# 華中科技大学

# 课 程 实 验 报 告

**课程名称：** 汇编语言程序设计实验

**实验时间：** 2017-5-14，19：00-21：30　　**实验地点：** 南一楼 804 室 08 号实验台

**指导教师：** 张　勇

**专业班级：** 七校联合二学位 计算机科学与技术 201502 班

**学　　号：** U201516431　　**姓　　名：** 刘云中

**同组学生：** 无　　**报告日期：** 2017 年　5　月 14 日

## 原创性声明

本人郑重声明：本报告的内容由本人独立完成，有关观点、方法、数据和文献等的引用已经在文中指出。除文中已经注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品或成果，不存在剽窃、抄袭行为。

特此声明！

学生签名：

日期：2017.5.14

## 成绩评定

| 实验完成质量得分（70 分）（实验步骤清晰详细深入，实验记录真实完整等） | 报告撰写质量得分（30 分）（报告规范、完整、通顺、详实等） | 总成绩（100 分） |
|---|---|---|
| | | |

指导教师签字：

日期：

# 目录

# 汇 编 语 言 程 序 设 计 实 验 报 告

## 1 实验目的与要求

本次实验的主要目的与要求有下面 6 点，所有的任务都会围绕这 6 点进行，希望大家事后检查自己是否达到这些目的与要求。

(1) 掌握汇编源程序编辑工具、汇编程序、连接程序、调试工具 TD 的使用；

(2) 理解数、符号、寻址方式等在计算机内的表现形式；

(3) 理解指令执行与标志位改变之间的关系；

(4) 熟悉常用的 DOS 功能调用；

(5) 熟悉分支、循环程序的结构及控制方法，掌握分支、循环程序的调试方法；

(6) 加深对转移指令及一些常用的汇编指令的理解。

## 2 实验内容

任务 1：《80X86 汇编语言程序设计》教材中 P31 的 1.14 题。

要求：(1) 直接在 TD 中输入指令，完成两个数的求和、求差的功能。求和/差后的结果放在(AH)中。

(2) 请事先指出执行指令后(AH)、标志位 SF、OF、CF、ZF 的内容。

(3) 记录上机执行后的结果，与（2）中对应的内容比较。

(4) 求差运算中，若将 A、B 视为有符号数，且 A>B, 标志位有何特点？ 若将 A、B 视为无符号数，且 A>B, 标志位又有何特点？

任务 2：《80X86 汇编语言程序设计》教材中 P45 的 2.3 题。

要求： (1) 分别记录执行到“MOV CX，10”和“INT 21H”之前的(BX), (BP),(SI),(DI)各是多少。

(2) 记录程序执行到退出之前数据段开始 40 个字节的内容，指出程序运行结果是否与设想的一致。

(3) 在标号 LOPA 前加上一段程序，实现新的功能：先显示提示信息“Press any key to begin!”，然后，在按了一个键之后继续执行 LOPA 处的程序。

# 3 实验过程

## 3.1 任务 1

### 3.1.1 实验步骤

1. 将课本 P31 页 1.14 题中的原码全部换算为补码，然后准备上机实验环境。

2. 不带任何参数直接启动 TD，进入 TD 的自定义调试模式，方可直接运行我们直接输入的代码。在 TD 代码窗口的当前光标下，直接键入汇编指令，即可弹出汇编指令的输入窗口。每输入一行，回车，当前行的汇编指令即更新为我们所输入者。光标自动移到下一行，以供继续输入。

3. 输入后，先确认 CS:IP 指向的是自己输入的第一条指令的位置，然后将光标移到输入的最后一条指令处，按 Alt+R 呼出“Run”菜单，选择“Go to cursor”，让程序运行到光标处。这时，寄存器窗口就会直接显示结果。

4. 按照第 2 步和第 3 步的方法，对 3 道小题进行检验如下：

表格 3.1 1 要执行的汇编指令与预期结果

| 小题号 | 汇编指令 | 预计的 AH 值 | 预计的标志位值 | |
|---|---|---|---|---|
| 1 | MOV AH, 0110011B<br>MOV AL, 1011010B<br>ADD AH, AL | 08DH | SF=1 | OF=1 |
| | | | CF=0 | ZF=0 |
| 2 | MOV AH, 11010111B<br>MOV AL, 10100011B<br>ADD AH, AL | 07AH | SF=0 | OF=1 |
| | | | CF=1 | ZF=0 |
| 3 | MOV AH, 1100101B<br>MOV AL, 1011101B<br>ADD AH,AL | 0C2H | SF=1 | OF=1 |
| | | | CF=0 | ZF=0 |

5. 输入 MOV AH,1OH；MOV AL,-5H；SUB AH,AL；观察标志位特点；
输入 MOV AH,0FFH；MOV AL,-5H；SUB AH,AL；观察标志位特点。

### 3.1.2 实验记录与分析

1. 实验环境条件（我的笔记本电脑）：Intel Core™ i7-5500U 2.40GHz，8GB 内存；WINDOWS 10 64 位下 DOSBox 0.72； MASM 6.0，TD.EXE 5.0。代码编辑工具为 Notepad ++。

