

# 操作系统实验（一）

## 一个最简操作系统的实现

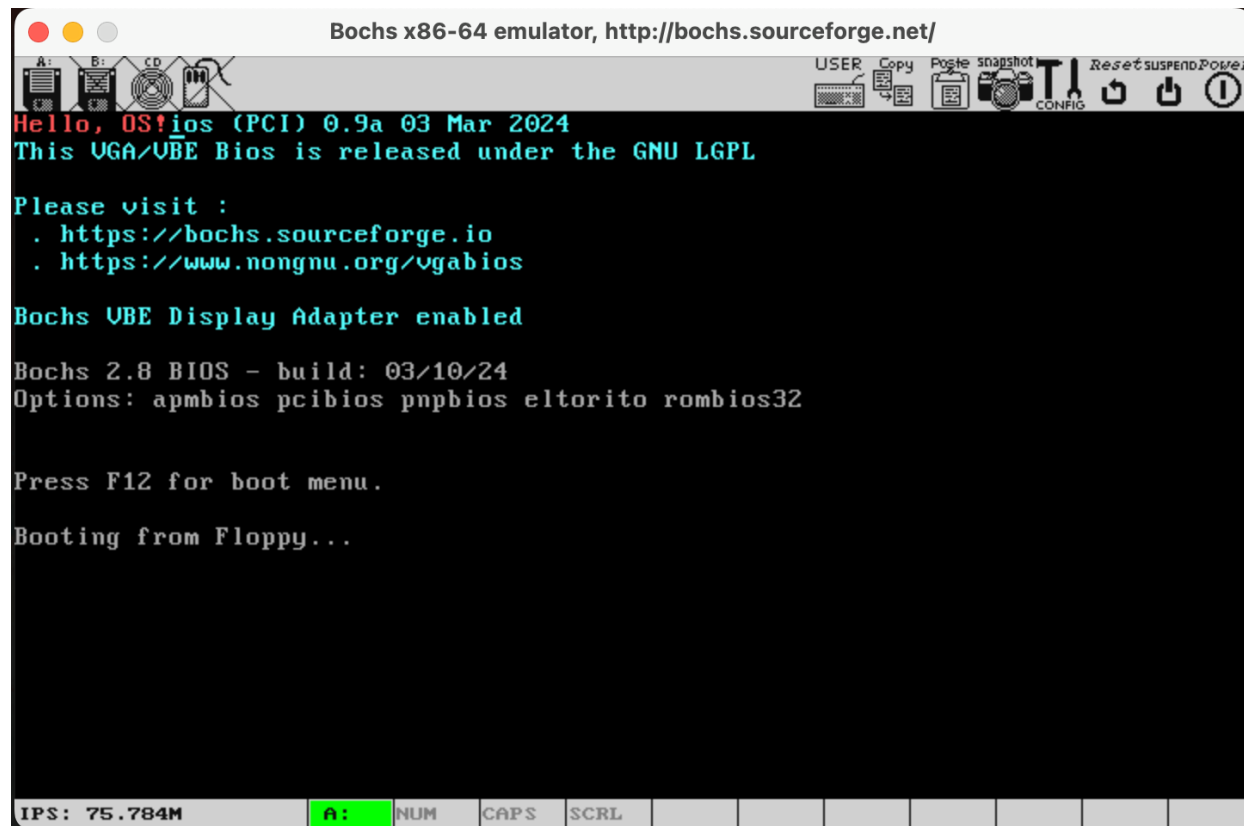
2024年03月13日

# 参考书籍



# 实验任务1

- Hello, OS!



# 准备工作 (1)



- 首先我们要有一台计算机。

## 1.9.2. How do you pronounce "Bochs"?

Phonetically the same as the English word "box". It's just a play on the word "box", since techies like to call their machines a "Linux box", "Windows box", ... Bochs emulates a box inside a box.

