SSD
6

icarnegie
curriculum powered by Carnegie Mellon

Lecture 4

Representation of Data

武汉大学 国际软件学院

Lecture 3 Representation of Data

The Contents in iCarnegie cover:

2.1 Bits and Bit Manipulation

2.2 Integers

2.3 Floating-Point Numbers

2.4 Structured Data

Lecture 3 Representation of Data

We will talk about:

- Numbering System
- Alphanumeric Expression
- Number Expression
- Structured Data in Memory
- Logic Operation

Lecture 3 Numbering System

- 十进制(Decimal)数字系统
- 二进制(Binary)数字系统
- 十六进制(Hexadecimal)数字系统

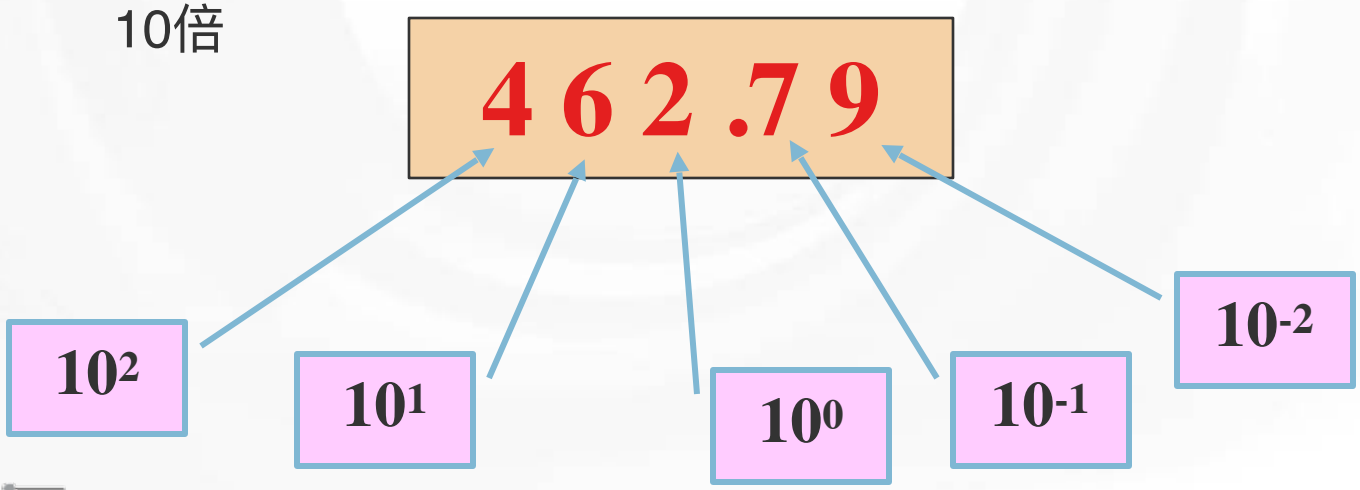
Lecture 3 Numbering System

- 十进制(Decimal)数字系统
  - 以 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 等十个不同符号(symbol)组成
  - 人类使用的数字系统: 人有十根指头
  - 超过九的数目就必须用多位数元(digits)表示

如 : 9+1=10, 99+1=100

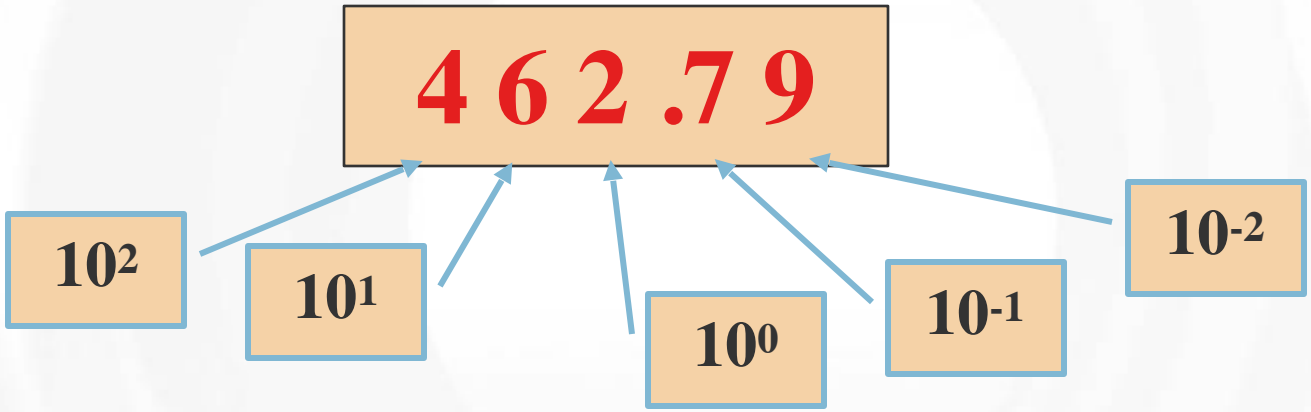
Lecture 3 Numbering System

- 十进制(Decimal)数字系统
  - 以10为基底的数字系统(Base 10 number system)
    - 不同位置数元代表不同的数量(quantity)
    - 小数点左边第一位为 1, 第二位为10, …
    - 每个位置所代表的数量为其右边位置所代表的数量之10倍



Lecture 3 Numbering System

- 十进制(Decimal)数字系统



$$4\times10^{2}+6\times10^{1}+2\times10^{0}+7\times10^{-1}+9\times10^{-2}$$

$$=400+60+2+0.7+0.09$$

$$=46279_{10}$$

Lecture 3 Numbering System

二进制(Binary)数字系统

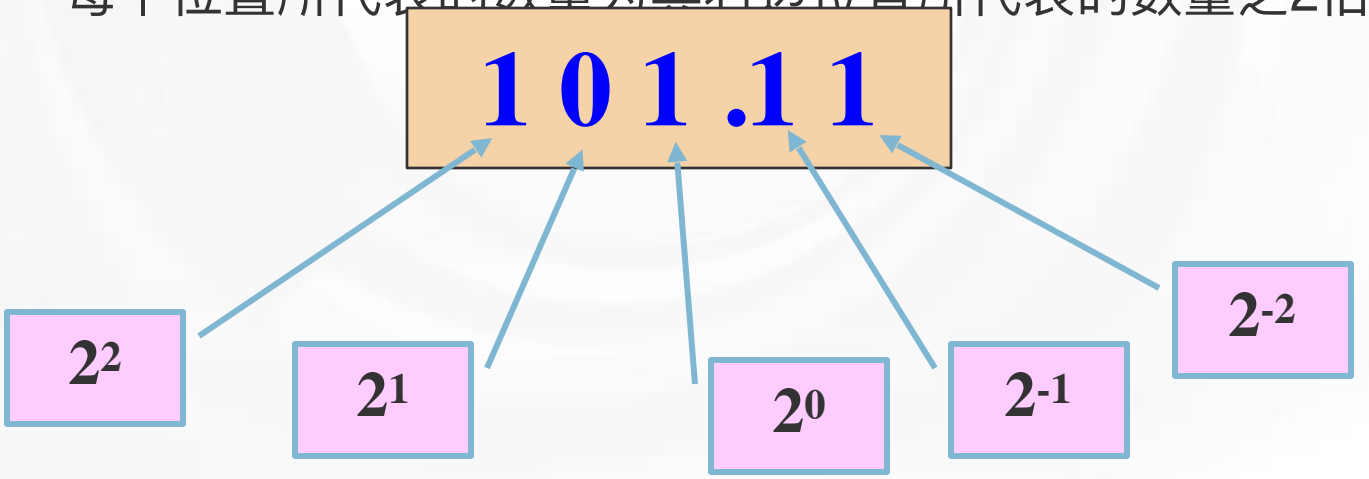
- 以 0, 1 两个符号组成
- 数字计算器使用的数字系统:电子开关(晶体管)有两种状态(ON:1, OFF:0)
- 超过一的数目就必须用多位数元(digits)表示

  如 : $1+1=10_2$, $11+1=100_2$
- binary digit (二进制数元) 简称 bit (位)

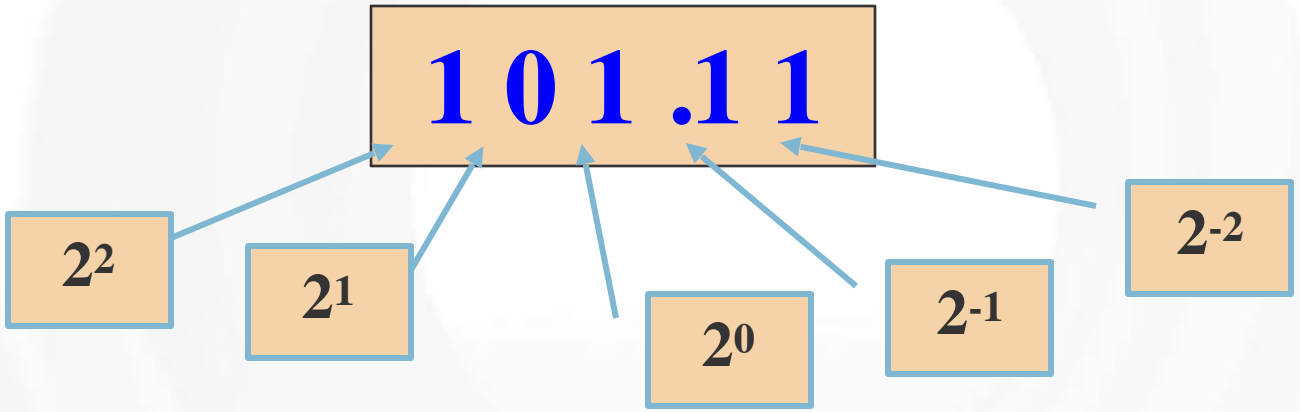
Lecture 3 Numbering System

二进制(Binary)数字系统

- 以2为基底的数字系统(Base 2 number system)
  - 不同位置数元代表不同的数量(quantity)
  - 小数点左边第一位为 1 (2^0)，第二位为2 (2^1)，…
  - 每个位置所代表的数量为其右边位置所代表的数量之2倍



