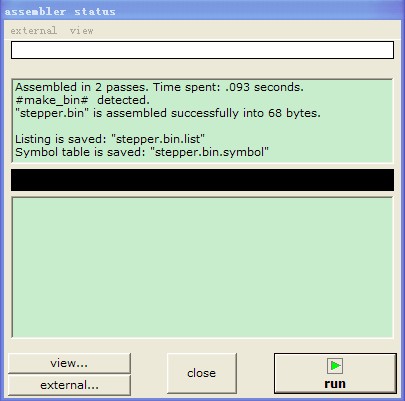
%----------最详细的emu教程--------%

Emu8086集源代码编辑器，汇编／反汇编工具以及可以运行debug的模拟器（虚拟机器）于一身，它优于一般编译器的地方在于提供了一个虚拟的80x86环境，拥有自己一套独立的“硬件”，可以完成一些纯软件编译器无法完成的功能例如Led显示，交通灯，步进电机等等，而且动态调试（DEBUG）时非常方便。

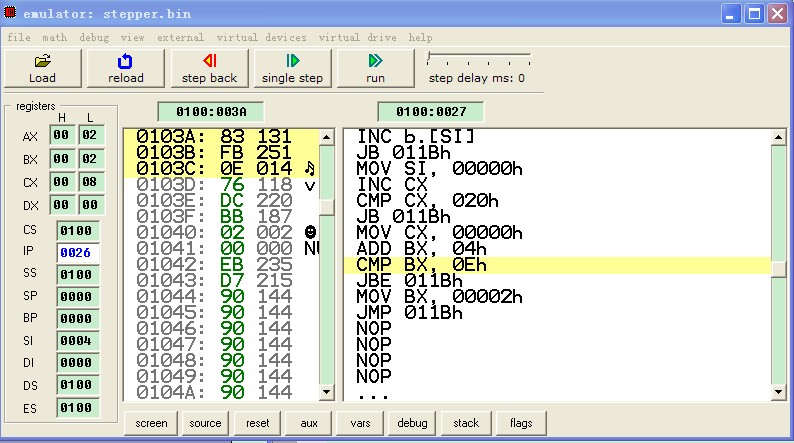
简单的例子：安装完成后选择 菜单栏中的文件 examples stepper motor 在编辑框出现了相应的源码。点击compile编译选择一个文件保存

保存完之后会弹出一个对话框



点击run按钮则程序开始运行

调试时主控界面如下：



再次点击run可以停止运行

单击reload可以从头开始执行程序

单击single step 可以单步调试。

单击step back可以返回到上一条指令（这个功能也是一般调试器没有的）。

界面左边是寄存器栏，这里可以动态的观察每一步的执行结果

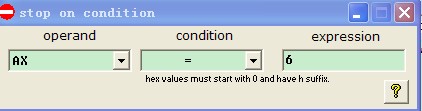
点击主控界面下面的screen可以显示模拟输出窗口

单击source可以查看源码窗口

Reset相当于上面的reload键。

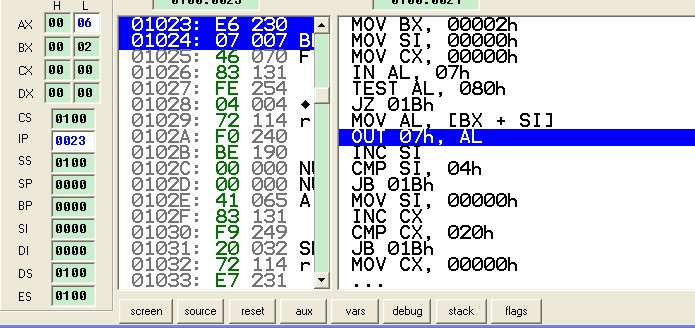
单击aux会出现一个菜单选择第一项memory可以观察程序内存区数值的变化。

选择stop on condition 可以设置条件断点：



上面的设定当ax的值是0x0006是断下来

单击run按钮可以看到断下来的时候ax值正好等于6



单击vars可以查看运行过程中变量的变化。

Debug可以更详细的显示每一步的调试结果

通过Stack（堆栈）可以观察函数调用的过程

Flags显示标志寄存器的值。如果刚刚执行的那一条指令修改了哪个标志位的值则以红色显示。

可以到这个网址去下载：

http://www.newhua.com/soft/16859.htm

目前网上很多人找这个软件的破解版。。其实根本不用破解。。

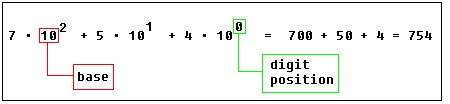
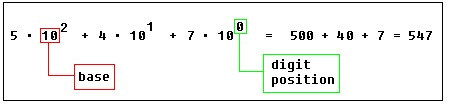
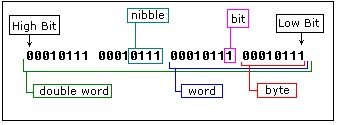
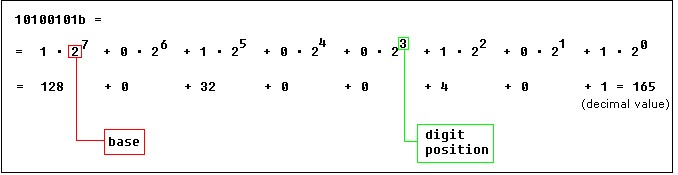
只要在用户名一栏输入任意的字符

注册码的前三位输入“112”就可以绕过注册认证了。。

至于为什么我就不多说了有兴趣的可以看一下我在看雪论坛发表的一篇文章：http://bbs.pediy.com/showthread.php?t=119741

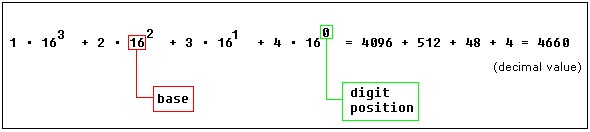
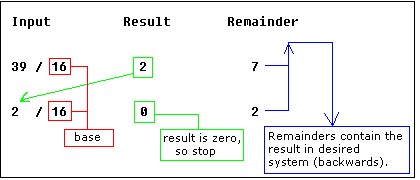
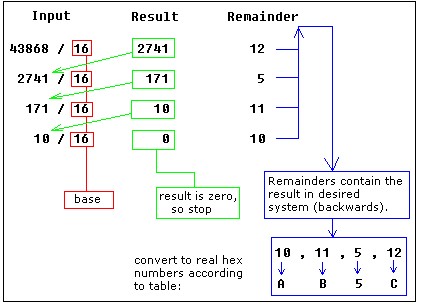
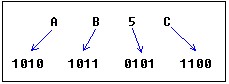
下面是在百度文库下载的一篇基础教程希望能对大家有所帮助。。。。

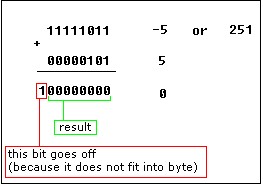
如何运行？  
１．在开始菜单选在它的图标，或者直接运行Emu8086．EXE  
２．在＂FILE＂菜单中选择＂SAMPLE＂  
３．点击＂Compile and Emulate＂按纽（或者按快捷键F５）  
４．点击＂Single Step＂按纽（或者按快捷键F８），可以查看代码如何运行．

