Bits, Bytes and Integers – Part 1

15-213/18-213/15-513: Introduction to Computer Systems
2nd Lecture, Aug. 31, 2017

Today's Instructor:

Randy Bryant

数据的表示和运算(chapter2,课件2-4)

- 分三个部分介绍
  - 非数值数据的表示、数据的存储
    - 数据宽度单位
    - 硬件特征：大端/小端、对齐存放
  - 数值数据的表示
    - 定点数的编码表示
    - 整数的表示

      无符号整数、带符号整数
    - 浮点数的表示
    - C语言程序的整数类型和浮点数类型

数据的表示和运算

- 分三个部分介绍（续）
  - 数据的运算
    - 按位运算和逻辑运算
    - 移位运算
    - 位扩展和位截断运算
    - 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
    - 浮点数的加减乘除运算

围绕C语言中的运算，解释其在底层机器级的实现方法

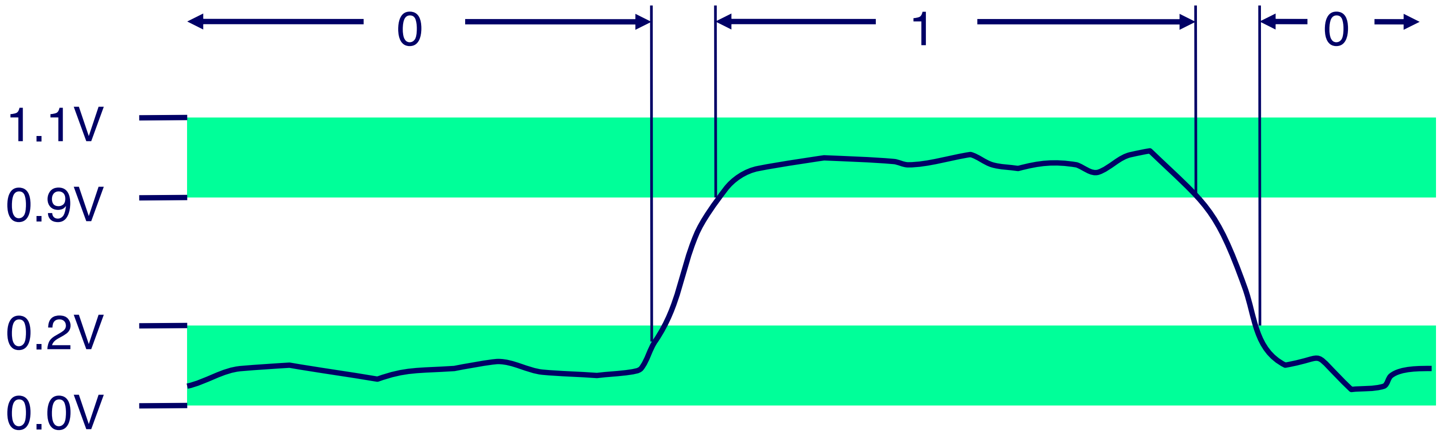
从高级语言程序中的表达式出发，用机器数在具体电路中的执行过程，来解释表达式的执行结果

Today: Bits, Bytes, and Integers

- **Representing information as bits**
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Everything is bits

- **Each bit is 0 or 1**
- **By encoding/interpreting sets of bits in various ways**
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- **Why bits? Electronic Implementation**
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



For example, can count in binary

- **Base 2 Number Representation**
  - Represent 15213_{10} as 11101101101101_2
  - Represent 1.20_{10} as $1.0011001100110011[0011]..._2$
  - Represent 1.5213×10^4 as $1.1101101101101_2 \times 2^{13}$

Encoding Byte Values

- **Byte = 8 bits**
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write $FA1D37B_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b

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

$$\texttt{15213:}\quad \underbrace{\texttt{0011}}_{\texttt{3}}\ \underbrace{\texttt{1011}}_{\texttt{B}}\ \underbrace{\texttt{0110}}_{\texttt{6}}\ \underbrace{\texttt{1101}}_{\texttt{D}}$$

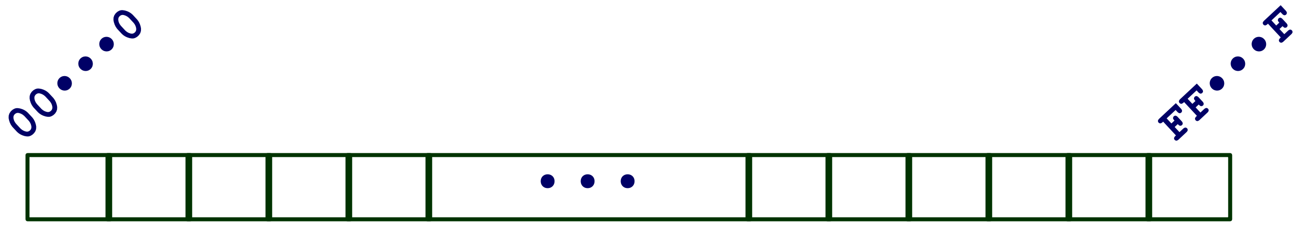
数据量的度量单位

- 存储二进制信息时的度量单位要比字节或字大得多
- 容量经常使用的单位有：
  - “千字节”(KB)，1KB=2^{10}字节=1024B
  - “兆字节”(MB)，1MB=2^{20}字节=1024KB
  - “千兆字节”(GB)，1GB=2^{30}字节=1024MB
  - “兆兆字节”(TB)，1TB=2^{40}字节=1024GB
- 通信中的带宽使用的单位有：
  - “千比特/秒”(kb/s)，1kbps=10^3 b/s=1000 bps
  - “兆比特/秒”(Mb/s)，1Mbps=10^6 b/s =1000 kbps
  - “千兆比特/秒”(Gb/s)，1Gbps=10^9 b/s =1000 Mbps
  - “兆兆比特/秒”(Tb/s)，1Tbps=10^{12} b/s =1000 Gbps

如果把b换成B，则表示字节而不是比特（位）

例如，10MBps表示 10兆字节/秒

Byte-Oriented Memory Organization



- **Programs refer to data by address**
  - Conceptually, envision it as a very large array of bytes
    - In reality, it's not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address

- **Note: system provides private address spaces to each "process"**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

数据的基本宽度

- “字”和 “字长”的概念不同
  - “字长”指数据通路的宽度。

    （数据通路指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，这些部件的宽度基本上要一致，才能相互匹配。因此，”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”表示被处理信息的单位，用来度量数据类型的宽度。
  - 字和字长的宽度可以一样，也可不同。

