Assembly - Study

一.运行第一个程序

1.首先简单编写一段代码

```
    STACK   SEGMENT  PARA  'STACK'                ;定义堆栈段，段名为 STACK（可以取其他的）
        DB 100 DUP('?')                          ;分配堆栈的大小，设置为100字节，以？填充
STACK   ENDS                                     ;堆栈段结束
DATA    SEGMENT                                  ;定义数据段，段名为 DATA （可以取其他的）
        STRING  DB 'HELLO!','$'                  ;定义字符串数据
DATA    ENDS
CODE    SEGMENT                                  ;定义代码段，段名为 CODE （可以取其他的）
        ASSUME CS:CODE, DS:DATA, SS:STACK        ;特别重要，将 CS、DS、SS 指向定义的段
START:  MOV  AX,  DATA                           ;程序开始的地方
        MOV  DS,  AX
        MOV  AX,  STACK
        MOV  SS,  AX
        LEA  DX,  STRING
        MOV  AH, 09H
        INT  21H
        MOV  AH,  4CH
        INT  21H
CODE    ENDS
        END START                                ;程序结束的地方
//输出HELLO!
```

2.对该代码文件进行编译，如果有错误会进行报错，如果没有错误的话，在编译器masm.exe运行的目录D:/Assembly/ASM下，会出现一个新的文件hello.obj

```
masm hello.asm
```

3.对obj文件进行连接，连接成可执行文件

```
link hello.obj
```

4.对可执行文件进行调试：

```
debug hello.exe
```

二.对上面程序的简单的解析

三.汇编指令

3.1 OFFSET和LEA

二者都是得到某一个变量或者其他的偏移地址，二者相比起来，offset是**伪指令**,(**伪指令**与**汇编指令**相对，汇编指令是有相应的机器码一一对应的，而伪指令是由编译器来执行的，就是只有编译器才认识的指令，编译器会进行相关的编译工作，最常见的伪指令，比如说我们描述代码段、数据段、堆栈段的时候，使用到的data segment ...data ends，这些都是伪指令，是由编译器进行编译的。)offset是伪指令，他是在编译的时候执行的，即编译阶段得出偏移地址，lea在程序执行的时候才会得到相应的偏移地址。

从速度上来讲，offset要比lea来的快！但是，从复杂度和灵活性来说，lea则远远超过offset。

seg也可以做到

3.2 INT 21H

我们以8086 CPU的汇编为例，输出一个字符串，就要使用如下的指令：

```
mov ah,09h
int 21h
```

我们上面的指令就是要让显示器输入一个字符串，我们要实现这个功能，我们实际上就是要调用DOS系统的功能来实现，DOS系统含有很多的功能，比如接受用户输入`(mov ah,01h)`，显示输出`(mov ah,02h)`，每一个功能都有一个编号，该编号是存储在ah寄存器里面。

```
mov ah, 09h
int 21h
```

上面的代码是先将ah寄存器里面的值赋为09h，然后调用汇编指令`int 21h`，该指令为中断指令，因为我们需要调用DOS里面的功能，所以我们需要说，大家都停止运行，这个时候DOS系统会查好ah里面的内容，然后执行相关的功能。

所以，我们可以将`int 21h`理解为命令系统开始工作的意思，至于做什么工作，那得看我们事先将什么编码放进ah寄存器中。

AH	功能	调用参数	返回参数
00	程序终止(同INT 20H)	CS=程序段前缀	
01	键盘输入并回显	AL=输入字符	
02	显示输出	DL=输出字符	
03	异步通讯输入	AL=输入数据	
04	异步通讯输出	DL=输出数据	
05	打印机输出	DL=输出字符	
06	直接控制台I/O	DL=FF(输入) DL=字符(输出)	AL=输入字符
07	键盘输入(无回显)	AL=输入字符	
08	键盘输入(无回显) 检测Ctrl-Break	AL=输入字符	
09	显示字符串	DS:DX=串地址 '$'结束字符串	
0A	键盘输入到缓冲区	DS:DX=缓冲区首地址 (DS:DX)=缓冲区最大字符数	(DS:DX+1)=实际输入的字符数
0B	检验键盘状态	AL=00 有输入 AL=FF 无输入	
0C	清除输入缓冲区并 请求指定的输入功能	AL=输入功能号 (1,6,7,8,A)	
0D	磁盘复位	清除文件缓冲区	
0E	指定当前缺省的磁盘驱动器	DL=驱动器号 0=A,1=B,...	AL=驱动器数
0F	打开文件	DS:DX=FCB首地址	AL=00 文件找到 AL=FF 文件未找到
10	关闭文件	DS:DX=FCB首地址	AL=00 目录修改成功 AL=FF 目录中未找到文件
11	查找第一个目录项	DS:DX=FCB首地址	AL=00 找到 AL=FF 未找到
12	查找下一个目录项	DS:DX=FCB首地址 (文件中带有*或?)	AL=00 找到 AL=FF 未找到
13	删除文件	DS:DX=FCB首地址	AL=00 删除成功 AL=FF 未找到
14	顺序读	DS:DX=FCB首地址	AL=00 读成功 =01 文件结束,记录中无数据 =02 DTA空间不够 =03 文件结束,记录不完整
15	顺序写	DS:DX=FCB首地址	AL=00 写成功 =01 盘满 =02 DTA空间不够
16	建文件	DS:DX=FCB首地址	AL=00 建立成功 =FF 无磁盘空间
17	文件改名	DS:DX=FCB首地址 (DS:DX+1)=旧文件名 (DS:DX+17)=新文件名	AL=00 成功 AL=FF 未成功
19	取当前缺省磁盘驱动器	AL=缺省的驱动器号 0=A,1=B,2=C,...	
1A	置DTA地址	DS:DX=DTA地址	

