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：分支程序实验和循环程序实验

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 2021 年 3 月 13 日

实验报告

【实验目的】

1. 学习分支程序和循环程序的相关设计

【实验要求】

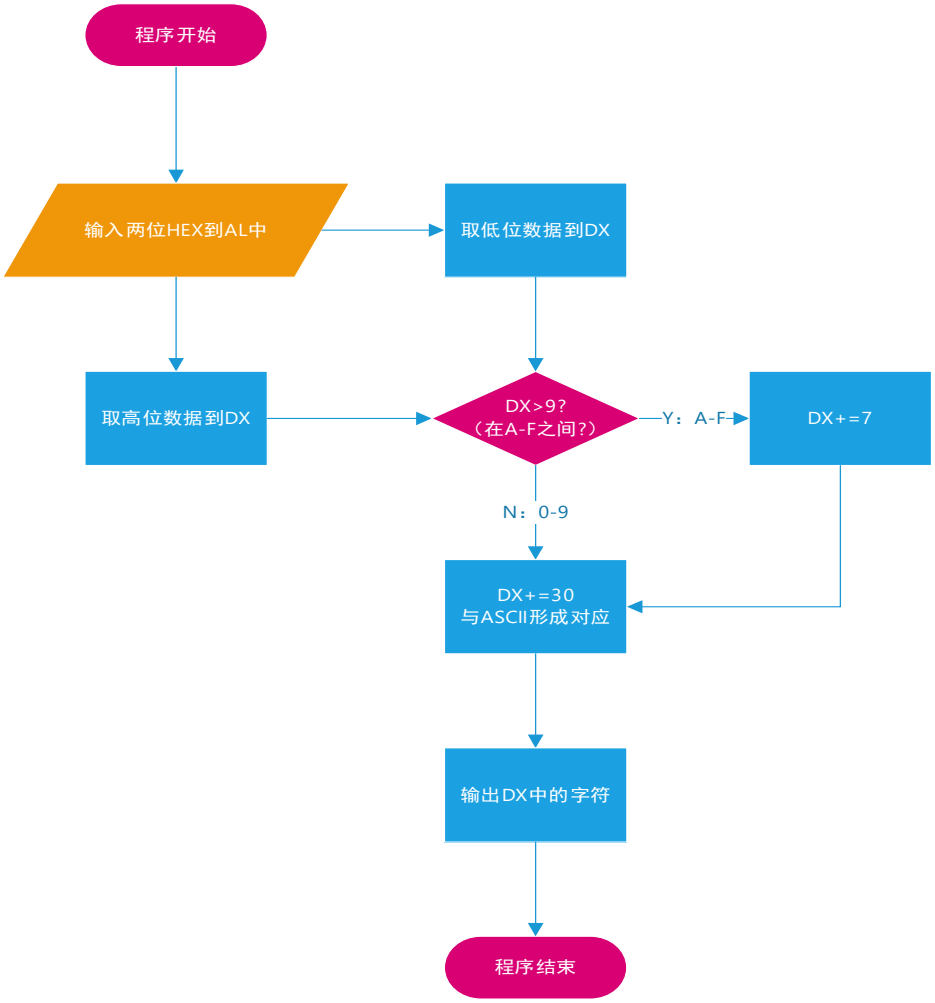
1. 完成实验内容，截图显示，写程序前必须学习画程序的流程图。
2. 读懂示例程序，如果需要可以修改语句或者采用其他更好的方法实现。
3. 要求对于 Code 段中每行程序都尝试写注释，即理解这行语句到底做了什么？

【实验具体内容】

1. 编写一个程序，把 AL 寄存器中的两位十六进制数显示出来
2. 编写一个程序，判别键盘上输入的字符；若是 1-9 字符，则显示之；若为 A-Z 或 a-z 字符，均显示′c′；若是回车字符<CR>(其 ASCII 码为 0DH)，则自动结束程序，若为其它字符则不显示，循环等待新的字符输入

【第一个实验：分支程序实验】

（1）实验流程图



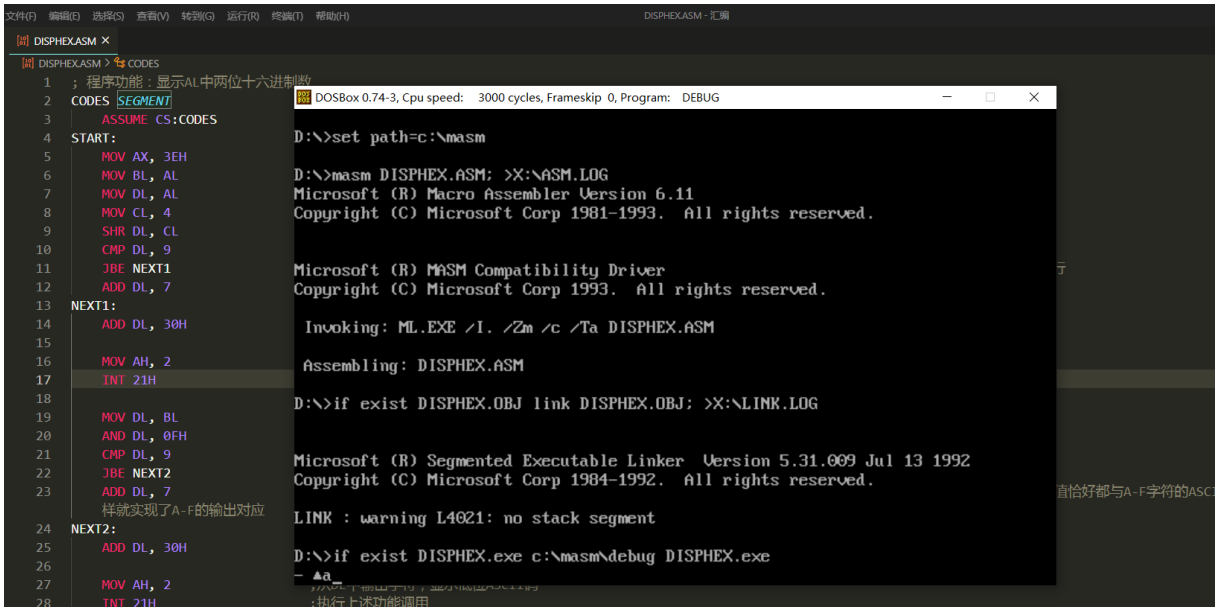
（2）实验源代码（示例程序代码）：代码已经添加注释

```
; 程序功能：显示 AL 中两位十六进制数
CODES SEGMENT
    ASSUME CS:CODES              ;伪指令，CS 段寄存器与 CODES 产生联系
START:
    MOV AL, 3EH                  ;向 AL 中写入原始数据 3E，这也是程序需要输出的数据，需要指出的是这里的数据可以是任意的两位 HEX 值
    MOV BL, AL                   ;向 BL 中写入 AL（3E），起到暂存数据的作用
    MOV DL, AL                   ;向 DL 中写入 AL（3E）
    MOV CL, 4                    ;向 CL 中写入 4，作为二进制下的逻辑右移位数，实现了 DL 十六进制角度上整体右移一位，起到了取高位的作用
    SHR DL, CL                   ;逻辑右移指令，实现了 DL 十六进制角度上整体右移一位
    CMP DL, 9                    ;比较指令，对两数相减进行操作，这里会改变 Flag 中部分内容供后续 JBE 指令作条件判断
    JBE NEXT1                    ;条件转移指令，如果低于或等于(<=)则跳转，与 CMP 连用相当于 if(DL<=9)=>跳转至 NEXT1 处向下执行，否则继续向下执行
    ADD DL, 7                    ;DL 自加 7，主要是处理十六进制 A-F 的显示问题，具体见下面的描述
NEXT1:
    ADD DL, 30H                  ;DL 自加 30，目的是与原字符的 ASCII 值进行匹配

    MOV AH, 2                    ;从 DL 中输出字符
    INT 21H                      ;执行上述功能调用

    MOV DL, BL                   ;DL 取出暂存于 BL 的原数据 3E
    AND DL, 0FH                  ;对两数进行与运算，结果存放至 DL 中，起到了取低位的作用
    CMP DL, 9                    ;比较指令，这里会改变 Flag 中部分内容供后续 JBE 指令作条件判断
    JBE NEXT2                    ;与 CMP 连用，相当于 if(DL<=9)=>跳转至 NEXT2 向下执行，否则一直向下执行
    ADD DL, 7                    ;DL 自加 7，如果不通过条件会+7 处理之后再进入 NEXT2，主要是因为大于 9 之后十六进制显示为 A-F，而整体+7 之后的十六进制值恰好都与 A-F 字符的 ASCII 值差 30，这样就实现了 A-F 的输出对应
NEXT2:
```

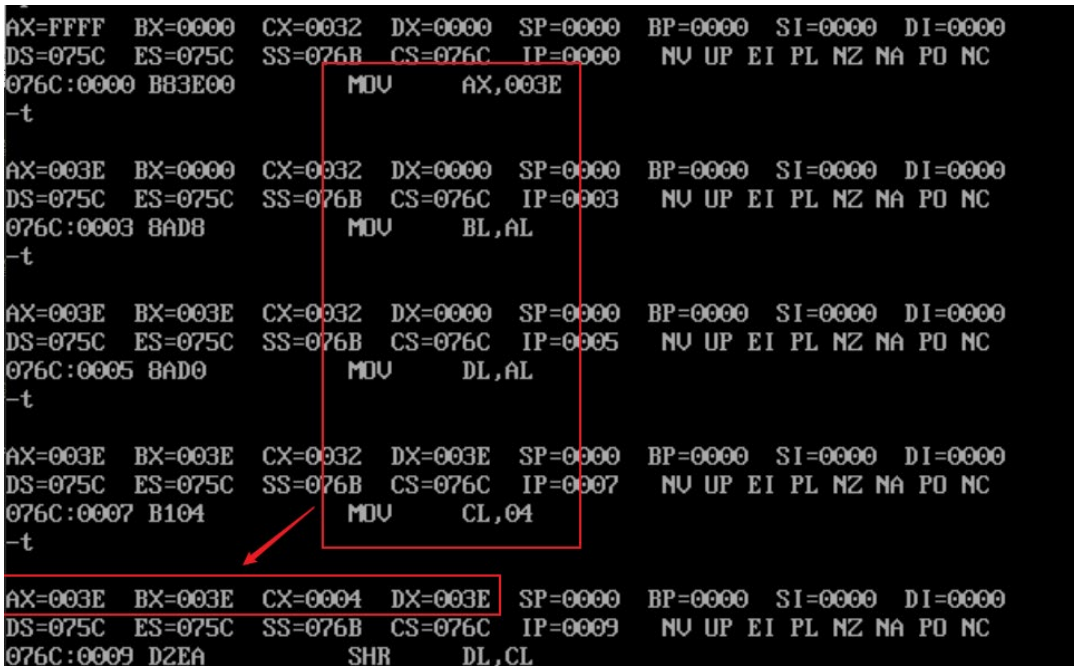
```
    ADD DL, 30H                 ;DL 自加 30，目的是与原字符的 ASCII 值进行匹配

    MOV AH, 2                   ;从 DL 中输出字符，显示低位 ASCII 码
    INT 21H                     ;执行上述功能调用

    MOV AH, 4CH                 ;返回命令行窗口
    INT 21H                     ;执行调用，实现返回命令行窗口
CODES ENDS
END START
```