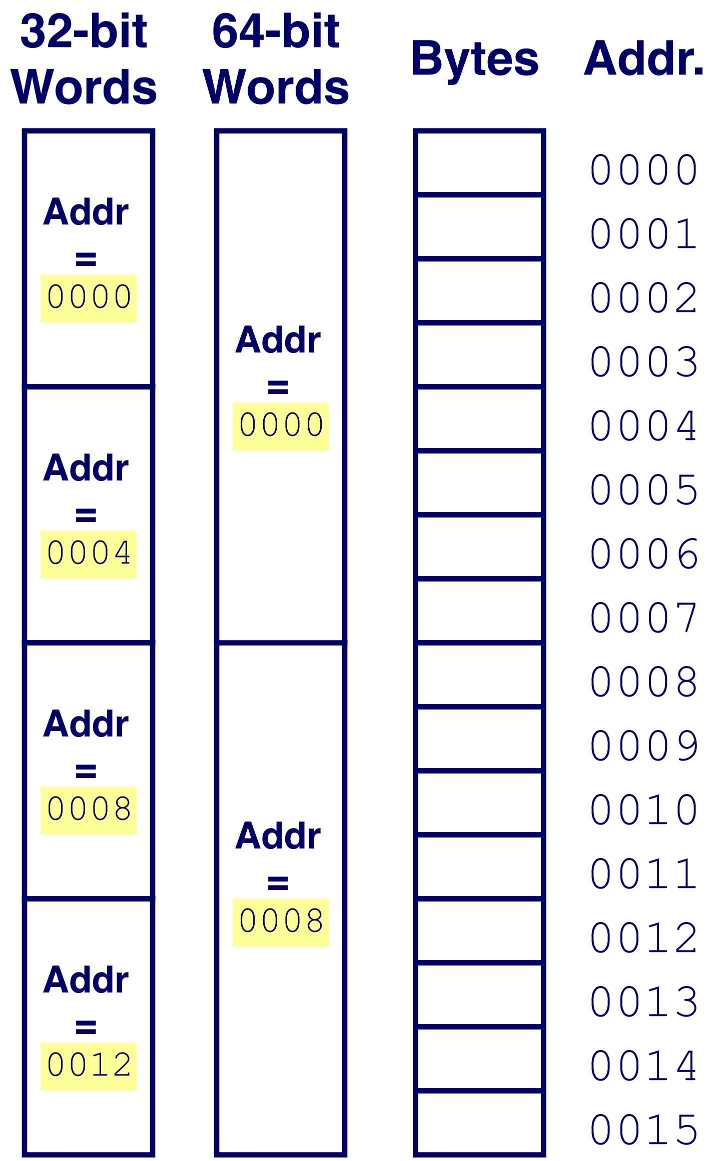
    例如，x86体系结构定义“字”的宽度为16位，但从386开始字长就是32位了。

Machine Words

- **Any given computer has a "Word Size"**
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
    - That's 18.4 X 10^{18}
  - Machines still support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes

Word-Oriented Memory Organization

- **Addresses Specify Byte Locations**
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



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
pointer	4	8	8

- **So, how are the bytes within a multi-byte word ordered in memory?**

数据的存储和排列顺序

- 80年代开始，几乎所有机器都用字节编址
- ISA设计时要考虑的两个问题：
  - 如何根据一个字节地址取到一个32位的字？ - 字的存放问题
  - 一个字能否存放在任何字节边界？ - 字的边界对齐问题

例如，若 int i = -65535，存放在内存100号单元（即占100#～103#），则用“取数”指令访问100号单元取出 i 时，必须清楚 i 的4个字节是如何存放的。

$65535=2^{16}-1$

$[-65535]_{补}=FFFF0001H$

	FF 103	FF 102	00 101	01 100	little endian word 100#
Word:	msb			lsb	
	100	101	102	103	big endian word 100#

大端方式（Big Endian）：MSB所在的地址是数的地址

e.g. IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

小端方式（ Little Endian）：LSB所在的地址是数的地址

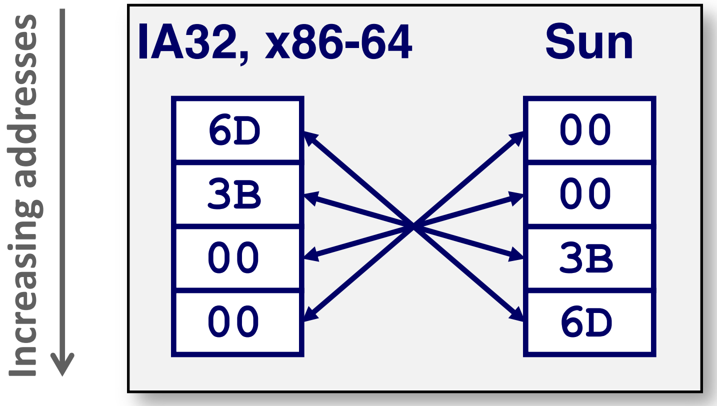
e.g. Intel 80x86, DEC VAX

有些机器两种方式都支持，可通过特定控制位来设定采用哪种方式。

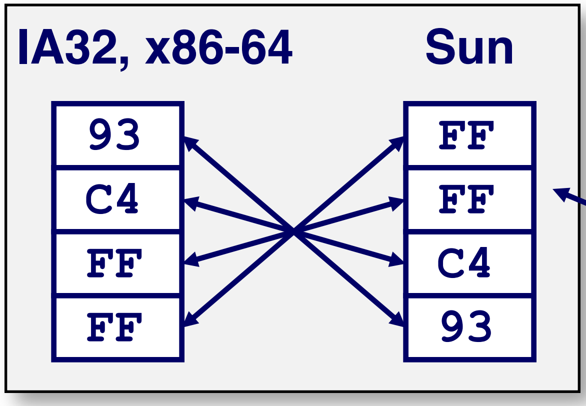
Representing Integers

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

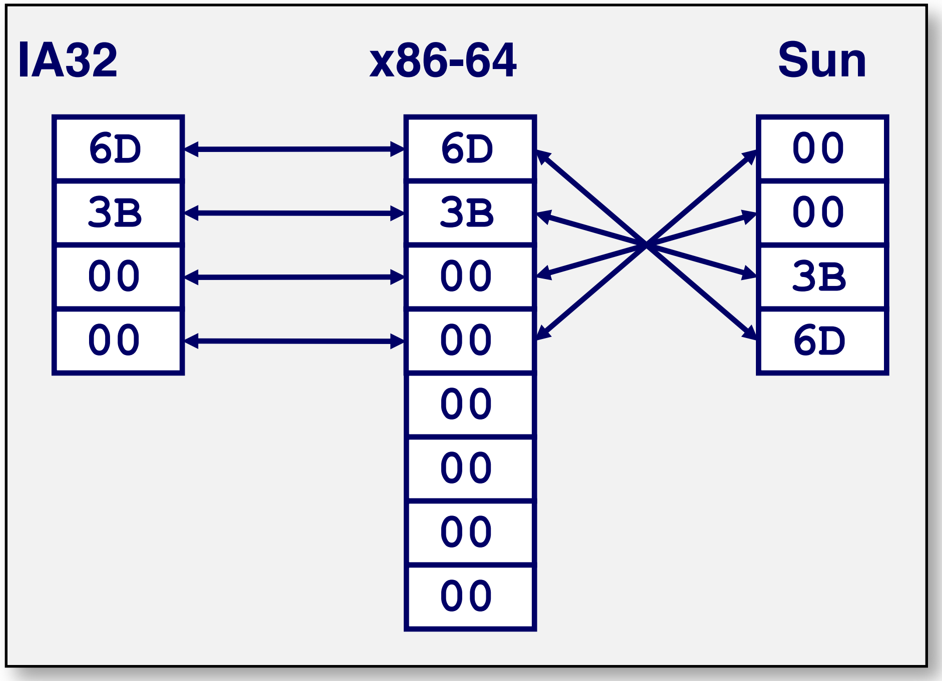
```
int A = 15213;
```



```
int B = -15213;
```



```
long int C = 15213;
```



