



# 计算机组织与体系结构

Computer Architectures

陆俊林



## 第二讲 计算机的基本结构

### 本讲要点

以个人计算机为主要关注点，简要分析“冯·诺依曼结构”的具体内容，及其与当前个人计算机内部结构的对应关系，然后从x86指令系统和地址空间入手分析其特点，最后讲解Intel格式的x86汇编语言，为后续学习做准备。



# 主要内容

通过学习本课程  
了解计算机的发展历程，理解计算机的组成原理，掌握计算机的设计方法



## I 冯·诺依曼计算机结构

## II x86指令系统概览

## III x86的地址空间

## IV x86汇编语言的格式

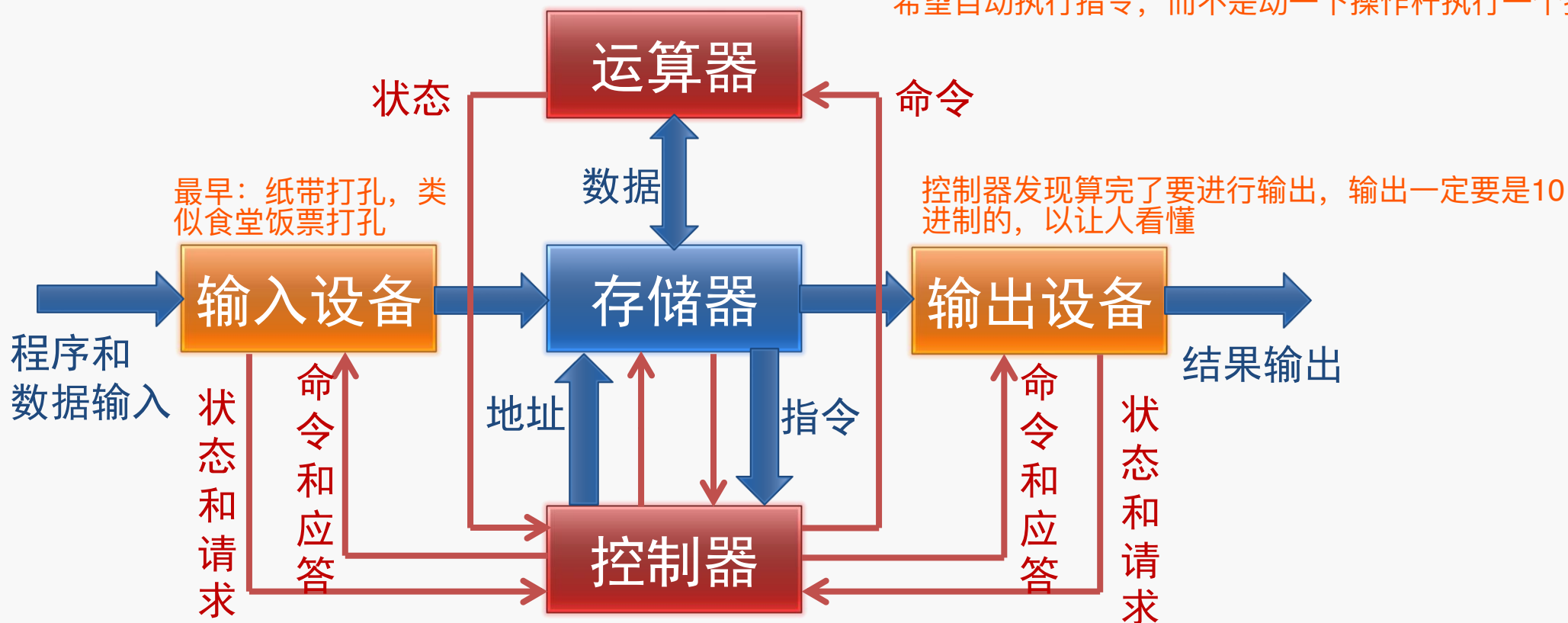


# 冯·诺依曼结构的要点

更早的计算机：使用操作杆表达指令，类似遥控器，重新编程要插拔线

- ① 计算机应由运算器、控制器、存储器、输入设备和输出设备共 **5个部分** 组成
- ② **数据和程序** 均以 **二进制代码** 形式不加区别地存放在存储器中，存放位置由存储器的地址指定 哈佛结构：存储器分为指令存储器和数据存储器
- ③ 计算机在工作时能够 **自动** 地从存储器中 **取出指令** 加以执行
- ④ .....

希望自动执行指令，而不是动一下操作杆执行一个指令



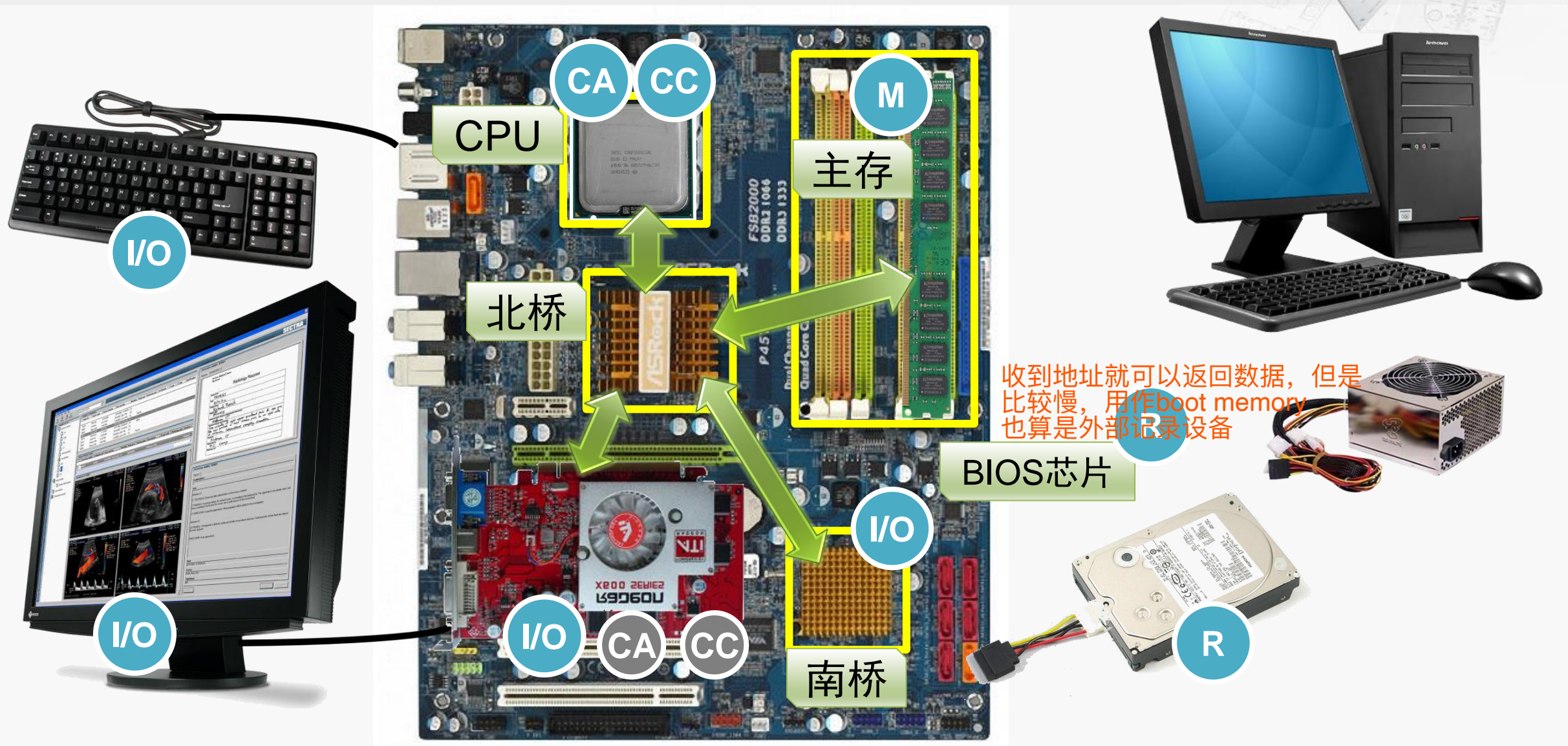
# 冯·诺依曼计算机的主要构成



- ① 运算器, CA: central arithmetical
- ② 控制器, CC: central control
- ③ 存储器, M: memory
- ④ 输入设备, I: input
- ⑤ 输出设备, O: output
- 🕒 外部记录设备, R: outside recording medium



# 冯·诺依曼结构原理与实现的对应



# 汇编语言

- ❏ 较低级的程序设计语言，主要是机器语言的符号化描述
- ❏ 通常为特定计算机或计算机系列专门设计

## x86汇编语言示例

```
MOV    AX,    0H
MOV    BX,    [20H]
ADD    AX,    BX
ADD    BX,    1H
JMP    label_next
```

## ARM汇编语言示例

```
MOV    R0,    #0
LDR    R2,    #0x10020 load
ADD    R0,    R0,    R2
ADD    R2,    R2,    #1
B      label_next branch
```

ICS: AT-T格式 (Linux), 这里的是Intel格式(Windows)

POWER

MIPS

SPARC

Alpha

# helloworld程序：从C语言到机器语言

标准输入输出函数库

函数printf在标准输入输出函数库中定义

```
#include <stdio.h>
```

C语言

```
int main ( )
```

```
{
```

```
    printf ( "hello, world\n" ) ;
```

```
    return 0;
```

```
}
```

```
.data
msg:      db "hello, world",10
len:      equ $-msg
```

```
...
```

```
main:
```

```
    mov     edx,len
    mov     ecx,msg
    mov     ebx,1
    mov     eax,4
```