十进制系统  
目前使用最多的是十进制．十进制系统有１０个数字０，１，２，３，４，５，６，７，８，９ 利用这些数字能表示任何数值，例如７５４这些数字是由每一位数字乘以“基数”的幂累加而成的（上一个例子中基数是10 因为十进制中有十个数字）。  
  
位置对于每一个数字是很重要的。例如，你将上一个例子中的“7”放到结尾：547  
数值就成为：  
  
**特别提醒**：任何数字的0次幂都是1，0的0次幂也是1  
QQ截图未命名  
二进制  
计算机没有人类聪明（至少现在是这样），制造一个只有开关或者称为 0，1 两种状态的电子机器很容易。计算机使用二进制系统，只有两个数字 **0, 1**基地为2每一位二进制数称作一位（BIT），4 BIT 组成一个半字节（NIBBLE），8BIT组成一个字节（BYTE），两个字节组成一个字（WORD），两个字组成一个双字（DOUBLE WORD）（很少使用）：  
  
习惯上在一串二进制后面加上“b”，这样，我们可以知道101b是二进制表示十进制的5。  
二进制**10100101b**表示十进制的165，计算方法如下：  


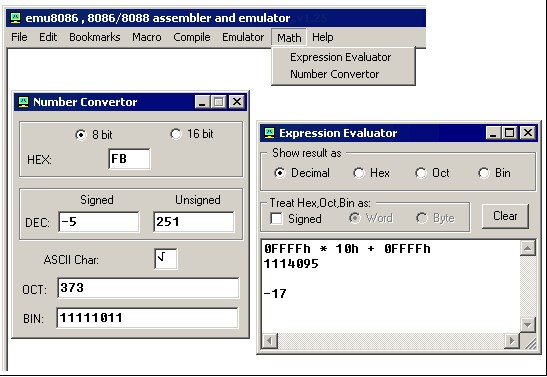
十六进制系统  
十六进制系统使用16个数字**0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F基底**是 **16**. 十六进制非常紧凑，便于阅读。将二进制转换为十六进制很容易，半字节（4bits） 对应一位十六进制如下表

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Decimal (base 10)** | **Binary (base 2)** | **Hexadecimal (base 16)** | | 0 | 0000 | 0 | | 1 | 0001 | 1 | | 2 | 0010 | 2 | | 3 | 0011 | 3 | | 4 | 0100 | 4 | | 5 | 0101 | 5 | | 6 | 0110 | 6 | | 7 | 0111 | 7 | | 8 | 1000 | 8 | | 9 | 1001 | 9 | | 10 | 1010 | A | | 11 | 1011 | B | | 12 | 1100 | C | | 13 | 1101 | D | | 14 | 1110 | E | | 15 | 1111 | F | | QQ截图未命名 |

习惯上我们在一个十六进制数的后面加上 **"H"**，以便和其他进制区别, 这样我们就知道 5Fh是一个十六进制数表示十进制的  95。习惯上，我们也在以字母开头（从Ａ到Ｆ）的十六进制数前面 加上**"0"　例如：** **0E120h**. 十六进制 **1234h**  等于　4660:  
  
十进制到另外进制的换算   
在换算中，将十进制数不断除以目标进制的基底，每一次都要记录下**商**和**余数**，直到商０。 **余数**用来表示结果。  
下面是一个十进制３９（基底是１０）到十六进制（基底是１６）的换算：  
结果为　**27H**  
上例中所有的余数都小于１０，不必使用字母。再举一个更复杂的例子：十进制 **43868** 换算为十六进制：  
结果是 **0AB5Ch**, 使用 [上面提到的表](mk:@MSITStore:C:\Documents%20and%20Settings\Administrator\桌面\Emu8086工具使用指南.CHM::/numbering_systems_tutorial.html#hextable#hextable) 将大于９的数字替换成字母。 运用同样的原理，我们可以换算为二进制（用２作除数），或者是先换算成十六进制，再用[上面的表](mk:@MSITStore:C:\Documents%20and%20Settings\Administrator\桌面\Emu8086工具使用指南.CHM::/numbering_systems_tutorial.html#hextable#hextable)　换算成二进制：  
于是，得到二进制： **1010101101011100b**

**有符号数**   
对于十六进制数**0FFh**无法确定它是正数还是负数，因为它可以表示十进制的"**255**" 或者　 "**- 1**"。  
８位可以表示**２５６**个状态，于是，我们可以假定前**１２８**个表示正数(**从0到127**)，接下来的１２８个数(**从128到256**)表示负数 。如果想表示"**- 5**"，我们从２５６中减去５，即　 **256 - 5 = 251**。用这种复杂的方法表示一个负数有着数学依据的，数学上"**- 5**" 加上 "**5**"等于０。当我们将两个８位的数字　**5** 和　**251**相加时，结果超过２５５，溢出处理为０！  


**128到256**高位始终是**１，**这个可以作为数字符号的标记 对于字（１６位），１６位有**65536**个状态，头３２７６８个状态(**从0到32767**)用来表示正数，下面的３２７６８个状态(**从32767到65535**) 表示负数

*Emu8086*带有数制转换工具，也可以计算各种数值表达式。选择菜单 **Math** 项:   
  
**Number Convertor** （数制转换）可以实现任意数制之间的转换。在文本框中填写源

数值，将自动转换到任意的数制。 可以作 **8 位** 或者 **16 位**转换。**Expression Evaluator（表达式计算）**可以用来计算不同数制的计算以及从一个进制到另一个进制的转换。输入表达式，按下回车，结果就会以你选定的进制表示。最长可以进行３２位的计算。当在**Signed**打钩选中时（除了八进制和双字），最前面的一位将被认作是符号位。这样以来，**0FFFFFFFFh**将被认为是十进制的　**－１**。例如，你计算　0FFFFh \* 10h + 0FFFFh ( 8086 CPU所能访问的最大内存地址)。如果你选中**Signed** 和 **Word**选项，结果是　-17　（因为表达式被认为是　(-1) \* 16 + (-1) ）。如果想按照无符号数计算，请不要选择**Signed**表达式为65535 \* 16 + 65535　计算结果将是1114095 同样你可以使用**Number Convertor**将非十进制换算为有**符号的十进制**，然后根据十进制计算。支持如下运算:

**~** **not　 (inverts all bits).**

**\***  **multiply.**

**/**  **divide**.

**%**  **modulus**.

**+**  **sum**.

**-** **subtract (and unary -).**

**<<** **shift left.**

**>>** **shift right.**

**&** **bitwise AND**.

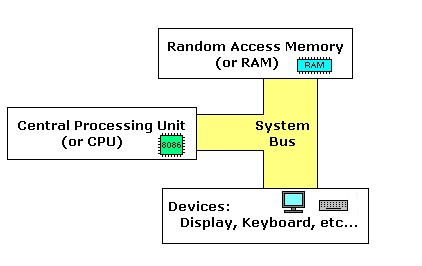
**^** **bitwise XOR**.