- 虚拟机: Bochs

- 安装Bochs虚拟机

- macOS: `brew install bochs`
    - Fedora: `dnf install bochs`
    - Ubuntu: `apt-get install bochs1 bochs-sdl/bochs-x2`

如果使用Ubuntu系统, 请注意:

1. Ubuntu 22.04的Bochs可能无法正常运行, 建议不要选择该版本。

<https://stackoverflow.com/questions/73067357/bochs-can-not-load-bootloader-using-a-floppy-image>

2. 根据Ubuntu版本的不同, 还需要额外安装不同的图形库。如Ubuntu 20需要sdl2库, 更早的版本可能需要x库 (对应安装bochs-sdl或bochs-x)。

# 准备工作 (2)

- 我们还需要装载操作系统的存储设备。
- 软盘（装有操作系统）
  - 创建虚拟软盘映像：`bxiimage`命令，涉及的配置
    - `fd`：软盘映像
    - `1.44M`：映像大小 1.44MB
    - `a.img`：映像名称（在往映像中写入操作系统时以及配置计算机时会用到）

# 开始我们自己的操作系统 (1)

- boot.asm

```
1      org 07c00h      ; 告诉编译器程序加载到07c00h处
2      mov ax, cs
3      mov ds, ax
4      mov es, ax
5      call DispStr    ; 调用显示字符串例程
6      jmp $           ; 无限循环
7
8  DispStr:
9      mov ax, BootMessage
10     mov bp, ax        ; ES:BP = 串地址
11     mov cx, 10        ; CX = 串长度
12     mov ax, 01301h    ; AH = 13, AL = 01h
13     mov bx, 000ch     ; 页号为0 (BH = 0) 黑底红字 (BL=0Ch, 高亮)
14     mov dl, 0
15     int 10h          ; 10h号中断
16     ret
17
18  BootMessage: db "Hello, OS!"
19  times 510-($-$$) db 0 ; 填充剩下的空间, 使生成的二进制代码恰好为512字节
20  dw 0xaa55           ; 结束标志
```

# 开始我们自己的操作系统 (2)

- 我们写的程序要让机器能够执行。
- NASM汇编
  - macOS: `brew install nasm`
  - Fedora: `dnf install nasm`
  - Ubuntu: `apt-get install nasm`

# 开始我们自己的操作系统 (3)

- 使用NASM来汇编`boot.asm`生成“操作系统” (`boot.bin`) 的二进制代码。
- 生成`boot.bin`:
  - `nasm boot.asm -o boot.bin`



# 开始我们自己的操作系统 (4)

- 有了“计算机”，也有了“软盘”，以及“操作系统” (`boot.bin`)，是时候将`boot.bin`写入“软盘”了。
- 使用`dd`命令将`boot.bin`写入软盘中
  - `dd if=boot.bin of=a.img bs=512 count=1`
    - `if`: 代表输入文件
    - `of`: 代表输出设备
    - `bs`: 代表一个扇区大小
    - `count`: 代表扇区数

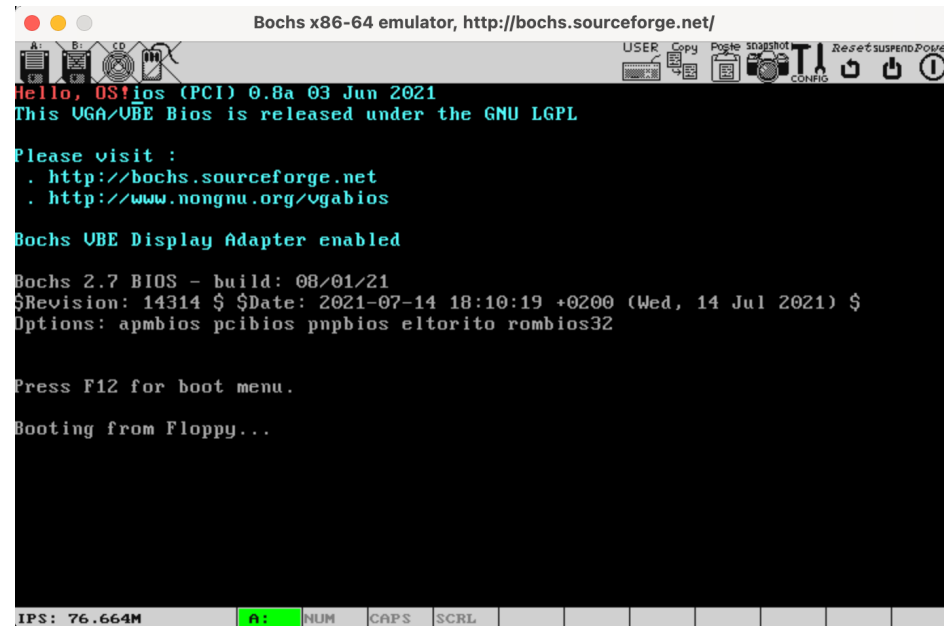
# 启动虚拟机 (1)