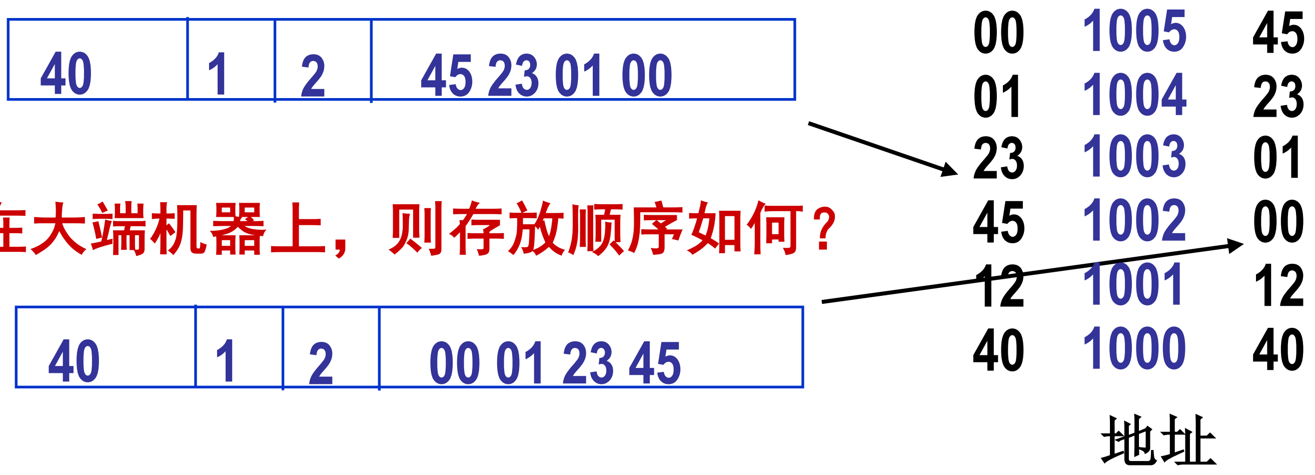
Two's complement representation

BIG Endian versus Little Endian

Ex3: Memory layout of a instruction located in 1000

假定小端机器中指令：mov AX, 0x12345(BX)

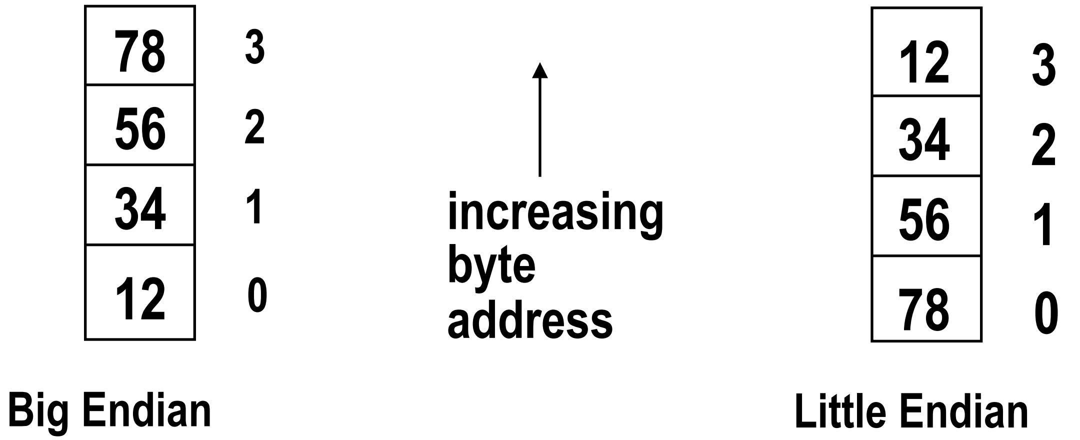
其中操作码mov为40H，寄存器AX和BX的编号分别为0001B和0010B，立即数占32位，则存放顺序为：



若在大端机器上，则存放顺序如何？

只需要考虑指令中立即数的顺序！

Byte Swap Problem（字节交换问题）



上述存放在0号单元的数据（字）是什么？12345678H?　78563412H?

