

本书出版时间：2012年10月底11月初。任何疑问，请到新浪微博@均陵鼠侠，或者加群92033881。

x86 汇编语言： 从实模式到保护模式

李 忠 王晓波 余 洁 著

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内 容 简 介

每一种处理器都有它自己的机器指令集，而汇编语言的发明则是为了方便这些机器指令的记忆和书写。尽管汇编语言已经较少用于大型软件程序的开发，但从学习者的角度来看，要想真正理解计算机的工作原理，掌握它内部的运行机制，学习汇编语言是必不可少的。

这套图书分为两册，采用开源的 NASM 汇编语言编译器和 VirtualBox 虚拟机软件，以个人计算机广泛采用的 Intel 处理器为基础，详细讲解了 Intel 处理器的指令系统和工作模式，以大量的代码演示了 16 / 32 / 64 位软件的开发方法。上册集中介绍处理器的 16 位实模式和 32 位保护模式，以及基本的指令系统；下册侧重于介绍 64 位工作模式、多处理器管理、高速缓存控制、温度和电源管理、高级可编程中断控制器、多媒体支持等。

这是一本有趣的书，它没有把篇幅花在计算一些枯燥的数学题上。相反，它教你如何直接控制硬件，在不借助于 BIOS、DOS、Windows、Linux 或者任何其他软件支持的情况下显示字符、读取硬盘数据、控制其他硬件等。本书可作为大专院校相关专业学生和计算机编程爱好者的教程。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

/ 主编. —北京: 电子工业出版社, 2012.9

ISBN 978-7-121-0-0

I . ①汇… II . ①… III. ① IV. ①

中国版本图书馆 CIP 数据核字（2012）第 号

责任编辑：董亚峰

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1 092 1/16 印张： 字数： 千字

印 次：2012 年 9 月第 1 次印刷

定 价：00.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

前　　言

尽管汇编语言也是一种计算机语言，但却是与众不同的，与它的同类们格格不入。一方面，处理器的工作是执行指令，用它所做的一切都是执行指令并获得结果；另一方面，汇编语言为每一种指令提供了简单好记、易于书写的符号化表示形式。

一直以来，人们对于汇编语言的认识和评价可以分为两种，一种是觉得它非常简单，另一种是觉得它学习起来非常困难。

你认为我会赞同哪一种？说汇编语言难学，这没有道理。学习任何一门计算机语言，都需要一些数制和数制转换的知识，也需要大体上懂得计算机是怎么运作的。在这个前提下，汇编语言是最贴近硬件实体的，也是最自然和最朴素的。最朴素的东西反而最难掌握，这实在说不过去。因此，原因很可能出在我们的教科书上，那些一上来就搞一大堆寻址方式的书，往往以最快的速度打败了本来激情高昂的初学者。

但是，说汇编语言好学，也同样有些荒谬。据我的观察，很多人掌握了若干计算机指令，会编写一个从键盘输入数据，然后进行加减乘除或者归类排序的程序后，就认为自己掌握了汇编语言。还有，直到现在，我还经常在网上看到学生们使用 DOS 中断编写程序，他们讨论的也大多是实模式，而非 32 位或者 64 位保护模式。他们知道如何编译源程序，也知道在命令行输入文件名，程序就能运行了，使用一个中断，就能显示字符。至于这期间发生了什么，程序是如何加载到内存中的，又是怎么重定位的，似乎从来不关心汇编语言的事。这样做的结果，就是让人以为汇编语言不过如此，没有大用，而且非常枯燥。

很难说我已经掌握了汇编语言的要义。但至少我知道，尽管汇编语言不适合用来编写大型程序，但它对于理解计算机原理很有帮助，特别是处理器的工作原理和运行机制。就算是为了这个目的，也应该让汇编语言回归它的本位，那就是访问和控制硬件（包括处理器），而不仅仅是编写程序，输入几个数字，找出正数有几个、负数有几个，大于 30 的有几个。

事实上，汇编语言对学习和理解高级语言，比如 C 语言，也有极大的帮助。老教授琢磨了好几天，终于想到一个好的比喻来帮助学生理解什么是指针，实际上，这对于懂得汇编语言的学生来说，根本就不算个事儿，并因此能够使老教授省下时间来喝茶。

对于一个国家来说，不能没有人来研究基础学科，尽管它们不能直接产生效益；而对于一个人来说，也不能没有常识。尽管常识不能直接挣钱吃饭，但它影响谈吐，影响你的判断力和决断力，决定着你接受新事物和新知识的程度。相应地，汇编语言就是计算机语言里的常识和基础。

这是继《穿越计算机的迷雾》之后，我写的第二本书。这本书与上本书有两点不同，第一，上一本花了 4 年才完成，而这本只用了一年，速度之快，令我自己咂舌；第二，上本书属于科普性质，漫谈计算机原理，这本书就相对专业了。那些还想把我的书当小说看的人，这回要失望了。

很多人可能会问我，为什么要写这样一本书。我只能说，我第一次学汇编的经历实在是太深刻了。我第一次学汇编语言是在 1993 年，手中的教材不能说不好，但学习起来实在很吃力。要知道，在那个年代，没有网络，要买到好书，还得到大武汉。就这样，我抱着两本书，反反复复地看，直到半年之后才懂得汇编语言是个什么东西。后来，虽然有心写一本汇编语言的书，一本不一样的汇编语言书，但始终没有时间和精力。

时间过得真快，转眼 20 年过去了。猛回头，我发现同学们依然在走我的老路，他们所用

的教材，都还是我那个年代的，至少区别不大，都还在讲 8086 处理器的实模式。保护模式是从哪个处理器开始引入的？当然是 80286。它是哪个年代的产品？1982 年！可是，直到现在，市面上也找不到太多能够把保护模式讲得比较清楚的图书。

也许我应该做点什么。不，事实上，我已经做了，那就是你手中的这本图书。王晓波和湖北经济学院的余洁共同参与了本书的创作。

在计划写这本书的时候，我就给自己画了几条线。首先不能走老路，一上来就讲指令、寻址方式，采用任务驱动方式来写，每一章都要做点事情，最好是比较有趣，足够引起读者的事情。在解决问题的过程中，引入一个个的指令，并进行讲解。一句话，我希望是润物细无声式的。

其次，汇编语言和硬件并举，完全抛弃 BIOS 中断和 DOS 中断，直接访问硬件，发挥汇编语言的长处。这样，读者才会深刻体会到汇编的妙处。

这套图书主要讲述 16 位实模式、32 位保护模式和 Intel-64 架构。引入虚拟 8086 模式是为了兼容传统的 8086 程序，现在看来已经完全过时，不再进行讲述。至于增强的 32 位模式 IA-32e，读者可以在读完这本书之后自学，也予以省略。

书中配套的程序清单和源代码以及可能用到的程序软件，感兴趣的读者可到电子工业出版社华信教育资源网下载（待定）。

本书原来有 18 章，后来，考虑到实模式的内容过多，而去掉了一章。这一章的标题是《聆听数字的声音》，讲述如何通过直接访问和控制 Sound Blaster 16 声卡来播放声音，对此感兴趣的朋友可以在配书光盘中找到它。

特别感谢长春电视台的王志强台长和台长助理周武军，上本书《穿越计算机的迷雾》出版后，台长王志强亲自过问出版情况，并给予我特别的奖励，希望大家同样能从这本书中读到他们对我的关怀和鼓励；同时也要感谢我的母亲、我爱人和我的女儿，她们是我的精神支柱。好友王南洋、桑国伟、刘维钊、蒋胜友、邱海龙、万利等负责了本书的一部分校对工作；好友周卫平帮我验证配书代码是否能够在他的机器上正常工作；如果想调试本书中的程序，可以使用 bochs 软件，它的视频教程是由王南洋制作的，在这里向他们表示感谢。在阅读本书的过程中，如果有任何问题，可以按以下电子邮件地址给我写信：leechung@126.com；或者进入我的博客参与讨论。博客地址是

<http://blog.163.com/leechung@126>

目 录



第1部分 预备知识

第1章 十六进制计数法	3
1.1 二进制计数法回顾	3
1.1.1 关于二进制计数法	3
1.1.2 二进制到十进制的转换	3
1.1.3 十进制到二进制的转换	4
1.2 十六进制计数法	4
1.2.1 十六进制计数法的原理	4
1.2.2 十六进制到十进制的转换	5
1.2.3 十进制到十六进制的转换	6
1.3 为什么需要十六进制	6
本章习题	7

第2章 处理器、内存和指令	8
----------------------	---

2.1 最早的处理器	8
2.2 寄存器和算术逻辑部件	8
2.3 内存储器	10
2.4 指令和指令集	11
2.5 古老的 Intel 8086 处理器	13
2.5.1 8086 的通用寄存器	13
2.5.2 程序的重定位难题	14
2.5.3 内存分段机制	17
2.5.4 8086 的内存分段机制	18
本章习题	21

第3章 汇编语言和汇编软件	22
----------------------	----

3.1 汇编语言简介	22
3.2 NASM 编译器	24
3.2.1 从网上下载 NASM 安装程序	24
3.2.2 安装 NASM 编译器	25
3.2.3 下载配书源码和工具	26
3.2.4 用 Nasmide 体验代码的书写和编译过程	28
3.2.5 用 HexView 观察编译后的机器代码	29
本章习题	30

第 4 章 虚拟机的安装和使用 31

4.1 计算机的启动过程.....	31
4.1.1 如何将编译好的程序提交给处理器	31
4.1.2 计算机的加电和复位	31
4.1.3 基本输入输出系统	32
4.1.4 硬盘及其工作原理	33
4.1.5 一切从主引导扇区开始	35
4.2 创建和使用虚拟机.....	35
4.2.1 别害怕，虚拟机是软件	35
4.2.2 下载 Oracle VM VirtualBox	36
4.2.3 安装 Oracle VM VirtualBox	36
4.2.4 创建一台虚拟 PC	37
4.2.5 虚拟硬盘简介	42
4.2.6 练习使用 FixVhdWr 工具向虚拟硬盘写数据	43

第 2 部分 16 位处理器下的实模式**第 5 章 编写主引导扇区代码 49**

5.1 欢迎来到主引导扇区.....	49
5.2 注释	49
5.3 在屏幕上显示文字.....	50
5.3.1 显卡和显存	50
5.3.2 初始化段寄存器	52
5.3.3 显存的访问和 ASCII 代码	53
5.3.4 显示字符	55
5.4 显示标号的汇编地址.....	56
5.4.1 标号	56
5.4.2 如何显示十进制数字	60
5.4.3 在程序中声明并初始化数据	61
5.4.4 分解数的各个数位	61
5.4.5 显示分解出来的各个数位	65
5.5 使程序进入无限循环状态.....	66
5.6 完成并编译主引导扇区代码.....	67
5.6.1 主引导扇区有效标志	67
5.6.2 代码的保存和编译	68
5.7 加载和运行主引导扇区代码.....	68
5.7.1 把编译后的指令写入主引导扇区	68
5.7.2 启动虚拟机观察运行结果	70
5.7.3 程序的调试	70

本章习题	71
第6章 相同的功能，不同的代码.....	72
6.1 代码清单 6-1	72
6.2 跳过非指令的数据区	72
6.3 在数据声明中使用字面值	72
6.4 段地址的初始化	73
6.5 段之间的批量数据传送	74
6.6 使用循环分解数位	75
6.7 计算机中的负数	76
6.7.1 无符号数和有符号数	76
6.7.2 处理器视角中的数据类型	80
6.8 数位的显示	82
6.9 其他标志位和条件转移指令	83
6.9.1 奇偶标志位 PF	83
6.9.2 进位标志 CF	83
6.9.3 溢出标志 OF	84
6.9.4 现有指令对标志位的影响	84
6.9.5 条件转移指令	85
6.10 NASM 编译器的\$和\$\$标记	87
6.11 观察运行结果	87
本章习题	88
第7章 比高斯更快的计算.....	89
7.1 从 1 加到 100 的故事	89
7.2 代码清单 7-1	89
7.3 显示字符串	89
7.4 计算 1 到 100 的累加和	90
7.5 累加和各个数位的分解与显示	90
7.5.1 堆栈和堆栈段的初始化	90
7.5.2 分解各个数位并压栈	92
7.5.3 出栈并显示各个数位	94
7.5.4 进一步认识堆栈	95
7.6 程序的编译和运行	96
7.7 8086 处理器的寻址方式	96
7.7.1 寄存器寻址	96
7.7.2 立即寻址	97
7.7.3 内存寻址	97
本章习题	101

第 8 章 硬盘和显卡的访问与控制.....102

8.1 本章代码清单.....	102
8.1.1 本章意图	102
8.1.2 代码清单 8-1.....	103
8.2 用户程序的结构.....	103
8.2.1 分段、段的汇编地址和段内汇编地址.....	103
8.2.2 用户程序头部	106
8.3 加载程序（器）的工作流程.....	109
8.3.1 初始化和决定加载位置	109
8.3.2 准备加载用户程序	110
8.3.3 外围设备及其接口	111
8.3.4 I/O 端口和端口访问.....	112
8.3.5 通过硬盘控制器端口读扇区数据	114
8.3.6 过程调用	116
8.3.7 加载用户程序	121
8.3.8 用户程序重定位	122
8.3.9 将控制权交给用户程序	126
8.3.10 8086 处理器的无条件转移指令	126
8.4 用户程序的工作流程.....	128
8.4.1 初始化段寄存器和堆栈切换	128
8.4.2 调用字符串显示例程	129
8.4.3 过程的嵌套	130
8.4.4 屏幕光标控制	131
8.4.5 取当前光标位置	131
8.4.6 处理回车和换行字符	132
8.4.7 显示可打印字符	133
8.4.8 滚动屏幕内容	134
8.4.9 重置光标	134
8.4.10 切换到另一个代码段中执行	135
8.4.11 访问另一个数据段	135
8.5 编译和运行程序并观察结果.....	135
本章习题	136

第 9 章 中断和动态时钟显示.....137

9.1 外部硬件中断.....	137
9.1.1 非屏蔽中断	138
9.1.2 可屏蔽中断	138
9.1.3 实模式下的中断向量表	140
9.1.4 实时时钟、CMOS RAM 和 BCD 编码	141

9.1.5 代码清单 9-1	145
9.1.6 初始化 8259、RTC 和中断向量表	145
9.1.7 使处理器进入低功耗状态	147
9.1.8 实时时钟中断的处理过程	148
9.1.9 代码清单 9-1 的编译和运行	150
9.2 内部中断	150
9.3 软中断	151
9.3.1 常用的 BIOS 中断	151
9.3.2 代码清单 9-2	155
9.3.3 从键盘读字符并显示	155
9.3.4 代码清单 9-2 的编译和运行	155
本章习题	156

第3部分 32位保护模式

第 10 章 32 位 Intel 微处理器编程架构	159
10.1.2 基本的工作模式	162
10.1.3 线性地址	163
10.2 现代处理器的结构和特点	164
10.2.1 流水线	164
10.2.2 高速缓存	165
10.2.3 乱序执行	165
10.2.4 寄存器重命名	166
10.2.5 分支目标预测	167
10.3 32 位模式的指令系统	168
10.3.1 32 位处理器的寻址方式	168
10.3.2 操作数大小的指令前缀	169
10.3.3 一般指令的扩展	171
本章习题	174

第 11 章 进入保护模式

11.1 代码清单 11-1	175
11.2 全局描述符表	175
11.3 存储器的段描述符	177
11.4 安装存储器的段描述符并加载 GDTR	180
11.5 关于第 21 条地址线 A20 的问题	182
11.6 保护模式下的内存访问	184
11.7 清空流水线并串行化处理器	188
11.8 保护模式下的堆栈	189
11.8.1 关于堆栈段描述符中的界限值	189

11.8.2 检验 32 位下的堆栈操作	191
11.9 程序的编译和运行	191
本章习题	192
第 12 章 存储器的保护	193
12.1 代码清单 12-1	193
12.2 进入 32 位保护模式	193
12.2.1 话说 mov ds,ax 和 mov ds,eax	193
12.2.2 创建 GDT 并安装段描述符	194
12.3 修改段寄存器时的保护	196
12.4 地址变换时的保护	198
12.4.1 代码段执行时的保护	198
12.4.2 堆栈操作时的保护	199
12.4.3 数据访问时的保护	201
12.5 使用别名访问代码段对字符排序 (xchg)	202
12.6 程序的编译和运行	204
本章习题	204
第 13 章 程序的动态加载和执行	205
13.1 本章代码清单	205
13.2 内核的结构、功能和加载	206
13.2.1 内核的结构	206
13.2.2 内核的加载	207
13.2.3 安装内核的段描述符	209
13.3 在内核中执行	212
13.4 用户程序的加载和重定位	214
13.4.1 用户程序的结构	214
13.4.2 计算用户程序占用的扇区数	216
13.4.3 简单的动态内存分配	217
13.4.4 段的重定位和描述符的创建	218
13.4.5 重定位用户程序内的符号地址	221
13.5 执行用户程序	225
13.6 代码的编译、运行和调试	227
本章习题	228
第 14 章 任务和特权级保护	229
14.1 任务的隔离和特权级保护	229
14.1.1 任务、任务的 LDT 和 TSS	229
14.1.2 全局空间和局部空间	232
14.1.3 特权级保护概述	233

14.2 代码清单 14-1	240
14.3 内核程序的初始化	240
14.3.1 调用门	241
14.3.2 调用门的安装和测试	243
14.4 加载用户程序并创建任务	246
14.4.1 任务控制块和 TCB 链	246
14.4.2 使用堆栈传递过程参数	248
14.4.3 加载用户程序	250
14.4.4 创建局部描述符表	250
14.4.5 重定位 U-SALT 表	251
14.4.6 创建 0、1 和 2 特权级的堆栈	252
14.4.7 安装 LDT 描述符到 GDT 中	253
14.4.8 任务状态段 TSS 的格式	254
14.4.9 创建任务状态段 TSS	257
14.4.10 安装 TSS 描述符到 GDT 中	258
14.4.11 带参数的过程返回指令	258
14.5 用户程序的执行	260
14.5.1 通过调用门转移控制的完整过程	260
14.5.2 进入 3 特权级的用户程序的执行	263
14.5.3 检查调用者的请求特权级 RPL	265
本章习题	267
第 15 章 任务切换	268
15.1 本章代码清单	268
15.2 任务切换前的设置	268
15.3 任务切换的方法	270
15.4 用 call/jmp/iret 指令发起任务切换的实例	273
15.5 处理器在实施任务切换时的操作	277
15.6 程序的编译和运行	279
本章习题	280
第 16 章 分页机制和动态页面分配	281
16.1 分页机制概述	281
16.1.1 简单的分页模型	281
16.1.2 页目录、页表和页	286
16.1.3 地址变换的具体过程	288
16.2 本章代码清单	289
16.3 使内核在分页机制下工作	289
16.3.1 创建内核的页目录和页表	289
16.3.2 任务全局空间和局部空间的页面映射	294

16.4 创建内核任务	300
16.4.1 内核的虚拟内存分配	300
16.4.2 页面位映射串和空闲页的查找	301
16.4.3 创建页表并登记分配的页	303
16.4.4 创建内核任务的 TSS	304
16.5 用户任务的创建和切换	305
16.5.1 多段模型和段页式内存管理	305
16.5.2 平坦模型和用户程序的结构	307
16.5.3 用户任务的虚拟地址空间分配	308
16.5.4 用户程序的加载	309
16.5.5 段描述符的创建（平坦模型）	312
16.5.6 重定位 U-SALT 并复制页目录表	313
16.5.7 切换到用户任务执行	315
16.6 程序的编译和执行	317
本章习题	317
第 17 章 中断和异常的处理	318
17.1 中断和异常	318
17.1.1 中断和异常概述	318
17.1.2 中断描述符表、中断门和陷阱门	321
17.1.3 中断和异常处理程序的保护	323
17.1.4 中断任务	324
17.1.5 错误代码	325
17.2 本章代码清单	326
17.3 内核的加载和初始化	327
17.3.1 彻底终结多段模型	327
17.3.2 创建中断描述符表	330
17.3.3 用定时中断实施任务切换	331
17.3.4 8259A 芯片的初始化	337
17.3.5 平坦模型下的字符串显示例程	339
17.4 内核任务的创建	340
17.4.1 创建内核任务的 TCB	340
17.4.2 宏汇编技术	341
17.5 用户任务的创建	343
17.5.1 准备加载用户程序	343
17.5.2 转换后援缓冲器的刷新	344
17.5.3 用户任务的创建和初始化	346
17.6 程序的编译和执行	348
本章习题	348

第 1 部分

预备知识

第 1 章 十六进制计数法

1.1 二进制计数法回顾

1.1.1 关于二进制计数法

在《穿越计算机的迷雾》那本书里我们已经知道，计算机也是一台机器，唯一不同的地方在于它能计算数学题，且具有逻辑判断能力。

与此同时，我们也已经在那本书里学到，机器在做数学题的时候，也面临着一个如何表示数字的问题，比如你采用什么办法来将加数和被加数送到机器里。

同样是在那本书里，我们揭晓了答案，那就是用高、低两种电平的组合来表示数字。如图 1-1 所示，参与计算的数字通过电线送往计算机器，高电平被认为是“1”，低电平被认为是“0”，这样就形成了一个序列“11111010”，这就是一个二进制数，在数值上等于我们所熟知的二百五，换句话说，等于十进制数 250。

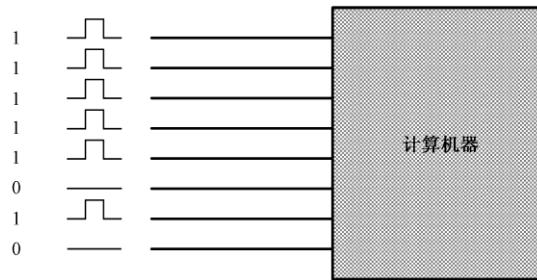


图 1-1 在计算机里，二进制数字对应着高低电平的组合

从数学的角度来看，二进制计数法是现代主流计算机的基础。一方面，它简化了硬件设计，因为它只有两个符号“0”和“1”，要得到它们，可以用最少的电路元件来接通或者关断电路就行了；另一方面，二进制数与我们熟悉的十进制数之间有着一对一的关系，任何一个十进制数都对应着一个二进制数，不管它有多大。比如，十进制数 5，它所对应的二进制数是 101，而十进制数 5785478965147 则对应着一长串“0”和“1”的组合，即 1010100001100001001011010010011110011011。

组成二进制数的每一个数位，称为一个比特（bit），而一个二进制数也可以看成是一个比特串。很明显，它的数值越大，这个比特串就越长，这是二进制计数法不好的一面。

1.1.2 二进制到十进制的转换

每一种计数法都有自己的符号（数符）。比如，十进制有 0、1、2、3、4、5、6、7、8、9 这十个符号；二进制呢，则只有 0、1 这两个符号。这些数字符号的个数称为基数。也就是说，

十进制有 10 个基数，而二进制只有两个。

二进制和十进制都是进位计数法。进位计数法的一个特点是，符号的值和它在这个数中所处的位置有关。比如十进制数 356，6 处在个位上，所以是“6 个”；5 处在十位上，所以是“50”；3 处在百位上，所以是“300”。即：

$$\text{百位 } 3、\text{十位 } 5、\text{个位 } 6 = 3 \times 10^2 + 5 \times 10^1 + 6 \times 10^0 = 356$$

这就是说，由于所处的位置不同，每个数位都有一个不同的放大倍数，这称为“权”。每个数位的权是这样计算的（这里仅讨论整数）：从右往左开始，以基数为底，指数从 0 开始递增的幂。正如上面的公式所清楚表明的那样，“6”在最右边，所以它的权是以 10 为底，指数为 0 的幂 10^0 ；而 3 呢，它的权则是以 10 为底，指数为 2 的幂 10^2 。

上面的算式是把十进制数“翻译”成十进制数。从十进制数又算回到十进制数，这看起来有些可笑，注意这个公式是可以推广的，可以用它来将二进制数转换成十进制数。

比如一个二进制数 10110001，它的基数是 2，所以要这样来计算与它等值的十进制数：

$$10110001B = 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 177D$$

在上面的公式里，10110001B 里的“B”表示这是一个二进制数，“D”则表示 177 是个十进制数。“B”和“D”分别是英语单词 Binary 和 Decimal 的头一个字母，这两个单词分别表示二进位和十进位的意思。

讲到这里，也请你算一算，二进制数 10000000 和 1101101100011011 分别等于十进制数的多少？

1.1.3 十进制到二进制的转换

为了将一个十进制数转换成二进制数，可以采用将它不停地除以二进制的基数 2，直到商为 0，然后将每一步得到的余数串起来即可。如图 1-2 所示，如果要将十进制数 26 转换成二进制数，那么可采用如下方法：

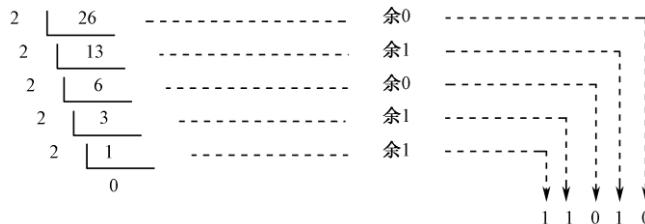


图 1-2 将十进制数 26 转换成二进制数

第 1 步，将 26 除以 2，商为 13，余数为 0；

第 2 步，用 13 除以 2，商为 6，余数为 1；

第 3 步，用 6 除以 2，商为 3，余数为 0；

第 4 步，用 3 除以 2，商为 1，余数为 1；

第 5 步，用 1 除以 2，商为 0，余数为 1，结束。

然后，从下往上，将每一步得到的余数串起来，从左往右书写，就是我们所要转换的二进制数。

1.2 十六进制计数法

1.2.1 十六进制计数法的原理

二进制数和计算机电路有着近乎直观的联系。电路的状态，可以用二进制数来直观地描述，而一个二进制数，也容易使我们仿佛观察到了每根电线上的电平变化。所以，我们才形象地说，二进制是计算机的官方语言。

即使是在平时的学习和研究中，使用二进制也是必需的。一个数字电路输入什么，输出什么，电路的状态变了，是哪一位发生了变化，研究这些，肯定要精确到每一个比特。这个时候，采用二进制是最直观的。

但是，二进制也有它的缺点。眼下看来，它最主要的缺点就是写起来太长，一点也不方便。为此，人们发明了十六进制计数法。至于为什么要发明另外一套计数方法，而不是依旧采用我们熟悉的十进制，下面就要为大家解释。

一旦知道二进制有两个数符“0”和“1”，十进制有十个数符“0”到“9”，那么我们就会很自然地认为十六进制一定有16个数符。

一点没错，完全正确。这16个数符分别是0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F。

你可能会觉得惊讶，字母怎么可以当做数字来使用？这样的话，那些熟悉的英语单词，像Face（脸）、Bad（坏的）、Bed（床）就都成了数。

这又有什么奇怪的？你觉得“0”、“5”、“9”是数字，而“A”、“B”不是数字，这是因为你已经从小习惯了这种做法。

对于自然数里的前10个，十进制和十六进制的表示方法是一致的。但是，9之后的数，两者表示方法就大相径庭了，如表1-1所示。

表1-1 部分十进制数和十六进制数对照表

十进制数	十六进制数	十进制数	十六进制数
0	0	17	11
1	1	18	12
2	2	19	13
3	3	20	14
4	4	21	15
5	5	22	16
6	6	23	17
7	7	24	18
8	8	25	19
9	9	26	1A
10	A	27	1B
11	B	28	1C
12	C	29	1D
13	D	30	1E
14	E	31	1F
15	F	32	20
16	10	33	21

很显然，一旦某个数位增加到9之后，下一次，它将变成A，而不是向前进位，因为这里是逢16才进位的。进位只发生在某个数位原先是F的情况下，比如1F，它加一后将会变成20。

1.2.2 十六进制到十进制的转换

要把一个十六进制数转换成我们熟悉的十进制数，可以采用和前面一样的方法。只不过，计算各个数位的权时，幂的底数是 16。比如将十六进制数 125 转换成十进制数的方法如下：

$$125H = 1 \times 16^2 + 2 \times 16^1 + 5 \times 16^0 = 293D$$

上式里，125 后面的“H”用于表明这是一个十六进制数，它是英语单词 Hexadecimal 的第一个字母，这个单词的意思是十六进制。

1.2.3 十进制到十六进制的转换

如图 1-3 所示，相应地，要把一个十进制数转换成十六进制数，则可以采取不停地除以 16 并取其余数的策略。

第 1 次，将 293 除以 16，商为 18，余 5；

第 2 次，用 18 除以 16，商为 1，余 2；

第 3 次，再用 1 除以 16，商为 0，余 1，结束。

然后，从下往上，将每次的余数 1、2、5 列出来，得到 125，这就是所要的结果。

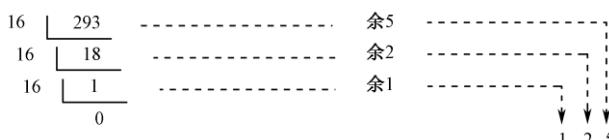


图 1-3 将十进制数 293 转换成十六进制数

1.3 为什么需要十六进制

为什么我们要发明十六进制计数法？为什么我们要学习它？

提出这样的问题，在我看来很有趣，也很有意义，但似乎从来没有人书上正面回答过。这样一来，可怜的学子们只能在掌握了十六进制若干年之后，在某一天里自己恍然大悟。

为了搞清楚这个问题，我们不妨来列张表（表 1-2），看看十进制数、二进制数和十六进制数之间，都有些什么有趣的规律和特点。

表 1-2 部分十进制数、二进制数和十六进制数对照表

十进制数	二进制数	十六进制数	十进制数	二进制数	十六进制数
0	0000	0	10	1010	A
1	0001	1	11	1011	B
2	0010	2	12	1100	C
3	0011	3	13	1101	D
4	0100	4	14	1110	E
5	0101	5	15	1111	F
6	0110	6	16	0001 0000	10
7	0111	7	17	0001 0001	11

8	1000	8	55	0011 0111	37
9	1001	9	195	1100 0011	C3

在上面这张表里（表 1-2），每一个二进制数在排版的时候，都经过了“艺术加工”，全都以 4 比特为一组的形式出现。不足 4 比特的，前面都额外加了“0”，比如 10，被写成 0010 的形式。就像十进制数一样，在一个二进制数的前面加多少个零，都不会改变它的值。

注意观察这张表并开动脑子，4 比特的二进制数，可以表示的数是 0000 到 1111，也就是十进制的 0~15，这正好对应于十六进制的 0~F。

在这个时候，如果将它们都各自加 1，那么，下一个二进制数是 0001 0000，与此同时，它对应的十六进制数则是 10，你会发现，它们有着如图 1-4 所示的奇妙对应关系。



图 1-4 十六进制的每一位与二进制数每 4 比特为一组的对应关系

再比如图 1-4 中的二进制数 1100 0011，它与等值的十六进制数 C3 也有着相同的对应关系。

也就是说，如果将一个二进制数从右往左，分成 4 比特为一组的形式，分别将每一组的值转换成十六进制数，就可以得到这个二进制数所对应的十六进制数。

这样一来，如果我们稍加努力，将 0~F 这 16 个数所对应的二进制数背熟，并能换算自如的话，那么，当我们看到一个十六进制数 3F8 时，我们就知道，因为 3 对应的二进制数为 0011，F 对应的二进制数是 1111，8 对应的二进制数是 1000，所以 3F8H=0011 1111 1000B。

同理，如果一个二进制数是 1101 0010 0101 0001，那么，将它们按 4 比特为一组，分别换算成十六进制数，就得到了 D251。

正如前面所说的，从事计算机的学习和研究（包括咱们马上就要进行的汇编语言程序设计），不可避免地要与二进制数打交道，而且有时还必须针对其中某些比特进行特殊处理。这个时候，如果想保留二进制数的直观性，同时还要求写起来简短，十六进制数是最好的选择。

本 章 习 题

1. 口算：

5H=____D	12D=____H	0FH=____D=____B
0CH=____D=____B	0AH=____D=____B	8D=____H=____B
0BH=____D=____B	0EH=____D=____B	10H=____D=____B

2. 口算：

10010B=____H	15H=____B	8FH=____B	200H=____B
11111111B=____H			

第 2 章 处理器、内存和指令

2.1 最早的处理器

1947 年，美国贝尔实验室的肖克利和同事们一起发明了晶体管。1958 年，也许是受够了在一大堆晶体管里连接那些杂乱无章的导线，另一个美国人杰克·基尔比发明了集成电路。接着，1971 年，在为日本人设计计算器芯片的过程中，受到启发的 Intel 公司生产了世界上第一个处理器 4004。

图 2-1 所示的是 4004 和它的设计者弗德里科·法金。

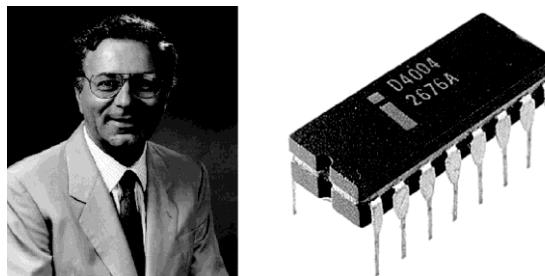


图 2-1 Intel 第一块处理器 4004 和它的设计者弗德里科·法金

今天我们可以知道，处理器（Processor）是一台电子计算机的核心，它会在振荡器脉冲的激励下，从内存中获取指令，并发起一系列由该指令所定义的操作。当这些操作结束后，它接着再取下一条指令。通常情况下，这个过程是连续不断、循环往复的。

2.2 寄存器和算术逻辑部件

为什么处理器能够自动计算，这个问题已经在我的上一本书《穿越计算机的迷雾》里讲过了，不过这些原理讲起来很费劲，花了整整一本书的篇幅。当然，如果你没看过这本书，也没关系，下面就来简单回顾一下。回顾这些知识很有用，因为只有这样你才能知道如何安排处理器做事情。

电子计算机能做很多事情。你能够知道明天出门要穿厚一点才不挨冻，是因为电子计算机算出了天气。除此之外，它还能让你看电影、听音乐、写文章、上网。尽管表面上看来，这些用处和算数学题没什么关系，但实质上，这些功能都是以数学计算为基础的。正是因为如此，人们才会把“计算”这个词挂在嘴边，什么“云计算”、“网络计算”、“64 位计算”，等等。

处理器不是法师手里的仙器，它之所以能计算数学题，是因为其特殊的设计。处理器是一个

“器”，即器件，不太大，有的是长方形，有的是正方形，就像饼干。实际上，它是一块集成电路。

如图 2-2 所示，在处理器的底部或者四周，有大量的引脚，可以接受从外面来的电信号，或者向外发出电信号。每个引脚都有自己的用处，在往电路板上安装的时候，不能接错。所以，如图中所示，处理器在生产的时候，都会故意缺一个角，这是一个参照标志，可以确保安装的人不会弄错。当然，并不是所有的处理器都会缺一个角，这不是一个固定不变的做法。

处理器的引脚很多，其中有一部分是用来将参与运算的数字送入处理器内部。假如现在要进行加法运算，那么我们要重复使用这些引脚，来依次将被加数和加数送入。

一旦被加数通过引脚送入处理器，代表这个二进制数字的一组电信号就会出现在与引脚相连的内部线路上。这是一排高低电平的组合，代表着二进制数中的每一位。这时候，必须用一个称为寄存器（Register）的电路锁住。之所以要这样做，是因为相同的引脚和线路马上还要用于输入加数。也正是因为这个原因，这些内部线路称为处理器内部总线。

同样地，加数也要锁进另一个寄存器中。如图 2-2 所示，寄存器 RA 和 RB 将分别锁存参与运算的被加数和加数。此后，RA 和 RB 中的内容不再受外部数据线的影响。

寄存器是双向器件，可以在一端接受输入并加以锁存，同时，它也会在另一端产生一模一样的输出。与寄存器 RA 和 RB 相连的，是算术逻辑单元，或者算术逻辑部件（Arithmetic Logic Unit, ALU），也就是图 2-2 中的桶形部分。它是专门负责运算的电路，可以计算加法、减法或者乘法，也可做逻辑运算。在这里，我们要求它做一次加法。

一旦寄存器 RA 和 RB 锁存了参与运算的两个数，算术逻辑部件就会输出相加的结果，这个结果可以临时用另外一个寄存器 RC 锁存，稍后再通过处理器数据总线送到处理器外面，或者再次送入 RA 或 RB。

处理器总是很繁忙的，在它操作的过程中，所有数据在寄存器里面都只能是临时存在一会儿，然后再被送往别处，这就是为什么它被叫做“寄存器”的原因。早期的处理器，它的寄存器只能保存 4 比特、8 比特或 16 比特，它们分别叫做 4 位、8 位和 16 位寄存器。现在的处理器，寄存器一般都是 32 位、64 位甚至更多。

如图 2-3 所示，8 位寄存器可以容纳 8 比特（bit），或者说 1 字节（Byte），这是因为

$$1 \text{ byte} = 8 \text{ bit}$$

另外，我们还要为这个字节的每一位编上号，编号是从右往左进行的，从 0 开始，分别是 0、1、2、3、4、5、6、7。在这里，位 0（第 1 位）是最低位，在最右边；位 7（第 8 位）是最高位，在最左边。

为了更好地理解上面这些概念，图中假定 8 位寄存器里存放的是二进制数 10001101，即十六进制的 8D。这时，它的最低位和最高位都是 1。

16 位寄存器可以存放 2 个字节，这称为 1 个字（word），各个数位的编号分别是 0~15，其中 0~7 是低字节，8~15 是高字节。实际上，“字”的概念出现得很早，也并非指 16 个比特。只是到了后来，才特指 16 个二进制位的长度。

32 位寄存器可以存放 4 个字节，这称为 1 个双字（double word），各个数位的编号分别是 0~31，其中 0~15 是低字，16~31 是高字。

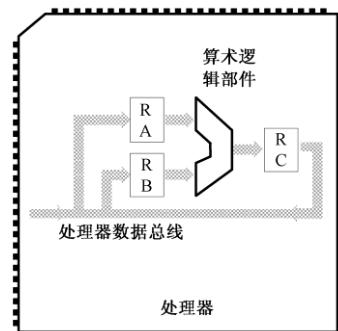


图 2-2 处理器进行数学运算的简单原理

尽管图中没有画出，但是 64 位寄存器可以容纳更多的比特，也就是 8 个字节，或者 4 个字。位数越多，寄存器所能保存的数越大，这是显而易见的。

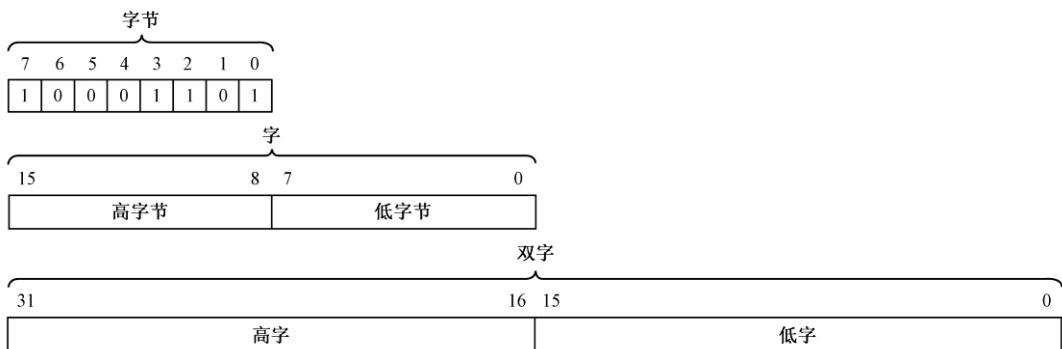


图 2-3 寄存器数据宽度示意

2.3 内 存 储 器

前面已经讲过，处理器的计算过程，实际上是借助于寄存器和算术逻辑部件进行的。那么，参与计算的数是从哪里来的呢？答案是一个可以保存很多数字的电路，叫做存储器（Storage 或 Memory）。

存储器的种类实际上是很多的，包括大家都知道的硬盘和 U 盘等。甚至寄存器就是存储器的一种。不过，我们现在所要讲到的存储器，则是另外一种东西。

如图 2-4 所示，这是所有个人计算机里都会用到的存储器，我们平时把它叫做内存条。这个概念是这么来的，首先，它是计算机内部最主要的存储器，通常只和处理器相连，所以叫做内存存储器或者主存储器，简称内存或主存。其次，它一般被设计成扁平的条状电路板，所以叫内存条。

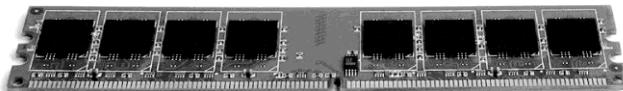


图 2-4 个人计算机里使用的内存条

如图 2-5 所示，和寄存器不同，内存用于保存更多的比特。对于用得最多的个人计算机来说，内存按字节来组织，单次访问的最小单位是 1 字节，这是最基本的存储单元。如图中所示，每个存储单元中，各位的编号分别是 0~7。

内存中的每字节都对应着一个地址，如图 2-5 所示，第 1 个字节的地址是 0000H，第 2 个字节的地址是 0001H，第 3 个字节的地址是 0002H，其他依次类推。注意，这里采用的是十六进制表示法。作为一个例子，因为这个内存的容量是 65536 字节，所以最后一个字节的地址是 FFFFH。

为了访问内存，处理器需要给出一个地址。访问包括读和写，为此，处理器还要指明，本次访问是读访问还是写访问。如果是写访问，则还要给出待写入的数据。

8 位处理器包含 8 位的寄存器和算术逻辑部件，16 位处理器拥有 16 位的寄存器和算术逻

辑部件，64位处理器则包含64位的寄存器和算术逻辑部件。尽管内存的最小组成单位是字节，但是，经过精心的设计和安排，它能够按字节、字、双字和四字进行访问。换句话说，仅通过单次访问就能处理8位、16位、32位或者64位的二进制数。注意，我说的是单次访问，而不是一个一个地取出每个字节，然后加以组合。

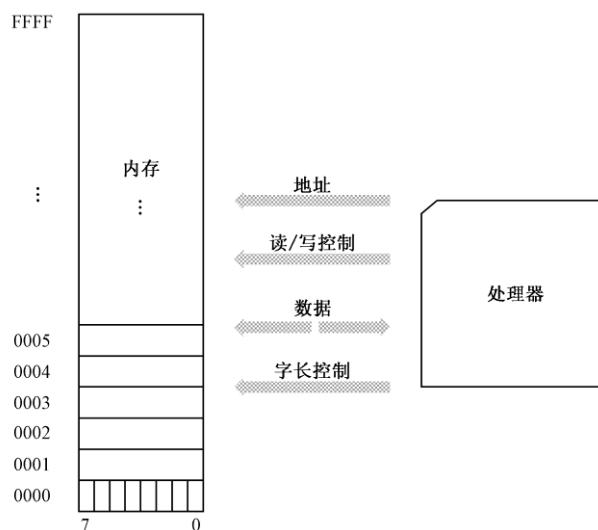


图 2-5 内存和内存访问示意图

如图 2-5 所示，处理器发出字长控制信号，以指示本次访问的字长是 8、16、32 还是 64。如果字长是 8，而且给出的地址是 0002H，那么，本次访问只会影响到内存的一字节；如果字长是 16，给出的地址依然是 0002H，那么实际访问的将是地址 0002H 处的一个字，低 8 位在 0002H 中，高 8 位在 0003H 中。

2.4 指令和指令集

从一开始，设计处理器的目标之一就是使它成为一种可以自动进行操作的器件。另外，还需要提供一种机制，来允许工程技术人员决定进行何种操作。

处理器何以能够自动进行操作，这不是本书的话题，大学里有这样的课程，《穿越计算机的迷雾》这本书也给出了通俗化的答案。

简单地说，处理器的设计者用某些数来指示处理器所进行的操作，这称为指令(Instruction)，或者叫机器指令，因为只有处理器才认得它们。比如，指令 F4H 表示让处理器停机，当处理器取到并执行这条指令后，就停止工作。指令是集中存放在内存里的，一条接着一条，处理器的工作是自动按顺序取出并加以执行。

如图 2-6 所示，从内存地址 0000H 开始（也就是内存地址的最低端）连续存放了一些指令。同时，假定执行这些指令的是一个 16 位处理器，拥有两个 16 位的寄存器 RA 和 RB。

一般来说，指令由操作码和操作数构成，但也有小部分指令仅有操作码，而不含操作数。如图 2-6 所示，停机指令仅包含 1 字节的操作码 F4，而没有操作数。指令的长度不定，短的指令仅有 1 字节，而长的指令则有可能达到 15 字节。

对处理器来说，指令的操作码隐含了如何执行该指令的信息，比如它是做什么的，以及怎么去做。第一条指令的操作码是 B8，这表明，该指令是一条传送指令，第一个操作数是寄存器，第二个操作数是直接包含在指令中的，紧跟在操作码之后，可以立即从指令中取得，所以叫做立即数（Immediate Operand）。同时，操作码还直接指出该寄存器是 RA。RA 是 16 位寄存器，这条指令将按字进行操作。所以，当这条指令执行之后，该指令的操作数（立即数）005DH 就被传送到 RA 中。

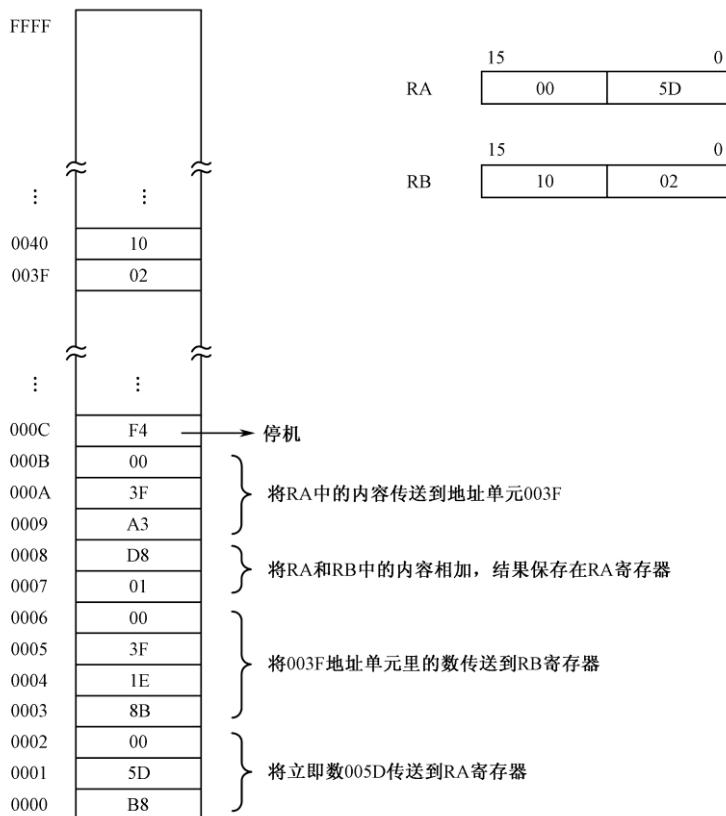


图 2-6 处理器指令在内存中的布局

既然操作码中隐含了这么多的信息，那么，处理器就可以“知道”每条指令的长度。这样，当它执行第一条指令 B8 5D 00 的时候，就已经知道，这是一个 3 字节指令，下一条指令位于 3 个字节之后，即内存地址 0003H 处。

注意字数据在内存中的存放特点。地址 0001H 和 0002H 里的内容分别是 5D 和 00，如果每次读一个字节，则从地址 0001H 里读出的是 5D，从 0002H 里读出的是 00。但如果以字的方式来访问地址 0001H，读到的就是 005DH。这种差别，跟处理器和内存之间的数据线连接方式有关。对于 Intel 处理器来说，如果访问内存中的一个字，那么，它规定高字节位于高地址部分，低字节位于低地址部分，这称为低端字节序（Little Endian）。至于其他公司的处理器，则可能情况正好相反，称为高端字节序。

对于复杂一些的指令来说，1 个字节的操作码可能不够用。所以，第 2 条指令的操作码为 8B 1E，它隐含的意思是，这是一条传送指令，第一个操作数是寄存器，而且是 RB 寄存器，第二个操作数是内存地址，要传送到 RB 寄存器中的数存放在该地址中。同时，这是一个字操

作指令，应当从第二个操作数指定的地址中取出一个字。

该指令的操作数部分是 3F 00，指定了一个内存地址 003FH。它相当于高级语言里的指针，当处理器执行这条指令时，会再次用 003FH 作为地址去访问内存，从那里取出一个字(1002H)，然后将它传送到寄存器 RB。注意，“传送”这个词带有误导性。其实，传送的意思更像是“复制”，传送之后，003FH 单元里的数据还保持原样。

通过这两条指令的比较，很容易分清指令中的“立即数”是什么意思。指令执行和操作的对象是数。如果这个数已经在指令中给出了，不需要再次访问内存，那这个数就是立即数，比如第一条指令中的 005DH；相反，如果指令中给出的是地址，真正的数还需要用这个地址访问内存才能得到，那它就不能称为立即数，比如第二条指令中的 003FH。

如图 2-6 所示，余下的三条指令，旁边都有注解，这里就不再一一解释了。如果一开始内存地址 003FH 中存放的是 1002H，那么，当所有这些指令执行完后，003FH 里就是最终的结果 105FH。

指令和非指令的普通二进制数是一模一样的，在组成内存的电路中，都是一些高低电平的组合。因为处理器是自动按顺序取指令并加以执行的，在指令中混杂了非指令的数据会导致处理器不能正常工作。为此，指令和数据要分开存放，分别位于内存中的不同区域，存放指令的区域叫代码区，存放数据的区域叫数据区。为了让处理器正确识别和执行指令，工程技术人员必须精心安排，并告诉处理器要执行的指令位于内存中的什么位置。

还是那句话，并非每一个二进制数都代表着一条指令。每种处理器在设计的时候，也只能拥有有限的指令，从几十条到几百条不等。一个处理器能够识别的指令的集合，称为该处理器的指令集。

2.5 古老的 Intel 8086 处理器

任何时候，一旦提到 Intel 公司的处理器，就不能不说 8086。8086 是 Intel 公司第一款 16 位处理器，诞生于 1978 年，所以说它很古老。

但是，在 Intel 公司的所有处理器中，它占有很重要的地位，是整个 Intel 32 位架构处理器（IA-32）的开山鼻祖。首先，最重要的一点是，它是一款非常成功的产品，设计先进，功能很强，卖得很好。

其次，8086 的成功使得市场上出现了大量针对它开发的软件产品。这样，当 Intel 公司要设计新的处理器时，它不得不考虑到兼容性的问题。要使得老的软件也能在新的处理器上很好地运行，必须要具备指令集和工作模式上的兼容性和一致性。Intel 公司很清楚，如果新处理器和老处理器不兼容，那么，新处理器越多，它扔掉的拥趸也就越多，要不了多久，这公司就不用再开了。

所以，当我们讲述处理器的时候，必须要从 8086 开始；而且，要学习汇编语言，针对 8086 的汇编技术也是必不可少的。

2.5.1 8086 的通用寄存器

8086处理器内部有8个16位的通用寄存器，分别被命名为AX、BX、CX、DX、SI、DI、BP、SP。“通用”的意思是，它们之中的大部分都可以根据需要用于多种目的。

如图2-7所示，因为这8个寄存器都是16位的，所以通常用于进行16位的操作。比如，可以在这8个寄存器之间互相传送数据，它们之间也可以进行算术逻辑运算；也可以在它们和内存单元之间进行16位的数据传送或者算术逻辑运算。

同时，如图2-7所示，这8个寄存器中的前4个，即AX、BX、CX和DX，又各自可以拆分成两个8位的寄存器来使用，总共可以提供8个8位的寄存器AH、AL、BH、BL、CH、CL、DH和DL。这样一来，当需要在寄存器和寄存器之间，或者寄存器和内存单元之间进行8位的数据传送或者算术逻辑运算时，使用它们就很方便。

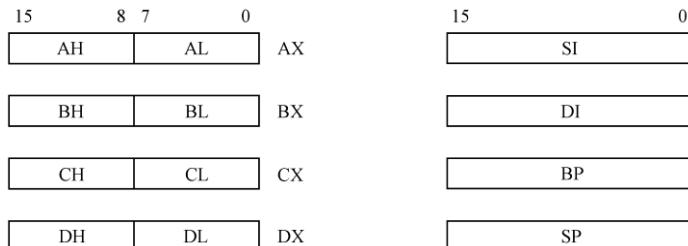


图2-7 8086的通用寄存器

将一个16位的寄存器当成两个8位的寄存器来用时，对其中一个8位寄存器的操作不会影响到另一个8位寄存器。举个例子来说，当你操作寄存器AL时，不会影响到AH中的内容。

2.5.2 程序的重定位难题

我们知道，处理器是自动化的器件，在给出了起始地址之后，它将从这个地址开始，自动地取出每条指令并加以执行。只要每条指令都正确无误，它就能准确地知道下一条指令的地址。这就意味着，完成某个工作的所有指令，必须集中在一起，处于内存的某个位置，形成一个段，叫做代码段。事情是明摆着的，要是指令并没有一条挨着一条存放，中间夹杂了其他非指令的数据，处理器将因为不能识别而出错。

为了做某件事而编写的指令，它们一起形成了我们平时所说的程序。程序总要操作大量的数据，这些数据也应该集中在一起，位于内存中的某个地方，形成一个段，叫做数据段。

段在内存中的位置并不重要，因为处理器是可控的，我们可以让它从内存的任何位置开始取指令并加以执行。这里有一个例子，如图2-8所示，我们有一大堆数字，现在想把它们加起来求出一个总和。

假定我们有16个数要相加，这些数都是16位的二进制数，分别是0005H、00A0H、00FFH、…。为了让处理器把它们加起来，我们应该先在内存中定义一个数据段，将这些数字写进去。数据段可以起始于内存中的任何位置，既然如此，我们将它定在0100H处。这样一来，第一个要加的数位于地址0100H，第二个要加的数位于地址0102H，最后一个数的地址是011EH。

一旦定义了数据段，我们就知道了每个数的内存地址。然后，紧挨着数据段，我们从内存地址0120H处定义代码段。严格地说，数据段和代码段是不需要连续的，但这里把它们挨在一

起更自然一些。为了区别数据段和代码段，我们使用了不同的底色。

代码段是从内存地址 0120H 处开始的,第一条指令是 A1 00 01,其功能是将内存单元 0100H 里的字传送到 AX 寄存器。指令执行后, AX 的内容为 0005H。

第二条指令是 03 06 02 01，功能是将 AX 中的内容和内存单元 0102H 里的字相加，结果在 AX 中。由于 AX 的内容为 0005H，而内存地址 0102H 里的数是 00A0H，这条指令执行后，AX 的内容为 00A5H。

第三条指令是 03 06 04 01，功能是将 AX 中的内容和内存单元 0104H 里的字相加，结果在 AX 中。此时，由于 AX 里的内容是 00A5H，内存地址 0104H 里的数是 00FFH，本指令执行后，AX 的内容为 01A4H。

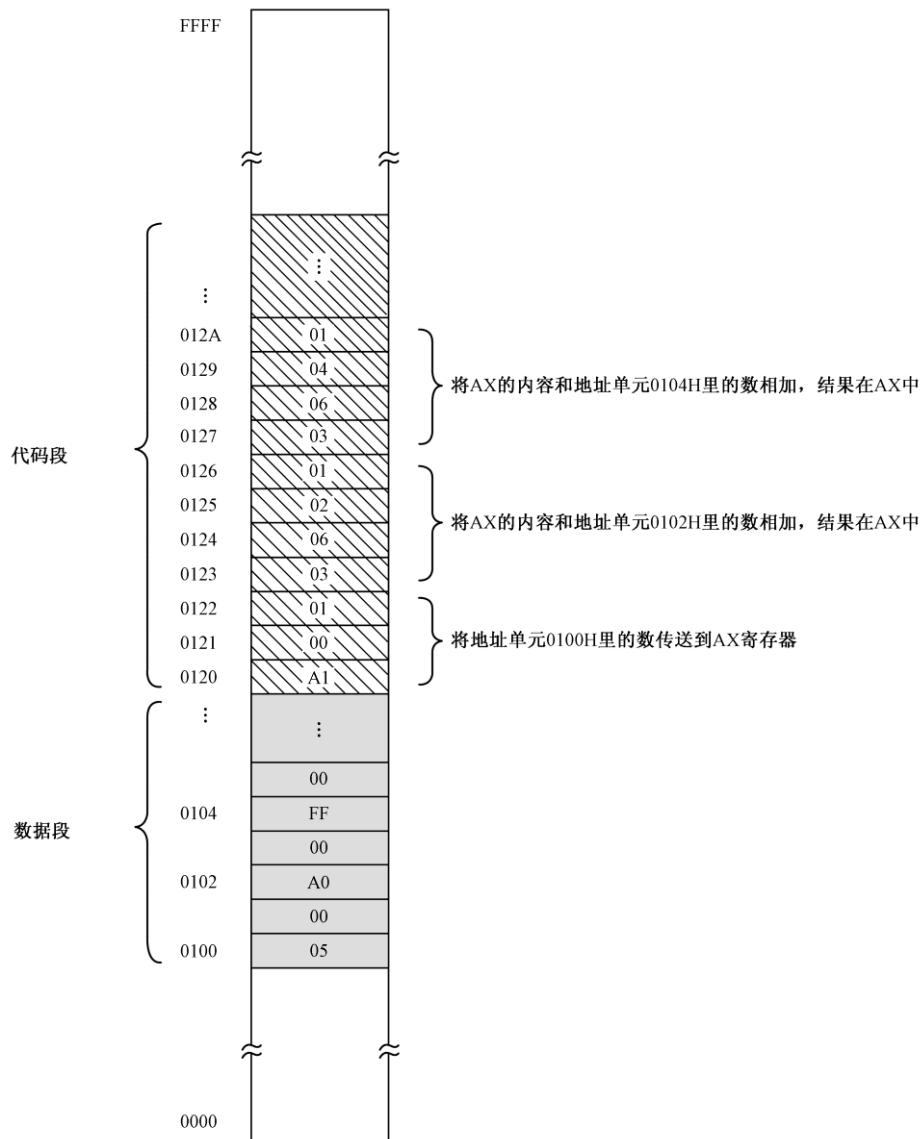


图 2-8 程序的代码段和数据段示例

后面的指令没有列出，但和前 2 条指令相似，依次用 AX 的内容和下一个内存单元里的字

相加，一直到最后，在 AX 中得到总的累加和。在这个例子中，我们没有考虑 AX 寄存器容纳不下结果的情况。当累加的总和超出了 AX 所能表示的数的范围（最大为 FFFFH，即十进制的 65535）时，就会产生进位，但这个进位被丢弃。

在内存中定义了数据段和代码段之后，我们就可以命令处理器从内存地址 0120H 处开始执行。当所有的指令执行完后，就能在 AX 寄存器中得到最后的结果。

看起来没有什么问题，一切都很完美，不是吗？那本节标题中所说的难题又从何而来呢？这里确实有一个难题。

在前面的例子中，所有在执行时需要访问内存单元的指令，使用的都是真实的内存地址。比如 A1 00 01，这条指令的意思是从地址为 0100H 的内存单元里取出一个字，并传送到寄存器 AX。在这里，0100H 是一个真实的内存地址，又称物理地址。

整个程序（包括代码段和数据段）在内存中的位置，是由我们自己定的。我们把数据段定在 0100H，把代码段定在 0120H。

问题是，大多数时候，整个程序（包括代码段和数据段）在内存中的位置并不是我们能够决定的。请想一想你平时是怎么使用计算机的，你所用的程序，包括那些用来调整计算机性能

的工具、小游戏、音乐和视频播放器等，都是从网上下载的，位于你的硬盘、U 盘或光盘中。即使有些程序是你自己编写的，那又如何？当你双击它们的图标，使它们在 Windows 里启动之前，内存已经被塞了很多东西，就算你是刚刚打开计算机，Windows 自己已经占用了很多内存空间，不然的话，你怎么可能在它上面操作呢？

在这种情况下，你所运行的程序，在内存中被加载的位置完全是随机的，哪里有空闲的地方，它就会被加载到哪里，并从那里开始被处理器执行。所以，前面那段程序不可能恰好如你所愿，被加载到内存地址 0100H，它完全可能被加载到另一个不同的位置，比如 1000H。但是，同样是那个程序，一旦它在内存中的位置发生了改变，灾难就出现了。

如图 2-9 所示，因为程序现在是从内存地址 1000H 处被加载的，所以，数据段的起始地址为 1000H。这就是说，第一个要加的数，其地址为 1000H，第二个则为 1002H，其他依次类推。代码段依然紧挨着数据段之后，起始地址相应地是 1020H。

只要所有的指令都是连续存放的，代码段位于内存中的什么地方都可以正常执行。所以，处理器可以按你的要求，从

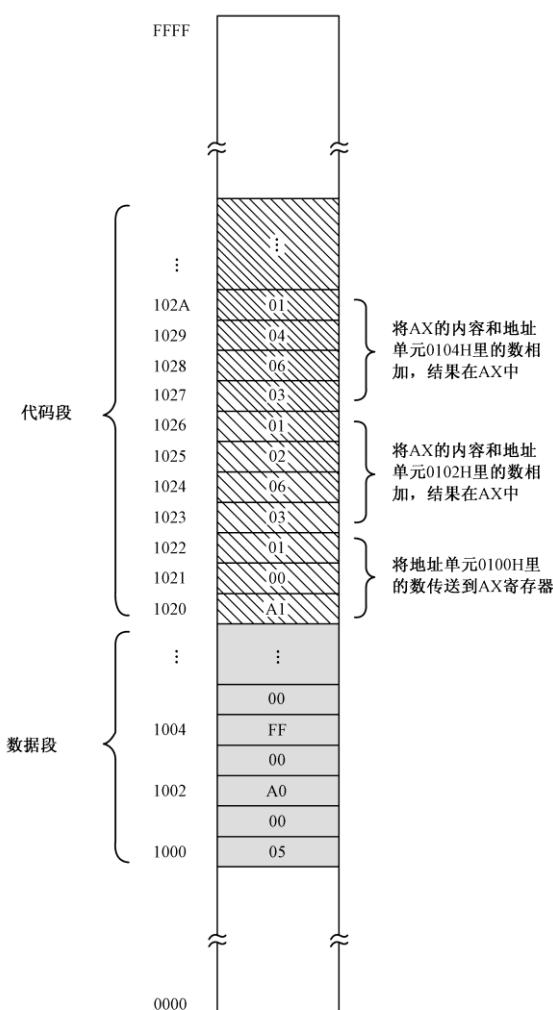


图 2-9 在指令中使用绝对内存地址的程序是不可重定位的

内存地址 1020H 处连续执行，但结果完全不是你想要的。

请看第一条指令 A1 00 01，它的意思是内存地址 0100 处取得一个字，将其传送到寄存器 AX。但是，由于程序刚刚改变了位置，它要取的那个数，现在实际上位于 1000H，它取的是别人地盘里的数！

这能怪谁呢？发生这样的事情，是因为我们在指令中使用了绝对内存地址（物理地址），这样的程序是无法重定位的。为了让你写的程序在卖给别人之后，可以在内存中的任何地方正确执行，就只能在编写程序的时候使用相对地址或者逻辑地址了，而不能使用真实的物理地址。当程序加载时，这些相对地址还要根据程序实际被加载的位置重新计算。

在任何时候，程序的重定位都是非常棘手的事情。当然，也有好几种解决的办法。在 8086 处理器上，这个问题特别容易解决，因为该处理器在访问内存时使用了分段机制，我们可以借助该机制。

2.5.3 内存分段机制

如图 2-10 所示，整个内存空间就像长长的纸条，在内存中分段，就像从长纸条中裁下一小段来。根据需要，段可以开始于内存中的任何位置，比如图中的内存地址 A532H 处。

在这个例子中，分段开始于地址为 A532H 的内存单元处，这个起始地址就是段地址。

这个分段包含了 6 个存储单元。在分段之前，它们在整个内存空间里的物理地址分别是 A532H、A533H、A534H、A535H、A536H、A537H。

但是，在分段之后，它们的地址可以只相对于自己所在的段。这样，它们相对于段开始处的距离分别为 0、1、2、3、4、5，这叫做偏移地址。

于是，当采用分段策略之后，一个内存单元的地址实际上就可以用“段：偏移”或者“段地址：偏移地址”来表示，这就是通常所说的逻辑地址。比如，在图 2-10 中，段内第 1 个存储单元的地址为 A532H:0000H，第 3 个存储单元的地址为 A532H:0002H，而本段最后一个存储单元的地址则是 A532H:0005H。

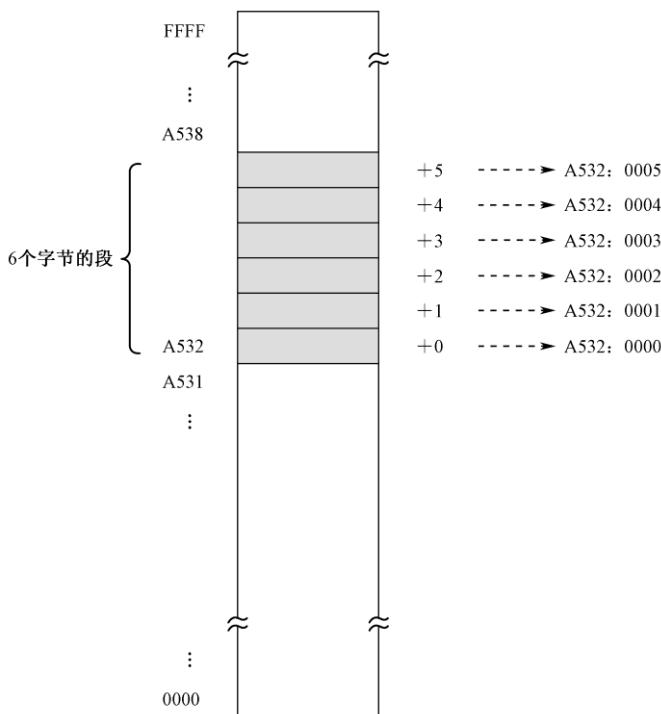


图 2-10 段地址和偏移地址示意图

为了在硬件一级提供对“段地址：偏移地址”内存访问模式的支持，处理器至少要提供两个段寄存器，分别是代码段（Code Segment，CS）寄存器和数据段（Data Segment，DS）寄存器。

对 CS 内容的改变将导致处理器从新的代码段开始执行。同样，在开始访问内存中的数据之前，也必须首先设置好 DS 寄存器，使之指向数据段。

除此之外，最重要的是，当处理器访问内存时，它把指令中指定的内存地址看成是段内的偏移地址，而不是物理地址。这样，一旦处理器遇到一条访问内存的指令，它将把 DS 中的数据段起始地址和指令中提供的段内偏移相加，来得到访问内存所需要的物理地址。

如图 2-11 所示，代码段的段地址为 1020H，数据段的段地址为 1000H。在代码段中有一条指令 A1 02 00，它的功能是将地址 0002H 处的一个字传送到寄存器 AX。在这里，处理器将 0002H 看成是段内的偏移地址，段地址在 DS 中，应该在执行这条指令之前就已经用别的指令传送到 DS 中了。

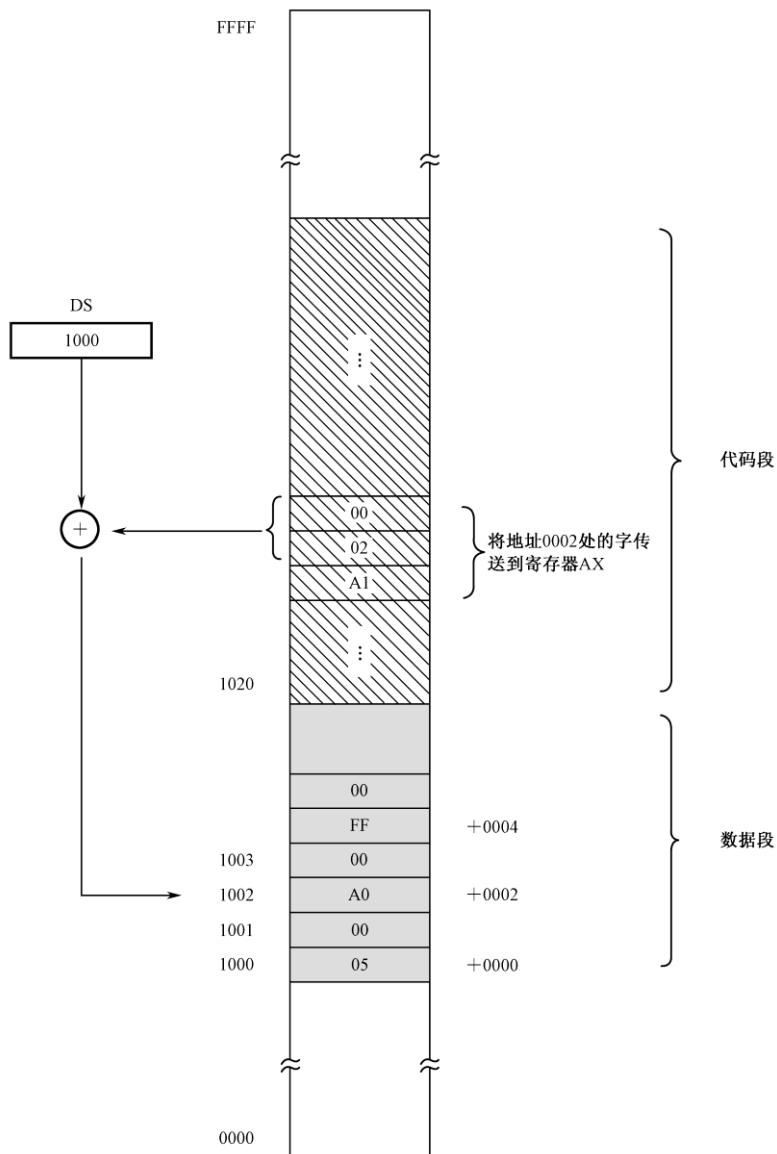


图 2-11 从逻辑地址到物理地址的转换过程

当执行指令 A1 02 00 时，处理器将把 DS 中的内容和指令中指定的地址 0002H 相加，得到 0002H。这是一个物理地址，处理器用它来访问内存，就可以得到所需要的数 00A0H。

如果一下次执行这个程序时，代码段和数据段在内存中的位置发生了变化，只要把它们的段地址分别传送到 CS 和 DS，它也能够正确执行。

2.5.4 8086 的内存分段机制

前面讲了如何从逻辑地址转换到物理地址，以使得程序的运行和它在内存中的位置无关。这种策略在很多处理器中得到了支持，包括 8086 处理器。但是，由于 8086 自身的局限性，它的做法还要复杂一些。

如图 2-12 所示，8086 内部有 8 个 16 位的通用寄存器，分别是 AX、BX、CX、DX、SI、

DI、BP、SP。其中，前四个寄存器中的每一个，都还可以当成两个8位的寄存器来使用，分别是AH、AL、BH、BL、CH、CL、DH、DL。

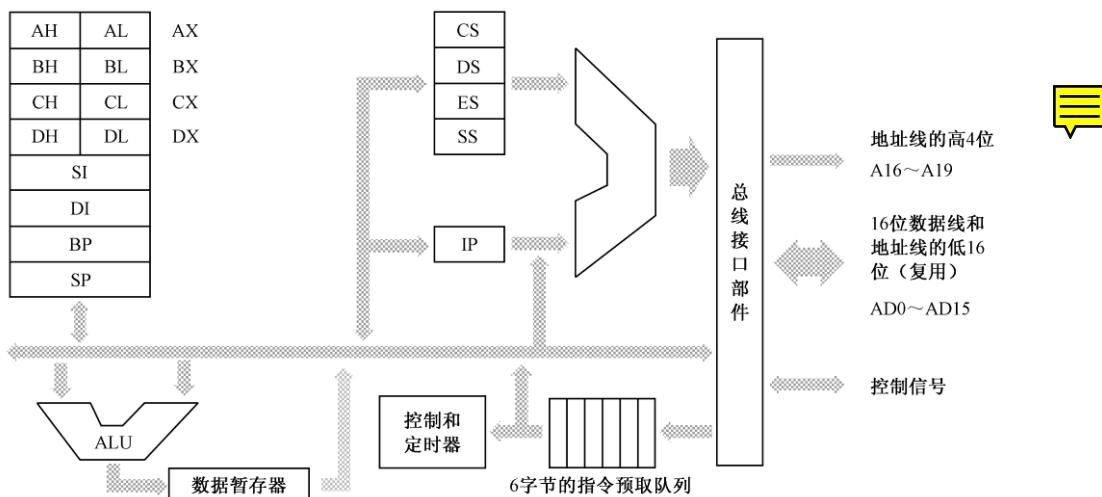


图2-12 8086处理器内部组成框图

在进行数据传送或者算术逻辑运算的时候，使用算术逻辑部件（ALU）。比如，将AX的内容和CX的内容相加，结果仍在AX中，那么，在相加的结果返回到AX之前，需要通过一个叫数据暂存器的寄存器中转。

处理器能够自动运行，这是控制器的功劳。为了加快指令执行速度，8086内部有一个6字节的指令预取队列，在处理器忙着执行那些不需要访问内存的指令时，指令预取部件可以趁机访问内存预取指令。这时，多达6个字节的指令流可以排队等待解码和执行。

8086内部有4个段寄存器。其中，CS是代码段寄存器，DS是数据段寄存器，ES是附加段（Extra Segment）寄存器。附加段的意思是，它是额外赠送的礼物，当需要在程序中同时使用两个数据段时，DS指向一个，ES指向另一个。可以在指令中指定使用DS和ES中的哪一个，如果没有指定，则默认是使用DS。SS是栈段寄存器，以后会讲到，而且非常重要。

IP是指令指针（Instruction Pointer）寄存器，它只和CS一起使用，而且只有处理器才能直接改变它的内容。当一段代码开始执行时，CS指向代码段的起始地址，IP则指向段内偏移。这样，由CS和IP共同形成逻辑地址，并由总线接口部件转换成物理地址来取得指令。然后，处理器会自动根据当前指令的长度来改变IP的值，使它指向下一条指令。

当然，如果在指令的执行过程中需要访问内存单元，那么，处理器将用DS的值和指令中提供的偏移地址相加，来形成访问内存所需的物理地址。

8086的段寄存器和IP寄存器都是16位的，如果按照原先的方式，把段寄存器的内容和偏移地址直接相加来形成物理地址的话，也只能得到16位的物理地址。麻烦的是，8086却提供了20根地址线。换句话说，它提供的是20位的物理地址。

提供20位地址线的原因很简单，16位的物理地址只能访问64KB的内存，地址范围是0000H~FFFFH，共65536个字节。这样的容量，即使是在那个年代，也显得捉襟见肘。注意，这里提到了一个表示内存容量的单位“KB”。为了方便，我们通常使用更大的单位来描述内存容量，比如千字节（KB）、兆字节（MB）和吉字节（GB），它们之间的换算关系如下：

$$1 \text{ KB} = 1024 \text{ Byte}$$

$$1 \text{ MB} = 1024 \text{ KB}$$

1 GB = 1024 MB

所以，65536 个字节就是 64KB，而 20 位的物理地址则可以访问多达 1MB 的内存，地址范围从 00000H 到 FFFFFH。问题是，16 位的段地址和 16 位的偏移地址相加，只能形成 16 位的物理地址，怎么得到这 20 位的物理地址呢？

为了解决这个问题，8086 处理器在形成物理地址时，先将段寄存器的内容左移 4 位（相当于乘以十六进制的 10，或者十进制的 16），形成 20 位的段地址，然后再同 16 位的偏移地址相加，得到 20 位的物理地址。比如，对于逻辑地址 F000H:052DH，处理器在形成物理地址时，将段地址 F000H 左移 4 位，变成 F0000H，再加上偏移地址 052DH，就形成了 20 位的物理地址 F052DH。

这样，因为段寄存器是 16 位的，在段不重叠的情况下，最多可以将 1MB 的内存分成 65536 个段，段地址分别是 0000H、0001H、0002H、0003H，……，一直到 FFFFH。在这种情况下，如图 2-13 所示，每个段正好 16 个字节，偏移地址从 0000H 到 000FH。

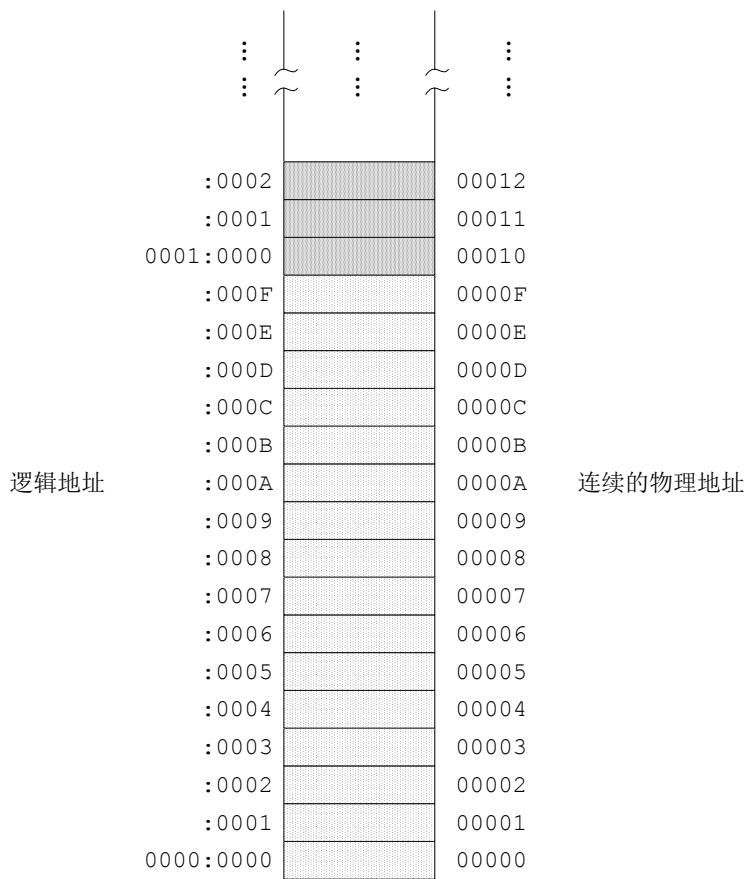


图 2-13 1MB 内存可以划分为 65536 个 16 字节的段

同样在不允许段之间重叠的情况下，每个段的最大长度是 64KB，因为偏移地址也是 16 位的，从 0000H 到 FFFFH。在这种情况下，1MB 的内存，最多只能划分成 16 个段，每段长 64KB，段地址分别是 0000H、1000H、2000H、3000H，…，一直到 F000H。

以上所说的只是两种最典型的情况。通常情况下，段地址的选择取决于内存中哪些

区域是空闲的。举个例子来说，假如从物理地址 00000H 开始，一直到 82251H 处都被其他程序占用着，而后面一直到 FFFFFH 的地址空间都是自由的，那么，你可以从物理内存地址 82251H 之后的地方加载你的程序。

接着，你的任务是定义段地址并设置处理器的段寄存器，其中最重要的是段地址的选取。因为偏移地址总是要求从 0000H 开始，而 82260H 是第一个符合该条件的物理地址，因为它恰好对应着逻辑地址 8226H:0000H，符合偏移地址的条件，所以完全可以将段地址定为 8226H。

但是，举个例子来说，如果你从物理内存地址 82255H 处加载程序，由于它根本无法表示成一个偏移地址为 0000H 的逻辑地址，所以不符合要求，段不能从这里开始划分。这里面的区别在于，82260H 可以被十进制数 16（或者十六进制数 10H）整除，而 82255H 不能。通过这个例子可以看出，8086 处理器的逻辑分段，起始地址都是 16 的倍数，这称为是按 16 字节对齐的。

段的划分是自由的，它可以起始于任何 16 字节对齐的位置，也可以是任意长度，只要不超过 64KB。比如，段地址可以是 82260H，段的长度可以是 64KB。在这种情况下，该段所对应的逻辑地址范围是 8226H:0000H～8226H:FFFFH，其所对应的物理地址范围是 82260～9225FH。

同时，正是由于段的划分非常自由，使得 8086 的内存访问也非常随意。同一个物理地址，或者同一片内存区域，根据需要，可以随意指定一个段来访问它，前提是那个物理地址位于该段的 64KB 范围内。也就是说，同一个物理地址，实际上对应着多个逻辑地址。

本 章 习 题

数据段寄存器 DS 的值为 25BCH 时，计算 Intel 8086 可以访问的物理地址范围。

第3章 汇编语言和汇编软件

3.1 汇编语言简介

在前面的章节里，我们讲到了处理器，也讲了处理器是如何进行算术逻辑运算的。为了实现自动计算，处理器必须从内存中取得指令，并执行这些指令。

指令和被指令引用的数据在内存中都是一些或高或低的电平，每一个电平都可以看成是一个二进制位（0或者1），8个二进制位形成一字节。

要解读内存中的东西，最好的办法就是将它们按字节转换成数字的形式。比如，下面这些数字就是存放在内存中的8086指令，我们用的是十六进制：

```
B8 3F 00 01 C3 01 C1
```

对于大多数人来说，他们很难想象上面那一排数字对应着下面几条8086指令：

将立即数003FH传送到寄存器AX；

将寄存器BX的内容和寄存器AX的内容相加，结果在BX中；

将寄存器CX的内容和寄存器AX的内容相加，结果在CX中。

即使是很经验的技术人员，要想用这种方式来编写指令，也是很困难的，而且很容易出错。所以，在第一个处理器诞生之后不久，如何使指令的编写变得更容易，就提上了日程。

为了克服机器指令难以书写和理解的缺点，人们想到可以用一些容易理解和记忆的符号，也就是助记符，来描述指令的功能和操作数的类型，这就产生了汇编语言（Assembly Language）。这样，上面那些指令就可以写成：

```
mov ax,3FH  
add bx,ax  
add cx,ax
```

对于那些有点英语基础的人来说，理解这些汇编语言指令并不困难。比如这句

```
mov ax,3FH
```

首先，`mov`是`move`的简化形式，意思是“移动”或者“传送”。至于“ax”，很明显，指的就是AX寄存器。传送指令需要两个操作数，分别是目的操作数和源操作数，它们之间要用逗号隔开。在这里，AX是目的操作数，源操作数是3FH。汇编语言对指令的大小写没有特别的要求。所以你完全可以这样写：

```
MOV AX,3FH  
mov ax,3fh  
MOV ax,3FH  
mov AX,3fh
```

在很多高级语言中，如果要指示一个数是十六进制数，通常不采用在后面加“H”的做法，

而是为它添加一个“0x”前缀。像这样：

```
mov ax, 0x3f
```

你可能想问一下，为什么会是这样，为什么会是“0x”？答案是不知道，不知道在什么时候，为什么就这样用了。这不得不让人怀疑，它肯定是一个非常随意的决定，并在以后形成了惯例。如果你知道确切的答案，不妨写封电子邮件告诉我。注意，为了方便，我们将在本书中采用这种形式。

在汇编语言中，使用十进制数是最自然的。因为 3FH 等于十进制数 63，所以你可以直接这样写：

```
mov ax, 63
```

当然，如果你喜欢，也可以使用二进制数来这样写：

```
mov ax, 00111111B
```

一定要看清楚，在那串“0”和“1”的组合后面，跟着字母“B”，以表明它是一个二进制数。

至于这句：

```
add bx, ax
```

情况也是一样。`add` 的意思是把一个数和另一个数相加。在这里，是把 BX 寄存器的内容和 AX 寄存器的内容相加。相加的结果在 BX 中，但 AX 的内容并不改变。

像上面那样，用汇编语言提供的符号书写的文本，叫做汇编语言源程序。为此，你需要一个字处理器软件，比如 Windows 记事本，来编辑这些内容。如图 3-1 所示，相信这些软件的使用都是你已经非常熟悉的。



图 3-1 用 Windows 记事本来书写汇编语言源程序

有了汇编语言所提供的符号，这只是方便了你自己。相反地，对人类来说通俗易懂的东西，处理器是无法识别的。所以，还需要将汇编语言源程序转换成机器指令，这个过程叫做编译（Compile）。

编译肯定还需要依靠一个软件，称为编译器，或编译软件。因为如果需要人类自己去做，还费这周折干嘛。另一方面，想想看，一个帮助人类生产软件的工具，自己居然也是一个软件，这很有意思。

从字处理器软件生成的是汇编语言源程序文件。编译软件的任务是读取这些文件，将那些符号转变成二进制形式的机器指令代码。它把这些机器代码存放到另一个文件中，叫做二进制文件或者可执行文件，比如 Windows 里以“.exe”为扩展名的文件，就是可执行文件。当需要用处理器执行的时候，再加载到内存里。

3.2 NASM 编译器

3.2.1 从网上下载 NASM 安装程序

每种处理器都可能会有自己的汇编语言编译器，而对于同一款处理器来说，针对不同的平台（比如 Windows 和 Linux），也会有不同版本的汇编语言编译器。

现存的汇编语言编译器有多种，用得比较多的有 MASM、FASM、TASM、AS86、GASM 等，每种汇编器都有自己的特色和局限性。特别是，有些还需要付费才能使用。不同于前面所列举的这些，在本书中，我们用的是另一款叫做 NASM 的汇编语言编译器。

NASM 的全称是 Netwide Assembler，它是可免费使用的开源软件。下面是它的下载地址：

<http://sourceforge.net/projects/nasm/files/>

我写这本书的时候，用的是微软 Windows 操作系统。而且，这本书的读者最好也使用 Windows，否则这本书所配套的工具软件可能无法使用。随着后 PC 时代的来临，Windows 正在逐渐受到平板电脑和手机的威胁，市场份额已经跌至 82%，创 20 年新低。但即使是在它占据绝大部分市场的年代，也还有很多其他操作系统运行在 Intel 处理器上，而且不乏拥趸，比如 DOS、OS/2、Linux 等。

理论上，不管用的是什么操作系统，Windows 也好，DOS 也好，Linux 也好，只要是针对 Intel 处理器开发的软件，底层的机器指令代码都是相同的，没有理由说某个软件只能在 Windows 操作系统上运行，而不能在 Linux 上运行。

事实上，仅仅具有一致的底层机器代码还远远不够。别忘了，这些代码要被处理器来依次执行，首先需要加载到内存并实施重定位。在这种情况下，除了那些真正用于做事的机器指令之外，软件还需要一些额外的信息来告诉操作系统，如何加载自己。更有甚者，Windows 会建议为它开发的软件应当包含一些图标或者图片。这就是为什么每个 Windows 软件都会显示一个图标的原因。

在这种情况下，因为每种操作系统都会根据自身的工作特点，定义自己所能识别的软件可执行文件格式，而缺乏通用性，尽管在这些软件里，真正用于计算 5+6 的机器指令都一模一样。

作为一款汇编软件，没有理由只考虑 Windows 用户而忽略其他操作系统平台上的人们。所以，如图 3-2 所示，在使用任意一款互联网浏览器转到上面指示的链接时，网页上将显示一些目录（文件夹）。基本上，每个目录的名字都指示出一个操作系统平台的名字，但也有个别例外。比如，“nasm documentation” 目录包含的是各个版本的 NASM 开发与帮助文档。

因为我们需要在 Windows 上学习汇编语言编程技术，所以可单击“Win32 binaries”，意思是针对 32 位 Windows 的二进制文件。在特定的场合，比如这里，“二进制文件”和“程序文件”以及“可执行文件”是一个意思。

单击之后，在另一个网页里，会出现更多的目录，这些都是历史版本。毕竟一款软件需要不停地改进，每改进一次，都会形成一个最新的版本。在这些页面里，可以单击列在最顶端的那个版本，通常它是最新的版本。

