窗体顶端

[8086 CPU 寄存器简介](http://www.cnblogs.com/BoyXiao/archive/2010/11/20/1882716.html)

2010-11-20 20:51 by Zachary.XiaoZhen, 13968 visits, [网摘](http://www.cnblogs.com/BoyXiao/archive/2010/11/20/1882716.html), [收藏](http://www.cnblogs.com/BoyXiao/AddToFavorite.aspx?id=1882716), [编辑](http://www.cnblogs.com/BoyXiao/admin/EditPosts.aspx?postid=1882716)

**引子**

打算写几篇稍近底层或者说是基础的博文，浅要介绍或者说是回顾一些基础知识，

自然，还是得从最基础的开始，那就从汇编语言开刀吧，

从汇编语言开刀的话，我们必须还先要了解一些其他东西，

像  CPU ，内存这些知识点还是理解深刻一点的比较好，

所以这一篇博文就绕着 80x86  CPU 中寄存器的基础部分下手，至于其他的一些将会在后续的博文中介绍 。

同时在这里说明一下，本篇博文介绍的算是比较详细的了，而且介绍的知识点也是比较多的，所以造成博文长度过长，

如果有兴趣想了解这一块的话，还请自行斟酌好阅读比例，建议分 3 次以上阅览 。

**读者定位**

本博文主要将介绍的是  8086 CPU 中的寄存器，既然是 8086 CPU 寄存器简介的话，

自然，面向的是初级一些的读者，其中不会涉及太多难点，同时，所有的介绍，我也会尽可能的从基础开始，

然后循序渐进的介绍，同时也会尽量的将知识点介绍详细，

介绍的过程中也会涉及到一些汇编程序代码，当然，采用的是最简单的方式介绍而已，

本篇博文也就是回顾一些基础知识，读者主要定位于想对 8086 CPU 有所了解，

希望对整个程序设计的底层有所了解的朋友，而且读者最好是拥有一定的计算机基础和汇编语言基础 。

**开头**

首先浅要介绍一下  Intel  CPU 的发展史吧：

Intel CPU 系列，最初是 4 位微处理器 4004，然后到到 8 位微处理器的 8008 ，

再到 8 微微处理器 8080,以及稍后的 16 位微处理器 8086，

由 8086 开始，Intel 进入现在所谓的  x86  时代 。

Intel  8086 为 16 位  CPU ，而因为在 8086 之前的 CPU 都是 8 位 CPU，这样也就造成了很多的外设也只支持 8 位，

因此  Intel  紧接着就退出了 8 位的 8088 CPU，因此  Intel 8088 也就可以看做是 8086 的 8 位版本；

如果是但从汇编语言的角度上来说，8086 和 8088 是没有区别的，即 8086 上跑的程序可以不加修改的移植到 8088 ，

8088 上跑的程序也可以不加修改的移植到 8086 上，

当然，还是有些特殊的地方是不同的，而这些基本上在这里可以忽略掉，

在 8088  CPU 之后，Intel  又推出了  80186 ，80286 ，这两款 CPU 均是 16 位  CPU ，

而对于 80186 来说，其与 8086 的区别可以简单的看做是 80186 多了几条指令而已，

而 80286 则不同，80286 的地址总线数目有了变化，

在 8086 , 8088 , 80186 上，CPU 的地址总线都是 20 根，即可最大寻址 220即达到 1MB 的寻址能力，

而对于 80286 CPU 来说，其地址总线数目达到了 24 根，从而最大寻址能力为 224 即 16MB，

由于支持更多的物理内存寻址，因此 80286 便开始成为了多任务，多用户系统的核心。

而后来，Intel  又推出了 80386 ，80386 为 32 位微处理器，Intel 80x86 家族的 32 位微处理器始于 80386；

同时 80386 也完全兼容先前的 8086/8088，80186，80286，并且 80386 全面支持 32 位数据类型和 32 位操作，

并且 80386 的数据总线根数和地址总线根数均达到了 32 根，从而可以最大物理寻址为 232即 4GB 。

而之后的 80486 也是 32 位微处理器，而后又出来了 Pentium 和 Pentium Pro 等等第五代微处理器，

这些处理器虽然也是 32 位微处理器，但是他们的数据总线和地址总线都有所扩展，

比如 Pentium 的数据总线达到 64 位，而 Pentium Pro 的地址总线位数达到了 36 位 。

好，关于 Intel CPU 的介绍就到这里了，下面就要开始回归中心，看 CPU 中的寄存器了，

首先，从学习的角度来说，从 8086/8088  CPU 下手是不错的选择，而我这里选择的也是 8086 CPU 而已，

说实在的，像 80386 CPU 我也还没有研究过，像奔腾这些，呵呵，扯更远了，

说到底也就只能拿 8086 出来晒晒而已，当然，从 8086 开始也是学习的最佳路径 。

说了这么久，到底寄存器是什么呢？其实很简单，寄存器就是个存储信息的单元或者说是器件又或者说是容器而已，

就比如内存也是一个存储介质或者说是存储单元而已，其实寄存器从理解上来说和内存差不多，

只不过寄存器（这里讨论的寄存器都是 CPU 中的寄存器，不包括外设上的寄存器）位于  CPU  内部，而内存位于 CPU 外部，

而且，寄存器比内存可是珍贵得多啊，就拿内存和硬盘来比，肯定是内存在使用上珍贵得多，是 PC 中的稀有资源，

而寄存器是 CPU 中的稀有资源，内存和寄存器相比就像硬盘和内存相比一样 。

而对于一个汇编程序员来说，CPU 中主要可以使用的也就是寄存器而已，汇编程序员可以使用指令来读写 CPU 中的寄存器，

从而可以实现对于 CPU 的控制，当然，不同的 CPU ，寄存器的个数和结构都是不一样的，

比如 8086 CPU 中，寄存器的个数也就 14 个而已，

并且 8086 CPU 中所有的寄存器的结构为 16 位，即一个寄存器中可以存放下 2B 即 2 个字节，

而到了 80386 CPU 中，寄存器的个数也比 8086 增多了，比如在 80386 中添加了系统地址寄存器等寄存器，

同时寄存器的结构也变了，比如在 80386 中绝大多数的寄存器为 32 位，而有些寄存器则是 16 位 。

**8086  CPU 中寄存器总共为 14 个，且均为 16 位 。**

即 **AX，BX，CX，DX，SP，BP，SI，DI，IP，FLAG，CS，DS，SS，ES** 共 14 个。

而这 14 个寄存器按照一定方式又分为了通用寄存器，控制寄存器和段寄存器。

**通用寄存器：**

**AX，BX，CX，DX 称作为数据寄存器：**

AX (Accumulator)：累加寄存器，也称之为累加器；

BX (Base)：基地址寄存器；

CX (Count)：计数器寄存器；

DX (Data)：数据寄存器；

**SP 和 BP 又称作为指针寄存器：**

SP (Stack Pointer)：堆栈指针寄存器；

BP (Base Pointer)：基指针寄存器；