2. 进入实验环境，启动 TD，开始键入命令。第一小题的操作过程如下图所示：

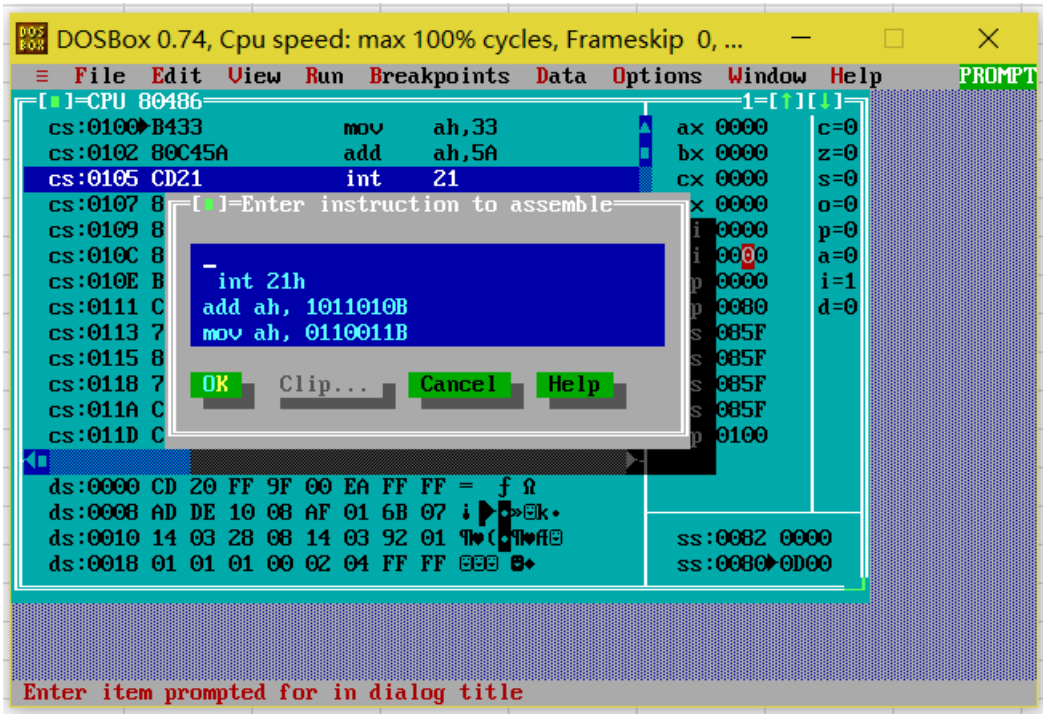


图 3.1 1　第一小题：输入汇编代码

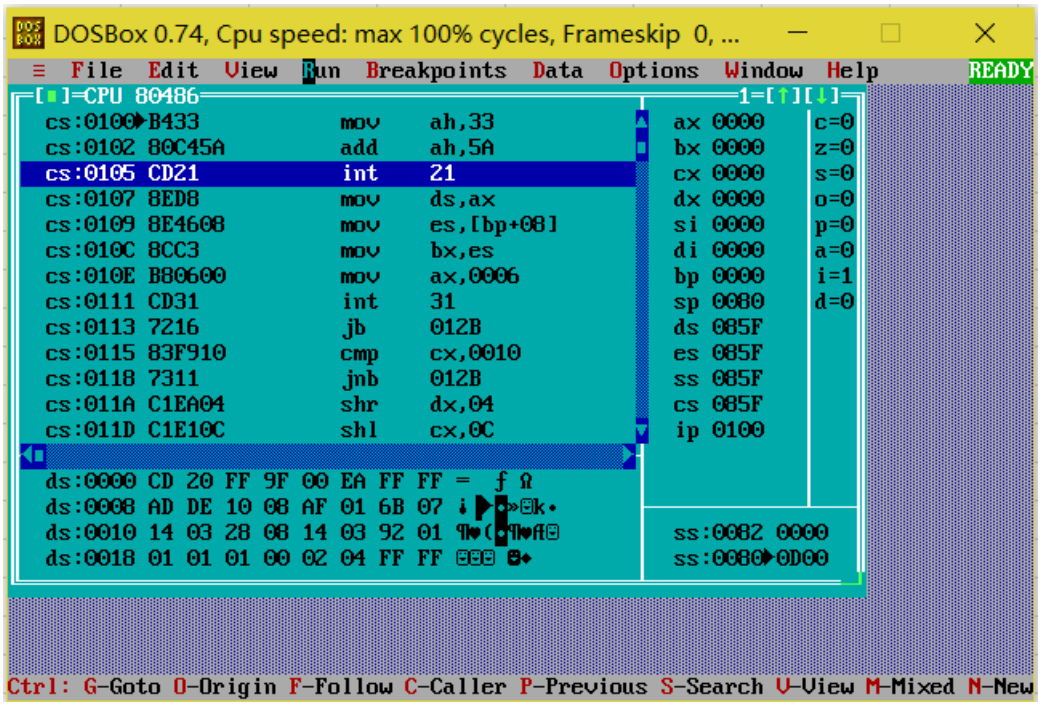


图 3.1 2　第一小题：所有代码输入完毕

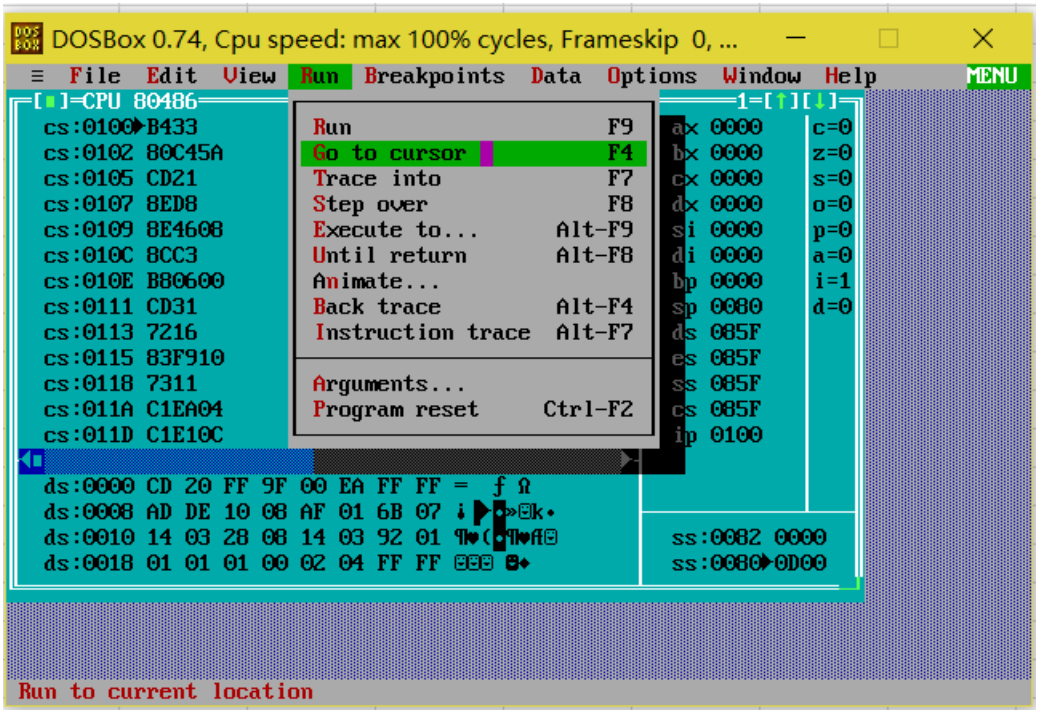


图 3.1 3 第一小题：设置“运行到光标处”