这样，你就会来到最后一个页面，看起来可能像图 3-3 那样。

The screenshot shows a web browser displaying a list of files from the NASM project page on SourceForge. The URL in the address bar is <http://sourceforge.net/projects/nasm/files/>. The page title is "The Netwide Asse...". A message at the top says "Looking for the latest version? Download nasm-2.07-installer.exe (687.4 kB)". Below this is a table listing 11 items:

Name	Modified	Size
Contributions	2009-09-25	
Linux RPM binaries	2009-07-21	
nasm sources	2009-07-21	
nasm documentation	2009-07-21	
Win32 binaries	2009-07-21	
OS_2 Binaries	2009-07-21	
DOS 32-bit binaries	2009-07-21	
Linux SRPM source archives	2009-07-21	
example code	2009-07-17	
DOS 16-bit binaries (OBSOLETE)	2005-02-27	
tools	2002-04-08	

Totals: 11 Items

图 3-2 sourceforge 网站展示了针对各种操作系统平台所开发的 NASM 汇编器

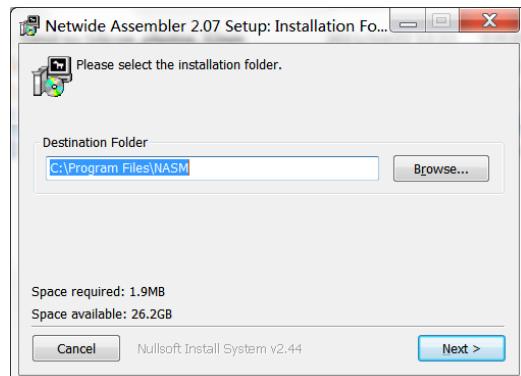


图 3-3 选择最终所要下载的 NASM 安装文件

前面的网页可能都一模一样，唯独这个页面。原因在于我制作这幅图片的时候，最新的 NASM 版本是 2.07。等到你拿起这本书，也到网上下载时，可能会有更新的版本。

无论如何，就以图 3-2 为例，虽然两个文件下载哪个都可以，但建议下载 nasm-2.07-installer.exe，原因在于这是一个安装程序，可以自动把它自己安装到你的计算机里。对于现在的人们来说，下载是很简单的事情。即使你不学习汇编语言，也经常下载 MP3 歌曲、高清电影、手机铃声或者手机游戏。

上面所说的，是如何从 NASM 的官网上下载它。当然，如果你愿意，也可以用搜索引擎，比如百度，以关键词“NASM”来搜索它。剩下的事情，就是从搜索结果里找出一个合适的下载链接。

3.2.2 安装 NASM 编译器

一旦顺利将 NASM 安装文件下载到计算机里，我们就可以立即运行它来自动完成整个安装过程。这个过程很简单，不过有两个步骤需要说明一下，第一个就是选择安装文件夹，如图 3-4 所示。

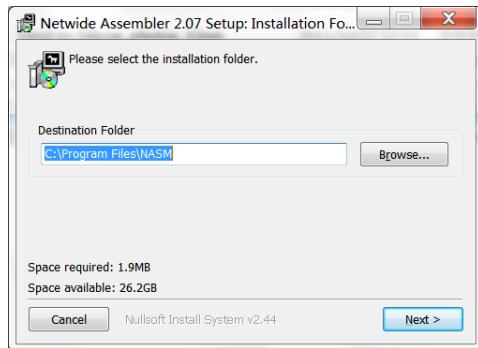


图 3-4 为在计算机上安装 NASM 准备文件夹

在这个安装环节的界面上，给出了默认的安装文件夹。你可以接受这个默认路径，也可以单击“Browse”（浏览）来选择一个不同的路径。无论如何，你一定要记住这个路径，因为以后还要用到它。

第二个步骤是安装选项。如图 3-5 所示，“Netwide Assembler 2.07”是必选的，因为你安装的目的就是为了使用它。“Start Menu Shortcuts”用于在桌面的开始菜单里安装一个菜单项，这个很有用，应当保持默认的勾选状态不变。至于“Desktop Icons”，则是用来在 Windows 桌面上建立应用程序图标。如果你不希望它破坏了漂亮的桌面，可以取消对它的选中。

如果你的操作系统是 Windows 7，某些版本的 NASM 编译器安装程序会提示软件兼容性的问题。在本书编写的时候，Windows 7 还是个新生事物，有些软件的更新还跟不上它的步伐。不过，这不是什么了不得的事情，你应该忽略这个善意的提醒，NASM 会工作得很好。

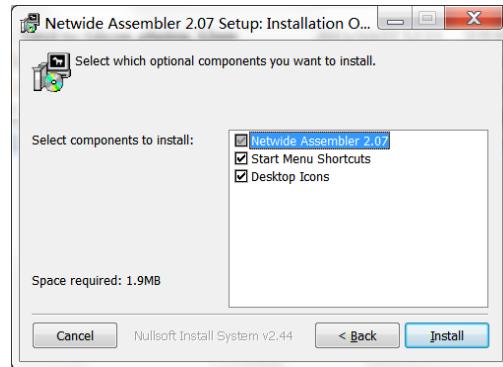


图 3-5 NASM 编译器安装选项界面

3.2.3 下载配书源码和工具

和你已经司空见惯的其他 Windows 应用程序不同，NASM 在运行之后并不会显示一个图形用户界面。相反地，它只能通过命令行使用。

比如，我们可以用 Windows 记事本编写一个汇编语言源程序，并把它保存到 NASM 工作

目录下（就是在前面安装 NASM 时所用的安装文件夹），文件名为 exam.asm。作为惯例，汇编语言源程序文件的扩展名是“.asm”，不过，你当然可以使用其他扩展名。

一旦有了一个源程序，下一步就是将它的内容编译成机器代码。为此，可以从开始菜单里找到“Netwide Assembler 2.07”，然后选择其下的“Nasm Shell”，这将打开一个命令行窗口。

接着，在命令行提示符后输入“nasm -f bin exam.asm -o exam.bin”并按 Enter 键，如图 3-6 所示。

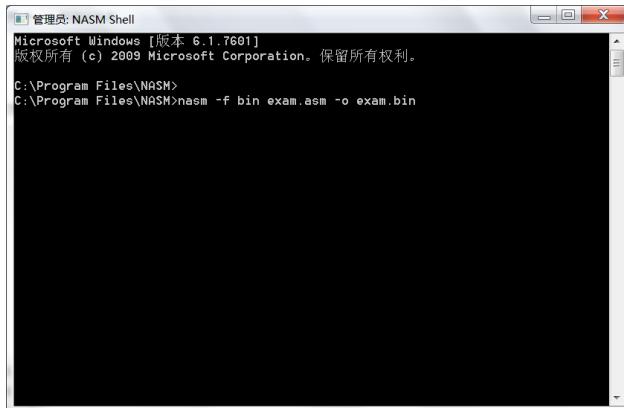


图 3-6 在命令行方式下使用 NASM 编译一个汇编源程序

NASM 需要一系列参数才能正常工作。**-f** 参数的作用是指定输出文件的格式（Format）。这样，**-f bin** 就是要求 NASM 生成的文件只包含“纯二进制”的内容。换句话说，除了处理器能够识别的机器代码外，别的任何东西都不包含。这样一来，因为缺少操作系统所需要的加载和重定位信息，它就很难在 Windows、DOS 和 Linux 上作为一个普通的应用程序运行。不过，这正是本书所需要的。

紧接着，exam.asm 是源程序的文件名，它是将要被编译的对象。

-o 参数指定编译后输出（Output）的文件名。在这里，我们要求 NASM 生成输出文件 exam.bin。

用来编写汇编语言源程序，Windows 记事本并不是一个好工具。同时，在命令行编译源程序也令很多人迷糊。毕竟，很多年轻的朋友都是用着 Windows 成长起来的，他们缺少在 DOS 和 UNIX 下工作的经历。

为了写这本书，我一直想找一个自己中意的汇编语言编辑软件。互联网是个大宝库，上面有很多这样的工具软件，但大多都包含了太多的功能，用起来自然也很复杂。我的愿望很简单，能够方便地书写汇编指令即可，同时还具有编译功能。毕竟我自己也不喜欢在命令行和图形用户界面之间来回切换。

在经历了一系列的失望之后，我决定自己写一个，于是就有了 Nasmide 这个小程序。不过遗憾的是，这个小程序却并非是用汇编语言书写的。

本书不配光盘，所以书中的所有源代码连同我自己写的小工具都只有通过网络下载方能使用。这是一个压缩文件，名字叫 booktool.zip，可以通过下面这个链接来下载它：

<http://download.csdn.net/detail/sholber/4561356>

如果此链接不可用，可以到电子工业出版社的网站上下载，或者直接给我写信，我会以最快的时间给予支持。

下载 booktool.zip 之后，需要先把它解压缩到一个文件夹里。文件并不大，所以解压缩的过程很快。完成之后，打开这个文件夹，里面的内容看起来应该像图 3-7 所示的那个样子：

其中，像 c05、c06、c07, …这些文件夹，包含的是各章的汇编源代码；nasmide.exe 就是我们现在所说的汇编小工具。

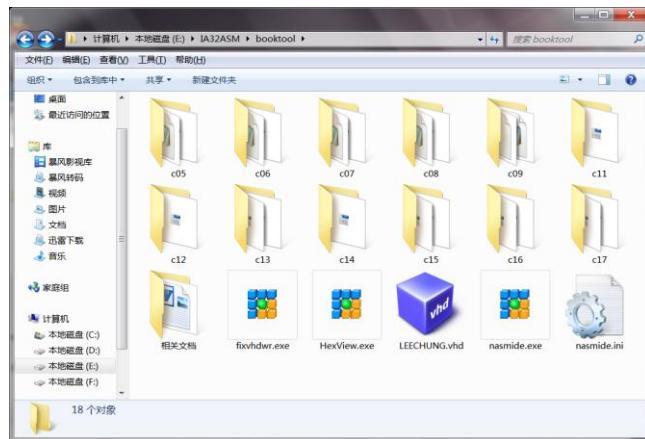


图 3-7 解压缩之后的配书源代码及工具

3.2.4 用 Nasmide 体验代码的书写和编译过程

现在，你可以双击 nasmide.exe 来运行它。启动之后，如图 3-8 所示，Nasmide 的软件界面分为三个部分。顶端是菜单，可以用来新建文件、打开文件、保存文件或者调用 NASM 来编译当前文档。

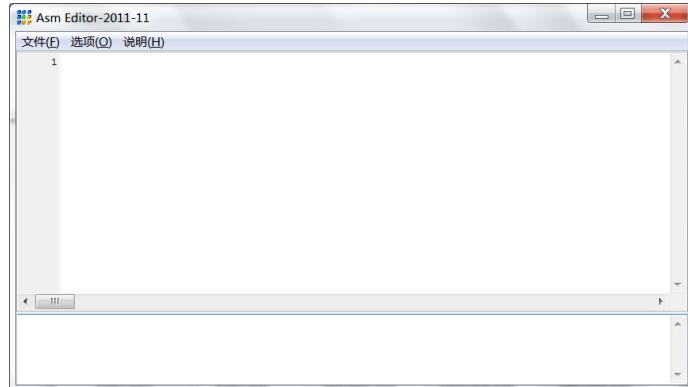


图 3-8 Nasmide 程序的基本界面

中间最大的空白区域是编辑区，用来书写汇编语言源代码。

窗口底部那个窄的区域是消息显示区。在编译当前文档时，不管是编译成功，还是发现了文档中的错误，都会显示在这里。

基本上，你现在已经可以在 Nasmide 里书写汇编语句了。不过，在此之前你最好先做一件事情。Nasmide 只是一个文本编辑工具，它自己没有编译能力。不过，它可以在后台调用 NASM 来编译当前文档，前提是它必须知道 NASM 安装在什么地方。

为此，你需要在菜单上选择“选项”→“编译环境设置”来打开如图 3-9 所示的对话框。

如图 3-9 所示，这个路径就是你在前面安装 NASM 时，指定的安装路径，包括可执行文件名 nasm.exe。

不同于其他汇编语言编译器，NASM 最让我喜欢的一个特点是允许在源程序中只包含指令，如图 3-10 所示。用过微软公司 MASM 的人都知道，在真正开始书写汇编指令前，先要穿靴戴帽，在源程序中定义很多东西，比如代码段和数据段等，弄了半天，实际上连一条指令还没开始写呢。

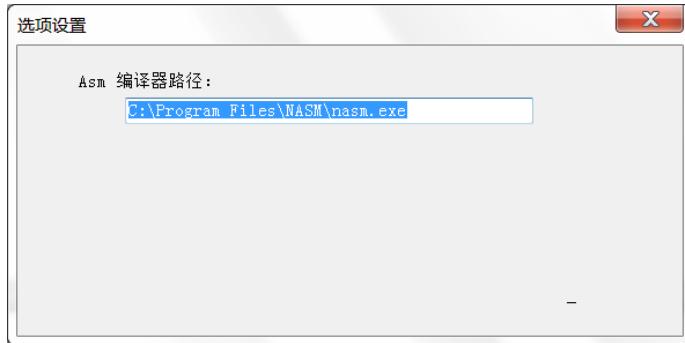


图 3-9 为 Nasmide 指定 NASM 编译器所在路径



图 3-10 NASM 允许在源文件中只包含指令

如图 3-10 所示，用 Nasmide 程序编辑源程序时，它会自动在每一行内容的左边显示行号。对于初学者来说，一开始可能会误以为行号也会出现在源程序中。不要误会，行号并非源程序的一部分，当保存源程序的时候，也不会出现在文件内容中。

让 Nasmide 显示行号，这是一个聪明的决定。一方面，我在书中讲解源程序时，可以说第几行到第几行是做什么用的；另一方面，当编译源程序的时候，如果发现了错误，错误信息中也会说明是第几行有错。这样，因为 Nasmide 显示了行号，这就很容易快速找到出错的那一行。

在汇编源程序中，可以为每一行添加注释。注释的作用是说明某条指令或者某个符号的含义和作用。注释也是源程序的组成部分，但在编译的时候会被编译器忽略。如图 3-10 所示，为了告诉编译器注释是从哪里开始的，注释需要以英文字母的分号 “;” 开始。

当源程序书写完毕之后，就可以进行编译了，方法是在 Nasmide 中选择菜单“文件”→“编译本文档”。这时，Nasmide 将会在后台调用 NASM 来完成整个编译过程，不需要你额外操心。如图 3-10 所示，即使只有三行的程序也能通过编译。编译完成后，会在窗口底部显示一条消息。

3.2.5 用 HexView 观察编译后的机器代码

编译成功完成之后，Nasmide 会在编辑窗口的底部显示相应的消息，同时显示了源文件名

称和编译之后的文件名称（含路径）。

尽管我们强调源文件和编译之后的文件具有不同的内容，但如果能用工具看一看，相信印象更为深刻。在前面下载的配书源码和工具里，有一个名为 HexView 的小程序，可以实现这个愿望。HexView 用于打开任意一个文件，以十六进制的形式从头到尾显示它每个字节的内容。

双击启动 HexView，然后选择菜单“文件”→“打开文件以显示”，在文件选择对话框里找到你在 3.2.4 节里编辑并保存的源程序文件。

如图 3-11 所示，文件选择之后，HexView 程序将以十六进制的形式显示刚刚选择的文件。

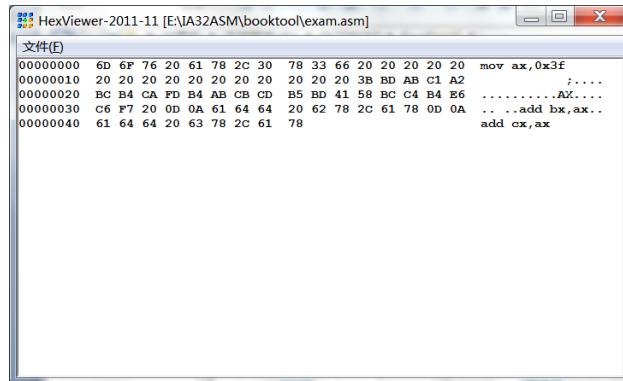


图 3-11 用 HexView 程序显示源程序文件的内容

在 HexView 中，文件的内容以十六进制的形式显示在窗口中间，以 16 个字节为一行，字节之间以空格分隔，所以看起来很稀疏。如果文件较大的话，则会分成很多行。

作为对照，每个字节还会以字符的形式显示在窗口右侧，如果它确实可显示为一个字符的话。如果该字节并非一个可以显示的字符，则显示一个替代的字符“.”。因为源程序中还有汉字注释，所以，如果细心的话，从图中可以算出每个汉字的编码是两个字节，比如“将”字的编码是 0xBD 0xAB。由于 HexView 以单字节的形式来显示每个字符，所以无法显示汉字。

左边的数字，是每一行第一个字节相对于文件头部的距离（偏移），也是以十六进制数显示的。

源程序很长，但是，编译之后的机器指令却很简短。

如图 3-12 所示，编译之后的文件只有 7 个字节，这才是处理器可以识别并执行的机器指令。

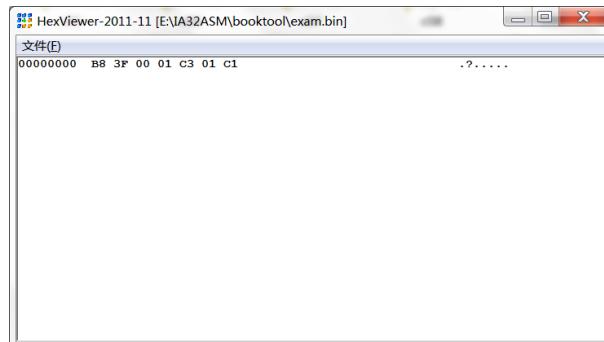


图 3-12 用 HexView 显示编译之后的文件内容

本 章 习 题

如图 3-11 所示, 请问:

- (1) 源程序共有 3 行, 每一行第一个字符的偏移分别是多少?
- (2) 该源程序文件的大小是多少字节?

第4章 虚拟机的安装和使用

4.1 计算机的启动过程

4.1.1 如何将编译好的程序提交给处理器

对于绝大多数编译好的程序来说，要想得到处理器的光顾，让它执行一下，必须借助于操作系统。就拿 Windows 来说，它为你显示每个程序的图标，允许你双击来运行它。在内部你看不见的层面上，它必须给将要运行的程序分配空闲的内存空间，并在适当的时候将程序提交给处理器执行。

每种操作系统都对它所管理的程序提出了种种格式上的要求。比如，它要求编译好的程序必须在文件的开始部分包含编译日期，是针对哪种操作系统编译的，程序的版本，第一条指令从哪里开始，数据段从哪里开始、有多长，代码段从哪里开始、有多长，等等，Windows 甚至建议你在文件中包含至少一个用于显示的图标。如果你不按它的要求来，它也不会给你面子，并直截了当地弹出一个对话框，如图 4-1 所示，告诉你它不准备，也没办法将你的程序提交给处理器。

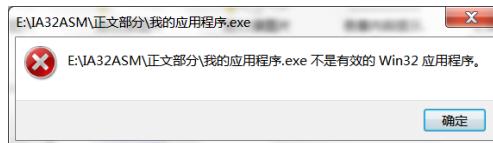


图 4-1 每种操作系统都会定义它自己的可执行文件格式

每种编译器都有能力针对不同的操作系统来生成不同格式的二进制文件，程序员所做的，就是在源程序中加入一些相关的信息，比如指定每个段的开始和结束，并在编译时指定适当的参数。如果你对此感兴趣，可以阅读 NASM 文档。这是一个 PDF 文件，在安装 NASM 的时候，它也会被安装。

在特定的操作系统上开发软件肯定不是一件容易的事情。但换个角度考虑一下，操作系统也是一个需要在处理器上运行的软件，只不过比起一般的程序而言，体积更为庞大，功能更为复杂而已。如果我们能绕过它，或者代替它，让计算机一开机的时候直接执行我们自己的软件，岂不更简单？

好，这个主意完全可行。那就让我们慢慢开始吧。

4.1.2 计算机的加电和复位

在处理器众多的引脚中，有一个是 RESET，用于接受复位信号。每当处理器加电，或者 RESET 引脚的电平由低变高时^①，处理器都会执行一个硬件初始化，以及一个可选的内部自测

^① 比如，当你按下主机箱面板上的 RESET 按钮时，就会导致 RESET 引脚电平的变化，从而使计算机热启

试 (Build-in Self-Test, BIST)，然后将内部所有寄存器的内容初始到一个预置的状态。

比如，对于 Intel 8086 来说，复位将使代码段寄存器 (CS) 的内容为 0xFFFF，其他所有寄存器的内容都为 0x0000，包括指令指针寄存器 (IP)。8086 之后的处理器并未延续这种设计，但毫无疑问，无论怎么设计，都是有目的的。

处理器的主要功能是取指令和执行指令，加电或者复位之后，它就会立刻尝试去做这样的工作。不过，在这个时候，内存中还没有任何有意义的指令和数据，它该怎么办呢？

在揭开谜底之前，我们先来看看内存的特点。

为了节约成本，并提高容量和集成度，在内存中，每个比特的存储都是靠一个极其微小的晶体管，外加一个同样极其微小的电容来完成的。可以想象，这样微小的电容，其泄漏电荷的速度当然也非常快。所以，个人计算机中使用的内存需要定期补充电荷，这称为刷新，所以这种存储器也称为动态随机访问存储器 (Dynamic Random Access Memory, DRAM)。随机访问的意思是，访问任何一个内存单元的速度和它的位置 (地址) 无关。举个例子来说，从头至尾在一盘录音带上找某首歌曲，它越靠前，找到它所花的时间就越短。但内存就不一样，读写地址为 0x00001 的内存单元，和读写地址为 0xFFFF0 的内存单元，所需要的时间是一样的。

在内存刷新期间，处理器将无法访问它。这还不是最麻烦的，最麻烦的是，在它断电之后，所有保存的内容都会统统消失。所以，每当处理器加电之后，它无法从内存中取得任何指令。

4.1.3 基本输入输出系统

Intel 8086 可以访问 1MB 的内存空间，地址范围为 0x00000 到 0xFFFFF。出于各方面的考虑，计算机系统的设计者将这 1MB 的内存空间从物理上分为几个部分。

8086 有 20 根地址线，但并非全都用来访问 DRAM，也就是内存条。事实上，这些地址线经过分配，大部分用于访问 DRAM，剩余的部分给了只读存储器 ROM 和外围的板卡，如图 4-2 所示。

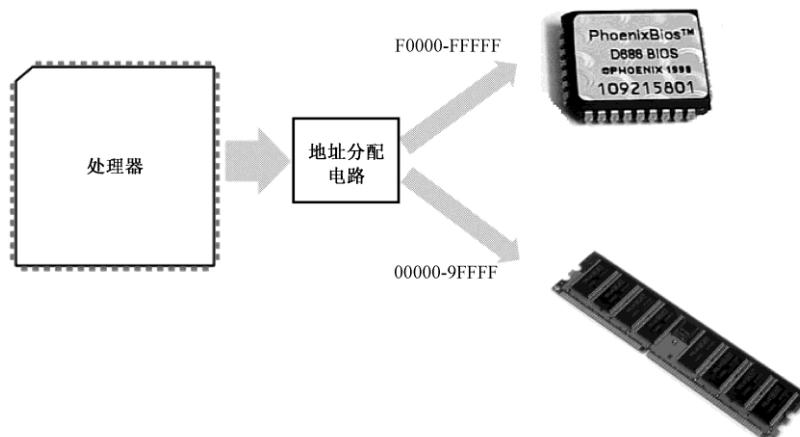


图 4-2 8086 系统的内存空间分配

与 DRAM 不同，只读存储器 (Read Only Memory, ROM) 不需要刷新，它的内容是预先写入的，即使掉电也不会消失，但也很难改变。这个特点很有用，比如，可以将一些程序指令

动。

固化在 ROM 中，使处理器在每次加电时都自动执行。处理器醒来后不能饿着，这是很重要的。

在以 Intel 8086 为处理器的系统中，ROM 占据着整个内存空间顶端的 64KB，物理地址范围是 0xF0000~0xFFFFF，里面固化了开机时要执行的指令；DRAM 占据着较低端的 640KB，地址范围是 0x00000~0x9FFFF；中间还有一部分，分给了其他外围设备，这个以后再说。因为 8086 加电或者复位时，CS=0xFFFF，IP=0x0000，所以，它取的第一条指令位于物理地址 0xFFFF0，正好位于 ROM 中，那里固化了开机时需要执行的指令。

处理器取指令执行的自然顺序是从内存的低地址往高地址推进。如果从 0xFFFF0 开始执行，这个位置离 1MB 内存的顶端（物理地址 0xFFFFF）只有 16 个字节的长度，一旦 IP 寄存器的值超过 0x0000F，比如 IP=0x0011，那么，它与 CS 一起形成的物理地址将因为溢出而变成 0x00001，这将回绕到 1MB 内存的最低端。

所以，ROM 中位于物理地址 0xFFFF0 的地方，通常是一个跳转指令，它通过改变 CS 和 IP 的内容，使处理器从 ROM 中的较低地址处开始取指令执行。在 NASM 汇编语言里，一个典型的跳转指令像这样：

```
jmp 0xf000:0xe05b
```

在这里，“jmp”是跳转（jump）的简化形式；0xf000 是要跳转到的段地址，用来改变 CS 寄存器的内容；0xe05b 是目标代码段内的偏移地址，用来改变 IP 寄存器的内容。一旦执行这条指令，处理器将开始从指定的“段：偏移”处开始重新取指令执行。

到了本书第 5 章我们就能接触跳转指令了，现在，我们只需要知道，指令的执行并非总是顺序的，有时候不得不根据某些条件来选择执行哪些指令，不执行哪些指令。这个时候，跳转指令是很常用的。

这块 ROM 芯片中的内容包括很多部分，主要是进行硬件的诊断、检测和初始化。所谓初始化，就是让硬件处于一个正常的、默认的工作状态。最后，它还负责提供一套软件例程，让人们在不必了解硬件细节的情况下从外围设备（比如键盘）获取输入数据，或者向外围设备（比如显示器）输出数据。设备当然是很多的，所以这块 ROM 芯片只针对那些最基本的、对于使用计算机而言最重要的设备，而它所提供的软件例程，也只包含最基本、最常规的功能。正因为如此，这块芯片又叫基本输入输出系统（Base Input & Output System, BIOS）ROM。在读者缺乏基础知识的情况下讲述 ROM-BIOS 的工作只会越讲越糊涂，所以这些知识将会分散在各个章节里予以讲解。

ROM-BIOS 的容量是有限的，当它完成自己的使命后，最后所要做的，就是从辅助存储设备读取指令数据，然后转到那里开始执行。基本上，这相当于接力赛中的交接棒。

4.1.4 硬盘及其工作原理

历史上，有多种辅助存储设备，比如软盘、光盘、硬盘、U 盘等，相对于内存，它们就是人们常说的“外存”，即外存储器（设备）。

从软盘（Floppy Disk）启动计算机，这已经是过去的事了。软盘的尺寸比烟盒稍大一点，但是比较薄，采用塑料作为基片，上面是一层磁性物质，可以用来记录二进制位。这种塑料介质比较柔软，所以称为软盘。

在数据记录原理上和软盘很相似的设备是硬盘（Hard Disk, HDD），而且它们几乎是同一个时代的产物。但是，与软盘不同，硬盘是多盘片、密封、高转速的，采用铝合金作为基片，并在表面涂上磁性物质来记录二进制位。这就使得它的盘片具有较高的硬度，故称为硬盘。

如图 4-3 所示，这是一块被拆开的硬盘，中间是用于记录数据的铝合金盘片，固定在中心

的轴上，由一个高速旋转的马达驱动。附着在盘片表面的扁平锥状物，就是用于在盘片上读写数据的磁头。

为了进一步搞清楚硬盘的内部构造，图 4-4 给出了更为详细的图示。



图 4-3 一块被拆开密封盖的硬盘

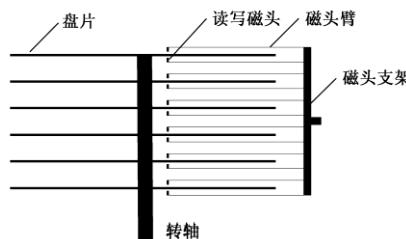


图 4-4 硬盘的结构示意图

硬盘可以只有一个盘片（这称为单碟），也可能有好几个盘片。但无论如何，它们都串在同一个轴上，由电动机带动着一起高速旋转。一般来说，转速可以达到每分钟 3600 转或者 7200 转，有的能达到一万多转，这个参数就是我们常说的“转/分钟”（Round Per Minute, RPM）。

每个盘片都有两个磁头（Head），上面一个，下面一个，所以经常用磁头来指代盘面。磁头都有编号，第 1 个盘片，上面的磁头编号为 0，下面的磁头编号为 1；第 2 个盘片，上面的磁头编号为 2，下面的磁头编号为 3，依次类推。

每个磁头不是单独移动的。相反，它们都通过磁头臂固定在同一个支架上，由步进电动机带动着一起在盘片的中心和边缘之间来回移动。也就是说，它们是同进退的。步进电动机由脉冲驱动，每次可以旋转一个固定的角度，即可以步进一次。

可以想象，当盘片高速旋转时，磁头每步进一次，都会从它所在的位置开始，绕着圆心“画”出一个看不见的圆圈，这就是磁道（Track）。磁道是数据记录的轨迹。因为所有磁头都是联动的，故每个盘面上的同一条磁道又可以形成一个虚拟的圆柱，称为柱面（Cylinder）。

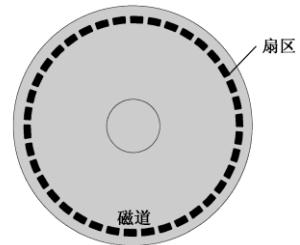
磁道，或者柱面，也要编号。编号是从盘面最边缘的那条磁道开始，向着圆心的方向，从 0 开始编号。

柱面是一个用来优化数据读写的概念。初看起来，用硬盘来记录数据时，应该先将一个盘面填满后，再填写另一个盘面。实际上，移动磁头是一个机械动作，看似很快，但对处理器来说，却很漫长，这就是寻道时间。为了加速数据在硬盘上的读写，最好的办法就是尽量不移动磁头。这样，当 0 面的磁道不足以容纳要写入的数据时，应当把剩余的部分写在 1 面的同一磁道上。如果还写不下，那就继续把剩余的部分写在 2 面的同一磁道上。换句话说，在硬盘上，数据的访问是以柱面来组织的。

实际上，磁道还不是硬盘数据读写的最小单位，磁道还要进一步划分为扇区（Sector）。磁道很窄，也看不见，但在想象中，它仍呈带状，占有一定的宽度。将它划分许多分段之后，每一部分都呈扇形，这就是扇区的由来。

每条磁道能够划分为几个扇区，取决于磁盘的制造者，但通常为 63 个。而且，每个扇区都有一个编号，与磁头和磁道不同，扇区的编号是从 1 开始的。

扇区与扇区之间以间隙（空白）间隔开来，每个扇区以扇区头开始，然后是 512 个字节的



数据区。扇区头包含了每个扇区自己的信息，主要有本扇区的磁道号、磁头号和扇区号，用来供硬盘定位机构使用。现代的硬盘还会在扇区头部包括一个指示扇区是否健康的标志，以及用来替换该扇区的扇区地址。用于替换扇区的，是一些保留和隐藏的磁道。

4.1.5 一切从主引导扇区开始

尽管我们使用硬盘的历史很长，但它一直没能退出舞台，这主要是因为它总能通过不断提高自己的容量来打败那些竞争者。20世纪90年代初，40MB的硬盘算是常见的，能拥有200MB的硬盘很让人羡慕。看看现在，500GB的硬盘也不算稀罕，而且价钱也很便宜。

前面说到，当ROM-BIOS完成自己的使命之前，最后要做的一件事是从外存储设备读取更多的指令来交给处理器执行。现实的情况是，绝大多数时候，对于ROM-BIOS来说，硬盘都是首选的外存储设备。

硬盘的第一个扇区是0面0道1扇区，或者说是0头0柱1扇区，这个扇区称为主引导扇区。如果计算机的设置是从硬盘启动，那么，ROM-BIOS将读取硬盘主引导扇区的内容，将它加载到内存地址0x0000:0x7c00处（也就是物理地址0x07C00），然后用一个jmp指令跳到那里接着执行：

```
jmp 0x0000:0x7c00
```

为什么偏偏是0x7c00这个地方？还不太清楚。反正当初定下这个方案的家伙已经被人说了很多坏话，我也不准备再多说什么了。

通常，主引导扇区的功能是继续从硬盘的其他部分读取更多的内容加以执行。像Windows这样的操作系统，就是采用这种接力的方法一步一步把自己运行起来的。

说到这里，我们可以想象，如果我们把自己编译好的程序写到主引导扇区，不也能够让处理器执行吗？

对于这种想法，我有一个好消息和一个坏消息要告诉你。

好消息是，这是可以的，而且这几乎是在不依赖操作系统的情况下，让我们的程序可以执行的唯一方法。

不过，坏消息是，如果你改写了硬盘的主引导扇区，那么，Windows和Linux，以及任何你正在使用的操作系统都会瘫痪，无法启动了。

那么，我们该怎么办呢？答案是在你现有的计算机上，再虚拟出一台计算机来。

4.2 创建和使用虚拟机

4.2.1 别害怕，虚拟机是软件

对于第一次听说虚拟机（Virtual Machine, VM）的人来说，可能以为还要再花钱买一台计算机，这恐怕是他们最担心的。所谓虚拟机，就是在你的计算机上再虚拟出另一台计算机来。这台虚拟出来的计算机，和真正的计算机一样，可以启动，可以关闭，还可以安装操作系统、安装和运行各种各样的软件，或者访问网络。总之，你在真实的计算机上能做什么，在它里面一样可以那么做。使用虚拟机，你会发现，在Windows操作系统里，居然又可以拥有另一套Windows。然而本质上，它只是运

行在物理计算机上的一个软件程序。

如图 4-5 所示，整个大的背景，是 Windows 7 的桌面，它安装在一台真实的计算机上。图中的小窗口，正是虚拟机，运行的是 Windows Server 2003。像这样，我们就得到了两台“计算机”，而且它们都可以操作。

虚拟机仅仅是一个软件，运行在各种主流的操作系统上。它以自己运行的真实计算机为模板，虚拟出另一套处理器、内存和外围设备来。它的处理能力，完全来自于背后那台真实的计算机。

尤其重要的是，针对某种真实处理器所写的任何指令代码，都可以正确无误地在该处理器的虚拟机上执行。实际上，这也是虚拟机具有广泛应用价值的原因所在。



图 4-5 虚拟机的实例

在过去的若干年里，虚拟机得到了广泛应用。为了研制防病毒软件、测试最新的操作系统或者软件产品，软件公司通常需要多台用于做实验的计算机。采用虚拟机，就可以避免反复重装软件系统的麻烦，当这些软件系统崩溃时，崩溃的只是虚拟机，而真实的物理计算机丝毫不受影响。

利用虚拟机来教学，本书不是第一个，国内外都流行这种教学方式。虚拟机利用软件来模拟完整的计算机系统，无须添加任何新的设备，而且与主计算机系统是隔离的，在虚拟机上的任何操作都不会影响到物理计算机上的操作系统和软件，这对拥有大量计算机的培训机构来说，可以极大地节省维护上的成本。

4.2.2 下载 Oracle VM VirtualBox

主流的虚拟机软件包括 VMWare、Virtual PC 和 VirtualBox，但只有 VirtualBox 是开源和免费的。

要使用 VirtualBox，首先必须从网上下载并安装它。这里是它的主页：

<https://www.virtualbox.org/>

通过这个主页，你可以找到最新的版本并下载它。为了方便，下面给出下载页面的链接：

<https://www.virtualbox.org/wiki/Downloads>

这个链接将带你到达类似于图 4-6 所示的这个页面。通常，我们应该在运行着 Windows 的主机上安装使用 VirtualBox，所以应当选择“VirtualBox 4.1.6 for Windows hosts”。当然，当本书出版的时候，

版本号可能已经不是 4.1.6 了，这个数字无关紧要，要选择最新的版本。

4.2.3 安装 Oracle VM VirtualBox

相对于前面安装的 NASM，VirtualBox 安装程序稍大些，4.1.6 版本有 90MB。安装过程也很简单，唯一需要说明的是软件特性的选择和安装路径，如图 4-7 所示。



图 4-6 VirtualBox 下载页面

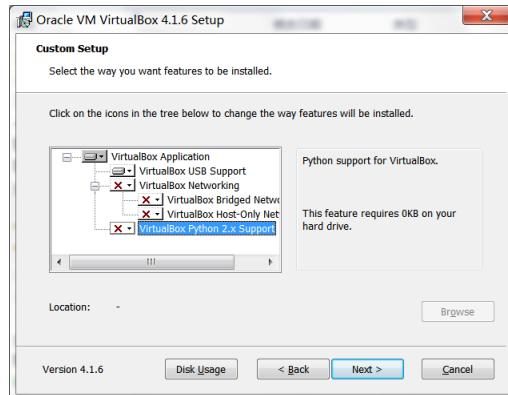


图 4-7 VirtualBox 安装选项

在这里，“VirtualBox Application”是虚拟机的主体部分，当然是必选的。通用串行总线(Universal Serial Bus, USB) 控制器也是必须安装的，我们可能要针对 USB 设备编写汇编语言程序，没有这个虚拟的“芯片”可不行。所以，应当选择完全安装“VirtualBox USB Support”(VirtualBox USB 支持)。

“VirtualBox Networking”特性用于使虚拟机提供对网络的支持。如果仅仅是通过本书学习汇编语言，不干别的，这个特性可以不用安装。但如果你想在虚拟机里安装其他操作系统，探索虚拟机的功能，还想在虚拟机里上网，也可以选择安装。

除了手工操作之外，VirtualBox 允许通过编程来完全控制虚拟机的行为。就像所有在 Windows 上运行的软件都可以调用操作系统提供的例程和服务一样，VirtualBox 也提供这样的手段。但是，不像 C++ 这样的编程语言，Python 这样的脚本语言接口并没有内置于虚拟机中。所以，如果你想用

Python 脚本语言来访问虚拟机，那么，就应当选择安装“VirtualBox Python 2.x Support”。当然，对于本书的读者来说，可以选择不安装这个特性。

4.2.4 创建一台虚拟 PC

安装之后，第一次启动时的 VirtualBox 如图 4-8 所示。

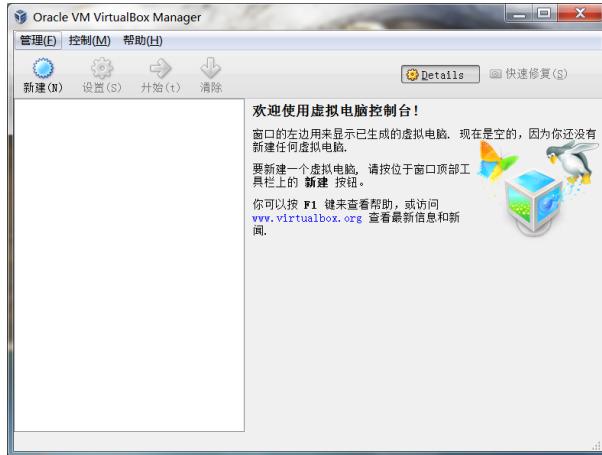


图 4-8 第一次启动时的 VirtualBox

你可能以为这个界面就是虚拟出来的计算机，其实不是。

这只是 VirtualBox 的控制台。要知道，VirtualBox 可以虚拟出多台计算机，而不仅仅是一台。所以，现在的任务是不花一分钱，不用走出家门，来安装一台“全新的计算机”。

要创建一台新的虚拟计算机，应该单击控制台界面上的“新建”按钮，或者选择菜单“控制”→“新建”。这时，会出现“欢迎使用新建虚拟电脑向导”，此时可单击“下一步”按钮。

如图 4-9 所示，紧接着，向导程序将询问这台计算机的名称和将要采用的操作系统。



图 4-9 填写计算机名称，并选择要在这台计算机上安装的操作系统

正如向导界面上的文字所描述的那样，计算机名称用来唯一地标识一台虚拟计算机。因为我们安装虚拟机的目的是学习汇编语言，那么，我们可以为这台计算机起个名字，叫“LEARN-ASM”。事实上，你可以取别的名字，只要你喜欢，这没有什么关系。

操作系统类型和版本的选择部分容易让人产生误解，以为VirtualBox会根据你的选择来安装一个现成的操作系统。实际上，这不可能。让你选择操作系统的唯一目的，是想根据你的选择，在后面的步骤中为你提供合理的硬件配置，比如内存容量和硬盘大小等。实际上，我们不准备安装任何操作系统，所以在“操作系统”一栏里选择“Other”（其他）；在“版本”一栏里选择“Other/Unknown”（其他/未知）。

一旦做出这种选择之后，紧接着，在下一步里，向导程序会结合真实主机的内存容量，以及你所选择的操作系统，来给出一个建议的内存容量配置。

如图4-10所示，在我的计算机上，它给出的建议值是64MB内存，因为我的主机上只有1GB的物理内存容量（从图中就可以看出）。当然，它允许你拖动滑块来调整这个数值。

调整好虚拟机的内存容量后，继续下一步。

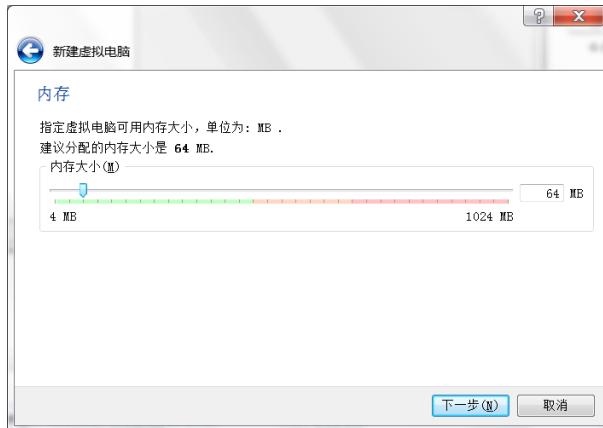


图4-10 调整虚拟计算机的内存容量

和真实的计算机一样，虚拟机也需要一个或几个辅助存储器（磁盘、光盘、U盘等）才能工作。不过，为它配备的并非真正的盘片，而是一个特殊的文件，故称为虚拟盘。这样，当一个软件程序在虚拟机里读写硬盘或者光盘时，虚拟机将把它转换成对文件的操作，而软件程序还以为自己真的是在读写物理盘片。在需要的时候随时创建，不需要时可以随时删除，这真是非常神奇的磁盘。

现在，当调整好虚拟机的内存容量后，下一步，将要为虚拟机配备虚拟盘。如图4-11所示，因为在正常情况下，所有的计算机都习惯从硬盘启动，故这个界面默认的是选择一个虚拟硬盘（图4-11）。

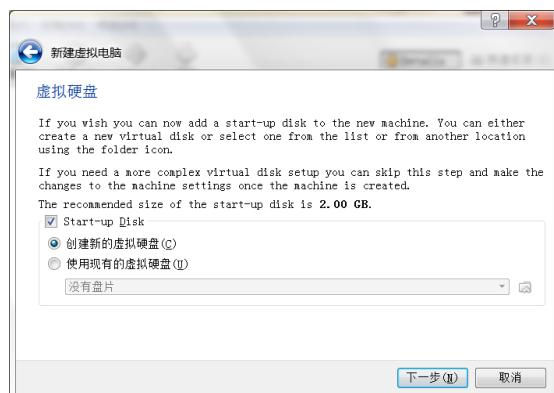


图4-11 为虚拟机配备虚拟盘

在这个界面上，你有两种选择，创建新的虚拟硬盘，或者使用现有的虚拟硬盘。基本上，你采

用哪种方式都可以。注意，那个复选框“Start-up Disk”用于指定是否从该硬盘启动。如果选择了它，那么，ROM-BIOS 程序将在开机自检后从这个硬盘里读取主引导扇区的内容。

如果你要创建新的虚拟硬盘，只需要单击“下一步”按钮。

除此之外，你还有另一个选择。前面你已经从网上下载了与本书配套的源码和工具，那是个压缩文件。解压之后，里面有一个现成的虚拟硬盘文件，文件名是 LEECHUNG.VHD，这是给你额外准备的，而且经过了测试，可以在你无法创建虚拟硬盘的时候派上用场。要选用这个虚拟硬盘，可以选择“使用现有的虚拟硬盘”，然后单击下拉列表框右边的小图标，在弹出的文件选择对话框里找到 LEECHUNG.VHD，并选择它。

当然，如果你选择的是“创建新的虚拟硬盘”，那后面的事情就要麻烦得多，一旦进入下一个步骤，向导程序将询问你想创建什么类型的虚拟硬盘，如图 4-12 所示。



图 4-12 虚拟硬盘类型选择

正如前面所说的，市面上有好几种流行的虚拟机软件，而每种虚拟机软件都企图制定自己的虚拟硬盘标准。因为虚拟硬盘实际是一个文件，所以，所谓虚拟硬盘标准，实际上就是该文件的格式。正是因为这样，虚拟硬盘类型说白了就是你准备采用哪家的虚拟硬盘文件格式。

因为虚拟硬盘实际上是一个文件，所以，通常来说，它的格式体现在它的文件扩展名上。比如上面的 LEECHUNG.VHD，采用的就是微软公司的 VHD 虚拟硬盘规范。VHD 规范最早起源于 Connectix 公司的虚拟机软件 Connectix Virtual PC，2003 年，微软公司收购了它并改名为 Microsoft Virtual PC。2006 年，微软公司正式发布了 VHD 虚拟硬盘格式规范。在本书配套的源代码和工具包里，有该规范的文档。

VDI 是 VirtualBox 自己的虚拟硬盘规范，VMDK 是 VMWare 的虚拟硬盘规范。采用哪个公司、哪个虚拟机软件的虚拟硬盘格式，对于普通的应用来说，这没什么关系，它们都能很好地工作。但是，对于本书和本书配套的工具来说，你必须选择“VHD (Virtual Hard Disk)”。具体原因，我们将在下一节讲述。

事实上，即使是 VHD，也分为两种类型：固定尺寸的和动态分配的。一个固定尺寸的 VHD，它对应的文件尺寸和该虚拟硬盘的容量是相同的，或者说是一次性分配够了的。比如，一个 2GB 的 VHD 虚拟硬盘，它对应的文件大小也是 2GB。

与此相反，一个动态分配的 VHD，它的文件尺寸是根据需要不断增长的，它的大小等于实际写入该虚拟硬盘上的数据量。

如图 4-13 所示，本书以及本书配套的工具仅支持固定尺寸的 VHD，所以你应该在进入这个界面之后选择“Fixed size”。



图 4-13 选择 VHD 硬盘类型

在选择使用 VHD 之后，还要指定该 VHD 的容量。如图 4-14 所示，你可以拖动滑块，在 4MB 和 2TB 之间随意指定一个容量。注意，1TB = 1024GB。



图 4-14 指定 VHD 的容量

不得不提醒你的是，应当指定 50MB 以上的硬盘大小，这是本书对你的要求。不过，也不需要太大，毕竟你的物理硬盘空间也一定很紧张。

除了指定虚拟硬盘的容量，另一个值得特别注意的问题是该虚拟硬盘的创建位置。默认情况下，它会被放在 Windows 用户文件夹下，而且对于初学者来说很不容易找到。其实，把它创建在配书工具所在的文件夹里是最方便的，因为我们以后要反复对它进行写入操作。为此，如图 4-14 所示，请在“位置”一栏，单击文本框右边的小图标，来选择一个容易找到的位置，比如配书工具所在的文件夹。

以上就是创建一台虚拟机要经历的步骤。当结束向导程序时，刚刚创建的虚拟机 LEARN-ASM 就会显示在 VirtualBox 控制台里，如图 4-15 所示。基本上，你现在就可以单击控制台界面上的“开始”来启动这台虚拟机。但是，别忙，你的虚拟硬盘里还没有东西呢。

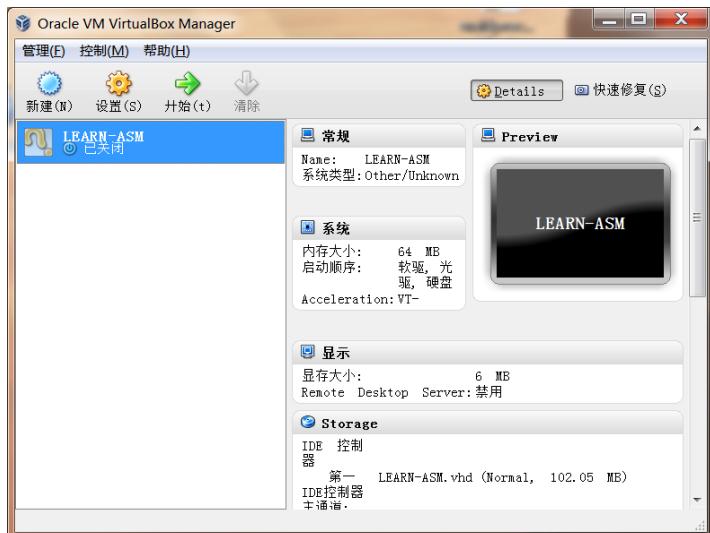


图 4-15 通过向导程序创建的 LEARN-ASM 虚拟机

4.2.5 虚拟硬盘简介

坦白地说，之所以要采用固定尺寸的 VHD 虚拟硬盘，是因为其简单性。我们知道，虚拟硬盘实际上是一个文件。固定尺寸的 VHD 虚拟硬盘是一个具有“.vhd”扩展名的文件，它仅包括两个部分，前面是数据区，用来模拟实际的硬盘空间，后面跟着一个 512 字节的结尾（2004 年前的规范里只有 511 字节）。

要访问硬盘，运行中的程序必须至少向硬盘控制器提供 4 个参数，分别是磁头号、磁道号、扇区号，以及访问意图（是读还是写）。

硬盘的读写是以扇区为最小单位的。所以，无论什么时候，要从硬盘读数据，或者向硬盘写数据，至少得是 1 个扇区。

你可能想，我只有 2 字节的数据，不足以填满一个扇区，怎么办呢？

这是你自己的事。你可以用无意义的废数字来填充，凑够一个扇区的长度，然后写入。读取的时候也是这样，你需要自己跟踪和把握从扇区里读到的数据，哪些是你真正想要的。换句话说，硬盘只是机械和电子的组合，它不会关心你都写了些什么。要是手机像人类一样智能，它一定会在坏人使用它的時候无法开机。

在 VHD 规范里，每个扇区是 512 字节。VHD 文件一开始的 512 字节，就对应着物理硬盘的 0 面 0 道 1 扇区。然后，VHD 文件的第二个 512 字节，对应着 0 面 0 道 2 扇区，后面的依次类推，一直对应到 0 面 0 道 n 扇区。这里， n 等于每磁道的扇区数。

再往后，因为硬盘的访问是按柱面进行的，所以，在 VHD 文件中，紧接着前面的数据块，下一个数据块对应的是 1 面 0 道 1 扇区，就这样一直往后排列，当把第一个柱面全部对应完后，再从第二个柱面开始对应。

如图 4-16 所示，为了标志一个文件是 VHD 格式的虚拟硬盘，并为使用它的虚拟机提供该硬盘的参数，在 VHD 文件的结尾，包含了 512 字节的格式信息。为了观察这些信息，我们使用了前面已经介绍过的配书工具 HexView。

如图 4-16 所示，文件尾信息是以一个字符串“conectix”开始的。这个标志用来告诉试图打开它的虚拟机，这的确是一个合法的 VHD 文件。该标志称为 VHD 创建者标识，就是说，该公司

(connectix) 创建了 VHD 文件格式的最初标准。

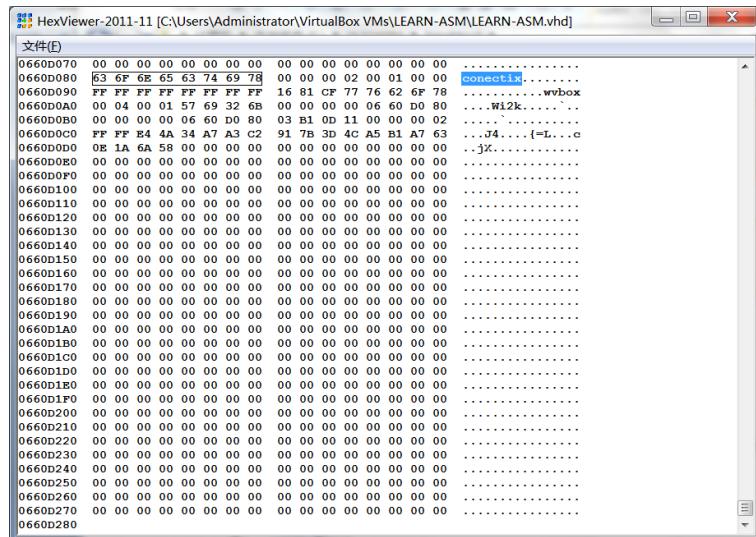


图 4-16 VHD 文件的格式信息

从这个标志开始，后面的数据包含了诸如文件的创建日期、VHD 的版本、创建该文件的应用程序名称和版本、创建该文件的应用程序所属的操作系统、该虚拟硬盘的参数（磁头数、每面磁道数、每磁道扇区数）、VHD 类型（固定尺寸还是动态增长）、虚拟硬盘容量等。

说到这里，也许你已经明白我为什么要在书中使用固定尺寸的 VHD。是的，因为它简单。为了学习汇编语言，我们不得不在硬盘上直接写入程序。因为 VHD 格式简单，所以我只花很少的时间就开发了一个虚拟硬盘写入程序，作为配书工具让大家使用，这就是下一节将要介绍的 FixVhdWr。

至于为什么要使用 VirtualBox 虚拟机，是因为它支持 VHD，而且是免费的。先前版本的 VirtualBox 可以识别 VHD，但不支持创建新的 VHD，尽管微软公司很早就公开了 VHD 规范。好消息是现在的 VirtualBox 也可以创建 VHD 了。

4.2.6 练习使用 FixVhdWr 工具向虚拟硬盘写数据

通常，VHD 是由虚拟机 VirtualBox 使用的。应用程序像往常一样，直接针对硬盘进行操作，而在底层，虚拟机将这些硬件访问转化为对文件的读写。

为了在处理器加电或者复位之后能够执行我们写的程序，势必要将这些程序写到硬盘的主引导扇区里，也就是 0 面 0 道 1 扇区，即使是在虚拟机工作环境中，也是这样。

为了做到这一点，需要一个专门针对虚拟硬盘进行读写的工具。我自己写了一个，就在配书源代码和工具里，名叫 FixVhdWr。

FixVhdWr 只针对固定尺寸的 VHD。当它启动之后，首先需要选择要读写的 VHD 文件。如图 4-17 所示，一旦这是个合法的 VHD 文件，它将读取该文件的结尾，并显示该虚拟硬盘的信息。

注意，因为 FixVhdWr 只针对固定尺寸的 VHD，所以，如果它检测到该 VHD 是一个动态虚拟硬盘，则“下一步”按钮处于禁止状态。

第二步是选择要写入虚拟硬盘的数据文件。毕竟，在任何操作系统中，数据都是以文件的方式组织的，如图 4-18 所示。



图 4-17 打开 VHD 文件并显示该虚拟硬盘的信息

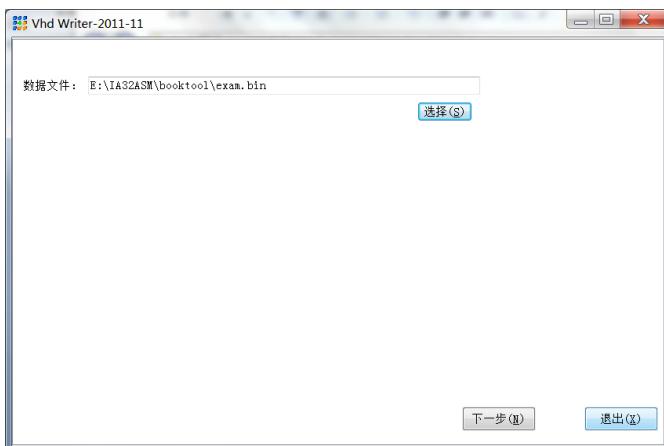


图 4-18 选择要写入虚拟硬盘的文件

最后一个界面，是执行写入操作，如图 4-19 所示，你应该选择第一种写入方式，即“LBA 连续直写模式”，并指定起始的逻辑扇区号。



图 4-19 指定数据写入时的起始逻辑扇区号

通常，一个扇区的尺寸是 512 字节，可以看成一个数据块。所以，从这个意义上来说，硬盘是一个典型的块（Block）设备。

采用磁头、磁道和扇区这种模式来访问硬盘的方法称为 CHS 模式，但不是很方便。想想看，如果有一大堆数据要写，还得注意磁头号、磁道号和扇区号不要超过界限。所以，后来引入了逻辑块地址（Logical Block Address，LBA）的概念。现在市场上销售的硬盘，无论是哪个厂家生产的，都支持 LBA 模式。

LBA 模式是由硬盘控制器在硬件一级上提供支持，所以效率很高，兼容性很好。LBA 模式不考虑扇区的物理位置（磁头号、磁道号），而是把它们全部组织起来统一编号。在这种编址方式下，原先的物理扇区被组织成逻辑扇区，且都有唯一的逻辑扇区号，

比如，某硬盘有 6 个磁头，每面有 1000 个磁道，每磁道有 17 个扇区。那么：

逻辑 0 扇区对应着 0 面 0 道 1 扇区；

逻辑 1 扇区对应着 0 面 0 道 2 扇区；

.....

逻辑 16 扇区对应着 0 面 0 道 17 扇区；

逻辑 17 扇区对应着 1 面 0 道 1 扇区；

逻辑 18 扇区对应着 1 面 0 道 2 扇区；

.....

逻辑 33 扇区对应着 1 面 0 道 17 扇区；

逻辑 34 扇区对应着 2 面 0 道 1 扇区；

逻辑 35 扇区对应着 2 面 0 道 2 扇区；

.....

逻辑 101999 扇区对应着 5 面 999 道 17 扇区，这也是整个硬盘上最后一个物理扇区。

这里面的计算方法是：

$$\text{LBA} = C \times \text{磁头总数} \times \text{每道扇区数} + H \times \text{每道扇区数} + (S - 1)$$

这里，LBA 是逻辑扇区号，C、H、S 是想求得逻辑扇区号的那个物理扇区所在的磁道、磁头和扇区号。

采用 LBA 模式的好处是简化了程序的操作，使得程序员不用关心数据在硬盘上的具体位置。对于本书来说，VHD 文件是按 LBA 方式组织的，一开始的 512 字节就是逻辑 0 扇区，然后是逻辑 1 扇区；最后一个逻辑扇区排在文件的最后（最后 512 个字节除外，那是 VHD 文件的标识部分）。

第 2 部分

8086 模式



第 5 章 编写主引导扇区代码

5.1 欢迎来到主引导扇区



本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：5-1（主引导扇区程序）

源程序文件：c05_mbr.asm

在前面的预备知识里，我们已经知道，处理器加电或者复位之后，如果硬盘是首选的启动设备，那么，ROM-BIOS 将试图读取硬盘的 0 面 0 道 1 扇区。传统上，这就是主引导扇区（Main Boot Sector, MBR）。

读取的主引导扇区数据有 512 字节，ROM-BIOS 程序将它加载到逻辑地址 0x0000:0x7c00 处，也就是物理地址 0x07c00 处，然后判断它是否有效。

一个有效的主引导扇区，其最后两字节应当是 0x55 和 0xAA。ROM-BIOS 程序首先检测这两个标志，如果主引导扇区有效，则以一个段间转移指令 jmp 0x0000:0x7c00 跳到那里继续执行。

一般来说，主引导扇区是由操作系统负责的。正常情况下，一段精心编写的主引导扇区代码将检测用来启动计算机的操作系统，并计算出它所在的硬盘位置。然后，它把操作系统的自举代码加载到内存，也用 jmp 指令跳转到那里继续执行，直到操作系统完全启动。

在本章中，我们将试图写一段程序，把它编译之后写入硬盘的主引导扇区，然后让处理器执行。当然，仅仅执行还不够，还必须在屏幕上显示点什么，要不然的话，谁知道我们的程序是不是成功运行了呢？

通过本章的学习，我们可以对处理器如何执行指令、如何访问内存以及如何进行算术逻辑运算有一个最基本的认知。

5.2 注 释

如本章代码清单 5-1 所展示的那样，在汇编语言源程序里，注释用于说明本程序的用途和编写时间等，可以单独成行，也可以放在每条指令的后面，解释本指令的目的和功能。注释不但有助于其他编程人员理解当前程序的编写思路和工作原理，而且也能帮助你自己在以后的某个时间重拾这些记忆。

注释必须以分号 “;” 开始。

在源程序编译阶段，编译器将忽略所有注释。因此，在编译之后，这些和生成机器代码无关的内容都统统消失了。

5.3 在屏幕上显示文字

5.3.1 显卡和显存

本程序首先要做的事是在屏幕上显示一行文字。当然，要想在屏幕上显示文字，就需要先了解文字是如何显示在屏幕上的。

为了显示文字，通常需要两种硬件，一是显示器，二是显卡。显卡的职责是为显示器提供内容，并控制显示器的显示模式和状态，显示器的职责是将那些内容以视觉可见的方式呈现在屏幕上。

一般来说，显卡都是独立生产、销售的部件，需要插在主板上才能工作。当然，像处理器、内存这样的东西，也位于主板上。每台计算机都有主板，它就在机箱内部，有时间你可以打开机箱来观察一下。

当然，显卡未必一定是独立的插卡。为了节省使用者的成本，有的显卡会直接做在主板上，这样的显卡也有个名字，叫集成显卡。

显卡控制显示器的最小单位是像素，一个像素对应着屏幕上的一个点。屏幕上通常有数十万乃至更多的像素，通过控制每个像素的明暗和颜色，我们就能让这大量的像素形成文字和美丽的图像。

不过，一个很容易想到的问题是，如何来控制这些像素呢？

答案是显卡都有自己的存储器，因为它位于显卡上，故称显示存储器（Video RAM：VRAM），简称显存，要显示的内容都预先写入显存。和其他半导体存储器一样，显存并没有什么特殊的地方，也是一个按字节访问的存储器件。

对显示器来说，显示黑白图像是最简单的，因为只需要控制每个像素是亮，还是不亮。如果把不亮当成比特“0”，亮看成比特“1”，那就好办了。因为，只要将显存里的每个比特和显示器上的每个像素对应起来，就能实现这个目标。

如图 5-1 所示，显存的第一个字节对应着屏幕左上角连续的 8 个像素；第 2 个字节对应着屏幕上后续的 8 个像素，后面的依次类推。

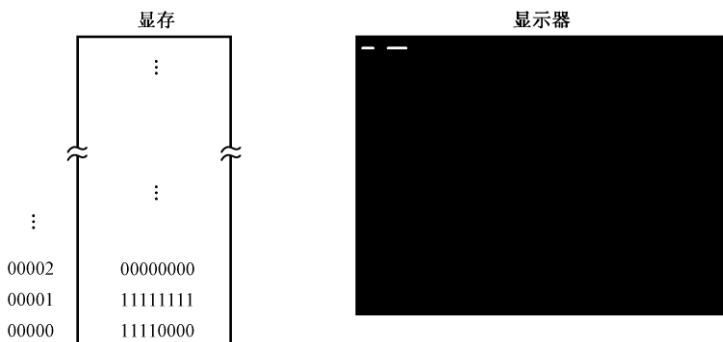


图 5-1 显存内容和显示器内容之间的对应关系

显卡的工作是周期性地从显存中提取这些比特，并把它们按顺序显示在屏幕上。如果是比特

“0”，则像素保持原来的状态不变，因为屏幕本来就是黑的；如果是比特“1”，则点亮对应的像素。

继续观察图 5-1，假设显存中，第 1 个字节的内容是 11110000，第 2 个字节的内容是 11111111，其他所有的字节都是 00000000。在这种情况下，屏幕左上角先是显示 4 个亮点，再显示 4 个黑点，然后再显示 8 个亮点。因为像素是紧挨在一起的，所以我们看到的先是一条白短线，隔着一定距离（4 个像素）又是一条白长线。

黑色和白色只需要 1 个比特就能表示，但要显示更多的颜色，1 个比特就不够了。现在最流行的，是用 24 个比特，即 3 个字节，来对应一个像素。因为 $2^{24}=16777216$ ，所以在这种模式下，同屏可以显示 16777216 种颜色，这称为真彩色。有关颜色的显示和它们与字长的关系，在《穿越计算机的迷雾》一书中有详细的介绍，这里不再赘述。

上面所讨论的，是人们常说的图形模式。图形模式是最容易理解的，同时对显示器来说也是最自然的模式。

现在是图形的时代，就连手机的屏幕都是五彩缤纷的。时光倒退到几十年前，在那个时代，真彩色还没有出现，显示器只能提供有限的色彩，处理器也不够强劲（以今天的眼光来看）。在这种情况下，人们不太可能认为图形显示技术有多么重要，因为他们不看高清电影，也没有数码相机，用计算机制作动画片更是不能想象的事。那个时候，人们的愿望很简单，只要能显示文字就行。

不管是显示图片，还是文字，对显示器来说没有什么不同，因为所有的内容都是由像素组成的，区别仅仅在于这些像素组成的是什么。有时候，人们会说，哦，显示的是一棵树；有时候，人们会说，哦，显示的是一个字母“H”。

问题是，操作显存里的比特，使得屏幕上能显示出字符的形状，是非常麻烦、非常繁重的工作，因为你必须计算该字符所对应的比特位于显存里的什么位置。

为了方便，工程师们想出了一个办法。就像一个二进制数既可以是一个普通的数，也可以代表一条处理器指令一样，他们认为每个字符也可以表示成一个数。比如，数字 0x4C 就代表字符“L”，这个数被称为是字符“L”的 ASCII 代码，后面会讲到。

如图 5-2 所示，可以将字符的代码存放到显存里，第 1 个代码对应着屏幕左上角第 1 个字符，第 2 个代码对应着屏幕左上角第 2 个字符，后面的依次类推。剩下的工作是如何用代码来控制屏幕上的像素，使它们或明或暗以构成字符的轮廓，这是字符发生器和控制电路的事情。

传统上，这种专门用于显示字符的工作方式称为文本模式。文本模式和图形模式是显卡的两种基本工作模式，可以用指令访问显卡，设置它的显示模式。在不同的工作模式下，显卡对显存内容的解释是不同的。

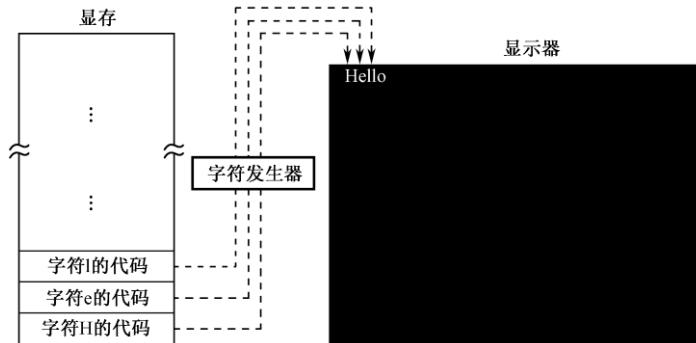


图 5-2 字符在屏幕上的显示原理

为了给出要显示的字符，处理器需要访问显存，把字符的 ASCII 码写进去。但是，显存是位于

显卡上的，访问显存需要和显卡这个外围设备打交道。同时，多一道手续自然是不好的，这当中最重要的考量是速度和效率。想想看，你让人传话给父母，和自己亲自往家里打电话，花费的时间是不一样的。为了实现一些快速的游戏动画效果，或者播放高码率的电影，不直接访问显存是办不到的。

为此，计算机系统的设计者们，这些敢想敢干的人，决定把显存映射到处理器可以直接访问的地址空间里，也就是内存空间里。

如图 5-3 所示，我们知道，8086 可以访问 1MB 内存。其中， $0x0000\sim9FFF$ 属于常规内存，由内存条提供； $0xF0000\sim0xFFFFF$ 由主板上的一个芯片提供，即 ROM-BIOS。

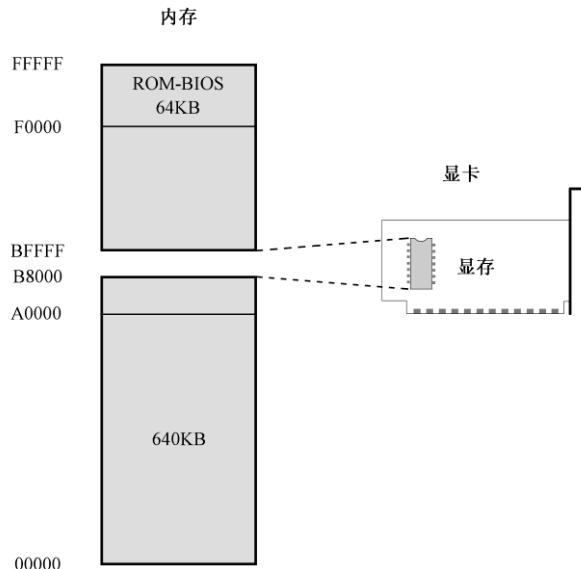


图 5-3 文本模式下显存到内存的映射

这样一来，中间还有一个 320KB 的空洞，即 $0xA0000\sim0xEFFFF$ 。传统上，这段地址空间由特定的外围设备来提供，其中就包括显卡。因为显示功能对于现代计算机来说实在是太重要了。

由于历史的原因，所有在个人计算机上使用的显卡，在加电自检之后都会把自己初始化到 80×25 的文本模式。在这种模式下，屏幕上可以显示 25 行，每行 80 个字符，每屏总共 2000 个字符。

所以，如图 5-3 所示，一直以来， $0xB8000\sim0xBFFFF$ 这段物理地址空间，是留给显卡的，由显卡来提供，用来显示文本。除非显卡出了毛病，否则这段空间总是可以访问的。如果显卡出了毛病怎么办呢？很简单，计算机一定不会通过加电自检过程，这就是传说中的严重错误，计算机是无法启动的，更不要说加载并执行主引导扇区的内容了。

5.3.2 初始化段寄存器

和访问主内存一样，为了访问显存，也需要使用逻辑地址，也就是采用“段地址：偏移地址”的形式，这是处理器的要求。考虑到文本模式下显存的起始物理地址是 $0xB8000$ ，这块内存可以看成是段地址为 $0xB800$ ，偏移地址从 $0x0000$ 延伸到 $0xFFFF$ 的区域，因此我们可以把段地址定为 $0xB800$ 。

访问内存可以使用段寄存器 DS，但这不是强制性的，也可以使用 ES。因为 DS 还有别的用处，所以在这里我们使用 ES 来指向显存所在的段。

源程序第 6、7 行，首先把立即数 0xB800 传送到 AX，然后再把 AX 的值传送到 ES。这样，附加段寄存器 ES 就指向 0xb800 段（段地址为 0xB800）。

你可能会想，为什么不直接这样写：

```
mov es, 0xb800
```

而要用寄存器 AX 来中转呢？

原因是不存在这样的指令，Intel 的处理器不允许将一个立即数传送到段寄存器，它只允许这样的指令：

```
mov 段寄存器, 通用寄存器
```

```
mov 段寄存器, 内存单元
```

没有人能够说清楚这里面的原因，Intel 公司似乎也从没有提到过这件事，尽管从理论上，这是可行的。我们只能想，也许 Intel 是出于好心，避免我们无意中犯错，毕竟，段地址一旦改变，后面对内存的访问都会受到影响。理论上，麻烦一点的方法，可以保证你确实知道自己在做什么。

5.3.3 显存的访问和 ASCII 代码

一旦将显存映射到处理器的地址空间，那么，我们就可以使用普通的传送指令（mov）来读写它，这无疑是非常方便的，但需要首先将它作为一个段来看待，并将它的基址传送到段寄存器。

为此，源程序的第 10、11 行，我们把 0xB800 作为段地址传送到附加段寄存器 ES，以后就用 ES 来读写显存。这样，段内偏移为 0 的位置就对应着屏幕左上角的字符。

在计算机中，每个用来显示在屏幕上的字符，都有一个二进制代码。这些代码和普通的二进制数字没有什么不同，唯一的区别在于，发送这些数字的硬件和接收这些数字的硬件把它们解释为字符，而不是指令或者用于计算的数字。

这就是说，在计算机中，所有的东西都是无差别的数字，它们的意义，只取决于生成者和使用者之间的约定。为了在终端和大型主机，以及主机和打印机、显示器之间交换信息，1967 年，美国国家标准学会制定了美国信息交换标准代码（American Standard Code for Information Interchange，ASCII），如表 5-1 所示。

表 5-1 ASCII 表

b ₆ b ₅ b ₄ b ₃ b ₂ b ₁ b ₀	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	'	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	\$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(8	H	X	h	x
1001	HT	EM)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[k	{

1100	FF	FS	,	<	L	\	l	
1101	CR	GS	-	=	M]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	-	o	DEL

在不同设备之间，或者在同一设备的不同模块之间有一个信息传递标准是非常必要的。想想看，当你用手机向朋友发送短消息时，这些文字当然被编码成二进制数字。如果对方的手机使用了不同的编码，那么他将无法正确还原这些消息，而很可能显示为乱码。

值得注意的是，ASCII是7位代码，只用了一个字节中的低7比特，最高位通常置0。这意味着，ASCII只包含128个字符的编码。所以，在表中，水平方向给出了代码的高3比特，而垂直方向给出了代码的低4比特。比如字符“*”，它的代码是二进制数的010 1010，即0x2A。

ASCII表中有相当一部分代码是不可打印和显示的，它们用于控制通信过程。比如，LF是换行；CR是回车；DEL和BS分别是删除和退格，在我们平时用的键盘上也是有的；BEL是振铃（使远方的终端响铃，以引起注意）；SOH是文头；EOT是文尾；ACK是确认，等等。

注意，一定要遵从约定。比如，你在处理器上编写程序算了一道数学题 $2+3$ ，你也希望把结果5显示在屏幕上。这个时候，算出的结果是0000 0101，即0x05。但是，数字5和字符5是不同的，显卡在任何时候都认为你发送的是ASCII码。所以，你不应该发送0x05，而应该发送0x35。

屏幕上的每个字符对应着显存中的两个连续字节，前一个是字符的ASCII代码，后面是字符的显示属性，包括字符颜色（前景色）和底色（背景色）。如图5-4所示，字符“H”的ASCII代码是0x48，其显示属性是0x07；字符“e”的ASCII代码是0x65，其显示属性是0x07。

如图5-4所示，字符的显示属性（1字节）分为两部分，低4位定义的是前景色，高4位定义的是背景色。色彩主要由R、G、B这3位决定，毕竟我们知道，可以由红(R)、绿(G)、蓝(B)三原色来配出其他所有颜色。K是闪烁位，为0时不闪烁，为1时闪烁；I是亮度位，为0时正常亮度，为1时呈高亮。表5-2给出了背景色和前景色的所有可能值。

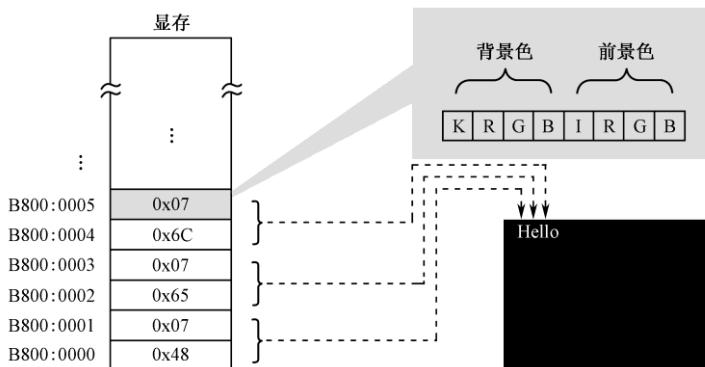


图5-4 字符代码及字符属性示意图

表5-2 80×25文本模式下的颜色表

R	G	B	背景色	前景色	
			K=0时不闪烁, K=1时闪烁	I=0	I=1
0	0	0	黑	黑	灰
0	0	1	蓝	蓝	浅蓝
0	1	0	绿	绿	浅绿
0	1	1	青	青	浅青

1	0	0	红	红	浅红
1	0	1	品(洋)红	品(洋)红	浅品(洋)红
1	1	0	棕	棕	黄
1	1	1	白	白	亮白

从表 5-2 来看，图 5-4 中的字符属性 0x07 可以解释为黑底白字，无闪烁，无加亮。

你可能觉得奇怪，当屏幕上一片漆黑，什么内容都没有的时候，显存里会是什么内容呢？

实际上，这个时候，屏幕上显示的全是黑底白字的空白字符，也叫空格字符（Space），ASCII 代码是 0x20，当你用大拇指按动键盘上最长的那个键时，就产生这个字符。因为它是空白，自然就无法在黑底上看到任何痕迹了。

5.3.4 显示字符

从源程序的第 10 行开始，到第 35 行，目的是显示一串字符“Label offset:”。为此，需要把它们每一个的 ASCII 码顺序写到显存中。

为了方便，多数汇编语言编译器允许在指令中直接使用字符的字面值来代替数值形式的 ASCII 码，比如源程序第 10 行：

```
mov byte [es:0x00], 'L'
```

这等效于

```
mov byte [es:0x00], 0x4C
```

尽管通过查表可以知道字符“L”的 ASCII 代码是 0x4C，但毕竟费事。不过，要在指令中使用字符的字面值，这个字符必须用引号围起来，就像上面一样。在源程序的编译阶段，汇编语言编译器会将它转换成 ASCII 码的形式。

当前的 mov 指令是将立即数传送到内存单元，目的操作数是内存单元，源操作数是立即数（ASCII 代码）。为了访问内存单元，只需要在指令中给出偏移地址，在这里，偏移地址是 0x00。

一般情况下，如果没有附加任何指示，段地址默认在段寄存器 DS 中。比如：

```
mov byte [0x00], 'L'
```

当执行这条指令后，处理器把段寄存器 DS 的内容左移 4 位（相当于乘以十进制数 16 或者十六进制数 0x10），加上这里的偏移地址 0x00，就得到了物理地址。

但是实际上，显存的段地址位于段寄存器 ES 中，我们希望使用 ES 来访问内存。因此，这里使用了段超越前缀“es:”。这就是说，我们明确要求处理器在生成物理地址时，使用段寄存器 ES，而不是默认情况下的 DS。

因为指令中给出的偏移地址是 0x00，且 ES 的值已经在前面被设为 0xB800，故它指向 ES 段中，偏移地址为 0 的内存单元，即 0xB800:0x0000，也就是物理地址 0xB8000，这个内存单元对应着屏幕左上角第一个字符的位置。

还需要注意的是，因为目的操作数给出的是一个内存地址，我们要用源操作数来修改这个地址里的内容，所以，目的操作数必须用方括号围起来，以表明它是一个地址，处理器应该用这个地址再次访问内存，将源操作数写进这个单元。实际上，这类似于高级语言里的指针。

最后，关键字“byte”用来修饰目的操作数，指出本次传送是以字节的方式进行的。在 16 位的处理器上，单次操作的数据宽度可以是 8 位，也可以是 16 位。到底是 8 位，还是 16 位，可以根据目的操作数或者源操作数来判断。遗憾的是，在这里，目的操作数是偏移地址 0x00，它可以是字节单元，也可以是字单元，到底是哪一种，无法判断；而源操作数呢，是立即数 0x4C，它既可以解

释为8位的0x4C，也可以解释为16位的0x004C。在这种情况下，编译器将无法搞懂你的真实意图，只能报告错误，所以必须用“byte”或者“word”进行修饰（明确指示）。于是，一旦目的操作数被指明是“byte”的，那么，源操作数的宽度也就明确了。相反地，下面的指令就不需要任何修饰：

```
mov [0x00], AL      ;按字节操作
mov AX, [0x02]       ;按字操作
```

因为屏幕上的一个字符对应着内存中的两个字节：ASCII代码和属性，所以，源程序第11行的功能是将属性值0x07传送到下一个内存单元，即偏移地址0x01处。这个属性可以解释为黑底白字，无闪烁，也无加亮，请参阅表5-2。

后面，从第12行开始，到第35行，用于向显示缓冲区填充剩余部分的字符。注意，在这个过程中，偏移地址一直是递增的。

5.4 显示标号的汇编地址

5.4.1 标号

处理器访问内存时，采用的是“段地址：偏移地址”的模式。对于任何一个内存段来说，段地址可以开始于任何16字节对齐的地方，偏移地址则总是从0x0000开始递增。

为了支持这种内存访问模式，在源程序的编译阶段，编译器会把源程序5-1整体上作为一个独立的段来处理，并从0开始计算和跟踪每一条指令的地址。因为该地址是在编译期间计算的，故称为汇编地址。汇编地址是在源程序编译期间，编译器为每条指令确定的汇编位置（Assembly Position），也就是每条指令相对于整个程序开头的偏移量，以字节计。当编译后的程序装入物理内存后，它又是该指令在内存段内的偏移地址。

如表5-3所示，在用我们的配书工具Nasmide书写并编译代码清单5-1后，除了生成一个以“.bin”为扩展名的二进制文件，还会生成一个以“.lst”为扩展名的列表文件。这张表列出的，就是本章代码清单5-1编译后生成的列表文件内容。

表5-3共分五栏，从左到右依次是行号、指令的汇编地址、指令编译后的机器代码、源程序代码和注释。可以看出，第一条指令mov ax,0xb800的汇编地址是0x00000000,对应的机器代码为B8 00 B8；第二条指令mov es,ax的汇编地址是0x00000003，机器代码为8E C0。

表5-3 代码清单5-1编译后的列表文件内容

	;代码清单5-1		
2	;文件名: c05_mbr.asm		
3	;文件说明: 硬盘主引导扇区代码		
4	;创建日期: 2011-3-31 21:15		
5			



```

6    00000000    B800B8        mov ax, 0xb800          ;指向文本模式的显示缓冲区
7    00000003    8EC0         mov es, ax
8
9
10   00000005    26C60600004C  mov byte [es:0x00], 'L'
11   0000000B    26C606010007  mov byte [es:0x01], 0x07
12   00000011    26C606020061  mov byte [es:0x02], 'a'
13   00000017    26C606030007  mov byte [es:0x03], 0x07
14   0000001D    26C606040062  mov byte [es:0x04], 'b'
15   00000023    26C606050007  mov byte [es:0x05], 0x07
16   00000029    26C606060065  mov byte [es:0x06], 'e'
17   0000002F    26C606070007  mov byte [es:0x07], 0x07
18   00000035    26C60608006C  mov byte [es:0x08], 'l'
19   0000003B    26C606090007  mov byte [es:0x09], 0x07
20   00000041    26C6060A0020  mov byte [es:0xa], ', '
21   00000047    26C6060B0007  mov byte [es:0xb], 0x07
22   0000004D    26C6060C006F  mov byte [es:0xc], "o"
23   00000053    26C6060D0007  mov byte [es:0xd], 0x07
24   00000059    26C6060E0066  mov byte [es:0xe], 'f'
25   0000005F    26C6060F0007  mov byte [es:0xf], 0x07
26   00000065    26C606100066  mov byte [es:0x10], 'f'
27   0000006B    26C606110007  mov byte [es:0x11], 0x07
28   00000071    26C606120073  mov byte [es:0x12], 's'
29   00000077    26C606130007  mov byte [es:0x13], 0x07
30   0000007D    26C606140065  mov byte [es:0x14], 'e'
31   00000083    26C606150007  mov byte [es:0x15], 0x07
32   00000089    26C606160074  mov byte [es:0x16], 't'
33   0000008F    26C606170007  mov byte [es:0x17], 0x07
34   00000095    26C60618003A  mov byte [es:0x18], ':'
35   0000009B    26C606190007  mov byte [es:0x19], 0x07
36
37   000000A1    B8[2E01]      mov ax, number          ;取得标号 number 的偏移地址
38   000000A4    BB0A00       mov bx, 10
39
40
41   000000A7    8CC9         mov cx, cs
42   000000A9    8ED9         mov ds, cx
43
44
45   000000AB    BA0000       mov dx, 0
46   000000AE    F7F3         div bx
47   000000B0    8816[2E7D]   mov [0x7c00+number+0x00], dl  ;保存个位上的数字
48

```

```

49                                ;求十位上的数字
50 000000B4    31D2          xor dx, dx
51 000000B6    F7F3          div bx
52 000000B8    8816[2F7D]    mov [0x7c00+number+0x01], dl  ;保存十位上的数字
53
54                                ;求百位上的数字
55 000000BC    31D2          xor dx, dx
56 000000BE    F7F3          div bx
57 000000C0    8816[307D]    mov [0x7c00+number+0x02], dl  ;保存百位上的数字
58
59                                ;求千位上的数字
60 000000C4    31D2          xor dx, dx
61 000000C6    F7F3          div bx
62 000000C8    8816[317D]    mov [0x7c00+number+0x03], dl  ;保存千位上的数字
63
64                                ;求万位上的数字
65 000000CC    31D2          xor dx, dx
66 000000CE    F7F3          div bx
67 000000D0    8816[327D]    mov [0x7c00+number+0x04], dl  ;保存万位上的数字
68
69                                ;以下用十进制显示标号的偏移地址
70 000000D4    A0[327D]      mov al, [0x7c00+number+0x04]
71 000000D7    0430          add al, 0x30
72 000000D9    26A21A00     mov [es:0x1a], al
73 000000DD    26C6061B0004  mov byte [es:0x1b], 0x04
74
75 000000E3    A0[317D]      mov al, [0x7c00+number+0x03]
76 000000E6    0430          add al, 0x30
77 000000E8    26A21C00     mov [es:0x1c], al
78 000000EC    26C6061D0004  mov byte [es:0x1d], 0x04
79
80 000000F2    A0[307D]      mov al, [0x7c00+number+0x02]
81 000000F5    0430          add al, 0x30
82 000000F7    26A21E00     mov [es:0x1e], al
83 000000FB    26C6061F0004  mov byte [es:0x1f], 0x04
84
85 00000101    A0[2F7D]      mov al, [0x7c00+number+0x01]
86 00000104    0430          add al, 0x30
87 00000106    26A22000     mov [es:0x20], al
88 0000010A    26C606210004  mov byte [es:0x21], 0x04
89
90 00000110    A0[2E7D]      mov al, [0x7c00+number+0x00]
91 00000113    0430          add al, 0x30

```

```

92 00000115    26A22200      mov [es:0x22], al
93 00000119    26C606230004   mov byte [es:0x23], 0x04
94
95 0000011F    26C606240044   mov byte [es:0x24], 'D'
96 00000125    26C606250007   mov byte [es:0x25], 0x07
97
98 0000012B    E9FDFF        infi: jmp near infi       ;无限循环
99
100 0000012E    0000000000    number db 0,0,0,0,0
101
102 00000133    00<rept>    times 203 db 0
103 000001FE    55AA          db 0x55, 0xaa

```

从表 5-3 中可以看出，在编译阶段，每条指令都被计算并赋予了一个汇编地址，就像它们已经被加载到内存中的某个段里一样。实际上，如图 5-5 所示，当编译好的程序加载到物理内存后，它在段内的偏移地址和它在编译阶段的汇编地址是相等的。