**SI 和 DI 又称作为变址寄存器：**

SI (Source Index)：源变址寄存器；

DI (Destination Index)：目的变址寄存器；

**控制寄存器：**

IP (Instruction Pointer)：指令指针寄存器；

FLAG：标志寄存器；

**段寄存器：**

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)：附加段寄存器；

**通用寄存器**

从上面可以知道，在 8086 CPU 中，通用寄存器有 8 个，分别是 AX，BX，CX，DX，SP，BP，SI，DI ，

至于为什么给它们取名做通用寄存器，那是因为，这些个寄存器每一个都有自己专门的用途，

比如 CX 作为计数寄存器，则是在使用 LOOP 指令循环时用来指定循环次数的寄存器，

如果它们每一个都只有一个专用的作用，那就它们只能称之为专用寄存器了，

正是因为这些个寄存器还可以用来传送数据和暂存数据，所以才称它们为通用寄存器 。

**下面就按顺序来一一介绍这几个通用寄存器了：**

**数据寄存器（AX，BX，CX，DX）：**

数据寄存器有 AX，BX，CX，DX 四个组成，

由于在 8086 之前的 CPU 为 8 位 CPU，所以为了兼容以前的 8 位程序，

在 8086 CPU 中，每一个数据寄存器都可以当做两个单独的寄存器来使用，

由此，每一个 16 位寄存器就可以当做 2 个独立的 8 位寄存器来使用了 。

**AX 寄存器可以分为两个独立的 8 位的 AH 和 AL 寄存器；**

**BX 寄存器可以分为两个独立的 8 位的 BH 和 BL 寄存器；**

**CX 寄存器可以分为两个独立的 8 位的 CH 和 CL 寄存器；**

**DX 寄存器可以分为两个独立的 8 位的 DH 和 DL 寄存器；**

**除了上面 4 个数据寄存器以外，其他寄存器均不可以分为两个独立的 8 位寄存器 ；**

注意在上面标志中的“独立”二字，这两个字表明 AH 和 AL 作为 8 位寄存器使用时，

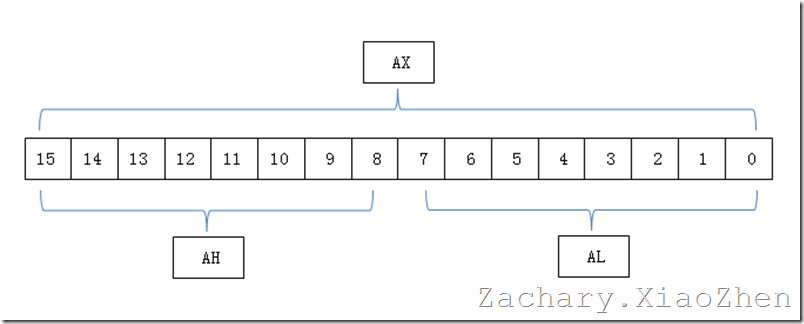
可以看做它们是互不相关的，也就是看做两个完全没有联系的寄存器 X 和 Y 即可，

比如指令   MOV   AH , 12H ，CPU 在执行时根本就不会知道 AL 中是什么鬼东西，因为它只认识  AH 。

**下面给出一幅 16 位数据寄存器的结构图：**

**表示 16 位 寄存器 AX 可以表示成两个 8 位寄存器，**

**其中 AH 表示高位的 8 位寄存器，AL 表示低位的 8 位寄存器 。**

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_2.png)

**AX 寄存器：**

如上所说，AX 的另外一个名字叫做累加寄存器或者简称为累加器，其可以分为 2 个独立的 8 位寄存器 AH 和 AL；

在写汇编程序时，AX 寄存器可以说是使用率最高的寄存器（不过，总共才那么 14 个寄存器，哪一个不经常使用咯？），

既然 AX 是数据寄存器的话，那么理所当然，其可以用来存放普通的数据，由于其是 16 位寄存器，

自然也就可以存放 16 位数据，但是因为其又可以分为 2 个独立的 8 位寄存器 AH 和 AL ，

所以，在 AH 和 AL 中又可以独立的存放 2 个 8 位的数据，

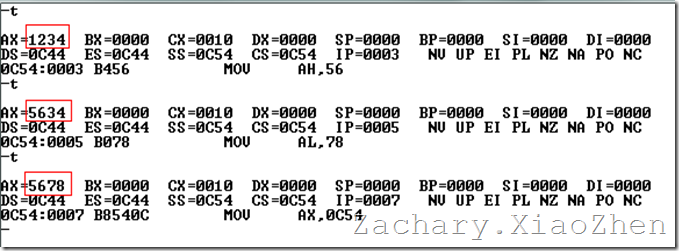
**可以有以下代码（即将 AX 当做普通的寄存器使用，即可以用来暂存数据）：**

MOV AX,1234H ;向寄存器 AX 传入数据 1234H

MOV AH,56H ;向寄存器 AX 的高 8 位寄存器 AH 中传入数据 56H

MOV AL,78H ;向寄存器 AX 的低 8 位寄存器 AL 中传入数据 78H

**3 条语句的执行过程如下：**

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image4_1.png)

**而既然 AX 又被称作为累加器，自然其还有一点点特殊的地方的：**

AX 寄存器还具有的特殊用途是在使用 DIV 和 MUL 指令时使用，

DIV 在 8086 CPU 中是除法指令，而在使用除法的时候有两种情况，即除数可以是 8 位或者是 16 位的，

而且除数可以存放在寄存器中或者是内存单元中，而至于被除数的话，自然，应该由 AX 来代替了，

当除数是 8 位时，被除数一定会是 16 位的，并且默认是放在 AX 寄存器中，

而当除数是 16 位时，被除数一定是 32 位的，因为 AX 是 16 位寄存器，自然，放不下 32 位的被除数，

所以，在这里还需要使用另一个 16 位寄存器 DX ，

其中 DX 存放 32 位的被除数的高 16 位，而 AX 则存放 32 位的被除数的低 16 位，

同时，AX 的作用还不仅仅是用来保存被除数的，当除法指令执行完成以后，

如果除数是 8 位的，则在 AL 中会保存此次除法操作的商，而在 AH 中则会保存此次除法操作的余数，

当然，如果除数是 16 位的话，则 AX 中会保存本次除法操作的商，而 DX 则保存本次除法操作的余数。

上面介绍的是 AX 寄存器在除法操作中的应用，下面还需要介绍一下 AX 在乘法操作中的应用，

当使用 MUL 做乘法运算时，两个相乘的数要么都是 8 位，要么都是 16 位，

如果两个相乘的数都是 8 位的话，则一个默认是放在 AL 中，

而另一个 8 位的乘数则位于其他的寄存器或者说是内存字节单元中，

而如果两个相乘的数都是 16 位的话，则一个默认存放在 AX 中，

另一个 16 位的则是位于 16 的寄存器中或者是某个内存字单元中。

同时，当 MUL 指令执行完毕后，如果是 8 位的乘法运算，则默认乘法运算的结果是保存在 AX 中，

而如果是 16 位的乘法运算的话，则默认乘法运算的结果有 32 位，

其中，高位默认保存在 DX 中，而低位则默认保存在 AX 中。

**AX 寄存器在 DIV  指令中的使用：**

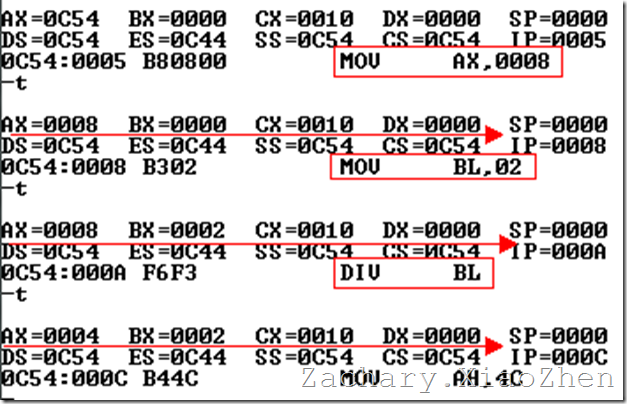
MOV DX,0H ;设置 32 位被除数的高 16 位为 0H

MOV AX,8H ;设置 32 位被除数的低 16 位为 8H

MOV BX,2H ;设置 16 位除数为 2H

DIV BX ;执行计算

**4 条语句的执行过程如下：**

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_8.png)

**AX 寄存器在 MUL  指令中的使用：**

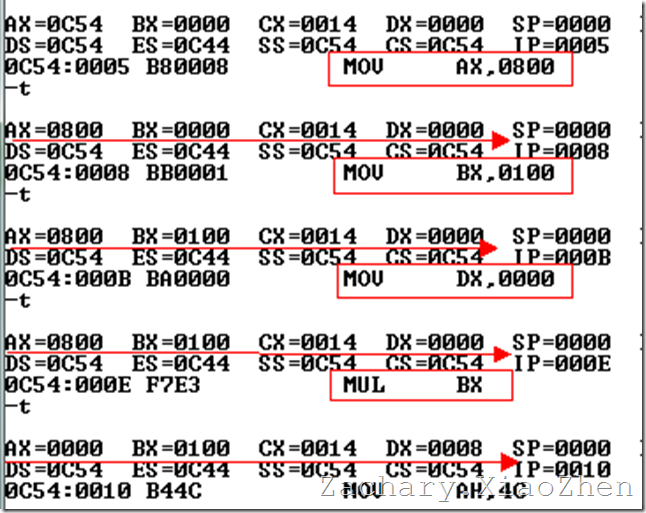
MOV AX,800H ;设置 16 位乘数为 800H

MOV BX,100H ;设置 16 位乘数为 100H

MOV DX,0H ;清空用来保存乘法结果的高 16 位

MUL BX ;执行计算

**4 条语句的执行过程如下：**

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_10.png)

