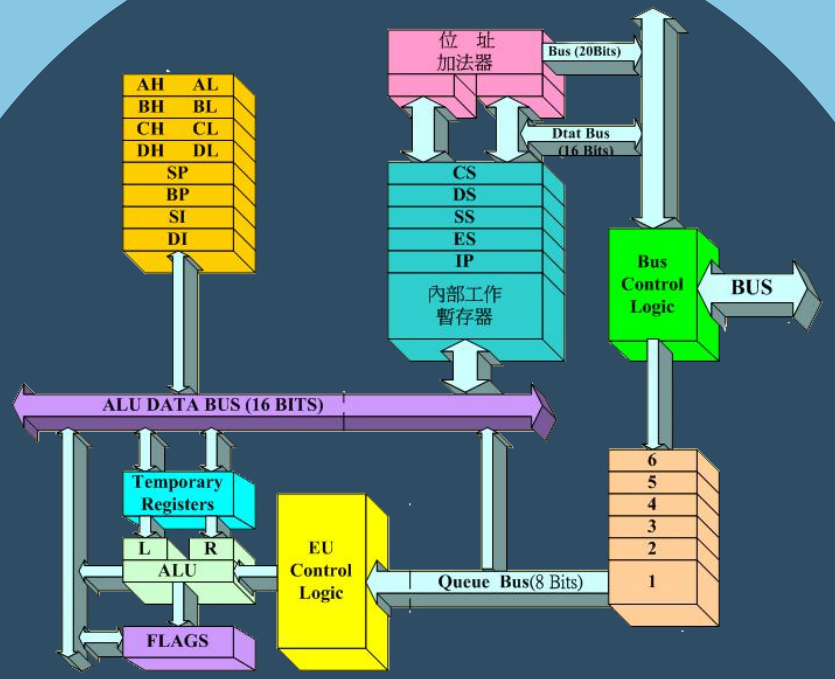
I. 绪　论

贺利坚　主讲



汇编语言程序设计

Assembly Language

汇编语言程序设计课程内容

1. 绪论
2. 访问寄存器和内存
3. 汇编语言程序
4. 内存寻址方式
5. 流程转移与子程序
6. 中断及其应用
7. 高级汇编语言技术

0101 我们要学汇编语言

0102 由机器语言到汇编语言

0103 计算机的组成

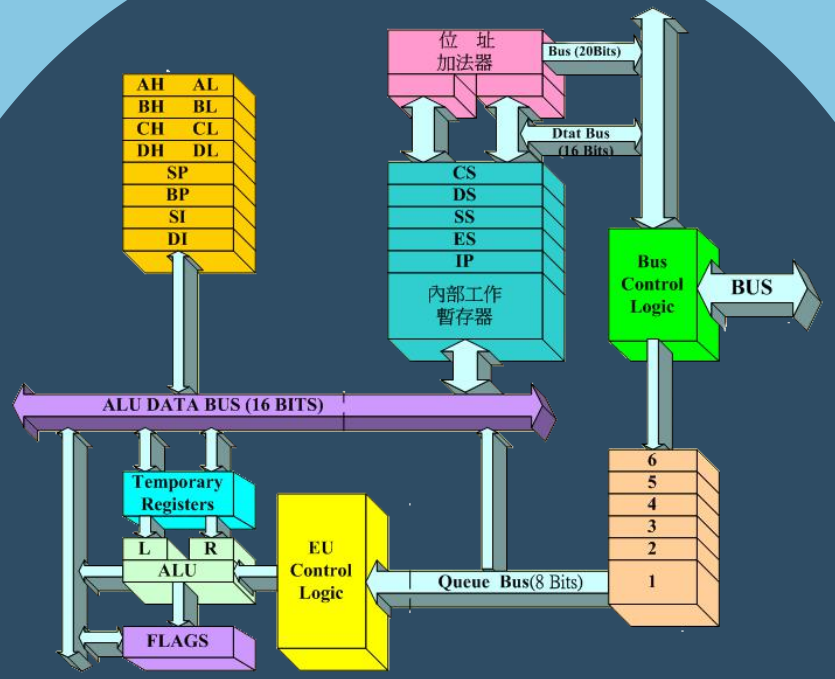
0104 内存的读写与地址空间

0105 汇编语言实践环境搭建

视频（共7个）	教材对应章节
0101 为什么要学汇编语言	
0102 由机器语言到汇编语言	1.1-1.3节
0103 计算机的组成	1.4-1.10节
0104 内存的读写与地址空间	1.11-1.15节
0105 汇编语言实践环境搭建	

为什么要学汇编语言

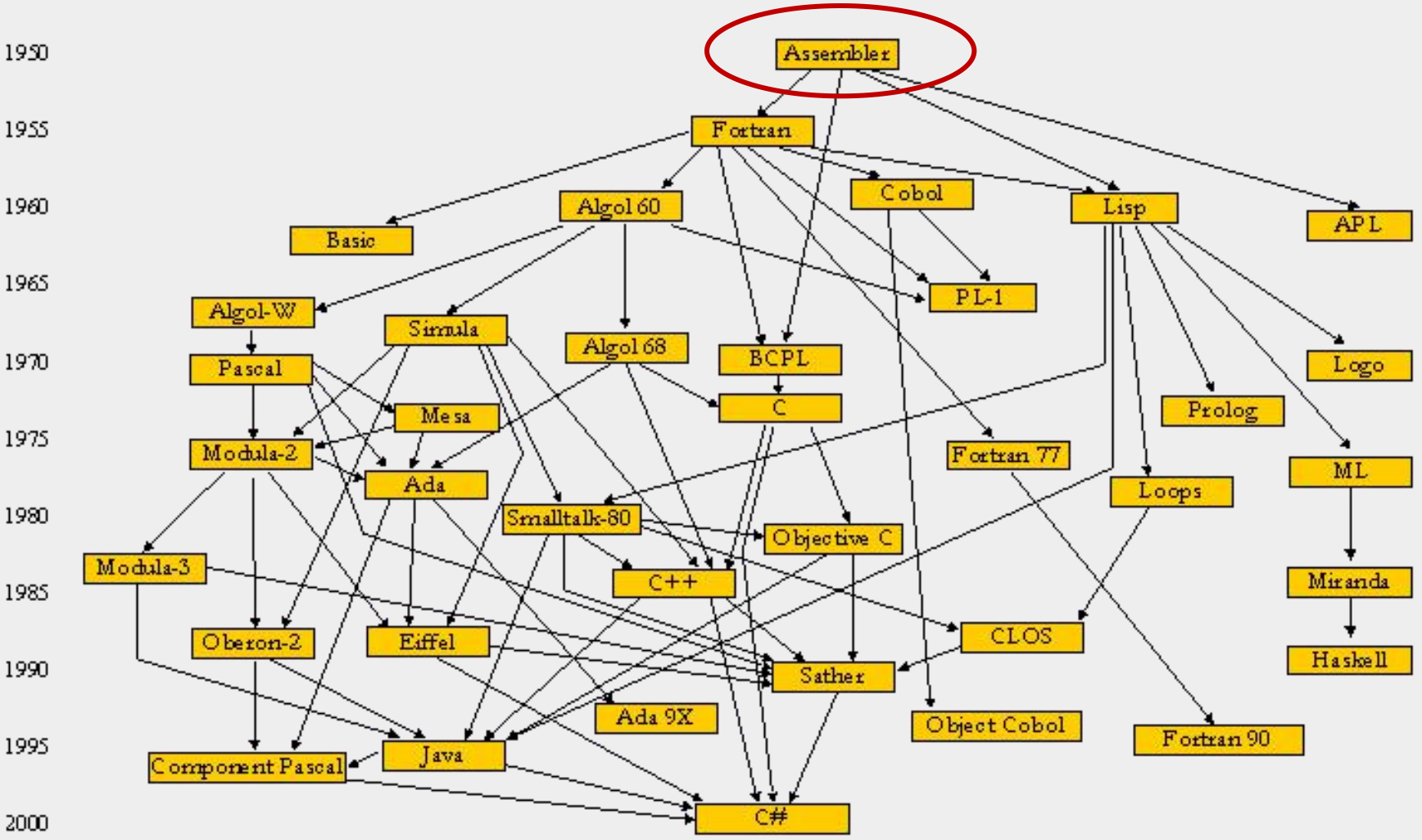
贺利坚　主讲



汇编语言程序设计

Assembly Language

程序设计语言家谱——汇编是老祖宗！



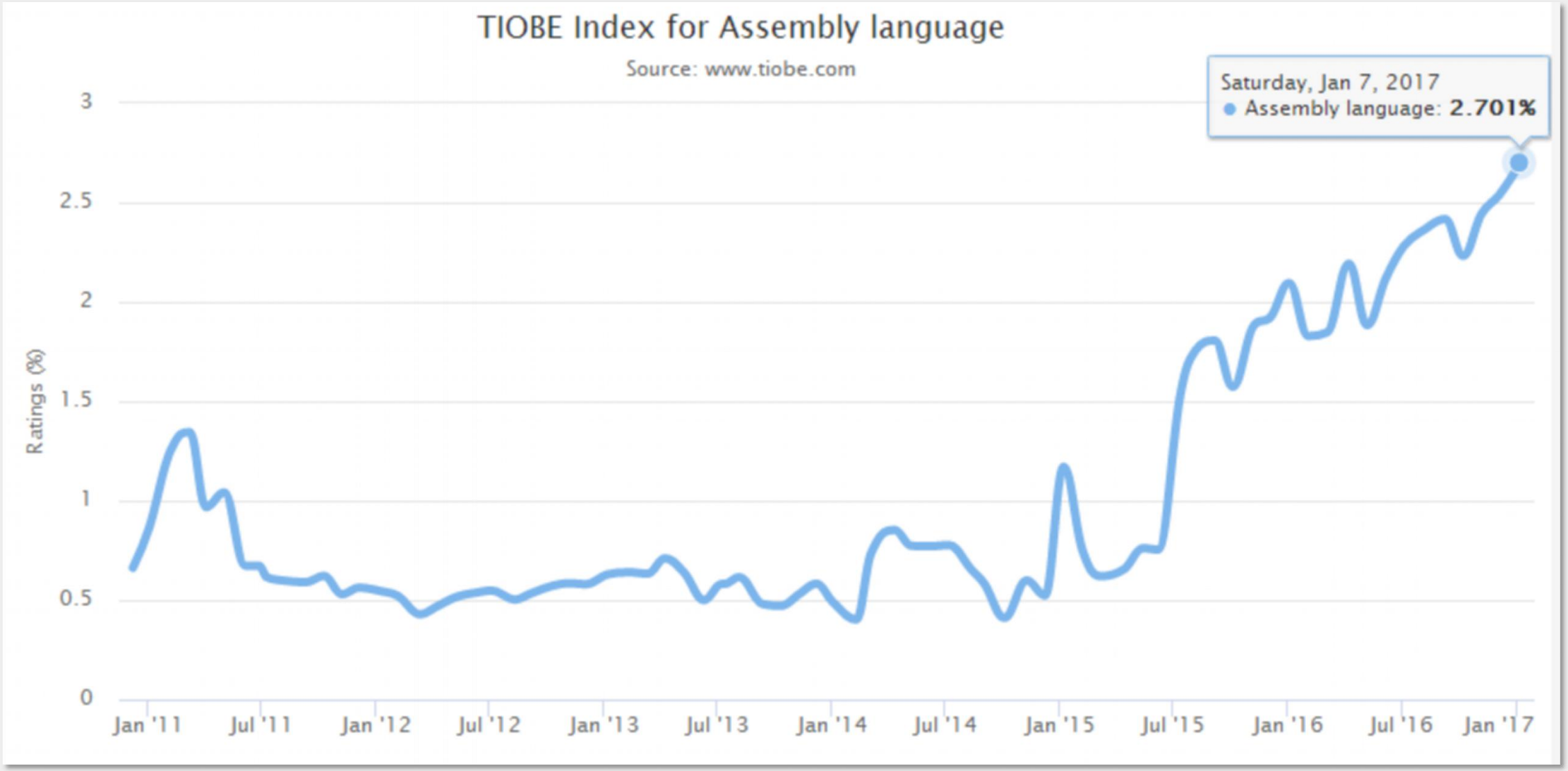
汇编老矣，尚能饭否？

年事已高的汇编，仍被广泛使用，甚至不可替代(TIOBE排行榜)

Jan 2017	Jan 2016	Change	Programming Language	Ratings	Change
1	1		Java	17.278%	-4.19%
2	2		C	9.349%	-6.69%
3	3		C++	6.301%	-0.61%
4	4		C#	4.039%	-0.67%
5	5		Python	3.465%	-0.39%
6	7	^	Visual Basic .NET	2.960%	+0.38%
7	8	^	JavaScript	2.850%	+0.29%
8	11	^	Perl	2.750%	+0.91%
9	9		Assembly language	2.701%	+0.61%
10	6	⌄⌄	PHP	2.564%	-0.14%

http://www.tiobe.com/tiobe-index//

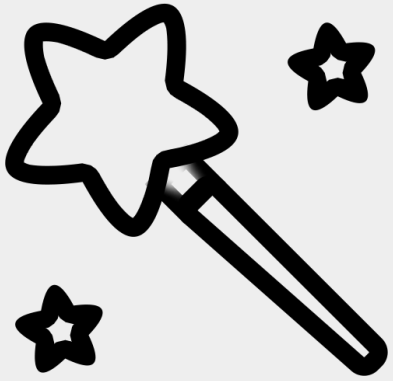
汇编语言的TIOBE指数--汇编近来风头更劲



http://www.tiobe.com/tiobe-index/assembly-language/

学习汇编语言的理由

- 汇编语言仍在发挥不可替代的作用
  - 效率
    - 运行效率：开发软件的核心部件，快速执行和实时响应。
    - 开发效率：做合适的事，开发效率无敌
  - 底层：计算机及外围设备的驱动程序
    - 操作系统的内核
    - 嵌入式系统：家用电器、仪器仪表、物联网……
- 汇编语言在学习计算机中起到的独特作用——直击计算机系统的核心
  - 便于加深对计算机原理和操作系统等课程的理解。
  - 通过学习和使用汇编语言，能够感知、体会和理解机器的逻辑功能
    - 向上为理解各种软件系统的原理，打下技术理论基础
    - 向下为掌握硬件系统的原理，打下实践应用基础。
  - 学会底层的程序调试和错误分析方法。



这门课要学什么？

- 课名：汇编语言程序设计
- 定位：理解硬件结构，掌握指令集，理解程序的运行过程
- 内容： 8088、8086指令集与汇编语言程序设计
- 教材：

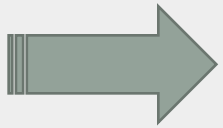
- 本课只解决入门
  - 降低入门难度
  - 关注核心思维与方法
- 进一步延伸
  - Inter 80x86汇编
  - Linux汇编
  - ARM 汇编

学习方法

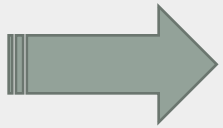
- 贯穿实践的方法
- 学会观察机器的内部状态
- 学习进程

视频

教材



检测



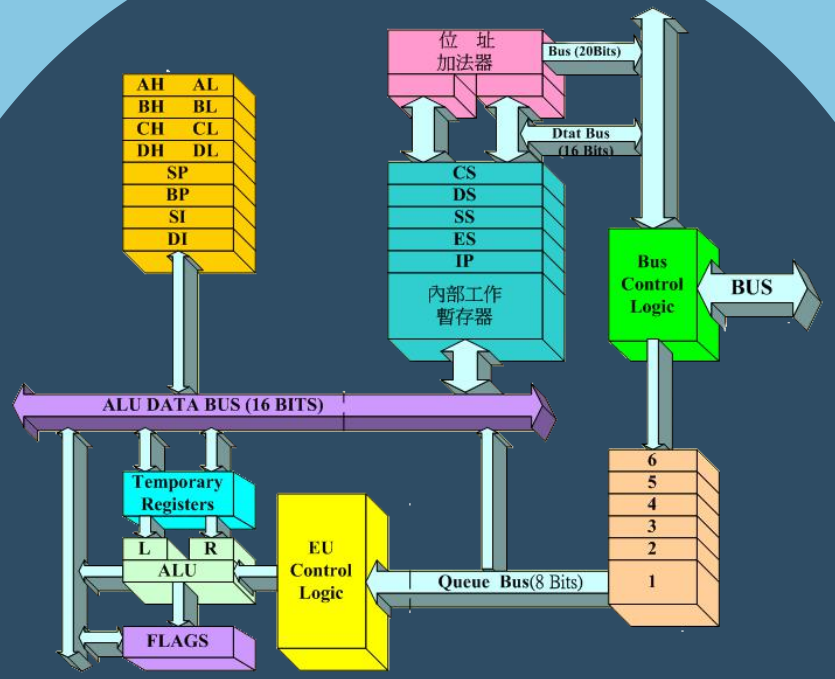
实验

学法

- 书：循序渐进
- 本：精熟学习

由机器语言到汇编语言

贺利坚　主讲

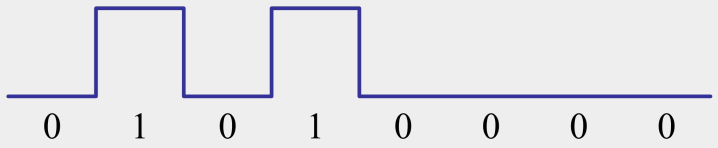


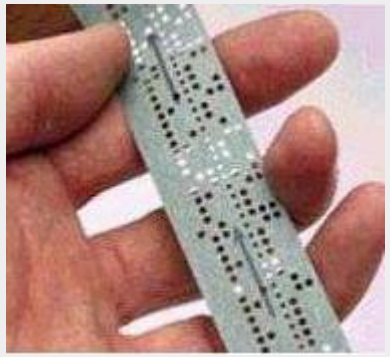
汇编语言程序设计

Assembly Language

机器语言与机器指令

- **机器语言**是机器指令的集合。
- **机器指令**是一台机器可以正确执行的命令。
- 机器指令由一串二进制数表示，例 01010000
- 电平脉冲：





- 早期程序员们的工作形态
  - 将 0、1 数字编程的程序代码打在纸带或卡片上，1打孔，0不打孔，再将程序通过纸带机或卡片机输入计算机，进行运算。

- 例：计算S = 768 + 12288 - 1280的程序
- 机器码：

10110000000000000000011
00000101000000000110000
00101101000000000000101

这个程序错哪了？

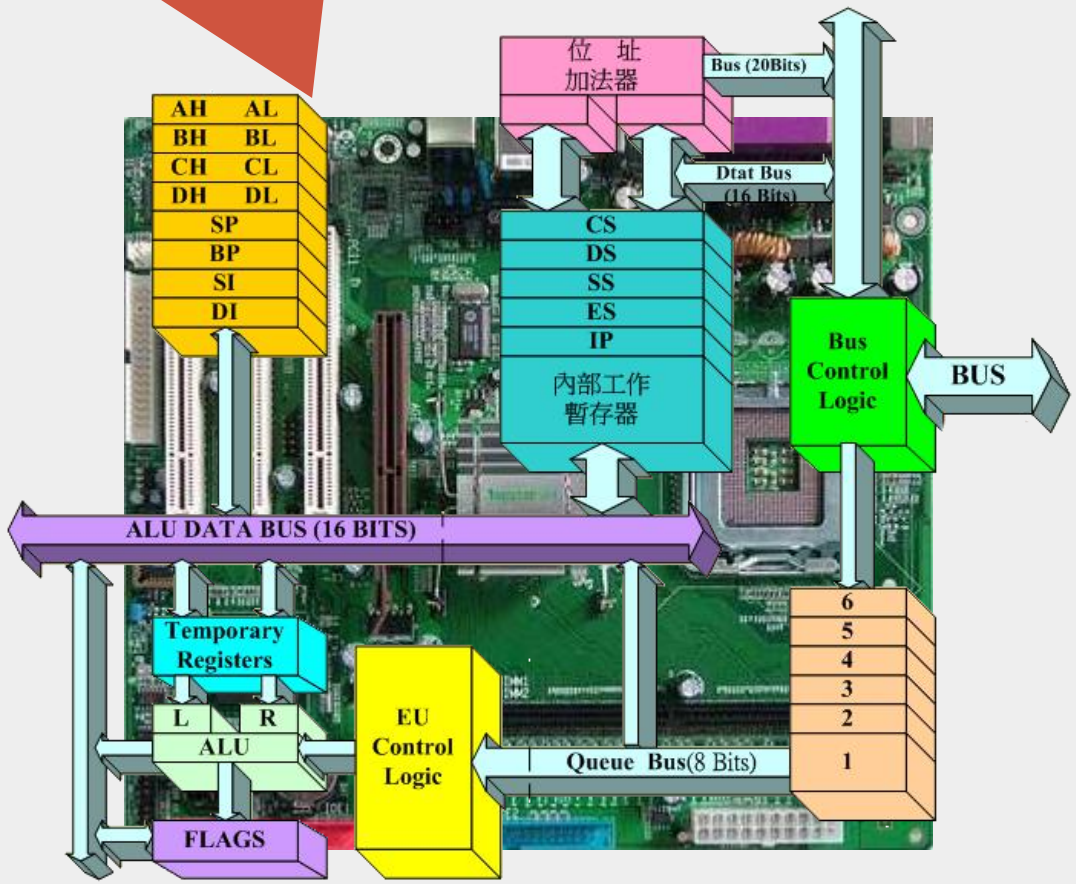
10110000000000000000011
00000101000000000110000
00010110100000000000101

汇编语言与汇编指令

- **汇编语言**的主体是汇编指令。
- **汇编指令**和机器指令的差别在于指令的表示方法上
  - 汇编指令是机器指令便于记忆的书写格式。
  - 汇编指令是机器指令的助记符。

寄存器：CPU中可以存储数据的器件。

一个CPU中有多个寄存器。



机器指令： **1000100111011000**

操作：将寄存器BX的内容送到AX中

汇编指令：**MOV AX, BX**

用汇编语言编写程序的工作过程

伪指令
——由编译器执行

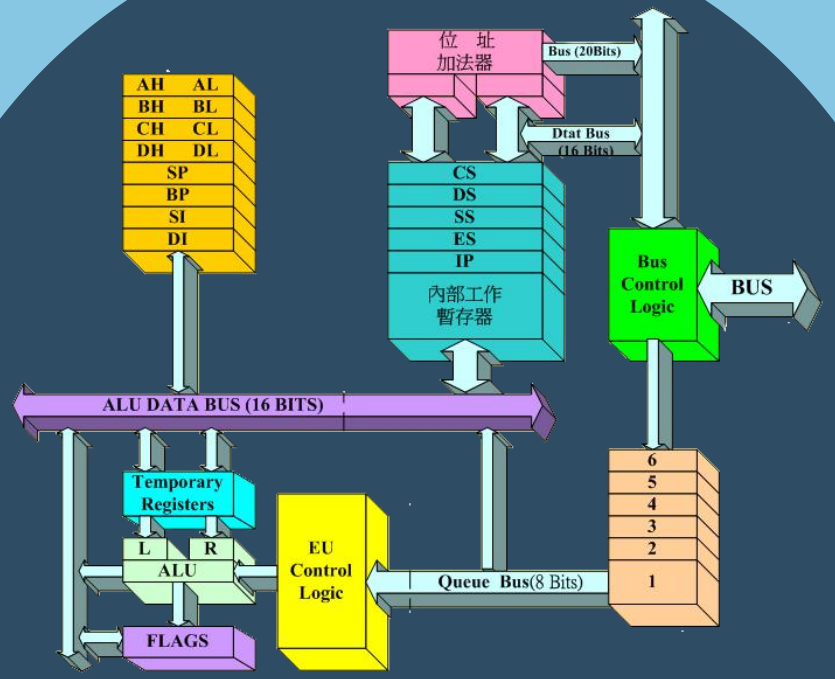
其它符号
——由编译器识别

```
;汇编语言程序示例
assume cs:codesg
codesg segment
start:
    mov ax,0123H
    mov bx,0456H
    add ax,bx
    add ax,ax

    mov ax,4c00h
    int 21h
codesg ends
end
```

汇编指令
——机器码的助记符

计算机的组成

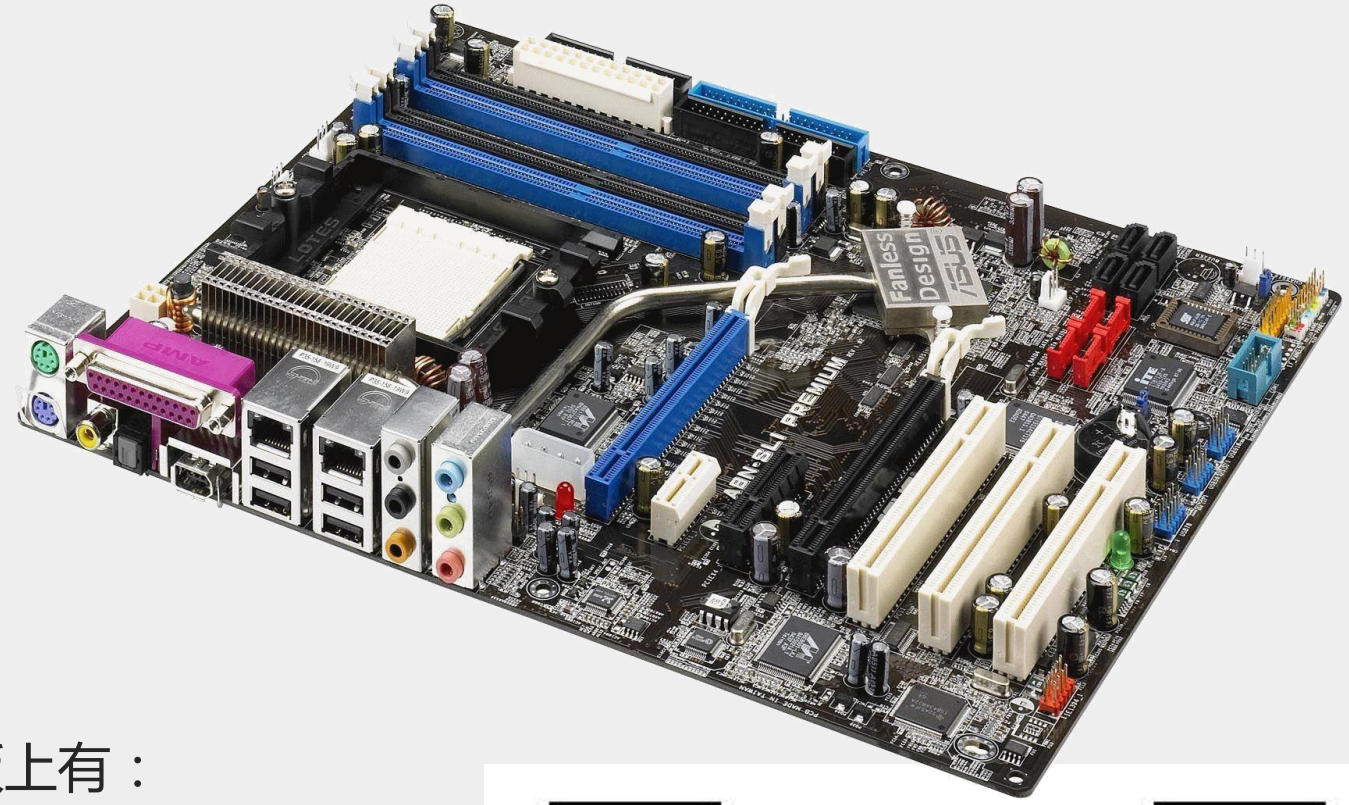
贺利坚　主讲

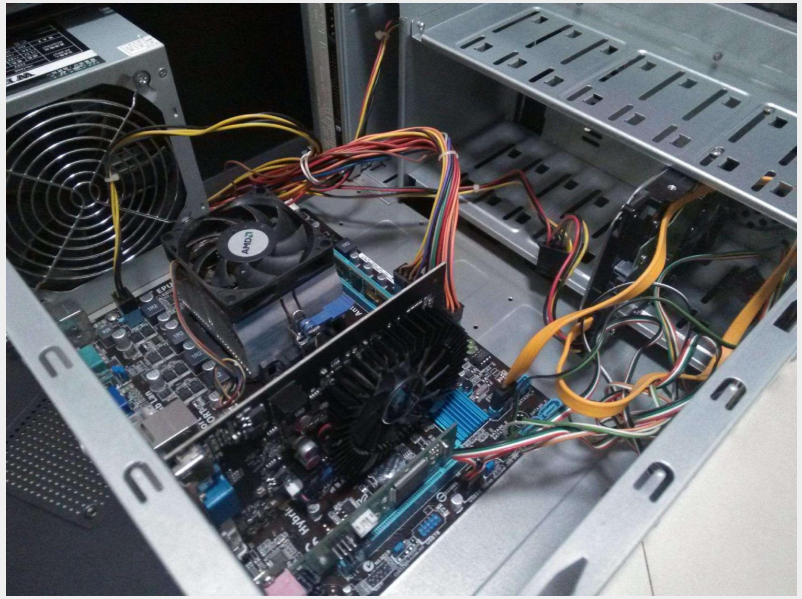


汇编语言程序设计

Assembly Language

“解剖”计算机

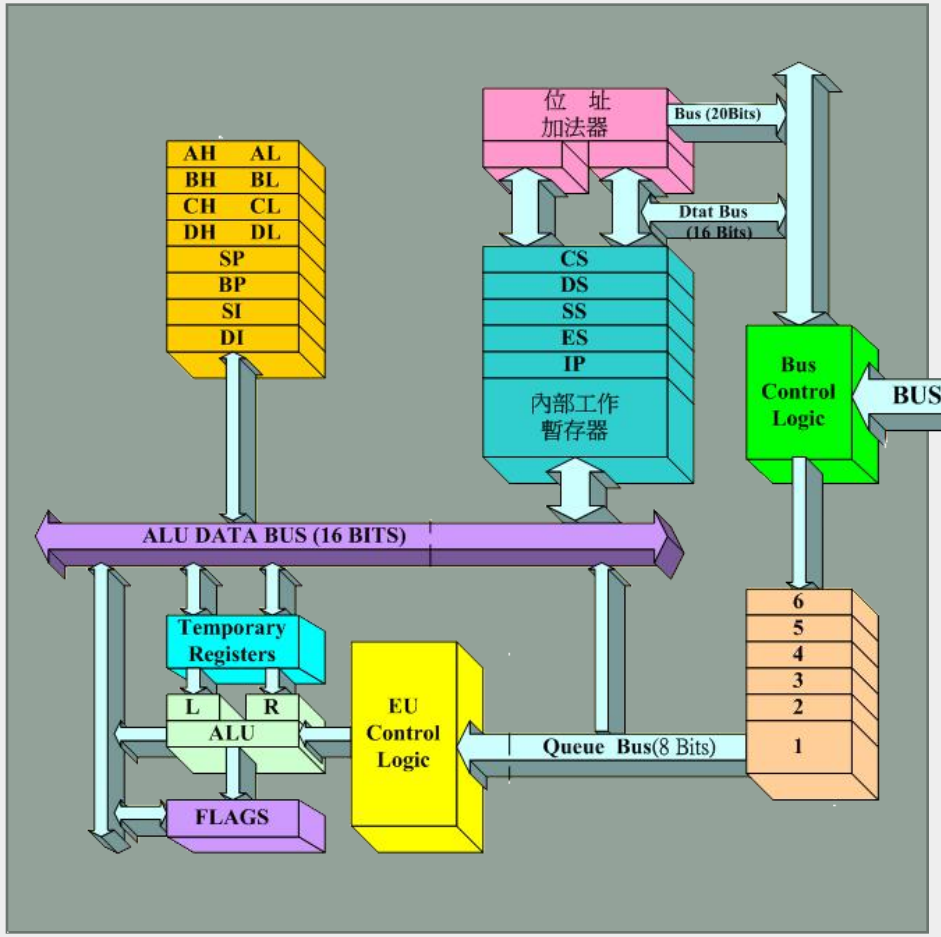


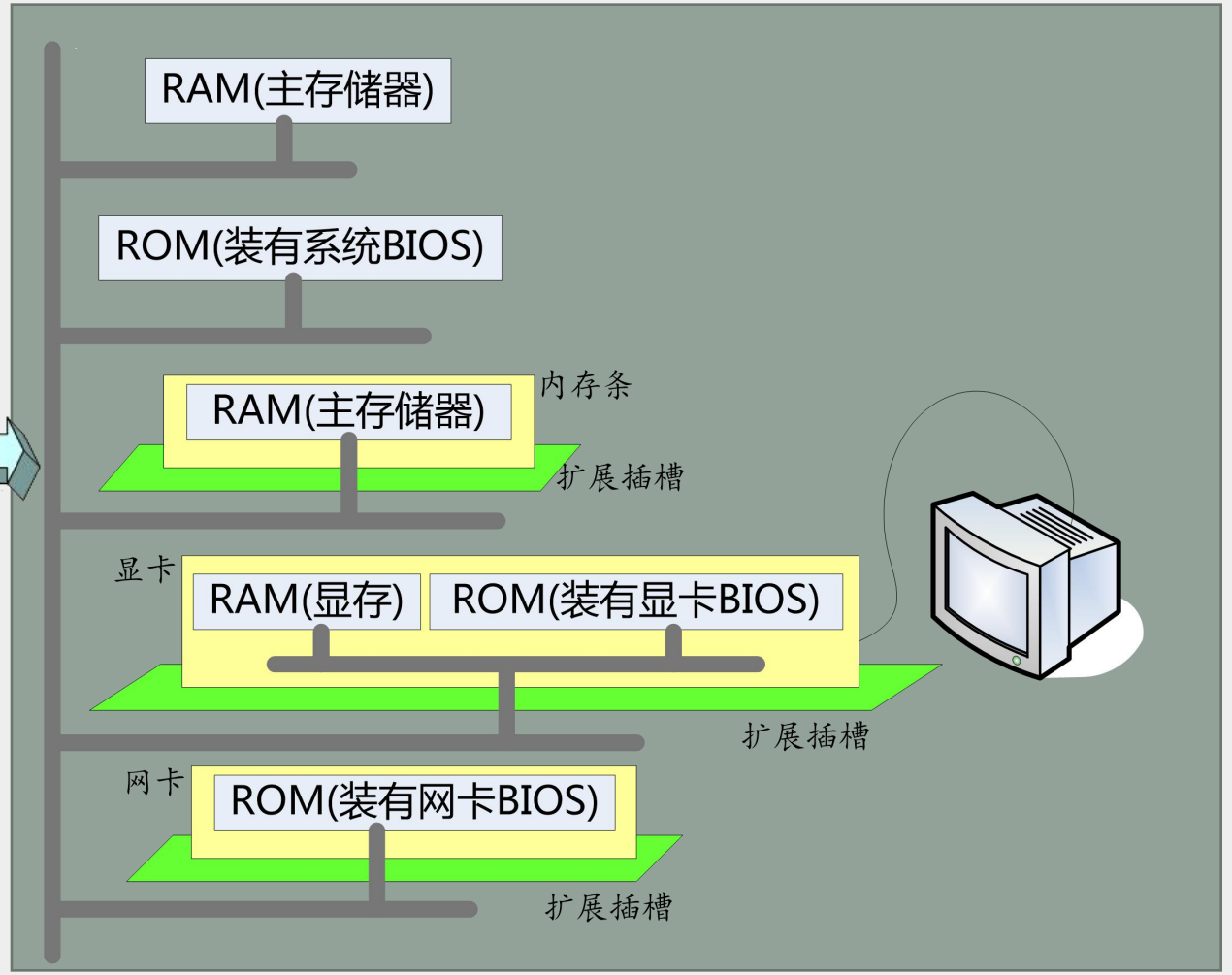


主板上有：

- CPU
- 总线
- 内存(条)
- 扩展槽(接外部设备)

计算机的组成





CPU 是计算机的核心部件，它控制整个计算机的运作并进行运算。要想让一个CPU工作，就必须向它提供指令和数据。

指令和数据在存储器（内存）中存放。

离开了内存，性能再好的CPU也无法工作。

指令和数据的表示

- 计算机中的数据和指令，存储在内存或磁盘上。
- 数据和指令，都是二进制信息。
- 问题：二进制信息1000100111011000是数据，还是指令？
  - 1000100111011000 –> 89D8H（数据）
  - 1000100111011000 –> MOV AX,BX（程序）
- 数据如何表示？
  - 1000100111011000**B**（二进制）
  - 89D8**H**（十六进制）
  - 104730**O**（八进制）
  - 35288**D**（十进制）
- 数据量：B、KB、MB、GB、TB...

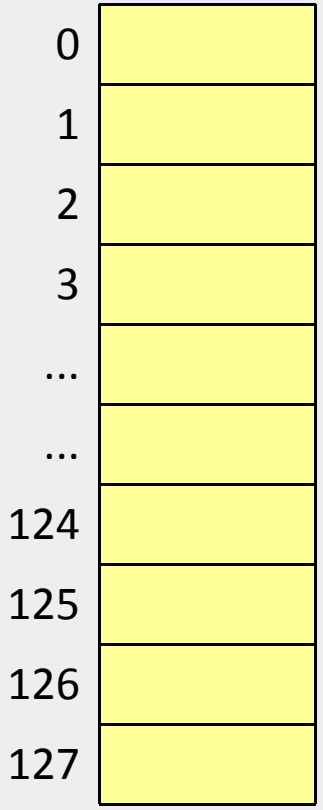
计算机中的存储单元

- 存储器被划分为若干个存储单元，每个存储单元从0开始顺序编号；
- 例如：

  一个存储器有128个存储单元，

  编号从0~127,

  如右图示：

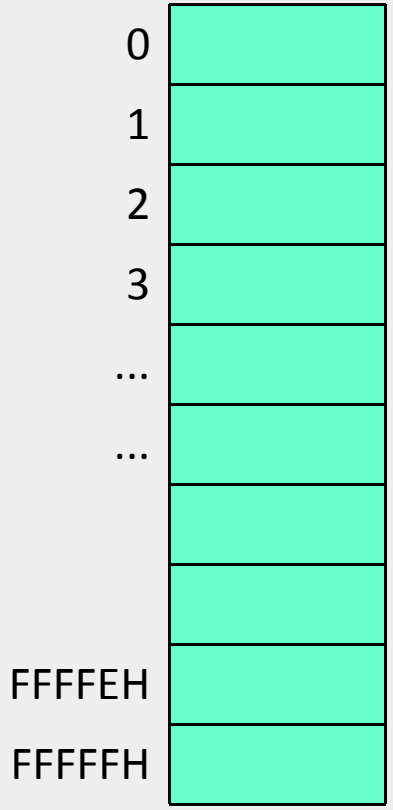


- 实际

  内存空间很“大”，

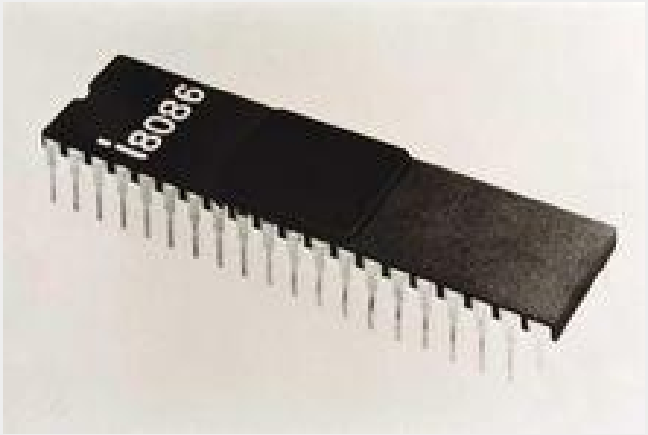
  8086有20条数据线，

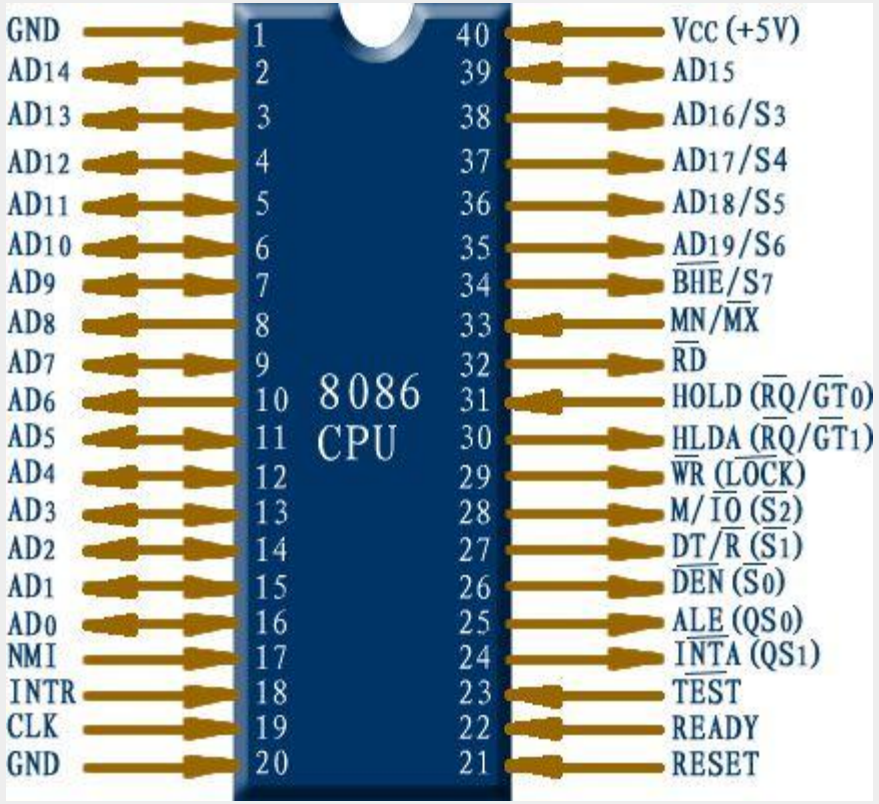
  寻址空间2^{20}，为1MB

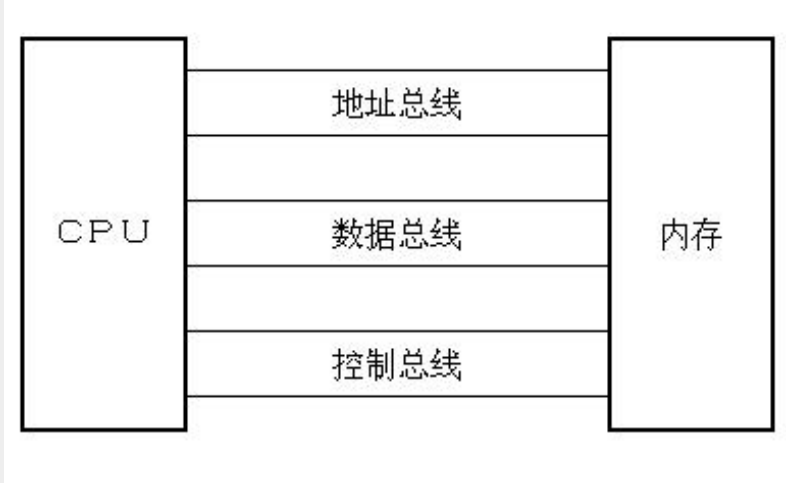


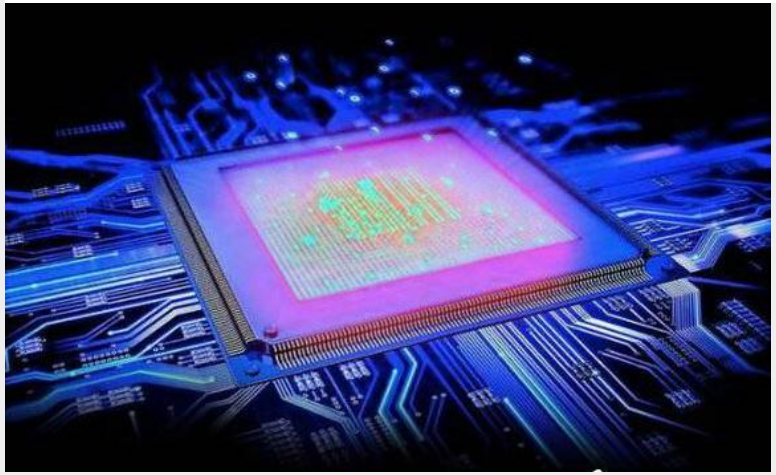
计算机中的总线

- 在计算机中专门有连接CPU和其他芯片的导线，通常称为总线。
  - 物理上：一根根导线的集合；
  - 逻辑上划分为
    - 地址总线
    - 数据总线
    - 控制总线



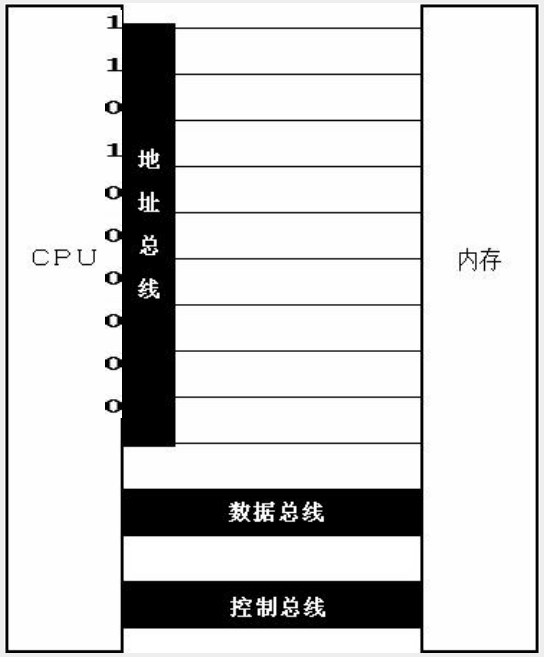




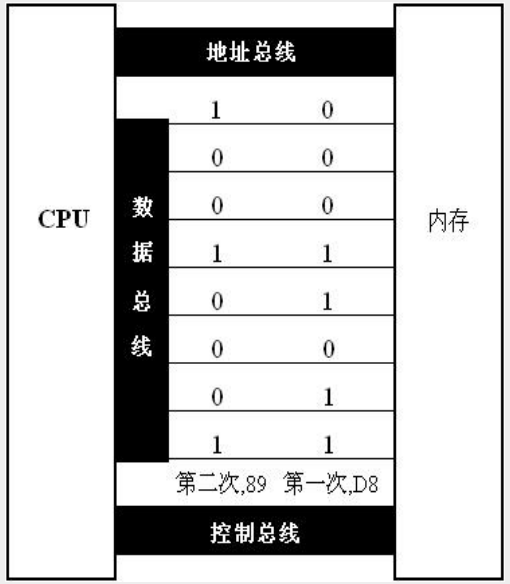


三类总线

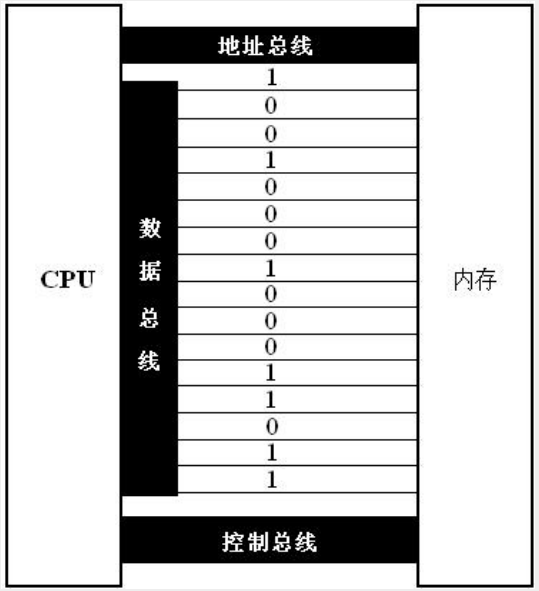
- CPU是通过**地址总线**来指定存储单元的。
- 地址总线宽度，决定了可寻址的存储单元大小。
- N根地址总线（宽度为N），对应寻址空间2^N。



- CPU与内存或其它器件之间的数据传送是通过**数据总线**来进行的。
- 数据总线的宽度决定了CPU和外界的数据传送速度。
- 例：向内存中写入数据89D8H时的数据传送

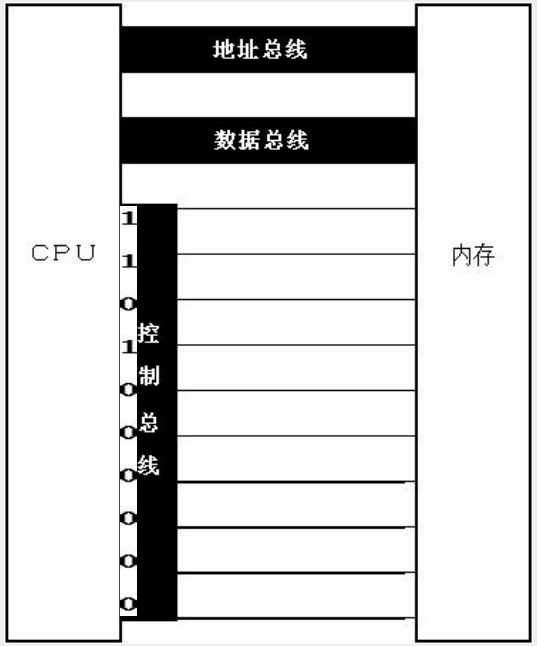


8088CPU(8位数据总线)上传送的信息



8086CPU(16位数据总线)上传送的信息

- CPU通过**控制总线**对外部器件进行控制。
- 控制总线是一些不同控制线的集合
- 控制总线宽度决定了CPU对外部器件的控制能力。

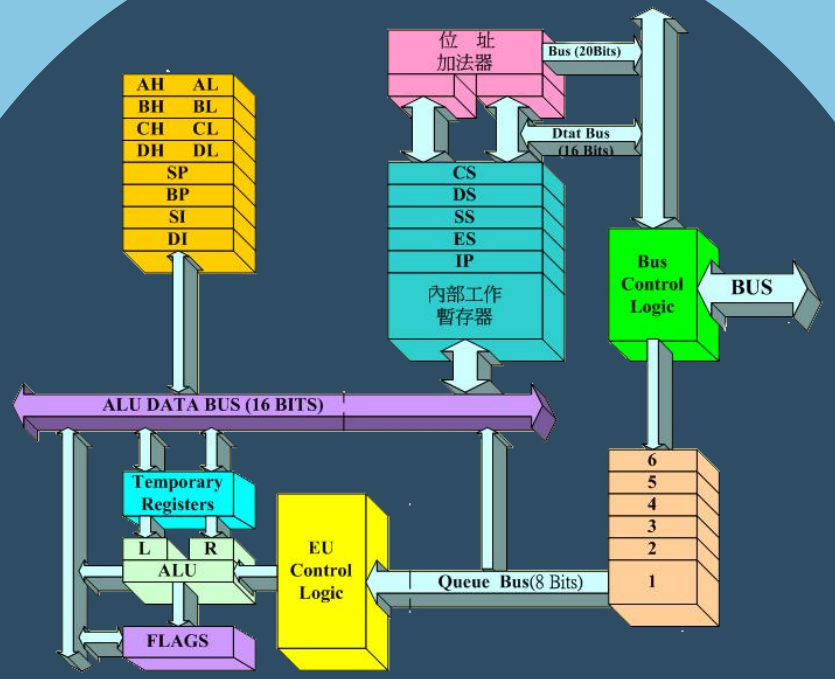


x86CPU性能一览

CPU	地址总线宽度	寻址能力	数据总线宽度	一次传送数据	读取1KB数据要读___次
8080	16	640KB	8	1B	1024
8088	20	1MB	8	1B	1024
8086	20	1MB	16	2B	512
80286	24	16MB	16	2B	512
80386	32	4GB	32	4B	256

内存的读写与地址空间

贺利坚　主讲



汇编语言程序设计

Assembly Language

CPU对存储器的读写

- CPU要想进行数据的读写，必须和外部器件进行三类信息的交互：
  - 存储单元的地址
    （地址信息）
  - 器件的选择，读或写命令
    （控制信息）
  - 读或写的数据
    （数据信息）
- 演示

  机器码： 10100000000000110000000

  16进制：A00300

  汇编指令：MOV AL,[3]

  含义：从3号单元读取数据送入寄存器AL

CPU对存储器的读写

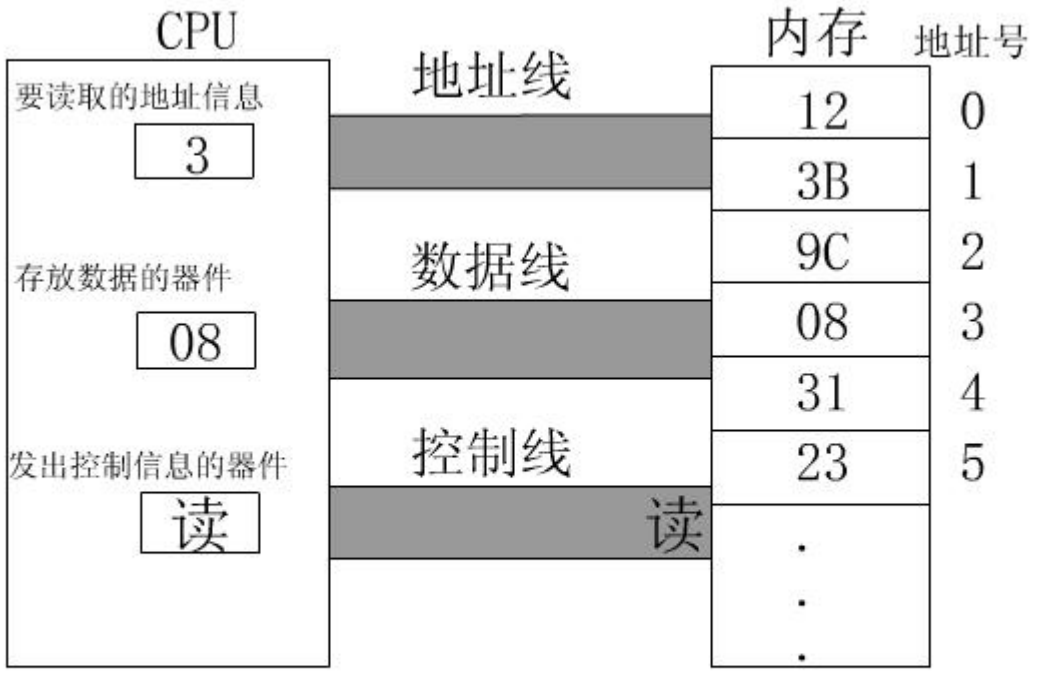
- CPU要想进行数据的读写，必须和外部器件进行三类信息的交互：
  - 存储单元的地址
    （地址信息）
  - 器件的选择，读或写命令
    （控制信息）
  - 读或写的数据
    （数据信息）
- 演示

  机器码： 10100000000000110000000

  16进制：A00300

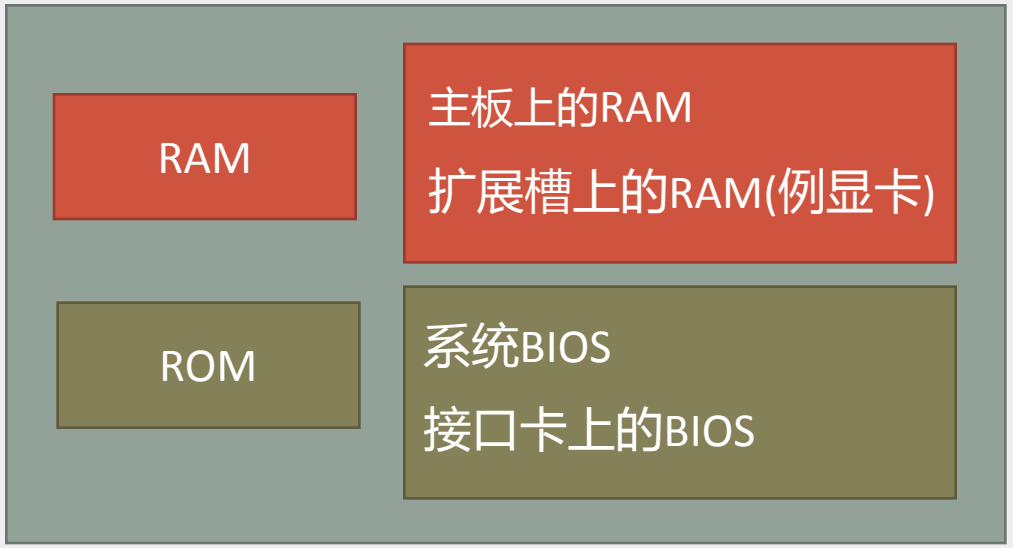
  汇编指令：MOV AL,[3]

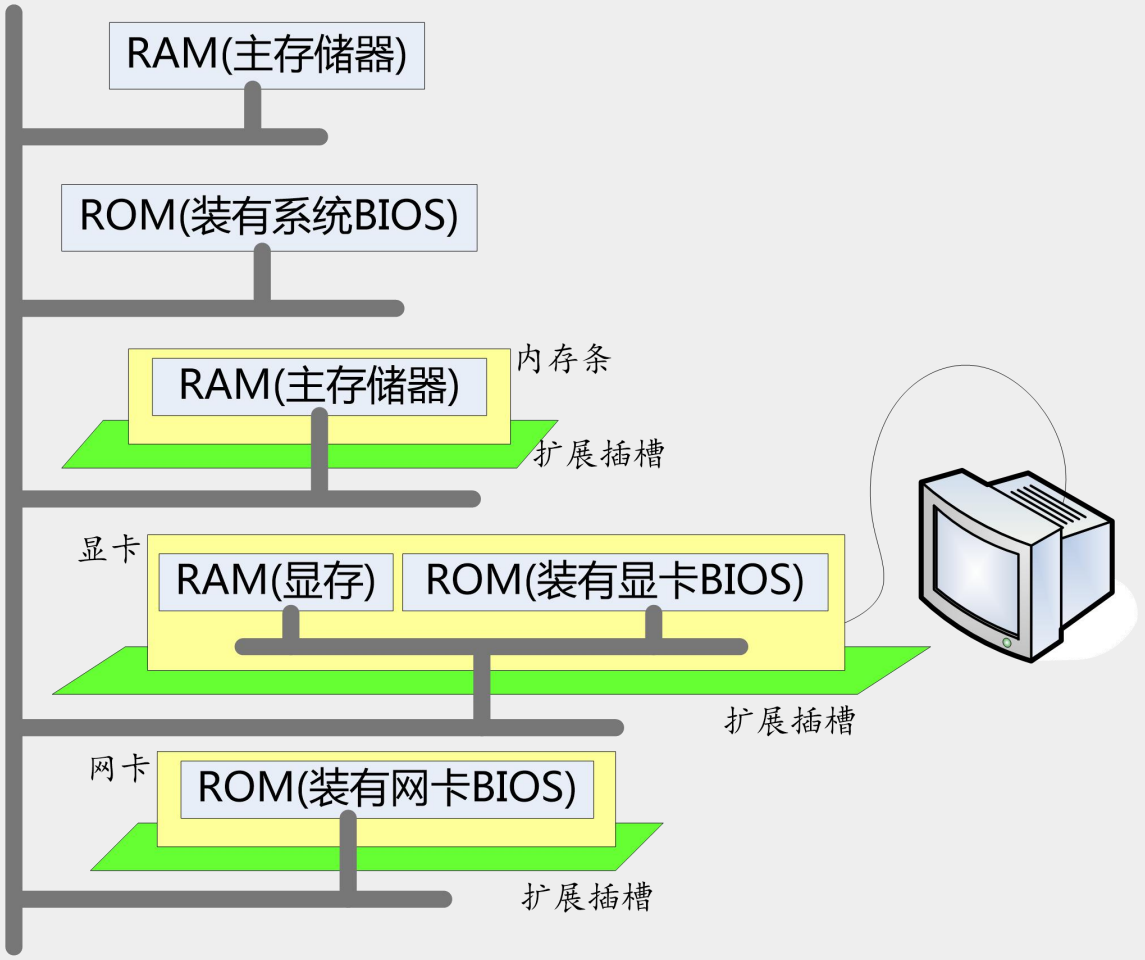
  含义：从3号单元读取数据送入寄存器AL



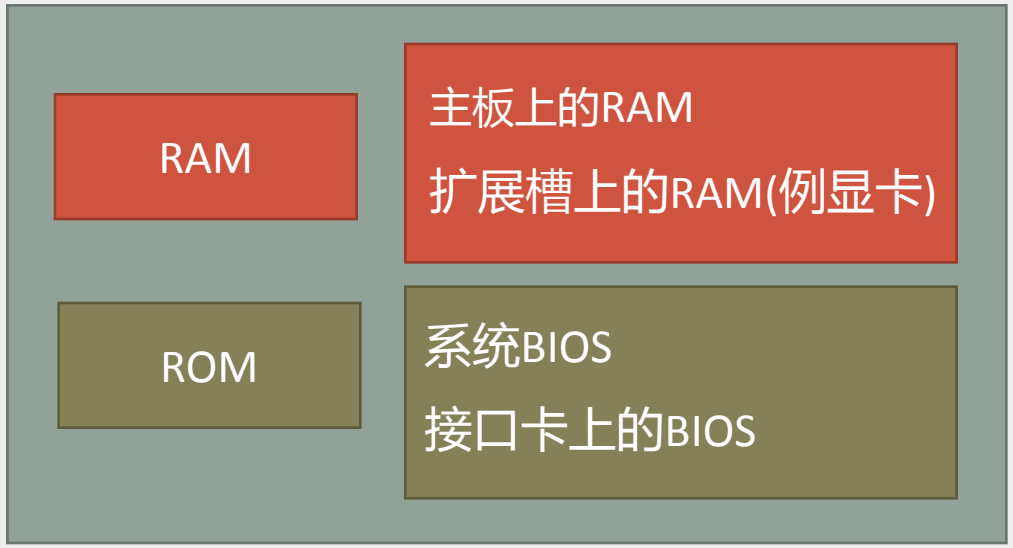
内存地址空间

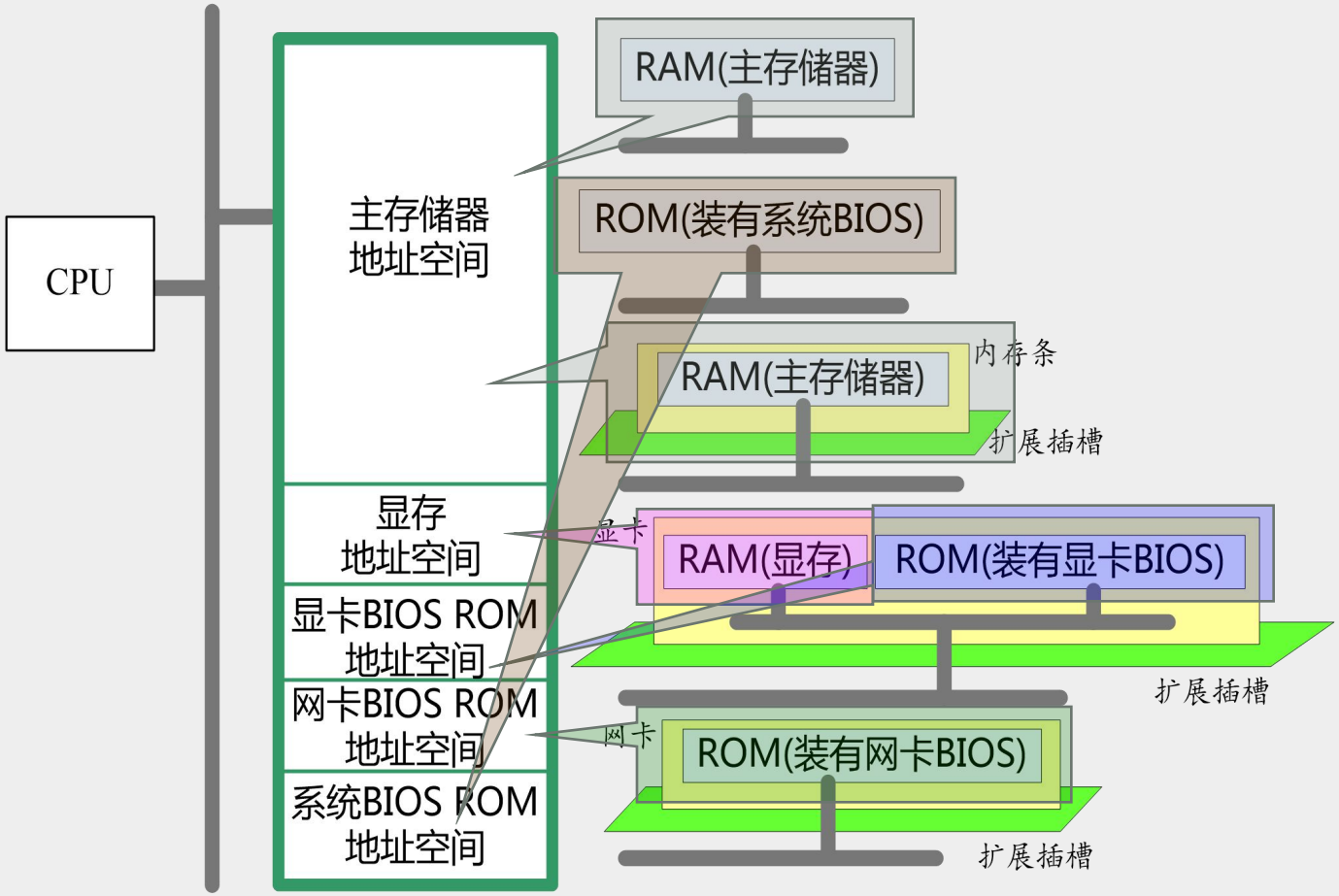
- 什么是内存地址空间
  - CPU地址总线宽度为N，**寻址空间**为2^NB
  - 8086CPU的地址总线宽度为20，那么可以寻址1MB个内存单元，其**内存地址空间**为1MB。
- 从CPU角度看地址空间分配





