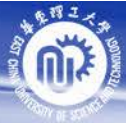


计算机原理

第二章 计算机的语言

——指令系统



Introduction to Instruction Set



25482+426858=??????????????

2.1.1 指令系统概述

```
a=25482;  
b=426858;  
c=a+b;
```

```
STORE a 25482;  
STORE b 25482;  
ADD c,a,b;
```

```
010100101001  
011100101010  
100110000010  
...
```

自然语言

高级程序语言

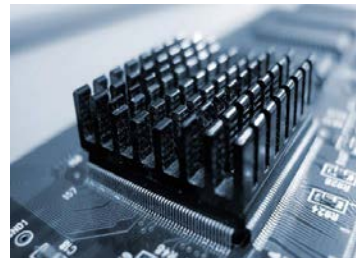
汇编语言

机器语言



系统概述

010100101001
011100101010
100110000010
....

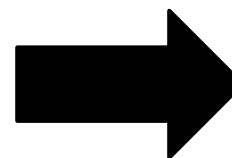


机器“语言”
二进制：0和1
硬件：逻辑电路



“指令” 计算机实现
某个基本操作的命令

指令：0101110010101010
基本操作：load r1 1382



“指令系统”
一台计算机的
所有指令
集合



指令

软件层
抽象

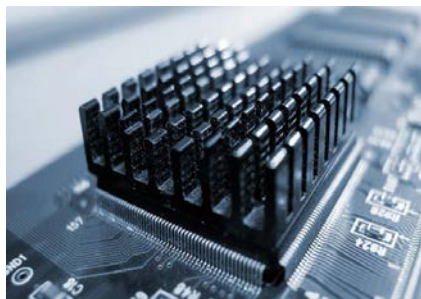
Applications

Compiler

OS

指令系统

计算机硬件和软件
的接口及界面



硬件层
抽象

Instruction
Processing

Input/
Output

Datapath & Control

Digital Design

Circuit Design



指令系统哪来的？

010100101001
011100101010
100110000010
....

1问：你是????? 怎么不认识！

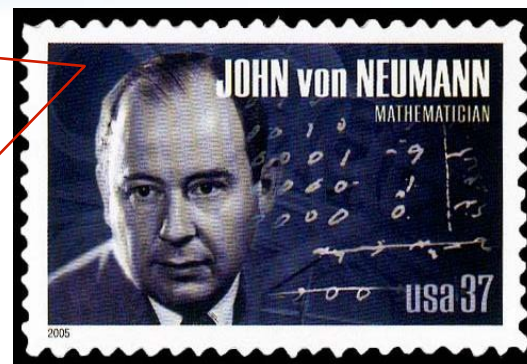
答：叫我约翰吧（约翰 冯 诺依曼）

2问：我的弟兄，指令系统是不是计算机自己产生的？

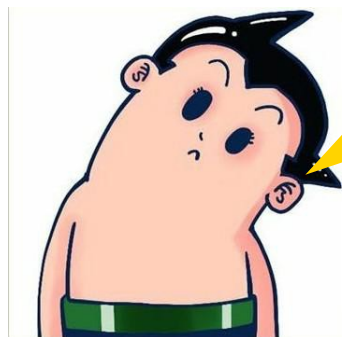
答：不是。你以为是孙悟空啊

3问：那是哪来的??? 莫非????

答：想想“存储程序”原理



现代计算机之父——冯·诺依曼

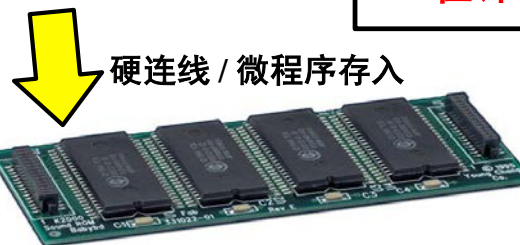


“存储程序”原理???
虽然我读书少，但在第一章刚好学过㉔

存储程序原理

指令系统
010100101001
.....

把设计好的指令系统预先实现 / 存放在计算机之中



硬连线 / 微程序存入



指令

指令系统放在哪儿很简单，
复杂的是**如何设计**指令系统

设计原则 完备

性

有效性

整性

兼容性

该有的都要有

简洁、加速常用操作、没有歧义

对称、匀齐、一致（简单源于规整）

之前/之后的都要能用

“兼容性” ???
2.1.1 指令系统概述

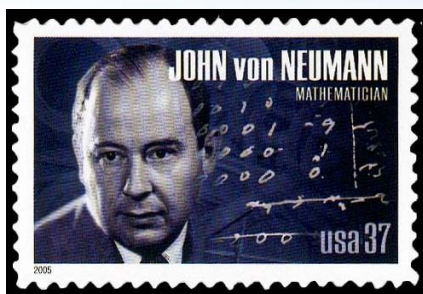
兼容性

- **向上（下）兼容**：按某档机器编制的程序，不加修改的就能运行于比它高（低）档的机器
- **向前（后）兼容**：按某个时期投入市场的某种型号机器编制的程序，不加修改就能运行于在它之前（后）投入市场的机器

机器档次

当前机器

时间



指令

指令系统放在哪儿很简单，
复杂的是**如何设计**指令系统

设计原则

完备性
有效性
规整性
兼容性

该有的都要有

简洁、加速常用操作、没有歧义

对称、匀齐、一致

之前/之后的都要能用

一个较完善的指令系统应该包括

数据传送指令

输入输出指令

算术运算指令

逻辑运算指令

系统控制指令

程序控制指令

Load/Save指令

In/Out指令

Add等指令

And等指令

中断等指令

Jump等指令



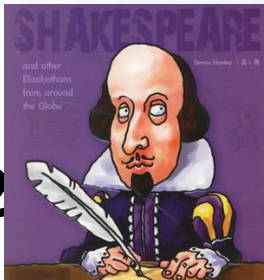
2.1.1 指令系统概述

2.1 什么是计算机的语言？

2.1.2 两种类型指令系统计算机：CISC与RISC

WALL·E





There are a thousand Hamlets in a thousand people's eyes.

2

类型指令系统计算机CISC与RISC

但是，指令系统的设计大体上**只有两种**，不是一千种



复杂指令集计算机
Complex Instruction Set Computer



精简指令集计算机
Reduce Instruction Set Computer

2



- (1) 指令系统复杂
- (2) 指令周期长
- (3) 各种指令都能访问存储器
- (4) 有专用寄存器
- (5) 采用微程序控制
- (6) 难以进行编译优化生成高效目标代码

出现较早，大而全

大家好，我是CISC

例：VAX-11/780小型机

- 16种寻址方式
- 9种数据格式
- 303条指令
- 一条指令包括1~2个字节的操作码和下续N个操作数说明符
- 一个说明符的长度达1~10个字节
- 除专门的存储器读写指令外，运算指令也能访问存储器

2



- (1) 采用微程序控制
- (2) 有专用寄存器

出现较早，大而全

大家好，我

➤难以保证设计的正确性，难以调试和维护

➤机器的时钟周期长，降低了系统性能

➤效率低下（*IBM*的测试发现）

➤指令系统中约占20%的简单指令，在程序中的比例约为



Top 10 80x86 Instructions

Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

简单指令占主要部分
使用频率高！

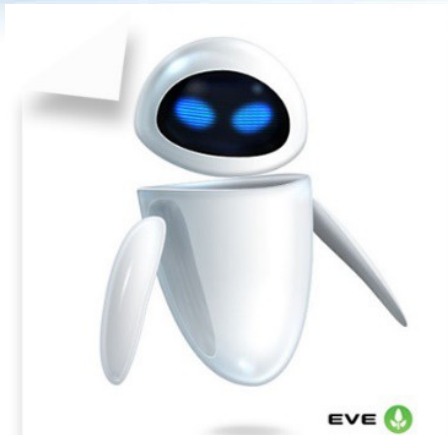
- Simple instructions dominate instruction frequency





John Cocke & 小而精

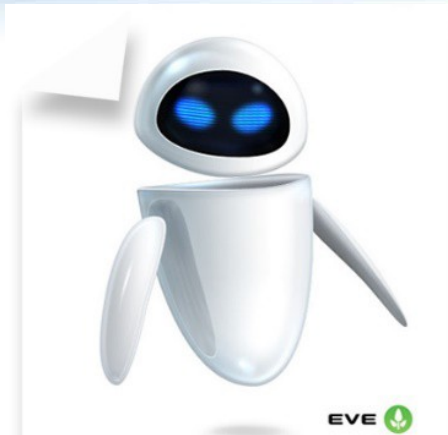
- (1) 简化的指令系统
- (2) 以寄存器-寄存器方式工作
- (3) 指令周期短
- (4) 采用大量通用寄存器，以减少访存次数
- (5) 采用组合逻辑电路控制，不用或少用微程序控制
- (6) 采用优化的编译系统，力求有效地支持高级语言程序



Hi, 我是RISC

John Cocke & 小而精

- (1) 简化的指令系统
- (2) 以寄存器-寄存器方式工作
- (3) 指令周期短
- (4) 采用大量通用寄存器，以减少访存次数
- (5) 采用组合逻辑电路控制，不用或少用微程序控制
- (6) 采用优化的编译系统，力求有效地支持高级语言程序



Hi, 我是RISC



我是CISC

- (1) 指令系统复杂
- (2) 各种指令都能访问存储器
- (3) 指令周期长
- (4) 有专用寄存器
- (5) 采用微程序控制
- (6) 难以进行编译优化生成高效目标代码

