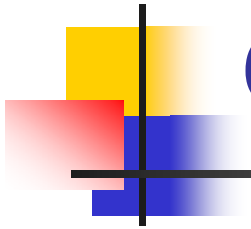
段寄存器

- 段寄存器就是提供段地址的
  8086CPU有4个段寄存器：
  CS、DS、SS、ES
  （第24页）

CS：代码段（Code Segment）
DS：数据段（Data Segment）
SS：堆栈段（Stack Segment）
ES：附加段（Extra Segment）

- 当**8086CPU**要访问内存时，由这**4**个段寄存器提供内存单元的段地址



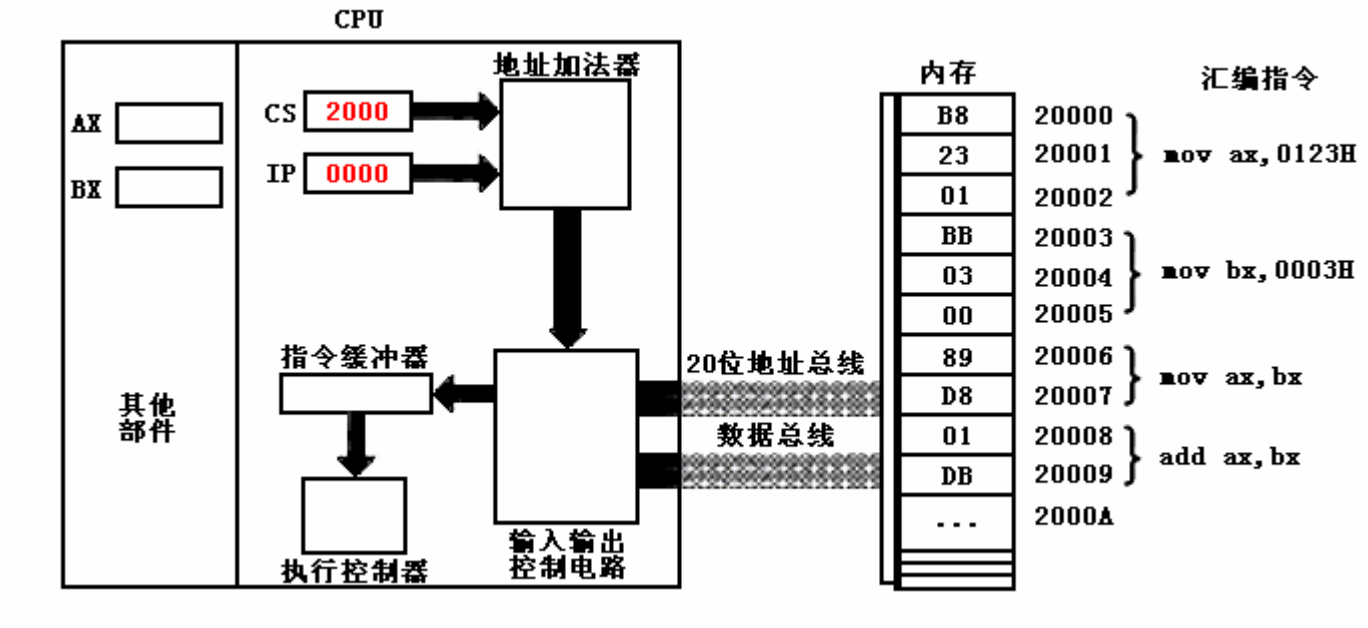
CS和IP

- CS和IP是8086CPU中最关键的寄存器，它们指示了CPU当前要读取指令的地址。

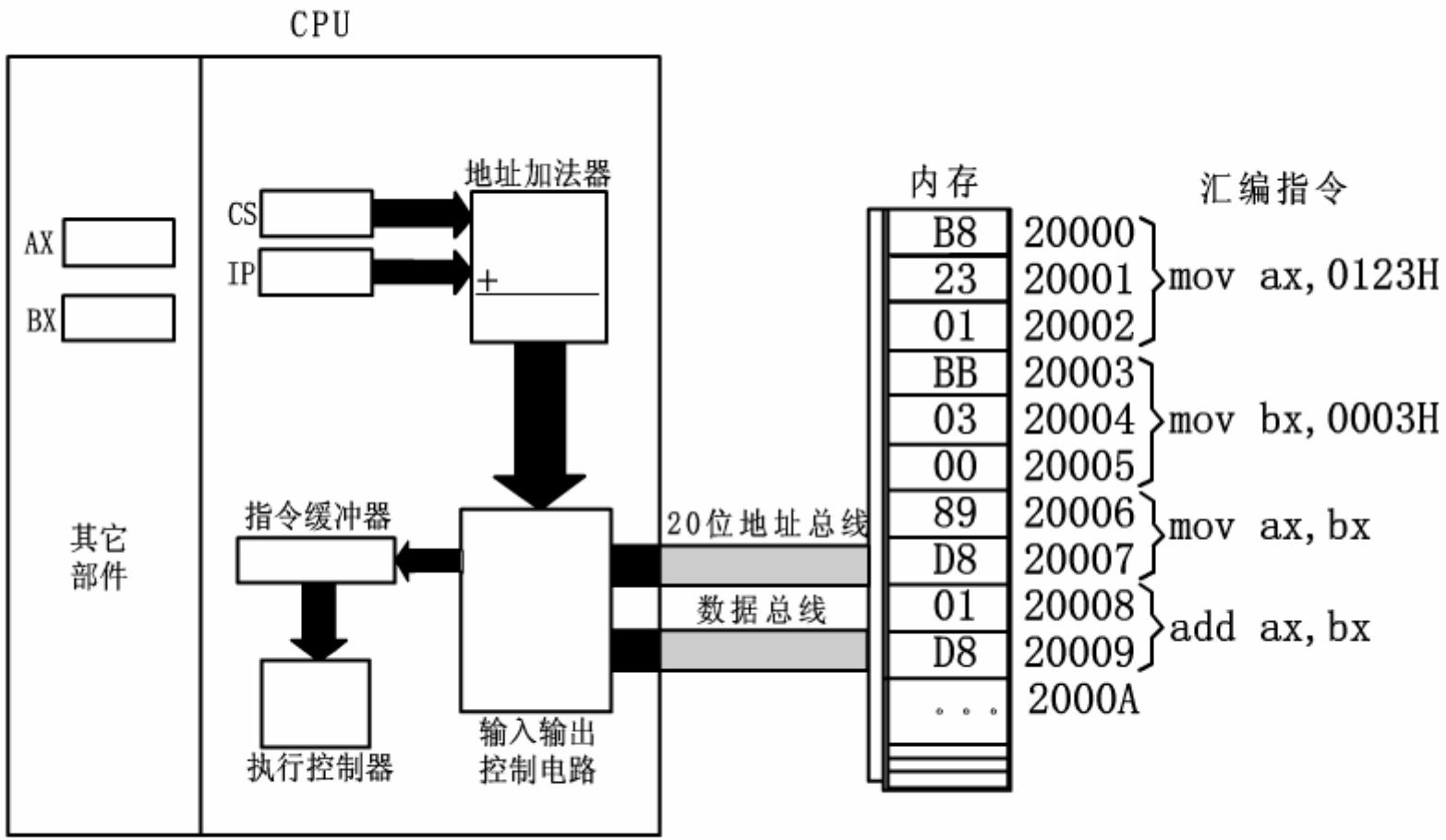
CS为代码段寄存器；

IP为指令指针寄存器。

8086PC读取和执行指令相关部件



- 8086PC读取和执行指令演示
- 8086PC工作过程的简要描述



play step step stop

8086PC工作过程的简要描述

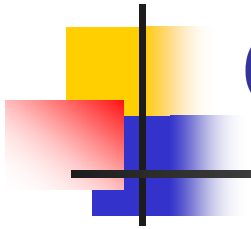
（1）从CS:IP指向内存单元读取指令，读取的指令进入指令缓冲器；

（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。

8086PC工作过程的简要描述

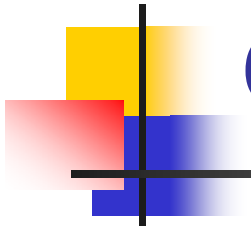
- 在 **8086CPU** 加电启动或复位后（即**CPU**刚开始工作时）**CS**和**IP**被设置为**CS=FFFFH**，**IP=0000H**，即在**8086PC**机刚启动时，**CPU**从内存**FFFF0H**单元中读取指令执行，**FFFF0H**单元中的指令是**8086PC**机开机后执行的第一条指令。



CS和IP

- 内存中指令和数据没有任何区别，都是二进制信息，CPU在工作的时候把有的信息看作指令，有的信息看作数据。
- CPU根据什么将内存中的信息看作指令？

  CPU将CS:IP指向的内存单元中的内容看作指令



CS和IP

- 在任何时候，**CPU**将**CS**、**IP**中的内容当作指令的段地址和偏移地址，用它们合成指令的物理地址，到内存中读取指令码，执行。
- 如果说，内存中的一段信息曾被**CPU**执行过的话，那么，它所在的内存单元必然被**CS:IP**指向过。

修改CS、IP的指令

- 在CPU中，程序员能够用指令读写的部件只有寄存器，程序员可以通过改变寄存器中的内容实现对CPU的控制。
- CPU从何处执行指令是由CS、IP中的内容决定的，程序员可以通过改变CS、IP中的内容来控制CPU执行目标指令。
- 我们如何改变CS、IP的值呢？

修改CS、IP的指令

- 8086CPU必须提供相应的指令
- 先回想我们如何修改AX中的值？
- mov指令不能用于设置CS、IP的值，8086CPU没有提供这样的功能。
- 8086CPU为CS、IP提供了另外的指令来改变它们的值：转移指令

如何修改AX中的值？

- **mov 指令**
- **如： mov ax, 123**
- **mov指令可以改变8086CPU大部分寄存器的值，被称为传送指令。**
- **能够通过mov 指令改变CS、IP的值吗？**

