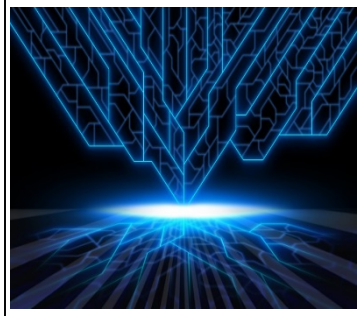
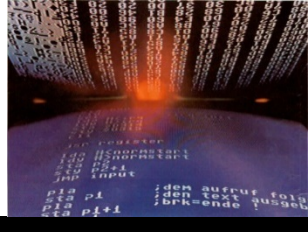
第4讲
数据的表示



吴海军

南京大学计算机科学与技术系

主要内容



- 数据的表示
  - 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

不同抽象层次中的数据表示

感觉媒体信息 ↔ 问题(应用)

↓ ↓

树、链表等结构化数据描述 ↔ 算法

↓ ↓

int, float, array, struct等类型 ↔ 程序(语言)

↓ ↓

指令指定寄存器或内存中数据 ↔ 指令集体系结构(ISA)

↓ ↓

ALU中运算或总线上传输的数据 ↔ 微体系结构

↓ ↓

逻辑门 ↔ 电路

↓ ↓

位信息 ↔ 器件(晶体管)