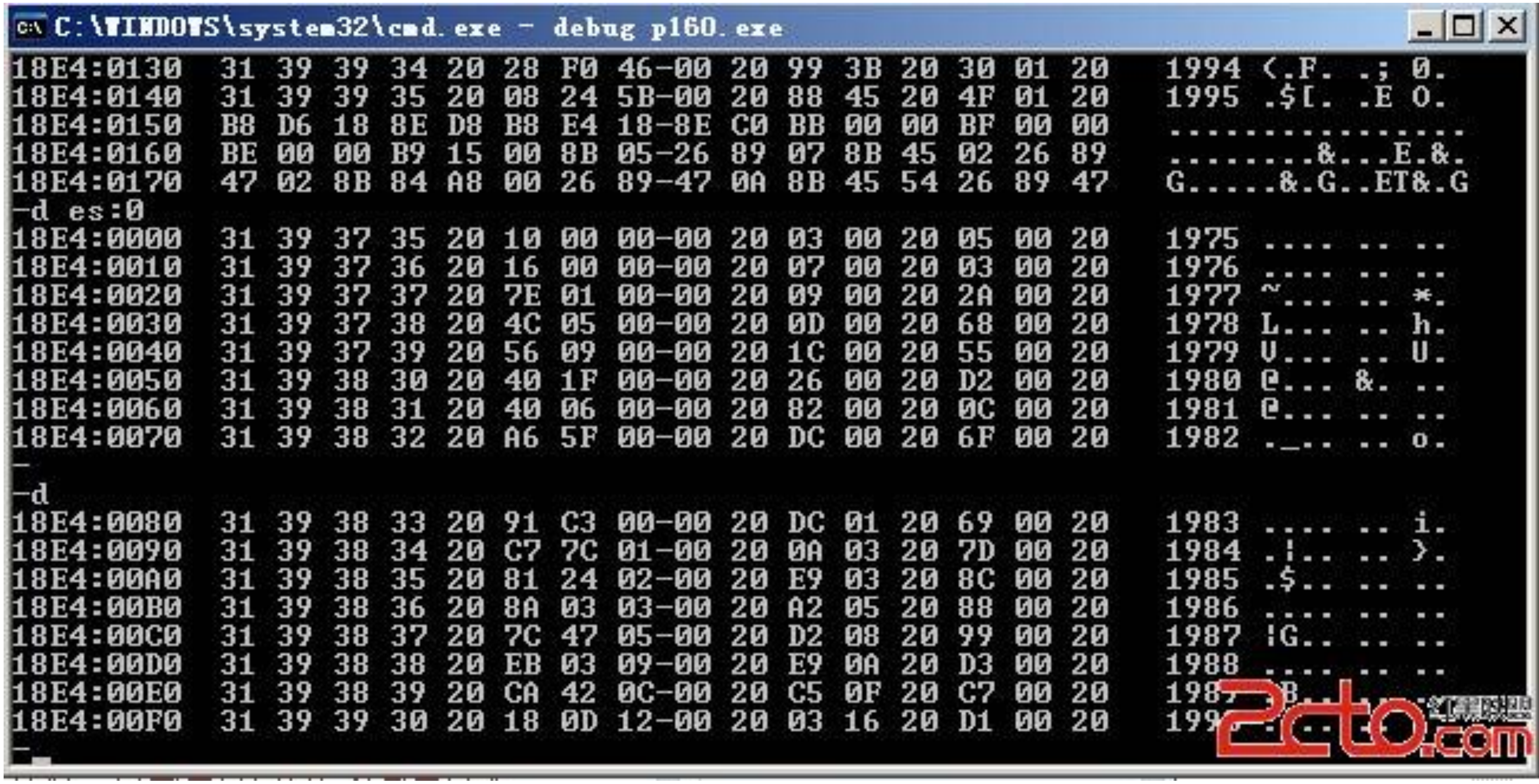
存放方式不同的机器间程序移植或数据通信时，会发生什么问题？

- 每个系统内部是一致的，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



```
C:\WINDOWS\system32\cmd.exe - debug p160.exe
18E4:0130  31 39 39 34 20 28 F0 46-00 20 99 3B 20 30 01 20   1994 (.F. .; 0.
18E4:0140  31 39 39 35 20 08 24 5B-00 20 88 45 20 4F 01 20   1995 .$[. .E O.
18E4:0150  B8 D6 18 8E D8 B8 E4 18-8E C0 BB 00 00 BF 00 00   ................
18E4:0160  BE 00 00 B9 15 00 8B 05-26 89 07 8B 45 02 26 89   ........&...E.&.
18E4:0170  47 02 8B 84 A8 00 26 89-47 0A 8B 45 54 26 89 47   G.....&.G..ET&.G
-d es:0
18E4:0000  31 39 37 35 20 10 00 00-00 20 03 00 20 05 00 20   1975 .... .. ..
18E4:0010  31 39 37 36 20 16 00 00-00 20 07 00 20 03 00 20   1976 .... .. ..
18E4:0020  31 39 37 37 20 7E 01 00-00 20 09 00 20 2A 00 20   1977 ~... .. *.
18E4:0030  31 39 37 38 20 4C 05 00-00 20 0D 00 20 68 00 20   1978 L... .. h.
18E4:0040  31 39 37 39 20 56 09 00-00 20 1C 00 20 55 00 20   1979 V... .. U.
18E4:0050  31 39 38 30 20 40 1F 00-00 20 26 00 20 D2 00 20   1980 @... &. ..
18E4:0060  31 39 38 31 20 40 06 00-00 20 82 00 20 0C 00 20   1981 @... .. ..
18E4:0070  31 39 38 32 20 A6 5F 00-00 20 DC 00 20 6F 00 20   1982 ._.. .. o.
-
-d
18E4:0080  31 39 38 33 20 91 C3 00-00 20 DC 01 20 69 00 20   1983 .... .. i.
18E4:0090  31 39 38 34 20 C7 7C 01-00 20 0A 03 20 7D 00 20   1984 .|.. .. }.
18E4:00A0  31 39 38 35 20 81 24 02-00 20 E9 03 20 8C 00 20   1985 .$.. .. ..
18E4:00B0  31 39 38 36 20 8A 03 03-00 20 A2 05 20 88 00 20   1986 .... .. ..
18E4:00C0  31 39 38 37 20 7C 47 05-00 20 D2 08 20 99 00 20   1987 |G.. .. ..
18E4:00D0  31 39 38 38 20 EB 03 09-00 20 E9 0A 20 D3 00 20   1988 .... .. ..
18E4:00E0  31 39 38 39 20 CA 42 0C-00 20 C5 0F 20 C7 00 20   198  .B..
18E4:00F0  31 39 39 30 20 18 0D 12-00 20 03 16 20 D1 00 20   199
-
```

Alignment(对齐)

Alignment: 要求数据的地址是相应的边界地址

- 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，也有位处理指令
- 各种不同长度的数据存放时，有两种处理方式:
  - 按边界对齐 （假定存储字的宽度为32位，按字节编址）
    - 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!

    （学了存储器组织后会更明白！）

每4个字节可同时读写

Alignment(对齐)