（3）实验代码、过程、相应结果（截图）并对实验进行说明和分析：

将示例代码在 VSCode 中编辑完成后另存为 DISPHEX.ASM，编译链接之后生成 EXE 文件，装载入 Debug 中进行分析，如下图所示

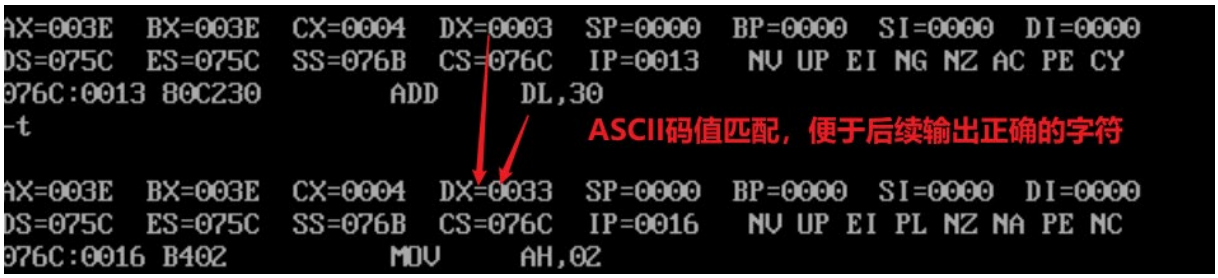


下面开始利用-t 进行指令的逐步执行和分析

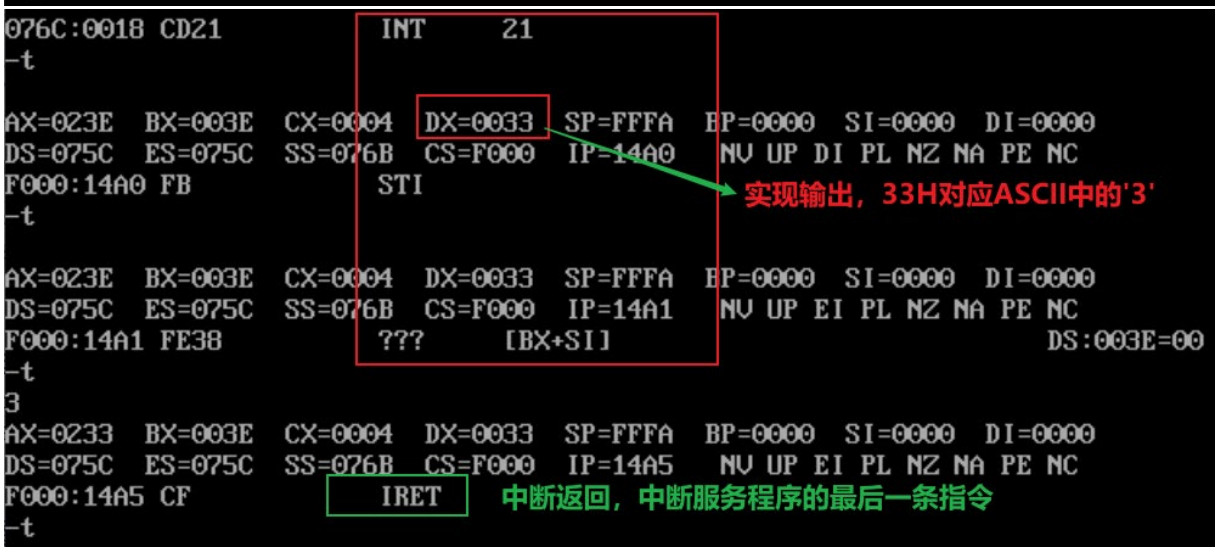


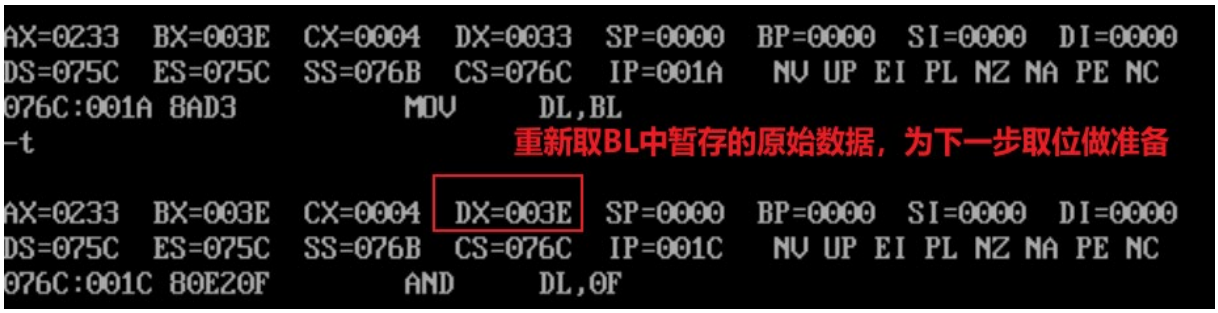
第一部分的指令执行情况如上图所示，这里实现了 **AL 读入原始数据，BL 暂存数据，CL 存储位移数，DL 存储输出量**

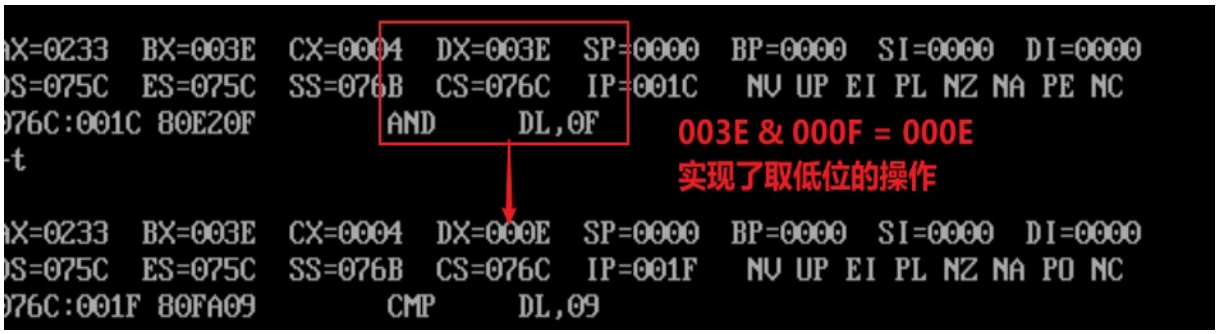
（这里跳转意味着不进行再处理的流程，直接进行 ASCII 的匹配过程）

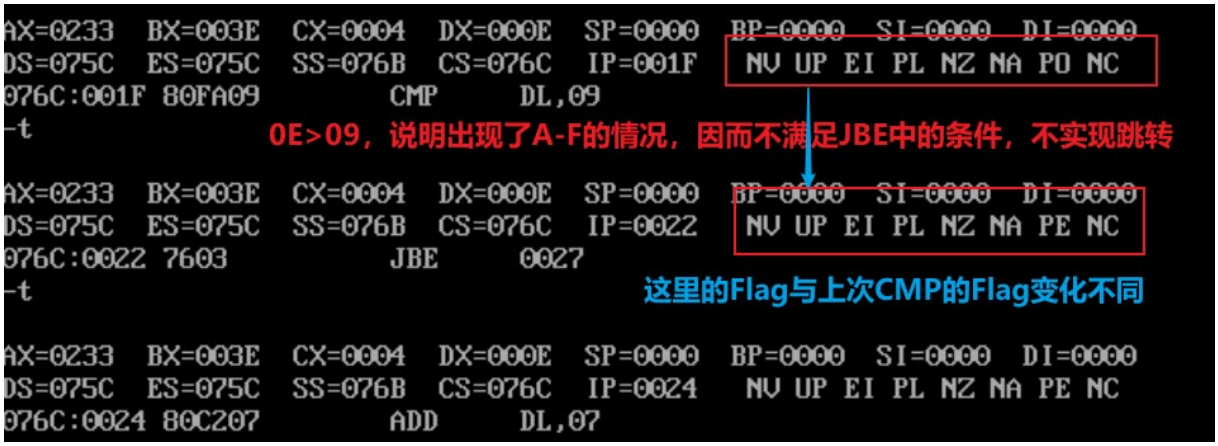

AX=003E BX=003E CX=0004 DX=0003 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0013 NV UP EI NG NZ AC PE CY
076C:0013 80C230 ADD DL,30
-t
ASCII码值匹配，便于后续输出正确的字符
AX=003E BX=003E CX=0004 DX=0033 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0016 NV UP EI PL NZ NA PE NC
076C:0016 B402 MOV AH,02

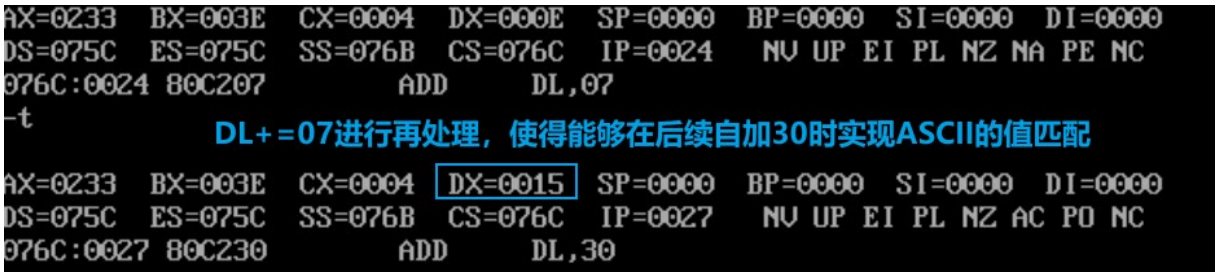

AX=003E BX=003E CX=0004 DX=0033 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0016 NV UP EI PL NZ NA PE NC
076C:0016 B402 MOV AH,02
-t
两条指令配合使用，实现输出DL中的ASCII匹配字符
AX=023E BX=003E CX=0004 DX=0033 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=0018 NV UP EI PL NZ NA PE NC
076C:0018 CD21 INT 21
-t

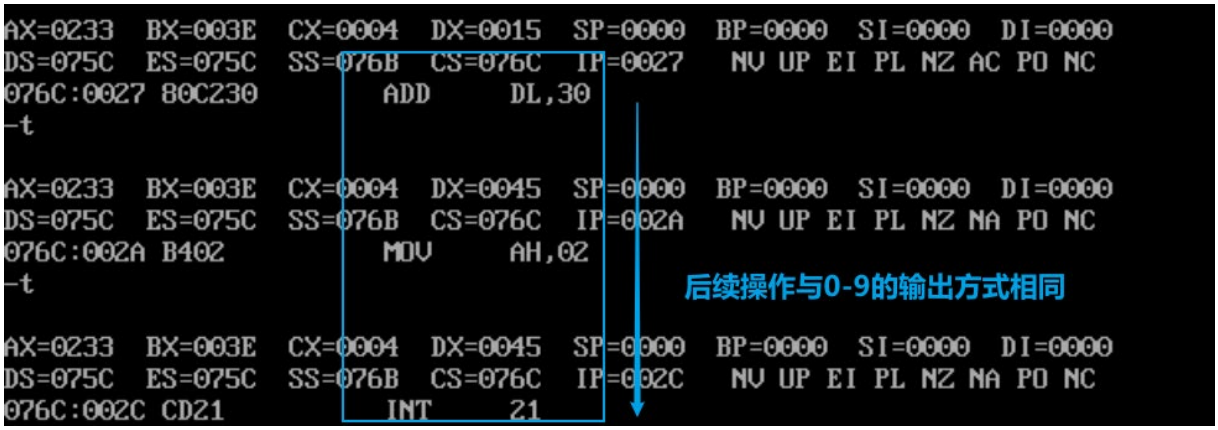

076C:0018 CD21 INT 21
-t
AX=023E BX=003E CX=0004 DX=0033 SP=FFFA BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=F000 IP=14A0 NV UP DI PL NZ NA PE NC
F000:14A0 FB STI
实现输出，33H对应ASCII中的'3'
-t
AX=023E BX=003E CX=0004 DX=0033 SP=FFFA BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=F000 IP=14A1 NV UP EI PL NZ NA PE NC
F000:14A1 FE38 ??? [BX+SI] DS:003E=00
-t
3
AX=0233 BX=003E CX=0004 DX=0033 SP=FFFA BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=F000 IP=14A5 NV UP EI PL NZ NA PE NC
F000:14A5 CF IRET
中断返回，中断服务程序的最后一条指令
-t


AX=0233 BX=003E CX=0004 DX=0033 SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=001A NV UP EI PL NZ NA PE NC
076C:001A 8AD3 MOV DL,BL
-t
重新取BL中暂存的原始数据，为下一步取位做准备
AX=0233 BX=003E CX=0004 DX=003E SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=001C NV UP EI PL NZ NA PE NC
076C:001C 80E20F AND DL,0F