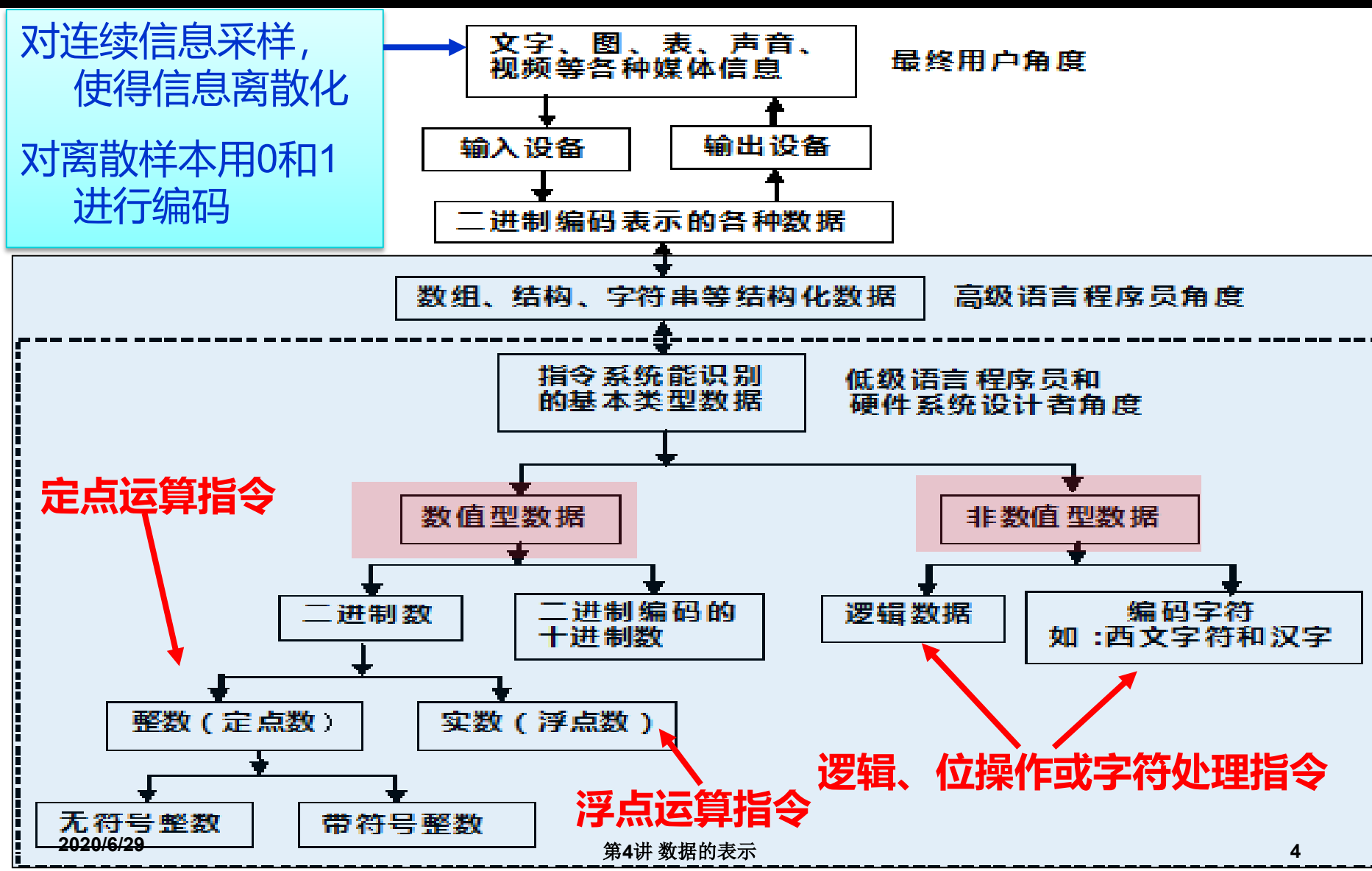
抽象概括

具体实现

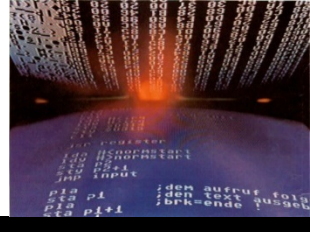
数据的表示

对连续信息采样，
使得信息离散化

对离散样本用0和1
进行编码



数值数据的表示



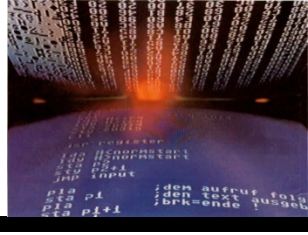
- 数值数据表示的三要素：进位数制、定/浮点表示、编码格式

  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：不确定！**

  - **进位数制**（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）

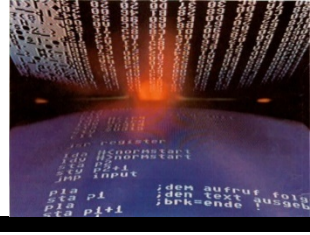
定点数和浮点数



- 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

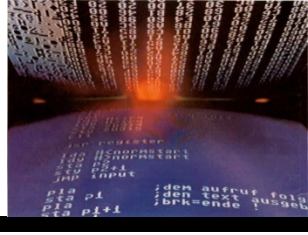


- 正负数如何表示？
- 用n位R进制数表示R^n个不同的正负整数值。
- 符号-数值表示法
  - 原码
  - 补码
  - 反码
- 二进制数加减运算

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：
  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，大于R^{n-1}的数表示负数。
- $D+D_{补}=R^n$，　　$0 \le D \le R^{n-1}-1$
  - 因为只有n位，最高位溢出，不能保存，则结果为0
  - $D+D_{补}=D-D=0$
  - $D_{补}=-D$

补码定义：n位二进制数，则：

$[X]_{补}=2^n+X \quad (-2^{n-1} \leqslant X < 2^{n-1}, \mathrm{mod}\ 2^n)$

二进制补码表示

补码定义：n位二进制数，则：

$[X]_{补}=2^n+X \quad (-2^{n-1}\leqslant X<2^{n-1}, \mathrm{mod}\ 2^n)$

- 二进制补码的性质：
  - 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} ～$2^{n-1}-1$
- 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：模减该负数的绝对值或按位取反再加1。

$$[x+y]_{补}=2^n+x+y=2^n+x+2^n+y=[x]_{补}+[y]_{补} \ (\mathrm{mod}\ 2^n)$$
$$[x-y]_{补}=2^n+x-y=2^n+x+2^n-y=[x]_{补}+[-y]_{补} \ (\mathrm{mod}\ 2^n)$$

符号位直接运算、0表示唯一、加减法统一。
现代计算机系统中整数都采用补码来表示！

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）

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**

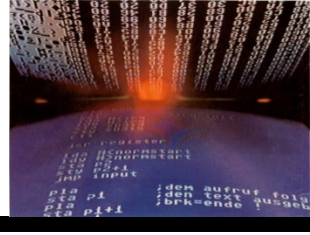
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