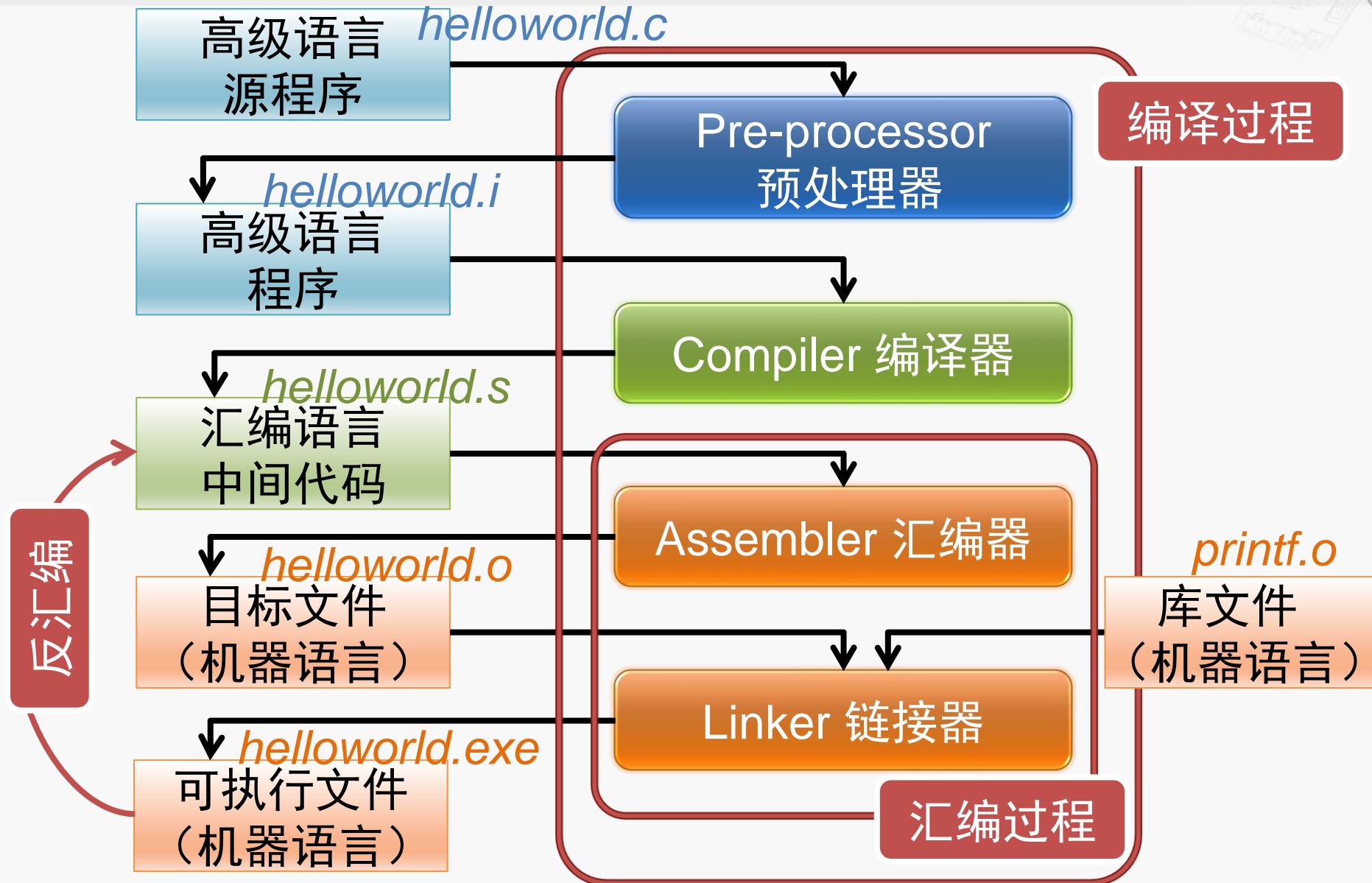
汇编语言

00000000	7F	45	4C	46	01	01	01	00	00	00	00	00	00	00	00
00000010	02	00	03	00	01	00	00	00	80	80	04	08	34	00	00
00000020	F8	00	00	00	00	00	00	00	34	00	20	00	02	00	28
00000030	05	00	04	00	01	00	00	00	00	00	00	00	00	80	04
00000040	00	80	04	08	A2	00	00	00	A2	00	00	00	05	00	00
00000050	00	10	00	00	01	00	00	00	A4	00	00	00	A4	90	04
00000060	A4	90	04	08	09	00	00	00	09	00	00	00	06	00	00
00000070	00	10	00	00	00	00	00	00	00	00	00	00	00	00	00
00000080	BA	09	00	00	00	B9	A4	90	00	00	00	00	00	00	B8
00000090	04	00	00	00	CD	80	BB	00	00	00	00	00	00	00	00
000000A0	CD	80	00	00	48	69	20	57	6F	72	6C	64	0A	00	00

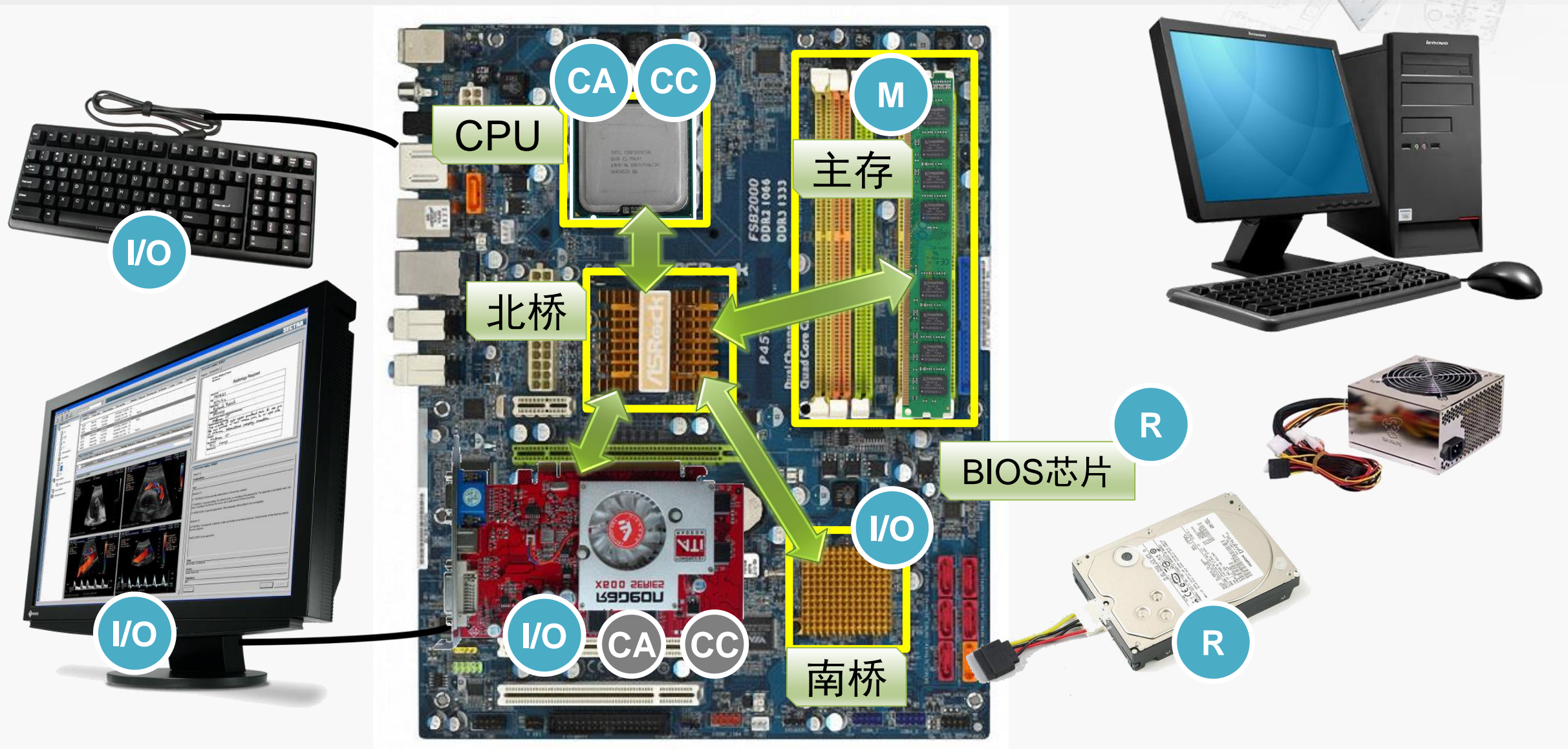
机器语言



# 高级语言、汇编语言与机器语言



# 如何在计算机上运行一个程序？



# 主要内容

通过学习本课程  
了解计算机的发展历程，理解计算机的组成原理，掌握计算机的设计方法

## I 冯·诺依曼计算机结构



## II x86指令系统概览

## III x86的地址空间

## IV x86汇编语言的格式



# x86体系结构

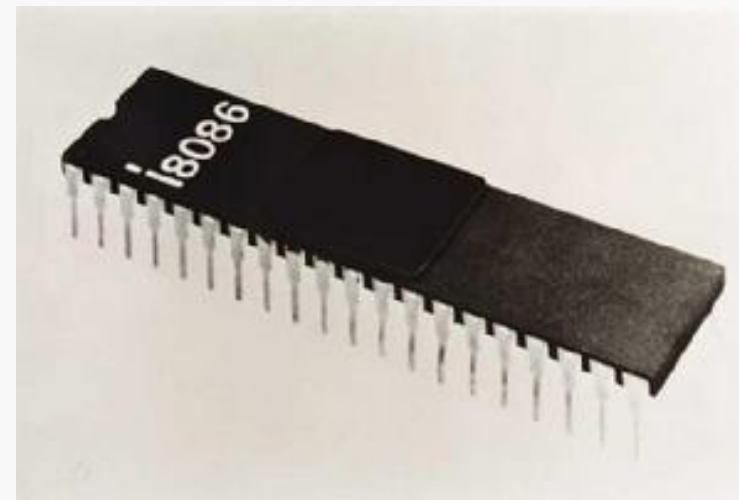
体系结构		厂商	微处理器型号	字长	年代
x86	“x86-16” “IA-16”	Intel	<b>8086</b> , 8088, 80186, 80188 80286	16位	1978年起
	IA-32	Intel	<b>80386</b> , 80486, Pentium, Pentium Pro/II/III/4, Core, Atom	32位	1985年起
		AMD	Am386, Am486, AM5x86, K5, K6, Athlon		
		Others	<b>Cyrix</b> 5x86; <b>VIA</b> C3/C7 <b>Transmeta</b> Crusoe, Efficeon		
	x86-64	AMD	<b>Opteron</b> , Athlon 64 Phenom, Phenom II	64位	2003年起
		Intel	Pentium 4 Prescott, Core 2 Core i3/i5/i7		
		Others	<b>VIA</b> Nano		