1B	取缺省驱动器FAT信息	AL=每簇的扇区数 DS:BX=FAT标识字节 CX=物理扇区大小 DX=缺省驱动器的簇数	
1C	取任一驱动器FAT信息	DL=驱动器号	同上
21	随机读	DS:DX=FCB首地址	AL=00 读成功 =01 文件结束 =02 缓冲区溢出 =03 缓冲区不满
22	随机写	DS:DX=FCB首地址	AL=00 写成功 =01 盘满 =02 缓冲区溢出
23	测定文件大小	DS:DX=FCB首地址	AL=00 成功(文件长度填入FCB) AL=FF 未找到
24	设置随机记录号	DS:DX=FCB首地址	
25	设置中断向量	DS:DX=中断向量 AL=中断类型号	
26	建立程序段前缀	DX=新的程序段前缀	
27	随机分块读	DS:DX=FCB首地址 CX=记录数	AL=00 读成功 =01 文件结束 =02 缓冲区太小,传输结束 =03 缓冲区不满
28	随机分块写	DS:DX=FCB首地址 CX=记录数	AL=00 写成功 =01 盘满 =02 缓冲区溢出
29	分析文件名	ES:DI=FCB首地址 DS:SI=ASCIIZ串 AL=控制分析标志	AL=00 标准文件 =01 多义文件 =02 非法盘符
2A	取日期	CX=年 DH:DL=月:日(二进制)	
2B	设置日期	CX:DH:DL=年:月:日	AL=00 成功 =FF 无效
2C	取时间	CH:CL=时:分 DH:DL=秒:1/100秒	
2D	设置时间	CH:CL=时:分 DH:DL=秒:1/100秒	AL=00 成功 =FF 无效
2E	置磁盘自动读写标志	AL=00 关闭标志 AL=01 打开标志	
2F	取磁盘缓冲区的首址	ES:BX=缓冲区首址	
30	取DOS版本号	AH=发行号,AL=版本	
31	结束并驻留	AL=返回码 DX=驻留区大小	
33	Ctrl-Break检测	AL=00 取状态 =01 置状态(DL) DL=00 关闭检测 =01 打开检测	DL=00 关闭Ctrl-Break检测 =01 打开Ctrl-Break检测
35	取中断向量	AL=中断类型	ES:BX=中断向量

36	取空闲磁盘空间	DL=驱动器号 0=缺省,1=A,2=B,...	成功:AX=每簇扇区数 BX=有效簇数 CX=每扇区字节数 DX=总簇数 失败:AX=FFFF
38	置/取国家信息	DS:DX=信息区首地址	BX=国家码(国际电话前缀码) AX=错误码
39	建立子目录(MKDIR)	DS:DX=ASCIIZ串地址	AX=错误码
3A	删除子目录（RMDIR）	DS:DX=ASCIIZ串地址	AX=错误码
3B	改变当前目录(CHDIR)	DS:DX=ASCIIZ串地址	AX=错误码
3C	建立文件	DS:DX=ASCIIZ串地址 CX=文件属性	成功:AX=文件代号 错误:AX=错误码
3D	打开文件	DS:DX=ASCIIZ串地址 AL=0 读 =1 写 =3 读/写	成功:AX=文件代号 错误:AX=错误码
3E	关闭文件	BX=文件代号	失败:AX=错误码
3F	读文件或设备	DS:DX=数据缓冲区地址 BX=文件代号 CX=读取的字节数	读成功: AX=实际读入的字节数 AX=0 已到文件尾 读出错:AX=错误码
40	写文件或设备	DS:DX=数据缓冲区地址 BX=文件代号 CX=写入的字节数	写成功: AX=实际写入的字节数 写出错:AX=错误码
41	删除文件	DS:DX=ASCIIZ串地址	成功:AX=00 出错:AX=错误码(2,5)

42	移动文件指针	BX=文件代号 CX:DX=位移量 AL=移动方式(0:从文件头绝对位移,1:从当前位置相对移动,2:从文件尾绝对位移)	成功:DX:AX=新文件指针位置 出错:AX=错误码
43	置/取文件属性	DS:DX=ASCIIZ串地址 AL=0 取文件属性 AL=1 置文件属性 CX=文件属性	成功:CX=文件属性 失败:CX=错误码
44	设备文件I/O控制	BX=文件代号 AL=0 取状态 =1 置状态DX =2 读数据 =3 写数据 =6 取输入状态 =7 取输出状态	DX=设备信息
45	复制文件代号	BX=文件代号1	成功:AX=文件代号2 失败:AX=错误码
46	人工复制文件代号	BX=文件代号1 CX=文件代号2	失败:AX=错误码
47	取当前目录路径名	DL=驱动器号 DS:SI=ASCIIZ串地址	(DS:SI)=ASCIIZ串 失败:AX=出错码
48	分配内存空间	BX=申请内存容量	成功:AX=分配内存首地 失败:BX=最大可用内存
49	释放内容空间	ES=内存起始段地址	失败:AX=错误码
4A	调整已分配的存储块	ES=原内存起始地址 BX=再申请的容量	失败:BX=最大可用空间 AX=错误码
4B	装配/执行程序	DS:DX=ASCIIZ串地址 ES:BX=参数区首地址 AL=0 装入执行	失败:AX=错误码

4C	带返回码结束	AL=返回码	
4D	取返回代码	AX=返回代码	
4E	查找第一个匹配文件	DS:DX=ASCIIZ串地址 CX=属性	AX=出错代码(02,18)
4F	查找下一个匹配文件	DS:DX=ASCIIZ串地址 (文件名中带有?或*)	AX=出错代码(18)
54	取盘自动读写标志	AL=当前标志值	
56	文件改名	DS:DX=ASCIIZ串(旧) ES:DI=ASCIIZ串(新)	AX=出错码(03,05,17)
57	置/取文件日期和时间	BX=文件代号 AL=0 读取 AL=1 设置(DX:CX)	DX:CX=日期和时间 失败:AX=错误码
58	取/置分配策略码	AL=0 取码 AL=1 置码(BX)	成功:AX=策略码 失败:AX=错误码
59	取扩充错误码	AX=扩充错误码 BH=错误类型 BL=建议的操作 CH=错误场所	
5A	建立临时文件	CX=文件属性 DS:DX=ASCIIZ串地址	成功:AX=文件代号 失败:AX=错误码
5B	建立新文件	CX=文件属性 DS:DX=ASCIIZ串地址	成功:AX=文件代号 失败:AX=错误码
5C	控制文件存取	AL=00封锁 =01开启 BX=文件代号 CX:DX=文件位移 SI:DI=文件长度	失败:AX=错误码