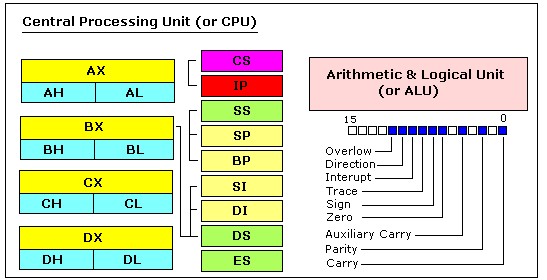
**|**  **bitwise OR.**

二进制必须有“ｂ”作结尾，例如00011011b 十六进制必须有"**h**"作结尾，另外，当地一位是字母时，最前面必须加上０，例如：0ABCDh八进制必须有"**o**"作结尾，例如：77o

什么是汇编语言？

汇编语言是底层编程语言。为了学习这门语言，你需要对于计算机结构有所了解。计算机系统模型如下：



**系统总线 system bus**（图中黄色部分）是将计算机各个部分连接到一起的部件。**CPU**是计算机的心脏，大部分的运算都是在CPU中完成的。**RAM**是读取并且存放将要执行的程序的地方。  
**CPU内部**   
  
通用寄存器  
8086CPU有8个通用寄存器，每一个寄存器都有自己的名称：

* **AX** 累加寄存器 accumulator register（分为 **AH / AL**）.
* **BX** 基址寄存器 base address register （分为 **BH / BL**）.
* **CX** 计数寄存器 count register（分为 **CH / CL** ）.
* **DX** 数据寄存器 data register （分为 **DH / DL**）.
* **SI** 源变址寄存器 source index register.
* **DI** 目的变址寄存器 destination index register.
* **BP** 基址指针寄存器 base pointer.
* **SP** 堆栈寄存器 stack pointer.

编程中，由程序员决定通用寄存器的具体用途。寄存器的主要目 的是保存数值（变量）。上面提到的寄存器是16位的，意思是: **0011000000111001b** （二进制），或者**12345**（十进制形式)。4个通用寄存器(AX, BX, CX, DX) 在使用时分为两个8位寄存器，例如 假设AX= **0011000000111001b**，AH=**00110000b** AL=**00111001b**。

当你修改其中任意8位值，整个16位寄存器的值同样改变。同样对于其他的3个寄存器，“H”表示高8位，“L”表示低8位。寄存器在CPU内部，访问中它们速度远远超过内存。因为，访问内存需要经过系统总线，所以时间要长一些。而访问寄存器中的数据几乎不需要时间。于是，编程中，应当尽量在寄存器中保存数据。虽然寄存器很小，并且这些寄存器都有具体用途，但他们依然是存放计算中临时数据的好地方。  
段寄存器

* **CS** 代码段寄存器，用来存放当前正在运行的指令
* **DS** 数据段寄存器，用来存放当前运行程序所用的数据
* **ES** 附加段寄存器，由程序员决定用途
* **SS** 堆栈段寄存器，指出堆栈所在区域

尽管容许在段寄存器中存放任何数据，但是这决不是 一个好主意。段寄存器有着非常特别的目的－－指出可以访问内存块的地址。段寄存器与通用寄存器协同工作就可以访问任意的内存区域。例如，如果我们打算访问物理地址是**12345h**（十六进制）的内存单元，我们应设置**DS = 1230h**  **SI = 0045h** 这样以来，我们便能访问超过一个寄存器（16位）所能表示的内存地址的范围。CPU计算物理地址的方法是将段寄存器乘以10H在加上一个特定的通用寄存器。(1230h \* 10h + 45h = 12345h):

QQ截图未命名这种，由两个寄存器生成的地址被称为有效地址 （effective address）

默认下，BX, SI 及 DI 与 DS协同工作，BP SP 与 SS 寄存器协同工作。其余的通用寄存器不能形成有效地址！同样，尽管BX可以形成有效地址，但是BH BL不能！控制寄存IP 指令指针寄存器 instruction pointer 、Flags Register 状态标志寄存器

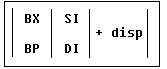
IP 始终同CS 协同工作，指出当前执行的指令。 Flags Register 完成一次数学运算后，由CPU自动修改，通过它可以得到当前结果类型，也可以作为跳转语句条件。通常你无法直接访问它们。

寻址方式

我们可以通过下面的四个寄存器来寻址 BX, SI, DI, BP.

通过计算[]符号中的值,我们可以访问到不同内存单元的值。具体组合请看下表：

|  |  |  |
| --- | --- | --- |
| [BX + SI] [BX + DI] [BP + SI] [BP + DI] | [SI] [DI] d16 (variable offset only) [BX] | [BX + SI] + d8 [BX + DI] + d8 [BP + SI] + d8 [BP + DI] + d8 |
| [SI] + d8 [DI] + d8 [BP] + d8 [BX] + d8 | [BX + SI] + d16 [BX + DI] + d16 [BP + SI] + d16 [BP + DI] + d16 | [SI] + d16 [DI] + d16 [BP] + d16 [BX] + d16 |

d8 - 表示８位偏移量 d16 - 表示１６位偏移量  
偏移量可以是一个立即数或者是一个变量的偏移，或者二者兼备。这取决于编译器如何计算单独的立即数。偏移量可以在［］符号里面或者外面，这不影响编译器生成相同的机器码。偏移量是一个有符号数，可以是正数或者负数。一般说来，8位或者16位，对于编译后的结果是有影响的。例如，假定 DS = 100, BX = 30, SI = 70。  
如下寻址方式 [BX + SI] + 25 计算物理地址为100 \* 16 + 30 + 70 + 25 = 1725  
默认下，DS 寄存器应用在除了BP寄存器之外的所有物理地址计算中，寄存器是和SS寄存器一起工作的。用过下面的表，你可以和轻松记住谁和谁是关联在一起使用的。  


上表中，你可以从每一列中选择一个或者忽略任意一个列。比如，可以看到，BX 和 BP始终不会选到一起。SI 和 DI不会选到一起。这是一个计算地址模式[BX+5] 段寄存器(CS, DS, SS, ES) 中数值被称作 "段偏移" 。目的寄存器(BX, SI, DI, BP) 中数值被称作"偏移量"

比如,ds中数值为1234h，si中数值为7890h,可以记作 1234:7890　物理地址为 1234h \* 10h + 7890h = 19BD0h在编译过程中使用如下声明数据类型

BYTE PTR - 表示字节 ； WORD PTR - 表示字（2个字节）  
例如：BYTE PTR [BX] ;按字节访问 ； WORD PTR [BX] ;按字访问  
Emu8086 容许使用如下更简洁的前缀  
b. - 等价于上面的 BYTE PTR ； w. - 等价于上面的 WORD PTR  
有时，编译器可以自动计算出数据类型，但是如果一个参与运算的数是立即数，这种方法就不可靠了。  
MOV 指令

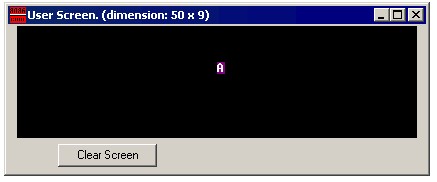
将第二个操作数（源）拷贝到第一个操作数（目的）指定位值 ，源操作数可以是立即数，通用寄存器或者内存单元，目的寄存器可以是通用寄存器或者内存单元 ，源和目的必须是同样大小，要么都是字节要么都是字

|  |
| --- |
| 操作类型如下:  MOV REG, memory MOV memory, REG MOV REG, REG MOV memory, immediate MOV REG, immediate  REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP. memory: [BX], [BX+SI+7],变量, 等等 immediate: 5, -24, 3Fh, 10001101b, 等等. |

|  |
| --- |
| mov　指令只支持如下段寄存器：  MOV SREG, memory MOV memory, SREG MOV REG, SREG MOV SREG, REG  SREG: DS, ES, SS, 注意 CS 只能作操作源 REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP. memory: [BX], [BX+SI+7], variable, 等等 |

