微型计算机原理及应用技术

计算机基础知识

码制

原码 反码 补码

原码

最高位为符号位，正数为0，负数为1，其余为为数值位

反码

- 正数：与原码相同
- 负数：符号位不变，其余为取反

> 8位反码中 +0 = 0000 0000 -0 = 1111 1111
> 表示范围 -127 --- + 127

补码

> 补码的作用是用加法代替减法

- 正数：与原码相同
- 负数：反码加一

> 8为补码中-0 = -0 = 0000 0000
> 符号位为1时，该数的绝对值为其余位取反加1，或减1取反
> 表示范围 -128 --- +127 -128 = 1111 1111

省流版本

- 正数：正反补码相同
- 负数：反码取反，补码加1

十进制	二进制	原码	反码	补码
+126	+111 1110	0111 1110	0111 1110	0111 1110
-126	-111 1110	1111 1110	1000 0001	1000 0010

定点与浮点

定点小数法

符号位	小数点(隐含)	数值部分(尾数)

定点整数法

符号位	数值部分(尾数)	小数点(隐含)

> 上述数据在进行运算时，可能需要选择适当比例因子

浮点表示法

阶符	阶码	数符	尾数
0	0011	0	1011

$$N = 2^{+11} \times 0.1011$$

在浮点数表示中，有一项规则被称为规格化数，即尾数最高位一定是1

> 为什么?
> 最高位是1能够保证在一定尾数下储存更多的数，同时也不能提高精度

对于部分汇编的补充

寄存器：CPU内部暂存数据

常用寄存器

16位	高8位	低8位
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

> 在8086中，寄存器都是16位的，但**有且仅有以上四个**寄存器允许拆出8位寄存器

指令助记符	目的操作数(DST)	源操作数(SRC)
mov	AL	97
add	AL	89

在add中，是目的操作数加源操作数，求得和放回到目前操作数

> SBU是减法，那么就是目的减源 ，结果放回目的
> 在汇编中，数字必须以0-9开头，变量必须以字母开头
> 值得注意的是，字节运算是8位；而字的运算是16位
> 目的操作数不能有直接数据出现

处理器状态字寄存器(PSW)/标志寄存器(FR)

有时候，需要根据运算结果来判断下一步操作
例如上述97+89例子会产生溢出，这就会产生一个状态
CPU会根据结果产生的状态，自动在FR中标志位置0或1
只有做了**运算**(如add可以,但mov就不行)，**才**会引起标志位变化

> 0110 0001 + 0101 1001 = 1011 1010
> 不要忘记8位表示，最高位是符号位而不是表示一个数

例如在FR中第六位的ZF(书P23)，若AL为0,ZF=1,否则ZF=0
在上述例子中，结果溢出了，那么FR中第11位OF(溢出标志位)=1
这些状态可以用作分支程序条件设计中条件

溢出的判断

1. 符号位与数据位最高位之间不存在进/借位情况，不溢出
2. 符号位与数据位最高位之间都存在进/借位情况(借两次)，不溢出
3. 一个进/借位一个不进/借位，溢出

逻辑运算

指令助记符	逻辑运算	用法
AND	与	AND DST,SRC
OR	或	OR DST,SRC
XOR	异或	XOR DST,SRC
NOT	非	NOT DST

> 逻辑运算也算运算，会影响FR的值；但是NOT运算不影响FR值

例如：
MOV AL,97
AND AL,89

什么时候用到与运算？
把某一操作数部分位清零或不变。清零与0，不变与1

AND AX,0FFF0H

> 把AX中后四位置0，其他不变。其中最后的H表示他是16位数，第一0表示他是数字而不是变量

什么时候用到或运算？
把某一操作数部分位置1，其他位不变。置1或1，不变或0

AND BX,000FH

> 把BX中后四位置1，其他不变

什么时候用到异或运算？
把某一操作数部分位取反，其他位不变。取反异或1，不变异或0

XOR AL,0F0H

> 把AL中后四位不变，其他取反

一般微处理器结构

外部结构简述

外部结构表现为数量有限的输入输出引脚，这些引脚构成了微处理器及总线
传输信息一种有三种：

- 数据信息 — 数据总线DB
- 地址信息 — 地址总线AB
- 控制信息 — 控制总线CB

总线：用于信息传输的通道
8086/8088采用双列直插式封装，共40个引脚

地址线简述

有20根地址线
$A_{15}-A_0$
$A_{19}-A_{16}$
这二十根地址线全部用于给外部存储器提供地址，所以可以寻址1MB
第16条地址线，同时也给外部I/O提供地址，可以寻址64KB

> 我他妈怎么知道8086给的地址是给存储器还是I/O?
> 通过$M/\overline{IO}$引脚知道

I/O端口

I/O端口就是寄存器，用来实现CPU与外设之间信息交换

> 所有的设备都不能之间与 CPU进行数据交换，必须要设计一个**接口电路**

简述例子

例如CPU想要给打印机传输信息，二者之间需要设计接口电路
什么时候CPU将打印信息给打印机，我怎么知道打印机准备好了?
打印机会把自己状态给到接口电路中状态输入端口，让CPU通过分配好的地址，来读该端口。

> 注意:只有CPU在读取某个设备时,对应端口才为二态(0/1),否则就是三态(高阻态).因为数据总线上会挂载多个设备,不能影响其他读写操作.

怎么输出数据?
用另一个I/O端口,分配新的端口地址,这个端口称为数据输出端口,CPU把数据给打印机

> 这个数据输出端口需要有锁存功能,锁存后是**有效数据**

打印机怎么知道数据有效呢?
设计命令输出端口,打印机读取该端口

> 以上所有端口均在I/O接口电路中

简述I/O端口

一个I/O端口,就是传输一种信息通道,这个端口是用寄存器设计的
一个I/O端口,至少占用一个I/O地址,称为I/O端口地址,简称**端口地址**(在64KB端口分配)

编址

统一编址

一个地址给I/O不能给存储器,给存储器不能给I/O

> 这时候I/O=存储器

缺点:浪费存储器地址空间
优点:程序设计简单,对存储器的指令也适用于I/O

独立编址

存储器地址空间与I/O地址空间独立,地址可以有重复的

> 8086CPU是这种,所以才需要有$M/\overline{IO}$引脚

优点:节约存储器地址空间
缺点:程序设计复杂,对存储器和端口操作的指令是两钟

独立编址下存储器与I/O寻址

存储器寻址

将10写到2000H地址单元
MOV AL 10
MOV [2000H] AL

> 直接寻址

MOV AL, 10
MOV BX, 2000
MOV [BX], AL

> 间接寻址,有中括号的就是地址,否则就是数据

I/O寻址

把01H写道地址为2000H的I/O中
MOV AL, 01H
MOV DX, 2000H
OUT DX, AL

> 给I/O写,是OUT,I/O读,是IN;给存储器写,是MOV,是两个不同指令

注意:I/O间接寻址,对应放地址的寄存器**不打中括号**;而对存储器间接寻址**打中括号**

内部结构

- ALU 进行算数逻辑运算
- 工作寄存器
  - 数据寄存器 AX 只能暂存数据
  - 地址寄存器 BX 不提供地址时可以用于暂存数据
- 控制器
  起中央指挥作用,例如取指令操作,可将指令暂存在指令寄存器,并通过译码分析指令
- I/O控制逻辑电路