**BX 寄存器：**

首先可以明确的是，BX 作为数据寄存器，表明其是可以暂存一般的数据的，

即在某种程度上，它和 AX 可以暂存一般性数据的功能是一样的，

其同样为了适应以前的 8 位 CPU ，而可以将 BX 当做两个独立的 8 位寄存器使用，即有 BH 和 BL，

除了暂存一般性数据的功能外，BX 作为通用寄存器的一种，BX 主要还是用于其专属功能 – 寻址（寻址物理内存地址）上，

BX 寄存器中存放的数据一般是用来作为偏移地址使用的，何为偏移地址呢？

既然是偏移地址的话，当然得有一个基地址了，而这个基地址其实就是段地址，这里就涉及到了段寄存器，

当然，在介绍 BX 寄存器的时候，我不会去介绍段寄存器，上面提到 BX 的主要功能是用在寻址上，

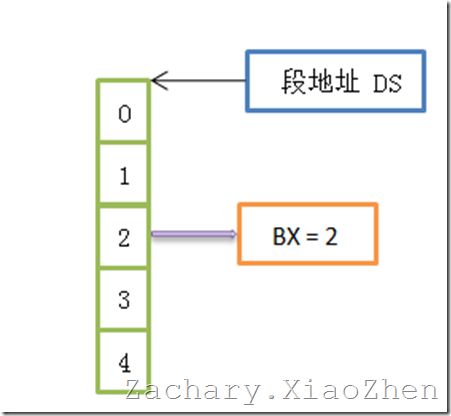
那么，其是如何寻址的呢？

对于寻址这个话题，我会在我的下一篇博文中作出详细的介绍，

而这里，我只点一下，在 8086 CPU 中，CPU 是根据 <段地址：偏移地址> 来进行寻址操作的，

而 BX 中存放的数据表示的是偏移地址的话，自然，便可以通过 <段地址：[BX]> 的方式来完成寻址操作了。

**为了介绍 BX 在寻址当中的作用，下面我给出一副示意图：**

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_6.png)

上面的示意图表示：可以令 BX = 2，然后通过 DS : [BX] 来访问到内存中段地址为 DS，且偏移量为 2 的内存单元了。

上面介绍的这种寻址方式是 BX 在寻址中最最简单的应用了，而对于稍微复杂的寻址方式，

还可以依赖于 SI，DI，BP 等寄存器来一起完成，当然，这会是下一篇博文将要介绍的内容了。

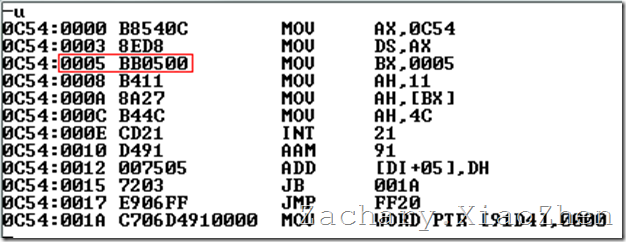
BX**寄存器在寻址中的使用：**

MOV BX,5H

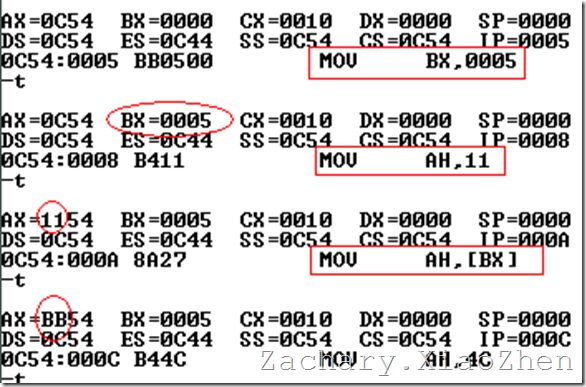
MOV AH,11H

MOV AH,[BX] ;设置 AX 的值为偏移地址为 BX 中的值时所代表的内存单元

**3 条语句的执行过程如下：**

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_12.png)

**从上图可以看出，在偏移地址为 5 时的内存单元中的数据位 BBH，**

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_14.png)

而从这幅图上面就可以看出，确实通过 [BX] 找到了偏移地址为 5 处的内存单元，并且将内存单元移入了 AH 中。

**CX 寄存器：**

CX 寄存器作为数据寄存器的一种呢，其同样具有和 AX，BX 一样的特点，即可以暂存一般性的数据，

同时还可以将其当做两个独立的 8 位寄存器使用，即有 CH 和 CL 两个 8 位寄存器，

当然，CX 也是有其专门的用途的，CX 中的 C 被翻译为 Counting 也就是计数器的功能，

当在汇编指令中使用循环 LOOP 指令时，可以通过 CX 来指定需要循环的次数，

而 CPU 在每一次执行 LOOP 指令的时候，都会做两件事：

一件就是令 CX = CX – 1，即令 CX 计数器自动减去 1；

还有一件就是判断 CX 中的值，如果 CX 中的值为 0 则会跳出循环，而继续执行循环下面的指令，

当然如果 CX 中的值不为 0 ，则会继续执行循环中所指定的指令 。

CX**寄存器在循环中的使用（输出 5 个白底蓝字的 A）：**

MOV AX,0B800H

MOV DS,AX ;使用 80x25 彩色字符模式，内存地址 0xB8000 - 0xBFFFFF

MOV BX,0 ;从 0xB8000 开始

MOV CX,5H ;循环 5 次

MOV DX,41H ;A 的16 进制为 41H

MOV AX,01110001B ;显示白底蓝字

s: MOV [BX],DX ;显示 ASCII 字符

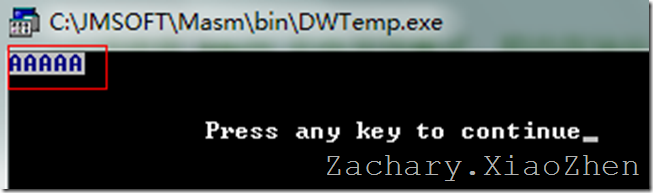
ADD BX,1

MOV [BX],AX ;设置字符显示属性

ADD BX,1

LOOP s

**语句的执行过程如下：**

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_16.png)

**DX 寄存器：**

DX 寄存器作为数据寄存器的一种，同样具有和 AX，BX，CX 一样的特点，即可以暂存一般性的数据，

同时还可以将其当做两个独立的 8 位寄存器使用，极有 DH 和 DL，

同时，DX 作为一个通用寄存器的话，自然其还有其他的用途，而关于 DX 在其他方面的用途，

其实在前面介绍 AX 寄存器时便已经有所介绍了，

即当在使用 DIV 指令进行除法运算时，如果除数为 16 位时，被除数将会是 32 位，而被除数的高 16 位就是存放在 DX 中，

而且执行完 DIV 指令后，本次除法运算所产生的余数将会保存在 DX 中，

同时，在执行 MUL 指令时，如果两个相乘的数都是 16 位的话，

那么相乘后产生的结果显然需要 32 位来保存，而这 32 位的结果的高 16 位就是存放在 DX 寄存器中 。

**DX 寄存器在 DIV  指令中的使用（即 2293812 / 256 = 8960  余数为 52）：**

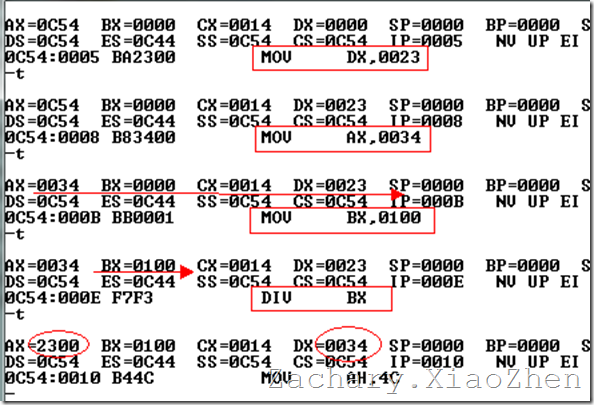
MOV DX,0023H ;32 位被除数的高 16 位

MOV AX,0034H ;32 位被除数的低 16 位

MOV BX,100H ;16 的除数

DIV BX

**语句的执行过程如下：**

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_18.png)  
可以看到在语句结束以后，AX = 2300H  即十进制的 8960，而 DX = 34H即十进制的 52 和我们的结果是一致的。

