

**信息科学与工程学院**

**2020－2021学年第二学期**

实 验 报 告

课程名称： 微处理器原理与应用

实验名称： 掌握DEBUG 基本命令及其功能

专 业 班 级 通信工程 二班

学 生 学 号 201922121209

学 生 姓 名 陈泽宇

实 验 时 间 2020年3月2日

实验报告

【实验目的】

掌握DEBUG 的基本命令及其功能

【实验要求】

1. 首先截屏显示你在Window下调试DEBUG的步骤。（WinXP可以直接使用，其他注明如何软件实现）
2. 在DEBUG状态下编写、运行程序的过程以及调试所中，对实验任务中的内容逐一调试，并对此过程中的问题进行分析，对执行结果进行分析。

【实验具体内容】

1. 查看CPU 和内存，用机器指令和汇编指令编程
2. 进一步用机器指令和汇编指令编程

【上篇：查看CPU 和内存，用机器指令和汇编指令编程】

【第一个实验】

使用Debug，将下面的程序段写入内存，逐条执行，观察每条指令执行后，CPU中相关寄存器中内容的变化。（逐条执行，每条指令执行结果截图）**可用E命令和A命令以两种方式将指令写入内存。注意用T命令执行时，CS:IP的指向。**

（1）需要用到的指令集合

机器码         汇编指令

**b8 20 4e**      m**ov ax,4E20H**

**05 16 14       add ax,1416H**

**bb 00 20       mov bx,2000H**

**01 d8          add ax,bx**

**89 c3          mov bx,ax**

**01 d8          add ax,bx**

**b8 1a 00       mov ax,001AH**

**bb 26 00       mov bx,0026H**

**00 d8          add al,bl**

**00 dc          add ah,bl**

**00 c7          add bh,al**

**b4 00          mov ah,0**

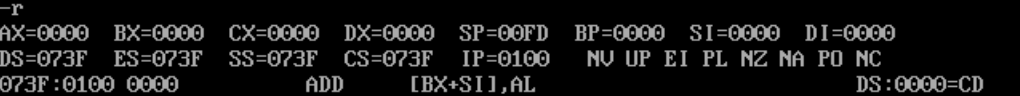
**00 d8          add al,bl**

**04 9c          add al,9CH**

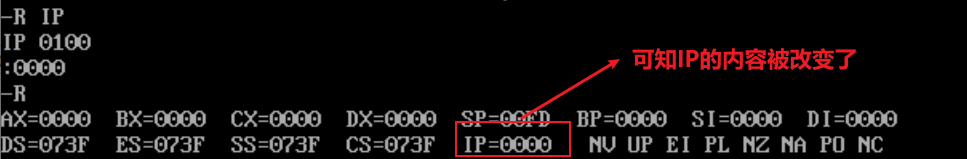
（2）实验过程、相应结果（截图）并对实验进行说明和分析：

**使用-E进行命令写入**

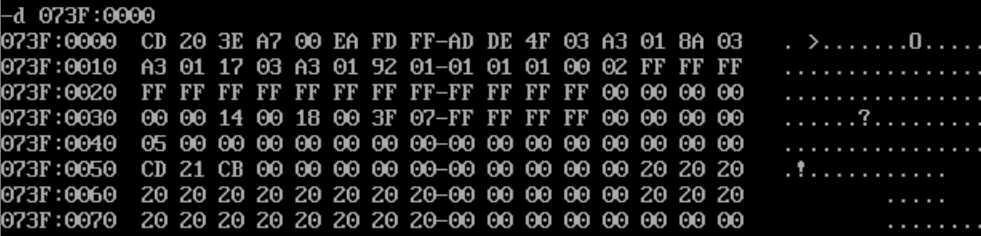
首先打开DosBox，完成初始化后首先查看内存情况



可知CS指向的是073F地址，IP指向0100，为便于说明，先初始化IP为0000，使用-r即可实现，如下图所示



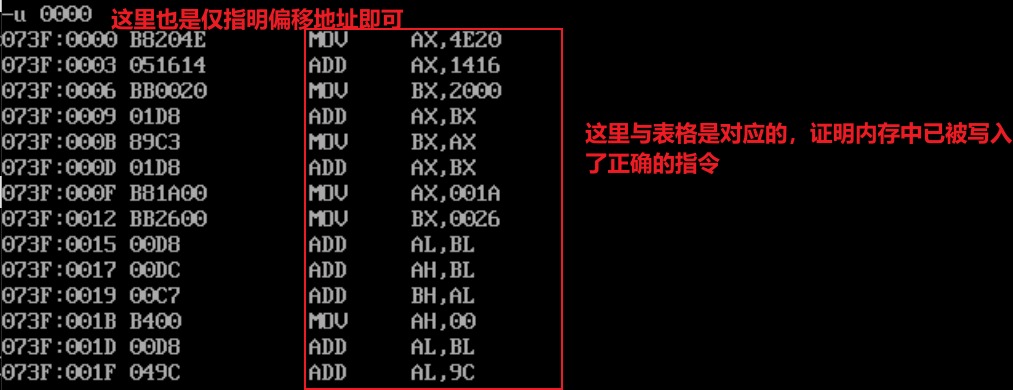
随后再改写073F:0000处的内存，向其中写入机器指令使用-d命令进行查看，未进行操作时，073F:0000及以后的内容如下所示



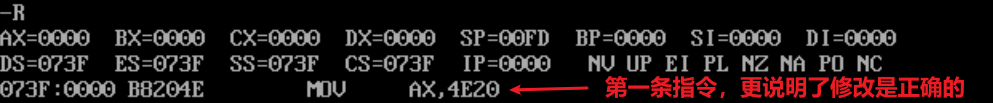
使用-e命令进行机器码的写入，并-d查看结果，如下图所示



这里为提高查看效率，利用-u反汇编指令进行查看



证明成功写入后-r查看内存

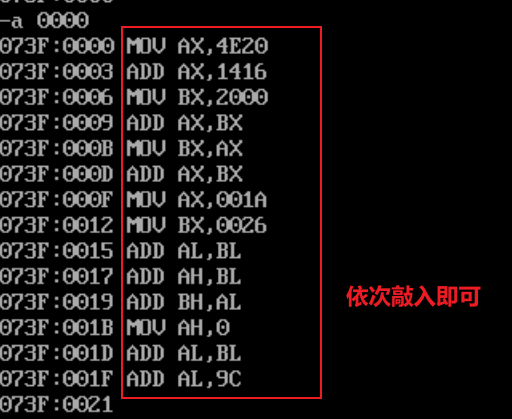


**使用-a命令进行写入**

使用该方式进行写入时操作相对简单，如下图所示，直接键入-a，回车执行后会出现键入命令的提示，未加参数时，这里是在073F:0100处进行命令的写入，如果要选择写入的位置，还是像之前的-e/-u，加入偏移地址即可，如下图所示