出现较早，大而全

John Cocke & 小而精

2.例2: 两种典型结构系统计算机CISC与RISC

- 加州伯克利大学的RISC I
- 斯坦福大学的MIPS
- IBM公司的IBM801



CISC与RISC之争

- 现代处理器大多采用RISC体系结构
- Intel x86为“兼容”需要，保留CISC风格，同时借鉴了RISC思想



2.1.2 两种类型指令系统计算机CISC与RISC



CISC与RISC的逐步融合



2.2 计算机的语言什么样？ —— 指令格式



指令含义

指令是指挥计算机实现**某个基本操作**的命令

决定

指令格式

什么操作？

操作码

操作码

操作的对象？

操作数

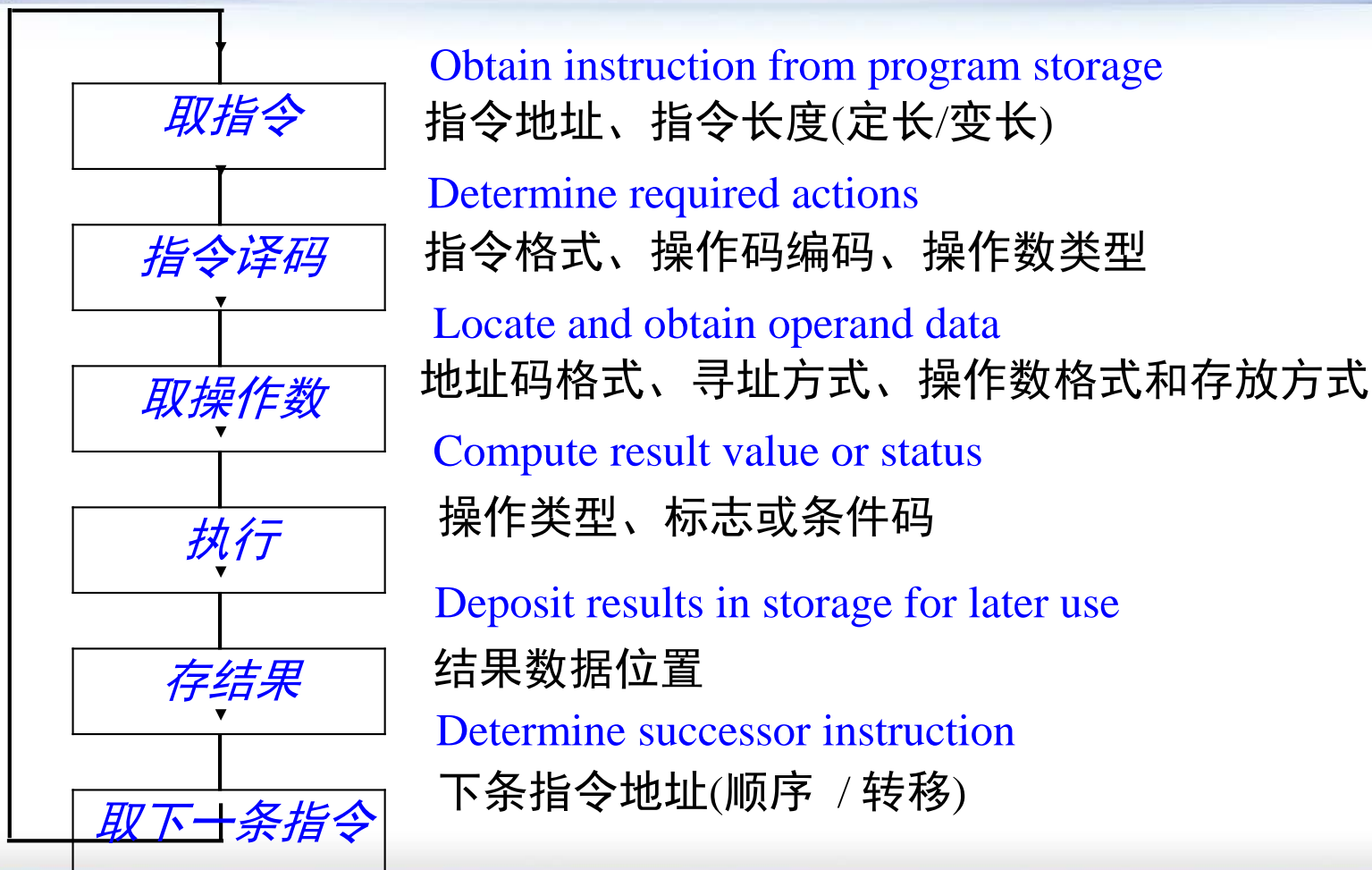
如何找到操作对象？

寻址方式

指令长度



从指令执行过程看指令设计涉及的问题





指令格式

操作码

地址码

指令长度的设计

1. 问：每条指令的长度可以是不一样的么？

指令长度

➤ 一条指令包含的二进制代码位数。

➤ 取决于操作码长度、操作数地址长度和地址个数。

❖ 定长指令字：所有指令的长度相同。需向最长指令看齐

❖ 变长指令字：不同指令的长度不同

设计原则之一

规整性



指令格式

操作码

地址码

操作码设计

设计原则之一

1. 问：每条指令的操作码可以是几个？

1. 答：只能是一个。

有效性

2. 问：具体的操作是怎么表示的？

2. 答：用一定长度的不同编码表示不同的操作？

完备性

设计原则之二

3. 问：长度是一定的么？怎么编码？

3. 答：不要着急，等等下一节告诉你。



指令格式

操作码

地址码

地址码设计

问：每条指令的地址码可以是几个？

答：0到多个，看操作码的需要了

□一条指令包含1个操作码和多个地址码

➤ 零地址指令

OP

堆栈

➤ 一地址指令

OP

A1

累加器

➤ 二地址指令(最常用) 三

OP

A1

A2

➤ 地址指令(RISC风格)