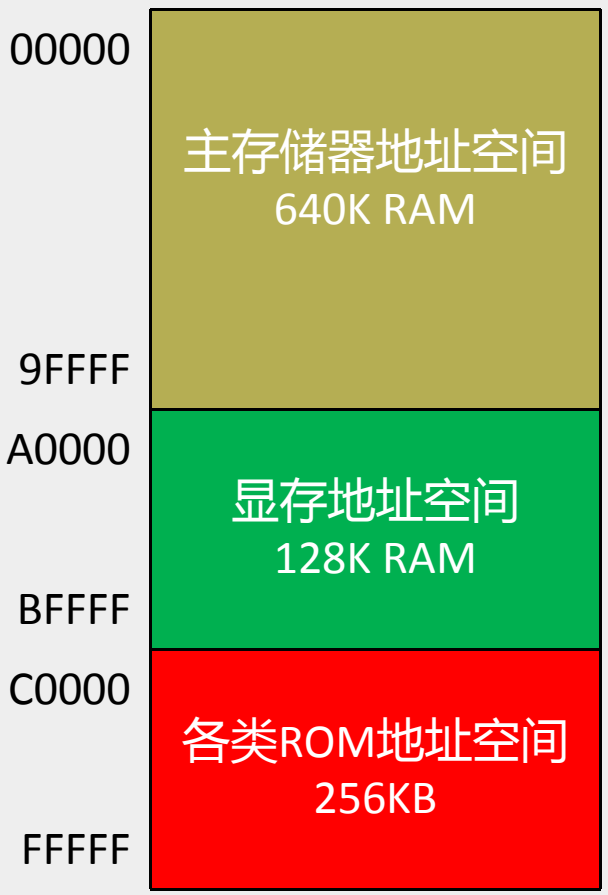
将各类存储器看作一个逻辑存储器——统一编址

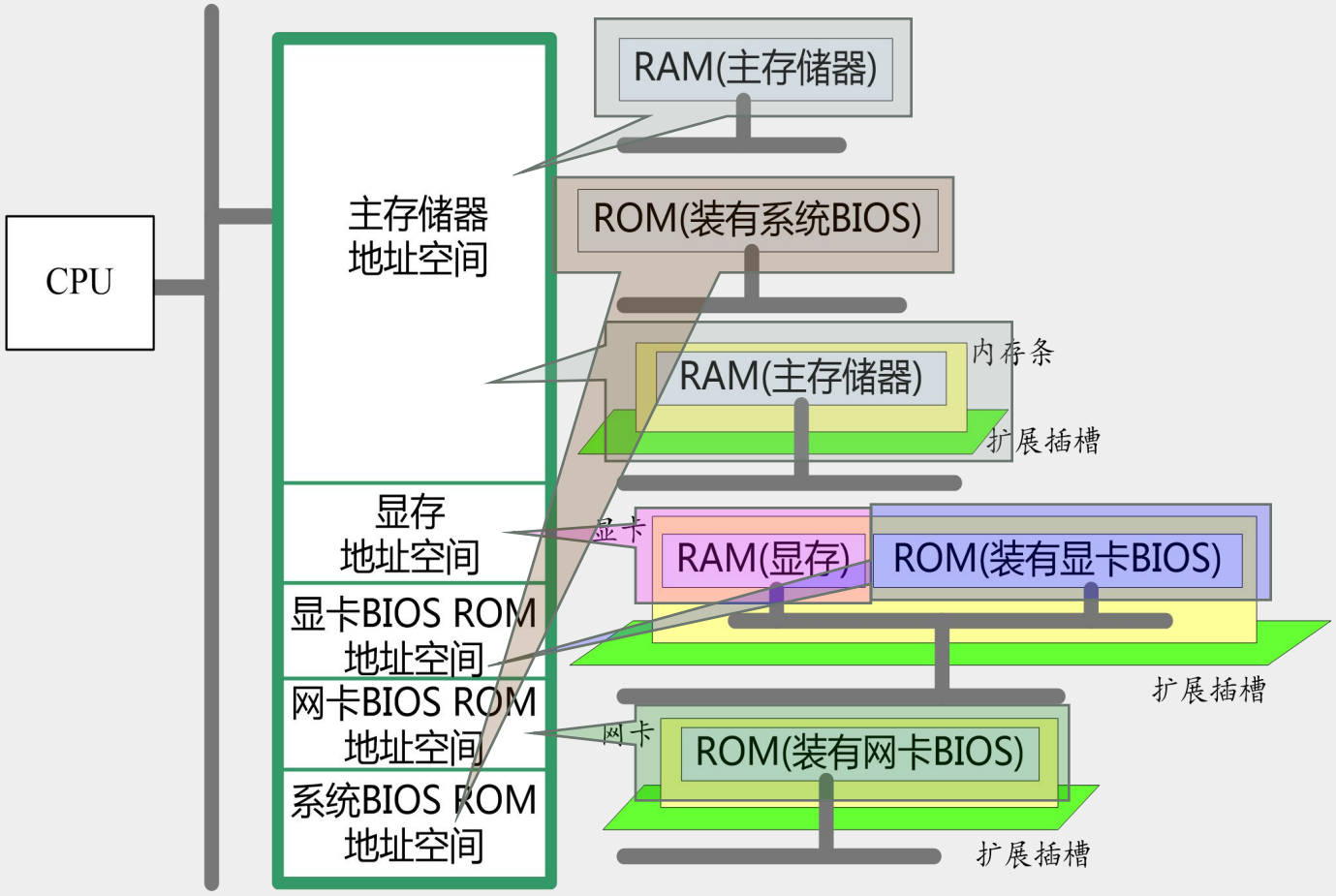




- 所有的物理存储器被看作一个由若干存储单元组成的逻辑存储器；
- 每个物理存储器在这个逻辑存储器中占有一个地址段，即一段地址空间；
- CPU在这段地址空间中读写数据，实际上就是在相对应的物理存储器中读写数据。

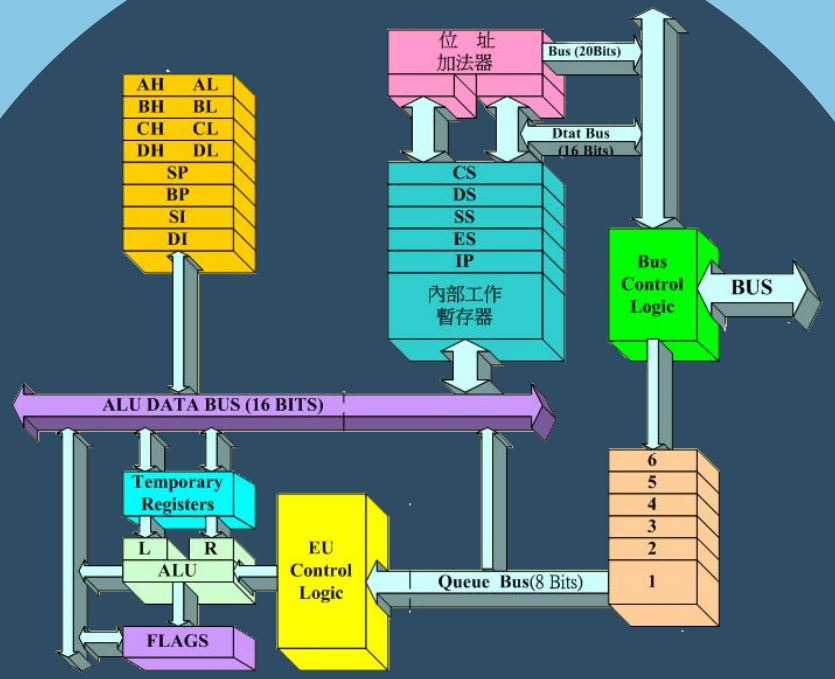
内存地址空间的分配方案——以8086PC机为例





I. 绪　论

贺利坚　主讲



汇编语言程序设计

Assembly Language

汇编语言程序设计课程内容

1. 绪论
2. 访问寄存器和内存
3. 汇编语言程序
4. 内存寻址方式
5. 流程转移与子程序
6. 中断及其应用
7. 高级汇编语言技术

0101 我们要学汇编语言

0102 由机器语言到汇编语言

0103 计算机的组成

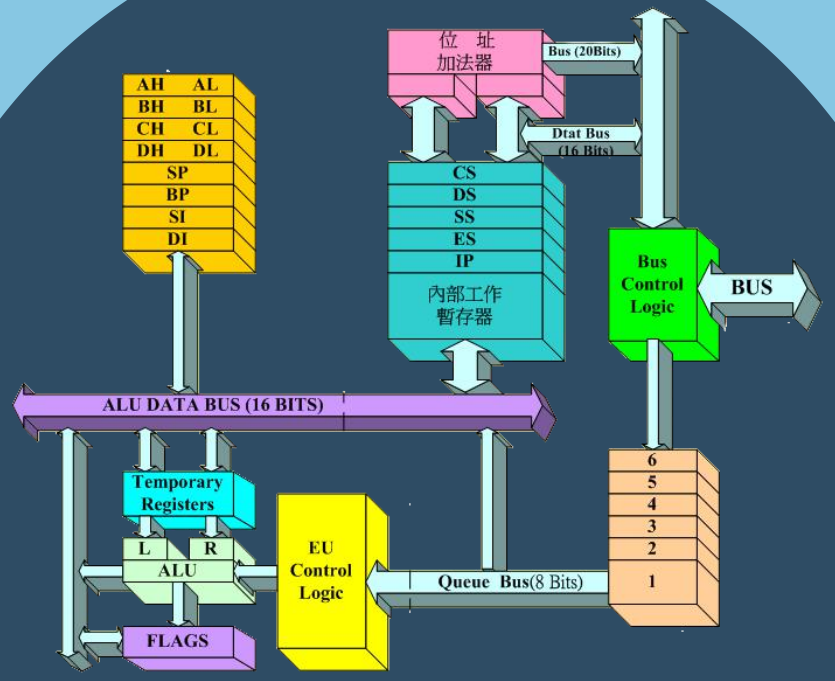
0104 内存的读写与地址空间

0105 汇编语言实践环境搭建

视频（共7个）	教材对应章节
0101 为什么要学汇编语言	
0102 由机器语言到汇编语言	1.1-1.3节
0103 计算机的组成	1.4-1.10节
0104 内存的读写与地址空间	1.11-1.15节
0105 汇编语言实践环境搭建	

寄存器及数据存储

贺利坚　主讲

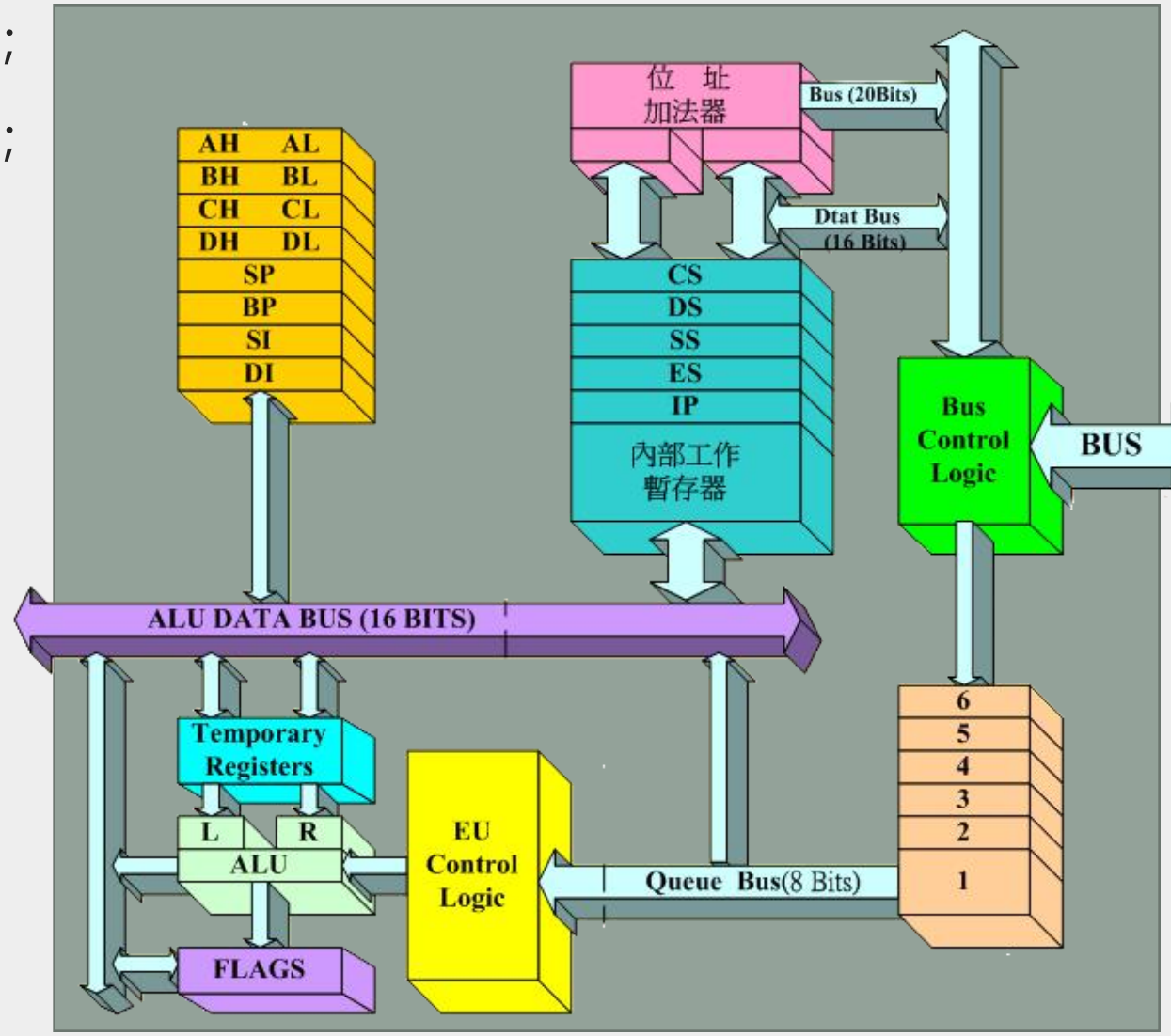


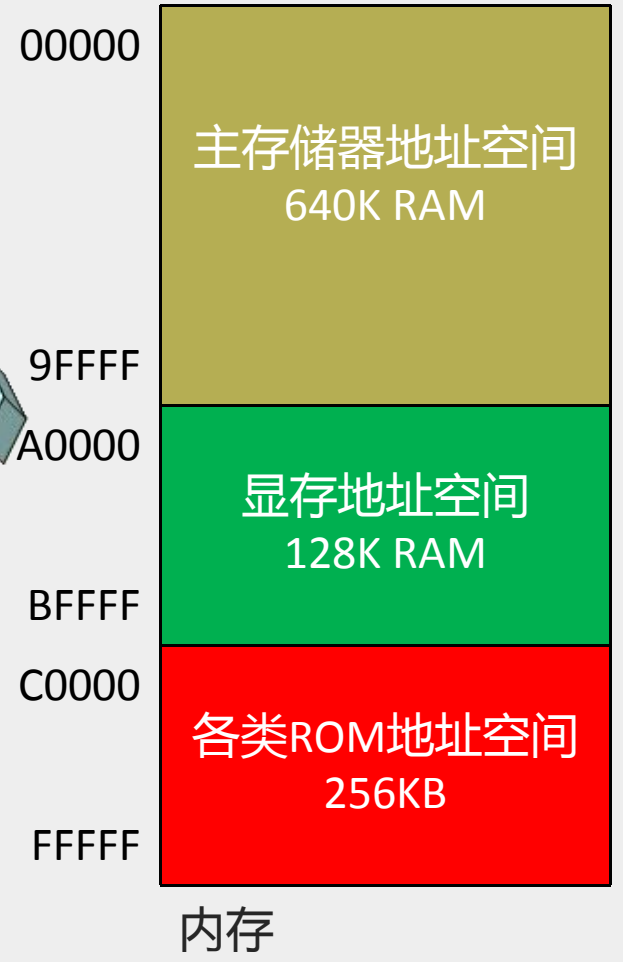
汇编语言程序设计

Assembly Language

CPU的组成

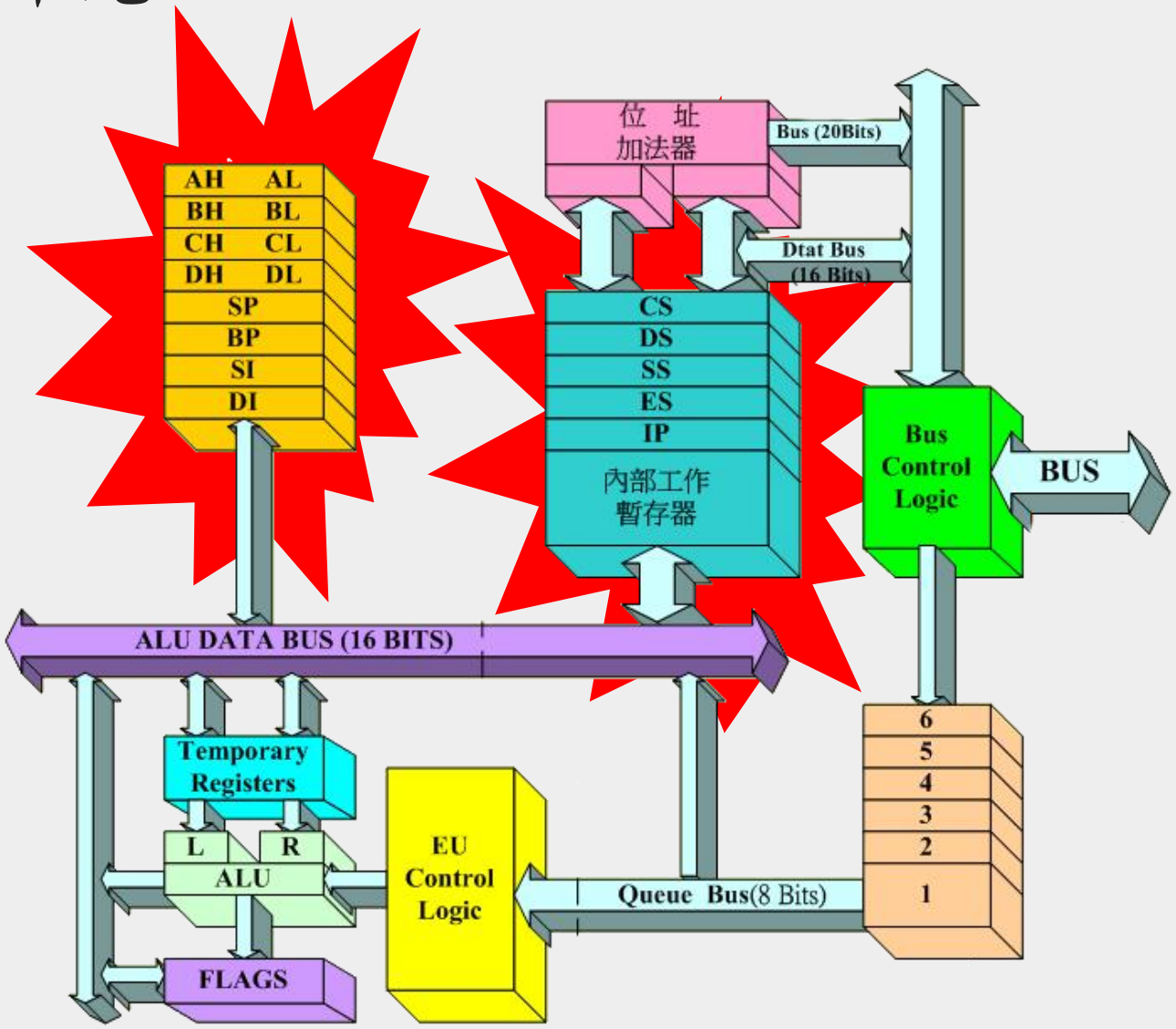
- 运算器进行信息处理；
- 寄存器进行信息存储；
- 控制器协调各种器件进行工作；
- 内部总线实现CPU内各个器件之间的联系。





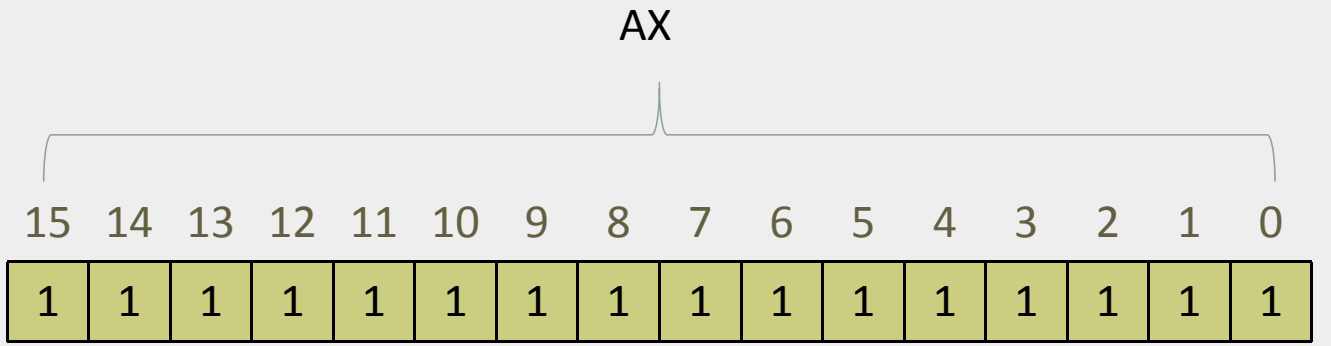
寄存器是CPU内部的信息存储单元

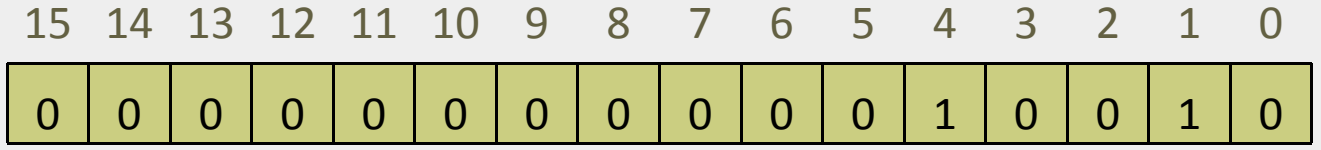
- 8086CPU有14个寄存器：
  - 通用寄存器：AX、BX、CX、DX
  - 变址寄存器：SI、DI
  - 指针寄存器：SP、BP
  - 指令指针寄存器：IP
  - 段寄存器：CS、SS、DS、ES
  - 标志寄存器：PSW
- 共性
  - 8086CPU所有的寄存器都是16位的，可以存放两个字节。

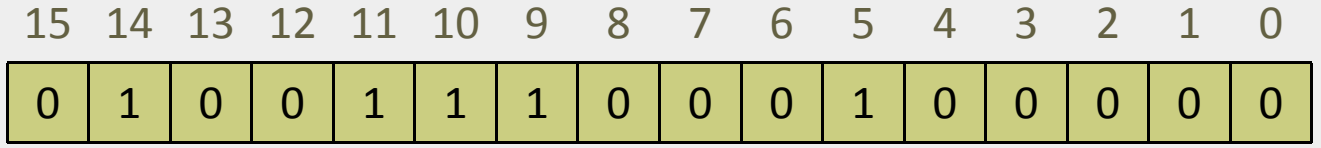


通用寄存器——以AX为例

- 一个16位寄存器存储一个16位的数据
  - 最大值？
  - 2^{16}-1
- 例：在AX中存储18D
  - 18D
    - --- 12H
    - --- 10010B
- 再例：在AX中存储20000D
  - 20000D
    - --- 4E20H
    - --- 0100111000100000B

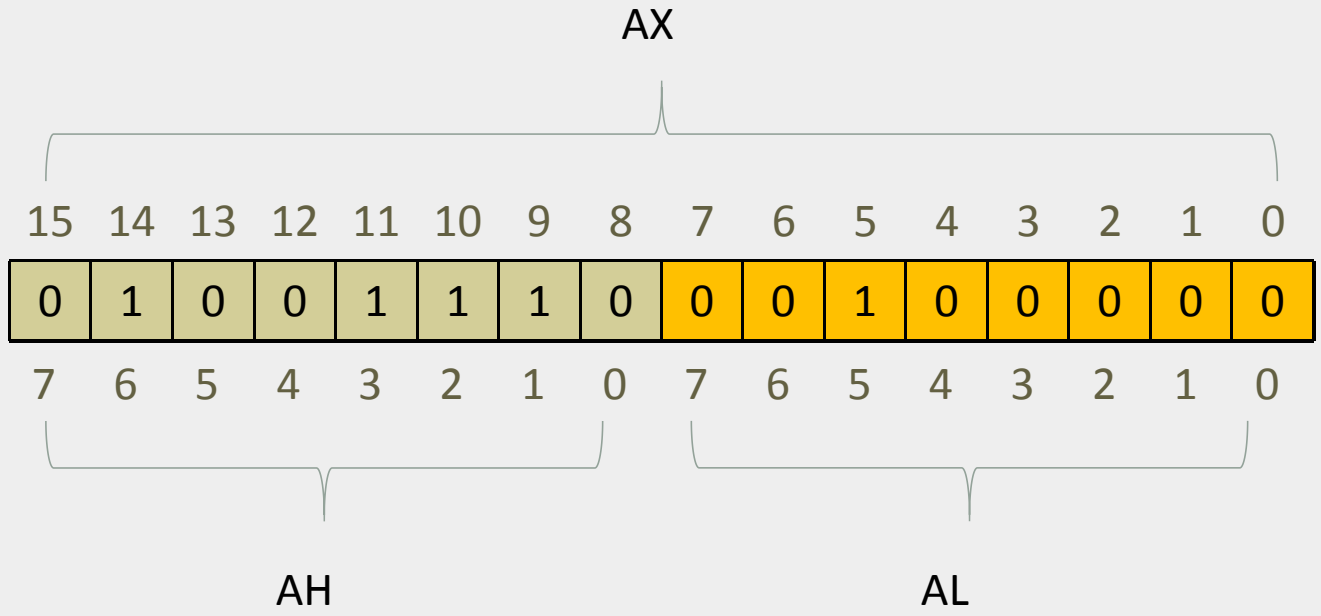






“横看成岭侧成峰“

- 问题
  - 8086上一代CPU中的寄存器都是8位的，如何保证程序的兼容性？
- 方案
  - 通用寄存器均可以分为两个独立的8位寄存器使用
- 细化
  - AX可以分为AH和AL
  - BX可以分为BH和BL
  - CX可以分为CH和CL
  - DX可以分为DH和DL

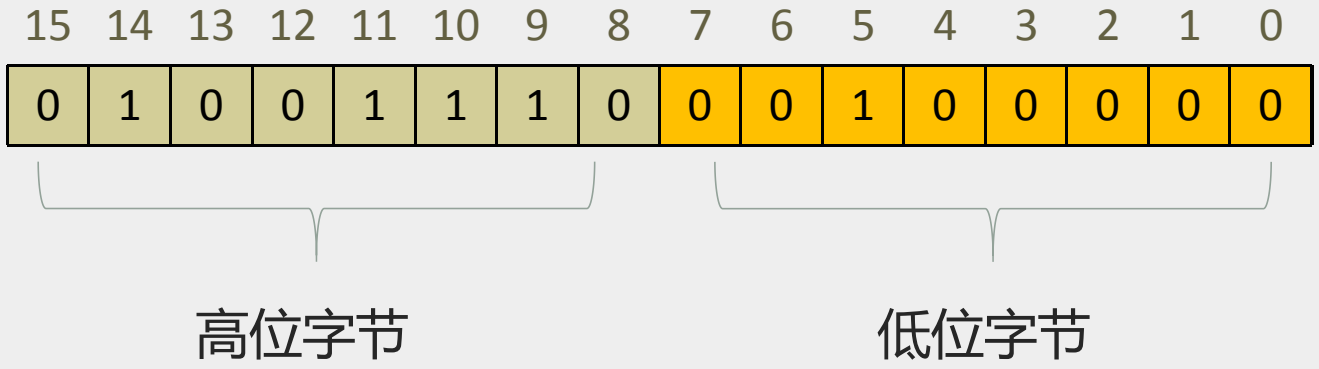


寄存器	寄存器中的数据	所表示的值
AX	0100111000100000	20000（4E20H）
AH	01001110	78（4EH）
AL	00100000	32（20H）

用十六进制可以直观的看出这个数据是由哪些8位数据构成。

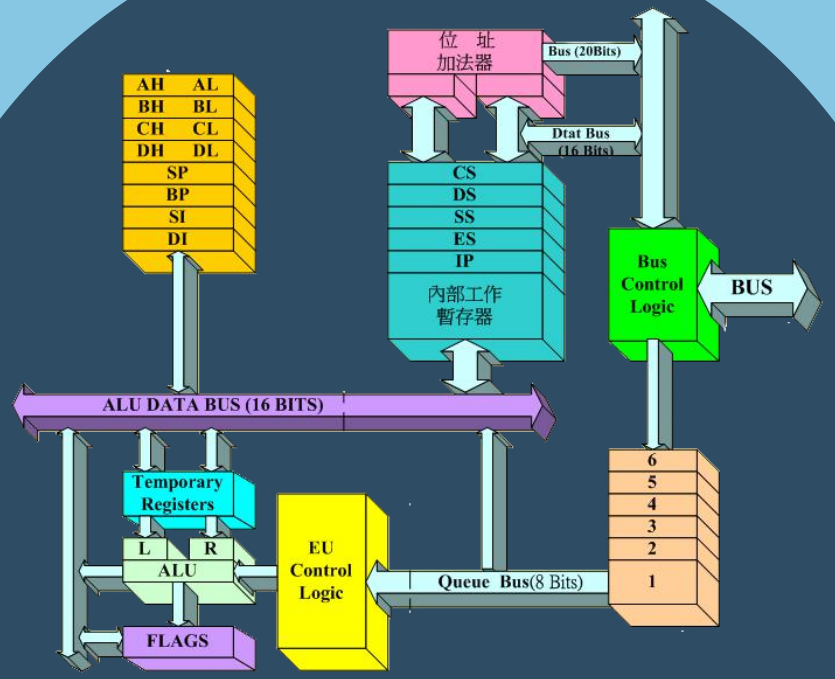
“字”在寄存器中的存储

- 8086是16位CPU
  - 8086的**字长**(word size)为16bit
- 一个**字**(word)可以存在一个16位寄存器中
  - 这个字的高位字节存在这个寄存器的高8位寄存器
  - 这个字的低位字节存在这个寄存器的低8位寄存器



mov和add指令

贺利坚　主讲



汇编语言程序设计

Assembly Language

学习汇编指令——用中学

汇编指令	控制CPU完成的操作	用高级语言的语法描述
mov ax,18	将18送入AX	AX = 18
mov ah,78	将78送入AH	AH = 78
add ax,8	将寄存器AX中的数值加上8	AX = AX + 8
mov ax,bx	将寄存器BX中的数据送入寄存器AX	AX = BX
add ax,bx	将AX,BX 中的内容相加，结果存在AX中	AX = AX + BX

注：汇编指令不区分大小写

写出汇编指令执行的结果(1)

程序段中的指令	指令执行后AX中的数据	指令执行后BX中的数据
mov ax, 4E20H	4E20H	0000H
add ax, 1406H	6226H	0000H
mov bx, 2000H	6226H	2000H
add ax, bx	8226H	2000H
mov bx, ax	8226H	8226H
add ax, bx	044CH	8226H

设原AX、BX中的值均为0000H

写出汇编指令执行的结果(2)

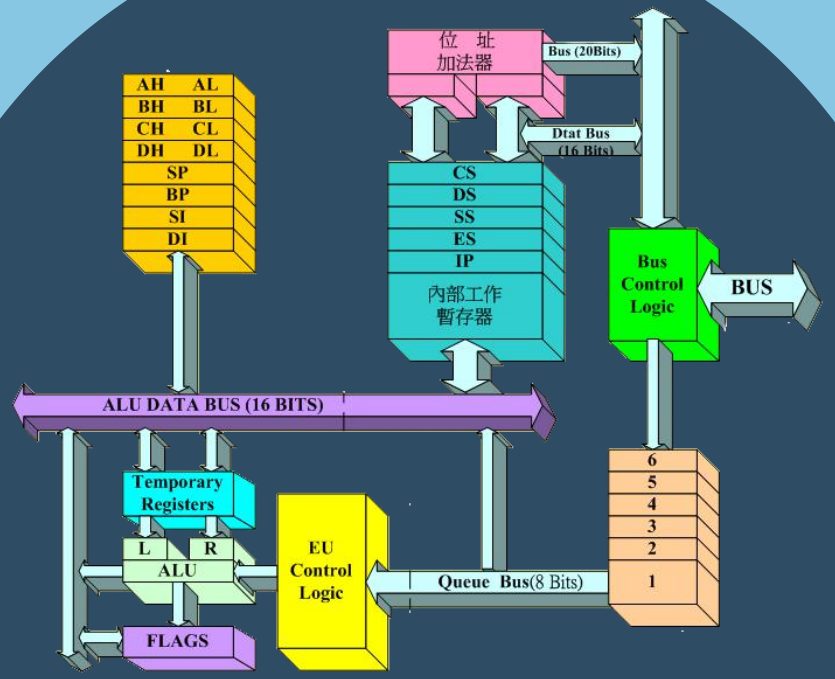
程序段中的指令	指令执行后AX中的数据	指令执行后BX中的数据
mov ax, 001AH	001AH	0000H
mov bx, 0026H	001AH	0026H
add al, bl	0040H	0026H
add ah, bl	2640H	0026H
add bh, al	2640H	4026H
mov ah, 0	0040H	4026H
add al, 85H	00C5H	4026H
add al, 93H	0058H	4026H

设原AX、BX中的值均为0000H

讨论：若执行add ax, 93H，AX结果为0158H

确定物理地址的方法

贺利坚　主讲

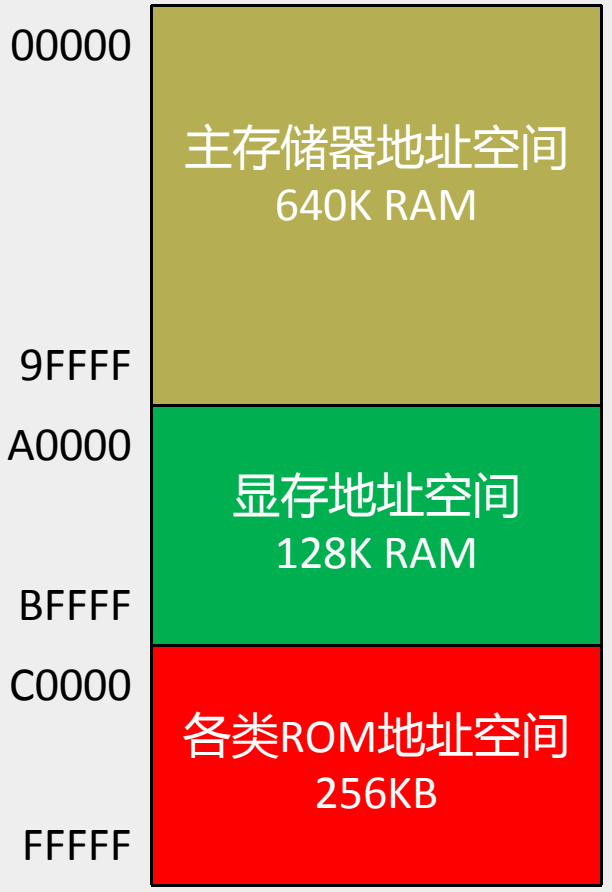


汇编语言程序设计

Assembly Language

物理地址

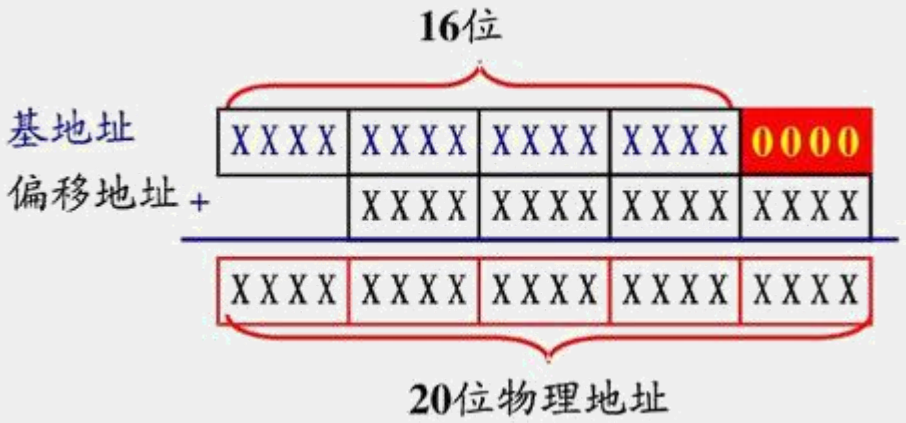
- CPU访问内存单元时要给出内存单元的地址。
- 所有的内存单元构成的存储空间是一个一维的线性空间。
- 每一个内存单元在这个空间中都有唯一的地址，这个唯一的地址称为**物理地址**。
- 事实
  - 8086有20位地址总线，可传送20位地址，**寻址能力为1M**。
  - 8086是16位结构的CPU
    - 运算器一次最多可以处理16位的数据，寄存器的最大宽度为16位。
    - 在8086内部处理的、传输、暂存的地址也是16位，**寻址能力也只有64KB**！
- 问题：8086如何处理在寻址空间上的这个矛盾？！



8086CPU对应的内存

8086CPU给出物理地址的方法

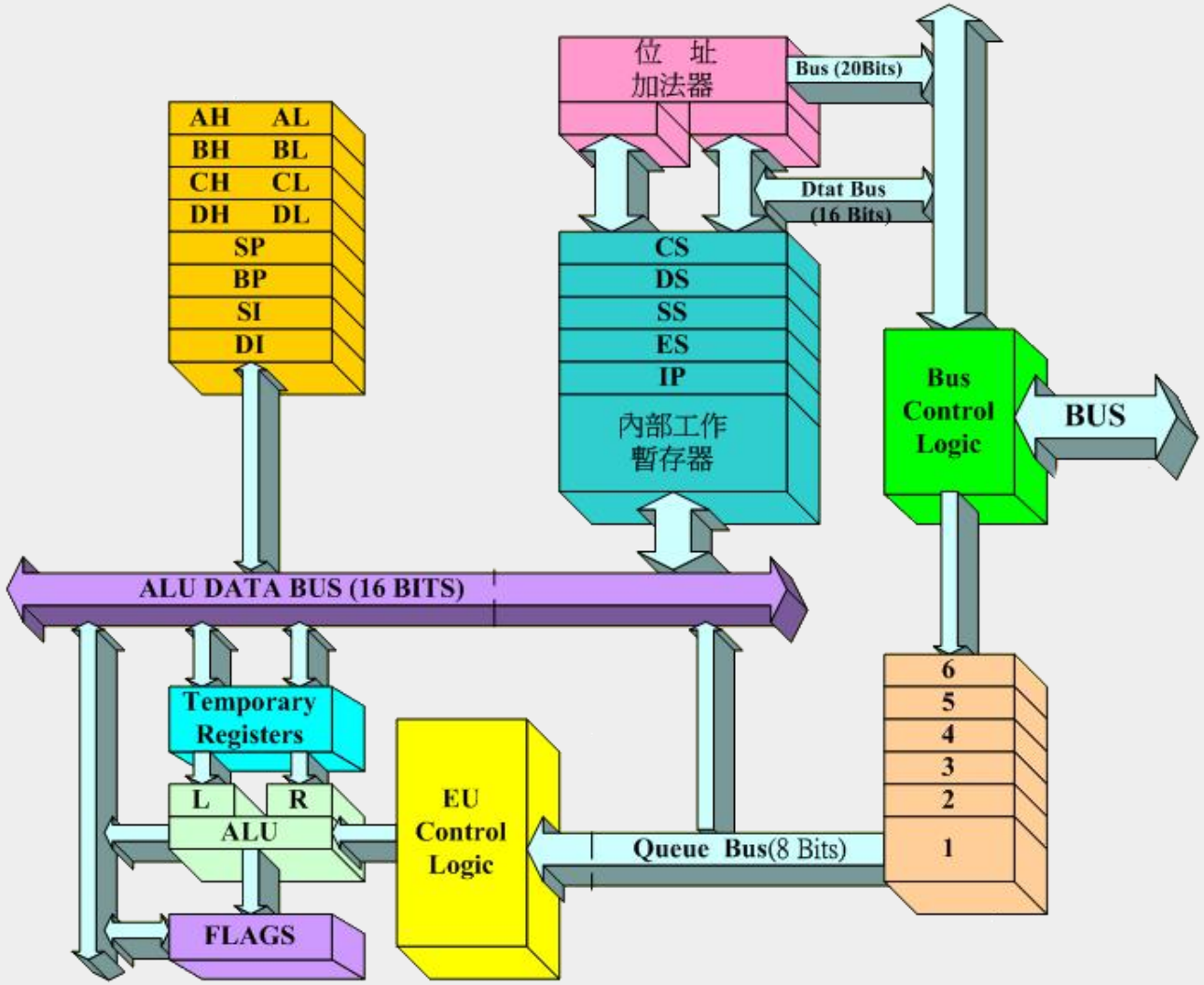
- 8086CPU的解决方法
  - 用两个16位地址(段地址、偏移地址)合成一个20位的物理地址。
- 地址加法器合成物理地址的方法
  - 物理地址=段地址×16+偏移地址



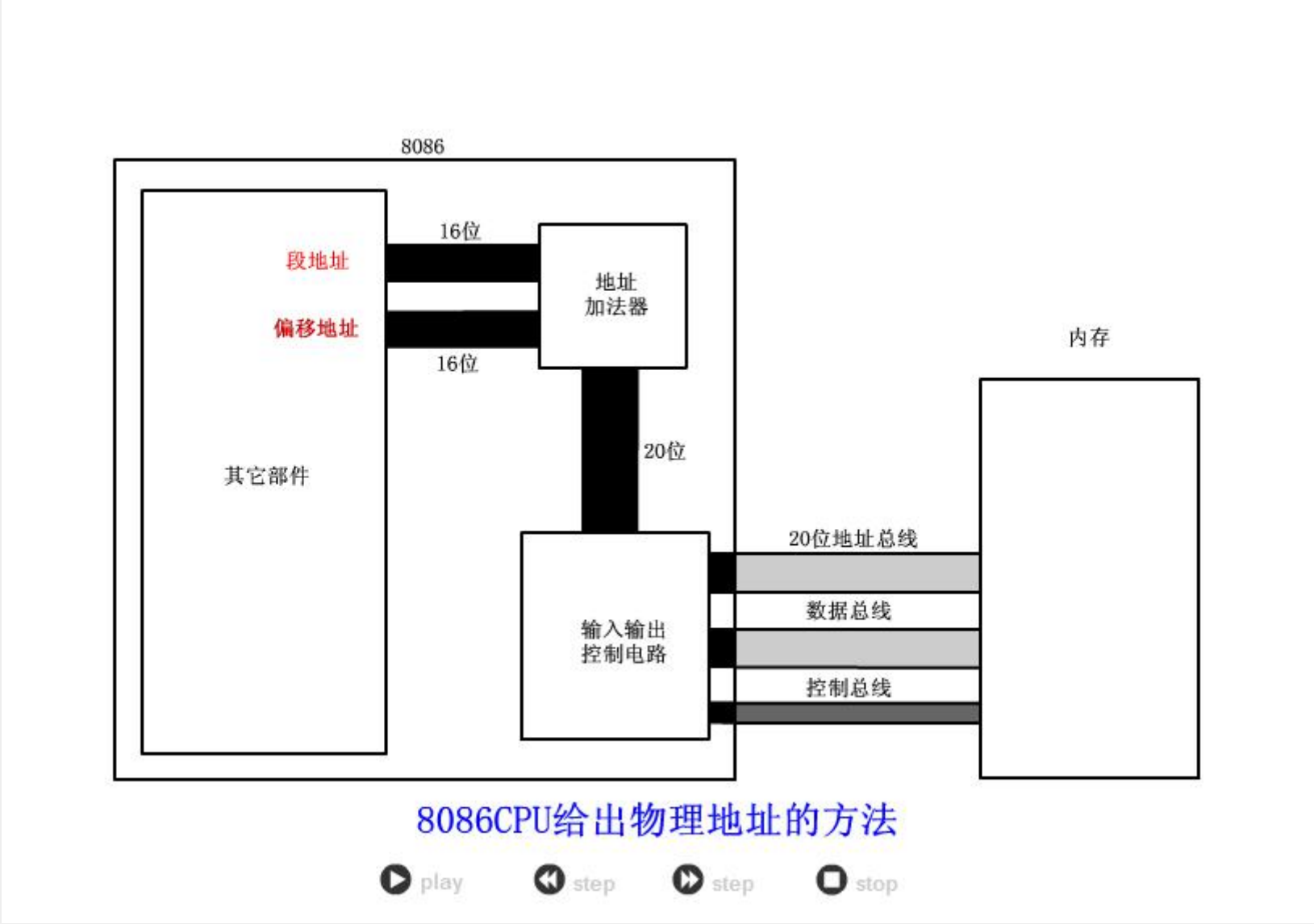
例：

段地址： 1230
\+ 偏移地址： 00C8

物理地址：123C8

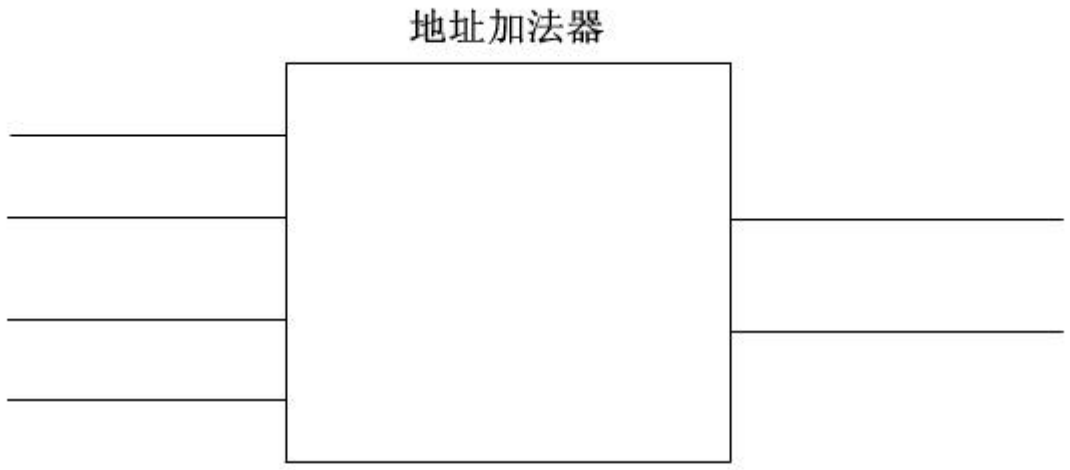


演示：物理地址=段地址×16+偏移地址



例：8086CPU访问地址为123C8H的内存单元

方法：物理地址=段地址×16+偏移地址




```
  段地址：  1230
+ 偏移地址：  00C8
----------------
  物理地址：123C8
```

思考：段地址是123CH，可否？

```
  段地址：  123C
+ 偏移地址：  0008
----------------
  物理地址：123C8
```

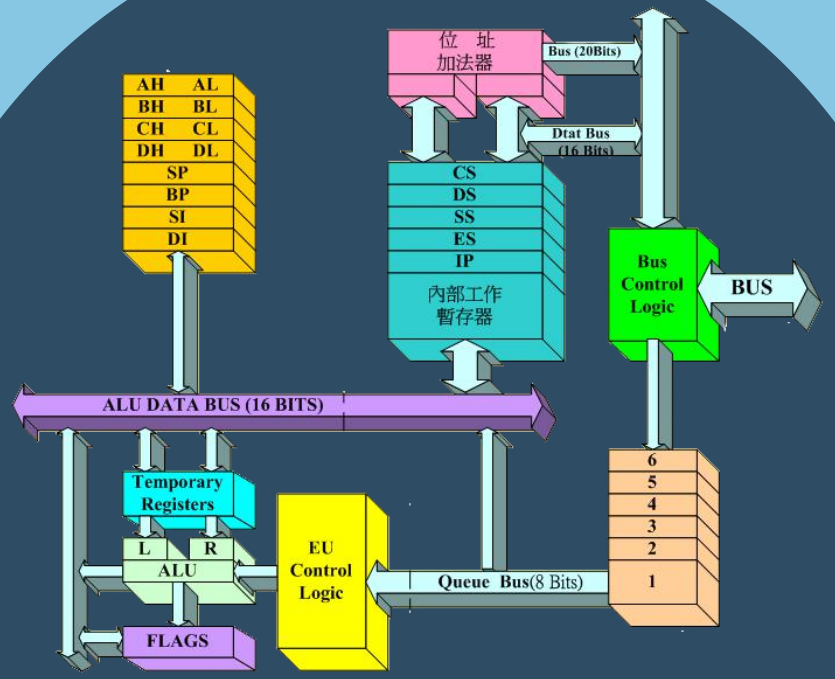
```
  段地址：  123B
+ 偏移地址：  0018
----------------
  物理地址：123C8
```

“段地址×16+偏移地址=物理地址”的本质含义

- 要解决的问题
  - 用两个16位的地址（段地址、偏移地址），相加得到一个20位的物理地址
- 本质含义
  - CPU在访问内存时，用一个基础地址（段地址×16）和一个相对于基础地址的偏移地址相加，给出内存单元的物理地址。

内存的分段表示法

贺利坚　主讲

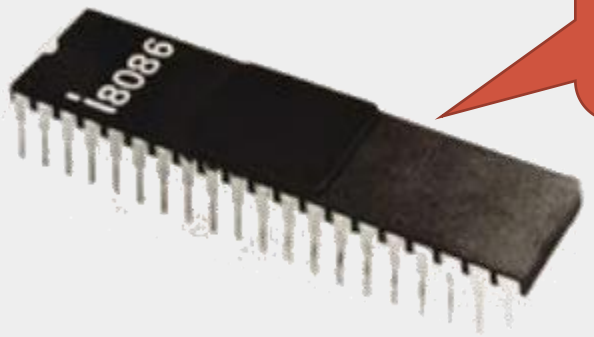


汇编语言程序设计

Assembly Language

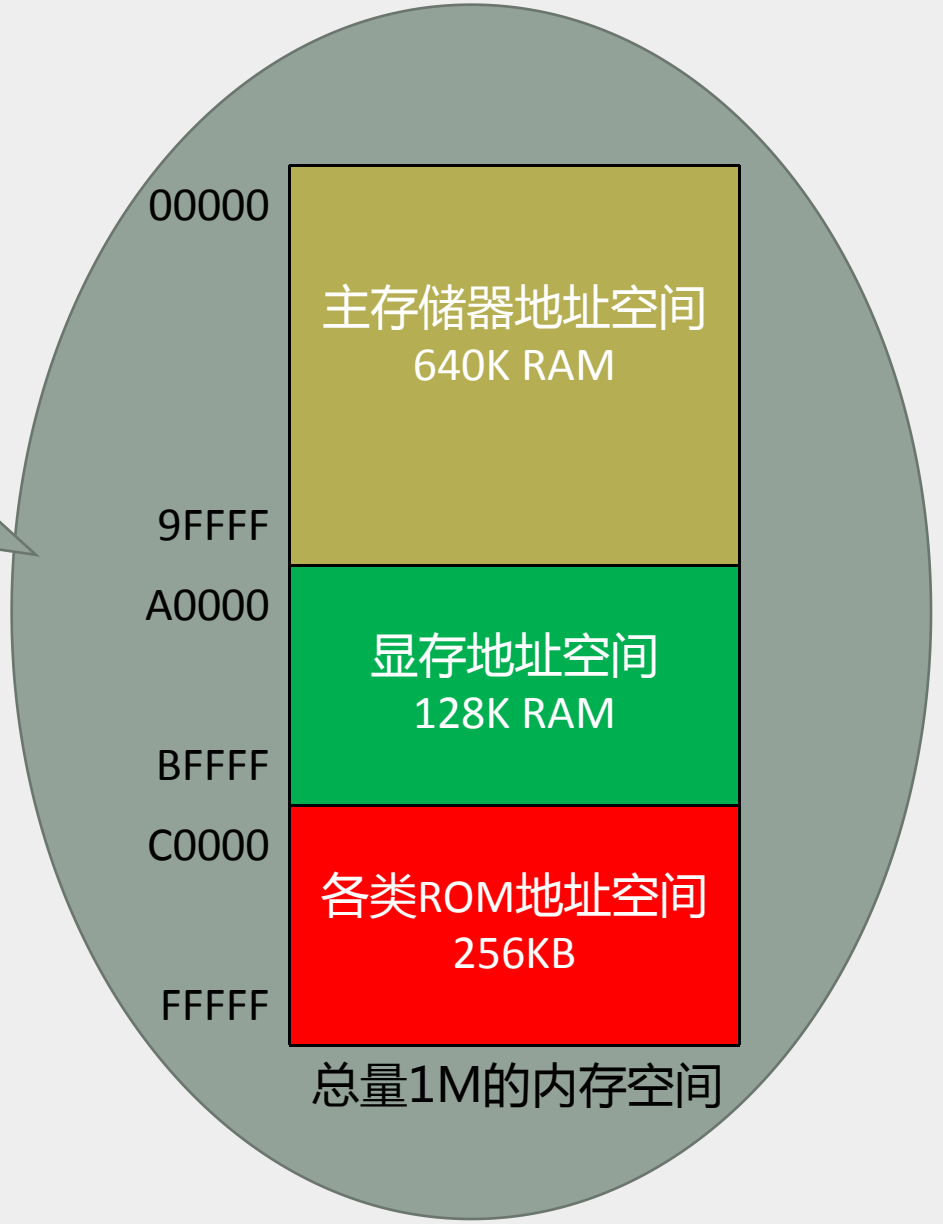
用分段的方式管理内存

- 8086CPU用“（段地址×16）+偏移地址=物理地址”的方式给出内存单元的物理地址。
- **内存并没有分段，段的划分来自于CPU！！！**

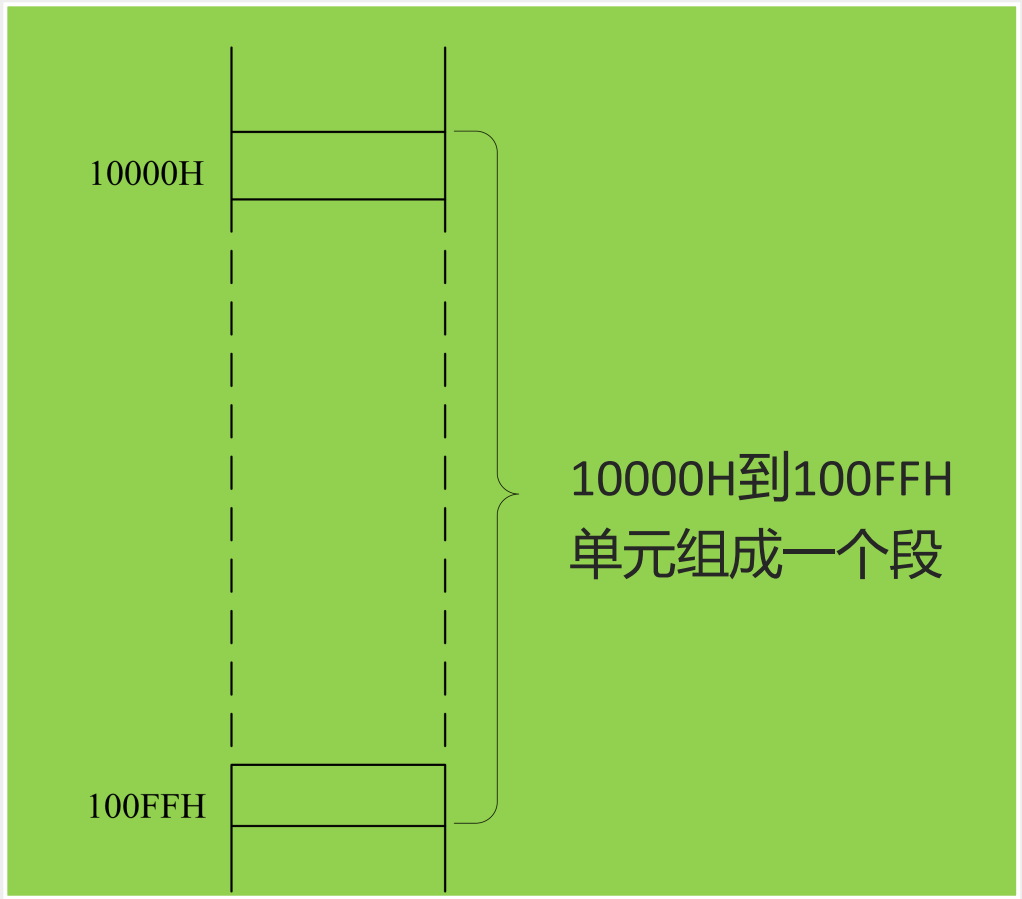


我本为一体，分段不分段，随意！

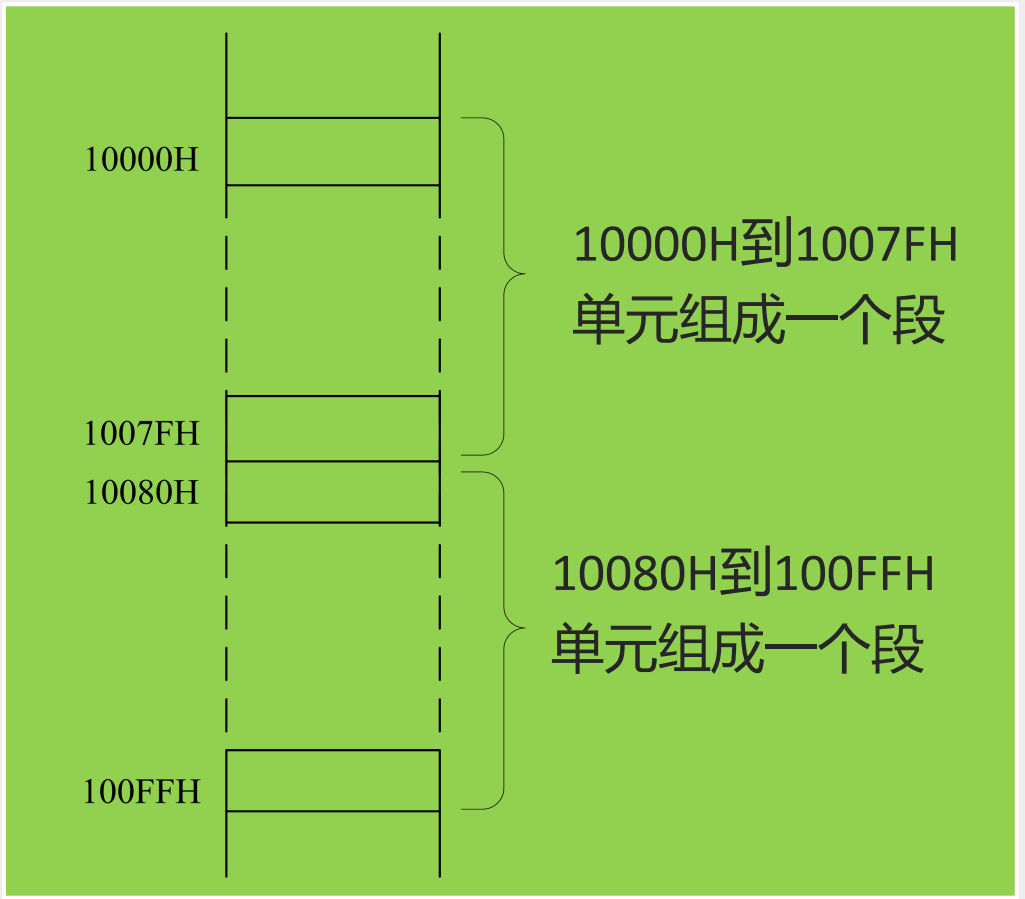
一分段靠我，依着我的方便来！



同一段内存，多种分段方案



起始地址（基础地址）为10000H，
段地址为1000H，大小为100H



起始地址（基础地址）为10000H和10080H，
段地址为1000H和1008H，大小均为80H

其他分法...

（1）段地址×16 必然是 16的倍数，所以一个段的起始地址也一定是16的倍数；
（2）偏移地址为16位，16 位地址的寻址能力为 64K，所以一个段的长度最大为64K。

用不同的段地址和偏移地址形成同一个物理地址

物理地址	段地址	偏移地址
21F60H	2000H	**1F60H**
	2100H	**0F60H**
	21F0H	**0060H**
	21F6H	**0000H**
	1F00H	**2F60H**

- 偏移地址16位，变化范围为0~FFFFH，用偏移地址最多寻址64KB。
- 例：给定段地址2000H，用偏移地址寻址的范围是：20000H~2FFFFFH，共64K

在8086PC机中存储单元地址的表示方法

例：数据在21F60H内存单元中，段地址是2000H，说法

（a）数据存在内存**2000:1F60**单元中；

（b）数据存在内存的2000H段中的1F60H单元中。

段地址很重要！——用专门的寄存器存放段地址。

4个段寄存器：

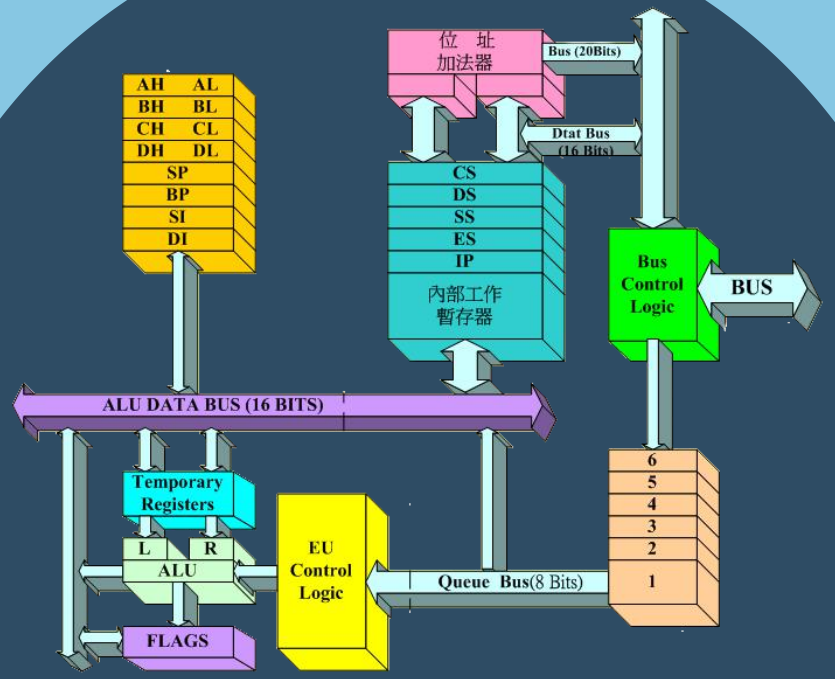
CS - 代码段寄存器　　DS - 数据段寄存器

SS - 栈段寄存器　　ES - 附加段寄存器

偏移地址可以用多种方法提供——8086丰富的取址方式。

Debug的使用

贺利坚　主讲



汇编语言程序设计

Assembly Language

Debug是什么？

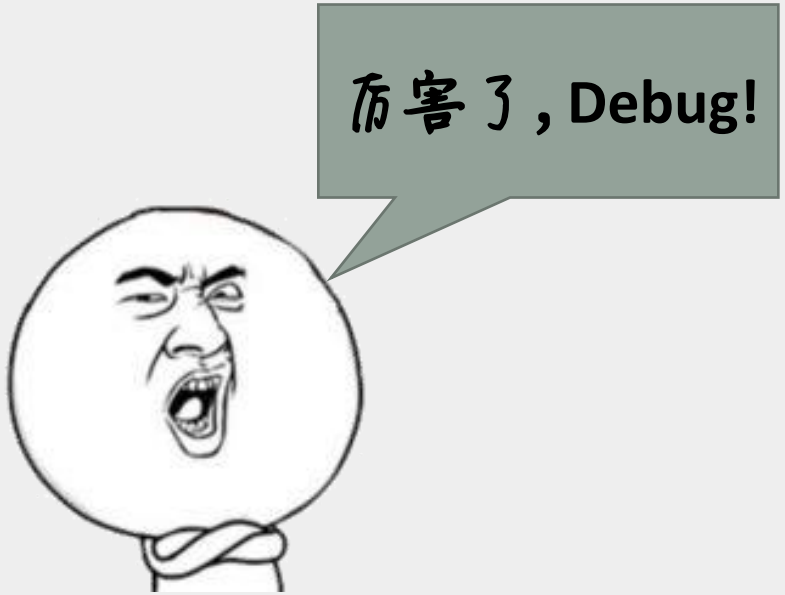
- Debug是DOS系统中的著名的调试程序，也可以运行在windows系统实模式下。
- 使用Debug程序，可以查看CPU各种寄存器中的内容、内存的情况，并且在机器指令级跟踪程序的运行！
- Debug就是传奇！

DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DEB...

```
073F:0112 01D8          ADD     AX,BX
-t

AX=8642  BX=4321  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0114   OV UP EI NG NZ NA PE NC
073F:0114 0000          ADD     [BX+SI],AL                      DS:4321=00
-
 ^ Error
-q

C:\>debug
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 B82301        MOV     AX,0123
-d
073F:0100  B8 23 01 BB 03 00 89 D8-01 D8 B8 21 43 BB FF FF    .#.........!C...
073F:0110  89 C3 01 D8 00 00 00 00-00 00 00 00 34 00 2E 07    ............4...
073F:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
073F:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
073F:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
073F:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
073F:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
073F:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
-_
```

Debug能做什么？

- 用R命令查看、改变CPU寄存器的内容
- 用D命令查看内存中的内容
- 用E命令改变内存中的内容
- 用U命令将内存中的机器指令翻译成汇编指令
- 用A命令以汇编指令的格式在内存中写入机器指令
- 用T命令执行机器指令
- ....



