问题一：简述 80x86 系列的发展历史

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

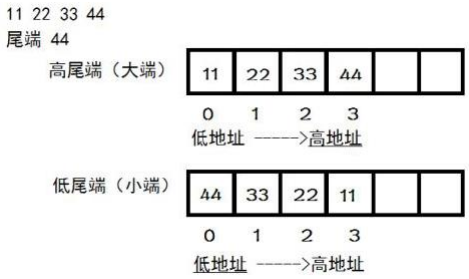
问题二：说明小端和大端的区别，并说明 80x86 系列采用了哪种方式？

（一） 小端

较高的有效字节存放在较高的存储器地址，较低的有效字节存放在较低的存储器地址。

（二） 大端

较高的有效字节存放在较低的存储器地址，较低的有效字节存放在较高的存储器地址。



80x86 系列仍然在使用小端存储（1234 存储时低地址是 34，高地址是 12）的方式

问题三：8086 有哪五种寄存器，请分别举例说明其作用？

数据寄存器、指针寄存器、变址寄存器、控制寄存器、段寄存器

前三个合称通用寄存器

（一） 通用寄存器

（1）数据寄存器

1. AH & AL=AX(accumulator)：**累加寄存器**，常用于运算;在乘除等指令中指定用来存放操作数，另外,所有的 I/O 指令都使用这一寄存器与外界设备传送数据。
2. BH&BL=BX(base):**基址寄存器**，常用于地址索引

3. CH&CL=CX(count):**计数寄存器**，常用于计数;常用于保存计算值，如在移位指令,循环(loop)和串处理指令中用作隐含的计数器.

4. DH&DL=DX(data):**数据寄存器**，常用于数据传递。

（2）指针寄存器

1. SP(Stack Pointer):**堆栈指针寄存器**，与 SS 配合使用，可指向目前的堆栈位置

2. BP(Base Pointer):**基址指针寄存器**，可用作 SS 的一个相对基址位置

（3）变址寄存器

1. SI(Source Index):**源变址寄存器**，可用来存放相对于 DS 段之源变址指针

2. DI(Destination Index):**目的变址寄存器**，可用来存放相对于 ES 段之目的变址指针。

这 4 个 16 位寄存器只能按 16 位进行存取操作，主要用来形成操作数的地址，用于堆栈操作和变址运算中计算操作数的有效地址。

（二） 控制寄存器

1. **指令指针寄存器 IP：**指令指针 IP 是一个 16 位专用寄存器，它指向当前需要取出的指令字节，当 BIU 从内存中取出一个指令字节后，IP 就自动加(取出该字节的长度，如:BIU 从内存中取出的是 1 个字节，IP 就会自动加 1，如果 BIU 从内存中取出的字节数长度为 3，IP 就自动加 3)，指向下一个指令字节。注意，IP 指向的是指令地址的段内地址偏移量，又称偏移地址(Offset Address)或有效地址(EA，Effective Address)。

2. **状态标志寄存器 FLAG：**80x86 有一个 16 位的标志性寄存器 FR，在 FR 中有意义的有 9 位，其中 6 位是状态位，3 位是控制位。标志寄存器(Flags Register,FR)又称程序状态字(Program Status Word,PSW)。这是一个存放条件标志、控制标志寄存器，主要用于反映处理器的状态和运算结果的某些特征及控制指令的执行。

（三） 段寄存器

为了运用所有的内存空间，8086 设定了四个段寄存器，专门用来保存段地址.

1. CS(Code Segment):代码段寄存器

2. DS(Data Segment):数据段寄存器

3. SS(Stack Segment):堆栈段寄存器

4. ES(Extra Segment):附加段寄存器。

问题四：什么是寻址？立即寻址和直接寻址的区别是什么？

找到操作数的地址（从而能够取出操作数）叫做**寻址**

立即寻址

MOV AX 1234H　　#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

问题五:请举例说明寄存器间接寻址、寄存器相对寻址、基址加变址寻址、相对基址加变址寻址四种方式的区别

寄存器间接寻址

MOV AX [BX]　操作数所在存储器的有效地址在指令中的寄存器(SI、DI、BX、BP)中

寄存器相对寻址

MOV AX　[SI+3]　操作数所在存储器的有效地址为指令中的寄存器加位移量

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，并以一个段寄存器作为地址基准

相对基址加变址

MOV AX [BX+DI+3] 操作数所在存储器的有效地址为指令中的基址寄存器加变址寄存器，再加位移量

问题六：请分别简述 MOV 指令和 LEA 指令的用法和作用

LEA 指令的功能是取偏移地址，例如 LEA AX,[1000H]，作用是将源操作数[1000H]的偏移地址 1000H 送至 AX；AX = 1000H

MOV 指令的功能是传送数据，例如 MOV AX,[1000H]，作用是将 1000H 作为偏移地址，寻址找到内存单元，将该内存单元中的数据送至 AX；AX = 1000H 对应的地址单元中的值

问题七：请说出主程序与子程序之间至少三种参数传递方式

（一）利用寄存器传递参数

利用寄存器传递参数就是把参数放在约定的寄存器中，这种方法的优点是实行简单和调用方便，但是由于寄存器的个数有限，并且寄存器往往还需要存放其他数据因此只适合传递参数比较少的情况

