HZ BOOKS
华章教育

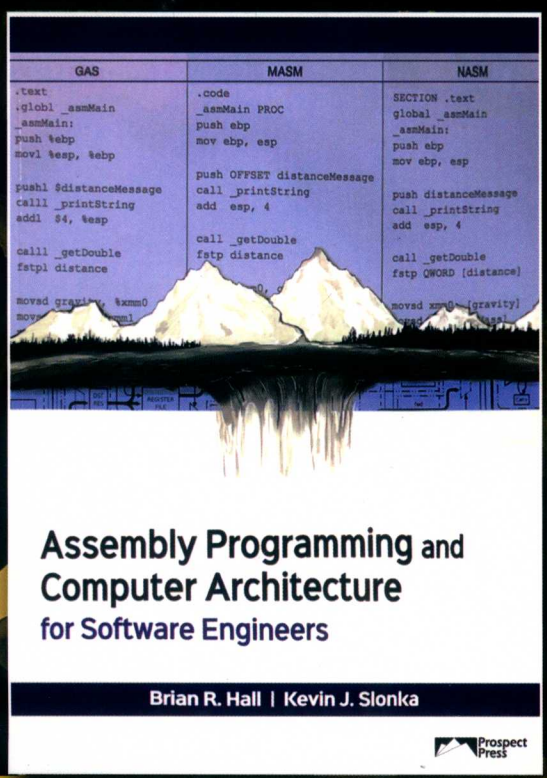
计 算 机 科 学 丛 书

# 汇编程序设计与计算机体系结构

## 软件工程师教程

[美] 布莱恩·R. 霍尔（Brian R. Hall）
凯文·J. 斯隆卡（Kevin J. Slonka） 著 爱飞翔 译

Assembly Programming and Computer Architecture for Software Engineers




机械工业出版社
China Machine Press

# 汇编程序设计与计算机体系结构 软件工程师教程

## Assembly Programming and Computer Architecture for Software Engineers

本书通过大量实例，循序渐进地讲解汇编语言的各种应用方式，并直观地演示怎样把汇编语言同高级语言结合，以制作出充分发挥硬件特性的程序。

本书的内容涵盖汇编语言与计算机体系结构的许多方面。从计算机编程语言与体系结构的基础知识开始，讲解计算机语言的转移流程、计算机组成部分、汇编语法、算术操作、流程控制、函数、字符串、结构体、浮点数运算等内容。之后讲解怎样把汇编与高级语言及系统软件结合起来使用，并介绍硬件及其元件的基本原理。

书中以x86与x86_64这两种主流架构为重点，兼顾AT&T及Intel语法，并适用于GAS、NASM及MASM三种常见的汇编器，以及Linux、macOS及Windows三种常用的操作系统，这使得身处各种开发环境中的软件工程师都能在书里找到可以直接运用的解决方案，并了解如何将其移植到其他环境。

华章教育服务微信号

投稿热线：(010) 88379604
购书热线：(010) 68326294
客服热线：(010) 88379426 88379833

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

封面设计 · 锡 林

上架指导：计算机\程序设计

ISBN 978-7-111-61516-3

9 787111 615163 >

定价：79.00元

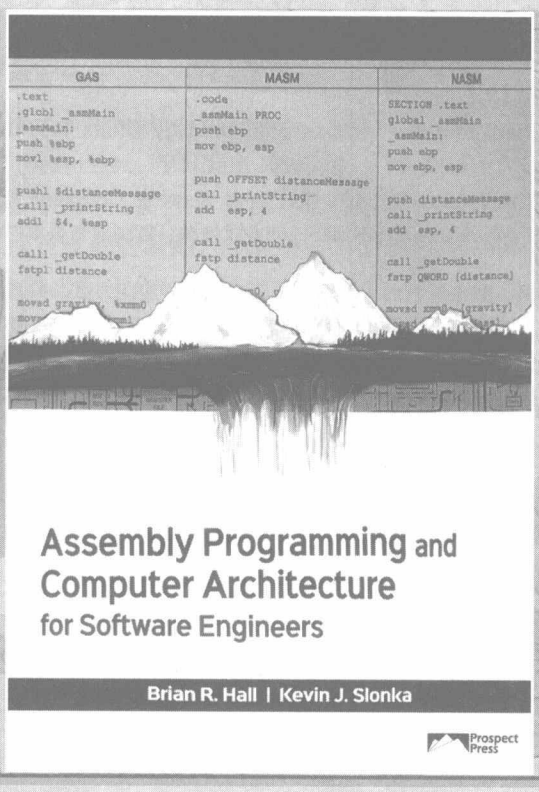
计 算 机 科 学 丛 书

# 汇编程序设计与计算机体系结构

## 软件工程师教程

[美] 布莱恩·R. 霍尔（Brian R. Hall）
凯文·J. 斯隆卡（Kevin J. Slonka） 著 爱飞翔 译

Assembly Programming and Computer Architecture for Software Engineers




机械工业出版社
China Machine Press

**图书在版编目（CIP）数据**

汇编程序设计与计算机体系结构：软件工程师教程 /（美）布莱恩·R. 霍尔（Brian R. Hall），（美）凯文·J. 斯隆卡（Kevin J. Slonka）著；爱飞翔译 . —北京：机械工业出版社，2019.1
（计算机科学丛书）
书名原文：Assembly Programming and Computer Architecture for Software Engineers

ISBN 978-7-111-61516-3

I. 汇… II. ① 布… ② 凯… ③ 爱… III. ① 汇编语言 - 程序设计 ② 计算机体系结构 IV. ① TP313 ② TP303

中国版本图书馆 CIP 数据核字（2018）第 276424 号

**本书版权登记号：图字 01-2018-4801**

Brian R. Hall, Kevin J. Slonka: Assembly Programming and Computer Architecture for Software Engineers (ISBN 978-1-943153-32-9) .

The original English language work has been published by:

Prospect Press

Burlington, Vermont, USA

Copyright © 2018. All rights reserved.

No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or any information storage and retrieval system, without permission, in writing, from the publisher.

This Chinese simplified edition is arranged with Prospect Press through John Scott & Company.

Chinese simplified language edition published by China Machine Press, Copyright © 2019.

本书中文简体字版由 Prospect Press 通过 John Scott & Company 授权机械工业出版社在全球独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

本书通过大量实例，循序渐进地讲解了汇编语言的各种应用方式，并直观地演示了怎样把汇编语言同高级语言结合，以制作出充分发挥硬件特性的程序。书中以 x86 与 x86_64 这两种主流架构为重点，兼顾 AT&T 及 Intel 语法，并适用于 GAS、NASM 与 MASM 这三种常见的汇编器，以及 Linux、macOS 与 Windows 这三种常用的操作系统，这使得身处各种开发环境中的软件工程师都能在书里找到可以直接运用的解决方案，并了解如何将其移植到其他环境。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码 100037）

| | |
|---|---|
| 责任编辑：卢　璐 | 责任校对：殷　虹 |
| 印　　刷：北京瑞德印刷有限公司 | 版　　次：2019 年 4 月第 1 版第 1 次印刷 |
| 开　　本：185mm×260mm　1/16 | 印　　张：19.5 |
| 书　　号：ISBN 978-7-111-61516-3 | 定　　价：79.00 元 |

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88379833
购书热线：（010）68326294
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

**版权所有·侵权必究**
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson、McGraw-Hill、Elsevier、MIT、John Wiley & Sons、Cengage 等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出 Andrew S. Tanenbaum、Bjarne Stroustrup、Brian W. Kernighan、Dennis Ritchie、Jim Gray、Afred V. Aho、John E. Hopcroft、Jeffrey D. Ullman、Abraham Silberschatz、William Stallings、Donald E. Knuth、John L. Hennessy、Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近 500 个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com
电子邮件：hzjsj@hzbook.com
联系电话：（010）88379604
联系地址：北京市西城区百万庄南街 1 号
邮政编码：100037




华章科技图书出版中心

# 译 者 序

Assembly Programming and Computer Architecture for Software Engineers

这是一本从汇编语言入手讲解计算机系统结构的书。谈起编程语言，你首先可能会想到一些常见的高级语言，尤其是功能比较多、写起来又很方便的高级语言。它们提供的抽象机制使开发者不用担心操作系统乃至底层硬件的细节，能够专注于应用层面的逻辑。

可是，如果过度依赖这种抽象机制，就会对程序的工作原理尤其是它在操作系统及底层硬件中的运作方式渐渐陌生。然而一些重要的开发环节，例如调试、测试、优化等，却正需要开发者了解这些原理及运作方式，这样才能节省时间并提高效率。

从应用的角度讲，学习汇编语言，或许并不是为了把整个项目都用汇编指令去写，而是为了能够在某些关键的地方采用插入内联式汇编代码等手法来利用处理器的特性，从而更加灵活地实现软件的需求。此外，调试及测试程序时，也可以通过高级语言与汇编指令之间的对应关系来把握程序的运行状况，从而更为迅速地找到错误，或发现那些可以继续优化的地方。了解汇编语言，还可以帮助你从编译后的机器码入手，研究各种编译选项对程序的运行效率及效果所产生的影响。

本书通过大量实例，循序渐进地讲解了汇编语言的各种应用方式，并直观地演示了怎样把汇编语言同高级语言结合，以制作出充分发挥硬件特性的程序。书中内容以 x86 与 x86_64 这两种主流架构为重点，兼顾 AT&T 及 Intel 语法，并适用于 GAS、NASM 与 MASM 这三种常见的汇编器，以及 Linux、macOS 与 Windows 这三种常用的操作系统，这使得身处各种开发环境中的软件工程师都能在书里找到可以直接运用的解决方案，并了解如何将其移植到其他环境。

翻译本书的过程中，得到了机械工业出版社华章公司诸位编辑与工作人员的帮助，在此深表感谢。

由于译者水平有限，错误与疏漏之处，请大家发邮件至 eastarstormlee@gmail.com，或访问 github.com/jeffreybaoshenlee/apcase-errata/issues 留言，给我以批评和指教。

爱飞翔

2018 年 8 月 9 日

## 本书的目标

这是一本讲解汇编语言与计算机体系结构的教材。笔者从实用的角度出发讲解相关的原理与技术。首先介绍计算机编程语言和体系结构的基础知识，然后详细讨论汇编语言，并以此为手段，帮助读者更好地理解计算机的体系结构以及汇编语言在软件开发工作中的用途。

现有的计算机体系结构教程有这样几个缺点：（1）书里所讲的体系结构不是业界的主流；（2）书的写法更适合学习计算机与电气工程专业的人阅读，而不适合学习计算机科学专业的人及软件工程师阅读；（3）书里只讲与某个具体的操作系统有关的开发环境；（4）定价过高；（5）缺乏实用的内容。笔者想为大家提供一本更好的教材。

## 本书所面向的读者

本书主要面向计算机科学及软件工程专业的大学生。阅读本书之前，应该先学习计算机入门课程并打下良好的编程基础。你至少要掌握数据结构，而且最好是从 C/C++ 语言的角度学习过数据结构。已经工作的人也能从这本书里发现很多有用的话题，你可以据此来自学，以提高编写底层代码与高级代码的水平。

## 本书的编写过程及学习方法

从事学术工作的过程中，笔者经常冒出写书的念头，其中很多想法都源自我们两个人所讲的课程，也就是计算机体系结构。几十年来，这门课一直是计算机科学与软件工程专业的重要内容。此外，我们也开设了一些更偏向于实用技术的课程。

读博士的时候，我们曾经随口说要合写一本书，但一直没动笔。现在，总算找到了可以写这样一本书的机会。写书要花很多工夫，而且需要相互协作。笔者教了多年的计算机体系结构课程，并且一直在与 Prospect 出版社的合作伙伴 Beth Golub 讨论相关的想法，最后，大家形成了一套颇为独特的理念，并打算以此为基础，写一本面向教师、学生与 IT 从业者的计算机体系结构教程，同时克服前面所说的缺点。也就是说，我们要写的这本书应该基于主流的体系结构，面向学习计算机科学专业的人及软件工程师，适合多种开发环境，定价要合理，而且内容也要实用。

确定了这个理念之后，我们开始用大量的时间撰写本书，这其中有许多时间花在了程序代码上。用汇编语言写代码是相当曲折的，但正因为如此也总能吸引我们去挑战。你会在学习过程中慢慢地体会到这种乐趣。想要把汇编语言与计算机体系结构这样一个复杂的话题写得简洁而清晰是很难的，而且其中哪些方面应该涵盖、哪些方面应该舍弃也不太容易决定，此外还要考虑如何安排讲解的顺序。这些都是撰写计算机书籍时经常遇到的问题。

对于许多学习计算机的人来说，通过汇编语言切入计算机体系结构是个特别好的方法，而且在用这种方法学习体系结构的过程中，你还能掌握实用的编程技巧。所以说，按照这种学习方法写出来的书，很适合用作相关课程的教材，而且也颇受学校、教师及学生的欢迎。

我们认为汇编语言与计算机体系结构这门课讲起来很有意思，想要通过本书把这些有趣的知识分享给大家，希望你也能和我们一样喜欢这些内容。

## 内容编排及各章目标

第 1 章与第 2 章，讲解计算机编程语言与计算机体系结构的基础知识。

第 3 章至第 5 章，介绍 x86 与 x86_64 的汇编语法以及各种指令。

第 6 章至第 8 章，讲解函数、结构体以及浮点数运算等更为复杂的话题。

第 9 章与第 10 章，讲解怎样把汇编与高级语言及系统软件结合起来使用，并讲解与计算机及体系结构有关的其他高级话题。

第 11 章讲解 x86 之外的其他架构。

第 12 章介绍硬件及其元件的基本原理。

**各章目标**

第 1 章：描述计算机语言的翻译流程

第 2 章：了解计算机与处理器的各个组成部分

第 3 章：知道各种汇编器在语法上的区别

第 4 章：执行基本的算术运算

第 5 章：控制程序的流程

第 6 章：按照约定来调用函数

第 7 章：使用字符串与结构体

第 8 章：执行浮点运算

第 9 章：整合底层代码与高级代码

第 10 章：发起系统调用

第 11 章：对比各种计算机架构

第 12 章：构建简单的电路与设备

**补充材料**

本书有 4 组补充材料，分别位于第 1 章与第 2 章之后，以及第 6 章、第 8 章、第 10 章之后，其中包含与对应章有关的程序及内容。

**附录**

本书共有 9 个附录。在附录的前面有一篇总的简介，用来描述这些附录的内容。这些附录分别提供了与下列 9 个方面有关的实用信息：汇编语言的语法、开发环境的配置、反汇编、调试、汇编代码与 C++ 代码的链接、调用函数时所需遵循的约定、CPUID 的用法、十进制与 ASCII 算术、内部函数。

## 学习资源

- 本书网站：http://www.prospectpressvt.com/titles/hall-assembly-programming/
- 本书源代码：https://github.com/brianrhall/Assembly
- 教师资源：http://prospectpressvt.com/titles/hall-assembly-programming/instructor-resources/
- 学生资源：http://prospectpressvt.com/titles/hall-assembly-programming/student-resources/
- 作者 Brian 的网站：http://www.brianrhall.net

- 作者 Kevin 的网站：http://www.kevinslonka.com
- 与某些具体内容有关的资源，请参阅各附录以及位于所有附录之前的简介。此外，每章开头的网络资源也包含相关的链接。

## 致谢

这本书能够问世得益于很多人的帮助，我们对此深表谢意。感谢诸位老师把我们带入这个领域。感谢 Beth Golub 给我们提供机会，令我们能够与 Prospect 出版社合作出版这本书，并感谢她在我们写书的过程中给予指导。感谢以下诸位审阅者抽出时间给我们提供宝贵的意见：

印第安纳大学东南分校的 John Doyle

阿肯色州立大学的 Hai Jiang

安大略理工大学的 Saad Khattak

威斯康星大学帕克塞德分校的 Susan Lincke

洛约拉马利蒙特大学的 Ray Toal

感谢诸位同事与学生在过去两年里为我们写作本书提供了很多便利。最后，尤其感谢我们的家人愿意拿出本来应该共处的时间让我们写书。Brian 要感谢 Narine 与 Aven，Kevin 要感谢未婚妻 Kelli，这本书能够写成，离不开你们的付出。

# 目　录

Assembly Programming and Computer Architecture for Software Engineers


出版者的话

译者序

前言

**第 1 章　编程语言及数据的基础知识** …… 1

1.1　开篇语 …… 1

1.2　简介 …… 3

1.3　计算机编程语言 …… 3

1.3.1　语言之间的关系 …… 3

1.3.2　翻译流水线 …… 3

1.3.3　编程语言与相应文件及编程工具之间的关系 …… 6

1.3.4　为什么要学习汇编语言 …… 7

1.4　数据的表示 …… 8

1.4.1　计数系统 …… 8

1.4.2　怎样表示整数 …… 8

1.4.3　怎样表示无符号的整数 …… 9

1.4.4　怎样表示带符号的整数 …… 12

1.4.5　怎样保存字符 …… 14

1.5　布尔表达式 …… 16

1.6　3 位计算机示例 …… 17

1.7　小结 …… 18

1.8　关键术语 …… 18

1.9　代码回顾 …… 19

1.10　习题 …… 20

1.11　作业 …… 21

**第 2 章　处理器与计算机系统体系结构** …… 22

2.1　简介 …… 22

2.2　体系结构概述 …… 22

2.3　处理器 …… 26

2.3.1　缓存与寄存器 …… 27

2.3.2　64 位处理器 …… 31

2.3.3　指令的执行 …… 31

2.3.4　指令流水线 …… 32

2.4　输入与输出 …… 33

2.5　小结 …… 35

2.6　关键术语 …… 35

2.7　习题 …… 36

2.8　作业 …… 37

**第 1 章与第 2 章补充材料　与体系结构有关的更多细节** …… 38

**第 3 章　汇编语言及其语法的基础知识** …… 41

3.1　简介 …… 41

3.2　基本元素 …… 42

3.2.1　汇编代码的五大支柱 …… 42

3.2.2　字面量 …… 46

3.2.3　标签与注释 …… 48

3.3　定义数据 …… 49

3.4　写出能够正常运行的范例程序 …… 55

3.5　小结 …… 56

3.6　关键术语 …… 56

3.7　代码回顾 …… 57

3.8　习题 …… 57

3.9　作业 …… 58

**第 4 章　基本指令** …… 60

4.1　简介 …… 60

4.2　数据的移动与算术运算 …… 61

4.2.1　移动数据 …… 61

4.2.2　加法与减法 …… 62

4.2.3　乘法与除法 …… 64

4.2.4　移位 …… 69

4.2.5　处理负值 …… 71

4.3　数据的寻址与传输 …… 72

4.3.1 数据对齐 …… 72
4.3.2 数据寻址 …… 73
4.3.3 数组 …… 75
4.3.4 改变数据的大小及类型 …… 78
4.4 小结 …… 79
4.5 关键术语 …… 80
4.6 代码回顾 …… 80
4.7 习题 …… 81
4.8 作业 …… 81

**第 5 章 中级指令** …… 83

5.1 简介 …… 83
5.2 按位执行的布尔运算 …… 83
5.3 分支 …… 88
5.3.1 无条件跳转 …… 88
5.3.2 有条件跳转 …… 88
5.3.3 复合条件 …… 91
5.4 重复执行 …… 92
5.4.1 用 CX/ECX/RCX 计数器实现循环 …… 92
5.4.2 用开发者自定义的计数器实现循环 …… 94
5.5 小结 …… 97
5.6 关键术语 …… 97
5.7 代码回顾 …… 97
5.8 习题 …… 98
5.9 作业 …… 99

**第 6 章 函数** …… 100

6.1 简介 …… 100
6.2 栈内存入门 …… 100
6.3 x86 与 x86_64 的调用约定 …… 101
6.3.1 cdecl（32 位） …… 102
6.3.2 stdcall（32 位） …… 109
6.3.3 x86_64（64 位） …… 110
6.3.4 有用的细节 …… 114
6.4 实现 …… 114
6.5 小结 …… 118
6.6 关键术语 …… 118
6.7 重要的寄存器（32 位和 64 位） …… 118
6.8 代码回顾 …… 119
6.9 与平台有关的注意事项 …… 119
6.10 习题 …… 120
6.11 作业 …… 120

**第 6 章补充材料 程序 6-3** …… 122

**第 7 章 与字符串有关的指令及结构体** …… 123

7.1 简介 …… 123
7.2 辅助指令 …… 123
7.3 基本字符串指令 …… 125
7.3.1 MOVS …… 125
7.3.2 CMPS …… 126
7.3.3 SCAS …… 129
7.3.4 STOS …… 131
7.3.5 LODS …… 132
7.4 结构体 …… 133
7.5 小结 …… 135
7.6 关键术语 …… 135
7.7 代码回顾 …… 135
7.8 习题 …… 136
7.9 作业 …… 137

**第 8 章 浮点运算** …… 138

8.1 简介 …… 138
8.2 浮点数的表示方式 …… 139
8.2.1 IEEE 表示法 …… 139
8.2.2 特殊值 …… 141
8.2.3 次正规数 …… 141
8.2.4 舍入 …… 142
8.3 浮点数的实现 …… 143
8.3.1 x87 …… 143
8.3.2 MMX——题外话 …… 152
8.3.3 SSE …… 154
8.3.4 XOP、FMA3/4、F16C——分化 …… 159
8.3.5 AVX …… 160

8.4 小结 …… 161
8.5 关键术语 …… 162
8.6 重要的寄存器（32 位和 64 位） …… 162
8.7 代码回顾 …… 162
8.8 习题 …… 164
8.9 作业 …… 165

第 8 章补充材料 第 8 章中的程序 …… 167

第 9 章 内联汇编与宏 …… 177
9.1 简介 …… 177
9.2 内联汇编 …… 177
9.2.1 与编译器有关的细节 …… 178
9.2.2 内联式的汇编语句 …… 178
9.2.3 为各种汇编方言提供支持 …… 184
9.2.4 注意事项 …… 185
9.3 宏 …… 186
9.3.1 对比宏与函数 …… 186
9.3.2 定义并调用宏 …… 186
9.4 小结 …… 188
9.5 关键术语 …… 188
9.6 习题 …… 188
9.7 作业 …… 189

第 10 章 与处理器及体系结构有关的高级话题 …… 191
10.1 简介 …… 191
10.2 处理器与系统的机能 …… 192
10.2.1 系统寄存器 …… 192
10.2.2 处理器模式 …… 193
10.2.3 内存模型 …… 193
10.2.4 代码范例 …… 197
10.3 中断与系统调用 …… 198
10.3.1 软件中断 …… 198
10.3.2 硬件中断 …… 204
10.3.3 通过 INT 进行系统调用（旧方法） …… 204
10.3.4 通过 SYSENTER、SYSCALL 及程序库 /API 进行系统调用（新方法） …… 207
10.4 小结 …… 214
10.5 关键术语 …… 215
10.6 代码回顾 …… 216
10.7 习题 …… 216
10.8 作业 …… 217

第 10 章补充材料 第 10 章中的程序和资源 …… 218

第 11 章 其他架构 …… 223
11.1 简介 …… 223
11.2 CISC 与 RISC …… 224
11.3 更多架构 …… 225
11.3.1 ARM …… 225
11.3.2 AVR …… 228
11.3.3 RISC-V …… 231
11.3.4 System-z/Architecture …… 233
11.4 量子架构 …… 234
11.5 小结 …… 236
11.6 关键术语 …… 236
11.7 习题 …… 236
11.8 作业 …… 237

第 12 章 硬件与电子元件 …… 239
12.1 简介 …… 239
12.2 电学基础 …… 239
12.2.1 基本的物理量 …… 239
12.2.2 交流与直流 …… 241
12.3 电子元件 …… 242
12.3.1 供电设备 …… 242
12.3.2 电阻器 …… 244
12.3.3 二极管 …… 245
12.3.4 电容器 …… 245
12.3.5 晶体管 …… 245
12.4 集成电路 …… 246
12.5 常见的实现产品 …… 247
12.5.1 计算机处理器 …… 247

12.5.2 Raspberry Pi ····· 247
12.5.3 Apple iPhone 与 Apple Watch ····· 248
12.6 小结 ····· 248
12.7 关键术语 ····· 248
12.8 习题 ····· 249
12.9 作业 ····· 250

附录简介 ····· 251

附录 A 各种汇编语法之间的对应关系 ····· 253

附录 B 为汇编语言配置开发环境 ····· 256

附录 C 反汇编 ····· 261

附录 D 用 GDB 工具调试汇编程序 ····· 268

附录 E 链接汇编代码及 C++ 代码 ····· 273

附录 F 函数与栈 ····· 278

附录 G CPUID 指令的用法 ····· 282

附录 H ASCII 和十进制算术运算 ····· 289

附录 I 编译器内部函数 ····· 293

第 1 章

Assembly Programming and Computer Architecture for Software Engineers

# 编程语言及数据的基础知识

## 目标

- 区分各种计算机编程语言
- 了解汇编语言的用途
- 在各种数制之间转换
- 解决基本的算术问题
- 理解字符的保存方法
- 求布尔表达式的值
- 解释计算机操作的基本原理

## 网络资源

**维基资源**

- https://en.wikibooks.org/wiki/X86_Assembly（汇编语言概述）
- https://en.wikibooks.org/wiki/X86_Disassembly（汇编器与反汇编器概述）

**技术资源**

- http://www.unicode.org
- http://www.unicode.org/charts/PDF/U0000.pdf（ASCII 字符表）
- http://devimages.apple.com/llvm/videos/LLVM_Assembler_Infrastructure.mov（LLVM-MC 概述）

**视频资源**

- 视频 1.1：欢迎词（https://youtu.be/HBwgXY88hyc）
- 视频 1.2：整数的符号问题（https://youtu.be/vHijiZMbj9E）
- 视频 1.3：3 位计算机示例（https://youtu.be/gYczcmDywag）

## 1.1 开篇语

感谢你阅读这本书。这是一本讲述汇编语言和计算机体系结构（architecture）[⊖]的教程，笔者将从实用的角度讲解这些话题。本书讨论了许多原理，而且会告诉大家如何运用这些原理。前两章介绍计算机编程语言与计算机体系结构的基础，后续章节以汇编语言为手段，帮助大家更好地理解计算机的体系结构，以及如何在软件开发工作中利用汇编语言。这本书的独特之处在于它从多方面来讲解：

- 本书基于 x86 及 x86_64 这两种主流系统结构讲解，你学到的知识也能够用于其他的体系结构。

⊖ 在不引起混淆的场合，也可以叫作系统架构、架构或系统结构。——译者注

- 本书提供的范例代码很有特点，它同时适用于 GAS、MASM 及 NASM 这三种汇编器（assembler），这意味着你可以在 Mac、Windows 及 Linux 等操作系统上学习，而且，这些代码也符合 AT&T 及 Intel 的语法规则。
- 本书同时提供了适用于 32 位环境及 64 位环境的范例代码，这两种环境对开发者都很重要。
- 本书的附录提供了很多实用的信息，能够帮助你学习并使用与汇编语言及计算机体系结构有关的技术。
- 有一些章节后面带有补充材料，提供了深入学习这些话题所需的知识。
- 本书提供了各种维基资源、技术资源与视频资源，以帮助读者深入探索相关的话题。
- 本书采用提示、编程知识与学习指南这三种形式来提供注解，以帮助读者更好地学习。




**提示：**GNU Assembler（GAS）是一款基于 Linux 的汇编器，主要供 GNU 项目使用，它产生于 1987 年。GAS 是 GNU Binutils 包的一部分，用来对 Linux 内核以及其他一些软件进行汇编。Netwide Assembler（NASM）是一款基于 Linux 的开源汇编器 / 反汇编器，适用于 x86 与 x86_64，它产生于 1996 年，是由一个很小的团队开发的。Microsoft Macro Assembler（MASM）是微软操作系统的专属汇编器，随 Visual Studio 一起发布，它产生于 1981 年。

- 为了便于学生学习，笔者用教材的风格写这本书，并确保其内容实用。

本书各章总览如图 1-1 所示。

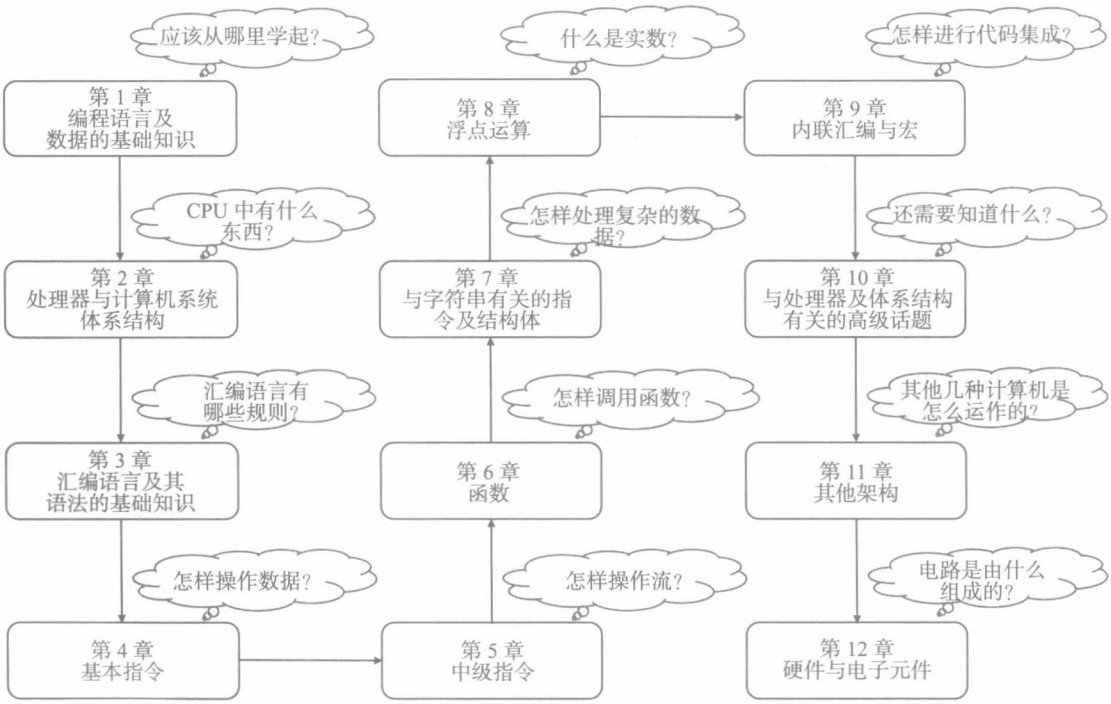



图 1-1　各章总览

## 1.2 简介

本章介绍计算机程序与计算机体系结构的基本概念。其中要谈到计算机语言之间的关系，以及汇编语言在整个语言体系中的地位。接下来讲解数据的表示方式、计数系统以及布尔表达式。然后会用 3 位的计算机系统来演示本章的概念。有了这一章所讲的基础知识，你就可以在第 2 章中深入学习具体的 x86/x86_64 架构了。你必须先看懂这一章，才能理解第 3 章开始所讲的汇编语言基础知识。

## 1.3 计算机编程语言

### 1.3.1 语言之间的关系

程序员整天要和计算机编程语言打交道，而要想学习汇编语言，首先就必须知道这种语言在整个计算机语言体系中处于什么位置。我们编程的时候一般用**高级语言**（high-level language）写代码，这些语言之所以叫作高级语言，是因为它们比较接近英语。本书使用 C++ 作为高级语言的代表。如范例 1-1 所示的 `if ... else` 控制结构能够体现出高级语言和英语的相似之处。

**范例 1-1 `if ... else` 控制结构**

| 含义相近的英语 | C++ |
| --- | --- |
| `if (this is true) {`<br>`  perform this series of actions`<br>`}`<br>`else {`<br>`  perform this series of actions`<br>`}` | `if ( x == y ) {`<br>`  cout << "X and Y are the same";`<br>`}`<br>`else {`<br>`  cout << "X and Y are different";`<br>`}` |

由于高级语言的抽象能力比较强，因此用这种语言写代码是很方便的，然而写出来的程序必须从高级的形式翻译成**机器级别的**（machine-level）形式，才能够交给计算机执行。当今的计算机处理器包含上亿个晶体管，它们能够在 1 和 0 之间切换，以允许或禁止电流通过。像这种根据两个状态而构建的系统，称为二进制系统（binary system）。必须把指令表示成二进制形式，才能将其交给处理器执行，那么，究竟应该怎样完成从高级语言到汇编语言再到机器级指令的转换工作呢？

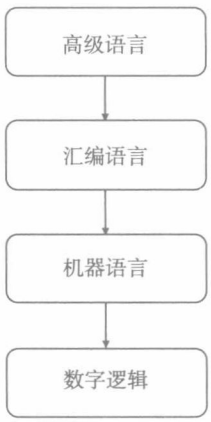


图 1-2 翻译流水线

### 1.3.2 翻译流水线

现在要来演示从高级到低级的处理过程。为了讲解得直观一些，大家可以把该过程想象成一条**翻译流水线**（translation pipeline），如图 1-2 所示。

**提示：**一种较为抽象的讲解方式是用虚拟机（virtual machine）这个概念来演示计算机体系结构中的各个层面，很多经典的范例都用到了它，你在读别的教科书或是听人讲解多层设计时，或许也会遇到这个说法。虚拟机可以用硬件或软件的方式实现。

用高级语言编写的程序必须经过翻译才能在底层运行。翻译有两种办法：一种是**解释**（interpreting），就是在执行程序的过程中翻译一行运行一行；另一种是**编译**（compiling），就是先把所有代码全都翻译完，然后再执行。Python 是解释型的高级编程语言，C++ 是编译型的高级编程语言。

高级语言里的一条语句（例如 `x - y + 2`）翻译成机器语言后通常会变成多条语句，也就是说，高级语言中的一条语句会翻译成好几条汇编指令或机器指令。

汇编语言是翻译流水线里的一个中间环节。用高级语言写的程序先翻译成汇编代码，然后再根据处理器的指令集架构翻译成机器语言。把汇编代码翻译成机器语言的过程通常称为编码（encoding），反向的过程，也就是把机器码翻译成汇编语言的过程，通常称为解码（decoding）。

汇编语言的语句与机器语言之间是一一对应的，这意味着，每一条汇编语言指令都对应于一条机器语言指令。后者通常用十六进制等中介的数值形式来表示，然后翻译为二进制，以便在物理硬件上实现，这就是数字逻辑（digital logic）。

**编程知识：**有一些编译器，例如 Microsoft 的 C 语言编译器，能够把用高级语言写成的代码直接翻译为机器语言，中间是否生成汇编码可以通过选项来控制。另一些编译器，例如 GCC，则要先翻译成汇编码，然后再翻译为机器代码。

**指令集架构**（Instruction Set Architecture，ISA）是计算机体系结构中与编程有关的方面。它指出了处理器所具备的指令、寄存器、内存架构、数据类型以及其他一些属性，以供程序员使用。你可以把 ISA 理解成计算机所说的语言，它能够促进软件与硬件之间的通信，见图 1-3。

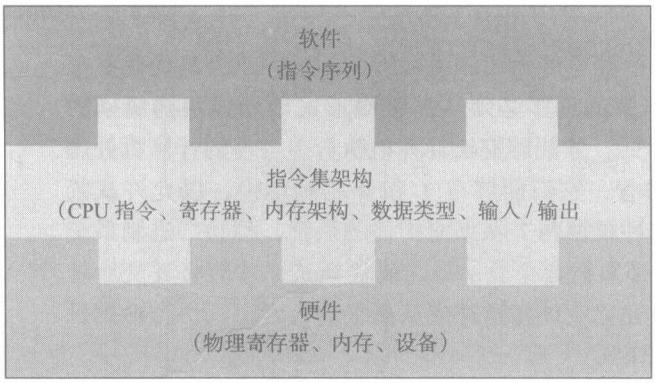



图 1-3　指令集架构

指令集架构分为复杂与精简两种。**复杂指令集**（Complex Instruction Set Computing，CISC）架构中的指令，其长度（也就是表示该指令所需的字节数）不固定。之所以说它复杂，是因为一条指令有可能要完成多项任务，比方说，既要访问某个内存地址，又要执行算术运算。与之相对的 ISA 设计称为**精简指令集**（Reduced Instruction Set Computing，RISC），其中的所有指令都一样长，而且只执行一项任务（例如只访问某个内存地址，不做其他的事情）。

x86 与 x86_64 都是 CISC 架构，而其他一些 ISA 设计方案则多为 RISC 架构。第 4、5 两章会详细讨论 x86 指令，那时你就会明白它为什么叫复杂指令集。此外，第 6、11 章会提

供一些反汇编的例子，这些例子能够显示出指令的长度不固定以及指令所执行的任务较为复杂这两项特点。本章后面的编程知识会提到可重定位的机器语言。

**编程知识：**反汇编是由反汇编器对包含机器语言的目标文件进行解码之后所输出的汇编代码，这实际上相当于把机器语言的二进制序列解码成汇编指令。大多数汇编器、编译器、调试器以及开发环境都提供反汇编功能，例如 NASM（ndisasm）、GDB、LLVM、Xcode 以及 Visual Studio。还有一些独立的反汇编器可供使用，像是 Capstone、IDA、`objdump` 与 `otool`。如何对目标文件做反汇编，请参阅本书附录 C。

| 32 位 x86 指令集架构的常用名称 | 64 位 x86 指令集架构的常用名称 |
| --- | --- |
| x86 | x64 |
| IA-32 | x86_64 |
| i386 | Intel：IA-32e（原名）、Intel 64（当前名称） |
| | AMD：AMD64 |

**提示：**32 位版的 x86 指令集通常称为 IA-32，此外也叫作 i386。64 位版的 x86 指令集通常称为 x64 或 x86_64，不过制造商也有可能使用其他一些叫法。请注意，64 位版的 x86 与 64 位的 Intel Architecture（简称 IA-64）是两种完全不同的架构，后者是 Itanium（安腾）系列处理器所用的指令集，它与 IA-32 及 x64 均不兼容。

表 1-1 举例说明了翻译流水线。先假设有一个名为 `sum` 的整数，然后用高级语言 C++ 编写一行代码，也就是 `sum = 5;`。接下来，用 GNU Assembler（GAS）将其翻译成遵循 AT&T 语法规则的汇编代码，进而根据 IA-32 指令集架构，把它翻译成机器语言，令其可以在 Intel 32 位处理器上执行。最后，以二进制的形式对十六进制形式的机器语言指令做物理实现（或者说数字实现）。

表 1-1　翻译流水线

| 级　别 | 语　言 | 代　码 |
| --- | --- | --- |
| 高级语言 | C++ | `sum = 5;` |
| 汇编语言 | GNU Assembler、AT&T 语法 | `movl $0x5, -0x8(%ebp)` |
| 机器语言 | IA-32 | `C745F805000000` |
| 数字逻辑 | 二进制（物理实现） | `1100 0111 0100 0101 1111 1000 0000 0101 0000 0000 0000 0000 0000 0000 0000` |

接下来再看三行代码，以巩固这种印象。范例 1-2 主要是想让大家明白汇编语言需要编码成对应的机器语言，而至于具体怎样对应则不是此处的重点。第 4、5 两章会详细讲解这些指令。

假设有一条 32 位的汇编指令：`MOV eax, 5`。如果用高级语言来表达该指令的意思，那么写出来的代码可能类似于 `variable = 5;`，然而从汇编语言的角度来看，这条指令强调的是：把立即值 5 移动（或者说复制）到 *eax* 寄存器里。MOV 操作码与 eax 寄存器都是 IA-32 指令集架构的一部分。像这种把某个立即值移动到 eax 寄存器中的操作方式，是与特定的机器语言相对应的，具体来说，就是会编码成十六进制形式的 `B8`，而作为操作数的立即值也会加以编码，并出现在 `B8` 的后面。

范例 1-2　把汇编指令编码成 IA-32 架构的机器语言

| 汇编语言 | 机器语言 |
| --- | --- |
| MOV eax, 5 | B8 05 00 00 00 |
| MOV eax, 32555 | B8 2B 7F 00 00 |
| MOV ebx, 5　　;a different register | BB 05 00 00 00 |

由于机器语言要用数字的形式来表示，因此，我们应该理解各种计数系统（number system，或者说数制），尤其是十六进制与二进制。机器语言是针对特定的处理器而言的，不过，同一个系列的处理器能够解读同一种机器语言，因此，针对 x86 处理器所写的代码可以用在该系列的任何一款处理器上，x86_64 也是如此。

**编程知识：**汇编代码无法在不同系列的处理器之间移植。比方说，针对 x86 系列的处理器所写的代码，无法在 SUN Sparc、IBM 370 以及 ARM Cortex 等系列的处理器上执行。

x86/x86_64 处理器家族，既包括 Intel 的 Pentium、Core-Duo 及 Core i7，也包括 AMD 的 Athlon、Phenom 及 Opteron。Intel 与 AMD 的处理器设计方案都实现了 x86 指令集，但实现该指令集所用的具体技术（也就是**微架构**，microarchitecture）却有所不同。

### 1.3.3　编程语言与相应文件及编程工具之间的关系

把翻译流水线中的各个环节与相应的文件及编程工具放在一起考虑，也有助于我们理解这些内容。图 1-4 描述了与各个级别的编程语言有关的文件，以及将代码从一种语言翻译为另一种语言时所用到的计算机工具。

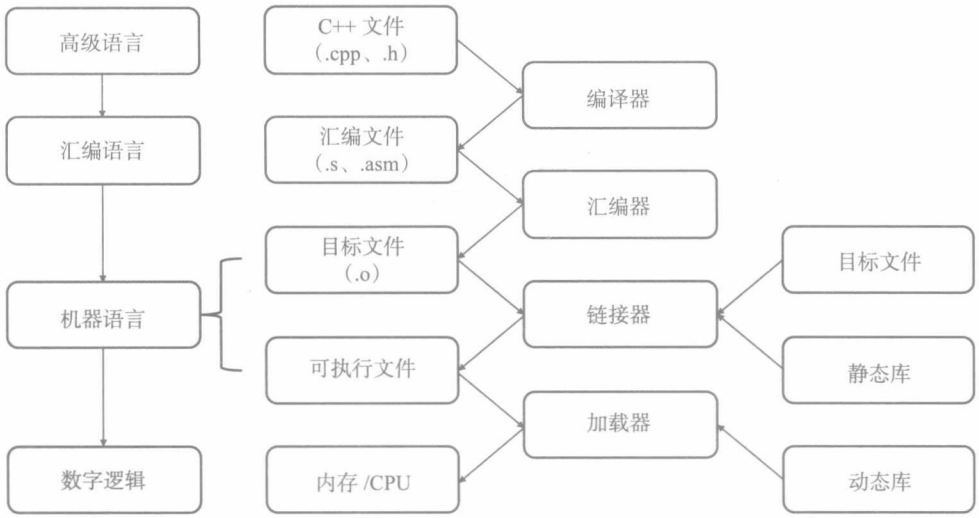



图 1-4　各个级别的编程语言与相应文件及编程工具之间的关系

**编译器**（compiler）根据处理器的指令集，把用高级语言写成的源代码转化成汇编语言这种中介形式。然后，**汇编器**（assembler）把这些汇编代码编码成**目标代码**（object code），这是一种可重定位的机器语言，其格式与具体的操作系统（Operating System，OS）平台有

关。表 1-2 列出了各种操作系统中常见的目标文件格式。**链接器**（linker）把多个目标文件及静态库（static library）合并起来，以形成一个可执行文件。最后，**加载器**（loader）把链接器在可执行文件中所生成的指令与动态库（dynamic library）中的指令相结合，将这些机器码载入内存，以供 CPU（Central Processing Unit，中央处理器）执行。

**编程知识：**什么叫作可重定位的机器语言？所谓可重定位，是指在将汇编代码翻译成目标代码时，采用一种通用的形式来表示这些代码，把第一条指令的开始位置记为 0x0h 这个地址，以后指令的位置都用它与前面指令相隔的字节数（或者说偏移量，offset）表示。这样的话，在将程序实际载入内存时，只需要把第一条指令放在某个真实的地址上就可以了，而后续的各条指令则可以根据其偏移量安排在适当的位置上。

| 可重定位的机器语言 | | | 程序运行时的机器语言 | | |
|---|---|---|---|---|---|
| main: | | | main: | | |
| 00000000 | b80a000000 | movl $0xa, %eax | 0x1fac <+0>: | movl | $0xa, %eax |
| 00000005 | 48 | decl %eax | 0x1fb1 <+5>: | decl | %eax |
| 00000006 | bb05000000 | movl $0x5, %ebx | 0x1fb2 <+6>: | movl | $0x5, %ebx |
| 0000000b | c3 | retl | 0x1fb7 <+11>: | retl | |

表 1-2 各操作系统常用的目标文件格式

| Windows | macOS | Linux |
|---|---|---|
| COFF、PE、Win32、Win64 | Mach-O | ELF、a.out（此格式已过时） |

### 1.3.4 为什么要学习汇编语言

有一个很关键的问题需要思考，这就是“为什么要学习汇编语言”。既然能用高级语言编写代码，那为什么还要学习底层的细节呢？这可以从多个方面回答：

- 学习汇编语言可以更好地理解计算机的操作。程序员可以通过汇编这种手段来了解计算机硬件、CPU 组件以及内存布局等细节，并知道计算机体系结构中的这些要素是怎样交互的。
- 编写汇编代码时，程序员可以频繁使用**调试器**来排查代码中的错误。它提供了汇编、反汇编等功能，还可以查看寄存器与内存信息，以便在底层单步地执行程序。理解汇编语言与计算机体系结构，能够帮助开发者更好地找到并修复代码中的 bug。
- 了解汇编语言及计算机操作，能够帮助开发者对高层及底层的实现问题做出更加明智的决策，从而可以充分利用硬件的特征来优化程序的效率。
- 懂得汇编语言可以令你在完成位操作（bit manipulation）及函数调用（function call，也称为 procedure call，过程调用）等任务时，省去很多抽象环节。
- 必须非常熟悉汇编才能够进入某些软件开发领域，例如嵌入式设备的编程、设备驱动程序的编程，以及操作系统编程等。

总之，学习汇编语言，可以令你更加全面地理解计算机的操作原理及代码的运行机制，从而能够更好地利用这些知识。此外，对于软件工程师来说，没有什么事情能比直接同处理器对话更令人惬意的了。

**学习指南：**参考本章最后的**冒泡排序**（bubble sort）范例，看看用汇编语言写出来的程序，其文件大小与内存占用量同 C++ 版本的程序有什么区别。

## 1.4　数据的表示

### 1.4.1　计数系统

多数计算机在物理层面都是以二进制的方式来运作的。例如，存储单元（memory cell）的状态只有两种，要么已充电，要么未充电；磁盘中的某个位置的状态也只有两种，要么已磁化，要么未磁化；CPU 中的晶体管同样如此，要么允许电流通过，要么禁止电流通过。由此可见，数据在底层可以用二进制来表示，这种表示方式只使用 1 与 0 两种数位。但是，该形式对于人类来说却比较麻烦，而且理解起来也很慢。因此，我们还会用其他一些计数系统（或者说数制）来简记二进制数。

**学习指南：**实际上，在执行机器语言所写的代码时，计算机只能采用二进制来运作。那么，为什么必须用二进制呢？

答案是：用二进制来表示数据并在双稳环境下运行程序，是比较可靠的。

具有下列物理特性的硬件元件，能够以二进制的方式运作：

- 元件有两种稳定的能量状态（energy state，能态），可以分别用 0 与 1 来表示，例如，全开 / 全关、完全充电 / 完全放电、带正电 / 带负电、已磁化 / 未磁化、反光 / 不反光。
- 两种状态可以为能量壁垒（energy barrier，能垒）所区隔（也就是说，不会同时处于两种状态）。
- 可以感知元件的状态。
- 可以在两种状态之间切换。

假如计算机以十进制或十六进制来计数，那么要保存数值就比较复杂了，因为硬件必须要能够表示 10 种或 16 种不同的状态才行。反之，如果用二进制来保存，那么只需要把相应的物理性质与两种状态对应起来就可以了，对于大多数计算机来说，这些物理性质指的就是元件的电气性质。

要想读懂并撰写底层程序代码，必须熟悉各种计数系统与计数格式。最常见的几种格式是：二进制、八进制、十进制、十六进制。每种进制都有一个底，它决定了每个位置上可以使用多少种符号来计数。表 1-3 详细列出了这四种常用的数制，其中，十进制与十六进制是汇编代码中最常见到的形式。

表 1-3　常用的数制

| 数　制 | 底 | 每个数位上可以使用的计数符号 |
| --- | --- | --- |
| 二进制 | 2 | 0 1 |
| 八进制 | 8 | 0 1 2 3 4 5 6 7 |
| 十进制 | 10 | 0 1 2 3 4 5 6 7 8 9 |
| 十六进制 | 16 | 0 1 2 3 4 5 6 7 8 9 A B C D E F |

### 1.4.2　怎样表示整数

从编程的角度来看，采用二进制意味着每个位置上只能填写 0 或 1，也可以说假（false）或真（true）。二进制数中的每一位叫作一个 bit（二进制位，也称为比特或位元），多个 bit 可以合成一个 byte（字节，也称为位元组），以存放整数（integer，也叫作 whole number）

等数值。表 1-4 列出了每一种表示方式所占据的二进制位数、字节数，以及特定的称谓，还列出了在**无符号**（unsigned）及**带符号**（signed）两种情况下的取值范围。无符号意味着只能用来表示 0 及正整数，而带符号则意味着还可以用来表示负整数。

表 1-4　整数的表示方式

| 位　数 | 字节数 | 名　称 | 无符号整数的取值范围 | 带符号整数的取值范围 |
|---|---|---|---|---|
| 8 位 | 1 字节 | byte（字节） | $0 \sim 2^8-1$ | $-2^7 \sim 2^7-1$ |
| 16 位 | 2 字节 | word（字） | $0 \sim 2^{16}-1$ | $-2^{15} \sim 2^{15}-1$ |
| 32 位 | 4 字节 | doubleword（双字，双倍字） | $0 \sim 2^{32}-1$ | $-2^{31} \sim 2^{31}-1$ |
| 64 位 | 8 字节 | quadword（四字，四倍字） | $0 \sim 2^{64}-1$ | $-2^{63} \sim 2^{63}-1$ |
| 80 位 | 10 字节 | tenbyte（十字节） | $0 \sim 2^{80}-1$ | $-2^{79} \sim 2^{79}-1$ |
| 128 位 | 16 字节 | double quadword（双四字，双四倍字） | $0 \sim 2^{128}-1$ | $-2^{127} \sim 2^{127}-1$ |

多个二进制位合起来表示某个数时，最左侧的位叫作最高有效位（Most Significant Bit，MSB），最右侧的位叫作最低有效位（Least Significant Bit，LSB）。以下面的 8 位（或者说 1 字节）无符号二进制数为例，如果 LSB（也就是 0 号位）由 0 变成 1，那么数值只会比原来大 1（也就是 2 的 0 次方），但如果 MSB（也就是 7 号位）由 0 变成 1，那么数值则要比原来大 128（也就是 2 的 7 次方）。




## 1.4.3　怎样表示无符号的整数

有两个与数制有关的技巧是应该掌握的，一个是在各种进制的数之间转换，另一个是完成加法这样的基本运算。表 1-5 列出了 0 至 15 之间的数所对应的十进制、二进制及十六进制表示方式。

表 1-5　无符号整数在三种数制之间的对应关系

| 十进制 | 十六进制 | 二进制 | 十进制 | 十六进制 | 二进制 |
|---|---|---|---|---|---|
| 0 | 0 | 0000 | 8 | 8 | 1000 |
| 1 | 1 | 0001 | 9 | 9 | 1001 |
| 2 | 2 | 0010 | 10 | A | 1010 |
| 3 | 3 | 0011 | 11 | B | 1011 |
| 4 | 4 | 0100 | 12 | C | 1100 |
| 5 | 5 | 0101 | 13 | D | 1101 |
| 6 | 6 | 0110 | 14 | E | 1110 |
| 7 | 7 | 0111 | 15 | F | 1111 |

下面举 9 个例子来演示怎样在各种进制的无符号数之间转换以及如何做加法。

1. 二进制 ↔ 十进制：用量表来转换，表中列有从 $2^0$ 至 $2^n$ 的 $n+1$ 个级别，其中 $n$ = 二进制位数 − 1

2. 二进制 → 十进制：用按位展开式来转换

3. 十进制 → 二进制：用除法来转换

4. 计算保存十进制值所需的二进制位数

5. 二进制数的加法

6. 十六进制 ↔ 二进制 ↔ 十进制
7. 十六进制 → 十进制：用按位展开式来转换
8. 十进制 → 十六进制：用除法来转换
9. 十六进制数的加法

**提示：**十进制数可以通过下列三种方式转换成以其他数为底的进制：

- base$^n$ 量表（参见范例 1-3 及范例 1-8）。
- 按位展开式：十进制值 = $(D_{n-1} \times B_{n-1}) + (D_{n-2} \times B_{n-2}) + \cdots + (D_1 \times B_1) + (D_0 \times B_0)$，其中 $B$ 是底，$D$ 是各数位的具体取值，$n$ 表示 $B$ 进制数共有多少位（参见范例 1-4 及范例 1-9）。
- 除法（参见范例 1-5 及范例 1-10）。

范例 1-3 在 8 位二进制数与十进制数之间转换。这个例子采用的具体数值是二进制的 00010111，也就是十进制的 23（$00010111_2 = 23_{10}$）。把二进制数转换成十进数时，先找出值为 1 的二进制位，然后在表中查出对应的十进制值，最后将其相加，这样就得到了与原数相等的十进制数。

**范例 1-3　二进制 ↔ 十进制（$2^n$ 量表）**

| | MSB | | | | | | | LSB |
|---|---|---|---|---|---|---|---|---|
| 二进制位 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| 2 的幂 | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| 十进制值 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 有待相加的值 | | | | 16+ | | 4+ | 2+ | 1 |
| 相加所得的十进制值 | | | | | | | | 23 |

范例 1-4 演示了怎样用按位展开式将二进制数转换成十进制数。这个例子采用的具体数值是二进制的 00010111，也就是十进制的 23。

**范例 1-4　二进制 → 十进制（按位展开式）**

十进制值 = $(D_{(n-1)} \times 2^{(n-1)}) + (D_{(n-2)} \times 2^{(n-2)}) + \ldots + (D_1 \times 2^1) + (D_0 \times 2^0)$
十进制值 = $(D_7 \times 2^7) + (D_6 \times 2^6) + (D_5 \times 2^5) + (D_4 \times 2^4) + (D_3 \times 2^3) + (D_2 \times 2^2) + (D_1 \times 2^1) + (D_0 \times 2^0)$
十进制值 = $(0 \times 2^7) + (0 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$
十进制值 = $(1 \times 2^4) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$
十进制值 = (16) + (4) + (2) + (1)
十进制值 = 23

范例 1-5 演示怎样用除法把十进制数转化成二进制数。本例采用的十进制数是 42，相当于二进制的 101010。首先，把十进制数 42 与二进制的底（也就是 2）相除，商是 21，余数是 0。把余数 0 写在除法的右侧，在最终的转换结果中，它就是最低有效位。接下来，反复做除法，也就是用前一次除法所得的商除以 2，并将余数记在相应除法的右侧，直到所得的商小于 2 为止。此时的商只可能是 0 或 1，该数在最终的转换结果中是最高有效位。现在，把最后除法所得的商写在转换结果的最左边，然后将历次除法所得的余数从下往上依次写在商的后面。

范例 1-6 演示了怎样计算保存某个十进制值所需的二进制位数。比方说，要保存 42 这个十进制值，就需要用 6 个二进制位。

**范例 1-5 十进制 → 二进制（除法）**

| 除法 | 商 | 余数 | |
|---|---|---|---|
| **42** / 2 | 21 | **0** | LSB |
| 21 / 2 | 10 | **1** | |
| 10 / 2 | 5 | **0** | |
| 5 / 2 | 2 | **1** | |
| 2 / 2 | **1** | **0** | ↑ |

→
MSB　　二进制值 = 101010

**范例 1-6 计算保存十进制值所需的二进制位数**

二进制位数 = floor($\log_2 n$) + 1
二进制位数 = floor($\log_2 42$) + 1
二进制位数 = floor(5.392317) + 1
二进制位数 = 6

**学习指南：** 你可能想用二进制位数 = ceiling($\log_2 n$) 来计算保存十进制值所需的二进制位数，这对于大多数情况（例如范例 1-6 中的情况）来说是成立的，但如果要计算的十进制值恰好是 2 的幂，比方说 32，那么就会得出错误的结果。ceiling($\log_2 32$) 的结果是 5，而实际上必须用 6 个二进制位才能将它表示出来，因为 $32_{10} = 100000_2$。

范例 1-7 演示了二进制数的加法。规则很简单：

| 0 + 0 = 0 | 0 + 1 或 1 + 0 = 1 | 1 + 1 = 0, 向高位进 1 | 1 + 1 + 1 = 1, 向高位进 1 |
|---|---|---|---|

**范例 1-7 二进制数的加法**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 进位 | | | | | 1 | 1 | | | |
| | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | $23_{10}$ |
| + | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | $70_{10}$ |
| | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | $93_{10}$ |

范例 1-8 演示了怎样在十六进制、二进制与十进制之间转换。十六进制数转二进制数所依据的规律是：前者中的每个数位是由后者中的四个数位组成的。十六进制数转十进制数，可以根据以 16 为底的 $16^n$ 量表来做（这就像范例 1-3 所演示的那样：把二进制数转换成十进制数，可以根据以 2 为底的 $2^n$ 量表来做）。这个例子所用的具体数值是：$0001101010110111_2 = 6839_{10} = 1AB7_{16}$。

**范例 1-8 十六进制 ↔ 二进制 ↔ 十进制**

| | 与该十六进制位相等的十进制数 | 1 | | | | 10 | | | | 11 | | | | 7 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 在十六进制与二进制之间转换 | 十六进制 | 1 | | | | A | | | | B | | | | 7 | | | |
| | 二进制位 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| 在十六进制与十进制之间转换 | 16 的幂 | $16^3$ | | | | $16^2$ | | | | $16^1$ | | | | $16^0$ | | | |
| | 对应的十进制值 | 4096 | | | | 256 | | | | 16 | | | | 1 | | | |
| | 有待相加的值 | (1 x 4096) + | | | | (10 x 256) + | | | | (11 x 16) + | | | | (7 x 1) | | | |
| | 相加所得的十进制值 | $6839_{10}$ | | | | | | | | | | | | | | | |

范例 1-9 演示了怎样用按位展开式把十六进制数转换成十进制数。这个例子所用的具体数值是 $1AB7_{16} = 6839_{10}$。

**范例 1-9　十六进制 → 十进制（按位展开式）**

十进制值 = $(D_{(n-1)} \times 16^{(n-1)}) + (D_{(n-2)} \times 16^{(n-2)}) + \ldots + (D_1 \times 16) + (D_0 \times 16^0)$
十进制值 = $(D_3 \times 16^3) + (D_2 \times 16^2) + (D_1 \times 16^1) + (D_0 \times 16^0)$
十进制值 = $(1 \times 16^3) + (A \times 16^2) + (B \times 16^1) + (7 \times 16^0)$
十进制值 = $(1 \times 4096) + (10 \times 256) + (11 \times 16) + (7 \times 1)$
十进制值 = (4096) + (2560) + (176) + (7)
十进制值 = 6839

范例 1-10 演示了怎样用除法把十进制数转换成十六进制数。这个例子所用的具体数值是 $8039_{10} = 01F67_{16}$。计算过程与范例 1-5 类似，只不过每次都是除以 16，而不是除以 2。

**范例 1-10　十进制 → 十六进制**

| 除法 | 商 | 余数 | |
|---|---|---|---|
| **8 039** / 16 | 502 | **7** | LSB |
| 502 / 16 | 31 | **6** | |
| 31 / 16 | 1 | **15** | F |
| 1 / 16 | **0** | **1** | ↑ |

→
MSB　　十六进制值 = 01F67

范例 1-11 演示了怎样根据公式来对十六进制数做加法。即如果 $x_i + y_i \geqslant b$，那么 $z_i = (z_i \text{ MOD } b)$，并向高位进 1。这个公式也适用于其他进制（$b$ 表示数制的底）。

这个例子所用的具体数值是 $4B2_{16} + 6A4_{16} = B56_{16}$，算式最左侧那一列所写的 $x_i$，$y_i$ 与 $z_i$，用来指代这三个数值的各个数位。

**范例 1-11　十六进制数的加法**

| 进位 | | **1** | | | |
|---|---|---|---|---|---|
| $x_i$ | | 4 | **B** | 2 | $1202_{10}$ |
| $y_i$ | + | 6 | **A** | 4 | $1700_{10}$ |
| $z_i$ | | B | **5** | 6 | $2902_{10}$ |

$z_i = z_i \text{MOD} b$
$z_i = (B + A) \text{ MOD } 16$
$z_i = (11 + 10) \text{ MOD } 16$
$z_i = 21 \text{ MOD } 16$
$z_i = 5$

## 1.4.4　怎样表示带符号的整数

**提示：**之所以考虑带符号的整数这一问题，是因为便于做减法。从电气工程的角度来看，加法电路很直观，实现起来也很简单，而三种基本的算术运算也都可以通过加法实现，一种是加法本身，另一种是减法（与正数相减，可以化为与负数相加），还有一种是乘法（可以化为连加）。因此，很多 CPU 电路都是加法电路。

在计数系统中，整数的符号可以用多种办法表示，例如**原码**、**反码**及**补码**。这三种表示方式都用最高有效位表示正负，0代表正，1代表负。当前的大多数架构（例如x86与x86_64）均采用补码来表示带符号的整数。范例1-12演示了这样两个带符号的整数以及与之相应的十进制值。

**范例 1-12　带符号的二进制整数**

| | MSB | | | | | | | LSB | |
|---|---|---|---|---|---|---|---|---|---|
| 正数 | **0** | 0 | 0 | 1 | 0 | 1 | 1 | 1 | $+23_{10}$ |
| 负数 | **1** | 0 | 0 | 1 | 0 | 1 | 1 | 1 | $-105_{10}$ |

**学习指南：**请参阅视频1-2，以了解带符号整数的三种表示方式之间有何区别。

原码与反码都有个缺点，就是0会以两种形式出现，一种是正0，一种是负0，而且这两种表示方式用硬件实现起来较为困难。与之相比，补码不会出现两个0的问题，而且它是根据加法逆元原则设计的，也就是任何数与其加法逆元相加，结果均为0。

求补码要分两步，首先分别反转每个二进制位，然后给反转的总结果加1。范例1-13与范例1-14演示了怎样求出二进制正整数与二进制负整数的补码。

**范例 1-13　求 $+23_{10}$ 的补码**

| 原值 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | $+23_{10}$ |
|---|---|---|---|---|---|---|---|---|---|
| 进位 | | | | | | | | | |
| 第一步（分别反转每个二进制位） | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 反码 |
| 第二步（加1）　+ | | | | | | | | 1 | |
| | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | $-23_{10}$（补码） |

对 $+23_{10}$ 求补码，得出的二进制数实际上是 $-23_{10}$，如果对这个数求补码，那又会回到原来的 $+23_{10}$。

**范例 1-14　求 $-105_{10}$ 的补码**

| 原值 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | $-105_{10}$ |
|---|---|---|---|---|---|---|---|---|---|
| 进位 | | | | | | | | | |
| 第一步（分别反转每个二进制位） | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 反码 |
| 第二步（加1）　+ | | | | | | | | 1 | |
| | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | $+105_{10}$（补码） |

**编程知识：**带符号的二进制数很有用，不过，程序员还必须学会另外一种更有用的数字表示法，即带符号的十六进制数，因为在调试程序的时候，寄存器与内存中的内容一般都是用十六进制表示的。

带符号的十六进制整数，其求补的过程与带符号的二进制整数类似，也分两步，首先分别对每个十六进制位求逆，也就是用15与该数位相减，然后给求逆后的总结果加1。下面这条规律可以迅速判断出十六进制整数的正负：

如果最高有效位（MSD）小于等于7，那么该值为正，若大于等于8，则为负。

范例 1-15 演示了怎样对 $+1AB7_{16}$ 求补。

**范例 1-15　求 $+1AB7_{16}$ 的补码**

↓ 如果最高有效位小于等于 7，那么是正数

| | | | | | |
|---|---|---|---|---|---|
| | **1** | A | B | 7 | $+6839_{10}$ |
| 第一步（分别对每个十六进制位求逆） | 15 − 1 | 15 − 10 | 15 − 11 | 15 − 7 | |
| 反码 | 14 (E) | 5 | 4 | 8 | |
| 第二步（加 1） | | | + | 1 | |
| 补码 | **E** | 5 | 4 | 9 | $-6839_{10}$ |

↑ 如果最高有效位大于等于 8，那么是负数

下面列出一些在各种进制的带符号整数之间转换的方法。

- 带符号的十进制 → 二进制：（1）把原数的绝对值转换成二进制；（2）若原数为负，则对转换出的二进制值求补。
- 带符号的十进制 → 十六进制：（1）把原数的绝对值转换成十六进制；（2）若原数为负，则对转换出的十六进制值求补。
- 带符号的十六进制 → 十进制：（1）若原数为负，则先对其求补；（2）将现在的十六进制数转换成十进制；（3）如果原值为负，则给转换出的十进制值添上负号。

在与十六进制负值相减的时候，可以先对值求补，这样就改变了它的符号，从而把减法化成了与正值之间的加法。

### 1.4.5　怎样保存字符

“A”或“$”这样的字符是逻辑符号，它在底层需要以数字的形式来表示，至于每一个字符究竟应该与哪个整数相对应，则由字符集确定，字符集也叫作字符映射或码表。现在简单介绍几种常见的字符集以及它们之间的关系。

有一种很旧而又很常见的字符集，叫作 ASCII（American Standard Code for Information Interchange，美国信息交换标准代码）。这是一种 7 位字符集，意味着它可以表示 128 种（也就是 $2^7$ 种）不同的字符，其中大部分是英文字母。C++ 等高级语言使用 ASCII 作为默认的字符集。许多网站都能找到完整的 ASCII 字符表，例如 https://en.wikipedia.org/wiki/ASCII。各范围内的 ASCII 字符如表 1-6 所示。

**表 1-6　各范围内的 ASCII 字符**

| 取值范围 | 字　符 |
|---|---|
| 0 ～ 31 | 非打印的控制字符 |
| 32 ～ 126 | 可打印的字符 |
| 127 | Delete（删除，这也是一种控制字符） |

像 ASCII 这样的码表，可以把字符很方便地与整数对应起来。比方说，范例 1-16 就列出了“A”与“$”这两个字符，并根据 ASCII 码表将其转换成对应的二进制整数。

**范例 1-16　ASCII 字符与整数之间的转换**

| ASCII | 十进制 | 十六进制 | 二进制 |
|---|---|---|---|
| A | 65 | 41 | 0100 0001 |
| $ | 36 | 24 | 0010 0100 |

ASCII 可以用很多方式来扩展，以表示其他拉丁系的字母及符号。其中最常见的一种标准扩展方式是 ISO（International Organization for Standardization，国际标准化组织）与 IEC（International Electrotechnical Commission，国际电工委员会）制定的 ISO/IEC 8859。它把 7 位的 ASCII 扩展成 8 位，从而可以表示 256 种（也就是 $2^8$ 种）字符。其他一些扩展方式也是把 ASCII 扩充为 8 位，例如 IBM Code page 437、Windows-1252 及 macOS Roman，这些字符集在 0 至 127 这一范围内与标准的 ASCII 相同，但在 128 至 255 这一范围内则采用不同的办法来编码。

到 2008 年左右，ISO/IEC 10646 与 Unicode 这样的映射方案成了最为流行的字符集。这些复杂的方案支持更多字符，同时还与标准的 ASCII 相兼容。Unicode 字符集能够以任何一种办法来编码，其中最常用的一种叫作 UTF-8（UTF 是 Unicode Transformation Format，Unicode 转换格式）。互联网上的很多网页用的都是这种**字符编码**（character encoding）方式，它支持 1 112 064 个字符，其中每个字符可能要用 1 至 4 个 octet 来表示（octet 指的是由 8 个二进制位所构成的组，相当于一个字节）。此外，还有其他一些 Unicode 字符编码方式，例如 UTF-16 用一个或两个 16 位的字（word）来表示字符，UTF-32 用一个 32 位的值来表示字符。

**编程知识：**Unicode 这样的字符集给其中的每个字符（或者说符号）都指定了一个独特的数字，这个数字有时称为代码点或码位（code point，例如 A 的代码点是 U+0041）。至于代码点究竟应该怎样化为二进制序列，则要由算法来决定，这个算法称为字符编码方式，比方说，如果采用 UTF-8 字符编码方式，那么 U+0041 代码点就会化为 01000001。

Unicode 字符集可以把每一个逻辑符号都映射到互不相同的代码点上，代码点的写法是字母 U 后面添一个加号，再跟十六进制值，如范例 1-17 所示。具体的 UTF 编码方式会把 Unicode 字符转换成一个二进制序列，这个序列可以表示成任意进制。不过要注意，同一个符号在不同的 UTF 编码方式下可能会转换成不同的序列，比方说，范例 1-17 中的版权符号按照 UTF-8 格式来编码所得的结果，就与 UTF-16 及 UTF-32 格式不同。由此可见，同一套字符集（例如 Unicode）里的字符在不同的字符编码方式（例如 UTF-8、UTF-16、UTF-32）下可能会转化为不同的二进制序列。

**范例 1-17　Unicode 字符集中的符号按照各种 UTF 格式来编码所得到的二进制序列**

| 符号 | Unicode | UTF-8（二进制） | UTF-8（十六进制） | UTF-16（十六进制） | UTF-32（十进制） | C++ |
|---|---|---|---|---|---|---|
| T | U+0054 | 01010100 | 0x54 | 0x0054 | 84 | "\u0054" |
| © | U+00A9 | 11000010:10101001 | 0xC2 0xA9 | 0x00A9 | 169 | "\u00A9" |

标准的 7 位 ASCII 码可以视为 ISO-8859 及各种 UTF 编码的子集。比方说，如果把编码中最右侧的二进制位叫作第 0 位，那么只需要给 ASCII 码的左方添一个 0，就可以构成有效的 UTF-8 编码了（此时，编码的第 7 位是 0）。UTF-8 编码的前 128 个字符与 ASCII 码是一一对应的。

字符串通常用字符序列来表示，其中的字符可以是 ASCII 码或 Unicode 码。比方说，如果用 ASCII 码来表示“ABC123”这个字符串，并以十六进制来书写编码后的字符序列，那么就是 41h, 42h, 43h, 31h, 32h, 33h, 0h。最后的 0h 称为 NULL 字符，表示字符串到此结束。

要注意：在编写及调试程序的时候，字符与字符序列一般会用对应的十六进制码来表示。本书第 3 章及第 7 章会详细讲解字符串。

## 1.5　布尔表达式

布尔逻辑是计算机与编程的基本知识。要想学习后续的章节，就必须先了解这些知识。布尔表达式可以用来编写判断语句，以实现某种决策，也可以用来执行按位运算。布尔表达式中的操作数都只能取 0/1 这两种值，或者说，要么是 false（假），要么是 true（真）。下面讲解四种布尔运算：NOT（非）、AND（与）、OR（或）、XOR（异或）。在不同的场合，这四种运算会用不同的符号来表示（参见表 1-7），然而我们在这里还是采用这些运算本身的名字来称呼它们。

表 1-7　用来表示布尔运算的符号

| 运算 | AND | OR | NOT | XOR |
|---|---|---|---|---|
| 数学 | ∧ · & | ∨ + \| | ¬ ' ~ ! | ⊕ |
| C++ | &（按位），&&（逻辑） | \|（按位），\|\|（逻辑） | ~（按位），!（逻辑） | ^（按位） |
| 汇编 | AND | OR | NOT | XOR |

布尔运算的规则（参见表 1-8 ～表 1-11）是：

- NOT：反转操作数（或者说，对操作数取反）。
- AND：只有当参与运算的所有操作数都是 true 时，结果才是 true，否则是 false。
- OR：只有当参与运算的所有操作数都是 false 时，结果才是 false，否则是 true。
- XOR：如果两个操作数中有且只有一个是 true，那么结果是 true。如果两个都是 true 或者 false，那么结果是 false。

表 1-8　NOT 运算

| *x* | NOT *x* |
|---|---|
| F | T |
| T | F |

表 1-9　AND 运算

| *x* | *y* | *x* AND *y* |
|---|---|---|
| F | F | F |
| F | T | F |
| T | F | F |
| T | T | T |

表 1-10　OR 运算

| *x* | *y* | *x* OR *y* |
|---|---|---|
| F | F | F |
| F | T | T |
| T | F | T |
| T | T | T |

表 1-11　XOR 运算

| *x* | *y* | *x* XOR *y* |
|---|---|---|
| F | F | F |
| F | T | T |
| T | F | T |
| T | T | F |

二进制的 0 表示 false，1 表示 true。范例 1-18 以两个二进制数为例，演示了一些布尔表达式的运算结果。

范例 1-18　二进制数的布尔运算

假设

| | |
|---|---|
| *x* | 1110 1101 |
| *y* | 0110 1010 |

结果

| | |
|---|---|
| NOT (*x*) | 0001 0010 |
| AND (*x* AND *y*) | 0110 1000 |
| OR (*x* OR *y*) | 1110 1111 |
| XOR (*x* XOR *y*) | 1000 0111 |

在后面的章节中大家就会看到，这四种布尔运算对汇编语言很有用。比方说，NOT 运算可以用来计算反码，AND 运算可以用来执行栈对齐，也可以切换大写与小写，OR 运算可以判断数字是大于、小于还是等于 0，XOR 运算可以用来清零，还可以判断某数是奇数还是偶数。

此外，通过**掩码**（mask），可以用 AND 运算来关闭二进制位（也就是将其设置成 0），用 OR 运算来打开二进制位（也就是将其设置成 1）。掩码是用来操作位域（bit field）以完成某种任务的值。范例 1-19 演示了掩码的三种用法。第一种是用 OR 操作把每个二进制位都设置成 1，第二种是用 AND 操作把每个二进制位都设置成 0，第三种更实用，可以判断 2 号位置上的二进制位（此处假设从 0 开始编号，最右侧的二进制位是 0 号）是否是 1。构造这样一个掩码，把有待判断的位置写成 1，其余的位置写成 0，然后对受测数字及该掩码求 AND，若结果不为 0，则说明受测数字的对应二进制位是 1。

**范例 1-19　掩码的用法**

| 用 OR 运算及掩码把二进制位设置成 1 | 用 AND 运算及掩码把二进制位设置成 0 | 用 AND 运算及掩码来判断某个位置是否为 1 |
|---|---|---|
| 1001 待操作的值<br>OR 1111 掩码<br>---<br>= 1111 操作后的值 | 1001 待操作的值<br>AND 0000 掩码<br>---<br>= 0000 操作后的值 | 1101 待判断的值<br>AND 0100 掩码<br>---<br>= 0100 判断结果：2 号位是 1 |

## 1.6　3 位计算机示例

**学习指南：**本节内容的演示请参阅视频 1.3。

程序员一般是在 32 位或 64 位环境下编写程序的，不过，用位数比较少的环境来形象地演示一些范例，更能帮助我们理解这些内容。因此，下面用 3 位计算机（如表 1-12 所示）来举一些例子。

**表 1-12　3 位计算机**

| 二进制 | 原码 | 反码 | 补码 | |
|---|---|---|---|---|
| 000 | 0 | 0 | 0 | 以 0 开头的都是非负数 |
| 001 | +1 | +1 | +1 | |
| 010 | +2 | +2 | +2 | |
| 011 | +3 | +3 | +3 | |
| 100 | −0 | −3 | −4 | 以 1 开头的都是负数 |
| 101 | −1 | −2 | −3 | |
| 110 | −2 | −1 | −2 | |
| 111 | −3 | −0 | −1 | |

无论计算机是多少位的，下面这几条规则均适用于补码表示法。

- 0 总是用所有二进制位都是 0 的数来表示
- −1 总是用所有二进制位都是 1 的数来表示
- 值最小的负整数总是用开头为 1 且其余二进制位均为 0 的数来表示
- 值最大的正整数总是用开头为 0 且其余二进制位均为 1 的数来表示

下面看看 3 位计算机是怎样用补码表示法来做整数运算的。这几个例子分别演示了几种不同的情况。

范例 1-20　用 3 位计算机做加减法

| #1 | | #2 | | #3 | | #4 | | #5 | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 010 | 3 | 011 | 2 | 010 | 3 | **0**11 | -4 | **1**00 |
| +1 | +001 | -1 | +111 | -4 | +100 | +2 | +**0**10 | +-3 | +**1**01 |
| 3 | 011 | 2 | ~~1~~010 | -2 | 110 | 5 | **1**01 | -7 | ~~**1**~~**0**01 |

第一个例子演示的是两个正数相加的情况。这种情况可以得到正确的结果。第二个例子是正数与正数相减，这可以化为正数与负数相加。由于运算过程中最高位发生了进位，因此，运算结果是 4 位，这超出了 3 位计算机的表示范围，于是，计算机就会把这个进位忽略掉，去掉该位之后的结果是正确的。第三个例子还是正数与正数相减，这依然可以化为正数与负数相加，与前例不同的是，这次的计算结果本身就在 3 位整数的表示范围之内。第四个例子是把两个 3 位正整数相加，然而运算结果却超出了 3 位整数的表示范围，这可以从结果的最高位来判断：由于该位是 1，因此，运算结果是负数。两个正值相加，结果为负，这意味着发生了溢出，此时，CPU 的状态寄存器会把表示溢出的标志位设置成 1，这个话题将在第 2 章讲解。第五个例子是负数加上负数，也就是 −4 加上 −3，虽然这两个值都在 3 位整数的表示范围内，但是其运算结果却超出了范围。运算结果是 4 位，去掉最左侧一位之后，在剩下来的 3 位中最高位是 0，因此，它是个正值。两个负数相加得到了正数，这意味着发生了溢出。从第 4 章开始，将会讲解怎样处理各种溢出情况。

**编程知识：**计算机怎么判断一个整数究竟是带符号的还是不带符号的？其实它也不知道。无论涉及的是什么值，CPU 都只是会在执行完算术运算后用布尔逻辑来设定相关的状态标志。至于这些标志是应该予以考虑还是可以忽略不计，则要由程序员依照该运算的具体情况来确定。状态标志及相关的话题会在后面的章节中详细讲解，尤其是第 2 章与第 4 章。

## 1.7　小结

本章讲解了计算机的基础知识，其中包括计算机编程语言之间的关系、数据的表示方式，以及布尔表达式等，并通过 3 位计算机演示了相关的内容。各种计算机语言合起来构成一套体系，而笔者则把这套体系比喻成一条翻译流水线。其中，汇编语言处在承上启下的位置，它使得开发者能够更好地理解并探究与计算机操作有关的一些底层细节。我们还看到了怎样用各种进制的计数系统来表示数据，以及如何用各种字符集来表示字符。然后，我们讨论了布尔运算，这些运算可以用来实现其他一些更为复杂的任务。最后，我们用 3 位的计算机系统演示了二进制补码表示法的运作原理。了解到这些基本的概念之后，大家就可以继续学习书中更为高级的内容了。

## 1.8　关键术语

American Standard Code for Information Interchange（ASCII 美国信息交换标准代码）

assembler（汇编器）

bit（二进制位；比特；位元）

byte（字节；位元组）
character encoding（字符编码）
character set（字符集）
compiler（编译器）
compiling（编译）
Complex Instruction Set Computing（CISC，复杂指令集）
computer architecture（计算机系统结构；计算机架构；计算机体系结构）
debugger（调试器）
disassembly（反汇编）
high-level language（高级语言）
Instruction Set Architecture（ISA）（指令集架构；指令集体系）
interpreting（解释）
linker（链接器）
loader（加载器）
machine-level（机器级别；机器层面）
mask（掩码）
microarchitecture（微架构；微体系结构）
object code（目标代码）
Reduced Instruction Set Computing（RISC，精简指令集）
signed（带符号的；有符号的）
translation pipeline（翻译流水线）
Two's Complement（补码）
Unicode（Unicode）
unsigned（无符号的；不带符号的）

## 1.9 代码回顾

笔者用冒泡排序来举例有两重含义：第一，是想告诉大家汇编语言并没有想象中枯燥；第二，是想展示它在文件尺寸与性能方面的优势（这只是个非常简单的例子，在更为复杂的一些情况下，它的优势会更加明显）。下表为两种语言实现的程序之间的对比：

用汇编语言及 C++ 语言实现的冒泡排序程序之间的对比

| | 内存占用量 | .s 文件 | .cpp 文件 | .o 文件 | 可执行文件 |
|---|---|---|---|---|---|
| 汇编语言 | 348KB | 346B | N/A | 1KB | 9KB |
| C++ 语言 | 424KB | 21KB | 410B | 3KB | 9KB |

请注意，汇编语言的 .s 文件（也就是包含汇编代码的源文件）与 C++ 编译器所产生的 .s 文件大小相差很远。C++ 编译器所生成的 .s 文件总共 21KB，其中可执行的代码大约 3KB，而由此生成的 .o 文件大小也是 3KB 左右。在最终的可执行文件上，两种语言相差不远，然而在内存占用量上，汇编语言则稍微低一些。

冒泡排序程序的汇编代码及 C++ 代码

```
# Assembly Bubble Sort
.data
array: .long 3, 2, 6, 4, 1
count: .long 5

.text
.globl _main
_main:

mov count, %ecx
dec %ecx

outerLoop:
push %ecx
lea array, %esi

innerLoop:
```

```
// C++ Bubble Sort
int main(){

  int  size = 5;
  int  value[5] = {3, 2, 6, 4, 1};
  int  i, tmp;
  bool swapped;

  do
  {
     swapped = false;
     for (i = 0; i < size - 1; i++){
        if (value[i + 1] < value[i])
        {
           tmp = value[i];
           value[i] = value[i + 1];
           value[i + 1] = tmp;
```

```
mov (%esi), %eax
cmp %eax, 4(%esi)
jg nextStep
xchg 4(%esi), %eax
mov %eax, (%esi)

nextStep:
add $4, %esi
loop innerLoop
pop %ecx
loop outerLoop

push $0
sub $4, %esp
mov $1, %eax
int $0x80
.end
```

```
            swapped = true;
        }
      }
    } while (swapped);

    return 0;
}
```

## 1.10　习题

### 填空题

1. x86 与 IBM 370 说的都是处理器的_____。
2. 汇编语言的语句与机器语言的语句之间是一对_____的关系。
3. 高级语言的语句与机器语言的语句之间是一对_____的关系。
4. 把上层语言翻译成下层语言有两种方式，分别叫作_____与_____。
5. x86 与 x86_64 用的都是_____指令集架构。
6. 在 x86 架构中，word（字）由_____个二进制位组成。
7. 由于比较可靠，因此计算机采用_____作为基本的计数系统（数值、进制）。
8. 表示十进制值所需的二进制位个数，可以根据公式_____来计算。
9. 单字节的无符号整数可以表示从_____到_____之间的值。
10. 把十六进制转换成二进制的时候，每个十六进制位用_____个二进制位来表示。
11. 在 x86 环境中，某数的最高有效位是 1，意味着该数是_____数。
12. 在补码表示法中，以 7 开头的十六进制值是_____值。
13. 单字节的带符号整数可以表示从_____到_____之间的值。
14. 字符是以_____的形式来存储的。
15. 有一种布尔运算会在两个操作数均为 true 的时候得出 false 的结果，这种运算是_____。
16. 如果计算结果超出了计算机所能表示的范围，那么我们就说该计算发生了_____。

### 判断题

1. 汇编代码可以在不同系列的处理器之间移植。
2. 在补码表示法中，0 有两种表示方式。
3. x86_64 架构与 IA-64 架构不同。
4. 汇编器会把用高级语言写成的语句编码为目标代码。
5. CPU 本身就能知道某个整数是正值还是负值。

## 1.11 作业

1. 阐述学习汇编语言的实际理由。
2. 补码表示法所依据的数学原理是什么？该原理有何意义？
3. 完成第 1 章的附加作业。
4. **挑战题：**对高级语言的目标文件做反汇编。例如可以从已有的 C++ 项目里选取一份目标文件做实验。操作系统与反汇编器可以任选。你可以参考本书的附录 C 来完成这道题。查看反汇编的结果，并找出占用字节数最少和最多的指令。

第 2 章

Assembly Programming and Computer Architecture for Software Engineers

# 处理器与计算机系统体系结构

## 目标

- 认识计算机系统体系结构的各个组成部分
- 了解各种类型的存储器与存储器位置之间的区别
- 了解 CPU 中的子组件
- 描述汇编语言所用到的 CPU 寄存器
- 在各种输入与输出方法之间权衡

## 网络资源

**维基资源**

- https://en.wikipedia.org/wiki/X86
- https://en.wikipedia.org/wiki/X86-64

**技术资源**

- http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- https://developer.amd.com/resources/developer-guides-manuals/

## 2.1 简介

本章概述 x86 与 x86_64 架构。想要高效地撰写汇编代码，就必须透彻地了解运行程序所用的硬件，其中尤为重要的是处理器。你必须知道它在执行周期的每一个步骤中是怎样处理数据的。第 1 章说过，汇编指令与处理器所实现的具体操作之间是直接对应的，由于有着这种直接对应的关系，因此必须懂得处理器架构才能把汇编程序写好，本书后面各章还要详细讲解这种关系。虽说本章的大部分内容都在谈处理器，但为了能把数据从输入到接受处理再到输出的整个流程讲清楚，笔者还会提到其他一些重要的组件。

本章可以帮助你牢固地掌握执行汇编程序所需的底层硬件，而后面各章则会教你怎样撰写运行在这些硬件之上的汇编程序。

> **提示：** 虽说处理器架构有很多种（例如 x86、x86_64、IA_64、ARM、Alpha、Sparc、PowerPC 等），但本书主要关注 x86 与 x86_64，因为它们是计算机市场上最常见的，同时也是汇编语言开发者最常使用的架构。

## 2.2 体系结构概述

宏观地来看，计算机系统是由几种主要的组件构成的，这些组件合起来就形成了我们

常说的计算机。**中央处理器**（Central Processing Unit，CPU）当然是计算机的中枢，然而计算机要想正常运作还必须有其他一些组件，例如主板、内存以及各种输入 / 输出（Input/Output，I/O）设备。图 2-1 描述了这些常见的组件一般是怎样安排在主板上的。

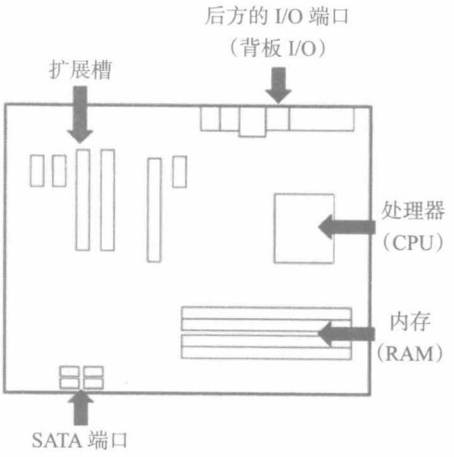



图 2-1 主板上的组件

**主板**是连接计算机主要组件的板卡，它相当于一套高速公路系统，以供数据在组件之间移动。主板通过各种连接手段与组件相连，例如通过内存插槽与内存条相连，通过 CPU 插座（CPU Socket）与 CPU 相连，通过数据线与硬盘及光驱相连，通过扩展槽与扩展卡相连，通过连接线与键盘、鼠标及其他一些外部 I/O 设备相连等。为了与这些组件通信，数据必须沿着主板中的通信渠道来传送，这些通信渠道称为总线（bus）。

**总线**是指主板中的一组线路或导线，用来在组件之间传输数据。从背面观察主板很容易就能看到这些总线，如图 2-2 所示。由于导线所经过的地方各层纤维玻璃之间含铜，因此这些地方的颜色看上去比周围浅。

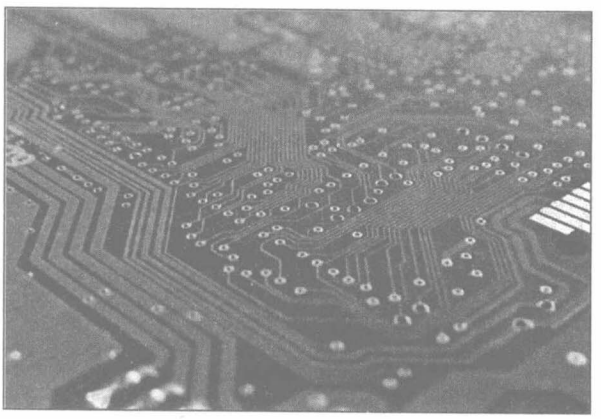


图 2-2 从背面观察到的主板

计算机中最主要的总线叫作**系统总线**（system bus），如图 2-3 所示，它实际上是数据总线、地址总线与控制总线这三者的合称。系统总线使得 CPU 能够与内存及计算机中的其他 I/O 设备相通信，为了确保通信正常，这些与 CPU 通信的设备必须与对应的总线连接起来。**数据总线**（data bus）用来在组件之间传递指令及数据，其中的指令指的是从内存中加载数

据、把数据放入内存，或是从光驱中读取数据等动作。为了正确传递信息，还必须传输指令所针对的内存地址和数据。**地址总线**（address bus）就负责地址方面的通信工作。此外，**控制总线**（control bus）在系统组件的通信工作中也很重要，它会在组件之间传输信号，使得组件能够在适当的时机通信，以确保同步运作（例如，某组件不应该在某设备处于繁忙状态时从该设备中读取数据）。

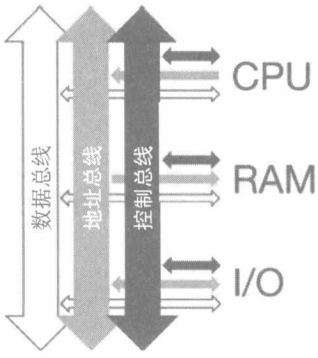



图 2-3　系统总线

除了系统总线，还需要一个部件才能使组件之间得以通信。这个部件就好比节拍器。节拍器按照一定的速度打拍子，使得表演者能够跟上音乐的节奏，而该部件也是按照一定的频率来产生脉冲，使得计算机的组件之间能够在适当的时机相互通信，而不会以任意的时间间隔随意地进行沟通。因为那样就有可能出现其中一个组件要求通信而另一个组件尚未准备好或暂时无法通信的情况。这个部件就是主板上的**系统时钟**（system clock）。

系统时钟的基本单位叫作**时钟周期**（clock cycle），它的前半段称为高电平期（up-tick），此时电压由低变高（用二进制来表示就是从 0 变成 1），后半段称为低电平期（down-tick），此时电压由高变低（或者说从 1 变成 0）。图 2-4 演示了整个时钟周期。

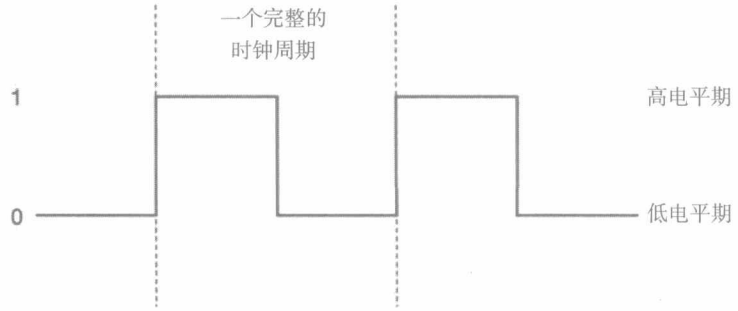



图 2-4　时钟周期

要传输数据就必须有地方来存储这些数据及指令，为此，计算机必须拥有**存储器**（memory）。然而在开始讨论存储器之前，首先必须知道计算机中有哪几种类型的存储器。图 2-5 描述了各类存储器之间的层次关系。

最顶层的存储器距离上最接近 CPU 的运算电路（arithmetic circuit，该电路通常位于 CPU 自身中），而最底层的存储器则离 CPU 最远。存储器离 CPU 越远，容量越大，价格越低，而速度也越慢；反之，离 CPU 越近，容量越小，价格越高，而速度也越快。至于速度、容量与成本之间的具体关系，则要看每个位置上的存储器是什么类型。

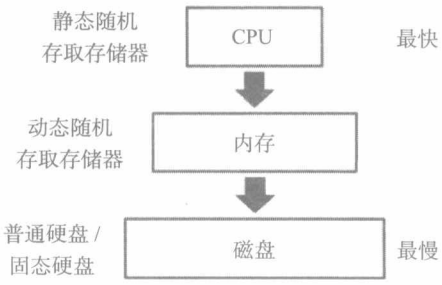



图 2-5　各类存储器之间的层次关系

**编程知识：**由于离 CPU 越远的存储器速度越慢，因此，开发汇编程序的人应该清醒地使用这些存储器。也就是说，如果程序中的数据很少且可以放入缓存（cache）中，那就不要把它放到内存（RAM）中，以免降低程序的运行速度。同理，如果程序中的数据可以放入内存中，那就不要将其写入磁盘，否则就得花时间去读写磁盘了。

离 CPU 最近的存储器是**静态随机存取存储器**（Static Random Access Memory），简称 SRAM。它实际上就在 CPU 芯片中。下一节会讲到，这种存储器通常称为缓存。SRAM 是速度最快的存储器类型。比它稍慢一些的是动态随机存取存储器（Dynamic RAM），简称 DRAM。主板上有一些靠近 CPU 的槽位，其中插着的内存条就属于这种类型的存储器。大家通常所说的主存（main memory）一般指的就是 DRAM，也可以简称 RAM。还有一种存储器比 DRAM 还慢，这就是机械硬盘或固态硬盘等**磁盘**（disk），主要相当于我们常说的硬盘，后面会讲到它。这种存储器用来长期保存数据，而不像缓存或 RAM 只用来暂时保存。向 CPU 发送数据或是从中获取数据的时候，必须与系统时钟的节奏相合，而访问这些类型的存储器中的数据时也同样要如此。

访问主存（也就是 DRAM）通常需要经过 4 个步骤，而每个步骤至少需要花费 1 个时钟周期来完成（有时还会更多），因此访问主存所需的总时间至少是 4 个时钟周期。

1. 把待读数据的内存地址放入地址总线。

2. 修改处理器的 Read（RD）标志，这是个起断言（assertion）作用的标志，也就是说，如果处理器在执行读取操作时发现该标志的值没有设置正确，就会 halt（暂停）。

3. 等待内存控制器给出响应（至少要 1 个时钟周期）。

4. 把数据从数据总线复制到目标位置。

操作内存的时候必须注意存储方式。计算机通常会依照具体的架构来把数据以字节为单位分块（chunk）保存（例如 x86 的块是 32 位，x86_64 的块是 64 位）。如果要把 32 位的十六进制值 `12345678h` 保存到计算机的内存中，首先需要像表 2-1 这样将其写为 4 个字节。由于每个十六进制位相当于 4 个二进制位，因此两个十六进制位就相当于 8 个二进制位，也就是 1 个字节。

表 2-1 把 32 位值分成 4 个字节

| 值 | | | |
|---|---|---|---|
| 字节 1 | 字节 2 | 字节 3 | 字节 4 |
| 12 | 34 | 56 | 78 |

为了保存该值，首先必须找到可以使用的内存位置。为简单起见，我们假设该值能够保存到 `0x00000000h` 这个位置上。一般来说都会像表 2-2 这样把数据最左侧的字节（本例中的 12）保存到地址编号最小的位置（也就是表格第一行所写的位置 0x00000000）上，而把数据最右侧的字节（本例中的 78）保存到地址编号最大的位置（也就是表格最后一行所写的位置，0x00000003）。

表 2-2 大端序

| 内存地址 | 数 据 | 内存地址 | 数 据 |
|---|---|---|---|
| 0x00000000 | 12 | 0x00000002 | 56 |
| 0x00000001 | 34 | 0x00000003 | 78 |

由于内存地址也是以字节（或者说 8 个二进制位）为单位增长的，因此，像这样按照从左到右的顺序把数值中的字节保存到内存中的对应地址是符合书写习惯的。不过要注意这里的单位是字节而不是数位。表 2-2 这种存储方式是把权重最大的字节（或者说最高有效字

节，most significant byte）保存到地址最低的地方，由于权重最**大**的字节（也就是“大端”，big end）出现在最**前**面，因此这种方式叫作**大端在前的字节顺序**（Big-Endian byte order），简称大端序或大尾序。还有一些计算机系统采用相反的顺序，把权重最**小**的字节（“小端”，little end）保存在最**前**面，这种保存方式称为**小端在前的字节顺序**（Little-Endian byte order），简称小端序或小尾序。表 2-3 演示了怎样按照小端序来保存值 12345678h。

表 2-3　小端序

| 内存地址 | 数　据 | 内存地址 | 数　据 |
|---|---|---|---|
| 0x00000000 | 78 | 0x00000002 | 34 |
| 0x00000001 | 56 | 0x00000003 | 12 |

**编程知识：**如果用常见的办法（也就是基本的变量）来访问内存中的数据，那么不用关心计算机用的是大端在前还是小端在前的字节顺序。但如果要通过内存地址来访问或查看某个位置上的数据，那就必须注意这个问题了，因为只有这样才能通过正确的地址找到你想要的数据。

Intel 的 x86 与 x86_64 采用小端序，Motorola 的 68XX、68XXX 系列处理器以及 IBM 的 Z 系列采用大端序。还有的处理器两种顺序都支持，例如 Sun SPARC、ARM 及 PowerPC，这称为双端序（bi-endian）。

## 2.3　处理器

CPU 或者说处理器可以视为计算机的大脑。在计算机系统中，主要的算术运算与逻辑运算都靠这个部件来处理。宏观地来看，CPU 由 4 个主要的部分组成，它们是：算术逻辑单元（Arithmetic Logic Unit，ALU）、控制单元（Control Unit，CU）、CPU 时钟（CPU clock）及存储器（包括缓存和寄存器），参见图 2-6。

ALU 是 CPU 中执行数学运算的子组件，它所执行的算术与逻辑运算针对的是整数型操作数（要注意操作数的类型是**整数**，浮点数由另一个组件处理）。CU 负责指挥 CPU 中的数据流，以确保 CPU 里的其他子组件能够在适当的时机接收到正确的数据，并做出相应的处理。CU 还必须把指令执行周期（Instruction Execution Cycle）安排好，使得 CPU 指令能够据此得以执行（此外，为了正确执行这些指令，计算机还需要完成其他一些子任务）。**CPU 时钟**与系统时钟不同，它是 CPU 本身的时钟，用来为 CPU 的操作计时。

有一个和 CPU 时钟相关的术语叫作时钟**频率**，它的单位是赫兹（Hertz，Hz）。说得简单一些：频率为 1 Hz 的处理器其时钟每秒震荡 1 次。这里所说的震荡一次，是指像图 2-4 那样完整地经历高电平期与低电平期。当前的处理器都是以 MHz（megahertz，兆赫）或 GHz（gigahertz，千兆赫）来描述频率的，1 MHz 意味着每秒震荡一百万次，1 GHz 意味着每秒震荡十亿次。如果要用时钟周期而不是频率来描述处理器的快慢，就给频率取倒数。例如，1GHz 的 CPU 其时钟每秒钟脉冲十亿次，因此每个时钟周期的长度就是十亿分之一秒。有一个重要的现象要注意：CPU 的

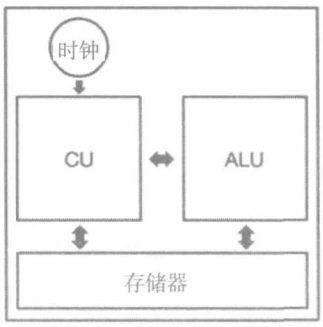



图 2-6　CPU 的组件

时钟频率是系统时钟频率的倍数，具体是几倍由**倍频系数**（multiplier）决定[⊖]。比方说，如果系统时钟运行在 800 MHz 的频率上，而倍频系数是 4，CPU 的频率就是 3.2 GHz。由于 CPU 的时钟运行速度为系统时钟的 4 倍，因此它在同样长的时间内所能完成的操作数量也是后者的 4 倍。

### 2.3.1 缓存与寄存器

由于处理器要对数据执行操作，因此必须要有地方来保存这些用作操作数的数据，此外，操作结果以及操作所涉及的地址也得有地方保存。前面讲解存储器的层次结构时曾经说过，离 ALU 越远，存储器中的数据访问起来越慢。于是，为了令 CPU 能够尽快执行操作，必须想办法把指令与数据放到它能够迅速访问的存储器中，也就是说，存放这些指令与数据的存储器离 ALU 及 CU 越近越好。这种紧邻 CPU 的存储器就叫作 cache（高速缓存，简称缓存），实际上，它跟逻辑电路一起位于 CPU 芯片中。缓存在存储器的层次结构中有其地位，然而它内部还有一套自己的层次结构。

当前的处理器缓存通常分为三个级别，分别是 L1 缓存（一级缓存）、L2 缓存（二级缓存）与 L3 缓存（三级缓存）。缓存本身的层次结构与存储器的层次结构都遵循同一条原则：距离 ALU 越远容量越大，价格也越便宜。L1 与 L2 缓存都离 ALU 很近，不过 L2 要比 L1 稍远一些，因此其容量也大一些。L3 缓存一般出现在多核处理器中，它为所有 CPU 核心所共享，而 L1 与 L2 缓存则每个 CPU 核心都配有一套，如图 2-7 所示。L3 缓存是静态存储器中的最后一层，如果数据不保存在该层及其上方的各层中，那就只好放到它下方的 RAM 中了。

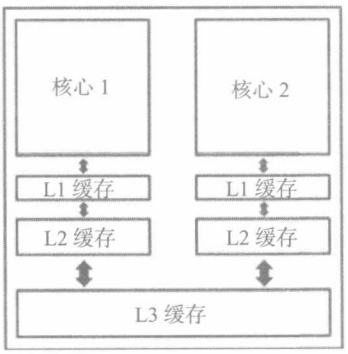



图 2-7 CPU 缓存

现在以 Intel Core i7 系列的处理器为例来说明缓存的层级：i7 处理器的每个核心都配有 64KB 的 L1 缓存，其中 32KB 保存指令，32KB 保存数据。此外，还配有 256KB 的 L2 缓存。L3 缓存的容量是 4MB ～ 24MB，具体要看你买的是便宜一些的型号还是贵一些的型号。

CPU 提供了一些与缓存有关的指令，然而开发者一般都不用专门编写代码去访问或操作缓存，因为缓存是由复杂的算法来控制的，以确保程序所需的数据能够尽量出现在缓存中，所以开发者通常不需要干预这套机制。比方说，如果程序频繁引用某个变量，那么处理器就有可能认为这份数据相当重要，从而将其预先获取（prefetch，简称预取）出来并放入缓存，使得程序以后访问该数据时能够快一些。CPU 会在执行程序的过程中随时根据情况来做出这种处理。

除了缓存之外，CPU 中还有一种存储器叫作**寄存器**（register），它的内容可以通过明确的地址来访问，而且在此类存储器中它是最快的一种。它位于整个存储器层级的最顶端，其容量比缓存更小，速度也比缓存更快。寄存器是最贴近 ALU 的一小块存储区域，用来保存执行指令时所涉及的操作数、地址及结果。

⊖ 俗称主频，主频 = 外频 × 倍频系数。——译者注



**提示**：处理器的每个核心都有自己的一套 L1 与 L2 缓存，与之类似，每个核心也都有自己的一套寄存器。然而，编写汇编代码的时候你不用指出当前操作的寄存器究竟处在哪个核心上，你只需要写出寄存器的名字就可以了，至于这个寄存器到底指的是哪个核心上的寄存器则由 CPU 决定。表 2-4 列出了这些名称。

表 2-4　x86 与 x86_64 的寄存器

| | 64 位 | 32 位 | 16 位 | | 64 位 | 32 位 | 16 位 |
|---|---|---|---|---|---|---|---|
| 通用目的寄存器 | RAX | EAX | AX | 段寄存器 | N/A | CS | CS |
| | RBX | EBX | BX | | | DS | DS |
| | RCX | ECX | CX | | | ES | ES |
| | RDX | EDX | DX | | | SS | SS |
| | RSI | ESI | SI | | | FS | |
| | RDI | EDI | DI | | | GS | |
| | RBP | EBP | BP | 指令指针寄存器 | RIP | EIP | IP |
| | RSP | ESP | SP | 标志寄存器 | RFLAGS | EFLAGS | FLAGS |
| | R8-R15 | | | | | | |

表 2-4 中的寄存器可以分成 4 类：通用目的寄存器（General Purpose Register）、段寄存器（Segment Register）、标志寄存器（Flags Register）及指令指针寄存器（Instruction Pointer Register）。汇编程序所操作的基本上都是通用目的寄存器，其中，32 位的通用寄存器有 8 个，64 位的有 16 个。通用寄存器用来执行计算或移动数据。由于 64 位处理器是在 32 位设计方案的基础上构建的，而 32 位处理器又是在 16 位设计方案的基础上构建的，因此新式处理器不仅可以通过寄存器本身的名字来使用该寄存器，而且还能通过旧式处理器所用的名字将其当成旧式的寄存器来使用。此外，如果你要使用的数据或是你要执行的运算只需占据 8 个二进制位，那么可以把 16 位的寄存器想象成两个 8 位的寄存器，这样就可以用它来保存两份数据了。图 2-8 以 64 位的 *rax* 寄存器为例来演示怎样以 8 位、16 位、32 位及 64 位的方式使用它。

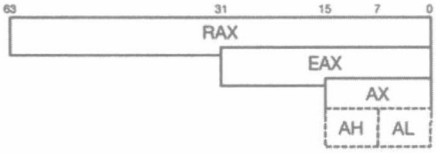



图 2-8　寄存器寻址

由于上述寄存器之间有所重叠，因此会引发一个问题：以不同的名义来操作寄存器会不会使其中的数据受到影响？比方说，如果以 *eax* 的名义保存了一个 32 位的值，然后又以 *ax* 的名义保存了一个 16 位的值，那么 *eax* 的低 16 位是得以保留，还是遭到覆盖？图 2-9 给出了解答。由该图可知，无论以什么名义来使用寄存器操作的都是同一套二进制位，具体到本例来说，这意味着无论你是用 *rax* 来操作，还是用 *eax*、*ax*、*ah* 或 *al* 来操作，你所操作的二进制位其实都位于 *rax* 寄存器本身的那 64 个二进制位中，而不是说总共有 128 个二进制位可以使用，其中 64 个划拨给 *rax* ，32 个划拨给 *eax*，16 个划拨给 *ax*，8 个划拨给 *ah*，8

