汇编语言 期末总结

数据组织与结构

H1

内存中的地址及引用

H2

在编程时，只能使用逻辑地址，不能使用物理地址。逻辑地址的基本结构就是`段地址:偏移地址`，段地址的值为`段起始地址 / 10h`，由此，段起始地址十六进制个位必须为0。在16位系统中，段地址和偏移地址都是8位的长度，因此一个段的长度应该是`10000h`，也即64KB。

H3

一个物理地址可以表示为多个逻辑地址，例如`12398h = 1234:0058 = 1235:0048 = 1230:0098 = ...`。其中，段地址的1相当于偏移地址的10h。

在寻址过程中，有两（三）种方法：

1. 直接寻址：用常数来表示变量的偏移地址，但是段地址必须用段寄存器（`cs, ds, ss, es, fs, gs`）表示，而不能用常数表示。如果要访问`ds:2000h`位置的变量，使用`ds:[2000h]`即可，其中`ds:2000`可以看作一个指针，而`ds:[2000h]`表示这个指针指向的对象。可以使用修饰词来对这个指针做出修饰，如`byte ptr`，其中`ptr`是指针（pointer）的缩写，而`byte`表示对象类型为byte，可使用的对象类型参考 **数据类型** 部分。
2. 间接寻址：用寄存器、寄存器 + 寄存器、寄存器 + 常数、寄存器 + 寄存器 + 常数的形式来表示变量的偏移地址，其中，可用于间接寻址的寄存器只有`bx, bp, si, di`四个，如果将寄存器组合使用，`bx`和`bp`不能同时使用，`si`和`di`不能同时使用。例子如`ds:[bx]`，`ds:[bx+si]`，`ds:[bp+6]`，`ds:[bp+si+8]`。前面同样可以使用修饰符进行修饰。
3. 在32位中，新增的寻址方法为寄存器 + 寄存器 * n + 常数，其中 n=2, 4, 8。这种方式可以认为方便了数组的访问，例如`[ebx+esi*4+6]`。与16位系统不同，32位系统中，对中括号中的寄存器没有要求，`ebp, ebx, esi, edi, eax, ecx, edx, esp`都可以放在中括号中。

在访问过程中，如果操作数的类型并非模糊的，则可以不使用修饰词，否则需要使用修饰符，例如`mov ds:[bx], 1`存在语法错误，因为`1`不能确定是何种类型的变量，也不确定占据多大的存储空间。

段覆盖（segment overriding）：在引用数据时，一般情况下，段地址被隐含为`ds`，在中括号中有`bp`时，段地址被隐含为`ss`。通过在操作数之前添加一个段前缀（segment prefix），例如`cs:`, `ds:`, `es:`, `ss:`来强制改变操作数的段地址，这就叫段覆盖。例如`[bp+2]`默认为`ss:[bp+2]`，可以改成`ds:[bp+2]`。

数据类型

在汇编语言定义变量时，并不需要区分符号数（signed）和非符号数（unsigned），其隐含于使用的指令之中。数据类型分别有五（六）种，表达如下：

H3

```
byte, db = define byte 字节，8位
word, dw = define word 字，16位
dword, dd = define double word 双字，32位
fword, df = define far word 48位（仅限于386以后的机型）
qword, dq = define quadruple word 64位
tbyte, dt = define ten bytes 80位
```

变量的定义语法配套地有两种：

```
a db 0FFh
b byte 0FFh

pi dd 3.14
pi dword 3.14
```

在本文档的例程中，统一使用前者（因为小白喜欢），如果要定义数组，则可以使用`dup`命令，例如`abc db 10 dup(0)`定义了一个名为`abc`的由十个 byte 类型的 0 组成的数组。可以使用`?`表示无初始值或初始值任意，例如`a db ?`

> 说明：在masm编译器中，事实上存在符号数类型sbyte, sword, sdword这三种类型实际上与byte, word, dword等价，在使用上并无不同，cf. *MASM v6.1 Programmer's guide* Chapter 4，Chapter 6中描述了实数类型real4, real8, real10，事实上也可以用相应位宽度定义。另外，在masm中，可以使用typedef语句，如：
>
> ```
> char typedef byte
> ```
>
> 就是把char定义成byte的同义词，具体参考*ibid.*

变量的二进制表示与扩充

理所当然地，我们可以写出整型变量所能表示的范围：

H3

宽度	符号数	非符号数
8位	[80h, 7Fh], i.e. [-128. 127]	[00h, 0FFh]
16位	[8000h, 7FFFh], i.e. [-32768, 32767]	[0000h, 0FFFFh]
32位	[80000000h, 7FFFFFFFh]	[00000000h, 0FFFFFFFFh]

