计 算 机 科 学 丛 书

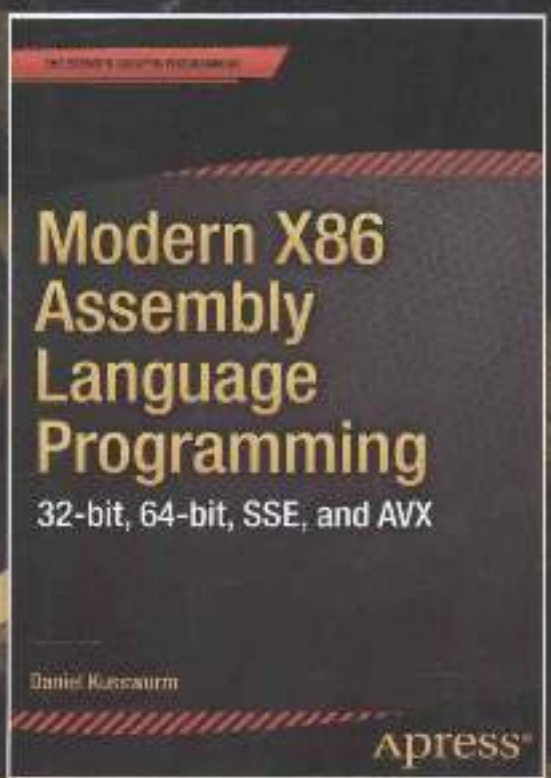
Apress

现代x86汇编语言程序设计

[美] 丹尼尔·卡斯沃姆（Daniel Kusswurm）著
张银奎 罗冰 宋维 张佩 等译

Modern X86 Assembly Language Programming

32-bit, 64-bit, SSE, and AVX



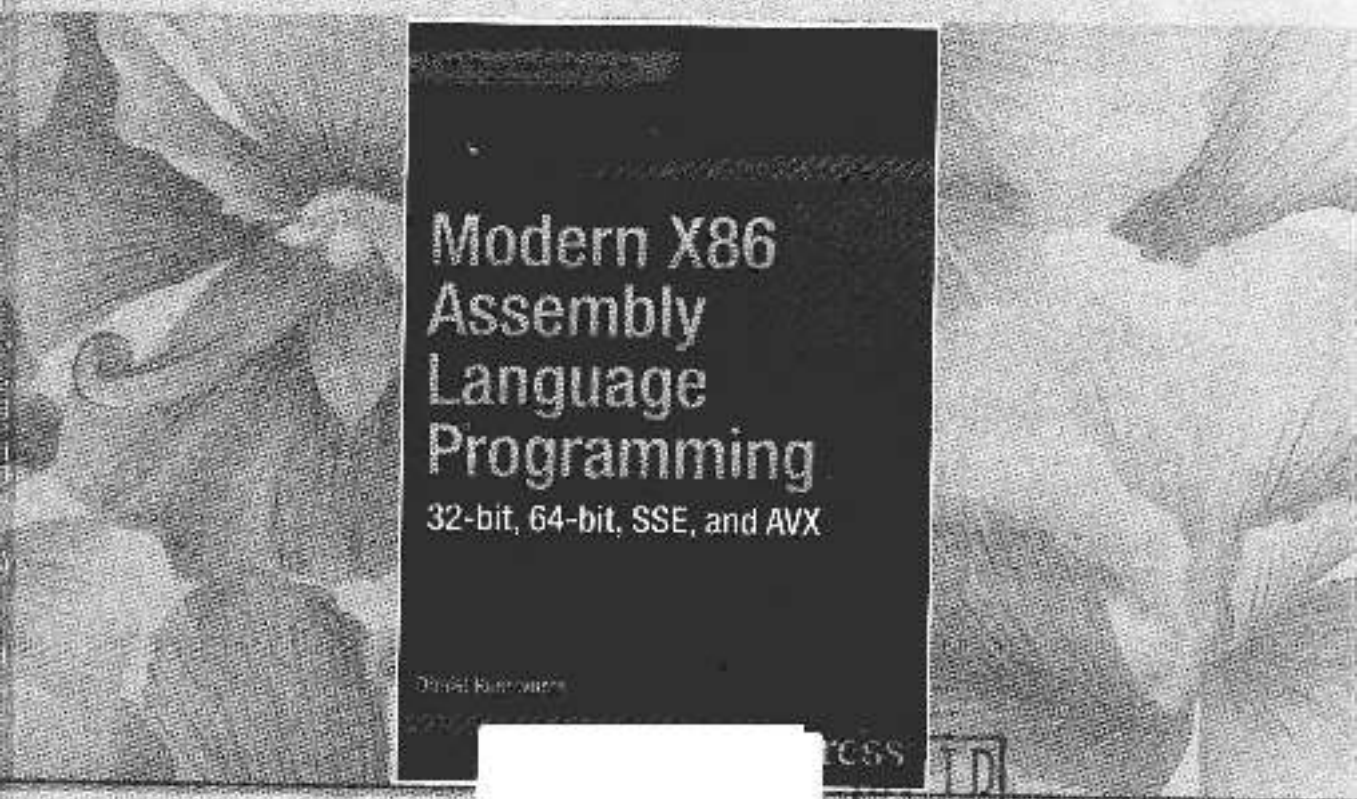
机械工业出版社
China Machine Press

计 [redacted] 丛 书

现代x86汇编语言程序设计

[美] 丹尼尔·卡斯沃姆（Daniel Kusswurm）著
张银奎 罗冰 宋维 张帆 等译

Modern X86 Assembly Language Programming
32-bit, 64-bit, SSE, and AVX



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

现代 x86 汇编语言程序设计 /（美）丹尼尔·卡斯沃姆（Daniel Kusswurm）著；张银奎等译．—北京：机械工业出版社，2016.7
（计算机科学丛书）
书名原文：Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX

ISBN 978-7-111-54278-0

I. 现… II. ①丹… ②张… III. 汇编语言－程序设计 IV. TP313

中国版本图书馆 CIP 数据核字（2016）第 163448 号

本书版权登记号：图字：01-2015-6379

Daniel Kusswurm：Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX（ISBN：978-1-4842-0065-0）.

Original English language edition published by Apress L. P., 2560 Ninth Street, Suite 318, Berkeley, CA 94710 USA. Copyright © 2014 by Daniel Kusswurm. Simplified Chinese-language edition copyright © 2016 by China Machine Press. All rights reserved.

This edition is licensed for distribution and sale in the People's Republic of China only, excluding Hong Kong, Taiwan and Macao and may not be distributed and sold elsewhere.

本书原版由 Apress 出版社出版。

本书简体字中文版由 Apress 出版社授权机械工业出版社独家出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

此版本仅限在中华人民共和国境内（不包括香港、澳门特别行政区及台湾地区）销售发行，未经授权的本书出口将被视为违反版权法的行为。

本书从应用编程的角度解释 x86 处理器的内部架构和执行环境，全面介绍如何用 x86 汇编语言编写可被高级语言调用的函数。主要内容包括：x86-32 核心架构（第 1 章和第 2 章），x87 浮点单元（第 3 章和第 4 章），MMX 技术（第 5 章和第 6 章），流式 SIMD 扩展（第 7 章至第 11 章），高级向量扩展（第 12 章至第 16 章），x86-64 核心架构（第 17 章和第 18 章），x86-64 SSE 和 AVX（第 19 章和第 20 章），高级主题（第 21 章和第 22 章）。书中包含了大量的示例代码，以帮助读者快速理解 x86 汇编语言编程和 x86 平台的计算资源。

本书可作为高等院校计算机及相关专业学生的教材，也可供想要学习 x86 汇编语言编程的软件开发者使用。

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：迟振春　　责任校对：殷　虹

印　　刷：北京市荣盛彩色印刷有限公司　　版　　次：2016 年 10 月第 1 版第 1 次印刷

开　　本：185mm × 260mm　1/16　　印　　张：30.75

书　　号：ISBN 978-7-111-54278-0　　定　　价：79.00 元

凡购本书，如有缺页、倒页、脱页，请本社发行部调换
客服热线：（010）88379991　88361066　　机械热线：（010）88379604
购书热线：（010）68326294　88379648　68995259　　读者信箱：hzjsj@hzbook.com

版权所有 · 侵权必究
开放无防伪样式为盗版
本书法律顾问：北京文成律师事务所　邓元 / 郭晓东

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，不仅奠划了研究的范畴，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，从他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”已经出版了近两百个品种，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

华章教育
华章科技图书出版中心

译者序

Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX

尼采曾经说过："没有可怕的深度，就没有美丽的水面。"这句话或许可以解释我们为什么要翻译这本书。

这本书确实是讨论汇编语言编程的。在这样一个各种脚本语言大行其道的"速成"时代，是否还有必要学习汇编语言呢？对于这个问题，简单回答是或否都可能失于片面。

不妨先讲个小故事。我曾经编写过一个C++程序，以多线程的方式计算Mercator级数。调试版本工作正常之后测试发布版本时发现了一个大问题：负责具体计算任务的工作线程纷纷完成计算并退出后，负责汇总结果的主线程却迟迟不给出结果，而且占用CPU很高。上调试器一看，CPU在以下while语句不停地循环。

```
while (pBoss->m_dwThreadCount != pBoss->m_dwCalcThreads)
{
    ///Sleep(1);
}   // busy wait until all threads are done with computation of partial sums
```

观察while语句中的两个变量，它们显然已经相等了：

```
+0x048 m_dwThreadCount  : 0xa
+0x04c m_dwCalcThreads  : 0xa
```

尝试单步跟踪，CPU始终不离开这行语句，仿佛被钉在这里一样。观察程序指针对应的汇编指令，我不禁愕然：

```
004012f0 ebfe            jmp     MultThrds!CBoxBoss::MasterThreadProc+0x40 (004012f0)
```

这是一条无条件跳转语句，跳转的目标地址居然就是同一条指令的地址。看来CPU在原地打转，根本没有判断两个变量是否相等。但这样说其实不准确，观察当前指令前面的指令，CPU曾经做过比较和判断：

```
004012e6 8b4e4c          mov     ecx,dword ptr [esi+4Ch]
004012e9 8b4648          mov     eax,dword ptr [esi+48h]
004012ec 3bc1            cmp     eax,ecx
004012ee 7402            je      MultThrds!CBoxBoss::MasterThreadProc+0x42 (004012f2)
```

问题的关键是while循环内部没有改变要判断的两个变量，而且定义这两个变量时又忘记使用volatile关键字来修饰，于是编译器在编译发布版本、做自动优化时，为了避免反复读取内存（内存在CPU外部，相对于读寄存器来说访问内存是比较大的开销），便只读取变量一次，也只比较一次，之后便只做跳转而不读取和比较了。增加volatile关键字后这个问题就解决了。

这个小故事说明了两个道理。

第一，汇编是CPU的语言，学会汇编可以为我们打开一扇窗，透过这扇窗我们可以穿越层层阻隔，探视深邃微妙的底层世界，理解CPU的行为，看它是在那个世界里欢快地奔跑还是遭遇陷阱停滞不前。

第二，编译器所做的自动优化有时会出乎我们预料的，可能让我们惊喜，也可能让我们沮丧。要理解编译器的优化行为，懂得汇编语言常常是必需的。

学会汇编语言的另一个好处是当我们对编译器自动优化所达到的效果不满意时，可以直接使用汇编语言编写代码，把性能提高到极致。一个典型的例子就是使用强大的 SIMD 指令（SSE 和 AVX）来优化各种图形图像应用的性能，比如视频编解码和各种机器学习算法。这本书很全面地介绍了 x86 CPU 的各种浮点计算技术，包括经典的 x87 指令，以及最近一些年流行的 SSE 指令和 AVX 指令。单凭这些内容，这本书便不愧“摩登”（Modern）之名。

尼采还说过一句我不愿意接受的话：“书的时代结束了，演员的时代开启了。”在眼下这样一个不是书的时代里，无论是写一本书还是翻译一本书都很艰难。太多干扰让我们无法安静地思考和遣词造句。我的多位格友（格物之友，搜索“格友”公众号可以了解更多）与我一起翻译了这本书，为了能有大块的时间专注于翻译并相互切磋，我们曾两度集结到苏州。第一次住在灵岩山下的木渎古镇。白天在灵岩山下的一个茶馆里挥汗如雨，晚上继续在宾馆里挑灯夜战。中午登上灵岩山，到灵岩寺里吃素面。虽然辛苦，但很充实和快乐。“无图无真相”，贴一张当时的照片吧！



照片中从左至右依次为：罗冰、高异明、宋维、张银奎、崔燚、王卫汉、王科平、张佩。以上格友承担了本书的主要翻译任务，此外，何旻、黎水芬和王建荣也参与了本书的少量翻译和校对工作，在此表示感谢。由于我们的水平有限，难免有翻译不当之处，希望各位读者批评指正。

x86 架构的第一代产品 8086 是从 1976 年开始研发的，距今已有 40 个年头。这 40 年中，基于 x86 CPU 开发的各种产品难以计数，这些产品让这个经典架构传遍了整个世界。学习 x86 汇编语言是了解这一经典架构的最好方式，当我们把一条条指令交给 x86 CPU 执行时，其实就是在和这个经典架构直接“交谈”。在这个交谈过程中，我们可以体会到其中所蕴藏着的智慧，感受到“软件的大美”！

张银奎

2016 年 8 月 16 日于格盈轩

从个人电脑发明那一天起，很多软件开发者就使用汇编语言编程，以解决各种各样的难题。在 PC 时代的早期，用 x86 汇编语言编写大段的程序或整个应用是很普遍的。即便是在 C、C++ 和 C# 等高级语言越来越流行的今天，许多软件开发者也仍然使用汇编语言来编写性能攸关的代码。虽然近些年编译器进步很快，编译出来的机器码变得更短、更快，但在某些情况下，软件开发者还是需要努力发挥汇编语言编程的优势。

现代 x86 处理器包含单指令多数据（SIMD）架构，这给我们提供了另一个持续关注汇编语言编程的原因。SIMD 架构的处理器可以同时计算多个数据，这可以显著提高那些需要实时响应的应用软件的性能。SIMD 架构也非常适合那些计算密集型的领域，比如图像处理、音视频编码、计算机辅助设计、计算机图形学和数据挖掘等。遗憾的是许多高级语言和开发工具不能完全发挥现代 x86 处理器的 SIMD 能力。而汇编语言恰恰可以让软件开发者充分利用处理器的全部计算资源。

现代 x86 汇编语言编程

本书是专门针对 x86 汇编语言编程的一本启发性教材，其主要目的是教你如何用 x86 汇编语言编写可被高级语言调用的函数。本书从应用程序编程的角度来解释 x86 处理器的内部架构。书中包含了非常多的示例代码，帮助你快速理解 x86 汇编语言编程和 x86 平台的计算资源。这本书的主要议题包括：

- x86-32 核心架构、数据类型、内部寄存器、内存寻址模式和基本指令集。
- x87 核心架构、寄存器栈、特殊寄存器、浮点编码和指令集。
- MMX 技术和对组合整数进行计算。
- 流式 SIMD 扩展（SSE）和高级向量扩展（AVX），包括内部寄存器、组合数据类型和浮点运算以及相关指令集。
- x86-64 核心架构、数据类型、内部寄存器、内存寻址模式和基本指令集。
- SSE 和 AVX 技术的 64 位扩展。
- x86 微架构和汇编语言优化技术。

在讨论其他内容之前，我想特别声明一下本书没有覆盖到的内容。本书没有介绍 x86 汇编语言的传统内容，比如 16 位实模式应用和分段内存模型。除了几处历史性的回顾和比较外，所有其他讨论和示例代码都是假定处于 x86 保护模式和平坦线性内存模型下。本书没有讨论 x86 的特权指令和用以支持开发操作系统内核的 CPU 功能，也没有介绍如何用 x86 汇编语言去开发操作系统或者设备驱动程序。不过，如果你真的想用 x86 汇编语言去开发那些系统软件，那么需要先充分理解这本书的内容。

虽然理论上仍然可以完全用汇编语言开发一个应用程序，但是现实中的各种需求使得这种方法很难实行。所以本书重点关注如何创建可被 C++ 调用的 x86 汇编语言模块和函数。本书中的所有示例代码和示例程序都是用微软的 Visual C++ 工具编写并使用微软的宏汇编器编译的。这两个工具都包含在微软的 Visual Studio 开发工具集里面。

目标读者

本书是针对下面几类软件开发者而编写的：

- 在 Windows 平台下开发应用程序并想用 x86 汇编语言提高程序性能的软件开发者。
- 在非 Windows 环境下开发应用程序并想要学习 x86 汇编语言编程的软件开发者。
- 对 x86 汇编语言编程有基本了解，想要学习 x86 的 SSE 和 AVX 指令集的软件开发者。
- 想要或需要更好理解 x86 平台（包括其内部架构和指令集）的软件开发者和计算机学院的学生。

本书主要是针对 Windows 平台上的软件开发者编写的，因为示例代码采用了 Visual C++ 和微软宏汇编编译器。但是，本书并不是一本介绍如何使用微软开发工具的书，非 Windows 平台开发者也可以从本书获益，因为大多数内容的编写和介绍并不依赖任何特别的操作系统。具有 C 和 C++ 编程经验有助于读懂本书的内容和示例代码，但是并不需要读者事先具有 Visual Studio 使用经验，也不需要先学习 Windows API。

内容概要

本书的主要目的是帮助你学习 x86 汇编语言编程。为了达到这个目的，你需要全面理解 x86 处理器的内部架构和执行环境。本书的章节和内容是按照这样的思路规划的。下面简要介绍一下本书的主要议题和各章节的内容。

x86-32 核心架构——第 1 章涵盖了 x86-32 平台的核心架构，讨论了这个平台的基本数据类型、内部架构、指令操作数和内存寻址模式。这一章也简要介绍了 x86-32 的核心指令集。第 2 章讲解了利用 x86-32 核心指令集和常用编程结构编写 x86-32 汇编语言程序的基础知识。第 2 章及其后章节讨论的示例代码都是可以独立运行的程序，这意味着你可以运行、修改或者用这些代码做一些实验来提高学习效果。

x87 浮点单元——第 3 章探讨 x87 浮点单元（FPU）的架构，描述了 x87 FPU 的寄存器栈、控制字寄存器、状态字寄存器和指令集。这一章还深入探讨了用于表达浮点数和某些特殊值的二进制编码方案。第 4 章包含了一些示例，用以演示如何用 x87 FPU 指令集进行浮点运算。对于那些需要维护 x87 FPU 代码或者要在不具有 x86-SSE 和 x86-AVX 的处理器（比如 Intel 的 Quark）上工作的读者来说，本章的内容是最适用的。

MMX 技术——第 5 章描述了 x86 的第一个 SIMD 扩展，即 MMX 技术。它分析了 MMX 技术的架构，包括它的寄存器组、操作数类型和指令集。这一章也讨论了一些相关课题，包括 SIMD 处理概念和组合整型运算。第 6 章包含了用以演示基本 MMX 运算的示例代码，包括组合整型运算（回绕方式和饱和方式）、整数乘法处理和如何正确地在 MMX 和 x87 FPU 代码之间切换。

流式 SIMD 扩展——第 7 章的焦点是流式 SIMD 扩展（SSE）。x86-SSE 为 x86 平台新增了一组 128 位的寄存器，并增加了一系列指令，用以支持不同的数据类型，包括组合整型、组合浮点数（单双精度）和字符串类型的数据。第 7 章还讨论了 x86-SSE 的标量浮点运算功能，对于那些需要进行标量浮点计算的应用程序来说，这个功能可以大大简化算法并提高性能。第 8 章到第 11 章包含了一系列使用 x86-SSE 指令集的示例代码。比如用 x86-SSE 的组合整型去进行图像处理，例如直方图构建和像素阈值化。这些章节也包含了示例代码来演示如何用 x86-SSE 对组合浮点数、标量浮点数和字符串进行计算和处理。

高级向量扩展——第 12 章探讨 x86 最新的 SIMD 扩展——高级向量扩展（AVX）。该章解释了 x86-AVX 执行环境、数据类型和寄存器组以及最新的三目指令格式。同时也讨论了 x86-AVX 的数据广播、收集、排列（permute）功能以及与 x86-AVX 一起引入的扩展，包括混合乘加（fused-multiply-add，FMA）、半精度浮点和新的通用寄存器指令。第 13 章到第 16 章包含了一系列示例代码来演示如何使用 x86-AVX 的各种计算资源，包括对组合整型、组合浮点和标量浮点操作数进行计算的 x86-AVX 指令。这些章节还包含了例子来演示数据广播、收集、排列和 FMA 指令的用法。

x86-64 核心架构——第 17 章探讨的是 x86-64 平台，包括这个平台的核心架构、所支持的数据类型、通用寄存器和状态标志。也解释了为支持 64 位操作数和内存寻址而对 x86-32 平台所做的扩展。这一章最后讨论了 x86-64 指令集，包括那些不再支持的指令。第 18 章用很多示例代码演示了 x86-64 汇编语言编程的基本知识。示例代码包括如何用不同大小的操作数进行整型运算、内存寻址模式、标量浮点运算以及常用的编程结构。第 18 章还介绍了 C++ 调用 x86-64 汇编语言函数的调用约定。

x86-64 SSE 和 AVX——第 19 章描述了如何在 x86-64 平台上使用增强的 x86-SSE 和 x86-AVX，讨论了各自的执行环境和扩展的数据寄存器组。第 20 章包含了在 x86-64 核心架构中使用 x86-SSE 和 x86-AVX 指令集的示例代码。

高级主题——本书最后两章讨论的是与 x86 汇编语言编程相关的高级内容和优化技术。第 21 章讨论了 x86 处理器微架构的关键点，包括它的前端流水线、乱序执行模型和内部执行单元。这一章还讨论了一些编程技术，让你的 x86 汇编语言程序在时间和空间上都很高效。第 22 章包含了几个展现高级汇编语言技术的示例代码。

附录——本书包括三个附录（可从下面的 Apress 网站或华章网站 www.hzbook.com 下载）。附录 A 介绍了使用微软的 Visual C++ 和宏汇编器的简要教程。附录 B 总结了 x86-32 和 x86-64 的调用约定，使用汇编语言编写的函数必须遵守这些约定才可以被 Visual C++ 函数所调用。附录 C 列出了关于 x86 汇编语言编程的参考文献和更多资源。

示例代码要求

可以从 Apress 网站 http://www.apress.com/9781484200650 下载本书的示例代码。编译和运行示例代码的软硬件要求如下：

- 一台包含新近微架构 x86 处理器的个人电脑。所有 x86-32、x87 FPU、MMX 和 x86-SSE 示例代码都可以运行在 Nehalem 或其以后的微架构处理器上。很多示例程序也可以在更老的微架构处理器上执行。AVX 和 AXV2 示例代码分别要求 Sandy Bridge 和 Haswell 微架构处理器。
- 微软 Windows 8.x 或者 Windows 7（Service Pack 1）。x86-64 示例代码需要运行在 64 位 Windows 操作系统上。
- Visual Studio Professional 2013 或者 Visual Studio Express 2013 for Windows Desktop。Express 版本可以从微软的网站 http://msdn.microsoft.com/en-us/vstudio 上免费下载。推荐使用 Update 3 版本。

警告 本书所有示例代码的主要目的是解释这本书中的议题和技术，很少考虑软件工程中的一些重要问题，比如异常的错误处理、安全性、稳定性、含入误差等。若你决定把这些示例代码用在自己的程序里，则应该考虑上述问题。

术语和惯例

这里给本书中使用的术语下个定义。函数、子程序或者过程是一段独立的可执行代码，接受 0 个或更多参数，完成一个操作，并且可以选择性地返回一个值。普通函数被处理器的调用指令所调用。线程是可以被操作系统管理和调度的最小执行单元。任务或者进程包含一个或多个共享同一逻辑内存空间的线程。应用或程序是包含至少一个任务的一个完整软件包。

术语 x86-32 和 x86-64 分别用来描述 x86 处理器的 32 位和 64 位特征、资源以及处理能力。x86 用以代表 32 位和 64 位架构的共有特征。x86-32 模式和 x86-64 模式表示处理器的特定执行环境，它们之间的主要不同是，x86-64 模式支持 64 位寄存器、操作数和内存寻址。x86-SSE 表示流式 SIMD 扩展，x86-AVX 表示高级向量扩展。当讨论特定 SIMD 增强指令时，会使用 SSE、SSE2、SSE3、SSSE3、SSE4、AVX 和 AVX2 这样的英文简称。

其他资源

Intel 和 AMD 提供了大量与 x86 有关的文档。附录 C 列出了许多对初学者和有经验的 x86 汇编语言程序员有用的资源。其中《Intel 64 and IA-32 Architectures Software Developer's Manual-Combined Volumes: 1, 2A, 2B 2C, 3A, 3B and 3C (Order Number: 325462)》的第二卷最为重要。这一卷中对每一条处理器指令都给出了非常全面的信息，包括详细的操作过程、使用的所有操作数、会影响到的状态标志和可能导致的异常。当你开发自己的 x86 汇编语言函数的时候，强烈建议你查阅这个文档来核对指令的用法。

致谢

出版一本书和拍一部电影有很多相似之处。电影的预告片赞美主角的出色表演，书的封面要“鼓吹”一下作者。不论是演员还是作者，他们的努力最终都要接受大众的检阅。而且，无论是出品一部电影还是出版一本书，都离不开执着的精神、高超的技能和富有创造力的专业幕后团队。本书也不例外。

首先我要感谢 Patrick Hauke，在本书的孕育阶段，他便积极支持这个项目并给我非常有价值的建议。还要感谢 Steve Weiss，他的丰富编辑经验指引我们克服了出版中的各种难题。我非常感谢 Melissa Maldonado 的努力，让我和其他每个人专注于本书的编写。我要感谢 Paul Cohen 细致的技术审查和很多切实可行的建议。文字编辑 Kezia Endsley 和校对 Ed Kusswurm 的辛勤工作和富有建设性的反馈都值得敬重和表扬。剩下的任何不完美之处我愿意负全责。

我还要感谢 Dhaneesh Kumar 和整个 Apress 出版社团队的奉献；感谢 Vyacheslav Klochkov 和 Mitch Bodart 指导我如何高效使用 FMA 指令，也要感谢我单位同事的支持和关注。最后，我要感谢我的父母 Armin 和 Mary 以及我的兄弟姐妹 Mary、Tom、Ed 和 John，感谢他们在我写这本书的时候给我很多鼓励。



保罗·科恩（Paul Cohen） 在x86架构的“孩提”时代（从8086开始）便加入了英特尔公司，工作26年后，从市场营销管理岗位上退休。目前他正与道格拉斯科技集团合作，代表英特尔和其他公司开发技术书籍。保罗还与青年企业家学院（YEA）合作，给初高中学生上课，教他们如何成为一个有自信心的真正企业家。同时，他也是俄勒冈州比佛顿市的交通专员和多个非营利组织的董事。

推荐阅读





汇编语言：基于x86处理器（原书第7版）

作者：基普·欧文 译者：贺莲 龚奕利 ISBN：978-7-111-53036-7 定价：99.00元

本书全面接触地讲述了汇编语言程序设计的各个方面，不仅是汇编语言本科课程的经典教材，也可作为计算机系统和体系结构的入门教材。本书专门为32位和64位Intel/Windows平台编写，用通俗易懂的语言描述学生需要掌握的核心概念，首要目标是教授学生编写并调试机器级程序，并帮助他们自然过渡到后续关于计算机体系结构和操作系统的高级课程。本书作者的网站携带了更多的资源和工具，教师和学生可以访问章目标、调试工具、补充文件，以及MASM和Visual Studio 2012的入门教程等。

主要特色

教授有效的设计技巧：自上而下的程序设计演示和讲解，让学生将技术应用于多个编程课程。

理论联系实践：学生将在机器级水平编写软件，为以后在任何操作系统/面向机器的环境中工作做好准备。

灵活的课程安排：教师可结合具体情况以不同的顺序和深度覆盖可选章节上版。

汇编语言（第2版）

作者：郑晓薇 编著 ISBN：978-7-111-44450-3 定价：39.00元

本书作者根据多年讲授汇编语言课程的教学经验以及对汇编语言课程的教学改革，以现代教育理论为基础，精心设计了本书的结构。全书以80X86系列微型计算机为基础，以MASM5.0为汇编上机实验环境，带点介绍Intel8086指令系统。第2版在上版的基础上修订了部分内容，特别是对实验内容进行了改进，增加了两节新的实验，以便适应更多的应用。

本书特色包括：（1）以实例驱动教学。（2）启发式设问引导教学。（3）考虑学习框架。（4）实验让编程穿始终。

出版者的话
译者序
前言
关于技术审校者

第 1 章 x86-32 核心架构 …… 1
1.1 简史 …… 1
1.2 数据类型 …… 3
1.2.1 基本数据类型 …… 3
1.2.2 数值数据类型 …… 4
1.2.3 组合数据类型 …… 5
1.2.4 其他数据类型 …… 6
1.3 内部架构 …… 6
1.3.1 段寄存器 …… 7
1.3.2 通用寄存器 …… 7
1.3.3 EFLAGS 寄存器 …… 8
1.3.4 指令指针 …… 9
1.3.5 指令操作数 …… 9
1.3.6 内存寻址模式 …… 10
1.4 指令集浏览 …… 11
1.4.1 数据传输 …… 13
1.4.2 二进制算术 …… 13
1.4.3 数据比较 …… 14
1.4.4 数据转换 …… 14
1.4.5 逻辑运算 …… 14
1.4.6 旋转和移位 …… 15
1.4.7 字节设置和二进制位串 …… 15
1.4.8 串 …… 16
1.4.9 标志操纵 …… 16
1.4.10 控制转移 …… 17
1.4.11 其他指令 …… 17
1.5 总结 …… 17

第 2 章 x86-32 核心编程 …… 18
2.1 开始 …… 18
2.1.1 第一个汇编语言函数 …… 19
2.1.2 整数系统和表达 …… 22
2.2 x86-32 编程基础 …… 24
2.2.1 调用约定 …… 25
2.2.2 内存寻址模式 …… 28
2.2.3 整数加法 …… 31
2.2.4 条件码 …… 34
2.3 数组 …… 39
2.3.1 一维数组 …… 39
2.3.2 二维数组 …… 42
2.4 结构体 …… 47
2.4.1 简单结构体 …… 47
2.4.2 动态结构体创建 …… 50
2.5 字符串 …… 52
2.5.1 字符计数 …… 52
2.5.2 字符串拼接 …… 54
2.5.3 比较数组 …… 57
2.5.4 反转数组 …… 60
2.6 总结 …… 62

第 3 章 x87 浮点单元 …… 63
3.1 x87 FPU 核心架构 …… 63
3.1.1 数据寄存器 …… 63
3.1.2 x87 FPU 专用寄存器 …… 64
3.1.3 x87 FPU 操作数和编码 …… 65
3.2 x87 FPU 指令集 …… 68
3.2.1 数据传输 …… 68
3.2.2 基本运算 …… 69
3.2.3 数据比较 …… 70
3.2.4 超越函数 …… 71
3.2.5 常量 …… 71
3.2.6 控制 …… 72
3.3 总结 …… 72

第 4 章 x87 FPU 编程 …… 73
4.1 x87 FPU 编程基础 …… 73

4.1.1 简单计算 …… 73
4.1.2 浮点比较 …… 76
4.2 x87 FPU 高级编程 …… 79
4.2.1 浮点数组 …… 79
4.2.2 超越指令（超越函数指令）…… 84
4.2.3 栈的高级应用 …… 87
4.3 总结 …… 92

第 5 章 MMX 技术 …… 93

5.1 SIMD 处理概念 …… 93
5.2 回绕和饱和运算 …… 94
5.3 MMX 执行环境 …… 95
5.4 MMX 指令集 …… 96
5.4.1 数据传输 …… 97
5.4.2 算术运算 …… 97
5.4.3 比较 …… 98
5.4.4 转换 …… 99
5.4.5 逻辑和位移 …… 99
5.4.6 解组和重排 …… 99
5.4.7 插入和提取 …… 100
5.4.8 状态和缓存控制 …… 100
5.5 总结 …… 100

第 6 章 MMX 技术编程 …… 101

6.1 MMX 编程基础 …… 101
6.1.1 组合整型加法 …… 102
6.1.2 组合整型移位 …… 108
6.1.3 组合整型乘法 …… 111
6.2 MMX 高级编程 …… 113
6.2.1 整数数组处理 …… 114
6.2.2 使用 MMX 和 x87 FPU …… 120
6.3 总结 …… 125

第 7 章 流式 SIMD 扩展 …… 126

7.1 x86-SSE 概览 …… 126
7.2 x86-SSE 执行环境 …… 127
7.2.1 x86-SSE 寄存器组 …… 127
7.2.2 x86-SSE 数据类型 …… 128
7.2.3 x86-SSE 的控制－状态寄存器 …… 128
7.3 x86-SSE 处理技术 …… 129
7.4 x86-SSE 指令集概览 …… 132
7.4.1 标量浮点数据传输 …… 133
7.4.2 标量浮点算术运算 …… 133
7.4.3 标量浮点比较 …… 134
7.4.4 标量浮点转换 …… 134
7.4.5 组合浮点数据传输 …… 135
7.4.6 组合浮点算术运算 …… 135
7.4.7 组合浮点比较 …… 136
7.4.8 组合浮点转换 …… 136
7.4.9 组合浮点重排和解组 …… 137
7.4.10 组合浮点插入和提取 …… 137
7.4.11 组合浮点混合 …… 137
7.4.12 组合浮点逻辑 …… 138
7.4.13 组合整数扩展 …… 138
7.4.14 组合整数数据传输 …… 138
7.4.15 组合整数算术运算 …… 139
7.4.16 组合整数比较 …… 139
7.4.17 组合整数转换 …… 139
7.4.18 组合整数重排和解组 …… 140
7.4.19 组合整数插入和提取 …… 140
7.4.20 组合整数混合 …… 141
7.4.21 组合整数移位 …… 141
7.4.22 文本字符串处理 …… 141
7.4.23 非临时数据传输和缓存控制 …… 142
7.4.24 其他 …… 142
7.5 总结 …… 143

第 8 章 x86-SSE 编程——标量浮点 …… 144

8.1 标量浮点运算基础 …… 144
8.1.1 标量浮点算术运算 …… 144
8.1.2 标量浮点数的比较 …… 148
8.1.3 标量浮点数的类型转换 …… 151
8.2 高级标量浮点编程 …… 157
8.2.1 用标量浮点指令计算球体表面积和体积 …… 157
8.2.2 用标量浮点指令计算平行四边形面积和对角线长度 …… 159

8.3 总结 …… 165

第 9 章 x86-SSE 编程——组合浮点 …… 166

9.1 组合浮点运算基础 …… 166
9.1.1 组合浮点算术运算 …… 167
9.1.2 组合浮点数的比较 …… 171
9.1.3 组合浮点数的类型转换 …… 175
9.2 高级组合浮点编程 …… 178
9.2.1 组合浮点数最小二乘法 …… 178
9.2.2 用组合浮点数进行 4×4 矩阵的计算 …… 183
9.3 总结 …… 192

第 10 章 x86-SSE 编程——组合整数 …… 193

10.1 组合整数基础 …… 193
10.2 高级组合整数编程 …… 197
10.2.1 组合整数立方阵 …… 197
10.2.2 组合整数图像分割 …… 203
10.3 总结 …… 214

第 11 章 x86-SSE 编程——字符串 … 215

11.1 字符串基础知识 …… 215
11.2 字符串编程 …… 221
11.2.1 计算字符串长度 …… 221
11.2.2 字符串转换 …… 224
11.3 总结 …… 231

第 12 章 AVX——高级向量扩展 …… 232

12.1 x86-AVX 概述 …… 232
12.2 x86-AVX 执行环境 …… 233
12.2.1 x86-AVX 寄存器组 …… 233
12.2.2 x86-AVX 数据类型 …… 235
12.2.3 x86-AVX 指令语法 …… 234
12.3 x86-AVX 功能扩展 …… 235
12.4 x86-AVX 指令集概述 …… 236
12.4.1 升级版的 x86-SSE 指令 …… 236
12.4.2 新指令 …… 239
12.4.3 功能扩展指令 …… 242
12.5 总结 …… 245

第 13 章 x86-AVX 标量浮点编程 …… 246

13.1 编程基础 …… 246
13.1.1 标量浮点运算 …… 246
13.1.2 标量浮点比较 …… 248
13.2 高级编程 …… 253
13.2.1 一元二次方程的根 …… 253
13.2.2 球面标系 …… 258
13.3 总结 …… 263

第 14 章 x86-AVX 组合浮点编程 …… 264

14.1 编程基础 …… 264
14.1.1 组合浮点运算 …… 265
14.1.2 组合浮点比较 …… 269
14.2 高级编程 …… 272
14.2.1 相关系数 …… 272
14.2.2 超等列均值 …… 278
14.3 总结 …… 283

第 15 章 x86-AVX 组合整型编程 …… 284

15.1 组合整型基础 …… 284
15.1.1 组合整型运算 …… 284
15.1.2 组合整数解包操作 …… 288
15.2 高级编程 …… 292
15.2.1 图像等亮裁剪 …… 293
15.2.2 图像阈值二分法 …… 299
15.3 总结 …… 307

第 16 章 x86-AVX 编程——新指令 … 308

16.1 检测处理器特性（CPUID） …… 308
16.2 数据操作指令 …… 314
16.2.1 数据广播 …… 314
16.2.2 数据混合 …… 317
16.2.3 数据排列 …… 322
16.2.4 数据收集 …… 326
16.3 融合乘加编程 …… 331
16.4 通用寄存器指令 …… 339
16.4.1 不影响标志位的乘法和移位操作 …… 339

16.4.2 增强的位操作 …… 342
16.5 总结 …… 345

第 17 章 x86-64 核心架构 …… 346

17.1 内部架构 …… 346
17.1.1 通用寄存器 …… 347
17.1.2 RFLAGS 寄存器 …… 348
17.1.3 指令指针寄存器 …… 348
17.1.4 指令操作数 …… 348
17.1.5 内存寻址模式 …… 349
17.2 x86-64 和 x86-32 的区别 …… 350
17.3 指令集概览 …… 351
17.3.1 基本指令使用 …… 351
17.3.2 系统指令 …… 352
17.3.3 新指令 …… 352
17.3.4 只被间接使用的资源 …… 353
17.4 总结 …… 353

第 18 章 x86-64 核心编程 …… 354

18.1 x86-64 编程基础 …… 354
18.1.1 整数算术运算 …… 355
18.1.2 内存寻址 …… 359
18.1.3 整型操作数 …… 362
18.1.4 浮点数运算 …… 365
18.2 x86-64 调用约定 …… 369
18.2.1 基本技术 …… 369
18.2.2 使用非易变寄存器 …… 372
18.2.3 使用非易变类型 XMM 寄存器 …… 376
18.2.4 简化序言和结语的案 …… 381
18.3 x86-64 数组和字符串 …… 386
18.3.1 二维数组 …… 386
18.3.2 字符串 …… 390
18.4 总结 …… 393

第 19 章 x86-64 单指令多数据流架构 …… 394

19.1 x86-SSE-64 执行环境 …… 394
19.1.1 x86-SSE-64 寄存器组 …… 394
19.1.2 x86-SSE-64 数据类型 …… 394
19.1.3 x86-SSE-64 指令集概述 …… 395
19.2 x86-AVX 执行环境 …… 395
19.2.1 x86-AVX-64 寄存器组 …… 395
19.2.2 x86-AVX-64 数据类型 …… 396
19.2.3 x86-AVX-64 指令集概述 …… 396
19.3 总结 …… 396

第 20 章 x86-64 单指令多数据流编程 …… 397

20.1 x86-SSE-64 编程 …… 397
20.1.1 直方图绘制 …… 397
20.1.2 图像转换 …… 402
20.1.3 向量数组 …… 410
20.2 x86-AVX-64 编程 …… 417
20.2.1 椭圆体计算 …… 417
20.2.2 RGB 图像处理 …… 421
20.2.3 矩阵求逆 …… 426
20.2.4 其他指令 …… 437
20.3 总结 …… 441

第 21 章 高级主题和优化技巧 …… 442

21.1 处理器微架构 …… 442
21.1.1 多核处理器概述 …… 442
21.1.2 超长构流水线功能 …… 443
21.1.3 执行引擎 …… 445
21.2 优化汇编语言代码 …… 446
21.2.1 基本优化 …… 446
21.2.2 浮点算术 …… 447
21.2.3 程序分支 …… 447
21.2.4 数据对齐 …… 448
21.2.5 SIMD 技巧 …… 449
21.3 总结 …… 449

第 22 章 高级主题编程 …… 450

22.1 无时态内存存储 …… 450
22.2 数据预取 …… 455
22.3 总结 …… 463

索引 …… 464

x86-32 核心架构

本章将从应用程序的角度解析 x86-32 核心架构。我们会从 x86 平台的简要历史开始介绍，以便为后续的讨论提供一个参考框架。接下来会回顾 x86 的数据类型，包括基本类型、数字类型和组合类型。而后，我们将深入挖掘 x86-32 的内部架构细节，包括执行单元、通用寄存器、状态标志、指令操作数和内存寻址模式。本章的最后一部分是对 x86-32 指令集的概览。

与 C 和 C++ 这样的高级语言不同，汇编语言编程需要软件开发者在写代码之前比较全面地理解目标处理器的架构特征。这一章的内容将满足大家的这一需要，并为理解第 2 章中的示例代码奠定基础。本章也为理解第 17 章将讨论的 x86-64 核心架构准备了一些基础。

1.1 简史

在深入解析 x86-32 平台的技术细节之前，让我们先来看一下它的简要历史，这将有助于理解这个架构是如何随着时间而演变的。在下面将要介绍的回顾内容中，我们将把主要精力放在那些对使用 x86 汇编语言的软件开发者产生重大影响的处理器产品和架构变化上。需要对 x86 产品线有更全面理解的读者请参考附录 C 所列出的资源。

x86-32 平台的第一个版本是 1985 年问世的英特尔 80386 微处理器。80386 扩展了其 16 位前身的架构，新增的特性包括 32 位宽的寄存器和数据类型、平坦地址模型、4GB 的逻辑地址空间以及分页虚拟内存。80486 处理器通过增加芯片上高速缓存（cache）和指令优化提高了性能。另外，大多数版本的 80486 CPU 都包含集成的 x87 浮点单元（FPU），不再像 80386 那样需要和分离的 80387 浮点单元协同工作。

1993 年推出的奔腾处理器进一步扩展了 x86-32 架构，并创了被称为 P5 的微架构。微架构定义了处理器内部单元的组织结构，包括寄存器文件、执行单元、指令流水线、数据总线和高速缓存。一种微架构常常被多个产品线的处理器所使用。P5 微架构在性能方面的增强包括并行的（两条）指令执行流水线、64 位外部数据总线，并把芯片上高速缓存分成指令缓存和数据缓存两种。P5 微架构的后续版本包含了一种新的计算资源，称为 MMX（1997 年）。MMX 技术支持使用 64 位宽的寄存器对组合整型做单指令多数据（SIMD）运算。

1995 年推出的奔腾 Pro 处理器和 1997 年推出的奔腾 II 处理器都使用了 P6 微结构。它所采用的三路超标量设计进一步扩展了 x86-32 平台。这种设计意味着可以在一个时钟周期内解码、分发和执行三条不同的指令（平均）。P6 微架构的其他改变包括支持指令的乱序执行、改进的分支预测算法以及投机执行。也是基于 P6 微架构的奔腾 III 处理器是在 1999 年发布的。它包含了一种新的 SIMD 技术，被称为流式 SIMD 扩展（streaming SIMD extension，SSE）。SSE 为 x86-32 平台增加了 8 个 128 位宽的寄存器以及支持组合单精度（32 位）浮点运算的指令。

2000 年，英特尔推出了一种名为 Netburst 的新型微架构，它包含了扩展 SSE 浮点能

力的 SSE2 技术。SSE2 支持组合双精度数据，还引入了新的指令让编程者可以使用 128 位的 SSE 寄存器来做组合整型运算和标量浮点运算。奔腾 4 处理器和它的几个变种都是基于 Netburst 微架构的。2004 年的升级版本 Netburst 微架构包含了 SSE3 和超线程技术。SSE3 增加了组合整数指令和组合浮点指令。超线程技术把处理器的前端指令流水线并行化以提高性能。支持 SSE3 的处理器包括 90nm（或者更小）版本的奔腾 4 处理器以及面向服务器的至强（Xeon）产品线。

2006 年，英特尔推出了称为酷睿（Core）的微架构。酷睿微架构包含了重新设计的 Netburst 前端流水线和执行单元，以便提高性能和降低功耗。它还引入了很多 x86-SSE 增强，包括 SSSE3 和 SSE4.1。这些扩展增加了新的组合整数指令和组合浮点指令，但没有增加新的寄存器和数据类型。基于酷睿微架构的处理器包括从酷睿 2 双核到酷睿 2 四核各个系列的 CPU 以及至强 3000/5000 系列。

在 2008 年年末，一种称为 Nehalem 的微架构接替了酷睿微架构。Nehalem 微架构把酷睿微架构去掉的超线程技术重新加入到 x86 平台。它还包含了 SSE4.2。这次对 x86-SSE 的最后一次增强加入了几条应用相关的加速指令。SSE4.2 还包含四条新的指令，可以使用 128 位宽的 x86-SSE 寄存器来辅助字符串处理。基于 Nehalem 微架构的处理器包括第一代的酷睿 i3、i5 和 i7 CPU 以及至强 3000、5000 和 7000 系列的 CPU。

在 2011 年，英特尔发布了 Sandy Bridge 微架构。Sandy Bridge 微架构引入了一种新的 x86 SIMD 技术，被称为高级向量扩展（Advanced Vector Extension），简称 AVX。AVX 增加了使用 256 位宽的寄存器的组合浮点运算（单精度双精度都支持）。AVX 还支持一种新的
2 指令格式，可以有三个操作符，这有助于减少在寄存器之间传递数据的次数。基于 Sandy Bridge 微架构的处理器包括第二代和第三代的酷睿 i3、i5 和 i7 CPU 以及至强的 E3、E5 和 E7 系列的 CPU。

在 2013 年，英特尔公布了 Haswell 微架构。Haswell 包含了 AVX2，对 AVX 技术做了扩展，可以使用 256 位宽的寄存器做组合整数运算。AVX2 还引入了一系列广播、收集和排列数据的指令集，用于完成高级的数据传输功能。Haswell 微架构的另一大特征是包含了对 FMA 运算（fused-multiply-add，融合乘加）的支持。使用 FMA，编程者只要用一条浮点指令就可以完成连续的求和运算和求积运算。Haswell 微架构还包含了几条新的通用寄存器指令。基于 Haswell 微架构的处理器包括第四代的酷睿 i3、i5 和 i7 CPU 以及至强 E3（v3）系列 CPU。