二进制(Binary)数字系统



$$1\times 2^2 + 1\times 2^0 + 1\times 2^{-1} + 1\times 2^{-2}$$

$$= 4 + 1 + 0.5 + 0.25$$

$$= 5.75_{10}$$

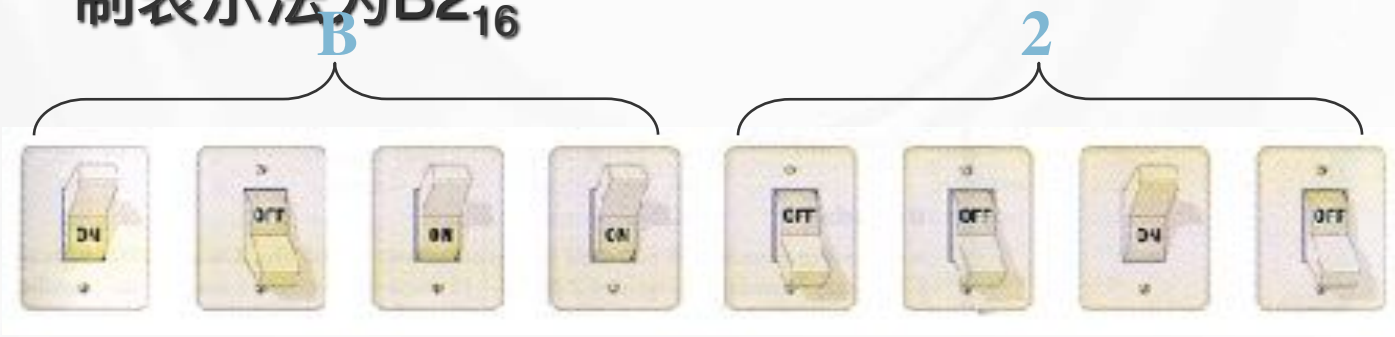
十六进制(Hexadecimal)数字系统

- 以 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F 等十六个符号组成
- 符号A~F 代表10~15
- 超过十五的数目就必须用多位数元(digits)表示

  如：$F+1=10_{16}$, $FF+1=100_{16}$

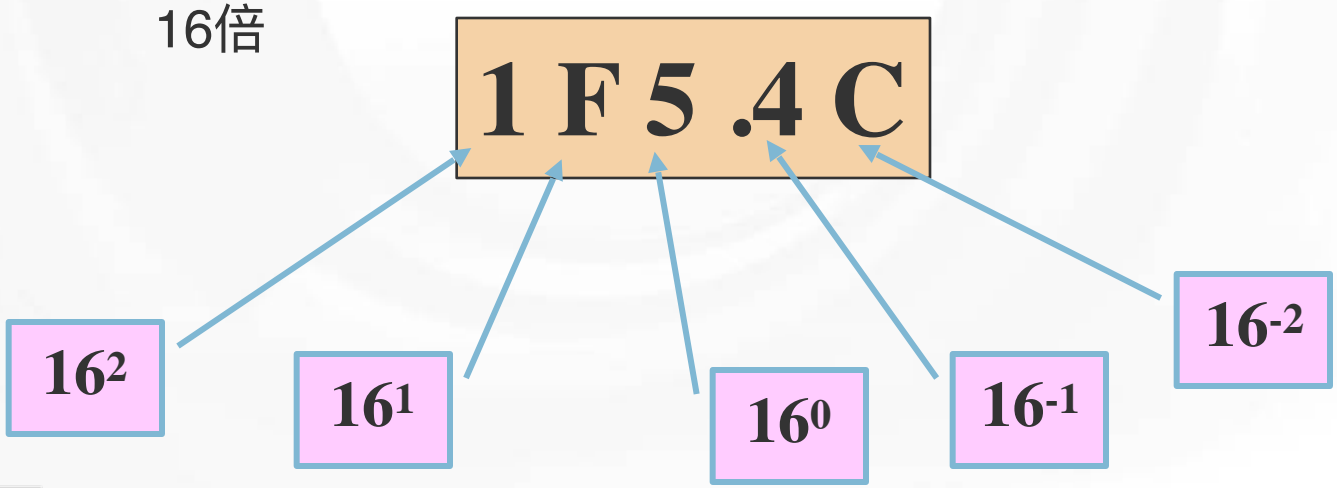
为何要谈十六进制数字系统

- 8个字节成一个字 (byte), 可表现256(2^8)种不同0/1组合
- 可用十六进制法精简表示一个字的内容
- 下图之内容以二进制表示为10110010_2,以十六进制表示法为$B2_{16}$



十六进制(Hexadecimal)数字系统

- 以16为基底的数字系统(Base 16 number system)
  - 不同位置数元代表不同的数量(quantity)
  - 小数点左边第一位为 1，第二位为16, …
  - 每个位置所代表的数量为其右边位置所代表的数量之16倍



Lecture 3 Words & Virtual Address Space

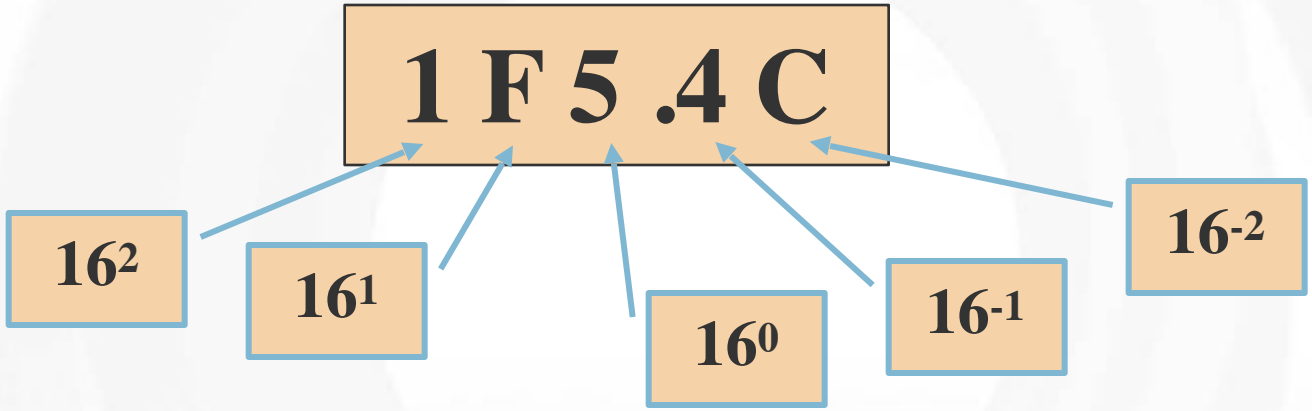




Every computer has a *word size*, indicating the nominal size of integer and pointer data. Since a virtual address is encoded by such a word, the most important system parameter determined by the word size is the maximum size of the virtual address space. That is, for a machine with an n-bit word size, the virtual addresses can range from 0 to $2^n - 1$, giving the program access to at most 2^n bytes.

Most computers today have a 32-bit word size. This limits the virtual address space to 4 gigabytes (written 4 GB), that is, just over 4×10^9 bytes. Although this is ample space for most applications, we have reached the point where many large-scale scientific and database applications require larger amounts of storage. Consequently, high-end machines with 64-bit word sizes are becoming increasingly commonplace as storage costs decrease.

十六进制(Hexadecimal)数字系统



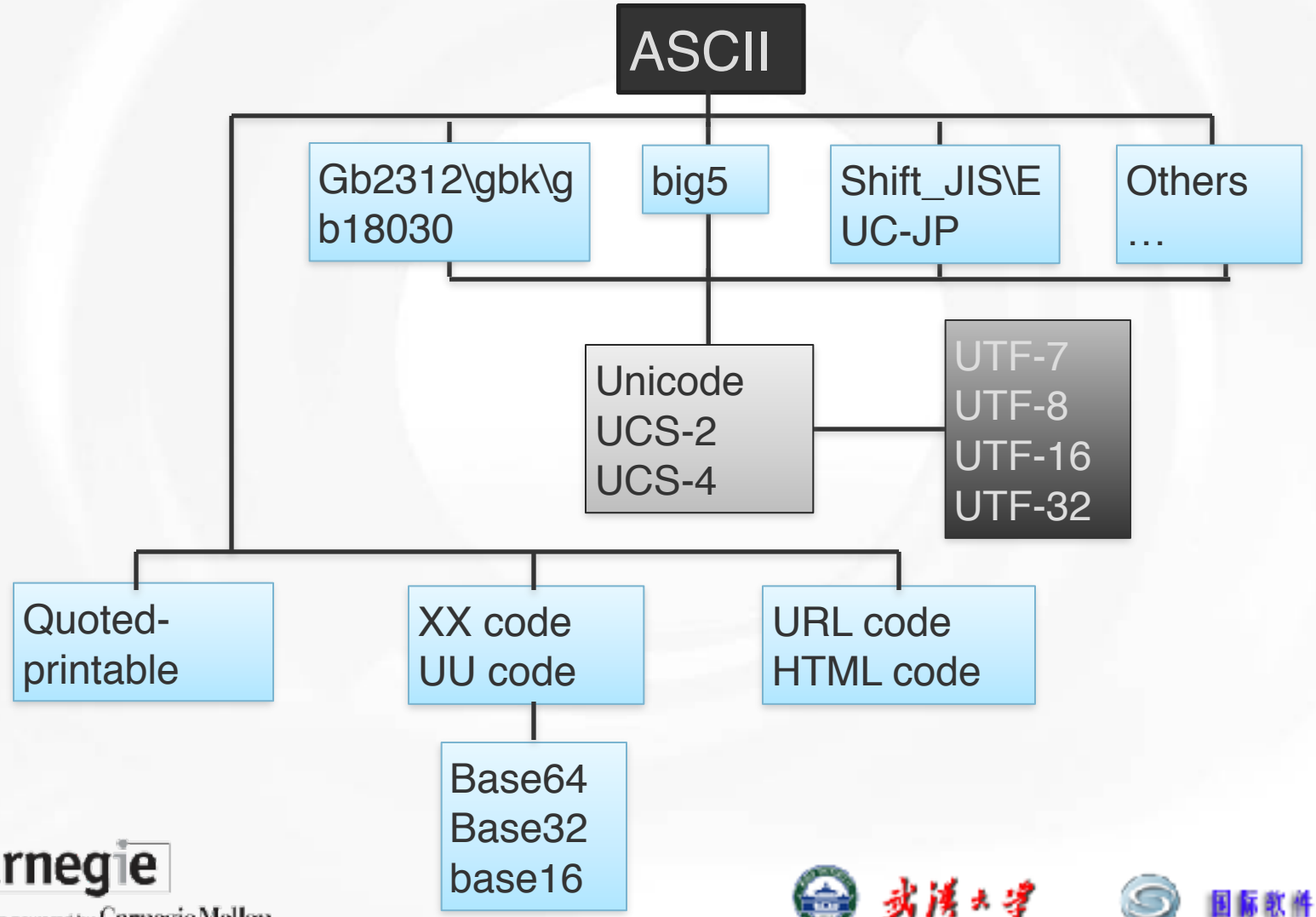
$$1\times16^{2}+15\times16^{1}+5\times16^{0}+4\times16^{-1}+12\times16^{-2}$$

$$=256+240+5+0.25+0.046875$$

$$=501.296875_{10}$$

Lecture 3 Alphanumeric Data Expre.

Alphanumeric Data Expression



Lecture 3 Alphanumeric Data Expre.

ASCII code

- American Standard Code for Information Interchange
- Standard ASCII code uses 7-digits to express 128(2^7) characters
  - 0~31, 127: control characters
  - 32~64: speccial characters(!, “, #, $, ..) and numberic characters (0, 1, …9)
  - 65~90: Capital letters
  - 97~122: Small letters

Lecture 3 Alphanumeric Data Expre.