- 启动还需要Bochs的配置文件。告诉Bochs, 你希望你的虚拟机是什么样子的, 如内存多大, 硬盘映像和软盘映像都是哪些文件等内容。
  - `display_library`: Bochs使用的GUI库, 一般情况下用`sdl2`就可以了。如果不支持, 尝试`x`等库。
  - `megs`: 虚拟机内存大小 (MB)
  - `floppya`: 虚拟机外设, 软盘为`a.img`文件
  - `boot`: 虚拟机启动方式, 从软盘启动

```
megs:32
display_library: sdl2
floppya: 1_44=a.img, status=inserted
boot: floppy
```

# 启动虚拟机 (2)

- 配置文件保存为 `bochsrc`，和 `a.img` 以及 `boot.bin` 放在同一目录下。
- 启动Bochs: `bochs -f bochsrc`
- 如果没有出现相应的文字，输入 `c` 就可以了。



# 一些问题

- 我们需要把`boot.bin`放在软盘的第一个扇区，为什么？
- 需要理解BIOS与OS的加载。

# BIOS

- 计算机上电启动后，首先会经过BIOS上电自检，这个过程BIOS会检测硬件设备是否存在问题。如果检测无误的话，将根据BIOS的启动项配置选择引导设备，目前BIOS支持的设备启动项有软盘启动、U盘启动、硬盘启动以及网络启动。
- 当BIOS自检结束后会根据启动选项设置去选择启动设备，即检测软盘的**第0磁头第0磁道第1扇区**，是否以**数值0x55和0xaa**两字节作为结尾。如果是，那么BIOS就认为这个扇区是一个**引导扇区**，进而把此扇区的数据复制到**物理内存地址0x7c00处**，随后将处理器的执行权移交给这段程序（跳转至0x7c00地址处执行）。

# BIOS

```
1      org 07c00h      ; 告诉编译器程序加载到07c00h处
2      mov ax, cs
3      mov ds, ax
4      mov es, ax
5      call DispStr    ; 调用显示字符串例程
6      jmp $           ; 无限循环
7
8  DispStr:
9      mov ax, BootMessage
10     mov bp, ax        ; ES:BP = 串地址
11     mov cx, 10        ; CX = 串长度
12     mov ax, 01301h    ; AH = 13, AL = 01h
13     mov bx, 000ch     ; 页号为0 (BH = 0) 黑底红字 (BL=0Ch, 高亮)
14     mov dl, 0
15     int 10h          ; 10h号中断
16     ret
17
18  BootMessage: db "Hello, OS!"
19  times 510-($-$$) db 0 ; 填充剩下的空间, 使生成的二进制代码恰好为512字节
20  dw 0xaa55           ; 结束标志
```

# org 07c00h (1)

- 为什么需要org 07c00h这一行代码？
  - BIOS将软盘内容放在07c00h位置，并不是由这行代码决定的。
- org是伪指令
  - 伪指令是指，不生成对应的二进制指令，只是汇编器使用的。也就是说，boot.bin文件里面，没有07c00h这个东西，BIOS不是因为这条指令才把代码放在07c00h的。
  - mov这种指令，就会生成二进制代码，可以直接告诉CPU该做什么。

## org 07c00h (2)

- `org 07c00h`的作用：告诉汇编器，当前这段代码会放在07c00h处。所以，如果之后遇到需要绝对寻址的指令，那么绝对地址就是07c00h加上相对地址。
  - 绝对地址：内存的实际位置（先不考虑内存分页一类逻辑地址）。
  - 相对地址：当前指令相对第一行代码的位置。
- 在第一行加上`org 07c00h`只是让编译器从相对地址07c00h处开始编译第一条指令，相对地址被编译加载后就正好和绝对地址吻合。



## org 07c00h (3)

- 如果去掉这一行会发生什么?
- 反编译boot1.bin
- **ndisasm boot1.bin -o 0**

00000000	8CC8	mov ax,cs
00000002	8ED8	mov ds,ax
00000004	8EC0	mov es,ax
00000006	E80200	call word 0xb
00000009	EBFE	jmp short 0x9
0000000B	B81E00	mov ax,0x1e
0000000E	89C5	mov bp,ax
00000010	B91000	mov cx,0x10
00000013	B80113	mov ax,0x1301
00000016	BB0C00	mov bx,0xc
00000019	B200	mov dl,0x0
0000001B	CD10	int 0x10
0000001D	C3	ret
0000001E	48	dec ax