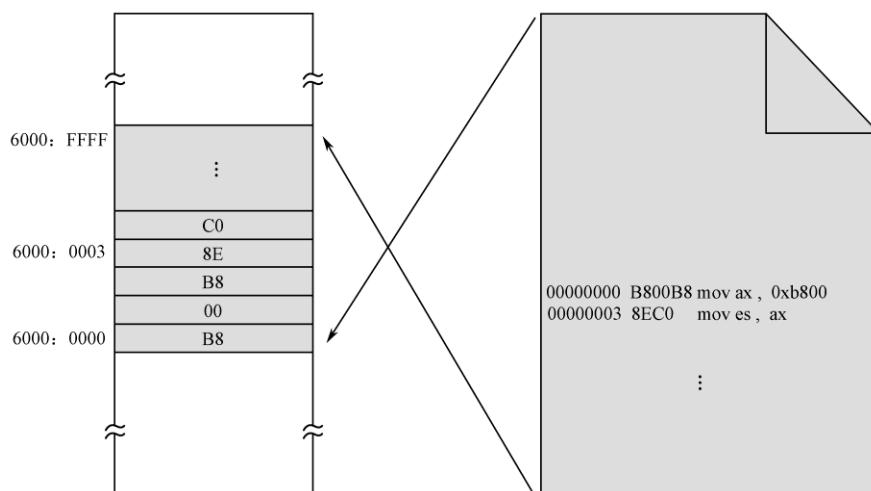


图 5-5 汇编地址和偏移地址的关系

正如图 5-5 所示，编译后的程序是整体加载到内存中某个段的，交叉箭头用于指示它们之间的映射关系。之所以箭头是交叉的，是因为源程序的编译是从上往下的，而内存地址的增长是从下往上的（从低地址往高地址方向增长）。

图 5-5 中假定程序是从内存物理地址 0x6000 开始加载的。因为该物理地址也对应着逻辑地址 0x6000:0x0000，因此我们可以说，该程序位于段 0x6000 内。

在编译阶段，源程序的第一条指令 `mov ax,0xb800` 的汇编地址是 0x00000000，而它在整个程序装入内存后，在段内的偏移地址是 0x0000，即逻辑地址 0x6000:0000，两者的偏移地址是一致的。

再看源程序的第二条指令，是 `mov es,ax`，它在编译阶段的汇编地址是 0x00000003。在整个程序装入内存后，它在段内的偏移地址是 0x0003，也没有变化。

这就很好地说明了汇编地址和偏移地址之间的对应关系。理解这一点，对后面的编程很重要。

在 NASM 汇编语言里，每条指令的前面都可以拥有一个标号，以代表和指示该指令的汇编地址。毕竟，由我们自己来计算和跟踪每条指令所在的汇编地址是极其困难的。这里有一个很好的例子，比如源程序第 98 行：

```
infi: jmp near infi
```

在这里，行首带冒号的是标号是“infi”。请看表 5-3，这条指令的汇编地址是 0x00000012D，故 infi 就代表数值 0x00000012D，或者说是 0x00000012D 的符号化表示。

标号之后的冒号是可选的。所以下面的写法也是正确的：

```
infi jmp near infi
```

标号并不是必需的，只有在我们需要引用某条指令的汇编地址时，才使用标号。正是因为这样，本章源程序中的绝大多数指令都没有标号。

标号可以单独占用一行的位置，像这样：

```
infi:  
    jmp near infi
```

这种写法和第 98 行相比，效果并没有什么不同，因为 infi 所在的那一行没有指令，它的地址就是下一行的地址，换句话说，和下一行的地址是相同的。

标号可以由字母、数字、“_”、“\$”、“#”、“@”、“~”、“.”、“?”组成，但必须以字母、“.”、“_”和“?”中的任意一个打头。

5.4.2 如何显示十进制数字

我们已经知道，标号可以用来代表指令的汇编地址。现在，我们要编写指令，在屏幕上把这个地址的数值显示出来。为此，源程序的第 37 行用于获取标号所代表的汇编地址：

```
mov ax, number
```

标号“number”位于源程序的第 100 行，只不过后面没有跟着冒号“：“。你当然可以加上冒号，但这无关紧要。注意，传送到寄存器 AX 的值是在源程序编译时确定的，在编译阶段，编译器会将标号 number 转换成立即数。如表 5-3 所示，标号 number 处的汇编地址是 0x012E，因此，这条语句其实就是这样

```
mov ax, 0x012E
```

问题在于，如果不是借助于别的工具和手段，你不可能知道此处的汇编地址是 0x012E。所以，在汇编语言中使用标号的好处是不必关心这些。

因此，当这条指令编译后，得到的机器指令为 B8[2E01]，或者 B8 2E 01。B8 是操作码，后面是字操作数 0x012E，只不过采用的是低端字节序。

十六进制数 0x012E 等于十进制数 302，但是，通过前面对字符显示原理的介绍，我们应该清楚，直接把寄存器 AX 中的内容传送到显示缓冲区，是不可能在屏幕上出现“302”的。

解决这个问题的办法是将它的每个数位单独拆分出来，这需要不停地除以 10。

考虑到寄存器 AX 是 16 位的，可以表示的数从二进制的 0000000000000000 到 1111111111111111，也就是十进制的 0~65535，故它可以容纳最大 5 个数位的十进制数，从个位到万位，比如 61238。那么，假如你并不知道它是多少，只知道它是一个 5 位数，那么，如何通过分解得到它的每个数位呢？

首先，用 61238 除以 10，商为 6123，余 8，本次相除的余数 8 就是个位数字；

然后，把上一次的商数 6123 作为被除数，再次除以 10，商为 612，余 3，余数 3 就是十位上的

数字：

接着，再用上一次的商数 612 除以 10，商为 61，余 2，余数 2 就是百位上的数字；

同上，再用 61 除以 10，商为 6，余 1，余数 1 就是千位上的数字；

最后，用 6 除以 10，商为 0，余 6，余数 6 就是万位上的数字。

很显然，只要把 AX 的内容不停地除以 10，只需要 5 次，把每次的余数反向组合到一起，就是原来的数字。同样，如果反向把每次的余数显示到屏幕上，应该就能看见这个十进制数是多少了。

不过，即使是得到了单个的数位，也还是不能在屏幕上显示，因为它们是数字，而非 ASCII 代码。比如，数字 0x05 和字符“5”是不同的，后者实际上是数字 0x35。

观察表 5-1，你会发现，字符“0”的 ASCII 代码是 0x30，字符“1”的 ASCII 代码是 0x31，字符“9”的 ASCII 代码是 0x39。这就是说，把每次相除得到的余数加上 0x30，在屏幕上显示就没问题了。

5.4.3 在程序中声明并初始化数据

可以用处理器提供的除法指令来分解一个数的各个数位，但是每次除法操作后得到的数位需要临时保存起来以备后用。使用寄存器不太现实，因为它的数量很少，且还要在后续的指令中使用。因此，最好的办法是在内存中专门留出一些空间来保存这些数位。

尽管我们的目的仅仅是分配一些空间，但是，要达到这个目的必须初始化一些初始数据来“占位”。这就好比是排队买火车票，你可以派任何无关的人去帮你占个位置，真正轮到你买的时候，你再出现。源程序的第 100 行用于声明并初始化这些数据，而标号 number 则代表了这些数据的起始汇编地址。

要放在程序中的数据是用 DB 指令来声明（Declare）的，DB 的意思是声明字节（Declare Byte），所以，跟在它后面的操作数都占一个字节的长度（位置）。注意，如果要声明超过一个以上的数据，各个操作数之间必须以逗号隔开。

除此之外，DW（Declare Word）用于声明字数据，DD（Declare Double Word）用于声明双字（两个字）数据，DQ（Declare Quad Word）用于声明四字数据。DB、DW、DD 和 DQ 并不是处理器指令，它只是编译器提供的汇编指令，所以称做伪指令（pseudo Instruction）。伪指令是汇编指令的一种，它没有对应的机器指令，所以它不是机器指令的助记符，仅仅在编译阶段由编译器执行，编译成功后，伪指令就消失了，所以在程序执行时，伪指令是得不到处理器光顾的，实际上，程序执行时，伪指令已不存在。

声明的数据可以是任何值，只要不超过伪指令所指示的大小。比如，用 DB 声明的数据，不能超过一个字节所能表示的数的大小，即 0xFF。我们在此声明了 5 个字节，并将它们的值都初始化为 0。

和指令不同，对于在程序中声明的数值，在编译阶段，编译器会在它们被声明的汇编地址处原样保留。

按照标准的做法，程序中用到的数据应当声明在一个独立的段，即数据段中。但是在这里，为方便起见，数据和指令代码是放在同一个段中的。不过，方便是方便了，但也带来了一个隐患，如果安排不当，处理器就有可能执行到那些非指令的数据上。尽管有些数碰巧和某些指令的机器码相同，也可以顺利执行，但毕竟不是我们想要的结果，违背了我们的初衷。

好在我们很小心，在本程序中把数据声明在所有指令之后，在这个地方，处理器的执行流程无法到达。

5.4.4 分解数的各个数位

源程序第41、42行，是把代码段寄存器CS的内容传送到通用寄存器CX，然后再从CX传送到数据段寄存器DS。在此之后，数据段和代码段都指向同一个段。之所以这么做，是因为我们刚才声明的数据是和指令代码混在一起的，可以认为是位于代码段中。尽管在指令中访问这些数据可以使用段超越前缀“CS:”，但习惯上，通过数据段来访问它们更自然一些。

前面已经说过，要分解一个数的各个数位，需要做除法。8086处理器提供了除法指令div，它可以做两种类型的除法。

第一种类型是用16位的二进制数除以8位的二进制数。在这种情况下，被除数必须在寄存器AX中，必须事先传送到AX寄存器里。除数可以由8位的通用寄存器或者内存单元提供。指令执行后，商在寄存器AL中，余数在寄存器AH中。比如：

```
div cl
div byte [0x0023]
```

前一条指令中，寄存器CL用来提供8位的除数。假如AX中的内容是0x0005，CL中的内容是0x02，指令执行后，CL中的内容不变，AL中的商是0x02，AH中的余数是0x01。

后一条指令中，除数位于数据段内偏移地址为0x0023的内存单元里。这条指令执行时，处理器将数据段寄存器DS的内容左移4位，加上偏移地址0x0023以形成物理地址。然后，处理器再次访问内存，从此处取得一个字节，作为除数同寄存器AX做一次除法。

任何时候，只要是在指令中涉及内存地址的，都允许使用段超越前缀。比如：

```
div byte [cs:0x0023]
div byte [es:0x0023]
```

话又说回来了，在一个源程序中，通常不可能知道汇编地址的具体数值，只能使用标号。所以，指令中的地址部分更常见的形式是使用标号。比如：

```
dividend dw 0x3f0
divisor db 0x3f

.....
mov ax, [dividend]
div byte [divisor]
```

上面的程序很有意思，首先，声明了标号dividend并初始化了一个字0x3f0作为被除数；然后，又声明了标号divisor并初始化一个字节0x3f作为除数。

在后面的mov和div指令中，是用标号dividend和divisor来代替被除数和除数的汇编地址。在编译阶段，编译器用具体的数值取代括号中的标号dividend和divisor。现在，假设dividend和divisor所代表的汇编地址分别是0xf000和0xf002，那么，在编译阶段，编译器在生成这两条指令的机器码之前，会先将它们转换成以下的形式：

```
mov ax, [0xf000]
div byte [0xf002]
```

当第一条指令执行时，处理器用0xf000作为偏移地址，去访问数据段（段地址在段寄存器DS中），来取得内存中的一个字0x3F0，并把它传送到寄存器AX中。

十进制数 2218367590 等于以下二进制数



图 5-6 用 DX:AX 分解 32 位二进制数示意图

10000100001110011001101001100110。在做除法之前，先要分成两段进行“切割”，以分别装入寄存器 DX 和 AX。为了方便，我们通常用“DX:AX”来描述 32 位的被除数。

同时，除数可以由 16 位的通用寄存器或者内存单元提供，指令执行后，商在 AX 中，余数在 DX 中。比如下面的指令：

```
div cx
div word [0x0230]
```

源程序第 45 行把 0 传送到 DX 寄存器，这意味着，我们是想把 DX:AX 作为被除数，即被除数的高 16 位是全零。至于被除数的低 16 位，已经在第 37 行的代码中被置为标号 number 的汇编地址。

回到前面的第 38 行，该指令把 10 作为除数传送到通用寄存器 BX 中。

一切都准备好了，源程序第 46 行，div 指令用 DX:AX 作为被除数，除以 BX 的内容，执行后得到的商在 AX 中，余数在 DX 中。因为除数是 10，余数自然比 10 小，我们可以从 DL 中取得。

第 1 次相除得到的余数是个位上的数字，我们要将它保存到声明好的数据区中。所以，源程序第 47 行，我们又一次用到了传送指令，把寄存器 DL 中的余数传送到数据段。

可以看到，指令中没有使用段超越前缀，所以处理器在执行时，默认地使用段寄存器 DS 来访问内存。偏移地址是由标号 number 提供的，它是数据区的首地址，也可以说是数据区中第一个数据的地址。因此，number 和 number+0x00 是一样的，没有区别。

因为我们访问的是 number 所指向的内存单元，故要用中括号围起来，表明这是一个地址。

令人不解的是，第 47 行中，偏移地址并非理论上的 number+0x00，而是 0x7c00+number+0x00。这个 0x7c00 是从哪里来的呢？

标号 number 所代表的汇编地址，其数值是在源程序编译阶段确定的，而且是相对于整个程序的开头，从 0 开始计算的。请看一下表 5-3 的第 37 行，这个在编译阶段计算出来的值是 0x012E。在运行的时候，如果该程序被加载到某个段内偏移地址为 0 的地方，这不会有什么问题，因为它们是一致的。

但是，事实上，如图 5-7 所示，这里显示的是整个 0x0000 段，其中深色部分为主引导扇区所处的位置。主引导扇区代码是被加载到 0x0000:0x7C00 处的，而非 0x0000:0000。对于程序的执行来说，这不会有什么问题，因为主引导扇区的内容被加载到内存中并开始执行时，

当第二条指令执行时，处理器采用同样的方法取得内存中的一个字节 0x3F，用它来和寄存器 AX 中的内容做除法。当然，除法指令 div 的功能你是知道的。

说了这么多，其实是在强调标号和汇编地址的对应关系，以及如何在指令中使用符号化的偏移地址。

第二种类型是用 32 位的二进制数除以 16 位的二进制数。在这种情况下，因为 16 位的处理器无法直接提供 32 位的被除数，故要求被除数的高 16 位在 DX 中，低 16 位在 AX 中。

这里有一个例子，如图 5-6 所示，假如被除数是十进制数 2218367590，那么，它对应着一个 32 位的二进制数

CS=0x0000, IP=0x7C00。

加载位置的改变不会对处理器执行指令造成任何困扰，但会给数据访问带来麻烦。要知道，当前数据段寄存器 DS 的内容是 0x0000，因此，number 的偏移地址实际上是 $0x012E+0x7C00=0x7D2E$ 。当正在执行的指令仍然用 0x012E 来访问数据，灾难就发生了。

所以，在编写主引导扇区程序时，我们就要考虑到这一点，必须把代码写成

```
mov [0x7c00+number+0x00], dl
```

指令中的目的操作数是在编译阶段确定的，因此，在编译阶段，编译器同样会首先将它转换成以下的形式，再进一步生成机器码：

```
mov [0x7d2e], dl
```

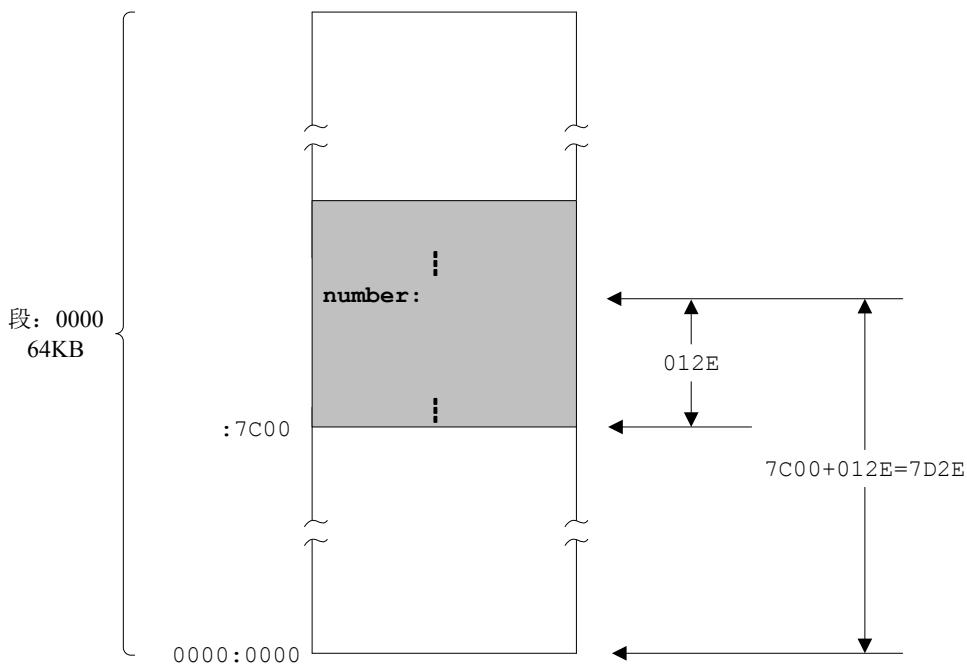


图 5-7 主引导程序加载到内存后的地址变化

这样，如表 5-3 的第 47 行所示，在编译后，编译器就会将这条指令编译成 88 16 2E 7D，其中前两个字节是操作码，后两个字节是低端字节序的 0x7D2E。当这条指令执行时，处理器将段寄存器 DS 的内容（和 CS 一样，是 0x0000）左移 4 位，再加上指令中提供的偏移地址 0x7D2E，就得到了实际的物理地址（0x07D2E）。

关于这条指令的另外一个问题，虽然目的操作数也是一个内存单元地址，但并没有用关键字“byte”来修饰。这是因为源操作数是寄存器 DL，编译器可以据此推断这是一个字节操作，不存在歧义。

现在已经得到并保存了个位上的数字，下一步是计算十位上的数字，方法是用上一次得到的商作为被除数，继续除以 10。恰好，AX 已经是被除数的低 16 位，现在只需要把 DX 的内容清零即可。

为此，代码清单 5-1 第 50 行，用了一个新的指令 xor 来将 DX 寄存器的内容清零。

xor，在数字逻辑里是异或（eXclusive OR）的意思，或者叫互斥或、互斥的或运算。《在穿越计算机的迷雾》里，已经花了大量的篇幅讲解数字逻辑。在数字逻辑里，如果 0 代表假，1 代表真，那么

```
0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0
```

xor 指令的目的操作数可以是通用寄存器和内存单元，源操作数可以是通用寄存器、内存单元和立即数（不允许两个操作数同时为内存单元）。而且，异或操作是在两个操作数相对应的比特之间单独进行的。

一般地，`xor` 指令的两个操作数应当具有相同的数据宽度。因此，其指令格式可以总结为以下几种情况：

`xor 8位通用寄存器, 8位立即数`，例如：`xor al, 0x55`

`xor 8位通用寄存器, 指向8位实际操作数的内存地址`，例如：`xor cl, [0x2000]`

`xor 8位通用寄存器, 8位通用寄存器`，例如：`xor bl, dl`

`xor 16位通用寄存器, 16位立即数`，例如：`xor ax, 0xf033`

`xor 16位通用寄存器, 指向16位实际操作数的内存地址`，例如：`xor bx, [0x2002]`

`xor 16位通用寄存器, 16位通用寄存器`，例如：`xor dx, bx`

`xor 指向8位实际操作数的内存地址, 8位立即数`，例如：`xor byte[0x3000], 0xf0`

`xor 指向8位实际操作数的内存地址, 8位通用寄存器`，例如：`xor [0x06], al`

`xor 指向16位实际操作数的内存地址, 16位立即数`，例如：`xor word [0x2002], 0x55aa`

`xor 指向16位实际操作数的内存地址, 16位通用寄存器`，例如：`xor [0x20], dx`

因为异或操作是在两个操作数相对应的比特之间单独进行，故，以下指令执行后，AX 寄存器中的内容为 0xF0F3。

```
mov ax, 0000_0000_0000_0010B
```

```
xor ax, 1111_0000_1111_0001B ; AX←1111_0000_1111_0011B, 即, 0xf0f3
```

注意，这两条指令的源操作数都采用了二进制数的写法，NASM 编译器允许使用下画线来分开它们，好处是可以更清楚地观察到那些感兴趣的比特。

回到当前程序中，因为指令 `xor dx,dx` 中的目的操作数和源操作数相同，那么，不管 DX 中的内容是什么，两个相同的数字异或，其结果必定为 0，故这相当于将 DX 清零。

值得一提的是，尽管都可以用于将寄存器清零，但是编译后，`mov dx,0` 的机器码是 BA 00 00；而 `xor dx,dx` 的机器码则是 31 D2，不但较短，而且，因为 `xor dx,dx` 的两个操作数都是通用寄存器，所以执行速度最快。

第二次相除的结果可以求得十位上的数字，源程序第 52 行用来将十位上的数字保存到从 `number` 开始的第 2 个存储单元里，即 `number+0x01`。

从源程序第 55 行开始，一直到第 67 行，做的都是和前面相同的事情，即，分解各位上的数字，并予以保存，这里不再赘述。

5.4.5 显示分解出来的各个数位

经过 5 次除法操作，可以将寄存器 AX 中的数分解成单独的数位，下面的任务是将这些数位显示出来，方法是从 DS 指向的数据段依次取出这些数位，并写入 ES 指向的附加段（显示缓冲区）。

因为在分解并保存各个数位的时候，顺序是“个、十、百、千、万”位，当在屏幕上显示时，却要反过来，先显示万位，再显示千位，等等，因为屏幕显示是从左往右进行的。所以，源程序第 70 行，先从数据段中，偏移地址为 `number+0x04` 处取得万位上的数字，传送到 AL 寄存器。当然，因为程序是加载到 `0x0000:0x7C00` 处的，所以正确的偏移地址是 `0x7C00+number+0x04`。

然后，源程序第 71 行，将 AL 中的内容加上 `0x30`，以得到与该数字对应的 ASCII 代码。在这里，`add` 是加法指令，用于将一个数与另一个数相加。

add 指令需要两个操作数，目的操作数可以是 8 位或者 16 位的通用寄存器，或者指向 8 位或者 16 位实际操作数的内存地址；源操作数可以是相同数据宽度的 8 位或者 16 位通用寄存器、指向 8 位或者 16 位实际操作数的内存地址，或者立即数，但不允许两个操作数同时为内存单元。相加后，结果保存在目的操作数中。比如：

```
add al,cl
add ax,0x123f
add [label_a],cx
add ax,[label_a]
add byte [label_a],0x08
```

源程序第 72 行，将要显示的 ASCII 代码传送到显示缓冲区偏移地址为 0x1A 的位置，该位置紧接着前面的字符串“Label offset:”。显示缓冲区是由段寄存器 ES 指向的，因此使用了段超越前缀。

源程序第 73 行，将该字符的显示属性写入下一个内存位置 0x1B。属性值 0x04 的意思是黑底红字，无闪烁，无加亮。

从源程序的第 75 行开始，到第 93 行，用于显示其他 4 个数位。

源程序第 95、96 行，用于以黑底白字显示字符“D”，意思是所显示的数字是十进制的。

5.5 使程序进入无限循环状态

数字显示完成后，原则上整个程序就结束了，但对处理器来说，它并不知道。对它来说，取指令、执行是永无止境的。程序有大小，执行无停息，它这么做的结果，就是会执行到后面非指令的数据上，然后……

问题在于我们现在的确无事可做。为避免发生问题，源程序第 98 行，安排了一个无限循环：

```
infi: jmp near infi
```

jmp 是转移指令，用于使处理器脱离当前的执行序列，转移到指定的地方执行，关键字 near 表示目标位置依然在当前代码段内。上面这条指令唯一特殊的地方在于它不是转移到别处，而是转移到自己。也就是说，它将会不停地重复执行自己。不要觉得奇怪，这是允许的。

处理器取指令、执行指令是依赖于段寄存器 CS 和指令指针寄存器 IP 的，8086 处理器取指令时，把 CS 的内容左移 4 位，加上 IP 的内容，形成 20 位的物理地址，取得指令，然后执行，同时把 IP 的内容加上当前指令的长度，以指向下一条指令的偏移地址。

但是，一旦处理器取到的是转移指令，情况就完全变了。

很容易想到，指令 jmp near infi 的意图是转移到标号 infi 所在的位置执行。可是，正如我们前面所说的，程序在内存中的加载位置是 0x0000:0x7C00，所以，这条指令应当写成

```
jmp near 0x7c00+infi
```

实际上，不加还好，加上了 0x7C00，就完全错了。

jmp 指令有多种格式。最典型地，它的操作数可以是直接给出的段地址和偏移地址，这称为绝对地址。比如：

```
jmp 0x5000:0xf0c0
```

此时，要转移到的目标位置是非常明确的，即，段地址为 0x5000，段内偏移地址为 0xf0c0。在这种情况下，指令的操作码为 0xEA，故完整的机器指令是：

```
EA C0 F0 00 50
```

处理器执行时，发现操作码为 0xEA，于是，将指令中给出的段地址传送到段寄存器 CS；将偏移地址传送到指令指针寄存器 IP，从而转移到目标位置处接着执行。

但是，在此处，jmp 指令使用了关键字“near”，且操作数是以标号（infi）的形式给出。这很容易让我们想到，这又是另一种形式的转移指令，转移的目标位置处在当前代码段内，指令中的操作数应当是目标位置的偏移地址。实际上，这是不正确的。

实际上，这是一个 3 字节指令，操作码是 0xE9，后跟一个 16 位（两字节）的操作数。但是，该操作数并非目标位置的偏移地址，而是目标位置相对于当前指令处的偏移量（以字节为单位）。在编译阶段，编译器是这么做的：用标号（目标位置）处的汇编地址减去当前指令的汇编地址，再减去当前指令的长度（3），就得到了 jmp near infi 指令的实际操作数。也不是编译器愿意费这个事，这是处理器的要求。

这样看来，jmp near infi 的机器指令格式和它的汇编指令格式完全不同，颇具迷惑性，所以一定要认清它的本质。这种转移是相对的，操作数是一个相对量，如果你人为地加上 0x7C00，那反而不对了。

在指令执行阶段，处理器用指令指针寄存器 IP 的内容加上该指令的操作数，再加上该指令的长度（3），就得到了要转移的实际偏移地址，同时 CS 寄存器的内容不变。因为改变了 IP 的内容，这直接导致处理器的指令执行流程转向目标位置。

jmp 指令具有多种格式，我们现在所用的，只是其中的一种，叫做相对近转移。有关其他格式，以及这些格式之间的差异，我们将在后面的章节里结合具体的实例进行讲解。

5.6 完成并编译主引导扇区代码

5.6.1 主引导扇区有效标志

主引导扇区在系统启动过程中扮演着承上启下的角色，但并非是唯一的选择。如果硬盘的主引导扇区不可用，系统还有其他选择，比如可以从光盘和 U 盘启动。

然而，如果不试试水的深浅就一个猛子扎下池塘，这并非一个明智之举。同样地，如果主引导扇区是无效的，上面并非是一些处理器可以识别的指令，而处理器又不加鉴别地执行了它，其结果是陷入宕机状态，更不要提从其他设备启动了。

为此，计算机的设计者们决定，一个有效的主引导扇区，其最后两个字节的数据必须是 0x55 和 0xAA。否则，这个扇区里保存的就不是一些有意而为的数据。

定义这两个字节很简单，伪指令 db 和 dw 就可以实现。源程序第 103 行就是 db 版本的实现，但没有标号。标号的作用是提供当前位置的汇编（偏移）地址供其他指令引用，如果没有任何指令引用这个地址，标号可以省略。这是两个单独的字节，所以 0x55 在前，0xAA 在后，即使编译之后也是这个顺序。

但是，如果采用 dw 版本，应该这样写：

```
dw 0xaa55
```

因为，在 Intel 处理器上，将一个字写入内存时，是采用低端字节序的，低字节 0x55 置入低地

址端（在前），高字节 0xAA 在高地址端（在后）。

麻烦在于，如何使这两个字节正好位于 512 字节的最后。前面的代码有多少个字节我们不知道，那是由 NASM 编译器计算和跟踪的。

我们当然有非常好的办法，但还不宜在这里说明。但是，经过计算和尝试，我知道，在前面的内容和结尾的 0xAA55 之间，有 203 字节的空洞。因此，源程序的第 102 行，用于声明 203 为 0 的数值来填补。

为了方便，伪指令 times 可用于重复它后面的指令若干次。比如

```
times 20 mov ax,bx
```

将在编译时重复生成 mov ax,bx 指令 20 次，即重复该指令的机器码（89 D8）20 次。

因此

```
times 203 db 0
```

将会在编译时保留 203 个为 0 的字节。

5.6.2 代码的保存和编译

本章的代码是现成的，配书源代码解压缩之后，可以在文件夹“c05”里找到，文件名为 c05_mbr.asm。打开该文件，将其编译成 c05_mbr.bin。

该文件的大小为 512 字节，可以用配书工具 HexView 来查看其内容，如图 5-8 所示。

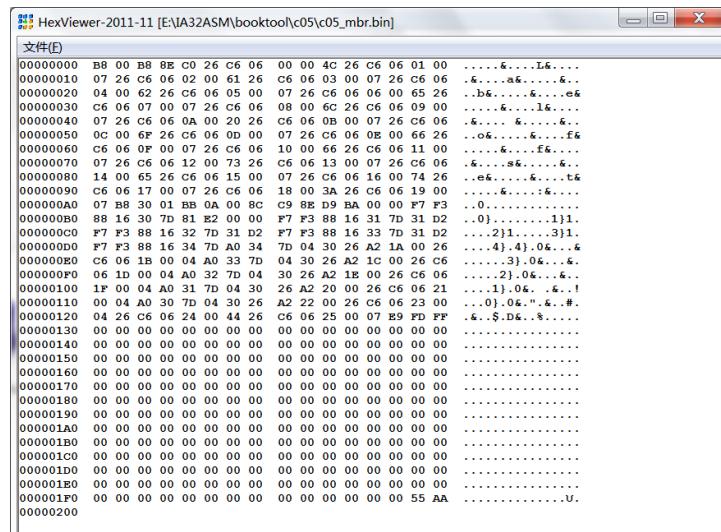


图 5-8 用配书工具 HexView 查看 c05_mbr.bin 的内容

显而易见，在编译之后，源程序中的标号、注释、伪指令都统统消失了，只剩下纯粹的机器指令和数据。那些需要在编译阶段决定的内容，也都有了确切的值。

5.7 加载和运行主引导扇区代码

5.7.1 把编译后的指令写入主引导扇区

在第4章，我们已经安装了VirtualBox虚拟机软件，并在它里面创建了一台名为LEARN-ASM的虚拟计算机。除此之外，还为它创建了一块虚拟硬盘。

虚拟硬盘其实是一个扩展名为“.vhd”的Windows文件，具体的文件名和创建位置只有你自己知道。但是，无论如何，你现在都可以将我们刚刚编译好的代码写入这个虚拟硬盘的主引导扇区里。

如图5-9所示，你首先要启动配书工具FixVhdWr，在第一个界面内选择要写入的虚拟硬盘文件。取决于你的实际情况，虚拟硬盘的文件名和路径可能与图中不同，仅供参考。



图 5-9 选择虚拟硬盘文件（参考图例）

然后，在下一个界面，选择刚刚编译好的二进制文件，如图 5-10 所示。

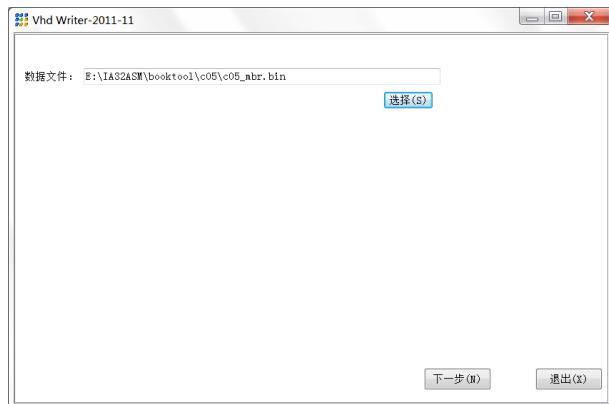


图 5-10 选择编译好的二进制文件（参考图例）

如图 5-11 所示，在最后一个界面，保持默认的选择（即选择“LBA 连续直写模式”），然后单击“写入文件”。当出现红色字体的“数据写入完成，本次共操作了 1 个扇区”时，说明数据的写入已经成功完成。



图 5-11 将二进制文件写入虚拟硬盘

最后要交待一句，千万不要在虚拟计算机 LEARN-ASM 运行的时候进行数据写入操作，因

为虚拟硬盘文件正被 VirtualBox 以独占的方式使用。否则的话，会导致数据写入失败。

5.7.2 启动虚拟机观察运行结果

在 VirtualBox 软件的主界面上，选择“LEARN-ASM”计算机，然后单击“运行”按钮。

如果你是第一次运行虚拟计算机，有可能会出现“首次运行向导”。如图 5-12 所示，这个向导程序的目的是指引你安装一个操作系统，比如 Windows 之类的。如果真的出现这么一个向导的话，直接单击“取消”按钮即可。

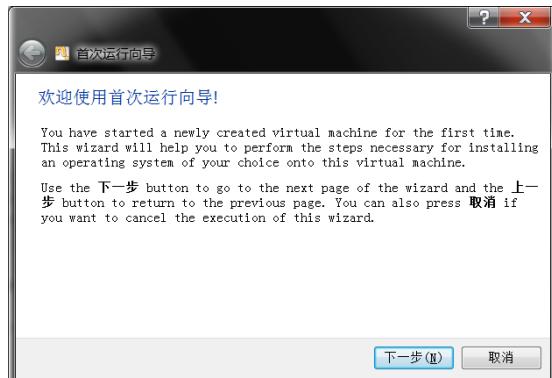


图 5-12 VirtualBox 虚拟机的首次运行向导

另外，取决于你的物理主机安装了什么类型的声卡。如果安装的是高清晰度音频（High Definition Audio, HDA），那么，虚拟机也会以此为样例创建一个虚拟声卡。

问题是，HDA 太过于智能化了，它甚至能够检测到你的扬声器和话筒是否已经插上。如果没有插上，VirtualBox 虚拟计算机在启动的过程中也会弹出一个问题报告对话框，大致的意思是有些设备不能打开，依赖于这些设备的程序可能会被挂起(待在内存里，不会得到处理器的光顾)。如果发生这样的事情，请选中“不要再显示这个信息”，然后直接单击“确定”按钮，如图 5-13 所示。

最后，如果一切顺利的话，程序的运行效果如图 5-14 所示。



图 5-13 高清晰度音频的问题报告对话框

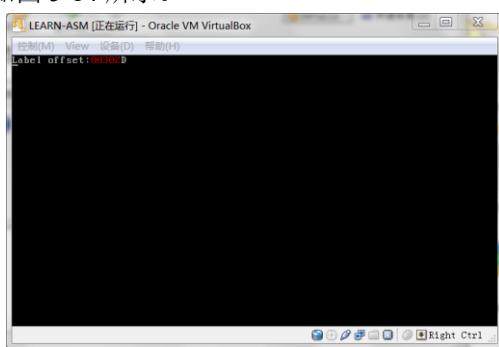


图 5-14 本章程序在虚拟计算机中的运行效果

5.7.3 程序的调试

程序员的工作就像是在历险，困难重重，途中不可避免地要遇上暗礁。有时候，少了一个字符，

或者多了一个字符，或者拼错了字符，程序就无法成功编译；有时候，尽管能够编译，但程序中存在逻辑错误，少写了语句，算法不对，运行的时候也得不到正确结果。

有时候，错误的原因很简单，就是因为马虎和误操作，但很难知道问题出在哪里。等到你终于发现的时候，一天，甚至几天的时间已经花掉了。在这种情况下，没有调试工具来找到程序中隐藏的错误是不行的。有时候，即使有调试工具的帮助，也会令人筋疲力尽，不过有总比没有好。

调试工具并不是智能到可以自动发现程序中的错误，这是不可能的。但是，它可以单步执行你的程序（每执行一条指令后就停下来），或者允许你在程序中设置断点，当它执行到断点位置时就停下来。这时，它可以显示处理器各个寄存器的内容，或者内存单元里的内容。因此，你可以根据机器的状态来判断程序的执行结果是否达到了预期。通过这种方式，你可以逐步逼近出现问题的地方，直到最终发现问题的所在。市面上有多种流行的程序调试工具软件，但它们通常都像你用的其它软件一样工作在操作系统之上。麻烦的是，本书中的程序全都只能运行在没有操作系统的裸机下。这意味着，所有流行的调试工具都不可用。不过，好消息是，一款叫做 bochs 的软件可以帮助你。

bochs 是开源软件，是你唯一可选择的调试器。开源意味着，你不用花钱购买就可以使用它。它用软件来模拟处理器取指令和执行指令的过程，以及整个计算机硬件。当它开始运行时，就直接模拟计算机的加电启动过程。正是因为如此，它才有可能做一些调试工作。

很重要的一点是，它本身就是一个虚拟机，类似于 VirtualBox。因此，它也就很容易让你单步跟踪硬盘的启动过程，查看寄存器的内容和机器状态。在本书中，我们的程序都是直接从 BIOS 那里接管处理器的控制权，因此，bochs 的这个特点正好能够用来完成调试工作。不像本书中使用的其他工具，bochs 的使用方法在网上很容易搜索到。网友王南洋为本书制作了一个 bochs 的简易教程，它的下载地址是：

<http://download.csdn.net/detail/sholber/4622176>



bochs 本身的下载地址是：

<http://bochs.sourceforge.net/>

最后要特别说明的是，bochs 要求的虚拟硬盘文件是 img 格式的，但允许你使用 VHD 文件，只是在启动时，会有一个小小的警告，不影响正常工作。

本 章 习 题

- 试找出以下程序片断中隐藏的问题并进行修正：

```
mov ax,21015
mov bl,10
div bl
and cl,0xf0
```

- 本章的程序在内存中的加载地址是 0x0000:0x7C00，此时，指令 jmp near infi 在段内的偏移地址是多少？试修改本章的源程序以显示该值。

- 汇编语言编译器采用助记符来方便指令的书写和阅读。比如，mov 是传送指令，div 是除法指令。假如 Intel 公司新推出一款处理器，该处理器新增了一条指令，其机器码为 CD 88。因为是新指令，你的 NASM 编译器肯定没有一个助记符与之相对应。在这种情况下，如何在你的程序中使用该指令？

第6章 相同的功能，不同的代码

汇编语言是最有效率的计算机语言，由于直接面向处理器编程，编译后的机器代码执行起来速度也是最快的。为了进一步讲解汇编语言的指令和语法，在本章里，我们采用不同的方法来实现和上一章相同的功能。总是一成不变地做事情是不对的，在生活中，我们需要根据不同的情况分别做出不同的应对，在计算机中，指令的执行并非总是按照它们的自然排列顺序来进行的，其执行流程也会因为各种原因发生变化。在本章，我们就来学习三种不同的程序流程控制方法。

不同的方法需要不同的指令，甚至还要引入更多的指令。与此同时，让大家见识并比较它们的不同之处，相信随着经验的增长，孰优孰劣，自有判断。

6.1 代码清单 6-1



本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：6-1（主引导扇区程序）

源程序文件：c06_mbr.asm

6.2 跳过非指令的数据区

如代码清单 6-1 所示，从源程序第 8 行到第 10 行，声明了非指令的数据。在程序的开始部分声明这些不可执行的内容是不安全的，为此，在这些数据之前，源程序的第 6 行，是一条转移指令 jmp near start，用来使处理器的执行流越过这些不可执行的数据，转移到后面的代码处执行。

正如我们在上一章里讲到的，像 jmp near start 这种指令，机器指令的操作码是 0xE9，操作数是一个 16 位的相对偏移量。

6.3 在数据声明中使用字面值

在第 5 章中，显示字符串“Label offset:”的方法是将每个字符的 ASCII 码包含在每条指令中，即它们是作为每条指令的操作数出现的。这种方法很原始，也很笨拙。而且，如果要改变显示的内容，则必须重新编写指令，很不方便。

在本章中，我们将要改变这种做法，使得显示字符串的手段更灵活，具体做法是专门定义一个存放字符串的数据区，当要显示它们的时候，再用指令取出来，一个一个地传送到显示缓冲区。这

这样一来，负责在屏幕上显示的指令就和要显示的内容无关了。

源程序的第 8、9 行，这两行的目的是声明要显示的内容。在 NASM 里，“\”是续行符，当一行写不下时，可以在行尾使用这个符号，以表明下一行与当前行应该合并为一行。

和上一章相同，在用伪指令 db 声明字符的 ASCII 码数据时也可以使用字面值。在编译阶段，编译器将把‘L’、‘a’等转换成与它们等价的 ASCII 代码。

除了 ASCII 码，这里还声明了每个字符的显示属性值 0x07，都是已经讲过的知识，相信很好理解。

6.4 段地址的初始化

汇编语言源程序的编译符合一种假设，即编译后的代码将从某个内存段中，偏移地址为 0 的地方开始加载。这样一来，如果有一个标号“label_a”，它在编译时计算的汇编地址是 0x05，那么，当程序被加载到内存后，它在段内的偏移地址仍然是 0x05，任何使用这个标号来访问内存的指令都不会产生问题。

但是，如果程序加载时，不是从段内偏移地址为 0 的地方开始的，而是 0x7c00，那么，label_a 的实际偏移地址就是 0x7c05。这时，所有访问 label_a 的指令仍然会访问偏移地址 0x05，因为这是在编译时就决定了的。实际上，这样的问题在上一章就遇到过。在那里，因为我们已经知道程序将来的加载位置是 0x0000:0x7c00，所以才有了这样古怪的写法：

```
mov [0x7c00+number+0x00], dl
```

不得不说，0x7c00 就是理论和现实之间的差距。

在主引导程序中，访问内存的指令很多，如果都要加上 0x7c00 无疑是很麻烦的，这个我们已经看到了。其实，产生这个问题的根源，就是因为程序在加载时，没有从段内偏移地址为 0 的地方开始。

好在 Intel 处理器的分段策略还是很灵活的，逻辑地址 0x0000:0x7c00 对应的物理地址是 0x07c00，该地址又是段 0x07C0 的起始地址。因此，这个物理地址其实还对应着另一个逻辑地址 0x07c0:0000，如图 6-1 所示。

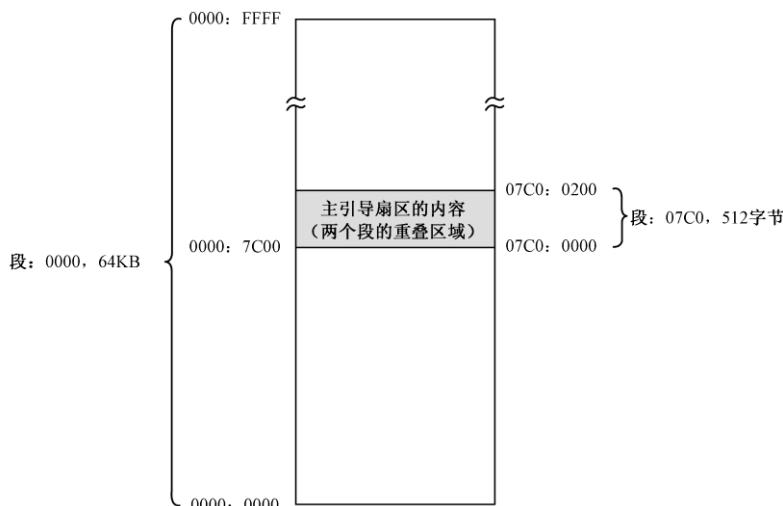


图 6-1 以两个逻辑段的视角看待同一个内存区域

看到了吧？我们可以把这个 512 字节的区域看成一个单独的段，段的基地址是 0x07C0，段长 512 字节。注意，该段的最大长度可以为 64KB。尽管 BIOS 将主引导扇区加载到物理地址 0x07c00 处，但我们却可以认为它是从 0x07c0:0x0000 处开始加载的。

在这种情况下，如果执行指令

```
mov [0x05], dl
```

那么，处理器将把数据段寄存器 DS 的内容(0x07c0)左移 4 位，加上指令中指定的偏移地址(0x05)，形成物理内存地址 0x07c05，并将寄存器 DL 中的内容传送到该处。

所以，源程序第 13、14 行，通过传送指令将数据段寄存器 DS 的内容设置为 0x07c0。和以前一样，源程序第 16、17 行，使附加段寄存器 ES 的内容指向显示缓冲区所在的段 0xb800。

6.5 段之间的批量数据传送

在本章中，要在屏幕上显示的内容，连同它们的显示属性值，都集中声明在一起。想显示它们？那就要将它们“搬”到 0xB800 段。有多种方法可以做到这一点，但 8086 处理器提供了最好的方法，那就是使用 movsb 或者 movsw 指令。

这两个指令通常用于把数据从内存中的一个地方批量地传送（复制）到另一个地方，处理器把它们看成是数据串。但是，movsb 的传送是以字节为单位的，而 movsw 的传送是以字为单位的。

movsb 和 movsw 指令执行时，原始数据串的段地址由 DS 指定，偏移地址由 SI 指定，简写为 DS:SI；要传送到的目的地址由 ES:DI 指定；传送的字节数（movsb）或者字数（movsw）由 CX 指定。除此之外，还要指定是正向传送还是反向传送，正向传送是指传送操作的方向是从内存区域的低地址端到高地址端；反向传送则正好相反。正向传送时，每传送一个字节（movsb）或者一个字（movsw），SI 和 DI 加 1 或者加 2；反向传送时，每传送一个字节（movsb）或者一个字（movsw）时，SI 和 DI 减去 1 或者减去 2。不管是正向传送还是反向传送，也不管每次传送的是字节还是字，每传送一次，CX 的内容自动减一。

如图 6-2 所示，在 8086 处理器里，有一个特殊的寄存器，叫做标志寄存器 FLAGS。作为一个例子，它的第 6 位是 ZF（Zero Flag），即零标志。当处理器执行一条算术或者逻辑运算指令后，算术逻辑部件送出的结果除了送到指令中指定位置（目的操作数指定的位置）外，还送到一个或非门。学过逻辑电路课程，或者看过《穿越计算机的迷雾》这本书的人都知道，或非门的输入全为 0 时，输出为 1；输入不全为 0，或者全部为 1 时，输出为 0。或非门的输出送到一个触发器，这就是标志寄存器的 ZF 位。这就是说，如果计算结果为 0，这一位被置成 1，表示计算结果为零是“真”的；否则清除此位（0）。

除此之外，它也允许通过指令设置一些标志，来改变处理器的运行状态。比如，第 10 位是方向标志 DF（Direction Flag），通过将这一位清零或者置 1，就能控制 movsb 和 movsw 的传送方向。

源程序第 19 行是方向标志清零指令 cld。这是个无操作数指令，与其相反的是置方向标志指令 std。cld 指令将 DF 标志清零，以指示传送是正方向的。

源程序第 20 行，设置 SI 寄存器的内容到源串的首地址，也就是标号 mytext 处的汇编地址。

源程序第 21 行，设置目的地的首地址到 DI 寄存器。屏幕上第一个字符的位置对应着 0xB800 段的开始处，所以设置 DI 的内容为 0。

第 22 行，设置要批量传送的字节数到 CX 寄存器。因为数据串是在两个标号 number 和 mytext

之间声明的，而且标号代表的是汇编地址，所以，汇编语言允许将它们相减并除以 2 来得到这个数值。需要说明的是，这个计算过程是在编译阶段进行的，而不是在指令执行的时候。除以 2 的原因是每个要显示的字符实际上占两字节：ASCII 码和属性，而 movsw 每次传送一个字。

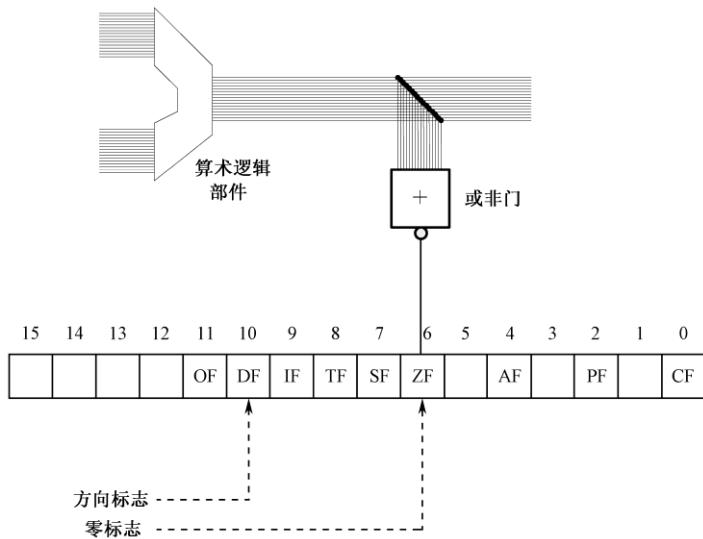


图 6-2 8086 处理器的标志寄存器

第 23 行，是 movsw 指令，操作码是 0xA5，该指令没有操作数。使用 movsw 而不是 movsb 的原因是每次需要传送一个字（ASCII 码和属性）。单纯的 movsw 只能执行一次，如果希望处理器自动地反复执行，需要加上指令前缀 rep (repeat)，意思是 CX 不为零则重复。rep movsw 的操作码是 0xF3 0xA5，它将重复执行 movsw 直到 CX 的内容为零。

6.6 使用循环分解数位

为了显示标号 number 所代表的汇编地址，源程序第 26 行用于将它的数值传送到寄存器 AX，这个和以前是一样的。

声明标号 number 并从此处开始初始化 5 字节的目的主要是保存数位，但同时我们还想显示它的汇编地址。为了访问标号 number 处的数位，需要获取它在内存段中的偏移地址。

为此，源程序第 29 行，通过将 AX 的内容传送到 BX，来使 BX 指向该处的偏移地址。实际上，这等效于

```
mov bx, number
```

只不过用寄存器传递来得更快，更方便。

第 29~37 行依旧做的是分解数位的事，但用了和以往不同的方法。简单地说，就是循环。循环依靠的是循环指令 loop，该指令出现在源程序的第 37 行：

```
loop digit
```

loop 指令的功能是重复执行一段相同的代码，处理器在执行它的时候会顺序做两件事：

将寄存器 CX 的内容减一；

如果 CX 的内容不为零，转移到指定的位置处执行，否则顺序执行后面的指令。

和源程序第 6 行的 jmp near start 一样，loop digit 指令也是颇具迷惑性的指令，它的机器指令操

作码是 0xE2，后面跟着一个字节的操作数，而且也是相对于标号处的偏移量，是在编译阶段，编译器用标号 digit 所在位置的汇编地址减去 loop 指令的汇编地址，再减去 loop 指令的长度（2）来得到的。

为了使 loop 指令能正常工作，需要一些准备。源程序第 30 行，将循环次数传送到 CX 寄存器。因为分解 AX 中的数需要循环 5 次，故传送的值是 5。

源程序第 31 行，将除数 10 传送到寄存器 SI。

源程序第 33~37 行是循环体，每次循环都会执行这些代码，主要是做除法并保存每次得到的余数。每次除法之前都要先将 DX 清零以得到被除数的高 16 位，这是源程序第 33 行所做的事情。

做完除法之后，第 35 行，将 DL 中得到的余数传送到由 BX 所指示的内存单元中去。这是我们第一次接触到偏移地址来自于寄存器的情况，而在此之前，我们仅仅是使用类似于下面的指令：

```
mov [0x05], dl
mov [number], al
mov [number+0x02], cl
```

尽管方式不同，但 mov [bx],dl 做相同的事情，那就是把 DL 中的内容，传送到以 DS 的内容为段地址，以 BX 的内容为偏移地址的内存单元中去。注意，指令中的中括号是必需的，否则就是传送到 BX 中，而不是 BX 的内容所指示的内存单元了。

在 8086 处理器上，如果要用寄存器来提供偏移地址，只能使用 BX、SI、DI、BP，不能使用其他寄存器。所以，以下指令都是非法的：

```
mov [ax], dl
mov [dx], bx
```

原因很简单，寄存器 BX 最初的功能之一就是用来提供数据访问的基址，所以又叫基址寄存器（Base Address Register）。之所以不能用 SP、IP、AX、CX、DX，这是一种硬性规定，说不上有什么特别的理由。而且，在设计 8086 处理器时，每个寄存器都有自己的特殊用途，比如 AX 是累加器（Accumulator），与它有关的指令还会做指令长度上的优化（较短）；CX 是计数器（Counter）；DX 是数据（Data）寄存器，除了作为通用寄存器使用外，还专门用于和外设之间进行数据传送；SI 是源索引寄存器（Source Index）；DI 是目标索引寄存器（Destination Index），用于数据传送操作，我们已经在 movsb 和 movsw 指令的用法中领略过了。

做完一次除法，并保存了数位之后，源程序第 36 行，用于将 BX 中的内容加一，以指向下一个内存单元。inc 是加一指令，操作数可以是 8 位或者 16 位的寄存器，也可以是字节或者字内存单元。从功能上讲，它和

```
add bx, 1
```

是一样的，但前者的机器码更短，速度更快。下面是两个例子：

```
inc al
inc word [label_a]
```

这上面的第 2 条指令，使用了关键字“word”，表明它操作的是内存中的一个字，段地址在段寄存器 DS 中，偏移地址等于标号 label_a 在编译阶段的汇编地址。

源程序第 37 行，正是 loop 指令。就像我们刚才说的，它将 CX 的内容减一，并判断是否为零。如果不为零，则跳转到标号 digit 所在的位置处执行。

很显然，在指令的地址部分使用寄存器，而不是数值或者标号（其实标号是数值的等价形式，在编译后也是数值）有一个明显的好处，那就是可以在循环体里方便地改变偏移地址，如果使用数值就不能做到这一点。

6.7 计算机中的负数

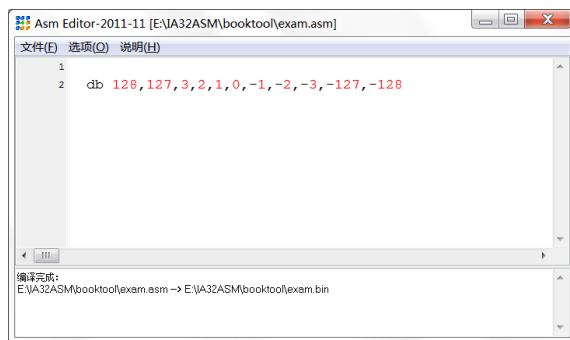
6.7.1 无符号数和有符号数

为了讲解后面的内容时能够顺利一些，现在我们离开源程序，来介绍一些题外的知识。

从本书的开篇到现在，我们一直没有提到负数，就好象世界上根本没有负数一样。计算机当然要处理负数，要不然它将没有多少实用价值。

在计算机中使用负数，这是一个容易令人产生迷惑的话题。不信？现在就开始了。

尽管我们从来没有考虑过数的正负问题，但是，事实上，我们在编写程序的时候，既可以用正数，也可以使用负数。如图 6-3 所示，我们在程序中用伪指令 db 声明了一些正数和一些负数。



```
Asm Editor-2011-11 [E:\IA32ASM\booktool\exam.asm]
文件(F) 选项(O) 说明(H)
1
2 db 128,127,3,2,1,0,-1,-2,-3,-127,-128
```

编译完成：
E:\IA32ASM\booktool\exam.asm -> E:\IA32ASM\booktool\exam.bin

图 6-3 在汇编源程序中使用负数的例子

图 6-4 显示了编译后的结果。用伪指令 db 声明的数据都只有一个字节的长度，所以很容易在这两幅图的各个数之间建立对应关系。

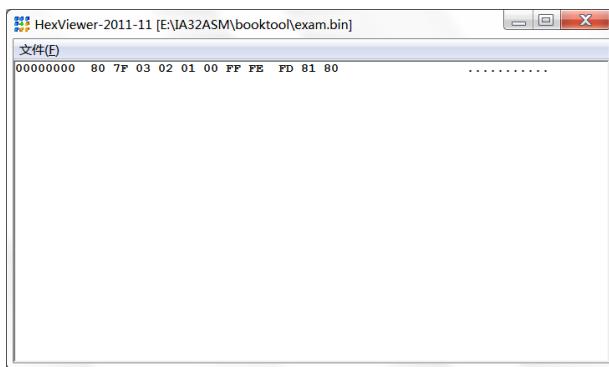


图 6-4 正数和负数编译后的结果

前面的正数都很好理解，十进制数 128 对应的二进制数是 10000000，对应的十六进制数是 0x80；十进制数 0 对应的二进制数是 00000000，对应的十六进制数是 0x00。为什么我们对此不感到新鲜？因为这显得非常自然，从本书一开始到现在，我们就是这样工作的。

真正的麻烦在于后面的负数，比如 -1，它在编译的时候，编译器会怎么做呢？

它很笨，但也很聪明。因为 -1 其实等于 0-1，它就知道可以做一次减法。当然，这个减法，

不是你已经熟悉的十进制减法，这没有用，你得做二进制的减法，也就是用二进制数 0 减去二进制数 1，结果是

注意左边的省略号，这是因为在相减的过程中，不停地向左边借位的结果。因此，可以说，这个数字是很长的，取决于你什么时候停止借位。

再比如十进制数-2，可以用 $0 - 2$ 来得到，在二进制的世界里，该减法是二进制数0减去二进制数10，结果是

同样，相减的过程要向左借位，所以这个数字相当长。但是，最右边那一位是 0。

在计算机中，数字保存在寄存器里，而在 16 位处理器里，寄存器通常是 8 位和 16 位的。因此，以上相减的结果，只能保留最右边的 8 位或者 16 位。举个例子，十进制数 -1 在寄存器 AL 中的二进制形式是

11111111

即 0xFF；十进制数 -2 在寄存器 AL 中的二进制形式是

11111110

即 0xFE。如果是 16 位的寄存器，则相应地，要保留相减结果的最右边 16 位。因此，十进制数 -1 在 AX 寄存器中的二进制形式是

1111111111111111

即 0xFFFF；十进制数 -2 在寄存器 AX 中的二进制形式是

1111111111111110

即 0xFFFFE。

当然，数据还可以保存在内存中，或者编译后的二进制文件中。在二进制文件中，数据是用伪指令 db 或者 dw 等定义的。但是，数据的表示形式和它们在寄存器中的形式相同，以下代码片段很清楚地说明了这一点。

```
data0 db -1      ; 初始化为 0xFF
data1 db -2      ; 初始化为 0xFE
data2 dw -1      ; 初始化为 0xFFFF
data3 dw -2      ; 初始化为 0xFFFFE
```

这是很令人吃惊的。因为我们知道，0xFF 等于十进制数 255，但现在它又是十进制数 -1，哪一个才是正确的呢？我们应该以哪一个为准呢？

好吧，假设这勉强能接受的话，那么，对照一下图 6-3 和图 6-4，你会发现，0x80 既是十进制数 128，又是十进制数 -128，到底哪一个是正确的呢？

这真是令人头疼的问题，不单单是对我们，对几十年前那些计算机工程师们来说也是如此。

一个良好的解决方案是，将计算机中的数分成两大类：无符号数和有符号数。无符号数的意思是我们不关心这些数的符号，因此也就无所谓正负，反正它们就是数而已，就像小学生一样，眼中只有自然数。在 8 位的字节运算中，无符号数的范围是 00000000~11111111，即十进制的 0~255；在 16 位的字运算中，无符号数的范围是 0000000000000000~1111111111111111，即十进制的 0~65535；在将来要讲到的 32 位运算中，无符号数的范围是 00000000000000000000000000~11111111111111111111111111111111，即十进制的 0~4294967295。很显然，我们以前使用的一直是无符号数。

相反地，有符号数是分正、负的，而且规定，数的正负要通过它的最高位来辨别。如果最高位是 0，它就是正数；如果是 1，就是负数。如此一来，在 8 位的字节运算环境中，正数的范围是 00000000~01111111，即十进制的 0~127；负数的范围是 10000000~11111111，即十进制的 -128~-1。

正的有符号数，和与它同值的无符号数相同，这没什么好说的，毕竟它们形式上相同，按相同的方式处理最为方便。但是，负数就不同了，在这里，10000000~11111111 这些负数，都是用 0 减去它们相对应的正数得到的。想知道它们各自对应的正数是谁吗？很简单，因为“负数的负数”是正数，所以只需要用 0 减去这个负数就行。所以，你可以试试看，因为

00000000-10000000=10000000 (十进制数 128)

00000000~11111111=00000001（十进制数1）

所以，10000000~11111111这个范围内的有符号数，对应着十进制数-128~-1。

顺便说一下，在8086处理器中，有一条指令专门做这件事，它就是neg。neg指令带有一个操作数，可以是8位或者16位的寄存器，或者内存单元。如

```
neg al
neg dx
neg word [label_a]
```

它的功能很简单，用0减去指令中指定的操作数。例子：如果AL中的内容是00001000（十进制数8），执行neg al后，AL中的内容变为11111000（十进制数-8）；如果AL中的内容为11000100（十进制数-60），执行neg al后，AL中的内容为00111100（十进制数60）。

相应地，在16位的字运算环境中，正数的范围是0000000000000000~0111111111111111，即十进制的0~32767，负数的范围是1000000000000000~1111111111111111，即十进制的-32768~-1。

不要给计算机和编译器添麻烦。既然你已经知道一个字节可以容纳的数据范围是十进制的-128~127，就不要这样写：

```
mov al,-200
```

寄存器AL只有8位，因此，编译后，-200将被截断，机器码为B0 38。你可以这样写：

```
mov ax,-200
```

这时，编译后的机器码为B8 38 FF。

同样的规则也适用于伪指令db和dw。举例（以下均为十进制数）：

db 255	;正确，可以看成声明无符号数
db -125	;正确，数据未超范围
db -240	;错误，超过字节所能容纳的数据范围，会被截断
dw -240	;正确，数据未超范围
dw -30001	;正确，数据未超范围

32位有符号数是16位和8位有符号数的超集，16位有符号数又是8位有符号数的超集，它们互相之间有重叠的部分。正数还好说，十进制数15，在8位运算环境中是00001111，在16位运算环境中是0000000000001111，没有什么区别。但是，同一个负数，其表现形式略有差别。比如十进制数-3，它在8位运算中是11111101，即0xFD；在16位运算中，则是1111111111111101，即0xFFFF。这种差别的来源很简单，我们已经讲过了，在计算机中，-3是用0减去3得到的，在8位运算中只能保留结果的低8位，即11111101（0xFD）；在16位运算中只能保留结果的低16位，即1111111111111101（0xFFFF）。

很显然，一个8位的有符号数，要想用16位的形式来表示，只需将其最高位，也就是用来辨别符号的那一位（几乎所有的书上都称之为符号位，实际上这并不严谨），扩展到高8位即可。为了方便，处理器专门设计了两条指令来做这件事：cbw(Convert Byte to Word)和 cwd(Convert Word to Double-word)。

cbw没有操作数，操作码为98。它的功能是，将寄存器AL中的有符号数扩展到整个AX。举个例子，如果AL中的内容为01001111，那么执行该指令后，AX中的内容为000000001001111；如果AL中的内容为10001101，执行该指令后，AX中的内容为111111110001101。

cwd也没有操作数，操作码为99。它的功能是，将寄存器AX中的有符号数扩展到DX:AX。举个例子，如果AX中的内容为010011110111001，那么执行该指令后，DX中的内容为0000000000000000，AX中的内容不变；如果AX中的内容为1000110110001011，那么执行该指令

后，DX 中的内容为 1111111111111111，AX 中的内容同样不变。

尽管有符号数的最高位通常称为符号位，但并不意味着它仅仅用来表示正负号。事实上，通过上面的讲述和实例可以看出，它既是数的一部分，和其他比特一起共同表示数的大小，同时又用来判断数的正负。

6.7.2 处理器视角中的数据类型

无符号数和有符号数的划分并没有从根本上打消我们的疑虑，即假如寄存器 AX 中的内容是 0xB23C，那么，它到底是无符号数 45628 呢，还是应当将其看成是 -19908？

答案是，这是你自己的事，取决于你怎么看待它。对于处理器的多数指令来说，执行的结果和操作数的类型没有关系。换句话说，无论你是从无符号数的角度来看，还是从有符号数的角度来看，指令的执行结果都是正确无误的。比如

```
mov ah, al
```

这条指令显然根本不考虑操作数的类型。再比如

```
mov ah, 0xf0
```

```
inc ah
```

在这里，0xf0 的二进制形式是 11110000，它既可以解释为无符号数 240（十进制），也可以解释为有符号数 -16，毕竟它的符号位是 1。无论如何，inc 是加一指令，这条指令执行后，AH 中的内容是二进制数 11110001，既是无符号数 241，也是有符号数 -15。

再考虑加法运算。比如

```
mov ax, 0x8c03
```

```
add ax, 0x05
```

0x8c03 的二进制形式是 1000110000000011，既可以看做是无符号数 35843（十进制），也可以看成是有符号数 -29693（十进制）。在运算过程中，数的视角要统一，如果把 0x8c03 看成是无符号数，那么 0x05 也是无符号数；如果 0x8c03 是有符号数，那么 0x05 也是有符号数。

关键是运算后的结果。很幸运的是，add 指令同样适用于无符号数和有符号数。所以，这两条指令执行后，AX 中的内容是 0x8c08，分别可以看成是无符号数 35848 和有符号数 -29688。

再来考虑一下减法。考虑一下，如果要计算 $10 - 3$ ，这其实可以看成是 $10 + (-3)$ 。因此，使用以下三条指令就可以完成减法运算：

```
mov ah, 10
```

```
mov al, -3
```

```
add ah, al
```

正是因为这个原因，很多处理器内部不构造减法电路，而是使用加法电路来做减法。

尽管如此，为了方便起见，处理器还是提供了减法指令 sub，该指令和加法指令 add 相似，目的操作数可以是 8 位或者 16 位通用寄存器，也可以是 8 位或者 16 位的内存单元；源操作数可以是通用寄存器，也可以是内存单元或者立即数（不允许两个操作数同时为内存单元）。比如

```
sub ah, al
```

```
sub dx, ax
```

```
sub [label_a], ch
```

因为处理器没有减法运算电路，所以，举例来说，sub ah,al 指令实际上等效于下面两条指令：

```
neg al
```

```
add ah,al
```

可以这么说，几乎所有的处理器指令既能操作无符号数，又能操作有符号数。但是，有几条指令除外，比如除法指令和乘法指令。

我们已经学过除法指令 div。严格地说，它应该叫做无符号除法指令（Unsigned Divide），因为这条指令只能工作于无符号数。换句话说，只有从无符号数的角度来解释它的执行结果才能说得通。举个例子：

```
mov ax,0x0400  
mov bl,0xf0  
div bl           ;执行后, AL 中的内容为 0x04, 即十进制数 4
```

从无符号数的角度来看，0x0400 等于十进制数 1024，0xf0 等于十进制数 240。相除后，寄存器 AL 中的商为 0x04，即十进制数 4，完全正确。

但是，从有符号数的角度来看，0x0400 等于十进制数 1024，0xf0 等于十进制数 -16。理论上，相除后，寄存器 AL 中结果应当是 0xc0。因其最高位是“1”，故为负数，即十进制数为 -64。

为了解决这个问题，处理器专门提供了一个有符号数除法指令 idiv（Signed Divide）。idiv 的指令格式和 div 相同，除了它是专门用于计算有符号数的。如果你决定要进行有符号数的计算，必须采用如下代码：

```
mov ax,0x0400  
mov bl,0xf0  
idiv bl           ;执行后, AL 中的内容为 0xc0, 即十进制数 -64
```

在用 idiv 指令做除法时，需要小心。比如用 0xf0c0 除以 0x10，也就是十进制数的除法 $-3904 \div 16$ 。你的做法可能会是这样的：

```
mov ax,0xf0c0  
mov bl,0x10  
idiv bl
```

以上的代码是 16 位二进制数除法，结果在寄存器 AL 中。除法的结果应当是十进制数 -244，遗憾的是，这样的结果超出了寄存器 AL 所能表示的范围，必然因为溢出而不正确。为此，你可能会用 32 位的除法来代替以前的做法：

```
xor dx,dx           ;如此一来, DX: AX 中的数成了正数  
mov ax,0xf0c0  
mov bx,0x10  
idiv bl
```

很遗憾，这依然是错的。十进制数 -3904 的 16 位二进制形式和 32 位二进制形式是不同的。前者是 0xf0c0，后者是 0xfffff0c0。还记得 cwd 吗？你应该用这条指令把寄存器 AX 中数的符号扩展到 DX。所以，完全正确的写法是这样的：

```
mov ax,0xf0c0  
cwd  
mov bx,0x10  
idiv bx
```



以上指令全部执行后，寄存器 AX 中的内容为 0xff0c，即十进制数 -244。

主动权在你自己手上，在写程序的时候，你要做什么，什么目的，你自己最清楚。如果是无符号数计算，必须使用 div 指令；如果你是在做有符号数计算，就应当使用 idiv 指令。

6.8 数位的显示

一旦各个数位都分解出来了，下面的工作就是在屏幕上显示它们。源程序第 40 行，将保存有各个数位的数据区首地址传送到基址寄存器 BX。

一共有 5 个数字要显示，其偏移地址分别是 BX (BX+0)、BX+1、BX+2、BX+3、BX+4。在这里，BX 是基地址，一般保持不变，如果令寄存器 SI 的内容为 4，并使 SI 的内容每次减一，则可以通过 BX+SI 来连续访问这 5 个数字。在这里，SI 的作用相当于索引，因此它被称为索引寄存器 (Index Register)，或者叫变址寄存器。另一个常用的变址寄存器是 DI。

因此，源程序第 41 行，把初始的索引值 4 传送到 SI 寄存器，这是由于要先显示万位上的数字。

源程序第 43 行，从指定的内存单元取出一个字节，传送到 AL 寄存器，偏移地址是 BX+SI。但是，它们之间的运算并非是在编译阶段进行的，而是在指令实际执行的时候，由处理器完成的。

源程序第 44 行，将 AL 中的数字加上 0x30，以得到它对应的 ASCII 码。

源程序第 45 行，将数字 0x04 传送到寄存器 AH。0x04 是显示属性，即前面讲过的黑底红字，无加亮，无闪烁。到此，AX 中是一个完整的字，前 8 位是显示属性值，后 8 位是字符的 ASCII 码。

源程序第 46 行，将 AX 中的内容传送到由段寄存器 ES 所指向的显示缓冲区中，偏移地址由 DI 指定。还记得吗，在前面使用 movsw 传送字符串“Label offset:”到显示缓冲区时，也使用了 DI，当时 DI 是指向显示缓冲区首地址的 (0)，而且每传送一次就自动加 2。传送结束后，DI 正好指向字符“：“的下一个存储单元。之后，DI 一直没用过，还保持着原先的内容。

注意，如图 6-5 所示，数据的传送是按低端字节序的，寄存器的低字节传送到显示缓冲区的低地址部分（字节），寄存器的高字节传送到显示缓冲区的高地址部分（字节）。

源程序第 47 行，将 DI 的内容加上 2，以指向显示缓冲区的下一个单元。

源程序第 48 行，将 SI 的内容减 1，使得下一次的 BX+SI 指向千位数字。dec 是减一指令，和 inc 指令一样，后面跟一个操作数，可以是 8 位或者 16 位的通用寄存器或者内存单元。

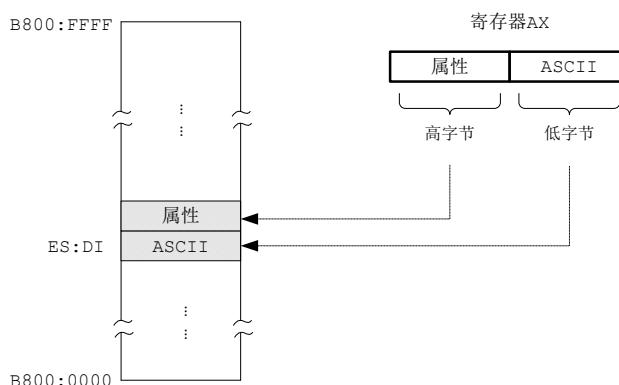


图 6-5 低端字节序的字传送示意图

源程序第 49 行，指令 jns show 的意思是，如果未设置符号位，则转移到标号“show”所在的位置处执行。如图 6-2 所示，Intel 处理器的标志寄存器里有符号位 SF (Sign Flag)，很多算术逻辑运算都会影响到该位，比如这里的 dec 指令。如果计算结果的最高位是比特“0”，处理器把 SF 位

置“0”，否则 SF 位置“1”。

处理器的任务是忠实地执行指令，多数时候，它不会知道你的意图，也不会知道你进行的是有符号数运算，还是无符号数运算。如果运算结果的最高位是“1”，它唯一所能做的，就是将 SF 标志置“1”，以示提醒，剩下的事，你自己看着办，它已经尽力了。

由于 SI 的初始值为 4，故第一次执行 dec si 后，si 的内容为 3，即二进制数 0000000000000011，符号位是比特“0”，处理器将标志寄存器的 SF 位清“0”。于是，当执行 jns show 时，符合条件，于是转移到标号“show”所在的位置处执行，等于是开始显示下一个数位。

当显示完最后一个数位后，SI 的内容是零。执行 dec si 指令后，由于产生了借位，实际的运算结果是 0xffff (SI 只能容纳 16 个比特)，因其最高位是“1”，故处理器将标志位 SF 置“1”，表明当前 SI 中的结果可以理解为一个负数 (-1)。于是，执行 jns show 时，条件不满足，接着执行后面第 51 行的指令。

jns 是条件转移指令，处理器在执行它的时候要参考标志寄存器的 SF 位。除了只是在符合条件的时候才转移之外，它和 jmp 指令很相似，它也是相对转移指令，编译后的机器指令操作数也是一个相对偏移量，是用标号处的汇编地址减去当前指令的汇编地址，再减去当前指令的长度得到的。

6.9 其他标志位和条件转移指令



在处理器内进行的很多算术逻辑运算，都会影响到标志寄存器的某些位。比如我们已经学过的加法指令 add、逻辑运算指令 xor 等。在下面的讲述中，请自行参考图 6-2。

6.9.1 奇偶标志位 PF

当运算结果出来后，如果最低 8 位中，有偶数个为 1 的比特，则 PF=1；否则 PF=0。例如：

```
mov ax,1000100100101110B      ;ax <- 0x892e
xor ax,3                      ;结果为 0x892d
```

顺序执行以上两条指令后，因为结果是 1000100100101101B，低 8 位是 00101110B，有偶数个 1，所以 PF=1。

再如：

```
mov ah,00100110B              ;ah <- 0x26
mov al,10000001B              ;al <- 0x81
add ah,al                     ;ah <- 0xa7
```

以上，因为最后 ah 的内容是 0xa7 (10100111B)，包含奇数个 1，故 PF=0。

6.9.2 进位标志 CF

当处理器进行算术操作时，如果最高位有向前进位或借位的情况发生，则 CF=1；否则 CF=0。比如：

```
mov al,10000000B              ;al <- 0x80
add al,al                     ;al <- 0x00
```

这里，寄存器 AL 自己和自己做加法运算，并因为最高位是 1 而产生进位。结果是，进位被丢

弃，AL 中的最终结果为零。进位的产生，使得 CF=1。同时，ZF=1，PF=1。

下面是因有借位而使得 CF 为 1 的例子：

```
mov ax, 0
sub ax, 1
```

CF 标志始终忠实地记录进位或者借位是否发生，但少数指令除外（如 inc 和 dec）。

6.9.3 溢出标志 OF

在所有的情况下，处理器都不知道你的意图，不知道你进行的是无符号数运算，还是有符号数运算。为此，它提供了这个标志。该标志的意思是，假定你进行的是有符号数运算，如果结果超出了目标操作数所能容纳的范围，OF=1；否则，OF=0。例如：

```
mov ah, 0xff
add ah, 2
```

执行以上两条指令后，进位标志 CF 为 1，这是肯定的了，因为最高位有进位。

寄存器 AH 可以容纳的数据范围是十进制的 -128~127，假如上面的运算是有符号数运算，那么，这实际上是在计算 $-1+2$ （十进制），AH 中的最终的结果是 1，没有超出 AH 所能表示的数的范围范围，因此 OF=0。

再看一个例子：

```
mov ah, 0x70
add ah, ah
```

首先，本次相加，用二进制数来说就是 $01110000 + 01110000 = 11100000$ ，最高位没有进位，故 CF=0。

其次，从无符号数的角度来看（十进制），即 $112 + 112 = 224$ ，并未超出一个字节所能容纳的数据上限 255，结果是正确的。

但是，从有符号数运算的角度来看（十进制），即 $112 + 112 = -32$ ，明显是错的。错误的原因是相加的结果（224）超出了一个字节所能容纳有符号数范围（十进制的 -128~127），所以破坏了符号位，使得结果变成了负数（-32）。在这种情况下，OF=1。

换句话说，在有符号数运算当中，溢出就意味着一个错误的计算结果。

既然如此，可以使用 16 位寄存器 AX，毕竟它能容纳的数据范围更大一些：

```
mov ax, 0x70
add ax, ax
```

这次，无论它是有符号数运算，还是无符号数运算，结果都是正确的。故 CF=0，OF=0。

6.9.4 现有指令对标志位的影响

由于是刚刚接触标志位，现将前面学过的指令对标志位的影响一一列举如下。在往后的学习中，但凡遇到新的指令时，除了讲解指令的功能和用法，也会说明其对标志位的影响。

add	OF、SF、ZF、AF、CF 和 PF 的状态依计算结果而定。
and	OF=0，CF=0；对 SF、ZF 和 PF 的影响依计算结果而定。
cbw	不影响任何标志位。
cld	DF=0，CF、OF、ZF、SF、AF 和 PF 未定义。未定义的意思是到目前为止还不打算让该指令影响到这些标志，因此，不要在程序中依赖这些标志。
cwd	不影响任何标志位。
dec	CF 标志不受影响，因为该指令通常在程序中用于循环计数，而且在循环体内通

	常有依赖 CF 标志的指令，故不希望它打扰 CF 标志；对 OF、SF、ZF、AF 和 PF 的影响依计算结果而定。
div/idiv	对 CF、OF、SF、ZF、AF 和 PF 的影响未定义。
inc	CF 标志不受影响，对 OF、SF、ZF、AF 和 PF 的影响依计算结果而定。
mov/movs	这类指令不影响任何标志位。
neg	如果操作数为 0，则 CF=0，否则 CF=1；对 OF、SF、ZF、AF 和 PF 的影响依计算结果而定。
std	DF=1，不影响其他标志位。
sub	对 OF、SF、ZF、AF、PF 和 CF 的影响依计算结果而定。
xor	OF=0，CF=0；对 SF、ZF 和 PF 依计算结果而定；对 AF 的影响未定义。

6.9.5 条件转移指令

“jcc”不是一条指令，而是一个指令族（簇），功能是根据某些条件进行转移，比如前面讲过的 jns，意思是 SF \neq 1（那就是 SF=0 了）则转移。方便起见，处理器一般提供相反的指令，如 js，意思是 SF=1 则转移。爱上网的朋友们容易把它理解成“奸商”。

在汇编语言源代码里，条件转移指令的操作数是标号。编译成机器码后，操作数是一个立即数，是相对于目标指令的偏移量。在 16 位处理器上，偏移量可以是 8 位（短转移）或者 16 位（相对近转移）。

相似地，jz 的意思是 ZF 标志为 1 则转移；jnz 的意思是 ZF 标志不为 1（为 0）则转移。

jo 的意思是 OF 标志为 1 则转移，jno 的意思是 OF 标志不为 1（为 0）则转移。

jc 的意思是 CF 标志为 1 则转移，jnc 的意思是 CF 标志不为 1（为 0）则转移。

jp 的意思是 PF 标志为 1 则转移，jnp 的意思是 PF 标志不为 1（为 0）则转移。爱上网的朋友们注意了，jp 可不是“极品”的意思。

转移指令必须出现在影响标志的指令之后，比如：

```
dec si
jns show
```

经验证明，像这种水到渠成的情况是很少的，多数时候，你会遇到一些和标志位关系不太明显的问题，比如，当 AX 寄存器里的内容为 0x30 的时候转移，或者当 AX 寄存器里的内容小于 0xf0 的时候转移，再或者，当 AX 寄存器里的内容大于寄存器 BX 里的内容时转移，这该怎么办呢？

好在处理器提供了比较指令 cmp，它需要两个操作数，目的操作数可以是 8 位或者 16 位通用寄存器，也可以是 8 位或者 16 位内存单元；源操作数可以是与目的操作数宽度一致的通用寄存器、内存单元或者立即数，但两个操作数同时为内存单元的情况除外。比如：

```
cmp al,0x08
cmp dx,bx
cmp [label_a],cx
```

cmp 指令在功能上和 sub 指令相同，唯一不同之处在于，cmp 指令仅仅根据计算的结果设置相应的标志位，而不保留计算结果，因此也就不会改变两个操作数的原有内容。cmp 指令将会影响到 CF、OF、SF、ZF、AF 和 PF 标志位。

比较是拿目的操作数和源操作数比，重点关心的是目的操作数。拿指令 cmp ax,bx 来说，我们关心的是 AX 中的内容是否等于 BX 中的内容，AX 中的内容是否大于 BX 中的内容，AX 中的内容是否

小于 BX 中的内容，等等，AX 是被测量的对象，BX 是测量的基准。比较的结果如表 6-1 所示。

表 6-1 各种比较结果和相应的条件转移指令

比较结果	英文描述	指令	相关标志位的状态
等于	Equal	je	相减结果为零才成立，故要求 ZF=1
不等于	Not Equal	jne	相减结果不为零才成立，故要求 ZF=0
大于	Greater	jg	适用于有符号数比较 要求：ZF=0（两个数不同，相减的结果不为零），并且 SF=OF（如果相减后溢出，则结果必须是负数，说明目的操作数大；如果相减后未溢出，则结果必须是正数，也表明目的操作数大些）
大于等于	Greater or Equal	jge	适用于有符号数的比较 要求：SF=OF
不大于	Not Greater	jng	适用于有符号数的比较 要求：ZF=1（两个数相同，相减的结果为零），或者 SF≠OF（如果相减后溢出，则结果必须是正数，说明源操作数大；如果相减后未溢出，则结果必须是负数，同样表明源操作数大些）
不大于等于	Not Greater or Equal	jnge	适用于有符号数的比较 要求：SF≠OF
小于	Less	jl	适用于有符号数的比较，等同于“不大于等于” 要求：SF≠OF
小于等于	Less or Equal	jle	适用于有符号数的比较，等同于“不大于” 要求：ZF=1（两个数相同，相减的结果为零），并且 SF≠OF（如果相减后溢出，则结果必须是正数，说明源操作数大；如果相减后未溢出，则结果必须是负数，同样表明源操作数大些）
不小于	Not Less	jnl	适用于有符号数的比较，等同于“大于等于” 要求：SF=OF
不小于等于	Not Less or Equal	jnle	适用于有符号数的比较，等同于“大于” 要求：ZF=0（两个数不同，相减的结果不为零），并且 SF=OF（如果相减后溢出，则结果必须是负数，说明目的操作数大；如果相减后未溢出，则结果必须是正数，也表明目的操作数大些）
高于	Above	ja	适用于无符号数的比较 要求：CF=0（没有进位或借位）而且 ZF=0（两个数不相同）
高于等于	Above or Equal	jae	适用于无符号数的比较 要求：CF=0（目的操作数大些，不需要借位）
不高于	Not Above	jna	适用于无符号数的比较，等同于“低于等于”（见后） 要求：CF=1 或者 ZF=1
不高子等于	Not Above or Equal	jnae	适用于无符号数的比较，等同于“低于”（见后） 要求：CF=1
低于	Below	jb	适用于无符号数的比较 要求：CF=1
低于等于	Below or Equal	jbe	适用于无符号数的比较 要求：CF=1 或者 ZF=1
不低于	Not Below	jnb	适用于无符号数的比较，等同于“高于等于” 要求：CF=0
不低于等于	Not Below or Equal	jnbe	适用于无符号数的比较，等同于“高于” 要求：CF=0 而且 ZF=0
校验为偶	Parity Even	jpe	要求：PF=1
检验为奇	Parity Odd	jpo	要求：PF=0

非常显而易见的是，如果你英语基础比较好，认识上面那些单词的话，这些指令都可以在短时间内轻松记住。英语基础不太好的人也不要灰心，事实上，根本不需要记住这些指令和它们的测试条件，因为我们平时很少用得了这么多。需要的时候再回过头来查查，这是个好办法，时间一长，自然就记住了。

最后一个要讲述的条件转移指令是 `jcxz` (jump if CX is zero)，意思是当 CX 寄存器的内容为零时则转移。执行这条指令时，处理器先测试寄存器 CX 是否为零。例如：

```
jcxz show
```

这里，“`show`”是程序中的一个标号。执行这条指令时，如果 CX 寄存器的内容为零，则转移；否则不转移，继续往下执行。

6.10 NASM 编译器的\$和\$\$标记

源程序第 51 行，用于在显示了各个数位之后，再显示一个字符“D”。目的地址是由 `ES:DI` 给出的，源操作数是立即数 `0x0744`，其中，高字节 `0x07` 是黑底白字的显示属性，低字节 `0x44` 是字符“D”的 ASCII 码。字的写入是按低端字节序的，请自行参照图 6-5。

整个程序到此结束。为了使处理器还有事做，源程序第 53 行，是一个无限循环。NASM 编译器提供了一个标记“\$”，该标记等同于标号，你可以把它看成是一个隐藏在当前行行首的标号。因此，`jmp near $` 的意思是，转移到当前指令继续执行，它和

```
infi: jmp near infi
```

是一样的，没有区别，但不需要使用标号，更不必为给标号起一个有意义的名字而伤脑筋。

和第 5 章一样，为了得到不多不少，正好 512 字节的编译结果，同时最后两个字节还必须是 `0x55` 和 `0xAA`，需要在所有指令的后面填充一些无用的数据。

源程序第 55 行，用于重复伪指令“`db 0`”若干次。重复的次数是由 `510-($-$)` 得到的，除去 `0x55` 和 `0xAA` 后，剩余的主引导扇区内容是 510 字节；\$是当前行的汇编地址；\$\$是 NASM 编译器提供的另一个标记，代表当前汇编节（段）的起始汇编地址。当前程序没有定义节或段，就默认地自成一个汇编段，而且起始的汇编地址是 0（程序起始处）。这样，用当前汇编地址减去程序开头的汇编地址（0），就是程序实体的大小。再用 510 减去程序实体的大小，就是需要填充的字节数。

就像处理器把内存划分成逻辑上的分段一样，源程序也应当按段来组织，划分成独立的代码段、数据段等。从本书第 8 章开始，将引入这方面的内容。

6.11 观察运行结果

编译本章的源程序，并用 `FixVhdWr` 将编译后的二进制文件写入虚拟硬盘的主引导扇区，然后启动 `VirtualBox`，观察运行后的结果。在你的程序无错的情况下，显示的效果应当如图 6-6 所示。

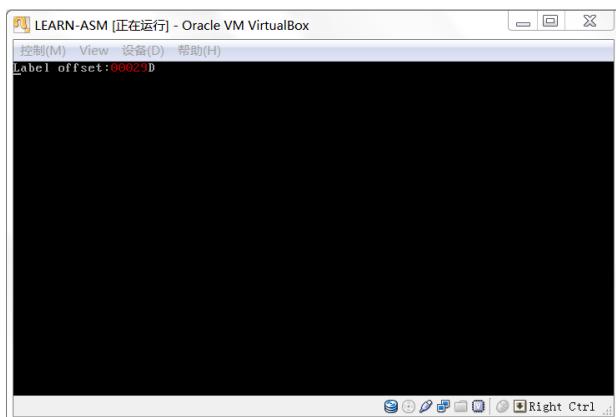


图 6-6 本章程序的运行结果

本 章 习 题

1. 在某程序中声明和初始化了以下的有符号数。请问，正数和负数各有多少？

```
data1 db 0x05,0xff,0x80,0xf0,0x97,0x30  
data2 dw 0x90,0xffff0,0xa0,0x1235,0x2f,0xc0,0xc5bc
```