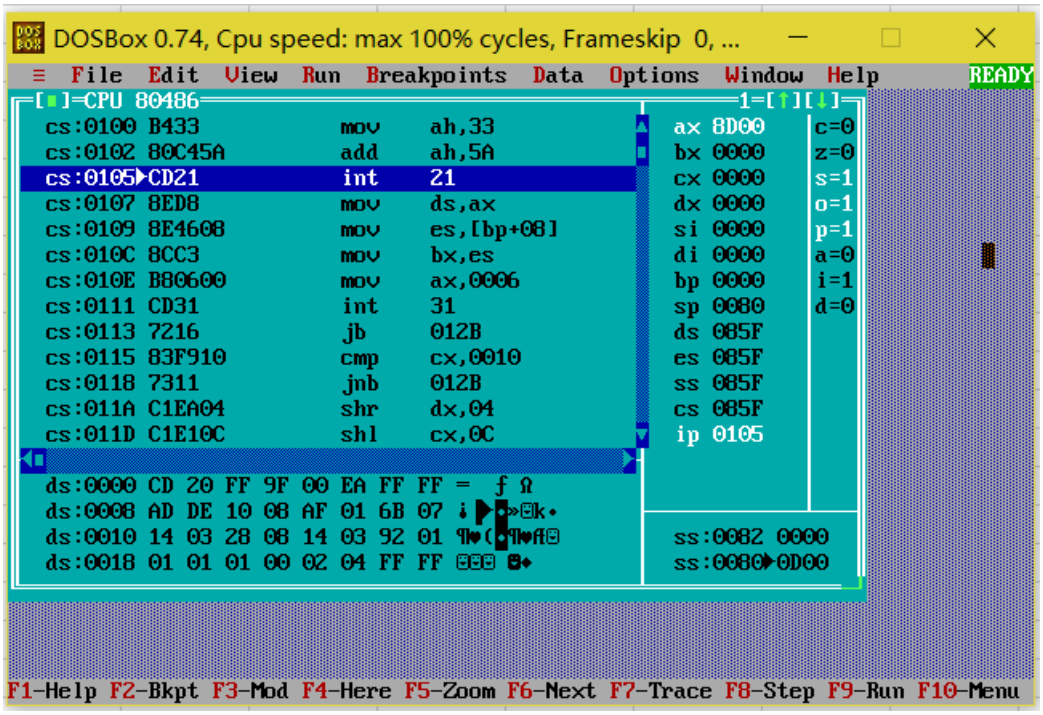


图 3.1 4 第一小题：运行结果

3.接下来按照同样的方法检验第二和第三小题，结果如下：

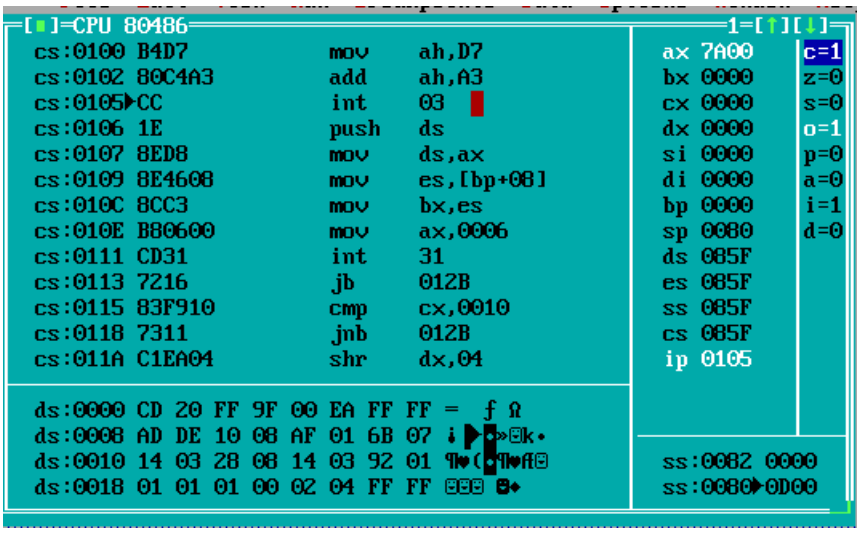


图 3.1 5 第二小题结果

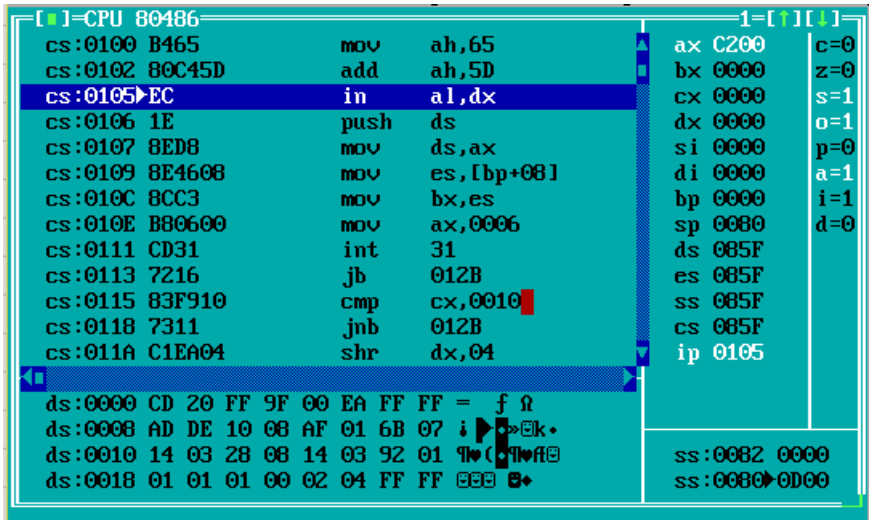


图 3.1 6 第三小题结果

**由此，原假设得证。**

4. 输入 MOV AH,10H；MOV AL,-5H；SUB AH,AL 后，执行结果如下图。由此可说明，标志位符合以下特点：

| | | | |
|---|---|---|---|
| SF=0 | OF=0 | CF=1 | ZF=0 |

这里选用的操作数 A＞B，相减时不会产生溢出，故 OF 值没有实质性影响。A 与 B 均为有符号数，二者相减，CF=1，产生了借位。

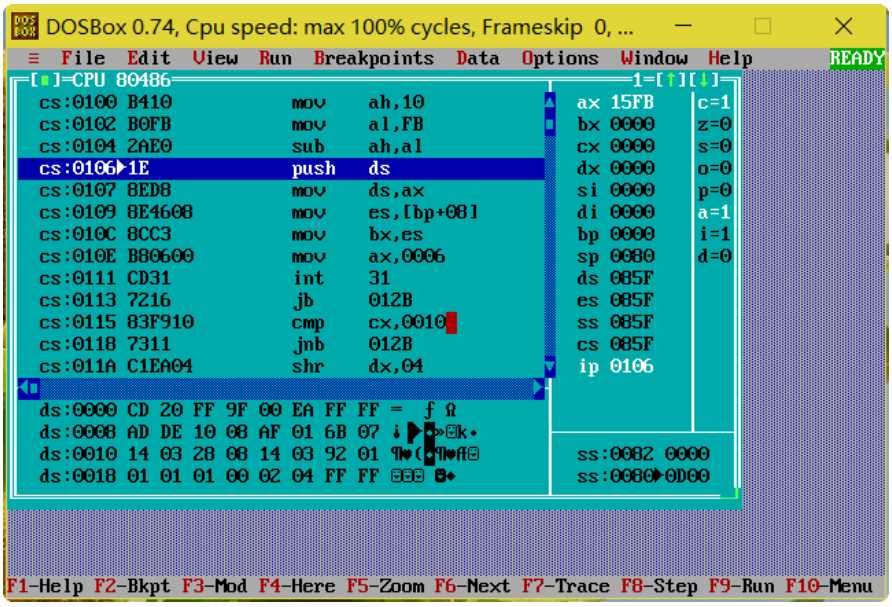


图 3.1 7　MOV AH,10H；MOV AL,-5H；SUB AH,AL 的执行结果

5. 同理，输入 MOV AH,0FFH；MOV AL,-5H；SUB AH,AL 后，执行结果如下图。由此可说明，标志位符合以下特点：

| | | | |
|---|---|---|---|