ASCII code table (7 bits)

"A"=65:

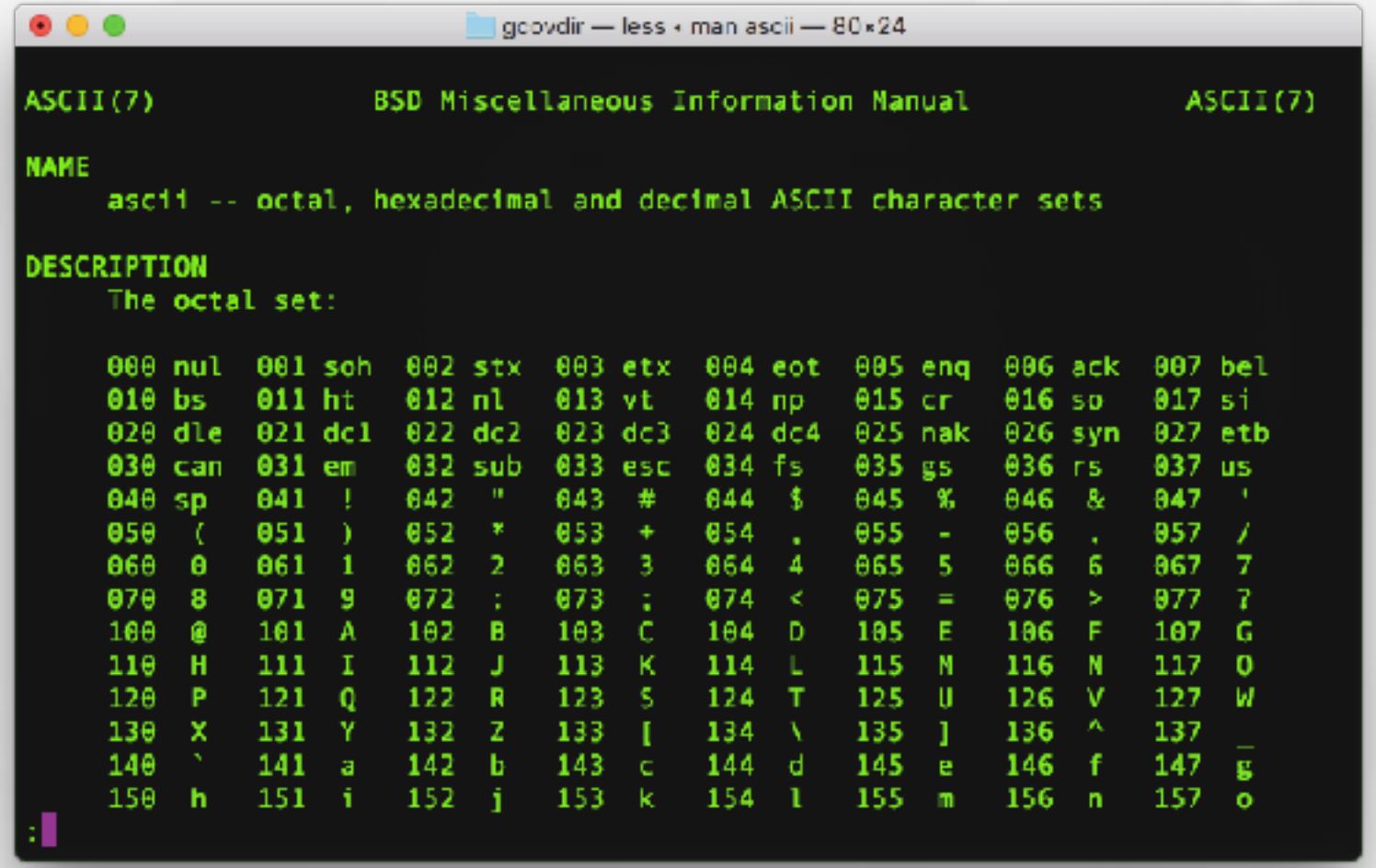
1000001_2

(41_{16})

654bit / 3210bit	000	001	010	011	100	101	110	111
0000	NUL	DEL	SPACE	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Lecture 3 Alphanumeric Data Expre.

In Linux\Unix\Mac: *"man ascii"*



Lecture 3 Alphanumeric Data Expre.

gb2312/gbk/gb18030/big5

我=> 1100111011010010

我=> 1100111001010010

Add minority words:

Hong Kong\Taiwan\...

gb2312

gbk

gb18030

big5

Alphanumeric Data Expre.

Unicode: UCS–2/UCS–4

Use 2–bytes: Max: 65536(2^{16}) UCS-2

Use 4–bytes: Max: 4294967296 (2^{32}) UCS-4

Unicode: for transportation

Transport 8 digits each time

For A，only 1 group (8 digits):
01000001

For “汉”，only 3 group (24 digits):
11100110 10110001 10001001

} Utf-8

So does utf-7，utf-16，utf-32

Quoted–printable: Base64/32/16

Use visible letters for ALL characters

Base64:

Capital letters set: W = {A,B,C…Z } totally: 26
Small letters set: w = {a,b,c…z } totally: 26
Numbers set: d = {0,1,2…9 } totally 10
Symbol set: s = {+,/ }
ABC(ascii)–> 01000001 01000010 01000011
–>010000 010100 001001 000011
–>QUJD(Base64)

Quoted–printable: Base64/32/16

Base64:

010000 010100 001001 000011→

00010000 00010100 00001001
00000011

–>QUJD(Base64)

Value	Char	Value	Char	Value	Char	Value	Char
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

Quoted–printable: Base64/32/16

Why base64?

One reason is that transferring images on web pages to base64 encoding makes images loaded prior to contents, decreasing http requests, and lowering server's overload.

```
background:url("data:image/gif;
base64,
R0lGODlhAQADAJEAAObm5t3d3ff39wAAACH5BAAAAAAALAAAAAABAAMAAAICDFQAOw=
repeat-x scroll 0 0 transparent;
-webkit-box-sizing:content-box;
-moz-box-sizing:content-box;
box-sizing:content-box;
*top:-1px;
_height:18px;
_line-height:18px
```

Quoted–printable: Base64/32/16

and so forth…

Base16:

Capital letters set: W = {A,B,C,D,E,F } totally: 6

Numbers set: d = {0,1,2…9 } totally 10

0xBFFF10E6

Lecture 3 Alphanumeric Data Expre.

URL，HTML code

URL code:

use“%+ascii”to encode the characters，for example：
http://www.wr.com?file=%2e%2e%2f%2e%2e%2fpasswd

HTML code:

use“&#+decimal number”or“&#x+hexdecimal number”to encode the characters，for example：
ab –> ac

Lecture 3 Alphanumeric Data Expre.

EXPAND

Encryption

- Encryption
  - Classical Encryption
    - Caesar encoding
    - Virginia encoding
    - Morse encoding
    - Hockshop encoding
  - Modern encryption
    - Symmetric encryption
    - Non-symmetric encryption

Symmetric encryption

Stream encryption algorithm:

```
  01101101 01101001 01100100 01101110 01101001 01100111 01101000 01110100
⊕ 01101011 11111010 01001000 11011000 01100101 11010101 10101111 00011100
  --------------------------------------------------------------------------
  00000110 10010011 00101100 10110110 00001100 10110010 11000111 01101000
```

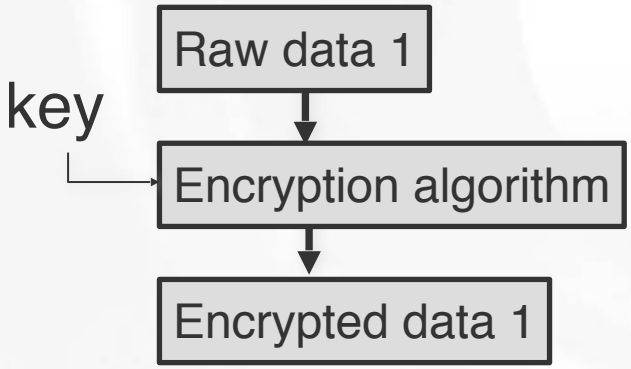
- Simple
- Unsafe
- Ease decrypted

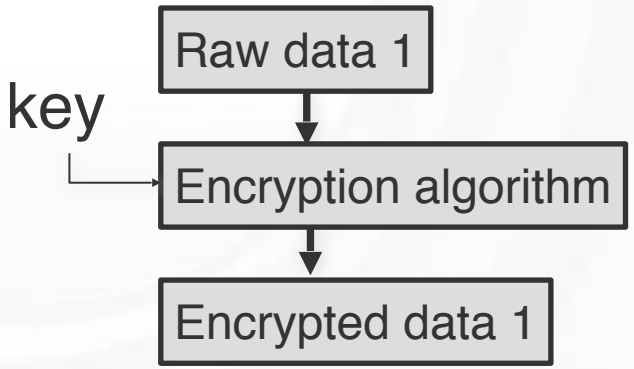
Symmetric encryption

EXPAND

Grouping encryption algorithm:

ECB mode





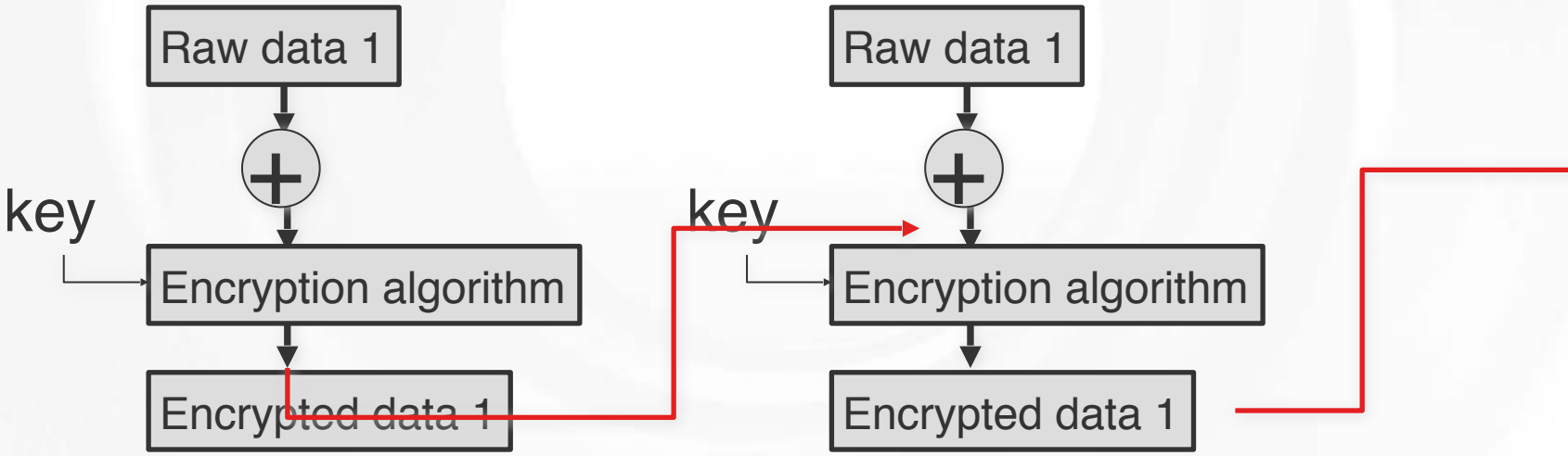
Each group is encrypted by keys independently

Symmetric encryption

EXPAND

Grouping encryption algorithm:

CBC mode



Each group is encrypted by keys evolsionaly!

