



中山大學
SUN YAT-SEN UNIVERSITY



计算机组成原理

第二章：指令：计算机的语言

中山大学计算机学院
陈刚

2022年秋季

本讲内容

□ 什么是计算机语言？

□ 指令集概述 Introduction to Instruction Set

□ 指令格式

□ 寻址方式

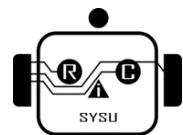
□ 数据表示方法

□ 数值数据表示

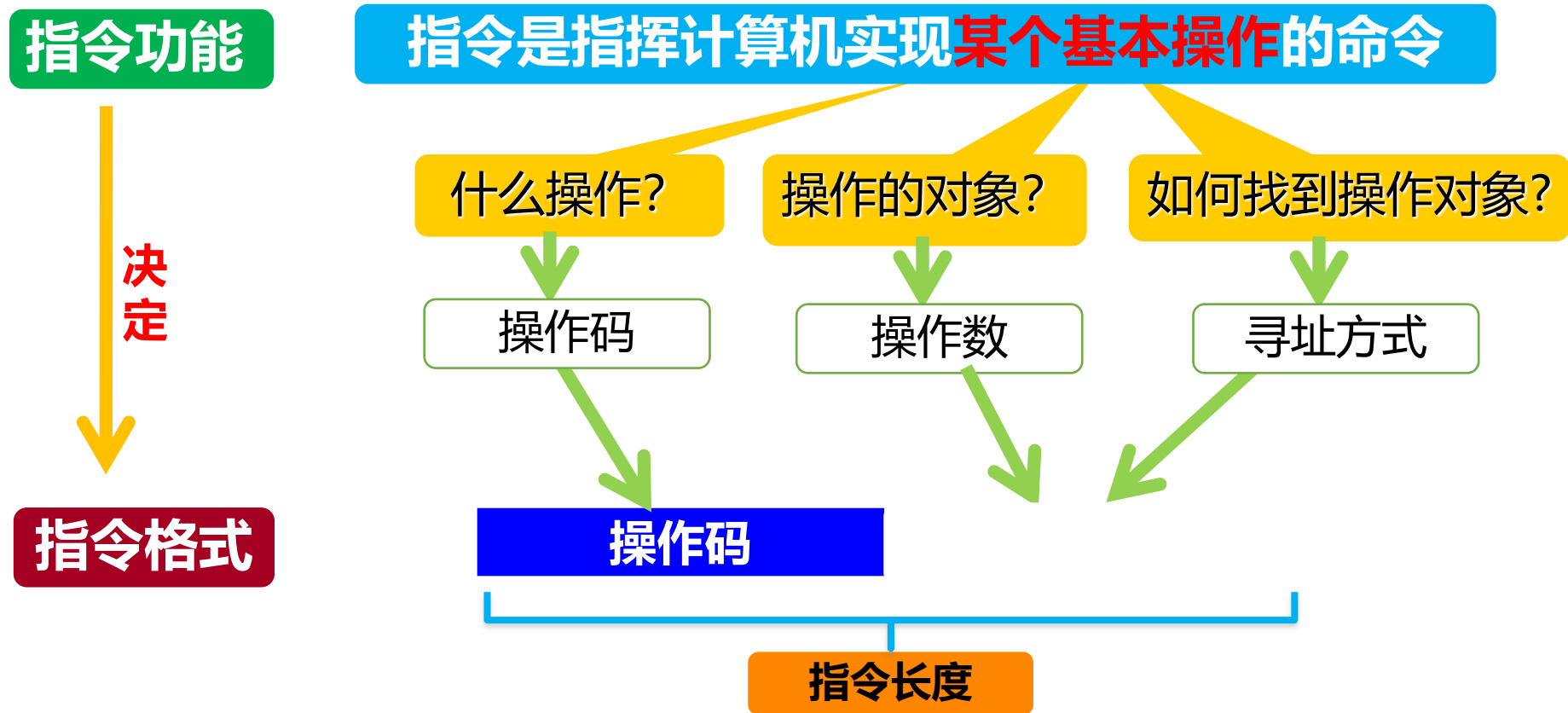
□ 非数值数据表示

□ 数据的存储

□ 数据的校验



回顾——指令格式



回顾—— 操作码设计

指令格式

操作码

地址码

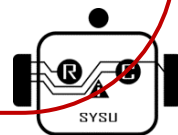
操作码长度

操作码长度问题

- 操作码的编码方式决定操作码的长度
 - Fixed Length Opcodes (定长操作码法)

- 指令的操作码部分采用**固定长度**的编码

例如：假设操作码固定为6位，则系统最多可表示 2^6 种指令



回顾—— 操作码设计

指令格式

操作码

地址码

操作码长度

操作码长度问题

□ 操作码的编码方式决定操作码的长度

➤ Fixed Length Opcodes (定长操作码法)

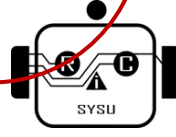
- 指令的操作码部分采用**固定长度**的编码

例如：假设操作码固定为6位，则系统最多可表示 2^6 种指令

➤ Expanding Opcodes (变长/扩展操作码法)

- 指令的操作码部分采用**可变长度**的编码

操作码的编码长度分成**几种固定长**的格式，操作码的位数**随地址数的减少而增加**，被大多数指令集采用



回顾——地址码结构

□ 一条指令包含1个**操作码**和多个**地址码**

■ 零地址指令

OP

(1) 无需操作数。如：空操作／停机等；(2) 所需操作数为默认的。如：堆栈等

■ 一地址指令

OP

A1

其地址既是源操作数地址，也是存放结果地址

(1) 单目运算：如：取反／取负等；(2) 双目运算：一操作数为默认的

■ 二地址指令(最常用)

OP

A1

A2

将其中一个源操作数地址作为存放结果地址

■ 三地址指令(RISC风格)

OP

A1

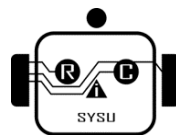
A2

A3

分别为双目运算中两个源操作数地址和一个结果地址

■ 多地址指令

用于成批数据处理的指令，如：向量指令等



回顾——基本寻址方式

■ 指令寻址——简单

- 正常：PC增值
- 跳转 (jump / branch / call / return)：同操作数的寻址

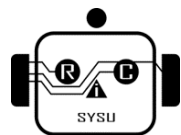
■ 操作数寻址——复杂

■ **寻址方式：**用来指定操作数或操作数所在位置的方法

- 形式地址：**指令中地址字段给出的逻辑地址**
- 有效地址：**根据指令的寻址方式计算得到的操作数地址**

■ 基本寻址方式

立即 / 直接 / 间接 / 寄存器 / 寄存器间接 / 偏移等



本讲内容

□ 什么是计算机语言？

□ 指令集概述 Introduction to Instruction Set

□ 指令格式

□ 寻址方式

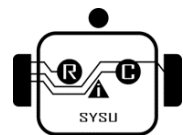
□ 数据表示方法

□ 数值数据表示

□ 非数值数据表示

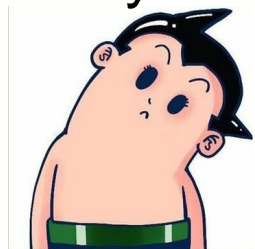
□ 数据的存储

□ 数据的校验



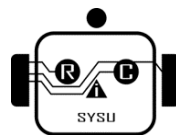
计算机中的（机器级）数据表示

Why do all the information inside the computer be represented by binary encoding?



为什么计算机内部所有信息都采用二进制编码表示？

- (1) 制造二个稳定态的物理器件比较容易
- (2) 二进制的编码、计数、运算规则简单
- (3) 与逻辑命题对应，便于逻辑运算，并能方便地用逻辑电路实现算术运算



计算机中的（机器级）数据表示

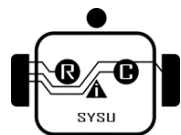


为什么计算机内部所有信息都采用二进制编码表示？

- (1) 制造二个稳定态的物理器件比较容易
- (2) 二进制的编码、计数、运算规则简单
- (3) 与逻辑命题对应，便于逻辑运算，并能方便地用逻辑电路实现算术运算

真值和机器数 Truth number and machine number

- 机器数：用0和1编码的计算机内部 0/1 序列
- 真值：机器数真正的值，即：现实中带正负号的数，人类认知中的数



计算机中的（机器级）数据表示



数据在计算机中是如何表示的？

正数？负数？整数？小数？逻辑数？

数据表示 —— 能被计算机**硬件直接识别**的数据类型

¥ & *
%

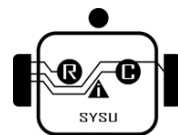
咱只听得懂
机器语言



1. 可用计算机硬件直接表示
2. 可以由计算机指令直接调用

(Data that can be directly represented by computer hardware.

Data that can be directly operated by a computer)