对于小数变量，IEEE754标准给出了`float`类型（32位）小数的表示：

```
例子：127.375
二进制表示：0|1000010 1|1111110 11000000 00000000
            | --------- ------------------------ 尾数
            | 偏置指数：8位非符号数
            符号位（1位）
十六进制表示：4C FE C0 00
在将其转换成十进制的过程中，先提取尾数：
1111110 11000000 00000000 并在其前面补上1.成为一个二进制小数
1.1111110 11000000 00000000
然后计算次数 = 133（偏置指数）- 127（常数）= 6
于是将小数点右移六位，得到 1111111.0 11000000 00000000
再将其转化为十进制 127.375
```

用以下C代码可以打出127.375内存中的4个字节：

```c
int main(){
    float f=127.375;
    unsigned char *p = (unsigned char *)&f;
    for(int i = 0; i < 4; i++)
        printf("%02X",  p[i]);
    return 0;
}
//输出：00C0FE42
//这里输出的形式是由于小端规则，参见下一节
```

> 说明：本文档中使用的C代码都来源于小白，但是由于我并不喜欢ANSI C的风格做了一点点无关痛痒的改动。在变量定义过程中，明确地强调了`unsigned`这一关键字，主要是出于对汇编语言的呼应。

当将一个宽度较小的值赋给一个宽度较大的值时，会发生扩充（extension），扩充包括零扩充（zero extension）和符号扩充（sign extension）两种：

```
当我们把一个8位的变量11111111B赋给16位的变量时
零扩充在前面填0，可以得到：
00000000 11111111B
-------- 扩充部分
符号扩充在前面填符号位，可以得到：
11111111 11111111B
-------- 扩充部分
可以发现，对于非符号型变量，零扩充保留了其原来的值，对于符号型变量，符号扩充保留了它的值
```

变量在内存中的存储

在变量存储过程中，先存放低八位，再存放高八位的规则称为小端规则（little-endian），例如，我们通过`a dw 1234h`定义了一个变量之后，数据在内存中的存储如下：

H3

```
地址      值       二进制
1000    0×34    0011 0100    低八位在前
1001    0×12    0001 0010    高八位在后
```

通过以下代码可以验证小端规则：

```c
int main(){
    unsigned short int a = 0×1234;
    unsigned char *p = (unsigned char *)&a;
    printf("%X %X", p[0], p[1]);
    return 0;
}
//输出：34 12
int main(){
    unsigned char a[2]={0×12, 0×34};
    unsigned short int *p = (unsigned short int *)a;
    printf("%X", *p);
    return 0;
}
//输出：3412
//第一个程序是储存的规则，第二个程序是读取的规则
```

变量与数组的引用

当我们在`data`段中定义变量如下时：

H3

```
data segment
abc db 1,2,3,4
xyz dw 789Ah, 0BCDEh
asd dd 12345678h, 56789ABCh; 首元素为asd[0]，末元素为asd[4]
data ends
```

我们有以下两种形式访问地址：

1. 直接地址，如`abc[1]`或`[abc+1]`，编译后变成`ds:[1]`
2. 间接地址，如`abc[bx]`或`[abc+bx]`，编译后变成`ds:[bx]`，方括号中的内容可以看作**内存中的地址及引用**部分的衍生，`abc`可以被解读为`offset abc`，即`abc`的偏移地址。

用`xyz`举例使得这里显得更加明显，`xyz[2]`和`[xyz+2]`会被编译成`ds:[6]`，`xyz[bx]`和`[xyz+bx]`会被编译成`ds:[bx+4]`

在完全方括号的形式中，相当于没有给出变量类型的明确说明，需要添加修饰词（参考**内存中的地址及引用**）

寄存器及其作用

16位CPU具有16位寄存器，其结构如下：

H3

```
AX: XXXX XXXX XXXX XXXX
    --------- ---------
        AH        AL
```

寄存器主要分以下几类：

1. 通用寄存器

   `AX, BX, CX, DX`称为数据寄存器，它们一般可以通用（`BX`的特殊性参考**内存中的地址和引用**部分），但是也有习惯性的用法：

   - `AX`（Accumulator）累加器，在乘除法运算，串运算和I/O指令中作为专用寄存器
   - `BX`（Base）基址寄存器，在寻址时常用来访问基址
   - `CX`（Count）计数寄存器，在循环和串操作中作为专用寄存器
   - `DX`（Data）数据寄存器，存放I/O端口地址，以及`DX:AX`用来存放一个双字，其中`DX`存放高16位