62	取程序段前缀	BX=PSP地址	

3.3 PTR

PTR--pointer（指针）的缩写。

PTR在汇编中是临时的类型转换，相当于C语言中的强制类型转换。

比如`mov ax,bx`是将BX寄存器里面的值赋予给ax，由于二者都是寄存器，长度已经确定（word），所以就没有必要使用ptr。

但是`mov ax,word ptr[bx]`，就不一样了。该指令所做的事情是将内存地址等于BX寄存器的值的数据赋予AX寄存器，但是你只给出一个地址，准确的来讲是指给出了一个数据的起始地址，至于该数据的大小怎么样我们是不知道的，所以说我们需要用word明确指出是字类型。如果不使用的话，即`mov ax,[bx]`则默认是传递一个字，即两个字节给ax寄存器。

以下例子，默认指明指令进行的是字节操作：

```
mov ax,1
mov bx,ds:[0]
mov ds,ax
mov ds:[0],ax
inc ax
add ax,1000
```

以下例子，寄存器指明了指令进行的是字节操作（因为是AL）

```
mov al,1
mov al,bl
mov al,ds:[0]
mov ds:[0],al
inc al
add al,100
```

以下例子，没有寄存器名存在的情况，在内存中操作我们不知道具体的大小，所以需要使用ptr来指明内存单元的大小，不指明的话，内存是一片连续的区域，操作就乱了：

```
mov word ptr ds:[0],1;该指令是指将立即数1存放到数据段的ds:0地址处，大小为一个字
inc word ptr [bx];该指令是指将以bx寄存器内容为起始地址大小为一个字的数据加一
inc word ptr ds:[0]
add word ptr [bx],2
```

在没有寄存器参与的内存单元访问指令中，使用`word ptr`或者`byte ptr`显性的指明所要访问的内存单元的长度是很必要的，否则CPU无法得知所要访问的单元，是字节单元还是字单元。

但是还有一些比较特殊的指令，比如说`push [100h]`就无需指明是将一个字节数据还是一个字数据压栈，因为`push`指令只进行字操作。

3.4 JMP寻址时候的short，near,far区别

jmp位无条件转移指令，可以指修改IP，也可以同时修改CS和IP。

jmp指令要给出两种信息：

a. 转移的目的地址；

b. 转移的距离（段间距离、段内距离、段内近距离）

示例1：

`jmp short 标号`转移标号处执行指令

IP修改范围位：-128~127，short说明进行的是短转移。

```
assume cs:code
code segment
start :
    mov ax,0
    jmp short s
    ass ax,1
s:  inc ax
code ends
end start
```

示例2：

`jmp near ptr 标号`，(IP) = (IP) + 16位位移，实现的是段内近转移

IP的修改范围：-32768~32767，用补码来表示

示例3：

`jmp far ptr 标号`实现的是段间转移（远转移）

既然都是段间转移，说明该指令改变的不仅仅是IP，还有CS的值；

3.5 proc near/far

PROC是定义子程序，即定义子函数的伪指令，位置在于子程序的开始处，他和endp分别表示子程序定义的开始和结束，二者必须成对的出现。

near属性，（段内近调用），调用程序和子程序位于同一代码段中，只能被相同的代码段的其他程序所调用；

far属性，（段间远调用），调用程序和子程序不在同一代码段中，可以被相同或者不同代码段的程序调用；

ps:主程序之所以定义为far的原因是：

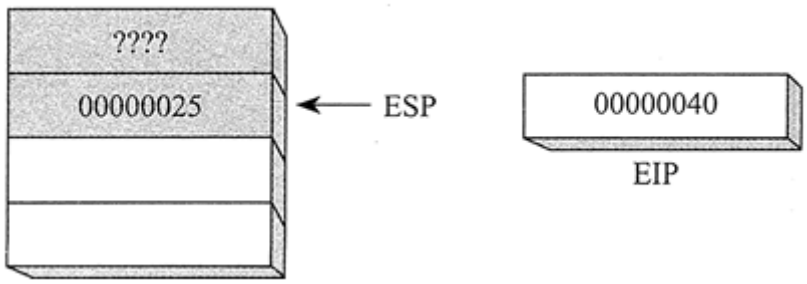
系统将主程序当作DOS调用的一个子程序，DOS内核程序和主程序不是在同一个段地址内，所以主程序要使用far。

3.6 CALL和RET

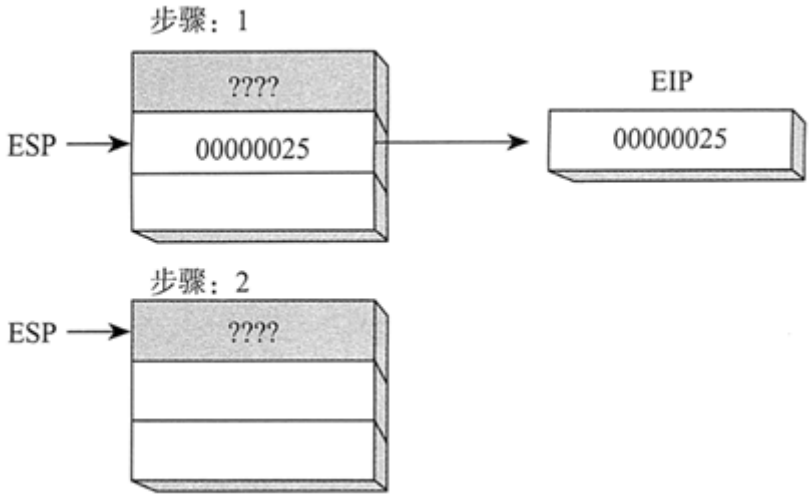
参考文章：http://c.biancheng.net/view/3537.html

