汇编期末课程报告

一.本课程重难点

1. 寻址方式

寻址方式最开始学的时候，不知所云，不知道用来做什么，但是在做实验的过程中，慢慢的体会到寻址方式的重要性，我的理解是寻址方式主要就是我们如何去取得一个数据，如何将该数据存放在指定的内存地址，我们怎么表示数据的地址以及内存地址就是"寻址方式"做的事情。以下是一共5种类型本学期学习的寻址方式。其中立即寻址和寄存器寻址不再展开叙述。其中最重要的还是存储器寻址和后面学习学到的I/O地址寻址。

1.1 存储器寻址

- 操作数存放在存储器中，而不是寄存器中（寄存器和存储器是不一样的概念，寄存器是独立存在的，我们经常用的寄存器有AX、BX等等这些，而存储器是什么呢，就是内存中的一个一个的内存单元所组成的存储器）。用存储器寻址的指令，操作数一定在数据段、堆栈段、附加段中的主存储器中，指令中一定包含有存储器单元的地址；
- 执行指令时，CPU先根据指令提供的地址信息，计算出偏移地址，然后和段寄存器相加，得到可以直接访问的内存地址，再从内存中取出操作数，执行响应的操作；
- 注意点：

a. 可以采用段跨越前缀方式来改变系统指定的默认段，默认的段是数据段DS；

b. 串处理（即字符串处理）必须要使用ES段（附加数据段）；

c. 栈操作指令必须使用SS段（堆栈段）；

d. 指令必须在CS（代码段）；

- 有效地址可以由以下四种成分组成：

a. 位移量；

b. ***基址，存放在基址寄存器中的内容，他是有效地址中的基址部分，通常用来指向数据段中数组或者字符串的首地址***；

c. 变址，存放在变址寄存器中的内容，通常用来访问数组中的某一个元素或者字符串中的某一个字符；

d. 比例因子，其值可为1，2，4，8。在寻址中，可用变址寄存器的内容乘以比例因子来取得变址量，对于访问元素长度为2，4，8字节的数据有用；

四种成分	16位寻址	32位寻址
位移量	0，8，16位	0，8，32位
基址寄存器	BX，BP（称为基址指针寄存器，一般是和堆栈段寄存器联用确定SS段中某一存储单元的地址）	任何32位通用寄存器（包括ESP）
变址（Index）寄存器	SI(Source Index)，DI(Destination Index)一般与DS联用，来确定数据段中某一个存储单元的地址	除ESP以外的32位通用寄存器
比例因子	无	1，2，4，8

1.1.1 直接寻址方式

```
mov al,[2000]
mov ax,[2000];默认是将字单元取出来传送到ax寄存器中
mov ax,es:[2000];段超越
mov word ptr [1234],eax;将eax里面的双字数据传送到内存地址为ds:1234的位置
```

1.1.2 寄存器间接寻址方式

直接寻址方式是直接将偏移地址放在操作数位上，这样难免有些不雅，所以说我们将偏移地址放在寄存器中，然后再来通过访问寄存器的内容来得知内存单元的偏移地址，存放偏移地址的寄存器称为间址寄存器。

间址寄存器又分为两类，一类是基址寄存器（BX，BP），一类是变址寄存器（SI，DI）

```
mov ax ,[bx];这个是一个很简单的寄存器间接寻址方式，该指令执行的内容不是将bx寄存器的数据传送到ax寄存器中，而是将以bx寄存器内容为起始地址的大小为一个字的数据
            ;传送到ax寄存器中
```

1.1.3 寄存器相对寻址方式

- 与寄存器间接寻址方式类似，但不同的是，需要另外指定一个位移量，对于16位系统来讲，位移量是8位或者16位；对于32位系统来讲，位移量是8位或者32位，位移量是一个带符号的整数；
- 例如`mov ax 10h[si]`，相当于`mov ax ,[si+10h]`
- 也可以结合段跨越前缀，例如`MOV DL,ES:STRING[SI]`该指令相当于`MOV DL,ES:[SI + STRING]`。

实际应用：

```
;当我们定义了一个数组的时候，我们会用到寄存器相对寻址方式
;例如:
data segment
    arrays dw 10 dup(0)
data ends
......
mov bx,0
mov ax,arrays[bx]
......
```

