

[1.地址、section、vstart深入理解](#)

[\(1\) 地址浅析](#)

[\(2\) section浅析](#)

[\(3\) vstart浅析](#)

[2.CPU的实模式](#)

[\(1\) CPU的工作原理](#)

[\(2\) 实模式下的寄存器](#)

[\(3\) 实模式下的ret, 函数返回值](#)

[\(4\) 实模式下的call](#)

[call的相对近调用:](#)

[call的间接绝对近调用:](#)

[call实模式直接绝对远程调用](#)

[call实模式间接绝对远程调用](#)

[\(5\) 实模式下的jmp](#)

[\(6\) 有条件转移](#)

[3.对显卡进行硬件编程](#)

1.地址、section、vstart深入理解

(1) 地址浅析

表示所在`section`的起始地址，典型代码有：`movax,`

`mov ax, [var]` 括号[]的作用是取目标地址处的内容，编译后内容为 `mov ax, [0xd]`

`$`是个隐藏的行号，表示本行的地址

双字节变量`var`，空间大小为2byte，一共有16位

分析反汇编代码，我们得到结论：

地址=上一个地址+上一个地址处内容的长度

(2) section浅析

现在可以趾高气昂的得出结论：关键字`section`没有对程序中的地址产生任何影响，默认情况下，有没有`section`都是一样的，地址仍然是相对于整个程序的顺延。

(3) vstart浅析

在`section`被给予一个`vstart`后，为`section`内的数据指定了一个虚拟的起始地址。

如果只根据这个地址，时无法找到目标数据的，也就是这是一个虚拟地址（和x86 CPU

开启分页后的虚拟地址不是同一个东西)

使用section.data.start，获取的仍然是段内偏移量。

倘若设定了vstart，就会让\$\$和\$都按照vstart开始编写地址顺序，这就为直接定位获得了方便。

这个vstart被用来计算程序段内的内存引用地址。

2.CPU的实模式

(1) CPU的工作原理

控制单元、运算单元、存储单元

控制单元（让CPU做什么）：指令寄存器IR，指令译码器ID、操作控制器OC（Operation Controller）

指令指针寄存器IP指向下一条待执行指令的地址。

IA32指令的格式：

前缀 操作码 寻址方式、操作数类型 立即数 偏移量

存储单元：CPU内部的L1、L2 cache缓存以及寄存器，待处理的数据放在这些存储单元内。

寄存器有两类：

程序员可以使用的寄存器，程序可见寄存器：通用寄存器、段寄存器

程序不可见寄存器：数据暂存寄存器

运算单元：负责计算+位操作

(2) 实模式下的寄存器

不可见寄存器：全局描述符表GDTR、中断描述符表寄存器IDTR、任务寄存器TR、控制寄存器CR0-3、指令指针寄存器IP、

可见寄存器：汇编语言可以直接操纵的寄存器，段寄存器、通用寄存器。

sreg段寄存器

CS：代码段寄存器（代码段是一个又一个的指令）

DS：数据段寄存器

SS：栈段寄存器

ES FS GS 附加段寄存器，额外使用的内容

在实模式下，默认的段寄存器都是16位的。

段内偏移地址：IP寄存器，这个是不可兼得

通用寄存器

无论在什么模式，通用寄存器都有8个，分别是AX、BX、CX、DX、SI、DI、BP、SP

约定了一些简单的用法：

cx用于循环次数控制、bx存储起始地址。esi为数据复制的源地址，edi作为目的地址。

栈指针寄存器bp，基址指针寄存器SP（可以配合SS，ss:sp的方式，像访问数据段那样，直接访问栈）

sp是栈顶指针，bp是可以把栈当做数据段访问的指针。

地址回卷：倘若用段基址左移四位+偏移地址，会存在内存溢出的问题，此时由于总线长度问题，地址会直接回卷，也就是对1MB（0xFFFFF）取模，这样就把传说中的高端内存区回卷了。