重复操作写入命令，如下图所示，0021处回车结束即可



综上，使用-a和-e都可以实现汇编指令的写入，下面开始使用-t命令进行汇编指令的执行。

这里为便于操作，直接采用asm整体文件写入debug的方式进行操作，代码和注释如下

DATA *SEGMENT*

DATA *ENDS*

STACKS *SEGMENT*

STACKS *ENDS*

CODES *SEGMENT*

          ASSUME CS:CODES, DS:DATA, SS:STACKS

    START:

          mov    ax,4E20H                   ;ax=4E20

          add    ax,1416H                   ;ax=ax+1416H

          mov    bx,2000H                   ;bx=2000H

          add    ax,bx                      ;ax=ax+bx

          mov    bx,ax                      ;bx=ax

          add    ax,bx                      ;ax=bx

          mov    ax,001AH                   ;ax=001AH

          mov    bx,0026H                   ;bx=0026H

          add    al,bl                      ;al+=bl

          add    ah,bl                      ;ah+=bl

          add    bh,al                      ;bh+=al

;高/低8位寄存器的相关加法运算

          mov    ah,0                       ;ah=0

          add    al,bl                      ;al+=bl

          add    al,9CH                     ;al+=9C

          ;程序结束

          MOV    AH, 4CH

          INT    21H

CODES *ENDS*

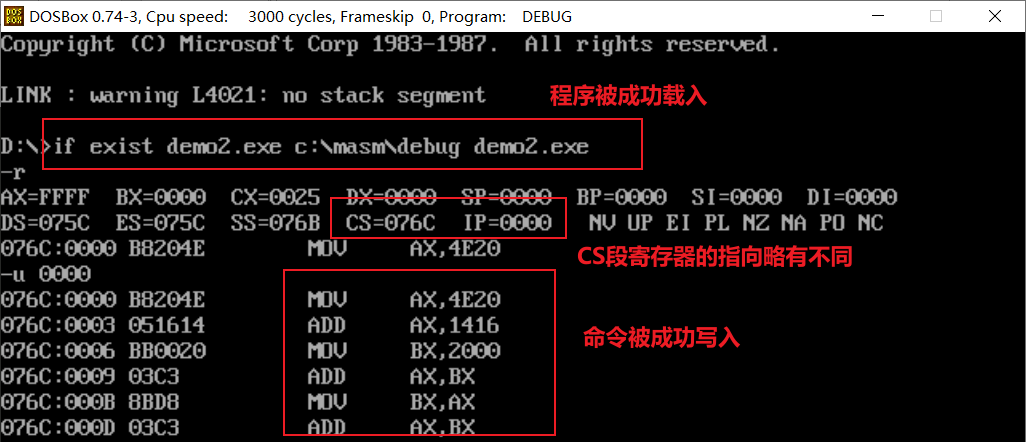
END START

然后直接右键debug即可

（VSCode的用法和使用已在实验1.1中介绍过，这里不再赘述）

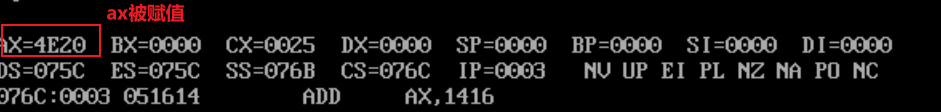


调试的结果如下图

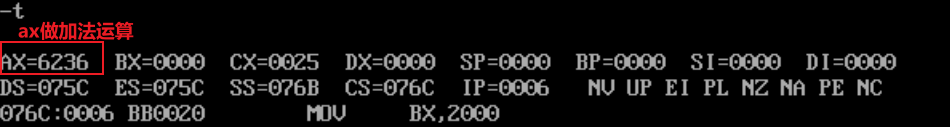


-t依次执行即可，特别需要指出的是，-t最后的汇编指令是**即将执行的指令，而非已经执行的指令。为便于说明问题，每条指令的代码和执行后状态相配套**

mov    ax,4E20H                   ;ax=4E20

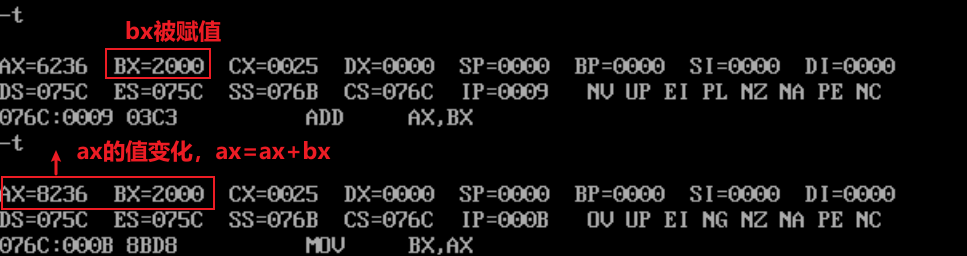


add    ax,1416H                   ;ax=ax+1416H



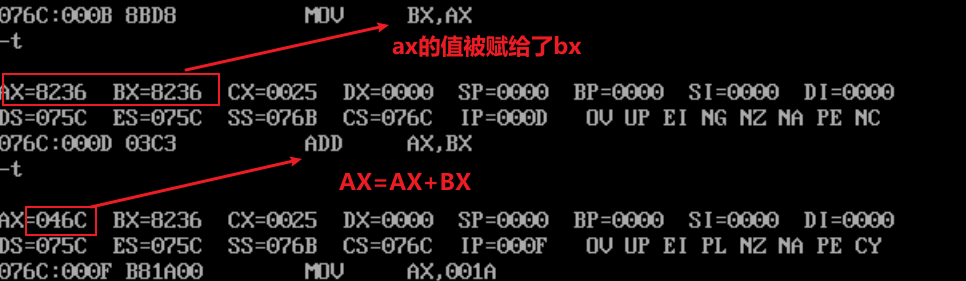
mov    bx,2000H                   ;bx=2000H

add    ax,bx                      ;ax=ax+bx



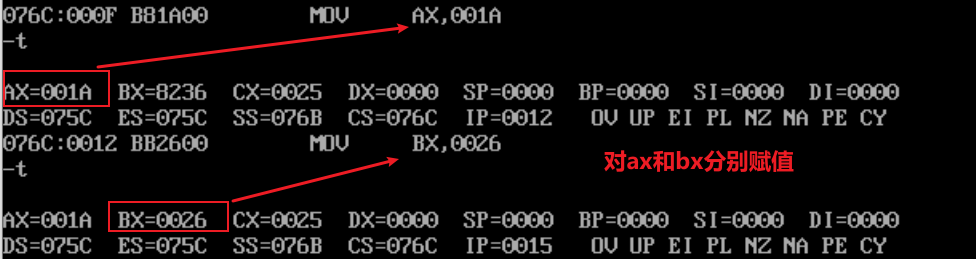
mov    bx,ax                      ;bx=ax