0000000B B81E00 mov ax,0x1e

取字符串, 由之前所说, 第一行可执行代码  
加载到内存的0x7c00处, 故字符串地址应该为  
0x7c00+0x1e

# org 07c00h (4)

- 修改boot1.asm

```
mov ax,cs
mov ds,ax
mov es,ax
call DispStr
jmp $
DispStr:
mov ax,BootMessage+07c00h
mov bp,ax
mov cx,16
mov ax,01301h
mov bx,000ch
mov dl,0
int 10h
ret
BootMessage:    db "Hello, OS world!"
times 510-($-$$) db 0
dw 0xaa55
```

```
000 8CC8
00000002 8ED8
00000004 8EC0
00000006 E80200
00000009 EBFE
0000000B B81E7C
0000000E 89C5
00000010 B91000
00000013 B80113
00000016 BB0C00
00000019 B200
0000001B CD10
0000001D C3
```

```
mov ax,cs
mov ds,ax
mov es,ax
call word 0xb
jmp short 0x9
mov ax,0x7c1e
mov bp,ax
mov cx,0x10
mov ax,0x1301
mov bx,0xc
mov dl,0x0
int 0x10
ret
```

# OS的加载

- 首先回忆一下我们在BIOS中提到的“操作系统”的加载过程：
  - BIOS程序检查软盘0面0磁道1扇区，如果扇区以0xaa55结束，则认定为引导扇区，将其512字节的数据加载到内存的07c00处，然后设置PC，跳到内存07c00处开始执行代码。
- 其实我们所提到的只是真正的操作系统内核加载过程中的第一步，真正的加载过程大概是这样：“引导-加载内核入内存-跳入保护模式-开始执行内核”，那是不是只要我们将加载内核入内存的代码写入我们之前写的boot.asm中就行了呢？

# 突破512字节的限制

- 其实，除了加载内核，我们要做的事还有准备保护模式（等讲到保护模式后再来理解）等，512字节显然不够，为了突破512字节的限制，我们引入另外一个重要的文件`loader.bin`，引导扇区只负责把loader加载入内存并把控制权交给他，这样将会灵活得多。
- 最终，由loader将内核加载入内存，才开始了真正操作系统内核的运行。