   `SI, DI`称为变址寄存器，`SI`（Source Index）称为源变址寄存器，`DI`（Destination Index）称为目标变址寄存器，常用于寻址

`SP，BP`称为指针寄存器，`SP`（Stack Pointer）称为堆栈指针寄存器，与堆栈段寄存器配合组成`SS:SP`为栈顶指针，不能用`SP`进行寻址操作。`BP`用以对堆栈段中的数据进行寻址

2. 段寄存器

   `CS，DS，SS，ES`称为段寄存器，用来存放段地址：

   - `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）附加数据段寄存器

3. 控制寄存器

   `IP，FL`称为控制寄存器，其中`IP`（Instruction Pointer）称为指令指针寄存器，用来存放代码段中的偏移地址，在运行过程中，`CS:IP`始终指向下一条指令的地址。

   `FL`（Flags）称为标志寄存器，它是16位的寄存器，但是只使用其中的九位，这个九位包含6个状态（条件码）标志和3个控制标志，如下：

```
15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
-- -- -- -- OF DF IF TF SF ZF -- AF -- PF -- CF
```

   其中状态标志用来记录程序运行结果的状态信息，它们的值是在一次运算之后计算机依据运行结果设定的，它们常常用来后续条件转移指令的转移控制条件，所以又称为条件码。

   - `OF`（Overflow Flag）溢出标志，对符号数而言，如果运算结果超出了其能表示的数值范围，则称为溢出发生，此时计算机将会置`OF`为`1`
   - `SF`（Sign Flag）符号标志，记录运算结果的符号，结果为负时置`1`，结果为正时置`0`
   - `ZF`（Zero Flag）零标志，运算结果为0时，置`1`，否则置`0`
   - `CF`（Carry Flag）进位标志，记录最高位向左边产生的进位值或借位值，若有进位或借位时，置`1`，否则置`0`
   - `AF`（Auxiliary Carry Flag）辅助进位标志，记录运算时第3位相左边产生的进位值或借位值，若有进位或借位时，置`1`，否则置`0`
   - `PF`（Parity Flag）奇偶标志，当结果第八位中`1`的个数为偶数时，置`1`，否则置`0`

   控制标志用来指导程序运行的过程，它们的值一般由用户程序设置，用以控制气候程序的运行。

   - `DF`（Direction Flag）方向标志：控制字符串的操作方向。当`DF=0`时为正方向（低地址到高地址），当`DF=1`是反方向

- `IF`（Interrupt Flag）中断标志：当`IF=1`时，允许中断，否则禁止中断（中断参考 **中断及其应用** 部分）
- `TF`（Trace/Trap Flag）跟踪/陷阱标志：当`TF=1`时，CPU会进入单步模式(single-step mode)。CPU在每执行完一条指令后,会自动在该条指令与下条指令之间插入一条`int 1h`指令并执行它，利用单步模式可以实现反调试，参考 **附录I：利用单步模式实现反调试**

32位寄存器拓展了16位的寄存器，除段寄存器外，在寄存器名称前加字母e访问32位寄存器，其结构如下：

```
EAX: XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
                         --------- ---------
                            AH        AL
                         -------------------
                                  AX
```

而段寄存器仍然保持16位

堆栈段的定义和作用

堆栈是一种遵循先进后出（FILO, First In Last Out）原则的数据类型，在汇编中，程序运行时会存在一个堆栈段，使用`push`和`pop`对堆栈中的数据进行操作。若未定义堆栈段，则自发地将代码段、数据段总和后本段剩余的空间划归堆栈段。 也可以自己定义堆栈段，例如：

```
stk segment stack
db 200h dup(0); 或写成dw 100h dup(0)
stk ends
;堆栈空间是stk:0到stk:1FFh
;程序开始运行时,ss=stk,sp=1FFh+1
```

具体堆栈结构的操作见**数据传送指令**部分，应用见**函数框架**部分

汇编语言中的指令

数据传送指令（Data Movement Instructions）

通用数据传送指令

`mov`

格式为`mov 操作数1，操作数2`。将`操作数2`的值赋给`操作数1`，其中`操作数2`可以为常量、变量或寄存器，`操作数1`可以为变量或寄存器，但由于内存访问的限制，不能同时访问两个变量，两个操作数的宽度应当相同。`mov`指令不会改变任何标志位