add    ax,bx                      ;ax=bx



mov    ax,001AH                   ;ax=001AH

mov    bx,0026H                   ;bx=0026H

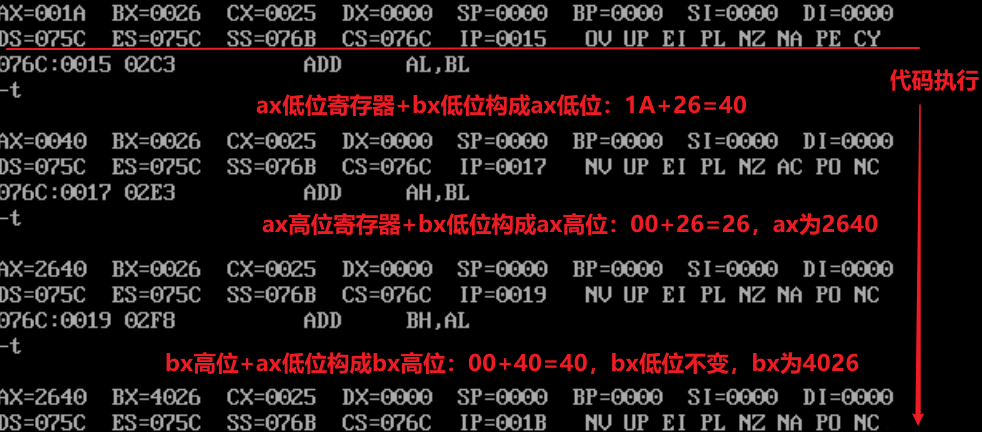


add    al,bl                      ;al+=bl

add    ah,bl                      ;ah+=bl

add    bh,al                      ;bh+=al

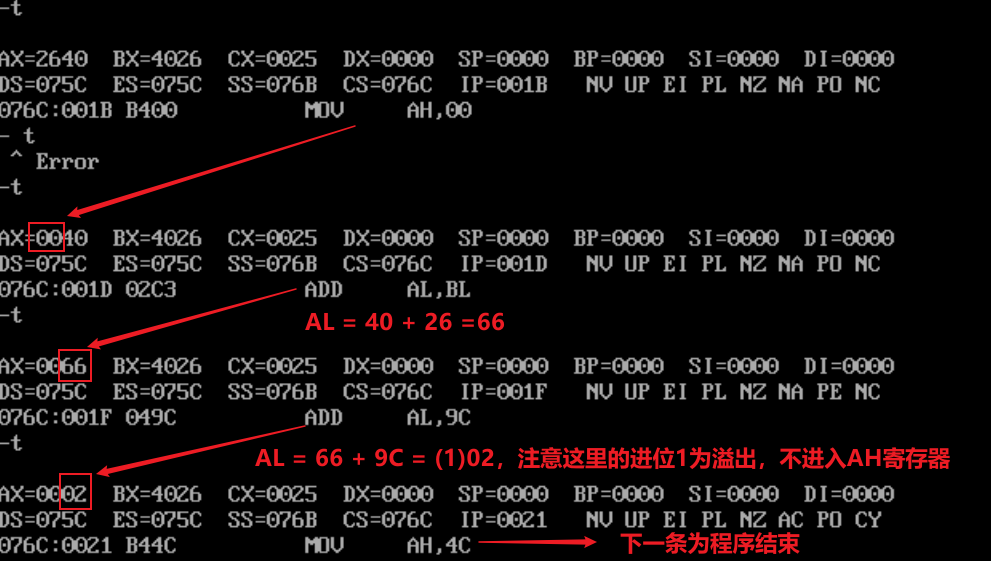
;高/低8位寄存器的相关加法运算



mov    ah,0                       ;ah=0

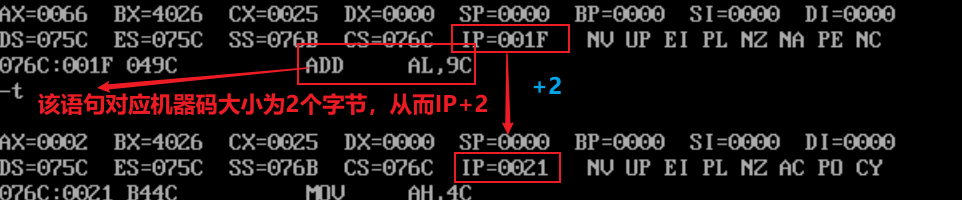
add    al,bl                      ;al+=bl