（二）利用约定的存储单元传递参数

在传递参数较多的情况下，可以利用约定的内存变量来传递参数，这种方法的优点是子程序要处理的数据或送出的结果都有独立的存储单元，编写程序的时候不容易出错但是，这种方法要占用一定的存储单元并且通用性较差

（三）利用堆栈传递参数

如果使用堆栈传递入口参数，那么主程序在调用子程序之前，把需要传递的参数依次压入堆栈，子程序从堆栈中取入口参数，如果使用堆栈传递出口参数，那么子程序在返回前，把需要返回的参数存入堆栈，主程序在堆栈中取出口参数即可

利用堆栈传递参数可以不用占用寄存器，也无需额外使用存储单元，由于参数和子程序的返回地址混在一起，有时还要考虑保护寄存器，所以比较复杂，通常利用堆栈传递参数的入口参数，而利用寄存器传递出口参数

（四）利用 CALL 后续区传递参数

CALL 后续区是指位于 CALL 指令后的存储区，主程序在调用子程序之前，把入口参数存入 CALL 指令后面的存储区，子程序根据保存在堆栈中的返回地址找到入口参数，这种传递参数的方法称为 CALL 后续传递参数法，由于这种方法把数据和代码混在一起，在 x86 系列中使用的不多

问题八：如何处理输入和输出，代码中哪里体现出来?

（一） 输入

```
mov eax, 3 ;OPCODE 3
mov ebx, 0 ;STDIN
mov ecx, digit ; mov the address of symbol of digit into ecx
mov edx, 1 ;Length
int 80h
```

（二） 输出

```
mov eax, 4 ;OPCODE 4
mov ebx, 1 ;STDOUT
mov ecx, string ; mov the address of symbol of string into ecx
mov edx, length ;Length
int 80h
```

问题九:有哪些段寄存器

CS(Code Segment):代码段寄存器

DS(Data Segment):数据段寄存器

SS(Stack Segment):堆栈段寄存器

ES(Extra Segment):附加段寄存器。

问题十：通过什么寄存器保存前一次的运算结果，在代码中哪里体现出来

1. 乘法高位在 edx，低位在 eax

2. 除法商在 eax，余数在 edx

如果 OPRD 是字节操作数,则把 AL 中的无符号数与 OPRD 相乘,16 位结果送到 AX 中;如果 OPRD 是字操作数,则把 AX 中的无符号数与 OPRD 相乘,32 位结果送到 DX 和 AX 对中,DX 含高 16 位,AX 含低 16 位。所以由操作数 OPRD 决定是字节相乘,还是字相乘。例如:

问题十一：解释 boot.asm 文件中，org0700h 的作用

BIOS 把代码放在 07c00h 不是这行代码决定的，它是伪指令，不会产生对应的二进制指令，它的作用是**告诉编译器代码放在 07c00h 处**，如果需要**绝对寻址**，绝对地址就是 07c00h 加上相对地址。编译器绝对寻址默认是从 0000h 开始运算，加上这条伪指令的话编译器会从 07c00h 开始编译第一条指令，下面的指令的相对地址被编译加载后就刚好和绝对地址吻合。**如果去掉第一行，需要把后面所有地址加上 07c00,效果一样**

问题十二：Boot.bin 应该放在软盘的哪一个扇区？为什么？

第一个扇区，因为开机会从 ROM 运行 bios 程序，他会检查软盘 0 面 0 磁道 1 扇区，如果扇区以 0xaa55 结束，则认为是引导扇区，将这 512 字节加载到内存的 07c00 处，并根据伪指令设置 PC 从 07c00 开始执行代码，以上的 0xaa55 以及 07c00 都是一种约定，BIOS 程序就是这样做的。

问题十三：Loader 的作用有哪些

为了突破 512 字节的限制，我们引入另外一个重要的文件，loader.asm,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 kernel 加载入内存，才开始了真正操作系统内核的运行。

(一) 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。

(二) 启动内存分页

(三) 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。

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。

问题十四：解释 NASM 语言中[]的作用

访问标签或者说地址中的内容要用[]

问题十五：解释语句 times 510-($-$$) db 0，为什么是 510？$和$$分别表示什么？

将 0 这个字节重复 510-（$-$$）次，直到程序有 510 字节为止，这样使生成的二进制代码恰好为 512 字节，形成引导扇区。因为最后的 0xaa55 占两个字节，形成 512 字节需要前面有 510 个字节，所以是 510，$表示当前指令被汇编后的地址，$$表示一个 section 的开始处被汇编后的地址，因为程序只有 1 个节所以实际上表示程序被编译后的起始地址即 07c00 以$-$$就等于本行距离程序开始处的绝对距离，本条指令之前的所有字节数

问题十六：解释配置文件 bochsrc 文件中各参数的含义

megs:32 虚拟机内存大小(32MB)

display_library:sdl bochs 使用的 GUI 库，在 Ubuntu 下是 sdl

floppya:1_44=a.img,status=inserted 虚拟外设，软盘为 a.img 文件。3.5 英寸、1.44MB 的软盘一直用于 PC 的标准数据传输，1_44 会从 a.img 中生成一个 1.44MB 的标准软盘映像，为什么替换成b.img是不可以的，猜测是不在引导扇区的问题。

boot:floppy 虚拟机启动方式，从软盘启动