第一章 实验入门

> - 这个世界上有10种人，其中有些是能理解二进制的，有些则不懂。
> - 这个世界上有10种人，其中有些是能理解三进制的，有些不懂，有些则把它和二进制弄混淆了。
> - 这个世界上有10种人，其中有些是能够理解二进制和三进制的，有些不懂，有些则开始怀疑十进制了。

摘要

在第一章中，同学们会学习到计算机的启动过程，IA-32处理器架构和字符显存原理。根据所学的知识，同学们能自己编写程序，然后让计算机在启动后加载运行，以此增进同学们对计算机启动过程的理解，为后面编写操作系统加载程序奠定基础。

内容

- 选择一个适合自己的操作系统windows、linux和macos。
- 安装bochs并使用bochs自带的工具生成硬盘。
- 学习x86汇编和nasm汇编，了解基本汇编语法。
- 学习操作系统启动相关过程，了解实模式和显存工作原理，编写Hello World汇编程序 `mbr.asm`。
- 使用nasm编译器将汇编程序 `mbr.asm` 编译成二进制文件 `mbr.bin`，使用工具将 `mbr.bin` 写入硬盘首扇区。
- 编写bochs运行所需配置文件 `bochs.bxrc`，使用bochs加载首扇区运行，学习bochs的debug指令。
- 编写实验报告。

参考资料

- x86汇编(Intel汇编)入门
- 《Intel汇编语言程序设计》第1-8章
- 《从实模式到保护模式》第1-8章

IA-32处理器

从汇编开始

我们先从汇编语言的学习开始我们的操作系统实验之旅。为什么需要汇编语言？机器只能理解机器语言，机器语言是我们在《计算机组成原理》实验课上为CPU编写的指令的01串。但是，用01串编程显然不利于程序员对程序的理解。于是汇编语言和编译器诞生了，汇编语言提供了一些过程的语言抽象，如加法指令的01串抽象成 `add`，减法指令的01串抽象成 `sub`。通过这些语言抽象，也就是汇编语法，程序员可以方便地编写相对于机器语言更易于理解的程序。然后我们使用编译器将汇编代码翻译成机器语言。即便汇编代码已经隐藏了机器语言的内容，但一些条件过程 `if...else` 使用汇编代码表达会变得繁琐。此时，高级语言和相应的编译器诞生了，例如C和GCC。但为什么我们从汇编语言开始而不从高级语言开始呢？这是因为我们编写的是操作系统，需要用到一些特权指令如读写硬件端口、关/开中断。而高级语言并未提供相应的指令，而只有汇编语言才提供。同时，高级语言先被翻译或解释成汇编语言，然后汇编语言再被翻译成机器语言。因此，操作系统最开始的编写只能使用汇编语言，然后基于汇编语言封装的接口，我们才能使用高级语言来简化我们的开发过程。

汇编语言与处理器体系结构密切相关，因此在学习汇编之前，必须学习基本的处理器体系结构。我们的操作系统实验基于的是IA-32处理器。

体系结构

IA-32处理器是指从Intel 80386开始到32位的奔腾4处理器，是最为经典的处理器架构。至此，Intel 32位的处理器也被称为x86处理器。IA-32处理器其有三种基本操作模式：保护模式、实地址模式(简称实模式)、系统管理模式和虚拟8086模式。我们在操作系统实验过程中仅用到实模式和保护模式。IA-32处理器的重要组成部分如下。

- **地址空间。**保护模式和实模式最大的不同是二者的地址总线。实模式使用20位地址总线、16位寄存器；保护模式使用32位地址总线、32位寄存器。因此，实模式的寻址空间为$2^{20} = 1MB$，保护模式的寻址空间为$2^{32} = 4GB$。这里的地址指的是内存地址。
- **基本寄存器。**寄存器是CPU内部的高速存储单元。IA-32处理器主要有8个通用寄存器eax, ebx, ecx, edx, ebp, esp, esi, edi、6个段寄存器cs, ss, ds, es, fs, gs、标志寄存器eflags、指令地址寄存器eip。
- **通用寄存器。**通用寄存器有8个，分别是eax, ebx, ecx, edx, ebp, esp, esi, edi，均是32位寄存器。通用寄存器用于算术运算和数据传输。32位寄存器用于保护模式，为了兼容16位的实模式，每一个32位寄存器又可以拆分成16位寄存器和8位寄存器来访问。例如ax是eax的低16位，ah是ax高8位，al是ax的低8位。ebx，ecx，edx也有相同的访问模式。如下所示。

0-31位	0-15位	8-15位	0-7位
eax	ax	ah	al
ebx	bx	bh	bl
ecx	cx	ch	cl
edx	dx	dh	dl

> eax中的e是extended的意思。

但是，esi，edi，ebp和esp并无8位的寄存器访问方式。如下所示。

0-31位	0-15位
esi	si
edi	di
esp	sp
ebp	bp

通用寄存器有如下特殊用法，我称之为约定俗成的规则。

  - eax在乘法和除法指令中被自动使用。通常称之为扩展累加寄存器。
  - ecx在loop指令中默认为循环计数器。
  - esp用于堆栈寻址。因此，我们绝对不可以随意使用esp。

    - esi和edi通常用于内存数据的高速传送，通常称之为扩展源指针和扩展目的指针寄存器。
    - ebp通常出现在高级语言翻译成的汇编代码中，用来引用函数参数和局部变量。除非用于高级语言的设计技巧中，ebp不应该在算术运算和数据传送中使用。ebp一般称之为扩展帧指针寄存器。

> 在操作系统实验中，我们常常会看到一些约定俗成的规则。这也是造成同学们在开始操作系统实验时觉得门槛较高的原因。同学们需要了解各种各样的规则，否则程序一定会出现bug，而这些bug是无法通过逻辑推导出来。因此，debug时往往需要重新浏览一遍相关的知识。这些规则的形成是在一个特定的历史进程中完成的，例如IA-32处理器启动先进入的是16位实模式，然后由实模式跳转到32位保护模式，而arm处理器一启动便可进入32或64位的寻址模式。但是，arm处理器于90年代发布，晚于IA-32处理器20年。

- **段寄存器**。段寄存器有cs, ss, ds, es, fs, gs，用于存放段的基地址，段实际上就是一块连续的内存区域。
- **指令指针**。eip存放下一条指令的地址。有些机器指令可以改变eip的地址，导致程序向新的地址进行转移，如ret指令。
- **状态寄存器**。eflags存放CPU的一些状态标志位。下面提到的标志如进位标志实际上是eflags的某一个位。常用的标志位如下。
    - 进位标志(CF)。在无符号算术运算的结果无法容纳于目的操作数时被置1。
    - 溢出标志(OF)。在有符号算术运算的结果无法容纳于目的操作数时被置1。
    - 符号标志(SF)。在算术或逻辑运算产生的结果为负时被置1。
    - 零标志(ZF)。在算术或逻辑运算产生的结果为0时被置1。

实地址模式

> 由于实模式的寄存器是16位的，因此下面出现的寄存器不带e。

在实地址模式下，IA-32处理器使用20位的地址线，可以访问$2^{20} = 1MB$的内存，范围时0x0000到0xFFFFF。但是，我们看到寄存器的访问模式只有32位，16位和8位，形如eax，ax，ah，al。因此，我们无法直接表示20位的内存地址。这在当时也给Intel工程师带来了极大的困扰，但是聪明的工程师想出来一种“段地址+偏移地址”的解决方案。段地址和偏移地址均为16位。此时，一个1MB中的地址，称为线性地址，即实际的地址，按如下方式计算出来。

$$线性地址 = (段地址 << 4) + 偏移地址$$

此时，线性地址一般记为“段地址:偏移地址”。段地址和偏移地址用16位表示，最大值均为0xFFFF，按上述计算方式的可表示的最大地址是大于20位地址线表示的1MB内存空间的。

段地址存放在段寄存器cs, ds, es, ss中，在编程中我们给出的地址(如下面提到的数据标号和代码标号)实际上是偏移地址，当我们要寻址时，CPU会自动根据段地址和偏移地址计算出线性地址，然后使用线性地址进行寻址。

> 在操作系统实验中，我们出现的许多bug往往就出现在地址上，例如段地址指定错误导致无法访问正确的变量，函数返回了一个不正确的地址。因此，我们在学习编写汇编代码时一定要弄清楚处理器的寻址方式。