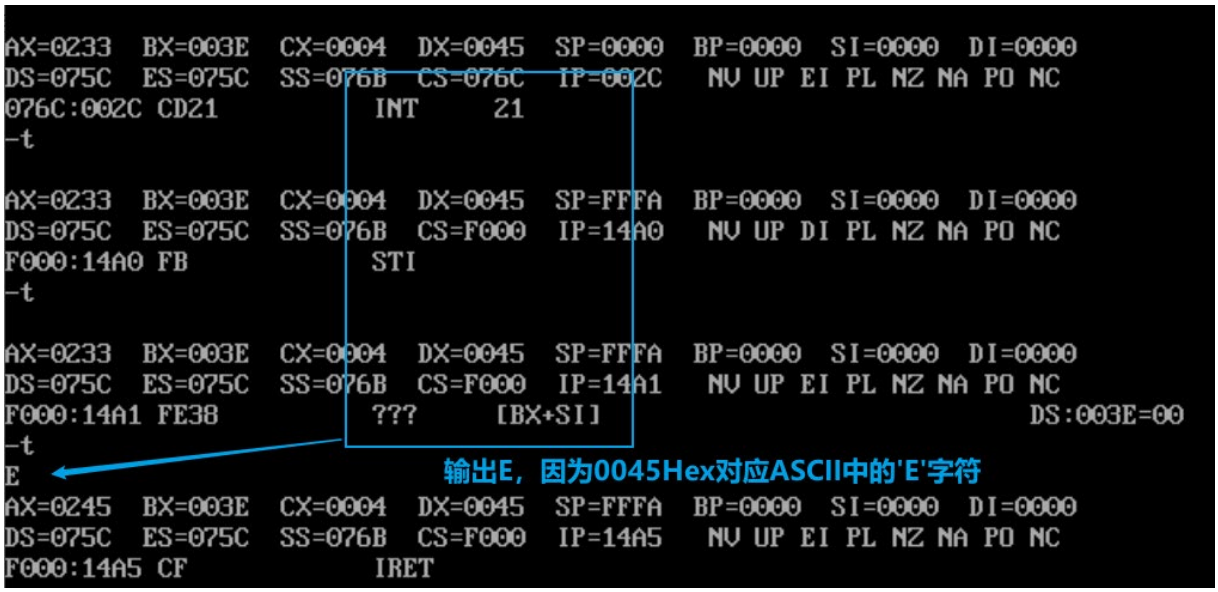

AX=0233 BX=003E CX=0004 DX=003E SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=001C NV UP EI PL NZ NA PE NC
076C:001C 80E20F AND DL,0F
-t
003E & 000F = 000E
实现了取低位的操作
AX=0233 BX=003E CX=0004 DX=000E SP=0000 BP=0000 SI=0000 DI=0000
DS=075C ES=075C SS=076B CS=076C IP=001F NV UP EI PL NZ NA PO NC
076C:001F 80FA09 CMP DL,09

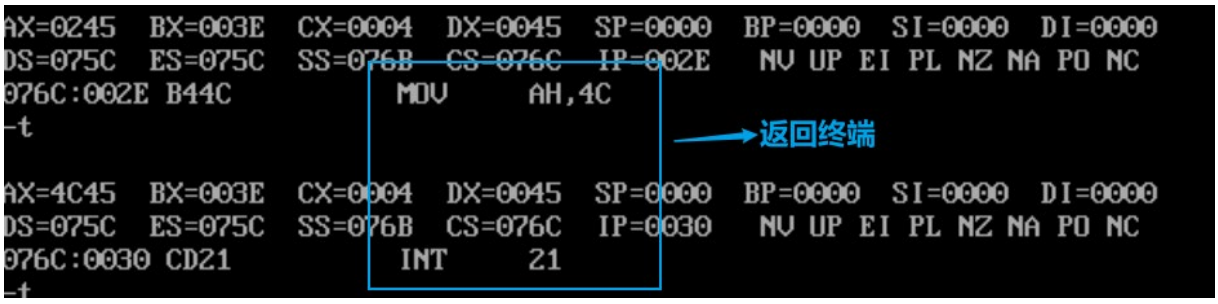






后续的输出过程与 0-9 的输出是完全相同的，这里不重复分析，仅给出执行过程

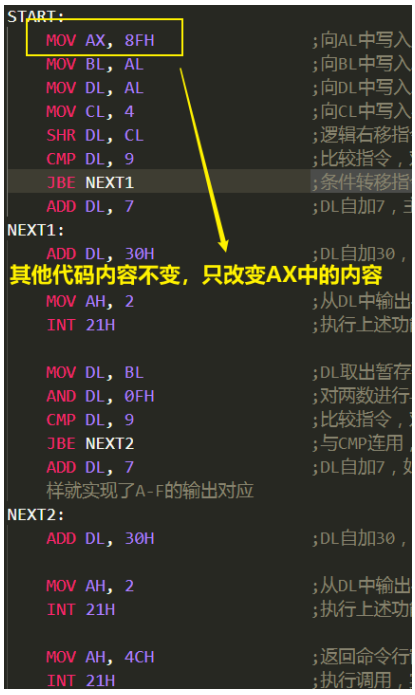




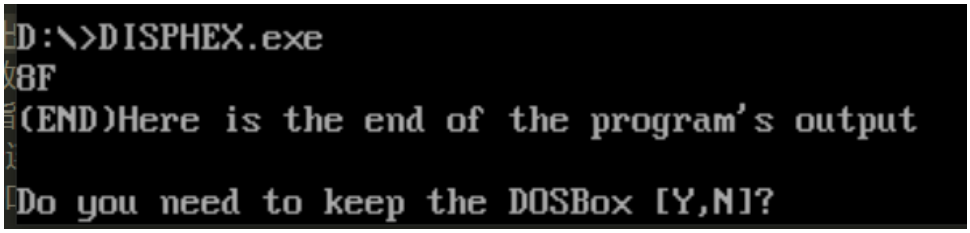
上图代表着程序的结束。

至此程序分析完毕，整个程序实现了预期的功能，即输出 AL 寄存器中的内容。（程序的完整注释已经附在了源代码部分）

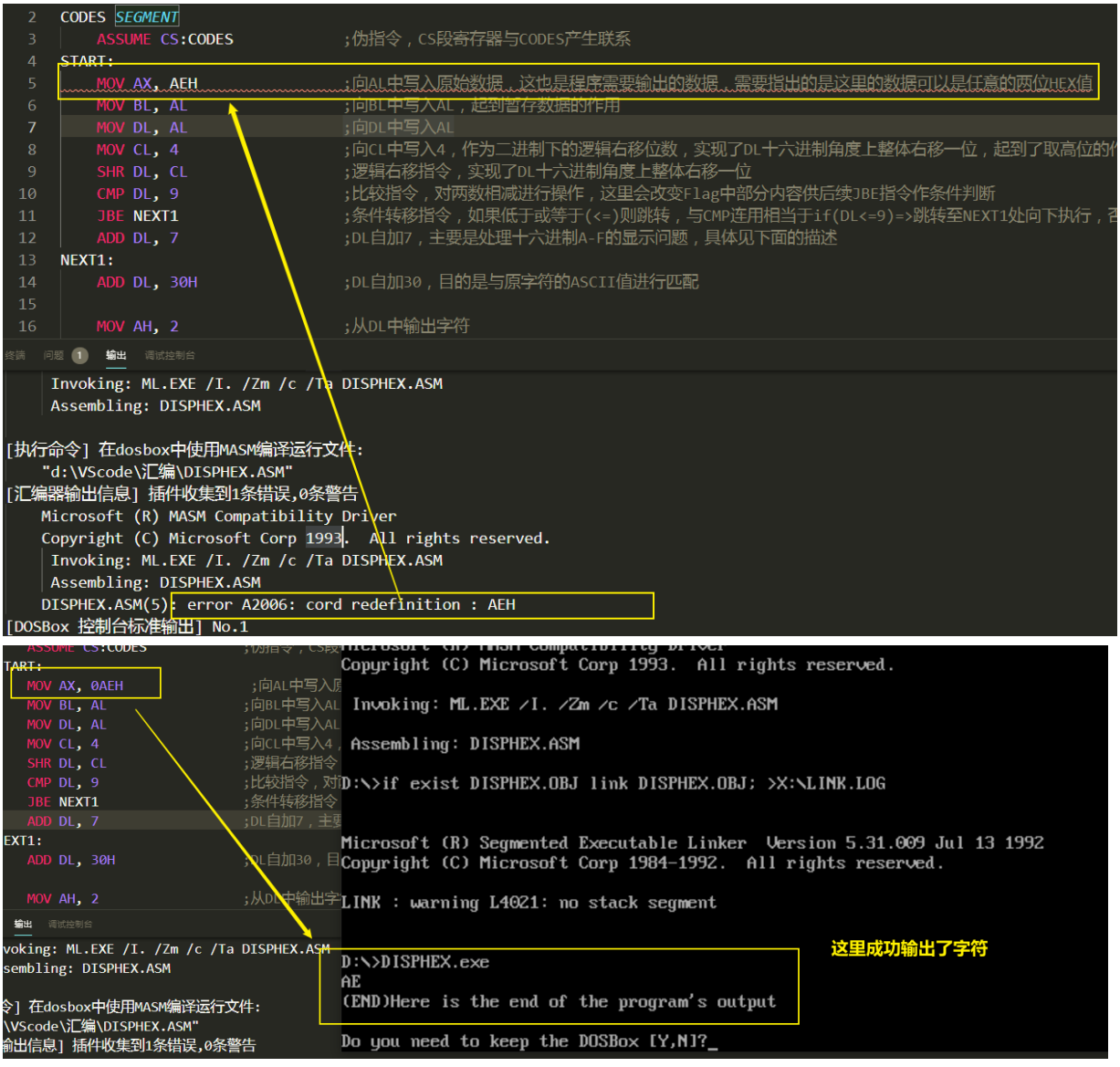
如果要实现任意字符的输出，只需要更改开头的 AL 内容即可，例如下图所示



运行结果如下所示，可见成功输出了 AL 寄存器中的内容



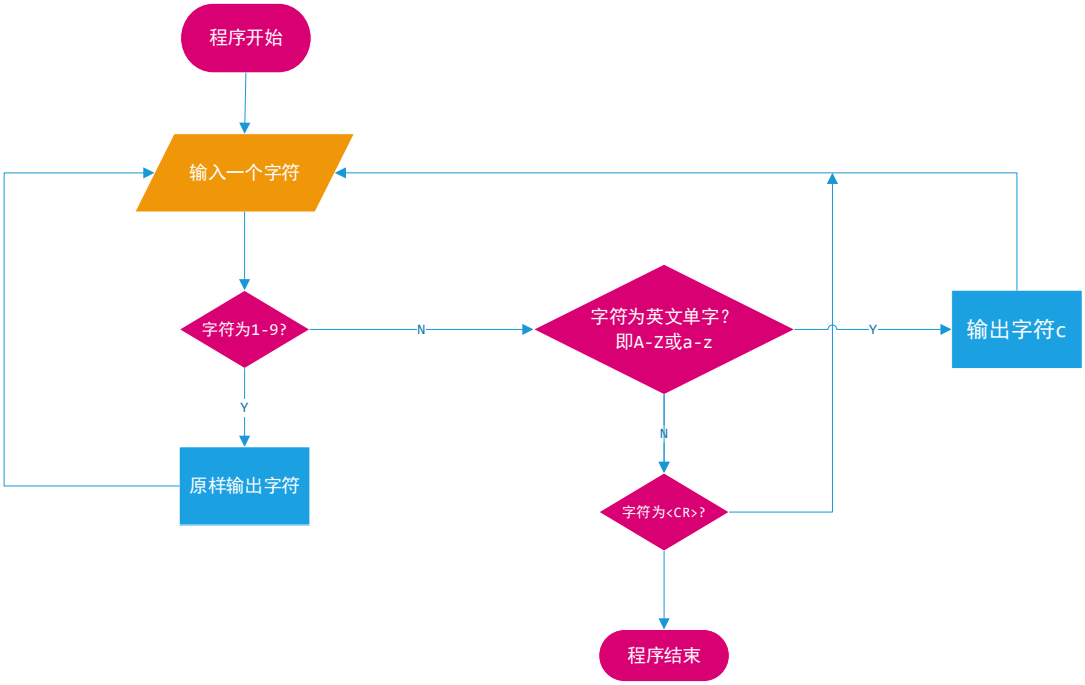
还需要注意的是，如果第一位为 A-F 的情况，需要在高位补零，否则会报错，如下图所示



