王 爽 著

Assembly Language

汇编语言（第4版）

为方便教学，给教师用户提供以下教学资源：

- 电子课件
- 多套试卷
- 自动评分考试系统

清华大学出版社

汇 编 语 言

(第 4 版)

王 爽 著

清华大学出版社
北 京

内 容 简 介

汇编语言是各种 CPU 提供的机器指令的助记符的集合，人们可以用汇编语言直接控制硬件系统进行工作。汇编语言是很多相关课程(如数据结构、操作系统、微机原理等)的重要基础。为了更好地引导、帮助读者学习汇编语言，作者以循序渐进的思想精心创作了这本书。本书具有如下特点：采用了全新的结构对课程的内容进行组织，对知识进行最小化分割，为读者构造了循序渐进的学习线索；在深入本质的层面上对汇编语言进行讲解；对关键环节进行深入的剖析。

本书可用作大学计算机专业本科生的汇编语言教材及希望深入学习计算机科学的读者的自学教材。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

汇编语言/王爽著. —4 版. —北京：清华大学出版社，2019.11
ISBN 978-7-302-53941-4

Ⅰ. ①汇…　Ⅱ. ①王…　Ⅲ. ①汇编语言—程序设计　Ⅳ. ①TP313

中国版本图书馆 CIP 数据核字(2019)第 224400 号

责任编辑： 章忆文　桑任松
装帧设计： 李　坤
责任校对： 王明明
责任印制： 丛怀宇
出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn, http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈： 010-62772015, zhiliang@tup.tsinghua.edu.cn
课件下载： http://www.tup.com.cn, 010-62791865
印 装 者： 北京密云胶印厂
经　　销： 全国新华书店
开　　本： 185mm×260mm　　**印　张：** 21.75　　**字　数：** 530 千字
版　　次： 2003 年 9 月第 1 版　2019 年 12 月第 4 版　　**印　次：** 2019 年 12 月第 1 次印刷
印　　数： 1～3000
定　　价： 49.00 元

产品编号：079708-01

前　　言

汇编语言是很多相关课程(如数据结构、操作系统、微机原理等)的重要基础。其实仅从课程关系的角度讨论汇编语言的重要性未免片面，概括地说，如果你想从事计算机科学方面的工作的话，汇编语言的基础是必不可缺的。原因很简单，我们的工作平台、研究对象都是机器，汇编语言是人和计算机沟通的最直接的方式，它描述了机器最终所要执行的指令序列。想深入研究英国文化，不会英语行吗？汇编语言是和具体的微处理器相联系的，每一种微处理器的汇编语言都不一样，只能通过一种常用的、结构简洁的微处理器的汇编语言来进行学习，从而达到学习汇编的两个最根本的目的：充分获得底层编程的体验，深刻理解机器运行程序的机理。这两个目的达到了，其他目的也就自然而然地达到了。举例来说，你在学习操作系统等课程时，对许多问题就会有很通透的理解。

学习不能在一台抽象的计算机上来进行，必须针对一台具体的计算机来完成学习过程。为了使学习的过程容易展开，我们采用以 8086CPU 为中央处理器的 PC 机来进行学习。8086CPU 满足的条件：常用而结构简洁，常用保证了可以方便地进行实践，结构简洁则便于进行教学。纯粹的 8086PC 机已经不存在了，对于现今的机器来讲，它已经属于古玩。但是，现在的任何一台 PC 机中的微处理器，只要是和 Intel 兼容的系列，都可以 8086 的方式进行工作。可以将一个奔腾系列的微处理器当作一个快速的 8086 微处理器来用。整个奔腾 PC 的工作情况也是如此，可以当作一台高速的 8086PC 来用。关于微处理器及相关的一些问题请参看附注 1。

为了更好地引导、帮助学习者学习汇编语言，作者精心创作了这本书。下面对教学思想和教学内容的问题进行一些探讨，希望在一些重要的问题上和读者达到共识。

1. 教学思想

一门课程是由相互关联的知识构成的，这些知识在一本书中如何组织则是一种信息组织和加工的艺术。学习是一个循序渐进的过程，但并不是所有的教学都是以这种方式完成的，这并不是我们所希望看到的事情，因为任何不以循序渐进的方式进行的学习，都将出现盲目探索和不成系统的情况，最终学习到的也大都是相对零散的知识，并不能建立起一个系统的知识结构。非循序渐进的学习，也达不到循序渐进学习所能达到的深度，因为后者是步步深入的，每一步都以前一步为基础。

你也许会问：“我们不是一直以循序渐进的方式学习吗？有哪本书不是从第一章到最后一章，又有哪门课不是从头讲到尾的呢？”

一本书从第一章到最后一章，一门课从头到尾，这是一个时间先后的问题，这并不等于就是以循序渐进的方式在学习。我们是否常有这样的感受？想认真地学习一门较难的课程，可是却经常看不懂书上的内容；有时觉得懂了，可又总有一种不能通透的感觉，觉得书上的内容再反复看，也不能深入下去了。这些情况都说明，我们并未真正以循序渐进的方式学习。

不能循序渐进地学习的根本原因在于：学习者所用的教材并未真正地按循序渐进的原则来构造。这不是一个简单的问题，不是按传统的方法划分一下章节就可以解决的。举例来说，在传统的汇编教材中，一般都在开始的章节中集中讲 CPU 的编程结构，这一章往往成为大多数初学者的障碍。这章所讲的内容有的需要了解其他的知识才能深入理解，可是这些知识都被忽略了；有的需要有编程经验才能深入理解，或不进行具体编程就根本无法理解，可编程要在后面的章节里进行……

为学习者构造合理的学习线索，这个学习线索应真正地遵循循序渐进的原则。我们需要打破传统的章节划分，以一种新的艺术来对课程的内容进行补充、分割、重组，使其成为一个个串联在学习线索上的完成特定教学功能的教学节点。本书以此作为创作的核心理念，打破了传统的章节划分，构造了合理的学习线索，将课程的内容拆解到学习线索中的各个教学节点中去。学习主线索上的教学节点有 4 类：①知识点(即各小节内容)；②检测点；③问题和分析；④实验。还有一种被称为附注的教学节点不在学习主线索之中，是由知识点引出的节点，属于选看内容。

应用这本书，读者将沿着学习线索来学习一个个知识点，通过一个个检测点，被线索引入到一个个问题分析之中，并完成一个个实验，线索上的每一个教学节点都是后续内容的基础。每一个节点的信息量或难度，又只比前面的多一点，读者在每一步的学习中都会有一种有的放矢的感觉。大的困难被分割，读者在学习的过程中可逐步克服。

这好似航行，我们为学习者设计一条航线，航线上分布着港口，每一个港口都是下一个港口的起点。漫长的旅途被一个个港口分割，我们通过到达每个港口来完成整个航行。

为了按循序渐进的原则构造学习线索，本书采用了一种全新的信息组织和加工艺术，我们称其为知识屏蔽。有的教材只注重知识的授予，并不注重知识的屏蔽。在教学中知识的屏蔽十分重要，这是一个重点突出的问题。计算机是一门交叉学科，一部分知识往往还连带着其他的相关内容，这些连带的相关内容如果处理不好，将影响学习者对目前要掌握的知识的理解。本书采用了知识屏蔽的方法，对教学内容进行了最小化分割，力求使我们在学习过程中所接触到的每一个知识点都是当前唯一要去理解的东西。我们在看到这个知识点之前，已理解了以前所有的内容；在学习这个知识点的过程中，以后的知识也不会对

我们造成干扰。我们在整个学习过程中，每一步都走得清楚而扎实，不知不觉中，由当初的一个简单的问题开始，在经历了一个每一步都相对简单的过程之后，被带入了一个深的层次。这同沿着楼梯上高楼一样，迈出的每一步都不高，结果却上了楼顶。

2. 本书的结构

本书由若干章构成，一章包含若干知识点，根据具体内容，还可能包含检测点、问题和分析、实验、附注等教学节点。书中的所有教学节点，除附注之外，都在一个全程的主线索之中。

由于本书具有很强的线索性，学习一定要按照教学的线索进行，有两点是必须要遵守的原则：①没有通过检测点不要向下学习；②没有完成当前的实验不要向下学习。

下面的表格详细说明了书中的各种教学节点和它们的组织情况。

教学节点详表

教学节点	说　明
知识点	学习者的主要知识来源。知识点以小节的形式出现，一个知识点为一个小节。每一个知识点都有一个相对独立的小主题。
附注	有些内容是对主要内容的拓展、加深和补充。这些内容如果放入正文中，会分散学习者对主体内容的注意力，同时也破坏了主体内容的系统性。我们把这些内容在附注中给出，供学习者选看。附注不在主线索之中，是主线索的引出内容。
检测点	检测点用来取得学习情况的反馈。只要通过了检测点，我们就得到了一个保证：已掌握了前面的内容。这是对学习成果的阶段性的肯定，有了这个肯定，可以信心十足地继续学习。如果没有通过检测点，需要回头再进行复习。有的检测点中也包含了一些具有教学功能的内容。
问题分析	引导学习者对知识进行深入的理解和灵活的应用。
实验	在本书中，实验也是在学习线索中的。有的教学内容就包含在编程的依据材料中。每一个实验都是后续内容的基础，实验的任务必须独立完成。我们可以这样看待实验的重要性，如果你没有完成当前的实验，就应停止继续学习，直到你独立完成实验。

3. 教学重心和内容特点

本书的教学重心是：通过学习关键指令来深入理解机器工作的基本原理，培养底层编程意识和思想。本着这个原则，本书的内容将和传统的教材有着很大的不同。

(1) 不讲解每一条指令的功能

指令仅仅是学习机器基本原理和设计思想的一种实例。而逐条地讲解每一条指令的功能，不是本书的职责所在，它应该是一本指令手册的核心内容。这就好像文学作品和字典

的区别，前者的重心在于用文字表达思想，后者讲解每个字的用法。

(2) 编程的平台是硬件而不是操作系统

这一点尤为重要，直接影响以后的操作系统的教学。我们必须通过一定的编程实践，体验一个裸机的环境，在一个没有操作系统的环境中直接对硬件编程。这样的体会和经验非常重要，这样我们才能真正体会到汇编语言的作用，并且看到没有操作系统的计算机系统是怎样的。这为以后的操作系统的学习打下了一个重要的基础。

(3) 着重讲解重要指令和关键概念

本书的所有内容都是围绕着“深入理解机器工作的基本原理”和“培养底层编程意识和思想”这两个核心目标来进行的。对所有和这两个目标关系并不密切的内容，都进行了舍弃。使学习者可以集中注意力真正理解和掌握那些具有普遍意义的指令和关键概念。

本书在深入到本质的层面上对重要指令和关键概念进行了讲解和讨论。这些指令和概念有：jmp、条件转移指令、call、ret、栈指令、int、iret、cmp、loop、分段、寻址方式等。

4. 读者定位

本书可用作大学计算机专业本科的汇编教材，和希望深入学习计算机科学的学习者的自学教材。本书的读者应具备以下基础：

(1) 具有计算机的使用经验；

(2) 具有二进制、十六进制等基础知识；

(3) 具有一门高级语言(BASIC、Pascal、C…)的基本编程基础。

5. 联系方法

作者的 E-mail 地址为：fewstu@163.com。

目　　录

第 1 章　基础知识 1

1.1　机器语言 1
1.2　汇编语言的产生 3
1.3　汇编语言的组成 3
1.4　存储器 4
1.5　指令和数据 4
1.6　存储单元 4
1.7　CPU 对存储器的读写 5
1.8　地址总线 6
1.9　数据总线 7
1.10　控制总线 8
1.11　内存地址空间(概述) 9
1.12　主板 9
1.13　接口卡 9
1.14　各类存储器芯片 10
1.15　内存地址空间 11

第 2 章　寄存器 14

2.1　通用寄存器 14
2.2　字在寄存器中的存储 16
2.3　几条汇编指令 17
2.4　物理地址 20
2.5　16 位结构的 CPU 20
2.6　8086CPU 给出物理地址的方法 20
2.7　“段地址×16+偏移地址=物理地址”的本质含义 22
2.8　段的概念 23
2.9　段寄存器 25
2.10　CS 和 IP 25
2.11　修改 CS、IP 的指令 32
2.12　代码段 34
实验 1　查看 CPU 和内存，用机器指令和汇编指令编程 35

第 3 章　寄存器(内存访问) 47

3.1　内存中字的存储 47
3.2　DS 和[address] 48
3.3　字的传送 49
3.4　mov、add、sub 指令 51
3.5　数据段 53
3.6　栈 56
3.7　CPU 提供的栈机制 57
3.8　栈顶超界的问题 61
3.9　push、pop 指令 63
3.10　栈段 68
实验 2　用机器指令和汇编指令编程 71

第 4 章　第一个程序 76

4.1　一个源程序从写出到执行的过程 76
4.2　源程序 77
4.3　编辑源程序 82
4.4　编译 83
4.5　连接 85
4.6　以简化的方式进行编译和连接 88
4.7　1.exe 的执行 89
4.8　谁将可执行文件中的程序装载进入内存并使它运行？ 89
4.9　程序执行过程的跟踪 91
实验 3　编程、编译、连接、跟踪 94

第 5 章　[BX]和 loop 指令 95

5.1　[BX] 97
5.2　Loop 指令 99
5.3　在 Debug 中跟踪用 loop 指令实现的循环程序 103
5.4　Debug 和汇编编译器 masm 对指令的不同处理 110

5.5 loop 和[bx]的联合应用.....112
5.6 段前缀.....116
5.7 一段安全的空间.....117
5.8 段前缀的使用.....120
实验 4 [bx]和 loop 的使用.....121

第 6 章 包含多个段的程序.....123

6.1 在代码段中使用数据.....123
6.2 在代码段中使用栈.....127
6.3 将数据、代码、栈放入不同的段.....130
实验 5 编写、调试具有多个段的程序.....133

第 7 章 更灵活的定位内存地址的方法.....138

7.1 and 和 or 指令.....138
7.2 关于 ASCII 码.....139
7.3 以字符形式给出的数据.....139
7.4 大小写转换的问题.....140
7.5 [bx+idata].....143
7.6 用[bx+idata]的方式进行数组的处理.....144
7.7 SI 和 DI.....147
7.8 [bx+si]和[bx+di].....149
7.9 [bx+si+idata]和[bx+di+idata].....150
7.10 不同的寻址方式的灵活应用.....152
实验 6 实践课程中的程序.....160

第 8 章 数据处理的两个基本问题.....161

8.1 bx、si、di 和 bp.....161
8.2 机器指令处理的数据在什么地方.....162
8.3 汇编语言中数据位置的表达.....162
8.4 寻址方式.....164
8.5 指令要处理的数据有多长.....165
8.6 寻址方式的综合应用.....166
8.7 div 指令.....169
8.8 伪指令 dd.....170
8.9 dup.....171
实验 7 寻址方式在结构化数据访问中的应用.....172

第 9 章 转移指令的原理.....175

9.1 操作符 offset.....175
9.2 jmp 指令.....176
9.3 依据位移进行转移的 jmp 指令.....177
9.4 转移的目的地址在指令中的 jmp 指令.....180
9.5 转移地址在寄存器中的 jmp 指令.....181
9.6 转移地址在内存中的 jmp 指令.....182
9.7 jcxz 指令.....184
9.8 loop 指令.....185
9.9 根据位移进行转移的意义.....186
9.10 编译器对转移位移超界的检测.....186
实验 8 分析一个奇怪的程序.....187
实验 9 根据材料编程.....187

第 10 章 CALL 和 RET 指令.....190

10.1 ret 和 retf.....190
10.2 call 指令.....192
10.3 依据位移进行转移的 call 指令.....192
10.4 转移的目的地址在指令中的 call 指令.....193
10.5 转移地址在寄存器中的 call 指令.....194
10.6 转移地址在内存中的 call 指令.....194
10.7 call 和 ret 的配合使用.....196
10.8 mul 指令.....199
10.9 模块化程序设计.....200
10.10 参数和结果传递的问题.....200
10.11 批量数据的传递.....201
10.12 寄存器冲突的问题.....203
实验 10 编写子程序.....206
课程设计 1.....211

第 11 章 标志寄存器.....213

11.1 ZF 标志.....213
11.2 PF 标志.....214

11.3 SF 标志 215
11.4 CF 标志 216
11.5 OF 标志 217
11.6 adc 指令 219
11.7 sbb 指令 222
11.8 cmp 指令 222
11.9 检测比较结果的条件转移指令 225
11.10 DF 标志和串传送指令 230
11.11 pushf 和 popf 233
11.12 标志寄存器在 Debug 中的表示 234
实验 11 编写子程序 234

第 12 章 内中断 236

12.1 内中断的产生 236
12.2 中断处理程序 237
12.3 中断向量表 237
12.4 中断过程 238
12.5 中断处理程序和 iret 指令 239
12.6 除法错误中断的处理 240
12.7 编程处理 0 号中断 240
12.8 安装 244
12.9 do0 246
12.10 设置中断向量 249
12.11 单步中断 249
12.12 响应中断的特殊情况 250
实验 12 编写 0 号中断的处理程序 251

第 13 章 int 指令 252

13.1 int 指令 252
13.2 编写供应用程序调用的中断例程 253
13.3 对 int、iret 和栈的深入理解 256
13.4 BIOS 和 DOS 所提供的中断例程 258
13.5 BIOS 和 DOS 中断例程的安装过程 258
13.6 BIOS 中断例程应用 259
13.7 DOS 中断例程应用 261
实验 13 编写、应用中断例程 262

第 14 章 端口 265

14.1 端口的读写 265
14.2 CMOS RAM 芯片 266
14.3 shl 和 shr 指令 267
14.4 CMOS RAM 中存储的时间信息 269
实验 14 访问 CMOS RAM 271

第 15 章 外中断 272

15.1 接口芯片和端口 272
15.2 外中断信息 272
15.3 PC 机键盘的处理过程 274
15.4 编写 int 9 中断例程 276
15.5 安装新的 int 9 中断例程 282
实验 15 安装新的 int 9 中断例程 285

第 16 章 直接定址表 287

16.1 描述了单元长度的标号 287
16.2 在其他段中使用数据标号 289
16.3 直接定址表 292
16.4 程序入口地址的直接定址表 296
实验 16 编写包含多个功能子程序的中断例程 299

第 17 章 使用 BIOS 进行键盘输入和磁盘读写 300

17.1 int 9 中断例程对键盘输入的处理 300
17.2 使用 int 16h 中断例程读取键盘缓冲区 301
17.3 字符串的输入 304
17.4 应用 int 13h 中断例程对磁盘进行读写 308
实验 17 编写包含多个功能子程序的中断例程 310
课程设计 2 312

综合研究 314

研究试验 1 搭建一个精简的 C 语言开发环境 315

研究试验 2 使用寄存器.....318
研究试验 3 使用内存空间.....319
研究试验 4 不用 main 函数编程.....322
研究试验 5 函数如何接收不定数量的参数.....326

附注.....327

附注 1 Intel 系列微处理器的 3 种工作模式.....327
附注 2 补码.....329
附注 3 汇编编译器(masm.exe)对 jmp 的相关处理.....330
附注 4 用栈传递参数.....333
附注 5 公式证明.....336

第 1 章　基 础 知 识

汇编语言是直接在硬件之上工作的编程语言，我们首先要了解硬件系统的结构，才能有效地应用汇编语言对其编程。在本章中，我们对硬件系统结构的问题进行一部分的探讨，以使后续的课程可在一个好的基础上进行。当课程进行到需要补充新的基础知识(关于编程结构或其他的)的时候，再对相关的基础知识进行介绍和探讨。我们的原则是，以后用到的知识，以后再说。

在汇编课程中我们不对硬件系统进行全面和深入的研究，这不在课程的范围之内。关于 PC 机及 CPU 物理结构和编程结构的全面研究，在《微机原理与接口》中进行；对于计算机一般的结构、功能、性能的研究在一门称为《组成原理》的理论层次更高的课程中进行。汇编课程的研究重点放在如何利用硬件系统的编程结构和指令集有效灵活地控制系统进行工作。

1.1　机 器 语 言

说到汇编语言的产生，首先要讲一下机器语言。机器语言是机器指令的集合。机器指令展开来讲就是一台机器可以正确执行的命令。电子计算机的机器指令是一列二进制数字。计算机将之转变为一列高低电平，以使计算机的电子器件受到驱动，进行运算。

上面所说的计算机指的是可以执行机器指令，进行运算的机器。这是早期计算机的概念。现在，在我们常用的 PC 机中，有一个芯片来完成上面所说的计算机的功能。这个芯片就是我们常说的 CPU(Central Processing Unit，中央处理单元)，CPU 是一种微处理器。以后我们提到的计算机是指由 CPU 和其他受 CPU 直接或间接控制的芯片、器件、设备组成的计算机系统，比如我们最常见的 PC 机。

每一种微处理器，由于硬件设计和内部结构的不同，就需要用不同的电平脉冲来控制，使它工作。所以每一种微处理器都有自己的机器指令集，也就是机器语言。

早期的程序设计均使用机器语言。程序员们将用 0、1 数字编成的程序代码打在纸带或卡片上，1 打孔，0 不打孔，再将程序通过纸带机或卡片机输入计算机，进行运算。

应用 8086CPU 完成运算 s=768+12288−1280，机器码如下：

```
101110000000000000000011
000001010000000000110000
001011010000000000000101
```

假如将程序错写成以下这样，请你找出错误。

```
101100000000000000000011
000001010000000000110000
000101101000000000000101
```

书写和阅读机器码程序不是一件简单的工作，要记住所有抽象的二进制码。上面只是一个非常简单的小程序，就暴露了机器码的晦涩难懂和不易查错。写如此小的一个程序尚且如此，实际上一个有用的程序至少要有几十行机器码，那么，情况将怎么样呢？

在显示器上输出“welcome to masm”，机器码如下：

```
00011110
101110000000000000000000
01010000
101110001100011000001111
1000111011011000
1011010000000110
1011000000000000
1011011100000111
101110010000000000000000
1011011000011000
1011001001001111
1100110100010000
1011010000000010
1011011100000000
1011011000000000
1011001000000000
1100110100010000
1011010000001001
10001101000101100010101000000000
1100110100100001
1011010000001010
10001101000101100011000100000000
1100110100100001
1011010000000110
1011000000010100
1011011100011001
1011010100001011
1011000100010011
1011011000001101
1011001000111100
1100110100010000
1011010000000010
1011011100000000
1011000000001100
1011001000010100
1100110100010000
1011010000001001
10001101000101100000000000000000
1100110100100001
11001011
```

看到这样的程序，你有什么感想？如果程序里有一个“1”被误写为“0”，又如何去查找呢？

1.2　汇编语言的产生

早期的程序员们很快就发现了使用机器语言带来的麻烦，它是如此难于辨别和记忆，给整个产业的发展带来了障碍。于是汇编语言产生了。

汇编语言的主体是汇编指令。汇编指令和机器指令的差别在于指令的表示方法上。汇编指令是机器指令便于记忆的书写格式。

例如：机器指令 1000100111011000 表示把寄存器 BX 的内容送到 AX 中。汇编指令则写成 mov ax,bx。这样的写法与人类语言接近，便于阅读和记忆。

操作：寄存器 BX 的内容送到 AX 中
机器指令：1000100111011000
汇编指令：mov ax,bx

(寄存器，简单地讲是 CPU 中可以存储数据的器件，一个 CPU 中有多个寄存器。AX 是其中一个寄存器的代号，BX 是另一个寄存器的代号。更详细的内容我们在以后的课程中将会讲到。)

此后，程序员们就用汇编指令编写源程序。可是，计算机能读懂的只有机器指令，那么如何让计算机执行程序员用汇编指令编写的程序呢？这时，就需要有一个能够将汇编指令转换成机器指令的翻译程序，这样的程序我们称其为编译器。程序员用汇编语言写出源程序，再用汇编编译器将其编译为机器码，由计算机最终执行。图 1.1 描述了这个工作过程。

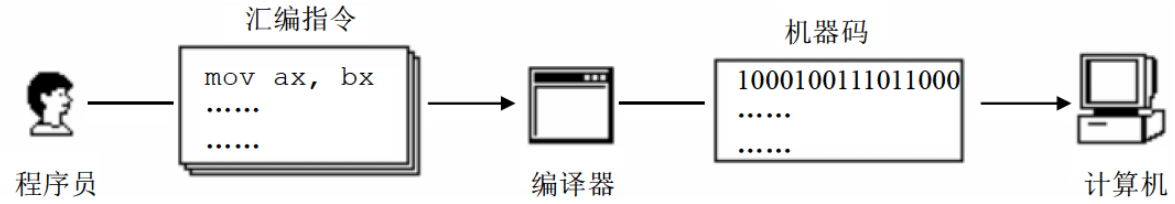


图 1.1　用汇编语言编写程序的工作过程

1.3　汇编语言的组成

汇编语言发展至今，有以下 3 类指令组成。

(1) 汇编指令：机器码的助记符，有对应的机器码。
(2) 伪指令：没有对应的机器码，由编译器执行，计算机并不执行。
(3) 其他符号：如+、-、*、/ 等，由编译器识别，没有对应的机器码。

汇编语言的核心是汇编指令，它决定了汇编语言的特性。

1.4 存 储 器

CPU 是计算机的核心部件，它控制整个计算机的运作并进行运算。要想让一个 CPU 工作，就必须向它提供指令和数据。指令和数据在存储器中存放，也就是我们平时所说的内存。在一台 PC 机中内存的作用仅次于 CPU。离开了内存，性能再好的 CPU 也无法工作。这就像再聪明的大脑，没有了记忆也无法进行思考。磁盘不同于内存，磁盘上的数据或程序如果不读到内存中，就无法被 CPU 使用。要灵活地利用汇编语言编程，我们首先要了解 CPU 是如何从内存中读取信息，以及向内存中写入信息的。

1.5 指令和数据

指令和数据是应用上的概念。在内存或磁盘上，指令和数据没有任何区别，都是二进制信息。CPU 在工作的时候把有的信息看作指令，有的信息看作数据，为同样的信息赋予了不同的意义。就像围棋的棋子，在棋盒里的时候没有任何区别，在对弈的时候就有了不同的意义。

例如，内存中的二进制信息 1000100111011000，计算机可以把它看作大小为 89D8H 的数据来处理，也可以将其看作指令 mov ax,bx 来执行。

```
1000100111011000  —> 89D8H  (数据)
1000100111011000  —> mov ax,bx  (程序)
```

1.6 存 储 单 元

存储器被划分成若干个存储单元，每个存储单元从 0 开始顺序编号，例如一个存储器有 128 个存储单元，编号从 0～127，如图 1.2 所示。

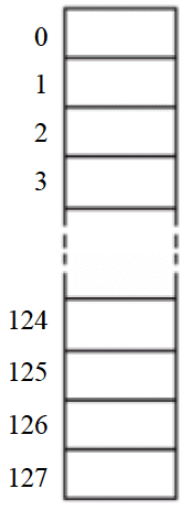


图 1.2 存储单元的编号

那么一个存储单元能存储多少信息呢？我们知道电子计算机的最小信息单位是 bit(音译为比特)，也就是一个二进制位。8 个 bit 组成一个 Byte，也就是通常讲的一个字节。微型机存储器的存储单元可以存储一个 Byte，即 8 个二进制位。一个存储器有 128 个存储单元，它可以存储 128 个 Byte。

微机存储器的容量是以字节为最小单位来计算的。对于拥有 128 个存储单元的存储器，我们可以说，它的容量是 128 个字节。

对于大容量的存储器一般还用以下单位来计量容量(以下用 B 来代表 Byte)：

1KB＝1024B　　1MB＝1024KB　　1GB＝1024MB　　1TB＝1024GB

磁盘的容量单位同内存的一样，实际上以上单位是微机中常用的计量单位。

1.7　CPU 对存储器的读写

以上讲到，存储器被划分成多个存储单元，存储单元从零开始顺序编号。这些编号可以看作存储单元在存储器中的地址。就像一条街，每个房子都有门牌号码。

CPU 要从内存中读数据，首先要指定存储单元的地址。也就是说它要先确定它要读取哪一个存储单元中的数据。就像在一条街上找人，先要确定他住在哪个房子里。

另外，在一台微机中，不只有存储器这一种器件。CPU 在读写数据时还要指明，它要对哪一个器件进行操作，进行哪种操作，是从中读出数据，还是向里面写入数据。

可见，CPU 要想进行数据的读写，必须和外部器件(标准的说法是芯片)进行下面 3 类信息的交互。

- 存储单元的地址(地址信息)；
- 器件的选择，读或写的命令(控制信息)；
- 读或写的数据(数据信息)。

那么 CPU 是通过什么将地址、数据和控制信息传到存储器芯片中的呢？电子计算机能处理、传输的信息都是电信号，电信号当然要用导线传送。在计算机中专门有连接 CPU 和其他芯片的导线，通常称为总线。总线从物理上来讲，就是一根根导线的集合。根据传送信息的不同，总线从逻辑上又分为 3 类，地址总线、控制总线和数据总线。

CPU 从 3 号单元中读取数据的过程(见图 1.3)如下。

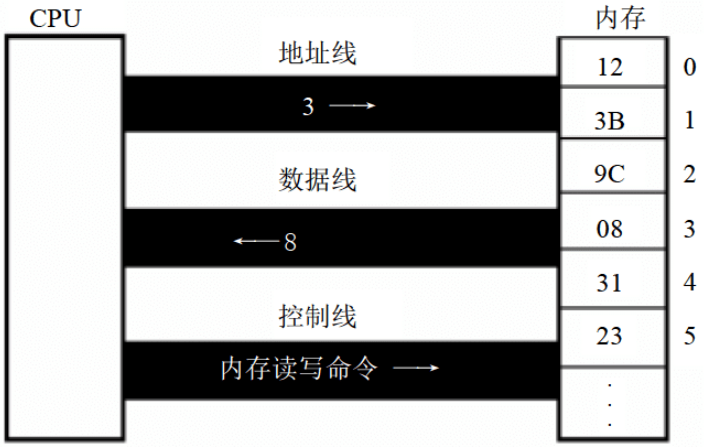


图 1.3　CPU 从内存中读取数据的过程

(1)　CPU 通过地址线将地址信息 3 发出。

(2)　CPU 通过控制线发出内存读命令，选中存储器芯片，并通知它，将要从中读取

数据。

(3) 存储器将 3 号单元中的数据 8 通过数据线送入 CPU。

写操作与读操作的步骤相似。如向 3 号单元写入数据 26。

(1) CPU 通过地址线将地址信息 3 发出。

(2) CPU 通过控制线发出内存写命令，选中存储器芯片，并通知它，要向其中写入数据。

(3) CPU 通过数据线将数据 26 送入内存的 3 号单元中。

从上面我们知道了 CPU 是如何进行数据读写的。可是，如何命令计算机进行数据的读写呢？

要让一个计算机或微处理器工作，应向它输入能够驱动它进行工作的电平信息(机器码)。

对于 8086CPU，下面的机器码，能够完成从 3 号单元读数据。

机器码： 101000010000001100000000
含义： 从 3 号单元读取数据送入寄存器 AX

CPU 接收这条机器码后将完成我们上面所述的读写工作。

机器码难于记忆，用汇编指令来表示，情况如下。

机器码： 10100001 00000011 00000000
对应的汇编指令：MOV AX,[3]
含义： 传送 3 号单元的内容入 AX

1.8 地 址 总 线

现在我们知道，CPU 是通过地址总线来指定存储器单元的。可见地址总线上能传送多少个不同的信息，CPU 就可以对多少个存储单元进行寻址。

现假设，一个 CPU 有 10 根地址总线，让我们来看一下它的寻址情况。我们知道，在电子计算机中，一根导线可以传送的稳定状态只有两种，高电平或是低电平。用二进制表示就是 1 或 0，10 根导线可以传送 10 位二进制数据。而 10 位二进制数可以表示多少个不同的数据呢？2 的 10 次方个。最小数为 0，最大数为 1023。

图 1.4 展示了一个具有 10 根地址线的 CPU 向内存发出地址信息 11 时 10 根地址线上传送的二进制信息。考虑一下，访问地址为 12、13、14 等的内存单元时，地址总线上传送的内容是什么？

一个 CPU 有 N 根地址线，则可以说这个 CPU 的地址总线的宽度为 N。这样的 CPU 最多可以寻找 2 的 N 次方个内存单元。

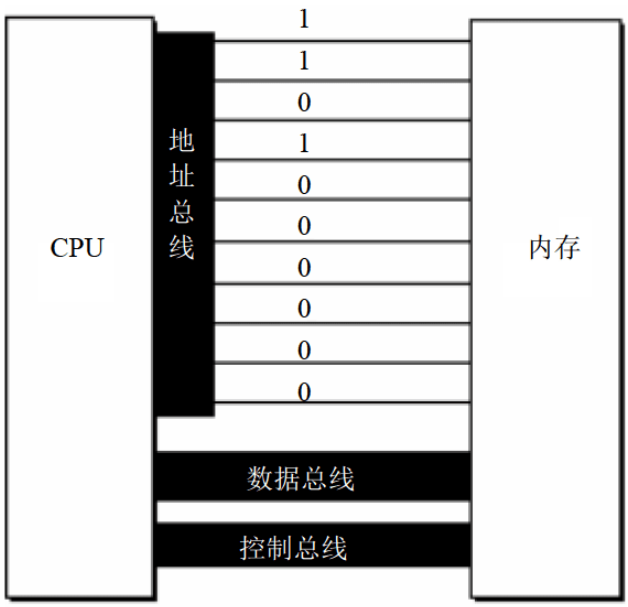


图 1.4　地址总线上发送的地址信息

1.9　数 据 总 线

CPU 与内存或其他器件之间的数据传送是通过数据总线来进行的。数据总线的宽度决定了 CPU 和外界的数据传送速度。8 根数据总线一次可传送一个 8 位二进制数据(即一个字节)。16 根数据总线一次可传送两个字节。

8088CPU 的数据总线宽度为 8，8086CPU 的数据总线宽度为 16。我们来分别看一下它们向内存中写入数据 89D8H 时，是如何通过数据总线传送数据的。图 1.5 展示了 8088CPU 数据总线上的数据传送情况；图 1.6 展示了 8086CPU 数据总线上的数据传送情况。

8088CPU 分两次传送 89D8，第一次传送 D8，第二次传送 89。

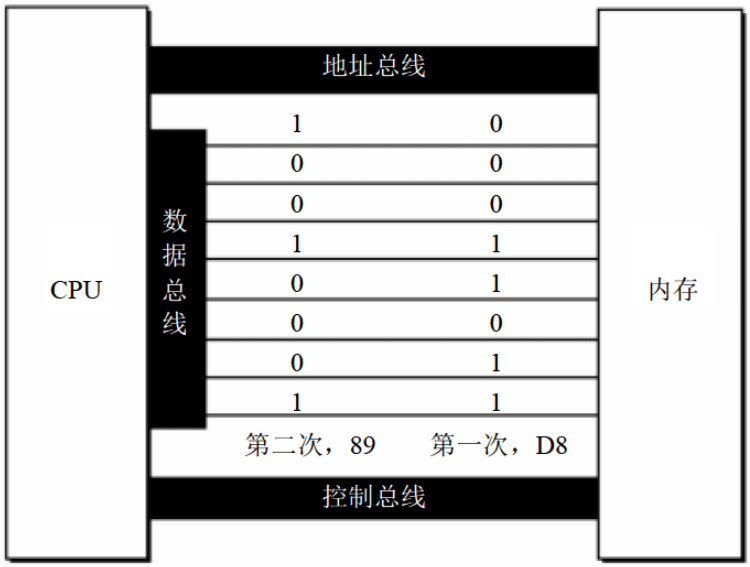


图 1.5　8 位数据总线上传送的信息

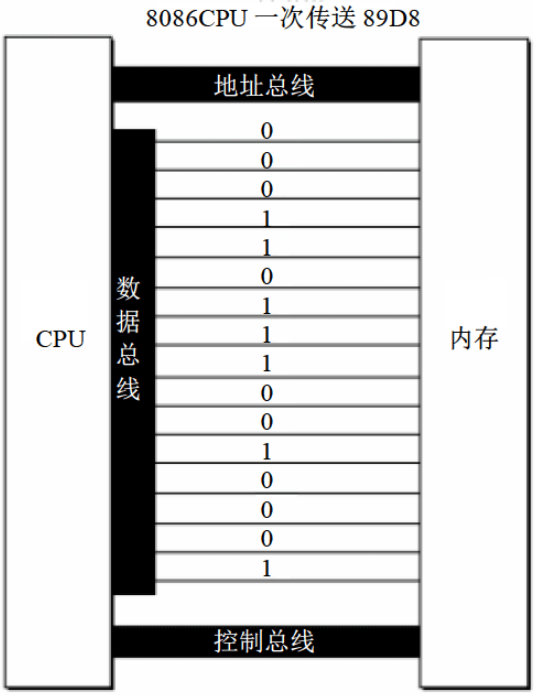


图 1.6　16 位数据总线上传送的信息

8086 有 16 根数据线，可一次传送 16 位数据，所以可一次传送数据 89D8H；而 8088 只有 8 根数据线，一次只能传 8 位数据，所以向内存写入数据 89D8H 时需要进行两次数据传送。

1.10　控 制 总 线

CPU 对外部器件的控制是通过控制总线来进行的。在这里控制总线是个总称，控制总线是一些不同控制线的集合。有多少根控制总线，就意味着 CPU 提供了对外部器件的多少种控制。所以，控制总线的宽度决定了 CPU 对外部器件的控制能力。

前面所讲的内存读或写命令是由几根控制线综合发出的，其中有一根称为“读信号输出”的控制线负责由 CPU 向外传送读信号，CPU 向该控制线上输出低电平表示将要读取数据；有一根称为“写信号输出”的控制线则负责传送写信号。

1.1~1.10　小　　结

(1) 汇编指令是机器指令的助记符，同机器指令一一对应。

(2) 每一种 CPU 都有自己的汇编指令集。

(3) CPU 可以直接使用的信息在存储器中存放。

(4) 在存储器中指令和数据没有任何区别，都是二进制信息。

(5) 存储单元从零开始顺序编号。

(6) 一个存储单元可以存储 8 个 bit，即 8 位二进制数。

(7) 1Byte＝8bit　1KB＝1024B　1MB＝1024KB　1GB＝1024MB。

(8) 每一个 CPU 芯片都有许多管脚，这些管脚和总线相连。也可以说，这些管脚引出总线。一个 CPU 可以引出 3 种总线的宽度标志了这个 CPU 的不同方面的性能：

地址总线的宽度决定了 CPU 的寻址能力；
数据总线的宽度决定了 CPU 与其他器件进行数据传送时的一次数据传送量；
控制总线的宽度决定了 CPU 对系统中其他器件的控制能力。

在汇编课程中，我们从功能的角度介绍了 3 类总线，对实际的连接情况不做讨论。

检测点 1.1

(1) 1 个 CPU 的寻址能力为 8KB，那么它的地址总线的宽度为_____。
(2) 1KB 的存储器有_____个存储单元。存储单元的编号从_____到_____。
(3) 1KB 的存储器可以存储_____个 bit，_____个 Byte。
(4) 1GB、1MB、1KB 分别是_____Byte。
(5) 8080、8088、80286、80386 的地址总线宽度分别为 16 根、20 根、24 根、32 根，则它们的寻址能力分别为：_____(KB)、_____(MB)、_____(MB)、_____(GB)。
(6) 8080、8088、8086、80286、80386 的数据总线宽度分别为 8 根、8 根、16 根、16 根、32 根。则它们一次可以传送的数据为：____(B)、____(B)、____(B)、___(B)、____(B)。
(7) 从内存中读取 1024 字节的数据，8086 至少要读____次，80386 至少要读____次。
(8) 在存储器中，数据和程序以_____形式存放。

1.11　内存地址空间(概述)

什么是内存地址空间呢？举例来讲，一个 CPU 的地址总线宽度为 10，那么可以寻址 1024 个内存单元，这 1024 个可寻到的内存单元就构成这个 CPU 的内存地址空间。下面进行深入讨论。首先需要介绍两部分基本知识，主板和接口卡。

1.12　主　　板

在每一台 PC 机中，都有一个主板，主板上有核心器件和一些主要器件，这些器件通过总线(地址总线、数据总线、控制总线)相连。这些器件有 CPU、存储器、外围芯片组、扩展插槽等。扩展插槽上一般插有 RAM 内存条和各类接口卡。

1.13　接　口　卡

计算机系统中，所有可用程序控制其工作的设备，必须受到 CPU 的控制。CPU 对外部设备都不能直接控制，如显示器、音箱、打印机等。直接控制这些设备进行工作的是插在扩展插槽上的接口卡。扩展插槽通过总线和 CPU 相连，所以接口卡也通过总线同 CPU 相连。CPU 可以直接控制这些接口卡，从而实现 CPU 对外设的间接控制。简单地讲，就是 CPU 通过总线向接口卡发送命令，接口卡根据 CPU 的命令控制外设进行工作。