H5

`xchg`

exchange的缩写，格式为`xchg 操作数1，操作数2`。将`操作数2`的值与`操作数1`交换，其中`操作数2`、`操作数2`可以为变量或寄存器，但不能同时访问两个变量，两个操作数的宽度应当相同。

H5

栈操作指令

`push`与`pop`

在16位汇编中，`push`和`pop`后面跟的对象必须是一个十六位的变量或寄存器，在8086中，`push`的对象不能是一个常数，但是在80386及以后的CPU中，这样的操作是允许的，且常数的宽度会被看作16位。

H4

H5

在`push 1234h`过程中：

1. `sp = sp - 2 = 200h - 2 = 1FEh`
2. 把`push`后面所跟的值`1234h`保存到`ss:sp`当前指向的内存单元中

```
ss:1FE 34h
ss:1FF 12h
```

在`pop`过程中：

1. 把当前`ss:sp`指向的字取出来，保存到`pop`后面所跟的变量中
2. `sp = sp + 2`

`pushf`与`popf`

属于`push`和`pop`的变体，将`FL`寄存器存入堆栈或将堆栈中的数据取出放入`FL`寄存器，是用来改变`FL`的一种重要方法

H5

`pusha`与`popa`

属于`push`和`pop`的变体，在16位系统下，`pusha`指令会执行以下动作：

H5

```
push ax
push cx
push dx
push bx
push sp
push bp
push si
push di
```

而`popa`指令则是对应的反向弹出

类型转换指令

`cwd`，`cdq`，`cbw`，`cwde`

`cwd`（convert word to doubleword）将`ax`中的内容符号扩充至`dx:ax`中，即将`dx`用`ax`中的符号位（最高位）填满

`cdq`（convert doubleword to quad-word）将`eax`中的内容符号扩充至`edx:eax`中

`cbw`（convert byte to word）将`al`中的内容符号扩充至`ax`中

`cwde`（convert word to doubleword extended）将`ax`中的内容符号扩充至`eax`中

`movsx`与`movzx`

`movsx`（move with sign extension）将宽度较小的值移给宽度较大的变量或寄存器，并进行符号扩充

`movzx`（move with zero extension）将宽度较小的值移给宽度较大的变量或寄存器，并进行零扩充

算术和位运算

整数的算数运算

`add`与`adc`

`add 操作数1, 操作数2`将`操作数2`的值与`操作数`相加之后赋给`操作数1`，其中`操作数2`可以为常量、变量或寄存器，`操作数1`可以为变量或寄存器，但不能同时访问两个变量，两个操作数的宽度应当相同。当发生进位时置`CF`为`1`，当结果为0时置`ZF`为`0`

`adc`（add with carry）在加法之后如果`CF`为`1`则多加一，与`add`配合可以实现一个宽度更高的加法，例如：

```
;将dx:ax与bx:cx相加到dx:ax
add ax, cx
adc bx, dx
```

`sub`与`sbb`

`sub 操作数1, 操作数2`将`操作数2`的值与`操作数`相减之后赋给`操作数1`，其中`操作数2`可以为常量、变量或寄存器，`操作数1`可以为变量或寄存器，但不能同时访问两个变量，两个操作数的宽度应当相同。当发生借位时置`CF`为`1`，当结果为0时置`ZF`为`0`

H5

`sbb`（subtract with borrow）在减法之后如果`CF`为`1`则多减一，与`sub`结合可以实现一个宽度更高的减法

`inc`与`dec`

`inc`（increment）将操作数的值增一，不改变`CF`，但是会改变其他五个标志位

`dec`（decrement）将操作数的值减一，描述同上

H5

`cmp`

`cmp`（compare）做和`sub`类似的操作，但是不保存结果，只改变标志位

`neg`

H5

`neg`（negate）做`0`减操作数的操作，取相反数

`mul`与`imul`

H5

`mul`（multiply）做非符号数乘法操作，后面跟一个操作数。如果操作数为8位，则将其乘以`al`，将结果储存到`ax`中；相对应的，16位乘以`ax`，结果储存到`dx:ax`中；32位乘以`eax`，结果储存到`edx:eax`