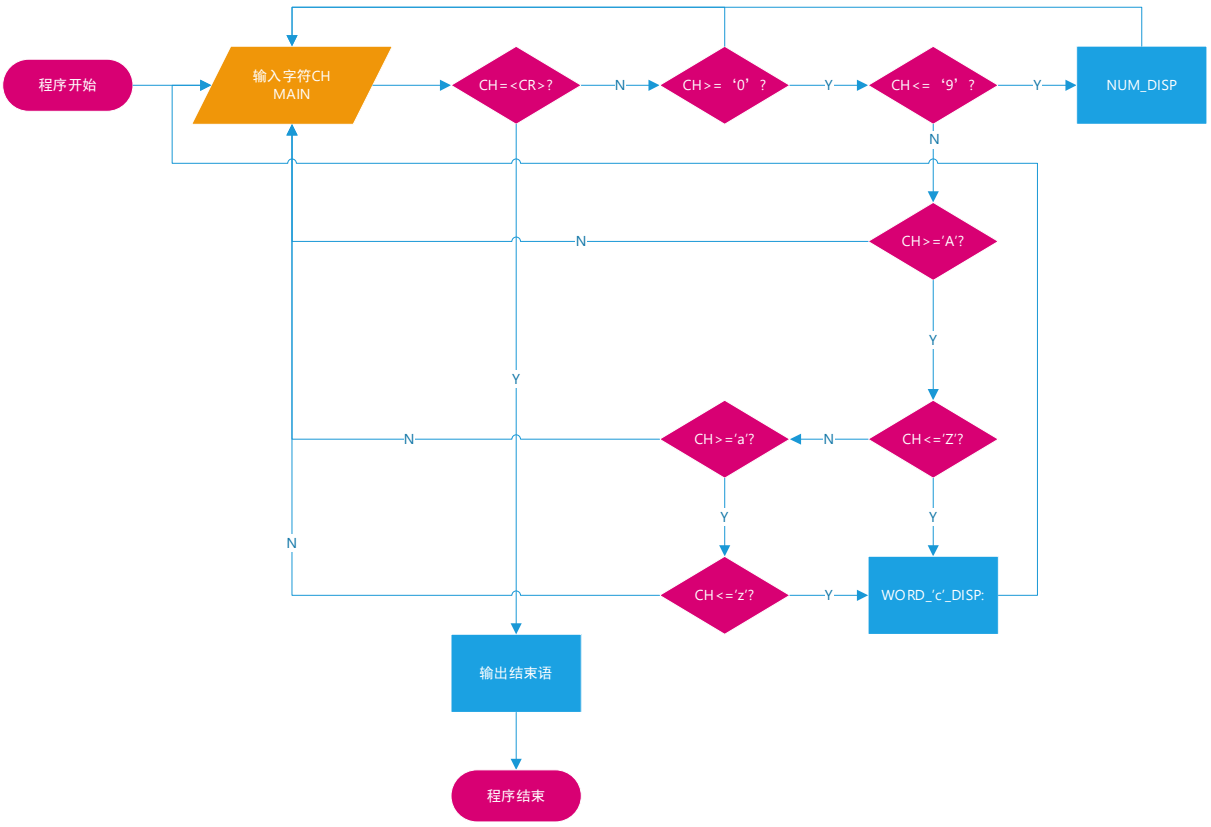
【第二个实验：循环程序实验】

（1） 实验流程图（具体见流程图文件）

程序设计思路流程图 Page 1



程序具体设计和执行过程流程图 Page2



（2） 实验源代码，其中代码已经添加注释

```
; 程序实现的相关功能：
; 数字输入输出样例：99
; 字母情况的输入输出样例：A c
; 其他情况的输入输出样例：$ <无字符>
DATA SEGMENT
MESG DB 'This is the end of the program.','$'
DATA ENDS
CODES SEGMENT
    ASSUME CS:CODES, DS:DATA
START:
MAIN:
    MOV  DL, 0AH
    MOV  AH, 2H
    INT  21H
    ;输出换行，在这里换行是为了便于观察输出情况和程序的执行情况
    MOV  AH, 1
    INT  21H
    ;系统等待输入一个字符，键入一个字符之后会自动转为 ASCII 值存入 AL 中
    CMP  AL,0DH
    ;如果输入字符为回车则跳到标识符 DIRCET_END 处执行
    JE   DIRCET_END
    CMP  AL,39H
    JBE  NUMBER                    ;如果=<9 则跳到标识符 NUMBER 处执行
    CMP  AL,41H                    ;（>9 成立）如果>=A 则跳到 WORD_处执行
    JAE  WORD_                     ;如上解析
    JMP  MAIN                      ;（<A 成立）继续输入字符
NUMBER:                            ;字符 1-9
    CMP AL, 31H                    ;判断是否>=1，匹配成功则进一步执行，否
则必然是除回车外的其他字符，进行返回字符重新输入
    JAE NUM_DISP
    JMP MAIN
NUM_DISP:                                  ;原样输出字符
    MOV DL, AL
    MOV AH, 2H
    INT 21H
    JMP MAIN
WORD_:                                     ;字符为英文单字 A-Z 或 a-z
```

```
    CMP AL, 5AH                         ;大于 A 的情况下与 Z 进行比较
    JBE WORD_DISP                       ;小于 Z 的情况：直接进入输出
    CMP AL, 61H                         ;(大于 Z 的情况下)与 a 进行比较
    JB MAIN                             ;小于 a 的情况：其他字符，跳转重新输出
    ;大于等于 a 的情况处理
    CMP AL, 7AH                         ;与 z 进行比较
    JBE WORD_DISP                       ;<=z 成立则跳转输出
    JMP MAIN                            ;不成立(>z)则说明是其他字符，重新输入
WORD_DISP:
    MOV DL, 20H
    MOV AH, 2H
    INT 21H
    MOV DL, 63H                         ;显示字符 c
    MOV AH, 2H
    INT 21H
    JMP MAIN
DIRCET_END:             ;字符为回车时，设计为在最后输出提示信息并结束程序
    MOV AX, DATA
    MOV DS, AX
    LEA DX, MESG
    MOV AH, 9
    INT 21H
    MOV AH, 4CH
    INT 21H
CODES ENDS
END START
```

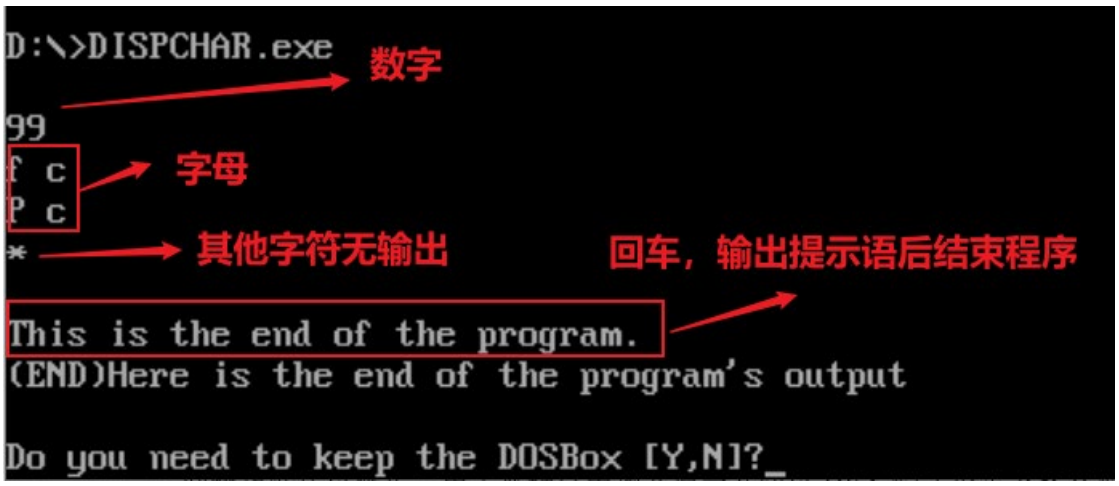
【编程逻辑简述】

这里的逻辑和实现过程比较直接。与流程图相对应，主要借用了**各个字符 ASCII 值的排序特点**和跳转指令，将代码的不同跳转处作以标记，由于从键盘中输入后会立即**以十六进制数 ASCII 码值**的形式转入到 AL 中。

借用这个特点便可以**逐层次**（回车->数字->字母）进行判断，利用 **CMP 和 JE/JBE 等指令配套**进行跳转，最终实现类似的循环式条件判断功能

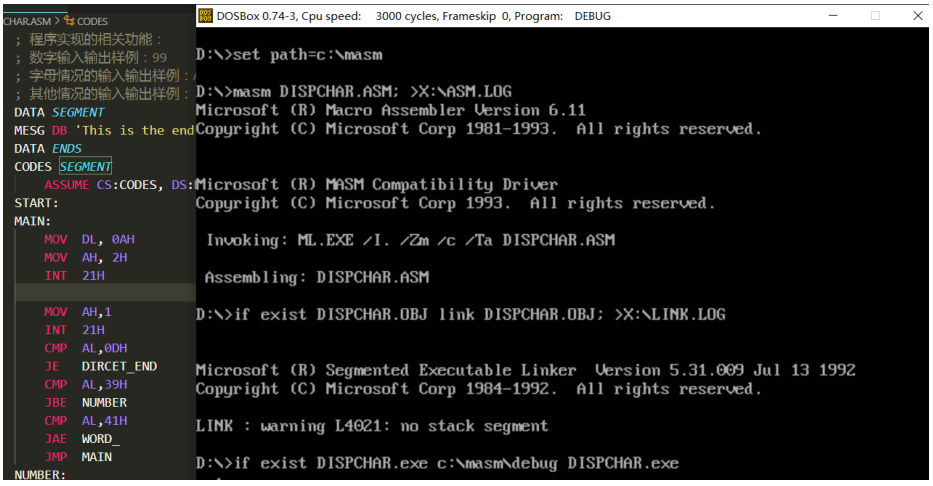
此外在程序的结束部分还利用了实验 1.1HelloWorld 的编程原型，实现回车后显示提示语并结束程序