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

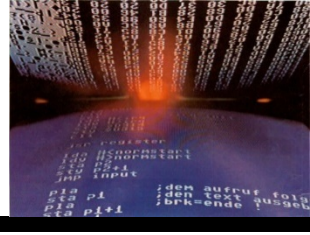


编码	无符号	原码	补码	反码	编码	无符号	原码	补码	反码
0000	0	0	0	0	1000	8	-0	-8	-7
0001	1	1	1	1	1001	9	-1	-7	-6
0010	2	2	2	2	1010	A	-2	-6	-5
0011	3	3	3	3	1011	B	-3	-5	-4
0100	4	4	4	4	1100	C	-4	-4	-3
0101	5	5	5	5	1101	D	-5	-3	-2
0110	6	6	6	6	1110	E	-6	-2	-1
0111	7	7	7	7	1111	F	-7	-1	-0

在带符号数编码中最高位为符号位

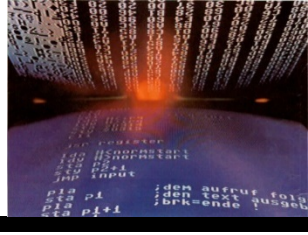
无符号数表示范围：0 ～2^n-1　　补码表示范围：-2^{n-1} ～$2^{n-1}-1$

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
- 无符号整数的编码中没有符号位。
- 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？

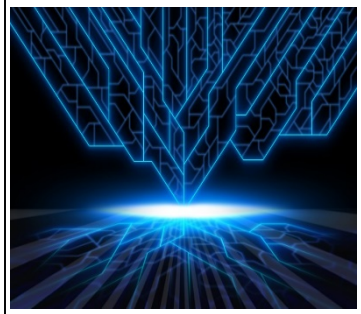
关系表达式	类型	结果	说明
0 == 0U	无符号	1	00…0B = 00…0B
-1 < 0	带符号	1	11…1B (-1) < 00…0B (0)
-1 < 0U	无符号	0*	11…1B (2^{32}-1) > 00…0B(0)
2147483647 > -2147483647-1	带符号	1	011…1B (2^{31}-1) > 100…0B (-2^{31})
2147483647U > -2147483647-1	无符号	0*	011…1B (2^{31}-1) < 100…0B(2^{31})
2147483647 > (int) 2147483648U	带符号	1*	011…1B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

非数值数据的表示

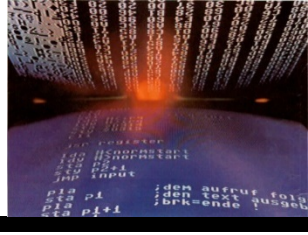
逻辑值

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。

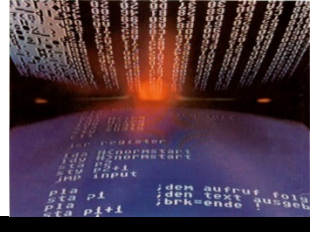
逻辑数据的编码表示

- 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

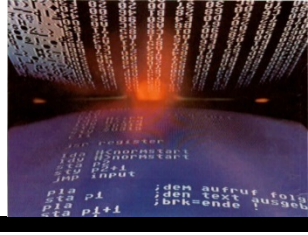


ASCII（美国标准信息交换码）字符表

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

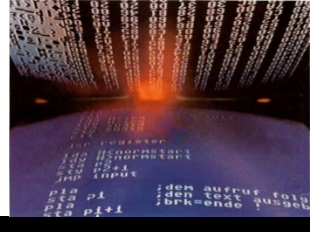


- GB2312-80编码：我国最早（1981年）发布的简体中文汉字编码国家标准。对汉字采用双字节编码，收录7445个字符，其中包括6763个汉字。

- GBK编码：1995年发布的汉字编码国家标准，是对GB2312编码的扩充。GBK字符集共收录21886个汉字和图形符号，包含中日韩和台湾地区繁体字在内的所有汉字。

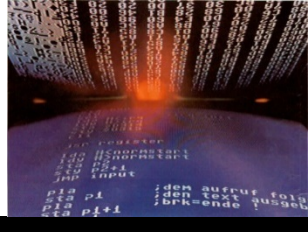
- GB18030-2005编码：是最新的汉字编码国家标准，对 GB 18030-2000的补充。与GB 2312-80和GBK兼容，共收录76556个字符，其中汉字70244个。覆盖简繁中文、日文、朝鲜语和中国少数民族文字，对汉字采用一、二、四字节变长编码。