# Intel 8086 (1978年)

## 8086的主要特点

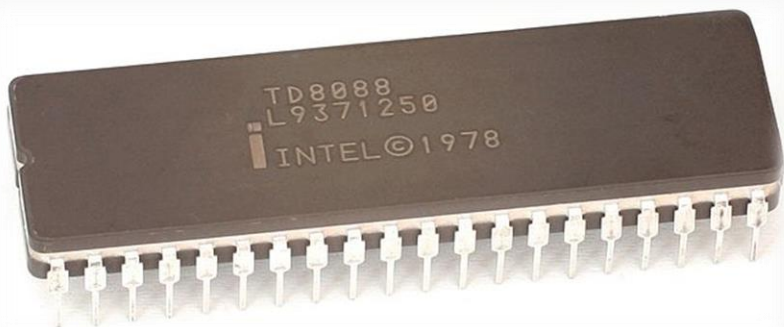
- ① 内部的通用寄存器为16位  
既能处理16位数据，也能处理8位数据
- ③ 对外有16根数据线和20根地址线  
可寻址的内存空间为1MByte ( $2^{20}$ )
- ③ 物理地址的形成采用“段加偏移”的方式



# 微型计算机的早期代表：IBM PC

## 🕒 1981年，IBM PC 5150诞生

- 售价约1600美元
- Intel 8088 CPU，主频4.77MHz，内存16KB
- 因开放性架构逐渐成为个人计算机的制造标准



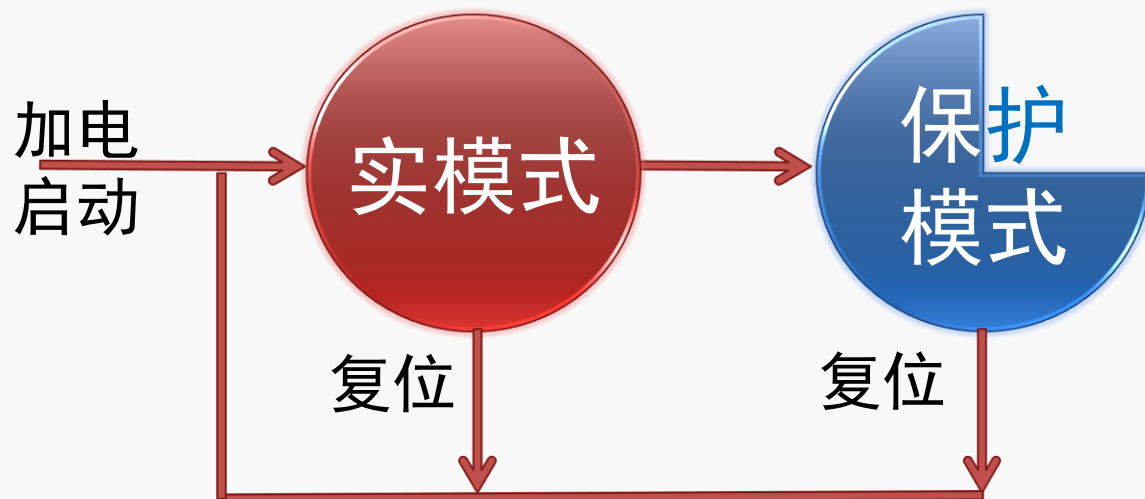
Intel 8088 CPU，1979年推出。  
8088是8086的简化版本，主要区别是数据总线只有8位宽。



# Intel 80286 (1982年)

## 80286的主要特点

- 地址总线扩展到24位，可寻址16MB的内存空间
- 引入了“保护模式”，但是机制有缺陷
  - \*例如，每个段仍为64KB，严重限制软件规模
- 为保持兼容，保留了8086的工作模式，被称为“实模式”



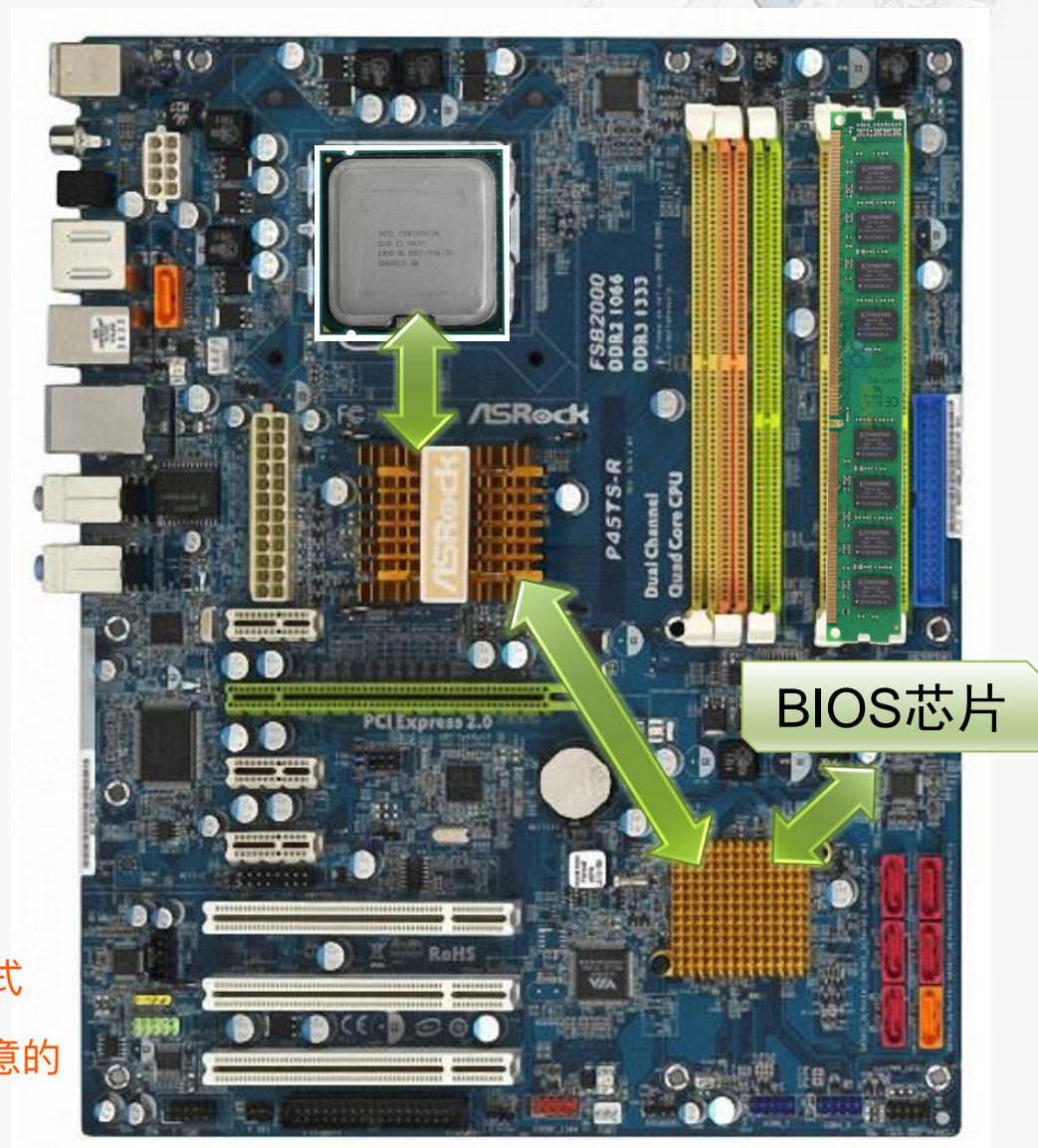
80286  
主频6~20MHz  
13.4万个晶体管



# 实模式 (Real Mode)

- 实模式，又称“实地址模式”
  - 80286及以上的微处理器采用8086的工作模式，即为**实模式**
  - 运行在实模式下的80x86微处理器像是一个更快的8086
  - 为兼容8086，所有x86处理器在加电或复位后首先进入实模式
  - 系统初始化程序在实模式下运行，为进入保护模式做好准备

内存寻址限制：在实模式下，处理器只能访问1MB的内存空间。这是因为在实模式下，地址线是20位的，能表示的最大地址空间是 $(2^{20} = 1\text{MB})$ 。  
没有内存保护：实模式不提供任何形式的内存保护机制，任何程序都可以访问任意的内存地址，这使得系统非常容易受到恶意软件的影响。