【程序运行结果】



（3） 实验代码、过程、相应结果（截图）并对实验进行说明和分析：

在 VSCode 中利用同样的方法加载程序到 Debug 中，如下图所示

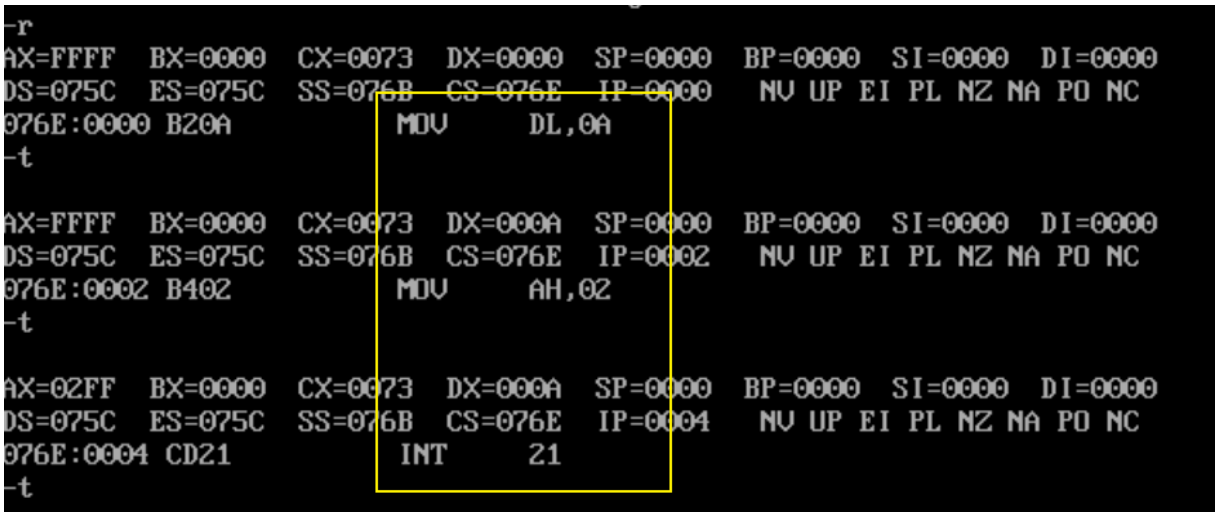


由于部分指令执行时需要执行较多的中间指令，在程序 Debug 过程中会部分使用-g 指令，类似于高级语言调试过程中的断点调试法

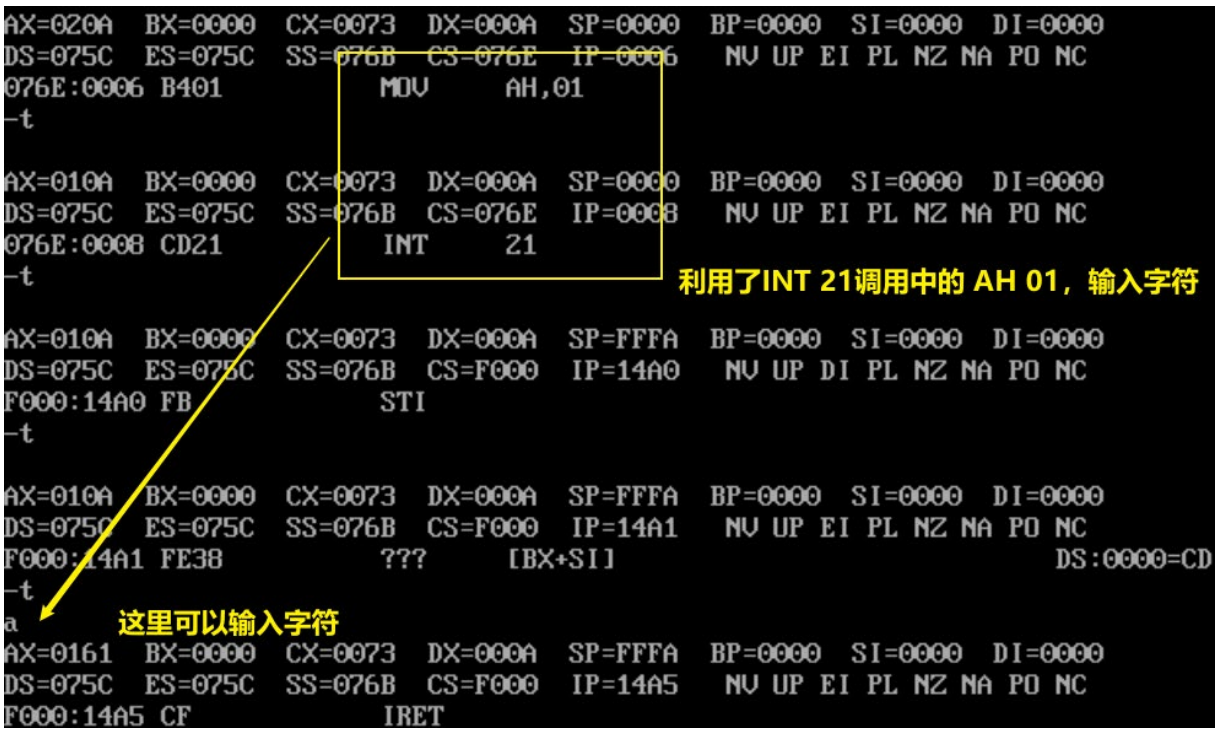
G 命令作用：执行汇编指令。

G 命令的使用方法是：G [=起始地址] [断点地址]，意思是从起始地址开始执行到断点地址。如果不设置断点，则程序一直运行到中止指令才停止。

首先是一般程序执行过程，如下图所示



这三段共同实现了换行操作，实际上是利用了 **INT 21H 调用中的字符输出功能**，其中，输出字符为 0A，即为**换行**



下面以输入字母、数字、输入回车、输入其他字符四种情况展开讨论不同输入情况下的执行过程

【输入字母】

如上图所示，输入字母 a 之后，根据 INT 21H 指令调用，可知 AL 会存入字母 ASCII 值对应的的十六进制值，如上图所示，AL=61H，恰好为 a 的 ASCII 值 97

根据流程图的执行逻辑，可知先与回车<CR>进行比较，如下图所示

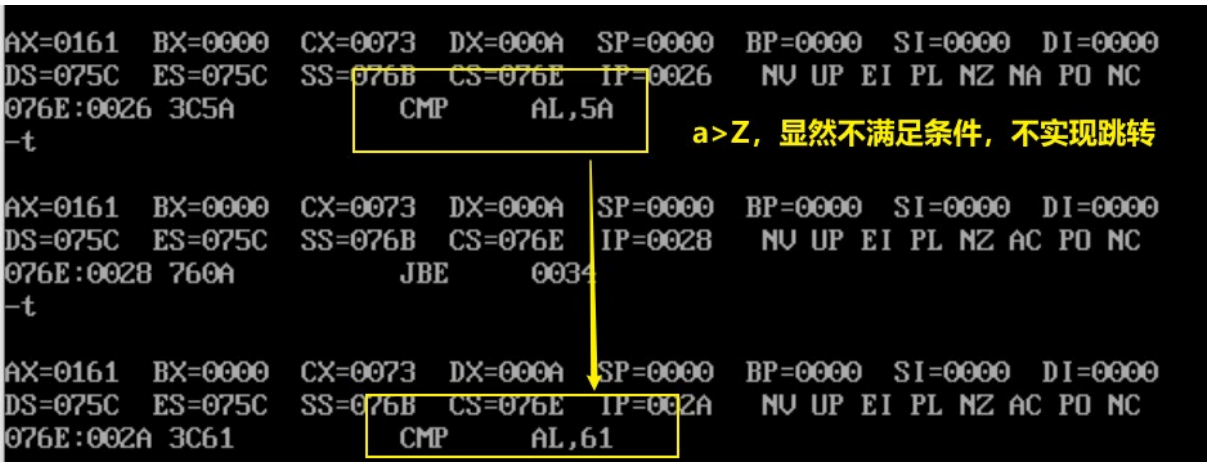
```
AX=0161  BX=0000  CX=0073  DX=000A  SP=0000  BP=0000  SI=0000  DI=0000
DS=075C  ES=075C  SS=076B  CS=076E  IP=000A   NV UP EI PL NZ NA PO NC
076E:000A 3C0D          CMP     AL,0D
-t

AX=0161  BX=0000  CX=0073  DX=000A  SP=0000  BP=0000  SI=0000  DI=0000
DS=075C  ES=075C  SS=076B  CS=076E  IP=000C   NV UP EI PL NZ AC PO NC
076E:000C 7434          JZ      0042
-t

AX=0161  BX=0000  CX=0073  DX=000A  SP=0000  BP=0000  SI=0000  DI=0000
DS=075C  ES=075C  SS=076B  CS=076E  IP=000E   NV UP EI PL NZ AC PO NC
076E:000E 3C39          CMP     AL,39
```

显然二者是不相等的，程序忽略跳转指令，直接进入下一步 CMP 执行，也就是与 9 进行比较，也就是说进入了数字匹配的流程。

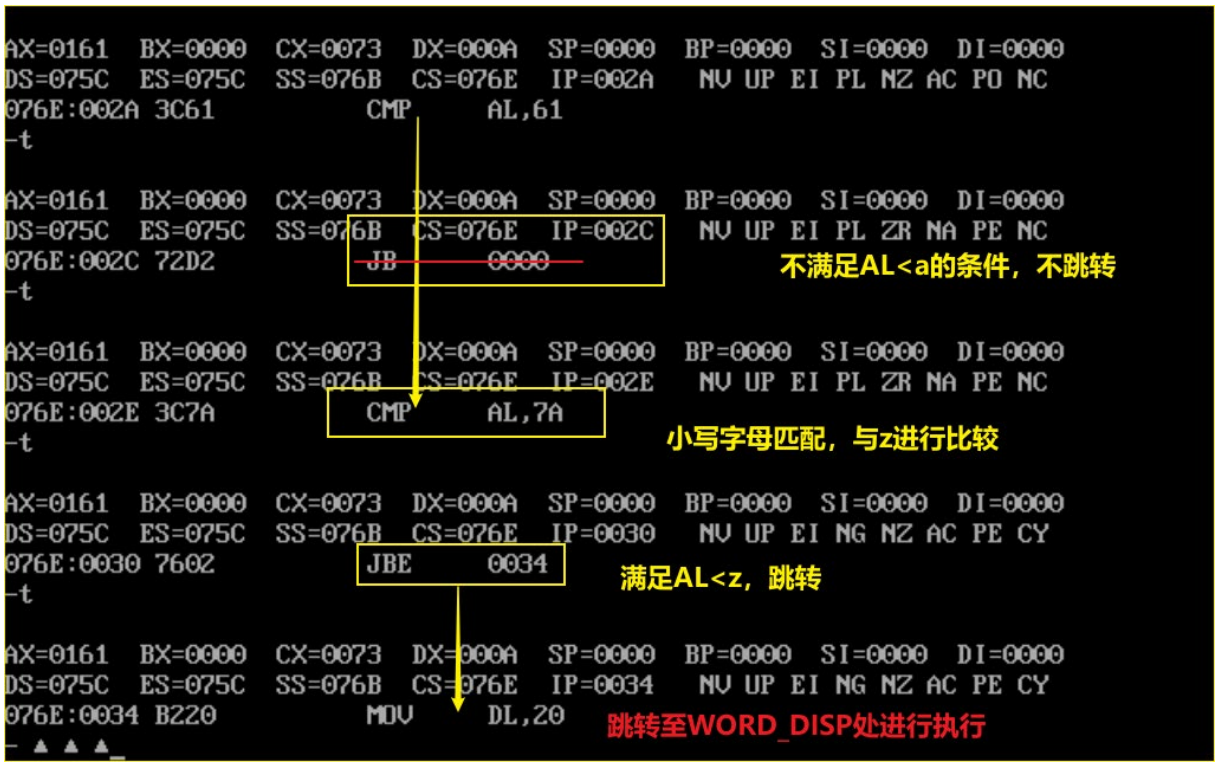
根据上图分析，可知进入了字母的匹配流程

显然这里的 AL 值大于 41（即为字符 A 的 HEX 值），从而如下图执行过程所示

可见这里开始在大于 A 的情况下和 Z 进行比较，如果小于 Z 就直接进行输出跳转如下图所示

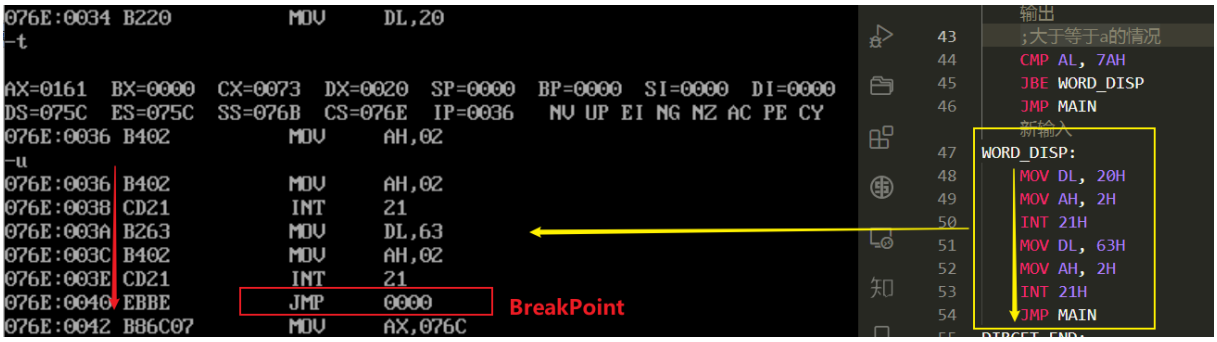


下面开始进行小写字母的匹配流程



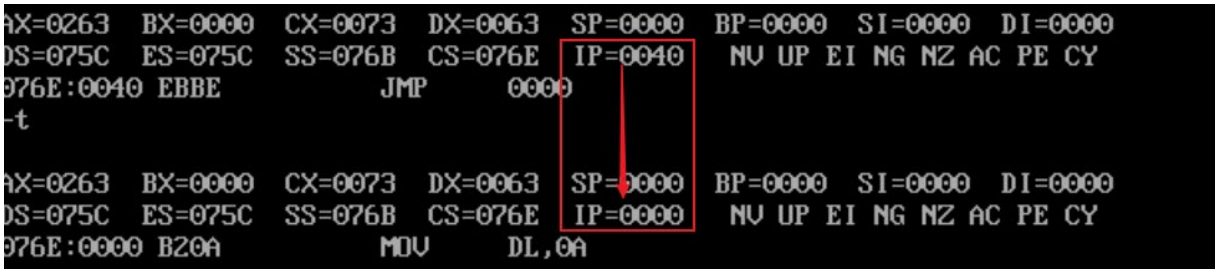
下面是字母输出，进行两次 INT 21H 调用，分别输出<空格>和字符'c'，这里比较简