2. 如果可能的话，尝试编写一个主引导扇区程序来做上面的工作。
3. 请问下面的循环将执行多少次：

```
mov cx,0  
delay: loop delay
```

第 7 章 比高斯更快的计算

7.1 从 1 加到 100 的故事

伟大的数学家高斯在 9 岁那年，用很短的时间完成了从 1 到 100 的累加。那原本是老师给学生们出的难题，希望他们能老老实实地待在教室里。

高斯的方法很简单，他发现这是 50 个 101 的求和： $100+1, 99+2, 98+3, \dots, 50+51$ ，于是他很快算出结果是 $101 \times 50 = 5050$ 。

1796 年，他 19 岁。有一天，他在导师每天布置的作业中遇到了一个前所未有的难题，那道题的要求是用圆规和一把没有刻度的直尺做出正 17 边形。这道题比老师平时布置的作业都难，但年青人不愿让自己的导师失望，花了一个晚上才解决了这个问题。

第二天，当导师看到这个结果时，惊呆了。“这道题原本不是给你的，只是一不小心夹在了给你的作业中。”导师激动地说，“你知不知道，你解开了一道有两千多年历史的数学难题，阿基米德没有做出来，牛顿也没有做出来，你竟然一个晚上就做出来了！你真是个天才！”

多年以后，高斯回忆起这一幕时说：“如果有人告诉我，这是一道有两千多年历史的数学难题，我不可能在一个晚上解决它。”

言归正传。从 1 加到 100，高斯发现了其中的规律，当然很快就能算出结果。但是计算机很蠢，它不懂什么规律，只能从 1 老老实实地加到 100。不过，它的强项就是速度，而且不怕麻烦，当高斯还在审题的时候，它就累加出结果了。

7.2 代码清单 7-1

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：7-1（主引导扇区程序）

源程序文件：c07_mbr.asm

7.3 显示字符串

源程序第 8 行，声明并初始化了一串字符（字符串），它的最终用途是要显示在屏幕上。我们可以直接用单引号把一串字符围起来，在编译阶段，编译器将把它们拆开，以形成一个个单独的字节。

为了跳过没有指令的数据区，源程序第 6 行是 `jmp near start` 指令。

源程序第 11~15 行用于初始化数据段寄存器 DS 和附加段寄存器 ES。

源程序第 18~28 行同样用于显示字符串，但采用了不同的方法，首先是用索引寄存器 SI 指向 DS 段内待显示字符串的首地址，即标号“message”所代表的汇编地址。然后，再用另一个索引寄存器 DI 指向 ES 段内的偏移地址 0 处，ES 是指向 0xB800 段的。

字符串的显示需要依赖循环。本次采用的是循环指令 loop。loop 指令的工作又依赖于 CX 寄存器，所以，源程序第 20 行，用于在编译阶段计算一个循环次数，该循环次数等于字符串的长度（字符个数）。

循环体是从源程序第 22 行开始的。首先从数据段中，逻辑地址为 DS:SI 的地方取得第一个字符，将其传送到逻辑地址 ES:DI，后者指向显示缓冲区。

紧接着，源程序第 24 行，将 DI 的内容加一，以指向该字符在显示缓冲区内的属性字节；第 25 行，在该位置写入属性值 0x07，即黑底白字。

源程序第 26、27 行，分别将寄存器 SI 和 DI 的内容加一，以指向源位置和目标位置的下一个单元。

源程序第 28 行，执行循环。loop 指令在执行时先将 CX 的内容减一，然后根据 CX 是否为零来决定是否开始下一轮循环。当 CX 为 0 的时候，说明所有的字符已经显示完毕。

7.4 计算 1 到 100 的累加和

接下来就是计算 1 到 100 的累加和了。处理器还没有智能到可以理解题意的程度，具体的计算方法和计算步骤只能由人来给出。

要计算 1 到 100 的累加和，可以采取这样的办法：先将寄存器 AX 清零，再用 AX 的内容和 1 相加，结果在 AX 中；接着，再用 AX 的内容和 2 相加，结果依旧在 AX 中，……，就这样一直加到 100。

为此，源程序第 31 行，用 xor 指令将寄存器 AX 清零；源程序第 32 行，将第一个被累加的数“1”传送到寄存器 CX。

源程序第 34 行就开始累加了，每次相加之后，源程序第 35 行，将 CX 的内容加一，以得到下一个将要累加的数。

源程序第 36 行，将 CX 的内容同 100 进行比较，看是不是已经累加到 100 了。如果小于等于 100，则继续重复累加过程，如果大于 100，就不再累加，直接往下执行。

最后，AX 中将得到最终的累加和。需要特别说明的是，AX 可以容纳的无符号数最大是 65535，再大就不行了。由于我们已经知道最终的结果是 5050，所以很放心地使用了寄存器 AX。要是你从 1 加到 1000，就得考虑使用两个寄存器来计算了。

7.5 累加和各个数位的分解与显示

7.5.1 堆栈和堆栈段的初始化

得到了累加和之后，下面的工作是将它的各个数位分解出来，并准备在屏幕上显示，好让我们知道这个数到底是多少。

和前两章不同，分解出来的各个数位并不保存在数据段中，而保存在一个叫做堆栈的地方。

堆栈（Stack）是一种特殊的数据存储结构，数据的存取只能从一端进行。这样，最先进去的数据只能最后出来，最后进去的数据倒是最先出来，这称为后进先出（Last In First Out, LIFO）。如图 7-1 所示，可以把堆栈看成一个一端开口的塑料瓶，1 号球最先放进去，3 号球最后放进去，只能在 3 号球和 2 号球分别取出后，才能把 1 号球取出来。

听起来像是在讲如何往盒子里放东西，或者从盒子里取东西。实际上，我们还是在讲内存，只不过是另一种特殊的读写方式而已。

和代码段、数据段和附加段一样，堆栈也被定义成一个内存段，叫堆栈段（Stack Segment），由段寄存器 SS 指向。

针对堆栈的操作有两种，分别是将数据推进堆栈（push）和从堆栈中弹出数据（pop）。简单地说，就是压栈和出栈。压栈和出栈只能在一端进行，所以需要用堆栈指针寄存器 SP（Stack Pointer）来指示下一个数据应当压入堆栈内的什么位置，或者数据从哪里出栈。

定义堆栈需要两个连续的步骤，即初始化段寄存器 SS 和堆栈指针 SP 的内容。源程序第 40~42 行用于将堆栈段的段地址设置为 0x0000，堆栈指针的内容设置为 0x0000。

到目前为止，我们已经定义了 3 个段，图 7-2 是当前的内存布局。总的内存容量是 1MB，物理地址的范围是 0x00000~0xFFFFF，其中，假定数据段的长度是 64KB（实际上它的长度无关紧要），占据了物理地址 0x07C00~0x17BFF，对应的逻辑地址范围是 0x07C0:0x0000~0x07C0:0xFFFF；代码段和堆栈段是同一个段，占据着物理地址 0x00000~0x0FFFF，对应的逻辑地址范围是 0x0000:0x0000~0x0000:0xFFFF。

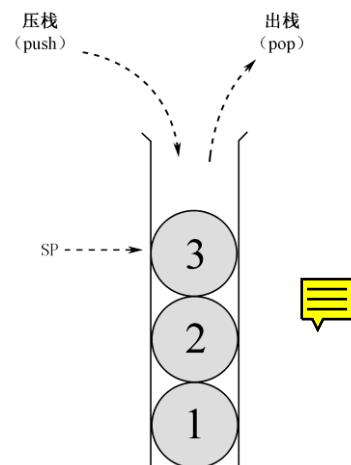


图 7-1 一个说明堆栈工作原理的类比



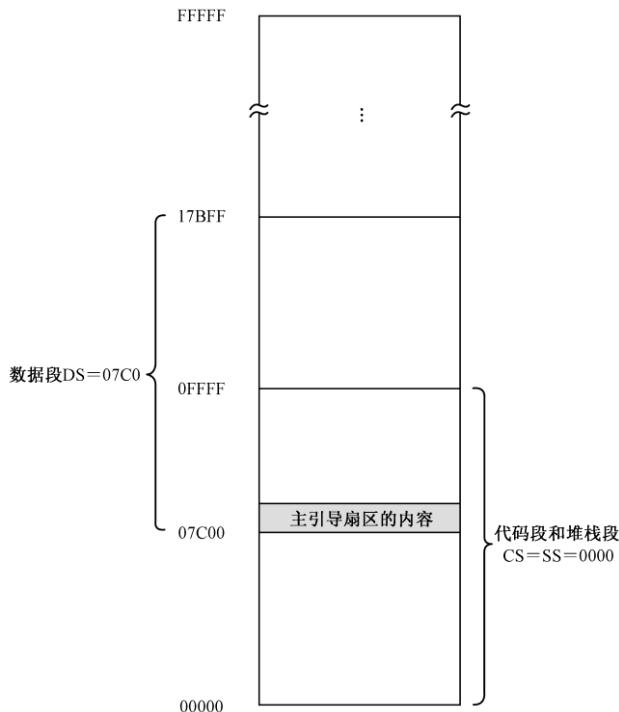


图 7-2 本章程序的内存布局

虽然代码段和堆栈段在本质上指向同一块内存区域，但是不要担心，主引导程序只占据着中间的一小部分，我们有办法让它们互不干扰。

7.5.2 分解各个数位并压栈

数位的分解还是得靠做除法。源程序第 44 行用于把除数 10 传送到寄存器 BX。

以往分解寄存器 AX 中的数时，固定是分解 5 次，得到 5 个数位。但这也存在一个缺点，如果 AX 中的数很小时，在屏幕上显示的数左边都是“0”，这当然是很别扭的。为此，本章的源程序做了改善，每次除法结束后，都做一次判断，如果商为 0 的话，分解过程可以提前结束。

但是，由于每次得到的数位是压入堆栈的，将来还要反序从堆栈中弹出，为此，必须记住实际上到底有多少个数位。源程序第 45 行，将寄存器 CX 清零，并在后面的代码中用于累计有多少个数位。

源程序第 47~53 行也是一个循环体，每执行一次，分解出一个数位。每次分解时，CX 加一，表明数位又多了一个，这是源程序第 47 行所做的事。

源程序第 48、49 行，将 DX 清零，并和 AX 一起形成 32 位的被除数。

分解出的数位将来要显示在屏幕上，为了方便，源程序第 50 行，直接将 AL 中的商“加上”0x30，以得到该数字所对应的 ASCII 码。

注意上一段话中的引号。这并不是真正的加法，or 并不是相加的指令，但由于此处的特殊情况，使得 or 指令的执行结果和相加是一样的。

与 xor 一样，or 也是逻辑运算指令。不同之处在于，or 执行的是逻辑“或”。数字逻辑中的“或”用于表示两个命题并列的情况。如果 0 代表假，1 代表真，那么：

```
0 or 0 = 0
```

```
0 or 1 = 1
1 or 0 = 1
1 or 1 = 1
```

在处理器内部，or 指令的目的操作数可以是 8 位或者 16 位的通用寄存器，或者包含 8/16 位实际操作数的内存地址，源操作数可以是与目的操作数数据宽度相同的通用寄存器、内存单元或者立即数。比如：

```
or al,cl
or ax,dx
or [label_a],bx
or byte [label_a],0x55
```

or 指令不允许目的操作数和源操作数都是内存单元的情况。当 or 指令执行时，两个操作数相对应的比特之间分别进行各自的逻辑“或”运算，结果位于目的操作数中。举个例子，以下指令执行后，寄存器 AL 中的内容是 0xff。

```
mov al,0x55
or al,0xaa
```

再来看源程序第 50 行，因为每次是除以 10，所以在寄存器 DL 中得到的余数，其高 4 位必定为 0。又由于 0x30 的低 4 位是 0，高 4 位是 3，所以，DL 中的内容和 0x30 执行逻辑“或”后，相当于是将 DL 中的内容和 0x30 相加。这是用逻辑“或”指令做加法的一个特例。

or 指令对标志寄存器的影响是：OF 和 CF 位被清零，SF、ZF、PF 位的状态依计算结果而定，AF 位的状态未定义。

与 or 相对应的是逻辑与“and”。如果 0 代表假，1 代表真，那么

```
0 and 0 = 0
0 and 1 = 0
1 and 0 = 0
1 and 1 = 1
```

相应地，处理器设计了 and 指令。在 16 位处理器上，and 指令的两个操作数都应当是字节或者字。其中，目的操作数可以是通用寄存器和内存单元；源操作数可以是通用寄存器、内存单元或者立即数，但不允许两个操作数同时为内存单元，而且它们在数据宽度上应当一致。比如：

```
and al,0x55
and ch,cl
and ax,dx
and [label_a],ah
and word [label_a],0xf0f0
and dx,[label_a]
```

注意，“label_a”是一个标号，下同。

当这些指令执行时，两个操作数对应的各个比特位分别进行逻辑“与”，结果保存在目的操作数中。因此，下面的这些指令执行后，寄存器 AX 中的结果是二进制数 1000000000000100，即 0x8004：

```
mov ax,1001_0111_0000_0100B
and ax,1000_0000_1111_0111B
```

and 指令执行后，OF 和 CF 位被清零，SF、ZF、PF 位的状态依计算结果而定，AF 位的状

态未定义。各个数位的 ASCII 码是压入堆栈中的。源程序第 51 行，push 指令的作用是将寄存器 DX 的内容压入堆栈中。在 16 位的处理器上，push 指令的操作数可以是 16 位的寄存器或者内存单元。例如：

```
push ax
push word [label_a]
```

你可能觉得奇怪，push 指令只接受 16 位的操作数，为什么要对内存操作数使用关键字“word”。事实上，8086 处理器只能压入一个字；但其后的处理器允许压入字、双字或者四字，因此，关键字是必不可少的。

就 8086 处理器来说，因为压入堆栈的内容必须是字，所以，下面的指令都是非法的：

```
push al
push byte [label_a]
```

处理器在执行 push 指令时，首先将堆栈指针寄存器 SP 的内容减去操作数的字长（以字节为单位的长度，在 16 位处理器上是 2），然后，把要压入堆栈的数据存放到逻辑地址 SS:SP 所指向的内存位置（和其他段的读写一样，把堆栈段寄存器 SS 的内容左移 4 位，加上堆栈指针寄存器 SP 提供的偏移地址）。

如图 7-3 所示，代码段和堆栈段是同一个段，所以段寄存器 CS 和 SS 的内容都是 0x0000。而且，堆栈指针寄存器 SP 的内容在源程序第 42 行被置为 0。所以，当 push 指令第一次执行时，SP 的内容减 2，即 $0x0000 - 0x0002 = 0xFFFF$ ，借位被忽略。于是，被压入堆栈的数据，在内存中的位置实际上是 0x0000:0xFFFF。push 指令的操作数是字，而且 Intel 处理器是使用低端字节序的，故低字节在低地址部分，高字节在高地址部分，正好占据了堆栈段的最高两个字节位置。

这只是第一次压栈操作时的情况。以后每次压栈时，SP 都要依次减 2。很明显，不同于代码段，代码段在处理器上执行时，是由低地址端向高地址端推进的，而压栈操作则正好相反，是从高地址端向低地址端推进的。

push 指令不影响任何标志位。

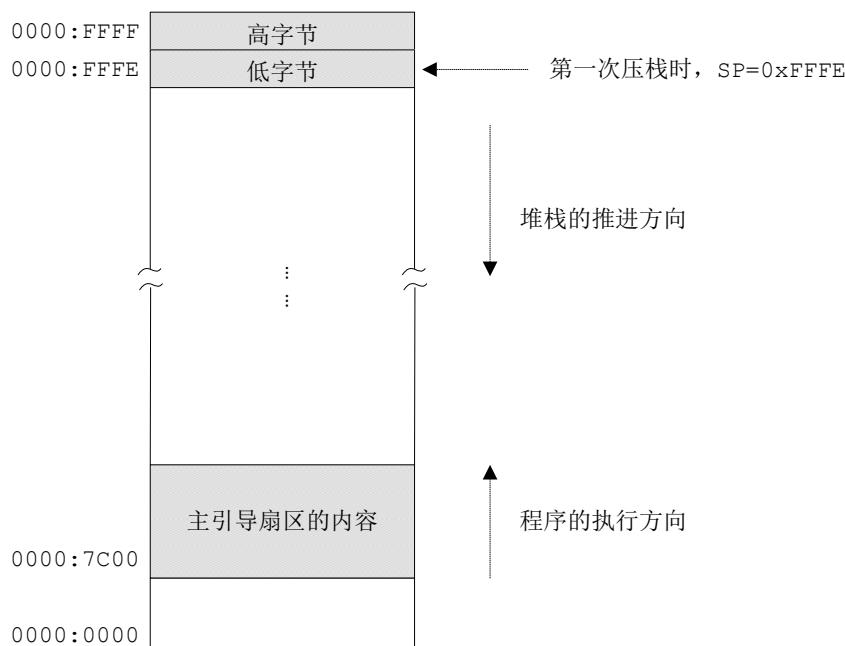


图 7-3 第一次执行压栈操作时的内存状态

源程序第 52、53 行，判断本次除法结束后，商是否为 0。如果不为零，则再循环一次；如果为零，则表明不需要再继续分解了。

7.5.3 出栈并显示各个数位

压栈的次数（数位的个数）取决于 AX 中的数有多大，位于寄存器 CX 中。数位是按“个位”、“十位”、“百位”、“千位”、“万位”的顺序依次压栈的（实际情况取决于数的大小），出栈正好相反。所以，可以顺序将它们弹出堆栈并显示在屏幕上。

源程序第 57 行，pop dx 指令的功能是将逻辑地址 SS:SP 处的一个字弹出到寄存器 DX 中，并将 SP 的内容加上操作数的字长（2）。

和 push 指令一样，pop 指令的操作数可以是 16 位的寄存器或者内存单元。例如：

```
pop ax
pop word [label_a]
```

pop 指令执行时，处理器将堆栈段寄存器 SS 的内容左移 4 位，再加上堆栈指针寄存器 SP 的内容，形成 20 位的物理地址访问内存，取得所需的数据。然后，将 SP 的内容加操作数的字长，以指向下一个堆栈位置。

pop 指令不影响任何标志位。

源程序第 58 行将弹出的数据写入显示缓冲区。索引寄存器 DI 的内容是在前面显示字符串时用过的，期间一直没有改变过，它现在指向显示缓冲区中字符串之后的位置。

接着，源程序第 59~61 行，将字符显示属性写入字符之后的单元，并再次递增 DI 以指向显示缓冲区中下一个字符的位置。

源程序第 62 行，每次执行 loop 指令时，处理器都是先将寄存器 CX 减一。当所有的数位都弹出和显示以后，CX 必定为零，这将导致退出循环。

当处理器最后一次执行出栈操作后，堆栈指针寄存器 SP 的内容将恢复到最开始设置时的状态，即它的内容重新为 0。

7.5.4 进一步认识堆栈

关于堆栈，这里有几点说明。

第一，push 指令的操作数可以是 16 位寄存器或者指向 16 位实际操作数的内存单元地址，push 指令执行后，压入堆栈中的仅仅是该寄存器或者内存单元里的数值，与该寄存器或内存单元不再相干。所以，下面的指令是合法而且正确的：

```
push cs
pop ds
```

这两条指令的意思是，将代码段寄存器的内容压栈，并弹出到数据段寄存器 DS。如此一来，代码段和数据段将属于同一个内存段。实际上，这两条指令的执行结果，和以下指令的执行结果相同：

```
mov ax,cx
mov ds,ax
```

第二，堆栈在本质上也只是普通的内存区域，之所以要用 push 和 pop 指令来访问，是因为你

把它看成堆栈而已。实际上，如果你把它看成是普通的数据段而忘掉它是一个堆栈，那么它将不再神秘。

引入堆栈和 push、pop 指令只是为了方便程序开发。临时保存一个数值到堆栈中，使用 push 指令是最简洁、最省事的，但如果你不怕麻烦，可以不使用它。所以，下面的代码可以用来取代 push ax 指令：

```
sub sp,2
mov bx,sp
mov [ss:bx],ax
```

同样，pop ax 指令的执行结果和下面的代码相同：

```
mov bx,sp
mov ax,[ss:bx]
add sp,2
```

但是显而易见，push 和 pop 指令更方便，毕竟与堆栈访问有关的一切都是由处理器自动维护的。

第三，要注意保持堆栈平衡，防止数据访问越界。尤其是在编写程序前，必须充分估计所需要的堆栈空间，以防止破坏有用的数据。特别是在堆栈段和其他段属于同一个段的时候。如图 7-3 所示，堆栈段和代码段属于同一个内存段，段地址都是 0x0000，段的长度都是 64KB。主引导程序的长度是 512(0x200)字节，从偏移地址 0x7c00 延伸到 0x7e00。堆栈是向下增长的，它们之间有 $0xffff - 0x7e00 + 1 = 0x8200$ 字节的空档。通常来说，我们的程序是安全的，因为不可能压入这么多的数据。

但是，不能掉以轻心，堆栈定义得过小，而且程序编写不当，导致堆栈破坏了有用数据的情况也时有发生。尽管不能完全阻止程序中的错误，但是，通过将堆栈定义到一个单独的 64KB 段，可能会好一点。这样，无论任何时候，即使是 push 指令位于一个无限循环中，堆栈指针寄存器 SP 的内容也永远只会在 0x0000~0xFFFF 之间来回滚动，不会影响到其他内存段。

7.6 程序的编译和运行

编译源程序 7-1，然后将生成的二进制文件 c07_mbr.bin 写入虚拟硬盘的主引导扇区，启动虚拟机观察程序运行结果。如果程序无误，结果应当如图 7-4 所示。

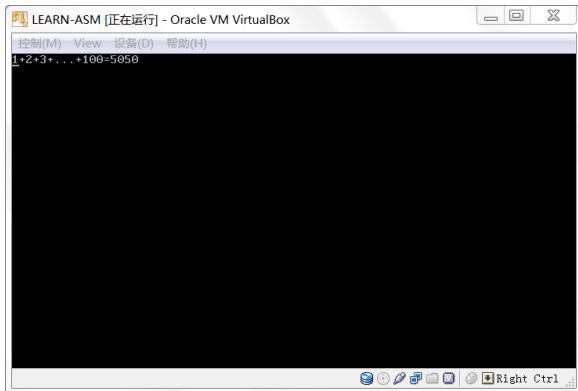


图 7-4 本章程序在虚拟机中的运行结果

7.7 8086 处理器的寻址方式

处理器的一生，是忙碌的一生，只要它工作着，就必定是在取指令和执行指令。它就像勤劳的牛，吃的是电，挤出来的还是电，不过是另一种形式的电。

多数指令操作的是数值。比如：

```
mov ax, 0x55aa
```

这条指令执行时，把 0x55aa 传送到寄存器 AX。再如：

```
add dx, cx
```

这是把寄存器 DX 中的数据和寄存器 CX 中的数据相加，并把结果保留在 DX 中，同时保持 CX 中原有的内容不变。

所以，如果你问处理器整天忙什么，它一定会说：“还能有什么，就是和数打交道！”

既然操作和处理的是数值，那么，必定涉及数值从哪里来，处理后送到哪里去，这称为寻址方式（Addressing Mode）。简单地说，寻址方式就是如何找到要操作的数据，以及如何找到存放操作结果的地方。

实际上，大多数的寻址方式我们都已经使用过，现在所做的只是一个完整的总结。当然，这里的讲解仅限于 16 位的处理器。

7.7.1 寄存器寻址

最简单的寻址方式是寄存器寻址。就是说，指令执行时，操作的数位于寄存器中，可以从寄存器里取得。这种寻址方式的例子还是很多的，比如：

```
mov ax, cx
```

```
add bx, 0xf000
inc dx
```

以上，第一条指令的两个操作数都是寄存器，是典型的寄存器寻址；第二条指令的目的操作数是寄存器，因此，该操作数也是寄存器寻址；第三条指令就更不用说了。

7.7.2 立即寻址

立即寻址又叫立即数寻址。也就是说，指令的操作数是一个立即数。比如：

```
add bx, 0xf000
mov dx, label_a
```

以上，第一条指令的目的操作数采用了寄存器寻址方式，用于提供被加数；第二个操作数（源操作数）用于给出加数 0xf000。这是一个直接给出的数值，是立即在指令中给出的，最终参与加法运算的就是它，不需要通过其他方式寻找，故称为立即数。这也是一种寻址方式，称为立即寻址。

在第二条指令中，目的操作数也采用的是寄存器寻址方式。尽管源操作数是一个标号，但是，标号是数值的等价形式，代表了它所在位置的汇编地址。因此，在编译阶段，它会被转化为一个立即数。因此，该指令的源操作数也采用了立即寻址方式。

7.7.3 内存寻址

寄存器寻址的操作数位于寄存器中，立即寻址的操作数位于指令中，是指令的一部分。传统上，这是两种速度较快的寻址方式。但是，它们也有局限性。一方面，我们不可能总是知道要操作的数是多少，因此也就不可能总是在指令中使用立即数；另一方面，寄存器的数量有限，不可能总指望在寄存器之间来回倒腾。

考虑到内存容量巨大，所以，在指令中使用内存地址，来操作内存中的数据，是最理想不过了。正是因为内存访问如此重要，处理器才拥有好几种内存寻址方式。

我们知道，8086 处理器访问内存时，采用的是段地址左移 4 位，然后加上偏移地址，来形成 20 位物理地址的模式，段地址由 4 个段寄存器之一来提供，偏移地址要由指令来提供。

因此，所谓的内存寻址，实际上就是寻找偏移地址，这称为有效地址（Effective Address，EA）。换句话说，就是如何在指令中提供偏移地址，供处理器访问内存时使用。

1. 直接寻址

使用该寻址方式的操作数是一个偏移地址，而且给出了该偏移地址的具体数值。比如：

```
mov ax, [0x5c0f]
add word [0x0230], 0x5000
xor byte [es:label_b], 0x05
```

但凡是表示内存地址的，都必须用中括号括起来。

以上，在第一条指令中，源操作数使用的是直接寻址方式，当这条指令执行时，处理器将数据段寄存器 DS 的内容左移 4 位，加上这里的 0x5c0f，形成 20 位物理地址。接着，从该物理地址处取得一个字，传送到寄存器 AX 中。

在第二条指令中，目的操作数采用的是直接寻址方式。当这条指令执行时，处理器用同样的方法，访问由段寄存器 DS 指向的数据段，并把指令中的立即数加到该段中偏移地址为 0x0230 的字单

元里。

尽管在第三条指令中，目的操作数使用了标号和段超越前缀，但它依然属于直接寻址方式。原因很简单，标号是数值的等价形式，在指令编译阶段，会被转换成数值；而段超越前缀仅仅用来改变默认的数据段。

2. 基址寻址

很多时候，我们会有一大堆的数据要处理，而且它们通常都是挨在一起，顺序存放的。比如：

```
buffer dw 0x20,0x100,0x0f,0x300,0xff00
```

假如要将这些数据统统加一，那么，使用直接寻址的指令序列肯定是这样的：

```
inc word [buffer]
inc word [buffer+2]
inc word [buffer+4]
...

```

这样做好吗？当然，程序本身是没有问题的。但是，考虑到它的效率和代码的简洁性，特别是这些工作用循环来完成会更好，可以使用基址寻址。所谓基址寻址，就是在指令的地址部分使用基址寄存器 BX 或者 BP 来提供偏移地址。比如：

```
mov [bx],dx
add byte [bx],0x55
```

以上，第一条指令中的目的操作数采用了基址寻址。在指令执行时，处理器将数据段寄存器 DS 的内容左移 4 位，加上基址寄存器 BX 中的内容，形成 20 位的物理地址。然后，把寄存器 DX 中的内容传送到该地址处的字单元里。

第二条指令中的目的操作数也采用的是基址寻址。指令执行时，将数据段寄存器 DS 的内容左移 4 位，加上寄存器 BX 中的内容，形成 20 位的物理地址。然后，将指令中的立即数 0x55 加到该地址处的字节单元里。

使用基址寻址可以使代码变得简洁高效。比如，可以用以下的代码来处理上面的批量加一任务：

```
mov bx,buffer
mov cx,4
lpinc:
inc word [bx]
add bx,2
loop lpinc
```

基址寻址的寄存器也可以是 BP。比如：

```
mov ax,[bp]
```

这条指令的源操作数采用了基址寻址方式。但是，与前面的指令相比，它稍微有些特殊。原因在于，它采用的是基址寄存器 BP，在形成 20 位的物理地址时，默认的段寄存器是 SS。也就是说，它经常用于访问堆栈。这条指令执行时，处理器将堆栈段寄存器 SS 的内容左移 4 位，加上寄存器 BP 的内容，形成 20 位的物理地址，并将该地址处的一个字传送到寄存器 AX 中。

我们知道，堆栈是后进先出的数据结构，访问堆栈的一般方法是使用 push 和 pop 指令。比如我们用以下的指令压入两个数据：

```
mov ax,0x5000
push ax
```

```
mov ax, 0x7000
push ax
```

很显然，如果要用 pop 指令弹出数据，就必须先弹出 0x7000，才能弹出 0x5000，除非你改变了堆栈指针 SP 的内容，否则这个顺序是不可能改变的。

但是，有时候我们希望，而且必须得越过这种限制，去访问栈中的内容，还不能破坏堆栈的状态，特别是堆栈指针寄存器 SP 的内容，使得 push 和 pop 操作能正常进行。一个典型的例子是高级语言里的函数调用，所有的参数都位于堆栈中。为了能访问到那些被压在栈底的参数，这时，BP 就能派上用场：

```
mov ax, 0x5000
push ax
mov bp, sp
mov ax, 0x7000
push ax
mov dx, [bp]           ;dx 中的内容为 0x5000
```

以上，在压入 0x5000 之后，立即将堆栈指针 SP 保存到 BP。后面，尽管栈顶的数据 0x7000 没有出栈，但依然可以用 BP 取出压在堆栈下面的 0x5000。如此一来，正常的 push 和 pop 操作照样进行，同时，还能访问到栈中的参数。

基址寻址允许在基址寄存器的基础上使用一个偏移量。有时候，这使得它更加灵活。比如：

```
mov dx, [bp-2]
```

处理器在执行时，将段寄存器 SS 的内容左移 4 位，加上 BP 的内容，再减去偏移量 2 以形成物理地址。这样一来，在保持基址寄存器 BP 内容不变的情况下，就可以访问栈中的任何元素。这里，偏移量仅用于在指令执行时形成有效地址，不会改变寄存器 BP 的原有内容。

这种增加偏移量的做法也适用于基址寄存器 BX。以下代码是前面那个批量加一任务的新版本：

```
xor bx, bx
mov cx, 4
lpinc:
    inc word [bx+buffer]
    add bx, 2
loop lpinc
```

以上代码和前一个版本相比，没有太大变化，区别仅仅在于，BX 现在是从 0 开始递增的，inc 指令操作数的偏移地址由 BX 和标号 buffer 所代表的值相加得到。相加操作在指令执行时进行，仅用于形成有效偏移地址，不会影响到 BX 寄存器的内容。

3. 变址寻址

变址寻址类似于基址寻址，唯一不同之处在于这种寻址方式使用的是变址寄存器（或称索引寄存器）SI 和 DI。例如：

```
mov [si], dx
add ax, [di]
xor word [si], 0x8000
```

和基址寻址一样，当带有这种操作数的指令执行时，除非使用了段超越前缀，处理器会访问由

段寄存器 DS 指向的数据段，偏移地址由寄存器 SI 或者 DI 提供。

同样地，变址寻址方式也允许带一个偏移量：

```
mov [si+0x100], al
and byte [di+label_a], 0x80
```

以上第二条指令中，尽管使用的是标号，但本质上属于一个编译阶段确定的数值。

4. 基址变址寻址

让处理器支持多种寻址方式会增加硬件上的复杂性，但可以增强它的数据处理能力，这么做是值得的。说到数据处理，下面是一个稍微复杂一些的任务：

```
string db 'abcdefghijklmnopqrstuvwxyz'
```

以上声明了标号“string”并初始化了 26 个字节的数据。现在，你的任务是，将这 26 字节的数据在原地反向排列。

这个问题不难，所以你可能很快想到使用堆栈，先将这 26 个数据压栈，再反向出栈，因为堆栈是后进先出的，正好符合要求。代码是这样的（代码段、堆栈段初始化的代码统统省略）：

```
mov cx, 26 ; 循环次数，从 26 到 1，共 26 次
mov bx, string ; 数据区首地址（基址）
lppush:
    mov al, [bx]
    push ax
    inc bx
    loop lppush ; 循环压栈

    mov cx, 26
    mov bx, string

lppop:
    pop ax
    mov [bx], al
    inc bx
    loop lppop ; 循环出栈
```

这的确是个好办法。不过，8086 处理器也支持一种基址加变址的寻址方式，简称基址变址寻址，可能用起来更方便。

使用基址变址的操作数可以使用一个基址寄存器（BX 或者 BP），外加一个变址寄存器（SI 或者 DI）。它的基本形式是这样的：

```
mov ax, [bx+si]
add word [bx+di], 0x3000
```

以上，第一条指令的源操作数采用了基址变址寻址。当处理器执行这条指令时，把数据段寄存器 DS 的内容左移 4 位，加上基址寄存器 BX 的内容，再加上变址寄存器 SI 的内容，共同形成 20 位的物理地址。然后，从该地址处取得一个字，传送到寄存器 AX 中。

第二条指令与第一条指令类似，只不过是加法指令，它的目的操作数采用了基址变址寻址，源操作数采用的是立即寻址。这条指令执行时，处理器访问由段寄存器 DS 指向的数据段，加上由 BX 和 DI 相加形成的偏移地址，共同形成 20 位的物理地址，然后将立即数 0x3000 加到该地址处的字

单元里。

采用基址变址寻址方式的排序代码如下：

```

mov bx, string           ; 数据区首地址
mov si, 0                ; 正向索引
mov di, 25               ; 反向索引

order:
    mov ah, [bx+si]
    mov al, [bx+di]
    mov [bx+si], al
    mov [bx+di], ah        ; 以上 4 行用于交换首尾数据
    inc si
    dec di
    cmp si, di
    jl order               ; 首尾没有相遇，或者没有超越，继续

```

和前面使用堆栈的代码相比，指令的数量没有明显减少，这说明任务还不够复杂，也许只能这么解释了。但是，它同样很方便，很有效，不是吗？

同样地，基址变址寻址允许在基址寄存器和变址寄存器的基础上带一个偏移量。比如：

```

mov [bx+si+0x100], al
and byte [bx+di+label_a], 0x80

```

本 章 习 题

1. 修改代码清单 7-1 的第 31~37 行，使用 loop 指令来计算累加和。要求：CX 寄存器既用来控制循环次数，同时还用来作为被累加的数。

2. 在 16 位的处理器上，做加法的指令是 add，但它每次只能做 8 位或 16 位的加法。除此之外，还有一个带进位加法指令 adc (Add With Carry)，它的指令格式和 add 一样，目的操作数可以是 8 位或 16 位的通用寄存器和内存单元，源操作数可以是与目的操作数宽度一致的通用寄存器、内存单元和立即数（但目的操作数和源操作数同为内存单元的除外）。不过，adc 指令在执行的时候，除了将目的操作数和源操作数相加，还要加上当前标志寄存器的 CF 位。也就是说，视 CF 位的状态，还要再加 0 或者加 1。这样一来，用 adc 指令配合 add 指令，就可以计算 16 位以上的加法。

adc 指令对 OF、SF、ZF、AF、CF 和 PF 的影响视计算结果而定。

现在，请编写一段主引导扇区程序，计算 1 到 1000 的累加和，并在屏幕上显示结果。

第8章 硬盘和显卡的访问与控制



8.1 本章代码清单

8.1.1 本章意图

我很想一口吃成个胖子，但这很不现实，汇编语言和处理器知识的学习也不例外。

一直以来，我在每一章都会介绍一些新的处理器指令，以及一些处理器工作方式的内容，比如分段、堆栈操作、循环等，本章当然也不例外。不得不说的是，因为涉及的东西较多，本章的学习任务尤其繁重。

总是把目光放在一个小小的主引导扇区上，这没什么意思。现在，是我们离开它，向自由天地迈进的时候了。但是，应该迈向哪里呢？

主引导扇区是处理器迈向广阔天地的第一块跳板。离开主引导扇区之后，前方通常就是操作系统的森林，也就是我们经常听说的 DOS、Windows、Linux、UNIX 等。

操作系统也是由一大堆指令组成的，之所以将其比作“森林”，是因为它包含了更多的指令，也许是几万条、几十万条，甚至几千万条的指令。相比之下，我们在前面编写的那些指令代码则相形见绌了。

和主引导扇区程序一样，操作系统也位于硬盘上。操作系统是需要安装到硬盘上的，这个安装过程不但要把操作系统的指令和数据写入硬盘，通常还要更新主引导扇区的内容，好让这块跳板直接连着操作系统。不像我们，一直用主引导扇区来显示字符和做加法。

在个人计算机中，最流行的操作系统无疑是 Microsoft Windows，它简单易用，功能强大，最主要的，还是一个图形化的软件。相比之下，年轻一代的人们很少知道 MS-DOS，它是 Windows 出现之前，最流行的个人计算机操作系统。不过，它不是图形界面的，而是 80×25 的字符显示模式。即使是这样，它也曾经是个人计算机最主要、最基本的软件配置，统治了个人计算机 15 年，直到 Windows 95 发布。

操作系统通常肩负着处理器管理、内存分配、程序加载、进程（即已经位于内存中的程序）调度、外围设备（显卡、硬盘、声卡等）的控制和管理等任务。举个例子来说，你每天都要使用的 Windows，它可以让你看到计算机内都有几块硬盘，都安装了哪些程序（通过图标来显示），并允许你双击图标运行这些程序，这都是托了操作系统（Windows）的福。要不然的话，这都是不可能的事。

凭个人之力，写一个非常完善的操作系统，这几乎是不可能的事。但是，写个小程序，模拟一

下它的某个功能，还是可以的。我们知道，编译好的程序通常都存放在像硬盘这样的载体上，需要加载到内存之后才能执行。这个过程并不简单，首先要读取硬盘，然后决定把它加载到内存的什么位置。最重要的是，程序通常是分段的，载入内存之后，还要重新计算段地址，这叫做段的重定位。

程序可以有千千万万个，但加载过程却是固定的。在本章，我们把主引导扇区改造成一个程序加载器，或者说是一个加载程序，它的功能是加载用户程序，并执行该程序（将处理器的控制权交给该程序）。

说到这里，我相信你一定很好奇，早就想知道 BIOS 是怎么把主引导扇区读到内存中的，又是怎么让它从 0x0000:0x7c00 处开始执行的。好吧，本章就可以满足你的好奇心。同时我相信，通过在本章里做这件事，可以加深你对处理器、操作系统，以及软件开发过程（包含用高级语言进行软件编写）的理解。

8.1.2 代码清单 8-1

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：8-1（主引导扇区程序/加载器），源程序文件：c08_mbr.asm

本章代码清单：8-2（被加载的用户程序），源程序文件：c08.asm

8.2 用户程序的结构

8.2.1 分段、段的汇编地址和段内汇编地址

处理器的工作模式是将内存分成逻辑上的段，指令的获取和数据的访问一律按“段地址：偏移地址”的方式进行。相对应地，一个规范的程序，应当包括代码段、数据段、附加段和堆栈段。这样一来，段的划分和段与段之间的界限在程序加载到内存之前就已经准备好了。

和我们以前面编写的源程序不同，代码清单 8-2 很长。当然，真正的不同之处在于，代码和数据是以段的形式组织的。当然，因为清单很长，看起来并不是非常明显。为了清楚起见，图 8-1 给出了整个源程序的组织结构。

NASM 编译器使用汇编指令“SECTION”或者“SEGMENT”来定义段。它的一般格式是

SECTION 段名称

或者

SEGMENT 段名称

每个段都要求给出名字，这就是段名称，它主要用来引用一个段，可以是任意名字，只要它们彼此之间不会重复和混淆。

NASM 编译器不关心段的用途，可能也根本不知道段的用途，不知道它是数据段，还是代码段，或是堆栈段。事实上，这都不重要，段只用来分隔程序中的不同内容。

不过，话又说回来了，作为程序员，每个段的用途，你自己是清楚的。所以，为每个段起一个直观好记的名字，那是应该的。如图 8-1 所示，第一个段的名字是“header”，表明它是整个程序的开头部分；第二个段的名字是“code”，表明这是代码段；第三个段的名字是“data”，表明这是数

据段。

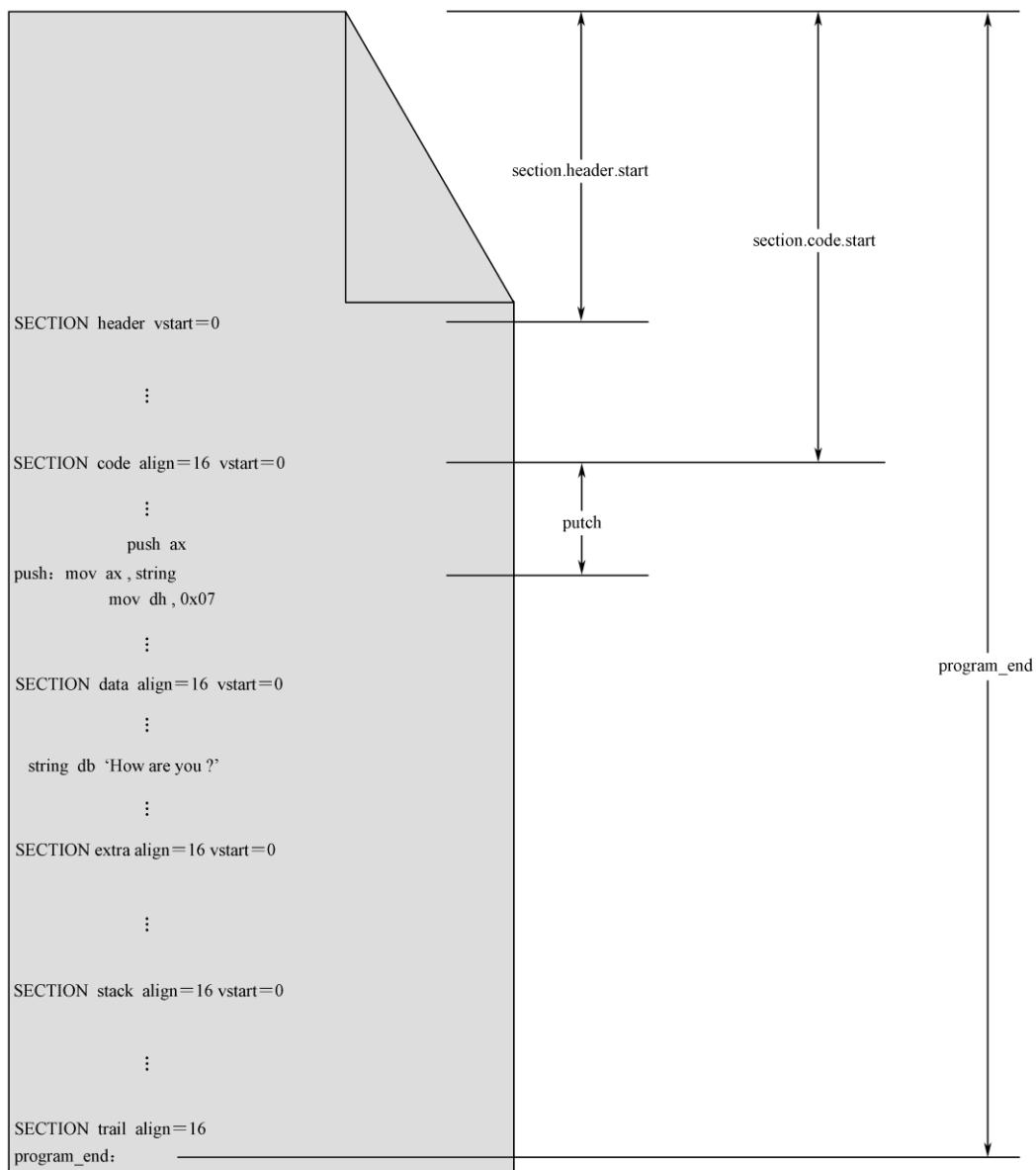


图 8-1 用户程序的一般结构

比较重要的是，一旦定义段，那么，后面的内容就都属于该段，除非又出现了另一个段的定义。另外，如图 8-2 所示，有时候，程序并不以段定义语句开始。在这种情况下，这些内容默认地自成一个段。最为典型的情况是，整个程序中都没有段定义语句。这时，整个程序自成一个段。

NASM 对段的数量没有限制。一些大的程序，可能拥有不止一个代码段和数据段。

Intel 处理器要求段在内存中的起始物理地址起码是 16 字节对齐的。这句话的意思是，必须是 16 的倍数，或者说该物理地址必须能被 16 整除。

相应地，汇编语言源程序中定义的各个段，也有对齐方面的要求。具体做法是，在段定义中使用“align=”子句，用于指定某个 SECTION 的汇编地址对齐方式。比如说，“align=16”就表示段是 16 字节对齐的，“align=32”就表示段是 32 字节对齐的。



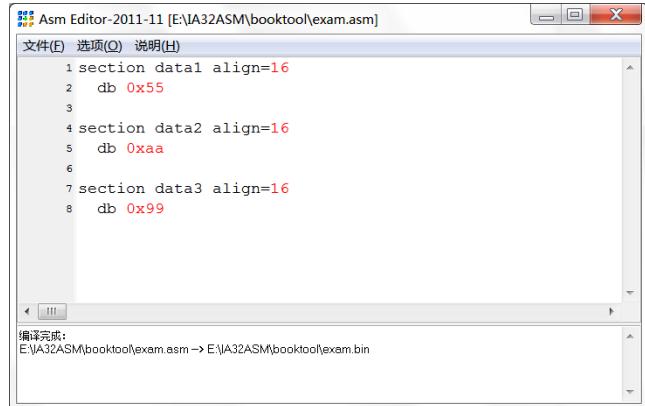
```

1      mov ax,cs
2      mov ds,ax
3
4      jmp $ 
5
6
7 section data
8 ;后面的内容都属于data段
9 num1 db 0x55
10 str1 db 'hello,world!'
11 str2 db 'This is chapter 8.'
12

```

图 8-2 程序并非以段定义开始的情况

在源程序编译阶段，编译器将根据 align 子句确定段的起始汇编地址。如图 8-3 所示，这里定义了三个段，分别是 data1、data2 和 data3，每个段里只有一个字节的数据，分别是 0x55、0xaa 和 0x99。



```

1 section data1 align=16
2 db 0x55
3
4 section data2 align=16
5 db 0xaa
6
7 section data3 align=16
8 db 0x99

```

编译完成:
E:\A32ASM\booktool\exam.asm -> E:\A32ASM\booktool\exam.bin

图 8-3 align 子句对段的影响（编译之前的源代码）

理论上，如果不考虑段的对齐方式，那么段 data1 的汇编地址是 0，段 data2 的汇编地址是 1，段 data3 的汇编地址是 2。

但是，在这里，每个段的定义中都包含了要求 16 字节对齐的子句，情况便不同了。如图 8-4 所示，这是编译后的结果，因为在段 data1 之前没有任何内容，故段 data1 的起始汇编地址是 0（在图中是 0x00000000），而且地址 0 本身就是 16 字节对齐的，符合 align 子句的要求。

段的汇编地址其实就是段内第一个元素（数据、指令）的汇编地址。因此，在段 data1 中声明和初始化的 0x55 位于汇编地址 0x00000000 处。

段 data2 也要求是 16 字节对齐的。问题是，从汇编地址 0x00000001 开始，只有 0x00000010（十进制的 16）才能被 16 整除。于是，编译器将 0x00000010 作为段 data2 的汇编地址，并在两个段之间填充 15 字节的 0x00（段 data1 只有 1 字节的长度）。

段 data3 的处理与前面两个段相同。因为段 data2 只有 1 字节，故也需要在它们之间填充 15 字节。这样，段 data3 的汇编地址就是 0x00000020（十进制的 32）。段 data3 也只有 1 字节（0x99），所以，汇编地址 0x00000020 处是 0x99，这也是编译结果中的最后一字节。

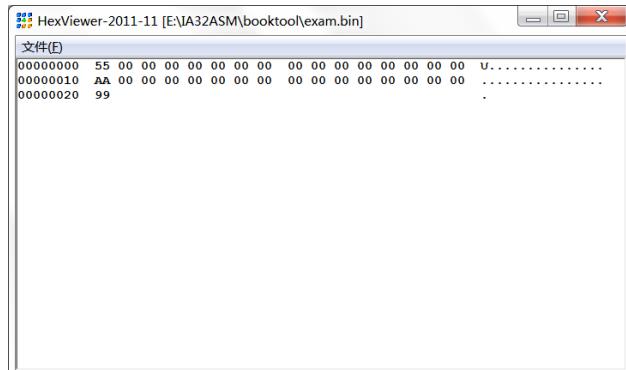


图 8-4 align 子句对段的影响（编译之后的二进制文件）

正如我们刚刚讨论过的，每个段都有一个汇编地址，它是相对于整个程序开头（0）的。为了方便取得该段的汇编地址，NASM 编译器提供了以下的表达式，可以用在你的程序中：

section.段名称.start

如图 8-1 所示，段 “header” 相对于整个程序开头的汇编地址是 section.header.start，段 “code” 相对于整个程序开头的汇编地址是 section.code.start。在这个例子中，因为段 “header” 是在程序的开始定义的，它的前面没有其他内容，故 section.header.start=0。

如图 8-1 所示，段定义语句还可以包含 “vstart=” 子句。尽管定义了段，但是，引用某个标号时，该标号处的汇编地址依然从整个程序的开头计算的，而不是从段的开头处计算的。

这就很麻烦（有时候也很有用）。因此，`vstart` 可以解决这个问题。如图 8-1 所示，“`putch`”是段 `code` 中的一个标号，原则上，该标号代表的汇编地址应该从程序开头计算。但是，因为段 `code` 的定义中有“`vstart=0`”子句，所以，标号“`putch`”的汇编地址要从它所在段的开头计算，而且从 0 开始计算。

如图 8-1 所示，同样的情形也出现在段 data 中。段 data 的定义中也有“vstart=0”子句，因此，当我们在段 code 中引用段 data 中的标号“string”时（`mov ax,string`），尽管在图中没有标明，标号“string”所代表的汇编地址是相对于其所在段 data 的。也就是说，传送到寄存器 AX 中的数值是标号 string 相对于段 data 起始处的长度。

但是，图中最后一个段 `trail` 的定义中没有包含 “`vstart=0`” 子句。那就对不起了，该段内有一个标号 “`program_end`”，它的汇编地址就要从整个程序开头计算。因为它是整个程序中的最后一行，从这个意义上来说，它所代表的汇编地址就是整个程序的大小（以字节计）。

8.2.2 用户程序头部

在上面，我们已经知道如何在用户程序中分段，也知道各种段子句对段的起始汇编地址和段内汇编地址的影响。现在，让我们结合本章中的实例来进一步加深认识。

浏览一下本章代码清单 8-2，你会发现，本章的用户程序实际上定义了 7 个段，分别是第 7 行定义的段 header、第 27 行定义的段 code_1、第 163 行定义的段 code_2、第 173 行定义的段 data_1、第 194 行定义的段 data_2、第 201 行定义的段 stack 和第 208 行定义的段 trail。

一般来说，加载器和用户程序是在不同的时间、不同的地方，由不同的人或公司开发的。这就意味着，它们彼此并不了解对方的结构和功能。事实上，也不需要了解。

如图 8-5 所示，它们彼此看对方都是一个黑盒子，并不了解对方是怎么编写的，是做什么的。但是，也不能完全是黑的，加载器必须了解一些必要的信息，虽然不是很多，但足以知道如何加载

用户程序。

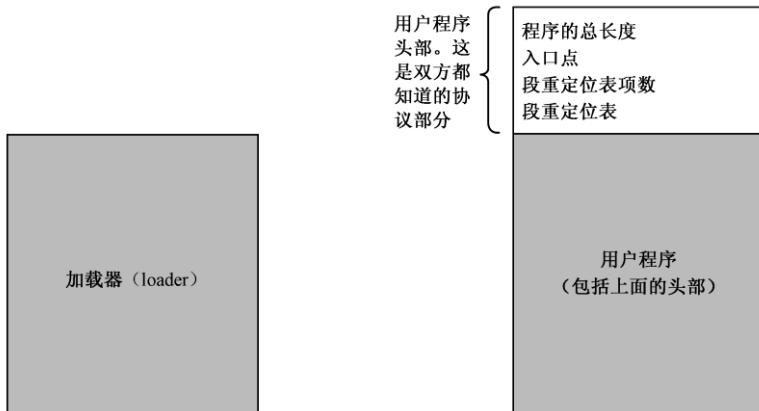


图 8-5 加载器与用户程序之间的协议部分示意图

这就涉及加载器的编写者，以及用户程序的编写者，他们之间是怎么协商的。他们之间必须有一个协议，或者说协定，比如说，在用户程序内部的某个固定位置，包含一些基本的结构信息，每个用户程序都必须把自己的情况放在这里，而加载器也固定在这个位置读取。经验表明，把这个约定的地点放在用户程序的开头，对双方，特别是对加载器来说比较方便，这就是用户程序头部。

头部需要在源程序以一个段的形式出现。这就是代码清单 8-2 的第 7 行：

```
SECTION header vstart=0
```

而且，因为它是“头部”，所以，该段当然必须是第一个被定义的段，且总是位于整个源程序的开头。

用户程序头部起码要包含以下信息。

① 用户程序的尺寸，即以字节为单位的大小。这对加载器来说是很重要的，加载器需要根据这一信息来决定读取多少个逻辑扇区（在本书中，所有程序在硬盘上所占用的逻辑扇区都是连续的）。

代码清单 8-2 中第 8 行，伪指令 dd 用于声明和初始化一个双字，即一个 32 位的数据。用户程序可能很大，16 位的长度不足以表示 65535 以上的数值。

程序的长度取自程序中的一个标号“program_end”，这是允许的。在编译阶段，编译器将该标号所代表的汇编地址填写在这里。该标号位于整个源程序的最后，从属于段“trail”。由于该段并没有 vstart 子句，所以，标号“program_end”所代表的汇编地址是从整个程序的开头计算的。换句话说，program_end 所代表的汇编地址，在数值上等于整个程序的长度。

双字在内存中的存放也是按低端序的。如图 8-6 所示，低字保存在低地址，高字保存在高地址。同时，每个字又按低端字节序，低字节在低地址，高字节在高地址。

② 应用程序的入口点，包括段地址和偏移地址。加载器并不清楚用户程序的分段情况，更不知道第一条要执行的指令在用户程序中的位置。因此，必须在头部给出第一条指令的段地址和偏移地址，这就是所谓的应用程序入口点（Entry Point）。

理想情况下，当用户程序开始运行时，执行的第一条指令是其代码段内的第一条指令。换句话说，入口点位于其代码段内偏移地址为 0 的地方。但是，情况并非总是如此。尤其是，很多程序并非只有一个代码段，比如本章源代码清单 8-2 就包含了两个代码段。所以，需要在用户程序头部明确给出用户程序在刚开始运行时，第一条指令的位置，也就是第一条指令在用户程序代码段内的偏移地址。

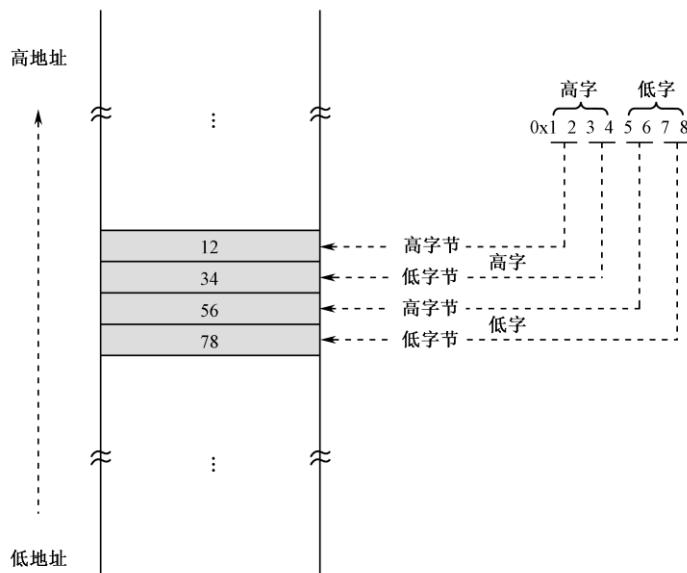


图 8-6 双字数据在内存中的布局

代码清单 8-2 第 11、12 行，依次声明并初始化了入口点的偏移地址和段地址。偏移地址取自代码段 code_1 中的标号“start”，段地址是用表达式 section.code_1.start 得到的。

代码段 code_1 是在代码清单 8-2 的第 27 行定义的：

```
SECTION code_1 align=16 vstart=0
```

显而易见的是，因为段定义中包含了“vstart=0”子句，故标号 start 所代表的汇编地址是相对于当前代码段 code_1 的起始位置，从 0 开始计算的。

入口点的段地址是用伪指令 dd 声明的，并初始化为汇编地址 section.code_1.start，这是一个 32 位的地址。不过，它仅仅是编译阶段确定的汇编地址，在用户程序加载到内存后，需要根据加载的实际位置重新计算（浮动）。

尽管在 16 位的环境中，一个段最长为 64KB，但它却可以起始于任何 20 位的物理地址处。你不可能用 16 位的单元保存 20 位的地址，所以，只能保存为 32 位的形式。

③ 段重定位表。用户程序可能包含不止一个段，比较大的程序可能会包含多个代码段和多个数据段。这些段如何使用，是用户程序自己的事，但前提是程序加载到内存后，每个段的地址必须重新确定一下。

段的重定位是加载器的工作，它需要知道每个段在用户程序内的位置，即它们分别位于用户程序内的多少字节处。为此，需要在用户程序头部建立一张段重定位表。

用户程序可以定义的段在数量上是不确定的，因此，段重定位表的大小，或者说表项数是不确定的。为此，代码清单 8-2 第 14 行，声明并初始化了段重定位表的项目数。因为段重定位表位于两个标号 header_end 和 code_1_segment 之间，而且每个表项占用 4 字节，故实际的表项数为

```
(header_end - code_1_segment) / 4
```

这个值是在程序编译阶段计算的，先用两个标号所代表的汇编地址相减，再除以每个表项的长度 4。

紧接着表项数的，是实际的段重定位表，每个表项用伪指令 dd 声明并初始化为 1 个双字。代码清单 8-2 一共定义了 5 个段，所以这里有 5 个表项，依次计算段开始汇编地址的表达式并进行初始化。

8.3 加载程序（器）的工作流程

8.3.1 初始化和决定加载位置

从大的方面来说，加载器要加载一个用户程序，并使之开始执行，需要决定两件事。第一，看看内存中的什么地方是空闲的，即从哪个物理内存地址开始加载用户程序；第二，用户程序位于硬盘上的什么位置，它的起始逻辑扇区号是多少。如果你连它在哪里都不知道，怎么找得到它呢！

现在，让我们把目光转移到代码清单 8-1，来看看加载器都做了哪些工作。

代码清单 8-1 第 6 行，加载器程序的一开始声明了一个常数（const）：

```
app_lba_start equ 100
```

常数是用伪指令 equ 声明的，它的意思是“等于”。本语句的意思是，用标号 app_lba_start 来代表数值 100，今后，当我们要用到 100 的时候，不这样写：

```
mov al,100
```

而是这样写：

```
mov al,app_lba_start
```

你可能会说，这样不是更麻烦吗？

不会的，实际上这很方便。用某些教材上的话说，程序中不该使用“不可思议的数”。想想看，如果在程序中的多个地方直接使用数值 100，那么，以后要修改它们，把它们改成 500，还得找到所有使用这个数值的位置，一一修改，万一漏掉一个呢？如果使用常量 app_lba_start，则只需要重新把这个常数的声明语句改成下面的形式，并重新编译即可。

```
app_lba_start equ 500
```

常数的意思是在程序运行期间不变的数。和其他伪指令 db、dw、dd 不同，用 equ 声明的数值不占用任何汇编地址，也不在运行时占用任何内存位置。它仅仅代表一个数值，就这么简单。

加载用户程序需要确定一个内存物理地址，这是在代码清单 8-1 第 151 行用伪指令 dd 声明的，并初始化为 0x10000 的。和前面一样，是用 32 位的单元来容纳一个 20 位的地址：

```
phy_base dd 0x10000
```

尽管我们用了一个好看的数 0x10000，但你完全可以把用户程序加载到其他地方，只要它是空闲的。比如，可以将这个数值改成 0x12340，唯一的要求是该地址的最低 4 位必须是 0，换句话说，加载的起始地址必须是 16 字节对齐的，这样将来才能形成一个

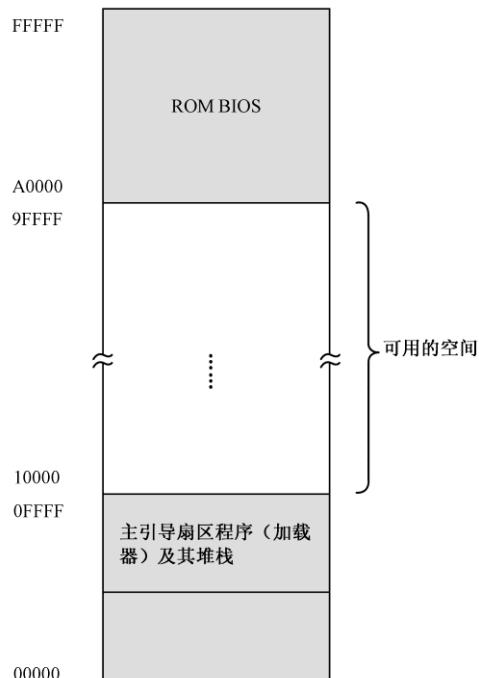


图 8-7 可用于加载用户程序的空间范围

有效的段地址。

如图 8-7 所示，物理地址 0xFFFF 以下，是加载器及其堆栈的势力范围；物理地址 A0000 以上，是 BIOS 和外围设备的势力范围，有很多传统的老式设备将自己的存储器和只读存储器映射到这个空间。

如此一来，可用的空间就位于 0x10000~9FFFF，差不多 500 多 KB。事实上，如果将低端的内存空间合理安排一下，还可以腾出更多空间，但是没有必要，我们用不了多少。

8.3.2 准备加载用户程序

和以往不同，我们将主引导扇区程序定义成一个段。代码清单 8-1 第 9 行：

```
SECTION mbr align=16 vstart=0x7c00
```

整个程序只定义了这一个段，所以它略显多余。之所以这么说，是因为，即使你不定义这个段，编译器也会自动把整个程序看成一个段。

但是，因为该定义中有“vstart=0x7c00”子句，所以，它就不那么多余了。一旦有了该子句，段内所有元素的汇编地址都将从 0x7c00 开始计算。否则，因为主引导程序的实际加载地址是 0x0000:0x7c00，当我们引用一个标号时，还得手工加上那个落差 0x7c00。

代码清单 8-1 第 12~14 行，用于初始化堆栈段寄存器 SS 和堆栈指针 SP。之后，堆栈的段地址是 0x0000，段的长度是 64KB，堆栈指针将在段内 0xFFFF 和 0x0000 之间变化。

代码清单 8-1 第 16、17 行，用于取得一个地址，用户程序将要从这个地址处开始加载。

该地址实际上是保存在标号 phy_base 处的一个双字单元里。这是一个 32 位的数，在 16 位的处理器上，只能用两个寄存器存放。如图 8-8 所示，32 位数内存中的存放是按低端序列的，高 16 位处在 phy_base+0x02 处，可以放在寄存器 DX 中；低 16 位处在 phy_base 处，可以用寄存器 AX 存放。

这两条指令中都使用了段超越前缀“cs:”。这是允许的，意味着在访问内存单元时，使用 CS 的内容作为段基址。之所以没有使用 DS 和 ES，是因为它们另有安排。

另外注意，因为段寄存器 CS 的内容是 0x0000，而且主引导扇区是位于 0x0000:0x7c00 处的，所以，理论上指令中的偏移地址应当是 0x7c00+phy_base。不过，因为我们定义段 mbr 的时候，使用了“vstart=0x7c00”子句，故段内所有汇编地址都是在 0x7c00 的基础上增加的，就不用再加上这个 0x7c00 了，直接是

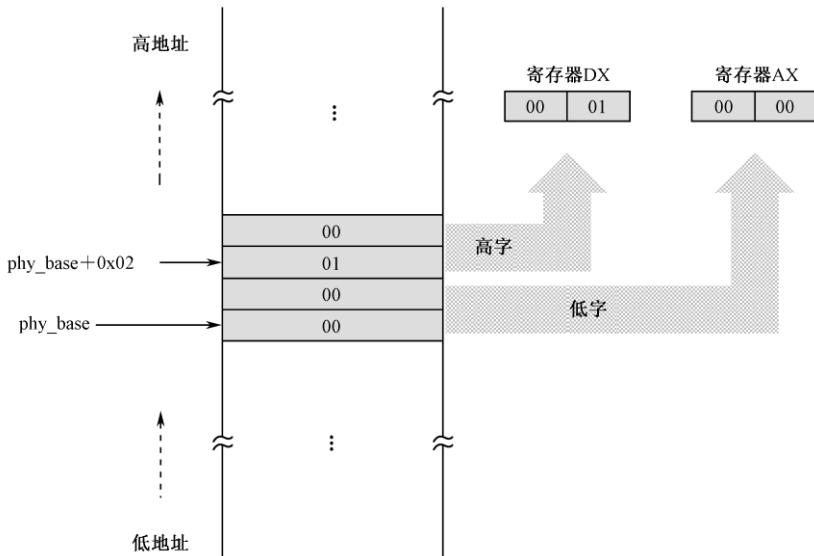


图 8-8 获取用于加载用户程序的物理地址

```
mov ax, [cs:phy_base]
mov dx, [cs:phy_base+0x02]
```

紧接着，代码清单 8-1 第 18~21 行，用于将该物理地址变成 16 位的段地址，并传送到 DS 和 ES 寄存器。因为该物理地址是 16 字节对齐的，直接右移 4 位即可。实际上，右移 4 位相当于除以 16 (0x10)，所以程序中的做法将这个 32 位物理地址 (DX:AX) 除以 16 (在寄存器 BX 中)，寄存器 AX 中的商就是得到的段地址 (在本程序中是 0x1000)。

8.3.3 外围设备及其接口

加载器的下一个工作是从硬盘读取用户程序，说白了就是访问其他硬件。和处理器打交道的硬件很多，不单单是硬盘，还有显示器、网络设备、扬声器（喇叭）和话筒（麦克风）、键盘、鼠标等。有时候，根据应用的场合，还会接一些你不认识和没见过的东西。

所有这些和计算机主机连接的设备，都围绕在主机周围，争着跟计算机说话，叫做外围设备 (Peripheral Equipment)。一般来说，我们把这些设备分成两种，一种是输入设备，比如键盘、鼠标、麦克风、摄像头等；另一种是输出设备，比如显示器、打印机、扬声器等。输入设备和输出设备统称输入输出 (Input/Output, I/O) 设备。

每一种设备都有自己的怪脾气，都有和别的设备不一样的工作方式。比如，扬声器需要的是模拟信号，每个扬声器需要两根线，用的插头也是无线电行业里的标准，话筒也是如此；老式键盘只用一根线向主机传送按键的 ASCII 码，而且一直采用 PS/2 标准；新式的 USB 键盘尽管也使用串行方式工作，但信号却和老式键盘完全不同。至于网络设施，现在流行的是里面有 8 根线芯的五类双绞线，里面的信号也有专门的标准。

一句话，不同的设备，有不同的连线数量，线里面传送的信号也不一样，而且各自的插头和插孔也千差万别，这该如何让处理器跟它打交道？

话虽这么说，但这些东西不让处理器访问和控制却不行。很明显，这里需要一些信号转换器和变速齿轮，这就是 I/O 接口。举几个例子，麦克风和扬声器需要一个 I/O 接口，即声卡，才能与处理器沟通；显示器也需要一个 I/O 接口，即显卡，才能与处理器沟通；USB 键盘同样需要一

个 I/O 接口，即 USB 接口，才能与处理器沟通。很显然，不同的外围设备，都有各自不同的 I/O 接口。

I/O 接口可以是一个电路板，也可能是一块小芯片，这取决于它有多复杂。无论如何，它是一个典型的变换器，或者说是一个翻译器，在一边，它按处理器的信号规程工作，负责把处理器的信号转换成外围设备能接受的另一种信号；在另一边，它也做同样的工作，把外围设备的信号转换成处理器可以接受的形式。

这还没完，后面还有两个麻烦的问题。

① 不可能将所有的 I/O 接口直接和处理器相连，设备那么多，还有些设备现在没有发明出来，将来一定会有。你怎么办？

② 每个设备的 I/O 接口都抢着和处理器说话，不发生冲突都难。你怎么办？

对第 1 个问题的解答是采用总线技术。总线可以认为是一排电线，所有的外围设备，包括处理器，都连接到这排电线上。但是，每个连接到这排电线上的器件都必须有拥有电子开关，以便它们随时能够同这排电线连接，或者从这排电线上断开（脱离）。这就好比是公共车道，当路面上有车时，你就必须退避一下，不能硬冲上去。因此，这排公共电线就称为总线（Bus）。

对第 2 个问题的解答是使用输入输出控制设备集中器（I/O Controller Hub, ICH）芯片，该芯片的作用是连接不同的总线，并协调各个 I/O 接口对处理器的访问。在个人计算机上，这块芯片就是所谓的南桥。

如图 8-9 所示，处理器通过局部总线连接到 ICH 内部的处理接口电路。然后，在 ICH 内部，又通过总线与各个 I/O 接口相连。

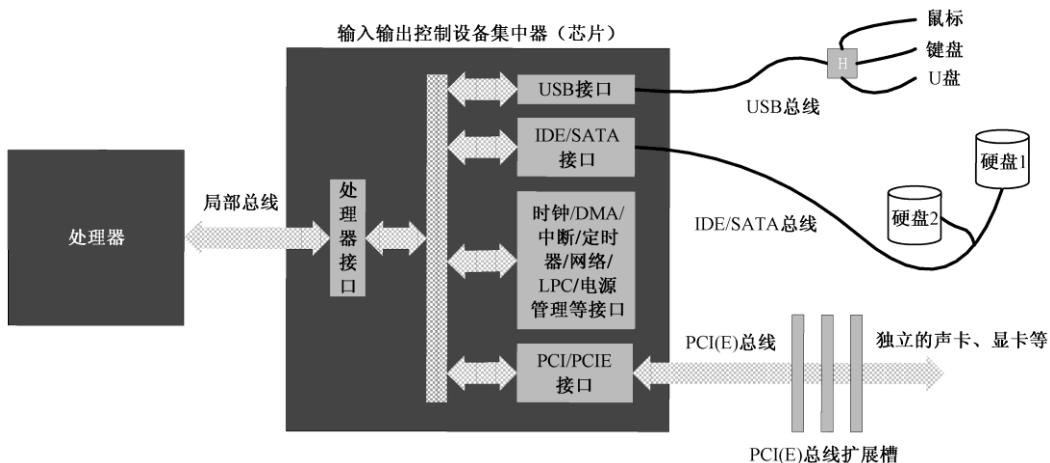


图 8-9 计算机内部总线系统示意图

在 ICH 内部，集成了一些常规的外围设备接口，如 USB、PATA (IDE)、SATA、老式总线接口 (LPC)、时钟等，这些东西对计算机来说必不可少，故直接集成在 ICH 内，我们后面还会详细介绍它们的功能。

除了这些常用的、必不可少的设备之外，有些设备你可能暂时用不上，也有些设备还没有发明出来，但迟早有可能连在计算机上。不管是什么设备，都必须通过它自己的 I/O 接口电路同 ICH 相连。为了方便，最好是在主板上做一些插槽，同时，每个设备的 I/O 接口电路都设计成插卡。这样，想接上该设备时，就把它的 I/O 接口卡插上，不需要时，随时拔下。

为了实现这个目的，或者说为了支持更多的设备，ICH 还提供了对 PCI 或者 PCI Express 总线的支持，该总线向外延伸，连接着主板上的若干个扩展槽，就是刚才说的插槽。举个实例，如果你想连

接显示器，那么就要先插入显卡，然后再把显示器接到显卡上。

除了局部总线和 PCI Express 总线，每个 I/O 接口卡可能连接不止一个设备。比如 USB 接口，就有可能连接一大堆东西：键盘、鼠标、U 盘等。因为同类型的设备较多，也涉及线路复用和仲裁的问题，故它们也有自己的总线体系，称为通信总线或者设备总线。比如图 8-9 所示的 USB 总线和 SATA 总线。

当处理器想同某个设备说话时，ICH 会接到通知。然后，它负责提供相应的传输通道和其他辅助支持，并命令所有其他无关设备闭嘴。同样，当某个设备要跟处理器说话，情况也是一样。

8.3.4 I/O 端口和端口访问



外围设备和处理器之间的通信是通过相应的 I/O 接口进行的。当然，这么说太过于笼统，所以必须具体到细节上来讲这件事。

具体地说，处理器是通过端口（Port）来和外围设备打交道的。本质上，端口就是一些寄存器，类似于处理器内部的寄存器。不同之处仅仅在于，这些叫做端口的寄存器位于 I/O 接口电路中。

端口是处理器和外围设备通过 I/O 接口交流的窗口，每一个 I/O 接口都可能拥有好几个端口，分别用于不同的目的。比如，连接硬盘的 PATA/SATA 接口就有几个端口，分别是命令端口（当向该端口写入 0x20 时，表明是从硬盘读数据；写入 0x30 时，表明是向硬盘写数据）、状态端口（处理器根据这个端口的数据来判断硬盘工作是否正常，操作是否成功，发生了哪种错误）、参数端口（处理器通过这些端口告诉硬盘读写的扇区数量，以及起始的逻辑扇区号）和数据端口（通过这个端口连续地取得要读出的数据，或者通过这个端口连续地发送要写入硬盘的数据）。

端口只不过是位于 I/O 接口上的寄存器，所以，每个端口有自己的数据宽度。在早期的系统中，端口可以是 8 位的，也可以是 16 位的，现在有些端口会是 32 位的。到底是 8 位还是 16 位，这是设备和 I/O 接口制造者的自由。比如，PATA/STAT 接口中的数据端口就是 16 位的，这有助于加快数据传输速率，提高传输效率。

端口在不同的计算机系统中有着不同的实现方式。在一些计算机系统中，端口号是映射到内存地址空间的。比如，0x00000~0xE0000 是真实的物理内存地址，而 0xE0001~0xFFFFF 是从很多 I/O 接口那里映射过来的，当访问这部分地址时，实际上是在访问 I/O 接口。

而在另一些计算机系统中，端口是独立编址的，不和内存发生关系。如图 8-10 所示，在这种计算机中，处理器的地址线既连接内存，也连接每一个 I/O 接口。但是，处理器还有一个特殊的引脚 M/IO#，在这里，“#”表示低电平有效。也就是说，当处理器访问内存时，它会让 M/IO# 引脚呈高电平，这里，和内存相关的电路就会打开；相反，如果处理器访问 I/O 端口，那么 M/IO# 引脚呈低平，内存电路被禁止。与此同时，处理器发出的地址和 M/IO# 信号一起用于打个某个 I/O 接口，如果该 I/O 接口分配的端口号与处理器地址相吻合的话。

Intel 处理器，早期是独立编址的，现在既有内存映射的，也有独立编址的。在本章中，我们只讲独立编址的端口。

所有端口都是统一编号的，比如 0x0001、0x0002、0x0003、…。每个 I/O 接口电路都分配了若干个端口，比如，I/O 接口 A 有 3 个端口，端口号分别是 0x0021~0x0023；I/O 接口 B 需要 5 个端口，端口号分别是 0x0303~0x0307。

一个现实的例子是个人计算机中的 PATA/SATA 接口（图 8-9），每个 PATA 和 SATA 接口分配了 8 个端口。但是，ICH 芯片内部通常集成了两个 PATA/SATA 接口，分别是主硬盘接口和副硬盘接口。这样一来，主硬盘接口分配的端口号是 0x1f0~0x1f7，副硬盘接口分配的端口号是 0x170~0x177。

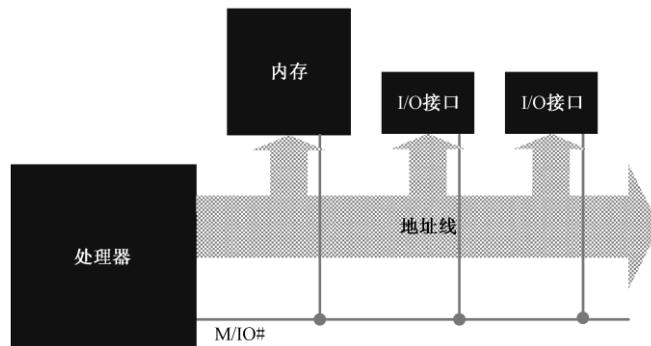


图 8-10 端口的访问和 M/IO#引脚

在 Intel 的系统中，只允许 65536(十进制数)个端口存在，端口号从 0 到 65535(0x0000~0xffff)。因为是独立编址，所以，端口的访问不能使用类似于 mov 这样的指令，取而代之的是 in 和 out 指令。

in 指令是从端口读，它的一般形式是

```
in al,dx
```

或者

```
in ax,dx
```

这就是说，in 指令的目的操作数必须是寄存器 AL 或者 AX，当访问 8 位的端口时，使用寄存器 AL；访问 16 位的端口时，使用 AX。in 指令的源操作数应当是寄存器 DX。

in al,dx 的机器指令码是 0xEC，in ax,dx 的机器指令码是 0xED，都是一字节的。之所以如此简短，是因为 in 指令不允许使用别的通用寄存器，也不允许使用内存单元作为操作数。

也许是出于方便，in 指令还有两字节的形式。此时，前一字节是操作码 0xE4 或者 0xE5，分别用于指示 8 位或者 16 位端口访问；后一字节是立即数，指示端口号。

因此，机器指令 E4 F0 就相当于汇编语言指令

```
in al,0xf0
```

而机器指令 E5 03 就相当于汇编语言指令

```
in ax,0x03
```

很显然，因为这种指令形式的操作数部分只允许一字节，故只能访问 0~255 (0x00~0xff) 号端口，不允许访问大于 255 的端口号。所以，下面的汇编语言指令就是非法的：

```
in ax,0x5fd
```

in 指令不影响任何标志位。

相应地，如果要通过端口向外围设备发送数据，则必须通过 out 指令。

out 指令正好和 in 指令相反，目的操作数可以是 8 位立即数或者寄存器 DX，源操作数必须是寄存器 AL 或者 AX。下面是一些例子：

```
out 0x37,al ;写 0x37 号端口 (这是一个 8 位端口)
```

```
out 0xf5,ax ;写 0xf5 号端口 (这是一个 16 位端口)
```

```
out dx,al ;这是一个 8 位端口，端口号在寄存器 DX 中
```

```
out dx,ax ;这是一个 16 位端口，端口号在寄存器 DX 中
```

和 in 指令一样，out 指令不影响任何标志位。

8.3.5 通过硬盘控制器端口读扇区数据

现在，让我们来看看硬盘。

硬盘读写的基本单位是扇区。就是说，要读就至少读一个扇区，要写就至少写一个扇区，不可能仅读写一个扇区中的几个字节。这样一来，就使得主机和硬盘之间的数据交换是成块的，所以硬盘是典型的块设备。

从硬盘读写数据，最经典的方式是向硬盘控制器分别发送磁头号、柱面号和扇区号（扇区在某个柱面上的编号），这称为 CHS 模式。这种方法最原始，最自然，也最容易理解。

实际上，在很多时候，我们并不关心扇区的物理位置，所以希望所有的扇区都能统一编址。这就是逻辑扇区，它把硬盘上所有可用的扇区都一一从 0 编号，而不管它位于哪个盘面，也不管它属于哪个柱面。

关于硬盘和逻辑扇区的知识前面已经有所介绍，这里不再赘述。最早的逻辑扇区编址方法是 LBA28，使用 28 个比特来表示逻辑扇区号，从逻辑扇区 0x00000000 到 0xFFFFFFFF，共可以表示 $2^{28} = 268435456$ 个扇区。每个扇区有 512 字节，所以 LBA28 可以管理 128 GB 的硬盘。

硬盘技术发展得非常快，最新的硬盘已经达到几百个吉字节的容量，LBA28 已经落后了。在这种情况下，业界又共同推出了 LBA48，采用 48 个比特来表示逻辑扇区号。如此一来，就可以管理 131072 TB 的硬盘容量了。

1TB = 1024GB

在本章中，我们将采用 LBA28 来访问硬盘。

前面说过，个人计算机上的主硬盘控制器被分配了 8 位端口，端口号从 0x1f0 到 0x1f7。假设现在要从硬盘上读逻辑扇区，那么，整个过程如下。

第 1 步，设置要读取的扇区数量。这个数值要写入 0x1f2 端口。这是个 8 位端口，因此每次只能读写 255 个扇区：

```
mov dx,0x1f2
mov al,0x01          ;1 个扇区
out dx,al
```

注意，如果写入的值为 0，则表示要读取 256 个扇区。每读一个扇区，这个数值就减一。因此，如果在读写过程中发生错误，该端口包含着尚未读取的扇区数。

第 2 步，设置起始 LBA 扇区号。扇区的读写是连续的，因此只需要给出第一个扇区的编号就可以了。28 位的扇区号太长，需要将其分成 4 段，分别写入端口 0x1f3、0x1f4、0x1f5 和 0x1f6 号端口。其中，0x1f3 号端口存放的是 0~7 位；0x1f4 号端口存放的是 8~15 位；0x1f5 号端口存放的是 16~23 位，最后 4 位在 0x1f6 号端口。假定我们要读写的起始逻辑扇区号为 0x02，可编写代码如下：

```
mov dx,0x1f3
mov al,0x02
out dx,al          ;LBA 地址 7~0
inc dx            ;0x1f4
mov al,0x00
out dx,al          ;LBA 地址 15~8
inc dx            ;0x1f5
out dx,al          ;LBA 地址 23~16
```

```

inc dx           ; 0x1f6
mov al,0xe0      ; LBA 模式，主硬盘，以及 LBA 地址 27~24
out dx,al

```

注意以上代码的最后 4 行，在现行的体系下，每个 PATA/SATA 接口允许挂接两块硬盘，分别是主盘（Master）和从盘（Slave）。如图 8-11 所示，0x1f6 端口的低 4 位用于存放逻辑扇区号的 24~27 位，第 4 位用于指示硬盘号，0 表示主盘，1 表示从盘。高 3 位是“111”，表示 LBA 模式。

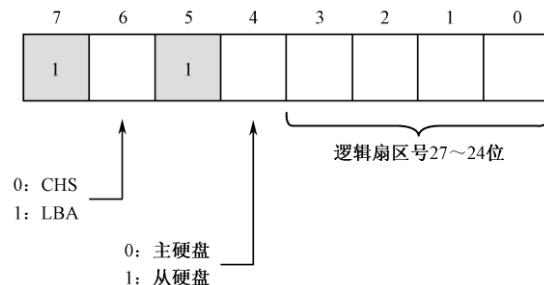


图 8-11 端口 1f6 各位的含义

第 3 步，向端口 0x1f7 写入 0x20，请求硬盘读。这也是一个 8 位端口：

```

mov dx,0x1f7
mov al,0x20      ; 读命令
out dx,al

```

第 4 步，等待读写操作完成。端口 0x1f7 既是命令端口，又是状态端口。在通过这个端口发送读写命令之后，硬盘就忙乎开了。如图 8-12 所示，在它内部操作期间，它将 0x1f7 端口的第 7 位置“1”，表明自己很忙。一旦硬盘系统准备就绪，它再将此位清零，说明自己已经忙完了，同时将第 3 位置“1”，意思是准备好了，请求主机发送或者接收数据（图 8-12）。完成这一步的典型代码如下：

```

mov dx,0x1f7
.waits:
    in al,dx
    and al,0x88
    cmp al,0x08
    jnz .waits      ; 不忙，且硬盘已准备好数据传输

```

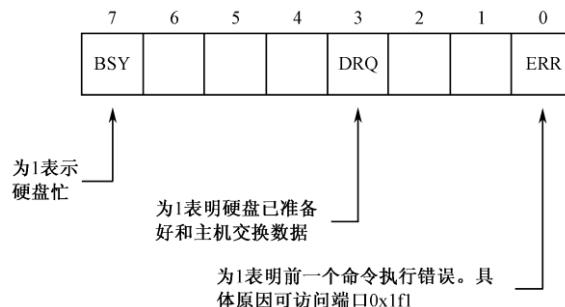


图 8-12 端口 0x1f7 部分状态位的含义

来看看指令 and al,0x88。0x88 的二进制形式是 10001000，这意味着我们想用这条指令保留住寄存器 AL 中的第 7 位和第 3 位，其他无关的位都清零。此时，如果寄存器 AL 中的二进制数是 00001000（0x08），那就说明可以退出等待状态，继续往下操作，否则继续等待。

第5步，连续取出数据。0x1f0是硬盘接口的数据端口，而且还是一个16位端口。一旦硬盘控制器空闲，且准备就绪，就可以连续从这个端口写入或者读取数据。下面的代码假定是从硬盘读一个扇区（512字节，或者256字节），读取的数据存放到由段寄存器DS指定的数据段，偏移地址由寄存器BX指定：

```
mov cx, 256           ;总共要读取的字数
mov dx, 0x1f0
.readw:
    in ax, dx
    mov [bx], ax
    add bx, 2
loop .readw
```

最后，0x1f1端口是错误寄存器，包含硬盘驱动器最后一次执行命令后的状态（错误原因）。

8.3.6 过程调用

读写硬盘是经常要做的事，尤其对于操作系统来说。即使是在本章的程序中，也多次发生。如果每次读写硬盘都按上面的5个步骤写一堆代码，程序势必很大，也会令人烦恼。

好在处理器支持一种叫过程调用的指令执行机制。过程（Procedure）又叫例程，或者子程序、子过程、子例程（Sub-routine），不管怎么称呼，实质都一样，都是一段普通的代码。处理器可以用过程调用指令转移到这段代码执行，在遇到过程返回指令时重新返回到调用处的下一条指令接着执行。

如图8-13所示，这是过程和过程调用的示意图。下面结合本章代码清单来具体说明。

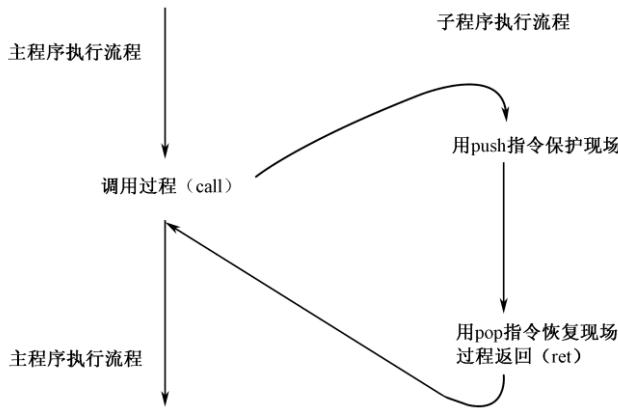


图8-13 过程和过程调用示意图

在8.3.1节里，我们已经定义了常量app_lba_start，它代表的值是100，也就是用户程序在硬盘上的起始逻辑扇区号。现在，代码清单8-1的第24~27行用于从硬盘上读取这个扇区的内容。这很好理解，因为不知道用户程序到底有多大，到底占用了多少个扇区，所以，可以先读它的第一个扇区。该扇区包含了用户程序的头部，而用户程序头部又包含了该程序的大小、入口点和段重定位表。所以，通过分析头部，就知道接着还要再读多少个扇区才能完全加载用户程序。