修改CS、IP的指令

- 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。

修改CS、IP的指令

- 仅修改IP的内容：

  jmp 某一合法寄存器

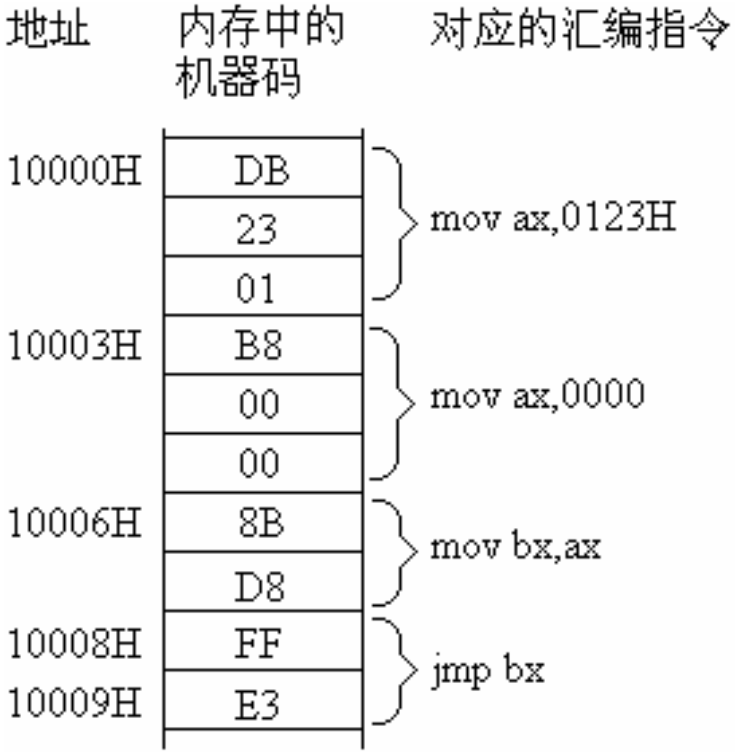
  jmp ax （类似于 **mov IP,ax**）

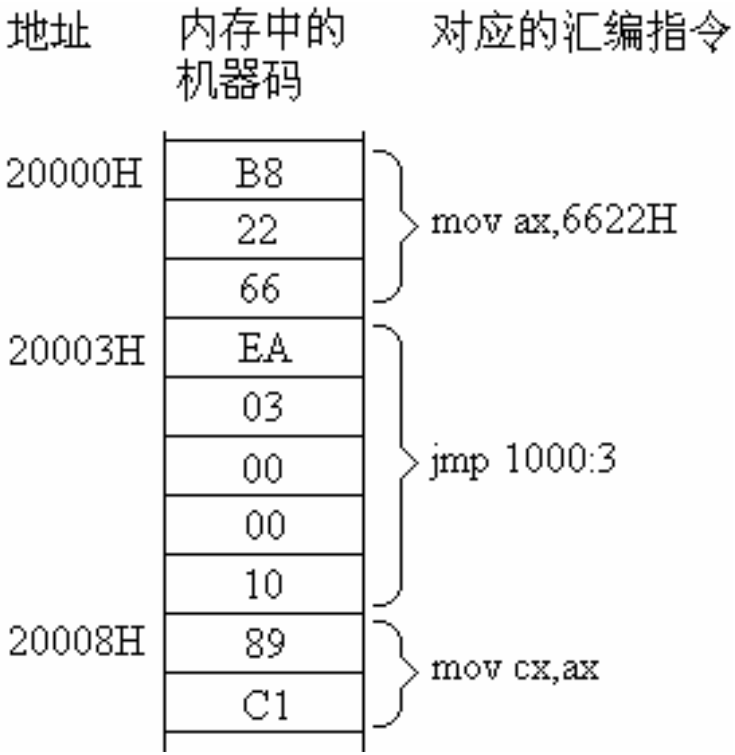
  jmp bx

  功能：用寄存器中的值修改IP。

问题分析

- 内存中存放的机器码和对应汇编指令情况：（初始：**CS=2000H，IP=0000H**）
- 请写出指令执行序列：





问题分析结果：

- （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

代码段

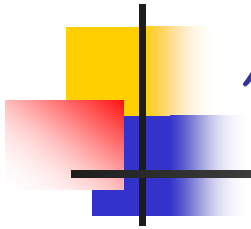
- 对于**8086PC**机，在编程时，可以根据需要，将一组内存单元定义为一个段。
- 可以将长度为 **N**（ **N**≤64KB ）的一组代码，存在一组地址连续、起始地址为 **16**的倍数的内存单元中，这段内存是用来存放代码的，从而定义了一个代码段。

- 例如

代码段

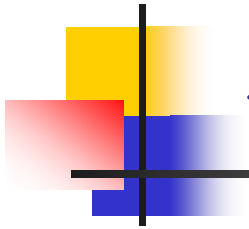
```
-a
0B07:0100   mov ax,0000
0B07:0103   add ax,0123
0B07:0106   mov bx,ax
0B07:0108   jmp bx
-u
0B07:0100    B80000       MOV AX,0000
0B07:0103    052301       ADD AX,0123
0B07:0106    89C3         MOV BX,AX
0B07:0108    FFE3         JMP BX
```

- 这段长度为 **10** 字节的字节的指令，存在从**0B07:0100~0B07:0108**的一组内存单元中，我们就可以认为，**0B170H~0B178H**这段内存单元是用来存放代码的，是一个代码段，它的段地址为**0B07H**，长度为**10**字节。



代码段

- 如何使得代码段中的指令被执行呢？
- 将一段内存当作代码段，仅仅是我们在编程时的一种安排，**CPU** 并不会由于这种安排，就自动地将我们定义得代码段中的指令当作指令来执行。
- **CPU** 只认被 **CS:IP** 指向的内存单元中的内容为指令。
- 所以要将**CS:IP**指向所定义的代码段中的第一条指令的首地址。
- **CS = 0B170H，IP = 0000H。**

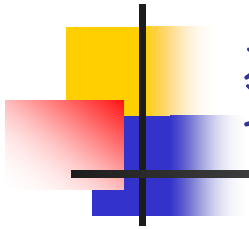


小结

- 1、段地址在8086CPU的寄存器中存放。当8086CPU要访问内存时，由段寄存器提供内存单元的段地址。8086CPU有4个段寄存器，其中CS用来存放指令的段地址。
- 2、CS存放指令的段地址，IP存放指令的偏移地址。
- 3、8086中，任意时刻，CPU将CS:IP指向的内容当作指令执行。

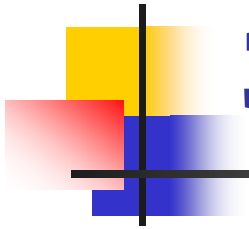
小结（续）

- 3、8086CPU的工作过程：
  - （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指向下一条指令；
  - （3）执行指令。（转到步骤（1），重复这个过程。）
- 4、8086CPU提供转移指令修改CS、IP的内容。



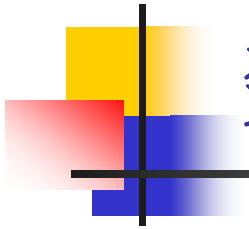
数据段

- 数据段的定义
  - 可以根据需要将一组内存单元定义为一个段
  - 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
- 比如我们用123B0H~123B9H这段空间来存放数据：
  - 段地址：123BH
  - 长度：10字节



3.5 数据段

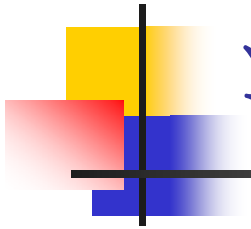
- 如何访问数据段中的数据呢？
  - 用 **ds** 存放数据段的段地址
  - 用相关指令访问数据段中的具体单元。



数据段

- 例如：将**123B0H~123BAH**的内存单元定义为数据段，代码如下：

```
mov ax, 123BH
mov ds, ax
```



汇编源程序

- 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。

定义一个段

- **segment和ends**
  - 功能：定义一个段
  - **segment**：说明一个段开始
  - **ends**：说明一个段结束。
- 一个段必须有一个名称来标识，格式：

  段名 **segment**

  段名 **ends**
- **segment和ends**是成对使用的伪指令，写汇编程序时，必须要用到的一对伪指令。

程序结束标记

- **end** 是一个汇编程序的结束标记，结束对源程序的编译
  - 如果程序写完了，要在结尾处加上伪指令**end** 。否则，编译器在编译程序时，无法知道程序在何处结束。
- 注意：不要搞混了**end**和**ends**。

寄存器与段的关联假设

- **assume**：含义为“假设”。
- 它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。
- 通过**assume**说明这种关联，在需要的情况下，编译程序可以将段寄存器和某一个具体的段相联系。