add    al,9CH                     ;al+=9C



至此程序执行完毕。

回顾整个程序执行的过程，每条指令执行完毕后**都会有IP的变化，并且IP指令的增加量和每条汇编指令的长度是相关的，举例说明如下图所示**



【第二个实验】

将下面3条指令写入从2000:0开始的内存单元中，利用这3条指令计算2的8次方

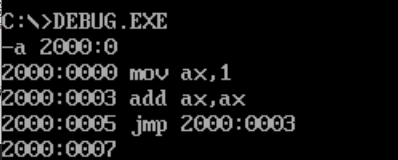
（1）需要用到的指令集合

mov ax,1

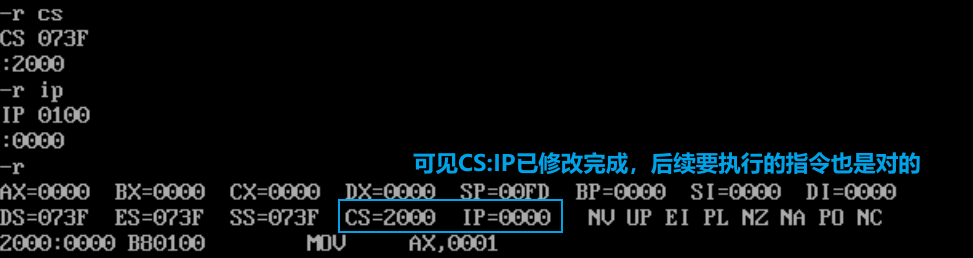
add ax,ax

jmp 2000:0003

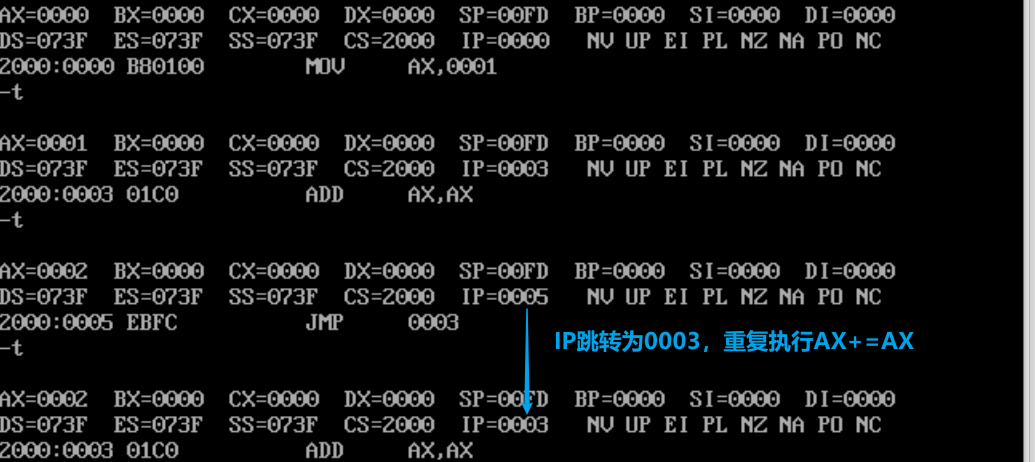
首先进入DEBUG.EXE，利用a命令写入



再使用-r命令进行CS:IP的修改，以在后续-t执行

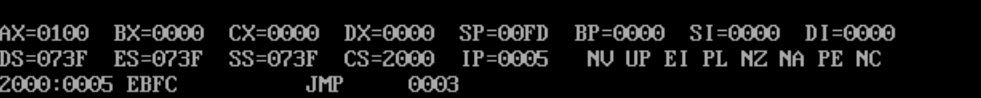


其中汇编指令jmp可以修改CS:IP，在该程序中实现类似循环的功能

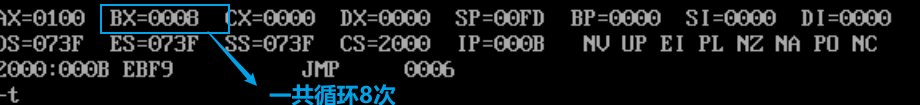


根据上述运行逻辑，AX在第一次跳转后再进行运算就是自乘2的效果，多次运行即可实现28的运算。

循环运算直至AX=0100后即可，此时换算成十进制即为16^2=2^8=256



**考虑如果要计算循环次数**，可以考虑仿照C/C++的思路以BX作为一个计数器，编辑完成代码和循环运算的结果如下

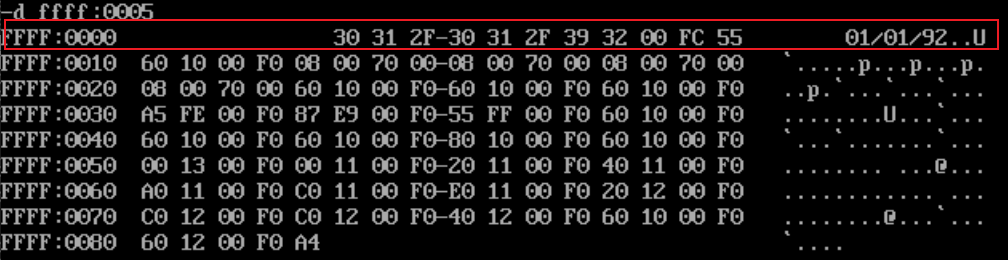


【第三个实验】 查看内存中的内容

PC机主板上的ROM中写有一个生产日期，在内存FFF00H~FFFFFH的某几个单元中，请找到这个生产日期并试图改变它。（内存ffff:0005~ffff:000C(共8个字节单元中)处）

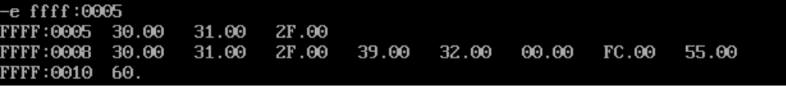
如果使用DOSBOX虚拟环境，这个显示是不正确的，想想为什么？

使用-d命令访问ffff:0005即可

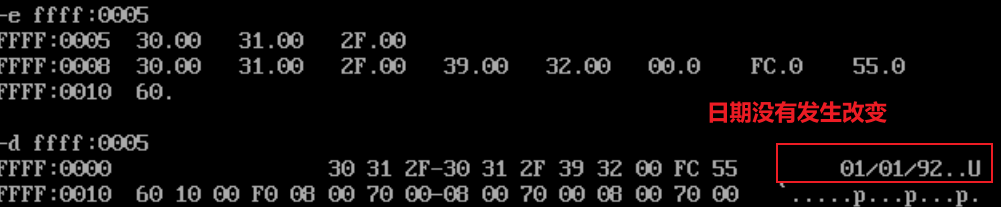


如图所示，日期显示为01/01/92

尝试修改该日期，使用-e命令修改，到FFFF:0010结束



再次-d查看，如下图所示，日期没有更改成功



修改日期失败，**是因为PC机主板上的ROM内容是只读的，因而不能通过该方式进行修改**，即使修改了也会自动恢复。

此外，如果使用DOSBOX虚拟环境，这个显示是不正确的，因为**DOSBOX相当于在系统内建立了一个虚拟机，即自行构建了一个虚拟的DOS操作系统**，所以显示的各个系统信息都是预设好的，和实际PC的主板ROM生产日期必然不会对应。

【下篇：用机器指令和汇编指令编程】

【第一个实验】

使用Debug，将上面的程序段写入内存，逐条执行，根据指令执行后的实际运行情况填空。（逐条执行，每条指令执行结果截图）

（1）需要用到的指令集合

mov ax,ffff

mov ds,ax

mov ax,2200

mov ss,ax

mov sp,0100

mov ax,[0] ;ax=

add ax,[2] ;ax=

mov bx,[4] ;bx=

add bx,[6] ;bx=

push ax ;sp= ;修改的内存单元的地址是 内容为

push bx ;sp= ;修改的内存单元的地址是 内容为

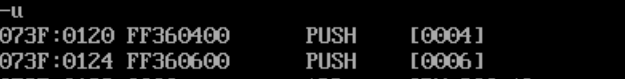
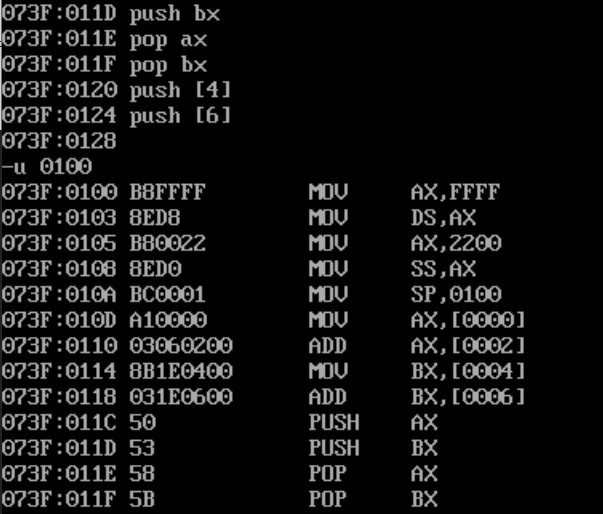
pop ax ;sp= ;ax=