段寄存器也有约定俗成的规则。一个典型的程序有3个段，数据段、代码段和堆栈段。

- cs包含16位代码段的基地址。
- ds包含16位数据段的基地址。
- ss包含一个16位堆栈段的基地址。
- es、fs和gs可以指向其他数据段的基地址。

由于段地址必须通过段寄存器给出，因此下面直接用段寄存器来代替描述段地址。

汇编基础

nasm是Intel汇编代码的编译器，支持linux，windows平台。

> 同学们看到的一些汇编书籍如王爽《汇编语言》，《Intel汇编语言程序设计》使用的是MASM汇编。MASM是在MS-DOS下运行的。而NASM汇编是跨平台的，且二者语法有部份不同。但无论是NASM还是MASM，都属于x86汇编(或称为Intel汇编)的内容，关于汇编指令如mov，add，jmp等常用指令是相同的。因此，如果下面的内容和nasm无关，则我会标注为x86汇编；否则我会标注为nasm汇编。

汇编代码一般保存在以 `.s` 或 `.asm` 为后缀的文件中。我们先从16位实模式编程开始我们的汇编之旅，基于IA-32处理器的x86汇编用到的主要寄存器如下所示。

寄存器	作用
ax	累加寄存器
cx	计数寄存器
dx	数据寄存器
ds	数据段寄存器
es	附加段寄存器
bx	基地址寄存器
si	源变址寄存器
di	目的变址寄存器
cs	代码段寄存器
ip	指令指针寄存器
ss	栈段寄存器
sp	栈指针寄存器
bp	基指针寄存器
flags	标志寄存器

x86汇编 注释

在汇编代码中，我们使用；来实现注释。；之后的和；位于同一行的字符都会被nasm编译器当成是注释的内容。例如

```
1    add eax, 3 ; 注释：定义一个初始值为3的一个字节变量
```

> 在汇编代码中，一行只能写一条汇编语句且无需以任何符号结尾。

nasm汇编 标识符

标识符是我们取的名字，用来表示变量、常量、过程或代码标号。创建标识符需要注意以下几点。

- 标识符和包含1-247个字符。
- 标识符大小写不敏感。
- 标识符的第一个字符必须是字母、下划线或@，后续字符可以是数字。
- 标识符不能与汇编器的保留字相同。

下面是一些有效的标识符。

```
varl
Count
_main
MAX
open_file
@myfile
```

nasm汇编 标号

标号是充当指令或数据位置标记的标识符，也就是说，标号的值就是其后的指令或数据位置的首地址。

数据标号。数据标号标识了变量的地址，为在代码中引用该变量提供了方便。在nasm汇编中，变量的类型有3种，如下所示。

数据类型	含义
db	一个字节
dw	一个字，2个字节
dd	双字，4个字节

数据标号的定义如下。

```
count dw 100
```

在一个标号后可以定义多个相同类型的数据项，数据项之间以`，`分隔。标号实际上是这些数据项的起始地址，类似于C语言中的数组。

```
array dw 1024, 2048
      dw 4096, 8192
```

上面的数据项按1024，2048，4096，8192的顺序存放，每个数据项占2个字节。

代码标号。代码标号标识了汇编指令的起始地址，通常作为跳转指令的操作数。操作数是指令的操作对象。例如加法指令的加数和被加数是加法指令的操作数。

```
L1:
    mov ax, bx
    ... ; 此处省略若干条指令
    jmp L1
```

注意，换行、空格并不会影响代码标号的值。即下面的 `L1` 和上面的 `L1` 的值相同。但代码标号后面必须要有 `:`，数据标号后面不能有 `:`。

```
L1: mov ax, bx
    ... ; 此处省略若干条指令
    jmp L1
```

x86汇编 数据传送指令

首先约定操作数的表示形式。

符号	含义
<reg>	寄存器，如ax，bx等
<mem>	内存地址，如标号var1，var2等
<con>	立即数，如3，9等

数据传送指令主要介绍 `mov` 指令。 `mov` 指令是将源操作数的内容复制到目的操作数中，用法如下。

```
mov <reg>, <reg>
mov <reg>, <reg>
mov <reg>, <mem>
mov <mem>, <reg>
mov <reg>, <con>
mov <mem>, <con>
```

在Intel汇编中，如果指令的操作数有两个，则目的操作数是第一个，源操作数是第二个。例如，对于下面的一条指令。

```
1    mov eax, ebx
```

其中，源操作数是ebx，目的操作数是eax，指令是把ebx的内容复制到eax中。

在汇编指令中，我们需要注意内存地址的使用。如果对内存地址的使用不明确，则程序一定会出现bug。因此，在介绍其他指令之前，下面先介绍内存地址的寻址方式。

nasm汇编 内存寻址方式

同学们已经在《计算机组成原理》课上学习到了许多的内存寻址方式，这里的寻址方式实际上和指令的寻址方式大致相同。寻址方式指的就是我们应该到哪里去取出我们的操作数。寻址方式主要有以下几种方式。

- **寄存器寻址。**寄存器寻址指的是操作数存放在寄存器中，如下所示，源操作数和目的操作数均存放在寄存器中。

  ```
  1    mov ax, cx ; ax = cx
  ```

- **立即数寻址。**操作数以立即数的方式出现在指令中，如下所示。

  ```
  1    mov ax, 17 ; ax = 17
  2    mov ax, tag ; ax = tag
  ```

  注意，tag是一个标号，如数据标号和代码标号。前面已经讲过，标号实际上就是指令或变量的起始地址。因此，在第2条指令中，我们是将tag表示的地址存放到了寄存器ax中。

- **直接寻址。**操作数存放在内存中，其偏移地址由立即数给出。例如，我们有一个16位的变量存放在起始地址位0x5c00处，其值为0xFF。我们需要将其读入到寄存器ax中，指令如下所示。

  ```
  1    mov ax, [0x5c00] ; ax = 0xFF
  ```

  标号是汇编地址，如果标号`tag`表示的是我们变量的起始地址，则也可以写成如下形式。

  ```
  1    mov ax, [tag] ; ax = 0xFF
  ```

  在根据偏移地址去取内存中的变量时，不要忘记加上`[]`，否则我们只是将变量地址放入到寄存器中。例如

  ```
  1    mov ax, tag ; ax = tag
  ```

此时ax的内容是tag的值，而不是tag这个地址指向的变量的值。我们前面提到过，在实模式中，变量的实际地址称为线性地址，线性地址的计算式子如下。

$$线性地址 = 段地址 << 4 + 偏移地址$$

但是上面我们给出的是偏移地址，那么段地址去了哪里呢？这就涉及到我反复强调的一些约定俗成的规则。我们指令中如果没有显式指定段地址，那么我们的地址就是偏移地址。此时，CPU在计算线性地址时用到了默认的段寄存器，规则如下。

- 访问数据段，使用段寄存器ds。
- 访问代码段，使用段寄存器cs。
- 访问栈段，使用段寄存器ss。

当然，我们也可以使用“段地址:偏移地址”的形式指定段地址，此时CPU不使用默认段寄存器的段地址，而是使用指令给出的段地址。例如，我们可以指定段地址为 `es` 段寄存器的内容。

```
mov ax, [es:tag]
```

因此，下面两条语句是等价的。

```
mov ax, [tag]
mov ax, [ds:tag]
```

因为上面的两条语句属于数据段，所以默认寄存器是 `ds` 。我们在使用跳转指令 `jmp` 或 `call` 时，若只给出偏移地址，那么默认段地址就是段寄存器 `cs` 的内容。如果偏移地址是由栈指针 `bp` 或 `sp` 给出的，那么默认段地址就是段寄存器 `ss` 的内容。

- **基址寻址**。基址寻址使用基址寄存器和立即数来构成真实的偏移地址。基址寻址类似于数组的寻址，基址寄存器只能是寄存器 `bx` 或 `bp` ，然后加上立即数共同构成偏移地址。使用 `bx` 做基址寄存器时，段地址寄存器默认为 `ds` ，使用 `bp` 时默认为 `ss` 。如下所示。

```
; 使用bx做基址寄存器时段寄存器为ds存放的内容
mov [bx], ax
mov ax, [bx]
mov [bx + 3], ax
mov ax, [bx + 3]
mov [bx + 3 * 4], ax
mov ax, [bx + 3 * 4]
; 使用bp做基址寄存器时段寄存器为ss存放的内容
mov [bp], ax
mov ax, [bp]
mov [bp + 3], ax
mov ax, [bp + 3]
mov [bp + 3 * 4], ax
mov ax, [bp + 3 * 4]
```

- **变址寻址**。变址寻址使用变址寄存器和立即数来构成真实的偏移地址。变址寄存器只能是`si`或`di`，默认段寄存器为`ds`。事实上，变址寻址和基址寻址类似，但很快我们可以看到，我们可以将二者结合起来寻址。变址寻址如下所示。

```
1    mov ax, [si + 4 * 4]
2    mov [di], 0x5
```

- **基址变址寻址**。我们通过基址寄存器、变址寄存器、立即数来构成真实的偏移地址。默认段地址由基址寄存器的类型确定，即`bx`对应`ds`、`bp`对应`ss`，如下所示。

```
1    mov [bx + si + 5 * 4], ax
2    mov [bx + di + 5 * 4], ax
3    mov ax, [bx + si + 5 * 4]
4    mov ax, [bp + si + 5 * 4]
5    mov ax, [bp + di + 5 * 4]
```