4.2 源程序

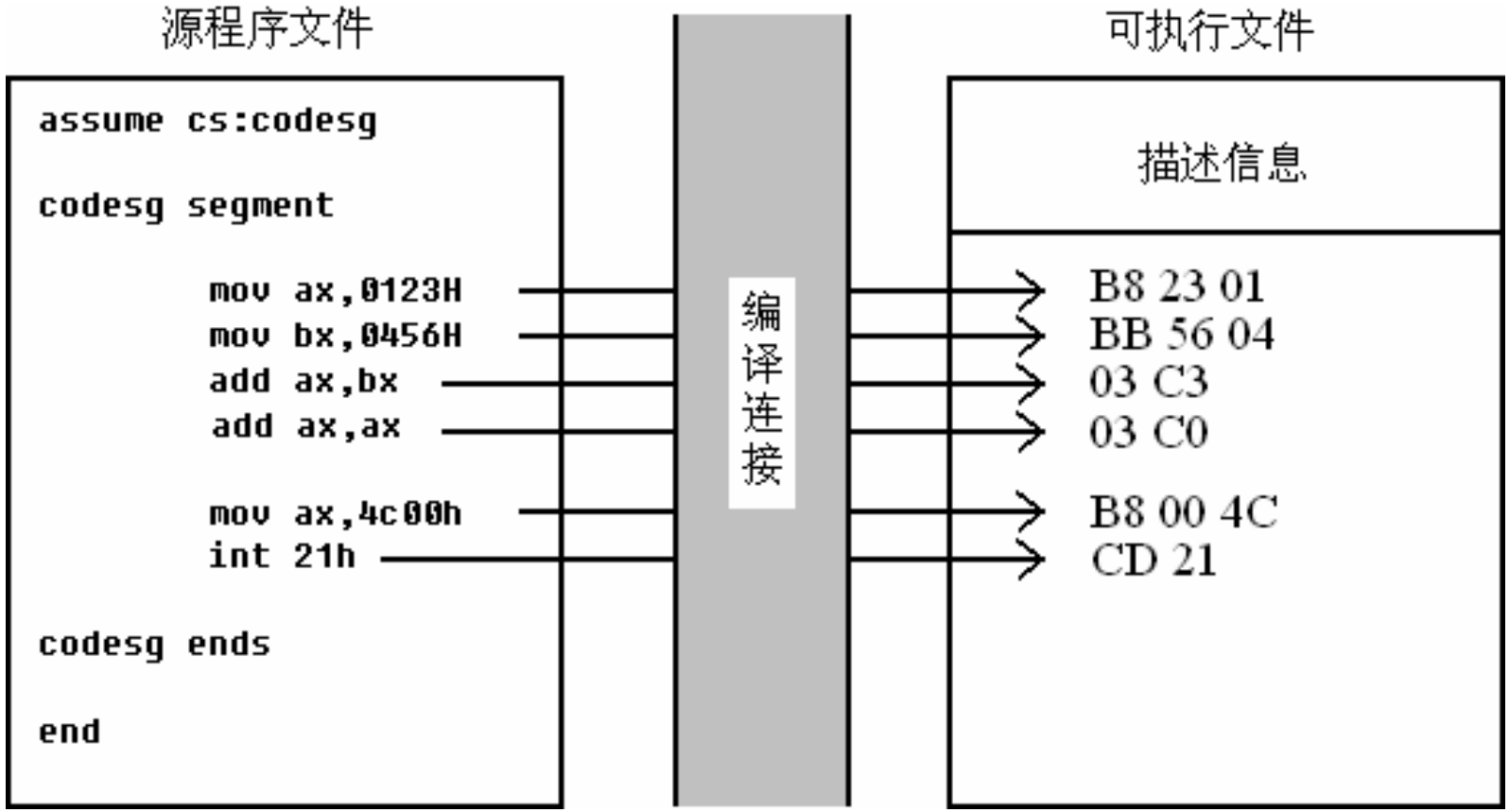
- 源程序中的“程序”
  - 汇编源程序：

    伪指令　（编译器处理）

    汇编指令（编译为机器码）

  - 程序：源程序中最终由计算机执行、处理的指令或数据
- 图示

程序经编译连接后变为机器码



4.2 源程序

- 标号
  - 一个标号指代了一个地址。
  - **codesg**：放在**segment**的前面，作为一个段的名称，这个段的名称最终将被编译、连接程序处理为一个段的段地址。

4.2 源程序

- 程序返回
  - 应该在程序的末尾添加返回的程序段。

    mov ax,4c00H

    int 21H

    - 这两条指令所实现的功能就是程序返回。

- 几个和结束相关的内容

段结束、程序结束、程序返回

目 的	相关指令	指令性质	指令执行者
通知编译器一个段结束	段名 ends	伪指令	编译时，由编译器执行
通知编译器程序结束	end	伪指令	编译时，由编译器执行
程序返回	mov ax,4c00H int 21H	汇编指令	编译时，由CPU执行

汇编语言中数据位置的表达

- 汇编语言中用来表达数据的位置:
  - **1、立即数（idata）**
  - **2、寄存器**
  - **3、段地址（SA）和偏移地址（EA）**

立即数（idata）

直接包含在机器指令中的数据（执行前在CPU 的指令缓冲器中）称为：立即数（idata)，在汇编指令中直接给出

例如：

mov ax,1

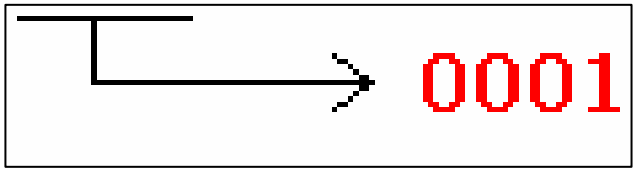
add bx,2000h

or bx,00010000b

mov al,'a'

对应机器码：

B80100



执行结果：**(ax) = 1**

寄存器

数据在寄存器中，在汇编指令中给出相应的寄存器名，如：

mov ax, bx

mov ds, ax

push bx

mov ds:[0], bx

push ds

mov ss, ax

mov sp, ax

mov ax, bx

对应机器码：**89D8**

执行结果：**(ax) = (bx)**

段地址（SA）和偏移地址（EA）

数据在内存中，在汇编指令中可用[X]的格式给出EA，SA在某个段寄存器中

mov ax, [bx]

对应机器码：**8B07**

执行结果: **(ax) = ((ds)×16+(bx))**

- 存放段地址的寄存器默认
  - 示例
- 存放段地址的寄存器显性给出
  - 示例

存放段地址的寄存器默认

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

段地址默认在ds中

段地址默认在ss中

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

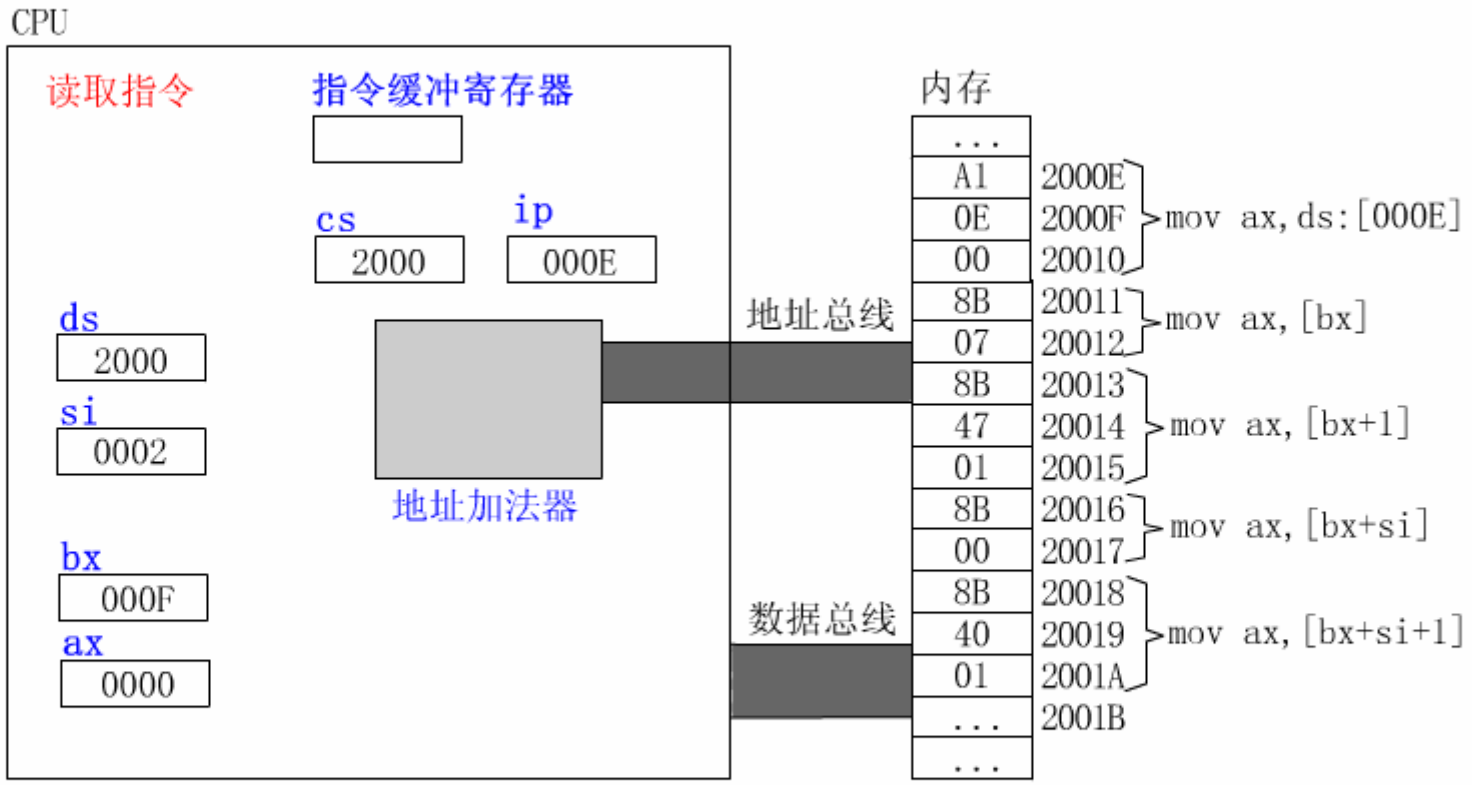
显性的给出存放段地址的寄存器

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

寻址方式	含义	名称	常用格式举例
[idata]	EA=idata;SA=(ds)	立即数 直接寻址	[idata]
[bx] [si] [di] [bp]	EA=(bx);SA=(ds) EA=(si);SA=(ds) EA=(di);SA=(ds) EA=(bp);SA=(ss)	寄存器 寄存器间接寻址	[bx]
[bx+idata] [si+idata] [di+idata] [bp+idata]	EA=(bx)+idata;SA=(ds) EA=(si)+idata;SA=(ds) EA=(di)+idata;SA=(ds) EA=(bp)+idata;SA=(ss)	寄存器立即数 寄存器相对寻址	用于结构体：[bx].idata 用于数组：idata[si] , idata[di] 用于二维数组：[bx][idata]
[bx+si] [bx+di] [bp+si] [bp+di]	EA=(bx)+(si);SA=(ds) EA=(bx)+(di);SA=(ds) EA=(bp)+(si);SA=(ss) EA=(bp)+(di);SA=(ss)	bx, bp, si, di 基址变址寻址	用于二维数组：[bx][si]
[bx+si+idata] [bx+di+idata] [bp+si+idata] [bp+di+idata]	EA=(bx)+(si)+idata; SA=(ds) EA=(bx)+(di)+idata; SA=(ds) EA=(bp)+(si)+idata; SA=(ss) EA=(bp)+(di)+idata; SA=(ss)	bx,bp,si,di立即数 相对基址变址寻址	用于表格（结构）中的数组项： [bx].idata[si] 用于二维数组： idata[bx][si]