pop bx ;sp= ;bx=

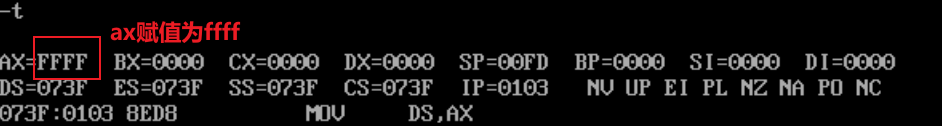
push [4] ;sp= ;修改的内存单元的地址是 内容为

push [6] ;sp= ;修改的内存单元的地址是 内容为

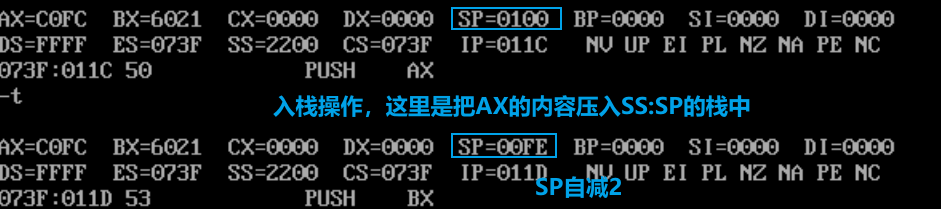
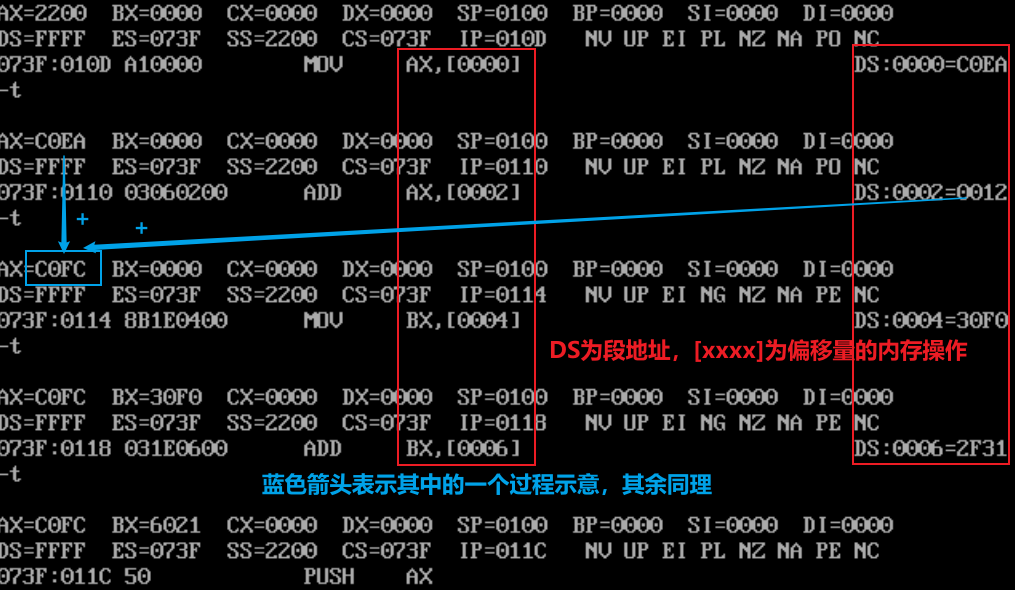
使用Debug -a命令进行写入，-u查看，如下图所示



