东南大学

《微机实验及课程设计》

实验报告

实验一、二　汇编指令与基础程序设计

姓　　名：邹滨阳　　学　　号：08022305

专　　业：自动化　　实 验 室：　金智楼 416

实验时间：2024 年 3 月 19 日　报告时间：2024 年 3 月 21 日

评定成绩：　　审阅教师：

实验一 指令与汇编语言基础

一. 实验目的与内容

本实验的目的是通过汇编语言编程，实现对存储单元中的数据进行操作，包括进入全屏命令行方式、构建个人实验环境、编写汇编程序实现特定功能、汇编源程序产生 OBJ 目标文件、链接 OBJ 目标文件产生 EXE 可执行文件、使用调试工具软件调试执行程序等。

二. 基本实验原理

1，进入全屏命令行方式、修改环境的方法：通过修改命令行方式的属性，设置为全屏模式。

2，确定源程序的存放目录，构建个人实验环境：选择一个合适的文件夹用于存放源程序，并设置环境变量。

3，建立、编辑汇编源程序：编写汇编程序框架，包括数据段、堆栈段和代码段，实现特定功能，如对存储单元中的数据进行操作。

4，使用汇编工具（MASM/TASM.EXE）编译源程序：在命令行模式下，使用汇编工具将源程序编译成 OBJ 目标文件。

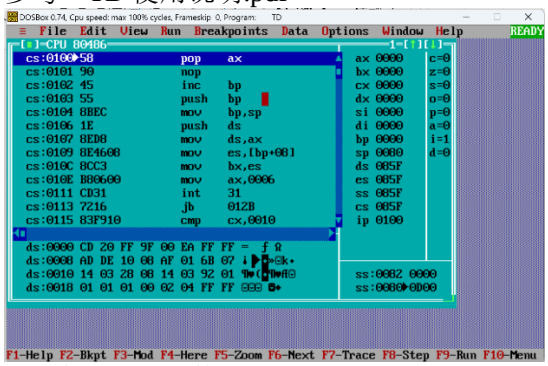
5，使用链接程序（LINK/TLINK.EXE）生成 EXE 可执行文件：使用链接程序将 OBJ 目标文件链接成 EXE 可执行文件。

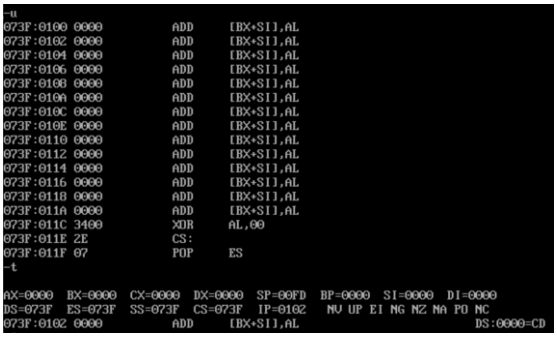
6，使用调试工具软件（TD.EXE/Debug.com）调试执行程序：通过调试工具软件对程序进行调试，观察 CPU 寄存器、存储器环境，单步、断点运行程序，验证程序的正确性。

三. 方案实现与测试

1,学会 TD 使用

参考“TD 使用说明.pdf”





2,学会 DEBUG 使用

F:\DEBUG<回车>进入命令行调试方式

参考“CMD 命令行使用.ppt”或“MASM_DEBUG 使用.pdf”

3,具体任务：编写小段程序，修改环境（寄存器、存储器（代码段、数据段、堆栈段），单步运行，观察各种寻址方式、指令结构、指令功能。**以下以 DEBUG 为例，**TD 自行参考。**注意在调试时不加 H 默认 16 进制。**

（1）基本指令调试

```
cs:0111 BB0020          mov     bx,2000
cs:0114 8B5702          mov     dx,[bx+02]
cs:0117 8D5702          lea     dx,[bx+02]
cs:011A FE07            inc     byte ptr [bx]
```

完成程序编写

然后 eds:2000 修改内存值

```
ds:2000 AB CD 12 34 56 45 55 8B ½=↕4VEUï
```

MOV BX,2000H ;在 debug 中不加 H 默认 16 进制（运行后 bx=2000H ip=0114 就是下一个指令的偏移量）

```
[■]=CPU 80486                    ds:2002 = 3412          1=[↑][↓]
cs:010C 8CC3          mov     bx,es          ax 0000    c=0
cs:010E B80600        mov     ax,0006        bx 2000    z=0
cs:0111 BB0020        mov     bx,2000        cx 0000    s=0
cs:0114►8B5702        mov     dx,[bx+02]     dx 0000    o=0
cs:0117 8D5702        lea     dx,[bx+02]     si 0000    p=0
cs:011A FE07          inc     byte ptr [bx]  di 0000    a=0
cs:011C 04C1          add     al,C1          bp 0000    i=1
cs:011E E10C          loopz   012C           sp 0080    d=0
cs:0120 0BD1          or      dx,cx          ds 085F
cs:0122 8B4606        mov     ax,[bp+06]     es 085F
cs:0125 1F            pop     ds             ss 085F
cs:0126 5D            pop     bp             cs 085F
cs:0127 4D            dec     bp             ip 0114
ds:2000 AB CD 12 34 56 45 55 8B ½=↕4VEUï
ds:2008 EC 1E 8E D8 FF 76 06 0E ∞▲Ä╪ v♦∫
ds:2010 E8 52 FB 1F 5D 4D CA 02 ΦR√▼]M╨☻
ds:2018 00 B8 10 14 45 55 8B EC  ┐►¶EUï∞      ss:0080►0D00
```

MOV DX,[BX+2] ;观察 MOV 和 LEA 指令的不同（运行后发现 dx=3412 是储存器中偏移量为 2002 的值，高字为 34，低字为 12）

```
cs:0114 8B5702        mov     dx,[bx+02]     dx 3412
cs:0117►8D5702        lea     dx,[bx+02]     si 0000
```

LEA DX, [BX+2] 运行后发现 dx=2002 所以 LEA 取的是[bx+02]的偏移量，也就是 bx+02=2002

```
cs:0114 8B5702        mov     dx,[bx+02]     dx 2002
cs:0117 8D5702        lea     dx,[bx+02]     si 0000
cs:011A►FE07          inc     byte ptr [bx]  di 0000
```