CALL指令调用一个程序，指挥处理器从新的内存地址开始执行。程序将使用RET指令将处理器转回到该过程被调用的程序点上。

调用的过程就是函数栈的工作方式，调用子函数的时候，提前需要保存CALL指令之后的指令的地址，然后将子函数的内存地址赋值给DS：IP。



待得子函数结束之后，执行RET指令，SP（栈指针）指向的堆栈值弹出到IP，然后SP得数值增加2，指向堆栈中的前一个值。



这里注意，我们主程序调用子程序的过程中栈的变化，否则的话，会导致在子程序中栈的混乱！

主程序使用命令`call`的过程中，系统中的执行的汇编代码如下：

```
push ip
inc ip
mov ax,ip
pop ip
push ax;将下一条指令的偏移地址入栈保存
mov ip,子程序的内存地址
；开始执行子程序
...
；当子程序执行到ret指令的时候：
pop ip;将之前保存的下一条指令的地址弹栈，放在ip寄存器中
add sp,2;栈指针，加2，指向栈顶
```

RET指令：

ret指令分为两种：`retn` `retf`

`retn`是近返回，用于段内返回，他返回堆栈中保存的本段内的偏移地址；

`pop ip`

`retf`是远返回，一般用于段间返回，他返回堆栈中保存的段地址和偏移地址

`pop ip; pop cs`

所以如果我们没有设计好对应的结束程序的代码的话，最好在ret之前利用DOS功能调用来结束程序。

3.7 高级压栈指令：

`pusha`:将寄存器`ax,cx,dx,bx,sp,bp,si,di`依次压入堆栈；

`popa`:将寄存器`di,si,bp,bx,dx,cx,ax`依次弹出堆栈；

`pushad`:将寄存器`EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI`依次压入堆栈；

`popad`:将寄存器`EDI,ESI,EBP,ESP.EBX,EDX,ECX,EAX`依次弹出堆栈；

3.8 串操作指令

串操作指令在一开始体会不到他的用处，就好比在学习高级语言的开始时，不理解strcpy这些函数一样，串操作指令就是和strcpy这样的函数一样的目的，都是对操作数进行操作的，***在汇编中涉及到操作串，肯定离不开si和di寄存器，si寄存器和ds寄存器结合指向数据段中的某一个串***；***di和ss寄存器结合用来指向附加段中的某一个串。***

3.9 TEST指令

格式：`TEST OPR1,OPR2`

执行的操作：`(OPR1 ^ OOPR2)`

两个操作数相与的结果不保存，只根据其结果来改变标志位；

示例：

```
1
```

3.10 汇编标志位 NV UP EI NG NZ AC PE CY

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	0	0	0	of	df	if	tf	sf	zf	0	af	0	pf	0	cf

ZF是flag的第6位，零标志位，判断结果是否为0，结果为0，ZF=1
PF是flag的第2位，奇偶标志位，运算结果二进制数中1的个数为偶数，PF=1
SF是flag的第7位，符号标志位，有符号数 运算结果为负数，SF=1
CF是flag的第0位，进位标志位，无符号数 运算结果有进/借位，CF=1
OF是flag的第11位，溢出标志位，有符号数 运算结果溢出，OF=1
DF是flag的第10位，方向标志位，DF=0 每次操作后 si,di递增
DF=1 每次操作后 si,di递减
TF是flag的第8位，TF=1，产生单步中断，引发中断过程
IF是flag的第9位，IF=1，CPU响应中断，引发中断过程
IF=0，不响应可屏蔽中断

add、sub、mul、div 、inc、or、and等运算指令影响flag
mov、push、pop等传送指令对flag没影响

abc 带位加法指令，利用CF位上记录的进位值
abc ax,bx 实现功能(ax)=(ax)+(bx)+CF

sbb 带位减法指令，利用CF位上记录的借位值
sbb ax,bx 实现功能(ax)=(ax)-(bx)-CF

OF 溢出标志位：Overflow of = ***OV NV***[No Overflow]；

ZF 零标志位：ZF=1，结果为0，***ZR[zero] ---- NZ[Not zero]***；

PF 奇偶标志位：PE=1，运算结果二进制数中的1的个数为偶数，***PE[even] 偶数 ---- PO[odd]奇数***；

SF 符号标志位：SF=1，运算结果为负数，***NG[negative] ---- PL[positive]***；

DF 方向标志位：DF=0，每次操作后，si,di,递增，DF=1，每次操作后，si,di递减，***DN[decrement] ---- UP[increment]***；

TF 产生单步中断，引发中断过程：TF=1，产生单步中断，引发中断过程；

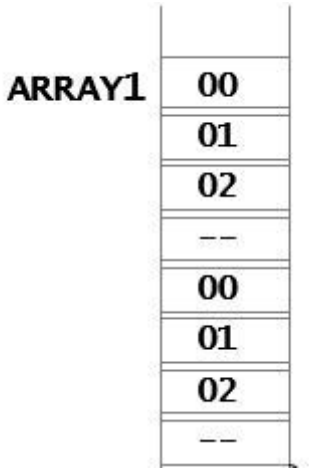
IF CPU响应中断，引发中断：IF=1，CPU响应中断；IF=0，不响应，屏蔽中断，***EI[enabled] ---- DI[disables]***；