个划拨给 *al*。同样的规律也适用于其他类似的情况：凡是以小寄存器的名义来操作大寄存器的，其实操作的都是大寄存器中与这个小寄存器相对应的那一部分二进制位。

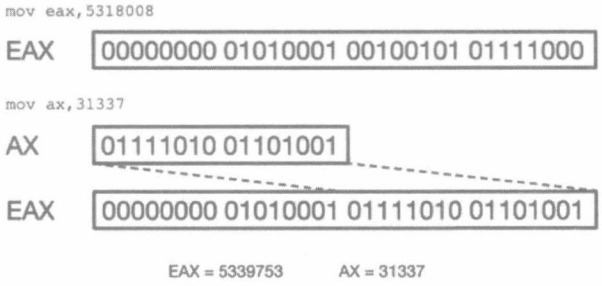



图 2-9 以 *ax* 的名义操作 *eax*，会破坏 *eax* 中已有的一部分数据

从图 2-9 中可以看出，如果把值复制到 *ax* 中，那么实际上会把 *eax* 的低 16 位给覆盖掉，导致 *eax* 中原有的值遭到破坏。之所以出现这种问题是因为 *eax* 与 *ax* 共用这 16 个二进制位。不过并非所有的寄存器都像 *rax* 这样可以通过 *rax*、*eax*、*ax*、*ah* 及 *al* 等不同的名义来使用，即便可以这样用，其二进制位的共享情况也未必和此处所举的例子相同。表 2-5 列出了 *rax*、*rbx*、*rcx* 及 *rdx* 这 4 个 64 位寄存器与其 32 位、16 位、8 位子寄存器之间的共用情况。另外 4 个 64 位寄存器，也就是 *rsi*、*rdi*、*rbp* 及 *rsp* 与其 32 位及 16 位子寄存器之间的共用情况参见表 2-6。

表 2-5 *rax*、*rbx*、*rcx* 及 *rdx* 与其子寄存器之间的重叠情况

| | 寄存器名称 | | | |
|---|---|---|---|---|
| 64 位 | RAX | RBX | RCX | RDX |
| 低 32 位 | EAX | EBX | ECX | EDX |
| 低 16 位 | AX | BX | CX | DX |
| 低 16 位中的高 8 位 / 低 16 位中的低 8 位 | AH/AL | BH/BL | CH/CL | DH/DL |

表 2-6 *rsi*、*rdi*、*rbp* 及 *rsp* 与其子寄存器之间的重叠情况

| | 寄存器名称 | | | |
|---|---|---|---|---|
| 64 位 | RSI | RDI | RBP | RSP |
| 低 32 位 | ESI | EDI | EBP | ESP |
| 低 16 位 | SI | DI | BP | SP |

尽管刚才提到的那些寄存器都可以用在汇编程序里，但是必须注意，有些寄存器是有特殊用途的。如果不加注意，那么保存于某个寄存器（例如 *rax*/*eax*）中的数据就有可能在执行完下一项操作之后遭到修改，从而产生违背开发者意图的效果。下面列出某些 64 位及 32 位寄存器的特殊用法，以提醒大家避免相关的编程错误。

- *rax*/*eax* 通常是默认的累加寄存器。乘法等操作会将其中一部分结果自动存放到 *rax*/*eax* 中，调用函数的时候也需要把返回值保存在 *rax*/*eax* 中。因此，执行这些操作时不要用 *rax*/*eax* 保存一般的数据。
- *rcx*/*ecx* 用来在执行循环的过程中记录循环计数器的值。因此，在循环内部不要用 *rcx*/*ecx* 保存一般的数据。
- *rbp*/*ebp* 用作栈帧中的帧指针，这会在第 6 章讲解。该寄存器用来指向栈中的数据，笔者建议只把它当作专门的寄存器来用。

- *rsp/esp* 是栈指针寄存器，这也是个与栈管理有关的寄存器，它一般指向活动栈帧的顶部。与前一个寄存器一样，这个寄存器也只应该当成专门的寄存器来用。
- *rsi/esi* 与 *rdi/edi* 是索引寄存器（index register，也称为变址寄存器），它和 STOSB、MOVSB 与 SCASB 这样的字符串操作结合起来使用，以便保存、加载或扫描大量的数据。这些操作实际上会把 CPU 置于一种自动循环模式中，这要比开发者手工编写循环更有效率。
- *rip/eip* 是扩展版的指令指针寄存器。这个寄存器用来指向内存中的地址，以表示接下来应该获取、解码并执行的指令，它是在程序运行过程中自动调整的，不应该通过编程的手段修改。
- *rflags/eflags* 是状态与控制寄存器，这会在接下来的内容里详细讲解。LAHF 与 SAHF 等特殊指令可以把 CPU 的一些状态标志载入 *ah* 寄存器，或是将 *ah* 寄存器里的值保存到状态寄存器中。除此以外，不应该用其他手段直接修改 *rflags/eflags*。该寄存器里的二进制位是在执行完算术运算之后根据一套布尔规则自动设置的。尽管 *rflags* 是 64 位，但其中能够用到的只有低 32 位，因此，x86 与 x86_64 处理器用的是同一套状态标志。

**CPU 标志**（CPU flag）是一些二进制位的统称，这些二进制位分别用来以某种方式控制 CPU 操作或反映 CPU 操作的状态。表 2-7 列出了大多数开发环境中值得关注的 8 个标志位，其中某些标志可以由开发者通过 LAHF 及 SAHF 指令来操作。表 2-8 列出了可以由这些指令所编辑的标志位。对于标志位来说，set（设置）的意思是将其设为 1（也称为置 1），clear（清除）的意思是将其设为 0（也称为置 0）。

表 2-7　值得关注的标志位

| 标　志 | 代　号 | 对应的二进制位在寄存器中的位置 | 描　述 |
|---|---|---|---|
| Carry（进位） | CF | 0 | 如果无符号整数的加法的最高有效位需要进位，或是无符号整数的减法的最高有效位需要借位，那么该标志就会得到设置。这有可能意味着结果太大无法保存到目标中，例如 8 位的寄存器就无法保存 255 + 5 的结果，因为计算的时候会产生从第 8 位到第 9 位的进位（假设最右侧的二进制位称作第 1 位） |
| Parity（奇偶；校验） | PF | 2 | 若结果中值为 1 的二进制位是偶数个，那么该标志就会得到设置。错误修正操作会用到这个标志 |
| Auxiliary Carry（辅助进位）或 Adjust（调整） | AF | 4 | 主要用在 BCD 运算中，BCD 指的是以二进制形式编码的十进制数。如果计算过程中低 4 位（这 4 个二进制位）需要向高 4 位进位或借位，那么该标志就会得到设置 |
| Zero（零） | ZF | 6 | 如果运算结果是 0，那么该标志就得到设置 |
| Sign（符号） | SF | 7 | 如果运算结果是负值（或者说如果运算结果用补码形式来表示，其最高有效位是 1），那么该标志就得到设置 |
| Interrupt（中断） | IF | 9 | 如果该标志得到设置，那么计算机就能够处理可屏蔽的硬件中断或者说中断请求（Interrupt Request，IRQ） |
| Direction（方向） | DF | 10 | 如果清除，那么字符串就会按照从左到右的顺序处理，体现为 rsi/esi 及 rdi/edi 这两个字符串索引寄存器会在每次字符串操作（例如 MOVS）之后递增。如果设置，那么字符串就会按照从右至左的方式处理，而相关寄存器则会在每次字符串操作之后递减 |

（续）

| 标　志 | 代　号 | 对应的二进制位在寄存器中的位置 | 描　述 |
| --- | --- | --- | --- |
| Overflow（溢出） | OF | 11 | 如果运算结果以补码的形式来表示时所需的二进制位个数超过了该运算所采取的个数，那么该标志就会设置。与 Carry 标志不同的是，此标志是针对带符号的整数而言的，如果运算结果的符号与操作数的符号相反，那么它就会得到设置，比方说，在 8 位环境中，127 + 127 的结果用补码表示需要 9 位（0 1111 1110），如果只看后 8 位（1111 1110），那么由于其中的最高有效位是 1，因此这是个负值（相当于十进制的 −2），于是执行完这次加法之后 OF 标志就会得到设置 |

**编程知识：**表 2-8 中，1、3、5 号二进制位是用 U 来表示的，意思是说，这些二进制位用不到（Unused）或是予以保留（reserved）。在通过 SAHF 指令把 *ah* 的内容保存到标志寄存器时，这些二进制位的值不应当受到影响，3、5 号二进制位始终应该是 0，1 号二进制位始终应该是 1。

表 2-8　可以通过 LAHF/SAHF 指令编辑的标志位及其序号

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| SF | ZF | U | AF | U | PF | U | CF |

### 2.3.2　64 位处理器

尽管前面已经讨论了 64 位处理器的某些特征，但还是有一些重要的细节必须说明。x86_64 指令集是对 x86 指令集（这是一种 32 位指令集）的扩充，因此能够在 32 位环境下执行的操作，同样可以放在 64 位的处理器中执行。由于 x86_64 处理器是 64 位的，因此其数据与地址都可以用 64 个二进制位来表示，然而，当前的 x86_64 处理器只用到了其中的低 48 位，所以说，尽管理论上能够在 $2^{64}$ 字节的地址空间中寻址，但实际上最多只支持 $2^{48}$ 字节的地址空间。不过这已经很大了，因为用 48 个二进制位来表示物理地址空间中的地址意味着 RAM（内存）容量可达 256 TB，这比 32 位处理器所支持的 4GB 要高出很多。除了支持更大的 RAM 地址空间，x86_64 处理器还多提供了 8 个通用的寄存器 R8 ～ R15。

### 2.3.3　指令的执行

要执行指令（例如 ADD）必须遵循一系列步骤，这些步骤合起来构成**指令执行周期**（Instruction Execution Cycle）。开发者可能会觉得 ADD 看上去只是一条指令而已，应该只需要一步就能执行完，但实际上要执行这条指令必须完成很多子任务，或者说历经很多阶段。从最宏观的角度看，可以像图 2-10 这样将执行过程划分成三大阶段，然而当前某些高端的处理器实际上会经过 20 个乃至更多个阶段。因此，开发者虽然只是简单地用 ADD 指令把两个值加起来，但在幕后却有可能要经过 20 个步骤才能走完指令执行周期，从而得出两数之和。




图 2-10　指令执行周期中的三大阶段

指令执行周期一般表示成 Fetch(获取)、Decode(解码) 及 Execute(执行) 这三个大的阶段。在**获取**阶段 CPU 会把将要执行的（也就是 *rip*/*eip* 所指向的）指令取出来。在**解码**阶段 CPU 要判断这是一条什么样的指令，它需要操作哪些数据。为此，它会查看指令码中的二进制位以了解该指令的类型（ADD、SUB，还是 MOV）以及是否包含操作数。如果包含，那还需要把操作数也获取过来。将所有的数据都获取到并加以解码之后 CPU 就可以**执行**这条指令了。如果执行之后产生了结果（例如 ADD 及 SUB 指令），那么 CPU 会把该结果保存到适当的地方（例如寄存器或内存）。

### 2.3.4　指令流水线

当前流行的这些处理器，其指令执行周期的三大阶段其实都包含着许多个小的阶段。设计处理器的人这些年来发现，如果同时执行指令周期中这些不同的阶段，那么处理器的性能可以发挥得更好，例如在解码第一条指令的同时把第二条指令获取出来，等到执行第一条指令的时候则对第二条指令进行解码，依此类推。这种重叠式的设计就叫作**指令流水线**（instruction pipelining，也称为指令管线化）技术，如图 2-11 所示。

用洗衣服的过程做例子可以帮你更好地理解指令流水线技术。如果只有一堆衣服要洗，那么通常的流程是：先把衣服汇集起来然后放入洗衣机里洗，接下来放入干衣机里烘干，最后把衣服放好。问题是如果有两批衣服要洗那应该怎么办？是先把其中一批按照这四个流程处理完然后再处理另外一批，还是在洗第一批衣服的同时就把第二批衣服汇集起来，从而可以在第一批衣服正烘干的时候洗第二批衣服？显然后面一种做法更省时间。这个例子很能说明指令流水线技术的原理。CPU 的获取、解码及执行功能是由不同的组件负责的，其中任何一个组件都无法身兼三种职能，然而这些组件却可以在同一时刻分别完成自己所能处理的工作，从而不会陷入无事可做的状态。这样能够更好地利用时间并提升处理器的效率。

> **提示：**在当前的处理器中，能够完成获取、解码与执行这三种功能的组件都不止一个，这样可以令 CPU 在同一时间段内尽量多处理一些指令。

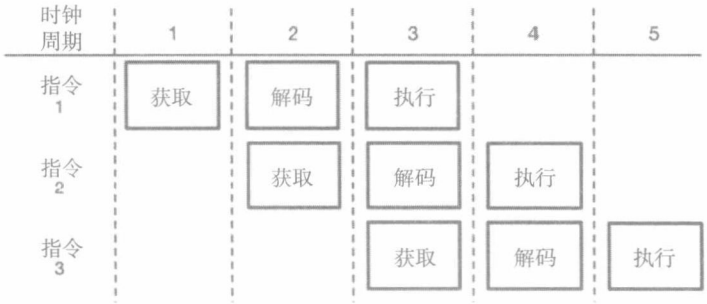



图 2-11　流水线式的指令执行周期

前面说过，指令执行周期可以分成多达 20 个阶段，具备这种规模的指令流水线称为**超流水线**（superpipeline），它使得 CPU 能够同时在多条指令上分别执行这 20 个不同的阶段。CPU 的流水线越多，同一时间段内所能做的工作也越多。假设每个阶段需要 1 个时钟周期，如果用不带流水线技术的处理器来执行这种需要 20 个步骤才能完成的指令，那么执行 10 条指令就至少需要 200 个时钟周期才行，因为必须等前一条指令经历完这 20 个阶段才能开始执行后一条指令。反之，如果像图 2-12 这样用支持流水线技术的 CPU 来执行，那么只需 29 个时钟周期即可，其处理速度是前者的 690%，而且所需执行的指令数越多速度上的优势就越大。

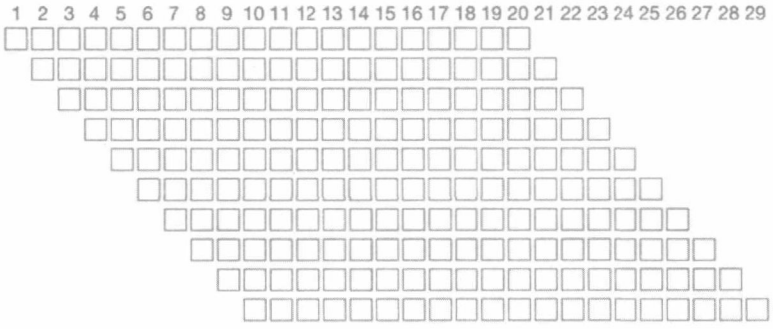



图 2-12 20 个阶段[⊖]的超流水线

## 2.4 输入与输出

目前为止，我们已经讨论了处理器中的某些细节，并且知道了它是怎样与主板中的其他主要部件相通信的。然而还有一个问题是处理器如何与外部设备进行通信。键盘、显示器、网卡等外部设备需要通过 **I/O 模块**（Input/Output Module）与电脑相连，而这些 I/O 模块同时也与系统总线相连，使得外部设备能够与计算机中的其他组件（例如 CPU）通信。除了在处理器与外部设备之间提供缓冲通信（buffered communication）机制之外，I/O 模块还可以完成其他一些重要的操作，例如传输数据、对命令进行解码，以及查询设备状态等。

处理器需要用控制器来精准地安排指令的执行工作，使指令之间不会互相干扰。与之类似，I/O 模块也必须设法在计算机内部的组件与连接到该模块的外部设备之间协调使双方能够有效沟通。前面说过，离 CPU 越远的设备速度越慢，由于外部设备正是计算机环境中离 CPU 最远的设备，因此它们的速度是最慢的。由于速度不同，I/O 模块必须提供数据缓冲区，使 CPU 的工作速度不会因为外部设备而受到影响，同时也令外部设备不会为 CPU 发来的大量数据所淹没。而且不同的外部设备其速度也有所区别（例如用刻录机刻录光盘的速度就与往优盘上复制数据不同），因此每个 I/O 模块都需要考虑与之相连的设备具有怎样的数据传输率，并据此做出调整。

> 📖 **学习指南：** 错误检测与修正算法有很多种，例如校验和、循环冗余校验（Cyclic Redundancy Check，CRC）、汉明码（Hamming Code）以及奇偶校验位（parity bit）等，你可以在网上搜索相关的资料并深入研究。

⊖ 也称为 20 层或 20 级。——译者注

I/O 模块的另一项关键工作是检测错误，也就是确保外部设备所接收到的数据与发送方所发送的数据是一致的。检测错误的办法有很多种，但目标都一样，就是确保数据准确无误。

处理器执行 I/O 操作通常可以采用 4 种方式：程序 I/O（Programmed I/O，PIO）、中断驱动 I/O（Interrupt-driven I/O）、直接内存访问（Direct Memory Access，DMA）以及 I/O 通道（I/O Channel）。在这 4 种方式中，**程序 I/O** 实现起来最简单，然而开销也最大，因此它是效率最低的方案，如图 2-13 所示。处理器在以 PIO 模式执行 I/O 操作时，首先需要向相关的 I/O 模块下达命令并等待回应。I/O 模块负责执行该命令并在执行完毕时把 I/O 状态寄存器中的相关二进制位设置好，以便告知处理器这条命令已经执行完了。处理器会定期检查 I/O 状态寄存器，看 I/O 模块有没有把这项操作执行完。这种定期查询的做法开销很大，在执行耗时较长的 I/O 操作时更是如此，因为处理器必须多次查询才能知道这项操作有没有执行完。

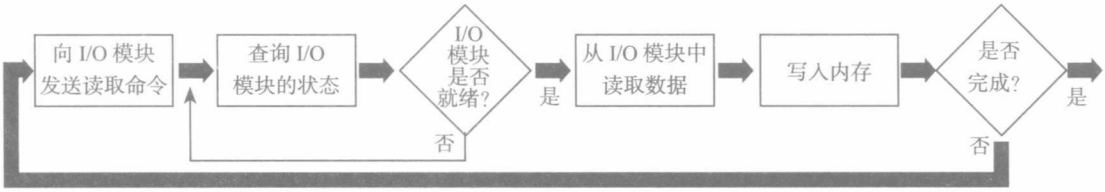



图 2-13　程序 I/O

如果处理器向 I/O 模块发送完命令之后不在那里干等着该命令执行完而是去执行其他操作，那么效率可能会高一些。这就是**中断驱动 I/O** 所采用的理念，如图 2-14 所示。处理器向 I/O 模块下达完命令之后就去执行别的指令了，只有当 I/O 模块发出中断请求时，它才会回过头来继续处理这项操作。中断会使处理器暂停其他的操作，先把早前未处理完的这项 I/O 操作执行完。由于处理器在每个指令周期的末尾都要检测中断，因此 I/O 模块在发出请求中断的消息之后，最多只需等待 1 个完整的指令周期就可以令相关的事务得到 CPU 的处理。I/O 模块在执行命令时几乎用不着处理器来干预，只是每次把数据写入主存，或是从主存中读取数据的时候需要与处理器相配合。

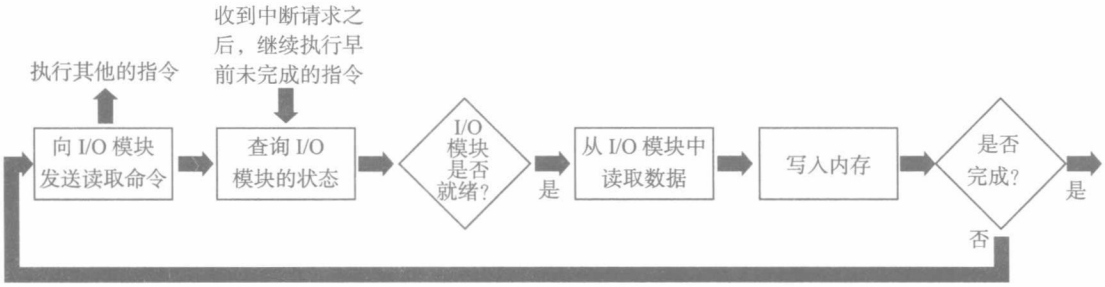



图 2-14　中断驱动 I/O

程序 I/O 与中断驱动 I/O 这两种处理方式均要求处理器必须全程参与，但由于很多 I/O 操作仅仅是在外部设备与 RAM 之间单纯地传输数据，因此这会令处理器把许多时钟周期都浪费在这种简单的任务上。为了避免浪费，我们可以采用图 2-15 所描述的这种方式来处理 I/O，这就是**直接内存访问**（Direct Memory Access，DMA）。该模式要求主板上必须有专门

用来执行 DMA 操作的模块。当处理器需要执行 I/O 操作时并不把信息直接发送到设备的 I/O 模块，而是发给刚才说的这个 DMA 模块。此模块会在某种程度上模仿 CPU 的处理方式，它每次令 CPU 暂停一个周期从而获得 I/O 操作的控制权，以便不受限制地访问系统总线（这种行为叫作周期挪用，cycle stealing）。DMA 模块执行完 I/O 操作之后，向处理器发送中断请求以便通知后者。处理器只需把 I/O 操作中不涉及数据传输的那部分指令执行完即可，这样可以节省很多时钟周期，令计算机得到更为有效的利用。

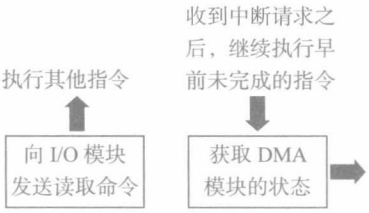



图 2-15 直接内存访问

DMA 模块看上去似乎解决了 CPU 必须花很多时间处理 I/O 操作这一问题，但实际上它并不是个完整的处理器，因此，必须挪用 CPU 的周期才能完成 DMA 任务。由此可见，与理想状况相比，CPU 还是会做一些工作。为了把周期挪用问题也解决掉，我们可以拿真正的处理器配置一个专门用来处理 I/O 操作的模块。这种技术叫作 **I/O 通道**，它使得该模块无须再挪用 CPU 的周期来执行任务。此技术有很多种形式，其中一种是每次只处理一台设备的 I/O 操作，还有一种是同时处理很多台设备的 I/O 操作（这也称作 multiplex[⊖]）。有了 I/O 通道，CPU 就不用处理那么多指令了。它只需要把最开始的那条指令告诉该通道，此后就由这个通道来接管并控制 I/O 模块，令其按照剩下的指令去完成 I/O 操作，而无须再占用 CPU 的周期。

## 2.5 小结

这一章到此结束，大家应该已经了解与处理器和体系结构有关的基础知识了。这些知识涵盖了学习汇编语言所需掌握的几个重要方面，其中包括 CPU、存储器、指令执行，以及 I/O。尽管这一章并没有讨论汇编代码，但要想理解后续各章所讲的汇编代码，就必须先看懂本章的概念。在阅读后面的章节并开始编写汇编程序时可能要经常翻回来看这一章，尤其是要参考其中与存储器及 CPU 寄存器有关的段落。

## 2.6 关键术语

address bus（地址总线）
Arithmetic Logic Unit（ALU，算术逻辑单元）
Big-Endian（大端序）
bus（总线）
cache（缓存）
Central Processing Unit（CPU，中央处理器）
clock cycle（时钟周期）
control bus（控制总线）
Control Unit（CU，控制单元）
CPU Clock（CPU 时钟）
data bus（数据总线）
Decode（解码）
Direct Memory Access（DMA，直接内存访问）
disk（磁盘）

⊖ I/O multiplexing 的中文说法，有 I/O 多路传输、I/O 多工、I/O 多路复用等。——译者注

Dynamic RAM（DRAM，动态随机存取存储器）
Execute（执行）
Fetch（获取；读取）
flag（标志）
frequency（频率）
I/O Channel（I/O 通道）
Input/Output Module（I/O Module，输入 / 输出模块）
Instruction Execution Cycle（指令执行周期）
Interrupt-driven I/O（中断驱动 I/O）
Little-Endian（小端序）
memory（存储器）
motherboard（主板）
multiplier（倍频系数）
pipelining（流水线）
Programmed I/O（PIO，程序 I/O）
register（寄存器）
Static Random Access Memory（SRAM，静态随机存取存储器）
superpipeline（超流水线）
system bus（系统总线）
system clock（系统时钟）

## 2.7　习题

### 填空题

1. 处理器一词还有另一种说法，是由 3 个英文单词的首字母所构成的简称，这就是：_____。
2. _____是一组线路，用来连接主板中的各个组件使其能够相互通信。
3. _____总线会发出信号，使各组件能够知道相关的指令与数据在存储器中的位置。
4. 在进入系统时钟的高电平期时电压会由_____变_____。
5. 存储器离 CPU 越近访问起来就越_____。
6. 如果按照_____的字节序来保存十六进制值 1984，那么它在存储器里就会以 8419 的顺序出现。
7. _____负责指挥 CPU 中的数据流。
8. CPU 的速度由两个因素决定：_____与_____。
9. _____缓存是最慢的 CPU 缓存。
10. *rcx* 寄存器的低 8 位也称作_____寄存器。
11. 32 位的栈指针寄存器叫作_____寄存器。
12. 在指令执行周期中，CPU 会于_____阶段来判定指令 / 操作码所表示的是何种操作。
13. 如果可以把各项操作所要经历的处理阶段相互错开，那么支持_____技术的 CPU 就能够在同一时间执行这些操作。
14. _____使得处理器把当前正在处理的这一系列操作暂时放下转而去处理令一件事。
15. CPU 开销最小的 I/O 处理方式是_____。

### 判断题

1. 主板与 RAM、扩展卡及外部 I/O 设备相连。
2. 把数值移动到 *edx* 寄存器会令 *dx* 寄存器遭到覆写。
3. 如果 *rflags* 寄存器的 SF 位是 1，那么意味着早前那项操作的结果是正值。
4. PIO 是最快的 I/O 处理方式。
5. I/O 通道至少需要两个 CPU 周期才能完成一项 I/O 操作。

## 2.8 作业

### 2.8.1 内存

有这样一个算式：`taxableIncome = salary - exempts - percent401k / 100 * salary`，其中的变量 `salary = 50000, exempts = 7000, percent401k = 4.5`。

在配有 Intel Core i7 处理器的系统中，`taxableIncome` 的值在内存中是怎样表示的？假设该值是从 `0x013A32A8h` 这一地址开始存放的。

### 2.8.2 指令执行

假设某程序有 220 条指令。如果把它放在下列三个处理器中运行，那么分别需要多少个时钟周期才能执行完毕？

- 不支持指令流水线技术的 3 阶段式处理器
- 配有 3 层指令流水线的处理器
- 配有 15 层指令流水线的处理器

### 2.8.3 输入 / 输出（挑战题）

如果在 Linux 系统的 `/dev/` 目录里执行 `ls -la` 命令，那么就会输出一份长清单，其中包含一些特殊文件，用以表示连接到该系统的每一个设备，这些文件均带有重要的信息。请通过上网搜索来研究这份清单看看其中涉及几种 I/O 设备，每种设备都是怎样表示的。

# 第 1 章与第 2 章补充材料
# 与体系结构有关的更多细节

## 程序加载

第 1 章说过，程序是由一个叫作程序加载器的工具来加载的。加载进来之后，CPU（主要是说它的 eip/rip 寄存器）指向程序的入口点，这个入口点可能叫作 main 或 start 等。下面描述启动程序的一般步骤。当用户开启程序（例如用鼠标双击程序的图标）时：

1. 操作系统把文件的大小以及该文件在磁盘中的物理位置等信息获取出来。
2. 操作系统在内存中寻找合适的地点分配空间，并把必要的信息放在描述符表（descriptor table）中（第 10 章会讲解描述符）。
3. 操作系统开始执行程序的第一条指令（也就是位于入口点的那条指令）。这使程序变为进程，并获得由系统所赋予的 ID。
4. 该进程自行运作，而操作系统则会对进程所发出的资源请求予以响应。
5. 进程结束并让出它所占据的内存。

## 提高存储器访问速度的措施

第 2 章说过，从主存（也就是 RAM）中读取数据不如从寄存器里读取快，因为后者更接近 ALU，然而程序不太可能把所有数据都放在寄存器里，它还是需要读取主存中的数据，于是，这些年来就产生了下列措施，旨在用更好的办法来执行指令并访问数据，以提升速度。

- **缓存**，可以缓解内存的访问速度较慢这一问题。如果要获取的指令或数据能在缓存中找到，那么就出现了缓存命中现象；若是不能，则称为缓存未命中。此外，指令可以成批地移入缓存或是从缓存里移出。
- **预取算法**（prefetching algorithm）可以根据计算机访问内存的模式把将来有可能需要的指令与数据填充到缓存及预取缓冲中。当前大多数 CPU 在流水线里都安排有预解码这一环节。
- **集成式内存控制器**（integrated memory controller）技术把内存控制器整合到 CPU 中，使其与 RAM 之间无须再通过前端总线及北桥来通信。
- **多核心 / 多处理器**技术使得系统能够把有待执行的指令分布到这些核心及处理器上。

## 对 CPU 起支持作用的处理器及组件

计算机系统中，有很多处理器及组件负责执行一些特殊的任务以支持 CPU 的工作。下面列出几个值得注意的组件：

- **浮点运算器**（Floating-Point Unit，FPU）。它通常与 CPU 相整合（参见第 6 章）。
- **时钟发生器**：为 CPU 产生时钟信号。
- **可编程中断控制器**（Programmable Interrupt Controller，PIC）：处理由键盘、硬盘及

其他硬件设备所产生的中断（参见第 10 章）。

- **可编程间隔计时器**：每秒钟把系统中断 18.2 次，用以更新系统时间与时钟、控制系统扬声器，并负责刷新主存。
- **图形处理器**（Graphical Processing Unit，GPU）：经过特别设计的处理器，能够高效地处理矩阵及向量并将可视数据显示到屏幕上。

### 指令流水线技术与多单元的处理器技术

第 2 章说过，指令流水线技术可以同时推进很多条指令处理流程从而提升执行速度。CPU 中通常包含许多单元，例如取指单元、解码单元、执行控制单元、寻址单元、内存控制器、乘法器、移位器、ALU、FPU 等。当前的 CPU 一般会在每个处理器及内核中配备多个单元。因此，考虑到硬件环境方面的因素，指令流水线技术可以用多种形式来实现。

**标量**（scalar）处理器每次只处理一份数据（在这里指的是一条指令），因此，执行指令的平均速度大约和时钟的速度一样。

**超标量**（superscalar）处理器可以同时获取多条指令，并将其派发到多个功能单元上分别执行。这样处理指令的速度要大于时钟的速度。

下面举几个多单元处理器的例子。在处理器的发展过程中，产生了很多种这样的处理器，这里所举的只是其中的几种而已。

- 范例 1：PowerPC 970 有两个 ALU，两个 FPU，两个加载 / 存储单元，两个 SIMD 单元，而且配有 10 ～ 16 层流水线，各单元的具体层数有所不同。
- 范例 2：Pentium 3 处理器有 3 个 ALU，Pentium 4 处理器有两个 ALU（然而每一个的时钟频率都是 P3 ALU 的两倍）。这两种处理器都有 1 个 FPU。
- 范例 3：AMD Athlon 处理器有 3 个 ALU、3 个 AGU（Address-Generation Unit，地址生成单元）及 1 个 FPU。
- 范例 4：Haswell 架构的 Intel Core i7 处理器支持 HyperThreading（超线程）技术，这意味着每个内核在逻辑上相当于两个处理器。4 核 i7 处理器的执行单元共有 8 个“端口”，因此每个时钟周期能够执行 8 条微操作（micro-op）。这种处理器有 4 个 ALU。

### 输入 / 输出系统

开发者如果能多了解一些与硬件及设备有关的知识，就可以更好地利用其特性来编写程序并优化程序代码。一般来说，I/O 分为以下三个层级。

- **应用程序层**（Application Level）：高级语言的程序库与 API（Application Programming Interface，应用程序编程接口）。
- **系统层**（System Level）：用汇编指令来执行系统调用。
- **UEFI 层**（UEFI Level）：UEFI（Unified Extensible Firmware Interface，统一可扩展固件接口）是一种底层软件，能够直接与硬件通信并提供启动及运行期服务。UEFI 是新式的解决方案，用来替代 BIOS（Basic Input/Output System，基本输入输出系统），它比后者更加先进。

**设备驱动程序**（Device Driver）包含一些例程，使得操作系统能够直接与硬件通信。

下面举一个例子来演示信息是怎样在整个 I/O 体系的各个层级之间交换的。

1. C++ 程序里有一条用高级语言写成的语句，它调用库函数以便向标准输出端写入字符串，这条语句可以是：`cout << “This is string” ;`。
2. 库函数调用系统例程并把相关的参数（例如字符串的地址与大小）传过去。
3. 系统在执行该例程的时候反复调用设备驱动程序或 UEFI 子例程，并把每个字符的 ASCII/Unicode 码及其颜色传过去。
   a. 设备驱动程序或 UEFI 子例程收到字符，将其与字体对应起来并把它发送给与视频控制器相连的硬件端口。
   b. 视频控制器通过定时的硬件信号把该字符打印到屏幕上。在此过程中，它要控制每个像素应该怎样显示。
   c. 系统调用另一个设备驱动程序或 UEFI 子例程将光标推进到下一个位置。

汇编语言很强大，它能够灵活地游走于各个层级。开发者可以在应用程序层调用库函数以执行控制台式的 I/O 或基于文件的 I/O。也可以在操作系统层中通过调用内核来完成这些 I/O 操作。此外，还能在 UEFI 层 / 设备驱动层中调用相关的函数，以控制与具体设备有关的特性。开发者需要在效率、结果与可移植性等因素之间权衡，以决定自己究竟应该在哪个层面上执行 I/O 操作。本书第 10 章将会讨论这个话题，还会讲解应用程序层面与系统层面的调用。

第 3 章

Assembly Programming and Computer Architecture for Software Engineers

# 汇编语言及其语法的基础知识

## 目标

- 了解各种汇编语法之间的区别
- 了解汇编代码的各个段落并知晓其用途
- 用正确的语法来定义数据
- 创建能够运行的汇编程序

## 网络资源

- https://sourceware.org/binutils/docs/as/（GAS 参考资料）
- https://msdn.microsoft.com/en-us/library/afzk3475.aspx（MASM 参考资料）
- http://www.nasm.us/doc/nasmdoc0.html（NASM 参考资料）

## 3.1 简介

这一章讲解 x86 及 x86_64 汇编语言。由于软件工程师通常会根据所要解决的问题来换用不同的操作系统，因此，本章与后续各章会同时采用三种较为流行的汇编器来进行讲解，也就是 GAS、MASM 与 NASM。这样写出来的范例能够运行在 Mac、Windows 与 Linux 这三种常见的操作系统上，使你可以通过与这些系统分别对应的三种开发环境（也就是 Apple Xcode、Microsoft Visual Studio，以及 Linux 系统的命令行界面）来学习汇编语言。

笔者会仔细地标出各种汇编器在语法上的区别。如果同一个意思必须按照三种不同的方式撰写才能分别适用于这三种汇编器，那么笔者就会以表格的形式对比这三种写法，并在表头标明各种写法所针对的汇编器。有的时候，笔者所举的范例代码只是为了解释某个概念，而不是打算讨论三种汇编器在语法上的区别，这些范例代码不会以表格的形式印出，而且也不会说明它针对的是哪一种汇编器。表 3-1 列出了各种汇编器所采用的汇编语法。

表 3-1 各种汇编器所遵循的语法规则

| 汇 编 器 | 语 法 | 开发环境 |
|---|---|---|
| GAS | AT&T | Apple Xcode |
| MASM | Intel | Microsoft Visual Studio |
| NASM | Intel | Linux 系统的命令行界面 |



**提示：** Xcode 并没有使用标准的 GNU Assembler（GAS），而是以 Clang 为前端，以 LLVM 为后端，自己整合了一种汇编器。这种 Clang/LLVM 汇编器与标准的 GAS 之间稍有区别，不过，由于 Clang 的目标是成为 GNU toolchain（GNU 工具链）的替代品，因此，在 Xcode 中写的汇编代码同样可以用当前版本的 GAS 来编译（笔者撰写本书时，GAS 的版本是 2.27）。请不要使用 macOS 自带的 GAS 版本（笔者撰写本书时，

版本是 1.38）。这本书里的代码都已经用 macOS 系统中的 Xcode 测试过了，而且也能够在 Linux 系统中用 2.x 版本的 GAS 来编译，也就是说，它同时兼容 macOS 及 Linux 系统。为了保持一致，笔者把这两种情况下所用的汇编器统称为 GAS。

读完这一章，你就能熟悉 x86 与 x86_64 汇编的基本结构及语法了。第 4 章及后续各章都要求你必须先很好地理解这一章所讲的基础汇编语法。

## 3.2　基本元素

与高级语言不同，汇编语言是一种底层语言，它的每一行代码只执行一项操作。要想写出汇编代码，必须了解与计算机体系结构有关的一些细节，例如 CPU 寄存器、标志位，以及浮点运算功能等。对于编程新手来说，通过这些底层细节编写汇编代码显得相当枯燥，而且即便是有经验的程序员有时也会产生这种感觉。不过，掌握底层细节能够令开发者写出控制能力更强、效率更高的代码，这种优势是高级语言所无法比拟的。等到熟练掌握了汇编语言，你可能就会发现自己能够写出设计上更加合理的高级语言代码。

### 3.2.1　汇编代码的五大支柱

学习汇编语言必须了解它的五大支柱，也就是：保留字（reserved word）、标识符（identifier）、命令（directive，也称为指示或伪操作）、区段（section 或 segment，简称段）以及指令（instruction）。本章后面的各节以及后续各章都要求你必须先清楚地理解这五个要素。现在，我们就通过一个简单的汇编程序帮助大家认清这 5 个概念。程序 3-1 首先定义一个 32 位的变量，然后将两个数字相加，最后把运算结果保存到该变量中。

**程序 3-1　汇编语言范例代码**

GAS

```
.data
sum: .long 0

.text
.globl _main
_main:
movl $25, %eax
movl $50, %ebx
addl %ebx, %eax
movl %eax, sum

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO,
dwExitCode:DWORD

.data
sum DWORD 0

.code
_main PROC
mov eax, 25
mov ebx, 50
add eax, ebx
mov sum, eax

INVOKE ExitProcess, 0
_main ENDP
END
```

NASM

```
SECTION .data
sum: DD 0

SECTION .text
global _main
_main:
mov eax, 25
mov ebx, 50
add eax, ebx
mov DWORD [sum], eax

mov eax, 1
mov ebx, 0
int 80h
```

很多编程语言里都有**保留字**，它是一种具备特定用途的字词。例如 MOV 就是个保留字，它代表一条特定的指令，即 MOV 指令。你不能把它当成变量名来用，也不能做其他用途。汇编语言的保留字不区分大小写。指令（例如 MOV）、命令（例如 PROC）、寄存器（例如 *eax*）、属性（例如可以当作 .MODEL 命令参数值的 FLAT）等，都是保留字。

表 3-2 用粗体标出了代码中的保留字。你会发现程序中有很多地方都用到了保留字。其中有一些地方是指令，还有一些则是命令或代码段。

表 3-2 范例代码中的保留字

| GAS | MASM | NASM |
|---|---|---|
| **.data**<br>sum: **.long** 0<br><br>**.text**<br>**.globl** _main<br>_main:<br>**movl** $25, **%eax**<br>**movl** $50, **%ebx**<br>**addl %ebx, %eax**<br>**movl %eax,** sum<br><br>**pushl** $0<br>**subl** $4, **%esp**<br>**movl** $1, **%eax**<br>**int** $0x80<br>**.end** | **.386**<br>**.MODEL FLAT, stdcall**<br>**.STACK** 4096<br>ExitProcess **PROTO,**<br>dwExitCode:**DWORD**<br><br>**.data**<br>sum **DWORD** 0<br><br>**.code**<br>_main **PROC**<br>**mov eax,** 25<br>**mov ebx,** 50<br>**add eax, ebx**<br>**mov** sum, **eax**<br><br>**INVOKE** ExitProcess, 0<br>_main **ENDP**<br>**END** | **SECTION .data**<br>sum: **DD** 0<br><br>**SECTION .text**<br>**global** _main<br>_main:<br>**mov eax,** 25<br>**mov ebx,** 50<br>**add eax, ebx**<br>**mov DWORD** [sum], **eax**<br><br>**mov eax,** 1<br>**mov ebx,** 0<br>**int** 80h |

**标识符**是由程序员所定义的名称，用来表示变量、常量及过程等事物，它最多可以包含 247 个字符。第一个字符不能是数字，且必须从英文字母（大写的 A 至 Z 及小写的 a 至 z）、下划线（_）、问号（?）、at 符号（@）及美元符号（$）这五种中选择（笔者不建议你选后三种），其后的字符则可以使用数字。笔者推荐你采用下面这几种写法来给标识符起名。

- userInputValue
- sum_of_values
- weight1

表 3-3 用粗体标出了范例代码中的标识符。

表 3-3 范例代码中的标识符

| GAS | MASM | NASM |
|---|---|---|
| .data<br>**sum:** .long 0<br><br>.text<br>.globl **_main**<br>**_main:**<br>movl $25, %eax<br>movl $50, %ebx<br>addl %ebx, %eax<br>movl %eax, **sum**<br><br>pushl $0<br>subl $4, %esp<br>movl $1, %eax<br>int $0x80<br>.end | .386<br>.MODEL FLAT, stdcall<br>.STACK 4096<br>**ExitProcess** PROTO,<br>**dwExitCode:**DWORD<br><br>.data<br>**sum** DWORD 0<br><br>.code<br>**_main** PROC<br>mov eax, 25<br>mov ebx, 50<br>add eax, ebx<br>mov **sum**, eax<br><br>INVOKE **ExitProcess**, 0<br>**_main** ENDP<br>END | **SECTION .data**<br>**sum:** DD 0<br><br>SECTION .text<br>global **_main**<br>**_main:**<br>mov eax, 25<br>mov ebx, 50<br>add eax, ebx<br>mov DWORD [**sum**], eax<br><br>mov eax, 1<br>mov ebx, 0<br>int 80h |

命令是与指令集无关的一些操作，可以指挥汇编器去做某件事，例如定义变量、指明内存段等。比方说，下面这行代码里的 DWORD 就是个命令，它告诉汇编器要在内存里保留 32 个二进制位，或者说保留一个双字（doubleword）。整行代码的意思是：用 42 这个值来初始化一个双字，并将其所在的这块内存起名叫作 answer，以供后续使用。

```
answer DWORD 42          ; DWORD is the directive
```

表 3-4 用粗体标出了范例代码中的命令。

表 3-4　范例代码中的命令

| GAS | MASM | NASM |
|---|---|---|
| **.data**<br>sum: **.long** 0<br><br>**.text**<br>**.globl** _main<br>_main:<br>movl $25, %eax<br>movl $50, %ebx<br>addl %ebx, %eax<br>movl %eax, sum<br><br>pushl $0<br>subl $4, %esp<br>movl $1, %eax<br>int $0x80<br>**.end** | **.386**<br>**.MODEL** FLAT, stdcall<br>**.STACK** 4096<br>ExitProcess **PROTO**,<br>dwExitCode:**DWORD**<br><br>**.data**<br>sum **DWORD** 0<br><br>**.code**<br>_main **PROC**<br>mov eax, 25<br>mov ebx, 50<br>add eax, ebx<br>mov sum, eax<br><br>**INVOKE** ExitProcess, 0<br>_main **ENDP**<br>**END** | **SECTION .data**<br>sum: **DD** 0<br><br>**SECTION .text**<br>**global** _main<br>_main:<br>mov eax, 25<br>mov ebx, 50<br>add eax, ebx<br>mov **DWORD** [sum], eax<br><br>mov eax, 1<br>mov ebx, 0<br>int 80h |

现在我们需要指出 MASM 所用的命令与另外两个汇编器之间的区别。你可能已经发现了：MASM 版本的程序在数据段之前多写了几行代码，这是另外两个版本所没有的。对于 32 位程序来说，这几行代码是必须编写的，因为它们会对程序环境做出必要的设置。表 3-5 列出了 MASM 版本特有的命令并加以解释。

表 3-5　用 MASM 汇编器撰写 32 位程序时专用的命令

| 命　令 | 描　述 |
|---|---|
| .386 | 启用 80386 式的处理器指令并禁用新指令。还有一些类似的命令也是用来启用某一套指令的，例如 .486、.586、.686、.MMX 及 .XMM |
| .MODEL | 设置内存模型。对于 32 位程序来说，该命令的第一个参数只能取 FLAT 这个值（用来表示 protected mode，保护模式）。此外，该命令还可以带有第二个参数，用来设置函数调用约定（function-calling convention），这会在第 6 章讲解 |
| .STACK | 为程序设置栈内存段的大小。必须在使用了 .MODEL 指令的情况下才能使用该指令。它所采用的默认值是 1024，但笔者建议你将其设为 4096，以使其与 32 位 Windows 的内存页（memory page）一样大 |

用 MASM 编写 64 位程序的时候不需要使用表 3-5 中的命令，因此代码与 GAS 及 NASM 版本的代码一样，直接以数据段开头。

**程序段**（program section 或 program segment）是用相关命令所标出的特殊段落。汇编器预先定义了几种这样的段落。表 3-6 列出了撰写汇编代码时常用的几种程序段。

表 3-6 各种汇编器所支持的程序段

| 命令 | | | 描述 |
|---|---|---|---|
| GAS | MASM | NASM | |
| .bss | .data | SECTION .bss | 未初始化的变量 |
| .data | | SECTION .data | 已初始化的变量 |
| .text | .code | SECTION .text | 可执行的代码 / 指令 |

表 3-7 用粗体标出了范例代码中与程序段有关的命令。

表 3-7 范例代码中与程序段有关的命令

| GAS | MASM | NASM |
|---|---|---|
| **.data**<br>sum: .long 0<br><br>**.text**<br>.globl _main<br>_main:<br>movl $25, %eax<br>movl $50, %ebx<br>addl %ebx, %eax<br>movl %eax, sum<br><br>pushl $0<br>subl $4, %esp<br>movl $1, %eax<br>int $0x80<br>.end | .386<br>.MODEL FLAT, stdcall<br>.STACK 4096<br>ExitProcess PROTO,<br>dwExitCode:DWORD<br><br>**.data**<br>sum DWORD 0<br><br>**.code**<br>_main PROC<br>mov eax, 25<br>mov ebx, 50<br>add eax, ebx<br>mov sum, eax<br><br>INVOKE ExitProcess, 0<br>_main ENDP<br>END | **SECTION .data**<br>sum: DD 0<br><br>**SECTION .text**<br>global _main<br>_main:<br>mov eax, 25<br>mov ebx, 50<br>add eax, ebx<br>mov DWORD [sum], eax<br><br>mov eax, 1<br>mov ebx, 0<br>int 80h |

后面有一节会讲到如何定义数据，那个时候再来详细解释这些程序段的用法以及它们所含的信息。

**指令**是程序中的可执行语句，我们将在第 4 章开始详细讲解，这里先简单地介绍一下。指令由两个基本部分组成，其语法如下：

```
mnemonic [operands]
```

其中 mnemonic（助记符）是指令的名称，开发者用它来指代某个架构所支持的一套指令集里的一条指令。助记符通常是个缩写形式或首字母缩略形式的词，实际上也可以认为是数字形式的操作码所对应的英语写法（参见第 1 章的范例 1-2）。表 3-8 用粗体标出了范例程序中的指令。

表 3-8 范例程序中的指令

| GAS | MASM | NASM |
|---|---|---|
| .data<br>sum: .long 0<br><br>.text<br>.globl _main<br>_main:<br>**movl** $25, %eax<br>**movl** $50, %ebx<br>**addl** %ebx, %eax | .386<br>.MODEL FLAT, stdcall<br>.STACK 4096<br>ExitProcess PROTO,<br>dwExitCode:DWORD<br><br>.data<br>sum DWORD 0 | SECTION .data<br>sum: DD 0<br><br>SECTION .text<br>global _main<br>_main:<br>**mov** eax, 25<br>**mov** ebx, 50<br>**add** eax, ebx |

（续）

| GAS | MASM | NASM |
| --- | --- | --- |
| **movl** %eax, sum<br><br>**pushl** $0<br>**subl** $4, %esp<br>**movl** $1, %eax<br>**int** $0x80<br>.end | .code<br>_main PROC<br>**mov** eax, 25<br>**mov** ebx, 50<br>**add** eax, ebx<br>**mov** sum, eax<br><br>INVOKE ExitProcess, 0<br>_main ENDP<br>END | **mov** DWORD [sum], eax<br><br>**mov** eax, 1<br>**mov** ebx, 0<br>**int** 80h |

有些指令不需要操作数，有些则需要一个、两个或三个操作数。下面这四行代码分别演示了这四种情况。

```
stc                 ; no operands, sets the carry flag
inc eax             ; 1 operand, increments eax by 1
mov eax, 5          ; 2 operands, moves literal value 5 into eax
imul eax, ebx, 5    ; 3 operands, multiplies literal value 5 and the
                    ; value in ebx and stores the result in eax
```

**学习指南：**现在你应该参照附录 B 与附录 D 搭建一套能够正常运作的编程环境。请在该环境中编写、执行并调试本章的范例程序 3-1，以便初步了解汇编程序的开发过程。在继续往下读之前，首先要保证这个范例程序能够在你的电脑上正常编译并运行，而且你要学会通过断点来暂停程序并逐行地调试该程序，以观察寄存器中的值是怎样在执行过程中发生变化的。做到了这一点之后，你就可以继续往下读了，阅读过程中，你可以根据学到的新知识来修改范例程序，使自己更好地理解这些概念。

### 3.2.2　字面量

**字面量**（literal value，字面值）也叫作**立即值**（immediate，立即数），用来表示那些由开发者明确指定的值，例如整数、实数、字符、字符串等。比方说，下面这行代码会把 5 这个字面值与 32 位寄存器 *eax* 的当前值相加，并将结果放入该寄存器。

```
add eax, 5
```

**编程知识：**立即值用在那种开发者已经提前知道指令里该用什么值的场合。很多程序都依赖于用户所输入的数据，但除此之外，可能还会用到一些固定的值。比方说，要编写一个简单的程序把体重从磅[⊖]换算为公斤。这个程序一方面要从用户那里获得以磅为单位的体重数据，另一方面则要将该数据与换算系数相乘（也就是乘以 0.45）将其转为以公斤为单位的结果。该程序所用到的这个换算系数就可以用字面量来表示。

**整数字面量**（integer literal）一般写为十进制（或者说以 10 为底的进制）的整数。在有些情况下也可以（甚至是必须）采用其他进制来写。MASM 及 NASM 允许开发者在字面量

⊖ 1 磅≈0.45 公斤。

之后写上一个表示基数的字符，以指出这个字面量所采用的进制。表 3-9 列出了这些字符。

表 3-9 MASM/NASM 所支持的基数字符，用来指出整数所采用的进制

| 基 数 | 进 制 |
| --- | --- |
| b | Binary（二进制，以 2 为底） |
| d | Decimal（十进制，以 10 为底） |
| h | Hexadecimal（十六进制，以 16 为底） |
| q，o | Octal（八进制，以 8 为底） |

下面这个例子演示了怎样用 MASM 及 NASM 所支持的这几种基数字符来表示十进制的 31。

MASM/NASM

```
00011111b   ; b is the radix character for binary
31          ; decimal values do not need radix characters
31d         ; but you can specify d for decimal
1Fh         ; h is the radix character for hexadecimal
37o         ; o is the radix character for octal
```

有一种情况要注意，十六进制数的数位中可能会出现 A 至 F 这几种英文字符。如果某个十六进制数（例如内存地址）是以英文字符开头的，那么汇编器就会把它当成标识符看待，为了令其能够正确地将该值解读为字面量，你应该在前面添一个 0：

MASM/NASM

```
0FFFF0342h   ; the actual value is FFFF0342 in hexadecimal
```

GAS 表示字面量的方式与 MASM 及 NASM 不同，而且要依照字面量所在的程序段来决定（程序段是汇编代码的五大支柱之一，参见 3.2.1 节）。.data 段里的字面量必须加上表示基数的前缀字符，用以指明字面量所采用的进制，而 .text 段里的字面量除了要加这种前缀，还得在其左侧多写一个美元符号（$），如表 3-10 所示。十进制数例外，它不需要添加表示基数的前缀字符。

表 3-10 在 GAS 代码里表示字面量时所要添加的前缀与基数字符

| 适用于 .data 段的前缀 | 适用于 .text 段的前缀 | 进 制 |
| --- | --- | --- |
| 0b | $0b | 二进制 |
| （不需要） | $ | 十进制 |
| 0x | $0x | 十六进制 |
| 0 | $0 | 八进制 |

下面这个例子演示了怎样在 .data 段及 .text 段中用不同的进制来表示十进制值 31。

GAS

```
.data                   .text
0b00011111              $0b00011111
31                      $31
0x1F                    $0x1F
037                     $037
```

**实数**（real number）要比整数复杂得多，我们留到第 8 章再讨论。

**字符字面量**（character literal）是由单个字符所构成的值，与整数字面量一样，也用来表示那种开发者在编写程序代码的时候就已经很清楚的值。MASM 与 NASM 中的字符字面

量可以用一对单引号括起来，也可以用一对双引号括起来，这两种写法是等效的。

MASM/NASM

```
"A"
'A'
```

GAS 里的字符字面量的写法要根据所在的程序段来定。.data 段里的字面量用一对单引号括起来，.text 段里的字面量还得在前面加上美元符号（$）。

GAS

```
.data
'A'
```

```
.text
$'A'
```



**提示：第 1 章说过，字符在存储器中是以整数形式的 ASCII 编码来表示的。例如‘A’这个字符在存储器中会表示成十进制的 65（也就是十六进制的 41）。**

**字符串字面量**（string literal）是由多个字符字面量所组成的，通常用来表示单词或短语。MASM 与 NASM 的字符串与单个字符类似，也可以用一对单引号或双引号括起来，但是 GAS 则要求必须使用双引号。如果字符串的内容本身就有引号，那么对于 MASM 及 NASM 来说，你必须用另外一种引号把这个字符串括起来，而对于 GAS 来说则必须使用反斜线对字符串里的引号做转义处理，否则该字符串就会提前在这里结束。

GAS

```
"This string \"contains\" double quotes!"
```

MASM/NASM

```
"I don't understand contractions."                ; strings that have one
'"Good job," said the father to his son.'         ; type of quotes on the
                                                  ; outside and a different
                                                  ; type on the inside
```

字符串通常保存成字节数组，其中的每个字节都与字符串中处在该位置上的字符相对应，字节的内容就是字符的 ASCII 码。下面这个例子演示了字符串字面量 `Daisy, daisy` 是怎样以 ASCII 码的形式在内存中表示的。数组里的每个元素的取值就是字符串中的对应字符所具备的 ASCII 码。

| 字符串中的字符 | D | a | i | s | y | , |  | d | a | i | s | y |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 该字符的十进制 ASCII 码 | 68 | 97 | 105 | 115 | 121 | 44 | 32 | 100 | 97 | 105 | 115 | 121 |

### 3.2.3　标签与注释

**标签**（label）可以用来划分代码，以表达某种与编程或设计有关的想法。它不仅可以令代码读起来更加清晰，而且有的时候还能够帮助开发者实现跳转或循环等功能，使得程序可以跳到标签所标识的这个地方。标签的写法是在标识符后面跟一个冒号：

```
identifier:
```

标签可以单独占据一行也可以与指令写在同一行。GAS 与 NASM 的标签用在 .text 段里

的标签区分大小写，MASM 的标签写在 .code 段里则不区分大小写。

```
userLoop:
          inc counter

otherLoop: inc counter2
```

第 5 章会讲解怎样用标签实现循环。

注释是每个程序中的重要组成部分，开发者可以通过它来解释自己为什么要这样写代码，并描述代码是如何运作的。至于代码做了些什么则无须通过注释来说明，因为代码本身通常就已经体现出了这一点。由于汇编语言的代码有时会通过一些原始的指令来实现相当抽象的任务，因此，尤其需要通过注释帮助阅读代码的人更好地理解这些代码。

汇编语言的注释可以分成两种：**单行注释**与**多行注释**。GAS 的单行注释以井号（#）开头，MASM 及 NASM 的单行注释以分号（;）开头。注释可以单独占据一行也可以写在某行代码的后面。

GAS

```
# Moves the counter value into eax
movl counter, %eax
```

MASM/NASM

```
mov eax, counter ; Moves the counter value into eax
```

多行注释只能在 GAS 与 MASM 中用。GAS 的多行注释与 C 语言风格的多行注释一样，也是用 /* 开头，用 */ 结尾，而且可以出现在代码中的任意位置。MASM 的多行注释必须与其他代码分别写在不同的行里，而且要由四部分组成：COMMENT、起始字符、注释文本、终止字符。起始字符与终止字符必须采用同一种字符，而且要注意不能与注释文本中的任何一个字符相同（否则注释就会提前结束）。MASM 的多行注释习惯上采用感叹号（!）作为起、止字符。

GAS

```
mov $10, %eax       /* This is a multi-line comment that is
                    partially on the same line as
                    an instruction and partially on
                    separate lines */
```

MAS

```
COMMENT !
     This is the section of the code where
     employee salaries are calculated. Note
     how the exclamation point is not in the
     text of the comment.
!
```

## 3.3 定义数据

汇编语言的数据类型根据数据的大小来确定（例如 8 位、16 位、32 位），而不像高级语言那样根据数据的内容来确定（例如整数、双精度浮点数、字符串）。无论数据是什么内容，你都应该从一套默认的数据类型中选出一种来定义这份数据。表 3-11 列出了各种汇编器所

支持的数据类型。

表 3-11　以汇编器所支持的默认数据类型来定义数据时所需使用的命令

| 命令 | | | 描述 |
|---|---|---|---|
| GAS | MASM | NASM | |
| .byte, .ascii | DB, BYTE | DB | 1 字节（8 位）整数 |
| | SBYTE | | 1 字节（8 位）带符号整数 |
| .word | DW, WORD | DW | 2 字节（16 位）整数 |
| | SWORD | | 2 字节（16 位）带符号整数 |
| .long | DD, DWORD | DD | 4 字节（32 位）整数 |
| | SDWORD | | 4 字节（32 位）带符号整数 |
| .quad | DQ, QWORD | DQ | 8 字节（64 位）整数 |
| | DT, TBYTE | DT | 10 字节（80 位）整数 |
| .octa | | | 16 字节（128 位）整数 |

从这张表格来看，MASM 支持的数据类型似乎比 GAS 及 NASM 更多，但其实并不能这样比较，因为 MASM 在执行很多操作的时候都会根据上下文推测数据的类型，而 GAS 与 NASM 则不会做出这样的假设。同一份数据在这三种汇编器里都是可以正常使用的，只不过后两者要求你必须明确说出指令所操作的是什么类型（或者多大）的数据。

**编程知识：**变量的大小一定要选对。开发者必须要考虑到变量的取值范围，从而采用尺寸合适的数据类型来定义该变量，以避免语义错误。这种错误属于逻辑上的错误，尽管它不会导致程序无法通过汇编或令程序在执行过程中崩溃，但产生的结果却是不正确的。比方说，如果你把用户输入的两个值保存到 8 位的变量中并对其相乘，那么就应该考虑到运算结果有可能超过 8 位，因此，要想正确地保存运算结果至少需要 16 个二进制位才行。如果保存运算结果的变量太小，就有可能令一部分数据丢失进而产生错误的结果。

**变量**采用下列语法来定义。

GAS/NASM

```
[标识符:] 命令 初始化器 [,初始化器]...
```

MASM

```
[标识符] 命令 初始化器 [,初始化器]...
```

无论采用哪种汇编器，.data 段里定义的变量都必须予以初始化，也就是必须具备初始值。

GAS

```
val1:       .byte 17
valArray:   .long 0xFFFFFF, 0xFFFFFE, 0xFFFFD
```

MASM

```
charInput   BYTE 'A'
myArray     DWORD 41h, 75, 0C4h, 01010101b
```

NASM

```
counter:    DB 0
wageArray:  DD 75, 100, 125
```

请注意，上述三个例子中的第二条语句都带有多个用逗号隔开的初始化器。这种一个标识符后面带有多个初始化器的写法可以用来创建**数组**（array），也就是创建一系列大小相同的值。这些值都可以通过数组的名称加以引用，该名称本身指的是序列中的第一个值（或序列中的第一个值在内存中的位置）。以 MASM 版本的代码为例，它并没有采用 4 个不同的标识符分别定义 4 个双字值，而是创建了名为 myArray 的数组，令其成为由 4 个双字值（4 个 32 位值）构成的序列。充当数组名称的标识符其实是指向数组中首个值的引用，它表示的是该值在内存中的位置。第 4 章会讲解怎样访问数组中的值。

有的时候，程序中需要定义**未初始化的变量**。这种变量不具备初始值，它主要是为了在内存中预留一定的空间以供开发者使用。对于这种变量，不同的汇编器所采用的定义方式之间有很大的区别。GAS 与 NASM 都要求 .data 段里的变量必须用明确的值来初始化，而 MASM 则允许开发者在该段中采用问号（?）充当变量的初始化器来定义未初始化的变量。

MASM

```
.data
num DWORD  6                ; defines an initialized identifier
sum SDWORD ?                ; defines an uninitialized identifier
myArray   BYTE 10 DUP (1)   ; defines an array of initialized bytes
myUArray  BYTE 10 DUP (?)   ; defines an array of uninitialized bytes
```

MASM 的初始化器可以用 DUP 括起来，以便反复创建大小与内容均相同的多个值。凡是有效的初始化器都可以用 DUP 命令加以重复，这也包括问号。在刚才那段代码中，myArray 是由 10 个字节所构成的数组，其中每个字节的初始值都是 1。myUArray 也是由 10 个字节所构成的数组，它的每一个字节都用 ? 加以初始化（或者说，它的每一个字节都不设定明确的初始值）。

GAS 与 NASM 要求未初始化的变量必须创建在 .bss 段中（bss 的意思是 Block Started by Symbol，以符号开始的块），而且要用特定的命令来创建。表 3-12 列出了这些命令。

**表 3-12 创建各种数据类型的未初始化变量时所需使用的命令**

| 命令 | | 描述 |
|---|---|---|
| GAS | NASM | |
| .lcomm | RESB | 保留 1 个字节（8 位） |
| | RESW | 保留 1 个字（16 位） |
| | RESD | 保留 1 个双字（32 位） |
| | RESQ | 保留 1 个四字（64 位） |
| | REST | 保留 1 个十字节（80 位） |

下面这两段代码演示了怎样用 GAS 与 NASM 来定义未初始化的变量。请注意，GAS 定义这种变量所用的语法是 [directive identifier, reserved_bytes]（[命

令 标识符,预留的字节数]),这与定义带有明确初始值的变量时所用的 [identifier: directive initializer]([标识符:命令 初始化器])有所不同。这两种汇编器都能够通过相应的命令,以单个变量或数组的形式来预留各种数据类型的未初始化空间。例如,NASM 版本的代码就预留了 64 个字节的空间并将其命名为 buffer,这块空间可以当成包含 64 个字节型数值的数组来使用。

GAS

```
.bss
.lcomm sum, 2          # Reserve 2 bytes
.lcomm answer, 4       # Reserves 4 bytes
```

NASM

```
SECTION .bss
memAddr: RESD 1        ; Reserves 1 DWORD (4 bytes)
buffer:  RESB 64       ; Reserves 64 bytes
```

前面说过,**字符串**是以 BYTE(字节)数组的方式存储的。字符串必须以 null 结尾,也就是说,其最后一个字节必须是值为 0 的 ASCII 码。不同的汇编器采用不同的方式来定义这种字符串。GAS 是在字符串末尾写上 \0,表示该字符串以 null 结尾,而 MASM 与 NASM 则是直接用字面量 0 来设置最后一个字节。此外,GAS 还可以用 .ASCIZ 命令给字符串末尾自动添加一个值为 0 的字节。

一个涉及字符串的重要问题是换行。不同的汇编器采用不同的方式换行。GAS 用转义符 \n 来插入换行符号(line-feed character,简称 LF)。MASM 用十六进制码 0Dh 与 0Ah 来表示 CR/LF 这两个符号并以此实现换行(CR 的意思是 carriage-return,回车)。NASM 则只用 0Ah 这一个十六进制码(也就是 LF)表示换行。

GAS

```
# The \n in the middle of the string adds a newline
motd: .ascii "The only way to win\nis not to play\0"
motd: .asciz "The only way to win\nis not to play"
```

MASM

```
; motd contains a single-line string
motd  BYTE "Welcome to Earth...",0

; motd2 contains a multi-line string with a newline at the end
motd2 BYTE "Thank you for using our system.",0Dh,0Ah
      BYTE "All of your activity will be monitored"
      BYTE "by our system administrators",0Dh,0Ah,0
```

NASM

```
; The 0Ah in the middle of the string adds a newline
motd: DB "How about a nice",0Ah,"game of chess",0
```

**符号常量**(symbolic constant)可以在 MASM 版本的汇编代码里取代某些变量,用以表示程序执行期间绝对不会变化的值。x86 的符号常量能够表示 32 位整数,x86_64 的符号常量能够表示 64 位整数,常量所表示的都是基于整数的数据。你可以用等号(=)定义这样的常量:

MASM

```
标识符 = 值
```

符号常量有一些优点。比方说，如果程序里需要多次用到某个值（例如雇员的时薪），那么专门用一个标识符来表示这个值要比每次用到的时候都写一遍更加合理，这是因为：(1) 可以给这个值起个有意义的名字，例如 HOURLY_WAGE，令代码更容易阅读；(2) 由于该值只需要在定义这个标识符的时候书写一次，因此，以后如果要加薪或减薪，也只需在这行代码中修改即可而不用把提到该值的每一个地方全都修改一遍。此外，MASM 里的符号常量还有个好处就是不占内存，因为 MASM 在对代码做汇编的时候，会把所有出现符号常量的地方都改成该常量所对应的实际值。

MASM

```
.data
HOURLY_WAGE = 75
.code
mov eax, HOURLY_WAGE
```

汇编的过程中上述代码会自动加以转换，所有出现符号常量的地方都会自动变成该常量所表示的字面量。

```
mov eax, 75    ; MASM automatically replaces the symbol with the value
```

采用等号来定义符号常量的写法只适用于 MASM 汇编器。此外，MASM 的符号常量还可以用来表示字符串值。

所有的汇编器都可以把表达式的值表示成符号常量。符号必须用 EQU 命令来定义。与定义变量时的要求不同，用 EQU 所创建的符号既可以出现在数据段也可以出现在代码段（对于 GAS 汇编器，它可以出现在 .data 与 .text 段，对于 MASM 汇编器，它可以出现在 .data 与 .code 段，对于 NASM 汇编器，它可以出现在 SECTION .data 与 SECTION .text 段）。这些汇编器在对代码做汇编的时候会把每一个出现这种符号的地方都替换成对应的表达式。

GAS

```
.equ identifier, expression    # numeric operations
.equ identifier, value         # a single value (constant)
.equ identifier, "symbol"      # another existing symbol
```

MASM

```
identifier EQU expression      ; numeric operations
identifier EQU symbol          ; another existing symbol
identifier EQU <text>          ; useful for non-integer data (float, string)
```

NASM

```
identifier: EQU expression     ; numeric operations
identifier: EQU value          ; a single value (constant)
identifier: EQU "text"         ; text can be four characters max (32-bit)
```

从刚才这些规则可以看出，各种汇编器在定义符号常量时所采用的办法有很大的区别。

笔者建议大家只针对数值表达式来使用 EQU 命令。

```
.equ test, (2 * 6 / 3)    # GAS
test EQU (2 * 6 / 3)      ; MASM
test: EQU (2 * 6 / 3)     ; NASM
```

EQU 命令还有一个用途，是通过创建符号常量来表示某个标识符所指代的数据占用了多大的内存空间。下面给出的 GAS 与 NASM 范例代码，采用表达式来计算字符串所占据的字节数。**当前位置计数器**（current location counter）用来指代位置计数器的当前内存地址（GAS 用句点 . 表示，MASM 及 NASM 用美元符号 $ 表示），将该地址与前一个字符串的起始内存地址相减即可算出字符串所占据的字节数。MASM 可以通过等号（=）把计算结果定义成标识符。

GAS

```
motd: .byte "Are you in the matrix?\0"
.equ len, (. - motd)
```

MASM

```
motd  BYTE "Are you in the matrix?",0
len = ($ - motd)
```

NASM

```
motd: DB "I'm in the matrix?",0
len:  EQU ($ - motd)
```

MASM 还提供了一种写法，能够创建动态符号，称为文本宏（text macro）。由 TEXTEQU 命令所创建的符号可以表示涉及其他符号的表达式，甚至还可以表示指令。

MASM

```
identifier TEXTEQU %(expression)
identifier TEXTEQU textmacro
identifier TEXTEQU <text>
```

下面这段范例代码就演示了怎样用这种符号来表示指令，请注意，由于指令不是整数表达式，因此它的左右两端是用尖括号括起来的。

MASM

```
MULTIPLIER = 2
freq    TEXTEQU %(400 * MULTIPLIER)
movFreq TEXTEQU <mov eax, freq>
```

把 movFreq 这个文本宏创建好之后，你可以在任何一行代码中通过该符号来执行它所表示的那条指令，也就是 mov eax, freq。

MASM

```
.code
movFreq
```

## 3.4 写出能够正常运行的范例程序

你必须先掌握这一章所讲的基础知识才能很好地理解后续各章，因此，尽管第 4 章与第 5 章会详细解释各条指令，但你在看那两章之前还是应该先写出一个可以正常运行的汇编程序，以便总结本章的这些基础知识。程序 3-2 就是这样的一个例子，笔者针对 3 种汇编器分别提供了 3 个版本。其中 GAS 版本的代码包含完整的注释，而 MASM 与 NASM 版本则只有代码没有注释。你可以从 https://github.com/brianrhall/Assembly 下载代码，其中的 3 个版本都带有完整的注释。

**学习指南：** 用你喜欢的开发环境编写下面这段范例代码，并确保它能够顺利运行。然后，故意制造一些语法错误，看看系统会给出怎样的提示。开发者常犯的错误有：拼错变量名，用尺寸不合的数据操作变量与寄存器，或是忘记在适当的地方写上冒号。你要通过实验来了解这些错误会引发什么样的错误信息，这样以后再遇到类似的信息时就能够知道代码错在哪里了。

**程序 3-2 一个可以正常运行的范例程序**

GAS

```
.data                                   # Section for variable definitions

decimalLiteral:     .byte 31            # Variable storing 31
hexLiteral:         .long 0xF           # Variable storing F (15 in decimal)
charLiteral:        .byte 'A'           # Variable storing 65 in decimal

# Variable containing a string that has a line break and is null-terminated
stringLiteral:      .ascii "This string has\na line break in it.\0"

# Variable that calculates the value of an expression to determine the
# length, in bytes, of the variable "stringLiteral" by subtracting the
# starting memory address of the variable from the current memory address
.equ lenString, (. - stringLiteral)

.bss                                    # Section for uninitialized variables
.lcomm unInitVariable, 4                # Uninitialized, 4-byte variable

.text                                   # Section for instructions
.globl _main                            # Make the label "_main"
                                        # available to the linker as an
                                        # entry point for the program
_main:                                  # Label for program entry

# Label and instruction on
# the same line below
partOne: movl $10, %eax                 # Assign 10 to the eax register
addl hexLiteral, %eax                   # Add the value in hexLiteral to
                                        # the contents of the eax register
                                        # and store the result in eax

partTwo:                                # Label on its own line
inc %eax                                # Increment the value in eax

pushl $0                                # Push the return value on the stack
subl $4, %esp                           # Pad the stack for INT requirement
movl $1, %eax                           # Set the system call to 1 for exit
int $0x80                               # Issue the kernel interrupt
.end                                    # End assembling
```

MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096

ExitProcess PROTO, dwExitCode:DWORD

.data
decimalLiteral  BYTE 31
hexLiteral      DWORD 0Fh
charLiteral     BYTE 'A'
unInitVariable  DWORD ?

stringLiteral
  BYTE "This is a stringthat",0dh,0ah
  BYTE "has a line break in it.",0
lenString       EQU ($ - stringLiteral)

.code
_main PROC

partOne: mov eax, 10
add eax, hexLiteral

partTwo:
inc eax

INVOKE ExitProcess, 0
_main ENDP
END
```

NASM

```
SECTION .data

decimalLiteral:   DB 31
hexLiteral:       DD 0Fh
charLiteral:      DB 'A'

stringLiteral:
  DB "This is a string that",0ah
  DB "has a line break in it.",0
lenString: EQU ($ - stringLiteral)

SECTION .bss
unInitVariable:   RESD 1

SECTION .text
global _main

_main:

partOne: mov eax, 10
add eax, hexLiteral

partTwo:
inc eax

mov eax, 1
mov ebx, 0
int 80h
```

**编程知识：**程序 3-2 演示了怎样在 32 位的环境（32 位的 Mac 程序、Windows 程序以及用 NASM 汇编器开发的 Linux 程序）中适当地终止或者说退出汇编程序。在其他的环境（例如用 GAS 汇编器开发的 32 位 Linux 程序，以及各种 64 位环境）中，可能需要改用另一套例行的流程来退出程序。本书附录 A 列出了用 GAS、NASM 及 MASM 这 3 种汇编器开发 32 位及 64 位程序时所需遵循的退出流程。至于怎样通过系统调用来执行这套流程，则会在第 10 章讲解。

## 3.5　小结

这一章讲解了汇编语言的基本要素。大家现在应该已经知道汇编语言的结构以及程序代码中的各个区段了，而且也应该学会了怎样在代码中定义变量并用数据对其做初始化。笔者讲解了各种汇编器在某些重要的语法（例如创建变量与编写注释）上有何区别。本章给出了两个范例程序，并分别针对 3 种汇编器提供了 3 种版本，你可以从这些范例出发学着编写代码并对其做汇编，以后编写程序时也可以把这些范例当作模板来参考。

## 3.6　关键术语

array（数组）
character literal（字符字面量）
comments（注释）
current location counter（当前位置计数器；现行位置计数器）
directives（命令；指示）

DUP
EQU
identifiers（标识符）
immediate（立即值；立即数）
instructions（指令）
integer literals（整数字面量）
labels（标签）
mnemonic（助记符）
multi-line comments（多行注释）
program sections（程序段；程序区段）
real numbers（实数）
reserved words（保留字）
single-line comments（单行注释）
string literals（字符串字面量）
strings（字符串）
symbolic constants（符号常量）
TEXTEQU
uninitialized variables（未初始化的变量）
variable（变量）

## 3.7 代码回顾

| | |
|---|---|
| .ASCIZ | GAS 的数据命令，用来给字符串末尾自动添加值为 0 的字节 |
| ADD | 加法指令，将一个寄存器 / 变量的值加到另一个寄存器 / 变量 |
| ADDL | GAS 的（长整数）加法指令，将一个寄存器 / 变量的值加到另一个寄存器 / 变量 |
| DUP | MASM 的命令，用来创建大小相同的一连串重复值 |
| EQU, .equ | 符号创建命令，用来根据表达式的结果创建新的符号 |
| INC | 递增指令，用来给寄存器 / 变量的值加 1 |
| MOV | 复制指令，把一个寄存器 / 变量中的值复制到另一个寄存器 / 变量 |
| MOVL | GAS 的（长整数）复制指令，把一个寄存器 / 变量中的值复制到另一个寄存器 / 变量 |
| STC | 设定处理器的进位标志所用的指令 |
| TEXTEQU | MASM 的命令，根据文本替换的结果创建新的符号 |

## 3.8 习题

### 填空题

1. _____值是由开发者直接指定的，而不是根据表达式求出来的。
2. 整数字面量默认采用_____进制。
3. 如果想在 NASM 代码中用十六进制来表示十进制值 50，那么应该写成_____。
4. 如果想在 GAS 代码中用二进制来表示十六进制值 0x34，那么应该写成_____。
5. 字符字面量以_____形式存放在内存中。
6. 本书只推荐用_____、_____及_____这三种字符来给标识符命名。
7. _____可以指挥汇编器去完成各种任务，例如定义变量、指明代码所在的内存区段等。
8. 标签后面必须写上_____号。
9. 只有_____及_____两种汇编器支持多行注释。
10. 编写 MASM 代码时，单行注释以_____号开头。
11. MASM 代码用_____命令来声明 32 位的带符号整数变量。
12. NASM 代码用_____命令来预留 64 位的未初始化内存。
13. 在 MASM 代码中，字符串中的换行用十六进制值_____表示。在 NASM 代码中，字符串中的换行用十六进制值_____表示。
14. 可以结合 EQU 命令与_____判断字符串的长度。
15. 指令所执行的操作可以简写成几个英文字母，这些简称叫作_____。

**判断题**

1. GAS 的单行注释以分号开头。
2. 有些指令不带操作数。
3. 代表当前位置的符号在 NASM 代码中用 $ 字符来表示。
4. GAS 的 .octa 命令用来声明 80 位的变量。
5. MASM 代码中的未初始化变量声明在 .bss 段。

## 3.9 作业

前两道题分别要求你必须用某种特定的汇编器来完成，之所以这样限定，是为了帮助你熟悉本书所用的几种汇编器。本章的其中一项目标就是使你了解这几种汇编器之间的区别。第 3 道题不限定具体的汇编器，你选用自己最喜欢的那一种来完成就可以了。

### 3.9.1 定义变量

用 GAS 汇编器制作一款程序并在其中完成下列任务。

- 用大小合适的数据类型来声明 sum 变量，并将其初始值正确地设置为 0x10000。
- 用名为 message 的变量来表示这三行文字（以及其中的断行）：

```
Welcome to Assembly programming.
Your grade will be randomly assigned
by the Intel 8086 processor!
```

- 声明不含初始值的 input 变量并使其能够容纳 0 至 255 之间的值。
- 用一条指令，把 input 变量的值设置成与十进制的 200 相对应的二进制字面量。

### 3.9.2 在表达式中运用符号

用 NASM 汇编器制作一款程序并在其中完成下列任务。

- 创建名为 message 的变量并用它来表示这两行文字（以及其中的断行）：

```
You already know what the next
variable will be, don't you?
```

- 用名为 length 的变量来表示 message 变量所指向的字符串的大小。
- 通过 EQU 命令创建名为 length5 的变量（该变量也必须位于 data 段中），并用它来表示 5 与 length 相加的结果。
- 用只带一个操作数的指令来给 length5 变量的值加 1（想一想加 1 又可以叫作什么？）。

### 3.9.3 语法翻译

把下面这段 GAS 程序转换成符合 MASM 语法或 NASM 语法的汇编代码。你可以参考附录 A 来完成。

```
# Syntax translation - GAS
.data

.bss
.lcomm letter, 1
.lcomm r, 4
.lcomm s, 4
.lcomm t, 4
.lcomm x, 2
.lcomm y, 2
.lcomm z, 2

.text
```

```
.globl _main
_main:

movb $0x77, letter
movl $0x5, r
movl $0x2, s
movw $0xa, x
movw $0x4, y

movw x, %ax
addw y, %ax
movw %ax, z

movw x, %ax
subw y, %ax
movw %ax, z

movl $0x0, %edx
movl r, %eax
movl s, %ecx
divl %ecx
movl %eax, t

movl $0x0, %edx
movl r, %eax
movl s, %ecx
divl %ecx
movl %edx, t

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

### 3.9.4 按照正确的顺序执行运算（挑战题）

编写程序计算表达式 answer = (A + B) - (C + D) 的值。

- 答案必须存放在大小合适的变量中，以正确保存根据给定的数据 (A, B, C, D) 算出的结果。
- 数据 (A, B, C, D) 的值必须保存在 *eax*、*ebx* 等寄存器里，而不能用变量来保存。
- 必须给数据 (A, B, C, D) 提供初始值。
- 创建一个字符串，以便在输出运算结果的时候可以将其用作提示语，这个字符串的内容可以是“The answer is:”(答案是：)。你也可以不打印这个字符串。
- 像 3-5 节里的程序 3-2 那样给每行代码加上注释，以简单地解释其含义。

第 4 章

Assembly Programming and Computer Architecture for Software Engineers

# 基本指令

## 目标

- 用整数做算术运算
- 在二进制位的层面上操纵数据
- 了解各种寻址模式之间的区别
- 构建数组
- 把数据转换成大小及类型有所不同的其他格式

## 网络资源

与汇编器有关的参考资料

- https://sourceware.org/binutils/docs/as/（GAS 参考资料）
- https://msdn.microsoft.com/en-us/library/afzk3475.aspx（MASM 参考资料）
- http://www.nasm.us/doc/nasmdoc0.html（NASM 参考资料）

x86 指令集里的各条指令

- http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (Volume 1, Chapter 5 Instruction Set Summary, Section 5.1)
- http://www.felixcloutier.com/x86/
- https://en.wikipedia.org/wiki/X86_instruction_listings
- http://www.nasm.us/doc/nasmdocb.html

## 4.1 简介

本章讲解基本的汇编指令以供你编写简单的程序。无论对哪一种软件来说，算术运算都是基础的组成部分，因此笔者首先介绍基本的算术指令。然后开始详细讲解怎样在执行这些操作的过程中正确地使用数据，这包括如何引用以及如何传输数据。我们会讲到怎样根据字节偏移量来手动地访问内存中某个位置上的数据，怎样创建并使用数组，以及怎样运用指针。

> **提示：** x86 与 x86_64 指令集是相当复杂的。我们在这一章乃至整本书中所看到的相关内容应该说是一种概述。笔者不会把每条指令都拿出来解释一遍，而是会提供足够的信息使你了解与指令有关的一些重要概念，并以此为基础学会编写 x86 汇编程序。至于指令集里的其他一些指令则可以在必要的时候再去深入探究。

不是每一条指令都能接受各种类型的操作数，因此，在本章以及后续各章所定义的语法规则中，笔者采用下面这套命名惯例来描述指令所支持的操作数类型。表 4-1 列出了三种操

作数并分别用一个字母来简记。

表 4-1 操作数的类型及简称

| 操 作 数 | 描 述 |
| --- | --- |
| L | 字面值（Literal，或者说立即值），例如 42 |
| M | 内存（Memory，或者说变量）操作数，例如 numOfStudents |
| R | 寄存器（Register），例如 *eax* |

某些指令支持多种类型的操作数，在这些情况下，笔者会写出它们的简称并用斜线将其隔开，例如 M/R 就表示此处既可以写变量又可以写寄存器。除了可以接受好几种操作数之外，有些指令还会限制操作数的大小。对于这些情况，笔者会在表示操作数类型的字母后面，用数字写出操作数最多可以占据几个二进制位，例如 L8、M16、M32/R32。

**编程知识：** 在很多情况下，MASM 会根据上下文来推测指令中的操作数是什么类型，有的时候 NASM 也是这样。与之相反，GAS 不会去推测数据的类型，而是通常在指令的末尾写上一个表示操作数大小的字母，这个字母与表 3-11 中相应命令的首字母是一样的，例如适用于字节的加法（ADD）指令就是 ADDB，适用于长整数的加法指令就是 ADDL。

在撰写指令的目标操作数时，如果要对变量解引用（dereference），NASM 要求你必须指出大小，也就是必须在变量名的前面写上一个表示尺寸的命令，例如用 BYTE 表示字节、WORD 表示字、DWORD 表示双字等，比方说像这样：`mov DWORD [test], eax`。

学完这一章，你就可以理解并执行基本的算术运算、数据寻址操作及数据传输操作了。

## 4.2 数据的移动与算术运算

第 3 章说过，汇编语言里的一行代码只对应于一项操作，而不像高级语言那样，某一行较为复杂的语句有可能会转换成很多项底层操作。例如用 C++ 语言所写的代码 `result = (var1 + 5) / (var2 - 4);`，看上去似乎只执行了一项操作，因为这条语句毕竟只有一行，但实际上处理器至少执行了 3 项不同的操作（一项是加法，一项是减法，另一项是除法），此外，要把运算结果保存到 result 里还必须再执行一项操作才行。用汇编语言编写代码时，必须注意它和高级语言之间的区别，这样才能写出合适的程序。

刚接触汇编语言的人经常会把各操作之间的顺序写错。高级语言能够自动依照适当的顺序来执行算术运算，但是汇编语言却要求开发者自己来安排指令之间的顺序，以达成正确的结果。

### 4.2.1 移动数据

为了能够完成一系列指令，在很多情况下都必须把数据从一个地方复制到另一个地方。例如刚才那行 C++ 代码要想顺利执行完毕，就必须把等式的计算结果保存到 result 变量里，不过在这之前，先得把合适的值复制到 var1 与 var2 两个变量中。用汇编语言来开发程序时尤其需要经常要在寄存器与内存之间移动数据。用来完成这项操作的是 MOV 指令，它会把数据从来源地点复制到目标地点。在下面所举的 GAS 代码中，MOV 指令后面的斜体

字母 S 需要替换成表示操作数实际尺寸的相应字母。

GAS

```
MOVS $L/M/%R, M/%R          # MOVL $10, sum
```

MASM

```
MOV M/R, L/M/R              ; MOV eax, sum
```

NASM

```
MOV SIZE [M]/R, L/[M]/R     ; MOV DWORD [sum], edx
```

MOV 指令有几条具体的要求：

- 两个操作数的大小必须相同（稍后我们会讲到，这项要求可以通过一些命令来放宽）。
- 两个操作数不能全是内存操作数（也就是说要想在两个内存操作数之间移动数据，必须用寄存器做中介）。
- 指令指针寄存器（*ip*/*eip*/*rip*）不能用作目标操作数。

有一种特殊的情况，是令两个位置上的数据彼此互换。这项需求可以用 MOV 指令实现，但是不能直接移动，而是要先把其中一份数据临时保存到某个地点。比方说，要想交换 *eax* 与 *ebx* 这两个寄存器里的数据，必须通过 3 个步骤完成。第一步，把 *ebx* 中的数据临时存放在某个地方，例如 *edx* 中；第二步，把 *eax* 中的数据复制到 *ebx*；第三步，把临时存放在 *edx* 中的数据复制到 *eax*。这套步骤需要用 3 条指令来交换 *eax* 与 *ebx*。有种更简单的办法只需一步即可交换数据，即使用 XCHG 指令。

GAS

```
XCHGS M/%R, M/%R            # XCHGL %eax, sum
```

MASM

```
XCHG M/R, M/R               ; XCHG eax, sum
```

NASM

```
XCHG SIZE [M]/R, [M]/R      ; XCHG DWORD [sum], edx
```

和 MOV 指令类似，XCHG 指令的两个操作数也不能全都是内存操作数。

还有一些指令用来移动特定类型的数据，这些指令适合实现较为复杂的需求，本章稍后会讲到它们。

> 📖 **学习指南：** 编写一个程序，试着把数据分别复制到第 2 章所提到的每一个寄存器中。注意，有一些寄存器，不能用作 MOV 指令的来源操作数或目标操作数。

### 4.2.2　加法与减法

一般的加减法涉及字面量与变量，不过，在讲这种加减法之前，先来看两种最简单的形式，也就是加 1 及减 1。开发过软件的人都知道，这是两种很重要的操作，可以实现许多功能，例如记录循环的执行次数，处理用户输入的信息，或是引用数组中的各个元素等。高级语言提供了相应的运算符用来递增或递减某个变量，与之类似，汇编语言也提供了递增或递

减指令。

INC 与 DEC 指令可以操作内存操作数（变量）或寄存器。这两个指令的目标很简单，就是给操作数加 1 或减 1。

GAS

```
INCS M/%R          # INCL sum
DECS M/%R          # DECL %eax
```

MASM

```
INC M/R            ; INC sum
DEC M/R            ; DEC eax
```

NASM

```
INC SIZE [M]/R     ; INC DWORD [sum]
DEC SIZE [M]/R     ; DEC eax
```

**编程知识：** NASM 对待变量的方式有点像 C++ 对待指针的方式：如果直接写出变量名本身，那么它会将其视为内存地址。要想引用该地址中的内容（也就是由变量名所指代的那份数据），必须用一对方括号给变量解引用（例如 [sum]），这就好比在 C++ 中用星号给指针解引用一样。

ADD 与 SUB 指令可以用字面量来做加减法，也可以用内存或寄存器中的值来计算。

GAS

```
# src, dest
ADDS $L/M/%R, M/%R       # ADDW $50, sum
SUBS $L/M/%R, M/%R       # SUBL val, %eax
```

MASM

```
; dest, src
ADD M/R, L/M/R           ; ADD sum, 50
SUB M/R, L/M/R           ; SUB eax, val
```

NASM

```
; dest, src
ADD SIZE [M]/R, L/[M]/R      ; ADD WORD [sum], 50
SUB SIZE [M]/R, L/[M]/R      ; SUB eax, [val]
```

**编程知识：** AT&T 语法（也就是 GAS 及 Clang 用的那种语法）与 Intel 语法（也就是 MASM 及 NASM 用的那种语法）之间有个重要的区别，即来源操作数和目标操作数的先后顺序。GAS 总是先写来源操作数而 MASM 与 NASM 则先写目标操作数。

还有一个指令也很有用，即 NEG 指令，它可以切换操作数的正负号，也就是对操作数的值求补。补码表示法在第 1 章中讲过。

GAS

```
NEGS M/%R          # NEGL %eax
```

MASM

```
NEG M/R                ; NEG sum
```

NASM

```
NEG SIZE [M]/R         ; NEG BYTE [sum]
```

程序 4-1 演示了 INC、DEC、ADD、SUB 与 NEG 指令的用法。首先，创建 sum 与 val 两个 32 位变量并为其指定初始值。然后将 *eax* 寄存器设为 0 并给它加 1。接下来，把十进制的字面量 200 加到 *eax* 寄存器的当前值中，并将结果保存在 *eax* 中。其后，从寄存器 *eax* 的值中减去变量 *val* 的值，并将结果保存在 *eax* 中。最后，把 *eax* 的值存放到变量 *sum* 中，给该变量减 1。程序 4-1 即将结束的时候 *sum* 变量的最终值是 −175（用 32 位补码来表示，就是 11111111111111111111111101010001）。

**程序 4-1　加法与减法**

GAS

```
.data
sum: .long 0
val: .long 25

.text
.globl _main
_main:

movl $0, %eax
incl %eax
addl $200, %eax
subl val, %eax
movl %eax, sum
decl sum
negl sum

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO,
dwExitCode:DWORD

.data
sum DWORD 0
val DWORD 25

.code
_main PROC

mov eax, 0
inc eax
add eax, 200
sub eax, val
mov sum, eax
dec sum
neg sum

INVOKE ExitProcess, 0
_main ENDP
END
```

NASM

```
SECTION .data
sum: DD 0
val: DD 25

SECTION .text
global _main
_main:

mov eax, 0
inc eax
add eax, 200
sub eax, [val]
mov [sum], eax
dec DWORD [sum]
neg DWORD [sum]

mov eax, 1
mov ebx, 0
int 80h
```

### 4.2.3　乘法与除法

乘除法要比加减法复杂一些。你可能觉得乘法指令与除法指令跟加法指令与减法指令一样，也是接受两个操作数并计算其结果，但对于 x86 汇编来说不一定是这样。操作数的个数取决于指令所针对的数据是否带符号。带符号的数据与无符号的数据都有各自的乘除法指令。我们先讲无符号的版本，因为带符号的更复杂。

MUL 指令用来给无符号的整数执行乘法。该指令只接受一个操作数即乘数（multiplier）。那么，被乘数（multiplicand）与积（product）保存在哪里呢？被乘数保存在与乘数尺寸相对应的累加寄存器中（例如 8 位乘法的被乘数保存在 8 位的 al 寄存器里，16 位、32 位及 64 位也可类推，参见表 4-2）。与被乘数一样，积的保存位置也无须手工指定，因为这是由 MUL 指令自动指定的。

表 4-2 执行无符号整数的乘法运算时所涉及的乘数、被乘数与积

| 乘 数 | 被 乘 数 | 积 |
|---|---|---|
| M8/R8 | *al* | *ax* |
| M16/R16 | *ax* | *dx:ax* |
| M32/R32 | *eax* | *edx:eax* |
| M64/R64 | *rax* | *rdx:rax* |

被乘数与积的存放地点是根据乘数的尺寸默认指定的。以范例 4-1 为例，在计算 8096 乘以 64 之前我们首先考虑这两个数的大小。两者中较大的是 8096，它需要用 13 个二进制位来保存，于是我们将其存放在某个至少能容纳 16 个二进制位的地方，这意味着另一个数 64 也必须保存在至少能容纳 16 个二进制位的地方，例如 *ax* 寄存器里。两个 16 位的数相乘结果很有可能超过 16 位，因此，乘积应该保存在能够容纳 32 个二进制位的地方。具体到本例来说，就是保存在 *dx:ax* 中（这种写法的意思是，乘积的高 16 位保存在 *dx* 里，低 16 位保存在 *ax* 里）。8096 × 64 的结果是 518144，需要用 19 个二进制位才能保存得下。*dx:ax* 是可以容纳这个结果的，但假如保存在只能容纳 16 个二进制位的地方就会溢出。

范例 4-1 两个无符号的 16 位整数相乘

```
mov ax, 64      ; Store the multiplicand for 16-bit multiplication
mov bx, 8096    ; Store the multiplier in a 16-bit register so when
                ; we multiply, AX is used as the multiplicand and
                ; a 32-bit result is produced
mul bx          ; bx * ax = 518144
                ;         = 00000000000001111110100000000000
                ;      dx = 0000000000000111, ax = 1110100000000000
```

**编程知识：**执行完乘法操作之后可以看看 CPU 的标志位。对于无符号乘法来说，如果乘积的高半段是 0，那么 Carry Flag（CF）就是 0，否则是 1。对于带符号乘法来说，如果乘积高半段的每一个二进制位都与乘积低半段的最高有效位相同，那么 CF 就是 0，否则是 1。

GAS

```
MULS M/%R           # MULW %ax
```

MASM

```
MUL M/R             ; MUL ax
```

NASM

```
MUL SIZE [M]/R      ; MUL DWORD [sum]
```

**带符号的整数**相乘要通过 IMUL 指令执行。与 MUL 一样，该指令也有单操作数的版本，在这种情况下，它所支持的操作数与表 4-2 中所列的相同。这条指令与 MUL 的主要区别在于它还有双操作数及三操作数的版本。双操作数的版本与 ADD 及 SUB 指令类似，也就是会把这两个操作数都当成运算的源数据，并且要将其中一个操作数用作运算的目标，以便存放计算结果。因此对于保存运算结果的目标操作数来说，IMUL 与 ADD 及 SUB 一样都会将其中原有的值覆盖掉。

如果不想让 IMUL 指令把这个值覆盖掉，那么可以使用三操作数的版本。该版本要求开发者指出乘数、被乘数与积的位置。下面列出了这 3 种版本在 3 种汇编器里的格式，尤其要注意双操作数与三操作数的版本，看看各操作数需要按照怎样的先后顺序来书写。这两个版本都会把乘积保存到某个寄存器里，三操作数的版本还要求开发者必须在规定的位置上提供字面量（或者说立即值）。

GAS

```
# One-operand - follows Table 4.2
IMULS M/%R                              # IMULL %ebx

# Two-operand - src, dst
IMULS $L/M/%R, %R                       # IMULW val, %ax

# Three-operand - imm, src, dst
IMULS $L, M/%R, %R                      # IMULL $10, val, %eax
```

MASM

```
# One-operand - follows Table 4.2
IMUL M/R                                ; IMUL ebx

# Two-operand - dst, src
IMUL R, L/M/R                           ; IMUL ax, val

# Three-operand - dst, src, imm
IMUL R, M/R, L                          ; IMUL eax, val, 10
```

NASM

```
# One-operand - follows Table 4.2
IMUL SIZE [M]/R                         ; IMUL DWORD [sum]

# Two-operand - dst, src
IMUL R, L/[M]/R                         ; IMUL ax, [val]

# Three-operand - dst, src, imm
IMUL R, [M]/R, L                        ; IMUL eax, [val], 10
```

**编程知识：**执行乘法与除法的时候，一定要选择尺寸合适的寄存器。如果为了节省空间而一味地选择小寄存器，那么有可能因为积或商过大无法存入目标寄存器而发生进位或溢出。

与 MUL 类似，除法也分为无符号与带符号两种。DIV 指令用来执行无符号整数的除法运算，它会将结果以商与除数的形式分别保存。

表 4-3　执行无符号整数的除法运算时涉及的除数、被除数、商与余数

| 除　数 | 被 除 数 | 商 | 余　数 |
|---|---|---|---|
| M8/R8 | *ax* | *al* | *ah* |
| M16/R16 | *dx:ax* | *ax* | *dx* |
| M32/R32 | *edx:eax* | *eax* | *edx* |
| M64/R64 | *rdx:rax* | *rax* | *rdx* |

使用 DIV 指令的时候只需要指定除数就可以了（如果按照分子与分母的形式来写除法算式，就意味着只需要指定分母，也就是位于分数线下面的数）。被除数（位于分数线上的数）需要提前加载到与除数尺寸相对应的寄存器里，如表 4-3 所示。此外，如果被除数的值单用 *ax* 就能表示得下，但规则要求必须用 *dx*:*ax* 合起来表示，那么被除数的正负号必须在高位寄存器里体现出来。比方说，在给带符号的整数做除法时，如果为负值的 16 位被除数需要用 *dx*:*ax* 合起来表示，那么其低位寄存器 *ax* 里的符号位（也就是处在最高有效位的 1）必须扩展到高位寄存器 *dx* 中，使得 *dx* 的每一个二进制位也都是 1。这么做是为了在数值不变的前提下，令 *dx*:*ax* 的正负号与 *ax* 相同。本章稍后就要讲到符号扩展。

下面以 32 / 3 为例，用范例 4-2 来演示 32 位无符号整数的除法运算。首先，必须把被除数提前加载到 *edx*:*eax* 中。十进制数 32 的二进制形式并没有超过 32 位，因此单用 *eax* 就可以表示得下，但为了符合 DIV 指令的规则，我们必须用 *edx*:*eax* 的格式保存它。32 是个正数，因此 *eax* 的最高有效位是 0，这意味着 *edx* 中的每一个二进制位也都必须是 0，只有这样，才能保证 *edx*:*eax* 合起来的值是与 *eax* 本身相同的正数。接下来，我们选用 *ecx* 寄存器来保存除数 3。然后执行除法指令。该指令会把计算所得的商 10 保存到 *eax* 寄存器里，并把余数 2 保存到 *edx* 寄存器里。

**范例 4-2　两个无符号的 32 位整数相除**

```
mov edx, 0     ; Load EDX:EAX with the dividend, 32
mov eax, 32
mov ecx, 3     ; Load ECX with the divisor, 3
div ecx        ; 32 / 3 = 10r2
               ; eax = 0000000Ah, edx = 00000002h
```

GAS

```
DIVS M/%R      # DIVW %ax
```

MASM

```
DIV M/R        ; DIV ax
```

NASM

```
DIV SIZE [M]/R      ; DIV DWORD [sum]
```

**带符号的整数**需要用 IDIV 指令来相除。与 IMUL 指令不同，该指令没有双操作数或三操作数的版本。你还是只提供一个操作数，即除数。

> **提示：**如果除法运算所得商太大无法保存到目标寄存器中，就会发生溢出。这会使处理器出现异常并且令程序暂停（halt）。为了尽量避免这种情况，你应该选择尺寸合适的操作数。

GAS

```
IDIVS M/%R     # IDIVW %ax
```

MASM

```
IDIV M/R       ; IDIV ax
```

NASM

```
IDIV SIZE [M]/R      ; IDIV DWORD [sum]
```

**编程知识：**与乘法指令不同，这两种除法指令都不会通过设定状态标志来反映与运算结果有关的信息，也就是说，相关标志的值是未定义（undefined）的。

程序 4-2 演示了 MUL、IMUL、DIV 及 IDIV 指令的用法。它用 MUL 指令以及三种形式的 IMUL 指令来计算 8 与 664751 的乘积，然后用带符号与不带符号的除法指令将 5318008 与 8 相除。

**程序 4-2　乘法与除法**

GAS

```
.data
mval: .long 664751
dval: .long 8

.text
.globl _main
_main:

# MUL 1-op
movl mval, %eax
movl $8, %ebx
mull %ebx

# IMUL 1-op
movl mval, %eax
movl $8, %ebx
imull %ebx

# IMUL 2-op
movl $8, %eax
imull mval, %eax

# IMUL 3-op
imull $8, mval, %eax

# DIV 1-op
movl $0, %edx
movl $5318008, %eax
movl dval, %ecx
divl %ecx

# IDIV 1-op
movl $0, %edx
movl $5318008, %eax
movl dval, %ecx
idivl %ecx

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO,
dwExitCode:DWORD

.data
mval DWORD 664751
dval DWORD 8

.code
_main PROC

; MUL 1-op
mov eax, mval
mov ebx, 8
mul ebx

; IMUL 1-op
mov eax, mval
mov ebx, 8
imul ebx

; IMUL 2-op
mov eax, 8
imul eax, mval

; IMUL 3-op
imul eax, mval, 8

; DIV 1-op
mov edx, 0
mov eax, 5318008
mov ecx, dval
div ecx

; IDIV 1-op
mov edx, 0
mov eax, 5318008
mov ecx, dval
idiv ecx

INVOKE ExitProcess, 0
_main ENDP
END
```

NASM

```
SECTION .data
mval: DD 664751
dval: DD 8

SECTION .text
global _main
_main:

; MUL 1-op
mov eax, [mval]
mov ebx, 8
mul ebx

; IMUL 1-op
mov eax, [mval]
mov ebx, 8
imul ebx

; IMUL 2-op
mov eax, 8
imul eax, [mval]

; IMUL 3-op
imul eax, [mval], 8

; DIV 1-op
mov edx, 0
mov eax, 5318008
mov ecx, [dval]
div ecx

; IDIV 1-op
mov edx, 0
mov eax, 5318008
mov ecx, [dval]
idiv ecx

mov eax, 1
mov ebx, 0
int 80h
```

### 4.2.4 移位

虽然有专门的指令用来做乘除法，但开发者应该知道，还有一种方式能够更快地执行这些操作。因为处理器是把值当成一系列二进制位来处理的，因此我们也可以在程序中按照这样的形式操纵数据。例如，乘法与除法就可以通过位操作实现，而不一定非得用 MUL 或 DIV 指令。

假设变量 val 的值是 20。

```
val BYTE 20
```

范例 4-3 演示了 val 值的二进制形式。

**范例 4-3 用二进制位来表示十进制的值 20**

| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

如果把范例 4-3 中的二进制位左移 1 位，那么值就会从 20 变成 40。如果左移两位就会从 20 变成 80。每左移 1 位就相当于乘以 2，左移 $n$ 位则相当于乘以 2 的 $n$ 次方，如表 4-4 所示。

**表 4-4 左移的位数与对应的放大倍数**

| 左移的位数 | 移动后的值是原值的几倍 |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |

根据范例 4-3 可知，将 20 左移两位就相当于乘以 4，移动后的值是 80（参见范例 4-4）。

**范例 4-4 左移位**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 左移之前 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 20 |
| 左移两位之后 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 80 |

注意看，原数的每一个二进制位都左移了两格，右方空出来的位置都填写 0。范例 4-4 演示的是正常情况，如果换一种情况，例如把 64 左移两位，那会怎么样？

**范例 4-5 无法通过左移位求出正确结果的情况**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 左移之前 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 64 |
| 左移两位之后 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

64 * 4 的积能不能用 8 个二进制位来表示？不能。因为积是 256，比 8 位无符号整数所能表示的最大值 255（也就是每个二进制位都是 1 的数）还要大。在移位的过程中，原来处在第 7 位（假设最右侧的二进制位叫作第 1 位）的 1 会溢出到第 9 位，如果仅按照 8 位二进制数来解读这个结果就会得到 0。原数中，位于第 8 位的 0 与位于第 7 位的 1 在移位之后分别变成第 10 位与第 9 位，都超出了 8 位二进制数的范围。

然而，移出去的二进制位其实也并没有完全丢失。每次向左移动 1 位的时候都会有 1 个二进制位超出表示范围。比方说，如果分两步来观察范例 4-5，会发现第一次向左移动 1 位会令原数的最高有效位 0 丢失，使得原本位于它右侧的 1 变成当前的最高有效位。第二次向左移动 1 位又会令当前的最高有效位 1 丢失。不过，每次移出去的二进制位其实是保存在 Carry Flag（CF，进位标志）中的。也就是说，第一次左移 1 位之后移出去的 0 会出现在 CF 里，第二次左移 1 位之后移出去的 1 也会出现在 CF 里，并取代刚才的 0 成为新的 CF

值。无论左移几位，在移出去的那些二进制位里总有 1 位能保存到 CF 中。尽管有这种机制，但在执行算术运算的时候还是应该确保数据本身完整无缺。

左移位可以用来实现与 2 的幂相乘，而右移位则可以用来实现与 2 的幂相除。它的规则与前者类似，但有一点要注意：通过右移位的方式计算被除数为奇数的除法肯定会丢失数据。因为奇数的最低有效位是 1，无论把它右移几位都会使这个值为 1 的二进制位出现在表示范围之外从而令运算结果不够准确。

左移位（shift left）指令 SHL 与右移位（shift right）指令 SHR 都可以把内存操作数或寄存器中的值移动一定的位数，这个位数用字面量来指定。

GAS

```
SHLS $L, M/%R      # SHLL $2, val
SHRS $L, M/%R      # SHRL $1, %ebx
```

MASM

```
SHL M/R, L         ; SHL val, 2
SHR M/R, L         ; SHR ebx, 1
```

NASM

```
SHL SIZE [M]/R, L      ; SHL DWORD [val], 2
SHR SIZE [M]/R, L      ; SHR ebx, 1
```

**学习指南：第 5 章会讲到分支，这是一种令程序在不同情况下进入不同路径的机制。程序可以根据 Carry Flag 的状态或某项数学运算的结果，来决定其后应该执行哪些指令。**

前面那些例子所说的移位是**逻辑移位**（logical bit shift），这会令某些二进制位出现在存储范围之外，同时会令空出来的那些数位都填上 0。无符号的数据通常是可以做逻辑移位的，然而若是对带符号的数据做逻辑移位，则有可能破坏其内容。比方说，如果把负值右移位，左侧就会出现一些值为 0 的二进制位，从而令整个数值的符号位由 1 变成 0。