# x86体系结构

体系结构		厂商	微处理器型号	字长	年代
x86	“x86-16” “IA-16”	Intel	<b>8086</b> , 8088, 80186, 80188 80286	16位	1978年起
	IA-32	Intel	<b>80386</b> , 80486, Pentium, Pentium Pro/II/III/4, Core, Atom	32位	1985年起
		AMD	Am386, Am486, AM5x86, K5, K6, Athlon		
		Others	<b>Cyrix</b> 5x86; <b>VIA</b> C3/C7 <b>Transmeta</b> Crusoe, Efficeon		
	x86-64	AMD	<u>Opteron</u> , Athlon 64 Phenom, Phenom II	64位	2003年起
		Intel	Pentium 4 Prescott, Core 2 Core i3/i5/i7		
		Others	<b>VIA</b> Nano		

# Intel 80386 (1985年)

当时个人计算机不高端



## 80386的主要特点

- 80x86系列中的第一款32位微处理器
- 地址总线扩展到32位，可寻址4GB的内存空间
- 改进了“保护模式”（例如，段范围可达4GB）
- 增加了“虚拟8086模式”，可以同时模拟多个8086微处理器

实模式

保护  
模式

虚拟  
8086  
模式



80386

主频12.5~33MHz  
27.5万个晶体管

# 保护模式（Protected Mode）



- 🔍 保护模式，可简写为“pmode”
  - 80386及以上的微处理器的主要工作模式
  - 支持多任务
  - 支持设置特权级
  - 支持特权指令的执行
  - 支持访问权限检查
  - 可以访问4GB的物理存储空间
  - 引入了虚拟存储器的概念

保护模式让操作系统加强了对应用软件的控制，使得系统运行更安全高效

# 虚拟8086模式（Virtual 8086 Mode）



- ④ 虚拟8086模式，又称“V86模式”
  - V86模式实际上是保护模式下一种特殊工作状态
  - V86模式下的微处理器类似于8086，但不等同
  
- ④ V86模式 与 实模式 的比较
  - 相同点
    - 可寻址的内存空间为1MB
    - “段加偏移”的寻址方式
  - 不同点
    - 对中断/异常的响应处理

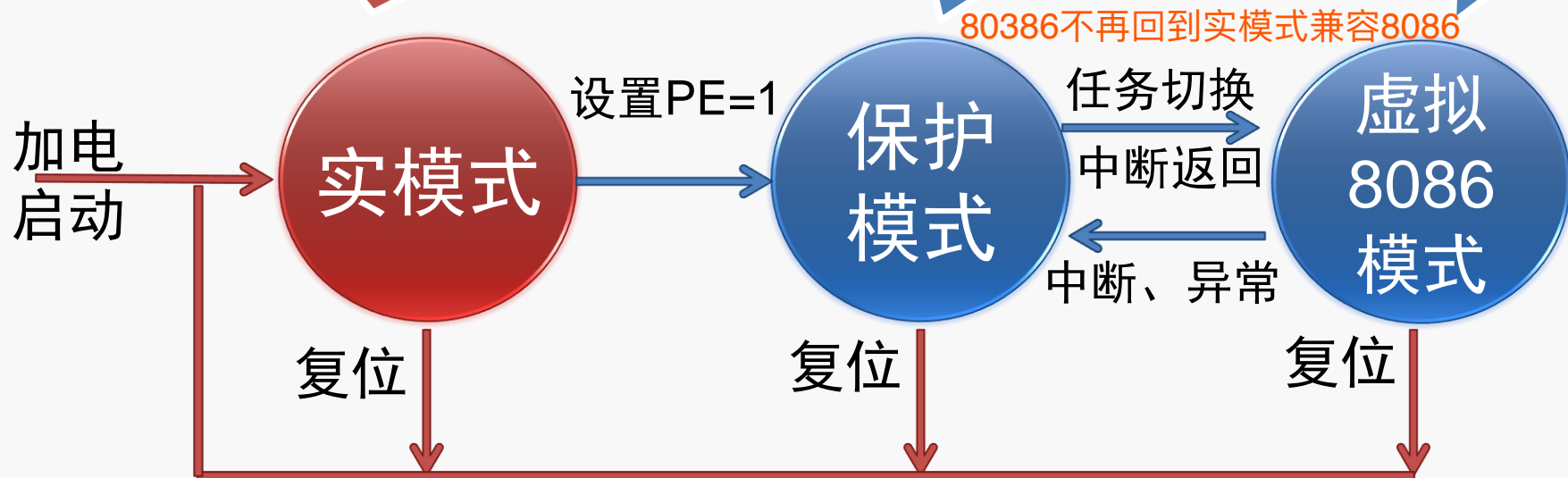


# 三种工作模式之间的转换

从加电启动或复位到操作系统运行之前

操作系统和应用程序的运行

运行兼容8086程序



\*注：PE即“保护模式允许”，是80x86控制寄存器CR0中的控制位

# x86体系结构

体系结构		厂商	微处理器型号	字长	年代
x86	“x86-16” “IA-16”	Intel	<b>8086</b> , 8088, 80186, 80188 80286	16位	1978年起
	IA-32	Intel	<b>80386</b> , 80486, Pentium, Pentium Pro/II/III/4, Core, Atom	32位	1985年起
		AMD	Am386, Am486, AM5x86, K5, K6, Athlon		
		Others	<b>Cyrix</b> 5x86; <b>VIA</b> C3/C7 <b>Transmeta</b> Crusoe, Efficeon		
	x86-64	AMD	<b>Opteron</b> , Athlon 64 Phenom, Phenom II	64位	2003年起
		Intel	Pentium 4 Prescott, Core 2 Core i3/i5/i7		
		Others	<b>VIA</b> Nano		

注： Intel提出的IA-64是独立于x86的一种新的体系结构，不兼容IA-32 因为不能兼容后来被放弃了

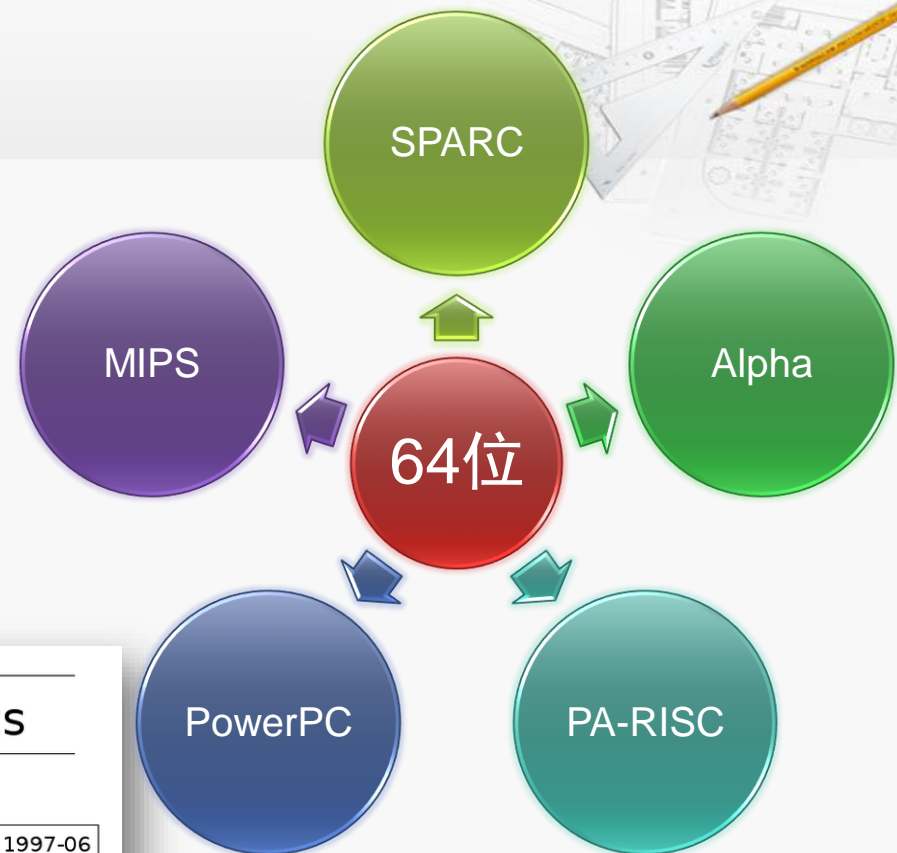
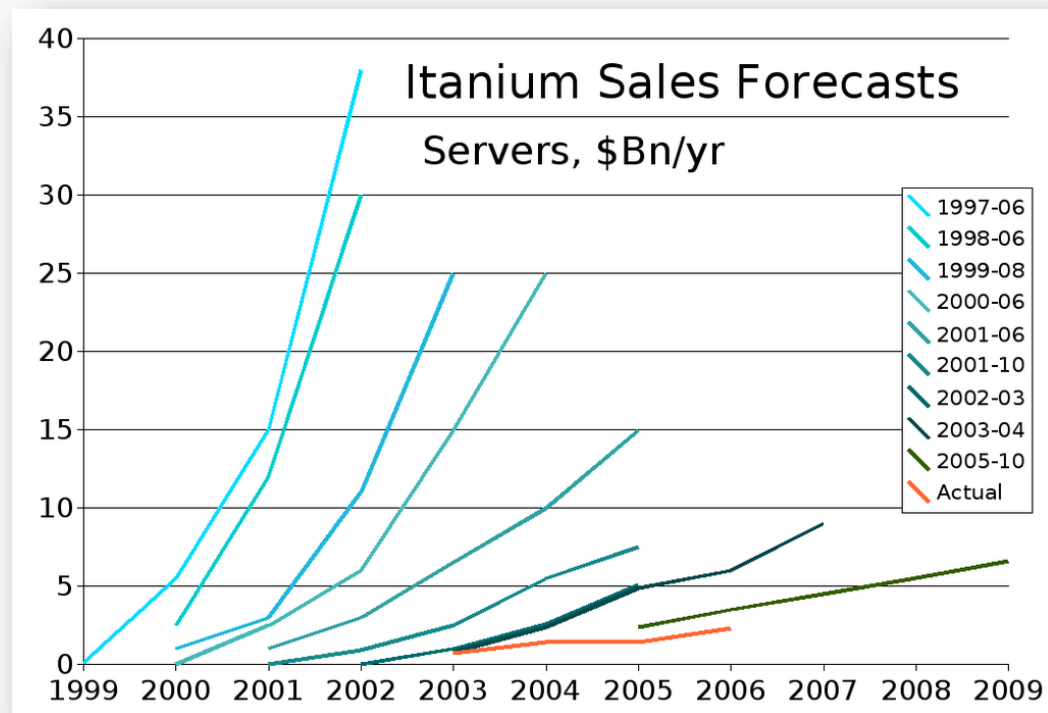
# AMD Opteron (2003年)