| SF=0 | OF=0 | CF=0 | ZF=0 |

这里选用的操作数 A＞B，相减时不会产生溢出，故 OF 值没有实质性影响。A 与 B 均为有符号数，二者相减，CF=0，并未产生借位。

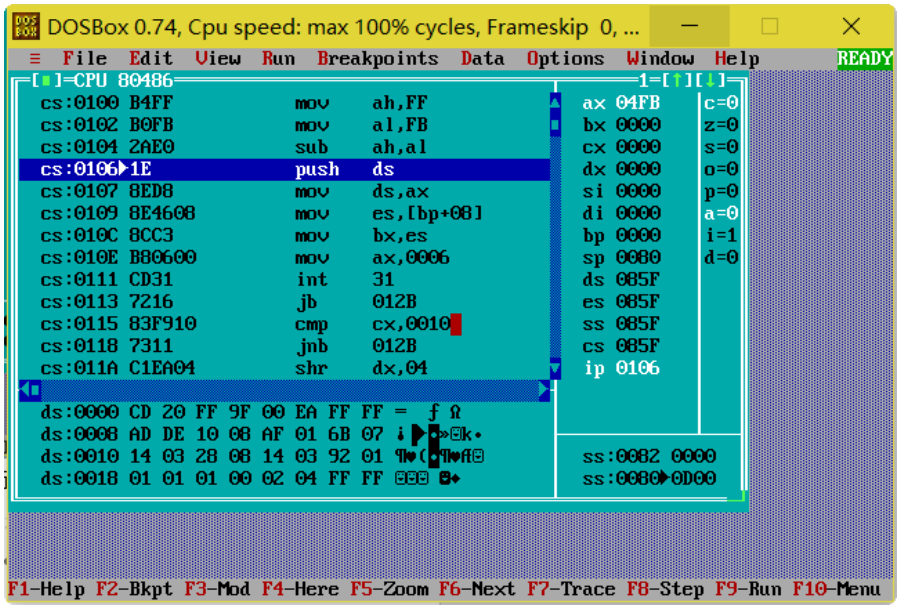


图 3.1 8　MOV AH,0FFH；MOV AL,-5H；SUB AH,AL 的执行结果

## 3.2 任务 2

### 3.2.1 实验步骤

1. 将课本上的源码完整录入计算机中，命名为 TASK2.ASM，并用 MASM 和 LINK 进行编译链接，得到可执行文件 TASK2.EXE。

2. 用 TD 打开 TASK2.EXE，确保 CS:IP 寄存器的值指向程序的首条语句所在地址，然后按 F7，进行单步运行。执行到“MOV CX,10”之前（即光标正好指向该语句）时，记录下(BX),(BP),(SI),(DI)这四个寄存器的值；同理，也记录下执行到“INT 21H”之前，上述寄存器的值。

除了单步运行，还有一种更便捷的方法，即设置断点。直接将光标定位到需要设置断点的代码处，按下 F2 即可。这时将光标移开该行，则可看见该行变为红色。要取消断点设置，同样按 F2。设置断点后，只要按 F9 执行程序，程序即会在每一个断点处暂停；反复按 F9，可遍历所有的断点，直至程序结束。