因为要多次读取硬盘，而每次的步骤又都差不多，所以，我们精心设计了一段通用的代码，它从代码清单8-1的第79行开始，一直到第131行结束，这就是我们所说的过程。

要调用过程，需要该过程的地址。一般来说，过程的第一条指令需要一个标号，以方便引用该过程。所以，代码清单 8-1 第 79 行是一个标号“read_hard_disk_0”，意思是读（第一个硬盘控制器的）主盘，当然，什么意思并不重要。

编写过程的好处是只用编写一次，以后只需要“调用”即可。所以，代码的灵活性和通用性尤其重要。具体到这里，就是每次读硬盘时的起始逻辑扇区号和数据保存位置都不相同，这就涉及所谓的参数传递。

参数传递最简单的办法是通过寄存器。在这里，主程序把起始逻辑扇区号的高 16 位存放在寄存器 DI 中（只有低 12 位是有效的，高 4 位必须保证为“0”），低 16 位存放在寄存器 SI 中（没办法，16 位的处理器无法直接处理 28 位的数据）；并约定将读出来的数据存放到由段寄存器 DS 指向的数据段中，起始偏移地址在寄存器 BX 中。

在调用过程前，程序会用到一些寄存器，在过程返回之后，可能还要继续使用。为了不失连续性，在过程的开头，应当将本过程要用到（内容肯定会被破坏）的寄存器临时压栈，并在返回到调用点之前出栈恢复。代码清单 8-1 的第 82~85 行，用于将过程中用到的寄存器压入堆栈保存。

后面的指令都很好理解，第 87~89 行，是向 0x1f2 端口写入要读取的扇区数。显而易见，每次读的扇区数是 1 个。

第 91~101 行，用于向硬盘接口写入起始逻辑扇区号的低 24 位。低 16 位在寄存器 SI 中，高 12 位在寄存器 DI 中，需要不停地倒换到寄存器 AL 中，以方便端口写入。

第 105 行，程序执行到这里时，寄存器 AH 的低 4 位是起始逻辑扇区号的 27~24 位，高 4 位是全“0”；寄存器 AL 中是 0xe0。执行 or 指令后，将会在寄存器 AL 中得到它们的组合值，高 4 位是 0xe，低 4 位是逻辑扇区号的 27~24 位。

第 118~124 行，用于反复从硬盘接口那里取得 512 字节的数据，并传送到段寄存器 DS 所指向的数据区中。每传送一个字，BX 的值就增 2，以指向下一个偏移位置。

第 126~129 行，用于把调用过程前各个寄存器的内容从堆栈中恢复。

最后，因为处理器是没有大脑的，所以需要一个明确的指令 ret 促使它离开过程，从哪里来回哪里去，这条指令稍后就会讲到。

有关过程的情况就是这些，下面回到前面，看看过程调用是如何发生的。

代码清单 8-1 第 24、25 行，用于指定用户程序在硬盘上的起始逻辑扇区号。我们定义的过程要求用 DI:SI 来提供这个扇区号，既然它是常数 100，很小的数值，可以直接传送到寄存器 SI，并将 DI 清零即可。

第 26 行用于指定存放数据的内存地址。前面几条指令已经将段寄存器 DS 设置好了，现在只需要将寄存器 BX 清零，以指向该段内偏移地址为 0 的地方，这就是当前指令要做的事。

一切都准备好了，第 27 行，开始调用过程 read_hard_disk_0。以后，我们将把过程所在的标号做为过程的名字，即过程名。

调用过程的指令是“call”。8086 处理器支持四种调用方式。

第一种是 16 位相对近调用。近调用的意思是被调用的目标过程位于当前代码段内，而非另一个不同的代码段，所以只需要得到偏移地址即可。

16 位相对近调用是三字节指令，操作码为 0xE8，后跟 16 位的操作数，因为是相对调用，故该操作数是当前 call 指令相对于目标过程的偏移量。计算过程如下：用目标过程的汇编地址减去当前 call 指令的汇编地址，再减去当前 call 指令以字节为单位的长度（3），保留 16 位的结果。举个例子：

```
call near proc_1
```

近调用的特征是在指令中使用关键字“near”。“proc_1”是程序中的一个标号。在编译阶段，

编译器用标号 proc_1 处的汇编地址减去本指令的汇编地址，再减去 3，作为机器指令的操作数。

关键字“near”不是必需的，如果 call 指令中没有提供任何关键字，则编译器认为该指令是近调用。因此，上面的指令与这条指令等效：

```
call proc_1
```

因为 16 位相对近调用的操作数是两个汇编地址相减的相对量，所以，如果被调用过程在当前指令的前方，也就是说，论汇编地址，它比 call 指令的要大，那么该相对量是一个正数；反之，就是一个负数。所以，它的机器指令操作数是一个 16 位的有符号数。换句话说，被调用过程的首地址必须位于距离当前 call 指令—32768~32767 字节的地方。

在指令执行阶段，处理器看到操作码 0xE8，就知道它应当调用一个过程。于是，它用指令指针寄存器 IP 的当前内容加上指令中的操作数，再加上 3，得到一个新的偏移地址。接着，将 IP 的原有内容压入堆栈。最后，用刚才计算出的偏移地址取代 IP 原有的内容。这直接导致处理器的执行流转移到目标位置处。

再看一个例子：

```
call 0x0500
```

很多人认为 0x0500 会原封不动地出现在该指令编译后的机器码中，我相信这只是他们一时糊涂。在 call 指令后跟一个标号，和跟一个数值没有什么不同。标号是数值的等价形式，是代表标号处的汇编地址。在指令编译阶段，它首先会被转化成数值。

所以，你在 call 指令后跟一个数值，只是帮了编译器的忙，帮它省了一个转化步骤，它依然会用这个数值减去当前指令的汇编地址，来得到一个偏移量。

第二种是 16 位间接绝对近调用。这种调用也是近调用，只能调用当前代码段内的过程，指令中的操作数不是偏移量，而是被调用过程的真实偏移地址，故称为绝对地址。不过，这个偏移地址不是直接出现在指令中，而是由 16 位的通用寄存器或者 16 位的内存单元给出。比如：

call cx	; 目标地址在 CX 中。省略了关键字“near”，下同
call [0x3000]	; 要先访问内存才能取得目标偏移地址
call [bx]	; 要先访问内存才能取得目标偏移地址
call [bx+si+0x02]	; 要先访问内存才能取得目标偏移地址

以上，第一条指令的机器码为 FF D1，被调用过程的偏移地址位于寄存器 CX 内，在指令执行的时候由处理器从该寄存器取得，并直接取代指令指针寄存器 IP 原有的内容。

第二条指令的机器码为 FF 16 00 30。当这条指令执行时，处理器访问数据段（使用段寄存器 DS），从偏移地址 0x3000 处取得一个字，作为目标过程的真实偏移地址，并用它取代指令指针寄存器 IP 原有的内容。

后面两条指令没什么好说的，只是寻址方式不同而已。

间接绝对近调用指令在执行时，处理器首先按以上的方法计算被调用过程的偏移地址，然后将指令指针寄存器 IP 的当前值压栈，最后用计算出来的偏移地址取代寄存器 IP 原有的内容。

由于间接绝对近调用的机器指令操作数是 16 位的绝对地址，因此，它可以调用当前代码段任何位置处的过程。

第三种是 16 位直接绝对远调用。这种调用属于段间调用，即调用另一个代码段内的过程，所以称为远调用（Far Call）。很容易想到，远调用既需要被调用过程所在的段地址，也需要该过程在段内的偏移地址。

“16 位”是针对偏移地址来说的，而不用于限定段地址；“直接”的意思是，段地址和偏移地址直接在 call 指令中给出了。当然，这里的地址也是绝对地址。比如：

```
call 0x2000:0x0030
```

这条指令编译后的机器码为 9A 30 00 00 20，0x9A 是操作码，后面跟着的两个字分别是偏移地址和段地址，按规定，偏移地址在前，段地址在后。

处理器在执行时，首先将代码段寄存器 CS 的当前内容压栈，接着再把指令指针寄存器 IP 的当前内容压栈。紧接着，用指令中给出的段地址代替 CS 原有的内容，用指令中给出的偏移地址代替 IP 原有的内容。这直接导致处理器从新的位置开始执行。

处理器是没有脑子的。如果被调用过程位于当前代码段内，而你又用这种指令格式来调用它，那么，处理器也会不折不扣地从当前代码段“转移”到当前代码段。

第四种是 16 位间接绝对远调用。这也属于段间调用，被调用过程位于另一个代码段内，而且，被调用过程所在的段地址和偏移地址是间接给出的。还有，这里的“16 位”同样是用来限定偏移地址的。下面是这种调用方式的几个例子：

```
call far [0x2000]
call far [proc_1]
call far [bx]
call far [bx+si]
```

间接远调用必须使用关键字“far”，这一点务必牢记。

因为是远调用，也就是段间调用，所以，必须给出被调用过程的段地址和偏移地址。但是，段地址和偏移地址在内存中的其他位置，指令中仅仅给出的是该位置的偏移地址，需要处理器在执行指令的时候自行按图索骥，找到它们。

以上，前两条指令是等效的，不同之处仅仅在于，第一条指令直接给出的是数值，而第二条指令用的是标号。但这无关紧要，在编译后，标号也会变成数值。

为了进一步说清间接远调用是怎么发生的，下面是一个实例。

假如在数据段内声明了标号 proc_1 并初始化了两个字：

```
proc_1 dw 0x0102,0x2000
```

这两个字分别是某个过程的段地址和偏移地址。按处理器的要求，偏移地址在前，段地址在后。也就是说，0x0102 是偏移地址；0x2000 是段地址。

那么，为了调用该过程，可以在代码段内使用这条指令：

```
call far [proc_1]
```

当这条指令执行时，处理器访问由段寄存器 DS 指向的数据段，从指令中指定的偏移地址处取得两个字（分别是段地址 0x2000 和偏移地址 0x0102）；接着，将代码段寄存器 CS 和指令指针寄存器 IP 的当前内容分别压栈；最后，用刚才取得的段地址和偏移地址分别取代 CS 和 IP 的原值。

至于后面的两条指令 call far [bx] 和 call far [bx+si]，仅仅是寻址方式上有所区别，指令执行过程大体上是一样的。

接着回到代码清单 8-1 第 27 行，很明显，

```
call read_hard_disk_0
```

就是我们刚刚讲的 16 位相对近调用，编译后的机器指令操作数是一个相对偏移量。由于这是段内调用，处理器执行这条指令时，用指令指针寄存器 IP 的内容加上指令中的偏移量，以及当前指令的长度，算出被调用过程的绝对偏移地址。接着，将 IP 的现行值压栈。最后，用刚刚计算出的偏移地址替代 IP 的当前内容。

过程 read_hard_disk_0 的功能和工作流程前面已经讲过了，不再赘述。这里只关心一个最重要的问题，那就是过程返回。

“过程”就是例行公事，可以随时根据需要调用，但过程执行完了呢，还得返回到调用点继续执行下一条指令，这称为过程返回（Procedure Return）。

处理器是个大笨蛋，你不提醒它，它就一直稀里糊涂地闷头工作。幸好，处理器的发明者们设计了返回指令 ret 和 retf。

ret 和 retf 经常用做 call 和 call far 的配对指令。ret 是近返回指令，当它执行时，处理器只做一件事，那就是从堆栈中弹出一个字到指令指针寄存器 IP 中。

retf 是远返回指令（return far），它的工作稍微复杂一点点。当它执行时，处理器分别从堆栈中弹出两个字到指令指针寄存器 IP 和代码段寄存器 CS 中。

如图 8-14 所示，在 call read_hard_disk_0 执行前，堆栈指针位于箭头①所指示的位置；call 指令执行后，由于压入了 IP 的内容，故堆栈指针移动到箭头②所指示的位置处；进入过程后，出于保护现场的目的，压入了 4 个通用寄存器 AX、BX、CX、DX，此时，堆栈指针继续向低地址方向推进到箭头③所指示的位置。

在过程的最后，是恢复现场，连续反序弹出 4 个通用寄存器的内容。此时，堆栈指针又回到刚进入过程内部时的位置，即箭头②处。最后，ret 指令执行时，由于处理器自动弹出一个字到 IP，故，过程返回后的瞬间，堆栈指针仍旧回到过程调用前，即箭头①所指示的位置。

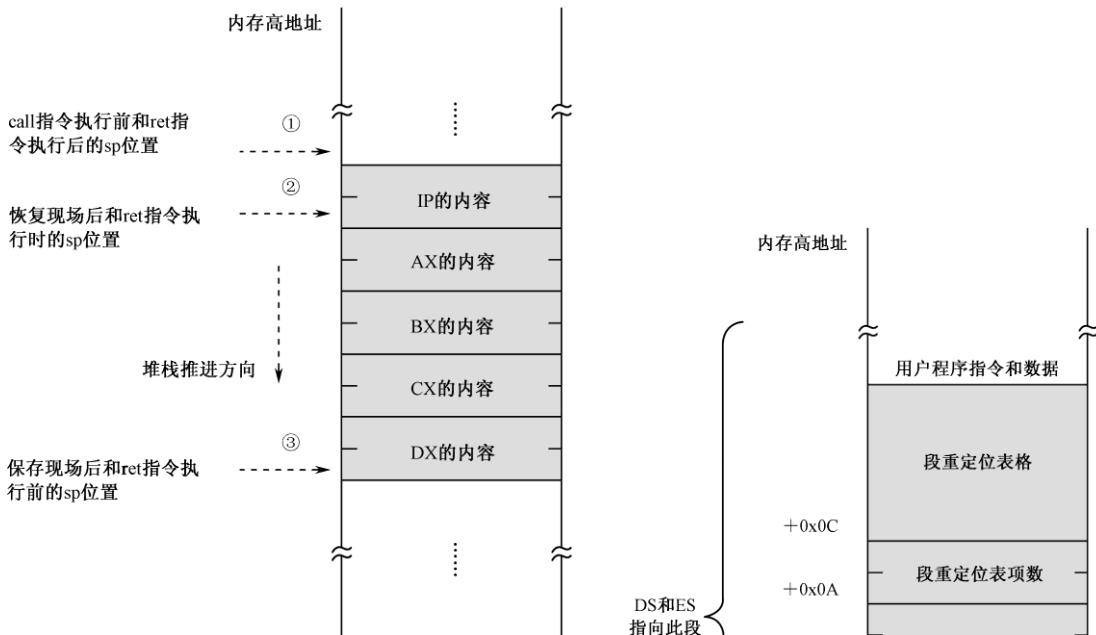


图 8-14 过程调用前后的堆栈变化

需要说明的是，尽管 call 指令通常需要 ret/retf 和它配对，遥相呼应，但 ret/retf 指令却并不依赖于 call 指令，这一点你马上就会看到。

call 指令在执行过程调用时不影响任何标志位，ret/retf 指令对标志位也没有任何影响。

8.3.7 加载用户程序

第一次读硬盘将得到用户程序最开始的 512 字节，这 512 字节包括最开始的用户程序头部，以及一部分实际的指令和数据。

为了将用户程序全部读入内存，需要知道它的大小，然后再进一步转换成它所用的扇区数。如图 8-15 所示，用户程序最开始的双字，就是整个程序的大小。

为此，代码清单 8-1 第 30、31 行，分别将该数值的高 16 位和低 16 位传送到寄存器 DX 和 AX。第 32 行，因为每扇区有 512 字节，故将 512 传送到 BX 寄存器，并在第 33 行用它来做除法运算。

在凑巧的情况下，用户程序的大小正好是 512 的整数倍，做完除法后，在寄存器 AX 中是用户程序实际占用的扇区数。但是，绝大多数情况下，这个除法会有余数。有余数意味着，最后一个扇区因为没有填满而落下了，没有纳入总扇区数。

关于这个问题，我们稍微解释一下。硬盘的读写是以扇区为单位的，如果要写入 513 字节，那么，它将只能填满一个扇区，还剩一字节。硬盘不管这些，它每次总是说：“来，给我 512 字节！”为此，软件的责任是，保证给硬盘的是 512 字节，如果不够，凑也要凑够。因此，513 字节会占用两个扇区，第二个扇区只有一字节是有用的，其他 511 字节都是用来填充的。至于某个扇区里，哪些数据是有用的，哪些是填充的，不是硬盘的责任，是软件的责任。就像本章的用户程序一样，通过构造一个头部，自行来跟踪自己的大小。

所以，代码清单 8-1 第 34 行，判断是否除尽。如果没有除尽，则转移到后面的代码，去读剩余的扇区；如果除尽了，则总扇区数减一。

为什么？为什么除不尽不管，除尽了还要减一？因为刚才已经预读了一个扇区。

注意，用户程序的长度有可能小于 512 字节，或者恰好等于 512 字节。在这两种情况下，当程序执行到第 38 行时，寄存器 AX 中的内容必然为零。所以，第 38 行是算术比较指令 cmp，第 39 行是条件转移指令，当寄存器 AX 中的内容为零时，就意味着用户程序已经全部读取，不再继续读了，毕竟用户程序只占用一个扇区，而刚才也已经读过了。

用户程序被加载的位置是由 DS 和 ES 所指向的逻辑段。一个逻辑段最大也才 64KB，当用户程序特别大的时候，根本容纳不下。想想看，段内偏移地址从 0x0000 开始，一直延伸到最大值 0xffff。再大的话，又绕回到 0x0000，以致于把最开始加载的内容给覆盖掉了。

其实，要解决这个问题最好的办法是，每次往内存中加载一个扇区前，都重新在前面的数据尾部构造一个新的逻辑段，并把要读取的数据加载到这个新段内。如此一来，因为每个段的大小是 512 字节，即，十六进制的 0x200，右移 4 位（相当于除以 16 或者 0x10）后是 0x20，这就是各个段地址之间的差值。每次构造新段时，只需要在前面段地址的基础上增加 0x20 即可得到新段的段地址。

这种做法好有一比，尺子很短，树很高，想只量一次是不可能的，于是只好分几次量，每量一次，将尺子往下挪一挪。

段地址的改变是临时的，毕竟只是为了读取硬盘，所以，代码清单 8-1 第 42 行，将当前数据段寄存器 DS 的内容压栈保存。

第 44 行，将用户程序剩余的扇区数传送到寄存器 CX，供后面的 loop 指令使用，因为我们准备采用循环的办法来读完用户程序。

第 46~48 行，将当前数据段寄存器 DS 的内容在原来的基础上增加 0x20，以构造出下一个逻辑段，为从硬盘上读取下一个 512 字节的数据做准备。

第 50 行，将寄存器 BX 清零。BX 被用做数据传输时的段内偏移，而且每次传输都是在一个新的段内进行，故偏移地址在每次传输前都应当是零。

第 51 行，每次读硬盘前，将寄存器 SI 的内容加一，以指向下一个逻辑扇区。

第 52~53 行，调用读硬盘的过程 `read_hard_disk_0`，并开始下一轮循环，直到所有的扇区都读完（寄存器 CX 的内容为 0）。

8.3.8 用户程序重定位

用户程序在编写的时候是分段的。因此，加载器下一步的工作是计算和确定每个段的段地址。

如图 8-16 所示，用户程序定义了 6 个段，在编译阶段，编译器为每个段计算了一个汇编地址。第一个段 `header` 位于整个程序的开头，所以其汇编地址为 0。从第二个段开始，每个段的汇编地址都是其相对于整个程序开头的偏移量，以字节为单位。因为我们不知道各个段的汇编地址到底是多少，故用字母来表示。这样，第二个段 `code_1` 的汇编地址是 `v`，第三个段 `code_2` 的汇编地址是 `w`，……，最后一个段 `stack` 的汇编地址是 `z`。

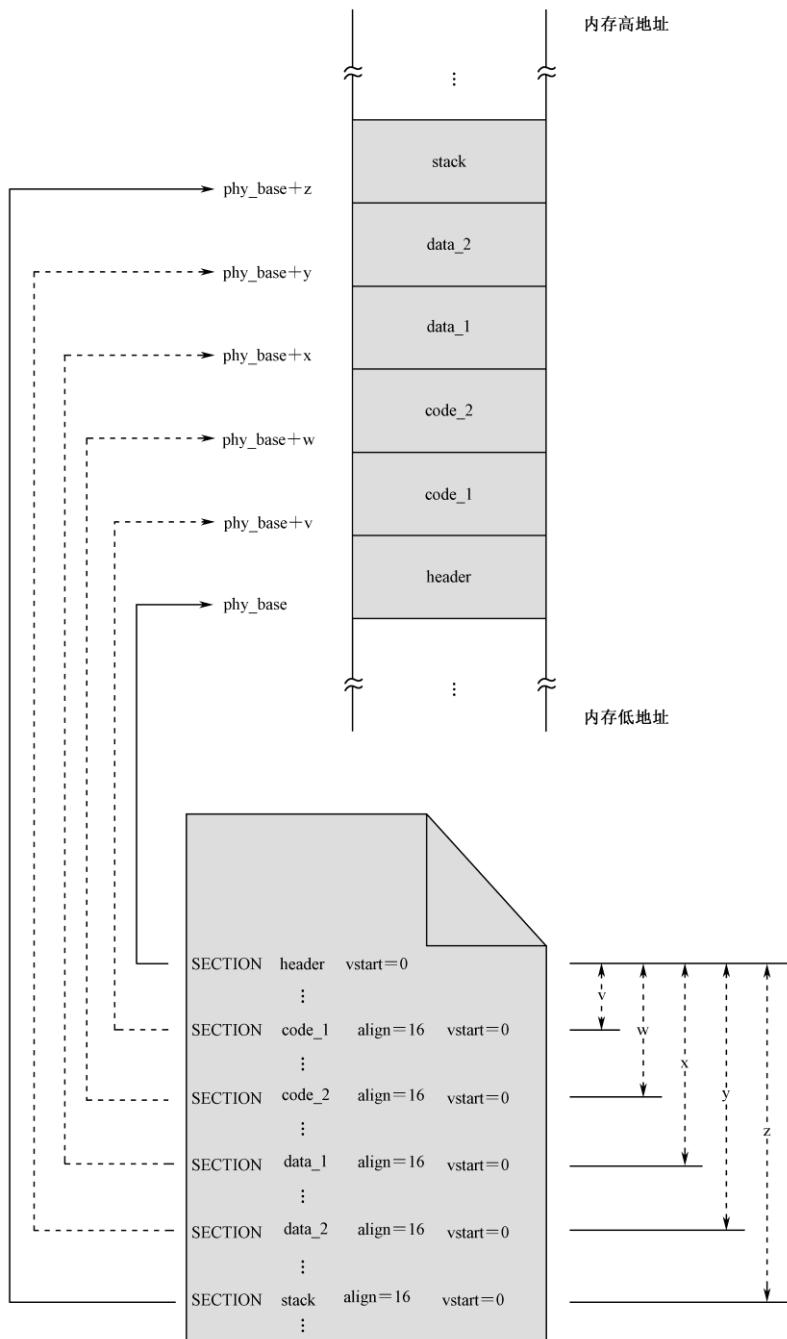


图 8-16 段的偏移地址和它在内存中的物理地址

现在，用户程序已经全部加载到内存里了，而且是从物理地址 `phy_base` 开始的。如此一来，每个段在内存中的物理地址都是基于 `phy_base` 的，第一个段 `header` 在内存中的起始物理地址是 `phy_base` (`phy_base+0`)，第二个段在内存中的起始物理地址是 `phy_base+v`，……，最后一个段 `stack` 则是 `phy_base+z`。

用于加载用户程序的物理地址 `phy_base` 是 16 字节对齐的，而用户程序中，每个段的汇编地址也是 16 字节对齐的。因此，每个段在内存中的起始地址也是 16 字节对齐的，将它们分别右移 4 位，

就是它们各自的逻辑段地址。

为此，代码清单 8-1 第 55 行，从堆栈中恢复数据段寄存器 DS 的内容，使其指向用户程序被加载的起始位置，也就是用户程序头部。

第 58~62 行用于重定位用户程序入口点的代码段。请参考图 8-15，用户程序头部内，偏移为 0x06 处的双字，存放的是入口点代码段的汇编地址。加载器首先将高字和低字分别传送到寄存器 DX 和 AX，然后调用过程 calc_segment_base 来计算该代码段在内存中的段地址。

过程 calc_segment_base（计算段基址）是在代码清单 8-1 的第 134 行定义的。它接受一个 32 位的汇编地址（位于寄存器 DX:AX 中），并在计算完成后向主程序返回一个 16 位的逻辑段地址（位于寄存器 AX 中）。

因为计算过程中要破坏寄存器 DX 的内容，因此，第 137 行用于将其压栈保存。

在 16 位的处理器上，每次只能进行 16 位数的运算。第 139 行，先将用户程序在内存中物理起始地址的低 16 位加到寄存器 AX 中。该指令的地址部分使用了段超越前缀“cs:”，而且也没有加上 0x7c00。原因前面已经解释过了，在本程序中，数据段和代码段是分离的，而且当前代码段的定义部分使用了“vstart=0x7c00”子句。

然后，第 140 行，再将该起始地址的高 16 位加到寄存器 DX 中。adc 是带进位加法，它将目的操作数和源操作数相加，然后再加上标志寄存器 CF 位的值（0 或者 1）。这样，分两步就可以完成 32 位数的加法运算。

现在，我们已经在 DX:AX 中得到了入口点代码段的起始物理地址，只需要将这个 32 位数右移 4 位即可得到逻辑段地址。麻烦在于它们分别在两个寄存器中，如何移动？

答案是分别移动，然后拼接。代码清单 8-1 第 141 行，使用逻辑右移指令 shr (SHift logical Right) 将寄存器 AX 中的内容右移 4 位。

如图 8-17 所示，逻辑右移指令执行时，会将操作数连续地向右移动指定的次数，每移动一次，“挤”出来的比特被移到标志寄存器的 CF 位，左边空出来的位置用比特“0”填充。

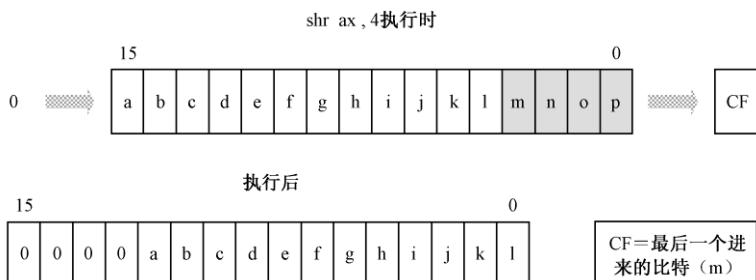


图 8-17 逻辑右移示意图

shr 指令的目的操作数可以是 8 位或 16 位的通用寄存器或者内存单元，源操作数可以是数字 1、8 位立即数或者寄存器 CL。我们已经介绍过寻址方式，往后，我们要用新的方法来表示指令的格式。就当前指令来说，该指令的格式为：

```
shr r/m8,1  
shr r/m16,1  
shr r/m8,imm8  
shr r/m16,imm8  
shr r/m8,cl  
shr r/m16,cl
```

以上，第一种指令格式的意思是，目的操作数可以是 8 位寄存器，或者指向 8 位实际操作数的

内存地址；源操作数是 1。对于内存地址的情况，可以使用任何一种我们讲过的内存寻址方式。举三个例子：

```
shr ah,1
shr byte [0x2000],1
shr byte [bx+si+0x02],1
```

第二种指令格式和第一种相似，只是目的操作数的长度不一样。注意，源操作数为 1 的逻辑右移指令是特殊设计的优化指令，比如以上的 shr ax,1，它的机器码是 D1 E8；而类似的指令 shr ax,5 则拥有完全不同的机器码 C1 E8 05。

第三种指令格式的意思是，目的操作数可以是 8 位寄存器，或者指向 8 位实际操作数的内存地址；源操作数是 8 位立即数。下面是两个例子：

```
shr al,0x20           ;右移 32 (0x20) 次
shr byte [bx+0x06],0x05 ;右移 5 次
```

第四种指令格式和第二种类似，只是数据宽度不同。

第五种指令格式的目的操作数可以是 8 位的寄存器，或者指向 8 位实际操作数的内存地址；源操作数在寄存器 CL 中。如果 shr 指令的源操作数是寄存器，则只能使用 CL。和一般的指令不同，寄存器 CL 只用来提供移动次数，而不用于限定和暗示目的操作数的字长。因此，对于目的操作数是内存地址的情况，必须用关键字 byte 或者 word 等来加以限定。比如：

```
shr al,cl
shr byte [bx],cl
```

最后一种指令格式适用于目的操作数的长度为字的情况。

注意，和 8086 处理器不同，80286 之后的 IA-32 处理器在执行本指令时，会先将源操作数的高 3 位清零。也就是说，最大的移位次数是 31。

shr 的配对指令是逻辑左移指令 shl (SHift logical Left)，它的指令格式和 shr 相同，只不过它是向左移动。

尽管 DX:AX 中是 32 位的用户程序起始物理内存地址，理论上，它只有 20 位是有效的，低 16 位在寄存器 AX 中，高 4 位在寄存器 DX 的低 4 位。寄存器 AX 经右移后，高 4 位已经空出，只要将 DX 的最低 4 位挪到这里，就可以得到我们所需要的逻辑段地址。为此，可以使用以下指令：

```
shl dx,12
or ax,dx
```

很显然，代码清单 8-1 并不是这么做的，为的是演示另一个不同的指令 ror (第 142 行)，也就是循环右移 (ROtate Right)。如图 8-18 所示，循环右移指令执行时，每右移一次，移出的比特既送到标志寄存器的 CF 位，也送进左边空出的位。

ror 的配对指令是循环左移指令 rol (ROtate Left)。ror、rol、shl、shr 的指令格式都是相同的。

因为是循环移位，移位后，寄存器 DX 的低 12 位是我们不需要的。所以，代码清单 8-1 的第 143 行，用 and 指令将其清零。

第 144 行，正式将寄存器 AX 和 DX 的内容合并，这就是我们要的段地址。

过程的最后，第 146~148 行，恢复寄存器 DX 的原始内容，并返回到调用程序那里。

现在，回到代码清单 8-1 的第 62 行，那条指令的功能是将刚刚计算出来的逻辑段地址回写到原处，仅覆盖低 16 位，高 16 位不用理会。

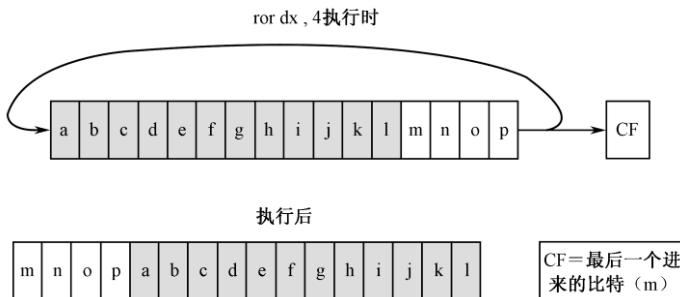


图 8-18 循环右移示意图

现在仅仅是处理了入口点代码段的重定位，下面开始正式处理用户程序的所有段，它们位于用户程序头部的段重定位表中。

重定位表的表项数存放在用户程序头部偏移 0x0a 处，如图 8-5 所示。代码清单 8-1 第 65 行，用于将它从该内存地址传送到寄存器 CX，供后面的循环指令使用。

段重定位表的首地址存放在用户程序头部偏移 0x0c 处，因此，第 66 行，将 0x0c 传送到基址寄存器 BX 中。以后，每次只要将 BX 的内容加上 4，就指向下一个重定位表项。

第 68~74 行是循环体，每次循环开始后，BX 总是指向需要重定位的段的汇编地址，而且都是双字，需要分别传送到寄存器 DX 和 AX。然后调用过程 `calc_segment_base` 计算相应的逻辑段地址，并覆盖到原来的位置（低字），最后将基址寄存器的内容加上 4，以指向下一个表项。当寄存器 CX 的内容为 0 时，循环结束，所有的段都处理完毕。

8.3.9 将控制权交给用户程序

现在，用户程序已经在内存中准备就绪，剩下的工作就是把处理器的控制权交给它。交接工作很简单，代码清单 8-1 第 76 行，加载器通过一个 16 位的间接绝对远转移指令，跳转到用户程序入口点。

如图 8-15 所示，入口点是两个连续的字，低字是偏移地址，位于用户程序头部内偏移为 0x04 的地方；高字是段地址，位于用户程序头部内偏移为 0x06 的地方。而且，因为加载器的辛勤工作，该段地址是已经重定位过的。

处理器执行指令

```
jmp far [0x04]
```

时，会访问段寄存器 DS 所指向的数据段，从偏移地址为 0x04 的地方取出两个字，并分别传送到代码段寄存器 CS 和指令指针寄存器 IP，以替代它们原先的内容。于是，处理器就像被洗脑了一样，自行转移到指定的位置处开始执行。

处理器已经跑到用户程序内部去执行了，所以接下来的工作是跟踪用户程序的工作流程。不过，在此之前，还是先总结一下无条件转移指令 `jmp` 的用法。

8.3.10 8086 处理器的无条件转移指令

(1) 相对短转移

相对短转移的操作码为 0xEB，操作数是相对于目标位置的偏移量，仅 1 字节，是个有符号数。

由于这个原因，该指令属于段内转移指令，而且只允许转移到距离当前指令-128~127字节的地方。相对短转移指令必须使用关键字“short”。例如：

```
jmp short infinite
```

在源程序编译阶段，编译器会检查标号 infinite 所代表的值，如果数值超过了一字节所能允许的数值范围，则无法通过编译。否则，编译器用目标位置的汇编地址减去当前指令的汇编地址，再减去当前指令的长度（2），保留1字节的结果，作为机器指令的操作数。

相对短转移指令的汇编语言操作数只能是标号和数值。下面是直接使用数值的情况：

```
jmp short 0x2000
```

但数值和标号是等价的。在编译阶段，都被用来计算一个8位的偏移量。

在指令执行时，处理器把指令中的操作数加上2，再加到指令指针寄存器IP上，这会导致指令的执行流程转向目标地址处。

（2）16位相对近转移

和相对短转移不同，16位相对近转移指令的转移范围稍大一些。它的机器指令操作码为0xE9，而且，该指令的长度为3字节，操作码0xE9后面还有一个16位（2字节）的操作数。

因为是近转移，故其属于段内转移。“相对”的意思同样是指它的操作数是一个相对量，是相对于目标位置处的偏移量。在源程序编译阶段，编译器用目标位置的汇编地址减去当前指令的汇编地址，再减去当前指令的长度（3），保留16位的结果，作为机器指令的操作数。由于这是一个16位的有符号数，故可以转移到距离当前指令-32768~32767字节的地方。

16位相对近转移指令应当使用关键字“near”，比如

```
jmp near infinite
jmp near 0x3000
```

在早先的NASM版本中，关键字near是可以省略的。若没有指定short或者near，那么，编译器自动默认是“near”的。但是最近的版本改变了这一规则。如果没有指定关键字short或者near，那么，如果目标位置距离当前指令-128~127字节，则自动采用short；否则，采用near。

（3）16位间接绝对近转移

这种转移方式也是近转移，即只在段内转移。但是，转移到的目标偏移地址不是在指令中直接给出的，而是用一个16位的通用寄存器或者内存地址来间接给出的。比如：

```
jmp near bx
jmp near cx
```

指令中的关键字“near”可以省略，间接绝对近转移原本就是near的。以上两条指令执行时，处理器将用寄存器BX或者CX的内容来取代指令指针寄存器IP的当前内容。

以上是目标偏移地址位于通用寄存器的情况。当然，该偏移地址也可位于内存中，而且这是最常见的情况。假如在某程序的数据段中声明了标号jump_dest并初始化了一个字：

```
jump_dest dw 0xc000
```

而且假定我们已经知道它是转移目标的起始偏移地址，那么，在该程序的代码段内，就可以使用以下的16位间接绝对近转移指令：

```
jmp [jump_dest] ;省略关键字“near”，本小节内下同
```

当这条指令执行时，处理器访问由段寄存器DS指向的数据段，从指令中指定的偏移地址处取得一个字（在这里是0xc000），并用该字取代指令指针寄存器IP的当前内容。

当然，既然是间接地寻找目标位置的偏移地址，其他寻址方式也是可以的。比如：

```
jmp [bx]
```

```
jmp [bx+si]
```

注意, `jmp bx` 和 `jmp [bx]` 是完全不同的, 不要犯迷糊。前者, 要转移的绝对偏移地址位于寄存器 BX 中; 后者, 偏移地址位于由 BX 所指向的内存字单元中。

(4) 16位直接绝对远转移

很早以前, 我们曾经见过这样的指令:

```
jmp 0x0000:0x7c00
```

在这里, `0x0000` 和 `0x7c00` 分别是段地址和偏移地址, 符合“段地址: 偏移地址”的表达习惯。在编译之后, 其机器指令为

```
EA 00 7C 00 00
```

`0xEA` 是操作码, 后面是操作数。注意, 字的存放是按照低端字节序的。而且, 在编译之后, 偏移地址在前, 段地址在后。执行这条指令后, 处理器用指令中给出的段地址代替段寄存器 CS 的原有内容, 用给出的偏移地址代替 IP 寄存器的原有内容, 从而跳转到另一个不同的代码段中, 即执行一个段间转移。

像这种直接在指令中给出段地址和偏移地址的转移指令, 就是直接绝对远转移指令。“16位”仅仅用来限定偏移地址部分, 指偏移地址是 16 位的。

(5) 16位间接绝对远转移 (`jmp far`)

远转移的目标地址可以通过访问内存来间接得到, 这叫间接远转移, 但是要使用关键字“`far`”。假如在某程序的数据段内声明了标号 `jump_far`, 并在其后初始化了两个字:

```
jmp_far dw 0x33c0,0xf000
```

这不是两个普通的数值, 它们分别是某个程序片断的偏移地址和段地址。为了转移到该程序片断上执行, 可以在使用下面的转移指令:

```
jmp far [jump_far]
```

关键字“`far`”的作用是告诉编译器, 该指令应当编译成一个远转移。处理器执行这条指令后, 访问段寄存器 DS 所指向的数据段, 从指令中给出的偏移地址处取出两个字, 分别用来替代段寄存器 CS 和指令指针寄存器 IP 的内容。

其实, 最好的例子还是本章代码清单 8-1 的第 76 行:

```
jmp far [0x04]
```

16 位间接绝对远转移指令的操作数可以是任何一种内存寻址方式。除了上面的例子外, 下面再给出几个:

```
jmp far [bx]
```

```
jmp far [bx+si]
```

最后, “16位”的意思是, 要转移到的目标位置的偏移地址是 16 位的。

8.4 用户程序的工作流程

8.4.1 初始化段寄存器和堆栈切换

现在轮到用户程序在处理器上执行了。

用户程序的入口点在代码清单 8-2 的第 135 行。因为加载器已经完成了重定位工作, 所以用户

程序的头等大事是初始化处理器的各个段寄存器 DS、ES、SS，以便访问专属于自己的数据。段寄存器 CS 就不用初始化了，那是加载器负责做的事。要不然用户程序怎么可能执行呢。

在刚刚进入用户程序时，段寄存器 DS 和 ES 依然指向段 header，而堆栈段寄存器 SS 依然指向加载器的堆栈空间。代码清单 8-2 的第 137、138 行，用于从头部取得用户程序自己的堆栈段的段地址，并传送到段寄存器 SS 中。

第 139 行，将标号 stack_end 所代表的数值传送到堆栈指针寄存器 SP。该标号是在第 205 行声明的，在它的前面，是伪指令 resb，用来保留 256 字节的堆栈空间。

伪指令 resb (REServe Byte) 的意思是从当前位置开始，保留指定数量的字节，但不初始化它们的值。在源程序编译时，编译器会保留一段内存区域，用来存放编译后的内容。当它看到这条伪指令时，它仅仅是跳过指定数量的字节，而不管里面的原始内容是什么。内存是反复使用的，谁也无法知道以前的使用者在这里留下了什么。也就是说，跳过的这段空间，每个字节的值是不确定的。

因此，

```
resb 256
```

将在编译后的内容中保留 256 字节。resb 不是唯一用来声明未初始化数据的指令。以下是另外一些：

resw 100	; 声明 100 个未初始化的字
resd 50	; 声明 50 个未初始化的双字

堆栈段 stack 的定义中有“vstart=0”子句，保留的 256 字节，其汇编地址分别是 0~255。所以，标号 stack_end 处的汇编地址实际上是 256。也就是说，代码清单 8-2 的第 139 行和以下指令等价：

```
mov sp,256
```

堆栈切换完毕之后，第 141、142 行，从用户程序头部取得数据段 data_1 的段地址，传送到段寄存器 DS 中。从此，DS 不再指向段 header，不能再用它访问用户程序头部了。

据此也可以看出，各个段寄存器的初始化顺序很重要。如果先初始化数据段和附加段，那么，段 header 中的数据将无法访问。

8.4.2 调用字符串显示例程

紧接着，用户程序要在屏幕上显示东西了。

要显示的内容位于段 data_1 中，该段当前正由段寄存器 DS 指向。代码清单 8-2 第 175 行，声明了标号 msg0 并初始化了一大堆字符。当然，因为字符太多，行太长，而我们还想能大致“看”到显示效果，所以分成了多行来初始化。

为太长的行使用续行符“\”当然是一个好主意，不过我们现在的做法是将太长的行分成几段，分别用伪指令 db 来初始化。在编译之后，它们仍然是紧挨在一起的，可以用唯一的标号 msg0 来引用。

在屏幕上显示字符，所做的仅仅是填充显存，只要所填充的内容不超过一屏所能显示的字符数，其他的事不需要你操心。当字符在一行上显示不下时，显示系统会自动移到下一行接着显示，这也和你无关。

不过，有时候我们希望有自行换行的能力，而不管那一行是否已经到头（屏幕最右边）。

这么做的目的通常是用来格式化文本段落。

再来说一下 ASCII 码。在 128 个 ASCII 代码中，大部分是可显示和打印的字符，还有一部分用于控制显示和打印那些字符的设备。比如 0x0d 是回车，0x0a 是换行。

回车和换行的概念最早起源于老式打字机。那种打字机上有滚筒，用于使纸张上下卷动，每敲击一个按键，字车往右移动一格，位于下一个可打印的位置。在这种古老而不失先进性的设备上，将字车推到最左边，也就是一行的开始，叫做回车（Carriage Return）；而拧一下滚筒，将纸上卷一行，叫做换行（Line Feed）。如果既回车，又换行，那么，字车将位于下一行的行首。这个过程通常叫做回车换行（CRLF）。

在刚刚有了电子计算机的时候，因为它又大又贵，只能通过远程终端来分享它的计算能力。这时候，用的是电传打字机，不需要人工操作即可显示和打印字符。当然，根据需要随时回车换行还是需要的。怎么办？那就是用 ASCII 码中的控制字符来命令电传打字机来做这件事。不知怎么回事，回车分配的 ASCII 码是 0x0d，换行分配的则是 0x0a。奇怪吗？没什么好奇怪的。

在个人计算机时代，为了在屏幕上显示字符，ASCII 码也被引入显示系统。不过，当我们向显存里写入 0x0d 和 0x0a 时，并不起任何作用，也没有任何效果，没有任何硬件对解释它们的意义负责。不过无所谓，对回车换行代码的解释可以由我们自己负责，现在所要做的，就是在字符串中，需要回车换行的地方按照老传统插入这两个代码。

正是由于以上的原因，在代码清单 8-2 的第 175~191 行，凡是需要回车换行的地方，都使用了 0x0d 和 0x0a。而且，在第 191 行，也就是所有要显示内容最后，是数值 0，用来标志字符串的结束，这样的字符串称为是 0 终止的字符串，在高级语言里经常使用。

段 data_1 的定义中包括“vstart=0”子句，故标号 msg0 的汇编地址是从该段的起始处（0）开始计算的。代码清单 8-2 的第 144、145 行，将该字符串的偏移地址传送到基址寄存器 BX，并调用过程 put_string。

8.4.3 过程的嵌套

过程 put_string 是在当前代码段定义的，位于代码清单 8-2 的第 28 行，用于显示给定的字符串。它接受两个参数 DS 和 BX，分别是字符串所在的段地址和偏移地址。另外，它要求字符串的最后一个数值是 0，作为终止的标记。

过程 put_string 的工作很简单，它循环从 DS:BX 中取得单个字符，判断它是否为 0。不为 0 则调用另一个过程 put_char，为 0 则返回主程序。

为此，代码清单 8-2 第 30 行，从当前数据段中取得一个字符，段地址在 DS 中，偏移地址由 BX 提供。

第 31 行，通过 or 指令来促成标志的产生，它的功能类似于

```
cmp cl,0
```

在这里，or 指令的两个操作数相同，都是寄存器 CL，一个数和它自己做“或”运算，结果还是它自己，但计算结果会影响标志寄存器中的某些位。如果 ZF 置位，说明取到了串结束标志 0，转移到第 38 行返回主程序；否则，将取到的字符作为参数调用另一个过程 put_char。

当过程 put_char 返回后，第 34 行，将寄存器 BX 的内容加一以指向下一个要显示的字符。

第 35 行，无条件转移到当前过程的开始处，重复取字符过程。

允许在一个过程中调用另一个过程，这称为过程嵌套。因为每次调用过程时，处理器都把

返回地址压在栈中，返回时从栈中取得返回地址，所以，只要堆栈是安全的，嵌套的过程都能层层返回。

过程嵌套的层数在原则上是没有限制的，唯一的限制是堆栈的大小。不要忘了，实模式下，堆栈的空间最大是 64KB，每执行一次过程调用需要 2 字节或 4 字节，这还没有包括在每个过程内部消耗的堆栈空间。

8.4.4 屏幕光标控制

过程 `put_char` 用于显示一个字符。但它与常规方法的不同之处在于，它能判断回车和换行，还能在超过屏幕上最后一行的时候上滚内容，就是我们经常说的卷屏或者滚屏。除此之外，它还使用了光标跟随技术。

光标（Cursor）是在屏幕上规律地闪动的一条小横线，通常用于指示下一个要显示的字符位置，这对很多年龄比较大的人来说很熟悉（前提是他们以前也用过计算机）。在那个时代，还没有基于图形显示技术的 Windows，所有的软件都在文本模式下工作，而基于硬件的光标只在文本模式下才会出现。

计算机技术发展得很快，很多硬件都已经或者即将淘汰，但显卡是个例外。即使是现在，多年前形成的 VGA 显示标准在每块显卡中都完好地保留下来了，包括对光标的 support。原因很简单，在显卡中集成一块支持 128 个 ASCII 代码的字符发生器非常方便，在程序中显示一个字符也只要给出它的 ASCII 码。显示图形的代价太大，在计算机加电启动的时候，以及其他一些根本没必要、也没条件使用图形模式的场合，这是最好的选择。

光标在屏幕上的位置保存在显卡内部的两个光标寄存器中，每个寄存器是 8 位的，合起来形成一个 16 位的数值。比如，0 表示光标在屏幕上第 0 行第 0 列，80 表示它在第 1 行第 0 列，因为标准 VGA 文本模式是 25 行，每行 80 个字符。这样算来，当光标在屏幕右下角时，该值为 $25 \times 80 - 1 = 1999$ 。

光标寄存器是可读可写的。你可以从中读出光标的位置，也可以通过它设置光标的位置。能够通过写入一个数值来设定光标的位置，这不是恩赐，而是责任，因为显卡从来不自动移动光标位置，这个任务是你的。现在你总算明白为什么它是可写的了吧？

8.4.5 取当前光标位置

显卡的操作非常复杂，内部的寄存器也不是一般地多。为了不过多占用主机的 I/O 空间，很多寄存器只能通过索引寄存器间接访问。

索引寄存器的端口号是 0x3d4，可以向它写入一个值，用来指定内部的某个寄存器。比如，两个 8 位的光标寄存器，其索引值分别是 14（0x0e）和 15（0x0f），分别用于提供光标位置的高 8 位和低 8 位。

指定了寄存器之后，要对它进行读写，这可以通过数据端口 0x3d5 来进行。

好，现在言归正传。过程 `put_char` 看起来并不太复杂，但实际上判断和分支较多。为了便于读者理解这段代码，也为了方便讲解，图 8-19 给出了它的工作流程图。

庞大复杂的建筑必须得有图纸才能施工，而绘制图纸的过程中你经常发现自己有更好的设计思路，更能知道如何盖这栋房子。编写复杂的程序前先画一画流程图，是程序员的基本素养，这有助于问题的解决。

代码清单 8-2 第 43~48 行，在过程 `put_char` 的开始部分先将用到的部分寄存器压栈保存，

其中包括两个段寄存器 DS 和 ES。

第 51~53 行，通过索引端口告诉显卡，现在要操作 0x0e 号寄存器。

第 54~56 行，通过数据端口从 0x0e 号端口读出 1 字节的数据，并传送到寄存器 AH 中，这是屏幕光标位置的高 8 位。

同样地，第 58~62 行，从 0x0f 号寄存器读出光标位置的低 8 位。现在，寄存器 AX 中是完整的光标位置数据。第 63 行，将这个数值传送到寄存器 BX 中保存，因为马上就要用到寄存器 AX。

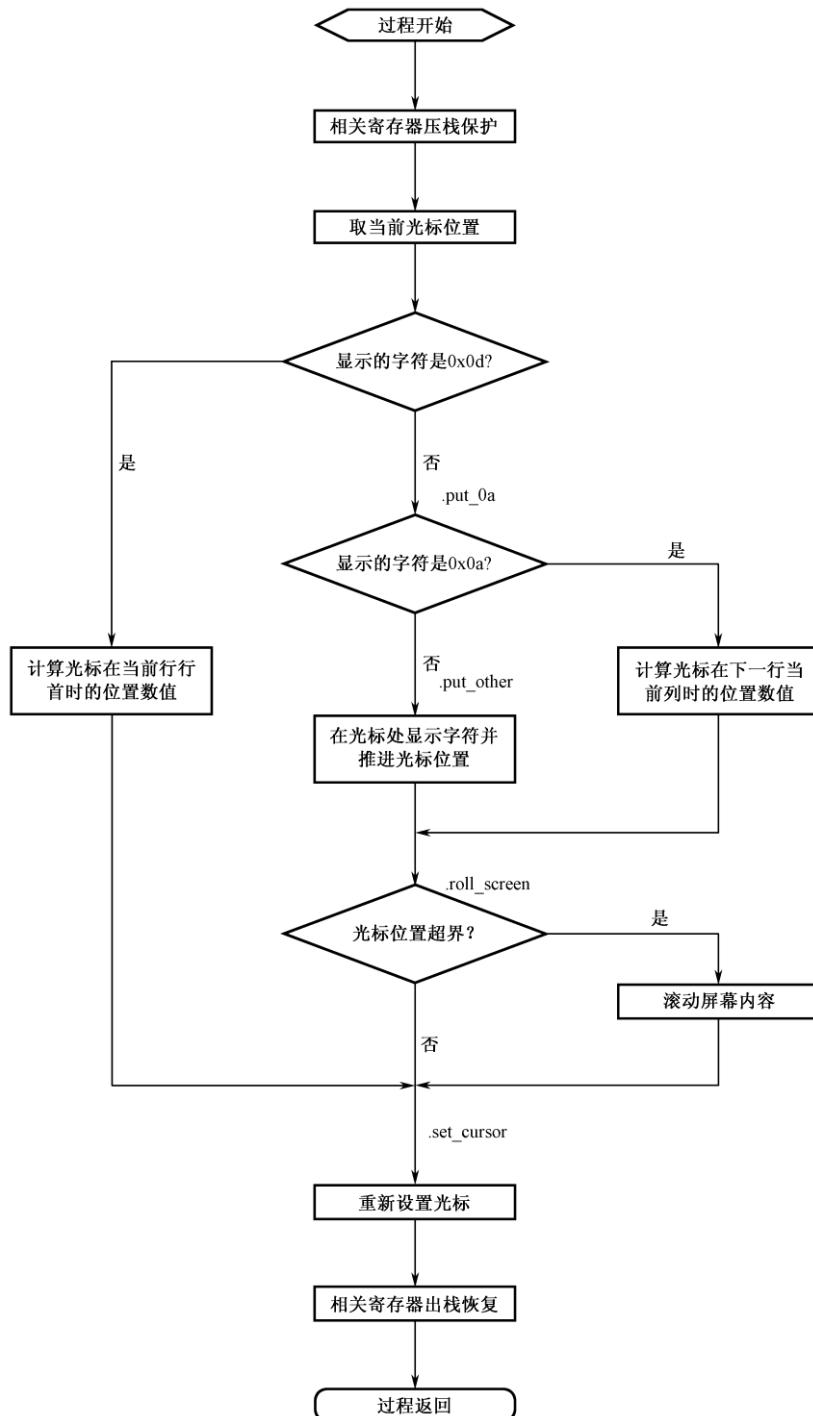


图 8-19 过程 put_char 的流程图

8.4.6 处理回车和换行字符

过程 put_char 仅接受一个寄存器参数 CL，用于提供要显示的 ASCII 码。常规字符和回车、换

行符将不同对待，为此，需要首先别出它们。

代码清单 8-2 第 65、66 行，先判断是不是回车符 0x0d。如果是的话，继续往下执行，如果不是，则转移到标号.put_0a 处执行。

先来看看如果是 0x0d 的情况。

如果是回车符 0x0d，那么，应将光标移动到当前行的行首。每行有 80 个字符，那么，用当前光标位置除以 80，余数不要，就可以得到当前行的行号。接着，再乘以 80，就是当前行行首的光标数值。

很好，代码清单 8-2 第 67~69 行，用寄存器 AX 中的光标位置除以寄存器 BL 中的 80，在 AL 中得到的是当前行的行号。

接着，第 70、71 行，将寄存器 AL 中的内容乘以寄存器 BL 中的 80，会在寄存器 AX 中得到当前行行首的光标值。该值依然传送到寄存器 BX 中保存。

和 div 指令相反，mul 是乘法指令，格式如下：

```
mul r/m8           ;AX=L×r/m8
mul r/m16          ;DX:AX=AX×r/m16
```

以上，“r”表示通用寄存器，“m”表示内存单元。就是说，mul 指令可以用 8 位的通用寄存器或者内存单元中的数和寄存器 AL 中的内容相乘，结果是 16 位，在 AX 寄存器中；也可以用 16 位的通用寄存器或者内存单元中的数和寄存器 AX 中的内容相乘，结果是 32 位，高 16 位和低 16 位分别在 DX 和 AX 中。

举几个例子：

```
mul bx
mul dx
mul byte [bx]      ;8 位内存单元
mul byte [bx+di]    ;8 位内存单元
mul word [0x2000]    ;16 位内存单元
```

mul 指令执行后，要是结果的高一半为全 0，则 OF 和 CF 清零，否则置 1。对 SF、ZF、AF 和 PF 标志的影响未定义。

第 72 行，转移到标号.set_cursor 处设置光标在屏幕上的位置。

如果要显示的字符不是 0x0d，那么，它有可能是 0x0a，或者是正常的可打印字符。这里的“打印”，可以理解为在屏幕上打印。

为此，第 75~77 行，先判断是不是 0x0a，如果不是，那就转移到标号.put_other 处，去正常显示可打印字符。如果是，那么，换行的意图是向下挪一行，只需要将寄存器 BX 的内容增加 80，即可得到新的光标位置数据。但是，不像回车，如果光标原先就在屏幕最后一行，那么，换行之后，会怎样呢？所以，第 78 行，立即转移到标号.roll_screen 处执行。在那里，将根据情况决定是否需要滚屏。

8.4.7 显示可打印字符

下面开始正常显示可打印字符。

第 81、82 行，将附加段寄存器 ES 设置为指向显存。注意，在过程开始处，已经将 ES 的内容压栈保存了，这里可以随意使用该寄存器。

标准模式下，屏幕上可以同时显示 2000 个字符。光标占用一个字符的位置，但整个屏幕只有

一个，只能出现在 2000 个字符位置中的一个上。典型地，程序员要用光标位置来记载和跟踪下一个字符应当显示在什么位置。光标用来指示字符位置，而一个字符在显存中对应两个字节。如此一来，可以将光标位置乘以 2，来得到该位置（字符）在显存中的偏移地址。

第 83 行，将寄存器 BX 的内容逻辑左移 1 次，这相当于将其乘以 2。毕竟只是乘以 2，而且 BX 中的数值不大，这样做，比使用乘法指令 mul 来得方便。

第 84 行，用 BX 的内容做为偏移地址，来访问段寄存器 ES 所指向的显存，来写入要显示的字符。你可能觉得奇怪，为什么后面没有写显示属性字节。原因很简单，在写入其他内容之前，显存里全是黑底白字的空白字符，所以不需要重写黑底白字的属性。过程 put_char 是以黑底白字来显示字符的。

第 87、88 行，将寄存器 BX 的内容除以 2，恢复它的光标位置身份。接着，将其增加 1（在数值上，将光标推进到下一个位置，毕竟还没开始设置光标呢）。指令 shr 是已经讲过的逻辑右移指令，相当于除以 2。

不管是换行，还是正常显示字符后推进光标，都会使寄存器 BX 的内容超过 1999。下面，就来判断这个情况，并决定是否滚动屏幕内容。

8.4.8 滚动屏幕内容

第 91、92 行，比较寄存器 BX 中的内容是否小于 2000。如果是的话，很好，很正常，直接转移到标号.set_cursor 处设置光标；否则继续往下执行以滚动屏幕内容。

滚动屏幕内容，实质上就是将屏幕上第 2~25 行的内容整体往上提一行，最后用黑底白字的空白字符填充第 25 行，使这一行什么也不显示。

为了加快速度，提高效率，程序里采用的是将数据从一个内存区域（块）搬运到另一个内存区域（块）的做法，核心指令是 movsw。

第 94~101 行，设定源区域从显存内偏移地址为 0xa0（屏幕第 2 行第 1 列的位置）的地方开始，该区域的段地址在段寄存器 DS 中，偏移地址在变址寄存器 SI 中；目标区域从显存内偏移地址为 0x00（屏幕第 1 行第 1 列的位置）的地方开始，该区域的段地址在段寄存器 ES 中，偏移地址在变址寄存器 DI 中。同时，设置方向标志，并在寄存器 CX 中设置要传送的字数 1920

（24 行乘以 80 个字符/行，再乘以每个字符占用的字节数 2，再除以 2 字节/字）。最后，执行 rep movsw 以完成传送工作。

屏幕最下面一行（第 25 行）还有原来的内容，必须予以清除。第 25 行第 1 列在显存中的偏移地址是 3840。为此，第 102~107 行，使用黑底白字的空白字符循环写入这一行。

最后，第 109 行，滚屏之后，光标应当位于最后一行的第 1 列，其数值为 1920，这一行的指令将这个新的数值传送到寄存器 BX 中。

8.4.9 重置光标

不管是回车、换行，还是显示可打印的字符，上面的各处都给出了光标位置的新数值。下面的工作就是按给出的数值在屏幕上设置光标。

第 112~123 行，还是依照老规矩，通过索引端口指定光标寄存器 0x0e 和 0x0f，并分别将寄存器 BX 中的高 8 位和低 8 位通过数据段口 0x3d5 写入它们。

最后，第 125~130 行，从堆栈中依次弹出并恢复各个寄存器的原始内容。

第 132 行，指令 `ret` 从堆栈中恢复指令指针寄存器 IP 的内容，返回到调用者 `put_string` 过程。当字符串 `msg0` 中所有的字符都显示完毕后，过程 `put_string` 返回到用户主程序，从第 147 行接着往下执行。

8.4.10 切换到另一个代码段中执行

在一个程序中，对段的数量没有限制。可以有多个代码段和多个数据段，甚至可以有多个堆栈段。在用户程序工作时，可以从一个代码段转到另一个代码段中执行，也可以根据需要，访问不同的数据段。

我们知道，ret 和 retf 指令分别用于近返回和远返回。人类最大的问题就是思维有定势，有时候不够开阔。尽管说是“返回”，但最重要的还是弄清它的原理和本质，才能灵活运用。

返回指令的动作是从堆栈中弹出内容到指令指针寄存器 IP，如果是远返回的话，还要接着弹出内容到代码段寄存器 CS。假如在此之前，栈顶的内容并非是用于返回的偏移地址和段地址，那么处理器当时就会傻了。

还是回到正题上来。假如要想切换到另一个代码段中执行，可以使用远调用指令（call far）或者远转移指令（jmp far），这是最正常不过的途径了。

问题在于，为了实现段间控制转移，必须事先开辟两个连续的内存单元，存放另一个代码段的入口点偏移地址和段地址，代价似乎有点高，这么做好像不太值得。

为了省事，可以使用指令 retf 来模拟段间返回，以实现段间转移。代码清单 8-2 第 147 行，先在堆栈中压入代码段 code_2 的段地址；接着，第 148、149 行，压入偏移地址，该偏移地址就是标号 begin 在编译阶段的汇编地址。8086 处理器不能在堆栈中压入立即数，所以只能通过寄存器 AX 来间接做这件事，现在的处理器都支持压入立即数：

```
push 0x55ff
```

当然，这是后话。

第 151 行，当处理器执行指令 retf 时，这个被蒙在鼓里的家伙从堆栈中将偏移地址和段地址分别弹出到代码段寄存器 CS 和指令指针寄存器 IP，于是控制立即转移到段 code_2 中，从标号 begin 处开始执行。

这段代码很好地证明了，尽管 call 和 call far 指令分别依赖于 ret 和 retf 指令，但后者却并不依赖于前者。它们经常在一起，但并不是夫妻。

8.4.11 访问另一个数据段

你可以在代码段 code_2 中做任何事。但是，我们这里什么也没干，仅仅是用相同的方法，再次返回到段 code_1 中。具体的做法，可以参考代码清单 8-2 第 166~170 行。

回到第 154、155 行，由于自从进入用户程序之后，段寄存器 ES 一直是指向头部段 header 的，所以，这两条指令用于将第二个数据段 data_2 的段地址传送到段寄存器 DS，这等于是换了一个数据段。

第二个数据段 data_2 是在第 194 行定义的，而且包含了“vstart=0”子句。在该段内，仅仅声明了标号 msg1 并初始化了一个字符串。当然，它也是 0 结尾的。

接着回到前面的第 157 行，将刚才那个字符串的起始偏移地址传送到寄存器 BX。第 158 行，调用过程 put_string 从屏幕的光标处开始显示该字符串。

8.5 编译和运行程序并观察结果

通常，用户程序执行完毕后，应当重新将控制返回到加载器，加载器可以重新加载和运行其他

程序，所有的操作系统都是这么做的。

遗憾的是，我们的加载器不提供这样的功能，而用户程序也没有将控制返回到加载器，而是直接进入无限循环：

```
jmp $
```

当然，这不是什么了不得的事情，将控制返回到加载器，其实现也不复杂。如果你有兴趣，可以试一试。但是，唯一麻烦的地方是堆栈，将控制返回的同时，也必须切换到加载器自己的堆栈，一定要小心！

对本章源代码的讲解到此结束。你可以在配书工具中找到源代码 c08_mbr.asm 和 c08.asm，或者自己手工编辑这两个文件。

首先编译源程序 c08_mbr.asm，将编译后得到的 c08_mbr.bin 文件写入虚拟硬盘主引导扇区（逻辑 0 扇区）。然后，编译源程序 c08.asm，并将生成的 c08.bin 文件写入虚拟硬盘的逻辑 100 扇区。

注意，在编译 c08.asm 时，编译器将会产生警告信息：

```
c08.asm:203: warning: uninitialized space declared in stack section: zeroing
```

这句话的意思是，c08.asm 源程序的第 203 行声明了未初始化的空间。

还记得吗？在那里，我们用 resb 伪指令保留了 256 字节的堆栈空间，这段空间是未初始化的。

源程序的编译过程也是排错过程，你该感到高兴，而不是害怕，只有合乎规范的程序才能最终获得通过。编译器通常会有两种提示，一种是错误，另一种是警告。

错误（Error）表明程序中有编译器不认识的指令、不正确的语法和无法解释的内容，在这种情况下，编译器简单地告诉你是哪一行有错误，以及什么性质的错误，并停止编译。

警告（Warning）通常表示程序中有一些不规范的指令用法。在这种情况下，编译器继续完成编译工作，生成编译结果。通常情况下，编译后的结果也能正常运行。

现在，启动虚拟机，正常情况下，运行结果应当如图 8-20 所示。

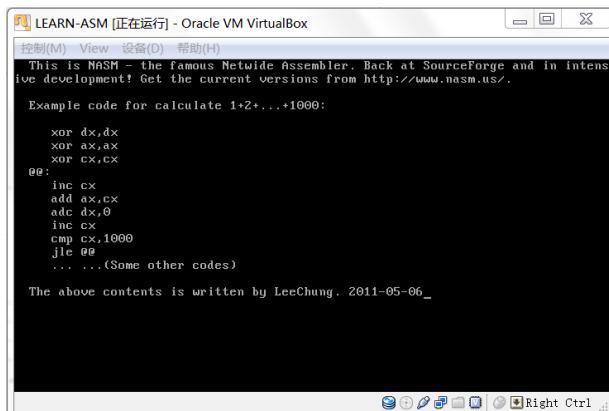


图 8-20 本章程序运行结果

本章习题

- 修改本章源程序 8-2，在不使用 retf 指令的情况下，从段 code_1 转移到段 code_2 执行。
- 思考一下，如果去掉代码清单 8-1 的第 38、39 行，会发生什么情况？

第9章 中断和动态时钟显示

在享受计算机给我们带来的便利和乐趣的同时，我仍然会时不时地说它的坏话。人们都说处理器是整个计算机的大脑，可是，处理器是一个非常精确，速度又快的傻子。

在计算机上执行的程序通常需要一些输入，输入可能来自于键盘、鼠标、硬盘、话筒、数码相机等，同时，处理后还需要输出，要送到输出设备，如显示器、硬盘、打印机、网络设备等。

一个程序只做自己的事，当它等待输入，或者等待输出时，它面对的是比处理器慢得多的外部设备。典型的情况下，硬盘的工作速度比处理器至少慢几千万甚至几亿倍，像打印机这类设备就更不用说了。在等待的时候，处理器唯一所能做的，就是不停地观察外部设备的状态变化。

计算机革命的早期，硬件资源极其昂贵和稀少。据说 20 世纪 60 年代，一台计算机的价格抵得上 300 辆野马跑车，月租金超过一万美金。这么昂贵的东西，不好好利用它就是一种罪过。

为了分享计算能力，处理器应当能够为多用户多任务提供硬件一级的支持。在单处理器的系统中，允许同时有多个程序在内存中等待处理器的执行。当一个程序正在等待输入输出时，允许另一个程序从处理器那里得到执行权。

如何把多个程序调入内存，是操作系统的事情，这个可以先放一放。现在的问题是，当一个程序执行时，它是不会知道还有别的程序正眼巴巴地等着执行。在这种情况下，中断（Interrupt）这种工作机制就应运而生了。

中断就是打断处理器当前的执行流程，去执行另外一些和当前工作不相干的指令，执行完之后，还可以返回到原来的程序流程继续执行。这就好比是你正在用手机听歌，突然来电话了。处理器（当然，手机也是有处理器的）必须中断歌曲的播放，来处理这件更为重要的事件。

自从中断这种工作机制产生之后，它就一直是各种处理器必须具备的机制。中断是怎么发生的，处理器又是怎么处理中断的，在这个过程中，我们又能做些什么，这都是本章将要告诉你的。

9.1 外部硬件中断

顾名思义，外部硬件中断，就是从处理器外面来的中断信号。当外部设备发生错误，或者有数据要传送（比如，从网络中接收到一个针对当前主机的数据包），或者处理器交给它的事情处理完了（比如，打印已经完成），它们都会拍一下处理器的肩膀，告诉它应当先把手头上的事情放一放，来临时处理一下。

如图 9-1 所示，外部硬件中断是通过两个信号线引入处理器内部的。从很早的时候起，也

就是 8086 处理器的时代，这两根线的名字就叫 NMI 和 INTR。

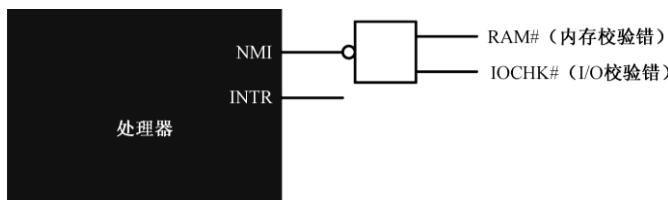


图 9-1 Intel 处理器上的不可屏蔽中断示意图

9.1.1 非屏蔽中断

在某些具有怀疑精神的人眼里，用两根信号线来接受外部设备中断可能是多余的，也许只需要一根就可以了。这似乎有此道理，但是，来自外部设备的中断很多，也不是每一个中断都是必须处理的。有些中断，在任何时候都必须及时处理，因为事关整个系统的安全性。比如，在使用不间断电源的系统中，当电池电量很低的时候，不间断电源系统会发出一个中断，通知处理器快掉电了。再比如，内存访问电路发现了一个校验错误，这意味着，从内存读取的数据是错误的，处理器再努力工作也是没有意义的。在所有这些情况下，处理器必须针对这些中断采取必要的措施，隐瞒真相必然会对用户造成不可挽回的损失。除此之外，更多的中断是可以被忽略或者延迟处理的，如果某个程序希望不被打扰的话。

在这种情况下，处理器的设计者希望通过两个引脚来明确区分不同性质的中断，这是很自然的事。首先，所有的严重事件都必须无条件地加以处理，这种类型的中断是不会被阻断和屏蔽的，称为非屏蔽中断（Non Maskable Interrupt，NMI）。

中断信号的来源，或者说，产生中断的设备，称为中断源。如图 9-1 所示，在传统的兼容模式下，NMI 的中断源通过一个与非门连接到处理器。处理器的 NMI 引脚是高电平有效的，而中断信号是低电平有效的。当不存在中断的时候，与非门的所有输入都为高，因此处理器的 NMI 引脚为低电平，这意味着没有中断发生。

当有任何一个非屏蔽的中断产生时，与非门的输出为高。Intel 处理器规定，NMI 中断信号由 0 跳变到 1 后，至少要维持 4 个以上的时钟周期才算是有效的，才能被识别。

注意，不要把这幅图当成是不变的真理，这是一个简化的示意图，不是真正的设备连接图。

当一个中断发生时，处理器将会通过中断引脚 NMI 和 INTR 得到通知。除此之外，它还应当知道发生了什么事，以便采取适当的处理措施。每种类型的中断都被统一编号，这称为中断类型号、中断向量或者中断号。但是，由于不可屏蔽中断的特殊性——几乎所有触发 NMI 的事件对处理器来说都是致命的，甚至是不可纠正的。在这种情况下，努力去搞清楚发生了什么，通常没有太大的意义，这样的事最好留到事后，让专业维修人员来做。

也正是这个原因，在实模式下，NMI 被赋予了统一的中断号 2，不再进行细分。一旦发生 2 号中断，处理器和软件系统通常会放弃继续正常工作的“念头”，也不会试图纠正已经发生的问题和错误，很可能只是由软件系统给出一个提示信息。

9.1.2 可屏蔽中断

和 NMI 不同，更多的时候，发往处理器的中断信号通常不会意味着灾难。当然，有时候也会非常紧急，比如，在一个由计算机控制的车床上，当零件快速通过铣具时，处理器应当立即处理中断，并向铣具发送信号，告诉它应当如何切削。

这类中断有两个特点，第一是数量很多，毕竟有很多外部设备；第二是它们可以被屏蔽，这样处理器就像是没听见、没看见一样，不会对它们进行处理。所以，这类硬件中断称为可屏蔽中断。尽管不处理中断就会把零件烧坏，但是否允许处理器看见该中断，是你自己的事，这是处理器赋予你的权利。

可屏蔽中断是通过 INTR 引脚进入处理器内部的，像 NMI 一样，不可能为每一个中断源都提供一个引脚。而且，处理器每次只能处理一个中断。在这种情况下，需要一个代理，来接受外部设备发出的中断信号。还有，多个设备同时发出中断请求的机率也是很高的，所以该代理的任务还包括对它们进行仲裁，以决定让它们中的哪一个优先向处理器提出服务请求。

如图 9-2 所示，在个人计算机中，用得最多的中断代理就是 8259 芯片，它就是通常所说的中断控制器，从 8086 处理器开始，它就一直提供着这种服务。即使是现在，在绝大多数单处理器的计算机中，也依然有它的存在。

Intel 处理器允许 256 个中断，中断号的范围是 0~255，8259 负责提供其中的 15 个，但中断号并不固定。之所以不固定，是因为当初设计的时候，允许软件根据自己的需要灵活设置中断号，以防止发生冲突。该中断控制器芯片有自己的端口号，可以像访问其他外部设备一样用 in 和 out 指令来改变它的状态，包括各引脚的中断号。正是因为这样，它又叫可编程中断控制器（Programmable Interrupt Controller，PIC）。

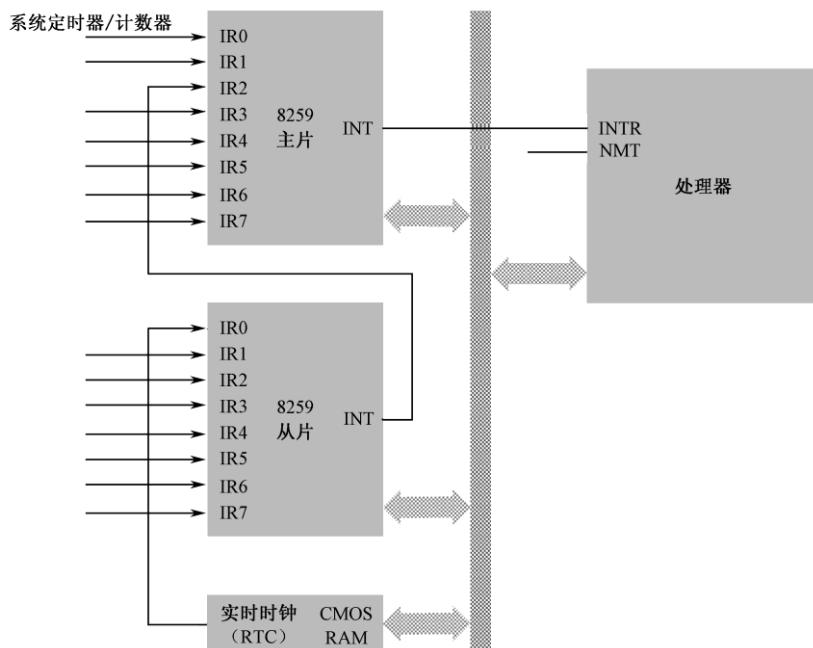


图 9-2 单处理器系统的中断机制

不知道是怎么想的，反正每片 8259 只有 8 个中断输入引脚，而在个人计算机上使用它，需要两块。如图 9-2 所示，第一块 8259 芯片的代理输出 INT 直接送到处理器的 INTR 引脚，这是主片（Master）；第二块 8259 芯片的 INT 输出送到第一块的引脚 2 上，是从片（Slave），两块芯片之间形成级联（Cascade）关系。

如此一来，两块 8259 芯片可以向处理器提供 15 个中断信号。当时，接在 8259 上的 15 个设备都是相当重要的，如 PS/2 键盘和鼠标、串行口、并行口、软磁盘驱动器、IDE 硬盘等。现在，这些设备很多都已淘汰或者正在淘汰中，根据需要，这些中断引脚可以被其他设备使用。

如图9-2所示，8259的主片引脚0（IR0）接的是系统定时器/计数器芯片；从片的引脚0（IR0）接的是实时时钟芯片RTC，该芯片是本章的主角，很快就会讲到。总之，这两块芯片的固定连接即使是在硬件更新换代非常频繁的今天，也依然没有改变。

在8259芯片内部，有中断屏蔽寄存器（Interrupt Mask Register, IMR），这是个8位寄存器，对应着该芯片的8个中断输入引脚，对应的位是0还是1，决定了从该引脚来的中断信号是否能够通过8259送往处理器（0表示允许，1表示阻断，这可能出乎你的意料）。当外部设备通过某个引脚送来一个中断请求信号时，如果它没有被IMR阻断，那么，它可以被送往处理器。注意，8259芯片是可编程的，主片的端口号是0x20和0x21，从片的端口号是0xa0和0xa1，可以通过这些端口访问8259芯片，设置它的工作方式，包括IMR的内容。

中断能否被处理，除了要看8259芯片的脸色外，最终的决定权在处理器手中。回到前面第6章，参阅图6-2，你会发现，在处理器内部，标志寄存器有一个标志位IF，这就是中断标志（Interrupt Flag）。当IF为0时，所有从处理器INTR引脚来的中断信号都被忽略掉；当其为1时，处理器可以接受和响应中断。

IF标志位可以通过两条指令cli和sti来改变。这两条指令都没有操作数，cli（CLear Interrupt flag）用于清除IF标志位，sti（SeT Interrupt flag）用于置位IF标志。

在计算机内部，中断发生得非常频繁，当一个中断正在处理时，其他中断也会陆续到来，甚至会有多个中断同时发生的情况，这都无法预料。不用担心，8259芯片会记住它们，并按一定的策略决定先为谁服务。总体上来说，中断的优先级和引脚是相关的，主片的IR0引脚优先级最高，IR7引脚最低，从片也是如此。当然，还要考虑到从片是级联在主片的IR2引脚上。

最后，当一个中断事件正在处理时，如果来了一个优先级更高的中断事件时，允许暂时中止当前的中断处理，先为优先级较高的中断事件服务，这称为中断嵌套。

9.1.3 实模式下的中断向量表

所谓中断处理，归根结底就是处理器要执行一段与该中断有关的程序（指令）。处理器可以识别256个中断，那么理论上就需要256段程序。这些程序的位置并不重要，重要的是，在实模式下，处理器要求将它们的入口点集中存放到内存中从物理地址0x00000开始，到0x003ff结束，共1KB的空间内，这就是所谓的中断向量表（Interrupt Vector Table, IVT）。

如图9-3所示，每个中断在中断向量表中占2个字，分别是中断处理程序的偏移地址和段地址。中断0的入口点位于物理地址0x00000处，也就是逻辑地址0x0000:0x0000；中断1的入口点位于物理地址0x00004处，即逻辑地址0x0000:0x0004；其他中断依次类推，总之是按顺序的。

当中断发生时，如果从外部硬件到处理器之间的道路都是畅通的，那么，处理器在执行完当前的指令后，会立即着手为硬件服务。它首先会响应中断，告诉8259芯片准备着手处理该中断。接着，它还会要求8259芯片把中断号送过来。

在8259芯片那里，每个引脚都赋予了一个中断号。而且，这些中断号是可以改变的，可以对8259编程来灵活设置，但不能单独进行，只能以芯片为单位进行。比如，可以指定主片的中断号从0x08开始，那么它每个引脚IR0~IR7所对应的中断号分别是0x08~0x0e。

中断信号来自哪个引脚，8259芯片是最清楚的，所以它会把对应的中断号告诉处理器，处理器拿着这个中断号，要顺序做以下几件事。

① 保护断点的现场。首先要将标志寄存器FLAGS压栈，然后清除它的IF位和TF位。TF是陷阱标志，这个以后再讲。接着，再将当前的代码段寄存器CS和指令指针寄存器IP压栈。

② 执行中断处理程序。由于处理器已经拿到了中断号，它将该号码乘以 4（毕竟每个中断在中断向量表中占 4 字节），就得到了该中断入口点在中断向量表中的偏移地址。接着，从表中依次取出中断程序的偏移地址和段地址，并分别传送到 IP 和 CS，自然地，处理器就开始执行中断处理程序了。

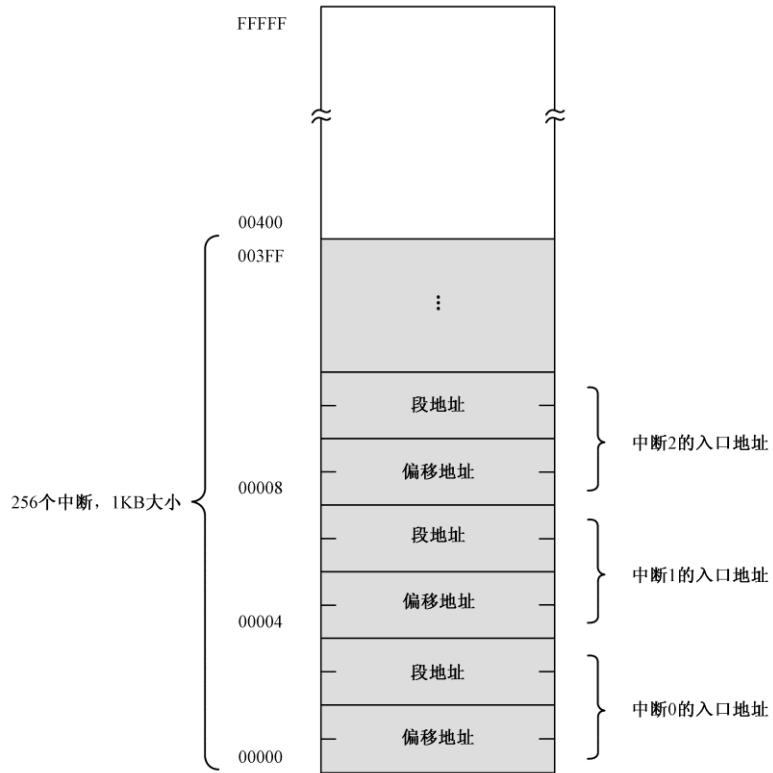


图 9-3 实模式下的中断向量表

注意，由于 IF 标志被清除，在中断处理过程中，处理器将不再响应硬件中断。如果希望更高优先级的中断嵌套，可以在编写中断处理程序时，适时用 sti 指令开放中断。

③ 返回到断点接着执行。所有中断处理程序的最后一条指令必须是中断返回指令 iret。这将导致处理器依次从堆栈中弹出（恢复）IP、CS 和 FLAGS 的原始内容，于是转到主程序接着执行。

iret 同样没有操作数，执行这条指令时，处理器依次从堆栈中弹出数值到 IP、CS 和标志寄存器。

顺便提醒一句，由于中断处理过程返回时，已经恢复了 FLAGS 的原始内容，所以 IF 标志位也自动恢复。也就是说，可以接受新的中断。

和可屏蔽中断不同，NMI 发生时，处理器不会从外部获得中断号，它自动生成中断号码 2，其他处理过程和可屏蔽中断相同。

中断随时可能发生，中断向量表的建立和初始化工作是由 BIOS 在计算机启动时负责完成的。BIOS 为每个中断号填写入口地址，因为它不知道多数中断处理程序的位置，所以，一律将它们指向一个相同的入口地址，在那里，只有一条指令：iret。也就是说，当这些中断发生时，只做一件事，那就是立即返回。当计算机启动后，操作系统和用户程序再根据自己的需要，来修改某些中断的入口地址，使它指向自己的代码。马上你就会看到，我们在本章也是这样做

的。

9.1.4 实时时钟、CMOS RAM 和 BCD 编码

也许你曾经觉得奇怪，为什么计算机能够准确地显示日期和时间？原因很简单，如图 9-2 所示，在外围设备控制器芯片 ICH 内部，集成了实时时钟电路（Real Time Clock, RTC）和两小块由互补金属氧化物（CMOS）材料组成的静态存储器（CMOS RAM）。实时时钟电路负责计时，而日期和时间的数值则存储在这块存储器中。

实时时钟是全天候跳动的，即使是在你关闭了计算机的电源之后，原因在于它由主板上的一个小电池提供能量。它为整台计算机提供一个基准时间，为所有需要时间的软件和硬件服务。不像 8259 芯片，有关 RTC CMOS 的资料相当少见，很不容易完整地找到，而 8259 的内容则铺天盖地，到处都是。所以，本章只是简要地介绍 8259，而尽量多说一些和 RTC 有关的知识。

早期的计算机没有 ICH 芯片，各个接口单元都是分立的，单独地焊在主板上，并彼此连接。早期的 RTC 芯片是摩托罗拉（Motorola）MS146818B，现在直接集成在 ICH 内，并且在信号上与其兼容。除了日期和时间的保存功能外，RTC 芯片也可以提供闹钟和周期性的中断功能。

日期和时间信息是保存在 CMOS RAM 中的，通常有 128 字节，而日期和时间信息只占了一小部分容量，其他空间则用于保存整机的配置信息，比如各种硬件的类型和工作参数、开机密码和辅助存储设备的启动顺序等。这些参数的修改通常在 BIOS SETUP 开机程序中进行。要进入该程序，一般需要在开机时按 DEL、ESC、F1、F2 或者 F10 键。具体按哪个键，视计算机的厂家和品牌而定。

RTC 芯片由一个振荡频率为 32.768kHz 的石英晶体振荡器（晶振）驱动，经分频后，用于对 CMOS RAM 进行每秒一次的时间刷新。

如表 9-1 所示，常规的日期和时间信息占据了 CMOS RAM 开始部分的 10 字节，有年、月、日和时、分、秒，报警的时、分、秒用于产生到时间报警中断，如果它们的内容为 0xC0~0xFF，则表示不使用报警功能。

表 9-1 CMOS RAM 中的时间信息

偏移地址	内 容	偏移地址	内 容
0x00	秒	0x07	日
0x01	闹钟秒	0x08	月
0x02	分	0x09	年
0x03	闹钟分	0x0A	寄存器 A
0x04	时	0x0B	寄存器 B
0x05	闹钟时	0x0C	寄存器 C
0x06	星期	0x0D	寄存器 D

CMOS RAM 的访问，需要通过两个端口来进行。0x70 或者 0x74 是索引端口，用来指定 CMOS RAM 内的单元；0x71 或者 0x75 是数据端口，用来读写相应单元里的内容。举个例子，以下代码用于读取今天是星期几：

```
mov al,0x06
out 0x70,al
in al,0x71
```

不得不说的是，从很早的时候开始，端口 0x70 的最高位（bit 7）是控制 NMI 中断的开关。当它

为 0 时，允许 NMI 中断到达处理器，为 1 时，则阻断所有的 NMI 信号，其他 7 个比特，即 0~6 位，则实际上用于指定 CMOS RAM 单元的索引号，这种规定直到现在也没有改变。

如图 9-4 所示，尽管端口 0x70 的位 7 不是中断信号，但它能控制与非门的输出，决定真正的 NMI 中断信号是否能到达处理器。

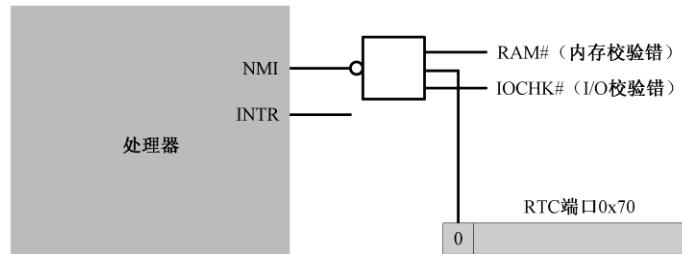


图 9-4 端口 0x70 的位 7 用于禁止或允许 NMI (仅为示意图)

通常来说，在往端口 0x70 写入索引时，应当先读取 0x70 原先的内容，然后将它用于随后的写索引操作中。但是，该端口是只写的，不能用于读出。在早期的系统中，计算机的制造成本很高，为了最大化地利用硬件资源，导致出现很多稀奇古怪的做法，这就是一个活生生的例子。