启动Debug

- 在DOS提示符下输入命令：debug

用R命令查看、改变CPU寄存器的内容

- R - 查看寄存器内容
- R 寄存器名 - 改变指定寄存器内容

用D命令查看内存中的内容

- D - 列出预设地址内存处的128个字节的内容
- D 段地址:偏移地址 - 列出内存中指定地址处的内容
- D 段地址:偏移地址 结尾偏移地址 - 列出内存中指定地址范围内的内容

用E命令改变内存中的内容

- E 段地址:偏移地址 数据1 数据2 ...
- E 段地址:偏移地址
  - 逐个询问式修改
  - 空格 - 接受，继续
  - 回车 - 结束

用U命令将内存中的机器指令翻译成汇编指令

- 有汇编指令
  - mov ax, 0123H
  - mov bx, 0003H
  - mov ax, bx
  - add ax, bx
- 对应的机器码为
  - B8 23 01
  - BB 03 00
  - 89 D8
  - 01 D8
- e 地址 数据 - 写入
- d 地址 - 查看
- u 地址 - 查看代码

用A命令以汇编指令的格式在内存中写入机器指令

- 有汇编指令
  - mov ax, 0123H
  - mov bx, 0003H
  - mov ax, bx
  - add ax, bx
- 对应的机器码为
  - B8 23 01
  - BB 03 00
  - 89 D8
  - 01 D8
- a 地址 - 写入汇编指令
- d 地址 - 查看数据
- u 地址 - 查看代码

用T命令执行机器指令

- t - 执行CS:IP处的指令

  mov ax, 0123H

  mov bx, 0003H

  mov ax, bx

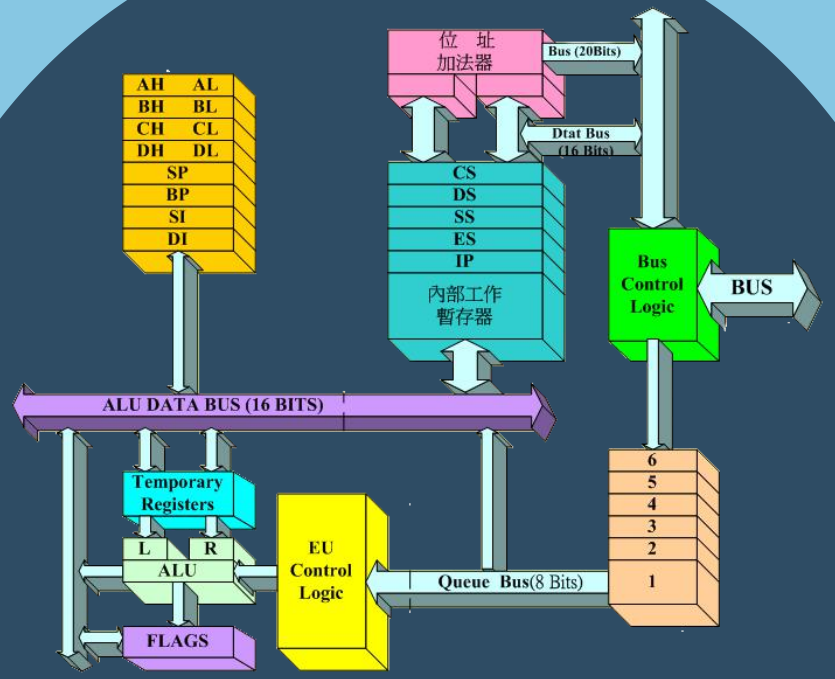
  add ax, bx

用Q命令退出Debug

- q - 退出Debug

CS、IP与代码段

贺利坚　主讲

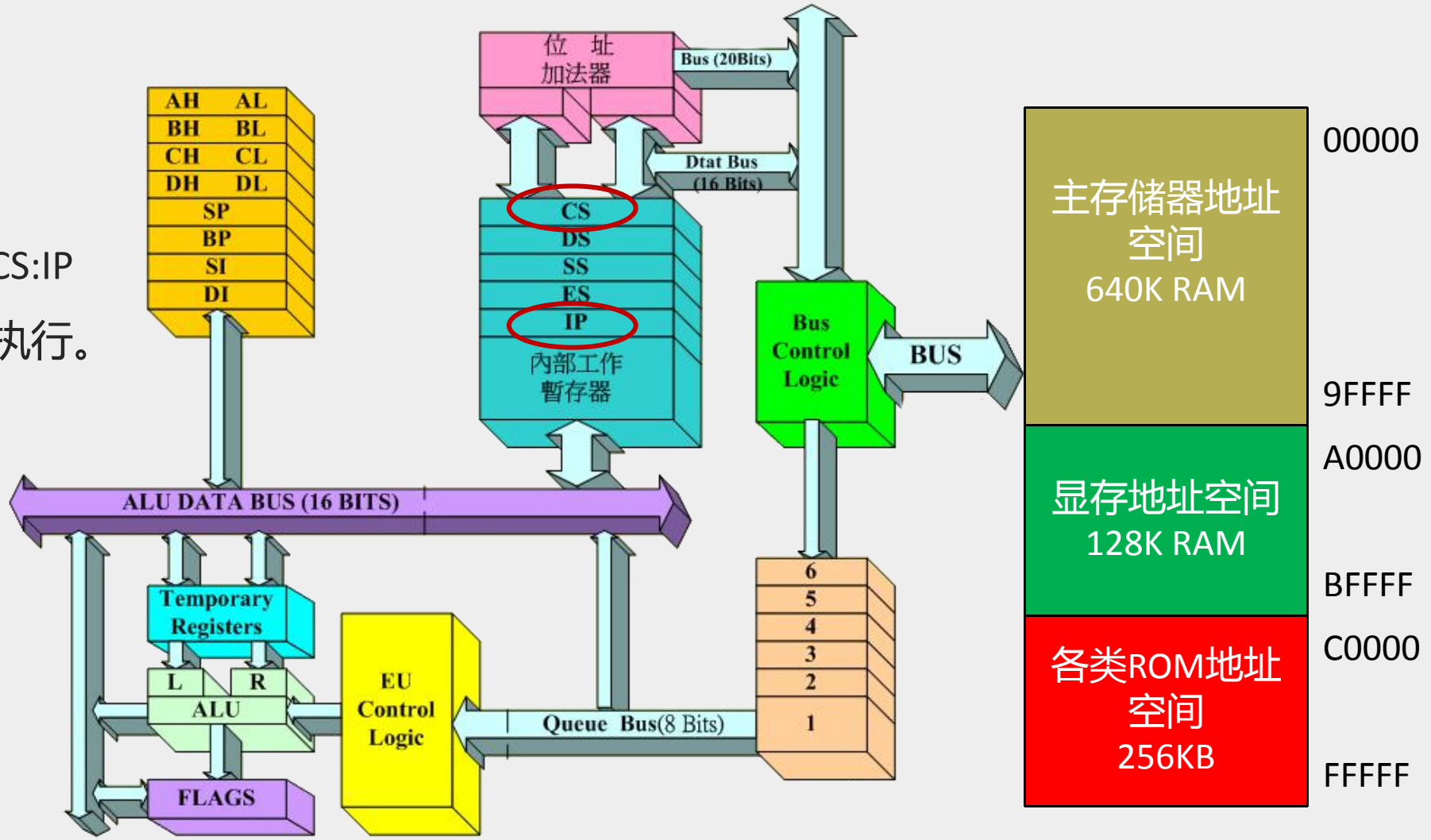


汇编语言程序设计

Assembly Language

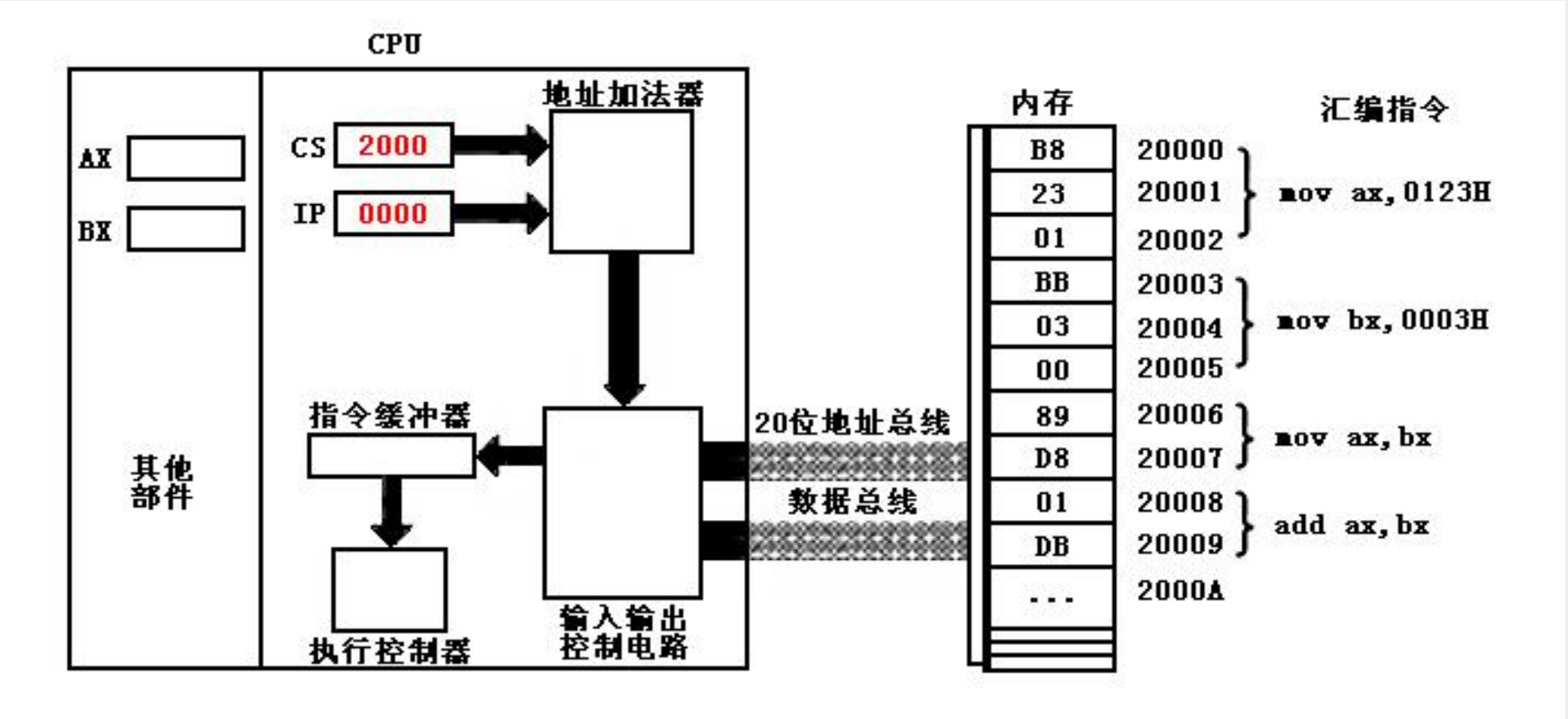
两个关键的寄存器

- CS：代码段寄存器
- IP： 指令指针寄存器
- CS:IP：CPU将内存中CS:IP指向的内容当作指令执行。



例示：在CS和IP指示下代码的执行

- 8086CPU当前状态：CS中内容为2000H，IP中内容为0000H
- 内存20000H~20009H处存放着可执行的机器代码



指令的执行过程...

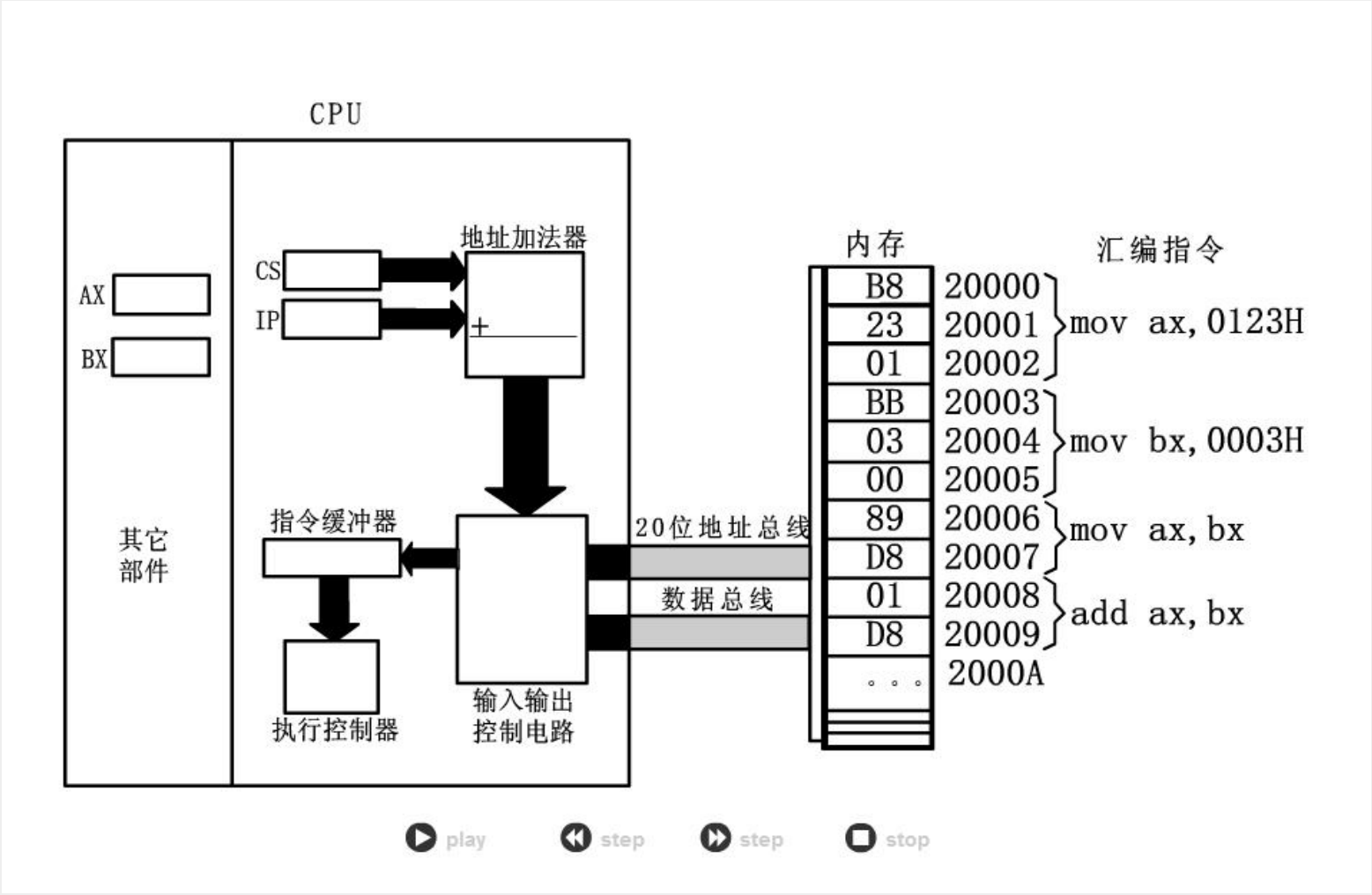
8086PC读取和执行指令演示

💻8086PC工作过程的简要描述：

（1）从CS:IP指向内存单元读取指令，读取的指令进入指令缓冲器；

（2）IP = IP + 所读取指令的长度，从而指向下一条指令；

（3）执行指令。 转到步骤（1），重复这个过程。



指令读取和执行的实证演示-Debug

- 用debug程序执行下面的代码

  mov ax, 0123H

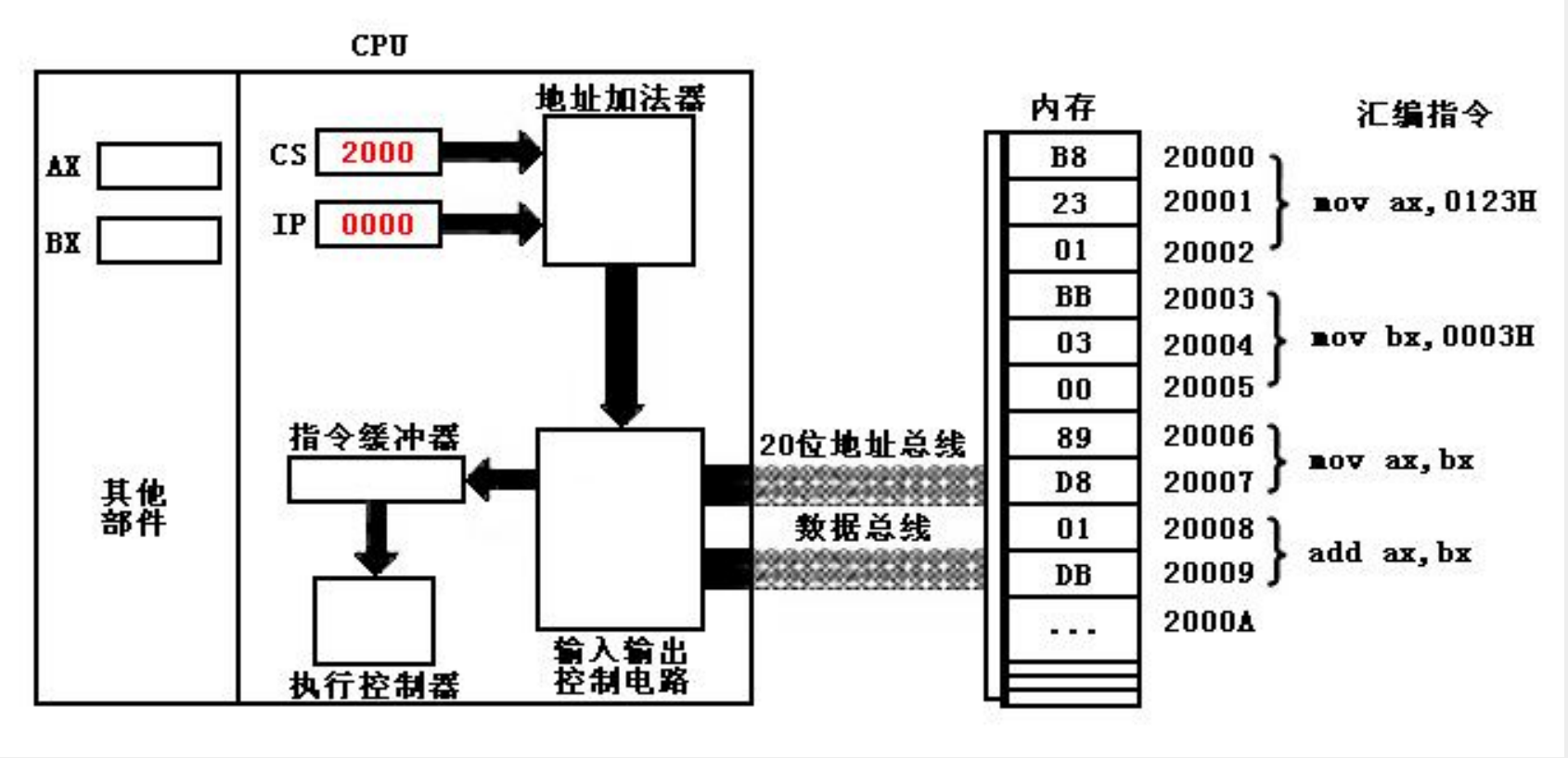
  mov bx, 0003H

  mov ax, bx

  add ax, bx

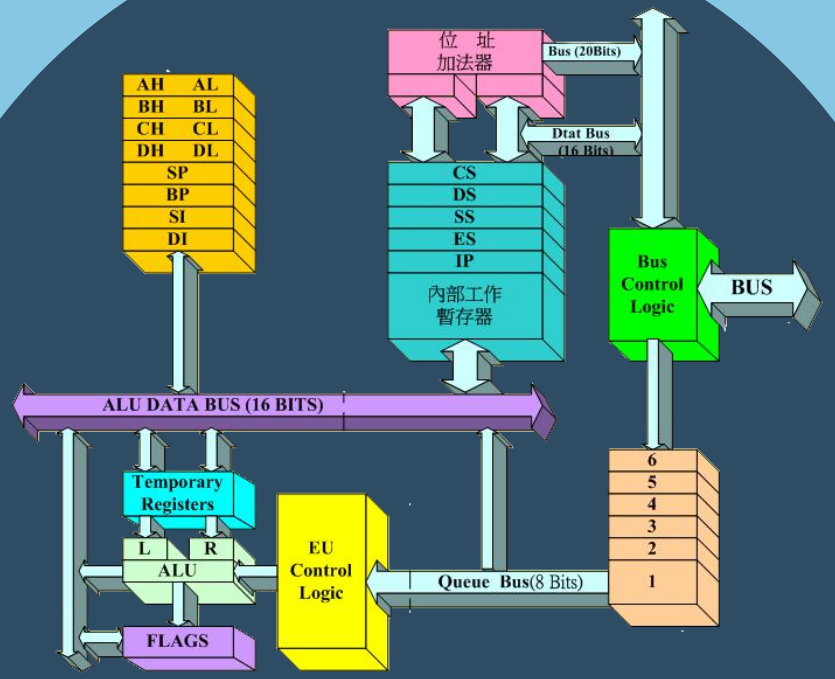
- a 地址 - 写入汇编指令
- u 地址 - 查看代码
- t - 执行CS:IP处代码

问：内存中有数据 B8 23 01 BB 03 00 89 D8 01 D8，
究竟用作一般数据，还是用作指令？
答：CPU将CS:IP指向的内存单元中的内容看作指令！



jmp指令

贺利坚　主讲



汇编语言程序设计

Assembly Language

修改CS、IP的指令

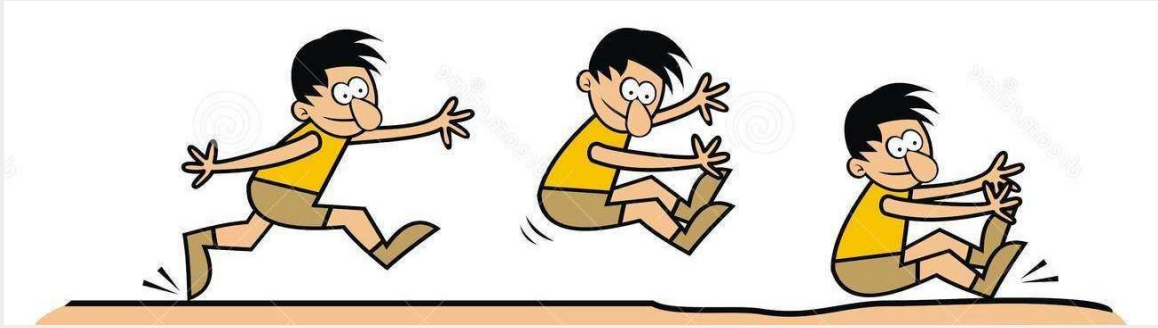
- 事实：执行何处的指令，取决于CS:IP
- 应用：可以通过改变CS、IP中的内容，来控制CPU要执行的目标指令
- 问题：如何改变CS、IP的值？
- 方法1：Debug 中的 R 命令可以改变寄存器的值——rcs, rip
  - Debug是调试手段，并非程序方式！
- 方法2：用指令修改

```
mov cs, 2000H
mov ip, 0000H
```

8086CPU不提供对CS和IP修改的指令！

- 方法3：转移指令 jmp

转移指令 jmp



- 同时修改CS、IP的内容

  jmp 段地址：偏移地址

  jmp 2AE3:3

  jmp 3:0B16

  功能：用指令中给出的段地址修改CS，偏移地址修改IP。

- 仅修改IP的内容

  jmp 某一合法寄存器

  jmp ax （类似于 mov IP, ax）

  jmp bx

  功能：用寄存器中的值修改IP。

问题分析

从20000H开始，执行的序列是：

（1）mov ax,6622

（2）jmp 1000:3

（3）mov ax,0000

（4）mov bx,ax

（5）jmp bx

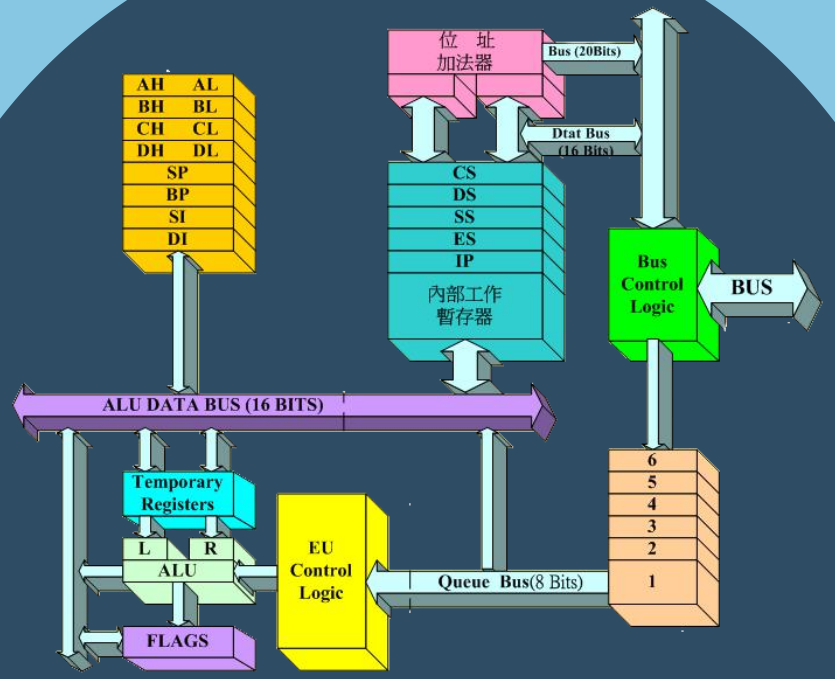
（6）mov ax,0123H

（7）转到第（3）步执行

CS	2000	IP	0000		
AX		BX		CX	

内存中字的存储

贺利坚　主讲

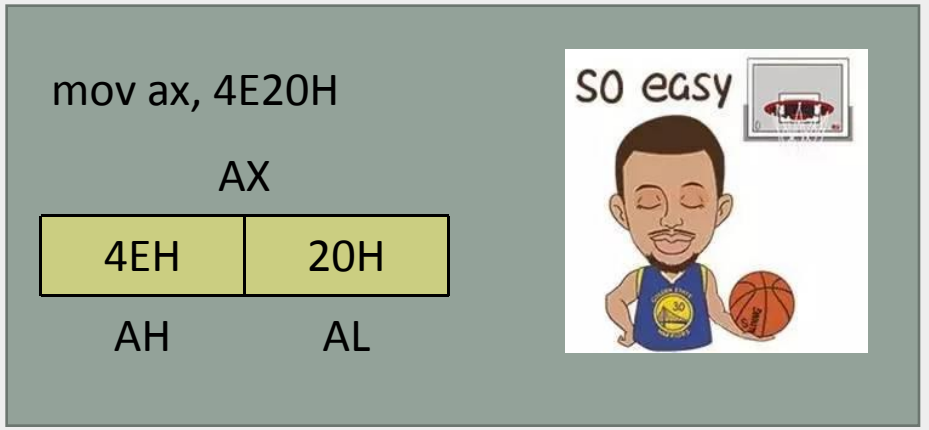


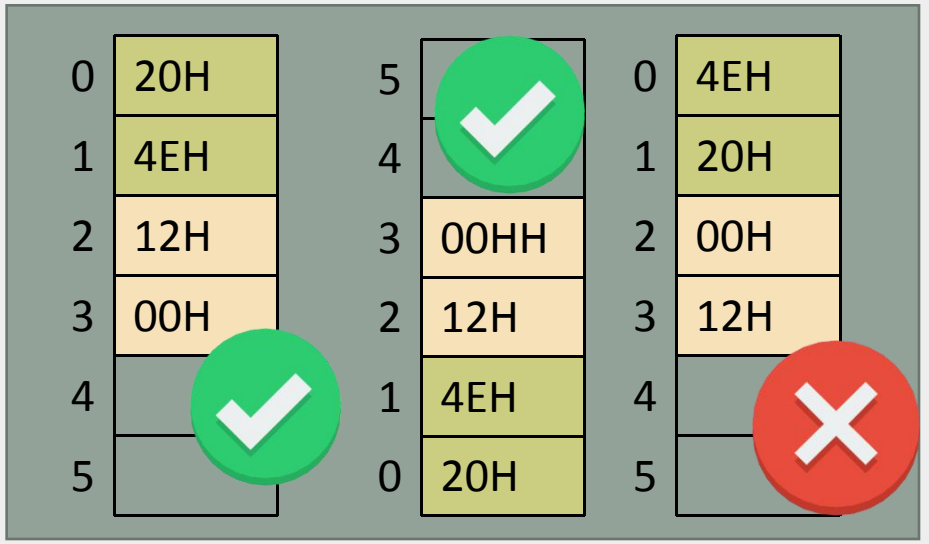
汇编语言程序设计

Assembly Language

内存中字的存储

- 事实：对8086CPU，16位作为一个字
- 问题
  - 16位的字存储在一个16位的寄存器中，如何存储？
- 回答
  - 高8位放高字节，低8位放低字节
- 问题
  - 16位的字在内存中需要2个连续字节存储，怎么存放？
- 回答
  - 低位字节存在低地址单元，高位字节存在高地址单元
  - 例：20000D（4E20H）存放0、1两个单元，18D（0012H）存放在2、3两个单元

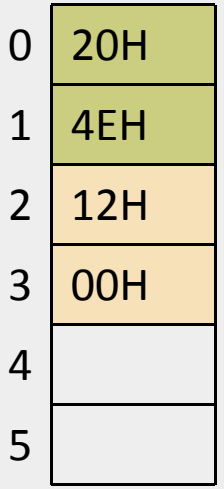




0号是低地址单元，1号是高地址单元。

字单元

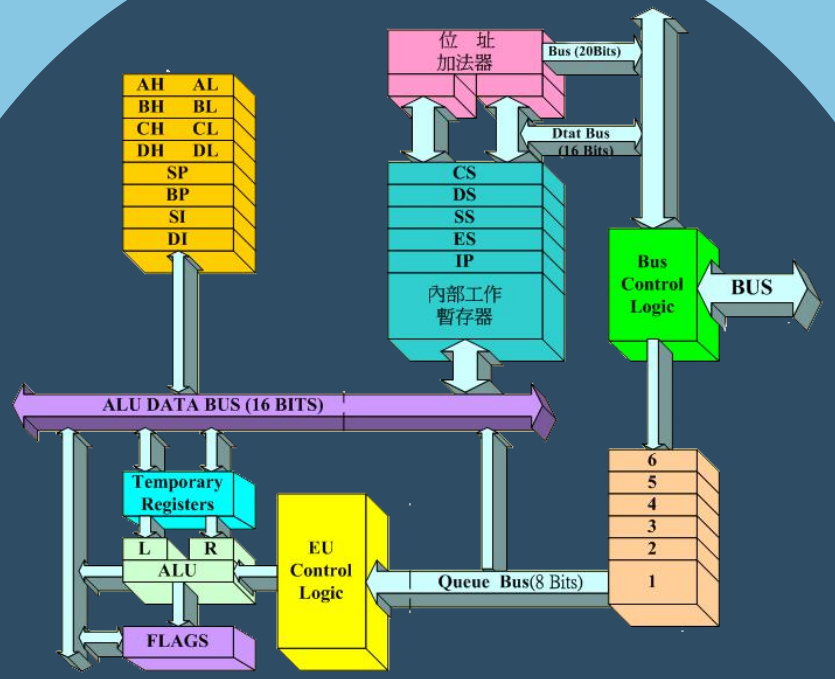
- 字单元：由两个地址连续的内存单元组成，存放一个字型数据（16位）
- 原理：在一个字单元中，低地址单元存放低位字节，高地址单元存放高位字节
  - 在起始地址为0的单元中，存放的是4E20H
  - 在起始地址为2的单元中，存放的是0012H



- 问题：
  - （1）0地址单元中存放的字节型数据是（ 20H ）
  - （2）0地址字单元中存放的字型数据是（ 4E20H ）
  - （3）2地址单元中存放的字节型数据是（ 12H ）
  - （4）2地址字单元中存放的字型数据是（ 0012H ）

用DS和[address]实现字的传送

贺利坚　主讲



汇编语言程序设计

Assembly Language

要解决的问题：CPU从内存单元中要读取数据

- 要求
  - CPU要读取一个内存单元的时候，必须先给出这个内存单元的地址；
- 原理
  - 在8086PC中，内存地址由段地址和偏移地址组成（段地址:偏移地址）
- 解决方案：DS和[address]配合
  - 用 DS寄存器存放要访问的数据的段地址
  - 偏移地址用[...]形式直接给出

- 例1

```
mov bx,1000H
mov ds,bx
mov al, [0]
```

将10000H(1000:0)中的数据读到al中

- 例2

```
mov bx,1000H
mov ds,bx
mov [0],al
```

将al中的数据写到10000H(1000:0)中

- 将段地址送入DS的两种方式

(1) mov ds, 1000H 

(2) mov bx, 1000H

mov ds, bx 

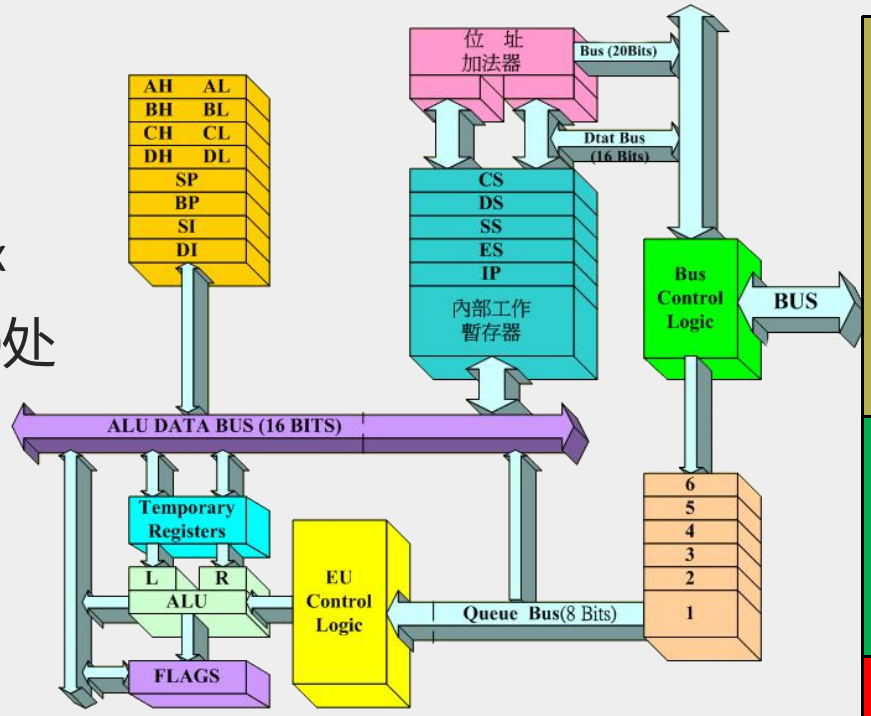
- 8086CPU不支持将数据直接送入段寄存器（硬件设计的问题）
- 套路：数据➔一般的寄存器➔段寄存器

字的传送

- 8086CPU可以一次性传送一个字(16位的数据)
- 例

```
mov bx, 1000H
mov ds, bx
mov ax, [0]   ;1000:0处的字型数据送入ax
mov [0], cx   ;cx中的16位数据送到1000:0处
```

10000H	23
10001H	11
10002H	22
10003H	66



案例1

💻内存

10000H	23
10001H	11
10002H	22
10003H	66

💻指令

```
mov ax, 1000H
mov ds, ax
mov ax, [0]
mov bx, [2]
mov cx, [1]
add bx, [1]
add cx, [2]
```

```
AX=         BX=         CX=         DX=         SP=         BP=         SI=         DI=
DS=         ES=         SS=         CS=         IP=          NV  UP  EI NG  NZ  NA  PO  NC
```

案例2

💻内存

10000H	23
10001H	11
10002H	22
10003H	11

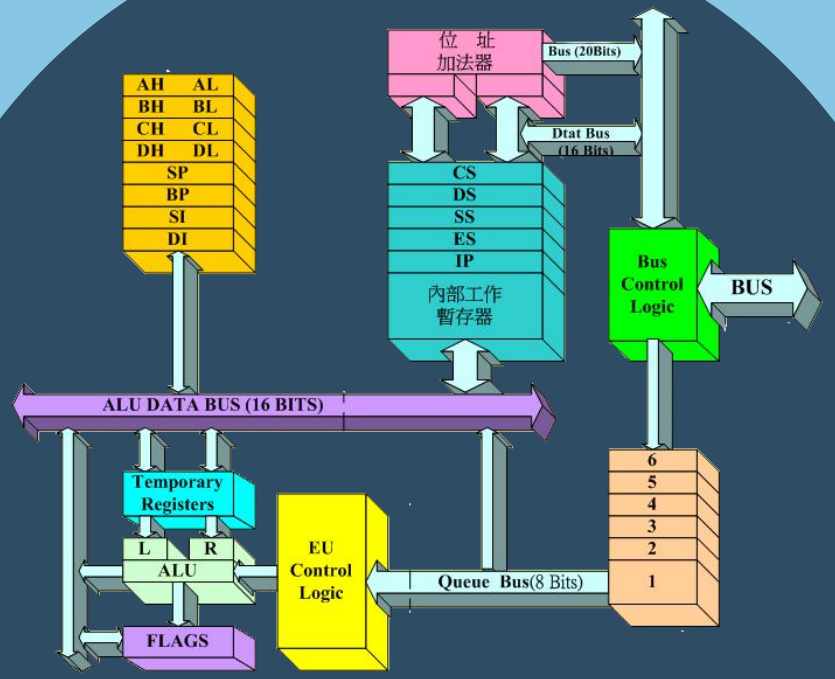
💻指令

```
mov ax, 1000H
mov ds, ax
mov ax, 2C31
mov [0], ax
mov bx, [0]
sub bx, [2]
mov [2], bx
```

```
AX=          BX=          CX=          DX=          SP=          BP=          SI=          DI=
DS=          ES=          SS=          CS=          IP=           NV  UP  EI NG  NZ  NA  PO  NC
```

DS与数据段

贺利坚　主讲



汇编语言程序设计

Assembly Language

对内存单元中数据的访问

- 对于8086PC机，可以根据需要将一组内存单元定义为一个段。
  - 物理地址=段地址×16+偏移地址
  - 将一组长度为N（N≤64K）、地址连续、起始地址为16的倍数的内存单元当作专门存储数据的内存空间，从而定义了一个数据段。
- 例：用123B0H~123B9H的空间来存放数据
  - 段地址：123BH　起始偏移地址：0000H　长度：10字节
  - 段地址：1230H　起始偏移地址：00B0H　长度：10字节
  - ....

将哪段内存当作数据段，段地址如何定，在编程时安排。

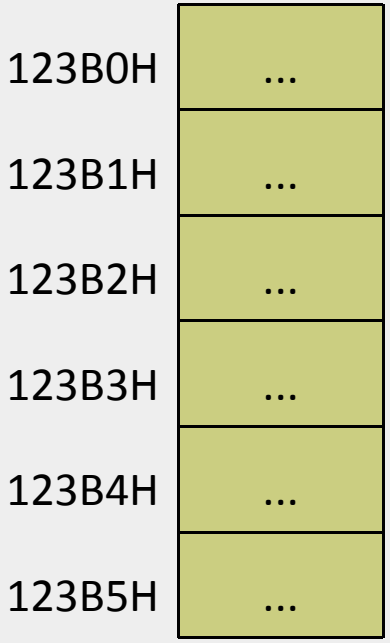
- 处理方法：(DS):([address])
  - 用DS存放数据段的段地址
  - 用相关指令访问数据段中的具体单元，单元地址由[address]指出

mov、add、sub...

将123B0H~123BAH的内存单元定义为数据段

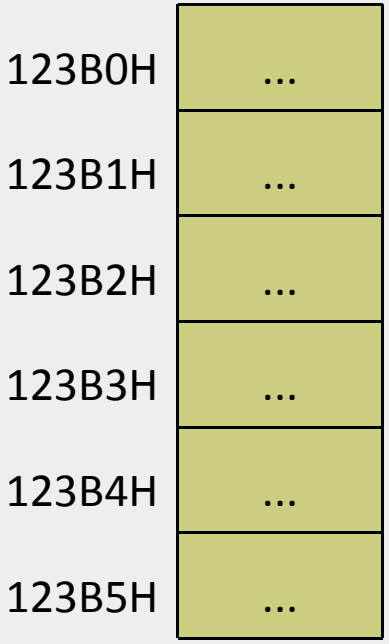
- 累加数据段中的前3个单元中的数据

```
mov ax, 123BH
mov ds, ax
mov al, 0
add al, [0]
add al, [1]
add al, [2]
```



- 累加数据段中的前3个**字型数据**

```
mov ax, 123BH
mov ds, ax
mov ax, 0
add ax, [0]
add ax, [2]
add ax, [4]
```



练习

预设数据

```
-E 0:0 70 80 F0 30 EF 60 30 E2 00 80 12 66 20 22 60
-E 0:10 62 26 E6 D6 CC 2E 3C 3B AB BA 00 00 26 06 66 68
-D 0:0 1F
0000:0000  70 80 F0 30 EF 60 30 E2-00 80 12 66 20 22 60 60   p..0.`0....f "``
0000:0010  62 26 E6 D6 CC 2E 3C 3B-AB BA 00 00 26 06 66 68   b&....<;....&.fh
```

预设代码

```
-A 073F:0100
073F:0100 mov ax, 1
073F:0103 mov ds, ax
073F:0105 mov ax, [0000]
073F:0108 mov bx, [0001]
073F:010C mov ax, bx
073F:010E mov ax, [0000]
073F:0111 mov bx, [0002]
073F:0115 add ax, bx
073F:0117 add ax, [0004]
073F:011B mov ax, 0
073F:011E mov al, [0002]
073F:0121 mov bx, 0
073F:0124 mov bl, [000C]
073F:0128 add al, bl
073F:012A
```

寄存器值

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=0000  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 B80100        MOV     AX,0001
```

提示：可以通过“R寄存器”命令修改，关键是CS和IP

执行代码

```
-t

AX=0001  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=0000  ES=073F  SS=073F  CS=073F  IP=0103   NV UP EI PL NZ NA PO NC
073F:0103 8ED8          MOV     DS,AX
-t_
```

给出00000H-0001F的数据，请写出下面代码的执行结果：

代码	AX	BX
mov ax,[0000]		
mov bx,[0001]		
mov ax,bx		
mov ax,[0000]		
mov bx,[0002]		
add ax, bx		
add ax,[0004]		
mov ax,0		
mov al,[0002]		
mov bx, 0		
mov bl, [000C]		
add al,bl		

①“人脑”计算；②“电脑”验证

用mov指令操作数据

指令形式	例示
mov 寄存器，数据	mov ax, 8
mov 寄存器，寄存器	mov ax, bx
mov 寄存器，内存单元	mov ax, [0]
mov 内存单元，寄存器	mov [0], ax
mov 段寄存器，寄存器	mov ds, ax

- 已知：mov 段寄存器，寄存器

  推测1➔ mov 寄存器，段寄存器

- 已知：mov 内存单元，寄存器

  推测2➔ mov 内存单元，段寄存器

  推测3➔ mov 段寄存器，内存单元

- 已知：mov 寄存器，数据

  推测4➔ mov 段寄存器，数据

大胆地假设，小心地求证。

验证1：

```
-a 073f:100
073F:0100 mov ax, ds
073F:0102
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 8CD8          MOV     AX,DS
-t

AX=073F  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0102   NV UP EI PL NZ NA PO NC
073F:0102 0000          ADD     [BX+SI],AL                   DS:0000=CD
-
```

验证2：

mov ax, 1000H

mov ds, ax

mov [0], ds

验证3：

mov ax, 1000H

mov ds, ax

mov ds, [0]

验证4：

mov ds, 8

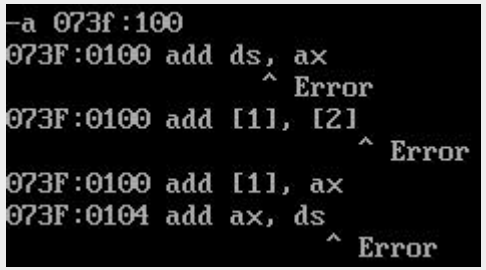
```
-a 073f:100
073F:0100 mov ds, 8
                  ^ Error
```

加法add和减法sub指令

add指令形式	例示
add 寄存器，数据	add ax, 8
add 寄存器，寄存器	add ax, bx
add 寄存器，内存单元	add ax, [0]
add 内存单元，寄存器	add [0], ax

sub指令形式	例示
sub 寄存器，数据	sub ax, 8
sub 寄存器，寄存器	sub ax, bx
sub 寄存器，内存单元	sub ax, [0]
sub 内存单元，寄存器	sub [0], ax

- 推测1➔ add 段寄存器，寄存器
- 推测2➔ add 内存单元，内存单元
- 推测...➔



用DS和[address]形式访问内存中数据段方法小结

💻指令

```
mov ax, 1000H
mov ds, ax
mov ax, 11316
mov [0], ax
mov bx, [0]
sub bx, [2]
mov [2], bx
```

💻结合体验品味

（1）字在内存中存储时 ，要用两个地址连续的内存单元来存放，字的低位字节存放在低地址单元中，高位字节存放再高地址单元中。

（2）用 mov 指令要访问内存单元，可以在mov指令中只给出单元的偏移地址，此时，段地址默认在DS寄存器中。

（3）[address]表示一个偏移地址为address的内存单元。

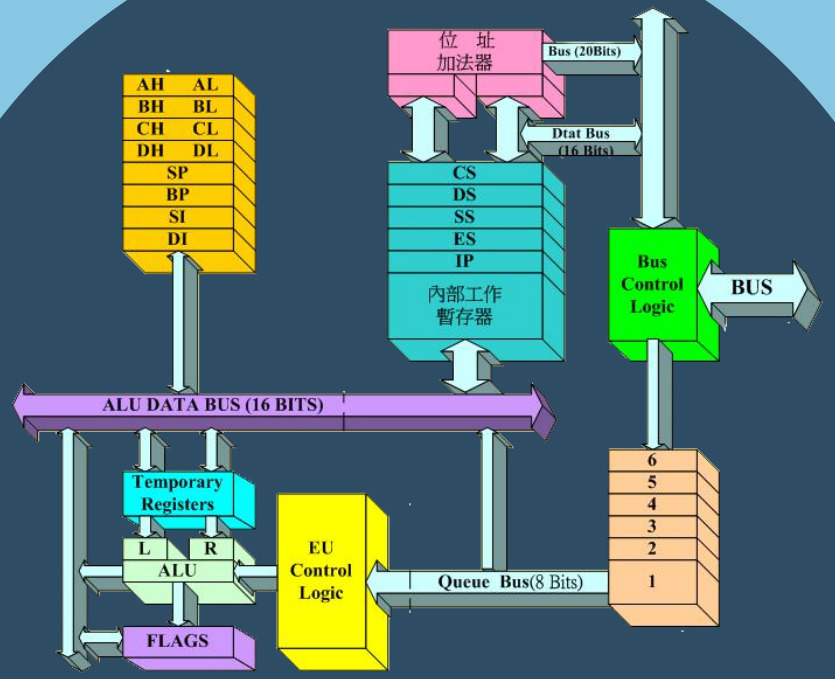
（4）在内存和寄存器之间传送字型数据时，高地址单元和高8位寄存器、低地址单元和低8位寄存器相对应。

（5）mov、add、sub是具有两个操作对象的指令，访问内存中的数据段（对照：jmp是具有一个操作对象的指令，对应内存中的代码段）。

（6）可以根据自己的推测，在Debug中实验指令的新格式。

栈及栈操作的实现

贺利坚　主讲

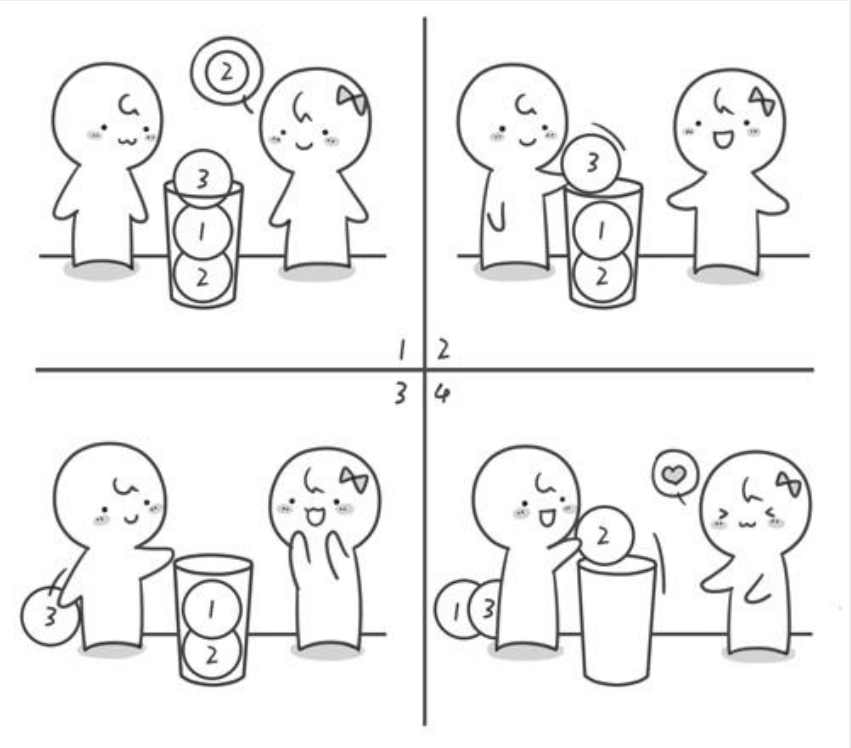


汇编语言程序设计

Assembly Language

栈结构

- 栈是一种只能在一端进行插入或删除操作的数据结构。
- 栈有两个基本的操作：入栈和出栈。
  - 入栈：将一个新的元素放到栈顶；
  - 出栈：从栈顶取出一个元素。
- 栈顶的元素总是最后入栈，需要出栈时，又最先被从栈中取出。
- 栈的操作规则：LIFO（Last In First Out，后进先出）
- CPU提供的栈机制
  - 现今的CPU中都有栈的设计。
  - 8086CPU提供相关的指令，支持用栈的方式访问内存空间。
  - 基于8086CPU的编程，可以将一段内存当作栈来使用。

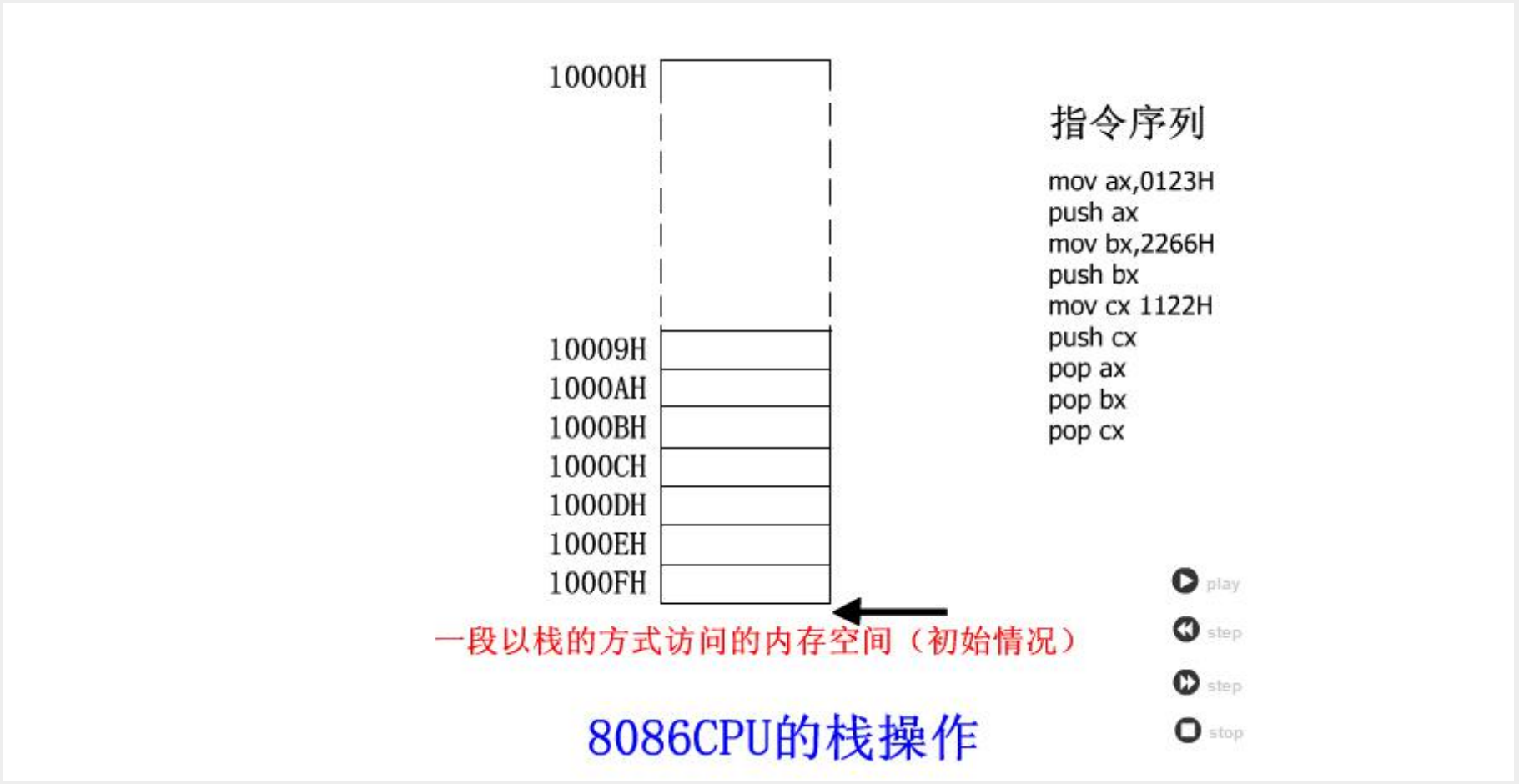


- PUSH(入栈)和 POP(出栈)指令
  - push ax：将ax中的数据送入栈中
  - pop ax：从栈顶取出数据送入ax

（以字为单位对栈进行操作）

例：设将10000H~1000FH内存当作栈来使用……

```
mov ax,0123H
push ax
mov bx,2266H
push bx
mov cx,1122H
push cx
pop ax
pop bx
pop cx
```



问题：

1、CPU如何知道一段内存空间被当作栈使用？

2、执行push和pop的时候，如何知道哪个单元是栈顶单元？

回答：8086CPU中，有两个与栈相关的寄存器：

栈段寄存器SS　　- 存放栈顶的段地址

栈顶指针寄存器SP - 存放栈顶的偏移地址

——任意时刻，SS:SP指向栈顶元素。

栈的操作

mov ax, 1000H

mov ss, ax

mov sp, 0010H

寄存器

AX	BX	SS	SP
1000H		1000H	0010

mov ax, 001AH

mov bx, 001BH

寄存器

AX	BX	SS	SP
001AH	001BH	1000H	0010

push ax

push bx

寄存器

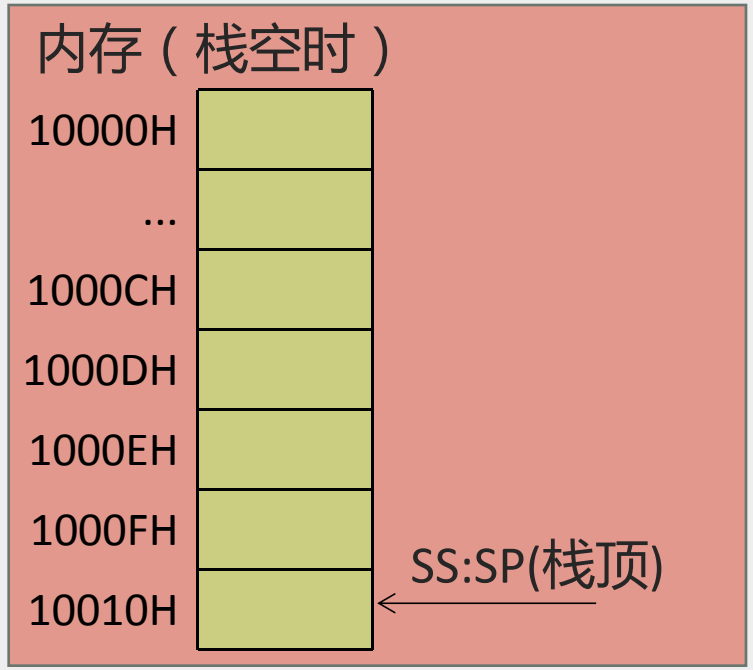
AX	BX	SS	SP
001AH	001BH	1000H	000C

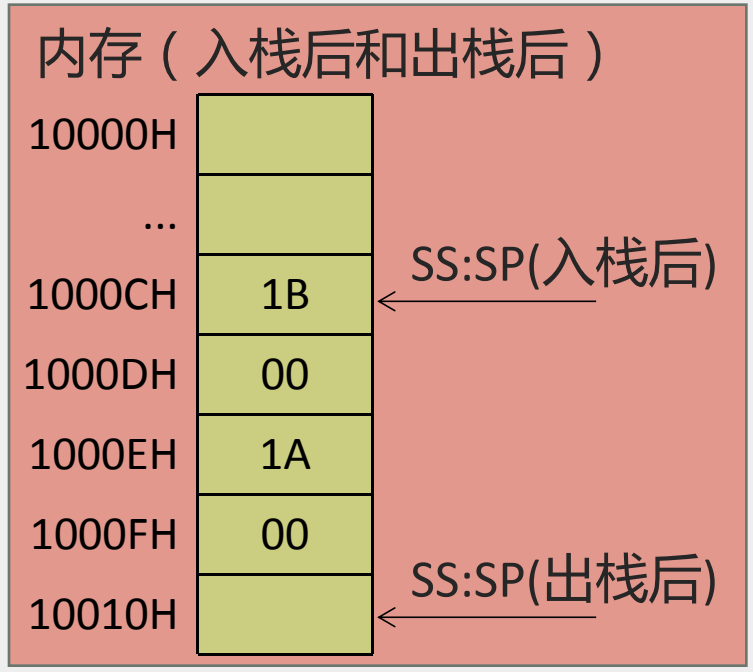
pop ax

pop bx

寄存器

AX	BX	SS	SP
001BH	001AH	1000H	0010

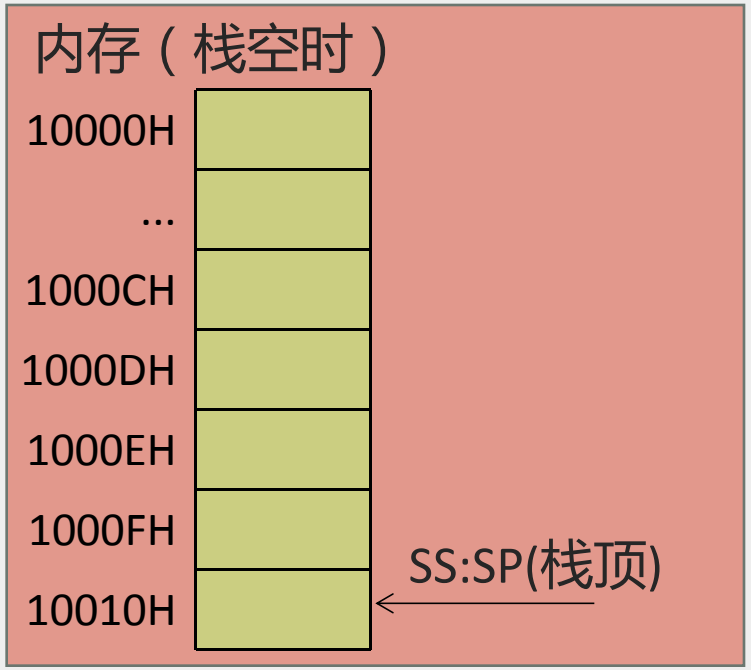


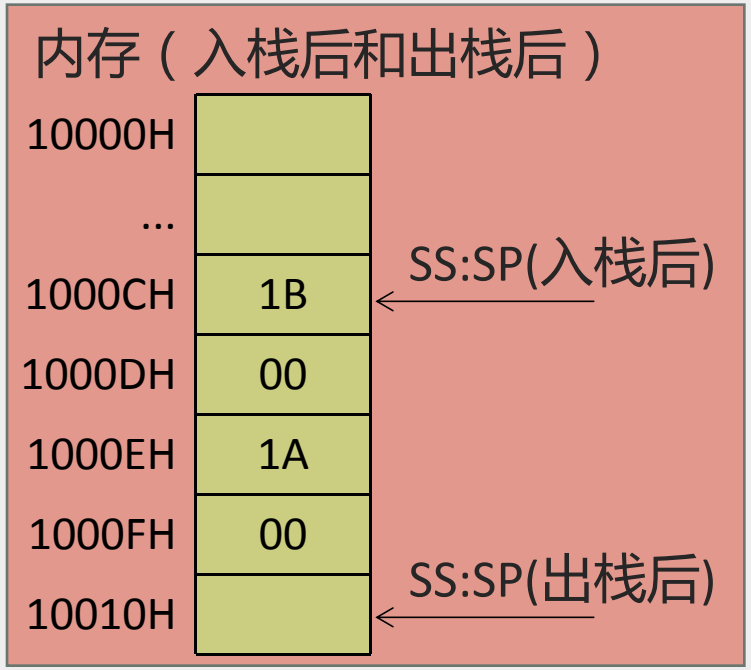


push 指令和pop指令的执行过程

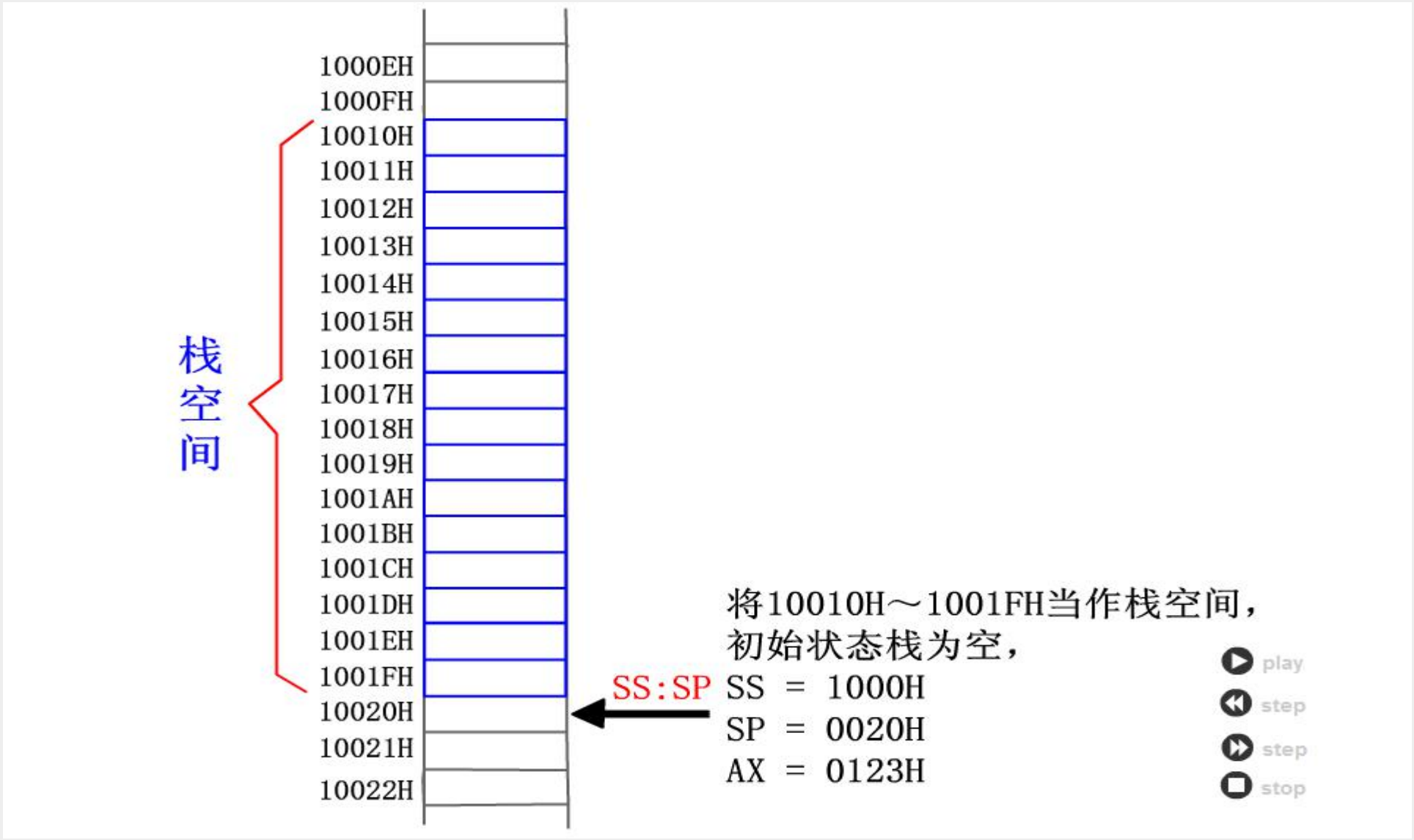
```
mov ax, 1000H
mov ss, ax
mov sp, 0010H