**DX 寄存器在 MUL  指令中的使用则各位可以参考在 AX 中 MUL 运算的使用，这里就不贴出来了。**

**指针寄存器（BP，SP）：**

**BP 寄存器：**

8086  CPU 中的指针寄存器包括两个，即 SP 和 BP ，在这里呢，我先只对 BP 寄存器做介绍，

因为 SP 寄存器实质上必须和 SS 段寄存器一起使用，所以，我将会把 SP 寄存器留到后面和 SS 段寄存器一起作介绍。

BP (Base Pointer)也就是基指针寄存器，它和其他的几个用来进行寻址操作所使用的寄存器（还有 BX，SI，DI）没有太大的区别，

关于 SI 和 DI 寄存器的下面请见下文。

首先，BP 寄存器作为通用寄存器的一种，说明其是可以暂存数据的，而后，BP 又不是数据寄存器，

也就意味着其不能分割成 2 个独立的 8 位寄存器使用，

而后当以 […] 的方式访问内存单元而且在 […] 中使用了寄存器 BP 的话，

那么如果在指令中没有明确或者说是显示的给出段地址时，

段地址则使用默认的 SS 寄存器中的值（BX，SI，DI 会默认使用 DS 段寄存器），

比如 DS:[BP] 则在这里明确给出了段地址位于 DS 中，

所以，这里代表的内存单元即是段地址为 DS ，偏移量为 BP 寄存器中的值的内存单元，

而如果单单是使用 [BP] 的话，则代表的内存单元是段地址为 SS，偏移量为 BP 寄存器中的值的内存单元。

并且 BP 寄存器主要适用于给出堆栈中数据区的偏移，从而可以方便的实现直接存取堆栈中的数据，

至于堆栈的话，会在后面的博文中介绍。

**在 8086 CPU 中，只有 4 个寄存器可以以  […]  的方式使用，这四个寄存器分别是 BX，SI，DI，BP。**

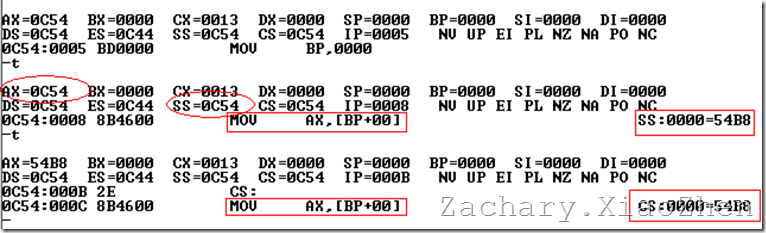
**下面的  Demo  是**BX**寄存器在寻址中的使用：**

MOV BP,0

MOV AX,[BP] ;将 SS:[BP] 代表的内存单元移入 AX 中

MOV AX,CS:[BP] ;将 CS:[BP] 代表的内存单元移入 AX 中

**语句的执行过程如下：**

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_20.png)

**变址寄存器（SI，DI）：**

首先，变址寄存器和上面介绍的指针寄存器（也就是 BP 和 SP），它们的功能其实都是用于存放某个存储单元地址的偏移，

或者是用于某组存储单元开始地址的偏移，即作为存储器指针使用，当然，由于变址寄存器和指针寄存器都是属于通用寄存器，

所以它们也可以保存算术结果或者说是具有暂存数据的功能，但是因为它们不是数据寄存器，所以无法分割成 2 个独立的 8 位寄存器使用，

关于变址寄存器和指针寄存器的详细使用，笔者将会在下一篇博文中作出最详细的介绍，

SI (Source Index) 是源变址寄存器，DI (Destination Index) 即是目的变址寄存器，

8086 CPU 中的 SI 寄存器和 DI 寄存器其实和 BX 寄存器的功能是差不多的，

只不过 SI 寄存器和 DI 寄存器均不是数据寄存器，所以它们不能够拆分为 2 个独立的 8 位寄存器，

而这也就是 SI 寄存器和 DI 寄存器与BX 寄存器所不同的地方，

既然，SI，DI 两个寄存器的功能和 BX 差不多，自然，SI 和 DI 中也是可以暂存一般性数据的，

同时，通过使用 SI 和 DI 寄存器也是可以用来完成寻址操作的。

**比如下面的代码就是可行的：**

MOV SI,0 ;初始化偏移地址为 0

MOV AX,[SI] ;将段地址为 DS 偏移地址为 SI 的内存单元中的值移入 AX 中

MOV AX,DS:[SI] ;将段地址为 DS 偏移地址为 SI 的内存单元中的值移入 AX 中

MOV AX,SS:[SI] ;将段地址为 SS 偏移地址为 SI 的内存单元中的值移入 AX 中

MOV DI,0 ;初始化偏移地址为 0

MOV AX,[DI] ;将段地址为 DS 偏移地址为 DI 的内存单元中的值移入 AX 中

MOV AX,DS:[DI] ;将段地址为 DS 偏移地址为 DI 的内存单元中的值移入 AX 中

MOV AX,SS:[DI] ;将段地址为 SS 偏移地址为 DI 的内存单元中的值移入 AX 中

**其他寄存器（CS，IP，SS，SP，DS，ES）**

由于段寄存器总是和其他一些像指针寄存器，变址寄存器，控制寄存器一起使用，

所以在这里，我并不会单独介绍段寄存器，而是将段寄存器和一些其他的常用寄存器搭配介绍 。

由于下面的介绍中会涉及到很多关于段和栈的概念，而段和栈的介绍又都必须关系到物理内存，

所以在介绍段寄存器以及其他一些呈协作关系的寄存器之前，还是先来介绍一下这几个基本的概念比较好。

**8086 CPU 访问内存（物理地址）：**

当 CPU 需要访问一个内存单元时，需要给出内存单元的地址，

而每一个内存单元在物理内存空间中都有一个唯一的地址，

即可以通过这个地址定位到内存单元，而这个地址即为物理地址。

CPU 通过地址总线将一个内存单元的物理地址送入存储器，

而后 CPU 便可以通过这个物理地址来访问这个物理地址所指向的内存单元了。

那么这个物理地址在 CPU 中是如何形成的呢？

首先，我们知道 8086 CPU 的地址总线是 20 根，

即每次都可以传输 20 位的地址，从而寻址能力有 220 也就是 1MB 的大小，

但是 8086 CPU 的寄存器只有 16 位，也就是在 8086 CPU 的内部，

一次性处理，传输，暂存的地址都只能是 16 位，

即 8086 CPU 不能完整的保存下一个物理地址（物理地址为 20 位），

如果单单以最简单的方式（即直接用 16 位寄存器来保存物理地址）的话，那么，寻址能力只有 216 ，也就是 64KB，

如果真以如此简单的方式的话，那么地址总线还需要 20 根干嘛呢？而且，难不成我们以后的内存就是 64KB 了吗？

当然不是的，8086 CPU 在这里采取了一定的措施从而使其寻址能力达到 1MB 。

8086 CPU 在内部通过两个 16 位的地址进行合成从而形成一个 20 位的物理地址，由此，8086 CPU 的寻址能力便可以达到 1MB 。

那么 8086 CPU 又是如何将两个 16 位的地址合成为一个20 位的物理地址的呢？

当 CPU 在访问内存时，其会使用一个 16 位的基地址，然后再使用一个 16 位的偏移地址，

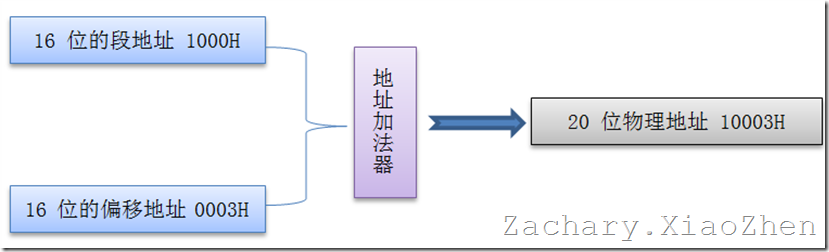
通过将基地址和偏移地址传入 8086  CPU 的地址加法器中进行合成即可以构造出 20 位的物理地址。

至于合成的方式如下：

基地址其实是通过一个 16 位的段地址来形成的，将一个段地址左移 4 位即形成了基地址，

而至于偏移地址的话，自然不必多说，为 16 位，通过将基地址和偏移地址相加便形成了 20 位的物理地址 。

**下面给出一幅示意图来表示物理地址的合成：**

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_24.png)