3.11 汇编指令dup的使用

示例1：

```
array1 db 2 dup(0,1,2,?)
```

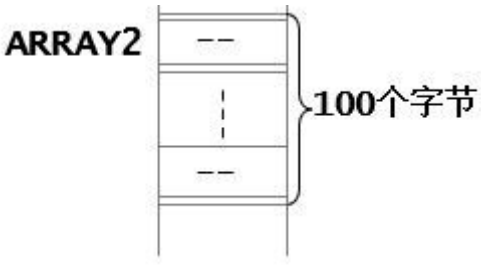
则其在内存中的数据结构是：



示例2：

```
array2 db 100 dup(?)
```

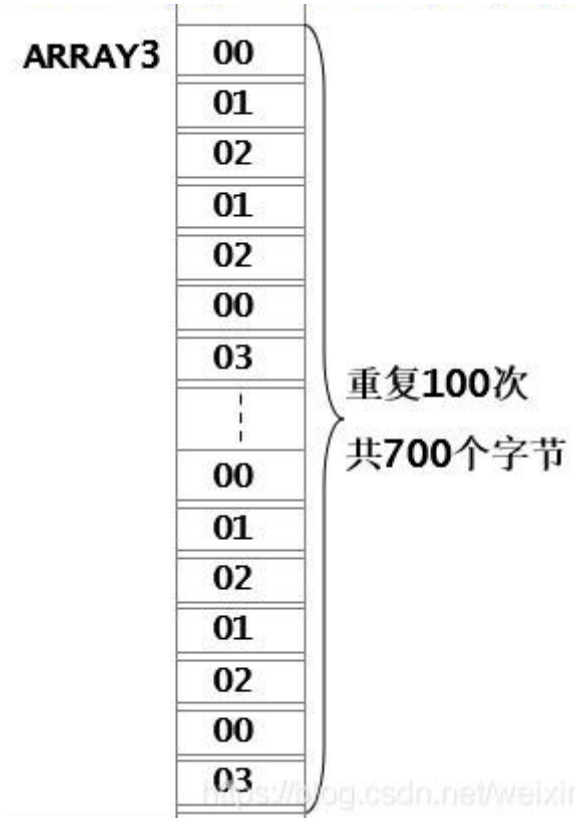
则其在内存中的数据结构是：



示例3（***dup的嵌套***）

```
array3 db 100 dup(0,2 dup(1,2),0,3)
```

则其在内存中的数据结构是：



四.汇编语言在数据段使用数据定义伪指令 定义 变量

```
1.db(byte,1字节);define byte
2.dw(word,2字节);define word
3.dd(double word,双字，四字节);define double
示例如下：
data segment
    a db 1;定义一个字节型变量，名称为a，初始值为01h
    b db "1";定义一个字符型变量，名称为b,初始值为"1"
    c dw 1;定义一个字型变量，名称为c,初始值为0001h
    d dd 1;定义一个双字型变量，名称为d,初始值为0000 0001h
    buf db 50 dup(1);定义连续50个字节型变量，名称为buf，初始值为01h
    buf1 db "1234567";定义一个字符型变量，名称为buf1,初始值为"1234567"
    ;dup的使用格式
    ;db 重复的次数 dup(重复的字节型数据)
    ;dw 重复的次数 dup(重复的字型数据)
    ;dd 重复的次数 dup(重复的双子型数据)

    ;举例
    ORG 1000H
    ;ORG伪指令的作用是给下面一条语句指定起始偏移地址，通常，段定义语句segment指出段的起点，偏移地址为    ;0，段内各个语句或者数据的地址，将会由段起始地址开始依次往后面后退.
    ;当要对某条指令或者某些数据规定特殊的存放地址时，可以使用ORG伪指令来实现
    X DB 12H
    Y DW X
    ;此处Y存放的不是X变量的值，而是X变量的地址
    Z DD Y


dataends
```

五. 与数据有关的寻址方式（针对某一个操作数所讲）

这种寻址方式用来确定操作数地址从而找到操作数。

讲人话就是，我们如果要使用到某一个数据的话，怎么利用汇编指令将该数据找到。

5.1 立即寻址方式（立即数寻址）

操作数直接放在操作数位上，作为指令的一部分。执行速度快，主要是给寄存器赋值。

```
MOV AX,14
;主要是给寄存器赋值，不能直接给段寄存器赋值
;前操作数是寄存器寻址，后操作数是立即数寻址
```

5.2 寄存器寻址方式

操作数在寄存器里面，直接去寄存器中寻找操作数。

```
MOV AX,BX;两个操作数都是寄存器寻址
```

5.3 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）

四种成分	16位寻址	32位寻址
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

5.3.1 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
```

5.3.2 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
```

5.3.3 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]
......
```

5.3.4 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]`；

5.3.5 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]
```

5.3.6 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]`

5.4 I/O地址寻址

5.4.1 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；

5.4.2 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`

5.5 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

5.5.1 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`

5.5.2 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
```

5.5.3 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`

5.5.4 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]
```

六. 汇编中常见操作

6.1 输出多位数

一般思路是利用除法，从而得到余数（每一个位数），然后将对应的余数转换成字符并输出。

> 除法DIV：
>
> 一般格式：div reg或者div 内存单元，reg里面和内存单元中存放的是除数，除数可分为8位和16位两种
>
> ***当除数为8位的时候：***
>
> - 被除数默认为16位，被除数默认存放在AX寄存器中
> - AL存储除法操作结果的商，AH存储除法的余数
>
> ***当除数为16位的时候：***
>
> - 被除数为32位，DX存放高16位，AX存放低16位
> - AX存储操作结果的商，DX存储操作结果的余数

```
;以下内容是定义的一个子程序，实现的功能是将多位数ax以十进制输出
decimal proc near
    push ax
    push cx
    push dx
    push bx

    cmp ax,0
    jge no_negative
    ;如果整数为负数，先输出负号，然后求补转化成整数再输出相应的数字
    mov bx,ax
```

```
    mov dl,'-'
    mov ah,2
    int 21h
    neg bx ;求补
    mov ax,bx

no_negative:
    mov cx,0
    mov bx,10
de:
    xor dx,dx
    div bx
    push dx ;将整数的低位压栈，低位先入栈
    inc cx;计数器，记录整数的位数
    cmp ax,0
    jnz de;非零跳转，如果这个整数还没有除尽，就继续除
de1:
    pop dx;将余数一位一位的输出，从高位往低位
    add dl,30h;将数字转化为字符
    mov ah,2 ;输出
    int 21h
    loop de1 ;循环输出

    pop bx
    pop dx
    pop cx
    pop ax
    ret;将处理器回到该程序被调用的程序上，即相当于高级语言中的return
decimal endp
```

6.2 输出16进制数字

6.3 简单的冒泡排序法

```
;该汇编程序的功能是实现基本的冒泡排序
data segment
    arrays dw dup(0)
data ends
assume ds:data,cs:code
code segment
main proc far
start:
    push ds
    sub ax,ax
    push ax
    mov ax,data
    mov ds,ax
    mov cx,10
    dec cx;最外层循环，一共要循环n-1次
loop1:
    mov di,cx
    mov bx,0
loop2:
    mov ax,arrays[bx]
```

```
    cmp ax,arrays[bx + 2];数据是以字为单位的
    jle continue;如果前面的数据值比后面的数据值小的话，继续往下面执行
    xchg ax,arrays[bx+2]
continue:
    add bx,2
    loop loop2

    mov cx,di;外层循环次数
    loop loop1
    ret
main endp
code ends
end start
```