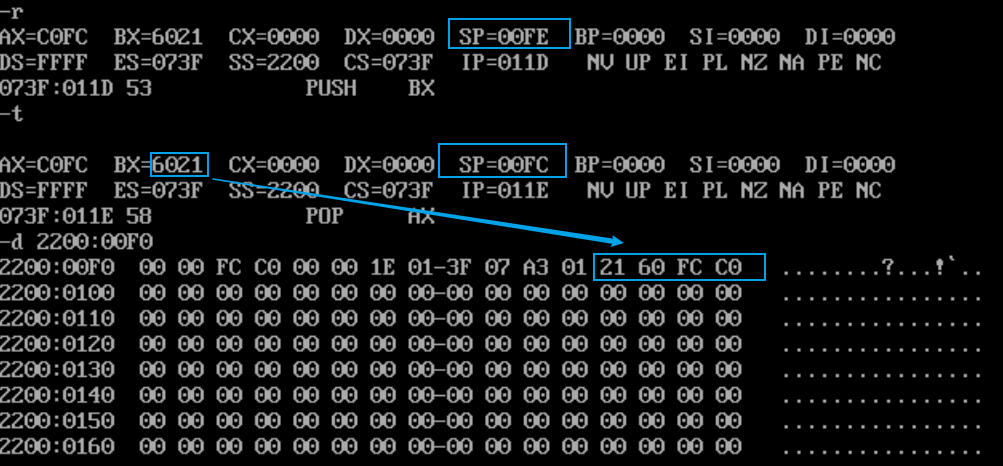
-t命令进行逐行的命令执行，如下图所示



这里SS段寄存器赋值后，在2200:0000处开辟了堆栈区。



两部命令执行完成后，利用-d命令查看SS:SP的内容

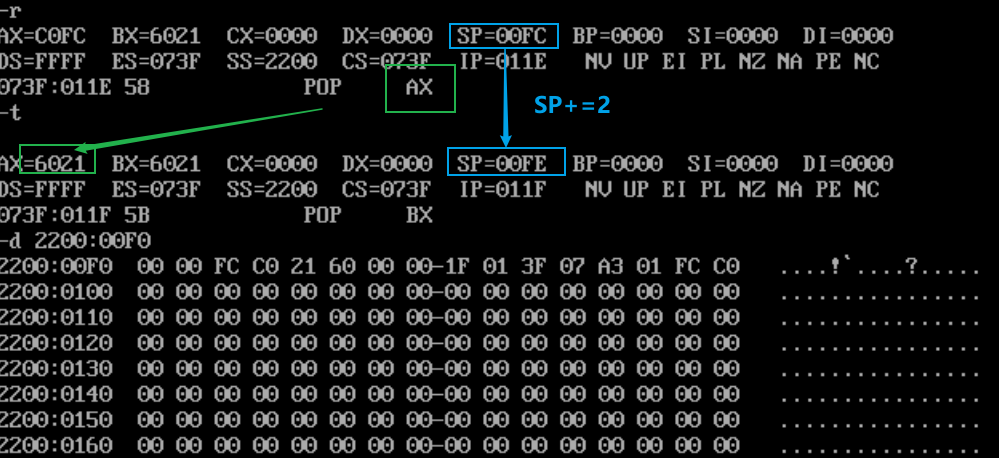


可见两个寄存器中的内容已经存入栈中，从实验结果来说，push的作用可总结如下，从结果中来看，实现了

1. SP自减2，SS:SP指向当前栈顶前边的单元，
2. 将寄存器中的内容送入SS:SP指向的内存单元处

这与汇编语言中的描述是一致的，同时从顺序来看，也是遵循了先入栈元素在栈最下方的规律。

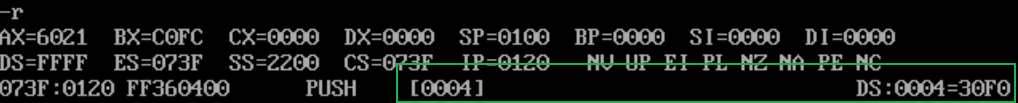
再次执行后续指令，如下图所示



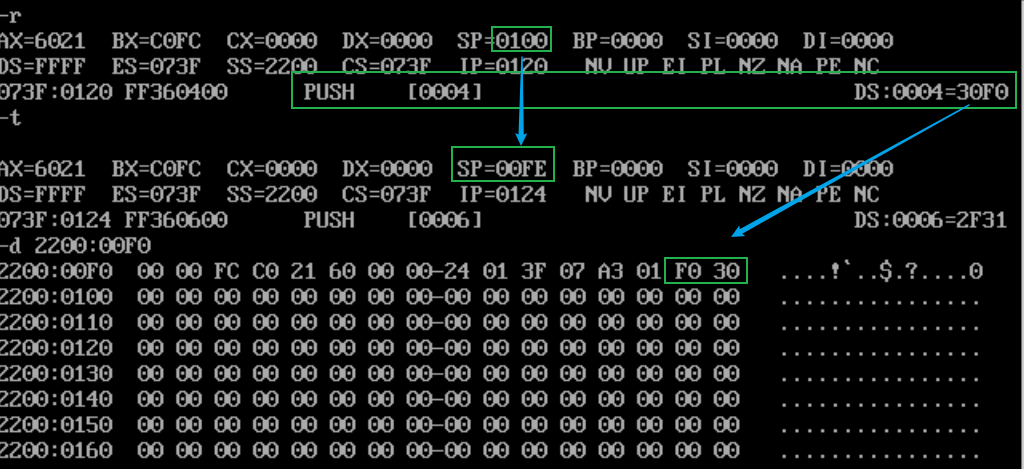
PUSH BX执行完成后，栈顶指向了2200:00FC，这是BX寄存器中的内容，从实验现象中可以得出这里将SS:SP指向的内存单元数据送入了AX中，同时实现了SP自加2，SS:SP指向当前栈顶下面的单元。这也与汇编语言中的描述相符，同时也与后入栈元素先出栈的规律相符合。