MOV指令不能用来设置CS和IP寄存器的值。

|  |
| --- |
| 下面是一个使用 MOV 指令的例子: #MAKE\_COM#       　　 ;  表示，这个是一个com程序 ORG 100h         　　　　 ;COM 程序必须的 MOV AX, 0B800h    　　; 将ax设置为 B800h. MOV DS, AX        　　　; 将 AX 值拷贝到 DS. MOV CL, 'A'       　　　　; 将ASCII 码 'A'的值传送到cl,这个值是 41h. MOV CH, 01011111b 　　; 将ch设置为二进制的01011111b MOV BX, 15Eh     　　　 ;  将 BX 设置成 15Eh. MOV [BX], CX    　　  ; 将 CX 放到 bx 指出的内存单元 B800:015E RET             　　　　　　  ; 返回操作系统 |

你可以将上面的程序贴入Emu8086代码编辑器，接下来按下［complie and emulate]　(或者按F5) 模拟窗口将显示这个程序已经调入，点击［single step］观察寄存器数值变化，你可以猜到　";"　表示注释，编译器忽略在";"后面的一切，程序结束后，你可以看到如下窗口  
  
事实上，上面程序是将字符直接写入显示内存。  
通过上面的例子，你可以发现 MOV　指令是非常有用的。

变量

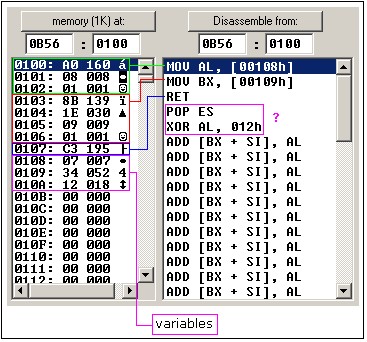
变量是一个内存地址。对于编程者来说，使用诸如名称为“var1”这样的 变量保存数据远远比使用5a73:235b这样的地址容易的多。特别是当你使用10个以上的变量的时侯。

编译器支持这两种变量 BYTE 和 WORD.（字节和字）

|  |
| --- |
| 声明变量的方法： name DB value 名称 DB 值 name DW value 名称 DW 值 DB - stays for Define Byte. DW - stays for Define Word. name －可以是任何字母与数字构成，但是必须由字母开头。可以通过不命名来声明一个 没有名称的的变量（这个变量只有地址，没有名称）  value - 可以是任何数值支持三种进制(十六进制,二进制和十进制),你可以使用"?"符号表示初始值没有确定。 |

你可能从第二章了解到， MOV 指令是将数值从源拷贝到目的。 让我们再看一个 MOV 指令的例子

|  |
| --- |
| #MAKE\_COM#  ORG 100h  MOV AL, var1  MOV BX, var2  RET ; stops the program.  VAR1 DB 7  var2 DW 1234h |

将上面的代码拷贝到emu8086源程序编辑器中，按下F5键编译 并在模拟器中执行。你会看到如下画面  
  
从画面可以看出，反编译后的代码同源程序很相似，不同的是变量被具体的内存地址取代。当编译器生成机器代码它会自动将变量名称用该变量的便宜量代替。默认情况下，DS 寄存器存放段偏移（当执行com文件的时侯，DS 寄存器的值同 CS 寄存器（代码段）的值一样）。内存第一列是偏移（offset），第二列是一个十六进制值（hexadecimal value），第三列是十进制（decimal value），最后一列是 ASCII 字符。编译器是非大小写敏感的，所以 “VAR1” 同 “var1” 都是同一个变量。

VAR1变量的偏移是0108h，物理地址是0b56:0108  
var2 变量的偏移是0109h，物理地址是 0b56:0109

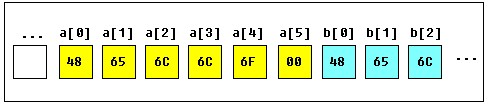
这个变量是字，它占用2字节。这里假定低字节存放在低地址，所以34h位于12h前面。  
你可以看到，在RET指令后面还有一些指令，这样是因为反编译工具无法判断数据从什么地方开始。同样，你可以写出直接使用DB的程序.

|  |
| --- |
| #MAKE\_COM#  ORG 100h  DB 0A0h  DB 08h  DB 01h  DB 8Bh  DB 1Eh  DB 09h  DB 01h  DB 0C3h  DB 7  DB 34h  DB 12h |

将上面的代码拷贝到emu8086原代码编辑器，按下F5键编译，并在模拟器中运行，你可以看到同样的反汇编结果，得到同样的功能。根据上面，你可以猜测，编译器将源程序转化为一些字节的集合，这个集合被称作机器代码(machine code)，处理器懂得他们，并且执行它们。ORG 100是一个编译指令（它告诉编译器如何处理源代码）当你使用变量的时侯，这条指令特别重要。它通知编译器可执行程序将被调入偏移量是100h（256字节）的位置，有了它，编译器就可以计算出所有变量的正确地址，然后用这些地址（偏移量）来代替变量名称。上面的这些指令不会真正的编译为任何机器代码。为何可执行程序总是被装入偏移量100h?操作系统在CS寄存器(代码段)存储着程序信息，比如命令行方式下的参数等等。尽管上面只是一个COM文件的例子，EXE文件调入在偏移量0000的位置，他使用特定的段保存变量。我们在下面会学习到关于EXE文件的知识。

数组  
数组可以看作是变量链。一个字符串是一个字节数组的例子，其中每一个字符都当作一个ASCII码的值（0....255）下面是一些定义数组的例子

a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h

b DB 'Hello', 0   
b是一个数组，当编译器发现引用了字符串值后，会自动将这些字符转化为对应的字节。下面图表表示的就是声明数组后在内存中的分布：  
  
你可以使用方括号做下标直接访问到数组中的值，例如：  
MOV AL, a[3] 同样，你还可以使用任意一个内存索引寄存器BX, SI, DI, BP，例如：

MOV SI, 3  
MOV AL, a[SI]  
如果你想声明比较复杂的数组，你可以使用DUP指令 形式如下number DUP ( value(s) )   
number - 重复的数量（任意常数）

value - 将要复制的表达式  
例如：c DB 5 DUP(9) 就相当于如下定义：c DB 9, 9, 9, 9, 9 另外一个例子：d DB 5 DUP(1, 2) 等同于d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2 当然，如果需要存放超过255或者小于－128的数值，你还可以使用DW来代替 DB。但是DW不能用于声明字符串。DUP命令展开后不能超过1020个字符（上一个例子中展开之后是13个字符），如果需要声明请将它们分成两行（这样，内存中得到的仍然是一个大数组）。取得变量地址 LEA指令（Load Effective Address 读取有效地址）或者OFFSET指令。OFFSET 和 LEA二者都能够获得变量的偏移量。LEA在使用中更有效，这是因为它能返回索引变量的地址。取得变量地址在很多情况下是非常有用的，例如你打算向一个过程传递参数。注意：在编译过程中使用如下声明数据类型 BYTE PTR - 表示字节 ；WORD PTR - 表示字（2个字节）  
例如：  
BYTE PTR [BX] ;按字节访问 ； WORD PTR [BX] ;按字访问  
Emu8086 容许使用如下更简洁的前缀

b. - 等价于上面的 BYTE PTR ；w. - 等价于上面的 WORD PTR  
有时，编译器可以自动计算出数据类型，但是如果一个参与运算的数是立即数，这种方法就不可靠了。