1.14 各类存储器芯片

一台 PC 机中，装有多个存储器芯片，这些存储器芯片从物理连接上看是独立的、不同的器件。从读写属性上看分为两类：随机存储器(RAM)和只读存储器(ROM)。随机存储器可读可写，但必须带电存储，关机后存储的内容丢失；只读存储器只能读取不能写入，关机后其中的内容不丢失。这些存储器从功能和连接上又可分为以下几类。

- 随机存储器
  用于存放供 CPU 使用的绝大部分程序和数据，主随机存储器一般由两个位置上的 RAM 组成，装在主板上的 RAM 和插在扩展插槽上的 RAM。
- 装有 BIOS(Basic Input/Output System，基本输入/输出系统)的 ROM
  BIOS 是由主板和各类接口卡(如显卡、网卡等)厂商提供的软件系统，可以通过它利用该硬件设备进行最基本的输入输出。在主板和某些接口卡上插有存储相应 BIOS 的 ROM。例如，主板上的 ROM 中存储着主板的 BIOS(通常称为系统 BIOS)；显卡上的 ROM 中存储着显卡的 BIOS；如果网卡上装有 ROM，那其中就可以存储网卡的 BIOS。
- 接口卡上的 RAM
  某些接口卡需要对大批量输入、输出数据进行暂时存储，在其上装有 RAM。最典型的是显示卡上的 RAM，一般称为显存。显示卡随时将显存中的数据向显示器上输出。换句话说，我们将需要显示的内容写入显存，就会出现在显示器上。

图 1.7 展示了 PC 系统中各类存储器的逻辑连接情况。

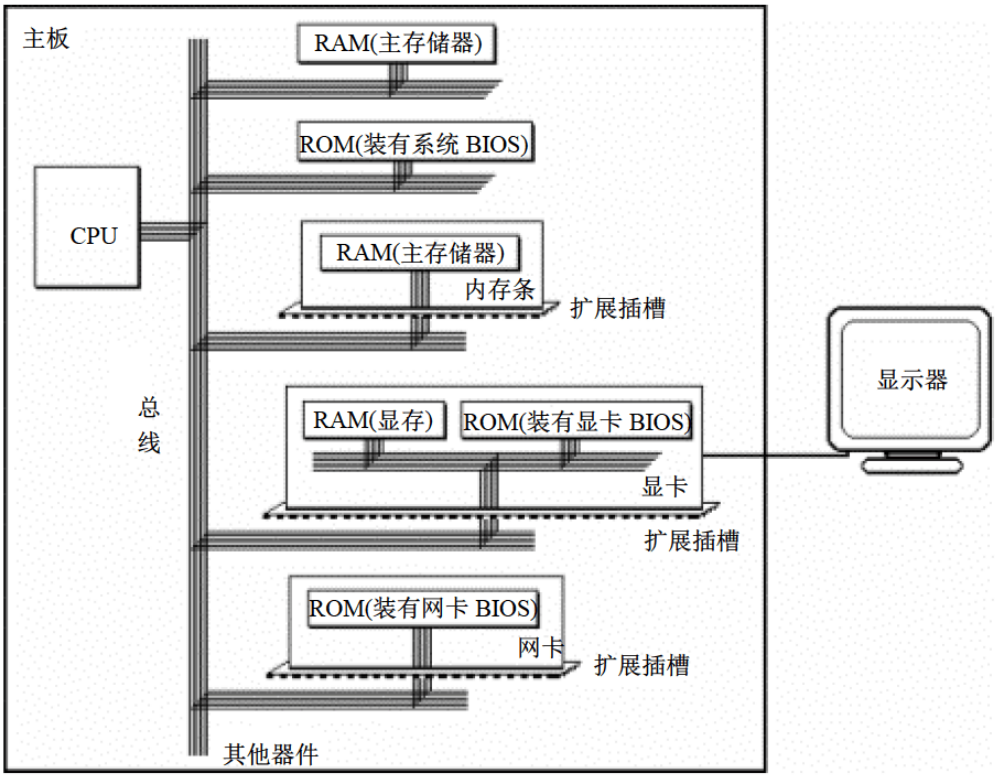


图 1.7 PC 机中各类存储器的逻辑连接

1.15　内存地址空间

上述的那些存储器，在物理上是独立的器件，但是在以下两点上相同。

- 都和 CPU 的总线相连。
- CPU 对它们进行读或写的时候都通过控制线发出内存读写命令。

这也就是说，CPU 在操控它们的时候，把它们都当作内存来对待，把它们总的看作一个由若干存储单元组成的逻辑存储器，这个逻辑存储器就是我们所说的内存地址空间。在汇编这门课中，我们所面对的是内存地址空间。

图 1.8 展示了 CPU 将系统中各类存储器看作一个逻辑存储器的情况。

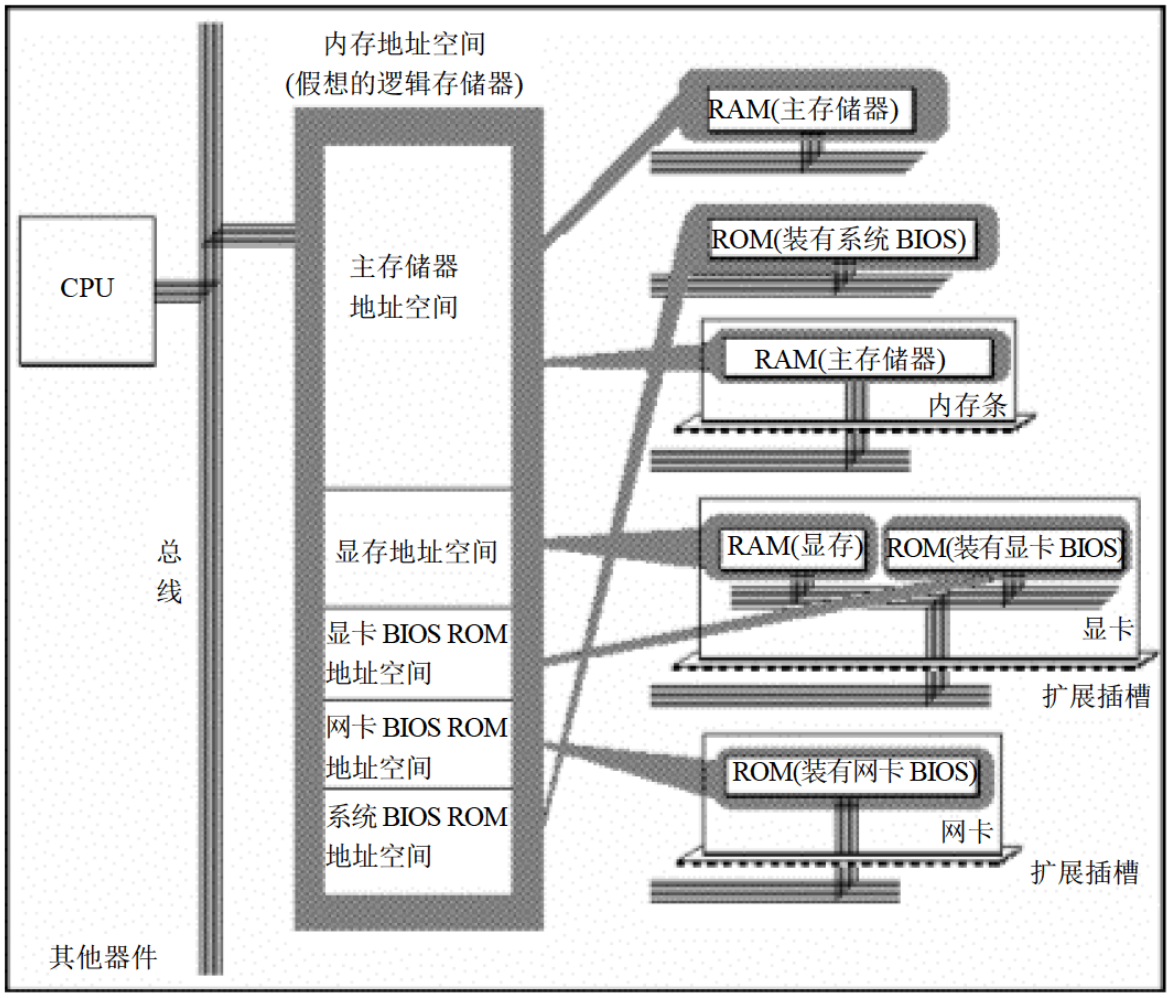


图 1.8　将各类存储器看作一个逻辑存储器

在图 1.8 中，所有的物理存储器被看作一个由若干存储单元组成的逻辑存储器，每个物理存储器在这个逻辑存储器中占有一个地址段，即一段地址空间。CPU 在这段地址空间中读写数据，实际上就是在相对应的物理存储器中读写数据。

假设，图 1.8 中的内存地址空间的地址段分配如下。

地址 0~7FFFH 的 32KB 空间为主随机存储器的地址空间；
地址 8000H~9FFFH 的 8KB 空间为显存地址空间；
地址 A000H~FFFFH 的 24KB 空间为各个 ROM 的地址空间。

这样，CPU 向内存地址为 1000H 的内存单元中写入数据，这个数据就被写入主随机存储器中；CPU 向内存地址为 8000H 的内存单元中写入数据，这个数据就被写入显存中，然后会被显卡输出到显示器上；CPU 向内存地址为 C000H 的内存单元中写入数据的操作是没有结果的，C000H 单元中的内容不会被改变，C000H 单元实际上就是 ROM 存储器中的一个单元。

内存地址空间的大小受 CPU 地址总线宽度的限制。8086CPU 的地址总线宽度为 20，可以传送 2^{20} 个不同的地址信息(大小从 0 至 $2^{20}-1$)。即可以定位 2^{20} 个内存单元，则 8086PC 的内存地址空间大小为 1MB。同理，80386CPU 的地址总线宽度为 32，则内存地址空间最大为 4GB。

我们在基于一个计算机硬件系统编程的时候，必须知道这个系统中的内存地址空间分配情况。因为当我们想在某类存储器中读写数据的时候，必须知道它的第一个单元的地址和最后一个单元的地址，才能保证读写操作是在预期的存储器中进行。比如，我们希望向显示器输出一段信息，那么必须将这段信息写到显存中，显卡才能将它输出到显示器上。要向显存中写入数据，必须知道显存在内存地址空间中的地址。

不同的计算机系统的内存地址空间的分配情况是不同的，图 1.9 展示了 8086PC 机内存地址空间分配的基本情况。

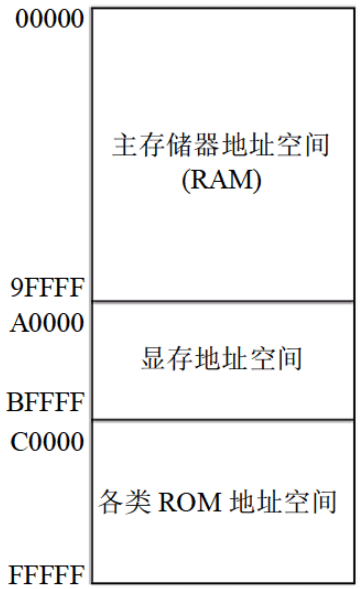


图 1.9　8086PC 机内存地址空间分配

图 1.9 告诉我们，从地址 0~9FFFF 的内存单元中读取数据，实际上就是在读取主随机

存储器中的数据；向地址 A0000~BFFFF 的内存单元中写数据，就是向显存中写入数据，这些数据会被显示卡输出到显示器上；我们向地址 C0000~FFFFF 的内存单元中写入数据的操作是无效的，因为这等于改写只读存储器中的内容。

内存地址空间

最终运行程序的是 CPU，我们用汇编语言编程的时候，必须要从 CPU 的角度考虑问题。对 CPU 来讲，系统中的所有存储器中的存储单元都处于一个统一的逻辑存储器中，它的容量受 CPU 寻址能力的限制。这个逻辑存储器即是我们所说的内存地址空间。

对于初学者，这个概念比较抽象，我们在后续的课程中将通过一些编程实践，来增加感性认识。

第 2 章　寄　存　器

一个典型的 CPU(此处讨论的不是某一具体的 CPU)由运算器、控制器、寄存器(CPU 工作原理)等器件构成，这些器件靠内部总线相连。前一章所说的总线，相对于 CPU 内部来说是外部总线。内部总线实现 CPU 内部各个器件之间的联系，外部总线实现 CPU 和主板上其他器件的联系。简单地说，在 CPU 中：

- 运算器进行信息处理；
- 寄存器进行信息存储；
- 控制器控制各种器件进行工作；
- 内部总线连接各种器件，在它们之间进行数据的传送。

对于一个汇编程序员来说，CPU 中的主要部件是寄存器。寄存器是 CPU 中程序员可以用指令读写的部件。程序员通过改变各种寄存器中的内容来实现对 CPU 的控制。

不同的 CPU，寄存器的个数、结构是不相同的。8086CPU 有 14 个寄存器，每个寄存器有一个名称。这些寄存器是：AX、BX、CX、DX、SI、DI、SP、BP、IP、CS、SS、DS、ES、PSW。我们不对这些寄存器进行一次性的介绍，在课程的进行中，需要用到哪些寄存器，再介绍哪些寄存器。

2.1　通用寄存器

8086CPU 的所有寄存器都是 16 位的，可以存放两个字节。AX、BX、CX、DX 这 4 个寄存器通常用来存放一般性的数据，被称为通用寄存器。

以 AX 为例，寄存器的逻辑结构如图 2.1 所示。

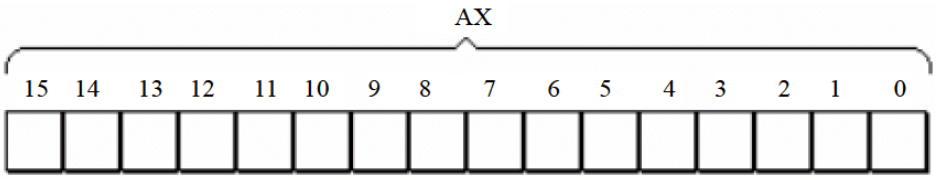


图 2.1　16 位寄存器的逻辑结构

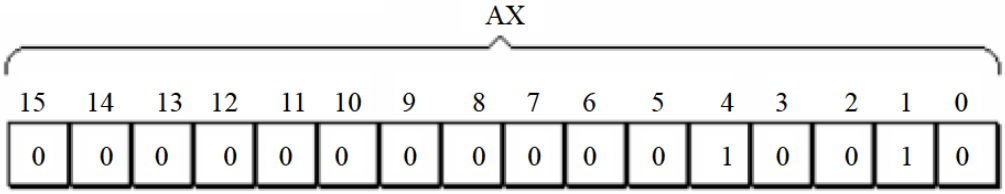
一个 16 位寄存器可以存储一个 16 位的数据，数据在寄存器中的存放情况如图 2.2 所示。

想一想，一个 16 位寄存器所能存储的数据的最大值为多少？

8086CPU 的上一代 CPU 中的寄存器都是 8 位的，为了保证兼容，使原来基于上代 CPU 编写的程序稍加修改就可以运行在 8086 之上，8086CPU 的 AX、BX、CX、DX 这 4 个寄存器都可分为两个可独立使用的 8 位寄存器来用：

- AX 可分为 AH 和 AL；
- BX 可分为 BH 和 BL；
- CX 可分为 CH 和 CL；
- DX 可分为 DH 和 DL。

数据：18
二进制表示：10010
在寄存器 AX 中的存储：



数据：20000
二进制表示：100111000100000
在寄存器 AX 中的存储：

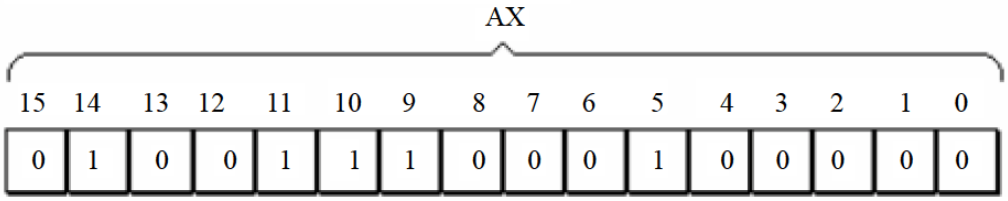


图 2.2　16 位数据在寄存器中的存放情况

以 AX 为例，8086CPU 的 16 位寄存器分为两个 8 位寄存器的情况如图 2.3 所示。

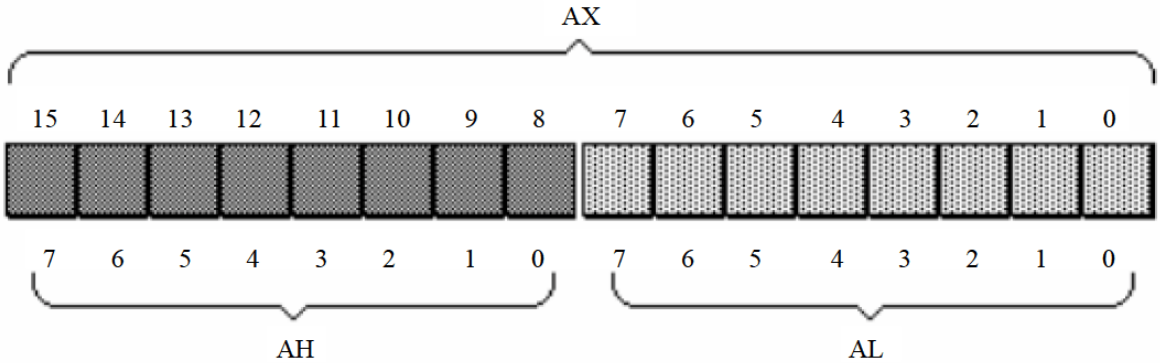
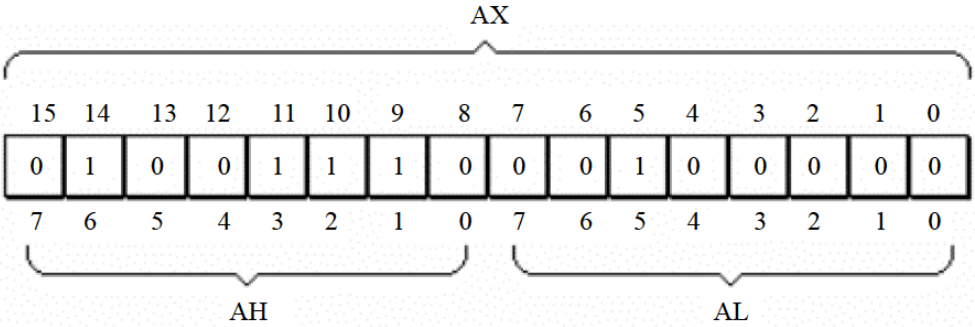


图 2.3　16 位寄存器分为两个 8 位寄存器

AX 的低 8 位(0 位~7 位)构成了 AL 寄存器，高 8 位(8 位~15 位)构成了 AH 寄存器。AH 和 AL 寄存器是可以独立使用的 8 位寄存器。图 2.4 展示了 16 位寄存器及它所分成的两个 8 位寄存器的数据存储的情况。

想一想，一个 8 位寄存器所能存储的数据的最大值为多少？



寄存器	寄存器中的数据	所表示的值
AX	100111000100000	20000(4E20H)
AH	01001110	78(4EH)
AL	00100000	32(20H)

图 2.4　16 位寄存器及所分成的两个 8 位寄存器的数据存储情况

2.2　字在寄存器中的存储

出于对兼容性的考虑，8086CPU 可以一次性处理以下两种尺寸的数据。

- 字节：记为 byte，一个字节由 8 个 bit 组成，可以存在 8 位寄存器中。
- 字：记为 word，一个字由两个字节组成，这两个字节分别称为这个字的高位字节和低位字节，如图 2.5 所示。

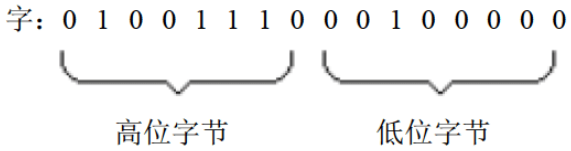


图 2.5　一个字由两个字节组成

一个字可以存在一个 16 位寄存器中，这个字的高位字节和低位字节自然就存在这个寄存器的高 8 位寄存器和低 8 位寄存器中。如图 2.4 所示，一个字型数据 20000，存在 AX 寄存器中，在 AH 中存储了它的高 8 位，在 AL 中存储了它的低 8 位。AH 和 AL 中的数据，既可以看成是一个字型数据的高 8 位和低 8 位，这个字型数据的大小是 20000；又可以看成是两个独立的字节型数据，它们的大小分别是 78 和 32。

关于数制的讨论

任何数据，到了计算机中都是以二进制的形式存放的。为了描述不同的问题，又经常将它们用其他的进制来表示。比如图 2.4 中寄存器 AX 中的数据是 0100111000100000，这就是 AX 中的信息本身，可以用不同的逻辑意义来看待它。可以将它看作一个数值，大小是 20000。

当然，二进制数 0100111000100000 本身也可表示一个数值的大小，但人类习惯的是十进制，用十进制 20000 表示可以使我们直观地感受到这个数值的大小。

十六进制数的一位相当于二进制数的四位，如 0100111000100000 可表示成：4(0100)、E(1110)、2(0010)、0(0000)四位十六进制数。

一个内存单元可存放 8 位数据，CPU 中的寄存器又可存放 n 个 8 位的数据。也就是说，计算机中的数据大多是由 1~N 个 8 位数据构成的。很多时候，需要直观地看出组成数据的各个字节数据的值，用十六进制来表示数据可以直观地看出这个数据是由哪些 8 位数据构成的。比如 20000 写成 4E20 就可以直观地看出，这个数据是由 4E 和 20 两个 8 位数据构成的，如果 AX 中存放 4E20，则 AH 里是 4E，AL 里是 20。这种表示方法便于许多问题的直观分析。在以后的课程中，我们多用十六进制来表示一个数据。

在以后的课程中，为了区分不同的进制，在十六进制表示的数据的后面加 H，在二进制表示的数据后面加 B，十进制表示的数据后面什么也不加。如：可用 3 种不同的进制表示图 2.4 中 AX 里的数据，十进制：20000，十六进制：4E20H，二进制：0100111000100000B。

2.3　几条汇编指令

通过汇编指令控制 CPU 进行工作，看一下表 2.1 中的几条指令。

表 2.1　汇编指令举例

汇编指令	控制 CPU 完成的操作	用高级语言的语法描述
mov ax,18	将 18 送入寄存器 AX	AX=18
mov ah,78	将 78 送入寄存器 AH	AH=78
add ax,8	将寄存器 AX 中的数值加上 8	AX=AX+8
mov ax,bx	将寄存器 BX 中的数据送入寄存器 AX	AX=BX
add ax,bx	将 AX 和 BX 中的数值相加，结果存在 AX 中	AX=AX+BX

注意，为了使具有高级语言基础的读者更好地理解指令的含义，有时会用文字描述和高级语言描述这两种方式来描述一条汇编指令的含义。在写一条汇编指令或一个寄存器的名称时不区分大小写。如：mov ax,18 和 MOV AX,18 的含义相同；bx 和 BX 的含义相同。

接下来看一下 CPU 执行表 2.2 中所列的程序段中的每条指令后，对寄存器中的数据进行的改变。

表 2.2　程序段中指令的执行情况之一(原 AX 中的值：0000H，原 BX 中的值：0000H)

程序段中的指令	指令执行后 AX 中的数据	指令执行后 BX 中的数据
mov ax,4E20H	4E20H	0000H
add ax,1406H	6226H	0000H
mov bx,2000H	6226H	2000H

续表

程序段中的指令	指令执行后 AX 中的数据	指令执行后 BX 中的数据
add ax,bx	8226H	2000H
mov bx,ax	8226H	8226H
add ax,bx	？(参见问题 2.1)	8226H

问题 2.1

指令执行后 AX 中的数据为多少？思考后看分析。

分析：

程序段中的最后一条指令 add ax,bx，在执行前 ax 和 bx 中的数据都为 8226H，相加后所得的值为：1044CH，但是 ax 为 16 位寄存器，只能存放 4 位十六进制的数据，所以最高位的 1 不能在 ax 中保存，ax 中的数据为：044CH。

表 2.3 中所列的一段程序的执行情况。

表 2.3　程序段中指令的执行情况之二(原 AX 中的值：0000H，原 BX 中的值：0000H)

程序段中的指令	指令执行后 AX 中的数据	指令执行后 BX 中的数据
mov ax,001AH	001AH	0000H
mov bx,0026H	001AH	0026H
add al,bl	0040H	0026H
add ah,bl	2640H	0026H
add bh,al	2640H	4026H
mov ah,0	0040H	4026H
add al,85H	00C5H	4026H
add al,93H	？(参见问题 2.2)	4026H

问题 2.2

指令执行后 AX 中的数据为多少？思考后看分析。

分析：

程序段中的最后一条指令 add al,93H，在执行前，al 中的数据为 C5H，相加后所得的值为：158H，但是 al 为 8 位寄存器，只能存放两位十六进制的数据，所以最高位的 1 丢失，ax 中的数据为：0058H。(这里的丢失，指的是进位值不能在 8 位寄存器中保存，但是 CPU 并不真的丢弃这个进位值，关于这个问题，我们将在后面的课程中讨论。)

注意，此时 al 是作为一个独立的 8 位寄存器来使用的，和 ah 没有关系，CPU 在执行

这条指令时认为 ah 和 al 是两个不相关的寄存器。不要错误地认为，诸如 add al,93H 的指令产生的进位会存储在 ah 中，add al,93H 进行的是 8 位运算。

如果执行 add ax,93H，低 8 位的进位会存储在 ah 中，CPU 在执行这条指令时认为只有一个 16 位寄存器 ax，进行的是 16 位运算。指令 add ax,93H 执行后，ax 中的值为：0158H。此时，使用的寄存器是 16 位寄存器 ax，add ax,93H 相当于将 ax 中的 16 位数据 00c5H 和另一个 16 位数据 0093H 相加，结果是 16 位的 0158H。

在进行数据传送或运算时，要注意指令的两个操作对象的位数应当是一致的，例如：

```
mov ax,bx
mov bx,cx
mov ax,18H
mov al,18H
add ax,bx
add ax,20000
```

等都是正确的指令，而：

```
mov ax,bl          (在 8 位寄存器和 16 位寄存器之间传送数据)
mov bh,ax          (在 16 位寄存器和 8 位寄存器之间传送数据)
mov al,20000       (8 位寄存器最大可存放值为 255 的数据)
add al,100H        (将一个高于 8 位的数据加到一个 8 位寄存器中)
```

等都是错误的指令，错误的原因都是指令的两个操作对象的位数不一致。

检测点 2.1

(1) 写出每条汇编指令执行后相关寄存器中的值。

```
mov ax,62627       AX=____________

mov ah,31H         AX=____________

mov al,23H         AX=____________

add ax,ax          AX=____________

mov bx,826CH       BX=____________

mov cx,ax          CX=____________

mov ax,bx          AX=____________

add ax,bx          AX=____________

mov al,bh          AX=____________

mov ah,bl          AX=____________

add ah,ah          AX=____________

add al,6           AX=____________
```

```
add al,al          AX=____________
mov ax,cx          AX=____________
```

(2) 只能使用目前学过的汇编指令，最多使用 4 条指令，编程计算 2 的 4 次方。

2.4 物 理 地 址

我们知道，CPU 访问内存单元时，要给出内存单元的地址。所有的内存单元构成的存储空间是一个一维的线性空间，每一个内存单元在这个空间中都有唯一的地址，我们将这个唯一的地址称为物理地址。

CPU 通过地址总线送入存储器的，必须是一个内存单元的物理地址。在 CPU 向地址总线上发出物理地址之前，必须要在内部先形成这个物理地址。不同的 CPU 可以有不同的形成物理地址的方式。我们现在讨论 8086CPU 是如何在内部形成内存单元的物理地址的。

2.5 16 位结构的 CPU

我们说 8086CPU 的上一代 CPU(8080、8085)等是 8 位机，而 8086 是 16 位机，也可以说 8086 是 16 位结构的 CPU。那么什么是 16 位结构的 CPU 呢？

概括地讲，16 位结构(16 位机、字长为 16 位等常见说法，与 16 位结构的含义相同)描述了一个 CPU 具有下面几方面的结构特性。

- 运算器一次最多可以处理 16 位的数据；
- 寄存器的最大宽度为 16 位；
- 寄存器和运算器之间的通路为 16 位。

8086 是 16 位结构的 CPU，这也就是说，在 8086 内部，能够一次性处理、传输、暂时存储的信息的最大长度是 16 位的。内存单元的地址在送上地址总线之前，必须在 CPU 中处理、传输、暂时存放，对于 16 位 CPU，能一次性处理、传输、暂时存储 16 位的地址。

2.6 8086CPU 给出物理地址的方法

8086CPU 有 20 位地址总线，可以传送 20 位地址，达到 1MB 寻址能力。8086CPU 又是 16 位结构，在内部一次性处理、传输、暂时存储的地址为 16 位。从 8086CPU 的内部结构来看，如果将地址从内部简单地发出，那么它只能送出 16 位的地址，表现出的寻址能力只有 64KB。

8086CPU 采用一种在内部用两个 16 位地址合成的方法来形成一个 20 位的物理地址。

8086CPU 相关部件的逻辑结构如图 2.6 所示。

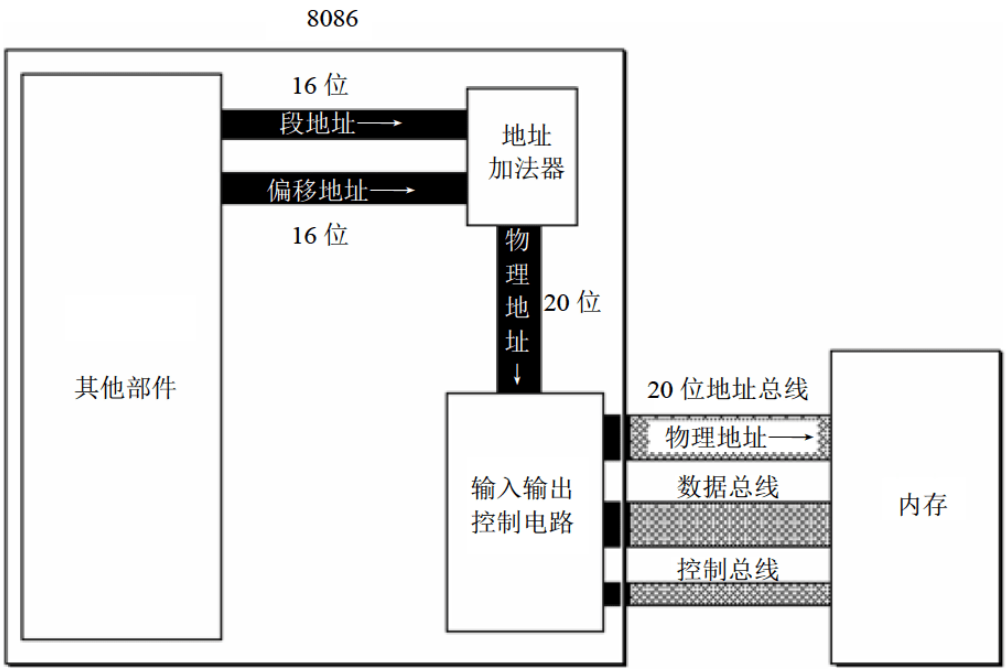


图 2.6　8086CPU 相关部件的逻辑结构

如图 2.6 所示，当 8086CPU 要读写内存时：

(1) CPU 中的相关部件提供两个 16 位的地址，一个称为段地址，另一个称为偏移地址；
(2) 段地址和偏移地址通过内部总线送入一个称为地址加法器的部件；
(3) 地址加法器将两个 16 位地址合成为一个 20 位的物理地址；
(4) 地址加法器通过内部总线将 20 位物理地址送入输入输出控制电路；
(5) 输入输出控制电路将 20 位物理地址送上地址总线；
(6) 20 位物理地址被地址总线传送到存储器。

地址加法器采用**物理地址=段地址×16+偏移地址**的方法用段地址和偏移地址合成物理地址。例如，8086CPU 要访问地址为 123C8H 的内存单元，此时，地址加法器的工作过程如图 2.7 所示(图中数据皆为十六进制表示)。

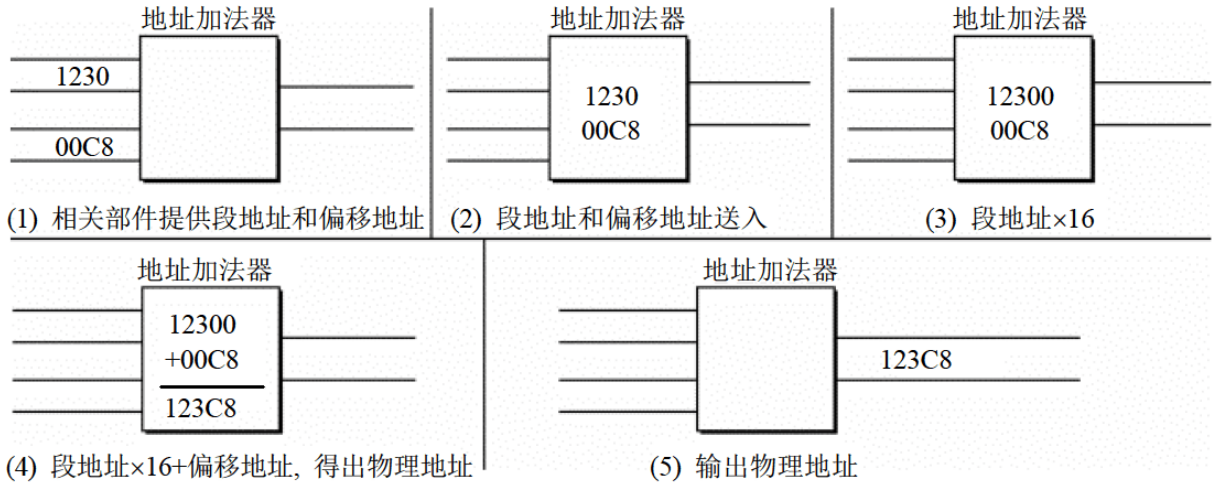


图 2.7　地址加法器的工作过程

由段地址×16 引发的讨论

“段地址×16”有一个更为常用的说法是左移 4 位。计算机中的所有信息都是以二进制的形式存储的，段地址当然也不例外。机器只能处理二进制信息，“左移 4 位”中的位，指的是二进制位。

我们看一个例子，一个数据为 2H，二进制形式为 10B，对其进行左移运算：

左移位数	二进制	十六进制	十进制
0	10B	2H	2
1	100B	4H	4
2	1000B	8H	8
3	10000B	10H	16
4	100000B	20H	32

观察上面移位次数和各种形式数据的关系，我们可以发现：

(1) 一个数据的二进制形式左移 1 位，相当于该数据乘以 2；
(2) 一个数据的二进制形式左移 N 位，相当于该数据乘以 2 的 N 次方；
(3) 地址加法器如何完成段地址×16 的运算？就是将以二进制形式存放的段地址左移 4 位。

进一步思考，我们可看出：一个数据的十六进制形式左移 1 位，相当于乘以 16；一个数据的十进制形式左移 1 位，相当于乘以 10；一个 X 进制的数据左移 1 位，相当于乘以 X。

2.7 “段地址×16+偏移地址=物理地址”的本质含义

注意，这里讨论的是 8086CPU 段地址和偏移地址的本质含义，而不是为了解决具体的问题而在本质含义之上引申出来的更高级的逻辑意义。不管以多少种不同的逻辑意义去看待“段地址×16+偏移地址=物理地址”的寻址模式，一定要清楚地知道它的本质含义，这样才能更灵活地利用它来分析、解决问题。如果只拘泥于某一种引申出来的逻辑含义，而模糊本质含义的话，将从意识上限制对这种寻址功能的灵活应用。

“段地址×16+偏移地址=物理地址”的本质含义是：CPU 在访问内存时，用一个基础地址(段地址×16)和一个相对于基础地址的偏移地址相加，给出内存单元的物理地址。

更一般地说，8086CPU 的这种寻址功能是**“基础地址+偏移地址=物理地址”**寻址模式的一种具体实现方案。8086CPU 中，段地址×16 可看作是基础地址。

下面，我们用两个与 CPU 无关的例子做进一步的比喻说明。

第一个比喻说明“基础地址+偏移地址=物理地址”的思想。

比如说，学校、体育馆、图书馆同在一条笔直的单行路上(参考图 2.8)，学校位于路的起点(从路的起点到学校距离是 0 米)。

图 2.8　学校、体育馆、图书馆的位置关系

你要去图书馆，问我那里的地址，我可以用两种方式告诉你图书馆的地址：

(1)　从学校走 2826m 到图书馆。这 2826m 可以认为是图书馆的物理地址。

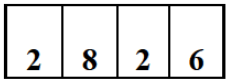
(2)　从学校走 2000m 到体育馆，从体育馆再走 826m 到图书馆。第一个距离 2000m，是相对于起点的基础地址，第二个距离 826m 是相对于基础地址的偏移地址(以基础地址为起点的地址)。

第一种方式是直接给出物理地址 2826m，而第二种方式是用基础地址和偏移地址相加来得到物理地址的。

第二个比喻进一步说明“段地址×16+偏移地址=物理地址”的思想。

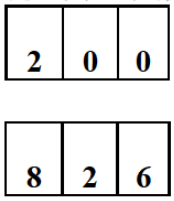
我们为上面的例子加一些限制条件，比如，只能通过纸条来互相通信，你问我图书馆的地址我只能将它写在纸上告诉你。显然，我必须有一张可以容纳 4 位数据的纸条，才能写下 2826 这个数据。

可以写下 4 位数据的纸条



可不巧的是，我没有能容纳 4 位数据的纸条，仅有两张可以容纳 3 位数据的纸条。这样我只能以这种方式告诉你 2826 这个数据。

两张可以写下 3 位数据的纸条



在第一张纸上写上 200(段地址)，在第二张纸上写上 826(偏移地址)。假设我们事前对这种情况又有过相关的约定：你得到这两张纸后，做这样的运算：200(段地址)×10+826(偏移地址)=2826(物理地址)。

8086CPU 就是这样一个只能提供两张 3 位数据纸条的 CPU。

2.8　段 的 概 念

我们注意到，“段地址”这个名称中包含着“段”的概念。这种说法可能对一些学习者产生了误导，使人误以为内存被划分成了一个一个的段，每一个段有一个段地址。如果

我们在一开始形成了这种认识，将影响以后对汇编语言的深入理解和灵活应用。

其实，内存并没有分段，段的划分来自于 CPU，由于 8086CPU 用“基础地址(段地址×16)+偏移地址=物理地址”的方式给出内存单元的物理地址，使得我们可以用分段的方式来管理内存。如图 2.9 所示，我们可以认为：地址 10000H~100FFH 的内存单元组成一个段，该段的起始地址(基础地址)为 10000H，段地址为 1000H，大小为 100H；我们也可以认为地址 10000H~1007FH、10080H~100FFH 的内存单元组成两个段，它们的起始地址(基础地址)为：10000H 和 10080H，段地址为：1000H 和 1008H，大小都为 80H。

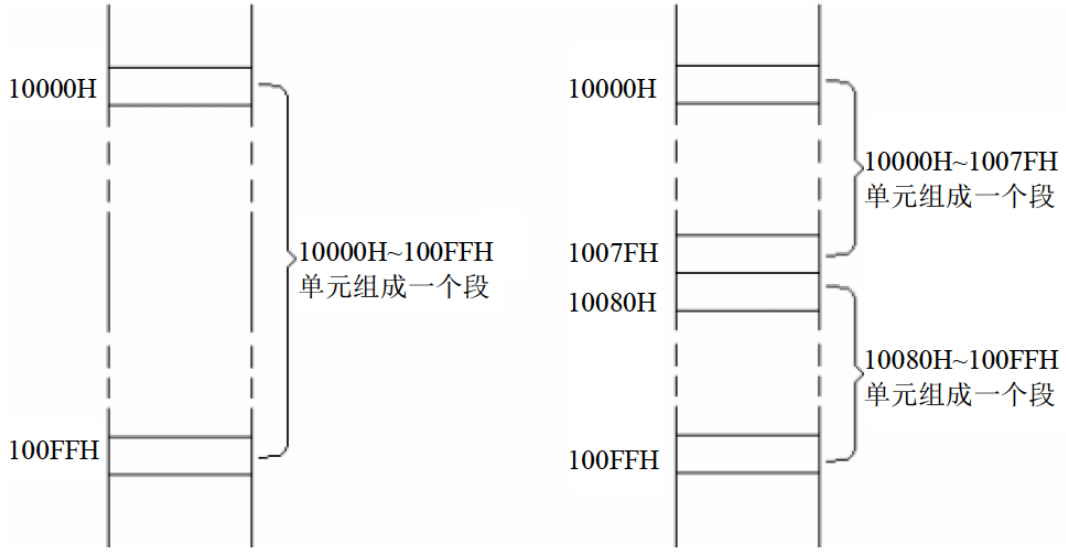


图 2.9 分段

以后，在编程时可以根据需要，将若干地址连续的内存单元看作一个段，用段地址×16 定位段的起始地址(基础地址)，用偏移地址定位段中的内存单元。有两点需要注意：段地址×16 必然是 16 的倍数，所以一个段的起始地址也一定是 16 的倍数；偏移地址为 16 位，16 位地址的寻址能力为 64KB，所以一个段的长度最大为 64KB。

