 截断：长转短
    - 强行将高位丢弃，故可能发生“溢出”



位扩展和位截断实例

```
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

例2（截断操作）：i 和 j 是否相等？

```
int i = 32768;
short si = (short) i;
int j = si;
```

不相等！

i = 32768　　00 00 80 00

si = -32768　　80 00

j = -32768　　FF FF 80 00

原因：对i截断时发生了“溢出”，即：32768截断为16位数时，因其超出16位能表示的最大值，故无法截断为正确的16位数！

数据运算

无符号和带符号整数的加减运算

无符号和带符号整数的乘除运算

变量与常数之间的乘除运算

基本运算类型

- C语言程序中的基本数据类型及基本运算类型
  - 基本数据类型
    - 无符号数、带符号整数、浮点数、位串、字符（串）
  - 基本运算类型
    - 算术、按位、逻辑、移位、扩展和截断、匹配
- 计算机如何实现高级语言程序中的运算？
  - 将各类表达式编译（转换）为指令序列
  - 计算机直接执行指令来完成运算

n位整数加/减运算器

先看一个C程序段：

```
int x=9, y=-6, z1, z2;
z1=x+y;
z2=x-y;
```

补码的定义　　假定补码有n位，则：

$[X]_{补}=2^n+X \quad (-2^n \leqslant X < 2^n, \bmod 2^n)$

问题：上述程序段中，x和y的机器数是什么？z1和z2的机器数是什么？

回答：x的机器数为$[x]_{补}$，　y的机器数为$[y]_{补}$；

z1的机器数为$[x+y]_{补}$；

z2的机器数为$[x-y]_{补}$。

因此，计算机中需要有一个电路，能够实现以下功能：

已知 $[x]_{补}$ 和 $[y]_{补}$，计算$[x+y]_{补}$ 和 $[x-y]_{补}$。

根据补码定义，有如下公式：

$$[x+y]_{补}=2^n+x+y=2^n+x+2^n+y=[x]_{补}+[y]_{补} \pmod{2^n}$$

$$[x-y]_{补}=2^n+x-y=2^n+x+2^n-y=[x]_{补}+[-y]_{补} \pmod{2^n}$$

$$[-y]_{补}=\overline{[y]}_{补}+1$$

n位整数加/减运算器

- 补码加减运算公式

$$[A+B]_{补} = [A]_{补} + [B]_{补} \quad (\bmod 2^n)$$

$$[A-B]_{补} = [A]_{补} + [-B]_{补} \quad (\bmod 2^n)$$

– 实现减法的主要工作在于：求$[-B]_{补}$

问题：如何求$[-B]_{补}$？

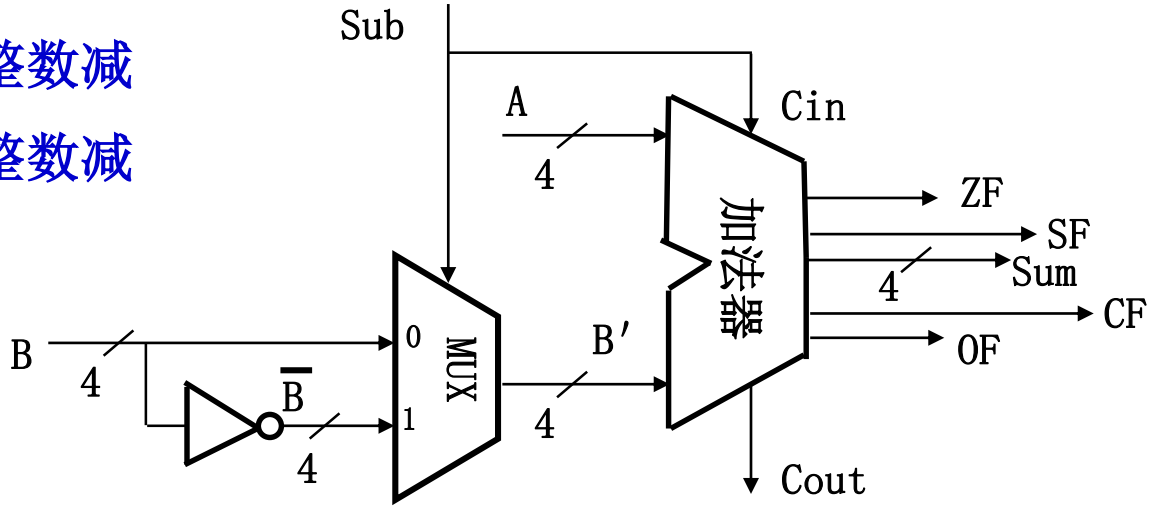
$$[-B]_{补} = \overline{[B]}_{补} + 1$$

- 利用带标志加法器，可构造整数加/减运算器，进行以下运算：

  无符号整数加、无符号整数减

  带符号整数加、带符号整数减

当Sub为1时，做减法
当Sub为0时，做加法

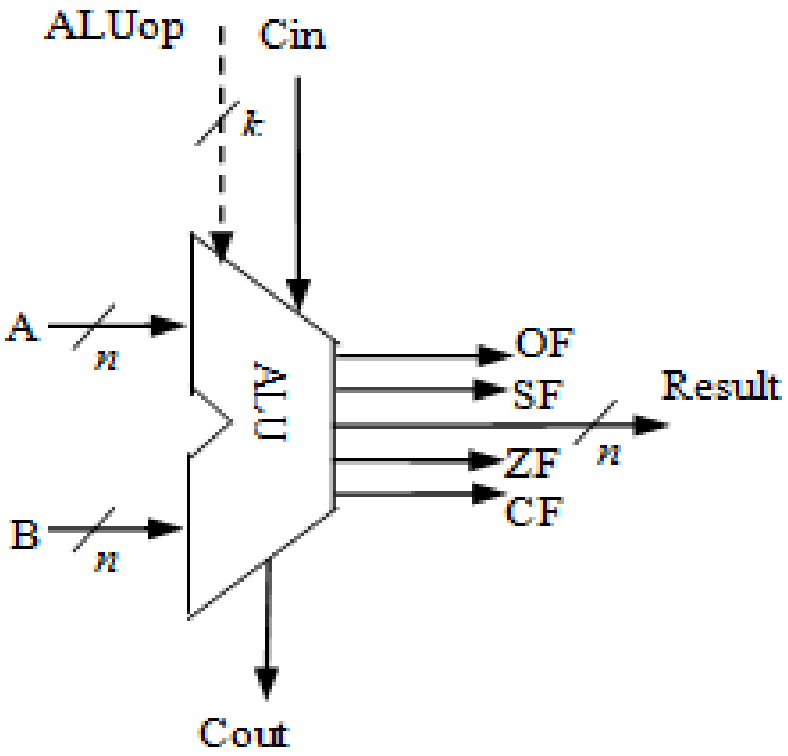