控制器

- 程序接收器(PC) 16位
  通过地址总线输出地址
  通过数据总线读取数据,将数据放入指令寄存器
- 指令寄存器(IR) 8位
- 指令译码器(ID) 8位
  检测IR,进行译码分析
- 控制逻辑部件
  在相应控制引脚发出相应命令,由ID控制

堆栈

由先进后出组织的一段存储器区域称为堆栈
对于8086而言,在1MB存储器的空间内划分出一个区域,定为堆栈区

指令助记符	含义
PUSH	入栈
POP	出栈

注意:对应8086而言,堆栈必须按照字操作

> 为什么要堆栈?
> 寄存器可以暂存数据有限,我再想要用寄存器时,可以将寄存器内数据入栈,再需要用时候,出栈

PUSH AX 一个字,可以
PUSH AL 不是一个字,不行

> 16位8086中,一个字=2字节=16位

我怎么知道堆栈堆哪呢?
由堆栈指针寄存器SP来提供,把SP的内容做地址,指向栈堆+1单元的地址

入栈

PUSH AX
PUSH BX
1.PUSH AX
SP = SP - 2 SP减2放回SP,注意,这两次SP指向的地址之间空出了一个字的单元

第二次SP
第一次SP

2. 将AX压入

AL(第二次SP)
AH
第一次SP

注意:AL放在低地址单元,AH放在高地址单元

3. PUSH BX
   重复上述操作

BL(第三次SP)
BH
AL(第二次SP)

AH
第一次SP

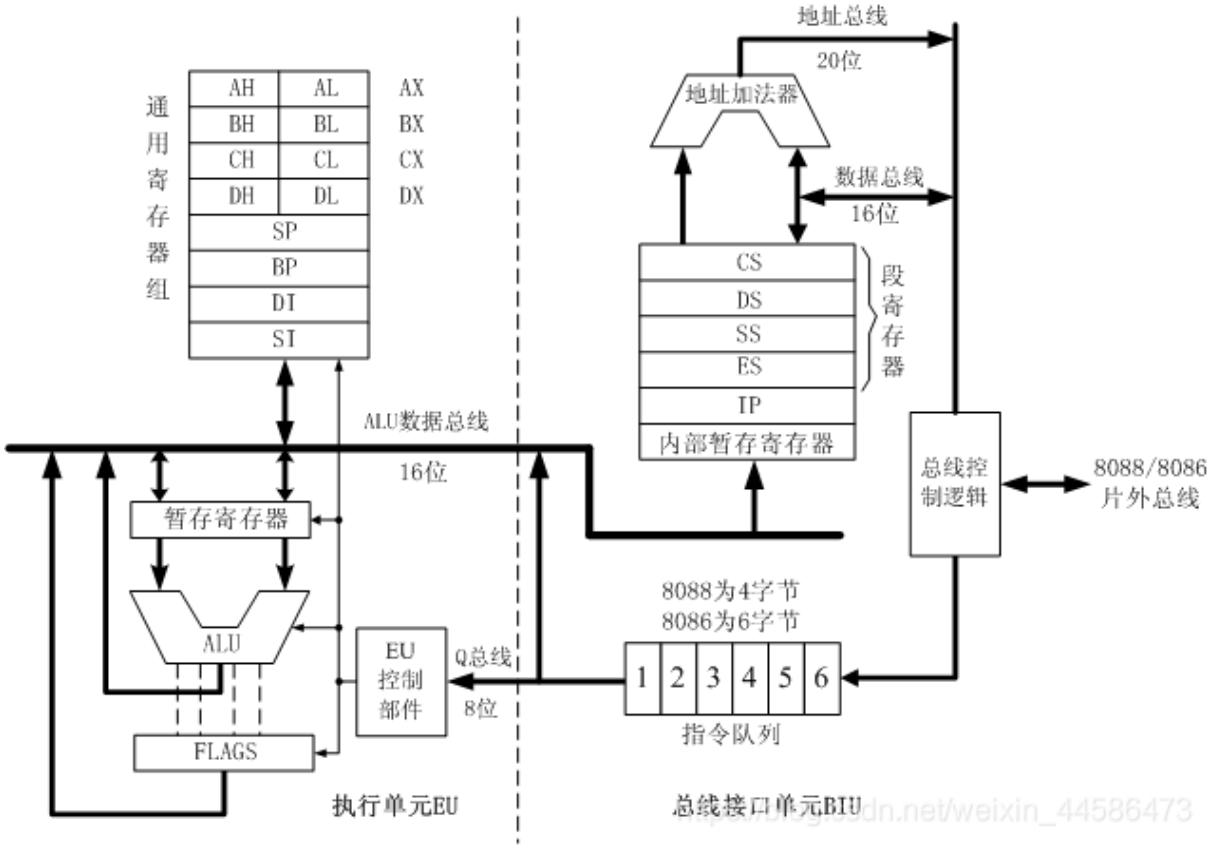
将SP所指单元称为栈顶,**栈顶是活动的,栈堆是固定的**

出栈

POP BX
POP AX

1. POP BX
   将SP内地址作为栈顶地址,从该地址单元读取**一个字**给BX
2. 将SP自动增二给SP SP = SP + 2
3. POP AX 重复上述操作,最后SP指针**回到**上图中第一次SP指向位置

8086内部结构



总线接口单元 BIU

组成结构

寄存器

CS:代码段寄存器
ES:附加数据段寄存器
SS:堆栈段寄存器
DS:数据段寄存器
IP:指令指针寄存器(8086中等效于一般微处理器中PC)

地址产生与总线控制逻辑

上述例子中,都是用16位地址空间(16根线,64KB)来寻址的(如间接寻址中BX和SP),实际上8086由20根线(1MB)可以给存储器寻址,如何解决?
将1MB空间分为多个逻辑段,每一个逻辑段最大64KB,这样就可以用16根线来寻址,也就出现了段地址
CS来提供代码段的段地址(例如:取指令所在单元的段地址由CS提供,取值地址在单元里面哪地址由IP提供)

> 简单来说,地址分为两个部分组成

> 1. 段地址,由段地址寄存器提供(16位)
> 2. 段内地址,相当于偏移量,后面章节会提到(16位)

由于我写程序时候只提供16位地址,现在想要得到20位地址,就应该需要一定运算,这个运算由地址产生于总线控制逻辑来进行运算

指令队列寄存器

类似于一般微处理器中IR(1个字节,8位),在8086中指令队列寄存器是6个字节(48位)的,相当于6个IR,没放满就一直取

> 对应8088而言,指令队列寄存器是4个字节的,其余完全相同

BIU作用

主要负责从外部存储器中取指令(代码段),并将取回指令放回指令队列中

执行单元 EU

主要作用

主要负责从指令队列中获取指令,遵循**先进先出**原则,并对获取的指令进行译码分析(执行该指令)

执行周期

从此我们可以看出在8086中,取指令和执行指令操作的同时进行的,比一般微处理器中取-执行=取=执行效率要高

1. 第一个周期
   BIU取指令,EU空闲
2. 第二个周期
   BIU继续取指令,执行BIU上一个周期获取的指令
3. 第三个周期
   重复2操作,并一致循环下去

什么时候BIU会停止取指令呢?