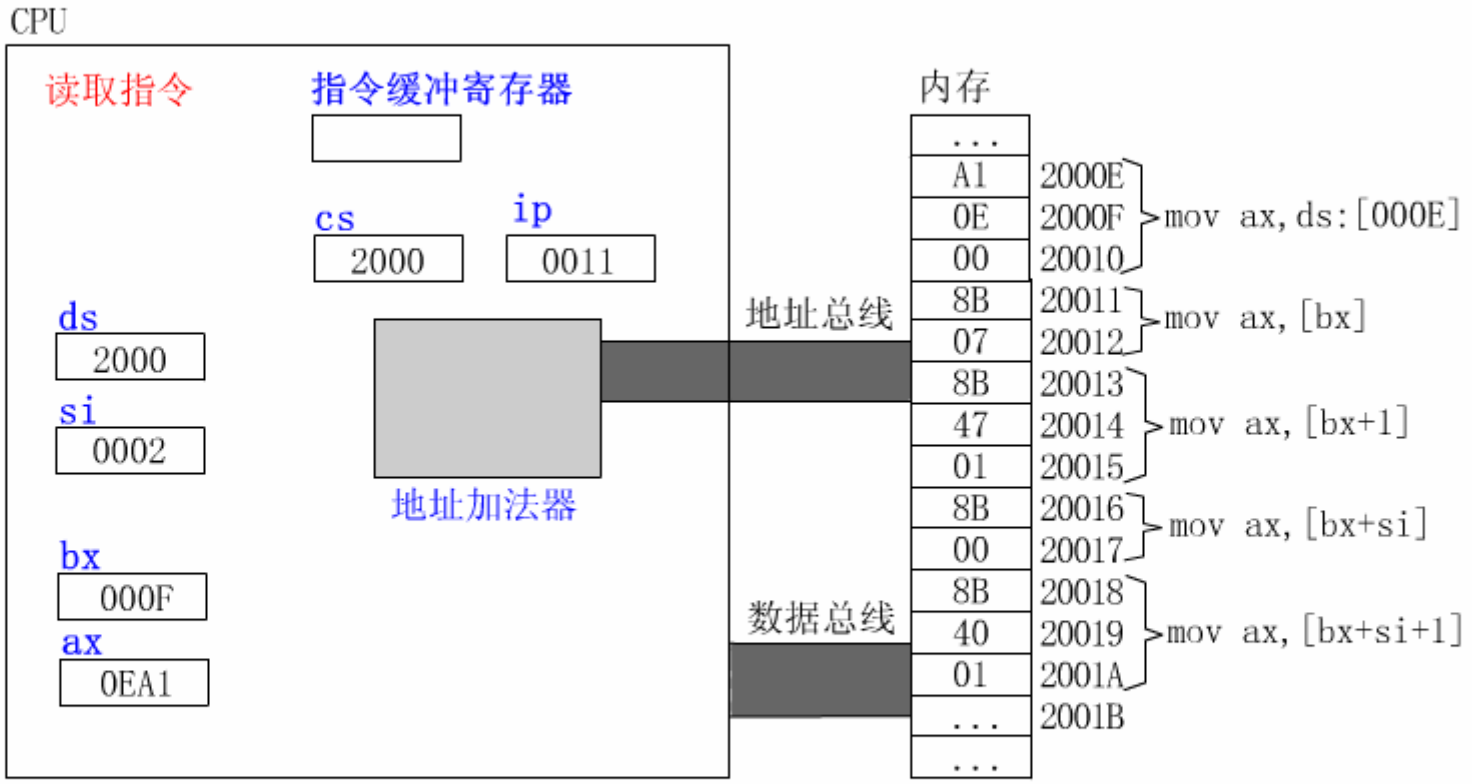
寻址方式

- 寻址方式
  - 演示1、直接寻址
  - 演示2、寄存器间接寻址
  - 演示3、寄存器相对寻址
  - 演示4、基址变址寻址
  - 演示5、相对基址变址寻址



