



山东大学

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中所，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 | mov 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

```
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=073F ES=073F SS=073F CS=073F IP=0100  NU UP EI PL NZ NA PO NC
073F:0100 0000          ADD     [BX+SI],AL          DS:0000=CD
```

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

```
-R IP
IP 0100
:0000
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=073F ES=073F SS=073F CS=073F IP=0000  NU UP EI PL NZ NA PO NC
```

可知IP的内容被改变了

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

```
-d 073F:0000
073F:0000  CD 20 3E A7 00 EA FD FF-AD DE 4F 03 A3 01 8A 03  .>.....0....
073F:0010  A3 01 17 03 A3 01 92 01-01 01 01 00 02 FF FF FF  .....
073F:0020  FF FF FF FF FF FF FF-FF FF FF FF 00 00 00 00  .....
073F:0030  00 00 14 00 18 00 3F 07-FF FF FF FF 00 00 00 00  .....?.....
073F:0040  05 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  .....
073F:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20  .!.....
073F:0060  20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20  .....
073F:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00  .....
```

使用-e 命令进行机器码的写入，并-d 查看结果，如下图所示

```
-e 0000  这里仅指明偏移地址即可
073F:0000  CD.b8  20.20  3E.4e  A7.05  00.16  EA.14  FD.bb  FF.00
073F:0008  AD.20  DE.01  4F.d8  03.89  A3.C3  01.01  8A.d8  03.b8
073F:0010  A3.1a  01.00  17.bb  03.26  A3.00  01.00  92.d8  01.00
073F:0018  01.dc  01.00  01.c7  00.b4  02.00  FF.00  FF.d8  FF.04
073F:0020  FF.9c

-d 073F:0000  可知命令已经写入内存中
073F:0000  B8 20 4E 05 16 14 BB 00-20 01 D8 89 C3 01 D8 B8  .N.....
073F:0010  1A 00 BB 26 00 00 D8 00-DC 00 C7 B4 00 00 D8 04  ...&.....
073F:0020  9C FF FF FF FF FF FF-FF FF FF FF 00 00 00 00  .....
073F:0030  00 00 14 00 18 00 3F 07-FF FF FF FF 00 00 00 00  .....?.....
073F:0040  05 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  .....
073F:0050  CD 21 CB 00 00 00 00 00-00 00 00 00 00 20 20 20  .!.....
073F:0060  20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20  .....
073F:0070  20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00  .....
```

这里为提高查看效率，利用-u 反汇编指令进行查看

```

-u 0000 这里也是仅指明偏移地址即可
073F:0000 B8204E MOV AX,4E20
073F:0003 051614 ADD AX,1416
073F:0006 BB0020 MOV BX,2000
073F:0009 01D8 ADD AX,BX
073F:000B 89C3 MOV BX,AX
073F:000D 01D8 ADD AX,BX
073F:000F B81A00 MOV AX,001A
073F:0012 BB2600 MOV BX,0026
073F:0015 00D8 ADD AL,BL
073F:0017 00DC ADD AH,BL
073F:0019 00C7 ADD BH,AL
073F:001B B400 MOV AH,00
073F:001D 00D8 ADD AL,BL
073F:001F 049C ADD AL,9C
    
```

这里与表格是对应的，证明内存中已被写入了正确的指令

证明成功写入后-r 查看内存

```

-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=073F ES=073F SS=073F CS=073F IP=0000 NU UP EI PL NZ NA PO NC
073F:0000 B8204E MOV AX,4E20 ← 第一条指令，更说明了修改是正确的
    
```

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

```

-a
073F:0100 默认会在0100处写入
-a 0000 → 添加偏移地址实现自定义
073F:0000
    
```

重复操作写入命令，如下图所示，0021 处回车结束即可

```

-a 0000
073F:0000 MOV AX,4E20
073F:0003 ADD AX,1416
073F:0006 MOV BX,2000
073F:0009 ADD AX,BX
073F:000B MOV BX,AX
073F:000D ADD AX,BX
073F:000F MOV AX,001A
073F:0012 MOV BX,0026
073F:0015 ADD AL,BL
073F:0017 ADD AH,BL
073F:0019 ADD BH,AL
073F:001B MOV AH,0
073F:001D ADD AL,BL
073F:001F ADD AL,9C
073F:0021
    
```

依次敲入即可

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）


```

DATA SEGMENT

DATA ENDS

STACKS SEGMENT

STACKS ENDS

CODES SEGMENT
    ASSUME CS:CODES, DS:DATA, SS:STACKS
START:
    mov ax,4E20H
    add ax,1416H
    mov bx,2000H
    add ax,bx
    mov bx,ax
    add ax,bx
    mov ax,001AH
    mov bx,0026H
    
```