H5

`imul`做符号数的乘法操作，有三种用法，后两种操作仅限于80386以后的CPU：

1. 单操作数形式，同`mul`指令
2. 双操作数形式，第一个操作数为寄存器，第二个操作数为寄存器或变量，将两个操作数相乘并放入第一个操作数中
3. 三操作数形式，第一个操作数为寄存器，第二个操作数为寄存器或变量，第三个操作数为字面量，将第二、三个操作数相乘放入第一个操作数中

这三种用法的共同点为：乘积的宽度都是乘数的两倍，`CF`和`OF`在存在向长于乘数的部分进位的时候置`1`，当这部分是后半部分的符号扩充是置`0`

`div`与`idiv`

`div`（division）做非符号数除法操作，单个操作数为除数，运算过程如下：

除数	被除数	商	余数
8位	`ax`	`al`	`ah`
16位	`dx:ax`	`ax`	`dx`
32位	`edx:eax`	`eax`	`edx`

`idiv`指令做符号数除法操作，结果与上面相同

当发生除以0错误或出现商无法完全保存的情况，会发生除法溢出，此时CPU会在除法指令之前产生一个`int 00h`中断，在dos系统下，`int 00h`会显示溢出信息并终止程序运行。不过,我们可以通过修改`0:0`处的远指针（即`int 00h`的目标函数地址或中断向量）把我们自己的函数如`int_00h`与`int 00h`中断进行绑定，从而使得`int 00h`发生时让CPU来执行我们自己定义的中断函数`int_00h`。例程见**附录II：修改`int 00h`捕捉除法溢出**

位运算

`or`, `and`, `xor`, `not`

`or`按位或，`and`按位与，`xor`按位异或，`not`按位取反

`test`

`test`实际上做的是按位与，不保存结果，只改变标志位

> 说明：判断`cl`为`0`的几种写法：
>
> ```
> test cl, cl
> or cl, cl
> and cl, cl
> or cl, 0
> cmp cl, 0
> ```

`bit`, `bts`, `btr`, `btc` **（略）**

`shl`与`shr`

逻辑左移和逻辑右移，左边或右边的空位一律填`0`，最后移出去的一位一律在`CF`中。

移位运算的第二个操作数均应该为立即数或`cl`，实际上，移位的次数大小不大于`31`，

H5 否则只取用其模`32`的余数

`sal`与`sar`

`sal`： shift arithmetic left 算术左移

`sar`： shift arithmetic right 算术右移

H5

`sal`及`sar`是针对符号数的移位运算，对负数右移的时候要在左边补`1`，对正数右移的时候左边补`0`，无论对正数还是负数左移右边都补`0`。显然`sal`≡`shl`。最后移出去的一位一律在`CF`中

`shld`与`shrd`(略)