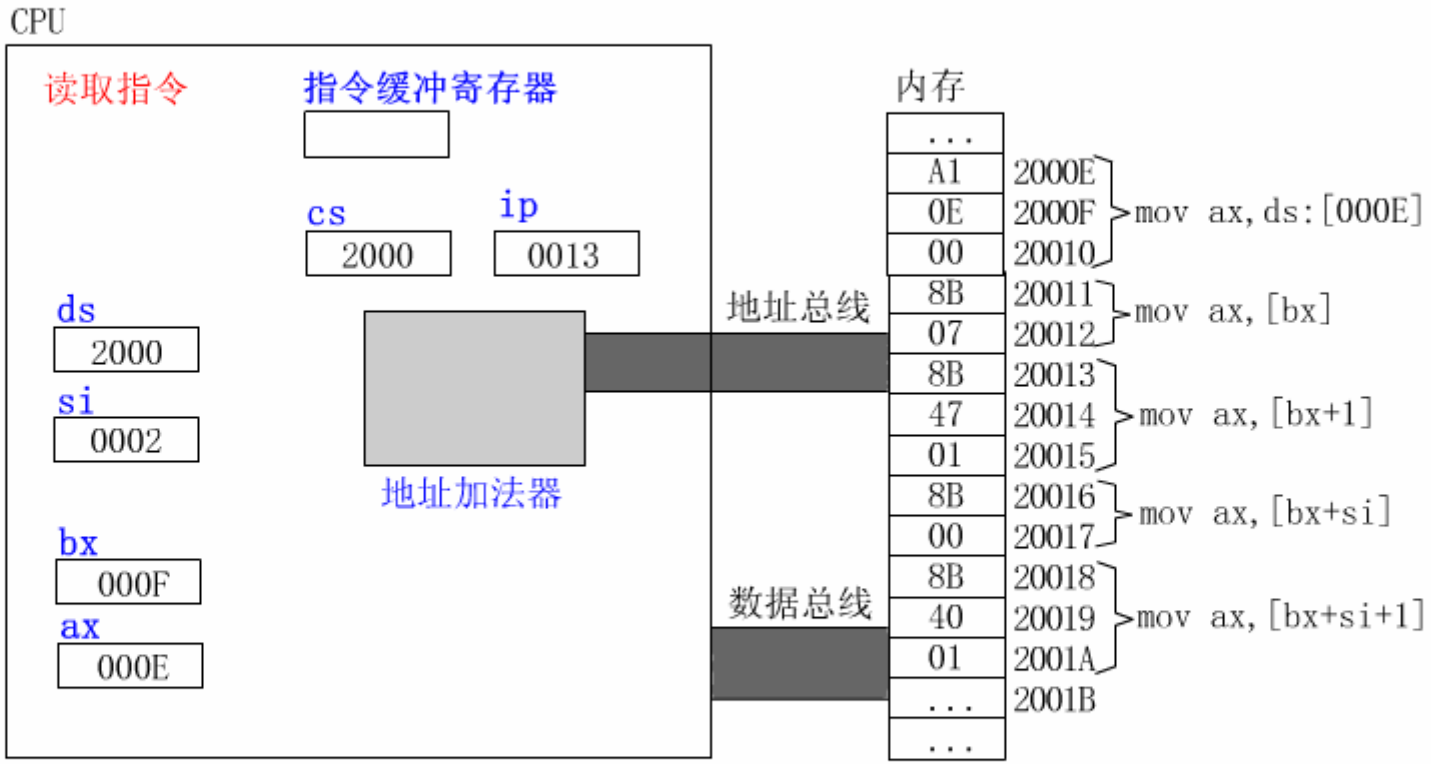
直接寻址过程演示

play step step stop



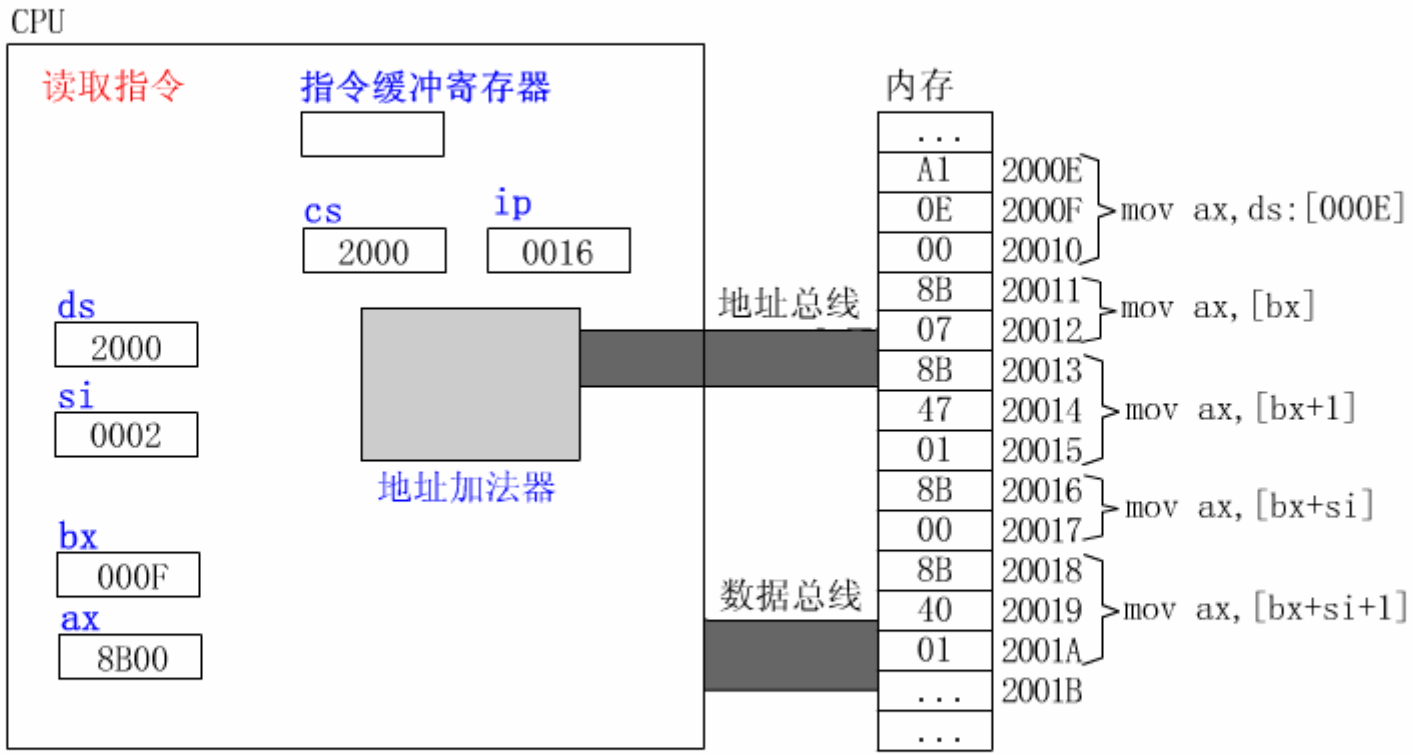
寄存器间接寻址过程演示

play step step stop



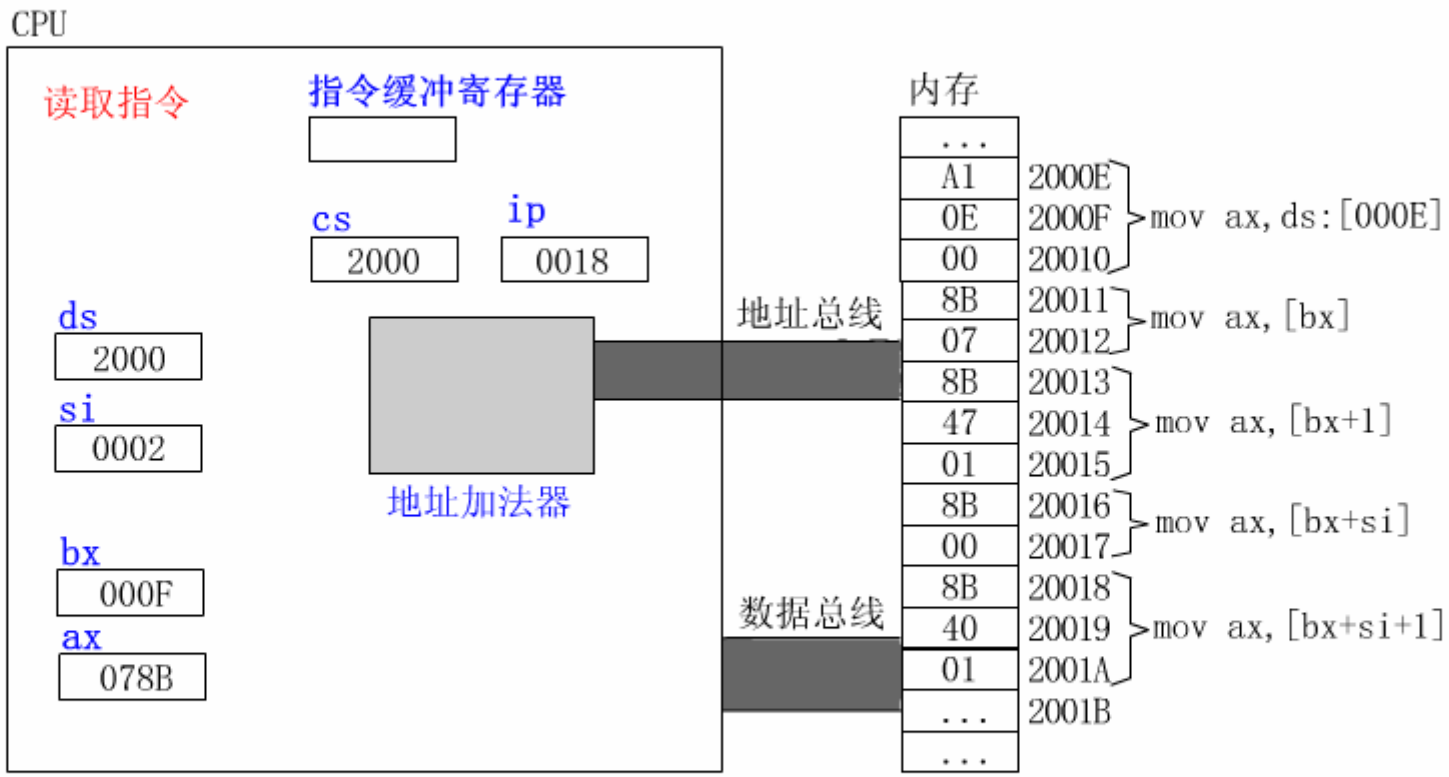
寄存器相对寻址过程演示

play step step stop



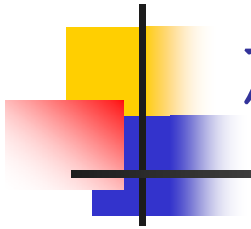
基址变址寻址过程演示

play step step stop



相对基址变址寻址过程演示

play step step stop

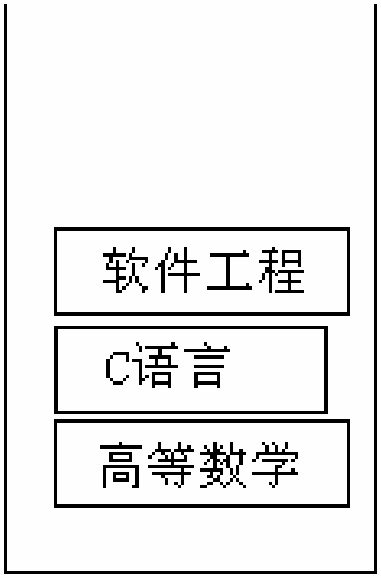


栈

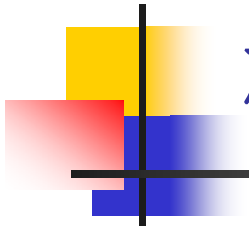
- 栈是一种具有特殊的访问方式的存储空间
  - 最后进入这个空间的数据，最先出去。
- 可以用一个盒子和3本书来描述

栈的操作方式

入栈的方式



出栈的方式



栈

- 栈的两个基本操作：
  - 入栈：将一个新的元素放到栈顶；
  - 出栈：从栈顶取出一个元素。
- 栈顶的元素最后入栈，出栈时，最先从栈中取出
- 栈的操作规则：**LIFO**

  （Last In First Out，后进先出）

CPU提供的栈机制

- 在基于**8086CPU**编程时，可以将一段内存当作栈来使用
- **8086CPU**提供的最基本入栈出栈指令：
  - **PUSH**（入栈）
  - **POP** （出栈）
  - **push ax**：将寄存器**ax**中的数据送入栈中
  - **pop ax**：从栈顶取出数据送到**ax**
- **8086CPU**的入栈和出栈操作都是以字为单位进行

进栈指令：　　PUSH SRC

执行操作：　　(SP) ← (SP) − 2

((SP)+1, (SP)) ← (SRC)

出栈指令：　　POP DST

执行操作：　　(DST) ← ((SP)+1, (SP))

(SP) ← (SP) ＋ 2

堆栈：“先进后出”的存储区，存在于堆栈段中，SP在任何时候都指向栈顶。

注意：

* 堆栈操作必须以字为单位。
* 不影响标志位
* 不能用立即寻址方式　　× PUSH 1234H
* DST不能是CS　　× POP CS

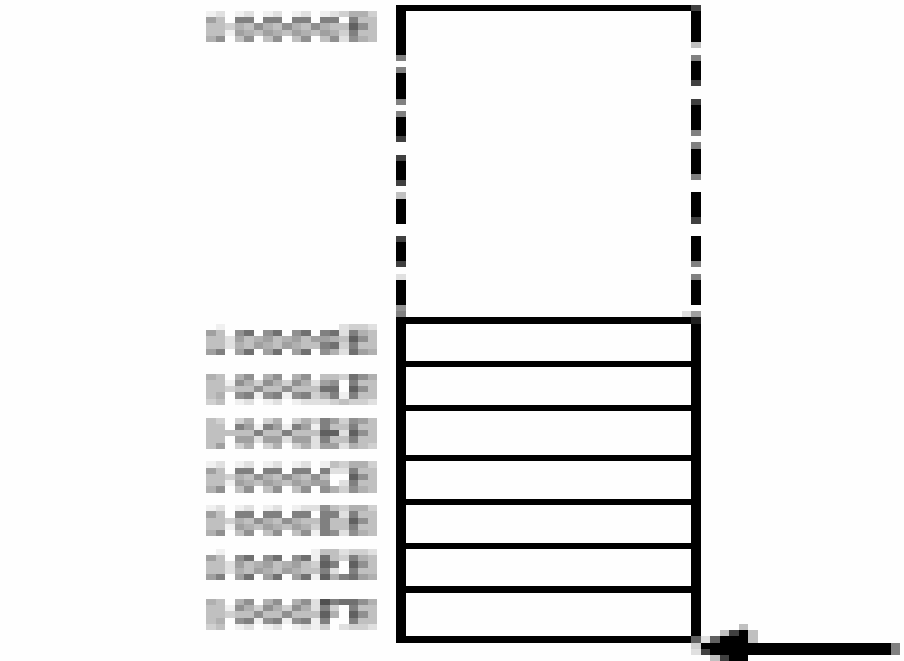
栈

- 例：将10000H~1000FH这段内存当作栈来使用，分析下面一段指令的执行过程：

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

注意：字型数据用两个单元存放，高地址单元放高 8 位，低地址单元放低8 位。

- 指令序列的执行过程演示

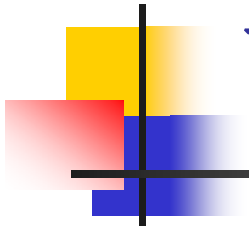


指令序列

```
mov ax,0123H
push ax
mov bx,2368H
push bx
mov cx 1122H
push cx
pop dx
pop bx
pop cx
```