INC BYTE PTR [BX] 这个程序是把寄存器中偏移量为 BX 的值＋1，原来的 DS:2000 是 AB＋1 后变成了 AC，由于是 AB 变成 AC 所以最高位符号位为 1，s=1,同时有偶数个 1，p=1

```
≡  File  Edit  View  Run  Breakpoints  Data  Options  Window  Hel
[■]=CPU 80486                                          1=[↑][↓]
cs:010C 8CC3          mov     bx,es          ax 0000    c=0
cs:010E B80600        mov     ax,0006        bx 2000    z=0
cs:0111 BB0020        mov     bx,2000        cx 0000    s=1
cs:0114 8B5702        mov     dx,[bx+02]     dx 2002    o=0
cs:0117 8D5702        lea     dx,[bx+02]     si 0000    p=1
cs:011A FE07          inc     byte ptr [bx]  di 0000    a=0
cs:011C►04C1          add     al,C1          bp 0000    i=1
cs:011E E10C          loopz   012C           sp 0080    d=0
cs:0120 0BD1          or      dx,cx          ds 085F
cs:0122 8B4606        mov     ax,[bp+06]     es 085F
cs:0125 1F            pop     ds             ss 085F
cs:0126 5D            pop     bp             cs 085F
cs:0127 4D            dec     bp             ip 011C
ds:2000 AC CD 12 34 56 45 55 8B ¼=↕4VEUï
ds:2008 EC 1E 8E D8 FF 76 06 0E ∞▲Ä╪ v♦∫
ds:2010 E8 52 FB 1F 5D 4D CA 02 ΦR√▼]M╨☻
ds:2018 00 B8 10 14 45 55 8B EC  ┐►¶EUï∞      ss:0080►0D00
```

在 CS:0100H 处插入指令

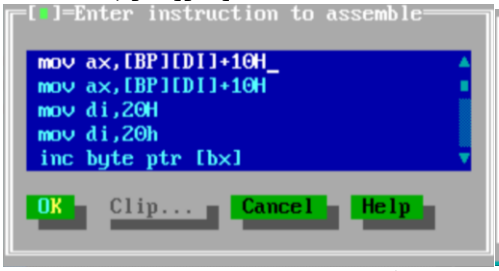
```
cs:0100 BF2000          mov     di,0020
cs:0103▶8B4310          mov     ax,[bp+di+10]
cs:0106 8D4310          lea     ax,[bp+di+10]
```

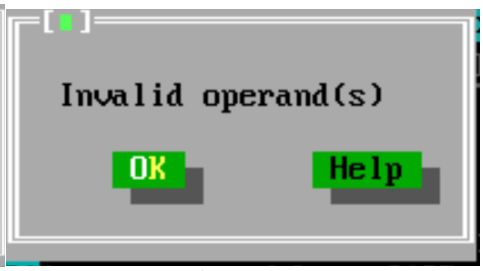
MOV DI, 20H 运行后发现 di 结果为 0020

```
cs:0100 BF2000          mov     di,0020          ax 0030
cs:0103▶8B4310          mov     ax,[bp+di+10]    bx 0000
cs:0106 8D4310          lea     ax,[bp+di+10]    cx 0000
cs:0109 FE07            inc     byte ptr [bx]    dx 0000
cs:010B 088CC3B8        or      [si-473D],cl     si 0000
cs:010F 06              push    es               di 0020
```

;先用 dss:* 观察内存值

MOV AX, [BP][DI]+10H （td 中无法使用）





MOV AX, [BP+DI+10]（运行后发现 ax=0000H，这是因为 BP+DI+10=0030,而 ds:0030 为 0,所以 AX=0）

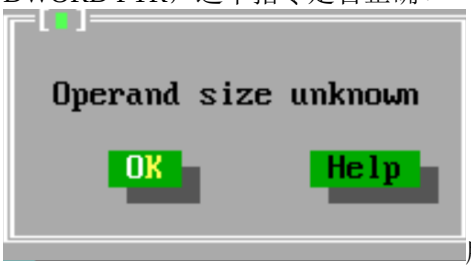
```
cs:0100 BF2000          mov     di,0020          ax 0000
cs:0103 8B4310          mov     ax,[bp+di+10]    bx 0000
cs:0106▶8D4310          lea     ax,[bp+di+10]    cx 0000
```

LEA AX, [BP+DI+10]（运行后发现 ax=0030H，这是因为 BP+DI+10=0030,而 LEA 取的是偏移量,所以 AX=0030）

```
cs:0100 BF2000          mov     di,0020          ax 0030
cs:0103 8B4310          mov     ax,[bp+di+10]    bx 0000
cs:0106 8D4310          lea     ax,[bp+di+10]    cx 0000
cs:0109▶FE07            inc     byte ptr [bx]    dx 0000
```

思考：

ADD [BX], 38H ;第二个操作数是常数，第一个操作数是存储器，不说明 BYTE PTR 或 DWORD PTR，这个指令是否正确？



所以可以看出并不正确，因为不确定大小

（2）JMP CALL PUSH POP 指令调试

;在 0200H 处插入以下指令

```
cs:0200 BB0001          mov     bx,0100
cs:0203 FFE3            jmp     bx
```

MOV BX, 100H; 确保原来 0100H 处有插入过指令

```
cs:0100 B91111          mov     cx,1111
```

```
cs:0200 BB0001          mov     bx,0100          ax 0000
cs:0203 FFE3            jmp     bx               bx 0100
```

可以发现 bx=0100

JMP BX

```
cs:0100 B91111          mov     cx,1111          ip 0100
```

发现段内跳转到了 IP=BX=100 的位置

```
cs:0200 BB0001          mov     bx,0100
cs:0203 FF27            jmp     [bx]
cs:0205 FF6F02          jmp     far [bx+02]

ds:0100 10 20 30 40 50 60 1E 8E ▶ 0@P`▲Ä
```

;同样方法测试以下指令, 先把内存 ds:100H 处赋值,只观察知否跳转即可

MOV BX, 100H

JMP WORD PTR[BX]

```
cs:2010 E852FB          call    1B65             ip 2010
```