## Opteron的主要特点

- x86扩展到64位的第一款微处理器
- 可以访问高于4GB的存储器
- 兼容32位x86程序，且不降低性能



Opteron  
主频1.4~3.5GHz  
工艺130~32nm



Intel Itanium

# x86-64的运行模式

兼容

运行模式	运行子模式	操作系统	已有程序的支持
长模式 Long mode	64位模式 64-bit mode	64位	需重新编译
	兼容模式 Compatibility mode	64位	不需要重新编译
传统模式 Legacy mode	保护模式 Protected mode	32位或16位	不需要重新编译
	虚拟8086模式 Virtual 8086 mode	32位或16位	不需要重新编译
	实模式 Real mode	16位	不需要重新编译



# 主要内容

通过学习本课程  
了解计算机的发展历程，理解计算机的组成原理，掌握计算机的设计方法

I 冯·诺依曼计算机结构

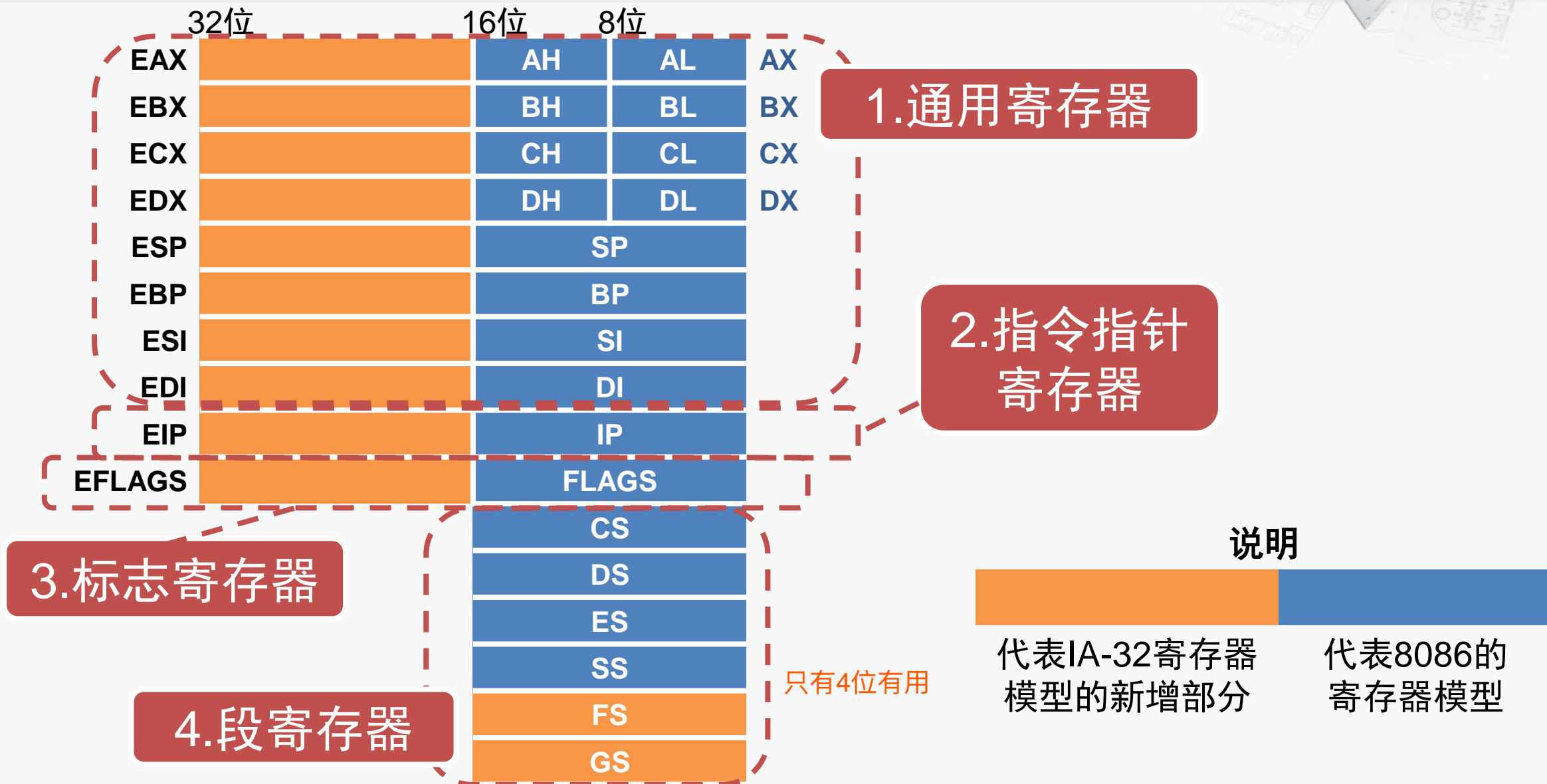
II x86指令系统概览

III x86的地址空间

IV x86汇编语言的格式



# IA-32和8086的寄存器模型





# 8086的指令指针寄存器

## 指令指针寄存器 IP (Instruction Pointer)

- 保存一个内存地址，指向当前需要取出的指令
- 当CPU从内存中取出一个指令后，IP会自动增加，指向下一指令的地址（注：实际情况会复杂的多）
- 程序员不能直接对IP进行存取操作
- 转移指令、过程调用/返回指令等会改变IP的内容