如：int i, short k, double x, char c, short j,……

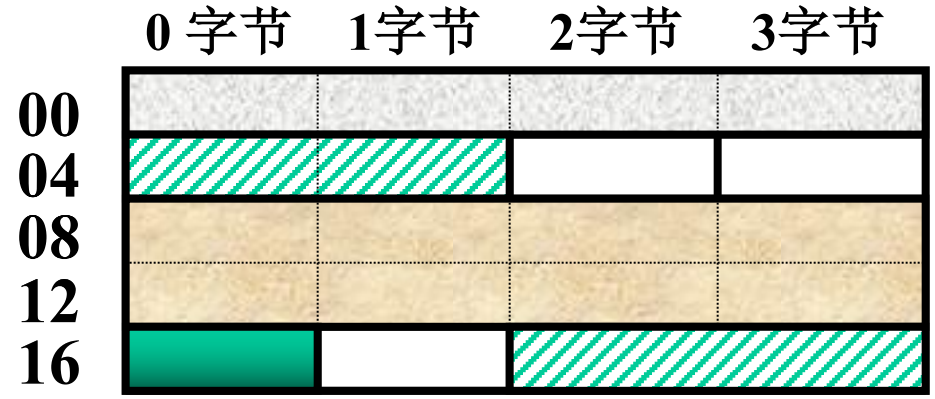
存储器按字节编址

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

按边界对齐

x：2个周期

j：1个周期



则：&i=0; &k=4; &x=8; &c=16; &j=18;……

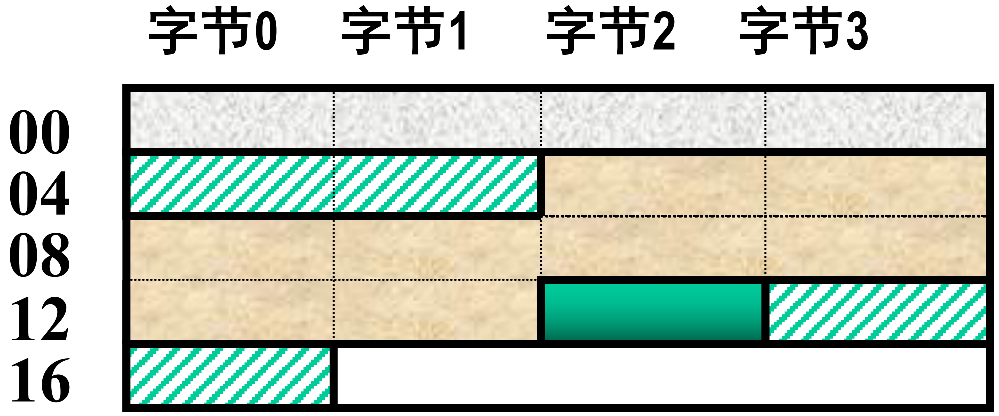
虽节省了空间，但增加了访存次数！

需要权衡，目前来看，浪费一点存储空间没有关系！

边界不对齐

x：3个周期

j：2个周期



则：&i=0; &k=4; &x=6; &c=14; &j=15;……

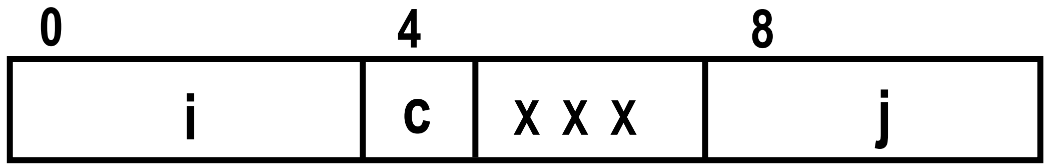
Alignment(对齐) 举例

例如，考虑下列两个结构声明：

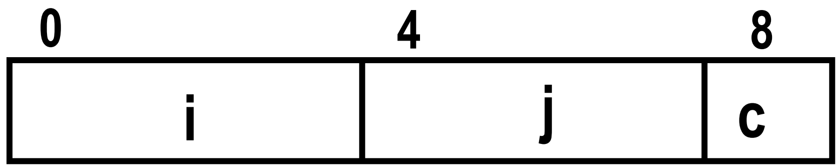
```
struct  S1 {
        int     i;
        char    c;
        int     j;
};
```

```
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**

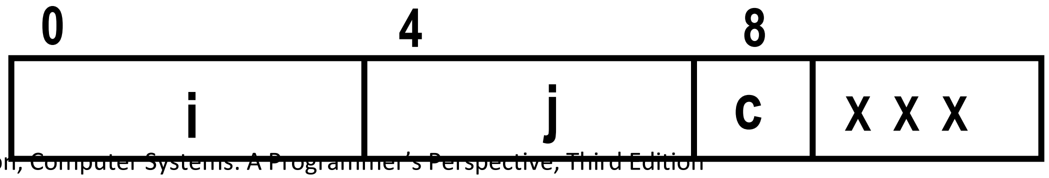
S1:


需要12个字节

S2:


只需要9个字节

对于 "struct S2 d[4]"，只分配9个字节能否满足对齐要求？ **不能！**

S2:


也需要12个字节

Today: Bits, Bytes, and Integers

- Representing information as bits
- **Bit-level manipulations**
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

逻辑数据的编码表示

- 表示
  - 用一位表示 。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，也是一串0/1序列，机器靠指令来识别。
- 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

Boolean Algebra

- **Developed by George Boole in 19th Century**
  - Algebraic representation of logic
    - Encode "True" as 1 and "False" as 0

And

- A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

- A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

Not

- ~A = 1 when A=0

~	
0	1
1	0

Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

General Boolean Algebras

- **Operate on Bit Vectors**
  - Operations applied bitwise

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  01000001     01111101     00111100     10101010
```

- **All of the Properties of Boolean Algebra Apply**

Example: Representing & Manipulating Sets

- **Representation**
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if j ∈ A
    - 01101001 { 0, 3, 5, 6 }
    - *76543210*
    - 01010101 { 0, 2, 4, 6 }
    - *76543210*
- **Operations**

&	Intersection	01000001	{ 0, 6 }
|	Union	01111101	{ 0, 2, 3, 4, 5, 6 }
^	Symmetric difference	00111100	{ 2, 3, 4, 5 }
~	Complement	10101010	{ 1, 3, 5, 7 }

Bit-Level Operations in C

- **Operations &, |, ~, ^ Available in C**
  - Apply to any "integral" data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (Char data type)**
  - `~0x41 →`
  - `~0x00 →`
  - `0x69 & 0x55 →`
  - `0x69 | 0x55 →`

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

Bit-Level Operations in C

- **Operations &, |, ~, ^ Available in C**
  - Apply to any "integral" data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (Char data type)**
  - `~0x41 → 0xBE`
    - ~0100 0001_2 → 1011 1110_2
  - `~0x00 → 0xFF`
    - ~0000 0000_2 → 1111 1111_2
  - `0x69 & 0x55 → 0x41`
    - 0110 1001_2 & 0101 0101_2 → 0100 0001_2
  - `0x69 | 0x55 → 0x7D`
    - 0110 1001_2 | 0101 0101_2 → 0111 1101_2

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

Contrast: Logic Operations in C

- **Contrast to Bit-Level Operators**
  - **Logic Operations &, ||, !**
    - View 0 as “Fals
    - Anything nonze
    - Alway
    - Early

Watch out for && vs. & (and || vs. |)… one of the more common oopsies in C programming

- **Example**
  - `!0x41`
  - `!0x00 →`
  - `!!0x41→   0x01`
  - `0x69 && 0x55 →   0x01`
  - `0x69 || 0x55 →   0x01`
  - `p && *p`   (avoids null pointer access)

Shift Operations

- **Left Shift:** **`x << y`**
  - Shift bit-vector **x** left **y** positions
    - – Throw away extra bits on left
    - Fill with 0's on right
- **Right Shift:** **`x >> y`**
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - **Logical shift**
    - Fill with 0's on left
  - **Arithmetic shift**
    - Replicate most significant bit on left
- **Undefined Behavior**
  - Shift amount < 0 or ≥ word size

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

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

西文字符的编码表示

- 特点
  - 是一种拼音文字，用有限几个字母可拼写出所有单词
  - 只对有限个字母和数学符号、标点符号等辅助字符编码
  - 所有字符总数不超过256个，使用7或8个二进位可表示
- 表示（常用编码为7位ASCII码）
  - 十进制数字：0/1/2…/9（30H）
  - 英文字母：A/B/…/Z/a/b/…/z（41H、61H）

  必须熟悉对应的ASCII码！
  - 专用符号：+/-/%/*/&/……
  - 控制字符（不可打印或显示）（回车：0DH；换行：0AH）
- 操作
  - 字符串操作，如:传送/比较　等

ASCII码

ASCII码	控制字符	ASCII码	字符	ASCII码	字符	ASCII码	字符
0	NUL	32	(space)	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	″	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	%	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	′	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	-	127	DEL

汉字及国际字符的编码表示

- **特点**
  - 汉字是表意文字，一个字就是一个方块图形。
  - 汉字数量巨大，总数超过6万字，给汉字在计算机内部的表示、汉字的传输与交换、汉字的输入和输出等带来了一系列问题。
- **编码形式**
  - 有以下几种汉字代码：
    - 输入码：对汉字用相应按键进行编码表示，用于输入
    - 内码：用于在系统中进行存储、查找、传送等处理
    - 字模点阵或轮廓描述: 描述汉字字模点阵或轮廓，用于显示/打印

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - **Representation: unsigned and signed**
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings
- Summary