至此，寻址方式已经讲述完毕。

前面已经讲过，汇编程序的许多错误实际上是由于寻址错误造成的，即CPU从一个错误的地址取了数据。若操作数存放在内存中时，同学们在寻址时千万不要漏掉`[]`，否则取出的只是数据的首地址。同时，当我们的地址没有指定段寄存器时，即`es:0x500`这种形式，那么此时这个地址是偏移地址，段地址存放在默认的段寄存器中。

基址寻址、变址寻址和基址变址寻址有一个共同点就是地址在寄存器中给出，这就给我们循环遍历数组带来方便。考虑如下一个数组。

```
1    array db 1, 2, 3, 4
```

我们希望依次将数组的4个元素放入到ax中，程序如下。

```
1    array db 1, 2, 3, 4
2    mov cx, 4
3    mov bx, 0
4    visit_array:
5        mov ax, [bx + array]
6        add bx, 1
7        loop visit_array
```

我们重点看`mov ax, [bx + array]`。`bx`每次循环都会加1，初始值为0，则ax被依次`mov`的值为1,2,3,4。上面这个程序告诉我们，基址寻址、变址寻址和基址变址寻址在循环访问数组等场景非常方便。注意，基址寄存器和变址寄存器也是约定俗成的。因此下面这条语句是无法通过nasm汇编器编译的。

```
mov ax, [cx]
```

x86汇编 算术和逻辑指令

算术和逻辑指令主要如下。

add指令是将两个操作数相加，并将相加后的结果保存到第一个操作数中，语法如下。

```
add <reg>,<reg>
add <reg>,<mem>
add <mem>,<reg>
add <reg>,<con>
add <mem>,<con>
; e.g.
add ax, 10 ; eax := eax + 10
add byte[tag], al
```

sub指示第一个操作数减去第二个操作数，并将相减后的值保存在第一个操作数，语法如下。

```
sub <reg>,<reg>
sub <reg>,<mem>
sub <mem>,<reg>
sub <reg>,<con>
sub <mem>,<con>
; e.g.
sub al, ah ; al := al - ah
sub ax, 126
```

inc，dec指令分别表示自增1或自减1，语法如下。

```
inc <reg>
inc <mem>
dec <reg>
dec <mem>
; e.g.
dec ax
inc byte[tag]
```

and, or, xor分别表示将两个操作数逻辑与、逻辑或和逻辑异或后放入到第一个操作数中，语法如下。

```
```

```
and <reg>,<reg>
and <reg>,<mem>
and <mem>,<reg>
and <reg>,<con>
and <mem>,<con>

or <reg>,<reg>
or <reg>,<mem>
or <mem>,<reg>
or <reg>,<con>
or <mem>,<con>

xor <reg>,<reg>
xor <reg>,<mem>
xor <mem>,<reg>
xor <reg>,<con>
xor <mem>,<con>
```

not表示对操作数每一位取反，语法如下。

```
not <reg>
not <mem>
; e.g.
not ax
not word[tag] ; 取反一个字,2个字节
not byte[tag] ; 取反一个字节
not dword[tag] ; 取反一个双字，4个字节
```

neg表示取负，语法如下。

```
neg <reg>
neg <mem>
```

shl,shr表示逻辑左移和逻辑右移，即空出的位补0，语法如下。

```
; cl是寄存器ecx的低8位寄存器

shl <reg>,<con>
shl <mem>,<con>
shl <reg>, cl
shl <mem>, cl

shr <reg>,<con>
shr <mem>,<con>
shr <reg>, cl
shr <mem>, cl
```

x86汇编 控制转移指令

我们知道，程序是顺序执行的。但是有时候我们的程序需要处理条件分支逻辑if...else或循环逻辑while等，此时，我们就需要改变程序的顺序执行逻辑。计算机每次都会从eip中取出下一条指令的地址，然后按地址取指令执行。因此，如果我们需要改变程序的执行逻辑，则需要改变eip寄存器的内容。但是，我们并不允许直接修改eip寄存器的内容，需要通过控制转移指令来完成。

jmp指令是无条件跳转指令，跳转到代码标号<label>的指令处执行，语法如下。

```
jmp <label>
```

jcondition是有条件跳转指令的统称，其根据机器状态寄存器eflags的内容来判断时候执行跳转，语法如下。一般来说，在jcondition语句之前都会紧邻一个cmp指令。

```
je <label>    ; jump when equal
jne <label>   ; jump when not equal
jz <label>    ; jump when last result was zero
jg <label>    ; jump when greater than
jge <label>   ; jump when greater than or equal to
jl <label>    ; jump when less than
jle <label>   ; jump when less than or equal to
```

cmp指令是将第一个操作数减去第二个操作数，并根据比较结果设置机器状态寄存器eflags中的条件码，用法如下。

```
cmp <reg>,<reg>
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>
```

一般来说，jcondition指令是和cmp指令搭配使用的，而且上面对jcondition的解释也是在搭配使用的环境下给出的直观解释，考虑下面这个用法。

```
cmp byte[tag], 10
jeq label
sub ax, 10
label: add ax, 10
```

这个用法的意思是说，比较标号tag处的变量和10，如果不等于，则跳转到标号 `label` 处执行，即执行语句 `add ax, 10`；否则，先执行 `sub ax, 10`，然后再执行 `add ax, 10`。

> 从这里例子我们可以看到，在 `jcondition` 指令和 `cmp` 指令搭配使用的环境下，我们可以不用去关心eflags的内容才能判断出程序的跳转逻辑。此时程序的跳转逻辑就是 `jcondition` 的字面含义，如本例的”jeq”的字面含义是“不等于就跳转”。条件跳转语句的用法需要同学们仔细体会。

x86汇编 栈操作指令

栈是一种后进先出的数据结构，在x86汇编中，栈的增长方式是从高地址向低地址增长。栈的操作指令有以下4条。

push指令是将操作数压入内存的栈中。

```
push <reg>
push <mem>
push <con>
```

pop指令将栈顶的数据放入到操作数中。

```
pop <reg>
pop <mem>
```

pushad指令是将ax, cx, dx, bx, sp, bp, si, di 依次压入栈中。由于栈是从高地址向增长地址，因此ax的数据位于高地址，di的数据位于低地址。

```
pushad ; 无操作数
```

`popad` 指令是对栈指令一系列的pop操作，pop出的数据放入到di, si, bp, sp, bx, dx, cx, ax中。

```
popad ; 无操作数
```

我们已经知道，sp被约定俗成为栈指针。这是因为我们执行栈操作命令时，实际上是对sp寄存器及其保存的地址进行操作，例如。

```
; 下面语句等价于 push ax
sub sp, 2          ; 从高地址向低地址增长，16位实模式
mov [sp], ax
;--------------------------
; 下面语句等价于 pop ax
mov ax, [sp]
add sp, 4
```

x86汇编 过程调用

call和ret指令是用来实现子过程(或者称函数，过程，意思相同)调用和返回。call指令首先将当前eip的内容入栈，然后将操作数的内容放入到eip中。ret指令将栈顶的内容弹出栈，放入到eip中。用法如下所示。

```
call my_function
add ax, 10
jmp $
; $在nasm汇编中表示当前地址，即jmp $指令开始的地址
; 因此这条语句实际上是在做死循环，程序到这里就不会往下执行了

my_function:
    sub ax, 10
    sub bx, 10
    ret
```

第1行使用call指令将eip的内容压栈后跳转到 `my_function` 标号表示的地址处执行，然后执行第8，9, 10行的语句，执行完ret指令后，栈顶的内容即call压栈的内容被传送到eip中。此时，程序执行call指令的下一条指令，即第2行的指令，最后程序在第3行做死循环。

实际上过程调用后，进入到子过程时，我们首先需要对子过程将要用到的寄存器进行压栈，最后在ret之前通过pop指令进行恢复，以达到执行子过程不会对调用者造成任何影响的效果。如果在子过程中不对子过程用到的寄存器进行进入压栈，退出弹栈的操作，那么子过程返回调用者时，调用者使用的就是子过程修改过的数据，这是非常危险的。因此，上面的程序修改如下。

```
call my_function
add ax, 10
jmp $
; $在nasm汇编中表示当前地址，即jmp $指令开始的地址
```

```
5   ; 因此这条语句实际上是在做死循环，程序到这里就不会往下执行了
6
7   my_function:
8       push ax
9       push bx
10
11      sub ax, 10
12      sub bx, 10
13      ; 后进先出
14      pop bx
15      pop ax
16
17      ret
```