IP寄存器的寻址能力：  
 $2^{16}=65536(64K)$ 字节单元

最多64K代码

8086对外有20位地址线  
寻址范围： $2^{20}=1M$ 字节单元

最多1m数据

16位	8位	
AH	AL	AX
BH	BL	BX
CH	CL	CX
DH	DL	DX
SP		
BP		
SI		
DI		
IP		
FLAGS		
CS		
DS		
ES		
SS		

# 段寄存器的说明

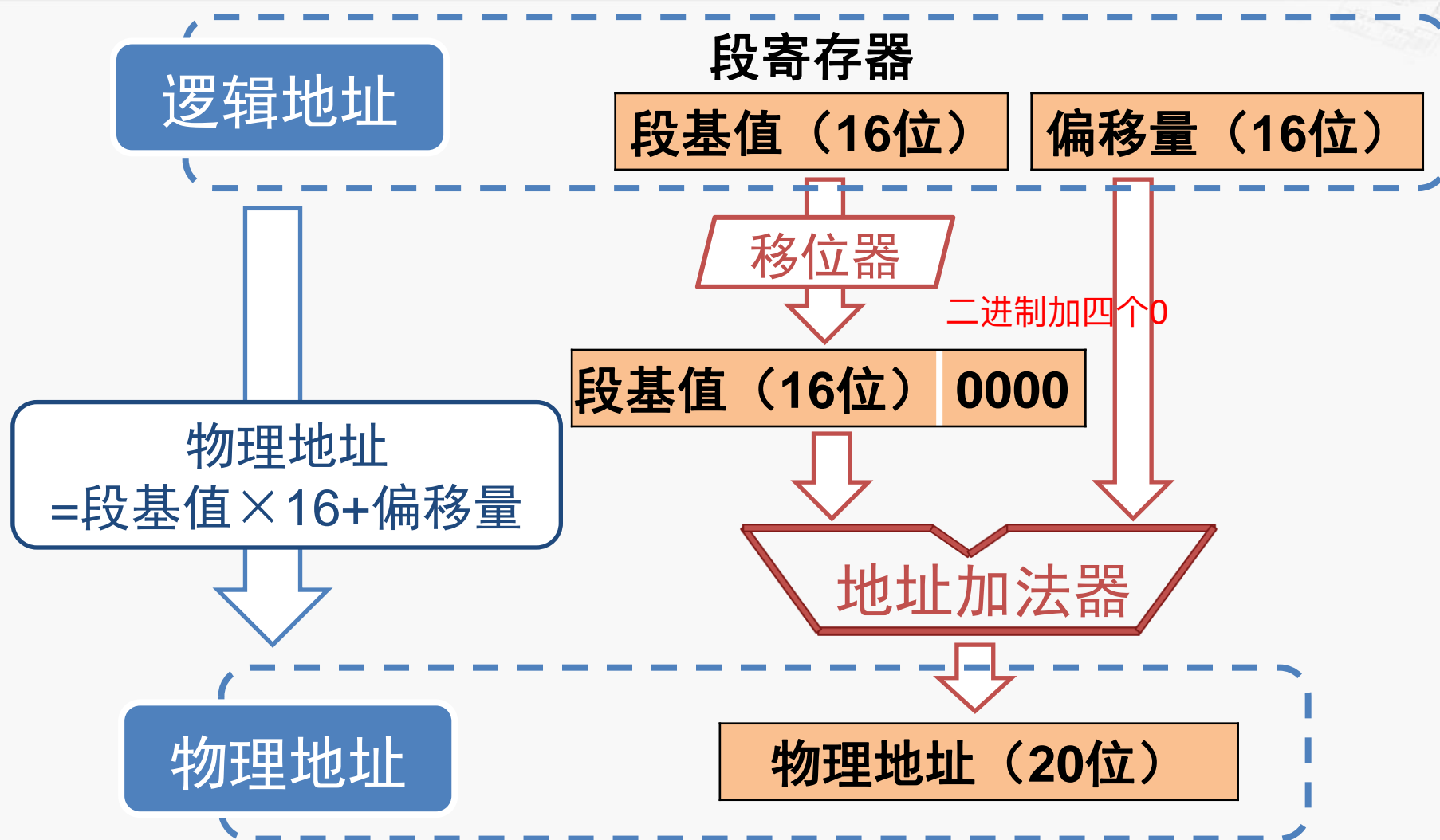


## ▶ 段寄存器

- 与微处理器中其他寄存器联合生成存储器地址
- 段寄存器的功能在实模式下和保护模式下是不相同的

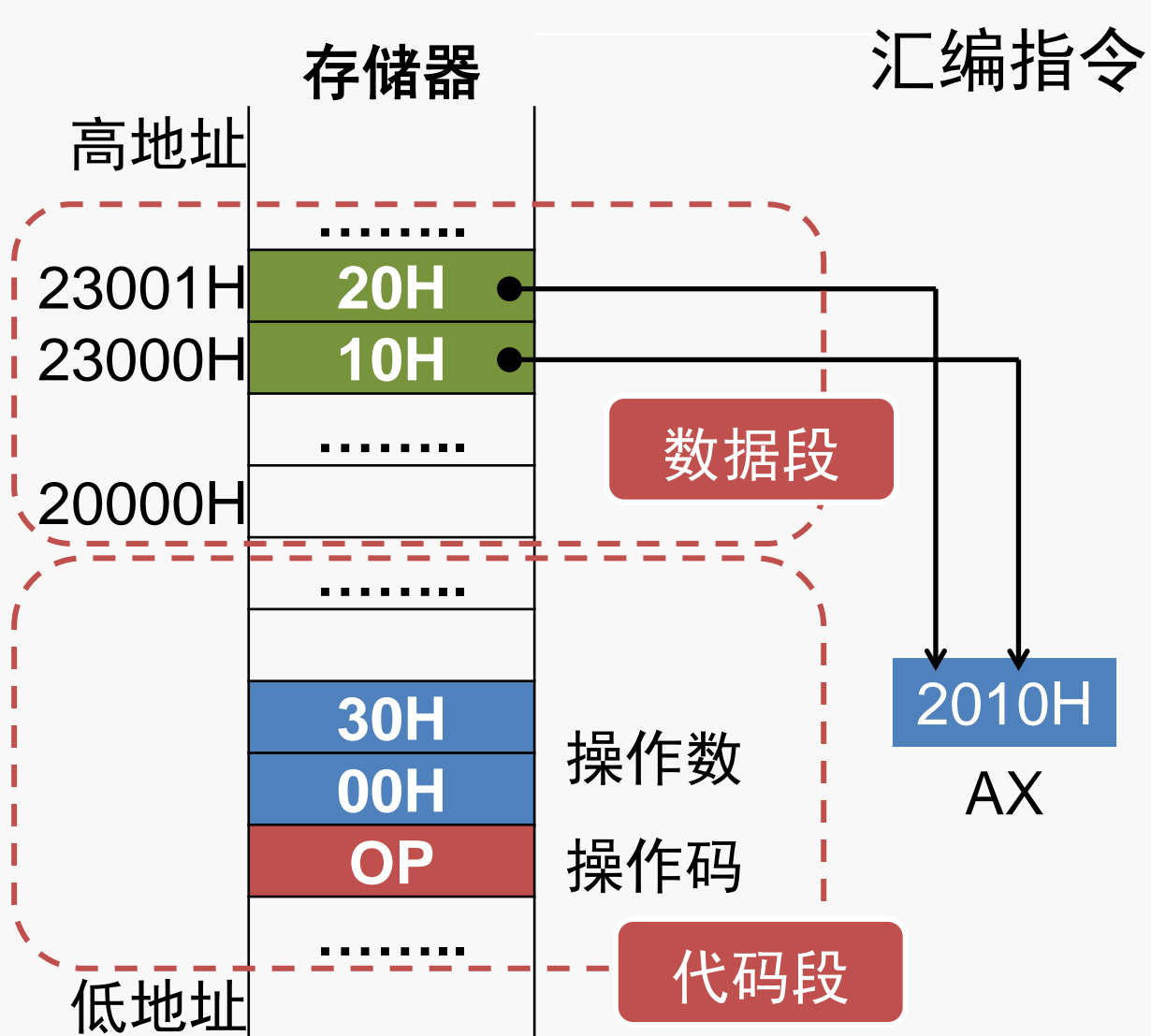
CS	代码段寄存器 (Code Segment)
DS	数据段寄存器 (Data Segment)
ES	附加段寄存器 (Extra Segment)
SS	堆栈段寄存器 (Stack Segment)
FS	80386新增的附加段寄存器
GS	80386新增的附加段寄存器

# 8086的物理地址生成





# “段加偏移”的编程实例



操作数默认存放在DS指向的数据段中，即  
[3000H]=DS:[3000H]

设：DS=2000H，  
则：物理地址  
=2000H×16+3000H  
=23000H

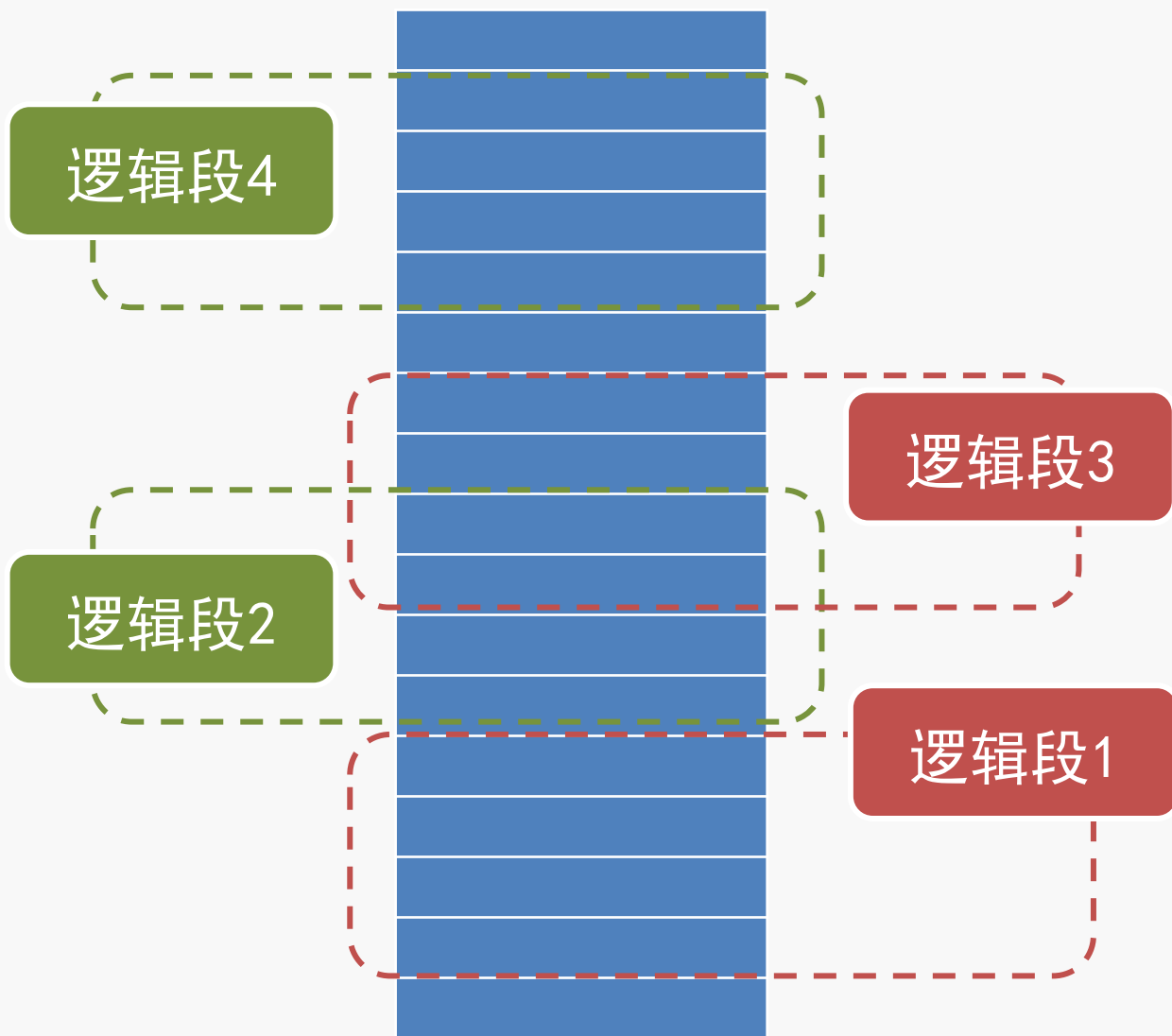
# 直观的存储器分段方法

- ④ 实现方法：将20位物理地址分成两部分
  - 高4位为段号，用“段号寄存器”来保存
  - 低16位为段内地址，也称“偏移地址”
- ④ 不足之处
  - 段号寄存器与其他寄存器不兼容，操作麻烦
  - 每个逻辑段固定占用64K字节，会浪费存储空间
- ④ 相比之下，8086的分段技术更为灵活



# 逻辑段在物理存储器中的位置

## 存储器



1M字节的存储空间分成许多逻辑段，每段最长64K字节，可以用16位地址进行寻址

编程时使用逻辑地址，不需要知道代码或数据在存储器中的具体物理位置，从而简化存储资源的管理

各个逻辑段在实际存储空间中可以完全分开，也可以部分重叠，甚至完全重叠

# (1) 代码段寄存器CS



## ❏ 代码段

- 一个存储区域，用以保存微处理器使用的代码（程序或过程）

## ❏ CS（Code Segment）

- 保存了代码段的起始地址
- 用CS:IP指示下一条要执行的指令地址

EIP/IP	Instruction Pointer	指令指针寄存器
--------	---------------------	---------



## (2) 数据段寄存器DS



### 数据段

- 一个存储区域，包含程序所使用的大部分数据

### DS (Data Segment)

- 保存了数据段的起始地址

### 实模式

16

- 数据段的长度限制为64KB

### 保护模式

32

- 数据段长度限制为4GB

**DATA SEGMENT ; 数据段**

NUM DW 0011101000000111B

NOTES DB 'The result is :', '\$'

**DATA ENDS ; 数据段结束**

**STACK SEGMENT ; 堆栈段**

STA DB 50 DUP(?)