mov ax, 001AH
mov bx, 001BH

push ax
push bx

pop ax
pop bx
```

- push ax
  - （1）SP=SP–2；
  - （2）将ax中的内容送入SS:SP指向的内存单元处，SS:SP此时指向新栈顶。
- pop ax
  - （1）将SS:SP指向的内存单元处的数据送入ax中；
  - （2）SP = SP+2，SS:SP指向当前栈顶下面的单元，以当前栈顶下面的单元为新的栈顶。
- 栈顶超界问题
  - 如何能够保证在入栈、出栈时，栈顶不会超出栈空间？





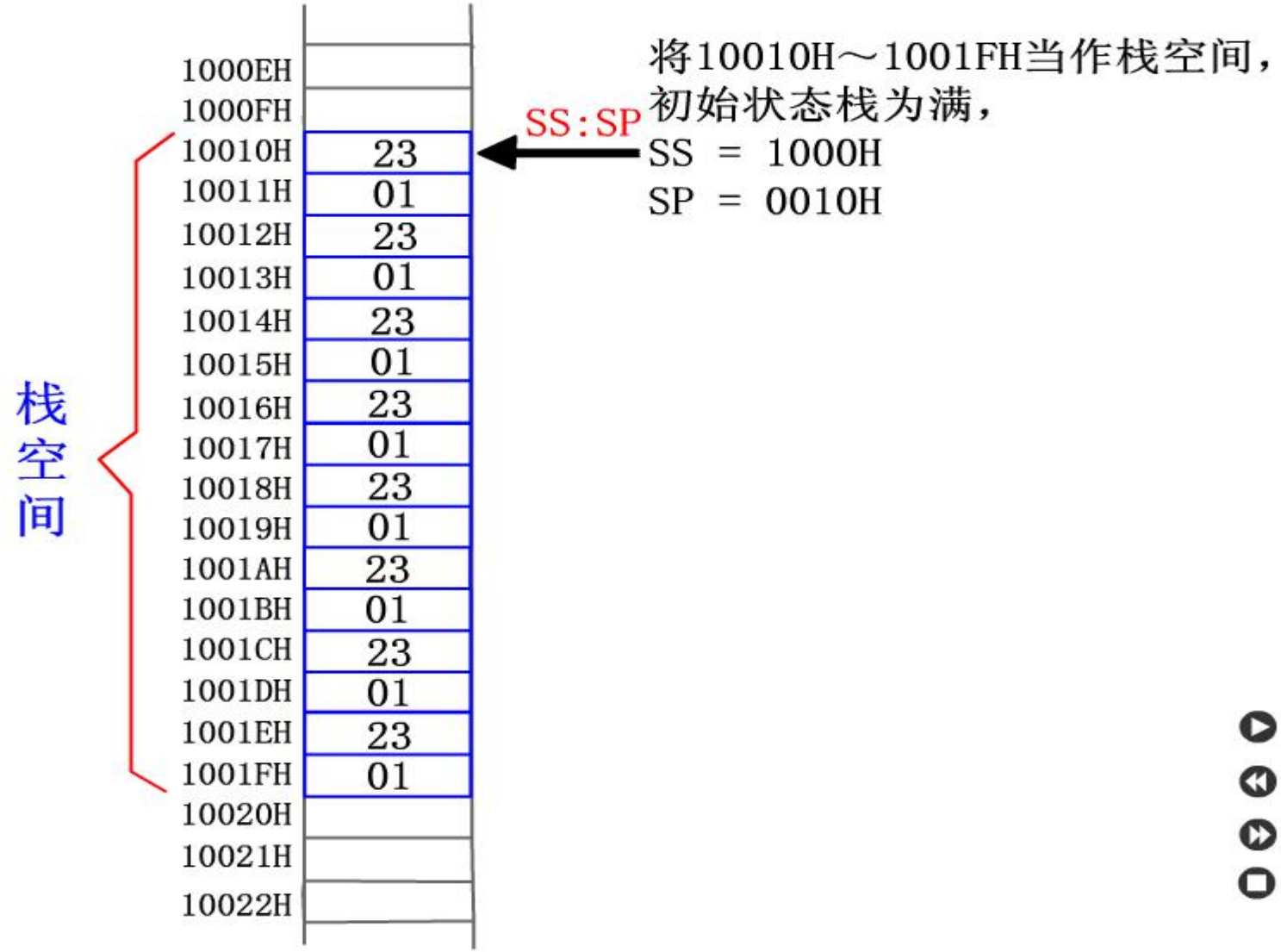
执行入栈(push)时，栈顶超出栈空间



当栈满的时候再使用push指令入栈，将发生栈顶超界问题。

栈顶超界是危险的。

执行出栈(pop)时，栈顶超出栈空间



当栈空的时候再使用pop指令出栈，将发生栈顶超界问题。

栈顶超界是危险的。

栈顶超界问题的解决

- 8086CPU不保证对栈的操作不会超界。

  8086CPU 只知道栈顶在何处（由SS:SP指示），不知道程序安排的栈空间有多大。

- 我们在编程的时候要自己操心栈顶超界的问题 ，要根据可能用到的最大栈空间，来安排栈的大小，防止入栈的数据太多而导致的超界；防止出栈时栈空了仍然继续出栈而导致的超界。

栈的小结

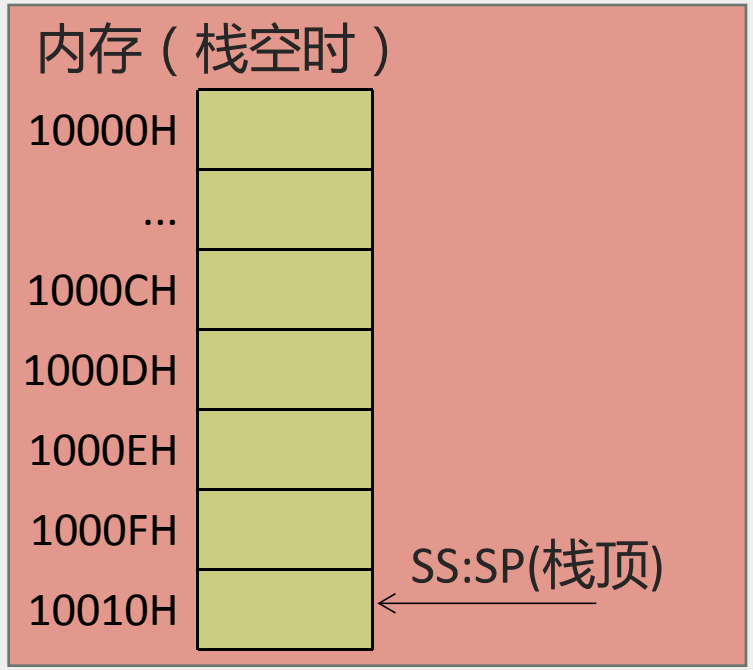
- push、pop 实质上就是一种内存传送指令，可以在寄存器和内存之间传送数据，与mov指令不同的是，push和pop指令访问的内存单元的地址不是在指令中给出的，而是由SS:SP指出的。
- 执行push和pop指令时，SP 中的内容自动改变。
- 8086CPU提供的栈操作机制：
  - 在SS，SP中存放栈顶的段地址和偏移地址，入栈和出栈指令根据SS:SP指示的地址，按照栈的方式访问内存单元。
  - push指令的执行步骤：

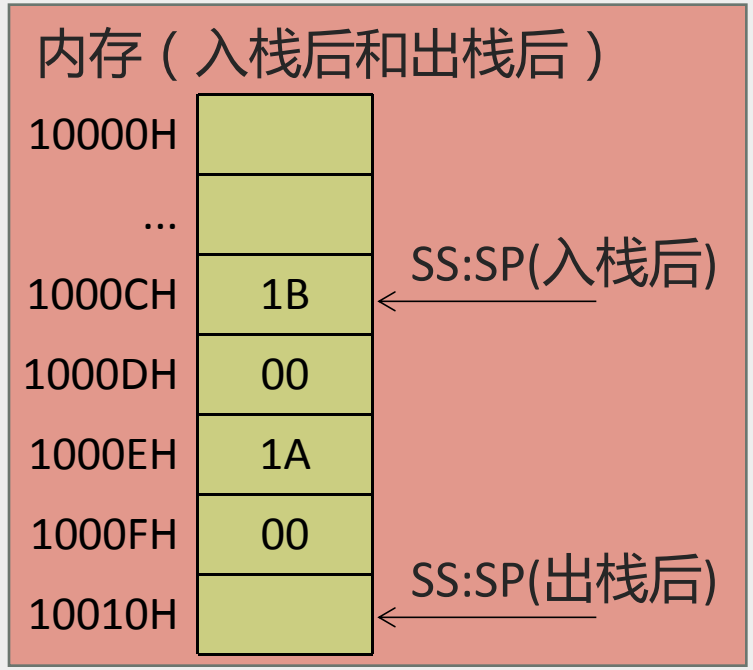
    1）SP=SP-2；

    2）向SS:SP指向的字单元中送入数据。
  - pop指令的执行步骤：

    1）从SS:SP指向的字单元中读取数据；

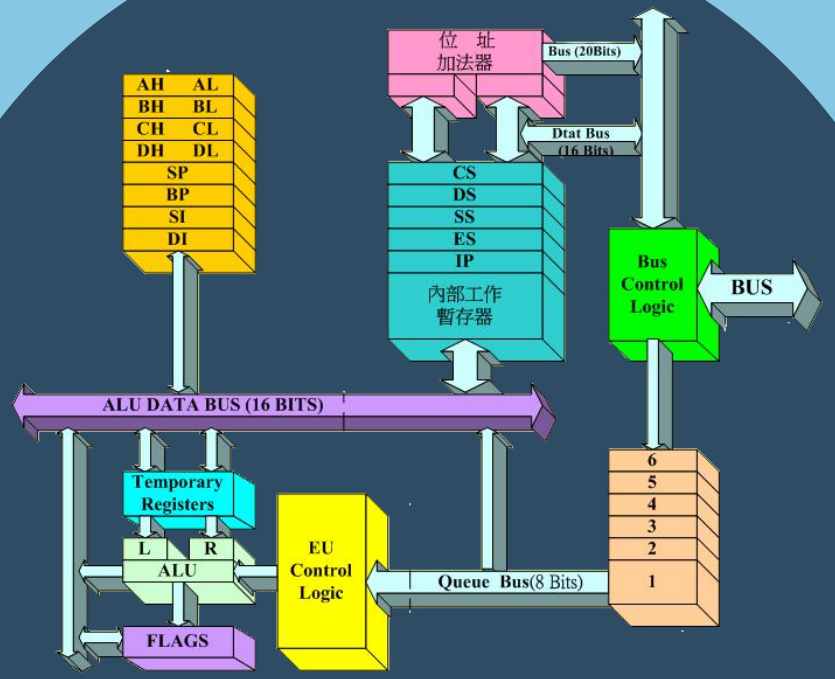
    2）SP=SP-2。





关于“段”的总结

贺利坚　主讲



汇编语言程序设计

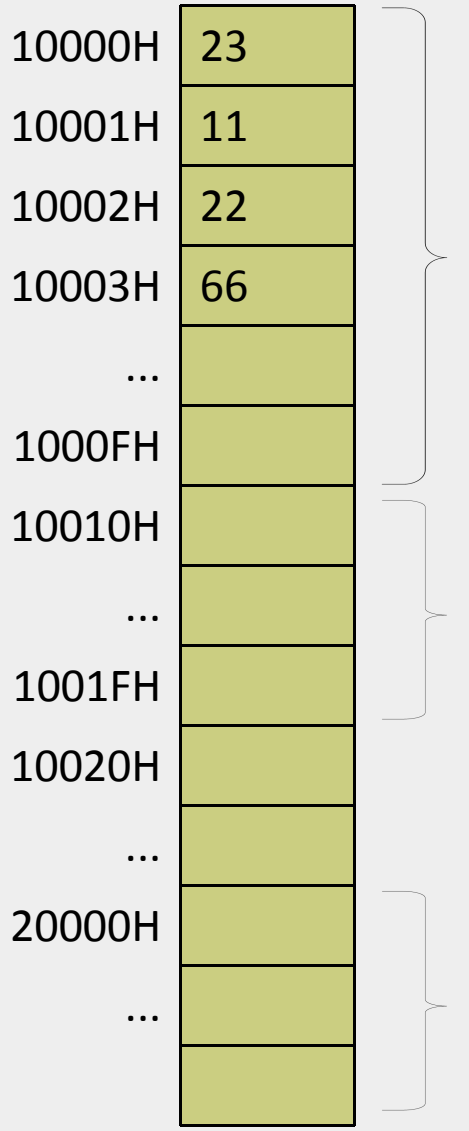
Assembly Language

各种段——

10000H	23
10001H	11
10002H	22
10003H	66

- 基础
  - 物理地址=段地址×16+偏移地址
- 做法
  - 编程时，可以根据需要将一组内存单元定义为一个段。
  - 可以将起始地址为16的倍数，长度为N（N≤64K）的一组地址连续的内存单元，定义为一个段。
  - 将一段内存定义为一个段，用一个段地址指示段，用偏移地址访问段内的单元——在程序中可以完全由程序员安排。
- 三种段
  - 数据段
    - 将段地址放在 DS中
    - 用mov、add、sub等访问内存单元的指令时，CPU将我们定义的数据段中的内容当作数据段来访问；
  - 代码段
    - 将段地址放在 CS中，将段中第一条指令的偏移地址放在IP中
    - CPU将执行我们定义的代码段中的指令；
  - 栈段
    - 将段地址放在SS中，将栈顶单元的偏移地置放在 SP 中
    - CPU在需要进行栈操作(push、pop)时，就将我们定义的栈段当作栈空间来用。

综合示例：按要求设置段并执行代码




```
mov bx, 1000H
mov ds, bx
mov bx, 1001H
mov ss, bx
mov sp, 10H
mov ax, [0]
mov bx, [2]
push ax
push bx
pop ax
pop bx
mov [0], ax
mov [2], bx
```

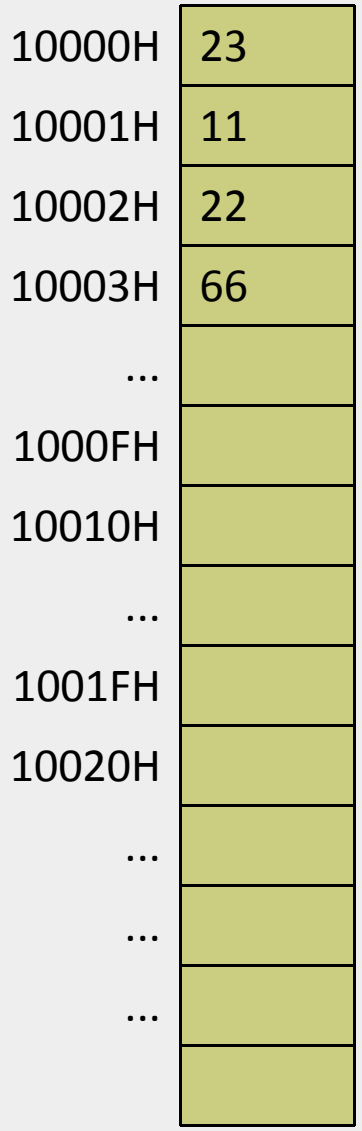
```
-rds
DS 1000
:1000
-rss
SS 1001
:1001
-rsp
SP 0010
:0010
-rcs
CS 2000
:2000
-rip
IP 0000
:0000
-e ds:0 23 11 22 66
-r
AX=0001  BX=0000  CX=0000  DX=0000  SP=0010  BP=0000  SI=0000  DI=0000
DS=1000  ES=073F  SS=1001  CS=2000  IP=0000   NV UP EI PL NZ NA PO NC
2000:0000 0000          ADD     [BX+SI],AL                      DS:0000=23
-d ds:0 f
1000:0000  23 11 22 66 00 00 00 00-00 00 00 00 00 00 00 00   #."f............
```

```
-a CS:0000
2000:0000 mov bx, 1000
2000:0003 mov ds, bx
2000:0005 mov bx, 1001
2000:0008 mov ss, bx
2000:000A mov sp, 10
2000:000D mov ax, [0]
```

```
-t

AX=0001  BX=1000  CX=0000  DX=0000  SP=0010  BP=00
DS=1000  ES=073F  SS=1001  CS=2000  IP=0003   NV U
2000:0003 8EDB          MOV     DS,BX
-t
```

```
-d ds:0 f
1000:0000  22 66 23 11 00 00 00 00-00 00 00 00 00 00 00 00   "f#.............
-
```

综合示例：三个段地址可以一样滴！



数据段

栈段

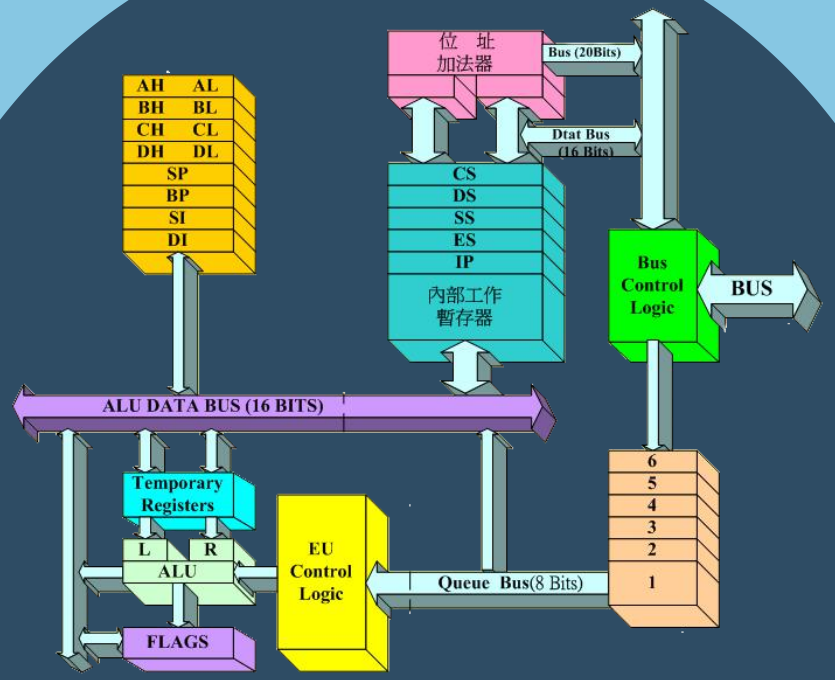
代码段

```
mov bx, 1000H
mov ds, bx
mov ss, bx
mov sp, 20H
mov ax, [0]
mov bx, [2]
push ax
push bx
pop ax
pop bx
mov [0], ax
mov [2], bx
```

```
-rds
DS 1000
:1000
-rss
SS 1000
:1000
-rcs
CS 1000
:1000
-rsp
SP 0020
:0020
-rip
IP 001F
:0020
-e ds:0 23 11 22 66
-r
AX=6622  BX=1123  CX=0000  DX=0000  SP=0020  BP=0000  SI=0000  DI=0000
DS=1000  ES=073F  SS=1000  CS=1000  IP=0020   NV UP EI PL NZ NA PO NC
1000:0020 0000          ADD     [BX+SI],AL                      DS:1123=00
```

导学-汇编语言程序

贺利坚　主讲



汇编语言程序设计

Assembly Language

汇编语言程序设计课程内容

1. 绪论
2. 访问寄存器和内存
3. 汇编语言程序
4. 内存寻址方式
5. 流程转移与子程序
6. 中断及其应用
7. 高级汇编语言技术

0401 用汇编语言写的源程序
0402 由源程序到程序运行
0403 用Debug跟踪程序的执行

0501 [...]和(...)
0502 Loop指令
0503 Loop指令使用再例
0504 段前缀的使用

0601 在代码段中使用数据
0602 在代码段中使用栈
0603 将数据、代码、栈放入不同段

各节与教材章节的对应关系

视频（共9个）	教材对应章节
0401 用汇编语言写的源程序	4.1-4.2节
0402 由源程序到程序运行	4.3-4.8节
0403 用Debug跟踪程序的执行	4.9节
0501 [...]和(...)	第5章序言部分+5.1节
0502 Loop指令	5.2节
0503 Loop指令使用再例	5.3节
0504 段前缀的使用	5.4-5.8节
0601 在代码段中使用数据	6.1节
0602 在代码段中使用栈	6.2节
0603 将数据、代码、栈放入不同段	6.3节

视频

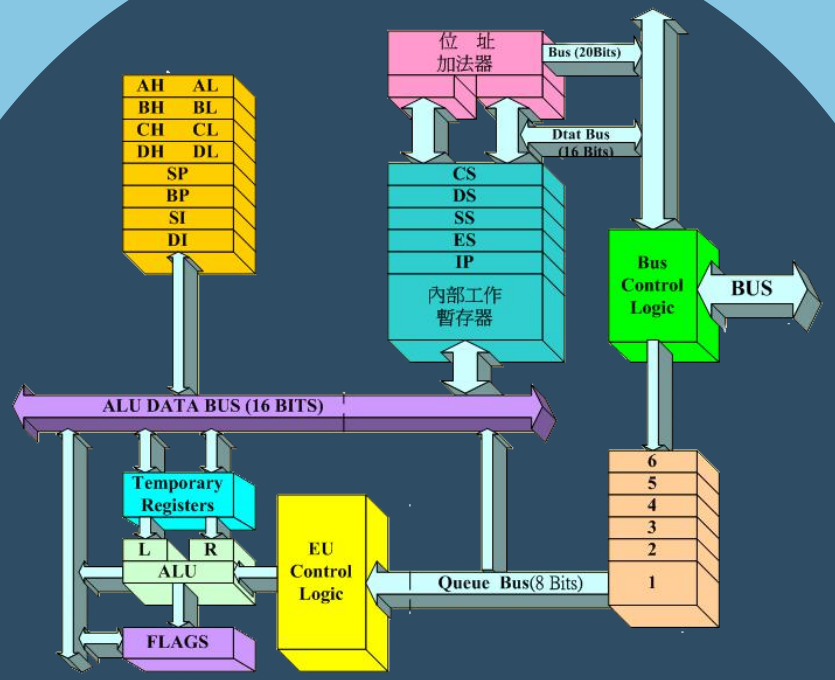
教材

检测

实验

用汇编语言写的源程序

贺利坚　主讲



汇编语言程序设计

Assembly Language

用汇编语言编写程序的工作过程

汇编程序：包含汇编指令和伪指令的文本


```
assume cs:codesg
codesg segment
    mov ax,0123H
    mov bx,0456H
    add ax,bx
    add ax,ax

    mov ax,4c00h
    int 21h
codesg ends
end
```

汇编指令，对应有机器码的指令，可以被编译为机器指令，最终被CPU执行。

伪指令

伪指令

没有对应的机器码的指令，最终不被CPU所执行。

谁来执行伪指令呢？

伪指令是由编译器来执行的指令，编译器根据伪指令来进行相关的编译工作。

程序返回（套路！）：程序结束运行后，将CPU的控制权交还给使它得以运行的程序（常为DOS系统）。

程序中的三种伪指令

```
assume cs:codesg

codesg segment

        mov ax,0123H

        mov bx,0456H

        add ax,bx

        add ax,ax



        mov ax,4c00h

        int 21h

codesg ends

end
```

段定义

- 一个汇编程序是由多个段组成的，这些段被用来存放代码、数据或当作栈空间来使用。
- 一个有意义的汇编程序中至少要有一个段，这个段用来存放代码。
- 定义程序中的段：每个段都需要有段名

  段名 **segment**　——段的开始

  ....

  段名 **ends**　——段的结束

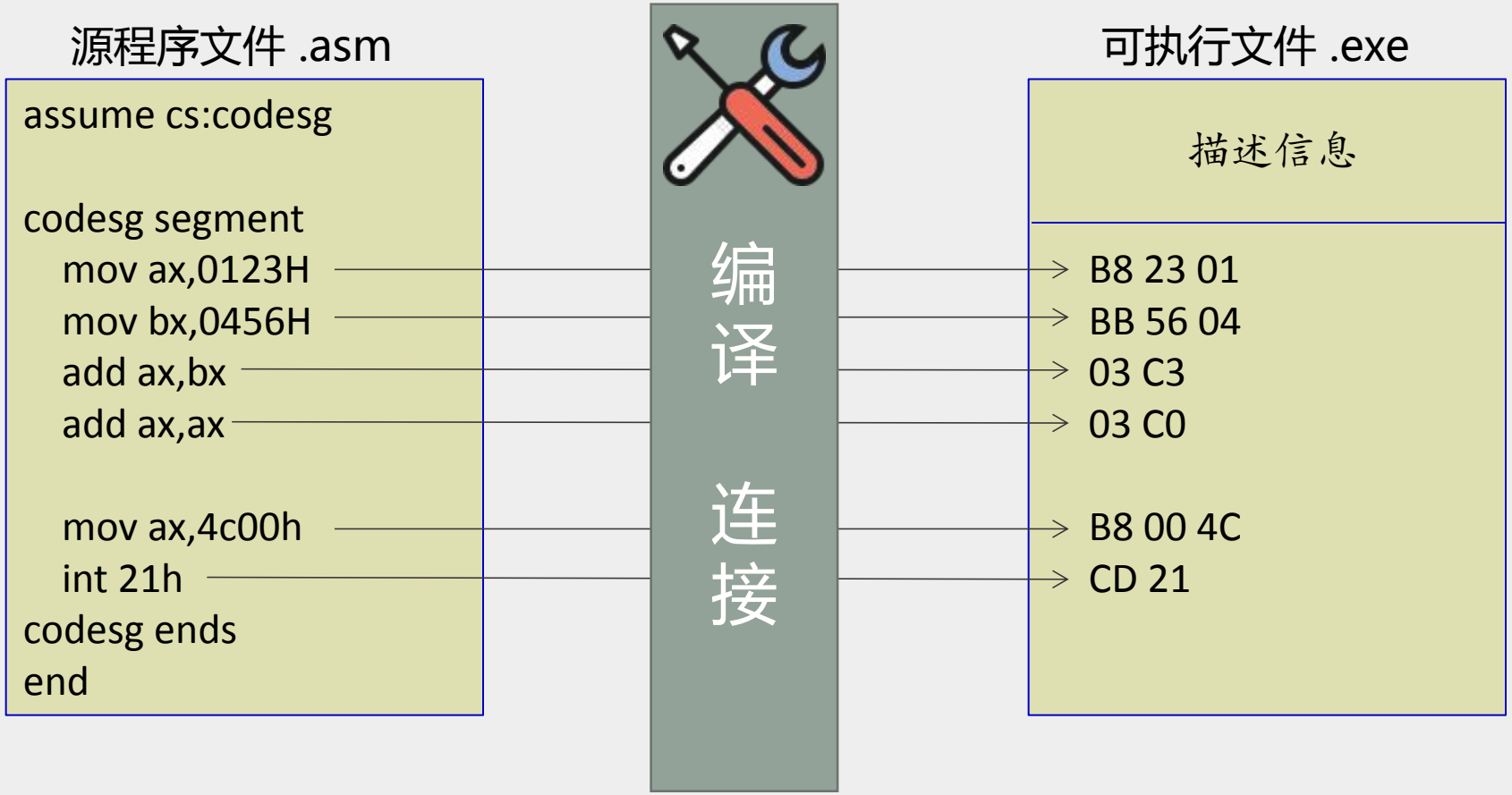
end (不是ends)

- 汇编程序的结束标记。若程序结尾处不加end，编译器在编译程序时，无法知道程序在何处结束。

assume(假设)

- 含义是假设某一段寄存器和程序中的某一个用 segment ... ends 定义的段相关联——assume cs:codesg指CS寄存器与codesg关联，将定义的codesg当作程序的代码段使用。

源程序经编译连接后变为机器码



汇编程序的结构

- 在Debug中直接写入指令编写的汇编程序

```
C:\>debug
-a
073F:0100 mov ax, 27A1
073F:0103 add ax, 892C
073F:0106
-
```

  - 适用于功能简单、短小精悍的程序
  - 只需要包含汇编指令即可
- 单独编写成源文件后再编译为可执行文件的程序
  - 适用于编写大程序
  - 需要包括汇编指令，还要有指导编译器工作的伪指令
  - 源程序由一些段构成，这些段存放代码、数据，或将某个段当作栈空间。
- ；---注释

```
assume cs:code,ds:data,ss:stack
data segment
  dw 0123H,0456H,0789H,0abcH,0defH
data ends
stack segment
  dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
stack ends
code segment
        mov ax,stack
        mov ss,ax
        mov sp,20h     ;设置栈段
        mov ax,data
        mov ds,ax      ;设置数据段
        mov bx,0
        mov cx,8
    s:  push [bx]
        add bx,2
        loop s
        ....
code ends
end
```

如何写出一个程序来?

例：编程求 2^3。

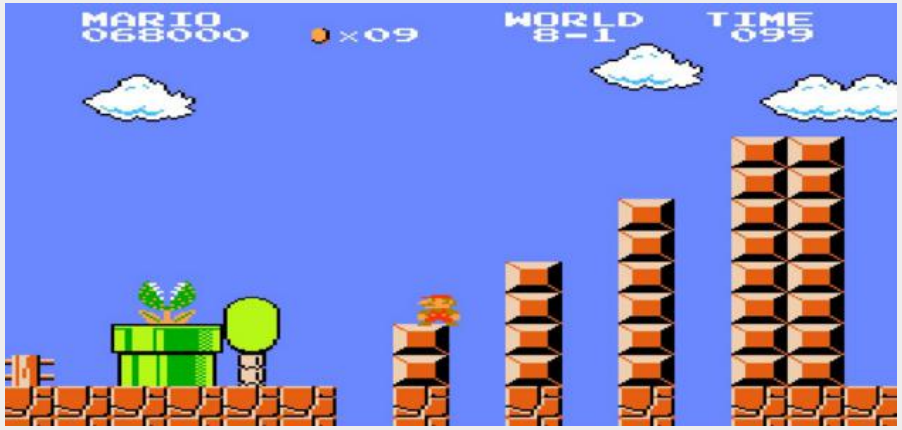
①定义一个段

②实现处理任务

③指出程序在何结束

④段与段寄存器关联

⑤加上程序返回的代码




```
abc segment

abc ends
```
①

```
abc segment
        mov ax,2
        add ax,ax
        add ax,ax
abc ends
```
②

```
abc segment
        mov ax,2
        add ax,ax
        add ax,ax
abc ends
end
```
③

```
assume cs:abc
abc segment
        mov ax,2
        add ax,ax
        add ax,ax
abc ends
end
```
④

```
assume cs:abc
abc segment
        mov ax,2
        add ax,ax
        add ax,ax

        mov ax,4c00h
        int 21h
abc ends
end
```
⑤

程序中可能的错误

- 语法错误
  - 程序在编译时被编译器发现的错误；
  - 容易发现下面程序中错误

```
aume cs:abc
abc segment
   mov ax,2
   add ax,ax
   add ax,sx
end
```

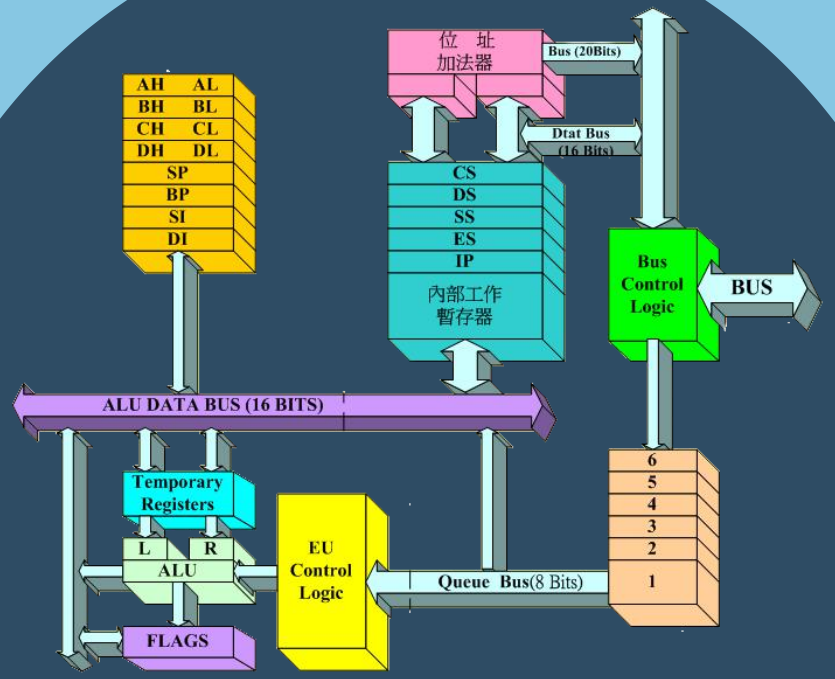
- 逻辑错误
  - 程序在编译时不能表现出来的、在运行时发生的错误；
  - 不容易发现下面程序中的错误

```
assume cs:abc
 abc segment
      mov ax,2
      add ax,ax
      add ax,bx
      mov ax,4c10H
      int 21H
   abc ends
   end
```

求 2^3

由源程序到程序运行

贺利坚　主讲

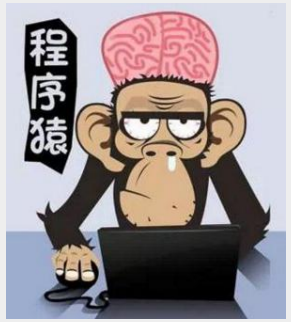


汇编语言程序设计

Assembly Language

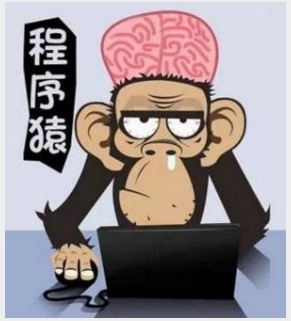
由写出源程序到执行可执行文件的过程

文本编辑



源程序文件 .asm

```
...
mov ax, 0123H
mov bx, 0456H
add ax, bx
...
```



编译

目标文件 .obj

```
...
B82301 BB5604
01DB
...
```

连接

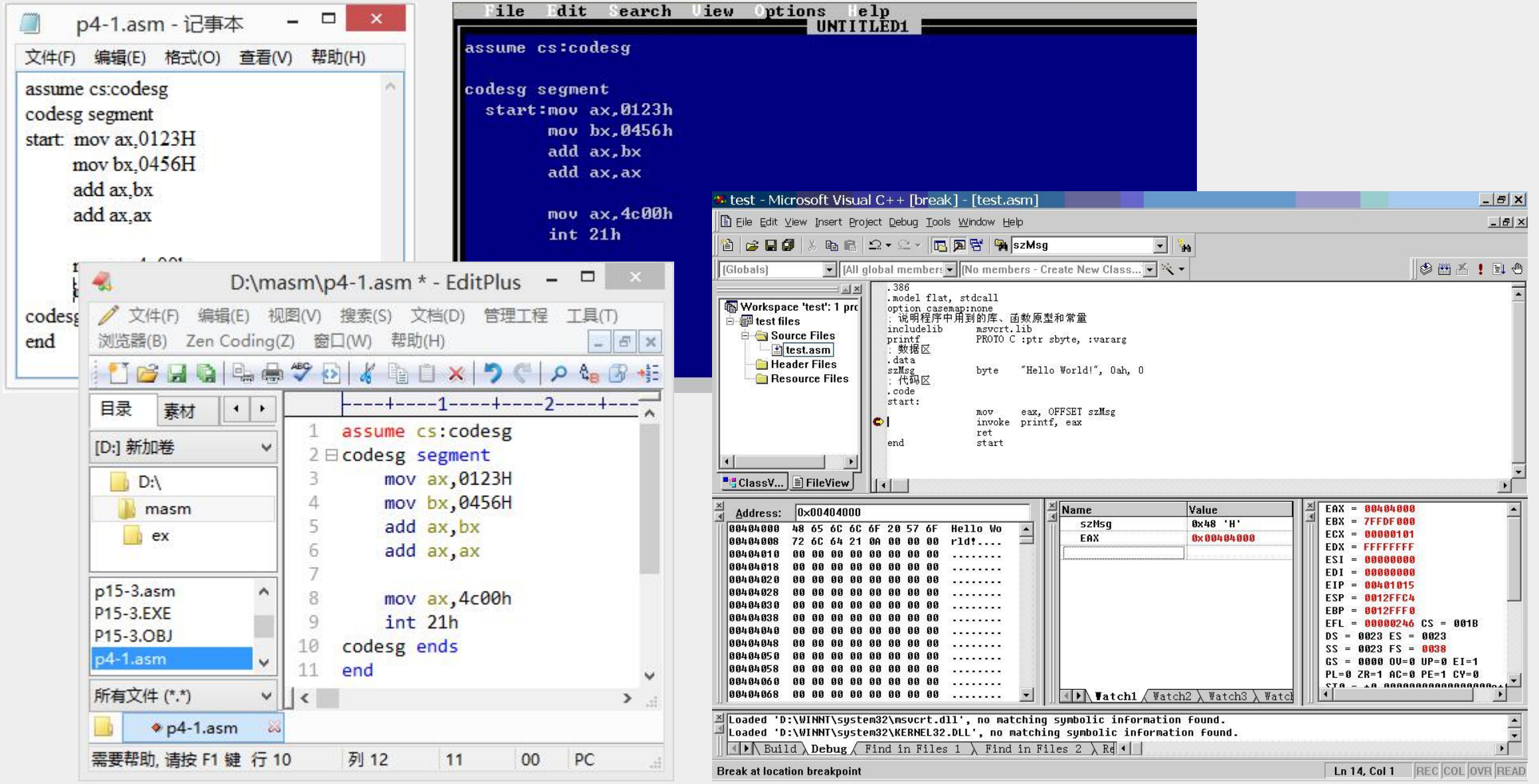
可执行文件 .exe

```
相关的描述信息
...
B82301 BB5604
01DB
...
```

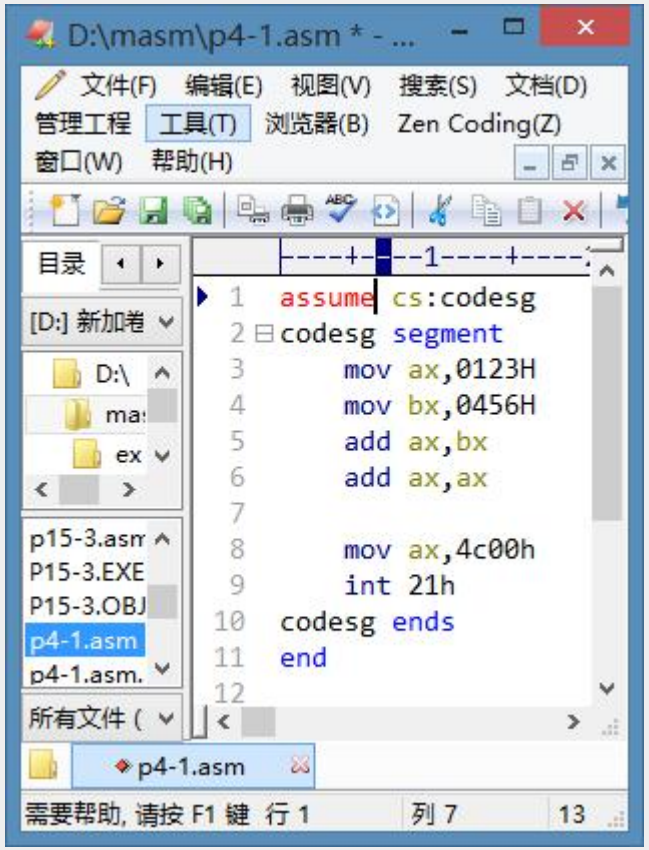
运行程序

计算机

编辑源程序


p4-1.asm - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
assume cs:codesg
codesg segment
start: mov ax,0123H
mov bx,0456H
add ax,bx
add ax,ax
end
File Edit Search View Options Help
UNTITLED1
assume cs:codesg
codesg segment
start:mov ax,0123h
mov bx,0456h
add ax,bx
add ax,ax
mov ax,4c00h
int 21h
D:\masm\p4-1.asm * - EditPlus
文件(F) 编辑(E) 视图(V) 搜索(S) 文档(D) 管理工程 工具(T)
浏览器(B) Zen Coding(Z) 窗口(W) 帮助(H)
目录 素材
[D:] 新加卷
D:\
masm
ex
p15-3.asm
P15-3.EXE
P15-3.OBJ
p4-1.asm
所有文件 (*.*)
p4-1.asm
1 assume cs:codesg
2 codesg segment
3 mov ax,0123H
4 mov bx,0456H
5 add ax,bx
6 add ax,ax
7
8 mov ax,4c00h
9 int 21h
10 codesg ends
11 end
需要帮助, 请按 F1 键 行 10 列 12 11 00 PC
test - Microsoft Visual C++ [break] - [test.asm]
File Edit View Insert Project Debug Tools Window Help
szMsg
(Globals)
(All global members
(No members - Create New Class...
Workspace 'test': 1 prc
test files
Source Files
test.asm
Header Files
Resource Files
ClassV...
FileView
.386
.model flat, stdcall
option casemap:none
; 说明程序中用到的库、函数原型和常量
includelib msvcrt.lib
printf PROTO C :ptr sbyte, :vararg
; 数据区
.data
szMsg byte "Hello World!", 0ah, 0
; 代码区
.code
start:
mov eax, OFFSET szMsg
invoke printf, eax
ret
end start
Address: 0x00404000
00404000 48 65 6C 6C 6F 20 57 6F Hello Wo
00404008 72 6C 64 21 0A 00 00 00 rld!....
00404010 00 00 00 00 00 00 00 00 ....
00404018 00 00 00 00 00 00 00 00 ....
00404020 00 00 00 00 00 00 00 00 ....
00404028 00 00 00 00 00 00 00 00 ....
00404030 00 00 00 00 00 00 00 00 ....
00404038 00 00 00 00 00 00 00 00 ....
00404040 00 00 00 00 00 00 00 00 ....
00404048 00 00 00 00 00 00 00 00 ....
00404050 00 00 00 00 00 00 00 00 ....
00404058 00 00 00 00 00 00 00 00 ....
00404060 00 00 00 00 00 00 00 00 ....
00404068 00 00 00 00 00 00 00 00 ....
Name Value
szMsg 0x48 'H'
EAX 0x00404000
Watch1 Watch2 Watch3 Watch
EAX = 00404000
EBX = 7FFDF000
ECX = 00000101
EDX = FFFFFFFF
ESI = 00000000
EDI = 00000000
EIP = 00401015
ESP = 0012FFC4
EBP = 0012FFF0
EFL = 00000246 CS = 001B
DS = 0023 ES = 0023
SS = 0023 FS = 0038
GS = 0000 OV=0 UP=0 EI=1
PL=0 ZR=1 AC=0 PE=1 CY=0
Loaded 'D:\WINNT\system32\msvcrt.dll', no matching symbolic information found.
Loaded 'D:\WINNT\system32\KERNEL32.DLL', no matching symbolic information found.
Build Debug Find in Files 1 Find in Files 2
Break at location breakpoint
Ln 14, Col 1
REC COL OVR READ

编译



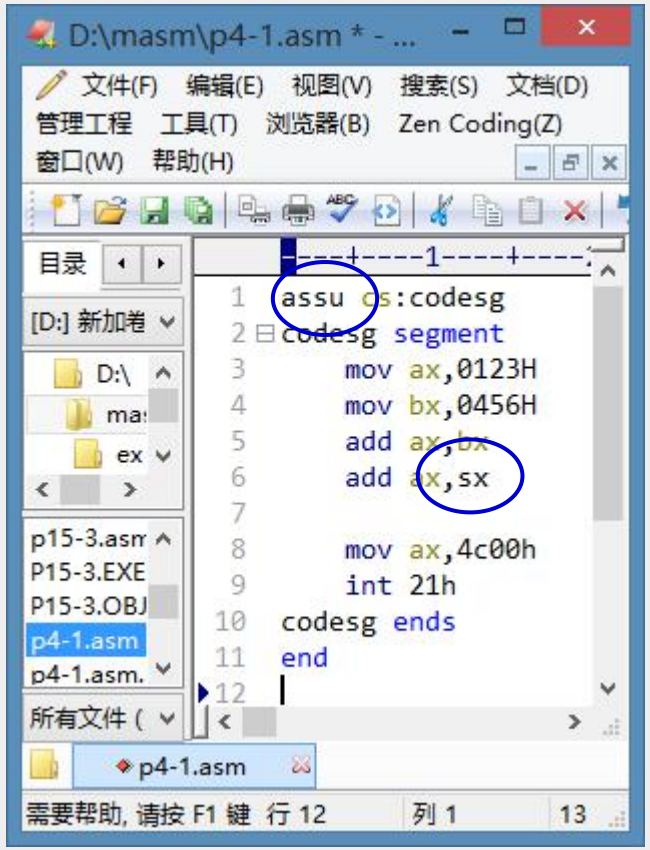

```
C:\>masm p4-1.asm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Object filename [p4-1.OBJ]:
Source listing  [NUL.LST]:
Cross-reference [NUL.CRF]:

  51798 + 464746 Bytes symbol space free

       0 Warning Errors
       0 Severe  Errors
```

```
C:\>masm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Source filename [.ASM]: p4-1.asm
Object filename [p4-1.OBJ]:
Source listing  [NUL.LST]:
Cross-reference [NUL.CRF]:

  51798 + 464746 Bytes symbol space free

       0 Warning Errors
       0 Severe  Errors
```

p4-1.asm	2017-2-5 22:20	ASM 文件	1 KB
P4-1.OBJ	2017-2-5 22:20	OBJ 文件	1 KB

- 目标文件（*.OBJ）是我们对一个源程序进行编译要得到的最终结果。
- 列表文件（*.LST）是编译器将源程序编译为目标文件的过程中产生的中间结果。
- 交叉引用文件（*.CRF）同列表文件一样，是编译器将源程序编译为目标文件过程中产生的中间结果。
- 对源程序的编译结束，编译器输出的最后两行告诉我们这个源程序没有警告错误和必须要改正的错误。

提示语法错误




```
C:\>masm p4-1.asm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Object filename [p4-1.OBJ]:
Source listing  [NUL.LST]:
Cross-reference [NUL.CRF]:
p4-1.asm(1): error A2105: Expected: instruction or directive
p4-1.asm(3): error A2062: Missing or unreachable CS
p4-1.asm(6): error A2009: Symbol not defined: SX

  51798 + 464746 Bytes symbol space free

      0 Warning Errors
      3 Severe  Errors
```

```
C:\>masm p4-1.asm;
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.


  51798 + 464746 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors

C:\>
```

两类错误

- Severe Errors
- 找不到所给出的源程序文件。

命令后加；以简化过程

连接

```
C:\>link p4-1

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Run File [P4-1.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
LINK : warning L4021: no stack segment
```

```
C:\>link

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Object Modules [.OBJ]: p4-1
Run File [P4-1.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
LINK : warning L4021: no stack segment
```

p4-1.asm	2017-2-5 22:20	ASM 文件	1 KB
P4-1.EXE	2017-2-6 15:54	应用程序	1 KB
P4-1.OBJ	2017-2-5 22:20	OBJ 文件	1 KB

```
C:\>link p4-11

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Run File [P4-11.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
LINK : fatal error L1093: P4-11.OBJ : object not found
```

- 可执行文件(.EXE)是我们对一个程序进行连接要得到的最终结果。
- 映像文件(.MAP)是连接程序将目标文件连接为可执行文件过程中产生的中间结果。
- 库文件(.LIB)里包含了一些可以调用的子程序，如果我们的程序中调用了某一个库文件中的子程序，就需要在连接的时候，将这个库文件和我们的目标文件连接到一起，生成可执行文件。
- no stack segment，一个“没有栈段”的警告错误 ，可以不理会这个错误。

- 连接中可能会遭遇错误
- 例：object nor found —— 找不到对象

```
C:\>link p4-1;

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

LINK : warning L4021: no stack segment
```

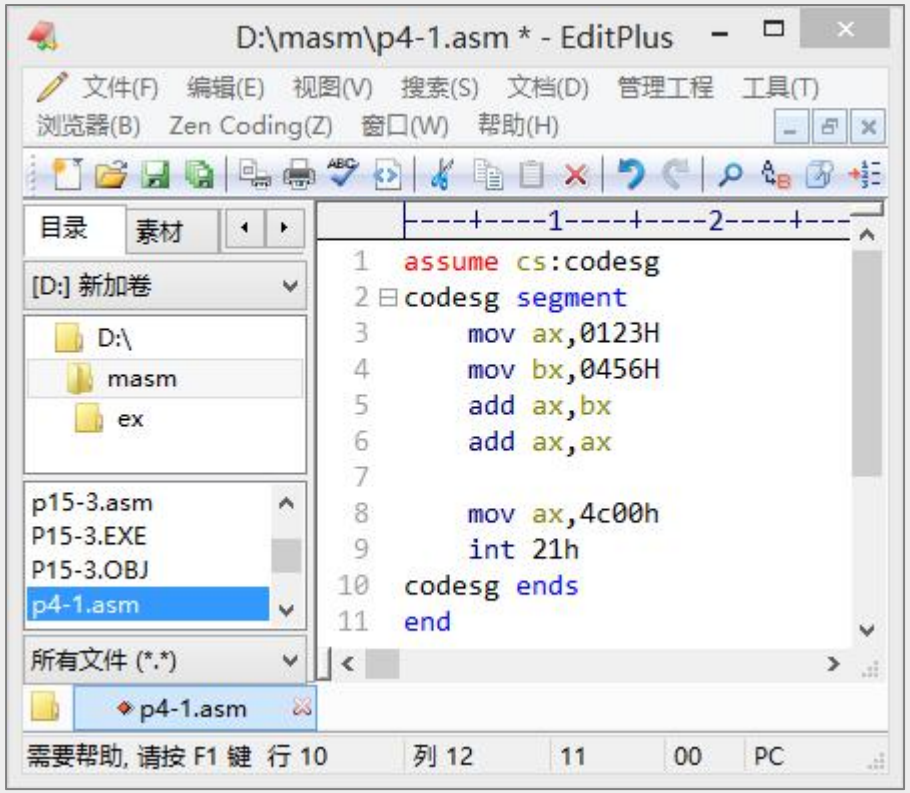
命令后加；以简化过程

执行可执行程序

```
C:\>dir
Directory of C:\.
.              <DIR>          06-02-2017 15:54
..             <DIR>          01-01-1980  0:00
EX             <DIR>          03-02-2017  8:32
4-1      ASM             193 05-02-2017 19:25
4-1      EXE             527 05-02-2017 19:26
4-1      OBJ              68 05-02-2017 19:25
DEBUG    EXE          20,634 10-01-2000 20:00
DESKTOP  INI             317 19-10-2008 14:34
EDIT     COM          69,886 10-01-2000 20:00
EDIT     EXE          30,776 04-04-1996 13:30
EDLIN    COM           4,608 08-03-1983 12:00
LINK     EXE          64,992 21-05-1992 10:22
MASM     EXE         103,184 21-05-1992 10:21
P4-1     ASM             178 05-02-2017 22:20
P4-1     EXE             527 06-02-2017 16:06
P4-1     OBJ              69 05-02-2017 22:20
   13 File(s)        295,959 Bytes.
    3 Dir(s)     262,111,744 Bytes free.

C:\>p4-1

C:\>
```

- 我们的程序没有像显示器输出任何信息。程序只是做了一些将数据送入寄存器和加法的操作，而这些事情，我们不可能从显示屏上看出来。
- 程序执行完成后，返回，屏幕上再次出现操作系统的提示符。

小结




```
C:\>masm p4-1.asm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights rese

Object filename [p4-1.OBJ]:
Source listing  [NUL.LST]:
Cross-reference [NUL.CRF]:

  51798 + 464746 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

```
C:\>link p4-1

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Run File [P4-1.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
LINK : warning L4021: no stack segment
```

```
C:\>p4-1

C:\>
```

源文件
.asm

目标文件
.obj

目标文件
.obj

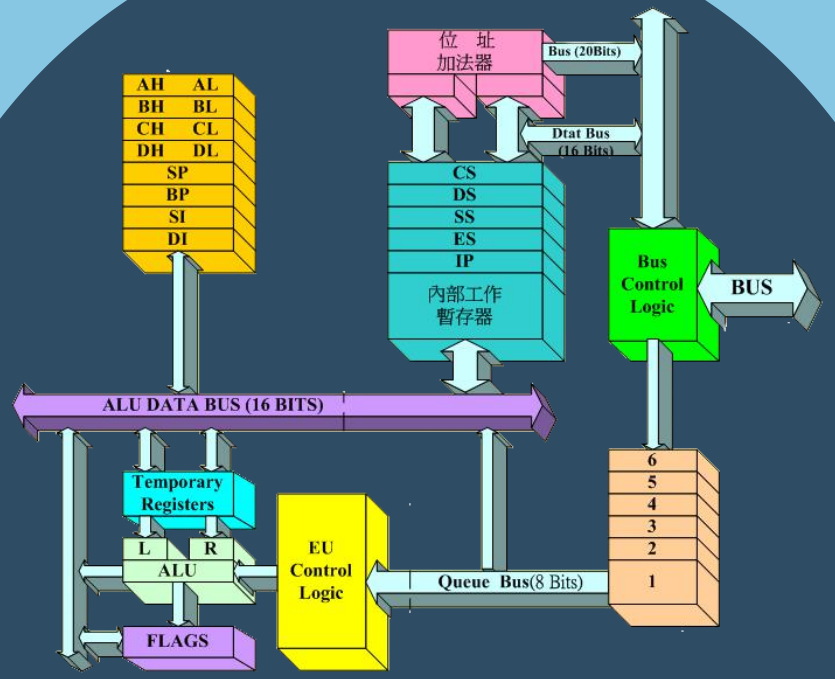
可执行文件
.exe

p4-1.asm	2017-2-5 22:20	ASM 文件	1 KB
P4-1.EXE	2017-2-6 15:54	应用程序	1 KB
P4-1.OBJ	2017-2-5 22:20	OBJ 文件	1 KB

... B82301 BB5604 01DB ...

运行及跟踪

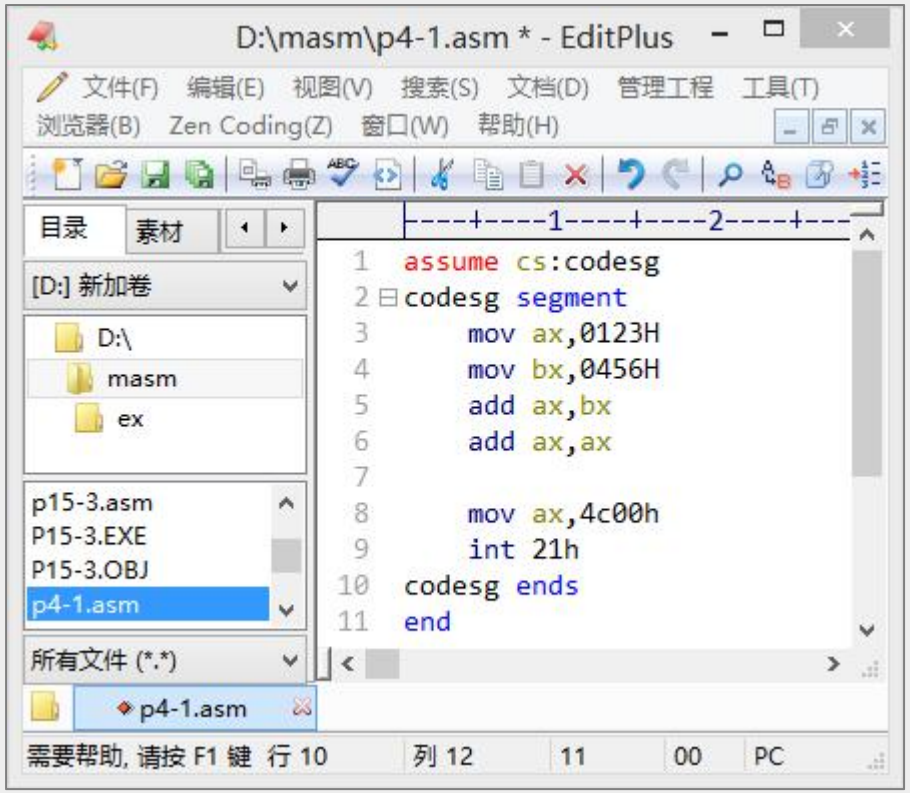
贺利坚　主讲



汇编语言程序设计

Assembly Language

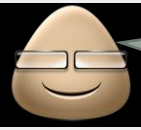
回顾




```
C:\>masm p4-1.asm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights rese

Object filename [p4-1.OBJ]:
Source listing  [NUL.LST]:
Cross-reference [NUL.CRF]:

  51798 + 464746 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

```
C:\>link p4-1

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Run File [P4-1.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
LINK : warning L4021: no stack segment
```

```
C:\>p4-1

C:\>
```



看看这个环节的门道。

源文件 .asm

↓

目标文件 .obj

目标文件 .obj

↓

可执行文件 .exe

p4-1.asm	2017-2-5 22:20	ASM 文件	1 KB
P4-1.EXE	2017-2-6 15:54	应用程序	1 KB
P4-1.OBJ	2017-2-5 22:20	OBJ 文件	1 KB

... B82301 BB5604 01DB ...

用Debug装载程序

```
1  assume cs:codesg
2  codesg segment
3      mov ax,0123H
4      mov bx,0456H
5      add ax,bx
6      add ax,ax
7
8      mov ax,4c00h
9      int 21h
10 codesg ends
11 end
```

DS=075A

有效代码共15(0FH)字节

```
C:\>debug p4-1.exe
-r
AX=FFFF  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0000   NV UP EI PL NZ NA PO NC
076A:0000 B82301        MOV     AX,0123
-
```

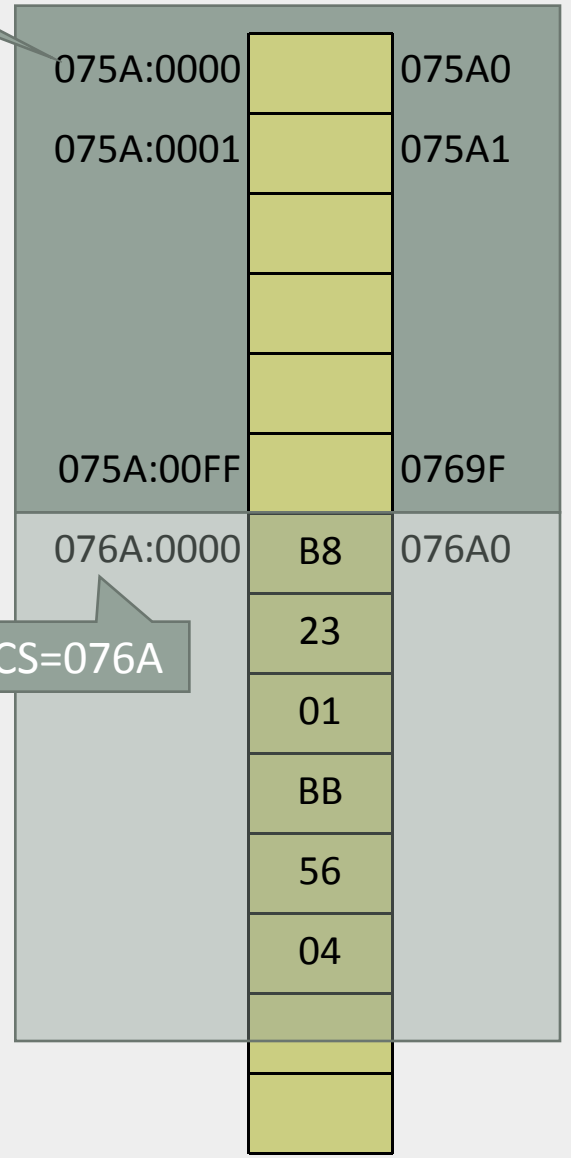
CS=076A

程序被装入内存的什么地方？

```
-u
076A:0000 B82301        MOV     AX,0123
076A:0003 BB5604        MOV     BX,0456
076A:0006 03C3          ADD     AX,BX
076A:0008 03C0          ADD     AX,AX
076A:000A B8004C        MOV     AX,4C00
076A:000D CD21          INT     21
076A:000F 7CF2          JL      0003
076A:0011 3D7400        CMP     AX,0074
076A:0014 7ED4          JLE     FFEA
076A:0016 EBEB          JMP     0003
```

DS=075A

一共256(100H)字节的程序段前缀(PSP)，作为数据区



小结

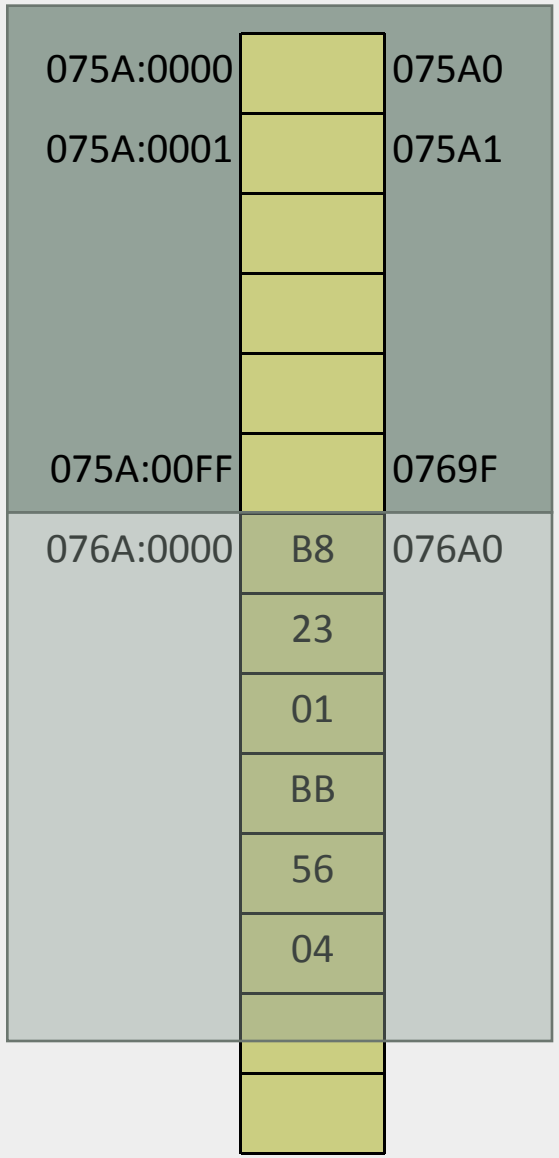
- 程序加载后，DS中存放着程序所在内存区的段地址，这个内存区的偏移地址为 0 ，则程序所在的内存区的地址为：DS:0。
- 这个内存区的前256 个字节存PSP，DOS用来和程序进行通信。
- 从 256字节处向后的空间存放的是程序，CS的值为DS+10H。
- 程序加载后，CX中存放代码的长度（字节）。

用Debug单步执行程序

```
1  assume cs:codesg
2  codesg segment
3      mov ax,0123H
4      mov bx,0456H
5      add ax,bx
6      add ax,ax
7
8      mov ax,4c00h
9      int 21h
10 codesg ends
11 end
```

```
C:\>debug p4-1.exe
-r
AX=FFFF  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0000   NV UP EI PL NZ NA PO NC
076A:0000 B82301        MOV     AX,0123
-t

AX=0123  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0003   NV UP EI PL NZ AC PO NC
076A:0003 BB5604        MOV     BX,0456
-t

AX=0123  BX=0456  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0006   NV UP EI PL NZ AC PO NC
076A:0006 03C3          ADD     AX,BX
-t

AX=0579  BX=0456  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0008   NV UP EI PL NZ NA PO NC
076A:0008 03C0          ADD     AX,AX
-t

AX=0AF2  BX=0456  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=000A   NV UP EI PL NZ AC PO NC
076A:000A B8004C        MOV     AX,4C00
-t

AX=4C00  BX=0456  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=000D   NV UP EI PL NZ AC PO NC
076A:000D CD21          INT     21
-t

AX=4C00  BX=0456  CX=000F  DX=0000  SP=FFFA  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=F000  IP=14A0   NV UP DI PL NZ AC PO NC
F000:14A0 FB            STI
```

一共256(100H)字节的程序段前缀(PSP)，作为数据区



其他方式执行

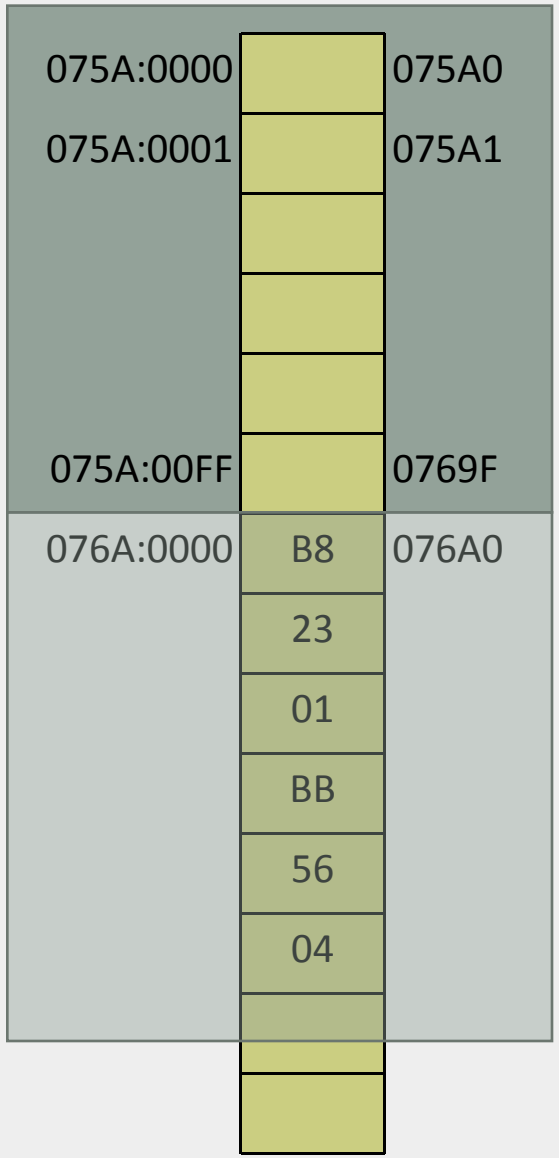
```
1  assume cs:codesg
2  codesg segment
3      mov ax,0123H
4      mov bx,0456H
5      add ax,bx
6      add ax,ax
7
8      mov ax,4c00h
9      int 21h
10 codesg ends
11 end
```

- 继续命令P(Proceed)：类似T命令，逐条执行指令、显示结果。但遇子程序、中断等时，直接执行，然后显示结果。
- 运行命令G(Go)：从指定地址处开始运行程序，直到遇到断点或者程序正常结束。

```
C:\>debug p4-1.exe
-r
AX=FFFF  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0000   NV UP EI PL NZ NA PO NC
076A:0000 B82301        MOV     AX,0123
-p

AX=0123  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0003   NV UP EI PL NZ NA PO NC
076A:0003 BB5604        MOV     BX,0456
-p

AX=0123  BX=0456  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0006   NV UP EI PL NZ NA PO NC
076A:0006 03C3          ADD     AX,BX
-p

AX=0579  BX=0456  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0008   NV UP EI PL NZ NA PO NC
076A:0008 03C0          ADD     AX,AX
-p

AX=0AF2  BX=0456  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=000A   NV UP EI PL NZ AC PO NC
076A:000A B8004C        MOV     AX,4C00
-p

AX=4C00  BX=0456  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=000D   NV UP EI PL NZ AC PO NC
076A:000D CD21          INT     21
-p

Program terminated normally
```

```
C:\>debug p4-1.exe
-r
AX=FFFF  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0000   NV UP EI PL NZ NA PO NC
076A:0000 B82301        MOV     AX,0123
-g

Program terminated normally
```

一共256(100H)字节的程序段前缀(PSP)，作为数据区



程序执行的不同方式

在DOS中执行

```
C:\>p4-1

C:\>
```

在Debug中执行

```
-t

AX=0123  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0003   NV UP EI PL NZ AC PO NC
076A:0003 BB5604        MOV     BX,0456
-t

AX=0123  BX=0456  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0006   NV UP EI PL NZ AC PO NC
076A:0006 03C3          ADD     AX,BX
-p

AX=4C00  BX=0456  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=000D   NV UP EI PL NZ AC PO NC
076A:000D CD21          INT     21
-p