第一个例子：

|  |
| --- |
| ORG 100h  MOV AL, VAR1 ; 将变量var1的数值放入al以便检查  LEA BX, VAR1 ; 将var1的地址存入 BX.  MOV BYTE PTR [BX], 44h ; 修改变量var1的内容  MOV AL, VAR1 ; 将变量VAR1的数值放入AL以便检查  RET  VAR1 DB 22h  END |

下面是另外一个例子，用OFFSET指令代替LEA：

|  |
| --- |
| ORG 100h  MOV AL, VAR1 ; 将变量VAR1的值放入AL以便检查.  MOV BX, OFFSET VAR1 ; 将变量VAR1的地址放入 BX.  MOV BYTE PTR [BX], 44h ; 修改变量VAR1内容  MOV AL, VAR1 ;将变量VAR1的值放入 AL以便检查.  RET  VAR1 DB 22h  END |

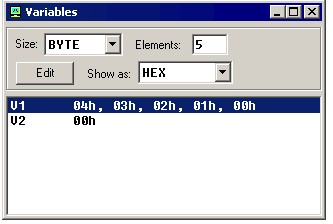
上面例子的功能相同。  
这些语句：  
LEA BX, VAR1  
MOV BX, OFFSET VAR1   
都将生成同样的机器代码: MOV BX, num，num 是16位变量偏移

请注意，只有这些寄存器可以放入方括号中（作为内存指针）BX, SI, DI, BP(请参考本教程前述章节)  
常量  
常量同变量很相似，但是它一直存在。定义一个变量之后，它的值 不会改变。使用EQU定义常量：name equ <任意表达式> 例如：

|  |
| --- |
| k EQU 5  MOV AX, k |

上面的例子等同于如下代码：

|  |
| --- |
| MOV AX, 5 |

在程序执行过程中你可以选择模拟器"View"菜单下的"Variables"  
  
你可以点一个变量然后设置Elements属性为数组大小来查看数组。汇编语言对于数据类型并不严格，这样以来所有的变量都可以被看 作是数组。变量可以显示为下列进制

HEX - 十六进制 hexadecimal (基底 16).

BIN - 二进制 (基底 2).

OCT - 八进制 (基底 8).

SIGNED - 有符号十进制 (基底 10).

UNSIGNED - 无符号十进制 (基底 10).

CHAR - ASCII 码 (一共有256个符号，其中一些符号是不可见的).

程序运行的时侯，你可以通过双击它来编辑变量值，或者选中之后点Edit按钮。

十六进制数值以"h"结尾，二进制以"b" 结尾，八进制以"o" 结尾，十进制没有结尾。字符串用这样的方式表示：'hello world',0  
(结尾以0表示)

数组按照如下输入：1, 2, 3, 4, 5

（数组可以是一组字节或者字，这取决于你想以字节还是字的方式编辑）  
表达式会自动计算，例如，输入如下表达式5 + 2会自动计算为7。等等....

中断   
中断是一系列功能调用。这些功能调用使得编程更加容易。比如，你想在打印机上输出一个字符，你只需要简单的调用中断，它将帮你完成所有的事情。另外还有控制磁盘和其他硬件工作的中断。我们将这些功能调用称作软件中断。不同的硬件同样可以触发中断，这些中断称作硬件中断。这里，我们只介绍软件中断(software interrupts)。触发一个软件中断，需要使用INT指令，它的使用方式非常简单：

INT value

上面value的取值范围是从 0 到 255 （或者0到0ffh），通常我们使用十六进制。 你也许猜测只有256个中断调用，但是这是不正确的。因为每一个中断都有子功能。

在调用一个中断的子功能之前，需要设置AH寄存器。每一个中断最多可以拥有256个子功能（于是，我们有256\*256＝65536个功能调用）。一般情况下使用AH寄存器，但是一些情况下可能使用另外的寄存器。通常，其他的寄存器是用来传递数据和参数的。

下面的例子调用了 INT 10h中断0Eh子功能输出字符串‘Hello!'。这个功能作用是在屏幕上显示一个字符，然后光标进一，如果需要还滚屏。

|  |
| --- |
| #MAKE\_COM# ; 生成com文件的指令  ORG 100h  ；我们使用的这个子功能没有返回值，  ；所以我们只用设置就可以了。  MOV AH, 0Eh ; 选择子功能  ；int 10h/0eh 子功能，输出放在  ；AL寄存器中的ASCII码对应的字符  MOV AL, 'H' ; ASCII码: 72  INT 10h ; 输出  MOV AL, 'e' ; ASCII 码: 101  INT 10h ; 输出  MOV AL, 'l' ; ASCII 码: 108  INT 10h ; 输出  MOV AL, 'l' ; ASCII 码: 108  INT 10h ; 输出  MOV AL, 'o' ; ASCII 码: 111  INT 10h ; 输出  MOV AL, '!' ; ASCII 码: 33  INT 10h ; 输出  RET ; 返回操作系统 |

将上述程序拷贝粘贴到Emu8086代码编辑器，点击 [Compile and Emulate] 按钮，运行！

常用函数库 - emu8086.inc

通过引用一些常用函数，可以使你编程更加方便。在你的程序中使用其他文件中的函数的方法是  
INCLUDE后面接上你要引用的文件名。编译器  
会自动在你源程序所在的文件夹中查找你引用的文件，如果没有找到，它将搜索Inc 文件夹。通常你无法完全理解 emu8086.inc（位于Inc文件夹）但是这没有关系，你只用知道它能做什么就足够了。要使用emu8086.inc中的函数，你应当在你程序的开头加上  
include 'emu8086.inc'

emu8086.inc 定义了如下的宏:

PUTC char - 将一个ascii字符输出到光标当前位值，只有一个参数的宏

GOTOXY col, row - 设置当前光标位置，有两个参数

PRINT string - 输出字符串，一个参数

PRINTN string - 输出字符串，一个参数。与print功能相同，不同在于输出之后自动回车

CURSOROFF - 关闭文本光标

CURSORON - 打开文本光标

使用上述宏的方法是：在你需要的位值写上宏名称加上参数。例如:

|  |
| --- |
| include emu8086.inc  ORG 100h  PRINT 'Hello World!'  GOTOXY 10, 5  PUTC 65 ; 65 - ASCII 码的 'A'  PUTC 'B'  RET ; 返回操作系统  END ; 停止编译器 |

当编译器运行你的代码时，它首先找到声明中的emu8086.inc文件，然后将代码中的宏用实际的代码替换掉。通常来说，宏都是比较小的代码段，经常使用宏会使得你的可执行程序特别大（对于降低文件大小来说使用过程更好）

emu8086.inc 同样定义了如下过程:

PRINT\_STRING - 在当前光标位置输出一个字符串字符串地址由DS:SI 寄存器给出使用时，需要在END前面声明DEFINE\_PRINT\_STRING 才能使用.