过去这些年 x86 平台的发展并不限于 SIMD 增强。2003 年，AMD 引入了 Opteron 处理器，它把 x86 的核心架构从 32 位扩展到了 64 位。2004 年英特尔如法炮制，从奔腾 4 的某些版本开始加入了本质上相同的 64 位扩展。所有基于酷睿、Nehalem、Sandy Bridge 和 Haswell 微架构的处理器都支持 x86-64 执行环境。

过去这些年中，英特尔还引入了几种针对特定应用优化的专用微架构。Bonnell 微架构是其中的第一个，2008 年发布的最初版本阿童木（Atom）处理器就是基于 Bonnell 微架构的。基于这种微架构的阿童木处理器支持 SSSE3。在 2013 年，英特尔推出了名为 Silvermont 的片上系统（System on a Chip，SoC）微架构，这一微架构专门针对智能手机和平板 PC 这样的移动设备做了优化。Silvermont 微架构也被应用到针对特定应用裁剪的处理器中，包括小型服务器、存储设备、网络通信设备以及嵌入式系统。基于 Silvermont 微架构的处理器包含了 SSE4.2，但缺少 x86-AVX。2013 年，英特尔还公布了一种超低功耗的 SoC 微架构，称

为夸克（Quark），专门为物联网（Internet-of-Things，IoT）和可穿戴设备而设计。基于夸克微架构的处理器只支持核心的 x86-32 和 x87 浮点指令集，不具有 x86-64 处理功能，也没有 MMX、x86-SSE 和 x86-AVX 技术所提供的任何 SIMD 功能。

最近一些年里，AMD 的处理器也在发展。在 2003 年，AMD 推出了一系列基于 K8 微架构的处理器。最初版本的 K8 包含了对 MMX、SSE 和 SSE2 的支持，后续版本加入了 SSE3。2007 年发布的 K10 微架构包含了被称为 SSE4a 的 SIMD 增强。SSE4a 包含了几条英特尔处理器不支持的屏蔽移位和流式存储指令。K10 之后，AMD 在 2010 年引入了一种新的名为 Bulldozer 的微架构。Bulldozer 微架构包含了 SSE3、SSE4.1、SSE4.2、SSE4a 以及 AVX 技术。它还加入了 FMA4，FMA4 是融合乘加（fused-multiply-add）的“四操作数”版本。与不支持 SSE4a 的情况类似，英特尔处理器中也没有 FMA4。继 Bulldozer 微架构之后，2012 年，AMD 推出的 Piledriver 微架构既包含了 FMA4，也包含了 FMA 的“三操作数”版本，一些 CPU 特征检测工具和第三方的文档将其称为 FMA3。

1.2 数据类型

x86-32 核心架构支持的数据类型很广泛，其中的大部分都是从少数的基本数据类型衍生出来的。应用程序最常操作的数据类型包括有符号和无符号整数、标量的单精度浮点和双精度浮点数、字符、文本串和组合数值（packed value）。这一节将比较详细地介绍这些数据类型以及 x86 支持的其他数据类型。

1.2.1 基本数据类型

基本数据类型是处理器执行程序时操纵的基本数据单位。x86 平台全面支持从 8 位（1 字节）到 256 位（32 字节）这一广泛区间内的基本数据类型。表 1-1 列出了这些类型以及它们的典型用途。

表 1-1 x86 的基本数据类型

数据类型	位数	典型用途
字节	8	字符、整数、二进制编码（BCD）的数值
字	16	字符、整数
双字	32	整数、单精度浮点数
四字	64	整数、双精度浮点数、组合整数
五字（Quintword）	80	双扩展精度浮点数、组合 BCD
双四字	128	组合整数、组合浮点数
四四字	256	组合整数、组合浮点数

不足为奇，大多数基本数据类型的长度都是 2 的整数幂。主要的特例是 80 位宽的“五字”（Quintword），它主要是用在 x87 浮点单元（FPU）中支持双扩展精度浮点数和组合 BCD 数值。

我们经常从右到左对基本数据类型的二进制位进行编号，0 和长度 -1 分别用来代表最低位和最高位。长度大于 1 个字节的基本数据类型在内存中按照最低位字节在最低地址的顺序连续存储。这种方式的内存布局被称为小端存储（little endian）。图 1-1 列出了基本数据类型的位编号和字节顺序。

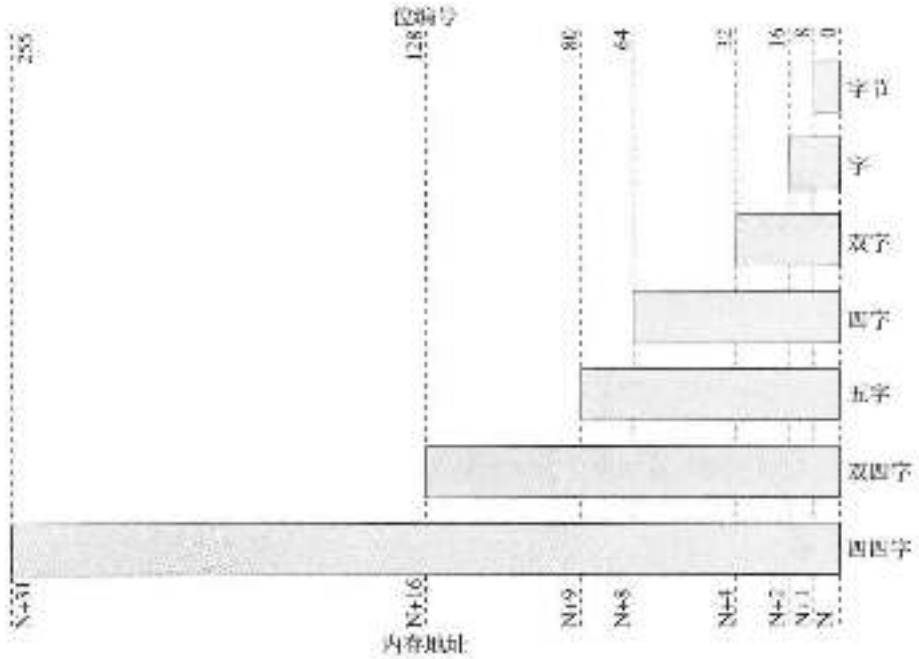


图 1-1 基本数据类型的位编号和字节顺序

如果基本数据类型（数据）的内存地址可以被它的长度（字节数）整除，那么这个数据就是内存对齐的。举例来说，如果一个双字整数的地址是可以被 4 整除的，那么它就是内存对齐的。类似地，如果“四字”整数的地址可以被 8 整除，那么它也是内存对齐的。除非操作系统特别启用，x86 处理器一般情况下不要求多字节数据类型必须内存对齐。一个例外是 x86-SSE 和 x86-AVX 指令集，它们通常要求按“双四字”(double quadword) 和“四四字”(quad quadword) 进行内存对齐。第 7 章和第 12 章将详细讨论 x86-SSE 和 x86-AVX 操作数的对齐需求。不论硬件是否有强制内存对齐规则，我们都强烈建议尽最大可能让多字节基本数据类型内存对齐，以避免处理器访问不对齐的数据时所产生的性能损失。

1.2.2 数值数据类型

数值数据类型是像整数或者浮点数这样的基本数值。CPU 能够识别的所有数值数据类型是
[5] 使用前一小节讨论的基本数据类型表示的。可以把数值数据类型分为两类：标量的和组合的。

标量数据类型用来做离散数值的运算。x86 平台支持一系列标量数据类型，有点像 C/C++ 中的基本数据类型。表 1-2 列出了这些数据类型。x86-32 指令集对 8 位、16 位和 32 位的标量整数运算具有内建的支持，包括有符号和无符号。一些指令也能操作 64 位的数值。不过完整的 64 位数值支持需要 x86-64 模式。

表 1-2 x86 数值数据类型

类 型	位数	等价的 C/C++ 类型
有符号整数	8	char
	16	short
	32	int, long
	64	long long

（续）

类　型	位数	等价的 C/C++ 类型
无符号整数	8	unsigned char
	16	unsigned short
	32	unsigned int, unsigned long
	64	unsigned long long
浮点数	32	float
	64	double
	80	long double

x87 FPU 支持三种不同的标量浮点编码方法，覆盖从 32 位到 80 位的各种浮点数。x86 汇编语言函数可以任意选用这些编码方法。但需要指出的是，C/C++ 中的 80 位双扩展精度浮点数的编码方法并不统一的。有些编译器对 double 和 long double 都使用 64 位编码。第 3 章介绍 x87 FPU 和它支持的数据类型时将详细讨论这些细节。

1.2.3 组合数据类型

x86 平台支持很多种组合数据类型（packed data type），用来对整数或者浮点数做 SIMD 计算。举例来说，一个 64 位宽的组合数据类型可以容纳八个 8 位整数、四个 16 位整数或者两个 32 位整数。一个 256 位宽的组合数据类型可以容纳很多种数据组合，包括 32 个 8 位整数、8 个单精度浮点数或者 4 个双精度浮点数。表 1-3 列出了有效的组合数据类型以及它们可以容纳的数据元素类型和最大数据元素个数。

6

表 1-3　组合数据类型

组合位数	数据元素类型	元素数量
64	8 位整数	8
	16 位整数	4
	32 位整数	2
128	8 位整数	16
	16 位整数	8
	32 位整数	4
	64 位整数	2
	单精度浮点数	4
	双精度浮点数	2
256	8 位整数	32
	16 位整数	16
	32 位整数	8
	64 位整数	4
	单精度浮点数	8
	双精度浮点数	4

正如本章前面所讨论的，在过去几年中，x86 平台一直在加入对 SIMD 的增强，从 MMX 技术到最近的 AVX2。这些增强的一个问题是表 1-3 中描述的组合数据类型和与其关联的指令并不被所有处理器支持。使用 x86 汇编语言编程的开发者需要记住这一点。幸运的是，有方法在运行期检查处理器的具体支持情况。

1.2.4 其他数据类型

x86 平台还支持其他几种零散的数据类型，包括串（string）、位域（bit field）、二进制位串以及二进制编码十进制数（BCD）。

所谓 x86 串，就是一块连续的字节、字或者双字，用来支持基于文本的数据类型和处理操作。举例来说，C/C++ 中的 char 和 wchar_t 通常会分别用 x86 的字节和字来实现。也可以用 x86 串来处理数组、位图以及类似的连续数据块。x86 指令集提供了一系列指令对串进行比较、加载、移动、扫描和存储操作。

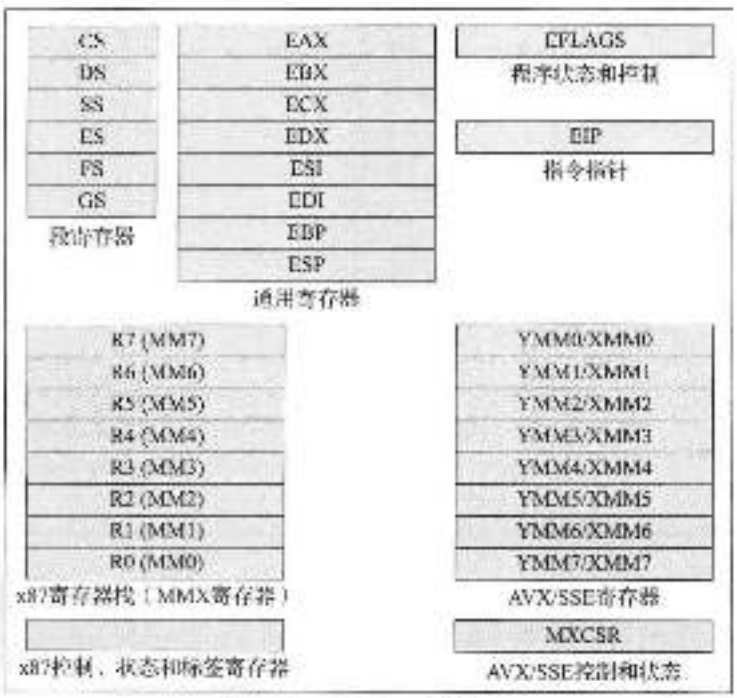
位域（bit field）是一串连续的二进制位，一些指令用它来作掩码值。一个位域可以从一
7 个字节的任意位开始，最多包含 32 个二进制位。

二进制位串最可以最多包含 $2^{32}-1$ 个二进制位的连续二进制位序列。x86 指令集包含了很多条指令来对二进制位串中的二进制位进行清零、置 1、扫描和测试（test）操作。

最后介绍一下二进制编码十进制数（binary-coded decimal，BCD）。它是使用 4 位无符号整数来表示的十进制数（0 – 9）。x86-32 指令集的指令可以对组合 BCD 数据（每个字节 2 个 BCD 数字）或者非组合 BCD 数据（每个字节 1 个 BCD 数字）进行基本运算。x87 FPU 也能够从内存中加载 80 位的 BCD 数，也可以把它们存回内存。

1.3 内部架构

从程序运行的角度来看，可以把一个 x86-32 处理器的内部架构划分为几个独立的执行单元，包括核心执行单元、x87 FPU 以及 SIMD 执行单元。很显然，任何程序都需要使用核心执行单元所提供的计算资源。是否使用 x87 FPU 和 SIMD 执行单元是根据需要选择的。图 1-2 勾勒出了 x86-32 处理器的内部架构。



8 图 1-2 x86-32 内部架构

这一节的余下部分将详细介绍 x86-32 的核心执行单元。我们将从这个单元的寄存器组开始，包括段寄存器、通用寄存器、状态标志寄存器和指令指针。接下来我们会讨论指令的操作数以及访问内存的寻址模式。其他执行单元将在本书的后续章节中讨论。第 3 章将探索 x87 FPU 的内部架构，第 5、7 和 12 章将分别深入探讨 MMX、x86-SSE 和 x86-AVX。

1.3.1 段寄存器

x86-32 核心执行单元使用段寄存器定义的逻辑内存类型供程序执行和存储数据使用。x86 处理器包含 6 个段寄存器，用来标记代码、数据和栈空间的内存块。当在 x86-32 保护模式执行时，段寄存器中存放的是段选择子，以其为索引可以在段描述符表中找到一个段描述符，段描述符里定义了段的各种操作属性。段的操作属性包括大小、类型（代码或者数据）以及访问权限（读或者写）。段寄存器的初始化和管理一般是由操作系统负责的。大多数 x86-32 应用程序都不需要去关心段寄存器是如何配置的。

1.3.2 通用寄存器

x86-32 核心执行单元包含 8 个 32 位的通用寄存器。这些寄存器主要用来做逻辑、算术和地址计算。也可以用它们来临时存放数据或者指向内存数据的指针。图 1-3 完整列出了所有通用寄存器，包括它们的名字（用来指定指令操作数）。除了支持 32 位操作数外，也可以使用通用寄存器来存放 8 位或者 16 位操作数。例如，一个函数可以使用寄存器 AL、BL、CL 和 DL 来访问 EAX、EBX、ECX 和 EDX 的最低字节（8 位）。类似的，可以使用 AX、BX、CX 和 DX 来访问低 16 位字。

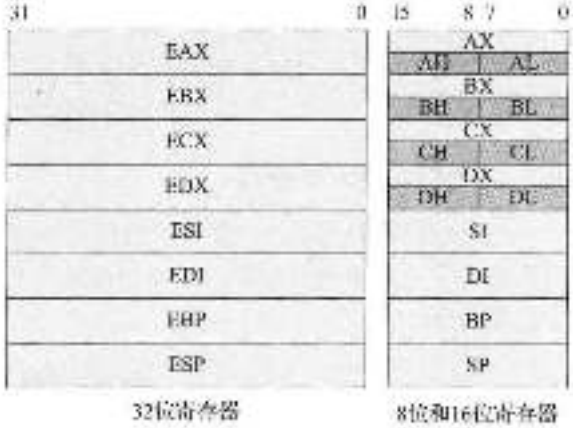


图 1-3 x86-32 通用寄存器

尽管名字叫通用寄存器，但是 x86-32 指令集还是对它们的用法强加了一些约束。一些指令需要或隐含使用特定的寄存器作为操作数。举例来说，某些 imul（有符号乘法）和 idiv（有符号除法）指令使用 EDX 寄存器来存放乘积或者被除数的高位双字。串指令需要把源操作数和目标操作数的地址分别放到 ESI 和 EDI 寄存器中。包含循环前缀的串指令必须使用 ECX 作为计数寄存器，而很多二进制移位和循环指令必须把位计数值加载到 CL 寄存器中。

x86-32 处理器使用 ESP 寄存器来支持栈有关的操作，比如函数调用和返回。栈本身只不过是操作系统分配给进程或者线程的一段连续内存空间。应用程序也可以使用栈来传递函数的参数以及存放临时数据。ESP 寄存器总是指向栈上的最顶一项。尽管可以把 ESP 寄存器当作通用寄存器来使用，但是这种用法是不切实际的，强烈建议不要这样做。通常把 EBP 寄存器用作基址指针（base pointer）来访问存储在栈上的数据项（也可以使用 ESP 寄存器做基址指针来访问栈上的数据）。当不用 EBP 做基址指针时，可以把它当作通用寄存器来使用。

一些指令对特定寄存器的隐含或者固定用法源自 x86 处理器的传统设计模式，可以追溯

到8086寄存器，其目的是为了提高代码的密度。从现代编程的角度来看，这意味着开发者在编写x86-32汇编程序时必须对这些传统的寄存器使用协议格外小心。表1-4列出了通用
[10] 寄存器的传统用法。

表1-4 通用寄存器的传统用法

寄存器	传统用法
EAX	累加器
EBX	内存指针，基址寄存器
ECX	循环控制，计数器
EDX	整数乘法，整数除法
ESI	串指令的源指针，索引寄存器
EDI	串指令的目标指针，索引寄存器
ESP	栈指针
EBP	栈帧基址指针

有几点需要解释一下：表1-4列出的传统用法只是一般情况，并不是强制的。举例来说，如果应用程序把ECX寄存器用作内存指针，那么x86-32指令集也并不阻止这样做，虽然ECX的传统用法是计数器。而且，x86汇编器也不强制这些传统用法。考虑到x86-32模式下只有很有限的通用寄存器可以使用，以非传统方式使用通用寄存器在很多时候是必需的。最后需要说明的是，表1-4列出的传统用法与C++等高级编程语言中定义的调用约定并不是一回事。第2章将进一步讨论和观察调用约定。

1.3.3 EFLAGS寄存器

EFLAGS寄存器包含一系列状态位，处理器使用它们来指示逻辑或者算术运算的结果。它还包含了一组系统控制位，主要供操作系统使用。表1-5列出了EFLAGS寄存器中的各
[11] 个二进制位。

表1-5 EFLAGS寄存器

位	名称	符号	用法
0	进位标志	CF	状态
1	保留		1
2	奇偶校验标志	PF	状态
3	保留		0
4	辅助进位标志	AF	状态
5	保留		0
6	零标志	ZF	状态
7	符号标志	SF	状态
8	陷阱标志	TF	系统
9	中断启用标志	IF	系统
10	方向标志	DF	控制
11	溢出标志	OF	状态
12	I/O特权级别位0	IOPL	系统
13	I/O特权级别位1	IOPL	系统

（续）

位	名称	符号	用法
14	任务嵌套	NT	系统
15	保留		0
16	恢复标志	RF	系统
17	虚拟 8086 模式	VM	系统
18	对齐检查	AC	系统
19	虚拟中断标志	VIF	系统
20	虚拟中断悬而未决	VIP	系统
21	ID 标志	ID	系统
22 ~ 31	保留		0

对应用程序来说，EFLAGS 寄存器中最重要的位是以下状态标志：辅助进位标志（AF）、进位标志（CF）、溢出标志（OF）、奇偶校验标志（PF）、符号标志（SF）以及 0 标志（ZF）。辅助进位标志用来指示 BCD 加减法的进位或者借位情况。当进行无符号整数运算时，如果发生溢出，处理器会设置进位标志。一些循环和移位指令也会使用进位标志。溢出标志用来指示有符号运算的结果太小或者太大。奇偶校验标志用来指示计算结果的最低字节（二进制表示）包含 1 的个数是奇为偶数。符号标志和零标志供逻辑和算术指令来指示结果是否为负数、0 或者正数。 [12]

EFLAGS 寄存器还包含了一个称为方向标志（DF）的控制位。应用程序可以设置或者清除方向标志，以定义串指令执行时 EDI 和 ESI 寄存器自动递增的方向（0－从低地址到高地址，1－从高地址到低地址）。EFLAGS 寄存器的其他各个位是供操作系统来管理中断、I/O 操作和支持程序调试。应用程序不应该修改这些位。应用程序也不应该修改任何保留位，也不能假定保留位的状态。

1.3.4 指令指针

指令指针寄存器（EIP）中包含着 CPU 将要执行的下一条指令的偏移。EIP 寄存器是由控制转移指令按隐含规则自动管理的。举例来说，call 指令（调用过程）会把 EIP 寄存器的值压到栈上，然后把程序控制转移到操作数所指定的地址。而 ret 指令（从过程返回）会把栈上的最顶一项弹出到 EIP 寄存器中，把程序控制转移到该项所指定的地址。

指令 jmp（跳转）和 jcc（满足条件则跳转）也会通过修改 EIP 寄存器的值来转移程序控制。与 call 指令和 ret 指令不同，所有的 x86-32 跳转指令执行时都不会访问栈。还应该说明的是，当前正在执行的任务是不可能直接访问到 EIP 寄存器的。

1.3.5 指令操作数

大多数 x86-32 指令都是使用操作数的，操作数代表着指令要进行操作的具体值。几乎所有指令都需要一个或者更多的源操作数和一个目标操作数，而且大多数指令都需要程序员显式指定源操作数和目标操作数。不过，也有一些指令是隐含指定操作数或者强制指定的。

可以把操作数分成三种基本的类型：立即数、寄存器和内存。立即数操作数是一个常量，它的值会被编码到指令当中，成为指令的一部分。这种操作数通常用来指定算术常量、

逻辑常量或者偏移值。只可以在源操作数中使用立即数。寄存器操作数包含在通用寄存器中。内存操作数指定的是数据在内存中的位置，可以包含本章前面讨论的各种数据类型。在一条指令中可以用内存操作数作为源操作数，或者作为目标操作数，但是不可以同时使用。
[13] 表 1-6 包含了一些指令示例，它们使用了不同类型的操作数。

表 1-6 指令操作数示例

类型	示例	等价的 C/C++ 语句
立即数	mov eax, 42	eax = 42
	imul ebx, 11h	ebx *= 0x11
	xor dl, 55h	dl ^= 0x55
	add esi, 8	esi += 8
寄存器	mov eax, ebx	eax = ebx
	inc ecx	ecx += 1
	add ebx, esi	ebx += esi
	imul ebx	edx:eax = eax * ebx
内存	mov eax, [ebx]	eax = *ebx
	add eax, [val1]	eax += *val1
	or ecx, [ebx+esi]	ecx |= *(ebx + esi)
	sub word ptr [edi], 12	*(short*)edi -= 12

表 1-6 中的 mul 指令（无符号乘法）是使用隐含操作数的一个例子。在这个例子中，隐式使用的寄存器 EAX 和显式使用的寄存器 EBX 被用作源操作数；隐式使用的寄存器对 EDX：EAX 被用作目标操作数。乘积的高位双字和低位双字分别存放在 EDX 和 EAX 寄存器中。

最后一个内存例子中的 word ptr 是汇编器使用的运算符，作用相当于 C++ 中的 cast 运算符。在这个例子中，EDI 寄存器指向的 16 位的内存数据会被减掉 12。如果没有这个类型转换运算符，那么这条汇编语句就是有歧义的，汇编器不能确定 EDI 寄存器所指向的操作数的大小。在这个例子中，操作数也可以是 8 位或者 32 位的数值。本书的编程章节将更多介绍汇编操作符以及汇编指示符的用法。

1.3.6 内存寻址模式

x86-32 指令集支持最多使用四个部分来指定内存操作数。这四个部分分别是：一个固定不变的位移值、一个基址寄存器、一个索引寄存器和一个放大因子。当处理器取到一条使用内存操作数的指令后，它会计算实际地址（effective address）以决定操作数的最终内存地址。实际地址是这样计算的：

Effective Address = BaseReg + IndexReg * ScaleFactor + Disp

[14] （实际地址 = 基址寄存器 + 索引寄存器 * 放大因子 + 位移）

可以使用任何通用寄存器作为基址寄存器（BaseReg）；索引寄存器（IndexReg）也是可以使用任何通用寄存器的，但 ESP 除外；位移（Disp）值是常数值，会被编码到指令当中；有效的放大因子（ScaleFactor）包括 1、2、4 和 8。最终的实际地址总是 32 位大小的。对于一条指令来说，没有必要显式指定计算实际地址的所有四个部分。x86-32 指令集支持 8 种不同的内存操作数寻址方式，如表 1-7 所示。

表 1-7 内存操作数的寻址方式

寻址方式	示例
Disp	mov eax, [MyVal]
BaseReg	mov eax, [ebx]
BaseReg + Disp	mov eax, [ebx+12]
Disp + IndexReg * SF	mov eax, [MyArray+esi*4]
BaseReg + IndexReg	mov eax, [ebx+esi]
BaseReg + IndexReg + Disp	mov eax, [ebx+esi+12]
BaseReg + IndexReg * SF	mov eax, [ebx+esi*4]
BaseReg + IndexReg * SF + Disp	mov eax, [ebx+esi*4+20]

表 1-7 中的例子演示了如何在 mov（Move）指令中使用不同格式的内存操作数。在这些例子中，实际地址所指向内存位置的双字数值会被复制到 EAX 寄存器中。

表 1-7 中列出的大多数寻址方式可以用来引用普通的数据类型，也可以用来引用数据结构。举例来说，简单的位移方式经常用来访问全局变量或者静态变量。基址寄存器方式与 C++ 中的指针很类似，用于引用某个值。访问数据结构中的某个字段时，可以使用基址寄存器指向结构体，用位移指定字段的偏移。索引寄存器用于访问数组中的某个元素。放大因子有助于访问数组中的各个元素，元素的类型可以是整数、单精度浮点数和双精度浮点数。而基址寄存器和索引寄存器结合起来使用对访问二维数组中的元素是很有用的。

1.4 指令集浏览

本小节将对 x86-32 指令集做一个简要的浏览，其目的是帮助大家对 x86-32 指令集有一个比较全面的理解。指令的描述力求简洁，因为随手可得的英特尔和 AMD 参考手册里包含了每一条指令的全部细节，包括执行经过、有效操作数、会影响的标志以及可能促发的异常。附录 C 包含了这些手册的完整列表。第 2 章中的编程示例演示了这些指令的用法，并给出了更多解释。

15

许多 x86-32 指令会更新 EFLAGS 寄存器中的一个或者多个状态标志。正如本章前面讨论的那样，这些状态标志提供了关于运算结果的更多信息。指令 jcc、cmovcc（Conditional Move）和 setcc（Set Byte on Condition）使用所谓的条件码来测试状态标志，要么测试其中的单个标志，要么测试多个标志的组合。表 1-8 列出了条件码、助记后缀以及这些指令使用的标志。注意在最后一列中，C++ 的运算符 ==、!=、&& 和 || 分别用来表示等于、不等于、逻辑与和逻辑或。

表 1-8 条件码、助记后缀和测试条件

条件码	助记后缀	测试条件
Above	A	CF == 0 && ZF == 0
Neither below or equal	NBE	
Above or equal	AE	CF == 0
Not below	NB	
Below	B	CF == 1
Neither above nor equal	NAE	
Below or equal	BE	CF == 1 | ZF == 1

（续）

条件码	助记后缀	测试条件
Not above	NA	
Equal	E	ZF = 1
Zero	Z	
Not equal	NE	ZF = 0
Not zero	NZ	
Greater	G	ZF = 0 && SF = OF
Neither less nor equal	NLE	
Greater or equal	GE	SF = OF
Not less	NL	
Less	L	SF != OF
Neither greater nor equal	NGE	
Less or equal	LE	ZF = 1 || SF != OF
Not greater	NG	
Sign	S	SF = 1
[16] Not sign	NS	SF = 0
Carry	C	CF = 1
Not carry	NC	CF = 0
Overflow	O	OF = 1
Not overflow	NO	OF = 0
Parity	P	PF = 1
Parity even	PE	
Not parity	NP	PF = 0
Parity odd	PO	

表 1-8 中的许多条件码被重复使用了不同的助记后缀，目的是为了提高程序的可读性。当前面提到的条件指令的操作数是无符号整数时，条件码里用的是“above”和“below”，如果是有符号整数，那么用的是“greater”和“less”。如果你对表 1-8 中的条件码定义有困惑或者觉得太抽象，不必担心，在后面的章节中你将发现包含条件码的例子遍布全书。

为了帮助你理解 x86-32 指令集，根据指令的功能，可以把它们划分为如下几大类：

- 数据传输
- 数据比较
- 数据转换
- 二进制算术
- 逻辑运算
- 旋转和移位
- 字节设置和二进制位串
- 串
- 标志操纵
- 控制转移
- 其他指令

[17] 在接下来的指令描述中，通用寄存器将被简称为 GPR。

1.4.1 数据传输

数据传输组的指令可以在两个通用寄存器之间或者通用寄存器和内存之间复制和交换数据，既支持按条件移动数据，也支持无条件移动数据。这个指令组还包括把数据压入栈或者从栈弹出数据的指令。表 1-9 简要描述了数据传输指令。

表 1-9 数据传输指令

助记符	描 述
mov	在内存和 GPR 之间复制数据，也可以把立即数复制到 GPR 或者内存
cmovcc	把内存或者 GPR 中的数据有条件地复制到 GPR。助记符中的 cc 代表表 1-8 中的条件码
push	把一个 GPR、内存位置或者立即数压到栈顶。这条指令把 ESP 的值减去 4，然后把指定的操作数复制到 ESP 指向的内存
pop	从栈上弹出最顶的一项。这条指令会把 ESP 指向的内存内容复制到指定的 GPR 或者内存位置，然后对 ESP 加 4
pushad	把所有八个 GPR 压入栈
popad	从栈中弹出数据，恢复所有八个 GPR 的内容。ESP 的栈中的值会被忽略
xchg	在两个 GPR 或者一个 GPR 和一个内存位置间交换数据。如果这条指令使用的是寄存器和内存间的交换，那么处理器会使用一个加锁的总线操作
xadd	在两个 GPR 或者一个 GPR 和一个内存位置间交换数据。两个操作数的和会被保存到目标操作数
movsx	把 GPR 或者指定内存位置的内容做符号扩展，并把结果复制到 GPR
movzx	把 GPR 或者指定内存位置的内容做零扩展，并把结果复制到 GPR

1.4.2 二进制算术

二进制算术组的指令用于对有符号和无符号整数进行加减乘除运算。这组指令也包括对组合 BCD 数据和非组合 BCD 数据进行调整。表 1-10 描述了二进制算术组的指令。 [18]

表 1-10 二进制算术指令

助记符	描 述
add	把源操作数和目标操作数相加。这条指令既可以用于有符号数，也可以用于无符号数
adc	把源操作数、目标操作数和 EFLAGS.CY 相加。这条指令既可以用于有符号整数，也可以用于无符号整数
sub	从目标操作数减去源操作数。这条指令既可以用于有符号整数，也可以用于无符号整数
sbb	从目标操作数减去源操作数和 EFLAGS.CY。这条指令既可以用于有符号整数，也可以用于无符号整数
imul	对两个操作数做有符号乘法。这条指令支持多种形式，包括单一的源操作数（AL、AX 或者 EAX 作为隐含的操作数）。算式的源操作数和目标操作数，还有一种三操作数变体（立即数据，内存 / 寄存器源和 GPR 目标）
mul	对源操作数和 AL、AX 或 EAX 寄存器做无符号乘法。结果会被保存到 AX、DX：AX 或者 EDX：EAX 寄存器
idiv	有符号除法，使用 AX、DX:AX 或者 EDX:EAX 作被除数，源操作数作除数。得到的商和余数会被保存到寄存器对 AL:AH、AX:DX 或者 EAX:EDX
div	无符号除法，使用 AX、DX:AX 或者 EDX:EAX 作被除数，源操作数作除数。得到的商和余数会被保存到寄存器对 AL:AH、AX:DX 或者 EAX:EDX
inc	对指定操作数加一。这条指令对 EFLAGS.CY 的值不会产生影响
dec	对指定操作数减一。这条指令对 EFLAGS.CY 的值不会产生影响
neg	计算指定操作数的 2 的补码

（续）

助记符	描　述
daa	跟随在对组合 BCD 数实施加法的指令之后，调整 AL 寄存器的值，以便产生正确的 BCD 结果
das	跟随在对组合 BCD 数实施减法的指令之后，调整 AL 寄存器的值，以便产生正确的 BCD 结果
aaa	跟随在对未组合 BCD 数实施加法的指令之后，调整 AL 寄存器内值，以便产生正确的 BCD 结果
aas	跟随在对未组合 BCD 数实施减法的指令之后，调整 AL 寄存器内值，以便产生正确的 BCD 结果
aam	跟随在对未组合 BCD 数实施乘法的指令之后，调整 AX 寄存器的值，以便产生正确的 BCD 结果
aad	调整 AX 寄存器的值，为非组合 BCD 除法做准备。这条指令只作对未组合 BCD 数实施除法的指令之前

19

1.4.3 数据比较

数据比较组的指令用于比较两个操作数，然后设置不同的状态标志，以指示比较的结果。表 1-11 列出了数据比较指令。

表 1-11 数据比较指令

助记符	描　述
cmp	通过从目标操作数中减去源操作数来比较两个操作数，然后设置状态标志。相应的结果会被丢弃。通常比较 jcc，cmovcc 和 setcc 指令之前
cmpxchg	把 AL、AX 或者 EAX 寄存器中的内容与目标操作数做比较，并根据比较结果进行交换
cmpxchg8b	把 EDX：EAX 与一个 8 字节内存操作数做比较，并根据结果进行交换

1.4.4 数据转换

数据转换组中的指令用于对 AL、AX 或者 EAX 寄存器中的整数做符号扩展。所谓符号扩展，就是把源操作数的符号位复制到目标操作数的高位。举例来说，如果要把 8 位数 0xe9（-23）做符号扩展到 16 位，便会得到 0xffe9。这组指令还包括那些把小端格式（little-endian）的数据转换到大端格式（big-endian）的指令。表 1-12 列出了数据转换组的各条指令。

表 1-12 数据转换指令

助记符	描　述
cbw	对寄存器 AL 做符号扩展，然后把结果保存到寄存器 AX
cwde	对寄存器 AX 做符号扩展，然后把结果保存到寄存器 EAX
cwd	对寄存器 AX 做符号扩展，然后把结果保存到寄存器对 DX:AX
cdq	对寄存器 EAX 做符号扩展，然后把结果保存到寄存器对 EDX:EAX
bswap	调整 32 位 GPR 中的字节，把原来的数据从小端格式转换成大端格式，或者做相反操作
movbe	把源操作数加载到临时寄存器，调整各个字节，然后把结果保存到目标操作数。这条指令可以把源操作数从小端格式转换为大端格式，或者相反。其中一个操作数必须是内存位置，另一个必须是 GPR
xlatb	通过存放在 EBX 寄存器内指向的数据表把 AL 寄存器中包含的数据转换为另一个数据

20

1.4.5 逻辑运算

逻辑运算组的指令用来对指定操作数按位做逻辑运算。处理器会更新状态标志 EFLAGS.PF、EFLAGS.SF 和 EFLAGS.ZF 以反映这些指令的结果，除非特别说明。表 1-13 列出了逻辑运

算组的各条指令。

表 1-13 逻辑指令

助记符	描 述
and	对源操作数和目标操作数做按位与操作
or	对源操作数和目标操作数做按位或操作
xor	对源操作数和目标操作数做按位异或操作
not	计算指定操作数的 1 的补数。这条指令不影响状态标志
test	对源操作数和目标操作数做按位与操作，然后丢弃结果。这条指令用来以无损的方式（non-destructively）设置状态标志

1.4.6 旋转和移位

这一组中的指令用来对操作数做旋转和移位。这些指令有几种格式，有的是对单个位进行操作，有的是对多个位进行操作。旋转和移动多个位时，使用 CL 寄存器来指定位的个数。旋转操作可以影响进位标志，也可以不影响进位标志。表 1-14 列出了旋转和移位指令。

表 1-14 旋转和移位指令

助记符	描 述
rcl	向左旋转指定的操作数。标志 EFLAGS.CY 作为其中的一部分
rcr	向右旋转指定的操作数。标志 EFLAGS.CY 作为其中的一部分
rol	向左旋转指定的操作数
ror	向右旋转指定的操作数
sal/shl	对指定的操作数进行算术左移
sar	对指定的操作数进行算术右移
shr	对指定的操作数进行逻辑右移
shld	对两个操作数进行双精度逻辑左移
shrd	对两个操作数进行双精度逻辑右移

[21]

1.4.7 字节设置和二进制位串

字节设置和二进制位串指令组包含了有条件地设置字节值的指令，也包含处理二进制位串的指令。表 1-15 简要描述了字节设置和二进制位串指令。

表 1-15 字节设置和二进制位串指令

助记符	描 述
setcc	如果 cc 指定的条件码为真，则把目标字节操作数设置为 1，否则设置为 0
bt	把指定的测试位复制到 EFLAGS.CY
bts	把指定的测试位复制到 EFLAGS.CY，然后把测试位设置为 1
btr	把指定的测试位复制到 EFLAGS.CY，然后把测试位设置为 0
btc	把指定的测试位复制到 EFLAGS.CY，然后把测试位设置为 0
bsf	扫描源操作数，寻找设置为 1 的最低位，把其索引保存到目标操作数。如果源操作数是 0，把 EFLAGS.ZF 设置为 1，否则把 EFLAGS.ZF 设置为 0
bsr	扫描源操作数，寻找设置为 1 的最高位，把其索引保存到目标操作数。如果源操作数是 0，把 EFLAGS.ZF 设置为 1，否则把 EFLAGS.ZF 设置为 0

1.4.8 串

串指令组包含的指令可以对文本串或者多个内存块进行比较、加载、移动、扫描和存储。所有串指令使用ESI寄存器作为源指针，EDI寄存器作为目标指针。串指令会根据方向标志（EFLAGS.DF）自动递增或者递减这两个寄存器。使用指令前缀rep、repe/repz或者
22 repne/repnz可以重复执行串操作，ECX寄存器用作循环计数器。表1-16列出了串指令。

表1-16 串指令

助记符	描 述
cmpsb cmpsw cmpsd	比较ESI寄存器和EDI寄存器所指向内存的值，设置状态标志来指示比较结果
lodsb lodsw lowsd	把ESI寄存器所指向内存的值加载到AL、AX或者EAX寄存器
movsb movsw movsd	将ESI寄存器指向的内存的值复制到EDI寄存器指向的内存
scasb scasw scasd	把EDI寄存器指向的内存值与寄存器AL、AX或者EAX中的值作比较，并把比较结果设置状态标志
stosb stosw stosd	把AL、AX或者EAX寄存器的内容保存到EDI寄存器指向的内存
rep	当条件ECX != 0为真时重复指定的串指令
repe repz	当条件ECX != 0 && ZF == 1为真时重复指定的串指令
repne repnz	当条件ECX != 0 && ZF == 0为真时重复指定的串指令

1.4.9 标志操纵

这一组指令用来操纵EFLAGS寄存器中的一些状态标志。表1-17列出了这些指令。

表1-17 标志操纵指令

助记符	描 述
clc	把EFLAGS.CY设置为0
stc	把EFLAGS.CY设置为1
cmc	反转EFLAGS.CY的状态
std	把EFLAGS.DF设置为1
23 cld	把EFLAGS.DF设置为0
lahf	把状态标志寄存器加载到AH寄存器。加载到AH寄存器中的位（从最高位到最低位）为：EFLAGS.SF、EFLAGS.ZF、0、EFLAGS.AF、0、EFLAGS.PF、1、EFLAGS.CF
sahf	把AH寄存器的值存储到状态标志。存储到状态标志的AH寄存器的位为（从最高位到最低位）：EFLAGS.SF、EFLAGS.ZF、0、EFLAGS.AF、0、EFLAGS.PF、1、EFLAGS.CF（其中的0和1表示设置这些位时会使用这些值，而不是用AH中的对应位）
pushfd	把EFLAGS寄存器压栈
popfd	从栈上弹出最顶一项，将其复制到EFLAGS寄存器。注意，EFLAGS寄存器中的保留位不会受影响

1.4.10 控制转移

控制转移组的指令用于执行跳转、函数调用和返回以及做循环。表 1-18 列出了控制转移指令。

表 1-18 控制转移指令

助记符	描 述
jmp	无条件跳转到操作数所指定的内存位置
jcc	如果指定的条件为真，则跳转到操作数所指定的内存位置。其中的 cc 代表表 1-8 中列出的条件码助记符
call	把 EIP 寄存器的内容压入栈，然后无条件跳转到操作数所指定的内存位置
ret	从栈上弹出目标地址，然后无条件跳转到那个地址
enter	通过初始化 EBP 和 ESP 寄存器来为函数建立函数参数和局部变量所需的栈帧
leave	通过恢复 EBP 和 ESP 寄存器来移除使用 enter 指令建立的栈帧
jecxz	如果条件 ECX == 0 为真，则跳转到指定的内存位置
loop	对 ECX 寄存器减 1，如果条件 ECX — 0 为真，则跳转到指定的内存位置
loope loopz	对 ECX 寄存器减 1，如果条件 ECX != 0 && ZF == 1 为真，则跳转到指定的内存位置
loopne loopnz	对 ECX 寄存器减 1，如果条件 ECX != 0 && ZF == 0 为真，则跳转到指定的内存位置

1.4.11 其他指令

这一组指令不能归入前面介绍的任何一组。表 1-19 列出了这一组的部分指令。

表 1-19 其他指令

助记符	描 述
bound	对数组索引进行验证检查，如果检测到超出边界的条件，处理器会产生一个异常
lea	计算源操作数的实际地址，并把结果保存到目标操作数（必须为通用寄存器）
nop	把命令指针（EIP）指向下一条指令，其他寄存器和标志都不改变
cupid	获取处理器内部识别信息和功能信息。可以使用这条指令在运行期确定某一种 SIMD 扩展是否存在，也可以用来判断处理器是否支持某种硬件功能

1.5 总结

本章分析了 x86-32 平台的核心架构，包括数据类型和内部架构，还介绍了编写应用程序所需的常用 x86-32 指令。如果你是第一次接触 x86 平台的内部架构或者汇编语言编程，那么可能会感觉前面讲的某些内容有点难懂。但是不要怕，正如在序言中所讲的，本书的所有章节要么是用来指导实践的，要么就是用来动手学习的。下一章会把注意力转向 x86 汇编语言编程的实战方面，将通过演示代码和具体的例子来理解本章讨论的概念。

第 2 章

Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX

x86-32 核心编程

上一章重点介绍的是 x86-32 平台的基本知识，包括数据类型、执行环境和指令集。这一章我们将把注意力集中到 x86-32 汇编语言编程。说得更具体些，我们将介绍如何使用汇编语言编写函数，并在 C++ 程序中调用。此外，本章还会介绍 x86 汇编语言的语法和语义。为了帮助大家理解这一章的理论性内容，本章配备了很多示例程序。

本章的内容是这样组织的。第一节描述如何编写一个简单的汇编语言函数。你会学习如何在 C++ 程序和 x86 汇编程序的函数之间传递参数和返回值。我们将顺带讨论使用 x86-32 指令时要注意的一些问题以及 Visual Studio 开发工具的使用方法。

第二节讨论的是 x86-32 汇编语言编程的基本知识。我们将介绍在函数间传递参数和返回值的更多细节，包括函数序言和结语。这一节还会回顾几个 x86 汇编语言编程的普遍话题，包括内存寻址模式、变量的用法以及条件指令。在汇编语言基础这一节之后，我们将讨论数组的用法，差不多所有应用程序都会在某种程度上使用数组。这一节将演示在汇编语言编程中使用一维数组和二维数组的技术。

很多应用程序还会用到结构体来创建和管理用户定义的数据类型。数组之后的一节将演示结构体的用法，并探讨在 C++ 和汇编语言的函数之间使用结构体应注意的几个问题。本章的最后一节将演示如何使用 x86 的串指令。这些指令经常被用来对文本串进行操作，但也可以用来处理数组的元素。

需要说明的是，本章中的示例代码主要用来演示 x86-32 指令集的用法以及汇编语言编程技术。所有代码力求直截了当，但却不一定是最优的，因为理解优化过的汇编语言代码是有难度的，尤其是对初学者而言。后续章节的示例代码会把更多的注意力放在如何编写高效率的代码上。第 21 章和第 22 章会专门介绍编写高效率汇编语言代码的很多策略。

2.1 开始

这一节将分析几个简单的程序，目的是说明如何在 C++ 函数和汇编语言函数之间传递数据。我们还将介绍如何使用常用的 x86-32 汇编语言指令和一些基本的汇编器指示符（assembler directive）。

正如在前言里所说明的，本书讨论的所有示例代码都是使用微软的 Visual C++ 和宏汇编器（MASM）所创建的，这两个工具都是 Visual Studio 的一部分。在开始学习本书的第一个示例程序之前，你应该先学习一点使用这些开发工具的基础知识。

Visual Studio 使用所谓的解决方案（solution）和项目（project）来简化应用程序开发。一个解决方案包含一个或者多个用以构建应用程序的项目。每个项目像容器一样，用以组织应用程序的文件，包括源代码、资源文件、图标、位图、HTML 和 XML 等。通常为应用的每个可构建的部件（比如可执行文件、动态链接库、静态链接库等）创建一个 Visual Studio 项目。你可以通过双击解决方案文件（.sln）来启动 Visual Studio 开发环境并加载本书的示例程序。附录 A 包含了一个简单的教程，教你如何创建 Visual Studio 解决方案以及创建包

含 C++ 文件和 x86 汇编语言文件的项目。

2.1.1 第一个汇编语言函数

我们要分析的第一个 x86-32 汇编语言程序名叫 CalcSum。这个示例程序演示的是一些基本的汇编语言概念，包括传递参数、使用栈和返回值。此外，这个例子还会演示如何使用基本的汇编器指示符。