TOP EQU LENGTH STA

**STACK ENDS ; 堆栈段结束**

**CODE SEGMENT ; 代码段**

ASSUME CS:CODE, DS:DATA, SS:STACK

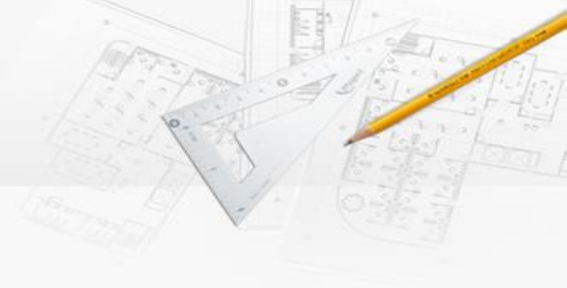
**BEGIN:**

MOV AX, DATA

MOV DS, AX ; 为DS赋初值

...

### (3) 附加段寄存器ES



#### 附加段

- 附加的数据段，也用于数据的保存
- 某些串操作指令将附加段作为其目的操作数的存放区域
- 长度限制与代码段及数据段相同

#### ES (Extra Segment)

- 保存了附加段的起始地址
- 用ES:DI指示串操作的目的操作数的地址

EDI/DI	destination index	目的变址寄存器
--------	-------------------	---------

# 段跨越前缀

- 如果数据存放在数据段以外的其它段（例如附加段），则应在指令中给出“段跨越前缀”

- 示例1:       MOV   AX, ES:[3000H]

- 示例2: ES: MOV   AX, [3000H]

## (4) 堆栈段寄存器SS

### 🔍 堆栈段

- 存储器中的一个特殊存储区
- 用以暂时存放程序运行中的一些数据和地址信息

### 🔍 SS (Stack Segment)

- 用以指示堆栈段的首地址
- ESP/SP 或 EBP/BP 指示堆栈顶的偏移地址
- 用SS:SP等组合操作堆栈中的数据

ESP	stack pointer	堆栈指针寄存器
EBP	(stack)base pointer	(堆栈) 基址指针寄存器

...

**DATA ENDS** ; 数据段结束

**STACK SEGMENT** ; 堆栈段

STA DB 50 DUP(?)

TOP EQU LENGTH STA

**STACK ENDS** ; 堆栈段结束

**CODE SEGMENT** ; 代码段

ASSUME CS:CODE, DS:DATA, SS:STACK

**BEGIN:**

MOV AX, DATA

MOV DS, AX ; 为DS赋初值

MOV AX, STACK

MOV SS, AX ; 为SS赋初值

MOV AX, TOP

MOV SP, AX ; 为SP赋初值

...



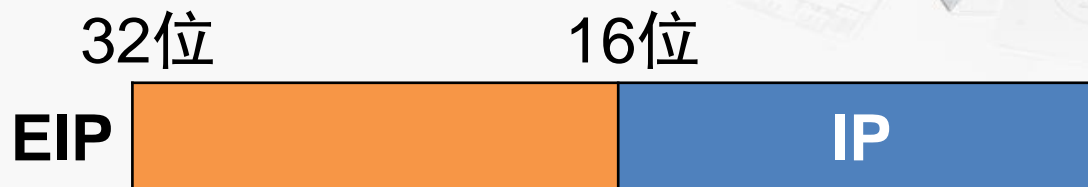
## (5) 新增的附加段寄存器FS和GS



- ④ 80386起新增了这两个附加段寄存器
- ④ FS和GS的功能和ES相同
- ④ 增加FS和GS可以减轻ES寄存器的负担，以便程序灵活访问相应的两个附加数据段

# IA-32的存储器寻址

以指令的寻址为例



## 实模式 CS:IP

所以实模式只能访问 $2^{20}=1\text{mb}$ 的内存

EIP寄存器的寻址能力：  
 $2^{32}=4\text{G}$ 字节单元

## 保护模式 CS:EIP

保护模式可以访问 $2^{32}=4\text{GB}$ 的内存，不是 $2^{48}$ 或 $2^{36}$ 因为寻址方式与实模式完全不同，增加了GDT来寻址

CS在保护模式变为段选择子代表在GDT中的偏移，16位是GDT总大小，GDT一项8字节，所以总共8192个表项

80386对外有32位地址线  
寻址范围： $2^{32}=4\text{G}$ 字节单元

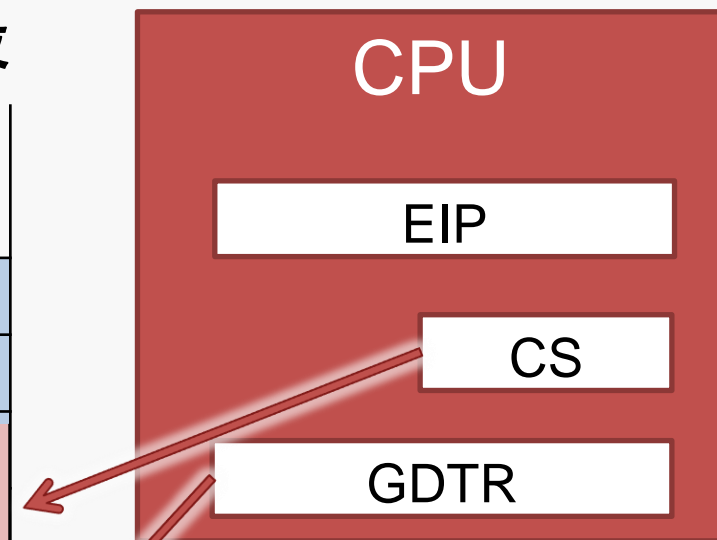
# IA-32的存储器寻址

<https://blog.csdn.net/l1004969690/article/details/113992196>

<https://www.kancloud.cn/digest/protectedmode/121465>

保护模式下，段基址不在CS中，而是在内存中  
存储器片段

高地址								
描述符8191								
描述符8190								
其中一个... 描述符→	字节7 基地址	字节6 其它	字节5 权限	字节4	字节3 基地址	字节2	字节1 段界限	字节0
描述符1				80286 24位地址总线				
描述符0								
低地址								



- GDT: 全局描述符表
- GDTR: 全局描述符表的地址寄存器
- GDT可在系统中的任何存储单元，通过GDTR定位



# x86-64的描述符

存储器片段

高地址								
描述符8191								
描述符8190								
其中一个... 描述符→	字节7 全为0	字节6 其它	字节5 权限	字节4	字节3 全为0	字节2	字节1 全为0	字节0
描述符1								
描述符0								
低地址								

注：描述符中没有了段基址和段界限，只有访问权限字节和若干控制位。所有的代码段都从地址0开始。



# 主要内容

通过学习本课程  
了解计算机的发展历程，理解计算机的组成原理，掌握计算机的设计方法

**I 冯·诺依曼计算机结构**

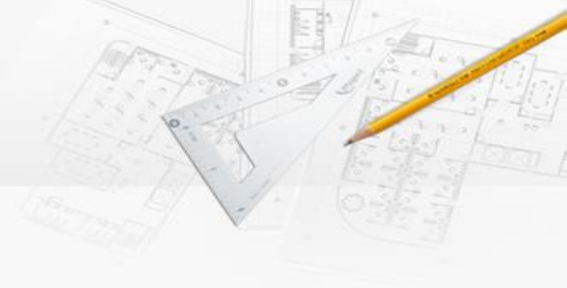
**II x86指令系统概览**

**III x86的地址空间**

**IV x86汇编语言的格式**



# Intel格式与AT&T格式



## Intel格式

- Intel制定，x86相关的文档手册使用该格式
- 主要应用在MS-DOS和Windows等系统中

## AT&T格式

- AT&T制定，起源于贝尔实验室研发的Unix
- 最初用在PDP-11/VAX等机型，后移植到x86
- 主要应用在Unix和Linux等系统中



# 区别1：前缀（后缀）

## Intel语法

- 寄存器和立即数都没有前缀
- 十六进制和二进制立即数后缀分别为h和b

## AT&T语法

- 寄存器使用前缀“%”，立即数使用前缀“\$”
- 十六进制立即数使用前缀“0x”

Intel语法	AT&T语法
<code>mov eax, 8</code>	<code>movl \$8, %eax</code>
<code>mov ebx, 0ffffh</code>	<code>movl \$0xffff, %ebx</code>
<code>int 80h</code>	<code>int \$0x80</code>

# Intel格式中数的表示



## ④ 整数

- 默认十进制，非默认基数的用字母后缀标明

**B**: 二进制 Binary

**D**: 十进制 Decimal

**H**: 十六进制 Hexadecimal

**O**或**Q**: 八进制 Octal

- 示例: 1011**B**, 35**D**, 6A**H**, 17**Q**
- 以字母开头的十六进制数必须加0
  - 示例: FEH→**0**FEH
- 字符串常数用单引号括起
  - 示例: 'The result is:'