如果想通过左移位或右移位的方式给带符号的整数做乘除法，必须使用**算术移位**（arithmetic bit shift）指令以便保留其符号位。SAL（Shift Arithmetic Left）是算术左移，SAR 是算术右移（Shift Arithmetic Right）。范例 4-6 演示了在 NASM 代码中对值为 -32（也就是 11100000b）的 8 位变量 val 做算术左移与算术右移的结果。

**范例 4-6　在 NASM 代码中执行算术移位**

| 指令 | 二进制形式的结果 | 十进制形式的结果 |
|---|---|---|
| SAL WORD [val], 2 | 10000000 | -128 |
| SAR WORD [val], 2 | 11111000 | -8 |

**编程知识：SAL（算术左移）与 SHL（逻辑左移）指令，其实是一样的（或者说是同义词），因为在该指令可以处理的范围内执行左移，并不会令负数的符号位发生变化。**

SAL 与 SAR 的语法跟对应的逻辑移位指令类似，都是将内存操作数或寄存器中的值移动一定的位数，这个位数也用字面量来指定。

GAS

```
SALS $L, M/%R      # SALL $2, val
SARS $L, M/%R      # SARL $1, %ebx
```

MASM

```
SAL M/R, L         ; SAL val, 2
SAR M/R, L         ; SAR ebx, 1
```

NASM

```
SAL SIZE [M]/R, L      ; SAL DWORD [val], 2
SAR SIZE [M]/R, L      ; SAR ebx, 1
```

### 4.2.5 处理负值

刚才讲了无符号数与带符号数的几种基本算术运算，现在，我们专门来谈带符号的数值。执行任何算术指令之前，都必须确保相关的数值及其符号是正确的，而且要将它放在合适的寄存器里。

如范例 4-7 所示，各种汇编器都允许开发者把负值简单地保存到寄存器或变量中。这些值会自动以补码的形式存放。

范例 4-7 负值的简单用法

| GAS | MASM | NASM |
|---|---|---|
| .data<br>val: .long -5<br><br>.text<br>movl $-10, %eax | .data<br>val DWORD -5<br><br>.code<br>mov eax, -10 | SECTION .data<br>val: DD -5<br><br>SECTION .text<br>mov eax, -10 |

有一种情况尤其需要把负值处理好，即在执行 IDIV 指令之前，必须先把负值适当地加载到相应的寄存器里。以 32 位带符号整数的除法为例，根据表 4-3 可知，此时的被除数应该表示成跨越 *edx* 与 *eax* 的 64 位带符号整数。比方说要计算 −534 除以 15，就必须先把被除数 −534 转换成 64 位补码，并将高 32 位存放在 *edx* 中，将低 32 位存放在 *eax* 中。把 −534 保存在 *eax* 中是很简单的，但是怎样将它的符号位（也就是 1）扩展到 *edx* 寄存器，从而令其中每个二进制位也都变成 1？

这可以通过相关的指令实现。表 4-5 列出了 4 种能够执行符号扩展（sign extension）的指令，也就是 CBW（将 BYTE 转成 WORD）、CWD（将 WORD 转成 DWORD）、CDQ（将 DWORD 转成 QWORD），以及 CQO（将 QWORD 转成 OCTA）。

表 4-5 符号扩展指令

| 指　令 | 从 | | 到 | |
|---|---|---|---|---|
| | 寄存器 | 大小 | 寄存器 | 大小 |
| CBW | *al* | 8 | *ax* | 16 |
| CWD | *ax* | 16 | *dx:ax* | 32 |
| CDQ | *eax* | 32 | *edx:eax* | 64 |
| CQO | *rax* | 64 | *rdx:rax* | 128 |

下面先举个例子，演示怎样对 8 位（也就是一个字节）的值执行符号扩展，令其变为 16 位（也就是一个字），然后再回到早前那个问题，谈谈如何按照该思路来正确地保存被除数 -534。

**范例 4-8　用 CBW 指令执行符号扩展**

| GAS | MASM/NASM |
|---|---|
| movw $0, %ax<br>movb $-5, %al<br>cbw | mov ax, 0　　; ax = 0000000000000000<br>mov al, -5　　; ax = 0000000011111011<br>cbw　　; ax = 1111111111111011 |

在范例 4-8 中，我们首先把 0 值复制到 16 位的 *ax* 寄存器中，以便将该寄存器的每一个二进制位都清空。然后把 -5 保存到 8 位寄存器 *al* 中。现在，*ax* 寄存器的低 8 位，也就是 *al* 寄存器中存放的是 -5 的补码。此时假如直接用 IDIV 指令来执行带符号的除法操作，那么它会把被除数当成 251（因为被除数 *ax* 的最高有效位是 0，这意味着它的值将按照正数来解读）。为了避免这个错误，我们必须把 *al* 的符号位扩展到 *ax* 中，也就是用 *al* 的最左侧那一位（本例中是 1）来填充 *ax* 的高 8 位，使得整个 16 位寄存器的内容变为被除数 -5 的补码。

现在回到 -534 / 15 的问题上。程序 4-3 给出了正确执行这次除法所需的汇编代码。把 -534 移动到 *eax* 寄存器之后，必须将它的符号扩展到 *edx*，这是通过 CDQ 指令实现的，该指令会将双字扩展成四字，也就是用 *eax* 的符号位来填充 *edx* 的每一个二进制位。设置好被除数之后，把除数放入 *ebx* 寄存器，接下来执行除法指令 IDIV。运行完该指令后，*eax* 中保存的是商也就是 FFFFFFDDh（相当于十进制的 -35），*edx* 中保存的是余数也就是 FFFFFFF7h（相当于十进制的 -9）。

**程序 4-3　被除数为负数的除法**

| GAS | MASM | NASM |
|---|---|---|
| .text<br>.globl _main<br>_main:<br><br>movl $-534, %eax<br>cdq<br>movl $15, %ebx<br>idivl %ebx<br><br>pushl $0<br>subl $4, %esp<br>movl $1, %eax<br>int $0x80<br>.end | .386<br>.MODEL FLAT, stdcall<br>.STACK 4096<br>ExitProcess PROTO,<br>dwExitCode:DWORD<br><br>.code<br>_main PROC<br><br>mov eax, -534<br>cdq<br>mov ebx, 15<br>idiv ebx<br><br>INVOKE ExitProcess, 0<br>_main ENDP<br>END | SECTION .text<br>global _main<br>_main:<br><br>mov eax, -534<br>cdq<br>mov ebx, 15<br>idiv ebx<br><br>mov eax, 1<br>mov ebx, 0<br>int 80h |

## 4.3　数据的寻址与传输

### 4.3.1　数据对齐

汇编代码的一项优势在于能够写出高效的程序，而其中一个重要的方面则体现在内存访

问上。CPU 访问存放在偶数地址上的数据要比访问存放在奇数地址上的数据更快。例如对于 32 位系统来说，如果能把数据以 32 位为单元来存放并保证结构体都处在 16 字节的边界上（也就是其内存地址为 16 的倍数），那么内存访问操作执行起来就比较简单。总之，把数据放在偶数地址上对程序的执行速度是有好处的。

GAS

```
.balign alignment     # .balign 2
```

MASM

```
ALIGN alignment       ; ALIGN 2
```

NASM

```
SECTION .data                          SECTION .bss
ALIGN alignment       ; ALIGN 2        ALIGNB alignment  ; ALIGNB 2
```

每种汇编器都有 1 条或多条命令用来修改位置计数器。GAS 用的是 .balign 命令，MASM 用的是 ALIGN 命令，NASM 则依据不同的程序段分别使用 ALIGN 或 ALIGNB 命令。这几种命令的参数都必须是整数，而且应该是 2 的幂。这些命令会推进位置计数器，直到它的值变为该整数的倍数为止。这可以用来确保数据出现在偶数的内存地址上，这样做的好处刚才已经讲过了。

**范例 4-9　在 MASM 代码中对齐数据**

```
; Assume the starting memory address is 0x00000000h
var BYTE 31       ; The next variable would start at 0x00000001h, which is odd
ALIGN 2           ; Ensure the next memory location is even
var2 WORD 1337    ; var2 starts at 0x00000002h
```

就目前来说，我们还没有必要专门通过代码中的命令来实现手工对齐，或是通过汇编器与编译器的命令选项来实现自动对齐，然而到了第 6 章及第 8 章就不是这样了，因为那些章节所讲的函数调用及向量数据等话题与数据对齐是有关系的。

**编程知识：**GAS 有很多种形式的对齐命令。标准的 .align 命令在 x86 与 x86_64 架构中确实与刚才所讲的 .balign 命令相同，但是在其他一些架构（例如 ARM 或 PPC）中却有着不同的行为。因此，为了使程序的行为能够在各种架构之间保持一致，笔者还是建议采用 .balign 命令来对齐数据。

### 4.3.2　数据寻址

谈到内存地址这个话题，我们就应该讲一讲它作为指令操作数可以有几种用法。前面那些代码都是拿变量做**直接寻址**（direct addressing），也就是直接使用由该标识符所代表的内存地址上的数据。但有的时候，我们还想像 C++ 等高级语言那样通过类似指针的方式间接地使用数据，也就是说，不直接使用某个内存地址上的数据，而是把该数据也当成地址来确定另外一个内存位置，转而使用那个位置上的数据。这叫作**间接寻址**（indirect addressing）。图 4-1 演示了两种寻址方式之间的区别。

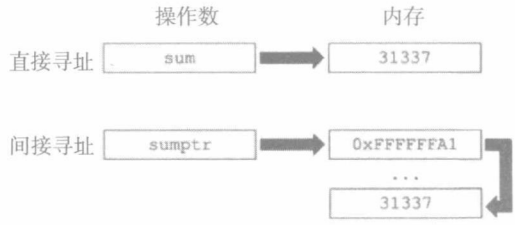



图 4-1　直接寻址与间接寻址

为了画起来简单一些，图 4-1 是用主存来举例的，其实寄存器也可以做直接寻址或间接寻址。这就是说，在指令中既可以用寄存器来表示它本身的值也可以把该值当成内存地址，转而使用那个地址上的值。无论针对主存还是针对寄存器，图 4-1 所描绘的原理都适用：直接寻址是直接访问某个值，而间接寻址则是通过值所代表的内存地址来访问另一个值。

知道这两种寻址方式之后就可以讲讲内存地址了。其中有个重要的问题，就是怎样判断变量所在的内存地址与汇编程序的数据段起点之间相隔多远。NASM 的变量表示的就是其内存地址，而不是该地址中的值（如果想使用这个值，要用方括号括起来，例如 [sum]）。在 MASM 代码中，操作数的地址可以结合 MOV 指令及 OFFSET 命令来获取。在 GAS 代码中，如果给变量名的前面加上美元符号，那么就意味着使用的是该变量的地址而不是变量本身的值。

下面这几段代码分别演示了怎样用三种汇编器来获取变量的地址。其中，GAS 与 MASM 版本的注释举例说明了如何将 32 位变量 sum 的内存地址保存到 *esi* 寄存器，NASM 版本的注释举例说明了如何将该变量的内存地址保存到另一个变量。

GAS

```
MOVS $M, M/%R      # MOVL $sum, %esi
```

MASM

```
MOV M/R, OFFSET M      ; MOV esi, OFFSET sum
```

NASM

```
MOV SIZE [M]/R, M      ; MOV DWORD [sumaddr], sum
```

**编程知识：**如果想在 GAS 版本的代码里通过 MOV 指令来操作某个变量的内存地址，那么该指令后面添加的字母必须与表示地址所用的二进制位个数相等。例如在 32 位环境下，由于内存地址要用 32 个二进制位来表示，因此，必须将其视为长整数型的数据来存放，这要求我们把字母 L 添加在 MOV 之后，也就是要用 MOVL 指令来操作。至于变量本身的类型则与此无关，不论变量本身是 byte 型、word 型，还是 long 型，它的地址在 32 位环境下都要用 32 个二进制位来表示。同理，如果要针对 64 位环境来开发程序，那么就要用 MOVQ 指令来保存变量的内存地址了，因为此环境下的内存地址是用 64 个二进制位（也就是一个四字（quadword））来表示的。

LEA（Load Effective Address，加载有效地址）指令在 32 位与 64 位模式之下都可以把操作数的地址加载到目标中。由于实际地址要在程序运行的时候才能够知道，因此，涉及这

些地址的操作应该用 LEA 来完成。

GAS

```
LEAS $M, %R        # LEAL sum, %esi
```

MASM

```
LEA R, M           ; LEA esi, sum
```

NASM

```
LEA R, [M]         ; LEA eax, [sum]
```

就获取内存地址而言，MOV 与 LEA 指令之间的重要区别在于，后者的目标操作数必须是寄存器。

### 4.3.3 数组

存放内存地址所带来的好处在实现较为复杂的功能时显得尤为突出。其中一种常见的情况就是用间接寻址来操纵数组。与高级语言一样，汇编语言里的**数组**也是由类型相同的一系列数据构成的，这些数据在内存中以相等的间隔存放。数组的名称实际上指的是该数组中的首个元素。假设有下面两个数组：一个是字节数组，另一个是双字数组。

**范例 4-10 MASM 中的数组**

```
arrayA BYTE 2, 4, 6, 8
arrayB DWORD 0FFFFFh, 0FFFFEh, 0FFFFDh, 0FFFFCh
```

每个数组都包含 4 个元素，不过，单用 `arrayA` 或 `arrayB` 这样的写法只能分别访问其中的首个元素。那么，后面那些元素应该如何访问呢？这需要结合早前讲过的指令来做。高级语言可以用下标访问数组，首个元素的下标是 0，后续元素依次类推（例如 `arrayA` 中的元素，可以用 `arrayA[0]`、`arrayA[1]` 等写法来访问）。编译器会判断数组元素的大小，并根据下标将位置计数器移动到适当的地方。而在用汇编语言写程序时，元素的大小与位置则要由开发者自己来判断并加以计算。

为了访问数组中的数据，我们必须知道如何计算每个元素的地址。第 2 章讲过，内存地址是以字节为单位来编排的。图 4-2 描绘了主存中的一段内容，并指出了 `arrayA` 及 `arrayB` 两个数组里的每个元素所对应的内存地址。

| arrayA | |
|---|---|
| **Address** | **Value** |
| 0x00000000 | 2 |
| 0x00000001 | 4 |
| 0x00000002 | 6 |
| 0x00000003 | 8 |

| arrayB | |
|---|---|
| **Address** | **Value** |
| 0x00000000 | FFFFF |
| 0x00000004 | FFFFE |
| 0x00000008 | FFFFD |
| 0x0000000C | FFFFC |

图 4-2 数组的寻址

`arrayA` 是 BYTE（字节）数组，因此，每个元素都与上一个元素相距 1 字节，`arrayB` 是 DWORD（双字）数组，因此，其每个元素都与上一个元素相距 4 字节（一个双字相当于 4 个字节）。

**编程知识：**访问数组元素时，一定要正确算出该元素距离数组开头有多少个字节。汇编器不会自动检查你所做的访问是否与元素之间的边界相合。如果你在访问 DWORD 数组的下一个元素时，没有正确地推进 4 个字节，而是只推进了 3 个字节，汇编器不会报错，然而你访问到的实际位置并不是你本来想要访问的位置。

第 3 章说过，字符串（也就是由字符所构成的数组）的长度可以通过当前位置计数器（current location counter）来计算。其实不只是字符串，其他数组所占据的字节数也可以用这个办法求出。

GAS

```
arrayA: .long 2, 4, 6, 8
.equ len, (. - arrayA)          # len = 16
```

MASM

```
arrayA DWORD 2, 4, 6, 8
len EQU ($ - arrayA)            ; len = 16
```

NASM

```
arrayA: DD 2, 4, 6, 8
len: EQU ($ - arrayA)           ; len = 16
```

现在重新讨论 LEA 与 MOV 指令，看看怎样把它们与字节偏移量（byte offset）相结合以操作位于某个内存地址上的数据。这里所说的字节偏移量使用常数表示。下面这些代码，演示了如何访问位于下标 [1] 与下标 [3] 处的元素，其中第一次访问是把元素用作来源操作数，第二次访问是把元素用作目标操作数。

GAS

```
                                # Store item 1 from arrayA in ebx
LEAS M, %R                      # LEAL arrayA, %eax
MOVS CONSTANT(%R), M/%R         # MOVL 1(%eax), %ebx
                                # %ebx = 4

                                # Store 10 into element 3 of arrayB
LEAS M, %R                      # LEAL arrayB, %eax
MOVS $L/%R, CONSTANT %R         # MOVL $10, 8(%eax)
                                # arrayB = 0xFFFFF,0xFFFFE,0xA,0xFFFFC
```

MASM

```
                                ; Store item 1 from arrayA in eax
MOV R, [M+CONSTANT]             ; MOV eax, [arrayA + 1]
                                ; eax = 4

                                ; Store 10 into item 3 of arrayB
MOV [M+CONSTANT], R             ; MOV [arrayB + 8], 10
                                ; arrayB = 0xFFFFF,0xFFFFE,0xA,0xFFFFC
```

NASM

```
                                ; Store item 1 from arrayA in eax
```

```
MOV R, [M+CONSTANT]              ; MOV eax, [arrayA + 1]
                                 ; eax = 4

                                 ; Store 10 into item 3 of arrayB
MOV SIZE [M+CONSTANT], L/R       ; MOV DWORD [arrayB + 8], 10
                                 ; arrayB = 0xFFFFF,0xFFFFE,0xA,0xFFFFC
```

**编程知识：** 在 GAS 代码中，如果要用 MOV 指令来操作内存地址，必须在后面写上一个与地址宽度相符的字母。LEA 指令也是如此，这个字母指的是该指令所取的内存地址用多少个二进制位来表示，而不是说数组中的每个元素有多大，因此，在 32 位环境下该指令要写成 leal，在 64 位环境下要写成 leaq。

**学习指南：** 在你习惯的开发环境里定义几个大小不同的数组，并试着通过计算字节偏移量来访问其中的元素。用调试器看看正确的偏移量与错误的偏移量分别会得出怎样的结果。

有些开发者还是喜欢像高级语言那样，在指令里通过下标而不是字节偏移量来访问数组元素。如果想这样做，可以修改刚才的范例代码。下面这种写法演示了怎样通过下标访问数组 arrayB 的最后一个元素（也就是下标为 3 的元素）。GAS 版本的代码是先把数组地址加载到某个寄存器里，然后把下标加载到另一个寄存器中，接下来通过 MOV 指令实现访问。访问的时候要指出存放数组地址及下标的两个寄存器以及每个数组元素所占据的字节数（例如 1、2、4、8）。

GAS

```
LEAS M, %R                       # LEAL arrayB, %eax
MOVS L, %R                       # MOVL $3, %edx
MOVS (%R, %R, L), %R             # MOVL (%eax, %edx, 4), %ebx
```

MASM/NASM

```
MOV R, L                         ; MOV edx, 3
MOV R, [M+R*L]                   ; MOV eax, [arrayB + edx * 4]
```

要想用与高级语言类似的风格书写这种汇编代码，就必须多费一条指令。这两种写法哪一种更好，可以根据你要制作的程序来选择。

最后介绍几个用 MASM 代码来操作数组时的专用命令。前面讲过如何用计算的办法确定数组所占据的字节数，而现在要介绍的这几条命令则可以更为灵活地查询与数组（或变量）有关的特征。TYPE 命令返回变量所占据的字节数，LENGTHOF 命令返回数组中的元素个数，SIZEOF 命令则返回整个数组所占据的总字节数（相当于把 TYPE 命令与 LENGTHOF 命令所返回的结果乘起来）。

MASM

```
MOV eax, TYPE arrayA             ; eax = 1
MOV ebx, LENGTHOF arrayA         ; ebx = 4
MOV ecx, SIZEOF arrayA           ; ecx = 4

MOV eax, TYPE arrayB             ; eax = 4
MOV ebx, LENGTHOF arrayB         ; ebx = 4
MOV ecx, SIZEOF arrayB           ; ecx = 16
```

程序 4-4 先创建含有 4 个元素的数组，然后读取其中的两个元素，接下来用其他数据覆盖另外两个元素当前的值。

**程序 4-4　数组**

GAS

```
.data
array: .long 1, 2, 3, 4

.text
.globl _main
_main:

# Load using
# byte offsets
leal array, %esi
movl (%esi), %eax
movl 4(%esi), %ebx

# Save using indices
movl $2, %edx
movl $10, (%esi, %edx, 4)
movl $3, %edx
movl $20, (%esi, %edx, 4)

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO,
dwExitCode:DWORD

.data
array DWORD 1, 2, 3, 4

.code
_main PROC

; Load using
; byte offsets
mov eax, [array]
mov ebx, [array + 4]

; Save using indices
mov edx, 2
mov [array+edx*4], 10
mov edx, 3
mov [array+edx*4], 20

INVOKE ExitProcess, 0
_main ENDP
END
```

NASM

```
SECTION .data
array: DD 1, 2, 3, 4

SECTION .text
global _main
_main:

; Load using
; byte offsets
mov eax, array
mov ebx, [array + 4]

; Save using indices
mov edx, 2
mov DWORD [array+edx*4], 10
mov edx, 3
mov DWORD [array+edx*4], 20

mov eax, 1
mov ebx, 0
int 80h
```

### 4.3.4　改变数据的大小及类型

有时，我们必须（或者说最好能够）把数据从一个地点移动到尺寸不同的另一个地点。比如要从 32 位变量 val 的值中取出低 16 位，并将其复制到 *ax* 这样的 16 位寄存器中（下面这段代码就是用 *ax* 来举例的）。由于大多数指令都会在来源与目标操作数的尺寸不符时发生错误，因此为了使代码能够顺利通过汇编，我们必须设法解决这个问题。GAS 版本的代码处理起来很简单，只需在 MOV 指令后面写上适当的字母即可，这个字母与要操作的数据所表现出的尺寸相对应。由于本例的目标是 16 位值，也就是一个 WORD，因此采用 MOVW 指令就可以了。NASM 版本的代码也比较简单，因为我们可以直接指出来源操作数的大小，这与前面的例子中指出目标操作数的大小时所用的写法类似。MASM 版本的代码需要使用一种新的命令即 PTR 命令，它会把早前定义的变量尺寸忽略掉，转而将该变量视为一个指向某份数据的指针，这份数据的大小由 PTR 前面的词决定。

GAS

```
MOVS M, %R      # MOVW val, %ax
```

MASM

```
MOV R, SIZE PTR M       ; MOV ax, WORD PTR val
```

NASM

```
MOV R, SIZE [M]         ; MOV ax, WORD [val]
```

把数据从空间较大的地方拿出来存入尺寸较小的位置，可以令程序运作得更加高效，然而还有一些情况则要求我们必须反过来处理。虽然前面讲过一组功能类似的命令（也就是CBW、CWD、CDQ 与 CQO），但那些命令是专门针对 *al*、*ax*、*eax* 及 *rax* 几个累加寄存器而言的。对于一般的寄存器来说，如果简单地把数据存放到大寄存器的低半区（例如存放到 *eax* 寄存器的 *ax* 部分），那么高半区里可能会有早前某些操作所遗留下来的数据，若是不考虑这一点就直接拿整个大寄存器（例如 *eax*）去操作，则有可能产生错误的结果。

为此，我们在将数值复制到大寄存器的低半区时，应该使用 MOVZ/MOVZX（ZX 表示 Zero eXtend，用 0 来扩展）与 MOVS/MOVSX（SX 表示 Sign eXtend，用符号位来扩展）两种指令，因为它们能够在复制的同时，用 0 位或符号位来填充高半区里的每一个二进制位。下面三段代码演示了怎样用这两种指令把 16 位变量 sum 的值复制到 32 位寄存器 *eax* 中。

GAS

```
MOVZSS M/%R, %R        # MOVZWL sum, %eax
MOVSSS M/%R, %R        # MOVSWL sum, %eax
```

MASM

```
MOVZX R, M/R           ; MOVZX eax, sum
MOVSX R, M/R           ; MOVSX eax, sum
```

NASM

```
MOVZX R, SIZE [M]/R     ; MOVZX eax, WORD [sum]
MOVSX R, SIZE [M]/R     ; MOVSX eax, WORD [sum]
```

MOVS 与 MOVZ 指令是需要稍作解释的，尤其是 GAS 版本的代码，因为它的指令后面带有两个字母，而不像平常的 GAS 版 MOV 指令那样只带一个字母。MASM 版本的代码会根据指令所涉及的寄存器，以及变量的定义方式推出这两个操作数的大小（对本例来说，它知道来源操作数 sum 早前已经定义成 16 位的 WORD 值了，而目标操作数 eax 则是个 32 位的寄存器，因此会执行从 16 位到 32 位的扩展）。NASM 版本会结合指令所涉及的寄存器与开发者在变量名前面所写的命令（例如 [sum] 前面的 WORD）来运作。GAS 版本要求开发者同时指出两者的大小，首先指出左侧操作数也就是来源操作数的大小，接着指出右侧操作数也就是目标操作数的大小。比方说，如果要把 16 位的值（也就是 WORD 值），以 0 位扩展的形式复制到某个能够保存 32 位值（也就是 LONG 值）的地方，那么就应该把 MOVZ 指令写成 MOVZWL。

x86 汇编指令的助记符，其写法在 3 种汇编器中通常是一样的，但刚才讲的两种指令属于例外情况。GAS 采用 AT&T 语法把 0 位扩展与符号扩展分别叫作 MOVZ 与 MOVS，并要求开发者在后面写上两个代表来源与目标操作数的后缀字母，而 MASM 与 NASM 则采用 Intel 语法，分别将其称为 MOVZX 与 MOVSX。

## 4.4 小结

这一章的总话题是如何操纵数据。我们以无符号及带符号两种形式举例讲解了基本的算术指令，并对比了直接寻址与间接寻址两种数据访问方式。其中，后一种方式在处理数组的时候很重要。此外，笔者还举例讲解了怎样对数据做移位以及如何在大小不同的两个地点之

间移动数据。学到了这些知识之后，你现在应该可以用汇编语言来完成一些简单的任务，例如在不同的温度表示法之间转换，计算面积，乃至列出一份简单的班级成绩表。

## 4.5　关键术语

.balign
ADD
ALIGN
ALIGNB
arithmetic bit shifting（算术移位）
array（数组；阵列）
CBW
CDQ
CQO
CWD
DEC
direct addressing（直接寻址；直接定址）
DIV
IDIV
IMUL
INC
indirect addressing（间接寻址；间接定址）
LEA
LENGTHOF
logical bit shifting（逻辑移位）
MOV
MOVS/MOVSX
MOVZ/MOVZX
MUL
NEG
OFFSET
PTR
SAL
SAR
SHL
SHR
SIZEOF
SUB
TYPE
XCHG

## 4.6　代码回顾

| | |
|---|---|
| .balign | GAS 版的汇编代码可以用这条命令把指令计数器推进到下一个合适的内存地址，该地址必定是命令参数的整数倍 |
| ADD | 对两个操作数执行加法 |
| ALIGN | MASM 与 NASM 版本的汇编代码可以用这条命令把指令计数器推进到下一个合适的内存地址，该地址必定是命令参数的整数倍 |
| ALIGNB | 与 ALIGN 功能相同，用在 NASM 版汇编代码的 .bss 段 |
| CBW | 将 *al* 的值转换到（扩展到）*ax* 中 |
| CDQ | 将 *eax* 的值扩展到 *edx:eax* 中 |
| CQO | 将 *rax* 的值扩展到 *rdx:rax* 中 |
| CWD | 将 *ax* 的值扩展到 *dx:ax* 中 |
| DEC | 给操作数减 1（对操作数做递减） |
| DIV | 执行无符号除法以求出商及余数 |
| IDIV | 执行带符号除法以求出商及余数 |
| IMUL | 执行带符号乘法 |
| INC | 给操作数加 1（对操作数做递增） |
| LEA | 将变量的有效地址载入寄存器 |
| LENGTHOF | MASM 版本的汇编代码可以用这条命令查询数组中的元素个数 |
| MOVS/MOVSX | 以符号位扩展（符号扩展）的形式把值从一个地方复制到另一个地方 |
| MOVZ/MOVZX | 以 0 位扩展（0 扩展）的形式把值从一个地方复制到另一个地方 |

（续）

| | |
|---|---|
| MUL | 执行无符号乘法 |
| NEG | 取操作数的相反数（对操作数求补） |
| OFFSET | MASM 版本的代码可以用这条命令查询某个变量的位置与程序数据段的起始位置之间相隔多远 |
| PTR | MASM 版本的代码可以用这条命令将操作数视为一个指针，从而令其指向大小不同的数据 |
| SAL | 对带符号的整数做算术左移位 |
| SAR | 对带符号的整数做算术右移位 |
| SHL | 对无符号的整数做左移位 |
| SHR | 对无符号的整数做右移位 |
| SIZEOF | MASM 版本的代码可以用这条命令获取数组所占据的总字节数 |
| SUB | 对两个操作数做减法 |
| TYPE | MASM 版本的代码可以用这条命令查询变量所占据的字节数 |

## 4.7 习题

### 填空题

1. INC 指令最多可以带有_____个操作数。
2. 在 NASM 版本的汇编代码中把 16 位变量 `val` 的值减 1，可以用 DEC _____指令实现。
3. 在 GAS 版本的汇编代码中，指令的首个操作数是_____操作数。
4. 在 MASM 版本的汇编代码中，指令的首个操作数是_____操作数。
5. NEG 指令会对操作数_____，从而能够将其由正值变为负值。
6. 在 MUL 指令中以 QWORD 命令来处理操作数会导致运算结果保存在_____中。
7. IMUL 指令可以接受_____个操作数。
8. 用 DIV 指令对 32 位的被除数做除法时，被除数必须保存在_____中。
9. 如果 DIV 指令的除数是 64 位，那么商与余数分别会保存在_____与_____中。
10. IDIV 指令可以接受_____个操作数。
11. CBW、CWD、CDQ 与 CQO 指令能够以_____扩展的形式对_____寄存器中的内容做扩展。
12. 把数据对齐到_____内存地址上，可以令处理器更为迅速地访问数据。
13. 如果变量的值是内存地址，则可以通过该变量做_____寻址。
14. GAS 版本的代码可以用_____指令来保存变量的地址。
15. _____指令以符号扩展的形式复制数值，_____指令以 0 位扩展的形式复制数值。

### 判断题

1. IDIV 指令可以接受 1 个、2 个或 3 个操作数。
2. 编写 NASM 版本的汇编代码时，只有在变量充当来源操作数的情况下才需要指出该变量的大小。
3. CDQ 指令能够把 *ebx* 寄存器里的值以符号扩展的形式扩展到 *edx:ebx* 寄存器中。
4. ALIGNB 命令只能用在 NASM 版汇编代码的 .bss 段里。
5. 如果要在 GAS 版本的代码里通过 MOVZ 或 MOVS 指令对数据做扩展，那么必须在指令后面添加两个字母，用以指出来源操作数及目标操作数的大小。

## 4.8 作业

### 4.8.1 会跳闸吗？

假设电脑配有 1200 瓦的电源并插在 120 伏的标准墙面插座上。编写程序求出电源以最大功率运

作时通过电线的电流大小。计算公式是：W = VA，其中 W 表示功率，单位是瓦特（Watt，瓦），V 表示电压，单位是伏特（Volt，伏），A 表示电流，单位是安培（Amperage，安）。

### 4.8.2　判断平均考试成绩是否及格

假设某学生在计算机体系结构这门课里共参加了 4 次考试，每次所得的分数以及该次考试的满分都列在下面。现在要求编写程序以百分数（或者说满分是 100 分）的形式算出该学生的平均成绩。计算结果要写成整数，用来表示百分号前面的那一部分，比方说，如果平均成绩是 75%，那么计算结果就应该是 75。

- 25/30
- 89/100
- 49/50
- 80/150

### 4.8.3　操纵元素大小不相同的两个数组（挑战题）

- 编写程序，把 WORD 数组中的每一个元素分别乘以 8，并将新值保存在 DWORD 数组的对应元素中。
- WORD 数组的 4 个元素，其初始值分别是 5，10，15 及 20。
- DWORD 数组的 4 个元素，其最终值应该是 40，80，120 及 160。

# 中级指令

## 目标

- 学会高效地运用按位布尔指令
- 创建控制结构与决策结构，以实现分支逻辑
- 创建可以重复执行的代码块

## 网络资源

**与汇编器有关的参考资料**

- https://sourceware.org/binutils/docs/as/（GAS 参考资料）
- https://msdn.microsoft.com/en-us/library/afzk3475.aspx（MASM 参考资料）
- http://www.nasm.us/doc/nasmdoc0.html（NASM 参考资料）

**x86 指令集里的各条指令**

- http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (Volume 1, Chapter 5 Instruction Set Summary, Section 5.1)
- http://www.felixcloutier.com/x86/
- https://en.wikipedia.org/wiki/X86_instruction_listings
- http://www.nasm.us/doc/nasmdocb.html

## 5.1 简介

本章要介绍一些概念及汇编指令，以供开发者控制程序的流程。我们将讲解第 1 章中提到的布尔表达式，并介绍如何用汇编代码做布尔代数运算。此外，还要给出很多种方法来演示怎样用汇编代码编写条件语句及循环结构。把这一章的内容与前面几章结合起来，可以为你打下坚实的基础，使你能够用贴近底层的办法来解决计算任务。

## 5.2 按位执行的布尔运算

**第 1 章**讲过 NOT（非）、AND（与）、OR（或）及 XOR（异或）等布尔运算，虽然当时已经提到了一些与布尔逻辑有关的概念，但是开发者还必须知道：怎样才能在程序代码中使用这些概念来编写布尔表达式。汇编代码中用到的数字一般是十进制或十六进制，然而有时也可以使用二进制形式，因为这样更便于在其中的各个二进制位上执行**按位的布尔运算**（Boolean bitwise operation）。这种二进制位层面的操作有很多用途，例如可以加密数据、更新数值（比方说，把寄存器清零）、执行与掩码有关的操作（例如控制 CPU 与 FPU 时，以及根据网络协议来通信时就有可能要执行这样的操作）以及控制周边的硬件等。

首先要讲的按位运算是 NOT，它会分别反转操作数的每一个二进制位。这实际上就是

在计算操作数的反码（或对操作数求反）。表 5-1 再次列出了第 1 章讲过的 NOT 运算规则。

用来做 NOT 运算的那条指令也叫作 NOT，它带有一个操作数。该指令将会对此数求反并把结果写入原位置。

表 5-1　NOT 运算

| x | NOT x |
|---|---|
| 0(F) | 1(T) |
| 1(T) | 0(F) |

GAS

```
NOTS M/%R        # NOTB %al
```

MASM

```
NOT M/R          ; NOT al
```

NASM

```
NOT SIZE [M]/R      ; NOT BYTE [val]
```

接下来，看看 AND 指令的真值表（truth table，参见表 5-2）。这张表列出了由两个操作数的不同取值形成的四种情况，并指出了每种情况的计算结果。

其中的 F（假）与 T（真），可以像表 5-3 这样分别改写成 0 与 1，从而更加直观地展示出二进制形式的运算规则。

表 5-2　AND 指令的真值表

| x | y | x AND y |
|---|---|---|
| F | F | F |
| F | T | F |
| T | F | F |
| T | T | T |

表 5-3　AND 指令的真值表

| x | y | x AND y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

用只包含 1 个二进制位的值做例子可以帮助大家理解 AND 操作的规则，不过，该操作的好处其实更多地体现在位数较多的值上，也就是程序中经常用到的那些数。有一种常见的用途是通过 AND 来判断某值是偶数还是奇数。判断的原理是：检查受测数字的最低有效位（LSB）是 0 还是 1。

```
                ↓
40 = 00101000 Even
47 = 00101111 Odd
```

虽然我们一眼就能看出某数是奇数还是偶数，然而要想用代码表示出来恐怕还是得费些功夫，例如可能要通过 IF 语句判断 LSB 是不是 1，或者判断与 2 相除是否有余数等。观察 AND 操作的真值表，我们可以发现只有当两个操作数都是 True 的时候结果才是 True。于是，要想判断某数的最低有效位是不是 1，只需要把它与另外一个最低有效位为 1 的数做 AND 运算就可以了。可是数字的其他几个二进制位应该怎么写呢？

以 8 位的环境为例，我们想构造这样一个数字——凡是奇数跟它做 AND 结果都是 1（也就是最低有效位为 1，其余二进制位均为 0 的数）。现在还是回到真值表，我们可以发现无论 AND 操作的第一个操作数是 0 还是 1，只要第二个操作数是 0，结果就是 0。于是，我们所要构造的数字除了最低有效位是 1 之外，其他的二进制位都应该是 0，这样就能实现本段开头所说的效果了。例如十进制值 47 是个奇数，如果把它与我们所构造的这个数字 1（二进制形式是 00000001）求 AND，那么结果就是 1。

```
    47          00101111
AND  1      AND 00000001
------      ------------
     1          00000001
```

由此可见，只要构造出合适的数字使其中的一个二进制位为 1 且其余位置均为 0，那么就可以通过对受测值与该数字做 AND 来判断此值是奇数还是偶数。

还有一个任务也可以通过 AND 操作迅速解决，即将 ASCII 字符从小写变成大写。用高级语言来转换大小写通常需要借助一些函数来实现相应的转换规则，然而汇编语言却必不如此。仔细观察大写与小写字母所对应的二进制值就可以发现，大写与小写是由第 5 个二进制位（假设最右侧的叫作第 0 位）来控制的，除此之外，其他那些二进制位是完全相同的。表 5-4 对比了小写字母 a 与大写字母 A 的二进制形式。

表 5-4 大小写转换

| 字 符 | 十六进制值 | 二进制值 |
|---|---|---|
| a | 61h | 01100001 |
| A | 41h | 01000001 |

↑

我们由此可以构建这样一个掩码：在无须改变的那些位置上写 1，在第 5 位上写 0。然后将小写字母与该掩码做 AND，这样就能令其第 5 位变成 0，从而把它转为大写。此外，OR 与 XOR 操作也可以用来转换大小写。前者用来将大写转成小写，后者可以实现双向切换：如果原字母是小写就将其改为大写，如果原字母是大写就将其改为小写。

```
    01100001 = a
AND 11011111 = mask
------------
    01000001 = A

    01000001 = A
XOR 00100000 = mask
------------
    01100001 = a
```

与先判断数值是否在 ASCII 字母的范围内，然后再根据转换方式来加 32 或减 32 的办法相比，通过位掩码（bit mask）来转换效率更高。

AND 操作所对应的指令也叫作 AND。下面举例说明怎样把 *al* 寄存器中的十进制值与 1 做 AND 运算。运算的结果仍然保存在 *al* 寄存器里。这三段代码分别采用三种不同的数制（十进制、二进制、十六进制）表示字面量 1。

GAS

```
ANDS $L/M/%R, M/%R      # ANDB $1, %al
```

MASM

```
AND M/R, L/M/R          ; AND al, 00000001b
```

NASM

```
AND SIZE [M]/R, L/[M]/R         ; AND al, 0x1
```

如果想实现跟 AND 运算相反的效果（也就是置 1 而不是清零），可以用 OR 指令，对两个操作数的每个二进制位分别执行 OR 运算。在讲解 OR 指令的用途之前，还是先来复习一下它的运算规则，参见表 5-5。

表 5-5　OR 运算

| x | y | x OR y | x | y | x OR y |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 |

从表中可以看出，两个操作数里只要有一个是 True（或者说是 1）结果就是 True。这条规律可以用来设定控制位或标志位。比方说，要写个简单的程序让顾客能够自己选择比萨饼中的馅料。假如分别用 4 个 8 位的变量来表示 4 种馅料，那么总共需要 32 位，但其实只用一个 8 位变量就够了，因为我们可以拿其中的 4 个二进制位来分别表示顾客有没有选择某种馅料。如果选了该位就是 1，如果没选该位就是 0。

```
      |----洋葱
      ||---青椒
      |||--香肠
      ||||-意大利辣香肠
      ↓↓↓↓
00000000 馅料
```

一开始，把表示馅料的 TOPPINGS 变量清零，这意味着此时的比萨饼是一块还没有加入任何馅料的芝士比萨。如果顾客需要加意大利辣香肠（pepperoni）与洋葱（onion），那我们就查出与二者相对应的掩码，也就是 1（00000001b）与 8（00000100b），然后依次把 TOPPINGS 变量与掩码做 OR 运算（例如先把它和 00000001b 做 OR 运算，这表示先给比萨饼中洒入意大利辣香肠），最后的总结果就是顾客所选的全部馅料。这种办法要比采用 4 个 8 位变量的办法节省 24 位。

```
   00000000 TOPPINGS              00000001 TOPPINGS
OR 00000001 PEPPERONI          OR 00001000 ONION
-----------                    -----------
   00000001 TOPPINGS              00001001 TOPPINGS
```

连续做两次 OR 运算之后，TOPPINGS 的第 0 位与第 3 位就变成了 1，这表示顾客要求给比萨饼中加入意大利辣香肠与洋葱。

**学习指南：** 用本章所学的内容，编写一个算法来解析 TOPPINGS 变量，看看顾客要求添加哪些馅料。

下面 3 条规则以及右侧的范例代码分别演示了 OR 指令在 3 种 x86/x86_64 汇编器中的用法。

GAS

```
ORS $L/M/%R, M/%R        # ORB $8, %al
```

MASM

```
OR M/R, L/M/R        ; OR al, 00001000b
```

NASM

```
OR SIZE [M]/R, L/[M]/R        ; OR al, 0x8
```

OR 还有一个用法，是判断某数为正、为负，还是为 0。只要把受测的数与它自身做 OR 运算，就可以根据结果判明这三种情况了。之所以能够如此，是因为 OR 与 AND 指令都会设置处

理器中的许多标志位，例如CF、OF、PF、SF及ZF等。下面这行代码对var变量（假设该变量的值是13）与其自身做OR运算，代码的注释中标出了开发者应该去判断的那两个标志。

```
or var, var          ; SF = 0 (positive), ZF = 0 (non-zero)
```

XOR（异或）指令与OR类似，用来在两个操作数的每一对二进制位之间执行XOR运算。不过，它与OR有个关键的区别在于：只有当两个操作数一个是True一个是False时结果才是True，如果两个操作数均为True或均为False，那么结果是False，参见表5-6。

表5-6 XOR运算

| *x* | *y* | *x* XOR *y* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

由于XOR运算是可逆的（reversible），因此成为很多对称加密（symmetric encryption）算法与数据存储算法中的重要环节。将原值与另一个值（此值称作键，key）连取两次XOR，就可以令运算结果回到原值。

```
    01101011 ORIGINAL              01111010 "ENCRYPTED"
XOR 00010001 KEY               XOR 00010001 KEY
------------                   ------------
    01111010 "ENCRYPTED"           01101011 ORIGINAL
```

XOR运算还可以用在硬盘技术尤其是RAID（Redundant Array of Independent Disks，独立硬盘冗余阵列）技术上。这种技术用许多块硬盘组成一个RAID集（RAID set），组成方式有很多种，例如RAID0、RAID1、RAID5等。RAID技术的主要优势在于数据冗余，也就是说，即便其中一块硬盘坏了，整个系统也依然可以设法把丢失的数据重新构建出来，从而正常运作下去。其中重建数据这一步可以通过XOR来完成。

假设系统有5块硬盘，其中4块用来保存数据，另一块用来存放校验信息（parity information），这是一种与RAID3类似的运作方式。校验信息中的每个二进制位是通过对另外4块硬盘的对应二进制位做XOR得来的，比方说，校验信息的第0位，其值等于对另外4块硬盘所保存的数据信息中的那4个第0位做XOR，校验信息的第1位，其值等于对另外4块硬盘所保存的数据信息中的那4个第1位做XOR。只要把校验信息单独保存到一块硬盘上，那么即便前4块硬盘中有一块坏掉了数据也不会丢失。比方说，如果drive1坏了，那么可以对drive0、drive2、drive3以及保存校验信息的parity drive中的相应二进制位做XOR，这样得到的结果就是drive1中的这个二进制位本来应该拥有的值。于是就实现了数据重建。

```
     ORIGINAL                      REBUILD
drive0 = 00000000             drive0 = 00000000
drive1 = 01010101  XOR        drive2 = 11110000  XOR
drive2 = 11110000   ↓         drive3 = 11111111   ↓
drive3 = 11111111             parity = 01011010
-----------------             -----------------
parity = 01011010             drive1 = 01010101
```

下面3条规则以及右侧的范例代码分别演示了XOR指令在3种x86/x86_64汇编器中的用法。

GAS

```
XORS $L/M/%R, M/%R          # XORB $8, %al
```

MASM

```
XOR M/R, L/M/R        ; XOR al, 00001000b
```

NASM

```
XOR SIZE [M]/R, L/[M]/R    ; XOR al, 0x8
```

## 5.3 分支

**分支**（branch），意思是说程序可以根据开发者所实现的逻辑进入不同的执行路径，甚至可以直接跳过某些指令。分支有两种形式，一种是无条件地进入某路径，另一种是根据条件来做测试，从而依照测试结果进入相应的路径。我们首先必须理解前一种形式，然后再来讲解条件分支（conditional branching，也叫作条件转移）。

### 5.3.1 无条件跳转

**第 3 章**讲过怎样用标签把程序划分成很多个代码块。

```
computePay:
            instruction
            instruction
addToList:
            instruction
            instruction
```

上述代码共有 4 条指令及 2 个标签。前面说过，标签只是用来在代码中标明某个位置而已，它本身并不占据内存空间。这两个标签把程序划分成了两块，`computePay` 标签下面的那两条指令是第一块，`addToList` 标签下面的那两条指令是第二块。

在汇编代码中使用标签的其中一个原因是要给 JMP 指令提供跳转目标。这条指令能够直接令程序跳转到某个地方。

```
top:
        mov al, 3
        add al, 5
        jmp bottom
middle:
        add al, 32
bottom:
        add al, 2
```

这段代码并不会从头到尾地执行，而是会在给 *al* 加了 `5` 之后（也就是在执行了 `add al, 5` 这条指令之后）直接跳到由 `bottom` 标签所标注的地方，也就是说，它会把 `middle` 标签所表示的那块代码完全跳过去。这意味着 `32` 不会加到 *al* 中。程序执行完毕之后 *al* 的值是 `10`。

无条件跳转指令 JMP 在 3 种汇编器里的用法是一样的。

GAS/MASM/NASM

```
JMP LABEL     ; JMP doTheMath
```

### 5.3.2 有条件跳转

现在可以谈谈怎样通过条件测试来跳转到不同的代码块了。首先我们要讨论有哪几种办

法可以做条件测试（conditional test，或者说可以根据条件来做测试），然后我们要讲解有哪几种办法可以令程序根据条件测试的结果转入不同的分支。

在汇编语言中执行条件测试有很多种办法，其中之一是使用 TEST 指令。该指令会对两个操作数执行 AND 运算，但并不修改二者的值，而且也不保存计算结果。不过，它会根据计算结果设置处理器的 PF、SF 及 ZF 等标志位。在结果中，如果值为 1 的二进制位是偶数个，PF 标志就是 1，否则为 0，SF 标志反映结果的最高有效位，ZF 标志反映结果是否为 0。

**提示：CF 与 OF 也会为 TEST 指令所修改。无论计算结果如何，这两个标志位总是会清零。**

GAS

```
TESTS $L/M/%R, M/%R        # TESTB $32, %al
```

MASM

```
TEST M/R, L/M/R            ; TEST al, val
```

NASM

```
TEST SIZE [M]/R, L/[M]/R       ; TEST BYTE [val], ah
```

还有一种方法也可以做条件测试，就是用 CMP 指令来比较（compare）两个操作数。这能够实现出与高级语言类似的条件判断逻辑（例如等于、大于、小于等于）。高级语言的决策结构的一个例子是 `if` 语句。该语句会根据条件测试的结果令程序转入不同的分支。

```
if (condition is true)
     Do this sequence of instructions
else
     Do this sequence of instructions
```

汇编语言的条件跳转至少要用两条指令实现，首先，用一条指令比较两个操作数，然后，用一条或多条指令来处理比较结果。基本的条件测试逻辑可以由 CMP 指令起头。该指令会用目标操作数减去来源操作数，并根据计算结果修改 CPU 的相关标志位。

**提示：与 TEST 指令类似，CMP 指令不会修改这两个操作数，而且也不保存计算结果。**

GAS

```
CMPS $L/M/%R, M/%R             # CMPB $32, %al
```

MASM

```
CMP M/R, L/M/R                 ; CMP al, val
```

NASM

```
CMP SIZE [M]/R, L/[M]/R        ; CMP BYTE [val], ah
```

比较了两个操作数之后可以用很多种办法来跳转。刚才说过，TEST 与 CMP 指令会设置各种 CPU 标志位，而几乎所有的跳转指令都是根据一个或多个标志位的值来决定程序走向的。表 5-7 列出了常用的跳转指令以及每条指令所判断的 CPU 标志位，同时还说明了该指令针对的是无符号数还是带符号数。

表 5-7　条件跳转指令

| 符　号 | 标　志 | 指　令 | 描　述 |
|---|---|---|---|
| 带符号数与无符号数 | OF = 1 | JO | 如果溢出，就跳转 |
| | OF = 0 | JNO | 如果不溢出，就跳转 |
| | PF = 1 | JP | 如果奇偶校验位置位，就跳转 |
| | PF = 0 | JNP | 如果奇偶校验位不置位，就跳转 |
| | SF = 1 | JS | 如果符号位置位，就跳转 |
| | SF = 0 | JNS | 如果符号位不置位，就跳转 |
| | ZF = 1 | JE | 如果相等，就跳转 |
| | | JZ | 如果为 0，就跳转 |
| | ZF = 0 | JNE | 如果不相等，就跳转 |
| | | JNZ | 如果不为 0，就跳转 |
| | *CX* = 0 | JCXZ | 如果 *CX* 寄存器为 0，就跳转 |
| | *ECX* = 0 | JECXZ | 如果 *ECX* 寄存器为 0，就跳转 |
| | *RCX* = 0 | JRCXZ | 如果 *RCX* 寄存器为 0，就跳转 |
| 带符号数 | SF ! = OF | JL | 如果小于，就跳转 |
| | | JNGE | 如果不大于或不等于，就跳转 |
| | SF = OF | JGE | 如果大于或等于，就跳转 |
| | | JNL | 如果不小于，就跳转 |
| | ZF = 1 or SF ! = OF | JLE | 如果小于或等于，就跳转 |
| | | JNG | 如果不大于，就跳转 |
| | ZF = 0 and SF ! = OF | JG | 如果大于，就跳转 |
| | | JNLE | 如果不小于或不等于，就跳转 |
| 无符号数 | CF = 1 | JB | 如果低于，就跳转 |
| | | JC | 如果进位标志置位，就跳转 |
| | | JNAE | 如果不高于或不等于，就跳转 |
| | CF = 0 | JAE | 如果高于或等于，就跳转 |
| | | JNB | 如果不低于，就跳转 |
| | | JNC | 如果进位标志不置位，就跳转 |
| | CF = 1 or ZF = 1 | JBE | 如果低于或等于，就跳转 |
| | | JNA | 如果不高于，就跳转 |
| | CF = 0 and ZF = 0 | JA | 如果高于，就跳转 |
| | | JNBE | 如果不低于或不等于，就跳转 |

**编程知识：**选择跳转指令的时候一定要注意操作数是无符号数还是带符号数。比方说，如果要判断某个带符号数是否比另一个大，那么应该选择 JG 指令，而不能选择适用于无符号数的 JA 指令，否则程序有可能转入错误的执行路径。

下面是一段用高级语言的 `if` 语句写成的代码，我们现在要通过适当的条件跳转指令将其转换为汇编代码。

```
if (wages >= 50000)
     taxes = 5000
else
     taxes = 2000
```

程序 5-1 演示了其中一种转换办法，并分别给出了适用于 3 种汇编器的 3 个版本。

程序 5-1 （32 位环境下的）条件跳转

GAS

```
.data
wages: .long 46000

.bss
.lcomm taxes, 4

.text
.globl _main
_main:

 movl $50000, %eax
 cmpl %eax, wages
 jae higher
 movl $2000, taxes
 jmp done

higher:
 movl $4000, taxes

done:
pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO,
dwExitCode:DWORD

.data
wages DWORD 46000
taxes DWORD ?

.code
_main PROC

 mov eax, 50000
 cmp wages, eax
 jae higher
 mov taxes, 2000
 jmp done

higher:
 mov taxes, 4000

done:
INVOKE ExitProcess, 0
_main ENDP
END
```

NASM

```
SECTION .data
wages: DD 46000

SECTION .bss
taxes: RESD 1

SECTION .text
global _main
_main:

 mov eax, 50000
 cmp DWORD [wages], eax
 jae higher
 mov DWORD [taxes], 2000
 jmp done

higher:
 mov DWORD [taxes], 4000

done:
mov eax, 1
mov ebx, 0
int 80h
```

请注意，程序 5-1 只用了一个条件跳转指令 JAE，而没有分别通过 JAE 与 JB 两条跳转指令来应对 wages 大于等于 50000 及小于 50000 这两种情况，因为这样写会增加 CPU 的工作量，使其必须再次检查相关的标志位。笔者直接把 else 语句后面的逻辑写在了目标为 higher 的条件跳转指令 JAE 之后，意思是先把 wages 变量与 50000 相对比，如果大于或等于就跳转到 higher 标签，否则，直接执行 JAE 下方的指令，这实际上也就相当于执行了 else 语句后面的逻辑。执行完之后，用 JMP 指令无条件地跳转到 done 标签，从而把实现 if 逻辑所用的 higher 代码块给跳过去。假如分别使用 JAE 指令与 JB 指令去判断大于等于 50 000 及小于 50 000 的情况，就要多做一次检测。这个例子巧妙地安排了指令之间的顺序，从而省去了这次检测。

### 5.3.3 复合条件

讲完条件跳转指令之后，现在可以看看复合条件了，这些逻辑要比前面那些例子稍微复杂一点。C++ 代码可以通过 &&（Logical AND，逻辑与）及 ||（Logical OR，逻辑或）这两种运算符来构造**布尔逻辑表达式**（Boolean logical expression），从而表达复合条件。

```
if ( a > b && b > c)
      x = 1;
```

```
if ( a > b || b > c)
      x = 1;
```

上面两段 C++ 代码分别在 if 语句中采用 && 及 || 操作符，实现了两种不同形式的复合逻辑。这些逻辑在汇编代码中可以通过**短路求值法**（short-circuit evaluation）来实现，如范例 5-1 所示（当然了，C++ 也用这个办法）。此办法的原理是，如果复合条件中的第一个小条件已经完全能够决定整个测试的结果，那就不用再判断第二个小条件了，只有在前者还不足以决定结果的时候才去判断后者。

范例 5-1　复合条件（采用遵循 Intel 语法的汇编代码来实现）

| && (Logical AND) | \|\| (Logical OR) |
|---|---|
| cmp ax, bx　; first expression<br>jbe next　; quit if false<br>cmp bx, cx　; second expression<br>jbe next　; quit if false<br>mov x, 1　; both are true<br>next: | cmp ax, bx　; first expression<br>ja L1　; if true skip to L1<br>cmp bx, cx　; second expression<br>jbe next　; false: skip to next<br>L1: mov x, 1　; true result<br>next: |

在 && 表达式的例子中，只有当左、右两个小条件都成立时整个条件才能成立（请回忆 AND 操作符的真值表）。于是，我们先判断 *ax* 的值是否低于或等于 *bx*，如果是，那说明左侧的小条件（ a > b ）不成立，此时无论右侧小条件是否成立，整个复合表达式的值都会是 False，因此没有必要去判断后者了，而是可以直接跳到 next 标签。只有当左侧的小条件成立（也就是 *ax* 的值高于 *bx*）时，我们才需要判断右侧的小条件是否同样成立。

|| 表达式的例子与 && 表达式稍有不同，这次判断的不是左侧小条件是否为 False（假），而是判断它是否为 True(真)。如果是，那就跳到 L1 标签，并通过 MOV 指令将 x 设置成 1。之所以能这样做，是因为只要左侧的小条件成立，那么无论右侧是否成立，整个表达式的值都会是 True，因此没有必要再去判断后者了。在这两个例子中我们都尽可能用数量最少的指令来完成任务，以求提升程序的效率。

## 5.4　重复执行

**重复**（repetition）也叫作循环（looping），意思是多次执行预先写好的某一组指令，具体的执行次数通常由某个计数器变量或条件来控制。比方说，在高级语言中可以用 for 循环及 while 循环来表示重复执行的控制结构。

```
for (int x = 0; x < 5; x++)          while (answer == "yes")
     Sequence of instructions              Sequence of instructions
```

这些结构所用到的关键字及格式，在汇编语言中不一定能找到对应的指令或命令，不过，我们还是可以通过适当的 x86 指令实现出相同的效果。

### 5.4.1　用 CX/ECX/RCX 计数器实现循环

由计数器所控制的循环很容易就能用汇编语言的 LOOP 指令实现出来。该指令会把某个“C 系列的寄存器”（也就是 *cx/ecx/rcx*）当作**递减计数器**（decrementing counter）来用，每循环一次就将该计数器的值减 1。值降为 0 的时候，结束循环并继续执行 LOOP 下方的指令。

> **编程知识：** *16 位环境下的循环计数器是 cx 寄存器，32 位环境下是 ecx 寄存器，64 位环境下是 rcx 寄存器。*

LOOP 指令的写法在 3 种汇编器中是相同的，都需要带有标签。

GAS/MASM/NASM

```
LOOP LABEL      ; LOOP sumLoop
```

程序 5-2 是个 64 位的汇编程序，它实现了高级语言中的 for 循环。首先要注意：我们通过按位的 XOR 指令把 *rax* 寄存器的值设置成了 0，这种做法在汇编代码中经常用来清零。

执行到 LOOP 指令时，程序会自动做 3 件事，第一，把 *rcx* 寄存器的值减 1，第二，判断它是否不等于 0，第三，如果不等于 0，那就跳到 LOOP 指令所指定的标签，否则，开始执行 LOOP 下方的那条指令。程序 5-2 总共要循环 5 次，每次都要给 *rax* 加 1，因此它的最终值是 5。

**程序 5-2 （64 位的）循环程序**

GAS

```
.text
.globl _main
_main:

xorq %rax, %rax
movq $5, %rcx

myLoop:
   incq %rax
   loop myLoop

movq $0x2000001, %rax
movq $0, %rdi
syscall
.end
```

MASM

```
extrn ExitProcess : proc

.code
_main PROC

xor rax, rax
mov rcx, 5

myLoop:
  inc rax
  loop myLoop

mov rcx, 0
call ExitProcess
_main ENDP
END
```

NASM

```
SECTION .text
global _main
_main:

xor rax, rax
mov rcx, 5

myLoop:
  inc rax
  loop myLoop

mov rax, 60
xor rdi, rdi
syscall
```

**学习指南：** 单步（step-by-step）地执行汇编程序有助于理解 LOOP 指令的运作方式。用你习惯的开发环境来调试程序 5-2，在即将执行 XOR 指令的地方添加断点，然后单步地执行程序（也就是每次只执行一条指令），观察 *rax* 及 *rcx* 的内容如何在循环的过程中发生变化。与调试有关的知识请参见附录 B 与附录 D。

单层循环写起来很简单，但是嵌套循环（nested loop）就得小心了，因为每执行一次 LOOP 指令，“C 系列的寄存器”也会递减 1 次，如果草率地把 C++ 循环直接对译成汇编代码，那就有可能发生范例 5-2 所演示的错误。

**范例 5-2 把 C++ 循环错误地翻译成汇编代码**

C++

```
int value = 0;
for (int x = 0; x < 2; x++)
  for (int y = 0; y < 3; y++)
    value++;
```

GAS

```
movl $0, value
movl $2, %ecx
outer:
   movl $3, %ecx
   inner:
      incl value
      loop inner
   loop outer
```

MASM/NASM

```
mov value, 0
mov ecx, 2
outer:
   mov ecx, 3
   inner:
      inc value
      loop inner
   loop outer
```

C++ 代码执行完之后 `value` 变量的值是 6，这是正确的，然而汇编代码的执行效果却不是这样，因为其中有两个错误。首先，外层循环刚刚把计数器的值设置成 2，立刻就让内层循环给改成了 3。其次，内层循环执行 3 次之后 `value` 的值是 3，而 *ecx* 的值则降为 0。此时，`loop inner` 指令发现 *ecx* 的值已经等于 0 了，于是就结束内层循环，继续执行下方的那条指令，也就是 `loop outer`，该指令首先将计数器 *ecx* 减 1，令其从 0 变成 -1，这个数的十六进制写法是 `0xFFFFFFFFh`，问题在于如果按照无符号数来解释这个值，那么它就是 2^32 -1，显然比 0 大（或者说，显然不等于 0），因此程序又跳回了 `outer` 标签，

从而陷入无限循环。这是一种相当恶劣的 bug，引发过许多问题。

> 学习指南：如果值已经降为 0 还要对它做递减，那么自然就会令其变为 -1，可是第 1 章说过，-1 的补码形式的每个二进制位都是 1，如果写成 32 位环境下的十六进制值就是 0xFFFFFFFFh。该值若按照无符号数来解读就是值最大的整数。有个很出名的 bug，发生在《Civilization》(《文明》) 这款游戏中，它会令 Gandhi（甘地）这个本来很平和的人物变得极其好斗，用核武器不停地轰炸城市。与这个 bug（当然你也可以把它叫作游戏的一项特性或功能）有关的信息请参见：http://civilization.wikia.com/wiki/Mohandas_Gandhi_(Civ1)

要想修复范例 5-2 中的错误，必须在每次到达内层循环时把 *ecx* 的值保存起来，这样就能够在每次执行完内层循环后将该值恢复过来，以便如实地反映外层循环的执行进度。范例 5-3 用正确的汇编代码表达了 C++ 的嵌套循环。这段代码假设数据段中已经定义了名为 `counter` 的变量。

**范例 5-3　把 C++ 循环正确地翻译成汇编代码**

| C++ | GAS | MASM/NASM |
|---|---|---|
| `for (int x = 0; x < 2; x++)`<br>`  for (int y = 0; y < 3; y++)`<br>`  value++;` | `movl $2, %ecx`<br>`outer:`<br>`  movl %ecx, counter`<br>`  movl $3, %ecx`<br>`  inner:`<br>`  incl value`<br>`  loop inner`<br>`  movl counter, %ecx`<br>`  loop outer` | `mov ecx, 2`<br>`outer:`<br>`  mov counter, ecx`<br>`  mov ecx, 3`<br>`  inner:`<br>`  inc value`<br>`  loop inner`<br>`  mov ecx, counter`<br>`  loop outer` |

这个例子告诉我们，使用 LOOP 指令的时候必须多加小心。除了与嵌套循环以及 LOOP 指令的默认行为有关的问题之外还有一些做法也有可能引发错误，例如：在循环内部使用了某条指令，而该指令会自动在某个“C 系列的寄存器”里保存运算结果，或是在循环内部让某个“C 系列的寄存器”肩负了其他的职责。为了避免由这些做法所引起的困惑，开发者可以自己定义计数器，并通过 CMP 指令来实现循环。

### 5.4.2　用开发者自定义的计数器实现循环

还有一种方式，也可以像范例 5-3 那样把 C++ 的 `for` 循环翻译成汇编代码，这就是范例 5-4 所演示的办法。它不用 LOOP 指令，而是通过 CMP 指令来判断开发者自己定义的计数器变量（也就是 `x` 与 `y`）是否为 0，从而决定是终止循环还是再执行一次。这段代码假设程序中已经定义了名为 `x`、`y` 及 `value` 的变量。每执行一次循环，就把相应的计数器变量（`x` 代表外层循环，`y` 代表内层循环）减 1，并通过 JNE（Jump if Not Equal）指令来判断其是否不等于 0，如果不等于就跳回循环开头。

范例 5-4 中有两个地方要说明。第一，汇编代码与 C++ 代码是按照相反的方式来操纵计数器的，前者是递减，后者是递增。当然你也可以按照自己喜欢的方式来编写这两种代码。

范例 5-4 用 CMP 指令实现循环

| C++ | GAS | MASM/NASM |
|---|---|---|
| for (int x = 0; x < 2; x++)<br>  for (int y = 0; y < 3; y++)<br>  value++; | movl $2, x<br>outer:<br>  movl $3, y<br>  inner:<br>  incl value<br>  decl y<br>  cmpl $0, y<br>  jne inner<br>  decl x<br>  cmpl $0, x<br>  jne outer | mov x, 2<br>outer:<br>  mov y, 3<br>  inner:<br>  inc value<br>  dec y<br>  cmp y, 0<br>  jne inner<br>  dec x<br>  cmp x, 0<br>  jne outer |

第二，这段代码是用变量充当计数器的，而汇编代码的一项主要优势则体现在它能够操纵 CPU 中的寄存器，使我们不必浪费内存空间去创建变量。用寄存器取代内存变量能够提升执行速度，而且第 2 章也说过，访问后者中的数据要花费更多的时钟周期。尽管编译器与 CPU 算法有可能在幕后自动执行优化，但你不应该指望这种情况一定会发生。你还是应该自己想办法在汇编代码中做优化。

范例 5-5 对范例 5-4 中的代码做了优化，用寄存器来取代变量。

范例 5-5 优化后的循环代码

| GAS | MASM/NASM |
|---|---|
| xorl %eax, %eax<br>movl $2, %ebx<br>outer:<br>  movl $3, %ecx<br>  inner:<br>  incl %eax<br>  decl %ecx<br>  cmpl $0, %ecx<br>  jne inner<br>  decl %ebx<br>  cmpl $0, %ebx<br>  jne outer | xor eax, eax<br>mov ebx, 2<br>outer:<br>  mov ecx, 3<br>  inner:<br>  inc eax<br>  dec ecx<br>  cmp ecx, 0<br>  jne inner<br>  dec ebx<br>  cmp ebx, 0<br>  jne outer |

执行完上述代码后，最终的结果 6 保存在 *eax* 寄存器里，而不是像范例 5-4 那样保存在 value 变量中。

**学习指南：还有一种办法也能够临时地保存数值，就是用 PUSH 及 POP 指令来操作栈。我们会在第 6 章讲到。**

高级语言里的另一种循环结构叫作 while 循环。范例 5-6 用两种方法实现了 C++ 语言的 while 循环，每种方法所用的跳转指令各有不同。

第一种方法（也就是 v1）与 C++ 代码的意思稍有不同。C++ 代码要判断 x 是否**小于** 50，而这段代码则判断 x 是否**大于或等于** 50，这样做可以提升循环的效率。第二种办法（也就是 v2）和 C++ 代码的意思一样，都在判断 x 是否小于 50，不过，由于汇编代码是一条一条往下执行的，因此，在 JB 指令后面还要再写一个条件与之相反的指令（例如此处的 JAE），使得程序能够在 x 大于或等于 50 的时候直接跳转到 done 标签，而不会继续往下进入 addone 代码块。

范例 5-6　用汇编代码实现 while 循环

| C++ | GAS v1 | NASM/MASM v1 | GAS v2 | NASM/MASM v2 |
|---|---|---|---|---|
| x = 30;<br>while (x < 50)<br>  x++; | movl $30, %eax<br>while_loop:<br>  cmpl $50, %eax<br>  jae done<br>  incl %eax<br>  jmp while_loop<br>done: | mov eax, 30<br>while_loop:<br>  cmp eax, 50<br>  jae done<br>  inc eax<br>  jmp while_loop<br>done: | movl $30, %eax<br>while_loop:<br>  cmpl $50, %eax<br>  jb addone<br>  jae done<br>addone:<br>  incl %eax<br>  jmp while_loop<br>done: | mov eax, 30<br>while_loop:<br>  cmp eax, 50<br>  jb addone<br>  jae done<br>addone:<br>  inc eax<br>  jmp while_loop<br>done: |

由于范例 5-3 至范例 5-6 中的代码都不是完整的汇编代码，因此，下面给出程序 5-3 与程序 5-4 这两段完整的程序，以演示怎样在 64 位环境下用 3 种汇编器分别编写嵌套的 for 循环及单层的 while 循环。程序 5-3 里的 *rax* 最终值是 6，程序 5-4 里的 rax 最终值是 50（这里所说的最终值都是当程序即将通过例行的代码来完成退出时的值）。

程序 5-3　在 64 位环境下实现嵌套的 for 循环

GAS

```
.text
.globl _main
_main:

xorq %rax, %rax
movq $2, %rbx
outer:
  movq $3, %rcx
  inner:
    incq %rax
    decq %rcx
    cmpq $0, %rcx
    jne inner
  decq %rbx
  cmpq $0, %rbx
  jne outer

movq $0x2000001, %rax
movq $0, %rdi
syscall
.end
```

MASM

```
extrn ExitProcess : proc

.code
_main PROC

xor rax, rax
mov rbx, 2
outer:
  mov rcx, 3
  inner:
    inc rax
    dec rcx
    cmp rcx, 0
    jne inner
  dec rbx
  cmp rbx, 0
  jne outer

mov rcx, 0
call ExitProcess
_main ENDP
END
```

NASM

```
SECTION .text
global _main
_main:

xor rax, rax
mov rbx, 2
outer:
  mov rcx, 3
  inner:
    inc rax
    dec rcx
    cmp rcx, 0
    jne inner
  dec rbx
  cmp rbx, 0
  jne outer

mov rax, 60
xor rdi, rdi
syscall
```

程序 5-4　在 64 位环境下实现 while 循环

GAS

```
.text
.globl _main
_main:

movq $30, %rax
while_loop:
  cmpq $50, %rax
  jae done
  incq %rax
  jmp while_loop
done:
```

MASM

```
extrn ExitProcess : proc

.code
_main PROC

mov rax, 30
while_loop:
  cmp rax, 50
  jae done
  inc rax
  jmp while_loop
done:
```

NASM

```
SECTION .text
global _main
_main:

mov rax, 30
while_loop:
  cmp rax, 50
  jae done
  inc rax
  jmp while_loop
done:
```

| GAS | MASM | NASM |
| --- | --- | --- |
| movq $0x2000001, %rax<br>movq $0, %rdi<br>syscall<br>.end | mov rcx, 0<br>call ExitProcess<br>_main ENDP<br>END | mov rax, 60<br>xor rdi, rdi<br>syscall |

由此可见，采用汇编语言开发程序的时候也能够像其他语言那样，通过适当地安排指令写出既简洁又高效的代码。

## 5.5 小结

这一章讨论了怎样通过布尔运算指令在二进制位的层面上操纵数据，还讲解了如何用各种办法给程序添加复杂的逻辑，例如令其能够在不同的条件下执行不同的指令。将这种依据条件而执行的逻辑与那种反复执行某段代码的循环逻辑结合起来，可以完成许多编程任务。笔者演示了怎样通过各种跳转指令以不同的方式操控程序的流程。凭借现在学到的这些内容，你应该能够写出较为复杂的汇编程序了。后面各章还要讲解更复杂的话题与技巧，所以，你必须先透彻地理解第 1 ～ 5 章的内容。

## 5.6 关键术语

AND
Boolean bitwise operations（按位布尔运算）
Boolean logical expressions（逻辑布尔运算）
branching（分支）
CMP
counter（计数器）
JMP
LOOP
NOT
OR
repetition (looping)（重复执行（循环））
short-circuit evaluation（短路求值法）
TEST
XOR

## 5.7 代码回顾

| | |
| --- | --- |
| AND | 按位的 AND 运算 |
| CMP | 默认会把来源操作数与目标操作数相减，使得后续指令可以通过相关标志位来执行操作 |
| JA | 如果高于，就跳转 |
| JAE | 如果高于或等于，就跳转 |
| JB | 如果低于，就跳转 |
| JBE | 如果低于或等于，就跳转 |
| JC | 如果进位标志置位，就跳转 |
| JCXZ | 如果 *CX* 寄存器为 0，就跳转 |
| JE | 如果相等，就跳转 |
| JECXZ | 如果 *ECX* 寄存器为 0，就跳转 |
| JG | 如果大于，就跳转 |
| JGE | 如果大于或等于，就跳转 |
| JL | 如果小于，就跳转 |
| JLE | 如果小于或等于，就跳转 |
| JMP | 无条件跳转 |

（续）

| | |
|---|---|
| JNA | 如果不高于，就跳转 |
| JNAE | 如果不高于或不等于，就跳转 |
| JNB | 如果不低于，就跳转 |
| JNBE | 如果不低于或不等于，就跳转 |
| JNC | 如果进位标志不置位，就跳转 |
| JNE | 如果不相等，就跳转 |
| JNG | 如果不大于，就跳转 |
| JNGE | 如果不大于或不等于，就跳转 |
| JNL | 如果不小于，就跳转 |
| JNLE | 如果不小于或不等于，就跳转 |
| JNO | 如果不溢出，就跳转 |
| JNP | 如果奇偶校验位不置位，就跳转 |
| JNS | 如果符号位不置位，就跳转 |
| JNZ | 如果不为 0，就跳转 |
| JO | 如果溢出，就跳转 |
| JP | 如果奇偶校验位置位，就跳转 |
| JRCXZ | 如果 *RCX* 寄存器为 0，就跳转 |
| JS | 如果符号位置位，就跳转 |
| JZ | 如果为 0，就跳转 |
| LOOP | 采用 CX/ECX/RCX 做计数器来执行循环 |
| NOT | 按位的 NOT 运算 |
| OR | 按位的 OR 运算 |
| TEST | 默认会对两个操作数执行 AND 运算 |
| XOR | 按位的 XOR 运算 |

## 5.8　习题

### 填空题

1. 对 1 与 0 做按位的 AND 运算，其结果是_____。
2. 10100100_____11010101 = 01110001。
3. 判断奇偶的常用办法，是通过_____运算来检查受测数字的最低有效位是否为 1。
4. 要想把多个标志位合起来表示在同一个变量中，可以通过_____运算实现。
5. 要想在汇编代码里实现分支必须先用_____来标注代码块。
6. _____指令能够无条件地把程序引向另一块代码。
7. 执行条件跳转指令之前必须先用_____指令来比较两个操作数。
8. CMP 指令会自动将两个操作数_____，并根据计算结果来设置处理器的相关标志位。
9. JNGE 指令的意思是，如果第一个操作数_____第二个操作数，那就跳转到某个标签。
10. 执行 JNA 条件跳转指令之前，必须先对两个_____符号的操作数做比较。
11. 在 CMP 指令后面，可以用_____指令来实现“如果 Sign Flag（符号位）为 0，那么就跳转到某个标签”的逻辑。
12. LOOP 指令可以方便地实现出基于计数器的循环结构，该指令是用_____来做计数器的。
13. 在 32 位环境下，LOOP 指令会自动对 *ecx* 做_____。
14. 开发者可以结合_____与_____指令来自行控制计数器并实现循环。

15. 用_____取代_____能够令程序执行得更快。

**判断题**

1. 按位的 AND 操作，经常用于把数值中的二进制位设置成 1。
2. JMP 指令无须根据处理器中的特定标志位来决定自己的行为。
3. JNZ 指令适用于带符号的数值。
4. LOOP 指令能够自动保留计数器的值，使得开发者可以在循环内部随意运用充当计数器的寄存器（也就是 *cx*/*ecx*/*rcx* 寄存器）。
5. 在汇编代码中必须分别用两个条件跳转指令来表达 `if` 与 `else` 这两种情况。

## 5.9 作业

### 5.9.1 斐波那契又来了

编写程序，对斐波那契数列（Fibonacci sequence）中处在 0 至 1 000 000 这个范围内的所有奇数求和。

### 5.9.2 字母的出现次数

编写程序，计算字母 a（小写或大写都算）在这个句子里出现了几次：`Assembly is the best programming language!`

### 5.9.3 准备输出（挑战题）

编写程序，把某个 32 位的数值（例如 `0xFFFFh`）由普通的形式转换成字节数组的形式，以便通过系统调用将其显示到屏幕上。你需要借助循环把数值中的各个数位转换成相应的 ASCII 码以供输出。这道题还是要求通过 `int 80h` 这样的系统调用把字符打印到控制台，而不允许使用 `printf` 等外部函数。

第 6 章

Assembly Programming and Computer Architecture for Software Engineers

# 函　数

## 目标

- 理解实现函数时所需遵循的步骤
- 对比各种函数调用约定
- 操纵栈内存
- 用汇编语言创建函数

## 网络资源

**维基资源**

- http://en.wikipedia.org/wiki/Calling_convention
- http://en.wikipedia.org/wiki/X86_calling_conventions
- http://en.wikibooks.org/wiki/X86_Disassembly/Calling_Conventions
- http://wiki.osdev.org/Calling_Conventions

**技术资源**

- https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/LowLevelABI/000-Introduction/introduction.html（OS X ABI 函数调用指南）
- https://msdn.microsoft.com/en-us/library/k2b2ssfy.aspx（x86 调用约定）
- https://msdn.microsoft.com/en-us/library/ms235286.aspx（x64 调用约定）
- https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI（AMD64 ABI）
- http://www.agner.org/optimize/calling_conventions.pdf（调用约定）

## 6.1　简介

这一章要讲解函数的实现细节，并演示寄存器与内存会怎样在执行函数的过程中发生变化。从底层入手来研究函数是很有帮助的，因为函数是编程语言中的基本结构，而这样的研究能够令软件开发者更好地理解它在这一层面上的实现方式。与函数有关的技术细节可以写上几百页，不过笔者在这里主要是想讲解几种最为常见的调用约定，提供一些范例帮助读者理解函数的运作原理，并告诉大家怎样用汇编语言来编写函数以及高级语言中的决定在底层会产生怎样的影响。

> ⟨!⟩ **提示：过程**（procedure）与**函数**（function）这两个词有时可以互换。有的书把不带返回值的叫作过程，把带有返回值的叫作函数。

## 6.2　栈内存入门

学习函数的调用约定之前，必须先了解**栈内存**（stack memory）的基础知识。这些知识

大家原来在上编程课的时候应该已经学过了，现在复习一遍。栈内存是为自动变量而设的一块区域（这里的自动变量是指局部变量，或者说非动态的变量）。调用函数的时候，需要用栈来保存函数中的局部变量，而函数结束的时候，则需要弃用[⊖]这些变量（因为它们现在已经不在作用域之内了），该函数早前占据的内存可以用来做其他事情。高级语言的一项特征在于它会自行管理栈内存（这有时也叫作运行时栈或运行期栈），相反，汇编语言不会这样做，而是需要你自己去管理。

用汇编语言做底层开发的时候，需要频繁管理栈内存，而且还必须理解操纵栈内存所用的一些指令，以便调试程序。下面列出与栈内存有关的重要事项。

- 栈会在调用函数时增长，并在调用结束时收缩。
- 栈会在创建（或者说推入 / 压入）局部变量时增长，并在弃用（或者说弹出）局部变量时收缩。
- 每个进程或线程的栈，其大小受操作系统限制，例如 Linux/Mac 系统默认是 8MB，Windows 默认是 1MB。
- 每次调用函数（这也包括调用主函数 `main()`，以及递归地调用自身）都会出现对应的**栈帧**（stack frame）。它是栈内存中的一块区域，随着函数调用而产生。
- 栈帧用来保存函数的局部变量。
- 栈内存是向下增长的，也就是说，推入栈中的新内容会出现在地址值较低的位置中。
- 32 位环境下，栈中的每个元素占据 4 个字节，64 位环境下占据 8 个字节。
- 数值默认按照**小端序**入栈（大端序与小端序，请参见第 2 章）。权重最高的字节保存到地址最高的位置上，权重最低的字节保存到地址最低的位置上。例如 32 位的十六进制值 `0x9BFB3701` 在栈内存中的存放顺序是 `01 37 FB 9B`。
- x86_64 的函数都必须按 16 字节对齐（所有平台都是这样）。

下一节要讲解与栈有关的细节。我们会讨论函数的调用约定，并给出用汇编语言所写的代码实例，在这个过程中，笔者将详细解释刚才提到的要点。

## 6.3　x86 与 x86_64 的调用约定

调用约定也叫作调用协议，用来确定函数在底层的实现流程，包括如何传递参数、如何管理栈内存，以及如何返回值。在讨论调用约定的时候，会用到 caller（调用方，主调方）及 callee（被调用方，受调方）这两个词，前者是指主动调用其他函数的函数（calling function，主调函数），而后者则是受到前者调用的函数（called function，被调函数）。在范例 6-1 所演示的 C++ 代码中，`main()` 是主调函数，`sum()` 是被调函数。

**范例 6-1　用 C++ 语言编写求和程序**

```
int sum(int, int);                          // sum function prototype

int main() {                                // main function is caller
    int num1 = 2, num2 = 4, answer;
    answer = sum(num1, num2);               // sum function is callee
    return 0;
}
```

⊖ 原书写的是 dereference，此处并不是对某个指针解除引用，以获取其所指向的值，而是说不再需要使用某个变量了。——译者注

```
int sum(int num1, int num2){
    return num1 + num2;
}
```

### 6.3.1 cdecl（32 位）

**提示：** 这里以 32 位环境为例来总论如何调用函数。这样做可以体现出在 32 位环境下调用函数时所需遵循的约定与 64 位环境有何区别。由于这两种架构都很常见，因此有必要对比它们之间的差异。

cdecl（C declaration）调用约定是 32 位平台最常见的一种约定，因为它是基于 C 语言标准而设立的。一般来说，GCC、Clang 与 Visual Studio 的 C 编译器默认都会使用 cdecl 约定。有些开发环境（例如图 6-1 中的 Visual Studio）允许开发者在项目属性中调整默认的约定方式。cdecl 有 4 项主要特征。

- 参数按照相反的顺序（也就是从右到左的顺序）入栈。
- *eax*、*ecx* 与 *edx* 需要由调用方保存（叫作易失性的寄存器），而其余的通用寄存器则需要由受调方来保存（叫作非易失性的寄存器）。因此，如果想令 *eax*、*ecx* 及 *edx* 寄存器在调用完函数之后还能保持调用之前的值，那么应该在主调函数中先将其保存起来，因为受调函数在执行过程中有可能会修改它们[⊖]。
- 在大多数情况下，返回值放在 *eax* 寄存器里，如果要返回的是浮点数，那么应该放在 *st(0)* 寄存器里。
- 栈由调用方负责清理。

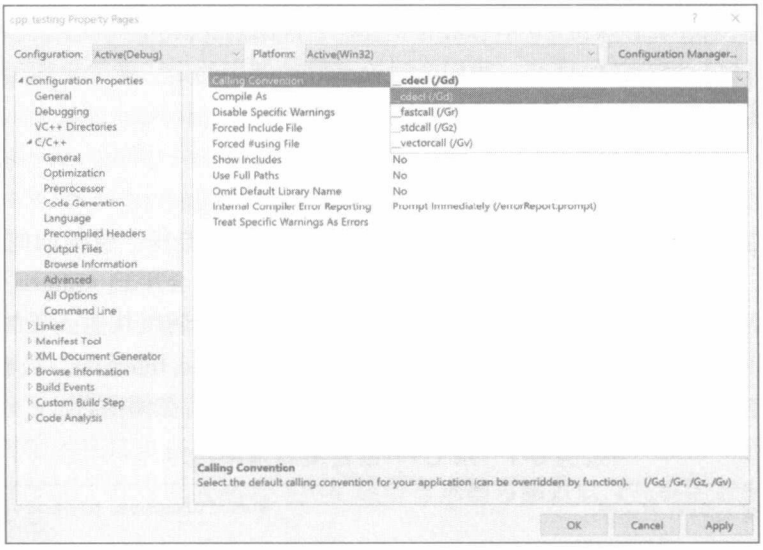



图 6-1　在 Visual Studio 中设置调用约定

C 语言支持参数数量可变的函数。由于这种函数的参数列表长度不固定，因此，受调方并不清楚自己到底接收了几个参数。于是，就要求栈必须由主调方来清理，而且每次调用的

⊖ 另一方面，如果被调函数需要修改这三者之外的其他寄存器，那么应该把寄存器的原值保存起来，并在返回的时候予以恢复，使得主调函数所观察到的值，能够与调用之前相同。——译者注

时候都得清理。C 语言里的 printf() 就属于这样的函数。

程序 6-1 演示了怎样在 x86 环境下按照 cdecl 约定来调用函数。这段汇编程序的含义与范例 6-1 里的 C++ 程序一样，都是对两个参数求和。在下面给出的 3 个版本中，笔者采用 GAS 版本来详细讲解 cdecl 约定。

我们可以用伪代码的形式分 4 个步骤阐述该程序的流程。

1. 声明并初始化变量 num1 及 num2。
2. 执行一些指令，以模拟调用 sum 函数之前所做的计算。
3. 调用 sum 函数，该函数会将 num1 与 num2 相加。
4. 执行一些指令，以模拟调用完 sum 函数之后所做的计算。

**编程知识：**理解函数在底层的实现方式对优化及调试工作很有帮助。在 x86 环境下，可以单独调整每一个函数的调用约定，而且在 x86 及 x86_64 平台中，你可以自行决定是应该把经常用到的操作专门包裹到一个函数中，还是只在每次用到它们的时候重复执行一遍。从底层的角度研究函数，可以了解到递归函数、内联函数、静态函数以及模板函数等机制的优点及缺点，由于大多数开发环境都能在调试程序的时候把反汇编所得到的指令展示出来，因此，这些知识能够令你更加明确地知道程序在各个时间点上都做了哪些事情，从而帮助你尽快查找并修复 bug。

**程序 6-1　在 x86 环境下用汇编语言实现求和**

GAS

```
.data
num1: .long 2
num2: .long 4

.text
.globl _main, _sum
_main:

movl $10, %eax
decl %eax
movl $5, %ebx

pushl num2
pushl num1
calll _sum
addl $8, %esp

addl %ebx, %eax
decl %eax

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80

_sum:
pushl %ebp
movl %esp, %ebp
pushl %ebx
movl 8(%ebp), %ebx
movl 12(%ebp), %eax
addl %ebx, %eax
```

MASM

```
.386
.MODEL FLAT, C
ExitProcess PROTO stdcall,
dwExitCode:DWORD

.data
num1 DWORD 2
num2 DWORD 4

.code
_main PROC

mov eax, 10
dec eax
mov ebx, 5

push num2
push num1
call _sum
add esp, 8

add eax, ebx
dec eax

INVOKE ExitProcess, 0
_main ENDP

_sum PROC
push ebp
mov ebp, esp
push ebx
mov ebx, [ebp + 8]
```

NASM

```
section .data
num1: dd 2
num2: dd 4

section .text
global _main, _sum
_main:

mov eax, 10
dec eax
mov ebx, 5

push DWORD[num2]
push DWORD[num1]
call _sum
add esp, 8

add eax, ebx
dec eax

mov eax, 1
mov ebx, 0
int 80h

_sum:
push ebp
mov ebp, esp
push ebx
mov ebx, [ebp + 8]
mov eax, [ebp + 12]
add eax, ebx
pop ebx
```

| GAS | MASM | NASM |
| --- | --- | --- |
| popl %ebx<br>popl %ebp<br>retl<br><br>.end | mov eax, [ebp + 12]<br>add eax, ebx<br>pop ebx<br>pop ebp<br>ret<br>_sum ENDP<br><br>END _main | pop ebp<br>ret |

📖 **学习指南：** 要想把这一章的内容学好，你可以试着将刚才的程序 6-1 输入到开发环境中。每写一段就看一下后面的解释，以便了解为什么要这样编代码，以及每段代码都会执行什么样的操作。

```
.data
num1: .long 2
num2: .long 4

.text
.globl _main, _sum
```

GAS 版本的代码是以 .data 段开头的，其中定义了值为 2 的变量 num1 以及值为 4 的变量 num2。这两个变量都是 32 位的。其后出现的是 .text 段，其中包含本程序的可执行指令。该段开头声明了两个全局函数 _main 与 _sum，前者是程序的入口点。

```
movl $10, %eax
decl %eax
movl $5, %ebx
```

接下来这 3 行代码用来模拟程序在调用函数之前所执行的一些运算，以演示寄存器是应该由主调方还是由受调方来保存。首先，把 10 移动到 *eax* 中，然后将该寄存器减 1，最后把 5 移动到 *ebx* 中。此时，*eax* = 9，*ebx* = 5。假如我们要保存的是 *eax*，那么由于它是个应该由主调方来保存的寄存器，因此这项工作必须由主调函数来做，也就是要在调用 _sum 函数之前将 *eax* 推入栈中，以便在调用完 _sum 并返回到 _main 时能够予以恢复。不过，我们这次要保存的不是 *eax*，而是 *ebx*。由于它是个应该由受调方来保存的寄存器，因此，要等程序由 _main 进入 _sum 函数之后再将其推入栈中。

```
pushl num2
pushl num1
calll _sum
addl $8, %esp
```

现在可以正式调用函数了。请注意，cdecl 要求按照反方向（也就是按照从右到左的顺序）把参数通过栈传递给受调函数。如果要调用的是 answer = sum(num1, num2);，那么应该先推入 num2，后推入 num1。按值传递的时候推入的是值本身，按引用传递的时候推入的是值的地址（参见范例 6-2）。传递完参数后，开始用 call 指令来调用 _sum 函数。等 _sum 函数执行完毕，_main 函数必须负责把栈清理干净。由于早前总共向栈中推入了两个 4 字节的值，因此，要给 *esp* 加 8，以便将这两个值“移除”。

**范例 6-2　在 32 位环境下按引用传递变量**

| GAS | MASM | NASM |
| --- | --- | --- |
| .data<br>num1: .long 2<br><br>.text<br>pushl $num1 | .data<br>num1 DWORD 2<br><br>.code<br>push OFFSET num1 | section .data<br>num1: dd 2<br><br>section .text<br>push num1 |

**学习指南：**本章要提到几个前面没有讲过的新指令，例如 PUSH、POP、CALL、RET 及 NOP，这些指令的详情可参阅本章末尾的**代码回顾**一节。我们在正文中讨论范例程序及范例代码时，主要关注的是代码的写法以及与操作有关的一些细节。

当程序 6-1 运行到这里的时候，我们应该看看反汇编之后的指令，并查看栈与寄存器的状态，同时还要了解一下指令所涉及的内存地址。尽管每次运行程序时具体地址会有所不同，但运行的原理是一样的。此外，再次提醒大家，栈是朝着低地址的方向增长的。

图 6-2 演示了参数尚未入栈的时候范例程序 Sum 所处的状态。此时 *eip* 指向 `0x1f86` 这个地址，该地址上的指令（`push 0x2004`）就是即将执行的指令。现在如果单步地执行这条指令，那么就会出现下列变化。

1. *eip* 将推进到 `0x1f8c`，用以指向接下来将要获取并加以执行的指令。

2. `0x2004` 地址上的数值将要推入栈中。看一下这个地址你就会发现，其中的值正是 `04 00 00 00`，也就是 `num2` 变量的取值。

3. *esp* 将会减 4，因为将要推入栈中的值是个 32 位的值，该值需要占据 4 个字节。

图 6-3 演示的是两个参数都入栈之后的程序状态。

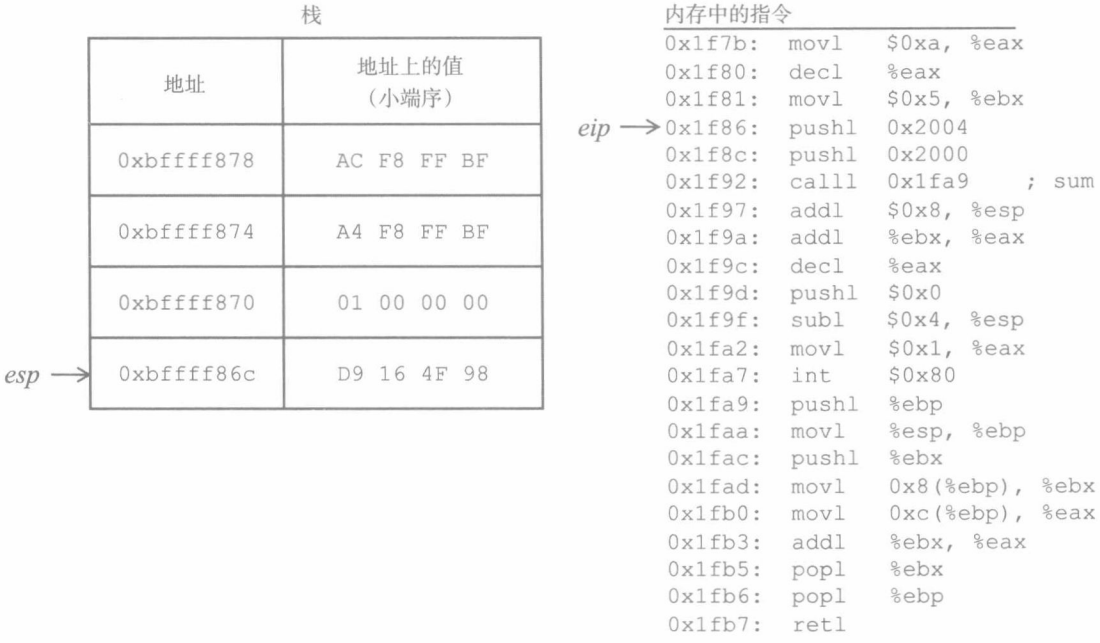



图 6-2　参数尚未入栈时的程序状态

此时即将执行的指令就是 `0x1f92` 地址上的 `calll 0x1fa9`（参见图 6-3 中的 *eip*）。这条 CALL 指令会做下面几件事情。

1. 把 `_main` 函数在执行完 `_sum` 之后接下来应该执行的指令所在的地址推入栈中，以便在调用完 `_sum` 函数后程序能够从这个地方继续往下执行。这实际上就是把 `calll` 指令下方的指令所在的地址推入栈中。具体到本例来说，就是将 `addl $0x8, $esp` 指令所在的地址 `0x1f97` 入栈。

2. 转到受调函数所在的地址，并从这个地址开始执行。具体到本例来说，就是从

0x1fa9 地址上的指令开始执行。

3. 相应地更新 *eip* 与 *esp*。

图 6-4 演示了程序在执行完 CALL 指令之后所处的状态。

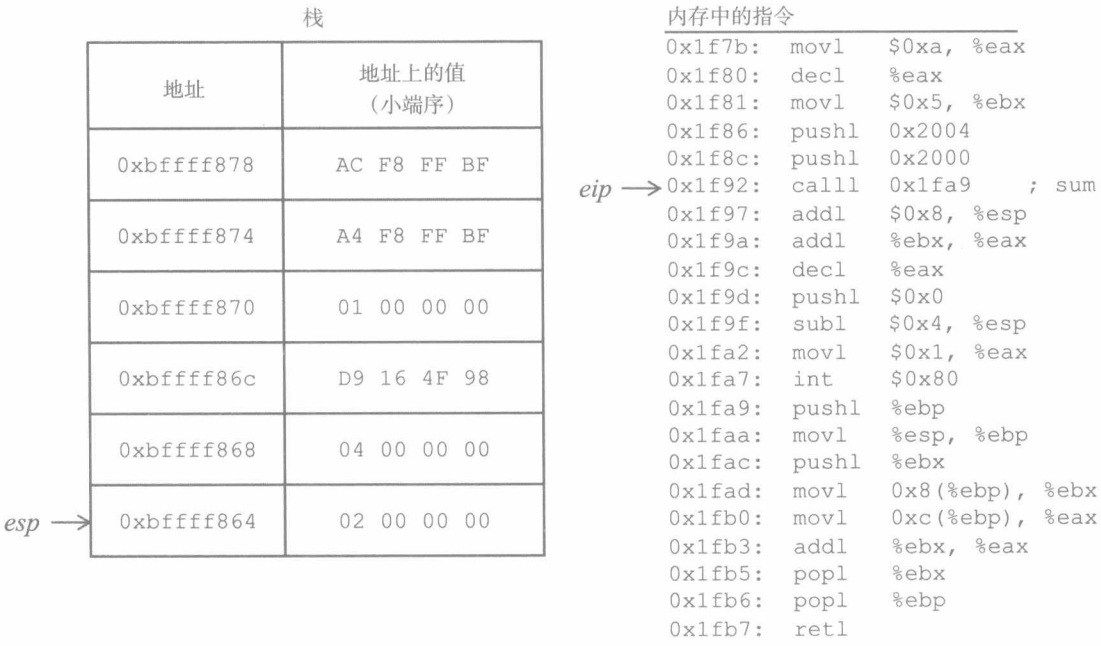



图 6-3　两个参数均已入栈后的程序状态

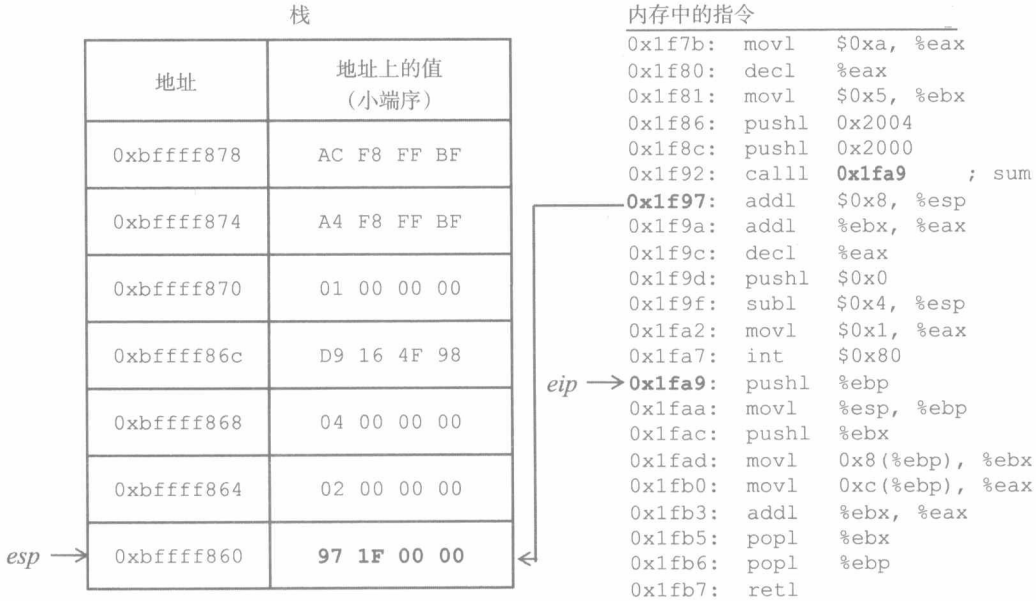



图 6-4　执行完 CALL 指令后的程序状态

进入 _sum 函数之后，需要建立与之对应的栈帧。前面说过，栈帧可以用来存放函数的局部变量，而 *ebp* 寄存器则要指向目前执行的栈帧所在的起始地址。*ebp* 很重要，因为它会给传递进来的参数以及本函数中的局部变量提供参照点，并使得开发者可以由此访问函数中

的数值。与此相对，*esp* 寄存器在 x86 环境下不应该用作访问参数的依据，由于它的职责就是指向栈顶，因此，只要有数值入栈或出栈它就会发生变化。反之，*ebp* 的值在函数执行过程中不会变化，从而可以当作固定的参照点来使用。

**编程知识：**在 32 位环境与 64 位环境下引用局部变量可以采用不同的办法。32 位环境下，一般用 *ebp* 作为基准点来访问参数，而 64 位环境下则不太这样使用。其原因有三条。第一，64 位环境下通常使用寄存器（而不是栈）来传递参数；第二，64 位环境下可以使用 RIP 相对寻址机制（参见附录 F 及第 6 章补充材料）；第三，64 位环境下，系统默认在栈内存中划分一块**暂存空间**，使我们可以直接以 *rsp* 寄存器为基准来引用局部变量。本章稍后会讨论怎样在 64 位环境下实现函数调用。

```
_sum:
pushl %ebp
movl %esp, %ebp
pushl %ebx
movl 8(%ebp), %ebx
movl 12(%ebp), %eax
addl %ebx, %eax
popl %ebx
popl %ebp
retl
```

_sum 函数的前两行代码叫作函数**序言**（function prologue，也称为函数开场），每次调用函数时都要从这些入口代码开始执行。由于主调函数（也就是本例中的 _main）的帧指针需要保存起来以便稍后恢复，因此，我们用 pushl %ebp 指令将其推入栈中。然后，通过 movl %esp, %ebp 指令使 *ebp* 指向当前的栈顶 *esp*。这两条指令合起来就相当于设立了新的栈帧。

本例要使用 *ebx* 寄存器来保存计算结果，而该寄存器是个应该由受调方保存的寄存器，因此，_sum 函数要将其推入栈中，以便稍后予以恢复。

图 6-5 演示了程序在设立了新的栈帧并把 *ebx* 寄存器的值保存起来之后所具备的状态。

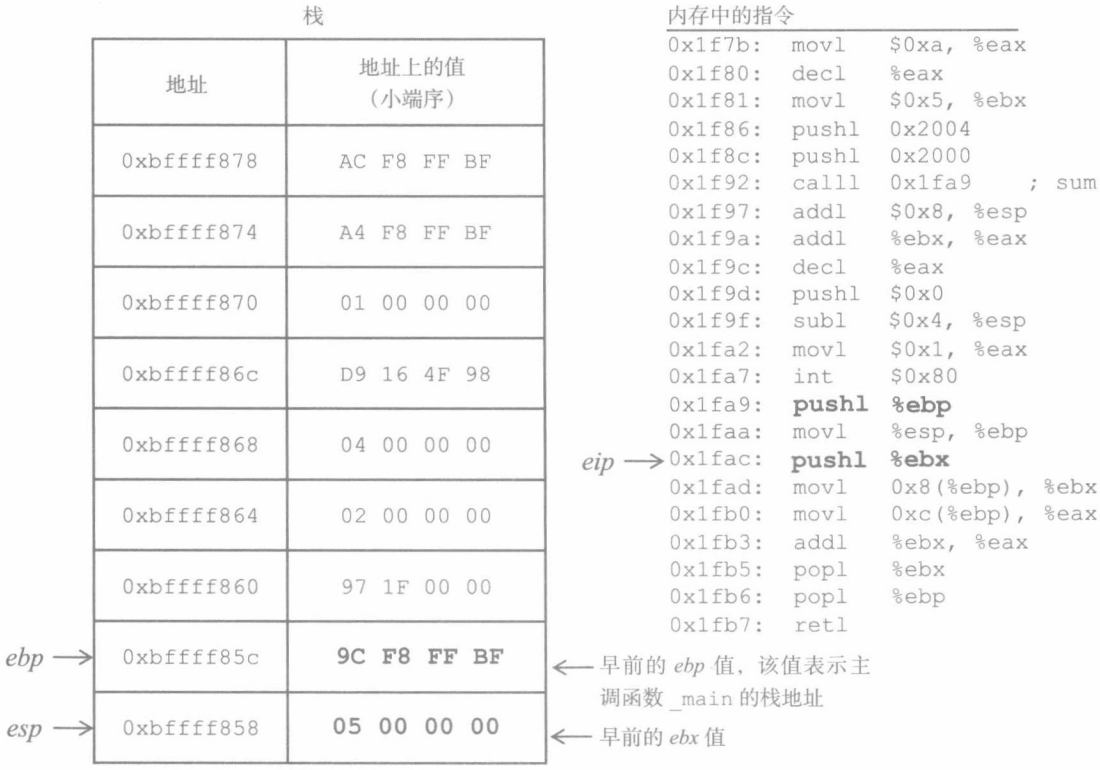



图 6-5　程序在建立起新的栈帧并将 *ebx* 入栈之后的状态

```
movl 8(%ebp), %ebx
movl 12(%ebp), %eax
addl %ebx, %eax

; ebx = 2
; eax = 4
; eax = 6
```

现在可以开始计算两数之和了，这有很多种办法。此处的办法是，把这两个值从栈帧中的对应位置上复制到寄存器里，然后将这两个寄存器的值相加。注意，*ebp* 是引用栈帧时的参照点。早前我们先把调用 _sum 函数所需的第二个及第一个参数推入了栈中，然后 CALL 指令又把调用完函数后所要执行的指令的地址推入了栈中，于是，*ebp* 指向新栈帧的起始地址之后，在它上方总共有三个值可以参照 *ebp* 来加以引用。

如果给 *ebp* 加 4（或者说，从 *ebp* 开始向上退 4 个字节），那么这个位置上的内容就是调用完 CALL 指令后应该执行的指令所具备的地址。如果给 *ebp* 加 8，那么就可以查出调用 _sum 函数时所传入的第一个参数，如果加 12，那么就可以查出第二个参数。由于 _sum 函数返回的是个整数值，因此，可以将其保存到 *eax* 寄存器里，以供主调方 _main 函数取用。返回值如果不是整数那么就比较复杂了，请参见本书附录 F 及本章的网络资源。

```
popl %ebx
popl %ebp
retl
```

接下来，准备把控制权返还给 _main 函数。刚开始执行 _sum 函数时，我们曾经用 PUSH 操作把 *ebx* 的原值推入栈中，于是，现在可以通过 POP 操作将其弹出，使得 *ebx* 恢复到调用 _sum 函数之前的值，也就是 5。然后，执行函数尾声[⊖]（function epilogue，函数收场），也就是退出函数时的例行代码。由于程序要从 _sum 返回到 _main，因此，必须恢复 *ebp* 的值，使它变得与 _main 函数的帧指针相同。这两条 POP 指令实际上是把栈中的两个元素相继弹出，从而令 *esp* 向上退了 8 个字节（每个元素占据 4 个字节），也就是令 *esp* 的值比原来大 8。

现在来到 RET 指令。该指令默认会把栈顶值弹出到 *eip* 中，使其指向调用完函数后应该执行的指令所在的地址，这个地址正是由早前调用此函数的 CALL 指令所推入的。具体到本例来说，*retl* 指令会把早前的 *calll* 指令所推入的地址 0x1f97 从栈中弹出来，并将其存入 *eip*，这样的话，*eip* 就可以指向 CPU 接下来应该执行的指令了。

图 6-6 演示了程序在执行完 RET 指令后所处的状态。现在，*eax* 的值等于 6，也就是 num1 与 num2 这两个参数值之和。*ebx* 的值等于 5，这与调用 _sum 函数之前是相同的。

```
addl $8, %esp
```

cdecl 式的函数调用约定要求栈必须由主调方（也就是本例中的 _main）来清理，也就是说，_main 函数必须把它早前推送给 _sum 过程的两个参数丢弃掉。由于早前向栈中推入了两个参数，而每个参数都占据 4 个字节，因此，清理栈的时候必须给 *esp* 加 8。向栈中推入数据会导致 *esp* 朝着内存地址较低的方向移动，这相当于给它做减法，而反过来说，如果令 *esp* 朝着内存地址较高的方向移动，或者给它做加法，那么实际效果就相当于丢弃栈中的数据。