七. 关于栈的123

栈和我们高级语言中的栈有所不一样，高级语言中，栈的作用主要是与函数的调用有关，涉及到参数进栈，函数地址进栈还有记录返回的内存地址进栈等等。

但是在汇编中，由于我们是在微观中看世界，所以很多操作都需要利用栈来操作进行，比如说在汇编中我们的寄存器的数量是有限的，不可能像高级语言中随便定义变量，所以说当我们有大量的变量需要大量的使用的话，我们就会先将寄存器中值压栈进行存储，下次使用的时候再出栈将其取出。

栈的实现离不开堆栈寄存器SS和栈顶寄存器SP，当栈为空的时候，栈顶指针指向***栈最底部的字单元的偏移地址+2***。

注意注意注意！！！！在栈中的操作（出栈和入栈）都是以字为单位的！！！

栈顶超界的问题，栈顶超界是十分的危险的，比如说我们有可能在栈空间的外面存放了具有其他用途的数据、代码等等，然后由于我们没有事先考虑好栈的空间大小问题，导致栈顶超界，从而造成数据、代码的意外改写。

所以说，我们在编程的时候需要自己操心栈顶会不会超界的问题，要根据可能要用到的最大栈空间，来安排栈的大小，防止入栈的数据过多导致栈顶超界，出栈的时候也要注意，防止将栈以外的数据出栈。

八. 一段安全的空间

在8086模式下，随意的向内存空间写入内容是很危险的，因为这段空间中很可能存放着重要的系统数据或者代码。比如下列指令：

```
mov ax,1000h
mov ds,ax
mov dl,0
mov ds:[0],al
```

我们之前很多都是为了说明某一些指令的功能才这样去书写，但是实际上来说，我们想要写入的地址中很可能存放着很重要的系统数据或者代码，如果我们将其改写的话，会引发系统的错误。

所以一般有以下规则：

> DOS方式下，一般情况，0：200~0：2ff空间中是没有系统或者其他程序的数据或者代码的；
>
> 所以说我们需要直接向一段内存中写入内容的时候，就是用0：200~0:2ff这一段空间；

九. 包含多个段的程序

通过上面所述，0：200~0：2ff空间是相对安全的，但是也不是说我们除了这一段空间，其他的内存空间我们不能使用，这样显然是不可能的！

在操作系统中，我们合法的通过操作系统取得的空间都是安全的，这是肯定的，我们向操作系统申请空间，操作系统是不会让一个程序所使用的空间和系他的程序相冲突的，在操作系统的允许下，我们是可以取得任意容量的空间的。

通常来说，程序取得所需空间的方法有两种：一种是在加载程序的时候，为程序分配（就是在一开始就将所有的变量申请空间）；二就是程序在执行的过程中向系统申请。

为了程序设计的清晰与方便，我们最好将代码、变量数据、堆栈有关分开存放，形成相应的段。否则不这样做的话，我们的程序就会显得十分的混乱。

程序1：

```
assume cs :code
code segment
    dw 0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h
    mov bx,0
    mov ax,0
    mov cx,8

s:add ax,cs:[bx]
    add bx,2
    loop s
    mov ax,4c00h
    int 21h
code ends

end
```

我们可以使用`r`命令查看当前寄存器内容，使用`u`命令可以查看程序：

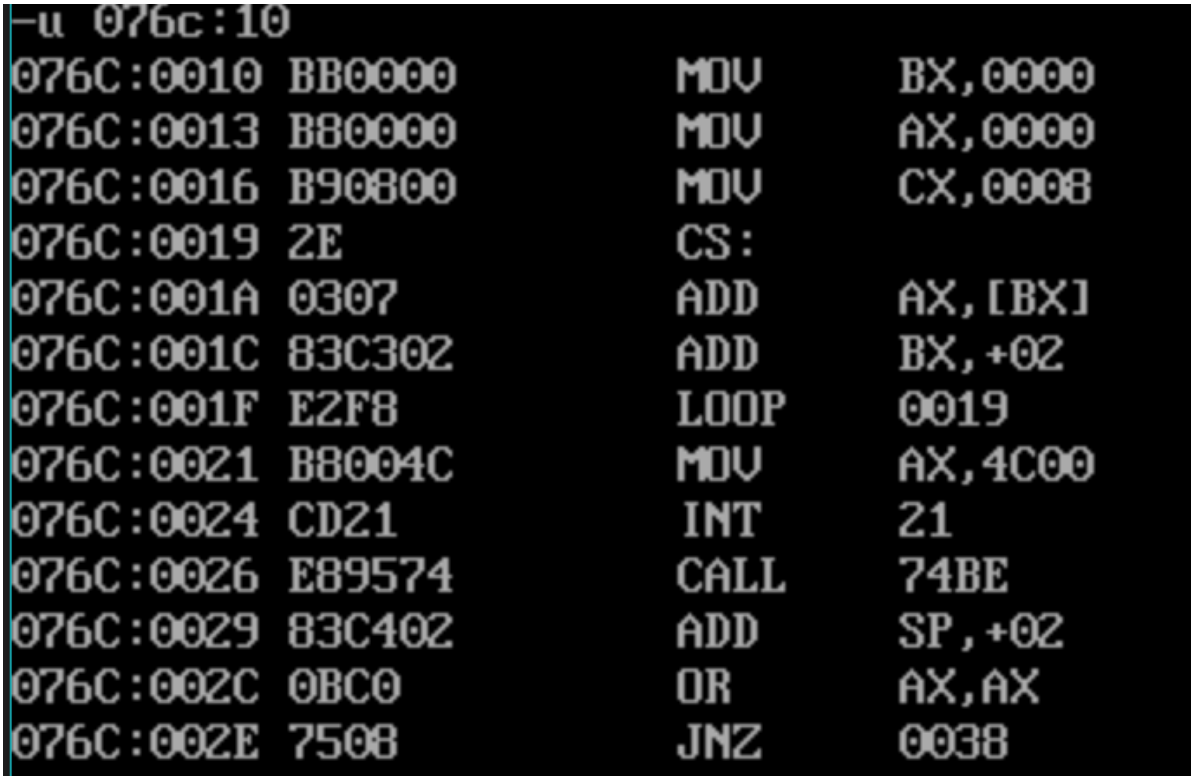
```
-u
076C:0021 B8004C          MOV     AX,4C00
076C:0024 CD21            INT     21
076C:0026 E89574          CALL    74BE
076C:0029 83C402          ADD     SP,+02
076C:002C 0BC0            OR      AX,AX
076C:002E 7508            JNZ     0038
076C:0030 B8FFFF          MOV     AX,FFFF
076C:0033 5E              POP     SI
076C:0034 8BE5            MOV     SP,BP
076C:0036 5D              POP     BP
076C:0037 C3              RET
076C:0038 B88001          MOV     AX,0180
076C:003B 50              PUSH    AX
076C:003C FF7604          PUSH    [BP+04]
076C:003F E85673          CALL    7398
```