Unicode编码



- ISO 10646标准所定义的标准字符集-通用字符集（Universal Character Set，UCS）。Unicode 是基于通用字符集的标准来发展。
- Unicode 编码系统可分为**编码方式**和**实现方式**两个层次。
  - 编码方式：UCS-2用两个字节编码，UCS-4用4个字节编码。
  - 实现方式：即UCS字符集转换格式(UCS Transformation Format，UTF)，将UCS的编码转换成为程序数据。UTF-8、UTF-16、UTF-32都是将UCS编码数字转换到程序数据的编码方案。

UTF-8编码

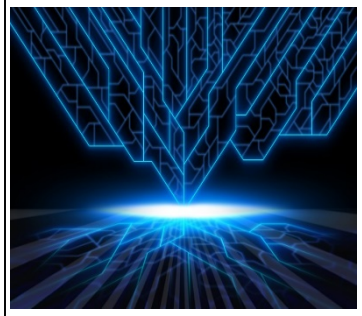


- UTF-8以字节为单位对Unicode进行编码。 对不同范围的字符使用不同长度的编码。对于0x00-0x7F之间的字符，UTF-8编码与ASCII编码完全相同。
- 从Unicode到UTF-8的编码方式如下：

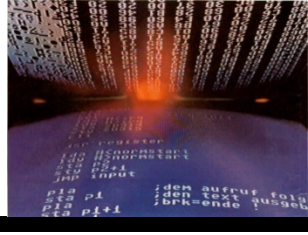
Unicode编码(16进制)	UTF-8 字节流(二进制)
000000 - 00007F	0xxxxxxx
000080 - 0007FF	110xxxxx 10xxxxxx
000800 - 00FFFF	1110xxxx 10xxxxxx 10xxxxxx
010000 - 10FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

- UTF-8编码的最大长度是4个字节。
- “汉”字的Unicode编码是0x6C49。0x6C49在0x0800-0xFFFF之间，使用3字节模板了：1110xxxx 10xxxxxx 10xxxxxx。将0x6C49写成二进制是：0110 1100 0100 1001， 用这个比特流依次代替模板中的x，得到：11100110 10110001 10001001，即E6 B1 89。

信息存储

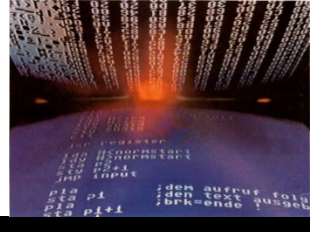


信息处理的单位



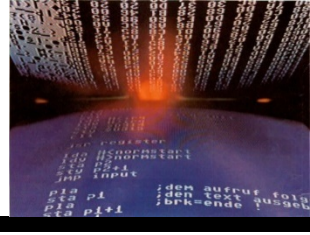
- 比特（bit）是计算机中处理信息的最小单位。
  - 字节(Byte)， 1B=8bits
  - 字（Word)，1W=2B=16bits
  - 千字节(KB)，1KB=2^{10}字节=1024B
  - 兆字节(MB)，1MB=2^{20}字节=1024KB
  - 吉字节 (GB)，1GB=2^{30}字节=1024MB
  - 太字节 (TB)，1TB=2^{40}字节=1024GB
  - 拍字节 (PB)，1PB=2^{50}字节=1024TB
- 存储二进制信息时的度量单位要比字节或字大得多

信息存储



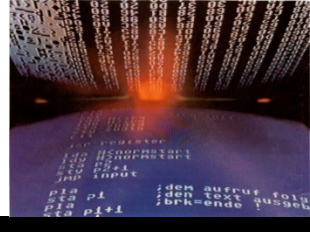
- 二进制信息存储的计量单位是“字节”(Byte)，也称“位组”
  - 现代计算机中，存储器按字节编址
  - 字节是最小可寻址单位（addressable unit）
- 所有的可能寻址的集合称为寻址空间。
- 系统中最大可寻址空间取决于**地址总线**的宽度。
- IA-32最大可寻址空间2^{32}B=4GB。