**段：**

至于段的话，其实在物理内存中是没有段这一概念的，事实上，段的概念来自于  CPU ，

因为 CPU 拥有段寄存器，既然在 CPU 中拥有了段寄存器，自然，在 CPU 中就肯定有段的概念了，

其实段也就是在编程时，我们将若干个地址连续的内存单元看做是一个段，

然后通过将一个段地址左移 4 位形成基地址，再通过这个基地址来定位这个段的起始地址，

然后，再通过偏移地址便可以精确定位到段中的内存单元了，由于段的起始地址是一个段地址左移 4 位，

所以很明显，段的起始地址肯定是 16 的倍数，而且由于一个段内部，只能通过偏移地址来定位，

而偏移地址为 16 位，所以一个段的长度也就是 216 也就是 64KB 的大小。

在编程时，可以讲一段内存定义成为一个段，而这里，我们又可以引出数据段，代码段，栈段这三种类型的段 。

**何为数据段呢？**其实就是我们自个儿定义一段内存（当然段起始地址肯定是 16 的倍数，并且段长度 <= 64KB），

然后我们在这个段里头存放我们所需要使用的数据，这就是数据段；

**何为代码段呢？**其实也很简单，也是我们自己在编程的时候定义一段内存，然后这段内存用来存放我们的代码（也就是指令），

既然是存放的代码，自然就称之为代码段；

**何为栈段呢？**至于栈段的话，有接触过数据结构的朋友应该是很清楚栈的，而这里我们也就是在内存中分配出一个段，

然后将这个段当做栈来使用，对于栈的介绍，详见下文；

这里呢，顺便还点出几个关于段寄存器的内容，当然下文还会详细介绍的，

首先，对于任何一个段来说，均有段地址，而这些段地址是存放在段寄存器中（段寄存器的作用也在于此），

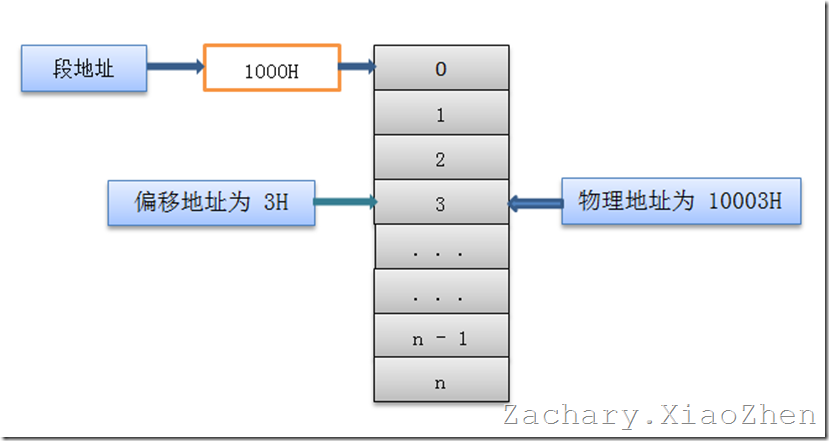
但是对于不同的段，它们默认的段地址存放在不同的段寄存器中，像

数据段来说，它的段地址存放在  DS （Data  Segment）寄存器中，

代码段的段地址存放在  CS （Code  Segment）寄存器中，

栈段的段地址存放在  SS （Stack  Segment）寄存器中 。

**下面给出一幅在段中寻址的示意图：**

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_27.png)

上面的示意图中，通过将段地址左移四位，然后与偏移地址相加便可以得到 20 位的物理地址了 。

**栈：**

8086  CPU 中提供了对栈的支持，并且其还提供了相应的指令来以栈的方式访问内存空间 。

**什么是栈？**

通过上面在段中的介绍，栈其实就是一个段，再说白一点，也就是一块内存，当然，这块内存是一块连续的内存 。

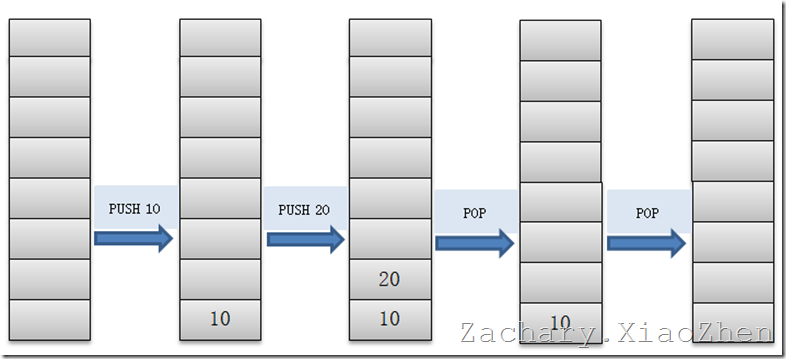
既然栈是一个段的话，那么当然就可以以使用段的方式来使用栈，当然，除了像段一样的使用栈以外，

栈还提供了其特殊的访问方式（如果和段一模一样的话，那还需要栈干吗呢？），

众所周知，栈是先进后出类型的数据结构，在 8086  CPU 中也是如此，

可以通过 **”PUSH“**  指令将数据压入栈中，然后再通过 **”POP“**  指令将栈顶的元素取出来 。

**下面给出一幅示意图来描述栈：**

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_29.png)

即通过 PUSH  10 来将元素 10 放入栈中，因为，先前栈中没有任何数据，所以，10 就会作为栈顶元素存在，

然后再在栈中压入元素 20 ，此时，栈顶中的元素就是 20 了，然后再使用  POP 指令将栈顶元素取出，

此时取出的栈顶元素是 20 ，取出 20 后，栈中便只剩下 10 了，自然 10 就成为了栈顶，

最后再通过 POP 指令将栈顶 10 取出，此时，栈便变成了空栈了 。

**好了，在介绍段寄存器之前的基础知识介绍就到这里了，下面开始正式介绍段寄存器以及与它们协作使用的寄存器。**

**CS 寄存器 和 IP 寄存器：**

经过前面对段的介绍，相信各位朋友对段寄存器应该也有一定的了解了，

下面将要介绍的是一组非常非常重要的寄存器，即 CS:IP 。

CS:IP 两个寄存器指示了 CPU 当前将要读取的指令的地址，其中  CS  为代码段寄存器，而   IP  为指令指针寄存器 。

什么叫做指示了 CPU 当前将要读取的指令呢？在 8086  CPU 中，为什么  CPU  会自动的执行指令呢？

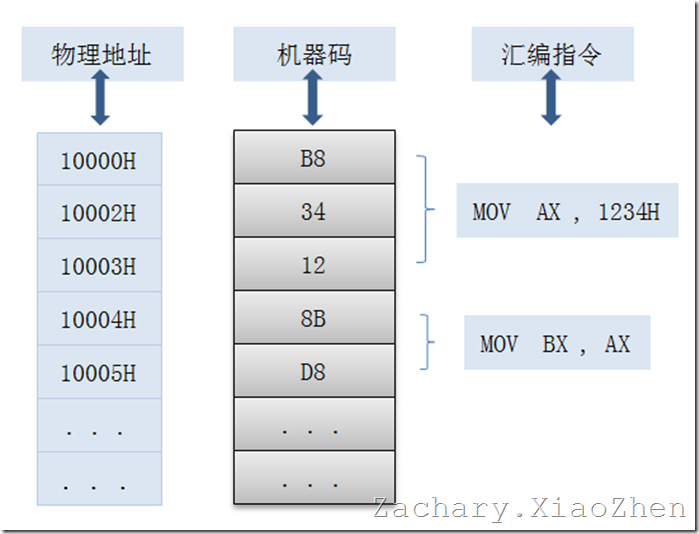
这些指令肯定是存放在内存中的，但是  CPU  怎么知道这些指令存放在内存的那个位置呢？

比如，我有下面的两条指令要执行：

MOV AX,1234H

MOV BX,AX

而假设这两条指令在内存中存放为：

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_33.png)

很显然， 1000H:0000H 指向的是  MOV  AX，1234H  的首地址，

如果 CPU 要读取到我的指令的话，很显然，必须要知道地址  1000H:0000H ，

然后  CPU  就可以根据这个首地址，将汇编指令**MOV  AX，1234H**  所对应的机器码读入到  CPU  的指令寄存器中，

最后便可以在  CPU  中进行处理了。

但关键是   CPU  如何知道我的  1000H:0000H  这个首地址？

其实这就需要使用到  CS:IP  这个寄存器组了 。

当我们运行一个可执行文件时，很明显，我们需要另外一个程序来将这个可执行文件加载到内存当中，

关于这个加载可执行文件的程序，我们在这里不管他，点一下即可，

一般是通过操作系统的外壳程序（也就是传说中的  Shell  程序），

Shell  将可执行文件加载到内存中以后，就会设置  CPU  中的两个寄存器，

即设置  CS:IP  两个寄存器指向可执行文件的起始地址，此后  CPU  便从这个起始地址开始读取内存中的指令，并且执行，

比如我们在写汇编程序时，通常会使用  START  标记，其实这个标记就是用来标记起始地址的，

当将一个汇编程序编译，连接成可执行文件以后，再通过操作系统的  Shell  程序将可执行文件加载到内存中以后，

这个  START  所标记处的地址就是整个可执行文件的起始地址了 。

**也就是说，当一个可执行文件加载到内存中以后，CS:IP  两个寄存器便指向了这个可执行文件的起始地址，**

**然后  CPU  就可以从这个起始地址开始往下读取指令，**

**当读取完指令后，CS:IP  将会自动的改变，基本上是改变  IP ，从而指向下一条要读取的指令，这样就可以执行这个可执行文件了** 。

**最后再对  CS:IP  总结一下：**

1. 你想让  CPU  执行哪行指令，你就让  CS:IP  指向保存有指令的那块内存即可。
2. 任何时候，CS:IP  指向的地址中的内容都是  CPU  当前执行的指令。