1. 队列取满
2. EU要使用总线(进行写操作)

BIU	取指令	取指令	帮助EU进行操作
EU	空闲	执行指令	执行指令(要用总线)
BUS	忙	忙	忙

也可以看出,相比于一般微处理器,8086总线利用率很高

组成结构

- 通用寄存器
  编写程序就要用,所以叫通用寄存器
- ALU及标志寄存器FR
- 内部控制逻辑电路

8086中寄存器的组织结构

共有14个16位寄存器

通用寄存器

总共有8个

数据寄存器

4个,均为16位,都可分为高8低8位两个寄存器来使用,有且仅有这8个8位寄存器
都可用于暂存数据,每一个都有特殊功能
AX:累加器,作为目的操作数
BX:基址寄存器,对某程序单元的进行读写操作,段内16位基本地址由BX提供
CX:计数寄存器,循环时每循环一次**自动**减1,有指令相互配合
DX:数据寄存器,操作I/O时可做端口地址寄存器,不用中括号

地址指针与变址寄存器

均为16位

地址指针寄存器

SP:堆栈指针寄存器,提供堆栈操作中堆栈段**段内地址的偏移量**

> 注意:SS提供的是段地址

BP:基址指针寄存器,提供**段内地址偏移量**

> 注意:BP所提供16位地址和BX提供16位地址不在同一个段内,BX默认在数据段,BP默认在堆栈段

MOV BX, 002H
MOV BP, 002H
MOV [BX], 34H 默认写到数据段(DS)
MOV [BP], 34H 默认写到堆栈段(SS)
若是想要让BP写到数据段:
MOV DS:[BP], 34H 强制写到数据段

> 这里的DS称为段超越前缀/段前缀,汇编时候会多一个前缀码

变址寄存器

均为地址寄存器
SI:源变址寄存器
DI:目的变址寄存器

> 注意:这两个默认找的都素数据段(DS)单元

SI和DI在字符串操作中(MOVS命令),指定好后运行会**自动**的变(加减),进行字符串操作,所以叫变址

段寄存器

段寄存器存的是**段地址**
CS:代码段寄存器
ES:附加数据段寄存器
SS:堆栈段寄存器
DS:数据段寄存器

> 注意:段寄存器操作系统会自己初始化

数据存放格式

字节型

> 伪指令:告诉汇编怎么编译的

DB 12H,12,-12
字节型数据伪指令，连续三个单元分别放12H,0CH,F4H

> 这里是将12，-12转为补码又换位16进制

字型

DW 5678H
低单元地址称字的地址，由于字占两个字节，8086里面都是以字节存储的，字的高位放到高地址单元，低位放到低地址单元

字单元地址为偶地址称对准的字单元，奇地址称未对准字单元

> 8086CPU对对准字单元速度比未对准的快一倍

双字单元

DD 783H

存储器分段与物理地址形成

存储器为什么要分段?
64K寻不了1M的

存储器分段

每个逻辑段，最大64K字节，每个逻辑段起始地址必须能被16整除

物理地址PA

物理地址是该单元实际存在地址
逻辑段的起始地址要求能被16整除，一个逻辑段最大64K
例如:
0段起始地址00000H
1段起始地址00010H
2段起始地址00020H

> 请注意，一个逻辑段最大可以划分64K字节，这里0段和1段之间间隔16字节，也就是说他俩之间存在重叠空间。同时对应1M空间可以划分出64K个逻辑段

逻辑地址

段地址	段内偏移地址(段内有效地址EA)	逻辑地址
0000H	0000H	0000H:0000H
0000H	0001H	0000H:0001H
0002H	0000H	0002H:0000H
0000H	0020H	0000H:0020H
0001H	0010H	0001H:0010H

注意：表格后三个虽然表示不同，但都是同一个物理地址00020H

物理地址形成

$$物理地址 = 段地址 \times 16 + 段内偏移地址$$

$段地址 \times 16$相当于段地址左移一位

8086指令系统补充内容

标号

如：next:
标号一旦定义具有三个属性

1. 段地址属性
2. 段内偏移地址属性
3. 类型属性

- NEAR 转译指令与专业标号在同一代码段，赋值-1
- FAR 不在同一代码段，赋值-2

常数及表达式

常数

16进制常数：以H结尾
10进制常数：以D结尾，可不写
2进制常数：以B结尾
字符常数：单引号包含，如‘A’
字符串常数：双引号包含，如“Hello”
随机：？表示随机数，开机随机赋值

表达式

算数表达式

mov 5+2*3

> 汇编出表达式的值，不是CPU计算的

逻辑表达式

AND AL,21H AND OFH
后面的是逻辑表达式，也算汇编计算，不是CPU计算

关系表达式

MOV AX,5 LT 3

> LT是小于,假取全0，真取全1

属性表达式

获取标号段地址属性(16位)：SEG
获取标号段内偏移地址属性(16位)：OFFSET
获取标号类型属性：TYPE

> MOV BX,OFFSET NEXT
> 获取NEXT标号的偏移地址放到BX

变量及定义变量伪指令

DB 定义字节型变量，8位，1个单元
DW 定义字型变量，16位，2个单元
DD 定义双字型变量，32位，4个单元

> DATE1 DB
> 在数据段定义DATE1变量，DATE1就相当于地址

变量一旦定义了，就具有五个属性

1. 该变量段地址属性 SEG
2. 该变量段内偏移地址属性 OFFSET
3. 该变量类型属性(字节型=1，字型=2) TYPE
4. 长度属性 LENGHT
5. 大小属性 SIZE

> MOV AL, TYPE DATE1 = MOVE AL,1

> $ 是汇编程序中表示当前汇编到的存储器的地址
> DW DATA2
> 可以用如MOV DATA2 $

用DB定义字符串时可定义无限长度的字符串，但是DW只能定义两个字符

> DATA2 DB 'hello'

DATA3 DW DATA2
这时候放的就是DATA2的OFFSET
DATA4 DD DATA2
这时候放的就是DATA2的OFFSET和SEG(偏移地址和段地址都放进去了)

DATA5 DB 4 DUP(?)