为了解决这个问题，同时也为了兼容以前的老式硬件，ICH 芯片允许通过切换访问模式来临时取得那些只写寄存器的内容，但这涉及更高层次的知识，已经超出了当前的话题范畴。现在，我们只想把问题搞得简单些，这么说吧，NMI 中断应当始终是允许的，在访问 RTC 时，我们直接关闭 NMI，访问结束后，再打开 NMI，而不管它以前到底是什么样子。

在早期，CMOS RAM 只有 64 字节，而最新的 ICH 芯片内则可能集成了 256 字节，新增的 128 字节称为扩展的 CMOS RAM。当然，在此之前，要先确保 ICH 内确实存在扩展的 CMOS RAM。

CMOS RAM 中保存的日期和时间，通常是以二进制编码的十进制数 (Binary Coded Decimal, BCD)，这是默认状态，如果需要，也可以设置成按正常的二进制数来表示。要想说明什么是 BCD 编码，最好的办法是举个例子。比如十进制数 25，其二进制形式是 00011001。但是，如果采用 BCD 编码的话，则一个字节的高 4 位和低 4 位分别独立地表示一个 0 到 9 之间的数字。因此，十进制数 25 对应的 BCD 编码是 00100101。由此可以看出，因为十进制数里只有 0~9，故用 BCD 编码的数，高 4 位和低 4 位都不允许大于 1001，否则就是无效的。

单元 0x0A~0x0D 不是普通的存储单元，而被定义成 4 个寄存器的索引号，也是通过 0x70 和 0x71 这两个端口访问的。这 4 个寄存器用于设置实时钟电路的参数和工作状态。

寄存器 A 和 B 用于对 RTC 的功能进行整体性的设置，它们都是 8 位的寄存器，可读可写，其各位的用途如表 9-2 和表 9-3 所示。

表 9-2 寄存器 A 各位功能说明

比 特 位	功 能
-------	-----

正处于更新过程中（Update In Progress, UIP）。该位可以作为一个状态进行监视。CMOS RAM 中的时间和日期信息会由 RTC 周期性地更新，在此期间，用户程序不应当访问它们。对当前寄存器的写入不会改变此位的状态。

0：更新周期至少在 488 微秒内不会启动。换句话说，此时访问 CMOS RAM 中的时间、日历和闹钟信息是安全的。

1：正处于更新周期，或者马上就要启动。

如果寄存器 B 的 SET 位不是 1，而且在分频电路已正确配置的情况下，更新周期每秒发生一次。在此期间，会增加保存的日期和时间、检查数据是否因超出范围而溢出（比如，31 号之后是下月 1 号，而不是 32 号），还要检查是否到了闹钟时间，最后，更新之后的数据还要写回原来的位置。

更新周期至少会在 UIP 置 1 后的 488 μ s 内开始，而且整个周期的完成时间不会多于 1984 μ s，在此期间，和日期时间有关的存储单元（0x00~0x09）会暂时脱离外部总线。为避免更新和数据遭到破坏，可以有两次安全地从外部访问这些单元的机会：当检测到更新结束中断发生时，可以有差不多 999ms 的时间用于读写有效的日期和时间数据；如果检测到寄存器 A 的 UIP 位为低（0），那么这意味着在更新周期开始前，至少还有 488 μ s 的时间。

续表

比特位	功能
6~4	分频电路选择 (Division Chain Select)。这 3 位控制晶体振荡器的分频电路。系统将其初始化到 010，为 RTC 选择一个 32.768kHz 的时钟频率
3~0	速率选择 (Rate Select, RS)。选择分频电路的分节点。如果寄存器 B 的 PIE 位被设置的话，此处的选择将产生一个周期性的中断信号，否则将设置寄存器 C 的 PF 标志位 0000: 从不触发中断 0001: 3.90625 ms 0010: 7.8125 ms 0011: 122.070 μs 0100: 244.141 μs 0101: 488.281 μs 0110: 976.5625 μs 0111: 1.953125 ms 1000: 3.90625 ms 1001: 7.8125 ms 1010: 5.625 ms 1011: 1.25 ms 1100 = 62.5 ms 1101: 125 ms 1110: 250 ms 1111: 500 ms

表 9-3 寄存器 B 各位功能说明

比特位	功能
7	更新周期禁止 (Update Cycle Inhibit, SET)。允许或者禁止更新周期 0: 更新周期每秒都会正常发生 1: 中止当前的更新周期，并且此后不再产生更新周期。此位置 1 时，BIOS 可以安全地初始化日历和时间
6	周期性中断允许 (Periodic Interrupt Enable, PIE) 0: 不允许 1: 当达到寄存器 A 中 RS 所设定的时间基准时，允许产生中断
5	闹钟中断允许 (Alarm Interrupt Enable, AIE) 0: 不允许 1: 允许更新周期在到达闹点并将 AF 置位的同时，发出一个中断
4	更新结束中断允许 (Update-Ended Interrupt Enable, UIE) 0: 不允许 1: 允许在每个更新周期结束时产生中断
3	方波允许 (Square Wave Enable, SQWE) 该位空着不用，只是为了和早期的 Motorola 146818B 实时时钟芯片保持一致
2	数据模式 (Data Mode, DM) 该位用于指定二进制或者 BCD 的数据表示形式 0: BCD 1: Binary
1	小时格式 (Hour Format, HOURFORM) 0: 12 小时制。在这种模式下，第 7 位为 0 表示上午 (AM)，为 1 表示下午 (PM) 1: 24 小时制
0	老软件的夏令时支持 (Daylight Savings Legacy Software Support, DSLSWS) 该功能已不再支持，该位仅用于维持对老软件的支持，并且是无用的

寄存器 C 和 D 是标志寄存器，这些标志反映了 RTC 的工作状态，寄存器 C 是只读的，寄存器 D 则可读可写，它们也都是 8 位寄存器，其各位的含义如表 9-4 和表 9-5 所示。特别是寄存器 C，

因为 RTC 可以产生中断，当中断产生时，可以通过该寄存器来识别中断的原因，比如，是周期性的中断，还是闹钟中断。

表 9-4 寄存器 C 各位功能说明

比特位	功 能
7	中断请求标志 (Interrupt Request Flag, IRQF) IRQF = (PF × PIE) + (AF × AIE) + (UF × UFE) 以上，加号表示逻辑或，乘号表示逻辑与。该位被设置时，表示肯定要发生中断。对寄存器 C 的读操作将导致此位清零
6	周期性中断标志 (Periodic Interrupt Flag, PF)。 若寄存器 A 的 RS 位为 0000，则此位是 0，否则是 1。对寄存器 C 的读操作将导致此位清零 注：程序可以根据此位来判断 RTC 的中断原因
5	闹钟标志 (Alarm Flag, AF)。 当所有闹点同当前时间相符时，此位是 1。对寄存器 C 的读操作将导致此位清零 注：程序可以根据此位来判断 RTC 的中断原因
4	更新结束标志 (Update-Ended Flag, UF) 紧接着每秒一次的更新周期之后，RTC 电路立即将此位置 1。对寄存器 C 的读操作将导致此位清零 注：程序可以根据此位来判断 RTC 的中断原因
3~0	保留，总是报告 0

表 9-5 寄存器 D 各位功能说明

比特位	功 能
7	有效 RAM 和时间位 (Valid RAM and Time Bit, VRT) 在写周期，此位应当始终写 0。不过，在读周期，此位回到 1。在 RTC 加电正常时，此位被硬件强制为 1
6	保留。总是返回 0。并且在写周期总是置 0
5~0	日期闹钟 (Date Alarm)，这些位保存着闹钟的月份数值

讲了这么多和 8259 以及 RTC 有关的内容，现在，我们想让 RTC 芯片定期发出一个中断，当这个中断发生的时候，还能执行我们自己编写的代码，来访问 CMOS RAM，在屏幕上显示一个动态走动的时钟。

9.1.5 代码清单 9-1

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：9-1（被加载的用户程序），源程序文件：c09_1.asm

9.1.6 初始化 8259、RTC 和中断向量表

本章提供的代码清单中，没有加载器程序。这是因为可以利用上一章提供的加载器来加载用户程序，只要符合规则，加载器是通用的。

用户程序的入口点在代码清单 9-1 的第 119 行，从这一行开始，到第 124 行，用于初始化各个段寄存器的内容。下面开始在中断向量表中安装实时时钟中断的入口点。既然本章的主题是中断，那么就很有必要强调一件事。当处理器执行任何一条改变堆栈段寄存器 SS 的指令时，它会在下一条指令执行完期间禁止中断。

堆栈无疑是很重要的，不能被破坏。要想改变代码段和数据段，只需要改变段寄存器就可以了。

但堆栈段不同，因为它除了有段寄存器，还有堆栈指针。因此，绝大多数时候，对堆栈的改变是分两步进行的：先改变段寄存器 SS 的内容，接着又修改堆栈指针寄存器 SP 的内容。

想象一下，如果刚刚修改了段寄存器 SS，在还没来得及修改 SP 的情况下，就发生了中断，会出现什么后果，而且要知道，中断是需要依靠堆栈来工作的。

因此，处理器在设计的时候就规定，当遇到修改段寄存器 SS 的指令时，在这条指令和下一条指令执行完毕期间，禁止中断，以此来保护堆栈。换句话说，你应该在修改段寄存器 SS 的指令之后，紧跟着一条修改堆栈指针 SP 的指令。

就代码清单 9-1 来说，在第 121、122 行执行期间，处理器禁止中断。再比如以下指令：

```
push cs
pop ss
mov sp,0
```

在后面两行指令执行期间，处理器禁止中断。

RTC 芯片的中断信号，通向中断控制器 8259 从片的第 1 个中断引脚 IR0。在计算机启动期间，BIOS 会初始化中断控制器，将主片的中断号设为从 0x08 开始，将从片的中断号设为从 0x70 开始。所以，计算机启动后，RTC 芯片的中断号默认是 0x70。尽管我们可以通过对 8259 编程来改变它，但是没有必要。

在安装中断向量之前，应该先显示些什么。第 126~130 行，显示两行提示信息，表明正在安装中断向量。这两个字符串位于第 286 行的数据段中。对于过程 put_string 没有什么好说的，它的代码和上一章相同，工作过程更没有区别。

为了修改某中断在中断向量表中的登记项，需要先找到它。第 132~135 行，将中断号 0x70 乘以 4，就是它在中断向量表内的偏移。

第 137 行，修改中断向量表时，需要先用 cli 指令清中断。当表项信息只修改了一部分时，如果发生 0x70 号中断，则会产生不可预料的问题。

第 139~141 行，将段寄存器 ES 压栈暂时保存，并使它指向中断向量表（所在的段）。

接着，第 142~145 行，访问中断向量表内 0x70 号中断的表项，分别写入新中断处理过程的偏移地址和段地址。新的中断处理过程是从标号 new_int_0x70 处开始的，而且位于当前代码段内。所以，该中断处理过程的偏移地址就是标号 new_int_0x70 的汇编地址（注意，段 code 的定义中带有 vstart=0 子句），段地址就是当前段寄存器 CS 的内容。表项修改完毕，从堆栈中恢复段寄存器 ES 的原始内容。

接下来，我们要设置 RTC 的工作状态，使它能够产生中断信号给 8259 中断控制器。

RTC 到 8259 的中断线只有一根，而 RTC 可以产生多种中断。比如闹钟中断、更新结束中断和周期性中断（参见表 9-3 和表 9-4）。RTC 的计时（更新周期）是独立的，产生中断信号只是它的一个赠品。所以，如果希望它能产生中断信号，需要额外设置。

以上所说的三种中断，我们只要设置一种就可以了。其实，最简单的就是设置更新周期结束中断。每当 RTC 更新了 CMOS RAM 中的日期和时间后，将发出此中断。更新周期每秒进行一次，因此该中断也每秒发生一次。

为了设置该中断，代码清单 9-1 第 147 行，将 RTC 寄存器 B 的索引 0x0b 传送到寄存器 AL。在访问 RTC 期间，最好是阻断 NMI，因此，第 148、149 行，先用 or 指令将 AL 的最高位置 1，再写端口 0x70。

第 150、151 行，用于通过数据端口 0x71 写寄存器 B。写的内容是 0x12，其二进制形式为 00010010，对照表 9-3，其意义不难理解：允许更新周期照常发生，禁止周期性中断，禁止闹钟功能，允许更新周期结束中断，使用 24 小时制，日期和时间采用 BCD 编码。

每次当中断实际发生时，可以在程序（中断处理过程）中读寄存器 C 的内容来检查中断的原因。比如，每当更新周期结束中断发生时，RTC 就将它的第 4 位置 1。该寄存器还有一个特点，就是每次读取它后，所有内容自动清零。而且，如果不读取它的话（换句话说，相应的位没有清零），同样的中断将不再产生。

为此，第 153~155 行，读一下寄存器 C 的内容，使之开始产生中断信号。注意，在向索引端口 0x70 写入的同时，也打开了 NMI。毕竟，这是最后一次在主程序中访问 RTC。

当然，如果采用周期性中断而不是更新周期结束中断，则稍微麻烦一些，因为要设置分频电路的分节点。以下代码片断用于产生 2 次/秒的周期性中断：

```
mov al,0xa
or al,0x80
out 0x70,al
in al,0x71
or al,0x0f           ; 设置 RTC 寄存器 A，使其每秒发生 2 次中断
out 0x71,al
```

除此之外，还要设置寄存器 B 的 PIE 位，以允许周期性中断。

RTC 芯片设置完毕后，再来打通它到 8259 的最后一道屏障。正常情况下，8259 是不会允许 RTC 中断的，所以，需要修改它内部的中断屏蔽寄存器 IMR。IMR 是一个 8 位寄存器，位 0 对应着中断输入引脚 IR0，位 7 对应着引脚 IR7，相应的位是 0 时，允许中断，为 1 时，关掉中断。

8259 芯片是我见过的芯片中，访问起来最麻烦，也是我最讨厌的一个。好在有关它的资料非常好找，这里就简单地进行讲解。代码清单 9-1 第 157~159 行，通过端口 0xa1 读取 8259 从片的 IMR 寄存器，用 and 指令清除第 0 位，其他各位保持原状，然后再写回去。于是，RTC 的中断可以被 8259 处理了。

第 161 行，sti 指令将标志寄存器的 IF 位置 1，开放设备中断。从这个时候开始，中断随时都会发生，也随时会被处理。

9.1.7 使处理器进入低功耗状态

RTC 更新周期结束中断的处理过程可以看成另一个程序，是独立的处理过程，是额外的执行流程，它随时都会发生，但和主程序互不相干。关于它的执行过程，马上就要讲到，现在继续来看主程序。

在为中断过程做了初始化工作之后，主程序还是要继续执行的。代码清单 9-1 第 163~167 行，用于显示中断处理程序已安装成功的消息。

接着，第 169~171 行，使段寄存器 DS 指向显示缓冲区，并在屏幕上的第 12 行 33 列显示一个字符“@”，该位置差不多是整个屏幕的中心。表达式 $12*160 + 33*2$ 是在指令编译阶段计算的，是该字符在显存中的位置。每个字符在显存中占 2 字节的位置，每行 80 个字符。

在此之后，主程序就无事可做了。第 174 行，hlt 指令使处理器停止执行指令，并处于停机状态，这将降低处理器的功耗。处于停机状态的处理器可以被外部中断唤醒并恢复执行，而且会继续执行 hlt 后面的指令。

所以，第 174~176 行用于形成一个循环，先是停机，接着某个外部中断使处理器恢复执行。一旦处理器的执行点来到 hlt 指令之后，则立即使它继续处于停机状态。

第 175 行，使用 not 指令将字符@的显示属性反转。not 是按位取反指令，其格式为

```
not r/m8
```

```
not r/m16
```

not 指令执行时，会将操作数的每一位反转，原来的 0 变成 1，原来的 1 变成 0。比如：

```
mov al,0x1f
```

```
not al           ; 执行后，AL 的内容为 0xe0
```

从显示效果上看，循环将显示属性反转将取得一个动画效果，可以很清楚地看到处理器每次从停机状态被唤醒的过程。not 指令不影响任何标志位。

相对于 jmp \$指令，使用 hlt 指令会大大降低处理器的占用率。Windows 7 操作系统有一个叫做 CPU 仪表盘的小工具，当使用 jmp \$指令时，你会看到处理器占用率是 100%；而在一个循环中使用 hlt 指令时，该占用率马上降到 10%左右，这还是在虚拟机环境下，毕竟宿主操作系统还要占用处理器时间。

9.1.8 实时时钟中断的处理过程

主程序就是这样了，停机、执行，接着停机。与此同时，中断也在不停地发生着，处理器还要抽出空来执行中断处理过程，下面就来看看 RTC 的更新周期结束中断处理，该中断处理过程从代码清单 9-1 的第 27 行开始。

第 28~32 行，先保护好现场，将后面用到的寄存器压栈保存。这一点特别重要，中断处理过程必须无痕地执行，你不知道中断会在什么时候发生，也不知道中断发生时，哪一个程序正在执行，所以，必须保证中断返回时，能还原中断前的状态。

第 34~40 行，用于读 RTC 寄存器 A，根据 UIP 位的状态来决定是等待更新周期结束，还是继续往下执行。UIP 位为 0 表示现在访问 CMOS RAM 中的日期和时间是安全的。注意第 36 行，用于把寄存器 AL 的最高位置 1，从而阻断 NMI。当然，这是不必要的，当 NMI 发生时，整个计算机都应当停止工作，也不在乎中断处理过程能否正常执行。

第 38 行从数据端口读取寄存器 A 的内容；第 39 行，test 指令用于测试寄存器 AL 的第 7 位是否为 1。

“test”的意思是“测试”。顾名思义，可以用这条指令来测试某个寄存器，或者内存单元里的内容是否带有某个特征。

test 指令在功能上和 and 指令是一样的，都是将两个操作数按位进行逻辑“与”，并根据结果设置相应的标志位。但是，test 指令执行后，运算结果被丢弃（不改变或破坏两个操作数的内容）。

test 指令需要两个操作数，其指令格式为

```
test r/m8,imm8
```

```
test r/m16,imm16
```

```
test r/m8,r8
```

```
test r/m16,r16
```

和 and 指令一样，test 指令执行后，OF=CF=0；对 ZF、SF 和 PF 的影响视测试结果而定；对 AF 的影响未定义。对于 test 指令的应用，这里有一个例子，比如，我们想测试 AI 寄存器的第 3 位是“0”还是“1”，可以这样编写代码：

```
test al,0x08
```

0x08 的二进制形式为 00001000，它的第 3 位是“1”，表明我们关注的是这一位。不管寄存器 AL 中的内容是什么，只要它的第 3 位是“0”，这条指令执行后，结果一定是 00000000，标志位 ZF=1；相反，如果寄存器 AL 的第 3 位是“1”，那么结果一定是 00001000，ZF=0。于是，根据 ZF 标志位的情况，就可以判定寄存器 AL 中的第 3 位是“0”还是“1”。

第 40 行, 如果 UIP 位是 0, 那么测试的结果是 ZF=1, 继续往下执行第 42 行; 否则, 说明 UIP 位是 1, 需要返回到第 34 行继续等待 RTC 更新周期结束。

正常情况下, 访问 CMOS RAM 中的日期和时间, 必须等待 RTC 更新周期结束, 所以上面的判断过程是必需的, 而这些代码也适用于正常的访问过程。但是, 当前中断处理过程是针对更新周期结束中断的, 而当此中断发生时, 本身就说明对 CMOS RAM 的访问是安全的, 毕竟留给我们的时间是 999 毫秒, 这段时间非常充裕, 这段时间能执行千万条指令。所以, 在这种特定的情况下, 上面的判断过程是不必要的。当然, 加上倒也无所谓。

第 42~52 行, 分别访问 CMOS RAM 的 0、2、4 号单元, 从中读取当前的秒、分、时数据, 按顺序压栈等待后续操作。

第 60~62 行, 读一下 RTC 的寄存器 C, 使得所有中断标志复位。这等于是告诉 RTC, 中断已经得到处理, 可以继续下一次中断。否则的话, RTC 看到中断未被处理, 将不再产生中断信号。RTC 产生中断的原因有多种, 可以在程序中通过读寄存器 C 来判断具体的原因。不过这里不需要, 因为除了更新周期结束中断外, 其他中断都被关闭了。

现在, 终于可以在屏幕上显示时间信息了。

第 64、65 行, 临时将段寄存器 ES 指向显示缓冲区。

第 67、68 行, 首先从堆栈中弹出小时数, 调用过程 `bcd_to_ascii` 来将用 BCD 码表示的“小时”转换成 ASCII。该过程是在第 105 行定义的, 调用该过程时, 寄存器 AL 中的高 4 位和低 4 位分别是“小时”的十位数字和个位数字。

第 108 行, 将寄存器 AL 中的内容复制一份给 AH, 以方便下一步操作。

第 109、110 行, 将寄存器 AL 中的高 4 位清零, 只留下“小时”的个位数字。接着, 将它加上 0x30, 就得到该数字对应的 ASCII 码。

十位上的数字在寄存器 AH 的高 4 位。第 112 行, 用右移 4 位的方法, 将它“拉”到低 4 位, 高 4 位在移动的过程中自动清零。

接着, 第 113、114 行, 用同样的办法来得到十位数字的 ASCII 码。此时, 寄存器 AH 中是十位数字的 ASCII 码, AL 中是个位数字的 ASCII 码, 它们将作为结果返回给调用者。

最后, 第 116 行用于返回调用者。

接着回到第 69 行, 为了连续在屏幕上显示内容, 最好是采用基址寻址来访问显存。这一行用于指定显示的内容位于显存的什么位置。实际上, 这里指定的是第 12 行 36 列。同以前一样, 每个字符在显存中占两个字节, 每行 80 个字符, 所以这里使用了表达式 $12*160 + 36*2$, 该表达式的值是在编译阶段计算的。

第 71、72 行, 分别将“小时”的两个数位写到显存中, 段地址在 ES 中, 偏移地址分别是由寄存器 BX 和 BX+2 提供的。这里没有写入显示属性, 这是因为我们希望采用默认的显示属性(屏幕是黑的, 默认的显示属性是 0x07, 即黑底白字)。

第 74、75 行, 用于在下一个屏幕位置显示冒号 “:”, 这是在显示时间时都会采用的分隔符。当然, 通过寄存器 AL 中转是多余的, 这两句可以直接写成

```
mov byte [es:bx+4], ':'
```

遗憾的是, 等我发现这个问题时, 本章已经快要写完了, 重新排版实在太费工夫。其实, 这不算是个问题, 无伤大雅, 难道不是吗?

为了验证 RTC 更新结束中断是每秒发生一次的, 第 76 行, 将冒号的显示属性(颜色)用 not 指令反转。就像手掌的两面一样, 每次发生中断时, 冒号的颜色将和上一次相反, 但永远在两个属性之间来回变化。到程序运行的时候你就会发现, 变化的频率是每秒一次。

剩下的指令都很好理解, 因为它们的工作是按相同的方法显示分钟数和秒数。第 78~90 行,

依次从堆栈中弹出分钟和秒的数值，并转换成 ASCII 码，然后显示在屏幕上，中间用冒号间隔。

在 8259 芯片内部，有一个中断服务寄存器（Interrupt Service Register, ISR），这是一个 8 位寄存器，每一位都对应着一个中断输入引脚。当中断处理过程开始时，8259 芯片会将相应的位置 1，表明正在服务从该引脚来的中断。

一旦响应了中断，8259 中断控制器无法知道该中断什么时候才能处理结束。同时，如果不清除相应的位，下次从同一个引脚出现的中断将得不到处理。在这种情况下，需要程序在中断处理过程的结尾，显式地对 8259 芯片编程来清除该标志，方法是向 8259 芯片发送中断结束命令（End Of Interrupt, EOI）。

中断结束命令的代码是 0x20。代码清单 9-1 第 92~94 行就用来做这件事。需要注意的是，如果外部中断是 8259 主片处理的，那么，EOI 命令仅发送给主片即可，端口号是 0x20；如果外部中断是由从片处理的，就像本章的例子，那么，EOI 命令既要发往从片（端口号 0xa0），也要发往主片。

最后，第 96~102 行，从堆栈中恢复被中断程序的现场，并用中断返回指令 iret 回到中断之前的地方继续执行。iret 的意思是 Interrupt Return。

9.1.9 代码清单 9-1 的编译和运行

本章的代码不包括加载器，也就是负责加载用户程序的主引导扇区代码，因为第 8 章已经提供了一个加载器，它同样可以加载本章的用户程序。

在完全理解了代码清单 9-1 的基础上，可以自行编辑和编译它，生成二进制文件。然后，使用 FixVhdWr 工具将其写入虚拟硬盘。和第 8 章一样，写入时的起始逻辑扇区号是 100，毕竟加载器每次要从这个地方读取和加载用户程序。

一旦所有工作都准备停当，即可启动虚拟机来观察运行结果。通常情况下，运行结果会如图 9-5 所示。

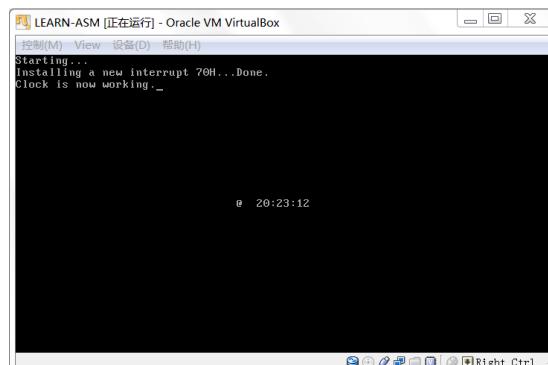


图 9-5 代码清单 9-1 编译和运行后的显示效果

在你欣赏程序的运行结果时，你一定会发现时间每秒更新一次，这从冒号的显示属性每秒反转一次可以看出来。与此不同的是，字符“@”却以很快的速度在闪烁。这意味着，把处理器从停机状态唤醒的不单单是实时时钟的更新周期结束中断，还有其他硬件中断，只不过我们不知道是谁而已。

9.2 内 部 中 断

和硬件中断不同，内部中断发生在处理器内部，是由执行的指令引起的。比如，当处理器检测

到 div 或者 idiv 指令的除数为零时，或者除法的结果溢出时，将产生中断 0（0 号中断），这就是除法错中断。

再比如，当处理器遇到非法指令时，将产生中断 6。非法指令是指指令的操作码没有定义，或者指令超过了规定的长度。操作码没有定义通常意味着那不是一条指令，而是普通的数。

内部中断不受标志寄存器 IF 位的影响，也不需要中断识别总线周期，它们的中断类型是固定的，可以立即转入相应的处理过程。

9.3 软 中 断

软中断是由 int 指令引起的中断处理。这类中断也不需要中断识别总线周期，中断号在指令中给出。int 指令的格式如下：

```
int3
int imm8
into
```

int3 是断点中断指令，机器指令码为 CC。这条指令在调试程序的时候很有用，当程序运行不正常时，多数时候希望在某个地方设置一个检查点，也称断点，来查看寄存器、内存单元或者标志寄存器的内容，这条指令就是为这个目的而设的。

指令都是连续存放的，因此，所谓的断点，就是某条指令的起始地址。int3 是单字节指令，这是有意设计的。当需要设置断点时，可以将断点处那条指令的第 1 字节改成 0xcc，原字节予以保存。当处理器执行到 int3 时，即发生 3 号中断，转去执行相应的中断处理程序。中断处理程序的执行也要用到各个寄存器，这会破坏它们的内容，但 push 指令不会。我们可以在该程序内先压栈所有相关寄存器和内存单元，然后分别取出予以显示，它们就是中断前的现场内容。最后，再恢复那条指令的第 1 字节，并修改位于堆栈中的返回地址，执行 iret 指令。

注意，int3 和 int 3 不是一回事。前者的机器码为 CC，后者则是 CD 03，这就是通常所说的 int n，其操作码为 0xCD，第 2 字节的操作数给出了中断号。举几个例子：

```
int 0x00      ;引发 0 号中断
int 0x15      ;引发 0x15 号中断
int 0x16      ;引发 0x16 号中断
```

into 是溢出中断指令，机器码为 0xCE，也是单字节指令。当处理器执行这条指令时，如果标志寄存器的 OF 位是 1，那么，将产生 4 号中断。否则，这条指令什么也不做。

9.3.1 常用的 BIOS 中断

可以为所有的中断类型自定义中断处理过程，包括内部中断、硬件中断和软中断。特别是考虑到处理器允许 256 种中断类型，而且大部分都没有被硬件和处理器内部中断占用。

编写自己的中断处理程序有相当大的优越之处。不像 jmp 和 call 指令，int 指令不需要知道目标程序的入口地址。远转移指令 jmp 和远调用指令 call 必须直接或者间接给出目标位置的段地址和偏移地址，如果所有这一切都是自己安排的，倒也不成问题，但如果想调用别人的代码，比如操作系统的功能，这就很麻烦了。举个例子来说，假如你想读硬盘上的一个文件，因为操作系统有这样的功能，所以就不必在自己的程序中再写一套代码，直接调用操作系统例程就可以了。

但是，操作系统通常不会给出或者公布硬盘读写例程的段地址和偏移地址，因为操作系统也是经常修改的，经常发布新的版本。这样一来，例程的入口地址也会跟着变化。而且，也不能保证每次启动计算机之后，操作系统总待在同一个内存位置。

因为有了软中断，这是个利好条件。每次操作系统加载完自己之后，以中断处理程序的形式提供硬盘读写功能，并把该例程的地址填写到中断向量表中。这样，无论在什么时候，用户程序需要该功能时，直接发出一个软中断即可，不需要知道具体的地址。

最有名的软中断是 BIOS 中断，之所以称为 BIOS 中断，是因为这些中断功能是在计算机加电之后，BIOS 程序执行期间建立起来的。换句话说，这些中断功能在加载和执行主引导扇区之前，就已经可以使用了。

BIOS 中断，又称 BIOS 功能调用，主要是为了方便地使用最基本的硬件访问功能。不同的硬件使用不同的中断号，比如，使用键盘服务时，中断号是 0x16，即

```
int 0x16
```

通常，为了区分针对同一硬件的不同功能，使用寄存器 AH 来指定具体的功能编号。举例来说，以下指令用于从键盘读取一个按键：

mov ah, 0x00	; 从键盘读字符
int 0x16	; 键盘服务。返回时，字符代码在寄存器 AL 中

在这里，当寄存器 AH 的内容是 0x00 时，执行 int 0x16 后，中断服务例程会监视键盘动作。当它返回时，会在寄存器 AL 中存放按键的 ASCII 码。

表 9-6 给出了常用的 BIOS 功能调用，一些过时的内容不再选取，可以参考其他资料。特别是，现在互联网上的资料很多。

表 9-6 常用的 BIOS 功能调用列表

INT	AH	功 能 说 明	入 口 参 数	返 回 参 数
-----	----	---------	---------	---------

10	0	设置显示方式	AL=00 40×25 黑白方式 AL=01 40×25 彩色方式 AL=02 80×25 黑白方式 AL=03 80×25 彩色方式 AL=04 320×200 彩色图形方式 AL=05 320×200 黑白图形方式 AL=06 320×200 黑白图形方式 AL=07 80×25 单色文本方式 AL=08 160×200 16 色图形 (PCjr) AL=09 320×200 16 色图形 (PCjr) AL=0A 640×200 16 色图形 (PCjr) AL=0B 保留 (EGA) AL=0C 保留 (EGA) AL=0D 320×200 彩色图形 (EGA) AL=0E 640×200 彩色图形 (EGA) AL=0F 640×350 黑白图形 (EGA) AL=10 640×350 彩色图形 (EGA) AL=11 640×480 单色图形 (EGA) AL=12 640×480 16 色图形 (EGA) AL=13 320×200 256 色图形 (EGA) AL=40 80×30 彩色文本 (CGE400) AL=41 80×50 彩色文本 (CGE400) AL=42 640×400 彩色图形 (CGE400)
----	---	--------	---

续表

INT	AH	功能说明	入口参数	返回参数
10	1	置光标类型	(CH) 0~3=光标起始行 (CL) 0~3=光标结束行	
10	2	置光标位置	BH=页号 DH,DL=行,列	
10	5	置显示页	AL=页号	
10	6	屏幕初始化或上卷	AL=上卷行数 AL=0 整个窗口空白 BH=卷入行属性 CH=左上角行号 CL=左上角行号 DH=右下角行号 DL=右下角行号	
10	7	屏幕初始化或上卷	AL=下卷行数 AL=0 整个窗口空白 BH=卷入行属性 CH=左上角行号 CL=左上角行号 DH=右下角行号 DL=右下角行号	
10	8	读光标位置的字符和属性	BH=显示页	AH=属性 AL=字符
10	9	在光标位置显示字符和属性	BH=显示页 AL=字符 BL=属性 CX=字符重复次数	
10	A	在光标位置显示字符	BH=显示页 AL=字符 CX=字符重复次数	
10	B	置彩色调板(320×200)	BH=彩色调板 ID BL=和 ID 配套使用的颜色	
10	C	写像素	DX=行(0~199) CX=列(0~639) AL=像素值	
10	D	读像素	DX=行(0~199) CX=列(0~639)	AL=像素值
10	E	显示字符(光标前移)	AL=字符 BL=前景色	
10	13	显示字符串(适用 AT)	ES:BP=串地址 CX=串长度 DH,DL=起始行,列 BH=页号 AL=0,BL=属性 串: char,char,... AL=1,BL=属性 串: char,char,... AL=2 串: char,char,... AL=3 串: char,char,...	光标返回起始位置 光标跟随移动 光标返回起始位置 光标跟随移动

续表

INT	AH	功能说明	入口参数	返回参数
13	2	读磁盘	AL=扇区数 CH,CL=磁盘号, 扇区号 DH,DL=磁头号, 驱动器号 ES:BX=数据缓冲区地址	读成功: AH=0 AL=读取的扇区数 读失败: AH=出错代码
13	3	写磁盘	同上	写成功: AH=0 AL=写入的扇区数 写失败: AH=出错代码
13	4	检验磁盘扇区	同上 (ES:BX 不设置)	成功: AH=0 AL=检验的扇区数 失败: AH=出错代码
13	5	格式化磁盘	ES:BX=磁道地址	成功: AH=0 失败: AH=出错代码
16	0	从键盘读字符		AL=字符码 AH=扫描码
16	1	读键盘缓冲区字符		ZF=0 AL=字符码 AH=扫描码 ZF=1 缓冲区空
16	2	读键盘状态字节		AL=键盘状态字节
17	0	打印字符 回送状态字节	AL=字符 DX=打印机号	AH=打印机状态字节
17	1	初始化打印机 回送状态字节	DX=打印机号	AH=打印机状态字节
17	2	取状态字节	DX=打印机号	AH=打印机状态字节
1A	0	读时钟		CH:CL=时:分 DH:DL=秒:1/100 秒
1A	1	置时钟	CH:CL=时:分 DH:DL=秒:1/100 秒	
1A	2	读实时时钟		CH:CL=时:分 (BCD) DH:DL=秒:1/100 秒 (BCD)
1A	6	置报警时间	CH:CL=时:分 (BCD) DH:DL=秒:1/100 秒 (BCD)	
1A	7	清除报警		

你可能觉得奇怪, BIOS 是怎么建立起这套功能调用中断的? 它又是怎么知道如何访问硬件的? 毕竟, 即使是它, 要访问硬件也得通过端口一级的途径。

答案是, BIOS 可能会为一些简单的外围设备提供初始化代码和功能调用代码, 并填写中断向量表, 但也有一些 BIOS 中断是由外部设备接口自己建立的。

首先, 每个外部设备接口, 包括各种板卡, 如网卡、显卡、键盘接口电路、硬件控制器等, 都有自己的只读存储器 (Read Only Memory, ROM), 类似于 BIOS 芯片, 这些 ROM 中提供了它自己的功能调用例程, 以及本设备的初始化代码。按照规范, 前两个单元的内容是 0x55 和 0xAA, 第三个单元是本 ROM 中以 512 字节为单位的代码长度; 从第四个单元开始, 就是实际的 ROM 代码。

其次, 我们知道, 从内存物理地址 A0000 开始, 到 FFFFF 结束, 有相当一部分空间是留给外围设备的。如果设备存在, 那么, 它自带的 ROM 会映射到分配给它的地址范围内。

在计算机启动期间，BIOS 程序会以 2KB 为单位搜索内存地址 C0000~E0000 之间的区域。当它发现某个区域的头两个字节是 0x55 和 0xAA 时，那意味着该区域有 ROM 代码存在，是有效的。接着，它对该区域做累加和检查，看结果是否和第三个单元相符。如果相符，就从第四个单元进入。这时，处理器执行的是硬件自带的程序指令，这些指令初始化外部设备的相关寄存器和工作状态，最后，填写相关的中断向量表，使它们指向自带的中断处理过程。

9.3.2 代码清单 9-2

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：9-2（被加载的用户程序/BIOS 中断演示程序），源程序文件：c09_2.asm

9.3.3 从键盘读字符并显示

代码清单 9-2 在框架上和前面的用户程序是一致的，差别在于代码段的功能上。

代码清单 9-2 第 28~32 行用于初始化各个段寄存器，这和以前的做法是相同的。

第 34~42 行用于在屏幕上显示字符串，采用的是循环的方法。循环用的是 loop 指令，为此，第 34 行用于计算字符串的长度，并传送到寄存器 CX 中，以控制循环的次数。第 35 行用于取得字符串的首地址。

向屏幕上写字符使用的是 BIOS 中断，具体地说就是中断 0x10 的 0x0e 号功能，该功能用于在屏幕上的光标位置处写一个字符，并推进光标位置。第 38~40 行分别按规范的要求准备各个参数，执行软中断。

第 41、42 行将递增寄存器 BX 中的偏移地址，以指向下一个字符在数据段中的位置。然后，loop 指令将寄存器 CX 的内容减 1，并在其不为零的情况下返回到循环体开始处，继续显示下一个字符。

剩下的工作内容既复杂，又简单。复杂是指，从键盘读取你按下的那个键，并把它显示在屏幕上，很复杂，需要访问硬件，写一大堆指令。简单是指，因为有了 BIOS 功能调用，这只需几条语句就能完成。

第 45、46 行使用软中断 0x16 从键盘读字符，需要在寄存器 AH 中指定 0x00 号功能。该中断返回后，寄存器 AL 中为字符的 ASCII 码。

第 48~50 行又一次使用了 int 0x10 的 0x0e 号功能，把从键盘取得的字符显示在屏幕上。

第 52 行，执行一个无条件转移指令，重新从键盘读取新的字符并予以显示。

9.3.4 代码清单 9-2 的编译和运行

将代码清单 9-2 编辑并编译后，用 FixVhdWr 程序将生成的二进制文件写入虚拟硬盘，起始的逻辑扇区号同样为 100。

如图 9-6 所示，启动虚拟机后，会看到一段欢迎的话。现在，你可以按下任何按键，它们将原样显示在“->”之后。慢慢试验，细细体会，你会发现某些按键的特点。比如，回车键（Enter）仅仅是将光标移到行首，退格键（Backspace）仅仅是将光标退后，并不破坏该位置上的字符。

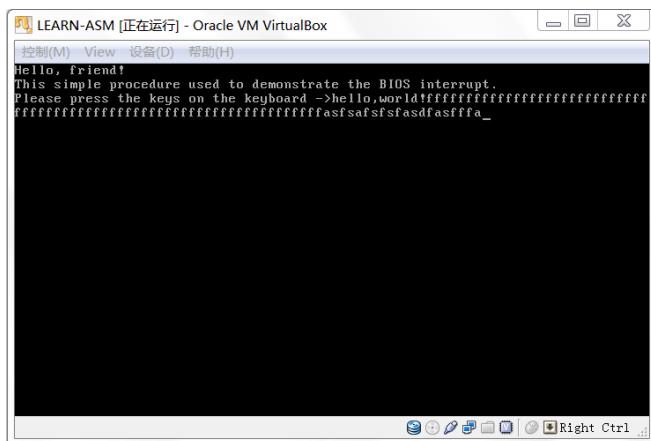


图 9-6 代码清单 9-2 编译并运行后的效果

本 章 习 题

1. 修改代码 9-1，对 8259 芯片编程，屏蔽除 RTC 外的其他所有中断，观察字符“@”的变化速度。
 2. 修改代码 9-1，使之用一种新的方法来产生中断信号。建议的方法是采用周期性中断。不过，这涉及选择分频电路的分节点，比如，你可以选择 250ms 或者 500ms，它们分别会在 1 秒种内产生 4 次或 2 次中断。

第 3 部分

32 位保护模式

第 10 章 32 位 Intel 微处理器编程架构

所谓处理器架构，或者处理器编程架构，是指一整套的硬件结构，以及与之相适应的工作状态，这其中的灵魄部分就是一种设计理念，决定了处理器的应用环境和工作模式，也决定了软件开发人员如何在这种模式下解决实际问题。

处理器架构实际上是不断扩展的，新处理器必须延续旧的设计思路，并保持兼容性和一致性；同时还会有所扩充和增强。

Intel 32 位处理器架构简称 IA-32 (Intel Architecture, 32-bit)，是以 1978 年的 8086 处理器为基础发展起来的。那个时候，他们只是想造一款特别牛的处理器，也没考虑到架构。尽管那帮人是专家，但和我们一样不是千里眼，这是很正常的。

正如我们已经知道的，8086 有 20 根地址线，可以寻址 1MB 内存。但是，它内部的寄存器是 16 位的，无法在程序中访问整个 1MB 内存。所以，它也是第一款支持内存分段模型的处理器。还有，8086 处理器只有一种工作模式，即实模式。当然，在那时，还没有实模式这一说。

由于 8086 处理器的成功，推动着 Intel 公司不断地研发更新的处理器，32 位的时代就这样到了。到目前为止，到底有多少种类型，我也说不清楚。尽管 8086 是 16 位的处理器，但它也是 32 位架构内的一部分。原因在于，32 位的处理器架构是从 8086 那里发展来的，是基于 8086 的，具有延续性和兼容性。

就我们曾经用过的产品而言，32 位的处理器有 32 根地址线，数据线的数量是 32 根或者 64 根。特别是最近最新的处理器，都是 64 根。因此，它可以访问 2^{32} ，即 4GB 的内存，而且每次可以读写连续的 4 字节或者 8 字节，这称为双字 (Double Word) 或者 4 字 (Quad Word) 访问。当然，如果你要按字节或者字来访问内存，也是允许的。

我总说，处理器虽小，功能却异常复杂。要想把 32 位处理器的所有功能都解释清楚，不是一件简单的事情。它不单单是地址线和数据线的扩展，实际上还有更多的部分，包括高速缓存、流水线、浮点处理部件、多处理器（核）管理、多媒体扩展、乱序执行、分支预测、虚拟化、温度和电源管理等。在这本书里，我的一个基本原则是，如果你不能讲清楚，干脆就不要提它。因此，我只讲那些现在用得上的东西。

10.1 IA-32 架构的基本执行环境

10.1.1 寄存器的扩展

在 16 位处理器内，有 8 个通用寄存器 AX、BX、CX、DX、SI、DI、BP 和 SP，其中，前 4 个还可以拆分成两个独立的 8 位寄存器来用，即 AH、AL、BH、BL、CH、CL、DH 和 DL。如图 10-1 所示，32 位处理器在 16 位处理器的基础上，扩展了这 8 个通用寄存器的长度。

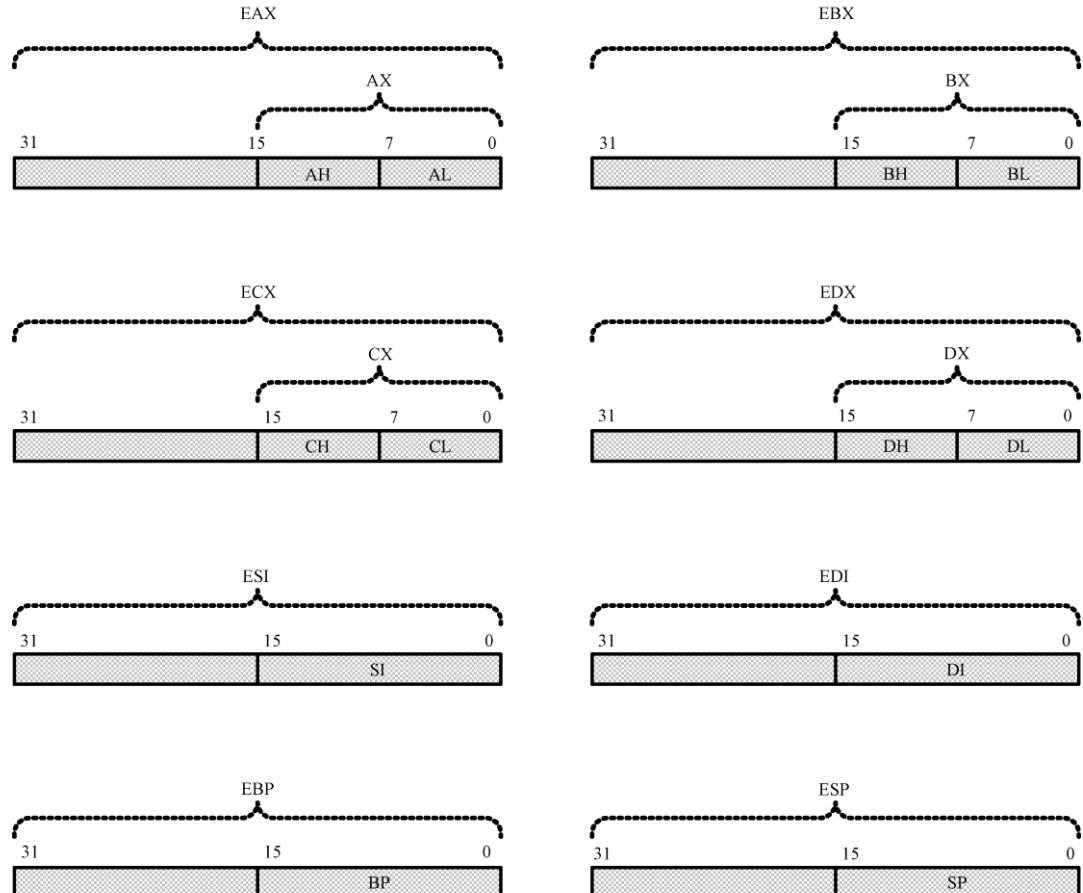


图 10-1 32 位处理器内部的通用寄存器

为了在汇编语言程序中使用经过扩展（Extend）的寄存器，需要给它们命名，它们的名字分别是 EAX、EBX、ECX、EDX、ESI、EDI、ESP 和 EBP。可以在程序中使用这些寄存器，即使是在实模式下：

```
mov eax,0xf0000005
mov ecx,eax
add edx,ecx
```

但是，就像以上指令所示的那样，指令的源操作数和目的操作数必须具有相同的长度，个别特殊用途的指令除外。因此，像这样的搭配是不允许的，在程序编译时，编译器会报告错误：

```
mov eax,cx ; 错误的汇编语言指令
```

如果目的操作数是 32 位寄存器，源操作数是立即数，那么，立即数被视为 32 位的：

```
mov eax,0xf5 ; EAX←0x000000f5
```

32 位通用寄存器的高 16 位是不可独立使用的，但低 16 位保持同 16 位处理器的兼容性。因此，在任何时候它们都可以照往常一样使用：

```
mov ah,0x02
mov al,0x03
add ax,si
```

可以在 32 位处理器上运行 16 位处理器上的软件。但是，它并不是 16 位处理器的简单增强。

事实上，32位处理器有自己的32位工作模式，在本书中，32位模式特指32位保护模式。在这种模式下，可以完全、充分地发挥处理器的性能。同时，在这种模式下，处理器可以使用它全部的32根地址线，能够访问4GB内存。

如图10-2所示，在32位模式下，为了生成32位物理地址，处理器需要使用32位的指令指针寄存器。为此，32位处理器扩展了IP，使之达到32位，即EIP。当它工作在16位模式下时，依然使用16位的IP；工作在32位模式下时，使用的是全部的32位EIP。和往常一样，即使是在32位模式下，EIP寄存器也只由处理器内部使用，程序中是无法直接访问的。对IP和EIP的修改通常是由某些指令隐式进行的，这些指令包括JMP、CALL、RET和IRET等等。

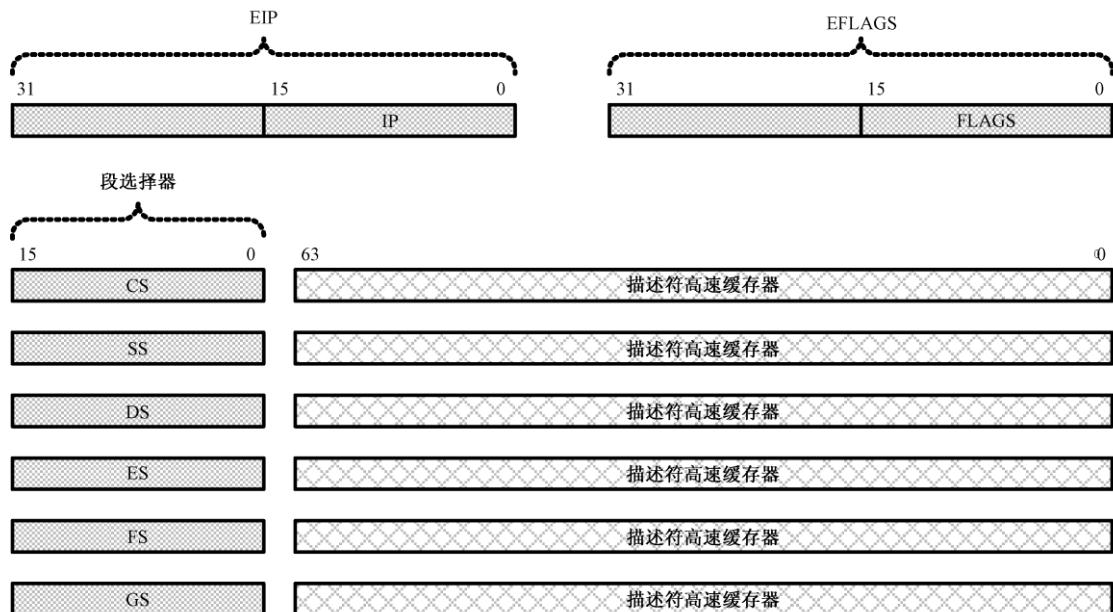


图10-2 32位处理器的指令指针、标志和段寄存器

另外，在16位处理器中，标志寄存器FLAGS是16位的，在32位处理器中，扩展到了32位，低16位和原先保持一致。关于EFLAGS中的各个标志位，将在后面的章节中逐一介绍。

在32位模式下，对内存的访问从理论上来说不再需要分段，因为它有32根地址线，可以自由访问任何一个内存位置。但是，IA-32架构的处理器是基于分段模型的，因此，32位处理器依然需要以段为单位访问内存，即使它工作在32位模式下。

不过，它也提供了一种变通的方案，即，只分一个段，段的基址是0x00000000，段的长度（大小）是4GB。在这种情况下，可以视为不分段，即平坦模型（Flat Mode）。

每个程序都有属于自己的内存空间。在16位模式下，一个程序可以自由地访问不属于它的内存位置，甚至可以对那些地方的内容进行修改。这当然是不安全的，也不合法，但却没有任何机制来限制这种行为。在32位模式下，处理器要求在加载程序时，先定义该程序所拥有的段，然后允许使用这些段。定义段时，除了基地址（起始地址）外，还附加了段界限、特权级别、类型等属性。当程序访问一个段时，处理器将用固件实施各种检查工作，以防止对内存的违规访问。

如图10-2所示，在32位模式下，传统的段寄存器，如CS、SS、DS、ES，保存的不再是16位段基地址，而是段的选择子，即，用于选择所要访问的段，因此，严格地说，它的新名字叫做段选择器。除了段选择器之外，每个段寄存器还包括一个64位的不可见部分，称为描述符高速缓存

器，里面有段的基址和各种访问属性。这部分内容程序不可访问，由处理器自动使用。

有关 32 位模式下的段和段的访问方法，将在后面的章节中予以详述，你在看这段文字的时候，也许有迷迷糊糊的感觉，没关系，这是正常的，到后面你就会感觉豁然开朗了。

最后，32 位处理器增加了两个额外的段寄存器 FS 和 GS。对于某些复杂的程序来说，多出两个段寄存器可能会令它们感到高兴。

10.1.2 基本的工作模式

8086 具有 16 位的段寄存器、指令指针寄存器和通用寄存器（CS、SS、DS、ES、IP、AX、BX、CX、DX、SI、DI、BP、SP），因此，我们称它为 16 位的处理器。尽管它可以访问 1MB 的内存，但是只能分段进行，而且由于只能使用 16 位的段内偏移量，故段的长度最大只能是 64KB。8086 只有一种工作模式，即实模式。当然，这个名称是后来才提出来的。

1982 年的时候，Intel 公司推出了 80286 处理器。这也是一款 16 位的处理器，大部分的寄存器都和 8086 处理器一样。因此，80286 和 8086 一样，因为段寄存器是 16 位的，而且只能使用 16 位的偏移地址，在实模式下只能使用 64KB 的段；尽管它有 24 根地址线，理论上可以访问 2^{24} ，即 16MB 的内存，但依然只能分成多个段来进行。

但是，80286 和 8086 不一样的地方在于，它第一次提出了保护模式的概念。在保护模式下，段寄存器中保存的不再是段地址，而是段选择子，真正的段地址位于段寄存器的描述符高速缓存中，是 24 位的。因此，运行在保护模式下的 80286 处理器可以访问全部 16MB 内存。

80286 处理器访问内存时，不再需要将段地址左移，因为在段寄存器的描述符高速缓存中有 24 位的段物理基地址。这样一来，段可以位于 16MB 内存空间中的任何位置，而不再限于低端 1MB 范围内，也不必非得是位于 16 字节对齐的地方。不过，由于 80286 的通用寄存器是 16 位的，只能提供 16 位的偏移地址，因此，和 8086 一样，即使是运行在保护模式下，段的长度依然不能超过 64KB。对段长度的限制妨碍了 80286 处理器的应用，这就是 16 位保护模式很少为人所知的原因。

实模式等同于 8086 模式，在本书中，实模式和 16 位保护模式统称 16 位模式。在 16 位模式下，数据的大小是 8 位或者 16 位的；控制转移和内存访问时，偏移量也是 16 位的。

1985 年的 80386 处理器是 Intel 公司的第一款 32 位产品，而且获得了极大成功，是后续所有 32 位产品的基础。本书中的绝大多数例子，都可以在 80386 上运行。和 8086/80286 不同，80386 处理器的寄存器是 32 位的，而且拥有 32 根地址线，可以访问 2^{32} ，即 4GB 的内存。

80386，以及所有后续的 32 位处理器，都兼容实模式，可以运行实模式下的 8086 程序。而且，在刚加电时，这些处理器都自动处于实模式下，此时，它相当于一个非常快速的 8086 处理器。只有在进行一番设置之后，才能运行在保护模式下。

在保护模式下，所有的 32 位处理器都可以访问多达 4GB 的内存，它们可以工作在分段模型下，每个段的基地址是 32 位的，段内偏移量也是 32 位的，因此，段的长度不受限制。在最典型的情况下，可以将整个 4GB 内存定义成一个段来处理，这就是所谓的平坦模式。在平坦模式下，可以执行 4GB 范围内的控制转移，也可以使用 32 位的偏移量访问任何 4GB 范围内的任何位置。32 位保护模式兼容 80286 的 16 位保护模式。

除了保护模式，32 位处理器还提供虚拟 8086 模式（V86 模式），在这种模式下，IA-32 处理器被模拟成多个 8086 处理器并行工作。V86 模式是保护模式的一种，可以在保护模式下执

行多个8086程序。传统上，要执行8086程序，处理器必须工作在实模式下。在这种情况下，为32位保护模式写的程序就不能运行。但是，V86模式提供了让它们在一起同时运行的条件。

V86模式曾经很有用，因为在那个时候，8086程序很多，而32位应用程序很少，这个过渡期是必需的。现在，这种工作模式已经基本无用了。

在本书中，32位模式特指IA-32处理器上的32位保护模式。不存在所谓的32位实模式，实模式的概念实质上就是8086模式。

10.1.3 线性地址

为IA-32处理器编程，访问内存时，需要在程序中给出段地址和偏移量，因为分段是IA-32架构的基本特征之一。传统上，段地址和偏移地址称为逻辑地址，偏移地址叫做有效地址（Effective Address，EA），在指令中给出有效地址的方式叫做寻址方式（Addressing Mode）。比如：

```
inc word [bx+si+0x06]
```

在这里，指令中使用的是基址加变址的方式来寻找最终的操作数。

段的管理是由处理器的段部件负责进行的，段部件将段地址和偏移地址相加，得到访问内存的地址。一般来说，段部件产生的地址就是物理地址。

IA-32处理器支持多任务。在多任务环境下，任务的创建需要分配内存空间；当任务终止后，还要回收它所占用的内存空间。在分段模型下，内存的分配是不定长的，程序大时，就分配一大块内存；程序小时，就分配一小块。时间长了，内存空间就会碎片化，就有可能出现一种情况：内存空间是有的，但都是小块，无法分配给某个任务。为了解决这个问题，IA-32处理器支持分页功能，分页功能将物理内存空间划分成逻辑上的页。页的大小是固定的，一般为4KB，通过使用页，可以简化内存管理。

如图10-3所示，当页功能开启时，段部件产生的地址就不再是物理地址了，而是线性地址（Linear Address），线性地址还要经页部件转换后，才是物理地址。

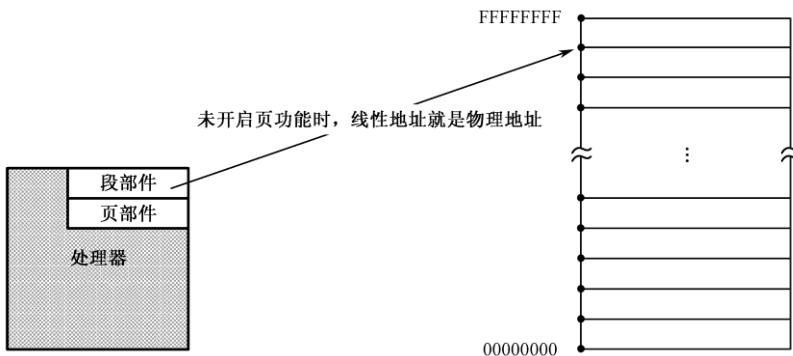


图10-3 线性地址和线性地址空间

线性地址的概念用来描述任务的地址空间。如图10-3所示，IA-32处理器上的每个任务都拥有4GB的虚拟内存空间，这是一段长4GB的平坦空间，就像一段平直的线段，因此叫线性地址空间。相应地，由段部件产生的地址，就对应着线性地址空间上的每一个点，这就是线性地址。

IA-32架构下的任务、分段、分页等内容，是本书的重点，要在后半部分详细论述。

10.2 现代处理器的结构和特点

10.2.1 流水线

处理器的每一次更新换代，都会增加若干新特性，这是很自然的。同时我们也会发现，老软件在新的处理器上跑得更快。这里面的原因很简单，处理器的设计者总是在想尽办法加快指令的执行。

早在 8086 时代，处理器就已经有了指令预取队列。当指令执行时，如果总线是空闲的（没有访问内存的操作），就可以在指令执行的同时预取指令并提前译码，这种做法是有效的，能大大加快程序的执行速度。

处理器可以做很多事情，换言之，能够执行各种不同的指令，完成不同的功能，但这些事情大都不会在一个时钟周期内完成。执行一条指令需要从内存中取指令、译码、访问操作数和结果，并进行移位、加法、减法、乘法以及其他任何需要的操作。

为了提高处理器的执行效率和速度，可以把一条指令的执行过程分解成若干个细小的步骤，并分配给相应的单元来完成。各个单元的执行是独立的、并行的。如此一来，各个步骤的执行在时间上就会重叠起来，这种执行指令的方法就是流水线（Pipe-Line）技术。

比如，一条指令的执行过程分为取指令、译码和执行三个步骤，而且假定每个步骤都要花 1 个时钟周期，那么，如图 10-4 所示，如果采用顺序执行，则执行三条指令就要花 9 个时钟周期，每 3 个时钟周期才能得到一条指令的执行结果；如果采用 3 级流水线，则执行这三条指令只需 5 个时钟周期，每隔一个时钟周期就能得到一条指令的执行结果。

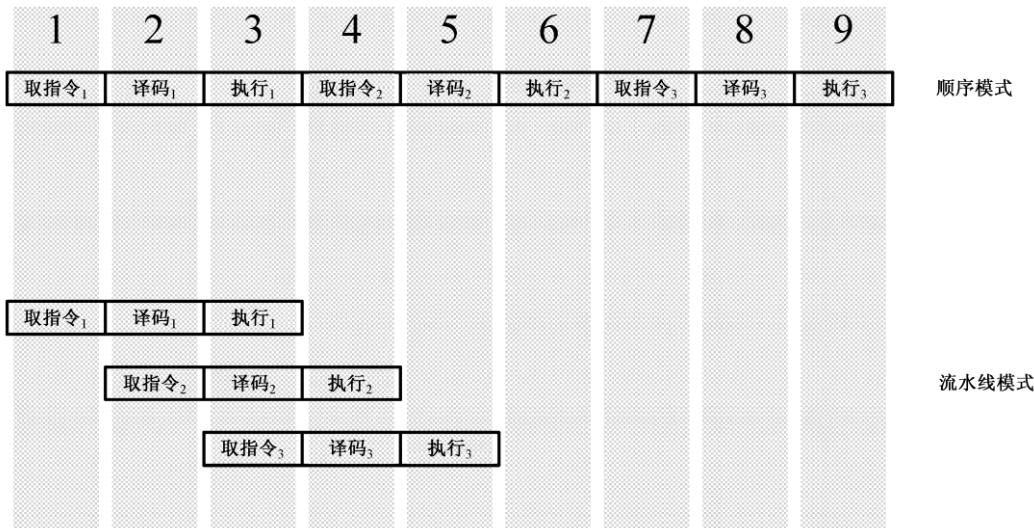


图 10-4 流水线的基本原理

一个简单的流水线其实不过如此，但是，它仍有很大的改进空间。原因很简单，指令的执行过程仍然可以继续细分。一般来说，流水线的效率受执行时间最长的那一级的限制，要缩短各级的执

行时间，就必须让每一级的任务减少，与此同时，就需要把一些复杂的任务再进行分解。比如，2000年之后推出的Pentium 4处理器采用了NetBurst微结构，它进一步分解指令的执行过程，采用了31级超深流水线。

10.2.2 高速缓存

影响处理器速度的另一个因素是存储器。从处理器内部向外看，它们分别是寄存器、内存和硬盘。当然，现在有的计算机已经用上了固态磁盘。

寄存器的速度是最快的，原因在于它使用了触发器，这是一种利用反馈原理制作的存储电路，在《穿越计算机的迷雾》那本书里，介绍得很清楚。触发器的工作速度是纳秒(ns)级别的，当然也可以用来做为内存的基本单元，即静态存储器(SRAM)，缺点是成本太高，价格也不菲。所以，制作内存芯片的材料一般是电容和单个的晶体管，由于电容需要定时刷新，使得它的访问速度变得很慢，通常是几十个纳秒。因此，它也获得了一个恰当的名字：动态存储器(DRAM)，我们所用的内存芯片，大部分都是DRAM。最后，硬盘是机电设备，是机械和电子的混合体，它的速度最慢，通常在毫秒级(ms)。

在这种情况下，因为需要等待内存和硬盘这样的慢速设备，处理器便无法全速运行。为了缓解这一矛盾，高速缓存(Cache)技术应运而生。高速缓存是处理器与内存(DRAM)之间的一个静态存储器，容量较小，但速度可以与处理器匹配。

高速缓存的用处源于程序在运行时所具有的局部性规律。首先，程序常常访问最近刚刚访问过的指令和数据，或者与它们相邻的指令和数据。比如，程序往往是序列化地从内存中取指令执行的，循环操作往往是执行一段固定的指令。当访问数据时，要访问的数据通常都被安排在一起；其次，一旦访问了某个数据，那么，不久之后，它可能会被再次访问。

利用程序运行时的局部性原理，可以把处理器正在访问和即将访问的指令和数据块从内存调入高速缓存中。于是，每当处理器要访问内存时，首先检索高速缓存。如果要访问的内容已经在高速缓存中，那么，很好，可以用极快的速度直接从高速缓存中取得，这称为命中(Hit)；否则，称为不中(Miss)。在不中的情况下，处理器在取得需要的内容之前必须重新装载高速缓存，而不只是直接到内存中去取那个内容。高速缓存的装载是以块为单位的，包括那个所需数据的邻近内容。为此，需要额外的时间来等待块从内存载入高速缓存，在该过程中所损失的时间称为不中惩罚(Miss Penalty)。

高速缓存的复杂性在于，每一款处理器可能都有不同的实现。在一些复杂的处理器内部，会存在多级Cache，分别应用于各个独立的执行部件。

10.2.3 乱序执行

为了实现流水线技术，需要将指令拆分成更小的可独立执行部分，即拆分成微操作(Micro-Operations)，简写为μ ops。

有些指令非常简单，因此只需要一个微操作。如：

```
add eax, ebx
```

再比如：

```
add eax, [mem]
```

可以拆分成两个微操作，一个用于从内存中读取数据并保存到临时寄存器，另一个用于将EAX寄

存器和临时寄存器中的数值相加。

再举个例子，这条指令：

```
add [mem],eax
```

可以拆分成三个微操作，一个从内存中读数据，一个执行相加的动作，第 3 个用于将相加的结果写回到内存中。

一旦将指令拆分成微操作，处理器就可以在必要的时候乱序执行（Out-Of-Order Execution）程序。考虑以下例子：

```
mov eax,[mem1]
shl eax,5
add eax,[mem2]
mov [mem3],eax
```

这里，指令 add eax,[mem2] 可以拆分为两个微操作。如此一来，在执行逻辑左移指令的同时，处理器可以提前从内存中读取 mem2 的内容。典型地，如果数据不在高速缓存中（不中），那么处理器在获取 mem1 的内容之后，会立即开始获取 mem2 的内容，与此同时，shl 指令的执行早就开始了。

将指令拆分成微操作，也可以使得堆栈的操作更有效率。考虑以下代码片断：

```
push eax
call func
```

这里，push eax 指令可以拆分成两个微操作，即可以表述为以下的等价形式：

```
sub esp,4
mov [esp],eax
```

这就带来了一个好处，即使 EAX 寄存器的内容还没有准备好，微操作 sub esp,4 也可以执行。call 指令执行时需要在当前堆栈中保存返回地址，在以前，该操作只能等待 push eax 指令执行结束，因为它需要 ESP 的新值。感谢微操作，现在，call 指令在微操作 sub esp,4 执行结束时就可以无延迟地立即开始执行。

10.2.4 寄存器重命名

考虑以下例子：

```
mov eax,[mem1]
shl eax,3
mov [mem2],eax
mov eax,[mem3]
add eax,2
mov [mem4],eax
```

以上代码片断做了两件事，但互不相干：将 mem1 里的内容左移 3 次（乘以 8），并将 mem3 里的内容加 2。如果我们为最后三条指令使用不同的寄存器，那么将更明显地看出这两件事的无关性。并且，事实上，处理器实际上也是这样做的。处理器为最后三条指令使用了另一个不同的临时寄存器，因此，左移（乘法）和加法可以并行地处理。

IA-32 架构的处理器只有 8 个 32 位通用寄存器，但通常都会被我们全部派上用场（甚至还觉得不够）。因此，我们不能奢望在每个计算当中都使用新的寄存器。不过，在处理器内部，却有大量

的临时寄存器可用，处理器可以重命名这些寄存器以代表一个逻辑寄存器，比如 EAX。

寄存器重命名以一种完全自动和非常简单的方式工作。每当指令写逻辑寄存器时，处理器就为那个逻辑寄存器分配一个新的临时寄存器。再来看一个例子：

```
mov eax, [mem1]
mov ebx, [mem2]
add ebx, eax
shl eax, 3
mov [mem3], eax
mov [mem4], ebx
```

假定现在 mem1 的内容在高速缓存里，可以立即取得，但 mem2 的内容不在高速缓存中。这意味着，左移操作可以在加法之前开始（使用临时寄存器代替 EAX）。为左移的结果使用一个新的临时寄存器，其好处是 EAX 寄存器中仍然是以前的内容，它将一直保持这个值，直到 EBX 寄存器中的内容就绪，然后同它一起做加法运算。如果没有寄存器重命名机制，左移操作将不得不等待从内存中读取 mem2 的内容到 EBX 寄存器以及加法操作完成。

在所有的操作都完成之后，那个代表 EAX 寄存器最终结果的临时寄存器的内容被写入真实的 EAX 寄存器，该处理过程称为引退（Retirement）。

所有通用寄存器，堆栈指针、标志、浮点寄存器，甚至段寄存器都有可能被重命名。

10.2.5 分支目标预测

流水线并不是百分之百完美的解决方案。实际上，有很多潜在的因素会使得流水线不能达到最佳的效率。一个典型的情况是，如果遇到一条转移指令，则后面那些已经进入流水线的指令就都无效了。换句话说，我们必须清空（Flush）流水线，从要转移到的目标位置处重新取指令放入流水线。

在现代处理器中，流水线操作分为很多步骤，包括取指令、译码、寄存器分配和重命名、微操作排序、执行和引退。指令的流水线处理方式允许处理器同时做很多事情。在一条指令执行时，下一条指令正在获取和译码。

流水线的最大问题是代码中经常存在分支。举个例子来说，一个条件转移允许指令流前往任意两个方向。如果这里只有一个流水线，那么，直到那个分支开始执行，在此之前，处理器将不知道应该用哪个分支填充流水线。流水线越长，处理器在用错误的分支填充流水线时，浪费的时间越多。

随着复杂架构下的流水线变得越来越长，程序分支带来的问题开始变得很大。让处理器的设计者不能接受，毕竟不中处罚的代价越来越高。

为了解决这个问题，在 1996 年的 Pentium Pro 处理器上，引入了分支预测技术（Branch Prediction）。分支预测的核心问题是，转移是发生还是不会发生。换句话说，条件转移指令的条件会不会成立。举个例子来说：

```
jne branch5
```

在这条指令还没有执行的时候，处理器就必须提前预测相等的条件在这条指令执行的时候是否成立。这当然是很困难的，几乎不可能。想想看，如果能够提前知道结果，还执行这些指令干嘛。

但是，从统计学的角度来看，有些事情一旦出现，下一次还会出现的概率较大。一个典型的例子就是循环，比如下面的程序片断：

```
xor si,si
```

```

lops:
...
cmp si,20
jnz lops

```

当 jnz 指令第一次执行时，转移一定会发生。那么，处理器就可以预测，下一次它还会转移到标号 lops 处，而不是顺序往下执行。事实上，这个预测通常是很准的。

在处理器内部，有一个小容量的高速缓存器，叫分支目标缓存器（Branch Target Buffer，BTB）。当处理器执行了一条分支语句后，它会在 BTB 中记录当前指令的地址、分支目标的地址，以及本次分支预测的结果。下一次，在那条转移指令实际执行前，处理器会查找 BTB，看有没有最近的转移记录。如果能找到对应的条目，则推测执行和上一次相同的分支，把该分支的指令送入流水线。

当该指令实际执行时，如果预测是失败的，那么，清空流水线，同时刷新 BTB 中的记录。这个代价较大。

10.3 32 位模式的指令系统

10.3.1 32 位处理器的寻址方式

在 16 位处理器上，指令中的操作数可以是 8 位或者 16 位的寄存器、指向 8 位或者 16 位实际操作数的 16 位内存地址，以及 8 位或 16 位的立即数。

如果指令中包含了内存地址操作数，那么，它必然是一个 16 位的段内偏移地址，称为有效地址。通过有效地址，可以间接取得 8 位或者 16 位的实际操作数。指定有效地址可以使用基址寄存器 BX、BP，变址（索引）寄存器 SI 和 DI，同时还可以加上一个 8 位或 16 位的偏移量。比如：

```

mov ax,[bx]
mov ax,[bx+di]
mov al,[bx+si+0x02]

```

以上，第 1 条指令，寄存器 BX 中的内容是指向 16 位实际操作数的 16 位地址；第 2 条指令，寄存器 BX 和 DI 的内容相加，形成指向 16 位实际操作数的 16 位地址；第 3 条指令，寄存器 BX、SI 和 8 位偏移量共同形成指向 8 位实际操作数的 16 位地址。

如图 10-5 所示，这是 16 位处理器的内存寻址方式示意图。从图中可以看出，允许使用基址寄存器 BX 或者 BP，同变址寄存器 SI 或者 DI 结合，再加上 8 位或者 16 位偏移量来寻址内存操作数。

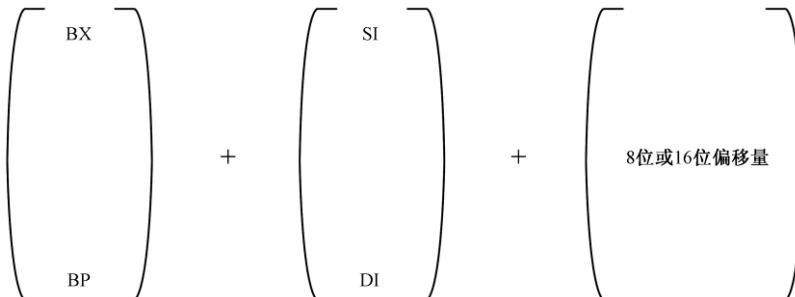


图10-5 16位处理器的内存寻址方式

32位处理器兼容16位处理器的工作模式，可以运行传统的16位代码。但是，它有自己独立的32位运行模式，而且只有在这种模式下才能发挥最高的运行效率。

在32位模式下，默认使用32位宽度的寄存器。如：

```
mov eax,ebx
```

如果指令中使用了立即数，那么，该数值默认是32位的：

```
mov ecx,0x55      ;ECX←0x00000055
```

还有，如果指令中的操作数是指向内存单元的地址，那么，该地址默认是32位的段内偏移地址，或者叫段内偏移量：

```
mov edx,[mem]      ;mem是一个32位的段内偏移地址
```

这就是说，如果指令中包含了内存地址操作数，那么，它必然默认地是一个32位的有效地址。通过有效地址，可以间接取得32位的实际操作数。如图10-6所示，指定有效地址可以使用全部的32位通用寄存器作为基址寄存器。同时，还可以再加上一个除ESP之外的32位通用寄存器作为变址寄存器。变址寄存器还允许乘以1、2、4或者8作为比例因子。最后，还允许加上一个8位或者32位的偏移量。

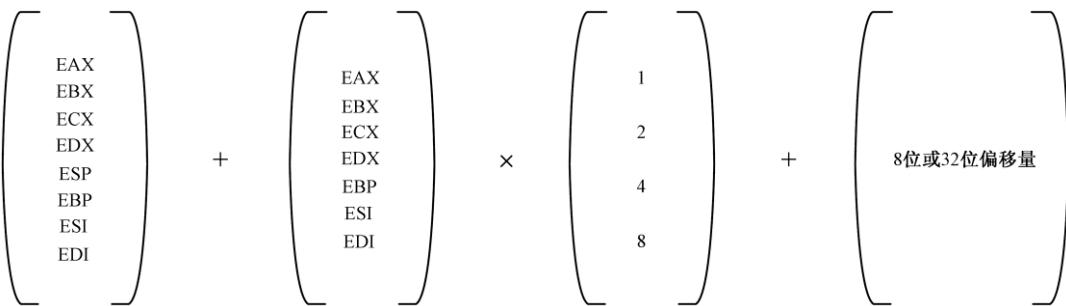


图10-6 32位处理器的内存寻址方式

以下是几个例子：

add eax,[0x2008]	;有效地址为0x00002008
sub eax,[eax+0x08]	;有效地址是32位的
mov ecx,[eax+ebx*8+0x02]	;有效地址是32位的

值得说明的是，在16位模式下，内存寻址方式的操作数不允许使用堆栈指针寄存器SP。因此，象这条指令就是不正确的：

```
mov ax,[sp]
```

但是，在32位模式下，允许在内存操作数中使用堆栈指针寄存器ESP。因此，下面的指令形式是合法的：

```
mov eax,[esp]
```

10.3.2 操作数大小的指令前缀

Intel处理器的指令系统比较复杂，这种复杂性来源于两个方面，一是指令的数量较多，二是寻址方式也很多。可以想象，为了组成这些众多的指令，必须有一套同样复杂的指令格式。

如图10-7所示，每一条处理器指令都可以拥有前缀，比如重复前缀（REP/REPE/REPNE）、段超越前缀（如ES:）、总线封锁前缀（LOCK）等。前缀是可选的，每个前缀的长度是1字节，每条

指令可以有 1~4 个前缀，或者不使用前缀。

前缀（如果有的话）的后面是操作码部分，指示执行什么样的操作，比如传送、加法、减法、乘法、除法、移位等。根据指令的不同，操作码的长度是 1~3 字节。同时，操作码还可以用来指示操作的字长，即数据宽度为字节还是字。

操作码之后是操作数类型和寻址方式部分。这部分是可选的，简单的指令不包含这一部分，稍微复杂一点的指令，这一部分只有 1 字节；最复杂的指令，可能有 2 字节。这部分给出了指令的寻址方式，以及寄存器的类型（用的是哪个寄存器）。

指令的最后是立即数和偏移量。如果指令中使用了立即数，那么立即数就在这一部分给出；如果指令使用了带偏移量的寻址方式，如：

```
mov cx, [0x2000]
mov ecx, [eax+ebx*8+0x02]
```

那么，偏移量 0x2000 和 0x02 也在这部分出现。取决于具体的指令，立即数可以是 1、2 或者 4 字节，偏移量部分与此相同。

前缀	操作码	寻址方式和操作数类型	立即数	偏移量
----	-----	------------	-----	-----

图 10-7 IA-32 的指令格式

上述的指令编码格式发源于 16 位处理器时代，并在 32 位处理器出现之后做了修改，主要是扩展了数据的宽度，其他都保持不变。毕竟，兼容性是首要考虑的因素。但是，这也带来了一些问题。考虑以下指令：

```
mov dx, [bx+si+0x02]
```

在 16 位指令编码格式中，这种内存单元到寄存器的传送指令使用了操作码 0x8B。如图 10-8(a) 所示，在操作码 0x8B 之后是 1 字节的寻址方式和操作数类型部分。位 7 和位 6 的值是 01，表示使用了基地址变址的寻址方式，而且带有 8 位偏移量；位 5~位 3 的值是 010，指示目的操作数为寄存器 DX；位 2~位 0 的值是 000，表示寻址方式为“BX+SI+8 位偏移量”。在该字节之后，是 1 字节的偏移量 0x02。因此，这条指令编译后的机器代码是

```
8B 50 02
```

32 位处理器使用相同的编码格式，但是，寻址方式和寄存器的定义却是另起炉灶的，完全不同于 16 位指令。如图 10-8 (b) 所示，在 32 位处理器上，位 7 和位 6 的值是 01，表示使用了基址寻址方式，而且带有 8 位偏移量；位 5~位 3 的值是 010，指示目的操作数为寄存器 EDX；位 2~位 0 的值是 000，表示寻址方式为 EAX+8 位偏移量。在该字节之后，是 1 字节的偏移量 0x02。因此，同样的机器指令码，却对应着不同的 32 位指令：

```
mov edx, [eax+0x02]
```

这就是说，相同的机器指令，在 16 位模式下和 32 位模式下的解释和执行效果是不同的。但是，别忘了，32 位处理器可以执行 16 位的程序，包括实模式和 16 位保护模式。为此，在 16 位模式下，处理器把所有指令都看成是 16 位的。举个例子，机器指令码 0x40 在 16 位模式下的含义是

```
inc ax
```

当处理器在 16 位模式下运行时，也可以使用 32 位的寄存器，执行 32 位的运算。为此，必须使用指令前缀 0x66 来临时改变这种默认状态，因为同一个指令码，在 16 位模式下和 32 位模式下具有不同的解释。因此，当处理器在 16 位模式下运行时，机器指令码

```
66 40
```

对应的指令不再是 inc ax，而是

```
inc eax
```

相反地，如果处理器运行在32位模式下，那么，处理器认为指令的操作数都是32位的，如果你加了前缀，这个前缀就用来指示指令是16位的。因此，指令前缀0x66具有反转当前默认操作数大小的作用。

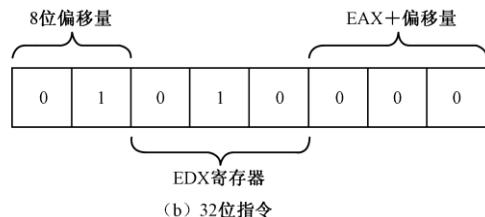
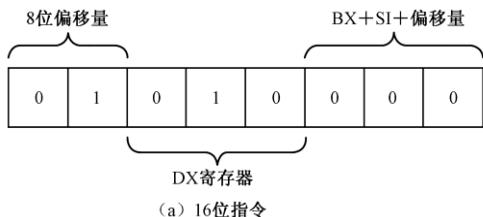


图10-8 16位指令和32位指令的寻址方式和操作数类型编码对比

在编写程序的时候，就应当考虑到指令的运行环境。为了指明程序的默认运行环境，编译器提供了伪指令bits，用于指明其后的指令应该被编译成16位的，还是32位的。比如：

```
bits 16
mov cx,dx      ;89 D1
mov eax,ebx     ;66 89 D8

bits 32
mov cx,dx      ;66 89 D1
mov eax,ebx     ;89 D8
```

注意，bits 16或者bits 32可以放在方括号中，也可以没有方括号。以下两种方式都是允许的：

```
[bits 32]
mov ecx,edx

bits 16
mov ax,bx
```

最后，16位模式是默认的编译模式。如果没有指定指令的编译模式，则默认是“bits 16”的。

有关寻址方式和指令前缀的话题比较复杂，在后面的章节里，我们将在适当的时候，结合程序和具体的指令进行讲解。

10.3.3 一般指令的扩展

由于32位的处理器都拥有32位的寄存器和算术逻辑部件，而且同内存芯片之间的数据通路至少是32位的，因此，所有以寄存器或者内存单元为操作数的指令都被扩充，以适应32位的算术逻辑操作。而且，这些扩展的操作即使是在16位模式下（实模式和16位保护模式）也是可用的。比如加法指令ADD，在32位处理器上，除了允许8位或者16位的操作数外，32位的操作数现在也是可用的：

```
add al,bl
add ax,bx
add eax,ebx
add dword [ecx],0x0000005f
```

除了双操作数指令，单操作数指令也同样允许 32 位操作数。比如：

```
inc al
inc dword [0x2000]
dec dword [eax*2]
```

我们已经接触过的逻辑移动指令，如 shl、shr 等，目的操作数也扩展至 32 位，但用于指定移动次数的源操作数足够应付 32 位的环境，没有变化。举例：

```
shl eax,1
shl eax,9
shl dword [eax*2+0x08],cl
```

和 16 位时代一样，在 32 位处理器上，逻辑移动指令的源操作数如果是寄存器的话，则依然必须使用 CL。同时，32 位处理器在实际执行时，要先将源操作数（在 CL 寄存器内）同 0x1F 做逻辑与。也就是说，仅保留源操作数的低 5 位，因此，实际移动的次数最大为 31。

在 16 位处理器上，loop 指令的循环次数在寄存器 CX 中。在 32 位处理器上，如果当前的运行模式是 16 位的（bits 16，8086 实模式或者 16 位保护模式），那么，loop 指令执行时，依然使用 CX 寄存器；否则，如果运行在 32 位模式下（bits 32），则使用的是 ECX 寄存器。

在 16 位处理器上，无符号数乘法指令 mul 的格式为

```
mul r/m8 ;AX ← AL×r/m8
mul r/m16 ;DX:AX ← AX×r/m16
```

在 32 位处理器上，除了依然支持上述操作外，还支持以下扩展的格式：

```
mul r/m32 ;EDX:EAX ← EAX×r/m32
```

这样，两个 32 位的数相乘，得到一个 64 位的结果。这里有个例子：

```
mov eax,0x10000
mov ebx,0x20000
mul ebx
```

有符号数乘法指令 imul 与此相同。

相应地，无符号数和有符号数除法也做了 32 位扩展：

```
div r/m32
idiv r/m32
```

在这里，被除数是 64 位的，高 32 位在 EDX 寄存器；低 32 位在 EAX 寄存器。除数是 32 位的，位于 32 位的寄存器，或者存放有 32 位实际操作数的内存地址。指令执行后，32 位的商在 EAX 寄存器，32 位的余数在 EDX 寄存器。

32 位处理器的堆栈操作指令 push 和 pop 也有所扩展，允许压入双字操作数。特别是，它现在支持立即数压栈操作。立即数压栈操作的指令格式为

```
push imm8 ;操作码为 6A
push imm16 ;操作码为 68
push imm32 ;操作码为 68
```

举个例子可能更清楚一些。比如：

```
push byte 0x55
```

在这里，关键字“byte”仅仅是给编译器用的，告诉它，压入的是字节（毕竟立即数 0x55 可以解释为字 0x0055 或者双字 0x00000055），而不是用来在编译后的机器指令前添加指令前缀。

这条指令的16位形式（用bits 16编译）和32位形式（用bits 32编译）是一样的，机器代码都是

```
6A 55
```

但是，当它执行时，就不同了。注意，无论在什么时候，处理器都不会真的压入一字节，要么压入字，要么压入双字。因此，在16位模式下，默认的操作数字长是16，处理器在执行时，将该字节的符号位扩展到高8位，然后压入堆栈，压栈时使用SP寄存器，且先将SP的内容减去2。这就是说，实际压入堆栈中的数值是0x0055；在32位模式下，压入的内容是该字节操作数符号位扩展到高24位的结果，即0x00000055。压栈时使用ESP寄存器，且先将ESP的内容减去4。

如果压入的是字操作数，则必须用关键字“word”来修饰。如：

```
push word 0xfffffb
```

在16位模式下，默认的操作数字长是16，处理器在执行时，直接压入该字，压栈时使用SP寄存器，且先将SP的内容减去2；在32位模式下，压入的内容是该操作数符号位扩展到高16位的结果，即0xFFFFFFFb，压栈时使用ESP寄存器，且先将ESP的内容减去4。

如果压入的是双字操作数，则必须用关键字“dword”来修饰。如：

```
push dword 0xfb
```

则无论是在16位模式下，还是在32位模式下，压入的都是0x000000FB，而且堆栈指针寄存器（SP或者ESP）都先减去4。

对于实际操作数位于通用寄存器，或者位于内存单元的情况，只能压入字或者双字，指令格式为：

```
push r/m16
```

```
push r/m32
```

如果是寄存器，则可以使用16位或者32位的通用寄存器。比如：

```
push ax
```

```
push edx
```

如果被压入的16位或者32位操作数位于内存单元中，则必须用关键字“word”或者“dword”修饰，以指示操作数的大小：

```
push word [0x2000]
```

```
push dword [ecx+esi*2+0x02]
```

无论被压入的数位于寄存器，还是位于内存单元，在16位模式下，如果压入的是字操作数，那么先将SP的内容减去2；如果压入的是双字，应当先将SP的内容减去4。在32位模式下，如果压入的是字操作数，那么先将ESP的内容减去2；如果压入的是双字，应当先将ESP的内容减去4。

压入段寄存器的操作比较特殊。以下是压入段寄存器的push指令格式：

push cs	;机器指令为0E
push ds	;机器指令为1E
push es	;机器指令为06
push fs	;机器指令为0F A0
push gs	;机器指令为0F A8
push ss	;机器指令为16