内存单元地址小结

CPU 访问内存单元时，必须向内存提供内存单元的物理地址。8086CPU 在内部用段地址和偏移地址移位相加的方法形成最终的物理地址。

思考下面的两个问题。

(1) 观察下面的地址，你有什么发现？

物理地址	段地址	偏移地址
21F60H	2000H	1F60H
	2100H	0F60H
	21F0H	0060H
	21F6H	0000H
	1F00H	2F60H

结论：CPU 可以用不同的段地址和偏移地址形成同一个物理地址。

比如 CPU 要访问 21F60H 单元，则它给出的段地址 SA 和偏移地址 EA 满足 SA×16+EA=21F60H 即可。

(2) 如果给定一个段地址，仅通过变化偏移地址来进行寻址，最多可定位多少个内存单元？

结论：偏移地址 16 位，变化范围为 0~FFFFH，仅用偏移地址来寻址最多可寻 64KB 个内存单元。

比如给定段地址 1000H，用偏移地址寻址，CPU 的寻址范围为：10000H~1FFFFH。

在 8086PC 机中，存储单元的地址用两个元素来描述，即段地址和偏移地址。

“数据在 21F60H 内存单元中。”这句话对于 8086PC 机一般不这样讲，取而代之的是两种类似的说法：①数据存在内存 2000:1F60 单元中；②数据存在内存的 2000H 段中的 1F60H 单元中。这两种描述都表示“数据在内存 21F60H 单元中”。

可以根据需要，将地址连续、起始地址为 16 的倍数的一组内存单元定义为一个段。

检测点 2.2

(1) 给定段地址为 0001H，仅通过变化偏移地址寻址，CPU 的寻址范围为____到____。

(2) 有一数据存放在内存 20000H 单元中，现给定段地址为 SA，若想用偏移地址寻到此单元。则 SA 应满足的条件是：最小为_____，最大为_____。

提示，反过来思考一下，当段地址给定为多少，CPU 无论怎么变化偏移地址都无法寻到 20000H 单元？

2.9 段 寄 存 器

我们前面讲到，8086CPU 在访问内存时要由相关部件提供内存单元的段地址和偏移地址，送入地址加法器合成物理地址。这里，需要看一下，是什么部件提供段地址。段地址在 8086CPU 的段寄存器中存放。8086CPU 有 4 个段寄存器：CS、DS、SS、ES。当 8086CPU 要访问内存时由这 4 个段寄存器提供内存单元的段地址。本章中只看一下 CS。

2.10 CS 和 IP

CS 和 IP 是 8086CPU 中两个最关键的寄存器，它们指示了 CPU 当前要读取指令的地址。CS 为代码段寄存器，IP 为指令指针寄存器，从名称上我们可以看出它们和指令的关系。

在 8086PC 机中，任意时刻，设 CS 中的内容为 M，IP 中的内容为 N，8086CPU 将从内存 M×16+N 单元开始，读取一条指令并执行。

也可以这样表述：8086 机中，任意时刻，CPU 将 CS:IP 指向的内容当作指令执行。

图 2.10 展示了 8086CPU 读取、执行指令的工作原理(图中只包括了和所要说明的问题

密切相关的部件，图中数字都为十六进制)。

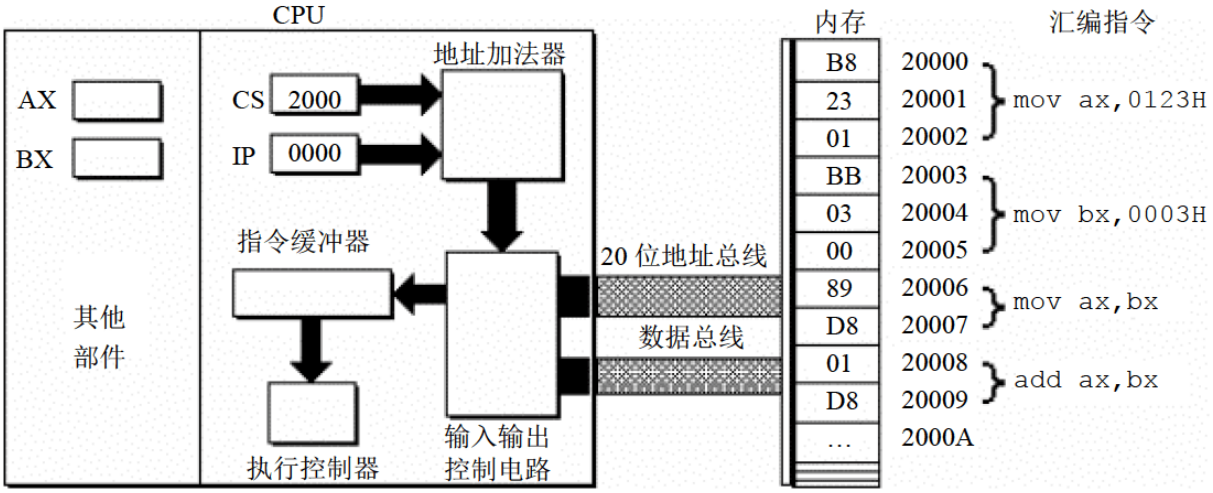


图 2.10　8086PC 读取和执行指令的相关部件

图 2.10 说明如下。

(1)　8086CPU 当前状态：CS 中的内容为 2000H，IP 中的内容为 0000H；

(2)　内存 20000H~20009H 单元存放着可执行的机器码；

(3)　内存 20000H~20009H 单元中存放的机器码对应的汇编指令如下。

地址：20000H~20002H，内容：B8 23 01，长度：3Byte，对应汇编指令：mov ax,0123H

地址：20003H~20005H，内容：BB 03 00，长度：3Byte，对应汇编指令：mov bx,0003H

地址：20006H~20007H，内容：89 D8，长度：2Byte，对应汇编指令：mov ax,bx

地址：20008H~20009H，内容：01 D8，长度：2Byte，对应汇编指令：add ax,bx

下面的一组图(图 2.11~图 2.19)，以图 2.10 描述的情况为初始状态，展示了 8086CPU 读取、执行一条指令的过程。注意每幅图中发生的变化(下面对 8086CPU 的描述，是在逻辑结构、宏观过程的层面上进行的，目的是使读者对 CPU 工作原理有一个清晰、直观的认识，为汇编语言的学习打下基础。其中隐蔽了 CPU 的物理结构以及具体的工作细节)。

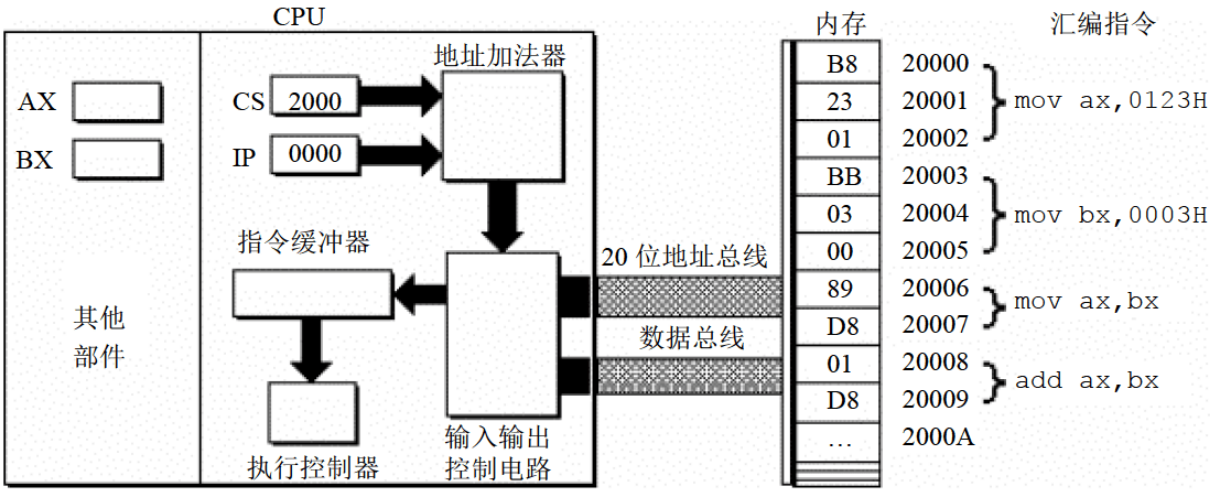


图 2.11　初始状态(CS:2000H，IP:0000H，CPU 将从内存 2000H×16+0000H 处读取指令执行)

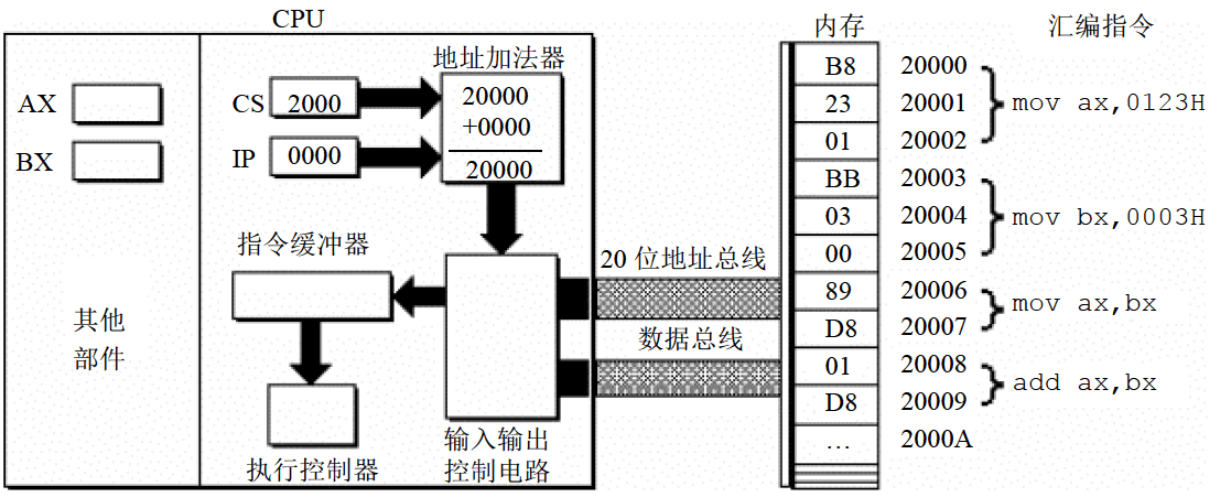


图 2.12　CS、IP 中的内容送入地址加法器(地址加法器完成：物理地址=段地址×16+偏移地址)

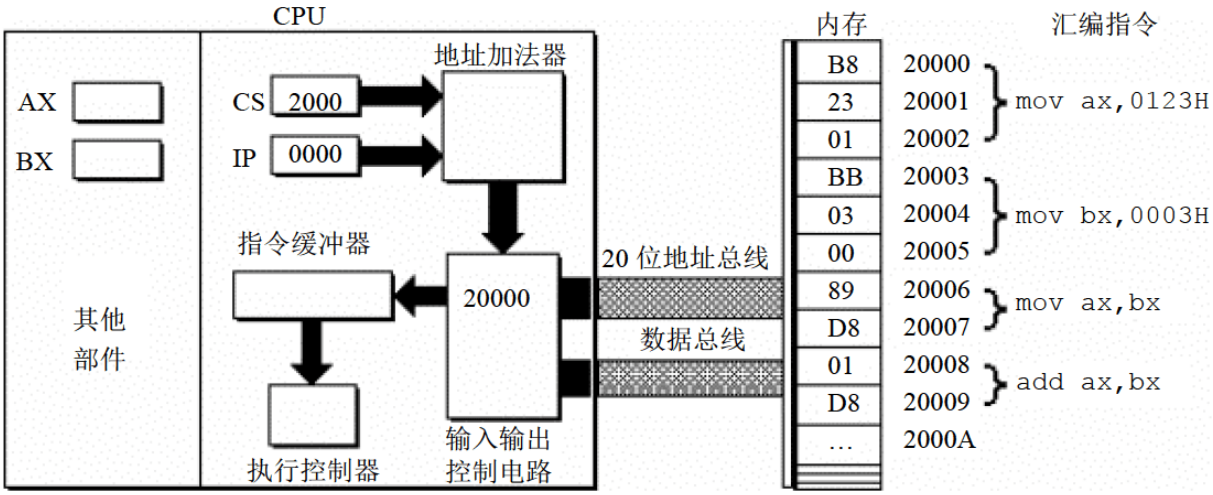


图 2.13　地址加法器将物理地址送入输入输出控制电路

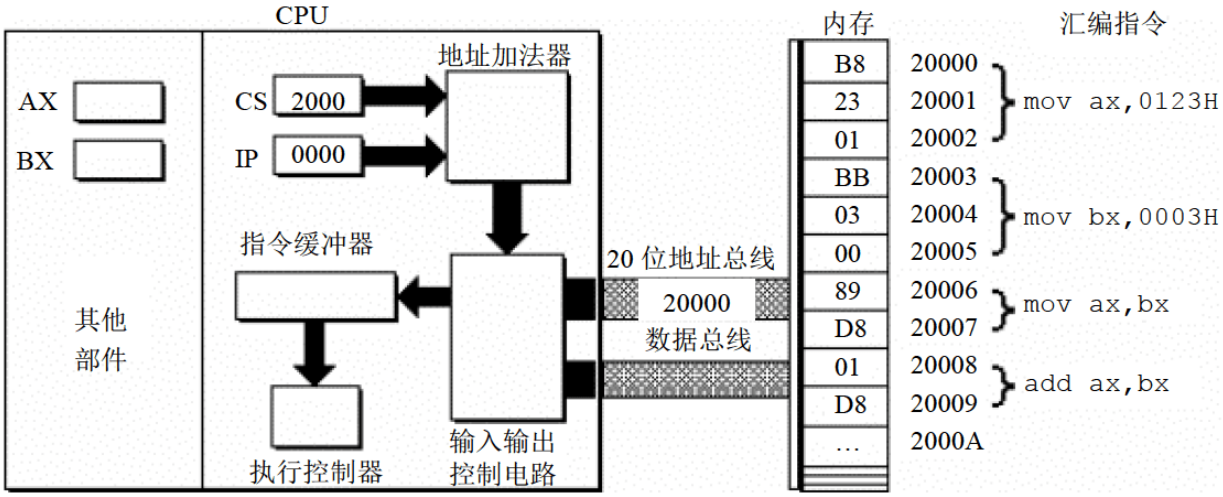


图 2.14　输入输出控制电路将物理地址 20000H 送上地址总线

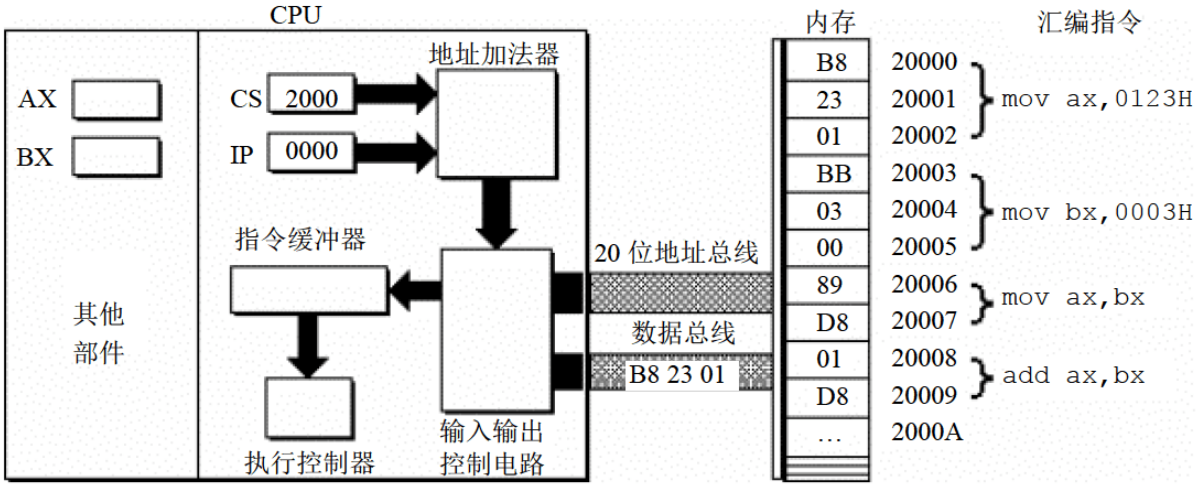


图 2.15　从内存 20000H 单元开始存放的机器指令 B8 23 01 通过数据总线被送入 CPU

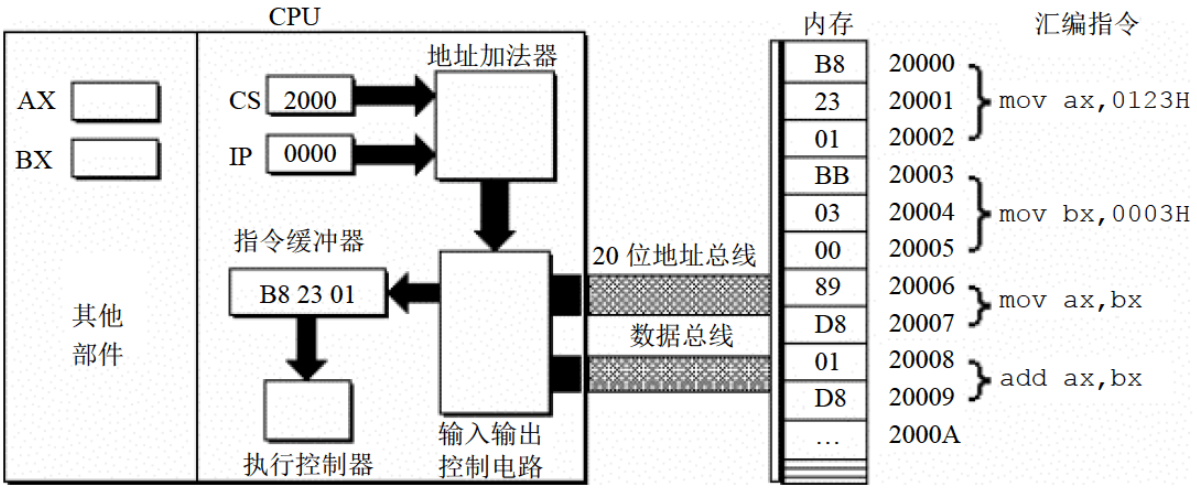


图 2.16　输入输出控制电路将机器指令 B8 23 01 送入指令缓冲器

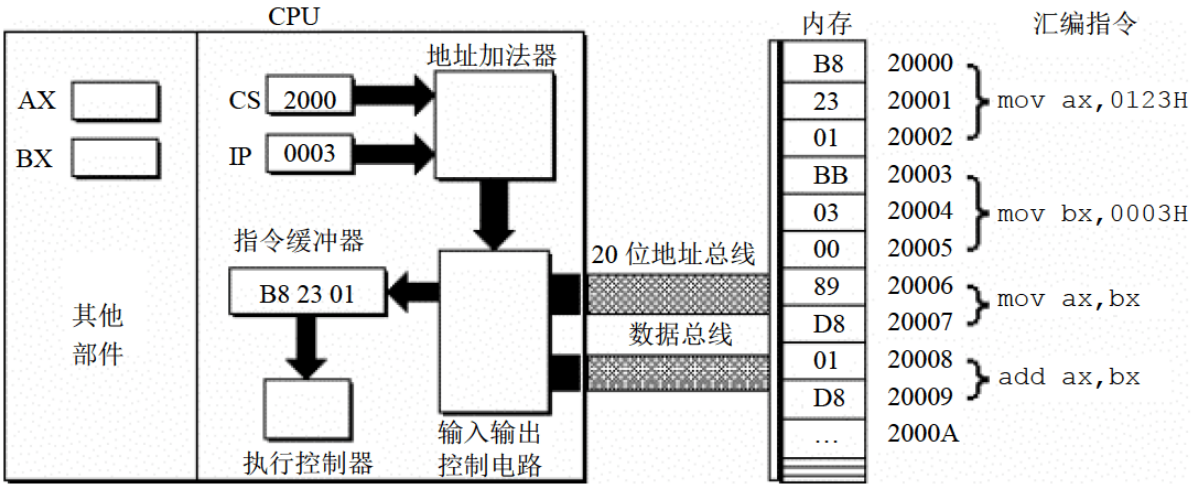


图 2.17　IP 中的值自动增加

(读取一条指令后，IP 中的值自动增加，以使 CPU 可以读取下一条指令。因当前读入的指令 B82301 长度为 3 个字节，所以 IP 中的值加 3。此时，CS:IP 指向内存单元 2000:0003。)

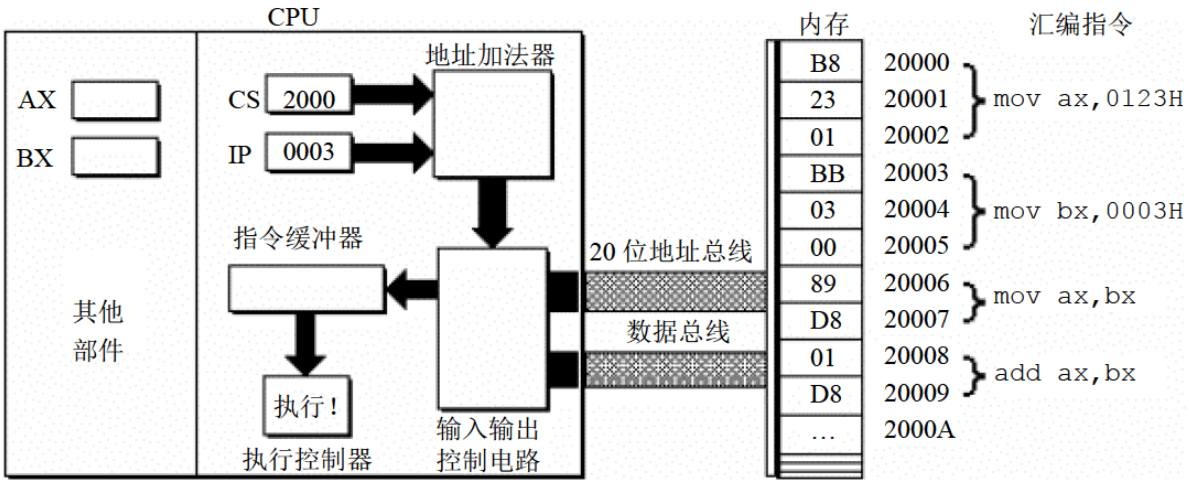


图 2.18　执行控制器执行指令 B8 23 01(即 mov ax,0123H)

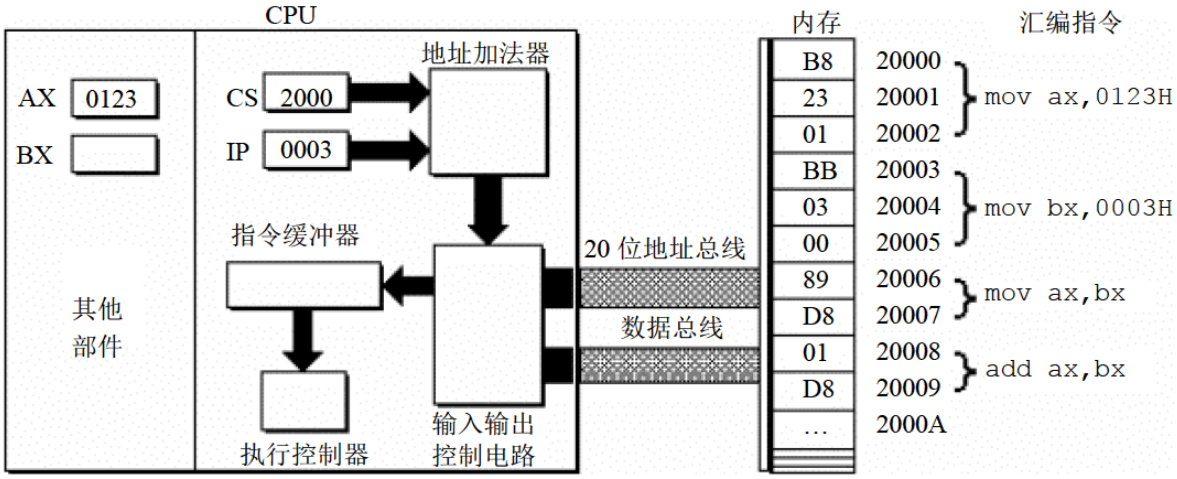


图 2.19　指令 B8 23 01 被执行后 AX 中的内容为 0123H
(此时，CPU 将从内存单元 2000:0003 处读取指令。)

下面的一组图(图 2.20~图 2.26)，以图 2.19 的情况为初始状态，展示了 8086CPU 继续读取、执行 3 条指令的过程。注意 IP 的变化(下面的描述中，隐蔽了读取每条指令的细节)。

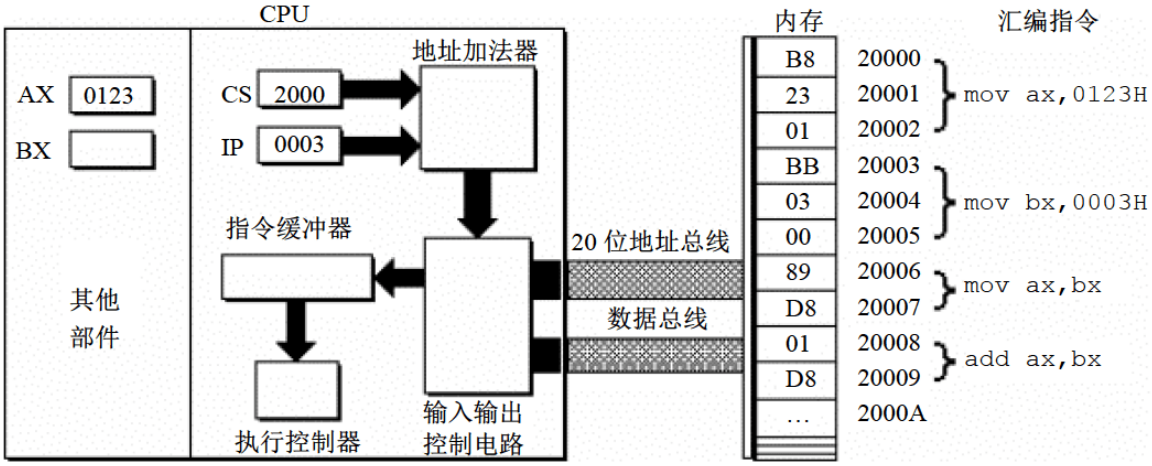


图 2.20　CS:2000H，IP:0003H(CPU 将从内存 2000H×16+0003H 处读取指令 BB 03 00)

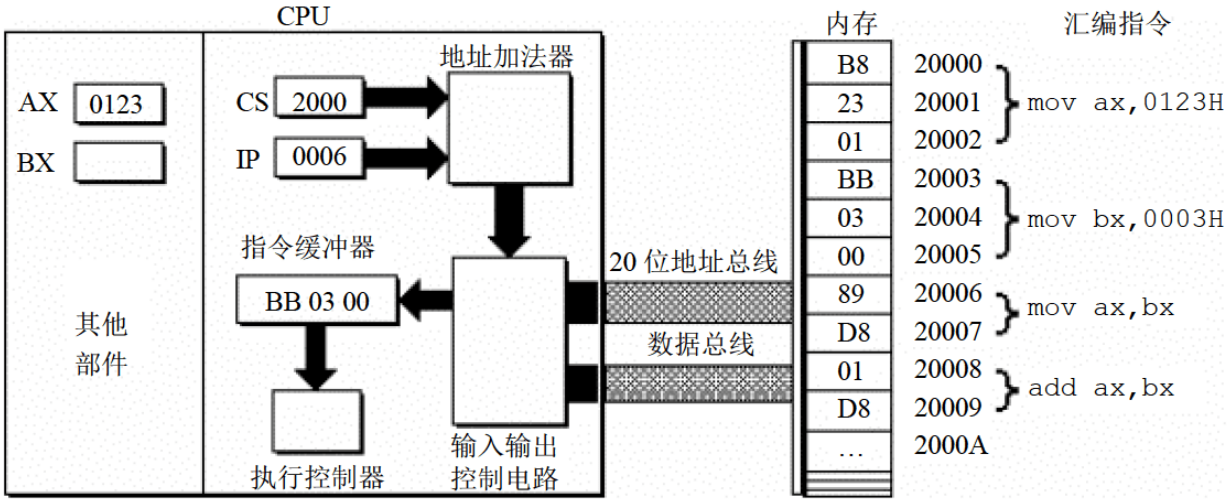


图 2.21 CPU 从内存 20003H 处读取指令 BB 03 00 入指令缓冲器(IP 中的值加 3)

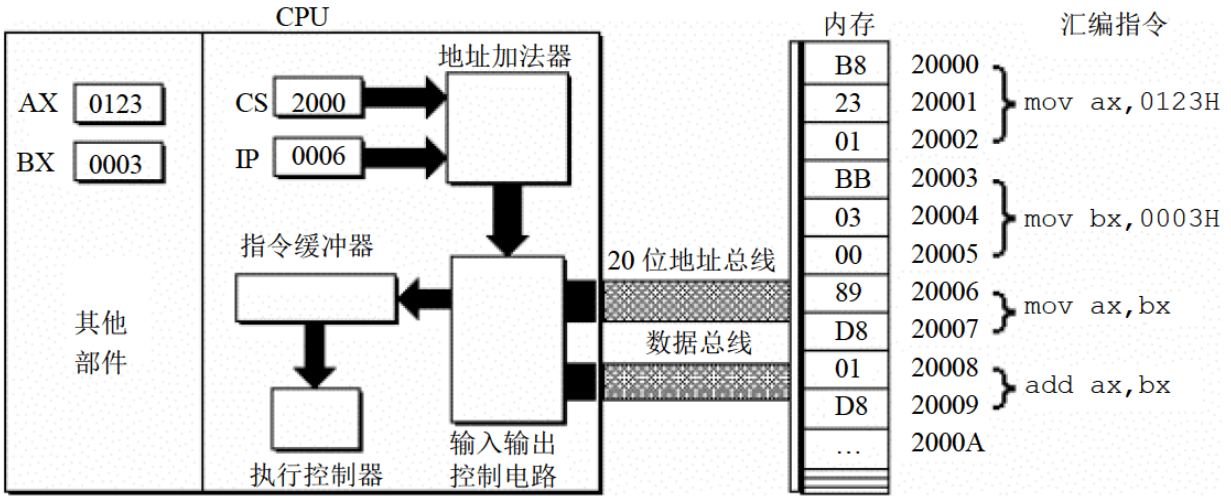


图 2.22 执行指令 BB 03 00(即 mov bx,0003H)

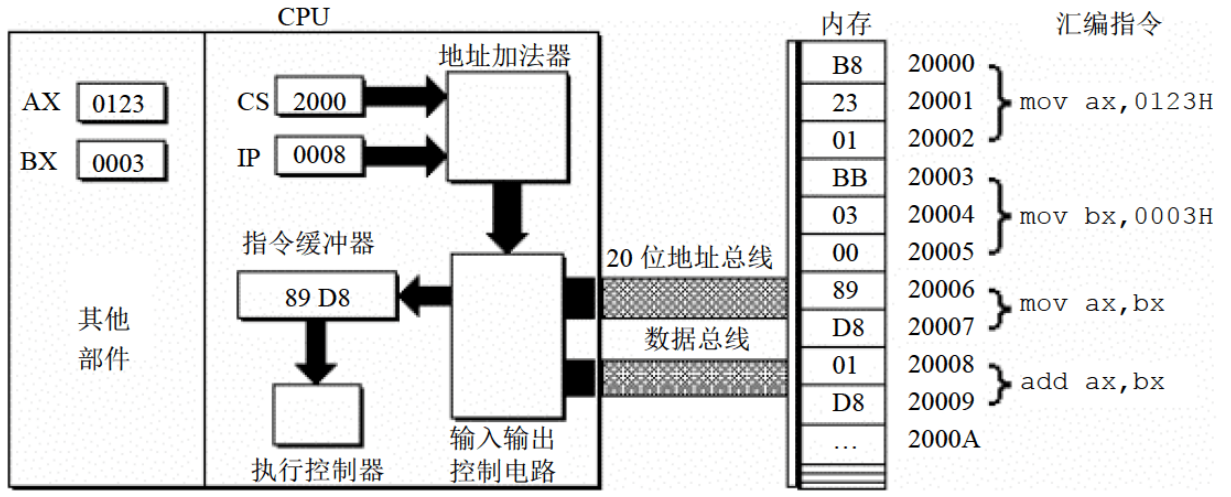


图 2.23 CPU 从内存 20006H 处读取指令 89 D8 入指令缓冲器(IP 中的值加 2)

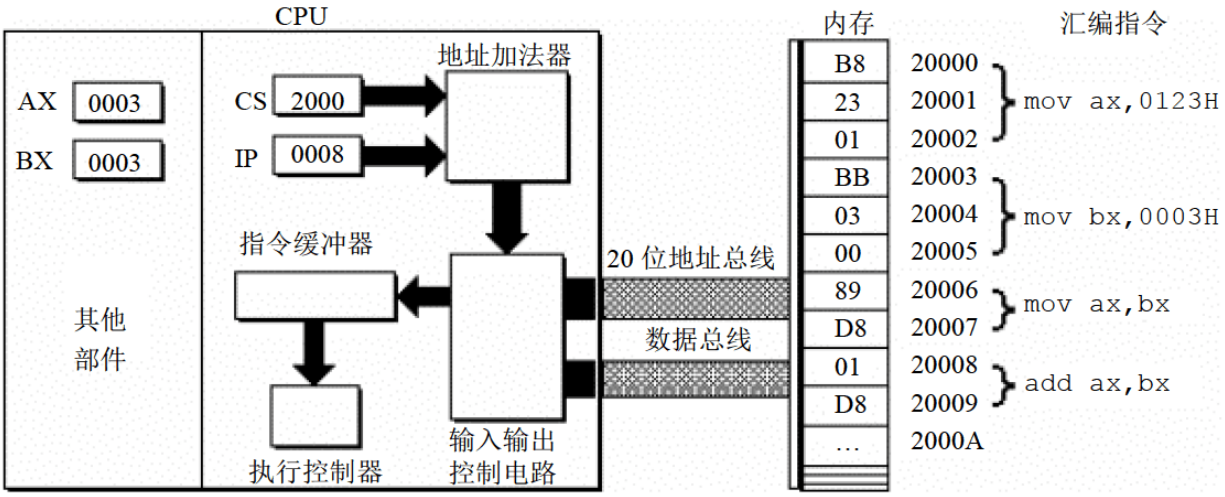


图 2.24　执行指令 89 D8(即 mov ax,bx)后，AX 中的内容为 0003H

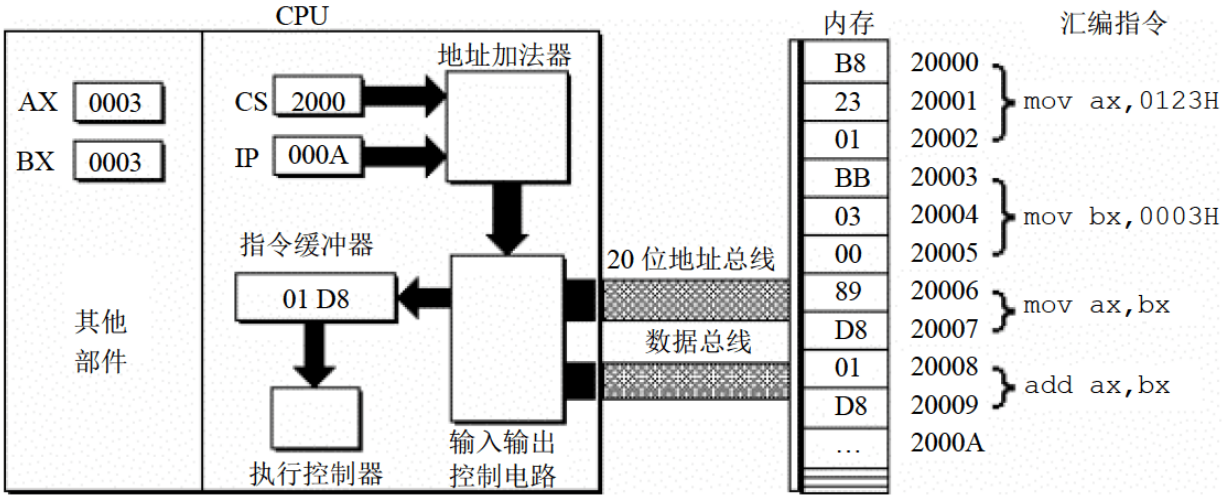


图 2.25　CPU 从内存 20008H 处读取指令 01 D8 入指令缓冲器(IP 中的值加 2)

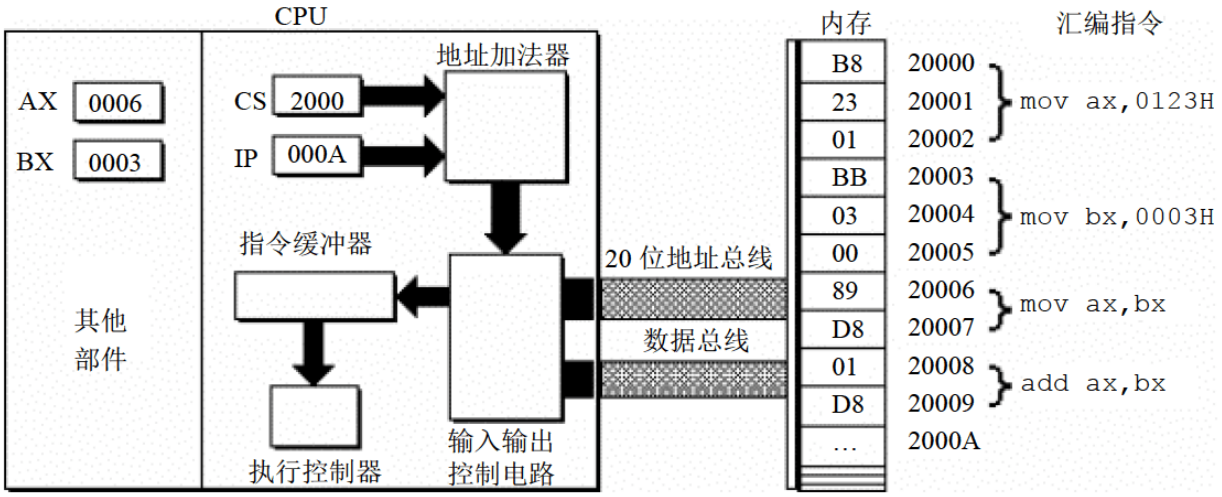


图 2.26　执行指令 01 D8(即 add ax,bx)后，AX 中的内容为 0006H

通过上面的过程展示，8086CPU 的工作过程可以简要描述如下。

(1) 从 CS:IP 指向的内存单元读取指令，读取的指令进入指令缓冲器；
(2) IP=IP+所读取指令的长度，从而指向下一条指令；
(3) 执行指令。转到步骤(1)，重复这个过程。

在 8086CPU 加电启动或复位后(即 CPU 刚开始工作时)CS 和 IP 被设置为 CS=FFFFH，IP=0000H，即在 8086PC 机刚启动时，CPU 从内存 FFFF0H 单元中读取指令执行，FFFF0H 单元中的指令是 8086PC 机开机后执行的第一条指令。

现在，我们更清楚了 CS 和 IP 的重要性，它们的内容提供了 CPU 要执行指令的地址。

我们在第 1 章中讲过，在内存中，指令和数据没有任何区别，都是二进制信息，CPU 在工作的时候把有的信息看作指令，有的信息看作数据。现在，如果提出一个问题：CPU 根据什么将内存中的信息看作指令？如何回答？我们可以说，CPU 将 CS:IP 指向的内存单元中的内容看作指令，因为，在任何时候，CPU 将 CS、IP 中的内容当作指令的段地址和偏移地址，用它们合成指令的物理地址，到内存中读取指令码，执行。如果说，内存中的一段信息曾被 CPU 执行过的话，那么，它所在的内存单元必然被 CS:IP 指向过。