PTHIS - 在当前光标位置输出一个字符串（同 PRINT\_STRING）一样，不同之处在于它从堆栈接收字符串。字符串终止符

应在call之后定义。例如

   CALL PTHIS  
db 'Hello World!', 0

使用时，需要在 END 前面声明

DEFINE\_PTHIS 。GET\_STRING - 从用户输入得到一个字符串，输入的字符串写入 DS:DI 指出的缓冲，缓冲区的大小由 DX设置。回车作为输入结束。使用时，需要在END前面声明

DEFINE\_GET\_STRING 。CLEAR\_SCREEN - 清屏过程(滚过整个屏幕),然后将光标设置在左上角. 使用时，需要在END前面声明DEFINE\_CLEAR\_SCREEN  。  
SCAN\_NUM - 取得用户从键盘输入的多位有符号数，并将输入存放在CX寄存器。 使用时，需要在 END前面声明      DEFINE\_SCAN\_NUM。

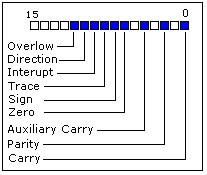
PRINT\_NUM - 输出AX寄存器中的有符号数。使用时，需要在END 前面声明 DEFINE\_PRINT\_NUM以及 DEFINE\_PRINT\_NUM\_UNS.

PRINT\_NUM\_UNS - 输出AX寄存器中的无符号数。使用时，需要在END 前面声明DEFINE\_PRINT\_NUM\_UNS.

使用上述过程，必须在你源程序的底部（但是在END之前！！！）声明这些函数，使用CALL指令后面接上过程名称来调用。例如：

|  |
| --- |
| include 'emu8086.inc'  ORG 100h  LEA SI, msg1 ; 要求输入数字  CALL print\_string ;  CALL scan\_num ; 读取数字放入cx  MOV AX, CX ; CX存放数值拷贝到AX; 输入如下字符  CALL pthis  DB 13, 10, 'You have entered: ', 0  CALL print\_num ; 输出 AX中的字符  RET ; 返回操作系统  msg1 DB 'Enter the number: ', 0  DEFINE\_SCAN\_NUM  DEFINE\_PRINT\_STRING  DEFINE\_PRINT\_NUM  DEFINE\_PRINT\_NUM\_UNS ; print\_num函数要求的  DEFINE\_PTHIS  END ; 结束 |

首先，编译器运行声明（对于宏只是展开）。当编译器遇到CALL指令，它 将用过程声明中的地址来替代过程名。程序在执行过程中遇到这个过程，便会直接跳转到过程。这是非常有用的，比如，即使在你的代码中执行100次一个过程，编译后的可执行文件也不会因此而增大多少。这样看起来很划算，是不是？后面你会学到更多的，现在只需要了解一点点基本原理。

运算与逻辑指令  
大多数运算与逻辑指令影响处理器的状态标记寄存器。  
  
　从上图可以看到，这是状态标记寄存器是一个16位寄存器，每一位称作一个标志位，可以取值 1 或者 0 。  
　**进位标志 Carry Flag (CF)** - 出现无符号（unsigned overflow）溢出该位设置成1。例如，计算 255+1（结果超出0...255）。没有溢出时该位为0。

**零标志 Zero Flag (ZF)** - 当结果为 0 时设置为1，结果不为 0 时设置为0。

**符号标志 Sign Flag (SF)** -  结果为负置1，结果为正置为0。事实上该位对于结果特别重要。

**溢出标志 Overflow Flag (OF)** -  当出现有符号数溢出设置为1。例如，计算100+50（结果超出－128－127的范围）。

**奇偶标志 Parity Flag (PF)** - 当结果操作数中1的个数为偶时置1，否则为0注意，如果结果是一个字，该标志只指示低8位。

**辅助进位标志 Auxiliary Flag (AF)** -  低4位向上进位时置1，否则为0（记录运算时第3位（半个字节）产生的进位值。例如，执行加法指令时，最高有效位有进位时置1，否则置0  
中断标志 Interrupt enable Flag (IF) - 当cpu容许中断时为1，否则为0

Direction Flag (DF) - 方向标志，在串处理指令中控制处理信息的方向用。当DF为1时，每次操作后使变址寄存器SI和DI减量，这样就使串处理从高地址向低地址方向处理。当DF为0时，则使SI和DI增量，使串处理从低地址向高地址方向处理。

这里有3组指令.

第一组： ADD, SUB,CMP, AND, TEST, OR, XOR  
支持如下操作数:

REG, memory  
memory, REG  
REG, REG  
memory, immediate  
REG, immediate

REG（寄存器）: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.  
memory（内存）: [BX], [BX+SI+7], 变量,等等...  
immediate（立即数）: 5, -24, 3Fh, 10001101b, 等等...  
执行之后，结果经常存放在第一个操作数中。CMP和TEST指令只影响标志位，并不返回数值（这两条指令是用来在程序运行中判断的）上述指令只影响如下标志位 ：

CF, ZF, SF, OF, PF, AF.

ADD - 将第二个操作数加至第一个操作数上

SUB - 从第一个操作数中减去第二个操作数

CMP - 从第一个操作数中减去第二个操作数，但只影响标志位.

AND - 两个操作数各个位逻辑与运算。运算法则如下

1 AND 1 = 1 1 AND 0 = 0  
0 AND 1 = 0 0 AND 0 = 0

只有当两个操作数都是1时，运算结果才是1。

TEST - 和上面的and 操作一样，但是只影响标志位。

OR - 两个操作数各个位逻辑或运算。运算法则如下

1 OR 1 = 1 1 OR 0 = 1  
0 OR 1 = 1 0 OR 0 = 0

如果操作数中有1那么结果一定是1。

XOR -  两个操作数各个位逻辑异或运算。运算法则如下

1 XOR 1 = 0 1 XOR 0 = 1  
0 XOR 1 = 1 0 XOR 0 = 0

当两个操作数不同时，结果为1。

第二组: MUL, IMUL, DIV, IDIV  
支持如下操作数:

REG  
memory

REG（寄存器）: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.  
memory（内存）: [BX], [BX+SI+7], variable, etc...  
MUL and IMUL 指令只影响 CF, OF标志位。  
 运算后如果结果超出范围，这些标记位置1，如果没有超过范围，置0  
DIV 和 IDIV 指令对于标志位无影响

MUL - 无符号乘:

当操作数是字节时: AX = AL \* 操作数.

当操作数是字时: (DX AX) = AX \* 操作数.

IMUL - 有符号乘法:

当操作数是字节时: AX = AL \* 操作数.

当操作数是字时: (DX AX) = AX \*  操作数.

DIV - 无符号除法:

当操作数是字节时: AL = AX / 操作数 AH = 余数（取模后的余数） .

当操作数是字时:  
AX = (DX AX) / 操作数 DX = 余数（取模后的余数） IDIV - 有符号除法:

当操作数是字节时:  
AL = AX / 操作数 AH =余数（取模后的余数）

当操作数是字时:  
AX = (DX AX) / 操作数 DX = 余数（取模后的余数） .