在深入探讨 CalcSum 程序的细节之前，我们先回顾一下当一个 C++ 函数调用另一个函数时的过程。像许多其他编程语言一样，C++ 使用面向栈的架构来支持参数传递和局部变量存储。在清单 2-1 中，函数 CalcSumTest 计算并返回三个整数的和。在从 _tmain 函数里调用这个 CalcSumTest 函数之前，a、b 和 c 的值会被从右到左依次压到栈上。在进入到 CalcSumTest 时，（编译器产生的代码）会初始化栈帧指针，用以协助访问 _tmain 函数压到栈上的三个整数参数。函数还会根据需要分配栈空间。接下来，CalcSumTest 会做求和计算，并把结果复制到预先指定好的返回值寄存器中，返回前先释放刚才分配的局部栈空间，而后返回到 _tmain。应该说明一下，尽管上面介绍的过程在概念是上精确的，但是如果启用了优化选项，那么今天的 C++ 编译器很可能对上面提到的栈操作进行简化。

清单 2-1 CalcSumTest.cpp

```
#include "stdafx.h"

int CalcSumTest(int a, int b, int c)
{
    return a + b + c;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int a = 17, b = 11, c = 14;
    int sum = CalcSumTest(a, b, c);

    printf("  a:   %d\n", a);
    printf("  b:   %d\n", b);
    printf("  c:   %d\n", c);
    printf("  sum: %d\n", sum);
    return 0;
}
```

28

前面介绍的函数调用过程也适用于从 C++ 函数调用汇编语言函数。清单 2-2 和清单 2-3 分别显示了 CalcSum 程序的 C++ 代码和汇编语言代码。在这个例子中，汇编语言函数被命名为 CalcSum_。因为 CalcSum_ 是本书的第一个 x86-32 汇编语言函数，所以有必要仔细地看一下清单 2-2 和清单 2-3。

清单 2-2 CalcSum.cpp

```
#include "stdafx.h"

extern "C" int CalcSum_(int a, int b, int c);

int _tmain(int argc, _TCHAR* argv[])
{
    int a = 17, b = 11, c = 14;
    int sum = CalcSum_(a, b, c);
```

```
    printf("  a:   %d\n", a);
    printf("  b:   %d\n", b);
    printf("  c:   %d\n", c);
    printf("  sum: %d\n", sum);
    return 0;
}
```

清单 2-3 CalcSum_.asm

```
    .model flat,c
    .code

; extern "C" int CalcSum_(int a, int b, int c)
;
; 描述：这个函数演示了如何在 C++ 和汇编语言函数间传递参数
;
; 返回值：a + b + c
CalcSum_ proc

; 初始化栈帧指针
        push ebp
        mov ebp,esp

; 加载参数值
        mov eax,[ebp+8]                 ; eax = 'a'
        mov ecx,[ebp+12]                ; ecx = 'b'
        mov edx,[ebp+16]                ; edx = 'c'

; 求和
        add eax,ecx                     ; eax = 'a' + 'b'
        add eax,edx                     ; eax = 'a' + 'b' + 'c'

; 恢复父函数的栈帧指针
        pop ebp
        ret

CalcSum_ endp
        end
```

29

注意 在本书的示例代码中，所有汇编语言文件、函数和全局变量名都以下划线结尾，以方便区分。

CalcSum.cpp 看起来很直截了当，只有几行代码需要解释一下。#include "stdafx.h" 这一行用来指定项目相关的头文件，这个头文件里面包含着经常使用的一些系统项目的头文件。每当创建一个新的 C++ 控制台应用项目时，Visual Studio 就会自动产生这个文件。extern "C" int CalcSum_(int a, int b, int c) 这一行是 C++ 的声明语句，用来定义汇编语言函数 CalcSum_ 的参数和返回值。它还告诉编译器对 CalcSum_ 函数使用 C 风格的命名规范，不要使用 C++ 的装饰名（C++ 装饰名包含用以支持重载的额外字符）。剩下的 CalcSum.cpp 代码主要是使用 printf 函数向标准控制台输出信息。

CalcSum_.asm 的开头几行是 MASM 指示符。MASM 指示符用来告诉汇编器如何执行某些操作。.model flat, c 指示符告诉汇编器产生适用于平坦内存模型的代码，并使用 C 风格的规范给公共符号命名。.code 指示符用以标示包含可执行代码的内存块的起点。在本书后面的章节中，我们会介绍更多的汇编指示符。接下来的几行是注释，汇编器会忽略出现在分号后的任何字符。语句 CalcSum_ proc 代表一个函数（或者过程）的开始。在接近源

文件末尾的 CalcSum _endp 语句用以标示函数的结束。应该说明的是，proc 和 endp 语句并不是可执行的指令，而是用以标示函数开始和结束的汇编器指示符。最末尾的 end 语句是另一个汇编器指示符，用来代表整个文件的结束。汇编器会忽略 end 指示符之后的任何文本。

CalcSum _函数中的第一条 x86-32 汇编语言指令是 push ebp（向栈压入一个双字）。这条指令会把调用者的 EBP 寄存器内容压到栈上。下一条指令是 mov ebp, esp（Move），把 ESP 的内容复制到 EBP，也就是把 EBP 初始化为栈帧指针，以便访问函数的参数。图 2-1 画出了执行完 mov ebp, esp 指令后的栈状态。保存 EBP 寄存器的值和初始化栈帧指针是函数序言的主要部分。本章后面会更深入地讨论函数序言。

在初始化栈帧指针后，我们使用了一系列 mov 指令将 a、b、c 这三个参数的值分别加载到寄存器 EAX、ECX 和 EDX 中。每个 mov 指令的源操作数使用了 BaseReg+Disp 形式的内存寻址模式来引用栈上的值（参见第 1 章关于内存寻址模式的内容）。把参数值加载到寄存器之后，就可以进行求和操作了。指令 add eax, ecx (Add) 对寄存器 EAX 和 ECX（包含了 a 和 b 的值）求和，结果放在 EAX 寄存器中。下一条指令 add eax, edx 把 c 加到上一步求得的和里，并把结果保存到 EAX。



图 2-1 初始化栈帧指针后栈的内容。栈上数据的偏移是相对于寄存器 EBP 和 ESP 的

使用 x86-32 汇编语言编写的函数必须使用 EAX 寄存器来向调用者返回 32 位的整数值。在我们当前讨论的函数中，因为 EAX 已经包含了正确的值，所以不再需要任何额外指令。指令 pop ebp（从栈上弹出数值）用来恢复调用者的 EBP 寄存器内容，这一行是函数结语代码的重要部分。本章后面会更详细地讨论函数结语。最后的 ret 指令（从过程返回）把程序控制权归还给调用函数 _tmain。输出 2-1 显示了运行示例程序 CalcSum 的结果。

输出 2-1 示例程序 CalcSum

```
a:  17
b:  11
c:  14
sum: 42
```

你可以通过如下步骤用 Visual Studio 观察源程序文件并运行这个示例程序：

1. 打开 Windows 的文件管理器（File Explorer），双击如下 Visual Studio 解决方案文件：Chapter02\CalcSum\CalcSum.sln。

2. 选择 View > Solution Explorer 打开 Solution Explorer 窗口。

3. 在 Solution Explorer 窗口的树控件中，展开标有 CalcSum 的节点和源文件。

4. 双击 CalcSum.cpp 和 CalcSum .asm 在编辑器中打开文件。

5. 如果想运行程序，那么选择 DEBUG > Start Without Debugging。

在本章的后续内容中，你将学习到使用 Visual Studio 的更多细节。

2.1.2 整数乘法和除法

我们要分析的下一个示例程序叫 IntegerMulDiv。这个程序演示的是如何使用 imul（带符号乘法）和 idiv（带符号除法）指令来对带符号整数做乘法和除法运算。这个例子还会演示如何在 C++ 函数和汇编语言函数之间使用指针传递数据。你可以在资源管理器中双击 Chapter02\IntegerMulDiv\IntegerMulDiv.sln 文件启动 Visual Studio 打开这个例子的解决方案文件。

清单 2-4 显示了 IntegerMulDiv.cpp 的源代码。靠近文件顶部的声明语句定义了用以计算结果的汇编语言函数 IntegerMulDiv_，它有五个参数：两个整数值和三个用于返回结果的整数指针。这个函数返回整数值用以指示是否发生了除零操作。IntegerMulDiv.cpp 的其余代码以几种方式调用汇编语言函数 IntegerMulDiv_ 对其进行测试。

[32]

清单 2-4 IntegerMulDiv.cpp

```
#include "stdafx.h"

extern "C" int IntegerMulDiv_(int a, int b, int* prod, int* quo, int* rem);

int _tmain(int argc, _TCHAR* argv[])
{
    int a = 21, b = 9;
    int prod = 0, quo = 0, rem = 0;
    int rc;

    rc = IntegerMulDiv_(a, b, &prod, &quo, &rem);
    printf(" Input1 - a:  %4d  b:   %4d\n", a, b);
    printf("Output1 - rc:  %4d  prod: %4d\n", rc, prod);
    printf("          quo: %4d  rem:  %4d\n\n", quo, rem);

    a = -29;
    prod = quo = rem = 0;
    rc = IntegerMulDiv_(a, b, &prod, &quo, &rem);
    printf(" Input2 - a:  %4d  b:   %4d\n", a, b);
    printf("Output2 - rc:  %4d  prod: %4d\n", rc, prod);
    printf("          quo: %4d  rem:  %4d\n\n", quo, rem);

    b = 0;
    prod = quo = rem = 0;
    rc = IntegerMulDiv_(a, b, &prod, &quo, &rem);
    printf(" Input3 - a:  %4d  b:   %4d\n", a, b);
    printf("Output3 - rc:  %4d  prod: %4d\n", rc, prod);
    printf("          quo: %4d  rem:  %4d\n\n", quo, rem);
    return 0;
}
```

清单 2-5 包含了使用汇编语言编写的 IntegerMulDiv_ 函数源代码。刚开始的几行和你在上一节学习到的示例程序的前几行很相似。其中的汇编器指示符用来定义内存模型和代码块的起始位置。函数序言包含了必要的指令用以保存调用者的 EBP 寄存器和初始化栈帧指针。它还包含了一条 push ebx 指令，用以把调用者的 EBX 寄存器保存到栈上。根据 Visual C++ 中关于 32 位应用程序的调用约定，被调用函数必须保持以下寄存器的值（在函数返回时仍是原来的值）：EBX、ESI、EDI 和 EBP。这些寄存器被称为非易变（non-volatile）寄存器。易变寄存器 EAX、ECX 和 EDX 不需要在函数调用时保持原来值。在本章后面的内容中，你将学习到关于 Visual C++ 调用约定的更多细节。

[33]

清单 2-5 IntegerMulDiv_.asm

```
        .model flat,c
        .code

; extern "C" int IntegerMulDiv_(int a, int b, int* prod, int* quo, int*
; rem);
;
; 描述：这个函数演示了 imul 和 idiv 指令的用法，也演示了指针的用法
;
; 返回值：0 Error（除数是 0）
;         1 Success（除数是 0）
;
; 计算：*prod = a * b;
;       *quo = a / b
;       *rem = a % b

IntegerMulDiv_ proc

; 函数序言
        push ebp
        mov ebp,esp
        push ebx

; 确保除数不为 0
        xor eax,eax                     ;设置错误返回码
        mov ecx,[ebp+8]                 ;ecx = 'a'
        mov edx,[ebp+12]                ;edx = 'b'
        or edx,edx
        jz InvalidDivisor               ;若 'b' 为 0，跳转

; 计算积并保存结果
        imul edx,ecx                    ;edx = 'a' * 'b'
        mov ebx,[ebp+16]                ;ebx = 'prod'
        mov [ebx],edx                   ;保存积

; 计算商和余数，并保存结果
        mov eax,ecx                     ;eax = 'a'
        cdq                             ;edx:eax 包含被除数
        idiv dword ptr [ebp+12]         ;eax = quo, edx = rem

        mov ebx,[ebp+20]                ;ebx = 'quo'
        mov [ebx],eax                   ;保存商
        mov ebx,[ebp+24]                ;ebx = 'rem'
        mov [ebx],edx                   ;保存余数
        mov eax,1                       ;设置成功返回码

; 函数结束
InvalidDivisor:
        pop ebx
        pop ebp
        ret
IntegerMulDiv_ endp
        end
```

图 2-2 显示了执行 push ebx 指令后的栈状态。在执行完函数序言后，参数 a 和 b 的值被分别加载到寄存器 ECX 和 EDX 中。接下来是一条 or edx, edx（逻辑或）指令。这条指令把 EDX 与它自己进行按位或操作，其目的是更新寄存器 EFLAGS 中的状态标志，同时保持 EDX 寄存器中的初始值。对参数 b 进行测试是为了防止除以零。紧接的那条 jz InvalidDivisor（Jump if Zero）指令是条件跳转指令，仅当 EFLAGS.ZF == 1 成立时才进行跳转。这条条件跳转指令的目标操作数是指定跳转目标（也就是要执行的下一条指令）的标号（label，如果

设置了零标志）。在这个示例程序中，标号 InvalidDivisor 位于清单的末尾，在结语指令之前。

如果 b 的值不是零，那么程序会继续执行 imul edx, ecx 指令。这条指令会对 EDX 和 ECX 的内容做有符号乘法运算，把结果截断为 32 位，并保存到 EDX 寄存器（imul 指令的单目 32 位操作数形式会把完整的乘积保存到寄存器对 EDX:EAX 中）。接下来的指令 mov ebx, [ebp+16] 会把指针 prod 加载到寄存器 EBX 中。之后的 mov [ebx], edx 指令把前面计算出的乘积保存到指定的内存单元中。

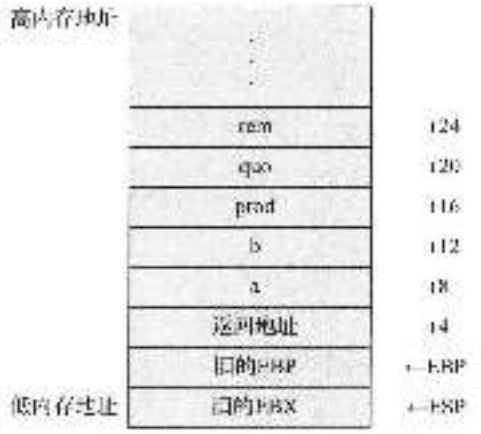


图 2-2 执行 push ebx 后的栈内容。图中显示的偏移相对于寄存器 EBP

33

IntegerMulDiv_ 函数的下一块代码会使用带符号整数除法指令计算 a/b 和 a%b。在 x86 处理器中，使用 32 位操作数的带符号整数除法要求必须把 64 位的被除数加载到寄存器对 EDX:EAX 中。指令 cdq（Convert Doubleword to Quadword）会把 EAX 寄存器的值（包含参数 a 的值，前面执行过 mov eax, ecx）做符号扩展，结果放到 EDX:EAX 寄存器对。接下来的指令 idiv dword ptr [ebp+12] 就进行有符号整数除法。注意 idiv 指令仅跟随一个源操作数：32 位（或双字）的除数，寄存器对 EDX:EAX 中的内容会被当作被除数。另外要说明的是，idiv 指令还可以用来做 8 位和 16 位有符号整数除法。这也是本例中使用 dword ptr 加以限定的原因。在执行完 idiv 指令后，寄存器 EAX 和 EDX 中分别包含的是商和余数。这些值会被保存到指针 quo 和 rem 指定的内存单元中。

IntegerMulDiv 函数的最后一块代码是函数的结语。在执行 ret 指令之前，使用 pop 指令恢复调用者的 EBX 和 EBP。如果某个汇编语言函数没能恰当地恢复非易变寄存器的值，那么很可能引起程序崩溃。如果 ESP 指向栈上的未知数据项或者包含了无效值，那么基本上可以肯定会发生崩溃。输出 2-2 是执行 IntegerMulDiv 的结果。

输出 2-2 示例程序 IntegerMulDiv

```
Input1 - a:     21  b:      9
Output1 - rc:    1  prod:  189
          quo:   2  rem:     3

Input2 - a:    -29  b:      9
Output2 - rc:    1  prod: -261
          quo:  -3  rem:    -2

Input3 - a:     29  b:      0
Output3 - rc:    0  prod:    0
          quo:   0  rem:     0
```

2.2 x86-32 编程基础

上一节的示例代码演示了 x86-32 汇编语言编程的概况。这一节将更系统地介绍 x86-32 汇编语言编程的基础知识。首先我们会更详细地介绍 x86-32 汇编语言函数被 C++ 调用时必须遵循的调用约定。接下来演示如何使用常用的内存寻址模式。再下面到例子会介绍对不

同长度的操作数做整数加法。最后一个例子的学习目标是条件码和条件指令。本节标题中的"基础"二字暗示着这一节内容的重要性。这些内容是有志于学习 x86 汇编语言编程的开发者所应该深入理解的关键知识。

2.2.1 调用约定

当使用 C++ 编写函数时，程序员经常声明局部变量来保存临时数据或者中间结果。例如，在清单 2-6 中，函数 LocalVars 包含三个局部变量：val1，val2 和 val3。如果 C++ 编译器决定使用栈上的内存来容纳这些变量或其他临时变量，那么分配和释放的代码一般放在函数的序言和结语中。汇编语言函数也可以使用同样的技术来分配栈上的空间，供局部变量使用或者满足其他用途。

清单 2-6 LocalVars.cpp

```
double LocalVars(int a, int b)
{
    int val1 = a * a;
    int val2 = b * b;
    double val3 = sqrt(val1 + val2);

    return val3;
}
```

我们要分析的下一个示例程序叫 CallingConvention，它有两个目的。首先，它演示了 C++ 调用约定的额外细节，包括在栈上分配局部变量。其次，这个程序演示了其他几个常用 x86-32 汇编语言指令的用法。清单 2-7 和清单 2-8 分别展示了 C++ 的 CallingConvention.cpp 和汇编语言的 CallingConvention_asm 的代码。对应的 Visual Studio 的解决方案文件是 Chapter02\CallingConvention\CallingConvention.sln。

清单 2-7 CallingConvention.cpp

```
#include "stdafx.h"

extern "C" void CalculateSums_(int a, int b, int c, int* s1, int* s2, int*
= s3);

int _tmain(int argc, _TCHAR* argv[])
{
    int a = 3, b = 5, c = 8;
    int s1a, s2a, s3a;

    CalculateSums_(a, b, c, &s1a, &s2a, &s3a);

    // 再次做求和计算，以便验证汇编语言函数 CalculateSums_() 的结果
    int s1b = a + b + c;
    int s2b = a * a + b * b + c * c;
    int s3b = a * a * a + b * b * b + c * c * c;

    printf("Input:  a:   %4d b:   %4d c:   %4d\n", a, b, c);
    printf("Output: s1a: %4d s2a: %4d s3a: %4d\n", s1a, s2a, s3a);
    printf("        s1b: %4d s2b: %4d s3b: %4d\n", s1b, s2b, s3b);

    return 0;
}
```

清单 2-8 CallingConvention_.asm

```
        .model flat,c
        .code

; extern "C" void CalculateSums_(int a, int b, int c, int* s1, int* s2, int*
↪s3);
;
; 描述：这个函数演示完整的汇编语言序言和结语
;
; 返回值：None
;
; 计算：*s1 = a + b + c
;       *s2 = a * a + b * b + c * c
;       *s3 = a * a * a + b * b * b + c * c * c

CalculateSums_ proc

; 函数序言
        push ebp
        mov ebp,esp
        sub esp,12                      ;分配局部存储空间
        push ebx
        push esi
        push edi

; 获取参数值
        mov eax,[ebp+8]                 ;eax = 'a'
        mov ebx,[ebp+12]                ;ebx = 'b'
        mov ecx,[ebp+16]                ;ecx = 'c'
        mov edx,[ebp+20]                ;edx = 's1'
        mov esi,[ebp+24]                ;esi = 's2'
38      mov edi,[ebp+28]                ;edi = 's3'

; 计算 's1'
        mov [ebp-12],eax
        add [ebp-12],ebx
        add [ebp-12],ecx                ;最终的 's1' 结果

; 计算 's2'
        imul eax,eax
        imul ebx,ebx
        imul ecx,ecx
        mov [ebp-8],eax
        add [ebp-8],ebx
        add [ebp-8],ecx                 ;最终的 's2' 结果

; 计算 's3'
        imul eax,[ebp+8]
        imul ebx,[ebp+12]
        imul ecx,[ebp+16]
        mov [ebp-4],eax
        add [ebp-4],ebx
        add [ebp-4],ecx                 ;最终的 's3' 结果

; 保存 's1'、's2' 和 's3'
        mov eax,[ebp-12]
        mov [edx],eax                   ;保存 's1'
        mov eax,[ebp-8]
        mov [esi],eax                   ;保存 's2'
        mov eax,[ebp-4]
        mov [edi],eax                   ;保存 's3'
```

```
; 函数尾语
        pop edi
        pop esi
        pop ebx
        mov esp,ebp                    ;释放局部存储空间
        pop ebp
        ret
CalculateSums_  endp
        end
```

可以看到，在清单 2-7 中 tmain 调用了一个名为 CalculateSums_ 的函数。CalculateSums_ 函数会对传入的三个整数参数求和。清单 2-8 包含了 CalculateSums_ 函数的汇编语言代码，其开头是大家熟悉的栈帧指针初始化以及把非易变寄存器保存到栈上的指令。在保存非易变寄存器之前，执行了一条 sub esp, 12(Substract) 指令。这条指令会把 ESP 寄存器的内容减去 12，这相当于在栈上分配了 12 字节的局部（而且私有）存储空间，供这个函数使用。使用 sub 指令而不是 add 指令的原因是 x86 的栈是向低地址方向生长的。这种从栈上分配局部存储空间的做法反映了函数序言中 mov ebp, esp 指令的另一个作用。当使用 EBP 寄存器来访问栈上的数据时，向正的方向偏移就可以引用参数，向负的方向偏移就可以引用局部变量，如图 2-3 所示。



图 2-3 包含局部存储空间的栈

在函数序言代码之后，参数 a、b 和 c 被分别加载到寄存器 EAX、EBX 和 ECX 中。计算 s1 的过程演示了如何在加法指令中使用内存作为目标操作数，而不是寄存器。需要说明的是，在这个特定的例子中，使用内存作目标操作数的效率是不高的；其实可以使用寄存器操作数来轻松计算出 s1。这里采用内存作操作数的目的是为了演示栈上局部变量的用法。

接下来的代码是使用双目形式的 imul 指令来计算 s2 和 s3。注意，计算 s3 的 imul 指令使用栈上的原始值来作源操作数，因为寄存器 EAX、EBX 和 ECX 不再包含 a、b 和 c 的原始值了。在计算好所需的结果后，接下来的代码把 s1、s2 和 s3 的值从它们在栈上的临时位置复制到调用者通过指针指定的内存位置。

CalculateSums_ 的最后部分是函数结语，使用 pop 指令来恢复非易变寄存器 EBX、ESI 和 EDI 的值。其中还有一条 mov esp, ebp 指令，其作用是释放前面分配的局部存储空间，并在执行 ret 指令前把 ESP 寄存器恢复成正确的值。执行这个 CallingConvention 示例程序的结果如输出 2-3 所示。

输出 2-3 示例程序 CallingConvention

```
Input:  a:     3 b:     5 c:     8
Output: s1a:   16 s2a:   98 s3a:  664
        s1b:   16 s2b:   98 s3b:  664
```

CalculateSums_ 中的函数序言和结语是很典型的，代表着从 Visual C++ 函数中调用汇

编语言函数时所必须遵守的调用约定。在这一章的后面，我们还会介绍一些与64位数值和C++结构体有关的调用约定。第4章会讨论与浮点数相关的调用约定。附录B对x86-32程序使用的Visual C++调用约定做了完整的归纳。

注意 本章讨论的汇编语言调用约定与其他操作系统和高级语言中的约定可能是不同的。如果你打算通过阅读本书学习x86汇编语言编程，并打算在另一种不同的执行环境下使用，那么你应该仔细查询目标平台的调用约定。

最后要说明的是，当编写汇编语言的函数序言和结语时，一般只应该包含这个函数真正需要的那些指令。比如，如果一个函数根本不使用寄存器EBX的值，那么就不必在函数序言和结语中保存和恢复它的值了。类似地，一个没有参数的函数可以不必初始化栈帧指针。如果省略函数序言和结语，可能出现问题的另一种情况是，一个汇编语言函数调用另一个没有保存非易变寄存器的汇编语言函数。对于这种情况，被调用函数应该确保非易变寄存器都被恰当地保存和恢复了。

2.2.2 内存寻址模式

在第1章中，我们介绍了x86指令集支持很多种寻址模式来引用内存中的操作数。在这一节，我们将通过一个很小的汇编程序实例来演示其中的几种。你还会学习到如何在汇编代码中定义字典表，以及如何定义C++函数中也可以访问到的全局变量。这个示例程序的名字叫MemoryAddressing。清单2-9和清单2-10分别包含了源文件MemoryAddressing.cpp和MemoryAddressing_.asm的内容。这个示例程序的Visual Studio方案文件路径为Chapter02\MemoryAddressing\MemoryAddressing.sln。

41

清单2-9 MemoryAddressing.cpp

```
#include "stdafx.h"

extern "C" int NumFibVals_;
extern "C" int MemoryAddressing_(int i, int* v1, int* v2, int* v3, int* v4);

int _tmain(int argc, _TCHAR* argv[])
{
    for (int i = -1; i < NumFibVals_ + 1; i++)
    {
        int v1 = -1, v2 = -1, v3 = -1, v4 = -1;
        int rc = MemoryAddressing_(i, &v1, &v2, &v3, &v4);

        printf("i: %2d  rc: %2d - ", i, rc);
        printf("v1: %5d v2: %5d v3: %5d v4: %5d\n", v1, v2, v3, v4);
    }

    return 0;
}
```

清单2-10 MemoryAddressing_.asm

```
        .model flat,c

; 简单查找表（.const 数据段只读）

        .const
FibVals dword 0, 1, 1, 2, 3, 5, 8, 13
        dword 21, 34, 55, 89, 144, 233, 377, 610
```

```
NumFibVals_ dword ($ - FibVals) / sizeof dword
            public NumFibVals_

; extern "C" int MemoryAddressing_(int i, int* v1, int* v2, int* v3, int*
↪  v4);
;
; 描述：这个函数演示了用于访问内存中操作数的多种寻址模式
;
; 返回值：0 = error(无效的表索引)
;        1 = success

        .code
MemoryAddressing_ proc
        push ebp
        mov ebp,esp                                                    [42]
        push ebx
        push esi
        push edi

; 确保'i'有效
        xor eax,eax
        mov ecx,[ebp+8]                 ;ecx = i
        cmp ecx,0
        jl InvalidIndex                 ;若 i < 0则跳转
        cmp ecx,[NumFibVals_]
        jge InvalidIndex                ;若 i >=NumFibVals_则跳转

; 示例1—基址寄存器
        mov ebx,offset FibVals          ;ebx = FibVals
        mov esi,[ebp+8]                 ;esi = i
        shl esi,2                       ;esi = i * 4
        add ebx,esi                     ;ebx = FibVals + i * 4
        mov eax,[ebx]                   ;加载表值
        mov edi,[ebp+12]
        mov [edi],eax                   ;保存到'v1'

; 示例2—基址寄存器+偏移量
; esi用作基址寄存器
        mov esi,[ebp+8]                 ;esi = i
        shl esi,2                       ;esi = i * 4
        mov eax,[esi+FibVals]           ;加载表值
        mov edi,[ebp+16]
        mov [edi],eax                   ;保存到'v2'

; 示例3—基址寄存器+索引寄存器
        mov ebx,offset FibVals          ;ebx = FibVals
        mov esi,[ebp+8]                 ;esi = i
        shl esi,2                       ;esi = i * 4
        mov eax,[ebx+esi]               ;加载表值
        mov edi,[ebp+20]
        mov [edi],eax                   ;保存到'v3'

; 示例4—基址寄存器+索引寄存器*比例因子
        mov ebx,offset FibVals          ;ebx = FibVals
        mov esi,[ebp+8]                 ;esi = i
        mov eax,[ebx+esi*4]             ;加载表值
        mov edi,[ebp+24]
        mov [edi],eax                   ;保存到'v4'
        mov eax,1                       ;设置成功返回码                [43]

InvalidIndex:
        pop edi
```

```
        pop esi
        pop ebx
        pop ebp
        ret
MemoryAddressing_ endp
        end
```

我们先分析名为 MemoryOperands_ 的汇编语言函数。在这个函数中，参数 i 的作用是作为包含整数常量的数组（或者称为查找表（lookup table））的索引。四个指针变量用来保存从字典表中使用不同寻址模式读到的值。紧接清单 2-10 顶部的那方右一个 .const 指示符，它的作用是定义包含只读数据的内存块。紧跟着 .const 指示符定义的就是名为 FibVals 的字典表。这个表包含 16 个双字整数；文字 dword 也是一个汇编器指示符，用来分配存储空间，并且可以选择性地初始化双字值（也可以使用 dd，它是 dword 的简写）。

NumFibVals_ dword ($ - FibVals) / sizeof dword 这一行为一个双字变量分配存储空间并把它初始化为字典表 FibVals 中包含的双字元素的个数。其中的 $ 字符是一个汇编器符号，代表的是位置计数器（location counter）（或基于当前内存块的偏移）的当前值。从 $ 减去 FibVals 的偏移得到的是以字节为单位的字典表大小，将其除以每个双字数据的字节数便得到了正确的元素个数。.const 节的几条语句与如下 C++ 语言中常用的定义和初始化数组中元素个数的变量的代码是等价的。

```
int Values[] = {10, 20, 30, 40, 50};
int NumValues = sizeof(Values) / sizeof(int);
```

.const 节的最后一行把 NumFibVals_ 声明为公共符号，以便在 _tmain 函数中也可以使用它。

下面我们来看一下 MemoryAddressing_ 函数的汇编语言代码。紧邻函数序言的是检查参数 i 的合理性，因为它会被用作字典表 FibVals 的索引。指令 cmp ecx, 0（比较两个操作数）把 ECX 寄存器（包含 i）和立即数 0 做比较。处理器通过从目标操作数中减去源操作数来执行比较动作，并根据相减的结果设置状态标志（结果并不保存到目标操作数）。如果条件 ecx < 0 为真，那么程序会转移到 jl（Jump if Less，小于则跳转）指令所指定的位置。接下来使用了类似的几条指令来判断 i 的值是否太大。对于这种情况，cmp ecx, [NumFibVals_] 指令把 ECX 和字典表的元素个数做比较。如果 ecx >= [NumFibVals]，那么程序会跳转到指令 jge（Jum if Greater or Equal，大于等于则跳转）所指定的目标位置。

MemoryAddressing_ 函数的其余指令演示了如何使用不同的内存寻址模式来访问字典表。第一个例子使用一个单一的基址寄存器来从表中读取一项。为了只使用一个基址寄存器，函数必须手工计算第 i 个表元素的地址。这是通过把 FibVals 的偏移（起始地址）与 i * 4 相加做到的。指令 mov ebx, offset FibVals 把字典表的起始地址加载到 EBX 寄存器。然后把 i 的值加载到 ESI 寄存器。紧跟的 shl esi, 2（逻辑左向移位）指令计算出第 i 个元素相对于字典表开始的偏移。add ebx, esi 指令计算最后的地址。一旦准备好元素的地址，就可以使用 mov eax, [ebx] 指令将其读出来，再保存到参数 v1 所指定的内存位置。

第二个例子演示的是使用 BaseReg+Disp 内存寻址模式来读取表项。与上一个例子类似，第 i 个表元素的偏移值（相对于表的起始地址）是通过 shl esi, 2 指令计算出来的。接下来使用 mov eax, [esi+FibVals] 指令来把正确的表项加载到 EAX 寄存器中。在这个例子中，处理器会通过把 ESI（基址寄存器）的内容与 FibVals（偏移）相加得到最终的有效地址。

第三个例子使用 BaseReg+IndexReg 内存寻址模式来读取表项。这个例子与第一个例子类似，不过是处理器在执行 mov eax, [ebx+esi] 指令时来计算最终的有效地址。第四个也是最后一个例子演示的是使用 BaseReg+IndexReg*ScaleFactor 模式来寻址。在这个例子中，FibVals 的偏移和 i 的值被分别加载到 EBX 寄存器和 ESI 寄存器。然后使用 mov eax, [ebx+esi*4] 指令来加载正确的表项到 EAX 寄存器中。

文件 MemoryAddressing.cpp（清单 2-10）中的主要代码是一个简单的循环，反复调用 MemoryOperands 函数，其中包含使用无效的索引值做调用。注意在循环中使用了 NumFibVals_ 变量，它是在汇编语言文件 MemoryOperands_.asm 中定义的公共符号。输出 2-4 是执行这个示例程序的结果。

输出 2-4 示例程序 MemoryAddressing

```
i: -1  rc:  0 - v1:    -1 v2:    -1 v3:    -1 v4:    -1
i:  0  rc:  1 - v1:     0 v2:     0 v3:     0 v4:     0
i:  1  rc:  1 - v1:     1 v2:     1 v3:     1 v4:     1
i:  2  rc:  1 - v1:     1 v2:     1 v3:     1 v4:     1
i:  3  rc:  1 - v1:     2 v2:     2 v3:     2 v4:     2
i:  4  rc:  1 - v1:     3 v2:     3 v3:     3 v4:     3
i:  5  rc:  1 - v1:     5 v2:     5 v3:     5 v4:     5
i:  6  rc:  1 - v1:     8 v2:     8 v3:     8 v4:     8
i:  7  rc:  1 - v1:    13 v2:    13 v3:    13 v4:    13
i:  8  rc:  1 - v1:    21 v2:    21 v3:    21 v4:    21
i:  9  rc:  1 - v1:    34 v2:    34 v3:    34 v4:    34
i: 10  rc:  1 - v1:    55 v2:    55 v3:    55 v4:    55
i: 11  rc:  1 - v1:    89 v2:    89 v3:    89 v4:    89
i: 12  rc:  1 - v1:   144 v2:   144 v3:   144 v4:   144
i: 13  rc:  1 - v1:   233 v2:   233 v3:   233 v4:   233
i: 14  rc:  1 - v1:   377 v2:   377 v3:   377 v4:   377
i: 15  rc:  1 - v1:   610 v2:   610 v3:   610 v4:   610
i: 16  rc:  0 - v1:    -1 v2:    -1 v3:    -1 v4:    -1
```

45

既然 x86 处理器中支持多种寻址模式，那么你可能想知道应该尽可能使用哪种模式。这个问题的答案与多个因素有关，包括寄存器的可用性、指令（或者指令序列）的预计执行次数、指令的顺序以及内存空间和执行时间的平衡等。还需要考虑一些硬件特征，比如处理器的流架构和高速缓存的大小。

当编写 x86 汇编语言函数时，一个指导原则是使用简单（一个寄存器或者偏移）的指令形式来引用内存中的操作数，而不是复杂的形式（多个寄存器）。这种方法的不足是往往需要程序员编写比较多的指令，可能需要占用较多的代码空间。简单形式可能带来的另一个优点是不需要额外指令在栈中保存非易变寄存器。第 21 和 22 章将详细讨论影响汇编语言程序效率的各种问题。

2.2.3 整数加法

Visual C++ 支持标准的 C++ 基础类型，比如 char、short、int、long long。这些刚好与 x86 的基础类型 byte、word、doubleword 和 quadword 一一对应。这一节的示例程序将演示对不同大小的整数做加法。你还会学习到如何在汇编语言函数中使用 C++ 文件中定义的全局变量以及一些常用的 x86 指令。这一节的 Visual Studio 解决方案文件叫 Chapter02\IntegerAddition\IntegerAddition.sln，清单 2-11 和清单 2-12 列出了源程序文件的内容。

清单 2-11 IntegerAddition.cpp

```
#include "stdafx.h"

extern "C" char GlChar = 10;
extern "C" short GlShort = 20;
extern "C" int GlInt = 30;
extern "C" long long GlLongLong = 0xCCCCCCCC3FFFFFFE;

extern "C" void IntegerAddition_(char a, short b, int c, long long d);

int _tmain(int argc, _TCHAR* argv[])
{
    printf("Before GlChar:     %d\n", GlChar);
    printf("       GlShort:    %d\n", GlShort);
    printf("       GlInt:      %d\n", GlInt);
    printf("       GlLongLong: %lld\n", GlLongLong);
    printf("\n");

    IntegerAddition_(3, 5, -37, 11);
    printf("After  GlChar:     %d\n", GlChar);
    printf("       GlShort:    %d\n", GlShort);
    printf("       GlInt:      %d\n", GlInt);
    printf("       GlLongLong: %lld\n", GlLongLong);
    return 0;
}
```

清单 2-12 IntegerAddition_.asm

```
        .model flat,c

; IntegerAddition.cpp 中有以下定义：
        extern GlChar:byte
        extern GlShort:word
        extern GlInt:dword
        extern GlLongLong:qword

; extern "C" void IntegerTypes_(char a, short b, int c, long long d);
;
; 描述：本函数演示了如何对不同大小的整数做加法操作
;
; 返回值：无

        .code
IntegerAddition_ proc

; 函数序言
        push ebp
        mov ebp,esp

; 计算 GlChar += a
        mov al,[ebp+8]
        add [GlChar],al

; 计算 GlShort += b，注意 'b' 在栈上的偏移
        mov ax,[ebp+12]
        add [GlShort],ax

; 计算 GlInt += c，注意 'c' 在栈上的偏移
        mov eax,[ebp+16]
        add [GlInt],eax
```

```
; 计算 GlLongLong += d, 注意 dword ptr 操作符和 adc 的用法
        mov eax,[ebp+20]
        mov edx,[ebp+24]
        add dword ptr [GlLongLong],eax
        adc dword ptr [GlLongLong+4],edx

; 函数结束
        pop ebp
        ret
IntegerAddition_ endp
        end
```

这个示例程序的 C++ 部分定义了四个全局变量，都是带符号整数，但基础类型各异。注意，每个变量的定义中都包含了标志 "C"，这是告诉编译器产生 C 风格的符号供链接器使用，不要产生 C++ 风格装饰过的符号。C++ 文件中包含了对汇编语言函数 IntegerAddition_ 的声明。这个函数会对四个全局变量做简单的整数加法。

靠近 IntegerAddition_.asm 顶部的是两条 extern 语句。与 C++ 中的对应部分类似，extern 指示符告诉汇编器这些名字的变量是定义在当前文件范围之外的。每个 extern 指示符还包含了变量的基本类型。指示符 extern 中还可以包含语言类型，来覆盖 .model flat, c 指示符中指定的默认类型。在本章后面的内容中，你会学习到如何使用 extern 指示符来引用外部函数。

在函数序言之后，IntegerAdditions_ 把参数 a 加载到寄存器 AL，然后使用 add [GlChar], al 指令计算 GlChar += a。接下来使用类似的形式来计算 GlShort += b，其中的寄存器 AX 是重要的差异；参数 b 在栈上的偏移是 +12，而不是你可能以为的 9。原因是 Visual C++ 在压栈时把 8 位和 16 位的数值扩展到 32 位，如图 2-4 所示。这样可以确保栈帧指针寄存器 ESP 总是可以按 32 位边界做内存对齐。

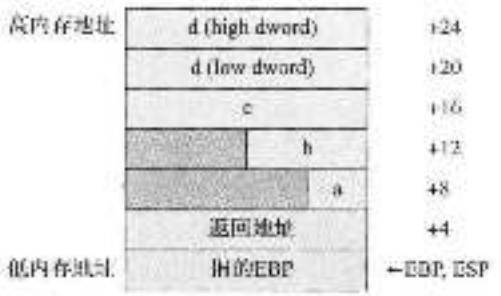


图 2-4 函数 IntegerAdditions_ 的栈参数布局

接下来使用一条 add [GlInt], eax 指令来计算 GlInt += c。参数 c 在栈上的偏移是 +16，它比 b 的偏移大 4 字节，而不是大 2 字节。最后的计算 GlLongLong+=d 使用 32 位加法指令进行。指令 add dword ptr [GlLongLong], eax 用来累加低地址的双字，adc dword ptr [GlLongLong+4], edx（Add With Carry，带进位加）用来完成 64 位加法。这里使用两条加法指令的原因是 x86-32 模式不支持用 64 位操作数做加法。指令 add 和 adc 包含了 dword ptr 运算符，因为 GlLongLong 变量的声明类型是四字（quadword）。执行这个 IntegerAddition 程序的结果见输出 2-5。

输出 2-5 示例程序 IntegerAddition

```
Before GlChar:     10
       GlShort:    20
       GlInt:      30
       GlLongLong: 4294967294

After  GlChar:     13
       GlShort:    25
       GlInt:      -7
       GlLongLong: 4294967305
```

2.2.4 条件码

这一节的最后一个示例程序演示如何使用x86的条件指令，包括jcc cmovcc(Conditional Move，条件移动）和setcc（Set Byte on Condition，根据条件设置字节）。条件指令的执行有很多不确定性，依赖于它指定的条件码和第1章讨论的状态标志。你已经看过几个条件转移指令的例子。函数IntegerMulDiv_（清单2-5）使用jz指令来预防可能的除零错误。在函数MemoryAddressingModes（清单2-10）中，cmp指令后的jl和jge指令用来验证表索引。这一节的示例程序名字是ConditionCodes，Visual Studio方案文件名为Chapter02\ConditionCodes\ConditionCodes.sln。清单2-13和清单2-14分别列出了ConditionCodes.cpp和ConditionCodes_asm的源代码。

清单2-13 ConditionCodes.cpp

```
#include "stdafx.h"

extern "C" int SignedMinA_(int a, int b, int c);
extern "C" int SignedMaxA_(int a, int b, int c);
extern "C" int SignedMinB_(int a, int b, int c);
extern "C" int SignedMaxB_(int a, int b, int c);

int _tmain(int argc, _TCHAR* argv[])
{
    int a, b, c;
    int smin_a, smax_a;
    int smin_b, smax_b;

    // SignedMin示例
    a = 2; b = 15; c = 8;
    smin_a = SignedMinA_(a, b, c);
    smin_b = SignedMinB_(a, b, c);
    printf("SignedMinA(%4d, %4d, %4d) = %4d\n", a, b, c, smin_a);
    printf("SignedMinB(%4d, %4d, %4d) = %4d\n\n", a, b, c, smin_b);

    a = -1; b = -22; c = 28;
    smin_a = SignedMinA_(a, b, c);
    smin_b = SignedMinB_(a, b, c);
    printf("SignedMinA(%4d, %4d, %4d) = %4d\n", a, b, c, smin_a);
    printf("SignedMinB(%4d, %4d, %4d) = %4d\n\n", a, b, c, smin_b);

    a = 17; b = 37; c = -11;
    smin_a = SignedMinA_(a, b, c);
    smin_b = SignedMinB_(a, b, c);
    printf("SignedMinA(%4d, %4d, %4d) = %4d\n", a, b, c, smin_a);
    printf("SignedMinB(%4d, %4d, %4d) = %4d\n\n", a, b, c, smin_b);

    // SignedMax示例
    a = 10; b = 5; c = 8;
    smax_a = SignedMaxA_(a, b, c);
    smax_b = SignedMaxB_(a, b, c);
    printf("SignedMaxA(%4d, %4d, %4d) = %4d\n", a, b, c, smax_a);
    printf("SignedMaxB(%4d, %4d, %4d) = %4d\n\n", a, b, c, smax_b);

    a = -3; b = 28; c = 15;
    smax_a = SignedMaxA_(a, b, c);
    smax_b = SignedMaxB_(a, b, c);
    printf("SignedMaxA(%4d, %4d, %4d) = %4d\n", a, b, c, smax_a);
    printf("SignedMaxB(%4d, %4d, %4d) = %4d\n\n", a, b, c, smax_b);

    a = -25; b = -37; c = -17;
```

```
    smax_a = SignedMaxA (a, b, c);
    smax_b = SignedMaxB_(a, b, c);
    printf("SignedMaxA(%4d, %4d, %4d) = %4d\n", a, b, c, smax_a);
    printf("SignedMaxB(%4d, %4d, %4d) = %4d\n\n", a, b, c, smax_b);
}
```

清单 2-14 ConditionCodes_.asm

```
        .model flat,c
        .code

; extern "C" int SignedMinA_(int a, int b, int c);
;
; 描述：使用条件跳转指令寻找三个带符号整数的最小值
;
; 返回值：min(a, b, c)

SignedMinA_ proc
        push ebp
        mov ebp,esp
        mov eax,[ebp+8]                 ;eax = 'a'
        mov ecx,[ebp+12]                ;ecx = 'b'

; 确定 min(a, b)
        cmp eax,ecx
        jle 答F
        mov eax,ecx                     ;eax = min(a, b)

; 确定 min(a, b, c)
答:     mov ecx,[ebp+16]                ;ecx = 'c'
        cmp eax,ecx
        jle 答+
        mov eax,ecx                     ;eax = min(a, b, c)

答:     pop ebp
        ret
SignedMinA_ endp

; extern "C" int SignedMaxA_(int a, int b, int c);
;
; 描述：使用条件跳转指令寻找三个带符号整数的最大值
;
; 返回值：max(a, b, c)

SignedMaxA_ proc
        push ebp
        mov ebp,esp
        mov eax,[ebp+8]                 ;eax = 'a'
        mov ecx,[ebp+12]                ;ecx = 'b'

        cmp eax,ecx
        jge 答+
        mov eax,ecx                     ;eax = max(a, b)

答:     mov ecx,[ebp+16]                ;ecx = 'c'
        cmp eax,ecx
        jge 答+
        mov eax,ecx                     ;eax = max(a, b, c)

答:     pop ebp
        ret
SignedMaxA_ endp
```

```
; extern "C" int SignedMin8_(int a, int b, int c);
;
; 描述：使用条件赋值指令寻找三个带符号整数的最小值
;
; 返回值：min(a, b, c)

SignedMin8_ proc
        push ebp
        mov ebp,esp
        mov eax,[ebp+8]                 ;eax = 'a'
        mov ecx,[ebp+12]                ;ecx = 'b'

; 使用 CMOVG 指令确定最小值
        cmp eax,ecx
        cmovg eax,ecx                   ;eax = min(a, b)
        mov ecx,[ebp+16]                ;ecx = 'c'
        cmp eax,ecx
        cmovg eax,ecx                   ;eax = min(a, b, c)

        pop ebp
        ret
SignedMin8_ endp

; extern "C" int SignedMax8 (int a, int b, int c);
;
; 描述：使用条件赋值指令寻找三个带符号整数的最大值
;
; 返回值：max(a, b, c)

SignedMax8_ proc
        push ebp
        mov ebp,esp
        mov eax,[ebp+8]                 ;eax = 'a'
        mov ecx,[ebp+12]                ;ecx = 'b'

; 使用 CMOVL 指令确定最大值
        cmp eax,ecx
        cmovl eax,ecx                   ;eax = max(a, b)
        mov ecx,[ebp+16]                ;ecx = 'c'
        cmp eax,ecx
        cmovl eax,ecx                   ;eax = max(a, b, c)

        pop ebp
        ret
SignedMax8_ endp
        end
```