OP

A1

A2

A3

通用寄存器

➤ 多地址指令

地址码个数与性能和实现难度密切相关



指令格式

操作码

地址码

操作码长度

操作码长度问题

- 关注程序代码长度时：变长指令字、变长操作码
- ❑ 操作码的编码方式决定操作码的长度
- 关注性能时：定长指令字、定长操作码
- 定长操作码法

➤ 变长/扩展操作码法

❑ 操作码长度和指令长度什么关系？

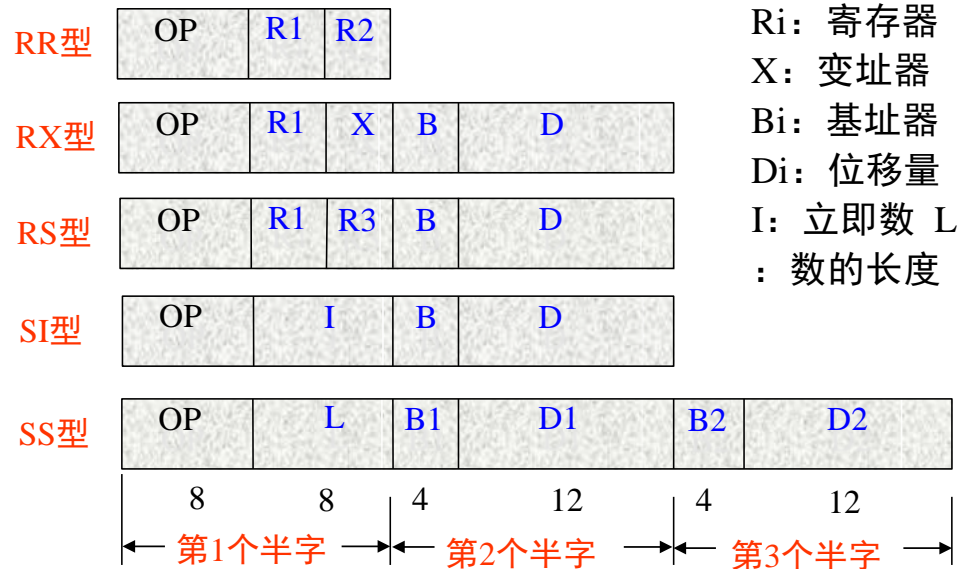
➤ 指令长度是否可变与操作码长度是否可变没有绝对联



定长操作码

基本思想：指令的操作码部分**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。



IBM370指令格式

特点

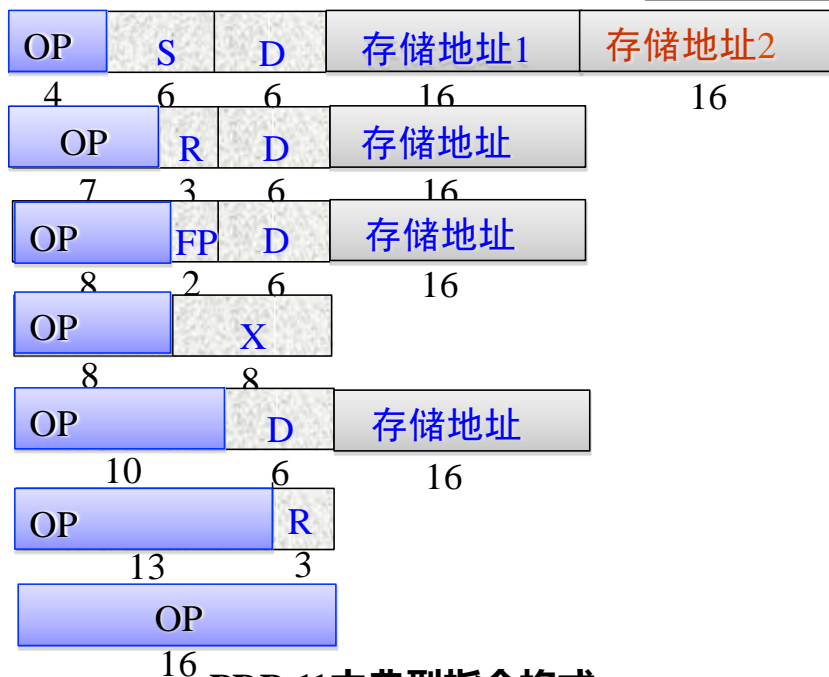
译码简单，但有信息冗余。

例：IBM360/370采用：8位定长操作码，最多可有256条指令。只提供了183条指令，有73种编码为冗余信息。



扩展操作码

❑ **基本思想**：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。



PDP-11中典型指令格式

优点

- ❑ 缩短指令长度
- ❑ 减少程序总位数
- ❑ 增加指令字所能表示的操作信息

一个重要原则：使用频度高的指令：短的操作码 使用频度低的指令：较长的操作码



扩展操作码

4 位操作码

8 位操作码

12 位操作码

16 位操作码

15				0
OP	A ₁	A ₂	A ₃	
0000	A ₁	A ₂	A ₃	
0001	A ₁	A ₂	A ₃	
⋮	⋮	⋮	⋮	
1110	A ₁	A ₂	A ₃	
1111	0000	A ₂	A ₃	
1111	0001	A ₂	A ₃	
⋮	⋮	⋮	⋮	
1111	1110	A ₂	A ₃	
1111	1111	0000	A ₃	
1111	1111	0001	A ₃	
⋮	⋮	⋮	⋮	
1111	1111	1110	A ₃	
1111	1111	1111	0000	
1111	1111	1111	0001	
⋮	⋮	⋮	⋮	
1111	1111	1111	1111	

等长扩展操作码示例
(4 - 8 - 12 - 16)

15条三地址指令

15条二地址指令

15条一地址指令

16条零地址指令



扩展操作码分析题

例：某指令系统指令字长16位，每个地址码为6位。若二地址指令15条，一地址指令34条，则剩下零地址指令最多有多少条？



例：某指令系统指令字长16位，每个地址码为6位。若二地址指令15条，一地址指令34条，则剩下零地址指令最多有多少条？

4 位操作码

15		0
OP	A ₁	A ₂
0000	A ₁	A ₂
0001	A ₁	A ₂
⋮	⋮	⋮
1110	A ₁	A ₂

15条二地址指令

10 位操作码

1111	000000	A ₂
1111	000001	A ₂
⋮	⋮	⋮
1111	011111	A ₂

32条一地址指令

10 位操作码

1111	100000	A ₂
1111	100001	A ₂

2条一地址指令



操作码计算题

1. 已知某指令的操作码为 0000，求该指令的操作码长度。

有多少条？

解：操作码按短到长进行扩

展编码 二地址指令

(0000



4 位操作码

15		0
OP	A ₁	A ₂
0000	A ₁	A ₂
0001	A ₁	A ₂
⋮	⋮	⋮
1110	A ₁	A ₂

15条二地址指令

10 位操作码

1111	000000	A ₂
1111	000001	A ₂
⋮	⋮	⋮
1111	011111	A ₂

32条一地址指令

10 位操作码

1111	100000	A ₂
1111	100001	A ₂

2条一地址指令

16 位操作码

1111	100010	000000
1111	100010	000001
⋮	⋮	⋮
1111	111111	111111

$(2^5-2) \times 2^6$ 条零地址指令



扩展操作码分析题

例：某指令系统指令字长16位，每个地址码为6位。若二地址指令15条，一地址指令34条，则剩下零地址指令最多有多少条？

解：操作码按短到长进行扩展编码 二地址指令



回顾一下指令格式设计的基本原则

指令尽量短小

码位数 指令编码必须具有

一的解释 指令字长应是字节

的整数倍 均衡设计、指令尽

量规整

合理选择地址字段的个数



地址个数

地址码结构

地址个数越少 → 指令长度越短 → 指令功能越简单 地址个数越少 →

所需指令条数越多 → 增加了程序复杂度和执行时间

地址的选择依赖于指令系统的结构

堆栈结构：零地址指令

累加器结构：一地址指令

通用寄存器结构：二、三地址指令

指令中地址
个数0~3个



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：另一操作数为默认的 如：累加器等



一条指令包含1个操作码和多个地址码

■ 二地址指令(最常用)

OP	A1	A2
----	----	----

分别存放双目运算中两个源操作数地址，并将其中一个地址作为存放结果地址

OP	A1	A2	A3
----	----	----	----

■ 三地址指令(RISC风格)

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



2.2.3 地址码结构

例： $Y = (A - B) \div (C + D \times E)$

二地址指令

指令		操 作
MOVE	Y,A	$Y \leftarrow (A)$
SUB	Y,B	$Y \leftarrow (Y) - (B)$
MOVE	T,D	$T \leftarrow (D)$
MUL	T,E	$T \leftarrow (T) \times (E)$
ADD	T,C	$T \leftarrow (T) + (C)$
DIV	Y,T	$Y \leftarrow (Y) \div (T)$

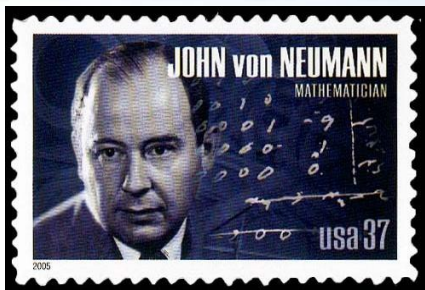
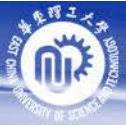
一地址指令

指令		操 作
LOAD	D	$AC \leftarrow (D)$
MUL	E	$AC \leftarrow (AC) \times (E)$
ADD	C	$AC \leftarrow (AC) + (C)$
STOR	Y	$Y \leftarrow (AC)$
LOAD	A	$AC \leftarrow (A)$
SUB	B	$AC \leftarrow (AC) - (B)$
DIV	Y	$AC \leftarrow (AC) \div (Y)$
STOR	Y	$Y \leftarrow (AC)$



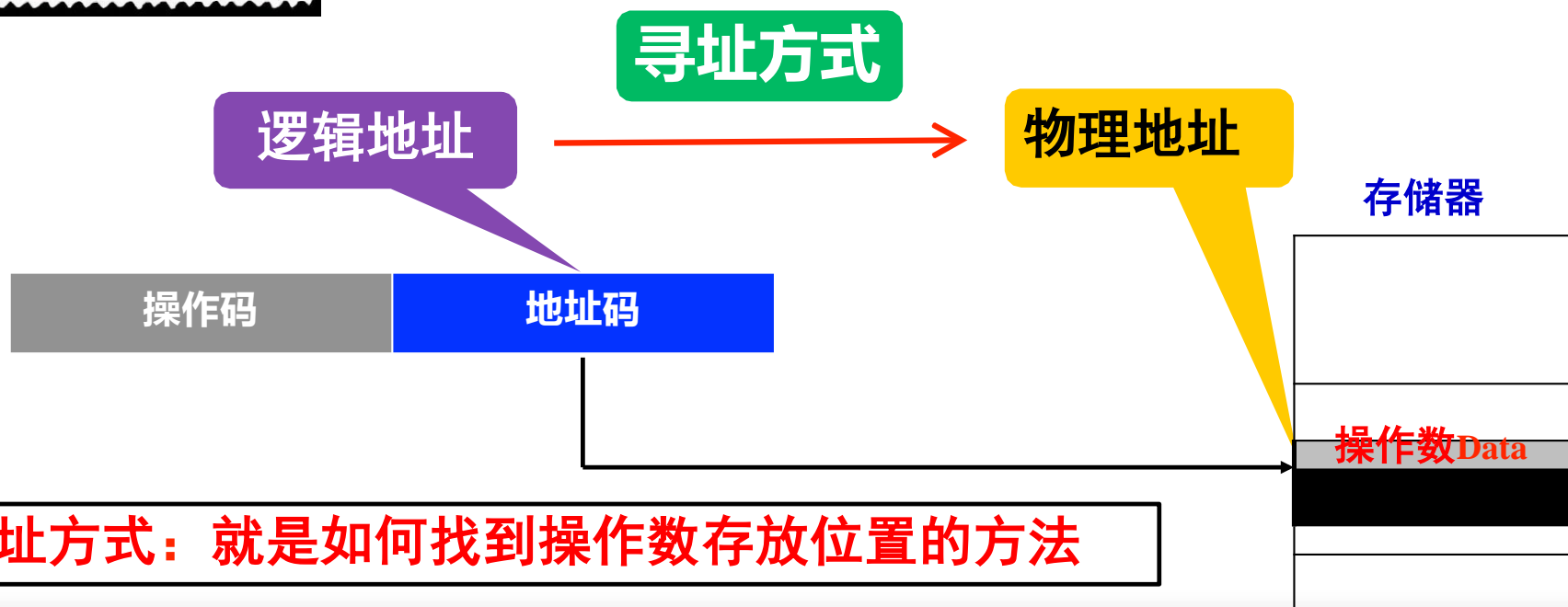
2.3 如何找到操作数？

—— 寻址方式

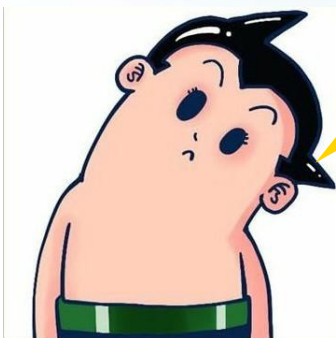


学习了操作码的编码之后，另一个问题就是**地址码如何编码**。

地址码编码由操作数的寻址方式决定。



寻址方式：就是如何找到操作数存放位置的方法



为什么不能**直接把操作数放在地址码里面**，
这样就**不用寻址方式**了????????