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 (字节)

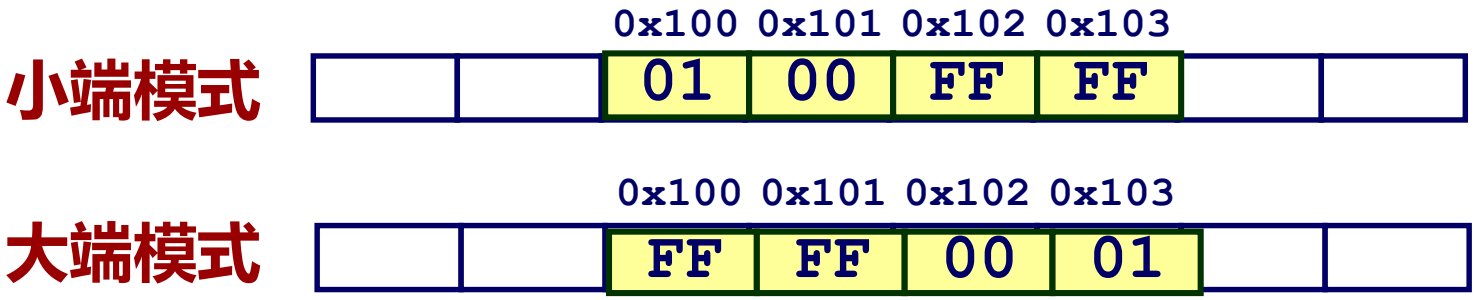
C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

- 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int x=-65535，存放地址从 0x100开始，则数据存放格式有：

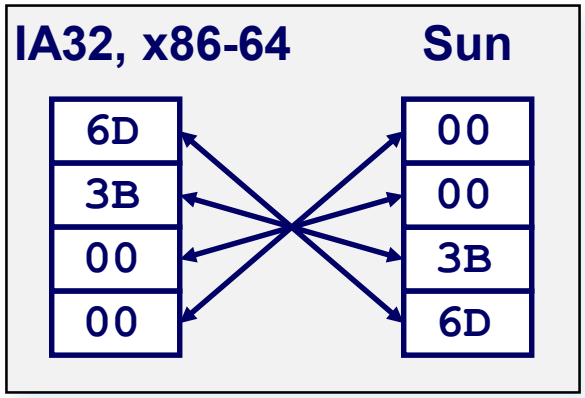


- 小端模式Little **Endian**：最低有效字节**LSB**存放在最小地址单元的方式。如：**Intel 80x86, DEC VAX**
- 大端模式 **Big Endian**：最高有效字节**MSB**存放在最小地址单元的方式。 如：**IBM 360/370, SUN, MIPS, Sparc, HP PA,Internet**