Encoding Integers

Unsigned

$$D4\,W^*Z+ \;=\; \sum_{K=2}^{Y-3} Z_K \cdot 4^K$$

Two's Complement

$$D4\,V^*Z+ \;=\; -Z_{Y-3} \cdot 4^{Y-3} + \sum_{K=2}^{Y-4} Z_K \cdot 4^K$$

Sign Bit

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

- **C `short` 2 bytes long**

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

- **Sign Bit**
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

Two-complement: Simple Example

	-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

Two-complement Encoding Example (Cont.)

```
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**

Numeric Ranges

- **Unsigned Values**
  - *UMin* = 0

    000…0
  - *UMax* = $2^w - 1$

    111…1

- **Two's Complement Values**
  - *TMin* = -2^{w-1}

    100…0
  - *TMax* = $2^{w-1} - 1$

    011…1
  - Minus 1

    111…1

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

"

Values for Different Word Sizes

	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

- **Observations**
  - $|TMin| = TMax + 1$
    - Asymmetric range
  - $UMax = 2 * TMax + 1$
- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values platform specific

Unsigned & Signed Numeric Values

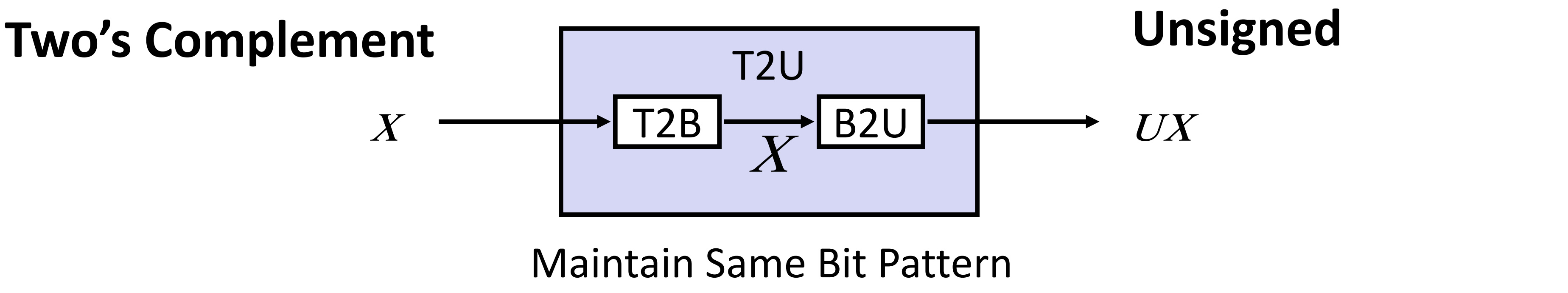
X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- **Equivalence**
  - Same encodings for nonnegative values
- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- **$\Rightarrow$ Can Invert Mappings**
  - U2B(x) = B2U^{-1}(x)
    - Bit pattern for unsigned integer
  - T2B(x) = B2T^{-1}(x)
    - Bit pattern for two's comp integer

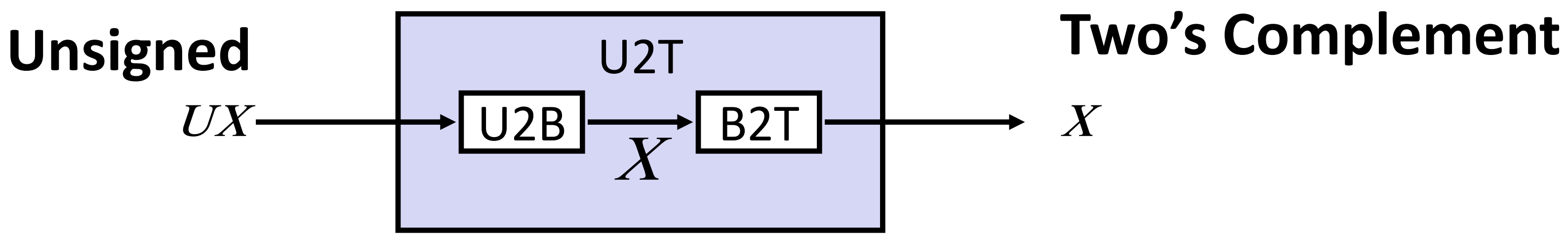
Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - **Conversion, casting**
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Mapping Between Signed & Unsigned



Maintain Same Bit Pattern



Maintain Same Bit Pattern

- **Mappings between unsigned and two's complement numbers:**
  Keep bit representations and reinterpret

Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
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	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

$\longrightarrow \boxed{\text{T2U}} \longrightarrow$

$\longleftarrow \boxed{\text{U2T}} \longleftarrow$

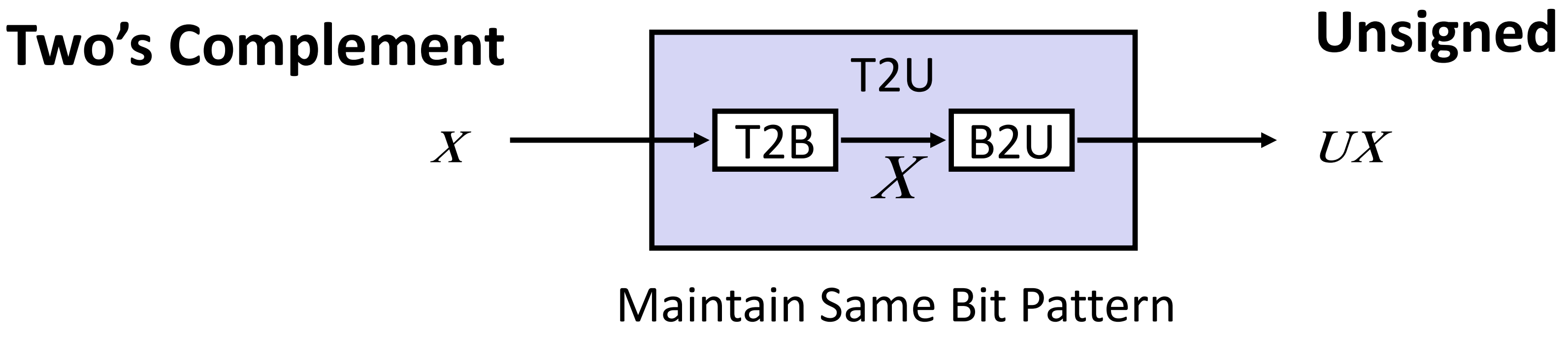
Mapping Signed ↔ Unsigned

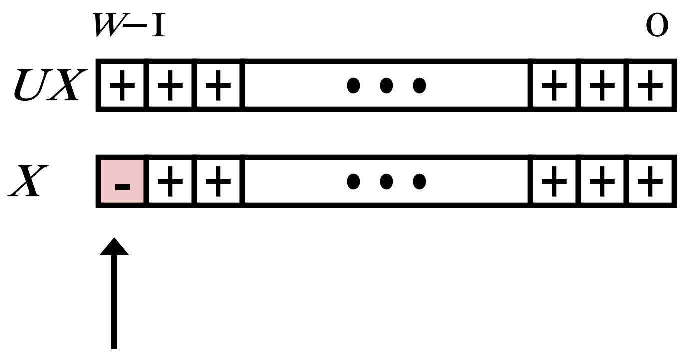
Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

Signed $\xleftrightarrow{=}$ Unsigned (0000–0111)

Signed $\xleftrightarrow{+/-\ 16}$ Unsigned (1000–1111)

Relation between Signed & Unsigned

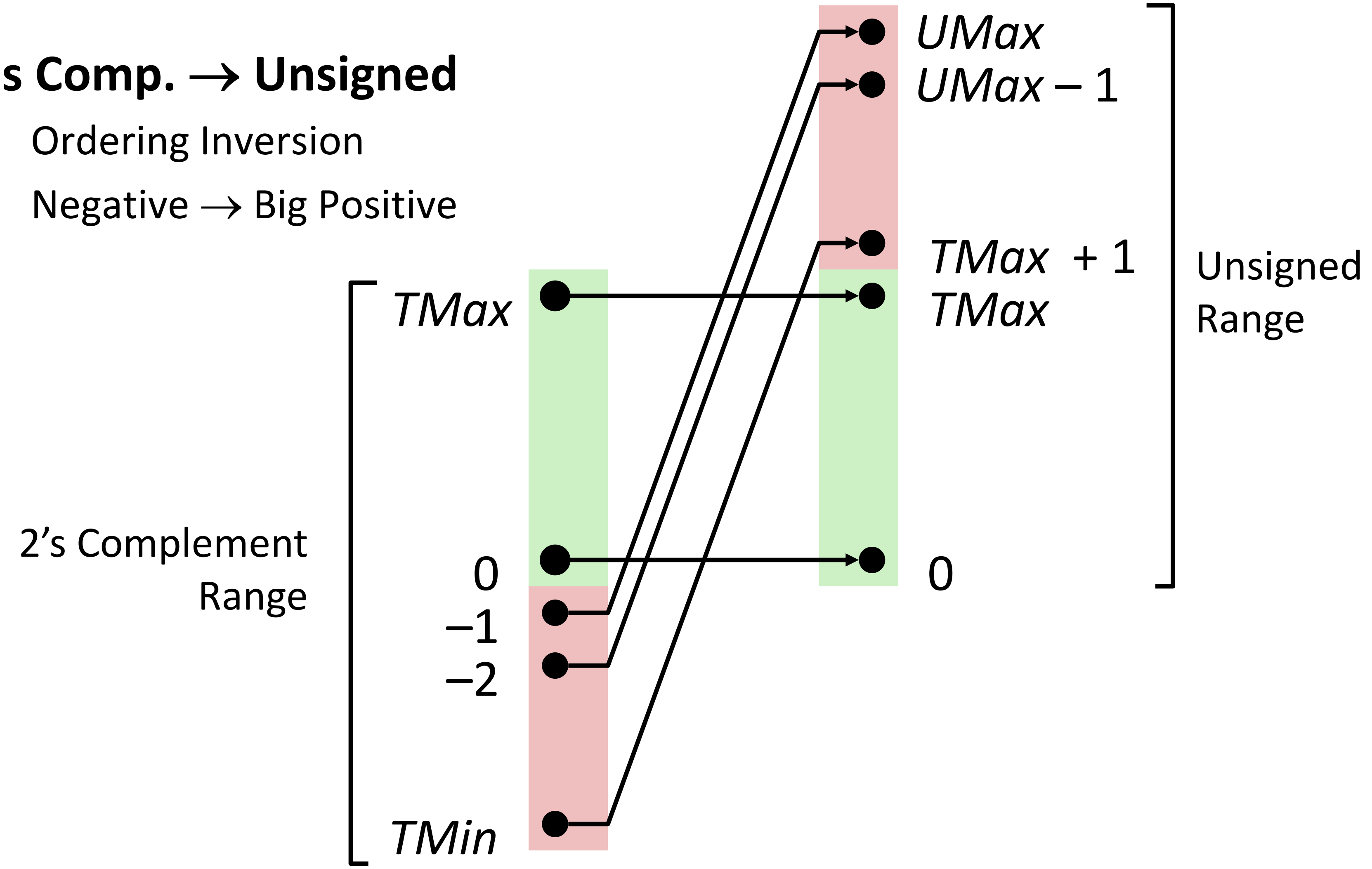




Large negative weight
becomes
Large positive weight

Conversion Visualized

- **2's Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive



补码↔无符号数

- 对于满足$TMin_w \leq x \leq TMax_w$的$x$(补码)有

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x > 0 \end{cases}$$

- 对于满足$0 \leq u \leq UMax_w$的u(无符号数)有

$$U2T_w(u) = \begin{cases} u - 2^w, & u > TMax_w \\ x, & u \leq TMax_w \end{cases}$$

Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be **signed integers**
  - Unsigned if have "**U**" as suffix