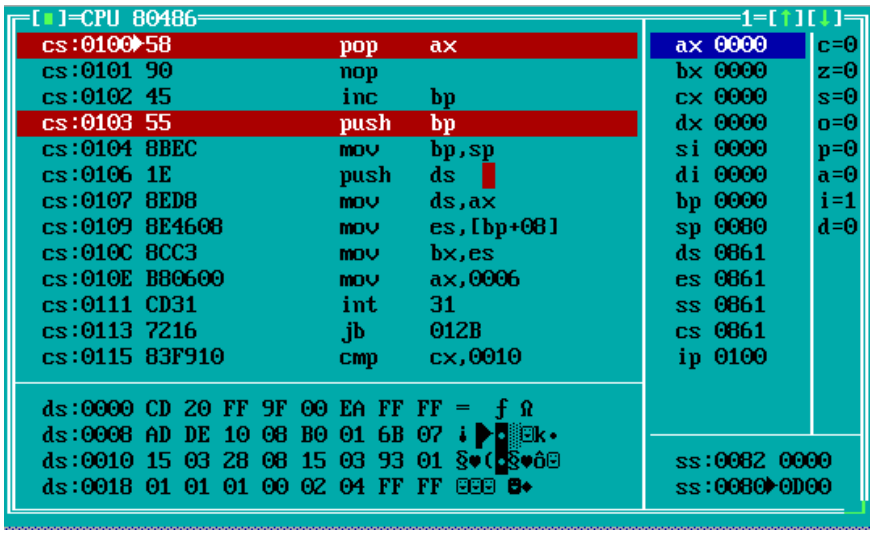


图 3.2 1 设置过断点的行在图中显示为红色（注意，图中光标在寄存器窗口）

3. 分别记录程序执行时和退出之前数据段头 40 个字节的内容。一般地，数据段的内容会在 TD 主界面的数据窗口中显示出来，如图所示：

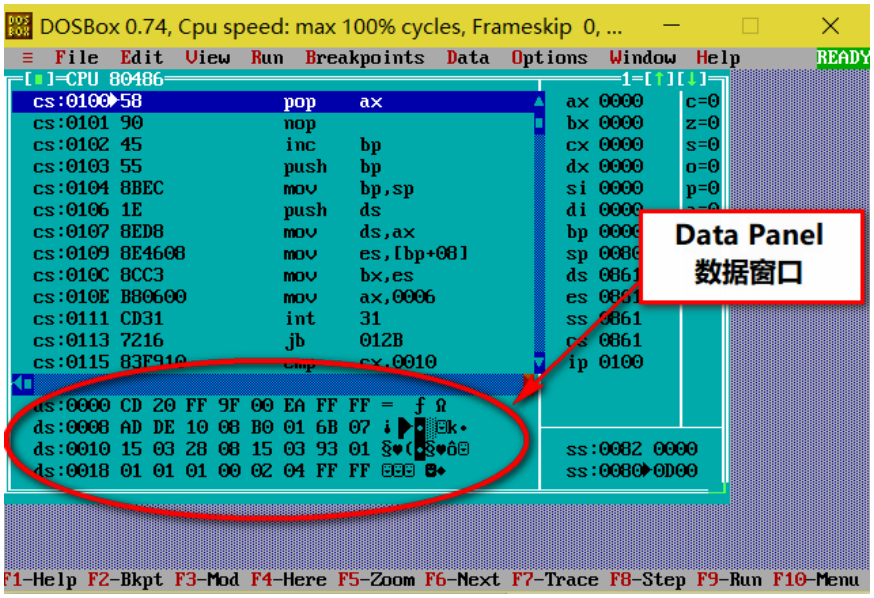


图 3.2 2　数据窗口

这里默认给出的是数据段 DS 的内容，但在单步执行的过程中，这个部分显示的内容会自动地变成 ES。这时，就需要我们手动设置了。

鼠标点击数据窗口，将焦点切换入其中，然后右键点击，弹出菜单：

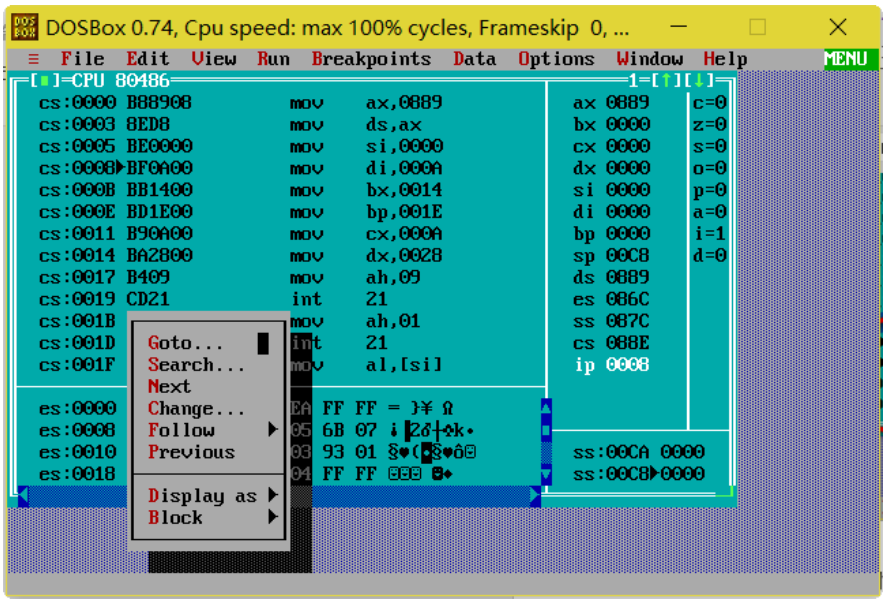


图 3.2 3　打开数据窗口的右键菜单

在菜单中选择“Goto…”，输入 DS 段的起始地址“DS:0000”。回车后，数据窗口中出现的数据就是 DS 段的数据了。

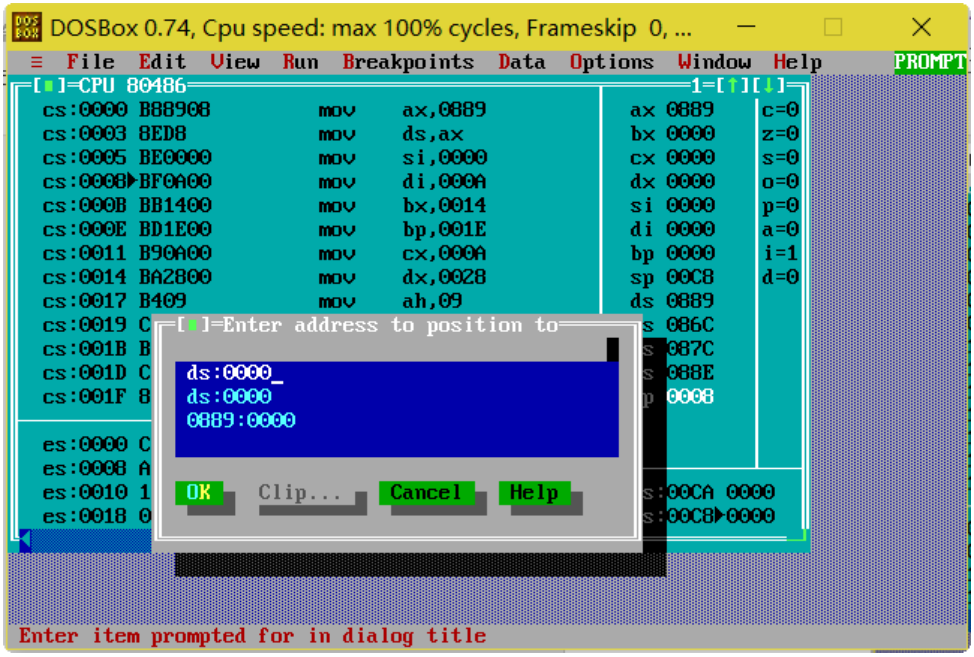


图 3.2 4　设置数据窗口显示的起始地址

4. 重新打开 TASK2.ASM，修改原有代码，以实现实验题目（3）的预期设计。

首先是提示信息的显示。在 DATA SEGMENT 段加入以下变量声明，存储提示信息的内容：

```
INFO1    DB  'Press any key to begin!', 0DH, 0AH, '$'
INFO2    DB  'Finished!', 0DH, 0AH, '$'
```

然后在“MOV CX, 10”和“LOPA:”之间加入一段代码，以打印输出第一条提示信息：

```
;Show info message
LEA DX, INFO1
MOV AH, 9
INT 21H
```

同理，在“JNZ LOPA”和“MOV AH, 4CH”之间加入一段代码，以打印输出第二条提示信息：

```
; Show a finish informer
LEA DX, INFO2
MOV AH, 9
INT 21H
```

其次是实现按键之后才执行“LOPA:”标记后续的程序。直接在“LOPA:”标记前加入下面一段程序，即可实现在显示完提示信息“Press any key to begin!”后，等待用户按键后继续程序执行：

```
; Wait for key input
MOV AH, 1
INT 21H
```