> 进入压栈一定要记得退出弹栈，否则ret指令弹出的栈顶内容将不会是call指令压入的返回地址。并且弹栈是切记后进先出的顺序。

计算机开机启动过程

小时候我曾经看过一部名为猪猪侠的动漫，在其中一集中，猪猪侠被一位大力士抓住了。大力士力大无比，自认为能举起宇宙中任何一件东西。于是大力士告诉猪猪侠，如果猪猪侠能告诉大力士世界上有什么东西是大力士举不起来的，那么大力士就放了猪猪侠，否则就要吃掉猪猪侠。猪猪侠想了想说：“你不能举起你自己！”大力士听了后为自己的狂妄而感到羞愧，于是便放了猪猪侠。

计算机是一个大力士，自认为能够运行任何程序。但是，计算机的启动需要程序加载，而计算机不启动则无法运行程序。也就是计算机这个大力士需要将自己举起来。于是聪明的工程师在早期想尽各种办法，把一小段程序装进内存，然后计算机才能正常运行。直到后面ROM(Read-Only memory)的发明才真正解决了这个问题。

经典的计算机的启动分为以下步骤。

> 经典的启动方式是指下面的计算机启动过程是x86架构下的计算机BIOS启动过程，而UEFI启动或在arm架构下则是另外一种启动方式。

1. **加电开机。**按下电源的开关，电源马上开始向主板和其它的设别开始供电。此时的电压还不是很稳定，主板上的控制芯片组会向CPU发出并保持一个reset（重置）信号，让CPU内部自动恢复到初始状态下。当芯片组检测到电源已经开始稳定的供电了，芯片组则开始撤去reset信号。此时，CPU马上开始从0xFFFF0处执行指令。这个地址位于系统的BIOS的地址范围内，其实放在这里的只是一条跳转指令，指向BIOS中真正的启动代码地方。BIOS，基本输入输出系统（Basic Input Output System），是一组固化到计算机内主板上一个ROM（Read-Only Memory）只读存储器。BIOS保存着计算机最重要的基本输入输出的程序、系统设置信息、开机上电自检程序和系统启动自检程序。
2. **BIOS启动。**BIOS启动后，第一件事情就是执行POST(Power-On-self-test)自检阶段，主要针对系统的一些关键设备是否存在或者是功能是否正常，如：内存、显卡等。如果在POST过程

中系统设备存在致命的问题，BIOS将会发出声音来报告检测过程中出现的错误，声音的长短及次数对应着系统的错误类型。POST过程会非常快速，对用户几乎感觉不出来。

3. **加载MBR。**BIOS按照设定好的启动顺序，将控制权交给排在第一位的存储设备，即设备的首扇区512字节，称为MBR(Master Boot Record, 主引导扇区)，并且将这512字节复制到放在0x7c00的内存地址中运行。存储设备一般分为若干个固定大小的块来访问，这个固定大小的块被称为扇区，而第1个扇区被称为首扇区。但在复制之前，计算机会根据MBR判断设备是不是可启动的，即有无操作系统。判断依据是检查MBR最后两个字节是否为0x55,0xAA。
4. **硬盘启动。**MBR只有512字节大小，程序可处理的逻辑有限。因此MBR会从存储设备中加载bootloader(启动管理器)，bootloader并无大小限制。bootloader的作用是初始化环境，然后从存储设备加载kernel(操作系统内核)到内存中。
5. **内核启动。**kernel加载入内存后，bootloader跳转到kernel处执行。至此，计算机启动完毕。

我们需要编写的内容是MBR，bootloader和kernel，而BIOS启动，POST，MBR被加载到0x7c00的过程由计算机自动完成。

> 为什么MBR是被加载到0x7c00，而不是加载到0x0000？有兴趣的同学可以参考Why BIOS loads MBR into 0x7C00 in x86 ?

环境的准备

Bochs

在学习完汇编和了解了计算机的启动过程之后，我们已经可以开始编写我们的存放在MBR的第一个程序。但是，我们是在做一个操作系统原型，我们的第一个程序是写入MBR的，因此我们不可能将这个程序写入到我们自己主机上的硬盘(有兴趣的同学可以尝试一下)。同时，我们现在的电脑CPU架构是x86_64架构，不兼容我们IA-32架构下的操作系统。因此，我们需要模拟出IA-32处理器的硬件环境，所用到的模拟器是bochs。

> 另外一个比较出名的模拟器是qemu，但qemu在笔者的电脑上的运行速度远不及bochs，并且调试过程比bochs繁琐。不过有兴趣的同学可以尝试下qemu，上面的陈述只是我的个人感受而已，而且清华大学的ucore(一个类似的操作系统教程)用的模拟器是qemu。

bochs是以GNU宽通用公共许可证发放的开放源代码的x86、x86-64、IBM PC兼容机模拟器和调试工具，支持处理器（包括保护模式）、内存、硬盘、显示器、以太网、BIOS、IBM PC兼容机的常见硬件外设的仿真。同学们首先去SF上将bochs下载下来并安装，我们待会需要用到bochs。

nasm

我们可以去nasm的官网上将nasm下载下来，然后安装或解压。

Hello World

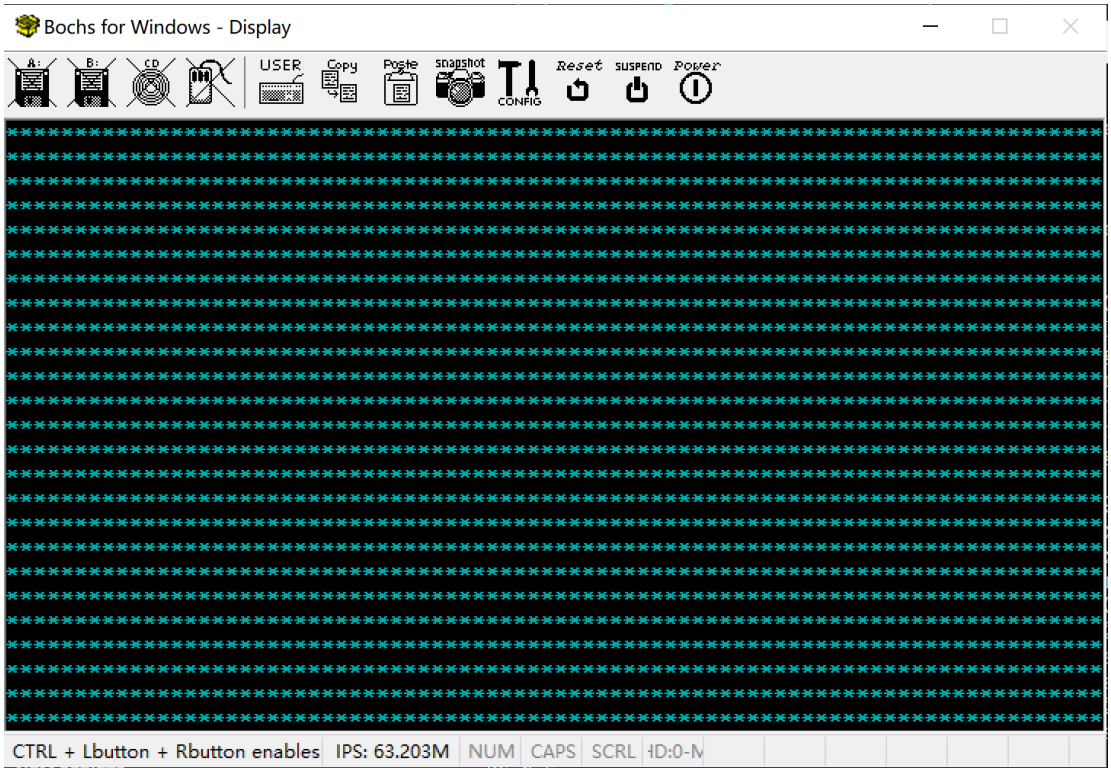
纸上得来终觉浅，绝知此事要躬行。我们现在终于可以开始写汇编代码了，我们需要做的任务如下。

> **Assignment**: 在MBR被加载到内存地址0x7c00后，向屏幕输出蓝色的Hello World。

字符显示的原理

我们已经知道，只要将我们的程序放入存储设备首扇区后，那么当计算机在加电启动时，计算机就会自动加载首扇区的512字节到内存地址0x7c00处执行。因此，我们现在关心的问题就是如何才能向屏幕输出字符。