> DOP为重复操作符,随机赋值重复四次(连着4个单元都是随机数)

DATA5 DW 3 DUP(?)

> 随机赋值重复三次(连着6个单元都是随机数)

长度属性

在变量名定义语句中所定义的变量个数，叫长度属性(LENGTH)

> 出现了DUP,那么该变量长度属性就是重复的次数
> DATA2 没有出现DUP，长度就是1
> DATA5 为4 DATA4为3

大小属性

在变量名定义语句中，所定义的所有变量所占的总的字节数(单元数),叫大小属性
SIZE = TYPE * LENGTH

> DATA5 大小是4

DATA1 DB 01H
MOV AX, WORD PTR DATA1

> 强制转换为字类型赋值给AX

同理
MOV [BX], 10H
也会因为类型不明确而报错，不知道BX间接寻址应该放字节还是字
MOV BYTE PTR [BX], 10H
明确BX间接寻址是一个字节单元

堆栈的注意事项

1. PUSH和POP只允许按字访问(16位)
2. 低字节放在偶地址，高字节放在低地址。**SP始终指向偶字节单元**
3. SP大小从FFFEH到0
4. PUSH后面不能接立即数，也不能POP CS因位CS不能作为目的操作数
5. POP是加，PUSH是减

数据传输指令简述

1. MOV
2. PUSH 与 POP
3. XCHG 数据交换指令
   XCHG AL,BL
   交换AL,BL的值
   > 注意:段地址寄存器不能作为操作数(目的和源都不行)，两个存储单元也不能直接交换数据(不能XCHG [0001H],[0006H])

4. XLAT 换码指令

   将BX与AL内容相加作为EA，根据该偏移量找到对应内容送到AL中

5. LEA 将源操作数的有效地址EA送到目的操作数

   > 注意:这里送的是偏移量，不是偏移量对应的值
   > LEA DI,TABLE 本质上等同于 MOV DI,OFFSET TABLE

6. LDS 见P82
7. LES 见P82
8. 标志寄存器传送指令 见P82

8086系统数据寻址方式

寻址:求得操作数所在地或所在存储器单元地址的方式
求得的操作数有如下两种用途:

1. 当作数据使用
2. 用作转移地址使用

> 注意:在本章节只介绍当作数据使用,下一章节介绍转移地址使用

寻址的操作数只能有以下四种:

1. 立即数
2. 通用寄存器
3. 段地址寄存器
4. 存储单元

立即寻址

MOV AX,080AH
将080H放到AX中
后面的080AH是**立即数**，所以叫立即寻址
注意事项:

1. 立即数只能是源操作数

寄存器寻址

MOV AX,BX
把BX数给AX
注意事项:

1. 类型要一致
   > MOV AL,BX 错误
2. 类型要明确
   > MOV [2000H], 56H 错误
   > MOV WORD PTR [2000H], 56H 正确
3. 二者**不能都是段地址寄存器**
4. CS和IP不能做目的操作数,但是可以做源操作数
5. DS,ES,SS做目的操作数,源操作数不能为立即数

存储器寻址

要找的操作数放在存储器单元中,存放操作数单元的EA由以下5种方式寻址:

直接寻址

操作数所在单元EA在指令中直接给出

MOV AX,[22A0H]
操作数的有效地址称为EA，也可理解为偏移量，在这里EA就是22A0H
这里EA是偏移地址，那么需要换算成实际地址(DS*16+22A0H)才能存入，默认段地址是DS

> 若是想强制改到别的代码段，可以参考前文的超越前缀

注意事项:

1. 存储单元可与通用寄存器互传,但是存储单元间**不能直接操作**(MOV,ADD,AND均不可)
   > MOV [2000H], [2002H] 错误
   > DAT1 DB 12H
   > DAT2 DB 56H
   > MOV DAT1, DAT2 错误

寄存器间接寻址

EA是有且仅由**地址寄存器**提供的 **(BX,SI,DI)**
特别注意:以上三个寄存器默认段地址都是DS
MOV AX,[SI]
把SI对应地址内的数给AX，注意这里也算偏移地址，要换算成实际地址

> MOV BX,OFFSET DAT1
> MOV AL,[BX]
> MOV [BX], 56H 错误,类型不明确,因为[BX]表示一个存储器单元,没有类型,56H是一个立即数,也没有类型

> 注意:在本册教材中,BP也算在间接寻址当中,即EA可以由BX,BP,SI,DI提供,但是西安电子课件大学视频中却强调BP不归属间接寻址当中

寄存器相对寻址

EA = BX/BP/SI/DI + 8位/16位偏移量(COUNT)

> 以下二者完全一致,区别就是基址寻址用BX/BP,变址寻址用SI/DI

再强调一遍:
BX,SI,DI默认DS
BP默认SS

MOV [BX]+3, AL
实际上完全等同于

1. [BX + 3]
2. 3 + [BX]
3. 3[BX]

若是CONT不写,如[BX],那么默认是0

基址寻址

EA = BX/BP + 8位/16位偏移量(COUNT)

MOV AX,COUNT[BP]
实际地址=SS*16+COUNT+BP
BP默认段地址是SS

变地址寻址

EA = SI/DI + 8位/16位偏移量(COUNT)

MOV AX,COUNT[SI]
实际地址=DS*16+COUNT+SI

变量作为COUNT

在上述例子中COUNT是一个常数3,但是这个COUNT可以是一个变量
MOV BX,0
MOV AL,OOH
MOV DAT1[BX],AL
那么他的实际地址应该表示为:
DS:OFFSET DAT1 + BX

> 这里代码段是DS,是因为DAT1存放在DS段中,与变量存放位置有关

段地址确认,变量的优先限权高于寄存器,即**用变量来确认**

例题

PUSH AX
PUSH BX
PUSH CX
要求在不破坏SP前提下,取出AX放入DX
MOV BP,SP
MOV DX,[BP]+4

基址变址寻址

EA = BX/BP + SI/DI
段地址的确认,由**基址寄存器来确认**

MOV [BX][SI],AL等同于[BX + SI]

基址变址且相对寻址

EA = BX/BP + SI/DI + 8位/16位偏移量
注意:偏移量要是变量,段地址由变量的段地址来确定,否则就由基址寄存器的段地址来确定

MOV DAT1[BX][SI],AL等同于[BX + SI + OFFSET DAT1]

字符串寻址

MOVSB

I/O端口寻址

IN AL,20H
把外设地址为20H的内容读入AL
OUT DX,AL
将AL内容输出给以DX的内容为地址外设中

> 注意，这里都不需要打括号

隐含寻址

AAA

PUSH AX 隐藏了目的操作数

> 实际上MOVSB,MOVSW这种字符串寻址也是隐含寻址,且源和目的操作数都是隐含指令,但是本书分开来写

注意事项精简版

1. **只有**BX,BP,SI,DI这四个寄存器可以出现在括号内
2. BX,BP不允许出现在同一个括号内
3. SI,DI不允许出现在同一个括号内
4. 源操作数的要求如上所示，**目的操作数均用寄存器表示**
5. CS段寄存器与立即数不能作为目的操作数
6. 立即数送给内存时要与内存变量保存一致