一段以栈的方式访问的内存空间（初始情况）

8086CPU的栈操作



两个疑问

- **CPU**如何知道一段内存空间被当作栈使用？
- 执行**push**和**pop**的时候，如何知道哪个单元是栈顶单元？
- 分析

结论：任意时刻，**SS:SP**指向栈顶元素。

对于两个疑问的分析

回想：**CPU**如何知道当前要执行的指令所在的位置？

寄存器**CS**和**IP**中存放着当前指令的段地址和偏移地址。

8086CPU中，有两个寄存器：

段寄存器**SS**　　存放栈顶的段地址

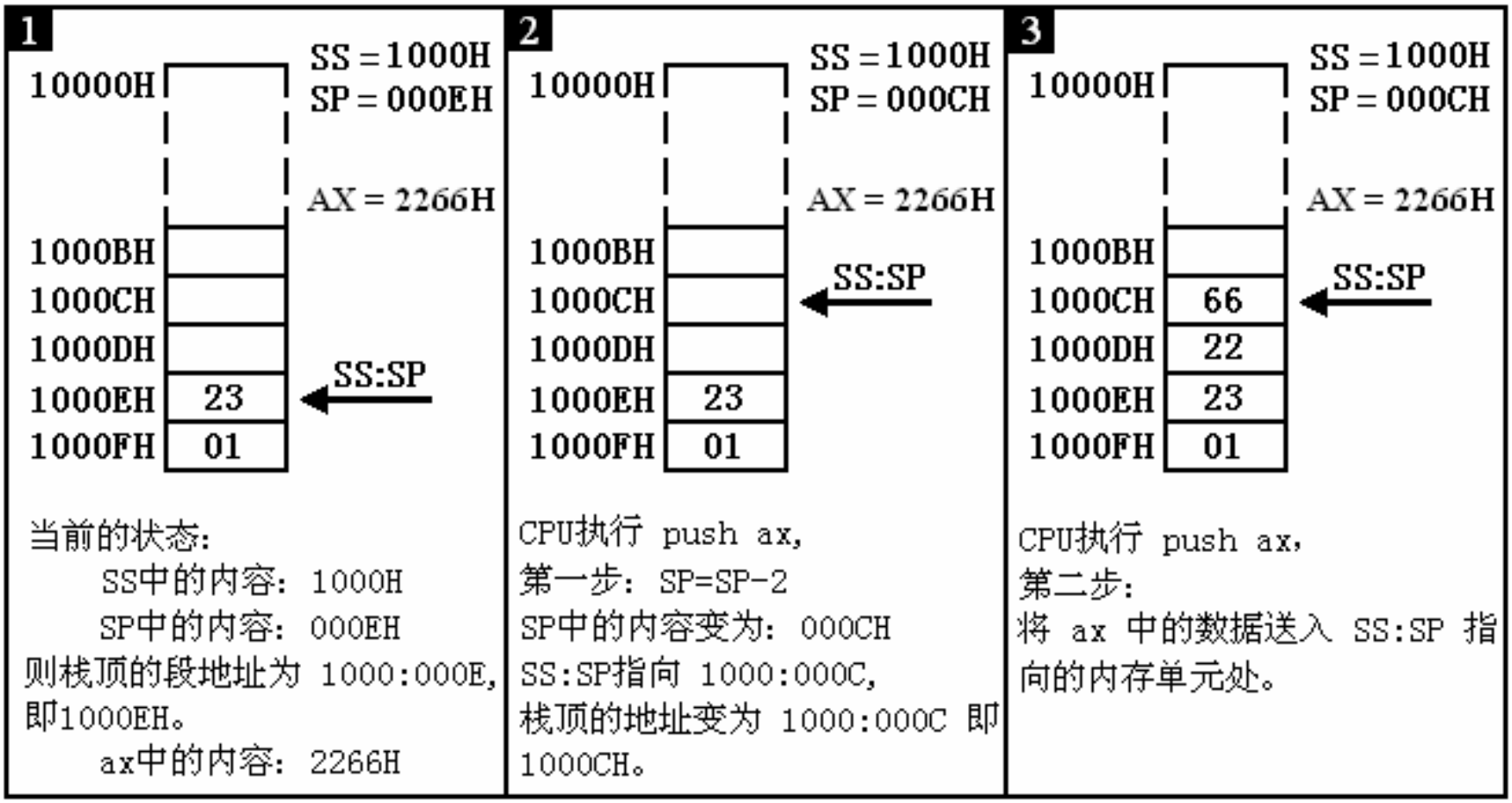
寄存器**SP**　　存放栈顶的偏移地址

任意时刻，**SS:SP**指向栈顶元素

push 指令的执行过程

- **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。**
  - 图示

push 指令的执行过程

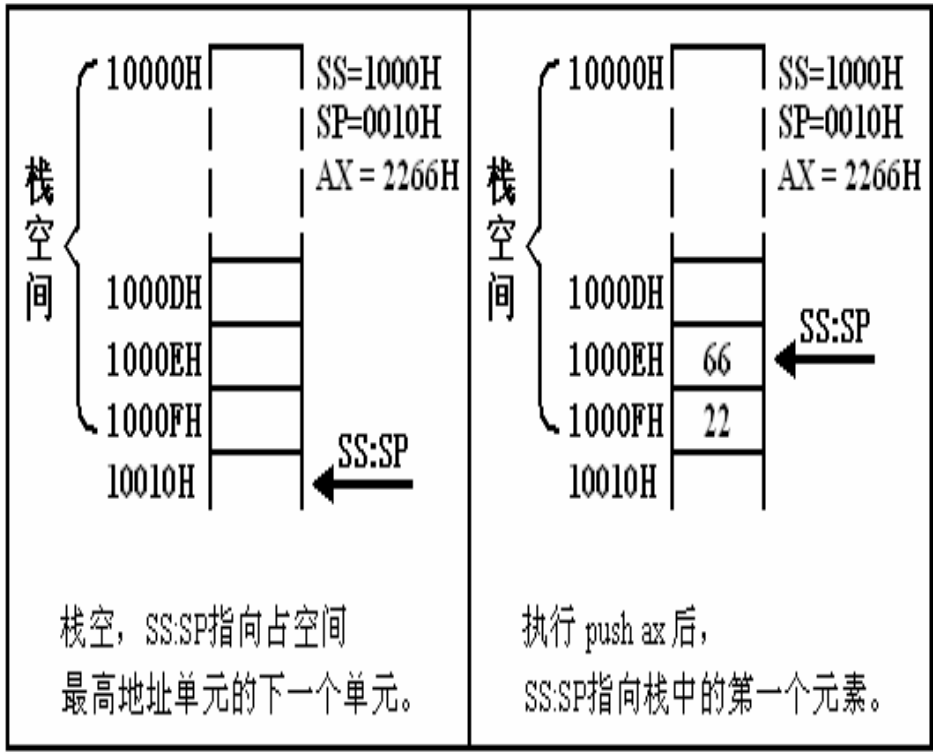


栈

- 问题3.6:
  如果我们将**10000H~1000FH**这段空间当作栈，初始状态栈是空的，此时，**SS=1000H，SP=？**

- 思考后看分析。

SP = 0010H



问题分析

- 将10000H~1000FH 这段空间当作栈段，SS=1000H，栈空间大小为16 字节 ，栈最底部的字单元地址为1000:000E
- 任意时刻，SS:SP指向栈顶，当栈中只有一个元素的时候，

  SS = 1000H，SP=000EH
- 栈为空，相当于栈中唯一的元素出栈，出栈后：SP=SP+2
- 原来为 000EH，加 2 后SP=10H，所以，当栈为空的时候：

  SS=1000H，SP=10H

换个角度看

问题分析（续）

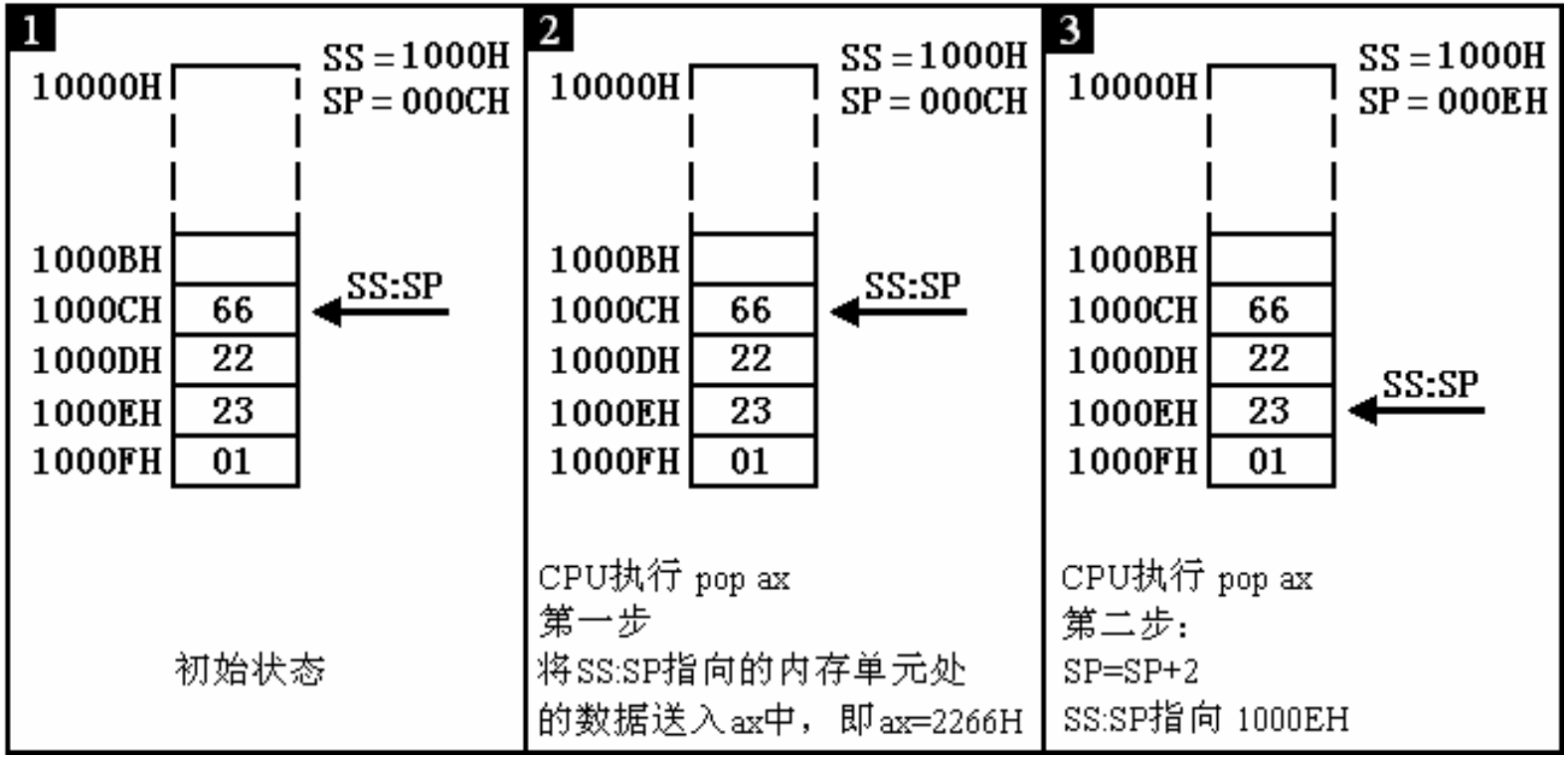
- 换个角度看：

  任意时刻，**SS:SP** 指向栈顶元素，当栈为空的时候，栈中没有元素，也就不存在栈顶元素，所以**SS:SP** 只能指向栈的最底部单元下面的单元，该单元的偏移地址为栈最底部的字单元的偏移地址+2，栈最底部字单元的地址为**1000:000E**，所以栈空时，**SP=0010H**。

pop 指令的执行过程

- **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。
  - 图示

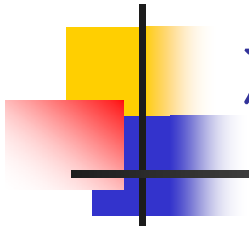
pop 指令的执行过程



- 注意

pop 指令的执行过程

- 注意：
  - 出栈后，**SS:SP**指向新的栈顶 **1000EH**，**pop** 操作前的栈顶元素，**1000CH** 处的**2266H** 依然存在 ，但是，它已不在栈中。
  - 当再次执行**push**等入栈指令后，**SS:SP**移至 **1000CH**，并在里面写入新的数据，它将被覆盖。