计算机中的（机器级）数据表示



数据在计算机中是如何表示的？

正数？负数？整数？小数？逻辑数？

数据表示——能被计算机**硬件直接识别**的数据类型

1. 可用计算机硬件直接表示
2. 可以由计算机指令直接调用

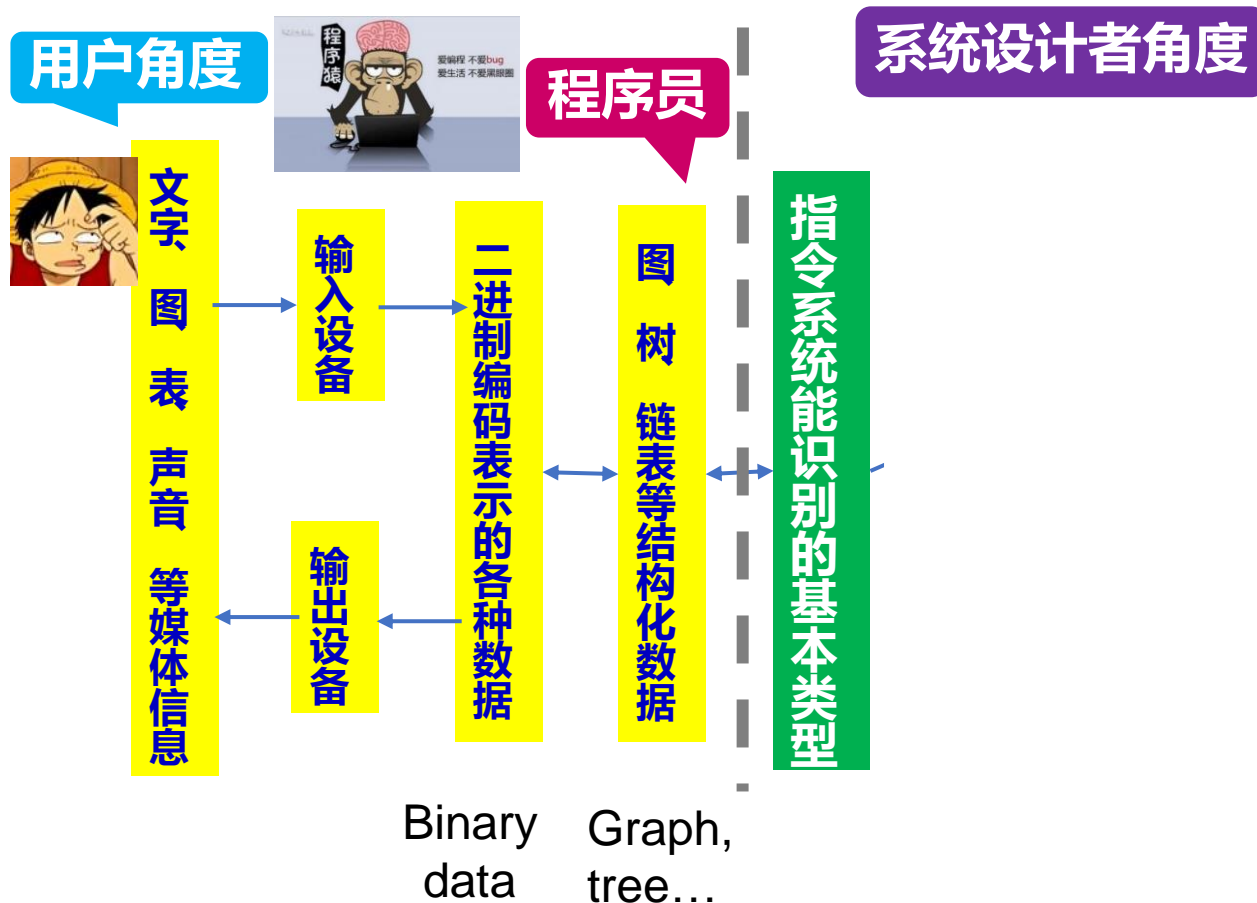
数据表示和数据结构的关系

data representation
and data structure

- 数据表示研究计算机**硬件可以直接识别**的数据类型
- 数据结构研究在数据表示基础之上，如何让计算机处理 **硬件不能直接识别**的数据类型

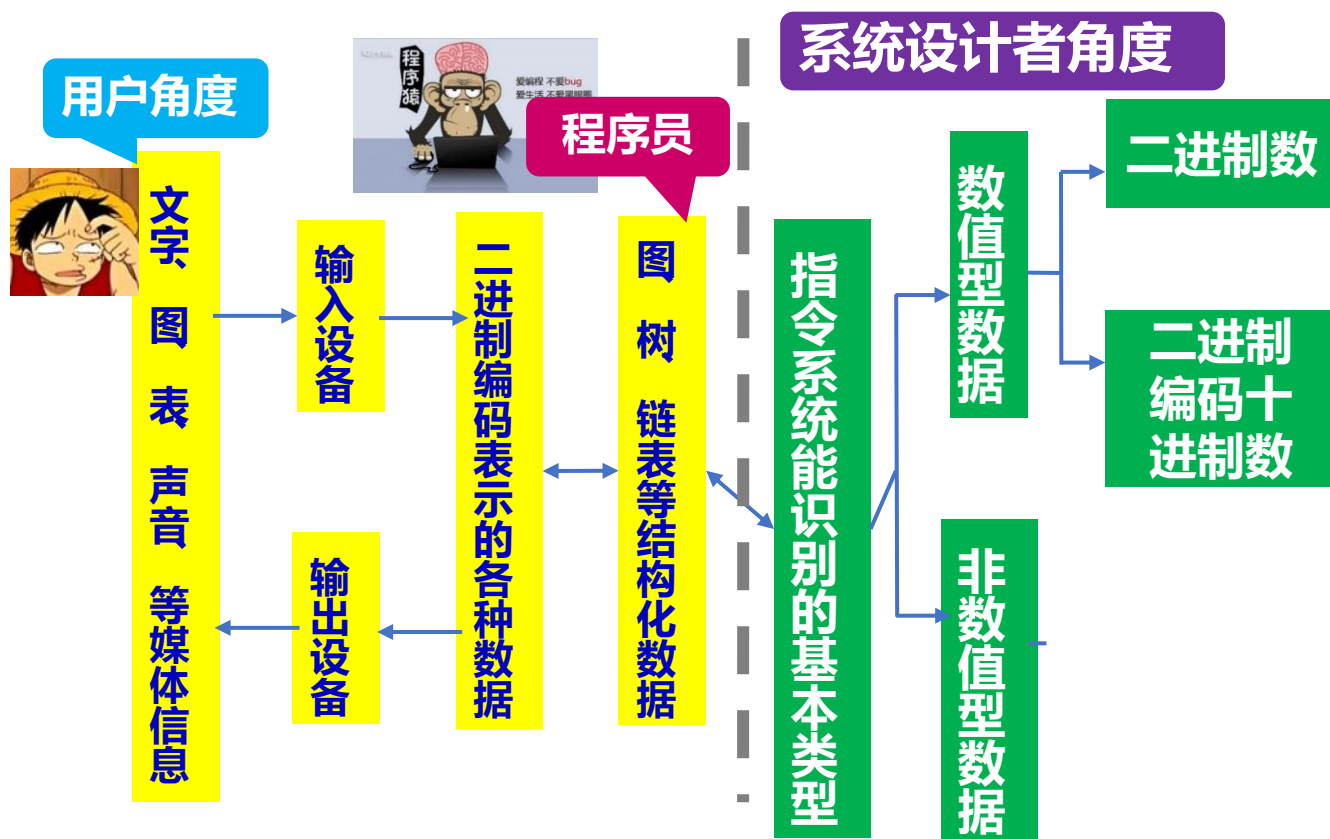
计算机中的（机器级）数据表示

计算机的外部信息与内部机器级数据



计算机中的（机器级）数据表示

计算机的外部信息与内部机器级数据

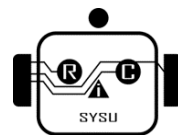


计算机中的（机器级）数据表示

C语言支持的整数和浮点数的各种数据类型

C语言 声明	Intel-IA (32) 数据类型(byte)	Compaq-Alpha (64) 数据类型(byte)
char	1	1
short int	2	2
int	4	4
unsigned int	4	4
long int	4	8
char *	4	8
float	单精度(4)	单精度(4)
double	双精度(8)	双精度(8)

C语言中数据类型的大小是以字节为单位



计算机中的（机器级）数据表示

数据宽度

位 bit

计算机处理、存储、传输信息的最小单位 (bit)

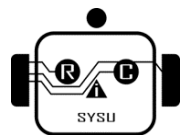
字节 byte

计算机中二进制信息的主要存放单位 (Byte)

现代计算机的主存是按字节编址，字节是最小可寻址单位

字 word

表示被处理信息的单位 (word)，用来度量数据类型的宽度



计算机中的（机器级）数据表示

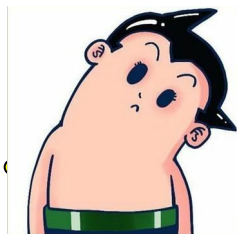
数据宽度

位 计算机处理、存储、传输信息的最小单位 (bit)

字节 计算机中二进制信息的计量单位 (Byte)
现代计算机的主存按字节编址，字节是最小可寻址单位

字 表示被处理信息的单位 (word)，用来度量数据类型的宽度

字长就是字的长度吗？

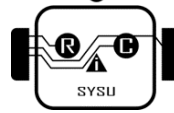


“字 word” 和 “字长 word length” 概念不同：

➤ 字长是指CPU中**数据通路的宽度**，等于CPU内部总线的宽度或运算器的位数或通用寄存器的宽度等

➤ 字是处理信息的单位，和字长的宽度可以一样，也可以不同，通常是字节的整数倍

当代的计算机已不再区分这两个概念



数值数据的定点表示

常用的数值数据

定点数、浮点数和十进制数

要解决的问题

第一个问题：正数与负数的表示？

Positive and negative representations

第二个问题：小数点的表示？

Representation of decimal point

第三个问题：零的表示？ Representation of zero

第四个问题：实数的表示？

Representation of real number

数值数据的定点表示



问题一：如何表示正数和负数？

解决方法：所有数前面**设置符号位**

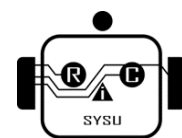


- '0' 表示正数； '1' 表示负数
- 第1位不具备数值的性质——符号的编码
- 原码、补码、反码(反码很少用)、移码

例1： $X = +90$ (十进制真值) = 1011010 (二进制真值)
用八位二进制原码表示：**0** 1011010

例2： $X = -90$ (十进制真值) = - 1011010 (二进制真值)
用八位二进制原码表示：**1** 1011010

Q1: Positive and negative representations?
Set symbol bit in front of all numbers



数值数据的定点表示

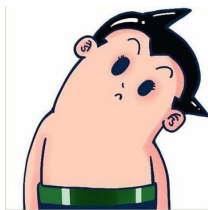


问题二：如何表示小数点？

定点数

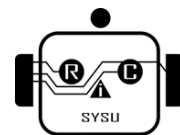
解决方法：小数点的位置固定

- 小数点左边的二进制数是整数，右边的是小数
- 计算机中通常将数分成定点整数和定点小数
- 小数点位置由定点数类型默认约定(即在程序员心里)



But 怎样表示定点数？

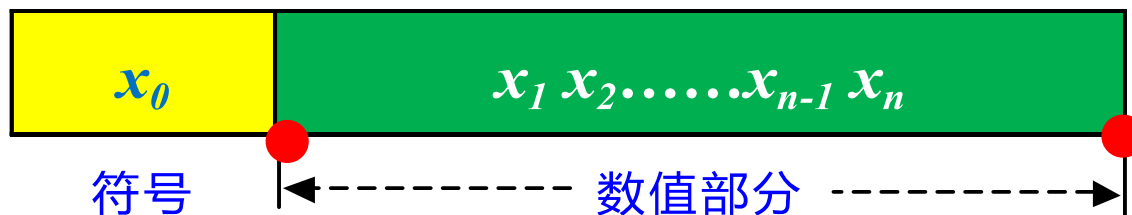
Q2: Representation of decimal point ?



数值数据的定点表示

定点数的格式

定点数 $x = x_0 x_1 x_2 \dots x_n$ 在计算机中表示

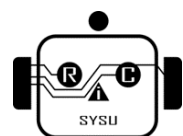


➤ **定点整数**：小数点固定在最低位的右边

表示范围为： $0 \leq |x| \leq 2^n - 1$

➤ **定点小数**：小数点固定在数值部分的最高位的左边
(在 2^0 与 2^{-1} 之间)

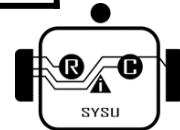
表示范围为： $0 \leq |x| \leq 1 - 2^{-n}$



数值数据的定点表示

C语言整数数据类型

C 声明	MIPS 机器	
	最小值	最大值
char	-128	127
unsigned char	0	255
short int	-32768	32767
unsigned short int	0	65535
int	-2^{31}	$2^{31} - 1$
unsigned int	0	$2^{32} - 1$
long int	-2^{31}	$2^{31} - 1$
unsigned long int	0	$2^{32} - 1$



数值数据的定点表示



问题三：如何表示零？

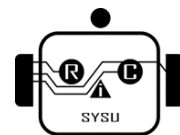
一个数不是正数，就是负数(由符号位决定)，**零**既不是正数又不是负数，**其符号位怎么办？**



就分正零和负零吧！！！！



Q3: Representation of zero ?



数值数据的定点表示



问题三：如何表示零？

一个数不是正数，就是负数(由符号位决定)，**零**
既不是正数又不是负数，**其符号位怎么办？**

$X = +0$ (十进制真值)

用八位二进制表示

原码 = 0 0000000

反码 = 0 0000000

补码 = 0 0000000

$X = -0$ (十进制真值)

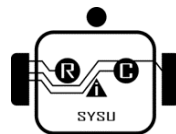
用八位二进制表示

原码 = 1 0000000

反码 = 1 1111111

补码 = 0 0000000

0的补码
表示是统
一的



Signed Integer Representation

有符号整数的表示

Sign & magnitude (8-bit example):

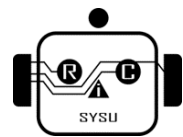
sign	7-bit magnitude (0 ... 127)
------	-----------------------------

Rules for addition, $a + b$:

- If $(a > 0 \text{ and } b > 0)$: add, sign 0
- If $(a > 0 \text{ and } b < 0)$: subtract, sign ...
- ... (有符号数加法怎样确定正负, 有4种情况)
- $+0, -0 \rightarrow$ are they equal? comparator must handle special case! 真值0怎么办: 正零, 负零

Cumbersome (这种表示方法麻烦)

- "Complicated" hardware: reduced speed / increased power
- *Is there a better way? (有没有更好的表示方法)*



补码表示法 Signed-2's complement Representation

1. 模和同余 $n = K * q + r$ (余数)

一个计量器的容量或一个计量单位叫做模数 (Module) , 简称模, 记作 M 或 mod 。

一个 n 位的二进制计数器的模是 2^n 。

对于两个整数 a 、 b , 若用某一正数 M 去除, 所得的余数相同, 则称 a 、 b 对模 M 是同余的。

当 a 、 b 对模 M 同余时, 就称 a 、 b 在以 M 为模时是相等的, 记作 $a = b \pmod{M}$ 。

例如, $a = 14$, $b = 24$, $M = 10$, 则用 10 去除 a 和 b , 余数都是 4, 所以 14 和 24 在以 10 为模时是同余的, 记作 $14 = 24 \pmod{10}$ 。