可以发现由于是 word ptr 而 20 为高字节 10 为低字节，所以对应的结果为 2010 即 ip=2010 跳转到 CS:2010

JMP DWORD PTR[BX+2] ;注意，这条指令在 DEBUG 中无法测试，只有在程序中测试；调试时要用 FAR PTR 替代

```
cs:4030 0000            add     [bx+si],al       cs 6050
                                                 ip 4030
```

可以发现 6050 为高字，4030 为低字，所以 cs=6050,ip=4030 实现了跳跃

```
cs:020E BB0020          mov     bx,2000
cs:0211 FF37            push    word ptr [bx]
cs:0213 58              pop     ax
```

;观察堆栈 `ds:2000 AC CD 12 34 56 45 55 8B ¼=↕4VEUï`

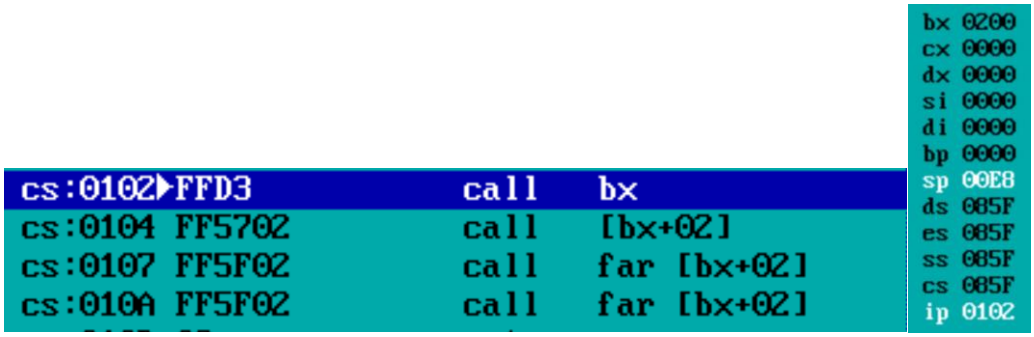
MOV BX，2000H `bx 2000  sp 0080`

PUSH [BX] `sp 007E` 发现 SP=SP-2 说明有新元素入栈根据 ds 的值和偏移量 2000 可以判断是 CDAC 这个数入栈

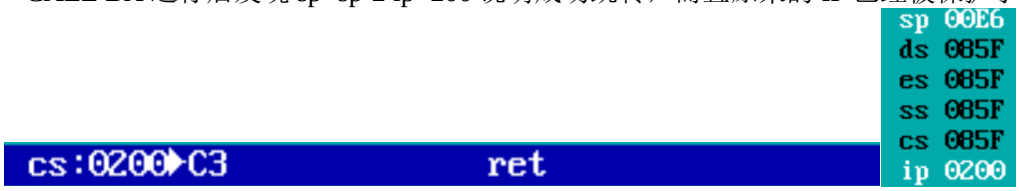
```
ax CDAC
bx 2000
cx 0000
dx 2002
si 0000
di 0000
bp 0000
sp 0080
```

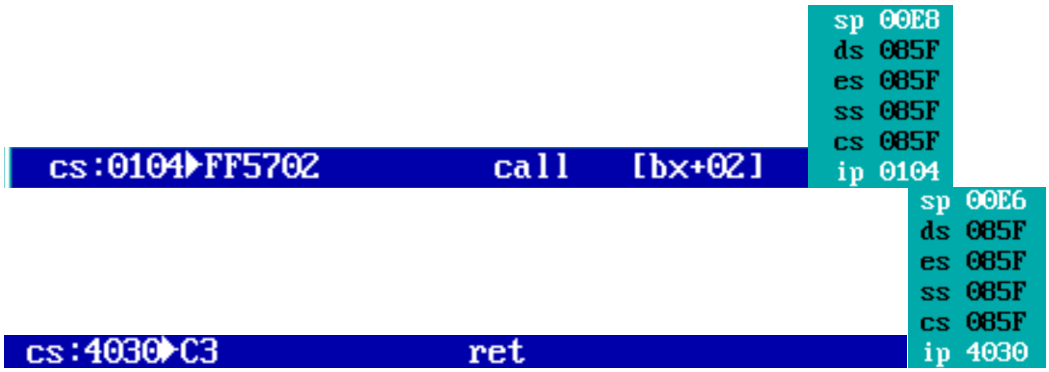
POP AX 发现 SP=SP+2 说明有元素出栈，而有之前推导的可以看出是 CDAC 出栈并传送给了 AX，所以 AX=CDAC