`rol`与`ror`

循环左移和循环右移，最后移出去的一位也会置入`CF`中

H5

`rcl`与`rcr`

H5

`rcl`： rotate through carry left 带进位循环左移

`rcr`： rotate through carry right 带进位循环右移

H5

这两个移位操作将`CF`看作寄存器较靠前一端的第一位

BCD码相关的指令（略）

小数运算（略）

字符串操作指令

H4

前缀`rep`和`repcc`

H4

`movs`系列

H3

`stos`系列

H4

H4

H4

`lods`系列

`scas`系列

`cmps`系列

跳转指令

`jmp`与`jcc`

`call`与`ret`

`callf`与`retf`

`int`与`iret`

`loop`

标志位控制指令

`stc`，`clc`，`cmc`

`sti`与`cli`

`std`与`cld`

`sahf`与`lahf`（略）

段寄存器指令

`lds`，`les`，`lfs`，`lgs`

端口编程指令：`in`与`out`

其他指令

`xlat`

`nop`

`lea`

多处理器接口：`esc`与`wait`（略）

模块化编程：`enter`与`leave`（略）

程序设计方法

循环和分支

函数和函数框架

常用中断

`int 00h`除法溢出

`int 01h`单步跟踪

`int 08h`时钟中断

`int 09h`键盘中断

`int 10h`显卡中断

`int 13h`硬盘中断

`int 16h`BIOS中断

`int 21h`DOS中断

端口编程

`60h`键盘


```
;--------------------------------------
;PrtSc/SysRq: E0 2A E0 37 E0 B7 E0 AA  ;
;Pause/Break: E1 1D 45 E1 9D C5        ;
;--------------------------------------
data segment
old_9h dw 0, 0
stop   db 0
key    db 0; key=31h
```

```
phead  dw 0
key_extend  db 'KeyExtend=', 0
key_up      db 'KeyUp=', 0
key_down    db 'KeyDown=', 0
key_code    db '00h ', 0
hex_tbl     db '0123456789ABCDEF'
cr          db  0Dh, 0Ah, 0
data ends

code segment
assume cs:code, ds:data
main:
   mov ax, data
   mov ds, ax
   xor ax, ax
   mov es, ax
   mov bx, 9*4
   push es:[bx]
   pop old_9h[0]
   push es:[bx+2]
   pop old_9h[2]    ; 保存int 9h的中断向量
   cli
   mov word ptr es:[bx], offset int_9h
   mov es:[bx+2], cs; 修改int 9h的中断向量
   sti
again:
   cmp [stop], 1
   jne again        ; 主程序在此循环等待
   push old_9h[0]
   pop es:[bx]
   push old_9h[2]
   pop es:[bx+2]    ; 恢复int 9h的中断向量
   mov ah, 4Ch
   int 21h

int_9h:
   push ax
   push bx
   push cx
   push ds
   mov ax, data
   mov ds, ax       ; 这里设置DS是因为被中断的不一定是我们自己的程
序
   in al, 60h       ; AL=key code
   mov [key], al
   cmp al, 0E0h
   je  extend
   cmp al, 0E1h
   jne up_or_down
extend:
   mov [phead], offset key_extend
   call output
```

```
    jmp check_esc
up_or_down:
    test al, 80h      ; 最高位=1时表示key up
    jz down
up:
    mov [phead], offset key_up
    call output
    mov bx, offset cr
    call display      ; 输出回车换行
    jmp check_esc
down:
    mov [phead], offset key_down
    call output
check_esc:
    cmp [key], 81h    ; Esc键的key up码
    jne int_9h_iret
    mov [stop], 1
int_9h_iret:
    mov al, 20h       ; 发EOI(End Of Interrupt)信号给中断控制器，
    out 20h, al
    ; 表示我们已处理当前的硬件中断(硬件中断处理最后都要这2条指令)。
    ; 因为我们没有跳转到的old_9h，所以必须自己发EOI信号。
    ; 如果跳到old_9h的话，则old_9h里面有这2条指令，这里就不要写。
    pop ds
    pop cx
    pop bx
    pop ax
    iret; 中断返回指令。从堆栈中逐个弹出IP、CS、FL。

output:
    push ax
    push bx
    push cx
    mov bx, offset hex_tbl
    mov cl, 4
    push ax   ; 设AL=31h=0011 0001
    shr al, cl; AL=03h
    xlat      ; AL = DS:[BX+AL] = '3'
    mov key_code[0], al
    pop ax
    and al, 0Fh; AL=01h
    xlat       ; AL='1'
    mov key_code[1], al
    mov bx, [phead]
    call display      ; 输出提示信息
    mov bx, offset key_code
    call display      ; 输出键码
    pop cx
    pop bx
    pop ax
    ret

```

```
display:
   push ax
   push bx
   push si
   mov si, bx
   mov bx, 0007h     ; BL = color
   cld
display_next:
   mov ah, 0Eh       ; AH=0Eh, BIOS int 10h的子功能，具体请查中断大全
   lodsb
   or al, al
   jz display_done
   int 10h           ; 每次输出一个字符
   jmp display_next
display_done:
    pop si
    pop bx
    pop ax
    ret
code ends
end main
```

70h与71h时钟

```
data segment
current_time db "00:00:00", 0Dh, 0Ah, "$"
data ends
code segment
assume cs:code, ds:data
main:
   mov ax, data
   mov ds, ax
   mov al, 4
   out 70h,al; index hour
   in al,71h ; AL=hour(e.g. 19h means 19 pm.)
   call convert; AL='1', AH='9'
   ;mov word ptr current_time[0],ax
   mov current_time[0], al
   mov current_time[1], ah
   mov al,2
   out 70h,al; index minute
   in  al,71h; AL=minute
   call convert
   mov word ptr current_time[3],ax;
   ;mov current_time[3], al
   ;mov current_time[4], ah
   mov al,0  ; index second
   out 70h,al
   in  al,71h; AL=second
```

```
   call convert
   mov word ptr current_time[6],ax
   mov ah, 9
   mov dx, offset current_time
   int 21h
   mov ah, 4Ch
   int 21h
;---------Convert----------------
;Input:AL=hour or minute or second
;      format:e.g. hour   15h means 3 pm.
;                  second 56h means 56s
;Output: (e.g. AL=56h)
;     AL='5'
;     AH='6'
convert:
    push cx
    mov ah,al ; e.g. assume AL=56h
    and ah,0Fh; AH=06h
    mov cl,4
    shr al,cl ; AL=05h
    ; shr:shift right右移
    add ah, '0'; AH='6'
    add al, '0'; AL='5'
    pop  cx
    ret
;---------End of Convert---------
code ends
end main
```