使用`d`命令可以查看程序的前16个字节的内容：

```
-d 076c:0
076C:0000  23 01 56 04 89 07 BC 0A-EF 0D ED 0F BA 0C 87 09   #.V.............
076C:0010  BB 00 00 B8 00 00 B9 08-00 2E 03 07 83 C3 02 E2   ................
076C:0020  F8 B8 00 4C CD 21 E8 95-74 83 C4 02 0B C0 75 08   ...L.!..t.....u.
076C:0030  B8 FF FF 5E 8B E5 5D C3-B8 80 01 50 FF 76 04 E8   ...^..]....P.v..
076C:0040  56 73 83 C4 04 89 46 FC-40 75 20 83 3E 13 02 18   Vs....F.@u .>...
076C:0050  75 13 E8 0B 3C B8 80 01-50 FF 76 04 E8 39 73 83   u...<...P.v..9s.
076C:0060  C4 04 89 46 FC 83 7E FC-FF 74 C5 FF 76 FC E8 1B   ...F..~..t..v...
076C:0070  79 83 C4 02 2B C0 5E 8B-E5 5D C3 90 55 8B EC 83   y...+.^..]..U...
```

可以看到程序的前16个字节存放着我们定义的数据；

我们使用指令`u 076c:10`可以查看程序中要执行的机器指令：



程序2：

在代码段中使用栈，主要问题是首先要有一段可以当作栈的内存空间，我们可以在程序中通过定义数据来取得一段空间，然后将这一段栈空间当作栈空间来用。

示例：

```
assume cs:codesg

codesg segment
    dw 0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h

    dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

start:
    mov ax,cs
    mov ss,ax   ;段寄存器之间不能相互赋值
    mov sp,30h  ;通过计算，我们可以知道，我们申请的从00h到10h是存放数据的，10h到30h是存放数据0的，就是我们即将用作栈的空间
                ;以上代码段，将段寄存器指向代码段，将栈底指针指向30h
                ;cs:10~cs:2f的内存空间当作栈来用，初始状态下栈为空，所以ss:sp需要指向栈底，2e加2就是30，因为在栈中是以字为单位操作的
    mov bx,0
    mov cx,8
s:
    push cs:[bx]
    add bx,2
    loop s
    mov bx,0
    mov cx,0
s0:
    pop cs:[bx]
    add bx,2
    loop s0
```

```
    mov ax,4c00h
    int 21h

codesg ends
end start

```

将数据、代码、栈放入不同的段中：

如果将数据、栈、代码都放入到一个段中，就会引发两个问题：

- 将他们放在一个段中使得程序显得混乱；
- 前面的程序中处理的数据比较少，用到的栈空间也少，加上没有多长的代码，放到一个段中没有问题。但是如果数据、栈、和代码需要的空间超过64kb，就不可以放在同一个段中了；

> ***一个段的容量不可以大于64KB，但是注意，这仅仅是我们在学习所用的8086模式的限制，并不是所有的处理器都是这样的！为什么呢？因为我们8086内部，能够一次性处理、传输、暂时存储的信息的最大长度是16位，所以说所存储的地址也是16位，最大也就是2^16，结果就是64KB.***
>
> ***但是8086CPU有20位地址总线，可以传送20位地址，所以为了合理的利用资源，8086CPU采用一种在内部用两个16位地址合成的方法来形成一个20位的物理地址。***
>
> ***就算是这样的话，我们偏移地址仍然只有16位，所以其寻址能力仍旧是64KB，所以一个段的长度最大为64KB。***

将数据、代码、栈放入不同的段中，无非就是多定义几个`segment`然后将所要定义的内容存放在相应的段中即可。

示例：

```
assume cs:code,ds:data,ss:stack

data segment
    dw 0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h
data ends

stack segment
    dw 16 dup(0)
stack ends

code segment
start:
    mov ax,stack;会被编译器处理为表示一个段地址的数值
    mov ss,ax
    mov sp,20h;设置段寄存器的值，设置栈指针寄存器的值

    mov ax,data
    mov ds,ax

    mov bx,0;ds:bx指向data段的第一个单元
    mov cx,8;设置计数器的值
s:
    push [bx]
    add bx,2
    loop s

```

```
    mov bx,0
    mov cx,8
s0:
    pop [bx]
    add bx,2
    loop s0

    mov ax,4c00h
    int 21h
code ends
end start;指明程序的入口在start处
```

注意注意！！！

CPU是一开始就知道了data、stack等各种代表着特殊含义的段了吗？

当然不是！！

源程序对于这三个段：`code segment` `stack segment` `data segment` 所作的处理：

1）我们将各个段起一个名字，目的在于使得程序便于阅读，CPU并不知道他们的存在；

2）在源程序中使用伪指令`assume cs:code,ds:data,ss:stack`将cs、ds、ss分别与code、data、stack段相连，这样做的目的仅仅是将具有一定用途的段和相关的寄存器相联系起来，CPU仍然不知道这些东西；

3）***我们在程序的最后，`end start`说明了程序的入口，这个入口将被写入可执行文件的描述信息，可执行文件中的程序被加载入内存中后，CPU中的CS:IP将设置指向这个程序入口，(注意，就是在这个时候，CS寄存器的值已经设置完毕)***