1.1.4 基址变址寻址方式

- 指的就是操作数的偏移地址一部分在基址寄存器，一部分在变址寄存器，基址寄存器的内容再加上变址寄存器的内容就是操作数的偏移地址。
- 例如`mov ax,[bx][si]`等价于`mov ax,[bx+si]`;

1.1.5 相对基址变址寻址方式

- 就是带位移量的基址变址寻址方式称为相对基址变址寻址
- 示例：

```
mov ax,100[bx][si];相当于mov ax,100[bx + si]
mov ax,100[bp][si]
```

1.1.6 比例变址寻址方式

- 只有在32位以及以后的80x86系统中使用
- 示例：`mov eax,table[ebp][edi*4]`

1.2 I/O地址寻址

1.2.1 direct i/o port addressing(直接端口寻址)

- 使用00h-ffh表示0-122个8位的I/O端口地址，就可以直接进行I/O端口寻址；
- 示例：`in al ,80h`将80h端口的数据输入到al寄存器中；
- `in ax,80h`将80h端口的数据输入到ax寄存器中；
- `out 80h,al`字节输出指令，将al寄存器的内容输出到80h端口；

1.2.2 indirect i/o port addressing(间接端口寻址)

- 如果端口大于256个，实际上是65536个端口，我们就必须使用间接寻址，就是先将端口号传送到dx寄存器中，然后将dx寄存器放到操作数上；
- `mov dx,200h`
- `out dx,al`
- `in ax ,dx`

1.3 与转移地址有关的指令

- 转移指令可以改变指令的执行顺序，进行跳转，比如说最常见的`JMP`等等。简而言之，就是改变CS和IP的值，从而改变下一条要执行的指令的物理地址；
- 转移的话，也分为段内转移以及段间转移，都可以使用直接寻址和间接寻址

1.3.1 段内直接寻址

`jmp label`label为转移的地址符号，也称为标号

示例：

```
    jmp short s
    add ax,1
s:  inc ax
```

该段指令很简单，就是跳转到s处执行指令`inc ax`

1.3.2 段内间接寻址

转移的偏移地址存储在寄存器里面或者存储在某一个存储单元中。

```
jmp bx;
jmp word ptr [bp];
jmp word ptr[bp + val];BP+VAL指向的内存单元的一个字传送给IP寄存器，作为偏移地址
```

1.3.3 段间直接寻址

利用操作符`far ptr`可以实现段间的转移，即同时修改CS和IP寄存器

示例：

`jmp far ptr temp`

1.3.4 段间间接寻址

即使用存储器中的两个相继字（连续的两个字单元）的内容来代替IP和CS寄存器的内容，以达到段间转移的目的

```
jmp dword ptr [si]
jmo dword ptr [addr]
```

2. 子程序设计

在高级语言中，我们有函数调用，在汇编中我们也有类似的模块，就是子程序，我们可以利用子程序将一个问题模块化。我们主要利用的`CALL`指令和`RETN`指令来实现调用子程序和子程序返回。其中涉及到我们需要注意调用子程序的时候，我们需要保护现场，将一些存放关键数据的值入栈保存，等到子程序执行完毕再进行弹栈操作。

还有就是涉及到子程序的参数传送，我们可以通过寄存器进行传参操作，一般我们高级语言中编译器编译之后就是利用寄存器传参。如果程序和调用的子程序再同一个源文件中，我们就可以直接访问源文件中的在数据段中定义的变量，但是这样的话会导致子程序固化，即子程序只能操作哪一些数据是定死的。所以说我们通过地址表传送变量地址，该方法就是在主程序中建立一个地址表，就是将所要使用的数据存放在指定的地址中，然后将该地址表的首地址存放到指定的寄存器红，以实现一个传参的过程。比如说下面操作就是我们在调用子程序之前先将我们参数的地址放入指定的地址表中：

```
mov table,offset num
mov table+2,offset n
mov table+4,offset total
mov bx,offset table
call proadd
```

还有就是通过堆栈传送参数或者参数的地址。我们在做电话簿实验的时候，对子程序的利用就印象颇深。

3. I/O程序设计

I/O程序主要就是设计到我们如何利用硬件接口以及I/O端口实现和硬件之间的交互。我们利用`IN`指令和`OUT`指令以及直接端口寻址和间接端口寻址来进行与硬件之间数据的传送。其中最重要的，我觉得还是中断的利用，因为没有中断的话，我们就没有那个空隙去从指定的端口读取数据。