7. 源操作数与目的操作数不能同时为存储器操作数，也不能同时为段地址寄存器
8. 立即数不能直接给段地址寄存器
9. IP寄存器不能作为源或目的操作数
10. MOV指令不改变标志位
11. 变量不能在指令中运算,如MOV AL,DAT1 + DAT2错误

8086系统关于转移地址寻址

段内转移

只有IP发生变化

段内直接寻址(段内相对寻址)

转移指令中**直接**给出转移地址(标号)

JMP L1

此时若是不跳转，执行完上述语句后，IP指向的值在该指令下一个单元，我们称此时IP为当前IP

JMP L1 汇编成机器码后为 操作码+偏移量(1或2字节)

> 注意:L1给的是地址，但是汇编后却是相对位移量；负向转移若是转移字节数<128，则占一个字节，正向转移或者负向转移>128都是2个字节

同时，若是你很确定转移不会超过128，那么可以用
JMP SHORT L1 这样偏移量就是1个字节(无论正向还是负向)

$$转移到目的IP值 = 当前IP值 + 偏移量$$

注意:8086指令系统中，所有的**条件转移指令**只能在**段内转移**，且转移范围-128至+127，因此其寻址方式是**段直接(相对)对寻址**

> 若是超出，只能用无条件转移指令(JMP)搭桥

段内间接寻址

转移到的地址间接放着，不直接给出
MOV BX,OFFSET L2
JMP BX

段间转移

段间直接寻址

例如:另一代码段第一条指令的标号是ABC
JMP FAR PRT ABC
那么以上指令被汇编为:
操作码(8位) + ABC的OFFSET(16位) + ABC的SEG(16位)

> 指令中直接给出转移的地址，故叫段间直接寻址

段间间接寻址

例:在前面基础上，我在DS段定义
TABLE DD ABC

> 在头两个字节单元放OFFSET，后两个放SEG

JMP TABLE

> 注意，这里TABLE一定要是双字单元(32位)

LEA BX, TABLE
JMP DWORD PTR TABLE [BX]

> DWORD相当于告诉指令要找双字单元

数据传输指令

本章节中，出SAHF,POPF外，其余指令对标志位均不产生影响

通用数据传输指令

MOV DST,SRC

取有效地址指令

LEA REG16, MEM

> 此REG16哪个寄存器都可以，但为了方便一班选BX,SI,BP,DI
> MEM的寻址可由五种寻址方式指明

取地址指针指令

LDS
LES

标志位传递指令

LAHF 取标志寄存器低8(0-7)位给AH
SAHF 将AH给标志寄存器低8位

数据交换指令

XCHG
注意:段地址寄存器不能交换

字节转化指令

XLAT

$$AL = (DS : (BX + AL))$$

即将BX+AL至作为EA,在DS段找到给AL

堆栈操作指令

PUSH SRC
POP DST

> 注意:以上两个指令的操作数不可是**立即数**,且为16位

PUSHF 标志寄存器压栈
POPF 弹出给标志寄存器

算数运算指令

- CPU只要进行运算，标志寄存器就会受到影响
- 段地址寄存器不参与运算

加法指令

1. ADD DST,SRC
   不带进位加法，DST = (DST) + (SRC)同时设置状态标志
2. ADC DST,SRC
   带进位加法， DST = (DST) + (SRC) + CF(进位标志位)同时设置状态标志
3. INC 增1指令
   INC DST
   DST = (DST) + 1同时设置状态标志(**除CF外,对CF没影响**)

减法指令

1. SUB DST,SRC
   不带借位减法，DST = (DST) - (SRC)同时设置状态标志
2. SBB
   带借位的减法， DST = (DST) - (SRC) - CF(进位标志位)同时设置状态标志
3. 减1指令
   DEC DST
   DST = (DST) - 1同时设置状态标志(**除CF外,对CF没影响**)
4. CMP 比较指令
   CMP DST,SRC
   (DST) - (SRC)，不影响源和目的操作数，**只是根据结果设置状态标志**
   主要是用于产生条件的(AF不算条件，但CMP会影响AF值)
5. NEG 求负指令
   NEG DST
   DST = 0 - (DST)同时设置状态标志

插入:条件转移指令

判断标志位然后转移的指令有:

> N表示Negative,J表示JUMP

- CF
  JC 标号

JNC 标号

- SF
  JS 标号
  JNS 标号
- ZF
  JZ或JE 标号
  JNZ或JNE 标号
- PF
  JP 标号
  JNP 标号
- OF
  JO 标号
  JNO 标号

α, β分别是无符号数，那么两数之间有如下关系:

1. 相等 E (不等是NE)
2. 高 A
3. 低 B

看CF标志位:
JB/JNAE 标号
JBE/JNA 标号

JA/JNBE 标号
JAE/JNB 标号
α, β分别是有符号数，那么两数之间有如下关系:

1. 相等 E (不等是NE)
2. 大 G
3. 小 L

SF与OF异或，为1表明目的操作数小于源操作数:
JL/JNGE 标号
JLE/JNG 标号

JG/JNLE 标号
JGE/JNL 标号

乘法指令

注意事项:

1. **目的操作数是隐含的**，隐含是被操作数
2. 目的操作数**不能是立即数**，可以说存储器或寄存器

3. 加法减法指令不分带符号数和无符号数，但是乘除法指令区分有无符号
4. 字节乘与字乘与源操作数为准
5. 字节乘:AL
   积:AX
   字乘:AX
   积:DX:AX

无符号数乘法

MUL SRC
指令执行后，只影响CF,OF，其余状态标志**没定义**(状态不确定，随机的)
$CF = OF = 0$
字节乘:说明AH高8位无效(都是0),只用管第八位就可
字乘:说明DX无效(都是0),只用管AX就可
$CF = OF = 1$
有有效积

有符号数乘法

IMUL SRC
$CF = OF = 0$
字节乘:说明AH高8位无效，AH的值是**AL符号位的扩展**

> AL最高位是符号位，AL最高位为0，AL全是0，为正；AL最高位为1，AL全是1，为负

字乘:说明DX无效，DX的值是**AX符号位的扩展**
$CF = OF = 1$
有有效积

除法指令

1. **目的操作数是隐含的**，是被除数(前面的)
2. 目的操作数**不能是立即数**，可以说存储器或寄存器
3. 字节除:AX/8位，商在AL，余数在AH
   字除:DX:AX/16位，商在AX，余数DX
4. 状态标志**均没定义**

无符号数除法

DIV SRC

带符号数除法

IDIV SRC
运算符号:

1. 被除数为正，除数为正，商为正，余数为正
2. 被除数为正，除数为负，商为正，余数为负
3. 被除数为负，除数为正，商为负，余数为负
4. 被除数为负，除数为负，商为正，余数为负