在16位模式下，先将SP的内容减去2，然后直接压入段寄存器的内容；在32位模式下，要先将段寄存器的内容用零扩展到32位，即高16位为全零。然后，将ESP的内容减去4，再压入扩展后的32位值。

本 章 习 题

1. 在编译阶段，如果指定的编译模式是 bits 16，那么，`mov bx,16` 的机器码为 BB 10 00。相反，`mov ebx,16` 的机器码为 66 BB 10 00 00 00。试问，如果指定了编译模式 bits 32，这两条指令编译后的机器码又分别是什么？

2. 以下程序片断：

```
bits 16  
mov bx,16      ;BB 10 00  
mul bx        ;F7 E3
```

将生成机器指令序列 BB 10 00 F7 E3。

当处理器在 32 位保护模式下执行这些代码时，会有什么问题？

第 11 章 进入保护模式

一般来说，操作系统负责整个计算机软、硬件的管理，它做任何事情都是可以的。但是，用户程序却应当有所限制，只允许它访问属于自己的数据，即使是转移，也只允许在自己的各个代码段之间进行。

问题在于，在实模式下，用户程序对内存的访问非常自由，没有任何限制，随随便便就可以修改任何一个内存单元。比如以下代码片断，它先保存当前的数据段地址，然后修改别人的数据，最后再回到原先的数据段：

```
mov cx,0x8000      ;逻辑段地址  
push ds  
mov ds,cx  
mov [0x05],dx      ;逻辑地址 0x8000:0x0005，即物理地址 0x80005  
pop ds
```

很显然，即使物理内存单元 0x80005 不属于当前程序，它照样可以切换到那里，并随意修改其中的内容。最恐怖的是，如果那个地方是操作系统或其他用户程序的“地盘”，那将带来不可预料的后果。通过这个例子，你就知道为什么很多人能通过修改内存中的数据来提升游戏人物的法力和生命值，并获得各种道具。

在多用户、多任务时代，内存中会有多个用户（应用）程序在同时运行。为了使它们彼此隔离，防止因某个程序的编写错误或者崩溃而影响到操作系统和其他用户程序，使用保护模式是非常有必要的。

11.1 代码清单 11-1

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：11-1（主引导扇区程序）

源程序文件：c11_mbr.asm

11.2 全局描述符表

我们知道，为了让程序在内存中能自由浮动而又不影响它的正常执行，处理器将内存划分成逻辑上的段，并在指令中使用段内偏移地址。在保护模式下，对内存的访问仍然使用段地址和偏移地址，但是，在每个段能够访问之前，必须先进行登记。

这种情况好有一比。就像是开公司做生意，在实模式下，开公司不需要登记，卖什么都没有人管，随时都可以开张。但在保护模式下就不行了，开公司之前必须先登记，登记的信息包括住址（段的起始地址）、经营项目（段的界限等各种访问属性）。这样，每当你做的买卖和

项目不符时，就会被阻止。对段的访问也是一样，当你访问的偏移地址超出段的界限时，处理器就会阻止这种访问，并产生一个叫做内部异常的中断。

和一个段有关的信息需要 8 个字节来描述，所以称为段描述符（Segment Descriptor），每个段都需要一个描述符。为了存放这些描述符，需要在内存中开辟出一段空间。在这段空间里，所有的描述符都是挨在一起，集中存放的，这就构成一个描述符表。

最主要的描述符表是全局描述符表（Global Descriptor Table，GDT），所谓全局，意味着该表是为整个软硬件系统服务的。在进入保护模式前，必须要定义全局描述符表。

如图 11-1 所示，为了跟踪全局描述符表，处理器内部有一个 48 位的寄存器，称为全局描述符表寄存器（GDTR）。该寄存器分为两部分，分别是 32 位的线性地址和 16 位的边界。32 位的处理器具有 32 根地址线，可以访问的地址范围是 0x00000000 到 0xFFFFFFFF，共 2^{32} 字节的内存，即 4GB 内存。所以，GDTR 的 32 位线性地址部分保存的是全局描述符表在内存中的起始线性地址，16 位边界部分保存的是全局描述符表的边界（界限），其在数值上等于表的大小（总字节数）减一。



图 11-1 全局描述符表寄存器 GDTR

换句话说，全局描述符表的界限值就是表内最后 1 字节的偏移量。第 1 字节的偏移量是 0，最后 1 字节的偏移量是表大小减一。如果界限值为 0，表示表的大小是 1 字节。

因为 GDT 的界限是 16 位的，所以，该表最大是 2^{16} 字节，也就是 65536 字节（64KB）。又因为一个描述符占 8 字节，故最多可以定义 8192 个描述符。实际上，不一定非得这么多，到底有多少，视需要而定，但最多不能超过 8192 个。

理论上，全局描述符表可以位于内存中的任何地方。但是，如图 11-2 所示，由于在进入保护模式之后，处理器立即要按新的内存访问模式工作，所以，必须在进入保护模式之前定义 GDT。但是，由于在实模式下只能访问 1MB 的内存，故 GDT 通常都定义在 1MB 以下的内存范围中。当然，允许在进入保护模式之后换个位置重新定义 GDT。

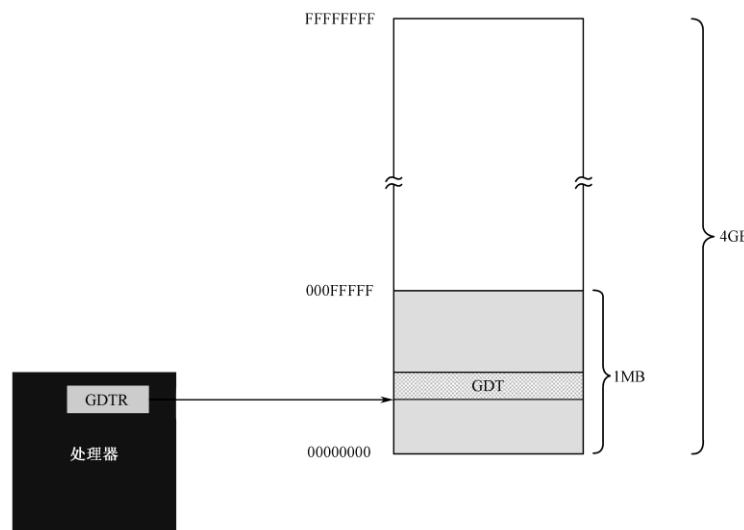


图 11-2 GDT 和 GDTR 的关系示意图

11.3 存储器的段描述符

和往常一样，在程序的开始部分要初始化段寄存器。代码清单 11-1 第 7~9 行用于初始化堆栈，使堆栈段的逻辑段地址和代码段相同，并使堆栈指针寄存器 SP 指向 0x7c00。这是个分界线，从这里，代码向上扩展，而堆栈向下扩展。

下面开始定义主引导扇区代码所使用的数据段、代码段和堆栈段。在保护模式下，内存的访问机制完全不同，即，必须通过描述符来进行。所以，这些段必须重新在 GDT 中定义。

先是确定 GDT 的起始线性地址。代码清单 11-1 第 96 行，声明了标号 `gdt_base` 并初始化了一个双字 `0x00007e00`，我们决定从这个地方开始创建全局描述符表（GDT）。这是有意的，如图 11-3 所示，在实模式下，主引导程序的加载位置是 `0x0000:0x7c00`，也就是物理地址 `0x07c00`。因为现在的地址是 32 位的，所以它现在对应着物理地址 `0x00007c00`。主引导扇区程序共 512（`0x200`）字节，所以，我们决定把 GDT 设在主引导程序之后，也就是物理地址 `0x00007e00` 处。因为 GDT 最大可以为 64KB，所以，理论上，它的尺寸可以扩展到物理地址 `0x00017dff` 处。

相应地，因为堆栈指针寄存器 `SP` 被初始化为 `0x7c00`，和 `CS` 一样，堆栈段寄存器 `SS` 被初始化为 `0x0000`，而且堆栈是向下扩展的，所以，从 `0x00007c00` 往下的区域是实际上可用的堆栈区域。只不过，该区域包含了很多 BIOS 数据，包括实模式下的中断向量表，所以一定要小心。这是没有办法的事，在实模式下，处理器不会为此负责，只能靠你自己。

实模式和保护模式在内存访问上是有区别的，在保护模式下，你不能说访问哪个段就访问哪个段，在访问之前，必须先在 GDT 内定义要访问的内存段。也许你觉得多此一举，“想访问哪段内存，我就在 GDT 中定义一个描述符，这和直接访问有什么区别？反正也能随心所欲，只不过多了一道手续，这又谈何限制和保护呢？”

实际上并非如此。如果整个计算机系统中只有一个程序在工作，那当然是正确的。问题在于，会有很多程序共同在操作系统上运行。想想你平时玩的电子游戏、音视频播放器、WPS、Word、Excel，它们都依靠 Windows 的支撑才能运行。所以，描述符不是由用户程序自己建立的，而是在加载时，由操作系统根据你的程序结构而建立的，而用户程序通常是无法建立和修改 GDT 的，也就只能老老实实地在自己的地盘上工作。在这种情况下，操作系统为你的程序建立了几个段，你就只能在这些段内工作，超出这个范围，或者未按预定的方法访问这些段，都将被处理器阻止。

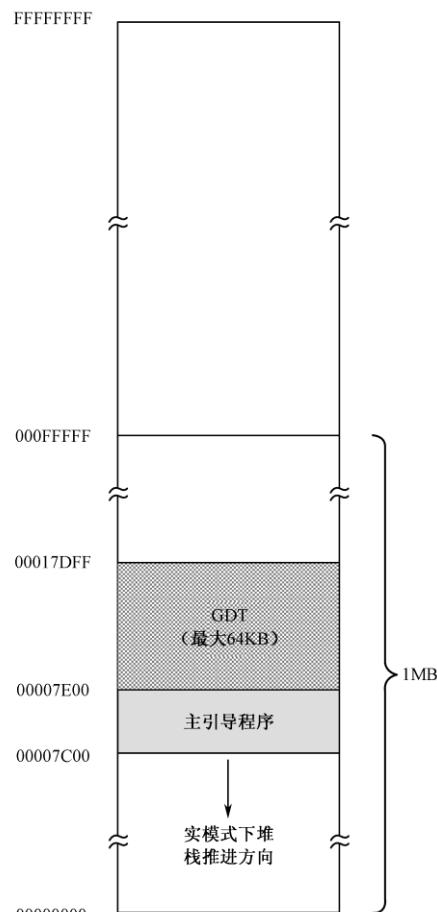


图 11-3 进入保护模式前的内存映象

一旦确定了 GDT 在内存中的起始位置，下一步的工作就是确定要访问的段，并在 GDT 中为这些段创建各自的描述符。

如图 11-4 所示，每个描述符在 GDT 中占 8 字节，也就是 2 个双字，或者说是 64 位。图中，下面是低 32 位，上面是高 32 位。



图 11-4 存储器的段描述符格式

很明显，描述符中指定了 32 位的段起始地址，以及 20 位的段边界。在实模式下，段地址并非真实的物理地址，在计算物理地址时，还要左移 4 位（乘以 16）。和实模式不同，在 32 位保护模式下，段地址是 32 位的线性地址，如果未开启分页功能，该线性地址就是物理地址。页功能将在第 16 章和第 17 章讲解，而且开启页功能需要做很多准备工作。目前，如果没有特别说明，线性地址就是物理地址。描述符中的段地址和段界限不是连续的，把它们分成几段似乎不科学。但这也是没有办法的事，这是从 80286 处理器上带来的后遗症。80286 也是 16 位的处理器，也有保护模式，但属于 16 位的保护模式。而且，其地址是 24 位的，允许访问最多 16MB 的内存。尽管 80286 的 16 位保护模式从来也没形成气候，但是，32 位处理器为了保持同 80286 的兼容，只能在旧描述符的格式上进行扩充，这是不得已的做法。

段地址可以是 0~4GB 范围内的任意地址，不过，还是建议应当选取那些 16 字节对齐的地址。尽管对于 Intel 处理器来说，允许不对齐的地址，但是，对齐能够使程序在访问代码和数据时的性能最大化。这一点，对于那些学过计算机原理，特别是了解内存芯片组织的人来说，是最清楚不过的。

20 位的段界限用来限制段的扩展范围。因为访问内存的方法是用段地址加上偏移量，所以，对于向上扩展的段，如代码段和数据段来说，偏移量是从 0 开始递增，段界限决定了偏移量的最大值；对于向下扩展的段，如堆栈段来说，段界限决定了偏移量的最小值。

G 位是粒度（Granularity）位，用于解释段界限的含义。当 G 位是“0”时，段界限以字节为单位。此时，段的扩展范围是从 1 字节到 1 兆字节（1B~1MB），因为描述符中的界限值是 20 位的。相反，如果该位是“1”，那么，段界限是以 4KB 为单位的。这样，段的扩展范围是从 4KB 到 4GB。

S 位用于指定描述符的类型（Descriptor Type）。当该位是“0”时，表示是一个系统段；为“1”时，表示是一个代码段或者数据段（堆栈段也是特殊的数据段）。系统段将在以后介绍。

DPL 表示描述符的特权级（Descriptor Privilege Level, DPL）。这两位用于指定段的特权级。共有 4 种处理器支持的特权级别，分别是 0、1、2、3，其中 0 是最高特权级别，3 是最低特权级别。刚进入保护模式时执行的代码具有最高特权级 0（可以看成是从处理器那里继承来的），这些代码通常都是操作系统代码，因此它的特权级别最高。每当操作系统加载一个用户程序时，它通常都会指定一个稍低的特权级，比如 3 特权级。不同特权级别的程序是互相隔离的，其互访是严格限制的，而且有些处理器指令（特权指令）只能由 0 特权级的程序来执行，为的就是安全。

在这里，描述符的特权级用于指定要访问该段所必须具有的最低特权级。如果这里的数值是 2，那么，只有特权级别为 0、1 和 2 的程序才能访问该段，而特权级为 3 的程序访问该段时，处理器会予以阻止。特权级将在以后专门讲解，谁也不希望自己的特权级最低，何况现在有随便决定段特权级别的自由。那么，好吧，我们现在一律将特权级设定为最高的 0。

P 是段存在位（Segment Present）。P 位用于指示描述符所对应的段是否存在。一般来说，描述符所指示的段都位于内存中。但是，当内存空间紧张时，有可能只是建立了描述符，对应的内存空间并不存在，这时，就应当把描述符的 P 位清零，表示段并不存在。另外，同样是在内存空间紧张的情况下，会把很少用到的段换出到硬盘中，腾出空间给当前急需内存的程序使用（当前正在执行的），这时，同样要把段描述符的 P 位清零。当再次轮到它执行时，再装入内存，然后将 P 置位 1。

P 位是由处理器负责检查的。每当通过描述符访问内存中的段时，如果 P 位是“0”，处理器就会产生一个异常中断。通常，该中断处理过程是由操作系统提供的，该处理过程的任务是负责将该段从硬盘换回内存，并将 P 置位 1。在多用户、多任务的系统中，这是一种常用的虚拟内存调度策略。当内存很小，运行的程序很多时，如果计算机的运行速度变慢，并伴随着繁忙的硬盘操作时，说明这种情况正在发生。

D/B 位是“默认的操作数大小”（Default Operation Size）或者“默认的堆栈指针大小”（Default Stack Pointer Size），又或者“上部边界”（Upper Bound）标志。

设立该标志位，主要是为了能够在 32 位处理器上兼容运行 16 位保护模式的程序。尽管这种程序现在已经非常罕见了，但它毕竟存在过，兼容，这是 Intel 公司能够兴旺发达的重要因素。

该标志位对不同的段有不同的效果。对于代码段，此位称做“D”位，用于指示指令中默认的偏移地址和操作数尺寸。D=0 表示指令中的偏移地址或者操作数是 16 位的；D=1，指示 32 位的偏移地址或者操作数。

举个例子来说，如果代码段描述符的 D 位是 0，那么，当处理器在这个段上执行时，将使用 16 位的指令指针寄存器 IP 来取指令，否则使用 32 位的 EIP。

对于堆栈段来说，该位被叫做“B”位，用于在进行隐式的堆栈操作时，是使用 SP 寄存器还是 ESP 寄存器。隐式的堆栈操作指令包括 push、pop 和 call 等。如果该位是“0”，在访问那个段时，使用 SP 寄存器，否则就是使用 ESP 寄存器。同时，B 位的值也决定了堆栈的上部边界。如果 B=0，那么堆栈段的上部边界（也就是 SP 寄存器的最大值）为 0xFFFF；如果 B=1，那么堆栈段的上部边界（也就是 ESP 寄存器的最大值）为 0xFFFFFFFF。

对于本书来说，它应当为“1”。本书不过多涉及 16 位保护模式，它已经非常罕见了。

L 位是 64 位代码段标志（64-bit Code Segment），保留此位给 64 位处理器使用。目前，我们将此位置“0”即可。

TYPE 字段共 4 位，用于指示描述符的子类型，或者说是类别。如表 11-1 所示，对于数据段来说，这 4 位分别是 X、E、W、A 位；而对于代码段来说，这 4 位则分别是 X、C、R、A 位。

表 11-1 代码段和数据段描述符的 TYPE 字段

X	E	W	A	描述符类别	含义
0	0	0	×	数据	只读
0	0	1	×		读、写
0	1	0	×		只读，向下扩展
0	1	1	×		读、写，向下扩展
X	C	R	A	描述符类别	含义
1	0	0	×	代码	只执行

1	0	1	×		执行、读
1	1	0	×		只执行，依从的代码段
1	1	1	×		执行、读，依从的代码段

表 11-1 中，X 表示是否可以执行（eXecutable）。数据段总是不可执行的，X=0；代码段总是可以执行的，因此，X=1。

对于数据段来说，E 位指示段的扩展方向。E=0 是向上扩展的，也就是向高地址方向扩展的，是普通的数据段；E=1 是向下扩展的，也就是向低地址方向扩展的，通常是堆栈段。W 位指示段的读写属性，或者说段是否可写，W=0 的段是不允许写入的，否则会引发处理器异常中断；W=1 的段是可以正常写入的。

对于代码段来说，C 位指示段是否为特权级依从的（Conforming）。C=0 表示非依从的代码段，这样的代码段可以从与它特权级相同的代码段调用，或者通过门调用；C=1 表示允许从低特权级的程序转移到该段执行。关于特权级和特权级检查的知识将在第 14 章介绍。R 位指示代码段是否允许读出。代码段总是可以执行的，但是，为了防止程序被破坏，它是不能写入的。至于是否有读出的可能，由 R 位指定。R=0 表示不能读出，如果企图去读一个 R=0 的代码段，会引发处理器异常中断；如果 R=1，则代码段是可以读出的，即可以把这个段的内容当成 ROM 一样使用。

也许有人会问，既然代码段是不可读的，那处理器怎么从里面取指令执行呢？事实上，这里的 R 属性并非用来限制处理器，而是用来限制程序和指令的行为。一个典型的例子是使用段超越前缀“CS:”来访问代码段中的内容。

数据段和代码段的 A 位是已访问（Accessed）位，用于指示它所指向的段最近是否被访问过。在描述符创建的时候，应该清零。之后，每当该段被访问时，处理器自动将该位置“1”。对该位的清零是由软件（操作系统）负责的，通过定期监视该位的状态，就可以统计出该段的使用频率。当内存空间紧张时，可以把不经常使用的段退避到硬盘上，从而实现虚拟内存管理。

AVL 是软件可以使用的位（Available），通常由操作系统来用，处理器并不使用它。如果你把它理解成“好吧，该安排的都安排了，最后多出这么一位，不知道干什么用好，就给软件用吧”，我也不反对，也许 Intel 公司也不会说些什么。

11.4 安装存储器的段描述符并加载 GDTR

现在开始安装各个描述符，让我们回到代码清单 11-1。

不要忘了，我们现在还处于实模式下。因此，在 GDT 中安装描述符，必须将 GDT 的线性地址转换成段地址和偏移地址。第 12 行，将 GDT 线性基地址的低 16 位传送到寄存器 AX 中。和从前一样，这里使用了段超越前缀“cs:”，表明是访问代码段中的数据；又因为主引导程序的实际加载位置是逻辑地址 0x0000:0x7c00，故标号 gdt_base 处的偏移地址是 gdt_base+0x7c00。

同样地，第 13 行将 GDT 线性基地址的高 16 位传送到寄存器 DX。

第 14~17 行将线性基地址转换成逻辑地址，方法是将 DX:AX 除以 16，得到的商是逻辑段地址，余数是偏移地址。接着，将 AX 中的逻辑段地址传送到数据段寄存器 DS 中，将偏移地址传送到寄存器 BX 中。

处理器规定，GDT 中的第一个描述符必须是空描述符，或者叫哑描述符或 NULL 描述符，相信后者对于有 C 语言经历的读者来说更容易接受。

很多时候，寄存器和内存单元的初始值会为 0，再加上程序设计有问题，就会在无意中用全 0 的索引来选择描述符。因此，处理器要求将第一个描述符定义成空描述符。

为此，第 20、21 行将两个全 0 的双字分别写入偏移地址为 BX 和 BX+4 的地方。

进入保护模式之后必然要从一个代码段开始执行。现在就来定义代码段描述符。

第 24、25 行，接着安装代码段描述符，该描述符的低 32 位是 0x7c0001ff，高 32 位是 0x00409800。结合图 11-4 可以分析出，该段的基本情况为：

线性基地址为 0x00007C00。

段界限为 0x001FF，粒度为字节 (G=0)。该段的长度为 512 字节。

属于存储器的段 (S=1)。

这是一个 32 位的段 (D=1)。

该段目前位于内存中 (P=1)。

段的特权级为 0 (DPL=00)。

这是一个只能执行的代码段 (TYPE=1000)。

很明显，该描述符所指向的段，就是现在正在执行的主引导程序所在的区域。如图 11-5 所示，这是描述符各字节在内存中的映象。Intel 处理器是低端字节序的，所以低双字在低地址端，高双字在高地址端；低字在低地址端，高字在高地址端；低字节在低地址端，高字节在高地址端。

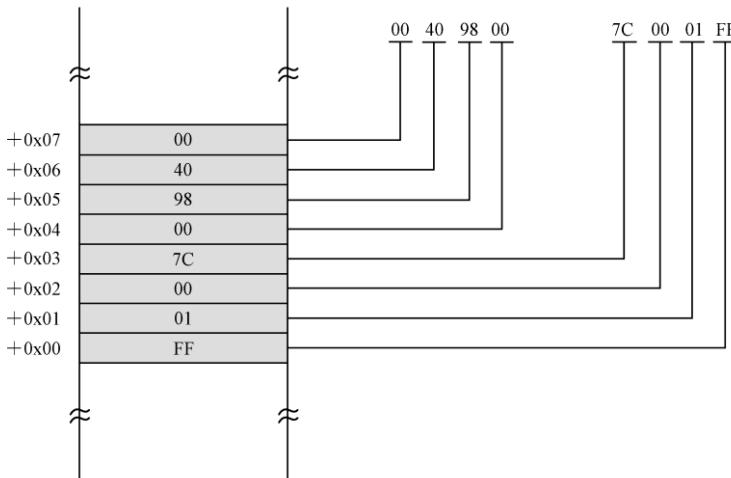


图 11-5 描述符各个字节在内存中的映象

第 28、29 行，用于安装一个数据段的描述符。对照图 11-4，很明显，这个段具有以下性质：

线性地址为 0x000B8000。

段界限为 0xFFFF, 粒度为字节 (G=0)。即，该段的长度为 64KB。

属于存储器的段 (S=1)。

这是一个 32 位的段 (D=1)。

该段目前位于内存中 (P=1)。

段的特权级为 0 (DPL=00)。

这是一个可读可写、向上扩展的数据段 (TYPE=0010)。

在屏幕上显示内容已经不是一次两次了，很容易看出，线性地址 0x000B8000 就是显存的起始地址，看起来，我们要用这个段来显示字符。

第 32、33 行，用于安装堆栈段的描述符。对照图 11-4，该段的性质如下：

线性基址为 0x00000000。
段界限为 0x07A00，粒度为字节 (G=0)。
属于存储器的段 (S=1)。
这是一个 32 位的段 (D=1)。
该段目前位于内存中 (P=1)。
段的特权级为 0 (DPL=00)。
这是一个可读可写、向下扩展的数据段，即堆栈段 (TYPE=0010)。

在这里，段界限的值 0x07a00 加上 1 (0x07a01)，就是 ESP 寄存器所允许的最小值。当执行 push、call 这样的隐式堆栈操作时，处理器会检查 ESP 寄存器的值，一旦发现它小于等于这里指定的数值，会引发异常中断。关于堆栈界限的讨论将在本章的后面接着进行。

好了，现在所有的描述符都已经安装完毕，接下来的工作是加载描述符表的线性基地址和界限到 GDTR 寄存器，这要使用 lgdt 指令，该指令的格式为

```
lgdt m48      ;lgdt m16&m32
```

这就是说，该指令的操作数是一个内存地址，指向一个包含了 48 位 (6 字节) 数据的内存区域。在 16 位模式下，该地址是 16 位的；在 32 位模式下，该地址是 32 位的。该指令在实模式和保护模式下都可以执行。

在这 6 字节的内存区域中，前 16 位是 GDT 的界限值，高 32 位是 GDT 的基址。在初始状态下（计算机启动之后），GDTR 的基址被初始化为 0x00000000；界限值为 0xFFFF。

该指向不影响任何标志位。

为此，代码清单 11-1 第 36 行，将 GDT 表的界限值 31 写入标号 gdt_size 所在的内存单元。这里共有 4 个描述符（包括空描述符），每个描述符占 8 字节，一共是 32 字节。GDT 表的界限值是表的总字节数减去一，所以是 31。

接着，第 38 行，把从标号 gdt_size 开始的 6 字节加载到 GDTR 寄存器。注意，到目前为止，我们依然工作在实模式下，而且不要忘了，指令中的偏移地址都要加上 0x7c00。

11.5 关于第 21 条地址线 A20 的问题

在即将进入保护模式之前，这里还涉及一个历史遗留问题，那就是处理器的第 21 根地址线，编号 A20。“A”是 Address 的首字符，就是地址，A0 是第一根地址线，A31 是第 32 根地址线，所以，A20 就是第 21 根地址线。在 8086 处理器上运行程序不存在 A20 问题，因为它只有 20 根地址线。

实模式下的程序只能寻址 1MB 内存，那是因为它依赖 16 位的段地址左移 4 位，加上 16 位的偏移地址来访问内存。当逻辑段地址达到最大值 0xFFFF 时，再加一，就会因进位而绕回到 0x0000，因为段寄存器只能保留 16 位的结果。至于段内偏移地址，也是如此。

这个问题可以从另一个角度来解释得更清楚一点。无论如何，从 8086 处理器外部来看，每次当物理地址达到最高端 0xFFFF 时，再加一，结果为 0x100000。但因为它只能维持 20 位的地址，故进位自然丢失，地址又绕回最低地址端 0x000000。程序员，你是知道的，他们喜欢钻研，更喜欢利用硬件的某些特性来展示自己的技术，很难说在当年有多少程序在依赖这个回绕特性工作着。

到了 80286 时代，处理器有 24 条地址线，地址回绕好象不灵了，因为比 0xFFFF 大的数是 0x100000，80286 处理器可以维持 24 位的地址数据，进位不会被丢弃。那个时代，正是商业机器公司 IBM 生意火红的时候，主导着个人计算机市场。为了能在 80286 处理器上运行 8086 程序而不会因地址线而产生问题，它们决定在主板上动一动手脚。

其实问题的解决办法很简单，只需要强制第 21 根地址线恒为“0”就可以了。这样，0xFFFF 加 1 的进位被强制为“0”，结果是 0x000000；再加 1，是 0x0000001，……，永远和实模式一样。

于是，如图 11-6 所示，IBM 公司使用一个与门来控制第 21 根地址线 A20，并把这个与门的控制阀门放在键盘控制器内，端口号是 0x60。向该端口写入数据时，如果第 1 位是“1”，那么，键盘控制器通向与门的输出就为“1”，与门的输出就取决于处理器 A20 是“0”还是“1”。

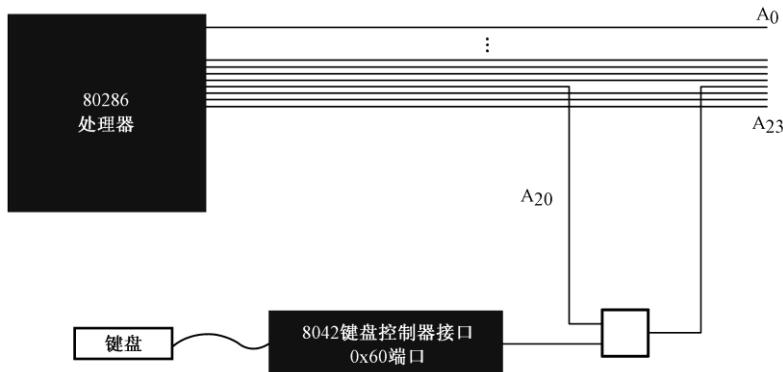


图 11-6 早期的 A20 控制策略

不过，这种做法非常烦琐，因为要访问键盘控制器，需要先判断状态，要等待键盘控制器不忙，至少需要十几个步骤，需要的指令数量比本章的代码清单 11-1 还多。

这种做法持续了若干年，直到 80486 处理器推出后，才有了更快速的办法。相信在此期间，Intel 公司和 IBM 公司都听到了不少的抱怨，为什么进入保护模式这么麻烦，一定要改改。从 80486 处理器开始，处理器本身就有了 A20M#引脚，意思是 A20 屏蔽（A20 Mask），它是低电平有效的。

如图 11-7 所示，输入输出控制器集中芯片 ICH 的处理器接口部分，有一个用于兼容老式设备的端口 0x92，第 7~2 位保留未用，第 0 位叫做 INIT_NOW，意思是“现在初始化”，用于初始化处理器，当它从 0 过渡到 1 时，ICH 芯片会使处理器 INIT#引脚的电平变低（有效），并保持至少 16 个 PCI 时钟周期。通俗地说，向这个端口写 1，将会使处理器复位，导致计算机重新启动。

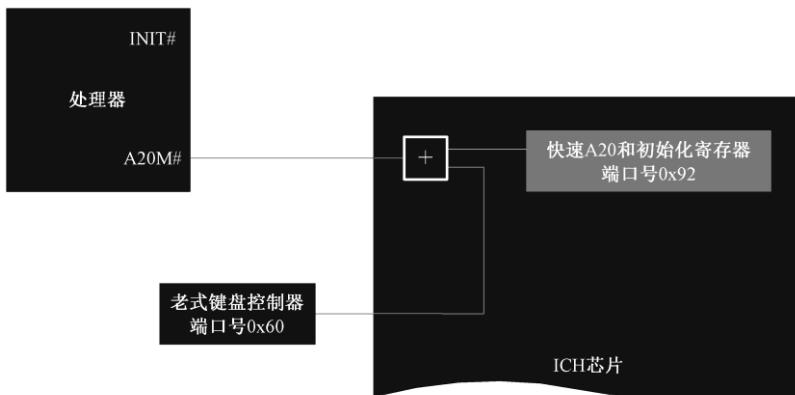


图 11-7 改进后的 A20 控制策略

端口 0x92 的位 1 用于控制 A20，叫做替代的 A20 门控制(Alternate A20 Gate, ALT_A20_GATE)，它和来自键盘控制器的 A20 控制线一起，通过或门连接到处理器的 A20M#引脚。和使用键盘控制器的端口不同，通过 0x92 端口显得非常迅速，也非常方便快捷，因此称为 Fast A20。

当 INIT_NOW 从 0 过渡到 1 时，ALT_A20_GATE 将被置“1”。这就是说，计算机启动时，第 21 根地址线是自动启用的。A20M#信号仅用于单处理器系统，多核处理器一般不用。特别是考虑到传统的键盘控制器正逐渐被 USB 键盘代替，这些老式设备也许很快就会消失。

接着来看代码清单 11-1。

端口 0x92 是可读写的，第 40~42 行，先从该端口读出原数据，接着，将第 2 位（位 1）置“1”，然后再写入该端口，这样就打开了 A20。

11.6 保护模式下的内存访问

一路披荆斩棘之后，你已经到达实模式和保护模式的分界线了。同时，你也会发现，控制这两种模式切换的开关原是在一个叫 CR0 的寄存器。

CR0 是处理器内部的控制寄存器（Control Register, CR）。之所以有个“0”后缀，是因为还有 CR1、CR2、CR3 和 CR4 控制寄存器，甚至还有 CR8。

CR0 是 32 位的寄存器，包含了一系列用于控制处理器操作模式和运行状态的标志位。如图 11-8 所示，它的第 1 位（位 0）是保护模式允许位（Protection Enable, PE），是开启保护模式大门的门把手，如果把该位置“1”，则处理器进入保护模式，按保护模式的规则开始运行。你可能会问，为什么只标识了一个 PE 位，还把图画那么大。很简单，随着讲解的深入，我们还要接触其他标志位，把图的比例画得一致更好一些。



图 11-8 控制寄存器 CR0 的 PE 位

保护模式下的中断机制和实模式不同，因此，原有的中断向量表不再适用，而且，必须要知道的是，在保护模式下，BIOS 中断都不能再用，因为它们是实模式下的代码。在重新设置保护模式下的中断环境之前，必须关中断，这就是第 44 行的用意。

第 46 行，将 CR0 寄存器中的原有内容传送到寄存器 EAX，准备修改它；第 47 行，将它的第 1 位（位 0）置“1”，其他各位保持原来的状态不变；第 48 行，将修改之后的内容重新写回 CR0，这直接导致处理器的运行变成保护模式。

我们知道，在实模式下，处理器访问内存的方式是将段寄存器的内容左移 4 位，再加上偏移地址，以形成 20 位的物理地址。

8086 处理器的段寄存器是 16 位的，共有 4 个：CS、DS、ES 和 SS。而在 32 位处理器内，段寄存器是 80 位的（16 位段选择器和 64 位描述符高速缓存器）。而且，在原先的基础上又增加了两个段寄存器 FS 和 GS。

如图 11-9 所示，32 位处理器的这 6 个段寄存器又分为两部分，前 16 位和 8086 相同，在实模式下，它们用于按传统的方式寻址 1MB 内存，使用方法也没有变化，所以使得 8086 的程序可以继

续在 32 位处理器上运行。同时，每个段寄存器还包括一个不可见的 64 位部分，称为描述符高速缓存器，用来存放段的线性基地址、段界限和段属性。既然不可见，那就是处理器不希望我们访问它。事实上，我们也没有任何办法来访问这些不可见的部分，它是由处理器内部使用的。

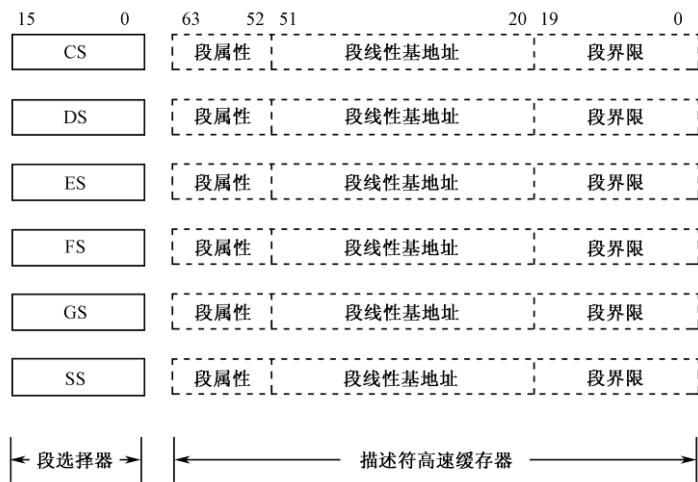


图 11-9 32 位处理器内的段寄存器

在实模式下，访问内存用的是逻辑地址，即将段地址乘以 16，再加上偏移地址。下面是一个例子：

```
mov cx, 0x2000
mov ds, cx
mov [0xc0], al
mov cx, 0xb800
mov ds, cx
mov [0x02], ah
```

以上，首先将段寄存器 DS 的内容置为 0x2000，这是逻辑段地址。接着，向该段内偏移地址为 0x00c0 的地方写入 1 字节（在寄存器 AL 中），写入时，处理器将 DS 的内容左移 4 位，加上偏移地址，实际写入的物理地址是 0x200c0。

在 8086 处理器上，这是正确的。但是，在 32 位处理器上，这个过程稍有不同。首先，每当引用一个段时，处理器自动将段地址左移 4 位，并传送到描述符高速缓存器。此后，就一直使用描述符高速缓存器的内容做为段地址。所谓引用一个段，就是执行将段地址传送到段寄存器的指令。如

```
jmp 0xf000:0x5000
```

以上是引用代码段的一个例子，因为代码段的修改通常是用转移和调用指令进行的。如果是引用数据段，则一般采用以下形式：

```
mov ax, 0x2000
mov ds, ax
```

只要不改变段寄存器 DS 的内容，以后每次内存访问都直接使用 DS 描述符高速缓存器中的内容。但是，在实模式下只能向段寄存器传送 16 位的逻辑段地址，故，处理器仍然只能访问 1MB 内存。也就是说，在实模式下，段寄存器描述符高速缓存器的内容仅低 20 位有效，高 12 位全部是零。

实模式下的 6 个段寄存器 CS、DS、ES、FS、GS 和 SS，在保护模式下叫做段选择器。和实模式不同，保护模式的内存访问有它自己的方式。在保护模式下，尽管访问内存时也需要指定一个段，

但传送到段选择器的内容不是逻辑段地址，而是段描述符在描述符表中的索引号。

如图 11-10 所示，在保护模式下访问一个段时，传送到段选择器的是段选择子。它由三部分组成，第一部分是描述符的索引号，用来在描述符表中选择一个段描述符。TI 是描述符表指示器（Table Indicator）， $TI=0$ 时，表示描述符在 GDT 中； $TI=1$ 时，描述符在 LDT 中。LDT 的知识将在后面进行介绍，它也是一个描述符表，和 GDT 类似。RPL 是请求特权级，表示给出当前选择子的那个程序的特权级别，正是该程序要求访问这个内存段。每个程序都有特权级别，也将在后面慢慢介绍，现在只需要将这两位置成“00”即可。

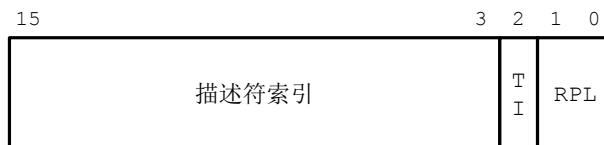


图 11-10 段选择子的组成

为了说明保护模式下的内存访问，让我们回到代码清单 11-1。前面已经创建了全局描述符表（GDT），而且在表中定义了 4 个段描述符。数据段描述符在 GDT 中的顺序是第 3 个，因为编号都是从 0 开始的，所以它的索引号（或者叫编号、槽位号）是 2。

代码清单 11-1 第 56、57 行，将描述符选择子 0x0010（二进制数 0000_0000_00010_0_00）传送到段选择器 DS 中。从选择子的二进制形式可以看出，指定的描述符索引号是 2，指定的描述符表是 GDT，请求特权级 RPL 是 00。

GDT 的线性地址在 GDTR 中，又因为每个描述符占 8 字节，因此，描述符在表内的偏移地址是索引号乘以 8。如图 11-11 所示，当处理器在执行任何改变段选择器的指令时（比如 pop、mov、jmp far、call far、iret、retf），就将指令中提供的索引号乘以 8 作为偏移地址，同 GDTR 中提供的线性地址相加，以访问 GDT。如果没有发现什么问题（比如超出了 GDT 的界限），就自动将找到的描述符加载到不可见的描述符高速缓存部分。

加载的部分包括段的线性地址、段界限和段的访问属性。在当前的例子中，线性地址是 0x000b8000，段界限是 0x0ffff，段的属性是向上扩展，可读写的数据段，粒度为字节。

此后，每当有访问内存的指令时，就不再访问 GDT 中的描述符，直接用当前段寄存器描述符高速缓存器提供线性地址。因此，第 60 行，因为指令中没有段超越前缀，故默认使用数据段寄存器 DS。如图 11-12 所示，执行这条指令时，处理器用 DS 描述符高速缓存中的线性地址加上指令中给出的偏移量 0x00，形成 32 位物理地址 0x000b8000，并将字符“P”的 ASCII 码写入该处。

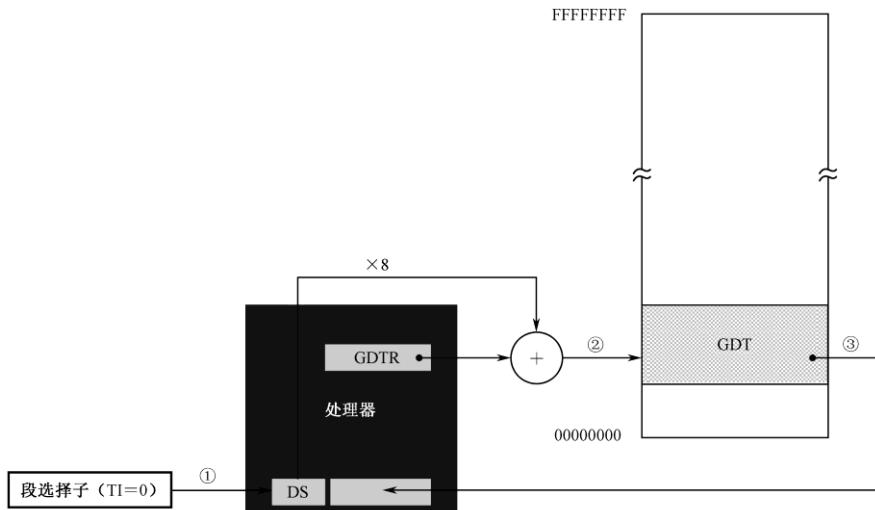


图 11-11 段选择器和描述符高速缓存器的加载过程

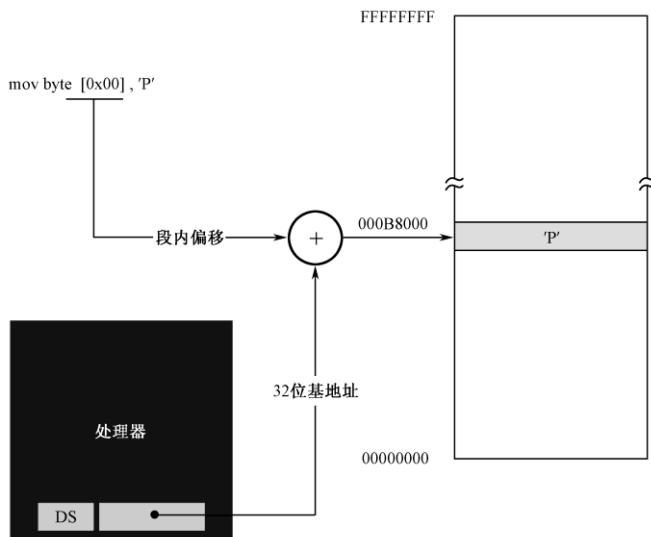


图 11-12 保护模式下的内存访问示意图

不单单是访问数据段，即使是处理器取指令执行时，也采用了相同的方法。如图 11-13 所示，在 32 位保护模式下，处理器使用的指令指针寄存器是 EIP。假设已经从描述符表中选择了一个段描述符，CS 描述符高速缓存器已经装载了正确的 32 位线性基址，那么，当处理器取指令时，会自动用描述符高速缓存器中的 32 位线性基址加上指令指针寄存器 EIP 中的 32 位偏移量，形成 32 位物理地址，从内存中取得执令并加以执行。同时，EIP 的内容自动增加以指向下一条指令。当前指令执行完毕之后，处理器接着按上述方式取下一条指令加以执行。

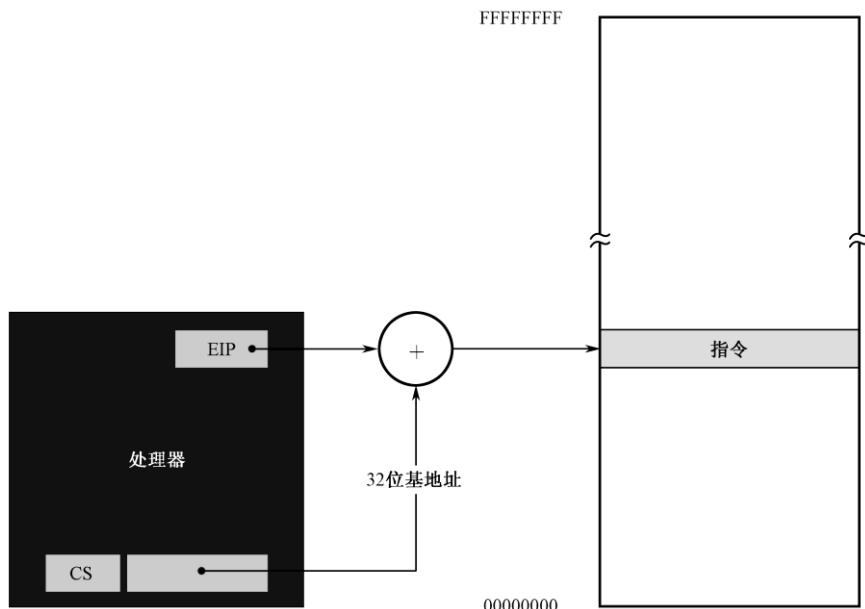


图 11-13 保护模式下处理器取指令的过程示意图

11.7 清空流水线并串行化处理器

看起来我们所讲的内容有些超前了，毕竟前面刚刚设置了控制寄存器 CR0 的 PE 位，处理器刚刚切换到保护模式下。看起来一切都很简单，拧一下钥匙，汽车就发动了，不是吗？

不是这样的。这里有两个亟待解决的问题。

第一，正如上一节所述，即使是在实模式下，段寄存器的描述符高速缓存器也被用于访问内存，仅低 20 位有效，高 12 位是全零。当处理器进入保护模式后，不影响段寄存器的内容和使用，它们依然是有效的，程序可以继续执行。但是，在保护模式下，对段的解释是不同的，处理器会把段选择器里的内容看成是描述符选择子，而不是逻辑段地址。因此，比较安全的做法是尽快刷新 CS、SS、DS、ES、FS 和 GS 的内容，包括它们的段选择器和描述符高速缓存器。

第二，在进入保护模式前，有很多指令已经进入了流水线。因为处理器工作在实模式下，所以它们都是按 16 位操作数和 16 位地址长度进行译码的，即使是那些用 bits 32 编译的指令。进入保护模式后，由于对段地址的解释不同，对操作数和默认地址大小的解释也不同，有些指令的执行结果可能会不正确，所以必须清空流水线。同时，那些通过乱序执行得到的中间结果也是无效的，必须清理掉，让处理器串行化执行，即，重新按指令的自然顺序执行。

怎么办呢？这里有一个两全其美的方案，那就是使用 32 位远转移指令 jmp 或者远过程调用指令 call。处理器最怕转移指令，遇到这种指令，一般会清空流水线，并串行化执行；另一方面，远转移会重新加载段选择器 CS，并刷新描述符高速缓存器中的内容。唯一的问题是，这条指令必须在 bits 16 下编译，使得处理器能够在 16 位模式下正确译码；同时，还必须编译成 32 位操作数的指令，使处理器在刚进入保护模式时能正确执行。

一个建议的方法是在设置了控制寄存器 CR0 的 PE 位之后，立即用 jmp 或者 call 转移到当前指令流的下一条指令上。为此，代码清单 11-1 第 51 行，用 32 位远转移指令来转移到紧挨着当前指令

的下一条指令：

```
jmp dword 0x0008:flush
```

这条指令是用“bits 16”编译的，而且使用了关键字“dword”，该关键字修饰偏移地址，意思是要求使用32位的偏移量。因此，会有指令前缀0x66，编译之后的结果是

```
66 EA 80 00 00 00 08 00
```

如果去掉以上机器码中的前缀0x66，它对应着一条32位指令（用bits 32编译）

```
jmp 0x0008:flush
```

可见，本质上，这两条指令的机器码是相同的。

因为处理器实际上是在保护模式下执行该指令的，因此，它会重新解释这条指令的含义。我们知道，操作数的默认大小（16位还是32位）是由描述符的D位决定的，确切地说，是由段寄存器的描述符高速缓存器中的D位决定的，毕竟，要访问一个段，必须首先将它的描述符传送到段寄存器的描述符高速缓存器中。当它刚进入保护模式时，CS的描述符高速缓存器依然保留着实模式时的内容，其D位是“0”，因此，在那个时刻，处理器运行在16位保护模式下。

因为处理器已经进入保护模式，所以，0x0008不再是逻辑段地址，而是保护模式下的段描述符选择子。在前面定义GDT的时候，它的第2个（1号）描述符对应着保护模式下的代码段。因此，其选择子为0x0008（索引号为1，TI位是0，RPL为00）。当指令执行时，处理器加载段选择器CS，从GDT中取出相应的描述符加载到CS描述符高速缓存。

保护模式下的代码段，基地址为0x00007c00，段界限为0x1ff，长度为0x200，正好对应着当前程序在内存中的区域。在这种情况下，上面那条指令执行时，目标位置在段内的偏移量就是标号flush的汇编地址，处理器用它的数值来代替指令指针寄存器EIP的原有内容。

在16位保护模式下执行带前缀0x66的指令，那么，很好，处理器会按32位的方式执行，使用32位的偏移量。于是，它将0x0008加载到CS选择器，并从GDT中取出对应的描述符，加载CS描述符高速缓存器；同时，把指令中给出的32位偏移量传送到指令指针寄存器EIP。很自然地，处理器就从新的位置开始取指令执行了。

从进入保护模式开始，之后的指令都应当是按32位操作数方式编译的。因此，第53行，使用了伪指令[bits 32]。当处理器执行到这里时，它会按32位模式进行译码，这正是我们所希望的。

代码清单11-1第56~74行，用于把描述符选择子0x10加载到段选择器DS，并自动加载描述符高速缓存器。因为该数据段实际上是文本模式下的显示缓冲区，故大部分指令都用于在屏幕上显示字符串“Protect mode OK”。保护模式下的数据段访问已经在上一节里讨论过了，这里不再赘述。另外，处理器模式的变化对外围设备没有影响，它们是无法感知的，而且只按自己的方式工作。

11.8 保护模式下的堆栈

11.8.1 关于堆栈段描述符中的界限值

第77~79行用于初始化保护模式下的堆栈。堆栈段描述符是GDT中的第4个（3号）描述符，堆栈的32位线性基地址是0x00000000，段界限为0x07a00，粒度为字节，属于可读可写、向下扩

展的数据段。

堆栈是向下扩展的，因此，描述符中的段界限，和向上扩展的段含义不同。对于向上扩展的段，段内偏移量是从 0 开始递增，偏移量的最大值是界限值和粒度的乘积；而对于向下扩展的段来说，因为它经常用做堆栈段，而堆栈是从高地址向低地址方向推进的，故段内偏移量的最小值是界限值和粒度的乘积加一。在 32 位代码中，是用 ESP 作为堆栈指针的。因此，这里的段界限，用来和段粒度一起，决定 ESP 寄存器所能具有的最小值。即，堆栈操作时，必须符合条件：

$$\text{ESP} > \text{段界限} \times \text{粒度值}$$

对于描述符中 G 位是“0”的段来说，粒度值是 1（字节）；而对于 G 位是“1”的段来说，粒度值是 4096（4KB）。

在当前代码中，ESP 寄存器的内容被初始化为 0x00007c00。假如此时执行以下指令：

```
push edx
```

那么，因为要压入一个 32 位数，所以处理器先将 ESP 的内容减去 4，再压入数据。此时，ESP 寄存器的内容为（扩展到 32 位）：

$$0x00007C00 - 4 = 0x00007BFC$$

在当前堆栈段的描述符中，段界限为 0x07a00，粒度是字节，故作为堆栈的界限，实际使用的数值是（扩展到 32 位）：

$$0x07A00 \times 1 = 0x00007A00$$

对于堆栈段来说，段界限的值加一，就是段内偏移量的最小值。因为要访问的段内偏移量 0x00007BFC 大于实际使用的段界限值 0x00007A00，故处理器允许执行该操作，并用描述符高速缓存中的 32 位基地址 0x00000000 加上这里的偏移量 0x00007BFC，共同形成 32 位线性地址访问堆栈，将寄存器 EDX 的内容压入。否则，处理器阻止当前操作，引发一个异常中断。

你可能觉得当前的堆栈段很完美。但不得不说，这是一个非常糟糕的堆栈定义。结合本章的程序，很明显，我们的本意是要定义一个只有 512 字节的堆栈空间，从物理地址 0x00007A00 开始，到物理地址 0x00007C00 结束，如图 11-14 所示。

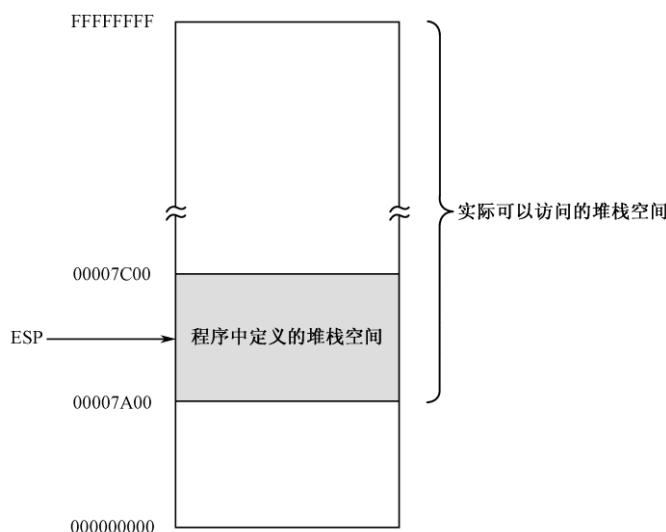


图 11-14 堆栈段的界限和堆栈的安全访问

尽管我们的本意是定义一个只有 512 个字节的堆栈，但是，从该段的描述符来看，这个段的空间却是非常巨大的。假如一切正常，特别是指令执行正常，那不会有什么问题。但是，在

程序失控的情况下，ESP 的内容可能会是任何预料不到的值，比如 0xFFFFFFFF。即使是这样，它也是合法的值，毕竟它大于 0x00007A00。因为当前堆栈段的线性基地址为 0x00000000，所以，实际可以访问的空间是从物理地址 0xFFFFFFFF 到 0x00007A00。显然，这超出了我们的预期。在下一章里，我们将继续讨论如何用更好的方法来创建堆栈。

11.8.2 检验 32 位下的堆栈操作

代码清单 11-1 中，最后的指令用于演示保护模式下的堆栈操作。我们已经知道，对于存储器的段来说，其描述符的 D/B 位，对于代码段来说，是 D 位；对于堆栈段来说，是 B 位。

隐式的堆栈操作（push、pop、call、ret 和 iret）涉及两个段：一个是指令所在的代码段；另一个是指令执行时，所使用的堆栈段。正如上一章所述，16 位下的堆栈操作，其默认的操作数大小是 16 位的，而且使用的堆栈指针寄存器是 SP；32 位下的隐式堆栈操作，其默认的操作数大小是 32 位的，使用 ESP 寄存器。

在本章里，当前程序的代码段，其描述符的 D 位是“1”，所以，当进行隐式的堆栈操作时，默认地，每次压栈操作时，压入的是双字；当前程序所使用的堆栈段，其描述符的 B 位也是“1”，默认地，使用堆栈指针寄存器 ESP 进行操作。为此，从第 81 行开始的指令用于检验这个事实。

因此，第 81 行，先保存当前堆栈指针的内容到 EBP 寄存器；接着，第 82 行，向堆栈中压入立即数。该立即数为字符“.”的 ASCII 码，这个值是在编译阶段计算的。

因为当前正在执行的代码段是 32 位的，其描述符的 D 位是“1”，故 push 指令默认的操作数大小是 32 位。正如在上一章里所讲的，关键字“byte”仅仅是给编译器用的，告诉它，该指令对应的格式为 push imm8，必须使用操作码 0x6A，而不是用来在编译后的机器指令前添加指令前缀。因此，该指令实际在处理器上执行时，压入堆栈中的是一个双字，也就是 4 字节，高 24 位是该字节符号的扩展。

当前指令执行时，所访问的堆栈，其描述符的 B 位也是“1”，故处理器在进行堆栈操作时，用的是 32 位堆栈指针寄存器 ESP。它首先将 ESP 的内容减去 4，再写入数值，数据保存的位置是 SS:ESP。

现在，理论上，将 EBP 的内容减去 4 之后，应该和 ESP 的内容相同。为了证实这一点，第 84~86 行，将原先保存的 EBP 内容减去 4，再和现行的 ESP 比较，看是否相等。如果相等，则立即将刚才压入的字符出栈，并显示在前面的字符串后。不存在从堆栈中弹出字节的指令，因为名义上可以压入字节，但实际上它们是作为 16 位或者 32 位有符号数压入的。

当然，如果经过验证，EBP 和 ESP 不相等，那么，将不会显示句点，直接转移到程序的最后，执行停机指令。因为现在已经禁止了中断，故除了 NMI，没有任何原因会导致处理器被激活。

11.9 程序的编译和运行

编译代码清单 11-1，生成二进制文件 c11_mbr.bin。用 FixVhdWr 工具将此文件写入虚拟硬盘的主引导扇区，然后启动虚拟机，如果没有问题的话，显示的结果应当如图 11-15 所示。

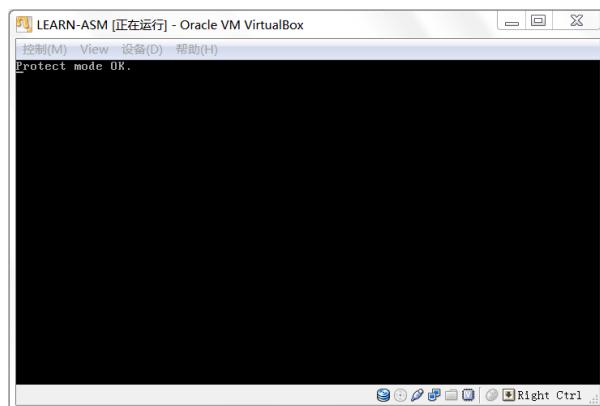


图 11-15 本章程序的运行结果

本 章 习 题

在我的计算机上，创建虚拟机时指定的内存容量是 64MB。因此，实际有效的物理地址范围是 0x00000000~0x03FFFFFF。我发现，如果将代码清单 11-1 的第 79 行改成以下指令，程序工作正常，能显示句点：

```
mov esp,0x04000003
```

但是，如果改成

```
mov esp,0x04000004
```

就不能显示句点。请问这是什么原因。注意，处理器在访问内存时，并不检验内存单元的有效性。

第 12 章 存储器的保护

处理器引入保护模式的目的是提供保护功能，其中很重要的一个方面就是存储器保护。存储器的保护功能可以禁止程序的非法操作，比如，向代码段写入数据、访问段界限之外的内存位置等。很多时候，这类问题都是由于编程疏漏引起的，属于有缺陷的软件，但也不排除软件的功能本身就是恶意的。不过，一旦能够及时发现和禁止这些非法操作，在程序失去控制之前引发异常中断，就可以提高软件的可靠性，降低整个计算机系统的安全风险。

凡事都有两面。利用存储器的保护功能，也可以实现一些有价值的功能，比如虚拟内存管理。当处理器访问一个实际上不存在的段时，会引发异常中断。操作系统可以利用这一点，通过硬盘来进行段的换入和换出，从而实现在较小的内存空间运行尽可能大、尽可能多的程序。

下面，我们通过实例来看一看，处理器是如何进行存储器的保护的。

12.1 代码清单 12-1

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：12-1（主引导扇区程序）

源程序文件：[c12_mbr.asm](#)

12.2 进入 32 位保护模式

12.2.1 话说 mov ds,ax 和 mov ds,eax

本章的代码和上一章有几分类似，但实质上有很大区别。

我们知道，段寄存器（选择器）的值只能用内存单元或者通用寄存器来传送，一般的指令格式为

```
mov sreg,r/m16
```

这里有一个常见的例子：

```
mov ds,ax
```

在 16 位模式下，传送到 DS 中的值是逻辑段地址；在 32 位保护模式下，传送的是段描述符的选择子。无论传送的是什么，这都不重要，重要的是，在 16 位模式和 32 位模式下，一些老式的编译器会生成不同的机器代码。下面是一个例证：

```
[bits 16]
mov ds,ax          ;8E D8

[bits 32]
mov ds,ax          ;66 8E D8
```

由于在 16 位模式下，默认的操作数大小是字（2 字节），故生成 8E D8 也不难理解。在 32 位模式下，默认的操作数大小是双字（4 字节）。由于指令中的源操作数是 16 位的 AX，故编译后的机器码前面应当添加前缀 0x66 以反转默认的操作数大小，即 66 8E D8。

很遗憾，由于这一点点区别，有前缀的和没有前缀的相比，处理器在执行时会多花一个额外的时钟周期。问题在于，这样的指令用得很频繁，而且牵扯到内存段的访问，自然也很重要。因此，它们在 16 位模式和 32 位模式下的机器指令被设计为相同。即都是 8E D8，不需要指令前缀。

这可难倒了很多编译器，它们固执地认为，在 32 位模式下，源操作数是 16 位的寄存器 AX 时，应当添加指令前缀。好吧，为了照顾它们，很多程序员习惯使用这种看起来有点别扭的形式：

```
mov ds, eax
```

你别说，还真有效，果然生成的是不加前缀的 8E D8。

说到这里，我觉得 NASM 编译器还是非常优秀的，起码它不会有这样的问题。因此，不管处理器模式如何变化，也不管指令形式如何变化，以下代码编译后的结果都一模一样：

```
[bits 16]
mov ds,ax          ;8E D8
mov ds,eax         ;8E D8

[bits 32]
mov ds,ax          ;8E D8
mov ds,eax         ;8E D8
```

和这个示例一样，其他从通用寄存器到段寄存器的传送也符合这样的编译规则。因此，代码清单 12-1 第 7、8 行，用于通过寄存器 EAX 来初始化堆栈段寄存器 SS。

12.2.2 创建 GDT 并安装段描述符

准备进入保护模式。

首先是创建 GDT，并安装刚进入保护模式时就要使用的描述符。第 12~15 行，首先计算 GDT 在实模式下的逻辑地址。在上一章里，GDT 的大小和线性基址分别是用两个标号 gdt_size 和 gdt_base 声明和初始化的：

```
gdt_size dw 0
gdt_base dd 0x0000007e00
```

但是，如后面的第 107、108 行所示，现在已经改成

```
pdgt dw 0
dd 0x00007e00
```

另外一个区别是计算 GDT 逻辑地址的方法。在 32 位处理器上，即使是在实模式下，也可以使用 32 位寄存器。所以，第 12 行，直接将 GDT 的 32 位线性基址传送到寄存器 EAX 中。

我们知道，32位处理器可以执行以下除法操作：

```
div r/m32
```

其中，64位的被除数在 EDX:EAX 中，32位被除数可以在32位通用寄存器中，也可以在32位内存单元中。因此，第13~15行，用64位的被除数 EDX:EAX 除以32位的除数 EBX。指令执行后，EAX中的商是段地址，仅低16位有效；EDX中的余数是段内偏移地址，仅低16位有效。

第17、18行，初始化段寄存器 DS，使其指向 GDT 所在的逻辑段。

第21、22行，安装空描述符。该描述符的槽位号是0，处理器不允许访问这个描述符，任何时候，使用索引字段为0的选择子来访问该描述符，都会被处理器阻止，并引发异常中断。在现实中，一个忘了初始化的指针往往默认值就是0，所以空描述符的用意就是阻止不安全的访问。很多人喜欢用这个槽位来记载一些私人信息，做一些特殊的用途，认为反正处理器也不用它。但是，这样做可能是不安全的，还没有证据表明 Intel 公司保证决不会使用这个槽位。

第25、26行，安装保护模式下的数据段描述符。参考前面的段描述符格式，可以看出，该段的线性地址位于整个内存的最低端，为0x00000000；属于32位的段，段界限是0xFFFF。但是要注意，段的粒度是以4KB为单位的。对于以4KB（十进制数4096或者十六进制数0x1000）为粒度的段，描述符中的界限值加1，就是该段有多少个4KB。因此，其实际使用的段界限为

$$(\text{描述符中的段界限值} + 1) \times 0x1000 - 1$$

将其展开后，即

$$\text{描述符中的段界限值} \times 0x1000 + 0x1000 - 1$$

因此，在换算成实际使用的段界限时，其公式为

$$\text{描述符中的段界限值} \times 0x1000 + 0xFFFF$$

这就是说，实际使用的段界限是

$$0xFFFF \times 0x1000 + 0xFFFF = 0xFFFFFFFF$$

也就是4GB。就32位处理器来说，这个地址范围已经最大了。一旦使用这个段，就可以访问0到4GB空间内的任意一个单元，这是本书开篇以来，从来没有过的事情。

第29、30行，安装保护模式下的代码段描述符。该段是32位的代码，线性地址为0x00007C00；段界限为0x001FF，粒度为字节。对于向上扩展的段来说，段界限在数值上等于段的长度减去1，因此该段的长度是0x200，即512字节。

根据上一章的经验，该段实际上就是当前程序所在的段（正在安装该描述符呢），也就是主引导程序所在的区域。尽管在描述符中把它定义成32位的段，但它实际上既包含16位代码，也包含32位代码。[bits 32]之前的代码是16位的，之后的代码是32位的。不过，在该描述符生效的时候，处理器的执行流已经位于32位代码中了。

第33、34行，安装保护模式下的数据段描述符。该段是32位的数据段，线性地址为0x00007C00；段界限为0x001FF，粒度为字节。可以看出，该描述符和前面的代码段描述符，描述和指向的是同一个段。你可能很想知道，这样做的用意何在？

参见上一章的表11-1，我们都已经知道，在保护模式下，代码段是不可写入的。所谓不可写入，并非是说改变了内存的物理性质，使得内存写不进去，而是说，通过该段的描述符来访问这个区域时，处理器不允许向里面写入数据或者更改数据。

但是，很多时候，又需要对代码段做一些修改。比如在调试程序时，需要加入断点指令int3。不管怎么样，如果需要访问代码段内的数据，只能重新为该段安装一个新的描述符，并将其定义为可读可写的数据段。这样，当需要修改代码段内的数据时，可以通过这个新的描述符来进行。

像这样，当两个以上的描述符都描述和指向同一个段时，把另外的描述符称为别名(alias)。

注意，别名技术并非仅仅用于读写代码段，如果两个程序想共享同一个内存区域，可以分别为每个程序都创建一个描述符，而且它们都指向同一个内存段，这也是别名应用的例子。

第 36、37 行，安装保护模式下的堆栈段描述符。该段的线性基地址是 0x00007C00，段界限为 0xFFFFE，粒度为 4KB。

尽管该段和代码段使用同一个线性基地址，但这不会有什问题，代码段是向上（高地址方向）扩展的，而堆栈段是向下（低地址方向）扩展的。至于段界限为 0xffffe，粒度为 4KB，我知道你可能会有某些疑问，这些事情马上就会讲到。

第 40 行，设置 GDT 的界限值为 39，因为这里共有 5 个描述符，总大小为 40 字节，界限值为 39。后面的代码用于进入保护模式，差不多和上一章相同，不再赘述。GDT 和 GDT 内的描述符，以及本章程序，它们在内存中的映象如图 12-1 所示。

12.3 修改段寄存器时的保护

随着程序的执行，经常要对段寄存器进行修改。此时，处理器在变更段寄存器以及隐藏的描述符高速缓存器的内容时，要检查其代入值的合法性。

代码清单 12-1 第 55 行，这是一条直接远转移指令：

```
jmp dword 0x0010:flush
```

这条指令会隐式地修改段寄存器 CS。

同样要修改段寄存器的指令还出现在第 59~68 行（以下粗体部分）：

```
mov eax,0x0018
mov ds,eax

mov eax,0x0008      ;加载数据段 (0:4GB) 选择子
mov es,eax
mov fs,eax
mov gs,eax

mov eax,0x0020      ;0000 0000 0010 0000
mov ss,eax
```

以上的指令涉及所有段寄存器，当这些指令执行时，处理器把指令中给出的选择子传送到段寄存器的选择器部分。但是，处理器的固件在完成传送之前，要确认选择子是正确的，并且该选择子



选择的描述符也是正确的。

在当前程序中，选择子的 TI 位都是 0，故所有的描述符都在 GDT 中。如图 12-2 所示，GDT 的基地址和界限，都在寄存器 GDTR 中。描述符在内存中的地址，是用索引号乘以 8，再和描述符表的线性基地址相加得到的，而这个地址必须在描述符表的地址范围内。换句话说，索引号乘以 8 得到的数值，必须位于描述符表的边界范围之内。换句话说，处理器从 GDT 中取某个描述符时，就要求描述符的 8 个字节都在 GDT 边界之内，也就是索引号 $\times 8 + 7$ 小于等于边界。

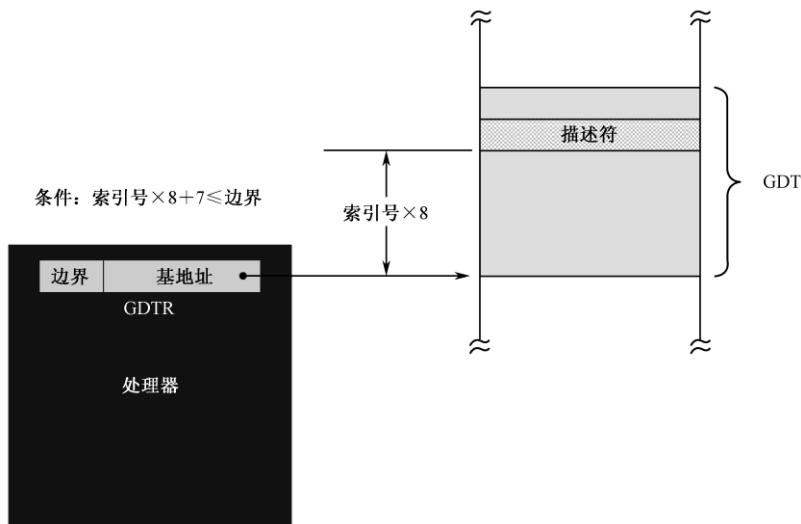


图 12-2 索引号的检查

如果检查到指定的段描述符，其位置超过表的边界时，处理器中止处理，产生异常中断 13，同时段寄存器中的原值不变。

以上仅仅是检查的第一步。要是通过了上述检查，并从表中取得描述符后，紧接着还要对描述符的类别进行确认。举个例子来说，若描述符的类别是只执行的代码段（表 11-1），则不允许加载到除 CS 之外的其他段寄存器中。

具体地说，首先，描述符的类别字段必须是有效的值，0000 是无效值的一个例子。

然后，检查描述符的类别是否和段寄存器的用途匹配。其规则如表 12-1 所示。

最后，除了按表 12-1 进行段的类别检查外，还要检查描述符中的 P 位。如果 P=0，表明虽然描述符已被定义，但该段实际上并不存在于物理内存中。此时，处理器中止处理，引发异常中断 11。一般来说，应当定义一个中断处理程序，把该描述符所对应的段从硬盘等外部存储器调入内存，然后置 P 位。中断返回时，处理器将再次尝试刚才的操作。

表 12-1 段的类别检查

段寄存器	数据段 (X=0)		代码段 (X=1)	
	只读 (W=0)	读写 (W=1)	只执行 (R=0)	执行、读 (R=1)
CS	N	N	Y	Y
DS	Y	Y	N	Y
ES	Y	Y	N	Y
FS	Y	Y	N	Y
GS	Y	Y	N	Y
SS	N	Y	N	N

如果 P=1，则处理器将描述符加载到段寄存器的描述符高速缓存器，同时置 A 位（仅限于当前讨论的存储器的段描述符）。

注意，如表中所指示的那样，可读的代码段类似于 ROM。可以用段超越前缀“cs:”来读其中的内容，也可以将它的描述符选择子加载到 DS、ES、FS、GS 来做为数据段访问。代码段在任何时候都是不可写的。

一旦上述规则全部验证通过，处理器就将选择子加载到段寄存器的选择器。显然，只有可以写入的数据段才能加载到 SS 的选择器，CS 寄存器只允许加载代码段描述符。另外，对于 DS、ES、FS 和 GS 的选择器，可以向其加载数值为 0 的选择子，即

```
xor eax,eax      ;eax = 0
mov ds, eax      ;ds <- 0
```

尽管在加载的时候不会有任何问题，但在，真正要用来访问内存时，就会导致一个异常中断。这是一个特殊的设计，处理器用它来保证系统安全，这在后面会讲到。不过，对于 CS 和 SS 的选择器来说，不允许向其传送为 0 的选择子。

继续回到代码清单 12-1 中来，第 55~68 行的指令执行之后，段寄存器 CS 指向 512 字节的 32 位代码段，基址是 0x00007C00；DS 指向 512 字节的 32 位数据段，该段是上述代码段的别名，因此基地址也是 0x00007C00；ES、FS 和 GS 指向同一个段，该段是一个 4GB 的 32 位数据段，基地址为 0x00000000；SS 指向 4KB 的 32 位堆栈段，基地址为 0x00007C00。

12.4 地址变换时的保护

12.4.1 代码段执行时的保护

在 32 位模式下，尽管段的信息在描述符表中，但是，一旦相应的描述符被加载到段寄存器的描述符高速缓存器，则处理器取指令和执行指令时，将不再访问描述符表，而是直接使用段寄存器的描述符高速缓存器，从中取得线性基地址，同指令指针寄存器 EIP 的内容相加，共同形成 32 位的物理地址从内存中取得下一条指令。不过，在指令实际开始执行之前，处理器必须检验其存放地址的有效性，以防止执行超出允许范围之外的指令。

每个代码段都有自己的段界限，位于其描述符中。实际使用的段界限，其数值和粒度（G）位有关，如果 G=0，实际使用的段界限就是描述符中记载的段界限；如果 G=1，则实际使用的段界限为

描述符中的段界限值 $\times 0x1000 + 0xFFFF$

该计算公式已经在前面出现过，不再解释。

代码段是向上（高地址方向）扩展的，因此，实际使用的段界限就是当前段内最后一个允许访问的偏移地址。当处理器在该段内取指令执行时，偏移地址由 EIP 提供。指令很有可能是跨越边界的，一部分在边界之内，一部分在边界之外，或者一条单字节指令正好位于边界上。因此，要执行的那条指令，其长度减 1 后，与 EIP 寄存器的值相加，结果必须小于等于实际使用的段界限，否则引发处理器异常。即：

$0 \leq (EIP + \text{指令长度} - 1) \leq \text{实际使用的段界限}$

在本章中，代码段描述符中给出的界限值是 0x001FF，粒度是字节，可以认为它就是段内最后

一个允许访问的偏移地址。如图 12-3 所示，在处理器取得一条指令后，EIP 寄存器的数值加上该指令的长度减 1，得到的结果必须小于等于 0x000001FF，如果等于或者超出这个数值，必然引发异常中断。

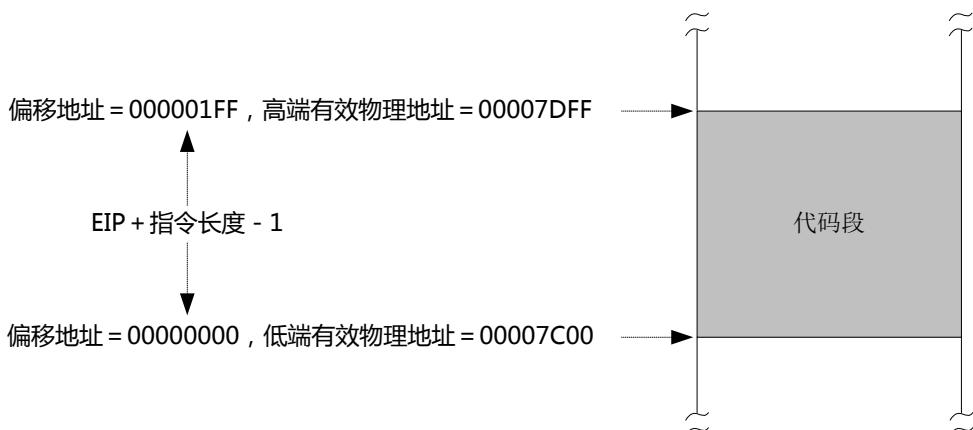


图 12-3 对代码段偏移地址的检查

做一个额外的例子，现在，假设当前代码段的粒度是 4KB，那么，因为描述符中的段界限值是 0x001FF，故实际使用的段界限是

$$0x1FF \times 0x1000 + 0xFFFF = 0x001FFFFF$$

可以认为，此数值就是当前段内最后一个允许访问的偏移地址。任何时候，EIP 寄存器的数值加上取得的指令长度减 1，都必须小于等于 0x001FFFFF，否则将引发处理器异常中断。

任何指令都不允许，也不可能向代码段写入数据。而且，只有在代码段可读的情况下（由其描述符指定），才能由指令读取其内容。

12.4.2 堆栈操作时的保护

在保护模式下操作时，堆栈是一个容易令人感到迷惑的话题。在截止到目前的所有例子中，堆栈段一直是使用向下扩展的内存段，段界限的检查和向上扩展的数据段和代码段不同。当然，堆栈也可以使用向上扩展的段，即，把数据段用做堆栈段。在这种情况下，对段界限的检查按数据段的规则进行，但是无论如何，堆栈本身始终总是向下增长的，即，向低地址方向推进。段的扩展方向用于处理器的界限检查，而对堆栈的性质以及在堆栈上进行的操作没有关系。在第 16、17 章中，我们会接触到用向上扩展的段做为堆栈段的情况，现在仍然只讨论向下扩展的堆栈段。

对堆栈操作的指令一般是 push、pop、ret、iret 等。这些指令在代码段中执行，但实际操作的却是堆栈段。

现在只讨论 32 位的堆栈段，即，其描述符 B 位是 1 的堆栈段。处理器在这样的段上执行压栈和出栈操作时，默认使用 ESP 寄存器。

和前面刚刚讨论过的代码段一样，在堆栈段中，实际使用的段界限也和粒度 (G) 位相关，如果 G=0，实际使用的段界限就是描述符中记载的段界限；如果 G=1，则实际使用的段界限为

$$\text{描述符中的段界限值} \times 0x1000 + 0xFFFF$$

堆栈段是向下扩展的，每当往堆栈中压入数据时，ESP 的内容要减去操作数的长度。所以，和向高地址方向扩展的段相比，非常重要的一点就是，实际使用的段界限就是段内不允许访问的最低端偏

移地址。至于最高端的地址，则没有限制，最大可以是0xFFFFFFFF。也就是说，在进行堆栈操作时，必须符合以下规则：

实际使用的段界限+1≤(ESP的内容-操作数的长度)≤0xFFFFFFFF

在上一章里，堆栈段的粒度是字节(G=0)，描述符中的段界限是0x07A00。此时，实际使用的段界限也是0x07A00。

假设现在ESP的内容是0x00007A04，那么，执行下面的指令时，会怎样呢？

```
push edx
```

因为是要压入一个双字(4字节)，故处理器在向堆栈中写入数据之前，先将ESP的内容减去4，得到0x7A00，这就是ESP寄存器在进行压栈操作时的新值。因为该值小于实际使用的段界限0x7A00加一(0x7A01)，因此不允许执行该操作。

但是，如果执行的是这条指令：

```
push ax
```

那么，因为要压入一个字(2字节)，故实际执行压栈操作时，ESP的内容是

0x7C04-2=0x7C02

结果大于实际使用的段界限加一，允许操作。

回到本章中，看代码清单12-1第67~69行。这三行设置堆栈的线性基址为0x00007C00，段界限为0xFFFFE，粒度为4KB，并设置堆栈指针寄存器ESP的初值为0。

因为段界限的粒度是4KB(G=1)，故实际使用的段界限为

0xFFFFE×0x1000+0xFFF=0xFFFFEFFF

又因为ESP的最大值是0xFFFFFFFF，因此，如图12-4所示，在操作该段时，处理器的检查规则是：

0xFFFFF000≤(ESP的内容-操作数的长度)≤0xFFFFFFFF

堆栈指针寄存器ESP的内容仅仅在访问堆栈时提供偏移地址，操作数在压入堆栈时的物理地址要用段寄存器的描述符高速缓存器中的段基址和ESP的内容相加得到。因此，该堆栈最低端的有效物理地址是

0x00007C00+0xFFFFF000=0x00006C00

最高端的有效物理地址是

0x00007C00+0xFFFFFFF=0x00007BFF

也就是说，当前程序所定义的堆栈空间介于地址为0x00006C00~0x00007BFF之间，大小是4KB。

现在结合该堆栈段，用一个实例来说明处理器的检查过程。代码清单第69行将ESP的初始值设定为0，因此，当第一次进行压栈操作时，假如压入的是一个双字(4字节)：

```
push ecx
```

因为压栈操作是先减ESP，然后再访问堆栈，故ESP的新值是(可以自行用Windows计算器算一下)

0-4=0xFFFFFFF

这个结果符合上面的限制条件，允许操作。此时，被压入的那个双字，其线性地址为

0x00007C00+0xFFFFFFF=0x00007BFC

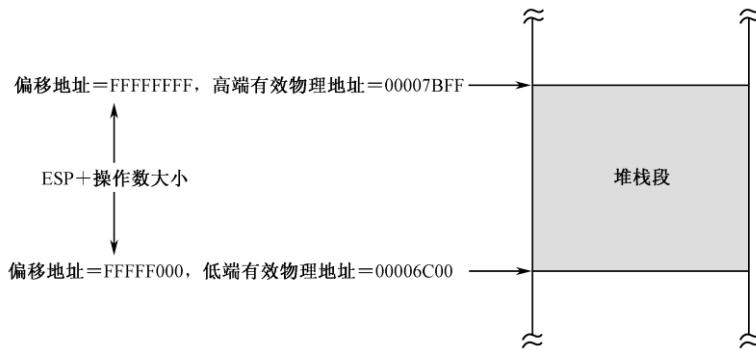


图 12-4 对堆栈段偏移地址的检查

尽管这里讨论的是 push 指令，但对于其他隐式操作堆栈的指令，比如 pop、call、ret 等，情况也没有什么不同，也要根据操作数的大小来检查是否违反了段界限的约束，以防止出现访问越界的情况。

12.4.3 数据访问时的保护

这里所说的数据段，特指向上扩展的数据段，有别于堆栈和向下扩展的数据段。

因为是向上扩展的，所以代码段的检查规则同样适用于数据段。不同之处仅仅在于，对于取指令来说，是否越界取决于指令的长度；而对于数据段来说，则取决于操作数的尺寸。考虑以下指令：

```
mov [0x2000], edx
```

这条指令将访问内存，并将 EDX 寄存器的内容写入当前段内偏移量为 0x2000 的双字单元。指令中给出了内存单元的有效地址 EA (0x2000)，也给出了操作数的大小 (4)。

很好，现在，当处理器访问数据段时，要依据以下规则进行检查：

$0 \leq (EA + \text{操作数大小} - 1) \leq \text{实际使用的段界限}$

在任何时候，段界限之外的访问企图都会被阻止，并引发处理器异常中断。

在 32 位处理器上，尽管段界限的检查总在进行着，但如果段界限具有最大值，则对任何内存地址的访问都将不会违例。比如本章就定义了一个具有 4GB 长的段，段的基地址是 0x00000000，段界限是 0xFFFF，粒度为 4KB。因此，实际使用的段界限是

$0xFFFF \times 0x1000 + 0xFFF = 0xFFFFFFFF$

在这样的段内，访问任何一个内存单元都是允许的，针对段界限的检查都会获得通过。

在 32 位模式下，处理器使用 32 位的段基址加上 32 位的偏移量，共同形成 32 位的物理地址来访问内存。段基址由段描述符指定，而偏移量由指令直接或者间接给出。很显然，在段最大的时候，可以自由访问 4GB 空间内的任何一个单元。

代码清单 12-1 第 71~74 行，从物理地址 0x000B8000 开始写入 16 字节的内容，用于演示 4GB 内存地址空间的访问。段寄存器 ES 当前正指向 0 到 4GB 的内存空间，其描述符高速缓存器中的基地址是 0x00000000，加上指令中提供的 32 位偏移量，所访问的地方正是显示缓冲区（显存）所在的区域。这其中的道理很简单，首先，内存的寻址依赖于段基址和偏移地址，段基址是 0，所以，可以把任何要访问的物理地址作为偏移量。

这 16 字节的内容是 8 个字符的 ASCII 码，以及它们各自的显示属性（颜色）。如图 12-5 所示，和往常一样，双字在内存中的写入依然是低端字节序的，这里再次展示一下，以帮助理解。

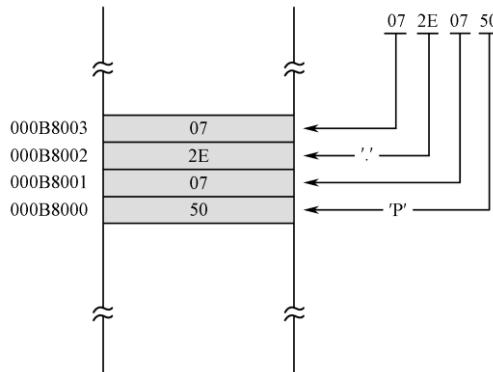


图 12-5 以低端字节序向内存中写入双字

要理解 32 位模式下的寻址，以及数据访问时的保护机制，这是一个很好的例子。

12.5 使用别名访问代码段对字符排序

接下来要做的事情是对一串散乱的字符进行排序。坦白地说，排序是假，主要目的是演示如何在保护模式下使用别名段。

字符串位于代码清单 12-1 的第 105 行，用标号 string 声明，并初始化为以下字符：

```
s0ke4or92xap3fv8giuzjcy5l1m7hd6bnqtw.
```

这串字符是主引导程序的一部分，在进入保护模式时，它就位于 32 位代码段中。代码段是用来执行的，能不能读出，取决于其描述符的类别字段。但是无论如何，它都不允许写入。

这可就难办了。我们想就地把这串字符按 ASCII 码从小到大排列，涉及原地写入数据的操作。好在前面已经建立了代码段的别名描述符，而且用段寄存器 DS 指向它。参见代码清单 12-1 第 59、60 行。

冒泡排序是比较容易理解的排序算法，但却并不是效率最高的，因此，速度自然也就很慢。如果字符串的长度（字符的数量）是 n 个，而且要从小到大排序，那么，可以将它们从头至尾两两比较，需要比较 $n-1$ 次。但是，不要高兴太早，这一次遍历只会使最大的那个字符慢慢地、像气泡一样移动到最右边。

所以，你需要多次进行这样的遍历才能完成所有字符的排序，每一次遍历都会使一个字符冒泡到正确的位置。可以计算，共需要 $n-1$ 次这样的遍历。有关冒泡排序算法的更多信息，请参考其它资料。

可见，这需要两个循环，一个外循环，用于控制遍历次数；一个内循环，用于控制每次遍历时的比较次数。在 32 位模式下，loop 指令所用的计数器不是 CX，而是 ECX。两个循环需要共用 ECX，这需要点技巧，那就是利用堆栈：

```

mov ecx, n-1           ; 控制遍历次数，内、外循环都用它
external:
    xor ebx, ebx        ; 清零，从字符串开头处比较
    push ecx

internal:

```

```

...
;对字符串两两比较

inc ebx
loop internal

pop ecx
loop external

```

我相信这段框架性的代码还是很好理解的。外循环总共执行 n-1 次。每执行一次外循环，内循环就会将一个数排到正确的位置，从而使下一次内循环少一次两两比对（少执行一次）。也就是说，ECX 寄存器的当前值总是内循环的次数，这就是为什么内循环的 loop 指令要使用外循环的 ECX 值。