你说的有道理,非常合理!!! 但是**地址码的位数有限**，**更大的操作数**如何处理?



操作数Data

操作码

地址码

操作数只能这么
多位吗? 更大的
数怎么办?



好吧，为什么不能**直接把操作数地址放在地址码里面**，这样也就**不用寻址方式**了???????

你说的也有道理！！！！但是，**地址码的位数有限**，能放下多少个操作数地址？
而且，寄存器毕竟是非常少的啊！！！！



操作数物理地址

操作码

地址码

m

地址码太短，只能访问 2^m 个内存单元或寄存器中的操作数？



好吧，为什么不能**直接把操作数地址放在地址码里面**，这样也就**不用寻址方式**了????????

你说的也有道理！！！！但是，**地址码的位数有限**，能放下多少个操作数地址？
而且，寄存器毕竟是非常少的啊！！！！



操作数物理地址

操作码

地址码

寻址方式出现的目的

- **扩大访存范围**
- **提高访问数据的灵活性和有效性**
- **支持软件技术的发展：多道程序设计**



2.3.1 寻址方式的概念

指令系统中的寻址

通常寻址方式特指“**操作数寻址**”

寻找：操作数 操作的
对象放在哪了？

寻找：指令
下一条指令在哪啊？

指令寻址

➤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

➤操作数寻址——复杂

- ❖操作数的来源：寄存器 / 外设端口 / 主(虚)存 / 栈顶
- ❖操作数的数据结构：位 / 字节 / 半字 / 字 / 双字 / 一维表 / ...



基本寻址方式 (1)

立即数寻址

操作码	目的操作数	Mod	Imme. Data
-----	-------	-----	------------

↑
源操作数

➤没错，就是之前讨论的——
不要寻址方式的方法

源操作数直接在指令中

- 指令地址字段直接给出操作数本身
- 立即数寻址只能作为双操作数指令的源操作数

例：MOV AX, 1000H

1.指令执行时间很短，
无需访存

2.操作数的大小受地址
字段长度的限制

3.广泛使用

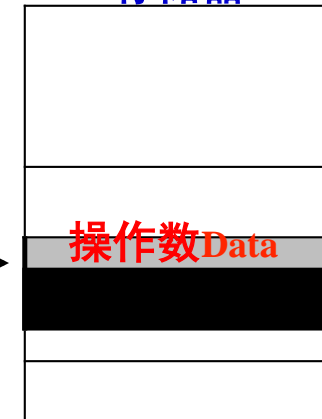


基本寻址方式 (2)

存储器直接寻址



存储器



➤ $EA = A$, Operand = (A)

➤ 例 : MOV AX , [1000H]

1. 处理简单、直接

2. 寻址空间受到指令的地址字段长度限制

3. 较少使用 , 8位计算机和一些16位计算机

操作数在存储器中, 指令地址字段直接给出操作数在存储器中的地址



基本寻址方式 (3)

寄存器直接寻址



通用寄存器组



➤ $EA = R$, $Operand = (R)$

➤ 例 : `MOV BX , AX`

1. 只需要很短的地址字段
2. 无需访存 , 指令执行速度快
3. 地址范围有限 , 可以编程使用的通用寄存器不多
4. 使用最多 , 是提高性能的常用手段

操作数在寄存器中 , 指令地址字段直接给出存放操作数的寄存器编号



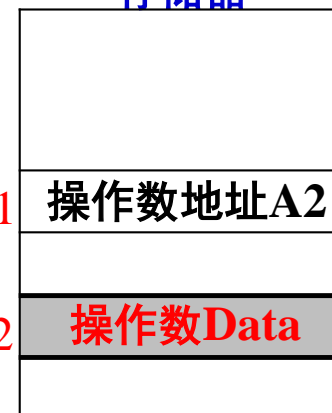
2.3.2 基本寻址方式

基本寻址方式 (4)

存储器间接寻址



存储器



$EA = (A)$, $Operand = ((A))$

例 : MOV R1, @(1000H)

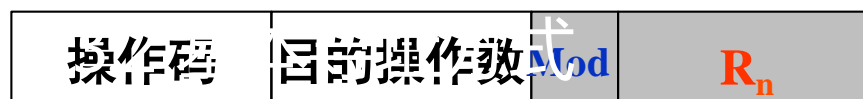
- 寻址空间大, 灵活, 便于编程
- 至少需要两次访存才能取到操作数
- 执行速度慢

- 操作数和操作数地址都在存储器中
- 指令地址字段直接给出操作数地址在存储器中的地址



基本寻址方式 (5)

寄存器间接寻址



存储器

通用寄存器组

R_n

操作数地址A

操作数Data

A

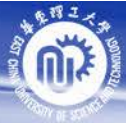
$EA = (R)$, $Operand = ((R))$

例 : `MOV AX, [BX]`

➤ 比存储器间接寻址少访问存储器一次

➤ 寻址空间大, 使用比较普遍

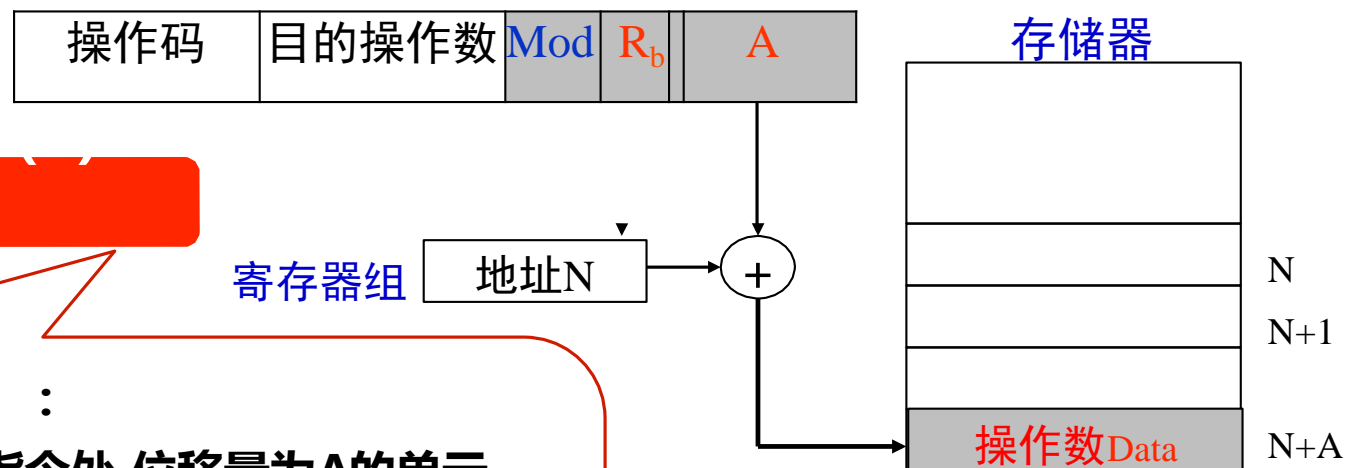
- 操作数在存储器中
- 操作数地址在寄存器中
- 指令地址字段给出的寄存器的内容是操作数在存储器中的地址



2.3.2 基本寻址方式

基本寻址方式 (6)

偏移寻址



EA =

寄存器组

地址N

+

存储器

N

N+1

N+A

操作数 Data

- $EA = (PC) + A$: 相对于当前指令处 位移量为A的单元
- $EA = (B) + A$ 相对于基址(B)处 位移量为A的单元
- $EA = (I) + A$ 相对于形式地址A处位移量为(I)的单元

直接寻址 + 寄存器间接寻址



2.3.2 基本寻址方式

基址寻址

◆◆对于一道程序，基址是不变的，程序中的所有地址都是相对于基址变化

◆◆在基址寻址中，偏移量位数较短

◆◆基址寻址立足于面向系统，主要是解决程序逻辑空间与存储器物理空间的无关性

变址寻址

◆◆对于变址寻址，形式地址给出的是一个存储器地址基准，变址寄存器存放的是相对于该基准地址的偏移量

◆◆而在变址寻址中，偏移量位数足以表示整个存储空间

◆◆而变址寻址立足于用户，主要是为编写高效访问一片存储空间的程序



2.3.2 基本寻址方式

基本寻址方式 (7)

堆栈寻址

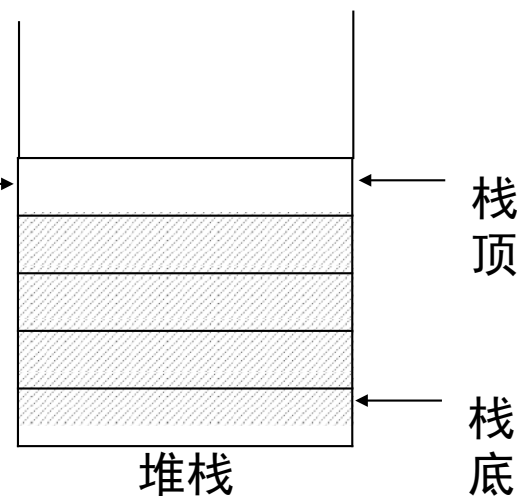
SP

A

- 堆栈的结构：**一段内存区域**
- 栈底、栈顶
- 堆栈指针(SP)：一个**特殊寄存器**，指向栈顶

PUSH (从寄存器到堆栈)

POP (从堆栈到寄存器)



零地址数



2.3.2 基本寻址方式

基本寻址方式 (7)