可推出, $a + M = a \pmod{M}$ $a + 2M = a \pmod{M}$

当 $a = 0$ 时, 有 $0 = M \pmod{M}$ 。 当 a 为负数时, 例如 $a = -2$

$$-2 + 10 = -2 \pmod{10}$$

$$-2 = 0 * 10 + (-2)$$

$$8 = -2 \pmod{10}$$

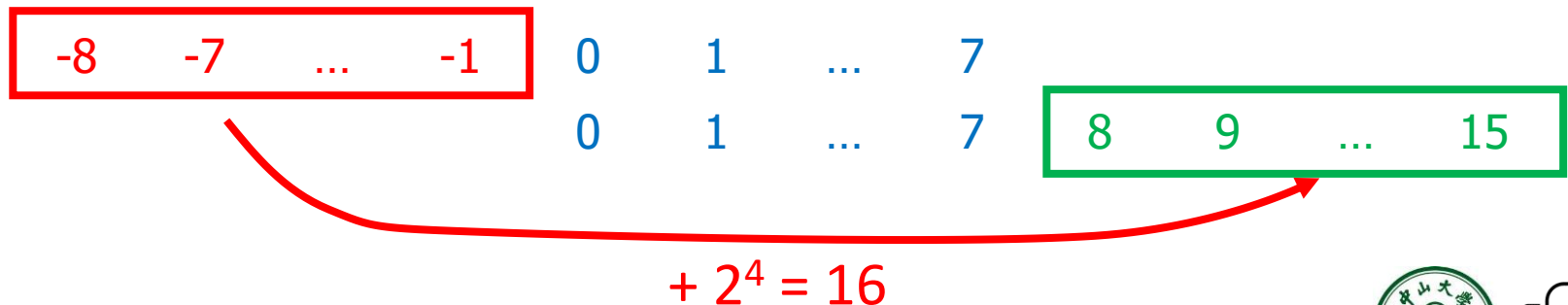
$$8 = 1 * 10 + (-2)$$

4-bit Example (用4位数的计算举例说明为什么使用补码表示)

Decimal				Binary			
7	7	7		0111			
+ -3	+ -3 + 16	+ 13		+ 1101			
4	4 + 16	4 + 16		1 0100	0100 + 1 0000		

- Map negative \rightarrow positive numbers

- Example for N=4-bit: $-3 \rightarrow 2^4 - 3 = 13$
- “Two’s complement”
- No special rules for adding positive and negative numbers



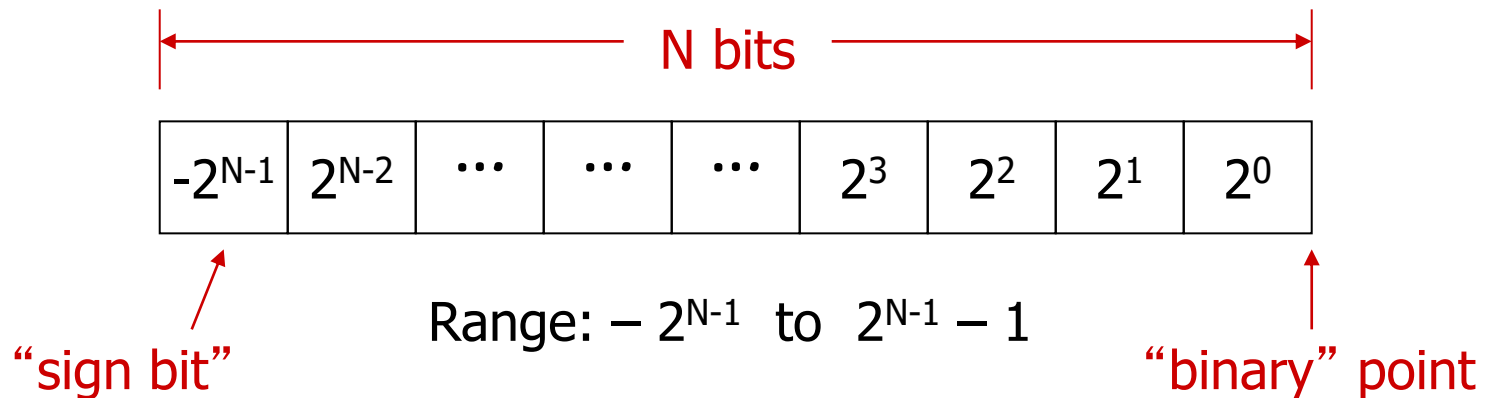
Two's Complement

(8-bit example)

Signed Decimal	Unsigned Decimal	Binary Two's Complement
-128	128	1000 0000
-127	129	1000 0001
...
-2	254	1111 1110
-1	255	1111 1111
0	0	0000 0000
1	1	0000 0001
...
127	127	0111 1111

Diagram illustrating the mapping of signed decimal values to unsigned decimal values and their binary two's complement representations. The table shows values from -128 to 127. A red bracket on the left indicates the range from -128 to -1, with a red arrow labeled '+256' pointing to the corresponding unsigned decimal values (128 to 255). A green bracket on the left indicates the range from 0 to 127, with a green arrow labeled '+0' pointing to the corresponding unsigned decimal values (0 to 127). A blue box highlights the first column of the binary two's complement values, showing the MSB (Most Significant Bit) for each value.

Note: Most significant bit (MSB) equals sign



简单地修正了无符号数的表示, 只是把符号位当做负数。

$$v = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$

符号数

8-bit 2's complement example:

$$\begin{aligned}
 11010110 &= -2^7 + 2^6 + 2^4 + 2^2 + 2^1 \\
 &= -128 + 64 + 16 + 4 + 2 = -42
 \end{aligned}$$

Why 2's Complement?

□ Benefit: 模 2^n 的运算，用补码表示减法变成了加负数

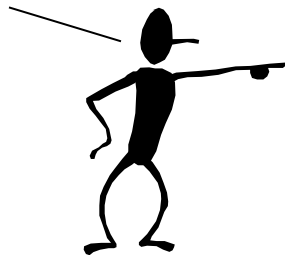
□ 只有加法没有减法运算，不用管符号位！

□ 处理无符号数同样适用！

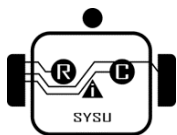
Example:

When using signed magnitude representations, adding a negative value really means to subtract a positive value. However, in 2's complement, adding is adding regardless of sign. In fact, you NEVER need to subtract when you use a 2's complement representation.

$$\begin{array}{rcl} 55_{10} & = & 00110111_2 \\ + 10_{10} & = & 00001010_2 \\ \hline 65_{10} & = & 01000001_2 \end{array}$$



$$\begin{array}{rcl} 55_{10} & = & 00110111_2 \\ + -10_{10} & = & 11110110_2 \\ \hline 45_{10} & = & 100101101_2 \end{array}$$



2's Complement

□ How to negate a number (怎样求补) ?

□ First complement every bit (i.e. $1 \rightarrow 0$, $0 \rightarrow 1$), then add 1 (按位求反加1)

□ Example: $20 = 00010100$, $-20 = 11101011 + 1 = 11101100$

□ Sign-Extension (符号扩展)

□ suppose you have an 8-bit number that needs to be “extended” to 16 bits

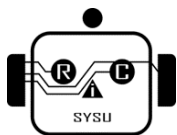
□ Why? Maybe because we are adding it to a 16-bit number...

■ Homework: Prove above two statement.

□ Examples

□ 16-bit version of 42 = 0000 0000 0010 1010

□ 8-bit version of -2 = 1111 1111 1111 1110



基本转换 (+ve ints)

□ Binary to Decimal (二进制转换10进制)

□ multiply each bit by its positional power of 2

□ add them together

$$v = \sum_{i=0}^{n-1} 2^i b_i$$

□ Decimal to Binary (10进制转换二进制)

□ Problem: given v , find b_i (inverse problem)

□ Hint: expand series

$$v = b_0 + 2b_1 + 4b_2 + 8b_3 + \dots + 2^{n-1}b_{n-1}$$

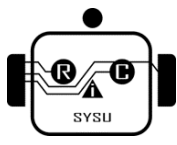
□ observe: every term is even except first

□ this determines b_0

□ divide both sides by 2

$$v \operatorname{div} 2 = b_1 + 2b_2 + 4b_3 + \dots + 2^{n-2}b_{n-1} \quad (\text{quotient})$$

$$v \operatorname{mod} 2 = b_0 \quad (\text{remainder})$$



Tutorial on Base Conversion (+ve ints)

□ Decimal to Binary

□ Problem: given v , find b_i (inverse problem)

$$v = b_0 + 2b_1 + 4b_2 + 8b_3 + \dots + 2^{n-1}b_{n-1}$$

□ Algorithm (除2求余) :

□ Repeat

□ divide v by 2

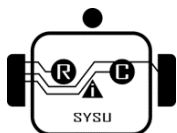
□ remainder becomes the next bit, b_i

□ quotient becomes the next v

□ Until v equals 0

□ Note: Same algorithm applies to other number bases

□ just replace divide-by-2 by divide-by- n for base n



Addition

4-bit Example

Unsigned

$$\begin{array}{r} 3_{\text{ten}} \\ + 4_{\text{ten}} \\ \hline 7_{\text{ten}} \end{array} \quad \begin{array}{r} 0011_{\text{two}} \\ + 0100_{\text{two}} \\ \hline 0111_{\text{two}} \end{array}$$

$$\begin{array}{r} 3_{\text{ten}} \\ + 11_{\text{ten}} \\ \hline 14_{\text{ten}} \end{array} \quad \begin{array}{r} 0011_{\text{two}} \\ + 1011_{\text{two}} \\ \hline 1110_{\text{two}} \end{array}$$

Signed (Two's Complement)

$$\begin{array}{r} 3_{\text{ten}} \\ + 4_{\text{ten}} \\ \hline 7_{\text{ten}} \end{array} \quad \begin{array}{r} 0011_{\text{two}} \\ + 0100_{\text{two}} \\ \hline 0111_{\text{two}} \end{array}$$

$$\begin{array}{r} 3_{\text{ten}} \\ + -5_{\text{ten}} \\ \hline -2_{\text{ten}} \end{array} \quad \begin{array}{r} 0011_{\text{two}} \\ + 1011_{\text{two}} \\ \hline 1110_{\text{two}} \end{array}$$

No special rules for two's complement signed addition

Overflow

4-bit Example

Unsigned

$$\begin{array}{r}
 13_{\text{ten}} \quad 1101_{\text{two}} \\
 + 14_{\text{ten}} \quad + 1110_{\text{two}} \\
 \hline
 27_{\text{ten}} \quad \textcircled{1} 1011_{\text{two}}
 \end{array}$$

carry-out and overflow

$$\begin{array}{r}
 7_{\text{ten}} \quad 0111_{\text{two}} \\
 + 1_{\text{ten}} \quad + 0001_{\text{two}} \\
 \hline
 8_{\text{ten}} \quad \textcircled{0} 1000_{\text{two}}
 \end{array}$$

no carry-out and no overflow

Carry-out \rightarrow Overflow

Signed (Two's Complement)

$$\begin{array}{r}
 3_{\text{ten}} \quad 1101_{\text{two}} \\
 + -2_{\text{ten}} \quad + 1110_{\text{two}} \\
 \hline
 -5_{\text{ten}} \quad \textcircled{1} 1011_{\text{two}}
 \end{array}$$

carry-out but no overflow

$$\begin{array}{r}
 7_{\text{ten}} \quad 0111_{\text{two}} \\
 + 1_{\text{ten}} \quad + 0001_{\text{two}} \\
 \hline
 -8_{\text{ten}} \quad \textcircled{0} 1000_{\text{two}}
 \end{array}$$

no carry-out but overflow

Carry-out \rightarrow ~~Overflow~~

Overflow Detection

4-bit Example

Unsigned

- Carry-out indicates overflow

Signed (Two's Complement)

- Overflow if
 - Signs of operands are equal
- AND*
- Sign of result differs from sign of operands
- No overflow when signs of operands differ

Overflow rules depend on operands (signed vs unsigned)

实数的表示



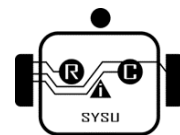
问题四：如何表示实数？

如何表示？

将实数分成两部分：尾数和指数(阶码)——浮点数的表示

$$\begin{aligned} 25.75 &= 0.2575 \times 10^2 \text{ (十进制)} \\ &= 11001.11 \text{ (二进制真值)} \\ &= 1.100111 \times 10^{100} \text{ (二进制)} \end{aligned}$$

Q4: Representation of real number ?