[52]

当编写代码实现某个算法时，常常需要判断两个数的最小值和最大值。Visual C++ 定义了两个宏 __min 和 __max 来支持这样的操作。ConditionCodes .asm 文件中包含了二参数版本的带符号整数最小值和最大值函数。编写这些函数的目的是为了演示如何正确使用 jcc 和 cmovcc 指令。

第一个函数用来找三个带符号整数的最小值，取名为 SignedMinA_。在函数序言之后，第一块代码使用 cmp eax, ecx 和 jle @F 来确定 min(a, b)。如我们在本章前面所讲过的，cmp 指令会从目标操作数中减去源操作数，并根据结果（结果并不保存到目标操作数）设置状态标志。指令 jle（Jump if Less or Equal，小于等于则跳转）的目标是 @F。它是一个汇编器符号，其作用是把最近的前向 @@ 标记用作条件跳转指令的目标（符号 @B 可以用作后向跳转）。在计算 min(a, b) 之后，接下来的代码块判断 min(min(a, b), c)，使用的技术相同。因为结果已

经在 EAX 寄存器中，所以接下来 SignedMinA_ 便可以执行函数结语并返回到调用者。

函数 SignedMaxA_ 使用同样的方法来找三个带符号整数的最大值。SignedMaxA_ 和 SignedMinA_ 之间的唯一区别是使用 jge（Jump if Greater or Equal，大于等于则跳转）指令，而不是 jle 指令。

编写函数 SignedMaxA_ 和 SignedMinA_ 的无符号整数版本并不难，只要把 jle 和 jge 指令分别修改为 jbe（Jump if Below or Equal）和 jae（Jump if Above or Equal）即可。回忆我们在第 1 章中介绍的内容，greater 和 less 用在有符号整数操作数的情况，而 above 和 below 用在无符号整数类作数的情况。

SignedMinMax_.asm 文件中还包含了 SignedMinB_ 和 SignedMaxB_ 函数。它们使用条件赋值指令而不是条件跳转指令来确定三个带符号整数的最小值和最大值。指令 cmovcc 测试指定的条件是否为真，如果为真就把源操作数复制到目标操作数。如果指定的条件为假，那么目标操作数的值不变。

阅读 SignedMaxB_ 函数的代码，你可能注意到每条 cmp eax, ecx 指令之后都跟着一条 cmovg eax, ecx 指令。如果 EAX 大于 ECX，那么这条 cmovg 指令会把 ECX 的内容复制到 EAX。函数 SignedMinB_ 使用的方法非常类似，只不过使用了 cmovl 而不是 cmovg 来保存较小的带符号整数。这两个函数的无符号版本也很容易得到，只要用 cmova 和 cmovb 分别替换 cmovg 和 cmovl 就可以了。执行 ConditionCodes 示例程序的结果如输出 2-6 所示。

输出 2-6 示例程序 ConditionCodes

```
SignedMinA(   2,   15,    8) =    2
SignedMinB(   2,   15,    8) =    2

SignedMinA(  -3,   22,   28) =   -3
SignedMinB(  -3,   22,   28) =   -3

SignedMinA(  17,   37,   11) =   11
SignedMinB(  17,   37,   11) =   11

SignedMaxA(  10,    5,    8) =   10
SignedMaxB(  10,    5,    8) =   10

SignedMaxA(  -5,   28,   15) =   28
SignedMaxB(  -5,   28,   15) =   28

SignedMaxA( -35,  -37,  -17) =  -17
SignedMaxB( -35,  -37,  -17) =  -17
```

使用条件赋值指令来减少条件跳转指令常常可以提高代码的执行速度，特别是当处理器无法精确预测跳转分支的时候。第 21 章将详细讨论与条件跳转指令有关的优化问题。

我们最后要介绍的条件指令是 setcc 指令。如它的名字所暗示的，如果测试结果为真，那么 setcc 指令会把 8 位的目标操作数设置为 1，不然的话，目标操作数会被设置为 0。在返回或者设置布尔值时，这条指令很有用，如清单 2-15 所示。在本书后面的例子中你还会看到一些使用 setcc 指令的例子。

清单 2-15 根据条件设置字节（setcc）指令的用法

```
; extern "C" bool SignedIsEQ(int a, int b);

SignedIsEQ  proc
```

```
        push ebp
        mov ebp,esp

        xor eax,eax
        mov ecx,[ebp+8]
        cmp ecx,[ebp+12]
        setg al

        pop ebp
        ret
SignedIsIQ_ endp
```

2.3 数组

数组是几乎所有编程语言都不可缺少的数据结构。在C++中数组和指针有着直接的联系，数组的名字实际上就是指向第一个元素的指针。而且，当数组用作C++函数的参数时，传递的就是指针，而不是在栈上拷贝整个数组。在运行期动态分配数组时，也要用到指针。这一节将分析一些操作数组的x86-32汇编语言函数。第一个例子演示如何访问一个一维数组的元素。第二个例子演示使用输入和输出数组来处理元素。最后一个例子演示操作二维数组的一些技术。

54

在分析正式的示例代码之前，我们先对C++中的数组概念做个快速回顾。阅读清单2-16所示的简单程序。在函数CalcArrayCubes中，数组x和y的元素被保存在栈上的连续内存块内。调用函数CalcArrayCubes导致编译器从左至右向栈上压入三个参数：n的值、指向x的第一个元素的指针以及指向y的第一个元素的指针。CalcArrayCubes函数内的for循环中有一条语句int temp = x[i]，它会把数组x的第i个元素的值赋给temp。这个元素的地址就是x加上i*4，因为一个int整数的大小就是4个字节。也可以用同样的方法来计算数组y的元素地址。

清单2-16 CalcArrayCubes.cpp

```
#include "stdafx.h"

void CalcArrayCubes(int* y, const int* x, int n)
{
    for (int i = 0; i < n; i++)
    {
        int temp = x[i];
        y[i] = temp * temp * temp;
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    int x[] = { 2, 7, -4, 6, -9, 12, 10 };
    const int n = sizeof(x) / sizeof(int);
    int y[n];

    CalcArrayCubes(y, x, n);

    for (int i = 0; i < n; i++)
        printf("i: %4d x: %4d y: %4d\n", i, x[i], y[i]);
    printf("\n");

    return 0;
}
```

2.3.1 一维数组

这一小节将分析几个示例程序，它们用来演示如何在 x86 汇编语言中使用一维数组。第一个示例程序叫 CalcArraySum，它包含了一个可以对整数数组求和的函数，还演示了如何循环访问数组的每个元素。清单 2-17 和清单 2-18 分别列出了 CalArraySum.cpp 和 CalcArraySum_.asm 文件的源代码。 55

清单 2-17 CalcArraySum.cpp

```
#include "stdafx.h"

extern "C" int CalcArraySum_(const int* x, int n);

int CalcArraySumCpp(const int* x, int n)
{
    int sum = 0;

    for (int i = 0; i < n; i++)
        sum += *x++;

    return sum;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int x[] = {1, 7, -3, 5, 2, 9, -6, 12};
    int n = sizeof(x) / sizeof(int);

    printf("Elements of x[]\n");
    for (int i = 0; i < n; i++)
        printf("%d ", x[i]);
    printf("\n\n");

    int sum1 = CalcArraySumCpp(x, n);
    int sum2 = CalcArraySum_(x, n);

    printf("sum1: %d\n", sum1);
    printf("sum2: %d\n", sum2);
    return 0;
}
```

清单 2-18 CalcArraySum_.asm

```
        .model flat,c
        .code

; extern "C" int CalcArraySum_(const int* x, int n);
;
; 描述：对带符号整数数组中各元素求和

CalcArraySum_ proc
        push ebp
        mov ebp,esp

; 汇数参数并初始化和
        mov edx,[ebp+8]                 ;edx = 'x'
        mov ecx,[ebp+12]                ;ecx = 'n'
        xor eax,eax                     ;eax = 和

; 确保 'n' 大于 0
```

56

```
        cmp ecx,0
        jle InvalidCount

; 计算数组元素的和
L1:     add eax,[edx]                   ;把下一个元素累加到和
        add edx,4                       ;把指针指向下一个元素
        dec ecx                         ;调整计数器
        jnz L1                          ;如果没完成就重复

InvalidCount:
        pop ebp
        ret
CalcArraySum_ endp
        end
```

注意 这一章的其余例子和后面例子的 Visual Studio 方案文件使用的命名规则为：Chapter##\<ProgramName>\<ProgramName>.sln。其中 ## 代表章号，<ProgramName> 代表示例程序的名字。

CalcArraySum 程序的 C++ 部分（清单 2-17）包含了一个名为 CalcArraySumCpp 的测试函数，它把一个带符号整数数组的所有元素累加起来。尽管这种做法是没有必要的，但是先用 C++ 编写一个函数，然后再编写 x86 汇编语言的等价版本在软件测试和调试中常常是很有用的。汇编语言函数 CalcArraySum_（清单 2-18）与 CalcArraySum 的计算结果一样。在函数序言之后，把指向数组 x 的指针加载到 EDX 寄存器。接下来，参数 n 的值被复制到 ECX 寄存器中。紧接的是一条 xor eax, eax（逻辑异或）指令，它把和初始化为 0。

只要四条指令就可以遍历数组并累加元素。指令 add eax, [edx] 把当前的数组元素累加到和中，然后对 EDX 寄存器加 4 让其指向数组的下一个元素。指令 dec ecx 从计数器中递减 1 并更新 EFLAGS.ZF 状态。这可以让 jnz（Jump if not Zero，不为零则跳转）指令在所有 n 个元素都累加之后结束循环。这几条指令组成的序列与 CalcArraySumCpp 函数中的 for 循环是等价的。执行 CalcArraySum 程序的结果如输出 2-7 所示。

57

输出 2-7 示例程序 CalcArraySum

```
Elements of x[]
1 7 -3 5 2 9 -6 12

sum1: 27
sum2: 27
```

使用数组编程时，经常需要编写函数一个一个地处理数组元素。我们前面在函数 CalcArrayCubes（清单 2-16）中看到的便是一个例子，该函数对输入数组的每个元素求立方，并把结果保存到另一个数组中。下一个示例程序名为 CalcArraySquares，演示了如何使用汇编语言函数执行类似的操作。清单 2-19 和清单 2-20 分别给出了 CalcArraySquares.cpp 和 CalcArraySquares_.asm 的源代码。

清单 2-19 CalcArraySquares.cpp

```
#include "stdafx.h"

extern "C" int CalcArraySquares_(int* y, const int* x, int n);

int CalcArraySquaresCpp(int* y, const int* x, int n)
```

```
{
    int sum = 0;

    for (int i = 0; i < n; i++)
    {
        y[i] = x[i] * x[i];
        sum += y[i];
    }

    return sum;
}

int _tmain(int argc, _TCHAR* argv[])
{
    int x[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
    const int n = sizeof(x) / sizeof(int);
    int y1[n];
    int y2[n];
    int sum_y1 = CalcArraySquaresCpp(y1, x, n);
    int sum_y2 = CalcArraySquares_(y2, x, n);

    for (int i = 0; i < n; i++)
        printf("i: %2d  x: %4d  y1: %4d  y2: %4d\n", i, x[i], y1[i], y2[i]);
    printf("\n");
    printf("sum_y1: %d\n", sum_y1);
    printf("sum_y2: %d\n", sum_y2);

    return 0;
}
```

58

清单 2-20 CalcArrayCubes.cpp

```
        .model flat,c
        .code

; extern "C" int CalcArraySquares_(int* y, const int* x, int n);
;
; 描述：计算 y[i] = x[i] * x[i]
;
; 返回值：数组 y 中各个元素的和

CalcArraySquares_ proc
        push ebp
        mov ebp,esp
        push ebx
        push esi
        push edi

; 函数参数
        mov edi,[ebp+8]                 ;edi = 'y'
        mov esi,[ebp+12]                ;esi = 'x'
        mov ecx,[ebp+16]                ;ecx = 'n'

; 初始化和寄存器，计算数组的字节数，
; 并初始化元素的偏移量寄存器
        xor eax,eax             ;eax = 'y' 数组的和
        cmp ecx,0
        jle EmptyArray
        shl ecx,2               ;ecx = 数组的字节数
        xor ebx,ebx             ;ebx = 数组元素偏移
```

```
; 重复循环，直至结束
L0:     mov edx,[esi+ebx]           ;加载下一个x[i]
        imul edx,edx                ;计算x[i] *x[i]
        mov [edi+ebx],edx           ;保存结果到y[i]
        add eax,edx                 ;更新和
        add ebx,4                   ;更新数组元素偏移
        cmp ebx,ecx
        jl L0                       ;若没有完成则跳转

EmptyArray:
        pop edi
        pop esi
        pop ebx
        pop ebp
        ret
CalcArraySquares_ endp
        end
```

函数 CalcArraySquares_（清单 2-20）计算数组 x 中的每个元素的平方，然后把结果保存到数组 y 的对应元素中。它也计算 y 中的元素和。在函数序言之后，寄存器 ESI 和 EDI 分别被初始化为指向 x 和 y 的指针。而后加载 n 到寄存器 ECX，并把用来计算和的 EAX 初始化为 0。在检查 n 的有效性之后，使用 shl ecx, 2 指令来计算数组的字节数，这个值用来终止循环，然后把 EBX 寄存器初始化为 0，后面将用它来记录数组 x 和 y 中的偏移。

处理循环使用 mov edx, [esi+ebx] 指令来把 x[i] 加载到寄存器 EDX 中，然后使用双目形式的 imul 指令来求平方。然后使用 mov[edi+ebx], ebx 指令把该值保存到 y[i] 中。指令 add eax, edx 把 y[i] 累加到寄存器 EAX 保存的和中。指令 add ebx, 4 用来计算指向 x 和 y 的下一个元素的偏移值。只要 EBX 寄存器中的偏移值小于 ECX 寄存器中的数组长度（字节为单位），处理循环便反复执行。执行 CalcArraySquares 程序的结果如输出 2-8 所示。

输出 2-8 示例程序 CalcArraySquares

```
i:  0  x:   2  y1:   4  y2:   4
i:  1  x:   3  y1:   9  y2:   9
i:  2  x:   5  y1:  25  y2:  25
i:  3  x:   7  y1:  49  y2:  49
i:  4  x:  11  y1: 121  y2: 121
i:  5  x:  13  y1: 169  y2: 169
i:  6  x:  17  y1: 289  y2: 289
i:  7  x:  19  y1: 361  y2: 361
i:  8  x:  23  y1: 529  y2: 529
i:  9  x:  29  y1: 841  y2: 841

sum_y1: 2397
sum_y2: 2397
```

2.3.2 二维数组

在 C++ 中，可以使用一个连续的内存块来存储二维数组或者矩阵的元素。编译器可以产生代码，用简单的指针变换来访问每个矩阵元素。程序员也可以手工调整指针来操作矩阵。本节的示例程序演示使用 x86 汇编语言来访问矩阵的元素。在阅读汇编程序之前，我们先仔细看一下 C++ 是如何处理内存中的矩阵的。

C++ 使用行优先（row-major）的排序方法在内存中组织二维矩阵。这种排序方法以“先

按行再按列”的方式来组织元素。举例来说，矩阵 int x[3][2] 的元素在内存中的顺序为：x[0][0]、x[0][1]、x[1][0]、x[1][1]、x[2][0]、x[2][1]。为了能访问数组中的特定元素，C++ 编译器必须知道行和列的索引、总的列数，以及起始地址。有了这些信息之后，便可以使用指针来访问元素了，如清单 2-21 所示。

清单 2-21 CalcMatrixCubes.cpp

```
#include "stdafx.h"

void CalcMatrixCubes(int* y, const int* x, int nrows, int ncols)
{
    for (int i = 0; i < nrows; i++)
    {
        for (int j = 0; j < ncols; j++)
        {
            int k = i * ncols + j;
            y[k] = x[k] * x[k] * x[k];
        }
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    const int nrows = 4;
    const int ncols = 3;
    int x[nrows][ncols] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 },
➥   { 10, 11, 12 } };
    int y[nrows][ncols];

    CalcMatrixCubes(&y[0][0], &x[0][0], nrows, ncols);

    for (int i = 0; i < nrows; i++)
    {
        for (int j = 0; j < ncols; j++)
            printf("(%2d, %2d): %6d, %6d\n", i, j, x[i][j], y[i][j]);
    }

    return 0;
}
```

61

在函数 CalcMatrixCubes（清单 2-21）中，某个矩阵元素的偏移是通过公式 i * ncols + j 唯一确定的，其中 i 和 j 是矩阵的行和列索引。在计算元素的偏移之后，便可以使用同样的格式来引用矩阵元素了，这与在一维数组中引用元素的方式是一样的。这一节的示例程序（名为 CalcMatrixRowColSums）使用上述方法来对矩阵的行和列来求和。清单 2-22 和清单 2-23 分别列出了 CalcMatrixRowColSums.cpp 和 CalcMatrixRowColSums_asm 的源代码。

清单 2-22 CalcMatrixRowColSums.cpp

```
#include "stdafx.h"
#include <stdlib.h>

// 函数 PrintResults 定义在 CalcMatrixRowColSumsMisc.cpp 中
extern void PrintResults(const int* x, int nrows, int ncols, int* row_sums,
➥   int* col_sums);

extern "C" int CalcMatrixRowColSums (const int* x, int nrows, int ncols,
➥   int* row_sums, int* col_sums);
```

```
void CalcMatrixRowColSumsCpp(const int* x, int nrows, int ncols, int*
➥ row_sums, int* col_sums)
{
    for (int j = 0; j < ncols; j++)
        col_sums[j] = 0;

    for (int i = 0; i < nrows; i++)
    {
        row_sums[i] = 0;
        int k = i * ncols;

        for (int j = 0; j < ncols; j++)
        {
            int temp = x[k + j];
            row_sums[i] += temp;
            col_sums[j] += temp;
        }
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    const int nrows = 7, ncols = 5;
62  int x[nrows][ncols];

    // 初始化测试矩阵
    srand(31);
    for (int i = 0; i < nrows; i++)
    {
        for (int j = 0; j < ncols; j++)
            x[i][j] = rand() % 100;
    }

    // 计算行和列的和
    int row_sums1[nrows], col_sums1[ncols];
    int row_sums2[nrows], col_sums2[ncols];

    CalcMatrixRowColSumsCpp((const int*)x, nrows, ncols, row_sums1,
    ➥ col_sums1);
    printf("\nResults using CalcMatrixRowColSumsCpp()\n");
    PrintResults((const int*)x, nrows, ncols, row_sums1, col_sums1);

    CalcMatrixRowColSums_((const int*)x, nrows, ncols, row_sums2,
    ➥ col_sums2);
    printf("\nResults using CalcMatrixRowColSums_()\n");
    PrintResults((const int*)x, nrows, ncols, row_sums2, col_sums2);

    return 0;
}
```

清单 2-23 CalcMatrixRowColSums_.asm

```
        .model flat,c
        .code

; extern "C" int CalcMatrixRowColSums_(const int* x, int nrows, int ncols,
➥➥ int* row_sums, int* col_sums);
;
; 描述：对二维数组中各行和列求和
;
; 返回值：0 - 'nrows' 或 'ncols' 无效或
```

```
;       1 = 成功

CalcMatrixRowColSums_ proc
        push ebp
        mov ebp,esp
        push ebx
        push esi
        push edi                                                    63

; 确保 'nrows' 和 'ncols' 二者有效
        xor eax,eax                     ;假设返回值
        cmp dword ptr [ebp+12],0        ;[ebp+12] = 'nrows'
        jle InvalidArg                  ;若 nrows <= 0, 则跳转
        mov ecx,[ebp+16]                ;ecx = 'ncols'
        cmp ecx,0
        jle InvalidArg                  ;jump if ncols <= 0

; 初始化 'col_sums' 数组各元素为零
        mov edi,[ebp+24]                ;edi = 'col_sums'
        xor eax,eax                     ;eax = 要填充的值
        rep stosd                       ;将数组填充为全零

; 初始化外层循环变量
        mov eax,[ebp+8]                 ;eax = 'x'
        xor esi,esi                     ;i = 0

; 外层循环
lp1:    mov edi,[ebp+20]                ;edi = 'row_sums'
        mov dword ptr [edi+esi*4],0     ;row_sums[i] = 0

        xor edi,edi                     ;j = 0
        mov edx,esi                     ;edx = i
        imul edx,[ebp+16]               ;edx = i * ncols

; 内层循环
lp2:    mov ecx,edx                     ;ecx = i * ncols
        add ecx,edi                     ;ecx = i * ncols + j
        mov eax,[ebx+ecx*4]             ;eax = x[i * ncols + j]
        mov ecx,[ebp+20]                ;ecx = 'row_sums'
        add [ecx+esi*4],eax             ;row_sums[i] += eax
        mov ecx,[ebp+24]                ;ecx = 'col_sums'
        add [ecx+edi*4],eax             ;col_sums[j] += eax

; 内层循环已经结束?
        inc edi                         ;j++
        cmp edi,[ebp+16]
        jl lp2                          ;若 j < ncols, 则跳转

; 外层循环是否结束?
        inc esi                         ;i++
        cmp esi,[ebp+12]
        jl lp1                          ;若 i < nrows, 则跳转
        mov eax,1                       ;设为成功返回码            64

InvalidArg:
        pop edi
        pop esi
        pop ebx
        pop ebp
        ret
CalcMatrixRowColSums_ endp
        end
```

先来看一下行-列求和算法的C++实现。清单2.22包含了一个名为CalcMatrixRowColSumsCpp的函数。这个函数遍历输入矩阵x，每一次循环时，它把当前矩阵元素累加到数组row_sums和col_sums的合适项中。函数CalcMatrixRowColSumsCpp使用了前面讨论的指针算术技术来唯一地引用矩阵元素。

清单2-23显示了行-列求和算法的汇编语言版本。在函数序言之后，先检查参数nrows和ncols能否被整除。接下来用rep stosd（Repeat Store String Doubleword，重复存储串双字）指令来把col_sums的元素都初始化为0。指令stosd把寄存器EAX内容存储到EDI指向的内存单元，然后更新EDI使其指向下一个数组元素。其中的rep助记符是指令前缀，它告诉处理器使用ECX寄存器作为计数器循环执行存储操作。在每次存储操作之后，ECX会被递减1，stosd反复执行，直到ECX等于0。在本章后面我们还会更详细地介绍x86的串处理指令。

函数CalcMatrixRowColSums_使用寄存器EBX来存放输入矩阵x的基址。寄存器ESI和EDI分别包含行和列索引。每个外层循环首先把row_sums[i]初始化为0，然后计算出k值。在每个内层循环中，计算当前矩阵元素的最终偏移。然后使用指令mov eax,[ebx+ecx*4]把矩阵元素加载到EAX中。接下来，再把EAX累加到数组row_sums和col_sums的对应项中。这一过程反复进行，直到矩阵x中的所有元素都被累加到总和数组中为止。注意这个函数广泛使用了BaseReg+IndexReg*ScaleFactor形式的内存寻址模式。这使得从矩阵x中加载元素和更新row_sums和col_sums中的元素变得很简单，如图2-5所示。执行这个CalcMatrixRowColSums程序的结果如输出2-9所示。

65

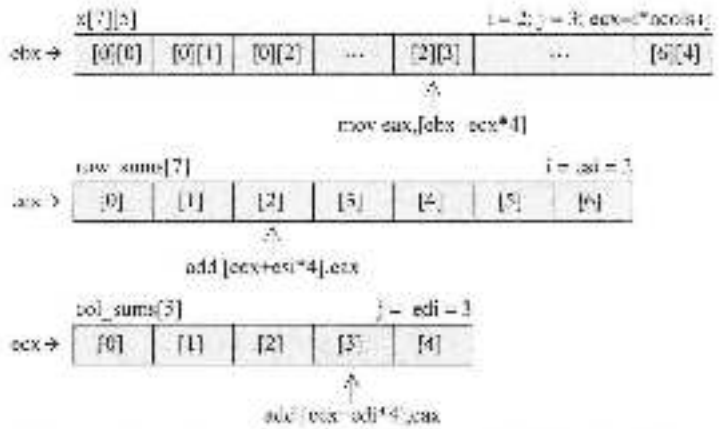


图2-5 函数CalcMatrixRowColSums_中使用的内存寻址模式

输出2-9 示例程序CalcMatrixRowColSums

```
Results using CalcMatrixRowColSumsCpp()
  81   76   96   48   72  ...  373
  76   53   99   93   23  ...  350
  80   73    4   75   24  ...  206
  48   99   69   96   88  ...  392
  37   67   40   92   88  ...  324
  15   80   16   62   72  ...  245
  93   23    4   55   22  ...  194

 469  477  528  521  388

Results using CalcMatrixRowColSums_()
  81   76   96   48   72  ...  373
```

76	59	99	93	23	--	350
40	73	4	75	23	--	205
40	99	69	96	88	--	392
37	67	40	92	88	--	324
15	83	26	62	73	--	249
90	23	4	56	33	--	194
369	477	328	521	388		

66

2.4 结构体

结构体是编程语言的一种基础构件，通过让程序员可以使用已经存在的数据类型定义新的数据类型。Visual C++ 和 MASM 都支持结构体。在这一节中，你将学习如何定义一个在 C++ 和汇编语言中都可以使用的公共结构体。我们还会讨论定义多种语言都要使用的结构体时程序员要注意的问题。除了基本的结构体用法，这一节的示例程序还会演示如何在汇编语言函数中调用标准的 C++ 库函数。你还会学到几条新的 x86-32 汇编语言指令。

在 C++ 中，结构体等价于类（class）。当使用 struct 关键字（而不是 class）定义一个数据类型时，默认所有的成员都是公开的。另一种定义结构体的方法是使用 C 风格的定义，比如 typedef struct{…}MyStruct;。这种风格适合定义只包含数据成员的结构体，本节的例子使用的就是这种方式。C++ 的结构体定义通常放在头文件里，以便可以很方便地在多个文件中引用。这种技术对于汇编语言编程也是适用的。不幸的是，没有办法只在一个头文件里定义结构体而在 C++ 和汇编语言源代码里都能引用。如果你要在 C++ 和汇编语言里使用同一个结构体，那么必须定义两次，而且必须保证它们在语义上等价。

2.4.1 简单结构体

我们要学习的第一个示例程序叫 CalcStructSum。这个程序演示了如何在两个函数中使用同一个基本的结构体，这两个函数一个是用 C++ 编写的，另一个是用 x86 汇编语言编写的。清单 2-24 和清单 2-25 各定义了一个简单的结构体，名为 TestStruct，这一节的所有示例程序都会使用这个结构体定义。

清单 2-24 TestStruct.h

```
typedef struct
{
    __int8  Val8;
    __int8  Pad8;
    __int16 Val16;
    __int32 Val32;
    __int64 Val64;
} TestStruct;
```

67

清单 2-25 TestStruct_inc

```
TestStruct struct
Val8    byte ?
Pad8    byte ?
Val16   word ?
Val32   dword ?
Val64   qword ?
TestStruct ends
```

清单 2-24 中的 C++ 定义使用了包含大小前缀数类型来定义每个成员，没有使用更常见的 ANSI 类型。包括我在内的很多开发者都喜欢使用包含大小的整数类型来定义结构体和函数参数，因为这样可以明确描述要操纵的数据类型的长度。关于 TestStruct 的另一个值得注意的细节是结构体中的 Pad8 成员。虽然这样做不是强制要求的，但是这种写法有助于提醒 C++ 编译器默认会把结构体成员按照自然边界对齐的事实。清单 2-25 中的 TestStruct 汇编版本看起来和它的 C++ 版本很类似。最大的不同是，汇编器不会对结构体成员按照其自然边界自动对齐。这里也必须定义 Pad8 成员。没有这个 Pad8 成员的话，C++ 版本和汇编版本在语义上就不同了。每个数据元素声明后面的问号（?）通知汇编器只进行存储空间分配，同时也是提醒程序员这样的结构体成员是没有初始化的。

清单 2-26 和清单 2-27 分别列出了 CalcStructSum 示例程序的 C++ 和汇编语言源代码。这个程序的 C++ 部分是很直截了当的。_tmain 函数声明了一个 TestStruct 结构体的实例，名为 ts。在初始化 ts 之后，调用了 CalcStructSumCpp 函数，这个函数会对 ts 的成员求和并返回 64 位整数的结果。而后 _tmain 函数会调用汇编语言函数 CalcStructSum_ 来做同样的求和计算。

清单 2-26 CalcStructSum.cpp

```
#include "stdafx.h"
#include "TestStruct.h"

extern "C" __int64 CalcStructSum_(const TestStruct* ts);

__int64 CalcStructSumCpp(const TestStruct* ts)
{
    return ts->Val8 + ts->Val16 + ts->Val32 + ts->Val64;
}

int _tmain(int argc, _TCHAR* argv[])
{
    TestStruct ts;

    ts.Val8  = -100;
    ts.Val16 = 2000;
    ts.Val32 = -300000;
    ts.Val64 = 40000000000;

    __int64 sum1 = CalcStructSumCpp(&ts);
    __int64 sum2 = CalcStructSum_(&ts);

    printf("Input: %d  %d  %d  %lld\n", ts.Val8, ts.Val16, ts.Val32,
    ↪ ts.Val64);
    printf("sum1:  %lld\n", sum1);
    printf("sum2:  %lld\n", sum2);

    if (sum1 != sum2)
        printf("Sum verify check failed!\n");

    return 0;
}
```

清单 2-27 CalcStructSum_.asm

```
        .model flat,c
        include TestStruct_.inc
        .code
```

```
; extern "C" __int64 CalcStructSum_(const TestStruct* ts);
;
; 描述：对 TestStruc 的成员求和
;
; 返回值：以 64 位整数表达的 ts 成员的和

CalcStructSum_ proc
        push ebp
        mov ebp,esp
        push ebx
        push esi

; 计算 ts->Val8 + ts->Val16，注意符号扩展到 32 位
        mov esi,[ebp+8]
        movsx eax,byte ptr [esi+TestStruct.Val8]
        movsx ecx,word ptr [esi+TestStruct.Val16]
        add eax,ecx

; 把和符号扩展到 64 位，结果保存到 ebx:ecx
        cdq
        mov ebx,eax
        mov ecx,edx

; 把 ts->Val32 累加到和
        mov eax,[esi+TestStruct.Val32]
        cdq
        add eax,ebx
        adc edx,ecx

; 把 ts->Val64 累加到和
        add eax,dword ptr [esi+TestStruct.Val64]
        adc edx,dword ptr [esi+TestStruct.Val64+4]

        pop esi
        pop ebx
        pop ebp
        ret
CalcStructSum_ endp
        end
```

在函数序言后，清单 2-27 中的 CalcStructSum_ 把指向 ts 结构体的指针加载到 ESI 寄存器。接下来的两条 movsx（Move with Sign Extension，带符号扩展赋值）指令分别把 ts->Val8 和 ts->Val16 加载到 EAX 和 ECX 寄存器。movsx 指令在把源操作数复制到目标操作数之前，会先创建一个临时拷贝，然后做符号扩展。正如这个函数所演示的，movsx 指令经常被用来把 8 位或者 16 位的源操作数加载到 32 位的寄存器。这两条 movsx 指令也演示了如何在汇编语言指令中引用结构体成员。从汇编器的角度来看，指令 movsx ecx, word ptr [esi+TestStruct.Val16] 就是 BaseReg+Disp 的内存寻址模式，因为汇编器会把 TestStruct.Val16 这样的结构体成员标识符消解为一个偏移常量。

CalcStructSum 函数使用 add eax, ecx 指令来计算 ts->Val8 和 ts->Val16 的和，然后使用 cdq 指令把得到的和做符号扩展，把 64 位的结果复制到 ECX:EBX 寄存器对。接下来，ts->Val32 的值被加载到 EAX 寄存器，符号扩展到 EDX:EAX 寄存器对，然后用 add 和 adc 指令将其与前面的临时结果累加。下一步是累加结构体的最后一个成员 ts->Val64，得到最终结果。Visual C++ 的调用约定要求 64 位的返回值放在 EDX:EAX 寄存器对中。既然最终结果已经在所要求的寄存器对中，那么就不用更多的 mov 指令了。输出 2-10 给出了运行这

个 CalcStructSum 程序的结果。

输出 2-10 示例程序 CalcStructSum

```
Input: -100  2000  -400000  4000000000
sum1:  3999970100
sum2:  3999970100
```

2.4.2 动态结构体创建

很多 C++ 程序使用 new 运算符来动态创建类或者结构体的实例。对于 TestStruct 这样只包含数据成员的结构体来说，可以使用标准库中的 malloc 函数来在运行期分配内存空间，
70 创建新的结构体实例。在这一节，你将看到如何在 x86 汇编语言中动态创建结构体实例，还会学习如何在汇编语言函数中调用 C++ 库函数。这一节的示例程序命名为 CreateStruct。清单 2-28 和清单 2-29 分别列出了 C++ 和汇编语言源文件。

清单 2-28 CreateStruct.cpp

```
#include "stdafx.h"
#include "TestStruct.h"

extern "C" TestStruct* CreateTestStruct_(__int8 val8, __int16 val16, __int32
↪ val32, __int64 val64);
extern "C" void ReleaseTestStruct_(TestStruct* p);

void PrintTestStruct(const char* msg, const TestStruct* ts)
{
    printf("%s\n", msg);
    printf("  ts->Val8:   %d\n", ts->Val8);
    printf("  ts->Val16:  %d\n", ts->Val16);
    printf("  ts->Val32:  %d\n", ts->Val32);
    printf("  ts->Val64:  %lld\n", ts->Val64);
}

int _tmain(int argc, _TCHAR* argv[])
{
    TestStruct* ts = CreateTestStruct_(40, -401, 400002, -4000000003LL);

    PrintTestStruct("Contents of TestStruct 'ts'", ts);

    ReleaseTestStruct_(ts);
    return 0;
}
```

清单 2-29 CreateStruct_.asm

```
        .model flat,c
        include TestStruct_.inc
        extern malloc:proc
        extern free:proc
        .code

; extern "C" TestStruct* CreateTestStruct_(__int8 val8, __int16 val16,
↪   __int32 val32, __int64 val64);
;
; 描述：创建一个新的 TestStruct 并初始化
;
71 ; 返回值：指向新的 TestStruct 的指针或者 NULL（如果发生错误）
```

```
CreateTestStruct_ proc
        push ebp
        mov ebp,esp

; 为新的 TestStruct 分配内存块，注意 malloc() 返回指针在 EAX 中
        push sizeof TestStruct
        call malloc
        add esp,4
        or eax,eax                          ; NULL 指针测试
        jz MallocError                      ; 若 malloc 失败，跳转

; 初始化新的 TestStruct
        mov dl,[ebp+8]
        mov [eax+TestStruct.Val8],dl

        mov cx,[ebp+12]
        mov [eax+TestStruct.Val16],cx

        mov ecx,[ebp+16]
        mov [eax+TestStruct.Val32],ecx

        mov ecx,[ebp+20]
        mov edx,[ebp+24]
        mov dword ptr [eax+TestStruct.Val64],ecx
        mov dword ptr [eax+TestStruct.Val64+4],edx

MallocError:
        pop ebp
        ret
CreateTestStruct_ endp

; extern "C" void ReleaseTestStruct (const TestStruct* p);
;
; 描述：释放前面创建的 TestStruct
;
; 返回值：无

ReleaseTestStruct_ proc
        push ebp
        mov ebp,esp

; 调用 free() 来释放前面创建的结构体
        push [ebp+8]
        call free
        add esp,4
        pop ebp
        ret
ReleaseTestStruct_ endp
        end
```

清单 2-28 中的 C++ 部分简单易懂，它包含了一些简单的测试代码来调用汇编语言函数 CreateTestStruct_ 和 ReleaseTestStruct_。在 CreateStructure_.asm（清单 2-19）的开头，extern malloc:proc 语句声明了外部 C—— 库函数 malloc。紧接的另一个 extern 语句是用来声明 free 函数的。与 C++ 版本不同，汇编语言中的 extern 语句不支持函数参数和返回类型。这意味着汇编器不能帮我们做静态的类型检查，程序员必须自己负责把正确的参数放在栈上。

在 CreateTestStruct_ 函数的序言之后，它先使用 malloc 函数来为 TestStruct 的新实例分配内存块。在使用 malloc 或者其他 C++ 运行时库函数时，调用函数必须遵守标准的 C++ 调用约定。指令 push sizeof TestStruct 把结构体 TestStruct 的字节数压到栈上。下面的 call

malloc 指令发起对 C++ 库函数的调用，而后的 add esp, 4 指令用来移除（释放）栈上的参数。与其他所有标准函数一样，malloc 也使用 EAX 来返回结果。在使用前我们先测试了返回指针的有效性。如果 malloc 返回的指针是有效的，新的结构体实例使用所提供的参数值进行初始化。最后把这个指针返回给调用者。

使用 malloc 分配的内存块在使用完毕后必须释放掉。要求调用者使用标准库函数 free 是一种办法，但是这样做就暴露了 CreateTestStruct_ 函数的内部细节，而且产生了不必要的依赖。在实际实现中，CreateTestStruct_ 可能使用一个池来管理预先分配好的 TestStruct 缓冲区。为此，CreateStructure_.asm 中定义了另一个名为 ReleaseTestStruct_ 的函数。在我们的例子中，ReleaseTestStruct_ 调用 free 来释放前面用 CreateTestStruct_ 函数分配的内存块。输出 2-11 给出了运行 CreateStruct 程序的结果。

输出 2-11 示例程序 CreateStruct

```
Contents of TestStruct 'ts'
  ts->Val8:   40
  ts->Val16:  -401
  ts->Val32:  400002
  ts->Val64:  -4000000003
```

2.5 字符串

x86 指令集包含了很多条指令来操纵字符串。按 x86 手册的说法，一个字符串就是由字节、字或者双字组成的一个连续序列。程序可以使用串指令来处理传统的“Hello, World”这样的文本串，也可以使用串指令来对数组或者内存块里的元素进行操作。在这一节中，我们将分析几个示例程序，学习如何使用 x86 的串指令来处理文本串和整数数组。

[73]

2.5.1 字符计数

这一节要学习的第一个示例程序名叫 CountChars，它演示的是如何使用 lods（Load String，加载字符串）指令来统计某个字符在一个文本串中的出现次数。清单 2-30 和清单 2-31 分别给出了这个程序的源代码。

清单 2-30 CountChars.cpp

```
#include "stdafx.h"

extern "C" int CountChars_(wchar_t* s, wchar_t c);

int _tmain(int argc, _TCHAR* argv[])
{
    wchar_t c;
    wchar_t* s;

    s = L"Four score and seven seconds ago, ...";
    wprintf(L"\nTest string: %s\n", s);
    c = L's';
    wprintf(L"  SearchChar: %c Count: %d\n", c, CountChars_(s, c));
    c = L'F';
    wprintf(L"  SearchChar: %c Count: %d\n", c, CountChars_(s, c));
    c = L'o';
    wprintf(L"  SearchChar: %c Count: %d\n", c, CountChars_(s, c));
    c = L'z';
```

```
    wprintf(L"  SearchChar: %c Count: %d\n", c, CountChars_(s, c));

    s = L"Red Green Blue Cyan Magenta Yellow";
    wprintf(L"\nTest string: %s\n", s);
    c = L'e';
    wprintf(L"  SearchChar: %c Count: %d\n", c, CountChars_(s, c));
    c = L'w';
    wprintf(L"  SearchChar: %c Count: %d\n", c, CountChars_(s, c));
    c = L'Q';
    wprintf(L"  SearchChar: %c Count: %d\n", c, CountChars_(s, c));
    c = L'l';
    wprintf(L"  SearchChar: %c Count: %d\n", c, CountChars_(s, c));

    return 0;
}
```

74

清单 2-31　CountChars_.asm

```
        .model flat,c
        .code

; extern "C" int CountChars_(wchar_t* s, wchar_t c);
;
; 描述：数字符 'c' 在串 's' 中的出现次数
;
; 返回值：字符 'c' 的匹配次数

CountChars_ proc
        push ebp
        mov ebp,esp
        push esi

; 加载参数并初始化计数寄存器
        mov esi,[ebp+8]                 ;esi = 's'
        mov cx,[ebp+12]                 ;cx = 'c'
        xor edx,edx                     ;edx = 匹配次数

; 重复循环直到扫描完整个串
@@:     lodsw                           ;把下一个字符加载到 ax
        or ax,ax                        ;测试是否结束
        jz @F                           ;如果发现结束则跳转
        cmp ax,cx                       ;测试当前字符
        jne @@                          ;如果不匹配则跳转
        inc edx                         ;更新匹配计数
        jmp @@

@@:     mov eax,edx                     ;eax = 字符计数
        pop esi
        pop ebp
        ret
CountChars_ endp
        end
```

汇编语言函数 CountChars_ 接受两个参数：文本串指针 s 和要搜索的字符 c。两个参数都使用了 wchar_t 类型，意味着文本串中的字符和要搜索的字符都是 16 位的。函数 CountChars_ 先是把 s 和 c 分别加载到 ESI 和 CX 寄存器，然后把 EDX 初始化为 0，以便可以用它来记录字符的出现次数。处理循环中使用了 lodsw（Load String Word，加载字符串字）指令来读每个文本串字符。这条指令把 ESI 指向的内存内容加载到 AX 寄存器，然后把 ESI

递加 2 以指向下一个字符。函数使用 or ax, ax 指令来测试字符串的结束字符（'\0'）。如果不
75 是串结束字符，就用 cmp ax, cx 指令来比较当前的文本串字符是否与要搜索的字符相同。如果检测到匹配，那么就把出现次数递增一。重复这个过程，直到遇到串结束字符。在扫描文本串之后，把最终的出现次数移动到 EAX 寄存器，并返回给调用者。输出 2-12 给出了执行这个程序的结果。

输出 2-12 示例程序 CountChars

```
Test string: Four score and seven seconds ago, ...
  SearchChar: s Count: 4
  SearchChar: F Count: 1
  SearchChar: a Count: 4
  SearchChar: z Count: 0

Test string: Red Green Blue Cyan Magenta Yellow
  SearchChar: e Count: 5
  SearchChar: w Count: 1
  SearchChar: q Count: 0
  SearchChar: l Count: 3
```

如果要把这个例子改写为处理 char 类型，而不是 wchar_t，那么很简单，只要把 lodsw 指令换为 lodsb（Load String Byte，加载字符串字节）指令即可。另外，应该使用 AL 而不是 AX 寄存器。x86 的指令的最后一个助记符总是用来表示要处理的操作数的大小。

2.5.2 字符串拼接

把两个文本串拼接在一起是很多程序经常要执行的操作。在 Visual C++ 应用程序中，可以使用库函数 strcat、strcat_s、wcscat 和 wcscat_t 来拼接两个字符串。这些函数的一个不足之处是它们只支持一个源串，如果要把多个串拼接在一起，就得要多次调用这些函数。这一节的示例程序命名为 ConcatStrings，演示如何用 scas（Scan String，扫描字符串）和 movs（Move String，移动字符串）指令拼接多个字符串。清单 2-32 和清单 2-33 分别给出了 ConcatStrings.cpp 和 ConcatStrings_.asm 文件的源代码。

清单 2-32 ConcatStrings.cpp

```
#include "stdafx.h"

extern "C" int ConcatStrings_(wchar_t* des, int des_size, const wchar_t* ↩
const* src, int src_n);

int _tmain(int argc, _TCHAR* argv[])
{
    printf("\nResults for ConcatStrings\n");

    // 目标缓冲区足够大
    wchar_t* src1[] = { L"One ", L"Two ", L"Three ", L"Four" };
    int src1_n = sizeof(src1) / sizeof(wchar_t*);
    const int des1_size = 64;
    wchar_t des1[des1_size];

    int des1_len = ConcatStrings_(des1, des1_size, src1, src1_n);
    wchar_t* des1_temp = (*des1 != '\0') ? des1 : L"empty";
    wprintf(L"  des_len: %d (%d) des: %s \n", des1_len, wcslen(des1_temp), ↩
    des1_temp);
```

```
    // 目标缓冲区太小
    wchar_t* src2_[] = { L"Red ", L"Green ", L"Blue ", L"Yellow " };
    int src2_n = sizeof(src2) / sizeof(wchar_t*);
    const int des2_size = 16;
    wchar_t des2[des2_size];

    int des2_len = ConcatStrings_(des2, des2_size, src2, src2_n);
    wchar_t* des2_temp = (*des2 != '\0') ? des2 : L"<empty>";
    wprintf(L"  des_len: %d (%d) des: %s \n", des2_len, wcslen(des2_temp), ↪
    des2_temp);

    // 测试空结果
    wchar_t* src3[] = { L"Airplane ", L"Car ", L"", L"Truck ", L"Boat " };
    int src3_n = sizeof(src3) / sizeof(wchar_t*);
    const int des3_size = 128;
    wchar_t des3[des3_size];

    int des3_len = ConcatStrings_(des3, des3_size, src3, src3_n);
    wchar_t* des3_temp = (*des3 != '\0') ? des3 : L"<empty>";
    wprintf(L"  des_len: %d (%d) des: %s \n", des3_len, wcslen(des3_temp), ↪
    des3_temp);

    return 0;
}
```