利用VSCode进行调试

Run Code Ctrl+Alt+N
 转到定义 F12
 转到引用 Shift+F12
 快速查看 >
 打开DOS环境
 运行当前程序(汇编+链接+运行)
 调试当前程序(汇编+链接+调试)
 Find All References Shift+Alt+F12
 更改所有匹配项 Ctrl+F2

调试的结果如下图

```

DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: DEBUG
Copyright (C) Microsoft Corp 1983-1987. All rights reserved.

LINK : warning L4021: no stack segment
D:\>if exist demo2.exe c:\masm\debug demo2.exe
-r
AX=FFFF BX=0000 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0000  NV UP EI PL NZ NA PO NC
076C:0000 B8204E  MOV AX,4E20
-u 0000
076C:0000 B8204E  MOV AX,4E20
076C:0003 051614  ADD AX,1416
076C:0006 BB0020  MOV BX,2000
076C:0009 03C3  ADD AX,BX
076C:000B 8BD8  MOV BX,AX
076C:000D 03C3  ADD AX,BX
    
```

程序被成功载入

CS段寄存器的指向略有不同

命令被成功写入

-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

```

AX=4E20 BX=0000 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0003  NV UP EI PL NZ NA PO NC
076C:0003 051614  ADD AX,1416
    
```

ax被赋值

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

```

AX=6236 BX=0000 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0006  NV UP EI PL NZ NA PE NC
076C:0006 BB0020  MOV BX,2000
    
```

ax做加法运算

mov bx,2000H ; bx=2000H

```
add    ax,bx                ;ax=ax+bx
```

bx被赋值

```

-t
AX=6236 BX=2000 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0009  OV UP EI PL NZ NA PE NC
076C:0009 03C3          ADD    AX,BX
-t
AX=8236 BX=2000 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=000B  OV UP EI NG NZ NA PE NC
076C:000B 8BD8          MOV    BX,AX
    
```

ax的值变化, ax=ax+bx

```
mov     bx,ax                ;bx=ax
```

```
add     ax,bx                ;ax=bx
```

```

076C:000B 8BD8          MOV    BX,AX
-t
AX=8236 BX=8236 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=000D  OV UP EI NG NZ NA PE NC
076C:000D 03C3          ADD    AX,BX
-t
AX=046C BX=8236 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=000F  OV UP EI PL NZ NA PE CY
076C:000F B81A00        MOV    AX,001A
    
```

ax的值被赋给了bx

AX=AX+BX

```
mov     ax,001AH            ;ax=001AH
```

```
mov     bx,0026H            ;bx=0026H
```

```

076C:000F B81A00        MOV    AX,001A
-t
AX=001A BX=8236 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0012  OV UP EI PL NZ NA PE CY
076C:0012 BB2600        MOV    BX,0026
-t
AX=001A BX=0026 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0015  OV UP EI PL NZ NA PE CY
    
```

对ax和bx分别赋值

```
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

```

AX=001A BX=0026 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0015 OV UP EI PL NZ NA PE CY
076C:0015 02C3 ADD AL,BL
-t
ax低位寄存器+bx低位构成ax低位: 1A+26=40
AX=0040 BX=0026 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0017 NU UP EI PL NZ AC PO NC
076C:0017 02E3 ADD AH,BL
-t
ax高位寄存器+bx低位构成ax高位: 00+26=26, ax为2640
AX=2640 BX=0026 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0019 NU UP EI PL NZ NA PO NC
076C:0019 02F8 ADD BH,AL
-t
bx高位+ax低位构成bx高位: 00+40=40, bx低位不变, bx为4026
AX=2640 BX=4026 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=001B NU UP EI PL NZ NA PO NC
    
```

代码执行

```

mov     ah,0                ;ah=0
add     al,bl               ;al+=bl
add     al,9CH              ;al+=9C
    
```

```

AX=2640 BX=4026 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=001B NU UP EI PL NZ NA PO NC
076C:001B B400 MOV AH,00
-t
^ Error
-t
AX=0040 BX=4026 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=001D NU UP EI PL NZ NA PO NC
076C:001D 02C3 ADD AL,BL
-t
AL = 40 + 26 = 66
AX=0066 BX=4026 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=001F NU UP EI PL NZ NA PE NC
076C:001F 049C ADD AL,9C
-t
AL = 66 + 9C = (1)02, 注意这里的进位1为溢出, 不进入AH寄存器
AX=0002 BX=4026 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0021 NU UP EI PL NZ AC PO CY
076C:0021 B44C MOV AH,4C
    