注意：字节内部的排列顺序也是小端模式。

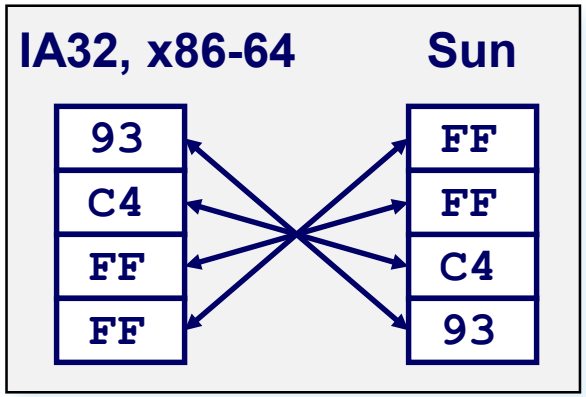
大端、小端转换

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。

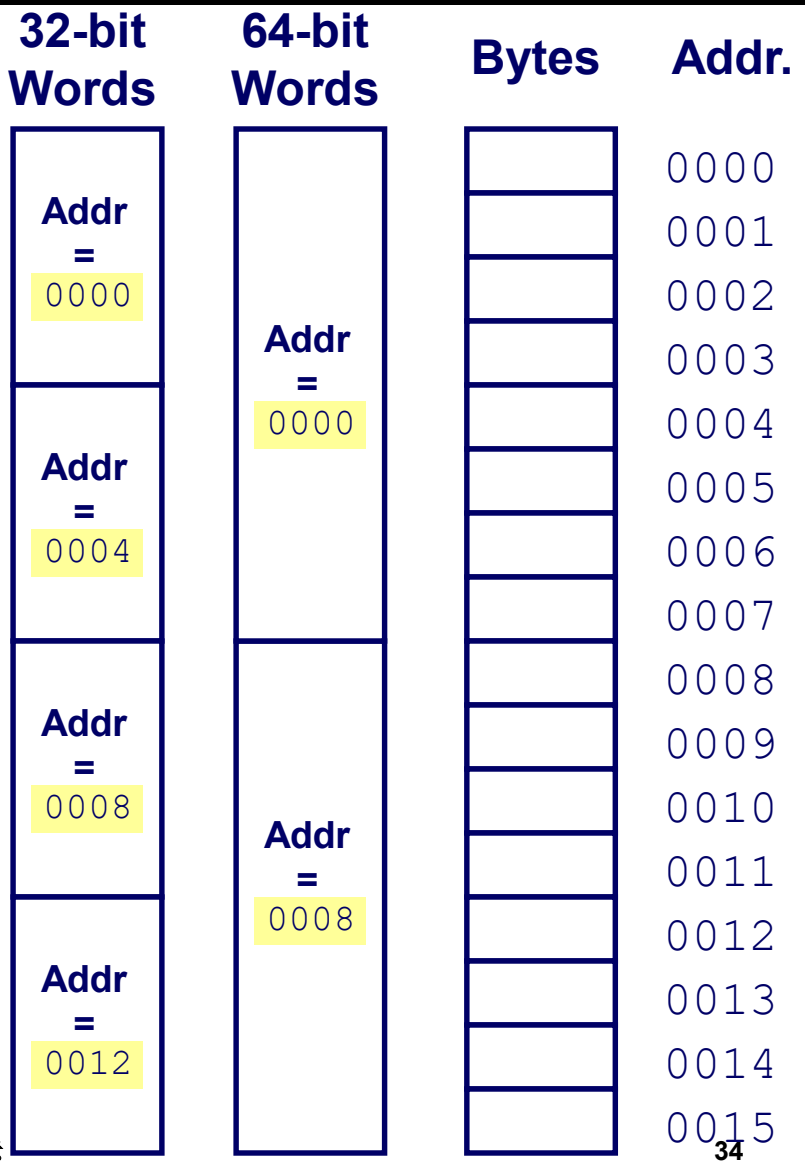
- 每个系统内部次序是一致的，但在系统间通信时可能会发生问题！
- 因为顺序不同，需要进行顺序转换
- 音、视频和图像等文件格式或处理程序都涉及到字节顺序问题

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

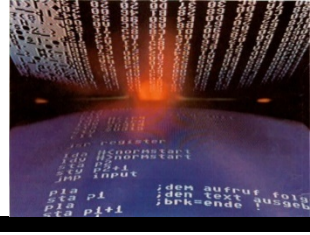
Big endian: Adobe Photoshop, JPEG, MacPaint, etc

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。
- 32位：每隔4个字节进行编码。
- 64位：每隔8个字节进行编码。



存储边界对齐



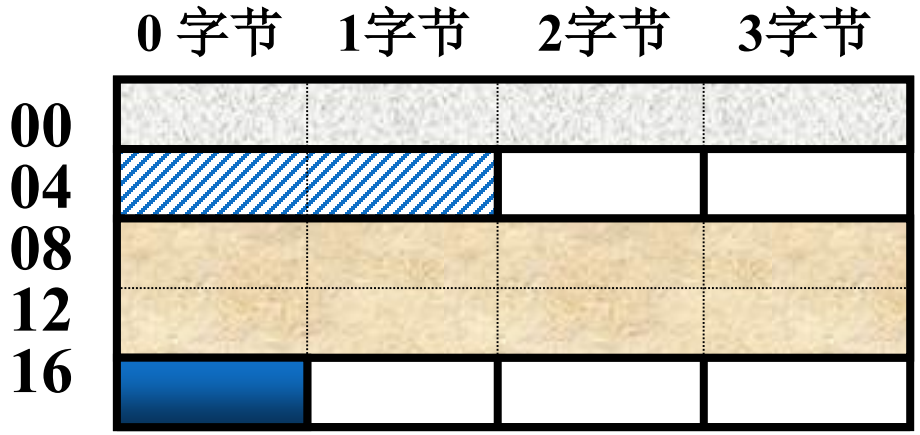
- 指令系统支持对字节、半字、字及双字的运算
- 各种不同长度的数据存放时，有两种处理方式：
  - 按边界对齐 （假定字长的宽度为32位，按字节编址）
    - 双字地址：8的倍数(地址的最低三位为0)
    - 字地址：4的倍数(地址的最低两位为0)
    - 半字地址：2的倍数(地址的末位为0)
    - 字节地址：任意
  - 不按边界对齐：顺序存放。