清单 2-33 ConcatStrings_.asm

```
        .model flat,c
        .code

; extern "C" int ConcatStrings_(wchar_t* des, int des_size, const wchar_t* ↪
 const* src, int src_n)
;
; 描述：对多个输入的字符串进行拼接
;
; 返回值：-1        如果 'des_size' 无效
;         n >=0     拼接好的字符串长度
;
; 局部变量：[ebp-4] = des_index
;           [ebp-8] = i

ConcatStrings_ proc
        push ebp
        mov ebp,esp
        sub esp,8
        push ebx
        push esi
        push edi

; 确保 'des_size' 有效
        mov eax,-1
        mov ecx,[ebp+12]                ;ecx = 'des_size'
        cmp ecx,0
        jle Error

; 目标缓冲区的初始化
        xor eax,eax
        mov ebx,[ebp+8]                 ;ebx = 'des'
        mov [ebx],ax                    ;*des = '\0'
        mov [ebp-4],eax                 ;des_index = 0
        mov [ebp-8],eax                 ;i = 0
```

```
; 重复循环，直到拼接完成
Lp1:    mov eax,[ebp-16]                ;eax = 'src'
        mov edx,[ebp-8]                 ;edx = i
        mov edi,[eax+edx*4]             ;edi = src[i]
        mov esi,edi                     ;esi = src[i]

; 计算 s[i] 的长度
        xor eax,eax
        mov ecx,-1
        repne scasw                     ;查找 '\0'
        not ecx
        dec ecx                         ;ecx = len(src[i])

; 计算 des_index + src_len
        mov eax,[ebp-4]                 ;eax= des_index
        mov edx,eax                     ;edx = des_index_temp
        add eax,ecx                     ;des_index + len(src[i])

; des_index + src_len >=des_size?
        cmp eax,[ebp+12]
        jge Done

; 更新 des_index
        add [ebp-4],ecx                 ;des_index += len(src[i])

; 复制 src[i] 到 &des[des_index] (esi 一般包含 src[i])
        inc ecx                         ;ecx = len(src[i]) + 1
        lea edi,[ebx+edx*2]             ;edi = &des[des_index_temp]
        rep movsw                       ;进行字符串移动

; 更新 i 并重复循环，直到完成
        mov eax,[ebp-8]
        inc eax
        mov [ebp-8],eax                 ;i++
        cmp eax,[ebp+20]
        jl Lp1                          ;若 i < src_n 则跳转

; 返回拼接好的串的长度
Done:   mov eax,[ebp-4]                 ;eax = des_index
Error:  pop edi
        pop esi
        pop ebx
        mov esp,ebp
        pop ebp
        ret
ConcatStrings_ endp
        end
```

78

我们先来看一下清单 2-32 中的 ConcatStrings.cpp。开头先是声明了 4 个汇编语言函数 ConcatStrings_，它包含四个参数：des 是存放最终串的目标缓冲区，des_size 是 des 缓冲区的大小，src 是指向数组的指针，数组里面包含了要拼接的文本串，文本串的个数是由 src_n 来指定的。函数 ConcatStrings_ 的返回值是 des 的长度或者 -1（如果 des_size 小于等于 0）。

tmain 函数中的测试代码演示了如何调用 ConcatStrings 函数。举例来说，如果 src 指向的文本串数组是 {"Red", "Green", "Blue"}，那么 des 中的最终串就是 "RedGreenBlue"，前提是 des 的大小足够大。如果 des 的大小不够，那么 ConcatStrings_ 会返回部分拼接的字符串。举例来说，如果 des_size 为 10，那么得到的最终串就是 "RedGreen"。

清单 2-32 中的 ConcatStrings_ 函数序言定义了两个局部变量：des_index 是串拷贝在

des 中的偏移，i 是当前串在 src 中的索引。在检查 des_size 的有效性之后，ConcatStrings 把 des 加载到 EBX 寄存器，并把缓存初始化为一个空的串。而后把 des_index 和 i 初始化为 0。随后的指令块开始了拼接循环，把字符串 src[i] 的指针加载到 ESI 和 EDI 寄存器。

接下来使用 repne scasw 和其他几条指令来确定 src[i] 的长度。其中的 repne（Repeat String Operation While not Equal，不相等时重复串操作）是一种指令前缀，当条件 ECX != 0 && EFLAGS.ZF == 0 为真时反复执行一条串指令。指令 repne scasw（Scan String Word，扫描字符串字）的确切操作是这样的：如果 ECX 不为 0，scasw 指令把 EDI 所指向的串字符与 AX 寄存器的内容做比较，并根据比较结果设置状态标志。然后自动把 EDI 寄存器递增 2，以指向下一个字符，并把 ECX 递减 1。这样的串处理操作被反复执行，只要前面提到的测试条件保持为真；不然的话，就结束循环。 79

在执行 repne scasw 指令之前，ECX 寄存器被初始化为 -1。在完成这条指令时，ECX 中包含的是 -（L+2），其中 L 代表的是字符串 src[i] 的实际长度。L 的值可以这样计算：先执行 not ecx（One's Complement Negation，1 取反），再 dec ECX（Decrement by 1，递减 1）。这等价于把 -（L+2）取反再减去 2。（这里描述的计算字符串长度的方法是很著名的 x86 编程技巧。）

在继续讲解之前，有必要指出 Visual C++ 的运行时环境假定 EFLAGS.DF 总是被清除的。如果某个汇编语言函数因为要对某个字符串做自动递减操作而设置了 EFLAGS.DF，那么在返回到调用者或者调用其他库函数之前，必须清除这个标志。后面的示例程序 ReverseArray 更详细地讨论了这个问题。

在计算好 len(src[i]) 之后，要做个检查，以确保 src[i] 可以放到目标缓冲区。如果 des_index + len(src[i]) 大于等于 des_size，那么这个函数就终止执行。否则，len(src[i]) 被累加到 des_index，再使用 rep movsw（Repeat Move String Word，重复移动字符串字）指令把 src[i] 复制到 des 中的合适位置。

指令 rep movsw 根据 ECX 寄存器中指定的长度，把 ESI 指向的串复制到 EDI 指向的内存位置。在执行串复制前，执行了一条 inc ecx 指令，目的是确保串结束字符 '\0' 也被复制到 des。初始化 EDI 寄存器到 des 中的正确位置的方法是使用 lea edi, [ebx+edx*2]（Load Effective Address，加载有效地址）指令计算源操作数的地址。之所以可以这样使用 lea 指令是因为 EBX 指向的是 des 的开始，EDX 中包含的是 des_index 与 len(src[i]) 累加之前的值。在复制串之后，更新变量 i 的值，如果它小于 src_n，那么就重复拼接循环。在完成拼接操作之后，把 des_index 加载到 EAX 寄存器，它就是 des 中的最终串的长度。输出 2-13 是执行 ConcatStrings 程序的结果。

输出 2-13 示例程序 ConcatStrings

```
Results for ConcatStrings
  des_len: 18 (18) des: One Two Three Four
  des_len: 15 (15) des: Red Green Blue
  des_len: 24 (24) des: Airplane Car Truck Boat
```

2.5.3 比较数组

正如这一节开头所说的，x86 的串指令也可以用来处理内存块。下面将讨论的 CompareArrays 程序演示的就是使用 cmps（Compare String Operands，比较串操作数）指令来

[80] 比较两个数组的元素。清单 2-34 和清单 2-35 分别给出了这个程序的 C++ 和汇编语言源代码。

清单 2-34 CompareArrays.cpp

```
#include "stdafx.h"
#include <stdlib.h>

extern "C" int CompareArrays (const int* x, const int* y, int n);

int _tmain(int argc, _TCHAR* argv[])
{
    const int n = 21;
    int x[n], y[n];
    int result;

    // 初始化测试数组
    srand(11);
    for (int i = 0; i < n; i++)
        x[i] = y[i] = rand() % 1000;

    printf("\nResults for CompareArrays\n");

    // 使用有效的 'n' 进行测试
    result = CompareArrays_(x, y, n);
    printf("  Test #1 - expected: %3d  actual: %3d\n",  1, result);

    // 测试第一个元素不匹配的情况
    x[0] -= 1;
    result = CompareArrays_(x, y, n);
    x[0] -= 1;
    printf("  Test #2 - expected: %3d  actual: %3d\n", 0, result);

    // 测试中间元素不匹配的情况
    y[n / 2] -= 2;
    result = CompareArrays_(x, y, n);
    y[n / 2] -= 2;
    printf("  Test #3 - expected: %3d  actual: %3d\n", n / 2, result);

    // 测试最后一个元素不匹配的情况
    x[n - 1] *= 3;
    result = CompareArrays_(x, y, n);
    x[n - 1] /=3;
    printf("  Test #4 - expected: %3d  actual: %3d\n", n - 1, result);

    // 测试两个数组完全相同的情况
    result = CompareArrays_(x, y, n);
    printf("  Test #5 - expected: %3d  actual: %3d\n", n, result);
    return 0;
}
```

[81]

清单 2-35 CompareArrays_.asm

```
        .model flat,c
        .code

; extern "C" int CompareArrays_(const int* x, const int* y, int n)
;
; 描述：逐一比较两个整数数组中的元素
;
; 返回值：-1          n 表明完全
;         0 <= i < n  第一个不匹配元素的索引
;         n           所有元素匹配
```

```
CompareArrays_ proc
        push ebp
        mov ebp,esp
        push esi
        push edi

; 加载并验证 'n'
        mov eax,-1                      ;无效的 'n' 返回码
        mov esi,[ebp+8]                 ;esi = 'x'
        mov edi,[ebp+12]                ;edi = 'y'
        mov ecx,[ebp+16]                ;ecx = 'n'
        test ecx,ecx
        jle @F                          ;若 'n' <= 0 则跳转
        mov eax,ecx                     ;eax = 'n'

; 比较数组
        repe cmpsd
        je @F                           ;数组相等

; 计算不匹配元素的索引号
        sub eax,ecx
        dec eax                         ;eax = 不匹配的索引

@@:     pop edi
        pop esi
        pop ebp
        ret
CompareArrays_ endp
        end
```

82

清单 2-34 中的汇编语言函数 CompareArrays_ 比较两个整数数组的元素，并返回第一个不匹配元素的索引。如果两个数组是完全相同的，那么返回的是元素的个数。返回 -1 代表错误情况。这个函数把指向数组 x 和 y 的指针分别加载到 ESI 和 EDI 寄存器。然后把元素的个数加载到 ECX 寄存器，并使用 test ecx, ecx 指令来检查它的有效性。test（Logical Compare，逻辑比较）指令会对两个操作数执行按位与（AND）操作，并根据结果设置状态标志 EFLAGS.ZF，EFLAGS.SF 和 EFLAGS.PF（EFLAGS.CF 和 EFLAGS.OF 被清零）。AND 操作的结果会被丢弃掉。编写汇编程序时，常常使用 test 指令来替代 cmp 指令，特别是测试前面的计算结果的时候，它具有指令编码更短的好处。

我们使用了 repe cmpsd（Compare String Doubleword，比较字符串双字）指令来比较两个数组。这条指令比较 ESI 和 EDI 指向的两个双字，并根据结果设置状态标志。每一次比较操作之后 ESI 和 EDI 寄存器会被自动递增（因为做的是双字比较，所以递增 4）。前缀 repe（Repeat While Equal，相等时重复）告诉处理器重复执行 cmpsw 指令，只要条件 ECX != 0 && EFLAGS.ZF == 1 为真。当双字比较完成时，程序会在数组相同（EAX 已经包含了正确的返回值）时执行条件跳转，不同时会计算出第一个不匹配元素的索引。输出 2-14 显示了执行 CompareArrays 的结果。

输出 2-14 示例程序 CompareArrays

```
Results for CompareArrays
  Test #1 - expected:  1  actual:  1
  Test #2 - expected:  9  actual:  9
  Test #3 - expected: 10  actual: 10
  Test #4 - expected: 20  actual: 20
  Test #5 - expected: 21  actual: 21
```

2.5.4 反转数组

这一节的最后一个示例程序叫ReverseArray，演示的是如何使用lods（Load String，加载字符串）指令反转（颠倒）一个数组的元素。与前面的例子不同，ReverseArray从最后一个元素到第一个元素做反向扫描，这需要修改控制标志EFLAGS.DF。清单2-36和清单2-37分别给出了ReverseArray.cpp和ReverseArray_.asm文件的源代码。

清单2-36 ReverseArray.cpp

```
#include "stdafx.h"
#include <stdlib.h>

83  extern "C" void ReverseArray_(int* y, const int* x, int n);
    int _tmain(int argc, _TCHAR* argv[])
    {
        const int n = 24;
        int x[n], y[n];

        // 初始化测试数组
        srand(31);
        for (int i = 0; i < n; i++)
            x[i] = rand() % 1000;

        ReverseArray_(y, x, n);

        printf("\nResults for ReverseArray\n");
        for (int i = 0; i < n; i++)
        {
            printf("  i: %5d  y: %5d  x: %5d\n", i, y[i], x[i]);
            if (x[i] != y[n - 1 - i])
                printf("  Compare failed!\n");
        }

        return 0;
    }
```

清单2-37 ReverseArray_.asm

```
        .model flat,c
        .code

; extern "C" void ReverseArray_(int* y, const int* x, int n);
;
; 描述：把数组x的元素以相反顺序保存到数组y
;
; 返回值：0 = 无效的'n'
;         1 = 成功

ReverseArray_ proc
        push ebp
        mov ebp,esp
        push esi
        push edi

; 假设参数，确保'n'是有效的
        xor eax,eax                     ;错误返回码
        mov edi,[ebp+8]                 ;edi = 'y'
        mov esi,[ebp+12]                ;esi = 'x'
        mov ecx,[ebp+16]                ;ecx = 'n'
        test ecx,ecx
84      jle Error                       ;若'n' <= 0则跳转
```

```
; 初始化指向 x[n - 1] 的指针，设置方向标志
        lea esi,[esi+ecx*4-4]           ;esi = &x[n - 1]
        pushfd                          ;保存当前方向标志
        std                             ;EFLAGS.DF = 1

; 反复循环直到完成数组反转
@@:     lodsd                           ;eax = *x--
        mov [edi],eax                   ;*y = eax
        add edi,4                       ;y++
        dec ecx                         ;n--
        jnz @@

        popfd                       ;恢复方向标志
        mov eax,1                   ;设置成功返回码

Error:  pop edi
        pop esi
        pop ebp
        ret
ReverseArray_ endp
        end
```

清单 2-36 中的 ReverseArray_ 函数以反向顺序把源数组的元素复制到目标数组。这个函数需要三个参数：指向目标数组的指针 y，指向源数组的指针 x 和元素个数 n。参数值分别被加载到 EDI、ESI 和 ECX 寄存器。

为了能颠倒源数组中的元素，需要找到最后一个数组元素 x[n-1] 的地址，这是通过指令 lea esi, [esi+ecx*4-4] 而实现的，这条指令计算源操作数的有效地址（也就是执行括号中指定的算术运算）。然后使用 pushfd（Push EFLAGS Register onto the Stack，将 EFLAGS 寄存器压到栈上）指令把 EFLAGS.DF 的当前状态保存到栈上，其后是 std（Set Direction Flag，设置方向标志）指令。把数组元素从 x 复制到 y 的过程是很直截了当的。先使用 lodsd（Load String Doubleword）指令从 x 加载一个元素到 EAX 并递减 ESI，然后把这个值保存到 EDI 指向的 y 的目标元素。指令 add edi, 4 把 EDI 指向 y 数组的下一个元素位置。而后再递减 ECX 寄存器，再次循环，直到完成整个反转操作。

在反转数组循环之后，用 popfd（Pop Stack into EFLAGS Register，从栈中弹出到 EFLAGS 寄存器）指令来恢复 EFLAGS.DF 的原来状态。读者看到这里可能要问一个问题：既然 Visual C++ 运行时环境总是认为 EFLAGS.DF 是清零的，那么在 ReverseArray_ 函数中为什么不用 cld（Clear Direction Flag，清除方向标志）指令来恢复 EFLAGS.DF，而是使用 pushfd/popfd 指令呢？对的，Visual C++ 运行时环境确实是假定 EFLAGS.DF 总是清零的，但是当程序执行时，它是无法强制这个策略的。因为 ReverseArray_ 函数被声明为公共函数，所以它可能被另一个违反 EFLAGS.DF 状态法则的汇编语言函数所调用。如果 ReverseArray_ 函数被包含在某个 DLL 中，那么它便可能被使用不同方向标志规范的某个函数所调用。因此，使用 pushfd/popfd 指令来保存 EFLAGS.DF 的状态是最稳妥的。输出 2-15 给出了执行 ReverseArray 程序的结果。 [85]

输出 2-15 示例程序 ReverseArray

```
Results for ReverseArray
 i:    0  y:   409  x:   139
 i:    1  y:    48  x:   240
 i:    2  y:   981  x:   971
```

```
i:    3  y:  643  x:  503
i:    4  y:  102  x:  927
i:    5  y:  114  x:  453
i:    6  y:  366  x:  547
i:    7  y:  697  x:   76
i:    8  y:   87  x:  789
i:    9  y:  466  x:  862
i:   10  y:  268  x:  268
i:   11  y:  862  x:  466
i:   12  y:  789  x:   87
i:   13  y:   76  x:  697
i:   14  y:  547  x:  366
i:   15  y:  453  x:  114
i:   16  y:  927  x:  102
i:   17  y:  503  x:  643
i:   18  y:  971  x:  981
i:   19  y:  240  x:   48
i:   20  y:  139  x:  609
```

2.6 总结

这一章介绍了很多x86汇编语言编程的内容，包括x86汇编语言函数的基础知识、调用约定、整数算术、内存寻址模式和条件码等关键主题，以及如何使用x86汇编语言来完成常见的编程任务，比如操作数组、结构体和文本串。

如果你是第一次游历汇编语言编程的世界，读到这里可能感到有点不知所措，不过请别担心。学习一门新编程语言的经验就是从编写简单的程序开始，然后逐渐学习这门语言的复杂内容，这本书的示例程序就是按这个目标来组织的，所有的汇编语言函数都比较短而且依赖很少，目的是方便大家动手练习和试验。我们常常用简单的控制台程序以避免过高的复杂度。

第1章和第2章解释了x86-32平台编程的关键要素和执行环境。接下来，你将使用这些知识来探索x86平台的其他宝藏，比如，第3章和第4章将介绍x87浮点单元。

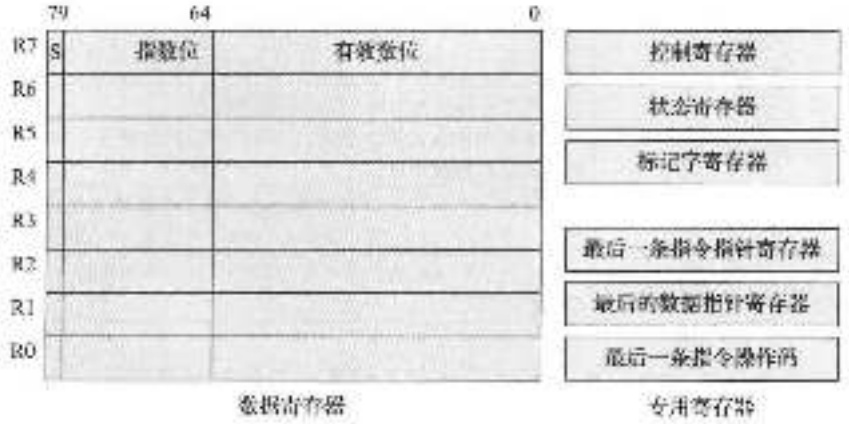
x87 浮点单元

x86 平台包含可以执行浮点算术的独立执行单元——x87 浮点单元（FPU）。其通过专用硬件实现基本的浮点运算，包括加、减、乘、除各类计算，并内建了平方根、三角函数和对数指令以进行复杂的数学运算。x87 FPU 支持多种数据类型，包括单双精度浮点数，有符号整型数和组合 BCD 码。

本章将详细探讨 x87 FPU 架构，读者可以了解到 x87 FPU 的主要组件：数据寄存器、控制寄存器以及状态寄存器。还将学习 x87 用来表示浮点数和某些特殊值的二进制编码方式。理解这些编码方案可以帮助软件工程师在大量使用浮点值的情景中，减少可能的浮点错误和提高算法的性能。本章结束部分介绍了 x87 FPU 的指令集。

3.1 x87 FPU 核心架构

总体来说，x87 FPU 包含 8 个 80 位宽的数据寄存器和一组专用寄存器。专用寄存器组包含一个控制寄存器和一个状态寄存器，程序员可以用它们来配置 x87 FPU 以及检测其当前状态。另外，专用寄存器组还包括若干辅助寄存器，主要由操作系统和浮点异常处理程序使用。图 3-1 描述了 x87 FPU 的主要部件。



注：S——符号位。

图 3-1 x87 FPU 核心架构

3.1.1 数据寄存器

x87 FPU 的八个数据寄存器是以栈结构组织的，所有计算指令都相对栈顶进行隐式或显示执行。x87 FPU 寄存器栈可以压入或弹出各种数据类型，包括有符号整型（16 位、32 位和 64 位）、浮点型（32 位、64 位和 80 位）和 80 位组合 BCD 码。不可以在 x87 FPU 数据寄存器和 x86-32 通用寄存器间直接传输数据，必须通过中间内存来执行此类操作。应当注意的是，这类数据传输和通常的 x86 数据传输不同。因为 x87 FPU 指令集不支持将操作数直

接入栈。除了极少范围的一部分常用的数值外，计算用的常数也必须通过内存操作数把数据加载到 x87 FPU 寄存器栈中。

x87 FPU 的数值格式、处理算法和异常信号处理都遵从 IEEE 的二进制浮点运算标准（IEEE 754-1985）。在内部处理时，x87 FPU 使用 80 位的扩展双精度浮点数格式。加载和保存 x87 FPU 寄存器值时，会自动进行内部格式和目标格式的转换，包括整型、浮点和 BCD 格式。

3.1.2 x87 FPU 专用寄存器

x87 FPU 包含几个专用寄存器，主要用来配置 FPU、检测状态以及协助进行异常处理。如图 3-2 所示，x87 FPU 控制寄存器允许一个任务启用或禁用各种浮点处理选项，包括异常、舍入方法精度等级。与 x86 其他大多数控制寄存器不同，修改 x87 FPU 控制寄存器不需要提升执行特权，应用程序可以根据算法的具体处理要求来配置 x87 FPU 控制寄存器。表 3-1
88 描述了 x87 FPU 控制寄存器中各个位域的含义。

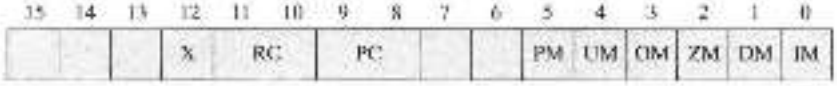


图 3-2 x87 FPU 控制寄存器

表 3-1 x87 FPU 控制寄存器位域

位	位域名称	作用描述
IM	无效操作掩码	无效操作异常掩码位：值为 1 时禁用该异常
DM	不合规格操作数掩码	不合规格操作数异常掩码位：值为 1 时禁用该异常
ZM	除零掩码	除数为 0 时异常掩码位：值为 1 时禁用该异常
OM	向上溢出掩码	向上溢出异常掩码位：值为 1 时禁用该异常
UM	向下溢出掩码	向下溢出异常掩码位：值为 1 时禁用该异常
PM	精度掩码	精度异常掩码位：值为 1 时禁用该异常
PC	精度控制位域	指定基本浮点运算的精度。有效选项包括单精度（00b）、双精度（10b）以及扩展精度（11b）
RC	舍入控制位域	指定对 x87 FPU 计算结果的舍入方式。有效选项包括就近舍入（00b）、朝 $-\infty$ 舍入（01b）、朝 $+\infty$ 舍入（10b）和朝 0 舍入或截断（11b）
X	无穷大控制位	允许处理无穷大值是为了兼容 80287 数学协处理器，现在的应用软件可以忽略此标志

在 x87 FPU 控制寄存器的异常掩码位上置 1 只禁止处理器异常产生，其状态寄存器总是记录发生的任何 x87 FPU 异常情况。应用程序不可以直接访问指定 x87 FPU 异常处理函数的内部处理器表。不过，大多数的 C 和 C++ 编译器都提供了库函数来允许应用程序指定
89 一个回调函数，当每 x87 FPU 异常发生时，这个回调函数会被调用。

x87 FPU 状态寄存器包含一个 16 位的值，让应用程序通过这些位来检查算术运算的结果，判断是否发生了异常，或者查询栈的状态信息。图 3-3 显示了 x87 FPU 状态寄存器各个位域的结构，表 3-2 介绍了每个状态寄存器位域的含义。

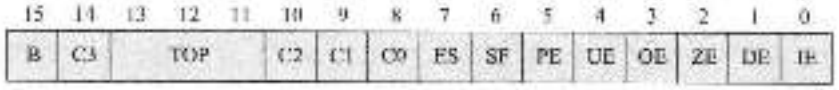


图 3-3 x87 FPU 状态寄存器

表 3-2 x87 FPU 状态寄存器位域

位	位域名称	作用描述
IE	无效操作异常	无效操作异常的状态位；指令使用了无效操作数时置为 1
DE	不合规格操作数异常	不合规格操作数异常的状态位；指令使用了非正常操作数时置为 1
ZE	除 0 异常	除 0 异常的状态位；当指令尝试除 0 时置为 1
OE	向上溢出异常	向上溢出异常的状态位；如果运算结果超出了目标操作数允许的最大值时置为 1
UE	向下溢出异常	向下溢出异常的状态位；如果运算结果小于目标操作数允许的最小值时置为 1
PE	精度异常	精度异常的状态位；如果运算结果不能用二进制精确表示目标操作数的格式时置为 1
SF	栈错误	值为 1 时表明发生了一个堆错误（无效操作异常标志也被置为 1）；条件码位 C1 表明堆错误的类型：向下溢出（C1=0）或者向上溢出（C1=1）
ES	错误摘要状态	表明至少设置了一个没有屏蔽掉的异常位
C0	条件码标志 0	x87 FPU 状态标志（详见后面的描述）
C1	条件码标志 1	x87 FPU 状态标志（详见后面的描述）
C2	条件码标志 2	x87 FPU 状态标志（详见后面的描述）
TOS	栈顶寄存器	三个位组合用来表明当前的栈顶寄存器
C3	条件码标志 3	x87 FPU 状态标志（详见后面的描述）
B	忙标志	ES 标志的副本，为了兼容 8087；现在的应用软件可以忽略此标志

当 x87 FPU 指令异常产生浮点错误时，其状态寄存器中的异常位将被置位。这些标志无法由处理器自动清除，必须使用 fclex 或者 fnclex（清除异常）指令手动复位。条件码标志显示了浮点计算和比较操作的结果，有些指令也用它们来指示错误和额外的状态信息。没有办法快直接测试 x87 FPU 状态寄存器（也被称为 x87 FPU 状态字），必须使用 fstsw 或者 fnstsw（存储 x87 FPU 状态字）指令将其复制到内存或者寄存器 AX 中。

x87 FPU 包含一个 16 位的标签字寄存器（tag word register），用来指示每个 80 位数据寄存器的内容。应用程序或者异常处理程序都可以检测标签字。常见的浮点寄存器标签状态包括“值有效”（00b），“值为 0”（01b），“特殊值”（10b）和“值为空”（11b）。其中，特殊的标签状态包括无效格式、不合规格（denormal）和无穷大，在后面的章节中会描述这些状态的含义。

另外，x87 FPU 还包含三个主要由操作系统和异常处理程序使用的寄存器：“最后一条指令指针”“最后一个数据指针”和“最后一条指令操作码”寄存器，它们可以让异常处理程序去探知触发异常指令的附加信息。“最后一条指令指针”和“最后一个数据指针”寄存器的大小取决于当前处理器的执行模式是 x86-32 还是 x86-64。“最后一条指令操作码”寄存器是 11 位，该寄存器包含了执行过的最后一条非控制类 x87 FPU 指令的低位操作码信息（一个 x87 FPU 指令操作码的高 5 位不被保存，因为这些位始终是 11011b）。

3.1.3 x87 FPU 操作数和编码

x87 FPU 指令集支持三种类型的内存操作数：有符号整型数、浮点数和组合（压缩）BCD 码。可用的有符号整型数包括字（16 位）、双字（32 位）和四字（64 位）。支持的浮点数类型包括：单精度（32 位）、双精度（64 位）和扩展双精度（80 位）。许多 C 和 C++ 编译

9 器使用单精度和双精度浮点操作数类型来分别实现 float 和 double 值。唯独组合 BCD 码是 80 位长。在第 1 章中描述了 x87 FPU 指令可以使用任何一种寻址模式，以指定在内存中的操作数。图 3-4 描述了所有有效的 x87 内存操作数类型的结构组成。

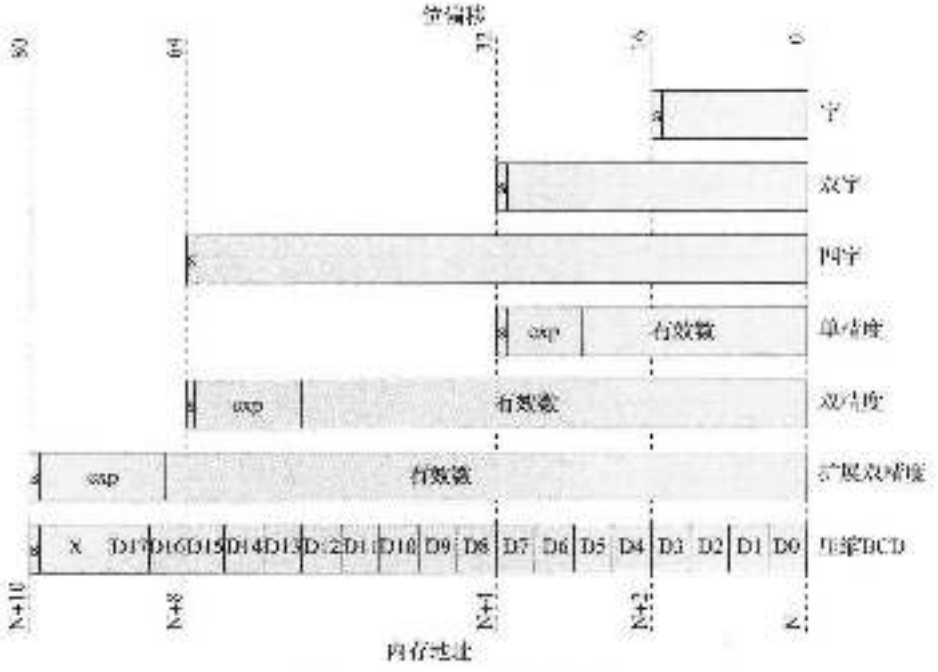


图 3-4 x87 内存操作数类型

x87 FPU 使用三个不同位域对浮点数进行编码：有效数位域、指数位域和符号位位域。有效数位域表示数字的有效数（或小数部分）。指数位域指定了在有效数中二进制小数点的实际位置，它决定了数字的表示范围。符号位指示了数字是正（s=0）还是负（s=1）。表 3-3 列出了用来编码单精度、双精度、扩展双精度浮点数的各种范围参数。

表 3-3 浮点范围参数

参数	单精度	双精度	扩展双精度
总位宽	32	64	80
有效数位宽	23	52	63
指数位宽	8	11	15
符号位宽	1	1	1
92 指数偏置	+127	+1023	+16383

图 3-5 演示了一个十进制数转换为 x87 FPU 兼容浮点数的过程。在这个例子中，数字 237.8325 从十进制数转换为单精度浮点数。整个过程中，首先将数字从基值 10 转换为基值 2，然后将得到的数转换为二进制科学型数，在符号 E 右侧的值是二进制指数。为了加快浮点比较运算，合适的编码方式是用校正指数替代真实的指数。对单精度浮点数而言，校正值（bias）是 +127。把真实的指数加上校正值变为得校正指数的二进制科学型。本例中，如图 3-5 所示，111b 加上 1111111b（+127），得到带校正指数的二进制科学型 10000110b。



例 3-5 单精度浮点数编码过程

当进行单精度或者双精度浮点数编码时，有效数的首位数字 1 是隐含的，在最终的二进制数中没有表示出来。而扩展双精度数编码时包含了首位数字 1。删除首位数字 1 形成规格化有效数后，符合 IEEE754 编码的三个位域就完成了，如表 3-4 所示。表中位域从左到右以 32 位值表示为 0x436DD000，这就是 237.8325 的单精度浮点数的最终编码形式。

表 3-4 IEEE754 标准位域

符号位	校正指数	规格化有效数
0	10000110	11011011101000000000000

93

针对一些特定条件下的计算，x87 FPU 的编码方案中还保留了小部分的位模式集合。第一组特殊值包括非规格化数。如之前的编码例子，对浮点数的标准编码中，会假定有效数的首位数字为 1。这种方式一个很大的缺点是无法准确表示接近于 0 的数。在这些情况下，x87 FPU 将使用非规格化格式，使得可以以比较低的精度编码接近于 0 的数（包括正数和负数）。非规格化很少发生，但是在使用它们的时候，x87 FPU 仍能正常地处理。在使用非规格化数可能产生问题的算法中，应用程序可以先对浮点数进行测试以确定其非规格化的状态，或者也可以配置 x87 FPU，让其产生一个向下溢出或非规格化异常，然后对此状况进行处理。

特殊值的另外一个应用是对浮点数 0 的编码。x87 FPU 支持两种不同的浮点 0 表示方法：正 0（+0.0）和负 0（-0.0）。负 0 可以在算法上或者由 x87 FPU 舍入模式产生。从计算角度来看，x87 FPU 对正 0 和负 0 的处理相同，程序员不必考虑。另外，x87 FPU 也包含了用来测试浮点数符号位的指令。

x87 FPU 编码方案中还支持正无穷和负无穷数。无穷数是在一些特定的数值算法下产生的，如向上溢出或者除 0。在本章前面讨论过，x87 FPU 可以配置为在向上溢出或者当程序试图除 0 时产生异常。

最后一种特殊值类型被称作 Not a Number（NaN）。NaN 数在浮点编码中不是一个有效的数字。x87 FPU 定义了两种类型的 NaN 数：信号（signaling）NaN（SNaN）和静默（quiet）NaN（QNaN）。SNaN 数是由软件产生的，FPU 在任何算术运算中都不会产生 SNaN。任何尝试通过指令使用 SNaN 数的行为都会触发无效操作异常，除非此异常被屏蔽了。SNaN 数

对测试异常处理程序非常有用，也可以利用它们来保护应用程序中的专有数字处理逻辑。x87 FPU 在处理某些无效计算产生的异常时，如果异常被屏蔽，会使用 QNaN 数作为缺省结果。比如，当使用 fsqrt 指令对负数进行开方时，x87 便使用一种唯一的 QNaN 编码（我们称之为未定义数）作为结果。QNaN 数也可由程序用作标志算法特殊错误或者其他不常见的数值条件。当使用 QNaN 数作为操作数时，程序将继续运行而不产生异常。

当为 x87 FPU 或者其他浮点数平台开发软件时，要记住所采用的编码方案只是实数的近似值。任何一种浮点数编码系统都无法使用有限的位数来表示无限的数，这导致了浮点舍入误差，将影响计算结果的准确性。此外，一些对整数和实数成立的数学属性在浮点数上不一定成立。比如，浮点乘法不一定符合结合律，如 (a*b)*c 可能不等于 a*(b*c)，具体还要取
94 决于 a、b 和 c 的值。开发高精度浮点算法的程序员需要注意这些问题。

3.2 x87 FPU 指令集

接下来的章节对 x87 FPU 指令集做简要概述。与在第 1 章介绍的 x86-32 指令集类似，本节的目的是使读者对 x87 FPU 指令集有一个大致的了解。本节介绍的每条 x87 FPU 指令的信息（包括有效操作数、受影响的条件码和控制字选项的效果）都参照了 AMD 和 Intel 公开发行的参考手册。这些手册和其他 x87 FPU 的文档资源都列在附录 C 中。第 4 章的示例代码将演示 x87 FPU 指令集的使用方法。

x87 FPU 指令集可以分为以下六类：

- 数据传输
- 基本运算
- 数据比较
- 超越函数
- 常量
- 控制

在接下来的指令描述中，ST(0) 表示 x87 FPU 寄存器堆栈的最顶部的值，ST(i) 表示从当前堆栈顶部开始的第 i 个寄存器。大多数的 x87 FPU 计算指令使用 ST(0) 作为隐式操作数，而 ST(i) 必须明确指定。

3.2.1 数据传输

数据传输组的指令用来把数据从 x87 FPU 寄存器栈中压入或者弹出。x87 FPU 根据操作数的数据类型是浮点、整型或压缩 BCD 值使用不同的指令助记符来执行压入（加载）和弹出（存储）操作。表 3-5 总结了数据传输指令。

表 3-5 x87 FPU 数据传输指令

助记符	描 述
fld	将浮点值压入寄存器栈，源操作数可以是 ST(i) 或内存地址
fild	从内存中读取一个有符号整型操作数，将该值转换为扩展双精度值，并将此结果加载到寄存器栈中
fbld	从内存中读取压缩 BCD 格式数，将该值转换为一个扩展双精度值，并将结果加载到堆栈
95 fst	拷贝 ST(0) 到 ST(i) 或内存位置
fstp	执行与 fst 同样的操作，并且进行弹栈操作
fist	将 ST(0) 中的值转换为一个整型数，并将结果保存到指定的内存位置

（续）

助记符	描　述
fistp	执行与 fist 同样的操作，并且进行弹栈操作
fisttp	利用截断把 ST(0) 中的值转换为整数，把结果保存到指定的内存位置，同时弹出堆栈。本指令在支持 SSE3 的处理器中才有效
fbstp	将 ST(0) 中的值转换为组合 BCD 格式，保存结果到指定内存储位置，并弹出堆栈
fxch	交换寄存器 ST(0) 和 ST(i) 的内容
fcmovcc	如果指定条件为真，则有条件地将 ST(i) 的内容复制到 ST(0)。有效的条件代码见表 3-6。通常在使用 fcmovcc 指令前会使用浮点比较指令，更多的信息参照关于浮点数比较的 3.2.3 节

表 3-6　fcmovcc 指令的条件码

条件码	描　述	测试条件
B	小于	CF==1
NB	不小于	CF==0
E	等于	ZF==1
NE	不等于	ZF==0
BE	小于或等于	CF==1 || ZF==1
NBE	不小于或等于	CF==0 && ZF==0
U	无序的	PF==1
NU	有序的	PF==0

3.2.3 节包含更多关于使用有序和无序的浮点比较的信息。

3.2.2　基本运算

基本运算组包含执行标准算术运算的指令，包括加、减、乘、除以及平方根。表 3-7 列出了这些指令。 96

表 3-7　x87 FPU 基本运算指令

助记符	描　述
fadd	源操作数和目标操作数相加。源操作数可以是内存单元或者 x87 FPU 寄存器，目标操作数必须是 x87 FPU 寄存器
faddp	ST(i) 和 ST(0) 相加，计算结果存入 ST(i)，同时弹出堆栈
fiadd	ST(0) 与指定的整型操作数相加，并把结果存入 ST(0)
fsub	从目标操作数（被减数）中减去源操作数（减数），结果存入目标操作数。源操作数可以是内存单元或者 x87 FPU 寄存器，目标操作数必须是 x87 FPU 寄存器
fsubr	从源操作数（被减数）中减去目标操作数（减数），结果存入目标操作数。源操作数可以是内存地址或者 x87 FPU 寄存器，目标操作数必须是 x87 FPU 寄存器
fsubp	从 ST(i) 中减去 ST(0)，保存差值到 ST(i)，弹出堆栈
fsubrp	从 ST(0) 中减去 ST(i)，保存差值到 ST(i)，弹出堆栈
fisub	从 ST(0) 中减去指定的整型操作数，保存差值到 ST(0)
fisubr	从指定的整型操作数中减去 ST(0)，保存差值到 ST(0)
fmul	源操作数和目标操作数相乘，乘积存入目标操作数。源操作数可以是内存地址或者 x87 FPU 寄存器，目标操作数必须是 x87 FPU 寄存器
fmulp	ST(i) 和 ST(0) 相乘，乘积存入 ST(i)，并弹出堆栈
fimul	ST(0) 与指定的整型操作数相乘，乘积存入 ST(0)
fdiv	目标操作数（被除数）除以源操作数（除数），源操作数可以是内存地址或者 x87 FPU 寄存器，目标操作数必须是 x87 FPU 寄存器

（续）

助记符	描　述
fdivr	源操作数（被除数）除以目标操作数（除数）。源操作数可以是内存地址或者 x87 FPU 寄存器，目标操作数必须是 x87 FPU 寄存器
fdivp	ST(0) 除以 ST(i)，商保存到 ST(i)，并弹出堆栈
97 fdivrp	ST(0) 除以 ST(i)，商保存到 ST(i)，并弹出堆栈
fidiv	ST(0) 除以指定的整型操作数，商保存到 ST(0)
fidivr	用指定的整型操作数除以 ST(0)，商保存到 ST(0)
fprem	计算 ST(0) 除以 ST(1)，得到的余数存入 ST(0)。这条指令常用在计算余数的循环中
fprem1	类似于 fprem 指令，不过计算余数的时候用的是 IEEE 754 标准指定的算法
fabs	计算 ST(0) 的绝对值，并将结果存入 ST(0) 中
fchs	补充 ST(0) 的符号位，并将结果保存到 ST(0)
frndint	对 ST(0) 中的值舍入到最接近的整型数，将结果存入 ST(0) 中。使用 x87 FPU 控制字中的 RC 位或来指定舍入的方式
fsqrt	计算 ST(0) 的平方根，结果存入 ST(0)
fxtract	分离 ST(0) 的指数部分和有效数部分，执行完指令后，ST(0) 中包含有效数，ST(1) 中包含指数

3.2.3 数据比较

数据比较组包含用于比较和测试浮点值的指令。如本章前面所讨论的，x87 FPU 状态字包含了一组用于指示算术和比较指令结果的条件码标志。表 3-8 列出了执行浮点比较或测试指令（例如 fcom、fucom、ficom、ftst 或 fxam 以及它们对应的带有弹栈功能的指令）后条件码的状态，表 3-9 总结了 x87 FPU 数据比较指令。

表 3-8　x87 FPU 比较指令的条件码标志

条件	C3	C2	C0
ST(0) > SRC_OP	0	0	0
ST(0) < SRC_OP	0	0	1
ST(0) = SRC_OP	1	0	0
98 无序的	1	1	1

表 3-9　x87 FPU 数据比较指令

助记符	描　述
fcom	比较 ST(0) 与 ST(i)，或者比较 ST(0) 与内存操作数，同时基于比较结果设置 x87 FPU 条件码标志
fcomp fcompp	比较 ST(0) 与 ST(i)，或者比较 ST(0) 与内存操作数，设置 x87 FPU 条件码标志，同时弹出堆栈。fcompp 指令进行两次弹栈
fucom	执行 ST(0) 和 ST(i) 的无序比较操作，根据结果设置 x87 FPU 条件码标志
fucomp fucompp	执行 ST(0) 与 ST(i) 的无序比较操作，设置 x87 FPU 条件码标志，并弹出堆栈。fucompp 弹栈两次
ficom	比较 ST(0) 与内存中的整型操作数，根据结果设置 x87 FPU 条件码标志
ficomp	比较 ST(0) 与内存中的整型操作数，设置 x87 FPU 条件码标志，同时弹出堆栈
fcomi	比较 ST(0) 与 ST(i)，同时根据结果直接设置 EFLAGS.CF、EFLAGS.PF 和 EFLAGS.ZF
fcomip	执行与 fcomi 指令同样的操作，同时弹出堆栈
fucomi	执行 ST(0) 和 ST(i) 的无序比较操作，同时根据结果直接设置 EFLAGS.CF、EFLAGS.PF 和 EFLAGS.ZF
fucomip	执行与 fucomi 指令同样的操作，同时弹出堆栈

（续）

助记符	描　述
ftst	比较 ST(0) 与 0.0，根据结果设置 x87 FPU 条件码标志
fxam	检查 ST(0) 并设置 x87 FPU 条件码标志，表明寄存器内容，可能内容包括非规范数、空操作、无穷大、NaN、正常有限数、不支持的格式和 0

没有那条条件转移指令可以直接测试 x87 的条件码标志。为了使程序按照条件码的状态跳转到指定位置，必须把这些标志转移到 x86 处理器的状态（EFLAGS）寄存器。数据比较的实现是通过指令序列 fstsw ax（AX 中存储的 x87 FPU 状态字）和 sahf（存储 AH 到标志寄存器）来实现的。它们分别将条件码标志 C0、C2 和 C3 复制到 EFLAGS.CF、EFLAGS.PF 和 EFLAGS.ZF。基于 P6 或更新微架构的处理器（包括自 1997 年销售的所有处理器）也可以使用 fcomi 和 fucomi 指令直接设置 EFLAGS.CF、EFLAGS.PF 和 EFLAGS.ZF。EFLAGS 状态位设置后，条件跳转指令可以使用表 3-6 中描述的条件码来执行。 99

与整数比较不同，浮点比较存在四种可能且相互排斥的结果：小于、等于、大于、无序。无序的浮点比较发生时，至少存在一个操作数是 NaN 或无效的浮点值。在一个有序的比较中，两个操作数都是有效的浮点数字。使用 x87 FPU 有序比较指令时，如果操作数是 NaN 或无效值，将导致处理器产生一个无效操作异常。如果无效操作异常被屏蔽，x87 FPU 条件码标志或 EFLAGS 状态位会进行相应的设置。使用 x87 FPU 无序比较指令时，如果操作数中存在一个 SNaN 或无效值，会触发 x87 FPU 无效操作异常。如果无效操作异常被屏蔽，会设置 x87 FPU 条件码标志或 EFLAGS 状态位。另外，执行无序比较指令时，如果使用的是 QNaN 操作数，会使得条件码标志或者 EFLAGS 状态位被置位，但是不会触发异常。

3.2.4 超越函数

超越（Transcendental）函数组的指令包含对浮点类操作数执行三角函数、对数和指数运算的各种指令。表 3-10 列出了这一组的指令。

表 3-10　x87 FPU 超越函数指令

助记符	描　述
fsin	计算 ST(0) 的正弦值并将结果存入 ST(0) 中
fcos	计算 ST(0) 的余弦值并将结果存入 ST(0) 中
fsincos	计算 ST(0) 的正弦和余弦值，执行完指令后，ST(0) 和 ST(1) 中分别包含原操作数的正弦和余弦值
fptan	计算 ST(0) 的正切值并将结果存入 ST(0) 中，同时将常数 1.0 压入堆栈
fpatan	计算 ST(1) 除以 ST(0) 的反正切值，同时将结果存入 ST(0)
f2xm1	计算 2^(ST(0)-1)，同时把结果存入 ST(0)，源操作数范围必须在 -1.0 至 +1.0 之间
fyl2x	计算 ST(1)*log2(ST(0))，结果存入 ST(1)，并弹出堆栈
fyl2xp1	计算 ST(1) * log2(ST(0)+1.0)，结果存入 ST(1)，并弹出堆栈
fscale	截断（向 0 舍入）ST(1) 的值，并将此值与 ST(0) 的指数部分相加。这一指令用来对 2 的整数幂做快速乘除计算