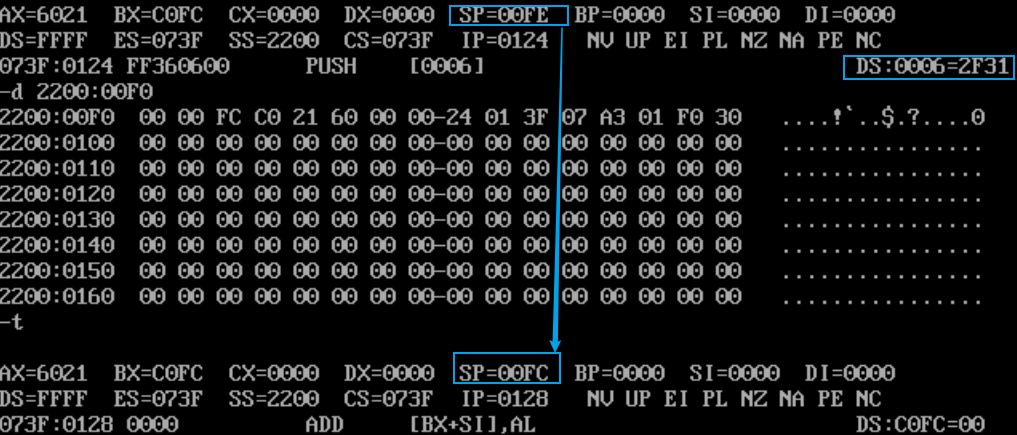
下面将30F0内容进行入栈，此时SP=0100



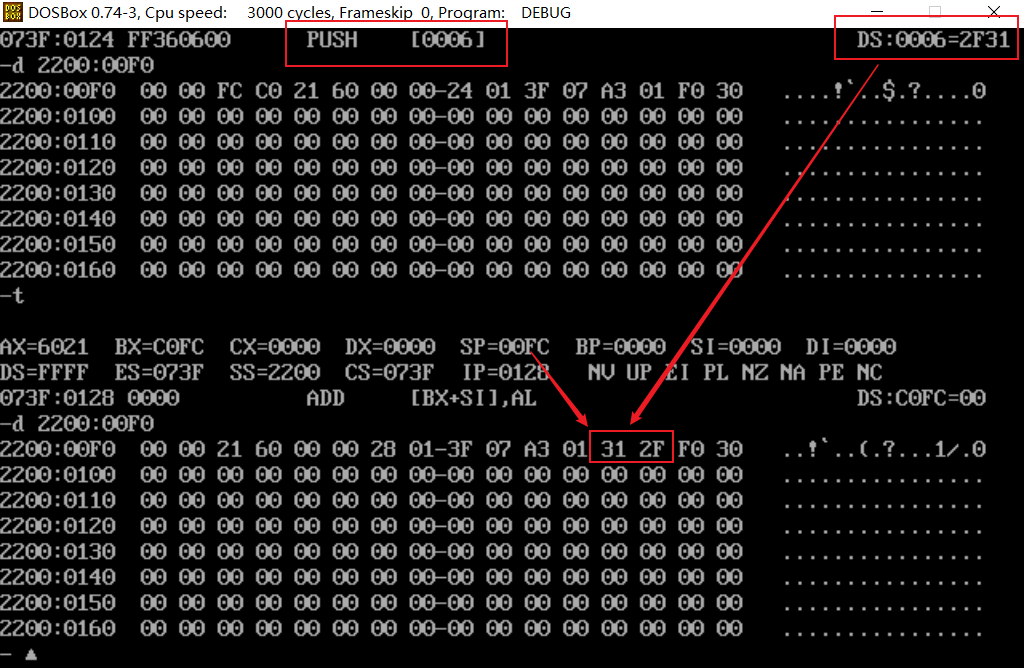
执行后如下图所示



这里F030实现了入栈，并对原来的栈区内存进行了覆盖。



执行PUSH [0006]如下图所示，与前面的命令类似。



总结程序如下：其中注释的数值表示为指令执行结束后的寄存器内容，“内容”为修改之后的内容

mov ax,ffff

mov ds,ax

mov ax,2200

mov ss,ax

mov sp,0100

mov ax,[0] ;ax=C0EA

add ax,[2] ;ax=C0FC

mov bx,[4] ;bx=30F0

add bx,[6] ;bx=6021

push ax ;sp=00FE;修改的内存单元的地址是2200:00FE内容为C0FC

push bx ;sp=00FC;修改的内存单元的地址是2200:00FC内容为6021

pop ax ;sp=00FE ;ax=6021

pop bx ;sp=0100 ;bx=C0FC

push [4] ;sp=00FE ;修改的内存单元的地址是2200:00FE 内容为30F0

push [6] ;sp=00FC ;修改的内存单元的地址是2200:00FC 内容为2F31

【第二个实验】

使用Debug，将下面的程序段写入内存，逐条执行，观察每条指令执行后，CPU中相关寄存器中内容的变化。（逐条执行，每条指令执行结果截图）如果有问题请说明原因

（1）需要用到的指令集合

mov ax,1000H

mov ds,ax

mov ds,[0]

add ds,ax

利用Debug的-a命令进行命令的写入，如下图所示



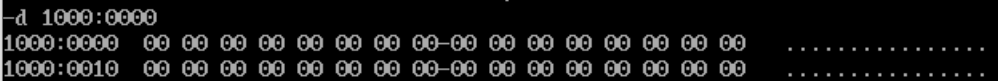
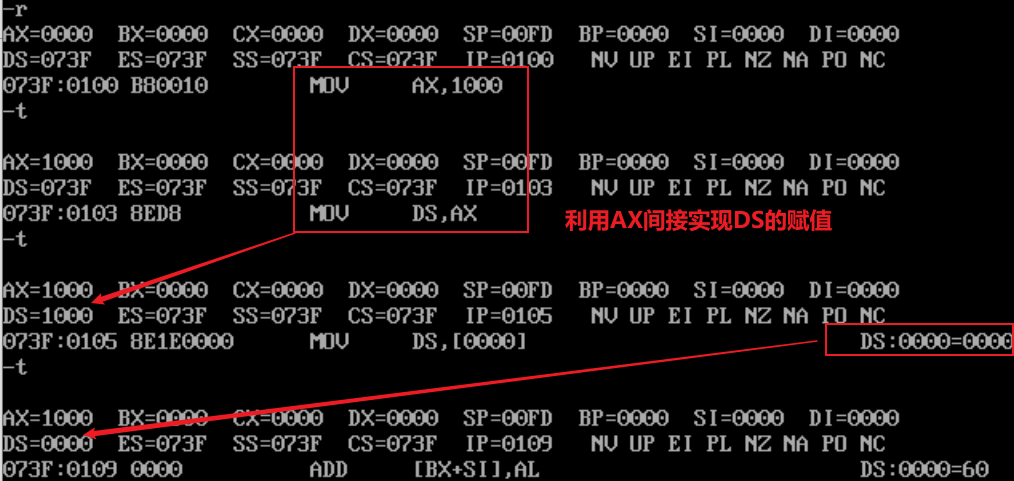
（一般来说当需要进行add的累加运算时，一般用通用寄存器进行计算，然后再用mov语句，把数值放到段寄存器中）

只有下面三条指令是可以执行的

mov ax,1000H

mov ds,ax

mov ds,[0]



（DS:0000处的内容如上图所示）

【第三个实验】

仔细观察下图中的实验过程，然后分析：为什么2000:0~2000:f中的内容会发生改变？



原因分析如下：

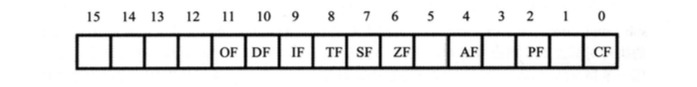
经过查阅资料，在自己设定SS寄存器的内容并新建立栈区后，系统会将原来的CS、IP、Flag（标志寄存器）以及其他数据寄存器中的中间数据等内容暂存到新开辟的栈区中进行保存，防止数据的丢失。

CPU内部的寄存器中，有一种特殊的寄存器具有以下三种作用。

1. 用来存储相关指令的某些执行结果
2. 用来为CPU执行相关指令提供行为依据
3. 用来控制CPU的相关工作方式

这种特殊的寄存器在8086CPU中被称为标志寄存器（flag）。8086CPU的标志寄存器有16位，其中存储的信息通常被称为程序状态字(PSW)。

其中Flag的标志位和显示对应如下所示



而Flag寄存器常用对应值如下所示

OF <-----> OV(1) ; NV(0)

DF <-----> DN(1) ; UP(0)

IF <-----> EI(1) ; DI(0)

SF <-----> NG(1) ; PL(0)

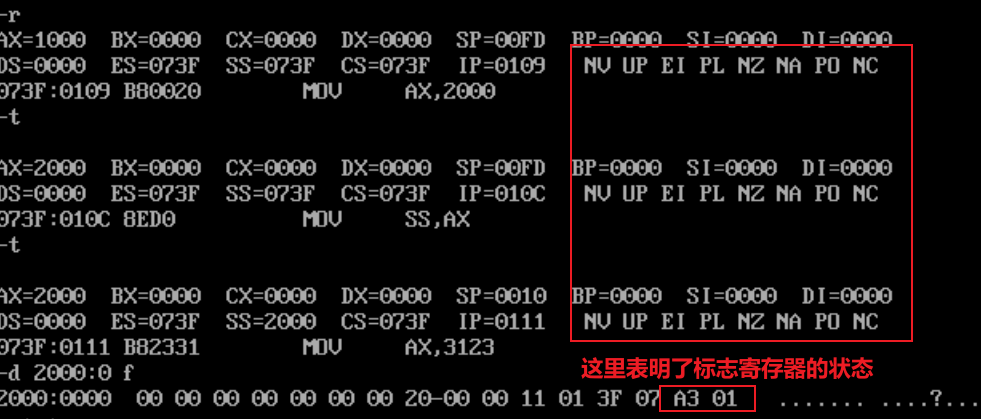
ZF <-----> ZR(1) ; NZ(0)

AF <-----> AC(1) ; NA(0)