从而开始执行相关的指令，标号start在code段中，这样的话，CPU就会将code段中的内容当作指令来处理。

在执行指令的过程中，我们设置了段寄存器的内容，设置了栈寄存器的内容，设置了数据段寄存起的内容，这个时候，CPU才认得我们的各个段。具体CPU如何处理我们定义的段的内容，是被当做指令执行，还是当作数据访问，还是当作栈的空间，取决于具体的汇编指令。

我们完全可以将数据段中的内容当作栈来使用，完全可以将栈中的内容当作数据来使用，取决于汇编指令。

检测小程序1：

将a段和b段中的数据分别相加，然后存储在c段中：

```
assume cs:code

a segment
    db 1,2,3,4,5,6,7,8
a ends

b segment
    db 1,2,3,4,5,6,7,8
b ends

c segment
    db 0,0,0,0,0,0,0,0
c ends
```

```
14
15  code segment
16  start:
17      mov ax,a
18      mov ds,ax
19      mov ax,c
20      mov es,ax;附加数据段寄存器
21      mov bx,0;基址寄存器，可以用作存储器指针来使用，必须使用bx寄存器当作存储器指针来使用，
        使用别的寄存器会报错，使用寄存器间接寻址的时候，只可以使用
        ;bx,bp,si,di这几个寄存器，使用变址（SI，DI）或者基址（BX,BP）
22      mov cx,8
23  s1:
24      mov al,[bx]
25      mov es:[bx],al
26      inc bx
27      loop s1
28      mov ax,b
29      mov ds,ax
30      mov bx,0
31      mov cx,8
32  s2:
33      mov al,[bx]
34      add es:[bx],al
35      inc bx
36      loop s2
37
38      mov ax,4c00h
39      int 21h
40
41  code ends
42  end start
```

检测程序2：

将a段的前8个字型数据存放到b段中：

```
1   assume cs:code
2
3   a segment
4       dw 1,2,3,4,5,6,7,8,9,0ah,0bh,0ch,0dh,0eh,0fh,0ffh
5   a ends
6
7   b segment
8       dw 8 dup(0)
9   b ends
10
11  code segment
12  start:
13      mov ax,a
14      mov ds,ax
15      mov bx,0
16      mov ax,b
17      mov es,ax
18  s:
19      push ds:[bx]
```

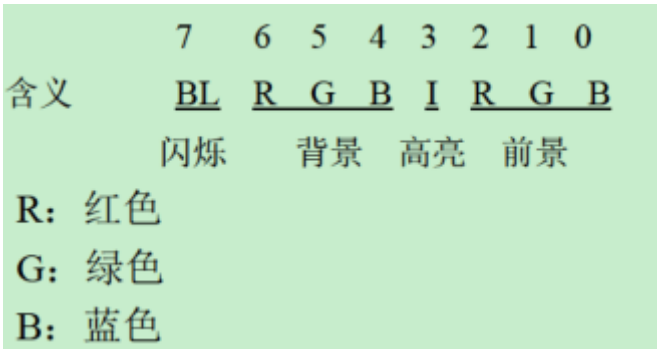
```
    pop es:[bx]
    add bx,2
    cmp bx,7
    je s
    mov ax,4c00h
    int 21h

code ends
end start
```

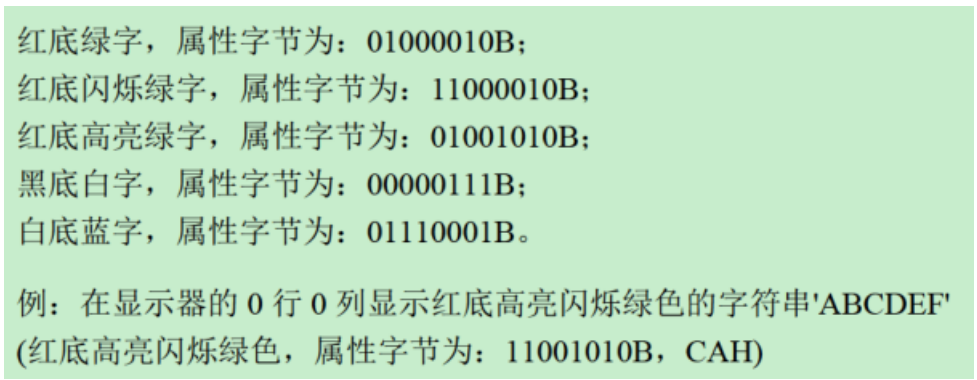
十.利用显存直接向屏幕输出内容

80x25彩色字符模式显示缓冲区的结构，也就是显存的结构：

- 内存地址空间中，B8000H~BFFFFH共32KB的空间，为80x25彩色字符模式的显示缓冲区。我们如果向这个地址空间直接写入数据的话，写入的内容就会立即显示在显示器上。
- 显示器可以显示25行，每一行80个字符，每一个字符可以有256中属性（背景色、前景色、闪烁、高亮等组合信息），一个字符占据两个字节的存储空间，低位字节存储存储字符的ASCII码，高位字节存储字符的属性。**注意如果再写入显存的时候没有指定字符的显示模式的话，是不会输出字符到显示器的。**
- 80x25模式下，一屏幕的内容在缓冲区中占据4000个字节。显示缓冲区分为8页，每页4KB（≈4000B，4000字节），显示器可以显示任意一页的内容。一般情况下，显示第0页的内容，也就是说通常情况下，B8000H~B8F9FH中的4000个字节的内容将会出现在显示器中；
- 一个在屏幕中显示的字符，具有前景（字符色），背景色，闪烁，高亮等信息：
- 属性字节的格式：



示例：



示例：向显存的第一页的中间部分输出`hello,word!`：

```
data    segment
    str db "hello,world!"
data    ends

```

```
assume  cs:code,ds:data

code segment
start:
    mov     ax,0b800h
    mov     es,ax
    mov     di,12*160+36*2
    mov     ax,data
    mov     ds,ax
    lea     si,str
    mov     cx,11

loop1:
    mov     al,[si]
    mov     ah,0cah
    mov     es:[di],ax
    inc     si
    inc     di
    inc     di
    loop    loop1

exit:
    mov     ax,4c00h
    int     21h

code ends

end start
```