⊖ 为了使文字读起来灵活一些，在与 Prologue（开场）对照时，将 epilogue 译作收场。单说 epilogue，则译为尾声。——译者注

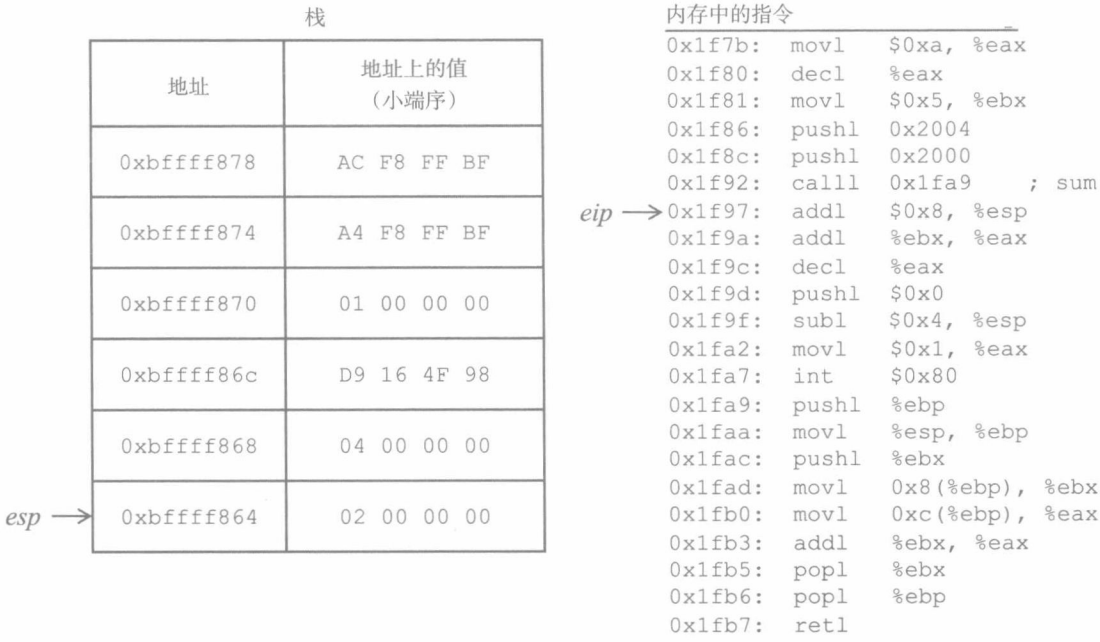



图 6-6　程序执行完 RET 指令之后的状态

```
addl %ebx, %eax
decl %eax

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
```

最后几行代码用来模拟 _main 函数在调用完 _sum 之后所执行的一些运算。本例是把 *ebx* 的值加到 *eax* 中，然后令 *eax* 递减，使其最终值变为 10。

GAS 版本的程序在退出之前需要做一套例行的操作。首先，把表示退出状态码（或者说返回码）的值 0 推入栈中，然后，令栈增长 4 个字节（或者说，用 4 个额外的字节来填充栈），接下来，把代号 1 放到 *eax* 寄存器里，用以表示即将退出程序，最后，向内核发出中断请求。这套例行操作的详情可参见第 10 章。

### 6.3.2　stdcall（32 位）

还有一种常见的约定方式叫作 stdcall，它因为被 Windows API 使用而广为人知。一般来说，它的规则与 cdecl 是相同的，只是在一个地方有所区别（参见下列 4 项中的最后一项）。

- 参数按照相反的顺序（也就是从右到左的顺序）入栈。
- *eax*、*ecx* 与 *edx* 应该由调用方保存，而其余的通用寄存器则应该由受调方来保存。
- 在大多数情况下，返回值放在 *eax* 寄存器里，如果要返回的是浮点数，那么应该放在 *st(0)* 寄存器里。
- **栈由受调方负责清理**。

stdcall 的好处是，受调函数可以在运行 RET（返回）指令的时候顺便把栈清理干净，从而使主调函数可以少写一些代码。由于清理栈所用的代码是写在受调函数中的，因此，每次调用这个函数时，这些代码都能得到执行，而不用在主调函数里重复一遍。栈能够由受调方清理，是因为 stdcall 不允许参数个数可变的函数，因此，受调方可以知道自己接收了几个参数，从而正确地把栈清理干净。

stdcall 的 RET 指令与 cdecl 稍有不同，它必须指出栈中有多少个字节的内容需要移除。假如程序 6-1 中的 MASM 版本要改用 stdcall 式的调用约定，那么必须修改下面这两个地方。

1. 把 `_main` 函数里的 `add esp, 8` 这一行删掉。

2. 将 `_sum` 函数里的返回指令改成 `ret 8`。

笔者是以 cdecl 约定为例来讲解如何实现函数调用的，这是因为，大多数 C/C++ 编译器采用的都是这种约定。它的好处在于，可以支持参数个数不固定的函数。尽管 stdcall 在 Windows API 之外的场合很少使用，但你还是可以在其他平台上把函数定义成 `__stdcall` 的形式，从而采用这种约定方式来调用，只不过具体的写法在各平台有所不同。这可以参考本章后面给出的范例 6-11。

### 6.3.3 x86_64（64 位）

64 位架构出现之后，函数的调用约定就变得更加规范了，它是在同一套大的约定方式之下形成了两种小的风格，一种是 Microsoft x64，另一种是 System V AMD64（AMD64）。x86_64 约定试图提升函数的执行速度，它允许程序通过寄存器来传递某些参数，而不一定非要通过栈来做（这与 32 位模式下的 fastcall 约定是类似的）。表 6-1 指出了这两种 64 位的函数调用约定所具备的特征。

> **提示：** fastcall（范例 6-11 给出了具体写法）也是一种可以通过寄存器来传递数值的约定方式，它运用在 32 位环境中。详情可参考本章的网络资源。

表 6-1　x86_64 调用约定

| 约　定 | 可以用来传递参数的寄存器 | 由主调方保存的寄存器 | 由受调方保存的寄存器 | 特　征 |
|---|---|---|---|---|
| Microsoft x64 | rcx/zmm0<br>rdx/zmm1<br>r8/zmm2<br>r9/zmm3 | rax,rcx,rdx<br>r8-r11,<br>st(0)-st(7)<br>k0-k7<br>xmm0-xmmk,<br>所有的 ymm 寄存器，<br>所有的 zmm 寄存器（除了低 128 位）<br>(xmm0-xmm5) | rbx,rsi,rdi, rbp,<br>r12 - r15,<br>xmm6-xmm15,<br>以及 zmm 寄存器的低 128 位 | 栈按 16 字节对齐<br>栈中不仅保存有返回地址与局部变量，而且其后还预留着 32 个字节的 shadow space<br>可以通过参数寄存器发送 4 个参数 |
| System V AMD64 (Linux, BSD, Mac,Gnu) | rdi<br>rsi<br>rdx<br>rcx<br>r8<br>r9<br>zmm0-7 | rax,rcx,rdx,rsi,rdi,<br>r8-r11,<br>st(0)-st(7),<br>k0-k7,<br>xmm0-xmm15,<br>ymm0-ymm15,<br>zmm0-zmm31 | rbx,rbp,<br>r12-r15 | 栈按 16 字节对齐<br>栈中不仅保存有返回地址，而且其后还预留着 128 个字节的 red zone<br>可以通过参数寄存器发送 14 个参数 |

在 Windows 64 位环境下，最多可以将 4 个参数按照表 6-1 中的顺序通过寄存器传递给受调函数。比方说，如果受调函数有 3 个参数，分别是 `(int, float, int)` 类型，那么主调函数就可以通过 *rcx*、*zmm1* 及 *r8* 寄存器来传递这 3 个参数。在 AMD64 系统中，最多可以通过寄存器发送 14 个参数。请注意，这两种调用规范所涉及的参数寄存器，其顺序是

有区别的。

在这两种情况下，如果要传递的参数比参数寄存器的数量还多，那么其余的参数依然遵循从右至左的顺序入栈。范例 6-3 演示了怎样按照 Microsoft x64 调用约定来传递参数。

**范例 6-3　按照 Microsoft x64 调用约定来传递参数**

```
int someFunction(int, int, float, float, char, char);  // prototype
x = someFunction(num1, num2, num3, num4, letter1, letter2);  // call
```

| 参数 | 传递媒介 |
| --- | --- |
| int num1 | *rcx* |
| int num2 | *rdx* |
| float num3 | *zmm2* |
| float num4 | *zmm3* |

| 参数 | 传递媒介 |
| --- | --- |
| char letter2 | 栈 - 第一个入栈 |
| char letter1 | 栈 - 第二个入栈 |

64 位模式下的函数必须按 16 字节对齐。x86_64 支持**流式 SIMD 扩展**（Streaming SIMD Extensions，SSE），这是一套有助于并行处理的指令。SSE 的操作一般都是 128 位的，将这个数字除以 8（因为 x86_64 的指令是按照字节来计算长度的，因此要将位换算成字节）就相当于 16 字节，这意味着 SSE 操作必须按 16 字节对齐。为了确保使用 SSE 指令的函数不会出现栈对齐错误，所有的栈帧都应该向 16 字节处对齐。此外，由于 CALL 指令默认会把 8 个字节的地址推入栈中，因此，在很多情况下，我们都需要再向栈中推入 8 字节，从而实现 16 字节对齐。

> **提示：**《Intel 64 and IA-32 Architectures Optimization Reference Manual》（http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf）是一份很好的参考资料，能帮助你理解优化与对齐问题，其中的 4.4.2 节，讲述了 SIMD 操作的栈对齐问题。本书的第 8 章也会详细讲解对齐以及 SSE 指令。

有很多办法都能使栈按照 16 字节对齐，然而这些办法在效果上可能略有区别，比方说，有的办法或许速度更快一些。其中一个办法是在进入函数时给 *rsp* 减 8，不过这也要求我们必须在退出函数时给它加 8。还有一个常见的办法是对 *rsp* 与 `-16`（写成 8 位的二进制形式，就是 `0xF0`）做 AND 运算，使得 *rsp* 退回到离它最近且值为 16 的倍数的地址（此地址的十六进制形式总是以 0 结尾）上。比方说，如果 *rsp* 当前指向 `0x44`，那么执行完 `0x44` AND `0xF0` 之后，它的值就变成了 `0x40`，这个地址是 16 的倍数。

如果是用 GCC 或 Clang/LLVM 来构建程序，那么还有第三个办法，就是给编译器传递 `-mstackrealign` 参数，这会令其在函数的开场及收场部分生成相应的代码，以实现栈对齐，如图 6-7 所示。




图 6-7　在 Xcode 开发环境中调整项目的编译设定，为其添加 `-mstackrealign` 标志

6.4节会讲到，函数的起始地址的十六进制形式一般以0结尾，这意味着它是按16字节对齐的。编译器可以用各种办法实现对齐。比方说，如果把C++版的Sum程序放在macOS上反汇编，那么就有可能看到范例6-4中的情况，这表示编译器（Clang/LLVM）是采用NOP指令（也就是第23行里的nopl）来确保对齐的。开发者当然也可以用这种办法来实现栈对齐，不过，笔者还是建议尽量把它交给编译器做。

**学习指南：** 用你所习惯的开发环境来编写范例6-1中的C++代码，并将其编译为64位的程序。然后对目标文件做反汇编，看看这款编译器是采用哪种办法来实现16字节对齐的。反汇编的步骤可参考本书附录C。

范例6-4　用NOP实现栈对齐

```
 1 testing.o:
 2 (__TEXT,__text) section
 3 _main:
 4 0000000000000000        55                    pushq   %rbp
 5 0000000000000001        4889e5                movq    %rsp, %rbp
 6 0000000000000004        4883ec20              subq    $0x20, %rsp
 7 0000000000000008        c745fc00000000        movl    $0x0, -0x4(%rbp)
 8 000000000000000f        c745f802000000        movl    $0x2, -0x8(%rbp)
 9 0000000000000016        c745f404000000        movl    $0x4, -0xc(%rbp)
10 000000000000001d        8b7df8                movl    -0x8(%rbp), %edi
11 0000000000000020        8b75f4                movl    -0xc(%rbp), %esi
12 0000000000000023        e800000000            callq   0x28
13 0000000000000028        488b3d00000000        movq    (%rip), %rdi
14 000000000000002f        8945f0                movl    %eax, -0x10(%rbp)
15 0000000000000032        8b75f0                movl    -0x10(%rbp), %esi
16 0000000000000035        e800000000            callq   0x3a
17 000000000000003a        be00000000            movl    $0x0, %esi
18 000000000000003f        488945e8              movq    %rax, -0x18(%rbp)
19 0000000000000043        89f0                  movl    %esi, %eax
20 0000000000000045        4883c420              addq    $0x20, %rsp
21 0000000000000049        5d                    popq    %rbp
22 000000000000004a        c3                    retq
23 000000000000004b        0f1f440000            nopl    (%rax,%rax)
24 __Z3sumii:
25 0000000000000050        55                    pushq   %rbp
26 0000000000000051        4889e5                movq    %rsp, %rbp
27 0000000000000054        897dfc                movl    %edi, -0x4(%rbp)
28 0000000000000057        8975f8                movl    %esi, -0x8(%rbp)
29 000000000000005a        8b75fc                movl    -0x4(%rbp), %esi
30 000000000000005d        0375f8                addl    -0x8(%rbp), %esi
31 0000000000000060        89f0                  movl    %esi, %eax
32 0000000000000062        5d                    popq    %rbp
33 0000000000000063        c3                    retq
```

**编程知识：** NOP指令有很多种变化形式，每种形式占据的字节数各有不同。具体到范例6-4来说，它使用的这种形式占据5个字节。请注意，_main与__Z3sumii的起始地址都是以0结尾的，一个是0x00，另一个是0x50。_main函数结束于0x4b这个地址，它后面还有一条占据5个字节的NOP指令，这使得__Z3sumii函数的起始地址变成0x50（0x4b + 0x05 = 0x50），从而实现了向16字节处对齐。《Intel

指令集参考手册》的第 2B 卷[⊖]，详细描述了 NOP 指令的各种形式，此外也可以查阅另一个网上资源：http://www.felixcloutier.com/x86/NOP.html。

x86_64 调用约定还有一个重要特征，就是在栈中预留额外的空间。Microsoft x64 约定所预留的空间，叫作 shadow space 或 home space，其长度是 32 个字节，用来放置参数寄存器中的数值。即便不需要传递任何参数，也依然得预留这么大的空间。如果受调函数经由 *r8*、*r9*、*rcx* 及 *rdx* 寄存器收到的参数还要在该函数调用其他函数的时候用到，那么这些参数的值就可以在 shadow space 中保存一份，除此之外，这块暂存空间（scratch space）也可以用来做其他的事情。

范例 6-5 以“FunctionA 调用 FunctionB”这个抽象的情境为例，描述了采用 Microsoft x64 约定来调用函数时栈中会出现哪些内容，其中也涉及 shadow space。

**范例 6-5　采用 Microsoft x64 约定来调用函数时栈中的各项内容（其中包括 shadow space）**

| 栈的内容 | 描述 |
| --- | --- |
| FunctionA 函数的参数 | 这些参数占据的字节数依具体情况而定 |
| shadow space（32 个字节） | 由 FunctionA 的开场代码所分配，以后调用其他函数时可以使用这一部分空间来暂存数据 |
| 返回地址（8 个字节） | 由 FunctionA 中的 CALL 指令所推入 |
| 早前的 *rbp* 值（8 个字节） | 由 FunctionB 的开场代码所推入 |
| FunctionB 函数的参数 | 这些参数占据的字节数依具体情况而定 |
| shadow space（32 个字节） | 由 FunctionB 的开场代码所分配，以后调用其他函数时可以使用这一部分空间来暂存数据 |

由高地址向低地址排布，并在必要时进行栈对齐



范例 6-6 节选了一段 MASM 版本的汇编代码，以突出其开场与收场部分。这段代码对应于早前 Sum 程序的 main() 函数，其中的指令是通过对 Visual Studio 开发环境所构建的 C++ 程序做反汇编而得来的。

**范例 6-6　对 x86_64 平台的 Sum 程序做反汇编所得到的 MASM 汇编代码**

| 反汇编 | 注释 |
| --- | --- |
| push rbp | ; main() 函数的开场部分 |
| push rdi | ; 保存帧指针 |
| sub rsp,148h | ; 把应该由受调函数来保存的寄存器推入栈中<br>; 分配足够大的栈空间，以容纳 main 函数的局部参数，并留出一些位置，以便在调用其他函数时能够用来保存参数（按照空间占用量最大的情况设想），此外，还要考虑 shadow space 及对齐问题 |
| lea rbp,[rsp+20h] | ; 令帧指针指向 *rsp* 上方 32 字节处，以表示 main 函数的帧边界下有 32 个字节的空间，能够用来保存参数寄存器里的参数值，使得 main 函数可以用这些参数调用其他函数 |
| . | |
| . | |
| . | |
| . | |
| . | ; main() 函数的尾声部分 |
| lea rsp,[rbp+128h] | ; 把早前分配的栈空间丢弃掉。请注意，由于 *rbp* 与这块空间的下界之间有 20h 个（也就是 32 个）字节，因此，只需要把 *rsp* 退回到 *rbp* 上方的 128h（148h - 20h = 128 h）处即可 |
| pop rdi | ; 把应该由受调函数来保存的寄存器从栈中弹出 |
| pop rbp | ; 把早前保存的帧指针从栈中弹出 |
| ret | ; 退出函数，并把返回地址弹出到 *rip* |

⊖ 网址是：https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf。——译者注

AMD64 调用约定没有 32 个字节的 shadow space 这一概念，但它会把存放返回地址所用的位置（*rsp* 就指向此处 ）下方的 128 个字节视为 red-zone。这是打算留给函数使用的一块临时存储空间，不会被系统侵扰。如果没有 PUSH、POP 或 CALL 指令涉及这块区域，那么 *rsp* 就不会改变，于是，开发者可以通过 *rsp* 来引用局部变量及标识符（例如在实现跳转和循环的时候，就可以这样引用相关的标识符）。预留这块区域是为了令 *rsp* 寄存器不会因局部变量而频繁地调整，进而节省时钟周期，以求优化程序的执行效率。此外，它还可以用来实现**末端函数**（leaf function，叶函数），也就是那种不会再继续调用其他函数的函数。主调方可以利用 red-zone 来完成末端函数所要执行的计算，这样就不用再通过 CALL 指令真的去调用它了，从而可以免去一些开销。

与 Microsoft x64 的 shadow space 不同，AMD 64 的 red-zone 并不需要由开发者或编译器专门去预留，它仅仅用来表达一项承诺，意思是说：系统信号及中断处理程序是不会触碰 *rsp* 下方那 128 个字节的。但是，如果调用了其他函数，那么主调函数的 red-zone 就会遭到破坏，因此，这块区域主要还是留给那些不会再调用其他函数的末端函数来用的，那些函数可以把它当作暂存空间。

### 6.3.4　有用的细节

- 如果要使用浮点运算器（Floating-Point Unit，FPU）栈，那么必须在进入及退出函数时清理干净，只有 *st(0)* 例外，因为它可以用来存放返回值。FPU 将在**第 8 章**讨论。
- 在 32 位环境下，如果按引用来传递参数，那么传递到栈中的应该是参数的地址。为了正确传递该地址，程序在运行过程中应该先通过 LEA 指令把地址放在某个中介寄存器里，然后再通过相关指令将其移入或者推入栈中。

```
lea -0x8(%ebp), %eax  # a parameter address is loaded into eax
mov %eax, (%esp)              # address copied to where esp is pointing
```

在 64 位环境下，程序则可以直接把按引用传递过来的参数地址放到寄存器中。

```
lea -0x8(%rbp), %rdi  # a parameter address is loaded into rdi
```

- 在 64 位环境下，如果调用像 `printf()` 这样参数个数可变的函数，那么 *al* 寄存器（也就是 *rax* 的低 8 位）就用来表示调用该函数时用到了几个 SSE 寄存器。
- 与调用约定有关的更多细节可以参考本章的网络资源。尤其是 Agner Fog 所写的资料（http://www.agner.org/optimize/calling_conventions.pdf），以及各种 ABI（Application binary interface，应用二进制接口）文档（例如：http://refspecs.linuxfoundation.org/elf/x86-64-abi-0.99.pdf[⊖] ）。这些资料相当有用，它们对比了各种调用约定之间的异同，并详细地描述了如何遵循这些约定来调用函数。

> **提示：** ABI 文档用来正式地描述软件在调用底层接口时所需满足的要求，它是由 Intel、AMD 及 Microsoft 这样的公司及制造商所发布的。

## 6.4　实现

前面有很多内容都在讨论各种汇编器怎样遵照**同一种**调用约定（也就是 cdecl）来实现本

⊖ 也可以参考：https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI。——译者注

章开头的 Sum 程序（此程序参见范例 6-1，用汇编语言实现出来的对应版本参见程序 6-1）。现在我们则应该看看，在 32 位及 64 位环境下，怎样依照各种**不同的**调用约定把 C++ 等高级语言写成的代码通过汇编指令实现出来。本节将要对比 32 位环境下的 cdecl、stdcall 及 fastcall 约定，笔者所依据的原始代码是在 Xcode 开发工具中以 C++ 语言写成的。

**程序 6-2　用 C++ 语言编写的 Sum 程序**

```
int sum(int, int);

int main() {
   int value1 = 2, value2 = 4, answer;
   answer = sum(value1, value2);
   return 0;
}

int sum(int value1, int value2){
   return value1 + value2;
}
```

**编程知识：**如果你是专门为某个平台而编写汇编代码，或是在高级语言中以内联的形式编写汇编代码，那么可以依照具体情况灵活选用最为合适的办法来实现函数。你可以巧妙地通过一系列指令实现该函数，从而在效率上胜过高级语言的编译器所产生的代码。比方说，32 位程序的参数其实也可以通过寄存器传递，你可以设法缩减指令的数量，使程序不用花那么多条指令在内存之间移动数据并执行运算。有时甚至可以把两个值拼起来用同一个寄存器传递。无论具体采用哪种做法，你都能够更为精细地控制函数的实现方式。

**范例 6-7　对 Sum 程序的 main() 函数做反汇编所得到的指令**

32 位 (cdecl)

```
1   chapter6`main:
2   0x1f50 <+0>:  pushl  %ebp
3   0x1f51 <+1>:  movl   %esp, %ebp
4   0x1f53 <+3>:  subl   $0x18, %esp
5   0x1f56 <+6>:  movl   $0x0, -0x4(%ebp)
6   0x1f5d <+13>: movl   $0x2, -0x8(%ebp)
7   0x1f64 <+20>: movl   $0x4, -0xc(%ebp)
8   0x1f6b <+27>: movl   -0x8(%ebp), %eax
9   0x1f6e <+30>: movl   -0xc(%ebp), %ecx
10  0x1f71 <+33>: movl   %eax, (%esp)
11  0x1f74 <+36>: movl   %ecx, 0x4(%esp)
12  0x1f78 <+40>: calll  0x1f90      ; sum
13  0x1f7d <+45>: xorl   %ecx, %ecx
14  0x1f7f <+47>: movl   %eax, -0x10(%ebp)
15  0x1f82 <+50>: movl   %ecx, %eax
16  0x1f84 <+52>: addl   $0x18, %esp
17  0x1f87 <+55>: popl   %ebp
18  0x1f88 <+56>: retl
```

64 位 (AMD64)

```
1   chapter6`main:
2   0x100000f60 <+0>:  pushq  %rbp
3   0x100000f61 <+1>:  movq   %rsp, %rbp
4   0x100000f64 <+4>:  subq   $0x10, %rsp
5   0x100000f68 <+8>:  movl   $0x0, -0x4(%rbp)
6   0x100000f6f <+15>: movl   $0x2, -0x8(%rbp)
7   0x100000f76 <+22>: movl   $0x4, -0xc(%rbp)
8   0x100000f7d <+29>: movl   -0x8(%rbp), %edi
9   0x100000f80 <+32>: movl   -0xc(%rbp), %esi
10  0x100000f83 <+35>: callq  0x100000fa0  ; sum
11  0x100000f88 <+40>: xorl   %esi, %esi
12  0x100000f8a <+42>: movl   %eax, -0x10(%rbp)
13  0x100000f8d <+45>: movl   %esi, %eax
14  0x100000f8f <+47>: addq   $0x10, %rsp
15  0x100000f93 <+51>: popq   %rbp
16  0x100000f94 <+52>: retq
```

面对用高级语言所写成的同一套语句，各种编译器可能会产生不同的底层代码，然而范例 6-7 里的这两段代码还算是比较接近的，因为它们都遵循相应的调用约定，只是 64 位的版本稍微高效一些。此外，它们的 `main()` 函数也都按 16 字节对齐，前者的起始地址是 `0x1f50`，后者是 `0x100000f60`，这两个地址都是 16 的倍数（参见第 2 行）。

进入 main() 函数时，这两个版本的程序都通过 SUB 指令来给三个局部变量预留空间（参见第 4 行），然后，也都通过 MOV 指令分别为每个变量赋予初始值（参见第 5 至 7 行）。范例 6-8 专门对比了第 4 ～ 7 这四行代码。

**范例 6-8　对 Sum 程序的 main() 函数做反汇编所得的指令（节选其中第 4 至 7 行）**

| 32 位 | | | | 64 位 | | | |
|---|---|---|---|---|---|---|---|
| 4 | 0x1f53 | subl | $0x18, %esp | 4 | 0x100000f64 | subq | $0x10, %rsp |
| 5 | 0x1f56 | movl | $0x0, -0x4(%ebp) | 5 | 0x100000f68 | movl | $0x0, -0x4(%rbp) |
| 6 | 0x1f5d | movl | $0x2, -0x8(%ebp) | 6 | 0x100000f6f | movl | $0x2, -0x8(%rbp) |
| 7 | 0x1f64 | movl | $0x4, -0xc(%ebp) | 7 | 0x100000f76 | movl | $0x4, -0xc(%rbp) |

两者之间的第一个显著区别出现在即将执行 CALL 指令之前，也就是范例 6-9 所演示的地方。32 位版本是把 value1 及 value2 的值分别复制到 *eax* 与 *ecx* 寄存器里，然后再把寄存器里的值复制到栈中（参见第 8 至 11 行）。之所以要复制两次，是因为 MOV 指令不能直接把一个内存地址上的值复制到另一个内存地址中，而是需要通过寄存器等媒介来中转。假如编译器当时是用 PUSH 指令实现的，那么现在反汇编所得到的代码会更简单一些。与 32 位版本不同，64 位版本只需要把值移动到对应的寄存器里就可以了（参见第 8 至 9 行），而不是移动到某个内存地址上。根据 AMD64 约定，函数的头两个参数应该分别用 *rdi* 与 *rsi* 寄存器表示，不过，由于这两个参数都是整型，而这种类型在 C++ 中仅占据 4 个字节（或者说 32 个二进制位），因此，可以直接使用 32 位的 *edi* 与 *esi* 寄存器来表示，而不必使用 64 位的 *rdi* 与 *rsi* 寄存器。

**范例 6-9　对 Sum 程序的 main() 函数做反汇编所得到指令（节选其中用来传递参数的部分）**

| 32 位 | | | | 64 位 | | | |
|---|---|---|---|---|---|---|---|
| 8 | 0x1f6b | movl | -0x8(%ebp), %eax | 8 | 0x100000f7d | movl | -0x8(%rbp), %edi |
| 9 | 0x1f6e | movl | -0xc(%ebp), %ecx | 9 | 0x100000f80 | movl | -0xc(%rbp), %esi |
| 10 | 0x1f71 | movl | %eax, (%esp) | | | | |
| 11 | 0x1f74 | movl | %ecx, 0x4(%esp) | | | | |

接下来，我们对比 sum() 函数本身的实现方式。

**范例 6-10　对 Sum 程序的 sum() 函数做反汇编所得的指令**

| 32 位 (cdecl) | | | | 64 位 (AMD64) | | | |
|---|---|---|---|---|---|---|---|
| 1 | chapter6`sum: | | | 1 | chapter6`sum: | | |
| 2 | 0x1f90 <+0>: | pushl | %ebp | 2 | 0x100000fa0 <+0>: | pushq | %rbp |
| 3 | 0x1f91 <+1>: | movl | %esp, %ebp | 3 | 0x100000fa1 <+1>: | movq | %rsp, %rbp |
| 4 | 0x1f93 <+3>: | subl | $0x8, %esp | 4 | 0x100000fa4 <+4>: | movl | %edi, -0x4(%rbp) |
| 5 | 0x1f96 <+6>: | movl | 0xc(%ebp), %eax | 5 | 0x100000fa7 <+7>: | movl | %esi, -0x8(%rbp) |
| 6 | 0x1f99 <+9>: | movl | 0x8(%ebp), %ecx | 6 | 0x100000faa <+10>: | movl | -0x4(%rbp), %esi |
| 7 | 0x1f9c <+12>: | movl | %ecx, -0x4(%ebp) | 7 | 0x100000fad <+13>: | addl | -0x8(%rbp), %esi |
| 8 | 0x1f9f <+15>: | movl | %eax, -0x8(%ebp) | 8 | 0x100000fb0 <+16>: | movl | %esi, %eax |
| 9 | 0x1fa2 <+18>: | movl | -0x4(%ebp), %eax | 9 | 0x100000fb2 <+18>: | popq | %rbp |
| 10 | 0x1fa5 <+21>: | addl | -0x8(%ebp), %eax | 10 | 0x100000fb3 <+19>: | retq | |
| 11 | 0x1fa8 <+24>: | addl | $0x8, %esp | | | | |
| 12 | 0x1fab <+27>: | popl | %ebp | | | | |
| 13 | 0x1fac <+28>: | retl | | | | | |

首先注意，范例 6-10 中的两个 sum() 函数都是按 16 字节对齐的，因为起始地址 0x1f90 与 0x100000fa0 都是 16 的倍数（参见第 2 行）。然后，观察针对 sum() 函数的栈帧所写的开场（参见第 2 至 3 行）与收场代码。接下来，还要注意，由于 sum() 的两个参数是按值传递的，因此，参数值会从栈中复制一份放到 sum() 函数栈帧内。

32 位版本（第 4 至 8 行）

- 预留空间，以便能够将两个参数的值复制到 sum() 函数的栈帧内（第 4 行）。
- 把 main() 函数通过栈传进来的参数值复制到寄存器里（第 5 至 6 行）。
- 把寄存器的值复制到刚才在 sum() 函数的栈帧内所预留的空间中（第 7 至 8 行）。

64 位版本（第 4 至 5 行）

- 把值从参数寄存器复制到栈中的 red-zone。

这里我们又一次看出，32 位版本的效率稍低一些，或者说，写得有点啰唆，这在某种程度上是按值传递参数引起的。这两套实现代码之间有个很重要的区别，在于 64 位版本的 sum() 函数不需要通过 SUB 指令调整 *rsp* 的位置，因为它的 red-zone 有足够的空间可以暂存计算结果，而且由于它不再继续调用其他函数，因此无须担心这些结果遭到破坏。

接下来，需要把两个值相加，并将结果保存到 *eax* 寄存器（参见 32 位版本的第 9 至 10 行，以及 64 位版本的第 6 至 8 行）。在返回之前，32 位版本还需要对 *esp* 加 8，以便清除早前复制进来的两个参数（参见第 11 行）。最后，两者都通过 POP 指令把栈顶元素弹出到 *ebp* 或 *rbp* 寄存器，并通过 RET 指令将控制权返回给主调函数（参见 32 位版本的第 12 至 13 行，以及 64 位版本的第 9 至 10 行）。

范例 6-11 演示了在 32 位环境下如何依照 stdcall 及 fastcall 这两种约定方式来分别实现对函数的调用。这两种实现方式，可以与依照 cdecl 约定所实现的 32 位版本以及依照 AMD64 约定所实现的 64 位版本相对比。

注意，stdcall 版本的 RET 指令，其写法是 `retl $0x8`（参见 sum 函数第 14 行）。还应该注意的是：如果仔细看一下 Xcode 实现 stdcall 约定时所用的方式，那么你就会发现，它在整体上是反过来实现 stdcall 的（这体现在值的入栈方式及 *esp* 的调整等方面）。fastcall 与 x86_64 版本几乎一样，都用寄存器来传递参数，只不过它用的是 *ecx* 与 *edx* 。

范例 6-11　按照 stdcall 及 fastcall 两种约定方式来实现 Sum 程序

32 位

```
int __stdcall sum(int, int);
```

```
1   chapter6`main:
2   0x1f50 <+0>:  pushl  %ebp
3   0x1f51 <+1>:  movl   %esp, %ebp
4   0x1f53 <+3>:  subl   $0x18, %esp
5   0x1f56 <+6>:  movl   $0x0, -0x4(%ebp)
6   0x1f5d <+13>: movl   $0x2, -0x8(%ebp)
7   0x1f64 <+20>: movl   $0x4, -0xc(%ebp)
8   0x1f6b <+27>: movl   -0x8(%ebp), %eax
9   0x1f6e <+30>: movl   -0xc(%ebp), %ecx
10  0x1f71 <+33>: movl   %esp, %edx
11  0x1f73 <+35>: movl   %ecx, 0x4(%edx)
12  0x1f76 <+38>: movl   %eax, (%edx)
13  0x1f78 <+40>: calll  0x1f90          ; sum
14  0x1f7d <+45>: subl   $0x8, %esp
15  0x1f80 <+48>: xorl   %ecx, %ecx
16  0x1f82 <+50>: movl   %eax, -0x10(%ebp)
17  0x1f85 <+53>: movl   %ecx, %eax
18  0x1f87 <+55>: addl   $0x18, %esp
19  0x1f8a <+58>: popl   %ebp
20  0x1f8b <+59>: retl
```

64 位 (AMD64)

```
int __fastcall sum(int, int);
```

```
1   chapter6`main:
2   0x1f50 <+0>:  pushl  %ebp
3   0x1f51 <+1>:  movl   %esp, %ebp
4   0x1f53 <+3>:  subl   $0x18, %esp
5   0x1f56 <+6>:  movl   $0x0, -0x4(%ebp)
6   0x1f5d <+13>: movl   $0x2, -0x8(%ebp)
7   0x1f64 <+20>: movl   $0x4, -0xc(%ebp)
8   0x1f6b <+27>: movl   -0x8(%ebp), %ecx
9   0x1f6e <+30>: movl   -0xc(%ebp), %edx
10  0x1f71 <+33>: calll  0x1f90          ; sum
11  0x1f76 <+38>: xorl   %ecx, %ecx
12  0x1f78 <+40>: movl   %eax, -0x10(%ebp)
13  0x1f7b <+43>: movl   %ecx, %eax
14  0x1f7d <+45>: addl   $0x18, %esp
15  0x1f80 <+48>: popl   %ebp
16  0x1f81 <+49>: retl
```

| 32 位 | 64 位 (AMD64) |
|---|---|
| 1 chapter6`sum:<br>2 0x1f90 <+0>: pushl %ebp<br>3 0x1f91 <+1>: movl %esp, %ebp<br>4 0x1f93 <+3>: subl $0x8, %esp<br>5 0x1f96 <+6>: movl 0xc(%ebp), %eax<br>6 0x1f99 <+9>: movl 0x8(%ebp), %ecx<br>7 0x1f9c <+12>: movl %ecx, -0x4(%ebp)<br>8 0x1f9f <+15>: movl %eax, -0x8(%ebp)<br>9 0x1fa2 <+18>: movl -0x4(%ebp), %ecx<br>10 0x1fa5 <+21>: addl %eax, %ecx<br>11 0x1fa7 <+23>: movl %ecx, %eax<br>12 0x1fa9 <+25>: addl $0x8, %esp<br>13 0x1fac <+28>: popl %ebp<br>14 0x1fad <+29>: retl $0x8 | 1 chapter6`sum:<br>2 0x1f90 <+0>: pushl %ebp<br>3 0x1f91 <+1>: movl %esp, %ebp<br>4 0x1f93 <+3>: subl $0x8, %esp<br>5 0x1f96 <+6>: movl %ecx, -0x4(%ebp)<br>6 0x1f99 <+9>: movl %edx, -0x8(%ebp)<br>7 0x1f9c <+12>: movl -0x4(%ebp), %ecx<br>8 0x1f9f <+15>: addl -0x8(%ebp), %ecx<br>9 0x1fa2 <+18>: movl %ecx, %eax<br>10 0x1fa4 <+20>: addl $0x8, %esp<br>11 0x1fa7 <+23>: popl %ebp<br>12 0x1fa8 <+24>: retl |

## 6.5　小结

大家现在应该能够更清楚地认识到函数在底层是怎样实现出来的。遵循调用约定，可以促使我们更为规范地使用寄存器以及与函数有关的栈内存。笔者着重讲解了 x86 及 x86_64 这两种平台上最为常见的几种调用约定，然而其中的原理也可以移用到其他平台。掌握与函数有关的底层细节有助于更好地优化用高级语言所写的代码，并且使你能够根据系统中的资源状况合理地改写代码，进而更为高效地调用函数。

## 6.6　关键术语

application binary interface（ABI，应用二进制接口）
callee（受调方；被调方）
caller（主调方）
calling conventions（调用约定）
cdecl
epilogue（函数收场；函数尾声）
fastcall
functions（函数
leaf functions（末端函数；叶函数）
Little-Endian（小端序）
Microsoft x64
procedures（例程）
prologue（函数开场；函数序言）
red-zone
shadow space (home space)
stack frame（栈帧）
stack memory（栈内存）
stdcall
Streaming SIMD Extensions (SSE)（流式 SIMD 扩展）
System V AMD64 (AMD64)

## 6.7　重要的寄存器（32 位和 64 位）

| | |
|---|---|
| *ebp/rbp* | 帧基指针，指向当前执行栈帧的起始处 |
| *esp/rsp* | 栈指针，应该总是指向栈顶，也就是指向最近推入栈中的值 |
| *eax/rax* | 累加器，在大多数情况下，函数的返回值都是放在这里的（有些情况例外，比方说，要返回的是浮点数） |
| *eip/rip* | 指令指针，指向接下来要执行的指令 |

## 6.8 代码回顾

| | |
|---|---|
| PUSH | 将（内存或寄存器中的）值复制到 *esp*/*rsp* 所指的内存位置，并且令 *esp* 减 4 或令 *rsp* 减 8 |
| POP | 将 *esp*/*rsp* 所指向的值复制到指定的目标（也就是内存中的某个位置或某个寄存器里），并且令 *esp* 加 4 或令 *rsp* 加 8 |
| 把标志寄存器中的内容推入栈中或把栈中的内容弹出到标志寄存器 | 对于 16 位的 *flags* 寄存器来说，这两种操作可以分别通过 PUSHF 与 POPF 完成。对于 32 位的 *eflags* 寄存器来说，这两种操作可以分别通过 PUSHFD 与 POPFD 完成（Intel 语法），或是分别通过 PUSHFL 与 POPFL 完成（AT&T 语法）。对于 64 位的 *rflags* 寄存器来说，这两种操作可以分别通过 PUSHFQ 与 POPFQ 完成 |
| 把所有的通用寄存器推入栈中或把栈中的内容弹出到各个通用寄存器 | 对于 16 位寄存器来说，这两种操作可以分别通过 PUSHA 与 POPA 指令完成，对于 32 位寄存器来说，这两种操作可以分别通过 PUSHAL 与 POPAL 指令完成（AT&T 语法），这些寄存器会按照 *eax*、*ecx*、*edx*、*ebx*、*esp*（执行操作之前的值）、*ebp*、*esi*、*edi* 的顺序处理。对于 64 位寄存器来说，没有相应的指令能够执行这两种操作，因此开发者需要逐个地处理这些寄存器 |
| CALL | 把 CALL 指令后面的指令的地址推入栈中，并将控制权交给 CALL 指令的目标（该目标通常用标签来表示） |
| LEA | 载入有效地址，也就是算出内存操作数在程序运行时的地址（或者说偏移量），并将其载入相应的寄存器 |
| NOP | 空操作（或者说无操作），这条指令也会占据内存空间并消耗时钟周期。它有许多种变化形式，有的只占一个字节，有的占据多个字节 |
| RET | 把栈顶的值弹出到 *eip*/*rip*，该值指的就是早前通过 CALL 指令调用本函数时推入栈中的内存地址（或者说内存位置）。在 32 位环境下，这条指令有两种写法，分别对应于 cdecl 及 stdcall 两种调用约定：<br>1. RET<br>2. RET count<br>count 指的是执行完返回（RET）操作之后，需要给 *esp* 加上多少个字节 |

## 6.9 与平台有关的注意事项

- 在 Windows 系统中编写汇编代码时，应该用 PROC 与 ENDP 命令指出例程的起点与终点。
- 在 Windows 系统中编写 MASM 汇编代码时，可以通过 .MODEL 命令指出整个程序所遵循的调用约定（例如 C 或 STDCALL），也可以在每个函数的 PROC 右侧专门指出该函数所遵循的调用约定，以便更为精细地控制程序。

**针对整个程序**

```
.MODEL FLAT, STDCALL
```

```
.MODEL FLAT, C
```

**针对具体例程**

```
sum PROC C
  code
sum ENDP
```

- 在 Windows 系统中编写 MASM 汇编代码时，可以通过 USES 命令指出一些寄存器，使得汇编器为它们生成相应的 PUSH 与 POP 指令。这些指令会自动出现在函数的开场与收场部分，从而能够在程序进入及退出该例程时得到执行（这种写法对于 32 位及 64 位模式都适用）。

```
sum PROC STDCALL USES esi edi
   code
sum ENDP
```

- 在 macOS 系统中，无论是 32 位还是 64 位模式，函数调用都按 16 字节对齐。

## 6.10　习题

### 填空题

1. 函数的局部变量保存在_____中。
2. _____定义了例程（或者说函数）在底层的实现流程。
3. 遵照_____约定来调用函数时，调用方必须负责把栈内存清理干净。
4. _____寄存器可以说是一种内容易失的寄存器，因为它的值有可能在执行受调函数的过程中被使用并覆写。
5. _____指令会把其后的指令所在地址自动推入栈中。
6. _____寄存器指向栈中最新推入的元素。
7. 进入函数与退出函数时需要执行的例行代码称为_____及_____。
8. _____及_____指令在向栈中推入元素或是从栈中弹出元素时，会自动地对栈指针寄存器做减法或加法。
9. 在 32 位环境中按照 stdcall 约定来调用函数时，需要在_____指令中指出栈中有多少个字节有待清理。
10. _____通常当作返回寄存器使用。
11. _____是一种 32 位的函数调用约定，它尽量通过寄存器传值。
12. 依照 x86_64 约定调用函数时，应该通过_____而不是通过栈来传递所有的参数或其中的一部分参数。
13. Microsoft x64 调用约定要求保留_____个字节的 shadow space，AMD64 调用约定要求保留_____个字节的 red-zone 空间。
14. _____函数指的是不再调用其他函数的函数。
15. _____规定了软件在使用底层（或者说机器级别的）接口时所需满足的要求。

### 判断题

1. 栈内存朝着高地址的方向增长（或者说，新分配的栈空间，其内存地址要比旧的空间高）。
2. cdecl 及 stdcall 调用约定都要求参数按照从右到左的顺序传递。
3. 64 位环境下的例程调用必须按 16 字节对齐。
4. FPU 栈必须在进入及退出函数时予以清理。
5. 所有的 x86_64 操作都是 64 位的。

## 6.11　作业

作业 6.1 与 作业 6.2 可参照本书附录 C 完成。

> 📖 **学习指南：把汇编语言同 C++ 集成起来，能够巩固你在本章所学到的概念，而且有助于提高编程技能。可以选用你习惯的调用约定来完成作业。**

作业 6.1 与作业 6.2 要求程序必须由汇编代码中的 `_main` 函数开始执行。例如对于作业 6.1 来说，首先应该执行以汇编语言写成的 `_main` 函数，然后，汇编代码应该调用以 C++ 语言写成的函数，以提示用户输入有待计算的数值，其后，这些值又应该由 C++ 返回给汇编代码。接下来，汇编代码要执行计算，并把结果发送给另一个 C++ 代码以供输出。

### 6.11.1　求和（也可以改用其他某种或某套算术运算）

编写程序，提示用户输入两个整数。然后，将两数相加并显示求和结果。C++ 函数只用来提示用

户输入数值，并在计算完毕后展示结果。控制程序流程、调用函数以及执行数学运算等工作，都要用汇编代码来写。可以提供适当的文字说明，以便在提示用户输入数值及显示运算结果的代码中使用。

### 6.11.2　算术与标志

编写程序，用开发者定义的数据执行一系列简单的计算（例如加法、减法、递增等），每次计算完毕后，都以十六进制或二进制的形式输出 *eflags* 寄存器的内容。C++ 函数只用于输出，其他工作都用汇编代码来写。可以提供适当的文字说明以便在输出寄存器内容的代码中使用。这道题还能够加以延伸，也就是每次运算之后，都根据 *eflags* 标志中的相关二进制位（CF、OF、SF、ZF）来解释这次运算所表现出的一些特点，例如是否进位、是否溢出、是否为负、是否为零等。

### 6.11.3　x86_64 环境下的排序函数（挑战题）

编写 x86_64 汇编程序，对开发者所定义的数组排序。数组大小及地址应该通过寄存器传给受到调用的排序函数。你可以自行选择并实现函数中的排序算法，例如可以使用简单的冒泡排序法（参见 1.9 节）。用包含 5 个或 10 个元素的数组测试该程序。

# 第 6 章补充材料
# 程序6-3

下面演示怎样实现适用于 x86_64 环境的 Sum（求和）程序。通过这个例子可以看出，很多细节问题其实有各种各样的实现方法，比方说，参数不仅可以按值传递，而且还可以像本例这样按引用传递。又比方说，我们可以像 64 位的 GAS 与 NASM 版本一样，利用 **RIP 相对寻址**（RIP-Relative Addressing）技术来获取变量的地址。MASM 版本的代码依照 Microsoft x64 调用规范的要求预留了 32 个字节的 home space。

程序 6-3　适用于 x86_64 的 Sum 程序

**GAS**

```
.data
num1: .quad 2
num2: .quad 4

.text
.globl _main, _sum
_main:

movq $10, %rax
decq %rax
movq $5, %rbx

leaq num1(%rip), %rdi
leaq num2(%rip), %rsi
callq _sum

addq %rbx, %rax
decq %rax

movq $0x2000001, %rax
movq $0, %rdi
syscall

_sum:
pushq %rbp
movq %rsp, %rbp
pushq %rbx
movq (%rdi), %rax
addq (%rsi), %rax
popq %rbx
popq %rbp
retq

.end
```

**MASM**

```
extrn ExitProcess : proc

.data
num1 QWORD 2
num2 QWORD 4

.code
_main PROC
mov rax, 10
dec rax
mov rbx, 5

lea rcx, num1
lea rdx, num2
call _sum

add rax, rbx
dec rax
mov rcx, 0
call ExitProcess
_main ENDP

_sum PROC
push rbp
push rbx
sub rsp, 20h
lea rbp, [rsp + 20h]
mov rax, [rcx]
add rax, [rdx]
lea rsp, [rbp]
pop rbx
pop rbp
ret
_sum ENDP
END
```

**NASM**

```
section .data
num1: dq 2
num2: dq 4

section .text
global _main, _sum
_main:

mov rax, 10
dec rax
mov rbx, 5

lea rdi, [rel num1]
lea rsi, [rel num2]
call _sum

add rax, rbx
dec rax

mov rax, 60
xor rdi, rdi
syscall

_sum:
push rbp
mov rbp, rsp
push rbx
mov rax, [rdi]
add rax, [rsi]
pop rbx
pop rbp
ret
```

# 与字符串有关的指令及结构体

## 目标

- 采用针对基本字符串（string primitive）的指令来操作字符串
- 反复执行基本字符串指令
- 通过结构体创建复合的数据类型

## 网络资源

- http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html（Volume 1, Chapter 5 Instruction Set Summary, Section 5.1）
- http://www.nasm.us/doc/nasmdoc4.html#section-4.11.10（NASM 的结构体）
- https://msdn.microsoft.com/en-us/library/tydf8khh.aspx（MASM 的结构体）
- https://sourceware.org/binutils/docs/as/Struct.html#Struct（GAS 的结构体）

## 7.1 简介

这一章主要讲解字符串与结构体。我们要介绍几种与操作字符串有关的辅助指令以及 5 条用来处理字符串数据的基本字符串指令，待处理的字符串数据以字节数组的形式放置于内存中。早前各章所讲的相关内容也可以用来操作字符串，然而本章要介绍的办法效率更高，而且所需的代码更少。此外，笔者还要介绍如何在汇编代码中使用复合数据类型与结构体。

## 7.2 辅助指令

讲解字符串指令之前必须先介绍两类指令，也就是方向指令与重复指令，因为字符串指令需要和它们搭配起来使用。如果不理解这两类指令，那么就很难看懂字符串指令，乃至无法正确地操作字符串。

表 7-1 列出了两条方向指令，分别用来给**方向标志**（direction flag）清零及置位。第 2 章说过，方向标志决定了内存地址在执行字符串指令的过程中是应该自动递增还是自动递减，这实际上也就决定了字符串的处理顺序是从左至右还是从右至左。

表 7-1 与方向标志有关的指令

| 指　令 | 描　述 | 效　果 |
| --- | --- | --- |
| CLD | 将方向标志清零 | 程序在反复执行字符串操作的过程中会按照从左到右的顺序读取字符（也就是先从地址较低的位置上读取） |
| STD | 将方向标志置位 | 程序在反复执行字符串操作的过程中会按照从右至左的顺序读取字符（也就是先从地址较高的位置上读取） |

CLD与STD指令最适合用在重复操作之前，以便将方向标志调整好。字符串的重复操作是通过重复指令实现的，这些指令的原理与第5章提到的JCXZ/JECXZ/JRCXZ类似。表7-2列出了三种形式的REP（repeat）指令，它们都采用C系列的寄存器做计数器。

表7-2　重复指令

| 指　令 | 描　述 | 效　果 |
| --- | --- | --- |
| REP | 当 *cx/ecx/rcx* ≠ 0 的时候重复 | 这是一种简单的重复逻辑，它根据寄存器的取值来运作，只要相关的C系列寄存器不是0就反复执行某操作 |
| REPE/REPZ | 当 *cx/ecx/rcx* ≠ 0 且 ZF = 1 的时候重复 | 这是一种基于寄存器的重复逻辑，但同时也会考虑ZF的取值。如果ZF清零，或是相关的C寄存器为0，那么就不再重复 |
| REPNE/REPNZ | 当 *cx/ecx/rcx* ≠ 0 且 ZF = 0 的时候重复 | 这是一种基于寄存器的重复逻辑，但同时也会考虑ZF的取值。如果ZF置位，或是相关的C寄存器为0，那么就不再重复 |

把调整方向标志的指令与重复指令结合起来，就能够自动处理字符串数据而不用手工编写循环。为了更好地理解重复指令的运作方式，下面详细列出4个步骤，使大家明白REP指令（或其变化形式）是如何自动执行字符串指令的。

1. 重复指令要判断相关的C系列寄存器是否大于0，如果已经等于0就不再重复。若是比0大则继续重复。

2. 执行字符串指令，根据结果设定相关的标志，并根据具体的指令与方向标志来递增或递减 *si/esi/rsi* 及 *di/edi/rdi*。

3. 如果这条重复指令是REPE/REPNE或REPZ/REPNZ，就根据ZF标志来判断是否应该继续重复。如果不应该就不再重复。

4. 如果第2步（对于另外两种形式来说是第2与第3步）顺利执行，那就将相关的C系列寄存器减1并回到第1步。

详细讲解字符串指令之前先看看下面这个例子。它把内存中的临时数据移动（或者说复制）到另一个地方，以便继续处理。如果我们提前就能够确定这些数据共有10个字符，那么可以直接把该值载入充当计数器的寄存器里，并将源地址与目标地址放入相应的寄存器中。接下来把处理文本时的方向设置好，然后反复执行mov操作。具体细节将在下一节详述。

范例7-1　用NASM汇编代码实现重复处理

```
mov ecx, 10             ; set the counter for 10 characters
xor esi, esi            ; start address of source string (0h)
mov edi, 10000h         ; start address of destination
cld                     ; left-to-right string order
rep movsb               ; repeat BYTE MOVS until ECX=0
```

范例7-2更加直观地演示了范例7-1中的REP指令是如何执行的。由于我们要用的是同时涉及 *esi* 及 *edi* 寄存器的MOV指令，因此应该将其写成MOVS（S表示String，字符串），又因为每次操作的数据都是1个字节，因此要加上后缀B（B表示Byte，字节），也就是MOVSB。REP指令会将MOVSB重复10次，这个次数就是 *ecx* 寄存器的初始值，由于DF标志是0，因此，MOVSB指令每次执行时会给 *esi* 及 *edi* 加1。等 *ecx* 降为0时，REP就结束了。

范例 7-2 MOVSB 操作通过 REP 指令得以反复执行

| 指令 | ECX | ESI | EDI |
|---|---|---|---|
| rep movsb | 10 | 0x00000000 | 0x00010000 |
| | 9 | 0x00000001 | 0x00010001 |
| | 8 | 0x00000002 | 0x00010002 |
| | 7 | 0x00000003 | 0x00010003 |
| | 6 | 0x00000004 | 0x00010004 |
| | 5 | 0x00000005 | 0x00010005 |
| | 4 | 0x00000006 | 0x00010006 |
| | 3 | 0x00000007 | 0x00010007 |
| | 2 | 0x00000008 | 0x00010008 |
| | 1 | 0x00000009 | 0x00010009 |

## 7.3 基本字符串指令

讲完重复指令的基础用法之后，现在就来看这 5 种处理字符串数据的指令。这些**基本字符串指令**（string primitive instruction）其实与早前章节所讲的 MOV、CMP 等指令是类似的，只不过专门用来处理字符数组而已。表 7-3 详细解释了每条指令的作用。

表 7-3 基本字符串指令

| 指　令 | 描　述 |
|---|---|
| MOVS | MOVS 与 MOV 指令类似，然而它专门用来将字符从一个地方复制到另一个地方。MOVS 默认会把由 *si/esi/rsi* 所引用的内存位置当成源数据所在的位置，并将其复制到由 *di/edi/rdi* 所引用的目标位置 |
| CMPS | CMPS 与 CMP 指令类似，然而它专门用来比较由 *si/esi/rsi* 及 *di/edi/rdi* 所引用的字符串。CMPS 最好是和 REPE/REPZ 结合起来使用，这样如果源数据中有某个字符与目标数据中的对应字符不同，那么比较就会失败 |
| SCAS | SCAS 用来扫描由 *di/edi/rdi* 所引用的字符串，以判断其中是否包含存储在 *al/ax/eax/rax* 中的字符，该字符具体由哪个寄存器表示要取决于指令的后缀。SCAS 最好是和 REPNE/REPNZ 结合起来使用，这样如果找到了待查的字符，那么就不再扫描了 |
| STOS | STOS 用来把 *al/ax/eax/rax* 中的值存储到由 *di/edi/rdi* 所引用的位置上。STOS 最好是与 REP 搭配起来使用，以便用默认值来初始化数组或类似的结构 |
| LODS | LODS 用来从 *esi* 所引用的位置上加载数值，并将其存入 *al/ax/eax/rax* 中。它不应该与重复指令合起来使用，因为那样做会导致累加寄存器频繁遭到覆写 |

**编程知识：**这些字符串指令都会根据方向标志（也就是 DF）的取值，给 *si/esi/rsi* 及 *di/edi/rdi* 加上或减去一定的字节数，使得程序能够按照开发者预想的方向来处理数组。即便不与 REP 搭配，这些字符串指令也依然会执行递增或递减运算。

### 7.3.1 MOVS

MOVS（move string，移动字符串）指令与 MOV 类似，但是有两个区别，第一，它的源操作数与目标操作数不需要写在指令中，而是默认由 *esi* 及 *edi* 等寄存器来表示。第二，所有的汇编器都要求在该指令后面写上一个与元素大小相对应的后缀字母。*si/esi/rsi* 寄存器指向待复制的数据所在的内存位置，*di/edi/rdi* 寄存器指向目标地址。在与 REP 相搭配用以连续复制多个字符的过程中，为了使下一次复制操作能够找到正确的字符，开发者必须指

出每个字符所占据的字节数，从而令 MOVS 可以根据 DF 标志适当地递增或递减相应的寄存器。

由于字符串指令每次所操作的数据可能是 1、2、4、8 或 16 个字节，因此，必须通过后缀指明具体的字节数才能正确地定位到其中的每一个字符。各种汇编器均要求添加这样的后缀，当然这对于习惯了 GAS 汇编器的开发者来说可能并不显得特殊，因为 GAS 本来就规定大多数指令都要添加后缀。第 3 章曾经讲过每一种后缀所代表的操作数大小，表 7-4 重新将其列出。

表 7-4　字符串指令所添加的后缀

| 大　小 | 后缀字母 |
|---|---|
| 字节（byte，8 位） | B |
| 字（word，16 位） | W |
| 双字（dword，32 位） | D |
| 四字（qword，64 位） | Q |
| 八字（octa，128 位） | O |

MOVS 指令最简单的用法是单独以 MOVS+ 后缀的形式来使用，例如 MOVSQ，不过正如早前所说，通常我们是重复执行这条指令的，而不是仅仅执行一次。范例 7-3 演示了 REP MOVS 这种组合形式并添加了必要的辅助指令。

范例 7-3　MOVS

| GAS | MASM | NASM |
|---|---|---|
| leal src, %esi<br>leal dst, %edi<br>movl $10, %ecx<br>cld<br>rep movsd | lea esi, src<br>lea edi, dst<br>mov ecx, 10<br>cld<br>rep movsd | lea esi, [src]<br>lea edi, [dst]<br>mov ecx, 10<br>cld<br>rep movsd |

范例 7-3 中的代码假设程序中已经有了名为 src 及 dst 的变量。该范例要通过这两个变量传输数据。首先，用两条指令分别把两个变量的内存地址加载到 *esi* 及 *edi* 寄存器中，接下来把 *ecx* 设置成 10，意思是要复制 10 个字符（或者说重复执行 10 次复制）。然后将 DF 清零以便从左至右地处理数据。最后把 MOVSD 指令与 REP 相搭配以便反复执行 MOVSD 操作直到 *ecx* 变成 0。要注意，由于 REP 每一轮都会递减 *ecx* 的值，因此它终究会降为 0。

图 7-1 分步演示了范例 7-3 中的 rep movsd 语句是如何执行的。

## 7.3.2　CMPS

CMPS（compare string，比较字符串）指令与 CMP 指令相似，但是它与 MOVS 一样也不带操作数，而是默认会通过 *si/esi/rsi* 及 *di/edi/rdi* 来确定有待比较的数据所在的内存地址，而且它也像 MOVS 那样最好是能够与重复指令结合起来使用。如果不搭配重复指令，而是单独使用这条指令，那么只能比较一个字符而不是整个字符串。

把重复指令与 MOVS 相搭配，可以按序**复制**字符，但若与 CMPS 相搭配则是按序**访问**字符，这有点像编程课中用高级语言编写的顺序访问算法（sequential access algorithm）。为了对比两个字符串，CMPS 指令必须从一个字符移动到下一个字符，然而它在移动的同时还需要判断：当前字符与另一个字符串里的对应字符是否相同。由此可见，与 CMPS 搭配的重复指令不应该是普通的 REP，因为这种指令只能简单地重复，而无法把判断结果考虑进来。我们要用的是另外两种形式的重复指令，也就是 REPE/REPZ 或 REPNE/REPNZ。

以 REPE/REPZ 为例，只有当两个字符串的对应字符相等时才会继续比较下一个字符（因为只有在这种情况下 CMPS 指令才会把 ZF 标志设置成 1，从而使重复指令能够继续执行）。CMPS 通过减法比较这两个字符，也就是把由 *si/esi/rsi* 所表示的字符从由 *di/edi/rdi* 所表示的字符值中减去，并根据结果修改相关的处理器标志，尤其是 ZF 标志。

图 7-2 演示了用 repe cmps 来对比 "Johanna" 与 "Johnnie" 两个字符串时的情况。

**编程知识：** REPE 与 REPZ 是同一个意思，因为它们执行相同的判断逻辑。REPNE 与 REPNZ 两者之间也是如此。

| ESI | 来源字符串 | | 目标字符串 | EDI |
|---|---|---|---|---|
| 0x00000000 | S | ECX=10 | S | 0x00010000 |
| 0x00000004 | e | ECX=9 | e | 0x00010004 |
| 0x00000008 | n | ECX=8 | n | 0x00010008 |
| 0x0000000C | t | ECX=7 | t | 0x0001000C |
| 0x00000010 | e | ECX=6 | e | 0x00010010 |
| 0x00000014 | n | ECX=5 | n | 0x00010014 |
| 0x00000018 | c | ECX=4 | c | 0x00010018 |
| 0x0000001C | e | ECX=3 | e | 0x0001001C |
| 0x00000020 | | ECX=2 | | 0x00010020 |
| 0x00000024 | 1 | ECX=1 | 1 | 0x00010024 |

图 7-1　分步骤演示 REP MOVSD 指令的执行过程

| ESI | 来源 | | 目标 | EDI |
|---|---|---|---|---|
| 0x00000000 | J | ECX=7 | J | 0x00010000 |
| 0x00000004 | o | ECX=6 | o | 0x00010004 |
| 0x00000008 | h | ECX=5 | h | 0x00010008 |
| 0x0000000C | a | ECX=4 | n | 0x0001000C |
| | n | | n | |
| | n | | i | |
| | a | | e | |

| 每次比较的结果 | |
|---|---|
| ECX | ZF |
| 6 | 1 |
| 5 | 1 |
| 4 | 1 |
| 4 | 0 |

由于 ZF=0，因此 REPE 会退出，而 ECX 的值则不再递减

图 7-2　分步骤演示 REP CMPSD 指令的执行过程

通过图 7-2 可以看出，每次重复时都是先通过 CMPS 指令来比较两个字符，只有当两个字符相等（也就是 ZF=1）时才会对 *cx*/*ecx*/*rcx* 递减。因此，第 4 轮循环时由于 CMP 指令发现对应的两个字符不相等，会将 ZF 设为 0，从而导致 REPE 指令提前结束，不再将 *ecx* 由 4 递减为 3[⊖]。

⊖ 实际上是先递减 *ecx* 寄存器然后判断 ZF，因此，本例的 *ecx* 会降为 3。参见：https://c9x.me/x86/html/file_module_x86_id_279.html。——译者注

执行完 REPE CMPS 之后需要立刻确认两个字符串是否相等。如果两个字符串一样长，那么只需根据最后一轮的 ZF 标志来判断即可。有人或许误认为可以通过 *cx/ecx/rcx* 的值判断：如果该寄存器的值是 0 说明两个字符串中的每一组对应字符都已经比较过了，因此两个字符串完全相同，如果不是 0 则说明 REPE 并没有把每一组对应的字符都比较过一遍，而是由于某一组字符不相同导致提前退出，这说明两个字符串不完全相同。像这样通过 *cx/ecx/rcx* 来判断在绝大多数情况下确实是正确的，但有一种情况例外，也就是两个字符串仅仅在最后一个字符上有所区别。在这种情况下，由于已经比较过最后一组字符了，因此计数器的值会降为 0，可是这并不意味着这一组字符肯定相同，因而也就无法确定两个字符串完全一样。要想确定这一点，应该看 ZF 是否为 1。

如果两个字符串长度不同，那么用 CMPS 去比较是有些困难的。此时，可以先比较字符串的长度，若是不相等则直接放弃，从而绕过这个难题。假如非要通过 CMPS 做比较，那么一定要把计数器的值设置成较短的字符串所具备的长度。如果以较长的字符串为标准，那么程序所访问的内存就有可能超出较短的字符串所在的范围，从而产生错误的结果。要记住，汇编语言不会自动检查是否越界。

CMPS 指令最基础的用法是单独以 CMPS+ 后缀的格式使用，例如 CMPSD。程序 7-1 演示了 CMPS 指令与 REPE 相结合的用法，这段代码在运用该指令之前会先执行必要的准备工作。

**程序 7-1　CMPS**

GAS

```
.data
src:
.ascii "Test sentence!\n\0"
.equ lenSrc, (. - src)
dst:
.ascii "Test sentence!\n\0"
.equ lenDst, (. - dst)

.text
.globl _main
_main:

# compare string lengths
movl $lenSrc, %eax
cmpl $lenDst, %eax
jne notequal

# when lengths are equal
# execute CMPS
leal src, %esi
leal dst, %edi
movl $lenSrc, %ecx
cld
repe cmpsb

# after CMPS, see if it
# ended successfully
jnz notequal

# equality things
jmp done
```

MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO,
dwExitCode:DWORD

.data
src DB "Test sentence",0
lenSrc EQU ($ - src)
dst DB "Test sentence",0
lenDst EQU ($ - dst)

.code
_main PROC

; compare string lengths
mov eax, lenSrc
cmp eax, lenDst
jne notequal

; when lengths are equal
; execute CMPS
lea esi, src
lea edi, dst
mov ecx, lenSrc
cld
repe cmpsb

; after CMPS, see if it
; ended successfully
jnz notequal
```

NASM

```
SECTION .data
src: DB "Test sentence",0
lenSrc: EQU ($ - src)
dst: DB "Test sentence",0
lenDst: EQU ($ - dst)

SECTION .text
global _main
_main:

; compare string lengths
mov eax, lenSrc
cmp eax, lenDst
jne notequal

; when lengths are equal
; execute CMPS
lea esi, [src]
lea edi, [dst]
mov ecx, lenSrc
cld
repe cmpsb

; after CMPS, see if it
; ended successfully
jnz notequal

; equality things
jmp done

notequal:
; non-equality things
```

| GAS | MASM | NASM |
| --- | --- | --- |
| notequal:<br># non-equality things<br><br>done:<br>pushl $0<br>subl $4, %esp<br>movl $1, %eax<br>int $0x80<br>.end | ; equality things<br>jmp done<br><br>notequal:<br>; non-equality things<br><br>done:<br>INVOKE ExitProcess, 0<br>_main ENDP<br>END | done:<br>mov eax, 1<br>mov ebx, 0<br>int 80h |

程序 7-1 先声明了两个字符串，以及两个用来表示字符串长度的符号。然后通过这两个符号检查两个字符串所占据字节数是否相同，如果不同就直接认定：这两个字符串不一样。如果相同就为执行 CMPS 指令做准备，也就是把两个字符串的内存地址分别载入 *esi* 与 *edi* 寄存器、把字串长度复制到 *ecx* 寄存器，并将 DF 标志清零，以便按从左至右的顺序处理。准备工作都做好之后就可以反复执行 CMPS 指令了。每次执行该指令的时候，都会在两个数组的对应字节之间做比较，这实际上是在比较两个字符串的对应字符。只要发现某一组字符不相同 REPE CMPS 就会以 ZF = 0 的状态退出，此时，它后面的那条 JNZ 指令会根据 ZF 标志来确定比较结果。如果 ZF = 0 则说明两个字符串不相等，于是就跳到 `notequal` 标签，若 ZF = 1 则说明 REPE CMPS 顺利执行完毕（也就是以 ZF=1 的状态退出），这意味着两个字符串从首字符开始，直到最后一个字符都完全相同，因此两者相等，于是，程序不通过 JNZ 指令跳转，而是直接执行其下方的 `jmp done` 指令。

### 7.3.3 SCAS

MOVS 与 CMPS 指令分别与普通的 MOV 及 CMP 指令相对应，只不过前两者专门用来处理字符串，而本节要讲的 **SCAS**（scan string，扫描字符串）指令则没有普通指令与之对应。这条指令实际上是在执行内置的顺序搜索算法，它与高级语言所写的算法相比有一个地方可能不太相同，那就是该算法在搜寻字符串的过程中查找的是单个字符，而不像高级语言算法那样查找的是由多个字符所构成的子字符串。不过，用高级语言实现的很多顺序搜索算法其实也可以用来查找单个字符。

SCAS 的正常流程是对 *di/edi/rdi* 所指的字符串做迭代，以便在其中寻找与 *al/ax/eax/rax* 里的目标字符相同的字符，如果能找到就提前退出。它的执行逻辑与 CMPS 恰好相反，CMPS 在继续执行下一轮之前，应该先确保对应位置上的字符**相同**，因此，它要与 REPE 或 REPZ 指令搭配，而 SCAS 在继续执行下一轮之前则应该先确保当前位置上的字符与待搜寻的字符**不同**，因为只有这样才有必要判断下一个字符。由此可见与它相搭配的应该是 REPNE 或 REPNZ 指令。

图 7-3 分步演示了用 REPNE SCASD 指令在字符串 `"Moore"` 里搜寻字母 `'r'` 的过程。

**编程知识：**与 CMPS 相搭配时，REPE/REPZ 指令不会在 CMPS 失败的时候递减 *cx/ecx/rcx* 寄存器[⊖]。与 SCAS 相搭配时，REPNE/REPNZ 指令总是会递减 *cx/ecx/rcx* 寄存器，而不考虑 SCAS 指令是成功还是失败。

⊖ 请参见 7.3.2 节的译者注。

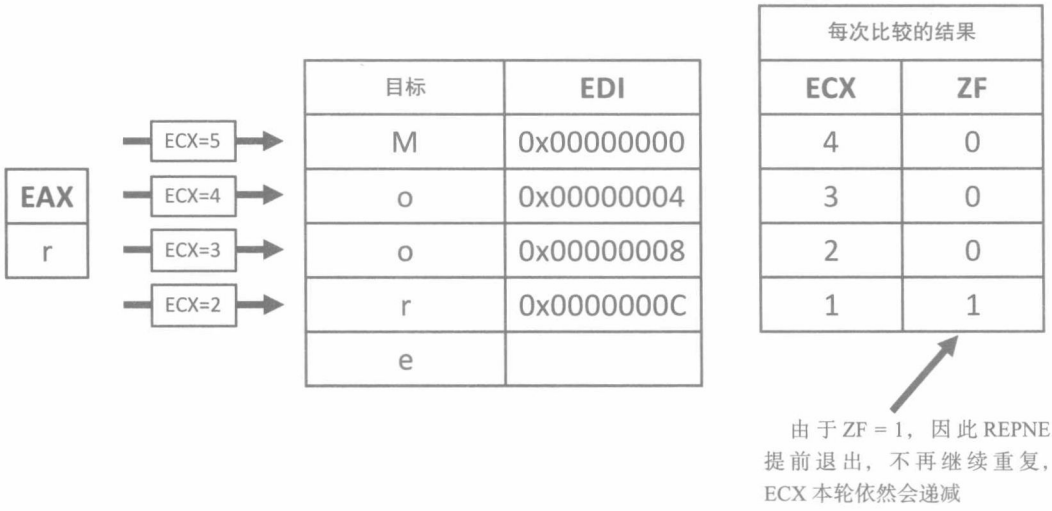



图 7-3　分步骤演示 REPNE SCASD 的执行过程

执行完 SCAS 之后需要判断程序有没有找到待搜寻的字符。原理与 CMPS 类似：如果查完整个字符串都没有发现该字符，那么计数器的值就是 0 而且 ZF = 0，如果发现了字符，那么计数器的值可能等于 0 也可能大于 0(具体是多少要看是在什么位置上找到这个字符的)，但是 ZF 总归是 1。因此，与 CMPS 一样，仅凭 ZF 来判断就够了。

SCAS 指令最基础的用法是单独以 SCAS+ 后缀的形式使用，例如 SCASW。程序 7-2 演示了怎样将其与重复指令结合起来使用。

程序 7-2　SCAS

GAS

```
.data
src:
.ascii "Test sentence\0"
.equ lenSrc, (. - src)
search: .ascii "s"

.text
.globl _main
_main:

xor %eax, %eax
movb search, %al
leal src, %edi
movl $lenSrc, %ecx
cld
repne scasb

# after SCAS, test for
# success
jnz notfound

# found things
jmp done

notfound:
# non-found things
```

MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO,
dwExitCode:DWORD

.data
src BYTE "Test sentence",0
lenSrc EQU ($ - src)
search BYTE "s"

.code
_main PROC

xor eax, eax
mov al, search
lea edi, src
mov ecx, lenSrc
cld
repne scasb

; after SCAS, test for
; success
jnz notfound

; found things
```

NASM

```
SECTION .data
src: DB "Test sentence",0
lenSrc: EQU ($ - src)
search: DB "s"

SECTION .text
global _main
_main:

xor eax, eax
mov al, [search]
lea edi, [src]
mov ecx, lenSrc
cld
repne scasb

; after SCAS, test for
; success
jnz notfound

; found things
jmp done

notfound:
; non-found things
```

| GAS | MASM | NASM |
| --- | --- | --- |
| done:<br>pushl $0<br>subl $4, %esp<br>movl $1, %eax<br>int $0x80<br>.end | jmp done<br><br>notfound:<br>; non-found things<br><br>done:<br>INVOKE ExitProcess, 0<br>_main ENDP<br>END | done:<br>mov eax, 1<br>mov ebx, 0<br>int 80h |

程序 7-2 的功能很简单，就是在字符串 "Test sentence" 中按顺序搜索字符 's'，其中的三个变量分别表示字符串本身、字符串的长度以及待搜索的字符。由于待搜索的字符必须放在累加寄存器里（8 位的是 *al*，16 位的是 *ax*，依此类推），因此需要先用 XOR 把 *eax* 清零。我们以字节为单位来搜索，于是，应该把待搜索的字符加载到 *al* 中。

和早前讲过的字符串指令不同，SCAS 指令只使用 *edi* 而不使用 *sdi*。它把 *edi* 所指的字符串当成待搜索的字符串。程序将字符串的长度载入 *ecx* 寄存器，然后把 DF 清零并反复执行 SCASB 操作。刚才说过，执行完之后需要根据 ZF 标志判断是否找到了待查的字符。如果 ZF 标志等于 0 JNZ 指令就会成功执行，从而跳转到 notfound 标签以表示没有找到该字符。如果 ZF 标志等于 1，那么 JNZ 指令就会失败以表示程序找到了这个字符，本例就属于这种情况，因为待查的字符 's' 确实在字符串中。此时，程序会继续往下走，我们可以给用户发送通知或是执行一些其他任务，不过，本例只是简单地跳转到 done 标签。

### 7.3.4 STOS

STOS（store string，存储字符串）指令很适合用来初始化数组。它会把累加寄存器里的值复制到由 di/edi/rdi 寄存器所表示的内存位置上。如果有多个位置（例如整个数组）都需要用这个值来初始化，那么可以将其与 REP 结合起来使用。

图 7-4 演示了怎样用 STOS 指令把数组中的 5 个元素都初始化为 0。

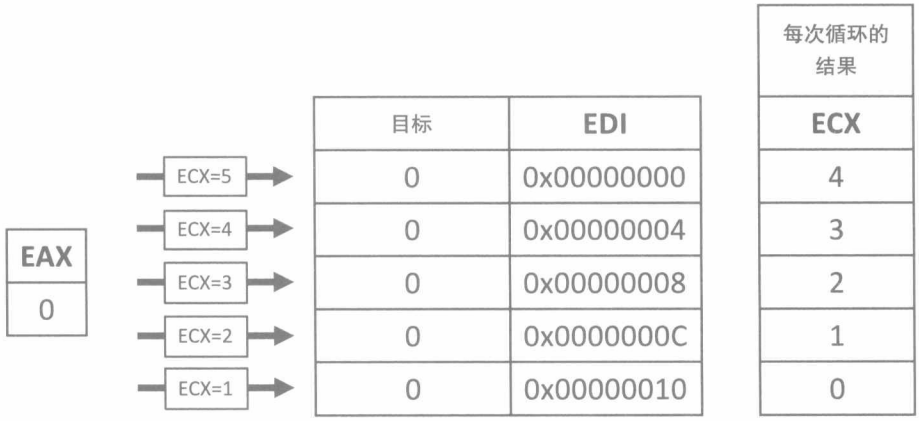



图 7-4 分步骤演示 REP STOSD 的执行过程

**编程知识：** STOS 无法用不同的值来初始化数组中的不同元素，它只能将同一个值复制到数组中的每一个位置上。

STOS 指令最基础的用法是单独以 STOS+ 后缀的形式使用，例如 STOSO。范例 7-4 演示了怎样将 STOS 与 REP 结合起来使用。

范例 7-4　STOS

| GAS | MASM | NASM |
|---|---|---|
| xorl eax, eax<br>movb $0, %al<br>leal dst, %edi<br>movl $10, %ecx<br>cld<br>rep stosb | xor eax, eax<br>mov al, 0<br>lea edi, dst<br>mov ecx, 10<br>cld<br>rep stosb | xor eax, eax<br>mov al, 0<br>lea edi, [dst]<br>mov ecx, 10<br>cld<br>rep stosb |

范例 7-4 的代码可以把变量 dst 所表示的数组中的 10 个元素都初始化为 0。由于本例的指令后缀是 B，因此数据是以字节为单位来处理的，这要求我们把有待复制给 dst 数组中每一个元素的值保存在 *al* 寄存器中，因为该寄存器的长度就是一个字节。

### 7.3.5　LODS

LODS（load string，加载字符串）指令与 STOS 相反，它不是把值从累加寄存器复制到 *di*/*edi*/*rdi* 所表示的地址上，而是从 *si*/*esi*/*rsi* 所指的地址上取值并将其复制到累加寄存器中。与 STOS 不同，这条指令一般不与重复指令相结合，因为那样做会导致累加寄存器频繁遭到覆写。图 7-5 就演示了 *eax* 的值是怎样在每一次执行 LTOS 时为新值所覆盖的。

LODS 指令最基础的形式是 LODS+ 后缀，例如 LODSB。范例 7-5 就演示了这种简单的用法。

范例 7-5　LODS

| GAS | MASM | NASM |
|---|---|---|
| movl $src, %esi<br>lodsd | lea esi, src<br>lodsd | lea esi, [src]<br>lodsd |

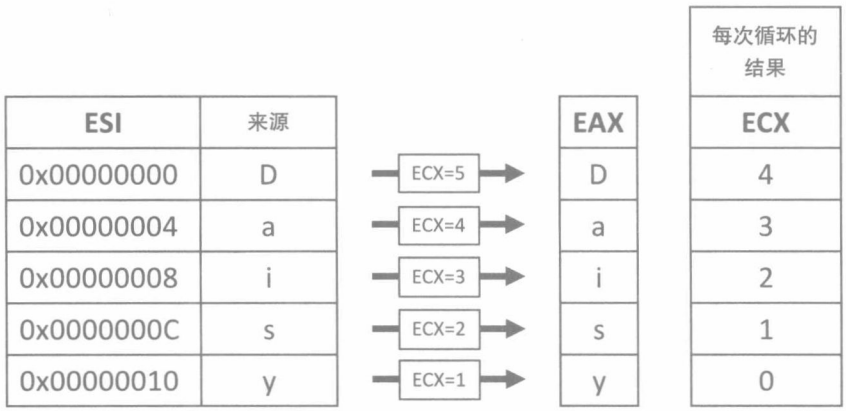



图 7-5　分步骤演示 LODSD 指令与 REP 相搭配时的执行情况

范例 7-5 中的代码把 src 字符串的起始地址加载到 *esi* 寄存器，并通过 LODSD 指令将长度为 32 个二进制位（或者说一个 DWORD，双字）的字符数据复制到 *eax*。由于这条 LODSD 指令并没有与 REP 搭配，因此只会执行一次，而不会在整个 src 字符串上迭代。

## 7.4 结构体

结构体是复合的数据类型，与高级语言类似，汇编语言中的结构体也是由用户所定义的数据类型，能够包含多个字段，这些字段的数据类型可以互不相同。数组中的所有元素都必须一样大，而结构体中的每个元素大小则未必相同，范例 7-6 就演示了这样一个结构体。

**范例 7-6 NASM 汇编代码中的结构体**

```
STRUC person
 .name: RESB 255
 .age:  RESD 1
 .size:
ENDSTRUC
```

范例 7-6 所定义的 name 字段是包含 255 个元素的 BYTE 数组，age 字段是单个的 DWORD 值。size 字段尤其值得注意。STRUC 结构体最后定义这样一个空标签是为了便于 NASM 在未初始化的程序段也就是 .bss 段）中给该结构体预留适当的空间。因为如果要给未初始化的变量预留空间，NASM 必须知道要预留的空间具体有多少个字节。对于原始的数据类型来说，这个信息一般是通过 DB、DD 等命令指定的，然而结构体并不是原始的数据类型。

没有能把整个结构体所占据的字节数告诉汇编器的命令，因此，它需要知道结构体的起始位置（也就是由结构体的标签 person 所标识的位置）及结束位置（也就是由最后 size 标签所标识的位置）。知道了这两个位置，NASM 就可以据此推断出结构体的总尺寸，这就好比知道了当前位置计数器与数组的起始位置我们就可以用 EQU 来确定数组长度。

讲完结构体的基础知识之后，现在来看看怎样在程序中实现它。由于结构体不是内置的数据类型，因此必须先把它定义出来，然后才能声明这种结构体的实例，在高级语言中使用结构体时也是如此。结构体必须定义在 code 与 data 段之外，也就是称为 absolute 段的地方，或者也可以定义在另一份文件中并通过 INCLUDE 命令（该命令的语法参见本章的 7.7 节）将其包含进来。这两种写法可以确保结构体在实例化之前已经具备适当的定义。

通过范例 7-7 中的代码可以看到：三种汇编器在涉及结构体的语法上是有区别的。MASM 与 NASM 都能够采用与高级语言类似的方式来定义并使用结构体，但 GAS 并没有对应的机制，它的 .struct 命令只是用来定义一条绝对表达式，并将其值赋给后面的标签（比方说，在本例中，name = 0，age = 10）。这些标签可以加上 $ 前缀，从而像普通的标识符那样用在 GAS 代码中。由于 GAS 的结构体并不像 MASM 或 NASM 那样运作，因此后面不再讨论 GAS 的 .struct 命令。

**范例 7-7 定义结构体**

| GAS | MASM | NASM |
| --- | --- | --- |
| .struct 0<br>name:<br>.struct name + 10<br>age: | person STRUCT<br>name DB 255 DUP(?)<br>age DD ?<br>person ENDS | STRUC person<br>.name: RESB 255<br>.age: RESD 1<br>.size:<br>ENDSTRUC |

范例 7-7 中的 MASM 与 NASM 汇编代码都实现了同样一个 person 结构体，它的 name 字段是包含 255 个元素的字节数组，age 字段是双字。在汇编代码中声明结构体的实

例并操作该实例时，与高级语言一样也采用圆点来访问字段。MASM 在声明结构体的时候能够判断出其中的字段就是属于该结构体的，而 NASM 则要求必须在字段前面明确写出圆点，如范例 7-7 所示。不加这个圆点，代码就无法通过汇编（这种在 NASM 中声明结构体成员时所用的写法与 GAS 的 .struct 指令不同，注意不要混淆）。

按照范例 7-7 中的写法把结构体定义好之后，就可以像范例 7-8 这样声明该结构体的实例了。下面这两段代码都声明了一个叫作 employee1 的实例。请注意，NASM 版本的代码需要通过 ISTRUC 与 IEND 来标出这个结构体实例。

**范例 7-8　声明结构体的实例**

| MASM | NASM |
| --- | --- |
| .DATA<br>employee1 person <"Waldo",55> | SECTION .data<br>employee1: ISTRUC person<br>AT person.name, DB "Waldo"<br>AT person.age, DD 55<br>IEND |

这两种汇编器都把结构体实例声明在 data 区段里，这与声明其他变量是一样的。MASM 版本的代码写起来更为简洁，然而其原则与 NASM 相同，就是都必须按照各字段的定义顺序来为其指定初始值。范例 7-8 中的 NASM 代码有两个地方值得注意。首先，在 AT 宏后面必须按照 struct_name.member_name（结构体名 . 成员名）的格式来说明各个字段，第二，在 ISTRUC 范围内必须用 data 段中的那种命令，也就是 DB、DD 等，来指定字段的尺寸，而不能像定义 STRUC 时那样用 bss 段中的相应命令（例如 RESB、RESD 等）来指定。

把结构体定义好并声明了它的实例之后，现在看看怎样使用这个实例。范例 7-9 中的两段代码都是把 age 字段的值复制到 *eax* 寄存器。通过这两段代码我们可以看出这两种汇编器怎样通过圆点来访问结构体中的字段。

**范例 7-9　使用结构体中的字段**

| MASM | NASM |
| --- | --- |
| .CODE<br>mov eax, employee1.age | SECTION .text<br>mov eax, [employee1 + person.age] |

MASM 汇编代码的写法与高级语言相同，而 NASM 的写法则与在汇编代码中使用数组时有些类似。要想访问结构体中某个成员字段的值就必须对适当的内存位置做解引用。与数组一样，这个成员字段的值也通过 [label + offset]（[ 标签 + 偏移量 ]）的形式访问，其中的 offset 必须是绝对表达式，用来表示以字节为单位的偏移量。我们必须通过 person.age 这种写法来表示结构体中的 age 字段与该结构体起始处的距离，并把距离加到 employee1 标签上，从而确定出 employee1 实例中的 age 字段所在的位置，而不能把 person.age 写成 employee1.age，因为 NASM 中的 employee1 只不过是个用来表示内存位置的简单标签而已，要想确定 employee1.age 的具体取值，NASM 汇编器不仅要知道 employee1 是个 person 类型的结构体，而且还要知道结构体中的 age 字段有多大，这样才能求出 employee1.age 的值。由此可见，employee1.age 的值对于 NASM 汇编器来说并不能够直接得出，这是一种间接引用式的写法，放在这里会产生汇编错误。NASM 版本的代码必须采用**定义**结构体时的标签（也就是本例中的 person）来表达

某字段相对于结构体起点的偏移量，而不能采用**声明**结构体实例时的标签（也就是本例中的employee1）。

**编程知识：**有的时候必须通过.ALIGN、.ALIGNB 等对齐命令来确保结构体及其成员能够对齐到适当的内存边界处。

最后再举一个例子以演示怎样声明未初始化的结构体。范例 7-10 用这样的结构体创建数组，其中，NASM 版本的代码需要用到结构体的 size 字段，而且还要专门用一个名为 empSize 的变量来表示数组的元素个数（也就是这个 empArr 数组中到底有几个 person 结构体）。

| MASM | NASM |
| --- | --- |
| .DATA<br>empArr person 10 DUP(?)<br><br>.CODE<br>mov eax, LENGTHOF empArr | SECTION .bss<br>empArr: RESB person.size*10<br>empSize: EQU ($ - empArr) / person.size |

NASM 版本的代码需要在声明完数组之后立刻用 EQU 命令及当前位置计数器来声明一个符号，以表示数组的元素个数。32 位的 MASM 不用这样做，因为它能够通过 LENGTHOF 指令动态地判断出数组中有多少个元素。这种写法只适用于 32 位环境，在 64 位环境下，MASM 版本的代码也必须像 NASM 那样通过 EQU 及当前位置计数器来确定数组中的元素个数。

## 7.5 小结

本章讲解了许多条用来高效地操纵字符串的指令。这些基本的字符串指令很适合与相关的辅助指令结合起来运用，以便逐个处理字符串中的各个字符。学过这些内容之后，你现在应该能够比较熟练地操作数组尤其是字符数组了。此外，本章还介绍了怎样在汇编代码中使用结构体，这使得你能够运用面向对象等较为复杂的方式来设计汇编程序。MASM 及 NASM 两种汇编器在结构体的写法上与高级语言相似，而 GAS 的写法则较为原始。

## 7.6 关键术语

CMPS（Compare String，比较字符串）
direction flag（方向标志）
instance（实例）
LODS（Load String，加载字符串）
MOVS（Move String，移动字符串）
REP（Repeat，重复）
SCAS（Scan String，扫描字符串）
STOS（Store String，存储字符串）
string primitive instructions（基本字符串指令）
structure（结构体）

## 7.7 代码回顾

| | |
| --- | --- |
| CLD | 清除方向标志（DF） |
| CMPS | 用 *si/esi/rsi* 及 *di/edi/rdi* 来对比两个字符串 |
| INCLUDE | 预处理命令，用来将起到支持作用的文件纳入程序代码中<br>GAS：.include "file.inc" |

（续）

| | |
|---|---|
| INCLUDE | NASM：`%include "file.inc"`<br>MASM：`INCLUDE file.inc` |
| LODS | 把字符串中的字符加载到寄存器（根据 *si/esi/rsi* 确定字符串） |
| MOVS | 复制字符串（根据 *si/esi/rsi* 及 *di/edi/rdi* 确定源字符串与目标字符串） |
| REP | 反复执行直到 *cx/ecx/rcx* = 0 |
| REPE | 只要 *cx/ecx/rcx* 不是 0 且 *si/esi/rsi* 所指的字符与 *di/edi/rdi* 所指的字符相等就反复执行 |
| REPNE | 只要 *cx/ecx/rcx* 不是 0 且 *si/esi/rsi* 所指的字符与 *di/edi/rdi* 所指的字符不相等就反复执行 |
| REPNZ | 只要 *cx/ecx/rcx* 不是 0 且 ZF 标志没有置位就反复执行。效果与 REPNE 相同 |
| REPZ | 只要 *cx/ecx/rcx* 不是 0 且 ZF 标志置位就反复执行。效果与 REPE 相同 |
| SCAS | 在字符串中扫描（或者说寻找）某个字符（根据 *di/edi/rdi* 确定字符串） |
| STD | 设定方向标志（DF） |
| STOS | 把寄存器的内容保存到字符串（根据 *di/edi/rdi* 确定字符串） |

## 7.8　习题

### 填空题

1. 在执行重复指令之前使用 STD 指令，可以确保程序按照_____的顺序读取字符。
2. 如果要在 ZF 标志为 0 的情况下重复执行，那么应该_____使用指令。
3. 执行字符串指令的时候必须在指令后面添加表示大小的字母，这种写法与_____汇编器对大多数指令的要求类似。
4. 如果要以 8 个字节为单位来操作字符串中的字符，那么在相应的基本字符串指令后面必须写上字母_____。
5. 在使用基本字符串指令的时候必须把_____保存在 *esi* 及 / 或 *edi* 寄存器里。
6. 与 CMPS 指令相搭配的重复指令应该是 REPZ，而不是普通的 REP，这是因为 REPZ 只会在 ZF 等于_____的情况下重复执行。
7. 如果以 CMPSW 的形式来执行 CMPS 指令，那么 *edi* 的值在每轮循环结束之后都要比上一轮大_____。
8. 在反复执行某条字符串指令之前，需要把_____的数量加载到相应的 C 系列寄存器（*cx/ecx/rcx*）里。
9. 如果 SCAS 指令在某一轮循环里找不到要查的字符，那么在执行完这轮循环后 ZF 的值是_____。
10. 重复地执行完 SCAS 之后，使用_____指令可以最为迅速地判断出结果，从而在找不到待查字符的情况下转入相应的代码分支。
11. 执行 STOS 指令的时候，程序默认只根据_____寄存器间接寻址。
12. LODS 会把_____所引用的值复制并保存到_____中。
13. _____指令经常与某一条重复指令搭配起来使用。
14. 编写 GAS 汇编代码的时候，除了可以用 LEA 指令来获取有效地址，还可以给变量加上_____前缀并改用 MOV 指令获取。
15. 如果与 MOVS 相搭配的重复指令是 REPZ 或 REPNZ，而不是普通的 REP，那么程序可能无法实现预想的效果，例如有可能过早地退出。这是因为 MOVS 并不修改_____标志。

### 判断题

1. CLD 指令会清除 ZF 标志（Zero Flag，零位标志）。
2. REPNE 会在 ZF 标志置位的情况下反复执行。
3. 用重复指令来操作字符串的时候，每一轮都是先递减计数器然后再执行字符串指令。
4. 如果要在对应字符相同的情况下继续比较两个字符串，那么与 CMPS 相搭配的重复指令就应该是 REPNZ。
5. 用 STOS 来初始化数组时，该指令会从相应的累加寄存器中取值。

## 7.9 作业

### 7.9.1 模仿判断器

编写程序，比较 "Try harder" 与 "Try harder still" 两个字符串，以判断它们是完全相同、仅在某个范围内相同，还是毫无共同之处。程序必须要能区分这三种情况。

### 7.9.2 大写转小写

编写程序，用适当的基本字符串指令来逐个处理 "I DON'T KNOW WHAT WE'RE YELLING ABOUT!" 字串里的字符，并将其转换成小写。转换后的字符串应该保存在另一个地方。

### 7.9.3 “反汇编”strstr() 函数（用汇编代码来实现这个函数）

C 语言中有个函数叫作 strstr()，用来在大字符串里搜寻某个小字符串，如果能找到这样的子串，就把大字符串里首次出现该子串的地方所对应的内存地址返回给调用方。现在要求用汇编语言撰写功能相仿的程序，该程序不返回内存地址而是返回大字符串中首次出现子串的这个位置与起始位置之间有多少个元素（或者说，子串首次出现在大字符串里的第几个字符上）。

第 8 章

Assembly Programming and Computer Architecture for Software Engineers

# 浮点运算

## 目标

- 了解浮点数的各种表示方式
- 把数值转换成 IEEE 浮点数
- 知道浮点数是怎样保存的
- 了解浮点数的实现方式 / 浮点数指令集
- 使用浮点数指令来操作浮点值

## 网络资源

**维基资源**

- https://en.wikipedia.org/wiki/X87
- https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions
- https://en.wikipedia.org/wiki/X86_instruction_listings#SSE_instructions
- https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

**技术资源**

- http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf
- http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf
- https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions

**浮点数的运算**

- https://docs.oracle.com/cd/E19957-01/806-3568/ncg_math.html（IEEE 算术）
- http://babbage.cs.qc.cuny.edu/IEEE-754/（IEEE 754 分析工具）

## 8.1 简介

本章概述 x86 与 x86_64 架构中的浮点数运算。首先讲解怎样表示并保存浮点数，这部分内容以 IEEE 规范为主。然后按时间顺序回顾浮点数的发展过程，并用实例来演示其间出现的各种实现方式。这部分内容涵盖 x87 FPU（x87 Floating-Point Unit）、SSE（Streaming SIMD Extension）及 AVX（Advanced Vector Extension）等几个典型。你必须掌握这一章所讲的内容才能够理解浮点数指令并学会在程序中运用这些指令。

📖 **学习指南：** 这一章里的程序 8-1 ～程序 8-8 都是 32 位程序。本章末尾的 8-7 节会给出一个用来做投资计算的 64 位范例程序。

## 8.2 浮点数的表示方式

计算机中的**浮点值**（floating-point value）是一种用来近似表示**实数**的数据形式，这种值的小数点前后通常都有一些数位。浮点数由**有效数**（significand）、基数与指数三部分组成，其中的有效数部分可以容纳固定个数的有效数位，而基数则用来与指数相配合，以便对有效数放大或缩小。




所谓浮点是指有效数部分中的小数点能够左右移动，只要相应地调整基数（例如十进制数的基数是 10）的指数，就能保持整个值不变。

| $12345 \times 10^0$ | $1234.5 \times 10^1$ | $123.45 \times 10^2$ | $12.345 \times 10^3$ | $1.2345 \times 10^4$ |
|---|---|---|---|---|

本书假设读者已经大致熟悉与浮点数有关的一些说法，并掌握了与浮点算术及分数 / 小数有关的数学知识。

### 8.2.1 IEEE 表示法

1985 年，电气电子工程师协会发布了 IEEE 754 技术标准，用以规范浮点数的计算。这份标准在 2008 年更新为 IEEE 754-2008 标准。尽管 IBM 与 Cray（克雷）等公司也提出过自己的标准，但占据主流地位的依然是 IEEE 754 标准。它定义了针对有限数、无限数及 NaN 等特殊值的算术格式，还对涉及浮点数的许多方面做出了规定，例如舍入规则、运算、需要处理的异常状况，以及用于交换的格式等。

表 8-1 列出了几种常见的 IEEE 浮点数格式，其中最常用的是单精度、双精度与扩展双精度格式。四倍精度虽然也定义在 IEEE 标准中，但是一般的硬件都不支持。个别软件支持这种格式，然而具体的支持手法却各不相同。比方说，四倍精度的浮点数可以用两个双精度值来实现，但是 Visual C++ 却把它实现成一个 `double`，而 PowerPC 平台的 GCC 则将其实现为 `long double`，此外，还有可能实现成一些非标准的类型，例如 GCC 的 `__float128` 及 Intel C/C++ 编译器的 `_quad`。

表 8-1 IEEE 754 的浮点数格式

| 格式 | 符号位数 | 指数的位数 | 有效数位数 | 二进制位总数 | 指数偏移值 | 能够精确地表示多少个二进制位 | 相当于能够精确地表示多少个十进制位 | 对应的 C++ 数据类型） |
|---|---|---|---|---|---|---|---|---|
| 单精度（binary32） | 1 | 8 | 23 | 32 | 127 | 24 | ~7 | float |
| 双精度（binary64） | 1 | 11 | 52 | 64 | 1023 | 53 | ~16 | double |
| 扩展双精度（binary80） | 1 | 15 | 63+1（明确地写出前导位） | 80 | 16 383 | 64 | ~19 | long double |
| 四倍精度（binary128） | 1 | 15 | 112 | 128 | 16 383 | 113 | ~34 | long double, __float128, _Quad |

IEEE 的二进制浮点数格式中有一部分叫作指数，它是用带有偏移量的形式来保存的。比方说单精度的浮点数，其指数用 8 个二进制位来表示，并以 127 为偏移量。这意味着必须

先给实际的指数加上 127 然后再予以保存，这样的话，无论实际的指数是多少，经过偏移处理后的指数都是正值（或者说，都是可以视为无符号数的值）。以单精度浮点数为例，它的实际指数可以位于 −126 ～ 127 之间，这意味着加上偏移量之后的指数可以位于 1 ～ 254 之间，这个范围内的值是能够用 8 位二进制正数来表示的。

浮点值通常以**正规形式**（normalized form[⊖]）来保存，为了将实际值转换为这种形式，小数点有可能要朝着某个方向移动，直至它左侧仅有一个数位为止。如果小数点是朝左移动的，那么就把移动过程中所越过的数位总量当成实际的指数；如果小数点是朝右移动的，则将这个总量的相反数视为实际指数。这样处理之后，有效数部分的首个二进制位就是个非零的二进制位，这意味着该位必定是 1，于是就不用专门写出这一位了。许多浮点数格式都不保存这个隐含的二进制位，以便能够多表示一位有效数字。比方说，单精度格式用 23 个二进制位来表示浮点数中的有效数部分，把隐含的二进制位算上就有 24 个精确的数位了。扩展双精度格式是个例外，它要求必须把有效数最左侧的二进制位明确地保存起来。

为了更好地理解 IEEE 表示法及其格式，下面举一个实际的例子，以演示怎样把 78.375 这个浮点值保存成单精度的二进制格式。范例 8-1 分步列出了将该值转换成 IEEE 754 单精度格式的过程。

**范例 8-1　把浮点值转换成 IEEE 754 单精度格式**

| 步骤 | 值 |
|---|---|
| 待转换的浮点值 | 78.375 |
| 1. 确定符号位：正数用 0 表示，负数用 1 表示 | 0 |
| 2. 将小数点左侧的数字转换成无符号的二进制数 | 78 = 1001110 |
| 3. 将小数点右侧的数字转换成分数连加的形式（有时只能取近似值，而无法完全精确地表示出来），连加式中的每一个分数的分母都是 2 的正整数次幂，而分子则对应于二进制小数中的相应数位（在分子为 1 的情况下，这个分数的值就是 2 的某个负整数次幂，参见表 8-2） | .375 = 3/8<br>3/8 = 0/2 + 1/4 + 1/8 + ... + (0 或 1)/$2^n$ ...<br>.375 的二进制小数形式 = .011 |
| 4. 把前三步得到的转换结果写成：sign left.right 的格式（sign 表示符号，left 表示小数点左侧数字的二进制形式，right 表示小数点右侧数字的二进制形式） | 0 1001110.011 |
| 5. 将有效数（即第 4 步中的 left.right 部分）正规化，也就是移动小数点直至其左方仅有一个二进制位且该位是 1 为止。根据移动的距离来确定实际的指数 | 0 1.001110011 × $2^6$ |
| 6. 给实际指数加上偏移量，将其转换成二进制格式所要求的偏移指数 | 6 + 127 = 133 = 10000101 |
| 7. 将第 5、6 两步的结果写成 sign exponent fraction 的格式（sign 表示符号，exponent 表示偏移后的指数，fraction 表示经过正规化处理的有效数） | 0 10000101 1.001110011 |
| 8. 在有效数部分中，把开头的二进制位去掉（因为这一位肯定是 1），并适当地在末尾补 0，使得总位数与 IEEE 754 格式所规定的最大位数相同 | 0 10000101 00111001100000000000000 |
| 9. 以 4 个二进制位为单位将二进制数转换成十六进制数 | 0100 0010 1001 1100 1100 0000 0000 0000<br>4 2 9 C C 0 0 0 |
| 10.（以小端在前的方式）保存到内存中 | 00 C0 9C 42 |

⊖ 又称为归约（化）形式、规范（化）形式、规格（化）形式或标准（化）形式，下同。——译者注

表 8-2 2 的负整数次幂

| 范例 8-1 中的十进制小数（.375）里的各个数位 | . | 3 | 7 | 5 | 0 |
|---|---|---|---|---|---|
| 二进制小数中的各个数位所具备的权重（都是 2 的负整数次幂） | | $2^{-1}$ | $2^{-2}$ | $2^{-3}$ | $2^{-4}$ |
| 该权重所对应的十进制分数值 | | 1/2 | 1/4 | 1/8 | 1/16 |
| 该权重所对应的二进制小数值 | | 0.1 | 0.01 | 0.001 | 0.000 1 |

## 8.2.2 特殊值

在 IEEE 浮点数表示法中特殊值是值得注意的。这些值包括带符号的 0、无穷，以及 NaN（Not-a-Number，非数）。IEEE 754 表示法中的 0 是区分正负号的，这意味着它有两种形式，一种是正 0，一种是负 0。对于大多数运算来说，这两种 0 都会得出相同的结果，但是个别情况例外，比如 1/−0 与 1/+0 的结果不同，前者是负无穷，后者是正无穷，此外对这两种 0 取 log 也会得到不同的结果。表 8-3 以十六进制形式列出了这些特殊的值。

表 8-3 IEEE 754 的特殊值

| 特殊值 | 十进制形式 | 单精度形式 | 双精度形式 | 扩展双精度形式 |
|---|---|---|---|---|
| +0 | 0.0 | 0000 0000 | 00000000 00000000 | 00000000 00000000<br>00000000 00000000 |
| −0 | −0.0 | 8000 0000 | 80000000 00000000 | 80000000 00000000<br>00000000 00000000 |
| +∞ | Inf | 7F80 000 | 7FF00000 00000000 | 7FFF0000 00000000<br>00000000 00000000 |
| −∞ | −Inf | FF80 0000 | FFF00000 00000000 | FFFF0000 00000000<br>00000000 00000000 |
| NaN | NaN | 7FC0 0000 | 7FF80000 00000000 | 7FFF8000 00000000<br>00000000 00000000 |

NaN 是一种用来表示未定义值或假想值的数据。NaN 在浮点数运算中有很多功能，比方说，可以充当无效操作的结果，这里的无效操作包括除以 0、对负值取平方根等。此外，如果拿 NaN 去做运算，绝大多数情况下运算结果也是 NaN。从技术角度看 NaN 分成两种，一种是符号 NaN，另一种是静默 NaN。前者会抛出无效操作**异常**（exception），该异常能够通过浮点数运算而传播，后者不抛出这样的异常。异常是一种需要由硬件或软件做出特殊处理的例外状况。比方说，编译器可能会给未经初始化的浮点变量都填上符号 NaN，如果程序在初始化这些变量之前就想使用它们，那么会导致异常。不同的编译器或程序可能会用不同的编码来表示 NaN，表 8-3 给出的这个 NaN 值只是用来举例的，不是说 NaN 只能用这个值来表示。

## 8.2.3 次正规数

**次正规数**（subnormal number）也叫作**非正规数**（denormalized number），它们可以表示正规范围以外的值。例如在单精度格式下，最小的正规数是 $2^{-126}$，约等于十进制的 $1.175 \times 10^{-38}$。如果用该值附近的数字做减法或除法，那么结果可能会比这个数更小，于是就发生了**下溢**（underflow）。从前有一种粗暴的做法是强行把这样的结果调整为 0，然而 IEEE 754 标准选择了另一种做法，叫作渐进式的下溢，可以把 0 至最小正规数之间的某些值给表示出来。这些值就称为次正规数。单精度格式中的逐渐下溢机制可以把小至 $2^{-149}$ 的

值表示出来（约等于十进制的 $1.401 \times 10^{-45}$）。总之，这种机制使得那些**极**小的数值也能够受到支持，尽管会损失一些精度，但总比直接调整成 0 要好。

范例 8-2 所用浮点值是 $1.01011111000000000000001111 \times 2^{-128}$，这个值不在正规数的表示范围内。根据逐渐下溢规则，我们通过移动小数点将指数由 −128 化为 −126，从而令其能够以次正规数的形式得以保存[⊖]。把小数点左移两位之后所得到的值，是 $0.0101011111000000000000011 \times 2^{-126}$，尽管它的精度不如原值（注意，原值的小数部分末尾有两个 1，这两位在移动的过程中丢失了），但毕竟与原值较为接近，这要比直接舍为 0 更好。

**范例 8-2　以次正规数的形式，来表示浮点值**

| $1.01011111000000000000001111 \times 2^{-128}$ |
| --- |
| $0.10101111100000000000000111 \times 2^{-127}$ |
| $0.01010111110000000000000011 \times 2^{-126}$ |

每一种精度格式所支持的正规数及次正规数的取值范围都各不相同。表 8-4 列出了各种精度格式所支持的最大正规数与最小正规数，表 8-5 列出了各种精度格式所支持的最大次正规数与最小次正规数。

**表 8-4　各种精度格式所支持的正规数取值范围**

| 精度 | 最大的正规数 | 最小的正规数 |
| --- | --- | --- |
| 单精度 | $3.40282347 \times 10^{+38}$ | $1.17549435 \times 10^{-38}$ |
| 双精度 | $1.7976931348623157 \times 10^{+308}$ | $2.2250738585072014 \times 10^{-308}$ |
| 扩展双精度 | $1.189731495357231765085759326628007 0 \times 10^{+4932}$ | $3.362103143112093506262677817321752 6 \times 10^{-4932}$ |

**表 8-5　各种精度格式所支持的次正规数取值范围**

| 精度 | 最大的次正规数 | 最小的次正规数 |
| --- | --- | --- |
| 单精度 | $1.17549421 \times 10^{-38}$ | $1.40129846 \times 10^{-45}$ |
| 双精度 | $2.2250738585072009 \times 10^{-308}$ | $4.9406564584124654 \times 10^{-324}$ |
| 扩展双精度 | $3.362103143112093506262677817321752 0 \times 10^{-4932}$ | $6.475175119438025110924438958227646 6 \times 10^{-4966}$ |

### 8.2.4　舍入

从技术角度来说，凡是能够以某种格式为计算机所存储的浮点值其实都是数位有限的有理数（也就是说，这种数的小数部分是可以用有限个数位表示出来的）。这意味着，无论采用哪种格式来表示，其所能容纳的数位都有限度，如果要表示的值的数位超过了这个限度，就**必须**做出裁剪。因此，像 π 这样的无理数（或者说无限不循环小数）以及小数部分无限循环的有理数，都只能以近似值来表示。如果一个数无法表示成数值与之完全等同的浮点格式，那就必须把该数舍入成某个能够精确加以表达的浮点值。比方说，如果要表示的这个数，其有效数位比某种格式的有效数部分所能容纳的数位还多，就会出现这种情况。

> **提示：**浮点数（或者说小数）能否用有限个数位来表示，在某种程度上要看采用的是什么进制（也就是要看基数是多少）。例如 1/2 在十进制之下可以写成有限小数 0.5，而 1/3 则无法写成有限小数的形式，因为小数点后面的 3 是无限循环的，也就是 0.333...