整数加/减运算部件

在整数加/减运算部件基础上，加上寄存器、移位器以及控制逻辑，就可实现ALU、乘/除运算以及浮点运算电路

算术逻辑部件（ALU）

- 进行基本算术运算与逻辑运算
  - 无符号整数加、减
  - 带符号整数加、减
  - 与、或、非、异或等逻辑运算
- 核心电路是整数加/减运算部件
- 输出除和/差等，还有标志信息
- 有一个操作控制端（ALUop），用来决定ALU所执行的处理功能。ALUop的位数k决定了操作的种类，例如，当位数k为3时，ALU最多只有2^3=8种操作。



ALU符号

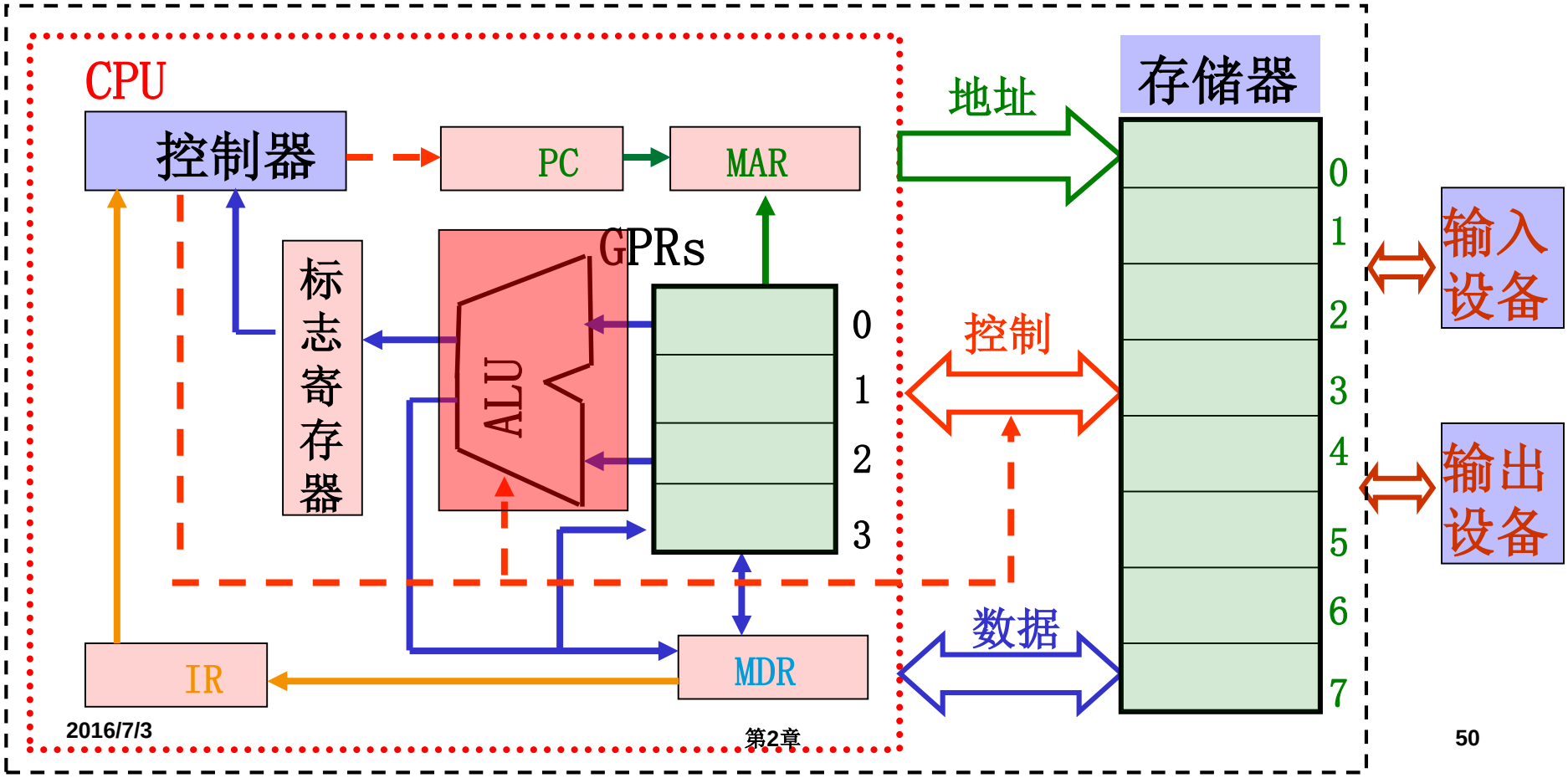
ALUop	Result	ALUop	Result	ALUop	Result	ALUop	Result
0 0 0	A加B	0 1 0	A与B	1 0 0	A取反	1 1 0	A
0 0 1	A减B	0 1 1	A或B	1 0 1	A⊕B	1 1 1	未用

回顾：计算机中最基本的部件

CPU：中央处理器；PC：程序计数器；MAR：存储器地址寄存器

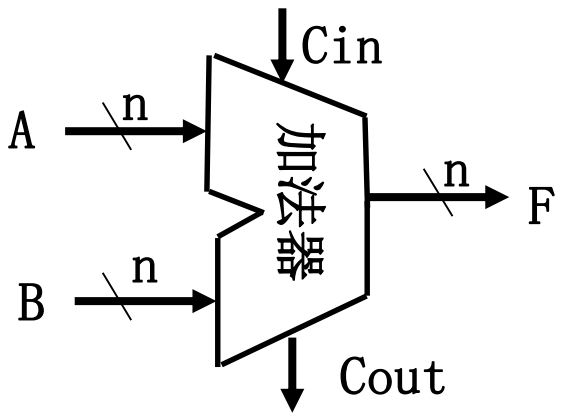
ALU：算术逻辑部件；IR：指令寄存器；MDR：存储器数据寄存器

GPRs：通用寄存器组（由若干通用寄存器组成）



整数加、减运算

- C语言程序中的整数有
  - 带符号整数，如char、short、int、long型等
  - 无符号整数，如unsigned char、unsigned short、unsigned等
- 指针、地址等通常被说明为无符号整数，因而在进行指针或地址运算时，需要进行无符号整数的加、减运算
- 无符号整数和带符号整数的加、减运算电路完全一样，这个运算电路称为整数加减运算部件，基于带标志加法器实现
- 最基本的加法器，因为只有n位，所以是一种模2^n运算系统！



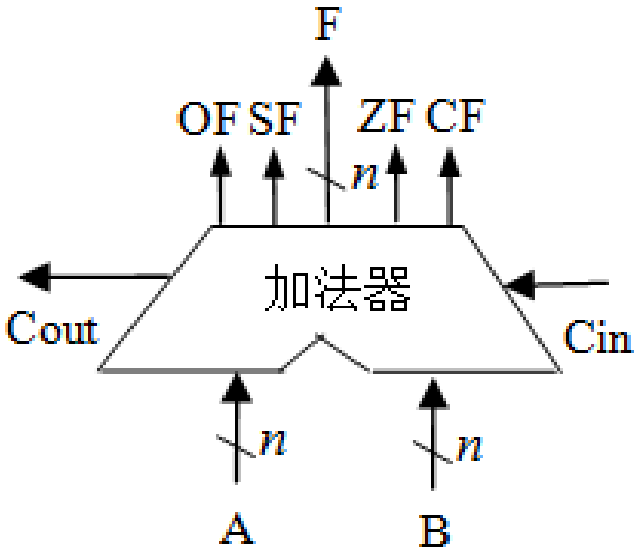
例：n=4，A=1001，B=1100

则：F=0101，Cout=1

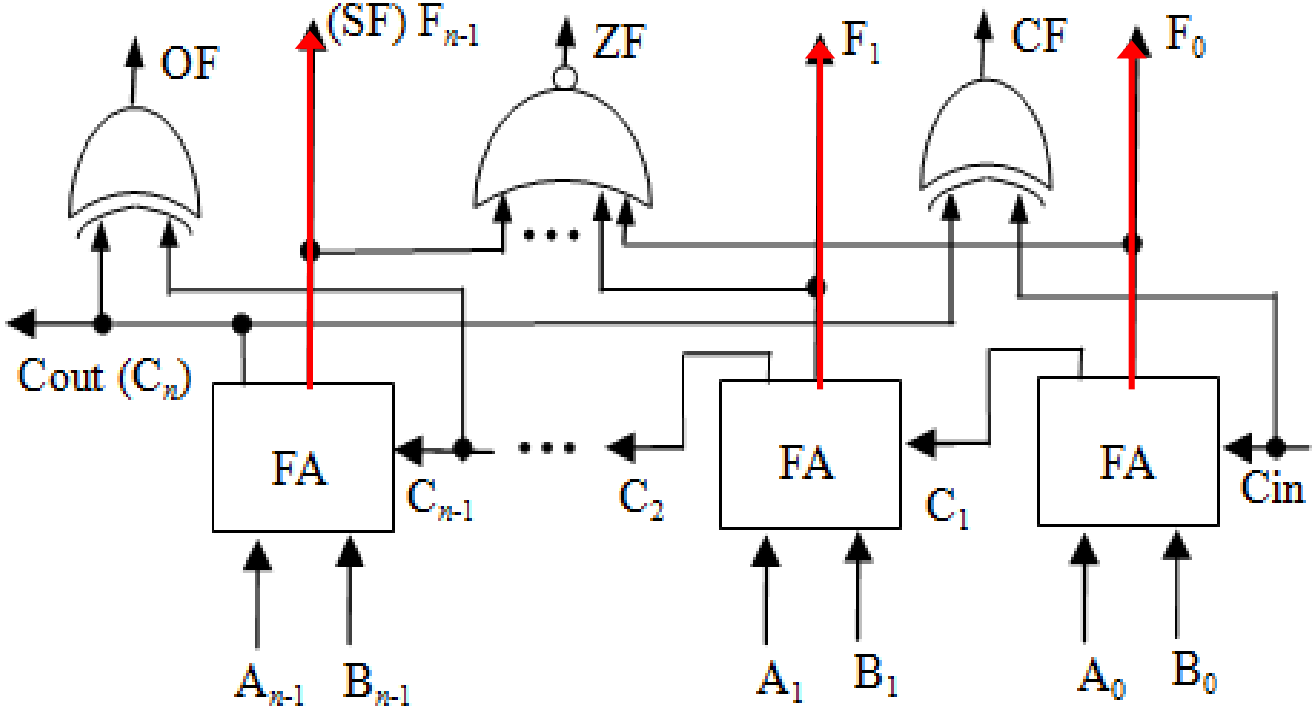
还记得这个加法器是如何实现的？

n位带标志加法器

- n位加法器无法用于两个n位带符号整数（补码）相加，无法判断是否溢出
- 程序中经常需要比较大小，通过（在加法器中）做减法得到的标志信息来判断



带标志加法器符号



带标志加法器的逻辑电路

溢出标志OF：

$OF=C_n \oplus C_{n-1}$

符号标志SF：

$SF=F_{n-1}$

零标志ZF=1当且仅当F=0；

进位/借位标志CF：

$CF=Cout \oplus Cin$

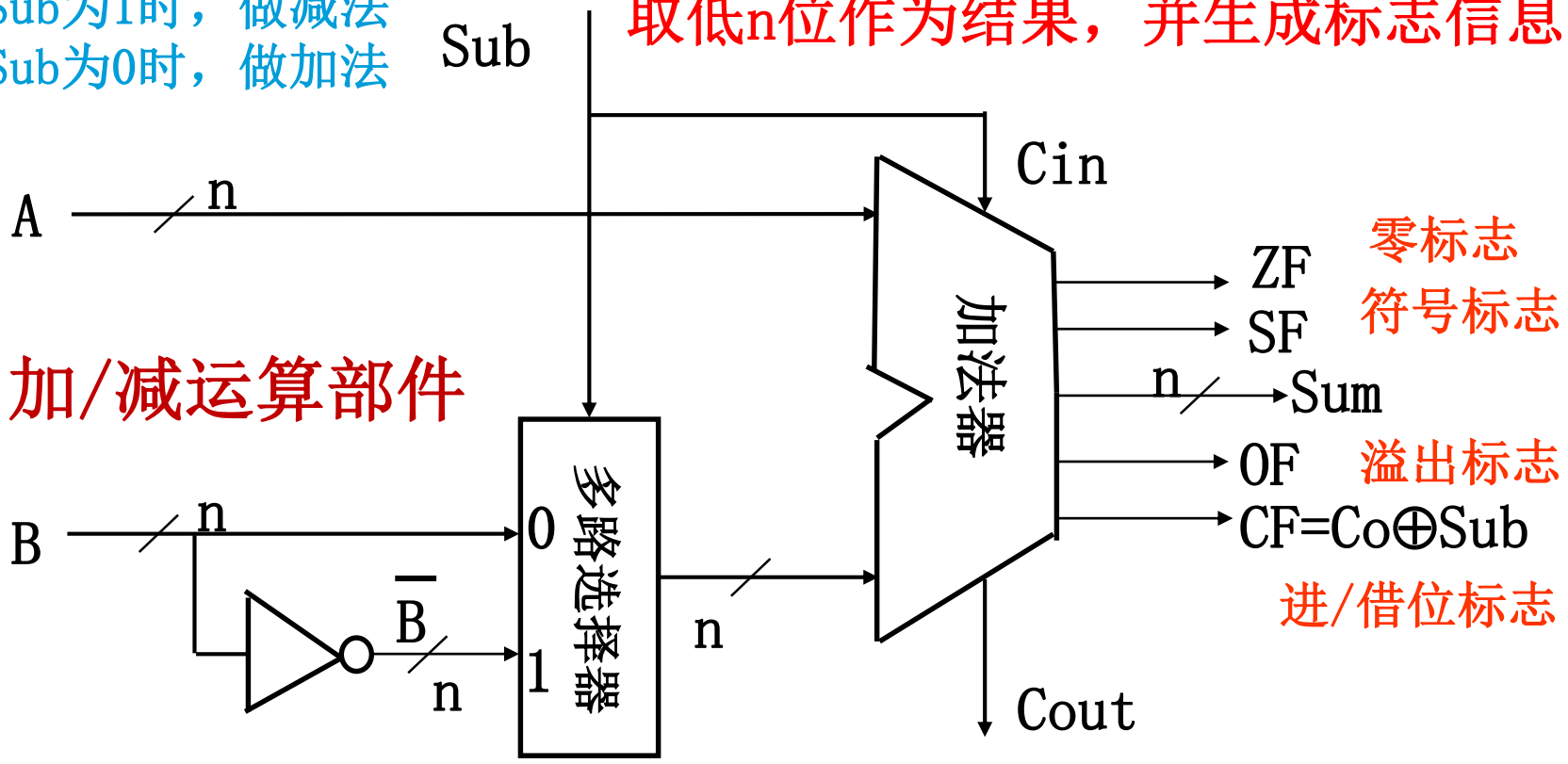
ALU的核心

重要认识1：计算机中所有算术运算都基于加法器实现！

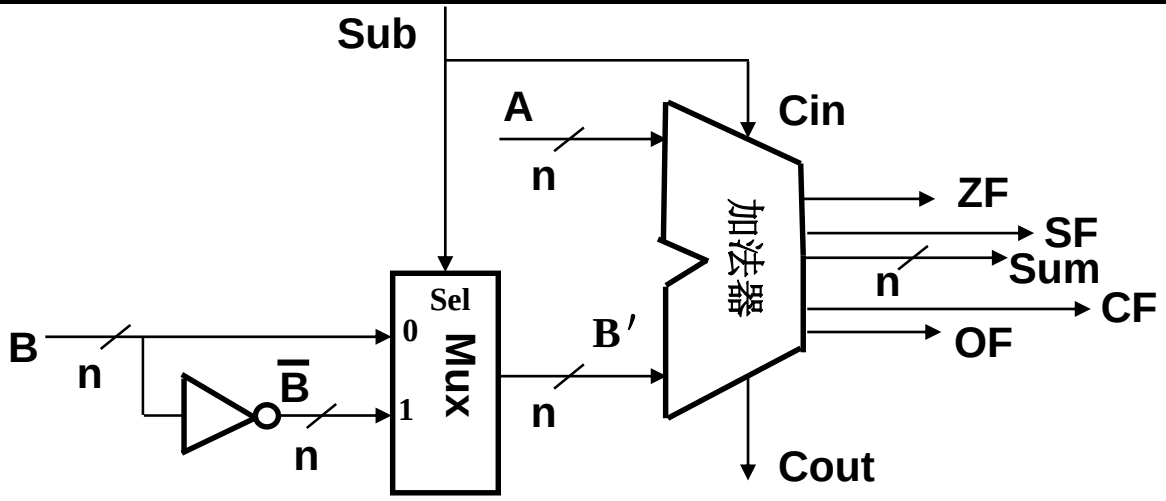
重要认识2：加法器不知道所运算的是带符号数还是无符号数。

重要认识3：加法器不判定对错，总是取低n位作为结果，并生成标志信息。

当Sub为1时，做减法
当Sub为0时，做加法



条件标志位（条件码CC）



整数加/减运算部件

问题：为什么要生成并保存条件标志？

为了在分支指令（条件转移指令）中被用作是否转移执行的条件！