Non-Integral Numbers

□ How about non-integers (非整数怎么表示) ?

□ examples

□ 1.234

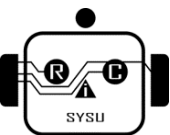
□ -567.34

□ 0.00001

□ 0.00000000000000012

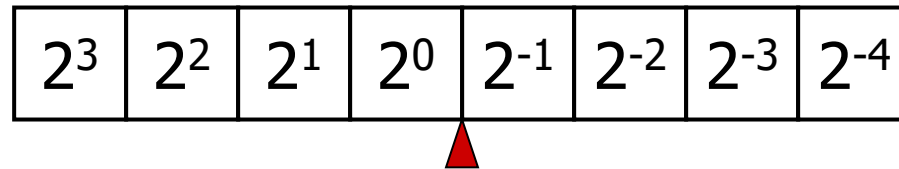
□ fixed-point representation (定点表示)

□ floating-point representation (浮点表示)



Fixed-Point Representation (定点表示)

- Set a definite position for the “binary” point
 - everything to its left is the integral part of the number
 - everything to its right is the fractional part of the number

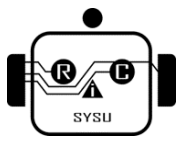


- (定点左边表示整数部分，右边表示小数部分)

$$\begin{aligned} 1101.0110 &= 2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} \\ &= 8 + 4 + 1 + 0.25 + 0.125 \\ &= 13.375 \end{aligned}$$

Or

$$1101.0110 = 214 * 2^{-4} = 214/16 = 13.375$$



Fixed-Point Base Conversion

□ Binary to Decimal

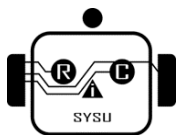
- multiply each bit by its positional power of 2
 - just that the powers of 2 are now negative
 - for m fractional bits
- $$v = \sum_{i=-1}^{-m} 2^i b_i$$

$$v = \frac{b_{-1}}{2} + \frac{b_{-2}}{4} + \frac{b_{-3}}{8} + \frac{b_{-4}}{16} + \dots + \frac{b_{-m}}{2^m}$$

$$v = 2^{-1}b_{-1} + 2^{-2}b_{-2} + 2^{-3}b_{-3} + 2^{-4}b_{-4} + \dots + 2^{-m}b_{-m}$$

□ Examples

- $0.1_2 = \frac{1}{2} = 0.5_{10}$
- $0.0011_2 = 1/8 + 1/16 = 0.1875_{10}$
- $0.001100110011_2 = 1/8 + 1/16 + 1/128 + 1/256 + 1/2048 + 1/4096 = 0.19995117187_{10}$ (getting close to 0.2)
- 0.0011_2 (repeats) $= 0.2_{10}$



Signed fixed-point numbers (有符号数的定点表示)

□ How do you incorporate a sign (怎样加入符号) ?

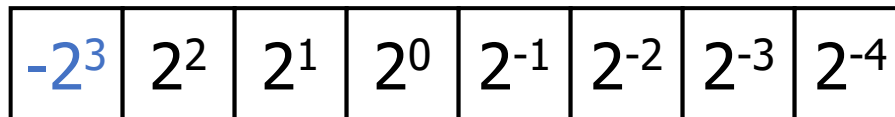
□ use sign magnitude representation

□ an extra bit (leftmost) stores the sign

□ just as in negative integers

□ 2' s complement

□ leftmost bit has a negative coefficient

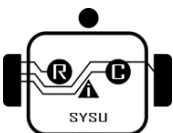


$$\begin{aligned} 1101.0110 &= -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} \\ &= -8 + 4 + 1 + 0.25 + 0.125 = -2.625 \end{aligned}$$

□ OR:

□ first ignore the binary point, use 2' s complement, put the point back

$$\begin{aligned} 1101.0110 &= (-128 + 64 + 16 + 4 + 2) * 2^{-4} \\ &= -42/16 = -2.625 \end{aligned}$$



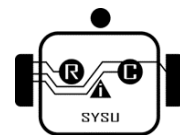
数值数据的浮点表示

科学计数法(Scientific Notation)

mantissa (尾数) \rightarrow 6.02 \times 10²¹ \leftarrow *exponent* (阶码, 指数)

\nwarrow \nearrow

decimal point (十进制小数点) *radix* (base, 基数)



2.4.3 数值数据的浮点表示

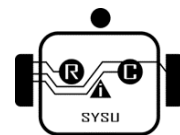
科学计数法(Scientific Notation)



for Binary Numbers(二进制数):

$$\begin{array}{ccc} \text{mantissa(尾数)} & & \text{exponent(阶码)} \\ \swarrow & & \swarrow \\ & 1.011_{\text{two}} \times 2^{-10} & \\ \nearrow & \nwarrow & \\ \text{binary point (二进制小数点)} & & \text{基为2} \end{array}$$

只要对尾数和指数分别编码，就可表示一个浮点数



数值数据的浮点表示

科学计数法(Scientific Notation)

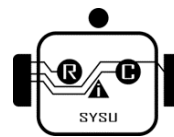
mantissa (尾数) \rightarrow 6.02 \times 10²¹ \leftarrow *exponent* (阶码, 指数)
 \nwarrow *decimal point* (十进制小数点) \nearrow *radix* (base, 基)

■ 同一个数有多种表示形式。例：对于数 1/1,000,000,000

➤ 1.0×10^{-9} , 0.1×10^{-8} , 10.0×10^{-10}

■ **Normalized form(规格化形式):** 小数点前只有一位非0数

➤ Normalized (唯一的规格化形式): 1.0×10^{-9}



数值数据的浮点表示

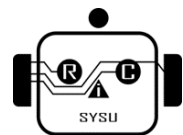
浮点数据表示的进一步改进

在浮点数总位数不变的情况下，为**提高**数据表示**精度**，使尾数的有效数字尽可能占满已有的位数

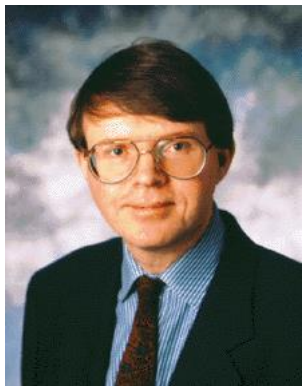


浮点数的规格化(Normalize): $\frac{1}{2} \leq |f| < 1$

- 右规: $|f| > 1$
右移1位，阶码加1



数值数据的浮点表示——IEEE 754



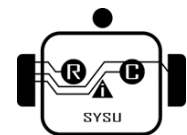
1985年制定了浮点数标准IEEE 754

符号 s (Sign)	阶码 e (整数)Exponent	尾数 f (小数)Significand
----------------	----------------------	-------------------------

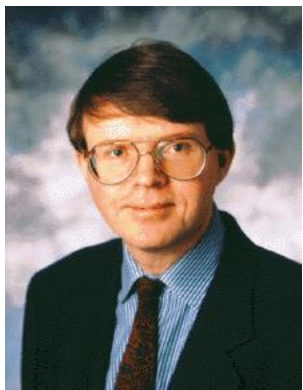
Prof. William Kahan ➤

符号s:

1 表示负数negative ; 0表示 正数positive



数值数据的浮点表示——IEEE 754



William Kahan

1985年制定了浮点数标准IEEE 754

符号 s (Sign)	阶码 e (整数) Exponent	尾数 f (小数) Significand
----------------	-----------------------	--------------------------

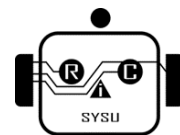
➤ 尾数 f

- 尾数为原码
- 规格化尾数最高位总是1，所以隐含表示，省1位
- 1 + 23 bits (single单精度), 1 + 52 bits (double双精度)

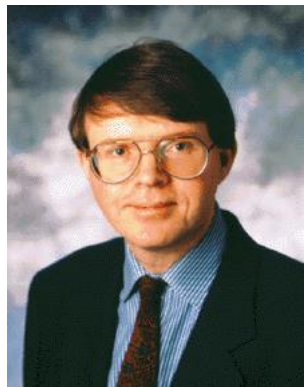
$$\begin{aligned} \text{尾数} &= 1 + \text{significand} \\ 0 &< \text{significand} < 1 \end{aligned}$$

$$\text{尾数精度} = \text{尾数的位数} + 1$$

(单精度SP: Single Precision 双精度DP: Double Precision)



数值数据的浮点表示——IEEE 754

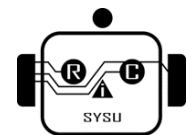


William Kahan


1985年制定了浮点数标准IEEE 754

符号 s (Sign)	阶码 e (整数)Exponent	尾数 f (小数)Significand
------------------	------------------------	---------------------------

➤ 阶码/指数 e : 移码



数值数据的浮点表示——IEEE 754



But 什么是excess (biased) notation——“移码表示”？

□ 将每一个数值加上一个偏置常数(Excess / bias)

例：

$$-8 (+8) \sim 0000_2$$

$$0 (+8) \sim 1000_2$$

...

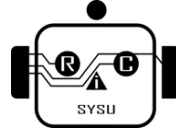
...

$$-7 (+8) \sim 0001_2$$

$$+7 (+8) \sim 1111_2$$

为什么要用移码来表示指数(阶码)？

便于浮点数加减运算时
进行对阶操作(比较大小)



数值数据的浮点表示——IEEE 754



But 什么是excess (biased) notation——“移码表示”？

□ 将每一个数值加上一个偏置常数(Excess / bias)

例： $-8 (+8) \sim 0000_2$ $0 (+8) \sim 1000_2$

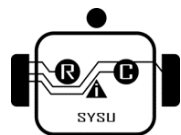
...

...

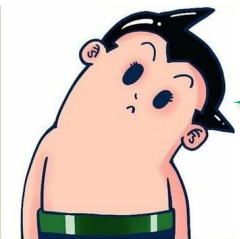
$-7 (+8) \sim 0001_2$ $+7 (+8) \sim 1111_2$

为什么要用移码来表示指数(阶码)?

例： $1.01 \times 2^{-2} + 1.11 \times 2^3$



数值数据的浮点表示——IEEE 754



But 什么是excess (biased) notation——“移码表示”？

□ 将每一个数值加上一个偏置常数(Excess / bias)

例： $-8 (+8) \sim 0000_2$ $0 (+8) \sim 1000_2$

...

...

$-7 (+8) \sim 0001_2$ $+7 (+8) \sim 1111_2$

为什么要用移码来表示指数(阶码)?

例： $1.01 \times 2^{-2} + 1.11 \times 2^3$

$1.01 \times 2^{-2+8} + 1.11 \times 2^{3+8}$

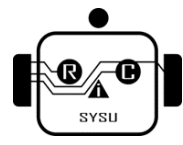
补码： $1110 < 0011$?

移码： $0110 < 1011$

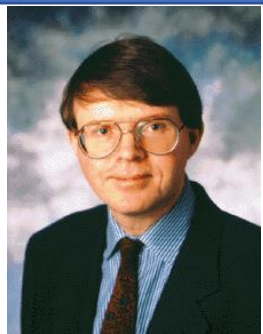
(-2) (3)

(6) (11)

简化比较



数值数据的浮点表示——IEEE 754



Prof. Kahan

符号 s (Sign)	阶码 e (整数)Exponent	尾数 f (小数)Significand
------------------	------------------------	---------------------------

➤ 阶码/指数 e : 移码

全0/全1编码用来表示特殊的值!