```

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

```

AX=0066 BX=4026 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=001F NU UP EI PL NZ NA PE NC
076C:001F 049C ADD AL,9C
-t
该语句对应机器码大小为2个字节, 从而IP+2
AX=0002 BX=4026 CX=0025 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0021 NU UP EI PL NZ AC PO CY
076C:0021 B44C MOV AH,4C
    
```


【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

```
C:\>DEBUG.EXE
-a 2000:0
2000:0000 mov ax,1
2000:0003 add ax,ax
2000:0005 jmp 2000:0003
2000:0007
```

再使用-r 命令进行 CS:IP 的修改，以在后续-t 执行

```
-r cs
CS 073F
:2000
-r ip
IP 0100
:0000
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=073F ES=073F SS=073F CS=2000 IP=0000  NU UP EI PL NZ NA PO NC
2000:0000 B80100      MOV     AX,0001
```

可见CS:IP已修改完成，后续要执行的指令也是对的

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

```
AX=0000 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=073F ES=073F SS=073F CS=2000 IP=0000  NU UP EI PL NZ NA PO NC
2000:0000 B80100      MOV     AX,0001
-t
AX=0001 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=073F ES=073F SS=073F CS=2000 IP=0003  NU UP EI PL NZ NA PO NC
2000:0003 01C0      ADD     AX,AX
-t
AX=0002 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=073F ES=073F SS=073F CS=2000 IP=0005  NU UP EI PL NZ NA PO NC
2000:0005 EBFC      JMP     0003
-t
AX=0002 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=073F ES=073F SS=073F CS=2000 IP=0003  NU UP EI PL NZ NA PO NC
2000:0003 01C0      ADD     AX,AX
```

IP跳转为0003，重复执行AX+=AX

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 2^8 的运算。

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

```

AX=0100 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=073F ES=073F SS=073F CS=2000 IP=0005  NU UP EI PL NZ NA PE NC
2000:0005 EBFC          JMP     0003
    
```

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

```

-u 2000:0
2000:0000 B80100      MOV     AX,0001
2000:0003 BB0000      MOV     BX,0000
2000:0006 01C0        ADD     AX,AX
2000:0008 83C301      ADD     BX,+01
2000:000B EBF9        JMP     0006
    
```

```

AX=0100 BX=0008 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=073F ES=073F SS=073F CS=2000 IP=000B  NU UP EI PL NZ NA PO NC
2000:000B EBF9        JMP     0006
-t
    
```

← 共循环8次

【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)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

```

-d ffff:0005
FFFF:0000      30 31 2F 30 31 2F 39 32 00 FC 55      01/01/92..U
FFFF:0010 60 10 00 F0 08 00 70 00 08 00 70 00 08 00 70 00      .p.p.p.
FFFF:0020 08 00 70 00 60 10 00 F0 60 10 00 F0 60 10 00 F0      .p. ....
FFFF:0030 A5 FE 00 F0 87 E9 00 F0 55 FF 00 F0 60 10 00 F0      ..U. ....
FFFF:0040 60 10 00 F0 60 10 00 F0 80 10 00 F0 60 10 00 F0      .....e...
FFFF:0050 00 13 00 F0 00 11 00 F0 20 11 00 F0 40 11 00 F0      .....e...
FFFF:0060 A0 11 00 F0 C0 11 00 F0 E0 11 00 F0 20 12 00 F0      .....e...
FFFF:0070 C0 12 00 F0 C0 12 00 F0 40 12 00 F0 60 10 00 F0      .....e...
FFFF:0080 60 12 00 F0 A4
    
```

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

```

-e ffff:0005
FFFF:0005 30.00 31.00 2F.00
FFFF:0008 30.00 31.00 2F.00 39.00 32.00 00.00 FC.00 55.00
FFFF:0010 60.
    
```

再次 -d 查看，如下图所示，日期没有更改成功

```

-e ffff:0005
FFFF:0005 30.00 31.00 2F.00
FFFF:0008 30.00 31.00 2F.00 39.00 32.00 00.0 FC.0 55.0
FFFF:0010 60.

日期没有发生改变

-d ffff:0005
FFFF:0000 30 31 2F-30 31 2F 39 32 00 FC 55 01/01/92..U
FFFF:0010 60 10 00 F0 08 00 70 00-08 00 70 00 08 00 70 00 .....p...p...p.
    