栈Stack是一种简单的线性表，栈基址寄存器SS，栈顶寄存器SP，栈自定义偏移寄存器BP，这几者共同构成了内核栈的内容？对滴。

栈溢出指的是栈段寄存器SS和栈顶指针SP指向的内存区域无法容纳数据，于是数据跨越了内存范围，出现了经典的栈溢出现象。

(3) 实模式下的ret，函数返回值

CPU的前进方向是CS:IP寄存器方向，CS是代码段段基址，IP是代码段段内偏移量。

CPU中的call、ret、jmp指令包含了许多微操作，保存现场等。

return (ret) 指令的功能是在内核栈栈顶弹出2byte，16位，替换IP（代码段内偏移），与近调用对应。

return far (retf) 指令在内核栈栈顶弹出4byte，顶上的2byte被赋予给IP，之后的2byte被赋予给CS（代码段基址寄存器），与远调用对应。

在x86体系中，CS:IP寄存器组代表了PC程序计数器。

(4) 实模式下的call

call：呼叫、调用

call的相对近调用：

近（两段代码在同一个代码段里，也就是CS寄存器一样），相对（调用的立即数是相对地址，从call func的末尾到func的开头的距离）这个是由硬件设计决定。

短转移：jmp -2和这个近调用是类似的。

call near program(相对地址)这里的地址在编译后是立即数

call的间接绝对近调用：

一般来说，是call 寄存器寻址 或 call [0x12345]内存寻址

当近调用发生时，CPU将IP寄存器的值压入栈，再把新的IP段内偏移量更新到IP寄存器上。

数据类型伪指令：byte、word、dword（双字，四字节）、qword（八字节）

call实模式直接绝对远程调用

目标指令和当前指令不再同一个段里面。

call 段基址（立即数）：段内偏移地址（立即数）

call实模式间接绝对远程调用

16位

远程调用，先把cs压栈，再把ip压栈，从而在后面可以把ip出栈、cs出栈，这个操作流程是非常清晰的。

(5) 实模式下的jmp

jmp命令就是简单的改变cs和ip寄存器，直接改变程序的运行顺序。

jmp命令实际上是短转移，和前面的call的相对近调用类似，用的是相对地址。

当然，写代码肯定用的是关键字，编译器会自动把关键字替换为相对地址值。

jmp是无条件转移

(6) 有条件转移

这里是一些汇编语言的基础，到时候用到的时候再去细看，现在没必要整的太清楚。都是在背诵概念。

3.对显卡进行硬件编程

南桥和北桥是一个仲裁及外设IO管理中心，CPU和多个IO接口通信时，为了防止冲突，需要有一层来仲裁，也就是输入输出控制中心I/O control hub，南桥。

北桥一般和高速设备通信，比如内存。

可以通过IO端口来直接操纵外设，这个是完全独立于操作系统的东西，算是微机的部分。

在IA32体系架构中，存储端口号的寄存器有16位，最大有65536个端口，0-65525

IA32没有进行内存映射，而是使用独立编址。CPU提供了专门的指令in、out来进行操作外设。

in指令从端口读取数据：

in al, dx

```
in ax, dx
```

dx就是存储端口号的寄存器，读取的数据被存储到前面的寄存器里面。

out指令向端口写数据：

```
out dx, al
```

```
out dx, ax
```

```
out 立即数, al
```

这里的用法是固定的，如果不放心，可以把dx数据存到内核栈，再用地址填充dx，即可操作外设端口。

端口号基本上只能用dx寄存器。

编写书本上的例程到MBR中，我们发现以下效果：



具有闪烁效果，还是非常舒适的。

x86架构应当也是内存映射，显存被映射到了指定的位置，直接向显存写入数据，即可完成效果哦。

下面还有一些对bochs的调试指南和硬盘的介绍，接下来主要看一看硬盘介绍。

后面借用MBR访问了磁盘，并且加载了第二扇区的loader程序，从而完成对内核的加载。

当操作系统内核加载到内存中，操作系统进入保护模式以后，我们就正式进入到了操作系统的各种机制中去了。

今天就暂时看到这里，明天继续往后看。