Program terminated normally
-
```

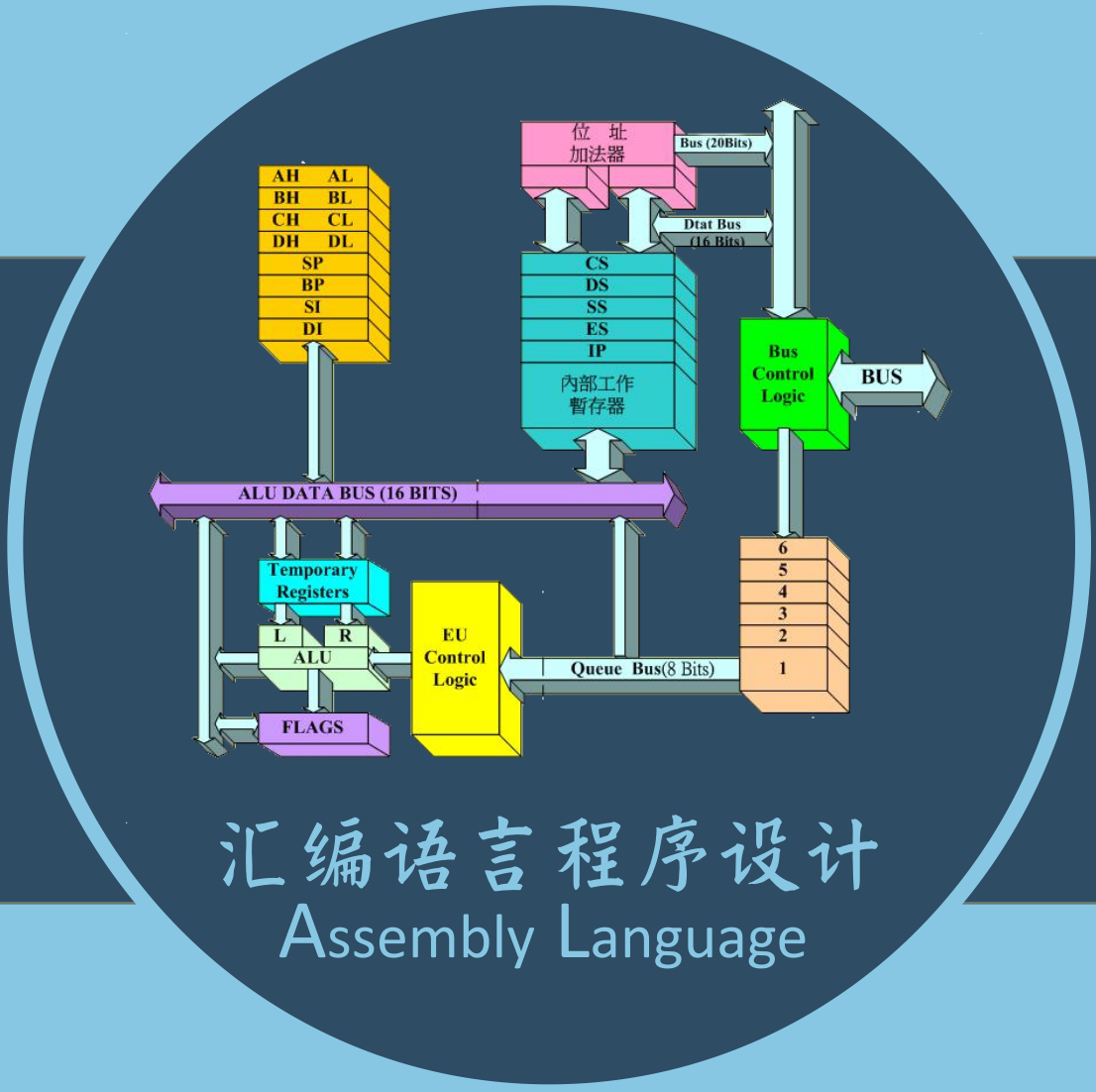
```
C:\>debug p4-1.exe
-r
AX=FFFF  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0000   NV UP EI PL NZ NA PO NC
076A:0000 B82301        MOV     AX,0123
-g

Program terminated normally
-
```

- 程序执行的“常态”
- DOS启动后，计算机由“命令解释器”（程序command.com）控制
- 运行可执行程序时，command将程序加载入内存，设置CPU的CS:IP指向程序的第一条指令（即程序的入口），使程序得以运行。
- 程序运行结束后，返回到“命令解释器”，CPU继续运行command。

- 程序执行处于开发周期的运行方式；
- 运行Debug时，command程序加载Debug.exe，debug 将程序加载入内存，程序运行结束后要返回到Debug中，使用Q命令退出Debug，将返回到command中。

[...]和(...)

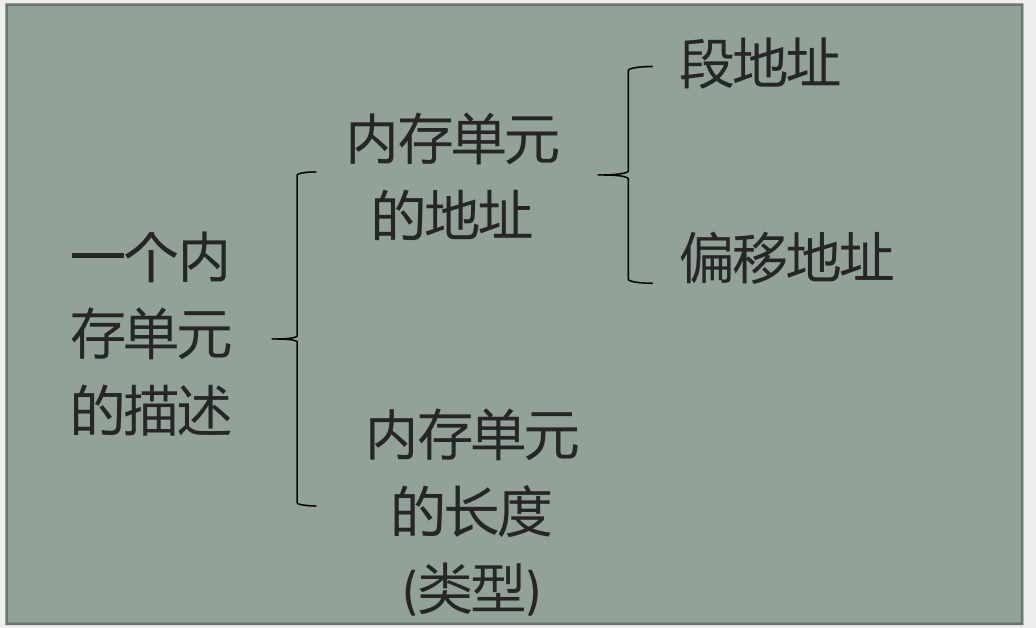
贺利坚　主讲


位 址
加法器
Bus (20Bits)
Dtat Bus
(16 Bits)
AH AL
BH BL
CH CL
DH DL
SP
BP
SI
DI
CS
DS
SS
ES
IP
内部工作
暂存器
Bus
Control
Logic
BUS
ALU DATA BUS (16 BITS)
Temporary
Registers
L
R
ALU
FLAGS
EU
Control
Logic
Queue Bus(8 Bits)
6
5
4
3
2
1
汇编语言程序设计
Assembly Language

[...]的规定与(...)的约定

[...]——(汇编语法规定)表示一个内存单元

指令	段地址	偏移地址	操作单位
mov ax, [0]	在DS中	在[0]中	字
mov al, [0]	在DS中	在[0]中	字节
mov ax,[bx]	在DS中	在[bx]中	字
mov al,[bx]	在DS中	在[bx]中	字节



(...)——(为学习方便做出的约定)表示一个内存单元或寄存器中的内容

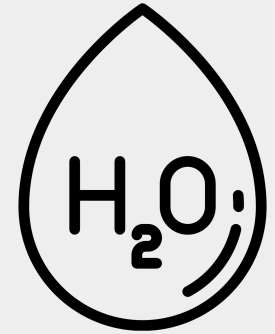
只能用寄存器及物理地址

描述对象	描述方法	描述对象	描述方法
ax中的内容为0010H	(ax)=0010H	2000:1000 处的内容为0010H	(21000H)=0010H
mov ax,[2]的功能	(ax)=((ds)*16+2)	mov [2], ax的功能	((ds)*16+2)=(ax)
add ax,2 的功能	(ax)=(ax)+2	add ax,bx的功能	(ax)=(ax)+(bx)
push ax的功能	(sp) = (sp)-2 ((ss)*16 + (sp))=(ax)	pop ax 的功能	(ax)=((ss)*16+(sp)) (sp)=(sp)+2

再约定：符号idata表示常量

例：

- mov ax,[idata]　：代表mov ax,[1]、 mov ax,[2]、 mov ax,[3]...
- mov bx,idata　：代表mov bx,1、 mov bx,2、 mov bx,3...
- mov ds,idata　：代表mov ds,1、 mov ds,2...(都是非法指令)



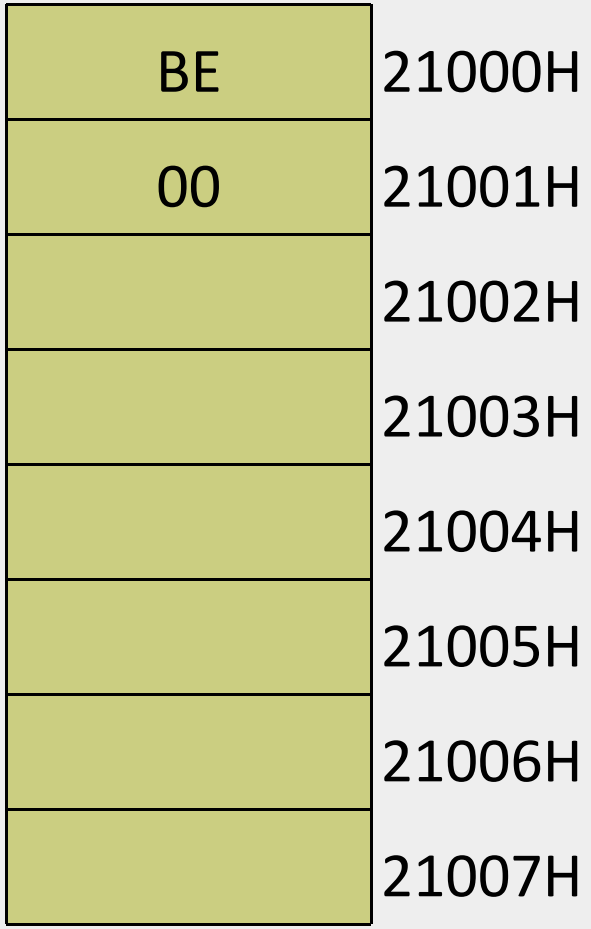
案例分析

```
mov ax,2000H
mov ds,ax
mov bx,1000H
mov ax,[bx]
inc bx
inc bx
mov [bx],ax
inc bx
inc bx
inc bx
mov [bx],ax
inc bx
mov [bx],al
inc bx
mov [bx],al
```

AX=	**BX=**
DS=	**ES=**

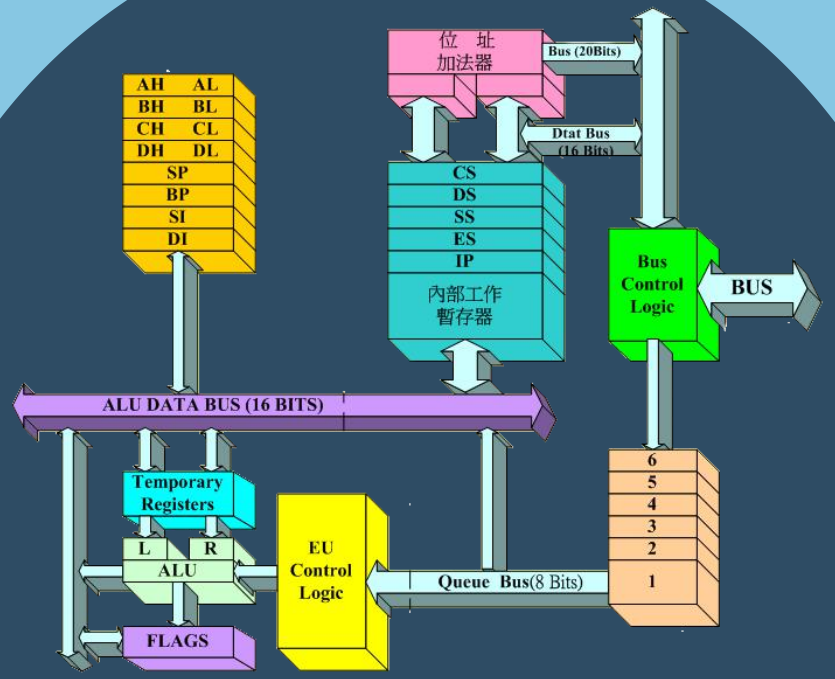
mov ax,[bx] --- (ax)=((ds) *16 +(bx))

mov [bx],ax --- ((ds)*16 +(bx))=(ax)



Loop指令

贺利坚　主讲



汇编语言程序设计

Assembly Language

Loop指令

- 功能：实现循环（计数型循环）
- 指令的格式

loop 标号

- CPU 执行loop指令时要进行的操作
  - ① (cx)=(cx)-1；
  - ② 判断cx中的值

    不为零则转至标号处执行程序

    如果为零则向下执行。
- 要求
  - cx 中要提前存放循环次数，因为(cx)影响着loop指令的执行结果
  - 要定义一个标号

```
1  ; loop指令示例程序
2  assume cs:code
3  code segment
4      mov ax,2
5      mov cx,11
6   s: add ax,ax
7      loop s
8
9      mov ax,4c00h
10     int 21h
11 code ends
12 end
```

本程序功能：2 -> 4 -> 8 -> 16 -> 32 -> ...

用loop指令编程实例

任务1：编程计算2^2

```
assume cs:code
code segment
        mov ax,2
        ; 用2+2 实现2*2
        add ax,ax

        mov ax,4c00h
        int 21h
code ends
end
```

任务2：编程计算2^3

```
assume cs:code
code segment
        mov ax,2
        add ax,ax
        add ax,ax

        mov ax,4c00h
        int 21h
code ends
end
```

任务3：编程计算2^12

```
assume cs:code
code segment
        mov ax,2
        ; 做11次add ax,ax

        mov ax,4c00h
        int 21h
code ends
end
```

```
 1  assume cs:code
 2  code segment
 3          mov ax,2
 4          mov cx,11
 5     s:   add ax,ax
 6          loop s
 7
 8          mov ax,4c00h
 9          int 21h
10  code ends
11  end
```

用cx和loop 指令相配合实现循环功能的三个要点：

（1）在cx中存放循环次数；

（2）用标号指定循环开始的位置；

（3）在标号和loop 指令的中间，写上要循环执行的程序段（循环体）。

用Debug执行程序

```
C:\>masm p5-1;
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.


  51798 + 464746 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors

C:\>link p5-1;

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

LINK : warning L4021: no stack segment
```

```
C:\>debug p5-1.exe
-u
076A:0000 B80200        MOV     AX,0002
076A:0003 B90B00        MOV     CX,000B
076A:0006 03C0          ADD     AX,AX
076A:0008 E2FC          LOOP    0006
076A:000A B8004C        MOV     AX,4C00
076A:000D CD21          INT     21
076A:000F 26FA          JBE     000B
```

```
C:\>debug p5-1.exe
-g

Program terminated normally
-
```

```
-g 0006

AX=0002  BX=0000  CX=000B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0006   NV UP EI PL NZ NA PO NC
076A:0006 03C0          ADD     AX,AX
-g 000D

AX=4C00  BX=0000  CX=0000  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=000D   NV UP EI PL NZ NA PE NC
076A:000D CD21          INT     21
```

```
-r
AX=FFFF  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0000   NV UP EI PL NZ NA PO NC
076A:0000 B80200        MOV     AX,0002
-t

AX=0002  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0003   NV UP EI PL NZ NA PO NC
076A:0003 B90B00        MOV     CX,000B
-t

AX=0002  BX=0000  CX=000B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0006   NV UP EI PL NZ NA PO NC
076A:0006 03C0          ADD     AX,AX
-t

AX=0004  BX=0000  CX=000B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0008   NV UP EI PL NZ NA PO NC
076A:0008 E2FC          LOOP    0006
-t

AX=0004  BX=0000  CX=000A  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0006   NV UP EI PL NZ NA PO NC
076A:0006 03C0          ADD     AX,AX
-
```

t命令和p命令的区别

```
-r
AX=FFFF  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0000   NV UP EI PL NZ NA PO NC
076A:0000 B80200        MOV     AX,0002
-t

AX=0002  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0003   NV UP EI PL NZ NA PO NC
076A:0003 B90B00        MOV     CX,000B
-t

AX=0002  BX=0000  CX=000B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0006   NV UP EI PL NZ NA PO NC
076A:0006 03C0          ADD     AX,AX
-t

AX=0004  BX=0000  CX=000B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0008   NV UP EI PL NZ NA PO NC
076A:0008 E2FC          LOOP    0006
-t

AX=0004  BX=0000  CX=000A  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0006   NV UP EI PL NZ NA PO NC
076A:0006 03C0          ADD     AX,AX
-
```

```
-g 0006

AX=0002  BX=0000  CX=000B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0006   NV UP EI PL NZ NA PO NC
076A:0006 03C0          ADD     AX,AX
-g 000D

AX=4C00  BX=0000  CX=0000  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=000D   NV UP EI PL NZ NA PE NC
076A:000D CD21          INT     21
```

```
-p

AX=0002  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0003   NV UP EI PL NZ NA PO NC
076A:0003 B90B00        MOV     CX,000B
-p

AX=0002  BX=0000  CX=000B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0006   NV UP EI PL NZ NA PO NC
076A:0006 03C0          ADD     AX,AX
-p

AX=0004  BX=0000  CX=000B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0008   NV UP EI PL NZ NA PO NC
076A:0008 E2FC          LOOP    0006
-p

AX=1000  BX=0000  CX=0000  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=000A   NV UP EI PL NZ NA PE NC
076A:000A B8004C        MOV     AX,4C00
```

- 继续命令P(Proceed)：类似T命令，逐条执行指令、显示结果。但遇子程序、中断等时，直接执行，然后显示结果。
- 运行命令G(Go)：从指定地址处开始运行程序，直到遇到断点或者程序正常结束；G命令还可以指定执行到的代码地址。

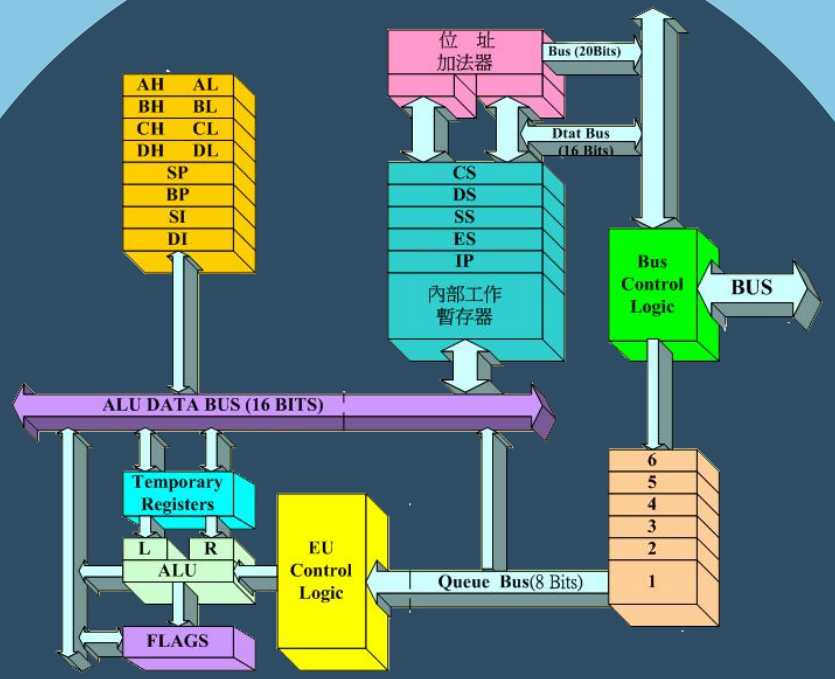
例：用Loop指令编程

- 问题：计算123x236，结果存储在ax中
- 方法：用加法实现乘法，将123连加236次

```
1   assume cs:code
2   code segment
3       mov ax,0
4       mov cx,236
5    s: add ax,123
6       loop s
7
8       mov ax,4c00h
9       int 21h
10  code ends
11  end
```

Loop指令使用再例

贺利坚　主讲



汇编语言程序设计

Assembly Language

再例：用Loop指令编程

问题：计算ffff:0006字节单元中的数乘以3，结果存储在dx中 → 先将内存中数据取出；连加3次，即乘以3。

程序

```
1   assume cs:code
2   code segment
3           mov ax,0ffffh
4           mov ds,ax
5           mov bx,6
6           mov al,[bx]
7           mov ah,0
8
9           mov dx,0
10          mov cx,3
11      s:  add dx,ax
12          loop s
13
14          mov ax,4c00h
15          int 21h
16  code ends
17  end
```

在汇编源程序中，数据不能以字母开头，要在ffff前面加0

(ax)=((ds) *16 +(bx))

设置循环次数

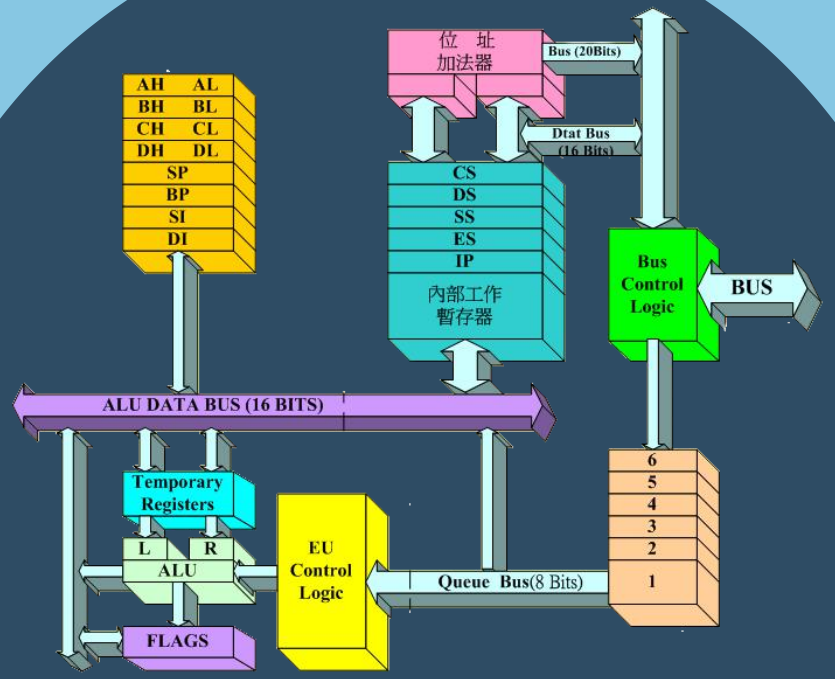
结果在dx中

其他必要的考虑：运算后的结果是否会超出dx所能存储的范围？

分析：ffff:0006 单元中的数是一个字节型的数据，范围在0~255之间，则用它和3相乘结果不会大于65535，不会出现超界。

段前缀的使用

贺利坚　主讲



汇编语言程序设计

Assembly Language

引入段前缀：一个“异常”现象及对策

```
C:\>debug
-a
073F:0100 mov ax, 2000
073F:0103 mov ds, ax
073F:0105 mov al, [0]
073F:0108 mov bl, [1]
073F:010C mov cl, [2]
073F:0110 mov dl, [3]
073F:0114
-u
073F:0100 B80020        MOV     AX,2000
073F:0103 8ED8          MOV     DS,AX
073F:0105 A00000        MOV     AL,[0000]
073F:0108 8A1E0100      MOV     BL,[0001]
073F:010C 8A0E0200      MOV     CL,[0002]
073F:0110 8A160300      MOV     DL,[0003]
073F:0114 0000          ADD     [BX+SI],AL
```

Debug中，mov al, [0]的功能是

——将DS:0存储单元的值传给AL

```
1  assume cs:code
2  code segment
3          mov ax,2000h
4          mov ds,ax
5          mov al,[0]
6          mov bl,[1]
7          mov cl,[2]
8          mov dl,[3]
9
10         mov ax,4c00h
11         int 21h
12 code ends
13 end
```

编译(masm)并连接(link)后...

```
C:\>debug p5-3.exe
-u
076A:0000 B80020        MOV     AX,2000
076A:0003 8ED8          MOV     DS,AX
076A:0005 B000          MOV     AL,00
076A:0007 B301          MOV     BL,01
076A:0009 B102          MOV     CL,02
076A:000B B203          MOV     DL,03
076A:000D B8004C        MOV     AX,4C00
076A:0010 CD21          INT     21
076A:0012 00508B        ADD     [BX+SI-25
```

编译好的程序中，mov al, [0]变成了将常量0传给AL

对策：在[idata]前显式地写上段寄存器

```
mov ax,2000h
mov ds,ax
mov bx,0
mov al,ds:[bx]
```

```
mov ax,2000h
mov ds,ax
mov al,ds:[0]
```

小结（在程序中）：

mov al,[0]：(al)=0，同mov al,0

mov al,ds:[0]：(al)=((ds)*16+0)

mov al,[bx]：(al)=((ds)*16+(bx))

mov al,ds:[bx]：与mov al,[bx]相同

这些出现在访问内存单元的指令中，用于显式地指明内存单元的段地址的“ds:”、“cs:”、“ss:”或“es:”，在汇编语言中称为**段前缀**。

访问连续的内存单元——loop和[bx]联手！

- 问题：计算ffff:0~ffff:b字节单元中的数据的和，结果存储在dx中
- 分析：

（1）运算后的结果是否会超出 dx 所能存储的范围？

ffff:0 ~ ffff:b内存单元中的数据是字节型数据，范围在0 ~ 255之间，12个这样的数据相加，结果不会大于 65535，可以在dx中存放下。

（2）是否可以将 ffff:0 ~ ffff:b中的数据直接累加到dx中？

add **dx**, ds:[addr] ;(dx)=(dx)+?

期望：取出内存中的8位数据进行相加

实际：取出的是内存中的16位数据

（3）是否可以将 ffff:0 ~ ffff:b中的数据直接累加到dl中？

add **dl**, ds:[addr] ;(dl)=(dl)+?

期望：取出内存中的8位数据相加

实际：取出的是内存中的8位数据，但很有可能造成进位丢失。

- 对策：取出8位数据，加到16位的寄存器

```
mov al, ds:[addr]
mov ah, 0
add dx, ax
```

程序：计算ffff:0~ffff:b单元中的数据的和，结果存储在dx中

```
assume cs:code
code segment
    mov ax,0ffffh
    mov ds,ax

    mov dx,0

    mov al,ds:[0]
    mov ah,0
    add dx,ax

    mov al,ds:[1]
    mov ah,0
    add dx,ax

    mov al,ds:[2]
    mov ah,0
    add dx,ax

    mov al,ds:[3]
    mov ah,0
    add dx,ax

    mov al,ds:[4]
    mov ah,0
    add dx,ax

    mov al,ds:[5]
    mov ah,0
    add dx,ax

    mov al,ds:[6]
    mov ah,0
    add dx,ax

    mov al,ds:[7]
    mov ah,0
    add dx,ax

    mov al,ds:[8]
    mov ah,0
    add dx,ax

    mov al,ds:[9]
    mov ah,0
    add dx,ax

    mov al,ds:[0ah]
    mov ah,0
    add dx,ax

    mov al,ds:[0bh]
    mov ah,0
    add dx,ax

    mov ax,4c00h
    int 21h
code ends
end
```

$$sum = \sum_{x=0}^{0bh} (0ffffh \times 10h + x)$$

- 改进
  - 用loop循环
- 方法
  - 循环次数由cx控制
  - 循环中要访问的内存单元的偏移地址放到 bx中，随循环递增，访问连续的内存单元。

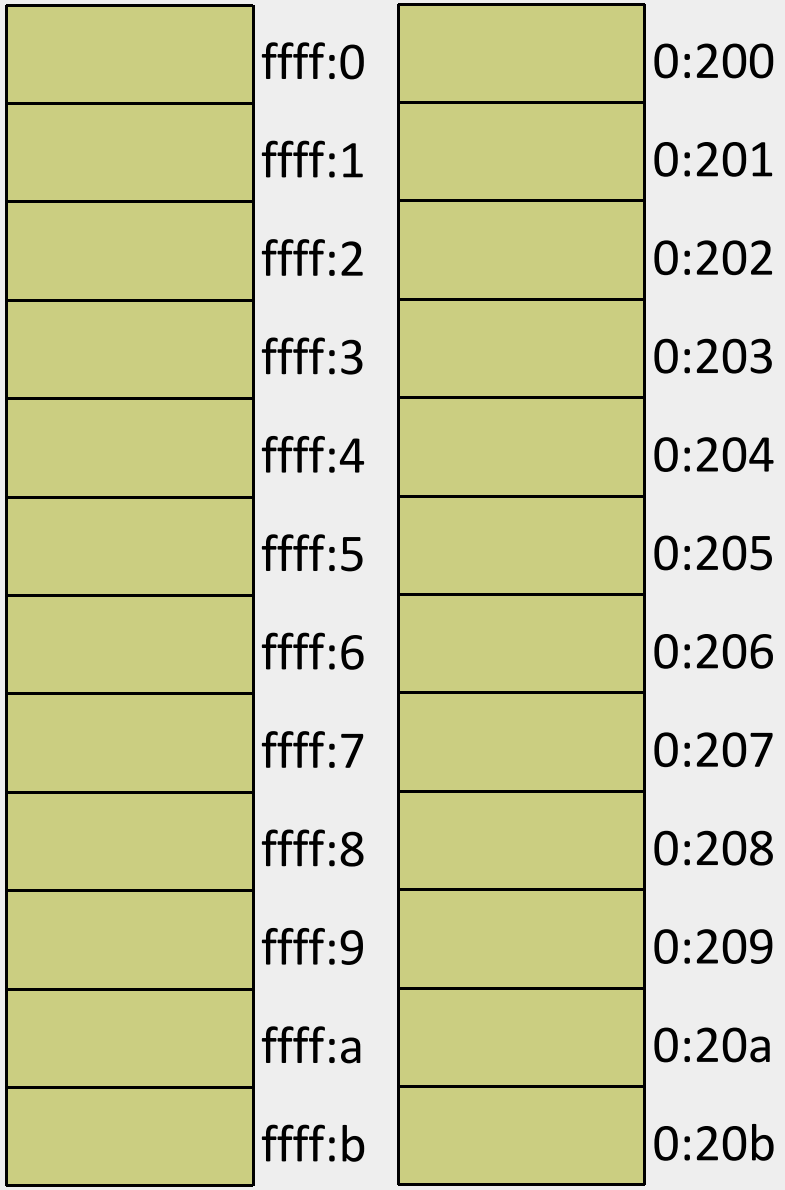
```
1  assume cs:code
2  code segment
3      mov ax,0ffffh
4      mov ds,ax
5
6      mov bx,0
7      mov dx,0
8      mov cx,12
9
10 s:  mov al,[bx]
11     mov ah,0
12     add dx,ax
13     inc bx
14     loop s
15
16     mov ax,4c00h
17     int 21h
18 code ends
19 end
```

段前缀的使用

问题：将内存ffff:0~ffff:b中的数据拷贝到 0:200~0:20b单元中。

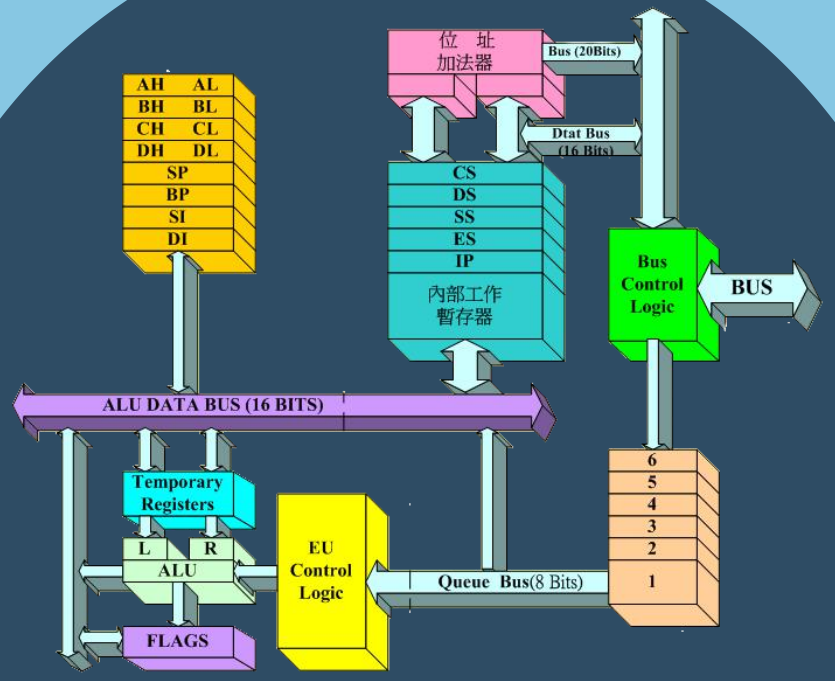
```
1  ; 初始方案
2  assume cs:code
3  code segment
4          mov bx,0
5          mov cx,12
6
7    s:    mov ax,0ffffh
8          mov ds,ax
9          mov dl,[bx]
10
11         mov ax,0020h
12         mov ds,ax
13         mov [bx],dl
14
15         inc bx
16         loop s
17
18         mov ax,4c00h
19         int 21h
20 code ends
21 end
```

```
1  ;使用附加段寄存器
2  assume cs:code
3  code segment
4          mov ax,0ffffh
5          mov ds,ax
6          mov ax,0020h
7          mov es,ax
8
9          mov bx,0
10         mov cx,12
11
12   s:    mov dl,[bx]
13         mov es:[bx],dl
14         inc bx
15         loop s
16
17         mov ax,4c00h
18         int 21h
19 code ends
20 end
```



在代码段中使用数据

贺利坚　主讲



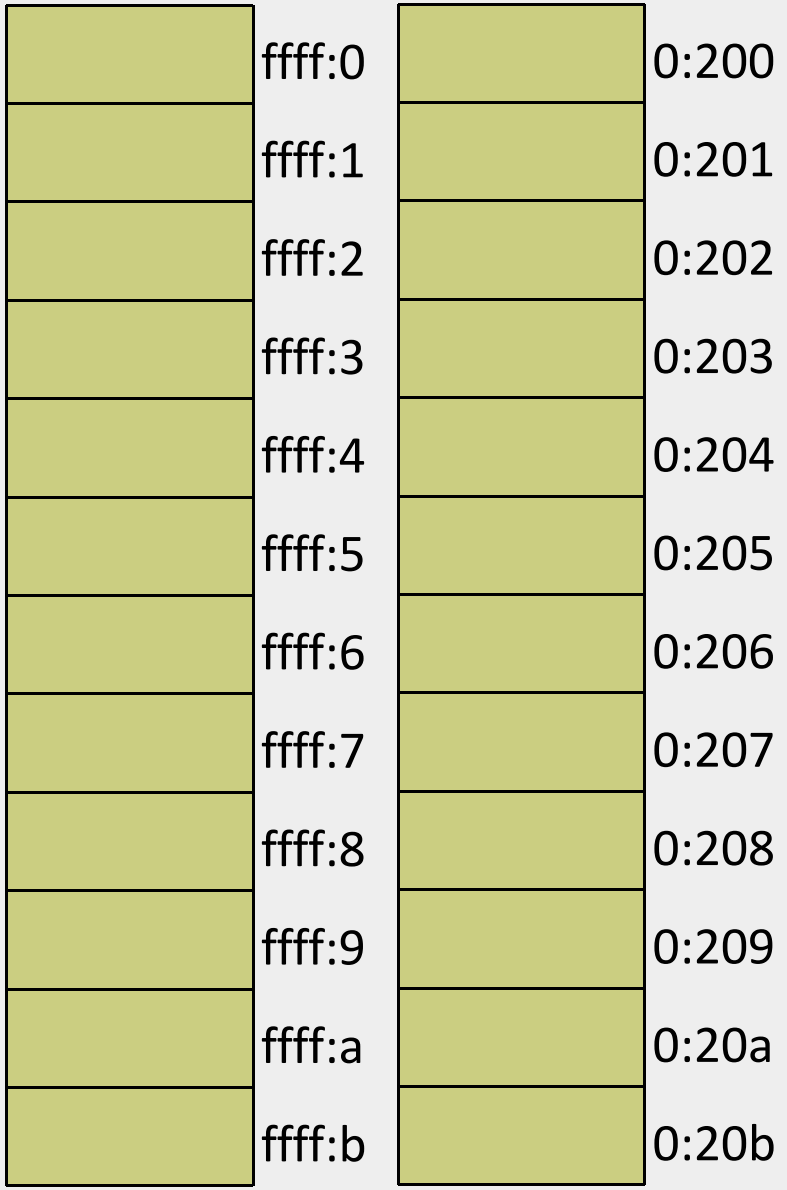
汇编语言程序设计

Assembly Language

问题：这样做是危险的！

- 例：将内存ffff:0~ffff:b中的数据拷贝到0:200~0:20b单元中。
- 问题
  - 程序中直接写地址，危险！
  - “安全”位置存放数据，存哪里？
- 对策
  - 在程序的段中存放数据，运行时由操作系统分配空间。
  - 段的类别：数据段、代码段、栈段
  - 各种段中均可以有数据
  - 可以在单个的段中安置，也可以将数据、代码、栈放入不同的段中。

```
1  ;使用附加段寄存器
2  assume cs:code
3  code segment
4      mov ax,0ffffh
5      mov ds,ax
6      mov ax,0020h
7      mov es,ax
8
9      mov bx,0
10     mov cx,12
11
12   s: mov dl,[bx]
13     mov es:[bx],dl
14     inc bx
15     loop s
16
17     mov ax,4c00h
18     int 21h
19 code ends
20 end
```



应用案例

- 问题：编程计算以下8个数据的和，结果存在ax 寄存器中

  0123H，0456H，0789H，0abcH，0defH，0fedH，0cbaH，0987H

- 解决方案1

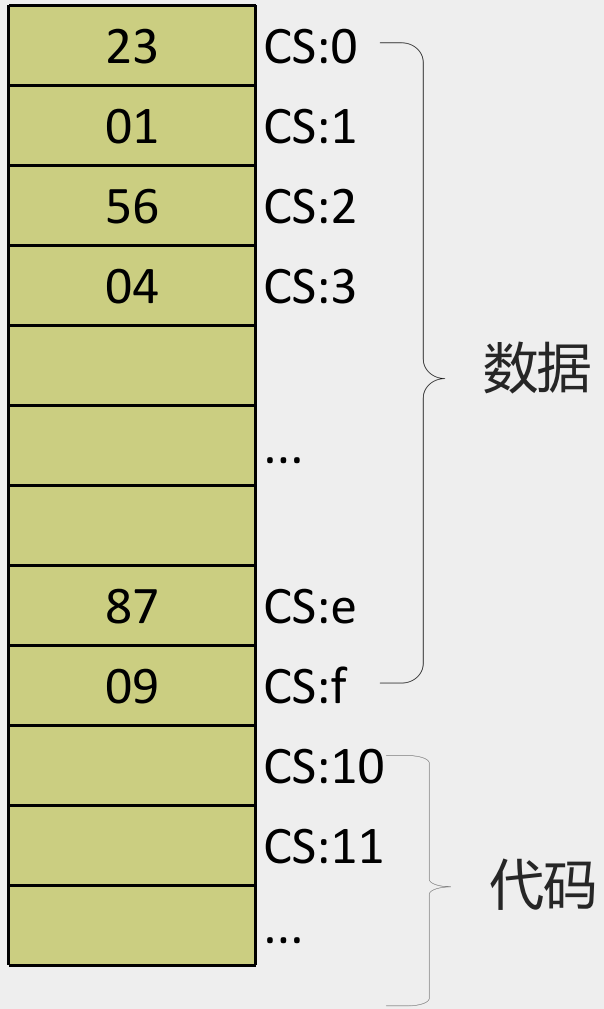
只要求数据本身，并未指定在哪些内存单元中！

在代码段中定义数据

```
1  assume cs:code
2  code segment
3      dw 0123H,0456H,0789H,0abcH,0defH,0fedH,0cbaH,0987H
4
5      mov bx,0
6      mov ax,0
7      mov cx,8
8
9    s: add ax,cs:[bx]
10     add bx,2
11     loop s
12
13     mov ax,4c00h
14     int 21h
15 code ends
16 end
```

dw: define word，定义字型数据

dw 定义一个字
db 定义一个字节
dd 定义一个双字



这个程序有问题！

```
1  assume cs:code
2  code segment
3      dw 0123H,0456H,0789H,0abcH,0defH,0fedH,0cbaH,0987H
4
5      mov bx,0
6      mov ax,0
7      mov cx,8
8
9    s: add ax,cs:[bx]
10     add bx,2
11     loop s
12
13     mov ax,4c00h
14     int 21h
15 code ends
16 end
```

真正的代码并不应该从0000开始

```
C:\>debug p6-1.exe
-r
AX=FFFF  BX=0000  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0000   NV UP EI PL NZ NA PO NC
076A:0000 2301          AND     AX,[BX+DI]                    DS:0000=20CD
-u
076A:0000 2301          AND     AX,[BX+DI]
076A:0002 56            PUSH    SI
076A:0003 0489          ADD     AL,89
076A:0005 07            POP     ES
076A:0006 BC0AEF        MOV     SP,EF0A
076A:0009 0DED0F        OR      AX,0FED
076A:000C BA0C87        MOV     DX,870C
076A:000F 09BB0000      OR      [BP+DI+0000],DI
076A:0013 B80000        MOV     AX,0000
076A:0016 B90800        MOV     CX,0008
076A:0019 2E            CS:
076A:001A 0307          ADD     AX,[BX]
076A:001C 83C302        ADD     BX,+02
076A:001F E2F8          LOOP    0019
076A:0021 B8004C        MOV     AX,4C00
076A:0024 CD21          INT     21
```

完全乱套的代码！

其实这一段本来就是数据+代码！

真正的代码，从0010开始

```
-d
076A:0000  23 01 56 04 89 07 BC 0A-EF 0D ED 0F BA 0C 87 09   #.V.............
076A:0010  BB 00 00 B8 00 00 B9 08-00 2E 03 07 83 C3 02 E2   ................
076A:0020  F8 B8 0  CD 21 E8 9F-0E 83 C4 04 3D FF FF 74   ...L.!......=..t
076A:0030  03      8B 56 FE 05 0C   ...../.P.F..V...
076A:0040  00      7B 0E 83 C4 04   .RP..H...P.{....
076A:0050  3D      26 8A 47 0C 2A   =..t.....^.&.G.*
```

从0000开始的是数据！

```
-u 0010
076A:0010 BB0000        MOV     BX,0000
076A:0013 B80000        MOV     AX,0000
076A:0016 B90800        MOV     CX,0008
076A:0019 2E            CS:
076A:001A 0307          ADD     AX,[BX]
076A:001C 83C302        ADD     BX,+02
076A:001F E2F8          LOOP    0019
076A:0021 B8004C        MOV     AX,4C00
076A:0024 CD21          INT     21
```

💻解决问题的关键：让数据从CS:0000开始，让代码从CS:0010开始！

这样改进

- 问题：编程计算以下8个数据的和，结果存在ax 寄存器中

  0123H，0456H，0789H，0abcH，0defH，0fedH，0cbaH，0987H

- 解决方案2

```
1  assume cs:code
2  code segment
3          dw 0123H,0456H,0789H,0abcH,0defH,0fedH,0cbaH,0987H
4
5  start:  mov bx,0
6          mov ax,0
7          mov cx,8
8
9      s:  add ax,cs:
10         add bx,2
11         loop s
12
13         mov ax
14         int
15 code ends
16 end start
```

定义一个标号，指示代码开始的位置。

end的作用：除了通知编译器程序结束外，还可以通知编译器程序的入口在什么地方。

```
assume cs:code
code segment
        :
      数据
        :
begin:
        :
        :
      代码
        :
        :
code ends
end begin
```

程序的一般框架

- 效果：程序加载后，CS:IP指向要执行的第一条指令在start处！

改过的程序木有问题鸟！

```
1  assume cs:code
2  code segment
3          dw 0123H,0456H,0789H,0abcH,0defH,0fedH,0cbaH,0987H
4
5  start: mov bx,0
6          mov ax,0
7          mov cx,8
8
9      s: add ax,cs:[bx]
10         add bx,2
11         loop s
12
13         mov ax,4c00h
14         int 21h
15 code ends
16 end start
```

```
C:\>debug p6-2.exe
-r
AX=FFFF  BX=0000  CX=0026  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0010   NV UP EI PL NZ NA PO NC
076A:0010 BB0000        MOV     BX,0000
-u
076A:0010 BB0000        MOV     BX,0000
076A:0013 B80000        MOV     AX,0000
076A:0016 B90800        MOV     CX,0008
076A:0019 2E            CS:
076A:001A 0307          ADD     AX,[BX]
076A:001C 83C302        ADD     BX,+02
076A:001F E2F8          LOOP    0019
076A:0021 B8004C        MOV     AX,4C00
076A:0024 CD21          INT     21
076A:0026 E89F0E        CALL    0EC8
076A:0029 83C404        ADD     SP,+04
076A:002C 3DFFFF        CMP     AX,FFFF
076A:002F 7403          JZ      0034
-
```

真正的代码从0010开始

是这些代码！

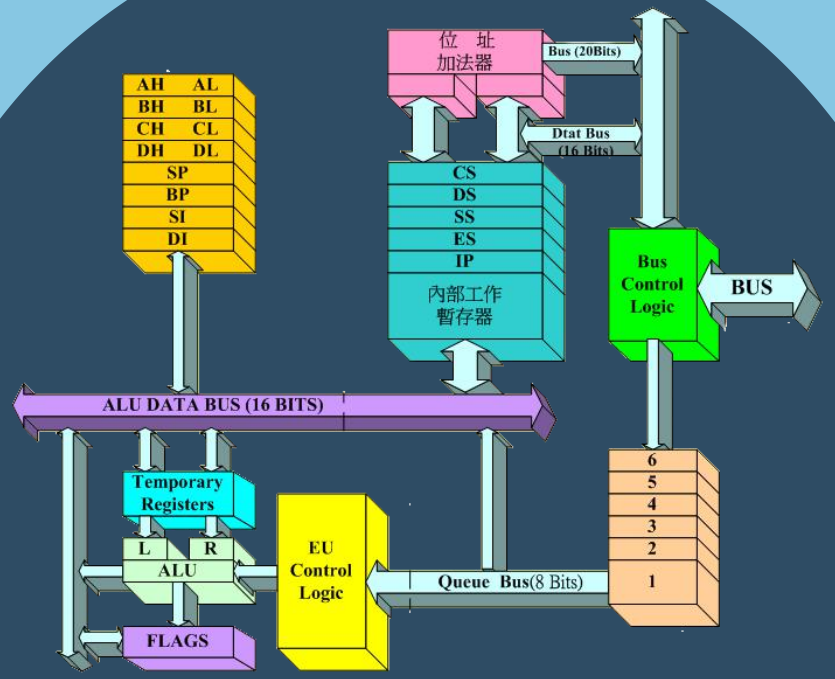
```
-d cs:0
076A:0000  23 01 56 04 89 07 BC 0A-EF 0D ED 0F BA 0C 87 09   #.V.............
076A:0010  BB 00 00 B8 00 00 B9 08-00 2E 03 07 83 C3 02 E2   ................
076A:0020  F8 B8 [illegible] CD 21 E8 9F-0E 83 C4 04 3D FF FF 74   ...L.!......=..t
076A:0030  [illegible] C 8B 56 FE 05 0C   ...../.P.F..V...
```

从CS:0000处开始的依然是数据！

在代码段中使用数据

在代码段中使用栈

贺利坚　主讲



汇编语言程序设计

Assembly Language

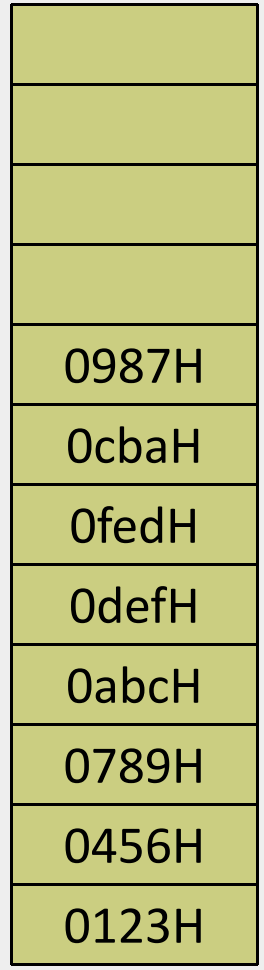
在代码段中使用栈：以数据逆序存放为例

问题：完成下面的程序，利用栈，将程序中定义的数据逆序存放。

```
assume cs:codesg
codesg segment
    dw 0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h
    ?

code ends
end
```

程序的思路大致如下：

- 程序运行时，定义的数据存放在cs:0~cs:F单元中，共8个字单元。
- 依次将这8个字单元中的数据入栈，然后再依次出栈到这 8 个字单元中，从而实现数据的逆序存放。
- 栈需要的内存空间，在程序中通过定义“空”数据来取得。

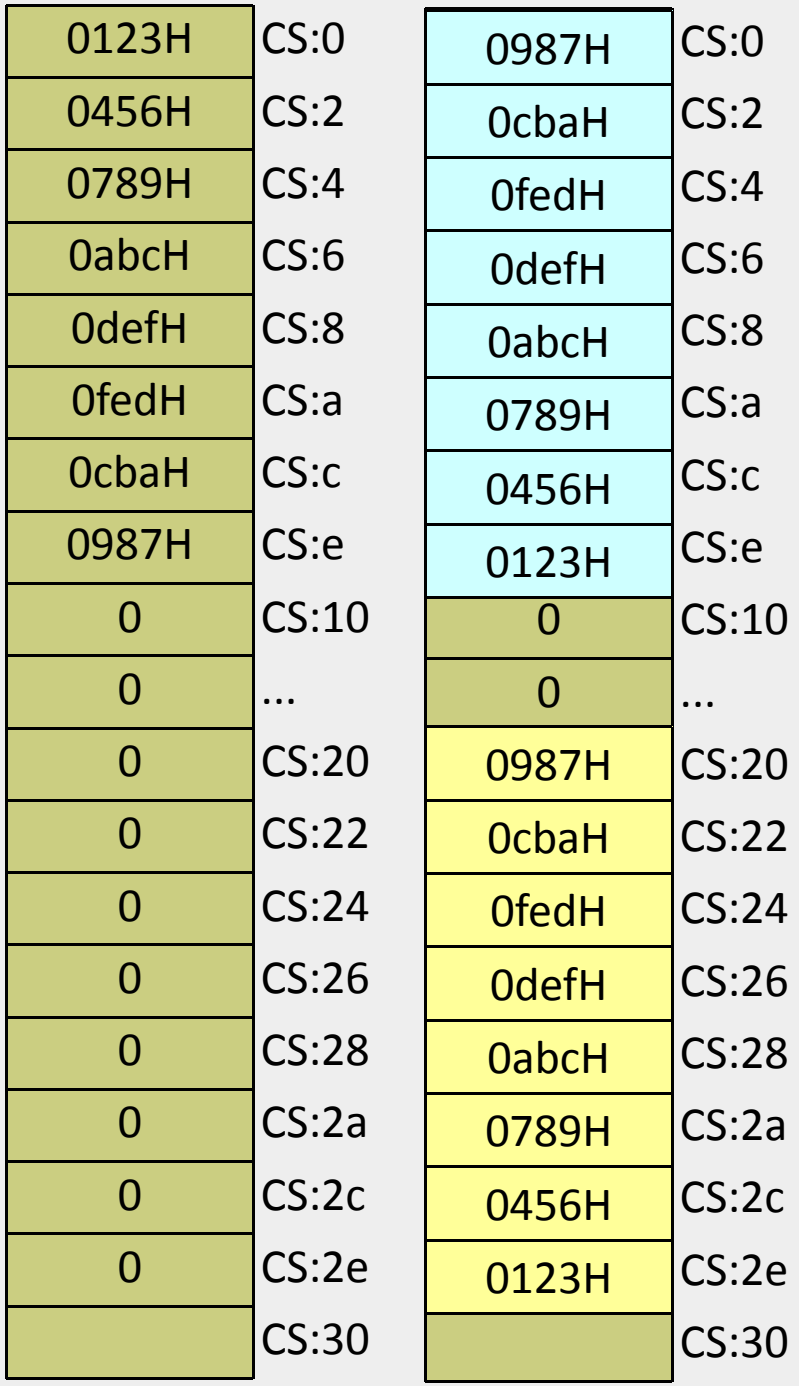


入栈后的数据

数据逆序存放程序

问题：完成下面的程序，利用栈，将程序中定义的数据逆序存放。

```
assume cs:codesg
codesg segment
        dw 0123H,0456H,0789H,0abcH,0defH,0fedH,0cbaH,0987H
        dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
start: mov ax,cs
        mov ss,ax
        mov sp,30h
        ;入栈
        mov bx,0
        mov cx,8
    s: push cs:[bx]
        add bx,2
        loop s
        ;出栈
        mov bx,0
        mov cx,8
    s0: pop cs:[bx]
        add bx,2
        loop s0
        mov ax,4c00h
        int 21h
codesg ends
end start
```



在Debug中的执行结果

```
1  assume cs:codesg
2  codesg segment
3          dw 0123H,0456H,0789H,0abcH,0defH,0fedH,0cbaH,0987H
4          dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
5
6   start: mov ax,cs
7          mov ss,ax
8          mov sp,30h
9          mov bx,0
10         mov cx,8
11     s:  push cs:[bx]
12         add bx,2
13         loop s
14
15         mov bx,0
16         mov cx,8
17     s0: pop cs:[bx]
18         add bx,2
19         loop s0
20
21         mov ax,4c00h
22         int 21h
23 codesg ends
24 end start
```

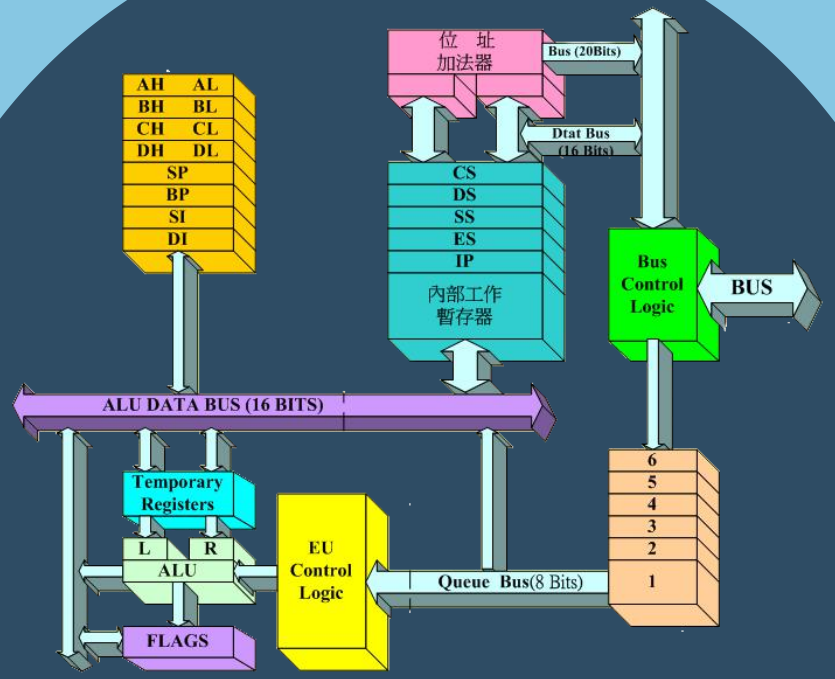
```
C:\>debug p6-3.exe
-r
AX=FFFF  BX=0000  CX=0058  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0030   NV UP EI PL NZ NA PO NC
076A:0030 8CC8          MOV     AX,CS
```

```
-u cs:0030
076A:0030 8CC8          MOV     AX,CS
076A:0032 8ED0          MOV     SS,AX
076A:0034 BC3000        MOV     SP,0030
076A:0037 BB0000        MOV     BX,0000
076A:003A B90800        MOV     CX,0008
076A:003D 2E            CS:
076A:003E FF37          PUSH    [BX]
076A:0040 83C302        ADD     BX,+02
076A:0043 E2F8          LOOP    003D
076A:0045 BB0000        MOV     BX,0000
076A:0048 B90800        MOV     CX,0008
076A:004B 2E            CS:
076A:004C 8F07          POP     [BX]
076A:004E 83C302        ADD     BX,+02
-
```

```
-d CS:0 2f
076A:0000  23 01 56 04 89 07 BC 0A-EF 0D ED 0F BA 0C 87 09   #.V.............
076A:0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
076A:0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-g

Program terminated normally
-d cs:0 2f
076A:0000  87 09 BA 0C ED 0F EF 0D-BC 0A 89 07 56 04 23 01   ............V.#.
076A:0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
076A:0020  87 09 BA 0C ED 0F EF 0D-BC 0A 58 00 6A 07 12 72   ..........X.j..r
-
```

将数据、代码、栈放入不同段

贺利坚　主讲



汇编语言程序设计

Assembly Language

评价这种方案

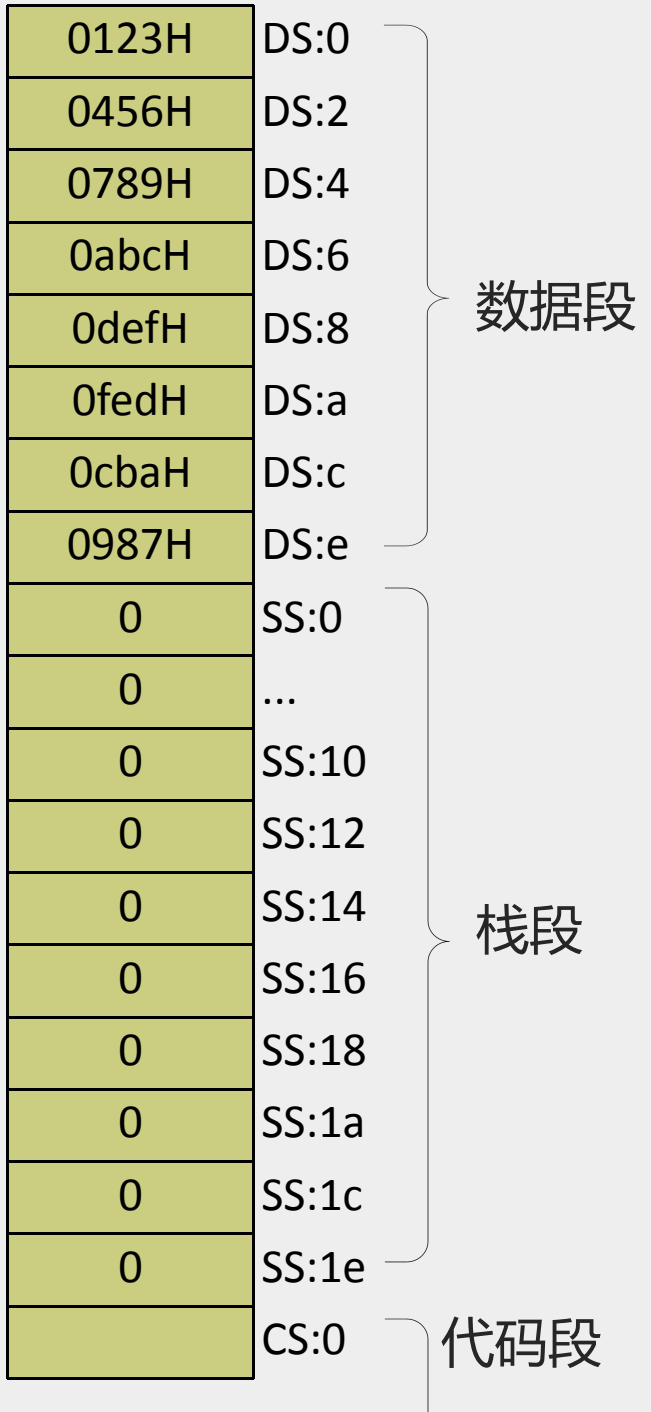
```
assume cs:codesg          ;用栈将数据逆序存放
codesg segment
        dw 0123H,0456H,0789H,0abcH,0defH,0fedH,0cbaH,0987H
        dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
start:  mov ax,cs
        mov ss,ax
        mov sp,30h
        ;入栈
        mov bx,0
        mov cx,8
    s:  push cs:[bx]
        add bx,2
        loop s
        ;出栈
        mov bx,0
        mov cx,8
    s0: pop cs:[bx]
        add bx,2
        loop s0
        mov ax,4c00h
        int 21h
codesg ends
end start
```

0123H	CS:0
0456H	CS:2
0789H	CS:4
0abcH	CS:6
0defH	CS:8
0fedH	CS:a
0cbaH	CS:c
0987H	CS:e
0	CS:10
0	...
0	CS:20
0	CS:22
0	CS:24
0	CS:26
0	CS:28
0	CS:2a
0	CS:2c
0	CS:2e
	CS:30

- 特点：数据、栈和代码都在一个段。
- 问题
  - 程序显得混乱，编程和阅读时都要注意何处是数据，何处是栈，何处是代码。
  - 只应用于要处理的数据很少，用到的栈空间也小，加上没有多长的代码。
- 对策：数据、栈和代码放在不同段。

将数据、代码、栈放入不同段

```
assume cs:code,ds:data,ss:stack
data segment
  dw 0123H,0456H,0789H,0abcH,0defH,0fedH,0cbaH,0987H
data ends
stack segment
  dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
stack ends
code segment
start:
    mov ax,stack
    mov ss,ax
    mov sp,20h
    mov ax,data
    mov ds,ax
    ; 初始化各段寄存器
    ;入栈
    mov bx,0
    mov cx,8
s:  push [bx]
    add bx,2
    loop s
    ;出栈
    mov bx,0
    mov cx,8
s0: pop [bx]
    add bx,2
    loop s0
    mov ax,4c00h
    int 21h
code ends
end start
```



在Debug中执行

```
1  assume cs:code,ds:data,ss:stack
2  data segment
3    dw 0123H,0456H,0789H,0abcH,0defH,0fedH,0cbaH,0987H
4  data ends
5  stack segment
6    dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
7  stack ends
8  code segment
9  start:mov ax,stack
10       mov ss,ax
11       mov sp,20h
12       mov ax,data
13       mov ds,ax
14
15       mov bx,0
16       mov cx,8
17    s:push [bx]
18       add bx,2
19       loop s
20
21       mov bx,0
22       mov cx,8
23   s0:pop [bx]
24       add bx,2
25       loop s0
26
27       mov ax,4c00h
28       int 21h
29 code ends
30 end start
```

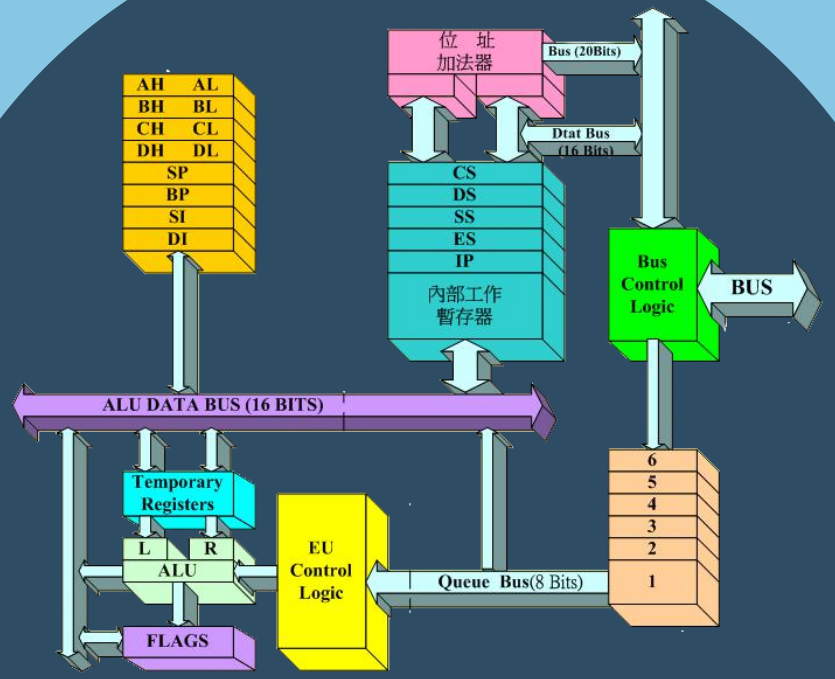
```
C:\>debug p6-4.exe
-r
AX=FFFF  BX=0000  CX=005C  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076D  IP=0000   NV UP EI PL NZ NA PO NC
076D:0000 B86B07        MOV     AX,076B
-
```

```
-u
076D:0000 B86B07        MOV     AX,076B
076D:0003 8ED0          MOV     SS,AX
076D:0005 BC2000        MOV     SP,0020
076D:0008 B86A07        MOV     AX,076A
076D:000B 8ED8          MOV     DS,AX
076D:000D BB0000        MOV     BX,0000
076D:0010 B90800        MOV     CX,0008
076D:0013 FF37          PUSH    [BX]
076D:0015 83C302        ADD     BX,+02
076D:0018 E2F9          LOOP    0013
076D:001A BB0000        MOV     BX,0000
076D:001D B90800        MOV     CX,0008
```

```
-d 076a:0 2f
076A:0000  23 01 56 04 89 07 BC 0A-EF 0D ED 0F BA 0C 87 09   #.V.............
076A:0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
076A:0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-g

Program terminated normally
-d 076a:0 2f
076A:0000  87 09 BA 0C ED 0F EF 0D-BC 0A 89 07 56 04 23 01   ............V.#.
076A:0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
076A:0020  87 09 BA 0C ED 0F EF 0D-BC 0A 2C 00 6D 07 12 72   ..........,.m..r
-
```

寄存器

贺利坚　主讲



汇编语言程序设计

Assembly Language

汇编语言程序设计课程内容

1. 绪论
2. 访问寄存器和内存
3. 汇编语言程序
4. 内存寻址方式
5. 流程转移与子程序
6. 中断及其应用
7. 高级汇编语言技术

0701 处理字符问题
0702 [bx+idata]方式寻址
0703 SI和DI寄存器
0704 [bx+si]和[bx+di]方式寻址
0705 [bx+si+idata]和[bx+di+idata]方式寻址
0706 不同的寻址方式的灵活应用
0707 不同寻址方式演示

0801 用于内存寻址的寄存器
0802 在哪里？有多长？
0803 寻址方式的综合应用
0804 用div指令实现除法
0805用dup设置内存空间

各节与教材章节的对应关系

视频（共12个）	教材对应章节
0701 处理字符问题	7.1-7.4
0702 [bx+idata]方式寻址	7.5-7.6
0703 SI和DI寄存器	7.7
0704 [bx+si]和[bx+di]方式寻址	7.8
0705 [bx+si+idata]和[bx+di+idata]方式寻址	7.9
0706 不同的寻址方式的灵活应用	7.10
0707 不同寻址方式演示	补充
0801 用于内存寻址的寄存器	8.1
0802 在哪里？有多长？	8.2-8.5
0803 寻址方式的综合应用	8.6
0804 用div指令实现除法	8.7
0805用dup设置内存空间	8.8-8.9

视频

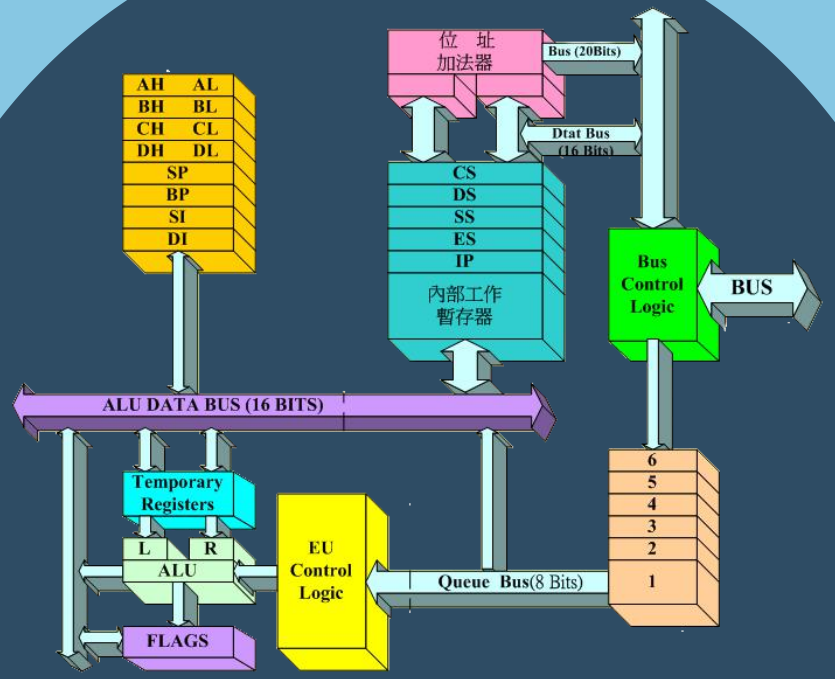
教材

检测

实验

处理字符问题

贺利坚　主讲



汇编语言程序设计

Assembly Language

处理字符问题

汇编程序中，用'……'的方式指明数据是以字符的形式给出的，编译器将把它们转化为相对应的ASCII码。