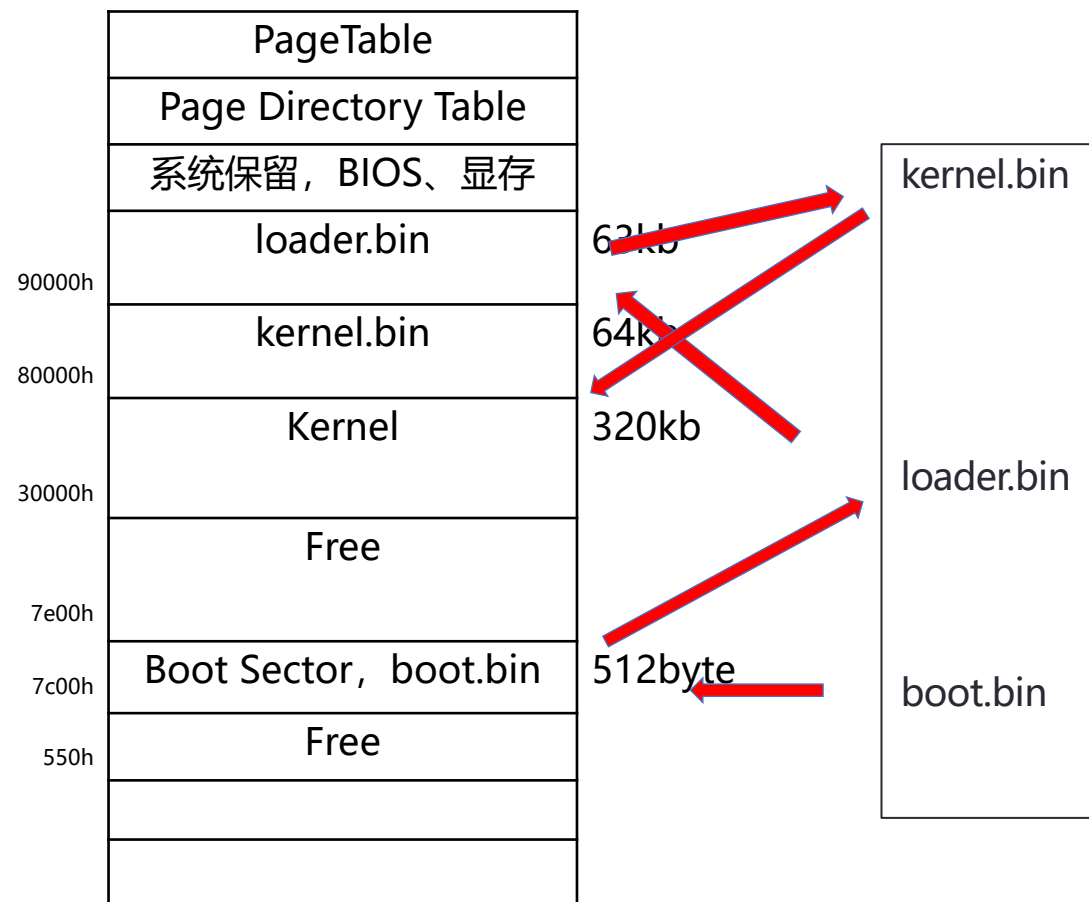
# Loader

- 跳入保护模式
  - 最开始的x86处理器16位，寄存器用ax, bx等表示，称为实模式。后来扩充成32位，eax, ebx等，为了向前兼容，提出了保护模式
  - 必须从实模式跳转到保护模式，才能访问1M以上的内存。
- 启动内存分页
- 从kernel.bin中读取内核，并放入内存，然后跳转到内核所在的开始地址，运行内核
  - 跟boot类似，使用汇编直接在软盘下搜索kernel.bin
  - 但是，不能把整个kernel.bin放在内存，而是要以ELF文件的格式读取并提取代码。

# Kernel

- 这才是真正的操作系统
- 内存管理，进程调度，图像显示，网络访问等等，都是内核的功能。
- 内核的开发使用高级语言
  - C语言可以更高效的编写内核
  - 但我们是在操作系统层面上编写另一个操作系统，于是生成的内核可执行文件是和当前操作系统平台相关的。比如Linux下是ELF格式，有许多无关信息，于是，内核并不能像`boot.bin`或`loader.bin`那样直接放入内存中，需要loader从`kernel.bin`中提取出需要放入内存中的部分。

# 启动流程和内存分布



# 实验任务2

- 为了帮助大家熟悉NASM汇编语言，我们还准备了一个编程小练习。使用NASM汇编语言实现**进制转换**。该任务要求原创，**不得抄袭**！
- 输入：<十进制数> <转换进制>
  - 十进制数的范围是 $[0, 10^{30}]$
  - 转换进制包括二进制b，八进制o和十六进制h
- 输出：转换后的结果，并继续接受下一次输入
  - 输入q结束程序



# 实验任务2

- 至少处理1种无效输入情况（如无效的数字、无效的转换类型或缺少转换类型等），保证程序不崩溃。

输入	输出
1024 b	0b100000000000
1024 o	0o2000
1024 h	0x400
1024 a / -1 b / 1024	Error
q	程序终止

# 实验任务2

评分标准：

- 完成所有功能，且检查时通过所有用例得6分；
- 代码质量不高会酌情扣分；
- 代码雷同或无法解释代码视为抄袭，本次实验计0分。

# 操作系统实验（一）

## 总结

2024年03月13日

# 环境搭建 (1)

- 安装GCC (一般情况下系统自带)
- 安装NASM
- 安装虚拟机Bochs
  - macOS: `brew install nasm bochs`
  - Fedora: `dnf install nasm bochs`
  - Ubuntu: `apt-get install nasm bochs bochs-sdl`

# 环境搭建 (2)

- 说明
  - 实验任务1在Bochs环境中运行，因此无论什么操作系统、架构，只要支持Bochs就行。
  - 实验任务2在本机运行，而不是在Bochs中运行。
    - 装载Apple Silicon芯片的Mac可能会在实验时时出现问题，建议使用X86-64的机器完成实验。