我在修改9号中断的实验中对中断的利用体会很深，我们在设计这个实验的时候，遇到一个问题，就是当我们一旦修改了9号中断的中断例程之后，我们就无法获取到我们键盘按下的键对应的ASCII码，因为9号中断的作用是将我们键盘按下的键的扫描码以及ASCII码放在BIOS缓冲区中，如果我们想要在屏幕中显示对应的字符，就得实现保存好9号中断向量地址，然后在我们设计的“9号中断”中调用9号中断，让其从80号端口读取数据并将其放在BIOS缓冲区中，这样就可以输出对应的字符。

4. 内外中断

关于中断，***端口和中断，是CPU进行IO的关键！***如果CPU离开了中断，基本上大部分的功能都会失效。CPU可以通过指令在内部进行各种运算，但是CPU除了有运算能力之外，还要有IO能力，即对外部设备进行控制，接收输入和输出,这个接收的过程就需要中断来辅助完成。

CPU与外设要通过接口进行交流，即IO操作。接口有两种类型，一种是控制器，另一种是适配器。控制器即IO设备本身或主板上的芯片组，比如磁盘控制器和USB控制器。适配器则是我们俗称的各种卡，比如图像适配器，即显卡，网络适配器，即网卡。这些控制器或者适配器芯片的内部有若干寄存器，CPU即将这些寄存器当做端口来访问，也就是我们之前所讲的端口寻址的目的所在！外设的输入并不是直接送入内存和CPU，而是送入相关的接口芯片的端口中，接着CPU利用端口寻址读入相关的数据。CPU向外设的输出也不是直接送入外设，而是先利用端口寻址送入端口，再由相关芯片送到外设。

CPU还可以向外设发出控制命令，这些控制命令也先送到相关接口芯片的端口中，然后由芯片根据命令对外设实施控制。CPU通过检测到外设发过来的中断信息，引发中断过程，从而来处理外设的输入。

中断又分为内中断和外中断，内中断是指CPU内部发来的中断，外中断指的是CPU外部，即外设发来的中断。二者是有一定的区别的，内中断的中断类型码是在CPU内部产生的，外中断的中断类型码是通过数据总线送入CPU的。

造成内中断的情况主要有以下几种：

- 除法错误，比如执行DIV指令的时候的除法溢出,***终端类型码为0***；
- 单步执行,***中断类型码为1***；
- 执行`into`指令,溢出中断指令，Interrupt If Overflow,***中断类型码为4***；
- 执行`int`指令，该指令的格式是`int n`，指令中的n为字节型立即数，是提供给CPU的中断类型码；

虽然我们知道了上面四种情况会造成中断，但是CPU并不知道，更不知道怎么去处理这些中断，所以我们后面又学习了中断向量表。我们不仅仅给每一种中断编上对应的号码，然后绘制一张表，表中存放的是中断编号以及中断的时候，调用的处理程序所在的段地址和偏移地址（子处理程序的入口地址），这样的话中断的时候，CPU就会知道去调用哪些处理程序了。

外中断主要来源于外中断源，就是造成中断的源头，外中断源又分为以下两类：

外中断源一共可以分为以下两类:

- 可屏蔽中断；

在内中断中将IF置为0的原因就是在进入中断处理程序之后，禁止一切其他的可屏蔽中断。

当然了，如果在中断处理程序中需要处理可屏蔽程序，可以将指令将IF置为1：

```
sti # 设置IF=1;
cli # 设置IF=0;
```

- 不可屏蔽中断；

不可屏蔽中断是CPU必须响应的外中断，不可屏蔽中断的中断类型码固定为2，所以说中断过程中，不需要取中断类型码。则不可屏蔽中断过程为：

- 标志寄存器入栈，IF=0，TF=0；
- CS、IP入栈；
- (IP)=(8),(CS)=(0AH)

我们最常见的外中断就是我们的键盘中断，我们每一次按下键就会引发9号中断，这个我们在以往的实验中可以体会到。

5. 学习收获与感想

通过这一学期的汇编的学习，感觉收获还是蛮大的，当然最大的收获就是基本上入门掌握了汇编这一门语言，这给我们后面制作操作系统提供了很大的遍历，以及对于读懂那些操作系统底层的汇编代码也有很大的帮助，对程序的理解更进了一步，并且通过反汇编我们可以去做很多有趣的事情，这我都觉得是这学期很大的收获，学习汇编过程，一开始虽然很难，但是一旦找到感觉了，平时多写一些代码，学习汇编就会变得如鱼得水。总而言之，这学期收获很大。