5. 再用 MASM 和 LINK 重新编译链接 TASK2.ASM，随后直接运行 TASK2.EXE，对程序效果进行检

验。

### 3.2.2 实验记录与分析

1. 将课本源码录入计算机后编译链接，用 TD 打开以进行单步调试。结果发现，自己的代码混杂在 TD 的众多代码当中，很难辨别出来，以至于在单步调试的时候，找不准程序的结束点，当然也就很难找出正确的执行结果了。

据观察，在用 TD 打开一个可执行文件后，其代码可分为两类：可执行文件中的汇编代码，以及 TD 自带的与被调试程序无关的代码。前者才是我们应该关注的，而后者的作用尚未明确，可能是示例代码（Demo），或者是对操作系统功能的展开演示。二者之间若没有明确的分隔，就很容易导致混同。

要想将二者分隔开来，最佳的方法莫过于养成一个好习惯——在程序结束处写上退出指令：

```
MOV AH, 4CH
INT 21H
```

这样就可以了，如下图所示：


图 3.2 5 加入退出指令之后的代码。退出指令正好为程序的结束标志

2. 给原有的 TASK2.ASM 加入退出指令后，重新编译并打开 TD 调试。原计划使用单步调试，但是观察源代码发现，源码规模相对较大，如果使用单步调试，则往往要按相当多次 F7 键，才能到达指定行，若粗心大意则很容易错过。因此决定改用设置断点的方式。

分别把光标定位到“MOV CX, 10”和“INT 21H”这两行，然后按下 F2 键，设置断点。**注意，这里的“INT 21H”指的是课本上程序结束处，也即本实验第（3）步对原程序改造前的“INT 21H”。**

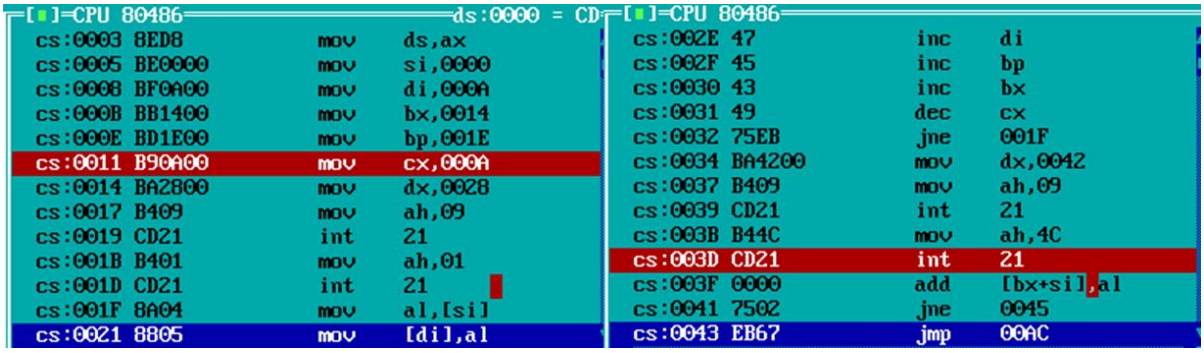


图 3.2 6　我设置的断点

3. 直接按快捷键 F9，运行程序。程序运行到断点处，会自动中止。在第一个断点“MOV CX, 10”处，可以看到执行到该语句之前的寄存器值，如下所示：

| | | | |
|---|---|---|---|
| BX=014H | BP=01EH | SI=0 | DI=0AH |

同理，第二个断点“INT 21H”处对应寄存器值如下：

| | | | |
|---|---|---|---|
| BX=01EH | BP=028H | SI=0AH | DI=014H |

4. 将断点取消，重新打开 TD，开始记录程序执行到退出前数据段 DS 的内容。先按 F7 单步执行，执行完“MOV DS, AX”给数据段寄存器赋起始偏移地址后，数据窗口中的内容由 DS 变为 ES。然后按照实验步骤部分的方法，使其显示 DS 的内容，此时数据窗口即显示程序开始执行时的 DS 内容。

接着，一路按 F7，执行到程序退出前，方可得到程序退出前的 DS 内容。值得注意的是，在程序执行过程中，数据窗口的内容一直在实时变化，在本程序中表现为内容的增加。

下图即为实验所要求的数据段内容，其中左为程序执行起始时，右为程序结束时：

表格 3.2 1　数据段内容，左为程序起始时，右为结束时


需要注意的是，TD 这个程序的设计，采用了当今 Windows 环境下流行的“灵活窗体布局”，即允许用户自由调整窗体内各个元素的位置、大小等。在 TD 的界面中，主操作区的窗口也可以被自由调整大小。要想如上图那样，一次性能把 8×5=40 个字节显示出来，不妨单击窗口右上角的上箭头，将之最大化，如图所示：