2.11 修改 CS、IP 的指令

在 CPU 中，程序员能够用指令读写的部件只有寄存器，程序员可以通过改变寄存器中的内容实现对 CPU 的控制。CPU 从何处执行指令是由 CS、IP 中的内容决定的，程序员可以通过改变 CS、IP 中的内容来控制 CPU 执行目标指令。

我们如何改变 CS、IP 的值呢？显然，8086CPU 必须提供相应的指令。我们如何修改 AX 中的值？可以用 mov 指令，如 mov ax,123 将 ax 中的值设为 123，显然，我们也可以用同样的方法设置其他寄存器的值，如 mov bx,123，mov cx,123，mov dx,123 等。其实，8086CPU 大部分寄存器的值，都可以用 mov 指令来改变，mov 指令被称为传送指令。

但是，mov 指令不能用于设置 CS、IP 的值，原因很简单，因为 8086CPU 没有提供这样的功能。8086CPU 为 CS、IP 提供了另外的指令来改变它们的值。能够改变 CS、IP 的内容的指令被统称为转移指令(我们以后会深入研究)。我们现在介绍一个最简单的可以修改 CS、IP 的指令：jmp 指令。

若想同时修改 CS、IP 的内容，可用形如“jmp 段地址:偏移地址”的指令完成，如

jmp 2AE3:3，执行后：CS=2AE3H，IP=0003H，CPU 将从 2AE33H 处读取指令。
jmp 3:0B16，执行后：CS=0003H，IP=0B16H，CPU 将从 00B46H 处读取指令。

“jmp 段地址:偏移地址”指令的功能为：用指令中给出的段地址修改 CS，偏移地址修改 IP。

若想仅修改 IP 的内容，可用形如“jmp 某一合法寄存器”的指令完成，如

jmp ax，指令执行前：ax=1000H，CS=2000H，IP=0003H
　　指令执行后：ax=1000H，CS=2000H，IP=1000H
jmp bx，指令执行前：bx=0B16H，CS=2000H，IP=0003H
　　指令执行后：bx=0B16H，CS=2000H，IP=0B16H

“jmp 某一合法寄存器”指令的功能为：用寄存器中的值修改 IP。

jmp ax，在含义上好似：mov IP,ax。

注意，我们在适当的时候，会用已知的汇编指令的语法来描述新学的汇编指令的功能。采用一种“用汇编解释汇编”的方法来使读者更好地理解汇编指令的功能，这样做有助于读者进行知识的相互融会。要强调的是，我们是用“已知的汇编指令的语法”进行描述，并不是用“已知的汇编指令”来描述，比如，我们用 mov IP,ax 来描述 jmp ax，并不是说真有 mov IP,ax 这样的指令，而是用 mov 指令的语法来说明 jmp 指令的功能。我们可以用同样的方法描述 jmp 3:01B6 的功能：jmp 3:01B6 在含义上好似 mov CS,3　mov IP,01B6。

问题 2.3

内存中存放的机器码和对应的汇编指令情况如图 2.27 所示，设 CPU 初始状态：CS=2000H，IP=0000H，请写出指令执行序列。思考后看分析。

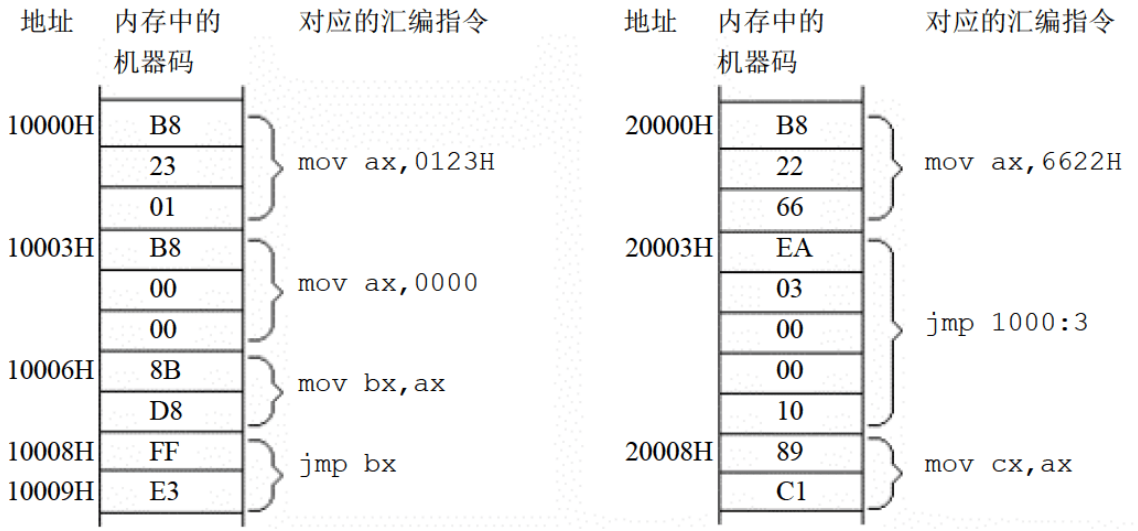


图 2.27　内存中存放的机器码和对应的汇编指令

分析：

CPU 对图 2.27 中的指令的执行过程如下。

(1)　当前 CS=2000H，IP=0000H，则 CPU 从内存 2000H×16+0=20000H 处读取指令，读入的指令是：B8 22 66(mov ax,6622H)，读入后 IP=IP+3=0003H；

(2)　指令执行后，CS=2000H，IP=0003H，则 CPU 从内存 2000H×16+0003H=20003H

处读取指令，读入的指令是：EA 03 00 00 10(jmp 1000:0003)，读入后 IP=IP+5=0008H；

(3) 指令执行后，CS=1000H，IP=0003H，则 CPU 从内存 1000H×16+0003H=10003H 处读取指令，读入的指令是：B8 00 00(mov ax,0000)，读入后 IP=IP+3=0006H；

(4) 指令执行后，CS=1000H，IP=0006H，则 CPU 从内存 1000H×16+0006H=10006H 处读取指令，读入的指令是：8B D8(mov bx,ax)，读入后 IP=IP+2=0008H；

(5) 指令执行后，CS=1000H，IP=0008H，则 CPU 从内存 1000H×16+0008H=10008H 处读取指令，读入的指令是：FF E3(jmp bx)，读入后 IP=IP+2=000AH；

(6) 指令执行后，CS=1000H，IP=0000H，CPU 从内存 10000H 处读取指令……

经分析后，可知指令执行序列为：

(1) mov ax,6622H
(2) jmp 1000:3
(3) mov ax,0000
(4) mov bx,ax
(5) jmp bx
(6) mov ax,0123H
(7) 转到第 3 步执行

2.12 代 码 段

前面讲过，对于 8086PC 机，在编程时，可以根据需要，将一组内存单元定义为一个段。我们可以将长度为 N(N≤64KB)的一组代码，存在一组地址连续、起始地址为 16 的倍数的内存单元中，我们可以认为，这段内存是用来存放代码的，从而定义了一个代码段。比如，将：

```
mov ax,0000      (B8 00 00)
add ax,0123H     (05 23 01)
mov bx,ax        (8B D8)
jmp bx           (FF E3)
```

这段长度为 10 个字节的指令，存放在 123B0H~123B9H 的一组内存单元中，我们就可以认为，123B0H~123B9H 这段内存是用来存放代码的，是一个代码段，它的段地址为 123BH，长度为 10 个字节。

如何使得代码段中的指令被执行呢？将一段内存当作代码段，仅仅是我们在编程时的一种安排，CPU 并不会由于这种安排，就自动地将我们定义的代码段中的指令当作指令来执行。CPU 只认被 CS:IP 指向的内存单元中的内容为指令。所以，要让 CPU 执行我们放在代码段中的指令，必须要将 CS:IP 指向所定义的代码段中的第一条指令的首地址。对于上面的例子，我们将一段代码存放在 123B0H~123B9H 内存单元中，将其定义为代码段，如果要让这段代码得到执行，可设 CS=123BH、IP=0000H。

2.9～2.12　小　　结

(1) 段地址在 8086CPU 的段寄存器中存放。当 8086CPU 要访问内存时，由段寄存器提供内存单元的段地址。8086CPU 有 4 个段寄存器，其中 CS 用来存放指令的段地址。

(2) CS 存放指令的段地址，IP 存放指令的偏移地址。

8086 机中，任意时刻，CPU 将 CS:IP 指向的内容当作指令执行。

(3) 8086CPU 的工作过程：

① 从 CS:IP 指向的内存单元读取指令，读取的指令进入指令缓冲器；
② IP 指向下一条指令；
③ 执行指令。(转到步骤①，重复这个过程。)

(4) 8086CPU 提供转移指令修改 CS、IP 的内容。

检测点 2.3

下面的 3 条指令执行后，CPU 几次修改 IP？都是在什么时候？最后 IP 中的值是多少？

```
mov ax,bx
sub ax,ax
jmp ax
```

实验 1　查看 CPU 和内存，用机器指令和汇编指令编程

1. 预备知识：Debug 的使用

我们以后所有的实验中，都将用到 Debug 程序，首先学习一下它的主要用法。

(1)　什么是 Debug?

Debug 是 DOS、Windows 都提供的实模式(8086 方式)程序的调试工具。使用它，可以查看 CPU 各种寄存器中的内容、内存的情况和在机器码级跟踪程序的运行。

(2)　我们用到的 Debug 功能。

- 用 Debug 的 R 命令查看、改变 CPU 寄存器的内容；
- 用 Debug 的 D 命令查看内存中的内容；
- 用 Debug 的 E 命令改写内存中的内容；
- 用 Debug 的 U 命令将内存中的机器指令翻译成汇编指令；
- 用 Debug 的 T 命令执行一条机器指令；
- 用 Debug 的 A 命令以汇编指令的格式在内存中写入一条机器指令。

Debug 的命令比较多，共有 20 多个，但这 6 个命令是和汇编学习密切相关的。在以后的实验中，我们还会用到一个 P 命令。

(3) 进入 Debug。

Debug 是在 DOS 方式下使用的程序。我们在进入 Debug 前，应先进入到 DOS 方式。用以下方式可以进入 DOS。

① 重新启动计算机，进入 DOS 方式，此时进入的是实模式的 DOS。
② 在 Windows 中进入 DOS 方式，此时进入的是虚拟 8086 模式的 DOS。

下面说明在 Windows 2000 中进入 Debug 的一种方法，在其它 Windows 系统中进入的方法与此类似。

选择【开始】菜单中的【运行】命令，如图 2.28 所示，打开【运行】对话框，如图 2.29 所示，在文本框中输入“command”后，单击【确定】按钮。

图 2.28 选择【运行】命令

图 2.29 在文本框中输入“command”

进入 DOS 方式后，如果显示为窗口方式，可以按下 Alt+Enter 组合键将窗口变为全屏方式。然后运行 Debug 程序，如图 2.30 所示。这个程序在不同的 Windows 系统中所在的路径不尽相同，在 Windows 2000 中通常在 c:\winnt\system 下。由于系统指定了搜索路径，所以在任何一个路径中都可以运行。

```
Microsoft(R) Windows DOS
(C)Copyright Microsoft Corp 1990-1999.

C:\>debug
_
```

图 2.30 运行 Debug 程序

(4) 用 R 命令查看、改变 CPU 寄存器的内容。

我们已经知道了 AX、BX、CX、DX、CS、IP 这 6 个寄存器，现在看一下它们之中的内容，如图 2.31 所示。其他寄存器如 SP、BP、SI、DI、DS、ES、SS、标志寄存器等我们先不予理会。

```
Microsoft(R) Windows DOS
(C)Copyright Microsoft Corp 1990-1999.

C:\>debug
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0CA2  ES=0CA2  SS=0CA2  CS=0CA2  IP=0100   NV UP EI PL NZ NA PO NC
0CA2:0100 027548        ADD     DH,[DI+48]                        DS:0048=00
-
```

图 2.31　使用 R 命令查看 CPU 中各个寄存器中的内容

注意 CS 和 IP 的值，CS=0CA2，IP=0100，也就是说，内存 0CA2:0100 处的指令为 CPU 当前要读取、执行的指令。在所有寄存器的下方，Debug 还列出了 CS:IP 所指向的内存单元处所存放的机器码，并将它翻译为汇编指令。可以看到，CS:IP 所指向的内存单元为 0CA2:0100，此处存放的机器码为 02 75 48，对应的汇编指令为 ADD DH,[DI+48](这条指令的含义我们还不知道，先不必深究)。

Debug 输出的右下角还有一个信息：“DS:0048=0”，我们以后会进行说明，这里同样不必深究。

还可以用 R 命令来改变寄存器中的内容，如图 2.32 所示。

```
C:\>debug
-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   NV UP EI PL NZ NA PO NC
0B39:0100 40            INC     AX
-r ax
AX 0000
:1111
-r
AX=1111  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B39  ES=0B39  SS=0B39  CS=0B39  IP=0100   NV UP EI PL NZ NA PO NC
0B39:0100 40            INC     AX
-
```

图 2.32　用 R 命令修改寄存器 AX 中的内容

若要修改一个寄存器中的值，比如 AX 中的值，可用 R 命令后加寄存器名来进行，输入“r ax”后按 Enter 键，将出现“：”作为输入提示，在后面输入要写入的数据后按 Enter 键，即完成了对 AX 中内容的修改。若想看一下修改的结果，可再用 R 命令查看，如图 2.32 所示。

在图 2.33 中，一进入 Debug，用 R 命令查看，CS:IP 指向 0B39:0100，此处存放的机器码为 40，对应的汇编指令是 INC AX；

```
C:\>debug
-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   NV UP EI PL NZ NA PO NC
0B39:0100 40            INC     AX
-r ip
IP 0100
:200
-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=0200   NV UP EI PL NZ NA PO NC
0B39:0200 5B            POP     BX
-r cs
CS 0B39
:ff00
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B39  ES=0B39  SS=0B39  CS=FF00  IP=0200   NV UP EI PL NZ NA PO NC
FF00:0200 51            PUSH    CX
-
```

图 2.33　用 R 命令修改 CS 和 IP 中的内容

接着，用 R 命令将 IP 修改为 200，则 CS:IP 指向 0B39:0200，此处存放的机器码为 5B，对应的汇编指令是 POP BX；

接着，用 R 命令将 CS 修改为 ff00，则 CS:IP 指向 ff00:0200，此处存放的机器码为 51，对应的汇编指令是 PUSH CX。

(5) 用 Debug 的 D 命令查看内存中的内容。

用 Debug 的 D 命令，可以查看内存中的内容，D 命令的格式较多，这里只介绍在本次实验中用到的格式。

如果我们想知道内存 10000H 处的内容，可以用“d 段地址:偏移地址”的格式来查看，如图 2.34 所示。

```
C:\>debug
-d 1000:0
1000:0000  72 64 73 20 63 6F 6D 6D-65 6E 74 73 20 28 72 65   rds comments (re
1000:0010  6D 61 72 6B 73 29 20 69-6E 20 61 20 62 61 74 63   marks) in a batc
1000:0020  68 20 66 69 6C 65 20 6F-72 20 43 4F 4E 46 49 47   h file or CONFIG
1000:0030  2E 53 59 53 2E 0D 0A 0D-0A 52 45 4D 20 5B 63 6F   .SYS.....REM [co
1000:0040  6D 6D 65 6E 74 5D 0D 0A-6B 53 75 73 70 65 6E 64   mment]..kSuspend
1000:0050  73 20 70 72 6F 63 65 73-73 69 6E 67 20 6F 66 20   s processing of
1000:0060  61 20 62 61 74 63 68 20-70 72 6F 67 72 61 6D 20   a batch program
1000:0070  61 6E 64 20 64 69 73 70-6C 61 79 73 20 74 68 65   and displays the
-
```

图 2.34 用 D 命令查看内存 1000:0 处的内容

要查看内存 10000H 处的内容，首先将这个地址表示为段地址:偏移地址的格式，可以是 1000:0，然后用“d 1000:0”列出 1000:0 处的内容。

使用“d 段地址:偏移地址”的格式，Debug 将列出从指定内存单元开始的 128 个内存单元的内容。图 2.34 中，在使用 d 1000:0 后，Debug 列出了 1000:0~1000:7F 中的内容。

使用 D 命令，Debug 将输出 3 部分内容(如图 2.34 所示)。

① 中间是从指定地址开始的 128 个内存单元的内容，用十六进制的格式输出，每行的输出从 16 的整数倍的地址开始，最多输出 16 个单元的内容。从图中，我们可以知道，内存 1000:0 单元中的内容是 72H，内存 1000:1 单元中的内容是 64H，内存 1000:0~1000:F 中的内容都在第一行；内存 1000:10 中的内容是 6DH，内存 1000:11 处的内容是 61H，内存 1000:10~1000:1F 中的内容都在第二行。注意在每行的中间有一个“-”，它将每行的输出分为两部分，这样便于查看。比如，要想从图中找出 1000:6B 单元中内容，可以从 1000:60 找到行，“-”前面是 1000:60~1000:67 的 8 个单元，后面是 1000:68~1000:6F 的 8 个单元，这样我们就可以从 1000:68 单元向后数 3 个单元，找到 1000:6B 单元，可以看到，1000:6B 中的内容为 67H。

② 左边是每行的起始地址。

③　右边是每个内存单元中的数据对应的可显示的 ASCII 码字符。比如，内存单元 1000:0、1000:1、1000:2 中存放的数据是 72H、64H、73H，它对应的 ASCII 字符分别是“r”、“d”、“s”；内存单元 1000:36 中的数据是 0AH，它没有对应可显示的 ASCII 字符，Debug 就用“.”来代替。

注意，我们看到的内存中的内容，在不同的计算机中是不一样的，也可能每次用 Debug 看到的内容都不相同，因为我们用 Debug 看到的都是原来就在内存中的内容，这些内容受随时都有可能变化的系统环境的影响。当然，我们也可以改变内存、寄存器中的内容。

我们使用 d 1000:9 查看 1000:9 处的内容，Debug 将怎样输出呢？如图 2.35 所示。

```
C:\>debug
-d1000:9
1000:0000                                 6E 74 73 20 28 72 65          nts (re
1000:0010  6D 61 72 6B 73 29 20 69-6E 20 61 20 62 61 74 63   marks) in a batc
1000:0020  68 20 66 69 6C 65 20 6F-72 20 43 4F 4E 46 49 47   h file or CONFIG
1000:0030  2E 53 59 53 2E 0D 0A 0D-0A 52 45 4D 20 5B 63 6F   .SYS.....REM [co
1000:0040  6D 6D 65 6E 74 5D 0D 0A-6B 53 75 73 70 65 6E 64   mment]..kSuspend
1000:0050  73 20 70 72 6F 63 65 73-73 69 6E 67 20 6F 66 20   s processing of
1000:0060  61 20 62 61 74 63 68 20-70 72 6F 67 72 61 6D 20   a batch program
1000:0070  61 6E 64 20 64 69 73 70-6C 61 79 73 20 74 68 65   and displays the
1000:0080  20 6D 65 73 73 61 67 65-20                         message
-
```

图 2.35　查看 1000:9 处的内容

Debug 从 1000:9 开始显示，一直到 1000:88，一共是 128 个字节。第一行中的 1000:0~1000:8 单元中的内容不显示。

在一进入 Debug 后，用 D 命令直接查看，将列出 Debug 预设的地址处的内容，如图 2.36 所示。

```
C:\>debug
-d
0CA2:0100  02 75 48 89 3E E6 99 FF-06 E6 99 C6 06 E8 99 FF   .uH.>...........
0CA2:0110  C6 06 E9 99 00 E8 99 00-AC E8 29 E2 34 00 91 0C   ..........).4...
0CA2:0120  74 34 3A 06 B6 96 74 2E-3A C3 74 2A 3C 3A 74 03   t4:...t.:.t*<:t.
0CA2:0130  E9 5F FF 80 3E A4 98 02-75 05 E8 74 00 EB D9 46   ._..>...u..t...F
0CA2:0140  EB 14 E9 4D FF BA 89 8A-E9 93 E5 BA B1 8B E9 8D   ...M............
0CA2:0150  E5 4E 5F 9D F9 C3 4E EB-51 80 CF 01 81 CD 00 80   .N_...N.Q.......
0CA2:0160  E8 DA E1 46 E8 AC DF 74-0D E8 45 00 AC E8 41 00   ...F...t..E...A.
0CA2:0170  81 CD 00 40 EB 34 3C 0D-75 09 B0 00 AA 81 CD 00   ...@.4<.u.......
-
```

图 2.36　列出 Debug 预设的地址处的内容

在使用“d 段地址:偏移地址”之后，接着使用 D 命令，可列出后续的内容，如图 2.37 所示。

也可以指定 D 命令的查看范围，此时采用“d 段地址:起始偏移地址 结尾偏移地址”的格式。比如要看 1000:0~1000:9 中的内容，可以用“d 1000:0 9”实现，如图 2.38 所示。

```
C:\>debug
-d1000:0
1000:0000  72 64 73 20 63 6F 6D 6D-65 6E 74 73 20 28 72 65   rds comments (re
1000:0010  6D 61 72 6B 73 29 20 69-6E 20 61 20 62 61 74 63   marks) in a batc
1000:0020  68 20 66 69 6C 65 20 6F-72 20 43 4F 4E 46 49 47   h file or CONFIG
1000:0030  2E 53 59 53 2E 0D 0A 0D-0A 52 45 4D 20 5B 63 6F   .SYS.....REM [co
1000:0040  6D 6D 65 6E 74 5D 0D 0A-6B 53 75 73 70 65 6E 64   mment]..kSuspend
1000:0050  73 20 70 72 6F 63 65 73-73 69 6E 67 20 6F 66 20   s processing of
1000:0060  61 20 62 61 74 63 68 20-70 72 6F 67 72 61 6D 20   a batch program
1000:0070  61 6E 64 20 64 69 73 70-6C 61 79 73 20 74 68 65   and displays the
-d
1000:0080  20 6D 65 73 73 61 67 65-20 22 50 72 65 73 73 20    message "Press
1000:0090  61 6E 79 0D 0A 6B 65 79-20 74 6F 20 63 6F 6E 74   any..key to cont
1000:00A0  69 6E 75 65 2E 2E 2E 2E-22 0D 0A 0D 0A 50 41 55   inue...."....PAU
1000:00B0  53 45 0D 0A 4D 44 69 73-70 6C 61 79 73 20 6D 65   SE..MDisplays me
1000:00C0  73 73 61 67 65 73 2C 20-6F 72 20 74 75 72 6E 73   ssages, or turns
1000:00D0  20 63 6F 6D 6D 61 6E 64-2D 65 63 68 6F 69 6E 67    command-echoing
1000:00E0  20 6F 6E 20 6F 72 20 6F-66 66 2E 0D 0A 0D 0A 20    on or off.....
1000:00F0  20 45 43 48 4F 20 5B 4F-4E 20 7C 20 4F 46 46 5D    ECHO [ON | OFF]
-
```

图 2.37　列出后续的内容

```
C:\>debug
-d1000:0 9
1000:0000  72 64 73 20 63 6F 6D 6D-65 6E                     rds commen
-
```

图 2.38　查看 1000:0~1000:9 单元中的内容

如果我们就想查看内存单元 10000H 中的内容，可以用图 2.39 中的任何一种方法看到，因为图中的所有“段地址:偏移地址”都表示了 10000H 这一物理地址。

```
C:\>debug
-d 1000:0 0
1000:0000  72                                                r
-
-d 0fff:10 10
0FFF:0010  72                                                r
-
-d 0100:f000 f000
0100:F000  72                                                r
-
-
```

图 2.39　用 3 种不同的段地址和偏移地址查看同一个物理地址中的内容

(6)　用 Debug 的 E 命令改写内存中的内容。

可以使用 E 命令来改写内存中的内容，比如，要将内存 1000:0~1000:9 单元中的内容分别写为 0、1、2、3、4、5、6、7、8、9，可以用“e 起始地址 数据 数据 数据 ……”的格式来进行，如图 2.40 所示。

```
C:\>debug
-d 1000:0 f
1000:0000  72 64 73 20 63 6F 6D 6D-65 6E 74 73 20 28 72 65   rds comments (re
-
-e 1000:0 0 1 2 3 4 5 6 7 8 9
-
-d 1000:0 f
1000:0000  00 01 02 03 04 05 06 07-08 09 74 73 20 28 72 65   ..........ts (re
-
```

图 2.40　用 E 命令修改从 1000:0 开始的 10 个单元的内容

图 2.40 中，先用 D 命令查看 1000:0~1000:f 单元的内容，再用 E 命令修改从 1000:0 开始的 10 个单元的内容，最后用 D 命令查看 1000:0~1000:f 中内容的变化。

也可以采用提问的方式来一个一个地改写内存中的内容，如图 2.41 所示。

```
-d 1000:10 19
1000:0010  6D 61 72 6B 73 29 20 69-6E 20                    marks) in
-e 1000:10
1000:0010  6D.0    61.1    72.2    6B.1c
-
```

图 2.41　用 E 命令修改从 1000:10 开始的 4 个单元的内容

如图 2.41 中，可以用 E 命令以提问的方式来逐个地修改从某一地址开始的内存单元中的内容，以从 1000:10 单元开始为例，步骤如下。

① 输入 e 1000:10，按 Enter 键。

② Debug 显示起始地址 1000:0010，和第一单元(即 1000:0010 单元)的原始内容：6D，然后光标停在“.”的后面提示输入想要写入的数据，此时可以有两个选择：其一为输入数据(我们输入的是 0)，然后按空格键，即用输入的数据改写当前的内存单元；其二为不输入数据，直接按空格键，则不对当前内存单元进行改写。

③ 当前单元处理完成后(不论是改写或没有改写，只要按了空格键，就表示处理完成)，Debug 将接着显示下一个内存单元的原始内容，并提示进行修改，读者可以用同样的方法处理。

④ 所有希望改写的内存单元改写完毕后，按 Enter 键，E 命令操作结束。

可以用 E 命令向内存中写入字符，比如，用 E 命令从内存 1000:0 开始写入数值 1、字符“a”、数值 2、字符“b”、数值 3、字符“c”，可采用图 2.42 中所示的方法进行。

```
C:\>debug
-e 1000:0 1 'a' 2 'b' 3 'c'
-
-d 1000:0 f
1000:0000  01 61 02 62 03 63 00 00-00 00 00 00 00 00 00 00   .a.b.c..........
-
```

图 2.42　用 E 命令向内存中写入字符

从图 2.42 中可以看出，Debug 对 E 命令的执行结果是，向 1000:0、1000:2、1000:4 单元中写入数值 1、2、3，向 1000:1、1000:3、1000: 5 单元中写入字符“a”、“b”、“c”的 ASCII 码值：61H、62H、63H。

也可以用 E 命令向内存中写入字符串，比如，用 E 命令从内存 1000:0 开始写入：数值 1、字符串“a+b”、数值 2、字符串“c++”、字符 3、字符串“IBM”，如图 2.43 所示。

(7) 用 E 命令向内存中写入机器码，用 U 命令查看内存中机器码的含义，用 T 命令执行内存中的机器码。

```
C:\>debug
-e 1000:0 1 "a+b" 2 "c++" 3 "IBM"
-
-d 1000:0 f
1000:0000  01 61 2B 62 02 63 2B 2B-03 49 42 4D 00 00 00 00   .a+b.c++.IBM....
-
```

图 2.43　用 E 命令向内存中写入字符串

如何向内存中写入机器码呢？我们知道，机器码也是数据，当然可以用 E 命令将机器码写入内存。比如我们要从内存 1000:0 单元开始写入这样一段机器码：

机器码	对应的汇编指令
b80100	mov ax,0001
b90200	mov cx,0002
01c8	add ax,cx

可用如图 2.44 中所示的方法进行。

```
C:\>debug
-e 1000:0 b8 01 00 b9 02 00 01 c8
-
```

图 2.44　用 E 命令将机器码写入内存

如何查看写入的或内存中原有的机器码所对应的汇编指令呢？可以使用 U 命令。比如可以用 U 命令将从 1000:0 开始的内存单元中的内容翻译为汇编指令，并显示出来，如图 2.45 所示。

```
C:\>debug
-e 1000:0 b8 01 00 b9 02 00 01 c8
-
-d 1000:0 1f
1000:0000  B8 01 00 B9 02 00 01 C8-03 49 42 4D 00 00 00 00 .........IBM.
1000:0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ............
-
-u 1000:0
1000:0000 B80100        MOV     AX,0001
1000:0003 B90200        MOV     CX,0002
1000:0006 01C8          ADD     AX,CX
1000:0008 034942        ADD     CX,[BX+DI+42]
1000:000B 4D            DEC     BP
1000:000C 0000          ADD     [BX+SI],AL
1000:000E 0000          ADD     [BX+SI],AL
1000:0010 0000          ADD     [BX+SI],AL
1000:0012 0000          ADD     [BX+SI],AL
1000:0014 0000          ADD     [BX+SI],AL
1000:0016 0000          ADD     [BX+SI],AL
1000:0018 0000          ADD     [BX+SI],AL
1000:001A 0000          ADD     [BX+SI],AL
1000:001C 0000          ADD     [BX+SI],AL
1000:001E 0000          ADD     [BX+SI],AL
-
```

图 2.45　用 U 命令将内存单元中的内容翻译为汇编指令显示

图 2.45 中，首先用 E 命令向从 1000:0 开始的内存单元中写入了 8 个字节的机器码；然后用 D 命令查看内存 1000:0~1000:1f 中的数据(从数据的角度看一下写入的内容)；最后用 U 命令查看从 1000:0 开始的内存单元中的机器指令和它们所对应的汇编指令。

U 命令的显示输出分为 3 部分，每一条机器指令的地址、机器指令、机器指令所对应的汇编指令。我们可以看到：

1000:0 处存放的是写入的机器码 b8 01 00 所组成的机器指令，对应的汇编指令是 mov ax,1；

1000:3 处存放的是写入的机器码 b9 02 00 所组成的机器指令；对应的汇编指令是 mov cx,2；

1000:6 处存放的是写入的机器码 01 c8 所组成的机器指令；对应的汇编指令是 add ax,cx；

1000:8 处存放的是内存中的机器码 03 49 42 所组成的机器指令；对应的汇编指令是 add cx,[bx+di+42]。

由此，我们可以再一次看到内存中的数据和代码没有任何区别，关键在于如何解释。

如何执行我们写入的机器指令呢？使用 Debug 的 T 命令可以执行一条或多条指令，简单地使用 T 命令，可以执行 CS:IP 指向的指令，如图 2.46 所示。

```
-e 1000:0 b8 01 00 b9 02 00 01 c8
-
-
-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   NV UP EI PL NZ NA PO NC
0B39:0100 40             INC     AX
-
-rcs
CS 0B39
:1000
-
-rip
IP 0100
:0
-
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B39  ES=0B39  SS=0B39  CS=1000  IP=0000   NV UP EI PL NZ NA PO NC
1000:0000 B80100         MOV     AX,0001
-t

AX=0001  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B39  ES=0B39  SS=0B39  CS=1000  IP=0003   NV UP EI PL NZ NA PO NC
1000:0003 B90200         MOV     CX,0002
-
```

图 2.46　使用 T 命令执行 CS:IP 指向的指令

图 2.46 中，首先用 E 命令向从 1000:0 开始的内存单元中写入了 8 个字节的机器码；然后用 R 命令查看 CPU 中寄存器的状态，可以看到，CS=0b39H、IP=0100H，指向内存 0b39:0100；若要用 T 命令控制 CPU 执行我们写到 1000:0 的指令，必须先让 CS:IP 指向 1000:0；接着用 R 命令修改 CS、IP 中的内容，使 CS:IP 指向 1000:0。

完成上面的步骤后，就可以使用 T 命令来执行我们写入的指令了(此时，CS:IP 指向我们的指令所在的内存单元)。执行 T 命令后，CPU 执行 CS:IP 指向的指令，则 1000:0 处的指令 b8 01 00(mov ax,0001)得到执行，指令执行后，Debug 显示输出 CPU 中寄存器的状态。

注意，指令执行后，AX 中的内容被改写为 1，IP 改变为 IP+3(因为 mov ax,0001 的指令长度为 3 个字节)，CS:IP 指向下一条指令。

接着图 2.46，我们可以继续使用 T 命令执行下面的指令，如图 2.47 所示。

```
AX=0001  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B39  ES=0B39  SS=0B39  CS=1000  IP=0003   NV UP EI PL NZ NA PO NC
1000:0003 B90200        MOV     CX,0002
-t

AX=0001  BX=0000  CX=0002  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B39  ES=0B39  SS=0B39  CS=1000  IP=0006   NV UP EI PL NZ NA PO NC
1000:0006 01C8          ADD     AX,CX
-t

AX=0003  BX=0000  CX=0002  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B39  ES=0B39  SS=0B39  CS=1000  IP=0008   NV UP EI PL NZ NA PE NC
1000:0008 40            INC     AX
-
```

图 2.47　用 T 命令继续执行

在图 2.47 中，用 T 命令继续执行后面的指令，注意每条指令执行后，CPU 相关寄存器内容的变化。

(8)　用 Debug 的 A 命令以汇编指令的形式在内存中写入机器指令。

前面我们使用 E 命令写入机器指令，这样做很不方便，最好能直接以汇编指令的形式写入指令。为此，Debug 提供了 A 命令。 A 命令的使用方法如图 2.48 所示。

```
C:\>debug
-a 1000:0
1000:0000 mov ax,1
1000:0003 mov bx,2
1000:0006 mov cx,3
1000:0009 add ax,bx
1000:000B add ax,cx
1000:000D add ax,ax
1000:000F
-
-d 1000:0 f
1000:0000  B8 01 00 BB 02 00 B9 03-00 01 D8 01 C8 01 C0 00   ........
-
```

图 2.48　用 A 命令向从 1000:0 开始的内存单元中写入指令

图 2.48 中，首先用 A 命令，以汇编语言向从 1000:0 开始的内存单元中写入了几条指令，然后用 D 命令查看 A 命令的执行结果。可以看到，在使用 A 命令写入指令时，我们输入的是汇编指令，Debug 将这些汇编指令翻译为对应的机器指令，将它们的机器码写入内存。

使用 A 命令写入汇编指令时，在给出的起始地址后直接按 Enter 键表示操作结束。

如图 2.49 中，简单地用 A 命令，从一个预设的地址开始输入指令。

```
C:\>debug
-a
0B39:0100 mov ax,1
0B39:0103 mov bx,2
0B39:0106 mov cx,3
0B39:0109 add ax,bx
0B39:010B add ax,cx
0B39:010D add ax,ax
0B39:010F
-
```

图 2.49　从一个预设的地址开始输入指令

本次实验中需要用到的命令

查看、修改 CPU 中寄存器的内容：R 命令

查看内存中的内容：D 命令

修改内存中的内容：E 命令(可以写入数据、指令，在内存中，它们实际上没有区别)

将内存中的内容解释为机器指令和对应的汇编指令：U 命令

执行 CS:IP 指向的内存单元处的指令：T 命令

以汇编指令的形式向内存中写入指令：A 命令

在预备知识中，详细讲解了 Debug 的基本功能和用法。在汇编语言的学习中，Debug 是一个经常用到的工具，在学习预备知识中，应该一边看书一边在机器上操作。

前面提到，我们的原则是：以后用到的，以后再说。所以在这里只讲了一些在本次实验中需要用到的命令的相关的使用方法。以后根据需要，我们会讲解其他的用法。

2. 实验任务

(1) 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。

```
  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
```

提示，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。

(2) 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。

```
mov ax,1
```

```
add ax,ax
jmp 2000:0003
```

(3) 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。

提示，如果读者对实验的结果感到疑惑，请仔细阅读第 1 章中的 1.15 节。

(4) 向内存从 B8100H 开始的单元中填写数据，如：

```
-e B810:0000 01 01 02 02 03 03 04 04
```

请读者先填写不同的数据，观察产生的现象；再改变填写的地址，观察产生的现象。

提示，如果读者对实验的结果感到疑惑，请仔细阅读第 1 章中的 1.15 节。

第 3 章　寄存器(内存访问)

第 2 章中，我们主要从 CPU 如何执行指令的角度讲解了 8086CPU 的逻辑结构、形成物理地址的方法、相关的寄存器以及一些指令。读者应在通过了前一章所有的检测点，并完成了实验任务之后，再开始学习当前的课程。这一章中，我们从访问内存的角度继续学习几个寄存器。

3.1　内存中字的存储

CPU 中，用 16 位寄存器来存储一个字。高 8 位存放高位字节，低 8 位存放低位字节。在内存中存储时，由于内存单元是字节单元(一个单元存放一个字节)，则一个字要用两个地址连续的内存单元来存放，这个字的低位字节存放在低地址单元中，高位字节存放在高地址单元中。比如我们从 0 地址开始存放 20000，这种情况如图 3.1 所示。

0	20H
1	4EH
2	12H
3	00H
4	
5	

图 3.1　内存中字的存储

在图 3.1 中，我们用 0、1 两个内存单元存放数据 20000(4E20H)。0、1 两个内存单元用来存储一个字，这两个单元可以看作一个起始地址为 0 的字单元(存放一个字的内存单元，由 0、1 两个字节单元组成)。对于这个字单元来说，0 号单元是低地址单元，1 号单元是高地址单元，则字型数据 4E20H 的低位字节存放在 0 号单元中，高位字节存放在 1 号单元中。同理，将 2、3 号单元看作一个字单元，它的起始地址为 2。在这个字单元中存放数据 18(0012H)，则在 2 号单元中存放低位字节 12H，在 3 号单元中存放高位字节 00H。

我们提出字单元的概念：字单元，即存放一个字型数据(16 位)的内存单元，由两个地址连续的内存单元组成。高地址内存单元中存放字型数据的高位字节，低地址内存单元中存放字型数据的低位字节。

在以后的课程中，我们将起始地址为 N 的字单元简称为 N 地址字单元。比如一个字单元由 2、3 两个内存单元组成，则这个字单元的起始地址为 2，我们可以说这是 2 地址字单元。

问题 3.1

对于图 3.1：

(1)　0 地址单元中存放的字节型数据是多少？
(2)　0 地址字单元中存放的字型数据是多少？

(3) 2 地址单元中存放的字节型数据是多少？
(4) 2 地址字单元中存放的字型数据是多少？
(5) 1 地址字单元中存放的字型数据是多少？

思考后看分析。

分析：

(1) 0 地址单元中存放的字节型数据：20H；
(2) 0 地址字单元中存放的字型数据：4E20H；
(3) 2 地址单元中存放的字节型数据：12H；
(4) 2 地址字单元中存放的字型数据：0012H；
(5) 1 地址字单元，即起始地址为 1 的字单元，它由 1 号单元和 2 号单元组成，用这两个单元存储一个字型数据，高位放在 2 号单元中，即：12H，低位放在 1 号单元中，即：4EH，它们组成字型数据是 124EH，大小为：4686。

从上面的问题中我们看到，任何两个地址连续的内存单元，N 号单元和 N+1 号单元，可以将它们看成两个内存单元，也可看成一个地址为 N 的字单元中的高位字节单元和低位字节单元。

3.2 DS 和[address]

CPU 要读写一个内存单元的时候，必须先给出这个内存单元的地址，在 8086PC 中，内存地址由段地址和偏移地址组成。8086CPU 中有一个 DS 寄存器，通常用来存放要访问数据的段地址。比如我们要读取 10000H 单元的内容，可以用如下的程序段进行。

```
mov bx,1000H
mov ds,bx
mov al,[0]
```

上面的 3 条指令将 10000H(1000:0)中的数据读到 al 中。

下面详细说明指令的含义。

```
mov al,[0]
```

前面我们使用 mov 指令，可完成两种传送：①将数据直接送入寄存器；②将一个寄存器中的内容送入另一个寄存器。

也可以使用 mov 指令将一个内存单元中的内容送入一个寄存器中。从哪一个内存单元送到哪一个寄存器中呢？在指令中必须指明。寄存器用寄存器名来指明，内存单元则需用内存单元的地址来指明。显然，此时 mov 指令的格式应该是：mov 寄存器名，内存单元地址。

“[…]”表示一个内存单元，“[…]”中的 0 表示内存单元的偏移地址。我们知道，

只有偏移地址是不能定位一个内存单元的，那么内存单元的段地址是多少呢？指令执行时，8086CPU 自动取 ds 中的数据为内存单元的段地址。

再来看一下，如何用 mov 指令从 10000H 中读取数据。10000H 用段地址和偏移地址表示为 1000:0，我们先将段地址 1000H 放入 ds，然后用 mov al,[0]完成传送。mov 指令中的[]说明操作对象是一个内存单元，[]中的 0 说明这个内存单元的偏移地址是 0，它的段地址默认放在 ds 中，指令执行时，8086CPU 会自动从 ds 中取出。

```
mov bx,1000H
mov ds,bx
```