## ④ 实数: $5.213 \times 10^{-6} \rightarrow 5.213\text{E-}6$

## 区别2：操作数方向

### Intel语法

- 第一个操作数是目的操作数
- 第二个操作数是源操作数

### AT&T语法

- 第一个数是源操作数
- 第二个数是目的操作数

Intel语法	AT&T语法
<code>mov eax, [ecx]</code>	<code>movl (%ecx), %eax</code>



## 区别3：内存单元操作数

### Intel语法

- 基寄存器用 “ [ ] ” 标明

### AT&T语法

- 基寄存器用 “ ( ) ” 标明

Intel语法	AT&T语法
<code>mov eax, [ebx+5]</code>	<code>movl 5(%ebx), %eax</code>

## 区别4：间接寻址方式

### Intel语法

- `segreg:[base+index*scale+disp]`

### AT&T语法

- `%segreg:disp(base,index,scale)`

Intel语法	AT&T语法
<code>mov eax, [ebx+20h]</code>	<code>movl 0x20(%ebx), %eax</code>
<code>add eax, [ebx+ecx*2h]</code>	<code>addl (%ebx,%ecx,0x2), %eax</code>
<code>lea eax, [ebx+ecx]</code>	<code>leal (%ebx,%ecx), %eax</code>
<code>sub eax, [ebx+ecx*4h-20h]</code>	<code>subl -0x20(%ebx,%ecx,0x4), %eax</code>



# 区别5：操作码后缀

## AT&T语法

- 操作码带后缀，以指出操作数的大小
  - l: 32位/长整数; w: 字/16位; b: 字节/8位

## Intel语法

- 内存单元操作数带前缀，以指出操作数的大小
  - dword ptr; word ptr; byte ptr

Intel语法	AT&T语法
mov al,bl	movb %bl,%al
mov ax,bx	movw %bx,%ax
mov eax,ebx	movl %ebx,%eax
mov eax, dword ptr [ebx]	movl (%ebx),%eax

# 汇编程序示例

```
DATA SEGMENT ; 数据段
    NUM DW 00111010000000111B
    NOTES DB 'The result is :', '$'
DATA ENDS ; 数据段结束

STACK SEGMENT ; 堆栈段
    STA DB 50 DUP(?)
    TOP EQU LENGTH STA
STACK ENDS ; 堆栈段结束

CODE SEGMENT ; 代码段
    ASSUME CS:CODE, DS:DATA, SS:STACK
    BEGIN:
        MOV AX, DATA
        MOV DS, AX ; 为DS赋初值
```


注意：  
这是一个完整的汇编程序，不是嵌入到高级语言中的汇编代码片段。

# 汇编程序示例（续1）

```
MOV    AX,  STACK
MOV    SS,  AX                ; 为SS赋初值
MOV    AX,  TOP
MOV    SP,  AX                ; 为SP赋初值
MOV    DX,  OFFSET  NOTES    ; 显示提示信息
MOV    AH,  9H
INT    21H
MOV    BX,  NUM                ; 将数装入BX
MOV    CH,  4                  ; CH作循环计数器
ROTATE :
MOV    CL,  4                  ; CL中放移位位数
ROL    BX,  CL
MOV    AL,  BL
AND    AL,  0FH                ; AL中为一个16进制数
```



## 汇编程序示例（续2）



```
        ADD    AL, 30H                ; 转换为ASCII码值
        CMP    AL, '9'               ; 是0~9的数码?
        JLE    DISPLAY
        ADD    AL, 07H               ; 在A~F之间
DISPLAY:
        MOV    DL, AL                ; 显示16进制数
        MOV    AH, 2
        INT    21H
        DEC    CH
        JNZ    ROTATE
        MOV    AX, 4C00H             ; 返回
        INT    21H
CODE      ENDS                    ; 代码段结束
END      BEGIN                   ; 模块结束
```

# 汇编语言程序的组成

## 分段结构

- 按段进行组织，最多由4个段组成（代码、数据、附加、堆栈）
- 每个段以“段名 SEGMENT”开始，以“段名 ENDS”结束

## 语句行

- 段由若干语句行组成
- 语句行的三种类型：指令、伪指令、宏指令

```
DATA SEGMENT
    ... ; 数据段语句
DATA ENDS

STACK SEGMENT
    ... ; 堆栈段语句
STACK ENDS

CODE SEGMENT
    ... ; 代码段语句
CODE ENDS
```

# 语句的执行



## ❏ 伪指令语句的执行

- 汇编器计算伪指令语句中表达式的值
- 不产生机器代码
- 汇编器解释伪指令语句的含义并遵照“执行”

## ❏ 指令语句的执行

- 汇编器计算指令语句中表达式的值
- 汇编器将指令语句翻译成机器指令代码
- 程序运行时，由CPU按机器指令代码的要求完成各种运算与操作

# 伪指令语句1. 数据定义



## ④ 变量

- 编程时只能确定其初始值，程序运行期间可修改其值的数据对象称为变量
- 变量是存储单元中的数据，可定义在任何段，但通常都定义在数据段（DS）和附加段（ES）
- 变量由伪指令说明符DB、DW、DD等定义

## ④ 示例

A DB 50, 60, 70, 80  
DW 50, 60, 70, 80  
DD 50, 60, 70, 80

在 x86 汇编语言中，DB, DW, 和 DD 是定义数据的伪指令，分别代表 Define Byte, Define Word 和 Define Doubleword。

定义了一个字节数组，包含了四个元素，它们的值分别是 50, 60, 70 和 80（十进制数），每个元素占用 1 个字节的空間

“变量名”就是变量地址的名字，也可称为“变量的符号地址”

# 伪指令语句2. 符号定义



## ❏ 常数、常量

- 编程时已经确定其值，程序运行期间不会改变其值的数据对象称为**常数**
- 常数表达式的名字称为**常量**
- 常量可用伪指令说明符 “**EQU**” 或 “**=**” 定义
- 常量不产生目标代码，不占用存储单元

## ❏ 示例

A EQU 7 ;

A EQU 8 ; 错误, “EQU” 左边的符号名不可重复定义

B = 7 ;

B = 8 ; 正确, “=” 左边的符号名可以重复定义



# 数据定义和符号定义的示例

```
DATA SEGMENT      ; 数据段
    W1    DW    1, 2, 3, 4, 5, 6, 7
    B1    DB    10, 20, 30, 40, 50
    N1    EQU   B1-W1      ; N1=14
DATA ENDS          ; 数据段结束
CODE SEGMENT       ; 代码段
    ASSUME CS:CODE, DS:DATA
    BEGIN:
        ...
        MOV     AX,    W1
        MOV     B1,    AL
        ...
CODE ENDS           ; 代码段结束
END BEGIN           ; 模块结束
```

# 伪指令语句3. 段定义



## ❏ 段定义说明符1：SEGMENT（段开始）

◦ 示例

**CODE SEGMENT**

## ❏ 段定义说明符2：ENDS（段结束）

◦ 示例

**CODE ENDS**

## ❏ 段定义说明符3：ASSUME（指定段寄存器）

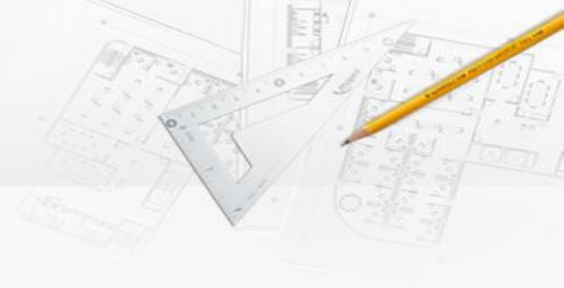
◦ 示例

**ASSUME CS:CODE, DS:DATA, SS:STACK**

# 示例

```
DATA    SEGMENT                ; 数据段
    NUM    DW    00111010000000111B
    NOTES  DB    'The result is :', '$'
DATA    ENDS                  ; 数据段结束
CODE    SEGMENT                ; 代码段
    ASSUME  CS:CODE, DS:DATA
    BEGIN:
        MOV  AX, DATA
        MOV  DS, AX           ; 为DS赋初值
        ...
CODE    ENDS                  ; 代码段结束
        END    BEGIN         ; 模块结束
```

# 伪指令语句4. 指定段内的偏移地址



## ❏ ORG说明符

- 格式：ORG 常数表达式
- 作用：指定当前可用的存储单元的**偏移地址**为**常数表达式的值**

## ❏ EVEN说明符

- 格式：EVEN
- 作用：将当前可用的存储单元的**偏移地址**调整为**最近的偶数值**

# 示例

**DATA      SEGMENT**

**ORG**    1000H

A DB    47H, 12H, 45H

**EVEN**

B DB    47H

**DATA      ENDS**

说明:

- ① **ORG指令将A的偏移地址部分指定为1000H**
- ② 从A开始存放3个字节变量，占用地址1000H、1001H和1002H
- ③ **EVEN指令会将B的偏移地址部分从1003H调整为偶数地址1004H**

# 伪指令语句5. 过程定义



## ④ PROC说明符 相当于定义了一个函数

- 格式：过程名 PROC 类型属性名
- 说明：从“过程名”代表的地址开始定义一个过程；“类型属性名”可选择NEAR（近过程）或FAR（远过程），默认为NEAR

## ④ ENDP说明符

- 格式：过程名 ENDP
- 说明：表示该过程到此结束。此处的“过程名”必须与过程开始时PROC左边的“过程名”相同



# 示例

```
...  
CODE      SEGMENT                ; 代码段  
    ASSUME  CS:CODE, DS:DATA  
    BEGIN:  
        ...  
        CALL  DISPLAY                ; 过程调用  
        ...  
  
        DISPLAY PROC  NEAR ; 过程定义  
            ...  
            RET  
        DISPLAY ENDP  
CODE      ENDS                    ; 代码段结束  
    END      BEGIN                ; 模块结束
```



# 本讲到此结束，谢谢 欢迎继续学习本课程

计算机组织与体系结构 Computer Architectures  
主讲：陆俊林