```
if (i>j)
{ …}
```

- 零标志ZF、溢出标志OF、进/借位标志CF、符号标志SF称为条件标志。
- 条件标志（Flag）在运算电路中产生，被记录到专门的寄存器中
- 存放标志的寄存器通常称为程序/状态字寄存器或标志寄存器。每个标志对应标志寄存器中的一个标志位。 如，IA-32中的EFLAGS寄存器

如何得到各个标志位

OF：若A与B同符号但与Sum不同符号，则1；否则0。SF：sum符号

ZF：如Sum为0，则1，否则0。CF：Cout ⊕ Cin

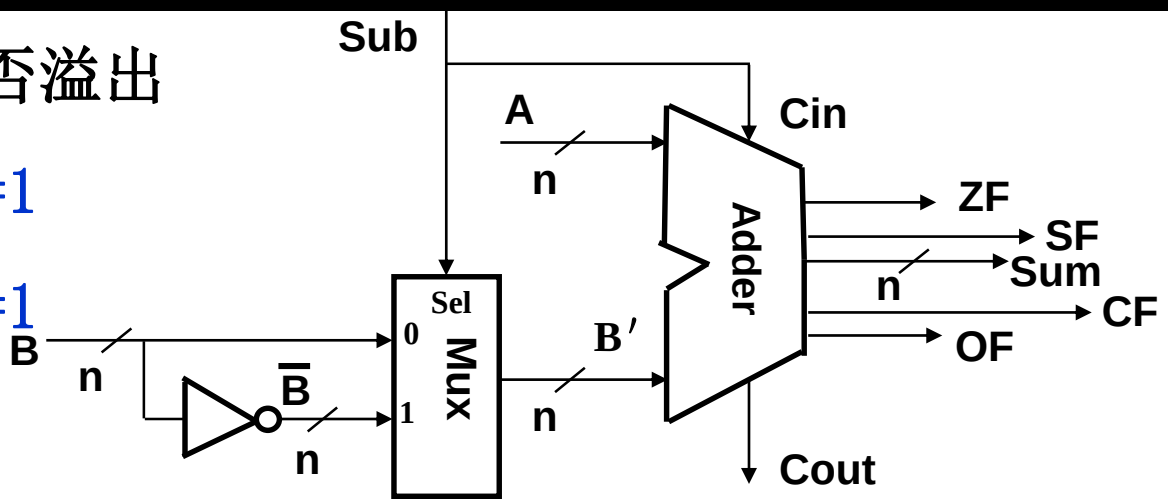
整数加法举例

做加法时，主要判断是否溢出

无符号加溢出条件：CF=1

带符号加溢出条件：OF=1

若n=8，计算107+46=?



整数加/减运算部件

107_{10}= 0110 1011$_2$

46_{10}= 0010 1110$_2$

0 1001 1001

进位是真正的符号：+153

两个正数相加，结果为负数，故溢出！即OF=1

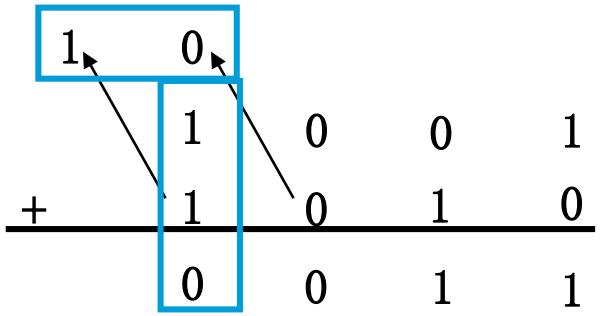
溢出标志OF=1、零标志ZF=0、符号标志SF=1、进位标志CF=0

无符号：sum=153，因为CF=0，故未发生溢出，结果正确！

带符号：sum= -103，因为OF=1，故发生溢出，结果错误！

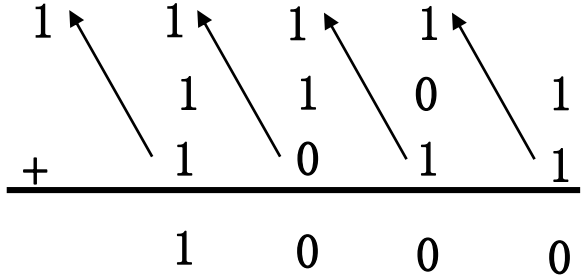
整数减法举例

-7- 6 = -7 + (-6) = +3 ✗
9 - 6 = 3 √



OF=1、ZF=0
SF=0、借位CF=0

-3 - 5 = - 3 + (- 5) = - 8 √
13 - 5 = 8 √



OF=0、ZF=0、
SF=1、借位CF=0

带符号溢出：（1）最高位和次高位的进位不同
（2）和的符号位和加数的符号位不同

无符号减溢出：差为负数，即借位CF=1

做减法以比较大小，规则：
Unsigned: CF=0时，大于
Signed：OF=SF时，大于

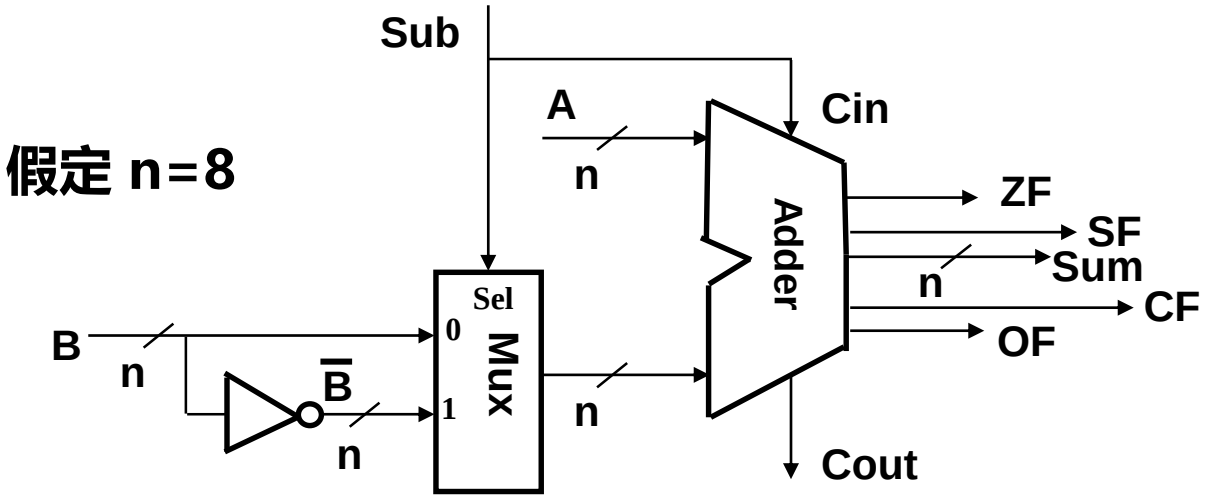
验证：9>6，故CF=0；13>5，故CF=0

验证：-7<6，故OF≠SF
-3<5，故OF≠SF

整数减法举例