若要用 mov al,[0]完成数据从 1000:0 单元到 al 的传送，这条指令执行时，ds 中的内容应为段地址 1000H，所以在这条指令之前应该将 1000H 送入 ds。

如何把一个数据送入寄存器呢？我们以前用类似“mov ax,1”这样的指令来完成，从理论上讲，我们可以用相似的方式：mov ds,1000H，来将 1000H 送入 ds。可是，现实并非如此，8086CPU 不支持将数据直接送入段寄存器的操作，ds 是一个段寄存器，所以 mov ds,1000H 这条指令是非法的。那么如何将 1000H 送入 ds 呢？只好用一个寄存器来进行中转，即先将 1000H 送入一个一般的寄存器，如 bx，再将 bx 中的内容送入 ds。

为什么 8086CPU 不支持将数据直接送入段寄存器的操作？这属于 8086CPU 硬件设计的问题，我们只要知道这一点就行了。

问题 3.2

写几条指令，将 al 中的数据送入内存单元 10000H 中，思考后看分析。

分析：

怎样将数据从寄存器送入内存单元？从内存单元到寄存器的格式是：“mov 寄存器名,内存单元地址”，从寄存器到内存单元则是：“mov 内存单元地址,寄存器名”。10000H 可表示为 1000:0，用 ds 存放段地址 1000H，偏移地址是 0，则 mov [0],al 可完成从 al 到 10000H 的数据传送。完整的几条指令是：

```
mov bx,1000H
mov ds,bx
mov [0],al
```

3.3　字 的 传 送

前面我们用 mov 指令在寄存器和内存之间进行字节型数据的传送。因为 8086CPU 是 16 位结构，有 16 根数据线，所以，可以一次性传送 16 位的数据，也就是说可以一次性传送一个字。只要在 mov 指令中给出 16 位的寄存器就可以进行 16 位数据的传送了。比如：

```
mov bx,1000H
mov ds,bx
mov ax,[0]        ;1000:0 处的字型数据送入 ax
mov [0],cx        ;cx 中的 16 位数据送到 1000:0 处
```

问题 3.3

内存中的情况如图 3.2 所示，写出下面的指令执行后寄存器 ax,bx,cx 中的值。

```
mov ax,1000H
mov ds,ax
mov ax,[0]
mov bx,[2]
mov cx,[1]
add bx,[1]
add cx,[2]
```

10000H	23
10001H	11
10002H	22
10003H	66

图 3.2　内存情况示意(1)

思考后看分析。

分析：

进行单步跟踪，看一下每条指令执行后相关寄存器中的值，见表 3.1。

表 3.1　指令执行与寄存器中的内容(1)

指　令	执行后相关寄存器中的内容	说　明
mov ax,1000H	ax=1000H	前两条指令的目的是将 ds 设为 1000H
mov ds,ax	ds=1000H	
mov ax,[0]	ax=1123H	1000:0 处存放的字型数据送入 ax： 1000:1 单元存放字型数据的高 8 位：11H， 1000:0 单元存放字型数据的低 8 位：23H， 所以 1000:0 处存放的字型数据为 1123H。 指令执行时，字型数据的高 8 位送入 ah，字型数据的低 8 位送入 al，则 ax 中的数据为 1123H
mov bx,[2]	bx=6622H	原理同上
mov cx,[1]	cx=2211H	
add bx,[1]	bx=8833H	
add cx,[2]	cx=8833H	

问题 3.4

内存中的情况如图 3.3 所示，写出下面的指令执行后内存中的值，思考后看分析。

```
mov ax,1000H
mov ds,ax
mov ax,11316
mov [0],ax
mov bx,[0]
sub bx,[2]
mov [2],bx
```

10000H	23
10001H	11
10002H	22
10003H	11

图 3.3　内存情况示意(2)

分析：

进行单步跟踪，看一下每条指令执行后相关寄存器或内存单元中的值，见表 3.2。

表 3.2　指令执行与寄存器中的内容(2)

指　令	执行后相关寄存器或内存单元中的内容	说　明
mov ax,1000H	ax=1000H	前两条指令的目的是将 ds 设为 1000H
mov ds,ax	ds=1000H	
mov ax,11316	ax=2C34H	十进制 11316，十六进制 2C34H
mov [0],ax	10000H 34 10001H 2C 10002H 22 10003H 11	ax 中的字型数据送到 1000:0 处： ax 中的字型数据是 2C34H， 高 8 位：2CH，在 ah 中， 低 8 位：34H，在 al 中， 指令执行时，高 8 位送入高地址 1000:1 单元，低 8 位送入低地址 1000:0 单元
mov bx,[0]	bx=2C34H	bx=bx 中的字型数据-1000:2 处的字型数据=2C34H-1122H=1B12H
sub bx,[2]	bx=1B12H	
mov [2],bx	10000H 34 10001H 2C 10002H 12 10003H 1B	bx 中的字型数据送到 1000:2 处

3.4　mov、add、sub 指令

前面我们用到了 mov、add、sub 指令，它们都带有两个操作对象。

到现在，我们知道，mov 指令可以有以下几种形式。

mov　寄存器，数据　　比如：mov ax,8
mov　寄存器，寄存器　　比如：mov ax,bx
mov　寄存器，内存单元　　比如：mov ax,[0]
mov　内存单元，寄存器　　比如：mov [0],ax
mov　段寄存器，寄存器　　比如：mov ds,ax

我们可以根据这些已知指令进行下面的推测。

(1) 既然有“mov 段寄存器，寄存器”，从寄存器向段寄存器传送数据，那么也应该有“mov 寄存器，段寄存器”，从段寄存器向寄存器传送数据。一个合理的设想是：8086CPU 内部有寄存器到段寄存器的通路，那么也应该有相反的通路。

有了推测，我们还要验证一下。进入 Debug，用 A 命令，如图 3.4 所示。

```
C:\>debug
-a
0B39:0100 mov ax,ds
0B39:0102
-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   NV UP EI PL NZ NA PO NC
0B39:0100 8CD8          MOV     AX,DS
-t

AX=0B39  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B39  ES=0B39  SS=0B39  CS=0B39  IP=0102   NV UP EI PL NZ NA PO NC
0B39:0102 CE            INTO
-
```

图 3.4 试验 mov ax,ds

图 3.4 中，用 A 命令在一个预设的地址 0B39:0100 处，用汇编的形式 mov ax,ds 写入指令，再用 T 命令执行，可以看到执行的结果，段寄存器 ds 中的值送到了寄存器 ax 中。通过验证我们知道，“mov 寄存器，段寄存器”是正确的指令。

(2) 既然有“mov 内存单元，寄存器”，从寄存器向内存单元传送数据，那么也应该有“mov 内存单元，段寄存器”，从段寄存器向内存单元传送数据。比如我们可以将段寄存器 cs 中的内容送入内存 10000H 处，指令如下。

```
mov ax,1000H
mov ds,ax
mov [0],cs
```

在 Debug 中进行试验，如图 3.5 所示。

```
C:\>debug
-a
0B39:0100 mov ax,1000
0B39:0103 mov ds,ax
0B39:0105 mov [0],cs
0B39:0109
-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   NV UP EI PL NZ NA PO NC
0B39:0100 B80010        MOV     AX,1000
-t

AX=1000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B39  ES=0B39  SS=0B39  CS=0B39  IP=0103   NV UP EI PL NZ NA PO NC
0B39:0103 8ED8          MOV     DS,AX
-t

AX=1000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1000  ES=0B39  SS=0B39  CS=0B39  IP=0105   NV UP EI PL NZ NA PO NC
0B39:0105 8C0E0000      MOV     [0000],CS                       DS:0000=0000
-t

AX=1000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1000  ES=0B39  SS=0B39  CS=0B39  IP=0109   NV UP EI PL NZ NA PO NC
0B39:0109 06            PUSH    ES
-d 1000:0
1000:0000  39 0B 00 00 00 00 00 00-00 00 00 00 00 00 00 00   9................
1000:0010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1000:0020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1000:0030  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1000:0040  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1000:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1000:0060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1000:0070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-
```

图 3.5 试验 mov [0],cs

图 3.5 中，当 CS:IP 指向 0B39:0105 的时候，Debug 显示当前的指令 mov [0000],cs，因为这是一条访问内存的指令，Debug 还显示出指令要访问的内存单元中的内容。由于指令中的 CS 是一个 16 位寄存器，所以要访问(写入)的内存单元是一个字单元，它的偏移地址为 0，段地址在 ds 中，Debug 在屏幕右边显示出“DS:0000=0000”，我们可以知道这个字单元中的内容为 0。

mov [0000],cs 执行后，CS 中的数据(0B39H)被写入 1000:0 处，1000:1 单元存放 0BH，1000:0 单元存放 39H。

最后，用 D 命令从 1000:0 开始查看指令执行后内存中的情况，注意 1000:0、1000:1 两个单元的内容。

(3) “mov 段寄存器，内存单元”也应该可行，比如我们可以用 10000H 处存放的字型数据设置 ds(即将 10000H 处存放的字型数据送入 ds)，指令如下。

```
mov ax,1000H
mov ds,ax
mov ds,[0]
```

可以自行在 Debug 中进行试验。

add 和 sub 指令同 mov 一样，都有两个操作对象。它们也可以有以下几种形式。

add	寄存器，数据	比如：add ax,8
add	寄存器，寄存器	比如：add ax,bx
add	寄存器，内存单元	比如：add ax,[0]
add	内存单元，寄存器	比如：add [0],ax
sub	寄存器，数据	比如：sub ax,9
sub	寄存器，寄存器	比如：sub ax,bx
sub	寄存器，内存单元	比如：sub ax,[0]
sub	内存单元，寄存器	比如：sub [0],ax

它们可以对段寄存器进行操作吗？比如“add ds,ax”。请自行在 Debug 中试验。

3.5 数　据　段

前面讲过(参见 2.8 节)，对于 8086PC 机，在编程时，可以根据需要，将一组内存单元定义为一个段。我们可以将一组长度为 N(N≤64KB)、地址连续、起始地址为 16 的倍数的内存单元当作专门存储数据的内存空间，从而定义了一个数据段。比如用 123B0H~123B9H 这段内存空间来存放数据，我们就可以认为，123B0H~123B9H 这段内存是一个数据段，它的段地址为 123BH，长度为 10 个字节。

如何访问数据段中的数据呢？将一段内存当作数据段，是我们在编程时的一种安排，

可以在具体操作的时候，用 ds 存放数据段的段地址，再根据需要，用相关指令访问数据段中的具体单元。

比如，将 123B0H~123B9H 的内存单元定义为数据段。现在要累加这个数据段中的前 3 个单元中的数据，代码如下。

```
mov ax,123BH
mov ds,ax           ;将 123BH 送入 ds 中，作为数据段的段地址
mov al,0            ;用 al 存放累加结果
add al,[0]          ;将数据段第一个单元(偏移地址为 0)中的数值加到 al 中
add al,[1]          ;将数据段第二个单元(偏移地址为 1)中的数值加到 al 中
add al,[2]          ;将数据段第三个单元(偏移地址为 2)中的数值加到 al 中
```

问题 3.5

写几条指令，累加数据段中的前 3 个字型数据，思考后看分析。

分析：

代码如下。

```
mov ax,123BH
mov ds,ax           ;将 123BH 送入 ds 中，作为数据段的段地址
mov ax,0            ;用 ax 存放累加结果
add ax,[0]          ;将数据段第一个字(偏移地址为 0)加到 ax 中
add ax,[2]          ;将数据段第二个字(偏移地址为 2)加到 ax 中
add ax,[4]          ;将数据段第三个字(偏移地址为 4)加到 ax 中
```

注意，一个字型数据占两个单元，所以偏移地址是 0、2、4。

3.1~3.5 小　　结

(1) 字在内存中存储时，要用两个地址连续的内存单元来存放，字的低位字节存放在低地址单元中，高位字节存放在高地址单元中。

(2) 用 mov 指令访问内存单元，可以在 mov 指令中只给出单元的偏移地址，此时，段地址默认在 DS 寄存器中。

(3) [address]表示一个偏移地址为 address 的内存单元。

(4) 在内存和寄存器之间传送字型数据时，高地址单元和高 8 位寄存器、低地址单元和低 8 位寄存器相对应。

(5) mov、add、sub 是具有两个操作对象的指令。jmp 是具有一个操作对象的指令。

(6) 可以根据自己的推测，在 Debug 中实验指令的新格式。

检测点 3.1

(1) 在 Debug 中，用“d 0:0 1f”查看内存，结果如下。

```
0000:0000 70 80 F0 30 EF 60 30 E2-00 80 80 12 66 20 22 60
0000:0010 62 26 E6 D6 CC 2E 3C 3B-AB BA 00 00 26 06 66 88
```

下面的程序执行前，AX=0，BX=0，写出每条汇编指令执行完后相关寄存器中的值。

```
mov ax,1
mov ds,ax
mov ax,[0000]      AX=_____
mov bx,[0001]      BX=_____
mov ax,bx          AX=_____
mov ax,[0000]      AX=_____
mov bx,[0002]      BX=_____
add ax,bx          AX=_____
add ax,[0004]      AX=_____
mov ax,0           AX=_____
mov al,[0002]      AX=_____
mov bx,0           BX=_____
mov bl,[000C]      BX=_____
add al,bl          AX=_____
```

提示，注意 ds 的设置。

(2) 内存中的情况如图 3.6 所示。

各寄存器的初始值：CS=2000H，IP=0，DS=1000H，AX=0，BX=0;

① 写出 CPU 执行的指令序列(用汇编指令写出)。

② 写出 CPU 执行每条指令后，CS、IP 和相关寄存器中的数值。

③ 再次体会：数据和程序有区别吗？如何确定内存中的信息哪些是数据，哪些是程序？

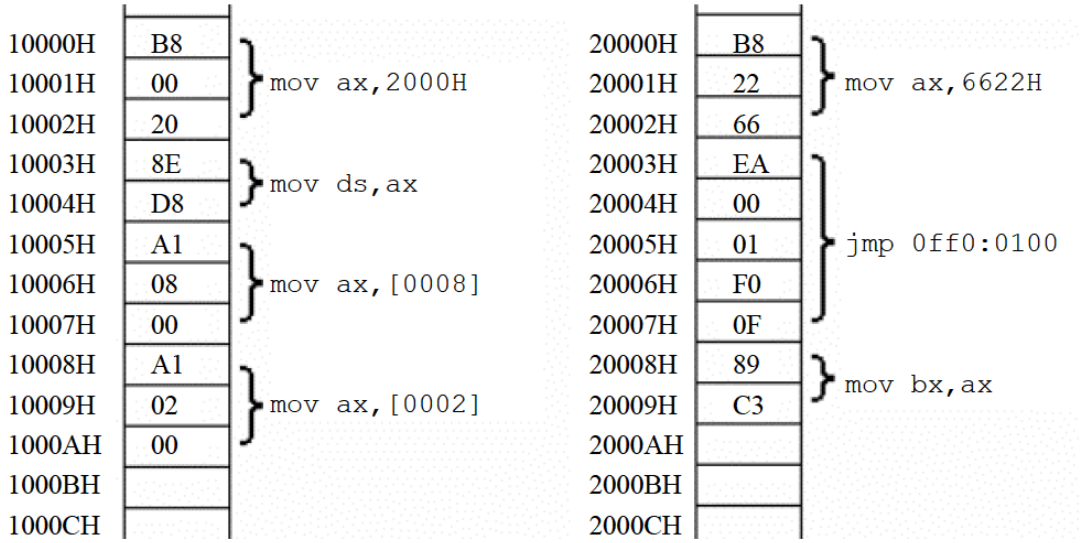


图 3.6 内存情况示意

3.6 栈

在这里，我们对栈的研究仅限于这个角度：栈是一种具有特殊的访问方式的存储空间。它的特殊性就在于，最后进入这个空间的数据，最先出去。

可以用一个盒子和 3 本书来描述栈的这种操作方式。

一个开口的盒子就可以看成一个栈空间，现在有 3 本书，《高等数学》、《C 语言》、《软件工程》，把它们放到盒子中，操作的过程如图 3.7 所示。

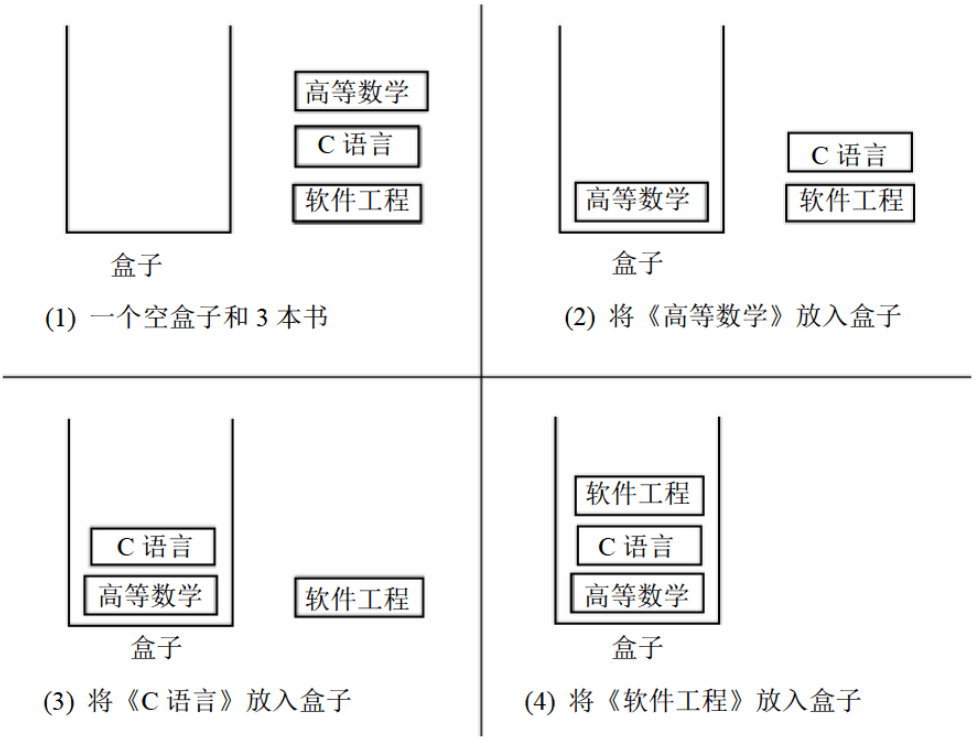


图 3.7 入栈的方式

现在的问题是，一次只允许取一本，我们如何将 3 本书从盒子中取出来？

显然，必须从盒子的最上边取。这样取出的顺序就是：《软件工程》、《C 语言》、《高等数学》，和放入的顺序相反，如图 3.8 所示。

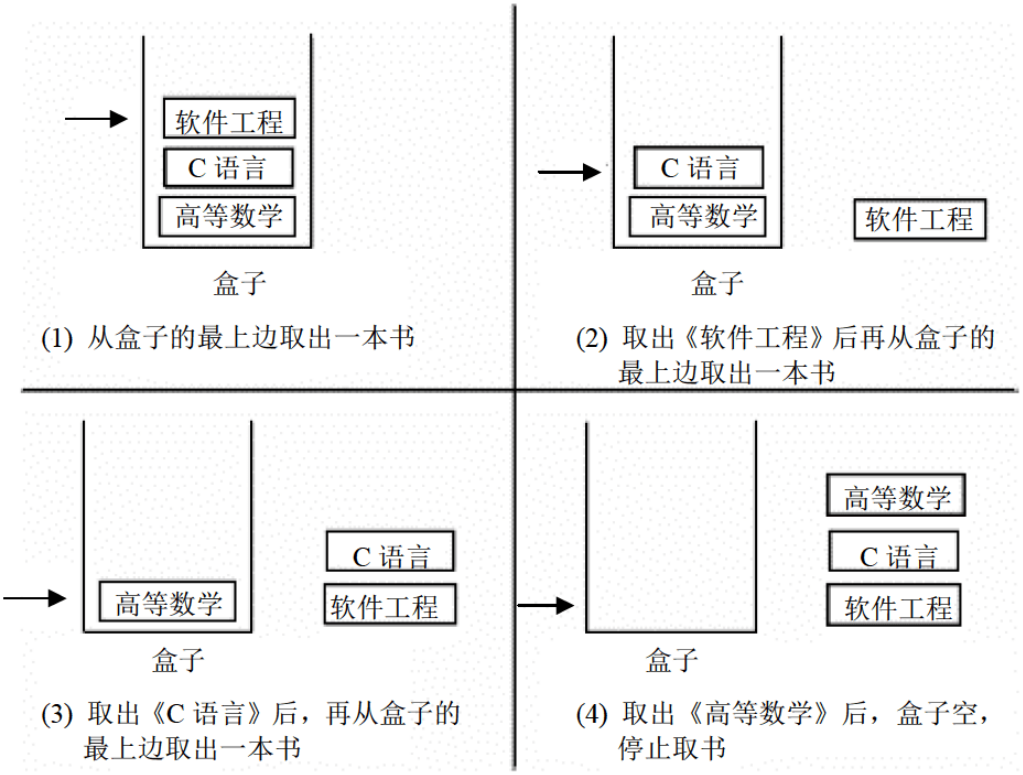


图 3.8　出栈的方式

从程序化的角度来讲，应该有一个标记，这个标记一直指示着盒子最上边的书。

如果说，上例中的盒子就是一个栈，我们可以看出，栈有两个基本的操作：入栈和出栈。入栈就是将一个新的元素放到栈顶，出栈就是从栈顶取出一个元素。栈顶的元素总是最后入栈，需要出栈时，又最先被从栈中取出。栈的这种操作规则被称为：LIFO(Last In First Out，后进先出)。

3.7　CPU 提供的栈机制

现今的 CPU 中都有栈的设计，8086CPU 也不例外。8086CPU 提供相关的指令来以栈的方式访问内存空间。这意味着，在基于 8086CPU 编程的时候，可以将一段内存当作栈来使用。

8086CPU 提供入栈和出栈指令，最基本的两个是 PUSH(入栈) 和 POP(出栈)。比如，push ax 表示将寄存器 ax 中的数据送入栈中，pop ax 表示从栈顶取出数据送入 ax。8086CPU 的入栈和出栈操作都是以字为单位进行的。

下面举例说明，我们可以将 10000H~1000FH 这段内存当作栈来使用。

图 3.9 描述了下面一段指令的执行过程。

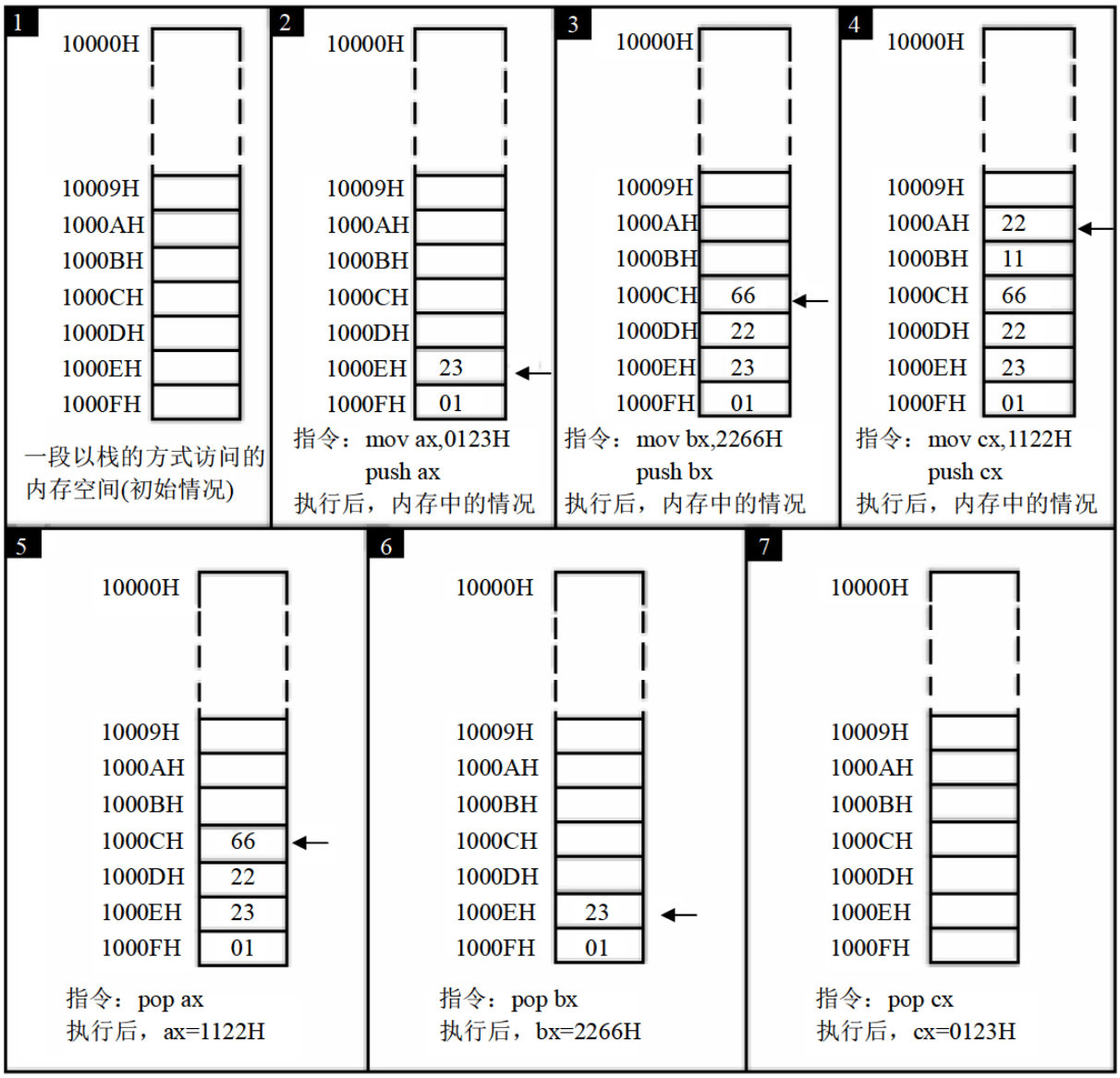


图 3.9　8086CPU 的栈操作

```
mov ax,0123H
push ax
mov bx,2266H
push bx
mov cx,1122H
push cx
pop ax
pop bx
pop cx
```

注意，字型数据用两个单元存放，高地址单元存放高 8 位，低地址单元存放低 8 位。

读者看到图 3.9 所描述的 push 和 pop 指令的执行过程，是否有一些疑惑？总结一下，大概是这两个问题。

其一，我们将 10000H~1000FH 这段内存当作栈来使用，CPU 执行 push 和 pop 指令时，将对这段空间按照栈的后进先出的规则进行访问。但是，一个重要的问题是，CPU 如何知道 10000H~1000FH 这段空间被当作栈来使用？

其二，push ax 等入栈指令执行时，要将寄存器中的内容放入当前栈顶单元的上方，成为新的栈顶元素；pop ax 等指令执行时，要从栈顶单元中取出数据，送入寄存器中。显然，push、pop 在执行的时候，必须知道哪个单元是栈顶单元，可是，如何知道呢？

这不禁让我们想起另外一个讨论过的问题，就是，CPU 如何知道当前要执行的指令所在的位置？我们现在知道答案，那就是 CS、IP 中存放着当前指令的段地址和偏移地址。现在的问题是：CPU 如何知道栈顶的位置？显然，也应该有相应的寄存器来存放栈顶的地址，8086CPU 中，有两个寄存器，段寄存器 SS 和寄存器 SP，栈顶的段地址存放在 SS 中，偏移地址存放在 SP 中。**任意时刻，SS:SP 指向栈顶元素。**push 指令和 pop 指令执行时，CPU 从 SS 和 SP 中得到栈顶的地址。

现在，我们可以完整地描述 push 和 pop 指令的功能了，例如 push ax。

push ax 的执行，由以下两步完成。

(1) SP=SP−2，SS:SP 指向当前栈顶前面的单元，以当前栈顶前面的单元为新的栈顶；
(2) 将 ax 中的内容送入 SS:SP 指向的内存单元处，SS:SP 此时指向新栈顶。

图 3.10 描述了 8086CPU 对 push 指令的执行过程。

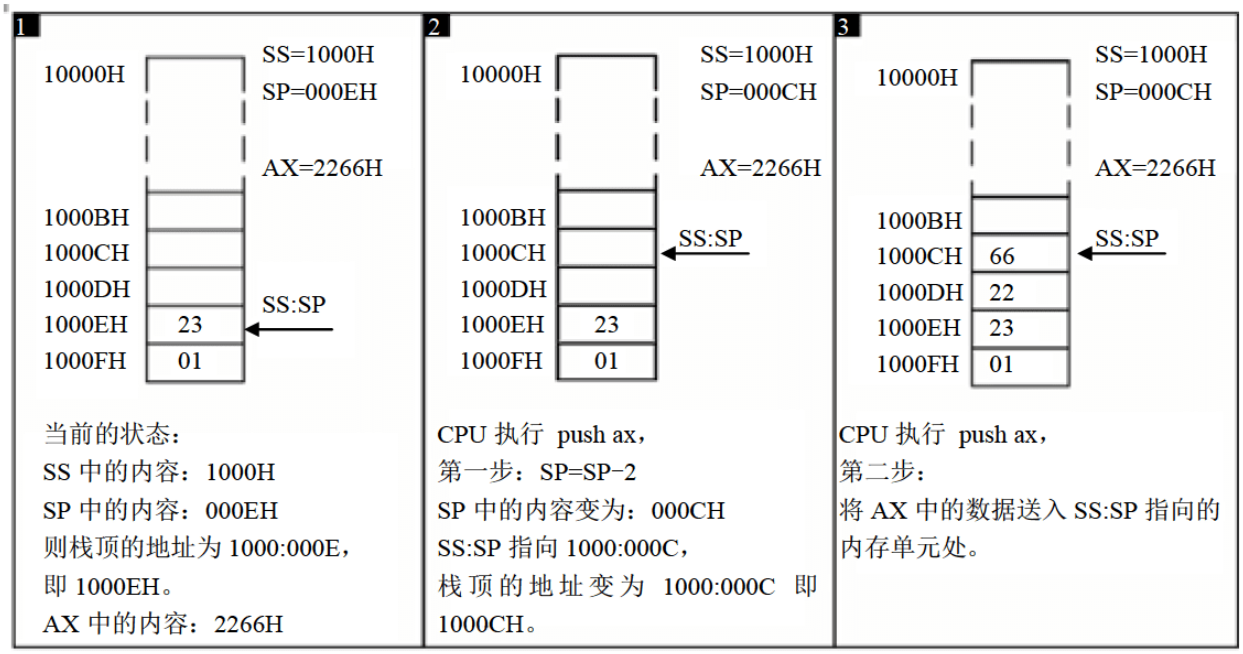


图 3.10　push 指令的执行过程

从图中我们可以看出，8086CPU 中，入栈时，栈顶从高地址向低地址方向增长。

问题 3.6

如果将 10000H~1000FH 这段空间当作栈，初始状态栈是空的，此时，SS=1000H，SP=？思考后看分析。

分析：

SP=0010H，如图 3.11 所示。

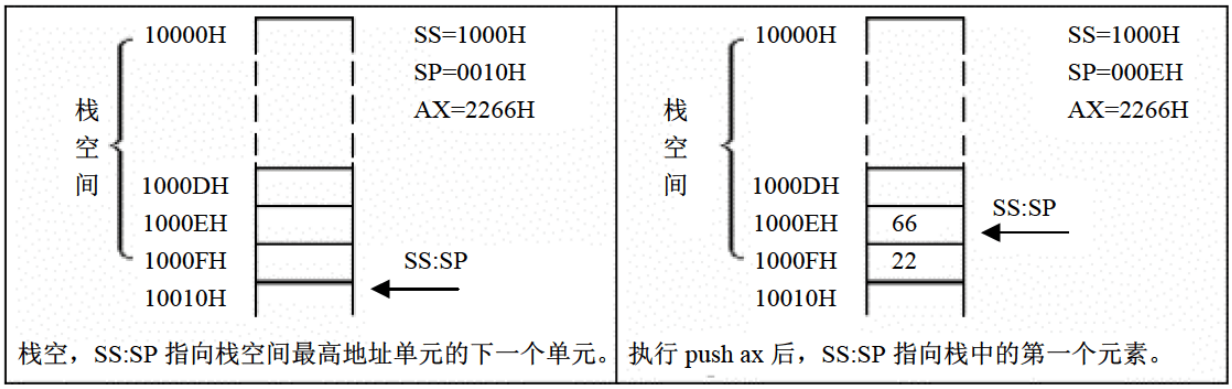


图 3.11 栈空的状态

将 10000H~1000FH 这段空间当作栈段，SS=1000H，栈空间大小为 16 字节，栈最底部的字单元地址为 1000:000E。任意时刻，SS:SP 指向栈顶，当栈中只有一个元素的时候，SS=1000H，SP=000EH。栈为空，就相当于栈中唯一的元素出栈，出栈后，SP=SP+2，SP 原来为 000EH，加 2 后 SP=10H，所以，当栈为空的时候，SS=1000H，SP=10H。

换一个角度看，任意时刻，SS:SP 指向栈顶元素，当栈为空的时候，栈中没有元素，也就不存在栈顶元素，所以 SS:SP 只能指向栈的最底部单元下面的单元，该单元的偏移地址为栈最底部的字单元的偏移地址+2，栈最底部字单元的地址为 1000:000E，所以栈空时，SP=0010H。

接下来，我们描述 pop 指令的功能，例如 pop ax。

pop ax 的执行过程和 push ax 刚好相反，由以下两步完成。

(1) 将 SS:SP 指向的内存单元处的数据送入 ax 中；

(2) SP=SP+2，SS:SP 指向当前栈顶下面的单元，以当前栈顶下面的单元为新的栈顶。

图 3.12 描述了 8086CPU 对 pop 指令的执行过程。

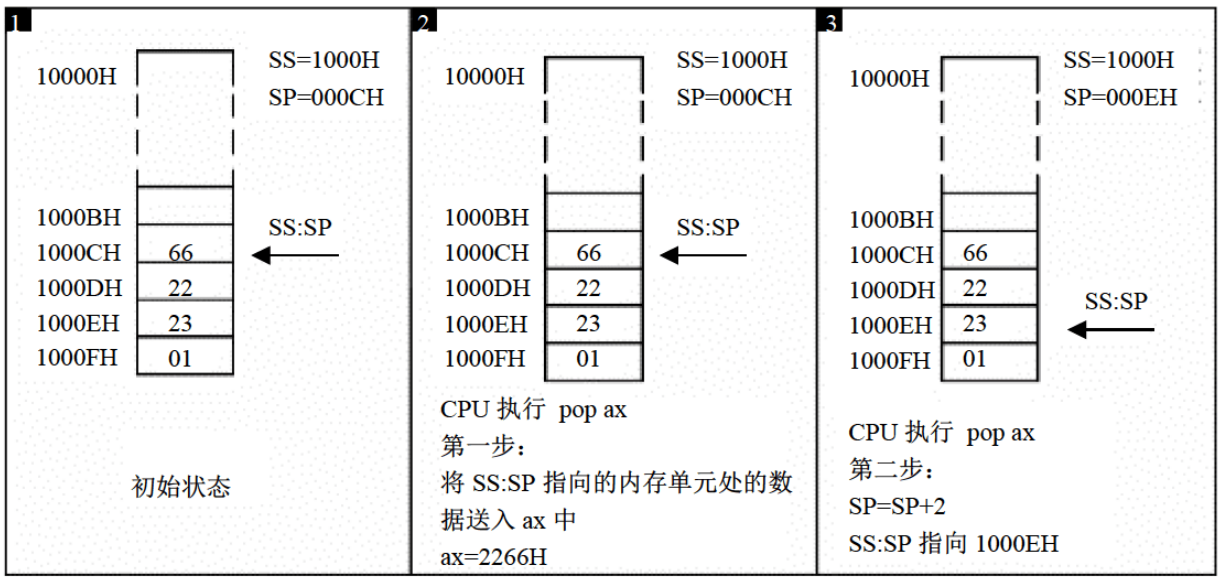


图 3.12　pop 指令的执行过程

注意，图 3.12 中，出栈后，SS:SP 指向新的栈顶 1000EH，pop 操作前的栈顶元素，1000CH 处的 2266H 依然存在，但是，它已不在栈中。当再次执行 push 等入栈指令后，SS:SP 移至 1000CH，并在里面写入新的数据，它将被覆盖。

3.8　栈顶超界的问题

我们现在知道，8086CPU 用 SS 和 SP 指示栈顶的地址，并提供 push 和 pop 指令实现入栈和出栈。

但是，还有一个问题需要讨论，就是 SS 和 SP 只是记录了栈顶的地址，依靠 SS 和 SP 可以保证在入栈和出栈时找到栈顶。可是，如何能够保证在入栈、出栈时，栈顶不会超出栈空间？

图 3.13 描述了在执行 push 指令后，栈顶超出栈空间的情况。

图 3.13 中，将 10010H~1001FH 当作栈空间，该栈空间容量为 16 字节(8 字)，初始状态为空，SS=1000H、SP=0020H，SS:SP 指向 10020H；

在执行 8 次 push ax 后，向栈中压入 8 个字，栈满，SS:SP 指向 10010H；

再次执行 push ax：sp=sp-2，SS:SP 指向 1000EH，栈顶超出了栈空间，ax 中的数据送入 1000EH 单元处，将栈空间外的数据覆盖。

图 3.14 描述了在执行 pop 指令后，栈顶超出栈空间的情况。

图 3.14 中，将 10010H~1001FH 当作栈空间，该栈空间容量为 16 字节(8 字)，当前状态为满，SS=1000H、SP=0010H，SS:SP 指向 10010H；