    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（4的倍数）开始的连续的1个、2个、3个或4个字节

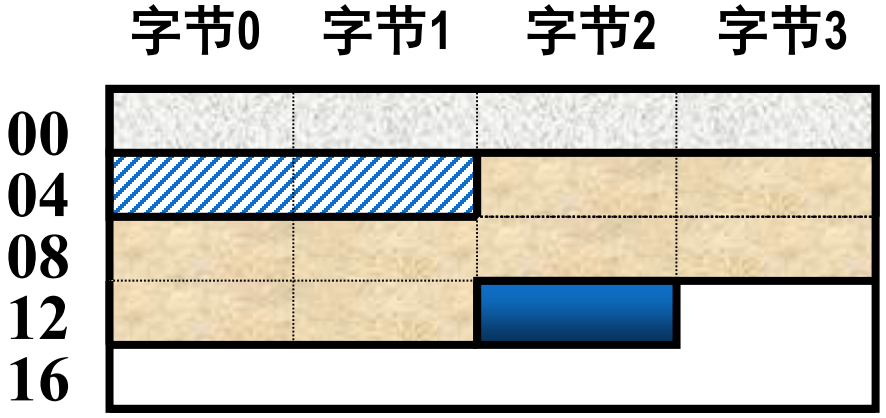
- 如：int i, short k, double x, char c



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！

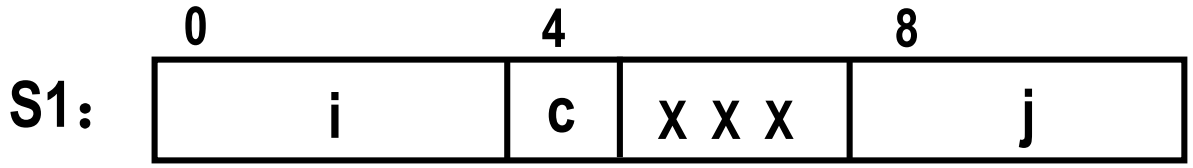
存储边界对齐

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

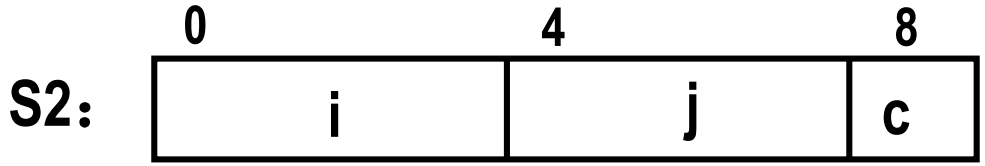
```
struct  S2 {
        int     i;
        int     j;
        char    c;
};
```

在结构类型中，按数据类型的长度，从高到低排列！

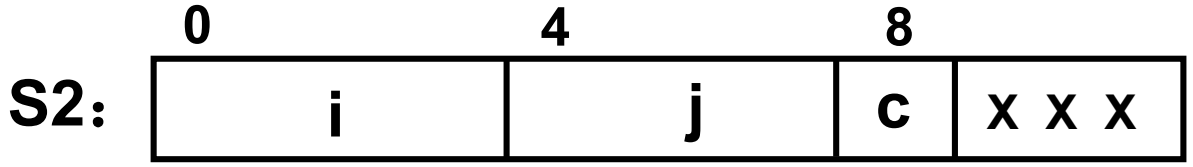
在要求对齐的情况下，哪种结构声明更好？



需要12个字节



只需要9个字节　S2比S1好



对于“struct S2 d[4]”，只分配9个字节能否满足对齐要求？