```
unsigned int x=134;
unsigned int y=246;
int m=x;
int n=y;
unsigned int z1=x-y;
unsigned int z2=x+y;
int k1=m-n;
int k2=m+n;
```

无符号和带符号加减运算都用该部件执行



x和m的机器数一样：1000 0110，y和n的机器数一样：1111 0110

z1和k1的机器数一样：1001 0000，CF=1，OF=0，SF=1

z1的值为144（=134-246+256，x-y<0），k1的值为-112。

无符号减公式：

$$result=\begin{cases} x-y & (x-y>0) \\ x-y+2^n & (x-y<0) \end{cases}$$

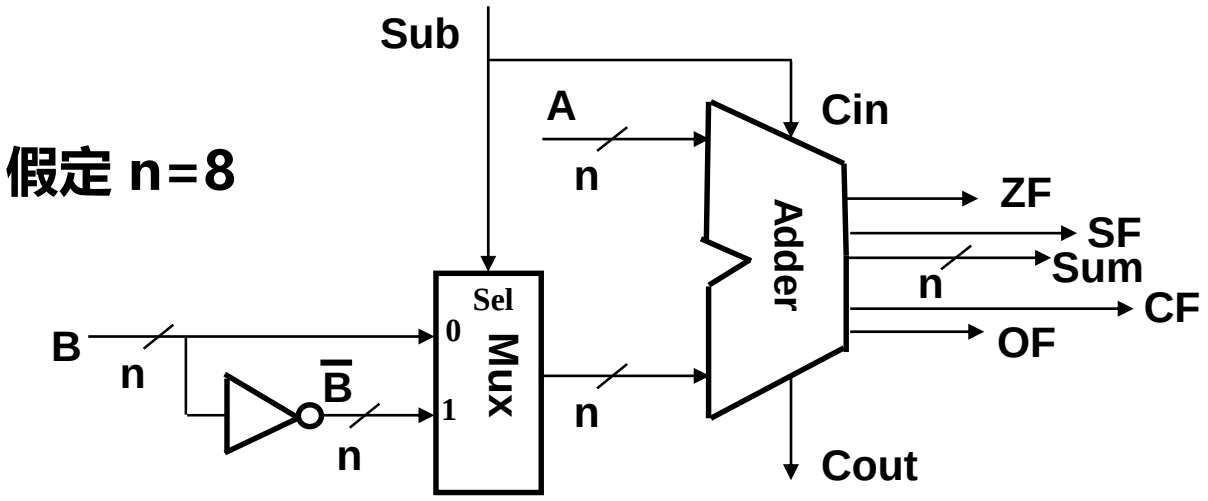
带符号减公式：

$$result=\begin{cases} x-y-2^n & (2^{n-1}\leqslant x-y) \quad \text{正溢出} \\ x-y & (-2^{n-1}\leqslant x-y<2^{n-1}) \quad \text{正常} \\ x-y+2^n & (x-y<-2^{n-1}) \quad \text{负溢出} \end{cases}$$

整数加法举例

```
unsigned int x=134;
unsigned int y=246;
int m=x;
int n=y;
unsigned int z1=x-y;
unsigned int z2=x+y;
int k1=m-n;
int k2=m+n;
```

无符号和带符号加减运算都用该部件执行



x和m的机器数一样：1000 0110，y和n的机器数一样：1111 0110

z2和k2的机器数一样：0111 1100，CF=1，OF=1，SF=0

z2的值为124（=134+246-256，x+y>256）

k2的值为124（=134+246-256，m+n>128，即正溢出）

带符号加公式：

$$result=\begin{cases} x+y-2^{n} & (2^{n-1}\leqslant x+y) \quad \text{正溢出} \\ x+y & (-2^{n-1}\leqslant x+y<2^{n-1}) \quad \text{正常} \\ x+y+2^{n} & (x+y<-2^{n-1}) \quad \text{负溢出} \end{cases}$$

无符号加公式：

$$result=\begin{cases} x+y & (x+y<2^{n}) \\ x+y-2^{n} & (2^{n}\leqslant x+y<2^{n+1}) \end{cases}$$

无符号整数加法溢出判断程序

如何用程序判断一个无符号数相加没有发生溢出

$$\text{result}=\begin{cases} x+y & (x+y<2^n) \\ x+y-2^n & (2^n \leqslant x+y<2^{n+1}) \end{cases}$$

发生溢出时，一定满足 result<x and result<y

否则，若$x+y-2^n \geq x$，则 $y \geq 2^n$ ，这是不可能的！

```
/* Determine whether arguments can be added without
overflow */
int uadd_ok(unsigned x, unsigned y)
{
     unsigned sum = x+y;
     return sum >= x;
}
```

带符号整数加法溢出判断程序

如何用程序判断一个带符号整数相加没有发生溢出

- /* Determine whether arguments can be added without overflow */

```
int tadd_ok(int x, int y) {
    int sum = x+y;
    int neg_over = x < 0 && y < 0 && sum >= 0;
    int pos_over = x >= 0 && y >= 0 && sum < 0;
    return !neg_over && !pos_over;
}
```

Add指令需要用以下公式：

CF=?, ZF=?, OF=?, SF=?

$$result=\begin{cases} x+y-2^{n} & (2^{n-1}\leqslant x+y) \quad \text{正溢出} \\ x+y & (-2^{n-1}\leqslant x+y<2^{n-1}) \quad \text{正常} \\ x+y+2^{n} & (x+y<-2^{n-1}) \quad \text{负溢出} \end{cases}$$

CF=0, ZF=0, OF=1, SF=1

CF=1, ZF=0, OF=1, SF=0

带符号整数减法溢出判断程序

$$\text{result}=\begin{cases} x+y-2^n & (2^{n-1}\leqslant x+y) & \text{正溢出} \\ x+y & (-2^{n-1}\leqslant x+y<2^{n-1}) & \text{正常} \\ x+y+2^n & (x+y<-2^{n-1}) & \text{负溢出} \end{cases}$$

带符号整数加

$$\text{result}=\begin{cases} x-y-2^n & (2^{n-1}\leqslant x-y) & \text{正溢出} \\ x-y & (-2^{n-1}\leqslant x-y<2^{n-1}) & \text{正常} \\ x-y+2^n & (x-y<-2^{n-1}) & \text{负溢出} \end{cases}$$

带符号整数减

模拟Sub指令需要！

以下程序检查带符号整数相减是否溢出有没有问题？

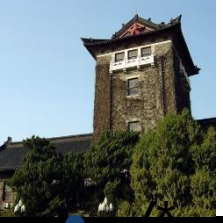
```
/* Determine whether arguments can be subtracted without
overflow */

/* WARNING: This code is buggy. */
int tsub_ok(int x, int y) {
      return tadd_ok(x, -y);
}
```

带符号减的溢出判断函数如何实现呢？

无符号减的溢出判断函数又如何实现呢？

当x>=0，y=0x80000000时，该函数判断错误



带符号整数减法溢出判断程序

```
/*
 * subOK - Determine if can compute x-y without overflow
 *   Example: subOK(0x80000000,0x80000000) = 1,
 *                  subOK(0x80000000,0x70000000) = 0,
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 20
 *   Rating: 3
 */
int subOK(int x, int y) {
  int diff = x+~y+1;
  int x_neg = x>>31;
  int y_neg = y>>31;
  int d_neg = diff>>31;
  /* Overflow when x and y have opposite sign, and d different from x */
  return !(~(x_neg ^ ~y_neg) & (x_neg ^ d_neg));
}
```

整数的乘运算

- 通常，高级语言中两个n位整数相乘得到的结果通常也是一个n位整数，也即，结果只取2n位乘积中的低n位。
  - 例如，在C语言中，参加运算的两个操作数的类型和结果的类型必须一致，如果不一致则会先转换为一致的数据类型再进行计算。

```
int mul(int x, int y)
{
    int z=x*y;
    return z;
}
```

x*y 被转换为乘法指令，在乘法运算电路中得到的乘积是64位，但是，只取其低32位赋给z。

整数的乘运算

在计算机内部，一定有$x^2 \geq 0$吗?

若x是带符号整数，则不一定！

如x是浮点数，则一定！

例如，当 n=4 时，$5^2=-7<0$ ！

```
int mul(int x, int y)
{
    int z=x*y;
    return z;
}
```

若x、y和z都改成unsigned类型，则判断方式为

乘积的高n位为全0，则不溢出

```
    0101
  × 0101
-------
    0101
+ 0101
-------
00011001
```

结果溢出

只取低4位，值为-111B=-7

高级语言程序如何判断z是正确值？

当 !x || z/x==y 为真时

编译器如何判断？

当 $-2^{n-1} \leqslant x*y < 2^{n-1}$ （不溢出）时

即：乘积的高n位为全0或全1，并等于低n位的最高位！

即：乘积的高n+1位为全0或全1

整数的乘运算

结论：假定两个n位无符号整数x_u和y_u对应的机器数为X_u和Y_u，$p_u=x_u \times y_u$，p_u为n位无符号整数且对应的机器数为P_u；

两个n位带符号整数x_s和y_s对应的机器数为X_s和Y_s，$p_s=x_s \times y_s$，p_s为n位带符号整数且对应的机器数为P_s。

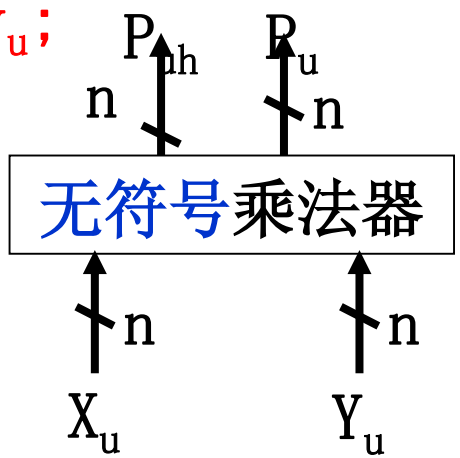
若$X_u=X_s$且$Y_u=Y_s$，则$P_u=P_s$。

无符号：若$P_{uh}=0$，则不溢出

带符号：若P_{sh}每位都等于P_s的最高位，则不溢出

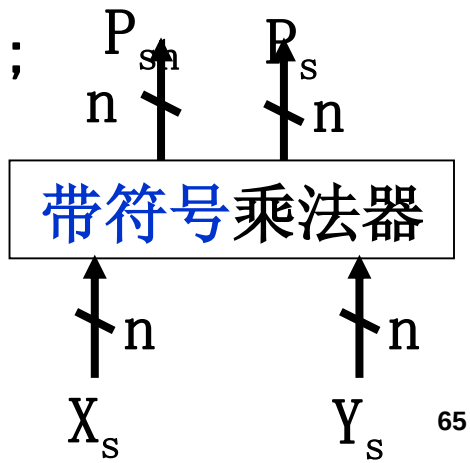
可用无符号乘来实现带符号乘，但高n位无法得到，故不能判断溢出。

$p_u=x_u*y_u$；



$P_{uh} \neq P_{sh}$

$p_s=x_s*y_s$；



整数的乘运算

- X*Y的高n位可以用来判断溢出，规则如下：
  - 无符号：若高n位全0，则不溢出，否则溢出
  - 带符号：若高n位全0或全1且等于低n位的最高位，则不溢出。

运算	x	X	y	Y	x×y	X×Y	p	P	溢出否
无符号乘	6	0110	10	1010	60	0011 1100	12	1100	溢出
带符号乘	6	0110	-6	1010	-36	1101 1100	-4	1100	溢出
无符号乘	8	1000	2	0010	16	0001 0000	0	0000	溢出
带符号乘	-8	1000	2	0010	-16	1111 0000	0	0000	溢出
无符号乘	13	1101	14	1110	182	1011 0110	6	0110	溢出
带符号乘	-3	1101	-2	1110	6	0000 0110	6	0110	不溢出
无符号乘	2	0010	12	1100	24	0001 1000	8	1000	溢出
带符号乘	2	0010	-4	1100	-8	1111 1000	-8	1000	不溢出

整数的乘运算

- 硬件保留2n位乘积，故指令的乘积可达2n位，可供编译器使用
- 如果程序不采用防止溢出的措施，且编译器也不生成用于溢出处理的代码，就会发生一些由于整数溢出而带来的问题。
- 指令：分无符号数乘指令、带符号整数乘指令
- 乘法指令的操作数长度为n，而乘积长度为2n，例如：
  - IA-32中，若指令只给出一个操作数SRC，则另一个源操作数隐含在累加器AL/AX/EAX中，将SRC和累加器内容相乘，结果存放在AX（16位时）或DX-AX（32位时）或EDX-EAX（64位时）中。

乘法指令可生成溢出标志，编译器也可使用2n位乘积来判断是否溢出！

整数乘法溢出漏洞

以下程序存在什么漏洞，引起该漏洞的原因是什么。