我们的显示屏实际上是按25x80个字符来排列的矩阵，我称之为显示矩阵，如下所示。

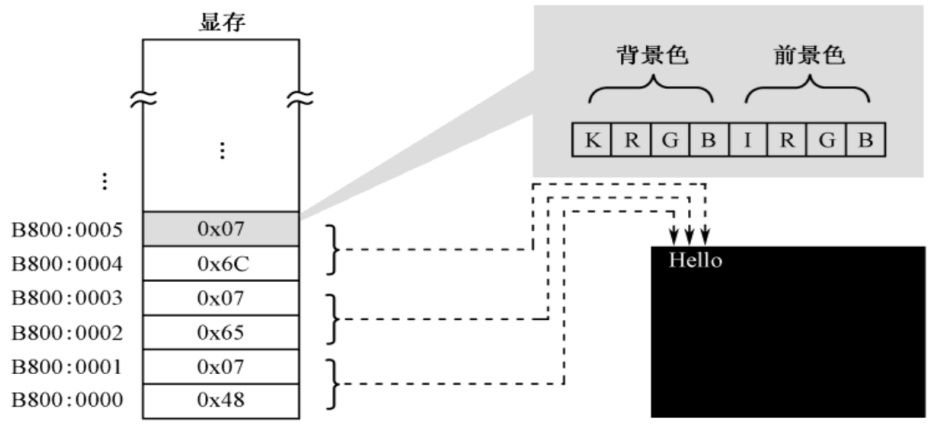


上面这张图是我用 * 填满了显示屏的各个显示位置而产生的。可以看到，25x80个字符的显示矩阵指的是横向80个字符，纵向25个字符。我们不妨记(x, y)为显示矩阵中的一个点，则我们有

$$x = 0, 1, \cdots, 24$$
$$y = 0, 1, \cdots, 79$$

如果我们想在第7行，第17列输出字符时，我们只需要将字符放入到显示矩阵的$(7, 17)$这个点即可。那么接下来的问题是我们如何将字符放入显示矩阵。

为了便于控制显示，IA-32处理器将显示矩阵映射到内存地址0xB8000~0xBFFFF处，这段地址称为显存地址。在文本模式下，控制器的最小可控制单位为字符。每一个显示字符自上而下，从左到右依次使用0xB8000~0xBFFFF中的两个字节表示。其中，低字节表示显示的字符，高字节表示字符的颜色属性，如下所示。



字符的颜色属性的字节高8位表示背景色，低八位表示前景色，如下所示。

R	G	B	背景色	前景色	
			K=0 时不闪烁，K=1 时闪烁	I=0	I=1
0	0	0	黑	黑	灰
0	0	1	蓝	蓝	浅蓝
0	1	0	绿	绿	浅绿
0	1	1	青	青	浅青
1	0	0	红	红	浅红
1	0	1	品（洋）红	品（洋）红	浅品（洋）红
1	1	0	棕	棕	黄
1	1	1	白	白	亮白

在上面的对显示矩阵的点的描述中，我们使用的是二维的点，但对应到显存是一维的，因此我们需要进行维度的转换，即显示矩阵的点(x, y)对应到显存的起始位置如下所示。

$$显存起始位置 = 0\text{xB8000} + 2 \cdot (80 \cdot x + y)$$

其中，(x, y)表示第x行第y列，公式中乘2的原因是每个显示字符使用两个字节表示。

我们来看个具体例子，在上面输出“Hello”的图中，我们在第0行第1列输出了背景色为黑色，前景色为白色的字符e，那么对应到显示矩阵的点是$(0, 1)$，此时显存的起始位置如下所示。

$$显存起始位置 = 0\text{xB8000} + 2 \cdot (80 \cdot 0 + 1) = 0\text{xB8002}$$

由于背景色是黑色且不闪烁，则表示颜色属性的字节的高4位是0x0；前景色为白色，则表示颜色属性的字节的低4位是0x7，因此颜色属性字节是0x07，此时我们将字符e放入地址0xB8002，将颜色属性0x07放入地址0xB8003，此时我们就可以在屏幕上看到背景色为黑色，前景色为白色的字符e了。

生成一个硬盘

讲到这里我们已经可以开始编写我们的Assignment代码了，但在此之前我们需要一个可以运行我们代码的环境。我们的存储设备是硬盘，因此我们用bochs安装目录下的工具 `bximage` 来生成一个硬盘。

> bochs的安装目录可能与同学们的安装方式、系统相关，注意自己找一下。

我们首先运行`bximage`，按如下步骤输入。

1. 输入1，表示要新建软盘或硬盘。
2. 输入hd，表示生成一张硬盘。生成的硬盘实际上就是一个指定大小的文件。
3. 输入flat，表示我们生成的是flat格式的硬盘。
4. 输入512，表示扇区大小为512.
5. 输入10，表示我们的硬盘大小为10MB。
6. 输入你喜欢的硬盘命名，例如我的输入hd.img。

然后我们看到在当前目录下生成了硬盘hd.img，过程如下所示。

```
========================================================================
                                bximage
  Disk Image Creation / Conversion / Resize and Commit Tool for Bochs
         $Id: bximage.cc 13481 2018-03-30 21:04:04Z vruppert $
========================================================================

1. Create new floppy or hard disk image
2. Convert hard disk image to other format (mode)
3. Resize hard disk image
4. Commit 'undoable' redolog to base image
5. Disk image info

0. Quit

Please choose one [0] 1

Create image

Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd] hd

What kind of image should I create?
Please type flat, sparse, growing, vpc or vmware4. [flat] flat

Choose the size of hard disk sectors.
Please type 512, 1024 or 4096. [512] 512

Enter the hard disk size in megabytes, between 10 and 8257535
[10] 10

What should be the name of the image?
[c.img] hd.img

Creating hard disk image 'hd.img' with CHS=20/16/63 (sector size = 512)

The following line should appear in your bochsrc:
  ata0-master: type=disk, path="hd.img", mode=flat
(The line is stored in your windows clipboard, use CTRL-V to paste)

Press any key to continue
```

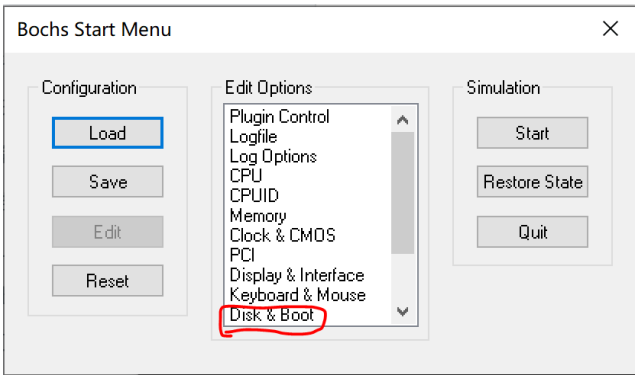
请注意如下输出的信息。

```
1    Creating hard disk image 'hd.img' with CHS=20/16/63 (sector size = 512)
```

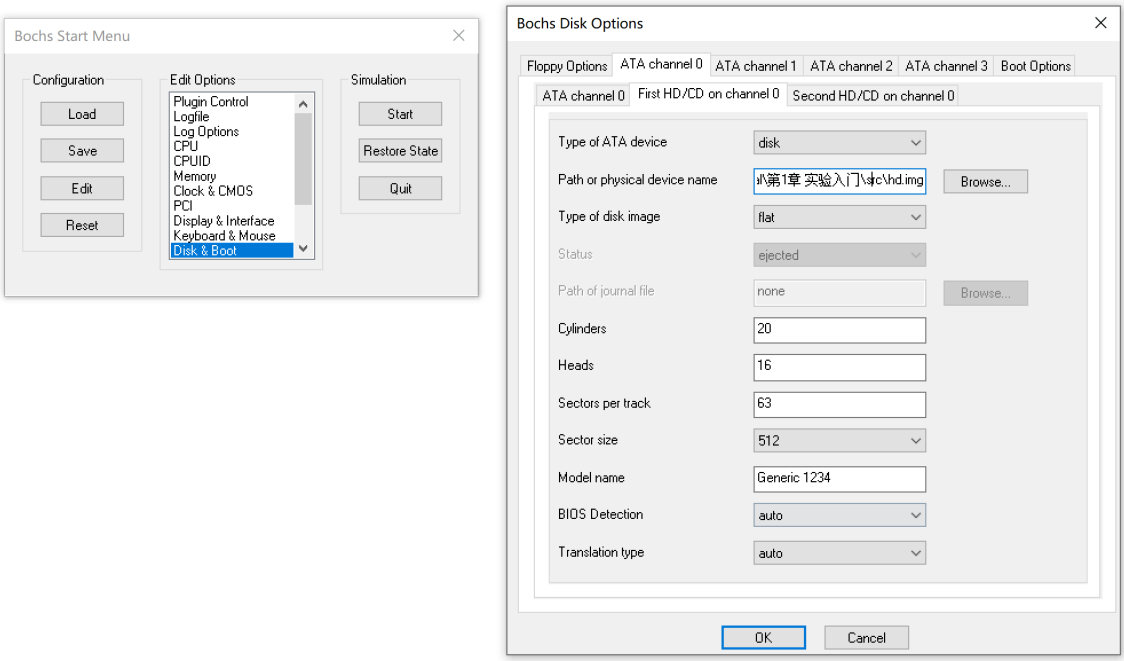
里面的CHS和sector size是磁盘的信息，我们等下配置bochs时需要用到。