⊖ 规则要求：次正规数的实际指数必须是 −126，在保存为二进制形式时，该指数需要加上 126 的偏移量。偏移后的指数是 0。——译者注

假如换用三进制，那么 1/2 就无法表示成有限小数了，而 1/3 反倒可以表示成有限小数 0.1。在二进制之下（也就是基数为 2 的情况下），如果某个小数可以拆解成有限个 2 的负整数次幂连加的形式，那么就可以表示为有限小数[⊖]，反之则不行。

IEEE 754 提供了几种不同的舍入模式，下面会举例说明它们之间的区别。如果要把 2.1234567890987654321 保存成 32 位的单精度格式，必须将其舍入，因为精准地表示这个值所需的二进制位数量比 32 位单精度格式所支持的数量还要多，于是就只能近似地保存成 2.12345671653747559。范例 8-3 演示了几种常用的舍入方式并以 π，3.5 及 −3.5 为例来演示这些值按照这几种方式会分别舍入到哪个整数。

范例 8-3 舍入模式

| 舍入模式 | 3.14159265358979324 | 3.5 | −3.5 |
|---|---|---|---|
| 向最近的值舍入（如果距离两端同样近，就取其中的偶值） | 3 | 4 | −4 |
| 向负无穷方向舍入 | 3 | 3 | −4 |
| 向正无穷方向舍入 | 4 | 4 | −3 |
| 向 0 舍入（也就是截取舍入或截尾舍入） | 3 | 3 | −3 |

用浮点值做运算或者在不同的浮点格式之间转换时，需要考虑舍入方法。大多数系统的默认舍入方法是**向最近的值舍入**（round to nearest）。本书 8.3.1 节将要讲解怎样设置舍入方法。

## 8.3 浮点数的实现

8086 系列的处理器并不是用来做浮点算术的，8086 与 80386 等处理器自身只能执行整数计算。为了支持浮点数，Intel 开发了 8087 及 80386 辅助处理器用以充当 FPU（floating-point unit，浮点运算单元），从而与 CPU 协同运作。这些 FPU 实现了名为 x87 的指令集架构。从 80486/487 开始，FPU 集成到了 CPU 中。FPU 技术在 x87 时代之后也有一些发展，例如 SSE（Streaming SIMD Extensions）及 VAX（Advanced Vector Extensions）。这一节将要按照时间顺序搭配一些代码范例来讲述 x86 架构中较为主要的几种浮点实现技术。

### 8.3.1 x87

x87 指令集本来是一套与 x86 架构相搭配的浮点数架构，然而这两者在设计上还是有很大区别的。x87 定义了自己的一套指令、寄存器及浮点数格式以执行浮点运算。表 8-6 列出了这些 x87 寄存器。这一节里的代码范例只会提到少量的 x87 指令。

**提示：** x87 的实现细节与指令详情，可参考《IA-32 Intel Architecture Software Developer's Manual Volume 1》的第 8 章。

⊖ 例如范例 8-1 中的 0.375，可以拆成 0.25+0.125，也就是 $2^{-2}+2^{-3}$，因此，它在二进制下是可以写成有限小数的。——译者注

表 8-6　x87 寄存器

| FPU 栈中的 80 位寄存器 | 其他 FPU 寄存器 | |
|---|---|---|
| R7 | | |
| R6 | Last Instruction Pointer（指向最后那条指令的指针） | —48 位 |
| R5 | Last Data/Operand Pointer（指向最后数据 / 操作数的指针） | —48 位 |
| R4 | Status Register（状态寄存器） | —16 位 |
| R3 | Control Register（控制寄存器） | —16 位 |
| R2 | Tag Register（标记寄存器 / 标识寄存器） | —16 位 |
| R1 | Opcode Register（操作码寄存器） | —11 位 |
| R0 | | |

浮点运算的通用寄存器是 *R0* 至 *R7* 这八个寄存器。系统每执行完一条 FPU 指令就会把该指令的内存地址放入 Last Instruction Pointer 中，如果该指令还带有操作数，那么操作数的内存地址会放在 Last Data Pointer 中。把刚刚执行过的那条指令及其操作数所处的内存地址记录下来，是为了给异常处理程序提供状态信息。表 8-7 列出了 *R0* 至 *R7* 寄存器以及这两个指针寄存器的长度、用途及各个二进制位的含义。

表 8-7　x87 通用寄存器、Last Instruction 寄存器及 Last Data 寄存器

| 寄存器 | 长度 | 用途 | 各个二进制位的含义 | | |
|---|---|---|---|---|---|
| R0 至 R7 | 80 个二进制位 | 保存浮点值 | 正负号 | 指数 | 有效数 |
| | | | 79 号位 | 78 至 64 号位 | 63 至 0 号位 |
| Last Instruction Pointer | 48 个二进制位 | 存放最后执行的那条非控制指令所在的地址 | 段选择器 | | 偏移量 |
| | | | 47 至 32 号位 | | 31 至 0 号位 |
| Last Data/Operand Pointer | 48 个二进制位 | 存放最后执行的那条非控制指令所涉及的数据或操作数的地址。如果这个寄存器中的值是有效的，那么肯定指的是某个内存操作数的地址 | 段选择器 | | 偏移量 |
| | | | 47 至 32 号位 | | 31 至 0 号位 |

**编程知识：** x87 的控制指令是：FINIT/FNINIT、FCLEX/FNCLEX、FLDCW、FSTCW/FNSTCW、FSTSW/FNSTSW、FSTENV/FNSTENV、FLDENV、FSAVE/FNSAVE、FRSTOR 及 WAIT/FWAIT，其余的都是非控制指令。用来与 Last Instruction Pointer 及 Last Data/Operand Pointer 相交互的指令是：FINIT/FNINIT（会把这两个指针寄存器清零）、FSAVE/FNSAVE（会把这两个指针寄存器清零）、FXSAVE、FSTENV/FNSTENV、FLDENV、FRSTOR 及 FXRSTOR。

讲解范例代码之前还必须理解另外两个 FPU 寄存器，也就是状态寄存器（Status Register）与控制寄存器（Control Register）。状态寄存器用来保存 FPU 的当前状态，如表 8-8 所示。其中 11 至 13 号位很关键，这三位合称 TOP，用来表示**栈顶**（top-of-stack）。

表 8-8　x87 状态寄存器

| 各个二进制位的含义（共 16 位） | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13...11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| FPU 是否忙碌 | 状况代号 3 | TOP（栈顶） | 状况代号 | | | 错误状态 / 中断请求 | 栈错误 | 异常标志 | | | | | |
| | | | 2 | 1 | 0 | | | P | U | O | Z | D | I |

FPU 的 8 个数据寄存器（也就是 *R0* 至 *R7*）形成循环栈，其中任何一个寄存器都有可能充当栈顶。如果向 *R0* 推入一个值，那么下一个值就会推入 *R7* 中。从栈中删除元素时也会出现这样的循环现象，只不过方向与推入元素时相反。为了利用这种特性，开发者并不直接以 *R0* 至 *R7* 这样的名称来指代寄存器，而是以 *ST0* 至 *ST7* 等写法来指代它们，哪个 *R* 寄存器充当栈顶就把哪个 *R* 寄存器叫作 *ST0*。这个 *R* 寄存器在整个 *R* 系列中的序号保存在 FPU 状态寄存器的 TOP 字段中。

状态寄存器中的各个二进制位，其含义如下：

- **FPU busy（15 号位）**：当 FPU 正在执行某条指令时，该位会设置成 1
- **Condition codes（8 至 10 号位，以及 14 号位）**：用来表示浮点算术的结果
- **TOP（11 至 13 号位）**：用来表示目前充当栈顶的 *R* 寄存器是第几个 *R* 寄存器
- **Error Status（7 号位）**：如果正在处理 FPU 异常，那么该位的值是 1
- **Stack Fault（6 号位）**：如果置位，那么意味着 FPU 栈发生了上溢或下溢（试图把数据加载到本身已经具备有效数值的寄存器中，或者试图从已经不含有效数值的寄存器里获取数据）
- **Exceptions（0 至 5 号位）**：P 位表示 precision（精度受损），U 位表示 underflow（下溢），O 位表示 overflow（上溢），Z 位表示 zero divide（除以 0），D 位表示 denormalized operand（非正规的操作数），I 位表示 invalid operation（无效操作）。其中，P、U、O 三种异常，有可能发生在执行完某项操作之后，而 Z、D、I 这三种异常则有可能发生在即将执行某项操作之前

范例 8-4 用一段 x87 代码来演示 FPU 寄存器的用法，这段代码节选自程序 8-1。请注意，这段代码是以 *ST#* 的形式来指代通用寄存器的，而不使用 *R#* 这样的写法。

**范例 8-4　在汇编代码中使用 FPU 寄存器**

| GAS | MASM | NASM |
|---|---|---|
| `finit`<br>`fldpi`<br>`flds value`<br>`fadd %st(1), %st` | `finit`<br>`fldpi`<br>`fld value`<br>`fadd ST(0), ST(1)` | `finit`<br>`fldpi`<br>`fld DWORD [value]`<br>`fadd ST0, ST1` |

表 8-9 描述了 FPU 的**控制寄存器**（Control Register），其中有一些二进制位用来控制无穷、舍入以及精度等几个方面，还有一些二进制位充当异常掩码。每一个掩码位都对应于状态寄存器中的相应异常位。执行完 FPU 初始化（FPU initialization，FINIT）指令之后，所有的掩码位都会默认设置成 1，这意味着所有的浮点异常都交由 FPU 处理。如果开发者把某个掩码位设置为 0，那么发生相应的浮点异常时系统就会产生中断，从而把异常交给程序来处理。6、7、13、14 及 15 号位是保留位，或者说暂时没有用到的二进制位。12 号位叫作 infinity control 位，当今的 x87 处理器已经不使用这一位了。

**表 8-9　x87 的控制寄存器**

| 各个二进制位的含义（共 16 位） | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | | | 无穷控制 | 舍入控制 | | 精度控制 | | | | 异常掩码 | | | | | |
| | | | | | | | | | | PM | UM | OM | ZM | DM | IM |

**每一个异常掩码位所控制的异常**：PM 控制 precision（精度受损），UM 控制 underflow

（下溢），OM 控制 overflow（上溢），ZM 控制 zero divide（除以 0），DM 控制 denormalized operand（非正规的操作数），IM 控制 invalid operation（无效操作）。

x87 FPU 接受 7 种格式的数字：单精度、双精度、扩展双精度、单字整数、双字整数、四字整数，以及压缩的 BCD 整数（packed BCD integer，其中 BCD 的意思是 binary coded decimal，也就是用二进制码来表示的十进制数）。表 8-10 列出了三种汇编器对 x87 数据类型的支持情况。

**学习指南：** BCD 及其用途，可参考本书附录 H。

表 8-10　各种汇编器对 x87 数据类型的支持情况

| x87 FPU 接受的格式 | 相应的 GAS 命令 | 相应的 MASM 命令 | 相应的 NASM 命令 | 该格式所占据的二进制位个数 |
|---|---|---|---|---|
| Single precision | .float or .single | REAL4 | DD | 32 |
| Double precision | .double | REAL8 | DQ | 64 |
| Double extended | .tfloat（支持情况不一） | REAL10 | DT | 80 |
| Word integer | .word | WORD | DW | 16 |
| Doubleword integer | .long or .int | DWORD | DD | 32 |
| Quadword integer | .quad | QWORD | DQ | 64 |
| Packed BCD integer | 不支持 | TBYTE | DT（例如：`DT 42p`） | 80 |

值加载到 FPU 的时候默认会转换成扩展双精度格式，这样可以令精确程度最高。如果要把这样的值存入内存，可以沿用扩展双精度格式，也可以转换成更为短小的格式。控制寄存器里的精度控制（PC）字段能够控制格式。

表 8-11 列出了精度控制字段的值及其含义。

**编程知识：** 扩展双精度与压缩 BCD 这两种格式的支持情况在各平台及汇编器之间可能有所不同。例如，与 Clang 集成到一起的汇编器（也就是 Xcode 中使用的汇编器）就不支持 `.tfloat` 命令，而版本较新的 GNU Assembler（GAS）则支持该命令。无论要使用哪种类型的浮点值，都可以在 .bss 段（也就是存放未初始化的值所用的区段）中把将该值所需的空间预留出来。某些编译器，会用两个 long 与一个 short“拼”成 10 个字节以支持需要用 80 个二进制位来表示的那种数据。GAS 虽然不支持压缩的 BCD 格式，但开发者可以自己实现。

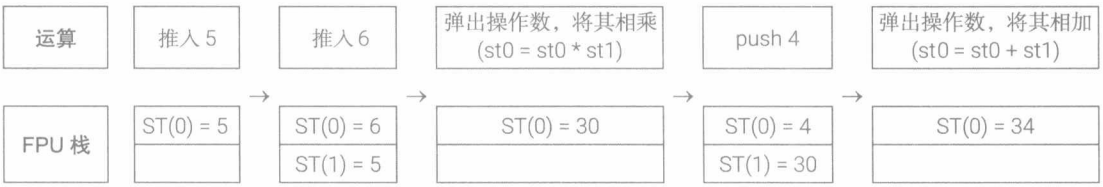
表 8-11　Precision Control（精度控制）字段的值及其含义

| PC 字段的值 | 该值所表示的精度 | PC 字段的值 | 该值所表示的精度 |
|---|---|---|---|
| 00 | 单精度 | 10 | 双精度 |
| 01 | 保留（未使用） | 11 | 扩展双精度 |

x87 FPU 采用**后缀表示法**（postfix notation）来做计算（这和某些计算器是一样的）。比方说，做加法的时候是先把有待相加的两个操作数推入栈中，然后再处理它们之间的加法。以 `(5 * 6) + 4` 为例，写成后缀表示法是 `5 6 * 4 +`，意思是先把 5 和 6 这两个操作数推入栈中，然后处理乘法（也就是把栈中充当操作数的那两个值 6 与 5 弹出，令其相乘，并把乘积 30 压入栈中），接下来把操作数 4 推入栈中，最后处理加法（也就是把栈中充当操作

数的那两个值 4 与 30 弹出，令其相加，并把结果 34 压入栈中）。范例 8-5 演示了怎样计算以后缀表示法来书写的表达式。

范例 8-5 后缀表示法




x87 的 FPU 指令以字母 F 开头。第二个字母如果是 B 或 I，意味着内存操作数应该解读为压缩的 BCD 格式或整数格式，若不是这两种字母则应解读为浮点数格式。比方说，同样是加载数值，如果用 FLD 指令则载入的是实数，如果用 FILD 指令则载入的是整数，若用 FBLD 指令载入的则是压缩的 BCD 整数。程序 8-1 演示了 x87 FPU 的运作方式以及几种基本的浮点数指令。要构建该程序，你应该用 C++ 语言把 `_printFloat` 与 `_printDouble` 函数实现出来，并将这段汇编代码与 C++ 代码相链接（具体做法参见附录 E）。

程序 8-1 x87 FPU

GAS

```
.extern _printFloat
.extern _printDouble

.data
value: .float 1.2

.bss
.lcomm r_value, 4
.lcomm f_result, 4
.lcomm d_result, 8

.text
.globl _asmMain
_asmMain:
 push %ebp
 movl %esp, %ebp

 finit
 fldpi
 flds value
 fadd %st(1), %st
 fists r_value

 fstps f_result
 push $0
 push f_result
 call _printFloat
 add $8, %esp

 fstpl d_result
 push d_result + 4
 push d_result
 call _printDouble
 add $8, %esp

 pop %ebp
 ret
.end
```

MASM

```
.686
.MODEL FLAT, C
.STACK 4096

_printFloat  PROTO C
_printDouble PROTO C

.data
value    REAL4 1.2
r_value  REAL4 ?
f_result REAL4 ?
d_result REAL8 ?

.code
_asmMain PROC
 push ebp
 mov ebp, esp
 finit
 fldpi
 fld value
 fadd ST(0), ST(1)
 fist r_value

 fstp f_result
 push 0
 push f_result
 call _printFloat
 add esp, 8

 fstp d_result
 push DWORD PTR [d_result + 4]
 push DWORD PTR [d_result]
 call _printDouble
 add esp, 8

 pop ebp
 ret
_asmMain ENDP
END
```

NASM

```
extern _printFloat
extern _printDouble

section .data
value: dd 1.2

section .bss
r_value:  resd 1
f_result: resd 1
d_result: resq 1

section .text
global _asmMain
_asmMain:
 push ebp
 mov ebp, esp

 finit
 fldpi
 fld DWORD [value]
 fadd ST0, ST1
 fist DWORD [r_value]

 fstp DWORD [f_result]
 push 0
 push DWORD [f_result]
 call _printFloat
 add esp, 8

 fstp QWORD [d_result]
 push DWORD [d_result + 4]
 push DWORD [d_result]
 call _printDouble
 add esp, 8

 pop ebp
 ret
```

笔者将根据GAS版本来讨论程序8-1，并依次讲解FINIT、FLDPI、FLDS、FADD、FISTS、FSTPS及FSTPL指令。FINIT指令用来初始化FPU，如果想把FPU恢复到默认状态就应该使用这条指令。为了帮助大家理解FPU栈的循环特性，我们假设目前充当栈顶的寄存器是*R3*寄存器（参见范例8-6），这意味着状态寄存器的TOP字段（也就是该寄存器的11至13号位）是011（相当于十进制的3）。

范例8-6　x87 FPU栈

| 数据寄存器 | 值 | 该寄存器在栈中的当前名称 |
|---|---|---|
| R7 | nan | ST4 |
| R6 | nan | ST3 |
| R5 | 0 | ST2 |
| R4 | nan | ST1 |
| R3 | -inf | ST0 (TOP:011) |
| R2 | nan | ST7 |
| R1 | 0 | ST6 |
| R0 | 0 | ST5 |

FPU在执行加载指令的时候会把TOP的值减1，以调整栈顶的位置，并将有待加载的值放在调整之后的栈顶寄存器（即当前的*ST0*寄存器）里。FLDPI就是一种数据加载指令，它会把圆周率π这项数据加载到*ST0*中（参见范例8-7）。

范例8-7　执行完FLDPI指令后的x87 FPU栈

| 数据寄存器 | 值 | 该寄存器在栈中的当前名称 |
|---|---|---|
| R7 | nan | ST5 |
| R6 | nan | ST4 |
| R5 | 0 | ST3 |
| R4 | nan | ST2 |
| R3 | -inf | ST1 |
| R2 | 3.14159265358979324 | ST0 (TOP:010) |
| R1 | 0 | ST7 |
| R0 | 0 | ST6 |

范例8-8　执行完FLDS指令后的x87 FPU栈

| 数据寄存器 | 值 | 该寄存器在栈中的当前名称 |
|---|---|---|
| R7 | nan | ST6 |
| R6 | nan | ST5 |
| R5 | 0 | ST4 |
| R4 | nan | ST3 |
| R3 | -inf | ST2 |
| R2 | 3.14159265358979324 | ST1 |
| R1 | 1.20000004768371582 | ST0 (TOP:010) |
| R0 | 0 | ST7 |

FLDS 指令的基本形式是 FLD，它会把 TOP 减 1 以调整栈顶的位置，然后将浮点值加载到当前的栈顶寄存器里。`flds value` 的意思是，把 `value` 变量所表示的值 1.2 视为单精度浮点数加载到栈顶存器里。通过范例 8-8 可以看出，这条指令的效果，相当于把扩展双精度格式所能表示的数中最接近 1.2 的数（也就是表格中的 `1.20000004768371582`）保存到位于栈顶的 *ST0* 寄存器里。

FADD 是一条算术指令，用来将两个 FPU 寄存器相加。`fadd %st(1), %st` 的意思是把 *ST(1)* 和 *ST(0)* 寄存器中的值相加并将结果存入 *ST(0)*，参见范例 8-9。编写 GAS 汇编代码时，栈顶寄存器既可以写成 *ST* 也可以写成 *ST(0)*。

**编程知识：GAS 与 MASM 都要求栈寄存器的序号必须用一对圆括号括起来，而 NASM 则不要求写出这对括号。**

范例 8-9　执行完 FADD 指令后的 x87 FPU 栈

| 数据寄存器 | 值 | 该寄存器在栈中的当前名称 |
|---|---|---|
| R7 | nan | ST6 |
| R6 | nan | ST5 |
| R5 | 0 | ST4 |
| R4 | nan | ST3 |
| R3 | -inf | ST2 |
| R2 | 3.14159265358979324 | ST1 |
| R1 | 4.34159270127350906 | ST0 (TOP:010) |
| R0 | 0 | ST7 |

`fists r_value` 是一种存储指令，这种指令把当前栈顶寄存器中的值保存到内存中。这些指令都是在 FST 这种基本形式上变化而来的。fists 中的 i 意思是告诉系统将 `r_value` 的值视为整数，而其中的 s 则是 GAS 汇编指令的后缀，意思是说 `r_value` 是一个与单精度浮点数尺寸相同的变量。由于要保存到的地方能够容纳 32 个二进制位，因此添加的这个后缀也应该表示一种 32 位的浮点数类型。

表 8-12 对比了各种汇编器所支持的存储指令。GAS 汇编器要求指令最后必须写上与数据类型相对应的后缀，s 表示 single precision，对应于单精度类型，l 表示 long，对应于双精度类型，t 表示 ten-byte，对应于扩展双精度。很多 FPU 指令都有两个版本，一个是只存储（或者说只复制）不弹出，另一个是存储并弹出（也就是先复制，然后给 TOP 加 1，以调整栈顶的位置）。

表 8-12　各汇编器所支持的 x87 FPU 存储指令

| 指　令 | GAS | MASM | NASM |
|---|---|---|---|
| 存储整数 | fists r_value | fist r_value | fist DWORD [r_value] |
| 存储单精度浮点值 | fsts f_value | fst f_value | fst DWORD [f_value] |
| 存储单精度浮点值并将其从栈中弹出 | fstps f_value | fstp f_value | fstp DWORD [f_value] |
| 存储双精度浮点值并将其从栈中弹出 | fstpl d_value | fstp d_value | fstp QWORD [d_value] |

`fists r_value` 的意思是把栈顶 *ST(0)* 的值向最近的整数舍入并保存到 `r_value` 内存位置上（也就是令 `r_value = 4`）。

FSTPS 是一种存储并弹出指令，它的基本形式是 FSTP。这种指令会把当前栈顶寄存器中的值保存到内存中，然后给 TOP 加 1，以实现将这个值从栈中弹出的效果。`fstps f_result` 的意思是把 *ST(0)* 的值保存到 `f_result` 内存变量中，然后将 TOP 加 1，这相当于将该值从栈中弹出。请注意，系统并没有真的把这个值从栈中删掉，而是仅仅把它所在的寄存器标记为空寄存器，使得其后的指令能够把新值加载到这个寄存器里，到了那时原有的值才会真正遭到覆盖。

**范例 8-10　执行完 FSTPS 指令之后的 x87 FPU 栈**

| 数据寄存器 | 值 | 该寄存器在栈中的当前名称 |
|---|---|---|
| R7 | nan | ST5 |
| R6 | nan | ST4 |
| R5 | 0 | ST3 |
| R4 | nan | ST2 |
| R3 | -inf | ST1 |
| R2 | 3.14159265358979324 | ST0 (TOP:010) |
| R1 | 4.34159270127350906 | ST7 |
| R0 | 0 | ST6 |

`fstpl d_result` 的意思是把栈顶寄存器 *ST(0)* 的值以双精度格式存储到 `d_result` 内存位置上，然后给 TOP 加 1，从而达成将该值从栈中弹出的效果，如范例 8-11 所示。

**范例 8-11　执行完 FSTPL 指令之后的 x87 FPU 栈**

| 数据寄存器 | 值 | 该寄存器在栈中的当前名称 |
|---|---|---|
| R7 | nan | ST4 |
| R6 | nan | ST3 |
| R5 | 0 | ST2 |
| R4 | nan | ST1 |
| R3 | -inf | ST0 (TOP:010) |
| R2 | 3.14159265358979324 | ST7 |
| R1 | 4.34159270127350906 | ST6 |
| R0 | 0 | ST5 |

8.2.4 节及本节都说过，默认的**舍入**（rounding）模式是向最近的偶值取整。这是由控制寄存器的第 10 及 11 号位，也就是舍入控制（RC）字段决定的。该字段的默认值是 00，代表向最近的偶值取整，开发者可以修改该字段从而采用其他的舍入模式。

> **提示：**向最近的偶值取整意味着计算结果会舍入到最接近它的浮点值。然而如果有两个值都同样接近，就取其中的偶值，也就是最低有效位为 0 的值。

程序 8-2 是 GAS 版本的汇编代码，其中有些语句的右侧写有注释。这段程序用来演示如何通过编程的方法修改舍入方式。你可以把它与第 8 章的补充材料中的 MASM 及 NASM 版本相对比。程序 8-2 还使用了几条原来没讲过的 x87 指令，即 FSTCW（store control

word，保存控制字）、FLDCW（load control word，载入控制字）以及 FRNDINT（round to integer，舍入为整数）。

程序 8-2 x87 FPU 的舍入模式

```
.data
default_cw:   .word 0x0000
nearest:      .word 0x0000
down:         .word 0x0400
up:           .word 0x0800
zero:         .word 0x0C00

.bss
.lcomm result, 4

.text
.globl _main
_main:

finit                    # initialize FPU
fstcw default_cw         # store initialized control word in default
movw default_cw, %ax     # move default control word to AX register
andb $0b11110011, %ah    # clear RC field (bits 10 and 11)
orw %ax, nearest         # set RC bits to 00 and store in nearest
orw %ax, down            # set RC bits to 01 and store in down
orw %ax, up              # set RC bits to 10 and store in up
orw %ax, zero            # set RC bits to 11 and store in zero

xor %ax, %ax             # sequence to verify bits
movw default_cw, %ax
movw nearest, %ax
movw down, %ax
movw up, %ax
movw zero, %ax

fldcw nearest            # set desired rounding mode

fldpi                    # load pi on the stack
frndint                  # round to integer (uses rounding mode)
fists result             # store from stack to memory
movl result, %eax        # mov rounded result to EAX

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

📖 **学习指南：**本章接下来的内容都会像程序 8-2 这样只展示 GAS 版本的代码，从而可以留出版面印刷行内注释，以帮助大家理解这些指令。对应的 MASM 及 NASM 版本请参见第 8 章的补充材料。

x87 FPU 中还有两个寄存器也用来保存与 FPU 有关的信息。其中一个是 Tag Register（标记寄存器），它用来表示 FPU 数据寄存器（也就是 *R0* 至 *R7* 这 8 个寄存器）中存放的是何种内容，如表 8-13 所示。有了 Tag Register，开发者就可以先判断相关寄存器中的内容是否合适，然后再去执行更为复杂的指令或解码任务。FSTENV/FNSTENV 或 FSAVE/FNSAVE 指令可以把 FPU 的 Tag Register 存储到内存中。软件不能直接修改该寄存器。

表 8-13　x87 的 Tag Register

| 二进制位（共 16 个） | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 与 R7 寄存器相对应的标记 | | 与 R6 寄存器相对应的标记 | | 与 R5 寄存器相对应的标记 | | 与 R4 寄存器相对应的标记 | | 与 R3 寄存器相对应的标记 | | 与 R2 寄存器相对应的标记 | | 与 R1 寄存器相对应的标记 | | 与 R0 寄存器相对应的标记 | |

与某个 *R* 寄存器相对应的 tag 字段如果是 00，那么表示该 *R* 寄存器中是个有效的非零值。如果是 01 则表示其内容是 0 或与 0 等价的值。如果是 10 则表示其内容是 NaN、inf（无穷）、-inf（负无穷）或非正规数等特殊值。如果是 11 则表示该 *R* 寄存器是空的。FINIT 指令或弹出指令会把 Tag Register 中的相应字段标为 empty。

最后要介绍的 x87 FPU 寄存器是 Opcode Register（操作码寄存器），它用来存放最近执行的那条非控制指令，如表 8-14 所示。该寄存器只关注指令的第一及第二个字节。所有的 FPU 操作指令其首字节的高 5 位均相同，因此 Opcode Register 中只存放该字节的低 3 位。

表 8-14　x87 的 Opcode Register

| 二进制位（共 11 位） | | |
|---|---|---|
| 指令的第一个字节 | | 指令的第二个字节 |
| 7.....3 | 2.....0 | 7.....0 |
| | x87 的 Opcode Register | |

与 Last Instruction 及 Last Data Pointer 寄存器一样，Opcode Register 所保存的也是状态信息，用来供开发者及异常处理程序使用。

### 8.3.2　MMX——题外话

之所以把这一节叫作题外话，是因为 MMX 技术本身并非用于浮点数计算，而是通过 FPU 寄存器来操作整数。Intel 在 1996 年发布 P55C（80503）处理器时引入了这项技术。许多人误以为 MMX 是 MultiMedia eXtension（多媒体扩展）的缩略词，其实它只是 Intel 的商标。MMX 处理器配有 8 个可用于整数运算的 64 位寄存器以及 47 条指令。这 8 个 64 位寄存器分别用 *MM0* 至 *MM7* 来表示，它们其实是 80 位 FPU 数据寄存器 *R0* 至 *R7* 中有效数那一部分的别名。

> 📖 **学习指南：**本书附录 G 给出了一些参考信息及范例代码，你可以据此判断处理器所具备的功能，尤其是判断它是否支持某种与浮点数有关的技术。

MX 技术有两大特性，一个是**压缩的**[⊖]整数型数据（packed integer data），另一个是**单指令流多数据流**（Single Instruction Multiple Data，SIMD）模型。64 位的 MMX 寄存器可以存放压缩形式的字节、字及双字。表 8-15 描述了 MMX 的数据格式。

表 8-15　MMX 的数据格式

| 数据类型 | 64 个二进制位 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 压缩的双字整数（共 2 个） | 63.....32 | | | | 31.....0 | | | |
| 压缩的单字整数（共 4 个） | 63.....48 | | 47.....32 | | 31.....16 | | 15.....0 | |
| 压缩的字节整数（共 8 个） | 63...56 | 55...48 | 47...40 | 39...32 | 31...24 | 23...16 | 15...8 | 7...0 |

向 MMX 寄存器中移入数据或是将 MMX 寄存器中的数据移出，依然要按照普通的方式

⊖ 指的是同一个寄存器或同一块存储区中连续地存放着多份数据。也称为包裹的。下同。——译者注

来做，或者要以四字（也就是 64 个二进制位）为单位来做，不过，保存在寄存器中的这些压缩数据，却可以在同一时间分别参与各自的算术运算。像这种通过一条指令并行地操作多个数据点的理念称为 SIMD 模型。比方说，可以先把由多份数据所构成的压缩数据通过 MOVQ（move quadword，移动四字）指令装入 MMX 寄存器，然后用 PADDSW 这样的算术指令对其中所压缩的那些数据分别进行运算，这样就相当于在一个时钟周期内，用一条指令计算了许多组数据。

下面来看程序 8-3 中的 GAS 代码。

**程序 8-3　MMX**

```
.data
w1: .word 1, 2, 3, 4
w5: .word 5, 6, 7, 8
result: .quad 0

.text
.globl _main
_main:

movq w1, %mm0          # move quadword w1-w4 to MM0
movq w5, %mm1          # move quadword w5-w8 to MM1

paddsw %mm1, %mm0      # add packed signed words
movq %mm0, result      # move data to result

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

`movq w1, %mm0` 这条指令会把由标识符 `w1` 所引领的四字移动到 *mm0* 寄存器中，这相当于把 `w1`、`w2`、`w3`、`w4` 四个单字一起移入 *mm0*。

| | *mm0* | | | |
|---|---|---|---|---|
| | word | word | word | word |
| `movq w1, %mm0` → | 1 | 2 | 3 | 4 |

接下来对 `w5`、`w6`、`w7`、`w8` 执行类似的操作，将其移入 *mm1* 寄存器。

| | *mm1* | | | |
|---|---|---|---|---|
| | word | word | word | word |
| `movq w5, %mm1` → | 5 | 6 | 7 | 8 |

现在，用 PADDSW 指令把 *mm1* 中所压缩的那四个单字值与 *mm0* 中所压缩的那四个对应的单字值分别相加，并把结果存入 *mm0*。由于这条指令只接受两个操作数，因此其中必然要有一个操作数既充当来源操作数又充当目标操作数。

| MMX | word | word | word | word |
|---|---|---|---|---|
| *mm0* | 1 | 2 | 3 | 4 |
| *mm1* | 5 | 6 | 7 | 8 |
| `paddsw %mm1, %mm0` | ↓ | ↓ | ↓ | ↓ |
| *mm0* | 6 | 8 | A | C |

如果需要的话，可以用`movq %mm0, result`这样的指令把*mm0*中的压缩数据移入内存。

**编程知识：**虽然MMX与FPU指令操作的是同一套寄存器，但这两种指令其实可以用在同一个程序中，只不过这样做会产生一些开销。使用MMX指令的时候，FPU会进入一种能够接受MMX数据的模式，并且会把整个x87 Tag Register中的所有Tag字段全都标记为有效，也就是把用来表示相关寄存器内容的tag字段都设为00。在MMX与FPU操作之间切换时应该采取预防措施，否则有可能导致数据受损或产生无意义的结果。切换到FPU操作之前，可以先用FSAVE/FXSAVE指令把MMX状态保存起来，以便在执行完FPU操作之后予以恢复。为了顺利切换到能够执行FPU运算的模式，你应该先用EMMS指令来设置FPU，该指令使得计算机能够接受FPU数据，并且会把Tag Register中的所有tag都设为11，用以表示相应的寄存器处于empty状态。在MMX与FPU之间切换会消耗时钟周期，因此不应该太过频繁。

MMX操作还有很多内容可以讲，但这里只是简单地介绍一下。实现细节与指令的详细用法请参见《IA-32 Intel Architecture Software Developer's Manual》第1卷第9章（这份技术手册的链接，参见本章开头的技术资源）。

### 8.3.3 SSE

新一代的浮点架构随着SSE（Streaming SIMD Extensions，流式SIMD扩展）技术与Pentium III处理器而登场，它是构建在MMX技术之上的。对MMX及SIMD模型所做的这种扩展可以进一步提升图形与网络应用程序的性能。MMX所处理的数据需要以压缩的整数形式保存在64位寄存器中，而SSE所处理的浮点数据则既可以用压缩的形式保存又可以直接用标量的形式保存，而且是保存在128位寄存器中的。请注意，凡是提到压缩形式的数据，就意味着这些数据能够在程序运行过程中并行地得到处理。这些年来，CPU的运算能力逐步提升，SSE技术也随之演变，现在已经出现了SSE、SSE2、SSE3、SSSE3及SSE4等好几代。

SSE指令可以分成四大类：

- 用于操作压缩浮点数及标量浮点数的指令
- SIMD整数指令
- 状态管理指令
- 缓存控制指令、用来对指令的执行顺序/内存的操作顺序做出安排的指令

这一节只简单地介绍几条SSE指令，详细内容可参考《IA-32 Intel Architecture Software Developer's Manual》第1卷第10及11章。

可以在32位模式下使用的128位SSE寄存器是*xmm0*至*xmm7* 8个XMM寄存器。在64位模式下还可以再使用8个寄存器，也就是*xmm8*至*xmm15*。必须注意，在兼容SSE的处理器中，这些寄存器并不是其他寄存器的别名而是单独的寄存器。XMM寄存器只能存放数据不能用来存放地址。

SSE还有一个32位的控制及状态寄存器叫作MXCSR，如表8-16所示，它与x87的控制寄存器及状态寄存器类似。MXCSR包含一些与浮点异常及舍入控制有关的标志与掩码，还有各种用来控制SIMD操作的标志。

表 8-16 SSE MXCSR 寄存器（控制及状态寄存器）

| 二进制位（共 32 个） | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31.....16 | 15 | 14–13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 保留 | FZ | RC | PM | UM | OM | ZM | DM | IM | DAZ | PE | UE | OE | ZE | DE | IE |

- **Exception flags（异常标志，0 至 5 号位）**：IE = invalid operation（无效操作），DE = denormalized operand（非正规的操作数），ZE = zero divide（除以〇），OE = overflow（上溢），UE = underflow（下溢），PE = precision（精度受损）。
- **DAZ（6 号位）**：denormalized operands are zeros。如果该位是 1，非正规的操作数就会在保留符号的前提下转换成零值，这与 IEEE 754 不兼容。
- **Exception masks（异常掩码，7 至 12 号位）**：用来控制相关异常的处理方式，这些异常的含义参见本列表的第一项。
- **Rounding Control（舍入控制，13 与 14 号位）**：同 x87 的舍入控制类似。
- **FZ（15 号位）**：flush to zero。将下溢的值一律调整为 0。这与 IEEE 754 不兼容。

SSE 只支持一种数据类型即单精度的浮点数，如表 8-17 所示。

表 8-17 SSE 所支持的单精度数据格式

| 32 位的单精度值 | 32 位的单精度值 | 32 位的单精度值 | 32 位的单精度值 |
|---|---|---|---|
| 127.....96 | 95.....64 | 63.....32 | 31.....0 |

SSE2 支持更多的数据类型，其中包括双精度浮点数、四字整数、双字整数、单字整数（也称为短整数，short）、字节整数，如表 8-18 所示。

表 8-18 SSE2 的数据格式

| 数据类型 | 128 个二进制位 | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 双四字（1 个） | 128 | | | | | | | | | | | | | | | |
| 压缩的双精度浮点数（两个） | 64 | | | | | | | | 64 | | | | | | | |
| 压缩的四字整数（两个） | 64 | | | | | | | | 64 | | | | | | | |
| 压缩的单精度浮点数（4 个） | 32 | | | | 32 | | | | 32 | | | | 32 | | | |
| 压缩的双字整数（4 个） | 32 | | | | 32 | | | | 32 | | | | 32 | | | |
| 压缩的单字整数（8 个） | 16 | | 16 | | 16 | | 16 | | 16 | | 16 | | 16 | | 16 | |
| 压缩的单字节整数（16 个） | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |

下面考虑一个常见的图形计算问题。假设 vectorA 与 vectorB 代表两个向量，每个向量都包含 4 个元素（或者说 4 个分量），这些元素均为单精度的浮点值。现在要把两个向量的对应元素分别相加并将结果保存在其中一个向量中。

```
float vectorA[] = {1.2, 3.4, 5.6, 7.8};
float vectorB[] = {7.8, 5.6, 3.4, 1.2};
```

用普通的方式执行该任务至少需要连写 4 行 C++ 代码，即便改用循环，也还是相当于按照先后顺序执行了 4 次单精度浮点数加法。

范例 8-12 用 C++ 代码将两个向量中的对应元素相加

```
# as individual statements
vectorA[0] = vectorA[0] + vectorB[0];
vectorA[1] = vectorA[1] + vectorB[1];
```

```
vectorA[2] = vectorA[2] + vectorB[2];
vectorA[3] = vectorA[3] + vectorB[3];

# as a loop
for(int i = 0; i <= 3; i++){
      vectorA[i] = vectorA[i] + vectorB[i];
}
```

如果改用SSE技术来做则只需执行1条指令即可将这4对元素同时相加。程序8-4通过ADDPS（add packed single precison，将两个压缩的单精度值相加）指令对这两个向量做SIMD加法。

**编程知识：**请注意，在汇编代码中声明向量的时候，或许要通过ALIGN命令将向量数据手工对齐到16字节边界处，否则就有可能发生内存访问违例。第6章说过，64位模式下，由于要使用128位的SSE指令，因此函数必须按16字节对齐，现在我们要注意，通过SSE指令来访问内存的时候也必须符合这个要求。

**程序8-4　通过一条SSE指令在压缩形式的数据上并行地执行多次加法操作**

```
.data
.balign 16              # 16-byte alignment
vectorA: .float 1.2, 3.4, 5.6, 7.8
vectorB: .float 7.8, 5.6, 3.4, 1.2

.bss
.lcomm result, 128      # space for storing results

.text
.globl _main
_main:

movaps vectorA, %xmm0   # move aligned packed SP vectorA to XMM0
addps  vectorB, %xmm0   # add vectorB to XMM0
movaps %xmm0, result    # move data to result

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

这段程序会把加法的计算结果保存到*xmm0*寄存器里，使得其中的四个浮点数变为：[9.0，9.0，9.0，9.0]。

SSE也能执行标量运算，由于这种运算是为了取代x87 FPU的对应功能而设计的，因此它的能力与后者类似，但并不完全相同。SSE没有提供π这样的常量，也没有专门用来计算对数、正弦及余弦的操作。SSE的标量指令仅使用XMM寄存器中最低的双字或四字部分来执行浮点运算。比方说，尽管*xmm0*含有4个32位的单精度浮点值，但是MOVSS（move scalar single-precision，移动标量形式的单精度值）指令只会把浮点值加载到*xmm0*中位置最低的双字中，而不会影响其他的3个双字。

从程序8-5中可以看出，MOVSS与ADDSS（add scalar single-precision，将标量形式的单精度值相加）指令只会修改*xmm0*的低32位。与之类似，MOVSD（move scalar double-precision，移动标量形式的双精度值）指令也只会修改*xmm0*的低64位。

**编程知识：**正如程序 8-4 与程序 8-5 所示，大多数 SSE 指令都有两种形式，一种针对标量形式的数据，另一种针对压缩形式的数据。例如 MOVSS 针对的是标量形式，MOVPS 针对的是压缩形式，ADDSS 针对的是标量形式，ADDPS 针对的是压缩形式等。目前的大多数 C++ 编译器都采用标量形式的 SSE 指令计算浮点数，而不会自动采用压缩形式的 SSE 指令来并行地执行多项计算。Intel 的 C++ 编译器是个例外。开发者可以通过 C++ 编译器的 intrinsics（内部函数 / 内置函数，第 9 章及附录 I 会讲到）来利用底层的 SSE 指令以实现并行运算。假如对程序 8-12 中的 C++ 代码反汇编，那么很有可能会发现它是在循环地执行标量形式的运算指令，而没有选用压缩形式的 SIMD 指令。

**程序 8-5　用 SSE 指令对标量执行运算**

```
.data
.balign 16
valueA: .float 1.2
pi: .double 3.14159265358979

.bss
.lcomm result, 4          # space for storing result

.text
.globl _main
_main:                    # scalar examples

movss  valueA, %xmm0      # move valueA to XMM0
addss  valueA, %xmm0      # add valueA to XMM0
movss  %xmm0, result      # store result
movl   result, %eax       # move result to EAX

movsd  pi, %xmm0          # move pi to XMM0

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

对于 MMX、SSE 及 SSE2 技术来说，所有的算术与比较运算都可以在寄存器所压缩的各项数据之间并行地执行。比方说，如果要在 *xmm0* 与 *xmm1* 中分别保存的那 4 个值之间做加法运算，并把结果存入 *xmm0*，那么只需要像程序 8-4 那样使用 ADDPS 指令就可以了，表 8-19 演示了该指令的计算方式。

**表 8-19　通过 SSE 指令在寄存器所压缩的各数据之间并行地执行运算**

| *xmm0* | A3 | A2 | A1 | A0 |
|---|---|---|---|---|
| *xmm1* | B3 | B2 | B1 | B0 |
| 运算 | ↓ | ↓ | ↓ | ↓ |
| *xmm0* | C3 | C2 | C1 | C0 |

SSE 3 除了能像早前的技术那样同时给某一对寄存器内的多份数据执行计算之外，还能在不同的寄存器上同时执行各自的运算，并把运算结果汇总到目标寄存器的相应部分中。前者称为 vertical operation（纵向运算或垂直运算），后者称为 horizontal operation（横向运算

或水平运算）。程序 8-6 演示了执行水平计算所用的 HADDPS 指令，表 8-20 演示了该指令的计算方式。

表 8-20　通过 SSE 指令在不同的寄存器上分别执行同一种运算

| *xmm* | *xmm1* | | | | *xmm0* | | | |
|---|---|---|---|---|---|---|---|---|
| 压缩于寄存器中的数据 | B3 | B2 | B1 | B0 | A3 | A2 | A1 | A0 |
| 运算 | ↓ | | ↓ | | ↓ | | ↓ | |
| *xmm0* | C3 | | C2 | | C1 | | C0 | |

程序 8-6　SSE3

```
.data
.balign 16
vectorA: .float 1.2, 3.4, 5.6, 7.8
vectorB: .float 7.8, 5.6, 3.4, 1.2

.text
.globl _main
_main:

movaps vectorA, %xmm0    # move vectorA to XMM0
movaps vectorB, %xmm1    # move vectorB to XMM1
haddps %xmm1, %xmm0      # horizontal add packed SP vectorB to vectorA

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

程序 8-6 会把 *xmm0* 的前两个值（1.2 及 3.4）、后两个值（5.6 及 7.8），以及 *xmm1* 的前两个值（7.8 及 5.6）与后两个值（3.4 及 1.2）各自相加，并把这 4 次加法的结果分别存放到 *xmm0* 的那四个位置中，令其变成 [4.6，13.4，13.4，4.6]。

SSSE3 只是小幅度地更新了 SSE3 技术，给其中添加了一些新的指令。这些指令大部分是为了优化具体的运算而设计的。SSE4 大幅更新了前代技术，它不再像早前的技术那样特别针对多媒体应用而设计。程序 8-7 通过几条指令演示了 SSE4 技术所支持的几种浮点运算与整数运算。

程序 8-7 中的 ROUNDPS 指令根据立即数 1 或 2 来决定舍入模式，1 表示向下舍入，2 表示向上舍入。尤其值得一提的是 CVTPS2DQ 指令，它可以把 *xmm* 寄存器所压缩的那 4 个单精度浮点数转换成 4 个带符号的双字整数。**转换**一词的英文是 convert，简写为 CVT，**压缩形式的单精度**浮点数，英文是 packed single-precision float，简写为 PS，**到**的英文是 to，与表示数字 2 的英文单词 two 谐音，因此可以简写为 2，**双字**整数的英文是 doubleword integer，简写为 D，由于这样的整数共有 4 个，因此，在后面用 Q（也就是 quad）表示数量，所以这条指令就写为 CVTPS2DQ（CVT—PS—2—DQ）。

SSE 以及由该技术所演化出的各种技术用新的方式实现了浮点数运算，使得多媒体应用程序能够并行地处理其数据。SSE ——尤其是 SSE4 ——为即将出现的下一代浮点运算技术打下了基础。

程序 8-7 SSE4

```
.data
.balign 16
vectorA: .float 1.2, 3.4, 5.6, 7.8
vectorB: .float 7.8, 5.6, 3.4, 1.2

.text
.globl _main
_main:

movaps  vectorA, %xmm0      # move aligned packed vectorA to XMM0
roundps $1, %xmm0, %xmm1    # round down(1) values in XMM0 and store in XMM1
cvtps2dq %xmm1, %xmm2       # convert SPF to int and store in XMM2

movaps  vectorB, %xmm3      # move aligned packed vectorB to XMM3
roundps $2, %xmm3, %xmm4    # round up(2) values in XMM3 and store in XMM4
cvtps2dq %xmm4, %xmm5       # convert SPF to int and store in XMM5

pmulld  %xmm2, %xmm5        # multiply doublewords and store in XMM5

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

程序 8-7 会令 *xmm5* 寄存器的内容变成 [8，18，20，14]。

> **提示：**各代 SSE 技术所支持的指令，可参见 Intel 的开发手册以及本书所提到的其他资源，那些资源中列出了这些指令并详细地给出了解释。此外，还有一个地方也列出了各代 SSE 技术所支持的 SIMD 指令：https://en.wikipedia.org/wiki/X86_instruction_listings#SIMD_instructions。

### 8.3.4 XOP、FMA3/4、F16C——分化

SSE4 登场之后，AMD（Advanced Micro Devices）公司打算推出下一代的 SIMD 扩展技术，并将其称为 SSE5，然而 Intel 公司却决定改用新的实现技术（将在下一节讲述），于是这两家芯片公司就各行其道了，这也就是本节的标题中所说的“分化”。在浮点数指令的发展过程中，这是个即将掀起巨大变化的时刻。

2009 年以后，各种微架构（或者说各系列的处理器，例如 AMD 的 Bulldozer 系列与 Piledriver 系列，以及 Intel 的 Haswell 系列与 Broadwell 系列）都开始集成 XOP（eXtended OPerations）、FMA（Fused Multiply–Add，积和熔加）等指令，并提供 F16C（Half-Precision Conversion，半精度转换）等操作。

XOP 指令集引入了横向整数加法指令、乘积累加指令，以及几条用以执行比较、移动、移位或旋转等操作的向量指令。

FMA 添加了用来执行积和熔加操作的指令，使得 `(a * b + c)` 这样的表达式只需一个步骤就能算出结果。FMA3 与 FMA4 之间的区别在于操作数的个数，前者使用 3 个操作数，后者使用 4 个操作数。完整的 FMA 运算的形式应该是 `destination = round(a * b + c)`（目标 = 舍入 `(a * b + c)`），然而 FMA3 会把 a、b 或 c 中的某个寄存器同时当成目标寄存器来用，与之相对，FMA 4 则单独用一个寄存器来表示目标。

> **提示：** 不要把 fused multiply–add 与 Intel 的 Fast Memory Access（快速内存访问）特性相混淆。

F16C 指令可以在半精度浮点数（16 位）与单精度浮点数（32 位）之间转换。有些情况下精度并不是特别重要，因此，为了节省空间可以把浮点值保存成半精度的格式，但如果要把这些值放在 SSE 寄存器里参与运算，就需要将其转为单精度格式。

上面这几种技术所支持的指令之间其实是有些重复的，于是，后来就出现了名为 AVX 的实现方式以统合这些技术。现在，我们就结束这简短的一节，继续往下讲解 AVX。

### 8.3.5　AVX

AVX（Advanced Vector Extensions，高级向量扩展）技术是基于早前的 SSE 实现方式而构建的，它大幅更新了 x86 架构。从 2011 年开始就有支持该技术的处理器发布。AVX 能够向后兼容 SSE。迄今为止出现了三套高级扩展指令，它们分别是：AVX 指令集、AVX2 指令集以及 AVX-512 指令集。这些扩展指令集较大地提升了寄存器的容量，并使得开发者能够运用更多的指令来编写程序。

表 8-21　各个 AVX 寄存器的容量

| AVX | AVX-512 | AVX2/AVX | |
|---|---|---|---|
| 二进制位 | 511....256 | 255....128 | 127....0 |
| x86 | ZMM0 | YMM0 | XMM0 |
| | ZMM1 | YMM1 | XMM1 |
| | ZMM2 | YMM2 | XMM2 |
| | ZMM3 | YMM3 | XMM3 |
| | ZMM4 | YMM4 | XMM4 |
| | ZMM5 | YMM5 | XMM5 |
| | ZMM6 | YMM6 | XMM6 |
| | ZMM7 | YMM7 | XMM7 |
| x86_64 | ZMM8 | YMM8 | XMM8 |
| | ZMM9 | YMM9 | XMM9 |
| | ZMM10 | YMM10 | XMM10 |
| | ZMM11 | YMM11 | XMM11 |
| | ZMM12 | YMM12 | XMM12 |
| | ZMM13 | YMM13 | XMM13 |
| | ZMM14 | YMM14 | XMM14 |
| | ZMM15 | YMM15 | XMM15 |
| AVX-512 | ZMM16–31 | YMM16–31 | XMM16–31 |

AVX 增设了一些 256 位的 YMM 寄存器使得开发者可以使用更多的寄存器空间。在 32 位模式下可以使用 *ymm0* 至 *ymm7* 8 个寄存器，在 64 位模式下可以使用 *ymm0* 至 *ymm15* 16 个寄存器（参见表 8-21）。AVX 还有一项新特性，就是支持三操作数的格式。大多数的 SSE 指令（比如算术指令）带有两个操作数，以加法为例，正如早前的范例程序所示，这两个操作数都充当计算的来源，然而其中必须有一个操作数同时充当计算的目标，这就使得该操作数的原值会在计算之后遭到覆盖（例如在 a、b 这两个操作数中，如果 a 同时还要充当目标

操作数，那么做完加法之后它的值就会变成 a + b)。AVX 支持三操作数格式，也就是可以把第三个操作数用作目标，从而令前两个操作数的值免于修改（例如它可以把 a + b 的结果保存到 c 中）。

AVX2 又添加了一些指令并且扩充了专门操作整数的指令，同时对三操作数格式也提供了更多支持，以便更好地执行涉及 3 个或 4 个操作数的 FMA3 或 FMA4 运算（例如 d = a * b + c）、位运算及乘法运算。

AVX-512 再度拓展了开发者可以使用的寄存器范围，令其能够访问 *zmm0* 至 *zmm31* 这 32 个 512 位的寄存器（参见表 8-21）。此外，为了兼容早前的 SSE 及 AVX 指令，它允许程序在 64 位模式下通过这些旧式的指令来访问 *zmm16* 至 *zmm31* 这 16 个新寄存器。

在数据类型上，AVX 也对压缩形式的单精度类型及双精度浮点类型做了扩充。例如 AVX 的 256 位 YMM 寄存器可以容纳多达 8 个单精度浮点数或多达 4 个双精度浮点数。虽然 AVX 向后兼容 SSE，但并不是每一种 AVX 指令都完全支持各种类型的数据。

程序 8-8 简短地演示了 AVX 运算。请注意，AVX 指令是以字母 v 开头的。更多信息请参见本章的网络资源。

程序 8-8 的效果是把 *ymm2* 寄存器的值变成 [9，9，9，9，9，9，9，9]，并把 *xmm4* 寄存器的值变成 [−1，−1，−1，−1]。

程序 8-8 AVX

```
.data
.balign 16
vectorA: .float 1.2, 3.4, 5.6, 7.8, 8.9, 9.0, 0.9, 9.8
vectorB: .float 7.8, 5.6, 3.4, 1.2, 0.1, 0.0, 8.1, -0.8
vectorC: .long 1, 1, 1, 1
vectorD: .long -2, -2, -2, -2

.text
.globl _main
_main:                              # AVX/AVX2 example

vmovaps  vectorA, %ymm0             # move vectorA to YMM0
vmovaps  vectorB, %ymm1             # move vectorA to YMM1
vaddps   %ymm0, %ymm1, %ymm2        # add vectorA and B, store in YMM2

vmovdqa  vectorC, %xmm3             # move vectorC to XMM3
vmovdqa  vectorC, %xmm4             # move vectorC to XMM4
vpsignd  vectorD, %xmm3, %xmm4      # negate sign of XMM3 values if vectorD
                                    #   values are < 0, store in XMM4
                                    # vpsign AVX support
                                    #   (CPUID for 128 or 256-bit capability)
pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

## 8.4 小结

本章综述了与 x86 架构有关的浮点数表示方式及实现办法。IEEE 754 是计算机在表示浮点数时所依据的标准，其中定义了单精度、双精度以及扩展双精度等几种格式。介绍完这些格式之后，笔者讲述了浮点数在实现手法上的演变，它经历了 x87 FPU、Streaming SIMD

Extensions（流式 SIMD 扩展）以及 AVX（高级向量扩展）等几个世代，而且还会以更快的速度继续发生变化。懂得了与浮点数架构有关的知识之后，你就可以在科学计算、图形、游戏编程等领域中以更加贴近底层的方式来操作浮点数，从而写出效率更高、性能更强的程序。

## 8.5　关键术语

Advanced Vector Extensions (AVX)(高级向量扩展)
binary coded decimal (BCD)（二进制编码的十进制数；二进码十进数）
exception（异常；例外）
eXtended Operations (XOP)
F16C
floating point (value)（浮点（值））
floating-point unit (FPU)（浮点运算器；浮点运算单元）
fused multiply–add (FMA)（积和熔加）
IEEE 754
MMX technology（MMX 技术）
normalized（正规（化）的；归约（化）的；规范（化）的；规格（化）的；标准（化）的）
not-a-number (NaN)（非数）
packed (data and operation)（压缩形式的（数据及运算））
postfix（后缀）
real number（实数）
round to nearest（向最近的值取整）
scalar (data and operation)（标量形式的（数据及运算））
significand（有效数）
single instruction, multiple data (SIMD)（单指令流多数据流；单指令多数据）
Streaming SIMD Extensions (SSE)（流式 SIMD 扩展）
subnormal number（次正规数）
top-of-stack (TOP)（栈顶）
underflow（下溢）
x87

## 8.6　重要的寄存器（32 位和 64 位）

| | |
|---|---|
| *st0* 至 *st7* | x87 寄存器，用来构成浮点栈 |
| *mm0* 至 *mm7* | MMX 寄存器，用来存放压缩形式的整数 |
| *xmm0* 至 *xmm7*，*xmm8* 至 *xmm15* | SSE 寄存器。在 32 位环境下只能使用前 8 个寄存器，在 64 位环境下这 16 个寄存器都可以使用 |
| *ymm0* 至 *ymm7*，*ymm8* 至 *ymm15* | AVX 寄存器。在 32 位环境下只能使用前 8 个寄存器，在 64 位环境下这 16 个寄存器都可以使用 |
| *zmm0* 至 *zmm15*，*zmm16* 至 *zmm31* | AVX-512 寄存器。在 32 位环境下只能使用前 16 个寄存器，在 64 位环境下这 32 个寄存器都可以使用。程序可通过 xmm 及 ymm 等写法来访问这些寄存器里的对应部分 |

## 8.7　代码回顾

本章介绍了三个代次的浮点数实现技术，也就是 x87、SSE 以及 AVX。当前大多数编译器都能很好地利用 SSE 标量指令来做基本的浮点算术。本章前面所举的那些范例程序都是 32 位的，因此，在这一节里笔者给出一个 64 位的程序，它利用 SSE 指令来计算每个月需要投入的资金。程序 8-9 所用的写法，其实与 C++ 编译器在编译 64 位的浮点运算程序时所用的办法很像。

第 8 章的补充材料中给出了该程序的 MASM 与 NASM 版本，也写出了用 C++ 语言所实现的相关函数，以及计算时所依据的投资公式。

程序 8-9 适用于 x86_64 环境的投资计算器

```
# This program calculates a monthly deposit needed to reach an investment
# value, given the future value, interest rate (%), and duration (months)
.extern _printString                    # declare external functions
.extern _printDouble
.extern _getDouble
.extern _getInt

.data
.balign 16                              # align data on 16-byte boundary
amountNeeded:   .double 0.0
interestRate:   .double 0.0
interest:       .double 0.0
monthlyPayment: .double 0.0
duration:       .quad   0               # integer data
const100:       .double 100.0
const12:        .double 12.0
const1:         .double 1.0

amountMessage:   .asciz  "       Enter Amount Needed:  $"
interestMessage: .asciz  "       Enter Interest Rate:  "
durationMessage: .asciz  "Enter Duration (in months):  "
paymentMessage:  .asciz  "    Monthly Payment Amount:  $"

.text
.globl _asmMain
_asmMain:

pushq %rbp
movq  %rsp, %rbp                        # align stack pointer

leaq amountMessage(%rip), %rdi          # pass char* to print
callq _printString
callq _getDouble                        # get and store amountNeeded
movsd %xmm0, amountNeeded(%rip)
leaq interestMessage(%rip), %rdi        # pass char* to print
callq _printString
callq _getDouble                        # get interestRate as a %

divsd %xmm1, %xmm0
movsd %xmm0, interestRate(%rip)         # store rate as decimal value

leaq durationMessage(%rip), %rdi        # pass char* to print
callq _printString
callq _getInt                           # get and store duration
movq %rax, duration(%rip)

movsd const12(%rip), %xmm0              # interest = interestRate / const12
movsd interestRate(%rip), %xmm1
divsd %xmm0, %xmm1
movsd %xmm1, interest(%rip)

movsd interest(%rip), %xmm0             # investment formula (see Ch8 Supplement)
movsd amountNeeded(%rip), %xmm1
mulsd %xmm0, %xmm1                      # interest * amount needed, then store
movsd %xmm1, %xmm2

addsd const1(%rip), %xmm0               # add 1 to interest
movq duration(%rip), %rdi
callq  pow                              # pow(interest, duration)
```

```
subsd const1(%rip), %xmm0              # subtract 1 from result
vdivsd %xmm0, %xmm2, %xmm1             # complete formula with division
movsd %xmm1, monthlyPayment(%rip)

leaq paymentMessage(%rip), %rdi        # pass char* to print
callq _printString
movsd monthlyPayment(%rip), %xmm0
callq _printDouble                     # print monthlyPayment

popq  %rbp
retq                                   # end _asmMain

_pow:                                  # pow function
pushq %rbp
movq %rsp, %rbp
movq %rdi, %rcx                        # use rcx for loop countdown
decq %rcx
movsd %xmm0, %xmm1

exp:                                   # multiply by value duration times
mulsd %xmm1, %xmm0
loop exp

popq  %rbp
retq                                   # end _pow

.end
```

## 8.8 习题

### 填空题

1. x86 处理器用_____格式表示浮点数。
2. 在浮点数的表示形式中，小数点左侧与右侧的那些数位都称为_____。
3. 为了能把指数以_____整数的形式存放，需要给真实的指数加上_____。
4. 浮点值应该加以_____（化）处理，也就是要移动其二进制形式中的小数点，使得左侧只有一个数位且该位的值是 1。
5. 8087 FPU 采用_____格式来执行算术运算。
6. 如果浮点运算的结果超出表示范围，那么就发生_____。
7. FPU 协处理器可以写成_____。
8. FPU 的控制寄存器中有两个二进制位用来控制舍入模式，这意味着总共有_____种舍入方式可供选用。
9. FPU 的寄存器栈中包含_____个寄存器，每个寄存器的宽度是_____个二进制位。
10. SSE（Streaming SIMD Extensions）技术使得_____指令可以同时操作_____数据。
11. MMX 技术引入了_____形式的数据类型。
12. AVX（Advanced Vector Extensions）技术增设了_____系列与_____系列的寄存器。
13. AVX 支持带有_____个操作数的指令。
14. 以硬件方式实现的 `destination = a + (b * c)` 运算，称为_____。
15. SIMD 的 `movss` 指令本身就可以体现出该指令所操作的是_____形式的数据，而不是压缩形式的数据。

### 判断题

1. FPU 能够用 NaN（非数的值）执行运算。
2. FPU 所检测到的任何异常总是会自动得到处理。
3. SSE 技术添加了 128 位宽的 XMM 寄存器。
4. AVX 能够向后兼容 SSE。
5. 一般来说，浮点数实现技术（也就是浮点数指令集）也能够操作整数。

## 8.9 作业

下面这些 C++ 代码是用来完成作业 8.1 与 8.2 的。在做这两道题的过程中，**不要**修改下列代码。

| Mac (Xcode) | Windows (Visual Studio)/Linux (g++) |
|---|---|
| `#include <iostream>`<br>`using namespace std;`<br><br>`extern "C" void asmMain();`<br><br>`extern "C" double getDouble(){`<br>`  double d;`<br>`  std::cin >> d;`<br>`  return d;`<br>`}`<br><br>`extern "C" void printString(char* s){`<br>`  std::cout << s;`<br>`  return;`<br>`}`<br><br>`extern "C" void printDouble(double d){`<br>`  cout << d << endl;`<br>`}`<br><br>`// main stub driver`<br>`int main(){`<br>`  asmMain();`<br>`  return 0;`<br>`}` | `#include <iostream>`<br>`using namespace std;`<br><br>`extern "C" void _asmMain();`<br><br>`extern "C" double _getDouble() {`<br>`  double d;`<br>`  std::cin >> d;`<br>`  return d;`<br>`}`<br><br>`extern "C" void _printString(char* s) {`<br>`  std::cout << s;`<br>`  return;`<br>`}`<br><br>`extern "C" void _printDouble(double d) {`<br>`  cout << d << endl;`<br>`}`<br><br>`// main stub driver`<br>`int main() {`<br>`  _asmMain();`<br>`  return 0;`<br>`}` |

### 8.9.1 计算圆柱的体积（x87）

借助上述 C++ 代码来编写程序，采用 x87 FPU 中的指令计算圆柱的体积。该程序要提示用户输入圆柱的底面半径及高度，然后据此计算体积并将其显示出来。你所写的汇编代码必须与上述 C++ 代码相链接，以运用其中的函数来实现输入及显示等功能。汇编代码要控制程序流程、调用函数并执行浮点算术，而且与运算有关的变量都必须声明在这些代码中。你可以提供一些文字信息以帮助用户输入数据并理解计算结果，用来保存这些文字信息的字符串也必须声明在汇编代码中。

圆柱体积计算公式：$V=\pi r^2 h$（$V$ 是体积，$\pi$ 是圆周率，$r$ 是底面半径，$h$ 是高）

### 8.9.2 计算轨道速度（SSE 标量指令）

借助上述 C++ 代码来编写程序，采用 SSE 中的标量指令计算环绕地球的物体所具备的轨道速度（orbital velocity）。程序要提示用户输入物体与地球表面之间的距离（单位是米），然后据此计算轨道速度，并以每秒的米数（m/s）的形式来展示计算结果。你所写的汇编代码必须与上述 C++ 代码相链接，以运用其中的函数实现输入及显示等功能。程序的流程、对函数的调用以及对浮点数所做的运算都必须用汇编代码书写，而且相关的变量也必须声明在汇编代码中。你可以提供一些文字信息以帮助用户输入数据并理解计算结果，用来保存这些文字信息的字符串也必须声明在汇编代码中。

轨道速度 $v$ 的计算公式：$v=\sqrt{\frac{Gm_e}{r}}$

其中：

$G$ 表示万有引力常数，它的值约等于 $6.6741\times10^{-11}$（单位是 $N\cdot m^2\,kg^{-2}$）

$m_e$ 表示地球的质量，它的值约等于 $5.9722\times10^{24}$（单位是 kg）

$r$ 表示轨道半径，也就是物体与地心之间的距离，它的单位是 m（米）。地球表面与地心之间的距离大约是 $6.371\times10^{6}$ m，你必须把这个值加到物体与地球表面之间的距离上。按照这些单位所算出的轨道速度 $v$，其单位是 m/s（米 / 秒）。

$$\text{mph（英里 / 小时）}=\frac{\text{m/s（米 / 秒）}}{0.44704}$$

### 8.9.3　使向量正规化（挑战题）

用汇编语言编写 32 位或 64 位的独立程序，该程序会定义一个二维向量与一个三维向量，然后对这两个向量做正规化处理。这种任务可以通过许多方式完成，不过笔者建议你利用 MMX、SSE 或 AVX 指令来做，以便发挥并行计算的优势。向量的具体取值可以由你自定。这道题还可以进行扩充，也就是再添加一项需求：计算出向量的反射向量。

对向量做正规化处理所依据的公式：$\check{\boldsymbol{V}}=\frac{\boldsymbol{V}}{|\boldsymbol{V}|}$，其中 $|\boldsymbol{V}|$ 指的是向量 $\boldsymbol{V}$ 的长度（也称为大小或模长）。

计算向量长度所依据的公式：二维向量为 $|\boldsymbol{V}|=\sqrt{x^2+y^2}$，三维向量为 $|\boldsymbol{V}|=\sqrt{x^2+y^2+z^2}$

# 第 8 章补充材料
# 第 8 章中的程序

下面这些程序可以与第 8 章正文中的 GAS 版程序相对比，这些程序使用 MASM 及 NASM 的语法书写。第 8 章正文中的程序 8-1，以及此处的程序 8-1b 所包含的三段程序，都需要与程序 8-1a 中对应的 C++ 代码相链接，以便实现汇编代码所要调用的 _printFloat 及 _printDouble 函数。请注意，针对 Windows/Visual Studio 及 Linux/g++ 环境所写的 C++ 代码，要在相关函数的开头添加下划线。程序 8-2 至程序 8-8，都是独立的汇编程序。此外还要注意，MASM 版本的代码中用到了一些与处理器及扩展有关的命令，例如 .686、.MMX、.XMM 等。

**程序 8-1a　需要与程序 8-1 及程序 8-1b 相链接的 C++ 函数代码**

```
// for Xcode on macOS (Clang/LLVM)
#include <iostream>
#include <iomanip>
using namespace std;

extern "C" void asmMain();

extern "C" void printFloat(float f){
   cout << setprecision(7) << f << endl;
}

extern "C" void printDouble(double d){
   cout << setprecision(15) << d << endl;
}

int main(){
   asmMain();
   return 0;
}
```

```
// for Visual Studio on Windows and g++ on Linux (e.g., Ubuntu)
#include <iostream>
#include <iomanip>
using namespace std;

extern "C" void _asmMain();

extern "C" void _printFloat(float f){
   cout << setprecision(7) << f << endl;
}

extern "C" void _printDouble(double d){
   cout << setprecision(15) << d << endl;
}

int main(){
   _asmMain();
   return 0;
}
```

程序 8-1b　x87 FPU

GAS

```
.extern _printFloat
.extern _printDouble

.data
value: .float 1.2

.bss
.lcomm r_value, 4
.lcomm f_result, 4
.lcomm d_result, 8

.text
.globl _asmMain
_asmMain:
push %ebp
movl %esp, %ebp

finit
fldpi
flds value
fadd %st(1), %st
fists r_value

fstps f_result
push $0
push f_result
call _printFloat
add $8, %esp

fstpl d_result
push d_result + 4
push d_result
call _printDouble
add $8, %esp

pop %ebp
ret
.end
```

MASM

```
.686
.MODEL FLAT, C
.STACK 4096

_printFloat  PROTO C
_printDouble PROTO C

.data
value    REAL4 1.2
r_value  REAL4 ?
f_result REAL4 ?
d_result REAL8 ?

.code
_asmMain PROC
push ebp
mov ebp, esp

finit
fldpi
fld value
fadd ST(0), ST(1)
fist r_value

fstp f_result
push 0
push f_result
call _printFloat
add esp, 8

fstp d_result
push DWORD PTR [d_result + 4]
push DWORD PTR [d_result]
call _printDouble
add esp, 8

pop ebp
ret
_asmMain ENDP
END
```

NASM

```
extern _printFloat
extern _printDouble

section .data
value: dd 1.2

section .bss
r_value:  resd 1
f_result: resd 1
d_result: resq 1

section .text
global _asmMain
_asmMain:
push ebp
mov ebp, esp

finit
fldpi
fld DWORD [value]
fadd ST0, ST1
fist DWORD [r_value]

fstp DWORD [f_result]
push 0
push DWORD [f_result]
call _printFloat
add esp, 8

fstp QWORD [d_result]
push DWORD [d_result + 4]
push DWORD [d_result]
call _printDouble
add esp, 8

pop ebp
ret
```

程序 8-2　x87 FPU 的舍入模式

GAS

```
.data
default_cw: .word 0x0000
nearest:    .word 0x0000
down:       .word 0x0400
up:         .word 0x0800
zero:       .word 0x0C00

.bss
.lcomm result, 4

.text
.globl _main
_main:

finit
```

MASM

```
.686
.MODEL FLAT, C
.STACK 4096

ExitProcess PROTO stdcall,
dwExitCode:DWORD

.data
default_cw  WORD 0000h
nearest     WORD 0000h
down        WORD 0400h
up          WORD 0800h
zero        WORD 0C00h
result DWORD ?
```

NASM

```
section .data
default_cw:  dw 0000h
nearest:     dw 0000h
down:        dw 0400h
up:          dw 0800h
zero:        dw 0C00h

section .bss
result: resd 1

section .text
global _main
_main:

finit
```

| GAS | MASM | NASM |
|---|---|---|
| fstcw default_cw<br>movw default_cw, %ax<br>andb $0b11110011, %ah<br>orw %ax, nearest<br>orw %ax, down<br>orw %ax, up<br>orw %ax, zero<br><br>xor %ax, %ax<br>movw default_cw, %ax<br>movw nearest, %ax<br>movw down, %ax<br>movw up, %ax<br>movw zero, %ax<br><br>fldcw nearest<br><br>fldpi<br>frndint<br>fists result<br>movl result, %eax<br><br>pushl $0<br>subl $4, %esp<br>movl $1, %eax<br>int $0x80<br>.end | .code<br>_main PROC<br><br>finit<br>fstcw default_cw<br>mov ax, default_cw<br>and ah, 11110011b<br>or nearest, ax<br>or down, ax<br>or up, ax<br>or zero, ax<br><br>xor ax, ax<br>mov ax, default_cw<br>mov ax, nearest<br>mov ax, down<br>mov ax, up<br>mov ax, zero<br><br>fldcw nearest<br><br>fldpi<br>frndint<br>fist result<br>mov eax, result<br><br>INVOKE ExitProcess, 0<br>_main ENDP<br>END | fstcw WORD [default_cw]<br>mov ax, [default_cw]<br>and ah, 11110011b<br>or [nearest], ax<br>or [down], ax<br>or [up], ax<br>or [zero], ax<br><br>xor ax, ax<br>mov ax, [default_cw]<br>mov ax, [nearest]<br>mov ax, [down]<br>mov ax, [up]<br>mov ax, [zero]<br><br>fldcw [nearest]<br><br>fldpi<br>frndint<br>fist DWORD [result]<br>mov eax, [result]<br><br>mov eax, 1<br>mov ebx, 0<br>int 80h |

程序 8-3 MMX

| GAS | MASM | NASM |
|---|---|---|
| .data<br>w1: .word 1, 2, 3, 4<br>w5: .word 5, 6, 7, 8<br>result: .quad 0<br><br>.text<br>.globl _main<br>_main:<br><br>movq w1, %mm0<br>movq w5, %mm1<br><br>paddsw %mm1, %mm0<br>movq %mm0, result<br><br>pushl $0<br>subl $4, %esp<br>movl $1, %eax<br>int $0x80<br>.end | .686<br>.MMX<br>.MODEL FLAT, C<br>.STACK 4096<br><br>ExitProcess PROTO stdcall,<br>dwExitCode:DWORD<br><br>.data<br>w1 WORD 1, 2, 3, 4<br>w5 WORD 5, 6, 7, 8<br>result QWORD 0<br><br>.code<br>_main PROC<br><br>lea eax, w1<br>lea ebx, w5<br>movq mm0, [eax]<br>movq mm1, [ebx]<br><br>paddsw mm0, mm1<br>movq result, mm0 | section .data<br>w1: dw 1, 2, 3, 4<br>w5: dw 5, 6, 7, 8<br>result: dq 0<br><br>section .text<br>global _main<br>_main:<br><br>movq mm0, [w1]<br>movq mm1, [w5]<br><br>paddsw mm0, mm1<br>movq [result], mm0<br><br>mov eax, 1<br>mov ebx, 0<br>int 80h |

| GAS | MASM | NASM |
| --- | --- | --- |
| | INVOKE ExitProcess, 0<br>_main ENDP<br>END | |

程序 8-4　通过一条 SSE 指令在压缩形式的数据上并行地执行多次加法操作

| GAS | MASM | NASM |
| --- | --- | --- |
| .data<br>.balign 16<br>vectorA: .float 1.2, 3.4, 5.6, 7.8<br>vectorB: .float 7.8, 5.6, 3.4, 1.2<br><br>.bss<br>.lcomm result, 128<br><br>.text<br>.globl _main<br>_main:<br><br>movaps vectorA, %xmm0<br>addps  vectorB, %xmm0<br>movaps %xmm0, result<br><br>pushl $0<br>subl $4, %esp<br>movl $1, %eax<br>int $0x80<br>.end | .686<br>.XMM<br>.MODEL FLAT, C<br>.STACK 4096<br><br>ExitProcess PROTO stdcall, dwExitCode:DWORD<br><br>.data<br>ALIGN 16<br>vectorA REAL4 1.2, 3.4, 5.6, 7.8<br>vectorB REAL4 7.8, 5.6, 3.4, 1.2<br>result  REAL4 ?, ?, ?, ?<br><br>.code<br>_main PROC<br><br>movaps xmm0, vectorA<br>addps  xmm0, vectorB<br>movaps result, xmm0<br><br>INVOKE ExitProcess, 0<br>_main ENDP<br>END | section .data<br>align 16<br>vectorA: dd 1.2, 3.4, 5.6, 7.8<br>vectorB: dd 7.8, 5.6, 3.4, 1.2<br><br>section .bss<br>result: resd 4<br><br>section .text<br>global _main<br>_main:<br><br>movaps xmm0, [vectorA]<br>addps  xmm0, [vectorB]<br>movaps [result], xmm0<br><br>mov eax, 1<br>mov ebx, 0<br>int 80h |

程序 8-5　用 SSE 指令对标量执行运算

| GAS | MASM | NASM |
| --- | --- | --- |
| .data<br>.balign 16<br>valueA: .float 1.2<br>pi: .double 3.14159265358979<br><br>.bss<br>.lcomm result, 4<br><br>.text<br>.globl _main<br>_main:<br><br>movss  valueA, %xmm0<br>addss  valueA, %xmm0<br>movss  %xmm0, result<br>movl  result, %eax<br><br>movsd  pi, %xmm0 | .686<br>.XMM<br>.MODEL FLAT, C<br>.STACK 4096<br><br>ExitProcess PROTO stdcall, dwExitCode:DWORD<br><br>.data<br>ALIGN 16<br>valueA REAL4 1.2<br>pi  REAL8 3.14159265358979<br>result REAL4 ?<br><br>.code<br>_main PROC<br><br>movss  xmm0, valueA<br>addss  xmm0, valueA | section .data<br>align 16<br>valueA: dd 1.2<br>pi: dq 3.14159265358979<br><br>section .bss<br>result: resd 1<br><br>section .text<br>global _main<br>_main:<br><br>movss  xmm0, [valueA]<br>addss  xmm0, [valueA]<br>movss  [result], xmm0<br>mov  eax, DWORD [result]<br><br>movsd  xmm0, [pi] |

| GAS | MASM | NASM |
|---|---|---|
| pushl $0<br>subl $4, %esp<br>movl $1, %eax<br>int $0x80<br>.end | movss result, xmm0<br>mov eax, result<br><br>movsd xmm0, pi<br><br>INVOKE ExitProcess, 0<br>_main ENDP<br>END | mov eax, 1<br>mov ebx, 0<br>int 80h |

程序 8-6 SSE3

| GAS | MASM | NASM |
|---|---|---|
| .data<br>.balign 16<br>vectorA: .float 1.2, 3.4, 5.6, 7.8<br>vectorB: .float 7.8, 5.6, 3.4, 1.2<br><br>.text<br>.globl _main<br>_main:<br><br>movaps vectorA, %xmm0<br>movaps vectorB, %xmm1<br>haddps %xmm1, %xmm0<br><br>pushl $0<br>subl $4, %esp<br>movl $1, %eax<br>int $0x80<br>.end | .686<br>.XMM<br>.MODEL FLAT, C<br>.STACK 4096<br><br>ExitProcess PROTO stdcall,<br>dwExitCode:DWORD<br><br>.data<br>ALIGN 16<br>vectorA REAL4 1.2, 3.4, 5.6, 7.8<br>vectorB REAL4 7.8, 5.6, 3.4, 1.2<br><br>.code<br>_main PROC<br><br>movaps xmm0, vectorA<br>movaps xmm1, vectorB<br>haddps xmm0, xmm1<br><br>INVOKE ExitProcess, 0<br>_main ENDP<br>END | section .data<br>align 16<br>vectorA: dd 1.2, 3.4, 5.6, 7.8<br>vectorB: dd 7.8, 5.6, 3.4, 1.2<br><br>section .text<br>global _main<br>_main:<br><br>movaps xmm0, [vectorA]<br>movaps xmm1, [vectorB]<br>haddps xmm0, xmm1<br><br>mov eax, 1<br>mov ebx, 0<br>int 80h |

程序 8-7 SSE4

| GAS | MASM | NASM |
|---|---|---|
| .data<br>.balign 16<br>vectorA: .float 1.2, 3.4, 5.6, 7.8<br>vectorB: .float 7.8, 5.6, 3.4, 1.2<br><br>.text<br>.globl _main<br>_main:<br><br>movaps vectorA, %xmm0<br>roundps $1, %xmm0, %xmm1<br>cvtps2dq %xmm1, %xmm2<br><br>movaps vectorB, %xmm3<br>roundps $2, %xmm3, %xmm4<br>cvtps2dq %xmm4, %xmm5<br><br>pmulld %xmm2, %xmm5 | .686<br>.XMM<br>.MODEL FLAT, C<br>.STACK 4096<br><br>ExitProcess PROTO stdcall,<br>dwExitCode:DWORD<br><br>.data<br>ALIGN 16<br>vectorA REAL4 1.2, 3.4, 5.6, 7.8<br>vectorB REAL4 7.8, 5.6, 3.4, 1.2<br><br>.code<br>_main PROC<br><br>movaps xmm0, vectorA<br>roundps xmm1, xmm0, 1<br>cvtps2dq xmm2, xmm1 | section .data<br>align 16<br>vectorA: dd 1.2, 3.4, 5.6, 7.8<br>vectorB: dd 7.8, 5.6, 3.4, 1.2<br><br>section .text<br>global _main<br>_main:<br><br>movaps xmm0, [vectorA]<br>roundps xmm1, xmm0, 1<br>cvtps2dq xmm2, xmm1<br><br>movaps xmm3, [vectorB]<br>roundps xmm4, xmm3, 2<br>cvtps2dq xmm5, xmm4<br><br>pmulld xmm5, xmm2 |

| GAS | MASM | NASM |
| --- | --- | --- |
| pushl $0<br>subl $4, %esp<br>movl $1, %eax<br>int $0x80<br>.end | movaps xmm3, vectorB<br>roundps xmm4, xmm3, 2<br>cvtps2dq xmm5, xmm4<br><br>pmulld xmm5, xmm2<br><br>INVOKE ExitProcess, 0<br>_main ENDP<br>END | mov eax, 1<br>mov ebx, 0<br>int 80h |

程序 8-8　AVX

| GAS | MASM | NASM |
| --- | --- | --- |
| .data<br>.balign 16<br>vectorA: .float 1.2, 3.4,<br>5.6, 7.8, 8.9, 9.0, 0.9, 9.8<br>vectorB: .float 7.8, 5.6,<br>3.4, 1.2, 0.1, 0.0, 8.1, -0.8<br>vectorC: .long 1, 1, 1, 1<br>vectorD: .long -2, -2, -2, -2<br><br>.text<br>.globl _main<br>_main:<br><br>vmovaps vectorA, %ymm0<br>vmovaps vectorB, %ymm1<br>vaddps %ymm0, %ymm1, %ymm2<br><br>vmovdqa vectorC, %xmm3<br>vmovdqa vectorC, %xmm4<br>vpsignd vectorD, %xmm3,<br>%xmm4<br><br>pushl $0<br>subl $4, %esp<br>movl $1, %eax<br>int $0x80<br>.end | .686<br>.XMM<br>.MODEL FLAT, C<br>.STACK 4096<br><br>ExitProcess PROTO stdcall,<br>dwExitCode:DWORD<br><br>.data<br>vectorA REAL4 1.2, 3.4, 5.6, 7.8,<br>8.9, 9.0, 0.9, 9.8<br>vectorB REAL4 7.8, 5.6, 3.4, 1.2,<br>0.1, 0.0, 8.1, -0.8<br>vectorC DWORD 1, 1, 1, 1<br>vectorD DWORD -2, -2, -2, -2<br><br>.code<br>_main PROC<br><br>lea eax, vectorA<br>lea ebx, vectorB<br>vmovups ymm0, YMMWORD PTR [eax]<br>vmovups ymm1, YMMWORD PTR [ebx]<br>vaddps ymm2, ymm1, ymm0<br><br>lea eax, vectorC<br>lea ebx, vectorD<br>vmovdqa xmm3, XMMWORD PTR [eax]<br>vmovdqa xmm4, XMMWORD PTR [ebx]<br>vpsignd xmm4, xmm3, vectorD<br><br>INVOKE ExitProcess, 0<br>_main ENDP<br>END | section .data<br>align 16<br>vectorA: dd 1.2, 3.4, 5.6,<br>7.8, 8.9, 9.0, 0.9, 9.8<br>vectorB: dd 7.8, 5.6, 3.4,<br>1.2, 0.1, 0.0, 8.1, -0.8<br>vectorC: dd 1, 1, 1, 1<br>vectorD: dd -2, -2, -2, -2<br><br>section .text<br>global _main<br>_main:<br><br>vmovaps ymm0, [vectorA]<br>vmovaps ymm1, [vectorB]<br>vaddps ymm2, ymm1, ymm0<br><br>vmovdqa xmm3, [vectorC]<br>vmovdqa xmm4, [vectorC]<br>vpsignd xmm4, xmm3,<br>vectorD]<br><br>mov eax, 1<br>mov ebx, 0<br>int 80h |

在刚才的程序 8-8 中，GAS 与 NASM 版本所使用的数据是经过对齐的，因此需要专门采用处理这种数据的指令，也就是 VMOVAPS（Vector Move Aligned Packed Single-precision，对已经对齐且各元素均以压缩形式的单精度值来表示的向量进行移动），反之，MASM 版本所使用的数据未经对齐，因此也应该采用与之相应的指令来处理，这就是 VMOVUPS（Vector Move Unaligned Packed Single-precision，对未经对齐且各元素均以压缩形式的单精度值来表示的向量进行移动）。MASM 版本的程序还演示了这样一种用法：它把向量的地址放置在寄存器中，然后用 YMMWORD 及 XMMWORD 命令将该地址所指向

的向量数据复制到相关的 YMM 或 XMM 寄存器里。每个平台都有可能针对数据的对齐方式提出各自的要求，而且开发者在该平台中所能使用的命令以及操纵数据时所用的办法或许也不一样。与平台有关的详细信息请参见各平台的文档。

## 投资计算器（x86_64）

本节所演示的程序可以与第 8 章的代码回顾中提到的程序 8-9 对照，它们都是为 x86_64 环境开发的，当时只给出了 GAS 版本，现在把 MASM 及 NASM 版本也列出来。附录 E 讲解了怎样把 C++ 代码同汇编代码链接起来，请参照其中涉及 64 位环境的部分来构建与你的平台相适应的程序。

投资公式：$\dfrac{(\text{利率} * \text{需要达到的总额})}{(pow((\text{利率}+1), \text{投资时长})-1)}$

**程序 8-9a　适用于 x86_64 的投资计算器程序中所用到的 C++ 函数**

```
// for Xcode on macOS (Clang/LLVM)
#include <iostream>
#include <iomanip>

extern "C" void asmMain();

extern "C" void printString(char* s){
   std::cout << s;
   return;
}

extern "C" void printDouble(double d){
   std::cout.setf(std::ios::fixed);
   std::cout.setf(std::ios::showpoint);
   std::cout.precision(2);
   std::cout << d << std::endl;
   return;
}

extern "C" double getDouble(){
   double d;
   std::cin >> d;
   return d;
}

extern "C" int getInt(){
   int i;
   std::cin >> i;
   return i;
}

//main stub driver
int main(){
   asmMain();
   return 0;
}

// for Visual Studio on Windows and g++ on Linux (e.g., Ubuntu)
#include <iostream>
#include <iomanip>

extern "C" void _asmMain();

extern "C" void _printString(char* s){
   std::cout << s;
   return;
}

extern "C" void _printDouble(double d){
   std::cout.setf(std::ios::fixed);
   std::cout.setf(std::ios::showpoint);
   std::cout.precision(2);
```

```
    std::cout << d << std::endl;
    return;
}

extern "C" double _getDouble(){
    double d;
    std::cin >> d;
    return d;
}

extern "C" int _getInt(){
    int i;
    std::cin >> i;
    return i;
}

//main stub driver
int main(){
    _asmMain();
    return 0;
}
```

MASM 版本的汇编代码有一些特殊的地方值得注意：

- 有待显示的第一条信息，也就是 amountMessage 所表示的那条信息，需要以回车符（值是 13）及换行符（值是 10）开头。
- _asmMain 及 _pow 函数必须预留 32 字节的 shadow space，否则程序会出现异常。
- MASM 版本的代码没有明确体现出 RIP 相对寻址机制。

**程序 8-9b　适用于 x86_64 的投资计算器程序，GAS 版本**

Part 1

```
.extern _printString
.extern _printDouble
.extern _getDouble
.extern _getInt

.data
.balign 16
amountNeeded:    .double 0.0
interestRate:    .double 0.0
interest:        .double 0.0
monthlyPayment: .double 0.0
duration:        .quad   0
const100:        .double 100.0
const12:         .double 12.0
const1:          .double 1.0

amountMessage:    .asciz
"       Enter Amount Needed:  $"
interestMessage: .asciz
"       Enter Interest Rate:  "
durationMessage: .asciz
"Enter Duration (in months):  "
paymentMessage:  .asciz
"    Monthly Payment Amount:  $"

.text
.globl _asmMain
_asmMain:

pushq  %rbp
movq   %rsp, %rbp
```

Part 2

```
leaq durationMessage(%rip), %rdi
callq  _printString
callq _getInt
movq %rax, duration(%rip)

movsd const12(%rip), %xmm0
movsd interestRate(%rip), %xmm1
divsd %xmm0, %xmm1
movsd %xmm1, interest(%rip)

movsd interest(%rip), %xmm0
movsd amountNeeded(%rip), %xmm1
mulsd %xmm0, %xmm1
movsd %xmm1, %xmm2

addsd const1(%rip), %xmm0
movq duration(%rip), %rdi
callq _pow
subsd const1(%rip), %xmm0
vdivsd %xmm0, %xmm2, %xmm1
movsd %xmm1, monthlyPayment(%rip)

leaq paymentMessage(%rip), %rdi
callq  _printString
movsd monthlyPayment(%rip), %xmm0
callq  _printDouble

popq   %rbp
retq

_pow:
pushq  %rbp
```

Part 1

```
leaq amountMessage(%rip), %rdi
callq _printString
callq _getDouble
movsd %xmm0, amountNeeded(%rip)

leaq interestMessage(%rip), %rdi
callq _printString
callq _getDouble

movsd const100(%rip), %xmm1
divsd %xmm1, %xmm0
movsd %xmm0, interestRate(%rip)
```

Part 2

```
movq %rsp, %rbp
movq %rdi, %rcx
decq %rcx
movsd %xmm0, %xmm1

exp:
mulsd %xmm1, %xmm0
loop exp

popq   %rbp
retq

.end
```

程序 8-9c 适用于 x86_64 的投资计算器程序，MASM 版本

Part 1

```
ExitProcess PROTO
_printString PROTO
_printDouble PROTO
_getDouble PROTO
_getInt PROTO

.data
align 16
amountNeeded   REAL8 0.0
interestRate   REAL8 0.0
interest       REAL8 0.0
monthlyPayment REAL8 0.0
duration       QWORD 0
const100       REAL8 100.0
const12        REAL8 12.0
const1         REAL8 1.0

amountMessage   BYTE
13, 10, "       Enter Amount Needed:  $",0
interestMessage BYTE
"       Enter Interest Rate:  ",0
durationMessage BYTE
"Enter Duration (in months):  ",0
paymentMessage  BYTE
"    Monthly Payment Amount:  $",0

.code
_asmMain PROC

push rbp
sub rsp, 20h
lea rbp, [rsp + 20h]

lea rcx, amountMessage
call _printString
call _getDouble
movsd amountNeeded, xmm0

lea rcx, interestMessage
call _printString
call _getDouble
```

Part 2

```
lea rcx, durationMessage
call _printString
call _getInt
mov duration, rax

movsd xmm0, const12
movsd xmm1, interestRate
divsd xmm1, xmm0
movsd interest, xmm1

movsd xmm0, interest
movsd xmm1, amountNeeded
mulsd xmm1, xmm0
movsd xmm2, xmm1

addsd xmm0, const1
mov rcx, duration
call _pow
subsd xmm0, const1
vdivsd xmm1, xmm2, xmm0
movsd monthlyPayment, xmm1

lea rcx, paymentMessage
call _printString
movsd xmm0, monthlyPayment
call _printDouble

lea rsp, [rbp]
pop rbp
ret
_asmMain ENDP

_pow PROC
push rbp
sub rsp, 20h
lea rbp, [rsp + 20h]
dec rcx
movsd xmm1, xmm0

exp:
mulsd xmm0, xmm1
loop exp
```

| Part 1 | Part 2 |
| --- | --- |
| movsd xmm1, const100<br>divsd xmm0, xmm1<br>movsd interestRate, xmm0 | lea rsp, [rbp]<br>pop rbp<br>ret<br>_pow ENDP<br><br>END |

**程序 8-9d　适用于 x86_64 的投资计算器程序，NASM 版本**

| Part 1 | Part 2 |
| --- | --- |
| extern _printString<br>extern _printDouble<br>extern _getDouble<br>extern _getInt<br><br>section .data<br>align 16<br>amountNeeded:　　dq 0.0<br>interestRate:　　dq 0.0<br>interest:　　dq 0.0<br>monthlyPayment: dq 0.0<br>duration:　　dq 0<br>const100:　　dq 100.0<br>const12:　　dq 12.0<br>const1:　　dq 1.0<br><br>amountMessage:　db<br>"　　Enter Amount Needed:　$",0<br>interestMessage: db<br>"　　Enter Interest Rate:　",0<br>durationMessage: db<br>"Enter Duration (in months):　",0<br>paymentMessage:　db<br>"　Monthly Payment Amount:　$",0<br><br>section .text<br>global _asmMain<br>_asmMain:<br><br>push　rbp<br>mov rbp, rsp<br><br>lea rdi, [rel amountMessage]<br>call _printString<br>call _getDouble<br>movsd [rel amountNeeded], xmm0<br><br>lea rdi, [rel interestMessage]<br>call _printString<br>call _getDouble<br><br>movsd xmm1, [rel const100]<br>divsd xmm0, xmm1<br>movsd [rel interestRate], xmm0 | lea rdi, [rel durationMessage]<br>call _printString<br>call _getInt<br>mov [rel duration], rax<br><br>movsd xmm0, [rel const12]<br>movsd xmm1, [rel interestRate]<br>divsd xmm1, xmm0<br>movsd [rel interest], xmm1<br><br>movsd xmm0, [rel interest]<br>movsd xmm1, [rel amountNeeded]<br>mulsd xmm1, xmm0<br>movsd xmm2, xmm1<br><br>addsd xmm0, [rel const1]<br>mov rdi, [rel duration]<br>call _pow<br>subsd xmm0, [rel const1]<br>vdivsd xmm1, xmm2, xmm0<br>movsd [rel monthlyPayment], xmm1<br><br>lea rdi, [rel paymentMessage]<br>call _printString<br>movsd xmm0, [rel monthlyPayment]<br>call _printDouble<br><br>pop rbp<br>ret<br><br>_pow:<br>push rbp<br>mov rbp, rsp<br>mov rcx, rdi<br>dec rcx<br>movsd xmm1, xmm0<br><br>exp:<br>mulsd xmm0, xmm1<br>loop exp<br><br>pop rbp<br>ret |

# 内联汇编与宏

## 目标

- 了解内联汇编的用途
- 编写内联式的汇编语句
- 在宏与函数调用之间对比
- 在汇编代码中创建宏

## 网络资源

**维基资源**

- https://wiki.osdev.org/Inline_Assembly
- https://en.wikipedia.org/wiki/Memory_ordering

**技术资源**

- https://msdn.microsoft.com/en-us/library/4ks26t93.aspx（Microsoft 的内联汇编器）
- https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html（Clang/GCC）
- https://gcc.gnu.org/onlinedocs/gcc/Machine-Constraints.html（x86 Machine Constraints）
- http://www.ibm.com/developerworks/library/l-ia/（Linux 系统的 x86 内联汇编）
- https://sourceware.org/binutils/docs/as/Macro.html（GAS 宏）
- https://msdn.microsoft.com/en-us/library/d4chx59e.aspx（MASM 宏）
- http://www.nasm.us/doc/nasmdoc4.html（NASM 宏）

## 9.1 简介

这一章将帮助大家熟悉内联汇编[⊖]与宏这两个编程概念。首先讲解内联汇编的用途，然后给出一些代码范例，并指出做内联汇编时应该注意的一些问题。接下来，在宏与函数之间对比，并通过范例来展示如何在汇编代码中创建宏。开发者可以用这一章所讲的技巧更为灵活地编写并调整程序代码。

## 9.2 内联汇编

内联汇编是一种在高级语言中嵌入汇编代码的办法。本书在提到高级语言时一直是用 C++ 语言举例的，这一章也是如此。尽管 C++ 本身已经很灵活、很强大了，但有的时候，程序员还是想用更贴近底层的办法来编程，比如想要自己控制优化机制与硬件。

还有一种办法也能把汇编语言的代码同高级语言的代码结合起来，这在本书第 6 章及附录 E 中提到过，就是用汇编代码来撰写函数，并将其放在一份文件中，然后在 C++ 代码中

⊖ inline Assembly，也称为行内汇编、内嵌汇编，下同。——译者注

调用这些函数。不过，这种办法需要执行相关的准备工作，而且还涉及函数调用，因此会引发一些开销。与之相比，撰写内联式的汇编代码在某些情况下可能更为简单。注意是在某些情况下，也就是说，具体哪一种办法好要依照具体情形来确定。

### 9.2.1　与编译器有关的细节

撰写内联式的汇编代码意味着我们是在以高级语言为主的代码库中工作的，因此，这种情况下要关注的应该是编译器而不是汇编器。每个编译器所支持的功能以及所拥有的特性都与其他编译器有所区别，因此，本章要从这些编译器里选出几个典型来加以讲解。我们在讨论汇编器的时候选了三种最为常见的产品，也就是GAS、MASM及NASM，而对于编译器我们同样打算选三个典型，也就是Clang、GCC及Visual C++。这样选取能够令笔者举出很多有用的示例，而且可以涵盖大多数操作系统。其中，Clang与GCC这两者在形式与功能上基本相同。

**提示：**截至Xcode 4.2版本，Clang仍是Xcode默认的编译器，当然它也可以用在Linux与Windows系统中。Clang与GCC默认采用AT&T汇编语法，Visual C++采用Intel语法。（前两者也可以切换到Intel语法。）

各种编译器会采用不同的做法来处理内联式的汇编代码。Xcode的做法是把Clang编译器与LLVM所集成的汇编器搭配起来使用，由后者处理内嵌的汇编语句。GCC的做法与之不同，在它的翻译流水线（参见第1章）中，有一个步骤是先根据高级语言的代码来创建汇编文件，然后再将其翻译为机器代码。因此，如果高级语言的代码所内嵌的汇编代码在编译过程中没有受到修改或优化，那么GCC实际上就相当于把这一部分汇编代码复制到它所输出的汇编文件中（这种文件的后缀名是.s），进而把这些代码与从高级语言转换而来的那些汇编代码一起交给GNU汇编器（GNU Assembler，也就是GAS）处理。Microsoft的Visual Studio采用Visual C++内置的汇编器处理内联汇编语句，而不会专门采用MASM那样单独的汇编器来做。

**学习指南：**Microsoft的x64 C/C++编译器不支持内联式的汇编代码，它改用编译器内部函数来完成相应的底层指令。详情可参考附录I及https://msdn.microsoft.com/en-us/library/wbk4z78b.aspx。

### 9.2.2　内联式的汇编语句

C++编译器通常用`asm`或`__asm`关键字表示内联式的汇编语句。现在来看一个简单的例子，范例9-1。它设定调试断点，使得程序能够在运行到该处时中断。

**范例9-1　用内联汇编语句设定断点**

| Clang/GCC | Visual C++ |
| --- | --- |
| `asm ("int $3");` | `__asm int 3` |

与一般的汇编相似，内联汇编也有很多微妙的地方值得注意。请观察表9-1所给出的代码模板，并将其与表9-2中的具体范例相对比。

表 9-1 内联汇编的代码模板

| | |
|---|---|
| 基本形式的 Clang/GCC 内联汇编 | `asm ("instruction 1 \n\t instruction 2");`<br><br>`asm ("instruction 1 \n\t"`<br>`"instruction 2");` |
| 扩展形式的 Clang/GCC 内联汇编 | `asm ("instruction 1 \n\t"`<br>`"instruction 2"`<br>`: output operands`<br>`: input operands`<br>`: clobbers);` |
| 单语句形式的 Visual C++ 内联汇编 | `__asm instruction 1` |
| 语句块形式的 Visual C++ 内联汇编 | `__asm {instruction 1`<br>`instruction 2};` |

表 9-2 内联汇编的具体代码范例

| | |
|---|---|
| 基本形式的 Clang/GCC 内联汇编 | `asm ("movl $42, %eax \n\t inc %eax");`<br><br>`asm ("movl $42, %eax \n\t"`<br>`"inc %eax");` |
| 扩展形式的 Clang/GCC 内联汇编 | `asm ("movl $42, %%eax \n\t"`<br>`"inc %%eax"`<br>`:`<br>`:`<br>`: "%eax");` |
| 单语句形式的 Visual C++ 内联汇编 | `__asm mov eax, 42` |
| 语句块形式的 Visual C++ 内联汇编 | `__asm {mov eax, 42`<br>`inc eax};` |

Clang/GCC 汇编器要求 asm 关键字后面必须写一对圆括号及一个分号，而汇编代码则要以字符串的形式写在一对双引号中，并放置在这对圆括号中。换行符与制表符用来分隔字符串中的各条汇编语句。Visual C++ 要求 __asm 关键字的后面要么直接写汇编语句，要么跟上一对花括号。如果汇编语句只有一条可以直接写出来，若有很多条则需放在花括号中。最后的分号可有可无。

**编程知识：** 开发者在使用 Clang/GCC 时可能会开启 -ansi 或 -std 这样的编译选项，从而导致 asm 及 inline 等关键字遭到禁用，这种情况下，可以把 asm 关键字写成 __asm__。

请注意，Clang 与 GCC 是有基本形式与扩展形式[⊖]之分的。尽量不要使用基本形式的 asm 内联，因为在访问全局变量时可能要做不同的处理，而且，这种形式还会假设汇编代码不修改通用的寄存器。然而基本形式的 asm 内联也有一些好处，由于它不一定非要写在 C/C++ 函数中，因此可以用来执行汇编器指令并撰写全局的汇编函数。此外，如果 C/C++ 函数声明成了 naked 函数，编译器就不会为其生成开场及收场代码（prologue and epilouge code），这就要求开发者必须使用基本形式的 asm 内联来为这样的函数撰写定制的开场及收场代码。扩展形式的 asm 内联可以更精细地控制与汇编代码有关的输出与输入，并指明代码所用到的寄存器。因此为了使格式保持一致，本节将采用扩展形式的 asm 来撰写适用于 Clang/GCC 编译器的内联汇编代码。Clang/GCC 要求扩展形式的 asm 代码必须写在 C/C++

⊖ 也称为基础形式与扩充形式。下同。——译者注

函数中，Visual C++ 则要求所有的内联汇编代码都必须这样写。

**提示：** GCC 并不会解析内联式的汇编指令，因此它根本就不知道这些语句是否有效。Clang/LLVM 虽然会对语法有误的内联汇编代码给出调试信息，但这些错误信息很难解读，而且出错的位置也不易确定，因为它是用模板化的设计来处理 asm 语句的，而且 GDB、LLDB 等调试器也不会单步地进入内联汇编语句中去调试，而是把整个 asm 视为一个步骤。如果要单步调试，可以在想插入断点的地方手工添加“int $3”指令。Visual Studio 无须使用 INT 即可单步调试 x86 内联汇编指令。

程序 9-1　简单的内联式汇编代码

Clang/GCC

```
#include <iostream>
using namespace std;

int main(){

int var1 = 1234, var2;

   asm ("mov %1, %%eax \n\t"
        "add $2, %%eax \n\t"
        "mov %%eax, %0 \n\t"
        :"=r" (var2)        /* %0: Out */
        :"r" (var1)         /* %1: In */
        :"%eax"             /* Overwrite */
       );

   cout << var2 << endl;
   return 0;
}
```

Visual C++

```
#include <iostream>
using namespace std;

int main(){

int var1 = 1234, var2;

    __asm {mov eax, var1
           add eax, 2
           mov var2, eax
    }

   cout << var2 << endl;
   return 0;
}
```

程序 9-1 很短小，它对比了不同的编译器在内联汇编的写法上具体有哪些区别，并演示了怎样使用内联的汇编代码来访问外围函数中的局部变量。

**编程知识：** 内联汇编中的注释既可以按汇编代码自身的格式来写也可以按 C++ 的格式来写，然而笔者强烈建议采用后一种写法。

首先可以看到：内联式的汇编代码是出现在 C++ 函数内部的，就本例来说，出现在 main 函数内部。至于这些汇编代码如何访问函数中的变量则要依具体的编译器来定。Visual C++ 可以直接访问 C 语言式的变量，也就是说，如果要引用函数里的 var1 变量，直接写 var1 就好。Clang/GCC 访问变量的方式比较复杂。对内联的汇编代码来说，用于输出和用于输入的变量都必须在模板中的相应列表中说明。当然了，这两份列表也可以留空，如果其中有内容，那么与下文将要提到的 clobbers 列表合计起来不得超过 30 项。

在 Clang/GCC 的内联式汇编代码中对涉及输出与输入的变量做出说明