```
/* 复制数组到堆中，count为数组元素个数 */
int copy_array(int *array, int count) {
     int i;
    /* 在堆区申请一块内存 */
     int *myarray = (int *) malloc(count*sizeof(int));
    if (myarray == NULL)
        return -1;
    for (i = 0; i < count; i++)
        myarray[i] = array[i];
    return count;
}
```

2002年，Sun Microsystems公司的RPC XDR库带的xdr_array函数发生整数溢出漏洞，攻击者可利用该漏洞从远程或本地获取root权限。

攻击者可构造特殊参数来触发整数溢出，以一段预设信息覆盖一个已分配的堆缓冲区，造成远程服务器崩溃或者改变内存数据并执行任意代码。

当参数count很大时，则count*sizeof(int)会溢出。如count=2^{30}+1时，count*sizeof(int)=4。

堆（heap）中大量数据被破坏！

变量与常数之间的乘运算

- 整数乘法运算比移位和加法等运算所用时间长，通常一次乘法运算需要多个时钟周期，而一次移位、加法和减法等运算只要一个或更少的时钟周期，因此，编译器在处理变量与常数相乘时，往往以移位、加法和减法的组合运算来代替乘法运算。

  例如，对于表达式x*20，编译器可以利用$20=16+4=2^4+2^2$，将x*20转换为(x<<4)+(x<<2)，这样，一次乘法转换成了两次移位和一次加法。

- 不管是无符号数还是带符号整数的乘法，即使乘积溢出时，利用移位和加减运算组合的方式得到的结果都是和采用直接相乘的结果是一样的。

整数的除运算

- 对于带符号整数来说，n位整数除以n位整数，除$-2^{n-1}/-1= 2^{n-1}$会发生溢出外，其余情况都不会发生溢出。Why?

  因为商的绝对值不可能比被除数的绝对值更大，因而不会发生溢出，也就不会像整数乘法运算那样发生整数溢出漏洞。

- 因为整数除法，其商也是整数，所以，在不能整除时需要进行舍入，通常按照朝0方向舍入，即正数商取比自身小的最接近整数（Floor，地板），负数商取比自身大的最接近整数（Ceiling，天板）。

  例如，7/2=?,　-7/2=?

  7/2=3,　-7/2=-3

- 整数除0的结果可以用什么机器数表示？

  整数除0的结果无法用一个机器数表示！

- 整数除法时，除数不能为0，否则会发生“异常”，此时，需要调出操作系统中的异常处理程序来处理。

整数的除运算

代码段一：

```
int a = 0x80000000;
int b = a / -1;
printf("%d\n", b);
```

运行结果为-2147483648

用objdump看代码段一的反汇编代码，得知除以 -1 被优化成取负指令neg，故未发生除法溢出

代码段二：