# 具体步骤

- Step 1. 使用NASM汇编`boot.asm`生成“操作系统”的二进制代码
  - `nasm boot.asm -o boot.bin`
- Step 2. 使用`bximage`命令生成虚拟软盘.
  - `bximage -> fd -> 1.44M -> a.img`
- Step 3. 使用`dd`命令将操作系统写入软盘
  - `dd if=boot.bin of=a.img bs=512 count=1`
- Step 4. 配置Bochs
- Step 5. 启动Bochs
  - `bochs -f bochsrc`
  - 如果一开始没显示, 说明是debug模式, 那么只需要按c即可显示。

# 操作系统实验（一）

## 附录: 8086寻址方式和指令系统

2024年03月13日

# 80x86简介 (1)

- 历史

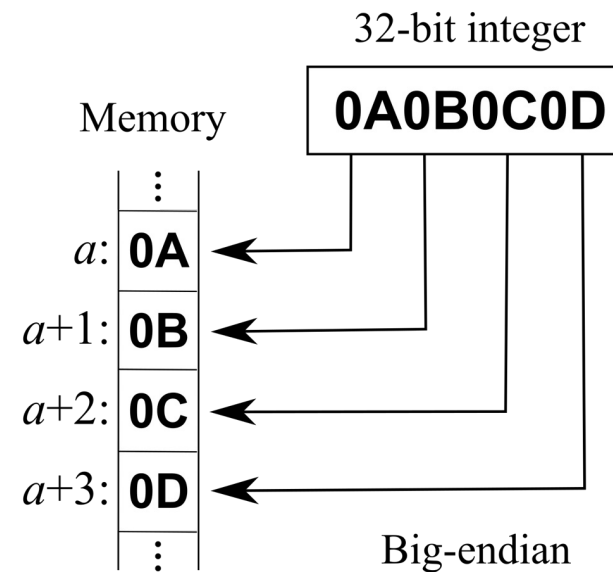
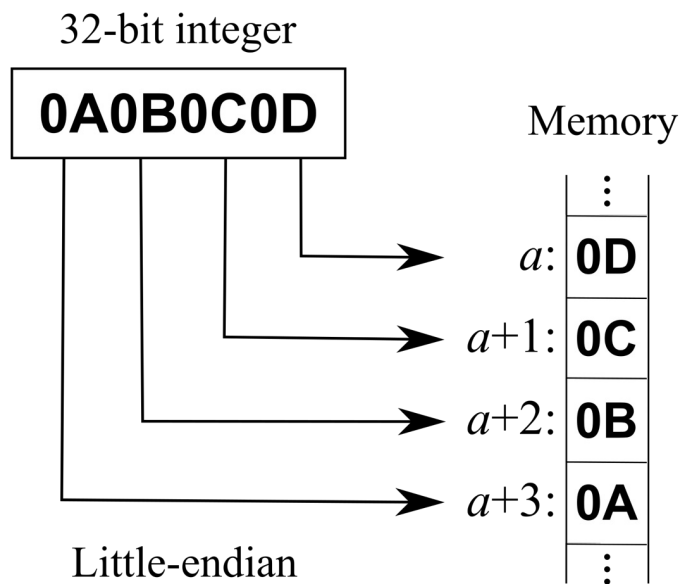
- 1978年6月，Intel推出第一款16位微处理器8086，采用20位地址线
- 1982年发布80286，主频提高至12MHz
- 1985年发布80386，处理器变为32位，地址线扩展至32位
- 1989年发布80486，1993年发布80586并命名为奔腾