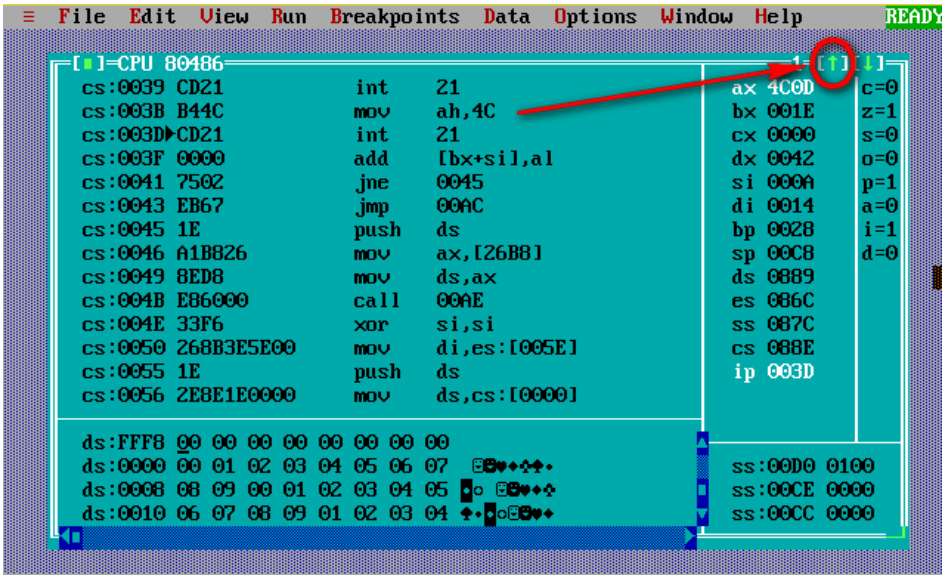


图 3.2 7　最大化主操作区窗口

从这一细节上足以见识到开发者的匠心独运。

5. 最后根据步骤（3）要求，进一步完善现有代码，再度重新编译链接。编译通过，运行一切正常，效果如下：

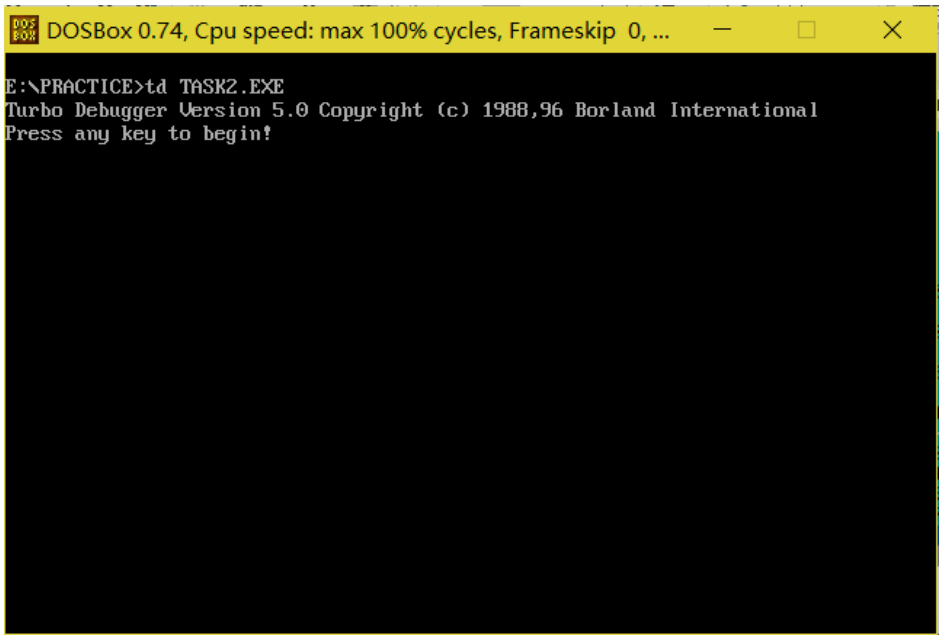


图 3.2 8　效果图一：在 TD 中运行

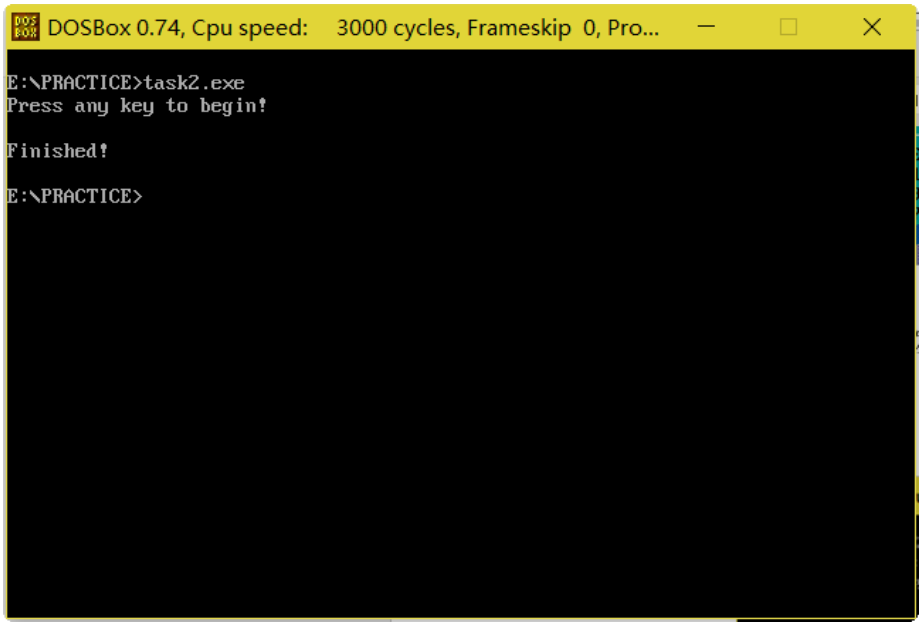


图 3.2 9　效果图二：独立运行

### 3.2.3 源程序

```
; 以下是根据要求（3）而进行调整的代码，可以实现在标号 LOPA 前等待用户输入
.386
STACK SEGMENT STACK USE16
 DB 200 DUP(0)
STACK ENDS

DATA SEGMENT USE16
 BUF1 DB   0,1,2,3,4,5,6,7,8,9
 BUF2 DB   10   DUP(0)
 BUF3 DB   10   DUP(0)
 BUF4 DB   10   DUP(0)
 INFO1     DB   'Press any key to begin!', 0DH, 0AH, '$'
 INFO2     DB   'Finished!', 0DH, 0AH, '$'
DATA ENDS


CODE SEGMENT USE16
 ASSUME    CS:CODE, DS: DATA, SS: STACK
START:
 MOV AX, DATA
 MOV DS, AX
 MOV SI, OFFSET BUF1
 MOV DI, OFFSET BUF2
 MOV BX, OFFSET BUF3
 MOV BP, OFFSET BUF4
 MOV CX, 10
```

```
 ;Show info message
 LEA DX, INFO1
 MOV AH, 9
 INT 21H

 ; Wait for key input
 MOV AH, 1
 INT 21H

LOPA:
 MOV AL, [SI]
 MOV [DI], AL
 INC AL
 MOV [BX], AL
 ADD AL, 3
 MOV DS:[BP], AL
 INC SI
 INC DI
 INC BP
 INC BX
 DEC CX
 JNZ LOPA

 ; Show a finish inform message
 LEA DX, INFO2
 MOV AH, 9
 INT 21H

 MOV AH, 4CH
 INT 21H

CODE ENDS
 END START
```

# 4 总结与体会

## 4.1 对实验所涉及知识点的主要认识与收获

### 4.1.1 任务一

通过任务一，我掌握了 TD 工具的基本使用，以及用 TD 来调试程序代码的能力。

TD 的诞生，面向的就是专业的开发者，所以它的功能尤为强悍，除了支持调试编译好的可执

行程序外，还支持即时运行汇编代码，实时查看寄存器、指示器的值。

其中最有代表性的特性，即为支持即时运行汇编代码：用户只需在程序中键入代码，即可立即运行，而无须专门去写 ASM 格式的源代码文件。在这样的环境下，我们随时可以把新学的汇编语句输入其中，尝试执行并查看效果，如此实践，完全可以加深对汇编指令的印象。所以说，要学习汇编语言，TD 是必不可少的良师益友。

其次，寄存器、指示器值的实时显示，也是相当惊艳的功能。各种常用寄存器与指示器的值整齐罗列在界面的右端，运行时只要哪个值发生了变化，TD 就会立即将其加亮显示，直观明了。同时，为了更好地支持调试，这些值都是可以修改的。