PF <-----> PE(1) ; PO(0)

CF <-----> CY(1) ; NC(0)

在Debug中，flag的内容如下图所示



从而可以推测出A3 01表示状态寄存器Flag的值。而其他内容分别表示CS、IP和已经存入数据的AX寄存器

【补充问题】

1. **在使用32位Windows系统时，为什么只能识别4G内存**？

**32位X86架构是指个人电脑的地址总线是32位的，CPU、内存控制器、操作系统都是按32位地址总线设计。**32位地址总线可以支持的内存地址代码是 4096MB，也就是有4GB的地址代码，可以编4GB个地址。这4GB个地址码正好可以分配给4GB内存。

**但是，这4GB个地址码不能全部分配给安装在主板上的物理内存。**因为个人电脑还有很多设备需要地址代码，以便CPU可以根据地址码找到它们，同时CPU和这些设备交换数据需要暂时存放数据的存储器 ——寄存器，这些寄存器也需要地址代码。比如硬盘控制器、软驱控制器、管理插在PCI槽上的 PCI卡的PCI总线控制器，PCI-E总线控制器和PCI-E显卡，它们都有寄存器都需要系统分配给它们地址代码。这些地址由系统分配，电脑用户在使用中感觉不到。这样一来，当我们为电脑插上总容量为4GB的内存时，就有一部分内存分配不到地址代码而不能使用。

1. **使用64位Windows系统时，内存的最大值是多少？**

32位操作系统支持的内存是232 bit，也就是4GB内存。而64位操作系统理论上的寻址空间为264 bit，转化单位为2,147,483,648GB

1. **L1、L2、L3高速缓存（Cache）的相关概念**

L1 Cache(一级缓存)是CPU第一层高速缓存，分为数据缓存和指令缓存。内置的L1高速缓存的容量和结构对CPU的性能影响较大，不过高速缓冲存储器均由静态RAM组成，结构较复杂，在CPU管芯面积不能太大的情况下，L1级高速缓存的容量不可能做得太大。一般服务器CPU的L1缓存的容量通常在32—256KB。

　　L2 Cache(二级缓存)是CPU的第二层高速缓存，分内部和外部两种芯片。内部的芯片二级缓存运行速度与主频相同，而外部的二级缓存则只有主频的一半。L2高速缓存容量也会影响CPU的性能，原则是越大越好，现在家庭用CPU容量最大的是512KB，而服务器和工作站上用CPU的L2高速缓存更高达256-1MB，有的高达2MB或者3MB。

　　L3 Cache(三级缓存)，分为两种，早期的是外置，现在的都是内置的。而它的实际作用即是，L3缓存的应用可以进一步降低内存延迟，同时提升大数据量计算时处理器的性能。降低内存延迟和提升大数据量计算能力对游戏都很有帮助。而在服务器领域增加L3缓存在性能方面仍然有显著的提升。比方具有较大L3缓存的配置利用物理内存会更有效，故它比较慢的磁盘I/O子系统可以处理更多的数据请求。具有较大L3缓存的处理器提供更有效的文件系统缓存行为及较短消息和处理器队列长度。

　　其实最早的L3缓存被应用在AMD发布的K6-III处理器上，当时的L3缓存受限于制造工艺，并没有被集成进芯片内部，而是集成在主板上。在只能够和系统总线频率同步的L3缓存同主内存其实差不了多少。后来使用L3缓存的是英特尔为服务器市场所推出的Itanium处理器。接着就是P4EE和至强MP。Intel还打算推出一款9MB L3缓存的Itanium2处理器，和以后24MB L3缓存的双核心Itanium2处理器。

　　但基本上L3缓存对处理器的性能提高显得不是很重要，比方配备1MB L3缓存的Xeon MP处理器却仍然不是Opteron的对手，由此可见前端总线的增加，要比缓存增加带来更有效的性能提升。

1. **HT技术（超线程）的相关介绍？**

超线程（HT, Hyper-Threading）是英特尔研发的一种技术，于2002年发布。超线程技术原先只应用于Xeon 处理器中，当时称为“Super-Threading”。之后陆续应用在Pentium 4 HT中。早期代号为Jackson。

通过此技术，英特尔实现在一个实体CPU中，提供两个逻辑线程。之后的Pentium D纵使不支持超线程技术，但就集成了两个实体核心，所以仍会见到两个线程。超线程的未来发展，是提升处理器的逻辑线程。英特尔于2016年发布的Core i7-6950X便是将10核心的处理器，加上超线程技术，使之成为20个逻辑线程的产品。

【实验心得】

本次实验主要以Debug的使用为主题，旨在进一步加深对汇编语言程序的相关理解。

首先是Debug的使用。Debug的时候给我最大的感受就是与平常编程时在IDE中的区别。IDE是通过按钮实现快捷调试，而汇编中的Debug操作起来有点像先前学习在命令行中尝试使用gdb进行C程序的调试过程，也是通过gdb a.exe进行程序的装载，后续再通过输入各种命令进行程序单步调试，进行部分变量的检测。这种命令行调试方式最初学的时候就感到不习惯，并且由于时间有限，所以后续也没有深入学习这方面的操作。现在再进行汇编Debug学习的时候感到似曾相识，感觉gdb的方式有些借鉴Debug吧，这也让我进一步感受到了在不同编程语言之间的联系。

在本次实验中对不同寄存器组的学习也是对不同计算机运行原理的学习。例如对CS:IP的学习，使得我对程序执行过程有了深入了解；对栈的学习也使得我对计算机的存储有了比较深入的了解。在这种相对底层的环境下，这样的学习方式或许会比继续高级语言的学习起到更好的效果。

同时在本次实验时，我也尝试将部分程序指令段写成.asm文件，利用masm统一加载入内存中，想规避-a一直写的麻烦，但是我忽略了一个问题，也就是-a直接写入和masm对.asm文件进行编译在DosBox中的结果是不同的，例如下列程序段

mov ax,[0]

add ax,[2]

mov bx,[4]

add bx,[6]

以第一个mov ax,[0]为例，masm将其理解为mov ax,0000，而非偏移量，而直接利用-a mov ax,[0]就不会出现这种问题，这就导致了程序运行结果的差异。这也与《汇编语言》一书中第四章的描述是一致的。所以在实验中进行创新需要综合进行考虑。不能因为怕麻烦就忽略程序运行是否合理而直接进行代码段的硬移植，这在任何平台/语言的编程中都是不合理并且需要规避的现象。

后续可以知道，只要在偏移地址中加入段前缀即可规避MASM的理解性错误

总之，这次实验进一步加深了我对汇编语言程序的相关理解，同时提升了我的综合分析能力和代码排错能力，这些都将成为我以后的宝贵财富。