100

3.2.5 常量

常量组包含用于加载常用的浮点常量值的指令。表 3-11 列出了常量组的指令。

表 3-11 x87 FPU 常量组指令

助记符	描 述
fld1	把常数 +1.0 压入 x87 FPU 寄存器栈
fldz	把常数 +0.0 压入 x87 FPU 寄存器栈
fldpi	把常数 π 压入 x87 FPU 寄存器栈
fld2e	把常数值 log2(e) 压入 x87 FPU 寄存器栈
fldln2	把常数值 ln(2) 压入 x87 FPU 寄存器栈
fld2t	把常数 log2(10) 压入 x87 FPU 寄存器栈
fldlg2	把常数 log10(2) 压入 x87 FPU 寄存器栈

3.2.6 控制

控制组包含用于管理 x87 FPU 控制寄存器和状态寄存器的指令，还包含便于管理 x87 FPU 的执行环境和运行状态的指令。表 3-12 中描述了这些控制组指令。以 fn 为前缀的指令在执行前会忽略所有尚未处理的未屏蔽浮点异常，标准前缀的指令在执行前会先处理任何尚未处理的未屏蔽浮点异常。

表 3-12 x87 FPU 控制指令

助记符	描 述
finit fninit	初始化 x87 FPU 至缺省状态
fincstp	通过对 x87 FPU 状态字中的 TOS 域加 1，来改变当前的堆栈指针位置。x87 FPU 数据寄存器和标记字的内容不被改变。也就是说，该指令不等同于出栈。此指令可用于手动管理 x87 FPU 寄存器栈
fdecstp	通过对 x87 FPU 状态字中的 TOS 域减 1，来改变当前的堆栈指针位置。x87 FPU 数据寄存器和标记字的内容不被改变。也就是说，该指令不等同于入栈。此指令可用于手动管理 x87 FPU 寄存器栈
ffree	通过设置相应的标记字状态为空，释放 x87 FPU 浮点寄存器
fldcw	从指定的内存位置加载 x87 FPU 控制字
fstcw fnstcw	把 x87 FPU 控制字保存到指定的内存位置
fstsw fnstsw	把 x87 FPU 状态字保存到 AX 寄存器或者内存位置
fclex fnclex	清除 x87 FPU 状态字位：PE、UE、OE、ZE、DE、IE、ES、SF 和 B。执行完此指令后，条件码标志 C0、C1、C2 和 C3 处于未定义状态
fstenv fnstenv	把当前 x87 FPU 执行环境保存到内存，包括控制字、状态字、标记字、x87 FPU 数据指针、x87 FPU 指令指针和 x87 FPU 最后一条指令操作码
fldenv	从内存中加载 x87 FPU 执行环境
fsave fnsave	保存当前 x87 FPU 运行状态，包括所有数据寄存器的内容和以下项：控制字、状态字、标记字、x87 FPU 数据指针、x87 FPU 指令指针和 x87 FPU 最后一条指令操作码
frstor	从内存中加载 x87 FPU 运行状态

3.3 总结

本章全面介绍了 x87 FPU 的核心架构，包括其数据类型、栈形式的寄存器集合、控制寄存器和状态寄存器。你还在这一章里学习了用来表示浮点数的二进制编码方法。如果你是第一次接触浮点架构，那么可能会感到比较深奥。在第 4 章中，我们会通过很多示例程序来演示如何使用 x87 FPU 指令集进行浮点计算，这会帮助你消除脑海中的困惑。

x87 FPU 编程

本章将更详细地介绍 x87 FPU 的架构和它的指令集。我们将通过 x86 汇编语言函数来演示 x87 FPU 编程的基本要领和高级技巧。首先，我们将从几个示例程序开始 x87 FPU 的探索之路，这几个例子演示了如何对浮点数执行基本的运算和比较操作。然后，我们将学习如何对浮点数组进行计算。本章的最后一部分示例程序将演示 x87 FPU 的超越指令，并顺便介绍如何有效使用 x87 FPU 的寄存器栈。我们假定读者在阅读本章内容时已经比较熟悉第 1、2、3 章中的材料。

注意　使用浮点运算开发软件需要多加一些小心。本章示例程序的主要目的是介绍 x87 FPU 的架构和指令集，没有涉及一些重要的浮点问题，比如舍入误差、数值一致性和病态函数。软件开发者在实际项目中设计和实现使用浮点运算的算法时，就必须要考虑这些问题。如果你有兴趣了解更多关于浮点运算的潜在陷阱，请查阅附录 C 中所列的参考资料。

4.1　x87 FPU 编程基础

本节将通过几个示例程序来演示 x87 FPU 编程的基本要领。第一个示例程序演示如何使用 x87 FPU 进行简单运算，同时展示如何声明和引用在内存中的整数和浮点常量。第二个示例程序说明如何用浮点数执行比较操作。你还将学习如何根据比较操作的结果进行条件跳转。 103

4.1.1　简单计算

我们要分析的第一个示例程序名叫 TemperatureConversions（温度转换）。这个示例程序包含了几个通过 x87 FPU 进行温度值转换的函数：华氏度转换成摄氏度和摄氏度转换成华氏度。尽管很简单，但这些温度转换函数展示了一些关键的 x87 FPU 编程概念，包括使用 x87 FPU 寄存器栈和处理内存中的浮点常量。它还演示了如何从汇编语言函数中返回一个浮点数值给调用者。清单 4-1 和清单 4-2 分别列出了这个程序的 C++ 和汇编语言源代码。

清单 4-1　TemperatureConversions.cpp

```
#include "stdafx.h"

extern "C" double FtoC (double deg f);
extern "C" double CtoF (double deg c);

int _tmain(int argc, _TCHAR* argv[])
{
    double deg_fvals[] = { 459.67, -40.0, 0.0, 32.0, 72.0, 98.6, 212.0};
    int nf = sizeof(deg_fvals) / sizeof(double);

    for (int i = 0; i < nf; i++)
    {
        double deg_c = FtoC (deg_fvals[i]);
        printf("i: %d  f: %10.4lf c: %10.4lf\n", i, deg_fvals[i], deg_c);
    }

    printf("\n");
```

```
    double deg_cvals[] = {-273.15, -40.0, -17.77, 0.0, 25.0, 37.0, 100.0};
    int nc = sizeof(deg_cvals) / sizeof(double);

    for (int i = 0; i < nc; i++)
    {
        double deg_f = CtoF_(deg_cvals[i]);
        printf("i: %d  c: %10.4lf f: %10.4lf\n", i, deg_cvals[i], deg_f);
    }

    return 0;
}
```

[104]

清单 4-2 TemperatureConversions_.asm

```
        .model flat,c
        .const

r8_5fFtoC   real8 0.5555555556          ; 5 / 9
r8_5fCtoF   real8 1.8                   ; 9 / 5
i4_32       dword 32

; extern "C" double FtoC_(double f)
;
; 描述：把华氏度转换为摄氏度温度
;
; 返回：以摄氏度表示的温度值

        .code
FtoC_   proc
        push ebp
        mov ebp,esp

        fld [r8_5fFtoC]                 ;加载 5/9
        fld real8 ptr [ebp+8]           ;加载 'f'
        fild [i4_32]                    ;加载常数 32
        fsubp                           ;ST(0) = f - 32
        fmulp                           ;ST(0) = (f - 32) * 5/9

        pop ebp
        ret
FtoC_   endp

; extern "C" double CtoF (double c)
;
; 描述：把摄氏度转换为华氏度温度
;
; 返回：以华氏度表示的温度值

CtoF_   proc
        push ebp
        mov ebp,esp

        fld real8 ptr [ebp+8]           ;加载 'c'
        fmul [r8_5fCtoF]                ;ST(0) = c * 9/5
        fiadd [i4_32]                   ;ST(0) = c * 9/5 + 32

        pop ebp
        ret
CtoF_   endp
        end
```

[105]

下面是摄氏度值转换到华氏度的公式及逆变换公式：

$$C=(F-32)\times 5/9 \quad F=C\times 9/5+32$$

清单 4-2 的开头是一个 .const 指示符，它定义了一个包含常量值的内存块的起始位置。不同于 x86 的通用指令集，x87 FPU 指令集不支持使用常数作为立即操作数。除去少数被常数加载指令支持的数值外，常量值必须从内存中加载。.const 段包括温度转换使用的两个比例因子，real8 指示符分配和初始化一个双精度浮点值。本节还声明了一个 dword（32 位）的整型数值 32。关于 .const 段还有一点值得注意，其中的数据排列是精心安排的，以保证每个常量都是内存对齐的。

在函数 FtoC_ 中，紧随其函数序言之后的第一条指令 fld [r8_SfFtoC]（加载浮点数值）把常数 5/9 加载（或者说压入）到 x87 FPU 寄存器栈。下一条指令 fld real8 ptr [ebp+8] 把华氏温度的参数值加载到 x87 FPU 寄存器栈。real8 ptr 操作符通知汇编器把内存中的操作数当作双精度浮点数处理（在这里也可以用操作符 qword ptr，但是用 real8 ptr 可以强调加载的是双精度浮点数）。

指令 fild [i4_32]（加载整型数）把双字整型数 32 从内存加载到 x87 FPU 栈。从内存中读取操作数的时候，x87 FPU 自动把它从有符号的双字整型数转换为内部格式，即扩展双精度浮点数。关于这种转换过程需要时间，比较重要的是使用 real8 代替 dword 定义常量 32。但在此函数中，这样做的目的是为了举例说明 fild 指令的用法。执行完 fild 指令后，x87 FPU 寄存器栈上包含了三个数值：常数 32.0、华氏温度参数和常数 5/9，如图 4-1 所示。



图 4-1 指令 fild 执行后的 x87 FPU 寄存器栈的内容

fsubp（减法）指令从 ST(1) 中减去 ST(0)，把差值保存到 ST(1)，并弹出 x87 FPU 堆栈。执行完这条指令后，ST(0) 包含值 F–32，ST(1) 中包含值 5/9。指令 fmulp 把 ST(1) 和 ST(0) 相乘，并把乘积保存到 ST(1)，同时弹出寄存器栈。执行完 fmulp（乘法）指令后，ST(0) 中包含转换成摄氏度的温度值，并且这是 x87 FPU 寄存器栈中的唯一项了。

Visual C++ 的 32 位程序调用约定规定函数必须使用 ST(0) 返回浮点值给调用者，所有其他的 x87 FPU 寄存器必须是空的；如果函数不需要返回一个浮点值，则整个 x87 FPU 寄存器栈必须是空的。函数如果修改了 x87 FPU 控制寄存器的标志，则必须在返回前将这些标志恢复成原来状态。在本例中，x87 FPU 寄存器栈已经包含了需要的返回值，因此在函数结束前不需要使用其他指令来收尾。

把温度值从摄氏度转换为华氏度的函数 CtoF_ 的过程类似于 FtoC_。两个函数的最大区别在于，前者执行算术运算的时候使用内存操作数，这使得所需的指令更少。函数 CtoF_ 的第一步操作是把摄氏度参数加载到 x87 FPU 栈中。然后，使用指令 fmul [r8_SfCtoF] 将 ST(0) 中的温度值和 9/5（或者 1.8）相乘，乘积保存到 ST(0)。最后一条指令 fiadd [i4_32]（加法）让 32 和 ST(0) 相加，计算出最终的华氏温度值。

这个例子的 C++ 文件在清单 4-1 中列出来了，它包含一些用来使用函数 FtoC_ 和 CtoF_ 的测试用例。TemperatureConversions 的输出结果见输出 4-1。最后要说明的是，这个示例程序中没有对理论上不可能的温度值进行有效性检查。例如，–1000 华氏度的温度可以作为函数 FtoC_ 的参数值，函数会忽略现实中的物理限制并执行计算。

输出 4-1 示例程序 TemperatureConversions

```
i: 0  f:  -459.6700 c:   273.1500
i: 1  f:   -40.0000 c:   -40.0000
i: 2  f:     0.0000 c:   -17.7778
i: 3  f:    32.0000 c:     0.0000
i: 4  f:    72.0000 c:    22.2222
i: 5  f:    98.6000 c:    37.0000
i: 6  f:   212.0000 c:   100.0000

i: 0  c:   -273.1500 f:  -459.6700
i: 1  c:   -40.0000 f:   -40.0000
i: 2  c:   -17.7700 f:     0.0140
i: 3  c:     0.0000 f:    32.0000
i: 4  c:    25.0000 f:    77.0000
i: 5  c:    37.0000 f:    98.6000
i: 6  c:   100.0000 f:   212.0000
```

107

4.1.2 浮点比较

下一个示例程序是 CalcSphereAreaVolume。这个程序演示了如何比较两个浮点数，同时描述了若干 x87 FPU 常数加载指令的用法。CalcSphereAreaVolume 的 C++ 和汇编语言的源代码分别见清单 4-3 和清单 4-4。

清单 4-3 CalcSphereAreaVolume.cpp

```
#include "stdafx.h"

extern "C" bool CalcSphereAreaVolume (double r, double* sa, double* v);

int _tmain(int argc, _TCHAR* argv[])
{
    double r[] = { -1.0, 0.0, 1.0, 2.0, 3.0, 5.0, 10.0, 20.0 };
    int num_r = sizeof(r) / sizeof(double);

    for (int i = 0; i < num_r; i++)
    {
        double sa = -1;
        double v = -1;
        bool rc = CalcSphereAreaVolume_(r[i], &sa, &v);

        printf("rc: %d  r: %8.2lf  sa: %10.4lf  v: %10.4lf\n", rc, r[i], sa,
⇒ v);
    }

    return 0;
}
```

清单 4-4 CalcSphereAreaVolume_.asm

```
        .model flat,c
        .const
r8_4p0  real8 4.0
r8_3p0  real8 3.0

; extern "C" bool CalcSphereAreaVolume_(double r, double* sa, double* v);
;
; 描述：计算球体表面积和体积
;
; 返回：0 = 无效半径值
;       1 = 有效半径值
```

108

```
        .code
CalcSphereAreaVolume_ proc
        push ebp
        mov ebp,esp

; 确保半径值是有效的
        xor eax,eax                     ;设置错误返回码
        fld real8 ptr [ebp+8]           ;ST(0) = r
        fldz                            ;ST(0) = 0.0, ST(1) = r
        fcomip st(0),st(1)              ;比较 0.0 和 r
        fstp st(0)                      ;从堆栈中移除 r
        jp Done                         ;如果是无序操作数，则跳转
        ja Done                         ;如果 r<0.0，则跳转

; 计算球体表面积
        fld real8 ptr [ebp+8]           ;ST(0) = r
        fld st(0)                       ;ST(0) = r, ST(1) = r
        fmul st(0),st(0)                ;ST(0) = r * r, ST(1) = r
        fldpi                           ;ST(0) = pi
        fmul [r8_4p0]                   ;ST(0) = 4 * pi
        fmulp                           ;ST(0) = 4 * pi * r * r

        mov edx,[ebp+16]
        fst real8 ptr [edx]             ;保存球体表面积

; 计算球体体积
        fmulp                           ;ST(0) = pi * 4 * r * r * r
        fdiv [r8_3p0]                   ;ST(0) = pi * 4 * r * r * r / 3

        mov edx,[ebp+20]
        fstp real8 ptr [edx]            ;保存体积
        mov eax,1                       ;设置成功返回码

Done:   pop ebp
        ret
CalcSphereAreaVolume_ endp
        end
```

球体的表面积和体积可以用下面的公式来计算：

$$sa=4\pi r^2 \quad v=4\pi r^3/3=(4\pi r^2)r/3$$

在 CalcSphereAreaVolume （见清单 4-4）的函数序言之后，进行了球体半径的有效性检测。首先，使用指令 fld real8 ptr [ebp+8] 把参数值 r 加载（压入）到 x87 FPU 寄存器栈中。然后，指令 fldz（加载常数 0.0）把浮点常数值 0.0 加载到寄存器栈中。指令 fcomip st(0), st(1)（比较浮点值并且设置 EFLAGS）比较 ST(0) 和 ST(1)（或者说是比较 0.0 和 r），并且根据结果设置状态标志，同时弹出 x87 FPU 寄存器栈。紧接着的指令 fstp st(0)（存储浮点值并且弹出寄存器栈）从 x87 FPU 寄存器栈中移出 r 值，同时使得栈变空。在测试状态标志前清空 x87 FPU 寄存器栈，这是为了遵守 Visual C++ 的调用约定，r 值应该变成无效。在清除 x87 FPU 寄存器栈后，有两个条件跳转指令 jp Done 和 ja Done，分别执行的是如果 r 是 NaN（或无效）或者小于 0.0 的跳转。

比较操作指令 fcomip 的执行细节值得我们仔细体会。这个指令从 ST(0) 减去 ST(1) 并设置状态标志，如表 4-1 所示（差值被丢弃了）。x87 FPU 的其他比较指令 fcomi、fucomi 和 fucomip 也通过同样的状态标志来反映它们的比较结果。fcomip 及这些类似的比较指令通过设置标志 EFLAGS.ZF、EFLAGS.PF 和 EFLAGS.CF，让函数可以通过条件跳转指令执行得

109

点相关的跳转决定，如表 4-2 所示。

表 4-1 f(u)comi(p) 指令设置的状态标志

条件	EFLAGS.ZF	EFLAGS.PF	EFLAGS.CF
ST(0) > ST(i)	0	0	0
ST(0) = ST(i)	1	0	0
ST(0) < ST(i)	0	0	1
无序的	1	1	1

表 4-2 执行 f(u)comi(p) 指令后的条件跳转

相关的操作符	条件跳转	EFLAGS 测试条件
ST(0) < ST(i)	jb	CF == 1
ST(0) <= ST(i)	jbe	CF == 1 || ZF == 1
ST(0) == ST(i)	jz	ZF == 1
ST(0) != ST(i)	jnz	ZF == 0
ST(0) > ST(i)	ja	CF == 0 && ZF == 0
ST(0) >= ST(i)	jae	CF == 0

应当注意的是表 4-1 中描述的状态寄存器的各种状态只有在 x87 FPU 无效操作异常（x87 FPU 控制寄存器中的 IM 位）被屏蔽（对 Visual C++ 而言的缺省状态）时才能设置。如果无效操作异常没有被屏蔽，并且有一个操作数是 NaN 类型或者是无效的，fcomi(p) 指令会触发一个异常。而当操作数是 SNaN 或无效时，fucomi(p) 指令也会触发异常。如果操作数是 QNaN，处理器将会设置状态标志来指示一个无序状态，但不产生异常。关于 x87 FPU 如何使用 NaN 数以及有序和无序的比较操作，在第 3 章中有详细的阐述。

110

如果 r 的值是有效的，函数使用指令 fld real8 ptr [ebp+8] 把 r 压入 x87 FPU 寄存器栈。接下来的指令 fld st(0) 拷贝栈顶的元素，同时将其压入 x87 FPU 寄存器栈。fmul st(0), st(0) 指令计算半径的平方值，并将结果存入 ST(0)；fldpi 指令把常数 π 压入 x87 FPU 寄存器栈。在执行完 fldpi 指令后，x87 FPU 寄存器栈包含三个项：ST(0) 中的常数 π、ST(1) 中的 r*r 以及 ST(2) 中的 r，如图 4-2 左部所示。最后使用两个乘法指令计算球体表面积：fmul [r8_4p0] 和 fmulp。计算出的结果由调用者使用指令 fst 存入指定的内存地址。此时 x87 FPU 寄存器栈中还包含两个值：ST(0) 中包含计算出的表面积和 ST(1) 中的半径，如图 4-2 右部所示。

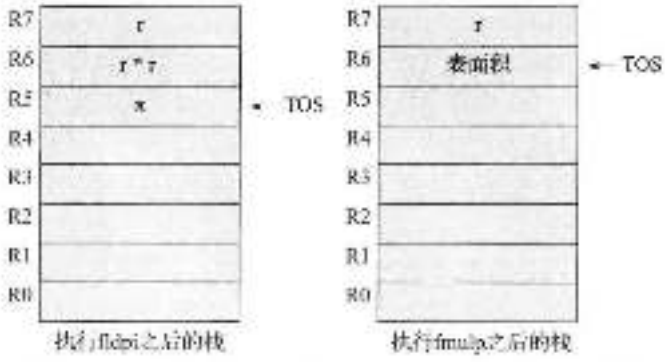


图 4-2 执行指令 fldpi 和 fmulp 之后 x87 寄存器栈的内容

这个函数利用 fmulp 指令和 fdiv[r8_3p0]（除法）指令，用球半径乘以表面积的乘积除以 3.0 得出球体体积。最终球体的体积使用 fstp real8 ptr[edx] 指令保存到调用者指定的内存位置。执行该指令后，x87 FPU 寄存器栈为空。输出 4-2 显示了示例程序 CalcSphereAreaVolume 的执行结果。

输出 4-2 示例程序 CalcSphereAreaVolume

```
rc: 0  r:    -1.00  sa:     1.0000  v:     1.0000
rc: 1  r:     0.00  sa:     0.0000  v:     0.0000
rc: 1  r:     1.00  sa:    12.5664  v:     4.1888
rc: 1  r:     2.00  sa:    50.2655  v:    33.5103
rc: 1  r:     3.00  sa:   113.0973  v:   113.0973
rc: 1  r:     5.00  sa:   314.1593  v:   523.5988
rc: 1  r:    10.00  sa:  1256.6371  v:  4188.7902
rc: 1  r:    20.00  sa:  5026.5482  v: 33510.3216
```

4.2 x87 FPU 高级编程

在上一节中，我们学习了如何使用 x87 FPU 进行一些基本的浮点运算。本章的剩余部分将重点介绍 x87 FPU 的高级编程技巧。我们以分析两个处理浮点数组的示例程序开始。然后是一个描述若干 x87 FPU 超越指令的例子，最后的示例程序重点演示了 x87 FPU 寄存器栈的使用。

4.2.1 浮点数组

接下来是两个使用 x87 FPU 进行浮点数组处理的示例程序。这两个示例程序还演示其他一些 x87 FPU 指令，包括平方根和浮点数条件传送。

第一个示例程序名为 CalcMeanStdev，此程序计算双精度浮点数组中样本均值和多个值的标准偏差。清单 4-5 和清单 4-6 分别包含了 C++ 和汇编语言程序文件的源代码。

清单 4-5 CalcMeanStdev.cpp

```
#include "stdafx.h"
#include <math.h>

extern "C" bool CalcMeanStdev_(const double* a, int n, double* mean, double*
stdev);

bool CalcMeanStdevCpp(const double* a, int n, double* mean, double* stdev)
{
        if (n <= 1)
            return false;

        double sum = 0.0;
        for (int i = 0; i < n; i++)
            sum += a[i];
        *mean = sum / n;

        sum = 0.0;
        for (int i = 0; i < n; i++)
        {
                double temp = a[i] - *mean;
                sum += temp * temp;
        }
```

```
        *stdev = sqrt(sum / (n - 1));
        return true;
}

int _tmain(int argc, _TCHAR* argv[])
{
    double a[] = { 10, 2, 33, 15, 43, 24, 75, 47, 18, 97};
    const int n = sizeof(a) / sizeof(double);
    double mean1, stdev1;
    double mean2, stdev2;

    CalcMeanStdevCpp(a, n, &mean1, &stdev1);
    CalcMeanStdev_(a, n, &mean2, &stdev2);

    for (int i = 0; i < n; i++)
        printf("a[%d] = %g\n", i, a[i]);

    printf("\n");
    printf("mean1: %g stdev1: %g\n", mean1, stdev1);
    printf("mean2: %g stdev2: %g\n", mean2, stdev2);
}
```

清单 4-6 CalcMeanStdev .asm

```
        .model flat,c
        .code

; extern "C" bool CalcMeanStdev(const double* a, int n, double* mean,↩
double* stdev);
;
; 描述：计算数组元素的均值和标准偏差
;
; 返回：0 = 'n' 无效
;       1 = 'n' 有效

CalcMeanStdev_  proc
        push ebp
        mov ebp,esp
        sub esp,4

; 确保 'n' 有效
        xor eax,eax
        mov ecx,[ebp+12]
        cmp ecx,1
        jle Done                        ;如果 n<=1，就转
        dec ecx
        mov [ebp-4],ecx                 ;保存 n-1，以备后用
        inc ecx

; 计算样本均值
        mov edx,[ebp+8]                 ;edx = 'a'
        fldz                            ;和变量 = 0.0

@@:     fadd real8 ptr [edx]            ;和变量 += *a
        add edx,8                       ;a++
        dec ecx
        jnz @@
        fidiv dword ptr [ebp+12]        ;均值 = 和/n

; 计算样本标准差
        mov edx,[ebp+8]                 ;edx = 'a'
```

```
        mov ecx,[ebp+12]                ;n
        fldz                            ;sum = 0.0, ST(1) = mean

@@:     fld real8 ptr [edx]             ;ST(0) = *a,
        fsub st(0),st(2)                ;ST(0) = *a - mean(mean 为样本均值)
        fmul st(0),st(0)                ;ST(0) = (*a - mean) ^ 2(mean 为样本均值)
        faddp                           ;更新和变量
        add edx,8
        dec ecx
        jnz @@
        fidiv dword ptr [ebp-4]         ;var = sum / (n - 1)(sum 为和变量)
        fsqrt                           ;最终的标准偏差

; 保存结果
        mov eax,[ebp+20]
        fstp real8 ptr [eax]            ;保存标准偏差
        mov eax,[ebp+16]
        fstp real8 ptr [eax]            ;保存样本均值
        mov eax,1                       ;设置计算成功的返回码

Done:   mov esp,ebp
        pop ebp
        ret
CalcMeanStdev_ endp
        end
```

114

下面显示例程序 CalcMeanStdev 用来计算样本均值和样本标准偏差的公式：

$$\bar{x}=\frac{1}{n}\sum_i x_i \qquad s=\sqrt{\frac{1}{n-1}\sum_i (x_i-\bar{x})^2}$$

注意 如果没有明确指定，本书中任何求和运算的范围都是 0 到 n-1。

紧跟在函数序言之后，函数 CalcMeanStdev_ 执行数组元素个数 n 的有效性验证。为了计算样本的标准偏差，数组中的元素个数必须大于或等于 2。验证完成之后，值 n-1 被计算出来并保存到堆栈上的局部变量中。至此，n 已经加载到寄存器 ECX 中。在减法运算中，整型数计算比浮点计算要快。把 n-1 存储到内存中也是因为 x86 不支持从通用寄存器到 x87 FPU 寄存器的数据传输。

样本均值的计算很简单，只需要 7 条指令。进入求和循环前，函数 CalcMeanStdev_ 使用 mov edx, [ebp+8] 指令初始化 EDX，使之指向数组 a。函数也使用了 fldz 指令使得 ST(0) 用作求和变量。在求和循环中，指令 fadd real8 ptr [edx] 把当前数组元素加到 ST(0) 中。因为在数组中有双精度浮点数，因此使用了 real8 ptr 操作符。在把当前数组元素加至 ST(0) 之后，指令 add edx, 8 更新寄存器 EDX，使其指向数组的下一个元素。重复求和循环，直到所有数组元素求和完成。然后使用 fidiv dword ptr [ebp+12] 指令计算样本均值，最终样本均值替代 ST(0) 中的数组元素之和。

样本标准偏差也使用求和循环来计算。进入循环前，寄存器 EDX 和 ECX 被重新初始化为数组的指针和循环计数器，fldz 指令把和初始化为 0.0。fldz 指令完成后，ST(0) 包含 0.0，ST1 包含样本均值。在求和循环中，每个数组元素都使用 fld real8 ptr [edx] 指令加载到 x87 FPU 寄存器栈中。fsub st(0), st(2) 指令从当前数组元素中减去之前计算的均值，并将差值存入到 ST(0) 中。然后再使用 fmul st(0), st(0) 指令对差值求平方，并使用 faddp 指令对方差求和。求和循环重复执行，直到数组中每个元素都被处理。

在求和循环完成后，x87 FPU 寄存器栈中包含两个值：ST(0) 中包含了总和，ST(1) 中

包含了样本均值。这个函数使用 fidiv dword ptr [ebp-4] 指令计算样本方差。回想一下，内
115 存位置 [ebp-4] 包含值 n-1，是在之前验证 n 的过程中产生的。继续样本方差的计算，函数使用 fsqrt（开方）指令计算最终的样本标准偏差，此指令将 ST(0) 中的值开方后存入 ST(0) 中。x87 FPU 寄存器栈中现在包含两个值：ST(0) 中的样本标准偏差和 ST(1) 中的样本均值。调用者使用 fstp 指令将这些值从 x87 FPU 堆栈移存至指定的内存位置。输出 4-3 显示了示例程序 CalcMeanStdev 的结果。

输出 4-3 示例程序 CalcMeanStdev

```
a[0] = 10
a[1] = 2
a[2] = 33
a[3] = 15
a[4] = 41
a[5] = 24
a[6] = 75
a[7] = 37
a[8] = 18
a[9] = 97

mean1: 35.2 stdev1: 29.8358
mean2: 35.2 stdev2: 29.8358
```

本节的第二个示例程序称为 CalcMinMax，用来寻找单精度浮点数组的最大值和最小值。C++ 和汇编语言文件分别如清单 4-7 和清单 4-8 所示。

清单 4-7 CalcMinMax.cpp

```
#include "stdafx.h"
#include <float.h>

extern "C" bool CalcMinMax_(const float* a, int n, float* min, float* max);

bool CalcMinMaxCpp(const float* a, int n, float* min, float* max)
{
    if (n <= 0)
        return false;

    float min_a = FLT_MAX;
    float max_a = -FLT_MAX;

    for (int i = 0; i < n; i++)
    {
        if (a[i] < min_a)
116         min_a = a[i];
        if (a[i] > max_a)
            max_a = a[i];
    }

    *min = min_a;
    *max = max_a;
    return true;
}

int _tmain(int argc, _TCHAR* argv[])
{
    float a[] = { 20, -12, 42, 97, 14, -26, 57, 74, -18, 63, 34,  9};
    const int n = sizeof(a) / sizeof(float);
```

```
    float min1, max1;
    float min2, max2;

    CalcMinMaxCpp(a, n, &min1, &max1);
    CalcMinMax_(a, n, &min2, &max2);

    for (int i = 0; i < n; i++)
        printf("a[%2d] = %8.2f\n", i, a[i]);

    printf("\n");
    printf("min1: %8.2f  max1: %8.2f\n", min1, max1);
    printf("min2: %8.2f  max2: %8.2f\n", min2, max2);
}
```

清单 4-8 CalcMinMax_.asm

```
        .model flat,c
        .const
r4_MinFloat dword 0ff7fffffh            ;最小浮点数
r4_MaxFloat dword  7f7fffffh            ;最大浮点数
        .code

; extern "C" bool CalcMinMax_(const float* a, int n, float* min, float*=
max);
;
; 描述：计算单精度浮点数组中的最大值和最小值
;
; 返回：0 - 'n' 无效
;       1 - 'n' 有效

CalcMinMax_ proc
        push ebp
        mov ebp,esp                                                    117

; 载入参数值并确保 'n' 有效
        xor eax,eax                     ;设置返回错误码
        mov edx,[ebp+8]                 ;edx = 'a'
        mov ecx,[ebp+12]                ;ecx = 'n'
        test ecx,ecx
        jle Done                        ;如果 'n' <= 0 则跳转

        fld [r4_MinFloat]               ;初始化 max_a 值
        fld [r4_MaxFloat]               ;初始化 min_a 值

; 寻找输入数组中的最大值和最小值
@@:     fld real4 ptr [edx]             ;加载 *a
        fld st(0)                       ;复制栈顶的 *a

        fcomi st(0),st(2)               ;把 *a 和 min_a 比较
        fcmovnb st(0),st(2)             ;保证 ST(0) 中包含最小值
        fstp st(2)                      ;保存新的最小值

        fcomi st(0),st(2)               ;比较 *a 和 max_a
        fcmovb st(0),st(2)              ;确保 ST(0) 包含最大值
        fstp st(2)                      ;保存新的最大值

        add edx,4                       ;指向 a[i] 的下一元素
        dec ecx
        jnz @@                          ;重复循环直到结束

; 保存结果
        mov eax,[ebp+16]
```

```
        fstp real4 ptr [eax]            ;保存最终的最小值
        mov eax,[ebp+20]
        fstp real4 ptr [eax]            ;保存最终的最大值
        mov eax,1                       ;设置成功的返回码

Done:   pop ebp
        ret
CalcMinMax_ endp
        end
```

在进行了 n 值的有效性验证之后，CalcMinMax_ 使用指令 fld [r4_MinFloat] 来初始化 x87 FPU 寄存器栈上的 max_a。接着使用指令 fld [r4_MaxFloat] 初始化变量 min_a。内存操作数 [r4_MinFloat] 和 [r4_MaxFloat] 在 .const 区定义，它们是以十六进制编码表示的 x87 FPU 支持的最小和最大的单精度浮点数值。

循环处理中，CalcMiniMax_ 首先使用 fld real4 ptr[edx] 和 fld st(0) 指令把当前的数组元素拷贝两份并存储在 x87 FPU 寄存器栈上。执行完这些指令后，x87 FPU 寄存器栈上包含了
118 a[i]、a[i]、min_a 和 max_a，如图 4-3 所示。指令 fcomi st(0), st(2) 比较 a[i] 和 min_a，并设置 EFLAGS 中的状态标志。条件转移指令 fcmovnb st(0), st(2)（浮点条件转移）将确保 ST(0) 中包含这两个生中的较小者。然后，指令 fstp st(2) 将 ST(0) 拷贝至 ST(2) 并且弹出 x87 FPU 寄存器栈，堆栈上 min_a 的值也被更新。在此指令执行后，x87 FPU 寄存器栈上有 a[i]、min_a 和 max_a。

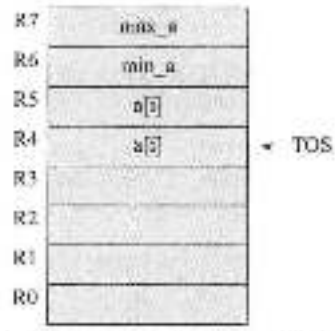


图 4-3 执行指令 fld st(0) 之后 x87 寄存器栈的内容

CalcMiniMax_ 使用类似的一系列指令来更新 max_a 的值。指令 fcomi st(0), st(2) 比较 a[i] 与 max_a，指令 fcmovb st(0), st(2) 保证 ST(0) 中包含较大值。然后，指令 fstp st(2) 更新 x87 FPU 寄存器栈上的 max_a。在循环处理完成后，x87 FPU 寄存器栈上包含了最终的 min_a 和 max_a。这些值随后被保存到相应的内存位置。运行 CalcMinMax 程序的结果见输出 4-4。

输出 4-4 示例程序 CalcMinMax

```
a[ 0] =   20.00
a[ 1] =  -12.00
a[ 2] =   42.00
a[ 3] =   97.00
a[ 4] =   14.00
a[ 5] =  -26.00
a[ 6] =   57.00
a[ 7] =   74.00
a[ 8] =  -18.00
a[ 9] =   63.00
a[10] =   34.00
a[11] =   -9.00

min1:   -26.00  max1:    97.00
119 min2:   -26.00  max2:    97.00
```

4.2.2 超越指令（超越函数指令）

接下来的示例程序名为 ConvertCoordinates，它演示了如何使用 x87 FPU 的超越指令。

程序中包含了几个对直角坐标和极坐标进行坐标转换的函数。清单 4-9 和清单 4-10 分别包含了该示例程序的 C++ 和汇编代码。

清单 4-9 ConvertCoordinates.cpp

```
#include "stdafx.h"

extern "C" void RectToPolar_(double x, double y, double* r, double* a);
extern "C" void PolarToRect_(double r, double a, double* x, double* y);

int _tmain(int argc, _TCHAR* argv[])
{
    double x1[] = { 0, 3, -3, 4, -4 };
    double y1[] = { 0, 3, -3, 4, -4 };
    const int nx = sizeof(x1) / sizeof(double);
    const int ny = sizeof(y1) / sizeof(double);

    for (int i = 0; i < ny; i++)
    {
        for (int j = 0; j < nx; j++)
        {
            double r, a, x2, y2;

            RectToPolar_(x1[i], y1[j], &r, &a);
            PolarToRect_(r, a, &x2, &y2);

            printf("[%d, %d]: ", i, j);
            printf("(%8.4lf, %8.4lf) ", x1[i], y1[j]);
            printf("(%8.4lf, %10.4lf) ", r, a);
            printf("(%8.4lf, %8.4lf)\n", x2, y2);
        }
    }

    return 0;
}
```

清单 4-10 ConvertCoordinates_.asm

```
        .model flat,c
        .const
DegToRad real8  0.01745329252
RadToDeg real8 57.2957795131
        .code                                                    120

; extern "C" void RectToPolar_(double x, double y, double* r, double* a);
;
; 描述：把直角坐标转换为极坐标

RectToPolar_ proc
        push ebp
        mov ebp,esp

; 计算角度。注意 fpatan 指令计算的是 atan2(ST(1)/ST(0))
        fld real8 ptr [ebp+16]          ;加载 y
        fld real8 ptr [ebp+8]           ;加载 x
        fpatan                          ;计算 atan2 (y/x)
        fmul [RadToDeg]                 ;角度值转换为弧度值
        mov eax,[ebp+28]
        fstp real8 ptr [eax]            ;保存角度

; 计算半径
```

```
        fld real8 ptr [ebp+8]           ;加载x
        fmul st(0),st(0)                ;x * x
        fld real8 ptr [ebp+16]          ;加载y
        fmul st(0),st(0)                ;y * y
        faddp                           ;x * x + y * y
        fsqrt                           ;sqrt(x * x + y * y)(求平方根)
        mov eax,[ebp+24]
        fstp real8 ptr [eax]            ;保存半径

        pop ebp
        ret
RectToPolar_    endp

; extern "C" void PolarToRect_(double r, double a, double* x, double* y);
;
; 描述：把极坐标转换为直角坐标

PolarToRect_ proc
        push ebp
        mov ebp,esp

; 计算 sin(a) 和 cos(a)
; 指令 fsincos 执行完后，ST(0) = cos(a)，ST(1) = sin(a)
        fld real8 ptr [ebp+16]          ;加载角度值
        fmul [DegToRad]                 ;角度值转换为弧度值
        fsincos                         ;计算 sin(ST(0)) 和 cos(ST(0))

[121]   fmul real8 ptr [ebp+8]          ;x = r * cos(a)
        mov eax,[ebp+24]
        fstp real8 ptr [eax]            ;保存x

        fmul real8 ptr [ebp+8]          ;y = r * sin(a)
        mov eax,[ebp+28]
        fstp real8 ptr [eax]            ;保存y

        pop ebp
        ret
PolarToRect_    endp
        end
```

在开始阅读代码前，让我们快速回顾一下二维坐标系的基础知识。一个二维屏幕上的点可以用 (x, y) 坐标唯一指定，值 x 和 y 分别表示点到两个垂直轴的距离。有序对 (x, y) 被称为直角坐标或笛卡儿坐标。在二维平面上的点也可以由半径矢量 r 和角度 θ 唯一表示，如图 4-4 所示，有序对 (r, θ) 被称为极坐标。

转换直角坐标和极坐标可以使用下面的公式：

$$r = \sqrt{x^2 + y^2} \qquad \theta = \text{atan2}(y/x), \quad \text{其中 } -\pi \le \theta \le +\pi$$

[122]

$$x = r\cos(\theta) \qquad y = r\sin(\theta)$$

文件 ConvertCoordinates_asm 中包含了函数 RectToPolar_，此函数可以把直角坐标转换为极坐标。在函数序言之后，参数 y 和 x 被两条 fld 指令加载到寄存器栈上。下一条指令 fpatan（反正切）计算 atan2(st(1)/st(0))，把求得的角度结果存储在 ST(1)，并弹出 x87 FPU 堆栈。指令 fmul [RadToDeg] 把弧度单位转换为角度单位，存储在 ST(0) 中。完成后极坐标角度保存在调用者指定的内存位置。

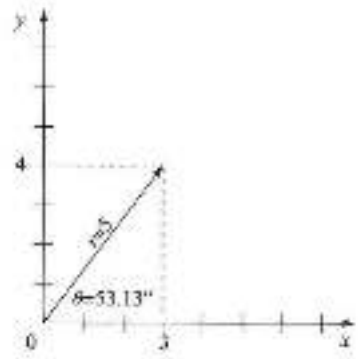


图 4-4 用直角坐标和极坐标表示点

r 值的计算方法如下：指令 fld real8 ptr [ebp+8] 把 x 加载到 x87 FPU 堆栈中，x 的平方数是使用指令 fmul st(0), st(0) 来计算的。然后使用同样的指令序列计算 y 的平方数。通过 faddp 指令将平方数相加，最后通过指令 fsqrt 得到最终的半径值 r，然后将其存储在指定的内存位置。

函数 RectToPolar_ 的反函数为 PolarToRect_。首先通过指令 fld real8 ptr [ebp+16] 把极坐标角度值加载到 x87 FPU 寄存器栈中。把角度单位转换为弧度单位是通过指令 fmul [DegToRad] 来实现的。之后使用指令 fsincos（正弦和余弦）计算 ST(0) 的正弦和余弦值。此指令完成后，正弦和余弦值分别存储在 ST(0) 和 ST(1) 中。补充一下，x87 FPU 也包含指令 fsin 和 fcos，然而，当两个值都需要时，使用 fsincos 指令更快些。

指令 fmul real8 ptr [ebp+8] 把极坐标半径乘以角度的余弦值，得到直角坐标的 x 值，将其存储在指定的内存位置。使用类似的指令序列，把极坐标半径乘以角度的正弦值，得到直角坐标的 y 值。输出 4-5 显示了示例程序 ConvertCoordinates 的运行结果。

输出 4-5 示例程序 ConvertCoordinates

```
[0, 0]: (  0.0000,   0.0000) (  0.0000,    0.0000) (  0.0000,   0.0000)
[0, 1]: (  0.0000,   3.0000) (  3.0000,   90.0000) ( -0.0000,   3.0000)
[0, 2]: (  0.0000,  -3.0000) (  3.0000,  -90.0000) ( -0.0000,  -3.0000)
[0, 3]: (  0.0000,   4.0000) (  4.0000,   90.0000) ( -0.0000,   4.0000)
[0, 4]: (  0.0000,  -4.0000) (  4.0000,  -90.0000) ( -0.0000,  -4.0000)
[1, 0]: (  3.0000,   0.0000) (  3.0000,    0.0000) (  3.0000,   0.0000)
[1, 1]: (  3.0000,   3.0000) (  4.2426,   45.0000) (  3.0000,   3.0000)
[1, 2]: (  3.0000,  -3.0000) (  4.2426,  -45.0000) (  3.0000,  -3.0000)
[1, 3]: (  3.0000,   4.0000) (  5.0000,   53.1301) (  3.0000,   4.0000)
[1, 4]: (  3.0000,  -4.0000) (  5.0000,  -53.1301) (  3.0000,  -4.0000)
[2, 0]: (  3.0000,   0.0000) (  3.0000,  180.0000) ( -3.0000,  -0.0000)
[2, 1]: (  3.0000,   3.0000) (  4.2426,  135.0000) ( -3.0000,   3.0000)
[2, 2]: ( -3.0000,  -3.0000) (  4.2426, -135.0000) ( -3.0000,  -3.0000)
[2, 3]: ( -3.0000,   4.0000) (  5.0000,  126.8699) ( -3.0000,   4.0000)
[2, 4]: ( -3.0000,  -4.0000) (  5.0000, -126.8699) ( -3.0000,  -4.0000)
[3, 0]: (  4.0000,   0.0000) (  4.0000,    0.0000) (  4.0000,   0.0000)
[3, 1]: (  4.0000,   3.0000) (  5.0000,   36.8699) (  4.0000,   3.0000)
[3, 2]: (  4.0000,  -3.0000) (  5.0000,  -36.8699) (  4.0000,  -3.0000)
[3, 3]: (  4.0000,   4.0000) (  5.6569,   45.0000) (  4.0000,   4.0000)
[3, 4]: (  4.0000,  -4.0000) (  5.6569,  -45.0000) (  4.0000,  -4.0000)
[4, 0]: ( -4.0000,   0.0000) (  4.0000,  180.0000) ( -4.0000,  -0.0000)
[4, 1]: ( -4.0000,   3.0000) (  5.0000,  143.1301) ( -4.0000,   3.0000)
[4, 2]: ( -4.0000,  -3.0000) (  5.0000, -143.1301) ( -4.0000,  -3.0000)
[4, 3]: ( -4.0000,   4.0000) (  5.6569,  135.0000) ( -4.0000,   4.0000)
[4, 4]: ( -4.0000,  -4.0000) (  5.6569, -135.0000) ( -4.0000,  -4.0000)
```


4.2.3 栈的高级应用

目前为止的示例程序都没有特意强调 x87 寄存器栈的限制，在最后的 x87 FPU 示例程序 CalcLeastSquares 中将着重讨论这一点。CalcLeastSquares 演示如何使用 x87 FPU 计算最小二乘法拟合直线，相应的源代码在 CalcLeastSquares.cpp 和 CalcLeastSquares_.asm 中，如清单 4-11 和清单 4-12 所示。

清单 4-11 CalcLeastSquares.cpp

```
#include "stdafx.h"
#include <math.h>
```

```
extern "C" double LsEpsilon_;
extern "C" bool CalcLeastSquares_(const double* x, const double* y, int n,↩
double* m, double* b);

bool CalcLeastSquaresCpp(const double* x, const double* y, int n, double* m,↩
double* b)
{
    if (n <= 0)
        return false;

    double sum_x = 0;
    double sum_y = 0;
    double sum_xx = 0;
    double sum_xy = 0;

    for (int i = 0; i < n; i++)
    {
        sum_x += x[i];
        sum_xx += x[i] * x[i];
        sum_xy += x[i] * y[i];
        sum_y += y[i];
    }

    double denom = n * sum_xx - sum_x * sum_x;

    if (LsEpsilon_ >=fabs(denom))
124     return false;
    *m = (n * sum_xy - sum_x * sum_y) / denom;
    *b = (sum_xx * sum_y   sum_x * sum_xy) / denom;
    return true;
}

int _tmain(int argc, _TCHAR* argv[])
{
    const int n = 7;
    double x[n] = { 0, 2, 4, 6, 8, 10, 12};
    double y[n] = { 51.125, 62.875, 71.25, 83.5, 92.75, 101.1, 110.5 };
    double m1 = 0, m2 = 0;
    double b1 = 0, b2 = 0;
    bool rc1, rc2;

    rc1 = CalcLeastSquaresCpp(x, y, n, &m1, &b1);
    rc2 = CalcLeastSquares_(x, y, n, &m2, &b2);

    for (int i = 0; i < n; i++)
        printf("%12.4lf, %12.4lf\n", x[i], y[i]);

    printf("\n");
    printf("rc1: %d  m1: %12.4lf  b1: %12.4lf\n", rc1, m1, b1);
    printf("rc2: %d  m2: %12.4lf  b2: %12.4lf\n", rc2, m2, b2);
    return 0;
}
```