;注意观察保护了什么断点（堆栈变化）、近还是远过程调用（指令指针变化），注意在模拟子过程处插入 RET

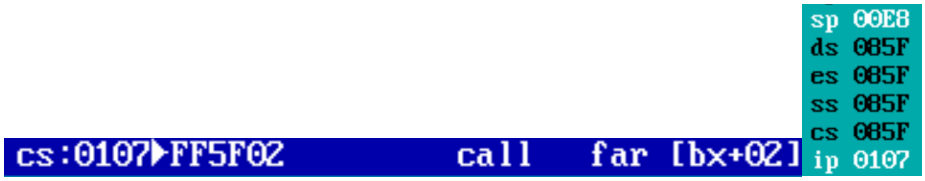


CALL BX 运行后发现 sp=sp-2 ip=200 说明成功跳转，而且原来的 IP 已经被保护了



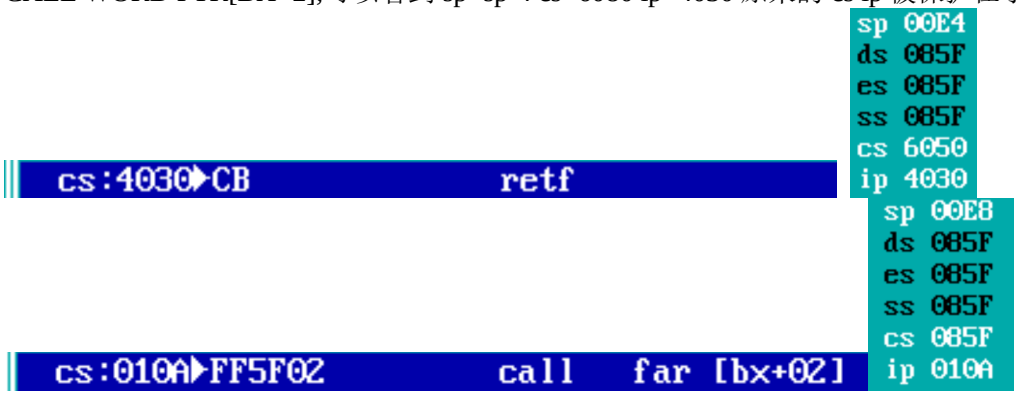


CALL DWORD PTR[BX+2] ;注意，这条指令在 DEBUG 中无法测试，只有在程序中测试



Ret 后发现 sp=sp+2 ip=0107 说明把栈口的数值弹出给了 ip 回到了原来的位置

CALL WORD PTR[BX+2];可以看到 sp=sp-4 cs=6080 ip=4030 原来的 cs ip 被保护在了栈中



Retf 后可以发现 sp=00E8 cs=085F ip=010A 说明原来保存在栈中的 cs 与 IP 被分别弹出给了 cs 与 ip 从而回到了原来的位置

CALL FAR PTR[BX+2] ;可以试一试（效果与刚刚的指令效果相同）

```
sp 00E8
ds 085F
es 085F
ss 085F
cs 085F
ip 010D
cs:4030▶CB              retf
```

;CALL BX 之后，模拟一段近过程（RET 指令结尾）并跟踪进入，观察如何从子过程返回。期间堆栈是如何变化的？CS、IP 如何变化？ 注：模拟远过程返回可用 RETF 指令返回。

答案如上：如果是 near 跳跃则是跳跃的时候 sp=sp-2，ip 存入栈中，如果是 far 跳跃 sp=sp-4，ip，cs 都存入栈中，而且两个都在对应的时刻 ret 和 retf 弹出给 ip 和 cs 从而回到原来的位置

思考题：

1）如何写一段指令，利用堆栈将 AX 与 BX 的值交换？期间堆栈是如何变化的？

PUSH AX 堆栈顶部添加了一个字，存储了原始的 AX 寄存器的值。

PUSH BX 堆栈顶部再次添加了一个字，存储了原始的 BX 寄存器的值。

POP AX 堆栈顶部的字被移除，同时将其值传送到寄存器 AX 中。即 BX 的值被转移到 AX 中

POP BX 堆栈顶部的另一个字被移除，同时将其值传送到寄存器 BX 中。即 AX 的值被转移到了 AX 中

（3）串操作指令调试

代码如下

```
cs:0100▶1E              push   ds                ax 0000
cs:0101 07              pop    es                bx 0000
cs:0102 B92000          mov    cx,0020           cx 0007
cs:0105 BE0000          mov    si,0000           dx 0000
cs:0108 BF2000          mov    di,0020           si 0019
cs:010B FC              cld                      di 0039
cs:010C F3A4            rep movsb                bp 0000
cs:010E 000600CD        add    [CD00],al         sp 0080
```

观察串操作指令加 REP 前缀后

PUSH DS（可以看到原来 sp=0080H,push 后变成了 007EH，说明 ds 的值被压入栈中）

```
cs:0100 1E              push   ds                ax 0000
cs:0101▶07              pop    es                bx 0000
cs:0102 B92000          mov    cx,0020           cx 0007
cs:0105 BE0000          mov    si,0000           dx 0000
cs:0108 BF2000          mov    di,0020           si 0019
cs:010B FC              cld                      di 0039
cs:010C F3A4            rep movsb                bp 0000
cs:010E 000600CD        add    [CD00],al         sp 007E
```

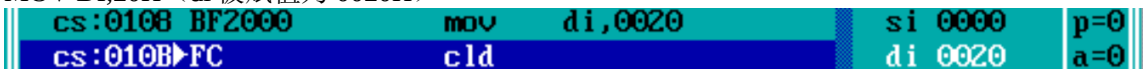
POP ES（sp=sp+2，说明 ds 的值被弹出给了 es，这时候 ds=es）

```
cs:0100 1E              push   ds                ax 0000
cs:0101 07              pop    es                bx 0000
cs:0102▶B92000          mov    cx,0020           cx 0007
cs:0105 BE0000          mov    si,0000           dx 0000
cs:0108 BF2000          mov    di,0020           si 0019
cs:010B FC              cld                      di 0039
cs:010C F3A4            rep movsb                bp 0000
cs:010E 000600CD        add    [CD00],al         sp 0080
```

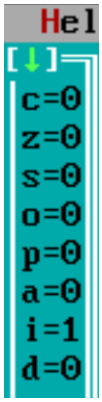
MOV CX,20H （cx 被赋值为 20H）

MOV SI,0（si 被赋值为 0000H）

MOV DI,20H（di 被赋值为 0020H）

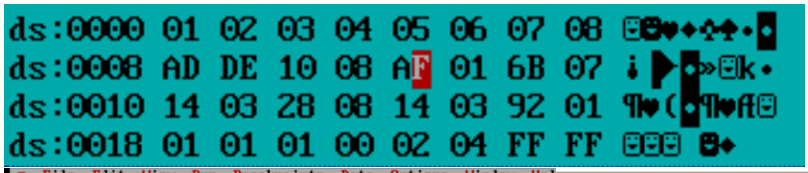


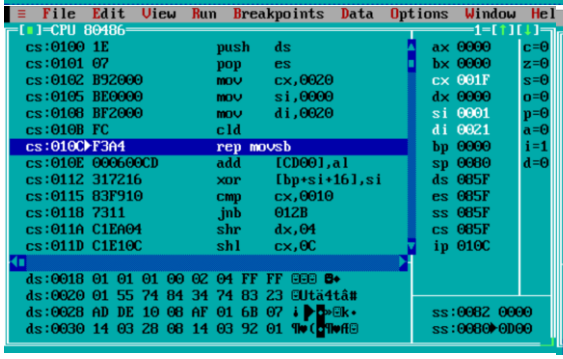
CLD（将方向标志 DF（Direction Flag）清除为 0，这表明在使用字符串操作指令时，数据将按照从源到目的的顺序进行操作。）