# 80x86简介 (2)

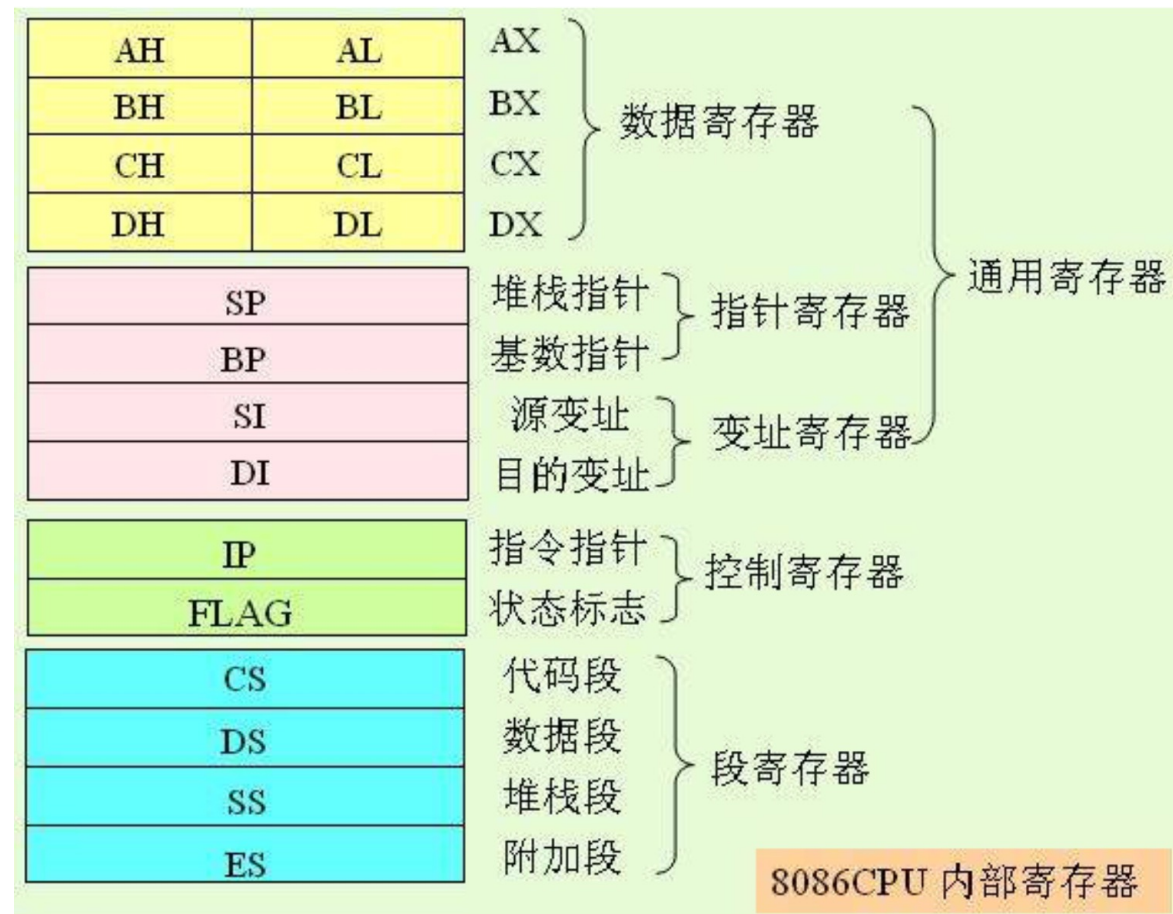
- 特点

- 采用复杂指令集
- 小端存储



# 8086的寄存器

- **SP**: 堆栈指针，与SS配合使用，指向目前的堆栈位置
- **BP**: 基指针寄存器，可用作SS的一个相对基址位置
- **SI**: 源变址寄存器，可用来存放相对于DS段的源变址指针
- **DI**: 目的变址寄存器，可用来存放相对于ES段的目的变址指针



# 8086的寻址方式 (1)

- 寻址
  - 找到操作数的地址(从而能够取出操作数)
- 8086的寻址方式
  - 立即寻址、直接寻址
  - 寄存器寻址、寄存器间接寻址、寄存器相对寻址
  - 基址加变址、相对基址加变址

# 8086的寻址方式 (2)

- 立即寻址
  - `MOV AX 1234H`
  - 直接给出了操作数，事实上没有“寻址”
- 直接寻址
  - `MOV AX [1234H]`
  - 直接给出了地址1234H，用[]符号取数

# 8086的寻址方式 (3)

- 寄存器寻址

- `MOV AX BX`
- 操作数在寄存器里，给出寄存器名即可取走操作数

- 寄存器间接寻址

- `MOV AX [BX]`
- 操作数有效地址在寄存器之中(SI、DI、BX、BP)

- 寄存器相对寻址

- `MOV AX [SI+3]`

# 8086的寻址方式 (4)

- 基址加变址

- `MOV AX [BX+DI]`
- 把一个基址寄存器(BX、BP)的内容，加上变址寄存器(SI、DI)的内容。

- 相对基址加变址

- `MOV AX [BX+DI+3]`

# 如何进行函数传参

- 利用寄存器传递参数
  - 缺点：能传递的参数有限，因为寄存器有限
- 利用约定的地址传递参数
- 利用堆栈传递参数（常用）