```
0U, 4294967259U
```

- **Casting**
  - **Explicit casting** between signed & unsigned same as U2T and T2U

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

  - **Implicit casting** also occurs via assignments and procedure calls

```
tx = ux;                    int fun(unsigned u);
uy = ty;                    uy = fun(tx);
```

Casting Surprises

- **Expression Evaluation**
  - If there is a mix of unsigned and signed in single expression, ***signed values implicitly cast to unsigned***
  - Including comparison operations **<, >, ==, <=, >=**
  - Examples for *W* = 32: **TMIN = -2,147,483,648 , TMAX = 2,147,483,647**

关系表达式	运算类型	结果	说明
0 == 0U			
-1 < 0			
-1 < 0U			
2147483647 > -2147483647-1			
2147483647U > -2147483647-1			
2147483647 > (int) 2147483648U			
-1 > -2			
(unsigned) -1 > -2			

C语言程序中的整数

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

带*的结果与常规预想的相反！

C语言程序中的整数

例如，考虑以下C代码：

```
int x = –1;
unsigned u = 2147483648;

printf (  “x = %u = %d\n” , x, x);
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。**

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类型，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int类型，因此“-2147483648 < 2147483647”按带符号整数比较，10……0B比01……1小，结果为true。

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。

Unsigned vs. Signed: Easy to Make Mistakes

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

- Can be very subtle

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

Summary
Casting Signed ↔ Unsigned: Basic Rules

- **Bit pattern is maintained**
- **But reinterpreted**
- **Can have unexpected effects: adding or subtracting 2^w**

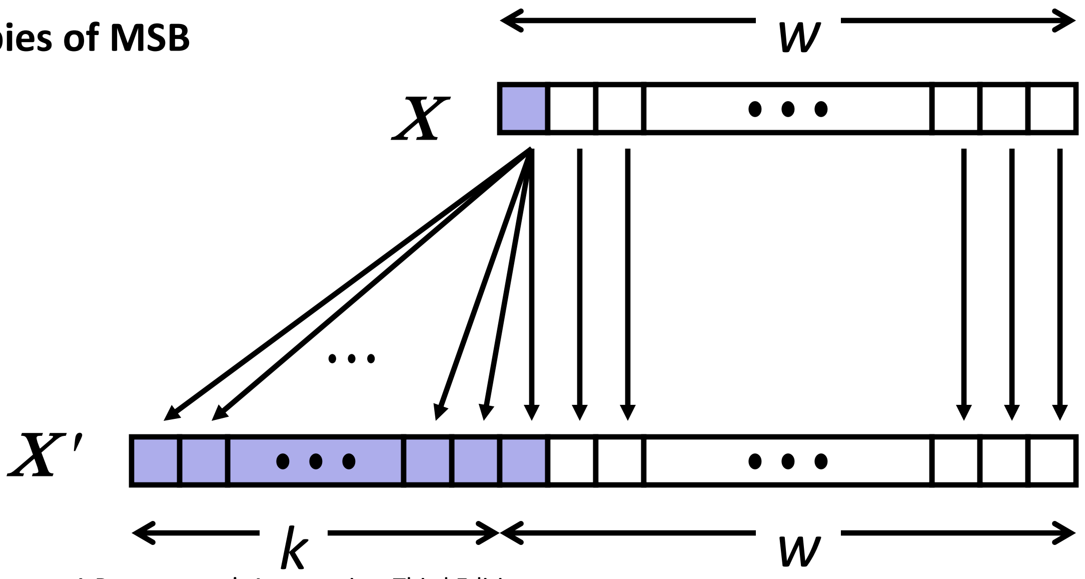
- **Expression containing signed and unsigned int**
  - `int` is cast to `unsigned`!!

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - **Expanding, truncating**
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

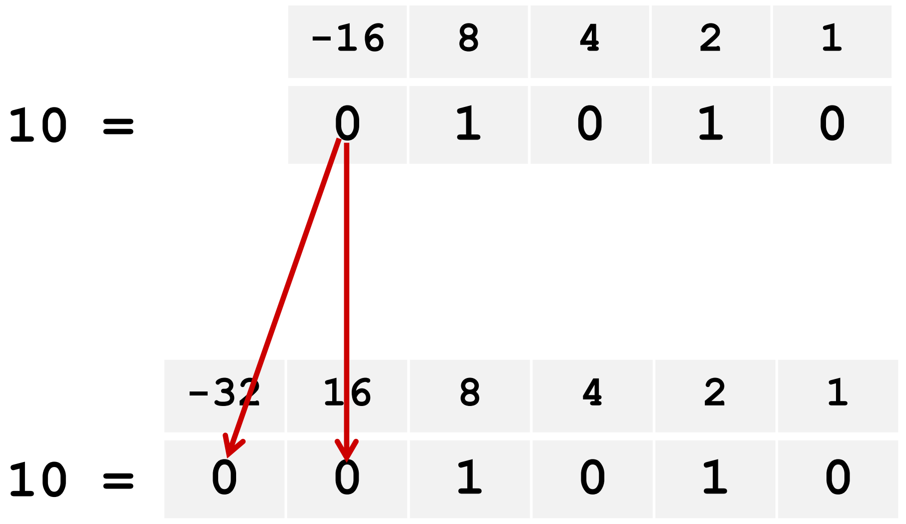
Sign Extension

- **Task:**
  - Given *w*-bit signed integer *x*
  - Convert it to *w*+*k*-bit integer with same value
- **Rule:**
  - Make *k* copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Sign Extension: Simple Example