Symmetric encryption

EXPAN

Comparison between CBC & ECB

Symmetric encryption

A story about Enigma

The Enigma machines were a series of electro-mechanical rotor cipher machines developed and used in the early- to mid-20th century to protect commercial, diplomatic and military communication. Enigma was invented by the German engineer Arthur Scherbius at the end of World War I.

Several different Enigma models were produced, but the German military models, having a plugboard, were the most complex. Japanese and Italian models were also in use.

Symmetric encryption

A story about Enigma

Around December 1932, *Marian Rejewski* of the Polish Cipher Bureau used the theory of permutations and flaws in the German military message procedures to break the message keys of the plugboard Enigma machine. Rejewski achieved this result without knowledge of the wiring of the machine, so the result did not allow the Poles to decrypt actual messages.

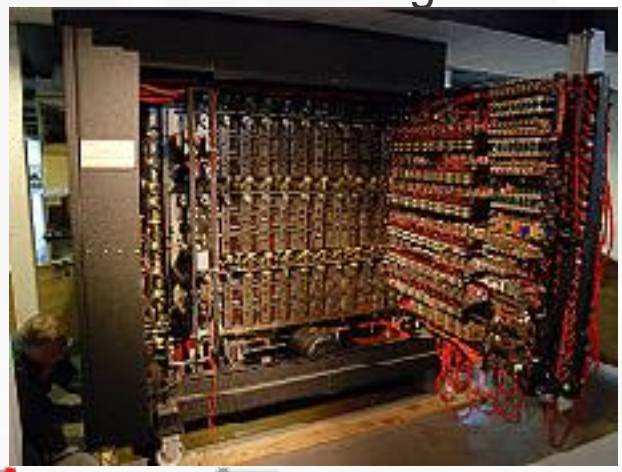
Combining three rotors from a set of five, the rotor settings with 26 positions, and the plugboard with ten pairs of letters connected, the military Enigma has 158,962,555,217,826,360,000 (nearly 159 quintillion) different settings.

Symmetric encryption

A story about Enigma

Alan Turing was the central force in continuing to break the Enigma code in the United Kingdom, during World War II.

The film *The Imitation Game* (2014) tells the story of Alan Turing and his attempts to crack the Enigma machine code during World War II.



Symmetric encryption

EXPAN

A story about Enigma

Symmetric encryption

A story about Enigma

Key(old) Key(new) Encrypted data
ABC → XYZ → PPX

XYZ → XYZ → QYP
Key(new) Raw data Encrypted data

Non-symmetric encryption

Suppose we have:

Two persons: a boy & a girl

Public data: a, p, y1, y2

where, a&p are prime numbers, p is quite big

Private data: x1, x2

Secret key: key

Non-symmetric encryption

Scenario 1:

Boy gives a private x1, and girl gives x2

Boy calculates: y1 = a^x1 mod p， gives it to girl

Girl calculates: y2 = a^x2 mod p， gives it to boy

Non-symmetric encryption

Scenario 2:

Boy gives a private x1, and girl gives x2

Boy calculates: Key1 = y2^x1 mod p

Girl calculates: Key2 = y1^x2 mod p

We can prove that Key1 = Key2:

Key1 =y2^x1 mod p = (a^x2)^x1 mod p = a^(x2*x1)mod p

Key2 = y1^x2mod p = (a^x1)^x2 mod p = a^(x1*x2) mod p

Lecture 3 Numeric Data Expression

数值(Numeric)资料表示法

- 整数表示法
  - 不带正负号表示法(Unsigned Integer)
  - 带正负号表示法(Signed Integer)
  - 1的补码表示法(
  - 2的补码表示法(2's
- 浮点数(实数)表示法

两个组合状态对应同一数
0。
算术运算很难处理。

Java supports only signed numbers

Lecture 3 Numeric Data Expression

C Declaration	Typical 32-bit	Compaq Alpha
char	1	1
short int	2	2
int	4	4
long int	4	8
char *	4	8
float	4	4
double	8	8

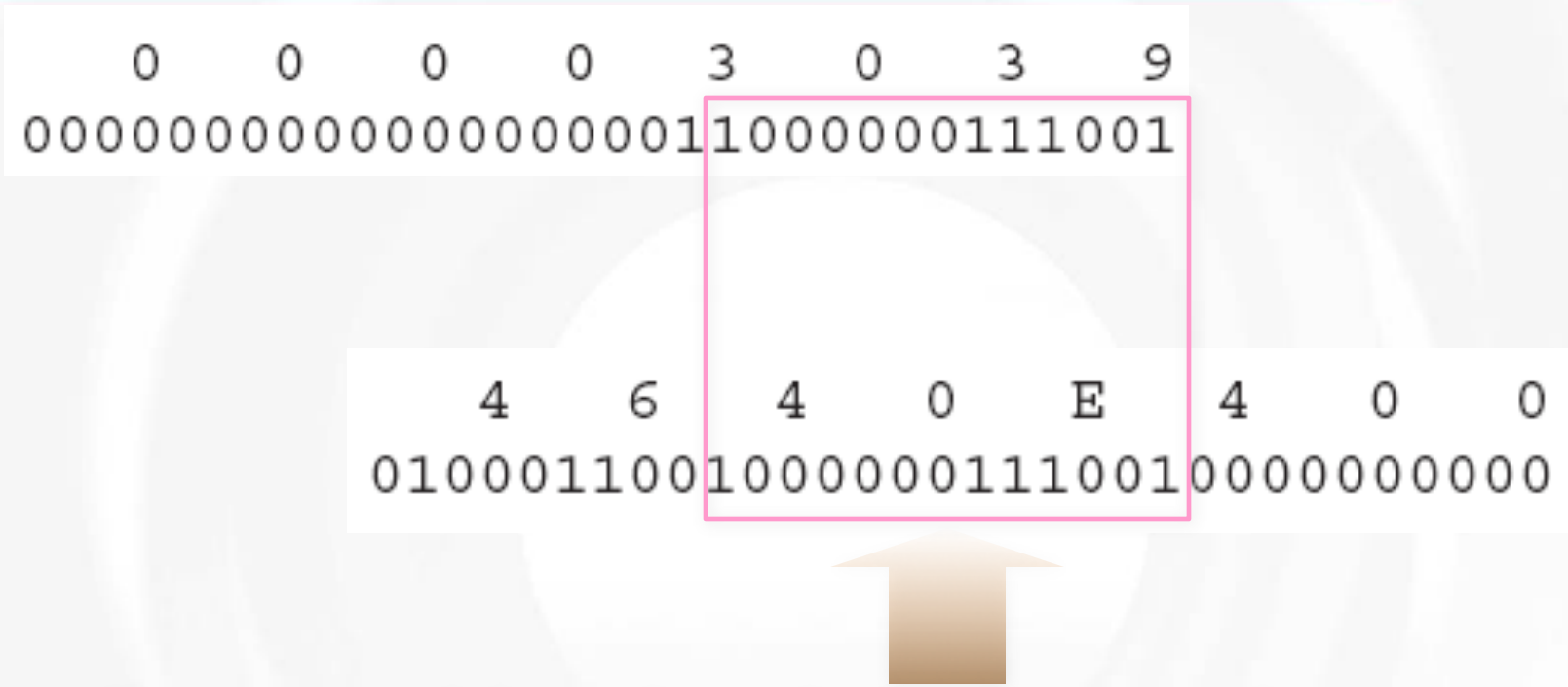
Sizes (in Bytes) of C Numeric Data Types. The number of bytes allocated varies with machine and compiler.

Lecture 3 Numeric Data Expression

Machine	Value	Type	Bytes (Hex)
Linux	12,345	int	39 30 00 00
NT	12,345	int	39 30 00 00
Sun	12,345	int	00 00 30 39
Alpha	12,345	int	39 30 00 00
Linux	12,345.0	float	00 e4 40 46
NT	12,345.0	float	00 e4 40 46
Sun	12,345.0	float	46 40 e4 00
Alpha	12,345.0	float	00 e4 40 46
Linux	&ival	int *	3c fa ff bf
NT	&ival	int *	1c ff 44 02
Sun	&ival	int *	ef ff fc e4
Alpha	&ival	int *	80 fc ff 1f 01 00 00 00

Byte Representations of Different Data Values. Results for int and float are identical, except for byte ordering. Pointer values are machine-dependent.

Lecture 3 Numeric Data Expression



If we expand these hexadecimal patterns into binary and shift them appropriately, we find a sequence of 13 matching bits, indicated above by a box.

Lecture 3 Numeric Data Expression

Type Forced Update

For example:

```
if (d <= TOTAL_ELEMENTS-2)
```

Homework:

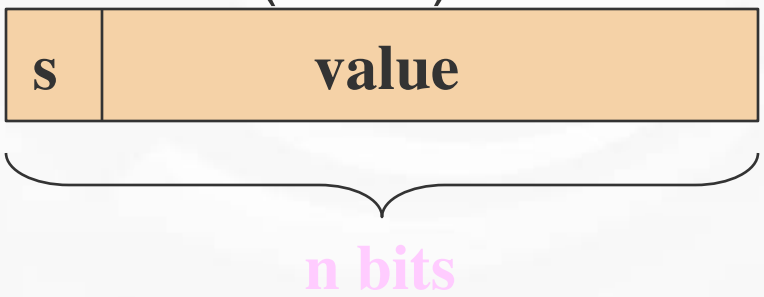
How to check if a number SIGNED or UNSIGNED by bit operation?

Lecture 3 Number Expression

带正负号表示法(Signed Integer)

- 以最前面一位表示正负号(+:0, -:1)
- 将数值以二进制值表示放入其余位中
- 若用n个位来表示一数则范围

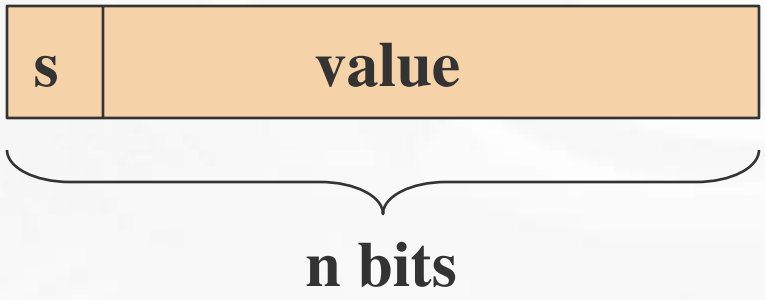
是$+2^{n-1}-1$ ~ $-(2^{n-1}-1)$



Lecture 3 Number Expression

带正负号表示法(Signed Integer)

- 右图是以八位表示signed Integer
- 注意:有+0, -0



value	signed integer
+0	00000000
+1	00000001
.	.
+127	01111111
-0	10000000
-1	10000001
.	.
-126	11111110
-127	11111111

Why need complement?