清单 4-12 CalcLeastSquares_.asm

```
        .model flat,c
        .const
        public LsEpsilon_
LsEpsilon_ real8 1.0e-12                    ;验证 denon 值不为零
        .code

; extern "C" bool CalcLeastSquares_(const double* x, const double* y, int n,↩
```

```
    double* x, double* b);
;
; 描述：计算最小二乘法拟合直线的斜率和截距
;
; 返回：0 = 出错
;       1 = 成功

CalcLeastSquares_ proc
        push ebp
        mov ebp,esp
        sub esp,8                       ;为局部内部变量 denom 留空间                    125
        xor eax,eax                     ;预置错误的返回码
        mov ecx,[ebp+16]                ;n
        test ecx,ecx
        jle Done                        ;如 n <= 0，跳转
        mov eax,[ebp+8]                 ;指向 x
        mov edx,[ebp+12]                ;指向 y

; 初始化所有和变量为 0
        fldz                            ;sum_xx
        fldz                            ;sum_xy
        fldz                            ;sum_y
        fldz                            ;sum_x
;STACK: sum_x, sum_y, sum_xy, sum_xx

@@:     fld real8 ptr [eax]             ;加载下一个 x
        fld st(0)
        fld st(0)
        fld real8 ptr [edx]             ;加载下一个 y
;STACK: y, x, x, x, sum_x, sum_y, sum_xy, sum_xx

        fadd st(5),st(0)                ;sum_y += y
        fmulp
;STACK: xy, x, x, sum_xx sum_y, sum_xy, sum_xx

        faddp st(5),st(0)               ;sum_xy += xy
;STACK: x, x, sum_x, sum_y, sum_xy, sum_xx

        fadd st(2),st(0)                ;sum_x += x
        fmulp
;STACK: xx, sum_x, sum_y, sum_xy, sum_xx

        faddp st(4),st(0)               ;sum_xx += xx
;STACK: sum_x, sum_y, sum_xy, sum_xx

; 更新指针并重复循环，直到每个元素都被处理
        add eax,8
        add edx,8
        dec ecx
        jnz @@

; 计算 denom = n * sum_xx - sum_x * sum_x
        fild dword ptr [ebp+16]         ;n
        fmul st(0),st(4)                ;n * sum_xx
;STACK: n * sum_xx, sum_x, sum_y, sum_xy, sum_xx

        fld st(1)
        fld st(0)
;STACK: sum_x, sum_x, n * sum_xx, sum_x, sum_y, sum_xy, sum_xx                           126

        fmulp
        fsubp
```

```
        fst real8 ptr [ebp-8]           ;保存 denom
;STACK: denom, sum_x, sum_y, sum_xy, sum_xx

; 验证 denom 值是否有效
        fabs                            ;fabs(denom)
        fld real8 ptr [lsEpsilon_]
        fcomp st(0),st(1)               ;比较 epsilon_ 和 fabs(denom)
        fstp st(0)                      ;从栈中移除 fabs(denom)
        jae InvalidDenom                ;如果 LsEpsilon_ >=fabs(denom), 则跳转
;STACK: sum_x, sum_y, sum_xy, sum_xx

; 计算斜率 slope = (n * sum_xy - sum_x * sum_y) / denom
        fild dword ptr [ebp+16]
;STACK: n, sum_x, sum_y, sum_xy, sum_xx

        fmul st(0),st(3)                ;n * sum_xy
        fld st(2)                       ;sum_y
        fld st(2)                       ;sum_x
        fmulp                           ;sum_x * sum_y
        fsubp                           ;n * sum_xy - sum_x * sum_y
        fdiv real8 ptr [ebp-8]          ;计算斜率
        mov eax,[ebp+20]
        fstp real8 ptr [eax]            ;保存斜率
;STACK: sum_x, sum_y, sum_xy, sum_xx

; 计算截距 intercept = (sum_xx * sum_y - sum_x * sum_xy) / denom
        fxch st(3)
;STACK: sum_xx, sum_y, sum_xy, sum_x

        fmulp
        fxch st(2)
;STACK: sum_x, sum_xy, sum_xx * sum_y

        fmulp
        fsubp
;STACK: sum_xx * sum_y - sum_x * sum_xy

        fdiv real8 ptr [ebp-8]          ;计算截距
        mov eax,[ebp+24]
        fstp real8 ptr [eax]            ;保存截距
        mov eax,1                       ;设置成功返回码

Done:   mov esp,ebp
        pop ebp
127     ret

InvalidDenom:
; 清除 x87 FPU 寄存器栈
        fstp st(0)
        fstp st(0)
        fstp st(0)
        fstp st(0)
        xor eax,eax                     ;设置错误返回码
        mov esp,ebp
        pop ebp
        ret
CalcLeastSquares_ endp
        end
```

简单线性回归是一种对两个变量进行线性建模的统计方法。最常见的是最小二乘拟合，它针对两个变量的一组样本数据，确定最佳的拟合曲线。在简单线性回归模型中，使用的曲

线是直线，其方程为 $y=mx+b$。在此公式中，x 表示自变量，y 表示因变量（或测量变量），m 为直线的斜率，b 是直线在 y 轴上的截距。最小二乘直线的斜率和截距是计算样本点和直线间最小误差平方和得到的。通过计算出的斜率和截距，最小二乘直线经常用于在 x 值已知的情况下求 y 值。如果有兴趣了解更多关于简单线性回归和最小二乘拟合，请查阅附录 C 中列出的资源。

在示例程序 CalcLeastSquares 中，下面的公式用来计算最小二乘斜率和截距点：

$$m=\frac{n\sum_i x_i y_i-\sum_i x_i\sum_i y_i}{n\sum_i x_i^2-\left(\sum_i x_i\right)^2} \qquad b=\frac{\sum_i x_i^2\sum_i y_i-\sum_i x_i\sum_i x_i y_i}{n\sum_i x_i^2-\left(\sum_i x_i\right)^2}$$

第一眼看去，斜率和截距公式可能会让人感到有些困难。然而，仔细查看一下，经过一系列的简化，公式的运作方式会变得明显。首先，计算斜率和截距的分母是相同的，这意味着该值只需要计算一次。其次，仅需要计算四个简单的求和量（或求和变量），如下所示：

$$sum\_x=\sum_i x_i \qquad sum\_y=\sum_i y_i$$

$$sum\_xy=\sum_i x_i y_i \qquad sum\_xx=\sum_i x_i^2$$

随后计算变量总和以及最小二乘斜率和截距，使用的都是比较简单的乘法、减法和除法。 128

函数 CalcLeastSquares_ 使用了 x87 FPU 的所有八个寄存器，因此某种程度上比本章其他的示例程序更复杂些。当编写使用超过四个 x87 FPU 寄存器栈的函数时，建议在代码中用注释来注明值保存在哪个寄存器上。在清单 4-12 中，以 STACK 字样开头的注释表示的是在执行完上一条指令后 x87 FPU 寄存器栈包含的内容。这些内容以 ST(0) 开始，以从栈顶到栈底的方式排列。

让我们读一下函数 CalcLeastSquares_ 的源代码。在进行了 n 值的有效性检查之后，一组 fldz 指令把 sum_x、sum_y、sum_xy 和 sum_xx 初始化为 0.0。上一句话中这些和变量的顺序代表了它们在 x87 FPU 寄存器栈中的位置从行 ST(0) 开始。这个顺序在执行循环处理时不会改变，但是和变量相对于栈顶的位置是会变化的。以 x87 FPU 寄存器栈保存中间值会略微增加算法的复杂性，但相比使用内存作为中间值，性能要更好。

在初始化和变量为 0.0 之后，函数进入了循环，重复进行数据传送并且计算和变量。在循环开始的时候，使用一组 fld 指令把当前的 x 值和 y 值加载到 x86 FPU 寄存器栈中。执行完这些指令后，x87 FPU 的寄存器栈已经充全满了，如图 4-5 所示（如果有另外一个值被载入 x87 FPU 的寄存器栈，sum_xx 的值将会丢失）。然后使用一系列的浮点加法和乘法运算计算和变量。注意某些 fadd 指令使用了不是 ST(0) 的目标操作数。同样需要特别注意的是，在计算的时候，和变量在栈上的相对位置也在改变。上面的迭代循环完成后，x87 FPU 的栈上包含了四个和变量的最终值。

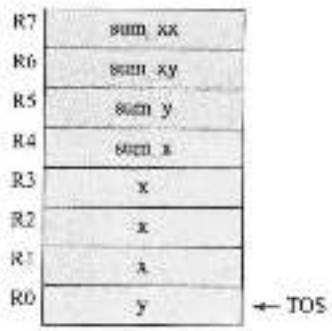


图 4-5 在执行完指令 fld real8 ptr [edx] 之后的 x87 FPU 栈的内容 129

接着计算斜率和截距公式中的共用分母（denom）。

这要求函数计算两个中间值 n*sum_xx 和 sum_x*sum_x。它使用了 fld 和 fmul 的变种指令。计算完成后，从前者减去后者，得到最终的分母值，保存到 x86 栈的临时局部变量中。在进行下一次计算前，会验证此值，以避免产生除 0 错误。如果分母无效，程序会跳转到函数 CalcLeastSuares_ 结尾附近的一段代码中，清理 x87 FPU 栈，同时寄存器 EAX 载入适当的错误代码，返回给调用者。

得到分母后，函数计算出最小二乘斜率。在计算斜率的过程中，在 x87 FPU 栈上的和变量的顺序一直保持着，因为在计算截距的过程中需要它们。斜率被保存在调用者指定的内存位置。计算最小二乘截距的时候，需要用两个 fxch 指令。这样做的原因是，所有的 x87 FPU 计算指令必须显式或者隐式地使用 ST(0) 作为操作数。最后的 fstp 指令把截距值保存在调用者指定的内存位置。同时产生一个空的 x87 FPU 寄存器栈。输出 4-6 显示了示例程序 CalcLeastSquares 的执行结果。

输出 4-6 示例程序 CalcLeastSquares

```
        0.0000,     51.1250
        2.0000,     62.8750
        4.0000,     71.2500
        6.0000,     85.5000
        8.0000,     92.7500
       10.0000,    101.1000
       12.0000,    110.5000

rc1: 1  m1:      4.9799  b1:     52.2920
rc2: 1  m2:      4.9799  b2:     52.2920
```

4.3 总结

在本章中，我们进一步学习了 x87 FPU 的架构以及如何利用这些资源进行各种浮点运算。与 x86 的通用寄存器不同，在对 x87 编程的时候，需要不同的思维方式才能有效地使用 x87 FPU 的栈结构寄存器组。随着你在这种架构上编程经验的不断增长，这种思维变化会逐步进入你的潜意识。

如果你熟悉 x86-SSE 的标量浮点功能，你可能会质疑本书为什么会花这么多的篇幅来介绍一个许多人认为过时的架构。这样做有几个原因。首先，必须承认在可预见的未来，仍有相当多的旧代码使用 x87 FPU 架构和指令。其次，即使编译器配置为使用 SSE2 指令产生浮点代码，Visual C++ 的 32 位程序调用约定仍然采用 x87 FPU 寄存器栈保存浮点返回值。这意味着程序员必须至少要对 x87 FPU 有一个基本的理解。最后一个理由，诸如 Intel Quark 之类的超低功耗微处理器架构并不提供 x86-SSE 浮点处理资源。对 Quark 或者其他类似平台的并发而言，除了在代码中使用 x87 FPU 来进行浮点运算外，没有其他选择。

MMX 技术

第 1 章到第 4 章集中讨论了 x86-32 平台的基本功能。你学习了 x86 的基本数据类型、通用寄存器、内存寻址模式以及 x86-32 的核心指令集。你还学习了许多示例代码，它们展示了 x86 汇编语言编程的基本要点，包括基本操作数、整型运算、比较运算、条件跳转以及常见数据结构的操作。第 3 章和第 4 章分析了 x87 FPU 的架构，包括栈形式的寄存器组以及如何使用汇编语言进行浮点运算。

接下来的十二章我们将集中讨论 x86 平台的单指令多数据（SIMD）功能。本章首先介绍 x86 的第一个 SIMD 扩展——MMX 技术。MMX 技术为 x86 平台增加了整数 SIMD 处理的能力。本章首先会以一些基本的 SIMD 处理概念开篇。接下来会描述一下回绕（wraparound）和饱和（saturated）整型运算的区别以及后者的使用场景。之后会讨论 MMX 执行环境，包括它的寄存器组和支持的数据类型。最后会对 MMX 指令集做一个总结。

MMX 技术为后续的 x86 SIMD 扩展（包括 x86-SSE 和 x86-AVX）奠定了一个基础，我们将会在第 7 章到第 16 章对 x86 SIMD 扩展进行讨论。如果你的终极目标是编写一个使用那些最新 SIMD 扩展的软件，那么我们强烈建议你先认真学习一下本章的内容，因为这最终会减少你在 x86 SIMD 学习过程中所花的总时间。

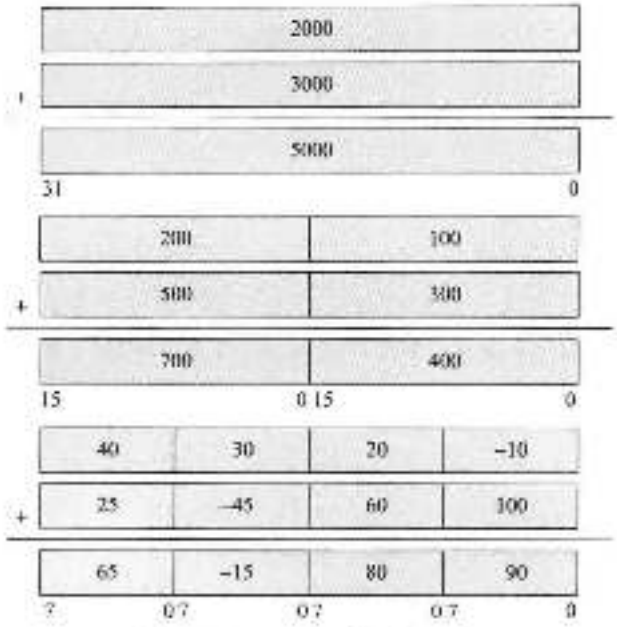
5.1 SIMD 处理概念

在讨论 MMX 技术的特点之前，本节先介绍一下基本的 SIMD 处理概念。正如 SIMD 的字面意义一样，一个 SIMD 计算单元会同时对多个数据项进行同样的操作。标准的 SIMD 操作包括基本运算（加法、减法、乘法和除法）、移位、比较和数据转换。处理器通过解析操作数在寄存器或者内存中的位模式（bit pattern）来实现 SIMD 运算。例如，一个 32 位寄存器能存放一个 32 位整型数值；它同时也能够存放 2 个 16 位整型数或者 4 个 8 位整型数，如图 5-1 所示。



图 5-1 存放不同大小整数的 32 位寄存器

针对图 5-1 所示的位模式，完全有可能对其中的所有数据元素都执行一种操作。图 5-2 中举了一个更加详细的例子。图中展示了对一个 32 位整型、两个 16 位整型以及四个 8 位整型数据的相加操作。我们可以看到，当同时使用多个数据项的时候，计算的性能会大幅提升，因为处理器可以并行地进行运算。在图 5-2 中，处理器可以对要处理的操作数同时进行 16 位或 8 位整型相加。



134 图 5-2 SIMD 整型加法的示例

在 x86 平台上，MMX 技术支持 64 位宽的寄存器和内存操作数。也就是说，一个 SIMD 运算可以操作两个 32 位、四个 16 位或者八个 8 位数据。此外，MMX SIMD 运算并不仅限于加法或者减法这样的简单算术运算。对于其他常见的运算，如移位、布尔运算、比较和数据转换也是支持的。MMX 技术还支持一些通常需要好几个指令来完成的高级原子操作。图 5-3 演示了 MMX pmaxub（无符号单字节组合整型的最大值）指令所做的运算，它利用 64 位 MM 寄存器来存放 8 位无符号整数。在这个例子中，处理器同时比较每对 8 位无符号整型数据，将其中较大的数值存放在目标操作数中。在本章后面的部分，我们将进一步介绍 MMX 指令集。

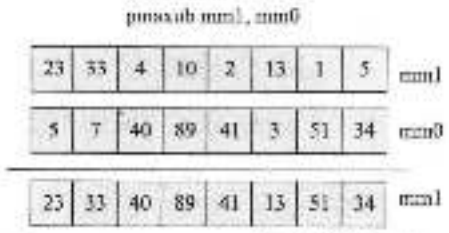


图 5-3 MMX pmaxub 指令的执行过程

5.2 回绕和饱和运算

MMX 技术中一个相当有用的功能就是支持饱和整型运算（saturated integer arithmetic）。在饱和整型运算中，计算的结果会被处理器自动剪裁，以保证数据不会上溢或下溢。这和普通的整型回绕运算不同，整型回绕运算会将上溢或下溢的结果保留。饱和运算在处理像素值时特别有用，因为不再需要逐一检查每个像素的计算结果是否发生上溢或者下溢。MMX 技术支持对 8 位和 16 位有符号和无符号整型数做饱和运算。

让我们看几个回绕和饱和运算的例子。图 5-4 演示了对一个 16 位有符号整数做加法的例子，分别使用了回绕运算和饱和运算。当使用回绕运算进行两个 16 位有符号整型数相加的时候，产生了一个上溢出；然而，当使用饱和运算的时候，计算结果被剪裁成了 16 位有符号整型数的最大值。图 5-5 展示了一个类似的例子，它使用了 8 位无符号整数。除了加法之外，MMX 技术也支持饱和整型减法，如图 5-6 所示。表 5-1 总结了相对不同大小和符号

135 类型整数做饱和运算时的范围边界。



图 5-4 16 位有符号整型加法——使用回绕运算和饱和运算

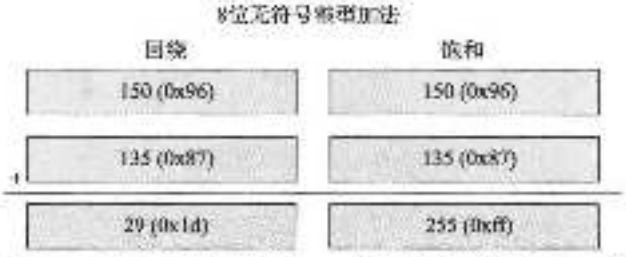


图 5-5 8 位无符号整型加法——使用回绕运算和饱和运算



图 5-6 16 位有符号整型减法——使用回绕运算和饱和运算

表 5-1 饱和运算的范围边界

整数类型	低边界	高边界
8 位有符号	-128 (0x80)	127 (0x7f)
8 位无符号	0	255 (0xff)
16 位有符号	-32768 (0x8000)	32767 (0x7fff)
16 位无符号	0	65535 (0xffff)

136

5.3 MMX 执行环境

从应用程序的角度来看，MMX 技术为 x86-32 核心平台增加了八个 64 位的寄存器，取名为 MM0 ~ MM7，如图 5-7 所示。利用这些寄存器，可以对八个 8 位整型、四个 16 位整型或两个 32 位整型数据做 SIMD 运算。而且，无符号和有符号整数都是支持的。也可以使用 MMX 寄存器对 64 位整型做某些有限次的运算。与 x87 FPU 寄存器组不同，MMX 寄存器是可以直接寻址的；没有使用基于栈的架构。MMX 寄存器不能用于浮点运算或寻址内存中的操作数。图 5-8 展示了 MMX 支持的组合（packed）数据类型。

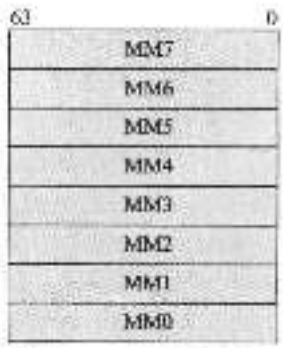
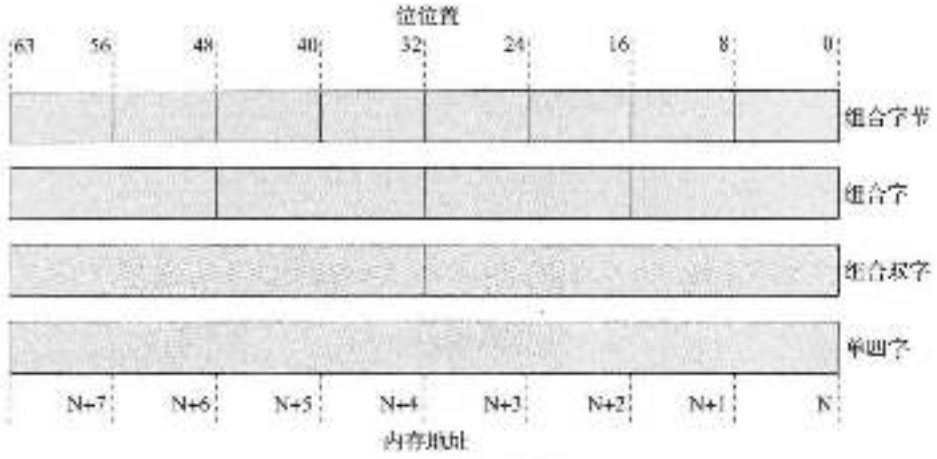


图 5-7 MMX 寄存器组



137 图 5-8 MMX 数据类型

在 x86 处理器内部，MMX 寄存器和 x87 浮点运算单元的寄存器是混叠的。也就是说，MMX 和 x87 浮点运算单元的寄存器使用了同一块存储区。这种混叠使得在混合使用 MMX 和 x87 浮点运算单元指令时有一些限制。为了避免 MMX 和 x87 FPU 执行单元之间的冲突，在从执行 MMX 指令到执行 x87 FPU 指令转换之前，MMX 的状态信息必须通过 emms（清除 MMX 技术状态）指令来清空。在第 6 章的示例代码中，我们会更详细地演示这一要求。如果没有正确使用 emms 指令，可能会导致 x87 浮点运算单元产生一些意外的异常或是无效的计算结果。

5.4 MMX 指令集

本节会对 MMX 指令集进行一个简要的介绍。和前几章的指令集概述类似，本节的目标是使读者对 MMX 指令集有一个基本的了解。至于 MMX 指令集详尽的信息，如可用的操作数和可能的异常，都可以在 AMD 和 Intel 的参考手册中找到。本书的附录 C 中就包括了这些手册的列表。在第 6 章中讨论的示例代码也会包含一些指令的详细介绍。

可以按功能把 MMX 指令集划分为以下八类：

- 数据传输
- 算术运算
- 比较
- 转换
- 逻辑和位移
- 解组和重排（shuffle）
- 插入和提取
- 状态和缓存控制

本节的指令集概述部分涵盖了 MMX 最初发布的所有指令集，同时也包括了在 x86 SSE 增强版本（SSE、SSE2、SSE3 或 SSSE3）中新增的指令。在表中我们总结了每个指令所需要的 MMX 或 x86-SSE 版本。除非特别注明，MMX 指令的源操作数可以是一个内存区域或是一个 MMX 寄存器；而目标操作数必须是一个 MMX 寄存器。当引用一个内存区域时，MMX 指令能使用第 1 章中介绍的任何一种 x86-32 寻址模式。内存操作数地址对齐不是必需的，然
138 而我们强烈建议进行对齐，因为从非对齐的内存区域中读取数据会需要多个时钟周期。

注意 MMX指令不会更新任何EFLAGS寄存器中的状态位。必须用软件程序来检查、判定或防止潜在的错误情况，如上溢出或下溢出。

大多数MMX指令的助记符使用字母b（字节）、w（字）、d（双字）以及q（四字）来标识需要处理元素的宽度。

5.4.1 数据传输

数据传输指令被用来在MMX寄存器、通用寄存器和内存之间复制组合整型数据。表5-2对数据传输指令进行了总结。

表5-2 MMX数据传输指令

助记符	描　述	版本
movd	复制MMX寄存器中的低位双字到一个通用寄存器或内存中。也可以把通用寄存器或内存中的数据复制到MMX寄存器的低位双字中	MMX
movq	把一个MMX寄存器的内容复制到另外一个MMX寄存器中。这个指令还能被用来把一个内存区域中的内容复制到一个MMX寄存器中；或者把MMX寄存器中的内容复制到内存中	MMX

5.4.2 算术运算

算术运算指令被用来对组合操作数进行基本算术运算（加法、减法以及乘法）。它也包括了一些执行高级运算的指令，如最小或最大、取平均、计算绝对值以及整数符号变换。除非另行说明，所有的算术运算指令都支持有符号和无符号整型。表5-3列举了MMX的算术运算指令。 139

表5-3 MMX算术运算指令

助记符	描　述	版本
paddb paddw paddd paddq	使用指定的操作数进行组合整型加法	MMX SSE2(paddq)
paddsb paddsw	用饱和运算对有符号组合整型进行加法计算	MMX
paddusb paddusw	用饱和运算对无符号组合整型进行加法计算	MMX
psubb psubw psubd psubq	使用指定的操作数进行组合整型减法。源操作数存放减数，目标操作数存放被减数	MMX SSE2(psubq)
psubsb psubsw	用饱和运算对有符号组合整型进行减法计算。源操作数存放减数，目标操作数存放被减数	MMX
psubusb psubusw	用饱和运算对无符号组合整型进行减法计算。源操作数存放减数，目标操作数存放被减数	MMX
pmaddwd	对有符号组合整型进行乘法，然后对结果中相邻的数据元素进行有符号的求和加法。这个指令可以用来进行整型向点乘运算	MMX
pmaddubsw	进行一个组合饱和乘法，其中源操作数中存放有符号的字节，目标操作数中存放无符号的字节。然后对产生的有符号单字值进行饱和运算相加，最后将结果存放在目标操作数中	SSSE3

（续）

助记符	描　述	版本
pmuludq	将源操作数中的低位双字与目标操作数的低位双字相乘，产生的四字结果存放在目标操作数中	SSE2
pmullw	使用单字值进行有符号组合整型乘法，然后将每个双字乘积中的低位单字存放在目标操作数中	MMX
pmulhw	使用单字值进行有符号组合整型乘法，然后将每个双字乘积中的高位单字存放在目标操作数中	MMX
pmulhuw	使用单字值进行无符号组合整型乘法，然后将每个双字乘积中的高位单字存放在目标操作数中	SSE
pmulhrsw	使用单字值进行无符号组合整型乘法，然后将双字乘积舍入到18位，再将位到16位，最后存放到目标操作数中	SSE3
pavgb pavgw	对指定操作数中无符号类型数据计算组合平均值	SSE
pmaxub	对两组无符号单字节组合整型数据进行比较，保存每个比较中较大的单字节数值	SSE
pminub	对两组无符号单字节组合类型数据进行比较，保存每个比较中较小的单字节数值	SSE
pmaxsw	对两组有符号单字组合整型数据进行比较，保存每个比较中较大的单字数值	SSE
pminsw	对两组有符号单字组合整型数据进行比较，保存每个比较中较小的单字数值	SSE
pabsb pabsw pabsd	计算每组组合整型数据元素中的绝对值	SSSE3
psignb psignw psignd	根据源操作数中对应数据元素的符号，对目标操作数中每个有符号类型数据进行取负、取零或保持不变的操作	SSSE3
phaddw phaddd	对源操作数和目标操作数中相邻的数据元素进行整型加法操作	SSSE3
phaddsw	利用饱和运算，对源操作数和目标操作数中相邻的数据元素进行有符号的整型加法操作，结果存放在目标操作数中	SSSE3
phsubw phsubd	对源操作数和目标操作数中相邻的数据元素进行整型减法操作	SSSE3
phsubsw	利用饱和运算，对源操作数和目标操作数中相邻的数据元素进行有符号的整型减法操作，结果存放在目标操作数中	SSSE3

[140]

[141]

5.4.3 比较

MMX 的比较指令会对两个组合操作数逐个进行比较，比较的结果存放在对应的目标操作数中。表 5-4 列举了 MMX 的比较指令。

表 5-4　MMX 比较指令

助记符	描　述	版本
pcmpeqb pcmpeqw pcmpeqd	逐个元素比较两个组合整型操作数是否相等。如果源操作数和目标操作数中数据元素相等，则对应的目标操作数中的数据元素被设置为全 1；如果不相等，则目标操作数中该数据元素被设置为全 0	MMX
pcmpgtb pcmpgtw pcmpgtd	逐个元素比较两个有符号组合整型操作数的大小。如果目标操作数中数据元素较大，则对应的目标操作数中的数据元素被设置为全 1；否则，目标操作数中的数据元素被设置为全 0	MMX

5.4.4 转换

MMX 的转换指令可以用来对操作数中的数据元素进行组合操作，它们使得整型数据的转换变得更为简单。表 5-5 介绍了 MMX 的转换指令。

表 5-5 MMX 转换指令

助记符	描 述	版本
packsswb packssdw	使用有符号饱和运算，将源操作数和目标操作数中的组合整型单字或双字转换为组合整型字节或单字	MMX
packuswb	使用无符号的饱和运算，将源操作数和目标操作数中的组合整型单字转换为组合整型字节	MMX

5.4.5 逻辑和位移

MMX 的逻辑与位移指令可以用来对操作数进行按位逻辑运算；也可以用来对组合操作数中的单个数据元素进行逻辑和算术位移。表 5-6 总结了逻辑和位移指令。 142

表 5-6 MMX 逻辑和位移指令

助记符	描 述	版本
pand	对指定的源操作数和目标操作数进行按位的逻辑与操作	MMX
pandn	对指定的源操作数和反转的目标操作数进行按位的逻辑与操作	MMX
por	对指定的源操作数和目标操作数进行按位的逻辑或操作	MMX
pxor	对指定的源操作数和目标操作数进行按位的逻辑异或操作	MMX
psllw pslld psllq	对目标操作数中的每个数据元素进行逻辑左移操作，低位用 0 补进。源操作数中存放着需要左移的位数，可以是内存地址、MMX 寄存器或者是立即操作数	MMX
psrlw psrld psrlq	对目标操作数中的每个数据元素进行逻辑右移操作，高位用 0 补进。源操作数中存放着需要右移的位数，可以是内存地址、MMX 寄存器或者是立即操作数	MMX
psraw psrad	对目标操作数中的每个数据元素进行算术右移操作，高位用符号位补进。源操作数中存放着需要右移的位数，可以是内存地址、MMX 寄存器或者是立即操作数	MMX
palignr	将目标操作数和源操作数组成一个临时数值，然后使用立即操作数指定的计数对这个临时数值进行按字节右移操作，将临时数值最右边的四字存入目标操作数	SSSE3

5.4.6 解组和重排

MMX 的解组与重排指令包括了对一个组合操作数中的数据元素进行交织（解组）的指令，也可以用于对组合操作数中的数据元素重新排序（重排）。这些指令如表 5-7 所示。 143

表 5-7 MMX 解组和重排指令

助记符	描 述	版本
punpckhbw punpckhwd punpckhdq	解组并交织源操作数和目标操作数中的高位数据元素。这些指令可以用于把字节转换字、字转换为双字以及双字转换为四字	MMX
punpcklbw punpcklwd punpckldq	解组并交织源操作数和目标操作数中的低位数据元素。这些指令可以用于把字节转换字、字转换为双字以及双字转换为四字	MMX

（续）

助记符	描　述	版本
pshufb	将操作数指定一个控制掩码。根据这个掩码对目标操作数中的字节进行重排操作。这个指令用于对组合操作数的字节进行重新排列	SSSE3
pshufw	立即操作数指定一个掩码，根据这个掩码对源操作数中的字进行重排操作。这个指令用于对组合操作数的字进行重新排列	SSE

5.4.7 插入和提取

MMX 的插入和提取指令可用于在 MMX 寄存器中插入或提取单字值。表 5-8 总结了插入和提取指令。

表 5-8　MMX 插入和提取指令

助记符	描　述	版本
pinstrw	复制通用寄存器中的低位单字，将其插入一个 MMX 寄存器中。插入的位置由立即操作数指定	SSE
pextrw	从 MMX 寄存器中提取一个单字，将其复制到一个通用寄存器的低位单字中。提取的位置由立即操作数指定	SSE

144

5.4.8 状态和缓存控制

MMX 的状态和缓存控制指令用于通知处理器从 MMX 到 x87 FPU 的状态转换，此外还包括了使用非临时提示执行内存存储的指令。非临时提示会通知处理器将数据直接写入内存，而不是暂时存入缓存中。这样可以提高音视频编码程序的缓存效率，因为缓存干扰被消除了。表 5-9 介绍了状态和缓存控制指令。

表 5-9　MMX 的状态和缓存控制指令

助记符	描　述	版本
emms	通过重置 x87 FPU 标签字来清除 MMX 的状态信息，用以标识所有的 x87 FPU 寄存器都已经被清空。这个指令在每次从 MMX 指令到 x87 FPU 指令的转换之前必须执行	MMX
movntq	使用非临时提示将 MMX 寄存器中的内容复制到内存中	SSE
maskmovq	使用非临时提示有条件地将 MMX 寄存器中的某些字节复制到内存中。另一个 MMX 寄存器中存放了一个掩码值，用于指定某些字节需要被复制。EDI 寄存器指向目标的内存位置	SSE

5.5 总结

本章讲解了 MMX 技术的基本概念，包括寄存器组、支持的数据类型以及指令集；我们也介绍了基本的 SIMD 处理概念以及回绕和饱和整型运算。至于软件开发者如何利用 MMX 技术来解决现实世界中的编程难题，答案并不像其他的 x86 计算资源（如 x87 FPU）那么明显。在第 6 章中，我们将通过一系列的示例代码来演示 MMX 技术的优越之处，并介绍如何正确使用 MMX 指令集。

145
|
146

MMX 技术编程

本章主要介绍 MMX 编程的方法。首先介绍 MMX 编程的基本要领，之后通过几个示例程序演示一些高级的 MMX 编程技巧，用以对整型数组处理进行加速。在阅读本章讨论的内容和示例程序前，请确保对本书第 5 章的内容已经比较熟悉。

我们在第 5 章中曾经提到，x86 SIMD 扩展包括 x86-SSE 和 x86-AVX，而 MMX 技术是它们的基础。这意味着，即使你的最终目标只是使用一种新扩展来开发代码，我们也强烈建议你认真阅读本章，以对 MMX 编程及其指令集建立全面的认识。此外，理解本章的内容对维护旧的 MMX 代码也是很有必要的。

6.1 MMX 编程基础

让我们先分析几个演示 MMX 编程基础的示例程序来开始探索 MMX 技术。第一个例子将演示如何对不同长度的整数（字节和字）数组合整型加法；第二个例子演示对组合操作数执行 MMX 移位指令；最后一个例子展示如何对组合有符号整数做乘法。本节的示例程序完全是为了教学目的，在接下来的几节中将会介绍如何使用 MMX 指令集实现比较完整的算法。

在阅读示例程序前，我们首先浏览一些数据类型，它们是为了简化 MMX 示例程序而定义的。头文件 MiscDefs.h 中包含若干 C++ typedef 语句，定义了常见的有符号和无符号整数类型。清单 6-1 给出了这些类型的定义，本章和接下来章节的示例程序会用到它。清单 6-2 中的头文件 MmxVal.h 中，声明了联合体（union）MmxVal，主要用来在 C++ 和汇编语言函数之间交换数据。MmxVal 联合体中声明的项与 MMX 支持的组合数据类型是对应的。联合体中还包含若干文本字符串辅助函数的声明，主要用来格式化和显示 MmxVal 变量的内容。MmxVal.cpp 中包含 ToString 转换函数的定义，其内容在这里没有列出来。这个文件包含在本书配套源代码的发布文件中，位于子目录 CommonFiles 下。

清单 6-1 MiscDefs.h

```
#pragma once

// 有符号整型类型定义
typedef __int8 Int8;
typedef __int16 Int16;
typedef __int32 Int32;
typedef __int64 Int64;

// 无符号整型类型定义
typedef unsigned __int8 Uint8;
typedef unsigned __int16 Uint16;
typedef unsigned __int32 Uint32;
typedef unsigned __int64 Uint64;
```

清单 6-2 MmxVal.h

```
#pragma once

#include "MiscDefs.h"

union MmxVal
{
    Int8  i8[8];
    Int16 i16[4];
    Int32 i32[2];
    Int64 i64;
    UInt8  u8[8];
    UInt16 u16[4];
    UInt32 u32[2];
    UInt64 u64;

    char* ToString_i8(char* s, size_t len);
    char* ToString_i16(char* s, size_t len);
    char* ToString_i32(char* s, size_t len);
    char* ToString_i64(char* s, size_t len);

    char* ToString_u8(char* s, size_t len);
    char* ToString_u16(char* s, size_t len);
    char* ToString_u32(char* s, size_t len);
    char* ToString_u64(char* s, size_t len);

    char* ToString_x8(char* s, size_t len);
    char* ToString_x16(char* s, size_t len);
    char* ToString_x32(char* s, size_t len);
    char* ToString_x64(char* s, size_t len);
};
```

148

6.1.1 组合整型加法

第一个示例程序为 MmxAddition，演示了针对组合有符号和无符号整数的 SIMD 加法。示例中也演示了如何使用回绕和饱和运算执行组合整型加法。除了演示组合整型加法，MmxAddition 也演示了一些 C++ 和汇编语言编程的通用技巧，包括联合体和跳转表。示例程序的源代码文件 MmxAddition.cpp 和 MmxAddition_.asm 分别如清单 6-3 和清单 6-4 所示。

清单 6-3 MmxAddition.cpp

```
#include "stdafx.h"
#include "MmxVal.h"

// 下面枚举类型中的枚举成员顺序必须与 MmxAddition_.asm 中定义的一致
enum MmxAddOp : unsigned int
{
    paddb,      // 回绕方式组合字节加法
    paddsb,     // 有符号饱和方式组合字节加法
    paddusb,    // 无符号饱和方式组合字节加法
    paddw,      // 回绕方式组合字加法
    paddsw,     // 有符号饱和方式组合字加法
    paddusw,    // 无符号饱和方式组合字加法
    paddd       // 回绕方式组合双字加法
};

extern "C" MmxVal MmxAdd (MmxVal a, MmxVal b, MmxAddOp op);

void MmxAddBytes(void)
```

```
{
    MmxVal a, b, c;
    char buff [256];

    // 组合字节加法——有符号整型
    a.i8[0] = 50;   b.i8[0] = 33;
    a.i8[1] = 80;   b.i8[1] = 64;
    a.i8[2] = -27;  b.i8[2] = -32;
    a.i8[3] = -70;  b.i8[3] = -80;

    a.i8[4] = -42;  b.i8[4] = 90;
    a.i8[5] = 60;   b.i8[5] = -85;
    a.i8[6] = 64;   b.i8[6] = 90;
    a.i8[7] = 100;  b.i8[7] = -30;

    printf("\n\nPacked byte addition - signed integers\n");
    printf("a:  %s\n", a.ToString_i8(buff, sizeof(buff)));
    printf("b:  %s\n", b.ToString_i8(buff, sizeof(buff)));

    c = MmxAdd (a, b, MmxAddOp::paddb);
    printf("\npaddb results\n");
    printf("c:  %s\n", c.ToString_i8(buff, sizeof(buff)));

    c = MmxAdd_(a, b, MmxAddOp::paddsb);
    printf("\npaddsb results\n");
    printf("c:  %s\n", c.ToString_i8(buff, sizeof(buff)));

    // 组合字节加法——无符号整型
    a.u8[0] = 50;  b.u8[0] = 33;
    a.u8[1] = 80;  b.u8[1] = 64;
    a.u8[2] = 132; b.u8[2] = 130;
    a.u8[3] = 200; b.u8[3] = 180;

    a.u8[4] = 42;  b.u8[4] = 90;
    a.u8[5] = 60;  b.u8[5] = 85;
    a.u8[6] = 140; b.u8[6] = 160;
    a.u8[7] = 10;  b.u8[7] = 14;

    printf("\n\nPacked byte addition - unsigned integers\n");
    printf("a:  %s\n", a.ToString_u8(buff, sizeof(buff)));
    printf("b:  %s\n", b.ToString_u8(buff, sizeof(buff)));

    c = MmxAdd_(a, b, MmxAddOp::paddb);
    printf("\npaddb results\n");
    printf("c:  %s\n", c.ToString_u8(buff, sizeof(buff)));

    c = MmxAdd_(a, b, MmxAddOp::paddusb);
    printf("\npaddusb results\n");
    printf("c:  %s\n", c.ToString_u8(buff, sizeof(buff)));
}

void MmxAddWords(void)
{
    MmxVal a, b, c;
    char buff [256];

    // 组合字加法——有符号整型
    a.i16[0] = 550;     b.i16[0] = 830;
    a.i16[1] = 30000;   b.i16[1] =5000;
    a.i16[2] = -270;    b.i16[2] = -320;
    a.i16[3] = -7000;   b.i16[3] = -32000;
```

[149]

[150]

```
    printf("\n\nPacked word addition - signed integers\n");
    printf("a:  %s\n", a.ToString_i16(buff, sizeof(buff)));
    printf("b:  %s\n", b.ToString_i16(buff, sizeof(buff)));

    c = MmxAdd_(a, b, MmxAddOp::paddw);
    printf("\npaddw results\n");
    printf("c:  %s\n", c.ToString_i16(buff, sizeof(buff)));

    c = MmxAdd_(a, b, MmxAddOp::paddsw);
    printf("\npaddsw results\n");
    printf("c:  %s\n", c.ToString_i16(buff, sizeof(buff)));

    // 组合字加法——无符号解出
    a.u16[0] = 50;      b.u16[0] = 30;
    a.u16[1] = 48000;   b.u16[1] = 20000;
    a.u16[2] = 132;     b.u16[2] = 130;
    a.u16[3] = 10000;   b.u16[3] = 60000;

    printf("\n\nPacked word addition - unsigned integers\n");
    printf("a:  %s\n", a.ToString_u16(buff, sizeof(buff)));
    printf("b:  %s\n", b.ToString_u16(buff, sizeof(buff)));

    c = MmxAdd_(a, b, MmxAddOp::paddw);
    printf("\npaddw results\n");
    printf("c:  %s\n", c.ToString_u16(buff, sizeof(buff)));

    c = MmxAdd_(a, b, MmxAddOp::paddusw);
    printf("\npaddusw results\n");
    printf("c:  %s\n", c.ToString_u16(buff, sizeof(buff)));
}

int _tmain(int argc, _TCHAR* argv[])
{
    MmxAddBytes();
    MmxAddWords();
    return 0;
}
```

清单 6-4 MmxAddition_.asm

```
        .model flat,c
        .code

; extern "C" MmxVal MmxAdd_(MmxVal a, MmxVal b, MmxAddOp add_op);
;
; 描述：大多数调用指令 padd* 的使用
;
; 返回：寄存器对 edx:eax 包含计算结果

MmxAdd_     proc
            push ebp
            mov ebp,esp

; 确保 'add_op' 是有效的
            mov eax,[ebp+24]            ;加载 'add_op'
            cmp eax,AddOpTableCount     ;与表格长度比较
            jae BadAddOp                ;如果 'add_op' 无效，则跳转

; 加载参数并执行指定的指令
            movq mm0,[ebp+8]            ;加载 'a'
            movq mm1,[ebp+16]           ;加载 'b'
            jmp [AddOpTable+eax*4]      ;跳转到指定的 'add_op'
```

```
MmxPaddb:   paddb mm0,mm1                   ;回绕方式组合字节加法
            jmp SaveResult

MmxPaddsb:  paddsb mm0,mm1                  ;有符号饱和方式的组合字节加法
            jmp SaveResult

MmxPaddusb: paddusb mm0,mm1                 ;无符号饱和方式的组合字节加法
            jmp SaveResult

MmxPaddw:   paddw mm0,mm1                   ;回绕方式组合字加法
            jmp SaveResult

MmxPaddsw:  paddsw mm0,mm1                  ;有符号饱和方式的组合字加法
            jmp SaveResult

MmxPaddusw: paddusw mm0,mm1                 ;无符号饱和方式的组合字加法
            jmp SaveResult

MmxPaddd:   paddd mm0,mm1                   ;回绕方式组合双字加法
            jmp SaveResult

BadAddOp:   pxor mm0,mm0                    ;如果 'add_op' 无效，返回 0

; 最终结果存入 edx:eax
SaveResult: movd eax,mm0                    ;eax = mm0 的低位双字部分
            pshufw mm2,mm0,01001110b        ;交换高位双字部分和低位双字部分
            movd edx,mm2                    ;edx:eax = 最终结果
            emms                            ;清除 MMX 状态
            pop ebp
            ret

; 下面表格中标号的顺序必须与 MmxAddition.cpp 中定义的枚举类型一致

        align 4
AddOpTable:
        dword MmxPaddb, MmxPaddsb, MmxPaddusb
        dword MmxPaddw, MmxPaddsw, MmxPaddusw, MmxPaddd
AddOpTableCount equ (S - AddOpTable) / size dword

MmxAdd  endp
        end
```

MmxAddition.cpp（见清单 6-3）的顶部定义了一个 C++ 枚举变量 MmxAddOp，它定义一系列的枚举成员来表示各种组合加法的类型。本例中，C++ 编译器为枚举成员赋以从 0 开始的连续无符号整型值。这些枚举成员的顺序非常重要，并且必须与定义在 MmxAddition_.asm（见清单 6-4）中的跳转表一致。在本节后面，可以更深入地了解此跳转表。在 MmxAddOp 的定义之后，声明了汇编语言函数 MmxAdd_。MmxVal 型参数 a 和 b 表示要相加的两个组合操作数，MmxAddop 型参数 add_op 指定组合加法的类型，计算后的组合数之和将以 MmxVal 类型返回。

源代码文件 MmxAddition.cpp 包含两个主要函数：MmxAddBytes 和 MmxAddWords。这些函数演示如何对组合字节和组合字操作数进行 MMX 加法。函数 MmxAddBytes 首先使用 8 位有符号整型数初始化两个 MmxVal 型变量，接着调用两次 x86 汇编语言函数 MmxAdd_，使用回绕运算和饱和运算执行组合整型加法。调用 MmxAdd_ 之后，执行结果会显示在屏幕上。然后使用类似的一系列 C++ 代码对 8 位无符号整型数执行组合加法。C++ 函数 MmxAddWords 把这种模式应用到 16 位有符号和无符号整型数。