将每一个数值加上一个偏置常数, 当编码位数为 n 时, 通常 bias 取 $2^{n-1}-1$

- 偏置常数为:
127 (单精度SP); 1023 (双精度DP)
- 单精度规格化数阶码范围为
0000 0001 (-126) ~ 11111110 (127)

单精度SP: Single Precision 双精度DP: Double Precision

数值数据的浮点表示——IEEE 754

- 偏置常数为:

127 (单精度SP); 1023 (双精度DP)

➤ 单精度浮点的偏置为什么是127

可用的移码范围 \longrightarrow 0000 0001 ~ 1111 1110

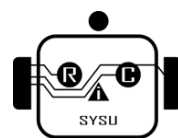
\downarrow \downarrow

1 254

我们希望的阶码范围 \longrightarrow -126 ~ +127 \longrightarrow 偏置127

如果我们希望的阶码范围是: -127 ~ +126 \longrightarrow 偏置128

我们貌似对宏观世界更感兴趣, 那就取127吧



数值数据的浮点表示——IEEE 754



Prof. Kahan

符号 s (Sign)	阶码 e (整数)Exponent	尾数 f (小数)Significand
----------------	----------------------	-------------------------

➤ 尾数 f

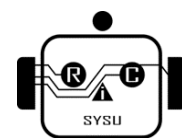
- 规格化尾数最高位总是1，所以隐含表示，省1位
- 1 + 23 bits (single单精度), 1 + 52 bits (double双精度)

➤ 阶码/指数 e: 移码

- 偏置常数为: 127 (单精度); 1023 (双精度)

SP(单精度)浮点数: $(-1)^S \times (1 + f) \times 2^{(\text{Exponent}-127)}$

DP(双精度)浮点数: $(-1)^S \times (1 + f) \times 2^{(\text{Exponent}-1023)}$



数值数据的浮点表示——IEEE 754



Prof. Kahan

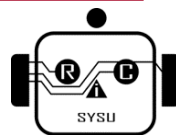
符号 s (Sign)	阶码 e (整数)Exponent	尾数 f (小数)Significand
------------------	------------------------	---------------------------

➤ 尾数 f

- 规格化尾数最高位总是1，所以隐含表示，省1位
- 1 + 23 bits (single单精度), 1 + 52 bits (double双精度)



- 浮点数的精度由尾数 f 的位数决定
- 浮点数的表示范围由基数 R 和阶码 e 的位数决定
- 阶码的位数和基数越大，表示的浮点数范围越大



数值数据的浮点表示——IEEE 754

练习1： 将二进制浮点表示转换成十进制数

例1: BEE00000H is the hex. Rep. Of an IEEE 754 SP FP number

1	011 1110	1	110 0000 0000 0000 0000 0000
---	----------	---	------------------------------

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

Sign: 1 => negative

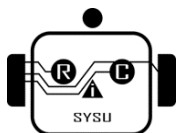
Exponent:

- $0111\ 1101_{\text{two}} = 125_{10}$
- Bias adjustment: $125 - 127 = -2$

Significand:

$$\begin{aligned} & 1 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5} + \dots \\ & = 1 + 2^{-1} + 2^{-2} = 1 + 0.5 + 0.25 = 1.75 \end{aligned}$$

Represents: $-1.75_{\text{ten}} \times 2^{-2} = -0.4375$



数值数据的浮点表示——IEEE754

练习2： 将十进制数转换成单精度浮点表示： -1.275×10^1

1. 计算真值Denormalize : -12.75

2. 整数部分的转换：

$$12 = 8 + 4 = 1100_2$$

3. 小数部分的转换：

$$0.75 = 0.5 + 0.25 = 0.11_2$$

4. 规格化：

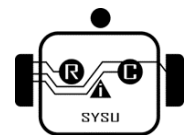
$$1100.11 = 1.10011 \times 2^{011}$$

5. 移码表示的阶码：

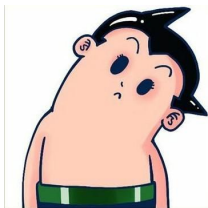
$$127 + 3 = 10000010_2$$

1	100 0001 0	100 1100 0000 0000 0000 0000
---	------------	------------------------------

十六进制表示： **C14C0000H**

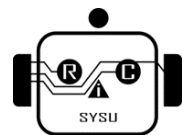


数值数据的浮点表示——IEEE 754



全0和全1编码用来表示什么特殊的值？

阶码(移码)	尾数	数据类型
1~254	任何值	规格化数（隐含小数点前为“1”）
0	0	?
0	非零的数	?
255	0	?
255	非零的数	?



数值数据的浮点表示——IEEE 754

□ 如何表示0?

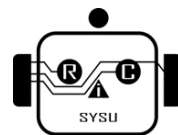
■ 阶码/指数: 全0 (Exponent: all zeros)

■ 尾数: 全0 (Significand: all zeros)

■ 符号位? 正/负皆可 (What about sign? Both cases valid.)

+0: 0 00000000 000000000000000000000000

-0: 1 00000000 000000000000000000000000



数值数据的浮点表示——IEEE 754

□ 如何表示0?

■ 阶码/指数: 全0

■ 尾数: 全0

■ 符号位? 正/负皆可

+0: 0 00000000 000000000000000000000000

-0: 1 00000000 000000000000000000000000

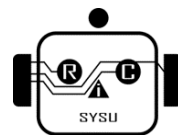
□ 如何表示 $+\infty/-\infty$?

■ 阶码/指数: 全1 ($11111111_2 = 255$)

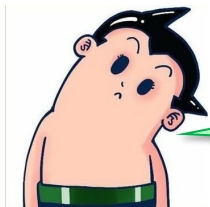
■ 尾数: 全0 (Significand: all zeros)

$+\infty$: 0 11111111 000000000000000000000000

$-\infty$: 1 11111111 000000000000000000000000



数值数据的浮点表示——IEEE 754



全0和全1编码用来表示什么特殊的值？

阶码(移码)	尾数	数据类型
1~254	任何值	规格化数（隐含小数点前为“1”）
0	0	0
0	非零的数	非规格化数 Denormalized numbers
255	0	$+\infty/-\infty$
255	非零的数	非数 NaN (Not a Number)

非数，示例：

$\text{sqrt}(-4.0) = \text{NaN}$

$+\infty + (-\infty) = \text{NaN}$

$0/0 = \text{NaN}$

$+\infty - (+\infty) = \text{NaN}$

数值数据的浮点表示——IEEE 754

➤ 为什么需要非规格化的数

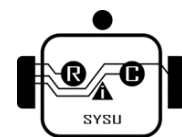
最小的规格化数是 1.0×2^{-126} ，我们需要更小的数怎么办？

非规格化数可表示 $0.00\dots01 \times 2^{-126}$ 至 $0.11\dots11 \times 2^{-126}$ 之间的数

描述	指数	小数	单精度		双精度	
			值	十进制	值	十进制
最小非格式化数	00...00	0...01	$2^{-23} \times 2^{-126}$	1.4×10^{-45}	$2^{-52} \times 2^{-1022}$	4.9×10^{-324}
最大非格式化数	00...00	1...11	$(1-\epsilon) \times 2^{-126}$	1.2×10^{-38}	$(1-\epsilon) \times 2^{-1022}$	2.2×10^{-308}
最小格式化数	00...01	0...00	1×2^{-126}	1.2×10^{-38}	(1×2^{-1022})	2.2×10^{-308}
最大格式化数	11..10	1...11	$(2-\epsilon) \times 2^{127}$	3.4×10^{38}	$(2-\epsilon) \times 2^{1023}$	1.8×10^{308}

➤ 为什么需要NaN

一些非法计算的结果，需要用特殊的形式标记



□ 表示规格化非0数 (Norms)

□ **Exponent**: nonzero

□ **Significand**: nonzero (大于等于1且小于2)

□ 表示非规格化数 (Denorms)

□ **Exponent**: all zeros

□ **Significand**: nonzero (大于0且小于1)

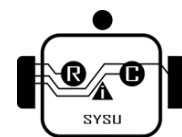
□ 表示非数 (NaN)

□ **Exponent** : 255

□ **Significand**: nonzero

□ **NaNs can help with debugging**

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	\pm denormalized number
1-254	Anything	1-2046	Anything	\pm floating-point number
255	0	2047	0	\pm infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)



关于IEEE 754的疑问

单精度可表示值的范围是多少？

The largest number for single: $+1.11\cdots1 \times 2^{127}$ 约 3.4×10^{38}

The least number for single: $+1.0\cdots0 \times 2^{-126}$

How about double? $+1.11\cdots1 \times 2^{1023}$

约 $+1.8 \times 10^{308}$

强制类型转换会如何？有效位数可能丢失

`i = int (float) i` How about double? True!

`f = float (int) f` How about double? Not always true!

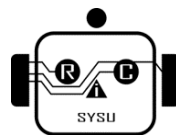
浮点加法满足结合律吗？False 小数部分可能丢失

$$x = -1.5 \times 10^{38}, \quad y = 1.5 \times 10^{38}, \quad z = 1.0$$

$$(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 = 1.0$$

$$x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) = 0.0$$

结论: $(x + y) + z \neq x + (y + z)$



将十进制数转换成浮点格式 (real*4)

[例3]: 26.0

[例4]: 0.75

[例5]: -2.5

将十进制数转换成浮点格式 (real*4)

[例3]: 26.0十进制26.0转换成二进制11010.0

规格化二进制数 1.10100×2^4

计算指数 $4+127=131$

符号位 指数部分 尾数部分

0 10000011 101000000000000000000000

以单精度 (real*4) 浮点格式存储该数

0100 0001 1101 0000 0000 0000 0000 0000 0x41D0 0000

[例4]: 0.75, 十进制0.75转换成二进制0.11

规格化二进制数 1.1×2^{-1} 计算指数 $-1+127=126$

符号位 指数部分 尾数部分

0 01111110 100000000000000000000000

以单精度 (real*4) 浮点格式存储该数

0011 1111 0100 0000 0000 0000 0000 0000 0x3F40 0000

[例5]: -2.5 十进制-2.5转换成二进制-10.1

规格化二进制数 -1.01×2^1

计算指数 $1+127=128$

符号位 指数部分 尾数部分

1 10000000 010000000000000000000000

以单精度 (real*4) 浮点格式存储该数

1100 0000 0010 0000 0000 0000 0000 0000

0xC020 0000

Exe1

执行下列C语言:

```
unsigned short x=65530;
```

```
unsigned int y=x;
```

得到y的机器码多少？

Exe1

执行下列C语言:

unsigned short x=65530;

unsigned int y=x;

得到y的机器码多少?

•65530二进制多少?

65535是0xFFFF, 65530是0xFFFF-5= 0xFFFA

•无符号扩展: 0x0000FFFA

Exe2

16位补码0x8FA0扩展为32位是多少？

Exe3

变量X,Y,Z, 其中X和Z为int型, Y为short型, 当X=127, Y=-9;
试问: 执行语句Z=X+Y之后, X,Y,Z分别是什么?

Exe3

变量X,Y,Z, 其中X和Z为int型, Y为short型, 当X=127, Y=-9;
试问: 执行语句Z=X+Y之后, X,Y,Z分别是什么?

$X = 0x0000007F$

$Y = \sim(0009) + 1 = FFF6 + 1 = FFF7$

$X + Y = (\text{扩展}) 0x0000007F + FFFFFFF7 = 1\ 0000\ 0000\ 0076$

符号不同, 没有溢出

最高位舍掉: 0000 0000 0076 (其实就是 $118 = 127 - 9$)

Exe4

单精度浮点数机器码表述为0x4510 0000, 表示的浮点数是
？

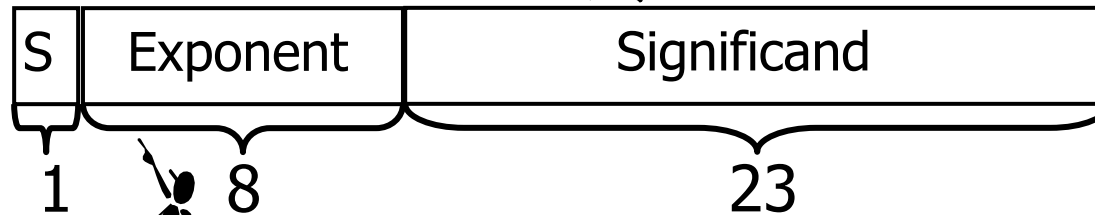
IEEE 754 Format

IEEE 754 Floating-Point Formats

Single-precision format

This is effectively a signed magnitude fixed-point number with a “hidden” 1.