当然，TD 是一款强大的工具，我们目前使用的功能仅仅是再基础不过的那么一小部分。更多的用途，以及潜在的惊艳，有待在日后的学习过程中慢慢发掘。

### 4.1.2 任务二

通过任务二，我掌握了 TD 工具的进阶使用——调试功能，以及初步编写汇编程序的能力。

当代 IDE 所具有的调试功能，TD 当然也有，最常用的莫过于单步调试和断点调试，二者各有千秋。其中单步调试适用于代码量少的汇编程序，以及任务一中的即时运行代码；断点调试适用于代码量大的汇编程序。当然，如果要观察程序运行的每一步中寄存器、指示器值的变化，则应选择单步调试。

对于正在学习汇编语言的我们来说，调试功能是学习利器，因为调试过程中，程序的走向、存储器值的变化均一览无余，跟着 TD 中的光标，就足以让我们明白一个程序是如何运行的。课本上的那段示例代码，采用了循环结构，是再好不过的学习素材。

接下来就是更上一层楼——完善程序。完善程序，首先就是要审题，明白出题人的要求，接着根据要求，确定如何完善。要求完成的内容很简单，就是在课本程序的循环体前打出一段提示文字，同时要求按键后才能继续执行。分析题目可知，这就是要新学习的系统功能调用运用到程序当中——等待用户输入（1 号系统功能）与打印输出字符串（9 号系统功能）。

由此分析后，接下来的工作，就是编写程序了。编写程序的过程，正好是对前段时间学习的一个大总结：真正将所学指令用于真正的程序当中，按照汇编语言的规范，与操作系统的约定，有序组织之。这意义尤为重大，尽管我们真正要写的代码不过寥寥数行。

其实，寥寥数行代码，照样可以写出成就感，毕竟这是我学习汇编以来，正常实现的“第一个程序”。

## 4.2 经验教训与发现的问题

### 4.2.1 汇编语言之初体验：习惯的转变

没有 100%顺利一遍即通过的实验，发现问题在所难免，尤其是汇编这样非常低层的编程语言。

汇编的底层性，决定了它不可能像高级语言那样给我们建构好完备的逻辑框架，并提供完备的逻辑查错机制；相反，连最基本的程序结构，尤其是条件结构和循环结构，都得我们纯手工地调用汇编指令来实现。

以一个简单的循环结构程序为例。在高级语言中，我们可以把循环体装到代码块当中，然后写上 for、while 这样的循环关键字，跟上由直观的表达式构成的循环条件，这样基本能保证我们的循环可以跑得起来。

但在汇编上，可不是如此了。我们得用再原始不过的跳转语句，配上标号，来实现循环体的重复执行；而条件判断，也得用 CMP 这样原始色彩浓厚的语句——连比较运算符都不支持。然而，这还不够，为了适应各种原始的运算，跳转语句俨然构成了一个庞大的家庭，兄弟姐妹多得不得了。想要正确地选用跳转语句，更得盯着标志位，明白什么样的标志位使哪个跳转语句起作用……

以上种种体会，对于早已习惯 C、Python、JavaScript 这样之高级语言的我来说，印象尤其深刻，这毕竟是两种截然不同的编程习惯在对比。但是，万事开头难，适应总需要那么一段时间，难捱的适应期过了，就是清爽自在的艳阳天了。

汇编语言学习，还有另外一大难点——数量庞大、含义各异的指令集。

关于指令含义的理解，其实是不难的——汇编语言的指令，绝大部分都是该指令英文描述之缩写（多为机器指令），辅以 OFFSET、ASSUME、SEGMENT 这样的词汇（多为伪指令）。就拿前面提到的跳转语句那一大家子为例子：

表格 4.2 1　跳转指令和英文全称对应关系举例

| JZ/JNZ | JG | JLE | JP | JO | JMP |
|---|---|---|---|---|---|
| Jump when (not) zero | Jump when greater than | Jump when larger than | Jump when pairs | Jump when overflow | Jump |

英文全称摆在这里，很好理解吧。

那么，指令执行之后会产生什么效果？执行结果会送往哪些寄存器？标志位会如何变化？每条指令都会给这三个问题不同的回答，有的回答甚至会颠覆我们的想象。这时，课本的作用就可以100%发挥了，几乎所有的常用指令在里面都有详尽的讲解，用到时及时查查。

随着学习过程的推进，我发现，汇编语言并没有我想象中的那么难。总结一下，学习过程中真正的难点，一是对这一全新编程习惯的接受、运用与掌握；二就是知道不同的指令、语句有什么样的含义。而它们，其实都能很容易被征服。

### 4.2.2　论退出指令 MOV AH, 4CH 的重要性

退出指令“MOV AH, 4CH; INT 21H”是我们必须掌握的汇编指令，主要用在汇编程序的结束处，用来结束程序，可看作是程序结束的标志。时刻不忘写该语句，是良好的编程习惯，有利于提高代码的可读性。

当然，实际编程当中，如果不写这两个语句，同样是可以通过 MASM 汇编器的编译的，而且编译链接出的可执行文件也是可以正常运行的。我猜想，操作系统在程序代码执行结束时，会隐含地

执行退出指令。所以在实践当中，退出指令也不是不可以省略。

但是在用 TD 调试程序的时候，问题可就没这么简单了。用 TD 打开一个未写退出指令的可执行文件，然后阅读窗口中的汇编代码，找出属于这个程序的代码，接着单步调试、断点调试……很快，你就会发现：找不着北了——要么在单步调试的过程中一不留神错过了关键的地方，要么眼睁睁看着程序神不知鬼不觉地运行结束。到底哪里才是真正的程序代码？

为什么会出现如此尴尬的情况？这源于 TD 本身的设计——包含预置代码。不带任何参数打开 TD，首先映入眼帘的一定是它代码窗口中的预置代码，而不是清清白白一个“Untitled Document（未命名文档）”。这些代码的作用尚不明确，很有可能是示例代码。然而，当打开一个程序后，窗口中的代码构成明显变得更加复杂了，除了被调试程序本身的代码，还包括 TD 预置的另一批代码（可能是对操作系统功能的演示），二者非常容易混淆。

要想将被调试程序的代码与 TD 的预置代码区分开来，最简单直接的方法，就是把退出指令写上。在 TD 中，这两行指令会被原原本本显示出来，看到这两行指令，方可知道程序的结束点就在那里，不必再费尽心思对照着 ASM 源码慢慢去找。

因此，为了更好地调试程序，更为了良好的编程习惯，退出指令还是记得写比较好。

# 参考文献

[1] 王元珍,曹忠升,韩宗芬.《80X86 汇编语言程序设计》.版本(2005 版).武汉:华中科技大学出版社,2005.