`40h`, `42h`, `43h`, `61h`声卡

H3

```
NOTE_1  =  440 ; 音调频率
NOTE_2  =  495
NOTE_3  =  550
NOTE_4  =  587
NOTE_5  =  660
NOTE_6  =  733
NOTE_7  =  825

ONE_BEEP  =  600 ; 一拍延时600ms
HALF_BEEP =  300 ; 半拍延时300ms

data segment
ticks dw 0
music dw  NOTE_5, ONE_BEEP
dw  NOTE_3, HALF_BEEP
dw  NOTE_5, HALF_BEEP
dw  NOTE_1*2, ONE_BEEP*2
dw  NOTE_6, ONE_BEEP
dw  NOTE_1*2, ONE_BEEP
```

```
dw  NOTE_5, ONE_BEEP*2
dw  NOTE_5, ONE_BEEP
dw  NOTE_1, HALF_BEEP
dw  NOTE_2, HALF_BEEP
dw  NOTE_3, ONE_BEEP
dw  NOTE_2, HALF_BEEP
dw  NOTE_1, HALF_BEEP
dw  NOTE_2, ONE_BEEP*4
dw  NOTE_5, ONE_BEEP
dw  NOTE_3, HALF_BEEP
dw  NOTE_5, HALF_BEEP
dw  NOTE_1*2, HALF_BEEP*3
dw  NOTE_7, HALF_BEEP
dw  NOTE_6, ONE_BEEP
dw  NOTE_1*2, ONE_BEEP
dw  NOTE_5, ONE_BEEP*2
dw  NOTE_5, ONE_BEEP
dw  NOTE_2, HALF_BEEP
dw  NOTE_3, HALF_BEEP
dw  NOTE_4, HALF_BEEP*3
dw  NOTE_7/2, HALF_BEEP
dw  NOTE_1, ONE_BEEP*4
dw  NOTE_6, ONE_BEEP
dw  NOTE_1*2, ONE_BEEP
dw  NOTE_1*2, ONE_BEEP*2
dw  NOTE_7, ONE_BEEP
dw  NOTE_6, HALF_BEEP
dw  NOTE_7, HALF_BEEP
dw  NOTE_1*2, ONE_BEEP*2
dw  NOTE_6, HALF_BEEP
dw  NOTE_7, HALF_BEEP
dw  NOTE_1*2, HALF_BEEP
dw  NOTE_6, HALF_BEEP
dw  NOTE_6, HALF_BEEP
dw  NOTE_5, HALF_BEEP
dw  NOTE_3, HALF_BEEP
dw  NOTE_1, HALF_BEEP
dw  NOTE_2, ONE_BEEP*4
dw  NOTE_5, ONE_BEEP
dw  NOTE_3, HALF_BEEP
dw  NOTE_5, HALF_BEEP
dw  NOTE_1*2, HALF_BEEP*3
dw  NOTE_7, HALF_BEEP
dw  NOTE_6, ONE_BEEP
dw  NOTE_1*2, ONE_BEEP
dw  NOTE_5, ONE_BEEP*2
dw  NOTE_5, ONE_BEEP
dw  NOTE_2, HALF_BEEP
dw  NOTE_3, HALF_BEEP
dw  NOTE_4, HALF_BEEP*3
dw  NOTE_7/2, HALF_BEEP
dw  NOTE_1, ONE_BEEP*3
```

```
dw  0, 0
data ends

code segment
assume cs:code, ds:data, ss:stk
main:
   mov ax, data
   mov ds, ax
   xor ax, ax
   mov es, ax
   mov bx, 8*4
   mov ax, es:[bx]
   mov dx, es:[bx+2]   ; 取int 8h的中断向量
   mov cs:old_int8h[0], ax
   mov cs:old_int8h[2], dx; 保存int 8h的中断向量
   cli
   mov word ptr es:[bx], offset int_8h
   mov es:[bx+2], cs   ; 修改int 8h的中断向量
   mov al, 36h
   out 43h, al
   mov dx, 0012h
   mov ax, 34DCh       ; DX:AX=1193180
   mov cx, 1000
   div cx              ; AX=1193180/1000
   out 40h, al
   mov al, ah
   out 40h, al         ; 设置时钟振荡频率为1000次/秒
   sti
   mov si, offset music
   cld
again:
   lodsw
   test ax, ax
   jz done
   call frequency
   lodsw
   call delay
   jmp again
done:
   cli
   mov ax, cs:old_int8h[0]
   mov dx, cs:old_int8h[2]
   mov es:[bx], ax
   mov es:[bx+2], dx   ; 恢复int 8h的中断向量
   mov al, 36h
   out 43h, al
   mov al, 0
   out 40h, al
   mov al, 0
   out 40h, al         ; 恢复时钟振荡频率为1193180/65536=18.2次/
秒
   sti
```

```
    mov ah, 4Ch
    int 21h

frequency:
    push cx
    push dx
    mov cx, ax   ; CX=frequency
    mov dx, 0012h
    mov ax, 34DCh; DX:AX=1193180
    div cx       ; AX=1193180/frequency
    pop dx
    pop cx
    cli
    push ax
    mov al, 0B6h
    out 43h, al
    pop ax
    out 42h, al ; n的低8位
    mov al, ah
    out 42h, al ; n的高8位
                ; 每隔n个tick产生一次振荡
                ; 振荡频率=1193180/n (次/秒)
    sti
    ret

delay:
    push ax
    cli
    in al, 61h
    or al, 3
    out 61h, al; 开喇叭
    sti
    pop ax
    mov [ticks], ax
wait_this_delay:
    cmp [ticks], 0
    jne wait_this_delay
    cli
    in al, 61h
    and al, not 3
    out 61h, al; 关喇叭
    sti
    ret

int_8h:
    push ax
    push ds
    mov ax, data
    mov ds, ax
    cmp [ticks], 0
    je skip
    dec [ticks]
```

```
skip:
   pop ds
   pop ax
   jmp dword ptr cs:[old_int8h]
old_int8h dw 0, 0
code ends

stk segment stack
dw 100h dup(0)
stk ends
end main
```