配置bochs模拟环境

我们双击bochs运行，点击`Disk&Boot`配置硬盘信息和启动顺序，后面bochs启动是会根据我们配置的信息来加载MBR。



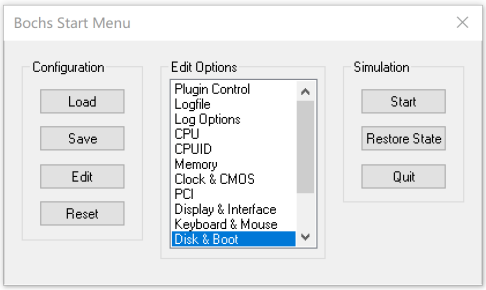
按之前根据bximage生成的信息填入，如果同学们按自己的参数生成硬盘，请以你的为准。

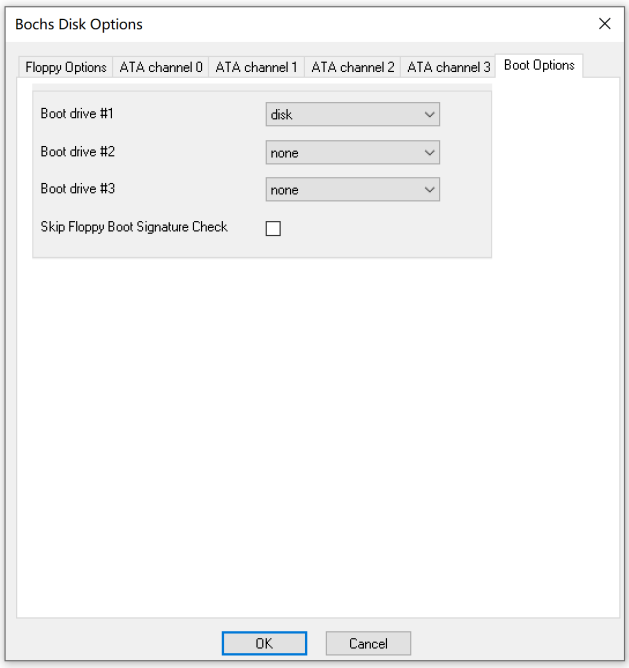


各参数解释如下。

- Path or physical device name：硬盘地址，也就是我们用bximage生成的img文件的地址。
- type of disk image: 硬盘格式，我使用的是flat格式。
- Cylinders/Heads/Sector per track/Sector size: 对应bximage最后输出的信息”CHS=20/16/63 (sector size = 512)”