第三组: INC, DEC, NOT, NEG

支持如下操作数:

REG  
memory

REG（寄存器）: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.  
memory（内存）: [BX], [BX+SI+7], variable, etc...  
INC, DEC 指令只影响如下标志位:  ZF, SF, OF, PF, AF.  
NOT 指令不影响任何标志位！  
NEG i指令只影响如下操作位： CF, ZF, SF, OF, PF, AF.

NOT - 对与操作数每一位取反

NEG - 对操作数取反  
实际上它对每一位取反然后在最后一位加1。例如5会变成－5，－2会变成2。（这里所说运算应当是计算机内部的补码运算）

程序控制转移  
对于编程来说控制程序走向是非常重要的事情，它是你的程序根据条件作出判断，跳转到相应的位值。  
无条件跳转  
控制程序转向的最基本的指令是JMP.   
使用形式如下： JMP label

在程序中声明/label/的方法很简单，只要在它名字后面加上“：”，

label可以由任何字符混合而成但是不能由数字开头，例如，下面是3个合法的label

label1:  
label2:  
a:

label可以在一条指令的前面声明，例如:

x1:  
MOV AX, 1  
x2: MOV AX, 2

下面是一个 JMP 指令的例子:

|  |
| --- |
| ORG 100h  MOV AX, 5 ; 将 AX 设置为 5.  MOV BX, 2 ; 将 BX 设置为 2.  JMP calc ; 跳转到 'calc'.  back: JMP stop ; 跳转到 'stop'.  calc:  ADD AX, BX ; 将 BX 加到 AX.  JMP back ; 返回 'back'.  stop:  RET ; 返回操作系统  END |

当然有更简单的计算这两个数字之和的方法，但是上面是一个JMP指令的很好的例子。

从例子中可以看出，JMP可以控制向前和向后。它可以转移到当前代码段的任意位置（65535字节）。短条件转移与JMP这一无条件转移指令不同，还有一些有条件跳转指令（只有在条件成立的时侯才跳转）。这些指令分为三组，第一组是只检测单独标记位，第二组比较有符号数，第三组比较无符号数。检测单独标记位的转移指令

|  |  |  |  |
| --- | --- | --- | --- |
| 指令 | 说明 | 条件 | 相反指令 |
| JZ , JE | 如果为0(相等)，转移 . | ZF =1 | JNZ, JNE |
| JC , JB, JNAE | 如果进位 (小于, 不大于等于)，转移 | CF = 1 | JNC, JNB, JAE |
| JS | 如果是负数，转移 | SF = 1 | JNS |
| JO | 如果溢出，转移 | OF = 1 | JNO |
| JPE, JP | 如果是偶数，转移 | PF = 1 | JPO |
| JNZ , JNE | 如果不为0（不相等），转移 | ZF = 0 | JZ, JE |
| JNC , JNB, JAE | 如果没有进位(大于, 大于等于)，转移 | CF = 0 | JC, JB, JNAE |
| JNS | 如果不是负数，转移 | SF = 0 | JS |
| JNO | 如果没有溢出，转移 | OF = 0 | JO |
| JPO, JNP | 如果不是偶数，转移 | PF = 0 | JPE, JP |

可以看到一些指令功能相同，对，他们编译之后生成相同机器码所以很容易理解为什么你编译 JE 指令而反编译得到的却是JZ.使用不同的名称是为了使程序更容易理解。比较有符号数的转移指令

|  |  |  |  |
| --- | --- | --- | --- |
| 指令 | 说明 | 条件 | 相反指令 |
| JE , JZ | 如果等于 (=)，如果为0，跳转 | ZF = 1 | JNE, JNZ |
| JNE , JNZ | 如果不等于 (<>)，如果不等于0，跳转 | ZF = 0 | JE, JZ |
| JG , JNLE | 如果大于 (>) 如果不小于等于 (not <=)，跳转 | ZF = 0 和 SF = OF | JNG, JLE |
| JL , JNGE | 如果小与Jump if Less (<) 如果不大于等于 (not >=)，跳转 | SF <> OF | JNL, JGE |
| JGE , JNL | 如果大于等于 (>=)，如果不小于 (not <)，跳转 | SF = OF | JNGE, JL |
| JLE , JNG | 如果小于等于 (<=)，如果不大于 (not >)，跳转 | ZF = 1或者SF <> OF | JNLE, JG |

<> - 符号表示不等于. 比较无符号数转移指令

|  |  |  |  |
| --- | --- | --- | --- |
| 指令 | 说明 | 条件 | 相反指令 |
| JE , JZ | 如果等于 (=).，如果为0，跳转 | ZF = 1 | JNE, JNZ |
| JNE , JNZ | 如果不等于(<>)，如果不为0，跳转 | ZF = 0 | JE, JZ |
| JA , JNBE | 如果大于 (>)，如果不小于等于(not <=)，跳转 | CF = 0 and ZF = 0 | JNA, JBE |
| JB , JNAE, JC | 如果小于 (<)，如果不大于等于(not >=)，如果进位，跳转 | CF = 1 | JNB, JAE, JNC |
| JAE , JNB, JNC | 如果大于等于(>=)，如果不小于 (not <)，如果没有进位，跳转 | CF = 0 | JNAE, JB |
| JBE , JNA | 如果小于或者等于(<=)，如果不大于 (not >)，跳转 | CF = 1or ZF = 1 | JNBE, JA |

一般来说，需要使用CMP指令来比较数值（该指令与 SUB(减法)

指令相近，只不过不保存结果，而只修改标值位）上面说法的意思是，例如：需要比较5 和2， 5－2 ＝3结果不是0（0标值位设置为 0）另一个例子比较 7和7

7 - 7 = 0结果为0！ (0标值位设置为1。 JZ 或者 JE 会转移).   
下面是一个 CMP 指令和条件转移指令的例子：

|  |
| --- |
| include emu8086.inc  include emu8086.inc  ORG 100h  MOV AL, 25 ; 设置AL为 25.  MOV BL, 10 ; 设置BL为10.  CMP AL, BL ; 比较 AL - BL.  JE equal ; 如果 AL = BL (ZF = 1) 跳转  PUTC 'N' ; 如果到这里，说明 AL <> BL,  JMP stop ; 打印'N', 跳转到结束  equal: ; 如果到这里  PUTC 'Y' ; 则 AL = BL,打印'Y'.  stop:  RET  END |

请用用不同的数字试验取代上述 AL 和 BL，点击[FLAGS]键打开标志，使用[Single Step]观察发生了什么，不要忘记每一次修改之后重新编译运行（快捷键F5）。

全部的条件转移指令都有一个很大的限制，就是与 JMP 指令不同，他们只能向前跳转127字节或者向后跳转128字节（注意大多数指令编译之后是3个或者更多字节）我们可以用如下小技巧解决这一问题：  
从上述表中找到一条相反条件的转移指令，令其跳转到 label\_x. 用JMP指令跳转到你想要的地方在JMP指令后面定义label\_x: label\_x: - 可以是任意合法标号.   
下面是一个例子:

|  |
| --- |
| include emu8086.inc  ORG 100h  MOV AL, 25 ; 设置 AL 为 25.  MOV BL, 10 ; 设置 BL 为 10.  CMP AL, BL ; 比较 AL - BL.  JNE not\_equal ; 如果 AL <> BL (ZF = 0)，转移  JMP equal  not\_equal:  ; 假定这里还有编译之后超过127字节的程序  PUTC 'N' ; 如果执行到这里，说明 AL <> BL,  JMP stop ; 打印 'N', 转移到程序结束。  equal: ; 如果执行到这里，  PUTC 'Y' ; 说明 AL = BL, 打印 'Y'.  stop:  RET ; 上述都要执行这一条  END |