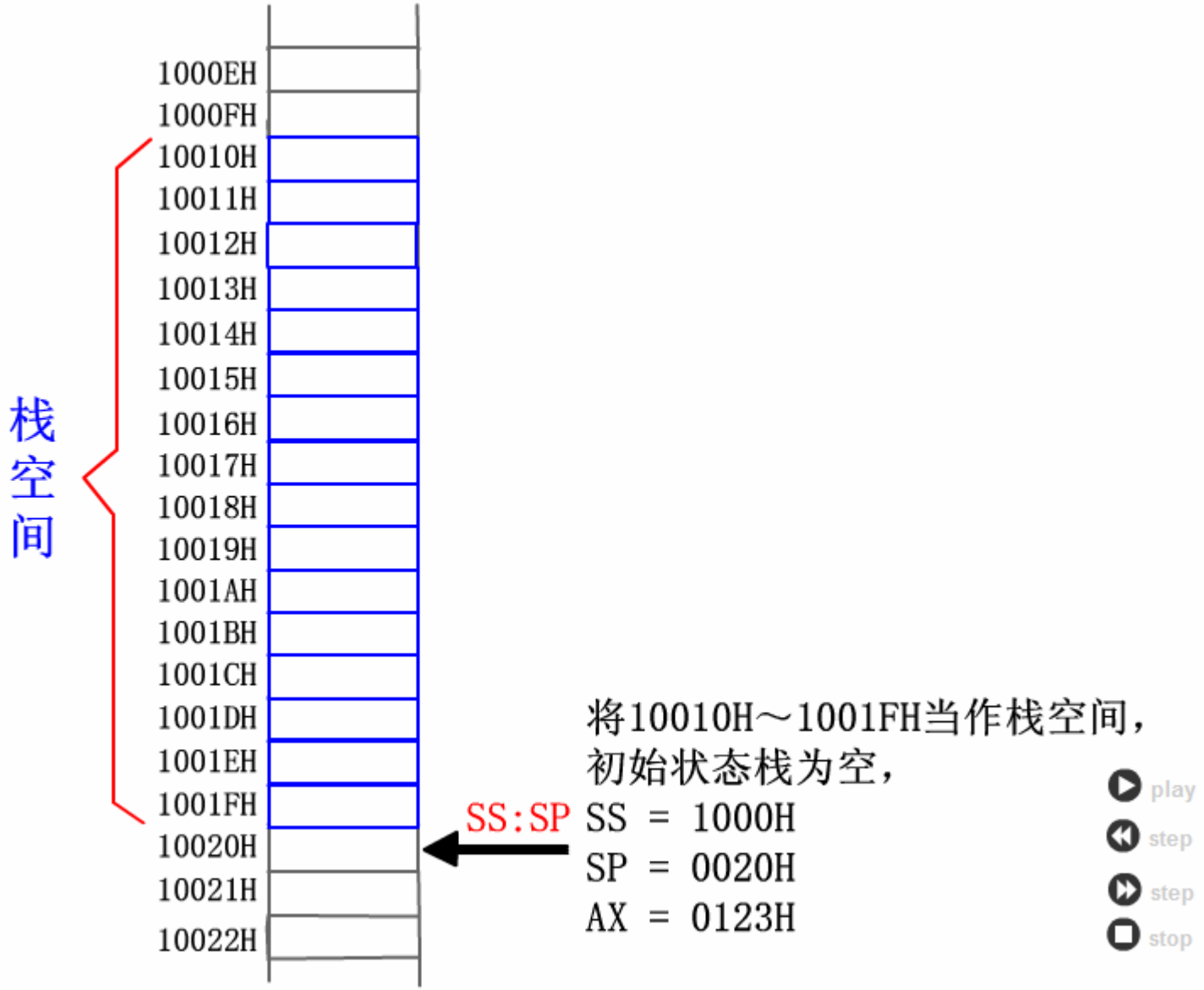
栈顶超界的问题

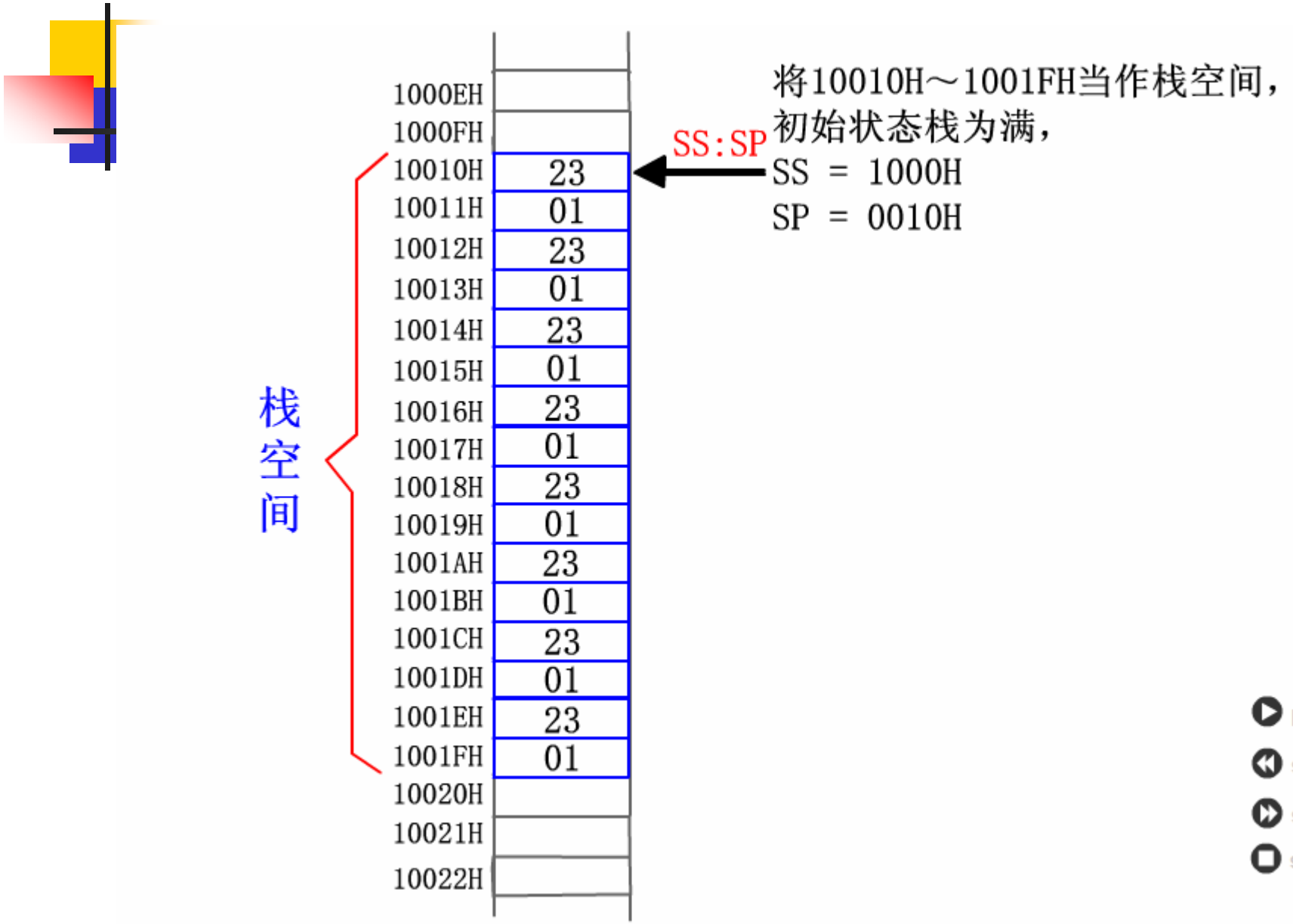
- **SS和SP**只记录了栈顶的地址，依靠**SS和SP**可以保证在入栈和出栈时找到栈顶。

- 可是，如何能够保证在入栈、出栈时，栈顶不会超出栈空间？

栈顶超界的问题

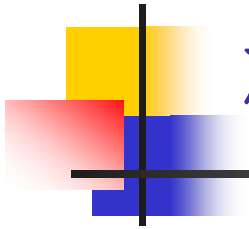
- 当栈满的时候再使用**push**指令入栈，
  栈空的时候再使用**pop**指令出栈，
  都将发生栈顶超界问题。


1000EH
1000FH
10010H
10011H
10012H
10013H
10014H
10015H
10016H
10017H
10018H
10019H
1001AH
1001BH
1001CH
1001DH
1001EH
1001FH
10020H
10021H
10022H
栈空间
SS:SP
将10010H～1001FH当作栈空间，
初始状态栈为空，
SS = 1000H
SP = 0020H
AX = 0123H
play
step
step
stop


1000EH
1000FH
10010H
10011H
10012H
10013H
10014H
10015H
10016H
10017H
10018H
10019H
1001AH
1001BH
1001CH
1001DH
1001EH
1001FH
10020H
10021H
10022H
23
01
23
01
23
01
23
01
23
01
23
01
23
01
SS:SP
将10010H～1001FH当作栈空间，
初始状态栈为满，
SS = 1000H
SP = 0010H
栈空间
play
step
step
stop

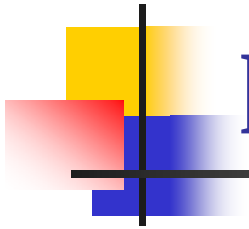
栈顶超界的问题

- 栈顶超界是危险的：
  因为：栈空间之外的空间里很可能存放了具有其他用途的数据、代码等
- 入栈出栈时的栈顶超界，将导致这些数据、代码意外改写而引发一连串的错误。
- **8086CPU**只考虑当前的情况：
  - 当前栈顶在何处；
  - 当前要执行的指令是哪一条。
- 结论



栈顶超界的问题

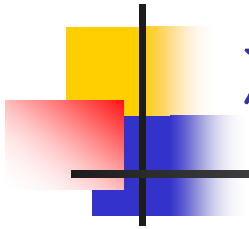
- 结论：
  - 在编程的时候自己操心栈顶超界的问题，根据可能用到的最大栈空间，来安排栈的大小，防止入栈的数据太多而导致的超界
  - 执行出栈操作时注意，防止栈空的时候继续出栈而导致超界。



push、pop指令

- **push和pop指令是可以在寄存器和内存之间传送数据的。**

- **push和pop指令的格式**



栈与内存

- 栈空间当然也是内存空间的一部分，它只是一段可以以一种特殊的方式进行访问的内存空间。

push、pop指令

- push和pop指令的格式（1）
  - push 寄存器：
    - 入栈，将一个寄存器中的数据入栈
  - pop 寄存器：
    - 出栈，用一个寄存器接收出栈的数据

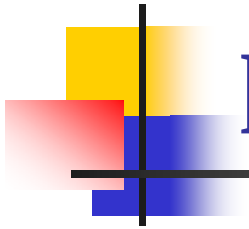
- 例如：push ax
  pop bx

push、pop指令

- **push和pop指令的格式（2）**
  - **push 段寄存器：**
    - 入栈，将一个段寄存器中的数据
  - **pop 段寄存器：**
    - 出栈，用一个段寄存器接收出栈的数据