为了简化电路的设计，可将减法运算转换成加法及补码（complement）运算来取代。

如：

X-Y

=X+(-Y)

=X+(Y的補數)

Lecture 3 Number Expression

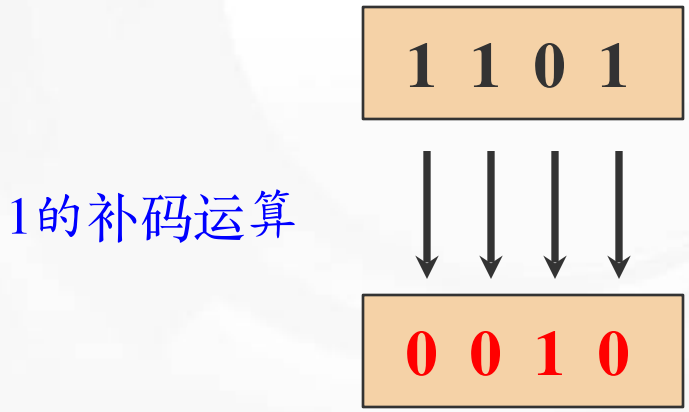
Why need complement?

- 补码是一种表示负数的方式
- 对每一 k 进位制的数字系统而言，其补码有两种：
  1. “k”的补码
     （又称基数补码：radix complement）
  2. “k-1”的补码
     （又称基数减一补码：radix minus one complement）

Lecture 3 Number Expression

1的补码表示法(1's complement)[或二进制反码]

- 1的补码运算(逐位not运算)
  - 二进制数值中每一位分别作not运算(1->0, 0->1)
  - 如对1101 执行 not 运算结果是 0010



Lecture 3 Number Expression

1的补码表示法(1's complement)

- 正数
  - 最前面一位为 0
  - 将数值以二进制值表示直接放入其余位中

- 如用16位表示 +5

  为 00000000 00000101 (0005_{16})

Lecture 3 Number Expression

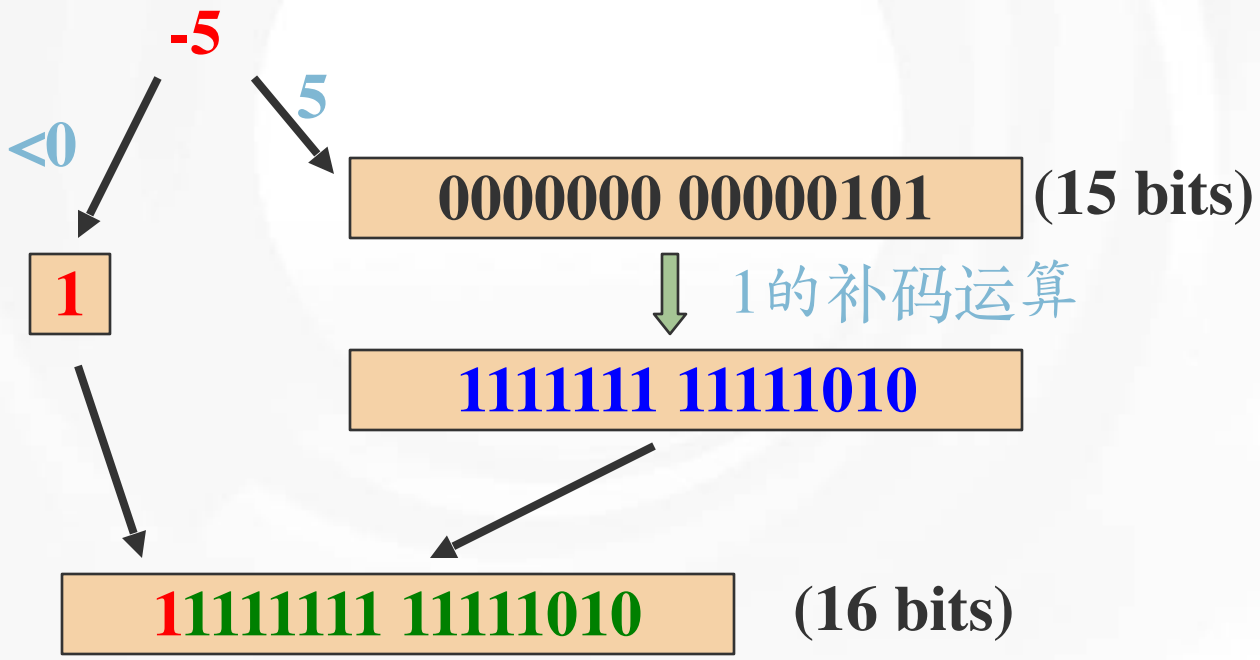
1的补码表示法(1's complement)

- 负数
  - 最前面一位为 1
  - 将数值之正数以二进制值表示,对其执行1的补码运算后,再将结果放入其余位中

Lecture 3 Number Expression

1的补码表示法(1's complement)

- 范例: 用16位表示 -5 为 FFFA



Lecture 3 Number Expression

1的补码表示法(1's complement)

- 右图是以八位表示

  1's complement integer

  注意:有+0, -0

- 若用n个位来表示

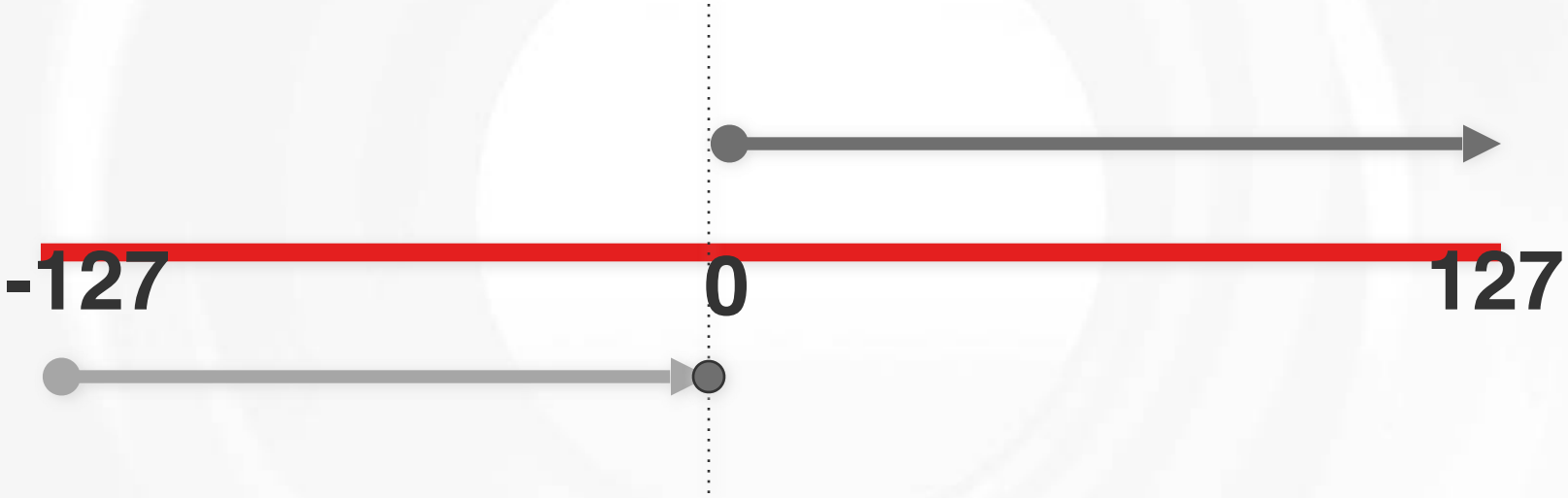
  一数则范围是

  $+2^{n-1}-1 \sim -(2^{n-1}-1)$

value	1's complement
+0	00000000
+1	00000001
.	.
+127	01111111
-127	10000000
-126	10000001
.	.
-1	11111110
-0	11111111

Lecture 3 Number Expression

1的补码表示法(1's complement)

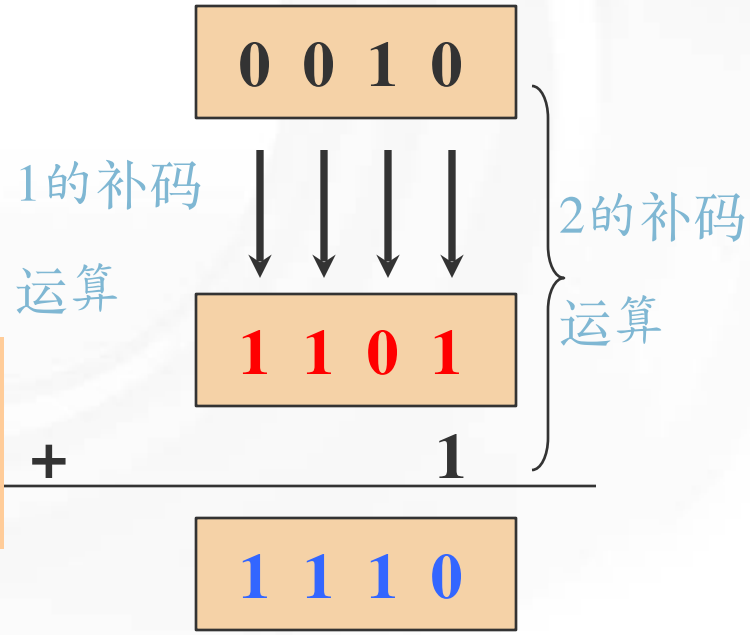


Lecture 3 Number Expression

2的补码表示法(2's complement)[现代计算机用]

- 2的补码运算
  - 二进制数值中每一位分别作not运算后再加1(溢位忽略)
  - 如0010 的2的补码是 1110
  - 口诀:

    2的补码=1的补码+1



$$B2T_{\omega}(x) = -x_{\omega-1}2^{\omega-1} + \sum_{i=0}^{w-2} x_i 2^i$$

Lecture 3 Number Expression

2的补码表示法(2's complement)

- 正數
  - 最前面一位为 0
  - 将数值以二进制值表示放入其余位中
  - 与正数以1的补码表示法相同

- 如用16位表示+5为

  00000000 00000101 (0005_{16})