二. 程序设计

> 编写程序，实现如下功能：屏幕清屏，中心位置显示数字倒数3秒 (3、2、1、0)，倒数结束之后显示：`YOUR PC HAS BEEN LOCKED!` 此时按键(除了ESC之外)无反应，模拟电脑死机。按下ESC可以退出程序。

1. 程序关键代码分析

屏幕清屏，是利用10号中断实现，本程序具体实现如下：

```
;定义宏指令，AL = 窗口滚动的行数(如果是0的话，就是清屏)
;(CH,CL) = 窗口的左上角位置
;(DH,DL) = 窗口的右下角位置
clear   macro a,b,c,d
    mov     al,0
    mov     bh,7
    mov     ch,a
    mov     cl,b
    mov     dh,c
    mov     dl,d
    mov     ah,6
    int     10h
endm
```

我们在屏幕的中央显示倒计时是通过依次显示3、2、1、0来实现的，其中设置有延时程序`delay`。我们是通过直接将字符传送到显存当中来实现显示数字的，具体代码如下：

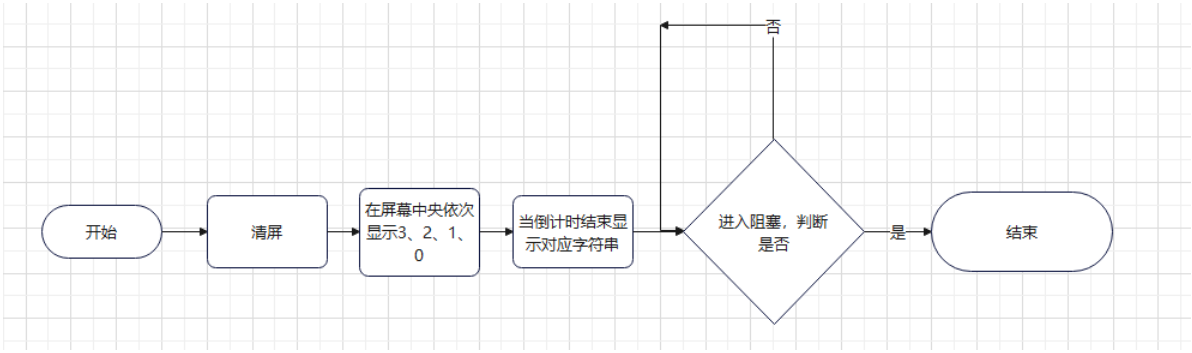
```
......
    mov     ax,0b800h
    mov     es,ax
......
puts   macro x

    mov     di,12*160+80
    mov     es:[di],ax
endm
```

内存地址空间中，B8000H~BFFFFH一共32KB的空间，为80x25彩色字符模式的显示缓冲区。如果我们向该地址空间中写入数据的话，写入的内容就会立即出现在显示器上。在80x25彩色模式下，显示器可以显示25行，每一行80个子符，每一个字符都有各自的属性，这样的话，一个字符就要占据两个字节，所以说屏幕的中间位置大概为`mov    di,12*160+80`