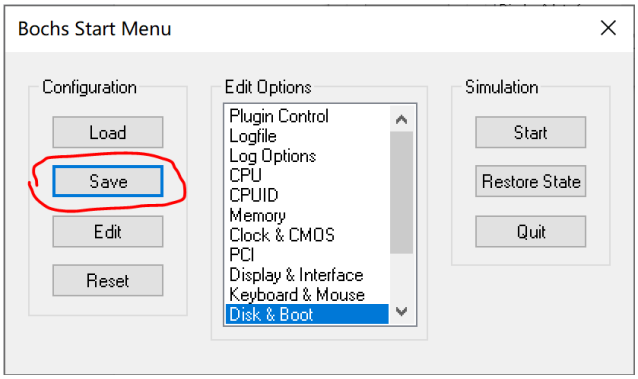
然后设置硬盘启动。



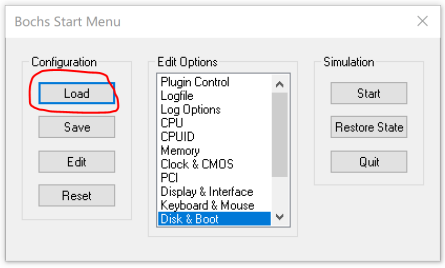


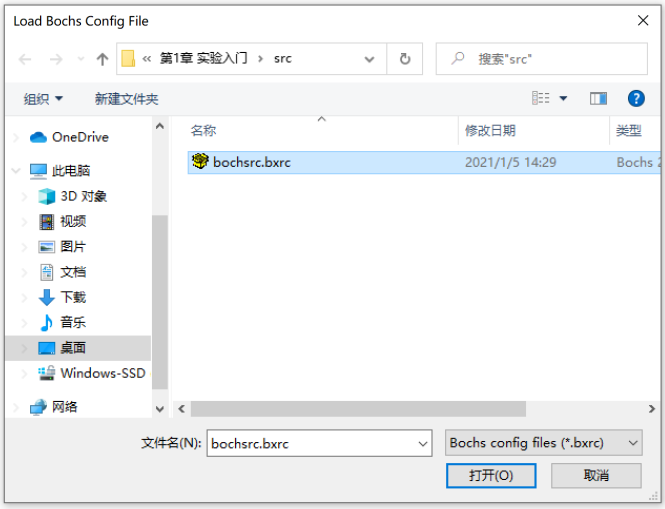
点击OK，完成配置。

由于上次配置的信息在下次启动bochs时不会被保存，为了避免每次都要重复配置，我们将本次配置的信息保存下来。点击save即可保存配置信息。

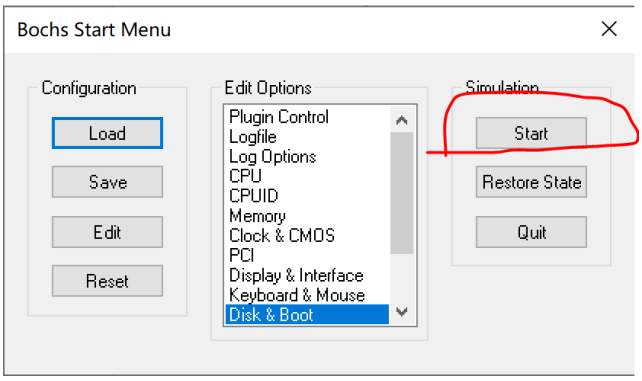


下次打开bochs时直接load即可，无需重新配置。

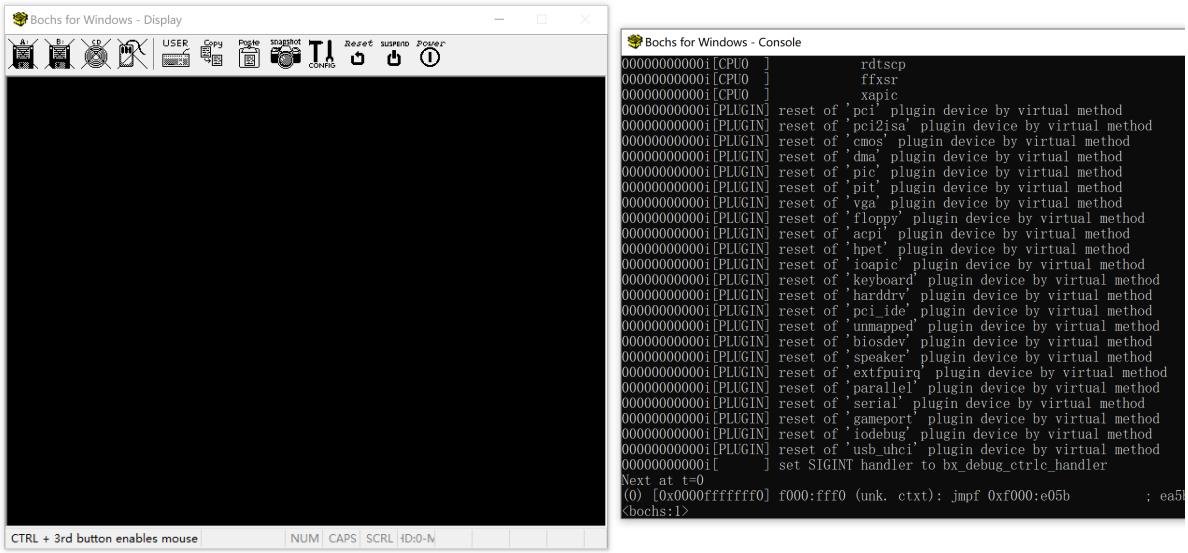




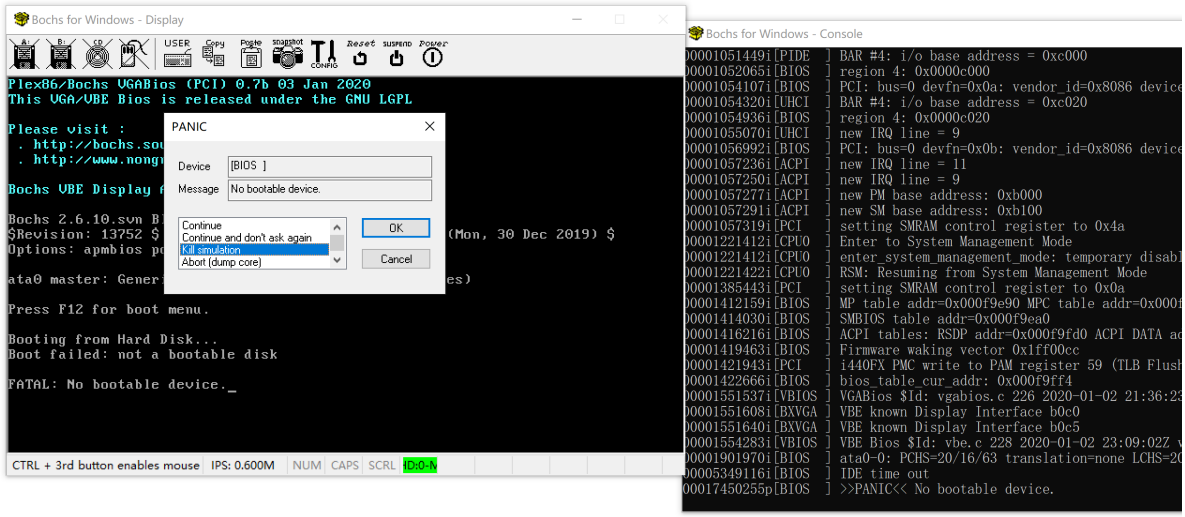
load之后点击stat即可启动。



start之后会出现两个窗口，一个是显示屏，一个是输入debug指令的。



我们在debug窗口输入c，表示continue，让计算机模拟启动然后运行。



为什么会出现“No bootable device”？这是因为计算机在启动后会加载MBR到0x7c00处运行，但在加载之前会检查MBR的最后两个字节是否为0x55,0xaa。如果是，则加载；否则认为是“No bootable device”。

简易Hello World

所以我们现在来编写我们MBR的代码。

> **Assignment**: 在MBR被加载到内存地址0x7c00后，向屏幕输出蓝色的Hello World。

```
[bits 16]
xor ax, ax ; eax = 0
; 初始化段寄存器, 段地址全部设为0
mov ds, ax
mov ss, ax
mov es, ax
mov fs, ax
mov gs, ax

; 初始化栈指针
mov sp, 0x7c00
mov ax, 0xb800
mov gs, ax


mov ah, 0x01 ;蓝色
mov al, 'H'
mov [gs:2 * 0], ax

mov al, 'e'
mov [gs:2 * 1], ax

mov al, 'l'
mov [gs:2 * 2], ax

mov al, 'l'
mov [gs:2 * 3], ax

mov al, 'o'
mov [gs:2 * 4], ax

mov al, ' '
mov [gs:2 * 5], ax

mov al, 'W'
mov [gs:2 * 6], ax

mov al, 'o'
mov [gs:2 * 7], ax

mov al, 'r'
```

```
42  mov [gs:2 * 8], ax
43
44  mov al, 'l'
45  mov [gs:2 * 9], ax
46
47  mov al, 'd'
48  mov [gs:2 * 10], ax
49
50  jmp $ ; 死循环
51
52  times 510 - ($ - $$) db 0
53  db 0x55, 0xaa
```

第1行的 `[bits 16]` 是告诉编译器按16位代码格式编译代码，不是实际的指令，因此指令是从第2行的 `xor ax, ax` 开始执行。

我们先将ax置为0，然后借助于ax将段寄存器清0。由于汇编不允许使用立即数直接对段寄存器赋值，所以我们需要借助于ax。

段寄存器初始化后，我们开始对显存地址赋值。由于显存地址是从 `0xB8000` 开始，而16位的段寄存器最大可表示 `0xFFFF`，因此我们需要借助于段寄存器来寻址到 `0xB8000` 处的地址。于是我们将段寄存器 `gs` 的值赋值为 `0xB800`。注意我们赋值的是 `0xB800` 而不是 `0xB8000`，同学们可以自行思考下原因。

然后我们依次对显存地址赋值来实现在显示屏上输出 `Hello World`。根据显存的显示原理，一个字符使用两个字节表示。因此，我们将ax的高字节部份ah赋值为颜色属性0x01，低字节部份赋值为对应的字符，然后依次放置到显存地址的对应位置。我们在对显存地址赋值时指定了段寄存器gs，因此CPU不会使用默认的段寄存器来计算线性地址。例如，我们想在第0行第1列输出蓝色字符e。

```
1   mov al, 'e'
2   mov [gs:2 * 1], ax
```

此时的线性地址的计算过程如下。

$$\text{线性地址} = \text{gs} << 4 + 2 * 1 = \text{0xB800} << 4 + 2 * 1 = \text{0xB8002}$$

恰好是对应的显存地址。

依次输出字符后，我们还没有实现下一步的工作，即bootloader加载内核。因此这里就在做死循环。代码的最后的 `times` 指令是汇编伪指令，表示重复执行指令若干次。 `$` 表示当前汇编地址， `$$` 表示代码开始的汇编地址。 `times 510 - ($ - $$) db 0` 表示填充字符0直到第510个字节。最后我们填充0x55，0xaa表示MBR是可启动的。

写完代码后我们使用nasm汇编器来将代码编译成二进制文件。

```
nasm -f bin mbr.asm -o mbr.bin
```

其中，`-f` 参数指定的是输出的文件格式，`-o` 指定的是输出的文件名。`mbr.bin` 中保存的是机器可以识别的机器指令。同学们可以使用诸如winhex等二进制文件查看器查看其中的内容。

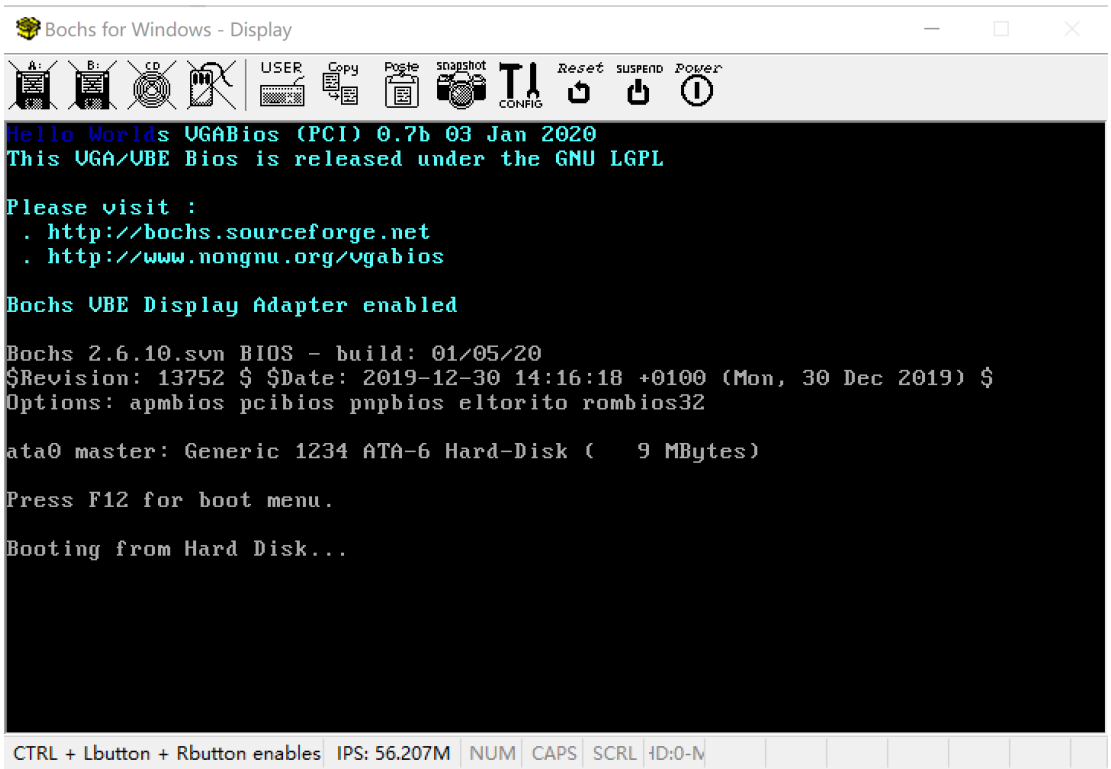
生成了MBR后，我们将其写入到硬盘的首扇区，写入的命令使用的是linux下的 `dd` 命令。windows下并没有 dd 命令，但我们可以到这里下载对应的程序，效果和使用方式和dd命令相同。我们使用dd命令将mbr.bin写入硬盘首扇区。当然，同学们可以使用图形化操作工具winhex等来写入。

```
.\dd.exe if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
```

参数的解释如下。

- `if` 表示输入文件。
- `of` 表示输出文件。
- `bs` 表示块大小，以字节表示。
- `count` 表示写入的块数目。
- `seek` 表示越过输出文件中多少块之后再写入。
- `conv=notrunc` 表示不截断输出文件，如果不加上这个参数，那么硬盘在写入后多余部份会被截断。