The 1 is hidden because it provides no information after the number is “normalized”



The exponent is represented in bias 127 notation. Why?

$$v = (-1)^S \times 1.\textit{Significand} \times 2^{\textit{Exponent}-127}$$

42.75 = 00101010.11000000₂

Normalize: 001.010101100000₂ × 2⁵

(127+5)

0 10000100 010101100000 000000000000₂

0100 0010 0010 1011 0000 0000 0000 0000₂

42.75 = 0x422B0000₁₆

Special values in the IEEE 754 standard.

Sign	8-Bit Based Exponent	23-Bit Normalized Fraction
[31]	[30:23]	[22:0]

$E_{bias}=2^{(8-1)}-1=127$ ，偏移量取127
指数E的取值范围是 $(-2^{(8-1)}+1)-2^{(8-1)}$, 即-127 -- 128
做为常规值， E的取值范围是-126 -- 127
最大值和最小值为

MinNorm	MaxNorm
$(-1)^s \times (2)^{-126} \times 1.0$	$(-1)^s \times (2)^{127} \times (2-2^{-23})$
S 00000001 000000000000000000000000	S 11111110 111111111111111111111111

Special Value	Exponent	Significand
+/- 0	0000 0000	0
Denormalized number	0000 0000	Nonzero
NaN	1111 1111	Nonzero
+/- infinity	1111 1111	0

Exe4

单精度浮点数机器码表述为0x4510 0000, 表示的浮点数是？

0x4510 0000

->0100 0101 0001 0000 0000 0000 0000 0000

尾数: 001 0000 0000 0000 0000 0000

阶数: 1000 1010

符号: 0

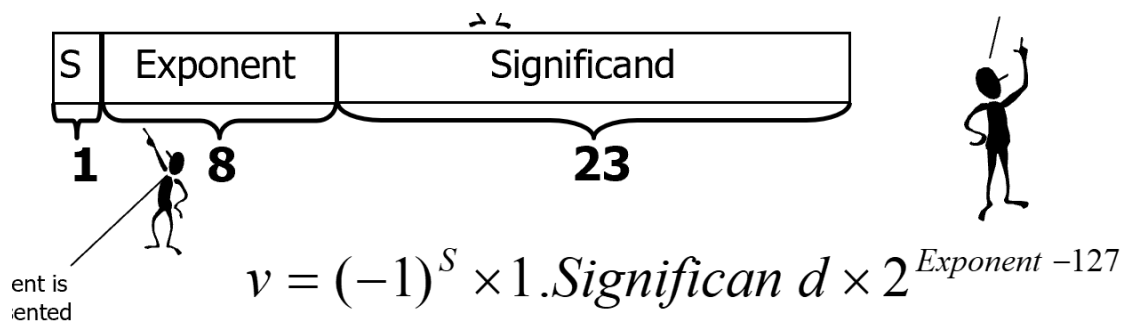
$$v = (-1)^s \times 1.\textit{Significand} \times 2^{\textit{Exponent}-127}$$

带入: 1.125×2^{11}

Exe5

已知: 两个实数 $x=-68$, $y=-8.25$, 他们在C语言中定义为float型变量, 分别存放在寄存器A和B中, 还有另外两个寄存器C和D中。

- (1) 寄存器A和B的机器码分别是?
- (2) $C=x+y$, 寄存器的机器码是?
- (2) $D=x-y$, 寄存器的机器码是?

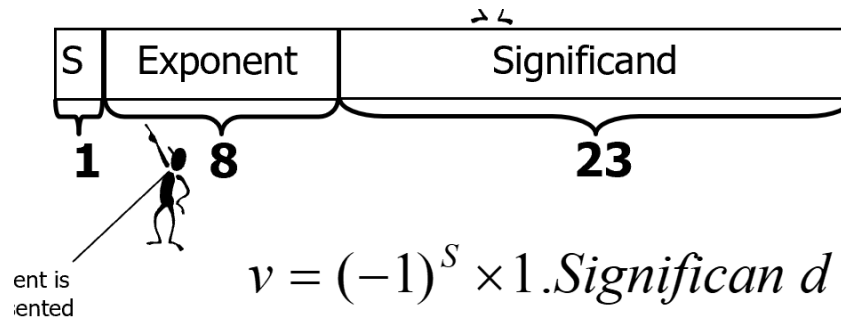


Exe5

已知: 两个实数 $x=-68$, $y=-8.25$, 他们在C语言中定义为float型变量, 分别存放在寄存器A和B中, 还有另外两个寄存器C和D中。

- (1) 寄存器A和B的机器码分别是?
- (2) $C=x+y$, 寄存器的机器码是?
- (2) $D=x-y$, 寄存器的机器码是?

$$X=-68=-1.0001 \times 2^6$$



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



$$S=1$$

$$\text{Exponent}=6+127=1000\ 0101$$

$$\text{Significand}=0001\ 0000\ 0000\ 0000\ 0000\ 0000$$

$$1\ 1000\ 0101\ 0001\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 = \text{C288}\ 0000$$

Exe5

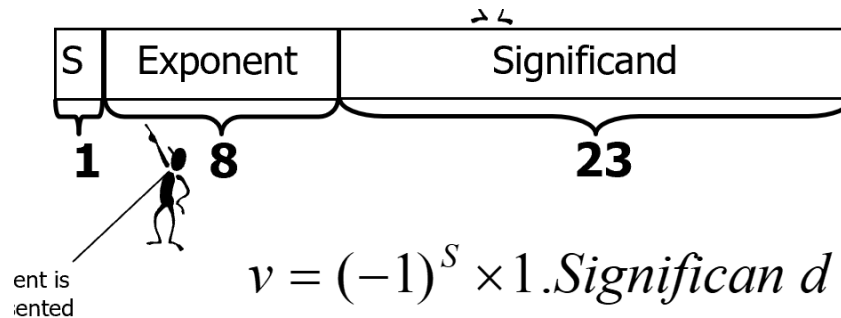
已知: 两个实数 $x=-68$, $y=-8.25$, 他们在C语言中定义为float型变量, 分别存放在寄存器A和B中, 还有另外两个寄存器C和D中。

(1) 寄存器A和B的机器码分别是?

(2) $C=x+y$, 寄存器的机器码是?

(2) $D=x-y$, 寄存器的机器码是?

$$y = -8.25 = -1000.01 \\ = -1.00001 \times 2^3$$



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

$$S=1$$

$$\text{Exponent} = 3 + 127 = 1000\ 0010$$

$$\text{Significand} = 0000\ 1000\ 0000\ 0000\ 0000\ 000$$

$$1\ 1000\ 0010\ 0000\ 1000\ 0000\ 0000\ 0000\ 000 = \text{C104}\ 0000$$

本讲内容

□ 什么是计算机语言？

□ 指令集概述 Introduction to Instruction Set

□ 指令格式

□ 寻址方式

□ 数据表示方法

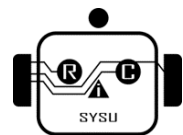
□ 数值数据表示

□ 数据的十进制表示

□ 非数值数据表示

□ 数据的存储

□ 数据的校验



数值数据的十进制表示



计算机中如何表示十进制数值?

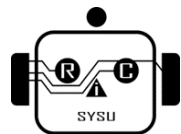
- 人们习惯用十进制数
- 可以减少二进制数和十进制数之间的转换

十进制数的二进制编码表示

➤ ASCII码

➤ BCD码

Q: How do decimal values be represented in computers?

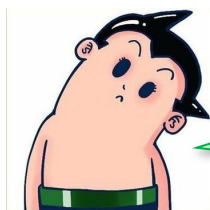


数值数据的十进制表示

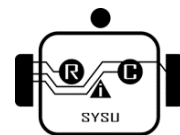
用ASCII码字符表示十进制数

- 把十进制数看成字符串
- 十进制0 ~ 9分别对应30H ~ 39H
- 1 位十进制数对应 8 位二进制数

十进制数	0	1	2	3	4	5	6	7	8	9
ASCII 编码	30H	31H	32H	33H	34H	35H	36H	37H	38H	39H



But 符号位怎样表示?
How to represent symbols?



数值数据的十进制表示

用ASCII码字符表示十进制数



ASCII码格式1 —— 前分隔数字串

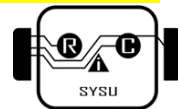
- 符号位单独用一个字节表示，位于数字串之前
- 正号 “+” 用 ASCII码 “2BH” 表示
- 负号 “-” 用 ASCII码 “2DH” 表示

例：十进制数+236表示为： **2B** 32 33 36H

0010 1011 0011 0010 0011 0011 0011 0110B

十进制数-2369表示为： **2D** 32 33 36 39H

0010 1101 0011 0010 0011 0011 0011 0110 0011 1001B



数值数据的十进制表示

用ASCII码表示十进制数

后嵌入数字串

➤ 符号位嵌入到最低位数字的ASCII码的高4位。省一个字节

➤ 正数：最低位数字的高4位：不变

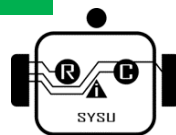
➤ 负数：最低位数字的高4位：变为0111

例：十进制数+236表示为：32 33 36H

0011 0010 0011 0011 0011 0110B

十进制数-2369表示为：32 33 36 79H

0011 0010 0011 0011 0011 0110 0111 1001B



数值数据的十进制表示



计算机中如何表示十进制数值？

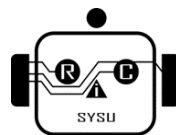
- 人们习惯用十进制数
- 可以减少二进制数和十进制数之间的转换

十进制数的二进制编码表示

➤ ASCII 码

占空间大，且需转
换成二进制数或
BCD码才能计算

Q: How do decimal values be represented in computers?



2.4.4 数值数据的十进制表示



计算机中如何表示十进制数值?

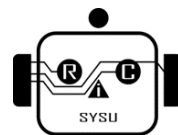
- 人们习惯用十进制数
- 可以减少二进制数和十进制数之间的转换

十进制数的二进制编码表示

➤ ASCII码

➤ BCD码

Q: How do decimal values be represented in computers?



2.4.4 数值数据的十进制表示

用BCD码表示十进制数



编码思想

- 0 ~ 9每个十进制数位至少用4位二进制位来表示
- 4位二进制位可以组合成16种状态，去掉10种状态后还有6种冗余状态



编码方法

➤ 十进制有权码

表示每个十进制数位的4个二进制数位(称基2码)都有一个确定的权，如8421码(从高到低各位二进制位对应的权值分别为8、4、2、1)

➤ 十进制无权码

表示每个十进制数位的4个基2码没有确定的权，如余3码和格雷码

➤ 其他编码方法 (5中取2码、独热码等)

2.4.4 数值数据的十进制表示



编码方法

十进制数	8421BCD码	余3码	格雷码
0	0000	0011	0000
1	0001	0100	0001
2	0010	0101	0011
3	0011	0110	0010
4	0100	0111	0110
5	0101	1000	1110
6	0110	1001	1010
7	0111	1010	1000
8	1000	1011	1100
9	1001	1100	0100

2.4.4 数值数据的十进制表示

用BCD码表示十进制数



编码思想

- 每个十进制数位至少用4位二进制位来表示
- 4位二进制位可以组合成16种状态，去掉前10种状态后还有6种冗余状态

符号位: “+” : 1100 ; “-” : 1101

例: $+236 = (1100\ 0010\ 0011\ 0110)_{8421}$ (占2个字节)

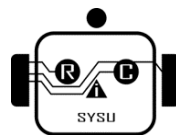
$-2369 = (1101\ 0010\ 0011\ 0110\ 1001)_{8421}$

(占2个半字节? ? ?)