```
1  assume cs:code, ds:data
2  data segment
3          db 'unIX'
4          db 'foRK'
5  data ends
6  code segment
7  start: mov al,'a'
8          mov bl,'b'
9          mov ax,4c00h
10         int 21h
11 code ends
12 end start
```

```
-u
076B:0000 B061          MOV     AL,61
076B:0002 B362          MOV     BL,62
076B:0004 B8004C        MOV     AX,4C00
076B:0007 CD21          INT     21
```

```
-r
AX=FFFF  BX=0000  CX=0019  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076B  IP=0000   NV UP EI PL NZ NA PO NC
076B:0000 B061          MOV     AL,61
```

DS:0，即075A0H是程序开始的地址，隔过100H程序段前缀……

```
-d 076a:0 f
076A:0000  75 6E 49 58 66 6F 52 4B-00 00 00 00 00 00 00 00   unIXfoRK........
```

大写	二进制	小写	二进制
A	01000001	a	01100001
B	01000010	b	01100010
C	01000011	c	01100011
D	01000100	d	01100100

小写字母的ASCII码值比大写字母的ASCII码值大20H。

大写+20H-->小写　　小写-20H-->大写

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂

大小写转换的问题

- 问题：对datasg中的字符串
  - 第一个字符串：小写字母转换为大写字母
  - 第二个字符串：大写字母转换为小写字母

```
b   62H   0110 0010B
B   42H   0100 0010B
```

```
I   49H   0100 1001B
i   69H   0110 1001B
```

```
    0110 0010 (b)
and 1101 1111
--------------
 =  0100 0010 (B)
```

逻辑与指令：and dest, src

```
    0100 1001 (I)
 or 0010 0000
--------------
  = 0110 1001 (i)
```

逻辑或指令：or dest, src

```
assume cs:codesg,ds:datasg
datasg segment
      db 'BaSiC'
      db 'iNfOrMaTiOn'
datasg ends
codesg segment
          ...
codesg ends
end start
```

对第一个字符串，BASIC
若字母是小写，转大写；
否则，不变

对第二个字符串，information
若字母是大写，转小写；
否则，不变

程序：解决大小写转换的问题

```
assume cs:codesg,ds:datasg
datasg segment
      db 'BaSiC'
      db 'iNfOrMaTiOn'
datasg ends

codesg segment
start:
      mov ax,datasg
      mov ds,ax
      ; 第一个字符串：小写字母转换为大写字母
      ; 第二个字符串：大写字母转换为小写字母
      mov ax,4c00h
      int 21h
codesg ends
end start
```

```
      mov bx,0
      mov cx,5
 s: mov al,[bx]
      and al,11011111b
      mov [bx],al
      inc bx
      loop s
```

```
      mov bx,5
      mov cx,11
s0: mov al,[bx]
      or al,00100000b
      mov [bx],al
      inc bx
      loop s0
```

```
1  assume cs:codesg,ds:datasg
2  datasg segment
3          db 'BaSiC'
4          db 'iNfOrMaTiOn'
5  datasg ends
6
7  codesg segment
8  start:
9          mov ax,datasg
10         mov ds,ax
11
12         mov bx,0
13         mov cx,5
14     s:  mov al,[bx]
15         and al,11011111b
16         mov [bx],al
17         inc bx
18         loop s
19
20         mov bx,5
21         mov cx,11
22     s0: mov al,[bx]
23         or al,00100000b
24         mov [bx],al
25         inc bx
26         loop s0
27
28         mov ax,4c00h
29         int 21h
30 codesg ends
31 end start
```

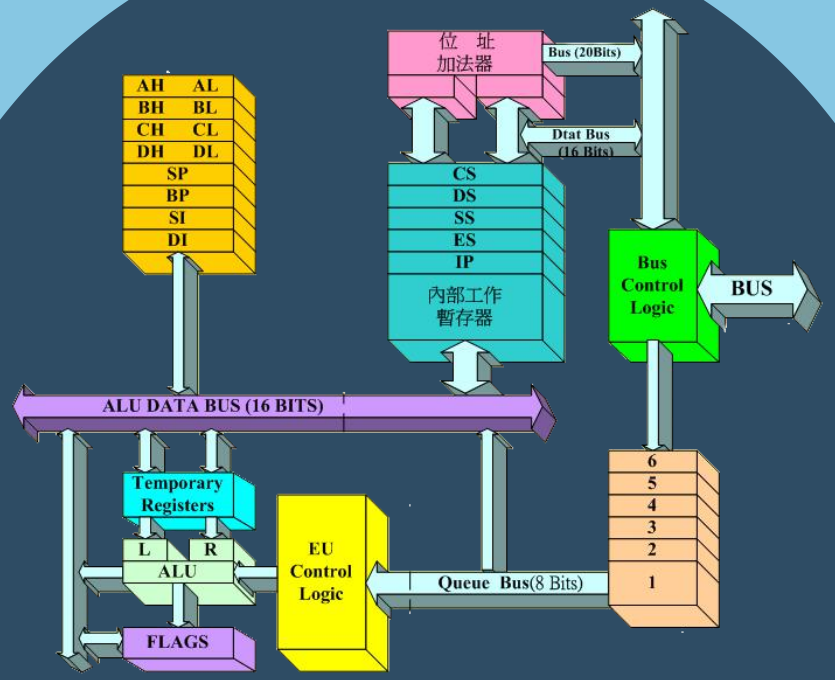
在Debug中执行程序

```
1  assume cs:codesg,ds:datasg
2  datasg segment
3          db 'BaSiC'
4          db 'iNfOrMaTiOn'
5  datasg ends
6
7  codesg segment
8  start:
9          mov ax,datasg
10         mov ds,ax
11
12         mov bx,0
13         mov cx,5
14     s:  mov al,[bx]
15         and al,11011111b
16         mov [bx],al
17         inc bx
18         loop s
19
20         mov bx,5
21         mov cx,11
22     s0: mov al,[bx]
23         or al,00100000b
24         mov [bx],al
25         inc bx
26         loop s0
27
28         mov ax,4c00h
29         int 21h
30 codesg ends
31 end start
```

```
AX=FFFF  BX=0000  CX=0038  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076B  IP=0000   NV UP EI PL NZ NA PO NC
076B:0000 B86A07        MOV     AX,076A
-d 076a:0
076A:0000  42 61 53 69 43 69 4E 66-4F 72 4D 61 54 69 4F 6E   BaSiCiNfOrMaTiOn
076A:0010  B8 6A 07 8E D8 BB 00 00-B9 05 00 8A 07 24 DF 88   .j...........$..
076A:0020  07 43 E2 F7 BB 05 00 B9-0B 00 8A 07 0C 20 88 07   .C........... ..
076A:0030  43 E2 F7 B8 00 4C CD 21-8B 46 FC 8B 56 FE 05 0C   C....L.!.F..V...
076A:0040  00 52 50 E8 EA 48 83 C4-04 50 E8 7B 0E 83 C4 04   .RP..H...P.{....
076A:0050  3D FF FF 74 03 E9 ED 00-C4 5E FC 26 8A 47 0C 2A   =..t.....^.&.G.*
076A:0060  E4 40 50 8B C3 8C C2 05-0C 00 52 50 E8 C1 48 83   .@P.......RP..H.
076A:0070  C4 04 50 8D 86 FA FE 50-E8 17 73 83 C4 06 8B B6   ..P....P..s.....
-g

Program terminated normally
-d 076a:0
076A:0000  42 41 53 49 43 69 6E 66-6F 72 6D 61 74 69 6F 6E   BASICinformation
076A:0010  B8 6A 07 8E D8 BB 00 00-B9 05 00 8A 07 24 DF 88   .j...........$..
076A:0020  07 43 E2 F7 BB 05 00 B9-0B 00 8A 07 0C 20 88 07   .C........... ..
076A:0030  43 E2 F7 B8 00 4C CD 21-8B 46 FC 8B 56 FE 05 0C   C....L.!.F..V...
076A:0040  00 52 50 E8 EA 48 83 C4-04 50 E8 7B 0E 83 C4 04   .RP..H...P.{....
076A:0050  3D FF FF 74 03 E9 ED 00-C4 5E FC 26 8A 47 0C 2A   =..t.....^.&.G.*
076A:0060  E4 40 50 8B C3 8C C2 05-0C 00 52 50 E8 C1 48 83   .@P.......RP..H.
076A:0070  C4 04 50 8D 86 FA FE 50-E8 17 73 83 C4 06 8B B6   ..P....P..s.....
-
```

[bx+idata]方式寻址

贺利坚　主讲

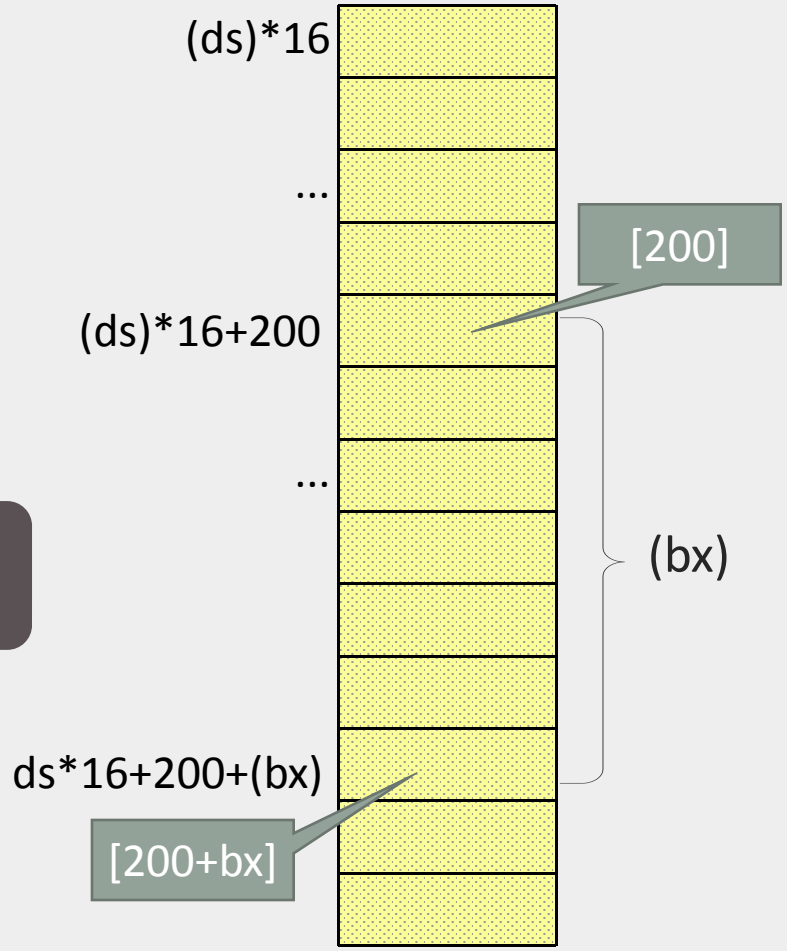


汇编语言程序设计

Assembly Language

[bx+idata]的含义

- [bx+idata]表示一个内存单元，它的偏移地址为(bx)+idata（bx中的数值加上idata）。
- mov ax,[bx+200] / mov ax, [200+bx] 的含义
  - 将一个内存单元的内容送入ax
  - 这个内存单元的长度为2字节（字单元），存放一个字
  - 内存单元的段地址在ds中，偏移地址为200加上bx中的数值
  - 数学化的描述为：(ax)=((ds)*16+200+(bx))
- 指令mov ax,[bx+200]的其他写法（常用）
  - mov ax,[200+bx]
  - mov ax,200[bx]
  - mov ax,[bx].200



示例

```
C:\>debug
-a
073F:0100 mov ax, 2000
073F:0103 mov ds, ax
073F:0105 mov bx, 1000
073F:0108 mov ax, [bx]
073F:010A mov cx, [bx+1]
073F:010D add cx, [bx+2]
073F:0110
-e 2000:1000 BE 00 06 00 00 00
```

```
-t

AX=2000  BX=1000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=2000  ES=073F  SS=073F  CS=073F  IP=0108   NV UP EI PL NZ NA PO NC
073F:0108 8B07          MOV     AX,[BX]                         DS:1000=00BE
-t

AX=00BE  BX=1000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=2000  ES=073F  SS=073F  CS=073F  IP=010A   NV UP EI PL NZ NA PO NC
073F:010A 8B4F01        MOV     CX,[BX+01]                      DS:1001=0600
-t

AX=00BE  BX=1000  CX=0600  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=2000  ES=073F  SS=073F  CS=073F  IP=010D   NV UP EI PL NZ NA PO NC
073F:010D 034F02        ADD     CX,[BX+02]                      DS:1002=0006
-t

AX=00BE  BX=1000  CX=0606  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=2000  ES=073F  SS=073F  CS=073F  IP=0110   NV UP EI PL NZ NA PE NC
073F:0110 00F0          ADD     AL,DH
```

应用：用[bx+idata]的方式进行数组的处理

- 问题：在codesg中填写代码，将datasg中定义的
  - 第一个字符串，转化为大写
  - 第二个字符串转化为小写。

```
assume cs:codesg,ds:datasg
datasg segment
    db 'BaSiC'
    db 'MinIX'
datasg ends
codesg segment
  start:
    ……
codesg ends
end start
```

是否可以对两个同长度的字符串“同步”操作？

```
    mov ax,datasg
    mov ds,ax
    mov bx,0
    mov cx,5
 s: mov al,[bx]
    and al,11011111b
    mov [bx],al
    inc bx
    loop s
    mov bx,5
    mov cx,5
 s0: mov al,[bx]
    or al,00100000b
    mov [bx],al
    inc bx
    loop s0
```

```
mov ax,datasg
    mov ds,ax
    mov bx,0
    mov cx,5
s: mov al,[bx]
    and al,11011111b
    mov [bx],al
    mov al,[5+bx]
    or al,00100000b
    mov [5+bx],al
    inc bx
    loop s
```

在Debug中执行

```
1  assume cs:codesg,ds:datasg
2  datasg segment
3          db 'BaSiC'
4          db 'MinIX'
5  datasg ends
6  codesg segment
7  start: mov ax,datasg
8          mov ds,ax
9
10         mov bx,0
11         mov cx,5
12 s:      mov al,[bx]
13         and al,11011111b
14         mov [bx],al
15
16         mov al,[5+bx]
17         or al,00100000b
18         mov [5+bx],al
19         inc bx
20         loop s
21
22         mov ax, 4c00h
23         int 21h
24 codesg ends
25 end start
```

```
C:\>debug p7-3.exe
-r
AX=FFFF  BX=0000  CX=0031  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076B  IP=0000   NV UP EI PL NZ NA PO NC
076B:0000 B86A07        MOV     AX,076A
-d 076a:0 f
076A:0000  42 61 53 69 43 4D 69 6E-49 58 00 00 00 00 00 00   BaSiCMinIX......
-g

Program terminated normally
-d 076a:0 f
076A:0000  42 41 53 49 43 6D 69 6E-69 78 00 00 00 00 00 00   BASICminix......
-
```

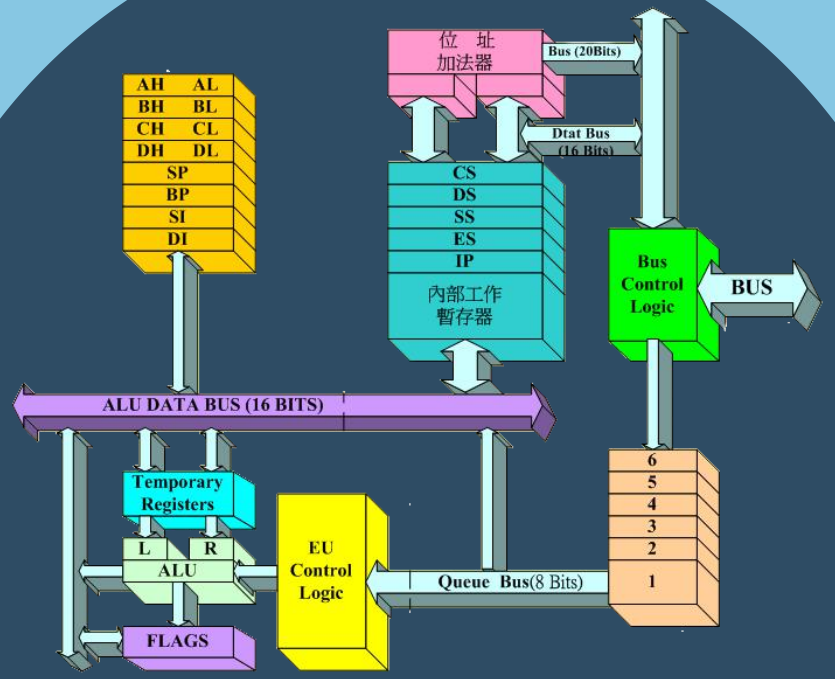
```
char a[5]="BaSiC";
char b[5]=""MinIX";
main(){
   int i;
   i=0;
   do{
      a[i]=a[i]&0xDF;
      b[i]=b[i]|0x20;
      i++;
   }
   while(i<5);
}
```

味道有点像！

[bx+idata]的方式为高级语言实现数组提供了便利机制。

SI和DI寄存器

贺利坚　主讲

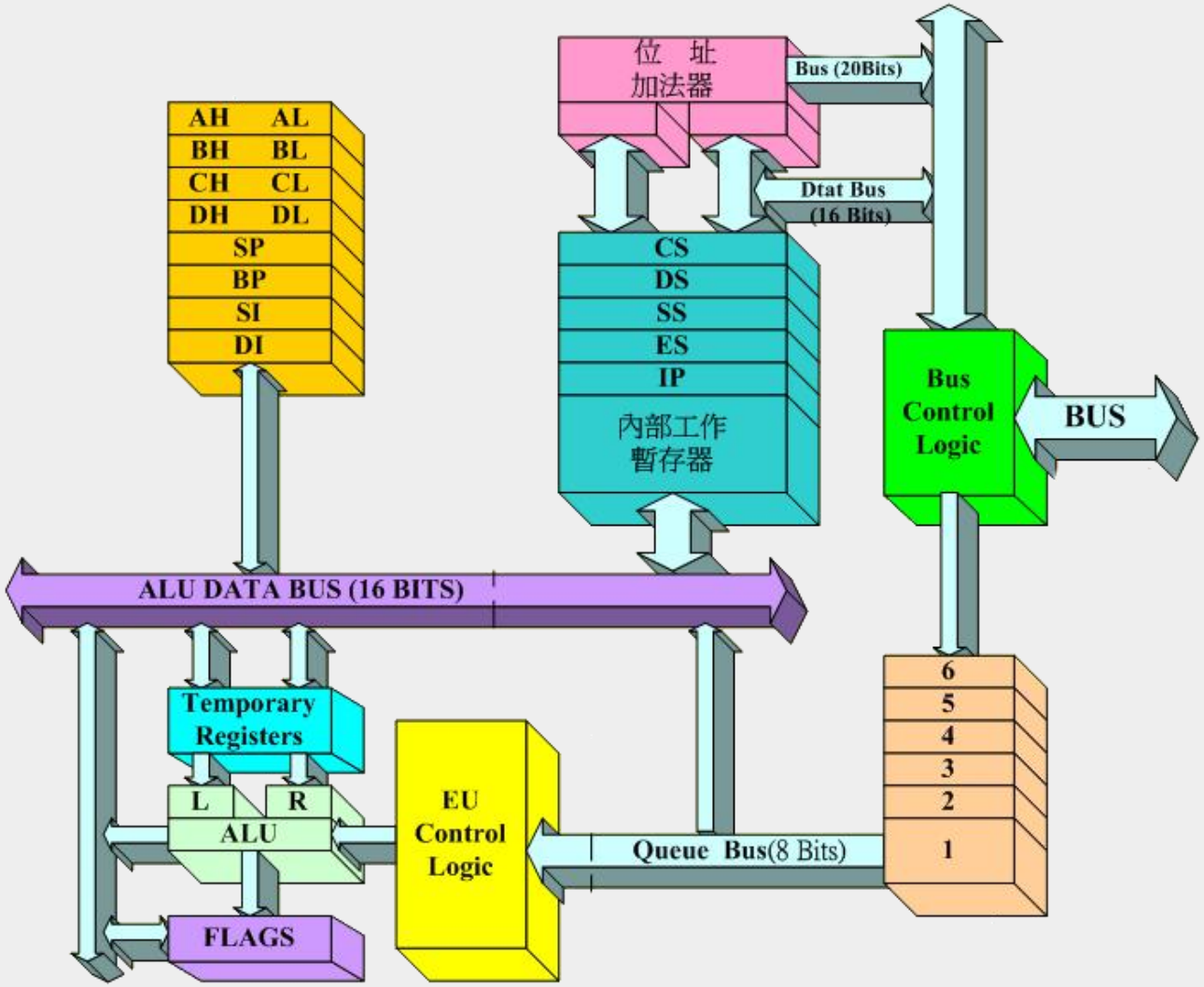


汇编语言程序设计

Assembly Language

CPU内部的寄存器

- 8086CPU有14个寄存器：
  - 通用寄存器：AX、 BX、 CX、 DX
  - 变址寄存器：SI、 DI
  - 指针寄存器：SP、 BP
  - 指令指针寄存器： IP
  - 段寄存器：CS、 SS、 DS、 ES
  - 标志寄存器：PSW



SI和DI常执行与地址有关的操作

- SI和DI是8086CPU中和BX功能相近的寄存器
  - 区别：SI和DI不能够分成两个8 位寄存器来使用。
- 下面的三组指令实现了相同的功能：

（1）mov bx,0
　　mov ax,[bx]

（2）mov si,0
　　mov ax,[si]

（3）mov di,0
　　mov ax,[di]

（1）mov bx,0
　　mov ax,[bx+123]

（2）mov si,0
　　mov ax,[si+123]

（3）mov di,0
　　mov ax,[di+123]

- BX：通用寄存器，在计算存储器地址时，常作为基址寄存器用
- SI：source index，源变址寄存器
- DI：destination index，目标变址寄存器

应用SI和DI

- 问题
  - 用寄存器SI和DI实现将字符串 ‘welcome to masm!’ 复制到它后面的数据区中。
- 程序定义

```
assume cs:codesg,ds:datasg
datasg segment
    db 'welcome to masm!'
    db '................'
datasg ends
codesg segment
    ....
codesg ends
end
```

- 源数据起始地址：datasg:0
- 目标数据起始地址：datasg:16

- 用ds:si 指向要复制的原始字符串
- 用 ds:di 指向目的空间
- 然后用一个循环来完成复制。

```
1  assume cs:codesg,ds:datasg
2  datasg segment
3        db 'welcome to masm!'
4        db '................'
5  datasg ends
6  codesg segment
7  start: mov ax,datasg
8        mov ds,ax
9
10       mov si,0
11       mov di,16
12       mov cx,8
13    s: mov ax,[si]
14       mov [di],ax
15       add si,2
16       add di,2
17       loop s
18
19       mov ax,4c00h
20       int 21h
21 codesg ends
22 end start
```

程序运行

```
1  assume cs:codesg,ds:datasg
2  datasg segment
3        db 'welcome to masm!'
4        db '................'
5  datasg ends
6  codesg segment
7  start: mov ax,datasg
8        mov ds,ax
9
10       mov si,0
11       mov di,16
12       mov cx,8
13    s: mov ax,[si]
14       mov [di],ax
15       add si,2
16       add di,2
17       loop s
18
19       mov ax,4c00h
20       int 21h
21 codesg ends
22 end start
```

```
-d 076A:0 1f
076A:0000  77 65 6C 63 6F 6D 65 20-74 6F 20 6D 61 73 6D 21   welcome to masm!
076A:0010  2E 2E 2E 2E 2E 2E 2E 2E-2E 2E 2E 2E 2E 2E 2E 2E   ................
-u
076C:0000 B86A07        MOV     AX,076A
076C:0003 8ED8          MOV     DS,AX
076C:0005 BE0000        MOV     SI,0000
076C:0008 BF1000        MOV     DI,0010
076C:000B B90800        MOV     CX,0008
076C:000E 8B04          MOV     AX,[SI]
076C:0010 8905          MOV     [DI],AX
076C:0012 83C602        ADD     SI,+02
076C:0015 83C702        ADD     DI,+02
076C:0018 E2F4          LOOP    000E
076C:001A B8004C        MOV     AX,4C00
076C:001D CD21          INT     21
076C:001F 0C00          OR      AL,00
-g

Program terminated normally
-d 076A:0 1f
076A:0000  77 65 6C 63 6F 6D 65 20-74 6F 20 6D 61 73 6D 21   welcome to masm!
076A:0010  77 65 6C 63 6F 6D 65 20-74 6F 20 6D 61 73 6D 21   welcome to masm!
```

程序还可以写作——

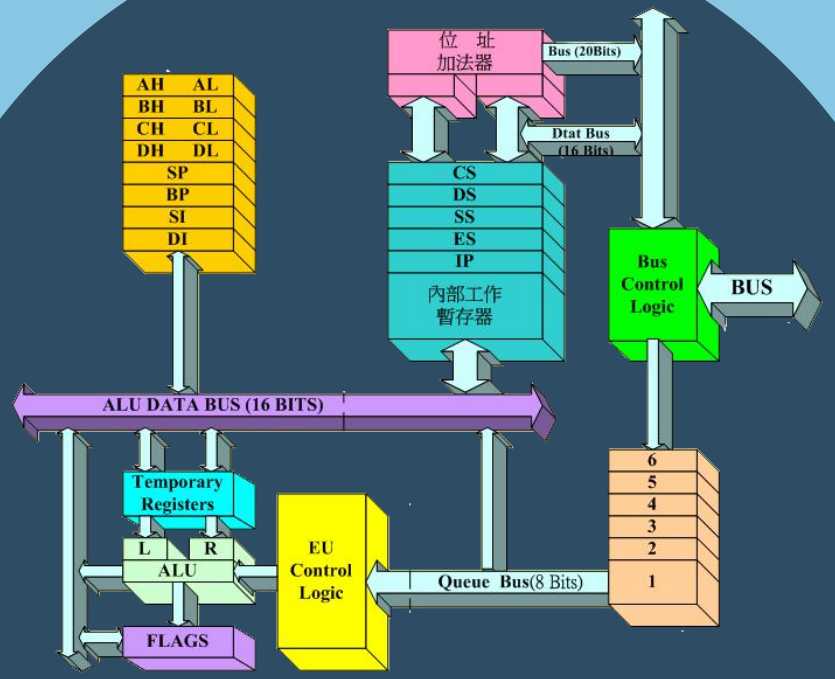
```
assume cs:codesg,ds:datasg
datasg segment
        db 'welcome to masm!'
        db '................'
datasg ends
codesg segment
start: mov ax,datasg
        mov ds,ax

        mov si,0
        mov di,16
        mov cx,8
    s:  mov ax,[si]
        mov [di],ax
        add si,2
        add di,2
        loop s

        mov ax,4c00h
        int 21h
codesg ends
end start
```

```
assume cs:codesg,ds:datasg
datasg segment
        db 'welcome to masm!'
        db '................'
datasg ends
codesg segment
start: mov ax,datasg
        mov ds,ax

        mov si,0
        mov cx,8
    s:  mov ax,0[si]
        mov 16[si],ax
        add si,2
        loop s

        mov ax,4c00h
        int 21h
codesg ends
end start
```

[bx+idata]形式

[bx+si]和[bx+di]方式寻址

贺利坚　主讲

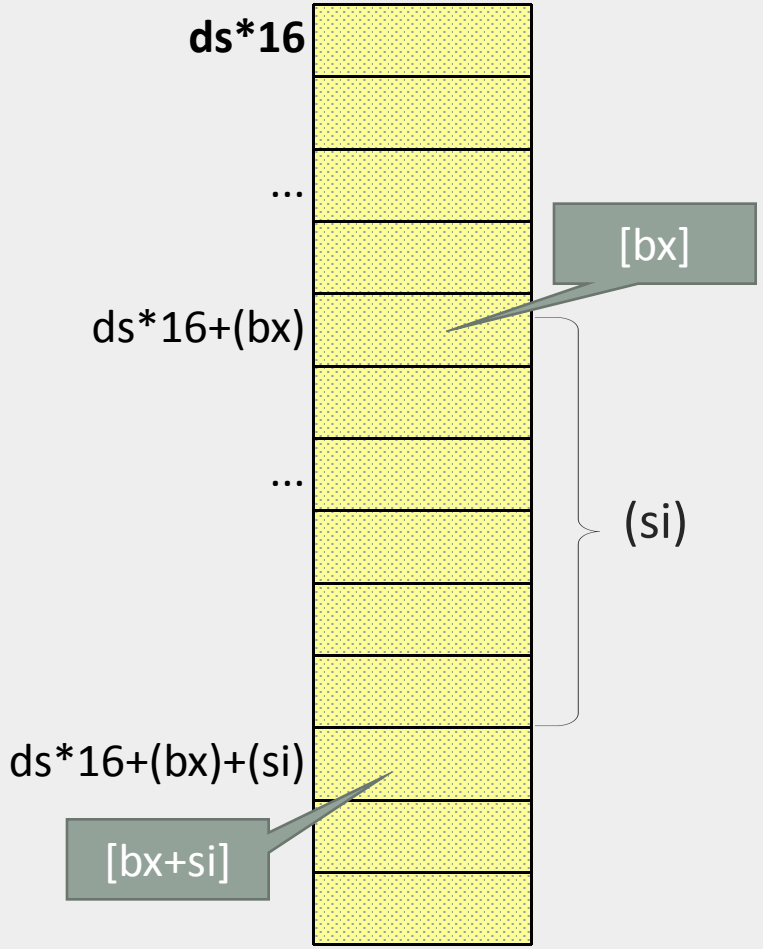


汇编语言程序设计

Assembly Language

[bx+si]和[bx+di]方式指定地址

- [bx+si]表示一个内存单元
  - 偏移地址为(bx)+(si)（即bx中的数值加上si中的数值）。
- 指令mov ax,[bx+si]的含义
  - 将一个内存单元的内容送入ax
  - 这个内存单元的长度为2字节（字单元），存放一个字
  - 偏移地址为bx中的数值加上si中的数值
  - 段地址在ds中
- 指令mov ax,[bx+si]的数学化的描述
  - (ax)=((ds)*16+(bx)+(si))
- mov ax,[bx+si]的其他写法
  - mov ax,[bx][si]



应用案例

- 内存中数据　2000:1000 BE 00 06 00 00 00 ……
- 程序执行后，ax、bx、cx中的内容？

DS		SI		DI	
AX		BX		CX	

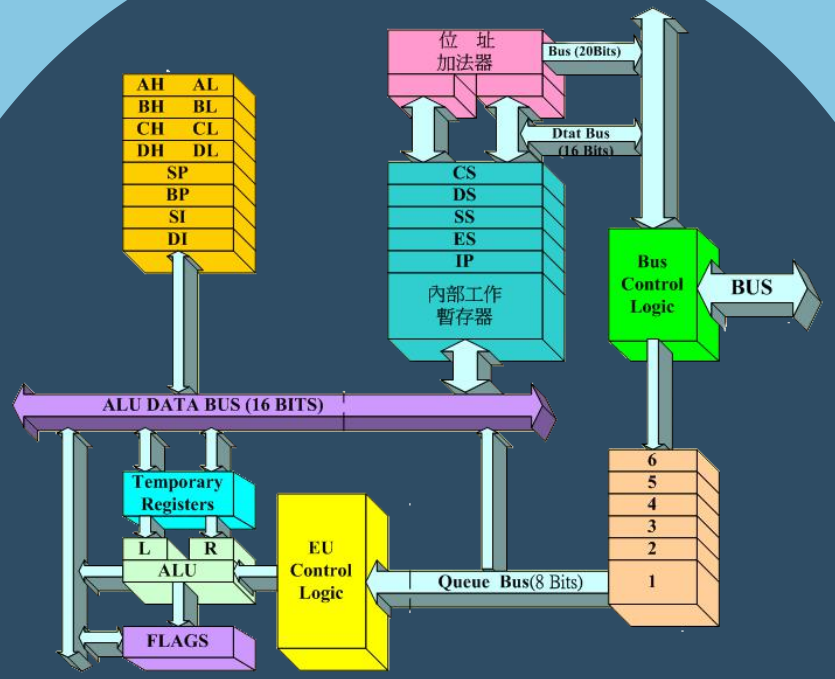
```
mov ax,2000H
mov ds,ax
mov bx,1000H
mov si,0
mov ax,[bx+si]
inc si
mov cx,[bx+si]
inc si
mov di,si
mov ax,[bx+di]
```

```
C:\>debug
-a
073F:0100 mov ax, 2000
073F:0103 mov ds, ax
073F:0105 mov bx, 1000
073F:0108 mov si, 0
073F:010B mov ax, [bx+si]
073F:010D inc si
073F:010E mov cx, [bx+si]
073F:0110 inc si
073F:0111 mov di, si
073F:0113 mov ax, [bx+di]
073F:0115
-
```

```
-e 2000:1000 BE 00 06 00 00 00
-g 0113

AX=00BE  BX=1000  CX=0600  DX=0000  SP=00FD  BP=0000  SI=0002  DI=0002
DS=2000  ES=073F  SS=073F  CS=073F  IP=0113   NV UP EI PL NZ NA PO NC
073F:0113 8B01          MOV     AX,[BX+DI]                      DS:1002=0006
-t

AX=0006  BX=1000  CX=0600  DX=0000  SP=00FD  BP=0000  SI=0002  DI=0002
DS=2000  ES=073F  SS=073F  CS=073F  IP=0115   NV UP EI PL NZ NA PO NC
073F:0115 0000          ADD     [BX+SI],AL                      DS:1002=06
-
```

[bx+si+idata]和[bx+di+idata]

贺利坚　主讲



汇编语言程序设计

Assembly Language

[bx+si+idata]和[bx+di+idata]方式指定地址

- [bx+si+idata]表示一个内存单元
  - 偏移地址为(bx)+(si)+idata，即bx中的数值加上si中的数值再加上idata
- 指令mov ax,[bx+si+idata]的含义
  - 将一个内存单元的内容送入ax
  - 这个内存单元的长度为2字节（字单元），存放一个字
  - 偏移地址为bx中的数值加上si中的数值再加上idata，段地址在ds中
- 数学化的描述
  - (ax)=((ds)*16+(bx)+(si)+idata)
- 指令mov ax,[bx+si+idata]的其他写法

mov ax,[bx+200+si]

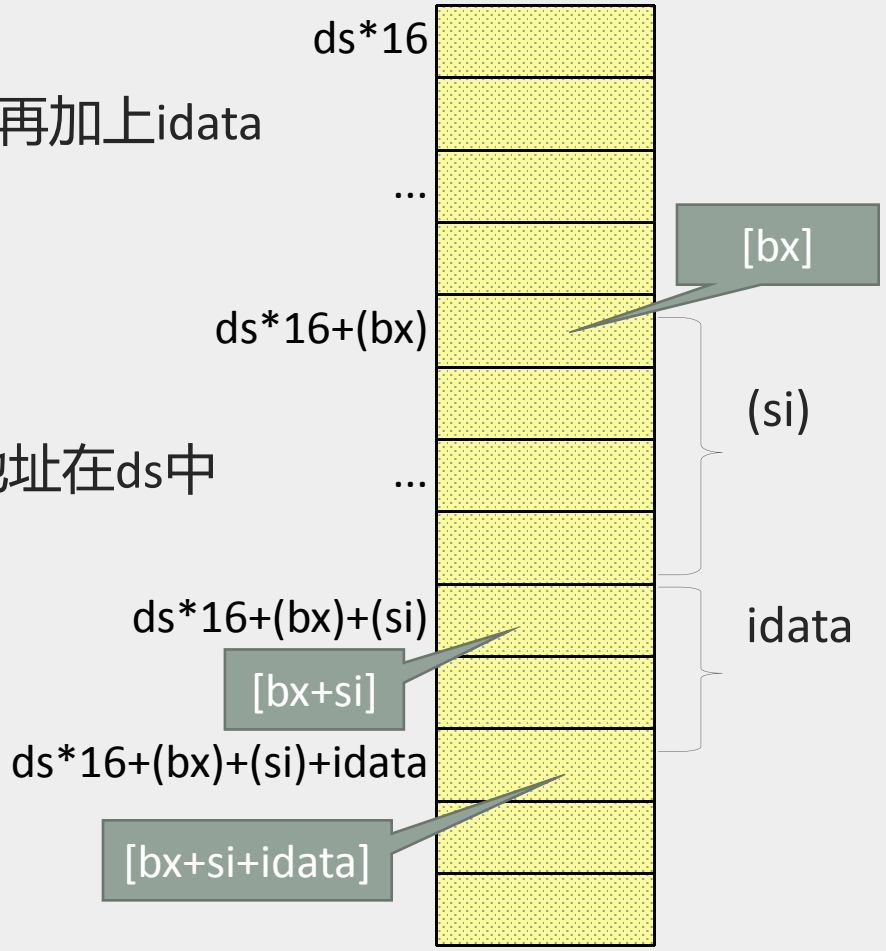
mov ax,[200+bx+si]

mov ax,200[bx][si]

mov ax,[bx].200[si]

mov ax,[bx][si].200

mov ax,[bx][si]



应用案例

- 内存中数据：2000:1000 BE 00 06 00 6A 22 ……
- 程序执行后，ax、bx、cx中的内容？

DS		SI		DI	
AX		BX		CX	

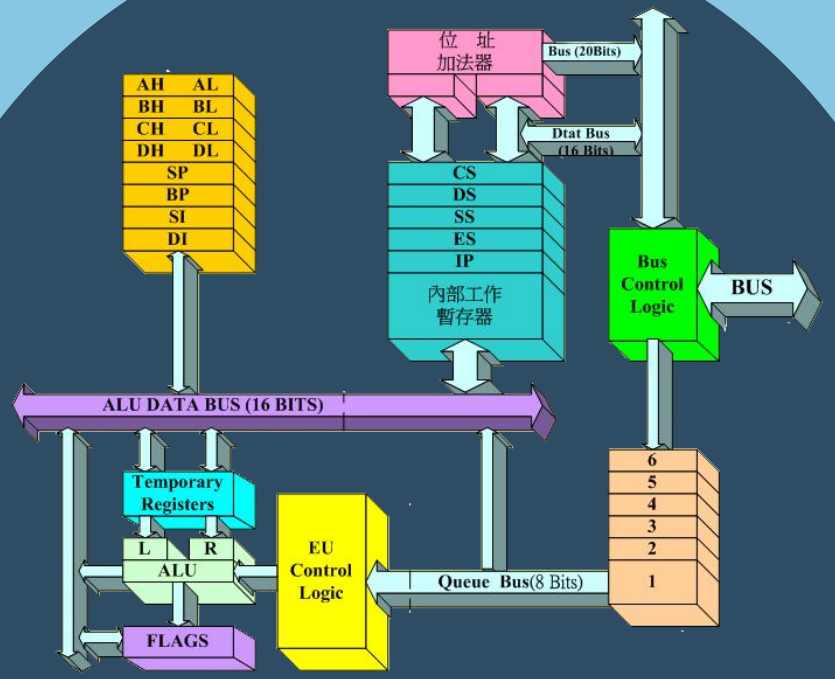
```
mov ax,2000H
mov ds,ax
mov bx,1000H
mov si,0
mov ax,[bx+2+si]
inc si
mov cx,[bx+2+si]
inc si
mov di,si
mov ax,[bx+2+di]
```

```
-a 073f:0100
073F:0100 mov ax, 2000
073F:0103 mov ds, ax
073F:0105 mov bx, 1000
073F:0108 mov si, 0
073F:010B mov ax, [bx+2+si]
073F:010E inc si
073F:010F mov cx, [bx+2+si]
073F:0112 inc si
073F:0113 mov di, si
073F:0115 mov ax, [bx+2+di]
073F:0118
-
```

```
-e 2000:1000 BE 00 06 00 6A 22
-g 010E

AX=0006  BX=1000  CX=0600  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0002
DS=2000  ES=073F  SS=073F  CS=073F  IP=010E   NV UP EI PL NZ NA PO NC
073F:010E 46            INC     SI
-g 0112

AX=0006  BX=1000  CX=6A00  DX=0000  SP=00FD  BP=0000  SI=0001  DI=0002
DS=2000  ES=073F  SS=073F  CS=073F  IP=0112   NV UP EI PL NZ NA PO NC
073F:0112 46            INC     SI
-g 0118

AX=226A  BX=1000  CX=6A00  DX=0000  SP=00FD  BP=0000  SI=0002  DI=0002
DS=2000  ES=073F  SS=073F  CS=073F  IP=0118   NV UP EI PL NZ NA PO NC
073F:0118 0000          ADD     [BX+SI],AL                  DS:1002=06
-
```

不同的寻址方式的灵活应用

贺利坚　主讲



汇编语言程序设计

Assembly Language

对内存的寻址方式

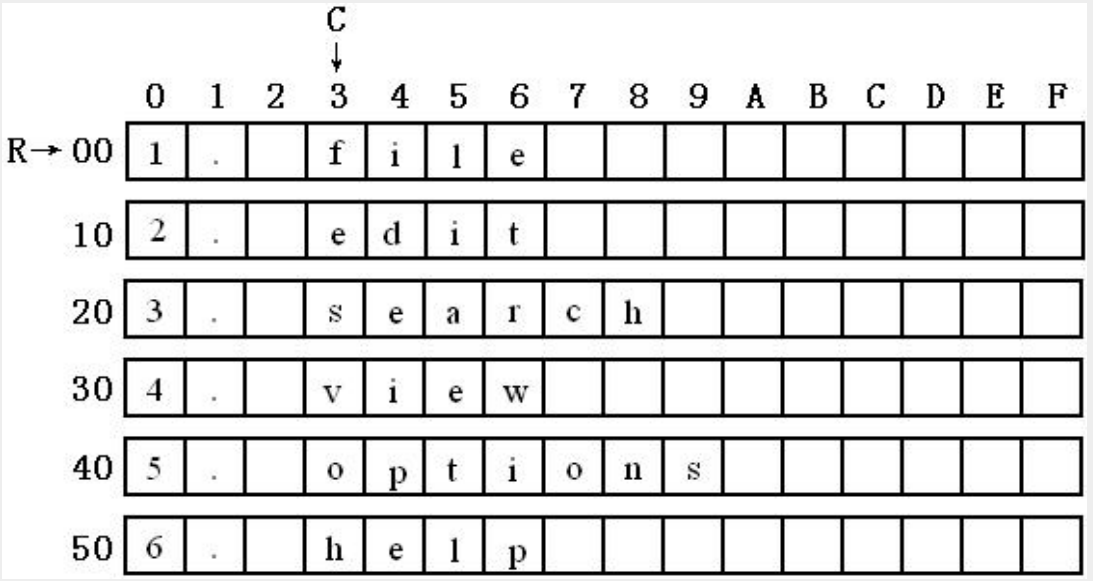
形式	名称	特点	意义	示例
[idata]	直接寻址	用一个常量/立即数来表示地址	用于直接定位一个内存单元	mov ax, [200]
[bx]	寄存器间接寻址	用一个变量来表示内存地址	用于间接定位一个内存单元	mov bx, 0 mov ax, [bx]
[bx+idata]	寄存器相对寻址	用一个变量和常量表示地址	可在一个起始地址的基础上用变量间接定位一个内存单元	mov bx, 4 mov ax, [bx+200]
[bx+si]	基址变址寻址	用两个变量表示地址		mov ax, [bx+si]
[bx+si+idata]	相对基址变址寻址	用两个变量和一个常量表示地址		mov ax, [bx+si+200]

案例1：灵活应用不同的寻址方式

💻问题：编程将datasg段中每个单词的头一个字母改为大写字母。

```
assume cs:codesg,ds:datasg
datasg segment
      db '1. file          '
      db '2. edit          '
      db '3. search        '
      db '4. view          '
      db '5. options       '
      db '6. help          '
datasg ends

codesg segment
 start: ....
     mov 4c00h
     int 21h
codesg ends
end start
```

[bx+idata]方式



datasg中的数据的存储结构

```
    R=第一行的地址
    mov cx,6
s:  改变R行3列的字母为大写
    R=下一行的地址
    loop s
```

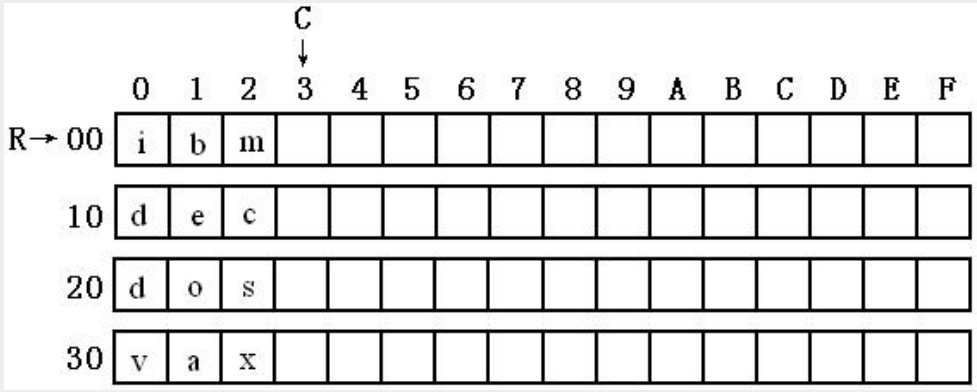
```
    mov ax,datasg
    mov ds,ax

    mov bx,0
    mov cx,6
s:  mov al,[bx+3]
    and al,11011111b
    mov [bx+3],al
    add bx,16
    loop s
```

案例2：灵活应用不同的寻址方式

- 问题：编程将datasg段中每个单词改为大写字母。

- 4 个字符串，看成一个 4行16列的二维数组
- 要修改二维数组的每一行的前3列
- 构造4x3次的二重循环

```
assume cs:codesg,ds:datasg
datasg segment
  db 'ibm             '
  db 'dec             '
  db 'dos             '
  db 'vax             '
datasg ends

codesg segment
 start: ......
codesg ends
end start
```

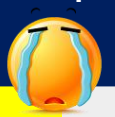


[bx+si] 方式

```
    R=第一行的地址；
    mov cx,4
s0: C=第一列的地址
    mov cx,3
 s: 改变R 行，C列字母为大写
    C=下一列的地址；
    loop s
    R=下一行的地址
    loop s0
```

```
    mov ax,datasg
    mov ds,ax
    mov bx,0
    mov cx,4
s0: mov si,0
    mov cx,3
 s: mov al,[bx+si]
    and al,11011111b
    mov [bx+si],al
    inc si
    loop s
    add bx,16
    loop s0
```

循环次数由cx定，可是，cx只有一个哇！



二重循环问题的处理-法1

问题：编程将datasg段中每个单词改为大写字母。

```
assume cs:codesg,ds:datasg
datasg segment
  db 'ibm          '
  db 'dec          '
  db 'dos          '
  db 'vax          '
datasg ends

codesg segment
 start: ......
codesg ends
end start
```

```
    ; 有缺陷的程序
    mov ax,datasg
    mov ds,ax
    mov bx,0
    mov cx,4
s0: mov si,0
    mov cx,3
 s: mov al,[bx+si]
    and al,11011111b
    mov [bx+si],al
    inc si
    loop s
    add bx,16
    loop s0
```

dx已经被用了呢？别的寄存器也有用处了呢？寄存器只有14个！

```
    mov ax,datasg
    mov ds,ax
    mov bx,0
    mov cx,4
s0: mov dx,cx
    mov si,0
    mov cx,3
 s: mov al,[bx+si]
    and al,11011111b
    mov [bx+si],al
    inc si
    loop s
    add bx,16
    mov cx,dx
    loop s0
```

将外层循环的cx值保存在dx中

cx设置为内存循环的次数

用dx中存放的外层循环的计数值恢复cx

(cx)=(cx)-1针对外层循环

方法1：用dx保存数据

二重循环问题的处理-法2、法3

方法2：用固定的内存空间保存数据

```
    mov ax,datasg
    mov ds,ax
    mov bx,0
    mov cx,4
s0: mov ds:[40H],cx
    mov si,0
    mov cx,3
 s: mov al,[bx+si]
    and al,11011111b
    mov [bx+si],al
    inc si
    loop s
    add bx,16
    mov cx,ds:[40H]
    loop s0
```

将外层循环的cx值保存在datasg:40H单元中

cx设置为内存循环的次数

用datasg:40H单元中的值恢复cx

方法3：用栈保存数据

```
stacksg segment
    dw 0,0,0,0,0,0,0,0
stacksg ends
```

```
    mov ax,stacksg
    mov ss,ax
    mov sp,16
    mov ax,datasg
    mov ds,ax
    mov bx,0
    mov cx,4
s0: push cx
    mov si,0
    mov cx,3
 s: mov al,[bx+si]
    and al,11011111b
    mov [bx+si],al
    inc si
    loop s
    add bx,16
    pop cx
    loop s0
```

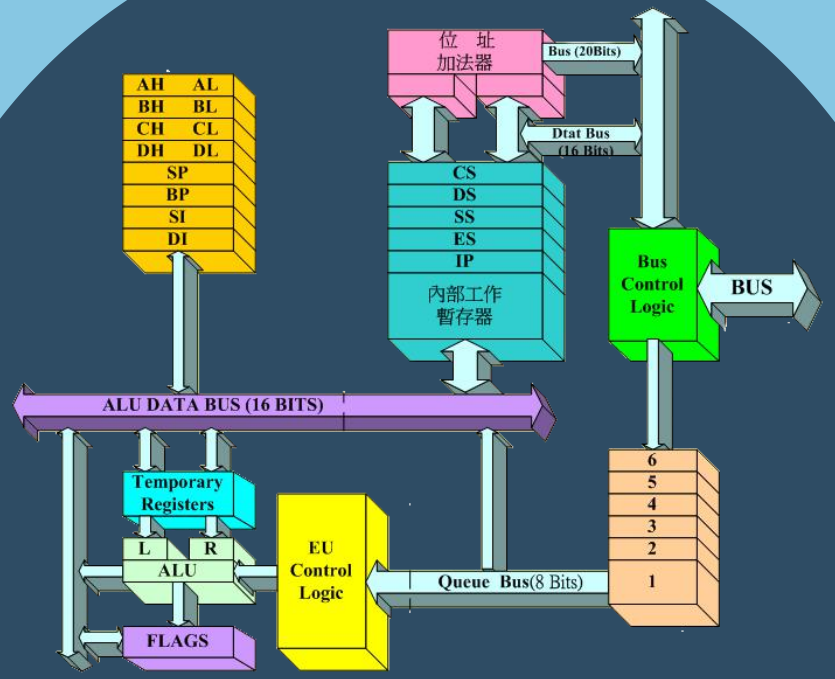
将外层循环的cx值压栈

cx设置为内存循环的次数

从栈顶弹出原cx的值，恢复cx

用于内存寻址的寄存器

贺利坚　主讲

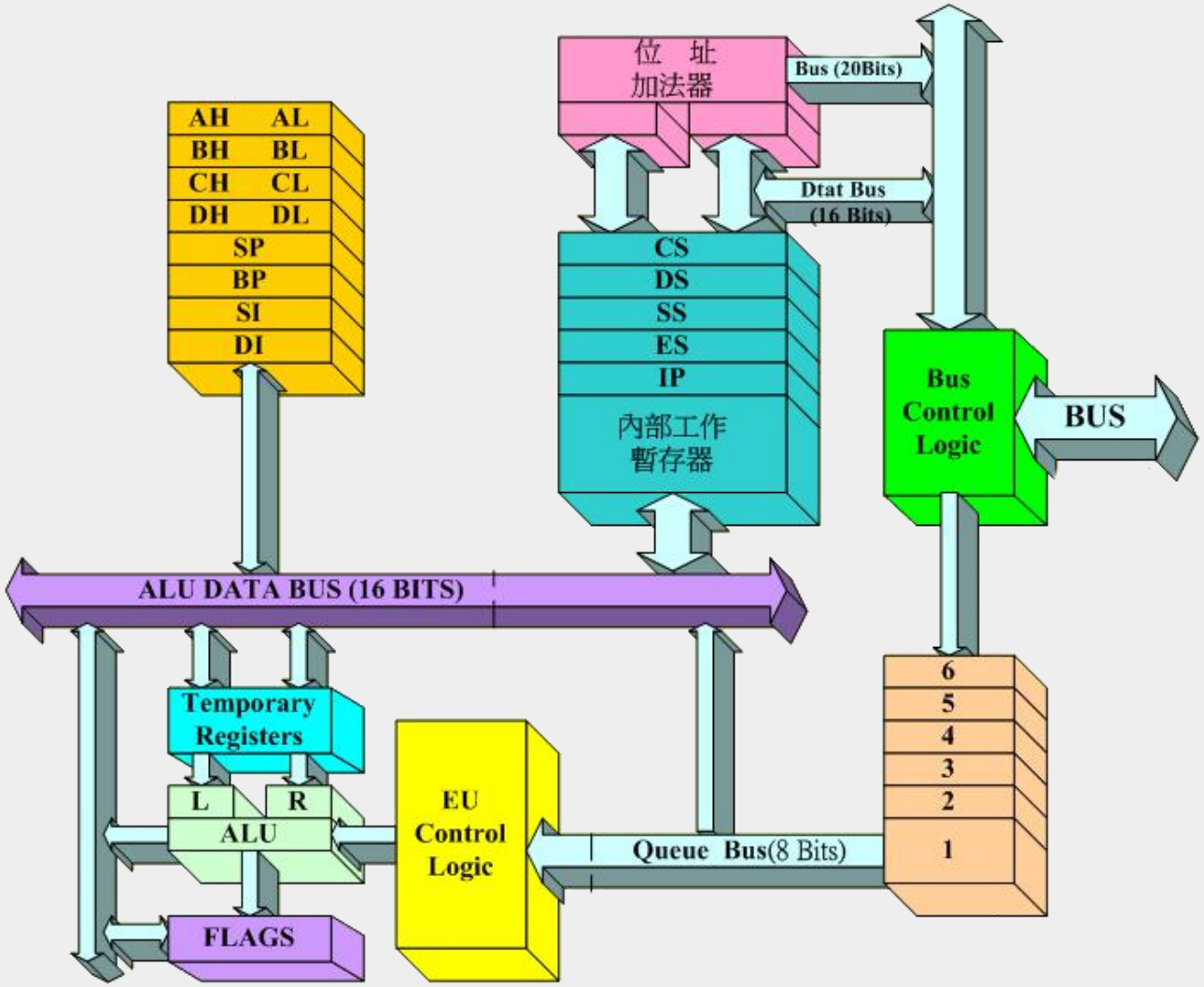


汇编语言程序设计

Assembly Language

哪些寄存器用于寻址?

- 8086CPU有14个寄存器：
  - 通用寄存器：AX、 BX、 CX、 DX
  - 变址寄存器：SI、 DI
  - 指针寄存器：SP、 BP
  - 指令指针寄存器： IP
  - 段寄存器：CS、 SS、 DS、 ES
  - 标志寄存器：PSW



用于内存寻址的寄存器用法

正确的指令

mov ax,[bx]

mov ax,[bx+si]

mov ax,[bx+di]

mov ax,[bp]

mov ax,[bp+si]

mov ax,[bp+di]

只有bx、bp、si、di可以用在[...]对内存单元寻址

正确的指令

mov ax,[bx]

mov ax,[si]

mov ax,[di]

mov ax,[bp]

mov ax,[bx+si]

mov ax,[bx+di]

mov ax,[bp+si]

mov ax,[bp+di]

mov ax,[bx+si+idata]

mov ax,[bx+di+idata]

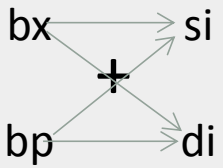
mov ax,[bp+si+idata]

mov ax,[bp+di+idata]

错误的指令

mov ax,[bx+bp]

mov ax,[si+di]



错误的指令

mov ax,[cx]

mov ax,[ax]

mov ax,[dx]

mov ax,[ds]

bx以外的通用寄存器、段寄存器不可以用在[...]中

bx、bp区别：

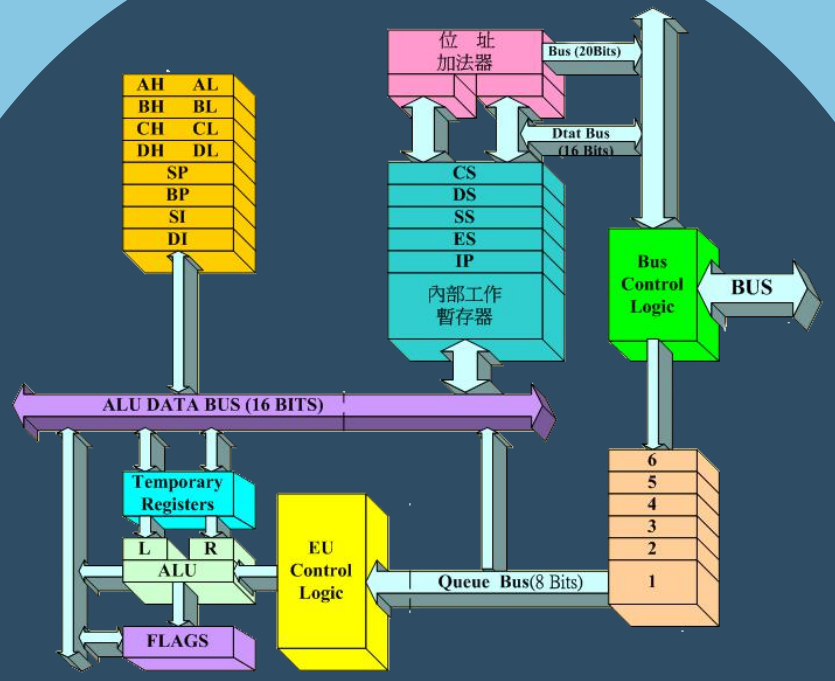
· bx默认指ds段；

· bp默认指ss段。

```
mov ax,[bp]            (ax)=((ss)*16+(bp))
mov ax,ds:[bp]         (ax)=((ds)*16+(bp))
mov ax,es:[bp]         (ax)=((es)*16+(bp))
mov ax,[bx]            (ax)=((ds)*16+(bx))
mov ax,ss:[bx]         (ax)=((ss)*16+(bx))
mov ax,[bp+idata]      (ax)=((ss)*16+(bp)+idata)
mov ax,[bp+si]         (ax)=((ss)*16+(bp)+(si))
mov ax,[bp+si+idata]   (ax)=((ss)*16+(bp)+(si)+idata)
```

在哪里？有多长？

贺利坚　主讲



汇编语言程序设计

Assembly Language

两个基本问题


```
mov ax, 0
mov ax, [0]
mov ax, [di]
mov ax, [bx+8]
mov ax, [bx+si]
mov ax, [bx+si+8]
mov ax, [bp]
mov ax, [bp+8]
mov ax, [bp+si]
mov ax, [bp+si+8]
…
```

（1）处理的数据在什么地方？

（2）要处理的数据有多长？

汇编语言中数据位置的表达

1、立即数（idata）

对于直接包含在机器指令中的数据,称为立即数（idata），数据包含在指令中

```
mov ax,1
add bx,2000h
or bx,00010000b
mov al,'a'
```

```
-a
073F:0100 mov ax, 1
073F:0103
-u 073f:0100
073F:0100 B80100    MOV     AX,0001
```

2、寄存器

指令要处理的数据在寄存器中，在汇编指令中给出相应的寄存器名。

```
mov ax,bx
mov ds,ax
push bx
mov ds:[0],bx
push ds
mov ss,ax
mov sp,ax
```

3、内存：段地址（SA）和偏移地址（EA）

指令要处理的数据在内存中，由SA:EA确定内存单元。

```
mov ax,[0]
mov ax,[di]
mov ax,[bx+8]
mov ax,[bx+si]
mov ax,[bx+si+8]
```

段地址默认在ds中

```
mov ax,[bp]
mov ax,[bp+8]
mov ax,[bp+si]
mov ax,[bp+si+8]
```

段地址默认在ss中

```
mov ax,ds:[bp]          ：(ax)=((ds)*16+(bp))
mov ax,es:[bx]          ：(ax)=((es)*16+(bx))
mov ax,ss:[bx+si]       ：(ax)=((ss)*16+(bx)+(si))
mov ax,cs:[bx+si+8]     ：(ax)=((cs)*16+(bx)+(si)+8)
```

显性的给出存放段地址的寄存器

指令要处理的数据有多长?

字word操作	字节byte操作	用word ptr或byte ptr指明

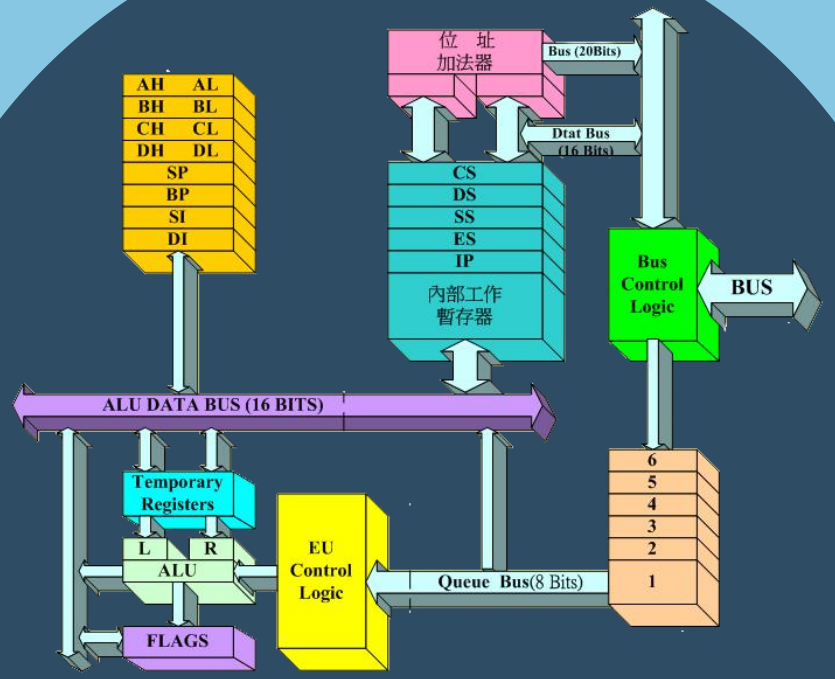
mov ax,1
mov bx,ds:[0]
mov ds,ax
mov ds:[0],ax
inc ax
add ax,1000
	mov al,1
	mov al,bl
	mov al,ds:[0]
	mov ds:[0],al
	inc al
	add al,100
		mov word ptr ds:[0],1
		inc word ptr [bx]
		inc word ptr ds:[0]
		add word ptr [bx],2
		
		mov byte ptr ds:[0],1
		inc byte ptr [bx]
		inc byte ptr ds:[0]
		add byte ptr [bx],2
		
		在没有寄存器参与的内存单元访问指令中，用word ptr或byte ptr显性地指明所要访问的内存单元的长度是很必要的，否则，CPU无法得知所要访问的单元是字单元，还是字节单元。

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=2000  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 B80020        MOV     AX,2000
-a 073f:0100
073F:0100 mov byte ptr [1000], 1
073F:0105
-e 2000:1000 FF FF FF FF FF FF FF FF
-T

AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=2000  ES=073F  SS=073F  CS=073F  IP=0105   NV UP EI PL NZ NA PO NC
073F:0105 C606001001    MOV     BYTE PTR [1000],01                 DS:10
-d 2000:1000
2000:1000  01 FF FF FF FF FF FF FF-00 00 00 00 00 00 00 00   ..........
```

```
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=2000  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 C70600100100  MOV     WORD PTR [1000],0001               DS:10
-a 073f:0100
073F:0100 mov word ptr [1000], 1
073F:0106
-e 2000:1000 FF FF FF FF FF FF FF FF
-t

AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=2000  ES=073F  SS=073F  CS=073F  IP=0106   NV UP EI PL NZ NA PO NC
073F:0106 06            PUSH    ES
-d 2000:1000
2000:1000  01 00 FF FF FF FF FF FF-00 00 00 00 00 00 00 00   ..........
```

寻址方式的综合应用

贺利坚　主讲



汇编语言程序设计

Assembly Language

应用问题

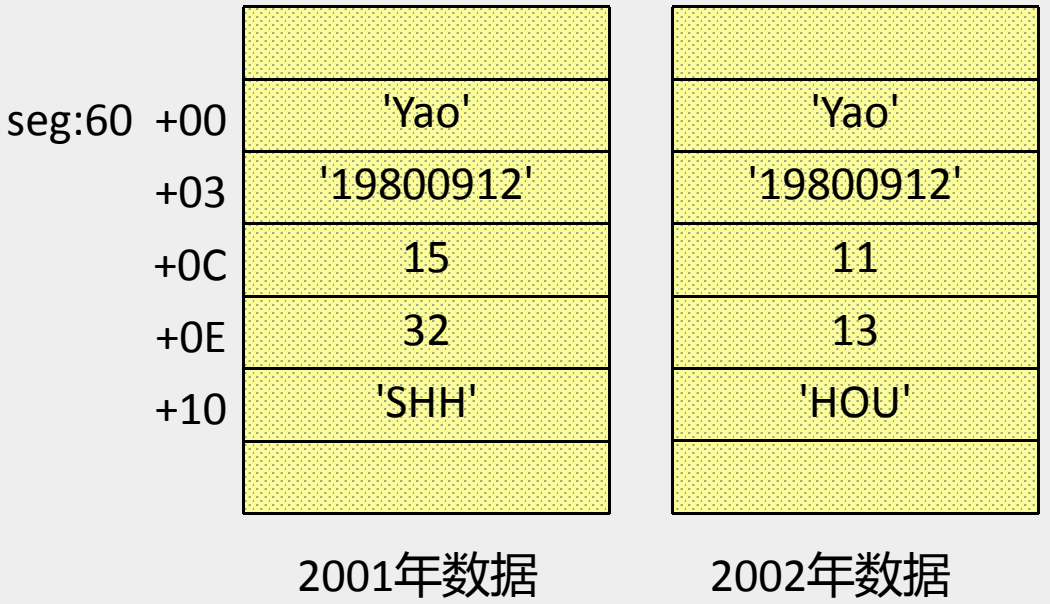
- 关于姚明2001年的一条记录：
  - 姓名：Yao
  - 生日：'19800912'
  - 球衣号码：15
  - 场均得分：32
  - 效力球队：SHH(上海)