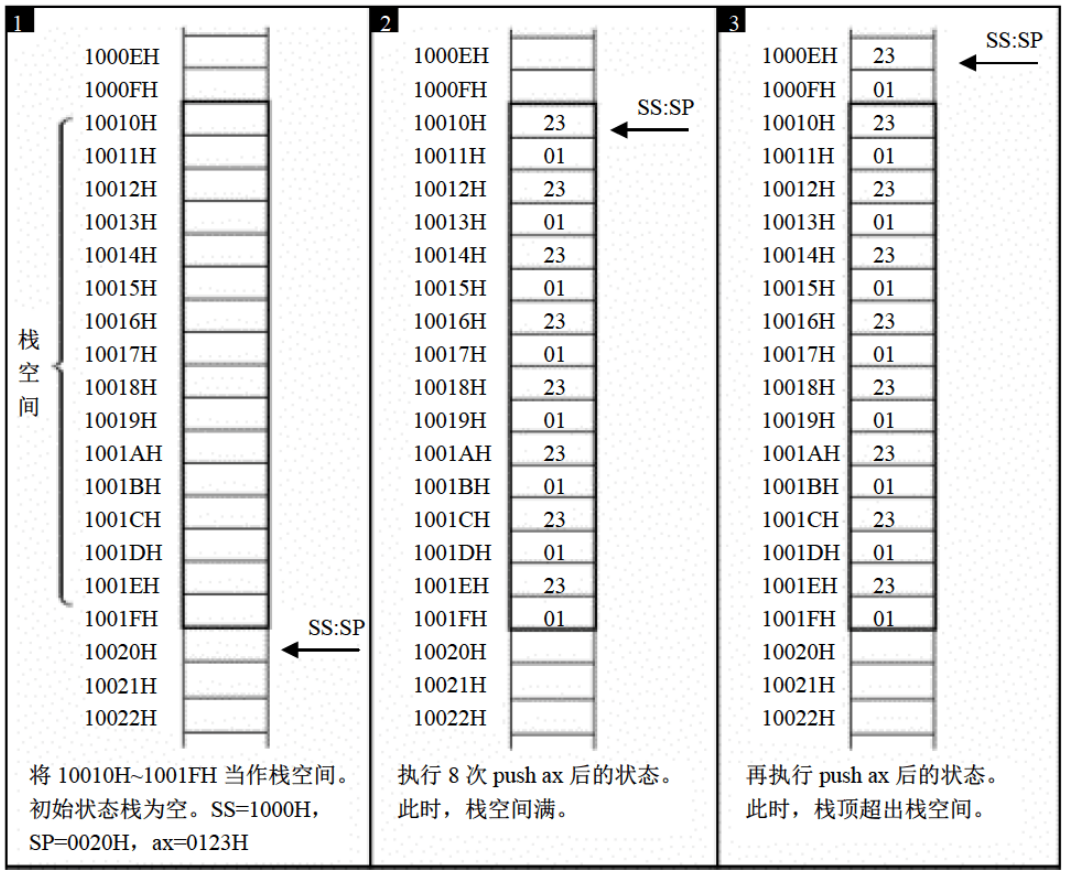


图 3.13　执行 push 后栈顶超出栈空间

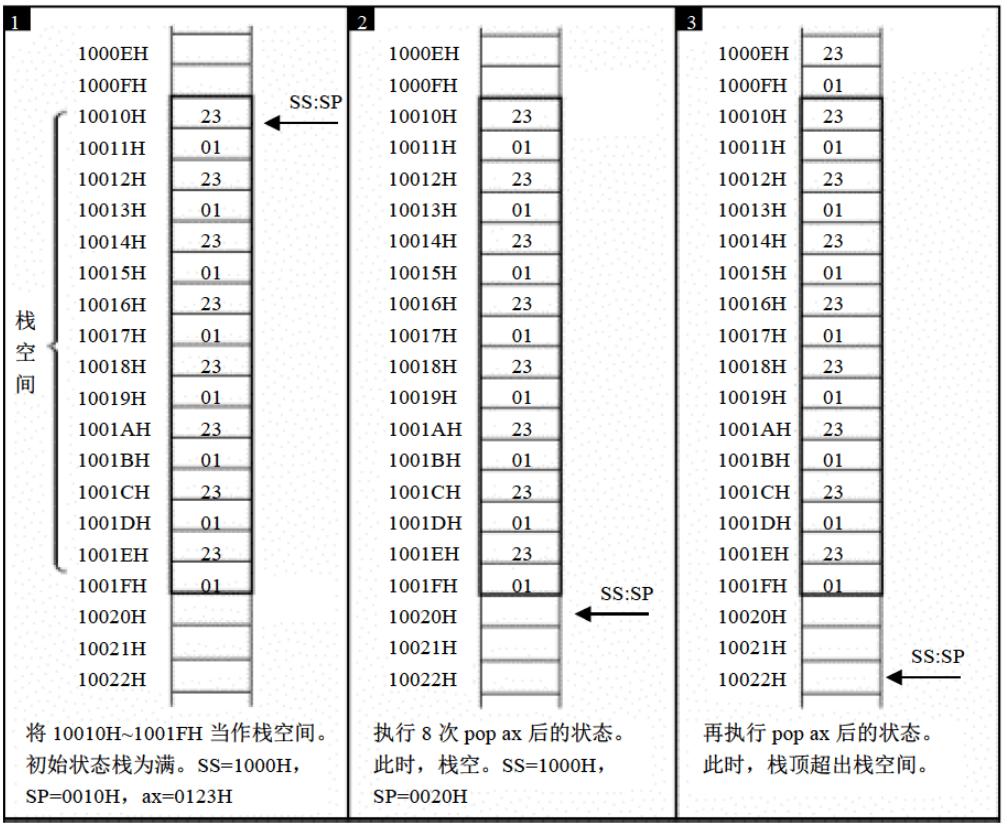


图 3.14　执行 pop 后栈顶超出栈空间

在执行 8 次 pop ax 后，从栈中弹出 8 个字，栈空，SS:SP 指向 10020H；

再次执行 pop ax：sp=sp+2，SS:SP 指向 10022H，栈顶超出了栈空间。此后，如果再执行 push 指令，10020H、10021H 中的数据将被覆盖。

上面描述了执行 push、pop 指令时，发生的栈顶超界问题。可以看到，当栈满的时候再使用 push 指令入栈，或栈空的时候再使用 pop 指令出栈，都将发生栈顶超界问题。

栈顶超界是危险的，因为我们既然将一段空间安排为栈，那么在栈空间之外的空间里很可能存放了具有其他用途的数据、代码等，这些数据、代码可能是我们自己程序中的，也可能是别的程序中的(毕竟一个计算机系统中并不是只有我们自己的程序在运行)。但是由于我们在入栈出栈时的不小心，而将这些数据、代码意外地改写，将会引发一连串的错误。

我们当然希望 CPU 可以帮我们解决这个问题，比如说在 CPU 中有记录栈顶上限和栈底的寄存器，我们可以通过填写这些寄存器来指定栈空间的范围，然后，CPU 在执行 push 指令的时候靠检测栈顶上限寄存器、在执行 pop 指令的时候靠检测栈底寄存器保证不会超界。

不过，对于 8086CPU，这只是我们的一个设想(我们当然可以这样设想，如果 CPU 是我们设计的话，这也就不仅仅是一个设想)。实际的情况是，8086CPU 中并没有这样的寄存器。

8086CPU 不保证我们对栈的操作不会超界。这也就是说，8086CPU 只知道栈顶在何处(由 SS:SP 指示)，而不知道我们安排的栈空间有多大。这点就好像 CPU 只知道当前要执行的指令在何处(由 CS:IP 指示)，而不知道要执行的指令有多少。从这两点上我们可以看出 8086CPU 的工作机理，它只考虑当前的情况：当前的栈顶在何处、当前要执行的指令是哪一条。

我们在编程的时候要自己操心栈顶超界的问题，要根据可能用到的最大栈空间，来安排栈的大小，防止入栈的数据太多而导致的超界；执行出栈操作的时候也要注意，以防栈空的时候继续出栈而导致的超界。

3.9　push、pop 指令

前面我们一直在使用 push ax 和 pop ax，显然 push 和 pop 指令是可以在寄存器和内存(栈空间当然也是内存空间的一部分，它只是一段可以以一种特殊的方式进行访问的内存空间。)之间传送数据的。

push 和 pop 指令的格式可以是如下形式：

```
push 寄存器        ;将一个寄存器中的数据入栈
pop 寄存器         ;出栈，用一个寄存器接收出栈的数据
```

当然也可以是如下形式：

```
push 段寄存器        ;将一个段寄存器中的数据入栈
pop 段寄存器         ;出栈，用一个段寄存器接收出栈的数据
```

push 和 pop 也可以在内存单元和内存单元之间传送数据，我们可以：

```
push 内存单元        ;将一个内存字单元处的字入栈(注意：栈操作都是以字为单位)
pop 内存单元         ;出栈，用一个内存字单元接收出栈的数据
```

比如：

```
mov ax,1000H
mov ds,ax            ;内存单元的段地址要放在 ds 中
push [0]             ;将 1000:0 处的字压入栈中
pop [2]              ;出栈，出栈的数据送入 1000:2 处
```

指令执行时，CPU 要知道内存单元的地址，可以在 push、pop 指令中只给出内存单元的偏移地址，段地址在指令执行时，CPU 从 ds 中取得。

问题 3.7

编程，将 10000H~1000FH 这段空间当作栈，初始状态栈是空的，将 AX、BX、DS 中的数据入栈。

思考后看分析。

分析：

代码如下。

```
mov ax,1000H
mov ss,ax            ;设置栈的段地址，SS=1000H，不能直接向段寄存器 SS 中送入
                     ;数据，所以用 ax 中转。

mov sp,0010H         ;设置栈顶的偏移地址，因栈为空，所以 sp=0010H。如果
                     ;对栈为空时 SP 的设置还有疑问，复习 3.7 节、问题 3.6
                     ;上面的 3 条指令设置栈顶地址。编程中要自己注意栈的大小。
push ax
push bx
push ds
```

问题 3.8

编程：

(1) 将 10000H~1000FH 这段空间当作栈，初始状态栈是空的；

(2) 设置 AX=001AH，BX=001BH；

(3) 将 AX、BX 中的数据入栈；

(4) 然后将 AX、BX 清零；
(5) 从栈中恢复 AX、BX 原来的内容。

思考后看分析。

分析：

代码如下。

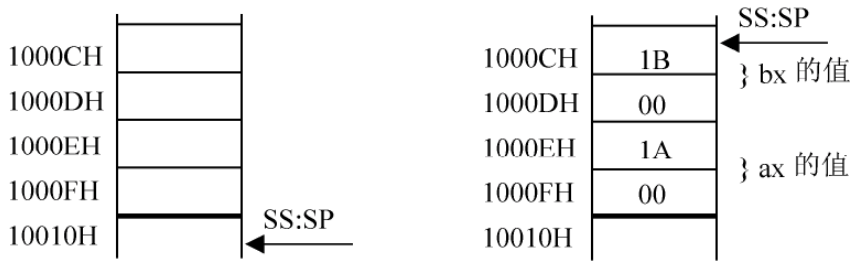
```
mov ax,1000H
mov ss,ax
mov sp,0010H          ;初始化栈顶，栈的情况如图 3.15(a)所示

mov ax,001AH
mov bx,001BH

push ax
push bx               ;ax、bx 入栈，栈的情况如图 3.15(b)所示

sub ax,ax             ;将 ax 清零，也可以用 mov ax,0，
                      ;sub ax,ax 的机器码为 2 个字节，
                      ;mov ax,0 的机器码为 3 个字节。
sub bx,bx

pop bx                ;从栈中恢复 ax、bx 原来的数据，当前栈顶的内容是 bx
pop ax                ;中原来的内容：001BH，ax 中原来的内容 001AH 在栈顶
                      ;的下面，所以要先 pop bx，然后再 pop ax。
```



(a) 栈空的情况　　(b) ax、bx 入栈的情况

图 3.15　栈的情况示意(1)

从上面的程序我们看到，用栈来暂存以后需要恢复的寄存器中的内容时，出栈的顺序要和入栈的顺序相反，因为最后入栈的寄存器的内容在栈顶，所以在恢复时，要最先出栈。

问题 3.9

编程：
(1) 将 10000H~1000FH 这段空间当作栈，初始状态栈是空的；
(2) 设置 AX=001AH，BX=001BH；
(3) 利用栈，交换 AX 和 BX 中的数据。

思考后看分析。

分析：

代码如下。

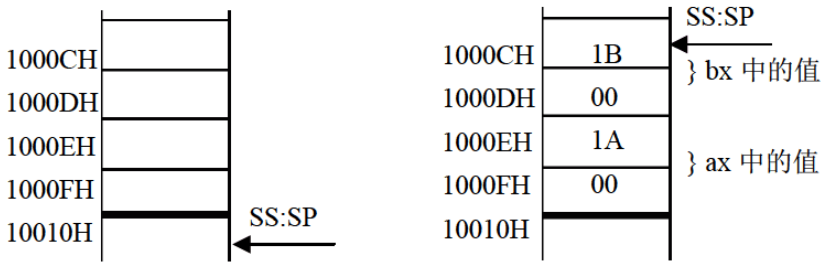
```
mov ax,1000H
mov ss,ax
mov sp,0010H          ;初始化栈顶，栈的情况如图 3.16(a)所示

mov ax,001AH
mov bx,001BH

push ax
push bx               ;ax、bx 入栈，栈的情况如图 3.16(b)所示

pop ax                ;当前栈顶的数据是 bx 中原来的数据：001BH;
                      ;所以先 pop ax，ax=001BH;

pop bx                ;执行 pop ax 后，栈顶的数据为 ax 原来的数据;
                      ;所以再 pop bx，bx=001AH;
```



(a) 栈空的情况　　(b) ax、bx 入栈的情况

图 3.16　栈的情况示意(2)

问题 3.10

如果要在 10000H 处写入字型数据 2266H，可以用以下的代码完成：

```
mov ax,1000H
mov ds,ax
mov ax,2266H
mov [0],ax
```

补全下面的代码，使它能够完成同样的功能：在 10000H 处写入字型数据 2266H。

要求：不能使用“mov 内存单元，寄存器”这类指令。

_____

_____

_____

```
mov ax,2266H
push ax
```

思考后看分析。

分析：

我们来看需补全代码的最后两条指令，将 ax 中的 2266H 压入栈中，也就是说，最终应由 push ax 将 2266H 写入 10000H 处。问题的关键就在于：如何使 push ax 访问的内存单元是 10000H。

push ax 是入栈指令，它将在栈顶之上压入新的数据。一定要注意：它的执行过程是，先将记录栈顶偏移地址的 SP 寄存器中的内容减 2，使得 SS:SP 指向新的栈顶单元，然后再将寄存器中的数据送入 SS:SP 指向的新的栈顶单元。

所以，要在执行 push ax 之前，将 SS:SP 指向 10002H(可以设 SS=1000H，SP=0002H)，这样，在执行 push ax 的时候，CPU 先将 SP=SP-2，使得 SS:SP 指向 10000H，再将 ax 中的数据送入 SS:SP 指向的内存单元处，即 10000H 处。

完成的程序如下。

```
mov ax,1000H
mov ss,ax
mov sp,2
mov ax,2266H
push ax
```

从问题 3.10 的分析中可以看出，push、pop 实质上就是一种内存传送指令，可以在寄存器和内存之间传送数据，与 mov 指令不同的是，push 和 pop 指令访问的内存单元的地址不是在指令中给出的，而是由 SS:SP 指出的。同时，push 和 pop 指令还要改变 SP 中的内容。

我们要十分清楚的是，push 和 pop 指令同 mov 指令不同，CPU 执行 mov 指令只需一步操作，就是传送，而执行 push、pop 指令却需要两步操作。执行 push 时，CPU 的两步操作是：先改变 SP，后向 SS:SP 处传送。执行 pop 时，CPU 的两步操作是：先读取 SS:SP 处的数据，后改变 SP。

注意，push，pop 等栈操作指令，修改的只是 SP。也就是说，栈顶的变化范围最大为：0~FFFFH。

提供：SS、SP 指示栈顶；改变 SP 后写内存的入栈指令；读内存后改变 SP 的出栈指令。这就是 8086CPU 提供的栈操作机制。

栈 的 综 述

(1) 8086CPU 提供了栈操作机制，方案如下。

在 SS、SP 中存放栈顶的段地址和偏移地址；

提供入栈和出栈指令，它们根据 SS:SP 指示的地址，按照栈的方式访问内存单元。

(2) push 指令的执行步骤：①SP=SP−2；②向 SS:SP 指向的字单元中送入数据。
(3) pop 指令的执行步骤：①从 SS:SP 指向的字单元中读取数据；②SP=SP+2。
(4) 任意时刻，SS:SP 指向栈顶元素。
(5) 8086CPU 只记录栈顶，栈空间的大小我们要自己管理。
(6) 用栈来暂存以后需要恢复的寄存器的内容时，寄存器出栈的顺序要和入栈的顺序相反。
(7) push、pop 实质上是一种内存传送指令，注意它们的灵活应用。

栈是一种非常重要的机制，一定要深入理解，灵活掌握。

3.10 栈　　段

前面讲过(参见 2.8 节)，对于 8086PC 机，在编程时，可以根据需要，将一组内存单元定义为一个段。我们可以将长度为 N(N≤64KB)的一组地址连续、起始地址为 16 的倍数的内存单元，当作栈空间来用，从而定义了一个栈段。比如，我们将 10010H~1001FH 这段长度为 16 字节的内存空间当作栈来用，以栈的方式进行访问。这段空间就可以称为一个栈段，段地址为 1001H，大小为 16 字节。

将一段内存当作栈段，仅仅是我们在编程时的一种安排，CPU 并不会由于这种安排，就在执行 push、pop 等栈操作指令时自动地将我们定义的栈段当作栈空间来访问。如何使得如 push、pop 等栈操作指令访问我们定义的栈段呢？前面我们已经讨论过，就是要将 SS:SP 指向我们定义的栈段。

问题 3.11

如果将 10000H~1FFFFH 这段空间当作栈段，初始状态栈是空的，此时，SS=1000H，SP=？

思考后看分析。

分析：

如果将 10000H~1FFFFH 这段空间当作栈段，SS=1000H，栈空间为 64KB，栈最底部的字单元地址为 1000:FFFE。任意时刻，SS:SP 指向栈顶单元，当栈中只有一个元素的时候，SS=1000H，SP=FFFEH。栈为空，就相当于栈中唯一的元素出栈，出栈后，SP=SP+2。

SP 原来为 FFFEH，加 2 后 SP=0，所以，当栈为空的时候，SS=1000H，SP=0。

换一个角度看，任意时刻，SS:SP 指向栈顶元素，当栈为空的时候，栈中没有元素，也就不存在栈顶元素，所以 SS:SP 只能指向栈的最底部单元下面的单元，该单元的地址为栈最底部的字单元的地址+2。栈最底部字单元的地址为 1000:FFFE，所以栈空时，

SP=0000H。

问题 3.12

一个栈段最大可以设为多少？为什么？

思考后看分析。

分析：

这个问题显而易见，提出来只是为了提示我们将相关的知识融会起来。首先从栈操作指令所完成的功能的角度上来看，push、pop 等指令在执行的时候只修改 SP，所以栈顶的变化范围是 0~FFFFH，从栈空时候的 SP=0，一直压栈，直到栈满时 SP=0；如果再次压栈，栈顶将环绕，覆盖了原来栈中的内容。所以一个栈段的容量最大为 64KB。

段 的 综 述

我们可以将一段内存定义为一个段，用一个段地址指示段，用偏移地址访问段内的单元。这完全是我们自己的安排。

我们可以用一个段存放数据，将它定义为“数据段”；
我们可以用一个段存放代码，将它定义为“代码段”；
我们可以用一个段当作栈，将它定义为“栈段”。

我们可以这样安排，但若要让 CPU 按照我们的安排来访问这些段，就要：

对于数据段，将它的段地址放在 DS 中，用 mov、add、sub 等访问内存单元的指令时，CPU 就将我们定义的数据段中的内容当作数据来访问；
对于代码段，将它的段地址放在 CS 中，将段中第一条指令的偏移地址放在 IP 中，这样 CPU 就将执行我们定义的代码段中的指令；
对于栈段，将它的段地址放在 SS 中，将栈顶单元的偏移地址放在 SP 中，这样 CPU 在需要进行栈操作的时候，比如执行 push、pop 指令等，就将我们定义的栈段当作栈空间来用。

可见，不管我们如何安排，CPU 将内存中的某段内容当作代码，是因 CS:IP 指向了那里；CPU 将某段内存当作栈，是因为 SS:SP 指向了那里。我们一定要清楚，什么是我们的安排，以及如何让 CPU 按我们的安排行事。要非常清楚 CPU 的工作机理，才能在控制 CPU 按照我们的安排运行的时候做到游刃有余。

比如我们将 10000H~1001FH 安排为代码段，并在里面存储如下代码：

```
mov ax,1000H
mov ss,ax
mov sp,0020H            ;初始化栈顶
mov ax,cs
mov ds,ax               ;设置数据段段地址
mov ax,[0]
```

```
add ax,[2]
mov bx,[4]
add bx,[6]
push ax
push bx
pop ax
pop bx
```

设置 CS=1000H，IP=0，这段代码将得到执行。可以看到，在这段代码中，我们又将 10000H~1001FH 安排为栈段和数据段。10000H~1001FH 这段内存，既是代码段，又是栈段和数据段。

一段内存，可以既是代码的存储空间，又是数据的存储空间，还可以是栈空间，也可以什么也不是。关键在于 CPU 中寄存器的设置，即 CS、IP，SS、SP，DS 的指向。

检测点 3.2

(1) 补全下面的程序，使其可以将 10000H~1000FH 中的 8 个字，逆序复制到 20000H~2000FH 中。逆序复制的含义如图 3.17 所示(图中内存里的数据均为假设)。

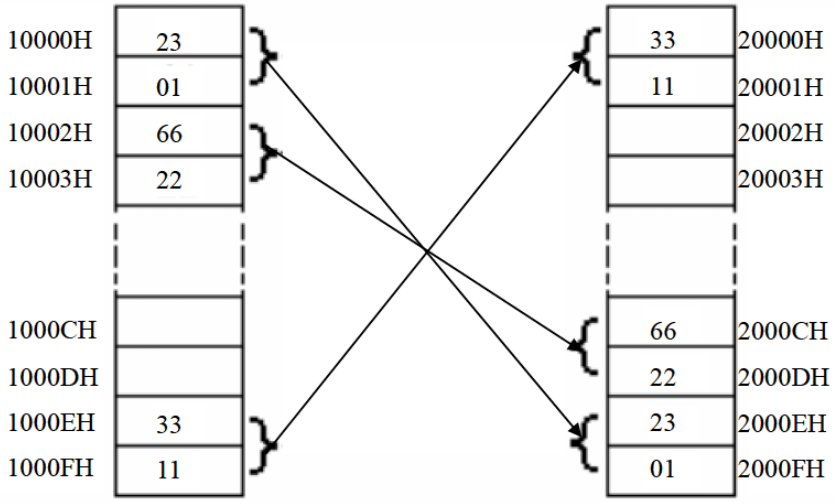


图 3.17 逆序复制示意图

```
mov ax,1000H
mov ds,ax

_____

_____

_____

push [0]
push [2]
push [4]
push [6]
push [8]
```

```
push [A]
push [C]
push [E]
```

(2) 补全下面的程序，使其可以将 10000H~1000FH 中的 8 个字，逆序复制到 20000H~2000FH 中。

```
mov ax,2000H
mov ds,ax

__________

__________

__________

pop [E]
pop [C]
pop [A]
pop [8]
pop [6]
pop [4]
pop [2]
pop [0]
```

实验 2　用机器指令和汇编指令编程

1. 预备知识：Debug 的使用

前面实验中，讲了 Debug 一些主要命令的用法，这里，再补充一些关于 Debug 的知识。

(1) 关于 D 命令。

从上次实验中，我们知道，D 命令是查看内存单元的命令，可以用“d 段地址:偏移地址”的格式查看指定的内存单元的内容，上次实验中，D 命令后面的段地址和偏移地址都是直接给出的。

现在，我们知道段地址是放在段寄存器中的，在 D 命令后面直接给出段地址，是 Debug 提供的一种直观的操作方式。D 命令是由 Debug 执行的，Debug 在执行“d 1000:0”这样的命令时，也会先将段地址 1000H 送入段寄存器中。

Debug 是靠什么来执行 D 命令的？当然是一段程序。

谁来执行这段程序？当然是 CPU。

CPU 在访问内存单元的时候从哪里得到内存单元的段地址？从段寄存器中得到。

所以，Debug 在其处理 D 命令的程序段中，必须有将段地址送入段寄存器的代码。

段寄存器有 4 个：CS、DS、SS、ES，将段地址送入哪个段寄存器呢？

首先不能是 CS，因为 CS:IP 必须指向 Debug 处理 D 命令的代码，也不能是 SS，因为 SS:SP 要指向栈顶。这样只剩下了 DS 和 ES 可以选择，放在哪里呢？我们知道，访问内存的指令如“mov ax,[0]”等一般都默认段地址在 ds 中，所以 Debug 在执行如“d 段地址:偏移地址”这种 D 命令时，将段地址送入 ds 中比较方便。

D 命令也提供了一种符合 CPU 机理的格式：“d 段寄存器:偏移地址”，以段寄存器中的数据为段地址 SA，列出从 SA:偏移地址开始的内存区间中的数据。以下是几个例子。

```
①  -r ds
   :1000
   -d ds:0          ;查看从 1000:0 开始的内存区间中的内容
②  -r ds
   :1000
   -d ds:10 18      ;查看 1000:10~1000:18 中的内容
③  -d cs:0          ;查看当前代码段中的指令代码
④  -d ss:0          ;查看当前栈段中的内容
```

(2) 在 E、A、U 命令中使用段寄存器。

在 E、A、U 这些可以带有内存单元地址的命令中，也可以同 D 命令一样，用段寄存器表示内存单元的段地址，以下是几个例子。

```
①  -r ds
   :1000
   -e ds:0 11 22 33 44 55 66        ;在从 1000:0 开始的内存区间中写入数据
②  -u cs:0          ;以汇编指令的形式，显示当前代码段中的代码，0 代码的偏移地址
③  -r ds
   :1000
   -a ds:0          ;以汇编指令的形式，向从 1000:0 开始的内存单元中写入指令
```

(3) 下一条指令执行了吗？

在 Debug 中，用 A 命令写一段程序：

```
mov ax,2000
mov ss,ax
mov sp,10               ;安排 2000:0000~2000:000F 为栈空间，初始化栈顶

mov ax,3123
push ax
mov ax,3366
push ax                 ;在栈中压入两个数据
```

仔细看一下图 3.18 中单步执行的结果，你发现了什么问题？

```
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
-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   NV 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   NV 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   NV UP EI PL NZ NA PO NC
0B39:0108 B82331        MOV     AX,3123
-
```

图 3.18　mov sp,10 到哪里去了？

在用 T 命令单步执行 mov ax,2000 后，显示出当前 CPU 各个寄存器的状态和下一步要执行的指令：mov ss,ax；

在用 T 命令单步执行 mov ss,ax 后，显示出当前 CPU 各个寄存器的状态和下一步要执行的指令……，在这里我们发现了一个问题：mov ss,ax 的下一条指令应该是 mov sp,10，怎么变成了 mov ax,3123？

mov sp,10 到哪里去了？它被执行了吗？

我们再仔细观察，发现：

在程序执行前，ax=0000，ss=0b39，sp=ffee
在用 T 命令单步执行 mov ax,2000 后，ax=2000；ss=0b39；sp=ffee
在用 T 命令单步执行 mov ss,ax 后，ax=2000；ss=2000；sp=0010

注意，在用 T 命令单步执行 mov ss,ax 前，ss=0b39，**sp=ffee**，而执行后 ss=2000，**sp=0010**。ss 变为 2000 是正常的，这正是 mov ss,ax 的执行结果。可是 sp 变为 0010 是怎么回事？在这期间，能够将 sp 设为 0010 的只有指令 mov sp,10，看来，mov sp,10 一定是得到了执行。

那么，mov sp,10 是在什么时候被执行的呢？当然是在 mov ss,ax 之后，因为它就是 mov ss,ax 的下一条指令。显然，在用 T 命令执行 mov ss,ax 的时候，它的下一条指令 mov sp,10 也紧接着执行了。

整理一下我们分析的结果：在用 T 命令执行 mov ss,ax 的时候，它的下一条指令 mov sp,10 也紧接着执行了。一般情况下，用 T 命令执行一条指令后，会停止继续执行，显示

出当前 CPU 各个寄存器的状态和下一步要执行的指令，但 T 命令执行 mov ss,ax 的时候，没有做到这一点。

不单是 mov ss,ax，对于如 mov ss,bx，mov ss,[0]，pop ss 等指令都会发生上面的情况，这些指令有哪些共性呢？它们都是修改栈段寄存器 SS 的指令。

为什么会这样呢？要想彻底说清楚这里面的来龙去脉，在这里还为时过早，因为这涉及我们在以后的课程中要深入研究的内容：中断机制，它是我们后半部分课程中的一个主题。现在我们只要知道这一点就可以了：Debug 的 T 命令在执行修改寄存器 SS 的指令时，下一条指令也紧接着被执行。

2. 实验任务

(1) 使用 Debug，将下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。

```
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_____
```

(2) 仔细观察图 3.19 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？

可能要再做些实验才能发现其中的规律。如果你在这里就正确回答了这个问题，那么要恭喜你，因为你有很好的悟性。大多数的学习者对这个问题还是比较迷惑的，不过不要紧，因为随着课程的进行，这个问题的答案将逐渐变得显而易见。

```
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 0
-
-d 2000:0 f
2000:0000  00 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   NV 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   NV 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   NV 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   ....... ...
-
```

图 3.19　用 Debug 进行的实验

第 4 章　第一个程序

终于可以编写第 1 个完整的程序了，我们以前都是在 Debug 中写一些指令，在 Debug 中执行。现在我们将开始编写完整的汇编语言程序，用编译和连接程序将它们编译连接成为可执行文件(如*.exe 文件)，在操作系统中运行。这一章中，我们将编写第一个这样的程序。

为了能够透彻地理解一个完整的程序(尽管它看上去十分简单)，我们将经历一个漫长的过程。

4.1　一个源程序从写出到执行的过程

图 4.1 描述了一个汇编语言程序从写出到最终执行的简要过程。具体说明如下。

第一步：编写汇编源程序。

使用文本编辑器(如 Edit、记事本等)，用汇编语言编写汇编源程序。

这一步工作的结果是产生了一个存储源程序的文本文件。

第二步：对源程序进行编译连接。

使用汇编语言编译程序对源程序文件中的源程序进行编译，产生目标文件；再用连接程序对目标文件进行连接，生成可在操作系统中直接运行的可执行文件。

可执行文件包含两部分内容。

- 程序(从源程序中的汇编指令翻译过来的机器码)和数据(源程序中定义的数据)
- 相关的描述信息(比如，程序有多大、要占用多少内存空间等)

这一步工作的结果：产生了一个可在操作系统中运行的可执行文件。

第三步：执行可执行文件中的程序。

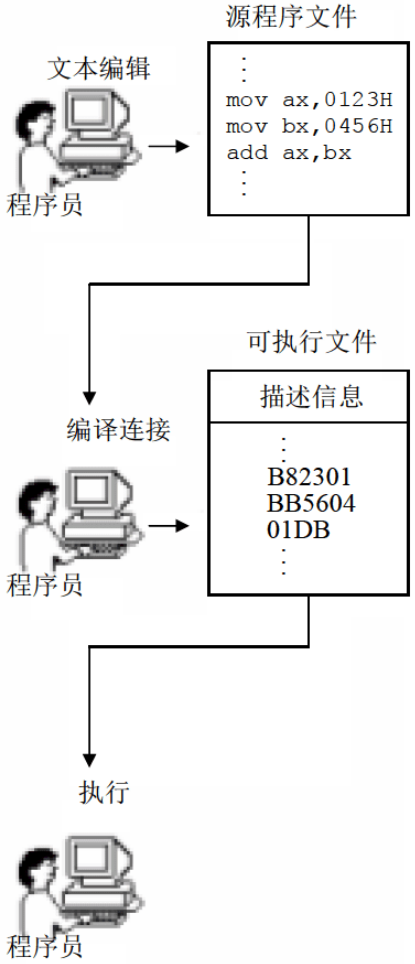


图 4.1　一个汇编语言程序从写出到执行的过程

在操作系统中，执行可执行文件中的程序。

操作系统依照可执行文件中的描述信息，将可执行文件中的机器码和数据加载入内存，并进行相关的初始化(比如设置 CS:IP 指向第一条要执行的指令)，然后由 CPU 执行程序。

下面我们将通过学习一个简单的程序来经历图 4.1 中所描述的过程。

4.2 源　程　序

下面就是一段简单的汇编语言源程序。

程序 4.1

```
assume cs:codesg

codesg segment

        mov ax,0123H
        mov bx,0456H
        add ax,bx
        add ax,ax

        mov ax,4c00H
        int 21H

codesg ends

end
```

下面对程序进行说明。

1. 伪指令

在汇编语言源程序中，包含两种指令，一种是汇编指令，一种是伪指令。汇编指令是有对应的机器码的指令，可以被编译为机器指令，最终为 CPU 所执行。而伪指令没有对应的机器指令，最终不被 CPU 所执行。那么谁来执行伪指令呢？伪指令是由编译器来执行的指令，编译器根据伪指令来进行相关的编译工作。

你现在能看出来程序 4.1 中哪些指令是伪指令吗？

程序 4.1 中出现了 3 种伪指令。

(1)　XXX segment

⋮

⋮

⋮

XXX ends

segment 和 ends 是一对成对使用的伪指令，这是在写可被编译器编译的汇编程序时，必须要用到的一对伪指令。segment 和 ends 的功能是定义一个段，segment 说明一个段开始，ends 说明一个段结束。一个段必须有一个名称来标识，使用格式为：

```
段名 segment
    ⋮
段名 ends
```

比如，程序 4.1 中的：

```
codesg segment        ;定义一个段，段的名称为“codesg”，这个段从此开始
    ⋮
codesg ends           ;名称为“codesg”的段到此结束
```

一个汇编程序是由多个段组成的，这些段被用来存放代码、数据或当作栈空间来使用。我们在前面的课程中所讲解的段的概念，在汇编源程序中得到了应用与体现，一个源程序中所有将被计算机所处理的信息：指令、数据、栈，被划分到了不同的段中。

一个有意义的汇编程序中至少要有一个段，这个段用来存放代码。

我们可以看到，程序 4.1 中，在 codesg segment 和 codesg ends 之间写的汇编指令是这个段中存放的内容，这是一个代码段(其中还有我们不认识的指令，后面会进行讲解)。

(2) end

end 是一个汇编程序的结束标记，编译器在编译汇编程序的过程中，如果碰到了伪指令 end，就结束对源程序的编译。所以，在我们写程序的时候，如果程序写完了，要在结尾处加上伪指令 end。否则，编译器在编译程序时，无法知道程序在何处结束。

注意，不要搞混了 end 和 ends，ends 是和 segment 成对使用的，标记一个段的结束，ends 的含义可理解为“end segment”。我们这里讲的 end 的作用是标记整个程序的结束。

(3) assume

这条伪指令的含义为“假设”。它假设某一段寄存器和程序中的某一个用 segment…ends 定义的段相关联。通过 assume 说明这种关联，在需要的情况下，编译程序可以将段寄存器和某一个具体的段相联系。assume 并不是一条非要深入理解不可的伪指令，以后我们编程时，记着用 assume 将有特定用途的段和相关的段寄存器关联起来即可。

比如，在程序 4.1 中，我们用 codesg segment ... codesg ends 定义了一个名为 codseg 的段，在这个段中存放代码，所以这个段是一个代码段。在程序的开头，用 assume cs:codesg 将用作代码段的段 codesg 和 CPU 中的段寄存器 cs 联系起来。

2. 源程序中的“程序”

用汇编语言写的源程序，包括伪指令和汇编指令，我们编程的最终目的是让计算机完成一定的任务。源程序中的汇编指令组成了最终由计算机执行的程序，而源程序中的伪指令是由编译器来处理的，它们并不实现我们编程的最终目的。这里所说的程序就是指源程序中最终由计算机执行、处理的指令或数据。

注意，以后可以将源程序文件中的所有内容称为源程序，将源程序中最终由计算机执行、处理的指令或数据，称为程序。程序最先以汇编指令的形式存在源程序中，经编译、连接后转变为机器码，存储在可执行文件中。这个过程如图 4.2 所示。

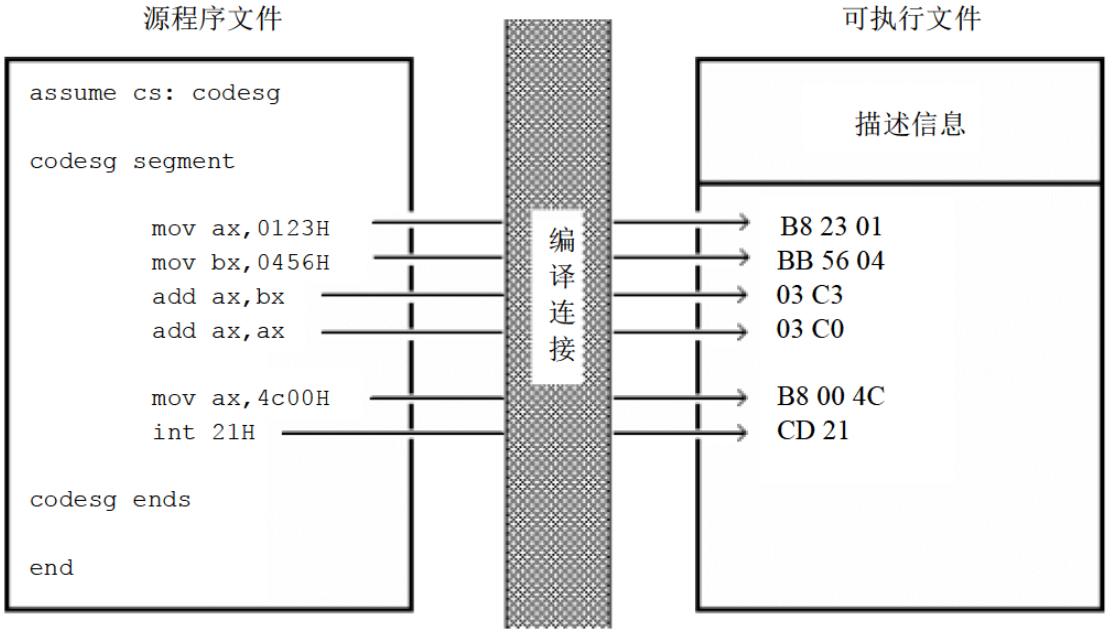


图 4.2　程序经编译连接后变为机器码

3. 标号

汇编源程序中，除了汇编指令和伪指令外，还有一些标号，比如“codesg”。一个标号指代了一个地址。比如 codesg 在 segment 的前面，作为一个段的名称，这个段的名称最终将被编译、连接程序处理为一个段的段地址。

4. 程序的结构

我们现在讨论一下汇编程序的结构。在前 3 章中，我们都是通过直接在 Debug 中写入汇编指令来写汇编程序，对于十分简短的程序这样做的确方便。可对于大一些的程序，就不能如此了。我们需要写出能让编译器进行编译的源程序，这样的源程序应该具备起码的结构。

源程序是由一些段构成的。我们可以在这些段中存放代码、数据，或将某个段当作栈空间。我们现在来一步步地完成一个小程序，从这个过程中体会一下汇编程序中的基本要素和汇编程序的简单框架。

任务：编程运算 2^3。源程序应该怎样来写呢？

(1) 我们要定义一个段，名称为 abc。

```
abc segment
      ⋮
abc ends
```

(2) 在这个段中写入汇编指令，来实现我们的任务。

```
abc segment

  mov ax,2
  add ax,ax
  add ax,ax

abc ends
```

(3) 然后，要指出程序在何处结束。

```
abc segment

  mov ax,2
  add ax,ax
  add ax,ax

abc ends

end
```

(4) abc 被当作代码段来用，所以，应该将 abc 和 cs 联系起来。(当然，对于这个程序，也不是非这样做不可。)

```
assume cs:abc

abc segment

    mov ax,2
    add ax,ax
    add ax,ax

abc ends

end
```

最终写成的程序如程序 4.2 所示。

程序 4.2

```
assume cs:abc

abc segment
```

```
  mov ax,2
  add ax,ax
  add ax,ax

abc ends

end
```

5. 程序返回

我们的程序最先以汇编指令的形式存在源程序中，经编译、连接后转变为机器码，存储在可执行文件中，那么，它怎样得到运行呢？

下面，我们在 DOS(一个单任务操作系统)的基础上，简单地讨论一下这个问题。

一个程序 P2 在可执行文件中，则必须有一个正在运行的程序 P1，将 P2 从可执行文件中加载入内存后，将 CPU 的控制权交给 P2，P2 才能得以运行。P2 开始运行后，P1 暂停运行。

而当 P2 运行完毕后，应该将 CPU 的控制权交还给使它得以运行的程序 P1，此后，P1 继续运行。

现在，我们知道，一个程序结束后，将 CPU 的控制权交还给使它得以运行的程序，我们称这个过程为：**程序返回**。那么，如何返回呢？应该在程序的末尾添加返回的程序段。

我们回过头来，看一下程序 4.1 中的两条指令：

```
mov ax,4c00H
int 21H
```

这两条指令所实现的功能就是程序返回。

在目前阶段，我们不必去理解 int 21H 指令的含义，和为什么要在这条指令的前面加上指令 mov ax,4c00H。我们只要知道，在程序的末尾使用这两条指令就可以实现程序返回。

到目前为止，我们好像已经遇到了几个和结束相关的内容：段结束、程序结束、程序返回。表 4.1 展示了它们的区别。

表 4.1　与结束相关的概念

目　的	相关指令	指令性质	指令执行者
通知编译器一个段结束	段名 ends	伪指令	编译时，由编译器执行
通知编译器程序结束	end	伪指令	编译时，由编译器执行
程序返回	mov ax,4c00H　int 21H	汇编指令	执行时，由 CPU 执行

6. 语法错误和逻辑错误

可见，程序 4.2 在运行时会引发一些问题，因为程序没有返回。当然，这个错误在编译的时候是不能表现出来的，也就是说，程序 4.2 对于编译器来说是正确的程序。

一般说来，程序在编译时被编译器发现的错误是语法错误，比如将程序 4.2 写成如下这样就会发生语法错误：

```
aume cs:abc

abc segment

  mov ax,2
  add ax,ax
  add ax,ax

end
```

显然，程序中有编译器不能识别的 aume，而且编译器在编译的过程中也无法知道 abc 段到何处结束。

在源程序编译后，在运行时发生的错误是逻辑错误。语法错误容易发现，也容易解决。而逻辑错误通常不容易被发现。不过，程序 4.2 中的错误却显而易见，我们将它改正过来：

```
assume cs:abc
abc segment

  mov ax,2
  add ax,ax
  add ax,ax

  mov ax,4c00H
  int 21H

abc ends
end
```