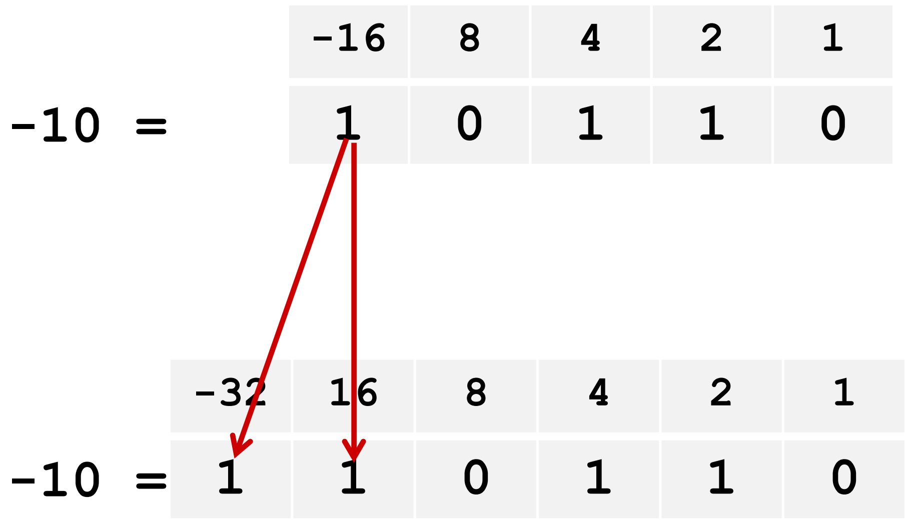
Positive number



	-16	8	4	2	1
10 =	0	1	0	1	0

	-32	16	8	4	2	1
10 =	0	0	1	0	1	0

Negative number



	-16	8	4	2	1
-10 =	1	0	1	1	0

	-32	16	8	4	2	1
-10 =	1	1	0	1	1	0

Larger Sign Extension Example

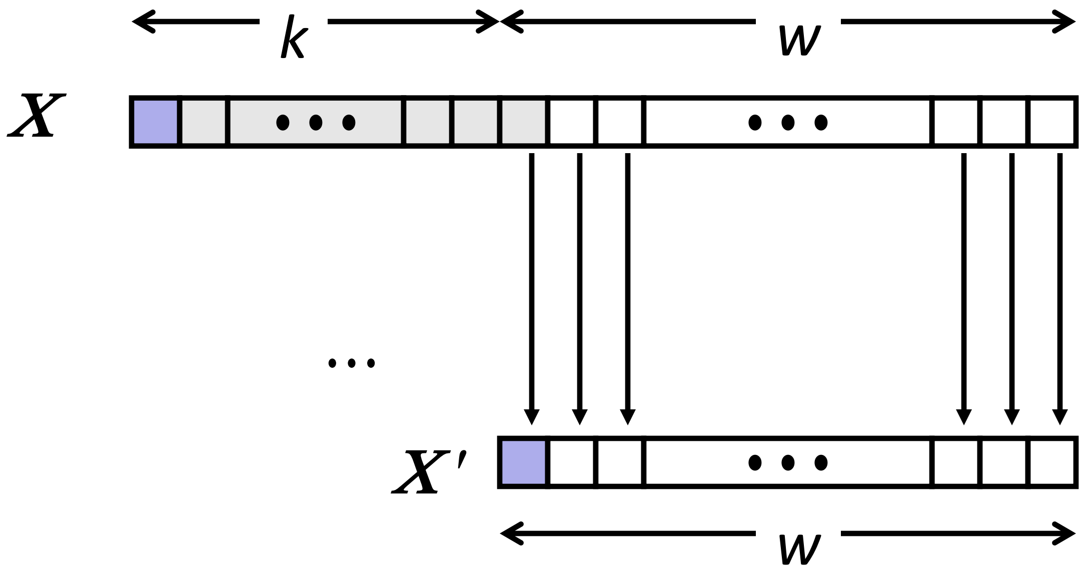
```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	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

- Converting from smaller to larger integer data type
- C automatically performs sign extension

Truncation

- **Task:**
  - Given k+*w*-bit signed or unsigned integer *X*
  - Convert it to *w*-bit integer X' with same value for "small enough" X
- **Rule:**
  - Drop top *k* bits:
  - $X' = x_{w-1}, x_{w-2}, \ldots, x_0$



Truncation: Simple Example

No sign change

	-16	8	4	2	1
2 =	0	0	0	1	0

	-8	4	2	1
2 =	0	0	1	0

`2 mod 16 = 2`

	-16	8	4	2	1
-6 =	1	1	0	1	0

	-8	4	2	1
-6 =	1	0	1	0

`-6 mod 16 = 26U mod 16 = 10U = -6`

Sign change

	-16	8	4	2	1
10 =	0	1	0	1	0

	-8	4	2	1
-6 =	1	0	1	0

`10 mod 16 = 10U mod 16 = 10U = -6`

	-16	8	4	2	1
-10 =	1	0	1	1	0

	-8	4	2	1
6 =	0	1	1	0

`-10 mod 16 = 22U mod 16 = 6U = 6`

举例

例1（扩展操作）：在大端机上输出 si, usi, i, ui的十进制和十六进制值是什么？

```
short  si = -32768;
unsigned short  usi = si;
int  i = si;
unsingned  ui = usi ;
```

```
si = -32768    80 00
usi = 32768   80 00
i = -32768     FF FF 80 00
ui = 32768     00 00 80 00
```

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！

Summary: Expanding, Truncating: Basic Rules

- **Expanding (e.g., short int to int)**
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- **Truncating (e.g., unsigned to unsigned short)**
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small (in magnitude) numbers yields expected behavior

小结

- 非数值数据的表示
  - 逻辑数据用来表示真/假或N位位串，按位运算
  - 西文字符：用ASCII码表示
  - 汉字编码
- 数据的宽度
  - 位、字节、字（不一定等于字长）
  - k /K / M / G / T / P / E / Z / Y 有不同的含义
- 数据的存储排列
  - 数据的地址：连续若干单元中最小的地址，即：从小地址开始存放数据
    - 问题：若一个short型数据si存放在单元0x08000100和0x08000101中，那么si的地址是什么?
  - 大端方式：用MSB存放的地址表示数据的地址
  - 小端方式：用LSB存放的地址表示数据的地址
  - 按边界对齐可减少访存次数

小结

- 在机器内部编码后的数称为机器数，其值称为真值
- 定义数值数据有三个要素：进制、定点/浮点、编码
- 整数的表示
  - 无符号数：正整数，用来表示地址等；带符号整数：用补码表示
- C语言中的整数
  - 无符号数：unsigned int (short / long)；带符号数： int (short / long)
- 十进制数的表示：用ASCII码或BCD码表示
- 整形运算
  - 算术运算：无符号整数、有符号整数
  - 按位运算： "|" 、 "&" 、 "~" 、 "^"
  - 逻辑运算： "| |" 、 "& &" 、 "！ "
  - 移位运算： "<<" 、 ">>" （逻辑移位、算术移位）
  - 位扩展和位截断：无符号和有符号