符号扩展指令

CBW (AL)扩展到(AX)中，AX内容是AL符号位扩展
CWD (AX)扩展到(DX)中，DX内容是AX符号位扩展
二者均**不影响状态标志**

BCD数调整指令、

BCD数分为:

1. 分离BCD数
   8位表示一个十进制位，实际上只是用低4位对应一个十进制位，高4位任意
2. 组合BCD数
   放一起了，每4位对应一个十进制位

对于加减法而言，二者都可用，但乘除法只能用分离BCD数
对于加减乘，都是先运算，再调整；除法是先调整，后运算

加法BCD调整

组合:DAA
分离:AAA
二者都是对**AL**调整

减法BCD调整

组合:DAS
分离:AAS
二者都是对**AL**调整

乘法BCD调整

AAM

乘法BCD调整

AAD

以上二者都是对**AX**调整

2进制数转10进制(BCD码)算法

除10取余算法

> 除1次，余数为10进制数个位，商再除1次，余数为是10进制数十位，以此类推

BUFF1 放2进制数，BUFF2放转后的10进制(BCD码)
BUFF1 DB ?
BUFF2 DB DUP(?)

MOV AL, BUFF1
MOV AH, 0 (被除数是AX，要清零保证AX就是AL)
MOV CL, 10
DIV CL
MOV BUFF2, AH
MOV AH, 0
DIV CL
MOV BUFF2 + 1, AH
MOV BUFF2 + 2, AL

移位类指令

1. 段寄存器不能参加移位
2. 当CNT=1时，移位次数可直接写出
   当CNT>1时，移位次数由**CL给出**
3. 操作类型只有**字或字节型**(8或16位)

移位指令

正常设置除AF外5个状态标志，AF未定义

> 注意:这里OF只要最高位发生变化OF就会置1

逻辑移位

被操作数看成无符号数

左移

SHL DST,CNT
左移CNT次

> 实际上左移1次，相当于×2;2次，×4;3次，×8

右移

SHR DST,CNT

> 以上两条指令，移出的位给CF，空缺的位补0

算数移位

被操作数看成带符号数

左移

SAL DST,CNT
算数左移和逻辑左移是一样的

右移

SAR DST,CNT
最高位不变(符号不变)，然后最高位往后移

循环移位指令

不带CF循环移位

ROL DST,CNT 循环左移
ROR DST,CNT 循环右移

带CF循环移位

RCL DST,CNT 循环左移
RCR DST,CNT 循环右移

以上四条只影响CF，OF，其余状态标志未定义

循环控制指令

LOOP

LOOP 标号

1. CX = CX - 1 (不影响标志位)
2. 若CX减1后值不为0，循环到标号对应目的地

LOOPZ(LOOPE)

LOOPZ(LOOPE) 标号

1. CX = CX - 1 (不影响标志位)
2. 若CX减1后值不等于0，**且ZF=1**，则循环到标号对应的目的地

> 注意:这条指令本身不影响ZF，ZF改变是前面运算改变的

适合于在指定区域中查找不同的字符，找到自动退出(结合CMP指令与INC指令)

> 插入:求字符串长度
> STR DB "114514 1919810"
> COUNT EQU $ - STR (EQU相当于把后面的表达式的值给COUNT)

LOOPNZ(LOOPNE)

LOOPNZ(LOOPNE) 标号

1. CX = CX - 1 (不影响标志位)
2. 若CX减1后值不等于0，**且ZF=0**，则循环到标号对应的目的地

> 注意:这条指令本身不影响ZF，ZF改变是前面运算改变的

适合于在指定区域中查找相同的字符，找到自动退出(结合CMP指令与INC指令)

JCXZ

JXCZ 标号
若CX=0，则循环到标号对应的目的地

逻辑运算指令

逻辑运算指令也算运算类指令，那么就必须有以下运算类指令要求:
如**段寄存器**不能参加运算

与

ADD DST,SRC

或

OR DST,SRC

异或

XOR DST,SRC

> 在执行以上三个运算后(与或非)，CF、OF自动清零;AF未定义;正常设置其他标志

非

NOT DST,SRC
非运算对6个状态标志无影响

测试

TEST DST,SRC
位对位与，但根据结果设置标志位，并**不破坏**源或目的操作数

逻辑运算应用

部分置0

什么时候用到与运算?
把某一操作数部分位清零或不变
清零与0，不变与1

AND AX,0FFF0H

> 把AX中后四位置0，其他不变

部分置1

什么时候用到或运算?
把某一操作数部分位置1，其他位不变
置1或1，不变或0

AND BX,000FH

> 把BX中后四位置1，其他不变

部分取反

什么时候用到异或运算?
把某一操作数部分位取反，其他位不变
取反异或1，不变异或0

XOR AL,0F0H

> 把AL中后四位不变，其他取反

大小写转换

大写字母和其小写字母的二进制位只有在**第5位**不同
即大写字母的第五位为0，小写字母的第五位为1

大写转小写

AND AL, 11011111B;DFH

小写转大写

OR AL, 00100000B;20H

输入输出指令

输入指令

IN DST,SRC
源操作数是一个端口地址

- 若分配端口地址在0~255，可以采用直接寻址(直接写出端口地址)
- 若分配端口地址超出255，必须采用间接寻址(范围0~65535)

```
MOV DX,端口地址
IN AL,DX;注意不用打中括号，且必须位DX，字操作用AX，字节操作用AL
```

输出指令

OUT DST,SRC

- 若分配端口地址在0~255，可以采用直接寻址(直接写出端口地址)
- 若分配端口地址超出255，必须采用间接寻址(范围0~65535)

部分其他指令

NOP 空操作指令
什么也不干，就等着

字符串操作指令

共同特点:

1. 源和目的串的寻址方式均为隐含寻址
   - 如果源串在存储器中，则存储器的地址由DS:SI提供
     如果源串在寄存器中，字在AL中，字节在AX中
   - 如果目的串在存储器，则存储器的地址**必须**由ES:DI提供
     如果目的串在寄存器中，字在AL中，字节在AX中
2. SI和DI会自动变，方向受DF控制，增减多少受操作类型控制
   - DF=0 自动增
   - DF=1 自动减
3. 串操作指令左侧可以增加重复前缀(重复操作符)，重复次数在CX中，

字符串传送指令

MOVSB 按字节
MOVSW 按字
MOVS

字符串比较指令

CMPSB 按字节
CMPSW 按字
CMPS
比较本质上就是做减法，所以指令执行同时会设置6个状态标志

串扫描(搜索)指令

SCASB 按字节
SCASW 按字
SCAS
这个本质上也是做减法，所以指令执行同时会设置6个状态标志

串装入指令

LODSB 按字节
LODSW 按字

字符串存储指令

存到存储器中
STOSB 按字节(从AL搬运)
STOSW 按字(从AX搬运)