单，不再赘述，利用-u 查看地址，**直接利用-g 直接执行到 JMP 处即可**，如下图所示



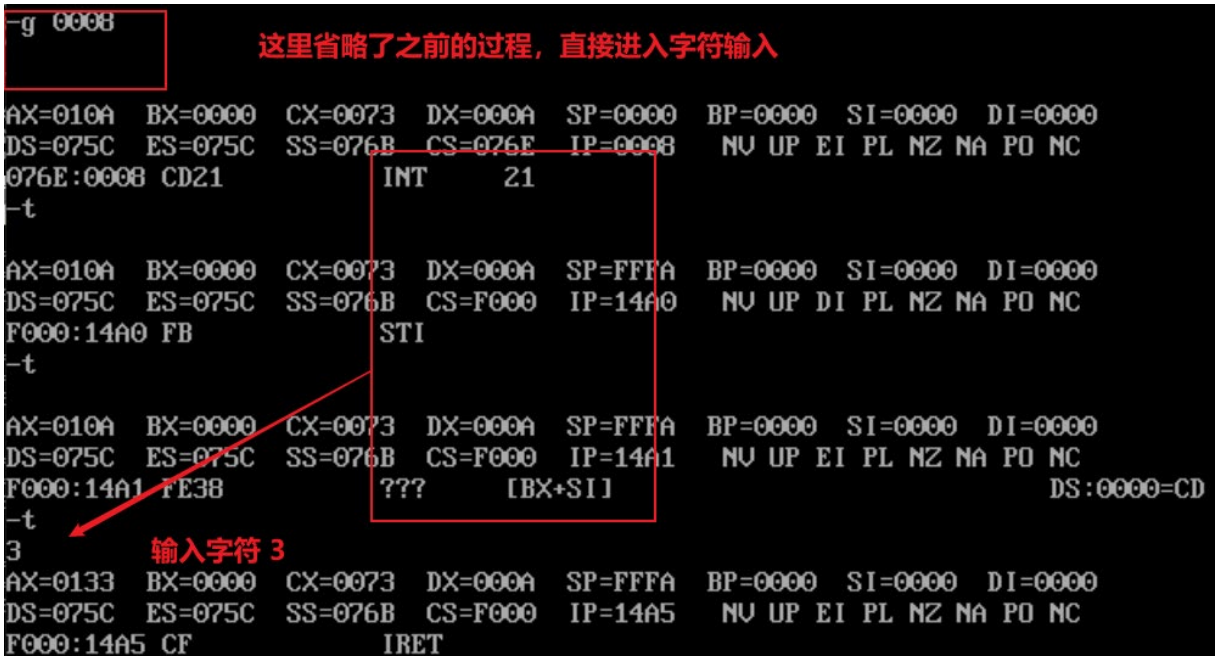
其中 20H 和 63H 分别表示空格和字符 c 的 Hex 值

执行完毕后无条件跳转至 MAIN 代码段进行字符的相关输入，重复后续过程，如下图所示

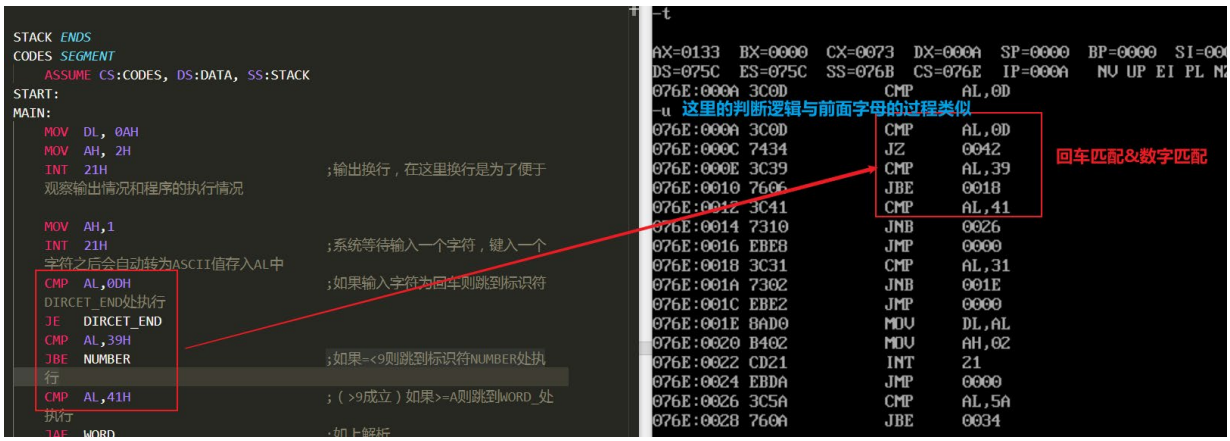


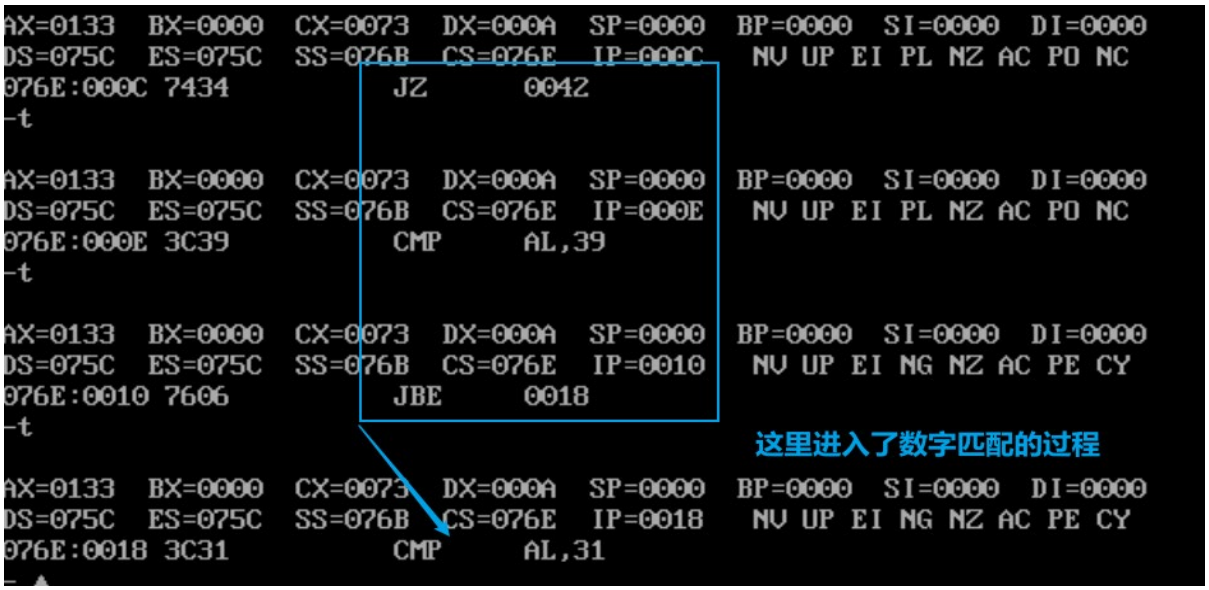
至此【字母输入】部分执行完毕

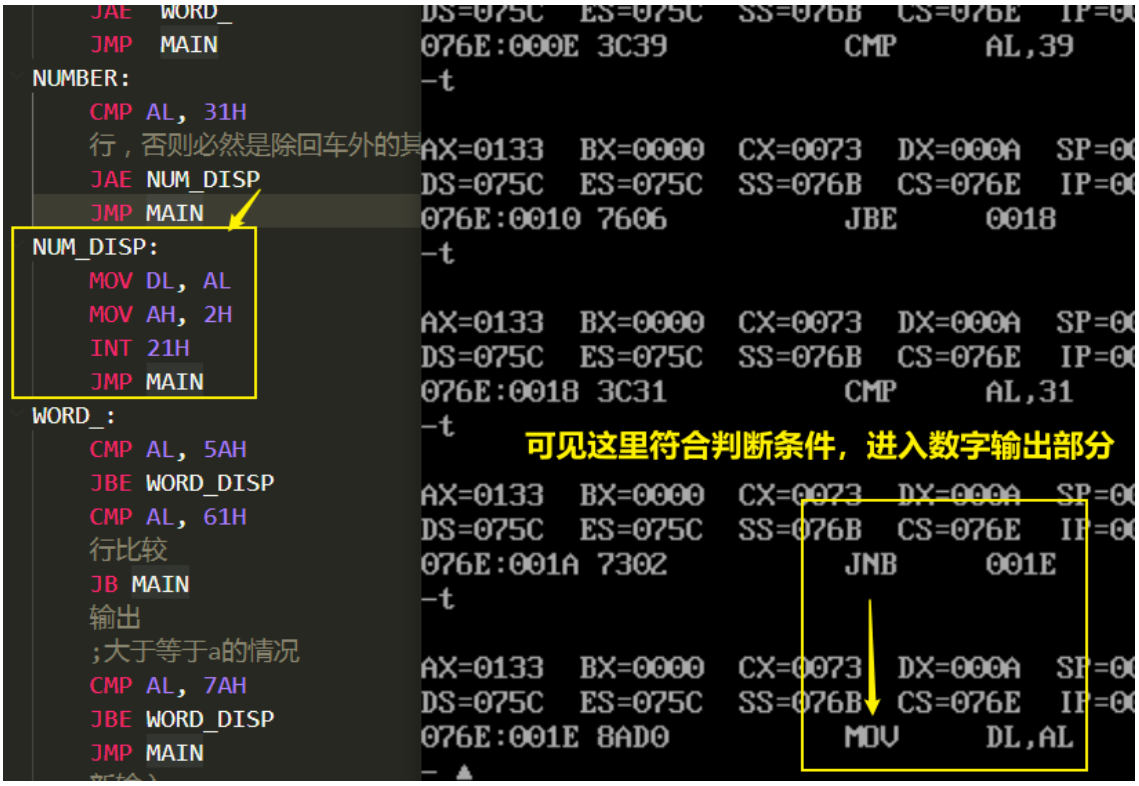
【数字输入】



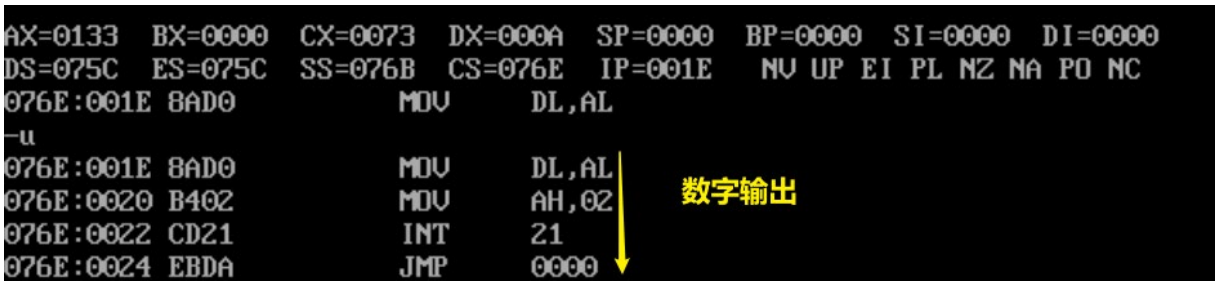
如图所示，这里输入了字符 3，后续的程序执行过程如下图所示


这里的判断逻辑与前面字母的过程类似
回车匹配&数字匹配

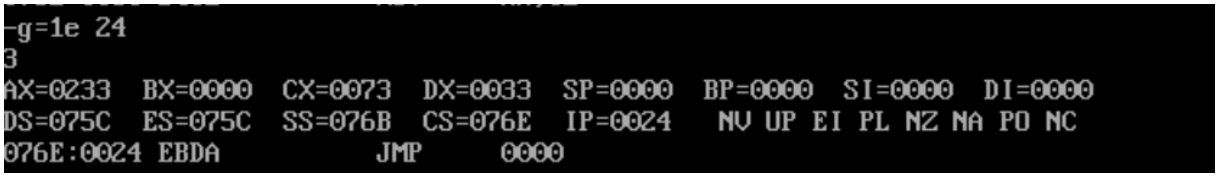

这里进入了数字匹配的过程


可见这里符合判断条件，进入数字输出部分

数字输出的程序段如下所示



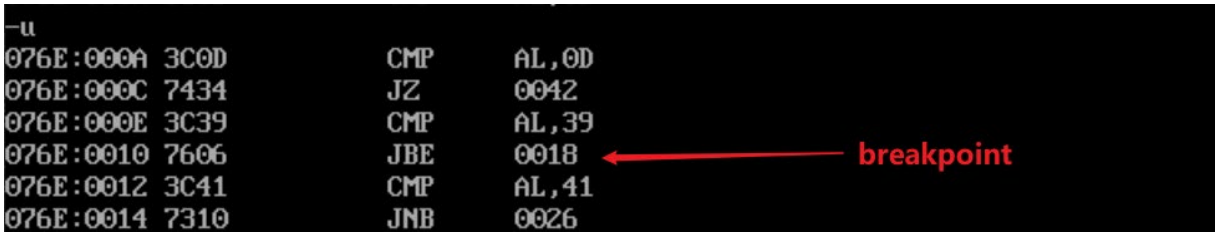
利用-g 进行批量执行，如下所示，可见成功输出原数字 3，并且在下一条指令要回到 MAIN 部分，也就是继续输入字符



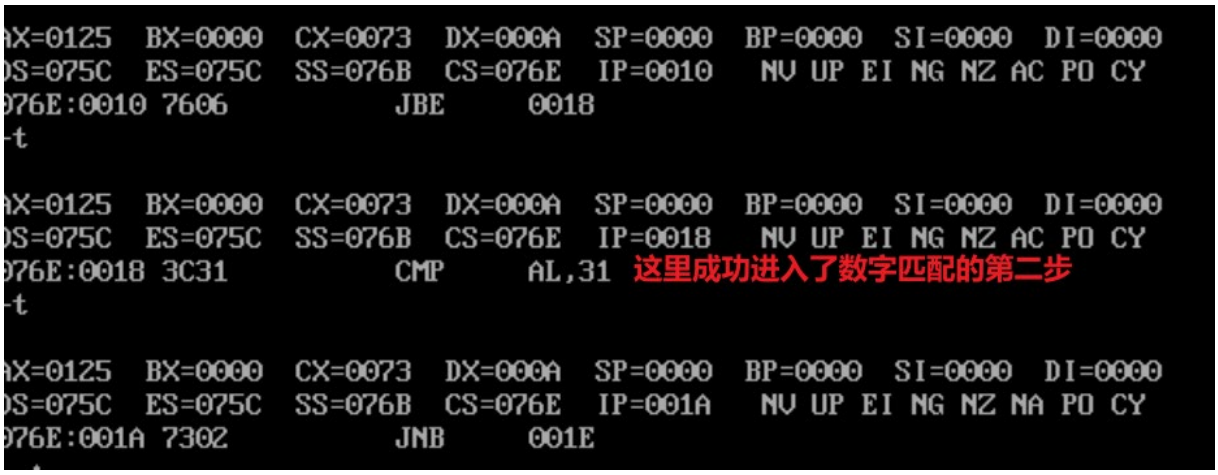