		2001年数据	2002年数据
seg:60	+00	'Yao'	'Yao'
	+03	'19800912'	'19800912'
	+0C	15	11
	+0E	32	13
	+10	'SHH'	'HOU'

- 2002年，姚明的信息有了变化：
  1、球衣号码变换成了11号
  2、场均得分为13
  3、效力球队变为NBA的休斯顿火箭队（HOU）
- 任务：编程修改内存中的过时数据。

解决方案

```
mov ax,seg
mov ds,ax
mov bx,60h
mov word ptr [bx+0ch],11
mov word ptr [bx+0eh],13

mov si,0
mov byte ptr [bx+10h+si],’H’
inc si
mov byte ptr [bx+10h+si],’O’
inc si
mov byte ptr [bx+10h+si],’U’
```



汇编指令中寻址的其他写法

- [bx+idata]
  - [bx].idata
- [bx+idata+si]
  - [bx].idata[si]

[bx+10h+si]

=

[bx].10h[si]

C语言和汇编的处理方式对比

```c
#include <stdio.h>
struct Player{
    char name[3];
    char birthday[9];
    int num;
    int ppg; //Points Per Game
    char team[3];
};
struct Player yao={"Yao", "19800912", 15, 32, "SHH"};
int main()
{
    int i;
    yao.num = 11;
    yao.ppg = 13;
    i = 0;
    yao.team[i] = 'H';
    i++;
    yao.team[i] = 'O';
    i++;
    yao.team[i] = 'U';
    return 0;
}
```

yao.team[i]：yao是一个变量名，指明了**结构体变量的地址**；team是一个名称，指明了**数据项**team的地址；i用来定位team**中的字符。**

用bx定位**整个结构体**；用idata定位结构体中的某一个**数据项**；用 si 定位**数据项中的元素**。

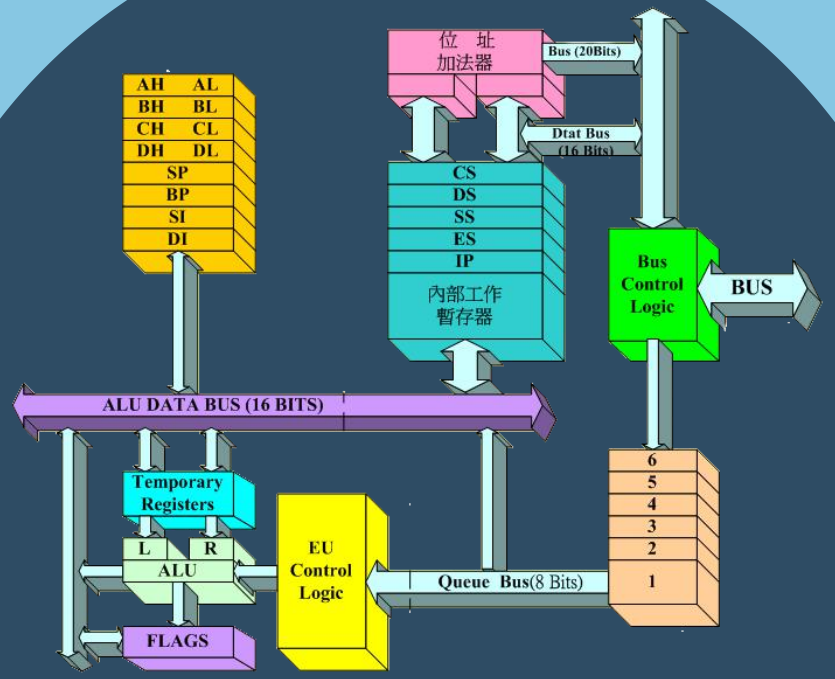
		2001年数据	2002年数据
seg:60	+00	'Yao'	'Yao'
	+03	'19800912'	'19800912'
	+0C	15	11
	+0E	32	13
	+10	'SHH'	'HOU'

```
mov ax,seg
mov ds,ax
mov bx,60h
mov word ptr [bx+0ch],11
mov word ptr [bx+0eh],13

mov si,0
mov byte ptr [bx+10h+si],'H'
inc si
mov byte ptr [bx+10h+si],'O'
inc si
mov byte ptr [bx+10h+si],'U'
```

8086CPU提供的如[bx+si+idata]的寻址方式为结构化数据的处理提供了方便，使得我们可以在编程的时候，从结构化的角度去看待所要处理的数据。

用div指令实现除法

贺利坚　主讲



汇编语言程序设计

Assembly Language

div 指令

- div是除法指令，使用div作除法的时候
  - 被除数：（默认）放在AX 或 DX和AX中
  - 除数：8位或16位，在寄存器或内存单元中
  - 结果：……
- div指令格式
  - div 寄存器
  - div 内存单元

除法的位数示例

6879H÷A2H：商A5，余FH

12345678H÷2EF7H：

商633AH，余2D82H

被除数	AX	DX和AX
除数	8位内存或寄存器	16位内存或寄存器
商	AL	AX
余数	AH	DX

示例指令	被除数	除数	商	余数
div bl	(ax)	(bl)	(al)	(ah)
div byte ptr ds:[0]	(ax)	((ds)*16+0)	(al)	(ah)
div byte ptr [bx+si+8]	(ax)	((ds)*16+(bx)+(si)+8)	(al)	(ah)
div bx	(dx)*10000H+(ax)	(bx)	(ax)	(dx)
div word ptr es:[0]	(dx)*10000H+(ax)	((ds)*16+0)	(ax)	(dx)
div word ptr [bx+si+8]	(dx)*10000H+(ax)	((ds)*16+(bx)+(si)+8)	(ax)	(dx)

切记提前在默认的寄存器中设置好被除数，且默认寄存器不作别的用处。

div 指令示例

- 编程：利用除法指令计算100001/100。
- 分析
  - 100001D=186A1H
  - 需要进行16位的除法
  - 用dx和ax两个寄存器联合存放186A1H
  - 用bx存放除数100D=64H

```
-a
073F:0100 mov dx, 1
073F:0103 mov ax, 86A1
073F:0106 mov bx, 64
073F:0109 div bx
073F:010B
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 BA0100        MOV     DX,0001
-g 073F:0109

AX=86A1  BX=0064  CX=0000  DX=0001  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0109   NV UP EI PL NZ NA PO NC
073F:0109 F7F3          DIV     BX
-t

AX=03E8  BX=0064  CX=0000  DX=0001  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=010B   NV UP EI PL NZ NA PO NC
073F:010B 0000          ADD     [BX+SI],AL                      DS:00
-
```

- 编程：利用除法指令计算1001/100。
- 分析
  - 进行8位除法即可
  - 在ax寄存器存放被除数3E9H
  - 用bx存放除数100D=64H

```
-a
073F:0100 mov ax, 3E9
073F:0103 mov bl, 64
073F:0105 div bl
073F:0107
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0100   NV UP EI PL NZ NA PO NC
073F:0100 B8E903        MOV     AX,03E9
-g 073F:105

AX=03E9  BX=0064  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0105   NV UP EI PL NZ NA PO NC
073F:0105 F6F3          DIV     BL
-t

AX=010A  BX=0064  CX=0000  DX=0000  SP=00FD  BP=0000  SI=0000  DI=0000
DS=073F  ES=073F  SS=073F  CS=073F  IP=0107   NV UP EI PL NZ NA PO NC
073F:0107 64            DB      64
-
```

在内存单元中实施除法

双字型数据的定义(例示)

```
data segment
        db 1    ;定义字节型数据01H，在data:0处，占1个字节
        dw      ;定义字型数据0001H，在data:1处，占2个字节
        dd 1    ;定义双字型数据00000001H，在data:3处，占2个字（4个字节）
data ends
```

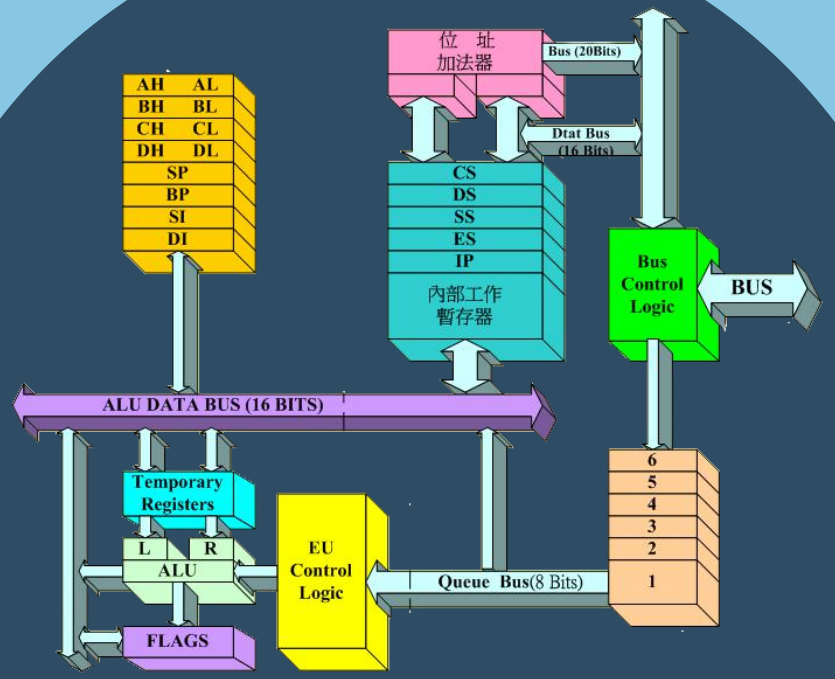
例：用div 计算data段中第一个数据除以第二个数据后的结果，商存放在第3个数据的存储单元中。

```
data segment
        dd 100001
        dw 100
        dw 0
data ends
```

```
mov ax,data
mov ds,ax
mov ax,ds:[0]
mov dx,ds:[2]
div word ptr ds:[4];
mov ds:[6],ax
```

用dup设置内存空间

贺利坚　主讲



汇编语言程序设计

Assembly Language

dup功能和用法

- 功能：dup和db、dw、dd 等数据定义伪指令配合使用，用来进行数据的重复。
- 示例

指令	功能	相当于
db 3 dup (0)	定义了3个字节，它们的值都是0	db 0,0,0
db 3 dup (0,1,2)	定义了9个字节，由0、1、2重复3次构成	db 0,1,2,0,1,2,0,1,2
db 3 dup (‘abc’,’ABC’)	定义了18个字节，构成'abcABCabcABCabcABC'	db ‘abcABCabcABCabcABC’

- dup的使用格式
  - db 重复的次数 dup （重复的字节型数据）
  - dw 重复的次数 dup （重复的字型数据）
  - dd 重复的次数 dup （重复的双字数据）

dup用途

例示：定义一个容量为 200 个字节的栈段

不采用dup的格式

```
stack segment
  dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
  dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
  dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
  dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
  dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
stack ends
```

采用dup的格式

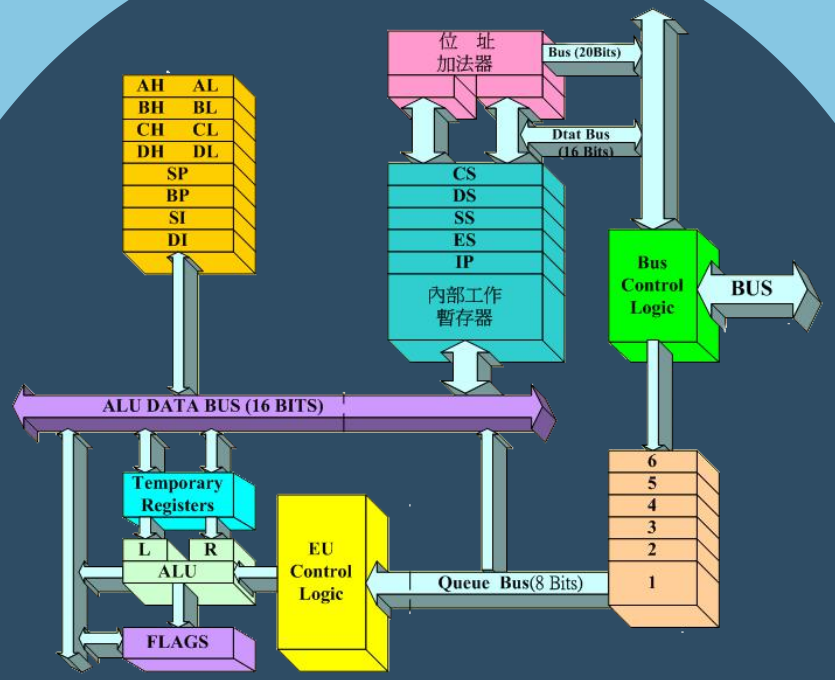
```
stack segment
        db 200 dup (0)
stack ends
```

再例

```
1  assume cs:code, ds:data
2  data segment
3      db 3 dup (0)
4      db 3 dup (0,1,2)
5      db 80 dup (0)
6      db 3 dup ('abc','ABC')
7  data ends
8  code segment
9      mov ax, data
10     mov ds, ax
11
12     mov ax, 4c00H
13     int 21H
14 code ends
15 end
```

导学-流程转移与子程序

贺利坚　主讲



汇编语言程序设计

Assembly Language

汇编语言程序设计课程内容

1. 绪论
2. 访问寄存器和内存
3. 汇编语言程序
4. 内存寻址方式
5. 流程转移与子程序
6. 中断及其应用
7. 高级汇编语言技术

0901 “转移”综述
0902 操作符offset
0903 jmp指令
0904 其他转移指令

1001 call指令和ret指令
1002 call 和 ret 的配合使用
1003 mul 指令
1004 汇编语言的模块化程序设计
1005 寄存器冲突的问题

1101 标志寄存器
1102 带进(借)位的加减法
1103 cmp和条件转移指令
1104 条件转移指令应用
1105 DF标志和串传送指令

各节与教材章节的对应关系

视频（共14个）	教材对应章节
0901 “转移”综述	第9章 引子
0902 操作符offset	9.1
0903 jmp指令	9.2-9.6，9.10
0904 其他转移指令	9.7-9.9
1001 call指令和ret指令	10.1-10.6
1002 call 和 ret 的配合使用	10.7
1003 mul 指令	10.8
1004 汇编语言的模块化程序设计	10.9-10.11
1005 寄存器冲突的问题	10.12
1101 标志寄存器	11.1-11.5，11.11-11.12
1102 带进(借)位的加减法	11.6-11.7
1103 cmp和条件转移指令	11.8-11.9
1104 条件转移指令应用	11.9
1105 DF标志和串传送指令	11.10

视频

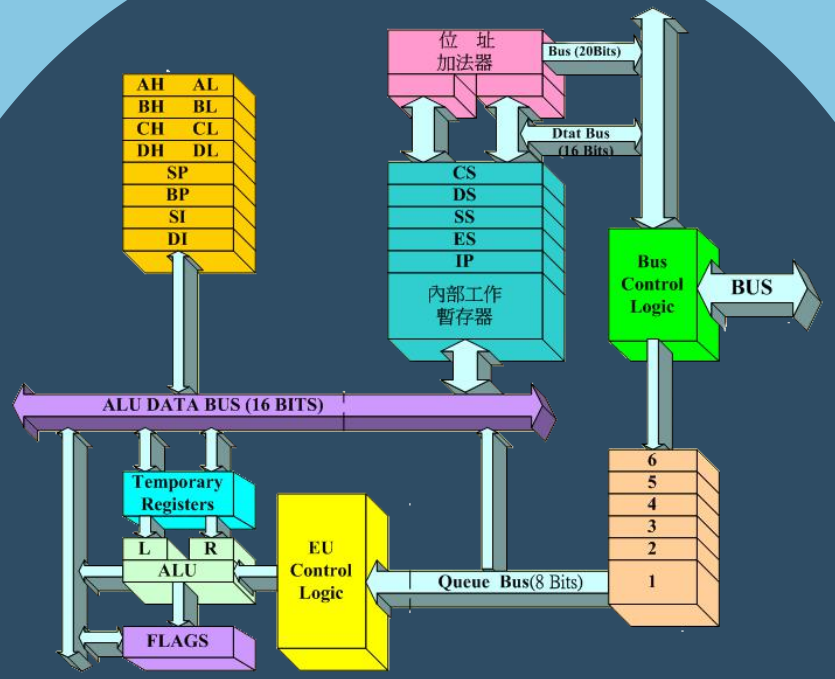
教材

检测

实验

“转移”综述

贺利坚　主讲

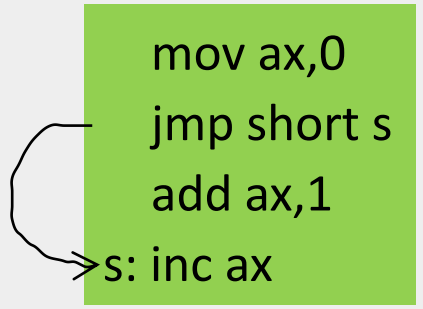


汇编语言程序设计

Assembly Language

转移综述

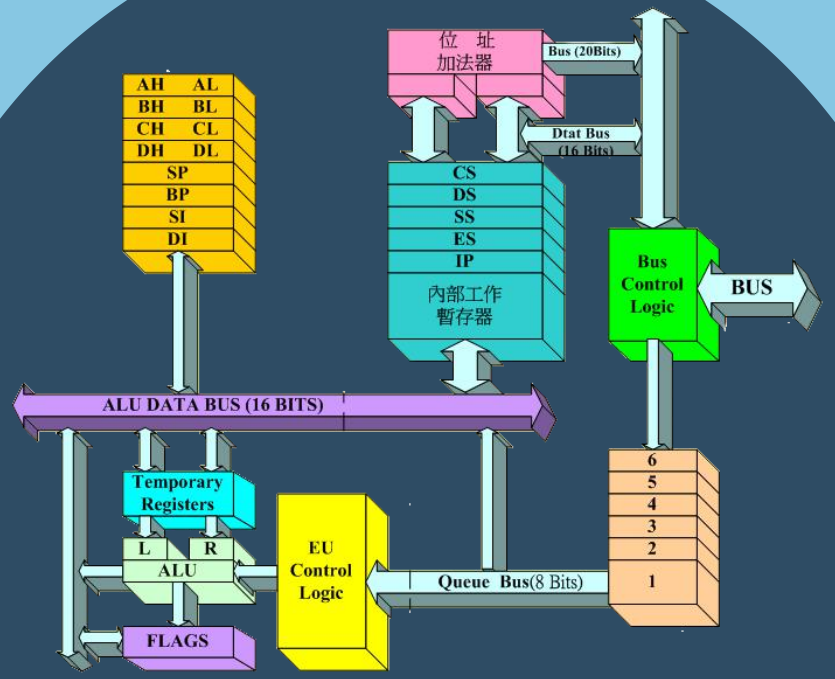
- 背景：一般情况下指令是顺序地逐条执行的，而在实际中，常需要改变程序的执行流程。
- 转移指令，
  - 可以控制CPU执行内存中某处代码的指令
  - 可以修改IP，或同时修改CS和IP的指令



- 转移指令的分类
  - 按转移行为
    - 段内转移：只修改IP，如jmp ax
    - 段间转移：同时修改CS和IP，如jmp 1000:0
  - 根据指令对IP修改的范围不同
    - 段内短转移：IP修改范围为-128~127
    - 段内近转移：IP修改范围为-32768~32767
  - 按转移指令
    - 无条件转移指令（如：jmp）
    - 条件转移指令（如：jcxz）
    - 循环指令（如：loop）
    - 过程
    - 中断

操作符offset

贺利坚　主讲



汇编语言程序设计

Assembly Language

用操作符offset取得标号的偏移地址

格式：

offset 标号

例：

```
assume cs:codeseg
codeseg segment
start: mov ax,offset start ; 相当于 mov ax,0
     s: mov ax,offset s      ; 相当于mov ax,3
codeseg ends
end start
```

```
1  assume cs:codeseg
2  codeseg segment
3  start: mov ax,offset start  ; 相当于mov ax,0
4       s: mov ax,offset s     ; 相当于mov ax,3
5  codeseg ends
6  end start
```

```
C:\>debug p9-1.exe
-u
076A:0000 B80000        MOV     AX,0000
076A:0003 B80300        MOV     AX,0003
076A:0006 C404          LES     AX,[SI]
076A:0008 3DFFFF        CMP     AX,FFFF
076A:000B 7403          JZ      0010
```

练习

问题：有如下程序段，添写2条指令，使该程序在运行中将s处的一条指令复制到s0处。

```
assume cs:codesg
codesg segment
  s: mov ax,bx
     mov si,offset s
     mov di,offset s0
     mov ax,cs:[si]
     mov cs:[di],ax
s0:  nop
     nop
codesg ends
ends
; nop的机器码占一个字节，起“占位”作用
```

分析

（1）s和s0处的指令所在的内存单元的地址是多少？

cs:offset s 和cs:offset s0

（2）将s处的指令复制到s0处，就是_____

就是将cs:offset s 处的数据复制到cs:offset s0处

（3）地址如何表示？

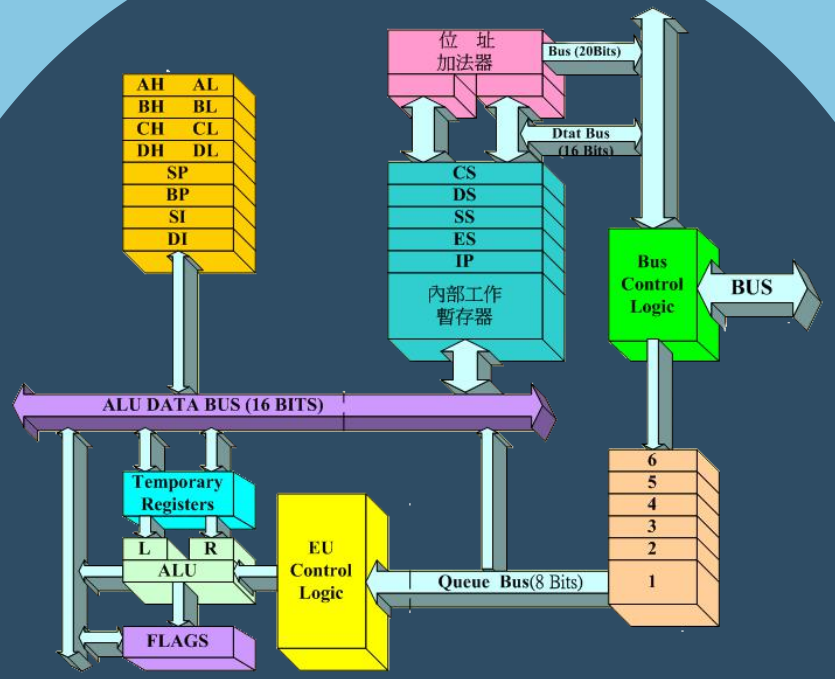
段地址已知在cs中，偏移地址已经送入si和di中

（4）要复制的数据有多长？

mov ax,bx指令的长度为两个字节，即1个字。

jmp指令

贺利坚　主讲

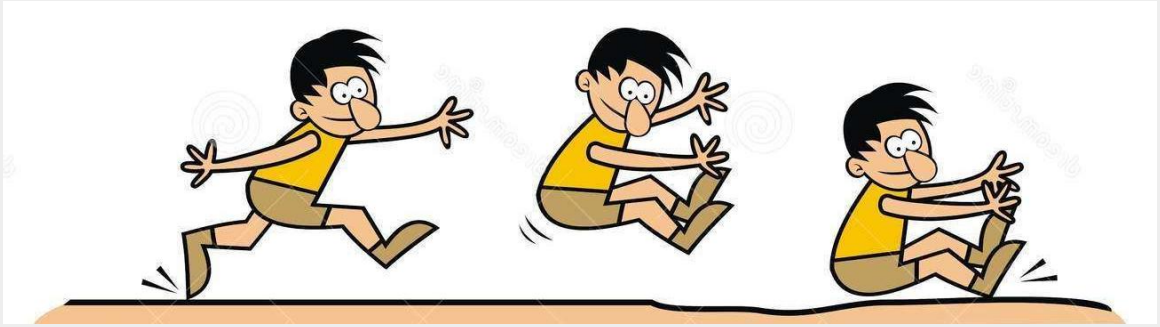


汇编语言程序设计

Assembly Language

jmp指令——无条件转移

- jmp指令的功能
  - 无条件转移，可以只修改IP，也可以同时修改CS和IP
- jmp指令要给出两种信息：
  - 转移的目的地址
  - 转移的距离
    - 段间转移（远转移）：　jmp 2000:1000
    - 段内短转移： jmp short 标号 ；IP的修改范围为 -128~127，8位的位移
    - 段内近转移： jmp near ptr 标号 ；IP的修改范围为 -32768~32767，16位的位移



jmp指令：依据位移进行转移

```
-a 073f:0100
073F:0100 mov ax, 0123
073F:0103 mov ax, [0123]
073F:0106 push [0123]
073F:010A
-u 073f:0100
073F:0100 B82301        MOV     AX,0123
073F:0103 A12301        MOV     AX,[0123]
073F:0106 FF362301      PUSH    [0123]
```

- 引子：常见指令中的立即数均在机器指令中有体现
- 问题：jmp short 指令中，转移到了哪里？
  - jmp short 的机器指令中，包含的是跳转到指令的相对位置，而不是转移的目标地址。

```
1  assume cs:codesg
2  codesg segment
3    start: mov ax,0
4           jmp short s
5           add ax,1
6         s: inc ax
7  codesg ends
8  end start
```

```
1  assume cs:codesg
2  codesg segment
3    start: mov ax,0
4           jmp short s
5           add ax,1
6           nop
7           nop
8         s: inc ax
9  codesg ends
10 end start
```

```
C:\>debug p9-2.exe
-u
076A:0000 B80000        MOV     AX,0000
076A:0003 EB03          JMP     0008
076A:0005 050100        ADD     AX,0001
076A:0008 40            INC     AX
```

```
C:\>debug p9-3.exe
-u
076A:0000 B80000        MOV     AX,0000
076A:0003 EB05          JMP     000A
076A:0005 050100        ADD     AX,0001
076A:0008 90            NOP
076A:0009 90            NOP
076A:000A 40            INC     AX
```

- 左边程序jmp short s指令的读取和执行：

（1）(IP)=0003，CS:IP指向EB 05(jmp 的机器码)

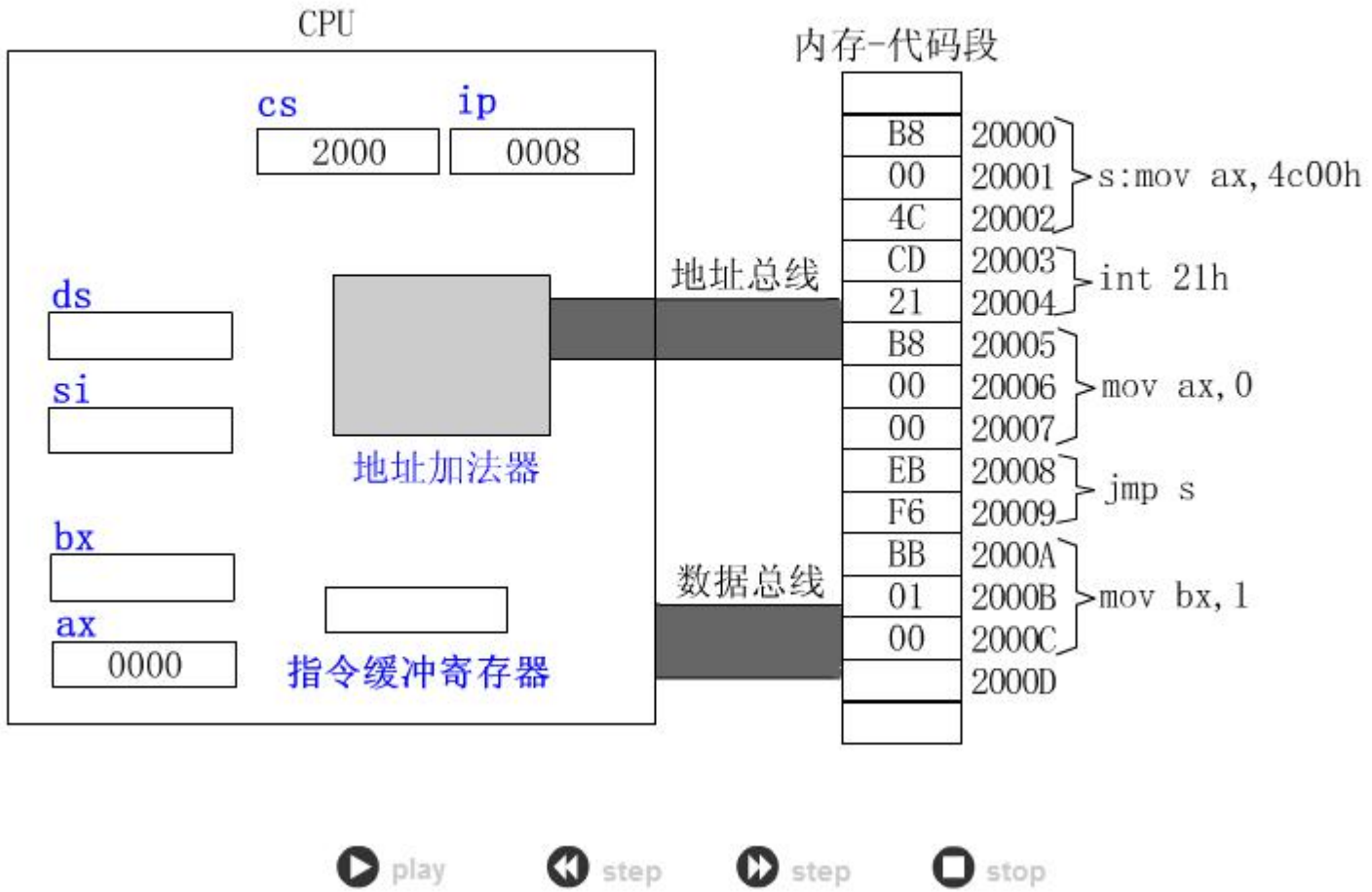
（2）读取指令码EB 05进入指令缓冲器；

（3）(IP)=(IP)+所读取指令的长度=(IP)+2=0005，CS:IP指向add ax, 0001；

（4）CPU执行指令缓冲器中的指令EB05；

（5）指令EB 05执行后，(IP)=(IP)+05=000AH，CS:IP指向inc ax

依据位移进行转移的jmp指令



play step step stop

两种段内转移

短转移： “jmp short 标号”

```
1  assume cs:codesg
2  codesg segment
3    start: mov ax,0
4           jmp short s
5           add ax,1
6         s: inc ax
7  codesg ends
8  end start
```

- 功能：(IP)=(IP)+8位位移
- 原理

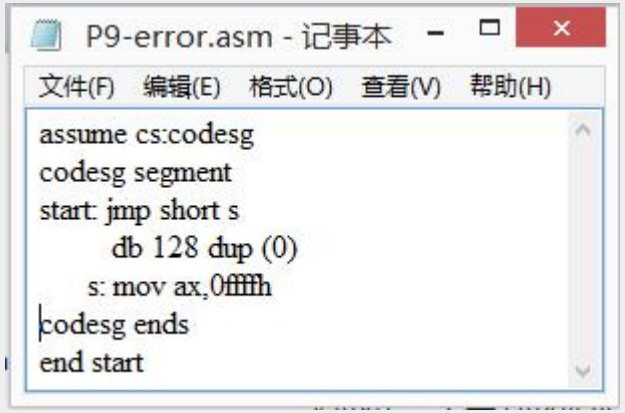
（1）8位位移=“标号”处的地址-jmp指令后的第一个字节的地址；

（2）short指明此处的位移为8位位移；

（3）8位位移的范围为-128~127，用**补码**表示；

（4）8位位移由编译程序在编译时算出。

我要这样！



近转移：指令“jmp near ptr 标号”

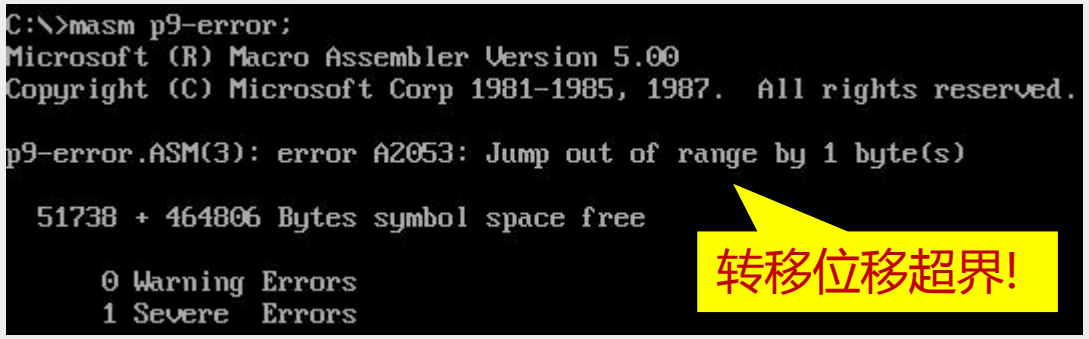
- 功能： (IP)=(IP)+16位位移
- 原理

（1）16位位移=“标号”处的地址-jmp指令后的第一个字节的地址；

（2）near ptr指明此处的位移为16位位移，进行的是段内近转移；

（3）16位位移的范围为 -32769~32767，用补码表示；

（4）16位位移由编译程序在编译时算出。



远转移：jmp far ptr 标号

远转移jmp far ptr 标号	近转移jmp near ptr 标号
段间转移	段内转移
far ptr指明了跳转到的**目的地址**，即包含了标号的段地址CS和偏移地址IP。	near ptr 指明了相对于当前IP的**转移位移**，而不是转移的目的地址。

```
1  assume cs:codesg
2  codesg segment
3 start: mov ax,0
4        mov bx,0
5        jmp far ptr s
6        db 256 dup (0)
7    s: add ax,1
8       inc ax
9  codesg ends
10 end start
```

```
C:\>debug p9-5.exe
-u
076A:0000 B80000        MOV     AX,0000
076A:0003 BB0000        MOV     BX,0000
076A:0006 EA0B016A07    JMP     076A:010B
076A:000B 0000          ADD     [BX+SI],AL
076A:000D 0000          ADD     [BX+SI],AL
-u 109
076A:0109 0000          ADD     [BX+SI],AL
076A:010B 050100        ADD     AX,0001
076A:010E 40            INC     AX
076A:010F 7DFF          JGE     0110
076A:0111 50            PUSH    AX
```

```
1  assume cs:codesg
2  codesg segment
3 start: mov ax,0
4        mov bx,0
5        jmp near ptr s
6        db 256 dup (0)
7    s: add ax,1
8       inc ax
9  codesg ends
10 end start
```

```
C:\>debug p9-4.exe
-u
076A:0000 B80000        MOV     AX,0000
076A:0003 BB0000        MOV     BX,0000
076A:0006 E90001        JMP     0109
076A:0009 0000          ADD     [BX+SI],AL
076A:000B 0000          ADD     [BX+SI],AL
-u 109
076A:0109 050100        ADD     AX,0001
076A:010C 40            INC     AX
076A:010D 8D867DFF      LEA     AX,[BP+FF7D]
076A:0111 50            PUSH    AX
```

转移地址在寄存器中的jmp指令

- 指令格式：jmp 16位寄存器
  - 功能：IP =（16位寄存器）
  - 举例：

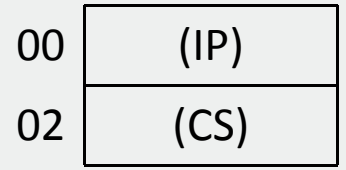
jmp ax

jmp bx

```
1  assume cs:codesg
2  codesg segment
3  start: mov ax,0
4         mov bx,ax
5         jmp bx
6         mov ax,0123H
7  codesg ends
8  end start
```

```
C:\>debug p9-6.exe
-u
076A:0000 B80000        MOV     AX,0000
076A:0003 8BD8          MOV     BX,AX
076A:0005 FFE3          JMP     BX
076A:0007 B82301        MOV     AX,0123
076A:000A 07            POP     ES
```

转移地址在内存中的jmp指令

<table>
<tr><th>jmp word ptr 内存单元地址</th><th>jmp dword ptr 内存单元地址</th></tr>
<tr><td>段内转移</td><td>段间转移</td></tr>
<tr><td>功能：从内存单元地址处开始存放着一个字，是转移的目的偏移地址。</td><td>功能：从内存单元地址处开始存放着两个字，高地址处的字是转移的目的段地址，低地址处是转移的目的偏移地址。
 
</td></tr>
<tr><td>mov ax,0123H
mov ds:[0],ax
jmp word ptr ds:[0]
执行后，(IP)=0123H

mov ax,0123H
mov [bx],ax
jmp word ptr [bx]
执行后，(IP)=0123H</td><td>mov ax,0123H
mov ds:[0],ax
mov word ptr ds:[2],0
jmp dword ptr ds:[0]
执行后，
(CS)=0
(IP)=0123H
CS:IP指向0000:0123

mov ax,0123H
mov [bx],ax
mov word ptr [bx+2],0
jmp dword ptr [bx]
执行后，
(CS)=0
(IP)=0123H
CS:IP指向0000:0123</td></tr>
</table>

jmp指令小结

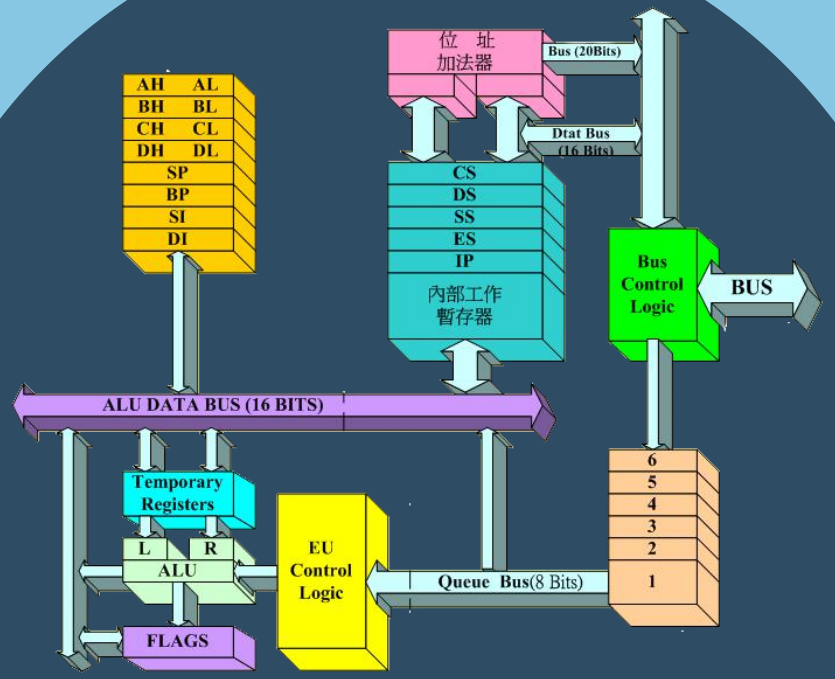
jmp指令格式	示例
jmp 标号	– 段间转移（远转移）：jmp far ptr 标号 – 段内短转移： jmp short 标号 ；8位的位移 – 段内近转移： jmp near ptr 标号 ；16位的位移
jmp 寄存器	– jmp bx ; 16位的位移
jmp 内存单元(表示跳转到的地址)	– 段内转移：jmp word ptr 内存单元地址 ;jmp word ptr [bx] – 段间转移：jmp dword ptr 内存单元地址 ;jmp dword ptr [bx]

在源程序中，不允许使用“jmp 2000:0100”的转移指令实现段间转移

- 这是在 Debug 中使用的汇编指令，汇编编译器并不认识
- 如果在源程序中使用，编译时也会报错。

其他转移指令

贺利坚　主讲



汇编语言程序设计

Assembly Language

jcxz指令

- 指令格式：jcxz 标号
- 功能：如果(cx)=0，则转移到标号处执行
  当(cx)≠0时，什么也不做（程序向下执行）
  - 当(cx)=0时，(IP)=(IP)+8位位移）
    - 8位位移=“标号”处的地址-jcxz指令后的第一个字节的地址；
    - 8位位移的范围为-128~127，用补码表示；
    - 8位位移由编译程序在编译时算出。
- jcxz是有条件转移指令
  - 所有的有条件转移指令都是短转移
  - 对IP的修改范围都为-128~127
  - 在对应的机器码中包含转移的位移，而不是目的地址

```
1  assume cs:codesg
2  codesg segment
3  start: mov ax,2000H
4         mov ds, ax
5         mov bx,0
6      s: mov cx, [bx]
7         jcxz ok
8         inc bx
9         inc bx
10        jmp short s
11     ok: mov dx, bx
12        mov ax, 4c00H
13        int 21H
14 codesg ends
15 end start
```

```
C:\>debug p9-7.exe
-u
076A:0000 B80020        MOV     AX,2000
076A:0003 8ED8          MOV     DS,AX
076A:0005 BB0000        MOV     BX,0000
076A:0008 8B0F          MOV     CX,[BX]
076A:000A E304          JCXZ    0010
076A:000C 43            INC     BX
076A:000D 43            INC     BX
076A:000E EBF8          JMP     0008
076A:0010 8BD3          MOV     DX,BX
076A:0012 B8004C        MOV     AX,4C00
076A:0015 CD21          INT     21
```

loop指令

- 指令格式：loop 标号
- 指令操作

（1）(cx)=(cx)-1;

（2）当(cx)≠0时，则转移到标号处执行

当(cx)=0时，程序向下执行

- 如果(cx)≠0，(IP)=(IP)+8位位移
  - 8位位移=“标号”处的地址-loop指令后的第一个字节的地址
  - 8位位移的范围为-128~127，用补码表示
  - 8位位移由编译程序在编译时算出

```
1  assume cs:codesg
2  codesg segment
3  start: mov cx, 6h
4         mov ax,10h
5      s: add ax, ax
6         loop s
7         mov ax, 4c00H
8         int 21H
9  codesg ends
10 end start
```

```
C:\>debug p9-8.exe
-u
076A:0000 B90600        MOV     CX,0006
076A:0003 B81000        MOV     AX,0010
076A:0006 03C0          ADD     AX,AX
076A:0008 E2FC          LOOP    0006
076A:000A B8004C        MOV     AX,4C00
076A:000D CD21          INT     21
076A:000F 01B85C00      ADD     [BX+SI+005C
```

loop s 在执行时只涉及到 s的位移（-4，前移4个字节，补码表示为FCH）

根据位移进行“相对”转移的意义

- 对 IP的修改是根据转移目的地址和转移起始地址之间的位移来进行

  jmp short 标号

  jmp near ptr 标号

  jcxz 标号

  loop 标号

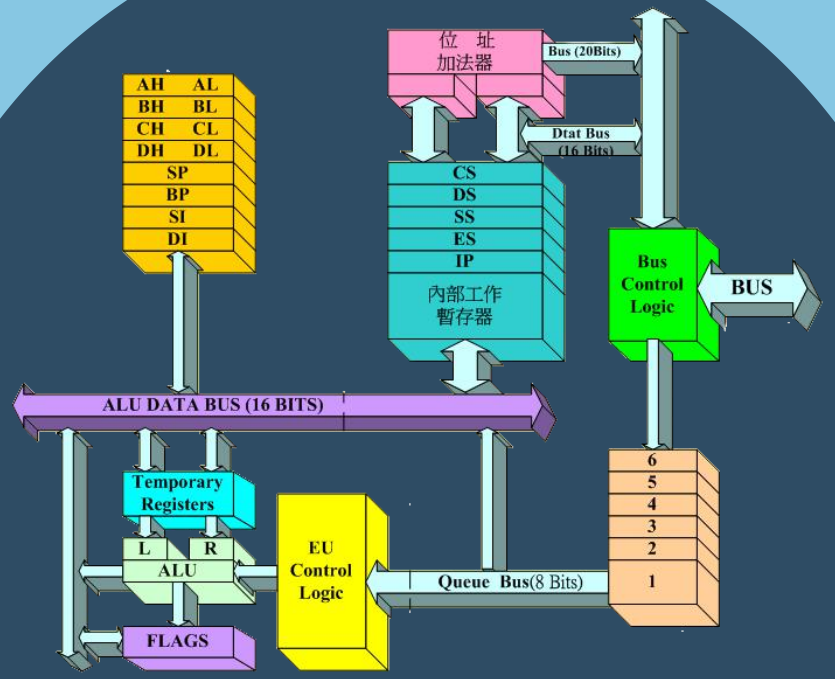
- 在它们对应的机器码中不包含转移的目的地址，而包含的是到目的地址的位移。
  - 如果 loop s 的机器码中包含的是 s 的地址，则就对程序段在内存中的偏移地址有了严格的限制，易引发错误。
  - 当机器码中包含的是转移的位移，无论 s 处的指令的实际地址是多少，loop指令转移的相对位移是不变的。
- 这样的设计，方便了程序段在内存中的浮动装配。

```
C:\>debug p9-7.exe
-u
076A:0000 B80020        MOV     AX,2000
076A:0003 8ED8          MOV     DS,AX
076A:0005 BB0000        MOV     BX,0000
076A:0008 8B0F          MOV     CX,[BX]
076A:000A E304          JCXZ    0010
076A:000C 43            INC     BX
076A:000D 43            INC     BX
076A:000E EBF8          JMP     0008
076A:0010 8BD3          MOV     DX,BX
076A:0012 B8004C        MOV     AX,4C00
076A:0015 CD21          INT     21
```

```
C:\>debug p9-8.exe
-u
076A:0000 B90600        MOV     CX,0006
076A:0003 B81000        MOV     AX,0010
076A:0006 03C0          ADD     AX,AX
076A:0008 E2FC          LOOP    0006
076A:000A B8004C        MOV     AX,4C00
076A:000D CD21          INT     21
076A:000F 01B85C00      ADD     [BX+SI+005C
```

call指令和ret指令

贺利坚　主讲

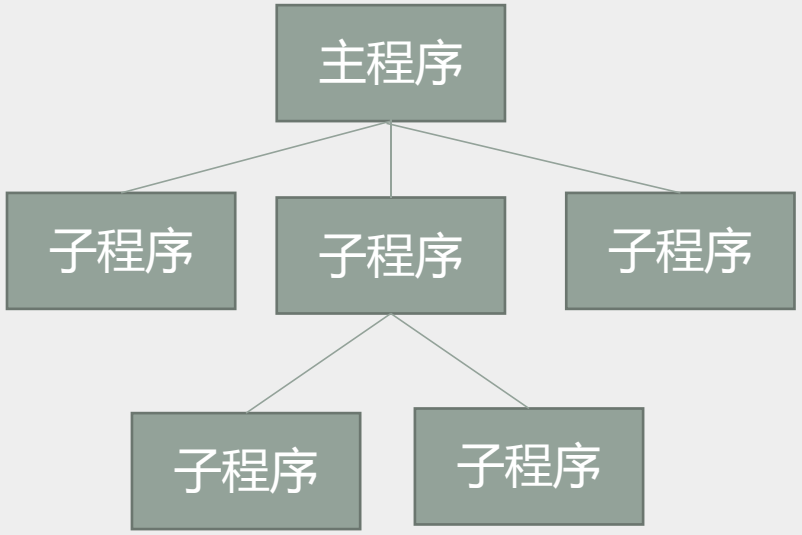


汇编语言程序设计

Assembly Language

模块化程序设计

```
#include <stdio.h>
int cube(int x);
int main()
{
   printf("%d\n",cube(2));
   return 0;
}
int cube(int x)
{
   int f;
   f=x*x;
   f=f*x;
   return f;
}
```



- 调用子程序：call指令
- 返回：ret 指令
- 示例

```
   mov ax, 0
   call s
   mov ax, 4c00h
   int 21h

s: add ax, 1
   ret
```

- 实质：流程转移指令，它们都修改IP，或同时修改CS和IP

call 指令

- 字面意思：调用子程序
- 实质：流程转移
  - call指令实现转移的方法和 jmp 指令的原理相似
- 格式：call 标号
- CPU执行call指令，进行两步操作：
  （1）将当前的 IP 或 CS和IP 压入栈中；
  （2）转移到标号处执行指令。
- call 标号
  - 16位位移=“标号”处的地址 - call指令后的第一个字节的地址；
  - 16位位移的范围为 -32768~32767，用补码表示；
  - 16位位移由编译程序在编译时算出。

```
call 标号
```

```
(1) (sp) = (sp) – 2
    ((ss)*16+(sp)) = (IP)
(2) (IP) = (IP) + 16位位移
```

```
mov ax, 0
call s
mov ax, 4c00h
int 21h

s: add ax, 1
ret
```

相当于：

```
push IP
jmp near ptr 标号
```

指令“call far ptr 标号”实现的是段间转移

- CPU执行“call far ptr 标号”时的操作

  (1) (sp) = (sp) – 2

  ((ss) ×16+(sp)) = (CS)

  (sp) = (sp) – 2

  ((ss) ×16+(sp)) = (IP)

  (2) (CS) = 标号所在的段地址

  (IP) = 标号所在的偏移地址

- “call far ptr 标号” 相当于

  push CS

  push IP

  jmp far ptr 标号

- “call 标号”类似”jmp near ptr 标号”，对应的机器指令中为相对于当前IP的**转移位移，而不是转移的目的地址，实现段内转移**。
- 指令“call far ptr 标号”**实现的是段间转移**！

```
    mov ax, 0
    call far ptr s
    ……

    mov ax, 4c00h
    int 21h

s: add ax, 1
    ret
```

转移地址在寄存器中的call指令

- 指令格式
  - call 16位寄存器
- 功能
  - (sp) = (sp) – 2
  - ((ss)*16+(sp)) = (IP)
  - (IP) = (16位寄存器)
- 相当于进行
  - push IP
  - jmp 16位寄存器

```
mov ax, 0
call ax
……

mov ax, 4c00h
int 21h
```

转移地址在内存中的call指令

- call word ptr 内存单元地址

  相当于：

  push IP

  jmp word ptr 内存单元地址

```
mov sp,10h
 mov ax,0123h
 mov ds:[0],ax
 call word ptr ds:[0]
执行后，(IP)=0123H，(sp)=0EH
```

- call dword ptr 内存单元地址

  相当于

  push CS

  push IP

  jmp dword ptr 内存单元地址

```
mov sp,10h
mov ax,0123h
mov ds:[0],ax
mov word ptr ds:[2],0
call dword ptr ds:[0]
执行后，(CS)=0，(IP)=0123H，(sp)=0CH
```

低地址放偏移地址

高地址放段地址

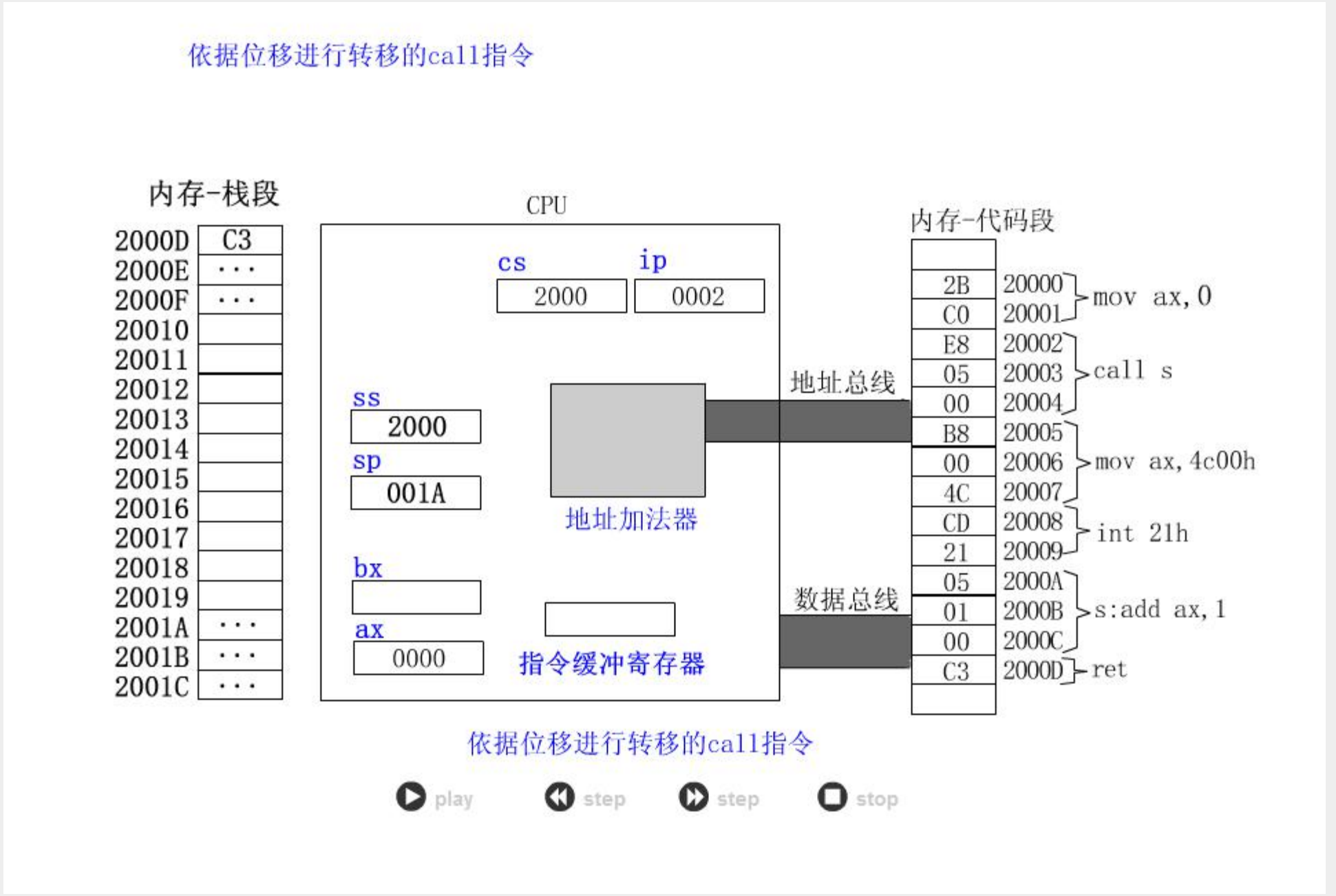
返回指令：ret 和 retf

	ret指令	retf指令
功能	用栈中的数据，修改IP的内容，从而实现近转移；	用栈中的数据，修改CS和IP的内容，从而实现远转移；
相当于	pop IP	pop IP pop CS
举例	1 assume cs:codesg,ss:stack 2 stack segment 3 db 16 dup (0) 4 stack ends 5 codesg segment 6 mov ax,4c00h 7 int 21h 8 start: mov ax,stack 9 mov ss,ax 10 mov sp,16 11 mov ax,0 12 push ax 13 mov bx,0 14 ret 15 codesg ends 16 end start	1 assume cs:codesg,ss:stack 2 stack segment 3 db 16 dup (0) 4 stack ends 5 codesg segment 6 mov ax,4c00h 7 int 21h 8 start: mov ax,stack 9 mov ss,ax 10 mov sp,16 11 mov ax,0 12 push cs 13 push ax 14 mov bx,0 15 retf 16 codesg ends 17 end start

演示

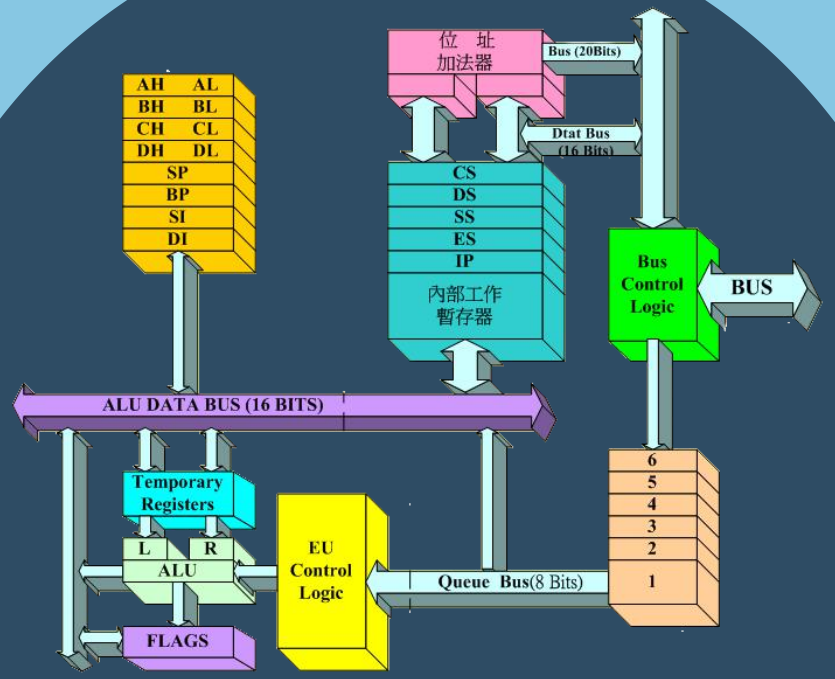
- CALL指令和RET指令执行的过程

```
    mov ax, 0
    call s
    mov ax, 4c00h
    int 21h

s: add ax, 1
    ret
```



call 和 ret 的配合使用

贺利坚　主讲



汇编语言程序设计

Assembly Language

具有子程序的源程序的框架

```
1  assume cs:code
2  code segment
3  main: ...
4        call sub1       ;调用子程序sub1
5        ...
6        mov ax, 4c00h
7        int 21h
8
9  sub1: ...             ;子程序sub1开始
10       call sub2       ;调用子程序sub1
11       ...
12       ret             ;子程序返回
13
14 sub2: ...             ;子程序sub2开始
15       ...
16       ret             ;子程序返回
17 code ends
18 end main
```

调用程序的框架

```
... ...
call 标号
... ...
```

子程序的框架

```
标号:
    指令
    ret
```

call 和 ret 的配合使用

例：

计算2的N次方，计算前，N的值由CX提供。

```
1   assume cs:code
2   code segment
3   start: mov ax,1
4          mov cx,3
5          call s
6          mov bx,ax
7          mov ax,4c00h
8          int 21h
9       s: add ax,ax
10         loop s
11         ret
12  code ends
13  end start
```

```
C:\>debug p10-3.exe
-u
076A:0000 B80100        MOV     AX,0001
076A:0003 B90300        MOV     CX,0003
076A:0006 E80700        CALL    0010
076A:0009 8BD8          MOV     BX,AX
076A:000B B8004C        MOV     AX,4C00
076A:000E CD21          INT     21
076A:0010 03C0          ADD     AX,AX
076A:0012 E2FC          LOOP    0010
076A:0014 C3            RET
```

call要用的栈呢？

```
-r
AX=FFFF  BX=0000  CX=0015  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0000   NV UP EI PL NZ NA PO NC
076A:0000 B80100        MOV     AX,0001
-t

AX=0001  BX=0000  CX=0015  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0003   NV UP EI PL NZ NA PO NC
076A:0003 B90300        MOV     CX,0003
-t

AX=0001  BX=0000  CX=0003  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0006   NV UP EI PL NZ NA PO NC
076A:0006 E80700        CALL    0010
-t

AX=0001  BX=0000  CX=0003  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0010   NV UP EI PL NZ NA PO NC
076A:0010 03C0          ADD     AX,AX
```

```
-t

AX=0008  BX=0000  CX=0001  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0012   NV UP EI PL NZ NA PO NC
076A:0012 E2FC          LOOP    0010
-t

AX=0008  BX=0000  CX=0000  DX=0000  SP=FFFE  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0014   NV UP EI PL NZ NA PO NC
076A:0014 C3            RET
-t

AX=0008  BX=0000  CX=0000  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=0009   NV UP EI PL NZ NA PO NC
076A:0009 8BD8          MOV     BX,AX
-t

AX=0008  BX=0008  CX=0000  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076A  IP=000B   NV UP EI PL NZ NA PO NC
076A:000B B8004C        MOV     AX,4C00
```

例：为call和ret指令设置栈

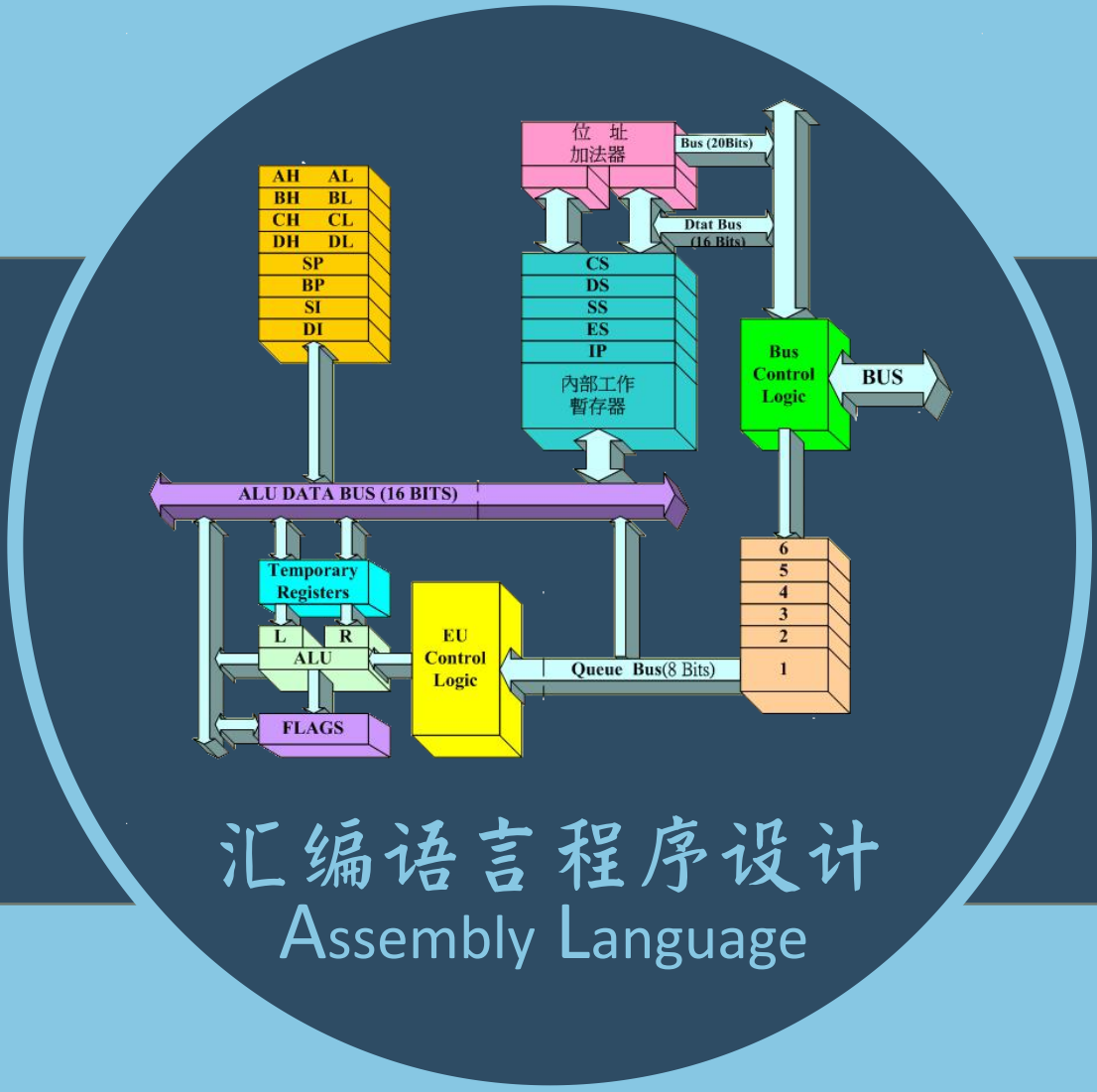
```
1  assume cs:code, ss:stack
2  stack segment
3          db  8 dup (0)
4          db  8 dup (0)
5  stack ends
6  code segment
7  start: mov ax,stack
8          mov ss,ax
9          mov sp,16
10         mov ax,1000
11         call s
12         mov ax,4c00h
13         int 21h
14     s:  add ax,ax
15         ret
16 code ends
17 end start
```

```
C:\>debug p10-4.exe
-u
076B:0000 B86A07        MOV     AX,076A
076B:0003 8ED0          MOV     SS,AX
076B:0005 BC1000        MOV     SP,0010
076B:0008 B8E803        MOV     AX,03E8
076B:000B E80500        CALL    0013
076B:000E B8004C        MOV     AX,4C00
076B:0011 CD21          INT     21
076B:0013 03C0          ADD     AX,AX
076B:0015 C3            RET
```

```
-g 000b

AX=03E8  BX=0000  CX=0026  DX=0000  SP=0010  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=076A  CS=076B  IP=000B   NV UP EI PL NZ NA PO NC
076B:000B E80500        CALL    0013
-d ss:0 f
076A:0000  00 00 00 00 00 00 00 00-00 00 0B 00 6B 07 A3 01   ............
-t

AX=03E8  BX=0000  CX=0026  DX=0000  SP=000E  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=076A  CS=076B  IP=0013   NV UP EI PL NZ NA PO NC
076B:0013 03C0          ADD     AX,AX
-d ss:0 f
076A:0000  00 00 00 00 E8 03 00 00-13 00 6B 07 A3 01 0E 00   ..........k.
-t

AX=07D0  BX=0000  CX=0026  DX=0000  SP=000E  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=076A  CS=076B  IP=0015   NV UP EI PL NZ AC PO NC
076B:0015 C3            RET
-t

AX=07D0  BX=0000  CX=0026  DX=0000  SP=0010  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=076A  CS=076B  IP=000E   NV UP EI PL NZ AC PO NC
076B:000E B8004C        MOV     AX,4C00
```

乘法：mul 指令

贺利坚　主讲



回顾：除法div 指令

- div是除法指令，格式为
  - div 寄存器
  - div 内存单元
- 使用div作除法的时候
  - 被除数：（默认）放在AX 或 DX和AX中
  - 除数：8位或16位，在寄存器或内存单元中
  - 结果：……

被除数	AX	DX和AX
除数	8位内存或寄存器	16位内存或寄存器
商	AL	AX
余数	AH	DX

用 mul 指令做乘法

格式

mul 寄存器

mul 内存单元

	8位乘法	16位乘法
被乘数(默认)	AL	AX
乘数	8位寄存器或内存字节单元	16位寄存器或内存字单元
结果	AX	DX（高位）和AX（低位）
例	mul bl -- (ax)=(al)*(bl) mul byte ptr ds:[0] -- (ax)=(al)*((ds)*16+0)	mul word ptr [bx+si+8] -- (ax)=(ax)*((ds)*16+(bx)+(si)+8)结果的低16位； (dx)=(ax)*((ds)*16+(bx)+(si)+8)结果的高16位；