```

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

```

073F:011D push bx
073F:011E pop ax
073F:011F pop bx
073F:0120 push [4]
073F:0124 push [6]
073F:0128
-u 0100
073F:0100 B8FFFF      MOV     AX,FFFF
073F:0103 8ED8        MOV     DS,AX
073F:0105 B80022      MOV     AX,2200
073F:0108 8ED0        MOV     SS,AX
073F:010A BC0001      MOV     SP,0100
073F:010D A10000      MOV     AX,[0000]
073F:0110 03060200    ADD     AX,[0002]
073F:0114 8B1E0400    MOV     BX,[0004]
073F:0118 031E0600    ADD     BX,[0006]
073F:011C 50          PUSH    AX
073F:011D 53          PUSH    BX
073F:011E 58          POP     AX
073F:011F 5B          POP     BX
-u
073F:0120 FF360400     PUSH    [0004]
073F:0124 FF360600     PUSH    [0006]
    
```

-t 命令进行逐行的命令执行，如下图所示

```

-t
AX=FFFF BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=073F ES=073F SS=073F CS=073F IP=0103  NV UP EI PL NZ NA PO NC
073F:0103 8ED8        MOV     DS,AX
073F:0103 8ED8        MOV     DS,AX
-t
AX=FFFF BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=FFFF ES=073F SS=073F CS=073F IP=0105  NV UP EI PL NZ NA PO NC
073F:0105 B80022      MOV     AX,2200
073F:0105 B80022      MOV     AX,2200
-t
AX=2200 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=FFFF ES=073F SS=073F CS=073F IP=0108  NV UP EI PL NZ NA PO NC
073F:0108 8ED0        MOV     SS,AX
073F:0108 8ED0        MOV     SS,AX
-t
AX=2200 BX=0000 CX=0000 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=FFFF ES=073F SS=2200 CS=073F IP=010D  NV UP EI PL NZ NA PO NC
073F:010D A10000      MOV     AX,[0000]
DS:0000=00EA
    
```

ax赋值为ffff

段寄存器间接赋值为FFFF

AX赋值，为SS赋值做准备

SS段寄存器赋值

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

- 12 -

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

```

-r
AX=C0FC BX=6021 CX=0000 DX=0000 SP=00FC BP=0000 SI=0000 DI=0000
DS=FFFF ES=073F SS=2200 CS=073F IP=011E NU UP EI PL NZ NA PE NC
073F:011E 58 POP AX SP+=2
-t
AX=6021 BX=6021 CX=0000 DX=0000 SP=00FE BP=0000 SI=0000 DI=0000
DS=FFFF ES=073F SS=2200 CS=073F IP=011F NU UP EI PL NZ NA PE NC
073F:011F 5B POP BX
-d 2200:00F0
2200:00F0 00 00 FC C0 21 60 00 00-1F 01 3F 07 A3 01 FC C0 .....!`.....?.....
2200:0100 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0110 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0120 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0130 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0140 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0150 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0160 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
    
```

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。

```

-t
      执行POP BX，此时AX内容出栈，存入BX中
AX=6021 BX=C0FC CX=0000 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=FFFF ES=073F SS=2200 CS=073F IP=0120 NU UP EI PL NZ NA PE NC
073F:0120 FF360400 PUSH [0004] DS:0004=30F0
-d 2200:00F0
2200:00F0 00 00 FC C0 21 60 21 60-00 00 20 01 3F 07 A3 01 .....!`!`.. .?...
2200:0100 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0110 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0120 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0130 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0140 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0150 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0160 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
    
```

下面将 30F0 内容进行入栈，此时 SP=0100

```

-r
AX=6021 BX=C0FC CX=0000 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=FFFF ES=073F SS=2200 CS=073F IP=0120 NU UP EI PL NZ NA PE NC
073F:0120 FF360400 PUSH [0004] DS:0004=30F0
    
```

执行后如下图所示

```

-r
AX=6021 BX=C0FC CX=0000 DX=0000 SP=0100 BP=0000 SI=0000 DI=0000
DS=FFFF ES=073F SS=2200 CS=073F IP=0120 NU UP EI PL NZ NA PE NC
073F:0120 FF360400 PUSH [0004] DS:0004=30F0
-t
AX=6021 BX=C0FC CX=0000 DX=0000 SP=00FE BP=0000 SI=0000 DI=0000
DS=FFFF ES=073F SS=2200 CS=073F IP=0124 NU UP EI PL NZ NA PE NC
073F:0124 FF360600 PUSH [0006] DS:0006=2F31
-d 2200:00F0
2200:00F0 00 00 FC C0 21 60 00 00-24 01 3F 07 A3 01 F0 30 ....!`..$.?....0
2200:0100 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0110 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0120 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0130 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0140 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0150 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0160 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
    