至此【数字输入】部分也分析完成了

【其他字符】

可见%所代表的 Hex 值是 25H，存入到 AL 中，显然满足数字匹配的第一步，考虑断点设为如下地址，-g 进行运行



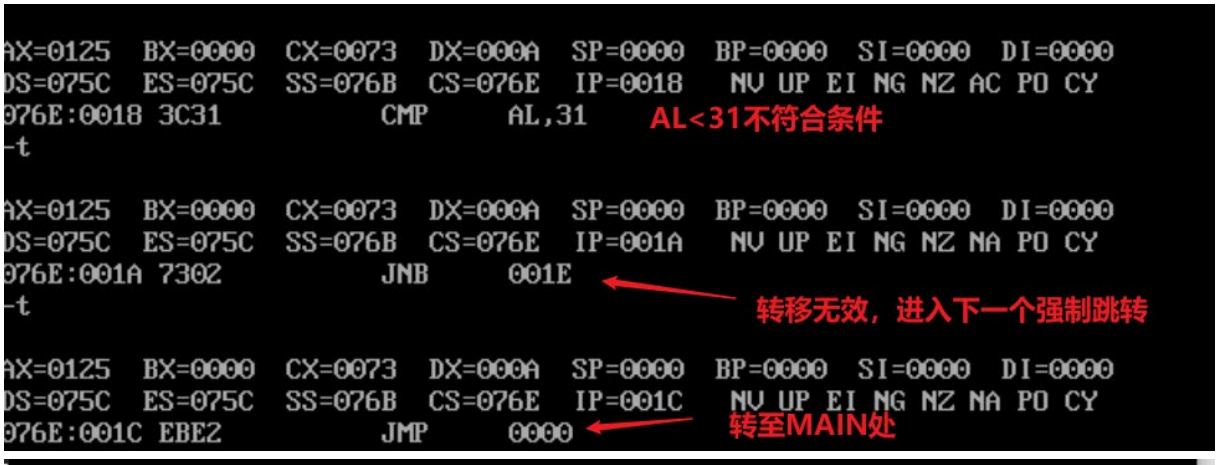
-g 10 执行，如下图所示

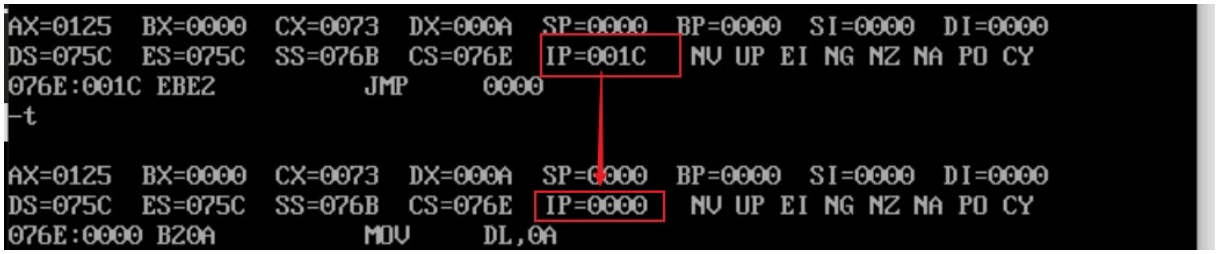



```
CMP AL, 31H
JAE NUM_DISP
JMP MAIN
```

这里判断是否>=1，匹配成功则进一步跳转到数字的输出部分，否则必然是除回车外的其他字符，就要跳转到进行返回字符重新输入的过程

显然这里不符合第二次匹配的要求，因而程序执行如下

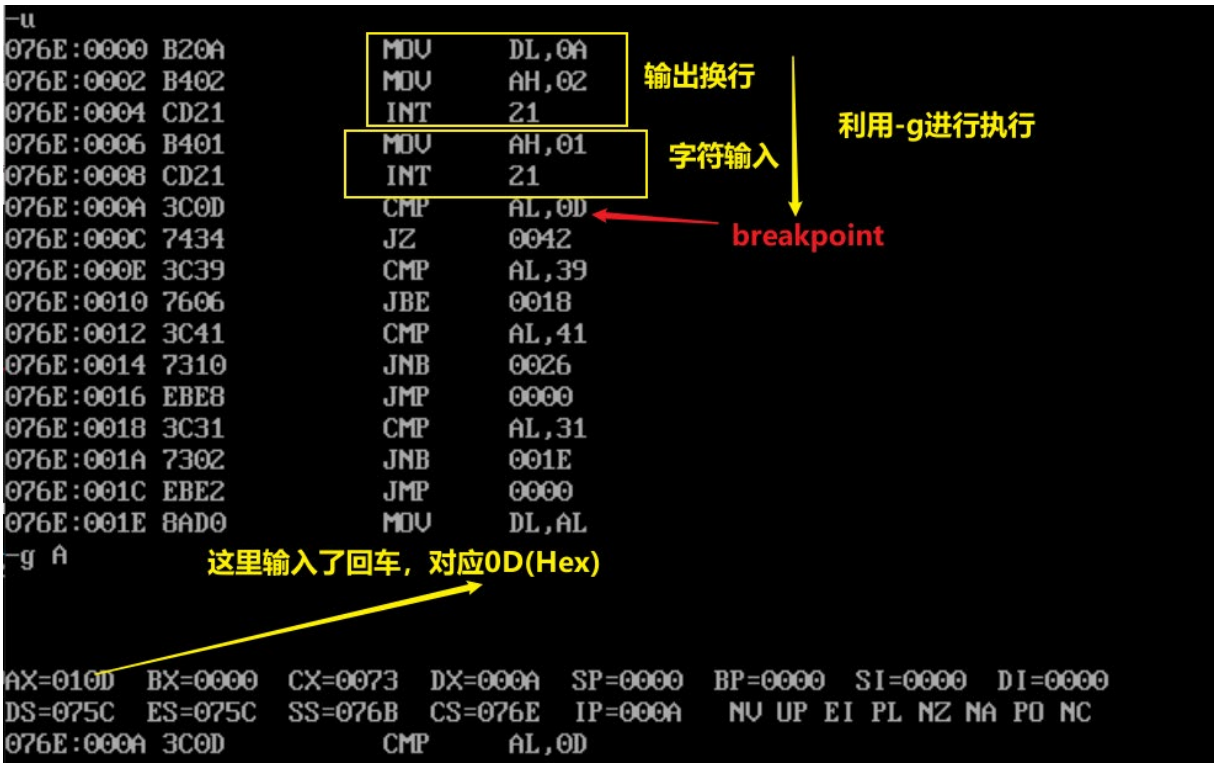


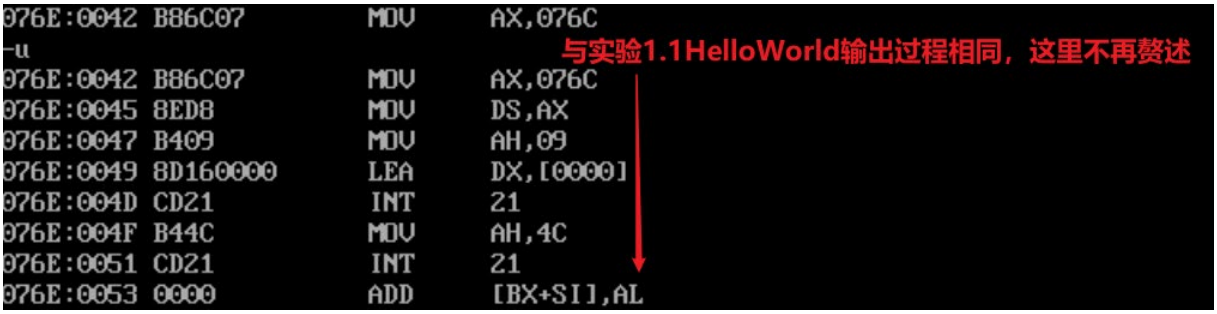


因而这里不输出字符，直接跳转至输入字符的程序部分。

至此【其他字符】部分也分析完成了。

【回车输入结束程序】





利用-g 缺省状态执行到底即可，如下图所示

```
-g
This is the end of the program.
Program terminated normally
```

可见这里输出的提示信息与下面的 DATA 段对应

```
DATA SEGMENT
MESG DB 'This is the end of the program.','$'
DATA ENDS
```