- 例如：**push ds**

  pop es



push、pop指令

- **push和pop指令的格式（3）**
  - **push 内存单元：**
    - 将一个内存单元处的字入栈（栈操作都是以字为单位）
  - **pop 内存单元：**
    - 出栈，用一个内存字单元接收出栈的数据
- 例如：**push [0]**
  pop [2]
- 指令执行时，可以在 **push、pop** 指令中给出内存单元的偏移地址，段地址从**ds**中取得。

push、pop指令

- 编程：将**10000H~1000FH** 这段空间当作栈，初始状态是空的，将 **AX、BX、DS**中的数据入栈。

- 思考后看分析。

问题分析

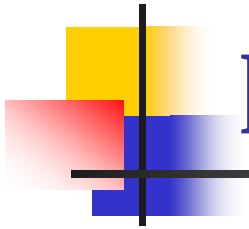
```
MOV  AX, 1000H
MOV  SS, AX

MOV  SP, 0010H

PUSH  AX
PUSH  BX
PUSH  DS
```

设置栈的段地址,**SS=1000H**,不能直接向段寄存器**SS**送入数据,所以用**AX**中转。

设置栈顶的偏移地址,因为栈为空，所以**SP=0010H**。

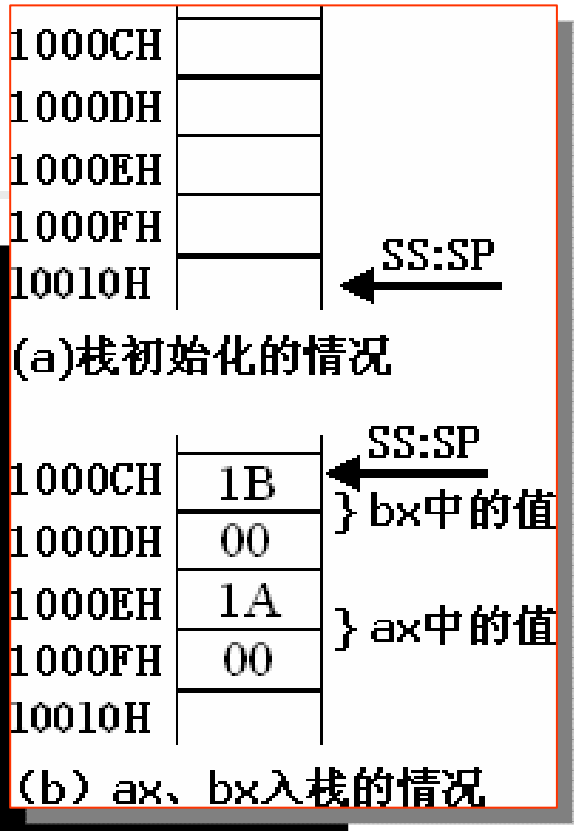


push、pop指令

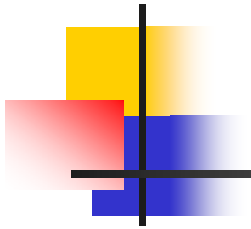
- 编程：
  （1）将10000H~1000FH 这段空间当作栈，初始状态是空的；
  （2）设置AX=001AH，BX=001BH；
  （3）将AX、BX中的数据入栈；
  （4）然后将AX、BX清零；
  （5）从栈中恢复AX、BX原来的内容。
- 思考后看分析。

问题分析

```
MOV  AX, 1000H
MOV  SS, AX
MOV  SP, 0010H    ;初始化栈顶,栈的情况如图a所示
MOV  AX, 001AH
MOV  BX, 001BH
PUSH   AX          ; ax入栈
PUSH   BX          ; bx入栈，如图b所示
MOV  AX, 0         ;将ax清零
MOV  BX, 0         ;将bx清零
POP  BX          ;从栈中恢复ax,bx原来的数据，当前栈顶内容是bx
POP  AX          ;中原来的内容：001BH，ax中原来的内容001AH在
                 ;栈顶的下面，所以要先pop bx，然后再pop ax。
```



- 结论



问题分析

- 从上面的程序我们看到，用栈来暂存以后需要恢复的寄存器中的内容时，出栈的顺序要和入栈的顺序相反，因为最后入栈的寄存器的内容在栈顶，所以在恢复时，要最先出栈。

push、pop指令

- 编程：

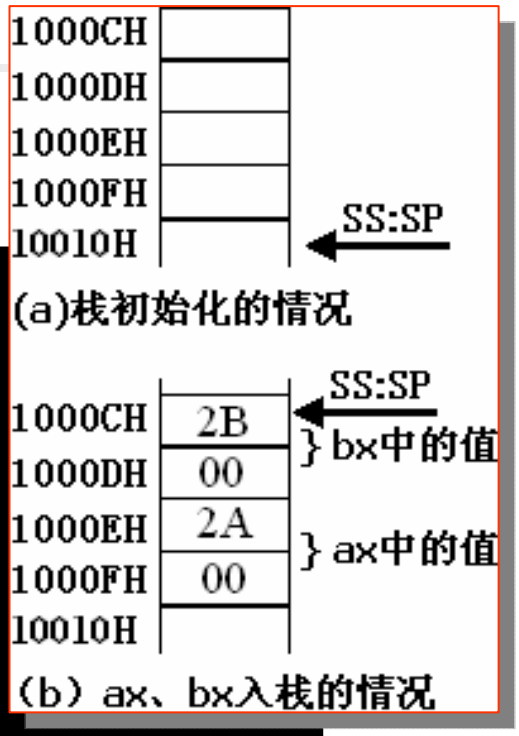
  （1）将10000H~1000FH 这段空间当作栈，初始状态是空的；

  （2）设置AX=002AH，BX=002BH；

  （3）利用栈，交换 AX 和 BX 中的数据。

- 思考后看分析。

问题分析




```
MOV  AX, 1000H
MOV  SS, AX
MOV  SP, 0010H    ;初始化栈顶,栈的情况如图a所示
MOV  AX, 002AH
MOV  BX, 002BH
PUSH   AX              ; ax入栈
PUSH   BX              ; bx入栈,如图b
POP  AX          ; 当前栈顶数据是bx中原来的数据：002B，所以先
                  ; pop ax，ax=002bH;
POP  BX          ;执行pop ax后，栈顶的数据为ax中原来的数据
                 ;所以再pop bx，bx=002AH。
```

push、pop指令

- 思考：如果要在10000H处写入字型数据2266H，怎么完成？ 考虑使用定义DS段，将数据送入段地址完成

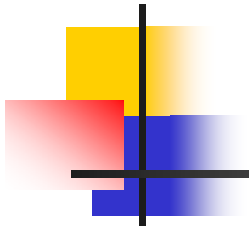
实现代码如下：

```
mov ax,1000H
mov ds,ax
mov ax,2266H
mov [0],ax
```

```
_____
_____
_____
mov ax, 2266H
push ax
```

要求：不能使用“mov 内存单元,寄存器”这类指令

- 补全右边的代码



问题分析

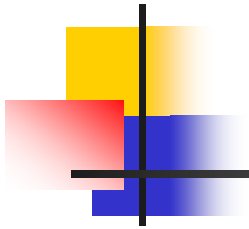
- 看需补全代码的最后两条指令，将**ax**中的**2266H**压入栈中，也就是说，最终应由**push ax**将**2266H**写入**10000H**处。
- 问题的关键：如何使**push ax**访问的内存单元是**10000H**。
- **push**指令是入栈指令。（注意执行过程）
- 完成程序

问题分析（续）

- 完成的程序：

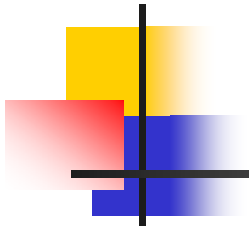
```
mov ax, 1000H
mov ss, ax
mov sp, 2
mov ax, 2266H
push ax
```

- 结论



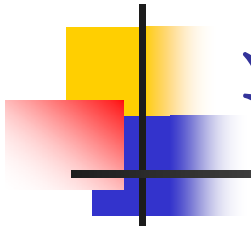
问题分析（续）

- 结论
  - **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
  - 同时，**push**和**pop**指令还要改变 **SP** 中的内容。



问题分析（续）

- **push和pop指令同mov指令不同：**
  - **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**
  - **CPU执行mov指令只需一步操作，就是传送，而执行push、pop指令却需要两步操作。**
- **执行push时：**
  - **先改变SP，后向SS:SP处传送。**
- **执行pop时：**
  - **先读取SS:SP处的数据，后改变SP。**



注意

- **push、pop 等栈操作指令，修改的只是SP。也就是说，栈顶的变化范围最大为：0~FFFFH（Why so?）**
- **提供：SS、SP指示栈顶；改变SP后写内存的入栈指令；读内存后改变SP的出栈指令。**
- **这就是8086CPU提供的栈操作机制。**

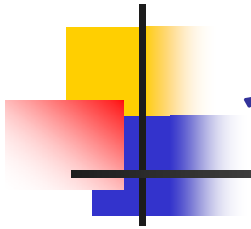
栈的综述

- （1）8086CPU提供了栈操作机制，方案如下：
  在SS，SP中存放栈顶的段地址和偏移地址；
  提供入栈和出栈指令，他们根据SS:SP指示的地址，按照栈的方式访问内存单元。
- （2）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。
- （3）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。

栈的综述（续）

- （4）任意时刻，SS:SP指向栈顶元素。
- （5）8086CPU只记录栈顶，栈空间的大小我们要自己管理。
- （6）用栈来暂存以后需要恢复的寄存器的内容时，寄存器出栈的顺序要和 入栈的顺序相反。
- （7）push、pop实质上是一种内存传送指令，注意它们的灵活应用。

- 栈是一种非常重要的机制，一定要深入理解，灵活掌握。



段的综述

- 我们可以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。这完全是我们自己的安排。
- 我们可以用一个段存放数据，将它定义为“数据段”；
- 我们可以用一个段存放代码，将它定义为“代码段”；
- 我们可以用一个段当作栈，将它定义为“栈段”；

段的综述（续）

- 数据段：将段地址放在 **DS**中
- 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
- 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** 将内存中的某段内存当作代码，是因为**CS:IP**指向了那里；**CPU**将某段内存当作栈，是因为 **SS:SP** 指向了那里。

段的综述（续）

- 一段内存，可以既是代码的存储空间，又是数据的存储空间，还可以是栈空间，也可以什么也不是。
- 关键在于**CPU**中寄存器的设置，即：

CS、IP、SS、SP、DS的指向。