```
int a = 0x80000000;
int b = -1;
int c = a / b;
printf("%d\n", c);
```

运行结果为“Floating point exception”，显然CPU检测到了异常

为什么显示是“浮点异常”呢？

为什么两者结果不同！

做实验看看，分析两者反汇编代码的异同！

变量与常数之间的除运算

- 对于整数除法运算，由于计算机中除法运算比较复杂，而且不能用流水线方式实现，所以一次除法运算大致需要几十个或更多个时钟周期，**比乘法指令的时间还要长**！
- 为了缩短除法运算的时间，编译器在处理一个变量与一个2的幂次形式的整数相除时，常采用右移运算来实现。
  - 无符号：逻辑右移
  - 带符号：算术右移
- 结果一定取整数
  - 能整除时，直接右移得到结果，移出的为全0

    例如，12/4=3：0000 1100>>2=0000 0011

    -12/4=-3：1111 0100 >>2=1111 1101
  - 不能整除时，右移移出的位中有非0，需要进行相应处理

变量与常数之间的除运算

- 不能整除时，采用朝零舍入，即截断方式
  - 无符号数、带符号正整数（地板）：**移出的低位直接丢弃**
  - 带符号负整数（天板）：加偏移量(2^k-1)，然后再右移k 位 ，低位截断（这里K 是右移位数）

    举例：

    无符号数　14/4=3：0000 1110>>2=0000 0011

    带符号负整数　-14/4=-3

    若直接截断，则 1111 0010 >>2=1111 1100=-4≠-3

    应先纠偏，再右移：k=2, 故$(-14+2^2-1)/4=-3$

    即： 1111 0010+0000 0011=1111 0101

    1111 0101>>2=1111 1101=-3

变量与常数之间的除运算—举例

- 假设*x*为一个int型变量，请给出一个用来计算*x*/32的值的函数div32。要求不能使用除法、乘法、模运算、比较运算、循环语句和条件语句，可以使用右移、加法以及任何按位运算。

解：若x为正数，则将x右移k位得到商；若x为负数，则x需要加一个偏移量(2^k-1)后再右移k位得到商。因为$32=2^5$，所以 k=5。

即结果为：(x>=0 ? x : (x+31))>>5

但题目要求不能用比较和条件语句，因此要找一个计算偏移量b的方式

这里，x为正时b=0，x为负时b=31. 因此，可以从x的符号得到b

x>>31 得到的是32位符号，取出最低5位，就是偏移量b。

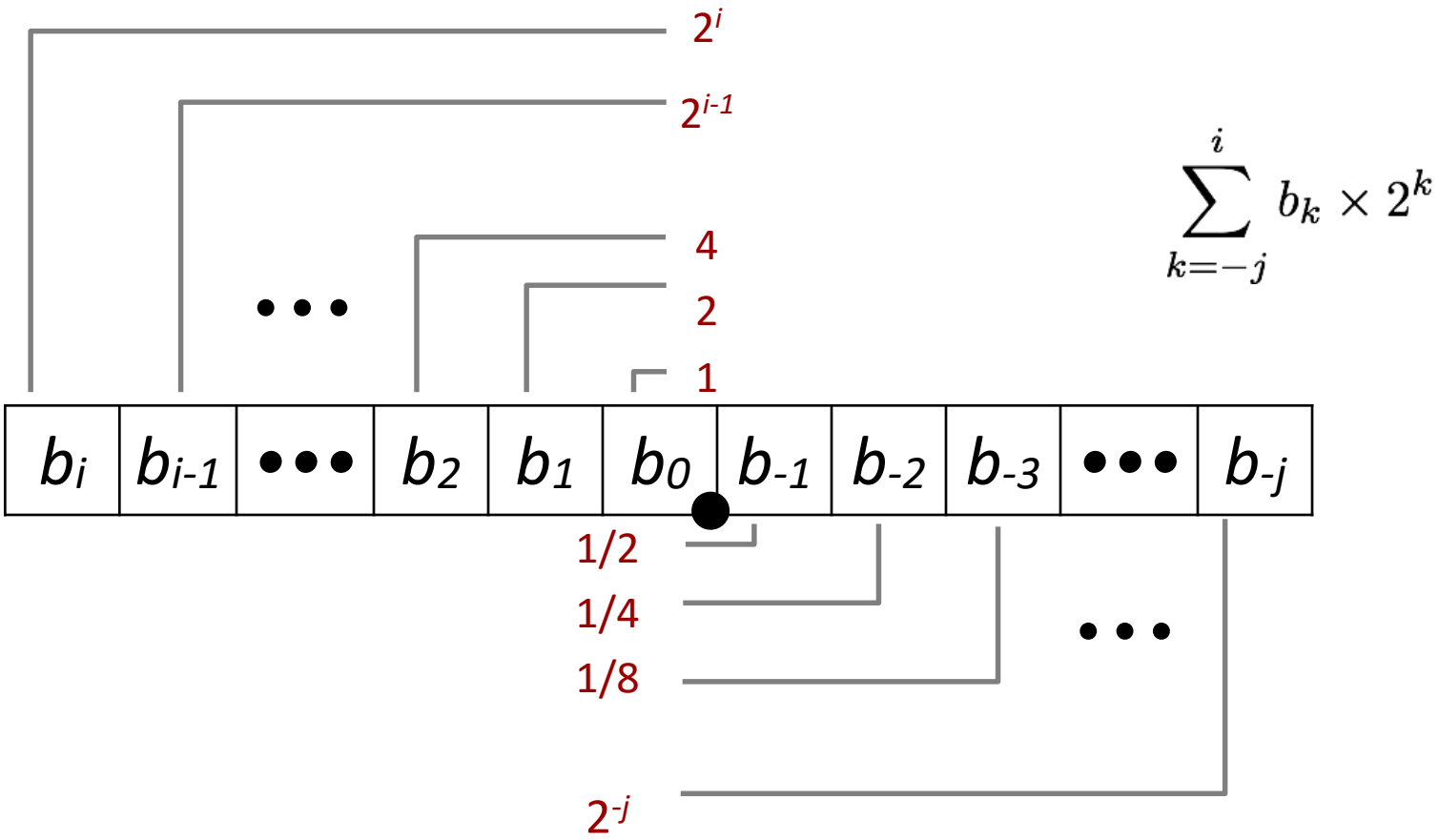
```
int div32(int x)
{  /* 根据x的符号得到偏移量b */
    int b=(x>>31) & 0x1F;
    return (x+b)>>5;
}
```

浮点数

浮点数

- 二进制小数表示
- IEEE浮点数标准定义
- 浮点数特性
- 浮点数运算
- 浮点数在C语言中的使用

二进制小数表示



$$\sum_{k=-j}^{i} b_k \times 2^k$$

二进制小数实例

Value	Representation
5 ¾	101.11_2
2 7/8	010.111_2
63/64	000.111111_2

Value	Representation
1/3	$0.0101010101[01]\ldots_2$
1/5	$0.00110011001[0011]\ldots_2$
1/10	$0.000110011001[0011]\ldots_2$

Value	Representation
13亿	$1100\ 0001\ 1011\ 0111\ 0001\ 0000\ 1000\ 0000\ 00_2$
地球的质量	5.98×10^{24} kg=（ ？ ）$_2$ 需要多少位二进制数表示

受限：

只能精确表示形如$x/2^k$

仅考虑有限长度的编码，只能近似表示小数的值

表示很大的数比较困难

浮点数标准-IEEE 754

直到80年代初，各个机器内部的浮点数表示格式还没有统一，因而相互不兼容，机器之间传送数据时，带来麻烦

1970年代后期，IEEE成立委员会着手制定浮点数标准

1985年完成浮点数标准IEEE 754的制定

现在所有计算机都采用IEEE 754来表示浮点数

This standard was primarily the work of one person, UC Berkeley math professor William Kahan.

1989
ACM Turing
Award Winner!

Prof. William Kahan

www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

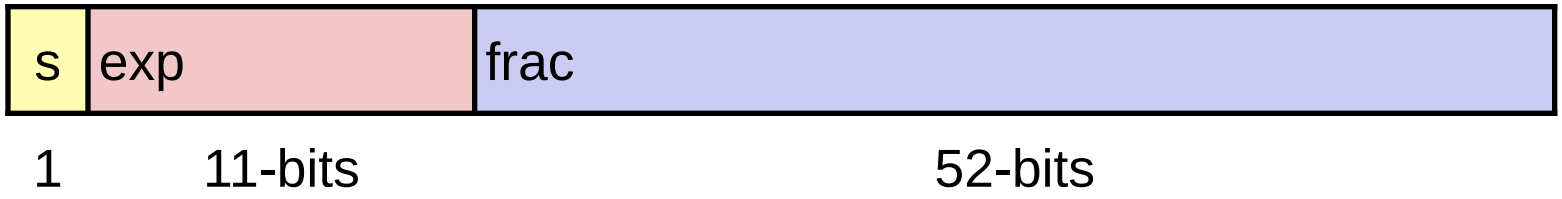
浮点数表示形式

- IEEE754浮点标准形式$V=(-1)^S \ M \ 2^E$
  - 符号位S(Sign)：决定这个数是正数(s=0)还是负数(s=1)。
  - 尾数M(significand)：是一个二进制小数，范围在[1.0，2.0）
  - 阶码E(exponent)：对浮点数加权重，权重是2的E次幂。
- 编码表示，分3个字段：
  - 1位符号字段s
  - k位阶码字段exp：阶码E的编码
  - n位小数字段frac： 尾数M的编码

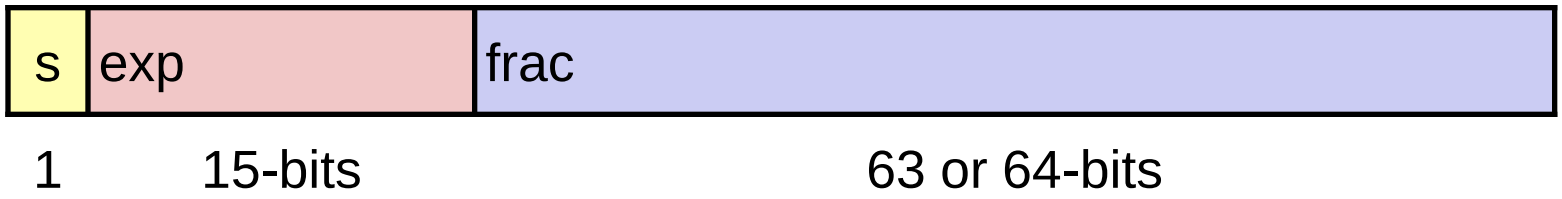
浮点数的精度

- 单精度：32位

- 双精度：64位



- 高精度: 80 位 (Intel only)



规格化数值

- 要求：阶码编码exp既不全0又不全1。
- 阶码编码exp表示阶码E的偏置(biased)形式：e***xp*** =E+ ***bias***
  - exp：无符号数
  - bias=$2^{k-1}-1$，k为阶码的位宽
    - 单精度：127 (Exp: 1…254, E: -126…127)
    - 双精度: 1023 (Exp: 1…2046, E: -1022…1023)
- 小数字段frac表示尾数M的编码M= 1.xxx…x_2
  - xxx…x: 小数字段数值
  - 最小值：000…0 (*M* = 1.0)
  - 最大值：111…1 (*M* = 2.0 – ε)

单精度浮点数示例

- 数值: Float F = 15213.0;
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$
- 小数字段Significand

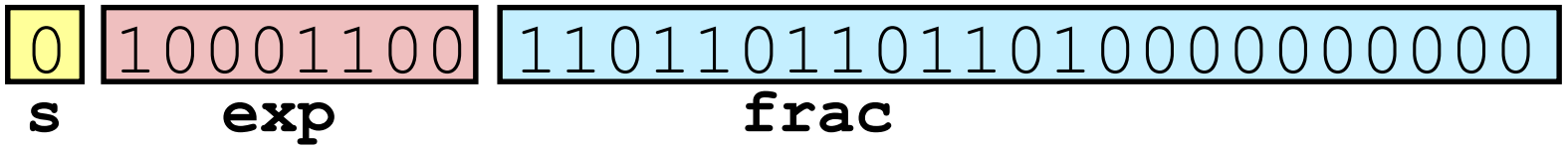
  $M = 1.\underline{1101101101101}_2$

  $\mathbf{frac} = \underline{1101101101101}0000000000_2$
- 阶码字段Exponent

  $E = 13$

  $Bias = 127$

  $Exp = 140 = \mathbf{10001100}_2$
- Result:



单精度浮点数示例

求0xBEE00000H表示的IEEE 754单精度浮点数值

1	011 11101	110 0000 0000 0000 0000 0000

$$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

符号位: 1 => negative

阶码字段:

- $0111\ 1101_2 = 125$
- 偏置校正: 125 - 127 = -2

小数字段:

$1 + 1x2^{-1} + 1x2^{-2} + 0x2^{-3} + 0x2^{-4} + 0x2^{-5} + \ldots$

$=1+2^{-1}+2^{-2} = 1+0.5+0.25 = 1.75$

数值表示: $-1.75x2^{-2} = -0.4375$

非规格化数值

- 阶码编码exp等于全0
- 阶码编码exp：$E = 1 - Bias$

- 小数字段frac表示尾数M的编码M= $0.xxx\ldots x_2$
  - xxx…x: bits of frac　　隐含非规格化小数字段M小于1
  - Exp= 000…0，frac= 000…0
    - 表示数值为0
    - 有正负0区分
  - Exp= 000…0，frac ≠ 000…0
    - 数值非常接近于0

特殊值

- 阶码编码exp等于全1
  - Exp= 111…1 ，frac= 000…0
    - 表示数值为∞
    - 当两个非常大的数相乘或除数为0时，表示数据溢出
    - 1.0/0.0 = −1.0/−0.0 = +∞，1.0/−0.0 = −∞
  - Exp= 111…10 ，frac ≠ 000…0
    - 表示NaN，不是一个数（Not a Number）
    - 一些运算结果不能是实数或无穷，就会返回该值
    - E.g., sqrt(–1), ∞ − ∞, ∞ × 0

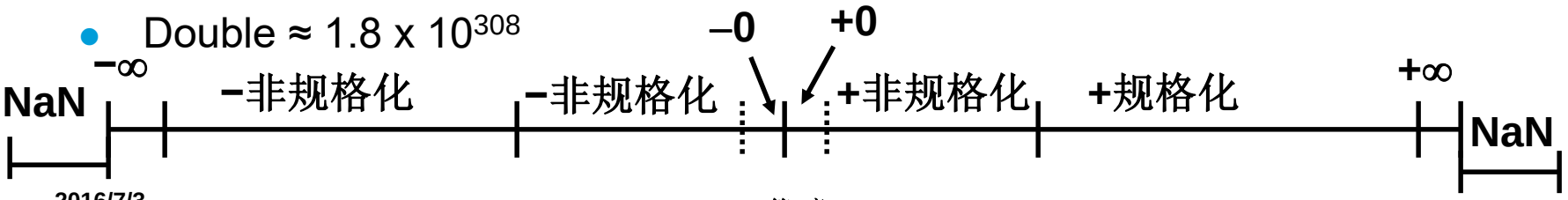
8位浮点数示(k-4,n=3)

正数表示范围

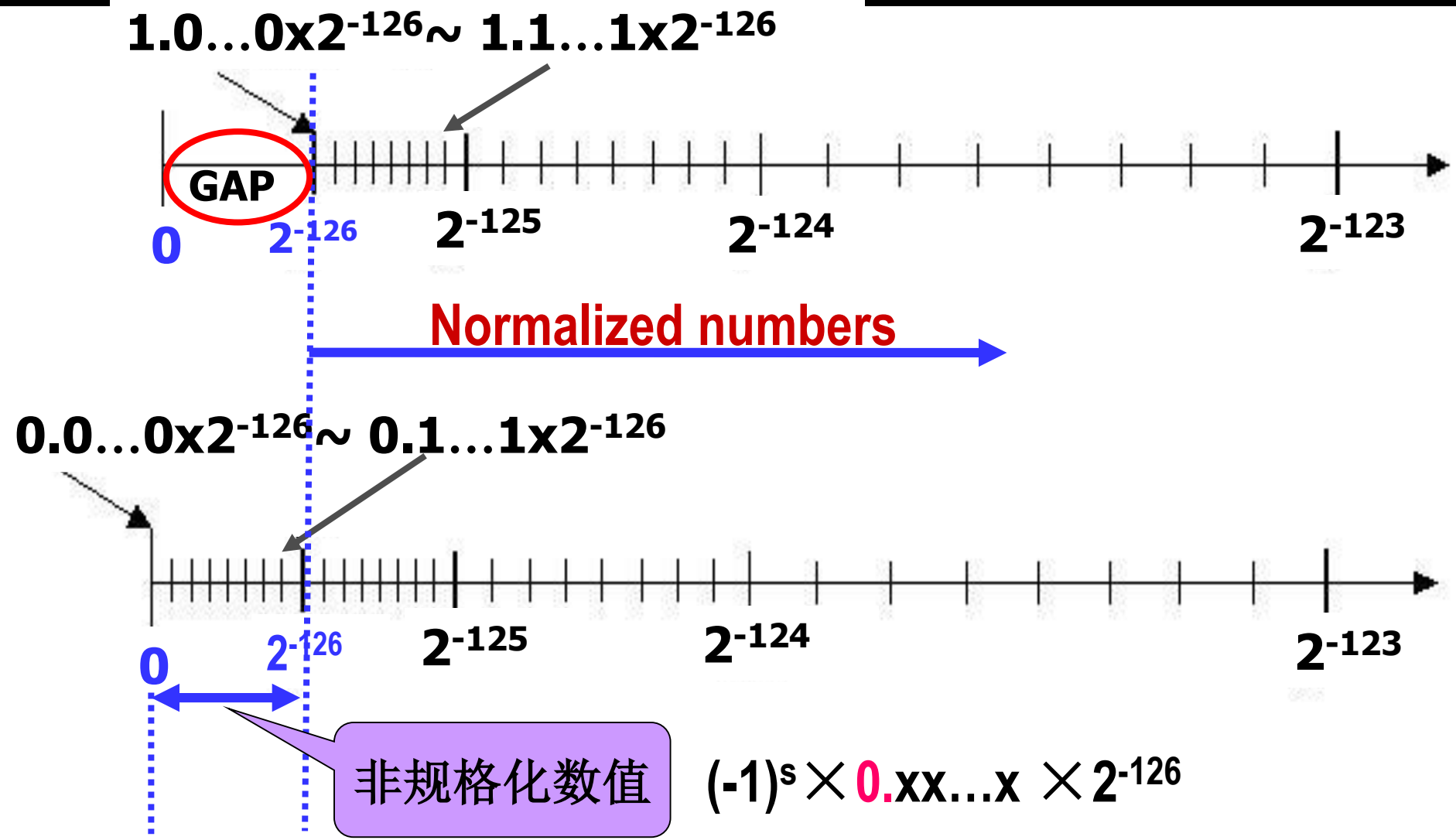
	s	exp	frac	E	Value	
非规格化数值	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	最小非规格化数值
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					数值接近0
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	最大非规格化数值
规格化数值	0	0001	000	-6	8/8*1/64 = 8/512	最小规格化数值
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	数值接近1
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	最大规格化数值
	0	1111	000	n/a	inf	

浮点数表示范围

Description	*exp*	*frac*	*Numeric Value*
• 0	00…00	00…00	+/- 0.0
• 最小的正非规格化数值	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
◦ Single ≈ 1.4×10^{-45}			
◦ Double ≈ 4.9×10^{-324}			
• 最大的非规格化数值	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
◦ Single ≈ 1.18×10^{-38}			
◦ Double ≈ 2.2×10^{-308}			
• 最小的正规格化数值	00…01	00…00	$1.0 \times 2^{-\{126,1022\}}$
• 1	01…11	00…00	1.0
• 最大的规格化数值	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$
◦ Single ≈ 3.4×10^{38}			
◦ Double ≈ 1.8×10^{308}			



单精度非规格化数值范围



浮点数运算及结果

设两个规格化浮点数分别为 $A=M_a \cdot 2^{Ea}$　　$B=M_b \cdot 2^{Eb}$ ，则：

$A \pm B = (M_a \pm M_b \cdot 2^{-(Ea-Eb)}) \cdot 2^{Ea}$　　（假设Ea>=Eb ）

$A*B = (M_a * M_b) \cdot 2^{Ea+Eb}$

$A/B = (M_a / M_b) \cdot 2^{Ea-Eb}$

1.5+1.5=?

1.5-1.0=?

上述运算结果可能出现以下几种情况：

SP最大指数为多少？127！

阶码上溢：一个正指数超过了最大允许值 =〉 +∞/-∞/溢出

阶码下溢：一个负指数超过了最小允许值 =〉 +0/-0

SP最小指数为多少？-126！

尾数溢出：最高有效位有进位 =〉 右规

尾数溢出，结果不一定溢出

非规格化尾数：数值部分高位为0 =〉 左规

右规或对阶时，右段有效位丢失 =〉 尾数舍入

运算过程中添加保护位

IEEE建议实现时为每种异常情况提供一个自陷允许位。若某异常对应的位为1，则发生相应异常时，就调用一个特定的异常处理程序执行。

IEEE754标准规定的五种异常情况

① 无效运算（无意义）

- 运算时有一个数是非有限数，如：

  加 / 减∞、0 x ∞、 ∞/∞等

- 结果无效，如：

  源操作数是NaN、0/0、x REM 0、 ∞ REM y 等

② 除以0（即：无穷大）

③ 数太大（阶码上溢）：对于SP，指阶码 E >1111 1110 （指数大于127）

④ 数太小（阶码下溢） ：对于SP，指阶码 E < 0000 0001（指数小于-126 ）

⑤ 结果不精确（舍入时引起），例如1/3，1/10等不能精确表示成浮点数

上述情况硬件可以捕捉到，因此这些异常可设定让硬件处理，也可设定让软件处理。让硬件处理时，称为硬件陷阱。

整除0和浮点数除0的问题