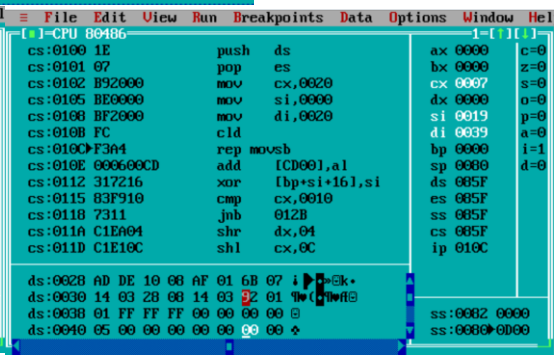


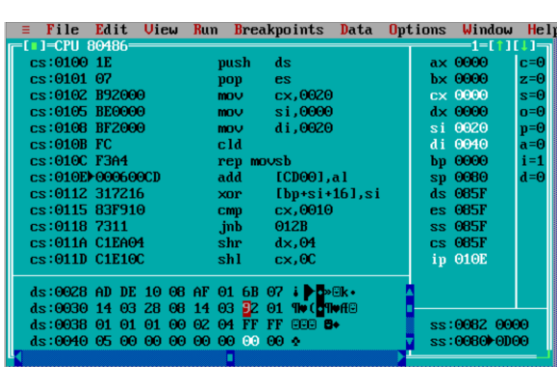
REP MOVSB（这是一个带有 REP 前缀的字符串传送指令，它告诉处理器要重复执行 MOVSB 指令（逐字节复制）的次数由 CX 寄存器中的值决定。因为前面设置了 CX 为 20H，所以这将执行 20H 次 MOVSB 操作，即将从源地址中的数据复制到目的地址中。接下来几张图记录复制的过程，可以看到储存器中的数值被逐渐复制）

这是复制的源数据









具体结果如上，可以看出完成了复制的效果

(4 Level 4: INT 指令调试)
调试验证软中断指令
先给对应段寄存器预先赋值

```
-rds
DS 073F
:2000
-rss
SS 073F
:5000
```

然后完成 程序编写

```
073F:0100 B80100        MOV     AX,0001
073F:0103 CD70          INT     70
073F:0105 BB0100        MOV     BX,0001
```

修改 INT 70 程序对应位置的修改

```
-e0:01C0
0000:01C0  80.11   12.22   00.33   F0.44
```

为跳转地址添加进入标志和返回

```
-u4433:2211
4433:2211 B90300        MOV     CX,0003
4433:2214 CF            IRET
```

记录开始各个寄存器的值

```
AX=0000  BX=0000  CX=0000  DX=0000  SP=00FF  BP=0000  SI=0000  DI=0000
DS=2000  ES=0000  SS=5000  CS=073F  IP=0103   NV UP EI NG NZ NA PE NC
```

开始运行程序，可以看到已经完成了对 AX 的修改

```
-t=100

AX=0001  BX=0000  CX=0000  DX=0000  SP=00FF  BP=0000  SI=0000  DI=0000
DS=2000  ES=0000  SS=5000  CS=073F  IP=0103   NV UP EI NG NZ NA PE NC
073F:0103 CD70          INT     70
```

可以看到堆栈已经记录了相关数据，并且堆栈上移

```
-dss:00F0
5000:00F0  00 14 22 01 00 01 00 00-00 05 01 3F 07 86 72 00
5000:0100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
5000:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
5000:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
5000:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
5000:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
5000:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
5000:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

然后可以看到进入了之前 int 70 修改后指向的地址

```
AX=0001  BX=0000  CX=0000  DX=0000  SP=00F9  BP=0000  SI=0000  DI=0000
DS=2000  ES=0000  SS=5000  CS=4433  IP=2211   NV UP DI NG NZ NA PE NC
4433:2211 B90300        MOV     CX,0003
```

然后修改了 CX

```
-t

AX=0001  BX=0000  CX=0003  DX=0000  SP=00F9  BP=0000  SI=0000  DI=0000
DS=2000  ES=0000  SS=5000  CS=4433  IP=2214   NV UP DI NG NZ NA PE NC
4433:2214 CF            IRET
```

成功返回，堆栈下移

```
AX=0001  BX=0000  CX=0003  DX=0000  SP=00FF  BP=0000  SI=0000  DI=0000
DS=2000  ES=0000  SS=5000  CS=073F  IP=0105   NV UP EI NG NZ NA PE NC
073F:0105 BB0100        MOV     BX,0001
```

但是相关的数据并没有修改，这是由于堆栈的特性，在入栈的时候才会修改

```
-dss:00F0
5000:00F0  00 00 00 14 22 01 00 00-00 05 01 3F 07 A3 01 00
5000:0100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
5000:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
5000:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
5000:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
5000:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
5000:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
5000:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

案例程序研究：

1，add.asm

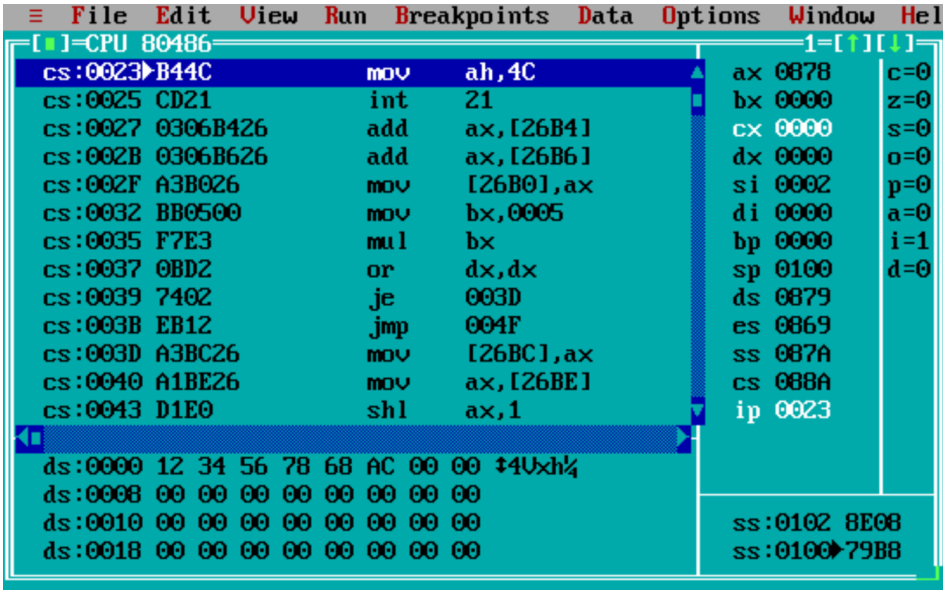
完成程序编写并导入 td

```
cs:0000▶B87908        mov    ax,0879
cs:0003 8ED8          mov    ds,ax
cs:0005 B87A08        mov    ax,087A
cs:0008 8ED0          mov    ss,ax
cs:000A BE0000        mov    si,0000
cs:000D B90200        mov    cx,0002
cs:0010 8A840000      mov    al,[si]
cs:0014 88840400      mov    [si+0004],al
cs:0018 8A840200      mov    al,[si+0002]
cs:001C 10840400      adc    [si+0004],al
cs:0020 46            inc    si
cs:0021 E2ED          loop   0010
cs:0023 B44C          mov    ah,4C
```