Lecture 3 Number Expression

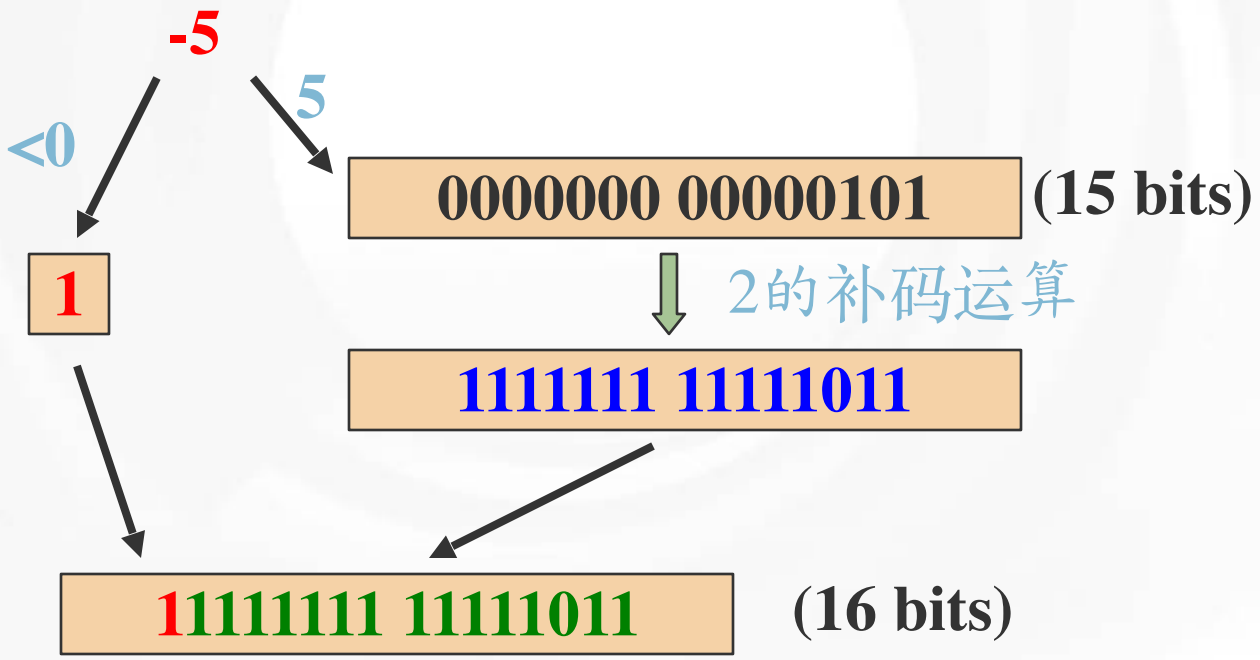
2的补码表示法(2's complement)

- 负数
  - 最前面一位为 1
  - 将数值之正数以二进制值表示,对其执行2的补码运算后,再将结果放入其余位中

Lecture 3 Number Expression

2的补码表示法(2's complement)

- 范例:用16位表示 -5 为 FFFB



Lecture 3 Number Expression

2的补码表示法(2's complement)

- 右图是以八位表示

  2's complement integer
- 注意:没有+0, -0
- 若用n个位来表示

  一数则范围是

  $+2^{n-1}-1 \sim -2^{n-1}$

value	2's complement
0	00000000
+1	00000001
.	.
+127	01111111
-128	10000000
-127	10000001
.	.
-2	11111110
-1	11111111

Lecture 3 Number Expression

“1 的补码”的特点：

- 计算容易，只需将 0 → 1；1 → 0 即可。
- 有正零与负零两种表示法容易混淆。

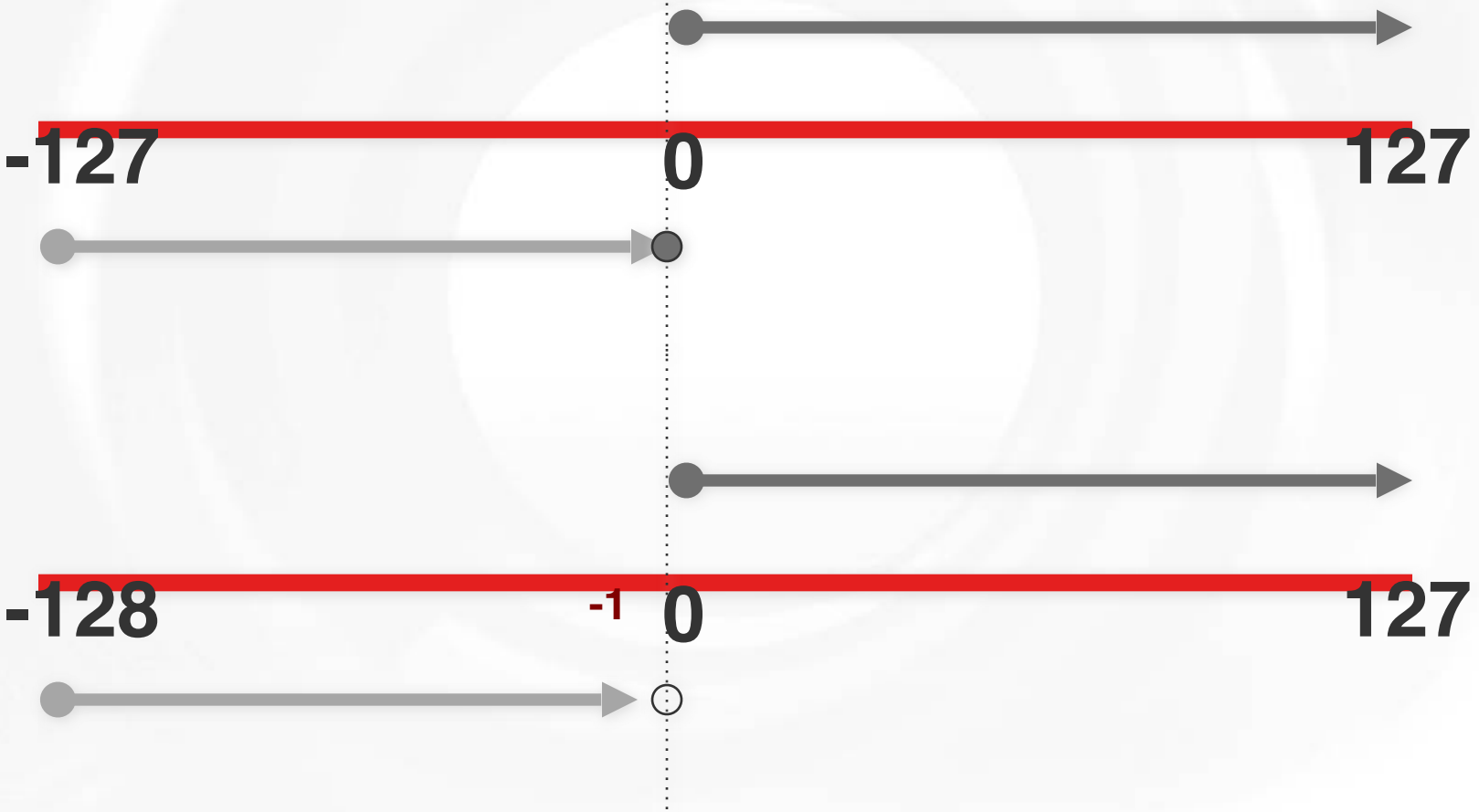
  以4bit为例，0000为正零，1111为负零
- 利用 1 的补码执行减法运算时，进位需做加法处理。

“2的补码”的特点：

- 零的表示法只有一种。
- 范围较“1 的补码”大
- 计算“2 的补码”的过程比计算“1的补码”复杂。
- 现代计算机使用

Lecture 3 Number Expression

2的补码 VS. 1的补码



Lecture 3 Number Expression

范例：

把 short 型-1以无符号类型输出，是多少?

1、假设short类型是2字节
2、则无符号是0~65535， 有符号为-32768~32767
3、-1在计算机中补码是：11111111，11111111
4、转换为无符号类型，则直接为11111111，11111111
5、换算为10进制，为65535

Lecture 3 Number Expression

范例: 对3做“2的补码运算”, 对结果再做“2的补码运算”

- 3以8bit表示法是 0000 0011 +3
- 上式之2的补码运算结果是 1111 1101 -3
- 上式之2的补码运算结果是 0000 0011 -(-3) => +3

Lecture 3 Number Expression

范例: 1的补码表示法 1110 1010 是代表何值?

- 第一位是1代表其值为负
- 其余位110 1010需执行1的补码运算 ,以求其正值
  - 对110 1010 执行1的补码运算之结果为 001 0101(21_{10})
  - 因为1的补码运算结果是+21, 所以原式是代表 -21

Lecture 3 Number Expression

范例: 2的补码表示法 1000 0000 是代表何值?

- 第一位是1代表负值
- 其余位000 0000需执行2的补码运算 ,以求其正值
  - 000 0000 之2的补码运算(1的补码+1)结果为 $10000000(128_{10})$
  - 因为运算结果是+128 所以原式是代表 –128
- 用八位的2的补码表示法范围 –128 ~ +127

Lecture 3 Number Expression

范例: 0000 0000 执行2的补码表示法结果为何?

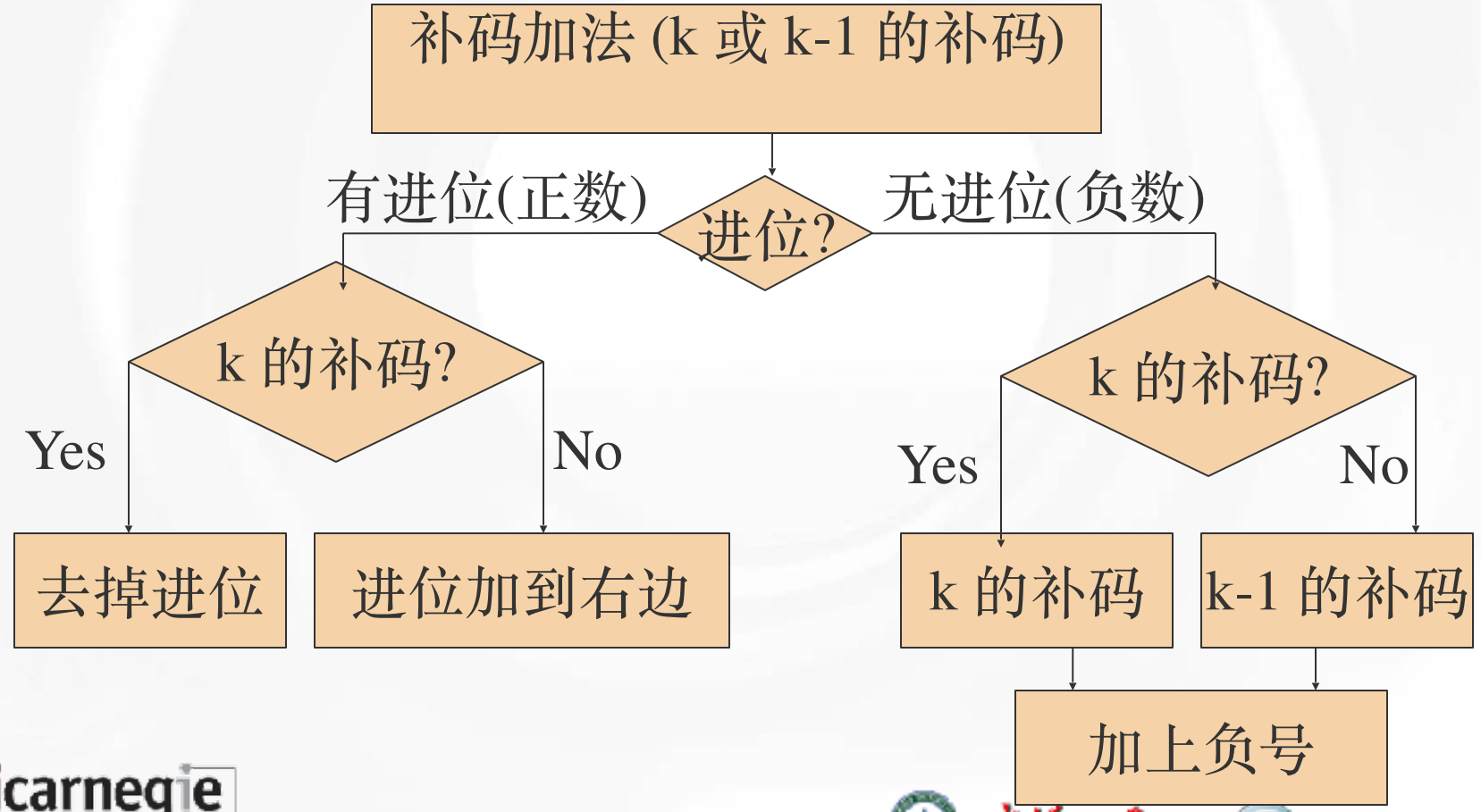
- 0000 0000 执行2的补码运算(1的补码+1)得到 1 0000 0000
- 忽略溢位(只有八个位)则得 0000 0000 (0_{10})