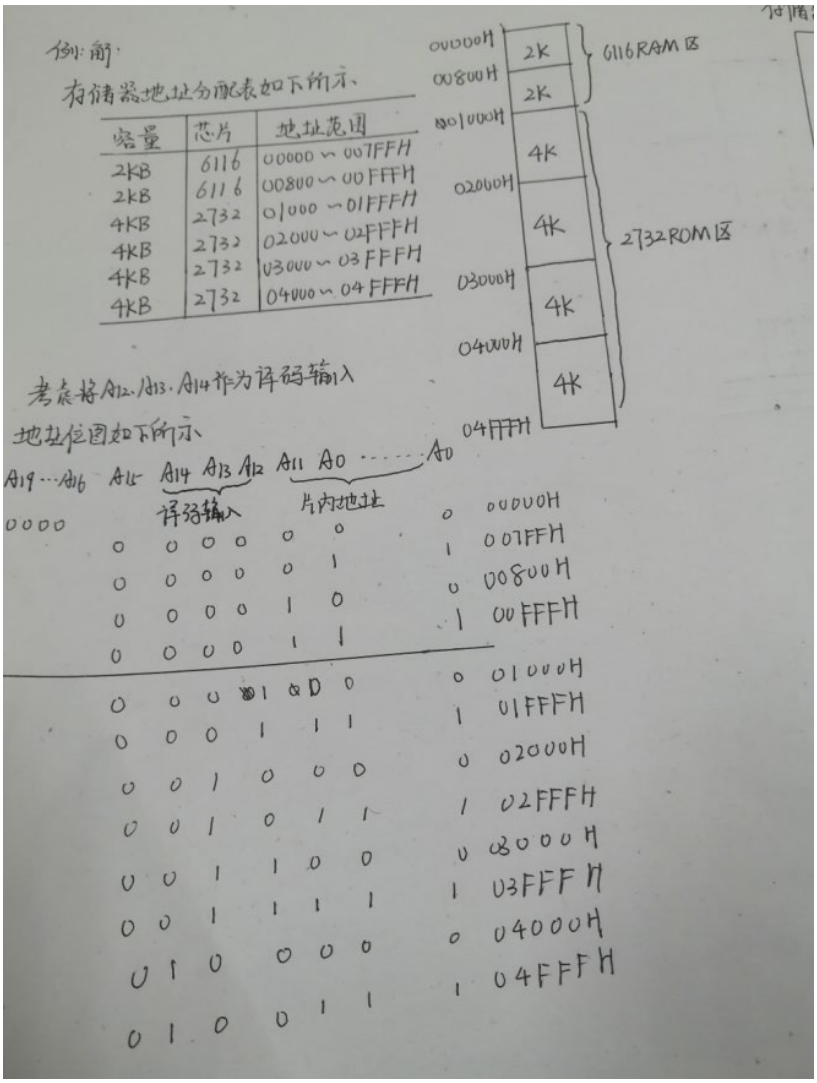
至此所有的输入代表情况分析完成，整个程序也分析完成了。

【课程附加问题】

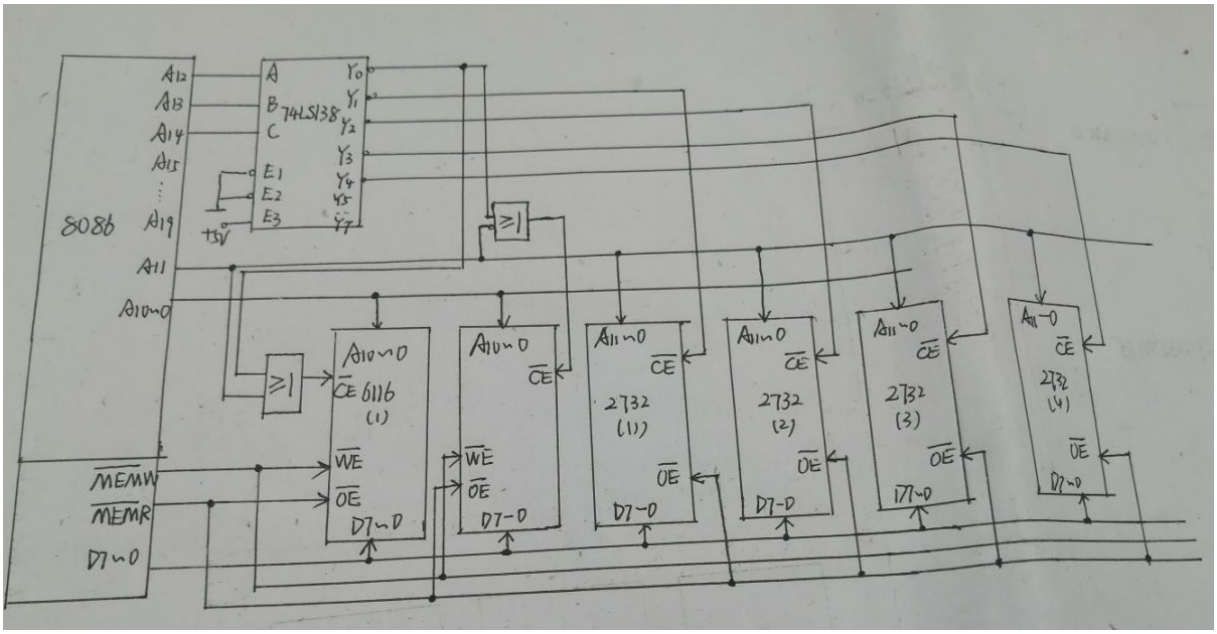
1. 存储体和总线连接找出一个例题并复现解答过程

设某微机系统地址线为 20 条，数据线为 8 条，采用 6116 设计 4KBRAM，起始地址为 00000H；采用 2732 设计 16KBROM，起始地址为 01000H，列出存储器地址分配表，画出硬件连接图

分析如下图所示



存储器连接如下图所示



2. 分析位宽的作用并举例说明。

显卡的性能表现主要体现在显存位宽，显存频率，显存容量。在这三个方面中显存位宽影响着渲染等效果的好坏，并且影响巨大。

显卡位宽指的是显存位宽，即显存在一个时钟周期内所能传送数据的位数，位数越大则瞬间所能传输的数据量越大，这是显存的重要参数之一

位宽是显卡速度高低的衡量标准之一，位宽属性参数比显存参数更能体现显卡的性能，相当于我们常说的马路车道数。

无论显存怎么改变，出发点都是因为对带宽的不断渴求，显存带宽一直是显卡一个很难攻破的瓶径所在，显存位宽在另一个方面决定了显存带宽的性能，显存带宽是指图形芯片与显存之间一次可读入的数据传输量，它是决定显卡性能和速度的主要因素，其计算公式为：显存带宽＝工作频率×显存位宽/8。以 Radeon 9600 和 Radeon 9600SE 为例，二者的显存频率都为 400MHZ，Radeon 9600 的位宽为 128Bit，其带宽就为 400×128/8=6.4G/s，而 Radeon 9600SE 的位宽只有 64Bit，其带宽也只有 400×64/8=3.2G/s。从这里我们很清楚的看到，显存位宽对显存的带宽起着举足轻重的作用，因为在相同频率下，64 位显存的带宽只有 128 位显存的一半（理论上，相同频率下的 64 位显卡性能只有 128 位显卡的一半），当遇到大量像素渲染工作时，因为显存位宽的限制会造成显存带宽的不足，最直接的后果就是导致传输数据的拥塞，速度明显下降，这也就是为什么 Radeon 9600SE 的性能无法与 Radeon 9600 相提并论的原因，所以在选择显卡的时务

必要关注显存位宽的大小。

3. 4K 对齐的含义

硬盘中文件保存的基本单元是扇区，不管文件大小，都要占用一个扇区的空间。机械硬盘一个扇区是 512 字节，固态使盘一个扇区是 4K 字节。

微软操作系统常用的 NTFS 格式，默认的扇区大小也是 512 字节，并且规定前 63 个扇区是保留的，也就是前 31.5K 字节的空间是不用的，数据从第 64 个扇区开始保存。这对于机械硬盘不是什么问题，但对于固态硬盘来说，数据保存从一开始就出现错位，一块数据横跨两个扇区的情况变得相当普遍，这意味着读写这块数据需要读写两个扇区，而闪存读写次数是受限制的，过多无意义的读写对固态硬盘的性能和寿命会造成很大的损伤。

由于 SSD 硬盘的读写机制特性，写入数据时，以 8 个扇区(4096KB)为一基本存储单元。写满后继续下一个 4K 区块写操作，若 SSD 硬盘没有 4K 对齐处理，数据写入会 4K“超界”，读取数据时会在超界处，造成二次往复读取，读取数据时间增加，读写效率降低。

“4K 对齐”指的是符合“4K 扇区”定义格式化过的硬盘，并且按照“4K 扇区”的规则写入数据。因为随着硬盘容量不断扩展，使得之前定义的每个扇区 512 字节不再是那么的合理，于是将每个扇区 512 字节改为每个扇区 4096 个字节，也就是现在常说的“4K 扇区”。随着 NTFS 成为了标准的硬盘文件系统，其文件系统的默认分配单元大小也是 4096 字节，为了使簇与扇区相对应,，即使物理硬盘分区与计算机使用的逻辑分区对齐，保证硬盘读写效率，所以有了“4K 对齐”概念。

【实验心得】

本次实验以输入输出为主题进行展开，并利用汇编语言进行简单的编程。

首先第一个实验主题为理解程序的执行过程，同时为第二个程序的编写做铺垫。

第一个实验的目的是输出 AL 中的两位十六进制数，通过对程序的分析和理解，我逐渐感受到了在汇编中条件跳转的处理逻辑。目前我所理解的条件跳转逻辑都是利用 CMP 指令进行比较，然后后面跟随 JBE/JE/JAE……等条件跳转指令实现的，在跳转时，汇编还有在程序对应的跳转处添加标号的操作，这一点有点像 C/C++中的 Switch-Case 语句规则。总之，通过学习第一个实验程序，该程序让我逐渐了解了汇编程序中的“智能化”处理手段，为后续的灵活编程打下了基础。

此外，在第一个实验中我也了解到了 INT 21H 调用的编程手段，通过为 AH 寄存器赋予不同的值，同时使用 INT 21H 进行调用，可以实现各种功能，好像已经封装好的库函数一样，在使用时直接调用就好了。比如最开始实验 1.1 中出现的 MOV AH, 9 调用和这两次实验都用到的 MOV AH,2 调用，我理解为类似 C/C++中的 scanf 和 printf 函数，只不过略有区别的一点在于 scanf 函数会在 shell 中“等待”输入，回车之后才一并读入

到程序（寄存器）中。这一点感觉与 MOV AH,1 的输入调用略有不同，汇编的输入调用在目前我碰到的情况都是输入一个字符之后立即读入，没有回车再读入的情况。

在第一个实验中，我逐步了解到了汇编相关的分支跳转、INT 21H 相关调用。而第二个实验实际上是考察对第一个实验的理解应用，在分析问题的时候，我借鉴了第一个程序的思路，与流程图相对应，借用了**各个字符 HEX 值的排序特点**和跳转指令，将代码的不同跳转处作以标记，由于从键盘中输入后会立即以 HEX 码（不是 ASCII 码）的形式转入到 AL 中。

借用这个特点便可以**逐层次**（回车->数字->字母）进行判断，利用 **CMP 和 JE/JBE 等指令配套**进行跳转，最终实现类似的循环式条件判断功能

此外在程序的结束部分还结合利用实验 1.1HelloWorld 的编程原型，实现回车后显示提示语并结束程序，实现了在汇编中的输出信息优化。

总体回顾这个程序，我发现程序具有模块化的设计特点，比如各个部分的标记对应表示其具有什么样的功能，这也和流程图是对应的。

第二个实验是我第一次编写出完整的汇编语言程序。虽然在思路上与第一个程序有部分借鉴，但是独立完成并排除各种 Bug 之后运行成功带给我的成就感仍然是很强的。在后续的输出优化过程中，也让我明白了细节的重要性，无论对于哪一门语言，缜密的逻辑和细节的重视都是至关重要的。

除此回顾两次实验流程，我也初步学习了 Visio 绘制流程图软件的使用，增强了我的综合实践能力。总之，这次实验是对我综合能力的一个显著提升，期望下次汇编实验能带给我更多的收获。