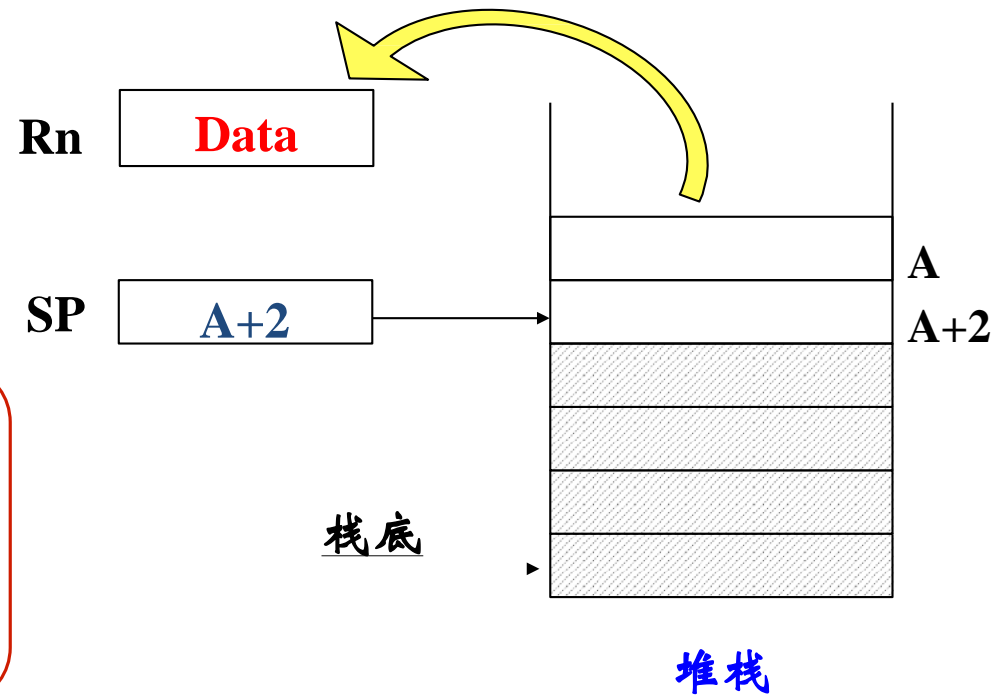
堆栈寻址

堆栈操作示例

➤ 堆栈操作

出栈操作: **POP Rn**

$Rn \leftarrow ((SP)), SP \leftarrow (SP) + 2$



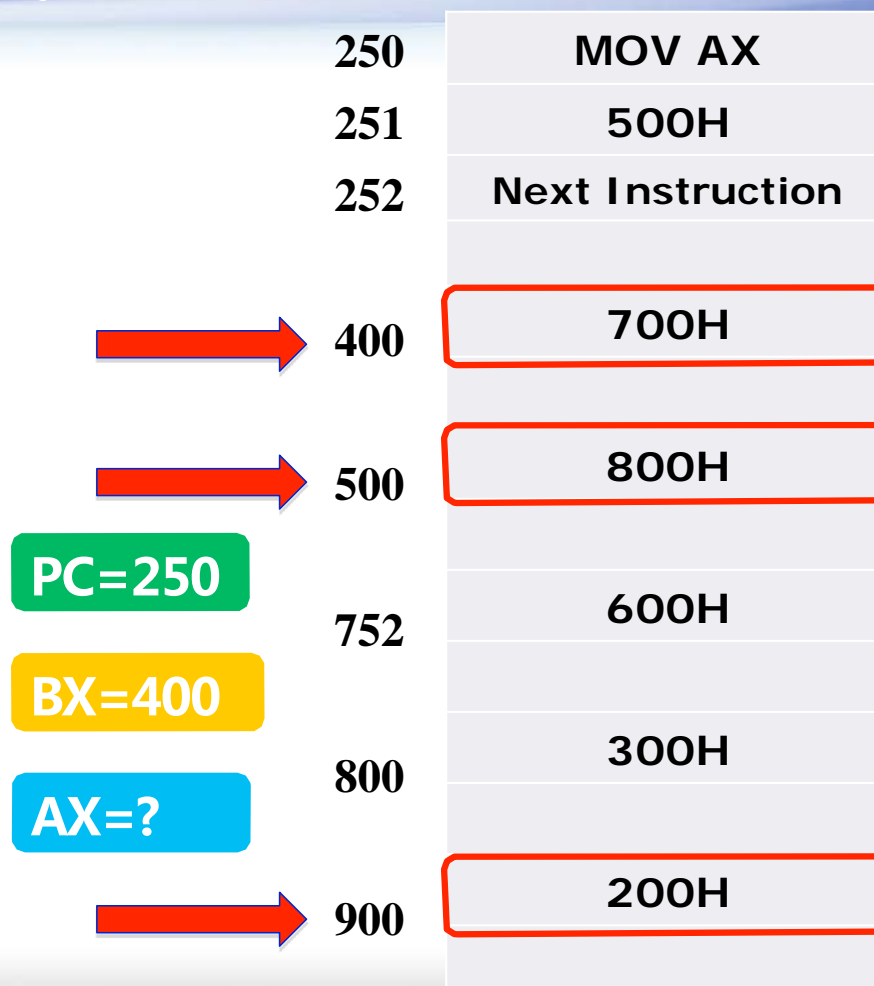


2.3.2 基本寻址方式

基本寻址方式举例

例：累加器型指令

- ①立即数寻址 MOV AX, 500H
- ②直接寻址 MOV AX, [500H]
- ③寄存器寻址 MOV AX, BX
- ④寄存器间接寻址 MOV AX, [BX]
- ⑤基址寻址 MOV AX, 500H[BX]





基本寻址方式的小结

方式	算法	主要优点	主要缺点
立即	操作数=A	指令执行速度快 有	操作数幅值有限
直接	$EA=A$	有效地址计算简单 有	地址范围有限
间接	$EA=(A)$	有效地址范围大	多次存储器访问
寄存器	操作数=(R)	指令执行快, 指令短	地址范围有限
寄存器间接	$EA=(R)$	地址范围大	一次存储器访问
偏移	$EA=A+(R)$	灵活	复杂
堆栈	$EA=栈顶$	指令短	
— 应用有限			





基本寻址方式的小结

方式	算法	主要优点	主要缺点
立即	操作数=A	指令执行速度快 有	操作数幅值有限
直接	$EA=A$	有效地址计算简单	地址范围有限
间接	$EA=(A)$	有效地址范围大	多次存储器访问
寄存器	操作数=(R)	指令执行快，指令短	地址范围有限
寄存器间接	$EA=(R)$	地址范围大	一次存储器访问
偏移	$EA=A+(R)$	灵活	复杂
堆栈	$EA=栈顶$	指令短	应用有限