4.3 编辑源程序

可以用任意的文本编辑器来编辑源程序，只要最终将其存储为纯文本文件即可。在我们的课程中，使用 DOS 下的 Edit。以程序 4.1 为例，说明工作过程。

(1) 进入 DOS 方式，运行 Edit，如图 4.3 所示。

```
C:\>edit_
```

图 4.3 运行 Edit

(2)　在 Edit 中编辑程序，如图 4.4 所示。

图 4.4　在 Edit 中编辑程序

(3)　将程序保存为文件 c:\1.asm 后，退出 Edit，结束对源程序的编辑。

4.4　编　　译

在 4.3 节中，完成对源程序的编辑后，得到一个源程序文件 c:\1.asm。可以对其进行编译，生成包含机器代码的目标文件。

在编译一个源程序之前首先要找到一个相应的编译器。在我们的课程中，采用微软的 masm5.0 汇编编译器，文件名为 masm.exe。假设汇编编译器在 c:\masm 目录下。可以按照下面的过程来进行源程序的编译，以 c:\1.asm 为例。

(1)　进入 DOS 方式，进入 c:\masm 目录，运行 masm.exe，如图 4.5 所示。

```
C:\masm>masm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Source filename [.ASM]: _
```

图 4.5　运行 masm.exe

图 4.5 中，运行 masm 后，首先显示出一些版本信息，然后提示输入将要被编译的源程序文件的名称。注意，“[.ASM]”提示我们，默认的文件扩展名是 asm，比如，要编译的源程序文件名是“p1.asm”，只要在这里输入“p1”即可。可如果源程序文件不是以 asm 为扩展名的话，就要输入它的全名。比如源程序文件名为“p1.txt”，就要输入全名。

在输入源程序文件名的时候一定要指明它所在的路径。如果文件就在当前路径下，只输入文件名就可以，可如果文件在其他的目录中，则要输入路径，比如，要编译的文件 p1.txt 在“c:\windows\desktop”下，则要输入“c:\windows\desktop\p1.txt”。

这里，我们要编译的文件是 C 盘根目录下的 1.asm，所以此处输入“c:\1.asm”。

(2) 输入要编译的源程序文件名后，按 Enter 键，屏幕显示如图 4.6 所示。

```
C:\masm>masm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Source filename [.ASM]: c:\1.asm
Object filename [1.OBJ]:
```

图 4.6 输入要编译的源程序文件名

图 4.6 中，在输入源程序文件名后，程序继续提示我们输入要编译出的目标文件的名称，目标文件是我们对一个源程序进行编译要得到的最终结果。注意屏幕上的显示：“[1.OBJ]”，因为我们已经输入了源程序文件名为 1.asm，则编译程序默认要输出的目标文件名为 1.obj，所以可以不必再另行指定文件名。直接按 Enter 键，编译程序将在当前的目录下，生成 1.obj 文件。

这里，也可以指定生成的目标文件所在的目录，比如，想让编译程序在“c:\windows\desktop”下生成目标文件 1.obj，则可输入“c:\windows\desktop\1”。

我们直接按 Enter 键，使用编译程序设定的目标文件名。

(3) 确定了目标文件的名称后，屏幕显示如图 4.7 所示。

```
C:\masm>masm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Source filename [.ASM]: 1.asm
Object filename [1.OBJ]:
Source listing  [NUL.LST]: _
```

图 4.7 确定目标文件名称

图 4.7 中，编译程序提示输入列表文件的名称，这个文件是编译器将源程序编译为目标文件的过程中产生的中间结果。可以让编译器不生成这个文件，直接按 Enter 键即可。

(4) 忽略了列表文件的生成后，屏幕显示如图 4.8 所示。

```
C:\masm>masm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Source filename [.ASM]: c:\1.asm
Object filename [1.OBJ]:
Source listing  [NUL.LST]:
Cross-reference [NUL.CRF]: _
```

图 4.8 忽略列表文件的生成

图 4.8 中，编译程序提示输入交叉引用文件的名称，这个文件同列表文件一样，是编译器将源程序编译为目标文件过程中产生的中间结果。可以让编译器不生成这个文件，直接按 Enter 键即可。

(5) 忽略了交叉引用文件的生成后，屏幕显示如图 4.9 所示。

```
C:\masm>masm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Source filename [.ASM]: c:\1.asm
Object filename [1.OBJ]:
Source listing  [NUL.LST]:
Cross-reference [NUL.CRF]:

  51522 + 422654 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors

C:\masm>_
```

图 4.9　源程序的编译结束

图 4.9 中，对源程序的编译结束，编译器输出的最后两行告诉我们这个源程序没有警告错误和必须要改正的错误。

上面我们通过对 C 盘根目录下的 1.asm 进行编译的过程，展示了使用汇编编译器对源程序进行编译的方法。按照上面的过程进行了编译之后，在编译器 masm.exe 运行的目录 c:\masm 下(即当前路径下)，将出现一个新的文件：1.obj，这是对源程序 1.asm 进行编译所得到的结果。当然，如果编译的过程中出现错误，那么将得不到目标文件。一般来说，有两类错误使我们得不到所期望的目标文件：

(1) 程序中有“Severe Errors”；
(2) 找不到所给出的源程序文件。

注意，在编译的过程中，我们提供了一个输入，即源程序文件。最多可以得到 3 个输出：目标文件(.obj)、列表文件(.lst)、交叉引用文件(.crf)，这 3 个输出文件中，目标文件是我们最终要得到的结果，而另外两个只是中间结果，可以让编译器忽略对它们的生成。在汇编课程中，我们不讨论这两类文件。

4.5 连　　接

在对源程序进行编译得到目标文件后，我们需要对目标文件进行连接，从而得到可执行文件。接续上一节的过程，我们已经对 c:\1.asm 进行编译得到 c:\masm\1.obj，现在再将 c:\masm\1.obj 连接为 c:\masm\1.exe。

我们使用微软的 Overlay Linker3.60 连接器，文件名为 link.exe，假设连接器在 c:\masm 目录下。可以按照下面的过程来进行程序的连接，以 c:\masm\1.obj 为例。

(1) 进入 DOS 方式，进入 c:\masm 目录，运行 link.exe，如图 4.10 所示。

图 4.10 中，运行 link 后，首先显示出一些版本信息，然后提示输入将要被连接的目标文件的名称。注意，“[.OBJ]”提示我们，默认的文件扩展名是 obj，比如要连接的目标文件名是“p1.obj”，只要在这里输入“p1”即可。可如果文件不是以 obj 为扩展名，就要输入它的全名。比如目标文件名为“p1.bin”，就要输入全名。

在输入目标文件名的时候，要注意指明它所在的路径。这里，要连接的文件是当前目录下的 1.obj，所以此处输入“1”。

```
C:\masm>link

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Object Modules [.OBJ]: _
```

图 4.10　运行 link.exe

(2) 输入要连接的目标文件名后，按 Enter 键，屏幕显示如图 4.11 所示。

```
C:\masm>link

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Object Modules [.OBJ]: 1
Run File [1.EXE]: _
```

图 4.11　确定要连接的目标文件名

图 4.11 中，在输入目标文件名后，程序继续提示我们输入要生成的可执行文件的名称，可执行文件是我们对一个程序进行连接要得到的最终结果。注意屏幕上的显示：“[1.EXE]”，因为已经确定了目标文件名为 1.obj，则程序默认要输出的可执行文件名为 1.EXE，所以可以不必再另行指定文件名。直接按 Enter 键，编译程序将在当前的目录下，生成 1.EXE 文件。

这里，也可以指定生成的可执行文件所在的目录，比如，想让连接程序在“c:\windows\desktop”下生成可执行文件 1.EXE，则可输入“c:\windows\desktop\1”。

我们直接按 Enter 键，使用连接程序设定的可执行文件名。

(3) 确定了可执行文件的名称后，屏幕显示如图 4.12 所示。

```
C:\masm>link

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Object Modules [.OBJ]: 1
Run File [1.EXE]:
List File [NUL.MAP]: _
```

图 4.12　确定可执行文件名

图 4.12 中，连接程序提示输入映像文件的名称，这个文件是连接程序将目标文件连接为可执行文件过程中产生的中间结果，可以让连接程序不生成这个文件，直接按 Enter 键即可。

(4) 忽略了映像文件的生成后，屏幕显示如图 4.13 所示。

图 4.13 中，连接程序提示输入库文件的名称。库文件里面包含了一些可以调用的子程序，如果程序中调用了某一个库文件中的子程序，就需要在连接的时候，将这个库文件

和目标文件连接到一起，生成可执行文件。但是，这个程序中没有调用任何子程序，所以，这里忽略库文件名的输入，直接按 Enter 键即可。

```
C:\masm>link

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Object Modules [.OBJ]: 1
Run File [1.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
```

图 4.13　忽略映像文件的生成

(5)　忽略了库文件的连接后，屏幕显示如图 4.14 所示。

```
C:\masm>link

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Object Modules [.OBJ]: 1
Run File [1.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
LINK : warning L4021: no stack segment

C:\masm>
```

图 4.14　忽略库文件的连接

图 4.14 中，对目标文件的连接结束，连接程序输出的最后一行告诉我们，这个程序中有一个警告错误：“没有栈段”，这里我们不理会这个错误。

上面我们通过对当前路径下的 1.obj 进行连接的过程，展示了使用连接器对目标文件进行连接的方法。按照上面的过程进行了连接之后，在连接器 link.exe 运行的目录 c:\masm 下(即当前路径下)，将出现一个新的文件：1.exe，这是对目标文件 1.obj 进行连接所得到的结果。当然，如果连接过程中出现错误，那么将得不到可执行文件。

连接的作用是什么呢？

对于连接，我们也不想过多地讨论。实际上，在汇编课程中，我们将会接触到许多知识、概念，对于这些，我们并不是都有深入讨论的必要。

这里再次强调一下，我们学习汇编的主要目的，就是通过用汇编语言进行编程而深入地理解计算机底层的基本工作机理，达到可以随心所欲地控制计算机的目的。基于这种考虑，我们的编程活动，大都是直接对硬件进行的。我们希望直接对硬件编程，却并不希望用机器码编程。我们用汇编语言编程，就要用到编辑器(Edit)、编译器(masm)、连接器(link)、调试工具(Debug)等所有工具，而这些工具都是在操作系统之上运行的程序，所以我们的学习过程必须在操作系统的环境中进行。我们在一个操作系统环境中，使用了许多工具，这势必要牵扯到操作系统、编译原理等方面的知识和原理。我们只是利用这些环境、工具来方便我们的学习，而不希望这些东西分散了我们的注意力。所以，对于涉及而又不在我们学习的主要内容之中的东西，我们只做简单的解释。

好了，我们简单地讲连接的作用，连接的作用有以下几个。

(1) 当源程序很大时，可以将它分为多个源程序文件来编译，每个源程序编译成为目标文件后，再用连接程序将它们连接到一起，生成一个可执行文件；

(2) 程序中调用了某个库文件中的子程序，需要将这个库文件和该程序生成的目标文件连接到一起，生成一个可执行文件；

(3) 一个源程序编译后，得到了存有机器码的目标文件，目标文件中的有些内容还不能直接用来生成可执行文件，连接程序将这些内容处理为最终的可执行信息。所以，在只有一个源程序文件，而又不需要调用某个库中的子程序的情况下，也必须用连接程序对目标文件进行处理，生成可执行文件。

注意，对于连接的过程，可执行文件是我们要得到的最终结果。

4.6 以简化的方式进行编译和连接

在前面的内容里，介绍了如何使用 masm 和 link 进行编译和连接。可以看出，我们编译、连接的最终目的是用源程序文件生成可执行文件。在这个过程中所产生的中间文件都可以忽略。我们可以用一种较为简捷的方式进行编译、连接。简捷的编译过程如图 4.15 所示。

```
C:\masm>masm c:\1;
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.


  51516 + 422660 Bytes symbol space free

       0 Warning Errors
       0 Severe  Errors

C:\masm>_
```

图 4.15 简捷的编译过程

注意图 4.15 中的命令行“masm c:\1;”，在 masm 后面加上被编译的源程序文件的路径、文件名，在命令行的结尾再加上分号，按 Enter 键后，编译器就对 c:\1.asm 进行编译，在当前路径下生成目标文件 1.obj，并在编译的过程中自动忽略中间文件的生成。

图 4.16 展示了简捷的连接过程。

```
C:\masm>link 1;

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

LINK : warning L4021: no stack segment

C:\masm>_
```

图 4.16 简捷的连接过程

注意图 4.16 中的命令行“link 1;”，在 link 后面加上被连接的目标文件的路径、文件

名，在命令行的结尾再加上分号，按 Enter 键后，连接程序就对当前路径下的 1.obj 进行处理，在当前路径下生成可执行文件 1.exe，并在过程中自动忽略中间文件的生成。

4.7　1.exe 的执行

现在，终于将我们的第一个汇编程序加工成了一个可在操作系统下执行的程序文件，我们现在执行一下，图 4.17 展示了 1.exe 的执行情况。

```
C:\masm>1
C:\masm>
```

图 4.17　执行 1.exe

奇怪吗？程序运行后，竟然没有任何结果，就和没有运行一样。那么，程序到底运行了吗？

程序当然是运行了，只是从屏幕上不可能看到任何运行结果，因为，我们的程序根本没有向显示器输出任何信息。程序只是做了一些将数据送入寄存器和加法的操作，而这些事情，我们不可能从显示屏上看出来。程序执行完成后，返回，屏幕上再次出现操作系统的提示符(图 4.17 中第 2 行)。

当然，我们不能总是写这样的看不到任何结果的程序，随着课程的进行，我们将会向显示器上输出信息，不过那将是几章以后的事情了，请耐心等待。

4.8　谁将可执行文件中的程序装载进入内存并使它运行？

我们在前面讲过，在 DOS 中，可执行文件中的程序 P1 若要运行，必须有一个正在运行的程序 P2，将 P1 从可执行文件中加载入内存，将 CPU 的控制权交给它，P1 才能得以运行；当 P1 运行完毕后，应该将 CPU 的控制权交还给使它得以运行的程序 P2。

按照上面的原理，再来看一下 4.7 节中 1.exe 的执行过程(思考相关的问题)。

(1)　在提示符“c:\masm”后面输入可执行文件的名字“1”，按 Enter 键。这时，请思考问题 4.1。

(2)　1.exe 中的程序运行。

(3)　运行结束，返回，再次显示提示符“c:\masm”。请思考问题 4.2。

问题 4.1

此时，有一个正在运行的程序将 1.exe 中的程序加载入内存，这个正在运行的程序是什么？它将程序加载入内存后，如何使程序得以运行？

问题 4.2

程序运行结束后，返回到哪里？

如果你对 DOS 有比较深入的了解，那么，很容易回答问题 4.1、问题 4.2 中所提出的问题。如果没有这种了解，可以先阅读下面的内容。

操作系统的外壳

操作系统是由多个功能模块组成的庞大、复杂的软件系统。任何通用的操作系统，都要提供一个称为 shell(外壳)的程序，用户(操作人员)使用这个程序来操作计算机系统进行工作。

DOS 中有一个程序 command.com，这个程序在 DOS 中称为命令解释器，也就是 DOS 系统的 shell。

DOS 启动时，先完成其他重要的初始化工作，然后运行 command.com，command.com 运行后，执行完其他的相关任务后，在屏幕上显示出由当前盘符和当前路径组成的提示符，比如："c:\"或"c:\windows"等，然后等待用户的输入。

用户可以输入所要执行的命令，比如，cd、dir、type 等，这些命令由 command 执行，command 执行完这些命令后，再次显示由当前盘符和当前路径组成的提示符，等待用户的输入。

如果用户要执行一个程序，则输入该程序的可执行文件的名称，command 首先根据文件名找到可执行文件，然后将这个可执行文件中的程序加载入内存，设置 CS:IP 指向程序的入口。此后，command 暂停运行，CPU 运行程序。程序运行结束后，返回到 command 中，command 再次显示由当前盘符和当前路径组成的提示符，等待用户的输入。

在 DOS 中，command 处理各种输入：命令或要执行的程序的文件名。我们就是通过 command 来进行工作的。

现在回答问题 4.1 和 4.2 中所提出的问题。

(1) 在 DOS 中直接执行 1.exe 时，是正在运行的 command，将 1.exe 中的程序加载入内存；

(2) command 设置 CPU 的 CS:IP 指向程序的第一条指令(即程序的入口)，从而使程序得以运行；

(3) 程序运行结束后，返回到 command 中，CPU 继续运行 command。

汇编程序从写出到执行的过程

到此，完成了一个汇编程序从写出到执行的全部过程。我们经历了这样一个历程：

编程 → 1.asm → 编译 → 1.obj → 连接 → 1.exe → 加载 → 内存中的程序 → 运行

(Edit) (masm) (link) (command) (CPU)

4.9　程序执行过程的跟踪

可以用 Debug 来跟踪一个程序的运行过程，这通常是必须要做的工作。我们写的程序在逻辑上不一定总是正确，对于简单的错误，仔细检查一下源程序就可以发现；而对于隐藏较深的错误，就必须对程序的执行过程进行跟踪分析才容易发现。

下面以在前面的内容中生成的可执行文件 1.exe 为例，讲解如何用 Debug 对程序的执行过程进行跟踪。

现在我们知道，在 DOS 中运行一个程序的时候，是由 command 将程序从可执行文件中加载入内存，并使其得以执行。但是，这样我们不能逐条指令地看到程序的执行过程，因为 command 的程序加载，设置 CS:IP 指向程序的入口的操作是连续完成的，而当 CS:IP 一指向程序的入口，command 就放弃了 CPU 的控制权，CPU 立即开始运行程序，直至程序结束。

为了观察程序的运行过程，可以使用 Debug。Debug 可以将程序加载入内存，设置 CS:IP 指向程序的入口，但 Debug 并不放弃对 CPU 的控制，这样，我们就可以使用 Debug 的相关命令来单步执行程序，查看每一条指令的执行结果。

具体方法如图 4.18 所示。

```
C:\masm>debug 1.exe
-_
```

图 4.18　用 Debug 加载程序

在提示符后输入“debug 1.exe”，按 Enter 键，Debug 将程序从 1.exe 中加载入内存，进行相关的初始化后设置 CS:IP 指向程序的入口。

接下来可以用 R 命令看一下各个寄存器的设置情况，如图 4.19 所示。

```
C:\masm>debug 1.exe
-r
AX=0000  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=129E  ES=129E  SS=12AE  CS=12AE  IP=0000   NV UP EI PL NZ NA PO NC
12AE:0000 B82301        MOV     AX,0123
-
```

图 4.19　程序加载后各个寄存器的内容

可以看到，Debug 将程序从可执行文件加载入内存后，cx 中存放的是程序的长度。1.exe 中程序的机器码共有 15 个字节。则 1.exe 加载后，cx 中的内容为 000FH。

现在程序已从 1.exe 中装入内存，接下来查看一下它的内容，可是我们查看哪里的内容呢？程序被装入内存的什么地方？我们如何得知？

这里，需要讲解一下在 DOS 系统中.EXE 文件中的程序的加载过程。图 4.20 针对我

们的问题，简要地展示了这个过程。

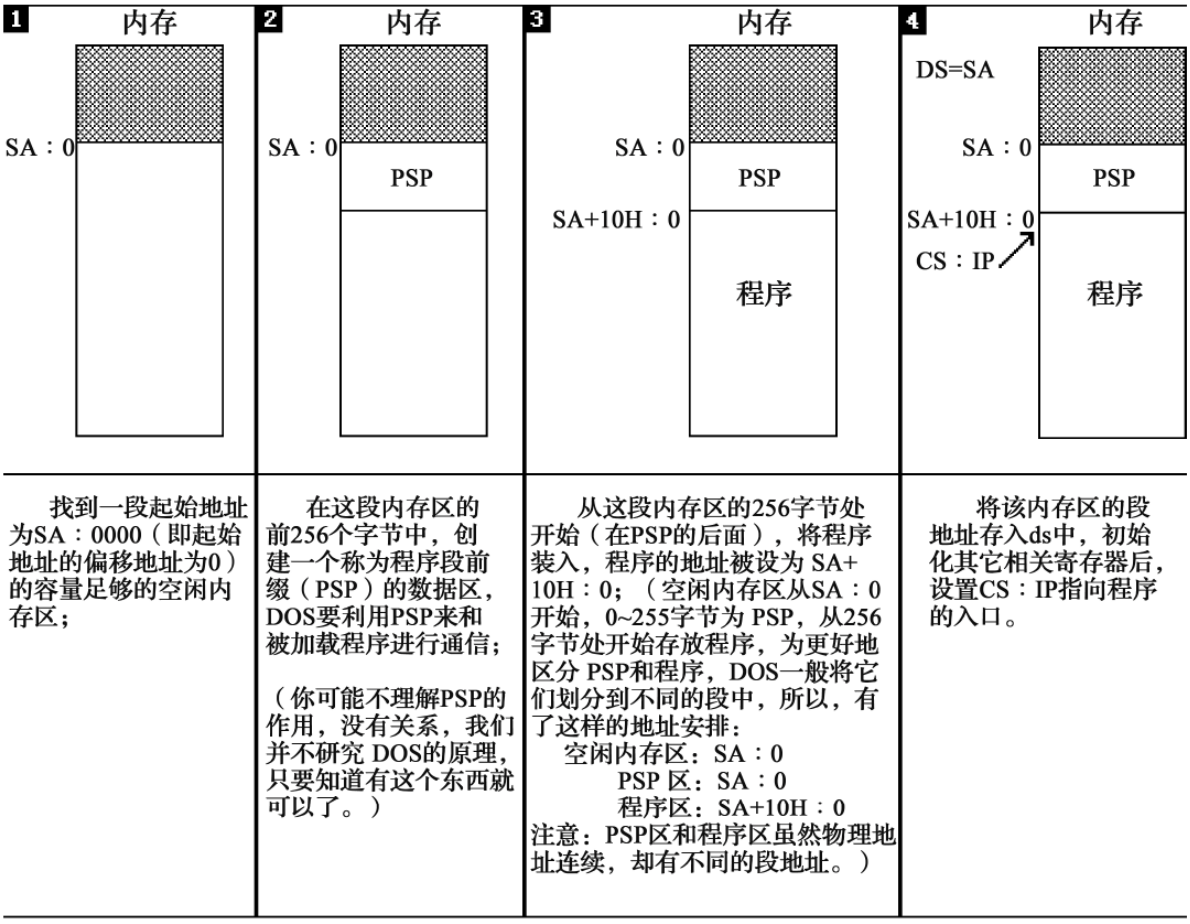


图 4.20　EXE 文件中程序的加载过程

注意，有一步称为重定位的工作在图 4.20 中没有讲解，因为这个问题和操作系统的关系较大，我们不作讨论。

那么，我们的程序被装入内存的什么地方？我们如何得知？从图 4.20 中我们知道以下的信息。

(1) 程序加载后，ds 中存放着程序所在内存区的段地址，这个内存区的偏移地址为 0，则程序所在的内存区的地址为 ds:0；

(2) 这个内存区的前 256 个字节中存放的是 PSP，DOS 用来和程序进行通信。从 256 字节处向后的空间存放的是程序。

所以，从 ds 中可以得到 PSP 的段地址 SA，PSP 的偏移地址为 0，则物理地址为 SA×16+0。

因为 PSP 占 256(100H)字节，所以程序的物理地址是：

SA×16+0+256 = SA×16+16×16+0 = (SA+16)×16+0

可用段地址和偏移地址表示为：SA+10H:0。

现在，我们看一下图 4.19 中 DS 的值，DS=129E，则 PSP 的地址为 129E:0，程序的地址为 12AE:0(即 129E+10:0)。

图 4.19 中，CS=12AE，IP=0000，CS:IP 指向程序的第一条指令。注意，源程序中的指令是 mov ax,0123H，在 Debug 中记为 mov ax,0123，这是因为 Debug 默认所有数据都用十六进制表示。

可以用 U 命令看一下其他指令，如图 4.21 所示。

```
C:\masm>debug 1.exe
-r
AX=0000  BX=0000  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=129E  ES=129E  SS=12AE  CS=12AE  IP=0000   NV UP EI PL NZ NA PO NC
12AE:0000 B82301        MOV     AX,0123
-u
12AE:0000 B82301        MOV     AX,0123
12AE:0003 BB5604        MOV     BX,0456
12AE:0006 03C3          ADD     AX,BX
12AE:0008 03C0          ADD     AX,AX
12AE:000A B8004C        MOV     AX,4C00
12AE:000D CD21          INT     21
12AE:000F 83E201        AND     DX,+01
12AE:0012 BA85E2        MOV     DX,E285
12AE:0015 2E            CS:
12AE:0016 A385E2        MOV     [E285],AX
12AE:0019 E9A5FC        JMP     FCC1
12AE:001C 803EE70400    CMP     BYTE PTR [04E7],00
-
```

图 4.21　1.exe 中程序的全部内容

可以看到，从 12AE:0000~12AE:000E 都是程序的机器码。

现在，我们可以开始跟踪了，用 T 命令单步执行程序中的每一条指令，并观察每条指令的执行结果，到了 int 21，我们要用 P 命令执行，如图 4.22 所示。

```
AX=0AF2  BX=0456  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=129E  ES=129E  SS=12AE  CS=12AE  IP=000A   NV UP EI PL NZ AC PO NC
12AE:000A B8004C        MOV     AX,4C00
-t

AX=4C00  BX=0456  CX=000F  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=129E  ES=129E  SS=12AE  CS=12AE  IP=000D   NV UP EI PL NZ AC PO NC
12AE:000D CD21          INT     21
-p

Program terminated normally
-
```

图 4.22　程序返回

图 4.22 中，int 21 执行后，显示出“Program terminated normally”，返回到 Debug 中。表示程序正常结束。注意，要使用 P 命令执行 int 21。这里不必考虑是为什么，只要记住这一点就可以了。

需要注意的是，在 DOS 中运行程序时，是 command 将程序加载入内存，所以程序运行结束后返回到 command 中，而在这里是 Debug 将程序加载入内存，所以程序运行结束后要返回到 Debug 中。

使用 Q 命令退出 Debug，将返回到 command 中，因为 Debug 是由 command 加载运行的。在 DOS 中用“debug 1.exe”运行 Debug 对 1.exe 进行跟踪时，程序加载的顺序是：command 加载 Debug，Debug 加载 1.exe。返回的顺序是：从 1.exe 中的程序返回到 Debug，从 Debug 返回到 command。

实验 3　编程、编译、连接、跟踪

(1)　将下面的程序保存为 t1.asm 文件，将其生成可执行文件 t1.exe。

```
assume cs:codesg

codesg segment

    mov ax,2000H
    mov ss,ax
    mov sp,0
    add sp,10
    pop ax
    pop bx
    push ax
    push bx
    pop ax
    pop bx

    mov ax,4c00H
    int 21H

codesg ends

end
```

(2)　用 Debug 跟踪 t1.exe 的执行过程，写出每一步执行后，相关寄存器中的内容和栈顶的内容。

(3)　PSP 的头两个字节是 CD 20，用 Debug 加载 t1.exe，查看 PSP 的内容。

注意，一定要做完这个实验才能进行下面的课程。

第 5 章　[BX]和 loop 指令

1. [bx]和内存单元的描述

[bx]是什么呢？和[0]有些类似，[0]表示内存单元，它的偏移地址是 0。比如在下面的指令中(在 Debug 中使用)：

```
mov ax,[0]
```

将一个内存单元的内容送入 ax，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址为 0，段地址在 ds 中。

```
mov al,[0]
```

将一个内存单元的内容送入 al，这个内存单元的长度为 1 字节(字节单元)，存放一个字节，偏移地址为 0，段地址在 ds 中。

要完整地描述一个内存单元，需要两种信息：①内存单元的地址；②内存单元的长度(类型)。

用[0]表示一个内存单元时，0 表示单元的偏移地址，段地址默认在 ds 中，单元的长度(类型)可以由具体指令中的其他操作对象(比如说寄存器)指出。

[bx]同样也表示一个内存单元，它的偏移地址在 bx 中，比如下面的指令：

```
mov ax,[bx]
```

将一个内存单元的内容送入 ax，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址在 bx 中，段地址在 ds 中。

```
mov al,[bx]
```

将一个内存单元的内容送入 al，这个内存单元的长度为 1 字节(字节单元)，存放一个字节，偏移地址在 bx 中，段地址在 ds 中。

2. loop

英文单词“loop”有循环的含义，显然这个指令和循环有关。

我们在这一章，讲解[bx]和 loop 指令的应用、意义和相关的内容。

3. 我们定义的描述性的符号：“()”

为了描述上的简洁，在以后的课程中，我们将使用一个描述性的符号“( )”来表示一个寄存器或一个内存单元中的内容。比如：

(ax)表示 ax 中的内容、(al)表示 al 中的内容；

(20000H)表示内存 20000H 单元的内容(()中的内存单元的地址为物理地址)；

((ds)*16+(bx))表示：

ds 中的内容为 ADR1，bx 中的内容为 ADR2，内存 ADR1×16+ADR2 单元的内容。

也可以理解为：ds 中的 ADR1 作为段地址，bx 中的 ADR2 作为偏移地址，内存 ADR1:ADR2 单元的内容。

注意，“( )”中的元素可以有 3 种类型：①寄存器名；②段寄存器名；③内存单元的物理地址(一个 20 位数据)。比如：

(ax)、(ds)、(al)、(cx)、(20000H)、((ds)*16+(bx))等是正确的用法；
(2000:0)、((ds):1000H)等是不正确的用法。

我们看一下(X)的应用，比如，

(1) ax 中的内容为 0010H，可以这样来描述：(ax)=0010H；
(2) 2000:1000 处的内容为 0010H，可以这样来描述：(21000H)=0010H；
(3) 对于 mov ax,[2]的功能，可以这样来描述：(ax)=((ds)*16+2)；
(4) 对于 mov [2],ax 的功能，可以这样来描述：((ds)*16+2)=(ax)；
(5) 对于 add ax,2 的功能，可以这样来描述：(ax)=(ax)+2；
(6) 对于 add ax,bx 的功能，可以这样来描述：(ax)=(ax)+(bx)；
(7) 对于 push ax 的功能，可以这样来描述：
(sp)=(sp)−2
((ss)*16+(sp))=(ax)
(8) 对于 pop ax 的功能，可以这样来描述：
(ax)=((ss)*16+(sp))
(sp)=(sp)+2

“(X)”所表示的数据有两种类型：①字节；②字。是哪种类型由寄存器名或具体的运算决定，比如：

(al)、(bl)、(cl)等得到的数据为字节型；(ds)、(ax)、(bx)等得到的数据为字型。

(al)=(20000H)，则(20000H)得到的数据为字节型；(ax)=(20000H)，则(20000H)得到的数据为字型。

4. 约定符号 idata 表示常量

我们在 Debug 中写过类似的指令：mov ax,[0]，表示将 ds:0 处的数据送入 ax 中。指令中，在“[…]”里用一个常量 0 表示内存单元的偏移地址。以后，我们用 idata 表示常量。比如：

mov ax,[idata] 就代表 mov ax,[1]、mov ax,[2]、mov ax,[3]等。
mov bx,idata 就代表 mov bx,1、mov bx,2、mov bx,3 等。

mov ds,idata　就代表 mov ds,1、mov ds,2 等，它们都是非法指令。

5.1　[BX]

看一看下面指令的功能。

```
mov ax,[bx]
```

功能：bx 中存放的数据作为一个偏移地址 EA，段地址 SA 默认在 ds 中，将 SA:EA 处的数据送入 ax 中。即：(ax)=((ds)*16+(bx))。

```
mov [bx],ax
```

功能：bx 中存放的数据作为一个偏移地址 EA，段地址 SA 默认在 ds 中，将 ax 中的数据送入内存 SA:EA 处。即：((ds)*16+(bx))=(ax)。

问题 5.1

程序和内存中的情况如图 5.1 所示，写出程序执行后，21000H~21007H 单元中的内容。

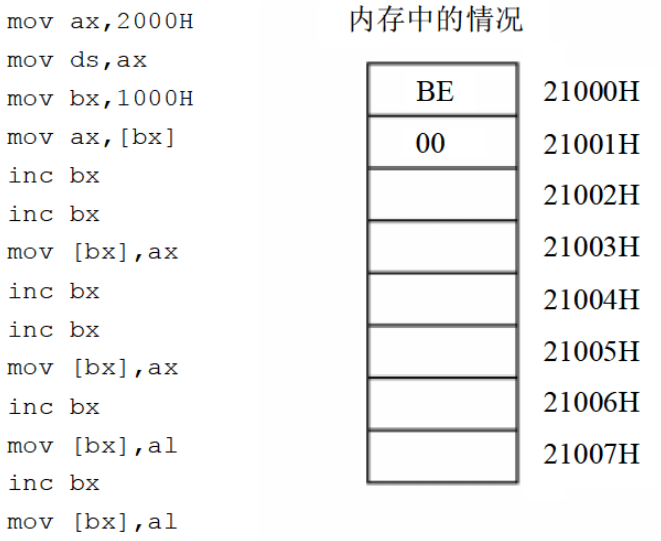


图 5.1　问题 5.1 程序和内存情况

思考后看分析。

注意，inc bx 的含义是 bx 中的内容加 1，比如下面两条指令：

```
mov bx,1
inc bx
```

执行后，bx=2。

分析：

(1) 先看一下程序的前 3 条指令：

```
mov ax,2000H
mov ds,ax
mov bx,1000H
```

这 3 条指令执行后，ds=2000H，bx=1000H。

(2) 接下来，第 4 条指令：

```
mov ax,[bx]
```

指令执行前：ds=2000H，bx=1000H，则 mov ax,[bx]将把内存 2000:1000 处的字型数据送入 ax 中。该指令执行后，ax=00beH。

(3) 接下来，第 5、6 条指令：

```
inc bx
inc bx
```

这两条指令执行前 bx=1000H，执行后 bx=1002H。

(4) 接下来，第 7 条指令：

```
mov [bx],ax
```

指令执行前：ds=2000H，bx=1002H，则 mov [bx],ax 将把 ax 中的数据送入内存 2000:1002 处。指令执行后，2000:1002 单元的内容为 BE，2000:1003 单元的内容为 00。

(5) 接下来，第 8、9 条指令：

```
inc bx
inc bx
```

这两条指令执行前 bx=1002H，执行后 bx=1004H。

(6) 接下来，第 10 条指令：

```
mov [bx],ax
```

指令执行前：ds=2000H，bx=1004H，则 mov [bx],ax 将把 ax 中的数据送入内存 2000:1004 处。指令执行后，2000:1004 单元的内容为 BE，2000:1005 单元的内容为 00。

(7) 接下来，第 11 条指令：

```
inc bx
```

这条指令执行前 bx=1004H，执行后 bx=1005H。

(8)　接下来，第 12 条指令：

```
mov [bx],al
```

指令执行前：ds=2000H，bx=1005H，则 mov [bx],al 将把 al 中的数据送入内存 2000:1005 处。指令执行后，2000:1005 单元的内容为 BE。

(9)　接下来，第 13 条指令：

```
inc bx
```

这条指令执行前 bx=1005H，执行后 bx=1006H。

(10) 接下来，第 14 条指令：

```
mov [bx],al
```

指令执行前：ds=2000H，bx=1006H，则 mov [bx],al 将把 al 中的数据送入内存 2000:1006 处。指令执行后，2000:1006 单元的内容为 BE。

BE	21000H
00	21001H
BE	21002H
00	21003H
BE	21004H
BE	21005H
BE	21006H
	21007H

图 5.2　内存中的情况

程序执行后，内存中的情况如图 5.2 所示。

5.2　Loop 指令

loop 指令的格式是：loop 标号，CPU 执行 loop 指令的时候，要进行两步操作，①(cx)=(cx)-1；②判断 cx 中的值，不为零则转至标号处执行程序，如果为零则向下执行。

从上面的描述中，可以看到，cx 中的值影响着 loop 指令的执行结果。通常(注意，我们说的是通常)我们用 loop 指令来实现循环功能，cx 中存放循环次数。

这里讲解 loop 指令的功能，关于 loop 指令如何实现转至标号处的细节，将在后面的课程中讲解。下面我们通过一个程序来看一下 loop 指令的具体应用。

任务 1：编程计算 2^2，结果存在 ax 中。

分析：设(ax)=2，可计算(ax)=(ax)*2，最后(ax)中为 2^2 的值。N*2 可用 N+N 实现，程序如下。

```
assume cs:code
code segment
  mov ax,2
  add ax,ax

  mov ax,4c00h
```

```
  int 21h
code ends
end
```

任务 2：编程计算 2^3。

分析：2^3=2*2*2，若设(ax)=2，可计算(ax)=(ax)*2*2，最后(ax)中为 2^3 的值。N*2 可用 N+N 实现，程序如下。

```
assume cs:code
code segment
  mov ax,2
  add ax,ax
  add ax,ax

  mov ax,4c00h
  int 21h
code ends
end
```

任务 3：编程计算 2^12。

分析：2^12=2*2*2*2*2*2*2*2*2*2*2*2，若设(ax)=2，可计算(ax)=(ax)*2*2*2*2*2*2*2*2*2*2*2，最后(ax)中为 2^12 的值。N*2 可用 N+N 实现，程序如下。

```
assume cs:code
code segment
  mov ax,2
  ;做 11 次 add ax,ax
  mov ax,4c00h
  int 21h
code ends
end
```

可见，按照我们的算法，计算 2^12 需要 11 条重复的指令 add ax,ax。我们显然不希望这样来写程序，这里，可用 loop 来简化我们的程序。

程序 5.1

```
assume cs:code
code segment
    mov ax,2
    mov cx,11
s:  add ax,ax
    loop s
```

```
    mov ax,4c00h
    int 21h
code ends
end
```

下面分析一下程序 5.1。

(1)　标号

在汇编语言中，标号代表一个地址，程序 5.1 中有一个标号 s。它实际上标识了一个地址，这个地址处有一条指令：add ax,ax。

(2)　loop s

CPU 执行 loop s 的时候，要进行两步操作：

①　(cx)=(cx)−1；

②　判断 cx 中的值，不为 0 则转至标号 s 所标识的地址处执行(这里的指令是 add ax,ax)，如果为 0 则执行下一条指令(下一条指令是 mov ax,4c00h)。

(3)　以下 3 条指令

```
    mov cx,11
s:  add ax,ax
    loop s
```

执行 loop s 时，首先要将(cx)减 1，然后若(cx) 不为 0，则向前转至 s 处执行 add ax,ax。所以，可以利用 cx 来控制 add ax,ax 的执行次数。

下面我们详细分析一下这段程序的执行过程，从中体会如何用 cx 和 loop s 相配合实现循环功能。

(1)　执行 mov cx,11，设置(cx)=11；
(2)　执行 add ax,ax(第 1 次)；
(3)　执行 loop s 将(cx) 减 1，(cx)=10，(cx) 不为 0，所以转至 s 处；
(4)　执行 add ax,ax(第 2 次)；
(5)　执行 loop s 将(cx) 减 1，(cx)=9，(cx) 不为 0，所以转至 s 处；
(6)　执行 add ax,ax(第 3 次)；
(7)　执行 loop s 将(cx) 减 1，(cx)=8，(cx) 不为 0，所以转至 s 处；
(8)　执行 add ax,ax(第 4 次)；
(9)　执行 loop s 将(cx) 减 1，(cx)=7，(cx) 不为 0，所以转至 s 处；
(10) 执行 add ax,ax(第 5 次)；
(11) 执行 loop s 将(cx) 减 1，(cx)=6，(cx) 不为 0，所以转至 s 处；
(12) 执行 add ax,ax(第 6 次)；
(13) 执行 loop s 将(cx) 减 1，(cx)=5，(cx) 不为 0，所以转至 s 处；

(14) 执行 add ax,ax(第 7 次)；
(15) 执行 loop s 将(cx) 减 1，(cx)=4，(cx) 不为 0，所以转至 s 处；
(16) 执行 add ax,ax(第 8 次)；
(17) 执行 loop s 将(cx) 减 1，(cx)=3，(cx) 不为 0，所以转至 s 处；
(18) 执行 add ax,ax(第 9 次)；
(19) 执行 loop s 将(cx) 减 1，(cx)=2，(cx) 不为 0，所以转至 s 处；
(20) 执行 add ax,ax(第 10 次)；
(21) 执行 loop s 将(cx) 减 1，(cx)=1，(cx) 不为 0，所以转至 s 处；
(22) 执行 add ax,ax(第 11 次)；
(23) 执行 loop s 将(cx) 减 1，(cx)=0，(cx) 为 0，所以向下执行。**(结束循环)**

从上面的过程中，我们可以总结出用 cx 和 loop 指令相配合实现循环功能的 3 个要点：

(1) 在 cx 中存放循环次数；
(2) loop 指令中的标号所标识地址要在前面；
(3) 要循环执行的程序段，要写在标号和 loop 指令的中间。

用 cx 和 loop 指令相配合实现循环功能的程序框架如下。

```
    mov cx,循环次数
s:
    循环执行的程序段
    loop s
```