完成对数据的初始化，可以看到 ds，cx，si 都被初始化了

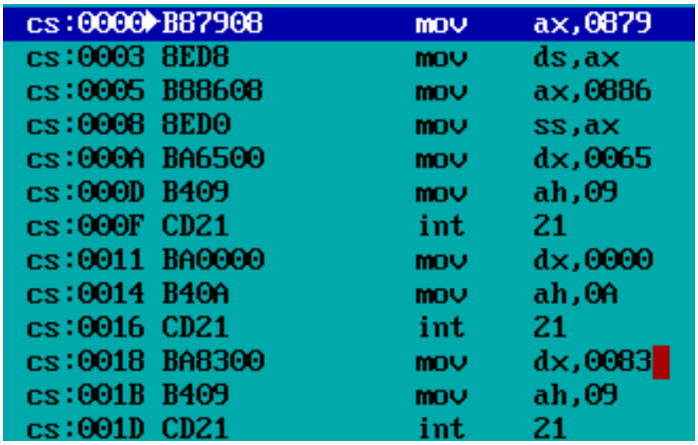
```
cs:0000 B87908        mov    ax,0879          ax 087A
cs:0003 8ED8          mov    ds,ax            bx 0000
cs:0005 B87A08        mov    ax,087A          cx 0002
cs:0008 8ED0          mov    ss,ax            dx 0000
cs:000A BE0000        mov    si,0000          si 0000
cs:000D B90200        mov    cx,0002          di 0000
cs:0010▶8A840000      mov    al,[si]          bp 0000
cs:0014 88840400      mov    [si+0004],al     sp 0100
cs:0018 8A840200      mov    al,[si+0002]     ds 0879
cs:001C 10840400      adc    [si+0004],al     es 0869
cs:0020 46            inc    si               ss 087A
cs:0021 E2ED          loop   0010             cs 088A
cs:0023 B44C          mov    ah,4C            ip 0010

ds:0000 12 34 56 78 00 00 00 00 ↕4Vx
```

接下来通过两次循环，对两个 4 为的 16 进制数做加法，同时使用 adc 命令，来让低字节的进位能参与高字节的预算，结果储存在 ds 段中

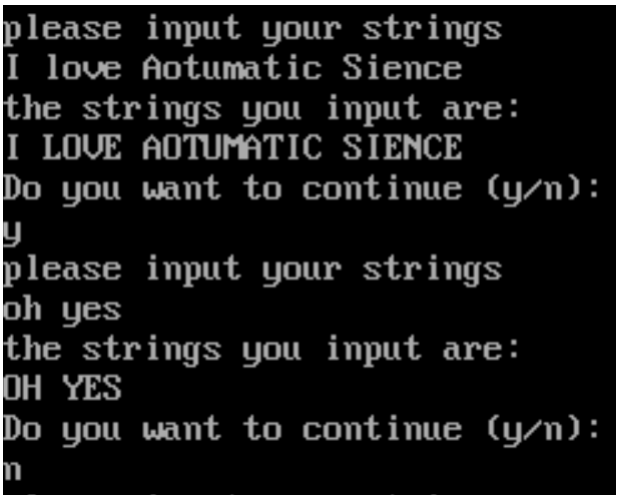


2，string.asm

完成程序编写并导入 td



运行整个程序观察效果



解读程序各个部分：

以下这几行代码的作用是完成了初始化，并通过 int 21（ah=0A）获得输入的字符串，同时利用 int 21（ah=09）输出了相关的提示词

```
cs:0003 8ED8            mov     ds,ax
cs:0005 B88608          mov     ax,0886
cs:0008 8ED0            mov     ss,ax
cs:000A BA6500          mov     dx,0065
cs:000D B409            mov     ah,09
cs:000F CD21            int     21
cs:0011 BA0000          mov     dx,0000
cs:0014 B40A            mov     ah,0A
cs:0016 CD21            int     21
cs:0018 BA8300          mov     dx,0083
cs:001B B409            mov     ah,09
cs:001D CD21            int     21
```

```
ds:0000 64 17 49 20 6C 6F 76 65 d↨I love
ds:0008 20 41 75 74 6F 6D 61 74  Automat
ds:0010 69 63 20 53 69 65 6E 63 ic Sienc
ds:0018 65 0D 00 00 00 00 00 00 e♪
```

这几行代码的功能是先设置好 si 和 cl 的初始值，然后开始循环，每次都截取字符串中的一个字母放入 dl 中，接着把 dl 与‘a’和‘z’做比较大或者小的话都跳转到直接输出模块，而如果在这个区间内就把 dl 减去 20，实现了从小写到大写的转化最后通过 int 21（ah=02）逐个输出字符

```
cs:0037 8A0E0100        mov     cl,[0001]
cs:003B BE0200          mov     si,0002
cs:003E▶8A940000        mov     dl,[si]
cs:0042 80FA61          cmp     dl,61
cs:0045 7208            jb      004F
cs:0047 80FA7A          cmp     dl,7A
cs:004A 7703            ja      004F
cs:004C 80EA20          sub     dl,20
cs:004F B402            mov     ah,02
cs:0051 CD21            int     21
cs:0053 46              inc     si
cs:0054 E2E8            loop    003E
```