$= (1101\ 0000\ 0010\ 0011\ 0110\ 1001)_{8421}$

(占3个字节)

补0: 使数占满一个字节



本讲内容

□ 什么是计算机语言？

□ 指令集概述 Introduction to Instruction Set

□ 指令格式

□ 寻址方式

□ 数据表示方法

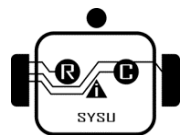
□ 数值数据表示

□ 数据的十进制表示

□ 非数值数据表示

□ 数据的存储

□ 数据的校验



字符数据的机器表示

西文字符的编码表示



特点

- 是一种拼音文字，用有限几个字母可以拼写出所有单词
- 只需对有限个少量字母和一些数学符号、标点符号等辅助字符进行编码
- 所有西文字符集的字符总数不超过256个，所以使用7或8个二进制位可表示

字符数据的机器表示

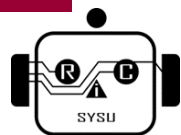
西文字符的编码表示

编码表示(常用编码为7位ASCII码)

- 十进制数字: 0/1/2.../9
- 英文字母: A/B/.../Z/a/b/.../z
- 专用符号: +/ - /%/*/&/.....
- 控制字符(不可打印或显示)

操作

- 字符串操作, 如:传送/比较等



字符数据的机器表示

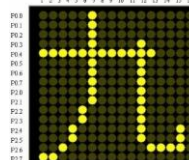
特点

汉字及国际字符编码表示

- 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

- 输入码：对每个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
- 字模点阵码或轮廓描述：描述汉字的字模点阵或轮廓，用于显示或打印



字符数据的机器表示——图像的编码表示

□ 图像在计算机中的表示

- 像素(pixel): 构成图像的最小单位, 即屏幕上的小圆点
- 颜色(color): 任何颜色都可由红、绿和蓝三色调配而成
- 像素的颜色: 用3个字节的二进制数表示



```
00000000h:  F D8 FF E1 AF A3 45 78 69 66 00 00 49 49 2A 00 ; 岑 xif..II*.
00000010h:  08 00 00 00 15 00 0F 01 02 00 12 00 00 00 0A 01 ; .....
00000020h:  00 00 10 01 02 00 0A 00 00 00 1C 01 00 00 1A 01 ; .....
00000030h:  05 00 01 00 00 00 26 01 00 00 1B 01 05 00 01 00 ; .....
00000040h:  00 00 2E 01 00 00 28 01 03 00 01 00 00 00 02 00 ; .....
00000050h:  08 00 31 01 02 00 0A 00 00 00 36 01 00 00 32 01 ; .....
00000060h:  02 00 14 00 00 00 40 01 00 00 13 02 03 00 01 00 ; .....
00000070h:  00 00 02 00 00 00 69 87 04 00 01 00 00 00 5C 01 ; .....
00000080h:  00 00 25 88 04 00 01 00 00 00 3A 89 00 00 01 A4 ; .....
00000090h:  03 00 01 00 00 00 00 00 A8 06 02 A4 03 00 01 00 ; .....
000000a0h:  00 00 01 00 00 00 03 A4 03 00 01 00 00 00 01 00 ; .....
000000b0h:  14 00 04 A4 05 00 01 00 00 00 54 01 00 00 05 A4 ; .....
000000c0h:  03 00 01 00 00 00 34 00 A8 06 06 A4 03 00 01 00 ; .....
000000d0h:  00 00 00 00 00 00 07 A4 03 00 01 00 00 00 00 00 ; .....
000000e0h:  1A 00 08 A4 03 00 01 00 00 00 00 00 A8 06 09 A4 ; .....
000000f0h:  03 00 01 00 00 00 00 00 A8 06 0A A4 03 00 01 00 ; .....
00000100h:  00 00 02 00 00 00 0C A4 03 00 01 00 00 00 00 00 ; .....
00000110h:  20 00 4C 89 00 00 4E 49 4B 4F 4E 20 43 4F 52 50 ; ..L?.NIKON CORP
00000120h:  4F 52 41 54 49 4F 4E 00 4E 49 4B 4F 4E 20 44 39 ; ORATION.NIKON D9
00000130h:  30 00 2C 01 00 00 01 00 00 00 2C 01 00 00 01 00 ; 0.....
00000140h:  00 00 56 65 72 2E 31 2E 30 30 20 00 32 30 31 32 ; ..Ver:1.00 .2012
00000150h:  3A 30 39 3A 31 33 20 31 32 3A 34 36 3A 30 31 00 ; :09:13 12:46:01.
00000160h:  01 00 00 00 01 00 00 00 1B 00 9A 82 05 00 01 00 ; .....
00000170h:  00 00 A6 02 00 00 9D 82 05 00 01 00 00 00 AE 02 ; ..?....
00000180h:  00 00 22 88 03 00 01 00 00 00 01 00 00 00 00 00 ; ..".....
00000190h:  07 00 04 00 00 00 30 32 32 31 03 90 02 00 14 00 ; .....0221.?.
000001a0h:  00 00 B6 02 00 00 04 90 02 00 14 00 00 00 CA 02 ; ..?...?
000001b0h:  00 00 01 91 07 00 04 00 00 00 01 02 03 00 02 91 ; ..?...?
000001c0h:  05 00 01 00 00 00 DE 02 00 00 04 92 0A 00 01 00 ; .....?
000001d0h:  00 00 E6 02 00 00 05 92 05 00 01 00 00 EE 02 ; ..?...?
000001e0h:  00 00 07 92 03 00 01 00 00 00 03 00 00 08 92 ; ..?...?
000001f0h:  03 00 01 00 00 00 00 00 08 01 09 92 03 00 01 00 ; .....?
00000200h:  00 00 00 00 00 00 0A 92 05 00 01 00 00 00 F6 02 ; .....?
```

本讲内容

□ 什么是计算机语言？

□ 指令集概述 Introduction to Instruction Set

□ 指令格式

□ 寻址方式

□ 数据表示方法

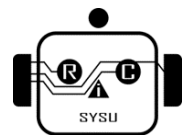
□ 数值数据表示

□ 数据的十进制表示

□ 非数值数据表示

□ 数据的存储

□ 数据的校验



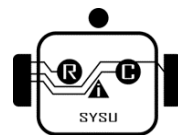
数据的度量与存储

数据的度量单位

描述信息不同
单位换算有差异

度量单位	缩写	存储二进制时 换算关系	描述计算机通信带宽时 换算关系
千字节	KB	$1\text{KiB}=2^{10}\text{字节}=1024\text{B}$	$1\text{KB}=10^3\text{字节}=1000\text{B}$
兆字节	MB	$1\text{MiB}=2^{20}\text{字节}=1024\text{KB}$	$1\text{MB}=10^6\text{字节}=1000\text{KB}$
千兆字节	GB	$1\text{GiB}=2^{30}\text{字节}$ $=1024\text{MB}$	$1\text{GB}=10^9\text{字节}=1000\text{MB}$
兆兆字节	TB	$1\text{TiB}=2^{40}\text{字节}=1024\text{GB}$	$1\text{TB}=10^{12}\text{字节}=1000\text{GB}$

(ch 1.1.1 Fig.1-1)



数据的度量与存储

数据存储方式

从80年代开始，几乎所有机器都采用**字节编址(Byte Addressing)**



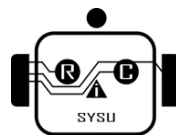
ISA设计时要考虑的两个问题

1、如何基于一个字节地址取到一个包含多个字节的字？

字的存放问题

2、一个字能否存放在任何字节边界？

字的边界对齐问题

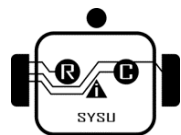


数据的度量与存储

➤ 数据存放方式

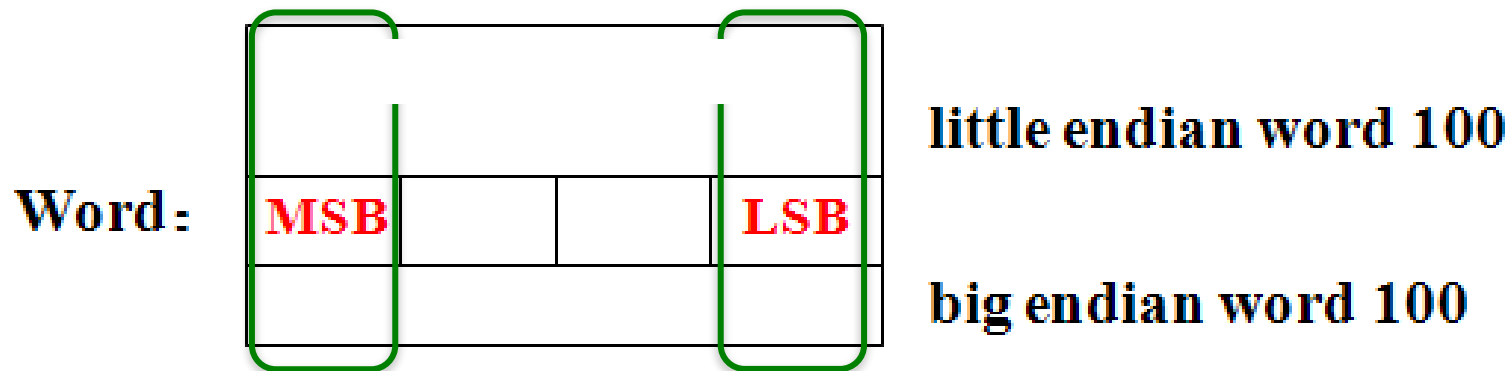
➤ 大端方式(Big Endian)

➤ 小端方式(Little Endian)



数据的度量与存储

例1: 若 $\text{int } i = 0x01234567$, 存放在内存100号单元, 用“取数”指令从内存100号单元取出 i 时, 程序员必须清楚数据 i 的4个字节在内存是如何存放的。



数据存放

- 大端方式(Big Endian): MSB(Most Significant Bit, 最高有效位)所在的地址是数的地址 (先存最高位)
e.g. IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- 小端方式(Little Endian): LSB(Least Significant Bit, 最低有效位)所在的地址是数的地址 (先存最低位)
e.g. Intel 80x86, DEC VAX

有些机器两种方式都支持, 需要通过特定控制位来设定 (ARM, Alpha)

数据的度量与存储

数据存放

例2: struct rec{
 char c;
 short si;
 int i;
 float sf;
} rec_var = {0xab, 2098, -1028, 0.625};

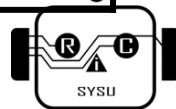
注: 2098=0x0832, -1028=0xffff fbfc, 0.625=0x3f20

大端地址映射

00	01	02	03	04	05	06	07
ab	08	32	ff	ff	fb	fc	3f
08	09	0A	0B	0C	0D	0E	0F
20	00	00	00	00	00	00	00

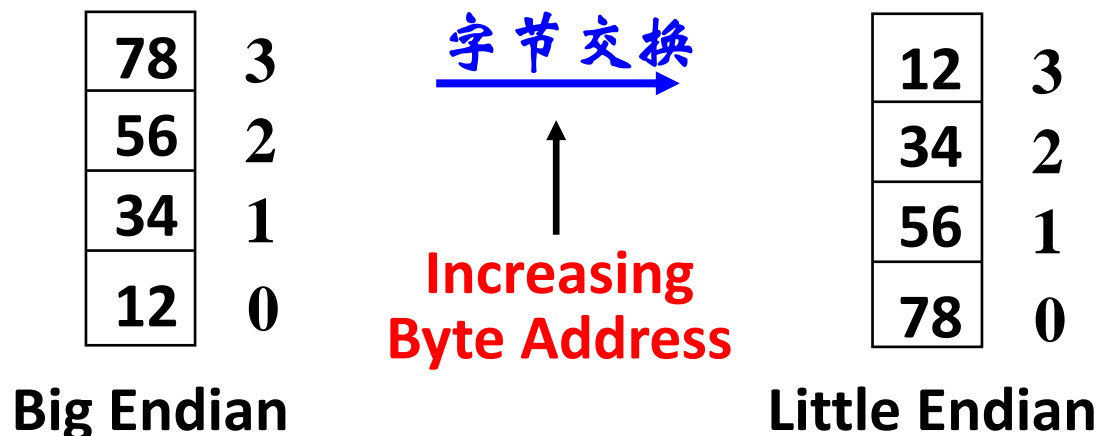
小端地址映射

07	06	05	04	03	02	01	00
20	ff	ff	fb	fc	08	32	ab
0F	0E	0D	0C	0B	0A	09	08
00	00	00	00	00	00	00	3f



数据的度量与存储

数据存放



为什么会发生字节交换呢？

存放方式不同的机器之间程序移植或数据通信时，可能发生问题

- 由于存放顺序不同，数据存储访问时，需要进行数据的顺序转换
- 音频、视频和图像等文件格式或处理程序都涉及字节的顺序问题

例： Little endian: GIF, PC Paintbrush, Microsoft RTF等

Big endian: Adobe Photoshop, JPEG, MacPaint等

数据的度量与存储

数据对齐



目前的计算机所用数据字长一般为32位或64位，而存储器地址是按字节编址

指令系统通常支持对字节、半字、字及双字的运算，还有一些按位处理的指令

不同长度的数据存放时，有两种处理方式

- 按边界对齐 (假定字的宽度为32位，存储器按字节编址)
 - ✓ 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



数据的度量与存储

数据对齐

例：假设数据顺序：字-半字-双字-字节-半字-.....