思考题：七种寻址方式中，哪些操作数在寄存器中？哪些操作数在存储器中？



那么多寻址方式，我怎么知道**具体的寻址方式是怎么确定的啊？**

嗯嗯！！问得非常好，让我来告诉你



寻址方式的确定

(1) 在操作码中给定寻址方式

如：MIPS指令中仅有一个主(虚)存地址，且指令中仅有一、二种寻址方式

(2) 专门的寻址方式位：如：X86指令

0/1字节	0/1字节	1/2字节	0/1字节	0/1/2字节	0/1/2字节
指令前缀	段超越	操作码	寻址方式	位移量	立即数



2.3.3 复合寻址方式和寻址方式实例

复合寻址

间接寻址

+

相对寻址

间接寻址

+

变址寻址

在寻址的灵活性和硬件的复杂性之间权衡

先间接

➤ 间接相对式 $EA = (PC) + (A)$

➤ 间接变址式 $EA = (X) + (A)$

后间接

➤ 变址间接式 $EA = ((X) + A)$

➤ 相对间接式 $EA = ((PC) + A)$



逻辑地址

□地址

□操作数的来源

立即数(立即寻址)：直接来自指令

寄存器(寄存器寻址)：来自32位 / 16位 / 8位

通用寄存器 存储单元(其他寻址)：需进行地址转换

逻辑地址 → 线性LA (→ 内存地址)
分段 分页

□地址

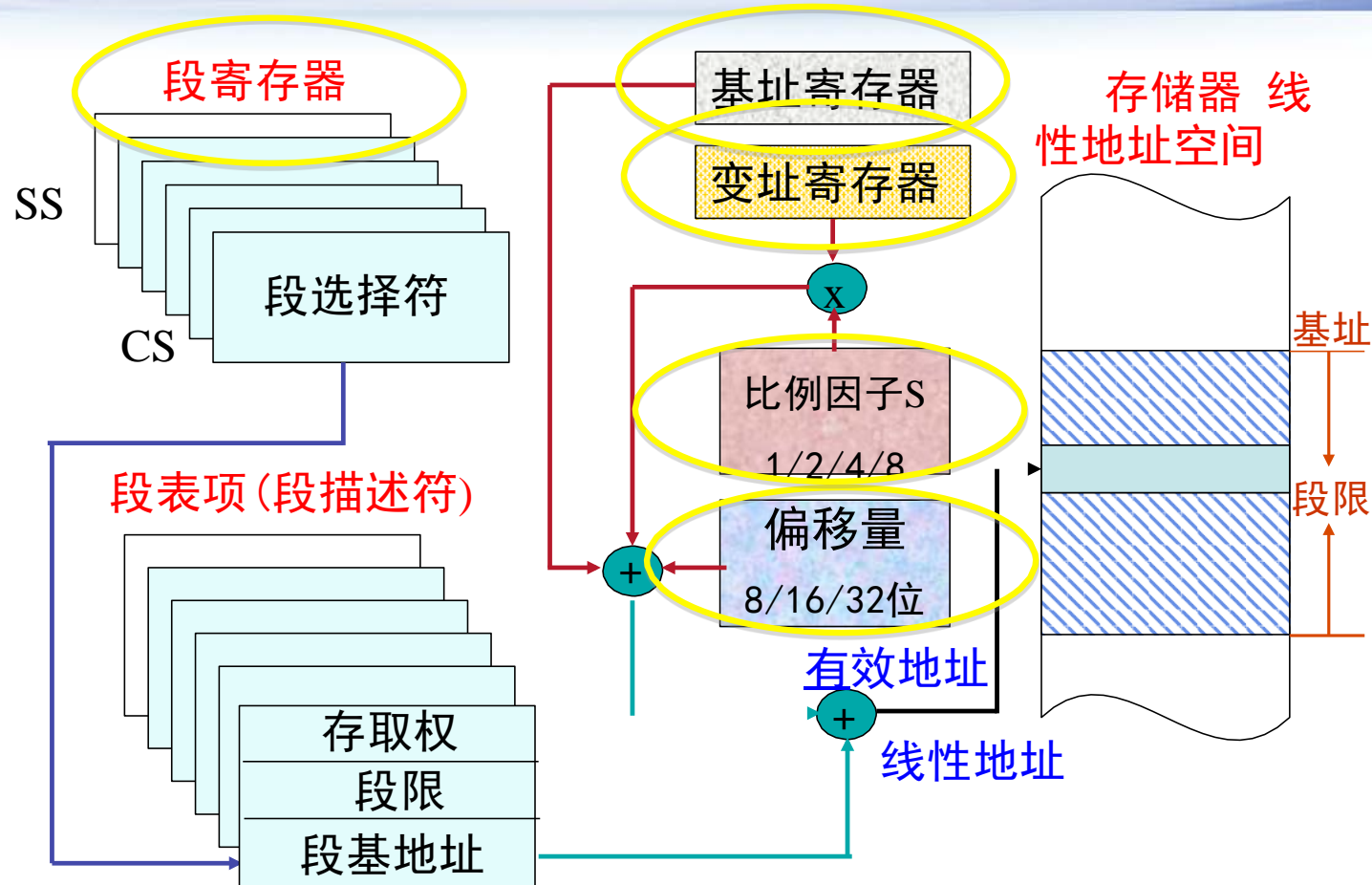
(1) 8/16/32位偏移量A (显式给出) (2)

段寄存器SR(段的起始地址；隐含或显式给出) (3) 基址寄存器B (显式给出，



2.3.3 复合寻址方式和寻址方式实例

Pentiu 处理器寻址方式





寻址方式

计算方法

立即(地址码A本身为操作数) 寄存
器(通用寄存器的内容为操作数)

操作数=A 操
作数= (R)

偏移量(地址码A给出8/16/32位偏移量)

$LA=(SR)+A$

基址(地址码B给出基址器编号) 基址

$LA=(SR)+(B)$

带偏移量(一维表访问) 比例变址带偏

$LA=(SR)+(B)+A$

移量(一维表访问) 基址带变址和偏移

量(二维表访问) 基址带比例变址和偏

移量(二维表访问) 相对(给出下一指令
的地址, 转移控制)

LA=线性地址; (X)=X中的内容; SR=段寄存器; PC=程序计数器; B=基址寄存器; I=变址寄存器



寻址方式

计算方法

立即(地址码A本身为操作数) 寄存
器(通用寄存器的内容为操作数)

操作数=A 操
作数= (R)

偏移量(地址码A给出8/16/32位偏移量)

$LA=(SR)+A$

基址(地址码B给出基址器编号) 基址

$LA=(SR)+(B)$

带偏移量(一维表访问) 比例变址带偏

$LA=(SR)+(B)+A$

移量(一维表访问) 基址带变址和偏移

$LA=(SR)+(I) \times S + A$

量(二维表访问) 基址带比例变址和偏

$LA=(SR)+(B)+(I) + A$

移量(二维表访问) 相对(给出下一指令
的地址, 转移控制)