这行代码线上输出提示信息，再是输入是否结束代表的标准，把结果和 y 字符做比较，如果相等则跳转到开头，重复过程，如果不是就结束程序

```
cs:0022 BA4200          mov     dx,0042
cs:0025 B409            mov     ah,09
cs:0027 CD21            int     21
cs:0029 B401            mov     ah,01
cs:002B CD21            int     21
cs:002D▶3C79            cmp     al,79
cs:002F 74D9            je      000A
cs:0031 B44C            mov     ah,4C
cs:0033 CD21            int     21
```

3，num.asm

完成程序编写并导入 td

```
cs:0000▶B87908          mov     ax,0879
cs:0003 8ED8            mov     ds,ax
cs:0005 B88408          mov     ax,0884
cs:0008 8ED0            mov     ss,ax
cs:000A C70609000000    mov     word ptr [0009],
cs:0010 BA0D00          mov     dx,000D
cs:0013 B409            mov     ah,09
cs:0015 CD21            int     21
cs:0017 BA0100          mov     dx,0001
cs:001A B40A            mov     ah,0A
cs:001C CD21            int     21
cs:001E C606000001      mov     byte ptr [0000],
cs:0023 E85800          call    007E
```

运行整个程序观察效果

```
please input your number (0~65535)
256
The number you input is:
0100
Do you want to continue (y/n):
```

解读程序各个部分：

程序以下片段和 string.asm 类似，都是负责输入和输出提示信息

```
cs:0003 8ED8            mov     ds,ax
cs:0005 B88408          mov     ax,0884
cs:0008 8ED0            mov     ss,ax
cs:000A C70609000000    mov     word ptr [0009],
cs:0010 BA0D00          mov     dx,000D
cs:0013 B409            mov     ah,09
cs:0015 CD21            int     21
cs:0017 BA0100          mov     dx,0001
cs:001A B40A            mov     ah,0A
cs:001C CD21            int     21
cs:001E C606000001      mov     byte ptr [0000],
```

完成清理和初始化

```
cs:007E F8              clc
cs:007F 33C0            xor     ax,ax
cs:0081 33DB            xor     bx,bx
cs:0083 33D2            xor     dx,dx
```

以下代码的意思是取最高位，判断其是否在‘0’到‘9’范围内，是的话就减去 30 变成 0~9 范围内，否的话就进入另外的环境

```
cs:0085 8A0E0200        mov     cl,[0002]
cs:0089 BE0200          mov     si,0002
cs:008C 8A9C0100        mov     bl,[si+0001]
cs:0090 80FB30          cmp     bl,30
cs:0093 722B            jb      00C0
cs:0095 80FB39          cmp     bl,39
cs:0098 7726            ja      00C0
cs:009A▶80EB30          sub     bl,30
```

以下代码是针对计算最高位对应 10 进制的值，重复最高位数次，不停把 cx 减一，同时在循环前就用栈把 cx 保护起来，用于之后的运算。在重复最高位数后得到 10 的几次方，再与储存在 bx 中位数的值相乘，最后保存在储存器指定位置【0009】处

```
cs:009D FEC9            dec     cl
cs:009F 80F900          cmp     cl,00
cs:00A2 7416            je      00BA
cs:00A4 B80100          mov     ax,0001
cs:00A7 51              push    cx
cs:00A8 F7260B00        mul     word ptr [000B]
cs:00AC E2FA            loop    00A8
cs:00AE 59              pop     cx
cs:00AF F7E3            mul     bx
cs:00B1 01060900        add     [0009],ax
```

指向的位后移跳转之前的代码

```
cs:00B7 46              inc     si
cs:00B8 EBD2            jmp     008C
```

完成从最高位到最低位的计算后在指定存储器【0009】处得到 16 进制下源数据的值

```
ds:0008 00 FF 07 0A 00 0D 0A 70
```

比对加输出相关提示信息

```
cs:0026 803E000000      cmp     byte ptr [0000],
cs:002B 7437            je      0064
cs:002D BA3400          mov     dx,0034
cs:0030 B409            mov     ah,09
cs:0032 CD21            int     21
```

把源数据 16 进制下的值赋给 ax，并通过 4 次右移得到最高位的值

```
cs:0034 A10900          mov     ax,[0009]
cs:0037 8AD4            mov     dl,ah
cs:0039 D0EA            shr     dl,1
cs:003B D0EA            shr     dl,1
cs:003D D0EA            shr     dl,1
cs:003F D0EA            shr     dl,1
```

保护 dx，ax 并通过判断 dx 中的值是 A~F 还是 0~9 进行相应的字符转化，同时更改 ah 对应的值并用 int 21 达成对转变后的值的输出，最后 pop dx，ax 完成保护数据的读取

```
cs:00C6 52              push    dx
cs:00C7 50              push    ax
cs:00C8 80FA09          cmp     dl,09
cs:00CB 7603            jbe     00D0
cs:00CD 80C207          add     dl,07
cs:00D0 80C230          add     dl,30
cs:00D3 B402            mov     ah,02
cs:00D5 CD21            int     21
cs:00D7 58              pop     ax
cs:00D8 5A              pop     dx
cs:00D9 C3              ret
```

对次高位用 and 运算，实现对次高位的保留，重复输出步骤

```
cs:0044 8AD4            mov     dl,ah
cs:0046 80E20F          and     dl,0F
cs:0049 E87A00          call    00C6
```

低位也是同理

```
cs:004C 8AD0            mov     dl,al
cs:004E D0EA            shr     dl,1
cs:0050 D0EA            shr     dl,1
cs:0052 D0EA            shr     dl,1
cs:0054 D0EA            shr     dl,1
cs:0056 E86D00          call    00C6
cs:0059 8AD0            mov     dl,al
cs:005B 80E20F          and     dl,0F
cs:005E E86500          call    00C6
```

完成是否重复的询问并且实现对判定字符的输入

```
cs:0064 BA7400          mov     dx,0074
cs:0067 B409            mov     ah,09
cs:0069 CD21            int     21
cs:006B BA5100          mov     dx,0051
cs:006E B409            mov     ah,09
cs:0070 CD21            int     21
cs:0072 B401            mov     ah,01
cs:0074 CD21            int     21
cs:0076 3C79            cmp     al,79
cs:0078 7490            je      000A
cs:007A B44C            mov     ah,4C
cs:007C CD21            int     21
```