```
[asmSymbolicName] "constraint" (CvariableName)
```

其中，asmSymbolicName 可以是任何一个有效的标识符名称，它也可以与外围代码中已经定义的 C 语言标识符同名，从而使内联式的汇编代码能够用同样的名称来引用这个变量。constraint（约束或限定）这一部分用来指定与操作数的用法及放置地点有关的信息。它可以是一个或一系列字母（这些字母的具体含义参见本章的网络资源）。如果某个操作数

是用来输出数值的，那么它必须先以修饰符开头，然后才能写上一个或多个字母。最常用的两个修饰符是‘=’与‘+’，前者表示该操作数在执行汇编指令之前的初始值并不重要，仅仅是供汇编代码写入数据的，后者表示汇编代码既要读取该操作数的初始值，又要向其中写入数据（也就是说这个参数既用于输入又用于输出）。有的时候，系统会先把那些仅仅用于输入的参数处理完，然后再处理那些供汇编代码输出数据的参数，这就促使它有可能会复用某些寄存器，令其在不同的阶段表示不同的参数。这样一来，如果其中有某个输出参数在处理输入参数的过程中已经写进了值，那么这个值就会把用同一个寄存器所表示的输入参数所具备的初始值给覆盖掉[⊖]。为了避免这种覆写的情况，可以在‘=’或‘+’后面写上‘&’符号，使得系统能够正确地实现这样的参数[⊜]。

在 Clang/GCC 的内联汇编模板中，还有一个部分叫作 clobbers 列表，列表中需要写上内联的汇编代码所修改的寄存器，以防止系统误以为这些寄存器的值不会为汇编代码所修改，从而执行一些相关的优化措施。系统可能会生成指令来保存它们的值，以便在执行完内联的 asm 语句后能够予以恢复。系统在选用寄存器来实现汇编代码中用于输入及输出的操作数时会避开 clobbers 列表中指明的项目。如果你所写的内联式汇编代码确实会顺带影响到某个寄存器的值，但你却没有将其写在 clobbers 列表中，那么程序可能会出错或发生意外的结果。

> 📖 **学习指南：**与本节有关的一些详细信息请参阅 GCC 开发文档中谈论 extended asm 的那一部分。比方说，clobbers 列表中可以出现“cc”与“memory”这样特殊的项目，它们的用法，可以参见：https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html。

请看范例 9-2 中的 4 个小例子。我们假设每个例子的 asm 语句之前都有 `int var1 = 1234, var2;` 这样一行代码。asm 中内嵌的汇编语句用来将输入参数（也就是 var1）的值复制到 *eax* 寄存器，并给该寄存器加 2，然后将相加所得的结果保存到输出参数（也就是 var2）中。

**范例 9-2　在 Clang/GCC 的内联汇编代码中访问外围函数中的变量**

| 范　例 | 代　码 | 反汇编 |
|---|---|---|
| **范例 9-2-1**　采用从 0 开始的序号来指代各参数，并限定这些参数只能用寄存器来实现（或者说只能保存到寄存器里）。clobbers 列表中还指出了这段代码所要修改的寄存器 | `asm ("  mov %1, %%eax \n\t"`<br>`"  add $2, %%eax \n\t"`<br>`"  mov %%eax, %0 \n\t"`<br>`: "=r" (var2)`<br>`: "r" (var1)`<br>`: "%eax"`<br>`);` | `movl $0x4d2, 0x1c(%esp)`<br>`movl 0x1c(%esp), %edx`<br>**`movl %eax, 0x14(%esp)`**<br>`movl %edx, %eax`<br>`addl $0x2, %eax`<br>`movl %eax, %edx`<br>`movl %edx, 0x18(%esp)` |

⊖ 例如在内联 mov %1, %0 及 add %2, %0 这样两条汇编指令时，由于 %0 变量的初始值并不影响汇编代码的运行效果，因此它应该视为一个用于输出的参数，而 %1 及 %2 则应视为用于输入的参数。假如系统决定采用同一个寄存器表示 %0 及 %2，那么执行完 mov 指令后 %2 的初始值就会遭到覆盖，使得 %2 现在的值实际上与 %1 及 %0 相同，因此，add%2, %0 就成了 add %1, %0。如果用 & 来约束 %0，那么系统就会明白：这个用于输出的参数在处理完所有的输入参数之前，其值是可能遭到修改的。因此，就不会把它和某个用于输入的参数（例如 %2）拿同一个寄存器来实现，以防后者的初始值为前者所覆盖。——译者注

⊜ GCC 的开发文档，将其称为 early-clobber。——译者注

| 范　例 | 代　码 | 反汇编 |
|---|---|---|
| **范例 9-2-2**　采用从 0 开始的序号来指代各参数，并限定这些参数只能用寄存器来实现。clobbers 列表中没有提到这段代码还会修改某个寄存器 | `asm ("  mov %1, %%eax \n\t"`<br>`"  add $2, %%eax \n\t"`<br>`"  mov %%eax, %0 \n\t"`<br>`: "=r" (var2)`<br>`: "r" (var1)`<br>`:`<br>`);` | `movl $0x4d2, 0x1c(%esp)`<br>`movl 0x1c(%esp), %edx`<br>`movl %edx, %eax`<br>`addl $0x2, %eax`<br>`movl %eax, %edx`<br>`movl %edx, 0x18(%esp)` |
| **范例 9-2-3**　采用符号名称来指代各参数并限定这些参数只能用内存来实现。clobbers 列表中还指出了这段代码所要修改的寄存器 | `asm ("  mov %[asm_var2], %%eax \n\t"`<br>`"  add $2, %%eax \n\t"`<br>`"  mov %%eax, %[asm_var1] \n\t"`<br>`: [asm_var2] "=m" (var2)`<br>`: [asm_var1] "m" (var1)`<br>`: "%eax"`<br>`);` | `movl $0x4d2, 0x1c(%esp)`<br>**`movl %eax, 0x14(%esp)`**<br>`movl 0x1c(%esp), %eax`<br>`addl $0x2, %eax`<br>`movl %eax, 0x18(%esp)` |
| **范例 9-2-3**　采用符号名称来指代各参数并限定这些参数可以用寄存器或内存来实现。clobbers 列表中没有提到这段代码还会修改某个寄存器） | `asm ("  mov %[var2], %%eax \n\t"`<br>`"  add $2, %%eax \n\t"`<br>`"  mov %%eax, %[var1] \n\t"`<br>`: [var2] "=rm" (var2)`<br>`: [var1] "rm" (var1)`<br>`:`<br>`);` | `movl $0x4d2, 0x1c(%esp)`<br>`movl 0x1c(%esp), %edx`<br>`movl %edx, 0x14(%esp)`<br>`movl 0x14(%esp), %eax`<br>`addl $0x2, %eax`<br>`movl %eax, 0x18(%esp)` |

范例 9-2-1 与范例 9-2-2 都采用从 0 开始的编号来指代外围函数中的各变量，由于我们并没有给这些变量指定符号名称，因此，需要按照其在 asm 的输出及输入列表中的出现顺序，分别以 %0、%1、%2 这样的写法来称呼。输出列表中的首个变量叫作 %0，排列完这份列表之后，继续排列输入列表中的参数直至最后一个参数为止。在范例 9-2-3 中，我们给内联的汇编代码所用到的那些参数分别起了另外一个名字以示区分，例如 C 语言式的变量 var1，在内联式的汇编代码中就叫作 asm_var1。范例 9-2-4 也给内联的汇编代码所使用的参数设定了符号名称，只不过这些名称与对应的 C 语言式变量名完全一样。

范例 9-2-1 与范例 9-2-2 给参数加上了约束符 'r'，意思是说在执行语句之前应该以某个通用的整数寄存器来存放这个参数。通过反汇编所得的指令可知，编译器是采用 *edx* 寄存器来满足这项约束的。范例 9-2-3 所加的约束修饰符是 'm'，意思是说把该变量保存到内存（memory）中就可以了，无须通过寄存器做中介。范例 9-2-4 同时指定了这两种约束。通过反汇编后的指令我们可以看出，编译器所选用的存放方式是符合约束要求的。

表 9-3 列出了 x86 常用的约束修饰符。（其中列出的这些不一定每个编译器都支持，而没有列出来的选项却有可能受到某些编译器支持，具体情况要根据编译器来定。）

**表 9-3　内联式汇编代码中经常用到的 x86 约束修饰符**

| 约　束 | 含　义 |
|---|---|
| r 或 R | 通用寄存器 |
| q | 某个受支持的寄存器中的低 8 位（例如 *al*） |
| Q | 某个受支持的寄存器中的高 8 位（例如 *ah*） |
| a, b, c, d | *ax/eax/rax, bx/ebx/rbx, cx/ecx/rcx, dx/edx/rdx* |
| S | *si/esi/rsi* |
| D | *di/edi/rdi* |
| A | *ax/eax/rax* 及 *dx/edx/rdx*，用来存放双字结果 |
| f | 任何一个 x87 FPU 栈寄存器 |
| t | x87 FPU 栈的栈顶（也就是 *st0*） |

（续）

| 约　束 | 含　义 |
| --- | --- |
| u | 从 x87 FPU 栈的栈顶算起，第二个寄存器（也就是 *st1*） |
| y | 任何一个 MMX 寄存器 |
| x | 任何一个 SSE 寄存器 |
| m | 任何一个内存操作数 |
| o | 可以用作偏移量（offsettable）的内存操作数 |
| V | 不用作偏移量的内存操作数 |
| g | 任何一个通用寄存器、内存或立即数 |
| X | 任何操作数 |
| p | 可以当成有效内存地址（也就是指针，pointer）来使用的操作数 |
| i | 可以当成立即常量（immediate constant）来使用的整数操作数，例如“i”(5) |
| F | 可以当成立即常量来使用的浮点操作数 |

范例 9-2-1 与范例 9-2-3，都在 clobbers 列表中提到了 *eax* 寄存器，用以表明内联的汇编代码会在执行过程中修改该寄存器的值。系统在看到这样的说明之后可能会生成相应的指令，以便在执行内联汇编代码之前先将寄存器中的值保存起来，从而能够在执行完毕之后予以恢复。

根据 AT&T 汇编语法，寄存器以一个 % 符号开头。不过，内联式的汇编代码中寄存器需要用两个 % 符号开头，也就是要在前面加上 %%，若只加一个 %，表示的则是这段内联代码所用的参数（或变量）。之所以要这样写，是因为内联代码会把 % 号当作转义符使用，以便输出它后面的符号所表示的字符，因此“%%”输出的是单个 % 字符，而“%=”则会针对代码库中的每一段 asm 代码输出一个独特的数字（这个数字可以用来创建标签，以供其他代码引用）。此外，“%{”、“%|”及“%}”这三种写法分别用来输出 {、| 及 } 字符。这三种字符必须用 % 来转义，假如直接写在内联的汇编代码中，那么系统就会将其当成特殊字符并解读成用来表示各种汇编方言的结构。如果要产生 %eax 这样的汇编代码，那么在内联的汇编语句中应该写成 %%eax，不过，在 clobbers 列表中提到 *eax* 时，只写一个 % 就好。

程序 9-2 演示了怎样用内联的汇编代码来做浮点运算。大家可以从中看 x87 及 SSE 标量指令的用法。

**程序 9-2　用内联的汇编代码做浮点运算**

Clang/GCC

```
#include <iostream>
using namespace std;

int main(){

float var = 0;

// x87 floating-point example
   asm("finit \n\t"
       "fldpi \n\t"
       "fsqrt \n\t"
       "fstps %[var]"
       :[var] "=m" (var)
       :
       :
      );
```

Visual C++

```
#include <iostream>
using namespace std;

int main(){

float var = 0;

// x87 floating-point example
   __asm{finit
         fldpi
         fsqrt
         fstp var
   };

   cout << var << endl;
```

| Clang/GCC | Visual C++ |
| --- | --- |
| cout << var << endl;<br><br>// SSE scalar example<br>var = 1.2;<br><br>asm("movss %0, %%xmm0 \n\t"<br>"addss %0, %%xmm0 \n\t"<br>"movss %%xmm0, %0 \n\t"<br>: "+x" (var)<br>:<br>: "%xmm0"<br>);<br><br>cout << var << endl;<br><br>return 0;<br>} | // SSE scalar example<br>var = 1.2;<br><br>__asm{movss xmm0, var<br>addss xmm0, var<br>movss var, xmm0<br>};<br><br>cout << var << endl;<br><br>return 0;<br>} |

### 9.2.3　为各种汇编方言提供支持

前面说过，Clang 与 GCC 默认使用 AT&T 语法规则，而 Visual C++ 则使用 Intel 语法规则。不过，Clang 与 GCC 能够在内联的汇编代码中使用多种汇编方言[⊖]。由 AT&T 语法切换到 Intel 语法能够使代码变得好懂一些。范例 9-3 对比了这两种写法。

范例 9-3　Clang/GCC 允许开发者采用各种方言来撰写内联的汇编代码

| AT&T dialect | Intel dialect |
| --- | --- |
| asm(".att_syntax \n\t"<br>"mov $5, %%eax \n\t"<br>"mov $3, %%ebx \n\t"<br>"add %%ebx, %%eax \n\t"<br>:<br>:<br>: "%eax", "%ebx"<br>); | asm(".intel_syntax \n\t"<br>"mov eax, 5 \n\t"<br>"mov ebx, 5 \n\t"<br>"add eax, ebx \n\t"<br>:<br>:<br>: "eax", "ebx"<br>); |

从这个例子可以看出，要想采用某种特定的语法来撰写内联汇编代码，一种简单的办法是在第一条命令中写上 `.att_syntax` 以表示这段代码用的是 AT&T 语法，或者写上 `.intel_syntax` 以表示这段代码用的是 Intel 语法[⊖]。

**编程知识：Clang 与 GCC 可以通过 -masm 这个编译选项来指定内联汇编代码所采用的方言。-masm=att 与 -masm=intel 分别表示 AT&T 语法及 Intel 语法。**

针对 x86 做开发的时候，Clang/GCC 允许在同一行内联汇编代码中使用多种方言，以便同时支持 AT&T 及 Intel 语法。这样的代码需要像范例 9-4 一样以 `{ dialect0 | dialect1 ...}` 的形式书写。

---

⊖ dialect。对于汇编语言这个大的**语言**来说，AT&T 语法的汇编语言和 Intel 语法的汇编语言是其下的两种**方言**。——译者注

⊖ 如果编译 Intel 版本的时候遇到错误，可以试着把 `.intel_syntax` 改为 `.intel_syntax noprefix`。——译者注

**范例 9-4 Clang/GCC 允许在同一行内联汇编代码中使用多种方言**

```
asm("mov{l $5, %%ebx | ebx, 5} \n\t"
    "mov{l $3, %%ecx | ecx, 3} \n\t"
    "xchg{l}\t {%%}ebx,\t {%%}ecx \n\t"
    :
    :
    :
   );
```

这段代码明确展示出了两种方言的区别。AT&T 版本给 mov 加了后缀'l'，用以表明该指令针对的是 long 数据（或者说长度为 32 个二进制位的数据）。此外还可以看出，指令的各部分之间是可以用制表符'\t'来分隔的。本例中的 xchg 指令就是如此。

### 9.2.4 注意事项

撰写内联汇编代码时要注意一些问题。首先，要考虑是否便于移植。C++ 这样的语言有个优点，就是很适合做跨平台的开发，然而一旦在其中嵌入汇编语句就不那么容易移植了。因此，如果程序必须跨平台，那么应该通过 #define、#include 以及头文件等机制把代码设计好，使得内联式的汇编语句能够正确地运作在各种平台上。

第二个值得注意的问题是编译器支持哪些汇编命令。虽然 x86 指令集中的各项指令通常都能在内联的汇编代码中使用，但是汇编命令可就不一定了。比方说，Visual C++ 用的并不是 MASM 而是它自己所集成的汇编器，因此，这种开发环境下所能使用的命令与 MASM 有所不同。声明数据类型所用的那些命令，例如 DB/BYTE、DD/DWORD 等，是不受支持的，因为数据必须声明在 C++ 代码中。此外，MASM 中的 MACRO、REPT、DUP 及 STRUCT 命令也不能用在 Visual C++ 的内联汇编代码中，不过 EVEN 与 ALIGN 命令却可以。要想知道每一种编译器以及它所集成的汇编器都支持什么样的指令，请参阅各自的开发文档。

第三个大问题在于优化。优化通常有两种形式，一种是由编译器所做的软件优化，另一种是由 CPU 所做的硬件优化。就软件方面的优化而言，如果你在 Debug Mode（调试模式）下运行内联的汇编代码，那么程序此时所表现出的行为与你本来想要实现的效果相比，应该是很接近的，尽管编译器可能会自行添加一些步骤（例如保存寄存器中的数据、依照约束来为变量做复制，或是对齐栈等），但这并不会大幅改变程序的效果。反之，如果是在 Release Mode（发布模式、发行模式）下运行，那么编译器就有可能运用一些复杂的优化措施来判断代码，并将那些它自以为多余的步骤及输出信息删掉。

你可以运用各种办法来限制这种优化措施，使得内联式的汇编代码不会遭到编译器修改，进而能够按照原样得以执行。其中一个办法是像范例 9-5 这样把 asm 语句声明成 volatile，这样就能禁止编译器做出某些优化了，不过 CPU 层面的优化依然有可能发生。对于 Clang/GCC 来说，基本形式的内联汇编全都默认为 volatile，而扩展形式的汇编则不是。

**范例 9-5 把 asm 声明为 volatile，以防编译器运用某些优化技术**

```
asm volatile ("movl $42, %%eax \n\t"
              "inc %%eax"
              :
              :
              : "%eax");
```

还有一种阻止编译器做优化的办法叫作 memory barrier（内存屏障）或 memory fence（内存围栏、内存栅栏），它最适合用在当前那些支持多线程的处理器上。这种屏障 / 围栏机制是想防止编译器或 CPU 重新安排指令之间的顺序。深入讨论 volatile 及 memory barrier 并不是本书的任务，然而笔者会在接下来的学习指南中给出一些正式的及非正式的网络资源以帮助大家切入这两个话题。

内联汇编对于开发者来说是个相当有用的工具，但是用起来要特别小心。

> 📖 **学习指南：** 下面给出一些技术文档的网址，以供你由此出发来探索 volatile 及 memory barrier 技术。
>
> - https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Volatile
> - https://brooker.co.za/blog/2013/01/06/volatile.html
> - https://en.wikipedia.org/wiki/Memory_barrier
> - https://www.kernel.org/doc/Documentation/memory-barriers.txt
> - http://bruceblinn.com/linuxinfo/MemoryBarriers.html
> - http://yarchive.net/comp/linux/memory_barriers.html
> - https://msdn.microsoft.com/en-us/library/f20w0x5e(v=vs.140).aspx

## 9.3　宏

**宏**（macro）是另一种实用的汇编技巧，它可以视为一个模块或一系列指令，并根据名称来加以调用。汇编代码中的宏的运作方式与 C++ 代码中的内联函数类似，都是用其中所包含的一系列语句或指令来替换位于调用点的宏名或函数名。（注意，内联函数是 C++ 本身的一种机制，不要与 C++ 代码中嵌入的内联汇编相混淆。）每调用一次宏就要发生一次代码替换或代码展开。

### 9.3.1　对比宏与函数

函数（参见第 6 章）与宏之间有什么区别？函数适合实现那种比较长而且需要频繁执行的代码。无论函数调用多少次，都只需要在程序中保留一份代码。每次调用函数时，系统会把控制权转移给内存中的这个函数。调用函数涉及传递参数、建栈以及清理栈等工作，从而会产生一定的开销。为了提高效率，这些比较长且比较复杂的函数代码只在程序中保留一份就可以了。

宏适合实现那种比较短而且需要频繁执行的代码。对程序做汇编的时候，汇编器每看到一个宏就会把该宏所代表的那一系列指令展开到这里，于是同样一段代码就有可能多次出现在程序中，这样做的好处是省去了调用函数所需的开销。宏与函数有一个共同点，就是都可以接受参数（argument 或 parameter[⊖]），不过函数的参数是在运行程序的时候才传递过去的，而宏的参数则是在做汇编的时候就已经替换好了。

### 9.3.2　定义并调用宏

表 9-4 演示了三种汇编器所支持的宏语法。尤其要注意怎样在宏代码中引用参数。GAS

⊖ 前者主要指调用时所传入的实际参数，后者主要指声明时所宣告的形式参数。——译者注

的做法是给参数前面加上 \ 符号，例如参数 arg1 需要写成 \arg1，NASM 采用 [%#] 的格式来指代参数，其中的 # 部分是从 1 开始的编号，例如第一个参数 arg1 就叫作 [%1]。MASM 直接按照参数的名称来引用。

表 9-4　怎样在三种汇编器中定义宏

| | |
|---|---|
| GAS | .macro identifier arg1, arg2...<br># instructions<br># args referenced as \arg1, \arg2...<br>.endm |
| MASM | identifier MACRO arg1, arg2...<br>; instructions<br>; args referenced as arg1, arg2...<br>ENDM |
| NASM | %macro identifier argCount<br>; args referenced as [%1], [%2]...<br>%endmacro |

程序 9-3 演示了如何用宏实现简单的加法。

程序 9-3　用宏来实现加法

GAS

```
.macro intAdd dest,
source1, source2
    movl \source1, %eax
    addl \source2, %eax
    movl %eax, \dest
.endm

.data
intA: .long 2
intB: .long 4
intC: .long 3
intD: .long 7
result: .long 0

.text
.globl _main
_main:

intAdd result, intA, intB
intAdd result, intC, intD

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096

ExitProcess PROTO,
dwExitCode:DWORD

intAdd MACRO dest, source1,
source2
    mov eax, source1
    add eax, source2
    mov dest, eax
ENDM

.DATA
intA DWORD 2
intB DWORD 4
intC DWORD 3
intD DWORD 7
result DWORD 0

.CODE
_main PROC

intAdd result, intA, intB
intAdd result, intC, intD

INVOKE ExitProcess, 0
_main ENDP
END
```

NASM

```
%macro intAdd 3
   mov eax, [%2]
   add eax, [%3]
   mov [%1], eax
%endmacro

section .data
intA: dd 2
intB: dd 4
intC: dd 3
intD: dd 7
result: dd 0

section .text
global _main
_main:

intAdd result, intA,
intB
intAdd result, intC,
intD

mov eax, 1
mov ebx, 0
int 80h
```

在程序 9-3 中，除去为了退出程序而写的例行指令以外，_main 函数中只有两行代码，这两行代码都是为了调用 intAdd 宏而写的。在对这段代码做汇编时，系统会将宏里的指令展开，使得 _main 函数变为范例 9-6 中的样子。

范例 9-6　将程序 9-3 中的宏展开

| GAS | MASM | NASM |
| --- | --- | --- |
| `_main:`<br><br>`movl intA, %eax`<br>`addl intB, %eax`<br>`movl %eax, result`<br>`movl intC, %eax`<br>`addl intD, %eax`<br>`movl %eax, result` | `_main PROC`<br><br>`mov eax, intA`<br>`add eax, intB`<br>`mov result, eax`<br>`mov eax, intC`<br>`add eax, intD`<br>`mov result, eax` | `_main:`<br><br>`mov eax, DWORD [intA]`<br>`add eax, DWORD [intB]`<br>`mov [result], eax`<br>`mov eax, DWORD [intC]`<br>`add eax, DWORD [intD]`<br>`mov [result], eax` |

宏可以写在程序段以外，也可以写在其中，无论采用哪种写法都应该在整个程序的范围内保持一致。还有一种管理宏的办法，是把所有的宏或者相关的一组宏单独放在一份文件中，然后通过 INCLUDE 汇编命令（参见第 7 章的代码回顾）将其包含进来。

MASM 的宏还具备一项特性，就是可以在参数后面写上 `:REQ`，以要求调用方必须指定这个参数而不能将其忽略。

MASM

```
identifier MACRO arg1:REQ
```

宏有很多用途，其中比较合理的几种是用来定制函数的开场及收场代码、给函数传递参数，以及表示那些写起来稍显麻烦但需要反复使用的单行指令。

## 9.4　小结

这一章讲述了怎样在 C++ 代码中内联汇编指令以及如何在汇编代码中创建宏。内联汇编与宏都是很有用的技巧，可以帮助开发者完成特定的任务。不过，与面对其他编程技巧时一样，你也必须清楚这两项技巧的好处与坏处（例如兼容问题、运行开销等）从而明智地使用它们。最后，要注意一项很基本的编程常识：C++ 的内联函数与内联汇编不是一回事，前者是 C++ 本身的机制，可以把内联函数里的 C++ 代码展开到外围的 C++ 代码中，这就好比汇编语言的宏可以把宏里的汇编指令展开到外围的指令中一样。

## 9.5　关键术语

AT&T syntax（AT&T 语法）
basic asm（基本形式的 asm；基础形式的 asm）
Clang
clobbers（内联汇编指令在执行过程中所要修改的内容）
constraints（约束）
extended asm（扩展形式的 asm；扩充形式的 asm）
GCC
inline Assembly（内联汇编；行内汇编）
Intel syntax（Intel 语法）
macro（宏；巨集）
memory barriers（内存屏障）
memory fences（内存围栏；内存栅栏）
Visual C++
volatile

## 9.6　习题

### 填空题

1. 许多编译器都采用它自身所_____的汇编器来处理内联式的汇编代码。

2. 内联式的汇编代码（或代码块）结构通常以_____关键字开头。
3. Clang 及 GCC 编译器支持两种形式的汇编，一种是_____形式，另一种是_____形式。
4. 在编写内联式的汇编代码时，可以通过_____来指定与 I/O 操作数的用法及其放置地点有关的信息。
5. 对于 Clang/GCC 的内联汇编代码来说，在描述输出操作数的那份列表中需要用_____来阐明每个操作数的读 / 写状态，以表明它是仅仅用于输出，还是既用作输入，又用作输出。
6. 在内联式汇编模板的输出列表中，如果用“rm”来约束某个操作数，那么意味着系统既可以把该操作数放在_____中，也可以将它放在_____中。
7. 如果开发者要求系统必须把某个操作数放在 SSE 寄存器里（或者说用 SSE 寄存器来表示这个操作数），那么应该施加_____约束。
8. 在汇编语言之下，采用 AT&T 语法的汇编语言与采用 Intel 语法的汇编语言，可以视为两种_____。
9. x86 指令集里的各种指令，一般来说是可以用在内联汇编代码中的。与指令相比，_____就显得较为局限，因为其中有很多都不受编译器支持。
10. 开发者可以用_____关键字来声明内联的 asm 语句，从而禁止编译器采取某些优化措施。
11. 汇编语言中的_____机制，在运作原理上与 C++ 语言中的内联函数相似。
12. 函数与宏都可以接受_____。
13. 在 GAS、MASM 及 NASM 代码中调用宏的时候，应该先写出宏的_____，然后写出它的各个参数，这些参数之间用逗号隔开。
14. 汇编程序中的宏既可以放在_____外面，也可以放在它内部。
15. _____汇编器可以用 REQ 来修饰宏的参数，从而要求调用方在使用这个宏的时候必须指明该参数的值。

**判断题**

1. 用 INT 指令来设定调试断点的时候，所采用的中断代号应该是 1。
2. 对于 Visual C++ 的内联汇编代码来说，凡是为代码所修改的寄存器都必须写在 clobbers 列表中。
3. 对于 Clang/GCC 的内联汇编代码来说，它可以把外围 C++ 代码中的某个变量名用作符号名称，从而在汇编指令中使用该名称来指代变量。
4. 如果开发者用 volatile 关键字来声明某一段内联式的 asm 汇编代码，那么任何优化措施都绝对不会运用在这段代码上。
5. 在 Visual C++ 的内联汇编代码中可以使用 MACRO 命令。

## 9.7 作业

### 9.7.1 用内联式的汇编代码做浮点运算

编写一个包含内联汇编代码的 C++ 程序。该程序应该通过 C++ 代码从用户那里获取两个浮点值。然后，程序进入内联汇编语句，以 FPU/SSE/AVX 指令来对用户输入的值做各种运算。每执行完一次运算，就应该把这次运算的结果保存到该程序所定义的某个变量中。将整段的内联汇编代码执行完之后，程序要通过 C++ 代码来打印刚才的各项运算结果。具体的运算由你自己选择。笔者建议你在运算过程中使用 π 这样的常数，并运用 FSQRT、SQRTSS 及 SQRTSD 等指令。

### 9.7.2 用宏来做浮点运算

编写汇编程序，在其中定义 3 个宏并分别测试（也就是调用）。第 1 个宏要对两个参数做浮点数减法。第 2 个宏要对两个参数做浮点数乘法。第 3 个宏由你自己来定。运算所涉及的参数可以由程序

中定义的数据来充当。运算结果可以放在内存中，也可以打印到控制台。所有的代码都必须用汇编语言写。

### 9.7.3　用内联的汇编代码实现加密与解密（挑战题）

编写 C++ 程序，把一份文本文件当成输入信息，由用户来选择是对其加密还是对其解密。程序根据用户所选的动作，执行相应的内联汇编代码以便加密或解密文件中的数据，最后把处理过的整份文件打印出来。加密 / 解密的过程至少要由 5 个步骤组成。你可以考虑用 XOR 运算或移位运算来对字符做加密或解密。

# 与处理器及体系结构有关的高级话题

## 目标

- 理解计算机体系结构与操作系统结构之间有何关联
- 了解系统管理寄存器
- 知道处理器的不同模式之间有什么区别
- 分辨各种内存模型
- 以汇编指令、程序库及 API 来做系统调用
- 以不同的路径来处理系统调用

## 网络资源

**维基资源**

- https://en.wikipedia.org/wiki/X86#Operating_modes
- https://en.wikipedia.org/wiki/X86_memory_segmentation
- https://en.wikipedia.org/wiki/System_call
- https://en.wikipedia.org/wiki/Global_Descriptor_Table
- http://wiki.osdev.org/GDT_Tutorial
- https://en.wikipedia.org/wiki/Interrupt
- http://wiki.osdev.org/Interrupt_Descriptor_Table

**技术资源**

- http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (Intel Developer Manuals)
- https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI (AMD64 ABI)
- https://msdn.microsoft.com/library/windows/desktop/hh920508.aspx (Windows API Index)

## 10.1 简介

本章有两个目标，它们对于理解汇编语言及计算机系统结构来说是很关键的。首先，要提供与处理器及体系结构有关的细节，以补充其他章节（尤其是第 2 章）所谈到的一些话题。笔者将会介绍系统寄存器、处理器模式以及内存模型等方面的知识。其次，要讲解与中断及系统调用（第 9 章提到过系统调用）有关的编程知识，令你能够更好地完成底层的编程任务。笔者会给出一些程序及范例代码，以演示怎样在各种平台上以各种方式执行系统调用。这一章要讲解许多奇妙的知识，而且会出现大量的首字母缩略词。

## 10.2　处理器与系统的机能

### 10.2.1　系统寄存器

第 2 章讲过 x86 及 x86_64 架构中的寄存器，但是当时并没有提到专为系统而设的寄存器。这一节就要简单地介绍某些 x86 系统寄存器，而且要再次介绍段寄存器（segment register）。这些知识可以帮助你更好地理解本章的其他内容。

系统寄存器包括下面几组：

- **控制寄存器**（control register）：用来表示处理器的模式以及与当前正在执行的任务有关的一些特征
  - *cr0* 至 *cr4*，其中 *cr1* 暂时保留不用。
  - *cr8* 寄存器，也叫作任务优先级寄存器（task priority register，TPR），用来安排外部中断的优先顺序，该寄存器只在 64 位模式下使用。
- **内存管理寄存器**（memory-management register）：用来指出保护模式的描述符表所在的位置
  - *gdtr*：全局描述符表寄存器（Global Descriptor Table Register）。
  - *ldtr*：局部描述符表寄存器（Local Descriptor Table Register）。
  - *idtr*：中断描述符表寄存器（Interrupt Descriptor Table Register）。
  - *tr*：任务寄存器（Task Register）。
- **特定于机器的寄存器**[⊖]（Machine-specific register，MSR）——用来控制并报告处理器的执行情况。其中某些寄存器与系统调用有明确的关系（后面还会再讲一些与 MSR 有关的知识）。

> **编程知识：**每一个内存管理寄存器都有对应的加载及存储指令，也就是说，刚才提到的 4 个寄存器可以分别通过 LGDT/SGDT、LLDT/SLDT、LIDT/SIDT 及 LRT/STR 来加载或存储。其中的加载指令仅供系统软件使用，以便将描述符表的线性内存地址载入相应的寄存器。一开始，所有的内存管理寄存器都默认设为 0x0h，并且会把尺寸限制设为 0xFFFFh。然后，处理器会在初始化的过程中把 *gdtr* 及 *idtr* 应该拥有的实际取值加载到这两个寄存器中，加载完毕后它们的值就不发生变化了，而 *ldtr* 及 *tr* 的值则会根据任务来切换。存储指令同样是供系统软件使用的，不过，应用软件在确有必要的时候也可以使用这些指令。

**段寄存器**（segment register）是一种 16 位宽的寄存器，代码段（code segment）、数据段（data segment）、栈段（stack segment）以及附加段（extra segment）都有对应的段寄存器，分别称为 *cs*、*ds*、*ss*、*es*，此外，还有两个通用的段寄存器叫作 *fs* 与 *gs*，这两者是由 Intel 80386 所添加的。*es* 是执行 MOVS、CMPS 及 SCAS 等字符串操作时默认使用的段寄存器，而 *fs* 与 *gs* 的用途则没有明确为硬件所定义。

对于刚才提到的这些系统寄存器与段寄存器来说，本章主要会从处理器模式、内存模型及系统调用等方面来讲解它们。其中，系统寄存器是提供给系统软件而不是给应用软件使用的，段寄存器基本上也是这样，只不过有的情况下应用软件也确实会用到。

---

⊖ 也写为 Model-specific register，特定于模型的寄存器、特定于型号的寄存器。——译者注

### 10.2.2 处理器模式

在整个 x86 时代，处理器模式都随着指令集与系统机能而逐渐演化。表 10-1 列出了 x86/x86_64 所支持的主要几种处理器模式，并写出了首个支持该模式的处理器，以及此模式能够在多大的内存空间中寻址。除了这几种模式外还有其他一些模式，例如系统管理模式以及虚拟 8086 模式，然而我们在这里重点关注三种主要的模式，也就是**实模式**（Real Mode）、**保护模式**（Protected Mode）及**长模式**（Long Mode）。Intel 的开发文档，经常把长模式称为 IA-32e。

表 10-1 处理器模式

| 模　式 | 实　模　式 | 保护模式 | 386 保护模式 | 长　模　式 |
|---|---|---|---|---|
| 第一个支持该模式的处理器 | Intel 8086 | Intel 80286 | Intel 80386 | AMD Opteron |
| 可寻址的内存空间 | $2^{20}$（1 megabyte，1MB） | $2^{24}$（16 megabytes，16 MB） | $2^{32}$（4 gigabytes，4GB） | 实际上：$2^{48}$（256 terabytes，256 TiB）<br>理论上：$2^{64}$（16 exabytes，16 EB）[⊖] |

Intel 8086 是一款支持 x86 指令集的 16 位处理器，它运行在实模式下，这意味着软件能够直接访问物理内存空间、I/O 通道以及周边硬件。之所以不限制访问是因为软件有时要直接与硬件沟通，例如在系统启动的时候。

> **提示：**当前的 x86 及 x86_64 处理器为了保持向后兼容，确实会以实模式来发起 boot 流程，但是很快就会切换到保护模式或长模式。这些处理器大多通过 UEFI 来配置硬件，不过，也可以用老式的 BIOS 方式来配置。

从 Intel 80286 开始，所有的 x86 处理器都支持保护模式，这也是 32 位 x86 系统的默认模式。i286 的保护模式有一些限制。一旦进入该模式就无法切换回实模式了，而且内存访问也变得复杂起来，这会占用一些时钟周期，使得处理器变慢。此外，可以寻址的内存空间依然较少。Intel 80386 完善了该模式并解决了刚才提到的那些问题。当前的处理器所支持的保护模式正是从 386 演化而来的，当然其间也经历了一些改进。

AMD Opteron 处理器支持长模式，从而可以使用 x86_64 指令集中的指令以及与之相关的寄存器。该模式必须搭配 64 位操作系统。长模式之下有几种小模式（sub-mode，子模式），其中**64 位模式**（64-bit Mode）适用于 64 位程序，而**兼容模式**（Compatibility Mode）则是一种增强版的保护模式，适用于 32 位及 16 位程序。在不使用长模式的情况下，x86_64 处理器是可以支持实模式及保护模式的。

> **学习指南：**附录 G 解释了怎样查询 CPU 是否具备某些扩展特性，你可以由此判断处理器支不支持长模式。

### 10.2.3 内存模型

计算机系统所用的内存模型[⊖]要根据可以使用的处理器模式以及操作系统的支持情况来

⊖ 目前的内存地址只用到了 64 位中的 48 位，因此，尽管理论上可以表示多达 16EB 的内存空间，但实际上只能在 256TB 的内存空间中寻址。

⊖ 这里的模型，也称为模式。下同。——译者注

决定。常见的两种模型叫作**分段**（segmented）模型及**平面**（flat）模型。分段内存模型会把地址空间分成不同的区段或者说段落，以供程序中的各个部分使用，例如数据可以放在一个区段中，代码可以放在另一个区段中。这种模型通过段基址加偏移量的方式来指定内存地址。

平面内存模型会把整个内存空间展示成一块连续的区域以供开发者及程序来使用，这种模型也叫作**线性**（linear）模型。所有的内存地址都是直接以它在线性空间中的位置来确定的，而不通过段与段中的偏移量确定。大多数计算机架构都采用平面内存模型，而 x86 架构则在这两种模型之间摇摆。

这两种模型的目标都是把逻辑地址（也就是抽象的引用）映射成物理地址（也就是实际的位置）。对于 x86 来说，16 位系统使用分段内存模型，实模式下的 32 位系统也是如此。保护模式下实际使用的同样是分段模型，但看起来却跟平面模型一样。x86_64 使用平面内存模型。

范例 10-1 演示了怎样在 x86 实模式下做分段寻址。假设这是个 16 位的环境，数据段从 `1234h` 开始，我们要找的数据与该段开头的距离是 `5678h`。这份数据所在的地址需要按照**段选择器**（segment selector）与偏移量的形式来书写，也就是要写成 `1234h:5678h`。然而这并不是物理地址。

**范例 10-1　在实模式下寻址**

| 逻辑地址 | 1234h:5678h | | | |
|---|---|---|---|---|
| 第 1 步 | 1234h | * 16 | = 12340h | |
| 第 2 步 | 12340h | + 5678h | = **179B8h** | ← 物理地址 |

第 1 章讲过怎样把十六进制数转换成二进制。`1234h` 写成二进制形式是 `0001 0010 0011 0100b`，表 10-1 中又说，实模式下的地址空间是 $2^{20}$，这意味着物理地址应该用 20 个二进制位表示。段选择器只有 16 个二进制位，剩下还有 4 位没有写出来，那 4 位默认是 0。因此，段选择器所表示的实际地址应该是 `0001 0010 0011 0100 0000b`。你也可以理解成：把段选择器的字面值乘以 16 令其变为 `12340h`。然后把这个值与 16 位偏移量相加，就可以得到物理地址。对于本例来说就是把 `12340h` 与 `5678h` 相加得到 `179B8h`。

x86 地址中的段选择器这一部分通常不会直接写成 `1234h` 的形式，而是要保存于某个 16 位的段寄存器中以充当段基址。比方说，如果 `1234h` 这个段基址保存于 *gs* 寄存器中，那么刚才的地址就可以写为 `gs:5678h`。

x86 在 32 位保护模式下的寻址方式与实模式既有相似之处又有所不同。首先，它用的还是分段内存模型，但由于引入了一些抽象机制，因此整个内存空间看上去好像是个线性的模型。其中一种抽象机制是描述符，另一种是分页。

笔者在早前的侧边栏中说过，当前很多 x86 处理器都是从实模式开始运行的，但很快就会切换到保护模式。在启动过程中系统会设定描述符，例如**中断描述符表**（Interrupt Descriptor Table，IDT）、**全局描述符表**（Global Descriptor Table，GDT）、**局部描述符表**（Local Descriptor Table，LDT）、**任务状态段**（Task State Segment，TSS）描述符，以及一个**代码段描述符**（Code Segment Descriptor）与一个**数据段描述符**（Data Segment Descriptor）。其中，GDT 与 LDT 是用来详细描述内存段的数据结构，可以指出内存段的基址、大小与访问权限。GDT 针对的是全局及共享区段，而每个运行中的程序 / 进程则有其各自的 LDT 可

以使用。TSS 用来存放任务所涉及的寄存器值、I/O 权限及栈指针，以便支持多任务处理（multitasking，尤其是暂停及切换任务）。IDT 与 TSS 将在本章稍后讲解。

> 📖 **学习指南：** 与 x86/x86_64 的启动流程及任务切换有关的详细信息，请参阅《Intel Developers Manual Volume 3A》的 9.8、9.9 及 9.10 节。设置好这些表格之后，系统通过 *cr0* 寄存器来启用保护模式。*cr0* 在 32 位模式下是个 32 位宽的寄存器，在 64 位模式下是个 64 位宽的寄存器。其中有两个二进制位与此处要讨论的保护模式有关，一个是 0 号位也就是 PE 位，它用来表示 Protected Mode Enable（保护模式是否启用），另一个是 31 号位也就是 PG 位，它用来表示 Paging（分页机制是否启用）。

下面我们还是采用范例 10-1 中的数据，也就是 `1234h:5678h` 这个逻辑地址，来演示保护模式下的分段寻址。在这种情况下，段选择器并不是直接指向某个内存区段，而是用作索引来查询 GDT 表。32 位环境下的 GDT 表，其每个条目均为 8 字节，64 位环境下是 16 字节。

**范例 10-2　保护模式下的寻址**

逻辑地址 → **1234h:5678h**

| | | |
|---|---|---|
| **第 1 步** | **1234h** 在 GDT 中查询 | = **90120h** |
| **第 2 步** | **90120h** + 05678h | = **95798h** |

物理地址 ↑

**在执行第 1 步的过程中，需要查询 GDT 表** ← TI

| | |
|---|---|
| NULL | 00h |
| 基址 | 08h |
| 基址 | ... |
| 基址 = **90120h** | **1234h** ← 索引 |

范例 10-2 是用 16 位的数值来举例的，然而大家要注意：32 位环境下的基址与偏移量其实是 32 位。而且与范例 10-1 类似，段选择器这一部分也保存在某个段寄存器中。保护模式下的段选择器的内容分成三块，如表 10-2 所示。

**表 10-2　段选择器的内容**

| 二进制位 | 15...3 | 2 | 1...0 |
|---|---|---|---|
| 内容 | 索引 | 表格指示符（TI） | 请求优先级（RPL，请求权限级别） |

其中，RPL 的值可以是 0 或 3，分别表示内核级别与用户级别（或者说应用程序级别）。TI 如果是 0 就在 GDT 中查找，若为 1 则在 LDT 中查找。索引字段就是查表时的依据。

段寄存器的具体值是由系统设定的，这要取决于系统所使用的内存模型、操作系统以及当前执行的程序。在 32 位环境下，*ds*、*ss* 及 *es* 寄存器一般都指向同一个位置，这是因为系统将这三者所指向的区段全都认定成用来保存数据的区段，于是就令它们均指向这样一个区段。*fs* 与 *gs* 是给开发者使用的，通常默认为 `0x0h`，用来表示整个地址空间的起始处，它们的取值也可以与 *ds* 相同。*cs* 是代码段的段选择器，该段用来保存当前正要执行的这些指令。

> 📖 **学习指南：** 前面的章节说过，指令指针寄存器（*ip*/*eip*）指向处理器即将获取并加以执行的下一条指令所在的地址。然而 x86 的 *ip*/*eip* 中所保存的这个地址实际上是个针对代码段而言的偏移量，因此必须把该值与 *cs* 结合起来才能够确定处理器获取指令时所依据的实际地址，也就是说要根据 *cs*:*ip*/*cs*:*eip* 来判断。其中，*cs* 是代码段的段选择器，*ip*/*eip* 是以偏移量形式所表示的指令地址。

**分页技术**给 x86 系统的地址转换过程又添加了一层抽象机制。笔者不打算详尽地讲述该机制，然而还是会给出一些基本的知识点。当前的各种操作系统都在使用分页机制，这意味着 *cr0* 的 PG 位（也就是 31 号位）处于置位状态（其值是 1）。启用分页之后，系统要通过 *cr3* 寄存器把线性虚拟地址（linear virtual address）转换成物理地址。*cr3* 中有一个字段用来保存当前任务的首个页目录所在的物理地址。图 10-1 演示了 x86 分页机制的结构安排以及 32 位虚拟地址的解读方式。

32 位的 x86 采用多级分页机制。它的**页面**（page，页）大小一般是 4 KB，在系统中以连续内存空间的形式出现。如果开启了 PSE（Page Size Extension，页面大小扩展）也就是 *cr4* 寄存器的 4 号位，那么系统就支持 2MB 及 4MB 的页面。**页表**（page table）是含有 1024 个元素的数组，其中每个元素都占据 32 个二进制位，这意味着整个页表的大小是 4 KB，刚好可以放入内存页面中。页表里的每个元素都指向某个页面所在的物理地址。在页面与页表之上还有一种更大的结构叫作**页目录**（page directory），是个包含 1024 个元素的数组，而且其中的每个元素同样占据 32 个二进制位，这些元素均指向某个页表所在的物理地址。

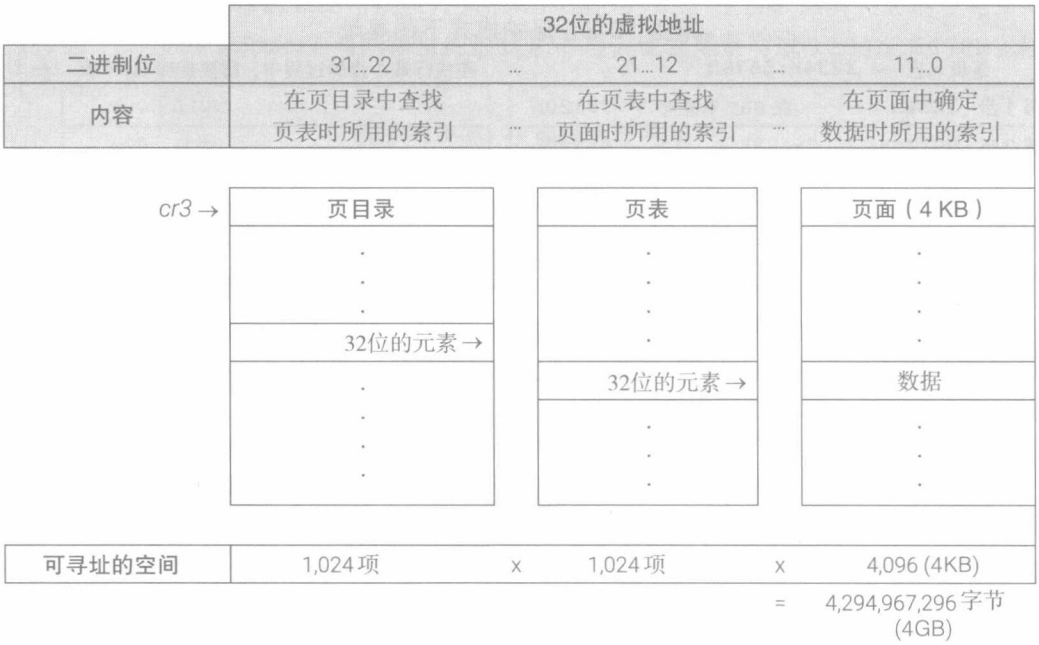



图 10-1　x86 的分页机制

系统获取到某个虚拟地址之后，要按照下列方式来确定数据的物理位置。首先，根据 *cr3* 找到当前页目录的基址。然后把虚拟地址的高 10 位（也就是 22 至 31 号位）当成索引，在页目录中查找与该索引相对应的元素，并根据该元素的取值找到对应的页表所在的基址。接下来，把虚拟地址的中间 10 位（也就是 12 至 21 号位）当成索引，在页表中查找与该索引相对应的元素，并根据该元素的取值找到对应的内存页面所在的基址。最后，把虚拟地址的低 12 位（也就是 0 至 11 号位）当成索引，在页面中查找与该索引相对应的元素，这个元素正是虚拟地址所指代的那份数据。

这种 32 位的虚拟地址使得 4GB（也就是 $2^{32}$ 个字节）的内存空间可以当成一个平面的模型来用，从而令程序数据能够以页面文件的形式出现在该空间中。系统会按照相应的方式解

读虚拟地址，并结合 *cr3* 寄存器的内容确定其物理地址。页面文件保存在辅助存储器中，并按需移动到主存及缓存中。如果要找的数据不在主存或缓存里，那么就称为**页缺失**（page fault），此时系统会从辅助存储器中获取相应的页面。启用了分页机制之后，分段就显得不那么重要了，因为每个程序都有自己的空间可以使用，而且能够以虚拟地址的形式引用该空间中的每个位置——这是一种很巧妙、很强大的设计方案。

> **提示：**分页对于程序和用户来说是透明的。在启用了分页的平面模型中，虚拟地址当成线性地址空间中的线性地址来看待。这样的线性空间会分成不同的页面，这些页面会与物理内存中的**帧**（frame）对应起来（这些帧是用来保存页面的）。

64 位的 x86_64 架构在寻址的时候几乎完全抛开了分段机制。长模式下是不进行分段的，而那些段寄存器在这种模式下的用法也各有不同。*cs*、*ds*、*ss* 与 *es* 寄存器，要么强行设为 0x0h，要么当成零来对待（也就是说，无论取什么值，一律视为 0），同时可寻址的空间也由 $2^{32}$ 上升到 $2^{64}$，这使得代码、数据与栈都可以出现在一个平面的地址空间中。尽管调试器可能根本就不显示段寄存器的值，但 *fs* 与 *gs* 或许仍有一些特殊的用途。在 Microsoft Windows 系统中，*gs* 用来指向由运行的线程所构成的线程块，Linux 内核用 *gs* 存放 CPU 数据。对于运行在兼容模式下的任务来说，依然会用到段寄存器。

x86_64 的分页机制在原有的三层结构上又加了一层。此时的 *cr3* 不直接指向页目录，而是指向**页目录指针表**（page directory pointer table），系统会通过该表中的元素（或者说指针）来确定对应的页目录所在的位置。查询页表与查询页目录时所用的索引都从 10 位降成了 9 位，多出来的这两位正是用来查询指针表的。

### 10.2.4 代码范例

x86 汇编指令通常会自动指定与该指令相对应的段寄存器，而无须开发者手工去指定。比方说，涉及栈的指令 PUSH 与 POP 会自动把栈段寄存器 *ss* 中的值当成段选择器来用，而访问数据的 MOV 指令则会自动把数据段寄存器 *ds* 中的值当成段选择器来用。

程序 10-1 是一段 x86 汇编代码，它用各种写法演示了如何通过段选择器来指定内存地址。这些段选择器通常是不用明确写出来的，然而程序 10-1 就是为了演示怎样明确地使用它们。与之相似的 NASM 及 MASM 版本，参见第 10 章的补充材料。

**程序 10-1 段选择器的用法**

GAS（32 位）

```
.data
array: .long 3, 2, 6, 4, 1

.text
.globl _main
_main:

movl $array, %eax          #1 array address to eax
movl 4(%eax), %ebx         #2 second element to ebx
movl %ds:array, %ecx       #3 first element to ecx
movl %ds:array+8, %edx     #4 third element to edx

movl %gs, %esi             #5 gs to esi
movl $42, %gs:(%esi)       #6 42 to gs:(offset in esi)
movl %gs:(%esi), %edi      #7 gs:offset to edi
```

GAS（32 位）

```
movl $43, %gs:0x04h        #8 43 to gs:offset
movl %gs:0x04h, %edi       #9 gs:offset to edi

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

第 1 行代码把数组中首个元素的地址移动到 *eax* 寄存器。第 2 行代码演示了一种典型的用法，也就是把 *eax* 中的地址与偏移量相加，并据此来执行 MOV 指令。由于 `array` 位于数据段中，因此可以给该标识符的前面写上代表数据段选择器的段寄存器，也就是 *ds*，第 3 及第 4 行代码演示了这种用法。

**编程知识：** 编写 MASM 版的汇编代码时，一般是无须指出段选择器的，然而有的情况下却必须这么做，例如在根据线性地址来移动数据时，要写成 `mov eax, ds:[0Fh]` 才行。附录 A 对比了 MASM 的 MOV 指令在写法上与 GAS 及 NASM 的 MOV 指令之间的区别。

第 5 至第 9 行演示了 *gs* 寄存器在 macOS 中的用法，这些代码纯粹是为了举例而写的。在 Linux 与 Windows 系统中不建议这样使用，因为 *fs* 与 *gs* 寄存器可能另有用途。第 5 行代码把 *gs* 中的默认值移动到 *esi* 中。尽管此时的 *gs* 与 *esi* 是相同的，但我们还是可以把前者视为段选择器，把后者视为偏移量，从而将立即值 42 移动到由这两个寄存器所确定的地址中，这就是第 6 行代码的含义。然后我们在第 7 行代码中把这个地址上的当前值（也就是 42）移动到 *edi* 寄存器中。最后，用第 8 及第 9 行代码把刚才那 3 行代码所实现的功能又执行了一遍，只不过这次是直接写出偏移量 `0x04h`，而不是用 *esi* 做偏移量。再强调一遍：这段代码仅仅用于演示段寄存器的写法。

**提示：** 本章接下来的内容会统一对待 macOS 与 BSD。尽管 macOS 的内核 XNU 是由 Mach 与 FreeBSD 混合而成的，但针对 macOS 所写的代码范例一般来说同样适用于 BSD 系统。

## 10.3　中断与系统调用

### 10.3.1　软件中断

**中断**（interrupt）是针对处理器而发的信号，用来表示某种必须立刻予以关注的情况。它可以分为硬件中断与软件中断两大类。这里主要谈**软件中断**（software interrupt，软中断）。软件中断的一种形式是应用程序主动给处理器发信号，可以通过很多办法来实现，例如可以用 INT 指令明确地产生软件中断。还有一种形式叫作**异常**（excpetion）或**陷阱**（trap），这表示处理器在执行指令的过程中遇到了某种状况，而当前正在运行的应用程序却没有处理或不能处理该状况。

📖 **学习指南：** 如果要编写系统软件（例如操作系统、驱动程序、系统工具程序等），一定要掌握与中断有关的知识，即便只开发一般的应用程序，这些知识也依然可以帮你更好地了解系统调用的演变历程。在旧式的 16 位 /32 位 x86 系统中要想命令操作系统执行一些与硬件或软件有关的操作，开发者所遵循的主要途径就是中断。新式的 32 位 /64 位 x86 系统已经不再用中断来执行这些系统任务了，而是提供了更为高效的 x86/x86_64 指令。本章稍后会继续谈论这个话题。

与 0 相除是一种很经典的软件异常。除数为 0 的情况可以用很多办法来处理，比方说下面这段 C++ 代码就演示了其中一种处理办法。

**范例 10-3　用不抛出异常的办法来应对除数为 0 的情况**

```
int divide(int numerator, int denominator){
   if (denominator == 0) {
      cout << ("ERROR: Cannot divide by zero.\n");
      return 1;
   }
   else
      return numerator / denominator;
}
```

范例 10-3 用较为简单的办法处理了一种有可能出现的异常状况，但是这种处理方式意义不大，因为它虽然通过 `if` 语句判断出了这种情况，但却只是把一行文本输出到了控制台，而没有把该情况反映给系统。范例 10-4 所采用的办法更好一些，它通过 `throw` 抛出异常，并用 C++ 的 try-catch 块捕获异常。尽管还是会给控制台发送文本，但这种通过**异常处理程序**（exception handler，也叫作异常处理器，就是 catch 结构中的那一部分代码）来应对的方式却对系统有着一定的意义。

**范例 10-4　用异常来应对除数为 0 的情况**

```
int divide(int numerator, int denominator){
   if (denominator == 0) {
      throw string("ERROR: Cannot divide by zero.\n");
   }
   else
      return numerator / denominator;
}
in main…
   try {quotient = divide(num1, num2);
        cout << "The quotient is: " << quotient << endl;
   }
   catch (string exceptionString) {
      cout << exceptionString;
   }
```

如果开发者并未使用软件形式的异常处理程序来应对除数为 0 的情况，而是令其直接出现在处理器层面，那会怎么样呢？此时，处理器就要试着执行这条除数为 0 的指令了，执行的时候会产生中断，然而由于这是个标准的处理器异常，因此它会自己来处理该异常。

**表 10-3　x86 中断**

| INT_NUM | 描　述 | INT_NUM | 描　述 |
|---|---|---|---|
| 0x00 | 除以 0 | 0x0B | 段不存在 |
| 0x01 | 调试异常 | 0x0C | 栈故障 / 栈错误 |

（续）

| INT_NUM | 描　述 | INT_NUM | 描　述 |
|---|---|---|---|
| 0x02 | 不可屏蔽的中断 | 0x0D | 一般的 / 通用的保护错误 |
| 0x03 | 调试断点 | 0x0E | 缺失故障 / 页面错误 |
| 0x04 | 溢出 | 0x0F | （保留） |
| 0x05 | 越界 / 超出范围 | 0x10 | x87 浮点错误 |
| 0x06 | 无效的操作码 | 0x11 | 对齐检查 |
| 0x07 | 协处理器（FPU）/ 设备不可用 | 0x12 | 机器检查 |
| 0x08 | 双重故障 / 双重错误 | 0x13 | SIMD 异常 |
| 0x09 | 协处理器段超出 | 0x14 | 虚拟化异常 |
| 0x0A | 无效的任务状态段 | 0x15–0xFF | 系统 / 用户定义的中断 |

**提示：**异常究竟是怎样得到处理的，可能要看计算工作是由 CPU 来做还是由 FPU 来做。数据如果像范例 10-3 一样是整数类型，那么 CPU 就会抛出中断号（INT_NUM）为 0x0h 的算术除法异常，而程序也将终止或崩溃。反之若是浮点数则有可能由 FPU 来处理，它会做得更平滑一些，也就是返回 INF（以表示无穷）并使得程序能够继续往下执行。

上一节提到过中断描述符表（Interrupt Descriptor Table，IDT）。IDT 在实模式下也叫作 IVT（Interrupt Vector Table，中断向量表），不过我们此处主要关注的是怎样在 32 位及 64 位模式下处理中断。按照传统的做法，IVT 通常是物理内存中的首个结构，它在 16 位环境下位于 0x0000h 至 0x03FFh 这一范围，在 32 位环境下位于 0x0000h 至 0x07FFh 这一范围，在 64 位环境下位于 0x0000h 至 0x0FFFh 这一范围。32 位及 64 位环境下的 IDT 其实可以放置在任何地方，而不一定非要出现在开头，*idtr* 寄存器用来保存 IDT 的物理基址及其所占据的字节数，其内容如表 10-4 所示。

表 10-4　IDTR 的内容

| 二进制位 | 47（79）　...　16 | 15　...　0 |
|---|---|---|
| 内容 | 32 位（或 64 位）的线性基址 | 16 位的值，表示 IDT（中断向量表）所占据的字节数 |

32 位模式下的 IDT 是个由描述符所构成的数组，其中的描述符都占据 8 个字节，有些描述符叫作中断门（interrupt gate），有些叫作陷阱门（trap gate），有些叫作任务门（task gate），它们全都可以统称为中断门。64 位模式（长模式及兼容模式）下的 IDT 描述符由 16 个字节组成，它在 8 字节的基础上添加并改动了一些内容。之所以变长，主要是为了保持对齐以便进行 rip 相对寻址。

表 10-5 展示了 32 位保护模式下的中断描述符，它是由 8 个字节所组成的。其中的 DPL（Descriptor Privilege Level，描述符优先级）与段描述符中的 RPL 类似，0 表示内核级别，3 表示应用程序级别。D 位（也就是第 6 个字节中的 11 号位）用来表示中断门的大小，1 代表 32 位，0 代表 16 位。P 位（也就是第 6 个字节中的 15 号位）用来表示该段是否存在于内存中。Target Segment Selector（目标段选择器，也就是第 3 至第 4 个字节）是个段选择器，用来选取 ISR（Interrupt Service Routine，中断服务例程）的目标代码所在的区段。Offset in Target Segment（目标段中的偏移量，也就是第 1 至 2 字节以及第 7 至 8 字节）是个偏移量，用来

表示 ISR 的入口点与目标段的起点之间的距离。

表 10-5 32 位环境下的 IDT 描述符所具备的内容

| 二进制位 | 31 ... 16 | | 15 | 14 13 | 12 ... 8 | 7...5 | 4 ... 0 |
|---|---|---|---|---|---|---|---|
| 内容 | Offset intarget tegment | | P | DPL | 0D110 | 000 | 保留 |
| 字节 | 8 | 7 | 6 | | | 5 | |

| 二进制位 | 31 ... 16 | | 15 ... 0 | |
|---|---|---|---|---|
| 内容 | Target segment selector | | Offest in target segment | |
| 字节 | 4 | 3 | 2 | 1 |

处理中断是个较为复杂的过程。如果系统在 32 位模式下产生中断，那么将会遵循下列流程来处理。

1. CPU 把中断号（例如 `int 3` 指令中的 3）乘以 8，因为每个中断描述符占据 8 字节（例如，要处理 3 号中断就将 3 与 8 相乘），然后将乘积加到 IDT 的基址上（该基址存放于 *idtr* 寄存器中）。

2. 把上一步所得到的乘积同 IDT 的尺寸相比较（该尺寸也存放在 *idtr* 中）以验证中断号是否有效。如果无效就产生 General Protection exception（GP 异常，也就是表 10-3 中的 General protection fault）。

3. 若有效，则将第 1 步所计算出的总和当成地址，以获取该地址上保存的 8 字节描述符。如表 10-5 所示，这个描述符中包含与中断服务例程（Interrupt Service Routine，ISR，也可以说 Interrupt handler，异常处理程序）有关的全部信息。

**32 位环境下的第 1 种情况**

如果中断处理程序需要以**序号较低**（或者说**权限较高**）的优先级来运行（比方说当前的应用程序运行在 PL3，而中断需要运行在 PL0），那么执行完上述 3 步之后，还要执行下列 4 个步骤：

4. 从 GDT 中查到相应的 TSS，以便将 ISR 所要采用的栈段选择器（stack segment selector，*ss*）与栈指针（stack point，*esp*）获取出来。

5. 处理器把即将打断的这个例程所具备的 *ss* 与 *esp* 推入新的活动栈（active stack，指的是 ISR 的栈）。

6. 处理器把当前的 *eflags*、*cs* 及 *eip* 推入活动栈。

7. 某些异常（例如 8 号、10 至 14 号、17 至 18 号异常）包含错误码，因此，在推入 *eip* 之后还要把错误码也推入活动栈中。

**32 位环境下的第 2 种情况**

如果要执行的中断处理程序与当前程序位于同一级别（比方说，当前的应用程序与要执行的中断都运行在 PL3），那么执行完早前说的那 3 步之后还要执行下列两个步骤：

4. 处理器把当前的 *eflags*、*cs* 及 *eip* 推入活动栈。

5. 某些异常（例如 8 号、10 至 14 号、17 至 18 号异常）包含错误码，因此，在推入 *eip* 之后，还要把错误码也推入活动栈中。

然后开始执行中断例程。执行完毕后用 IRET 指令返回。该指令是特殊的 RET 指令，它会把 *eip*、*cs* 及 *eflags* 等寄存器的旧值恢复过来（相当于把早前压入栈中的值弹出到相应的寄存器），使得原来中断了的任务，可以从中断的地方开始继续往下执行。IRET 所弹出的

数据的数量由双方任务的优先级来决定。




图 10-2　在 32 位环境下处理中断时，栈的变化过程

图 10-2 演示了在 32 位的 x86 环境下处理中断时栈的变化过程。在第 1 种情况下，发生中断的进程与中断例程具有不同的栈。在第 2 种情况下，发生中断的进程与中断例程本身合用同一个栈。

64 位模式下的软件中断处理起来与 32 位相似，只是有些地方不同。首先，中断例程是用 64 位模式下的代码书写的。其次，IDT 中的每个条目不再是 8 个字节，而是变为 16 个字节。这种中断描述符与 32 位环境下的描述符相比，最显著的区别在于多出来了一个表示**中断栈表**（Interrupt Stack Table，IST）的字段，该字段由 3 个二进制位组成，我们将在下面的第 4 种情况中讲解。

> **提示：**兼容模式实际上是增强版的 32 位模式，但它的中断门及例程需要以 64 位的方式来定义，因为该模式严格来说是 64 位长模式的一种子模式。

如果中断发生在 32 位环境下，那么推入栈中的数据量要看发生中断的程序与中断例程所在的优先级是否相同。图 10-2 演示了这两种情况。与 32 位环境不同，如果中断发生在 64 位环境下，那么 *ss*:*rsp* 的值总是要推入栈中，而且即便该中断没有使用错误码，栈中也还会给错误码预留一个位置。与在 64 位环境下调用普通例程时相似，调用中断的时候也必须按 16 字节对齐，以便更为有效地使用 128 位的 *xmm* 寄存器。

64 位的 IRET 指令的行为与 32 位的指令基本相同，只不过弹出的是 64 位值。此外还有一个区别：由于发生 64 位中断时 *ss*:*rsp* 的值总是要入栈，因此，纯 64 位模式下的 IRET 指令也总是要把早前压入栈中的值弹回到 *ss*:*rsp*，与之相对，在兼容模式下只有当双方优先级不相同的时候，它才会把早前压入栈中的值弹回到 *ss*:*rsp* 的寄存器，以保持向后兼容。

64 位处理器要分 4 种情况来处理中断。

**64 位环境下的第 1 种情况**

如果处理器按照传统的 32 位模式运行，那么栈的切换过程就和 32 位环境一样。

**64 位环境下的第 2 种情况**

在 64 位模式的兼容模式下，如果双方任务的优先级不同，那么栈的切换方式与 32 位环境下的第 1 种情况相比是稍有区别的，这尤其体现在第 4 步中。此时，系统会用 TSS 中的

值来更新 *rsp* 并把 *ss* 设为 NULL，同时将其 RPL（请求优先级）字段设置成中断例程所处的优先级，以支持嵌套中断。从中断返回的时候，*ss* 与 *rsp* 会予以恢复。

**64 位环境下的第 3 种情况**

在纯 64 位模式下，如果用来查询 IST 的那 3 位索引是 0（也就是 000b），就按照刚才说的第 2 种情况处理。

**64 位环境下的第 4 种情况**

在纯 64 位模式下，如果用来查询 IST 的那 3 位索引不是 0（也就是位于 001b 至 111b 之间），就启用 IST 切换机制。由于某些中断必须在已经确认正常的栈[⊖]中执行，因此，需要通过某种办法切换到这样的栈。在 32 位模式下，这是通过 IDT 里的任务门实现的，而在 64 位模式下则要通过 IST 来实现。

> 📖 **学习指南：** 任务门也是一种中断形式，系统会像正常切换任务时那样从 TSS 中找到与该中断相对应的条目，并由此进入任务门。中断处理程序的 TSS 中有一个字段叫作 Previous TSS link（指向前一个 TSS 的链接），它指向早前发生中断的任务。在处理任务门的过程中其他中断是遭到禁用的。Non-Maskable Interrupt（NMI，不可屏蔽的中断）、Double Fault（双重故障）及 Machine Check（机器检查）等，都属于任务门。

IST 是 TSS 中的一块区域，专门用来在 64 位环境下切换栈。64 位环境下的 IDT 中有一个 3 位的字段，用以指向 IST 中的某个条目，由于该字段的取值范围是 001b 至 111b（也就是十进制的 1 至 7），因此 IST 中可以有 7 个条目与之相应。每个条目都可以视为一个指针，用以指向某个有效的栈，这使得系统能够把该指针所指的位置加载到 *rsp* 中，从而用新的栈去处理中断。图 10-3 简单地演示了 64 位环境下的 IST 机制。

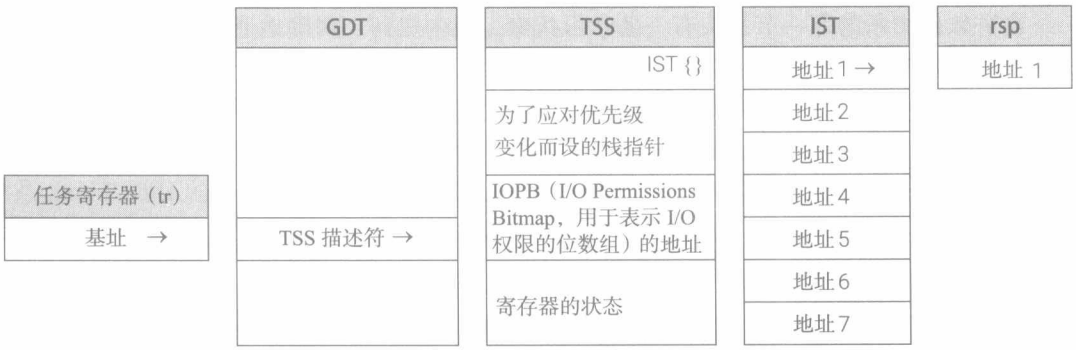



图 10-3 64 位环境下的中断栈表（IST）

系统可以在 IDT 中通过 Task Register（*tr*，任务寄存器）所包含的地址值找到 TSS 描述符，并根据该描述符中的地址找到 TSS（Task State Segment，任务状态段）。TSS 中的 IST 区域中包含 7 个地址，每个地址都可以指向某个已经确认无误的栈。如果中断描述符的 IST 字段不是 0，那么系统就可以把该字段的值当作索引，在 TSS 的 IST 区域中找到相应的地址并将其加载到 *rsp* 中。

第 4 种情况除了在 IST 上有所不同之外，其余都和 **64 位环境下的第 2 种情况**一样。将 *rsp* 指向某个没有问题的栈之后，系统会把 *ss* 设为 NULL，并将 RPL 设置成与当前中断的

⊖ 原文为 known good stack，直译是已知的好栈。——译者注

优先级相同的值。发生中断的进程所具备的 *ss*、*rsp*、*rflags*、*cs* 及 *rip* 值会推入新栈，然后，系统开始处理这个中断。

### 10.3.2　硬件中断

**硬件中断**（hardware interrupt）也会给处理器发信号，但这个信号并不是由软件发出的，而是由某个与系统相通信的设备通过**中断请求线**（interrupt request line，IRQ line）所发。IRQ line 使得处理器能够与硬件设备相通信。处理器用一些号码来表示通过 IRQ line 所发出的硬件中断。这些 IRQ line 是用**可编程的中断控制器**（Programmable interrupt controller，PIC）连接到处理器上的。老式的 x86 系统采用两个 Intel 8259 做 PIC，每个 PIC 支持 8 条 IRQ line，这 16 条 IRQ line 分别叫作 IRQ0 至 IRQ15。

当前的 x86 系统采用 Intel 的 APIC 来取代老式的 PIC，它能够在多处理器的系统中更为灵活地派发中断。APIC 理论上支持多达 255 条中断线，不过，大多数系统目前只支持 24 个 IRQ。老式的 PIC 会用专门的 IRQ line 来表示特定的中断，例如用 IRQ0 表示系统时钟，用 IRQ 14 表示磁盘的主 ATA 通道。尽管在新式的 APIC 中这些都可以重新编程，然而大多数操作系统还是遵照传统的办法来使用 IRQ。

### 10.3.3　通过 INT 进行系统调用（旧方法）

从第 1 章的代码回顾中所提到的冒泡排序程序开始，一直到本章的程序 10-1，凡是独立的 32 位 GAS 与 NASM 代码，其最后一条指令均为 `int $0x80` 或 `int 80h`。此外，对于第 9 章的范例 9-1 中所含的内联式汇编代码来说，Clang/GCC 版本与 Visual C++ 版本也都在第一行里用到了 INT 指令，前者用的是 `int $3`，后者是 `int 3`。接下来，我们就要再举一些例子来讲解 INT 指令的用法。

在讲软件中断的那一节开头有一条学习指南，其中提到：明确地通过中断来执行任务是一种老式的做法，用于在 16/32 位 x86 系统中完成某些工作。网上以及其他书籍中也能找到很多旧式的范例代码，它们都通过 INT 指令来执行系统调用。**系统调用**（system call）是向**操作系统内核**（operating system kernel）所发出的请求，旨在使系统完成某件事情。有些情况下，这种请求是通过 API 发出的，例如 Windows 系统就是这样。32 位的 Unix/Linux 系统通过 `int 80` 做系统调用，Microsoft DOS 用的是 `int 21h`，后续版本的 Windows（例如 Windows 2000、Windows XP）用的是 `int 2Eh`。

> 📖 **学习指南：** 笔者打算以 macOS 及 Linux 为例来演示 INT 指令，因为该指令在这两个系统中的用法较为一致，而且也比较直观。对于 Windows 系统来说，笔者会采用 Windows API 讲解，这种办法比 INT 稳定，而且是当前通行的实用方法。

通过 `int 80h` 处理系统调用时，遵循如下流程。

1. 执行 INT 指令。
2. 处理器把运行级别从 PL3（应用程序级别）切换到 PL0（内核级别）。
3. 通过 IDT 中的相关描述符找到与 INT 80h 对应的例程。
4. 把控制权交给内核，以便处理由系统调用号（system call number）所表示的请求。
5. 处理该请求。
6. 将优先级重新设为 PL3。

7. 把控制权返还给调用 INT 指令的程序。

程序 10-2 演示了怎样在 GAS 代码里使用 INT 指令。与之类似的 NASM 版程序请参见第 10 章的补充材料。

**程序 10-2 用中断指令插入调试断点并执行 write 系统调用，以便向控制台写入文本**

GAS（32 位）

```
.data
output:    .ascii "Computer Architecture\n"
.equ len, (. - output)

.text
.globl _main
_main:            # Print "Computer Architecture"

int $3            # Set breakpoint (remove if desired)
pushl $len        # String length
pushl $output     # String address
pushl $1          # Print to STDOUT (File Descriptor 1, FD1)
pushl $0          # Extra push to align stack
movl $4, %eax     # Use SYSCALL 4 (WRITE), prints to screen
int  $0x80        # system call interrupt
addl $16, %esp    # Clean up stack

pushl $len        # BSD assumes that a function calls INT 80h
pushl $output     # as opposed to issuing INT 80h directly
pushl $1          # Print using kernel call
movl $4, %eax
call _kernel
addl $12, %esp

pushl $0          # return value
subl $4, %esp     # align stack
movl $1, %eax     # call number for exit
int $0x80         # system call interrupt

_kernel:          # For any system call
   int $0x80
   ret

.end
```

程序 10-2 刚开始执行的时候，会通过 `int $3` 指令给处理器发送信号以设定调试断点。如果程序是在 Debug 模式下运行的，那么执行到 `int $3` 这里就会暂停。谈到断点，Xcode 与 Visual Studio 等图形化的开发环境可以通过图形界面来设置，而 GDB 与 LLDB 等命令行调试器则可以通过简单的命令在任何一个标签处设置。不过，除了这两种方式之外，你还可以像本例这样通过 `INT 3` 指令以编程方式来实现。这种做法只应该用在调试版的程序中。

程序 10-2 把字符串“Computer Architecture”打印两遍，第一次是通过系统调用打印的，第二次是通过函数打印的，这个函数的功能也是执行系统调用。进行系统调用时，你必须告诉系统自己想要做什么事情，例如要向控制台中写入信息，或是从文件中读取数据。要做的这件事由**系统调用号**（system call number）表示。这个号码以及执行系统调用所必备的参数可以放在寄存器里也可以放在栈中，有的情况下还会分别放在这两个地方。具体要根据系统来定。

📖 **学习指南：**第 10 章的补充材料中给出了与系统调用表有关的信息，还提供了一些网络资源链接。

在 macOS/BSD 系统中以 32 位模式使用 INT 指令时，参数要放在栈中，调用号要放在 *eax* 寄存器里。Linux 系统要求把这两者都放在寄存器里。

- macOS/BSD：按反向顺序将各参数推入栈中，把调用号移入 *eax* 发送 INT。
- Linux：将各参数依次移入 *ebx*、*ecx*、*edx*、*esi*、*edi* 及 *ebp* 寄存器中，把调用号移入 *eax* 发送 INT。

通过范例 10-5 可以看出，系统调用与函数调用很像。其实，系统调用就是在调用例程或函数，只不过它是通过中断门来调用的。Linux 的做法接近于 32 位环境下的 fastcall 约定，而 macOS/BSD 的做法则接近于 cdecl 约定，它按照从右至左的顺序把参数推入栈中。BSD 的 `syscalls.master` 文件中写有 `write` 这个系统调用的原型，从中可以看出各参数的顺序。

```
4...{user_ssize_t write(int fd, user_addr_t cbuf, user_size_t nbyte);}
```

**范例 10-5　用 INT 指令执行 `write` 系统调用，以便向控制台写入文本**

| macOS/BSD – GAS | Linux – NASM |
|---|---|
| `pushl $len      # length`<br>`pushl $output   # address`<br>`pushl $1        # FD`<br>`pushl $0        # align`<br>`movl $4, %eax   # call number`<br>`int  $0x80` | `mov ebx, 1        ; FD`<br>`mov ecx, output   ; address`<br>`mov edx, len      ; length`<br>`mov eax, 4        ; call number`<br>`int 80h` |

参照程序 10-2 及范例 10-5，我们可以知道执行这次系统调用所遵循的步骤。首先，把字符串长度放入栈中，然后把字符串的位置放入栈中，接下来把代表 STDOUT（也就是标准输出端）的文件描述符 1 推入栈中，最后，把一个零值推入栈中以便将栈对齐，这样做是为了遵守 macOS 系统对 32 位程序所提出的要求。把参数准备好之后，将代表 `write`（写入）操作的调用号码 4 放入 *eax* 寄存器，并通过 `int  $0x80` 指令执行系统调用。等系统将字符串打印到控制台之后，我们按照早前推入的参数个数来调整栈指针，以便将栈清理干净。由于早前推入了 4 个参数，而每个参数占据 4 个字节，因此需要给 *esp* 加上 16（4 × 4 = 16）。

注意观察 `write` 这个系统调用的原型，它的返回值是 `user_ssize_t` 类型。大多数系统调用都会返回某个值，该值通常存放在 *eax*/*rax* 中。对于程序 10-2 来说，如果你在执行完中断指令之后查看 *eax* 的状态，那么就会发现它的值是 `0x16h`（也就是 `22d`），这意味着刚才那次系统调用在控制台上打印了 22 个字符。其他一些系统调用例如获取系统时间的调用，可能会把天数、小时、分钟等信息放入不同的寄存器中。你可以在自己所用的平台中搜寻与某个系统调用有关的文档，以便详细了解它的返回值（第 10 章的补充材料中也给出了一些资源）。

程序 10-2 还演示了怎样通过名为 `_kernel` 的函数来执行系统调用，这是 macOS/BSD 系统的推荐做法，当然也可以用在其他的操作系统上。要注意的是，如果你用含有 INT 及 RET 的函数执行系统调用，就无须再编写 PUSH 指令来实现栈对齐了。

程序 10-3 又是一个演示 INT 用法的例子。这次我们把程序进程的 ID 打印到控制台。查询进程 ID 所用的系统调用号是 `0x14`，程序将该号码载入 *eax* 寄存器然后发出中断，系统会把进程 ID 返回到 *eax* 寄存器里。然后，程序通过循环把其中的每一位数字转换成对应的 ASCII 字符，以便于打印。最后，还是像程序 10-2 一样通过 INT 指令将这个表示进程 ID 的字符串打印到控制台。

对于底层编程来说 INT 指令确实很有必要，而且具备许多优点，不过，执行系统调用

的方法总是在不断演变。尽管现在我们依然可以用 INT 实现系统调用，但必须对这样做所引发的一些开销有一定的认识。

程序 10-3 通过 INT 指令打印进程 ID

GAS（32 位）

```
.data
   decimal: .long 10
.bss
.lcomm PID, 4     # Reserve 4 bytes for a process id

.text
.globl _main
_main:                          # Print Process ID

movl $0x14, %eax                # GET_PID call
int $0x80                       # syscall
                                # process ID is returned in EAX
movl $(PID+4), %edi             # Pointer to end of PID string
convert:
    xorl    %edx, %edx
    divl    decimal             # Divide by 10
    addl    $48, %edx           # Add 48 to remainder
    movb    %dl, (%edi)         # Store
    decl    %edi                # Next digit
    cmpl    $0, %eax            # Is EAX = 0?
    jg      convert             # If greater than 0, more digits

pushl $6                        # string length
pushl $PID                      # string address
pushl $1                        # file descriptor STDOUT (1)
pushl $0                        # align stack
movl $4, %eax                   # system call 4 (write)
int $0x80                       # system call
addl $16, %esp                  # clean stack

pushl $0                        # return value
subl $4, %esp                   # align stack
movl $1, %eax                   # system call 1 (exit)
int $0x80                       # system call
.end
```

### 10.3.4 通过 SYSENTER、SYSCALL 及程序库 /API 进行系统调用（新方法）

从 Pentium II（奔腾 II）处理器开始 Intel 引入了执行**快速系统调用**（fast system call）的指令。其中，最先登场的是 SYSENTER 以及与之配套的 SYSEXIT 指令，这两者可以用在 32 位及 64 位模式下。然后又出现了 64 位长模式下的 SYSCALL 以及与它相配套的 SYSRET 指令，这两者可以在 64 位模式下使用。这一批指令之所以叫作快速调用指令，是因为它们想要更为高效地切换优先级。

**编程知识：SYSENTER 与 SYSEXIT 指令以及 SYSCALL 与 SYSRET 指令，并不像 CALL 与 RET 指令那样是调用与返回的关系，而是用来在 PL0 与 PL3 之间切换。**

与其他较为高深的体系结构问题一样，这个话题也需要深入研究，你可以参考第 10 章的补充材料。切换 PL 的工作由操作系统负责，这意味着在某些情况下快速调用要比通过 INT 进行调用更为迅速，而在另外一些情况下则更为缓慢。究竟是快还是慢要看操作系统的

是怎么实现的，此外，处理器流水线也会影响调用的速度。

网上以及其他书籍中经常出现许多与系统调用有关的范例（这些范例通常都比较旧），而且还会告诉你哪种调用方式更好。在阅读这些内容的时候你应该注意：系统调用的处理方式经常在变化，就连同一个操作系统的不同版本之间可能都会有所区别，甚至在操作系统与版本均相同的情况下，也还是有可能因为各种各样的细节而产生差异。

比方说，尽管早前版本的 Linux 内核（例如 2.6 版、3.x.y 版）在处理 INT、SYSENTER 及 SYSCALL 时所用的办法有很大区别，但在较近的版本（例如笔者编写本书时的 4.x.y 版）中，这些办法却合并了起来，也就是说新版内核在处理这三者时所用的实现方式实际上是一致的。无论是通过 INT、SYSENTER，还是通过 SYSCALL 来做系统调用，内核都会按照同一条路径来处理。

Windows 系统也是如此，只不过它用的是另一套实现方式。运行于用户空间中的程序很少会直接使用 INT 指令做系统调用，因为开发者在编写汇编代码时可以方便地通过 Windows API 执行调用。Windows API 构建在（由 `ntdll.dll` 所提供的）Windows Native API 之上，这些 Native API 是一种用 C 语言及汇编语言所写的底层例程，它们通过 INT、SYSENTER 及 SYSCALL 执行系统调用。虽然 Windows、Linux 以及 macOS/BSD 的实现机制各有区别，但有一点是相通的：开发者一般都采用程序库或 API 中的 wrapper function（包裹函数[⊖]）来做系统调用，从这些高层函数到达底层指令需要经过很多层抽象机制。不仅在用户代码与处理器之间需要抽象，用户代码与内核之间也同样需要抽象。

本书的 MASM 范例程序（适用于 Windows 系统）都调用下面这个 Windows API 来实现退出，这与 GAS 及 MASM 版本中相应的 INT 调用在功能上是类似的。32 位的 MASM 范例代码中经常出现这种写法：

```
ExitProcess PROTO, dwExitCode:DWORD
; code
INVOKE ExitProcess, 0
```

其中的 `ExitProcess` 是 Windows API 中的函数，它其实是 `Kernel32.lib/dll` 中的一种系统调用，用于结束调用该函数的进程。刚才的第一行代码指出 `ExitProcess` 函数的原型，使得系统明白程序想要使用这个函数。最后一行代码通过 INVOKE 命令（这也算是一种 CALL（调用））执行第一行所指出的 `ExitProcess` 系统例程。

程序 10-4 是个 32 位的范例程序，用来演示怎样通过 Windows API 输出信息，它的功能与程序 10-2 及程序 10-3 相似。这段代码还体现出了 CALL 与 INVOKE 的区别，用后者来传递参数并执行调用要比用前者简单一些。范例 10-6 给出了相关方法的 C++ 原型，虽说这两组方法都是通过 `WINAPI` 这种约定方式来调用的，但是与之对应的例程却位于不同的程序库中。

**程序 10-4　通过 Windows API 执行系统调用**

MASM(32-bit)

```
.386
.MODEL FLAT, stdcall
.STACK 4096

GetCurrentProcessId PROTO
```

⊖ 也称为封装函数，下同。——译者注

MASM(32-bit)

```
ExitProcess PROTO, dwExitCode:DWORD
MessageBoxA PROTO, hwnd:DWORD, text:DWORD, caption:DWORD, buttons:DWORD

.DATA
output BYTE "Computer Architecture",0
caption BYTE "Message", 0
decimal DWORD 10
PID DWORD ?

.CODE
_main PROC                          ; Print "Computer Architecture"

   push 0                           ; MB_OK (0 = OK button only)
   push OFFSET caption              ; lpCaption
   push OFFSET output               ; lpText
   push 0                           ; hWnd handle owner window (0 = no owner)
   call MessageBoxA

   INVOKE GetCurrentProcessId
   mov PID, eax

   mov edi, OFFSET [PID+3]          ; Pointer to end of PID string
convert:
   xor    edx, edx
   div    decimal                   ; Divide by 10
   add    edx, 48                   ; Add 48 to remainder
   mov    BYTE PTR [edi], dl        ; Store
   dec    edi                       ; Next digit
   cmp    eax, 0                    ; Is EAX = 0?
   jg     convert                   ; If greater than 0, more digits

   INVOKE MessageBoxA, 0, OFFSET PID, OFFSET caption, 0

INVOKE ExitProcess, 0
_main ENDP
END
```

范例 10-6 Windows API 的 C++ 原型

| 函数 | MessageBox | ExitProcess 及 GetCurrentProcessId |
|---|---|---|
| 库 | User32.lib/dll | Kernel32.lib/dll |
| 原型 | int WINAPI MessageBox(<br>_In_opt_ HWND hWnd,<br>_In_opt_ LPCTSTR lpText,<br>_In_opt_ LPCTSTR lpCaption,<br>_In_ UINT uType<br>); | VOID WINAPI ExitProcess(<br>_In_ UINT uExitCode<br>);<br>DWORD WINAPI GetCurrentProcessId(void); |

**编程指南：** INVOKE 命令可以在 32 位版的 MASM 代码中使用，但是不能用在 64 位版的 MASM 代码中。如果要针对 Microsoft x64 编程，那么必须先把参数放入适当的位置，然后通过 CALL 指令来实现系统调用。请参见第 6 章的表 6-1 及本章的程序 10-8。

将进程 ID 从数值转换为 ASCII 字符的时候，MASM（以及 NASM）版本的代码与程序 10-3 中的 GAS 代码相比有个微小的区别：它只把 *edi* 偏移了 3 个字节，而 GAS 版本则偏移

了4个字节。输出字符串时所传递的值以及输出的方式在各个版本的代码中可能会有所区别，这是因为进程ID的长度要依照具体的操作系统来定。

> **提示：**尽量通过程序库或API来执行系统调用而不要直接使用汇编指令。这样做有两大优势：代码写起来更为一致，而且便于移植。

从底层的角度看，执行系统调用时都会遵循下面这套流程。

1. 保存当前的上下文。
2. 把优先级变为PL0。
3. 载入并执行系统例程。
4. 把优先级重设为PL3。
5. 返回到某个有效的位置（通常指的是执行系统调用的过程停下来的地方）。

那么，有没有不用切换PL、不用保存上下文就能完成系统调用的办法呢？确实有，而且我们也经常这么做，只不过这种办法对开发者来说是透明的，因此很多人意识不到罢了。Linux系统的内核可以创建名为VDSO（Virtual Dynamic Shared Object，虚拟动态共享对象）的虚拟内存页，并将其与内存中的所有用户进程相链接。VDSO中含有常用的系统调用例程。如果用户空间里的应用程序做出系统调用，那么实际上会触发相当抽象的`__kernel_vsyscall`机制，无论应用程序是按照常见的方式通过libc、glibc及unistd等程序库来调用，还是直接通过汇编指令调用，都是如此。这个解决办法很巧妙：由于VDSO是由内核提供到用户空间中的，因此它可以像调用普通函数那样执行系统例程而无须切换上下文或优先级。不过要注意并非所有的系统例程都在VDSO中。

> **注意：**各平台以及同一个平台的不同版本可能会用不同的方式来提升系统调用的速度。旧式的Linux内核具备`vsyscall`机制，但是它比VDSO的限制更多。macOS原来用的是`commpage`，但是现在已经改用其他机制来处理系统例程了（例如程序库以及x86_64 ABI）。Windows系统用的是动态链接库（Dynamic-Link Library，DLL）。总之，系统例程会以很多种方式加以抽象，而且抽象方式也在逐渐变化。

本章开头提到过MSR，也就是特定于机器的寄存器，操作系统在内核初始化进程中会用到这些寄存器。它们是为了便于执行快速系统调用而设的，并且有专门的读取与写入指令，也就是RDMSR及WRMSR。开发者不应该随意操作MSR。

下面几个寄存器都属于MSR寄存器：

- 32位
  - IA32_SYSENTER_CS：存放段选择器，以便选取PL0（内核）的代码段
  - IA32_SYSENTER_EIP：存放想要执行的系统例程的首条指令所在的地址
  - IA32_SYSENTER_ESP：存放PL0栈的地址
- 64位
  - IA32_STAR：存放PL0代码（调用目标）与PL3代码（调用方）的*cs*及*ss*
  - IA32_LSTAR：存放想要执行的系统例程的首条指令所在的地址
  - IA32_FMASK：存放*rflags*掩码值

从上面所列的这些MSR可以看出MSR中保存着切换例程的上下文时所要用到的地址

与相关信息。操作系统在初始化的过程中会查询 MSR，以便把执行系统调用所依循的路径制定好，等操作系统初始化完毕并正常运作时，就可以按照早前制定好的路径来处理相关的系统调用了。

了解到系统调用的不同方式之后，我们现在就来看看下面这几段范例程序。用 SYSENTER 来执行调用的时候，传递参数的方式与 INT 类似，但是用 SYSCALL 执行时采用的方式则有所不同。

**程序 10-5**：NASM 版本的 32 位程序，采用 SYSENTER 执行系统调用。该程序适用于 Ubuntu Linux 系统。

该程序通过 32 位的 SYSENTER 指令执行系统调用：把参数依序移入 *ebx*、*ecx*、*edx*、*esi*、*edi* 及 *ebp* 中；把系统调用号移入 *eax* 中；执行 SYSENTER；执行完毕后，返回值位于 *eax* 中。

**程序 10-6**：GAS 及 NASM 版本的 64 位程序，采用 SYSCALL 执行系统调用。这两个程序分别适用于 macOS/BSD 及 Ubuntu Linux 系统。

该程序通过 64 位的 SYSCALL 指令（适用于 AMD64）执行系统调用：把参数依序移入 *rdi*、*rsi*、*rdx*、*r10*、*r8* 及 *r9* 中，把调用号移入 *rax* 中；执行 SYSCALL；执行完毕后，返回值位于 *rax* 寄存器里（负值意味着发生了错误）。

**程序 10-7**：用 C++ 语言所写的 32/64 位程序。适用于 macOS/BSD 及 Linux 的版本通过 unistd 及 libc 库来执行系统调用，适用于 Windows 的版本通过 Windows API 来执行系统调用。

**程序 10-8**：MASM 版本的 64 位程序，它是用汇编代码写成的，该程序通过 Windows API，实现了程序 10-7 的功能。

该程序在 Microsoft x64 环境下执行系统调用：把参数依序移入 *rcx*、*rdx*、*r8* 及 *r9* 寄存器，如果还有其他参数要传递，那就依照反向顺序将其推入栈中；执行 CALL 指令；返回值一般位于 *rax* 中。

**编程知识：请参阅附录 G 来检测处理器对 SYSENTER 的支持情况。**

程序 10-5 演示了怎样在 32 位的 Linux 环境中用 SYSENTER 指令来执行系统调用。该程序也会像程序 10-2 那样，在控制台上打印一条消息，只不过它用的是 SYSENTER 指令而非 INT 指令。这样做会引发另外一些开销，因为我们必须把寄存器及堆栈上下文推入栈中，以便在执行完系统调用之后能够将上下文恢复过来，为此，还必须创建相应的返回点（也就是代码中的 `sysenter_ret`）。

**程序 10-5　SYSENTER**

Linux（NASM 32 位）

```
section .data
output: DB "Computer Architecture",0Ah
len: EQU ($ - output)

section .text
global _main
_main:

mov ebx, 1
```

Linux（NASM 32 位）

```
mov ecx, output
mov edx, len
mov eax, 4

push sysenter_ret
push  ecx
push  edx
push  ebp
mov   ebp, esp
sysenter
sysenter_ret:

mov eax, 1
mov ebx, 0
int 80h
```

程序 10-6 演示了怎样在 64 位的 macOS/BSD 及 Linux 系统中通过 SYSCALL 把文本写入文件[⊖]。与 32 位环境下的程序 10-2 及程序 10-5 不同，编写 64 位程序时不使用 EQU 命令。macOS/BSD 与 Linux 版本之间有 3 个重要的区别：第一，系统调用号不同，第二，对 len 变量做 rip 相对寻址时，所用的写法不同，第三，在表示文件标志时，macOS/BSD 用的是十六进制，Linux 用的是八进制。

第 10 章的补充材料中还会再次列出程序 10-6，并且会加上注释，以便详细解释相关的系统调用号。请注意，在 XNU 内核中要把该号码所属的 system call class[⊖] 左移 24 位，并与号码本身相加。例如 Unix/BSD 类别的调用要加上 0x2000000，而 Mach 类别的调用则要加上 0x1000000，这意味着 Unix/BSD 类别下的 1 号系统调用应该写成 0x2000001，Mach 类别下的 1 号系统调用应该写成 0x1000001。

**程序 10-6　在 64 位系统中执行 SYSCALL**

SYSCALL 在 macOS/BSD 系统中（GAS 64 位）

```
.data
output: .ascii "Computer Architecture\n"
len: .quad (. - output)
file_name: .asciz "output.txt"
file_handle: .long 0

.text
.globl _main
_main:

# write to console
movq $0x2000004, %rax
movq $1, %rdi
leaq output(%rip), %rsi
movq len(%rip), %rdx
syscall
```

SYSCALL 在 Linux 系统中（NASM 64 位）

```
section .data
output: DB "Computer Architecture",0Ah
len: DQ ($ - output)
file_name: DB "output.txt",0
file_handle: DD 0

section .text
global _main
_main:

# write to console
mov rax, 1
mov rdi, 1
lea rsi, [rel output]
mov rdx, [len]
syscall
```

⊖ 为了顺利编译该程序，可能需要在执行 nasm 时使用 -f elf64 选项，或在代码开头加上 BITS 64 这一行命令。——译者注

⊖ 系统调用类别，也称为 SYSCALL_CLASS，参见：github.com/apple/darwin-xnu/blob/master/osfmk/mach/i386/syscall_sw.h。——译者注

| SYSCALL 在 macOS/BSD 系统中 (GAS 64 位) | SYSCALL 在 Linux 系统中 (NASM 64 位) |
|---|---|
| `# create and open file`<br>`movq $0x2000005, %rax`<br>`leaq file_name(%rip), %rdi`<br>`movq $0x0201, %rsi`<br>`movq $0666, %rdx`<br>`syscall`<br>`movq %rax, file_handle(%rip)`<br><br>`# write to file`<br>`movq $0x2000004, %rax`<br>`movq file_handle(%rip), %rdi`<br>`leaq output(%rip), %rsi`<br>`movq len(%rip), %rdx`<br>`syscall`<br><br>`# close file`<br>`movq $0x2000006, %rax`<br>`movq file_handle(%rip), %rdi`<br>`syscall`<br><br>`# exit`<br>`movq $0x2000001, %rax`<br>`movq $0, %rdi`<br>`syscall`<br>`.end` | `# create and open file`<br>`mov rax, 2`<br>`lea rdi, [rel file_name]`<br>`mov rsi, 0101o`<br>`mov rdx, 0666o`<br>`syscall`<br>`mov [rel file_handle], rax`<br><br>`# write to file`<br>`mov rax, 1`<br>`mov rdi, [rel file_handle]`<br>`lea rsi, [rel output]`<br>`mov rdx, [len]`<br>`syscall`<br><br>`# close file`<br>`mov rax, 3`<br>`mov rdi, [rel file_handle]`<br>`syscall`<br><br>`# exit`<br>`mov rax, 60`<br>`xor rdi, rdi`<br>`syscall` |

程序 10-7 演示了怎样在 C++ 代码中通过程序库与 API 来执行系统调用。Unix 版本借助 unistd.h 头文件及 libc 库发出系统调用，它所用到的 `open()`、`read()`、`write()` 及 `close()` 都是 wrapper function（包裹函数），在执行这些函数时，内核会根据其中所封装的逻辑做出适当的系统调用。此程序假设当前已经有一个名叫 `input.txt` 的文本文件了，它会根据 `num` 常量的取值（本例为 13）从该文件内读取若干字符，并将其写入控制台。如果文件的内容是“Computer Architecture”，那么打印到控制台上的就是前 13 个字符，即“Computer Arch”。Windows 版的程序 10-7 也是用 C++ 语言写的，它通过对应的 Windows API 完成系统调用。

**程序 10-7　通过程序库与 API 执行系统调用**

| mac0s/BSD 及 Linux (C++ 32/64 位) | Windows API (C++32/64 位) |
|---|---|
| `#include <unistd.h>`<br>`#include <fcntl.h>`<br><br>`int main()`<br>`{`<br>`  const int num = 13;`<br>`  char inputFile[] = "input.txt";`<br>`  char buffer[num + 1] = {NULL};`<br><br>`  //open file`<br>`  int FD = open(inputFile, O_RDONLY);`<br><br>`  //read file`<br>`  read(FD, buffer, num);`<br><br>`  //print to STDOUT`<br>`  write(STDOUT_FILENO, buffer, num);`<br><br>`  //close file` | `#include <Windows.h>`<br>`using namespace std;`<br><br>`int main()`<br>`{`<br>`  const int num = 13;`<br>`  char inputFile[] = "input.txt";`<br>`  char caption[] = "Message";`<br>`  char buffer[num + 1] = {NULL};`<br>`  DWORD charsRead = 0;`<br><br>`  //open file`<br>`  HANDLE FD = CreateFile(inputFile,`<br>`  GENERIC_READ,`<br>`  0,`<br>`  NULL,`<br>`  OPEN_EXISTING,`<br>`  FILE_ATTRIBUTE_NORMAL,`<br>`  NULL);` |

| mac0s/BSD 及 Linux (C++ 32/64 位 ) | Windows API (C++32/64 位 ) |
| --- | --- |
| close(FD);<br><br>return 0;<br>} | //read file<br>bool read = ReadFile(FD,<br>buffer,<br>num,<br>&charsRead,<br>NULL);<br><br>//print to Message Box<br>int button = MessageBox(0,<br>buffer,<br>caption,<br>MB_OK);<br><br>//close file<br>bool closed = CloseHandle(FD);<br>return 0;<br>} |

程序 10-8 是个 Windows 程序，其功能与程序 10-7 相同，然而它是用 64 位 MASM 汇编代码写的。第 6 章详细讲解了怎样在 64 位 Windows 程序中遵照 Microsoft x64 约定来调用函数。整个程序都带有注释，而且分成了几个部分以便于大家理解，例如在给栈分配空间的那一段中就分别解释了其中那三行 SUB 指令的含义，使你明白从 *esp* 中减去的值是为了给哪些数据预留空间。

## 10.4　小结

现在总算把这些复杂的知识讲过一遍了。对于处理器与体系结构来说，你还有许多细节需要了解，这一章只是描绘出了这些知识的轮廓并给出了必要的信息，使你能够由此出发继续学习系统调用与底层系统编程。此外，笔者还概括地讨论了与处理器模式及内存模型有关的系统机能。要是想深入研究这些话题，可以从第 10 章的补充材料入手，不过有言在先：这些话题可是相当费脑筋的。

**程序 10-8　在汇编代码中，按照 Microsoft x64 约定来调用 Windows API**

Windows (MASM 64-bit)

```
extrn ExitProcess : proc
extrn MessageBoxA : proc
extrn CreateFileA : proc    ; A = ANSI, W = Unicode
extrn ReadFile : proc
extrn CloseHandle : proc
extrn GetLastError : proc   ; for troubleshooting if needed

.DATA
num QWORD 13
inputFile BYTE "input.txt",0
caption BYTE "Message", 0
buffer BYTE 14 DUP (0)

;return data
FD QWORD ?
read QWORD ?
charsRead QWORD 0
button QWORD ?
closed QWORD ?

.CODE
```

Windows (MASM 64-bit)

```
_main PROC                                  ; Print "Computer Arch"

  sub rsp, 10h                              ; reserve for return and rbp
  sub rsp, 18h                              ; reserve for parameters
  sub rsp, 20h                              ; reserve shadow space for regs

  lea rcx, inputFile                        ; address of file name
  mov rdx, 80000000h                        ; constant for GENERIC_READ
  xor r8, r8                                ; 0 share mode
  xor r9, r9                                ; 0 security
  mov QWORD PTR [rsp+48h-28h], 3            ; reverse order, 3 open existing
  mov QWORD PTR [rsp+48h-20h], 80h          ; flags FILE_ATTRIBUTE_NORMAL
  mov QWORD PTR [rsp+48h-18h], 0            ; 0 template
  call CreateFileA
  mov FD, rax                               ; save handle

  mov rcx, FD                               ; pass FD
  lea rdx, buffer                           ; pass address of buffer
  mov r8, num                               ; pass buffer size
  lea r9, charsRead                         ; pass address of charsRead
  mov QWORD PTR [rsp+48h-28h], 0            ; 0 overlap
  call ReadFile
  mov read, rax                             ; save characters read

  xor rcx, rcx                              ; 0 handle owner
  lea rdx, buffer                           ; pass text address
  lea r8, caption                           ; pass caption address
  xor r9, r9                                ; 0 MB_OK
  call MessageBoxA
  mov button, rax                           ; save button clicked

  mov rcx, FD                               ; pass handle
  call CloseHandle
  mov closed, rax                           ; save status

  add rsp, 48h                              ; clean up stack

  mov rcx, 0                                ; return value
  call ExitProcess                          ; exit

_main ENDP
END
```

## 10.5 关键术语

64-bit mode（64 位模式）
application programming interface (API)（应用程序编程接口）
code segment descriptor（代码段描述符）
compatibility mode（兼容模式）
control registers（控制寄存器）
data segment descriptor（数据段描述符）
descriptor privilege level (DPL)（描述符优先级）
exception (trap)（异常（陷阱））
exception handler（异常处理程序；异常处理器）
fast system calls（快速系统调用）
flat (linear) memory model（平面内存模型（线性内存模型））
frame (memory)（（内存中的）帧）
general protection (GP) exception（通用保护异常；一般性保护异常；GP 异常）
Global Descriptor Table (GDT)（全局描述符表）
hardware interrupt（硬件中断；硬中断）
interrupt（中断）
Interrupt Descriptor Table (IDT)（中断描述符表）
interrupt request (IRQ) line（中断请求线；IRQ 线）
interrupt service routine (ISR)（中断服务例程）
interrupt stack table (IST)（中断栈表）
interrupt vector table (IVT)（中断向量表）
kernel（内核）
Local Descriptor Table (LDT)（局部描述符表）

long mode (IA-32e)（长模式（IA-32e 模式））
machine-specific registers (MSRs)（特定于机器的寄存器）
memory management registers（内存管理寄存器）
page (paging)（页面；页（分页））
page directory（页目录）
page directory pointer table（页目录指针表）
page fault（缺页故障；页错误）
page size extension (PSE)（页面尺寸扩展）
page table（页表）
privilege level (PL)（优先级；特权级）
protected mode（保护模式）
real mode（实模式）
segment selector（段选择器；段选择子）
segment registers（段寄存器）
segmented memory model（分段内存模型）
software interrupt（软件中断；软中断）
system call（系统调用）
system call number（系统调用号）
task gates（任务门）
task state segment (TSS)（任务状态段）
virtual dynamic shared object (VDSO)（虚拟动态共享对象）

## 10.6　代码回顾

| | |
|---|---|
| INT | 发出软中断信号 |
| IRET | 特殊形式的 RET 指令，用于从中断例程返回 |
| SYSENTER | Intel 指令，用于发起快速系统调用 |
| SYSEXIT | 与 SYSENTER 相配套的指令 |
| SYSCALL | x86_64 快速调用 |
| SYSRET | 与 SYSCALL 相配套的指令 |
| CALL | 用来调用例程 |
| INVOKE | MASM32 命令，用来便捷地调用 Windows API |

## 10.7　习题

### 填空题

1. 典型的 x86 32 位系统运行在_____模式下。
2. x86_64 系统采用_____内存模型。
3. _____寄存器保存的是指令在 Code Segment（代码段）内的偏移量。
4. 当前的内存架构有两种抽象形式，分别叫作_____与_____。
5. 虚拟内存页面，其尺寸恰好与物理内存_____相同。
6. _____是由处理器所发出的信号，用来表示当前应用程序所无法处理的某种状况。
7. 由于要保存上下文情境并切换_____，因此中断的处理过程会有所延迟。
8. 某些中断必须放在已经确认无误（known good）的_____中加以处理，其地址由 IST 维护。
9. IRQ line（IRQ 线）是用来_____中断的。
10. 执行 INT 指令，是一种向_____发起系统调用的方式。
11. 用来做系统调用的_____指令只能运行于 64 位模式之下。
12. 通过程序库与 API 做系统调用有两个好处，第一是写法更为一致，第二是便于_____。
13. 内核在初始化的过程中会用到_____，以便更为抽象地管理系统例程。
14. 类 Unix 系统遵照_____ABI（Application Binary Interface，应用程序二进制接口）来实现 64 位系统调用。
15. _____调用约定使用 *rcx* 寄存器传递参数。

**判断题**

1. 为了限制访问系统寄存器，指令集没有提供对应的载入及存储指令。
2. x86 及 x86_64 系统不再使用实模式。
3. 异常处理程序实际上是一种函数（或者说软件例程）。
4. 所有的 64 位中断都必须以 64 位例程的形式来执行，即便在兼容模式下也是如此。
5. 各种汇编器均支持 INVOKE 命令。

## 10.8 作业

### 10.8.1 复制文件

在系统中创建一份文件，将用户对该文件的权限设为只读（read-only，此模式用八进制表示是 444o）。然后，用你习惯的平台及汇编器编写程序，把该文件复制到新的地方。你需要获取这两个文件的 handle（操作标识；句柄），以便将前一份文件的内容复制给后一份文件，并把用户对后者的权限设为读 / 写（read/write，此模式用八进制表示是 666o）。不要改动原来的文件。你可以显示一条信息告诉用户文件已经复制完毕。程序可以采用本章所提到的各种系统调用，但是整个代码都必须用汇编来写。