由此亦可验证2的补码表示法并没有+0, -0 之分

Lecture 3 Number Expression

CONCLUSIONS： You will also find:

K's Complement



Lecture 3 Number Expression

浮点数(实数)表示法 [previous]

- Fixed-Point Representations
- Some of the bits represent a fractional part- just as we use digits to the right of the decimal point to denote the fractional part of a number
- Each sample of digital audio is a 16-bit, fixed-point binary number representing a fraction between -1 and +1

Lecture 3 Number Expression

Positive side on Fix-Point formats:

- Allow fractions to be represented to any desired precision

Negative side

- Precision lost in computations
- Worse, the problem is that in many computations the magnitudes vary widely:

If you keep the decimal point in a fixed location, then any time you divide or multiply by a value that is not close to one, you will risk either overflow because the number becomes too large, or a loss of precision, because the number becomes too small

Lecture 3 Number Expression

浮点数(实数)表示法 [at present]

- 克服以上缺点
- 可表示有小数点之数值(fractional)
- 表示的范围很大,但精度(有效位数)有限制
- 不同计算机内使用的浮点数格式可能,视CPU设计而定
- 考虑精确性多于考虑速度和效率

Lecture 3 Number Expression

浮点数(实数)表示法 [examples]

Decimal	Binary	Floating-Point	
		Exponent	Mantissa
5	101	11	.10100000
10	1010	100	.10100000
20	10100	101	.10100000
40	101000	110	.10100000
80	1010000	111	.10100000
160	10100000	1000	.10100000

Lecture 3 Number Expression

浮点数(实数)表示法原理[principles]

- The left-most bit of the mantissa is always one
- By keeping the mantissa shifted as far left as possible, it will always have the most room to the right and, therefore, the most precision
- Floating-point numbers are usually approximations, so we want to shift the mantissa left (and adjust the exponent accordingly) as much as possible.
- To skip the left most 1 makes room for one more bit of precision

Lecture 3 Number Expression

Intel x86 CPU三种浮点数格式

- IEEE规格(Institute of Electrical and Electronics Engineers)
- 单精度(Single Precision)格式
  - 4 字节
- 双精度(Double Precision)格式
  - 8 字节
- 扩展双精度(Extended Double Precision)格式
  - 10 字节

Lecture 3 Number Expression

IEEE 4 bytes浮点数格式组成及概念

- 正负号(Sign)(b_{31})
  - + : 0, - : 1
- 指数(Exponent)(b_{30} ...b_{23})
  - 资料表示法 :unsigned
  - 指数次方值(P_e) : powered by 2
  - 指数偏差值(bias):127
- 底数(Mantissa) (b_{22}...b_0)
  - 数值正规化
  - 假想小数点 : 1.

IEEE 4 bytes浮点数格式:指数部分(Exponent)

- 长度:8个位
- 资料表示法
  - 不带正负号整数(0~255)
- 指数次方值 (Pe)
  - 指数是2的次方倍
- 指数偏差值(bias)
  - 以127为偏差值,所以真正的指数值是 -127 ~ +128

Lecture 3 Number Expression

IEEE 4 bytes浮点数格式:底数部分(Mantissa)

- 长度:23个位
- 数值基底(Bm) :Base 2
- 数值正规化
- 假想小数点 : 1.

Lecture 3 Number Expression

正规化(Normalization)

- 用底数及指数方式表示一个实数并不唯一
  - 如

$$11.101_2 = 11101_2 \times 2^{-3}$$
$$= 0.11101_2 \times 2^{2}$$
$$= 1.1101_2 \times 2^{1} = \ldots$$

- 限定小数点出现的位置则称正规化
  - IEEE 限定小数点出现在第一个1之后

- 0 的浮点数表示法为特例: 32 位均为 0

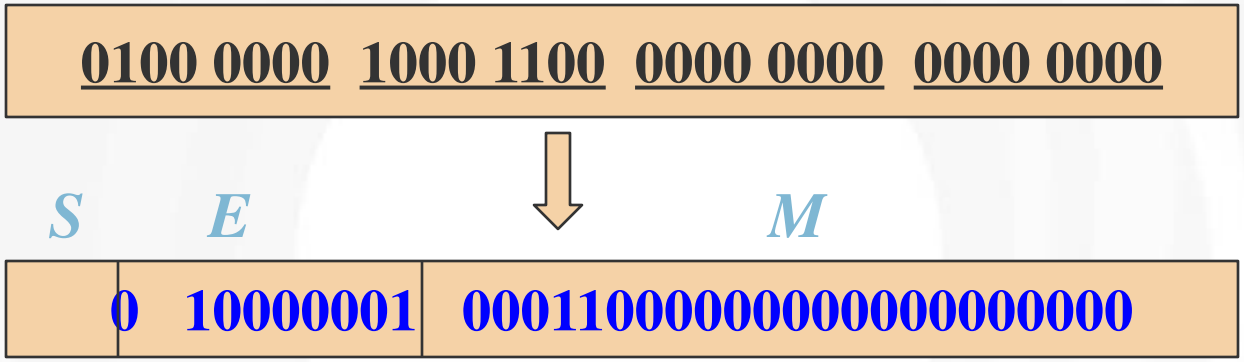
Lecture 3 Number Expression

-1^Sign * 2^(Exponent-127) * (1 + Mantissa * 2^-23)

- *Sign* is either 0 or 1.
- *Exponent* determines the general magnitude of the floating-point number
- To represent small numbers, *Exponent* is *biased* by -127
- So, 2^-127, is about 0.00000000000000000000000000000000000059 In any case, it is a very small number
- *Mantissa*, a 23-bit unsigned value, ranges from 0 to 2^23-1. And (*Mantissa* * 2^-23) ranges from 0 to almost 1
- Finally, floating-point value varies from 2^-127 (10^-38) to 2^128 (10^38).

Lecture 3 Number Expression

范例: IEEE 4 bytes浮点数 40 8D 00 00 代表何值



$$value = S \times (1.M)_2 \times 2^{E-127}$$

$$= +(1.00011)_2 \times 2^{129-127}$$

$$= +(1.00011)_2 \times 2^2 = 100.011_2$$

$$= 4.375_{10}$$

Lecture 3 Number Expression

范例: -66.5以IEEE 4 bytes 的浮点数格式表示

$$-66.5_{10} = -1000010.1_2$$

正规化 $$= -1 \times (1.0000101)_2 \times 2^6$$

$$= -1 \times (1.0000101)_2 \times 2^{133-127}$$

$$= S \times (1.M)_2 \times 2^{E-127}$$

$S = 1$

$M = 0000101_2$

$E = 133_{10}$

$= 10000101_2$

S	E	M
1	10000101	00001010000000000000000

1100 0010 1000 0101 0000 0000 0000 0000 → C2 85 00 00

Lecture 3 Number Expression

How to represent π?

$$\frac{223}{71} < \pi < \frac{22}{7}$$ (Archimedes, 250 B.C.)

0x40490FDB

Lecture 3 Number Expression

A story on Floating Point numbers

On February 25, 1991, during the GulfWar, an American PatriotMissile battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks and killed 28 soldiers. The U. S. General Accounting Office (GAO) conducted a detailed analysis of the failure and determined that the underlying cause was an imprecision in a numeric calculation.

Lecture 3 Number Expression

A story on Floating Point numbers

The Patriot system contains an internal clock, implemented as a counter that is incremented every 0.1 seconds. To determine the time in seconds, the program would multiply the value of this counter by a 24-bit quantity that was a fractional binary approximation to 1/10 . In particular, the binary representation of 1/10 is the nonterminating sequence:

$$0.000110011[0011]\ldots\ldots_2$$

where the portion in brackets is repeated indefinitely. The computer approximated 0.1 using just the leading bit plus the first 23 bits of this sequence to the right of the binary point. Let us call this number x.

A story on Floating Point numbers

A. *What is the binary representation of x – 0.1?*
B. *What is the approximate decimal value of x – 0.1?*
C. *The clock starts at 0 when the system is first powered up and keeps counting up from there. In this case, the system had been running for around 100 hours. What was the difference between the time computed by the software and the actual time?*

Lecture 3 Number Expression

A story on Floating Point numbers

A. $0.0000000000000000000000001100[1100]\cdots_2$

B. $9.54 \times 10^{-8} \times 100 \times 60 \times 60 \times 10 \approx 0.343.$

C. $0.343 \times 2000 \approx 687.$

Lecture 3 Number Expression

Special Values in IEEE Floating Point numbers

- e_{min} & e_{max} refer to the boundary of normal EXP values, *i.e.* -126 & 127
- Thus, those special EXP values for FP are e_{min} - 1 & e_{max} + 1;

指数 Exponent	尾数 Mantissa	值 Values	
$e_{min} \le e \le e_{max}$	——	$1.f \times 2^e$	Normal
$e = e_{min} - 1$	$f \neq 0$	$0.f \times 2^{e_{min}}$	Denormalization
$e = e_{min} - 1$	$f = 0$	0	Zero
$e = e_{max} + 1$	$f \neq 0$	NaN	Not-A-Number
$e = e_{max} + 1$	$f = 0$	∞	Infinite

Lecture 3 Number Expression

How to represent 0?