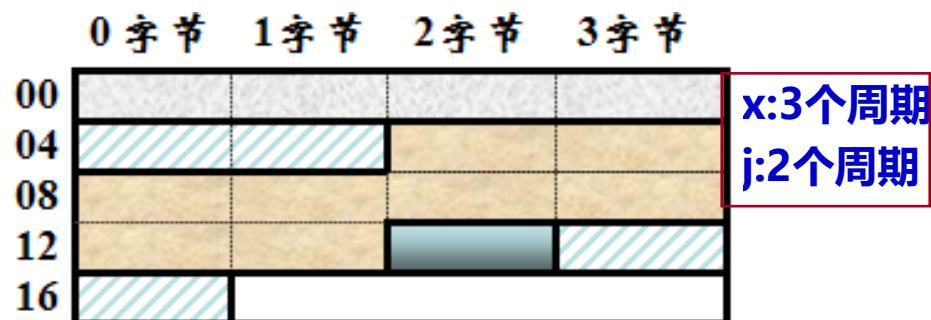
如：int i, short k, double x, char c, short j,.....

边界对齐



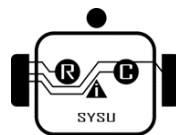
则：&i=0; &k=4; &x=8;
&c=16; &j=18;.....

边界不对齐



则：&i=0; &k=4; &x=6;
&c=14; &j=15;.....

增加了访存次数！！！



本讲内容

□ 什么是计算机语言？

□ 指令集概述 Introduction to Instruction Set

□ 指令格式

□ 寻址方式

□ 数据表示方法

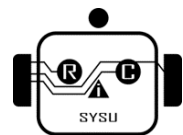
□ 数值数据表示

□ 数据的十进制表示

□ 非数值数据表示

□ 数据的存储

□ 数据的校验



数据的检/纠错

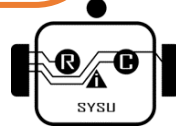
日常汉语书写中难免出现**错别字**，**纠正**是关键



计算机工作过程中是否会发生**数据错误**呢？



数据存取和传送时，由于元器件故障或噪音干扰等原因会出现差错

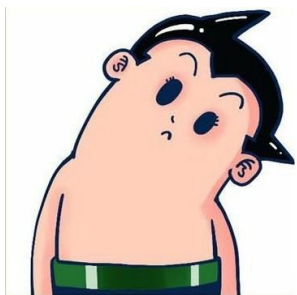


数据出错的结果会如何？

具体措施



- 从计算机硬件本身的可靠性入手，在电路、电源、布线等各方面采取必要的措施，提高计算机的抗干扰能力
- 采取相应的数据检错和校正措施，自动地发现并纠正错误



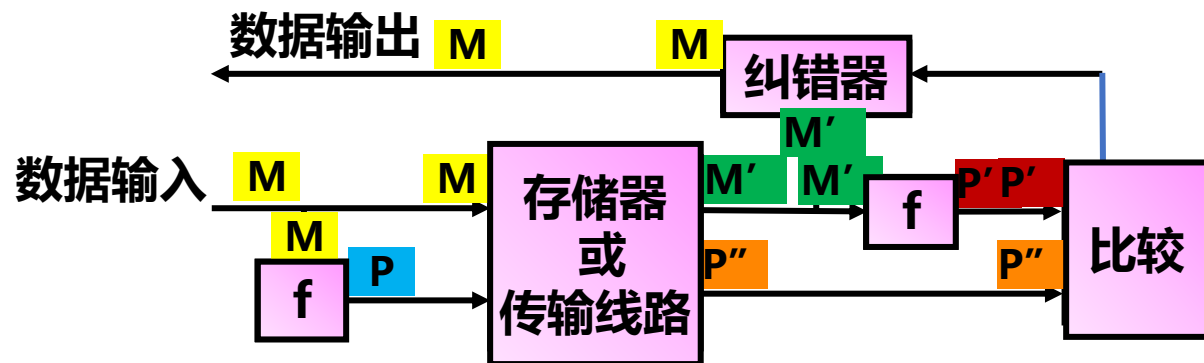
如何进行数据错误检测与校正？

采用“**冗余校验**”思想，即除原数据信息外，还增加若干位编码，这些新增的代码被称为**校验位**

数据的检/纠错与奇偶校验码



工作原理



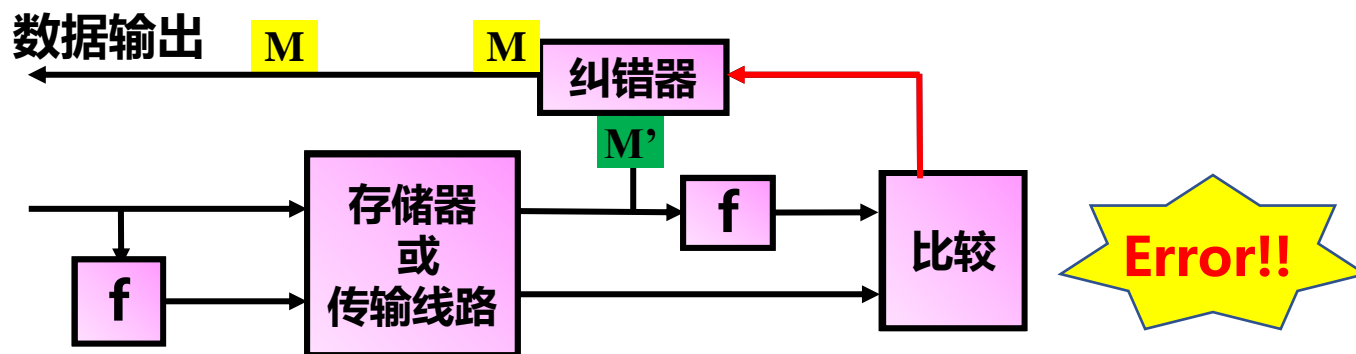
相同，即 M' 等于 M

① 未检测到错误，得到的数据位直接送出

数据的检/纠错与奇偶校验码

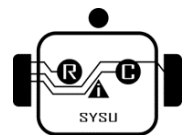


工作原理



① 未检测到错误，得到的数据位直接送出

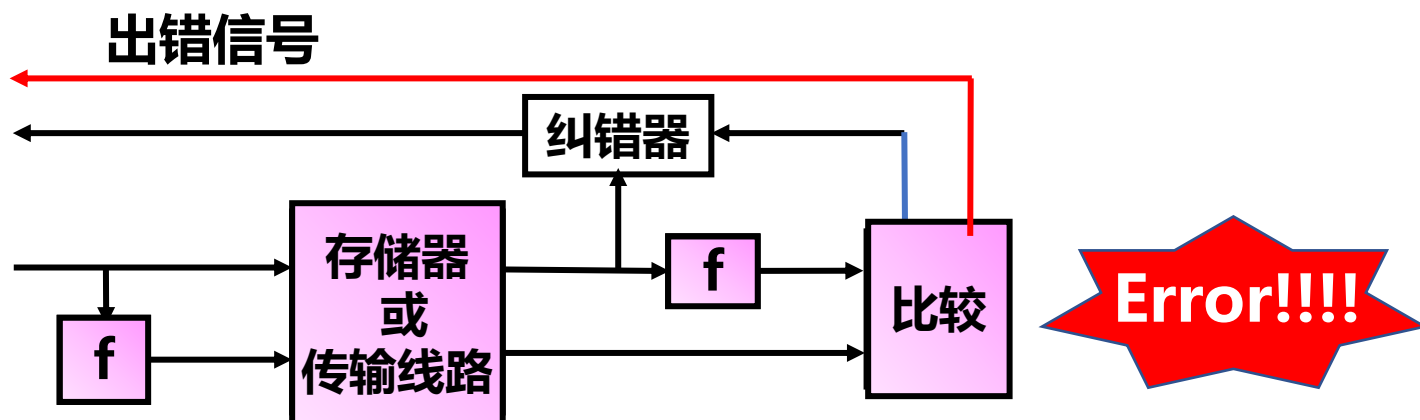
② 检测到错误并可以纠错。数据位和比较结果一起送入纠错器，将正确数据位送出



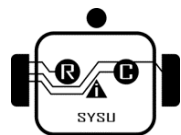
数据的检/纠错与奇偶校验码



工作原理



- ① 未检测到错误，得到的数据位直接送出
- ② 检测到错误并可以纠错。数据位和比较结果一起送入纠错器，将正确数据位送出
- ③ 检测到错误，但无法确认哪位出错，不能进行纠错处理，此时，报告出错情况



奇偶校验码



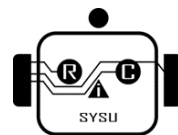
基本思想

增加一位奇(偶)校验位并一起存储或传送，由目的部件将得到的相应数据和校验位，再求新校验位，最后根据新校验位确定是否发生错误



实现原理

假设数据 $B = b_{n-1}b_{n-2} \dots b_1b_0$ 从源部件传送至目的部件。
在目的部件接收到的数据为 $B' = b_{n-1}' b_{n-2}' \dots b_1' b_0'$



数据的检/纠错

奇偶校验码

第一步:

在源部件求出奇(偶)校验位P

$$\begin{array}{l} \text{奇校验, 则 } P = b_{n-1} \oplus b_{n-2} \oplus \dots \oplus b_1 \oplus b_0 \oplus 1 \\ \text{偶校验, 则 } P = b_{n-1} \oplus b_{n-2} \oplus \dots \oplus b_1 \oplus b_0 \end{array}$$

第二步:

在目的部件求出奇(偶)校验位P''

$$\begin{array}{l} \text{奇校验, 则 } P'' = b'_{n-1} \oplus b'_{n-2} \oplus \dots \oplus b'_1 \oplus b'_0 \oplus 1 \\ \text{偶校验, 则 } P'' = b'_{n-1} \oplus b'_{n-2} \oplus \dots \oplus b'_1 \oplus b'_0 \end{array}$$

第三步:

计算最终的校验位P*, 并根据其值判断有无奇偶错

假定P在目的部件接收到的值为P', 则
 $P^* = P' \oplus P''$

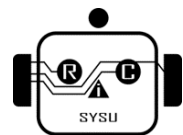
- ① 若 $P^* = 1$, 则表示数据有奇数位错
- ② 若 $P^* = 0$, 则表示数据正确或有偶数个错

奇偶校验码

- 若两个数中有奇数位不同，则它们相应的校验位就不同；
- 若有偶数位不同，则虽校验位相同，但至少有一位数据位不同

👉 特点

- 只能发现奇**数位出错**，不能发现偶数位出错，而且也不能确定发生错误的位置，**不具有纠错能力**
- 开销小，适用于校验一字节长的代码，故常被用于存储器读写检查或按字节传输过程中的数据校验



小结——数据的表示

● 数值表示

- 定点数表示(整数、小数)
- 浮点数的表示 (IEEE 754表示)
- 十进制数的表示

● 非数值表示

- 字符表示

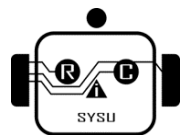
● 数据的度量

● 数据的存储

- 大端方式
- 小端方式

● 数据对齐问题

- 按边界对齐, 可以提高访存效率



联系方式

□ Acknowledgements:

□ This slides contains materials from following lectures:

- Computer Architecture (ETH, NUDT, USTC)

□ Research Area:

- 计算机视觉与机器人应用计算加速,
- 人工智能和深度学习芯片及智能计算机

□ Contact:

- 中山大学计算机学院
- 管理学院D101 (图书馆右侧)
- 机器人与智能计算实验室
- cheng83@mail.sysu.edu.cn

2022/9/23