另外，可以使用立即数来代替标号。立即数前使用“$”编译器将直接得到偏移。例如:

|  |
| --- |
| ORG 100h  ; 无条件向前转移  ; 跳过后面2字节  JMP $2  a DB 3 ; 1 byte.  b DB 4 ; 1 byte.  ; JCC 跳过 7 字节:  ; (JMP 本身占用 2 字节)  MOV BL,9  DEC BL ; 2 bytes.  CMP BL, 0 ; 3 bytes.  JNE $-7  RET  END |

堆栈   
堆栈是内存中用于保存临时数据的一片区域.当使用CALL指令时，堆栈用于保存过程的返回地址，RET指令能够从堆栈中取得该地址并使程序返回到那里。当使用INT指令，发生的也与此类似。  
堆栈保存标志寄存器，代码段和偏移量。IRET指令用来从中断返回。 我们同样可以使用堆栈保存任何数据。对于堆栈的操作只有两条：  
PUSH - 将16位数值压入堆栈. POP - 将16位数值从堆栈中弹出

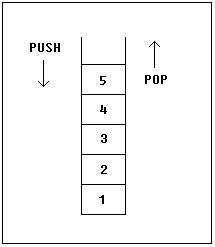
|  |
| --- |
| PUSH 指令的使用方法:  PUSH REG PUSH SREG PUSH memory PUSH immediate  REG（寄存器）: AX, BX, CX, DX, DI, SI, BP, SP. SREG（段寄存器）: DS, ES, SS, CS. memory（内存）: [BX], [BX+SI+7], 16 位变量, 等等... immediate（立即数）: 5, -24, 3Fh, 10001101b,等等... |

|  |
| --- |
| POP 指令的使用方法:  POP REG POP SREG POP memory  REG（寄存器）: AX, BX, CX, DX, DI, SI, BP, SP. SREG（段寄存器）: DS, ES, SS, (除了 CS). memory（内存）: [BX], [BX+SI+7], 16位变量, 等等... |

注意：

PUSH and POP 都只操作16位数据!

注意：

在80186其极以后的CPU中才能使用 PUSH 立即数这样的指令堆栈使用LIFO（后进先出）算法，意思是：加入我们按照如下顺序压入数值： 1, 2, 3, 4, 5  
 再使用POP指令弹出，结果将是 5 4 3 2 1  
  
注意，有多少条PUSH指令就要对应有多少条POP指令，否则堆栈会被占用，无法正确返回操作系统。前面讲过使用RET指令返回操作系统，所以在程序开始时会将返回地址压入堆栈（通常都是0000h）I PUSH 和 POP指令在我们寄存器不够用的时侯特别有用，我们有如下技巧：

将寄存器原始数值存入堆栈（使用 PUSH）使用寄存器从堆栈中弹出寄存器原先数值再放入寄存器（使用POP） 下面是一个例子:

|  |
| --- |
| ORG 100h  MOV AX, 1234h  PUSH AX ; 将 AX 存入堆栈.  MOV AX, 5678h ; 修改 AX 值  POP AX ;返回 AX 原先的值  RET  END |

堆栈的另外一个作用是交换数值，下面是一个这样的例子:

|  |
| --- |
| ORG 100h  MOV AX, 1212h ; 将 1212h 存入 AX.  MOV BX, 3434h ; 将 3434h 存入 BX  PUSH AX ; 将 AX 数值存入堆栈.  PUSH BX ; 将 BX 数值存入堆栈  POP AX ; BX原值存入AX  POP BX ; AX原值存入BX  RET  END |

之所以能这样是因为堆栈是用LIFO（后进先出）算法，当我们压入1212h和3434h之后，使用pop弹出我们首先得到的是3434h然后才是1212h

堆栈的内存区域由SS寄存器（堆栈段），SP寄存器（栈指针）设置设置。一般来说操作系统在程序开始时会设置这些。 "PUSH 源"  指令做如下工作：

将SP寄存器减 2

将源的值写入内存SS:SP地址处

"POP 目的"  指令做如下工作:

内存SS:SP地址处数值写入目的

将SP寄存器加2

由 SS:SP 指出的地址称作堆栈顶  
对于COM文件，堆栈段通常就是代码段，堆栈指针设置为  
0FFFEh.在地址SS:0FFFEh处存放程序结束时RET指令返回地址。你可以点击[stack]按钮直接观察堆栈操作。堆栈顶由“<”符号标记。

宏

宏与过程很相似，但并不是完全相似。宏看起来像过程，但是当你的代码编译完成之后就消失了，取而代之的是真正的代码。如果你声明一个宏，而在代码中从来没有调用，编译器在编译过程中将忽略它。

|  |
| --- |
| 宏的定义 :  name MACRO [参数,]  <指令>  ENDM |

与过程不同，宏要求定义参数并使用。例如:

|  |
| --- |
| MyMacro MACRO p1, p2, p3  MOV AX, p1  MOV BX, p2  MOV CX, p3  ENDM  ORG 100h  MyMacro 1, 2, 3  MyMacro 4, 5, DX  RET |

上述代码在编译过程中将展开成:

MOV AX, 00001h  
MOV BX, 00002h  
MOV CX, 00003h  
MOV AX, 00004h  
MOV BX, 00005h  
MOV CX, DX

|  |
| --- |
| 关于宏与过程需要注意如下要点:  当你想使用一个过程，你应该使用CALL指令，例如：CALL MyProc 当你想使用一个宏，你只需要输入它的名称。例如:MyMacro  过程是存在于内存中某一特定位值的，即使你调用这个过程100次，cpu只是执行内存中这一段的代码。在遇到 RET指令后还会回到调用该过程的位值。这是通过使用堆栈保存返回地址来实现的。CALL指令占用3字节，无论调用多少次过程，最终输出的可执行文件并不会因此而显著增大。  宏会在程序代码中展开。如果你使用相同的宏100次，输出的可执行文件将会变得越来越大，因为每一次调用宏中的指令都会插入到调用宏的位值。  你可以使用堆栈或者通用寄存器来向过程传递参数向宏传递参数的方法是在宏名称后面直接接上参数。例如: MyMacro 1, 2, 3  用ENDM指令结束宏就足够了标记过程结束，你需要在ENDP指令前加上过程名称 |

宏会直接在代码中展开，因此，如果你在宏中使用标记，当宏被调用2次或两次以上的时侯就会出现  
"Duplicate declaration"（重复定义） 这一错误。为了避免该错误  
在变量，标记或者过程名称之前加上“local”指令。例如:

|  |
| --- |
| MyMacro2 MACRO  LOCAL label1, label2  CMP AX, 2  JE label1  CMP AX, 3  JE label2  label1:  INC AX  label2:  ADD AX, 2  ENDM  ORG 100h  MyMacro2  MyMacro2  RET |

I若过你打算在很多程序中使用宏，将所有的宏存放在一个文件中不失为一个好办法。将那个文件放在INC目录下，使用 INCLUDE 文件名 就可以在你的程序中调用宏了。