### 10.8.2 日期与时间

用你习惯的平台与汇编器制作程序，通过系统调用来获取当前的日期及时间，并将其显示到屏幕上（可以打印到控制台，也可以展示到消息框中）。你可以使用本章提到过的任何一种系统调用，但是整个程序必须用汇编代码来写。假如改用高级语言中的程序库及 API 来获取时间并调整其格式，那么这道题就相当简单了。反过来说，用汇编语言实现这些功能是有些棘手的，并不像听上去那样容易。

### 10.8.3 CPUID 及 Vendor ID（挑战题）

本章的最后一条编程指南中提到，你可以参考附录 G 来判断处理器是否支持 SYSENTER 指令，然而在这之前，你首先要判断系统支不支持 CPUID。这项任务有助于你更好地理解 CPU 的特性。

用你习惯的平台与汇编器编写程序，判断系统是否支持 CPUID 指令，然后把结果打印到屏幕上（可以打印到控制台，也可以展示到消息框）。整个程序必须完全用汇编语言写，不能像附录 G 那样用汇编指令及 C++ 混合而成。

当前的各种 CPU 其实都支持 CPUID，因此，你在打印支持情况的时候，还应该把 vendor ID（厂商 ID）也打印出来。例如 vendor ID 的值可能是 GenuineIntel。笔者建议你参考下面三份资料：

- 附录 G
- https://en.wikipedia.org/wiki/CPUID
- https://wiki.osdev.org/CPUID

另外，别忘了参阅正式的 Intel 开发文档，以了解 CPUID 指令的用法。

# 第 10 章补充材料
# 第 10 章中的程序和资源

## 程序

程序 10-1　段选择器的用法

MASM (32-bit)

```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO, dwExitCode:DWORD

.DATA
array DWORD 3, 2, 6, 4, 1
.CODE
_main PROC
   mov eax, OFFSET array
   mov ebx, [eax+4]
   mov ecx, ds:[array]
   mov edx, ds:[array+8]
   INVOKE ExitProcess, 0
_main ENDP
END
```

NASM (32-bit)

```
section .data
array: dd 3, 2, 6, 4, 1

section .text
global _main
_main:

mov eax, array
mov ebx, [eax+4]
mov ecx, [ds:array]
mov edx, [ds:array+8]

mov eax, 1
mov ebx, 0
int 80h
```

程序 10-2　用中断指令插入调试断点，并执行 `write` 系统调用，以便向控制台写入文本

NASM (32-bit)

```
section .data
output: DB "Computer Architecture",0Ah
len: EQU ($ - output)

section .text
global _main
_main:          ; Print "Computer Architecture"

;int 3h         ; Set breakpoint (remove if desired)
mov ebx, 1      ; File descriptor STDOUT (1)
mov ecx, output ; String address
mov edx, len    ; String length
mov eax, 4      ; Use SYSCALL 4 (WRITE), prints to screen
int 80h         ; SYSCALL

mov eax, 1
mov ebx, 0
int 80h
```

程序 10-3　通过 INT 指令打印进程 ID

NASM (32-bit)

```
section .data
    decimal: dd 10
section .bss
    PID: resd 1
```

NASM (32-bit)

```
section .text
global _main
_main:                          ; Print Process ID

mov eax, 14h                    ; GET_PID call
int 80h                         ; system call
                                ; process ID is returned in EAX

mov edi, PID+3                  ; Pointer to end of PID string

convert:
    xor     edx, edx
    div     DWORD [decimal] ; Divide by 10
    add     edx, 48         ; Add 48 to remainder
    mov     BYTE [edi], dl  ; Store
    dec     edi             ; Next digit
    cmp     eax, 0          ; Is EAX = 0?
    jg      convert         ; If greater than 0, more digits

mov ebx, 1      ; File descriptor STDOUT (1)
mov ecx, PID    ; String address
mov edx, 6      ; String length
mov eax, 4      ; system call 4 (WRITE), prints to screen
int 80h         ; system call

mov eax, 1       ; system call 1 (exit)
mov ebx, 0       ; return value
int 80h          ; system call
```

程序 10-6a 在 64 位系统中执行 SYSCALL

GAS (64-bit)

```
.data
output:     .ascii "Computer Architecture\n"
len: .quad (. - output)
file_name: .asciz "output.txt"
file_handle: .long 0

.text
.globl _main
_main:

movq $0x2000004, %rax         # system write (4)
movq $1, %rdi                 # file handle STDOUT (1)
leaq output(%rip), %rsi       # address
movq len(%rip), %rdx          # length
syscall

movq $0x2000005, %rax         # system open (5)
leaq file_name(%rip), %rdi    # path
movq $0x0201, %rsi            # flags (02=O_CREAT, 01=O_WRONLY)
movq $0666, %rdx              # mode (permissions)
syscall
movq %rax, file_handle(%rip)

movq $0x2000004, %rax         # system write (4)
movq file_handle(%rip), %rdi  # file handle
leaq output(%rip), %rsi       # address
movq len(%rip), %rdx          # length
syscall
```

GAS (64-bit)

```
movq $0x2000006, %rax        # system close (6)
movq file_handle(%rip), %rdi # file handle
syscall

movq $0x2000001, %rax        # system exit (1)
movq $0, %rdi                # return value
syscall
.end
```

**程序 10-6b　在 64 位系统中执行 SYSCALL**

NASM (64-bit)

```
section .data
output: DB "Computer Architecture",0Ah
len: DQ ($ - output)
file_name: DB "output.txt",0Ah
file_handle: DD 0

section .text
global _main
_main:

mov rax, 1                   ; system write (1)
mov rdi, 1                   ; file handle STDOUT (1)
lea rsi, [rel output]        ; address
mov rdx, [len]               ; length
syscall

mov rax, 2                   ; system open (2)
lea rdi, [rel file_name]     ; path
mov rsi, 0101o               ; flags (01=O_CREAT, 01=O_WRONLY)
mov rdx, 0666o               ; mode (permissions)
syscall
mov [rel file_handle], rax

mov rax, 1                   ; system write (1)
mov rdi, [rel file_handle]   ; file handle
lea rsi, [rel output]        ; address
mov rdx, [len]               ; length
syscall

mov rax, 3                   ; system close (3)
mov rdi, [rel file_handle]   ; file handle
syscall

mov rax, 60                  ; system exit
xor rdi, rdi                 ; return value
syscall
```

## 资源

### 正式文档

如果你想了解与中断及系统调用有关的寄存器和指令，那么最好是从正规的文档入手。这些信息散布在 Intel 开发手册的各卷及各章中，因此，笔者推荐你下载合订本并在其中搜寻相关的话题。

《Intel 64 and IA-32 Architectures Software Developer's Manual》

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

《System V Application Binary Interface, AMD64 Architecture Processor Supplement》

https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI

《Microsoft Windows API Index》

https://msdn.microsoft.com/library/windows/desktop/hh920508.aspx

**系统调用表**

系统调用表以及其中每种调用的号码在各系统之间是有所区别的。下面列出 macOS/BSD 及 Linux 系统中的相关文件，并指出这些文件可能存放在哪里。

macOS/BSD

文件：`syscalls.h`、`fcntl.h`

位置：`/usr/include/sys`

文件：`syscalls.master`

http://opensource.apple.com/source/xnu/xnu-3248.50.21/bsd/kern/syscalls.master

文件：`syscall_sw.h`

用途：定义 system call class（做系统调用时，要确定该调用所属的 class 号码，将其左移 24 位，并与调用号本身相加。例如当 class 号码为 2 时，表示这种系统调用属于 Unix/BSD 类别的调用，因此，系统调用号 1 要写成 0x200001，当 class 号码为 1 时，表示属于 Mach 类别的系统调用，因此，系统调用号 1 要写成 0x1000001）。

与 64 位的系统调用表有关的参考资料

https://sigsegv.pl/osx-bsd-syscalls/

其他参考资料：

https://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/x86-system-calls.html

Linux

文件：`syscall_32.tbl`、`syscall_64.tbl`

位置：`/arch/x86/entry/syscalls`

文件：`fcntl.h`

位置：`/usr/include/` 或 `/usr/include/bits`

与系统调用表有关的参考资料：

http://lxr.linux.no/linux+v4.6.4/arch/x86/entry/syscalls/

http://syscalls.kernelgrok.com（只适用于 32 位环境）

http://www.escapin.it/assembler/asm_syscall.pdf（只适用于 32 位环境）

Windows

DOS API：https://en.wikipedia.org/wiki/MS-DOS_API

将 DOS 调用移植到 WinAPI：https://msdn.microsoft.com/en-us/library/aa984837(v=vs.71).aspx

32 位 Native API：http://j00ru.vexillium.org/ntapi/

64 位 Native API：http://j00ru.vexillium.org/ntapi_64/

**其他在线资源**

Linux Inside – Interrupts（中断）：

https://0xax.gitbooks.io/linux-insides/content/Interrupts/

Linux Inside – System Calls（系统调用）：

https://0xax.gitbooks.io/linux-insides/content/SysCall/

The Definitive Guide to Linux System Calls（Linux 系统调用权威指南）：

http://blog.packagecloud.io/eng/2016/04/05/the-definitive-guide-to-linux-system-calls/

Virtual Dynamic Shared Object (VDSO) man page（VDSO 的 man 手册页）：

http://man7.org/linux/man-pages/man7/vdso.7.html

LKML.ORG thread on merging system call paths in Linux（LKML.ORG 网站的讨论帖，涉及怎样把 Linux 中的系统调用路径合并起来）：

https://lkml.org/lkml/2015/9/1/562

LKML.ORG thread on sysenter and performance in Linux（LKML.ORG 网站的讨论帖，涉及 Linux 系统执行 sysenter 的效率）：

https://lkml.org/lkml/2002/12/9/13

https://lkml.org/lkml/2002/12/18/218

Sysenter Based System Call Mechanism in Linux 2.6 (outdated, but useful explanation)（Linux 2.6 中，基于 sysenter 的系统调用机制，内容虽然过时了，但是其中所做的解释很有帮助）：

http://articles.manugarg.com/systemcallinlinux2_6.html

Nt vs. Zw – Clearing Confusion on the Native API（比较 Nt 系列与 Zw 系列的 API，以消除对 Native API 的困惑）：

http://www.osronline.com/article.cfm?id=257

# 其他架构

## 目标

- 对比 CISC 与 RISC 架构
- 研究各种架构设计方案
- 分析怎样针对 x86 之外的架构撰写汇编程序
- 了解计算机架构的发展趋势

## 网络资源

**维基资源**

- https://en.wikipedia.org/wiki/Category:Instruction_set_architectures
- https://en.wikipedia.org/wiki/ARM_architecture
- https://en.wikipedia.org/wiki/Atmel_AVR
- https://en.wikipedia.org/wiki/RISC-V
- https://en.wikipedia.org/wiki/Z/Architecture
- https://en.wikipedia.org/wiki/Quantum_computing

**技术资源与研究资料**

- https://developer.arm.com/products/architecture（ARM 技术资料）
- http://infocenter.arm.com/help/index.jsp（ARM 参考手册）
- http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf（ATmega 数据手册）
- https://riscv.org/specifications/（RISC-V 规范）
- https://github.com/riscv（GitHub 网站中的 RISC-V）
- https://www.kernel.org/doc/Documentation/s390/Debugging390.txt（s/390 及 z/Architecture 调试资料）
- http://www.ibm.com/developerworks/library/l-basics-inline-assembly/index.html（z/Architecture 汇编）
- http://www.cs.ucsb.edu/~chong/QC/（量子计算概述）

## 11.1 简介

本书根据 x86 及 x86_64 架构讲解汇编语言及架构等概念，这样做不仅仅是因为这两种架构本身值得讲解，更因为它们是 PC（个人电脑）的主流架构。不过除此之外，还有其他一些架构与处理器设计也在持续地演化。本章就要详细地讨论 CISC 与 RISC，并举出一些与其他架构有关的例子，以展现计算机架构当前的变化情况及未来的发展趋势。

## 11.2　CISC 与 RISC

第 1 章在讨论指令集架构的时候简单介绍了 CISC 与 RISC。现在回忆一下：指令集架构，通常可以分为复杂与精简两种。**复杂指令集**（Complex Instruction Set Computing，CISC）架构中的指令长度不一。之所以称为复杂指令集，是因为某条指令可能会执行多项任务（例如既访问内存中的某个位置又执行算术运算）。反之，**精简指令集**（Reduced Instruction Set Computing，RISC）架构中的指令长度通常都一样，而且每条指令只执行一项任务（例如要么访问内存中的某个位置，要么将两个寄存器相加，但不会身兼二职）。

这种把运算操作（例如加法 ADD）与内存访问相分离的设计叫作**加载 / 存储设计**（load/save design）。RISC 架构采用的就是这种设计，这意味着它必须用专门的指令将操作数载入内存，或是将其保存到内存中。例如要想执行 ADD 运算，首先必须从内存中获取操作数并将其放在寄存器中，然后把两个寄存器相加，最后将结果放回内存。这种设计实际上使得 RISC 系统的寄存器比 CISC 系统更多，由于它的静态 RAM 较多而动态 RAM 较少，因此可以节省空间及耗电量，这很适合用在移动设备与嵌入式设备中。

CISC 采用的不是加载 / 存储设计，因为它的操作指令还有可能要负责访问内存。仍然以 ADD 运算为例，CISC 指令集中用来执行该运算的指令可以把某个寄存器中的操作数与内存中某个位置上的操作数直接相加。由于 CISC 处理器会频繁地访问动态内存，因此高速缓存就显得相当重要，只有把它做好才能提升速度与效率。在当前的架构设计中，RISC 与 CISC 都通过大量的晶体管来实现解码器、逻辑单元及**微码**（microcode）机制。因此从生产角度来看，这两种设计其实都很注重缩减晶体管空间，以便装配更多的寄存器或缓存。而且这两者都可以通过多个核心与多个逻辑单元，利用分割指令及数据缓存等机制来实现并行计算。

CISC 与 RISC 架构的例子如表 11-1 所示。

表 11-1　CISC 与 RISC 架构举例

| CISC | RISC | 增强的 RISC |
| --- | --- | --- |
| x86, x86_64, IBM 360 – Z, Motorola 68k, PDP | ARM, RISC-V, MIPS, SPARC | PowerPC, ATmega AVR, PIC |

> **提示：**微码是一种依照具体的处理器而设定的逻辑，通常位于预留的高速内存中。它会把机器指令与系统操作翻译成一系列电路级的操作。微码可以视为一个经过编程的层，它定义了某条指令究竟应该怎样在硬件上执行。

如果不考虑指令流水线这一因素，那么 RISC 与 CISC 系统之间的区别主要体现在指令长度与内存访问上。前者的指令长度是固定的而且大多数指令会在一个时钟周期内执行完毕，而后者的指令长度则不固定，由于指令较为复杂，因此许多指令要花费多个时钟周期才能执行完。这是理论上的差别，从实际角度来看，由于这两种处理器均会运用一些增强技术，因此它们在同一个时钟周期内其实都是可以执行多条指令的。

精简指令集不一定意味着指令的数量少，而复杂指令集也不一定意味着指令的数量多。例如，DEC 在 20 世纪 60 年代生产了一种 PDP（Programmable Data Processor）系统，叫作 PDP-8，这是个采用复杂指令集的计算机，但它仅有 8 种指令。其中较为复杂的主要是 IOT（input/output transfer）与 OPR（micro-coded operations）指令。OPR 指令的长度是 12 位，其

中有 3 位用来表示即将执行的一系列操作位于哪一组中，另外 9 位则用来指明具体的操作及其先后顺序。与此相反，Apple、IBM 及 Motorola 所推出的 PowerPC 虽然用的是精简指令集架构，但是指令的数量却多达 230 种。

> **提示：** 在计算机市场中，x86 CISC 架构几十年来一直是个人电脑的主流架构，而 IBM 的 CISC 设计则在大型电脑市场中占优。移动市场（手机市场）主要由 ARM 等采用 RISC 指令集的处理器所占领，其他的嵌入式设备及特殊设备也偏向于采用 RISC 设计。

RISC 系统倾向于减少专门针对某种数据类型的指令。这种系统中的数据类型以及与之配套的指令一般都比较原始，例如数据类型只分为整数与浮点数等几种，而且它把这几种类型只视为由二进制位所构成的序列。CISC 系统通常会用专门的指令来支持字符串等更为复杂的数据类型。例如早前讲解 x86 架构时，就提到过 MOVS（移动字符串）与 CMPS（比较字符串）指令。

在 CISC 与 RISC 之间对比并不是为了说明哪种架构更好，而是想帮大家找到最适合某台设备或某个应用的架构。本书前面一直在讲解 CISC 架构，下面会介绍几种 RISC 架构，并讲解一种 x86 与 x86_64 之外的 CISC 架构。

## 11.3 更多架构

本节要简单地介绍各种计算机市场中所流行的其他几个架构。讲解每个架构时，笔者都会扼要地叙述它的历史，以及它的指令集架构与寄存器，最后给出一段范例代码。书中还有一些链接，给出了深入研究这些架构所需的资料。在讲解过程中，我们还会将这些架构与 x86 进行对比。

### 11.3.1 ARM

ARM 本来是 Acorn RISC Machines 的缩写，但现在已经成了 ARM Holdings 公司的名字。该公司研发了多种 RISC 处理器架构，它的主要工作是架构与核心设计，它会把这些设计授权给其他公司以供它们去制造相应的产品。前面说过，RISC 设计方案所采用的指令比较简单，而且用来实现逻辑单元与内存访问的晶体管也比较少，因此效率比较高，这使它逐渐垄断了移动设备的市场。采用 ARM 处理器的设备有：Microsoft Surface、Apple iPhone/iPad、Canon 数码相机、Nintendo DS，以及许多 GPS 导航系统。

ARM 在设计上经历了好几代的变化，不过此处主要关注两个领域，一个是 32 位的 ARM 架构 AArch32，一个是 64 位的 ARM 架构 AArch64。ARM 架构有三个不同的 profile（配置）：Application（A，应用）、Realtime（R，实时）及 Microcontroller（M，微控制器）。A 系列的 profile 最为流行，32 位与 64 位处理器中都有采用这种 profile 的产品，用在运行用户应用程序的设备上。R 系列都是 32 位处理器，这种 profile 适合用在实时的，或者对安全要求很高的系统中，例如交通工具与医疗设备。M 系列也是 32 位的，这种 profile 适用于单片机，通常出现在 Arduino 及 NXP 生产的单片机板上。

x86 架构的处理器有保护模式与长模式之分，与之类似，ARM 处理器也可以运行在各种模式下。其中用户模式主要给非特权的处理任务使用，此外还有 10 种特权模式用来执行中断及异常等任务。大多数 ARM 处理器都采用“获取（fetch）– 解码（decode）– 执行

(execute)”这样的传统三阶段流水线，不过，最近出现的几种版本，如 ARM8 与 ARM9，装配有更为复杂的流水线。

ARM 的指令集采用的是加载 / 存储设计，它有一套独特的寄存器，而且通常使用宽度固定的指令，这些指令基本上可以在一个时钟周期内执行完毕。为了提高速度，ARM 在设计上与 x86 有一些区别，例如它只在有需要的时候才会于执行完算术运算之后修改状态码，而且会用专门的寄存器来保存调用完函数之后需要返回到的地址，此外，它的大多数指令都自带条件逻辑，因此不用在执行完之后再去执行分支指令。最后这一点与其他架构区别很大。AArch32 的指令一般用 4 个二进制位来做 predicate（谓词，也就是状态码选择器），以表明这条指令是应该无条件地执行，还是只应该在某种情况下执行。这样做可以免去分支机制以及由此引发的开销，只不过它可能还要多花一个时钟周期来考虑那条只应该在另外一种情况下执行的指令，然后发现当前状态与该条件不符，从而不予执行。ARM 在 AArch64 架构上采用另一种办法来处理分支，它的指令通常不像 AArch32 那样自带分支逻辑。分支预测机制的具体支持情况要根据 profile 及核心来确定。

ARM 处理器支持哪些寄存器，要看它的核心是 32 位还是 64 位，此外还要看是否支持 Thumb 指令集（一种采用 16 位编码的 ARM 指令集）以及它使用的是哪种 VFP（Vector Floating-Point unit，向量浮点运算单元）。下面给出用户模式中两套最为常见的组合。

- AArch32：16 个 32 位的寄存器 *r0* ～ *r15*，其中 *r13* 用作栈指针（Stack Pointer，SP），*r14* 用作链接寄存器（Link Register，LR），用来保存函数调用的返回地址，r15 用作程序计数器（Program Counter，PC）。此外还有一个 32 位的寄存器叫作当前程序状态寄存器（Current Program Status Register，CPSR），其中包含一些与处理器管理有关的二进制位以及一些与算术有关的标志。许多 32 位 ARM 核心都配有这样一种 VFP，它有 16 个或 32 个 64 位的浮点寄存器，可以存放单精度或双精度值。
- AArch64：31 个 64 位的通用寄存器 *x0* ～ *x30*，它们的 32 位形式为 *w0* ～ *w30*，其中 *w29* 用作帧指针（Frame Pointer，FP），*x30* 用作 *LR*。此外还有专门的 *SP* 寄存器、*PC* 寄存器，以及 32 个 128 位的 FPU 寄存器。同时使用 Advanced SIMD extension（也就称为 *NEON*）指令集，支持 8 位、16 位、32 位及 64 位的整数，此外也支持单精度及双精度 SIMD 运算。

尽管 AArch64 的内存空间与地址都是 64 位，但指令依然是 32 位的。其中的操作数可以是 32 位也可以是 64 位。不同的 ARM 核心可能会采用不同的设计方案与增强技术。

**程序 11-1　用 ARM 汇编代码所写的 Blink（闪烁）程序，适用于 Arduino Due**

ARM

```
// written in Arduino 1.6.10

void setup() {
  asm volatile(                    // same as pinMode(13, OUTPUT)
    "mov r0, %[led] \n\t"          // move I/O address to r0
    "mov r1, #1     \n\t"          // store 1 in r1
    "lsl r1, #27    \n\t"          // shift left 27 bits, mask for pin 13
    "str r1, [r0]   \n\t"          // store contents of r1 into address in r0
    : : [led] "r" (&REG_PIOB_OER) // I/O Port B enable output register
  );
}

void loop() {
```

ARM

```
    asm volatile(
      "push {r0-r3, lr}    \n\t"    // save registers
      "mov r0, %[ledOn]    \n\t"
      "mov r1, #1          \n\t"
      "lsl r1, #27         \n\t"
      "str r1, [r0]        \n\t"    // turn LED on
      "mov r0, #1000       \n\t"    // constant parameter for delay(n)
      "bl delay            \n\t"    // branch to ARM delay(n) function
      "mov r0, %[ledOff]   \n\t"
      "mov r1, #1          \n\t"
      "lsl r1, #27         \n\t"
      "str r1, [r0]        \n\t"    // turn LED off
      "mov r0, #1000       \n\t"
      "bl delay            \n\t"
      "pop {r0-r3, lr}     \n\t"    // restore register values
      "bx lr               \n\t"    // branch to address in link register
      : : [ledOn] "r" (&REG_PIOB_SODR), [ledOff] "r" (&REG_PIOB_CODR)
    );  // SODR: set output data register; CODR: clear output data register
}
```

为了演示 ARM 架构笔者给出了程序 11-1，这是个适用于 Arduino Due 单片机的程序。Arduino Due 配有 32 位的 ARM Cortex-M3 处理器，该处理器的运行频率是 84MHz。Arduino 设备上的程序要用一种类似于 C 语言的方式来写，而且要由 AVR-GCC 编译。程序 11-1 把一些 ARM 汇编指令内联到了这种类似 C 语言的代码中，这正如第 9 章的范例代码把 x86 的汇编指令内联到了 C 语言的代码中一样。程序 11-1 是针对 Arduino 设备而写的 Blink 程序，功能是让板卡上的 LED 灯闪烁，此处以这个程序为例主要是为了展示 ARM 架构与其他架构在指令助记符、寄存器以及语法等方面的共性与区别。

ARM 的 MOV、PUSH 及 POP 这三个指令助记符与 x86 中的写法相同，此外还有几个不一样的助记符，例如 LSL（Logical Shift Left，逻辑左移）、STR（STore Register，存储寄存器）以及 BL（带有链接的分支）。寄存器的写法是 r 后面跟上寄存器的序号，这在刚才介绍 AArch32 的时候已经说过了，数值型的立即值以 # 开头。由于 AVR-GCC 也是一种 GCC，因此内联汇编代码所依循的模板格式与早前讲过的范例代码差不多，也分成 output（输出）、input（输入）以及 clobbers（内联的汇编代码所修改的寄存器）这三部分。代码中的注释详细解释了指令的具体含义。

**范例 11-1　对程序 11-1 中的 `setup()` 做反汇编**

ARM

```
80148: 4b03          ldr    r3, [pc, #12]    ; (80158 <setup+0x10>)
8014a: 4618          mov    r0, r3
8014c: f04f 0101     mov.w  r1, #1
80150: ea4f 61c1     mov.w  r1, r1, lsl #27
80154: 6001          str    r1, [r0, #0]
80156: 4770          bx     lr
80158: 400e1010      .word  0x400e1010
```

把程序 11-1 的 `setup()` 函数在范例 11-1 中反汇编之后可以看到 ARM AArch32 的一些特征。首先，指令长度均为 32 位，例如 `f04f 0101` 这样的指令就是如此。只不过其中有些指令是以压缩形式的 16 位 Thumb 指令出现的（例如 `4b03`），这样可以使代码更加紧凑。其次，字长是 32 位。再次，编译器会利用 PC 及 LR 寄存器来生成开场及收场代码。最

后，它会把 LSL 指令改写为 MOV 指令的一部分，使得程序能够在执行这条 MOV 指令的过程中顺带完成移位操作（对于 ARM 来说，这样的任务仍然只需要用 1 个时钟周期即可完成）。为了对程序 11-1 做反汇编，笔者把编译过程中创建的 `.elf` 临时文件拿出来，将其中的内容交给 Arduino 硬件工具包里的 `objdump` 来处理。这个 `objdump` 是专为 ARM 而设计的版本。

> 学习指南：程序 11-1 及程序 11-2 都是针对 Arduino 设备而写的，与这些设备有关的信息可参见下面两个链接：
>
> - https://www.arduino.cc/en/Main/ArduinoBoardDue
> - https://www.arduino.cc/en/Main/ArduinoBoardUno

### 11.3.2　AVR

AVR 是一种高级的（增强的）8 位 RISC 架构，它由 Atmel 所开发。讲这个架构是因为它采用了简洁的 8 位设计方案与简单的指令集，并且在 Arduino Uno 这样的单片机板上很流行。AVR 的设计方案由另一个架构改编而来，那种架构叫作 Harvard 架构，它把指令与数据存储从物理层面分隔开来。AVR 对此做了修改：尽管程序与数据依然保存在（同一条内存的）不同空间中，但是有一些特殊的指令可以把程序内存当成数据访问。

和其他架构一样，AVR 也分成不同的系列。此处关注的是相当流行的 8 位 ATmega AVR 设计。某些单片机上用的是一种 32 位版本，叫作 AVR32，它主要是为了同 ARM 竞争，其架构与许多较为复杂的处理器核心类似，也支持 SIMD 等技术。本节要讲的 ATmega 系列在单片机市场上占据很高的份额，而且尤其适合做原型开发。笔者以其中的 ATmega328p 为例来讲述该系列的特性，在笔者写作本书时，这个型号所采用的设计方案是整个系列中最新的。其实 ATmega 系列中的各种型号都差不多，只是内存大小有所区别。

> 学习指南：Harvard 架构最先出现在 20 世纪 40 年代早期的 Harvard Mark I（马克一号）计算机上，这种计算机当时用来执行与战争有关的计算工作。由于 Mark I 是较早出现的一种主要计算机系统，因此，计算机科学专业的入门课程通常会在讲解计算机发展史的时候提到它。尽管这台机器出现时间很早，但是它所采用的某些设计理念仍然体现在当代的架构中。

刚才说过，AVR 是把程序与数据分别存放在两块内存中。它的所有内存空间都呈线性排布（内存空间是平面的）。ATmega328p 的程序内存采用的是 32k 闪存，其中 0.5k 预留给 bootloader（引导装载程序）。虽然 ATmega 是 8 位架构，但程序指令却是 16 或 32 位的，因此内存空间可以用 16 位的地址来表示，这使得开发者能够在 `0x0000` 至 `0x3FFF` 这个范围内访问。数据内存采用的是 2k 静态 RAM，其中包含通用寄存器堆、I/O 寄存器、扩展 I/O 空间以及用作程序栈空间的主数据内存，如表 11-2 所示。访问数据内存需要两个时钟周期，但由于它的 32 个通用寄存器直接与 ALU 相连，因此，那些只涉及寄存器的操作可以在一个时钟周期内完成。此外，还有 1k 的 EEPROM（Electrically-Erasable Programmable Read-Only Memory，电可擦除可编程只读存储器）可以用来存放数据。

表 11-2 ATmega328p 的 SRAM 数据内存布局

| SRAM 中的区域 | 地址范围 |
|---|---|
| 32 个通用寄存器 | 0x0000 ～ 0x001F |
| 64 个 I/O 寄存器 | 0x0020 ～ 0x005F |
| 160 个扩展 I/O 寄存器 | 0x0060 ～ 0x00FF |
| 数据内存 | 0x0100 ～ 0x08FF |

寄存器是 8 位的，可以用来存放 8 位的输入操作数与输出操作数，如果把两个寄存器合起来用，还可以表示 16 位操作数。32 个通用寄存器为 *r0* ～ *r31*，此外还有一个 8 位的 SREG（Status Register，状态寄存器），以及 SPL（Stack Pointer Low，栈指针的低半段）与 SPH（Stack Pointer High，栈指针的高半段）这两个 8 位寄存器，此二者合起来构成 16 位的 SP（Stack Pointer，栈指针）以指向内存空间中的某个位置。表 11-3 演示了 SREG 中的各个二进制位，这与 x86 的 flags 寄存器类似。

表 11-3 AVR 的状态寄存器（SREG）

| 二进制位 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 含义 | interrupt (I) | bit copy (T) | half carry (H) | sign (S) | overflow(V) | negative (N) | zero (Z) | carry (C) |

从表 11-2 中可知，这些寄存器本身也是有内存地址的，因此，开发者可以把一个 16 位的值当成内存空间中的位置（或者说当成指针）来用，使其指向某个寄存器。*r26* ～ *r31* 这 6 个 8 位寄存器可以两两成组，构成 3 个 16 位寄存器，也就是表 11-4 中的 X、Y、Z 寄存器。这些寄存器能够存放指向某个寄存器的指针。

表 11-4 AVR 的 X、Y、Z 寄存器

| 寄存器指针 | *X* | | *Y* | | *Z* | |
|---|---|---|---|---|---|---|
| 使用的寄存器 | R27 (XH) | R26 (XL) | R29 (YH) | R28 (YL) | R31 (ZH) | R30 (ZL) |

程序 11-2 是针对 Arduino Uno 单片机写的，用来演示 AVR 架构。这种控制器配有 8 位的 ATmega328p 处理器，其运行频率是 16MHz。该程序与程序 11-1 类似，都以内联形式来书写汇编代码，只不过它写的是 AVR 汇编代码而非 ARM 汇编代码。此处给出这个程序主要是想展示 AVR 架构与其他架构在指令助记符、寄存器以及写法等方面的区别。该程序的功能是令板卡上的 LED 灯闪烁。

程序 11-2 定义了一些常量，以此来演示 AVR 的某些特点。其中一个特点在于，内联式的 AVR 汇编代码能够与外围的 C 语言代码很好地融合起来（其实 ARM 及 x86 的内联式汇编代码也是如此）。`setup()` 函数与程序 11-1 中的同名函数类似，体现了系统常量的用法，而 `loop()` 函数则演示了怎样给这些常量起一个更有意义的别名，以便在内联式的汇编代码中使用。这个程序本来可以写得更有效率一些，不过此处所用的这种分步写法更加清晰，而且能够显示出 AVR 汇编的特点。

ATmega 的寄存器是 8 位的，但程序使用的 `DelayTime` 变量却是 32 位的。虽然这个变量的实际值只需要 24 个二进制位就能表示得下，但此处还是选用了 32 位的整数，以便将来可以把每次闪烁的时间设置得更久一些。要想在 8 位环境下操作 32 位整数，需要把它切割成 4 个长度为 8 位的小块，然后在这些小块上执行适当的算术指令（例如本例中的一系列减法指令），以便对整个数值做运算。第 9 章说过 GCC 的内联式汇编代码，如果不给 output 列表与 input 列表里的参数定义符号名称，那么就可以用 %0、%1、%2 这样的编号来引用这

些参数。AVR 还允许开发者以 A、B、C、D 这 4 个字母分别指代 32 位数据中从高到低的那 4 个 8 位的小块。于是 %D2 指的就是 %2 参数中权重最低的字节，而 %2 参数则是这段内联汇编代码的 input 列表中所写的第 2 个（从 0 开始编号）参数，即 DelayTime。mainLoop 这段代码把 DelayTime 的值分别放在 *r16*～*r19*，以及 *r20*～*r23* 这两处，以便给表示开灯状态与关灯状态的两个计数器设定初始值。

input 列表里的参数和 clobbers 列表里的寄存器的写法与 x86 内联汇编代码相似，只不过表达约束的字母有所不同（例如 AVR 架构用的是 I、n、d 等字母）。开发者可以给参数定义符号名称，从而在汇编代码中采用 %[port] 这样的写法指代该参数，而无须再使用 %0 这样的序号式写法。AVR 的某些助记符与 x86 及 ARM 一样，例如 MOV，但是减法、分支以及 I/O 指令则有所不同。程序 11-2 中的那些注释更为详细地解释了语句的含义。

**程序 11-2　用 AVR 汇编代码所写的 Blink（闪烁）程序，适用于 Arduino Uno 设备**

AVR

```
// written in Arduino 1.6.10
#define LEDPort PORTB           // Arduino pin 13 is bit 5 of port B
#define LEDPort_Direction DDRB  // DDRB is the port B direction register
#define LEDBit 5                // Constant for bit 5
#define Clock_MHz 16
#define MilliSec 1000000
#define DelayTime (uint32_t)((Clock_MHz * MilliSec / 5)) // set to any rate

void setup() {          // same as pinMode(13, OUTPUT)
  asm volatile (        // shows default parameter use instead of defines
    "sbi %0, %1 \n\t"  // sets port and bit direction
    : : "I" (_SFR_IO_ADDR(DDRB)), "I" (DDB5) // I: 6-bit positive const int
  );}

void loop() {
  asm volatile (
     " mainLoop: "                          // move DelayTime to registers
     "    mov r16, %D2  \n\t"               // LSB of DelayTime
     "    mov r17, %C2  \n\t"               // A2, B2, C2, D2 each is 8 bits
     "    mov r18, %B2  \n\t"
     "    mov r19, %A2  \n\t"               // MSB of DelayTime
     "    mov r20, %D2  \n\t"
     "    mov r21, %C2  \n\t"
     "    mov r22, %B2  \n\t"
     "    mov r23, %A2  \n\t"
     "    sbi %[port], %[ledbit] \n\t"      // set I/O bit (turn LED on)
     " onLoop: "
     "    subi r23, 1   \n\t"               // subtract constant from register
     "    sbci r22, 0   \n\t"               // subtract w/carry const from reg
     "    sbci r21, 0   \n\t"
     "    sbci r20, 0   \n\t"
     "    brcc onLoop   \n\t"               // branch if carry cleared
     "    cbi  %[port], %[ledbit] \n\t"     // clear I/O bit (turn LED off)
     " offLoop:"
     "    subi r19, 1   \n\t"
     "    sbci r18, 0   \n\t"
     "    sbci r17, 0   \n\t"
     "    sbci r16, 0   \n\t"
     "    brcc offLoop \n\t"
     :                                      // no output variables
     : [port] "n" (_SFR_IO_ADDR(LEDPort)), // input variables
       [ledbit] "n" (LEDBit),               // n: integer with known value
       "d" (DelayTime)                      // d: greater than r15
```

AVR

```
    : "r16","r17","r18","r19","r20","r21","r22","r23"  // clobbers
    );}
```

### 11.3.3 RISC-V

虽然很多软件都是私有的，但软件行业已经朝着开放软件的方向行动了。硬件行业也一样，尽管计算机历史上的大多数指令集架构都是专属于某个公司的，但是开放硬件与开放指令集架构的设计理念已经越来越流行了。例如，Arduino 单片机就是一种开源硬件。在指令集架构方面也有开源的例子，例如位于业界前沿的 RISC-V（读作 risk-five）就是一种开源的 RISC 架构。

RISC-V 架构诞生于研究及教学活动中，这些活动主要由加州大学伯克利分校所引领。虽然该架构现在仍然用在研究机构与专业场合中，但由于它是开源的而且有 RISC-V Foundation 组织推进，因此将来会在计算机行业发展得很好。RISC-V 既可以由开源的处理器实现，也可以由不开源的处理器实现。

> 📖 **学习指南：** 与 RISC-V 有关的设计特性、规格、新闻及信息，可以参见下面这两个链接：
>
> - https://riscv.org/
> - https://riscv.org/risc-v-foundation/

和本章讲解其他架构时一样，笔者在这里讲 RISC-V 架构主要也是为了激发兴趣，促使你继续探索这个领域。它的指令集用的是可伸缩的地址空间（例如可以是 32 位、64 位或 128 位），其指令编码的长度也同样可以伸缩。RISC-V 指令的基本格式叫作 **RV32I**，这是一种 32 位的格式，然而也能够以 16 位的小块为单位编码成任意的长度，例如 16 位长、32 位长、48 位长、64 位长。它的 32 个通用寄存器为 *x0* ～ *x31*，其中 *x0* 的值总是零。每个线程都有自己的寄存器状态，而且有各自的程序计数器寄存器及控制与状态寄存器（CSR，control and status register）。由于内存模型较为灵活，而且每个程序线程都是按照一定顺序独立执行的，因此，有时可能要通过 FENCE 这样的特殊指令来确保操作的效果能够得以同步。

对于 **RISC-V 调用约定**（RISC-V calling convention）来说，每一个通用寄存器都可能扮演着某种角色。例如 *x1* 用作链接寄存器，以存放返回地址（RA），*x2* 用作栈指针（SP），*x10* ～ *x17* 用作参数寄存器（此时称为 *a0* ～ *a7*）。其他的寄存器也有着特定的用途。

> 📖 **学习指南：** 为什么需要有一个值总是为 0 的寄存器呢？因为这样做可以用更少的指令来实现某些操作，进而降低逻辑门的数量。比方说，有了这种常为 0 的寄存器之后，就可以不再用 MOV 指令来移动（或者说复制）寄存器的值了。要想在 RISC-V 中把某个寄存器的值移入另一个寄存器，可以通过 `addi rd, rs, r0` 这样的加法指令实现。它的意思是把源寄存器 *rs* 的值与常为 0 的寄存器 *r0* 的值相加，并将结果保存到目标寄存器 *rd* 中，这实际上达成了 MOV 的效果。

RISC-V 的基本实现方式只包含整数操作。如果要支持更为复杂的操作就必须扩展。

例如，要想支持乘法（multiplication）与除法，就必须实现 M 扩展。要想支持原子操作（atomic operation），就必须实现 A 扩展。厂商可以实现 F 扩展以支持单精度浮点操作，这会多出来 32 个浮点寄存器以及一个控制与状态寄存器。D 扩展能够支持双精度浮点数，并且会把 *f* 系列的寄存器拓宽到 64 位。此外，RISC-V 还规划了其他一些扩展，例如用来支持四倍精度的 Q 扩展，以及 16 位压缩格式的 C 扩展。

程序 11-3 的代码是从 RISC-V 在 GitHub 网站所发布的内容（https://github.com/riscv）中找到的，笔者修改了其中的某些注释。通过这段代码可以看出，RISC-V 汇编代码的写法与 x86 架构的 GAS 代码很像，因为 RISC-V 工具链实际上就是一种标准的 GNU 交叉编译器，只不过它是针对 RISC-V 而设的。这种代码可以分别用 RISC-V 版本的 `gcc`、`as` 与 `ld` 工具来做编译、汇编或链接。`.data`、`.align`、`.dword` 等命令的用法，以及标签的写法（例如代码中的 `fail:`）都与 GAS 一样。语句右侧的注释详细解释了语句的含义。请注意，字长是 32 位。

**程序 11-3　RISC-V 单元测试**

RISC-V

```
# Modified from https://github.com/riscv/riscv-tests
.include "riscv_test.h"         # defines the macros (in all-caps)

RVTEST_RV64U                    # define TVM used by program

# Test code
RVTEST_CODE_BEGIN               # Start of test code
        lw      x2, testdata    # load word into x2 (41)
        addi    x2, 1           # add 1 to x2 (our result)
        sw      x2, result      # store word into memory overwriting -1.
        li      x3, 42          # load immediate into x3 (desired result)
        bne     x2, x3, fail    # branch if not equal: Fail if no match
        RVTEST_PASS             # else success
fail:
        RVTEST_FAIL
RVTEST_CODE_END                 # End of test code

# Input data section
# This section is optional, the data is NOT saved in the output
.data
        .align 3
testdata:
        .dword 41

# Output data section
RVTEST_DATA_BEGIN   # Start of test output data region
        .align 3
result:
        .dword -1
RVTEST_DATA_END     # End of test output data region
```

这几节举例讲解了 ARM、AVR 以及 RISC-V 这几种与 CISC 相抗衡的 RISC 架构。除此之外还有一种 RISC 架构也很流行，即 Power Architecture（参见 https://www.power.org/），它其实是一系列技术的统称，这些技术广泛地使用于游戏系统、服务器，以及超级计算机等高端系统中。

### 11.3.4 System-z/Architecture

如果直接对比 RISC 指令集与 CISC 指令集的数量，那么 RISC 更多一些，但是，在个人计算机市场上占据主流地位的 x86 指令集，采用的却是 CISC 设计，而且，在大型计算机市场上占主流地位的 IBM System – z/Architecture 用的也是 CISC 设计。System/ 架构是一种强大的 32 位架构，用于 IBM 的大型计算机。它最先出现在 1964 年的 System/360 机型中。其后又出现了 System/370 机型，该系列的机型在 1970 ～ 1990 这 20 年间不断地演化。接下来，出现了 System/390 机型，这些型号的计算机在 1990 ～ 2000 年间占据着市场，此后诞生了 zSeries（z 系列）。z/Architecture 是 IBM 给自己的 64 位指令集所起的名字，它可以兼容早至 System/360 的许多种机型。IBM 也开发专门的操作系统，以发挥其 CISC 设计的优势，例如 z/OS 就属于这种系统，此外，还有 Linux on System z 可供选用。

z/Architecture 的所有东西几乎都是 64 位的，只不过由于要保持向后兼容，它需要顾及早期 System/ 架构中的 32 位操作并为其实现 64 位版本，这使得 z/Architecture 同时支持 32 位与 64 位的操作。从 32 位变为 64 位使得指令数量增加到三百多种。地址空间是 64 位的意味着可以使用的空间多达 16 EB。截至笔者写书时，寄存器的大小尚不一致：16 个通用的寄存器（*r0* ～ *r15*，或者 *gpr0* ～ *gpr15*）是 64 位的，16 个控制寄存器（*cr0* ～ *cr15*）是 64 位的，16 个存取寄存器（access register，*ar0* ～ *ar15*）是 32 位的，16 个浮点寄存器（*fp0* ～ *fp15*）是 64 位的，此外还有一个 32 位的浮点控制（Floating-Point Control，FPC）寄存器和一个 128 位的程序状态字（Program Status Word，PSW）寄存器。该寄存器依然用作程序计数器，此外还有其他一些功能。

为了演示 z/Architecture，笔者根据 www.kernel.org 网站中的调试范例编写了范例 11-2 与范例 11-3。现在假设我们有下面这个简单的 C++ 程序以供测试。

> **编程知识：** Denis Barrow 所写的《Debugging on Linux for s/390 & z/Architecture》一文提供了与内存布局、栈帧及寄存器用法有关的信息。这篇文档的链接是：https://www.kernel.org/doc/Documentation/s390/Debugging390.txt。

范例 11-2 供测试用的 C++ 程序

C++

```
int test (int b) {
  return (b + 5);
}

int main(int argc,char *argv[])
{
  return (test(5));
}
```

范例 11-3 是对范例 11-2 做反汇编得到的，它能够体现出 z/Architecture 的几项特征。第一可以看到这些汇编指令遵循的是 AT&T 语法。第二，CISC 中的指令长短不一，这尤其体现在 `main()` 的反汇编代码上。第三，参数要尽可能通过寄存器来传递。第四，系统会自动为函数生成开场代码与收场代码，以便正确地保存并恢复相关的寄存器。在 z/Architecture 的调用约定中某些寄存器有着特殊的作用，例如 *r11* 用作帧指针，*r14* 用作返回地址，*r15* 用作栈指针，*r2* 用作返回值，*r7* 用作指向参数的指针等。如果执行顺利则 `test`

应该返回整数 10。

System/ 与 z/Architecture 所采用的 CISC 指令集可以很好地同 x86 的 CISC 指令集相对照。如果你对 64 位大型计算机上的软件开发感兴趣，那么应该继续研究 z/Architecture。

**范例 11-3　对范例 11-2 做反汇编所得到的指令**

z/Architecture

```
00000000800005b0 <test>:
int test(int b)
{ return(b+5);
 800005b0:    a7 2a 00 05    ahi    %r2,5     # add 5 to r2
 800005b4:    b9 14 00 22    lgfr   %r2,%r2   # downcast to integer (long)
 800005b8:    07 fe          br     %r14      # branch to address in r14
 800005ba:    07 07          bcr    0,%r7     # conditional branch
}

00000000800005bc <main>:
main(int argc,char *argv[])
{
 800005bc:    eb bf f0 58 00 24  stmg    %r11,%r15,88(%r15)  # save registers
 800005c2:    b9 04 00 1f        lgr     %r1,%r15            # load stack pointer
 800005c6:    a7 fb ff 60        aghi    %r15,-160           # point to mem pool
 800005ca:    e3 10 f0 00 00 24  stg     %r1,0(%r15)         # save location
return(test(5));
 800005d0:    a7 29 00 05        lghi    %r2,5                 #load 5 into r2
 800005d4:    c0 e5 ff ff ff ee  brasl   %r14,800005b0 <test>  #branch to test
 800005da:    e3 40 f1 10 00 04  lg      %r4,272(%r15)         #r4 <- ret address
 800005e0:    eb bf f0 f8 00 04  lmg     %r11,%r15,248(%r15)   #restore registers
 800005e6:    07 f4              br      %r4                   #branch, return
}
```

## 11.4　量子架构

随着科学与技术的发展，量子架构（quantum architecture）正在掀起波澜。这里的波字既可以指一种潮流和趋势，又可以指物理上的波。计算机领域始终要设法利用各种能源。比方说，电能与磁能就是可以用于计算的传统能源。然而量子系统所使用的能源却是光子能这样更为基本的能源，这些光子是离散量，也可以说是**量子**（quantum）。于是，**量子架构**指的就是根据基本粒子来构建信息单位和通信单位的架构。

传统架构的信息单位是位，它是二进制数位，可以用来表示 0 或 1，直到状态改变为止。换句话说，在状态稳定的情况下，某个位要么始终处在值为 0 的状态，要么始终处在值为 1 的状态。第 1 章讨论了怎样用物理性质来表示这种二进制位，例如可以用存储单元是否充电来表示，也可以用晶体管是否允许电流通过来表示。

量子架构的信息单位是量子位（qubit），它是量子数位，尽管观测到的状态是确定的，但这种状态实际上由两个状态经过**态叠加**（superposition）而形成。换句话说，虽然观测到的值不是 0 就是 1，但无论是哪个值，其实都是由两种状态同时叠加而成的。物理粒子（以及它们的某些性质）可以用来表示量子位，例如光子的偏振（polarization，它可以有水平及垂直两种状态），又例如原子的自旋（spin，它可以有上旋及下旋两种状态）。

传统的字节由 8 个位组成，由于每个位可以有两种取值，因此整个字节可以有 $2^8$（也就是 256）种取值。某个字节的具体取值肯定是这 256 种中的一种。与之类似，量子字节（qubyte）由 8 个量子位组成，由于每个量子位可以有两种取值，因此整个量子字节也可以

有 256 种取值。但是无论你观测到的量子字节值是这 256 种取值中的哪一种，它其实都是这 256 种状态同时叠加所产生的结果。也就是说，在观测时量子字节值的这 256 种可能性会塌陷成一种可能性，也就是你观察到的那种。

〈!〉**提示：**传统位与量子位之间的区别可以从概率的角度来表述。比方说有 $a$、$b$ 这样两种状态。在经典系统中，某个位处于 $a$ 状态的概率与它处于 $b$ 状态的概率相加肯定等于 1，也就是 $a+b=1$。而在量子系统中，量子位处在某种状态的概率等于对各系数的量取平方（这个平方后的值就是该系数所对应的那种状态的观测概率），并将其相加。这个总和也等于 1（$|a|^2+|b|^2=1$），这意味着它必然处在 $a$ 所指的状态也必然同时处在 $b$ 所指的状态，只不过这两种状态的相位差（phase difference）是值得关注的。

量子具备一项很有益的特性可以用来做计算，这就是 entanglement（纠缠），它是量子之间表现出的一种不同寻常的联系，这种联系跟距离远近无关。比方说，如果两个量子位相互纠缠，那么观测到的值必然相对。例如当光子 A 与光子 B 相纠缠时，如果你测量到光子 A 具备水平偏振，那么测量光子 B，一定会发现它具备垂直偏振，反过来也一样。如果把 A 认定为 0，那么 B 肯定是 1。由于有这样的相互关系，所以同时可以操作多个状态。

量子计算的这种特性对于实际的计算工作有什么好处呢？要理解这个问题关键是要考虑：传统计算机解决问题的方法，在状态叠加与量子纠缠的情况下是什么样的。计算机要通过算法解决问题。如果输入给算法的数据可能有 256 种，那么为了验证该算法能否正确处理这些数据，我们必须对其中的每一种数据依次运行算法，这样总共要运行 256 次。反之，如果改用量子计算机，那么只需要一个逻辑门就能够同时判断这 256 种可能性了，这会令计算速度呈现指数式增长。

不过，量子计算依然有一些问题没有解决。首先，量子的某些性质（例如纠缠）还没有得到充分理解。其次，量子位容易受环境影响，而且容易丧失其状态，这称为**退相干**（decoherence）。要想做量子计算，我们必须把量子位的初始状态设置得与某个具体任务（例如某个打算用 Shor 算法（秀尔算法）来解决的任务）相符，然后用一系列固定的**量子逻辑门**来操纵这些量子位，以产生传统意义上的运算结果。此外还有其他一些困难，例如怎样把量子位初始化为某种已知的状态，怎样测量状态，怎样在量子位之间通信，以及量子位的生命期问题及容错问题。

这些挑战使得大规模生产量子计算机显得颇为困难，而且目前并没有很好的办法。世界上确实有少数几台量子计算机，然而它们都在研究机构与大公司中，而且一般都只有几个量子位。另一方面，传统的计算机系统可以考虑通过其他方式去利用光技术，以提升计算能力。比方说，可以在传统的计算机系统中利用光子互连提升芯片内部与芯片之间的通信速度（参见下面的学习指南）。此外，光纤在速度与能耗方面都比传统的总线要好，而现在已经有人把这两者运用在同一块芯片上了。研究者还发现了一些办法能够在无须改变生产流程的前提下把光子技术集成到传统计算机中，这些办法成本较低，而且很容易批量生产。

📖 **学习指南：**下面给出光子互连的范例，该范例所涉及的处理器基于 RISC-V 架构。

引　自：Sun, C. et al. Single-chip microprocessor that communicates directly using light. Nature 528, 534–538 (2015). doi:10.1038/nature16454.

链接：http://www.nature.com/nature/journal/v528/n7583/full/nature16454.html

## 11.5　小结

本章没有讲解 x86/x86_64 架构，而是介绍了几种常见的 RISC 及 CISC 指令集架构。学习架构与学习编程语言类似，最好能够精通某一种而不是略懂很多种。如果你能够很好地掌握 x86 指令集，那么其他指令集（例如本章提到的这几种）学起来也就不是特别困难了。计算机架构会持续演进，市场也会不断变化，然而基本的架构原则依然相当稳固，这些原则还是可以广泛地运用在各种设计上。当然只是现在如此，将来怎样还要再观察。

## 11.6　关键术语

AArch32
AArch64
ARM
ATmega AVR
atomic operations（原子操作）
AVR
AVR32
bit（（传统计算机中的）位）
complex instruction set computing（CISC，复杂指令集）
decoherence（退相干）
entanglement（纠缠）
Harvard（哈佛架构）
load/store design（加载 / 存储设计）
microcode（微码）
Power Architecture
quantum（量子）
quantum architecture（量子架构）
quantum logic gates（量子逻辑门）
qubit（（量子计算机中的）量子位）
qubyte（量子字节）
reduced instruction set computing（RISC，精简指令集）
RISC-V
RV32I
RVG（指的是 RISC-V 调用约定）
superposition（态叠加）
System/360
System/370
System/390
Thumb
vector floating-point unit（VFP，向量浮点运算单元）
z/Architecture

## 11.7　习题

### 填空题

1. x86_64 是一种采用_____指令集的架构。
2. _____架构在移动计算机市场占主导地位。
3. AArch32 指令中的状态码选择器也称为_____。
4. ARM Cortex-M3 处理器采用的是_____profile。
5. ARM 架构用_____寄存器保存函数调用的返回地址。
6. 由于存在_____指令集，RISC ARM 处理器指令的长度可能彼此不同。
7. AVR 架构是基于_____架构设计的。
8. AVR 是一种_____位的架构，它的地址空间是_____位。
9. _____是一种开源的指令集架构。
10. 执行过程不能间断的一系列指令称为_____操作[⊖]。
11. RISC-V 遵照_____调用约定来调用函数。

⊖ 本书正文没有给出这种操作的定义，只提到了它的名字。参见 11.3.3 节及本章的关键术语。——译者注

12. _____架构的信息单元由基本粒子来充当。
13. 一个光子可以同时处在多种状态中，这项能力叫作_____。
14. 量子的_____现象是指量子位的状态由于环境因素而丢失。
15. 如果两个量子位相互纠缠，那么其中一个的观测值总是与另一个_____。

**判断题**

1. CISC 架构采用的是加载 / 存储设计。
2. 指令集架构中的指令数量与该架构是 RISC 还是 CISC 无关。
3. 对于内联式汇编代码模板中的约束来说，所有的架构采用的都是同一套字符。
4. 64 位架构的寄存器也必须是 64 位的。
5. 对于采用 n 个二进制位来表示数值的经典系统以及采用 n 个量子位来表示数值的量子系统来说，你所观测到的值都是 $2^n$ 种取值中的某一种。

## 11.8 作业

### 11.8.1 关于 CPU 架构的研究报告

针对某款 CPU 撰写两页或三页技术研究报告，这款 CPU 所实现的指令集架构不能是 x86/x86_64，也不能是本章已经讲过的这四种（ARM Cortex-M3、AVR ATmega、RISC-V、z/Architecture）。比方说，可以选择 MIPS R4000、Sun UltraSPARC IV、PowerPC G5、MC 68000、Cell 或 PIC 单片机来研究。

报告需要包含下列内容：

- 技术简介或技术概况，其中要提到曾经用过或正在使用这款 CPU 的设备。
- 指令集架构的历史与细节，其中要包括该架构的概况，例如，它是 RISC 架构还是 CISC 架构？有多少种指令？最适合做哪种类型的应用或计算？你可以参照本章在介绍那四种架构时所采用的格式来撰写这一部分内容。
- 指令流水线的结构，例如，从获取到执行的整个周期是如何设计的？有哪些改进？
- 内存方面的细节（主存、缓存、寄存器、虚拟内存）。
- 其他值得关注的内容，例如某段范例代码。

### 11.8.2 针对 x86/x86_64 以外的架构，撰写汇编程序

选一台你能够操作的设备或模拟器，它的指令集架构不能是 x86/x86_64。用你喜欢的（或者必须采用的）开发环境编写一段针对该架构的汇编代码，用以执行某一项或某几项任务。这些任务由你自选。程序必须包含至少 20 条汇编指令，每行汇编代码都要带有注释，以说明这行代码的用途或是其中所执行的操作。整个程序必须能够顺利地编译 / 汇编并执行，以便演示。你可以编写独立的汇编程序，也可以编写内联式的汇编代码。

### 11.8.3 针对 x86/x86_64 以外的架构，做反汇编

选一个用高级语言所编写的程序，这种高级语言可以是 C、C++、Objective-C 或 Java 等语言，也可以是某种高级语言在单片机上的专门版本。该程序必须是针对 x86/x86_64 以外的架构而写的，或者可以移植到 x86/x86_64 以外的架构。拿你喜欢的反汇编器（或是该架构所要求的反汇编器）对这个程序做反汇编，将反汇编所得的内容保存到文本文件中，并给其中至少 30 条汇编语句加上注释。对它做反汇编的这款程序可以是你自己写的程序，也可以是你能够找到其源代码的某款程序。如果反汇编后的语句超过 30 条（很有可能出现这种情况），那么从中任选一段含有 30 条语句的代码来添加注释。

### 11.8.4 设计 ISA（挑战题）

如果由你来设计指令集架构，那你会把它设计成什么样子？撰写一份抽象的概述文档，给出这种架构的基本信息，例如，这是个多少位的架构？是否采用加载 / 存储设计？寄存器是怎么布局的？是

否支持浮点数？内存模型如何设计？指令流水线如何设计？指令的格式是什么？采用什么样的调用约定？此外，还要给出指令集的样例以及范例代码，说明这种指令集架构与其他指令集架构之间的区别以及它最适合用在哪些设备上或哪些场合中，并解释其原因。

这道题促使你思考：现有的指令集架构存在哪些有待解决的问题，以及还可以在哪些地方加以改进或增强。当然了，你也可以直接从自己的思路出发，不过最好是能够参阅其他指令集架构的规范文档。这种文档大多是好几百页，但如果只考虑基本信息、简要描述、某些示意图以及用来演示原理的范例代码，那么应该可以压缩到 5 ～ 10 页。

# 硬件与电子元件

## 目标

- 了解电子元件
- 描述硬件中的各个部分所具备的功能
- 求解电学方程
- 拆解电子设备，以观察其基本组件

## 网络资源

- https://en.wikipedia.org/wiki/Electric_current
- https://en.wikipedia.org/wiki/Voltage
- https://en.wikipedia.org/wiki/Electric_power
- https://en.wikipedia.org/wiki/Electrical_resistance_and_conductance

## 12.1 简介

欢迎来到最后一章。你能读到这里已经很了不起了。前面那些章节详细讲解了汇编语言及 x86/x86_64 架构，并且概述了软件工程师有可能遇到的其他一些架构。由于这本书主要是写给软件工程师看的，而不是写给计算机及电气工程师，因此笔者并没有讨论电气设计，也没有谈到用来给计算机架构与编程提供支持的那些元件。现在，我们就要在这一章中谈谈这些内容，使你能够大致了解这些电学知识。此外，笔者还会给出一些必要的信息，帮助你构建简单的硬件设备。这些围绕硬件而写的内容可以帮助你更好地学习相关的计算机课程，更好地完成自己或是公司的项目，而且对于设备原型开发、嵌入式开发，或单片机板与元件的开发工作也很有帮助。总之，这些内容可以让你在面对硬件时更有准备。

## 12.2 电学基础

### 12.2.1 基本的物理量

首先，一定要理解电学的基本物理量，这包括电流（current）、电压（voltage）、电功率（power）以及电阻（resistance）。电流通常用大写字母 I 表示（I 是 intensity 的首字母），电压可以用 E 或 V 表示[⊖]，但 V 一般用来表示电压的单位伏特而不是电压这个物理量本身。功率通常用字母 P 表示，电阻通常用字母 R 表示。这些基本的物理量及其单位列在表 12-1 中。

⊖ 有时也可以写成 U。——译者注

表 12-1　电学的基本物理量及其单位

| 物理量 | | 单位 | |
|---|---|---|---|
| 名　称 | 缩　写 | 名　称 | 缩　写 |
| Current（电流） | I | Ampere（安培） | A（安） |
| Voltage（电压） | E 或 V | Volt（伏特） | V（伏） |
| Power（电功率） | P | Watt（瓦特） | W（瓦） |
| Resistance（电阻） | R | Ohm（欧姆） | Ω（欧） |

接下来笔者给每个物理量下定义，以帮助你更好地理解它们之间的关系。

**电流**可以定义成电子通过某个点的速度。用安培（Ampere，Amp）做电流的单位可以更好地描述它的强度（intensity）。安培数越大，意味着通过导线的电子流动得越快。比方说有这样两个电器，一个是台灯，另一个是吸尘器的发动机。点亮灯泡只需要很小的电流，它的安培数比较低，而开动吸尘器则需要较大的电流，它的安培数比较高。有的时候电器运作所需的电流数又称为 draw（牵引，引出），这指的是该电器为了运作必须有多大的电流经过它。

**学习指南：**通过某个点的电流为 1 安培，意味着每秒钟大约有 $6.24\times10^{18}$ 个（6 240 000 000 000 000 000）电子通过该点。

**提示：**在房屋中同时使用多台高安培数（或者说大电流）的设备可能会令断路器跳闸，从而导致停电。如果同时运作的电器所需使用的电流总量超出了电路的负荷能力，那么断路器就会跳闸。比方说，标准的家用断路器所允许的最大电流是 15A，如果同时使用一台 10A 的吹风机与一台 8A 的吸尘器，那么总电流就达到 18A，超出了 15A 的最大限制，因此会跳闸。

**电压**指的是两个点之间的电位差，有的时候也叫作**电势差**（electrical potential difference），它的单位是**伏特**（Volt）。电流与电压之间的区别可以通过比喻来说明。下面就用水流及水压（water pressure）举例。

假设把软管连到水龙头上并且只把龙头开一半。此时，软管中的任何一点都会有一定量的水从这里流过（这个水流就好比电学中的电流）。如果软管的开口是标准的 1 英寸[⊖]规格，那么此时流出的水并没有太大的力道，因为开口太大了，不会形成很强的水压（这个水压就好比电学中的电压）。这种水流比较弱，无法用来强力清洁。

不过，在这个环境中有两个因素可以改变，第一是开口的大小。如果拿拇指把大部分管口堵住，水流就会变强，由于开口面积小了，因此水压（相当于电压）增大，然而这同时也会令水流变小，因为流出管口的水量小了。现在的水流会比原来稍稍强一些。还有一个可变的因素在于龙头，如果既用拇指堵住大部分开口（相当于增加电压）又把龙头完全拧开（相当于增大电流），那么水流就会强得多。

接下来要讲的物理量是**功率**，它是电流所做的总功，单位是**瓦特**（Watt，简称瓦）。仍然以刚才的软管为例，我们发现：增加其中一个物理量或是同时增加两个物理量可以令电功率变大，这说明电功率与电压及电流有着直接的正向关系。这种关系可以用下面这个代数公式来表达：

⊖ 1 英寸≈ 0.0254 米。

$$P=I*E$$

只要知道其中两个量，就可以通过基本的数学运算求出第三个量。比方说，知道某台电器的功率以及插座的输出电压就可以算出电流，从而知道使用这台电器会不会导致跳闸。如果空调标称的功率是 1560W，电压是 120V，那么重新排列上述公式就可以算出空调运作所需的电流。

| $P=I*E$ | $I=\frac{P}{E}$ | $E=\frac{P}{I}$ |
|---|---|---|

$$I=\frac{1560\text{W}}{120\text{V}} \qquad 13\text{A}=\frac{1560\text{W}}{120\text{V}}$$

最后要讲的物理量是**电阻**，它是电流因为材料、环境或元件等因素而受到的阻碍。由于现实中的导体都不完美，因此，电能总会在传输过程中有所流失。不过，某些情况下可能会人为地提升电阻，以确保某个元件的电能可以达到必要的水平。电阻的单位是欧姆（Ohm，简称欧）。**欧姆定律**表述了电阻与另外两个物理量之间的关系，通过该定律可以看出：在电阻（R）不变的情况下，电压（E）与电流（I）成正比。

| $E=I*R$ | $I=\frac{E}{R}$ | $R=\frac{E}{I}$ |
|---|---|---|

利用欧姆定律以及刚才讲的与电功率、电流及电压有关的公式，我们可以根据 P、I、E、R 这四个物理量中的两个求解出另外两个。下面举几个例子。

假设 $P$=1600W，$E$=110V，求 $I$ 及 $R$：

$$I=\frac{1600\text{W}}{110\text{V}} \qquad 14.54\text{A}=\frac{1600\text{W}}{110\text{V}} \qquad R=\frac{110\text{V}}{14.54\text{A}} \qquad 7.57\Omega=\frac{110\text{V}}{14.54\text{A}}$$

假设 $I$=30A，$R$=10Ω，求 $E$ 及 $P$：

$E$=30A*10Ω　　300V=30A*10Ω　　$P$=30A*300V　　9000W=30A*300V

## 12.2.2 交流与直流

理解了这几个基本的物理量之后，我们来重新讲解电流，并且开始讨论与元件和设备有关的一些内容。电通常以两种形式输送，这就是**交流**（alternating current，AC）和**直流**（direct current，DC），分别参见图 12-1 与图 12-2。

墙上的插座输出的一般是交流电，它的电流方向总是不停地改变，如图 12-1 所示。反转的频率（交替的频率）以赫兹（Hertz，简称 Hz，赫）表示。1 Hz 意味着每秒振荡 1 次。美国的标准交流电是 60 Hz，这意味着电流每秒要从 +120V 到 −120V 变化 60 次。

大多数电子设备所用的电是直流（DC）电，它通常用正电压来表示，例如 CD-ROM 光驱需要 +5V 的 DC，LED 灯需要 +2.7V 的 DC。虽然当今的电子设备基本上要使用直流电，但插座输出的却是交流电，既然这样，那为什么还能把电脑与墙上的插座相连呢？这两者不仅类型不同（一个是直流，一个是交流），而且电压也不一样，交流电通常是 110 至 220 伏，这会把电脑元件烧坏。下一节讲解电子元件的时候就要解释如何应对这种电流不匹配的现象。

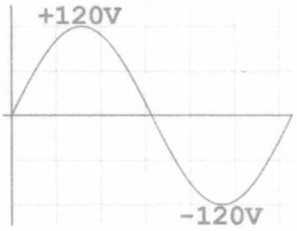



图 12-1　交流（AC）




图 12-2　直流（DC）

**学习指南：**交流电之所以成为家庭用电的标准形式，是因为它的传输效率较高（三相交流电采用较小的中性导线，减轻了发电机与电动机的摩擦，并且也无须设计复杂的电动机）而成本较低。这也印证了推崇交流电的威斯汀豪斯在当年的电流传输形式之争中确实胜过了推崇直流电的爱迪生。

## 12.3　电子元件

### 12.3.1　供电设备

前面说过，要把插座输出的 120V 交流电输送给需要在适当的直流电压（例如 +3.3V、+5V 等）下工作的各种计算机设备，就必须改变电流的形式。这项电流处理工作由两个电子元件负责，它们都位于计算机的电源供应器中，无论是台式机的电源盒还是笔记本电脑用的类似砖块的电源都具备这样两个元件。这就是变压器（transformer）与整流器（rectifier）。

**变压器**，是一种通过电磁感应（electromagnetic induction）来提升或降低交流电压的元件，它把 120V 的电压降低到计算机内部元件能够承受的程度。电源供应器通常包含两个或三个变压器，用来降低各种信号的电压，使得计算机内部的元件都能够得到各自所要求的电力。

电压降低后，需要用另一个元件将其从交流变为直流，这个元件就是**整流器**，它使得电流只朝着一个方向流动（前面讲过，交流电有正、负两个方向）。除此之外它还有另一项职能，就是确保电流不会反向流动。假如反向流动，电流就会流回到变压器，如果计算机由于故障而引发浪涌，还会出现更加危险的情况，即电流会把插座烧坏。稍后我们会讲到整流器如何控制电流安全。

对于计算机的供电来说，还有一种电子元件也应该讨论一下。有人想在开车的时候把电子设备接到点烟器上。由于点烟器输出的是直流电，因此就算你把笔记本电源的二项或三项插头插到转换插座并把转换插座接到点烟器，也还是不行，因为笔记本电源的输入端必须是交流电，而汽车的点烟器输出的却是直流电。在这种情况下，你需要使用图 12-3 所演示的这种**逆变器**。它的功能与变压器相反，它是把直流电转换成交流电。有了逆变器，点烟器所给出的直流电就可以变为交流电，从而输入笔记本的电源，而电源又（通过变压器）把交流电转回具备适当电压的直流电，以供笔记本正常运作。

**提示：**逆变器的功率和电压各有不同。购买逆变器的时候，要注意看它的功率与电压能否带动你的电脑。

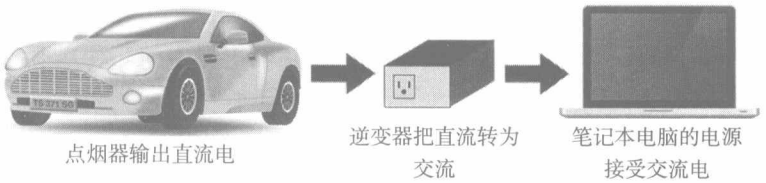



图 12-3　逆变器的用法

电脑这样的设备确实要有充足的电力才能运作，因此它们的电源要接在插座上，然而还有很多电子设备，例如手机、MP3、手电筒等，不需要那么大的电力，这种设备一般由电池供电。这是一种带有正极和负极，并通过外部连线来给设备供电的元件。电池的电来自其中的电化学池内所发生的反应。

由于电池的形体比较小，因此单靠一块电池通常很难形成设备运作所需的电压。有些电器确实可以只用一节电池，但是由于耗电量较大，因此电池很快就没电了。这两个问题可以用各种办法来解决，这些办法都涉及同时使用多节电池。如果电器所需的电压比单个电池所能提供的电压高，那么可以把许多节电池串联起来，也就是把前一个电池的负极与后一个电池的正极相接，这样得到的电压是这些电池的电压之和，而电流则保持不变。

图 12-4 演示了怎样给标准的电视遥控器供电。它需要有 +6V 的直流电及 1800mA 的电流才能运作。每一节电池的电流是 1800mA，直流电压是 +1.5V，因此，将 4 节这样的电池串联起来就可以形成 +6V 的直流电压，并保持 1800mA 不变。

**提示：** mA 是 milliamp 的缩写，意思是毫安。这一章还会用到其他一些缩写，这些缩写都遵循国际单位制的词头形式，例如 milli 前缀表示毫（千分之一），mega 前缀表示兆（百万），giga 前缀表示千兆（一千个百万；十亿）。

**提示：** 如果想把连接电池所用的导线长度缩至最短，那么可以交错地连接，也就是将第 2 节电池上下颠倒，使得第一节电池的负极直接与位于它右侧的第二节电池的正极相接，而不用像图 12-4 那样把导线拉到右上角去。你可能见过需要这样来安装电池的设备。




图 12-4　AA 电池的串联




图 12-5　AA 电池的并联

有些电器仅需一节电池就能带动，但是只能使用很短的一段时间。如果有足够的空间可以放置电池，那么设计者可能会采用多节电池并联的方案，从而在电压不变的前提下令总电流等于各电池的电流之和（这与串联恰好构成对照）。并联的时候电流是同时通过这些电池

的，如图 12-5 所示，而不是像串联那样依次通过。

接下来，我们要在这一节的其他几个小节中分别讲解电器中的几种基本元件。

### 12.3.2　电阻器

前面介绍欧姆定律的时候提到过电阻。由于导线与电容都不可能用完全理想的材料来制作，因此，任何电路中都有电阻。某些电路还需要进一步降低电流，因此会人为地增加电阻。这种减缓电子流动的元件叫作**电阻器**（resistor），如图 12-6 所示。



从左至右的色环，分别为棕色、绿色、红色、金色

图 12-6　电阻器

电阻器的电阻值各有不同。这些值以欧姆为单位，并通过电阻器上的色环（color band）来表示。典型的电阻器标有 4 个色环，其电阻值要按照从左至右的方向解读（最右侧的色环一般是金色或银色）。前两个色环表示电阻基本值的十位数与个位数，第三个是倍数，也就是 10 的某个整数次方，第四个是精确度或误差。表 12-2 列出了这些色环的含义。

现在举两个例子。首先看图 12-6 中的电阻器，它有四个色环，分别是棕色、绿色、红色、金色。棕色对应的数位是 1，绿色对应的数位是 5，因此这两个色环合起来表示 15，红色对应的倍数是 $10^2$，因此电阻值就是 $15 \times 10^2$=1500 Ω。第四个色环是金色，这表示实际的电阻值与这个电阻器标称的电阻值之间可能有 −5% 到 +5% 的差距。

再看另一个例子：假设某个电阻器只有三个色环，分别是红色、黑色、橙色。红色对应的数位是 2，黑色是 0，因此这两个色环合起来表示 20，橙色对应的倍数是 $10^3$，所以，电阻值是 $20 \times 10^3$=20000 Ω。由于没有第四个色环，因此，它的误差范围是 ±20%。

**学习指南：**与电池一样，电阻或其他元件之间也可以串联或并联。串联时电流不变，**电压等于各电阻的电压之和**，并联时电压不变，**电流等于各电阻的电流之和**。将这两个规律写成算式，就是 $R_T I=R_1 I+R_2 I+R_3 I+R_4 I$ 与 $E/R_T=E/R_1+E/R_2+E/R_3+E/R_4$，分别化简，即可得出：串联的总电阻 $R_T=R_1+R_2+R_3+R_4$，并联的总电阻的倒数 $1/R_T=1/R_1+1/R_2+1/R_3+1/R_4$。

表 12-2　电阻器的色环所表示的含义

| 颜　色 | 色　环 | | |
|---|---|---|---|
| | 第 1 个色环 | 第 2 个色环 | 第 3 个色环 |
| 黑（Black） | 0 | 0 | $10^0$ |
| 棕（Brown） | 1 | 1 | $10^1$ |
| 红（Red） | 2 | 2 | $10^2$ |
| 橙（Orange） | 3 | 3 | $10^3$ |
| 黄（Yellow） | 4 | 4 | $10^4$ |
| 绿（Green） | 5 | 5 | $10^5$ |
| 蓝（Blue） | 6 | 6 | $10^6$ |
| 紫（Violet） | 7 | 7 | $10^7$ |
| 灰（Gray） | 8 | 8 | $10^8$ |
| 白（White） | 9 | 9 | – |

（续）

| 颜　色 | 第 4 个色环 |
| --- | --- |
| 金（Gold） | ±5% |
| 银（Silver） | ±10% |
| 无（None） | ±20% |

还有一种电阻器是可变电阻器，它也经常称为电位器，这种电阻器的电阻是可以改变的。你不妨将其理解为一种可以来回调整的电阻器，其电阻会随着调整到的位置发生变化。汽车收音机的音量控制按钮就属于电位器。

### 12.3.3 二极管

讲整流器的时候笔者说过，它除了能转换电流之外，还能防止电流回流。这种控制电流方向以防其回流的元件就是**二极管**（diode），如图 12-7 所示。



图 12-7　二极管

二极管的原理是：它在其中一个方向上具备低电导（conductance）或零电导，而在另外一个方向上则具备高电导或无穷电导。也可以换一种说法，从电阻的角度来描述：它在其中一个方向上具备高电阻或无穷电阻，而在另外一个方向上则具备低电阻或零电阻。二极管上的灰带用来表示电流的方向，以图 12-7 为例，电流可以从这个二极管左侧到达右侧。有一种常见的二极管叫作**发光二极管**（light-emitting diode，LED），如果按照正确的方向来连接它就可以发光。

### 12.3.4 电容器

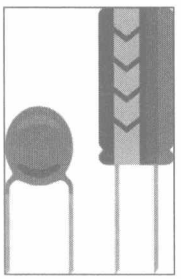


图 12-8　电容器

接下来要讲**电容器**（capacitor），这是一种临时存储电荷的元件，图 12-8 展示了两种电容器。

电路中的电容器通常用来过滤电流。它的另一项功能是用来确保：无论电力来源是否稳定（例如是否会突降或激增），电路中的下一个元件总能获得适当的电压。由于很多电子元件都对功率波动较为敏感，因此需要由电容器来减缓这种波动，以延长这些元件的寿命。只要电容器中还有电荷（也就是电子还未耗尽），就能够起到稳定功率的作用。电容器可以想象成充电电池，电量一旦耗尽就会失效，必须再度充电才能恢复效用。然而这种比喻也不完全贴切，因为在电容器没有电的情况下，电流照样能够通过它，只是不如电容器完全充好电时那样稳定，而电池没电的时候电流则无法通过。

由于不同的电子元件需要的电压不一样，因此电容器也分为很多种。它们要按照存储电子的能力，也就是**电容量**（capacitance）来划分。电容量的单位是**法拉**（Farad），简称法（F）。1F 的电容器可以存放 $6.24 \times 10^{18}$ 个电子，当今的大多数计算机元件都用不到这么多电量，因此，在构建电路时经常用的是皮法（picofarad，pF）、纳法（nanofarad，nF）或微法（microfarad，μF）等较小的单位。

### 12.3.5 晶体管

最后一个要讲的基本电子元件叫作**晶体管**（transistor），下一节的内容——乃至整个现代计算机领域——都要以此为基础。这是一种可以当作信号放大器或开关来用的元件，当代

的电子设备都在这种基本元件的基础上构建而成。目前的计算机所使用的晶体管是由硅制作的，而不是像早前的晶体管那样由锗制作。不过，在设计电路的时候还可以选用不同类型的晶体管。晶体管的三个极（terminal）叫作基极（base）、发射极（emitter）、集电极（collector），如图 12-9 所示。

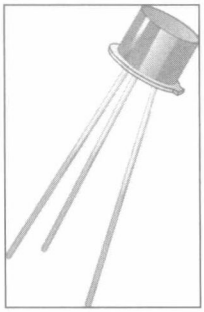


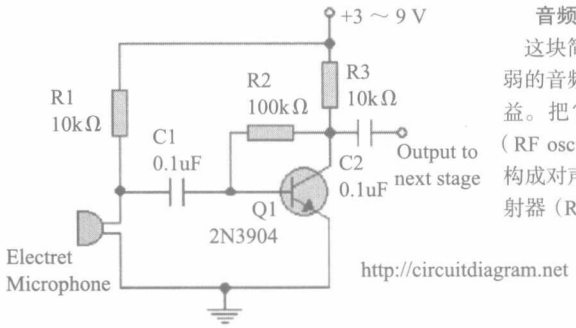
图 12-9　晶体管

给基极通电可以令发射极与集电极之间形成通路，从而使电流得以通过。放大程度与施加给基极的电压成正比。晶体管有一项用途软件工程师应该很熟悉，就是用来表示只有关与开这两种取值的二元信息。充当开关的晶体管可以按不同方式组合起来构成逻辑门，以便执行算术运算及布尔运算（例如第 1 章及第 2 章所讲的 AND、OR、NOT 等）。

> **提示：**晶体管的发射极与集电极之间有时导电有时不导电，这种特性，就是**半导体**（semiconductor，semi- 前缀表示“半”）一词的来源。硅以及早前充当晶体管材料的锗，都具备这种部分导电的能力，或者说具备可以在开与关这两种状态之间切换的能力。

## 12.4　集成电路

前面讲的那些电子元件合起来可以构成电路，以完成某种任务。构建电路的第一步是设计。电路设计通常用软件来做，这些软件用图 12-10 这样的**电路图**（schematic）表示电路。图中的 R1、R2、R3 是电阻器，C1 与 C2 是电容器（用两条平行的竖线表示），Q1（橙色的圆圈）是晶体管。电阻器旁边写有欧姆数，电容器旁边写有法拉数，晶体管旁边写有型号，用以表示其功能。




**音频前置放大器范例**

这块简单的电路能够给较弱的音频信号提供很好的增益。把它放在射频振荡器（RF oscillator）前端，可以构成对声音很敏感的射频发射器（RF transmitter）

图 12-10　前置放大器的电路

做好电路图之后，工程师可能会用图 12-11 的**面包板**构建一个极其简单的版本。这种板子上横竖都排列有很多插孔，使得工程师可以用导线将电子元件连接起来，而无须借助电焊。这种实验用的电路板就是专门用来构建临时电路的，以便随时添加、移除或移动元件。

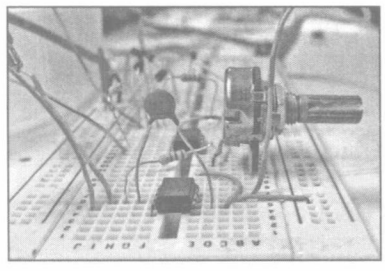


图 12-11　面包板

电路构建出来并通过测试之后需要以更固定的形式生产。很多年前生产出来的电路，其实与图 12-11 中的样子并没有太大区别。导线与元件虽然焊接起来了，但

仍然暴露在外面。电子元件越做越小，但是这个暴露在外的问题依然得不到解决。1958 年，在 Texas Instruments（德州仪器）工作的杰克·基尔比（Jack Kilby，1923 ～ 2005）提出了一种解决办法，与此同时 Fairchild Semiconductor（仙童半导体，也称为飞兆半导体、快捷半导体）的罗伯特·诺伊斯（Robert Noyce，1927 ～ 1990，后来与人创办了 Intel 公司）也拿出了一种方案。Kilby 把各种材料都构建在一块完整的电路上，而不是手工将每个元件连接起来。今天所说的**集成电路**（integrated circuit）究竟是由 Kilby 还是由 Noyce 首创，还在激烈的争论中。图 12-12 是 Kilby 集成电路。

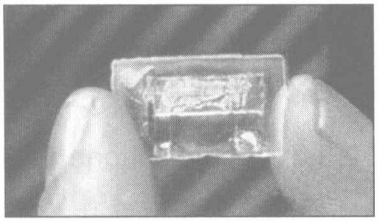


图 12-12 Kilby 集成电路（此图在德州仪器公司的许可下使用）

集成电路是电路发展史上的里程碑，它首次把元件与导线集成到了一块电路中，但仍然有一些问题要解决，例如导线会凸起而且容易损坏。这些问题很快就在 1959 年由 Fairchild Semiconductor 的 Jean Hoerni（1924 ～ 1997）解决了。他发明了**平面工艺**（planar process），这种工艺把电路当成带有绝缘区（insulated area）与掺杂区（doped area）的二维平面，从而使导电材料可以放在其内部。这种工艺还使机器能够采用分层的办法把整块电路板自动制作出来，从而不会令导线或元件过于突出。当今的电器所用的电路就是采用这种平面工艺制作出来的。图 12-13 是一块印刷电路板（printed circuit board，PCB）。

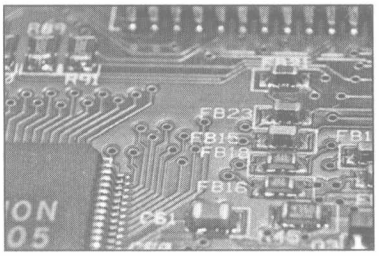


图 12-13 印刷电路板

前面提到的这些成就，再加上其他许多人的努力，使得我们今天能够用上功能强大、形体小巧而且安全可靠的各种电子设备。

## 12.5 常见的实现产品

现在有越来越多的设备是用刚才讨论过的技术制造而成的，其中某些设备是专门给爱好者玩的，还有一些则是大家经常用到的，有些设备是奢侈品，也有些是必需品。最后这一节会简单地介绍几种常见的产品，它们之所以能够问世，正得益于早前讲到的那些元件与技术。

### 12.5.1 计算机处理器

Intel 公司（Intel 这个词是由 Integrated 和 Electronics 混合而来的）最早生产的处理器每次只能操作 4 位数据，其时钟频率仅为 740 kHz（kilohertz，千赫）。然而到了笔者写这本书的时候，它已经推出了 Intel Core i7 这样强大的处理器（参见图 12-14），它能够操作 64 位（乃至更多位）的数据，其时钟频率超过 4 GHz（gigahertz），也就是说每秒钟能执行四十多亿条指令。

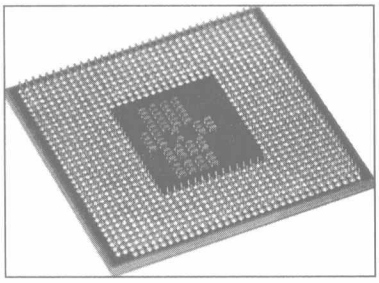


图 12-14 Intel Core i7

### 12.5.2 Raspberry Pi

图 12-15 中的 Raspberry Pi（树莓派）是一台小巧的电脑，不过它的功能却很完备。虽

然只有一副扑克那样大，但是各种接口都很齐全，例如 HDMI、USB、Ethernet（以太网）、WiFi、Bluetooth（蓝牙）、音频，以及 GPIO（通用输入 / 输出，General Purpose Input/Output）针脚等，这些接口可以与外部设备相连接。对于剪辑电影或渲染 3D 动画等工作来说 Raspberry Pi 或许显得有些吃力，然而用它来玩怀旧游戏已经足够了，此外它还可以当遥控器控制家中的“智能”设备，或是给信息安全工程师充当测试机，以进行网络渗透测试。Raspberry Pi 是电脑迷的理想机型，因为它价格低廉而且易于扩展。

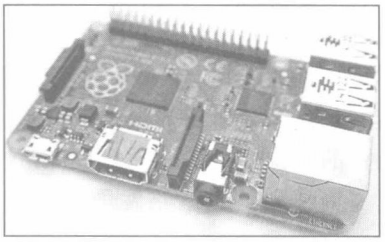


图 12-15　Raspberry Pi

### 12.5.3　Apple iPhone 与 Apple Watch

不知你有没有发现，笔者其实是很喜欢用 Apple 产品的人，所以在这里自然要用图 12-16 中的 Apple iPhone 和 Apple Watch 来说明智能手机与其他“智能设备”。

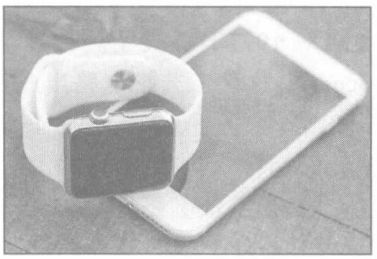


图 12-16　Apple iPhone 与 Apple Watch

现在大家使用的这些电子设备，其处理能力已经超过当年辅助人类登月的那些计算机了，这不得不叫人赞叹。由于这是一本讲汇编语言的书，因此，这里给出一个与汇编有关的典故：图 12-17 描述的是 Apollo（阿波罗）计划的主软件工程师 Margaret Hamilton（玛格丽特·汉密尔顿）站在一大叠打印出来的汇编代码旁边，那些代码是给 NASA 的 Apollo Guidance Computer（阿波罗导航计算机）写的。

现在的计算机很多都可以放入口袋中或是戴在身上，例如 Apple 的 iPhone 以及各种 Android 手机，Apple Watch、Samsung Gear、Motorola Moto 360 等各种智能手表，以及 Google Glass 与 FitBit 公司生产的其他智能设备等。这些设备能够诞生，是因为有了集成电路技术且业界能够不断缩小电子元件的体积并提高其性能。



图 12-17　为 NASA 的 Apollo guidance computer 而写的汇编代码

## 12.6　小结

最后这一章讨论了电学的基础知识，我们从基本的电学物理量讲起，逐渐谈到当代的电子设备。现在你应该熟悉电与各种电子元件的特性了，这些元件都是操控或利用电的某些性质来实现其功能的。除了定义基本的电学物理量并给出公式之外，我们还形象地展示了某些电子元件。由这些元件所形成的电路可以有无穷的变化，从而能够用来做很多事情，本章最后展示的那几种流行设备正说明了这一点。恭喜你读完全书，愿数位的力量与你同在。

## 12.7 关键术语

alternating current (AC)（交流）
ampere（安培）
anode（阳极）
battery（电池）
breadboard（面包板；实验电路板）
capacitance（电容量）
capacitor（电容器）
cathode（阴极）
current（电流）
diode（二极管；二极体）
direct current (DC)（直流）
draw（牵引）
farads (F)（法拉（法））
Hertz (Hz)（赫兹（赫））
integrated circuit（集成电路）
inverter（逆变器）
light-emitting diode (LED)（发光二极管）
Ohm's Law（欧姆定律）
ohms（欧姆；欧）
parallel（并联）
planar process（平面工艺；平面处理技术）
potentiometer（电位器）
power（功率）
printed circuit board (PCB)(打印电路板）
rectifier（整流器）
resistance（电阻）
resistor（电阻器）
schematic（电路图）
semiconductor（半导体）
series（串联）
transformer（变压器）
transistor（晶体管）
voltage（电压）
volts（伏特；伏）
watts（瓦特；瓦）

## 12.8 习题

### 填空题

1. 电阻值的单位欧姆可以用希腊字母_____表示。
2. 两点之间的电势差或电位差叫作_____。
3. 电脑运转所需的电流通常比手电筒_____。
4. 瓦特是用来度量电_____的单位。
5. 根据电功率的公式：知道功率与电压，可以求出_____。
6. 由于现实中的导体不是理想导体，因此它们会增大整个电路的_____。
7. 根据欧姆定律及电功率的公式：知道电阻及电流，可以求出_____及_____。
8. 墙面插座所供应的电一般是_____流电。
9. 与直流电不同，交流电的电流方向总是按照某种频次而发生变化，用来衡量这种变化速度的物理量叫作_____。
10. _____的另一项功能是防止电流回流。
11. 电池所提供的电是_____流电。
12. _____是一种能够减缓电子流动速度的元件。
13. 电容量的标准单位是_____，然而大部分电容器的电容量都远远达不到 1 个这样的单位。
14. _____这种电子元件，早前是用锗做的，后来改用硅做。
15. 打印电路板（PCB）之所以能产生，是因为先有了_____技术。

### 判断题

1. 变压器能把电流从交流的形式变成某些电子设备需要的直流形式。
2. 电流只能从一个方向通过二极管。

3. 如果电子设备需要的电压较大而对电流没有太大要求，那么可以把多节电池串联起来供电。
4. 电阻器上如果只有 3 条色环，那么它的误差率就是 ±5%。
5. 可以随意插拔各种电子元件以供电路工程师来测试电路的设备，叫作集成电路板。

## 12.9　作业

### 12.9.1　可别跳闸了！

假如你在家办公而且要用到下面这四个电器，那么同时打开它们会不会让 15A 的电闸跳闸？接下来请列出不会跳闸的所有组合方式，并写出每种情况下的总电流。

- 空调（1000 W）
- 电脑（500 W）
- 台灯（5 W）
- 碎纸机（5 A）

### 12.9.2　通过电阻降低电流

假如你要构建一块小型电路。电源是 +5V、500 mA 的直流电，而电路中的元件最大只能承受 +5V、200 mA 的直流电。由于元件特别敏感，因此必须选用欧姆数合适且误差尽可能小的电阻器来降低流经元件的电流强度。为此你应该选用什么样的电阻器呢？（请用色环标出这种电阻器的规格。）

### 12.9.3　研究形体较小的计算机

利用网络资源与其他资源选出一款体积较小的计算机或单片机，例如 Raspberry Pi 或 Arduino 开发板，并研究一下怎样有创意地使用这种计算机。针对该计算机写一篇 2 至 3 页的文章，在你所能找到的使用方式中描述 3 种最有意义的用法，此外，再叙述一种你自己设计出来的用法。

# 附录简介

## 各附录的用途

欢迎查阅本书的附录，这些附录会对正文所提到的许多话题给出实用的指导说明。

**附录概述**

- 附录 A：各种汇编语法之间的对应关系
- 附录 B：配置开发环境
- 附录 C：反汇编
- 附录 D：在命令行界面中，用 GDB 工具调试汇编程序
- 附录 E：给汇编代码及 C++ 代码做链接
- 附录 F：函数与栈
- 附录 G：使用 CPUID
- 附录 H：对 ASCII 字符做十进制算术运算
- 附录 I：编译器内部函数

笔者决定把这些目录做得特别实用，令读者能够直接按照其中的步骤来操作，例如附录 B、附录 D 及附录 E 就是这样。附录 A 是编写汇编程序时的好帮手，笔者自己也经常查阅。其他附录不仅包含辅助信息，而且会详细叙述某些编程操作的步骤，使你能够更好地理解本书正文中的内容。附录中还包含一些资源与链接，有助于你研究特定的话题。

## 怎样寻找未收入附录中的内容

很容易就能在其他地方找到的那些内容没有收在附录中。例如 x86、x86_64、x87、SIMD、AVX 等指令集就完全可以在 Intel 与 AMD 的规范文档以及网上的各种资源中找到。ASCII 码表也很容易搜索到，因此，笔者在附录中不会直接给出这些资源，而是会列出一些链接，使你能够由此发现这些重要的文档。此外，正文每一章的网络资源中也提供了许多资源链接。

在指令集这一方面，目前最好的资源是《Intel 64 and IA-32 Software Developer's Manual》的第 1 卷第 5 章，整本手册中其实都包含与指令有关的信息，但从它的第 1 卷第 5 章开始看起可以收到很好的效果。笔者建议你去 Intel 网站下载该手册：

- http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

AMD 手册可以在 AMD 开发者网站上找到：

- http://developer.amd.com/resources/developer-guides-manuals/

Agner Fog 写了一些很好的技术文档，其中谈论了指令、调用约定以及优化技术。

- http://www.agner.org/optimize/#manuals

如果你不喜欢看 PDF，而是更愿意上网查阅各种指令的细节信息，那么可以参考这几份指令索引表。

- http://www.felixcloutier.com/x86/

- http://x86.renejeschke.de
- https://en.wikipedia.org/wiki/X86_instruction_listings
- http://www.nasm.us/doc/nasmdocb.html

ABI 规范文档也很有用。下面给出几个较好的链接：

- https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI
- https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/LowLevelABI/000-Introduction/introduction.html

如果要了解某种汇编器所支持的指令及语法，那么应该去查询该汇编器的参考手册。

- https://sourceware.org/binutils/docs/as/（GAS 汇编器）
- http://www.nasm.us/doc/（NASM 汇编器）
- http://www.nasm.us/docs.php（NASM 汇编器）
- https://msdn.microsoft.com/en-us/library/afzk3475.aspx（MASM 汇编器）
- https://developer.apple.com/library/mac/documentation/DeveloperTools/Reference/Assembler/000-Introduction/introduction.html（虽然已经过时，但仍可以参考）

希望这些附录能够对你有所帮助。如果你觉得还应该增设其他附录，请告诉笔者。

# 各种汇编语法之间的对应关系

## 本附录的用途

这篇附录会列出 3 种常见汇编器的通用规则，并通过表格在这三套汇编语法之间对比。

## 各汇编器的规则

- GAS 代码中的寄存器要加上 % 前缀。
- GAS 代码中的立即值要加上 $ 前缀。
- GAS 代码还用 $ 前缀来取变量的地址。
- MASM 与 NASM 代码用 $ 表示**当前位置计数器**（current location counter），GAS 用圆点（.）来表示它。
- GAS 代码要先写来源操作数，后写目标操作数。
- MASM 及 NASM 代码要先写目标操作数，后写来源操作数。
- GAS 代码在指令后面用 b、w、l 及 q 等后缀来表示该指令所涉及的操作数有多大。
- GAS 与 NASM 的标识符区分大小写。
- MASM 的标识符默认不区分大小写，但是可以添加 `option casemap:none` 命令来区分（这一般添加在 .MODEL 命令之后）。
- NASM 代码用不加括号的 *ST0*、*ST1* 等写法来表示 FPU 栈寄存器。
- GAS/MASM 通常用 *%st(1)/ST(1)*、*%st(2)/ST(2)* 这样带括号的写法来表示 FPU 栈寄存器。
- GAS 使用 .equ 命令将表达式的值设定给某个符号，NASM 用的是 EQU 命令，MASM 可以用 = 号，也可以用 EQU 命令。
- 这 3 种汇编器都用单引号或双引号把字符串括起来。
- MASM 的指令会自动根据某些因素（例如数据的大小）做出相应的处理，因此，有的时候很难看清楚这样一条指令究竟会如何执行。
- NASM 通常不要求开发者用表示数据大小的命令来修饰源操作数，但如果用了也是可以的，而目标操作数的大小则必须明确，不然就要通过命令来指明。

```
mov eax, [test]          ; source size is not required
mov eax, DWORD [test]    ; but can be used
mov DWORD [test], eax    ; required for destination
```

| 操　作 | GAS | NASM | MASM |
|---|---|---|---|
| 清除寄存器（以 *eax* 为例） | `xorl %eax, %eax` | `xor eax, eax` | |
| 把 *eax* 的内容移动（或者说复制）到 *esi* | `movl %eax, %esi` | `mov esi, eax` | |
| 把 *ax* 的内容移动到 *si* | `movw %ax, %si` | `mov si, ax` | |

（续）

| 操　作 | GAS | NASM | MASM |
|---|---|---|---|
| 把字节型的立即值 4 移动到 *al* | `movb $4, %al` | `mov al, 4` | |
| 把地址 0xf 移动到 *eax* | `movl 0x0f, %eax` | `mov eax, [0x0f]` | `mov eax, ds:[0fh]` |
| 把 temp 变量的内容移入 *eax* | `movl temp, %eax` | `mov eax, DWORD [temp]` | `mov eax, temp` |
| 通过 MOV 指令把 temp 变量的地址移入 *eax* | `movl $temp, %eax` | `mov eax, temp` | `mov eax, OFFSET temp` |
| 通过 LEA 指令把 temp 变量的地址载入 *eax*（32 位环境） | `leal temp, %eax` | `lea eax, [temp]` | `lea eax, temp` |
| 通过 LEA 指令把 temp 变量的地址载入 *eax*（64 位环境） | `leaq temp(%rip), %rax` | `lea rax, [rel temp]` | `lea rax, temp` |
| 把 *eax* 的内容移入 temp 变量 | `movl %eax, temp` | `mov DWORD [temp], eax` | `mov temp, eax` |
| 把字节型的立即值 2 移入 temp 变量 | `movl $2, temp` | `mov BYTE [temp], 2` | `mov temp, 2` |
| 把字节型的立即值 2 移入由 *eax* 所指向的内存 | `movb $2, (%eax)` | `mov BYTE [eax], 2` | `mov BYTE PTR [eax], 2` |
| 把单字（word）型的立即值 4 移入由 *eax* 所指向的内存 | `movw $4, (%eax)` | `mov WORD [eax], 4` | `mov WORD PTR [eax], 4` |
| 把双字型的立即值 6 移入由 *eax* 所指向的内存 | `movl $6, (%eax)` | `mov DWORD [eax], 6` | `mov DWORD PTR [eax], 6` |
| include 命令的写法，该命令用来把某份文件包含到当前文件中 | `.include "file.ext"` | `%include "file.ext"` | `INCLUDE file.ext` |
| 标识符的写法 | `identifier: type value` | | `identifier type value` |
| 通过当前位置计数器（也就是紧跟在数组声明之后的那行代码所在的地址）来获取数组所占据的字节数 | `aSize: .long (. - array)` | `aSize: EQU ($ - array)` | `aSize = ($ - array)` |
| 用 EQU 命令创建符号，并使用该符号 | `.equ temp, (2*6/3)`<br>`mov $temp, %eax` | `temp: EQU (2*6/3)`<br>`mov eax, temp` | `temp EQU (2*6/3)`<br>`mov eax, temp` |
| 在内存中预留 64 个字节的空间 | `.space 64` | `resb 64` | `db 64 DUP (?)` |
| 创建不经初始化的 32 位变量 temp | `.lcomm temp, 4` | `temp: resd 1` | `temp DWORD ?` |
| 创建 32 位变量 temp，并将其初始值设置成 5 | `temp: .long 5` | `temp: dd 5` | `temp DWORD 5` |
| 创建元素为 32 位值的数组 | `temp: .long 5, 10, 15` | `temp: dd, 5, 10, 15` | `temp DWORD 5, 10, 15` |
| 创建内容为 Hello, World 的字符串（范例代码有可能印刷成多行，但实际是写在一行中的） | `identifier: .ascii "Hello, World"` | `identifier: db 'Hello, World'` | `identifier BYTE "Hello, World"` |
| 创建带有新行符及 null 字符的 Hello, World 字符串（范例代码有可能印刷成多行，但实际是写在一行中的 | `identifier: .asciz "Hello, World\n"` | `identifier: db 'Hello, World', 10, 0` | `identifier BYTE "Hello, World", 10, 0` |

（续）

| 操　作 | GAS | NASM | MASM |
| --- | --- | --- | --- |
| 函数的结构 | identifier:<br>...<br>ret | | identifier PROC<br>...<br>ret<br>identifier ENDP |
| 程序段 | .data<br>.bss<br>.text | SECTION .data<br>SECTION .bss<br>SECTION .text | .data<br>.code |
| 数据类型 | .byte<br>.word<br>.long<br>.quad | db　BYTE<br>dw　WORD<br>dd　DWORD<br>dq　QWORD | |
| 声明类型相同的多份数据（范例代码有可能印刷成多行，但实际是写在一行中的） | identifier: .fill count, size, value | identifier: TIMES count type value | identifier type count DUP (value) |
| 定义宏（范例代码有可能印刷成多行，但实际是写在一行中的） | .macro identifier arg1, arg2...<br>args referenced as \arg1<br>.endm | %macro identifier argcount args<br>referenced as [%1]<br>%endmacro | identifier MACRO arg1, arg2...<br>args referenced as arg1<br>ENDM |
| 使用宏 | identifier param1, param2, etc... | | |
| （单行的）注释 | # this is a comment | ; this is a comment | |
| 在 32 位环境下从 _main 中退出所需编写的例行代码 | # for GAS/Clang on Mac<br>pushl $0<br>subl $4, %esp<br>movl $1, %eax<br>int $0x80<br># for GAS/Clang on Linux<br>mov $1, %eax<br>mov $0, %ebx<br>int $0x80 | ; for NASM on Linux<br>mov eax, 1<br>mov ebx, 0<br>int 80h<br>; for NASM on Mac<br>push DWORD 0<br>sub esp, 4<br>mov eax, 1<br>int 80h | ; before .data segment<br>ExitProcess PROTO, dwExitCode:<br>DWORD<br>; before _main ENDP<br>INVOKE ExitProcess, 0 |
| 在 64 位环境下从 _main 中退出所需编写的例行代码 | # for GAS/Clang on Mac<br>movq $0x2000001, %rax<br>movq $0, %rdi<br>syscall<br># for GAS/Clang on Linux<br>mov $60, %rax<br>xor %rdi, %rdi<br>syscall | ; for NASM on Linux<br>mov rax, 60<br>xor rdi, rdi<br>syscall<br>; for NASM on Mac<br>mov rax, 2000001h<br>mov rdi, 0<br>syscall | ; before .data segment<br>extrn ExitProcess : proc<br>; before _main ENDP<br>mov rcx, 0<br>call ExitProcess |

# 为汇编语言配置开发环境

## 本附录的用途

这篇附录讲述如何配置开发环境，使你能够在该环境中用汇编语言来编写程序。其中，Windows 系统用 Visual Studio 开发环境与 MASM 汇编器来举例，macOS 系统用 Xcode 开发环境来举例（其中的 Clang/LLVM 组合是一种兼容于 GAS 的汇编方案），Linux 系统用 NASM 汇编器与 ld 链接器来举例。笔者提供几种基本的配置方案，你也可以自己研究其他选项的用法。本附录中的范例代码都是 32 位的。

## Windows 系统：Visual Studio（2017 版）开发环境搭配 MASM 汇编器

1. 如果还没有安装 Visual Studio，就先把它装上。

2. 打开 Visual Studio，新建空白的 C++ 项目，把项目名称起好。

3. 创建并开启该项目之后，在 Solution Explorer（解决方案资源管理器）中用鼠标右键单击此项目，选择 Build Dependencies（生成依赖项）菜单中的 Build Customizations（生成自定义）菜单项。

4. 勾选 masm 复选框。单击 OK（确定）按钮。

5. 在 Solution Explorer 中右击 Source Files（源文件），单击 Add（添加）菜单中的 New item ...（新建项…）菜单项。

6. 在对话框的左侧界面中点击 Visual C++ 下面的 Utility（实用工具），然后在中间的界面里选择 Text File(文本文件)。给这个新文件起名，并为它加上 .asm 后缀。单击 Add(添加)按钮。

7. 在 Solution Explorer 中右击刚才创建的 .asm 文件，单击 Properties（属性）菜单项。单击左侧界面中的 General（常规）项。如果右侧界面中的 Item Type（项类型）没有设置，那就把它设为 Microsoft Macro Assembler。单击 OK 按钮。

8. 在 Solution Explorer 中右击项目本身，单击 Properties(属性）菜单项。在左侧界面中，应该可以看到 Microsoft Macro Assembler 这一项。如果看不到，就回到第 2 步重做，如果能看到就继续执行第 9 步。

9. 在 Properties（属性页）对话框的左侧界面中单击 Linker -> System（链接器 -> 系统）项。在右侧界面中配置 SubSystem（子系统），从下拉列表中选取 Windows (/SUBSYSTEM:WINDOWS)。

10. 在 Properties 对话框的左侧界面中单击 Linker -> Advanced（链接器 -> 高级）项。在右侧界面中配置 Entry Point（入口点），你要根据自己在汇编代码中给主函数所起的名字来相应地输入 main 或 _main 等内容。笔者在整本书中都把主函数起名叫作 `_main`。单击 OK 按钮。

11. 在 .asm 文件中编写一段汇编代码。

```
; First Program for MASM

.386
.MODEL FLAT, stdcall
.STACK 4096

ExitProcess PROTO, dwExitCode:DWORD

.DATA
num DWORD 80
sum DWORD ?

.CODE
_main PROC
   mov eax, num
   add eax, 20
   mov sum, eax

   INVOKE ExitProcess, 0
_main ENDP
END
```

12. 在适当的位置添加断点（例如可以设定在 `mov eax, num` 这条语句上）。

13. 生成并运行该程序。

14. 当程序在断点处暂停的时候，你可以根据自己的喜好在 Visual Studio 界面里安排一些小窗口。笔者建议你通过 Visual Studio 的 Debug -> Windows（调试 -> 窗口）菜单，把下列窗口显示出来。

- Registers（寄存器）。
- Memory（内存）(至少要把 Memory 1 调出来)。
- Disassembly（反汇编）。
- Autos（自动窗口）。