```

这里 F030 实现了入栈，并对原来的栈区内存进行了覆盖。

```

AX=6021 BX=C0FC CX=0000 DX=0000 SP=00FE BP=0000 SI=0000 DI=0000
DS=FFFF ES=073F SS=2200 CS=073F IP=0124 NU UP EI PL NZ NA PE NC
073F:0124 FF360600 PUSH [0006] DS:0006=2F31
-d 2200:00F0
2200:00F0 00 00 FC C0 21 60 00 00-24 01 3F 07 A3 01 F0 30 ....!`..$.?....0
2200:0100 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0110 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0120 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0130 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0140 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0150 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
2200:0160 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
-t
AX=6021 BX=C0FC CX=0000 DX=0000 SP=00FC BP=0000 SI=0000 DI=0000
DS=FFFF ES=073F SS=2200 CS=073F IP=0128 NU UP EI PL NZ NA PE NC
073F:0128 0000 ADD [BX+SI],AL DS:C0FC=00
    
```



(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

```
-a
073F:0100 mov ax,1000
073F:0103 mov ds,ax
073F:0105 mov ds,[0]
073F:0109 add ds,ax
          ^ Error
```

在写入命令后报错，说明不可以对ds进行直接操作

(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]
```

```
-r
AX=0000 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=073F ES=073F SS=073F CS=073F IP=0100  NU UP EI PL NZ NA PO NC
073F:0100 B80010      MOV     AX,1000
-t
AX=1000 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=073F ES=073F SS=073F CS=073F IP=0103  NU UP EI PL NZ NA PO NC
073F:0103 8ED8      MOV     DS,AX
-t
AX=1000 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=1000 ES=073F SS=073F CS=073F IP=0105  NU UP EI PL NZ NA PO NC
073F:0105 8E1E0000  MOV     DS,[0000]
-t
AX=1000 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=0000 ES=073F SS=073F CS=073F IP=0109  NU UP EI PL NZ NA PO NC
073F:0109 0000      ADD     [BX+SI],AL
-t
-d 1000:0000
1000:0000  00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
1000:0010  00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
```

利用AX间接实现DS的赋值

DS:0000=0000

DS:0000=60

(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？

```

C:\>debug
~a
0B39:0100 mov ax,2000
0B39:0103 mov ss,ax
0B39:0105 mov sp,10
0B39:0108 mov ax,3123
0B39:010B push ax
0B39:010C mov ax,3366
0B39:010F push ax
0B39:0110
~
~e 2000:0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
~
~d 2000:0 f
2000:0000 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
~
~r
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B39 ES=0B39 SS=0B39 CS=0B39 IP=0100 NU UP EI PL NZ NA PO NC
0B39:0100 B80020 MOV AX,2000
~t
AX=2000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0B39 ES=0B39 SS=0B39 CS=0B39 IP=0103 NU UP EI PL NZ NA PO NC
0B39:0103 8ED0 MOV SS,AX
~t
AX=2000 BX=0000 CX=0000 DX=0000 SP=0010 BP=0000 SI=0000 DI=0000
DS=0B39 ES=0B39 SS=2000 CS=0B39 IP=0108 NU UP EI PL NZ NA PO NC
0B39:0108 B82331 MOV AX,3123
~d 2000:0 f
2000:0000 00 00 00 00 00 00 00 20-00 00 08 01 39 0B 9D 05 .....
~
    
```

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

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|---|----|---|----|---|----|
| | | | | OF | DF | IF | TF | SF | ZF | | AF | | PF | | CF |

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

```

-r
AX=1000 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=0000 ES=073F SS=073F CS=073F IP=0109  NV UP EI PL NZ NA PO NC
073F:0109 B80020      MOV     AX,2000
-t

AX=2000 BX=0000 CX=0000 DX=0000 SP=00FD BP=0000 SI=0000 DI=0000
DS=0000 ES=073F SS=073F CS=073F IP=010C  NV UP EI PL NZ NA PO NC
073F:010C 8ED0      MOV     SS,AX
-t

AX=2000 BX=0000 CX=0000 DX=0000 SP=0010 BP=0000 SI=0000 DI=0000
DS=0000 ES=073F SS=2000 CS=073F IP=0111  NV UP EI PL NZ NA PO NC
073F:0111 B82331      MOV     AX,3123
-d 2000:0 f
2000:0000 00 00 00 00 00 00 00 20-00 00 11 01 3F 07 A3 01 .....?...
```

这里表明了标志寄存器的状态

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。

2. 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 2^{32} bit，也就是 4GB 内存。而 64 位操作系统理论上的寻址空间为 2^{64} bit，转化单位为 2,147,483,648GB

3. 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。

4. 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在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。

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。