```
#include <conio.h>
#include <stdio.h>
int    main()
{
        int    a=1,b=0;
        printf( "Division    by    zero:%d\n ",a/b);
        getchar();
        return    0;
}
int    main()
{
        double    x=1.0,y=-1.0,z=0.0;
        printf( "division    by    zero:%f    %f\n ",x/z,y/z);
        getchar();
        return    0;
}
```

为什么整数除0会发生异常而浮点数除0不会？

问题一：为什么整除int型会产生错误？是什么错误？

二：用double型的时候结果为1.#INF00和-1.#INF00，作何解释???

浮点数加/减运算

- 十进制科学计数法的加法例子

  $0.123 \times 10^5 + 0.560 \times 10^2$

  其计算过程为：

  $0.123 \times 10^5 + 0.560 \times 10^2 = 0.123 \times 10^5 + 0.000560 \times 10^5$

  $= (0.123 + 0.00056) \times 10^5 = 0.12356 \times 10^5 = 0.124 \times 10^5$

  进行尾数加减运算前，必须“对阶”！最后还要考虑舍入

  计算机内部的二进制运算也一样！

- “对阶”操作：目的是使两数阶码相等
  - 小阶向大阶看齐，阶小的那个数的尾数右移，右移位数等于两个阶码差的绝对值
  - IEEE 754尾数右移时，要将隐含的“1”移到小数部分，高位补0，移出的低位保留到特定的“附加位”上

浮点数加减法基本要点

（假定：Xm、Ym分别是X和Y的尾数，　Xe和Ye 分别是X和Y的阶码 ）

(1)　求阶差：Δe=Ye － Xe （若Ye ＞ Xe，则结果的阶码为Ye）

(2)　对阶：将Xm右移Δe位，尾数变为 $Xm*2^{Xe-Ye}$（保留右移部分：附加位）

(3)　尾数加减：　$Xm*2^{Xe-Ye}$ ± Ym

(4) 规格化：

当尾数高位为0，则需左规：尾数左移一次，阶码减1，直到MSB为1

每次阶码减1后要判断阶码是否下溢（比最小可表示的阶码还要小）

当尾数最高位有进位，需右规：尾数右移一次，阶码加1，直到MSB为1

每次阶码加1后要判断阶码是否上溢（比最大可表示的阶码还要大）

阶码溢出异常处理：阶码上溢，则结果溢出；阶码下溢，则结果为0

(5) 如果尾数比规定位数长，则需考虑舍入（有多种舍入方式）

(6) 若运算结果尾数是0，则需要将阶码也置0。为什么？

尾数为0说明结果应该为0（阶码和尾数为全0）。

浮点数加法运算举例

Example：用二进制形式计算 0.5 +(- 0.4375) =?

解：$0.5 = 1.000 \times 2^{-1}$，$-0.4375 = -1.110 \times 2^{-2}$

对　　阶：$-1.110 \times 2^{-2} \rightarrow -0.111 \times 2^{-1}$

加　　减：$1.000 \times 2^{-1} + (-0.111 \times 2^{-1}) = 0.001 \times 2^{-1}$

左　　规：$0.001 \times 2^{-1} \rightarrow 1.000 \times 2^{-4}$

判溢出：　无

结果为：$1.000 \times 2^{-4} = 0.0001000 = 1/16 = 0.0625$

问题：为何IEEE 754 加减运算右规时最多只需一次？

因为即使是两个最大的尾数相加，得到的和的尾数也不会达到4，故尾数的整数部分最多有两位，保留一个隐含的“1”后，最多只有一位被右移到小数部分。

Rounding Digits(舍入位)

举例：十进制数，最终有效位数为 3，采用两位附加位（G、R）。

问题：若没有舍入位，采用就近舍入到偶数，则结果是什么？

结果为2.36！精度没有2.37高！

$$
\begin{array}{r}
2.3400 * 10^2 \\
0.0253 * 10^2 \\
\hline
2.3653 * 10^2
\end{array}
$$

IEEE Standard: four rounding modes（用图说明）

round to nearest (default)
round towards plus infinity (always round up)
round towards minus infinity (always round down)
round towards 0

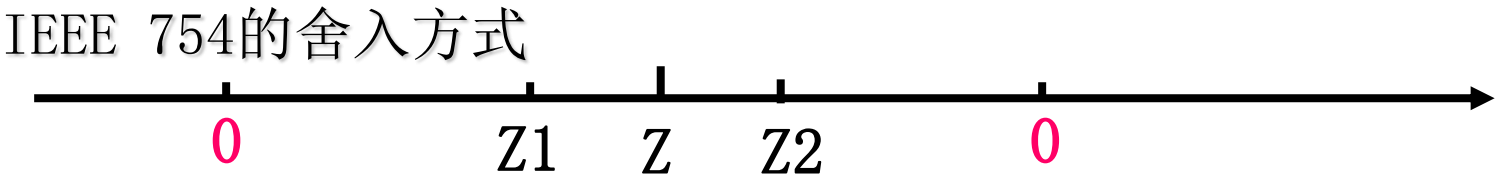
注：ULP=units in the last place.

round to nearest: 简称为就近舍入到偶数

round digit < 1/2 then truncate (截去)
> 1/2 then round up (add 1 to ULP)
= 1/2 then round to nearest even digit

可以证明默认方式得到的平均误差最小。

IEEE 754的舍入方式的说明



（ Z1和Z2分别是结果Z的最近的可表示的左、右两个数 ）

(1) 就近舍入：舍入为最近可表示的数

非中间值：0舍1入；

中间值：强迫结果为偶数-慢

例如：附加位为
01：舍
11：入
10：（强迫结果为偶数）

例：1.110111 → 1.1110;　　1.110101 → 1.1101;
1.110110 → 1.1110;　　1.111110 → 10.0000;

(2) 朝+∞方向舍入:舍入为Z2(正向舍入)

(3) 朝-∞方向舍入:舍入为Z1(负向舍入)

(4) 朝0方向舍入：截去。正数：取Z1;　负数：取Z2

浮点数舍入举例

例：将同一实数分别赋值给单精度和双精度类型变量，然后打印输出。