应用实例

	8位乘法	16位乘法
被乘数(默认)	AL	AX
乘数	8位寄存器或内存字节单元	16位寄存器或内存字单元
结果	AX	DX（高位）和AX（低位）

（1）计算100*10

分析：100和10小于255，可以做8位乘法

程序：

```
mov al,100
mov bl,10
mul bl
```

结果：(ax)=1000（03E8H）

（2）计算100*10000

分析：100小于255，可10000大于255，所以必须做16位乘法

程序：

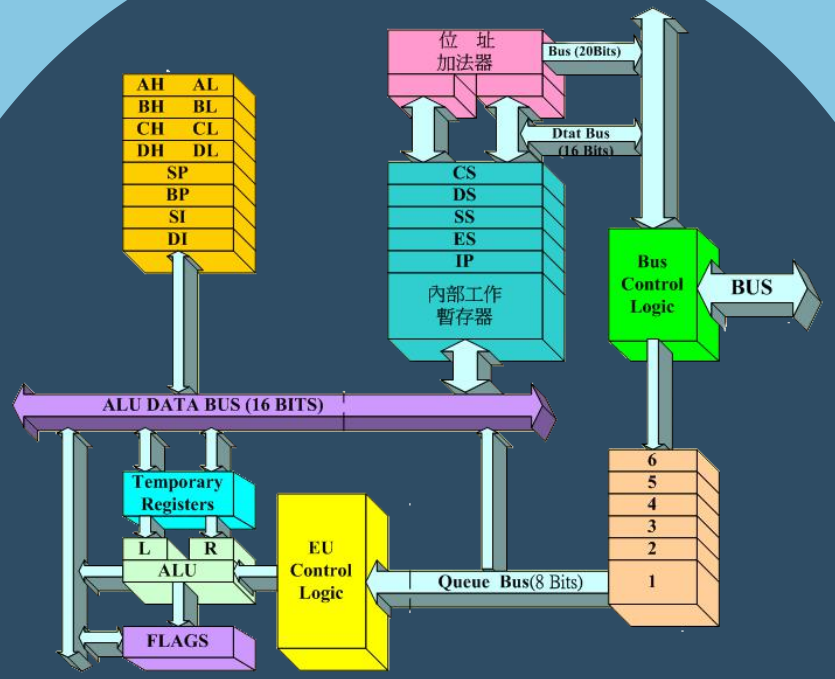
```
mov ax,100
mov bx,10000
mul bx
```

结果：(dx)=000FH，(ax)=4240H，

即：F4240H=1000000

汇编语言的模块化程序设计

贺利坚　主讲

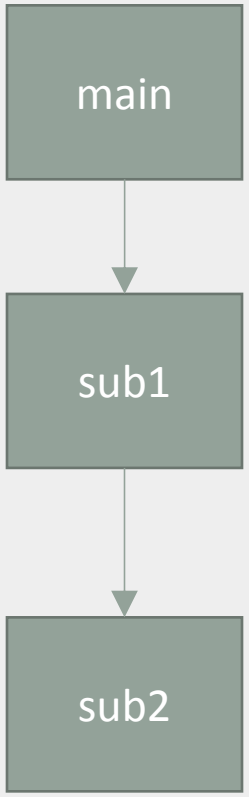


汇编语言程序设计

Assembly Language

模块化程序设计

```
1  assume cs:code
2  code segment
3  main: ...
4        call sub1
5        ...
6        mov ax, 4c00h
7        int 21h
8
9  sub1: ...
10       call sub2
11       ...
12       ret
13
14 sub2: ...
15       ...
16       ret
17 code ends
18 end main
```



- 调用子程序：call指令
- 返回：ret 指令
- 子程序：根据提供的参数处理一定的事务，处理后，将结果（返回值）提供给调用者。

参数和结果传递的问题

- 问题：根据提供的N，计算N的3次方。
- 考虑
  - （1）我们将参数N存储在什么地方？
  - （2）计算得到的数值，存储在什么地方？
- 方案
  - 用寄存器传递参数
  - 用内存单元进行参数传递
  - 用栈传递参数

```
1  assume cs:code
2  code segment
3  main: ...
4        call sub1
5        ...
6        mov ax, 4c00h
7        int 21h
8
9  sub1: ...
10       call sub2
11       ...
12       ret
13
14 sub2: ...
15       ...
16       ret
17 code ends
18 end main
```

```
#include <stdio.h>
int cube(int x);
int main()
{
   printf("%d\n",cube(2));
   return 0;
}
int cube(int x)
{
   int f;
   f=x*x;
   f=f*x;
   return f;
}
```

用寄存器来存储参数和结果是最常使用的方法

- 问题：根据提供的N，计算N的3次方。
- 考虑
  - （1）将参数N存储在什么地方？
  - （2）计算得到的数值，存储在什么地方？
- 用寄存器传递参数
  - 参数放到 bx 中，即(bx)=N
  - 子程序中用多个 mul 指令计算 N^3
  - 将结果放到 dx 和 ax中：(dx:ax)=N^3

编程任务：计算data段中第一组数据的 3 次方，
结果保存在后面一组dword单元中。

```
assume cs:code
data segment
    dw 1,2,3,4,5,6,7,8
    dd 0,0,0,0,0,0,0,0
data ends
code segment
start:mov ax,data
    mov ds,ax
    mov si,0
    mov di,16
    ;循环处理
    mov ax,4c00h
    int 21h
    code ends
end start
```

```
    mov cx,8
s: mov bx,[si]
    call cube
    mov [di],ax
    mov [di].2,dx
    add si,2
    add di,4
    loop s
```

```
cube: mov ax,bx
      mul bx
      mul bx
      ret
```

```
; 汇编子程序
cube: mov ax,bx
      mul bx
      mul bx
      ret
```

如果需要传递的数据有3个、4个或更多，寄存器不够了，怎么办？

```
C:\>debug p10-5.exe
-g

Program terminated normally
-d 076a:0 2f
076A:0000  01 00 02 00 03 00 04 00-05 00 06 00 07 00 08 00
076A:0010  01 00 00 00 08 00 00 00-1B 00 00 00 40 00 00 00
076A:0020  7D 00 00 00 D8 00 00 00-57 01 00 00 00 02 00 00
```

用内存单元批量传递数据

- 方案
  - 将批量数据放到内存中，然后将它们所在内存空间的首地址放在寄存器中，传递给需要的子程序。
  - 对于具有批量数据的返回结果，也可用同样的方法。
- 编程：将data段中的字符串转化为大写。

```
assume cs:code
data segment
    db 'conversation'
data ends
code segment
  start: ……
code ends
end start
```

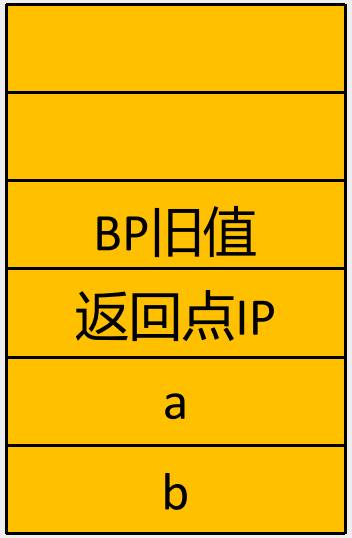
```
C:\>debug p10-6.exe
-r
AX=FFFF  BX=0000  CX=002A  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=0769  CS=076B  IP=0000   NV UP EI PL NZ NA PO NC
076B:0000 B86A07        MOV     AX,076A
-g

Program terminated normally
-d 076a:0 f
076A:0000  43 4F 4E 56 45 52 53 41-54 49 4F 4E 00 00 00 00   CONVERSATION....
-
```

```
code segment
start:  mov ax,data
        mov ds,ax
        mov si,0
        mov cx,12
        call capital
        mov ax,4c00h
        int 21h
capital: and byte ptr [si],11011111b
        inc si
        loop capital
        ret
code ends
```

用栈传递参数

- 原理：由调用者将需要传递给子程序的参数压入栈中，子程序从栈中取得参数
- 任务：计算(a – b) ^ 3 ，a、b 为 word 型数据。
  - 进入子程序前，参数a、b入栈
  - 调用子程序，将使栈顶存放IP
  - 结果：(dx : ax) = (a – b) ^ 3
- 例：设 a = 3 、b = 1 ，计算：(a – b) ^ 3

```
mov  ax , 1
push  ax
mov  ax , 3
push  ax
call  difcube
```



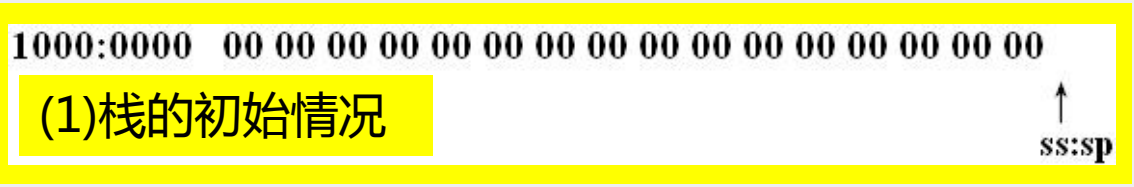
指令 ret n 的含义

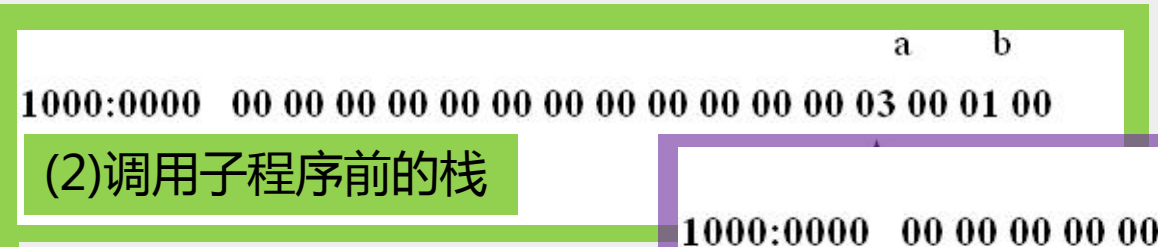
```
pop  ip
add  sp , n
```

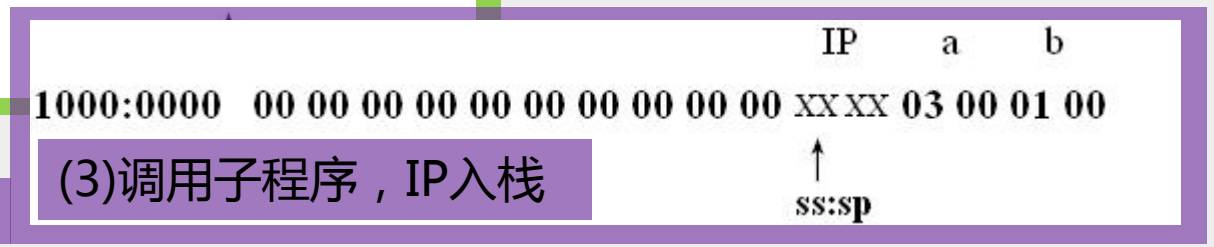
```
difcube : push  bp
      mov  bp , sp
      mov  ax , [ bp + 4 ] ; 将栈中a的值送入ax 中
      sub   ax , [ bp + 6 ] ; 减栈中b的值
      mov  bp , ax
      mul  bp
      mul  bp
      pop  bp
      ret  4
```

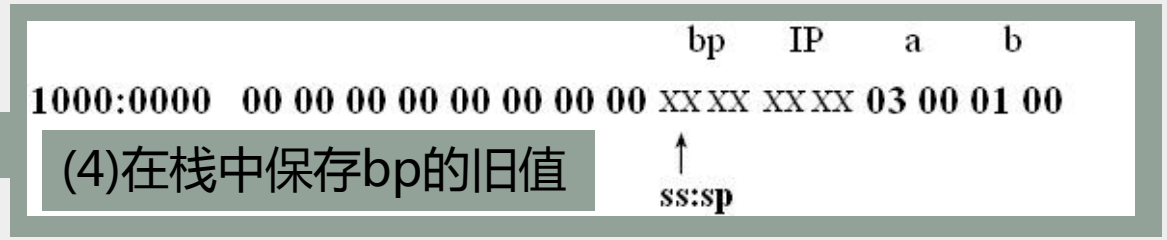
程序的执行过程中栈的变化

```
code segment
start:  mov  ax , 1
        push  ax
        mov  ax , 3
        push  ax
        call  difcube
        mov ax,4c00h
        int 21h
difcube : push  bp
        mov  bp , sp
        mov  ax , [ bp + 4 ]
        sub   ax , [ bp + 6 ]
        mov  bp , ax
        mul  bp
        mul  bp
        pop  bp
        ret  4
code ends
```



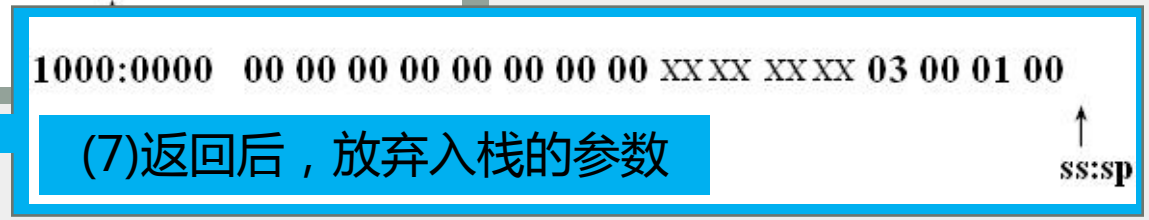






重要技术：子程序要用bp，为避免丢失有用数据，先入栈，返回前出栈。

(5) 从栈中获得参数并计算
计算结果(返回值)在dx和ax中

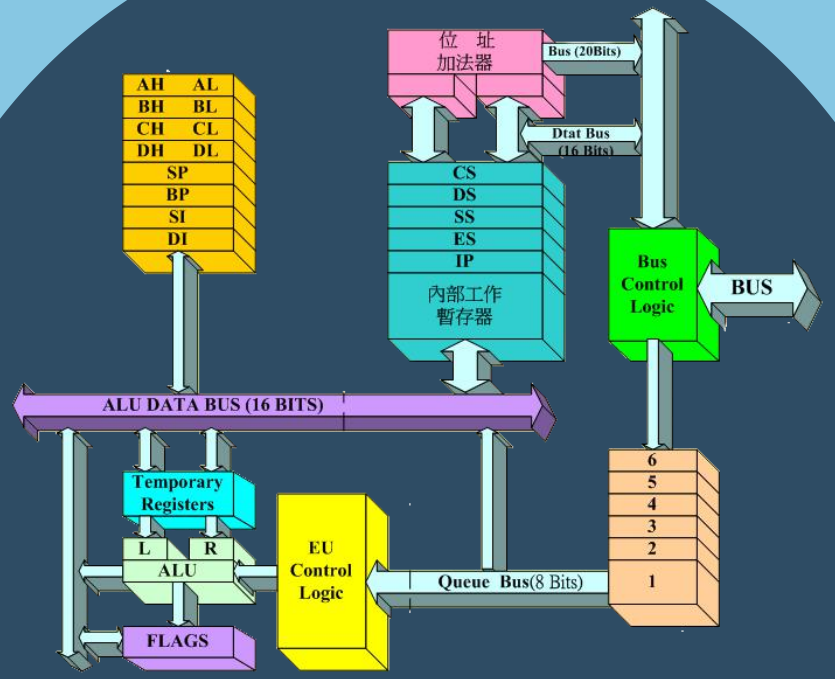


小结：参数和结果传递的问题

- 问题：根据提供的N，计算N的3次方。
- 考虑
  - （1）我们将参数N存储在什么地方？
  - （2）计算得到的数值，存储在什么地方？
- 方案
  - 用寄存器传递参数
  - 用内存单元进行参数传递
  - 用栈传递参数

寄存器冲突问题

贺利坚　主讲



汇编语言程序设计

Assembly Language

引子

问题：编程将data段中的字符串转化为大写。

```
assume cs:code
data segment
    db 'conversation'
data ends
code segment
  start: ……
code ends
end start
```

db ‘conversation’,0

```
        mov ax,data
        mov ds,ax
        mov si,0
        mov cx,12
        call capital
        mov ax,4c00h
        int 21h
capital: and byte ptr [si],11011111b
        inc si
        loop capital
        ret
```

代码：编程将data段中的字符串转化为大写

```
assume cs:code
data segment
        db 'conversation',0
 data ends
code segment
start:  mov ax,data
        mov ds,ax
        mov si,0
        call capital
        mov ax,4c00h
        int 21h
capital: mov cl, [si]
           mov ch, 0
           jcxz ok
           and byte ptr [si], 11011111b
           inc si
           jmp short capital
      ok:  ret
code ends
end start
```

;设置字符串的起始地址，并调用子程序

实用性更好的子程序。

子程序依次读取每个字符进行检测，如果不是0，进行大写的转化，如果是0，结束处理。
——不再需要字符串的长度作为参数。

cx既用于循环，又用于读取数据——冲突！

```
; 再例将以下字符串转为大写
assume cs:code
data segment
        db 'word',0
        db 'unix',0
        db 'wind',0
        db 'good',0
data ends
```

```
code segment
start:  mov ax,data
        mov ds,ax
        mov bx,0
        mov cx,4
      s: mov si,bx
        call capital
        add bx,5
        loop s
        mov ax,4c00h
        int 21h
capital: mov cl,[si]
         mov ch,0
         jcxz ok
         and byte ptr [si],11011111b
         inc si
         jmp short capital
     ok: ret
code ends
end start
```

寄存器冲突问题的解决

- 两个可能方案
  - （1）在编写调用子程序的程序时 ，注意看看子程序中有没有用到会产生冲突的寄存器
    - 如果有，调用者使用别的寄存器；
  - （2）在编写子程序的时候，不要使用会产生冲突的寄存器。
- 我们希望
  - （1）编写调用了程序的程序的时候不必关心子程序到底使用了哪些寄存器；
  - （2）编写子程序的时候不必关心调用者使用了哪些寄存器；
  - （3）不会发生寄存器冲突。

调用子程序的程序会很麻烦，必须要小心检查所调用的子程序中是否有将产生冲突的寄存器。

要调用子程序，必须看到子程序源码！？

子程序应该是独立的，编写子程序的时候无法知道也不必知道将来的调用情况。

子程序标准框架：

子程序开始：子程序中使用的寄存器入栈

子程序内容

子程序使用的寄存器出栈

返回（ret、retf）

可行的解决方案：在子程序的开始，将要用到的所有寄存器中的内容都保存起来，在子程序返回前再恢复。

寄存器冲突问题的解决示例

子程序标准框架：

子程序开始：子程序中使用的寄存器入栈

子程序内容

子程序使用的寄存器出栈

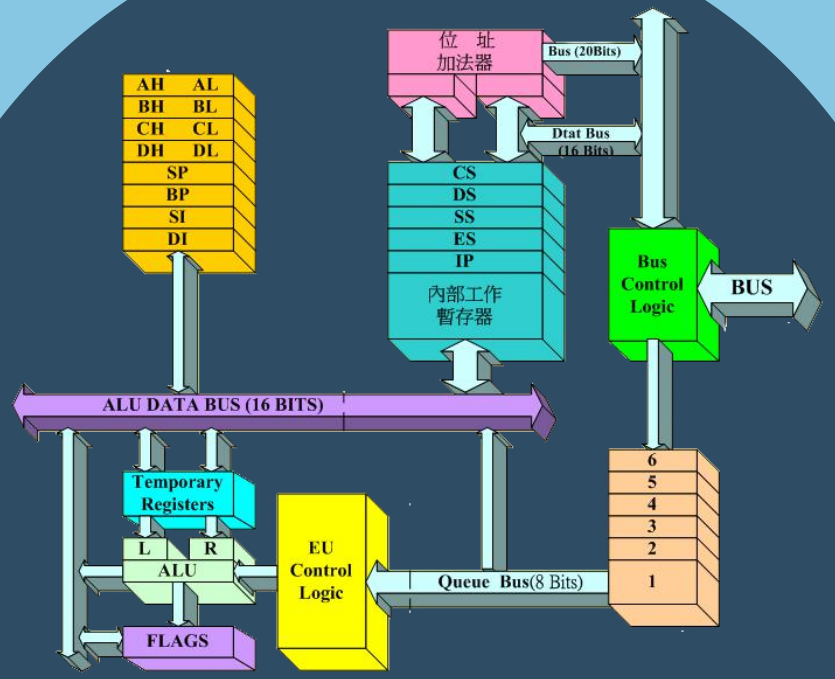
返回（ret、retf）

```
capital: push cx
         push si
change: mov cl,[si]
         mov ch,0
         jcxz ok
        and byte ptr [si],11011111b
        inc si
        jmp short change
    ok: pop si
        pop cx
        ret
```

```
assume cs:code
data segment
        db 'word',0
        db 'unix',0
        db 'wind',0
        db 'good',0
data ends

code segment
  start: mov ax,data
         mov ds,ax
         mov bx,0
         mov cx,4
       s: mov si,bx
         call capital
         add bx,5
         loop s
         mov ax,4c00h
         int 21h
         ; 子程序
code ends
end start
```

标志寄存器

贺利坚　主讲



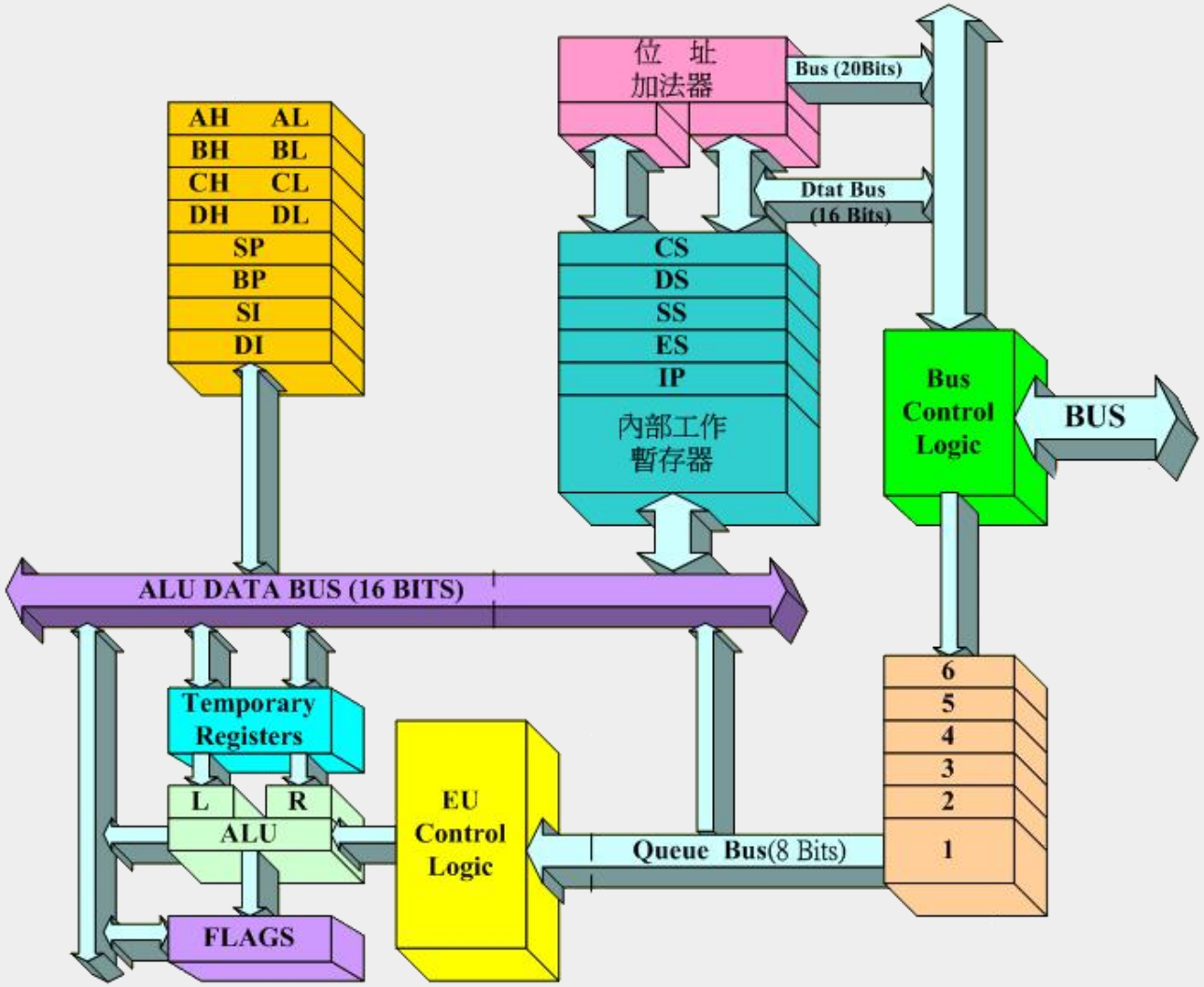
汇编语言程序设计

Assembly Language

标志寄存器

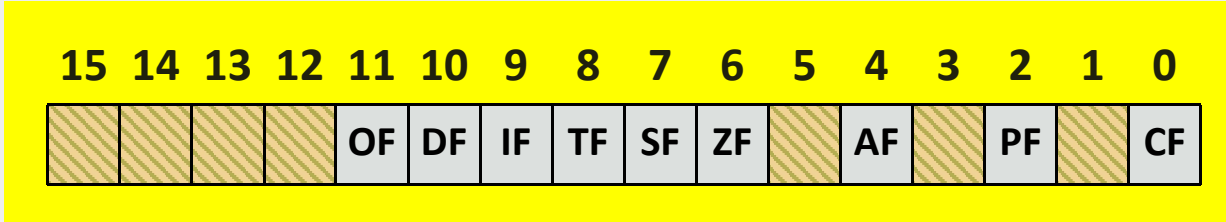
- 8086CPU有14个寄存器：
  - 通用寄存器：AX、 BX、 CX、 DX
  - 变址寄存器：SI、 DI
  - 指针寄存器：SP、 BP
  - 指令指针寄存器： IP
  - 段寄存器：CS、 SS、 DS、 ES
  - 标志(flag)寄存器：PSW/FLAGS

别称：程序状态字



认识标志寄存器的特殊之处

- 标志寄存器的结构
  - flag寄存器是按位起作用的，也就是说，它的每一位都有专门的含义，记录特定的信息。
  - 8086CPU中没有使用flag的1、3、5、12、13、14、15位，这些位不具有任何含义。
- 标志寄存器的作用
  - 用来存储相关指令的某些执行结果
  - 用来为CPU执行相关指令提供行为依据
  - 用来控制CPU的相关工作方式



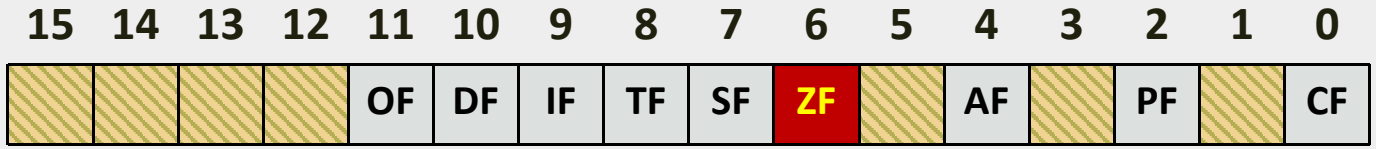
- 观察寄存器的值

```
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=**** ES=**** SS=**** CS=**** IP=0100  NV UP EI PL NZ NA PO NC
                                        ↑  ↑     ↑  ↑     ↑  ↑
                                        OF DF    SF ZF    PF CF
```

- 直接访问标志寄存器的方法
  - pushf ：将标志寄存器的值压栈；
  - popf ：从栈中弹出数据，送入标志寄存器中。

	标志	值为1	值为0	意义	
Overflow	OF	OV	NV	溢出	
Direction	DF	DN	UP	方向	
Sign	SF	NG	PL	符号	Positive /negative
Zero	ZF	ZR	NZ	零值	
Parity	PF	PE	PO	奇偶	odd/even
Carry	CF	CY	NC	进位	

ZF-零标志(Zero Flag)

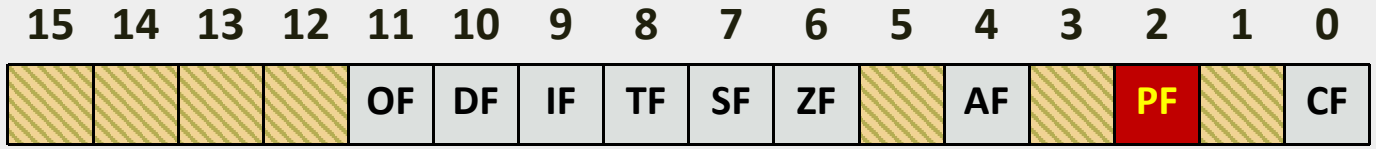


- ZF标记相关指令的计算结果是否为0
  - ZF=1，表示“结果是0”，1表示“逻辑真”
  - ZF=0，表示“结果不是0”，0表示“逻辑假”
- 示例

指令	执行结果
mov ax,1 and ax,0	ZF=1， 表示“结果是0”
mov ax,1 or ax,0	ZF=0， 表示“结果非0”

- 在8086CPU的指令集中，有的指令的执行是影响标志寄存器的，比如：add、sub、mul、div、inc、or、and等，它们大都是运算指令，进行逻辑或算术运算；
- 有的指令的执行对标志寄存器没有影响，比如：mov、push、pop等，它们大都是传送指令。
- 使用一条指令的时候，要注意这条指令的全部功能，其中包括执行结果对标记寄存器的哪些标志位造成影响。

PF-奇偶标志(Parity Flag)



- PF记录指令执行后，结果的所有二进制位中1的个数：
  - 1的个数为偶数，PF = 1；
  - 1的个数为奇数，PF = 0。
- 示例

指令	执行结果
mov al,1 add al,10	结果为0000 1011B = 0000 0001B + 0000 1010B 其中有3（奇数）个1，则PF=0；
mov al,1 or al,2	结果为00000011B = 0000 0001B or 0000 0010B 其中有2（偶数）个1，则PF=1；

SF-符号标志(Sign Flag)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

- SF记录指令执行后，将结果视为有符号数
  - 结果为负，SF = 1；
  - 结果为非负，SF = 0。
- 示例

指令	执行结果
mov al,10000001B add al,1	结果al 为10000010B，为负数，则SF=1；
sub ax, ax	结果ax为0，为非负数，故SF=0；

1000 0010B作为有符号数对应-111 1110B，即-126D

1000 0010B作为无符号数对应+1000 0010B，即+130D

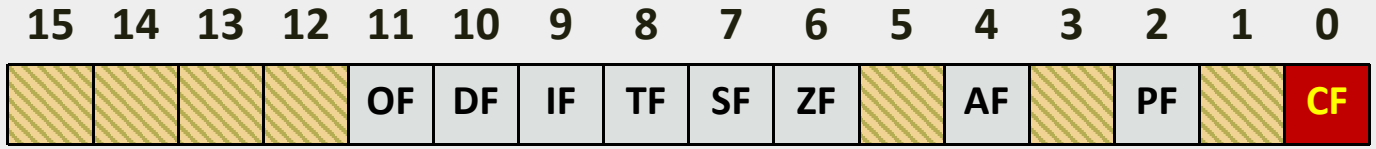
1000 0010B究竟算正数还是负数？

基础：有符号数与补码

- 计算机中有符号数一律用补码来表示和存储。
- 正整数的补码是其二进制表示，与原码相同
  - 例：+9的补码是00001001
- 负整数的补码，将其对应正数二进制的所有位取反（包括符号位，0变1，1变0）后加1
  - 例：-5的补码
    - -5对应正数5（00000101）→所有位取反（11111010）→加1(11111011)
    - 所以-5的补码是11111011。

SF 标志是CPU对有符号数运算结果的一种记录。
将数据当作有符号数来运算的时候，通过SF可知结果的正负；将数据当作无符号数来运算，SF的值则没有意义，虽然相关的指令影响了它的值。

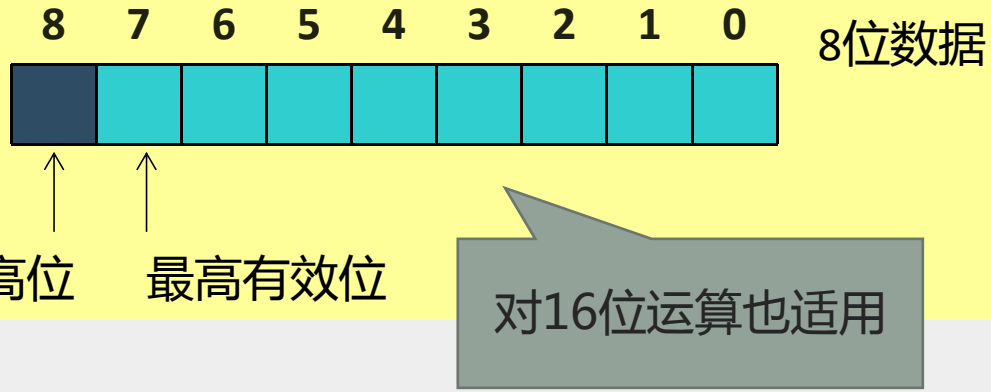
CF-进位标志(Carry Flag)



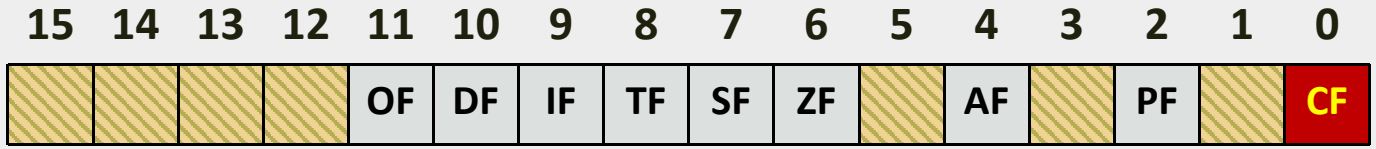
- 在进行无符号数运算的时候，CF记录了运算结果的最高有效位向更高位的进位值，或从更高位的借位值。
- CF记录指令执行后，
  - 有进位或借位，CF = 1
  - 无进位或借位，CF = 0
- 示例

指令	执行结果
mov al,98H add al,al	(al)=30H，CF=1，CF记录了最高有效位向更高位的进位值
add al,al	(al)=60H，CF=0，CF记录了最高有效位向更高位的进位值
sub al,98H	(al)=C8H，CF=1，CF记录了向更高位的借位值

- 对于位数为N的无符号数来说，其对应的二进制信息的最高位即第N-1位，是最高有效位
- 假想存在的第N位，就是相对最高有效位的更高位。



OF-溢出标志(Overflow Flag)



- 在进行有符号数运算的时候，如结果超过了机器所能表示的范围称为**溢出**。
- OF记录有符号数操作指令执行后，
  - 有溢出，OF = 1
  - 无溢出，OF = 0
- 示例

指令	执行结果
mov al,98 add al,99	(al)=197，超出了8位有符号数的范围(-128~127)，OF=1
mov al,0F0H add al,88H	(al)=(-16)+(-120)=-136，有溢出，OF=1

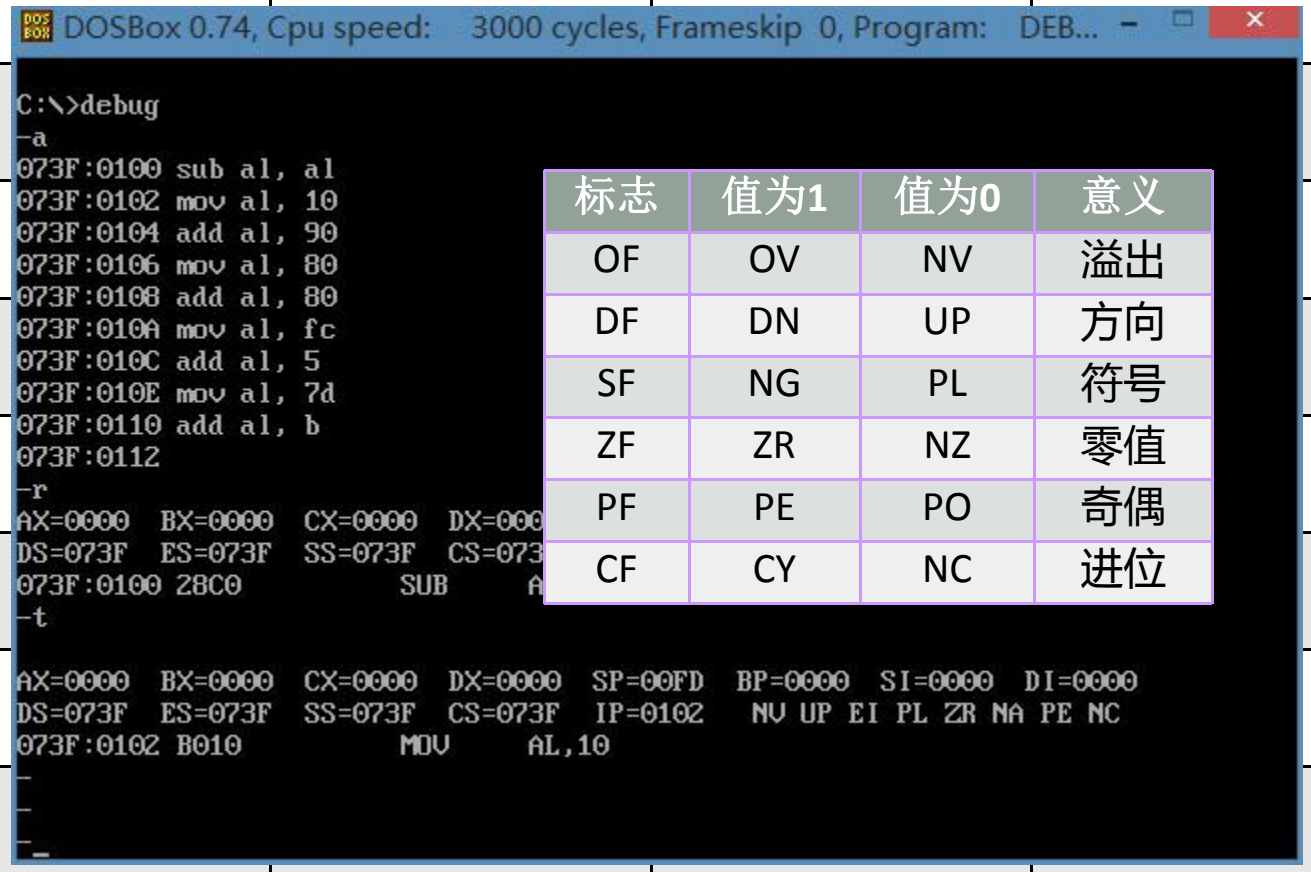
- 机器所能表达的范围
  - 以8位运算为例，结果用 8 位寄存器或内存单元来存放，机器所能表示的范围就是-128~127。
  - 同理，对于16 位有符号数，机器所能表示的范围是-32768~32767。
- 注意，此处溢出只是对有符号数运算而言。

- CF和OF的区别
  - CF是对无符号数运算有意义的进/借位标志位
  - OF是对有符号数运算有意义的溢出标志位
- 应用

指令	执行结果
mov al,0F0H add al,88H	CF=1, OF=1，当无符号数运算有进位，当有符号数运算有溢出

综合：一条指令会带来多个标志寄存器的变化

指令	CF	OF	SF	ZF	PF
sub al, al					
mov al, 10h					
add al, 90h					
mov al, 80h					
add al, 80h					
mov al, 0FCh					
add al, 05h					
mov al, 7Dh					
add al, 0Bh					



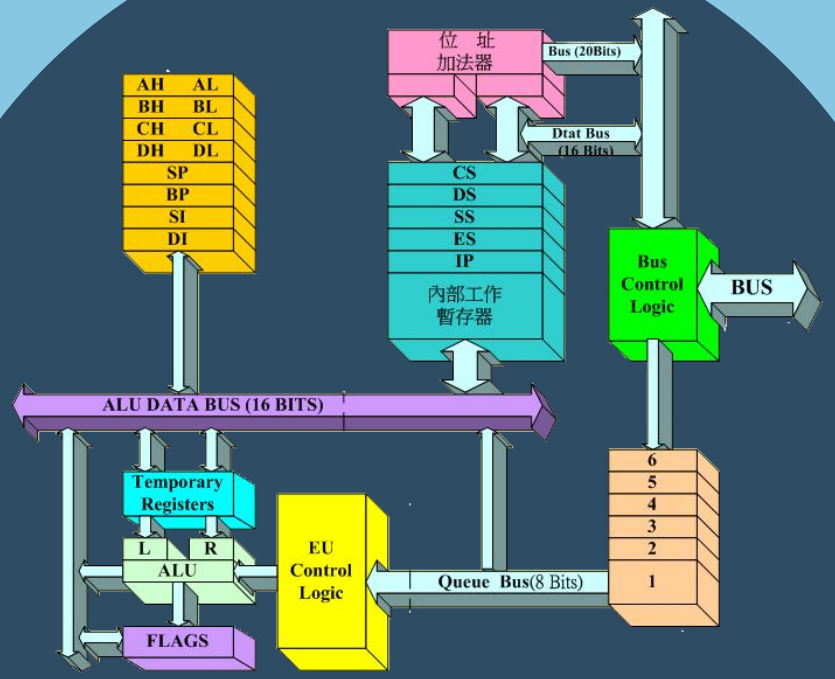
标志	值为1	值为0	意义
OF	OV	NV	溢出
DF	DN	UP	方向
SF	NG	PL	符号
ZF	ZR	NZ	零值
PF	PE	PO	奇偶
CF	CY	NC	进位

综合：一条指令会带来多个标志寄存器的变化

指令	CF	OF	SF	ZF	PF
sub al, al	0	0	0	1	1
mov al, 10h					
add al,90h					
mov al, 80h					
add al, 80h					
mov al, 0FCh					
add al, 05h					
mov al 7Dh					
add al, 0Bh					

带进(借)位的加减法

贺利坚　主讲



汇编语言程序设计

Assembly Language

adc-带进位加法指令

- adc是带进位加法指令 ，它利用了CF位上记录的进位值。
  - 格式：adc 操作对象1,操作对象2
  - 功能：操作对象1=操作对象1+操作对象2+CF
  - 例：adc ax,bx 实现的功能是：(ax)=(ax)+(bx)+CF
- 实例

指令	mov al,98H
	add al,al
	adc al,3
		mov ax,1
		add ax,ax
		adc ax,3
			mov ax,2
			mov bx,1
			sub bx,ax
			adc ax,1

结果	(ax)=34H	(ax)=5	(ax)=4
解释	adc执行时，相当于计算： (ax)+3+CF=30H+3+1=34H	adc执行时，相当于计算： (ax)+3+CF=2+3+0=5	adc执行时，相当于计算： (ax)+1+CF=2+1+1=4

adc指令应用：大数相加

- 问题：8086指令提供add指令，完成8位或16位加法，有更大的数相加时，如何做？
  - 32位、64位、24位？

- 例：编程计算1EF000H+201000H，结果放在ax（高16位）和bx（低16位）中
- 解决思路：先将低16位相加，然后将高16 位和进位值相加
- 程序：

```
mov ax,001EH
mov bx,0F000H
add bx,1000H
adc ax,0020H
```

```
      001E F000H
+)    0020 1000H
-------------------
             ???
```

- 例：计算 1E F000 1000H+20 1000 1EF0H，结果放在ax（高16位），bx（次高16位），cx（低16位）中。
- 解决思路：……
- 程序：

```
mov ax,001EH
mov bx,0F000H
mov cx,1000H
add cx,1EF0H
adc bx,1000H
adc ax,0020H
```

```
      001E F000 1000H
+)    0020 1000 1EF0H
-----------------------
                   ???
```

128位数据的相加

- 问题：编写一个子程序，对两个128位数据进行相加。
  - 名称：add128
  - 功能：两个逆序存放的128位数据进行相加

```
data segment
  dw 0A452H,0A8F5H,78E6H,0A8EH,8B7AH,54F6H,0F04H,671EH
  dw 0E71EH,0EF04H,54F6H,8B7AH,0A8EH,78E6H,58F5H,0452H
data ends
```

数据为128位，需要8个字单元，由低地址单元到高地址单元，依次存放由低到高的各个字。

- 分析
  - ds:si指向存储第一个数的内存空间
  - ds:di指向存储第二个数的内存空间
  - 运算结果存储在第一个数的存储空间中。

```
  671E 0F04 54F6 8B7A 0A8E 78E6 A8F5 A452H
+)0452 58F5 78E6 0A8E 8B7A 54F6 EF04 E71EH
 ------------------------------------------
                                      ???
```

```
code segment
start : mov ax,data
        mov ds,ax
        mov si,0
        mov di,16
        mov cx,8
        call add128
        mov ax,4c00h
        int 21h
add128:                    ;定义子程序
code ends
end start
```

```
;寄存器压栈
  push ax
  push cx
  push si
  push di
  sub ax,ax
s: mov ax,[si]
  adc ax,[di]
  mov [si],ax
  inc si
  inc si
  inc di
  inc di
  loop s
; 寄存器出栈
  pop di
  pop si
  pop cx
  pop ax
ret
```

讨论

sub ax, ax可否替换为mov ax, 0

两个inc di是否可以替换为add di, 2

sbb指令

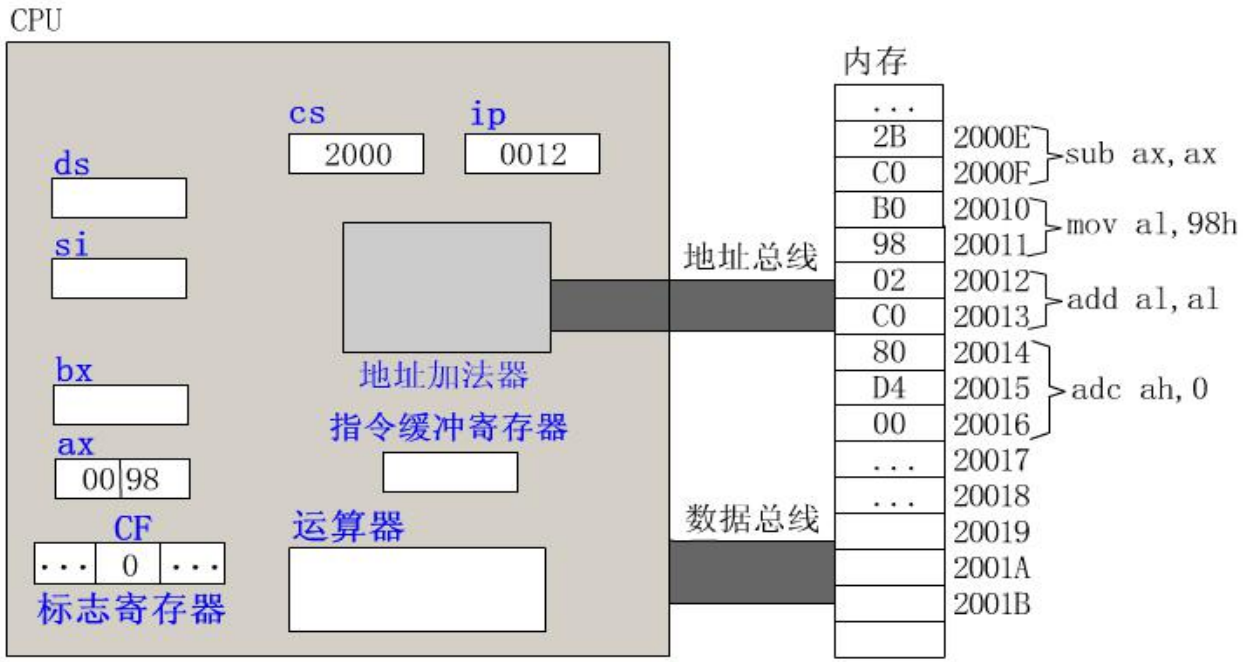
- sbb：带借位减法指令
  - 格式：sbb 操作对象1,操作对象2
  - 功能：
    操作对象1=操作对象1–操作对象2–CF
  - 与sub区别：利用CF位上记录的借位值
  - 比如：sbb ax,bx
  - 实现功能： (ax) = (ax) – (bx) – CF

- 应用：对任意大的数据进行减法运算
- 例如：计算003E1000H–00202000H
  结果放在ax，bx中
- 程序

```
mov bx,1000H
mov ax,003EH
sub bx,2000H
sbb ax,0020H
```

```
    003E 1000H
-)  0020 2000H
-------------
         ???
```

adc指令执行过程

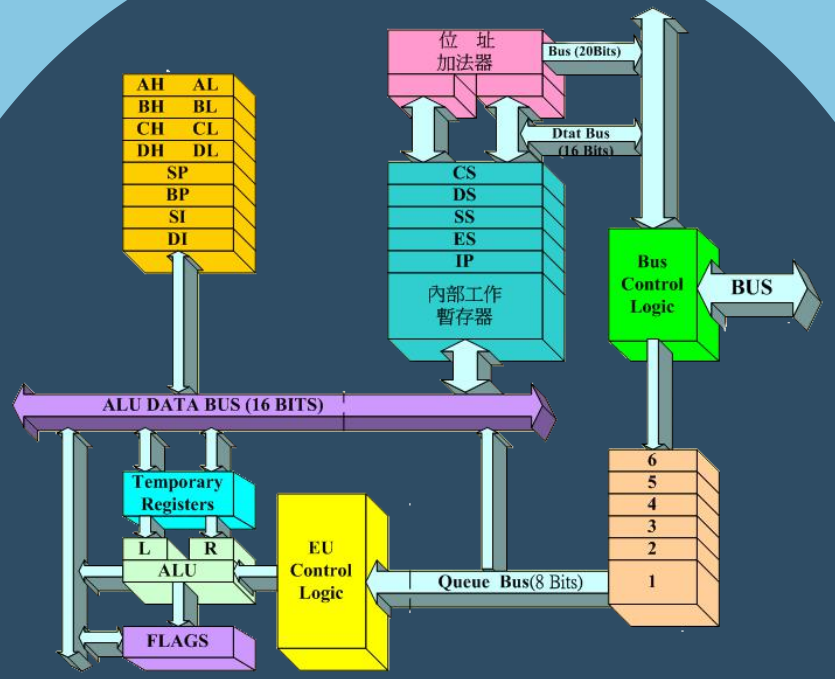


adc指令的执行过程

play step step stop

cmp与条件转移指令

贺利坚　主讲



汇编语言程序设计

Assembly Language

cmp指令

- cmp指令
  - 格式：cmp 操作对象1,操作对象2
  - 功能：计算操作对象1–操作对象2
- 应用
  - 其他相关指令通过识别这些被影响的标志寄存器位来得知比较结果。
- 例如

✓cmp 是比较指令，功能相当于减法指令，只是不保存结果。

✓cmp 指令执行后，将对标志寄存器产生影响。

应用方法：用标志寄存器值，确定比较结果。

指令	cmp ax,ax	mov ax,8
		mov bx,3
		cmp ax,bx

功能	做(ax)–(ax)的运算，结果为0，但并不在ax中保存，仅影响flag的相关各位。	(ax)=8, (bx)=3
标志寄存器	ZF=1 PF=1 SF=0 CF=0 OF=0	ZF=0 PF=1 SF=0 CF=0 OF=0

无符号数比较与标志位取值

- 思路：通过cmp 指令执行后相关标志位的值，可以看出比较的结果
- 指令：cmp ax,bx

比较关系	(ax) ？ (bx)	(ax) - (bx)特点	标志寄存器
等于	(ax) = (bx)	(ax) - (bx) = 0	ZF = 1
不等于	(ax) ≠ (bx)	(ax) - (bx) ≠ 0	ZF = 0
小于	(ax) < (bx)	(ax) - (bx) 将产生借位	CF = 1
大于等于	(ax) ≥ (bx)	(ax) - (bx) 不必借位	CF = 0
大于	(ax) > (bx)	(ax) - (bx) 既不借位，结果又不为0	CF = 0且ZF = 0
小于等于	(ax) ≤ (bx)	(ax) - (bx) 或者借位，或者结果为0	CF = 1 或 ZF = 1

比较指令的设计思路，即：通过做减法运算影响标志寄存器，标志寄存器的相关位的取值，体现比较的结果。

有符号数比较与标志位取值

- 问题：用cmp来进行有符号数比较时，CPU用哪些标志位对比较结果进行记录？
- 示例指令：cmp ah,bh

比较关系	(ax) ？ (bx)	(ax) - (bx)特点	标志寄存器
等于	(ah) = (bh)	(ah) - (bh) = 0	ZF = 1
不等于	(ah) ≠ (bh)	(ah) - (bh) ≠ 0	ZF = 0
小于	(ax) < (bx)	(ax) - (bx) 为负，且不溢出	SF = 1且OF=0
大于	(ax) > (bx)	(ax) - (bx) 为负，且溢出	SF = 1且OF = 1
大于等于	(ax) ≥ (bx)	(ax) - (bx) 为非负，且无溢出	SF = 0且OF = 0
小于等于	(ax) < (bx)	(ax) - (bx) 为非负，且有溢出	SF = 0 或 OF = 1

仅凭结果正负（SF）无法得出结论，需要配合是否溢出（OF）得到结论。推导略。

条件转移指令

套路	cmp oper1, oper2 ;或者其他影响标志寄存器的指令 jxxx 标号

根据单个标志位转移的指令

指令	含义	测试条件
je/jz	相等/结果为0	ZF=1
jne/jnz	不等/结果不为0	ZF=0
js	结果为负	SF=1
jns	结果非负	SF=0
jo	结果溢出	OF=1
jno	结果溢出	OF=0
jp	奇偶位为1	PF=1
jnp	奇偶位不为1	PF=0
jb/jnae/jc	低于/不高于等于/有借位	CF=1
jnb/jae/jnc	不低于/高于等于/无借位	CF=0

根据无符号数比较结果进行转移的指令

指令	含义	测试条件
jb/jnae/jc	低于则转移	CF=1
jnb/jae/jnc	低于则转移	CF=0
jna/jbe	不高于则转移	CF=1或ZF=1
ja/jnbe	高于则转移	CF=0且ZF=0

根据有符号数比较结果进行转移的指令

指令	含义	测试条件
jl/jnge	小于则转移	SF = 1且OF=0
jnl/jge	不小于转移	SF = 0且OF = 0
jle/jng	小于等于则转移	SF = 0 或 OF = 1
jnle/jg	不小于等于则转移	SF = 1且OF = 1

```
j-Jump  e-Equal  n-Not  b-Below  a-Above  L-less  g-Greater
s-Sign  C-carry  p-Parity o-Overflow   z-Zero
```

条件转移指令的使用

- jxxx系列指令和cmp指令配合，构造条件转移指令
  - 不必再考虑cmp指令对相关标志位的影响和jxxx指令对相关标志位的检测
  - 可以直接考虑cmp和jxxx指令配合使用时表现出来的逻辑含义。
- jxxx系列指令和cmp指令配合实现高级语言中if语句的功能

- 例1：如果(ah)=(bh)，则(ah)=(ah)+(ah)，否则(ah)=(ah)+(bh)

```
    cmp ah,bh
    je s
    add ah,bh
    jmp short ok
 s: add ah,ah
ok: ret
```

```
if(a==b){
   a=a+a;
}
else{
   a=a+b;
}
```

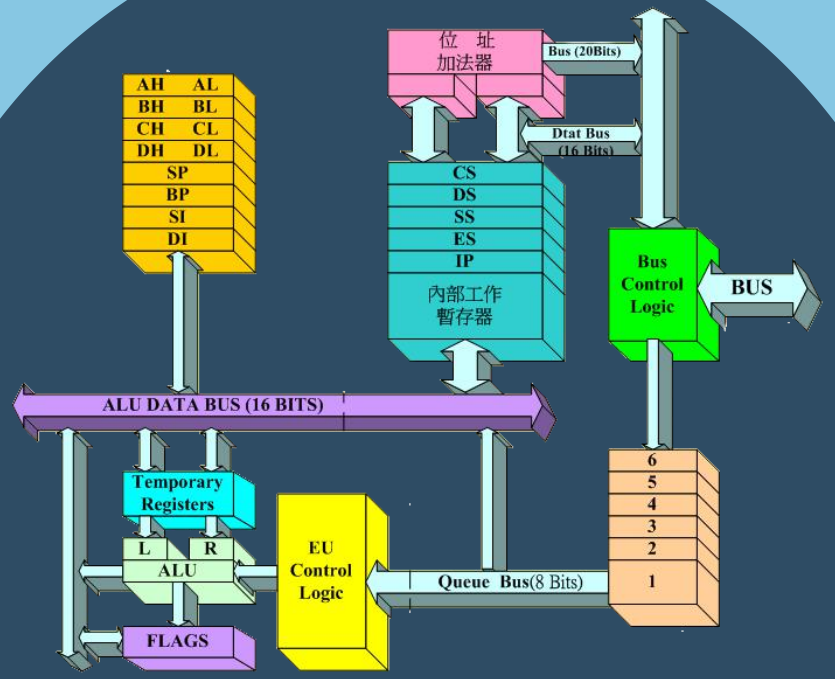
- 例2：如果(ax)=0，则(ax)=(ax)+1

```
    ...
    ; ax获得值
    add ax,0
    jnz s
    inc ax
s: ...
```

```
if(a==0)
{
   a++;
}
```

条件转移指令应用

贺利坚　主讲



汇编语言程序设计

Assembly Language

条件转移指令

- 条件转移指令：jxxx——je/jna/jae...
  - 可以根据某种“条件”，决定是否“转移”程序执行流程。
  - “转移” = 修改IP
- 如何检测条件？
  - 通过检测标志位，由标志位体现条件
  - 条件转移指令通常都和cmp相配合使用，cmp指令改变标志位
- 例：双分支结构的实现


```
    cmp ah,bh
    je s
    add ah,bh
    jmp short ok
 s: add ah,ah
ok: ret
```

```
if(a==b){
  a=a+a;
}
else{
  a=a+b;
}
```

应用示例

给出下面一组数据：

```
data segment
  db 8,11,8,1,8,5,63,38
data ends
```

请编程实现如下统计，用ax保存统计结果

（1）统计数值为8的字节的个数

（2）统计数值大于8的字节的个数

（3）统计数值小于8的字节的个数

（1）编程思路：初始设置(ax)=0，然后用循环依次比较每个字节的值，找到一个和8相等的数就将ax的值加1。

解法1

```
5  code segment
6  start:
7       mov ax,data
8       mov ds,ax
9       mov bx,0
10      mov ax,0
11      mov cx,8
12  s:  cmp byte ptr [bx],8
13      jne next
14      inc ax
15  next:
16      inc bx
17      loop s
18
19      mov ax,4c00h
20      int 21h
21 code ends
```

解法2

```
5  code segment
6  start:
7       mov ax,data
8       mov ds,ax
9       mov bx,0
10      mov ax,0
11      mov cx,8
12    s: cmp byte ptr [bx],8
13      je ok
14      jmp short next
15    ok: inc ax
16  next: inc bx
17      loop s
18
19     mov ax,4c00h
20     int 21h
21 code ends
```

应用示例

给出下面一组数据：

```
data segment
  db 8,11,8,1,8,5,63,38
data ends
```

请编程实现如下统计，用ax保存统计结果

（1）统计数值为8的字节的个数

（2）统计数值大于8的字节的个数

（3）统计数值小于8的字节的个数

（2）初始设置(ax)=0，然后用循环依次比较每个字节的值，找到一个大于8的数就将ax的值加1。

```
code segment
start:
        mov ax,data
        mov ds,ax
        mov bx,0
        mov ax,0
        mov cx,8
    s:  cmp byte ptr [bx],8
        jna next
        inc ax
next:   inc bx
        loop s

    mov ax,4c00h
    int 21h
code ends
```

应用示例

给出下面一组数据：

```
data segment
  db 8,11,8,1,8,5,63,38
data ends
```

请编程实现如下统计，用ax保存统计结果

（1）统计数值为8的字节的个数

（2）统计数值大于8的字节的个数

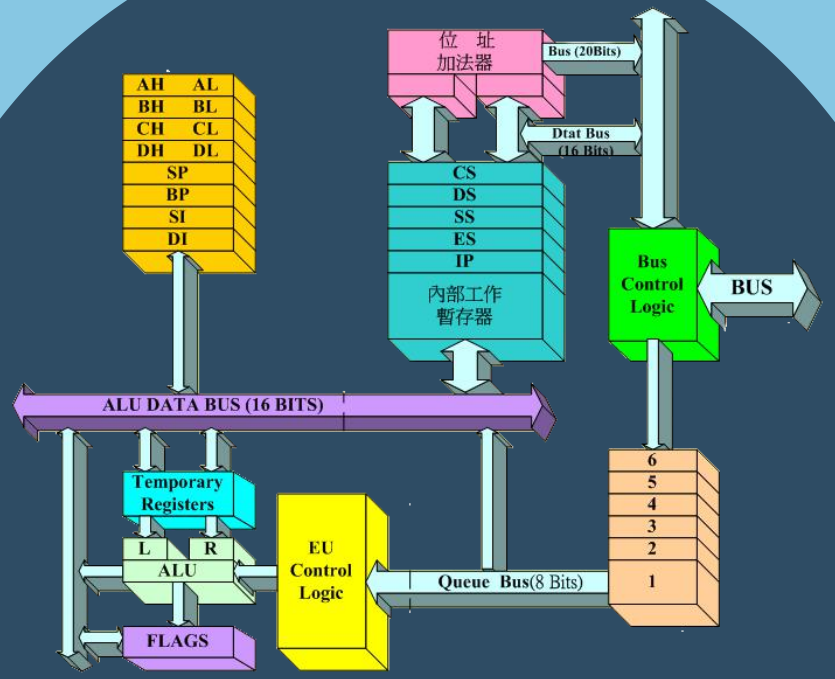
（3）统计数值小于8的字节的个数

（3）初始设置(ax)=0，然后用循环依次比较每个字节的值，找到一个小于8的数就将ax的值加1。

```
 5  code segment
 6  start:
 7          mov ax,data
 8          mov ds,ax
 9          mov bx,0
10          mov ax,0
11          mov cx,8
12      s:  cmp byte ptr [bx],8
13          jnb next
14          inc ax
15    next: inc bx
16          loop s
17
18          mov ax,4c00h
19          int 21h
20  code ends
```

DF标志和串传送指令

贺利坚　主讲



汇编语言程序设计

Assembly Language

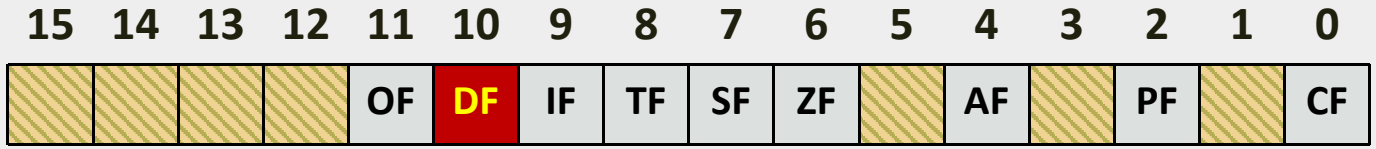
问题的提出

💻编程：将data段中的第一个字符串复制到它后面的空间中。

```
data segment
    db 'Welcome to masm!'
    db 16 dup (0)
data ends
```


```
code segment
start: mov ax,data
       mov ds,ax
       mov si,0
       mov di,16
       mov cx,8
    s: mov ax,[si]
       mov [di],ax
       add si,2
       add di,2
       loop s

       mov ax,4c00h
       int 21h
code ends
end start
```

DF标志和串传送指令



DF-方向标志位（Direction Flag）

- 功能
  - 在串处理指令中，控制每次操作后si，di的增减。
  - DF = 0：每次操作后si，di递增；
  - DF = 1：每次操作后si，di递减。
- 对DF位进行设置的指令：
  - cld指令：将标志寄存器的DF位设为0(clear)
  - std指令：将标志寄存器的DF位设为1(setup)

```
data segment
    db 'Welcome to masm!'
    db 16 dup (0)
data ends

code segment
start:
    ;设置寄存器
    mov ax,data
    mov ds,ax
    mov si,0
    mov es,ax
    mov di,16
    cld
    ;循环传送
    mov cx ,16
s:  movsb
    loop s
    mov ax,4c00h
    int 21h
code ends
```

串传送指令1： movsb

功能：（以字节为单位传送）

(1) ((es)×16 + (di)) = ((ds) ×16 + (si))

(2) 如果DF = 0则： (si) = (si) + 1

(di) = (di) + 1

如果DF = 1则：(si) = (si) - 1

(di) = (di) - 1

串传送指令2：movsw

功能：（以字为单位传送）

(1) ((es)×16 + (di)) = ((ds) ×16 + (si))

(2) 如果DF = 0则： (si) = (si) + 2

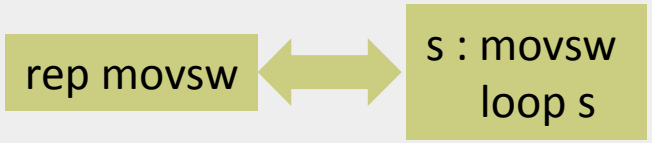
(di) = (di) + 2

如果DF = 1则：(si) = (si) - 2

(di) = (di) - 2

rep指令

- rep指令常和串传送指令搭配使用
- 功能：根据cx的值，重复执行后面的指令
- 用法：




```
1  assume cs:code, ds:data
2  data segment
3      db 'Welcome to masm!'
4      db 16 dup (0)
5  data ends
6  code segment
7  start:
8      mov ax,data
9      mov ds,ax
10     mov si,0
11     mov es,ax
12     mov di,16
13     cld
14     mov cx,8
15     rep movsw
16
17     mov ax,4c00h
18     int 21h
19 code ends
20 end start
```

应用实例

任务：用串传送指令，将F000H段中的最后16个字符复制到data段中。

data segment

db 16 dup (0)

data ends

F000H段的最后一个字符的位置：F000:FFFF

```
-d f000:fff0 ffff
F000:FFF0  EA C0 12 00 F0 30 31 2F-30 31 2F 39 32 00 FC 55   .....01/01/92..U
-d 076a:0 f
076A:0000  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-g 16

AX=076A  BX=0000  CX=0000  DX=0000  SP=0000  BP=0000  SI=FFEF  DI=FFFF
DS=F000  ES=076A  SS=0769  CS=076B  IP=0016   NV DN EI PL NZ NA PO NC
076B:0016 B8004C        MOV     AX,4C00
-d 076a:0 f
076A:0000  EA C0 12 00 F0 30 31 2F-30 31 2F 39 32 00 FC 55   .....01/01/92..U
-
```

```
1  assume cs:code, ds:data
2  data segment
3      db 16 dup (0)
4  data ends
5  code segment
6  start:
7      mov ax,0f000h
8      mov ds,ax
9      mov si,0ffffh
10     mov ax,data
11     mov es,ax
12     mov di,15
13     mov cx,16
14     std
15     rep movsb
16
17     mov ax,4c00h
18     int 21h
19 code ends
20 end start
```