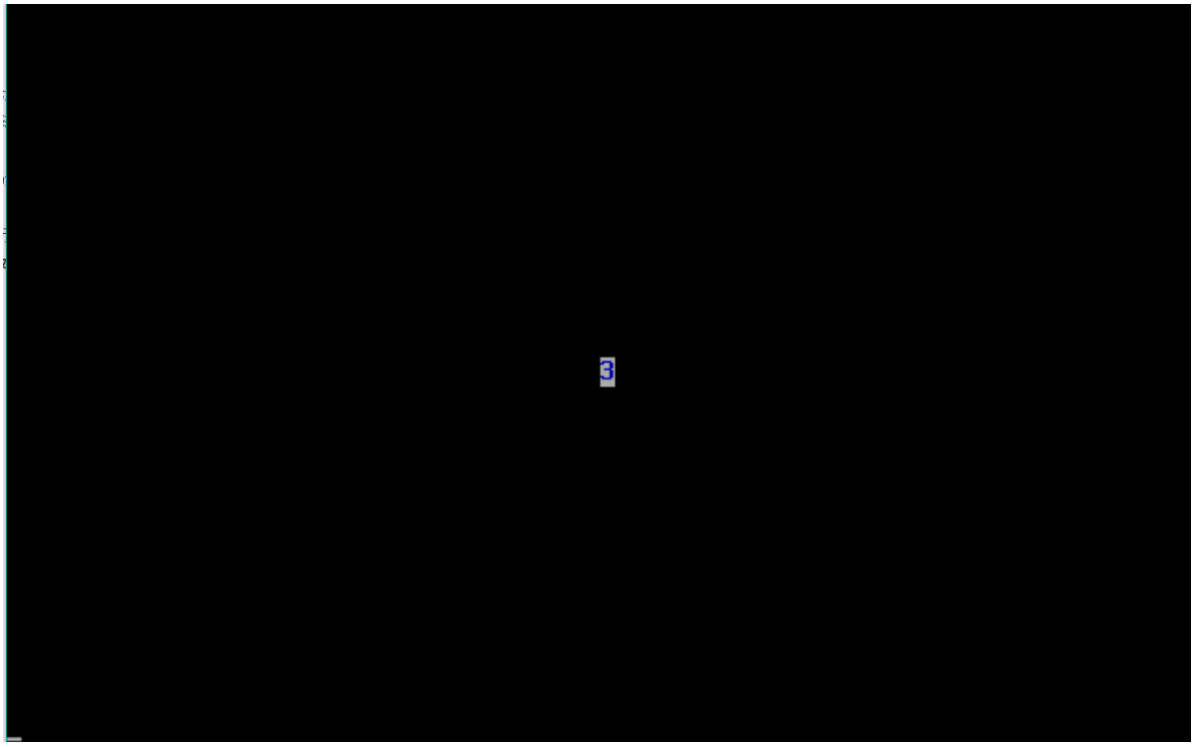
接着就是阻塞部分了，我们会一直利用16号中断从BIOS缓冲区中读取数据，除非读取的数据是ESC，否则什么都不响应，以此来模拟电脑死机，具体实现部分为：

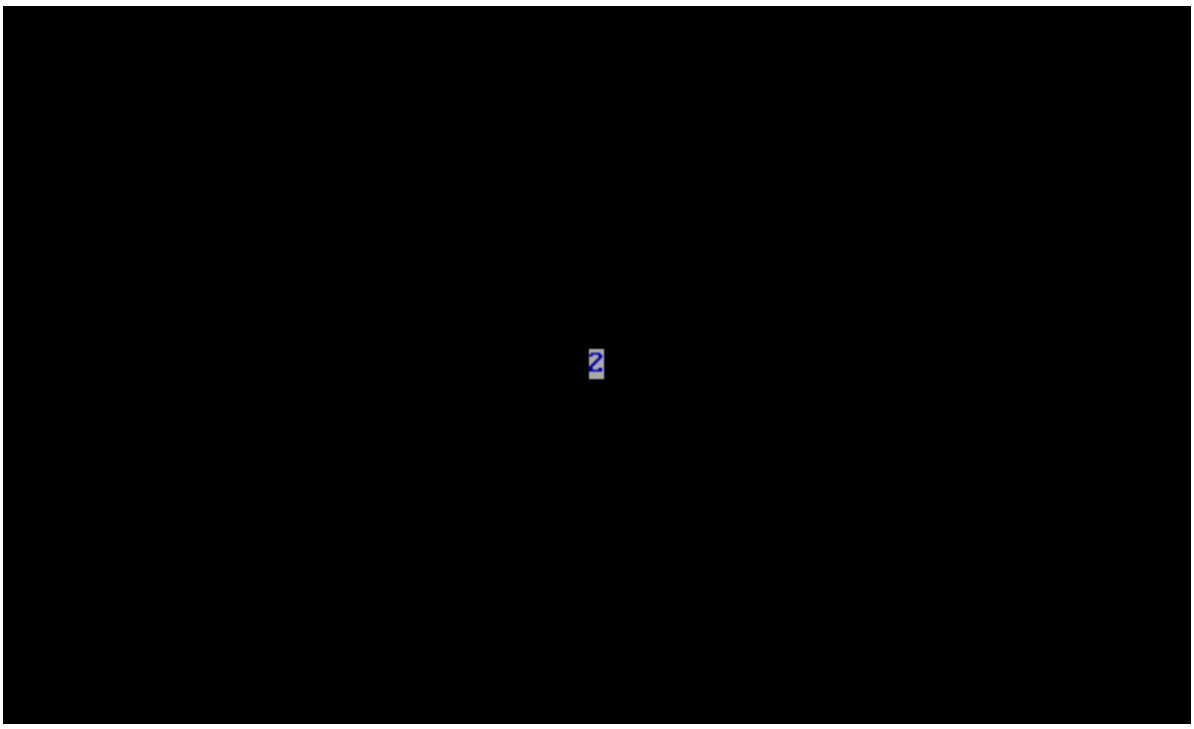
```
block   macro
block_loop:
    mov     ah,0
    int     16h
    cmp     ah,01h
    jne     block_loop
    clear   0,0,24,79
    mov     ah,4ch
    int     21H
endm
```