**下面我们来看一个  Demo，并详细观察其执行的过程：**

ASSUME CS:CODES

CODES SEGMENT

START:

MOV AX,1234H

MOV BX,AX

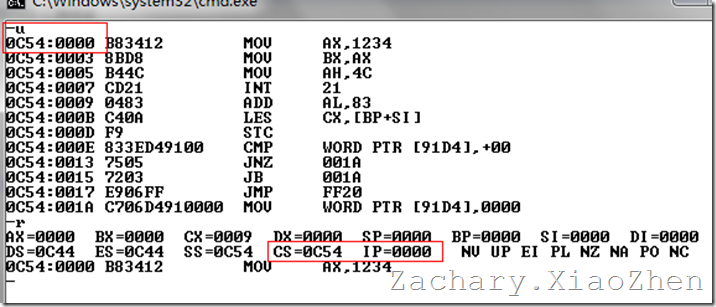
MOV AH,4CH

INT 21H

CODES ENDS

END START

**语句的执行过程如下：**

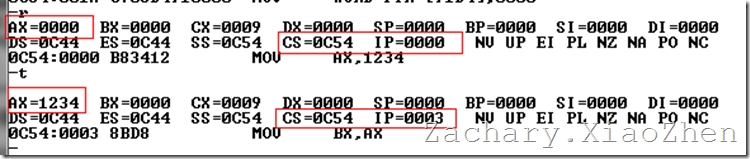
[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_35.png)

从上面的截图中可以看出，当我使用  Shell （在  DOS  下也就是  Command  命令解释器）将可执行文件加载进内存后，

可以看到，整个程序的起始地址为   0C54H : 0000 H  ，并且，可以看到  CS  的地址为  0C54H ，IP  的地址为  0000H，

这正好吻合我们上面对  CS:IP  的分析，很明显，CPU  将会读取 **MOV    AX ，1234H**   到 CPU 中并且执行 ，

然后我们继续向下看：

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_37.png)

可以看到，我们单步执行后，AX 中的值编成了  1234H ，而  IP  寄存器中的值变成了  0003H，

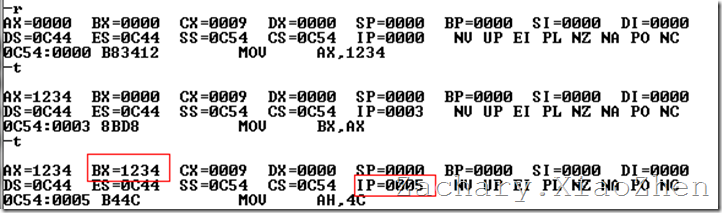
对于  AX  中的值的改变，我们是能够理解的，但是   IP  中的值为什么会从  0000H  变到  0003H  呢？

从最上面的一幅关于指令在内存中的存放可以看出**MOV    AX ，1234H**   在内存中需要  3 个内存单元存放，

也就是  CPU  为了执行**MOV    AX ，1234H**   这条指令，已经将内存中相对应的 3  个内存单元读入内存中了，

执行完这条指令后，自然，CPU  就要将偏移地址向下移动  3  个单元，从而使得  CS:IP  指向下一条需要执行的指令了 ，

为了更深刻的理解，我们再来继续看执行过程，

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_39.png)

从最上面的一幅关于指令在内存中的存放可以看出 **MOV    BX ，AX**  在内存中只占  2  个内存单元，

这也就是为什么  IP  这一次只向下移动了  2  个单元的缘故 。

**关于  CS: IP  的遐想：**

从上面关于  CS:IP  的介绍中，我们可以大胆的猜想，我们只需要通过手动的改变  CS:IP  所指向的内存地址，

让  CS:IP  指向我们另外的代码，那么我们就可以让  CPU  执行我们自己指定的代码了 。

即可以通过修改  CS:IP  来达到我们想要让  CPU  干什么它就干什么的目的 。

上面的虽然是遐想，但是大家要相信，我们写的是汇编，不是  JAVA  也不是  NET  ，

所以我们还真的可以达到上面的目的，也就是说我们的遐想其实是可以实现的，当然这还是有一定的限制的 ，

关于这个遐想呢，可能会在我后续的博文中有所介绍，不过感兴趣的当然可以自己去尝试了，蛮有味的哦 。

**SS 寄存器和 SP 寄存器：**

根据前面对栈的介绍，相信各位对栈也肯定是有一定了解了的，更何况，估计大家也是职场打滚多年的，

要是栈都没用过的话，那也确实蛮悲剧的 ，所以，我在这里也不会对栈做十分详细的介绍了，

但是，最基本的介绍还是要的，毕竟在底层的话，不像高级语言那么方便，可以直接一个  Stack  就 OK 的，

在底层涉及的是栈在内存中的具体实现 。

不知道，大伙有没有注意笔者在本篇博文的上面介绍关于栈的知识时，我并没有提到如何找到这个栈，

我只提到了一个栈就是先进后出操作，同时可以使用  **”PUSH“**和  **”POP“**指令，

然后就是稍微带了一下  SS 这个寄存器的介绍，

我们虽然在内存中是可以方便的定义一个栈了，但是，我们为什么要定义这么一个栈呢？

自然，是为了操作方便，同时提供给  CPU  使用的，

既然  CPU  要使用的话，自然，CPU  又必须根据一定的方式找到这个栈，

而这就需要使用  SS 和  SP 寄存器了 。

同时，一个栈也就是一块内存区域，通过上面的介绍，我们也知道了如果要在一块内存中精确地定位到内存单元的话（寻址），

我们必须要有基地址（也就是段地址左移  4  位）和偏移地址，自然，要在一个栈中寻址的话，也需要段地址和偏移地址，

而对于一个栈来说，我们使用的最多的是什么呢？

当然是栈顶了，因为只有栈顶可以用来存取数据，所以对于一个栈来说，我们只需要有栈顶的段地址和偏移地址即可，

而对于栈顶的段地址，其是存放在段寄存器  SS  中的，而对于栈顶的偏移地址，其则是存放在  SP  寄存器中的 。

**记住，在任何时刻，SS:SP  都是指向栈顶元素 。**

其实关于栈的使用还是比较简单的，但是要注意的是  8086  CPU  并不会保证我们对栈的操作会不会越界 。

所以我们在使用栈的时候需要特别注意栈的越界问题 。

当使用  **PUSH**指令向栈中压入 1 个字节单元时，SP = SP - 1；即栈顶元素会发生变化；

而当使用  **PUSH**指令向栈中压入  2 个字节的字单元时，SP = SP – 2 ；即栈顶元素也要发生变化；

当使用  **POP**指令从栈中弹出 1 个字节单元时， SP = SP + 1；即栈顶元素会发生变化；

当使用  **POP**指令从栈中弹出 2 个字节单元的字单元时， SP = SP + 2 ；即栈顶元素会发生变化；

**下面通过一个  Demo 来介绍栈的使用：**

ASSUME CS:CODES

CODES SEGMENT

START:

MOV AX,1000H ;首先是定义好栈的段地址

MOV SS,AX

MOV AX,10H ;再定义好栈的长度（初始时刻的栈顶偏移地址即栈的长度）

MOV SP,AX

MOV AX,0001H

PUSH AX

MOV AX,0002H

PUSH AX

MOV AX,0003H

PUSH AX

MOV AX,0004H

PUSH AX

MOV AX,0005H

PUSH AX

POP AX

POP AX

POP AX

POP AX

POP AX

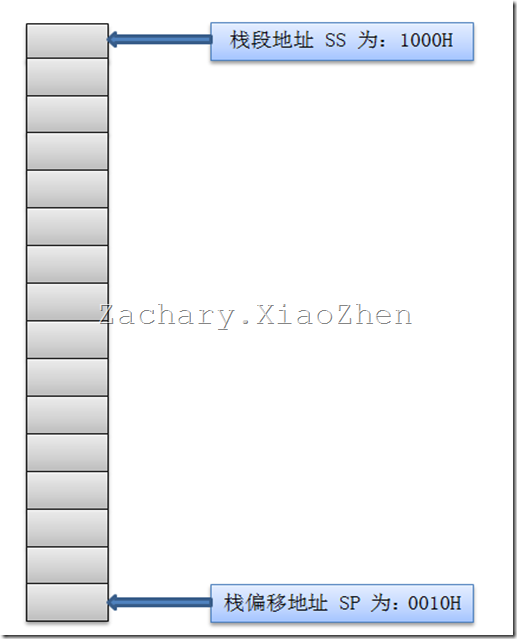
MOV AH,4CH

INT 21H

CODES ENDS

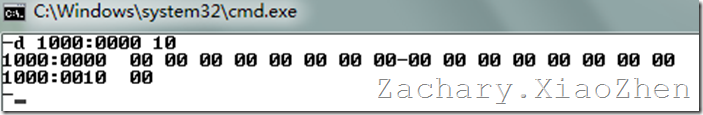
END START

然后我们来看栈在内存中的结构图：

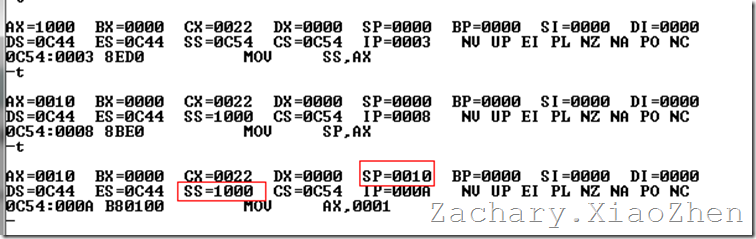
[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image127.png)

**语句的执行过程如下：**

首先我们来看尚未执行上述任何指令时栈中的数据情况：

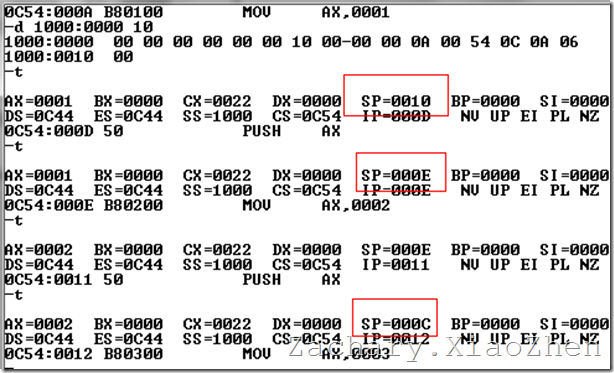
[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_43.png)

然后我们再来依次执行上述指令：

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_45.png)

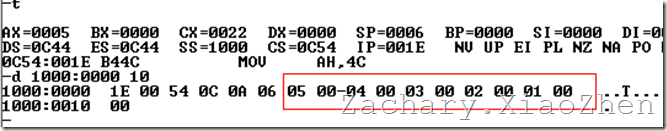
从上副截图中可以看出已经设置好了  SS:SP ，也就是栈已经设置 OK 了，

下面开始往栈中压入数据了，

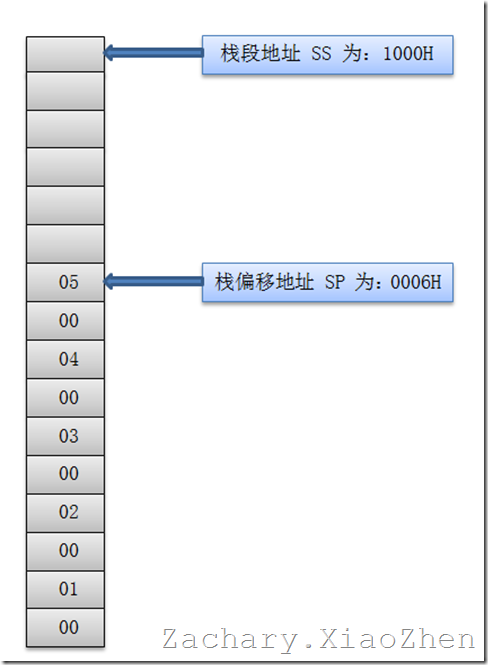
[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_47.png)

由于我们压入栈中的数据为字数据，即占 2 个内存单元，所以，每次  SP = SP – 2 ；

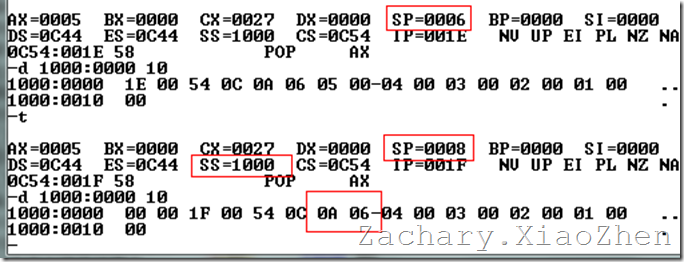
将 5 个字型数据压入栈中后，我们可以来查看栈中的数据了，

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_49.png)

因此，在内存中的一个好看点的结构图如下所示：

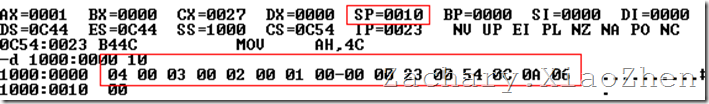
[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_51.png)

下面开始进行出栈操作了

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_55.png)

由于我们弹出栈时的数据为字数据，即占 2 个内存单元，所以，每次  SP = SP + 2 ；

将 5 个字型数据全部弹出栈中后，我们可以来查看栈中的数据了，

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_57.png)

可以看到 SP 变成了初始状态了，也就是说栈中所有的数据已经全部弹出了，虽然我们查看内存时看到的不是 0 ，

但是我们看到的这些数据都是无效的，我们这里不理会 。

**DS 寄存器和 ES 寄存器：**

DS  寄存器和  ES  寄存器都属于段寄存器，其实它们和  CS  寄存器以及  SS  寄存器用起来区别不大，

既然是段寄存器的话，自然它们存放的就是某个段地址了 。

通过上面对基础知识的介绍呢，我们已经知道，如果  CPU  要访问一个内存单元时，

我们必须要提供一个指向这个内存单元的物理地址给  CPU ，

而我们也知道在  8086  CPU  中，物理地址是由段地址左移 4  位，然后加上偏移地址形成的，

所以，我们也就只需要提供段地址和偏移地址即 OK 。

8086  CPU  呢，提供了一个  DS  寄存器，并且通常都是通过这个  DS  段寄存器来存放要访问的数据的段地址 。

DS（Data  Segment）：很显然，DS 中存放的是数据段的段地址 。

但是这里不得不再点一下，那就是我们对段的支持是在  CPU  上体现的，而不是在内存中实现了段，

所以事实上我们使用的段其实是一个逻辑概念，即是我们自己定义的，

再说白了，我定义一个段，我说它是数据段那它就是数据段，我说它是代码段那么它就是代码段，

它们其实都是一块连续的内存而已，至于为什么要区分为数据段和代码段，

很明显，是用来给我们编程提供方便的，即我们在自己的思想上或者说是编码习惯上规定，

数据放数据段中，代码放代码段中 。而我们在使用数据段的时候，为了方便或者说是代码的编写方便起见，

我们一般把数据段的段地址放在  DS  寄存器中，当然，如果你硬要觉得  DS  不顺眼，那你可以换个  ES  也是一样的，

至于  ES（Extra  Segment）  段寄存器的话，自然，是一个附加段寄存器，如果再说得过分点，

就当它是个扩展吧，当你发现，你几个段寄存器不够用的时候，你可以考虑使用   ES  段寄存器，

在使用方式上，则和其他的段寄存器没什么区别  。

**下面看一个介绍使用  DS  寄存器的  Demo：**

ASSUME CS:CODES

CODES SEGMENT

START:

MOV AX,1000H

MOV DS,AX

MOV AL,1

MOV BX,0

MOV CX,5 ;设计一个循环，让其循环 5 次

s: MOV [BX],AL ;这里 [BX] 并没有指定段地址哦

INC AL

INC BX

LOOP s

MOV AH,4CH

INT 21H

CODES ENDS

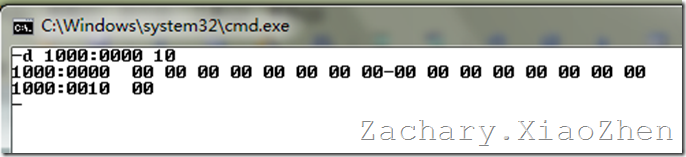
END START

上面的代码所做的事情，就是循环将  1，2，3，4，5 写入到地址  1000H：0000H ，1000H：0001H，

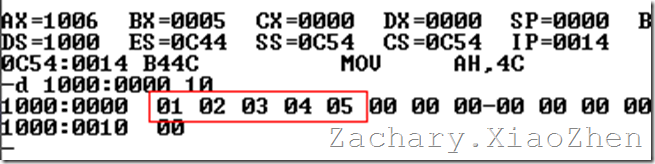
1000H：0002H，1000H：0003H，1000H：0004H  中，

**语句的执行过程如下：**

首先我们来看尚未执行上述任何指令时栈中的数据情况：

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_22.png)

而当循环执行完成以后，我们再来看内存  1000H：0000H 处的值：

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_30.png)

在这里，我们可以看到确实达到了我们预期的效果，但是大家注意看代码：

s: MOV [BX],AL ;这里 [BX] 并没有指定段地址哦

INC AL

INC BX

LOOP s

这里可以看到，我们在  **[BX]**中并没有给其指定段地址，而只有一个偏移地址，

但是根据我们一开始的介绍，必须要有段地址和偏移地址才能够定位内存单元，

莫非这里出问题了？

其实不是的，因为我们在最前面定义了段地址   DS  为  1000H，

当我们定义好段地址后，每一次  CPU  执行到  **[BX]**  时，便会自动或者说是默认的从  DS  中取值，

并且将取得的值作为段地址，因此，当  **[BX]**  为  0001H  时，CPU  会从   DS  中取得一个  1000H ，

由这两个一合成即可以得到正确的物理地址   1000H：0000H 。

最后还提醒一点，那就是   8086  CPU  不支持直接将一个数据送入段寄存器中，

**也就是下面的做法是错误的：**

MOV DS,1000H

**标志寄存器（FLAG）:**

前面呢，已经介绍了  8086  CPU  14 个寄存器中的 13 个了，下面我们将介绍最后一个寄存器也就是  FLAG  寄存器，

FLAG  寄存器之所以放到最后一个介绍，是因为其和其他的一些寄存器不同，像   AX，BX，CX，DX  这些寄存器来说，

它们都是用来存放数据的，当然  FLAG  中存放的也是数据啦，

呵呵，不过，AX，BX 这些寄存器中的数据是作为一个整体使用的，

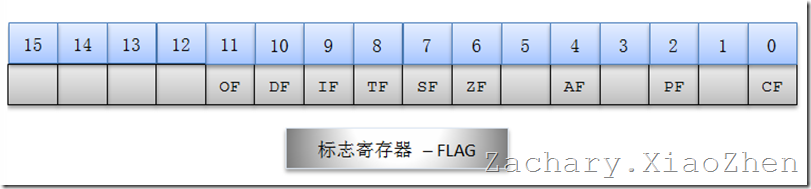
最多也就分成一个  AL  和  AH  使用而已，但是在  FLAG  中，数据是按位起作用的，

也就是说，FLAG  中的每一个位都表示不同的状态，

由于一个位也就能表示  0  和  1 ，自然，FLAG  中的每一个位就是用来描述状态的，

而且  FLAG  寄存器中存储的信息通常又被称作程序状态字（PSW） 。

**下面我给出一幅  FLAG  寄存器中各个位的示意图：**

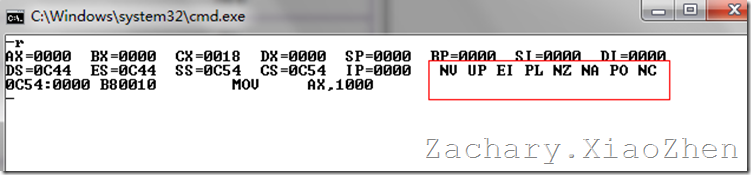
[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_40.png)

从上面这幅图中可以看出，FLAG  的第  0  个位表示的是 CF  ，第 2 个位表示的是  PF  ，与此类推 . . . .

**首先，我们来看一个列表：**

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_60.png)

上面的这个表怎么看呢？我们通过看下面一幅截图就知道了 。

[](http://images.cnblogs.com/cnblogs_com/BoyXiao/Windows-Live-Writer/80x86--CPU-_11929/image_62.png)

从上面的标记中可以看出，从左到右依次代表   OF，DF，SF，ZF，PF，CF  标志位的值，

再通过与上面的表格相对照可以知道：

**OF = 0 ；**

**DF = 0 ；**

**SF = 0 ；**

**ZF = 0 ；**

**PF = 0 ；**

**CF = 0  ;**

至于为什么我们在  Debug  模式下，使用  R  命令时，只会列出这几个标志位，我菜的话是因为相对来说，

列出的这几个标志位更为常用，其他的几个标志位并不经常使用的缘故吧 。

**下面我们就按不同的位来分别介绍这些位所描述的状态，以及它们代表的意义：**

**CF（Carry  FLag） - 进位标志（第 0 位）：**

**CF：    进位标志是用来反映计算时是否产生了由低位向高位的进位，或者产生了从高位到低位的借位 。**

if(运算过程中产生了进位或者借位)

{

CF = 1;

}

else

{

CF = 0;

}

**PF（Parity  FLag） - 奇偶标志（第 2 位）：**

**PF：    奇偶标志是用来记录相关指令执行后，其结果的所有的  Bit  位中  1  的个数是否为偶数 。**

if(运算结果中 1 的个数为偶数)

{

PF = 1;

}

else

{

PF = 0;

}

**AF（Auxiliary  Carry  FLag） - 辅助进位标志（第 4 位）：**

**AF：    用来辅助进位标志 。**

if(字节操作中发生低半个字节向高半个字节借位或者进位 || 字操作中发生低字节向高字节借位或者进位)

{

AF = 1;

}

else

{

AF = 0;

}

**ZF（Zero  FLag） – 零标志（第 6 位）：**

**ZF：    记录的是相关的指令执行完毕后，其执行的结果是否为  0 。**

if(执行的结果 == 0)

{

ZF = 1;

}

else

{

ZF = 0;

}

**SF（Sign  FLag） - 符号标志（第 7 位）：**

**SF：    符号标志，其记录相关指令执行完以后，其结果是否为负数 。**

if(运算结果为负数)

{

SF = 1;

}

else

{

SF = 0;

}

**TF（Trap  FLag） - 追踪标志（第 8 位）：**

**TF：    追踪标志，主要是用于调试时使用 。**

if(TF == 1)

{

CPU 进入单步方式;

}

**IF（Interrupt-Enable  FLag） - 中断允许标志（第 9 位）：**

**IF：    中断允许标志，其决定  CPU  是否能够响应外部可屏蔽中断请求（以后会做详细介绍） 。**

if(IF == 1)

{

CPU 能够响应外部的可屏蔽中断请求;

}

else

{

CPU 不能够响应外部的可屏蔽中断请求;

}

**DF（Direction  FLag） - 方向标志（第 10 位）：**

**DF：    方向标志，其用于在串处理指令中，用来控制每次操作后  SI  和  DI  是自增还是自减 。**

if(DF == 0)

{

SI++;

DI++;

}

else

{

SI--;

DI--;

}

**OF（OverFlow  FLag） - 溢出标志（第 11 位）：**

**OF：    溢出标志，其通常记录了有符号数运算的结果是否发生了溢出 。**

if(运算发生溢出)

{

OF = 1;

}

else

{

OF = 0;

}

**总结**

上面呢，从最简单的开始，循序渐进的介绍了  8086  CPU  中的各个寄存器，

同时也通过一些  Demo  来列举了各个寄存器的使用，

由于写的比较基础，而且量也比较多，所以，造成博文过长了，读者需一定耐心才能看完，

写本篇博文呢，并不是说将来要用汇编去开发个什么东东，

实质上，笔者学习汇编的目的也不在此，只是因为先前在接触到底层的寄存器以及内存时，

笔者总有一丝不爽的感觉，总是感觉不得要领，所以才会开始汇编的学习，

此次推出本系列博文，本意也并不是说要学习汇编做开发，只是为了提升内功而已 。