```
#include <stdio.h>
main()
{
        float a;
        double b;
        a = 123456.789e4;
        b = 123456.789e4;
        printf(“%f/n%f/n”,a,b);
}
```

运行结果如下：

```
        1234567936.000000
        1234567890.000000
```

为什么float情况下输出的结果会比原来的大？这到底有没有根本性原因还是随机发生的？为什么会出现这样的情况？

问题：为什么同一个实数赋值给float型变量和double型变量，输出结果会有所不同呢？

C语言中的浮点数类型

- C语言中有float和double类型，分别对应IEEE 754单精度浮点数格式和双精度浮点数格式
- long double类型的长度和格式随编译器和处理器类型的不同而有所不同，IA-32中是80位扩展精度格式
- 从int转换为float时，不会发生溢出，但可能有数据被舍入
- 从int或 float转换为double时，因为double的有效位数更多，故能保留精确值
- 从double转换为float和int时，可能发生溢出，此外，由于有效位数变少，故可能被舍入
- 从float 或double转换为int时，因为int没有小数部分，所以数据可能会向0方向被截断

Questions about IEEE 754

- What's the range of representable values?

  The largest number for single: +1.11…1X 2^{127}

  How about double?

  约 **+1.8 x 10^{308}**

  约 **+3.4 x 10^{38}**

- What about following type converting: not always true!

```
if ( i == (int) ((float) i) )  {
        printf ("true");
}
if ( f == (float) ((int) f) )  {
        printf ("true");
}
```

How about double? **True!**

How about double? **Not always true!**

- How about FP add associative? FALSE!

  x = – 1.5 x 10^{38}, y = 1.5 x 10^{38}, z = 1.0

  (x+y)+z = (–1.5x10^{38}+1.5x10^{38}) +1.0 = 1.0

  x+(y+z) = –1.5x10^{38}+ (1.5x10^{38}+1.0) = 0.0

浮点运算举例

- 对于以下给定的关系表达式，判断是否永真。

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

Assume neither d nor f is NaN

- x == (int)(float) x 否
- x == (int)(double) x 是
- f == (float)(double) f 是
- d == (float) d 否
- f == -(-f) 是
- 2/3 == 2/3.0 否
- d < 0.0 ⇒ ((d*2) < 0.0) 是
- d > f ⇒ -f > -d 是
- d * d >== 0.0 是
- (d+f)-d == f 否

自己写程序测试一下！

浮点运算举例

- 1996年6月4日，Ariana 5火箭初次航行，在发射仅仅37秒钟后，偏离了飞行路线，然后解体爆炸，火箭上载有价值5亿美元的通信卫星。
- 原因是在将一个64位浮点数转换为16位带符号整数时，产生了溢出异常。溢出的值是火箭的水平速率，这比原来的Ariana 4火箭所能达到的速率高出了5倍。在设计Ariana 4火箭软件时，设计者确认水平速率决不会超出一个16位的整数，但在设计Ariana 5时，他们没有重新检查这部分，而是直接使用了原来的设计。
- 在不同数据类型之间转换时，往往隐藏着一些不容易被察觉的错误，这种错误有时会带来重大损失，因此，编程时要非常小心。

浮点运算举例

- 1991年2月25日，海湾战争中，美国在沙特阿拉伯达摩地区设置的爱国者导弹拦截伊拉克的飞毛腿导弹失败，致使飞毛腿导弹击中了沙特阿拉伯载赫蓝的一个美军军营，杀死了美国陆军第十四军需分队的28名士兵。其原因是由于爱国者导弹系统时钟内的一个软件错误造成的，引起这个软件错误的原因是浮点数的精度问题。
- 爱国者导弹系统中有一个内置时钟，用计数器实现，每隔0.1秒计数一次。程序用0.1的一个24位定点二进制小数x来乘以计数值作为以秒为单位的时间。
- 0.1的二进制表示是一个无限循环序列：0.00011[0011]…，x=0.000 1100 1100 1100 1100 1100B。显然，x是0.1的近似表示，0.1-x

  = 0.000 1100 1100 1100 1100 1100 [1100]… -

  0.000 1100 1100 1100 1100 1100B，即为：

  =0.000 0000 0000 0000 0000 0000 1100 [1100]…B

  $=2^{-20}\times 0.1 \approx 9.54\times 10^{-8}$

浮点运算举例

已知在爱国者导弹准备拦截飞毛腿导弹之前，已经连续工作了100小时，飞毛腿的速度大约为2000米/秒，则由于时钟计算误差而导致的距离误差是多少？

100小时相当于计数了$100\times60\times60\times10=36\times10^5$次，因而导弹的时钟已经偏差了$9.54\times10^{-8}\times36\times10^5\approx0.343$秒

因此，距离误差是2000×0.343秒 ≈ 687米

小故事：实际上，以色列方面已经发现了这个问题并于1991年2月11日知会了美国陆军及爱国者计划办公室（软件制造商）。以色列方面建议重新启动爱国者系统的电脑作为暂时解决方案，可是美国陆军方面却不知道每次需要间隔多少时间重新启动系统一次。1991年2月16日，制造商向美国陆军提供了更新软件，但这个软件最终却在飞毛腿导弹击中军营后的一天才运抵部队。

浮点运算举例

- 若x用float型表示，则x的机器数是什么？0.1与x的偏差是多少？系统运行100小时后的时钟偏差是多少？在飞毛腿速度为2000米/秒的情况下，预测的距离偏差为多少？
  - 0.1= 0.0 0011[0011]B=+1.1 0011 0011 0011 0011 0011 00B×2^{-4}，故x的机器数为0 011 1101 1 100 1100 1100 1100 1100 1100
  - Float型仅24位有效位数，后面的有效位全被截断，故x与0.1之间的误差为：|x－0.1|=0.000 0000 0000 0000 0000 0000 0000 1100 [1100]…B。这个值等于2^{-24}×0.1 ≈ 5.96×10^{-9}。100小时后时钟偏差5.96×10^{-9}×36×10^{5} ≈ 0.0215秒。距离偏差0.0215×2000≈43米。比爱国者导弹系统精确约16倍。
- 若用32位二进制定点小数x=0.000 1100 1100 1100 1100 1100 1100 1101 B表示0.1，则误差比用float表示误差更大还是更小？
  - 当x=0.000 1100 1100 1100 1100 1100 1100 1101 B时，与0.1之间的误差约为：|x－0.1|=0.000 0000 0000 0000 0000 0000 0000 0000 00 1100 [1100]…B。这个值等于2^{-30}×0.1 ≈ 9.31×10^{-11}。100小时后时钟偏差9.31×10^{-11}×36×10^{5} ≈ 0.000335秒。预测的距离偏差仅为0.000335×2000 ≈ 0.67米。

浮点运算举例

- 从上述结果可以看出：
  - 用32位定点小数表示0.1 ，比采用float精度高64倍
  - 用float表示在计算速度上更慢，必须先把计数值转换为IEEE 754格式浮点数，然后再对两个IEEE 754格式的数相乘，故采用float比直接将两个二进制数相乘要慢得多
- Ariana 5火箭和爱国者导弹的例子带来的启示

✓ 程序员应对底层机器级数据的表示和运算有深刻理解

✓ 计算机世界里，经常是“差之毫厘，失之千里”，需要细心再细心，精确再精确

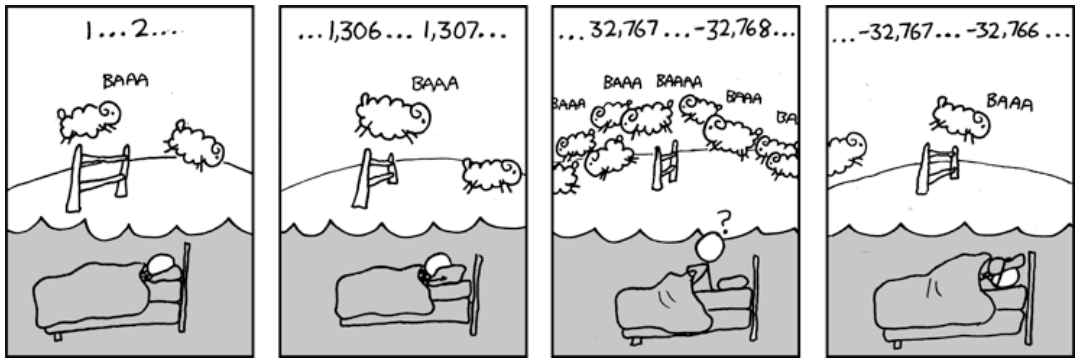
✓ 不能遇到小数就用浮点数表示，有些情况下（如需要将一个整数变量乘以一个确定的小数常量），可先用一个确定的定点整数与整数变量相乘，然后再通过移位运算来确定小数点

二进制数据的度量单位

- 位/比特（bit）是二进制信息的最小单位。
- 二进制信息存储的计量单位
  - 字节(Byte)， 1B=8bits
  - 字（Word)，1W=2B=16bits
  - 千字节(KB)，1KB=2^{10}字节=1024B
  - 兆字节(MB)，1MB=2^{20}字节=1024KB
  - 吉字节 (GB)，1GB=2^{30}字节=1024MB
  - 太字节 (TB)，1TB=2^{40}字节=1024GB
  - 拍字节 (PB)，1PB=2^{50}字节=1024TB

数据表示的局限性

- *Int’s 不是整数，Float’s不是实数*



- 举例
  - $x^2 \geqslant 0$?
    - Float’s: 是!
    - Int’s:
      - 40000 * 40000 --> 1600000000
      - 60000 * 60000 -694967296
  - (x + y) + z = x + (y + z)?
    - Unsigned & Signed Int’s: 是!
    - Float’s:
      - (1e20 + -1e20) + 3.14 --> 3.14
      - 1e20 + (-1e20 + 3.14) --> **0**

课后作业

- 练习题：2.10，2.11，2.14，2.23，2.35，2.47，2.52
- 课后作业：2.63，2.75，2.81，2.86，2.90