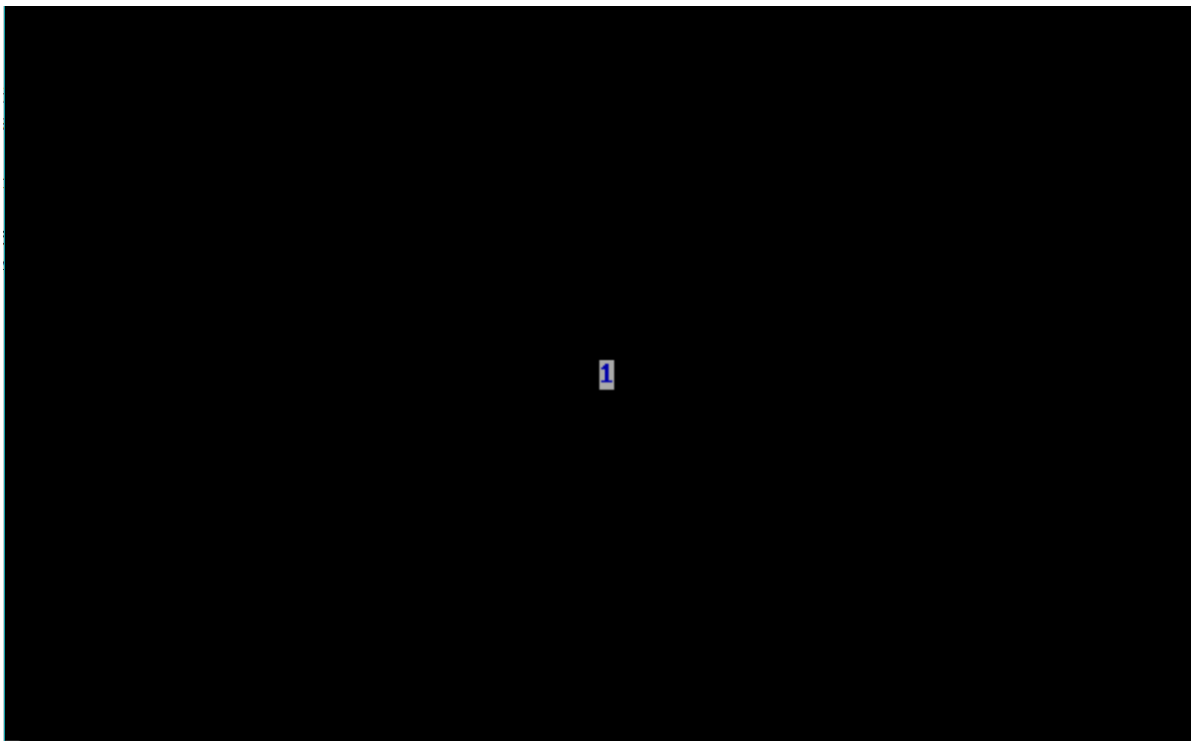
2.程序流程图

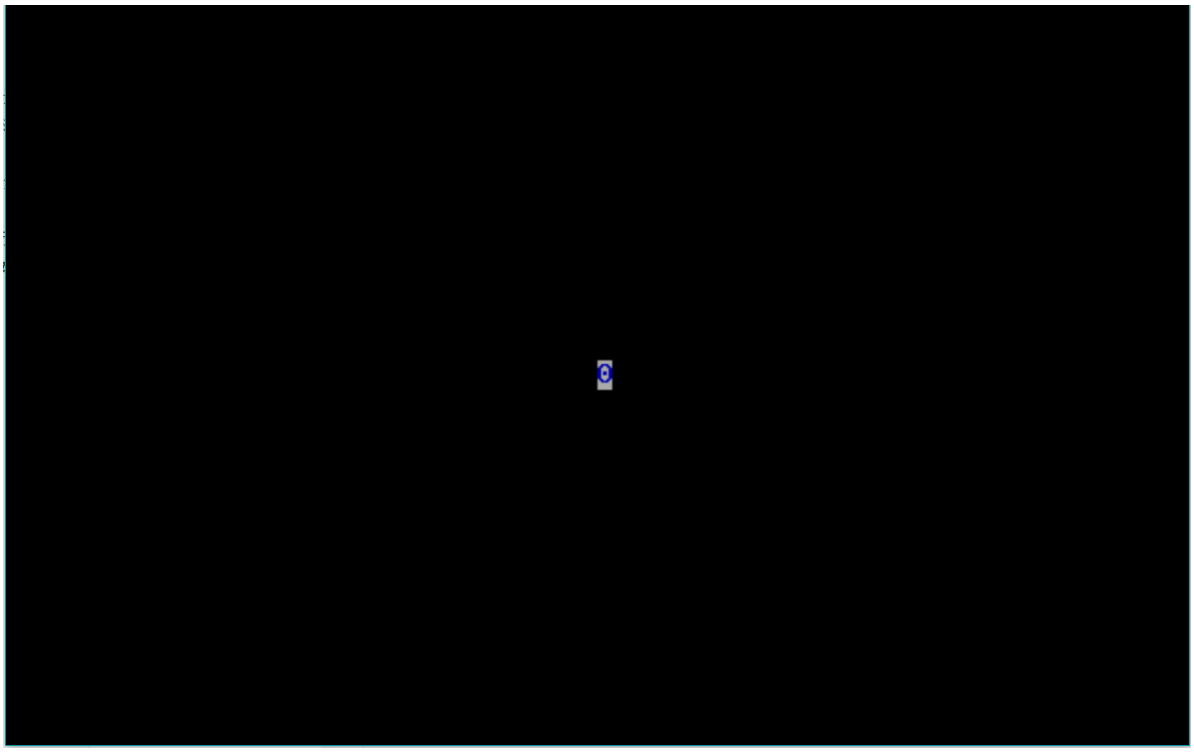


附录1：程序执行结果






1


0

附录2：程序具体代码

```
assume cs:code,ds:data

;定义宏指令，用来清屏用
clear   macro a,b,c,d
    mov     al,0
    mov     bh,7
    mov     ch,a
    mov     cl,b
    mov     dh,c
    mov     dl,d
    mov     ah,6
    int     10h
endm


puts   macro x
    mov     di,12*160+80
    mov     es:[di],ax
endm

; 阻塞，对按下的字符进行判断
block   macro
block_loop:
    mov     ah,0
    int     16h
    cmp     ah,01h
    jne     block_loop
    clear   0,0,24,79
    mov     ah,4ch
    int     21H
endm
```

```
32
33  data segment
34      ouput db "YOUR PC HAS BEEN LOCKED!"
35  data ends
36
37  code segment
38  start:
39      clear   0,0,24,79
40
41      mov     ax,0b800h
42      mov     es,ax
43
44      mov     ax,data
45      mov     ds,ax
46
47      mov     ax,7133h
48      puts    ax
49      call    delay
50
51      mov     ax,7132h
52      puts    ax
53      call    delay
54
55      mov     ax,7131h
56      puts    ax
57      call    delay
58
59      mov     ax,7130h
60      puts    ax
61      call    delay
62      mov     di,12*160+30*2
63      lea     si,ouput
64      mov     cx,24
65
66  loop1:
67      mov     al,[si]
68      mov     ah,0cah
69      mov     es:[di],ax
70      inc     si
71      inc     di
72      inc     di
73      loop    loop1
74
75      block
76
77  delay:
78      push ax
79      push dx
80      mov dx,0012h
81      mov ax,0
82  s1:
83      sub ax,1
84      sbb dx,0
85      cmp ax,0
86      jne s1
```

```
    cmp dx,0
    jne s1
    pop dx
    pop ax

    ret
code ends

end start
```