15. 你可以在 Registers 窗口中单击鼠标右键，选择 Flags（标志）菜单项，这样就可以观察 *flags* 寄存器的值了。你也可以选择其他寄存器以观察它们的值。

16. 如果想让 Visual Studio 在给代码做汇编的时候把 listing file（清单文件 / 列表文件）也创建出来，那么就打开项目的 Properties 对话框，并单击左侧界面中的 Microsoft Macro Assembler -> Listing File（清单文件）项。在右侧界面中配置下列三项：

- 把 Generate Preprocessed Source Listing 设为 Yes。
- 把 List All Available Information 设为 Yes。
- 把 Assembled Code Listing File 设为 $(ProjectName).lst，如果你想把清单文件保存到其他地方，那就指定相应的文件路径。

重新运行程序，这样就可以创建出清单文件了。创建好之后，你可能要再次打开项目的 Properties 对话框把 Generate Preprocessed Source Listing 选项恢复成 No，以便顺利地调试程序[⊖]。

17. 创建出来的清单文件应该叫作 ProjectFolder\ProjectName\ProjectName.lst，Windows 系统会把 ProjectName.lst 文件的类型认定成 MASM Listing。比方说，masm_testing 项目的清单文件，可能会放在这里：C:\Documents\Visual Studio 2017\Projects\masm_testing\masm_

⊖ 否则可能会在生成解决方案的时候发生 LNK1104 错误。——译者注

testing\masm_testing.lst

18.（这一步是可选的）打开项目的 Properties 对话框，在左侧界面中，单击 Linker -> Advanced 项。在右侧界面中配置 Image Has Safe Exception Handlers（映像具有安全异常处理程序），从下拉列表框里选择 No。单击 OK 按钮。

## macOS 系统：Xcode（8.2 版）开发环境搭配与 GAS 兼容的 Clang/LLVM 工具

1. 如果没有安装 Xcode，就从 AppStore 中安装。

2. 打开 Xcode 新建项目。

3. 这个项目应该采用 macOS -> Application 下面的 Command Line Tool 模板。

4. 给项目指定合适的 Product Name，例如可以起名叫作 asm_testing。

5. Language 应该设为 C++。

6. 把项目创建到合适的文件夹中。

7. 在 Project Navigator 界面中选择 main.cpp，按下〈Delete〉键，选择 Move to Trash。

8. 在 Project Navigator 界面中右击本项目的文件夹，选择 New File... 菜单项。在对话框的左侧界面中选择 macOS 下面的 Other 项，在右侧界面中选择 Assembly File（这种文件的图标是一个大写的 S 字母）。然后，单击 Next 按钮。

9. 给该文件起名，然后单击 Create 按钮。

10. 在这个 .s 文件中编写汇编代码。

```
# First Program for GAS, Clang/LLVM

.data
num: .long 80

.bss
.lcomm sum, 4

.text
.globl _main
_main:
   movl num, %eax
   addl $20, %eax
   movl %eax, sum

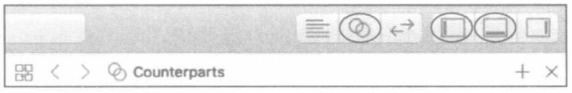
   pushl $0
   subl $4, %esp
   movl $1, %eax
   int $0x80
.end
```

11. 在 Project Navigator 界面中单击位于顶层的项目图标，在项目的 Build Settings 中，把 Architectures 选项设为 32-bit Intel (i386)。

12. 在适当的位置添加断点（例如可以把断点设定在 `movl num, %eax` 这行语句上）。

13. 构建并运行程序。

14. 根据你的喜好调整 Xcode 界面中各窗口的位置。笔者建议你采用下面这种布局（按钮显示为蓝色表示这个按钮已经按下）。




15. 在左下角的 Debug Area 区域中可以查看寄存器的值，在右下角可以查看输出信息。

16. 如果要查看反汇编代码，就在 Assistants Area 窗口（可以通过刚才那张图中画有两个重叠圆圈的按钮，把这个窗口显示出来）中点击 Counterparts（参见上图），然后选择菜单底部的 Disassembly 菜单项。

17. 如果要创建单独的清单文件，笔者推荐你使用 `otool` 工具。

- 打开 macOS 系统的 Terminal.app（终端 / 终端机）程序，通过命令进入 Xcode 项目的目标文件所在的文件夹。（有一种简单的办法是先打出“`cd`”这三个字符，然后在 Finder 中找到该文件夹，将其拖放到 Terminal 中）
- 例如，目标文件的路径可能是：asm_testing/DerivedData/asm_testing/Build/Intermediates/asm_testing.build/Debug/asm_testing.build/Objects-normal/i386/asm_testing.o
- 在 Terminal 中运行 `otool` 命令：`otool -dtvj asm_testing.o > disassembly.txt`
- 如果想了解各种输出选项的用法，就直接在命令行里打 `otool` 这几个字符，并按〈Return〉（回车）键
- 刚才那行命令中用了 4 个选项，`d` 表示打印数据段，`t` 表示打印文本段，`v` 表示详尽地打印，`j` 表示打印字节形式的操作码

## Linux 系统：NASM 汇编器搭配 ld 链接器

笔者用 Ubuntu 系统举例，如果你用的是其他 Linux 发行版请调整相应的步骤。

1. 打开 Terminal（终端 / 终端机）程序，执行下面这两条命令，以确保系统中安装了 gcc 及 g++ 工具

- `~$ gcc -v`
- `~$ g++ -v`

2. 为了确保 32 位及 64 位程序都能在系统中运行，你需要安装最新的 multilib 文件：

- `~$ sudo apt-get install gcc-5-multilib`
- `~$ sudo apt-get install g++-5-multilib`

3. 安装 NASM 汇编器然后查看其版本，以确保版本号大于或等于 2.11。

- `~$ sudo apt-get install nasm`
- `~$ nasm -v`

4. 用你喜欢的编辑器（例如 gedit、vim 或 vi 等），在适当的目录中创建后缀名为 .s 或 .asm 的文件，并在其中编写汇编代码。

```
; First Program for NASM

SECTION .data
num: dd 80

SECTION .bss
```

```
sum: resd 1

SECTION .text
global _main
_main:
   mov eax, DWORD [num]
   add eax, 20
   mov DWORD [sum], eax

mov eax, 1
mov ebx, 0
int 80h
```

5. 用 nasm 命令对这个文件做汇编以生成目标文件，然后用 ld 命令把目标文件链接成可执行的文件。

- ~$ nasm -f elf32 -o testing.o testing.asm
- ~$ ld -e _main -melf_i386 -o testing testing.o

6. 运行程序：~$ ./testing。

7. 如果要在命令行界面中调试该程序，那么请参阅附录 D。

8. 如果要创建 listing file（清单文件 / 列表文件）或是要对目标文件做其他操作，可以用 objdump 或 otool 这样的工具来完成。不过最简单的办法是在对源代码做汇编的时候指定 -l 标志：

- ~$ nasm -f elf32 -o testing.o -l testing.lst testing.asm

# 反 汇 编

## 本附录的用途

附录 B 提到了怎样在每一种操作系统中针对给定的程序来创建包含反汇编代码的清单文件，而本附录则要讲解反汇编文件的基本要素，并演示如何在 Visual Studio、Xcode 及 GDB 工具中查看汇编与反汇编代码。

## 范例程序

程序 C-1 有一个经过初始化的变量叫作 num，还有一个未经初始化的变量叫作 sum。在 _main 中，它把 num 移动到 *eax* 寄存器，然后给该寄存器加上 20，接下来把相加的结果从寄存器移入 sum，最后退出程序。

**程序 C-1　为了演示反汇编而使用的样板程序**

GAS

```
.data
num: .long 80

.bss
.lcomm sum, 4

.text
.globl _main

_main:
movl num, %eax
addl $20, %eax
movl %eax, sum

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096

ExitProcess PROTO, dwExit-
Code:DWORD

.DATA
num DWORD 80
sum DWORD ?

.CODE
_main PROC
mov eax, num
add eax, 20
mov sum, eax

INVOKE ExitProcess, 0
_main ENDP
END
```

NASM

```
SECTION .data
num: dd 80

SECTION .bss
sum: resd 1

SECTION .text
global _main
_main:
mov eax, [num]
add eax, 20
mov [sum], eax

mov eax, 1
mov ebx, 0
int 80h
```

## Windows 系统

采用附录 B 所说的办法可以创建出范例 C-1 这样的反汇编文件（也就是刚才说的清单文件）。下面展示该文件的缩略版，笔者删掉了 MASM 清单文件尾部默认包含的“Procedures, parameters, and locals”以及“Symbols”段落。

由于 MASM 所生成的目标文件包含的是**可重定向的**机器语言代码，因此，这些机器指令在该文件中的内存偏移量是从 0h 开始的。位于左侧偏移量与右侧汇编指令中间的那些十六进制数字就是机器指令。在这份文件中，经过初始化的数据（例如用 num DWORD 80 命令所初始化的 num）直接写成十六进制的原始值（例如 00000050，这就是十进制数 80 的十六进制形式）。此外要注意对比程序 C-1 的 _main 代码与反汇编文件中的相关指令之间有何异同。

范例 C-1　用 MASM 制作出来的反汇编文件

| 内存偏移量 | 机器指令 | 汇编指令 |
|---|---|---|

```
Microsoft (R) Macro Assembler Version 14.10.25017.0          03/15/17 16:45:40
testing.asm                                                  Page 1 - 1
                        .386
                        .MODEL FLAT, stdcall
                        .STACK 4096

                        ExitProcess PROTO, dwExitCode:DWORD

 00000000               .DATA
 00000000 00000050              num DWORD 80
 00000004 00000000              sum DWORD ?

 00000000               .CODE
 00000000               _main PROC
 00000000  A1 00000000 R        mov eax, num
 00000005  83 C0 14             add eax, 20
 00000008  A3 00000004 R        mov sum, eax

                                INVOKE ExitProcess, 0
 0000000D  6A 00            *   push   +000000000h
 0000000F  E8 00000000 E    *   call   ExitProcess
 00000014               _main ENDP
                        END

Microsoft (R) Macro Assembler Version 14.10.25017.0          03/15/17 16:45:40
testing.asm                                                  Symbols 2 - 1

Segments and Groups:

N a m e                 Size     Length   Align   Combine Class

FLAT . . . . . .  GROUP
STACK  . . . . . .  32 Bit   00001000 DWord    Stack   'STACK'
_DATA  . . . . . .  32 Bit   00000008 DWord    Public  'DATA'
_TEXT  . . . . . .  32 Bit   00000014 DWord    Public  'CODE'
```

文件尾部有个叫作“Segments and Groups”（段与组）的地方列出了各区段的大小。其中，_DATA 区段的大小表示所有经过初始化的变量与所有未经初始化的变量一共占据多少字节。_TEXT 段的大小表示机器指令所占据的字节总数（与这些指令相对应的汇编指令，写在 MASM 源文件的 .CODE 段中，只不过在生成清单文件的时候自动改名为 _TEXT 了，在其他两种版本的源文件里，这些汇编指令应该写在 .text 段中）。每一个十六进制的数位占据 4 个二进制位，因此，两个这样的数位就占据 8 个二进制位，或者说占据 1 个字节。这样算起来，_main 中的代码一共有 20 个字节，写成十六进制形式就是 00000014 个字节。顺便说一下，STACK 段的大小是 00001000 个字节，换算成十进制形式就是 4096 个字节，这与源文件里的 .STACK 4096 命令是相符的。

如果想用 Visual Studio 调试程序并查看其反汇编代码，那就先设置断点然后运行程序，等程序停下来的时候单击 Windows -> Debug 菜单下的 Disassembly 菜单项，如图 C-1 所示。

Disassembly 窗口显示出来之后，你会看到图 C-2 这样的内容。

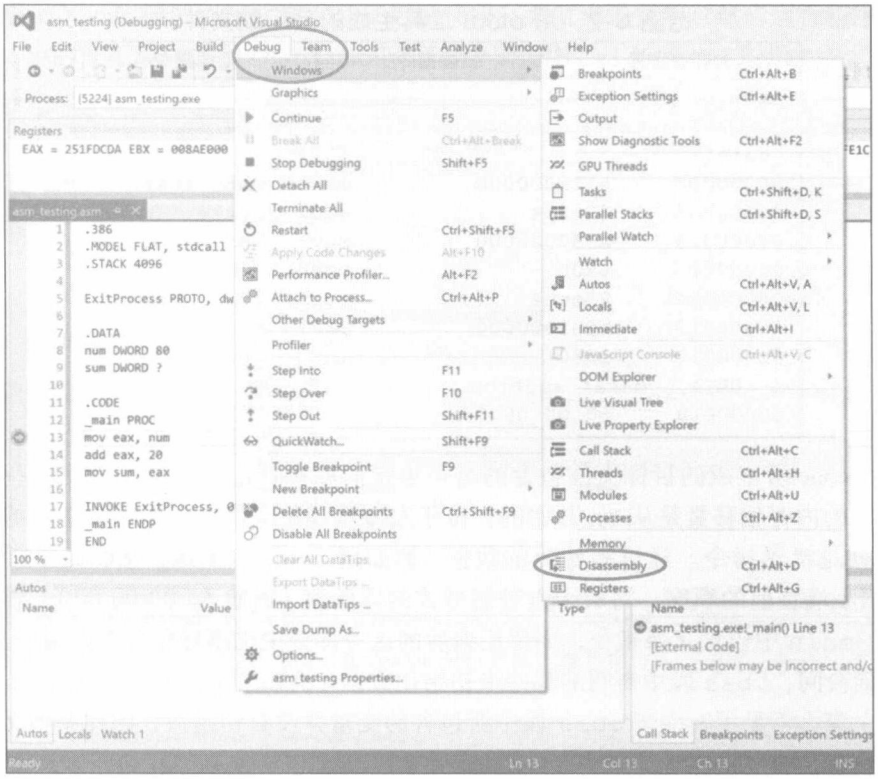



图 C-1 在 Visual Studio 开发环境中，查看程序的反汇编代码

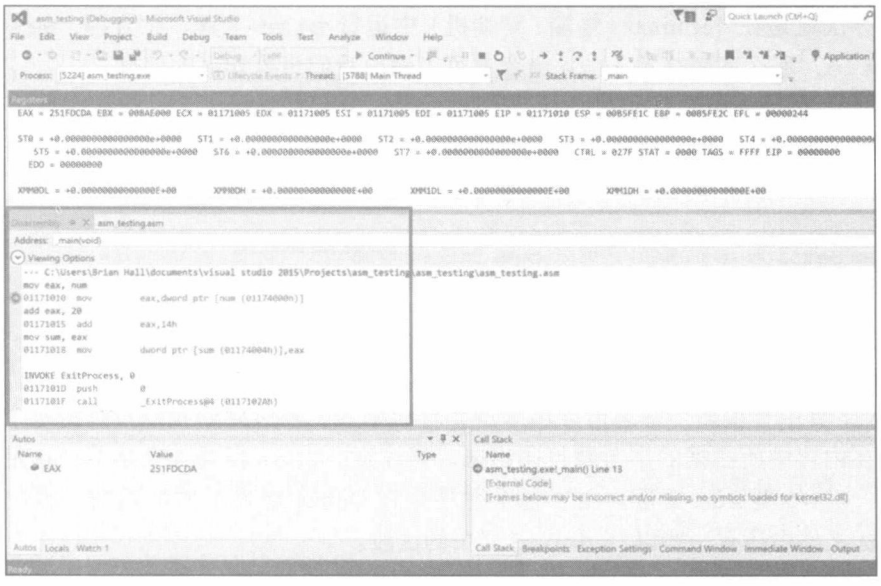



图 C-2 Visual Studio 开发环境所显示的 Disassembly 窗口

## macOS 系统

根据附录 B 中的办法可以创建出范例 C-2 的反汇编文件（或者说清单文件）。

范例 C-2　用 otool 工具生成的反汇编文件

| 内存偏移量 | 机器指令 | 汇编指令 |
|---|---|---|
| main.o: | | |
| (__TEXT,__text) section | | |
| _main: | | |
| 00000000 | a119000000 | movl num, %eax |
| 00000005 | 83c014 | addl $0x14, %eax |
| 00000008 | a333020000 | movl %eax, sum |
| 0000000d | 6a00 | pushl $0x0 |
| 0000000f | 83ec04 | subl $0x4, %esp |
| 00000012 | b801000000 | movl $0x1, %eax |
| 00000017 | cd80 | int $0x80 |
| (__DATA,__data) section | | |
| 00000019 | 50 00 00 00 | |

由于 Xcode 所生成的目标文件包含的是可重定向的机器语言代码，因此这些机器指令在该文件中的内存偏移量是从 0h 开始的。位于左侧偏移量与右侧汇编指令中间的那些十六进制数字就是机器指令。经过初始化的数据（例如用 num:　.long　80 命令所初始化的 num）按照小端在前的顺序，写成十六进制形式的原始值（例如 50 00 00 00）。请注意，程序 C-1 的 _main 中包含 7 条指令，而反汇编后的这一部分中也刚好包含 7 条指令。

你可能会问，.bss 段中声明的那些未初始化数据（例如由 .lcomm sum, 4 命令所声明的 sum）怎么不见了？由于 .bss 段中所包含的变量并没有初始值，因此系统无须保存这样的值，目标文件（扩展名通常是 .o）只需要把 .bss 中的未初始化变量在程序运行时所需的总空间记录下来就可以了。

有一种简单的办法可以查出目标文件给 .bss 段、其他区段乃至整个程序预留了多大空间，这个办法就是在 Terminal（终端 / 终端机）中运行 size 工具。比方说，如果目标文件叫作 test.o，那么下面这条 size 命令就可以输出我们想要知道的信息。

```
$ size -m test.o
Segment : 567
        Section (__TEXT, __text): 25
        Section (__DATA, __data): 4
        Section (__DATA, __bss): 4
        Section (__DWARF, __debug_info): 298
        Section (__DWARF, __debug_abbrev): 40
        Section (__DWARF, __debug_aranges): 32
        Section (__DWARF, __debug_line): 164
total 567
```

size 工具中有很多个标志可以用来调整输出格式，不过此处我们只是把每个区段的名称以及它在内存中所占据的字节数打印出来。请注意，.bss 段是 4 个字节，这足以容纳未初始化的 .long 型 sum 变量。由于这个程序是在 Debug 模式下做汇编的，因此 Xcode 所生成的目标文件中还有一些以 __debug 开头的区段。

为了更清楚地理解这些信息的含义，我们来计算一下程序的可执行部分占据多少个字节。算的时候别忘记把最后一条指令，也就是位于 00000017 这个偏移地址处的那条指令算进去。算出的结果是十进制的 25，这个大小与 size 命令所给出的 __TEXT 段是相同的。

```
Section (__TEXT, __text): 25
0x17 (beginning of last instruction) + 0x02 (cd08) = 0x19h = 25d
```

如果想用 Xcode 调试程序并查看其反汇编代码，那就先设置断点然后运行程序。Xcode 中的反汇编界面和 Oisassembily 窗口分别如图 C-3 与图 C-4 所示。

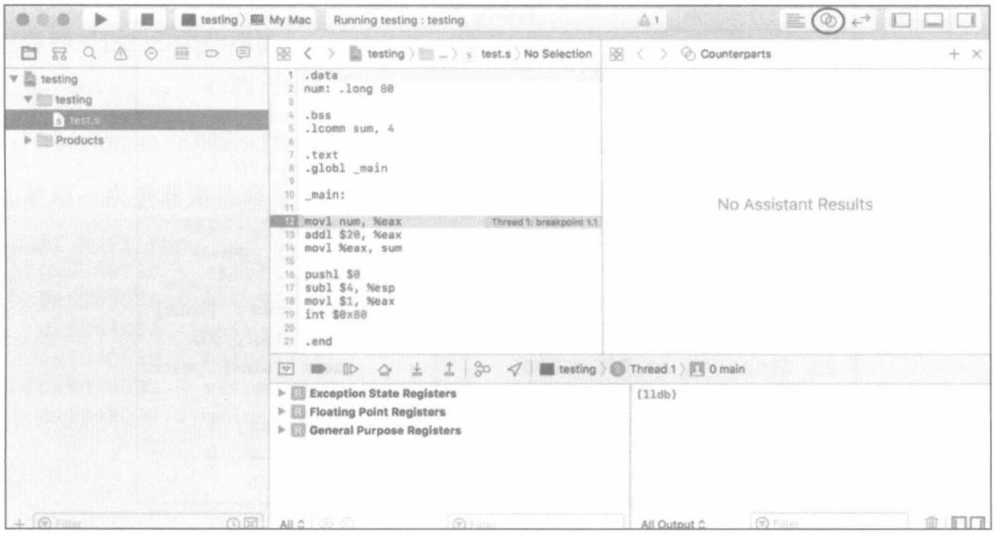



图 C-3 在 Xcode 中打开反汇编界面




图 C-4 Xcode 中的 Disassembly 窗口

单击 Assistant Editor 按钮，单击 Counterparts 这几个字，在下拉框中选择 Disassembly 这一项。假如当前调试的程序是用 C++ 等语言写的，那么可以选择 Assembly 来查看汇编代码，从而更细致地观察程序的执行过程。

## Linux 系统

根据附录 B 中的办法可以创建出范例 C-3 这样的反汇编文件（或者说清单文件）。

在 Linux 系统中，`size` 工具所支持的输出选项可能与 macOS/BSD 系统不同，此处只

给出默认的格式：

范例 C-3　NASM 所生成的反汇编代码

| 内存偏移量 | 机器指令 | 汇编指令 |
|---|---|---|
| 1 | | section .data |
| 2 00000000 | 50000000 | num: dd 80 |
| 3 | | |
| 4 | | section .bss |
| 5 00000000 | <res 00000004> | sum: resd 1 |
| 6 | | |
| 7 | | section .text |
| 8 | | global _main |
| 9 | | _main: |
| 10 00000000 | A1[00000000] | mov eax, [num] |
| 11 00000005 | 83C014 | add eax, 20 |
| 12 00000008 | A3[00000000] | mov [sum], eax |
| 13 | | |
| 14 0000000D | B801000000 | mov eax, 1 |
| 15 00000012 | BB00000000 | mov ebx, 0 |
| 16 00000017 | CD80 | int 80h |

```
$ size test.o
text    data     bss       dec       hex       filename
  25       4       4        33        21       test.o
```

与 macOS 版本一样，text、data、bss 段所占据的字节数也分别是 25、4、4。bss 段中所包含的变量没有初始值，因此系统无须保存这样的值，目标文件（扩展名通常是 .o）只需要把 bss 中的未初始化变量在程序运行时所需的总空间记录下来就可以了。

现在，笔者给出两种在 GNU Debugger（也就是 GDB）中查看反汇编代码的办法。其中比较直观的一种是使用 layout asm 命令。用 GDB 打开你要调试的程序（GDB 的用法可参阅附录 D），在 _main 这里设置断点，然后用 run 命令运行程序，接下来执行这条命令：

```
(gdb) layout asm
```

在 GDB 工具中查看指令如图 C-5 所示。

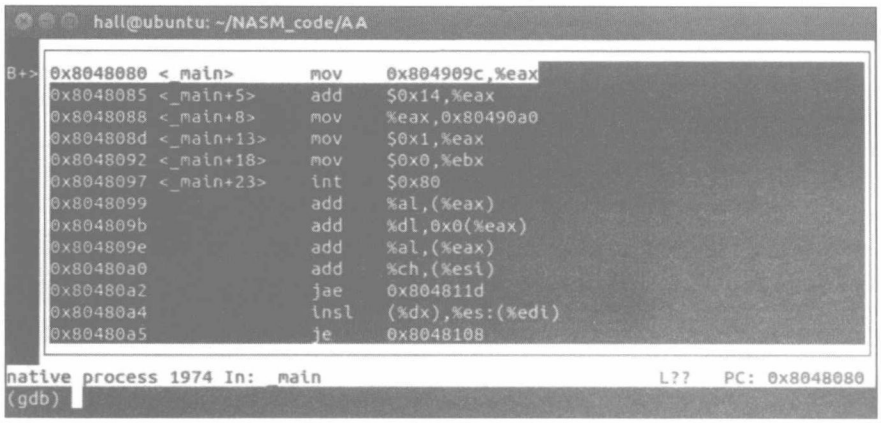



图 C-5　在 GDB 工具中，以 ASM Layout 界面查看指令

另一种办法是借助 Program Counter（PC，程序计数器）来查看当前这条指令的反汇编

信息。它所指的地方其实就是 *ip*/*eip*/*rip* 寄存器所指向的那条指令。下面这条命令中的 x，意思是 examine（检视 / 查看），i 的意思是当作 instruction（指令）来查看，$pc 这一部分代表要查看的内容所在的内存地址，本例中这个地址存放在 pc 寄存器里。你可以参阅附录 D 来了解各种 GDB 命令的用法。

```
Breakpoint 1, 0x08048080 in _main ()
(gdb) x/i $pc
=> 0x8048080 <_main>:          mov    0x804909c,%eax
```

如果想一次查看多条命令，就在 i 的前面写上你想查看的命令数量，比方说：

```
(gdb) x/6i $pc
=> 0x8048080 <_main>:          mov    0x804909c,%eax
   0x8048085 <_main+5>:        add    $0x14,%eax
   0x8048088 <_main+8>:        mov    %eax,0x80490a0
   0x804808d <_main+13>:       mov    $0x1,%eax
   0x8048092 <_main+18>:       mov    $0x0,%ebx
   0x8048097 <_main+23>:       int    $0x80
```

# 用 GDB 工具调试汇编程序

## 本附录的用途

这篇附录用来讲解怎样对 NASM 代码做汇编与链接，以及如何用 GNU Debugger（GDB）对制作出的程序做调试。通过底层指令观察程序的走向与行为是一项很有用的技能，然而要想在 Unix/Linux 的命令行环境中查看这些指令可能稍微有些困难，这正是本附录所要解决的问题。笔者假设你已经安装了 NASM 汇编器、GCC 或 Clang 编译器以及 GDB 调试器。

程序 D-1　Sum（求和）程序

| 适用于 macOS 系统的 32 位 NASM 程序代码 | 适用于 Linux 系统的 32 位 NASM 程序代码 |
|---|---|
| section .data<br>num1: dd 2<br>num2: dd 4<br><br>section .text<br>global _main, _sum<br>_main:<br><br>mov eax, 10<br>dec eax<br>mov ebx, 5<br><br>push DWORD[num2]<br>push DWORD[num1]<br>call _sum<br>add esp, 8<br><br>push DWORD 0<br>sub esp, 4<br>mov eax, 1<br>int 80h<br><br>_sum:<br>push ebp<br>mov ebp, esp<br>push ebx<br>mov ebx, [ebp + 8]<br>mov eax, [ebp + 12]<br>add eax, ebx<br>pop ebx<br>pop ebp<br>ret | section .data<br>num1: dd 2<br>num2: dd 4<br><br>section .text<br>global _main, _sum<br>_main:<br><br>mov eax, 10<br>dec eax<br>mov ebx, 5<br><br>push DWORD[num2]<br>push DWORD[num1]<br>call _sum<br>add esp, 8<br><br>mov eax, 1<br>mov ebx, 0<br>int 80h<br><br><br>_sum:<br>push ebp<br>mov ebp, esp<br>push ebx<br>mov ebx, [ebp + 8]<br>mov eax, [ebp + 12]<br>add eax, ebx<br>pop ebx<br>pop ebp<br>ret |

## 源代码

为了讲解 GDB 工具中的各种调试命令，我们首先得把要调试的程序构建出来，本附录所用的是一个以 NASM 代码写成的 32 位程序。这个程序的源代码应该编写并保存于 sum.asm 这样的文件中，该文件的扩展名可以是 .asm，也可以是 .s。

## 汇编并链接

**汇编**（Assembling），是指根据 .asm 或 .s 文件中的代码来产生后缀名为 .o 的目标文件。针对每一种操作系统生成的目标文件的格式可能会有所不同。Linux 系统的目标文件标准格式是 ELF，mac OS 系统是 MACH-O，Windows 系统是 COFF。

- Linux 系统可以用下面这样的命令给 sum.asm 做汇编（本附录用 .asm 后缀名表示适用于 Linux 系统的汇编源文件）：

```
nasm -f elf -o sum.o sum.asm
```

- Mac 系统可以用下面这样的命令给 sum.s 做汇编（本附录用 .s 后缀名表示适用于 macOS 系统的汇编源文件）：

```
nasm -f macho -o sum.o -l sum.lst sum.s
```

注意看 nasm 命令的选项。`-f` 标志的后面写的是目标文件的格式，`-o` 标志的后面写的是输出的目标文件应该叫什么名字。最后的文件名指的是有待汇编的源代码。针对 macOS 系统而写的那条命令中还用到了 `-l` 标志，该标志后面写的是清单文件的名称，有的时候，你可能想把做汇编所得到的程序与做汇编之前的源代码相互核对一下，看看其中的指令及内存偏移量是否一致，这就需要用到这份清单文件了。

`elf` 与 `macho` 都是 32 位格式，如果要针对 64 位环境做汇编，应该使用 `elf64` 及 `macho64` 格式。

**链接**（Linking），是指根据后缀名为 .o 的目标文件，生成能够在 Unix 系统上运行的可执行文件。这可以通过 ld 这样的链接器来做，也可以通过 gcc 或 clang 这样的编译器来做。

- 在 Linux 系统中可以用下面这样的 `ld` 命令给 sum.o 文件做链接。

```
ld -e _main -melf_i386 -o sum sum.o
```

`-e` 标志用来指定程序的入口点，对于本例来说这个入口点是 `_main`。`-melf_i386` 这种写法用来指定链接器所针对的目标文件以及它要模拟的目标架构，在本例中，目标文件的格式是 `elf`，可执行文件的目标架构是 `i386`。选项开头的 m 代表 emulation。`-o` 标志的后面写的是要输出的可执行文件应该叫什么名字。最后的 `sum.o` 就是当前正要做链接的这个文件。

- 还有一个办法是用编译器做链接，而不是直接使用链接器。比方说，在 Linux 及 Mac 系统上可以选用下面这两条命令对 sum.o 文件做链接：

```
clang -o sum -m32 sum.o
gcc -o sum -m32 sum.o
```

首先出现的 clang 或 gcc 是编译器的名字。`-o` 选项后面写的是输出的可执行文件名。`-m32` 的意思是针对 32 位架构做链接。最后的 `sum.o` 是你要做链接的文件。给 64 位架构做链接时要把 `-m32` 改成 `-m64`。

## 用 GDB 调试程序

对代码做汇编及链接之后，应该能够得到一份可执行的文件。在 Terminal（终端机 / 终端）中运行这样一条命令就可以用 GDB 打开 sum 程序了：

```
gdb sum
```

如果你是在 macOS 中通过 MacPorts 安装 GDB 的，那么必须使用别名 ggdb。

```
ggdb sum
```

执行完命令之后，GDB 调试器就会打开有待调试的程序，并把它的一些信息列出来。笔者假设 sum 程序就位于你执行 gdb 时所在的目录中。你也可以用完整路径的形式来指定需要调试的程序。

GDB 的调试命令有很多种，此处只讲解最基本的几种，使你能够由此入手来使用调试器。调试之前必须先设置断点。本例把断点设置在程序刚刚进入 _main 的地方。

```
(gdb) break _main
```

请注意，如果要通过 break 命令给 _main 设置断点，那么在某些系统（例如大多数 Linux 发行版）中 _main 开头的下划线必须写出来，而在另一些系统（例如 macOS、FreeBSD）中则不用，因为 gdb 默认会给 main 添加下划线，无须用户明确指出。

```
(gdb) break main
```

GDB 会把断点所在的内存位置回报给我们。

```
Breakpoint 1 at 0x1fce
```

此外，还可以在 _sum 这里也设置一个断点。GDB 同样会把这个断点的内存位置显示出来。

```
(gdb) break sum
Breakpoint 2 at 0x1ff5
```

现在可以用 run 命令运行程序了。GDB 告诉我们程序已经启动，然后停在它所遇到的第一个断点处。

```
(gdb) run
Starting program: /pathToSumProgram/sum
Breakpoint 1, 0x00001fce in main ()
```

接下来，我们查看程序当前的状态，尤其要查看寄存器与内存的状态。info registers 命令可以把所有的寄存器列出来。

```
(gdb) info registers
eax            0x1fce   8142
ecx            0x0      0
edx            0x984f16d9     -1739647271
ebx            0xbffffb54     -1073743020
esp            0xbffffabc     0xbffffabc
ebp            0xbffffae4     0xbffffae4
esi            0x0      0
edi            0x0      0
eip            0x1fce   0x1fce <main>
eflags         0x246    [ PF ZF IF ]
cs             0x1b     27
ss             0x23     35
ds             0x23     35
es             0x23     35
fs             0x0      0
gs             0xf      15
```

print 命令可以专门查看某一个寄存器的状态，它会以十进制的形式打印其内容。

```
(gdb) print $eax
$1 = 8142
```

如果要用其他格式来查看，那就把代表该格式的字母放在 print 命令后面。例如下面这种写法，会以十六进制的形式打印 *eax* 的内容。

```
(gdb) print/x $eax
$2 = 0x1fce
```

x 命令能够查看某一段内存中的多项数据，它后面可以带有数据的个数、显示的格式、每份数据的大小，以及首份数据所在的内存地址。例如下面这条命令的意思是：查看 8 份数据的十六进制值，每份数据的大小都是一个单字，这些数据位于 *esp* 寄存器所指向的内存地址处。命令中的第 1 个 x 表示 examine（查看、检视），斜线后面的 8 表示要查看 8 份数据，第 2 个 x 表示以 hexadecimal（十六进制）格式来查看，它后面的字母 w 表示每份数据的大小都相当于一个 word（字），最后的 $esp 表示要查看的这些数据，出现在由 *esp* 寄存器所指向的地方。

```
(gdb) x/8xw $esp
0xbffffabc: 0x984f16d9  0x00000001  0xbffffaec  0xbffffaf4
0xbffffacc: 0xbffffb54  0x8fe00000  0xbffffadc  0x00000000
```

你也可以用函数的名称来表示这些数据的起始地址，例如：

```
(gdb) x/16xw main
0x1fce <main>:    0x00000ab8  0x05bb4800  0xff000000  0x00200435
0x1fde <main+16>: 0x0035ff00  0xe8000020  0x00000007  0x0108c483
0x1fee <main+32>: 0x55c348d8  0x8b53e589  0x458b085d  0x5bd8010c
0x1ffe <sum+13>:  0x0002c35d  0x00040000  0x00000000  0x00000000
```

上面的命令都是在程序开始执行第一条指令之前所运行的。接下来我们就要正式执行这些指令了，在执行过程中可以反复检视寄存器与内存的值以了解程序的运行情况。首先要执行的是下面这条指令：

```
mov eax, 10.
```

nexti 命令（简写为 ni）能够执行当前指令，并使程序停在下一条指令处。GDB 会给出下一条指令的地址。

```
(gdb) nexti
0x00001fd3 in main ()
```

我们再次查看 *eax* 寄存器：

```
(gdb) print $eax
$3 = 10
(gdb) print/x $eax
$4 = 0xa
```

接下来执行后面 3 条指令。程序在执行完 push DWORD[num2] 之后会停在 push DWORD[num1] 这里。

```
(gdb) nexti 3
0x00001fdf in main ()
```

刚才已经把 num2 推入栈中了，于是我们用下面这条指令来看看内存的变化情况。

```
(gdb) x/8xw $esp
0xbffffab8: 0x00000004  0x984f16d9  0x00000001  0xbffffaec
0xbffffac8: 0xbffffaf4  0xbffffb54  0x8fe00000  0xbffffadc
```

注意栈顶的值是 0x00000004，这正是刚才推入的 num2 值。接下来再执行一条指令，然后查看栈的内容。

```
(gdb) nexti
0x00001fe5 in main ()
(gdb) x/8xw $esp
0xbffffab4: 0x00000002  0x00000004  0x984f16d9  0x00000001
0xbffffac4: 0xbffffaec  0xbffffaf4  0xbffffb54  0x8fe00000
```

由于两个参数都已经入栈，因此现在可以通过 call _sum 指令来调用 _sum 函数。此时如果使用 nexti 命令来执行，那么程序将**跨过**整个 _sum 函数，并停留在 add esp,8 这条指令处。然而我们想要做的是**步入** _sum 函数中一条一条地执行指令，因此应该改用 stepi 命令。另一种办法是在 _sum 处设置断点，然后执行 nexti 命令，这也会令程序停在 _sum 函数开头，而不会直接把整个函数都执行完。

```
(gdb) stepi
0x00001ff1 in sum ()
```

现在你可以在 _sum 函数中单步执行各条指令了。执行过程中可以查看寄存器与内存，以确保程序的行为与你想要实现的效果相符。如果你要直接把程序运行完或是运行到下一个断点处，那就执行 continue 命令（简写为 c）。

```
(gdb) continue
[Inferior 1 (process 78292) exited normally]
```

刚才那段信息中的 inferior 是 GDB 用来管理程序执行状态的一种手段，它通常是与某个进程关联起来的。如果待调试的程序还没有执行完毕就运行 quit（退出）命令，GDB 会给出下面这样的提示：

```
Inferior 1 [process 78868] will be killed.
Quit anyway? (y or n)
```

如果程序在你执行完 continue 命令之后得以运行完毕，那么你可以直接执行 quit 命令来退出 GDB 调试器，这种情况下它不会给出刚才那样的提示。

```
(gdb) quit
```

最后要提醒大家，layout asm 命令能够把调试界面展示得更直观一些，使你看到程序中的各条指令以及这些指令所在的内存地址。早前的附录 C 也提到过这个命令。

```
(gdb) layout asm
```

## 资源

下面这几份资源详细给出了 nasm、ld 及 gdb 工具的各种命令及选项。

- NASM 手册（第 2 章）：https://www.nasm.us/doc/nasmdoc2.html
- ld 的手册页面：https://linux.die.net/man/1/ld
- GDB 手册：https://sourceware.org/gdb/current/onlinedocs/gdb/
- GDB 命令索引表：https://sourceware.org/gdb/current/onlinedocs/gdb/Command-and-Variable-Index.html

# 链接汇编代码及 C++ 代码

## 本附录的用途

这份附录会告诉你怎样创建一套模板，以便将汇编代码及 C++ 代码混合起来编程。Windows 系统是用 Visual Studio 开发环境搭配 MASM 汇编器来举例的，macOS 系统用的是 Xcode 开发环境及 Clang/LLVM 工具（这与 GAS 相兼容），Linux 系统用的是 NASM 汇编器。

## Windows 系统：Visual Studio 开发环境搭配 MASM 汇编器

- 打开 Visual Studio，新建空白的 C++ 项目。
- 添加一份新的文件并命名，例如叫作 main.cpp。
- 把下列代码放入该文件中。

```
// main.cpp
extern "C" void asmMain();

int main(){
        asmMain();
        return 0;
}
```

- 新建另外一份文件并命名，例如叫作 code.asm。
- 把下列代码放入该文件中。

32 位

```
; code.asm
.386
.MODEL FLAT, stdcall
.STACK 4096
.CODE
asmMain PROC C
    ; your code will go here
    ret
asmMain ENDP
END
```

64 位

```
; code.asm
.CODE
asmMain PROC
        ; your code will go here
        ret
asmMain ENDP
END
```

- 32 位版的代码定义了名为 asmMain 的过程，并宣称该过程采用的是 cdecl 调用约定。如果想令所有的过程都采用这种约定方式，那就把 .MODEL 命令改成 .MODEL FLAT, C。
- 64 位版的代码不需要指出调用类型，因为只能使用 x64 调用约定这一种。
- 开启 Developers Command Prompt。你可以在 Start Menu 中搜索 dev 或 command prompt，也可以自己寻找这个程序。它的具体路径取决于 Visual Studio 的版本及安装位置。比方说可能安装在下面这个地方。后面给出两个链接以供参考。

路径：C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\Common7\Tools

链接：

https://msdn.microsoft.com/en-us/library/ms229859.aspx

https://msdn.microsoft.com/en-us/library/f35ctcxw.aspx

- 在 Developers Command Prompt 中，通过 cd（change directory，改变目录）命令进入 code.asm 所在的文件夹，然后运行适当的命令对 .asm 文件做汇编，以产生目标文件。

| 32 位 | 64 位 |
| --- | --- |
| ml /c /Cx /coff code.asm | ml64 /c /Cx code.asm |

ml 是 32 位的 MASM 汇编器与链接器

它有可能位于：C:\Program Files(x86)\Microsoft Visual Studio 14.0\VC\bin\ml.exe

ml64 是 64 位的 MASM 汇编器与链接器

它有可能位于：C:\Program Files(x86)\Microsoft Visual Studio 14.0\VC\bin\x86amd_64\ml64.exe

/c 的意思是，只汇编不链接

/Cx 的意思是，保持 public 符号及 external 符号的大小写不变

/coff 的意思是，产生 coff 格式的目标文件（该选项只在 32 位的环境中才需要使用，ml64 不支持这个选项）

ml 及 ml64 这两个命令行工具的用法，请参见：https://msdn.microsoft.com/en-us/library/s0ksfwcf.aspx

有的时候，你可能要把 ml 或 ml64 所在的路径添加到 Windows 系统的 PATH Environment Variable（环境变量）中，具体情况如何，要根据 Visual Studio 与 Windows 的版本以及 Visual Studio 的安装位置来定。下面举一个路径做例子，并给出两个参考文档。

C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Tools\MSVC\14.10.25017\bin\HostX64\x64

为命令行生成设置路径和环境变量：

https://msdn.microsoft.com/en-us/library/f2ccy3wt.aspx

怎样把路径添加到 PATH 环境变量中：

https://msdn.microsoft.com/en-us/library/office/ee537574.aspx

- 回到 Visual Studio，给项目添加“Existing item...”（现有项…），选取上一步所创建出来的 code.obj 文件。
- 生成并运行程序，以确保整个项目能够正常运作。
- 现在，你可以在 code.asm 中添加代码，以调用 main.cpp 中所定义或包含的 C/C++ 函数，也可以反过来在 main.cpp 中添加代码，以调用 code.asm 中的汇编函数。
- 如果你在 code.asm 中创建了函数，那么别忘了把函数的原型添加到 .DATA 段之前。

例如：inputInteger PROTO C

- 注意：如果修改了代码，那么**必须**通过 Developers Command Prompt 重新对 .asm 文

件做汇编，这样才能在重新构建并运行程序的时候看到改动后的效果。

- 你可以在 C++ 文件中添加调试断点，但是不能在汇编文件中添加。假如要在汇编文件中设置断点，那么应该使用 `int 3` 指令。

## macOS 系统：Xcode 开发环境搭配与 GAS 兼容的 Clang/LLVM 工具

- 打开 Xcode，新建空白的 C++ 项目。
- 添加新文件并命名。例如可以叫作 main.cpp。
- 把下列代码放入 main.cpp 中。

```
// main.cpp
extern "C" void asmMain();

int main(){
      asmMain();
      return 0;
}
```

- 再添加一份新的文件并为其命名。例如可以叫作 code.s。
- 把下列代码放入 code.s 中。

```
# code.s
.data

.bss

.text
.globl _asmMain
_asmMain:
      # your code will go here
ret
.end
```

- 点击 Project Navigator 中的项目。将右侧的界面向下滚动，找到 Apple LLVM 8.0 - Custom Compiler Flags 这个区域，在其中的 Other C Flags 项中添加 `-mstackrealign` 标志。该标志会重新调整函数调用所涉及的栈，使得早前那种向 4 字节处对齐的遗留代码与当前这种向 16 字节处对齐的新式代码能够共存。macOS 系统的 32 位与 64 位代码都必须向 16 字节处对齐。
- 构建并运行程序以确保项目能够正常运作（该程序同时适用于 32 位及 64 位环境）。
- 你现在可以在 code.s 中添加代码以调用 main.cpp 中所定义或包含的 C/C++ 函数，也可以反过来在 main.cpp 中添加代码，以调用 code.s 中的汇编函数。
- 在 code.s 文件中调用 C/C++ 函数时，应该在函数名的前面添加下划线。就算函数在 .cpp 文件中不以下划线开头，你也还是要在 .s 文件中这样做。

## Linux 系统：NASM 汇编器

选用你喜欢的开发环境或编辑器来执行下列步骤。

- 新建文件并命名。例如可以叫作 main.cpp。
- 把下面这些代码放入 main.cpp 中。

```
// main.cpp
extern "C" void asmMain();

int main(){
        asmMain();
        return 0;
}
```

- 再新建一个文件并命名。例如可以叫作 code.asm。
- 把下面这些代码放入 code.asm 中。

```
; code.asm
SECTION .data

SECTION .bss

SECTION .text
global asmMain
asmMain:
        ; your code will go here
ret
```

- 构建并运行程序以确保项目能够正常运作。在 64 位的 Linux 系统中执行下面这些命令之前，必须确保系统中已经安装了 gcc 及 g++ multilib。
  - 开发 32 位程序

    汇编：`nasm -f elf32 code.asm`（也可以把 `elf32` 直接写成 `elf`）

    编译并链接：`g++ -m32 main.cpp code.o -o test`

    运行：`./test`
  - 开发 64 位程序

    汇编：`nasm -f elf64 code.asm`

    编译并链接：`g++ main.cpp code.o -o test`

    另一种写法：`g++ -m64 main.cpp code.o -o test`

    运行：`./test`
- 现在，你可以在 code.asm 中添加代码，以调用 main.cpp 中所定义或包含的 C/C++ 函数，也可以反过来在 main.cpp 中添加代码，以调用 code.asm 中的汇编函数。

## extern "C" 是什么意思

给函数前面加上 extern "C" 可以把名称重整（name mangling）机制关掉。之所以有这个机制，是因为 C++ 语言与 C 语言不同，它还要考虑**重载**（overload）的问题，也就是多个函数采用了同一个名称，只是在参数列表上有所区别。为了支持重载，C++ 编译器需要重新整理函数的名称，使得系统可以区分这些原本同名的函数。具体的做法是在函数的前面及后面添加一些符号，用以表示与此函数有关的信息。每一款编译器几乎都有自己的一套重整方式。

请看第 6 章的范例 6-4。从第 24 行可以看出，sum 函数的名字变成了 __Z3sumii。Clang/LLVM 编译器会用 __Z 这个前缀来表示受过重整的符号，其后的 3 表示原函数的名字由多少个字符组成，接下来的 sum 就是重整之前的原函数名，其后的 ii 表示函数接受两

个整型的参数。当然了，每一款编译器可能都会按自己的办法来调整函数名称。你可以上网搜寻各种编译器所用的调整办法，也可以从维基百科的 https://en.wikipedia.org/wiki/Name_mangling 页面入手来研究名称重整机制。

如果启用了名称重整，那么你必须知道某个 C++ 函数经过重整之后的名字才能够在汇编代码中调用它。你要是不想去记这些名称重整规则，那就把该功能关掉，这样可以直接使用开发者（或者说你自己）所起的原名。如果要在汇编代码中调用 C++ 函数，那么必须先在 C++ 代码中把该函数标注成 `extern "C"`，在 C++ 代码中调用汇编函数时也应该这样写。

附录 F

Assembly Programming and Computer Architecture for Software Engineers

# 函数与栈

## 本附录的用途

这篇附录描述怎样操作函数与栈，在学习第 6 章的内容时可以经常参考这里。请注意，栈操作一般是通过 PUSH 与 POP 指令实现的，此外也可以通过 ADD 与 SUB 指令来调整栈指针寄存器（*esp*/*rsp*）的值，从而实现栈操作。

## 寄存器

| | |
|---|---|
| *ebp*/*rbp* | 代表栈帧基址的指针，指向当前栈帧的起始处，开发者可以根据它来引用栈中的参数 |
| *esp*/*rsp* | 栈指针。总是应该指向栈顶（也就是指向最新入栈的值） |
| *eax*/*rax* | 累加器。在大多数情况下用来存放函数的返回值，但并不是所有情况都如此 |
| *eip*/*rip* | 指令指针。指向接下来要执行的那条指令，在 64 位程序中也用来做 *rip* 相对寻址（参见第 6 章的补充材料中的程序 6-3） |
| *st(0)*/*stmm0* | x87 FPU 栈寄存器，用来存放充当返回值的浮点数 |
| *xmm0*/*zmm0* | SSE（Streaming SIMD Extensions）寄存器，用来存放充当返回值的浮点数 |

## 栈帧

栈帧按照下列步骤建立：

1. 把（按值传递的）参数所具备的值，以及（按引用传递的）参数所在的地址，放到适当的寄存器里，或将其推入栈中。

2. 调用函数，这会令函数的返回地址入栈。

3. 进入函数后，把 *ebp*/*rbp* 推入栈中。

4. 凡是由受调方来保存的寄存器都应该推入栈中（这通常是针对 64 位程序而言的）。

5. 把 *ebp*/*rbp* 设置成与 *esp*/*rsp* 相同的值。

6. 从 *esp*/*rsp* 中减去某个表示字节数量的值，以便给本函数开辟一块空间，使其能够把自己所用到的局部值复制到这里或者把本函数所用到的局部值推入栈中（这通常是针对 32 位程序而言的）。对于 64 位程序来说，它通常可以把寄存器中的值复制到刚才开辟的 scratch space 中，以进行操作。

开发者通常可以根据 *ebp*/*rbp* 来引用栈帧中的参数。在 64 位程序中，如果不再执行 PUSH、POP 或 CALL 指令，那么也可以用 *rsp* 来引用栈帧中的参数。不过，64 位模式下还是 RIP 相对寻址用得比较多一些，这也是 x86_64 的默认做法。

**RIP 相对寻址**（RIP-relative addressing）是一种与绝对位置无关的（或者说独立于绝对位置的，position-independent）方法，它根据某份数据与当前指令指针（也就是 *rip* 指向的地方）之间的偏移量来引用这份数据。有一些 64 位汇编器（例如 GAS、NASM）支持这种寻址办法，但是 MASM x64 汇编器（`ml64.exe`）不支持。对于 `ml64.exe` 来说，它依然采用与 32 位模式相似的办法访问变量与地址，这种访问方式与 *rip* 相对寻址方式之间的转化

工作会由链接器自动完成。你可以查看第 6 章的补充材料中所给出的 64 位程序，以对比这 3 种汇编器所采用的寻址办法有何区别。

范例 F-1 用来演示刚才那一套步骤中的第 6 步，也就是为局部变量创建暂存空间。请注意，使用完这些局部变量之后，可以选用两种办法来清理它们，一种是像 GAS 代码那样把暂存空间所占据的字节数加到 *esp* 上，令其回到与 *ebp* 相同的位置上，还有一种是像 MASM 与 NASM 代码那样，直接把 *ebp* 的值赋给 *esp*，从而令其变得与 *ebp* 相同。

**范例 F-1 创建暂存空间以保存局部变量（32 位程序）**

| GAS | MASM | NASM |
|---|---|---|
| `_Example:`<br>`push %ebp`<br>`movl %esp, %ebp`<br>`subl $8, %rsp`<br>`movl $10, -4(%ebp)`<br>`movl $20, -8(%ebp)`<br>`addl $8, %esp`<br>`popl %ebp`<br>`retl` | `_Example PROC`<br>`push ebp`<br>`mov ebp, esp`<br>`sub esp, 8`<br>`mov DWORD PTR[ebp-4], 10`<br>`mov DWORD PTR[ebp-8], 20`<br>`mov esp, ebp`<br>`pop ebp`<br>`ret`<br>`_Example ENDP` | `_Example:`<br>`push ebp`<br>`mov ebp, esp`<br>`sub esp, 8`<br>`mov DWORD [ebp-4], 10`<br>`mov DWORD [ebp-8], 20`<br>`mov esp, ebp`<br>`pop ebp`<br>`ret` |

如果想获取栈参数的地址，那么必须通过 LEA 指令（例如 `lea esi, variable1`），因为参数在内存中的具体位置只有等程序运行的时候才能知道。

## 传递参数

- 32 位程序，通过调用栈来传递参数。
- 64 位程序，通过寄存器来传递参数（Windows 系统最多可以用 4 个寄存器，Mac/Linux 最多可以用 14 个寄存器），如果还有参数需要传递，那么再通过栈来传递。
- 用寄存器传递参数时要注意，如果寄存器中包含有用的值，应该先将它临时保存起来，以便在调用完函数之后恢复。
- 如果参数要**按值传递**（pass-by-value），就把该值复制一份并推入栈中。在汇编代码中，你应该直接把变量的名字写在 PUSH 指令中，例如 `push num`、`push DWORD [num]`。
- 如果参数要**按引用传递**（pass-by-reference），就把参数所处的地址推入栈中。在汇编代码中，你应该通过 LEA 指令来明确地体现这种用法。
- 如果要传递的是数组，应该把首个元素所在的地址推入栈中。数组总是按引用传递的。
- 如果你通过栈中的空间来传递参数，函数返回之后记得把栈清理干净。
- 如果按照 cdecl（C declaration）调用约定来编程，应该依照相反的顺序传递参数，而且主调方应该在 CALL 指令执行完毕之后，通过 ADD *清理字节数*，*esp* 指令来清理栈。
- 如果按照 STDCALL 约定来编程，也应该依照相反的顺序传递参数，只是清理栈的工作需要由受调方通过 `ret` *清理字节数*指令来完成。

## 使用参数

早前的栈帧那一节说过，栈中的参数可以用多种办法来访问。范例 F-2 演示了几种明确访问栈参数的写法。

范例 F-2　使用栈参数

| 模式 | GAS | MASM | NASM |
|---|---|---|---|
| 32 位 | `movl 4(%ebp), %eax` | `mov eax, [ebp + 4]` | `mov eax, [ebp + 4]` |
| 64 位 | `movl -0x4(%rbp), %rsi` | `mov rsi, [rbp - 4]` | `mov rsi, [rbp - 4]` |

## 返回值

**在 32 位程序中：**

- 32 位或者更短的整数值，应该放在 *eax* 中返回。
- 33 至 64 位的整数值（例如 C++ 语言中的 `long long int` 型整数值），应该以 *eax:edx* 的形式返回。
- 浮点数应该放在 FPU 栈的 *st(0)* 寄存器里返回，如果支持 *xmm0* 寄存器，那么还应该通过它返回。

**在 64 位程序中：**

- 64 位或者更短的整数值，应该放在 *rax* 中返回。
- 浮点数应该放在 *xmm0*/*zmm0* 中返回。

**在 32 位及 64 位程序中**，如果要传递比 64 位更长的值，或是结构体、对象等数据，那么应该遵循下列步骤：

- 主调函数分配足够的栈空间，用以保存数据。
- 把栈地址（也就是指向相关对象 / 成员的指针）传给受调函数。
- 受调函数（通过指针）把要返回的数据写入这个地址。
- 如果要按值传递，那就通过通用的寄存器（例如 *eax*/*rax*、*edx*/*rdx*、*edi*/*rdi*）把数据 / 对象复制到栈中的新位置上，并把复制好的数据 / 对象传给受调函数，以供其操作。

**32 位程序的注意事项：**

- 如果按值传递，返回的时候 *ecx* 寄存器应该指向对象 / 数据在栈中的位置。
- 如果按引用传递，主调函数应该令 *eax* 指向这个数据 / 对象，使得受调函数能够用 *eax* 来操纵该数据 / 对象。

**64 位程序的注意事项：**

- 如果按值传递，返回的时候应该让通用寄存器（例如 *rax*、*dl*）指向相应数据 / 对象在栈中的位置。
- 如果按引用传递，主调函数应该令 *rdi* 指向相关的对象 / 数据，从而使受调函数能够通过 *rdi* 来操纵这个数据 / 对象。

## x86_64 程序还应该注意的问题

- 参数可以长达 8 个字节。
- 放在寄存器或内存中的参数值如果少于 64 位，那么系统不会自动用 0 来扩展它。
- Microsoft x64 程序必须根据平台的要求预留 shadow space（参见第 6 章的 6.3.3 节）。
- 函数必须按 16 字节对齐。

- 主调函数负责清理栈。

## 编写 MASM 汇编代码时需要注意的问题

- PROC 命令用来声明过程（也就是函数）及其属性与参数。
- PROTO、EXTERNDEF 及 EXTERN 命令用来标注程序所用到的外部函数。在这三种命令中应该优先考虑使用 PROTO。
- ENTER 与 LEAVE 命令会自动为受调函数创建并清理栈帧。
- LOCAL 命令可以用来声明局部变量，并为其指定名称与大小。
- INVOKE 命令只能用在 32 位环境中，它可以取代 CALL。通过该命令，开发者只需编写一行代码就能传递多个参数（参见第 10 章的程序 10-4）：

```
INVOKE displayArray, OFFSET array, LENGTHOF array, TYPE array
```

- ADDR 运算符只能在 32 位环境中使用，编写 INVOKE 语句的时候可以通过它来传递指针：

```
INVOKE displayArray, ADDR array
```

# CPUID 指令的用法

## 本附录的用途

这篇附录用来演示怎样通过 CPUID 指令来判断处理器是否支持某些功能。阅读本附录之前，你应该先看完第 9 章的内容，因为接下来的范例所要测试的功能有很多都是在第 9 章中讲到的。我们判断 CPU 是否支持 x87 FPU、MMX、SSE（及其各种变化形式）、AVX（及其各种变化形式）、SYSENTER/SYSEXIT、SYSCALL/SYSRET、FMA、F16C、RDRAND 及 RDSEED（这两条指令可以用作随机数生成器）。下面的例子既会测试标准的功能也会测试扩展的功能。

## 技术资源

- Intel 64 and IA-32 Software Developer's Manual, Volume 1, Chapter 18
- Intel 64 and IA-32 Software Developer's Manual, Volume 2A, Chapter 3
- AMD Architecture Programmer's Manual:
  http://developer.amd.com/resources/developer-guides-manuals/
- AMD CPUID Specification:
  http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/25481.pdf
- Microsoft __cpuid():
  https://msdn.microsoft.com/en-us/library/hskdteyh.aspx

## 网络资源

- https://en.wikipedia.org/wiki/CPUID（这个页面可以让你迅速而直观地查到每一个二进制位所表示的功能）
- https://wiki.osdev.org/CPUID

## 判断 CPU 是否支持某项功能

本附录最后的程序 G-1 是个带有内联式汇编代码的 C++ 程序，它能够指出 CPU 所支持的功能。其中的汇编代码是按照 Xcode/GCC 风格所写的，适用于 Visual C++ 的格式，列在附录末尾的表 G-2 中。在 Microsoft 系统上编程的时候要注意，Microsoft x64 不支持内联式的汇编代码，因此，你要么编写 x86 程序，要么就完全用汇编代码来编写程序或模块，此外，也可以考虑采用内置的 `__cpuid()` 函数。程序 G-1 在大多数平台上都可以直接运行，或者仅仅需要稍作修改就能够正常运行。

在使用 CPUID 指令之前，你也可以先判断一下自己的系统究竟支不支持这条指令（当然绝大多数都是支持的）。这正是第 10 章的作业 10.3 所要求的任务。

要想判断 CPU 是否具备某个功能，最基本的流程是把相关的数值加载到适当的寄存器

中，这些数值用来表示你想查询的是哪一类信息，例如标准功能、扩展功能、缓存信息，或内核的拓扑结构。大多数情况下，这个表示信息类别的值应该加载到 *eax* 寄存器中，然后执行 CPUID 指令，最后从包含查询结果的通用寄存器（*eax*、*ebx*、*ecx*、*edx*）中提取相应的二进制位，以确认 CPU 是否具备你要查的那项功能。如果与该功能相对应的二进制位是 1（或者说已置位，set）就表示支持，若是 0（或者说已清除 / 未置位，clear）就表示不支持。比方说，如果 *edx* 的首个二进制位（或者叫作 0 号二进制位）是 1，就表示处理器具备 Onboard x87 FPU。

接下来，笔者选取程序 G-1 中的几个片段来讲解，并给出每个片段所涉及的 CPU 功能。每项功能前面都标注了对应的二进制位在保存查询结果的通用寄存器中所处的位置，如果这个位置上的二进制位是 1 就表明 CPU 支持该功能。程序 G-1 采用 bitset 数据结构来存放通用寄存器中表示功能支持情况的二进制位，下面列出的功能只是其中的一部分。你可以修改程序，从而将每项功能的支持情况都判断出来，也可以按照你自己的需要重新选一些功能去判断。完整的程序以及该程序所输出的结果列在本附录末尾。

## 标准功能

如果想获取代表 CPU 标准功能的功能位，就把 1 加载到 *eax* 中然后执行 CPUID 指令。该指令会把这些功能位，放在 *edx* 及 *ecx* 中。范例 G-1 中的这段代码可以用来判断 CPU 是否支持某项标准功能。

范例 G-1　判断 CPU 是否支持某项标准功能

```
"movl $1, %%eax \n\t"                    // EAX = 1
"cpuid \n\t"                             // execute CPUID
"movl %%edx, %[features1] \n\t"
"movl %%ecx, %[features2] \n\t"
```

| EDX | ECX |
| --- | --- |
| Bit 0 – FPU<br>Bit 11 – SEP (SYSENTER/SYSEXIT)<br>Bit 23 – MMX<br>Bit 25 – SSE<br>Bit 26 – SSE2 | Bit 0 – SSE3<br>Bit 9 – SSSE3<br>Bit 12 – FMA3<br>Bit 19 – SSE4.1<br>Bit 20 – SSE4.2<br>Bit 28 – AVX<br>Bit 29 – F16C (half-precision)<br>Bit 30 – RDRAND (random number generator) |

## 扩展功能

如果想获取 CPU 扩展功能（或者说扩充功能）位，就把 7 加载到 *eax*，把 0 加载到 *ecx*，然后执行 CPUID 指令。该指令会把这些功能位放在 *ebx* 及 *ecx* 中。范例 G-2 中的这段代码可以用来判断 CPU 是否支持某项扩展功能。

范例 G-2　判断 CPU 是否支持某项扩展功能

```
"movl $7, %%eax \n\t"                    // EAX = 7
"xorl %%ecx, %%ecx \n\t"                 // ECX = 0
"cpuid \n\t"
"movl %%ebx, %[eFeatures1] \n\t"
```

| EBX |
| --- |
| Bit 5 – AVX2<br>Bit 16 – AVX512f<br>Bit 17 – AVX512dq<br>Bit 18 – RDSEED<br>Bit 21 – AVX512ifma |

## 判断 CPU 是否支持其他一些扩展功能

如果处理器允许CPUID指令的功能参数大于80000000h，那么你就可以把80000001h加载到 *eax* 中，然后执行 CPUID 指令。你可以根据该指令在 *edx* 及 *ecx* 中所返回的功能位来判断 CPU 是否还支持其他一些扩展功能。范例 G-3 中的这段代码可以用来判断 CPU 是否支持其他一些扩展功能。

**范例 G-3　判断 CPU 是否还支持其他一些扩展功能**

```
"movl $80000001h, %%eax \n\t"          // EAX = 80000001h
"cpuid \n\t"
"movl %%edx, %[eFeatures2] \n\t"
```

| EDX |
| --- |
| Bit 11 – SYSCALL/SYSRET |

之所以要举这个例子，是因为 64 位模式支持 SYSCALL 指令而 32 位模式不支持。因此，这项特性可以用来预测 CPU 当前是否具备 64 位 CPU 才有的那些功能。

## 举例说明程序 G-1 的输出信息

表 G-1 节录了程序 G-1 在三种 CPU 上的输出结果，为了节省篇幅，表格只展示了 CPU 是否支持各项功能，而没有展示程序 G-1 所输出的相关功能位，因此，下面就举两个例子来看看这些功能位是什么样子。范例 G-4 是在某款推出于 2012 年中期的 Intel Core i7 处理器上，以 32 位模式执行 CPUID 指令所得到的功能位，范例 G-5 则是在某款推出于 2015 年中期的 Intel Core i7 处理器上，以 64 位模式执行 CPUID 指令所得到的功能位。

**范例 G-4　在某款推出于 2012 年中期的 Intel Core i7 处理器上，以 32 位模式执行 CPUID 指令所得到的功能位**

```
CPUID bits (right-to-left)
----------
10111111111010111111101111111111 - EDX bits, EAX=1
01111111011110101110001111111111 - ECX bits, EAX=1
00000000000000000000001010000001 - EBX bits, EAX=7 & ECX=0
00101000000100000000000000000000 - EDX bits, EAX=80000001h
```

**范例 G-5　在某款推出于 2015 年中期的 Intel Core i7 处理器上，以 64 位模式执行 CPUID 指令所得到的功能位**

```
CPUID bits (right-to-left)
----------
10111111111010111111101111111111 - EDX bits, EAX=1
01111111111110101111101111111111 - ECX bits, EAX=1
00000000000000000010011110101011 - EBX bits, EAX=7 & ECX=0
00101100000100000000100000000000 - EDX bits, EAX=80000001h
```

表 G-1 在三种处理器上运行程序 G-1 所得到的结果

| Intel 核心 2 Duo（2009 年初）<br>2.93 GHz, 2 核心，32 位模式 | Intel 核心 i7（2012 年中）<br>2.6 GHz, 4 核心，32 位模式 | Intel 核心 i7（2015 年中）<br>2.5 GHz, 4 核心，64 位模式 |
|---|---|---|
| CPU Features<br>------------<br>x87 FPU: Yes<br>SYSENTER/SYSEXIT: Yes<br>MMX: Yes<br>SSE: Yes<br>SSE2: Yes<br>SSE3: Yes<br>SSSE3: Yes<br>FMA3: No<br>SSE4.1: Yes<br>SSE4.2: No<br>AVX: No<br>F16C: No<br>RDRAND: No<br><br>Extended Features<br>-----------------<br>AVX2: No<br>AVX512f: No<br>AVX512dq: No<br>RDSEED: No<br>AVX512ifma: No<br><br>More Extended Features<br>----------------------<br>SYSCALL/SYSRET: No | CPU Features<br>------------<br>x87 FPU: Yes<br>SYSENTER/SYSEXIT: Yes<br>MMX: Yes<br>SSE: Yes<br>SSE2: Yes<br>SSE3: Yes<br>SSSE3: Yes<br>FMA3: No<br>SSE4.1: Yes<br>SSE4.2: Yes<br>AVX: Yes<br>F16C: Yes<br>RDRAND: Yes<br><br>Extended Features<br>-----------------<br>AVX2: No<br>AVX512f: No<br>AVX512dq: No<br>RDSEED: No<br>AVX512ifma: No<br><br>More Extended Features<br>----------------------<br>SYSCALL/SYSRET: No | CPU Features<br>------------<br>x87 FPU: Yes<br>SYSENTER/SYSEXIT: Yes<br>MMX: Yes<br>SSE: Yes<br>SSE2: Yes<br>SSE3: Yes<br>SSSE3: Yes<br>FMA3: Yes<br>SSE4.1: Yes<br>SSE4.2: Yes<br>AVX: Yes<br>F16C: Yes<br>RDRAND: Yes<br><br>Extended Features<br>-----------------<br>AVX2: Yes<br>AVX512f: No<br>AVX512dq: No<br>RDSEED: No<br>AVX512ifma: No<br><br>More Extended Features<br>----------------------<br>SYSCALL/SYSRET: Yes |

## 演示 CPUID 指令用法的程序

程序 G-1 通过 CPUID 指令查询处理器是否支持某些功能[⊖]

包含内联式 GAS 汇编代码的 C++ 程序

```
// CPUID Features - macOS (Clang/LLVM), Linux (GCC) - 32-bit
// Copyright (c) 2017 Hall & Slonka

#include <iostream>
#include <bitset>
using namespace std;

int main()
{
   bitset<32> features1;     // standard features in EDX
   bitset<32> features2;     // standard features in ECX
   bitset<32> eFeatures1;    // extended features in EBX
   bitset<32> eFeatures2;    // extended features in EDX

   asm ("movl $1, %%eax \n\t"                  // EAX = 1
        "cpuid \n\t"                           // execute CPUID
        "movl %%edx, %[features1] \n\t"
        "movl %%ecx, %[features2] \n\t"
        "movl $7, %%eax \n\t"                  // EAX = 7
        "xorl %%ecx, %%ecx \n\t"               // ECX = 0
```

⊖ 为了顺利编译并构建程序，可能要在 asm(...); 代码块中，把第 9 行的 movl 指令所用到的 $80000001h，写成 $0x80000001。——译者注

```
      "cpuid \n\t"
      "movl %%ebx, %[eFeatures1] \n\t"
      "movl $80000001h, %%eax \n\t"        // EAX = 80000001h
      "cpuid \n\t"
      "movl %%edx, %[eFeatures2] \n\t"
      :[features1] "=m"(features1),        // outputs
       [features2] "=m"(features2),
       [eFeatures1] "=m"(eFeatures1),
       [eFeatures2] "=m"(eFeatures2)
      :                                    // inputs
      :"eax","%ebx","%ecx","%edx"          // clobbered registers
     );

// binary output of features
// output in reverse due to little-endian

cout << "CPUID bits (right-to-left)\n";
cout << "----------\n";
cout << features1 << " - EDX bits, EAX=1\n";
cout << features2 << " - ECX bits, EAX=1\n";
cout << eFeatures1 << " - EBX bits, EAX=7 & ECX=0\n";
cout << eFeatures2 << " - EDX bits, EAX=80000001h\n\n";

cout << "CPU Features\n";
cout << "------------\n";

// FPU
if (features1[0] == 1) {
   cout << "x87 FPU: Yes\n";
}
else {cout << "x87 FPU: No\n";}

// SEP (SYSENTER/SYSEXIT)
if (features1[11] == 1) {


 cout << "SYSENTER/SYSEXIT: Yes\n";
}
else {cout << "SYSENTER/SYSEXIT: No\n";}

// MMX
if (features1[23] == 1) {
   cout << "MMX: Yes\n";
}
else {cout << "MMX: No\n";}

// SSE
if (features1[25] == 1) {
   cout << "SSE: Yes\n";
}
else {cout << "SSE: No\n";}

// SSE2
if (features1[26] == 1) {
   cout << "SSE2: Yes\n";
}
else {cout << "SSE2: No\n";}

// SSE3
if (features2[0] == 1) {
   cout << "SSE3: Yes\n";
}
else {cout << "SSE3: No\n";}
```

```
// SSSE3
if (features2[9] == 1) {
   cout << "SSSE3: Yes\n";
}
else {cout << "SSSE3: No\n";}

// FMA3
if (features2[12] == 1) {
   cout << "FMA3: Yes\n";
}
else {cout << "FMA3: No\n";}

// SSE4.1
if (features2[19] == 1) {
   cout << "SSE4.1: Yes\n";
}
else {cout << "SSE4.1: No\n";}

// SSE4.2
if (features2[20] == 1) {
   cout << "SSE4.2: Yes\n";
}
else {cout << "SSE4.2: No\n";}

// AVX
if (features2[28] == 1) {
   cout << "AVX: Yes\n";
}
else {cout << "AVX: No\n";}

// F16C (half-precision)
if (features2[29] == 1) {
   cout << "F16C: Yes\n";
}
else {cout << "F16C: No\n";}

// RDRAND (random number generator)
if (features2[30] == 1) {
   cout << "RDRAND: Yes\n";
}
else {cout << "RDRAND: No\n";}

cout << "\n";

cout << "Extended Features\n";
cout << "-----------------\n";

// AVX2
if (eFeatures1[5] == 1) {
   cout << "AVX2: Yes\n";
}
else {cout << "AVX2: No\n";}

// AVX512f
if (eFeatures1[16] == 1) {
   cout << "AVX512f: Yes\n";
}
else {cout << "AVX512f: No\n";}

// AVX512dq
if (eFeatures1[17] == 1) {
```

```
    cout << "AVX512dq: Yes\n";
}
else {cout << "AVX512dq: No\n";}

// RDSEED
if (eFeatures1[18] == 1) {
    cout << "RDSEED: Yes\n";
}
else {cout << "RDSEED: No\n";}

// AVX512ifma
if (eFeatures1[21] == 1) {
    cout << "AVX512ifma: Yes\n";
}
else {cout << "AVX512ifma: No\n";}

cout << "\n";

cout << "More Extended Features\n";
cout << "----------------------\n";

// SYSCALL/SYSRET
if (eFeatures2[11] == 1) {
    cout << "SYSCALL/SYSRET: Yes\n";
}
else {cout << "SYSCALL/SYSRET: No\n";}

return 0;
}
```

## 用来执行 CPUID 指令的内联式汇编代码

表 G-2 展示了程序 G-1 中用来调用 CPUID 指令的那段内联式汇编代码。左侧是适用于 Xcode/GCC 开发环境的代码，右侧是适用于 Visual C++ 开发环境的代码。

表 G-2　用来执行 CPUID 指令的内联式汇编代码

| Xcode/GCC（32 位或 64 位） | Visual C++（32 位） |
| --- | --- |
| asm ("movl $1, %%eax \n\t"<br>"cpuid \n\t"<br>"movl %%edx, %[features1] \n\t"<br>"movl %%ecx, %[features2] \n\t"<br>"movl $7, %%eax \n\t"<br>"xorl %%ecx, %%ecx \n\t"<br>"cpuid \n\t"<br>"movl %%ebx, %[eFeatures1] \n\t"<br>"movl $80000001h, %%eax \n\t"<br>"cpuid \n\t"<br>"movl %%edx, %[eFeatures2] \n\t"<br>:[features1] "=m"(features1),<br>[features2] "=m"(features2),<br>[eFeatures1] "=m"(eFeatures1),<br>[eFeatures2] "=m"(eFeatures2)<br>:<br>:"eax","%ebx","%ecx","%edx"<br>); | __asm {mov eax, 1<br>cpuid<br>mov features1, edx<br>mov features2, ecx<br>mov eax, 7<br>xor ecx, ecx<br>cpuid<br>mov eFeatures1, ebx<br>mov eax, 80000001h<br>cpuid<br>mov eFeatures2, edx<br>}; |

# ASCII 和十进制算术运算

## 本附录的用途

这篇附录会举例说明怎样用 BCD（二进制编码的十进制数，简称二进码十进数，Binary-Coded Decimal）对 ASCII 字符做十进制算术运算。BCD 可以在（32 位的）x86 环境下使用，虽然它不受（64 位的）x86_64 环境支持，但仍然有一些用途。

## ASCII 算术

用户通过键盘输入给程序的内容，或是程序通过系统调用所获取的某些系统信息，一般都是用 ASCII 格式来表示的，这种编码格式把其中的每一个字符都与表示该字符的特定整数对应起来（参见第 1 章）。高级语言会通过转换机制，把用户通过键盘输入的每一个数位从 ASCII 码转换成与该数位相等的整数值。现在来看范例 H-1 中的 C++ 代码。

范例 H-1　输入整数

```
int num1;
cout << "Enter num1: ";
cin >> num1;
num1++;
cout << "num1: " << num1;
```

如果输入“2”，那么系统刚开始收到的值其实是 32h，这是作为字符的 2 所具备的 ASCII 码，然后，程序会把 32h 转换成与 2 这个数字在数值上完全相等的整数，从而令其能够直接参与递增等算术运算。在向控制台输出信息时系统则要反过来，把作为数字的 3 转换成与 3 这个字符相对应的 ASCII 码，也就是 33h。这两次转换都是自动完成的，无须开发者干预。表 H-1 列出了“0”至“9”这 10 个可以用作十进制数位的 ASCII 字符，以及它们所对应的 ASCII 码。

表 H-1　与十进制数所用的 10 种数位相对应的 ASCII 码

| 可以在十进制数中充当数位的字符 | 该字符的 ASCII 码（以十六进制书写） | 可以在十进制数中充当数位的字符 | 该字符的 ASCII 码（以十六进制书写） |
|---|---|---|---|
| 0 | 30 | 5 | 35 |
| 1 | 31 | 6 | 36 |
| 2 | 32 | 7 | 37 |
| 3 | 33 | 8 | 38 |
| 4 | 34 | 9 | 39 |

可是，如果要在底层操纵数值，那么为了正确地处理输入与输出，开发者必须自己来转换 ASCII 码。第 10 章的程序 10-3 很好地演示了怎样把 Process ID（进程 ID）中的每一个十进制数位都转换成与之对应的 ASCII 码。

获取用户所输入的整数字符然后执行（加法等）算术运算，最后把结果显示给用户——

这一整套任务通常可以用两种办法完成。第一种是把整数字符从 ASCII 码转换成与该字符所代表的整数相等的值，然后将转换后的值相加，最后再转换回 ASCII 码，第二种是在对应数位的 ASCII 码上直接做运算。比方说下面这个例子：为了计算 num1 与 num2 的和，首先计算它们的个位数之和，也就是把 num1 的个位数 2 作为字符时的 ASCII 码直接与 num2 的个位数 6 **作为字符时**的 ASCII 码相加，并对结果做出适当调整，这样就得到了 8 作为字符时的 ASCII 码，写成算式就是：32h（作为字符的 2 所对应的 ASCII 码）+ 36h（作为字符的 6 所对应的 ASCII 码）− 30h（做出调整时所用的值）= 38h（作为字符的 8 所对应的 ASCII 码）。算完个位之后，继续算十位、百位、千位。这样算出来的值可以直接当成 ASCII 码显示给用户。

```
Enter num1: 3402
Enter num2: 1256
The sum is: 4658
```

BCD 就是这样一种便于显示数值的机制，它采用固定个数的二进制位来表示十进制数中的每个数位。BCD 分为**压缩的 BCD**（packed BCD）与**非压缩的 BCD**（unpacked BCD）两种格式，前者一般出现在较老的系统上，它采用 nibble（半字节，或者说 4 个二进制位）表示十进制数中的一个数位，由于十进制数的数位只有 0 至 9 这 10 种，因此完全可以用 4 个二进制位来表示。后者主要用在较新的系统中，它用 byte（字节）来表示十进制数中的一个数位，如果要表示的十进制数含有 4 个数位，那么该数所对应的 BCD 码就占据 4 个字节（或者说 32 个二进制位）。

下面举一个例子来说明早前那段话中所提到的两种办法有什么区别。范例 H-1 演示的是第一种办法，下面的范例 H-2 演示第二种办法，它是根据 H-1 修改的。

**范例 H-2　输入字符**

```
char num1;
cout << "Enter num1: ";
cin >> num1;
num1++;
cout << "num1: " << num1;
```

假如在范例 H-2 中添加了适当的断点，就会发现：如果用户给程序输入的是 1，那么程序所保存的其实是 1 这个字符所对应的 ASCII 码 31h，而 num1++; 这条语句则是对这个 ASCII 码直接做递增运算，这样所得到的结果是 ASCII 码 32h，这正是 2 作为字符时所具备的 ASCII 码，因此，它可以把此时的 num1 直接打印到控制台上，令用户看到“2”这个字符。范例 H-2 并没有像范例 H-1 那样，先把 31h 这个 ASCII 码转换成数字 1 然后再对它做递增，最后把运算结果从数字 2 转换成 ASCII 码 32h。

## 用非压缩的 BCD 码做 ASCII 算术运算

对于非压缩的 BCD 码来说，它的高 4 位总是 0000，而作为字符的 0 至 9 的 ASCII 码的高四位总是 0011（如果写成十六进制，那么左侧那一位总是 3）。下面以十六进制的形式列出表示 3402 这个十进制数所用的非压缩 BCD 码及 ASCII 码。

非压缩的 BCD 码：03　04　00　02

ASCII 码：　　33　34　30　32

如果你要在底层处理 ASCII，那么可以借助 x86 指令集中的一些特殊指令方便地把运算结果转换成相应的 ASCII 码，这些指令在表 H-2 中列出。在对非压缩的 BCD 执行算术运算之后，可以利用这些指令来调整运算结果，进而将调整后的结果顺利地转为 ASCII 码。也就是说，只需要把调整后的字节值同 30h 做按位的 OR 运算，就可以得到与这个十进制数位相对应的 ASCII 码。这些 ASCII 调整指令会考虑到进位与借位。

表 H-2　针对非压缩 BCD 码的 ASCII 调整指令

| ASCII 调整指令 | 描　述 |
|---|---|
| AAA | ASCII adjust after addition（在做完加法之后执行 ASCII 调整） |
| AAS | ASCII adjust after subtraction（在做完减法之后执行 ASCII 调整） |
| AAM | ASCII adjust after multiplication（在做完乘法之后执行 ASCII 调整） |
| AAD | ASCII adjust after division（在做完除法之后执行 ASCII 调整） |

程序 H-1 演示了 AAA 指令的用法。

程序 H-1　用 AAA 指令调整两个非压缩 BCD 码的相加结果（32 位）

GAS

```
.data
num1: .byte '8'
num2: .byte '2'

.text
.globl _main
_main:

xorw %ax, %ax      # must clear
movb num1, %al
addb num2, %al
aaa                # ascii adjust
orw $0x3030, %ax   # convert to ascii

pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO,
dwExitCode:DWORD
.DATA
num1 BYTE '8'
num2 BYTE '2'

.CODE
_main PROC

xor ax, ax
mov al, num1
add al, num2
aaa
or ax, 3030h

INVOKE ExitProcess, 0
_main ENDP
END
```

NASM

```
section .data
num1: DB '8'
num2: DB '2'

section .text
global _main
_main:

xor ax, ax
mov al, [num1]
add al, [num2]
aaa
or ax, 3030h

mov eax, 1
mov ebx, 0
int 80h
```

## 用压缩的 BCD 码做 ASCII 算术运算

x86 指令集中还有一些可以处理压缩 BCD 码的特殊指令。刚才说过，这种格式的 BCD 码会用 4 个二进制位来表示十进制数中的一个数位，而不像非压缩的 BCD 那样采用 8 个二进制位。这意味着一个字节能够表示十进制数中的两个数位。如果把十进制数直接当成十六进制数来用，也就是把每个数位都照字面写下来，然后添上代表十六进制的前缀 0x 或后缀 h，这样写出来的值恰好与原十进制数的压缩 BCD 码相同。范例 H-3 就是用这种办法来声明数据的。

范例 H-3　用压缩的 BCD 码来表示十进制数（适用于 GAS 汇编器）

```
bcd1: .byte 0x42          # 42 decimal
bcd2: .word 0x2001        # 2,001 decimal
bcd3: .long 0x12345678    # 12,345,678 decimal
```

针对压缩 BCD 码的调整指令只有两种，分别用来在执行加法与减法之后做出调整，在表 H-3 中列出。

表 H-3　针对压缩 BCD 码的 Decimal 调整指令

| Decimal 调整指令 | 描　述 |
|---|---|
| DAA | Decimal adjust after addition（在做完加法之后执行 Decimal 调整） |
| DAS | Decimal adjust after subtraction（在做完减法之后执行 Decimal 调整） |

程序 H-2 演示了 DAA 指令的用法。

程序 H-2　用 DAA 指令调整两个压缩 BCD 码的相加结果（32 位）

GAS

```
.data
.text
.globl _main
_main:



xor %ax, %ax      # clear AX
xor %bx, %bx      # clear BX
mov $0x42, %al    # 42 decimal = 2A hex
add $0x68, %al    # 68 decimal = 44 hex
daa               # 110 decimal = 6E hex
                  # CF = 1, AL = 10
adc $0, %bx       # BX = 0 + BX(0) + CF
cmp $0, %bx       # if CF set(1)
jz done
carry:
add $1, %ah       # set next byte to 1

done:
pushl $0
subl $4, %esp
movl $1, %eax
int $0x80
.end
```

MASM

```
.386
.MODEL FLAT, stdcall
.STACK 4096
ExitProcess PROTO,
dwExitCode:DWORD
.DATA
.CODE
_main PROC

xor ax, ax
xor bx, bx
mov al, 42h
add al, 68h
daa

adc bx, 0
cmp bx, 0
jz done
carry:
add ah, 1

done:
INVOKE ExitProcess, 0
_main ENDP
END
```

NASM

```
SECTION .data
SECTION .text
global _main
_main:



xor ax, ax
xor bx, bx
mov al, 42h
add al, 68h
daa

adc bx, 0
cmp bx, 0
jz done
carry:
add ah, 1

done:
mov eax, 1
mov ebx, 0
int 80h
```

## 资源

- 《Intel 64 and IA-32 Software Developer's Manual》第 1 卷，第 4 章，4.7 节
- 《Intel 64 and IA-32 Software Developer's Manual》第 1 卷，第 7 章，7.3.3 节
- https://en.wikipedia.org/wiki/Binary-coded_decimal

# 编译器内部函数

## 本附录的用途

这份附录介绍编译器内部函数，这些函数通常并不是在某种编程语言的软件包或程序库中实现的，而是由编译器来实现并加以处理。它们跟内联函数比较像，也就是说，编译器每次遇到这种函数都会将其替换成一系列指令，用以表达该函数想要执行的操作。由于内部函数内建于编译器中，因此它们经过了高度的优化，这些优化通常是针对 SIMD 等操作而言的，旨在提升并行效果。

本书正文经常用 Clang、GCC 与 Visual C++ 这三种 C++ 编译器来举例，它们都实现了与 x86/x86_64 的 SIMD 操作有关的编译器内部函数。此外第 9 章还说过，Microsoft x64 不支持内联式的汇编代码，因此，为了能在高级语言的程序中执行与汇编指令类似的底层操作，开发者必须借助编译器内部函数。有些编译器所提供的内部函数专门用在特定的操作系统平台上，而另外一些编译器所提供的内部函数则可以移植到各种平台上。需要注意的是：不同的编译器以及同一种编译器的不同版本，所支持的内部函数可能也各有不同。

## 资源

- https://software.intel.com/sites/landingpage/IntrinsicsGuide/（Intel 网站中的交互式界面，这个界面做得很好）
- https://software.intel.com/en-us/node/523351（Intel 网站的编译器内部函数参考页面）
- https://msdn.microsoft.com/en-us/library/26td21ds.aspx（Microsoft 网站的编译器内部函数参考页面）
- Intel 64 and IA-32 Software Developer's Manual, Volume 2A, Chapter 3, Section 3.1.1.10 "Intel C/C++ Compiler Intrinsics Equivalents Section"
- Intel 64 and IA-32 Software Developer's Manual, Volume 2C, Appendix C "Intel C/C++ Compiler Intrinsics and Functions Equivalents"

## 编译器内部函数所涉及的数据类型

汇编代码中的 SSE 数据必须对齐到 16 字节的边界上，与之类似，针对 SSE 而设的编译器内部函数的数据也必须这样对齐。表 I-1 列出了与 MMX 及 SSE 有关的编译器内部函数所用到的数据类型。

表 I-1　针对 MMX 及 SSE 指令集里的相关功能而设置的编译器内部函数所用到的数据类型

| 数据类型 | 描　述 |
| --- | --- |
| __m64 | 某个 MMX 寄存器的内容（8 个 8 位的值、4 个 16 位的值、两个 32 位的值或 1 个 64 位的值） |
| __m128 | SSE 内部函数所用到的某个 XMM 寄存器的内容（可能是一个纯量，也可能是连在一起的 4 个 32 位的单精度浮点值） |

（续）

| 数据类型 | 描　述 |
| --- | --- |
| __m128d | SSE 内部函数所用到的某个 XMM 寄存器的内容（可能是一个纯量，也可能是连在一起的 2 个 64 位的双精度浮点值） |
| __m128i | SSE 内部函数所用到的某个 XMM 寄存器的内容（可以是 16 个 8 位的整数、8 个 16 位的整数、4 个 32 位的整数，或两个 64 位的整数） |

表 I-2 列出了与 AVX 及 AVX2 有关的内部函数所用到的数据类型，这些数据必须对齐到 32 字节边界处。针对 AVX 的内部函数基本上也可以使用 SSE 数据类型，因为 AVX 与 AVX-512 是向后兼容的。

**表 I-2　针对 AVX 指令集里的相关功能而设置的编译器内部函数所用到的数据类型**

| 数据类型 | 描　述 |
| --- | --- |
| __m256 | AVX 内部函数所用到的某个 YMM 寄存器中的内容（8 个连在一起的 32 位单精度浮点值） |
| __m256d | AVX 内部函数所用到的某个 YMM 寄存器中的内容（4 个连在一起的 64 位双精度浮点值） |
| __m256i | AVX 内部函数所用到的某个 YMM 寄存器中的内容（4 个连在一起的 64 位双精度浮点值） |

表 I-3 列出了与 AVX-512 有关的编译器内部函数所用到的数据类型，这些数据必须对齐到 64 字节边界处。

**表 I-3　针对 AVX-512 指令集里的相关功能而设置的编译器内部函数所用到的数据类型**

| 数据类型 | 描　述 |
| --- | --- |
| __m512 | AVX-512 内部函数所用到的某个 ZMM 寄存器中的内容（16 个连在一起的 32 位单精度浮点值） |
| __m512d | AVX-512 内部函数所用到的某个 ZMM 寄存器中的内容（8 个连在一起的 64 位单精度浮点值） |
| __m512i | AVX-512 内部函数所用到的某个 ZMM 寄存器中的内容（可以是 64 个 8 位的整数、32 个 16 位的整数、16 个 32 位的整数，或 8 个 64 位的整数） |

编译器内部函数所涉及的数据类型使用起来是有某些限制的，在 Intel 的相关资料中涵盖了这部分内容。在使用这些数据类型之前，必须先把定义该类型的头文件包含进来。表 I-4 中列出了所需包含的头文件。比方说，要使用 SSE 数据类型就必须包含 <xmmintrin.h>，要使用 AVX 数据类型就必须包含 <immintrin.h>。用 Clang 或 GCC 的时候，可以将 <x86intrin.h> 包含进来，这样就相当于把与各种指令集扩展有关的那些头文件全都包含进来。适用于 Visual C++ 的头文件叫作 <intrin.h>。

**表 I-4　使用编译器内部函数时所需包含的头文件**

| 内部函数所针对的指令集扩展 | 需要包含的头文件 | 内部函数所针对的指令集扩展 | 需要包含的头文件 |
| --- | --- | --- | --- |
| MMX | <mmintrin.h> | SSE4.2 | <nmmintrin.h> |
| SSE | <xmmintrin.h> | SSE4a, XOP | <ammintrin.h> |
| SSE2 | <emmintrin.h> | AES | <wmmintrin.h> |
| SSE3 | <pmmintrin.h> | AVX, FMA | <immintrin.h> |
| SSSE3 | <tmmintrin.h> | AVX512 | <zmmintrin.h> |
| SSE4.1 | <smmintrin.h> | | |

## 编译器内部函数的用法

大多数的内部函数都比较简单，只需要用一条相关的指令就能把它想要执行的操作实现出来。还有一些内部函数则必须结合多条指令才能实现。

**编译器内部函数的名称模板**

```
_mm_<operations>_<saffixs>
_mm_<操作>_<后缀>
```

从上述模板可以看出，内部函数的名称与 ADD 或 SUB 等汇编指令的名称在形式上是相似的。它们都通过后缀来表示要操作的是什么样的数据。就内部函数的后缀部分而言，首先出现的是 p、ep 或 s 等表示数据形式的字母，p 意味着数据以压缩的形式保存，ep 是 p 系列的内部函数在某个扩展指令集中的对应版本，s 代表数据以纯量的（scalar，也就是单独出现的）形式保存。这些字母后面还有一些字母，用来表示每一份数据的类型。比方说，ps 表示压缩形式的**单精度**数据，ss 表示纯量形式的单精度数据。表 I-5 列出了可以加在 p、ep 或 s 等字母后面，用以表示数据类型的一些后缀。

表 I-5 编译器内部函数中用以表示数据类型的一些后缀

| 用以表示数据类型的后缀 | 该后缀的含义 | 用以表示数据类型的后缀 | 该后缀的含义 |
|---|---|---|---|
| s | 单精度浮点数 | u32 | 无符号的 32 位整数 |
| d | 双精度浮点数 | i64 | 带符号的 64 位整数 |
| i8 | 带符号的 8 位整数 | u64 | 无符号的 64 位整数 |
| u8 | 无符号的 8 位整数 | i128 | 带符号的 128 位整数 |
| i16 | 带符号的 16 位整数 | i256 | 带符号的 256 位整数 |
| u16 | 无符号的 16 位整数 | i512 | 带符号的 512 位整数 |
| i32 | 带符号的 32 位整数 | | |

这些编译器内部函数的完整格式，是以刚才的名称模板为基础再加上返回值类型与参数列表构成的。

**编译器内部函数**

```
datatype intrinsic_name(parameters);
返回值类型 内部函数的名称(参数列表);
```

其中，datatype 表示返回值的类型，它可以是 void、int，也可以是适用于内部函数的其他类型，例如 __m64、__m128i 等。intrinsic_name 是内部函数的名称，它是根据刚才那份名称模板构造的，用来表示这个内部函数所要执行的操作。parameters 是这个内部函数所要操作的数据。

表 I-6 举了三个内部函数做例子。第一个是针对 SSE 的内部函数，用来对压缩形式的单精度值做加法。第二个是针对 AVX 的内部函数，用来把经过对齐的、压缩形式的双精度值，加载（或者说移动）到 YMM 寄存器。第三个是针对 AVX-512 的内部函数，可以根据压缩形式的单精度值计算出这些值的正弦值。与前两个例子不同，这个内部函数无法与指令集中的等效指令对应起来。

表 I-6 编译器内部函数举例

| 与该函数相对应的汇编指令助记符（指令所属的扩展集） | 描　述 | 编译器内部函数 |
|---|---|---|
| ADDPS（SSE） | 对压缩形式的单精度值做加法 | `__m128 _mm_add_ps(__m128 a, __m128 b)` |
| VMOVAPD（AVX） | 将经过对齐且以压缩形式存放的双精度值移动到某处 | `__m256d _mm256_load_pd(double const *a)` |
| 无（AVX-512） | 根据压缩形式的 32 位浮点数计算其正弦值 | `__m512 _mm512_sin_ps(__m512 v)` |

## 程序

接下来提供一些范例程序，以演示编译器内部函数的功能及用法。笔者建议你先把这些范例程序试着运行一遍，然后再研究其他的内部函数。由于接下来将会出现一些通过反汇编所得到的指令片段，因此，你应该先把**附录 C** 复习一遍以了解反汇编。用你习惯的开发环境和编译器构建程序 I-1，并在 `return 0;` 这行语句上设置断点，以观察局部变量与寄存器的状态。

**程序 I-1　通过编译器内部函数来加载压缩形式的单精度值**

C++

```
#include <xmmintrin.h>
int main()
{
    float array[4] = { 1.0, 2.0, 3.0, 4.0 };
    __m128 result = _mm_load_ps(array);      // MOVAPS

   return 0;  // set breakpoint here
}
```

程序 I-1 创建了名为 `array` 的数组，该数组的每个元素都是 32 位的浮点数，然后创建类型为 `__m128` 的 `result` 变量，并用 `_mm_load_ps` 这个编译器内部函数，把 `array` 数组中那些压缩形式的单精度浮点值加载到某个 SSE 寄存器中。请注意，调用 `_mm_load_ps` 函数时所传的参数是数组 `array` 的地址。

范例 I-1 是对程序 I-1 做反汇编所得到的信息，其中的大部分指令都在给 `main` 函数设立栈帧并保存该函数的局部变量。用箭头标出来的那两条指令才是这里要关注的内容。第一条指令是根据 *rcx* 寄存器中的地址（也就是 `array` 数组的地址）确定该地址上的数据，并将这些数据通过 MOVAPS 操作移动到 *xmm0* 寄存器。第二条指令把 *xmm0* 寄存器中的值复制到 `result` 变量。

哪一个寄存器会用来实现 MOVAPS 操作，这要由编译器决定，只不过在程序 I-1 所处的这种情况下，编译器很可能选的是 *xmm0* 寄存器。

**范例 I-1　对程序 I-1 做反汇编所得到的结果**

Xcode (Clang/LLVM)-AT&T 语法

```
cpp_testing`main:
    0x100000f10 <+0>:  pushq  %rbp
    0x100000f11 <+1>:  movq   %rsp, %rbp
    0x100000f14 <+4>:  subq   $0x40, %rsp
    0x100000f18 <+8>:  movq   0xe1(%rip), %rax
    0x100000f1f <+15>: leaq   -0x20(%rbp), %rcx
    0x100000f23 <+19>: movq   (%rax), %rdx
    0x100000f26 <+22>: movq   %rdx, -0x8(%rbp)
    0x100000f2a <+26>: movl   $0x0, -0x2c(%rbp)
    0x100000f31 <+33>: movq   0x68(%rip), %rdx
    0x100000f38 <+40>: movq   %rdx, -0x20(%rbp)
    0x100000f3c <+44>: movq   0x65(%rip), %rdx
    0x100000f43 <+51>: movq   %rdx, -0x18(%rbp)
    0x100000f47 <+55>: movq   %rcx, -0x28(%rbp)
    0x100000f4b <+59>: movq   -0x28(%rbp), %rcx
->  0x100000f4f <+63>: movaps (%rcx), %xmm0
->  0x100000f52 <+66>: movaps %xmm0, -0x40(%rbp)
    0x100000f56 <+70>: movq   (%rax), %rax
    0x100000f59 <+73>: cmpq   -0x8(%rbp), %rax
```

Xcode (Clang/LLVM)-AT&T 语法

```
0x100000f5d <+77>: jne    0x100000f6b
0x100000f63 <+83>: xorl   %eax, %eax
0x100000f65 <+85>: addq   $0x40, %rsp          ; breakpoint
0x100000f69 <+89>: popq   %rbp
0x100000f6a <+90>: retq
0x100000f6b <+91>: callq  0x100000f70
```

如果设定了断点，那你就可以看到：*xmm0* 中从低到高依次存有 1.0、2.0、3.0 及 4.0 这四个单精度浮点值。array 数组中索引较小（较先出现的、地址较低的、偏左的）浮点值会出现在 *xmm0* 的低端（或者说右端），而索引较大（较后出现的、地址较高的、偏右的）的浮点值则会出现在 *xmm0* 的高端（或者说左端）。如果直接把保存在 result 变量中的 *xmm0* 当成一个单精度浮点型的标量来获取（也就是通过 result[0] 这种写法来获取），那么获取到的实际上是 *xmm0* 中位置最低（或者说最靠右）的浮点数，也就是 array 数组中下标最小（或者说最靠左）的浮点数：array[0]。程序 I-2 通过 float element = result[0]; 这条语句演示了这个现象。

**程序 I-2　访问 result 数组中的首个元素**

```
#include <xmmintrin.h>
int main()
{
    float array[4] = { 1.0, 2.0, 3.0, 4.0 };
    __m128 result = _mm_load_ps(array);

    float element = result[0];

    return 0;
}
```

执行完刚才说的那行赋值语句之后，element 变量的值是 1.0，由此可见，1.0 保存在 result[0] 中，2.0 保存在 result[1] 中，3.0 与 4.0 分别保存在 result[2] 与 result[3] 中。如果想按照字面顺序，让数组中靠右的（索引较大的）浮点值也同样出现在某个 SSE 寄存器的右端（低端），那么应该在 C++ 代码中改用 _mm_set_ps 这个内部函数来实现。

**范例 I-2　通过编译器内部函数设置压缩形式的单精度值**

```
__m128 result = _mm_set_ps(1.0, 2.0, 3.0, 4.0);
float element = result[0];
```

如果执行范例 I-2 中的代码，那么 element 变量的值就是 4.0，因为 _mm_set_ps 与 _mm_load_ps 恰好相反，它在 result[0] 中保存的是 4.0，在 result[1] 中保存的是 3.0，而在 result[2] 与 result [3] 中所保存的则分别是 2.0 与 1.0。

我们通过编译器内部函数把待操作的数值加载到某个 SSE 寄存器中，并将其赋给类型合适的变量，接下来就可以通过这个变量轻松地执行底层操作了。例如可以像程序 I-3 这样，只用一次乘法就同时算出四个浮点值的平方。该程序是根据程序 I-1 修改而来的。

**程序 I-3　对 result 中的几个浮点数分别取平方**

```
#include <xmmintrin.h>
int main()
```

```
{
    float array[4] = { 1.0, 2.0, 3.0, 4.0 };
    __m128 result = _mm_load_ps(array);

    result = result * result;

    return 0;
}
```

对程序 I-3 做反汇编之后可以发现：编译器实际上是用 MULPS 这样的 SSE 指令来实现乘法操作的，而没有用普通的乘法指令。范例 I-3 演示了反汇编的结果。

**范例 I-3　对程序 I-3 做反汇编所得到的指令（节选）**

```
0x100000f3b <+59>: movaps %xmm0, -0x40(%rbp)
0x100000f3f <+63>: mulps  -0x40(%rbp), %xmm0
0x100000f43 <+67>: movaps %xmm0, -0x40(%rbp)
```

从上述范例可以看出，程序先把 *xmm0* 寄存器中的值复制到内存中的某个区域（也就是 `result` 所在的区域），然后用 MULPS 指令，把该区域内的几个浮点数同 *xmm0* 寄存器中对应的浮点数相乘。最后，把保存在 *xmm0* 寄存器中的相乘结果又复制回（**`result`** 所在的）内存。另外，我们还可以像范例 I-4 这样直接调用名为 `_mm_mul_ps` 的编译器内部函数，而不编写 `result = result * result;` 这条语句。

**范例 I-4　通过编译器内部函数对压缩形式的单精度值做乘法**

```
result = _mm_mul_ps(result, result);
```

程序 I-4 演示了编译器内部函数的另一种用法，也就是用这些函数所涉及的数据类型来声明 union(联合体）中的字段，使得开发者一方面能够把这个 union 当成 array 数组来操作，从而分别处理其中的各个元素，另一方面，又能把它当成扩展指令集中的一种 vector 向量，从而针对整份数据运用某些指令。比方说，当这个 union 作为 vector 出现的时候，你可以获取整个 vector 的地址，而当它作为 array 出现的时候，则可以通过 `[]` 来访问 array 中的单个元素并获取该元素的地址。

程序 I-4 还演示了一个与 AVX2 扩展有关的编译器内部函数，即 `_mm256_add_epi8`。AVX 技术引入了 256 位的 YMM 系列寄存器以及相关的浮点指令。AVX2 在此基础上又添加了一些指令，使得开发者可以把 YMM 的内容当成整数来运算。如果要用 Clang 或 GCC 编译器来编译这种带有 AVX2 指令的代码，那么必须指定 `-mavx2` 编译选项。除了 AVX2 之外，其他一些扩展指令集也有对应的编译选项，例如 SSE 对应于 `-msse`，SSE 4 及 SSE 4.2 分别对应于 `-msse4` 及 `-msse4.2` 等。还有一种办法是采用 `-march=native` 选项来编译，这样就可以在代码中使用处理器所支持的全部扩展指令了。可以通过 CPUID 来查询当前的处理器支持哪些扩展指令集，本书附录 G 讲解了它的用法。

程序中的 `U32i` 是一种 union，其中的 `v` 字段是数据类型为 `__m256i` 的 256 位整型向量（integer vector），与该字段共享同一块空间的 `a` 字段是元素类型为 `int8_t` 的数组，该数组共包含 32 个 8 位整数。这样的 union 使得开发者能够通过数组 `a` 来访问向量 `v` 中的各个元素（或者说各个分量）。程序 I-4 首先创建了名为 `testing` 的 union 结构，然后通过 `for` 循环来设定 `testing` 中的值，也就是把 0 至 31 这 32 个整数依次赋给向量 `v` 中的 32 个分量，接下来，用 `testing` 做参数调用编译器内部函数。

**程序 I-4 对压缩形式的 32 个 8 位整数做加法**

C++

```
#include <immintrin.h>
// also set -mavx2 compiler flag

union U32i
{
    __m256i v;
    int8_t a[32];
};

int main()
{
    U32i testing;

    for(int i = 0; i < 32; i++){
        testing.a[i] = i;
    }

    __m256i v2 = _mm256_add_epi8(testing.v, testing.v);  // VPADDB

    _mm256_load_si256(&v2);  // VMOVDQA

    return 0;  // set breakpoint here
}
```

程序创建了一个 __m256i 类型的整型向量 v2，用来保存对 testing.v 做加法所得到的结果。这次加法是把 testing.v 加到了它自己身上，实际上相当于把 testing.v 内的每一个整数值都同时乘以 2。注意 _mm256_add_epi8 内部函数的 _epi8 后缀，表示该函数是个扩展版本（e）的函数，用来处理压缩（p）形式的 8 位整数（i8）。所谓扩展版本，意思是说这个内部函数针对的是 AVX 指令集而非 SSE 指令集，其中所涉及的寄存器也是 YMM 系列的寄存器而非 XMM 系列的寄存器。调用完这个函数之后，程序又调用名为 _mm256_load_si256 的编译器内部函数，从而将 v2 这个 256 位整型向量中的那些带符号整数加载到某个 YMM 寄存器中。与这两个编译器内部函数有关的反汇编片段，参见范例 I-5。

**范例 I-5 对程序 I-4 做反汇编所得到的结果（片段）**

```
0x100000f6f <+79>:  vmovaps 0x40(%rsp), %ymm0
0x100000f75 <+85>:  vmovaps %ymm0, 0xa0(%rsp)
0x100000f7e <+94>:  vmovaps %ymm0, 0x80(%rsp)
0x100000f87 <+103>: vmovaps 0xa0(%rsp), %ymm1
0x100000f90 <+112>: vpaddb %ymm0, %ymm1, %ymm0
0x100000f94 <+116>: vmovaps %ymm0, (%rsp)
```

这是一段通过反汇编所得到的指令，它可以分成 5 个步骤：（1）把 vector 的值移动到 *ymm0* 寄存器；（2）通过 VMOVAPS 指令把 vector 的值复制到内存中的两个位置上（这实际上是给调用 _mm256_add_epi8 内部函数所用的那两个参数分别做了一份复制）；（3）把 vector 的值复制到 *ymm1* 寄存器；（4）用 VPADDB 指令将 *ymm0* 与 *ymm1* 相加，并把结果保存到 *ymm0*；（5）把结果存回内存中。

请注意，尽管这些编译器内部函数在形式上也是带有返回值及参数的函数，但是，程序在执行它们的时候其实并没有调用 *call* 指令，从这个角度来说，这些函数的运作方式与内联函数比较像。不过，它们在某些行为上与普通的函数还是很接近的，例如在刚才的例子

中，程序会把执行内部函数所用参数的 testing.v 像普通函数的参数那样以按值传递的方式复制到内存中，这实际上相当于给参数做复制。此外还要注意，虽然某些编译器内部函数确实有与之对应的 SSE 或 AVX 指令（例如刚才的例子所调用的第二个内部函数 _mm256_load_si256），但是编译器在编译程序的时候不一定会用这条指令来实现这个内部函数，因为它或许能够借助其他指令更为高效地完成这项任务。具体到程序 I-4 来说，由于相加的结果本身就保存在 *ymm0* 寄存器中，因此第二个内部函数 _mm256_load_si256 所要达成的效果实际上已经提前实现了，于是编译器可以直接把这个 load 操作优化掉。