代码清单 12-1 第 77 行，用后面的标号 pdgt 减去声明字符串的标号 string，就是字符串的长度，再减去一，就是控制循环的次数。

第 79 行，将循环次数压栈，因为内循环会改变 ECX 的内容。

第 80 行，清零 BX 寄存器。该寄存器在每次内部循环之前清零，用于从字符串的开始处进行比对。之所以没有使用 EBX，是因为要让你知道，32 位代码中也可以使用 16 位的寄存器来寻址。注意，我们知道，在 32 位模式下，如果指令的操作数是 16 位的，要加前缀 0x66。相似地，在 32 位模式下，如果要在指令中使用 16 位的有效地址，那么，必须为该指令添加前缀 0x67。因此，当指令

```
mov eax, [bx]
```

用 bits 32 编译后，会有指令前缀 0x67；在 32 位模式下执行时，处理器会用数据段描述符中给出的 32 位数据段基地址，加上 BX 寄存器的 16 位偏移量，形成 32 位线性地址。

实际进行字符比对的代码是第 81~91 行。首先一次性读取两个字符到 AX 寄存器中。当前的数据段是由段寄存器 DS 指向的，其描述符给出的基地址为 0x00007C00，字符串的首地址就是标号 string 的汇编地址，寄存器 BX 用来指定字符串内的偏移量。

接着，对寄存器 AH 和 AL 的内容进行比较。如图 12-6 所示，AL 中存放的是前一个字符，AH 中存放的是后一个字符。如果前一个字符较大，则交换 AH 和 AL 的内容，然后重新写回原来的字单元。然后，将 BX 寄存器的内容加一，以指向下一个字符。

xchg 是交换指令，用于交换两个操作数的内容，源操作数和目的操作数都可以是 8/16/32 位的寄存器，或者指向 8/16/32 位实际操作数的内存单元地址，但不允许两者同时为内存地址。其格式为

```

xchg r/m8,r8
xchg r/m16,r16
xchg r/m32,r32
xchg r8,m8
xchg r16,m16
xchg r32,m32

```

举个例子：

```

mov ecx,0xf000f000
mov edx,0xabcdef00
xchg ecx,edx

```

以上指令执行后，寄存器 ECX 中的内容为 0xABCDDEF00，EDX 中的内容为 0xF000F000。

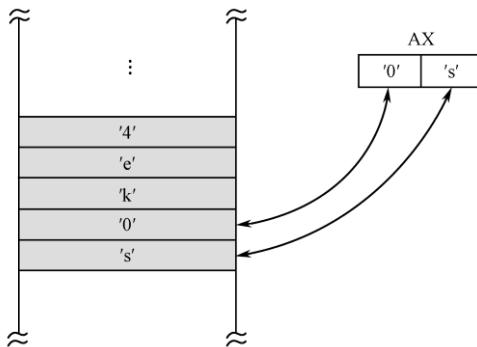


图 12-6 通过 AX 寄存器比对和排序相邻字符

第 93~100 行用于显示最终的排序结果，同样使用了循环，循环次数就是字符串的长度。和排序的时候不同，现在终于使用 EBX 了，这将提供 32 位的偏移地址。

第 96 行，向寄存器 AH 传送的是字符的显示属性（颜色），0x07 表示黑底白字，我们已经无数次重复说过了。

第 98 行是向显存中传送字符及其显示属性：

```
mov [es:0xb80a0+ebx*2],ax
```

段寄存器 ES 是在刚进入保护模式时设置的，它指向 0~4GB 内存的段。0xb80a0 等于 0xb8000 加上十进制数 160 (0xa0)。在显存中，偏移量为 160 的地方对应着屏幕第 2 行第 1 列。32 位处理器提供了强大的寻址方式，可以在基址寄存器的基础上使用比例因子，这里是将 EBX 寄存器的内容乘以 2。当 EBX 的内容为 0、1、2、3、…时，计算出来的有效地址分别是 0xb80a0、0xb80a2、0xb80a4、0xb80a6、…，后面的依次类推，很容易看到使用比例因子的好处。注意，该表达式的值是在本指令执行时，由处理器来计算的。

最后，在完成了所有的工作之后，第 102 行，hlt 指令使处理器处于停机状态。

12.6 程序的编译和运行

本章代码清单 12-1 所对应的源程序文件是 c12_mbr.asm，用 Nasmide 工具将它打开并编译，生成二进制文件 c12_mbr.bin 并写入虚拟硬盘的主引导扇区。

然后，启动虚拟机 LEARN-ASM，观察运行结果。正常情况下，屏幕显示如图 12-7 所示。

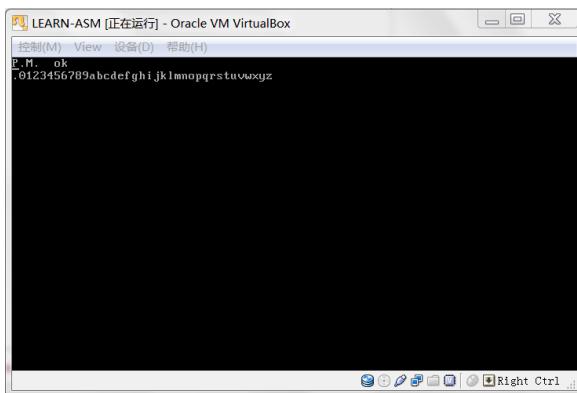


图 12-7 本章程序运行结果

本 章 习 题

1. 修改本章代码清单，使之可以检测 1MB 以上的内存空间（从地址 0x00100000 开始，不考虑高速缓存的影响）。要求：对内存的读写按双字的长度进行，并在检测的同时显示已检测的内存数量。建议对每个双字单元用两个花码 0x55AA55AA 和 0xAA55AA55 进行检测。

2. 有一个向下扩展的段，描述符中的 B 位和 G 位都是“1”。请问，如果希望段的大小为 8KB，那么，描述符中的界限值应当是多少？

第 13 章 程序的动态加载和执行

像我一样，很多人在了解了保护模式的基本工作原理之后，会产生一个疑问。那就是，所有的段在使用之前，都必须以描述符的形式在描述符表中进行定义，那么，像操作系统这样的软件，又怎么能够加载和执行其他各种用户程序呢？

未必所有人都会产生这样的疑惑，但我确实算一个，可能我还够聪明。事实上，这仅仅是一层窗户纸，一旦捅破了，才发现原来竟是那么简单。从某种意义上来说，保护模式的工作机制对用户的加载和执行非但没有增加困难，反而带来了很大的便利。

一套能够充分说明问题的例子需要很大的代码量，也许把本书的汉字都去掉，全部换成代码也不够。不过，只要能说明问题，也不一定非得完善周全、面面俱到。因此，本章中用于加载和处理用户程序的做法，不一定，甚至根本就不是操作系统采用的方法。这一点，务必明了。

计算机硬件之上是软件。软件分两个层次，一是操作系统，二是应用（用户）程序。通常，用户程序只关心问题的解，就是采用各种算法来解决实际问题。至于软件是怎么加载到内存的，怎么定位的，不是它所操心的事。但是，它有义务提供一些必要的信息，来帮助操作系统将自己加载到内存中。

相反，操作系统则必须考虑采用什么方法来加载用户程序，并在适当的时候将处理器的执行流转移到用户代码中去。同时，为了减轻用户程序的工作量，操作系统还应当管理硬件，并提供大量的例程供用户程序使用。比如，显示一个字符串，就不要让用户自己来写代码了，直接调用操作系统的代码即可。但操作系统和用户程序应当协商一种机制，让用户程序能够在使用这些例程时，不必考虑和关心它们的位置。

本章提供了一个小小的“操作系统”，因为当不起这么大的名称，所以叫“内核”或者“核心”。即使是这样，它依然当不起，因为它实在是太简单了。不过，也没有办法，就这么凑合着叫吧。

内核不能放到主引导扇区里，毕竟它都很大。所以，计算机首先从主引导程序开始执行，主引导程序负责加载内核，并转交控制权。然后，内核负责加载用户程序，并提供各种例程给用户程序调用。提供给用户程序调用的例程也叫应用程序接口（Application Programming Interface，API），本章用简单的方法来允许用户程序使用 API 工作。

13.1 本章代码清单

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：13-1（主引导扇区程序），源程序文件：`c13_mbr.asm`

本章代码清单：13-2（微型内核），源程序文件：`c13_core.asm`

本章代码清单：13-3（被加载的用户程序），源程序文件：`c13.asm`



13.2 内核的结构、功能和加载

13.2.1 内核的结构

内核分为四个部分，分别是初始化代码、内核代码段、内核数据段和内核例程段，主引导程序也是初始化代码的组成部分。

初始化代码用于从 BIOS 那里接管处理器和计算机硬件的控制权，安装最基本的段描述符，初始化最初的执行环境。然后，从硬盘上读取和加载内核的剩余部分，创建组成内核的各个内存段。初始化代码大部分位于代码清单 13-1 中。

内核的代码和数据位于代码清单 13-2 中。如图 13-1 所示，内核代码段是在第 385 行定义的，用于分配内存，读取和加载用户程序，控制用户程序的执行。

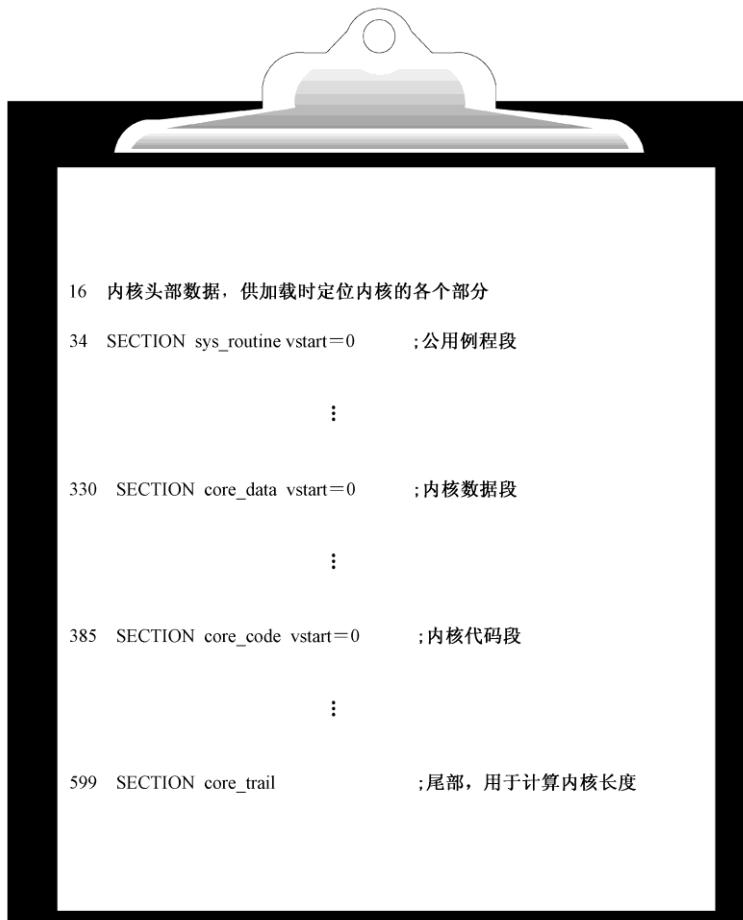


图 13-1 内核程序的各个组成部分

内核数据段是在第 330 行定义的，提供了一段可读写的内存空间，供内核自己使用。

内核例程段是在第 34 行定义的，用于提供各种用途和功能的子过程以简化代码的编写。这些例程既可以用于内核，也供用户程序调用。

除了以上的内容之外，内核文件还包括一个头部，记录了各个段的汇编位置，这些统计数据用于告诉初始化代码如何加载内核。

回到代码清单 13-2 的开头。

从第 7 行开始，一直到第 12 行，用于声明常数。很明显，这是一些内存段的选择子，它们对应的描述符会在内核初始化的时候创建。组成内核的各个段和内核本身一样稳定，在系统运行期间不会变来变去。因此，将它们声明为常数，可以为引用它们带来方便。我们知道，伪指令 `equ` 仅仅是允许我们用符号代替具体的数值，但声明的数值并不占用空间。

内核文件的真正开始部分是头部，偏移量为 0x00 的地方是一个双字，记录了整个内核文件的大小，以字节为单位；偏移量为 0x04 的地方是公用例程段的起始汇编地址，是一个双字；偏移量为 0x08 的地方是核心数据段的起始汇编地址，也是一个双字；偏移量为 0x0C 的地方是核心代码段的起始汇编地址，双字大小；从偏移量为 0x10 开始的地方用于指示内核入口点，在主引导程序加载了内核之后，可以从这里把处理器的控制权交给内核代码。注意，不要忘了这个表达式，我们以前学过的，它用来得到段的起始汇编地址：

```
section.<段名称>.start
```

入口点共有 6 字节，低地址部分是一个双字，指示段内偏移，将来会传送到指令指针寄存器 EIP，它来自一个标号 `start`，位于第 531 行；高地址部分是一个字，指定一个内存代码段的选择子。在这里，填充的是刚刚在第 7 行声明过的常数 `core_code_seg_sel`，在数值上等于 0x38。

13.2.2 内核的加载

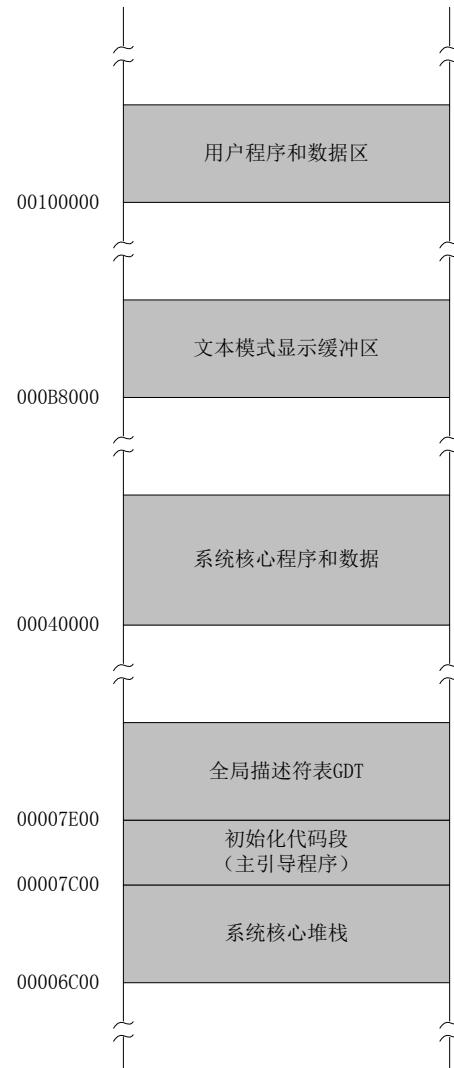
现在来看代码清单 13-1，也就是主引导程序。

第 6 行和第 7 行声明了两个常数，分别是内核程序在硬盘上的位置，以及它将要被加载的物理内存地址。声明常数的好处你也知道，将来改起来方便。

接下来，从第 9 行开始，一直到第 55 行，是为进入保护模式做准备。如图 13-2 所示，因为主引导程序的加载位置是物理地址 0x00007C00，所以，从这个位置往上是 512 字节的初始化代码段，从这个位置往下是 4KB 的内核堆栈。

全局描述符表（GDT）是不可或缺的，和从前一样，我们将它定义在从物理地址 0x00007E00 开始的地方，紧挨着初始化代码段。GDT 可大可小，最大能达到 64KB，所以，它的空间一定要留够。

和 GDT 一样，内核程序的大小也是不定的，但可以规定它的起始位置。在这里，我们决定将它加载到从物理内存地址 0x00040000 开始的地方。从这个地方往上，一直到 0x0009FFFF，都是它的地盘，取决于它到底有多大，想用多少就用多少。从 0x000A0000 往上，是 ROM BIOS，硬件专有的。





显示器是窥视程序工作的窗口，显示功能自然少不了。因此，从 0x000B8000 往上的 32KB，是文本模式的显示缓冲区。

最后，从 1MB 开始的大量空间是留给用户程序用的，具体数量取决于你到底安装了多少物理内存。对于本章来说，程序都很小，功能都很简单，用不了多少内存空间，都才几 KB、几十 KB；但是，你平时所用的 Windows、Linux 和 MacOS，以及运行于其上的程序，都是 VIP、大客户，动辄几 MB、几百 MB。

在进入保护模式之前，初始化程序已经在全局描述符表（GDT）中安装了几个必要的描述符。如图 13-3 所示，第一个是用于访问 0~4GB 内存的数据段，它很重要，内核只有在具备了访问全部 4GB 内存空间的能力时，才能随心所欲地做任何事情。

第二个是初始化代码段，也就是主引导程序所在的段。进入保护模式后，要继续执行主引导程序的后半部分代码，必须按处理器的要求，为它创建描述符。

最后两个分别是初始的堆栈段和显示缓冲区的描述符。这里定义的堆栈在初始化过程中就要使用，而在进入内核之后，它又是内核的堆栈。

创建这些描述符的代码位于代码清单 13-1 的第 19~40 行，这几个描述符都和上一章差不多，而且用于创建它们的代码也基本相同，不再逐个讲解。

+20	文本模式显存 (000B8000~000BFFFF)	0x20
+18	初始堆栈段 (00006C00~00007C00)	0x18
+10	初始代码段 (00007C00~00007DFF)	0x10
+08	0~4GB数据段 (00000000~FFFFFFF)	0x08
+00	空描述符	0x00

图 13-3 进入保护模式前创建的描述符

下面开始加载内核。

首先是初始化各个段寄存器以访问相应的内存段。第 59、60 行，使 DS 指向全部 4GB 的内存空间；第 62~64 行，使 SS 指向初始的堆栈空间，并初始化堆栈指针寄存器 ESP 的内容为 0。第一个数据压入时，因为堆栈的操作是先减 ESP 的值，再保存数据，所以，如果是压入一个字，ESP 的内容为 0xFFFFFFF；如果压入的是双字，ESP 的内容为 0xFFFFFFF。

接下来是从硬盘把内核程序读入内存，第 67~69 行，它在硬盘上的起始逻辑扇区号和物理内存地址已经由两个常数给出，现分别将它们传送到 EAX 和 EDI 寄存器。

初始化代码并不知道内核有多大，所以也就不知道应该读多少个扇区。不过，它可以先读一个扇区，因为那里包含着内核的头部数据，根据这些数据，就可以知道内核的总扇区数。

和以前一样，我们把读硬盘扇区的指令归拢到一起，做成可以反复调用的过程 `read_hard_disk_0`，它位于第 138~192 行。基本上，它的工作过程和具体的代码都和从前一样，但略有不同。首先，该过程要求使用 EAX 寄存器来传入 28 位的逻辑扇区号。我们现在已经可以使用 32 位的寄存器了，再也不会因为 16 位寄存器太小，无法容纳 28 位的逻辑扇区号而发愁。

其次，这里使用 EBX 寄存器来传入偏移地址。因为在 32 位模式下，可以访问全部 4GB 内存，允许使用 32 位的偏移地址。这是好事，我们再也不需要为 64KB 的段而受折磨了。

最后一个不同之处在于，过程返回时，会使 EBX 寄存器的值比原来多 512。这是有意的，

因为在 32 位模式下，内存的访问不再受 64KB 限制，所以就能够连续访问。这里，每次将 EBX 寄存器的内容加上 512，目的是指向下一个内存块，我相信这种工作方式会给调用它的主程序带来方便。

接下来是取得内核的长度，并计算它所占用的扇区数。

因为段寄存器 DS 是指向 4GB 内存段的，其描述符高速缓存中的基地址是 0x00000000，故，第 75 行，可以直接用 EDI 寄存器中的数值作为偏移量来访问内存，最终生成的线性地址在数值上和 EDI 寄存器的内容相同。当前指令的功能是取得内核的总长度，因为它就位于内核的偏移 0 处。

第 75~77 行，将取得的总字节数除以 512，就能在 EAX 寄存器中得到内核所占用的扇区数。不过，在没能整除的情况下，实际的扇区总数要比 EAX 寄存器中的值多一。

但是，我们要的是剩余扇区数，毕竟已经读了一个。为此，第 79~81 行，先判断 EDX 寄存器中的余数是否为零。取决于 EDX 的实际内容，or 指令会影响 ZF 标志位。如果 EDX 不为零，则 EAX 寄存器里实际上就是剩余的扇区数，因为它比实际的扇区数少一。相反，如果 EDX 的内容为零，则 EAX 中的内容就是总扇区数，还要用 dec 指令减一才行。

无论是哪种情况，指令的执行流程都会到达第 83 行。这个地方指令是

```
or eax, eax
```

这条指令的工作是检查 EAX 寄存器，看它的内容是否为零。第 84 行，如果为零，说明内核就占用了一个扇区（确实够小的，但一般不太可能），于是不再读硬盘，直接转到标号 setup 处执行。

第 87~93 行，用于从硬盘读取剩余的扇区，用的是 loop 指令循环读取，循环的次数在 ECX 寄存器中。再重复一遍，32 位模式下的循环指令需要使用 ECX 寄存器，而不是 CX。如果没有第 83、84 行的条件判断，而且剩余扇区数为 0，那么，这里的循环将执行 0xFFFFFFF+1 次，显然不是我们希望的。

13.2.3 安装内核的段描述符

要使内核工作起来，首要的任务是为它的各个段创建描述符。换句话说，还要为 GDT 续添新的描述符。进入保护模式前，我们在代码清单 13-1 的第 42 行使用指令

```
lgdt [cs: pgdt+0x7c00]
```

来加载全局描述符表寄存器（GDTR），标号 pgdt 所指向的内存位置包含了 GDT 的基地址和大小。现在，我们的任务是重新从标号 pgdt 处取得 GDT 的基地址，为其添加描述符，并修改它的大小，然后用 lgdt 指令重新加载一遍 GDTR 寄存器，使修改生效。

但是，如果忽略了一件事，你可能不会得逞。标号 pgdt 所指向的内存区域位于主引导程序内，而我们当前正在保护模式下执行主引导程序。保护模式下的代码段只是用来执行的，是否能读出，取决于其描述符的类别字段，但无论如何它都不能写入。

对代码段实施保护的意思是通过代码段描述符不能修改段中的内容，但这意味着通过其他描述符做不到。想想看，我们拥有一个指向全部 4GB 内存空间的描述符，标号 pgdt 所指向的内存位置不单单是在主引导程序内，同时也是 4GB 内存空间的一部分。

如图 13-4 所示，标号 pgdt 在数值上等于它距离段首的偏移量，也就是编译阶段的汇编地址。主引导程序的物理起始地址是 0x00007C00，故 pgdt 在 4GB 段内的偏移量是 0x00007C00+pgdt。

这样，为了得到 GDT 的基地址，代码清单 13-1 第 96 行，使用了指令

```
mov esi, [0x7c00+pgdt+0x02]
```

注意，指令中的表达式是在编译阶段计算的。默认的段寄存器是 DS，当这条指令执行时，



处理器用 DS 描述符高速缓存器中的 32 位线性基地址 0x00000000 加上用该表达式计算出的偏移量来访问内存。

现在可以创建与内核相关的其他段描述符。首先是公共例程段。如图 13-5 所示，内核头部偏移 0x04 处的一个双字，就是公共例程段的起始汇编地址。由于内核被加载的物理地址是由 EDI 寄存器指向的，所以，第 99 行，直接访问 4GB 内存段，从该偏移位置取出公共例程段的起始汇编地址。

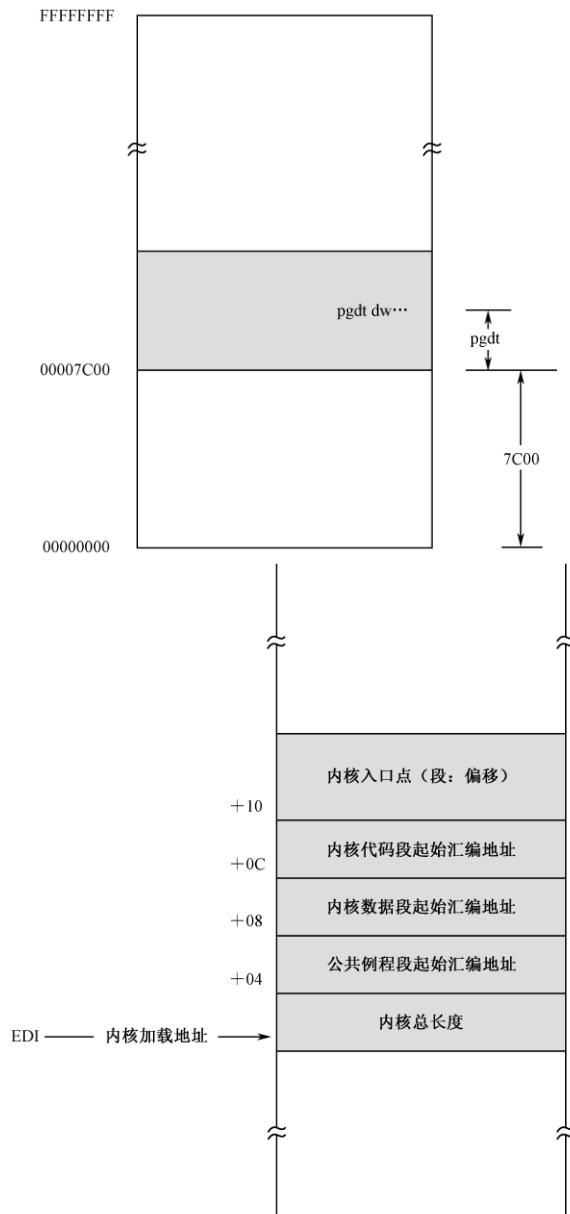


图 13-4 通过 4GB 数据段访问代码段内的数据

图 13-5 内核头部的组成

创建描述符还需要知道段界限。在内核中，各个段有着确定的先后次序，而且是紧挨着的。公共例程段的后面是内核数据段，用内核数据段的起始汇编地址，减去公共例程段的起始汇编地址，再减去一，就是公共例程段的段界限，这就是第 100~102 行所做的工作。对于向上扩展

的段来说，段界限在数值上等于段的长度减去一，这个必须要清楚。

第 103 行，用公共例程段的起始汇编地址，加上内核的加载地址，就是公共例程段的基址。

在已经知道某个内存段的细节时，写出它的描述符是很容易的。比如，如果已经知道堆栈的基地址是 0x00007C00，粒度是 4KB，大小是 8KB，那么，它的描述符就可以直接给出：

```
0x00CF96007C00FFFD
```

问题是，这种清楚明白的情形不常见。在百分之九十以上的场合，段的信息也是确定的，但它只有在程序运行的时候才能确定，它们都是在程序运行时，根据实际情况得到的随机值。为此，就需要利用指令来以不变应万变，“拼凑”出描述符来。

既然是灵活的方法，还能以不变应万变，就应该定义成过程，以方便在需要的时候随时调用。在这里，我们的方法是使用过程 `make_gdt_descriptor`。

过程 `make_gdt_descriptor` 位于代码清单 13-1 的 195~217 行，调用该过程需要三个参数，分别是段的线性基地址、段界限和段的属性值。段的基址用 `EAX` 寄存器传入；段界限用 `EBX` 寄存器传入，但只用其低 20 位；段属性用 `ECX` 寄存器传入，各属性位在 `ECX` 寄存器中的分布和它们在描述符高 32 位中的时候一样，其他和段属性无关的位都清零。

因此，第 104 行，将段属性值 0x00409800 传送到 `ECX` 寄存器。结合第 11 章的图 11-4，可以知道，这是一个 $P=1$ 、 $D=1$ 、 $G=0$ 、 $DPL=0$ 、 $S=1$ ， $TYPE=1000$ 的（代码）段描述符。第 105 行，调用过程创建描述符，下面来看看具体的创建过程。

代码清单 13-1 的第 201~203 行用于构造描述符的低 32 位。首先是将 32 位段基地址从 `EAX` 寄存器复制一份给 `EDX` 寄存器，过一会儿构造描述符的高 32 位时，还要用到基地址。

描述符的低 32 位中，高 16 位是基地址；低 16 位是段界限，所以，第 202~203 行，将 `EAX` 寄存器中的 32 位基地址左移 16 次，使基地址部分就位。然后，把 `BX` 寄存器中的段界限用 `or` 指令安排就位。这样，描述符的低 32 位就构造完毕了。

相比之下，描述符的高 32 位构造起来比较麻烦。如图 13-6 所示，描述符高 32 位的标准形态是有两个基地字段和一个段界限字段。基地址在 `EDX` 寄存器中有备份，执行第 205~207 行的指令后，会使基地址部分在两边就位。

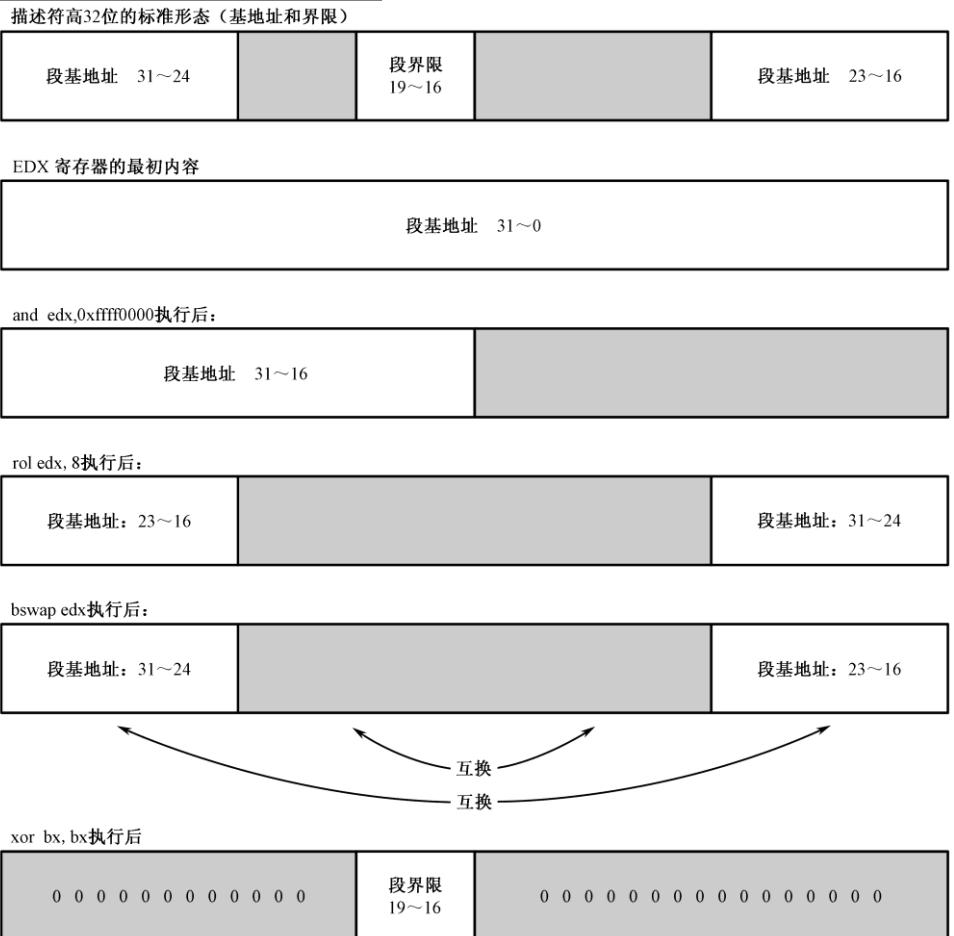


图 13-6 描述符高 32 位的构造过程

`bswap` 是字节交换指令(Byte Swap), 在标准的 32 位处理器上只允许 32 位的寄存器操作数, 其格式为

bswap r32

处理器执行该指令时，按如下过程操作（DEST 是指令中的操作数，TEMP 是处理器内的临时寄存器）：

```

TEMP ← DEST
DEST[7:0] ← TEMP[31:24];
DEST[15:8] ← TEMP[23:16];
DEST[23:16] ← TEMP[15:8];
DEST[31:24] ← TEMP[7:0];

```

接下来，要在描述符的高 32 位中装配段界限字段。第 209、210 行，先清除 EBX 寄存器的低 16 位，然后同 EDX 寄存器合并。

最后, 第 212 行, 将 ECX 寄存器中的段属性与 EDX 寄存器中的描述符高 32 位合并。至此, 我们就在 EDX:EAX 中得到了完整的 64 位描述符。第 214 行, ret 指令将控制返回到调用者。

现在，回到主程序，来看第 106、107 行，ESI 寄存器的内容是 GDT 的基地址，这两条指令访问 4GB 的段，定位到 GDT，在原先的基础上，再添加一个描述符，就是我们刚刚创建的描述

符。

第 110~129 行，用于安装内核数据段和内核代码段的描述符，也采用了相同的过程，不再一一讲解。

第 131 行，通过 4GB 的数据段访问 pgdt，修改它的界限值。现在，GDT 中已经有 8 个描述符，故其总长度为 64 字节。相应地，界限值为 63。

第 133 行，通过 4GB 的数据段访问 pgdt，重新加载 GDTR，使上面那些对 GDT 的修改生效。

至此，内核已经全部加载完毕，图 13-7 是内核加载完成之后的 GDT 布局。

第 136 行，通过 4GB 的数据段访问内核的头部，用间接远转移指令从给定的入口进入内核执行。观察图 13-5，再参考代码清单 13-2 就可以明白，在内核头部偏移 0x10 处，是 6 字节的内核入口点。前面是 32 位的段内偏移地址，后面是 16 位的段选择子，指向内核代码段。在这里，段选择子直接使用固定的数值不是一个好主意，怕的是往后内核有重大调整时，会改变描述符的次序。在这种情况下，如果别处改了，这里忘了修改，就一定会出现问题。

+38	核心代码段（位于核心数据段之后）	0x38
+30	核心数据段（位于系统公用例程段之后）	0x30
+28	系统公用例程段（起始地址为 00040000）	0x28
+20	文本模式显存（000B8000~000BFFFF）	0x20
+18	初始堆栈段（00006C00~00007C00）	0x18
+10	初始代码段（00007C00~00007DFF）	0x10
+08	0~4GB 数据段（00000000~FFFFFFFF）	0x08
+00	空描述符	0x00

图 13-7 内核加载完成后的 GDT 布局

13.3 在内核中执行

现在转到代码清单 13-2，这是内核的主体部分。

从主引导程序转移到内核之后，处理器会从第 532 行开始执行，因为这里是内核的入口。

第 532、533 行，初始化段寄存器 DS，使它指向内核数据段。然后，第 535、536 行，调用公共例程段内的一个过程来显示字符串。该 call 指令属于直接远转移，指令中给出了公共例程段的选择子和段内偏移量。字符串是在第 362 行，用标号 message_1 声明，并初始化了一段文字，意思是“如果你看到这段信息，那么这意味着我们正在保护模式下运行，内核已经加载，而且显示例程工作得也很完美。”

显示例程 put_string 位于公共例程段内，是在第 37 行定义的。基本上，它的代码组成和工作原理都和从前一样，但也有不同之处。首先，这里的代码是 32 位模式的，字符串的地址由 DS:EBX



传入，过程返回时用 `retf` 指令，而不是 `ret`。这意味着，必须以远过程调用的方式使用它。

和往常一样，`put_string` 在内部调用了另一个过程 `put_char`。注意，第 110~113 行，`movsd` 用于在两个内存区域间传送双字数据（一次传送 4 字节）。不管是 `movsb`，还是 `movsw`，抑或是 `movsd`，在 16 位模式下，是把由 DS:SI 指定的源操作数传送到由 ES:DI 指定的目的地。但是，在 32 位模式下，源和目的则分别是 DS:ESI 和 ES:EDI。

再回到 539 行，下面的工作是显示处理器品牌信息。

处理器的功能是强劲的，这个没有人怀疑。同时，在处理器内部也隐藏着太多的秘密，除了处理器的型号，还有大量的特性信息，比如高速缓存的数量、是否具备温度和电源管理功能、逻辑处理器的数量、高级可编程中断控制器的类型、线性（物理）地址的宽度、是否具有多媒体扩展和单指令多数据指令等特性。

处理器功能强了是好事，大家都很欢喜。麻烦在于，很多新功能是处理器更新换代的产物，只存在于最新的版本中，旧的处理器没有。比如多媒体扩展指令可以加速多媒体的处理速度，但用了新指令的软件不能运行在旧的处理器上，因为它们不支持。可怕之处在于，没有人知道自己的软件被终端销售商卖给了谁，更不知道那个谁用的是什么处理器。

因此，你的软件应当准备两套方案，而且，在决定使用哪套方案之前，必须探测和挖掘处理器内部的秘密，好知道该怎么办。Intel 公司显然洞悉了市场上发生的一切，它们给出的方案是使用 `cpuid` 指令。

`cpuid` 指令（CPU Identification）用于返回处理器的标识和特性信息。EAX 用于指定要返回什么样的信息，也就是功能。有时候，还要用到 ECX 寄存器。`cpuid` 指令执行后，处理器将返回的信息放在 EAX、EBX、ECX 或者 EDX 中。

`cpuid` 指令是从 80486 处理器的后期版本开始引入的，从此以后，每款处理器都会对可以返回的信息有所扩充。原则上，在使用 `cpuid` 指令前，先要检测处理器是否支持该指令；接着再用 `cpuid` 指令检测是否支持所需要的功能。

如图 13-8 所示，在 32 位处理器上，原先的标志寄存器 FLAGS 也相应地扩充到了 32 位，以支持更多的标志。扩充之后的标志寄存器称为 EFLAGS 寄存器，它的 ID 标志位（位 21）如果为“0”，则不支持 `cpuid` 指令；反之，该处理器支持 `cpuid` 指令。80486 处理器已经很久远了，我想没有谁还在使用这样的计算机，况且它已经停产。一般情况下，不需要检测处理器是否支持 `cpuid` 指令。

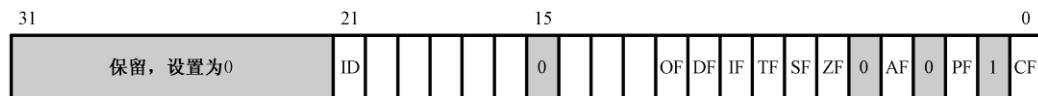


图 13-8 扩展到 32 位长度的标志寄存器 EFLAGS

图 13-8 中，灰色的部分是保留位，通常设置为固定的值。EFLAGS 还包括更多的标志位，图中未予显示，仅在以后用到的时候一一介绍。

为了探测处理器最大能够支持的功能号，应该先用 0 号功能来执行 `cpuid` 指令：

```
mov eax,0
cpuid
```

处理器执行后，将在 EAX 寄存器返回最大可以支持的功能号。同时，还在 EBX、ECX 和 EDX 中返回处理器供应商的信息。对于 Intel 处理器来说，返回的信息如下：

`EBX ← 0x756E6547` （对应字符串“Genu”，“G”在 BL 中，其他类推）

`EDX ← 0x49656E69` （对应字符串“ineI”，“i”在 DL 中，其他类推）

ECX \leftarrow 0x6C65746E (对应字符串“ntel”，“n”在CL中，其他类推)

组合起来就是“GenuineIntel”。

要返回处理器的品牌信息，需要使用 0x80000002~0x80000004 号功能，分三次进行。注意，该功能仅被奔腾 4 (Pentium 4) 之后的处理器支持，所以，正确的做法是先用 0 号功能执行 cpuid 指令，以判断自己的处理器是否支持。代码清单 13-2 并没有这样做，因此可视为一个反面典型。

第 539~558 行，分别用三种功能号执行 cpuid 指令，返回三组字符串，共 48 个字符，依次保存在核心数据段中，起始位置是由标号 cpu_brand 指定的。第 381 行，声明了标号 cpu_brand，并初始化了 52 字节，足以容纳这些数据。

从处理器返回的数据都是现成的 ASCII 码。第 560~565 行，先在屏幕上留出空行，再显示处理器品牌信息，然后再留空，以突出要显示的内容。

13.4 用户程序的加载和重定位

好了，现在我们可以开始加载用户程序了。

用户程序加载之前，要先显示一段信息，意思是要加载用户程序了。这是第 567、568 行的工作。字符串位于内核数据段中，第 367 行声明了标号 message_5 并初始化了字符串。

第 569 行用于指定用户程序的起始逻辑扇区号。在指令中直接指定数值不是一个好习惯，正确的做法是用伪指令 equ 声明成常数，并放到整个程序的起始部分以便修改。

内核的主要任务就是加载和执行用户程序。通常情况下，这样的工作会反复进行。为了方便，一般要定义成可反复调用的过程。在这里，我们也是这样做的，过程的名字叫 load_relocate_program。该过程位于第 387 行，作用是加载和重定位用户程序。从代码清单中可以看出，它是内核代码段的一个内部过程。

13.4.1 用户程序的结构

用户程序必须符合规定的格式，才能被内核识别和加载。通常情况下，流行的操作系统会规定自己的可执行文件格式，一般都比较复杂，这种复杂性和操作系统自身的复杂性是息息相关的。

现在转到代码清单 13-3，来看看用户程序的结构。

所有操作系统的可执行文件都包括文件头，这里也不例外。事实上，这也是我们熟悉的、一贯的做法。在文件头内的偏移 0 处，是一个双字，指示了用户程序的大小，以字节为单位。

偏移量为 0x04 处的双字是头部的长度，以字节为单位。

偏移量为 0x08 处的双字是为堆栈保留的，和早先的做法不同，内核不要求用户程序提供堆栈空间，而改由内核动态分配，以减轻用户程序编写的负担。当内核分配了堆栈空间后，会把堆栈段的选择子填写到这里，用户程序开始执行时，可以从这里取得该选择子以初始化自己的堆栈；

偏移量为 0x0c 处的双字是要求分配的堆栈大小，即，用户程序编写者建议的堆栈大小，以 4KB 为单位。如果是 1，就是希望分配 4KB 的堆栈空间；如果是 2，就是希望分配 8KB 的堆栈空间，依此类推。

偏移量为 0x10 处的双字，是用户程序入口点的 32 位偏移地址。



偏移量为 0x14 处的双字，是用户程序代码段的起始汇编地址。当内核完成对用户程序的加载和重定位后，将把该段的选择子回填到这里（仅占用低字部分）。这样一来，它和 0x10 处的双字一起，共同组成一个 6 字节的入口点，内核从这里转移控制到用户程序。

偏移量为 0x18 处的双字，是用户程序代码段的长度，以字节为单位。

偏移量为 0x1c 处的双字，是用户程序数据段的起始汇编地址，当内核完成用户程序的加载和重定位后，将把该段的选择子回填到这里（仅占用低字部分）。

偏移量为 0x20 处的双字，是用户程序数据段的长度，以字节为单位。

除了加载和重定位用户程序外，内核还应当提供一些例程供用户程序调用。操作系统对于普通用户来说，是赏心悦目的界面和快捷直观的操作方式，对程序员来说，则是一个巨大的例程库，节省了时间，减少了工作量，甚至不需要直接访问硬件。

操作系统提供的编程接口就是 API，这是一大堆例程（过程），需要的时候直接调用即可。问题在于，它们在操作系统内部，对任何人来说都是不可见的，更别想知道它们的入口地址。但是，call 指令是需要直接或间接提供一个地址的。另一方面，即使你知道它们的地址，调用的时候也有风险，因为操作系统也需要升级换代，这些地址可能改变。当你的程序在新操作系统上工作时，就要出问题。

为了使开发人员能够利用它所提供的 API，操作系统至少要公开它们。在早期的系统中，这些 API 以中断号的方式公布，因为它们是通过软中断进入的。不过，另一种常见的办法是使用符号名。比如，操作系统提供了一个例程，用于显示光标跟随的字符串，那么，它可以公布一个符号名：

PrintString

当然，它肯定不会同时公布一个段地址和偏移地址，因为它也不能保证地址不会变化。在操作系统的开发手册中，会列出所有符号名。符号名在高级语言里就是库函数名。

回到代码清单 13-3 中来。

内核要求，用户程序必须在头部偏移量为 0x28 的地方构造一个表格，并在表格中列出所有要用到的符号名。每个符号名的长度是 256 字节，不足部分用 0x00 填充，这意味着每个符号名的长度最多可以是 256 个字符。在用户程序加载后，内核会分析这个表格，并将每一个符号名替换成相应的内存地址，这就是过程的重定位。为了方便起见，我们把该表格叫做“符号-地址检索表”（Symbol-Address Lookup Table，SALT）。不要上网搜索这个词，也不要查别的资料，这不是一个标准，是我自己随心所欲、特立独行的产物。

第 29~36 行声明了三个标号，并分别初始化了三个符号名，每一个 256 字节，不足部分是用 0 填充的。每个符号名都以“@”开始，这并没有任何特殊意义，仅仅在概念上用于表示“接口”的意思。为了计算需要填充多少个 0，它们都使用了相似的表达式，比如：

```
times 256-($-PrintString) db 0
```

这里，先计算出符号名的实际字符数，即\$-PrintString，再用 256 减去实际字符数，就得到了伪指令 db 的重复次数。

SALT 表可大可小，内核需要知道它在哪里结束。第 26 行，用于初始化 SALT 表的项数，也就是符号名的数量，它是用表格的总长度除以每个符号名的长度（256）得到的。

事实上，即使是大多数汇编语言，也不需要亲自构造文件头，那是链接器（Linker）的工作。但是，链接器是为流行的操作系统服务的，用于构造他们可以识别的可执行文件格式。我们不想把问题搞得太复杂，就本书的篇幅和宗旨来说，迎合“流行”所要花费的代价实在太大，管中窥豹、点到即止不是很好吗？

13.4.2 计算用户程序占用的扇区数

再次回到代码清单 13-2。

用户程序的加载是在例程 `load_relocate_program` 内进行的，该过程需要用 ESI 寄存器传入用户程序的起始逻辑扇区号。当过程返回时，在 AX 寄存器内包含了指向用户头部段的选择子。

第 396、397 行，因为在过程中要用到 DS 和 ES，故将其原先的内容压栈保存。

为了得到用户程序的大小，需要先预读它的第一个扇区，第 399~404 行就在做这件事。首先，使段寄存器 DS 指向内核数据段；然后，调用读硬盘的过程 `read_hard_disk_0` 来预读用户程序。进入过程前，EAX 寄存器的内容是用户程序的起始逻辑扇区号；数据的存放地点是内核缓冲区 `core_buf`，它位于内核数据段中，是在第 376 行声明和初始化的。在内核中开辟出一段固定的空间，对于分析、加工和中转数据都比较方便。

接下来的工作是计算用户程序到底占用了多少个扇区。用户程序的总大小就在头部内偏移量为 0x00 的地方，因此，第 407 行直接访问内核缓冲区取得这个双字。

用户程序的大小（总字节数）不一定恰好是 512 的整数倍。也就是说，最后一个扇区未必是满的。因此，如果直接除以 512，可能会使结果（除法的商）比实际的扇区数少一。通常情况下，需要判断除法的余数，根据余数是否为零，来决定实际的扇区总数，这不可避免地要使用判断和条件转移指令。

在早先的处理器中，转移指令是影响处理器速度的重大因素之一，因为它会使流水线中那些已经预取和译码的指令失效。在较晚的处理器中，普遍采用了分支预测技术，但并不总能保证预测是准的。因此，最好的办法就是尽量不使用转移指令。为了帮助程序员部分地戒掉使用转移指令的欲望，处理器引入了条件传送指令 `cmovecc`。

`cmovecc` 指令是从 P6 处理器族开始引入的，因此并非所有处理器都支持它。如果你想知道确切的结果，可以先以 1 号功能执行 `cpuid` 指令：

```
mov eax,1  
cpuid
```

当处理器执行这两条指令后，会在 EBX、ECX 和 EDX 寄存器返回丰富的信息，以指示各种详尽的处理器特性。此时，检查 EDX 寄存器的第 16 位（bit 15），当它是“1”时，表明处理器支持 `cmovecc` 指令。

条件转移指令和传送指令相结合的产物，既有条件转移指令的多样性，又执行的是传送操作。但是，和 `mov` 指令不同的是，它的目的操作数只允许是 16 位或者 32 位通用寄存器，源操作数只能是相同宽度的通用寄存器和内存单元，以下是几个常用的例子：

<code>cmovez ax,cx</code>	; 为零则传送
<code>cmovenz eax,[0x2000]</code>	; 不为零则传送
<code>cmove ebx,ecx</code>	; 相等则传送
<code>cmoveg cx,[0x1000]</code>	; 不大于则传送
<code>cmove l edx,ecx</code>	; 小于则传送

条件传送指令是很多的。在第 6 章的表 6-1 中，列举了所有的条件转移指令。完整的 `cmovecc` 指令列表，可以在表 6-1 的基础上，将那些指令的首字母“j”换成“cmove”即可。

`cmovecc` 指令不影响 EFLAGS 寄存器中的任何标志位。相反地，它的执行过程要依赖于这些标志，就象条件转移指令一样。

言归正传，为了不使用条件转移指令而又能算出用户程序实际占用的扇区数，需要一点技



巧。考察一下，你会发现，所有能被 512 整除的数，其最低端的 9 个比特都是“0”。比如：

0x200 (对应的十进制数为 512)	-> 0000 0010 0000 0000B
0x400 (对应的十进制数为 1024)	-> 0000 0100 0000 0000B
0x600 (对应的十进制数为 1536)	-> 0000 0110 0000 0000B
0x800 (对应的十进制数为 2048)	-> 0000 1000 0000 0000B
0xE00 (对应的十进制数为 3584)	-> 0000 1110 0000 0000B

很好，第 408 行，将用户程序的总大小从 EAX 寄存器传送到 EBX 寄存器，等于是做个备份，因为后面还要用到；第 409、410 行，先用 and 指令将其最低的 9 个比特清零，等于是去掉那些不足 512 的零头，然后，再将其加上 512，等于是将那些零头凑整。

但是，若人家原本就是 512 的整数倍，你这么做无疑是多加了一个扇区。因此，第 411、412 行，先测试 EAX 寄存器的最低 9 个比特，如果测试的结果是它们不全为零，则采用凑整的结果；如果为全零，则 cmovcc 指令什么也不做，依然采用用户程序原本的长度值。

13.4.3 简单的动态内存分配

下面的工作是把用户程序从硬盘上读到内存中。我们以前的做法是指定一个区域，比如物理地址 0x100000，然后把程序加载到那里。如果要加载的程序很多，这就会成为一种需要仔细规划的工作，每个程序加载到哪里，都需要一一指定。

在流行的操作系统里，内存管理是一项重要而又严肃的工作，不用说也相当复杂。它要记住所有可以分配的内存，将它们分成块。这样，当要求分配内存时，内存管理程序将查找并分配那些大小相符的空闲块；当占用这些块的用户终止执行后，还要负责回收它们，以便再用于分配；当内存空间紧张，找不到空闲块，或者空闲块的大小不能满足需求时，内存管理程序还要负责查找那些很少被访问的块，将其中的数据移到硬盘上，腾出空间来满足当前的需求。下次当这些块再次被用到时，再用同样的办法从硬盘调回内存。

讲了这么多，你可能以为我们现在就要写一个内存管理程序。不，不会的，这不太现实。就我们目前的需求来说，只需要一个简单的内存分配程序就可以了，这就是 `allocate_memory` 例程。

`allocate_memory` 例程位于代码清单 13-2 的公共例程段中，它仅仅需要通过 ECX 寄存器传入希望分配的字节数。当过程返回时，ECX 寄存器包含了所分配内存的起始物理地址。

`allocate_memory` 的内存分配策略非常简单。请看代码清单 13-2 的第 335 行，在内核数据段中声明了标号 `ram_alloc`，并初始化为一个双字 0x00100000，这就是可用于分配的初始内存地址。很显然，这个位置正好在 1MB 之外。每次请求分配内存时，`allocate_memory` 过程仅简单地返回该内存单元的值，作为所分配内存的起始地址。同时，将这个值加上所分配的长度，作为下次分配的起始地址写回该内存单元。

因此，在进行了必要的现场压栈保护之后，第 239~247 行，先使段寄存器 DS 指向内核数据段以访问标号 `ram_alloc` 所指向的内存单元；然后，计算下次可用于分配的起始内存地址并存放到 EAX 寄存器中；最后，在 ECX 中得到本次分配到的起始内存地址，这个值将返回给调用者。当然，在这个过程中没有检测是否超越了实际拥有的物理内存。我们的程序都非常小，现在哪台计算机没有几十兆、几百兆甚至几个吉的内存呢？

原则上，将 EAX 寄存器中的值写回 `ram_alloc` 所指向的双字单元即可。不过，32 位的计算机系统建议内存地址最好是 4 字节对齐的，这样做好处是访问速度最快。为此，在将 EAX 寄

存器的值写回内存之前，最好使之成为可被 4 整除的值，这种数值的特点是最低两个比特为“0”。

第 249~254 行，先将 EAX 寄存器的内容传送到 EBX 进行备份；接着，强制 EBX 中的地址对齐在下一个 4 字节边界，对齐之后的值肯定会比原先大；然后，看一看原始分配的起始地址（在 EAX 寄存器中）是否是 4 字节对齐的，如果不是，就采用对齐之后的值；如果原本就是 4 字节对齐的，那么，依然采用原值；最后，将这个值写回到原内存单元中，做为下次内存分配的起始地址。

过程 `allocate_memory` 是用 `retf` 指令返回的。因此，它只能通过远过程调用来进入。

13.4.4 段的重定位和描述符的创建

接着回到 `load_relocate_program` 过程。

在 13.4.2 节里，我们算出了用户程序的总长度，而且已经被调整为可以被 512 整除的数。第 414、415 行，用这个数值去调用 `allocate_memory` 过程分配内存。分配到手的内存块，起始地址在 ECX 寄存器中。

第 416 行，将 ECX 寄存器的内容传送到 EBX，其动机是作为起始地址从硬盘上加载整个用户程序。

第 417 行，将该首地址压栈保存，其目的是用于在后面访问用户程序头部。

第 418~420 行，用户程序的总长度除以 512，得到它所占用的扇区总数。

第 421 行，将扇区数传送到 ECX 寄存器，用于控制后面的循环次数。该循环是用来加载整个用户程序的。

第 423、424 行，使段寄存器 DS 指向 4GB 的内存段，这样就可以加载用户程序了。

第 428~430 行，循环读取硬盘以加载用户程序。读取的次数由 ECX 控制；加载之前，其首地址已经位于 EBX 寄存器。起始逻辑扇区号原本是通过 ESI 寄存器传入的，循环开始之前已经传送到 EAX 寄存器（第 426 行）。

既然用户程序已经全部读入内存，现在的任务就是根据它的头部信息来创建段描述符。

第 433 行，从堆栈中弹出用户程序首地址到 EDI 寄存器，它是在前面第 417 行压入的，该地址也是用户程序头部的起始地址。

第 434~438 行，读用户程序头部信息，根据这些信息创建头部段描述符。在主引导程序里，有一个创建描述符的例程，在内核中，也编写了一个同样的例程 `make_seg_descriptor`，甚至它们所用的指令都一模一样。它属于公共例程段，是在第 308 行定义的。

该过程要求用 EAX 寄存器传入段的基址，这是第 434 行的工作。段界限由 EBX 寄存器传入，第 435、436 行访问 4GB 内存段，从用户程序头部偏移 0x04 处取出段长度，减一后形成段界限。第 437 行用于给出头部段的属性值。

从过程返回时，EDX:EAX 中包含了 64 位的段描述符。紧接着，第 439 行调用公共例程段内的另一个过程 `set_up_gdt_descriptor`，把该描述符安装到 GDT 中。

`set_up_gdt_descriptor` 也属于公共例程段，是在第 263 行定义的，它需要通过 EDX:EAX 传入描述符作为唯一的参数。该过程返回时，CX 寄存器中包含了那个描述符的选择子。

要在 GDT 内安装描述符，必须知道它的物理地址和大小。而要知道这些信息，可以使用指令 `sgdt`（Store Global Descriptor Table Register），它用于将 GDTR 寄存器的基地址和边界信息保存到指定的内存位置。`sgdt` 指令的格式为

```
sgdt m
```

其中，m 是一个 6 字节内存区域的首地址。该指令不影响任何标志位。



第 332、333 行，在内核数据段中，声明了标号 pgdt，并初始化了 6 字节，供 sgdt 指令使用。低 2 字节用于保存 GDT 的界限（大小）；高 4 字节用于保存 GDT 的 32 位物理地址。

回到例程 set_up_gdt_descriptor 中。第 270~276 行，在压栈保存了 DS 和 ES 的原始内容后，使 DS 指向内核数据段。紧接着，使用 sgdt 指令取得 GDT 的基地址和大小。

第 278、279 行，使段寄存器 ES 指向 4GB 内存段以操作全局描述符表（GDT）。

下面的工作是计算描述符的安装地址。这个地址可以这样计算：先得到描述符表的界限值，将它加一，得到描述符表的总字节数，这实际上也是新描述符在 GDT 内的偏移量。然后，用 GDT 的线性地址加上这个偏移量，就是用于安装新描述符的线性地址。

第 281 行，先访问内核数据段，取得 GDT 的界限值。注意，这里出现了一个新指令 movzx，其作用是带零扩展的传送（Move with Zero-Extend），指令格式为

```
movzx r16,r/m8
movzx r32,r/m8
movzx r32,r/m16
```

也就是说，movzx 指令的目的操作数只能是 16 位或者 32 位的通用寄存器，源操作数只能是 8 位或者 16 位的通用寄存器，或者指向一个 8 位或 16 位内存单元的地址。而且，很有意思的是，目的操作数和源操作数的大小是不同的。这里有几个例子：

```
movzx cx,al
movzx eax,byte [0x2000]
movzx ecx,bx
```

对于上面的第一个例子，如果指令执行前，AL 寄存器的内容是 0xC0，那么，指令执行后，CX 寄存器的内容为 0x00C0；对于第二个例子，处理器访问段寄存器 DS 所指向的段，从偏移地址 0x2000 处取得一字节，左边添加 24 个“0”，使之扩展到 32 位，然后传送到 EAX 寄存器；对于第三个例子，如果指令执行前，BX 寄存器的内容为 0x55AA，那么，指令执行后，ECX 寄存器的内容为 0x000055AA。

另一个非常有用的指令是 movsx，意思是带符号扩展的传送（Move with Sign-Extension），指令格式为

```
movsx r16,r/m8
movsx r32,r/m8
movsx r32,r/m16
```

和 movzx 不同，movsx 在执行扩展时，用于扩展的比特取自源操作数的符号位。比如

```
mov al,0x08
movsx cx,al      ;CX=0x0008，因为 AL 的最高位是“0”
```

```
mov al,0xf5
movsx ecx,al      ;ECX=0xFFFFFFFF5，因为 AL 的最高位是“1”
```

GDT 的界限是 16 位的，允许 64KB 的大小，即 8192 个描述符，似乎不需要使用 32 位的寄存器 EBX。事实上，还是需要的，因为后面要用它来计算新描述符的 32 位线性地址，加法指令 add 要求的是两个 32 位操作数。

第 282 行，将 GDT 的界限值加 1，就是 GDT 的总字节数，也是新描述符在 GDT 内的偏移量。不过，很奇怪的是，我们用的是指令

```
inc bx
```

而不是

```
inc ebx
```

这是为什么呢？

这是有道理的。就一般的情况来说，在这里用这两条指令的哪一条，都没有问题。但是，如果这是启动计算机以来，第一次在 GDT 中安装描述符，可能就会产生问题。在初始状态下，也就是计算机启动之后，这时还没有使用 GDT，GDTR 寄存器中的基地址为 0x00000000，界限为 0xFFFF（事实上，在 BIOS 执行硬件检测时，会设置 GDT，对保护模式进行测试。至于是否恢复 GDTR 的原始内容，尚不清楚）。

当 GDTR 寄存器的界限部分是 0xFFFF 时，表明 GDT 中还没有描述符。因此，将此值加 1，结果是 0x10000，由于该寄存器的界限部分只有 16 位，所以只能容纳 16 位的结果，即 0x0000，这就是第一个描述符在表内的偏移量。

同样的道理，因为 EBX 寄存器中的内容是 GDT 的界限值 0x0000FFFF，如果执行的是指令

```
inc ebx
```

那么，EBX 寄存器中的内容将是 0x00010000，以它作为第一个描述符的偏移量显然是不对的。相反，如果执行的是指令是

```
inc bx
```

那么，因为 BX 寄存器只有 16 位，故，结果为 0x0000，进位被丢弃（决不会影响 EBX 寄存器的高 16 位）。此指令执行后，EBX 寄存器的内容是 0x00000000。

第 283 行，用计算出来的偏移量加上 GDT 的基地址，结果就是新描述符的线性地址。事实上，这三行或许可以按以下方法来简单处理，就没那么啰嗦了：

```
xor ebx,ebx  
mov bx,[pgdt] ;GDT 界限  
inc bx ;GDT 总字节数，也是下一个描述符偏移  
add ebx,[pgdt+2] ;下一个描述符的线性地址
```

但是，少用一条指令似乎更好，谁知道呢！

既然已经知道新描述符应该安装在哪里，第 285、286 行，访问段寄存器 ES 所指向的 4GB 内存段，将 EDX:EAX 中的 64 位描述符写入由 EDI 寄存器所指向的偏移处。

第 288~290 行，访问内核数据段，将 GDT 的界限值加上 8，然后用 lgdt 指令重新加载 GDTR，使新的描述符生效。GDTR 寄存器中的界限值总是单数（8 的整数倍减 1），包括它的初始值 0xFFFF。所以，每次只要加上新描述符的实际大小就能得到正确的界限值。

最后，第 292~297 行，根据 GDT 的新界限值，来生成相应的段选择子。具体的算法是，取得 GDT 的当前界限值，除以 8，余数丢弃。描述符的索引是从 0 开始编号的，界限值总是比 GDT 的总字节数小 1。因此，界限值除以 8，一定会有余数（余 7，丢弃不用），商就是我们所要得到的描述符索引号。最后，将索引号左移 3 次，留出 TI 位和 RPL 位（TI=0，指向 GDT，RPL=00），这就是要生成的选择子。

第 299~306 行，恢复调用之前的现场，返回调用者。返回时用了 retf 指令，因此，本过程只能通过远过程调用的方式进入。

继续回到过程 load_relocate_program。

安装了用户程序头部段的描述符后，第 440 行，将该段的选择子写回到用户程序头部，供用户程序在接管处理器控制权之后使用。实际上，在内核向用户程序转交控制权时，也要用到。

第 443~460 行，用于重定位用户程序代码段和数据段，并创建和安装相应的描述符，整个过程都是一样的，也很容易理解。



唯一不同的是堆栈段，堆栈所用的空间不需要用户程序提供，而是由内核动态分配。内核分配堆栈空间时，是以 4KB 为单位的，也就是说，每次分配至少是 4KB 的倍数。至于到底分配多少，用户程序应该根据自己的实际需求提出建议。

第 463 行，从用户程序头部偏移为 0x0C 的地方获得一个建议的堆栈大小。这是一个倍率，至少应当为 1，说明用户程序希望分配 4KB 堆栈。如果为 2，说明希望分配 8KB；为 3 则表明希望分配 12KB，依此类推。

第 464、465 行，计算堆栈段的界限。如果堆栈段的粒度是 4KB，那么，用 0xFFFF 减去倍率，就是用来创建描述符的段界限。举例来说，如果用户程序建议的倍率是 2，那么，这意味着他想创建的堆栈空间为 8KB。因此，段的界限值为

$$0xFFFF - 2 = 0xFFFFD$$

那么，当处理器访问该堆栈段时，实际使用的段界限为

$$0xFFFFD \times 0x1000 + 0xFFFF = 0xFFFFFDFFF$$

堆栈是向下扩展的，访问 32 位的堆栈，要使用堆栈指针寄存器 ESP，其最大值是 0xFFFFFFFF。因此，ESP 的值只允许在 0xFFFFDFFF 和 0xFFFFFFFF 之间变化，共 8KB。

第 466~469 行，用 4096（4KB）乘以倍率，得到所需要的堆栈大小，然后，用这个值去申请内存。这是一个 32 位的无符号数乘法，指令格式为

```
mul r/m32
```

这里，用 EAX 寄存器的值，乘以另一个 32 位的数（可以在通用寄存器或者内存单元里），在 EDX:EAX 中得到 64 位的乘法结果。

注意，allocate_memory 过程返回所分配内存的低端地址。和一般的数据段不同，堆栈描述符中的基址，应当是堆栈空间的高端地址。所以，第 470 行，用 allocate_memory 返回的低端地址，加上堆栈的大小，得到堆栈空间的高端地址。

第 471~473 行，依次调用两个例程，来生成和安装堆栈段的描述符。注意堆栈的属性值，它指明了这是一个 32 位的堆栈段，粒度为 4KB。

第 474 行，将堆栈段的选择子写回到用户程序头部，供用户程序在接管处理器控制权之后使用。

13.4.5 重定位用户程序内的符号地址

为了使用内核提供的例程，用户程序需要建立一个符号-地址对照表（SALT）。这样，当用户程序加载后，内核应该根据这些符号名来回填它们对应的入口地址，这称为符号地址的重定位。显然，重定位的过程就是字符串匹配和比较的过程。

为了对用户程序内的符号名进行匹配，内核也必须建立一张符号-地址对照表（SALT）。

内核的 SALT 表位于代码清单 13-2 的内核数据段中，从第 338 行开始，一直到第 357 行结束。实际上，这个表是可以根据需要扩展的。

如图 13-9 所示，用户程序内的 SALT 表，每个条目是 256 字节，用于容纳符号名，不足 256 字节的，用零填充。内核中的 SALT 表，每个条目则包括两部分，第一部分也是 256 字节的符号名；第二部分有 6 字节，用于容纳 4 字节的偏移地址和 2 字节的段选择子，因为符号名是用来描述例程的，这 6 字节就是例程的入口地址。

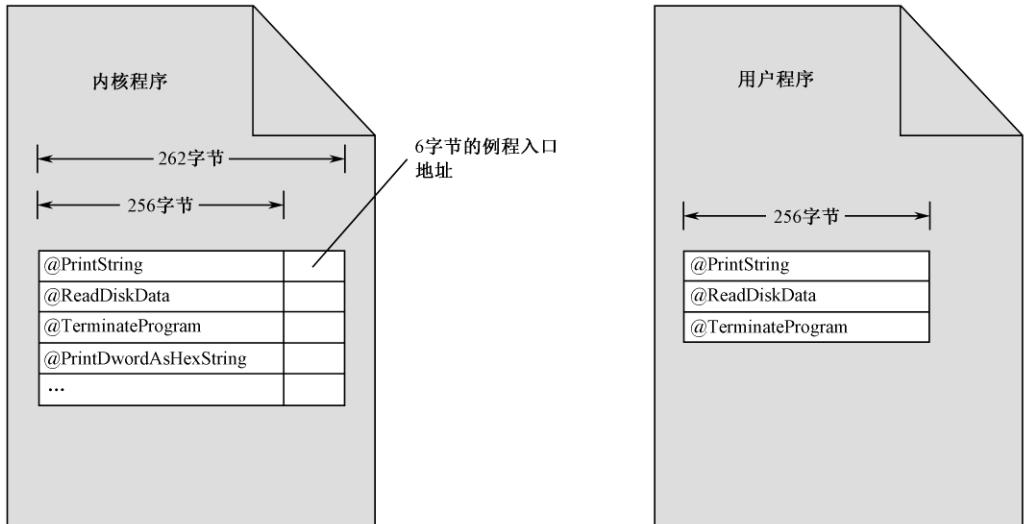


图 13-9 内核和用户程序内的符号表结构

举个内核中的例子：

```

salt_1 db '@PrintString'
times 256-($-salt_1) db 0
dd put_string
dw sys_routine_seg_sel

```

这是内核 SALT 表的第一个条目。它初始化了一个 256 字节的符号名，该名称的前 12 个字符是“@PrintString”，因为不足 256 字节，后面填充 244 个 0x00。

在该条目的后面，先是一个双字，初始化为 put_string 例程的偏移地址。这就是说，PrintString 其实就是 put_string 的别名，调用 PrintString，其实是调用 put_string 例程。在用户程序内，只能通过远过程调用来进入该例程，所以，该条目的最后是一个字，用公共例程段的选择子来初始化，因为 put_string 例程位于公共例程段。

在内核 SALT 表中，比较有意思的是最后一个条目：

```

salt_4 db '@TerminateProgram'
times 256-($-salt_4) db 0
dd return_point
dw core_code_seg_sel

```

在这里，从名字可以看出，“TerminateProgram”的意思是终止程序。当用户程序调用该过程时，意味着结束用户程序，将控制返回到内核。

当用户程序终止并返回时，返回点位于标号 return_point 所在的位置。该标号位于第 582 行，属于内核代码段。在这一行之前，是内核将控制权交给用户程序的指令。

内核的 SALT 表是静态的，适用于所有要加载的用户程序，理所当然地要比用户程序的 SALT 表大，因为它要提供所有可被用户程序调用的过程列表。至于用户程序，根据需要，它只会列出自己用到的那些。

在用户程序加载时，内核的任务是比对这两张 SALT 表，并将用户程序 SALT 表中的符号名替换成相应的入口地址。为了便于说明，用户程序的 SALT 表简称 U-SALT，内核的 SALT 表简称 C-SALT。

基本的算法是使用内外层循环，外循环依次从 U-SALT 表中取出条目，每取出一个条目，



就进入内循环进行比对；内循环遍历 C-SALT 中的每一个条目，同外循环输入的条目进行比对。

比对的过程就是两个字符串的比较过程，可以使用 `cmps` 指令（Compare String Operands）。该指令有 3 种基本的形式，分别用于字节、字和双字的比较：

<code>cmpsb</code>	; 字节比较
<code>cmplw</code>	; 字比较
<code>cmpsd</code>	; 双字比较

在 16 位模式中，源字符串的首地址由 DS:SI 指定，目的字符串的首地址由 ES:DI 指定；在 32 位模式下，则分别是 DS:ESI 和 ES:EDI。在处理器内部，`cmps` 指令的操作是把两个操作数相减，然后根据结果设置标志寄存器中相应的标志位。

取决于标志寄存器 EFLAGS 中的 DF 位，如果 $DF=0$ ，表明是正向比较，也就是按地址递增的方向比较，这些指令执行后，SI (ESI) 和 DI (EDI) 的内容分别加 1、加 2 和加 4；否则，如果 $DF=1$ ，表明是反向比较，这些指令执行后，SI (ESI) 和 DI (EDI) 的内容分别减 1、减 2 和减 4。

单纯的 `cmps` 指令只比较一次，它属于推一下才动一动的那种类型。所以，需要加指令前缀 `rep` 使比较连续进行。连续比较的次数由 CX (ECX) 寄存器控制，在 16 位模式下，使用 CX 寄存器；在 32 位模式下，使用 ECX 寄存器，举个例子：

```
[bits 32]
rep cmpsd
```

该指令执行时，每次比较 4 字节，连续比较直至 ECX 寄存器的内容为零。

问题是，用 `rep` 前缀比不出个所以然来，你就是重复比较 100000 次，也看不出两个字符串哪里不同。所以，针对 `cmps` 指令，应当使用 `repe` (`repz`) 和 `repne` (`repnz`) 前缀，前者的意思是“若相等（为零）则重复”，后者的意思是“若不等（非零）则重复”。但无论是哪种情况，总的比较次数由 CX (ECX) 控制，表 13-1 显示了这几种控制手段的区别。

表 13-1 重复前缀

重复前缀	终止条件一	终止条件二
<code>rep</code>	$(E)CX=0$	无
<code>repz/repe</code>	$(E)CX=0$	$ZF=0$
<code>repnz/repne</code>	$(E)CX=0$	$ZF=1$

可见，`repe/repz` 用于搜索第一个不匹配的字节、字或者双字，`repne/repnz` 用于搜索第一个匹配的字节、字或者双字。无论如何，匹配和不匹配的位置分别由(E)SI 和(E)DI 寄存器指示。

言归正传，我们继续回到代码清单 13-2 中来。

如图 13-10 所示，为了重定位 U-SALT，我们打算用 DS:ESI 指向 C-SALT，用 ES:EDI 指向 U-SALT。第 477、478 行，访问 4GB 内存段，从用户程序头部偏移为 0x04 的地方取出刚刚安装好的头部段选择子，并使段寄存器 ES 指向用户程序头部段，因为 U-SALT 位于用户程序头部段内。

第 479、480 行，使段寄存器 DS 指向内核数据段。因为 C-SALT 位于内核数据段中。

第 482 行，清标志寄存器 EFLAGS 中的方向标志，使 `cmps` 指令按正向进行比较。

实施比较的算法我们已经介绍过了。外循环的作用是依次从 U-SALT 中取出各个条目，因此，第 484 行，将取的次数（条目的个数）从用户程序头部取出，传送到 ECX 寄存器。

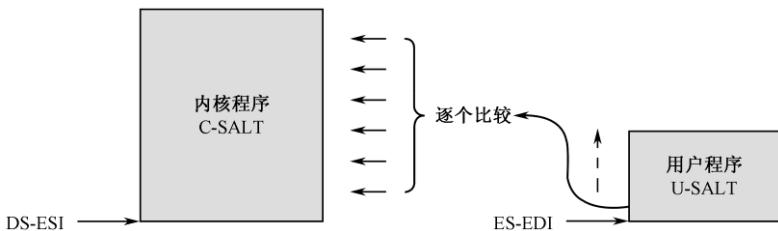


图 13-10 U-SALT 和 C-SALT 的比对过程示意图

接着，第 485 行，用于将 U-SALT 在头部段内的偏移量传送到 EDI 寄存器。刚才我们已经使段寄存器 ES 指向了头部段。

外循环的结构如下所示，这是从代码清单中抽出来的，行号也保持不变。

```
486 .b2:
```

```
487     push ecx
```

```
488     push edi
```

```
489
```

;此处放置内循环代码，用于实际进行比较。

```
512     pop edi
```

```
513     add edi,256
```

```
514     pop ecx
```

```
515     loop .b2
```

由于内循环也要使用 ECX 和 EDI 寄存器，并有可能破坏它们的内容，因此，在进入内循环之前，要对它们压栈保护，以便退出内循环后继续使用。外循环的任务是从 U-SALT 中依次取出表项，因此，当内循环完成比对后，第 512、513 行，从堆栈中弹出 EDI 寄存器的原始内容，并加上 256，以指向下一个条目。第 514、515 行，从堆栈中弹出 ECX 寄存器的原值。loop 指令将 ECX 的内容减一，根据结果判断是否继续循环。

对于外循环所指向的每一个条目，内循环要用它和 C-SALT 中的所有条目进行比对，内循环的代码如下：

```
490     mov ecx,salt_items
```

```
491     mov esi,salt
```

```
492 .b3:
```

```
493     push edi
```

```
494     push esi
```

```
495     push ecx
```

;这里放置实际进行比对的代码

```
506     pop ecx
```

```
507     pop esi
```

```
508     add esi,salt_item_len
```

```
509     pop edi
```



510 loop .b3

每次从外循环进入内循环时，都要重新设置比对次数，并重新使 ESI 寄存器指向 C-SALT 的开始处，这是第 490、491 行的工作。标号 salt_item_len 是在第 359 行声明的，并用一个表达式初始化。每个条目的长度都是相同的，用当前汇编地址减去标号 salt_4 的汇编地址，即\$-salt_4，就是每个条目的长度（字节数）。事实上，这个数值是在编译阶段由编译器计算的，在数值上等于 262。

标号 salt_items 是在第 360 行声明的，并初始化为一个表达式。该表达式的意思是，用整个 C-SALT 的长度，除以每个条目的长度，就是条目的个数。

对于内循环的每一次执行，都要把 ESI、EDI 和 ECX 压栈保护，以免在比对的过程中用到并破坏这些寄存器。每次比对结束后，第 506~509 行，依次弹出这些寄存器的值，并把 ESI 的内容加上 C-SALT 每个条目的长度（262 字节），以指向下一个 C-SALT 条目。第 510 行，loop 指令执行时，将 ECX 的内容减一并判断是否继续循环。

第 497~503 行，是整个比对过程的核心部分。每当处理器执行到这里时，DS:ESI 和 ES:EDI 都各自指向 C-SALT 和 U-SALT 中的某个条目：

```
497     mov ecx, 64
498     repe cmpsd
499     jnz .b4
500     mov eax, [esi]
501     mov [es:edi-256],eax
502     mov ax,[esi+4]
503     mov [es:edi-252],ax
504 .b4:
```

因为每个条目的符号名部分是 256 字节，每次用 cmpsd 指令比较 4 字节，故每个条目至多需要比对 64 次。第 497 行把立即数 64 传送到 ECX 寄存器以控制整个比对过程。

第 498 行，开始比对，直到发现一个不相符的地方。

如果两个字符串相同，则需要连续比对 64 次，而且，在比对结束时， $ZF=1$ ，表示最后 4 字节也相同；如果两个字符串不同，比对过程会提前结束，且 $ZF=0$ 。在最坏的情况下，这两个字符串可能只有最后 4 字节是不同的。在这种情况下，也需要比对 64 次，但 $ZF=0$ 。

无论哪种情况，如果在退出 repe cmpsd 指令时 ZF=0，即表明两个字符串是不同的。所以，第 499 行，如果 ZF=0，则表明两个字符串不同，直接转移到内循环的末尾，以开始下一次内循环。

如果两个字符串是相同的，那么，比较指令执行后，ESI 寄存器正好指向 C-SALT 每个条目后的入口数据。要知道，C-SALT 中的每个条目是 262 字节，最后的 6 字节分别是偏移地址和段选择子。

因此，现在的任务是将这结尾的 6 字节传送到 U-SALT 当前条目的开始部分，这是第 500~503 行的工作。最后的结果是，U-SALT 中的当前条目，其开始的 6 字节被改写为一个入口地址。

13.5 执行用户程序

在 load_relocate_program 过程的最后，第 517 行，把用户程序头部段的选择子传送到 AX 寄存器。第 519~528 行，从堆栈中弹出并恢复各个寄存器的原始内容，并返回到调用者。AX 寄存器中的选择子是作为参数返回到主程序的。主程序将用它来找到用户程序的入口，并从那里进入。

从 load_relocate_program 过程返回后，第 572、573 行用于在屏幕上显示信息，表示加载和重定位工作已经完成。

第 575 行，保存内核的堆栈指针。这是通过将 ESP 寄存器的当前值写入内核数据段中来完成的。写入的位置是由标号 esp_pointer 指示的，位于第 378 行，初始化为一个双字。在进入用户程序后，用户程序应当切换到它自己的堆栈。从用户程序返回时，还要从这个内存位置还原内核堆栈指针。

第 577 行，使段寄存器 DS 指向用户程序头部。这是通过将用户程序头部段选择子传送到 DS 来办到的。在用户程序头部段内偏移 0x10 处，是用户程序的入口点，分别是 32 位的偏移量和 16 位的代码段选择子。第 579 行，执行一个间接远转移，进入用户程序内接着执行。

现在转到代码清单 13-3。

用户程序的入口点是在第 56 行。进入用户程序开始执行时，段寄存器 DS 是指向头部段的。第 57、58 行，使段寄存器 FS 指向头部段，因为后面要调用内核过程，而这些过程都要求使用 DS，所以要把 DS 解放出来。

第 60~62 行，切换到用户程序自己的堆栈，并初始化堆栈指针寄存器 ESP 的内容为 0。

第 64、65 行，设置段寄存器 DS 到用户程序自己的数据段。

第 67、68 行，调用内核过程显示字符串，以表明用户程序正在运行中。该内核过程要求用 DS:EBX 指向零终止的字符串。

第 70~72 行，调用内核过程，从硬盘读一个扇区。从内核代码清单可以知道，ReadDiskData 过程的内部名称是 read_hard_disk_0。所以，ReadDiskData 需要传入两个参数，第一个是 EAX 寄存器，传入要读的逻辑扇区号；第二个是 DS:EBX，传入缓冲区的首地址，毕竟读出来的数据要有个地方保存。缓冲区位于用户程序的数据段中，是在第 43 行用标号 buffer 声明的，并初始化了 1024 字节的空间。要读的逻辑扇区号是 100，在此之前，我们应当在这个扇区里写一些东西。这件事我们马上就要讲到。

第 74~78 行，先调用内核过程显示一个题头，接着，再次调用内核过程显示刚刚从硬盘读出的内容。

在做完了上述事情之后，用户程序的任务也就完成了。第 80 行，调用内核过程，以返回到内核。

再次回到代码清单 13-2。

在内核中，用户程序的返回点位于第 582 行。

在重新接管了处理器的控制权后，第 583、584 行，使段寄存器 DS 重新指向内核数据段。

第 586~588 行，切换堆栈，使堆栈段寄存器 SS 重新指向内核堆栈段，并从内核数据



段中取得和恢复原先的堆栈指针位置。

第 590、591 行，显示一条消息，表示现在已经回到了内核。

对于一个操作系统来说，下面的任务是回收前一个用户程序所占用的内存，并启动下一个用户程序。但是，我们现在无事可做，所以，第 596 行，使处理器进入停机状态。别忘了，在进入保护模式之前，我们已经用 cli 指令关闭了中断，所以，除非有 NMI 产生，处理器将一直处于停机状态。

13.6 代码的编译、运行和调试

首先编译本章所有的源程序文件，它们是 c13_mbr.asm、c13_core.asm 和 c13.asm，这将分别生成 c13_mbr.bin、c13_core.bin 以及 c13.bin。

使用配书工具 FixVhdWr 分别将这些二进制文件写入虚拟硬盘。c13_mbr.bin 的起始逻辑扇区号是 0，因为它是主引导代码；c13_core.bin 的起始逻辑扇区号是 1；c13.bin 的起始逻辑扇区号是 50。除了 c13_mbr.bin 外，其他文件的写入位置可以改变，但前提是需要修改使用它们的源代码。

用户程序的功能是读取逻辑扇区 100，并显示其内容。为此，需要找一个文本文件，并将它写入该扇区。在配书源代码中，提供了一个文本文件 diskdata.txt，其大小是 512 字节。如图 13-11 所示，它包含了 512 字节的英文文本。

不强迫你一定要使用这个文件。你完全可以选用其他文件，文件的内容也无所谓，但最好是可读的 ASCII 字符。

使用配书工具 FixVhdWr 将你采用的文本文件写入虚拟硬盘，逻辑扇区号是 100。如果你采用的是其他文件，它或许很长，会连续写入多个扇区。这无所谓，用户程序只读取第一个。

最后，启动虚拟机时，如果一切正常，所显示的画面将如图 13-12 所示。

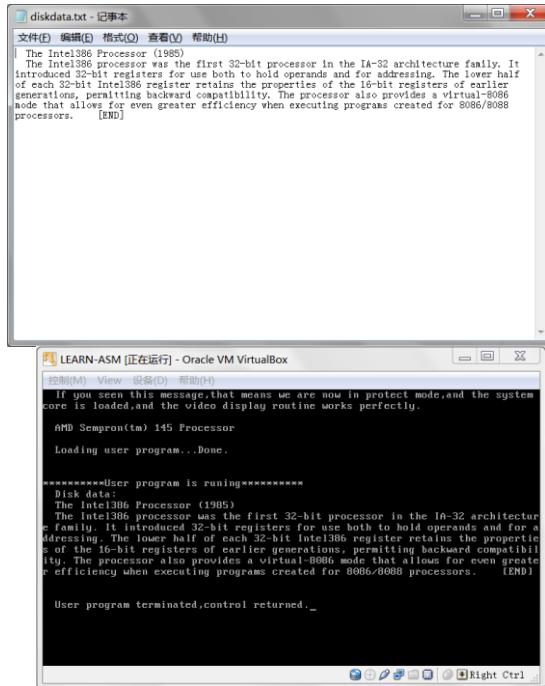


图 13-11 diskdata.txt 文件的内容

图 13-12 本章程序的运行结果

具有讽刺意味的是，我在这里大书特书、侃侃而谈 INTEL 的处理器，但是，从截图上可以看出，我用的处理器却是 AMD 生产的。至于你的计算机用了什么处理器，你自己看看吧，屏幕上的显示会说明一切的。

随着程序代码量的增大，程序的编写和调试也会变得越来越困难。特别是当问题发生的时候，追查出错的位置和错误的原因都需要花费大量的时间、消耗大量的精力。



有时候，最简单的方法却很有效。比如，可以写一个特殊的过程，用来显示某个寄存器的内容。如果你的程序运行时出了问题，可以在有重大嫌疑的指令前后安排一些调用该过程的代码，看看是哪里不正常。这些用于调试程序的位置，叫做检查点。

为了方便调试程序，代码清单 13-2 提供了一个过程 `put_hex_dword`，用于以十六进制的形式显示 EDX 寄存器的内容。

该过程位于第 202 行，它的工作原理很简单，EDX 寄存器是 32 位的，从右到左，将它以 4 位为一组，分成 8 组。每一组的值都在 $0 \sim 15$ ($0x0 \sim 0xf$) 之间，我们把它转换成相应的字符'0' ~ 'F'即可。

为了将数值转换成可显示的 ASCII 码，可以使用处理器的查表指令 `xlat` (Table Look-up Translation)，该指令要求事先在 DS:(E)BX 处定义一个用于转换编码的表格，在 16 位模式下，使用 BX 寄存器；在 32 位模式下，使用 EBX 寄存器。指令执行时，处理器访问该表格，用 AL 寄存器的内容作为偏移量，从表格中取出一字节，传回 AL 寄存器。

代码清单 13-2 定义的表格在第 374 行。在那里，声明了标号 `bin_hex`，并初始化了 16 个字符，这是一个二进制到十六进制的对照（检索）表。偏移（索引）为 0 的位置是字符'0'；偏移（索引）为 $0x0f$ 的位置是字符'F'。

第 209、210 行，使段寄存器 DS 指向内核数据段，因为对照表 `bin_hex` 位于内核数据段中。

第 212 行，使 EBX 寄存器指向检索（对照表）的起始处。

转换过程使用了循环，每次将 EDX 寄存器的内容循环左移 4 位，共需要循环 8 次。每次移位后的内容被传送到 EAX 寄存器，并用 `and` 指令保留低 4 位，高位清零。第 218 行，`xlat` 指令用 AL 寄存器中的值作为索引访问对照表，取出相应的字符，并回传到 AL 寄存器。

每次从检索（对照）表中得到一个字符，就要调用 `put_char` 过程显示它。但 `put_char` 过程需要使用 CL 寄存器作为参数。因此，第 220 行，在显示之前先要将 ECX 寄存器压栈保护。

`xlat` 指令不影响任何标志位。

市面上有很多高级的调试工具，但都不适合在本书中使用。原因很简单，它们都要求运行在一些流行的操作系统上，如 Linux 和 Windows。而在本书中，一切都建立在裸机之上。

如果你愿意尝试，Bochs 是一个不错的选择。它也是一个虚拟机软件，特别地，它是用软件来模拟处理器的工作，它假装自己是一个真正的处理器，煞有介事地做取指令、译码和执行的工作。但它做得很好，和真正的处理器相比，仅仅是慢了很多。

Bochs 的另一个厉害之处是它的裸机调试功能。利用它，你可以单步执行本书中的程序、设置断点、显示处理器内部所有寄存器的内容、显示包括 GDT 在内的各种内存表等。

这是一个开源软件。下点工夫，相信你会喜欢它的。

本章习题

在本章中，用户程序只给出建议的堆栈大小，但并不提供堆栈空间。现在，修改内核程序和用户程序，改由用户程序自行提供堆栈空间。

要求：堆栈段必须定义在用户程序头部之后。

关于这道题，我想说的是，一定要考虑周全。

第 14-17 章內容从略