$LA=(SR)+(B)+(I) \times S + A$

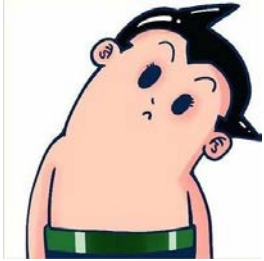
转移地址= $(PC)+A$

LA=线性地址; (X)=X中的内容; SR=段寄存器; PC=程序计数器; B=基址寄存器; I=变址寄存器



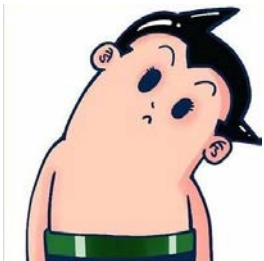
2.4 数据在计算机中如何表示？

——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



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

COMPUTER PRINCIPLE

真值和机器数

机器数：用0和1编码的计算机内部 0/1



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



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

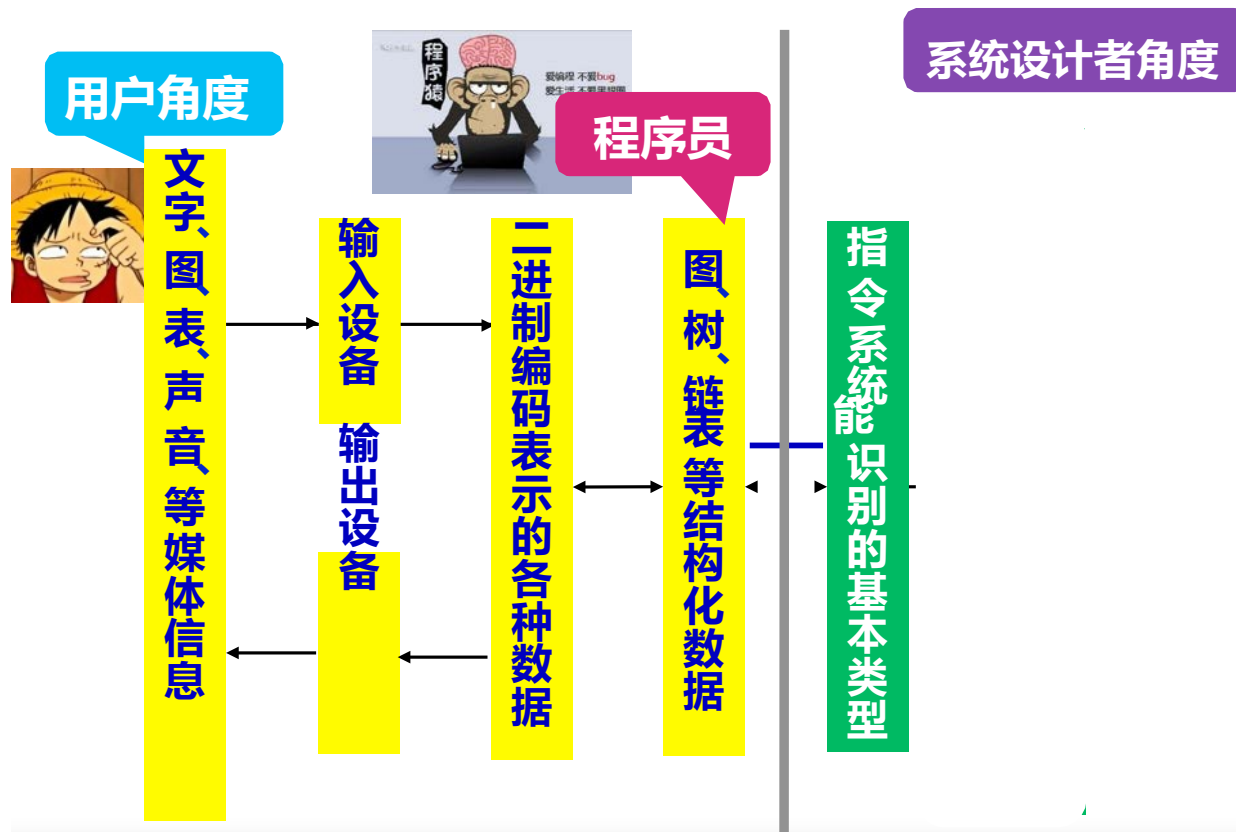
数据表示和数据结构的关系

- 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



2.4.1 计算机中的（机器级）数据表示

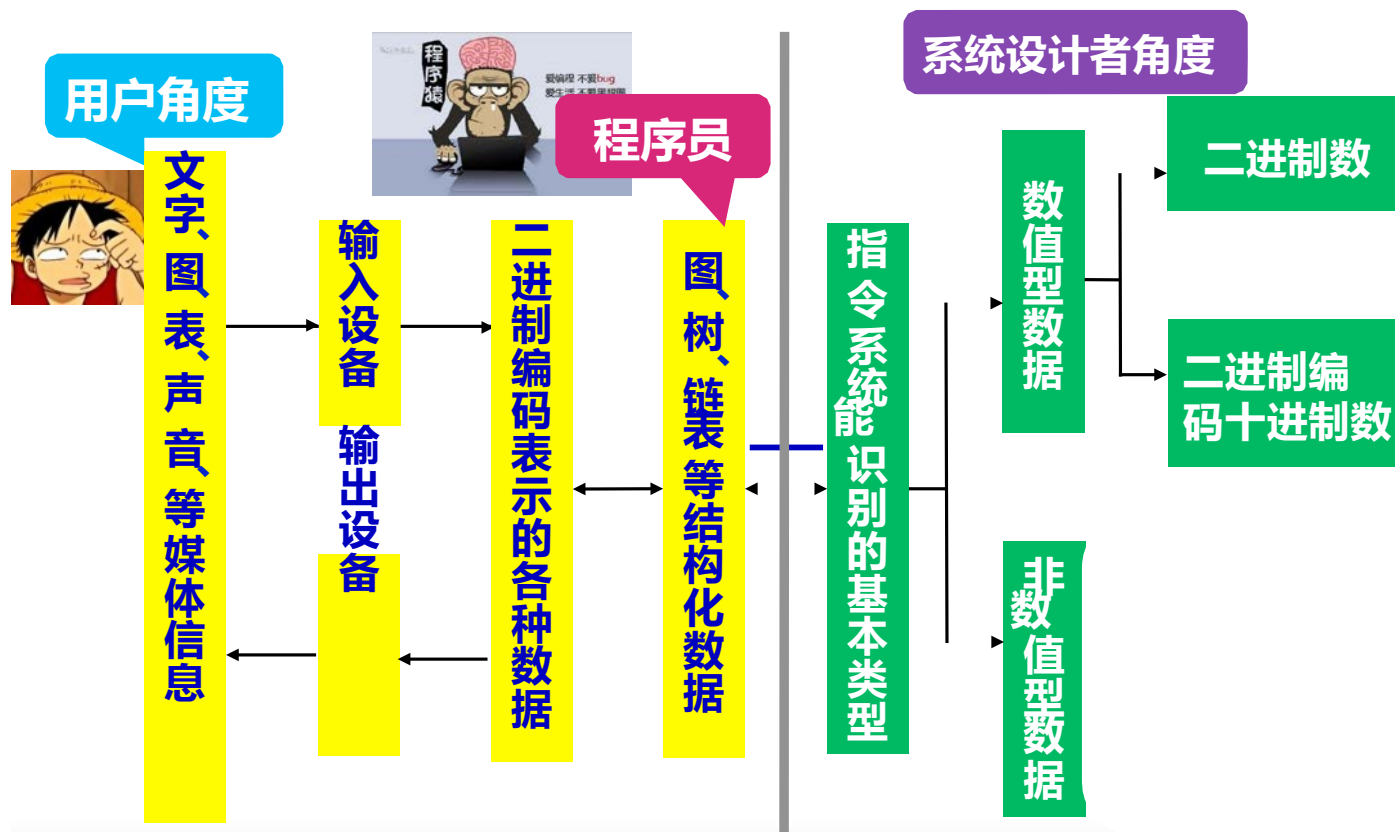
计算机的外部信息与内部机器级数据





2.4.1 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





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(字节)	Compaq-Alpha (64) 数据类型(字 节)
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



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) 现代计算机的主存是按字节编址，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



数据宽度

位

计算机处理、存储、传输信息的最小单位 (bit)

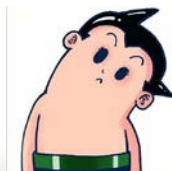
字节

计算机中二进制信息的计量单位 (Byte) 现代计算机的主存是按字节编址，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” 和 “字长” 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**
- 字和字长的宽度可以一样，也可以不同，通常是字节的整数倍



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？

第二个问题：小数点的表示？

第三个问题：零的表示？

第四个问题：实数的表示？



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

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



2.4.2 数值数据的定点表示

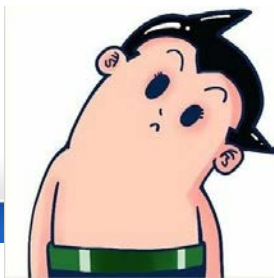


问题二：如何表示小数点？

定点数

解决方法：小数点的位置固定

- 小数点左边的进数是整数,右边是小数
- 计算机通常将数分成整数和定点数
- 小数点位置由数据类型默认决定(即在程序员心里)

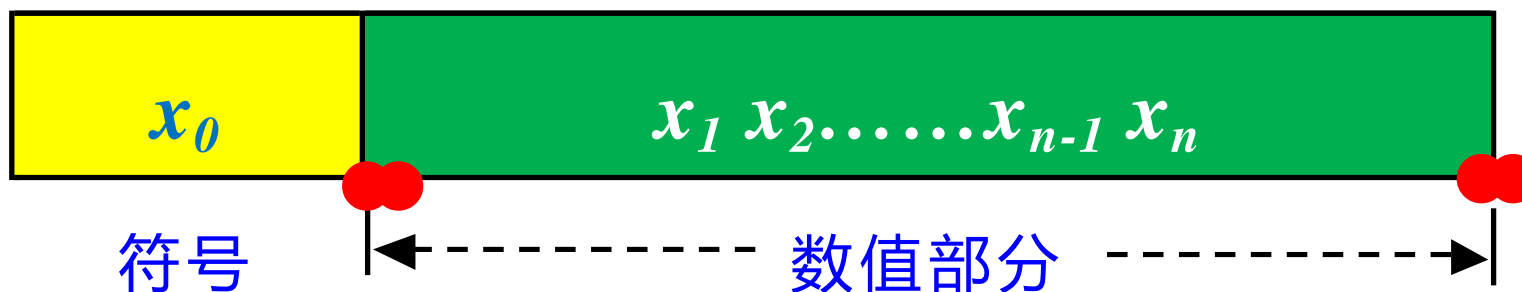


But 怎样表示定点数？



定点数的格式

定点数 $t = t_0 t_1 t_2 \dots t_n$ 在计算机中表示



➤ **定整数:** 小数点固定在最低位右边

表示范围为: $0 \leq |x| \leq 2^n - 1$

➤ **定小数:** 小数点固定在数值部分最高位左边(在 2^0 与 2^{-1} 之间) 表示范围为: $0 \leq |x| \leq 1 - 2^{-n}$



C语言整数数据类型

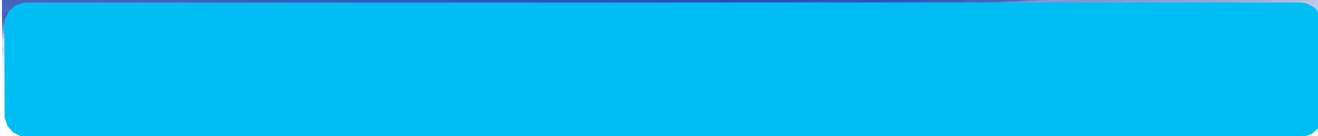
C 声明	MIPS 机器 最	
	小 值	最大 值
char	-128	127
unsigned char	0	255
short int	-32768	32767
unsigned short int	0	65535
int	-2^{31}	$2^{31} - 1$
unsigned int	0	$2^{32} - 1$
long int	-2^{31}	$2^{31} - 1$
unsigned long int	0	$2^{32} - 1$



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！！！！



$X = +0$ (十进制真值)
用八位二进制表示 原
码 = $0\ 0000000$

$X = -0$ (十进制真值)
用八位二进制表示 原
码 = $1\ 0000000$



不

0的补码表
是统一的



2.4.3 数值数据的浮点表示



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

$$\begin{aligned} 25.75 &= 2.575 \times 10^1 \text{ (十进制)} \\ &= 11001.11 \text{ (二进制真值)} \\ &= 1.100111 \times 10^{100} \text{ (二进制)} \end{aligned}$$



2.4.3 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)

\swarrow \searrow

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



科学计数法(Scientific Notation)



for Binary Numbers(二进制数):

mantissa(尾数) → 1.011_{two} × 2^{-10} ← *exponent(阶码)*

binary point (二进制小数点) → 1.011_{two} ← *基为2*

只要对尾数和指数分别编码，就可表示一个浮点数



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}





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



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 : $|f| > 1$

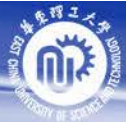
右移1位，阶码加1

11.01100111 $\times 10^{101}$
0.11011001 $\times 10^{111}$

➤➤左规 : $|f| < \frac{1}{2}$

左移1位，阶码减1

0.00110011 $\times 10^{101}$
0.11001100 $\times 10^{011}$



Prof William Kahan

1985年制定了浮点数标准IEEE 754

符号s	阶码e(整数)	尾数f(小数)
-----	---------	---------

➤ 符号s:

□ 1 表示负数 ; 0表示 正数



Prof Kahan

1985年制定了浮点数标准IEEE 754

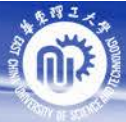
符号s	阶码e(整数)	尾数f(小数)
-----	---------	---------

➤ 尾数f

尾数 = 1 + significand
 $0 < \text{significand} < 1$

- 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)

尾数精度 = 尾数的位数 + 1



Prof Kahan

1985年制定了浮点数标准IEEE 754

符号s	阶码e(整数)	尾数f(小数)
-----	---------	---------

➤ 阶码/指数e : 移码

 将每一个数值加上一个偏置常数(Excess / bias)

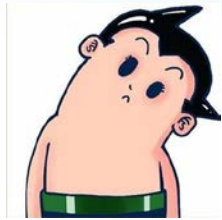
$$-8 \ (+8) \sim 0000_2$$

0 (+8) ~ 1000₂

...

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

便于浮点数加减运算时
进行对阶操作



*But*什么是“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) ~ 1000₂

...

-7 (+8) ~ 0001₂

...

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

为什么要用移码来表示指数(阶码)?

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

 将每一个数值加上一个偏置常数(Excess / bias)

为什么要用移码来表示指数(阶码)?