保护模式编程

附录

附录I：利用单步模式实现反调试

```
code segment
assume cs:code, ds:code
main:
    jmp begin
old1h dw 0, 0
prev_addr dw offset first, code;前条指令的地址
begin:
    push cs
    pop ds ;DS=CS
    xor ax, ax
    mov es, ax
    mov bx, 4
    push es:[bx]
    pop old1h[0]
    push es:[bx+2]
    pop old1h[2]
    mov word ptr es:[bx], offset int1h
    mov word ptr es:[bx+2], cs
    pushf ;save old FL
    pushf
    pop ax
    or ax, 100h ;1 0000 0000
    push ax
    popf ;TF=1
first:
    nop ;当某条指令执行前TF=1,则该条指令执行后会
        ;自动执行int 01h单步中断
single_step_begin:
;first int 1h
    xor ax, ax
    mov cx, 3
```

```
next:
    add ax, cx
    nop
    loop next
    popf ;restore old FL, TF=0
;last int 1h
    nop;
single_step_end:
    push old1h[0]
    pop es:[bx]
    push old1h[2]
    pop es:[bx+2]
    mov ah, 4Ch
    int 21h
int1h:
    push bp
    mov bp, sp
    push bx
    push es
    les bx, dword ptr cs:[prev_addr]
    inc byte ptr es:[bx]; 加密上一条指令
    les bx, dword ptr [bp+2]
    ;mov bx, [bp+2]
    ;mov es, [bp+4]; es:bx→下条指令的首字节
    dec byte ptr es:[bx]; 解密下一条指令
    mov cs:prev_addr[0], bx
    mov cs:prev_addr[2], es
    pop es
    pop bx
    pop bp
    iret
code ends
end main
```

附录II：修改`int 00h`捕捉除法溢出

H3

```
code segment
assume cs:code
old_00h dw 0, 0
int_00h:
    mov ch, 10h
    iret
main:
    push cs
    pop ds
    xor ax, ax
    mov es, ax
    mov bx, 0
    mov ax, es:[bx]
    mov dx, es:[bx+2]
```

```
    mov old_00h[0], ax
    mov old_00h[2], dx
    mov word ptr es:[bx], offset int_00h
    mov es:[bx+2], cs
    mov ax, 123h
    mov ch, 1
    div ch
    mov ax, old_00h[0]
    mov dx, old_00h[2]
    mov es:[bx], ax
    mov es:[bx+2], dx
    mov ah, 4Ch
    int 21h
code ends
end main
```