写入MBR后我们就可以启动bochs来模拟计算机启动了，启动后的效果如下。可以看到第一行已经输出“Hello World”。



至此，我们的工作已经完成了。

Bochs的debug指令

同学们自己在写程序的时候一般不会一次便运行成功，此时便需要一些debug手段。bochs常用的debug命令如下，命令的输入是在bochs的debug窗口。

命令	用于
b 地址	设置断点
c	继续执行代码到下一次断点
s	执行下一条指令
n	执行下一条指令但遇到函数不进入
r	查看普通寄存器的值
sreg	查看段寄存器的值
print-stack	查看堆栈的值
Ctrl-C	停止执行，返回到命令行提示符
x /nuf 地址	查看线性地址处的内存内容。 n 指定要显示的内存单元的数量 u 显示的内存单元的大小，如下参数之一 b 单个字节 h 半个字(2 字节) w 一个字(4 字节) f 打印的格式。如下类型之一 x 按照十六进制形式打印 d 按照十进制形式打印 u 以无符号的10进制打印 o 按照八进制形式打印 t 按照二进制行是打印
xp /nuf 地址	查看物理地址处的内存内容。

课后练习

1. 请你谈谈对多层语言模型的理解，即为什么需要有机器语言、汇编语言和高级语言三层?
2. 请你描述下IA-32处理器的种类和用法，例如eax又可以分为哪几个寄存器来访问？esp的用途是什么?
3. eflags的各个位有什么含义?

4. 什么是线性地址？实模式的寻址模式是什么？地址空间大小如何？
5. nasm汇编中的内存寻址方式有哪些？语法是什么？请分别描述。
6. 在什么情况下会使用默认寄存器 `cs`， `ds`， `ss`？如何避免CPU在计算线性地址时使用默认寄存器？
7. 我们已经知道， `push ax` 等价于下面的语句。

```
; 下面语句等价于 push ax
sub sp, 2          ; 从高地址向低地址增长，16位实模式
mov [sp], ax
```

请问为什么不是等价于下面的语句？

```
; 下面语句等价于 push ax
mov [sp], ax
sub sp, 2          ; 从高地址向低地址增长，16位实模式
```

为什么不是等价于下面的语句？

```
; 下面语句等价于 push ax
sub sp, 2          ; 从高地址向低地址增长，16位实模式
mov sp, ax
```

8. `jmp` 指令和 `call` 指令有什么不同之处？
9. 请写出下列伪代码对应的汇编语句。其中，ax，bx，cx是寄存器。

```
if ax == 16 then
    bx = 17
else
    cx = bx
```

10. 请写出 `pushad`、 `popad` 对应的汇编语句。
11. 下面的代码是否有错？如有错请指出，并描述程序最终的执行结果是什么。

```
call my_function
add ax, 10
jmp $
; $在nasm汇编中表示当前地址，即jmp $指令开始的地址
; 因此这条语句实际上是在做死循环，程序到这里就不会往下执行了

```

```
my_function:
    push ax
    push bx

    sub ax, 10
    sub bx, 10
    ; 后进先出
    pop ax

    ret
```

12. 下面的代码是否有错，如有错请指出，并描述程序最终的执行结果是什么。

```
call my_function
add ax, 10
jmp $
; $在nasm汇编中表示当前地址，即jmp $指令开始的地址
; 因此这条语句实际上是在做死循环，程序到这里就不会往下执行了

my_function:
    push bx
    push ax

    sub ax, 10
    sub bx, 10
    ; 后进先出
    pop bx
    pop ax

    ret
```

13. 下面的代码是否有错，如有错请指出，并描述程序最终的执行结果是什么。

```
call my_function
add ax, 10
jmp $
; $在nasm汇编中表示当前地址，即jmp $指令开始的地址
; 因此这条语句实际上是在做死循环，程序到这里就不会往下执行了

my_function:
    push bx
    push ax

    sub ax, 10
    sub bx, 10
    ; 后进先出
```

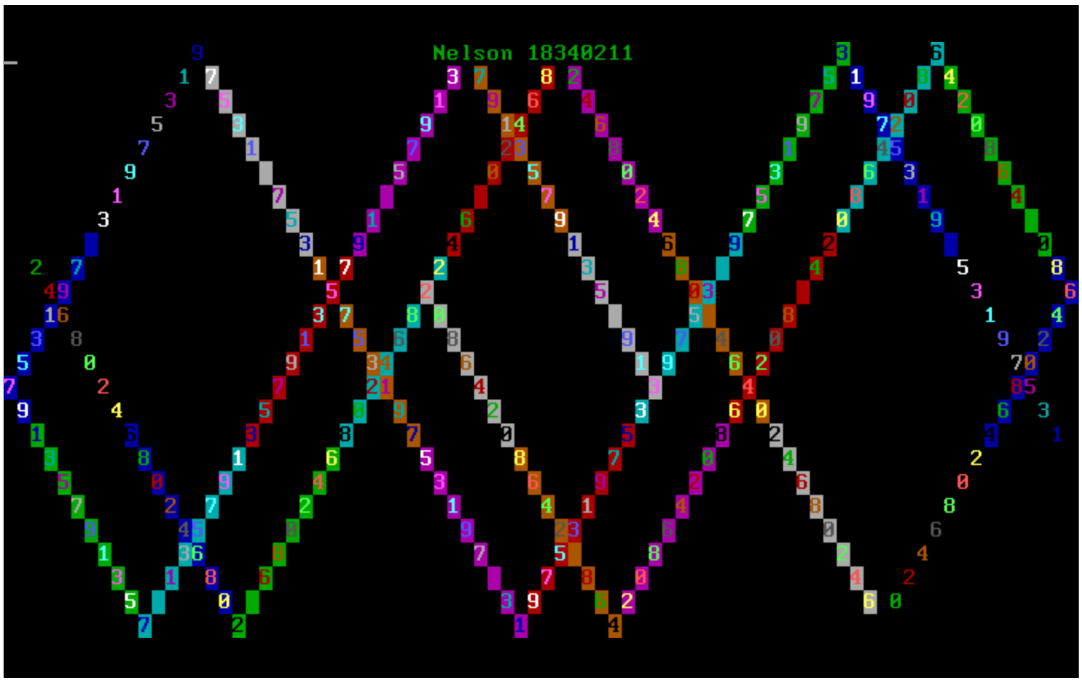
```
    pop ax
    pop bx

    ret
```

14. 人为什么不能举起自己？请尝试从物理学的角度加以分析。
15. 请复现Assignment。
16. 请使用Assignment测试bochs的debug指令。
17. 请简要谈谈本次教程的不足之处。

Bonus

1. 字符弹射程序。请编写一个字符弹射程序，其从点$(2,0)$处开始向右下角45度开始射出，遇到边界反弹，反弹后按45度角射出，方向视反弹位置而定。同时，你可以加入一些变色的效果。注意，你的程序应该不超过510字节，否则无法放入MBR中被加载执行。示例效果如下。



2. 下面的例子也是输出“Hello World”，而且比示例更加简单。但是，由于我时常在凌晨4点时为同学们编写tutorial，导致下面的代码是在意识不清醒的情况下写出的。显然，代码无法输出我想要的结果。我因为提早了睡眠时间，无法进行debug工作，恳请同学们帮助我完成这一过程。同学们需要完成的任务如下。
    - **请指出问题出现的原因并想出两种不同的修改方法。**注意，修改方法不包括直接将字符输出到对应的内存，而是要通过标号 `string` 来获取相应的字符，循环打印。(tips: 数据(包括string)和代码都会被加载到0x7c00处执行)
    - 请解释寄存器 `cx` 除了用在了 `mov` 指令中，还在哪里被使用了。
    - 判断代码中 `[si + string]` 和 `[gs:bx]` 的内存访问方式，并指出对应的段寄存器、偏移地址和线性地址的计算公式。

```
[bits 16]
xor ax, ax ; eax = 0
```

```
3   ; 初始化段寄存器, 段地址全部设为0
4   mov ds, ax
5   mov ss, ax
6   mov es, ax
7   mov fs, ax
8   mov gs, ax
9
10  ; 初始化栈指针
11  mov sp, 0x7c00
12  mov ax, 0xb800
13  mov gs, ax
14
15  mov ah, 0x01 ;蓝色
16  mov cx, 11   ; 11是'hello world'的长度
17  mov si, 0
18  mov bx, 0
19
20  print_hello_world:
21        mov al, [si + string]
22        mov [gs:bx], ax
23        inc si
24        add bx, 2
25        loop print_hello_world
26
27  jmp $ ; 死循环
28
29  string db 'Hello World'
30
31  times 510 - ($ - $$) db 0
32  db 0x55, 0xaa
```