- *Zero* is represented by an exponent of zero ("0") and a mantissa of zero ("0").
- *Sign* bit can be either "0" or "1“: +0 and -0

Exponent 0?

When the exponent is 0, the mantissa represents a number between 0 and 1 rather than between 1 and 2. These denormalized values are used when the magnitude is too small to be expressed by the exponent alone. Denormalized values can have magnitudes as small as 10^-45-but at reduced precision since some high-order bits of the mantissa will be zero.

Lecture 3 Number Expression

Exponent 11111111

An exponent of 11111111 (binary) with a zero mantissa is used to represent plus or minus infinity, two special values that are reserved to indicate the result of an overflowing arithmetic operation. An Exponent of 11111111 (binary) with a non-zero Mantissa denotes a special value called NaN (for "Not-a-Number") for cases where a number cannot be determined, for example, the square root of -1.0.

DeNormalized Values

- Denormalized numbers serve two purposes
- First, they provide a way to represent numeric value 0, since with a normalized number we must always have M >= 1, and hence we cannot represent 0.
- With IEEE floating-point format, the values are +0 & -0 are considered different in some ways and the same in others.
- A second function of denormalized numbers is to represent numbers that are very close to 0.0.

Lecture 3 Number Expression

Rounding

Floating-point arithmetic can only approximate real arithmetic, since the representation has limited range and precision. Thus, for a value *x*, we generally want a systematic method of finding the "closest" matching value *x'* that can be represented in the desired floating-point format.

Mode	$1.40	$1.60	$1.50	$2.50	$–1.50
Round-to-even	$1	$2	$2	$2	$–2
Round-toward-zero	$1	$1	$1	$2	$–1
Round-down	$1	$1	$1	$2	$–2
Round-up	$2	$2	$2	$3	$–1

Lecture 3 Number Expression

Rounding (assuming a 32-bit int):

- From *int* to *float*, the number cannot overflow, but it may be rounded.
- From *int* or *float* to *double*, the exact numeric value can be preserved because *double* has both greater range (i.e., the range of representable values), as well as greater precision (i.e., the number of significant bits).
- From *double* to *float*, the value can overflow to +∞or - ∞, since the range is smaller. Otherwise it may be rounded since the precision is smaller

Lecture 3 Number Expression

Rounding (assuming a 32-bit int):

- From *float* or *double* to *int* the value will be truncated toward zero. For example 1.999 will be converted to 1, while -1.999 will be converted to -1. Note that this behavior is very different from rounding. Furthermore, the value may overflow. The C standard does not specify a fixed result for this case, but on most machines the result will either be $TMax_w$ or $TMin_w$, where w is the number of bits in an int.

- *See double_number.dsw*

Lecture 3 Number Expression

A story on Rounding

Converting large floating-point numbers to integers is a common source of programming errors. Such an error had particularly disastrous consequences for the maiden voyage of the Ariane 5 rocket, on June 4, 1996. Just 37 seconds after lift-off, the rocket veered off its flight path, broke up, and exploded. On board the rocket were communication satellites, valued at $500 million.

Lecture 3 Number Expression

A story on Rounding

- A later investigation showed that the computer had sent invalid data to the computer controlling the engine nozzles. Instead of sending flight control information, it had sent a diagnostic bit pattern indicating that, in an effort to convert a 64-bit floating point number into a 16-bit signed integer, an overflow had been encountered.
- The value that overflowed measured the horizontal velocity of the rocket, which could be more than five times higher than that achieved by the earlier Ariane 4 rocket. In the design of the Ariane 4 software, they had carefully analyzed the numeric values and determined that the horizontal velocity would never overflow a 16-bit number. Unfortunately, they simply reused this part of the software in the Ariane 5 without checking the assumptions on which it had been based.

Lecture 3 Number Expression

总结： Intel x86 CPU三种浮点数格式比较

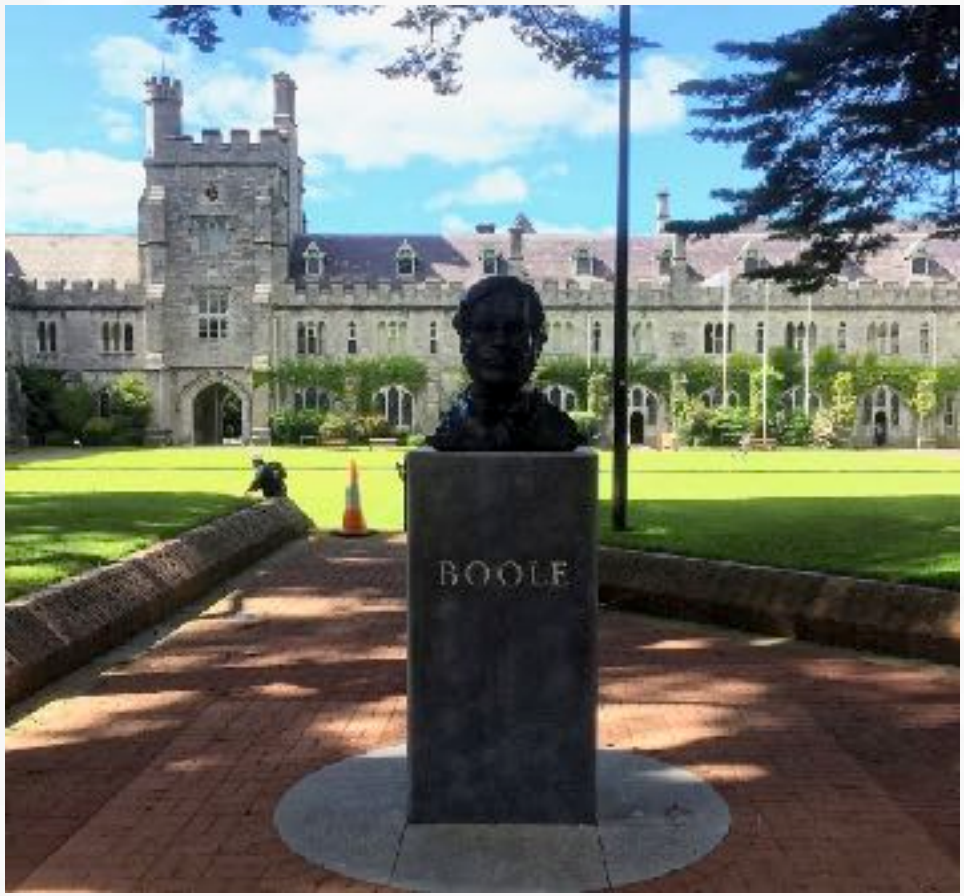
	单精度	双精度	扩展双精度
总字节数	32	64	80
底数字节数	23	52	63
有效位数	7	15	19
指数字节数	8	11	15
指数偏差值	127	1023	16383
表示范围	10^{-38}~10^{38}	10^{-308}~10^{308}	10^{-4932}~10^{4932}

Lecture 3 Boolean Algebra

- From George Boole (1850), so as to *Boolean Algebra*
- Boolean Algebra(布尔代数)
- Boolean Variable(布尔变量):
  - 0 和 1
  - true and false
- Logic operations
  - AND, OR, XOR(Exclusive OR), NOT …, etc.

Lecture 3 Boolean Algebra

George Boole (1815~1864)



UCC:
University College Cork
June 30, 2017

Lecture 3 Boolean Algebra

George Boole (1815~1864)

UCC:
University College Cork
June 30, 2017

Lecture 3 Boolean Algebra

Logic operation vs. propositional logic

~p	Not	¬p
P & Q	And	P ∧ Q
P I Q	Or	P v Q
P ^ Q	Exclusive-Or	P ª Q

Lecture 3 Boolean Algebra

~	
0	1
1	0

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

Shared properties

Property	Integer ring	Boolean algebra
Commutativity	$a+b=b+a$ $a \times b=b \times a$	$a \mid b=b \mid a$ $a \& b=b \& a$
Associativity	$(a+b)+c=a+(b+c)$ $(a \times b) \times c=a \times (b \times c)$	$(a \mid b) \mid c=a \mid (b \mid c)$ $(a \& b) \& c=a \& (b \& c)$
Distributivity	$a \times (b+c)=(a \times b)+(a \times c)$	$a \& (b \mid c)=(a \& b) \mid (a \& c)$
Identities	$a+0=a$ $a \times 1=a$	$a \mid 0=a$ $a \& 1=a$
Annihilator	$a \times 0=0$	$a \& 0=0$
Cancellation	$-(-a)=a$	$\sim(\sim a)=a$

Lecture 3 Boolean Algebra

Unique to RINGS

Inverse	a + -a = 0	--

Unique to Boolean algebras

Distributivity	--	a | (b & c) = (a | b) & (a | c)
Complement	--	a | ~a = 1 a & ~a = 0
Idempotency	--	a & a = a a | a = a
Absorption	--	a | (a & b) = a a & (a | b) = a
DeMorgan's laws	--	~ (a & b) = ~a | ~b ~ (a | b) = ~a & ~b

- AND(与):两数皆为1时，结果为1，否则为0。
- OR(或):两数有一为1时，结果为1，否则为0。
- XOR(互斥):两数不相同时，结果为1，否则为0。
- NOT(非): 1‘s complement.

C Expression	Binary Expression	Binary Result	C Result
~0x41	~[01000001]	[10111110]	0xBE
~0x00	~[00000000]	[11111111]	0xFF
0x69 & 0x55	[01101001] & [01010101]	[01000001]	0x41
0x69 | 0x55	[01101001] | [01010101]	[01111101]	0x7D

```
void inplace_swap ( int *x, int *y)
{
        *x = *x ^ *y;  //step 1
        *y = *x ^ *y;  //step 2
        *x = *x ^ *y;  //step 3
}
```

What if using float type instead of int?

Wrong! Bit operations must be done between two INT operands!

Lecture 3 Logic Operation

- || && ! >> <<
- Logic operation treats any nonzero argument as representing TRUE and argument 0 as representing FALSE
- They return either 1 or 0, indicating a result either TRUE or FALSE

Expression	Value	Expression	Value
x & y	0x02	x && y	0x01
x | y	0xF7	x || y	0x01
~x | ~y	0xFD	!x || !y	0x00
x & !y	0x00	x && ~y	0x01

X=0x66
Y=0x93

Lecture 3 Numeric Data Expression

Type Forced Update

For example:

```
if (d <= TOTAL_ELEMENTS-2)
```

Homework:

How to check if a number SIGNED or UNSIGNED by bit operation?

Lecture 3 Next Time……

- To be finished:
  - **All quizzes in Unit 1**
  - **Excise 1**
- Pre-Read:
  - **Memory Layout and Allocation**
    - 3.1 Several Uses of Memory
    - 3.2 Memory Bugs
- Do:
  - **Multiple-Choice Quizzes （Unit 2）**

Lecture 3 In the Lab……

- Finish Excise 1:
- Do:
  - **Excise 2**