问题 5.2

编程，用加法计算 123*236，结果存在 ax 中。思考后看分析。

分析：

可用循环完成，将 123 加 236 次。可先设(ax)=0，然后循环做 236 次(ax)=(ax)+123。

程序如下。

程序5.2

```
assume cs:code
code segment
    mov ax,0
    mov cx,236
  s:add ax,123
    loop s
```

```
    mov ax,4c00h
    int 21h
code ends
end
```

问题 5.3

改进程序 5.2，提高 123*236 的计算速度。思考后看分析。

分析：

程序 5.2 做了 236 次加法，我们可以将 236 加 123 次。可先设(ax)=0，然后循环做 123 次(ax)=(ax)+236，这样可以用 123 次加法实现相同的功能。

5.3　在 Debug 中跟踪用 loop 指令实现的循环程序

考虑这样一个问题，计算 ffff:0006 单元中的数乘以 3，结果存储在 dx 中。

我们分析一下。

(1)　运算后的结果是否会超出 dx 所能存储的范围？

ffff:0006 单元中的数是一个字节型的数据，范围在 0~255 之间，则用它和 3 相乘结果不会大于 65535，可以在 dx 中存放下。

(2)　用循环累加来实现乘法，用哪个寄存器进行累加？

将 ffff:0006 单元中的数赋值给 ax，用 dx 进行累加。先设(dx)=0，然后做 3 次(dx)=(dx)+(ax)。

(3)　ffff:6 单元是一个字节单元，ax 是一个 16 位寄存器，数据的长度不一样，如何赋值？

注意，我们说的是“赋值”，就是说，让 ax 中的数据的值(数据的大小)和 ffff:0006 单元中的数据的值(数据的大小)相等。8 位数据 01H 和 16 位数据 0001H 的数据长度不一样，但它们的值是相等的。

那么我们如何赋值？设 ffff:0006 单元中的数据是 XXH，若要 ax 中的值和 ffff:0006 单元中的相等，ax 中的数据应为 00XXH。所以，若实现 ffff:0006 单元向 ax 赋值，应该令(ah)=0，(al)=(ffff6H)。

想清楚以上的 3 个问题之后，编写程序如下。

程序 5.3

```
assume cs:code
code segment
    mov ax,0ffffh
    mov ds,ax
    mov bx,6                ;以上，设置 ds:bx 指向 ffff:6
    mov al,[bx]
    mov ah,0                ;以上，设置(al)=((ds*16)+(bx)), (ah)=0
    mov dx,0                ;累加寄存器清 0
    mov cx,3                ;循环 3 次
  s:add dx,ax
    loop s                  ;以上累加计算(ax)*3
    mov ax,4c00h
    int 21h                 ;程序返回
code ends
end
```

注意程序中的第一条指令 mov ax,0ffffh。我们知道，大于 9FFFh 的十六进制数据 A000H、A001H…C000H、c001H…FFFEH、FFFFH 等，在书写的时候都是以字母开头的。而**在汇编源程序中，数据不能以字母开头**，所以要在前面加 0。比如，9138h 在汇编源程序中可以直接写为“9138h”，而 A000h 在汇编源程序中要写为“0A000h”。

下面我们对程序的执行过程进行跟踪。首先，将它编辑为源程序文件，文件名定为 p3.asm；对其进行编译连接后生成 p3.exe；然后再用 Debug 对 p3.exe 中的程序进行跟踪。

用 Debug 加载 p3.exe 后，用 r 命令查看寄存器中的内容，如图 5.3 所示。

```
C:\masm>debug p3.exe
-r
AX=0000  BX=0000  CX=001B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B2D  ES=0B2D  SS=0B3D  CS=0B3D  IP=0000   NV UP EI PL NZ NA PO NC
0B3D:0000 B8FFFF        MOV     AX,FFFF
-
```

图 5.3 用 r 命令查看寄存器

图 5.3 中(ds)=0B2DH，所以，程序在 0B3D:0 处(如果读者还不清楚这是为什么，可以复习 4.9 节的内容)。我们看一下，(cs)=0B3DH，(IP)=0，CS:IP 正指向程序的第一条指令。再用 u 命令看一下被 Debug 加载入内存的程序，如图 5.4 所示。

可以看到，从 0B3D:0000~0B3D:001A 是我们的程序，0B3D:0014 处是源程序中的指令 loop s，只是此处 loop s 中的标号 s 已经变为一个地址 0012h。如果在执行“loop 0012”时，cx 减 1 后不为 0，“loop 0012”就把 IP 设置为 0012h，从而使 CS:IP 指向 0B3D:0012 处的 add dx,ax，实现转跳。

```
-u 0B3D:0
0B3D:0000 B8FFFF        MOV     AX,FFFF
0B3D:0003 8ED8          MOV     DS,AX
0B3D:0005 BB0600        MOV     BX,0006
0B3D:0008 8A07          MOV     AL,[BX]
0B3D:000A B400          MOV     AH,00
0B3D:000C BA0000        MOV     DX,0000
0B3D:000F B90800        MOV     CX,0003
0B3D:0012 03D0          ADD     DX,AX
0B3D:0014 E2FC          LOOP    0012
0B3D:0016 B8004C        MOV     AX,4C00
0B3D:0019 CD21          INT     21
0B3D:001B E83E0D        CALL    0D5C
0B3D:001E 83C404        ADD     SP,+04
-
```

图 5.4　用 u 命令查看被 Debug 加载入内存的程序

我们开始跟踪，如图 5.5 所示。

```
-r
AX=FFFF  BX=0000  CX=001B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0000   NV UP EI PL NZ NA PO NC
0B3D:0000 B8FFFF        MOV     AX,FFFF
-t

AX=FFFF  BX=0000  CX=001B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0003   NV UP EI PL NZ NA PO NC
0B3D:0003 8ED8          MOV     DS,AX
-t

AX=FFFF  BX=0000  CX=001B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0005   NV UP EI PL NZ NA PO NC
0B3D:0005 BB0600        MOV     BX,0006
-t

AX=FFFF  BX=0006  CX=001B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0008   NV UP EI PL NZ NA PO NC
0B3D:0008 8A07          MOV     AL,[BX]                          DS:0006=32
-
```

图 5.5　ds:bx 指向 ffff:6 单元

图 5.5 中，前 3 条指令执行后，(ds)=ffffh，(bx)=6，ds:bx 指向 ffff:6 单元。Debug 显示出当前要执行的指令“mov al,[bx]”，因为是读取内存的指令，所以 Debug 将要访问的内存单元中的内容也显示出来，可以看到屏幕最右边显示的“ds:0006=32”，由此，我们可以方便地知道目标单元(ffff6)中的内容是 32h。

继续执行，如图 5.6 所示。

```
AX=FFFF  BX=0006  CX=001B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0008   NV UP EI PL NZ NA PO NC
0B3D:0008 8A07          MOV     AL,[BX]                          DS:0006=32
-t

AX=FF32  BX=0006  CX=001B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=000A   NV UP EI PL NZ NA PO NC
0B3D:000A B400          MOV     AH,00
-t

AX=0032  BX=0006  CX=001B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=000C   NV UP EI PL NZ NA PO NC
0B3D:000C BA0000        MOV     DX,0000
-
```

图 5.6　从 ffff:6 单元向 ax 赋值

图 5.6 中，这两条指令执行后，(ax)=0032h，完成了从 ffff:6 单元向 ax 的赋值。

继续，如图 5.7 所示。

```
AX=0032  BX=0006  CX=001B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=000C   NV UP EI PL NZ NA PO NC
0B3D:000C BA0000        MOV     DX,0000
-t

AX=0032  BX=0006  CX=001B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=000F   NV UP EI PL NZ NA PO NC
0B3D:000F B90800        MOV     CX,0003
-t

AX=0032  BX=0006  CX=0003  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0012   NV UP EI PL NZ NA PO NC
0B3D:0012 03D0          ADD     DX,AX
-
```

图 5.7　初始化累加寄存器和循环计数寄存器

图 5.7 中，这两条指令执行后，(dx)=0，完成对累加寄存器的初始化；(cx)=3，完成对循环计数寄存器的初始化。

下面，将开始循环程序段的执行。我们继续，如图 5.8 所示。

```
AX=0032  BX=0006  CX=0003  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0012   NV UP EI PL NZ NA PO NC
0B3D:0012 03D0          ADD     DX,AX
-t

AX=0032  BX=0006  CX=0003  DX=0032  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0014   NV UP EI PL NZ NA PO NC
0B3D:0014 E2FC          LOOP    0012
-t

AX=0032  BX=0006  CX=0002  DX=0032  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0012   NV UP EI PL NZ NA PO NC
0B3D:0012 03D0          ADD     DX,AX
-t
```

图 5.8　第一次循环

图 5.8 中，CPU 执行 0B3D:0012 处的指令“add dx,ax”后，(IP)=0014h，CS:IP 指向 0B3D:0014 处的指令“loop 0012”。CPU 执行“loop 0012”，第一步先将(cx)减 1，(cx)=2；第二步因(cx)不等于 0，将 IP 设为 0012h。指令“loop 0012”执行后，(IP)=0012h，CS:IP 再次指向 0B3D:0012 处的指令“add dx,ax”，这条指令将再次得到执行。注意，“loop 0012”执行后(cx)=2，也就是说，“loop 0012”还可以进行两次循环。

接着，将重复执行“add dx,ax”和“loop 0012”，直到(cx)=0 为止，如图 5.9 所示。

```
AX=0032  BX=0006  CX=0002  DX=0032  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0012   NV UP EI PL NZ NA PO NC
0B3D:0012 03D0          ADD     DX,AX
-t

AX=0032  BX=0006  CX=0002  DX=0064  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0014   NV UP EI PL NZ NA PO NC
0B3D:0014 E2FC          LOOP    0012
-t

AX=0032  BX=0006  CX=0001  DX=0064  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0012   NV UP EI PL NZ NA PO NC
0B3D:0012 03D0          ADD     DX,AX
-t

AX=0032  BX=0006  CX=0001  DX=0096  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0014   NV UP EI PL NZ NA PE NC
0B3D:0014 E2FC          LOOP    0012
-t

AX=0032  BX=0006  CX=0000  DX=0096  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0016   NV UP EI PL NZ NA PE NC
0B3D:0016 B8004C        MOV     AX,4C00
-
```

图 5.9　得到计算结果

注意图 5.9 中，最后一次执行“loop 0012”的结果。执行前(cx)=1，CPU 执行“loop 0012”，第一步，(cx)=(cx)-1，(cx)=0；第二步，因为(cx)=0，所以 loop 指令不转跳，(IP)=0016h，CPU 向下执行 0B3D:0016 处的指令“mov ax,4c00”。

在完成最后一次“add dx,ax”后，(dx)=96h，此时 dx 中为累加计算(ax)*3 的最后结果。

我们继续，将程序执行完，如图 5.10 所示。

```
AX=0032  BX=0006  CX=0000  DX=0096  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0016   NV UP EI PL NZ NA PE NC
0B3D:0016 B8004C        MOV     AX,4C00
-t

AX=4C00  BX=0006  CX=0000  DX=0096  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0019   NV UP EI PL NZ NA PE NC
0B3D:0019 CD21          INT     21
-p

Program terminated normally
-
```

图 5.10　程序返回

图 5.10 中，执行完最后两条指令后，程序返回到 Debug 中。注意“int 21”要用 p 命令执行。

上面，我们通过对一个循环程序的跟踪，更深入一步地讲解了 loop 指令实现循环的原理。下面，我们将程序 5.3 改一下，计算 ffff:0006 单元中的数乘以 123，结果存储在 dx 中。

这很容易完成，只要将循环的次数改为 123 就可以了。程序如下。

程序5.4

```
assume cs:code
code segment
    mov ax,0ffffh
    mov ds,ax
    mov bx,6                ;以上，设置 ds:bx 指向 ffff:6
    mov al,[bx]
    mov ah,0                ;以上，设置(al)=((ds*16)+(bx))，(ah)=0
    mov dx,0                ;累加寄存器清 0
    mov cx,123              ;循环 123 次
s:  add dx,ax
    loop s                  ;以上累加计算(ax)*123
    mov ax,4c00h            ;程序返回
    int 21h
```

```
code ends
end
```

我们用 Debug 对这个程序的循环程序段进行跟踪，现在有这样一个问题：前面的 7 条指令，即标号 s 前的指令，已经确定在逻辑上完全正确，我们不想再一步步地跟踪了，只想跟踪循环的过程。所以希望可以一次执行完标号 s 前的指令。可以用一个新的(对我们来说是新的，因为以前没用过)Debug 命令 g 来达到目的。

下面来实际操作一下，我们用程序 5.4 生成最终的可执行文件“c:\masm\p4.exe”，用 Debug 加载 p4.exe，然后看一下程序在内存中的情况，如图 5.11 所示。

图 5.11 中，循环程序段从 CS:0012 开始，CS:0012 前面的指令，我们不想再一步步地跟踪，希望能够一次执行完，然后从 CS:0012 处开始跟踪。可以这样来使用 g 命令，“g 0012”，它表示执行程序到当前代码段(段地址在 CS 中)的 0012h 处。也就是说“g 0012”将使 Debug 从当前的 CS:IP 指向的指令执行，一直到(IP)=0012h 为止。具体的情况如图 5.12 所示。

```
C:\masm>debug p4.exe
-r
AX=0000  BX=0000  CX=001B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B2D  ES=0B2D  SS=0B3D  CS=0B3D  IP=0000   NV UP EI PL NZ NA PO NC
0B3D:0000 B8FFFF        MOV     AX,FFFF
-u 0b3d:0
0B3D:0000 B8FFFF        MOV     AX,FFFF
0B3D:0003 8ED8          MOV     DS,AX
0B3D:0005 BB0600        MOV     BX,0006
0B3D:0008 8A07          MOV     AL,[BX]
0B3D:000A B400          MOV     AH,00
0B3D:000C BA0000        MOV     DX,0000
0B3D:000F B97B00        MOV     CX,007B
0B3D:0012 03D0          ADD     DX,AX
0B3D:0014 E2FC          LOOP    0012
0B3D:0016 B8004C        MOV     AX,4C00
0B3D:0019 CD21          INT     21
0B3D:001B E83E0D        CALL    0D5C
0B3D:001E 83C404        ADD     SP,+04
-_
```

图 5.11　程序在内存中的情况

```
-g 0012

AX=0032  BX=0006  CX=007B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0012   NV UP EI PL NZ NA PO NC
0B3D:0012 03D0          ADD     DX,AX
_
```

图 5.12　CS:0012 前的程序段被执行

图 5.12 中，Debug 执行“g 0012”后，CS:0012 前的程序段被执行，从各个相关的寄存器中的值，我们可以看出执行的结果。

下面我们对循环的过程进行跟踪，如图 5.13 所示。

图 5.13 中，我们跟踪了两次循环的过程。其实，通过这两次循环过程，已经可以确定循环程序段在逻辑上是正确的。我们不想再继续一步步地观察循环的过程了，怎样让程序向下执行呢？继续像从前那样使用 t 命令？显然这是不可行的，因为还要进行 121((cx)=79h)次循环，如果像前两次那样使用 t 命令，我们得使用 121*2=242 次 t 命令才

能从循环中出来。

```
AX=0032  BX=0006  CX=007B  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0012   NV UP EI PL NZ NA PO NC
0B3D:0012 03D0          ADD     DX,AX
-t

AX=0032  BX=0006  CX=007B  DX=0032  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0014   NV UP EI PL NZ NA PO NC
0B3D:0014 E2FC          LOOP    0012
-t

AX=0032  BX=0006  CX=007A  DX=0032  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0012   NV UP EI PL NZ NA PO NC
0B3D:0012 03D0          ADD     DX,AX
-t

AX=0032  BX=0006  CX=007A  DX=0064  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0014   NV UP EI PL NZ NA PO NC
0B3D:0014 E2FC          LOOP    0012
-t

AX=0032  BX=0006  CX=0079  DX=0064  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0012   NV UP EI PL NZ NA PO NC
0B3D:0012 03D0          ADD     DX,AX
-
```

图 5.13　两次循环的过程

这里的问题是，我们希望将循环一次执行完。可以使用 p 命令来达到目的。再次遇到 loop 指令时，使用 p 命令来执行，Debug 就会自动重复执行循环中的指令，直到(cx)=0 为止。具体情况如图 5.14 所示。

```
AX=0032  BX=0006  CX=0079  DX=0064  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0012   NV UP EI PL NZ NA PO NC
0B3D:0012 03D0          ADD     DX,AX
-t

AX=0032  BX=0006  CX=0079  DX=0096  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0014   NV UP EI PL NZ NA PE NC
0B3D:0014 E2FC          LOOP    0012
-p

AX=0032  BX=0006  CX=0000  DX=1806  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0016   NV UP EI PL NZ NA PE NC
0B3D:0016 B8004C        MOV     AX,4C00
-
```

图 5.14　用 p 命令执行 loop 指令

图 5.14 中，在遇到“loop 0012”时，用 p 命令执行，Debug 自动重复执行“loop 0012”和“add dx,ax”两条指令，直到(cx)=0。最后一次执行“loop 0012”后，(cx)=0，(IP)=0016h，当前指令为 CS:0016 处的“mov ax,4c00”。

当然，也可以用 g 命令来达到目的，可以用“g 0016”直接执行到 CS:0016 处。具体情况如图 5.15 所示。

```
AX=0032  BX=0006  CX=0079  DX=0064  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0012   NV UP EI PL NZ NA PO NC
0B3D:0012 03D0          ADD     DX,AX
-g 0016

AX=0032  BX=0006  CX=0000  DX=1806  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0016   NV UP EI PL NZ NA PE NC
0B3D:0016 B8004C        MOV     AX,4C00
-
```

图 5.15　用 g 命令执行

5.4 Debug 和汇编编译器 masm 对指令的不同处理

本节知识点为下面课程的顺利进行提供一点预备知识。

我们在 Debug 中写过类似的指令：

```
mov ax,[0]
```

表示将 ds:0 处的数据送入 ax 中。

但是在汇编源程序中，指令“mov ax,[0]”被编译器当作指令“mov ax,0”处理。

下面通过具体的例子来看一下 Debug 和汇编编译器 masm 对形如“mov ax,[0]”这类指令的不同处理。

任务：将内存 2000:0、2000:1、2000:2、2000:3 单元中的数据送入 al、bl、cl、dl 中。

(1) 在 Debug 中编程实现：

```
mov ax,2000
mov ds,ax
mov al,[0]
mov bl,[1]
mov cl,[2]
mov dl,[3]
```

(2) 汇编源程序实现：

```
assume cs:code
code segment

 mov ax,2000h
 mov ds,ax
 mov al,[0]
 mov bl,[1]
 mov cl,[2]
 mov dl,[3]

 mov ax,4c00h
 int 21h

code ends
end
```

我们看一下两种实现的实际实施情况：

(1) Debug 中的情况如图 5.16 所示。

```
C:\masm>debug
-a
0AE8:0100 mov ax,2000
0AE8:0103 mov ds,ax
0AE8:0105 mov al,[0]
0AE8:0108 mov bl,[1]
0AE8:010C mov cl,[2]
0AE8:0110 mov dl,[3]
0AE8:0114
-u
0AE8:0100 B80020        MOV     AX,2000
0AE8:0103 8ED8          MOV     DS,AX
0AE8:0105 A00000        MOV     AL,[0000]
0AE8:0108 8A1E0100      MOV     BL,[0001]
0AE8:010C 8A0E0200      MOV     CL,[0002]
0AE8:0110 8A160300      MOV     DL,[0003]
```

图 5.16　Debug 对“mov al,[0]”等指令的解释

(2) 将汇编源程序存储为 compare.asm，用 masm、link 生成 compare.exe，用 Debug 加载 compare.exe，如图 5.17 所示。

```
C:\masm>debug compare.exe
-r
AX=0000  BX=0000  CX=0012  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B2D  ES=0B2D  SS=0B3D  CS=0B3D  IP=0000   NV UP EI PL NZ NA PO NC
0B3D:0000 B80020        MOV     AX,2000
-u 0b3d:0000
0B3D:0000 B80020        MOV     AX,2000
0B3D:0003 8ED8          MOV     DS,AX
0B3D:0005 B000          MOV     AL,00
0B3D:0007 B301          MOV     BL,01
0B3D:0009 B102          MOV     CL,02
0B3D:000B B203          MOV     DL,03
0B3D:000D B8004C        MOV     AX,4C00
0B3D:0010 CD21          INT     21
```

图 5.17　masm 对“mov al,[0]”等指令的解释

从图 5.16、图 5.17 中我们可以明显地看出，Debug 和编译器 masm 对形如“mov ax,[0]”这类指令在解释上的不同。我们在 Debug 中和源程序中写入同样形式的指令：“mov al,[0]”、“mov bl,[1]”、“mov cl,[2]”、“mov dl,[3]”，但 Debug 和编译器对这些指令中的“[idata]”却有不同的解释。Debug 将它解释为“[idata]”是一个内存单元，“idata”是内存单元的偏移地址；而编译器将“[idata]”解释为“idata”。

那么我们如何在源程序中实现将内存 2000:0、2000:1、2000:2、2000:3 单元中的数据送入 al、bl、cl、dl 中呢？

目前的方法是，可将偏移地址送入 bx 寄存器中，用[bx]的方式来访问内存单元。比如我们可以这样访问 2000:0 单元：

```
mov ax,2000h
mov ds,ax           ;段地址 2000h 送入 ds
mov bx,0            ;偏移地址 0 送入 bx
mov al,[bx]         ;ds:bx 单元中的数据送入 al
```

这样做是可以，可是比较麻烦，我们要用 bx 来间接地给出内存单元的偏移地址。我们还是希望能够像在 Debug 中那样，在“[ ]”中直接给出内存单元的偏移地址。这样做，在汇编源程序中也是可以的，只不过，要在“[ ]”的前面显式地给出段地址所在的段寄存器。比如我们可以这样访问 2000:0 单元：

```
mov ax,2000h
mov ds,ax
mov al,ds:[0]
```

比较一下汇编源程序中以下指令的含义。

“mov al,[0]”，含义：(al)=0，将常量 0 送入 al 中(与 mov al,0 含义相同)；
“mov al,ds:[0]”，含义：(al)=((ds)*16+0)，将内存单元中的数据送入 al 中；
“mov al,[bx]”，含义：(al)=((ds)*16+(bx))，将内存单元中的数据送入 al 中；
“mov al,ds:[bx]”，含义：与“mov al,[bx]”相同。

从上面的比较中可以看出：

(1) 在汇编源程序中，如果用指令访问一个内存单元，则在指令中必须用“[…]”来表示内存单元，如果在“[]”里用一个常量 idata 直接给出内存单元的偏移地址，就要在“[]”的前面显式地给出段地址所在的段寄存器。比如

```
mov al,ds:[0]
```

如果没有在“[]”的前面显式地给出段地址所在的段寄存器，比如

```
mov al,[0]
```

那么，编译器 masm 将把指令中的“[idata]”解释为“idata”。

(2) 如果在“[]”里用寄存器，比如 bx，间接给出内存单元的偏移地址，则段地址默认在 ds 中。当然，也可以显式地给出段地址所在的段寄存器。

5.5 loop 和[bx]的联合应用

考虑这样一个问题，计算 ffff:0~ffff:b 单元中的数据的和，结果存储在 dx 中。

我们还是先分析一下。

(1) 运算后的结果是否会超出 dx 所能存储的范围？

ffff:0~ffff:b 内存单元中的数据是字节型数据，范围在 0~255 之间，12 个这样的数据相加，结果不会大于 65535，可以在 dx 中存放下。

(2) 我们能否将 ffff:0~ffff:b 中的数据直接累加到 dx 中？

当然不行，因为 ffff:0~ffff:b 中的数据是 8 位的，不能直接加到 16 位寄存器 dx 中。

(3) 我们能否将 ffff:0~ffff:b 中的数据累加到 dl 中，并设置(dh)=0，从而实现累加到 dx 中？

这也不行，因为 dl 是 8 位寄存器，能容纳的数据的范围在 0~255 之间，ffff:0~ffff:b 中的数据也都是 8 位，如果仅向 dl 中累加 12 个 8 位数据，很有可能造成进位丢失。

(4) 我们到底怎样将 ffff:0~ffff:b 中的 8 位数据，累加到 16 位寄存器 dx 中？

从上面的分析中，可以看到，这里面有两个问题：类型的匹配和结果的不超界。具体地说，就是在做加法的时候，我们有两种方法：

① (dx)=(dx)+内存中的 8 位数据；
② (dl)=(dl)+内存中的 8 位数据。

第一种方法中的问题是两个运算对象的类型不匹配，第二种方法中的问题是结果有可能超界。

怎样解决这两个看似矛盾的问题？目前的方法(在后面的课程中我们还有别的方法)就是得用一个 16 位寄存器来做中介。将内存单元中的 8 位数据赋值到一个 16 位寄存器 ax 中，再将 ax 中的数据加到 dx 上，从而使两个运算对象的类型匹配并且结果不会超界。

想清楚以上的问题之后，编写程序如下。

程序5.5

```
assume cs:code

code segment

    mov ax,0ffffh
    mov ds,ax                ;设置(ds)=ffffh

    mov dx,0                 ;初始化累加寄存器，(dx)=0

    mov al,ds:[0]
    mov ah,0                 ;(ax)=((ds)*16+0)=(ffff0h)
    add dx,ax                ;向 dx 中加上 ffff:0 单元的数值

    mov al,ds:[1]
    mov ah,0                 ;(ax)=((ds)*16+1)=(ffff1h)
    add dx,ax                ;向 dx 中加上 ffff:1 单元的数值

    mov al,ds:[2]
    mov ah,0                 ;(ax)=((ds)*16+2)=(ffff2h)
    add dx,ax                ;向 dx 中加上 ffff:2 单元的数值

    mov al,ds:[3]
    mov ah,0                 ;(ax)=((ds)*16+3)=(ffff3h)
    add dx,ax                ;向 dx 中加上 ffff:3 单元的数值

    mov al,ds:[4]
    mov ah,0                 ;(ax)=((ds)*16+4)=(ffff4h)
    add dx,ax                ;向 dx 中加上 ffff:4 单元的数值

    mov al,ds:[5]
```

```
    mov ah,0                ;(ax)=((ds)*16+5)=(ffff5h)
    add dx,ax               ;向 dx 中加上 ffff:5 单元的数值

    mov al,ds:[6]
    mov ah,0                ;(ax)=((ds)*16+6)=(ffff6h)
    add dx,ax               ;向 dx 中加上 ffff:6 单元的数值

    mov al,ds:[7]
    mov ah,0                ;(ax)=((ds)*16+7)=(ffff7h)
    add dx,ax               ;向 dx 中加上 ffff:7 单元的数值

    mov al,ds:[8]
    mov ah,0                ;(ax)=((ds)*16+8)=(ffff8h)
    add dx,ax               ;向 dx 中加上 ffff:8 单元的数值

    mov al,ds:[9]
    mov ah,0                ;(ax)=((ds)*16+9)=(ffff9h)
    add dx,ax               ;向 dx 中加上 ffff:9 单元的数值

    mov al,ds:[0ah]
    mov ah,0                ;(ax)=((ds)*16+0ah)=(ffffah)
    add dx,ax               ;向 dx 中加上 ffff:a 单元的数值

    mov al,ds:[0bh]
    mov ah,0                ;(ax)=((ds)*16+0bh)=(ffffbh)
    add dx,ax               ;向 dx 中加上 ffff:b 单元的数值

    mov ax,4c00h            ;程序返回
    int 21h

code ends

end
```

上面的程序很简单，不用解释，你一看就懂。不过，在看懂了之后，你是否觉得这个程序编得有些问题？它似乎没有必要写这么长。这是累加 ffff:0~ffff:b 中的 12 个数据，如果要累加 0000:0~0000:7fff 中的 32K 个数据，按照这个程序的思路，将要写将近 10 万行程序(写一个简单的操作系统也就这个长度了)。

问题 5.4

应用 loop 指令，改进程序 5.5，使它的指令行数让人能够接受。

思考后看分析。

分析：

可以看出，在程序中，有 12 个相似的程序段，我们将它们一般化地描述为：

```
mov al,ds:[X]       ;ds:X 指向 ffff:X 单元
mov ah,0            ;(ax)=((ds)*16+(X))=(ffffXh)
add dx,ax           ;向 dx 中加上 ffff:X 单元的数值
```

我们可以看到，12 个相似的程序段中，只有 mov al,ds:[X]指令中的内存单元的偏移地址是不同的，其他都一样。而这些不同的偏移地址是在 0≤X≤bH 的范围内递增变化的。

我们可以用数学语言来描述这个累加的运算：$sum=\sum_{X=0}^{0bH}(ffffh*10h+X)$。

从程序实现上，我们将循环做。

```
(al)=((ds)*16+X)
(ah)=0
(dx)=(dx)+(ax)
```

一共循环 12 次，在循环开始前(ds)=ffffh，X=0，ds:X 指向第一个内存单元。每次循环后，X 递增，ds:X 指向下一个内存单元。

完整的算法描述如下。

初始化：

```
(ds)=ffffh
X=0
(dx)=0
```

循环 12 次：

```
(al)=((ds)*16+X)
(ah)=0
(dx)=(dx)+(ax)
X=X+1
```

可见，表示内存单元偏移地址的 X 应该是一个变量，因为在循环的过程中，偏移地址必须能够递增。这样，在指令中，我们就不能用常量来表示偏移地址。我们可以将偏移地址放到 bx 中，用[bx]的方式访问内存单元。在循环开始前设(bx)=0，每次循环，将 bx 中的内容加 1 即可。

最后一个问题是，如何实现循环 12 次？我们的 loop 指令该发挥作用了。

更详细的算法描述如下。

初始化：

```
(ds)=ffffh
(bx)=0
(dx)=0
(cx)=12
```

循环 12 次：

```
s:(al)=((ds)*16+(bx))
  (ah)=0
  (dx)=(dx)+(ax)
  (bx)=(bx)+1
  loop s
```

最后，我们写出程序。

程序 5.6

```
assume cs:code
code segment

    mov ax,0ffffh
    mov ds,ax
    mov bx,0                ;初始化 ds:bx 指向 ffff:0

    mov dx,0                ;初始化累加寄存器 dx，(dx)=0

    mov cx,12               ;初始化循环计数寄存器 cx，(cx)=12

s:  mov al,[bx]
    mov ah,0
    add dx,ax               ;间接向 dx 中加上((ds)*16+(bx))单元的数值
    inc bx                  ;ds:bx 指向下一个单元
    loop s

    mov ax,4c00h
    int 21h

code ends
end
```

在实际编程中，经常会遇到，用同一种方法处理地址连续的内存单元中的数据的问题。我们需要用循环来解决这类问题，同时我们必须能够在每次循环的时候按照同一种方法来改变要访问的内存单元的地址。这时，就不能用常量来给出内存单元的地址(比如，[0]、[1]、[2]中，0、1、2 是常量)，而应用变量。“mov al,[bx]”中的 bx 就可以看作一个代表内存单元地址的变量，我们可以不写新的指令，仅通过改变 bx 中的数值，改变指令访问的内存单元。

5.6 段 前 缀

指令“mov ax,[bx]”中，内存单元的偏移地址由 bx 给出，而段地址默认在 ds 中。我们可以在访问内存单元的指令中显式地给出内存单元的段地址所在的段寄存器。比如：

(1)　mov ax,ds:[bx]

将一个内存单元的内容送入 ax，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址在 bx 中，段地址在 ds 中。

(2)　mov ax,cs:[bx]

将一个内存单元的内容送入 ax，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址在 bx 中，段地址在 cs 中。

(3)　mov ax,ss:[bx]

将一个内存单元的内容送入 ax，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址在 bx 中，段地址在 ss 中。

(4)　mov ax,es:[bx]

将一个内存单元的内容送入 ax，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址在 bx 中，段地址在 es 中。

(5)　mov ax,ss:[0]

将一个内存单元的内容送入 ax，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址为 0，段地址在 ss 中。

(6)　mov ax,cs:[0]

将一个内存单元的内容送入 ax，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址为 0，段地址在 cs 中。

这些出现在访问内存单元的指令中，用于显式地指明内存单元的段地址的“ds:”“cs:”“ss:”“es:”，在汇编语言中称为**段前缀**。

5.7　一段安全的空间

在 8086 模式中，随意向一段内存空间写入内容是很危险的，因为这段空间中可能存放着重要的系统数据或代码。比如下面的指令：

```
mov ax,1000h
mov ds,ax
mov al,0
mov ds:[0],al
```

我们以前在 Debug 中，为了讲解上的方便，写过类似的指令。但这种做法是不合理的，因为之前我们并没有论证过 1000:0 中是否存放着重要的系统数据或代码。如果 1000:0 中存放着重要的系统数据或代码，“mov ds:[0],al”将其改写，将引发错误。

比如下面的程序。

程序 5.7

```
assume cs:code
code segment

 mov ax,0
 mov ds,ax
 mov ds:[26h],ax

 mov ax,4c00h
 int 21h

code ends
end
```

将源程序编辑为 p7.asm，编译、连接后生成 p7.exe，用 Debug 加载，跟踪它的运行，如图 5.18 所示。

```
C:\masm>debug p7.exe
-r
AX=0000  BX=0000  CX=000D  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B2D  ES=0B2D  SS=0B3D  CS=0B3D  IP=0000   NV UP EI PL NZ NA PO NC
0B3D:0000 B80000        MOV     AX,0000
-u 0b3d:0
0B3D:0000 B80000        MOV     AX,0000
0B3D:0003 8ED8          MOV     DS,AX
0B3D:0005 A32600        MOV     [0026],AX
0B3D:0008 B8004C        MOV     AX,4C00
0B3D:000B CD21          INT     21
```

图 5.18　用 Debug 加载程序 5.7

图 5.18 中，我们可以看到，源程序中的“mov ds:[26h],ax”被 masm 翻译为机器码“a3 26 00”，而 Debug 将这个机器码解释为“mov [0026],ax”。可见，汇编源程序中的汇编指令“mov ds:[26h],ax”和 Debug 中的汇编指令“mov [0026],ax”同义。

我们看一下“mov [0026],ax”的执行结果，如图 5.19 所示。

图 5.19 中，是在 windows 2000 的 DOS 方式中，在 Debug 里执行“mov [0026],ax”的结果。如果在实模式(即纯 DOS 方式)下执行程序 p7.exe，将会引起死机。产生这种结果的原因是 0:0026 处存放着重要的系统数据，而“mov [0026],ax”将其改写。

图 5.19　改写 0:0026 处存放的重要系统数据

可见，在不能确定一段内存空间中是否存放着重要的数据或代码的时候，不能随意向其中写入内容。

不要忘记，我们是在操作系统的环境中工作，操作系统管理所有的资源，也包括内存。如果我们需要向内存空间写入数据的话，要使用操作系统给我们分配的空间，而不应直接用地址任意指定内存单元，向里面写入。下一章我们会对“使用操作系统给我们分配的空间”有所认识。

但是，同样不能忘记，我们正在学习的是汇编语言，要通过它来获得底层的编程体验，理解计算机底层的基本工作机理。所以我们尽量直接对硬件编程，而不去理会操作系统。

我们似乎面临一种选择，是在操作系统中安全、规矩地编程，还是自由、直接地用汇编语言去操作真实的硬件，了解那些早已被层层系统软件掩盖的真相？在大部分的情况下，我们选择后者，除非我们就是在学习操作系统本身的内容。

注意，我们在纯 DOS 方式(实模式)下，可以不理会 DOS，直接用汇编语言去操作真实的硬件，因为运行在 CPU 实模式下的 DOS，没有能力对硬件系统进行全面、严格的管理。但在 Windows 2000、Unix 这些运行于 CPU 保护模式下的操作系统中，不理会操作系统，用汇编语言去操作真实的硬件，是根本不可能的。硬件已被这些操作系统利用 CPU 保护模式所提供的功能全面而严格地管理了。

在后面的课程中，我们需要直接向内存中写入内容，可我们又不希望发生图 5.19 中的那种情况。所以要找到一段安全的空间供我们使用。在一般的 PC 机中，DOS 方式下，DOS 和其他合法的程序一般都不会使用 0:200~0:2ff(00200h~002ffh)的 256 个字节的空间。所以，我们使用这段空间是安全的。不过为了谨慎起见，在进入 DOS 后，我们可以先用 Debug 查看一下，如果 0:200~0:2ff 单元的内容都是 0 的话，则证明 DOS 和其他合法的程序没有使用这里。

为什么 DOS 和其他合法的程序一般都不会使用 0:200~0:2ff 这段空间？我们将在以后的课程中讨论这个问题。

好了，我们总结一下：

(1) 我们需要直接向一段内存中写入内容；

(2) 这段内存空间不应存放系统或其他程序的数据或代码，否则写入操作很可能引发错误；

(3) DOS 方式下，一般情况，0:200~0:2ff 空间中没有系统或其他程序的数据或代码；

(4) 以后，我们需要直接向一段内存中写入内容时，就使用 0:200~0:2ff 这段空间。

5.8 段前缀的使用

我们考虑一个问题，将内存 ffff:0~ffff:b 单元中的数据复制到 0:200~0:20b 单元中。

分析一下。

(1) 0:200~0:20b 单元等同于 0020:0~0020:b 单元，它们描述的是同一段内存空间。

(2) 复制的过程应用循环实现，简要描述如下。

初始化：

```
X=0
```

循环 12 次：

```
将 ffff:X 单元中的数据送入 0020:X(需要用一个寄存器中转)
X=X+1
```

(3) 在循环中，源始单元 ffff:X 和目标单元 0020:X 的偏移地址 X 是变量。我们用 bx 来存放。

(4) 将 0:200~0:20b 用 0020:0~0020:b 描述，就是为了使目标单元的偏移地址和源始单元的偏移地址从同一数值 0 开始。

程序如下。

程序5.8

```
assume cs:code

code segment

    mov bx,0                ;(bx)=0，偏移地址从 0 开始
    mov cx,12               ;(cx)=12，循环 12 次

s:  mov ax,0ffffh
    mov ds,ax               ;(ds)=0ffffh
    mov dl,[bx]             ;(dl)=((ds)*16+(bx))，将 ffff:bx 中的数据送入 dl

    mov ax,0020h
    mov ds,ax               ;(ds)=0020h
    mov [bx],dl             ;((ds)*16+(bx))=(dl)，将中 dl 的数据送入 0020:bx

    inc bx                  ;(bx)=(bx)+1
    loop s

    mov ax,4c00h
    int 21h

code ends

end
```

因源始单元 ffff:X 和目标单元 0020:X 相距大于 64KB，在不同的 64KB 段里，程序 5.8 中，每次循环要设置两次 ds。这样做是正确的，但是效率不高。我们可以使用两个段寄存器分别存放源始单元 ffff:X 和目标单元 0020:X 的段地址，这样就可以省略循环中需要重复做 12 次的设置 ds 的程序段。

改进的程序如下。

程序5.9

```
assume cs:code

code segment

    mov ax,0ffffh
    mov ds,ax                ;(ds)=0ffffh

    mov ax,0020h
    mov es,ax                ;(es)=0020h

    mov bx,0                 ;(bx)=0，此时 ds:bx 指向 ffff:0，es:bx 指向 0020:0

    mov cx,12                ;(cx)=12，循环 12 次

s:  mov dl,[bx]              ;(dl)=((ds)*16+(bx))，将 ffff:bx 中的数据送入 dl
    mov es:[bx],dl           ;((es)*16+(bx))=(dl)，将 dl 中的数据送入 0020:bx
    inc bx                   ;(bx)=(bx)+1
    loop s

    mov ax,4c00h
    int 21h

code ends

end
```

程序 5.9 中，使用 es 存放目标空间 0020:0~0020:b 的段地址，用 ds 存放源始空间 ffff:0~ffff:b 的段地址。在访问内存单元的指令“mov es:[bx],al”中，显式地用段前缀“es:”给出单元的段地址，这样就不必在循环中重复设置 ds。

实验 4　[bx]和 loop 的使用

(1) 编程，向内存 0:200~0:23F 依次传送数据 0~63(3FH)。

(2) 编程，向内存 0:200~0:23F 依次传送数据 0~63(3FH)，程序中只能使用 9 条指令，9 条指令中包括“mov ax,4c00h”和“int 21h”。

(3) 下面的程序的功能是将“mov ax,4c00h”之前的指令复制到内存 0:200 处，补全程序。上机调试，跟踪运行结果。

```
assume cs:code
code segment
```

```
  mov ax,________
  mov ds,ax
  mov ax,0020h
  mov es,ax
  mov bx,0
  mov cx,________
s:mov al,[bx]
  mov es:[bx],al
  inc bx
  loop s
  mov ax,4c00h
  int 21h
code ends
end
```

提示：

(1) 复制的是什么？从哪里到哪里？

(2) 复制的是什么？有多少个字节？你如何知道要复制的字节的数量？

注意，一定要做完这个实验才能进行下面的课程。