四．提高与创新研究

完成对阶乘算法的设计，要求输入一个 1~8 的数字，并用 10 进制形式输出答案
完成对数据段的书写，并写好相关的提示指令
Ten 存储的是 10 这个常量，input 是输入的数字，right 是判断这个 0 是否属于高位

```
data segment
right dw 0
input db 6,7 dup(0)
output dw 0000h
ten dw 000ah
string1 db 0dh,0ah,'please input your number (0~8)',0dh,0ah,'$'
string2 db 0dh,0ah,'The anser is:',0dh,0ah,'$'
string3 db 0dh,0ah,'Do you want to continue (y/n):',0dh,0ah,'$'
string4 db 0dh,0ah,'The number you input is not from 0~8',0dh,0ah,'$'
data ends
```

完成对堆栈段的书写

```
stacks segment stack
    db 256 dup(0)
stacks ends
```

定义相关的数据段，并且完成对 ds，ss 的赋值

```
code segment
   assume cs:code,ds:data,ss:stacks
main proc far
start:  mov ax,data
     mov ds,ax
        mov ax,stacks
        mov ss,ax
```

完成对提示信息的书写，并读取数字字符，得到需要进行阶乘的数字

```
loop1:  mov output,0000h
        mov right,0
    mov dx,offset string1
     mov ah,09h
        int 21h
        mov dx,offset input
        mov ah,0ah
     int 21h
    mov dx,offset string2
     mov ah,09h
        int 21h
```

接着进入 change 函数，也就是计算函数

```
    call change
```

这里是先对相关的数据进行清零和初始化，然后对输入的数据进行判定，判断是否在 0~8 的范围内，在范围内的话则把字符转化成数字，否则报错。

```
change proc near
    clc
    xor ax,ax
    xor bx,bx
    xor dx,dx
    mov bl,input[2]
    cmp bl,30h
    jb error1
    cmp bl,38h
    ja error1
    sub bl,30h
```

如果是错误的输入的话，先保护 dx，ax 然后输出错误信息，最后结束程序，返回到原函数

```
  error1:
  push dx
  push ax
  mov dx,offset string4
       mov ah,09h
          int 21h
  pop ax
  pop dx
  over: ret
  change endp
```

判断是否是 0 的特殊情况，不是的话就跳过

```
        mov ax,1
        cmp bx,0
        jne special
        mov bx,1
        special:
```

用 loop3 完成对阶乘的计算，通过先对 cx 赋值然后不断减一并相乘从而实现阶乘的功能

```
    mov cx,bx
loop3:  mul cx
    loop loop3
```

先是通过把 ax 导入 output，然后把 output 赋值给 bx，然后把 ax 赋值成 10000，作为最高位，应为 8 的阶乘最高位对应的是 10000，同时把 cx 赋值为。然后进入循环，先是比较 ax 和 bx，如果 ax 小于 bx 则让 bx 减去 ax 同时 cx 加一，然后再次比较 ax 和 bx，如果 bx 大于 ax 则循环，再次比较 ax 和 bx。如果 ax 大于 bx，则 ax 等于 ax/10 代表位数下降。同时比较 cx 和 right，初始时 right 为 0，如果 cx 等于 right 等于 0，则不输出，如果不等于 0，则输出 cx，同时把 right 设置成 0AH 代表这已经不是最高位了，可以输出 0 了。如果符合输出要求，call disp 完成输出。最后将 ax 与 0 比较，如果大于 0，则重复循环，等于 0 的话代表这个是个位，也就是最后一位，说明已经完成了全部的 10 进制转化。

综上所述：以下代码完成了把 16 进制的 4 位数转化成立 10 进制数并输出

```
    mov output,ax
    mov bx,output
    mov ax,10000
    mov cx,0
loop2:  cmp ax,bx
    ja next
    sub bx,ax
    inc cx
    cmp ax,bx
    jbe loop2
next: div ten
    cmp cx,right
    je high1
    mov right,0AH
    call disp
    mov cx,0
high1:  cmp ax,0
    jnz loop2
    jmp over
```

这里完成了对单个数字的输出，也就是每一位对应的字符。先保护 cx，ax，dx 然后把数字转化成字符，然后用 int 21（ah=02h）完成了输出

```
disp proc near
    push cx
    push ax
    push dx
    add cl,30h
    mov ah,02h
    mov dl,cl
    int 21h
    pop dx
    pop ax
    pop cx
    ret
disp endp
```

询问是否重复操作

```
        mov dx,offset string3
        mov ah,09h
           int 21h
           mov ah,1
        int 21h
        cmp al,'y'
        jz loop1
        mov ah,4ch
           int 21h
```

五．分析与总结

本次《微机实验及课程设计》中的汇编指令与基础程序设计实验，是对计算机底层原理和汇编语言应用的重要探索。通过该实验，我深入理解了汇编语言的基本概念、指令结构以及程序设计的过程。

理论与实践结合：实验内容涵盖了汇编指令的基础知识，从修改环境、编写汇编程序、编译链接到调试执行程序，全方位地展示了汇编语言的应用。理论知识通过实践得到了巩固和运用。

操作系统底层原理理解：通过实验过程，我深入了解了操作系统底层的工作原理，包括如何进入全屏命令行方式、如何修改环境以及如何利用汇编语言对存储单元进行操作。这为进一步理解操作系统和计算机组成原理奠定了基础。

调试与错误排查：在实验中，我学会了使用调试工具软件进行程序调试，观察了 CPU 寄存器、存储器环境的变化，并且通过分析程序执行过程中的错误情况，提高了错误排查和修正的能力。

创新能力培养：实验报告中的提高与创新研究部分，展示了对阶乘算法的设计和对 16 进制数到 10 进制数的转换实现。这表现了对汇编语言应用的创新思维和实践能力。

知识应用与总结：通过本次实验，我不仅掌握了汇编语言的基础知识和编程技能，还深刻理解了计算机底层运行原理。实验报告的撰写过程也促使我对实验过程和结果进行了系统性的总结和归纳，巩固了所学知识。

综上所述，该实验在提高我对计算机底层原理的理解和汇编语言应用能力的同时，培养了我的创新思维和问题解决能力。通过实践探索，我对计算机科学领域有了更深层次的认识，并且为今后的学习和工作打下了坚实的基础。