重复前缀

REP 相当于执行了一次loop

- 字符串传输指令
- 字符串装入指令
- 字符串存入指令

REPZ/REPE 相当于执行loopz

- 字符串比较指令
- 字符串扫描指令
  REONZ/REPNE 相当于执行loopnz
- 字符串比较指令
- 字符串扫描指令

标志位

CLD CF=0 递增
STD CF=1 递减

汇编语言程序设计基础

段定义伪指令

用于定义一个逻辑段
段名 SEGMENT [定位类型] [组合类型] ['类别']
一个逻辑段定义开始伪指令，段名第一个不为数字
段名 ENDS
段定义结束伪指令
注意:

1. 段名要一致
2. 要配对使用
3. 段名一旦用SEGMENT，段名本身就具有段地址属性/偏移地址属性(0)，直接用段名相当于给段地址，也算立即数寻址

参数

定位类型

告诉汇编程序本逻辑段起始地址的要求

1. PAGE(页)型
   1页=256字节，本段起始地址必须能被256整除
2. PARA(节)型
   1节=16字节，本1段起始地址必须能被16整除
   > 如果不写，默认节型
3. WORD(字)型
   本段起始地址必须能被2整除
4. BYTE(字节)型
   本段起始地址必须能被1整除

组合类型

在多模块程序设计中，告诉Link.exe(链接程序)同段名同组合类型的逻辑段如何链接

1. STACK型
   本段位为堆栈段

```
STACK SEGMENT STACK
      DB 256 DUP(?)
 STACK ENDS
```

在MASM.exe 5.0版本以上，这么定义会自动将SS，SP初始化，否则要自己初始化
2. NONE型
本逻辑段与任何逻辑段没有任何关系，同时也是缺省型

类别

类别在写的时候，必须要用**单引号**引起，且没有任何意义，目的是提高程序可读性

8086汇编语言源程序完整结构

```
STACK SEGMENT STACK
    DB 256 DUP(?)
TOP LABEL WORD
STACK EDNS

DATA SEGMENT
    DB "HELLO WORD"
    VAR1 DB 10H
    X_BYTE LABEL BYTE
    X_WORD DW 1234H
    COUNT EQU 5 + 3
    X = 1
DATA ENDS

CODE SEGMENT
    ASSUME CS:CODE,DS:DATA,ES:DATA,SS:STACK
    START: MOV AX,DATA
           MOV DS,AX
           MOV ES,AX
           MOV AX,STACK
           MOV SS,AX
           MOV SP,OFFSET TOP

           MOV AL,BYTE PTR X_WORD
           MOV AX,X_BYTE

           MOV AH,4CH
           INT 21H
CODE ENDS
     END START
```

LABEL 伪指令

TOP具有该单元段地址，偏移地址，类型的属性，但**不占用**该存储单元

> 本程序里，是为了将TOP的OFFSET给SP才这么设计(SP一开始时，处于栈底下一个单元)，实际上可以自动初始化SP

```
X_BYTE LABEL BYTE
X_WORD DW 1234H
```

此时想对该单元做字处理用X_BYTE变量，字节处理用X_WOED变量
等同于用PTR

ASSUME 段寻址伪指令

告诉段寄存器，哪个段名和其建立关系
这种关系是一种承诺关系(类似于唐妞不等式:我接受 $\neq$ 我同意)，并不代表段地址真的给CS,DS,ES,SS了,是需要用户自己初始化的

END 汇编结束伪指令

总汇编结束伪指令,后面要写起始地址表达式，目的是告诉系统，表达式对应的段地址给CS，偏移地址给IP

> CS和IP是系统完成的而非用户完成的

EQU 等值伪指令与 = 等号伪指令

右边的值均可由左边符号常量来替代

1. 二者右边的值范围均为0~65535
2. 用EQU定义的不能再次定义了，也就是说COUNT EQU 5+3只能出现一次
3. X = 1, X = X + 1，对于一个符号常量，等号伪指令可以多次定义

ORG 定位伪指令

ORG 0004H
为了对准地址(偶地址就是对准了)，相当于他下面的那一个单元的偏移地址是0004H

返回DOS系统

```
MOV AH,4CH
INT 21H
```

是为了结束用户程序，返回DOS系统中

子程序调用与返回指令

子程序(过程)定义

子程序名 PROC [类型]
类型

1. NEAR型 可缺省
2. FAR型 不可缺省

在下方编写指令，完成功能叫子程序体(过程体)
编写完成后写 RET或RETF(RETF也可写成RET)

子程序设计

```
子程序名 PROC [类型]
子程序体
RET
子程序名 ENDP
```

注意事项:

1. PROC与RET左侧子程序名要一致
2. PROC与ENDP成对使用
3. 子程序名一旦定义就有三个属性
    - 段地址属性(子程序体第一条指令的段地址)
    - 偏移地址(子程序体第一条指令的偏移)
    - 类型

子程序调用与返回

段内调用与返回

```
CODE SEGMENT
    ASSUME CS:CODE
START:
    CALL SUB1
    MOV AH,4CH
    INT 21H

SUB1 PROC

    RET
SUB1 ENDP

CODE ENDS
END START
```

执行内容

1. 执行CALL时当前IP值自动入栈，SP减2，同时IP加上一个偏移量
2. 执行RET指令时，从栈顶弹出一个值给IP，SP加2

段间子程序调用与返回

```
CODE1 SEGMENT
    ASSUME CS:CODE
START:
    CALL FAR PTR SUB1
    MOV AH,4CH
    INT 21H
CODE1 ENDS

CODE2 SEGMENT
ASSEME CS:CODE2
SUB1 PROC FAR

    RET
SUB1 ENDP
CODE2 ENDS

END START
```

执行内容

1. 先将CS入栈，IP入栈
2. 把子程序段地址给CS，偏移地址给IP
3. 弹栈两次，分别给IP和CS

注意:子程序里面PUSH/POP会影响压入的CS/IP值

中断调用与返回指令

中断源

外部中断源

- NMI引脚
  无条件响应，外部非可屏蔽中断请求，上升沿有效
- INTR引脚
  受到IF中断标志位控制，外部可屏蔽中断请求，上升沿有效
  IF = 1，CPU处于开中断状态，允许响应

内部中断源

1. 除法出错中断源
   除数为0或商溢出引起
2. 单步中断源
   TF = 1 时中断
3. INTO 溢出中断

```
ADD AX, BX
INTO
```

这里的INTO是一条指令，如果OF = 1，即发生溢出则产生溢出中断
4. 断点中断源
断点中断(单字节中断)指令 INT3
5. INT N指令

中断类型号

用于识别中断源的号码0~255，最多可管理256级中断

- 除法错 0
- 单步中断 1
- 断点中断 3
- 溢出中断 4
- INT N
- NMI 2
- INTR 的中断类型号是不确定，必须外挂8259中断控制芯片进行管理