COMPUTER PRINCIPLE



Prof Kahan

1985年制定了浮点数标准IEEE 754

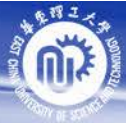
符号s	阶码e(整数)	尾数f(小数)
-----	---------	---------

➤ 阶码/指数e：移码

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}

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

全0/全1编码用来表示特殊的值！



Prof Kahan

1985年制定了浮点数标准IEEE 754

符号s	阶码e(整数)	尾数f(小数)
-----	---------	---------

➤ 尾数f

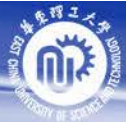
- 规格化尾数最高位总是1，所以隐含表示，省1位
- 1 + 23 bits (single单精度), 1 + 52 bits (double双精度)

➤ 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: $(-1)^s \times (1 + f) \times 2^{(\text{Exponent}-1023)}$



Prof Kahan

1985年制定了浮点数标准IEEE 754

符号s	阶码e(整数)	尾数f(小数)
-----	---------	---------

➤ 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)



➤ 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



练习1：将二进制浮点表示转换成十进制数

例1：BEE00000H：一个IEEE 754 浮点数的十六进制表示

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

单精度SP

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

符号：1 => negative 阶码

:

• $0111\ 1101_{\text{two}} = 125_{10}$

• Bias adjustment: $125 - 127 = -2$ 尾

数：

$1 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-5} + \dots$

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

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

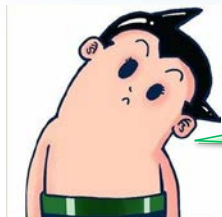


练习2：将十进制数转换成单精度浮点表示： -1.275×10^1

1. 计算真值： -12.75
2. 整数部分的转换 (Convert integer part) : $12 = 8 + 4 = 1100_2$
3. 小数部分的转换 (Convert fractional part) :
 $0.75 = 0.5 + 0.25 = 0.11_2$
4. 规格化 (Put parts together and normalize) : $1100.11 = 1.10011 \times 2^{011}$
5. 移码表示的阶码 (Convert exponent) :
 $127 + 3 = 10000010_2$

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

十六进制表示： **C14C0000H**



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 (隐含小数点前为“1”)	规格化数
0	0	?
0	非零的数	?
255	0	?
255	非零的数	?



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

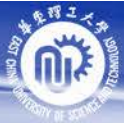
$+\infty$: 0 11111111 000000000000000000000000

$-\infty$: 1 11111111 000000000000000000000000



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
255	0	$+\infty/-\infty$
255	非零的数	非数NaN(Not a Number)



IEEE754 标准定义的浮点格式参数

*尾数精度 = 尾数位数 + 1

参数	单精度	扩充单精度	双精度	扩充双精度
总位数	32	≥43	64	≥79
阶码位数	8	≥11	11	≥15
阶码编码	+127 移码	(未定义)	+1023 移码	(未定义)
最大阶码值	+127	+1023	+1023	+16383
最小阶码值	-126	-1022	-1022	-10382
可表示的阶码个数	254	(未定义)	2046	(未定义)
尾数位数*	23	≥31	52	≥63
可表示的尾数个数	2^{23}	(未定义)	2^{52}	(未定义)
可表示的数据总数	1.98×2^{31}	(未定义)	1.99×2^{63}	(未定义)
数值表示范围 (十进制)	$10^{-38}, 10^{+38}$	(未定义)	$10^{-308}, 10^{+308}$	(未定义)



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

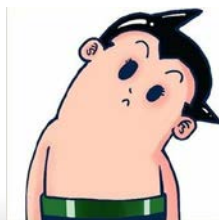
➤BCD码



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？



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



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



计算机为什么要用十进制数表示数值？

- 人们习惯用十进制数
- 某些系统为了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



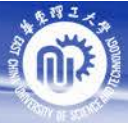
计算机为什么要用十进制数表示数值？

- 人们习惯用十进制数
- 某些系统为了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



用BCD码表示十进制数



➤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

➤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)



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)



2.4.5 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



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



汉字及国际字符编码表示



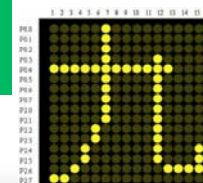
特点

- 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

- 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





2.4.6 数据的度量与存储



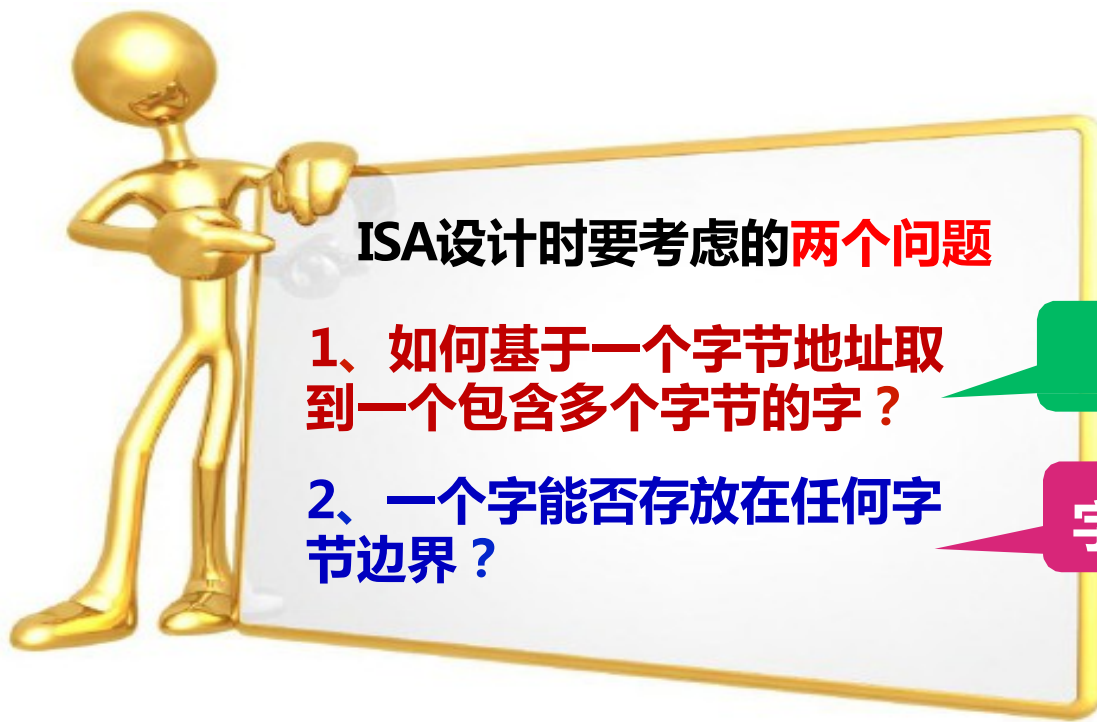
数据的度量单位

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	$1\text{KB}=2^{10}\text{字节}=1024\text{B}$	$1\text{KB}=10^3\text{字节}=1000\text{B}$
兆字节	MB	$1\text{MB}=2^{20}\text{字节}=1024\text{KB}$	$1\text{MB}=10^6\text{字节}=1000\text{KB}$
千兆字节	GB	$1\text{GB}=2^{30}\text{字节}=1024\text{MB}$	$1\text{GB}=10^9\text{字节}=1000\text{MB}$
兆兆字节	TB	$1\text{TB}=2^{40}\text{字节}=1024\text{GB}$	$1\text{TB}=10^{12}\text{字节}=1000\text{GB}$

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**



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？
- 2、一个字能否存放在任何字节边界？

字的存放问题

字的边界对齐问题



数据存放

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。

Word:	01	23	45	67	little endian word 100
	103	102	101	100	
	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，需要通过特定的控制位来设定



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

➤任何像音频、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



数据对齐



➤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

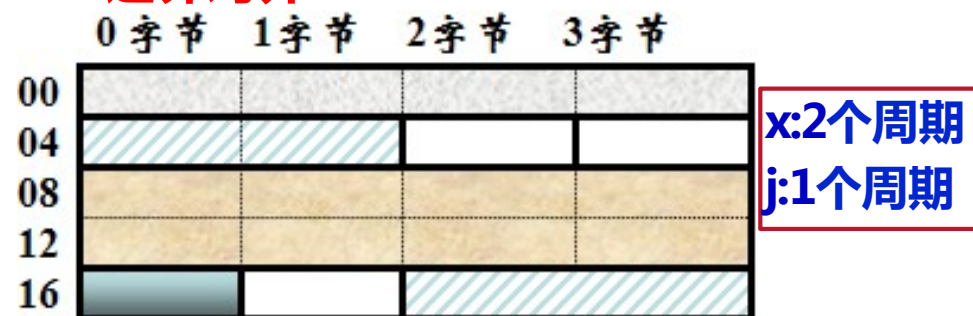
不同长度的数据存放时, 有两种处理方式



数据对齐

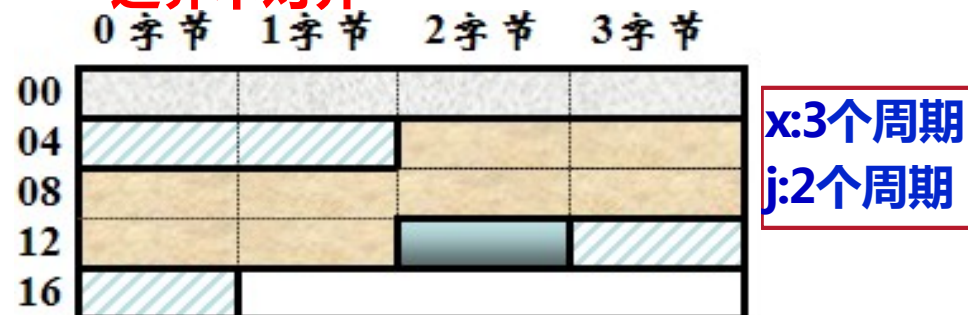
例：假设数据顺序：字-半字-双字-字节-半字-.....如：
`int i, short k, double x, char c, short j,.....`

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！！！！