中断向量

中断服务程序的入口地址(32位 = 16位段地址 + 16位偏移地址)

```
总断请求 INT
中断现场 MOV AX,BX
```

中断现场会把PSW(标志寄存器)，CS，IP的值依次压栈，称**现场保护**
然后中断后要转入的程序称为中断服务程序

中断向量表

把存储器中最低1024字节(00000H~003FFH)，作为中断向量表,里面存放中断向量，一个向量占用四个字节单元，从中断类型号0 ~256依次存放

$$N(\text{中断类型号}) \times 4 = \text{中断向量表中对应首地址}$$

头两个单元存的是偏移地址，后两个单元存的是段地址
所以，在设计程序时，要把中断服务程序入口地址写在对应的中断向量表中存储器单元里，称中断向量的建立

中断服务程序最后一条指令是IRET，这样就可以把栈内的IP,CS,PSW依次出栈，给到相应的寄存器，恢复现场

中断响应

内部中断与外部非可屏蔽中断

1. 保护现场，依次入栈
2. 清除IF，目的是为了不会响应外部可屏蔽中断
3. TF清零，目的是不会单步执行
4. 查中断向量表，进入中断
5. 恢复中断现场

外部可屏蔽中断

1. IF = 1时，保护现场，依次入栈
2. 清除IF，目的是为了不会响应外部可屏蔽中断
3. TF清零，目的是不会单步执行
4. 从8259中断控制芯片中获取中断类型号
5. 8086给$\overline{INTA}$端口一个低电平脉冲，表明收到请求
6. 8086再给$\overline{INTA}$端口一个低电平脉冲，8259会把具体中断源的值送到数据线，8086同时读取
7. 恢复中断现场

> 8086可以给8259 OUT出第一个位置上(IR0)的中断类型号，其余的依次加1

有N个8259芯片，级联在一切，可以管理$8 \times N - N$个中断

系统功能调用

DOS操作系统(高级功能调用)INT 21H
BIOS(低级功能调用)INT 10H

DOS功能调用

输入一个字符

01H 等待用户从键盘输入一个字符，回显在屏幕当前光标处，被按键字符放在AL中，检测ctrl break

```
MOV AH,01H
INT 21H; DOS功能调用,调用01H的功能
CMP AL,'1';看看输入的是不是1
```

07H 等待用户从键盘输入一个字符，被按键字符放在AL中，不检测ctrl break
08H 等待用户从键盘输入一个字符，被按键字符放在AL中，检测ctrl break

输出一个字符

02H 向屏幕输出一个字符，被输出字符放在DL中

```
MOV AH,02H
MOV DL,'5';被输出字符放在DL中
INT 21H
```

05H 向打印机输出一个字符，被输出字符放在DL中

```
;回车换行子程序,回车是将光标放到最左端,换行是换到下一行
MOV AH,2
MOV DL,0DH;回车
INT 21H
MOV AH,2
MOV DL,0AH;换行
INT 21H
```

直接控制台输入输出

06H 直接控制台输入输出

1. DL在00H~FEH之间，输出(屏幕上显示)

2. DL是FFH，输入(从键盘输入)，且**不等待**

```
FUNC1:
  MOV AH,06H
  MOV DL,0FFH;输入
  INT 21H;若没按键ZF=1，有按键，ZF=0，按键放AL中
  JZ NAJ;有按键接着走，没按键跳到NAJ
  CMP AL,1BH;ESC键
  JE START;跳到程序最前，返回主菜单
  JMP FUNC1;不是ESC这个功能从做一遍
NAJ:
```

输出一串字符

09H 向屏幕输出一串字符

```
DATA SEGMENT
STRING DB '2.Find the max of string:','0DH','01H','$'
DATA ENDS
CODE SEGMENT
  ASSUME CS:CODE,DS:DATA;
START:
  MOV AX,DATA
  MOV DS,AX
  ;做好输出准备，即输出字符串段地址给DS，偏移地址给DX
  MOV AH,09H
  LEA DX,STRING
  INT 21H;最后的$是结束符,不显示,等同于C中\0
  MOV AH,4CH
  INT 21H
  CODE ENDS
END START
```

输入一串字符

0AH 从键盘输入一串字符，有等待、回显功能
必须要定义键盘输入缓冲区

```
DATA SEGMENT
KEYBUF1 DB 20H;允许用户输入的字符个数是20H个，最大能输入255个，键盘缓冲区从这里开始
        DB ?;保留,用于统计实际输入的字符个数(自动)
KEYBUF2 DB 20H DUP(?);定义了32个字节(8位)空间，存放实际使用字符串,定义标号是方便找
DATA ENDS
CODE SEGMENT
  ASSUME CS:CODE,DS:DATA
START:
  MOV AX,DATA
  MOV DS,AX
  ;做好输入准备，即缓冲区段地址给DS，偏移地址给DX
  MOV AH,0AH
  MOV DX,OFFSET KEBUF1
  INT 21H;等待输入，直到给回车结束，回车会放入缓冲区内，但被字符串长度不会算回车
  CODE ENDS
END START
```

时间的设置与获取

2DH 时间设置

```
MOV AH,2DH
MOV CH,时
MOV CL,分
MOV DH,秒
MOV DL,百分秒
INT 21H;AL=00设置成功，AL=FF设置失败
```

2CH 时间获取

```
MOV AH,2CH
INT 21H;获取的时间放在上述寄存器中
```

BIOS功能调用

00H 设置屏幕分辨率

02H 设置光标位置

```
;设置屏幕分辨率
MOV AH,00H
MOV AL,3;设置成彩色文本模式，80×25，2000个字符
MOV BL,0;设置页号，第0页
INT 10H;一调用，就有清屏功能，光标回到左上角
;设置光标位置
MOV AH,02H
MOV DH,行
MOV DL,列
MOV BL,0;设置在0页，也可不写
INT 10H
```

宏指令

宏指令定义

MACRO 宏指令定义伪指令

```
宏指令名 MACRO [形参1,型参2,····]
;在此用指令开始编写程序完成功能，称为宏体

ENDM;宏定义结束伪指令，不用写宏名
```

宏展开

调用宏时，宏体插入过程叫宏展开
但是，宏指令里面有标号，通过宏指令调用两次以上，会报标号重复定义
那么要在宏内声明该标号是局部标号

```
DATAMOV MACRO
LOCAL NEXT;声明局部标号
NEXT:

    ENDM
```

例子

将某一通用寄存器左移或右移几次

```
SHIFT MACRO [REG,DIR,N]
    MOV CL,N
    DIR REG,CL;这里的DIR是4个移位指令
    ;S&DIR REG,CL 这里的DIR是4个移位指令一部分，即前面的S不用写了，就给后面的就可了
    ENDM
```

光标回车换行

```
CRLF MACRO
    MOV AH,2
    MOV DL,ODH
    INT 21H
    MOV AH,2
    MOV DL,0AH
    INT 21H
    ENDM
```