现在，让我们来看看汇编语言文件 MmxAddition_asm（见清单 6-4）。文件底部可以看到之前提过的跳转表 AddOpTable，它包含一系列的汇编语言标志，这些标志定义在函数 MmxAdd_ 中。每个标志之后会执行不同的 MMX 加法指令。跳转表的使用机制之后会进行描述。符号 AddOpTableCount 定义了跳转表的表项数目，用来验证函数参数 add_op 的有效性。语句 align 4 指示汇编编译器将表 AddOpTable 进行双字对齐，以防止未对齐的内存访问。另外，要注意跳转表定义在 proc 和 endp 语句之间。也就是说，为此表格分配的内存在代码段中。很明显，跳转表中没有包含执行指令，这可以解释为什么它放在 ret 指令之后。同时也表示
[153] 跳转表是只读的，对于任何试图对表格进行写入的操作，处理器将会产生一个异常。

在其函数序言之后，函数 MmxAdd_ 将 add_op 的值加载到寄存器 EAX 中。指令 mov eax, [ebp+24] 用来加载 add_op，+24 是因为 MmxVal 类型的参数 a 和 b 按值传递，每个参数需要 8 字节的栈空间。指令 cmp 以及其后的 jae 条件跳转指令用来验证 add_op 值的有效性。

验证了 add_op 的有效性之后，函数 MmxAdd_ 使用指令 movq mm0, [ebp+8]（四字长数据移动）将参数 a 加载到寄存器 MM0。第二条 movq 指令加载参数 b 到 MM1。接下来的指令 jmp [AddOpTable+eax*4] 直接跳转到指定的 MMX 加法指令处。在执行这条指令时，处理器把指令操作数所指定内存位置的内容加载至寄存器 EIP。当前的示例程序中，处理器从内存位置 [AddOpTable+eax*4]（注意寄存器 EAX 中存有 add_op）中读取新的 EIP 值。因为所有在 AddOpTable 中的标记值都对应一条 padd 指令，所以跳转动作后处理器将执行一条 MMX 加法指令。

每条 MMX 加法指令会根据字长要求和运算方式（回绕运算或者饱和运算）来计算 MM0 和 MM1 的和。在每个 MMX 加法之后，会无条件跳转到一段保存计算结果的公用代码。注意，如果 add_op 值无效，会执行 pxor mm0, mm0 指令（逻辑异或），并将寄存器 MM0 清为全 0。

在标号 SaveResult 之后，计算好的组合和被保存起来。注意看保存时使用的指令。Visual C++ 调用约定中，使用寄存器对 EDX:EAX 作为 64 位的返回值，就是说，MM0 的内容必须拷贝到这两个寄存器中。指令 movd eax, mm0（双字数据移动）将 MM0 的低双字部分拷贝到 EAX 中。有些奇怪的是，没有相应的 MMX 指令把 MMX 寄存器的高双字部分拷贝到通用寄存器中。为了绕过这个限制，指令 pshufw mm2, mm0, 01001110b（重排组合字）用来将 MM0 的高双字和低双字部分的数据进行交换。这条指令使用 8 位立即操作数来指定字重排（也就是排序）模式。要从右往左读这个模式，每两位二进制对应一个字在目标操作数中的位置（如位 0 - 1= 字 0，位 2 - 3= 字 1，等等）[1]。立即数中每两位二进制的值表示将源部分源操作数字拷贝到指定的目标操作数字，如图 6-1 所示。在这些指令执行完后，指
[154] 令 movd edx, mm2 把正确的返回值存入寄存器 EDX。

函数结束前调用了指令 emms（清空 MMX 技术状态）。如之前在第 5 章中讨论的，在执行完任何 MMX 指令后，必须调用 emms 指令来恢复 x87 FPU 的正常浮点运算。如果在执行完 MMX 指令后没有使用 emms，直接执行 x87 FPU 指令，x87 FPU 可能会产生异常或者得到一个无效结果。

---

[1] 实际上使用了 8 位立即数来对应操作数。操作数是四字长，分为四个部分，从右往左数为字 0、字 1、字 2、字 3。重排的问题，从源操作数拷贝数据到目标操作数，立即数规定了从源操作数的哪部分拷贝数据。——译者注



图 6-1 pshufw 指令的操作过程

输出 6-1 显示了示例程序 MmxAddition 的执行结果。这些测试用例演示了分别使用回绕方式加法和饱和方式加法进行有符号和无符号整型数运算的各种组合。第一个测试案例中，是字节整型相加。需要注意两种方式的差别，以 80 和 64 相加为例，使用回绕方式加法（paddb results），得到的结果是 -112（溢出了）；使用饱和方式加法（paddsb results），得到的结果为 127。在字节整数范围的另一端，将 -70 和 -80 相加，采用回绕方式相加则得到 106（另一端溢出）；饱和方式相加则得到 -128。输出 6-1 也说明了使用无符号字节整数、有符号字整数和无符号字整数在回绕方式加法和饱和方式加法之间的差异。

输出 6-1 示例程序 MmxAddition

```
Packed byte addition - signed integers
a:    50   80   27   70   42   60   60  100
b:    30   64   32  -80   90   85   90   30

paddb results
c:    80 -112   -59  106   48  -25 -102   70

paddsb results
c:    80  127   -59 -128   48  -25  127   70

Packed byte addition - unsigned integers
a:    50   80  142  200   42   60  140   10
b:    30   64  130  180   90   85  160   14

paddb results
c:    80  144    6  124  132  145   44   24

paddusb results
c:    80  144  255  255  132  145  255   24

Packed word addition - signed integers
a:      550    30000     -270    -7000
b:      830     5000     -320   -32000

paddw results
c:     1380   -30536     -590    26536

paddsw results
c:     1380    32767     -590   -32768

Packed word addition - unsigned integers
a:       50    48000      132    10000
b:       30    20000      130    60000

paddw results
```

155

```
c:      80    2464      262     4464

paddusw results
c:      80    65535     262     65535
```

6.1.2 组合整型移位

下一个示例程序是 MmxShift，它演示了 MMX 移位指令的使用。源文件 MmxShift.cpp 和 MmxShift_.asm 分别列于清单 6-5 和清单 6-6 中。MmxShift 的逻辑组织和程序 MmxAddition 非常类似，所以对其的注释也比较简单。

清单 6-5 MmxShift.cpp

```
#include "stdafx.h"
#include "MmxVal.h"

// 下面枚举类型中的枚举成员顺序必须与MmxShift_.asm中定义的一致

enum MmxShiftOp : unsigned int
{
    psllw,      // 字类型逻辑左移
    psrlw,      // 字类型逻辑右移
    psraw,      // 字类型算术右移
    pslld,      // 双字类型逻辑左移
    psrld,      // 双字类型逻辑右移
    psrad,      // 双字类型算术右移
};

extern "C" bool MmxShift_(MmxVal a, MmxShiftOp shift_op, int count, MmxVal*
b);

void MmxShiftWords(void)
{
    MmxVal a, b;
    int count;
    char buff[256];

    a.u16[0] = 0x1234;
156 a.u16[1] = 0xFF00;
    a.u16[2] = 0x00CC;
    a.u16[3] = 0x8080;
    count = 2;

    MmxShift_(a, MmxShiftOp::psllw, count, &b);
    printf("\nResults for psllw - count = %d\n", count);
    printf("a: %s\n", a.ToString_x16(buff, sizeof(buff)));
    printf("b: %s\n", b.ToString_x16(buff, sizeof(buff)));

    MmxShift_(a, MmxShiftOp::psrlw, count, &b);
    printf("\nResults for psrlw - count = %d\n", count);
    printf("a: %s\n", a.ToString_x16(buff, sizeof(buff)));
    printf("b: %s\n", b.ToString_x16(buff, sizeof(buff)));

    MmxShift_(a, MmxShiftOp::psraw, count, &b);
    printf("\nResults for psraw - count = %d\n", count);
    printf("a: %s\n", a.ToString_x16(buff, sizeof(buff)));
    printf("b: %s\n", b.ToString_x16(buff, sizeof(buff)));
}
```

```
void MmxShiftDwords(void)
{
    MmxVal a, b;
    int count;
    char buff[256];

    a.u32[0] = 0x00010001;
    a.u32[1] = 0x80008000;
    count = 3;

    MmxShift_(a, MmxShiftOp::pslld, count, &b);
    printf("\nResults for pslld - count = %d\n", count);
    printf("a: %s\n", a.ToString_x32(buff, sizeof(buff)));
    printf("b: %s\n", b.ToString_x32(buff, sizeof(buff)));

    MmxShift_(a, MmxShiftOp::psrld, count, &b);
    printf("\nResults for psrld - count = %d\n", count);
    printf("a: %s\n", a.ToString_x32(buff, sizeof(buff)));
    printf("b: %s\n", b.ToString_x32(buff, sizeof(buff)));

    MmxShift_(a, MmxShiftOp::psrad, count, &b);
    printf("\nResults for psrad - count = %d\n", count);
    printf("a: %s\n", a.ToString_x32(buff, sizeof(buff)));
    printf("b: %s\n", b.ToString_x32(buff, sizeof(buff)));
}

int _tmain(int argc, _TCHAR* argv[])
{
    MmxShiftWords();
    MmxShiftDwords();
    return 0;
}
```

157

清单 6-6 MmxShift_.asm

```
        .model flat,c
        .code

; extern "C" bool MmxShift_(MmxVal a, MmxShiftOp shift_op, int count,←
MmxVal* b);
;
; 描述：本函数演示了各类 MMX 移位指令的使用
;
; 返回：0 - 无效的 'shift_op' 参数
;       1 - 成功

MmxShift_   proc
            push ebp
            mov ebp,esp

; 确保参数 'shift_op' 的有效性
            xor eax,eax                 ;设置错误返回码
            mov edx,[ebp+16]            ;加载 'shift_op'
            cmp edx,ShiftOpTableCount   ;与表项数目比较
            jae BadShiftOp              ;如果 'shift_op' 无效，则跳转

; Jump to the specfied shift operation
            mov eax,1                   ;设置成功返回码
            movq mm0,[ebp-8]            ;加载 'a'
            movd mm1,dword ptr [ebp-20] ;加载 'count' 参数的字部分
            jmp [ShiftOpTable+edx*4]
```

```
MmxPsllw:   psllw mm0,mm1                   ;字类型逻辑左移
            jmp SaveResult

MmxPsrlw:   psrlw mm0,mm1                   ;字类型逻辑右移
            jmp SaveResult

MmxPsraw:   psraw mm0,mm1                   ;字类型算术右移
            jmp SaveResult

MmxPslld:   pslld mm0,mm1                   ;双字类型逻辑左移
158         jmp SaveResult

MmxPsrld:   psrld mm0,mm1                   ;双字类型逻辑右移
            jmp SaveResult

MmxPsrad:   psrad mm0,mm1                   ;双字类型算术右移
            jmp SaveResult

BadShiftOp: pxor mm0,mm0                    ;如果 'shift_op' 无效，清零

SaveResult: mov edx,[ebp+24]                ;edx 指向 'b'
            movq [edx],mm0                  ;保存移位结果
            emms                            ;清除 MMX 状态

            pop ebp
            ret

; 下面表格中标号的顺序必须与 MmxShift.cpp 中定义的枚举类型一致

            align 4
ShiftOpTable:
            dword MmxPsllw, MmxPsrlw, MmxPsraw
            dword MmxPslld, MmxPsrld, MmxPsrad
ShiftOpTableCount equ ($ - ShiftOpTable) / size dword

MmxShift_ endp
        end
```

在源文件 MmxShift.cpp（见清单 6-5）的顶部，定义了 C++ 枚举变量 ShiftOp，它的各个成员用来对应各类 MMX 移位操作。接着声明了 x86 汇编语言函数 MmxShift_，其函数原型与 MmxAdd 稍有不同，MmxShift_ 把其结果存储在调用者提供的内存位置，而不是返回 MmxVal 类型数值。

在其序言之后，汇编语言函数 MmxShift_（见清单 6-6）验证参数 shift_op 的有效性。接着用指令 movq mm0, [ebp+8] 把 a 加载到 MMX 寄存器 MM0，用指令 movd mm1, dword ptr [ebp+20] 把移位计数值加载到 MM1 的低双字部分。移位计数值用来指定 a 中的字或双字移动的位数（MMX 移位指令中，移位数目可以由立即数来指定）。指令 jmp [ShiftOpTable+edx*4] 将程序控制转移到指定的 MMX 移位指令。完成移位操作后，结果保存到调用者指定的内存位置。注意，在本例中没有必要使用 pshufw 指令来交换结果中的高低双字部分，因为可以用 movq 指令把整个 64 位值直接存入内存。输出 6-2 显示了示例程
159 序 MmxShift 的执行结果。

输出 6-2 示例程序 MmxShift

```
Results for psllw - count = 2
a: 1234 FF00 00CC 8080
```

```
b: 48D0 FC00 0330 0200

Results for psrlw - count = 2
a: 1234 FF00 00CC 8080
b: 048D 3FC0 0033 2020

Results for psraw - count = 2
a: 1234 FF00 00CC 8080
b: 048D FFC0 0033 E020

Results for pslld - count = 3
a: 00010001 80008000
b: 00080008 00040000

Results for psrld - count = 3
a: 00010001 80008000
b: 00002000 10001000

Results for psrad - count = 3
a: 00010001 80008000
b: 00002000 F0001000
```

6.1.3 组合整型乘法

本节的最后一个示例程序是 MmxMultiplication，演示如何使用组合字型操作数执行有符号整型乘法。源文件 MmxMultiplication.cpp 和 MmxMultiplication_asm 分别列于清单 6-7 和清单 6-8 之中。两个字型整型数（16 位）相乘，总是会生成双字型整型数（32 位）。记住这一点对理解后面的程序很有帮助。

清单 6-7 MmxMultiplication.cpp

```
#include "stdafx.h"
#include "MmxVal.h"

extern "C" void MmxMulSignedWord (MmxVal a, MmxVal b, MmxVal* prod_lo,↵
MmxVal* prod_hi);

int _tmain(int argc, _TCHAR* argv[])
{
    MmxVal a, b, prod_lo, prod_hi;
    char buff[256];

    a.i16[0] = 10;      b.i16[0] = 2000;
    a.i16[1] = 30;      b.i16[1] = -4000;
    a.i16[2] = -50;     b.i16[2] = 6000;
    a.i16[3] = -70;     b.i16[3] = -8000;

    MmxMulSignedWord (a, b, &prod_lo, &prod_hi);

    printf("\nResults for MmxMulSignedWord\n");
    printf("a: %s\n", a.ToString_i16(buff, sizeof(buff)));
    printf("b: %s\n\n", b.ToString_i16(buff, sizeof(buff)));
    printf("prod_lo: %s\n", prod_lo.ToString_i32(buff, sizeof(buff)));
    printf("prod_hi: %s\n", prod_hi.ToString_i32(buff, sizeof(buff)));

    return 0;
}
```

160

清单 6-8 MmxMultiplication_asm

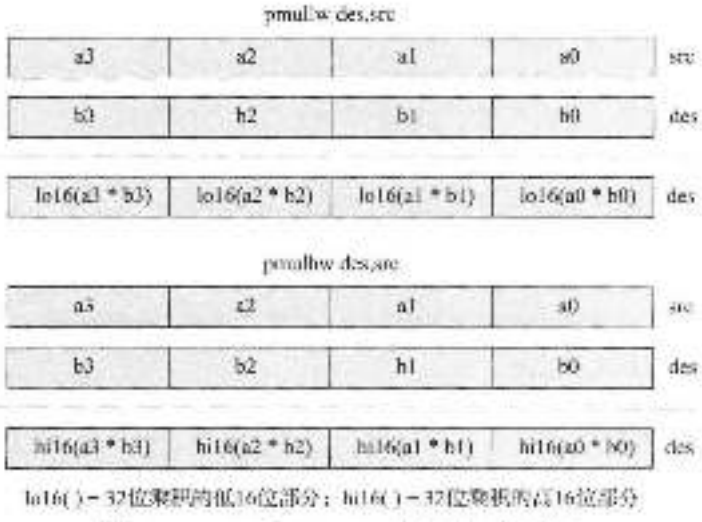
```
        .model flat,c
        .code

; extern "C" void MmxMulSignedWord_(MmxVal a, MmxVal b, MmxVal* prod_lo,↩
MmxVal* prod_hi)
;
; 描述：本函数演示两个组合有符号字型操作数的 SIMD 乘法，得到的双字型积存入调用者指定的存储单元

MmxMulSignedWord_ proc
        push ebp
        mov ebp,esp

; 加载参数 'a' 和 'b'
        movq mm0,[ebp+8]                ;mm0 = 'a'
        movq mm1,[ebp+16]               ;mm1 = 'b'

; 对组合有符号字型实行乘法
        movq mm2,mm0                    ;mm2 = 'a'
        pmullw mm0,mm1                  ;mm0 = 乘积的低位部分
        pmulhw mm1,mm2                  ;mm1 = 乘积的高位部分

; 解组并将乘积的高位低位交错，形成最终的组合双字类型
        movq mm2,mm0                    ;mm2 = product low result
        punpcklwd mm0,mm1               ;mm0 = low dword products
161     punpckhwd mm2,mm1               ;mm2 = high dword products

; 保存组合双字结果
        mov eax,[ebp+24]                ;eax = 指向 'prod_lo'
        mov edx,[ebp+28]                ;edx = 指向 'prod_hi'
        movq [eax],mm0                  ;保存结果的低双字部分
        movq [edx],mm2                  ;保存结果的高双字部分

        pop ebp
        ret
MmxMulSignedWord_ endp
        end
```

我们开始分析汇编语言函数 MmxMulSignedWord_（见清单 6-8）。此函数将两个组合有符号字操作数相乘，并将乘积的组合双字型结果存入内存。在共函数序言之后，MmxVal 类型的参数 a 和 b 分别被加载到寄存器 MM0 和 MM1 中。指令 pmullw mm0, mm1（组合整型乘法，同时保存低位部分结果）将两个组合有符号字型字相乘，把乘积的低 16 位部分存入寄存器 MM0。类似的，指令 pmulhw mm1, mm2（组合整型乘法，同时保存高位部分结果）进行乘法运算并把乘积的高 16 位部分存入寄存器 MM1（注意，在执行指令 pmullw 之前，MM0 已经拷贝到 MM2 了）。图 6-2 展示了 pmullw 和 pmulhw 指令的执行过程。

在进行了上述计算之后，高位结果和低位结果必须重新组织，以形成最终的组合有符号双字型结果。这个任务由指令 punpcklwd（解组低位数据，字型转为双字型）和 punpckhwd（解组高位数据，字型转为双字型）指令完成。这些指令对组合字型值进行解组和交错，如图 6-3 所示。MMX 指令集中也有类似的指令，对组合字节型值和组合双字型值进行解组和交错操作，读者可以在本章稍后部分见到组合字节的例子。

函数 MmxMulSignedWord_ 的最后几条指令把 MM0 和 MM2 中的组合双字型值存入调用者指定的内存位置。源文件 MmxMultiplication.cpp（见清单 6-7）中包含一些用来测试例子的简单代码 MmxMulSignedWord_ 并打印出执行结果。注意在显示文本字符串的时候，对

prod_lo 和 prod_hi 的格式化输出使用了成员函数 MmxVal::ToString_i32，这是因为这两个示例都包含两个双字型。输出 6-3 显示了示例程序 MmxMultiplication 的执行结果。



lo16( ) – 32位乘积的低16位部分；hi16( ) – 32位乘积的高16位部分

图 6-2 pmullw 和 pmulhw 指令的执行过程

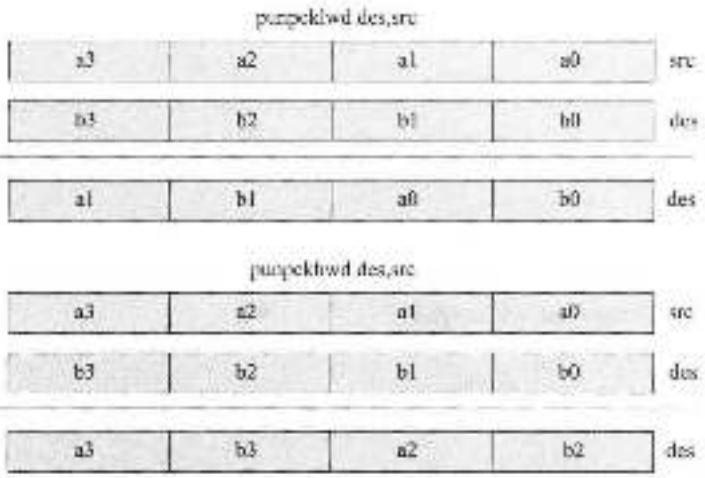


图 6-3 punpcklwd 和 punpckhwd 指令的执行过程

输出 6-3 示例程序 MmxMultiplication

```
Results for MmxMul5igned Word_
a:        10        30       -50       -70
b:      2000     -4000      6000     -8000

prod_lo:         20000       -120000
prod_hi:       -300000        560000
```

162 ↓ 163

6.2 MMX 高级编程

前一节的示例程序旨在对 MMX 编程做一个入门级的介绍。每个程序都包含一个简单的 x86 汇编语言函数，这些函数对联合体 MmxVal 类型的实例执行各种 MMX 指令。对现实

世界的某些应用程序来讲，创建类似于我们前面所看到的函数或许是可接受的。然而，为了充分利用 x86 的 SIMD 架构带来的好处，我们需要编写包含完整算法的函数。这正是本节的焦点。

接下来的几个示例程序演示了利用 MMX 指令集处理 8 位无符号整型数组的方法。第一个程序中，我们将学习如何确定一个数组中的最大值和最小值。该程序有某种实用性，因为在数字图像中经常使用 8 位无符号整型数组来表示位于内存中的图像，且很多图像处理算法需要确定一幅图像中的最小（最暗）和最大（最亮）的像素。第二个程序演示了如何计算一个数组的平均值。这是图像处理领域中另一种很实用的算法。

在本节的前言中我们已经讲过 x86 汇编语言可以被用于提升某些算法的性能，但到目前为止还没有看到任何证据来证实这个说法。这种情况将在本节得到改观。为了量化时间和便于比较，两个示例程序中的关键算法既有 C++ 版本，也有 x86 汇编语言版本。关于时间测量的具体细节将会在后文中讨论。

6.2.1 整数数组处理

本节的示例程序名叫 MmxCalcMinMax，它可以计算一个 8 位无符号整数数组的最大值和最小值。该程序还展示了一些技术，这些技术可被用来衡量一个 x86 汇编语言函数的性能。源代码文件 MmxCalcMinMax.h、MmxCalcMinMax.cpp 和 MmxCalcMinMax_.asm 分别在清单 6-9、清单 6-10 和清单 6-11 中给出。

清单 6-9 MmxCalcMinMax.h

```
#pragma once

#include "MiscDefs.h"

// MmxCalcMinMax.cpp 中定义的函数
extern bool MmxCalcMinMaxCpp(Uint8* x, int n, Uint8* x_min, Uint8* x_max);

// MmxCalcMinMaxTimed.cpp 中定义的函数
extern void MmxCalcMinMaxTimed(void);

// MmxCalcMinMax_.asm 中定义的函数
extern "C" bool MmxCalcMinMax_(Uint8* x, int n, Uint8* x_min, Uint8* x_max);

// 通用常量
const int NUM_ELEMENTS = 0x800000;
const int SRAND_SEED = 14;
```

清单 6-10 MmxCalcMinMax.cpp

```
#include "stdafx.h"
#include "MmxCalcMinMax.h"
#include <stdlib.h>

extern "C" int NMIN = 16;          // 数组元素的最小个数

bool MmxCalcMinMaxCpp(Uint8* x, int n, Uint8* x_min, Uint8* x_max)
{
    if ((n < NMIN) || ((n & 0x0f) != 0))
        return false;

    Uint8 x_min_temp = 0xff;
```

```
    Uint8 x_max_temp = 0;

    for (int i = 0; i < n; i++)
    {
        Uint8 val = *x++;

        if (val < x_min_temp)
            x_min_temp = val;
        else if (val > x_max_temp)
            x_max_temp = val;
    }

    *x_min = x_min_temp;
    *x_max = x_max_temp;
    return true;
}

void MmxCalcMinMax()
{
    const int n = MAX_ELEMENTS;
    Uint8* x = new Uint8[n];

    // 无已知的最小值和最大值初始化测试数组
    srand(SRAND_SEED);
    for (int i = 0; i < n; i++)
        x[i] = (Uint8)((rand() % 240) + 10);

    x[n / 4] = 4;
    x[n / 2] = 252;
    bool rc1, rc2;
    Uint8 x_min1 = 0, x_max1 = 0;
    Uint8 x_min2 = 0, x_max2 = 0;

    rc1 = MmxCalcMinMaxCpp(x, n, &x_min1, &x_max1);
    rc2 = MmxCalcMinMax_(x, n, &x_min2, &x_max2);

    printf("\nResults for MmxCalcMinMax()\n");
    printf("rc1: %d  x_min1: %3u  x_max1: %3u\n", rc1, x_min1, x_max1);
    printf("rc2: %d  x_min2: %3u  x_max2: %3u\n", rc2, x_min2, x_max2);
    delete[] x;
}

int _tmain(int argc, _TCHAR* argv[])
{
    MmxCalcMinMax();
    MmxCalcMinMaxTimed();
    return 0;
}
```

165

清单 6-11 MmxCalcMinMax_.asm

```
        .model flat,c
        .const
StartMinVal qword 0ffffffffffffffffh    ;初始化组合最小值
StartMaxVal qword 0000000000000000h    ;初始化组合最大值
        .code
        extern NMIN:dword               ;数组元素最小长度

; extern "C" bool MmxCalcMinMax_(Uint8* x, int n, Uint8* x_min, Uint8*
x_max);
;
```

```
; 描述：下面的函数计算一个8位无符号单字节数组的最小值和最大值
;
; Returns:      0 = invalid 'n'
;               1 = success

MmxCalcMinMax   proc
        push ebp
        mov ebp,esp

; 确保n是有效的
        xor eax,eax                     ;设置错误返回码
        mov ecx,[ebp+12]                ;ecx = 'n'
        cmp ecx,[MMIN]
        jl Done                         ;如果 n < MMIN 就跳转
        test ecx,0fh
166     jnz Done                        ;如果 n & 0xOf != 0 就跳转

; 初始化
        shr ecx,4                       ;ecx = 16字节块的数量
        mov edx,[ebp+8]                 ;edx = 指向 'x' 的指针
        movq mm4,[StartMinVal]
        movq mm6,mm4                    ;mm6:mm4 当前最小值
        movq mm5,[StartMaxVal]
        movq mm7,mm5                    ;mm7:mm5 当前最大值

; 扫描数组以获取最小值和最大值
@@:     movq mm0,[edx]                  ;mm0 = 组合的8字节
        movq mm1,[edx+8]                ;mm1 = 组合的8字节
        pminub mm6,mm0                  ;mm6 = 更新过的最小值
        pminub mm4,mm1                  ;mm4 = 更新过的最小值
        pmaxub mm7,mm0                  ;mm7 = 更新过的最大值
        pmaxub mm5,mm1                  ;mm5 = 更新过的最大值
        add edx,16                      ;将 edx 指向下一个16字节块
        dec ecx
        jnz @@                          ;如果还有数据就跳转

; 确定最终的最小值
        pminub mm6,mm4                  ;mm6[63:0] = 最终的8个最小值
        pshufw mm0,mm6,00001110b        ;mm0[31:0] = mm6[63:32]
        pminub mm6,mm0                  ;mm6[31:0] = 最终的4个最小值
        pshufw mm0,mm6,00000001b        ;mm0[15:0] = mm6[31:16]
        pminub mm6,mm0                  ;mm6[15:0] = 最终的两个最小值
        pextrw eax,mm6,0                ;ax = 最终的两个最小值
        cmp al,ah
        jbe @F                          ;如果 al <= ah 就跳转
        mov al,ah                       ;al = 最终的最小值
@@:     mov edx,[ebp+16]
        mov [edx],al                    ;保存最终的最小值

; 确定最终的最大值
        pmaxub mm7,mm6                  ;mm7[63:0] = 最终的8个最大值
        pshufw mm0,mm7,00001110b        ;mm0[31:0] = mm7[63:32]
        pmaxub mm7,mm0                  ;mm7[31:0] = 最终的4个最大值
        pshufw mm0,mm7,00000001b        ;mm0[15:0] = mm7[31:16]
        pmaxub mm7,mm0                  ;mm7[15:0] = 最终的两个最大值
        pextrw eax,mm7,0                ;ax = 最终的两个最大值
        cmp al,ah
        jae @F                          ;如果 al >=ah 就跳转
        mov al,ah                       ;al = 最终的最大值
@@:     mov edx,[ebp+20]
167     mov [edx],al                    ;保存最终的最大值
```

```
; 清除 MMX 状态并设置返回码
        emms
        mov eax,1

Done:   pop ebp
        ret
MmxCalcMinMax_  endp
        end
```

文件 MmxCalcMinMax.cpp（清单 6-10）的开头是一个名叫 MmxCalcMinMaxCpp 的函数。这个函数是 C++ 实现的算法，算法该算法对一个 8 位无符号整数数组进行扫描，以找出其中的最大值和最小值。该函数的参数包括指向数组的指针、数组中的元素个数以及用以返回最大值和最小值的指针。算法本身包含一个简单循环，该循环检查每个数组元素以确定其是否比当前的最小值更小或比当前的最大值更大。MmxCalcMinMaxCpp 函数中需要注意的一点是数组的大小必须大于等于 16 且能被 16 整除。这样限制有两个原因。第一，可以简化实现与 MmxCalcMinMaxCpp 相同算法的汇编语言函数的代码。第二，不需要增加额外的代码来处理零头像素块，以提升性能。

MmxCalcMinMax.cpp 文件中还包含一个名叫 MmxCalcMinMax 的函数，它会初始化一个测试数组、调用相应的 C++ 和汇编语言函数并显示结果。除了两个用于确认算法正确性的元素以外，测试数组是用标准库函数 rand 来初始化的。函数 MmxCalcMinMax 是从 _tmain 中调用的。另外要注意的是 _tmain 还调用了一个名为 MmxCalcMinMaxTimed 的函数，其代码用来衡量 MmxCalcMinMax 和 MmxCalcMinMax_ 的性能。本节后续将讨论该函数。

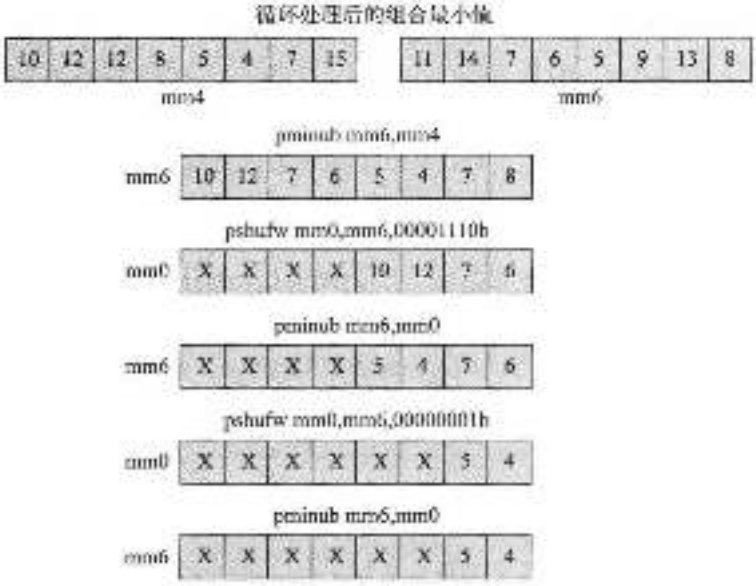
汇编语言函数 MmxCalcMinMax_（清单 6-11）实现了与前面的 C++ 函数相同的算法，但有一个显著的不同，即它使用 8 字节组合数据来处理数组元素，这是 MMX 寄存器中能存放的 8 位整数的最大个数。函数 MmxCalcMinMax_ 的开始会检验参数 n 的大小是否有效。

验证过 n 以后，函数 MmxCalcMinMax_ 进行了一些基本的初始化工作。shr ecx, 4 指令用于计算数组中的 16 字节块的个数。之所以要计算 16 字节块的个数，是因为 MmxCalcMinMax_ 的循环处理中每次迭代会分析 16 个字节，这比每次迭代分析 8 字节稍快。我们将在第 21 章了解到这是为什么。在将 EDX 寄存器初始化为指向数组 x 的指针以后，函数代码装载了寄存器对 MM6:MM4 和 MM7:MM5，以便记录数组中当前的最小值和最大值。注意，与 C++ 算法实现不同的是，汇编语言版本会同时记录 16 个最小值和最大值。在循环处理结束以后，算法会使用前面提到的寄存器对来确定最终的数组最小值和最大值。

MmxCalcMinMax_ 中的循环处理非常简短，每次迭代中，指令 movq mm0,[edx] 和 movq mm1,[edx+8] 将下一个像素块加载到寄存器 MM0 和 MM1 中。然后程序使用两个 pminub（组合无符号字节整数的最小值）指令将该像素块与当前的最小值作对比。pminub 指令对指定寄存器中参与对比的元素做无符号比较，并将较小的元素存放到目标操作数中。当前的最大值会以同样的方式利用两个连续的 pmaxub（组合无符号字节整数的最大值）指令来更新。对数组的处理会一直持续到所有的元素都被检查完。循环处理完成以后，寄存器对 MM6：MM4 和 MM7：MM5 分别包含了组合最小值和最大值。 168

循环处理结束后有一系列指令将 MM6:MM4 中的 16 个值递减到最终的最小值。这是由一系列 pminub、pshufw 和 pextrw（提取字）指令来实现的，如图 6-4 所示。pminub mm6, mm4 指令会将组合最小值的数量由 16 个降为 8 个，这意味着此时的最小值可以放到单个

MMX 寄存器中。接下来 pshufw mm0, mm6, 00001110b 指令将 MM6 中的高位 4 字节的值复制到 MM0 中的低位字节中。然后 pminub mm6, mm0 指令将最小值的数量由 8 降为 4（MM6 和 MM0 中的高位双字是不需要关注的值）。接下来 pshufw/pminub 指令序列将最小值的数量降为 2。最终的最小值由以下方法决定。首先，pextrw eax, mm6, 0 指令将 MM6 中的低位字型元素（由立即操作数 0 指定）复制到寄存器 EAX 中的低位字中；EAX 中的高位字被设为 0。这条指令执行过后，两个倒数第二最小值被存放在寄存器 AH 和 AL 中。最终的最小值使用 x86 的比较和条件跳转指令确定。



注：X 表示可以忽略

169 图 6-4 组合最小值的递减。该图中的数值演示了 MMX 指令的执行过程

函数 MmxCalcMinMax_使用相似的指令序列计算数组中的最大值。唯一的不同是使用了 pmaxub 指令而不是 pminub 以及一个不同的条件跳转。输出 6-4 给出了示例程序 MmxCalcMinMax 的执行结果。

输出 6-4 示例程序 MmxCalcMinMax

```
Results for MmxCalcMinMax()
rc1: 1  x_min1:   4  x_max1: 252
rc2: 1  x_min2:   4  x_max2: 252

Results for MmxCalcMinMaxTimed()
x_min1:   4  x_max1: 252
x_min2:   4  x_max2: 252

Benchmark times saved to file __MmxCalcMinMaxTimed.csv
```

输出 6-4 引用了 CSV（comma-separated value，逗号分隔的数值）文件的内容。该文件包含了性能测量数据。这个文件是由函数 MmxCalcMinMaxTimed 创建的，该函数测量了函数 MmxCalcMinMaxCpp 和 MmxCalcMinMax_的性能。清单 6-12 给出了 MmxCalcMinMaxTimed.cpp 文件的源代码。

清单 6-12 MmxCalcMinMaxedTimed.cpp

```
#include "stdafx.h"
#include "MmxCalcMinMax.h"
#include "ThreadTimer.h"
#include <stdlib.h>

void MmxCalcMinMaxTimed(void)
{
    // 强制当前线程在一个处理器上执行
    ThreadTimer::SetThreadAffinityMask();

    const int n = NUM_ELEMENTS;
    Uint8* x = new Uint8[n];

    // 用已知的最小值和最大值初始化测试数组
    srand(SRAND_SEED);
    for (int i = 0; i < n; i++)
        x[i] = (Uint8)((rand() % 240) + 10);

    x[n / 4] = 4;
    x[n / 2] = 252;

    const int num_it = 100;
    const int num_alg = 2;
    const double et_scale = 1.0e6;

    double et[num_it][num_alg];
    Uint8 x_min1 = 0, x_max1 = 0;
    Uint8 x_min2 = 0, x_max2 = 0;
    ThreadTimer tt;

    for (int i = 0; i < num_it; i++)
    {
        tt.Start();
        MmxCalcMinMaxCpp(x, n, &x_min1, &x_max1);
        tt.Stop();
        et[i][0] = tt.GetElapsedTime() * et_scale;
    }

    for (int i = 0; i < num_it; i++)
    {
        tt.Start();
        MmxCalcMinMax_(x, n, &x_min2, &x_max2);
        tt.Stop();
        et[i][1] = tt.GetElapsedTime() * et_scale;
    }

    const char* fn = "__MmxCalcMinMaxTimed.csv";
    ThreadTimer::SaveElapsedTimeMatrix(fn, (double*)et, num_it, num_alg);

    printf("\nResults for MmxCalcMinMaxTimed()\n");
    printf("x_min1: %3u  x_max1: %3d\n", x_min1, x_max1);
    printf("x_min2: %3u  x_max2: %3d\n", x_min2, x_max2);
    printf("\nBenchmark times saved to file %s\n", fn);
    delete[] x;
}
```

170

函数 MmxCalcMinMaxTimed 依靠一个名叫 ThreadTimer 的 C++ 类来测量一段代码的执行要花多长时间。这个类使用了一些 Windows API 函数（比如 QueryPerformanceCounter 和

QueryPerformanceFrequency）来实现一个简单的软件秒表。成员函数 ThreadTimer::Start 和 ThreadTimer::Stop 用来记录一个系统计数器的数值，而 ThreadTimer::GetElapsedTime 用来计算计数器启动和停止时相差的秒数。类 ThreadTimer 中还包含了一些用于管理进程和线程亲缘性的成员函数。这些成员函数可以为一个进程或线程选择一个特定的 CPU 来执行，这可以增加时间测量的准确性。类 ThreadTimer 的源代码没有在这里给出，只包含在软件包中供下载。表 6-1 中包含了在不同 CPU 上 MmxCalcMinMaxCpp 和 MmxCalcMinMax_ 函数的
[171] 平均执行时间。

表 6-1 MmxCalcMinMax 函数的平均执行时间（单位：微秒）

CPU	MmxCalcMinMaxCpp (C++)	MmxCalcMinMax_ (x86-32 MMX)
Intel Core i7-4770	10813	364
Intel Core i7-4600U	12833	570
Intel Core i3-2310M	20980	950

在输出 6-4 中我们可以看到，运行可执行文件 MmxCalcMinMax.exe 会生成 MmxCalcMinMaxTimed.csv 文件。这个 EXE 文件是由 Visual C++ 编译的，使用了 Release Configuration 选项和代码优化的缺省设置。所有的时间测量都是基于普通的台式机和笔记本电脑进行的，运行的操作系统是 Windows 8.x 或 Windows 7 Service Pack 1。在运行示例程序的可执行文件之前，我们并没有尝试将电脑之间的硬件、软件、操作系统或配置的差异纳入考虑范围之内。

表 6-1 中的数值是使用 Excel 表格函数 TRIMMEAN(array, 0.10) 和 CSV 文件中的执行时间算出来的。对于示例程序 MmxCalcMinMax，基于 x86-32 MMX 实现的最大最小值算法明显超越了 C++ 版本的算法一大截。但我们必须指出示例程序 MmxCalcMinMax 中所看到的性能提升不是特别典型。然而，使用 x86 汇编语言来显著提升运算性能并非偶然，特别是当发挥出处理器的 SIMD 并行计算功能时。在本书的其他章节，我们将看到更多提升算法性能的例子。

与汽车燃料经济学评估和智能手机电池续航时间预测一样，软件性能测试也不是精确的科学。测试的进行以及结果的得出都必须基于某种特定的场景，这种场景对于性能测试本身和目标执行环境都应该是合理的。我们这里使用的测试执行时间的方法是有一定意义的，但测试可能有所不同，取决于测试电脑的配置。在进行性能测试时，我们会发现在绝大多数情况下，执行时间的相对差异比绝对值更有价值。

6.2.2 使用 MMX 和 x87 FPU

本章的最后一个例子名为 MmxCalcMean，它的作用是计算一个 8 位无符号整数数组的算术均值。这个例子还将演示如何对组合无符号整数扩容（size-promote）以及如何在一个包含 MMX 指令的函数中使用 x87 FPU 指令。清单 6-13、清单 6-14 和清单 6-15 列出了这个例
[172] 子的源代码。

清单 6-13 MmxCalcMean.h

```
#pragma once

#include "MiscDefs.h"
```

```
// MmxCalcMean.cpp 中定义的函数
extern bool MmxCalcMeanCpp(const Uint8* x, int n, Uint32* sum_x, double* mean);

// MmxCalcMeanTimed.cpp 中定义的函数
extern void MmxCalcMeanTimed(void);

// MmxCalcMean.asm 中定义的函数
extern "C" bool MmxCalcMean(const Uint8* x, int n, Uint32* sum_x, double* mean);

// 公共常量
const int NUM_ELEMENTS = 0x800000;
const int SRAND_SEED = 25;
```

清单 6-14 MmxCalcMean.cpp

```
#include "stdafx.h"
#include "MmxCalcMean.h"
#include <stdlib.h>

extern "C" int NMIN = 16;              // n 的个数最小值
extern "C" int NMAX = 16777216;        // n 的个数最大值

bool MmxCalcMeanCpp(const Uint8* x, int n, Uint32* sum_x, double* mean_x)
{
    if ((n < NMIN) || (n > NMAX) || ((n & 0x0f) != 0))
        return false;

    Uint32 sum_x_temp = 0;
    for (int i = 0; i < n; i++)
        sum_x_temp += x[i];

    *sum_x = sum_x_temp;
    *mean_x = (double)sum_x_temp / n;
    return true;
}

void MmxCalcMean()
{
    const int n = NUM_ELEMENTS;
    Uint8* x = new Uint8[n];
    srand(SRAND_SEED);
    for (int i = 0; i < n; i++)
        x[i] = rand() % 256;

    bool rc1, rc2;
    Uint32 sum_x1 = 0, sum_x2 = 0;
    double mean_x1 = 0, mean_x2 = 0;

    rc1 = MmxCalcMeanCpp(x, n, &sum_x1, &mean_x1);
    rc2 = MmxCalcMean_(x, n, &sum_x2, &mean_x2);

    printf("\nResults for MmxCalcMean()\n");
    printf("rc1: %d sum_x1: %u mean_x1: %12.6lf\n", rc1, sum_x1, mean_x1);
    printf("rc2: %d sum_x2: %u mean_x2: %12.6lf\n", rc2, sum_x2, mean_x2);
    delete[] x;
}

int _tmain(int argc, _TCHAR* argv[])
{
```

173

```
    MmxCalcMean();
    MmxCalcMeanTimed();
    return 0;
}
```

清单 6-15 MmxCalcMean .asm

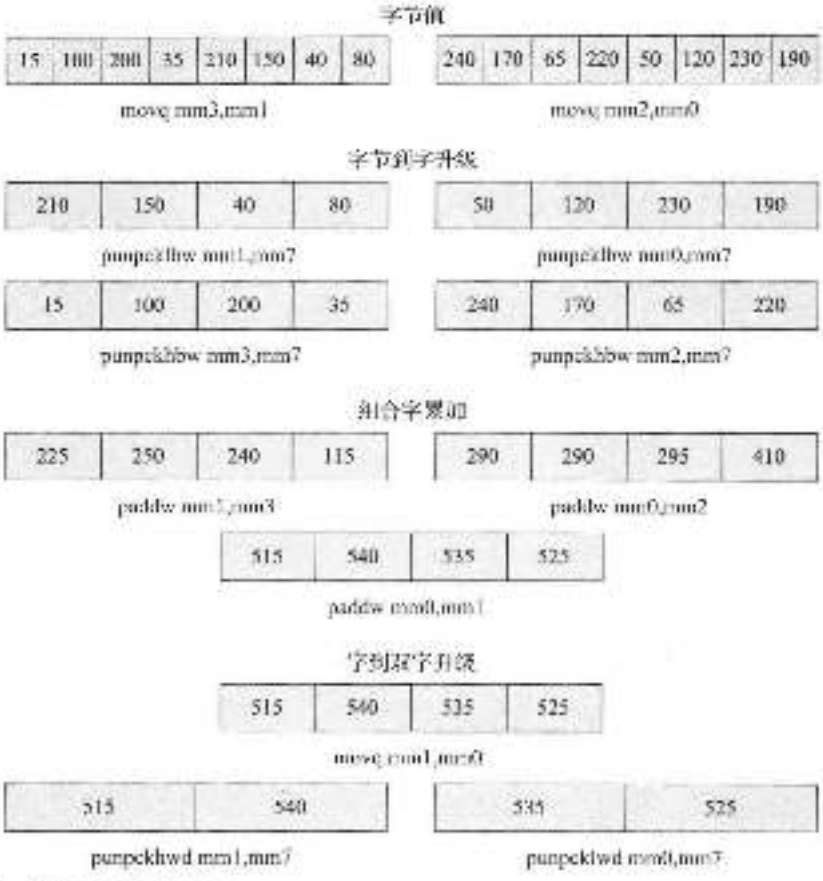
```
        .model flat,c
        .code
        extern NMIN:dword, NMAX:dword       ;数组长度的最小和最大值

; extern "C" bool MmxCalcMean (const UInt8* x, int n, UInt32* sum_x, double* ↩
  mean);
;
; 描述：本函数计算 8 位无符号整数数组的和及均值
;
; 返回值：0 - 'n' 无效
;         1 - 成功

MmxCalcMean_ proc
        push ebp
        mov ebp,esp
        sub esp,8                           ;供 x87 传输的局部变量

; 检查 n 是否有效
        xor eax,eax                         ;设置错误返回码
        mov ecx,[ebp+12]
        cmp ecx,[NMIN]
174     jl Done                             ;如果 n < NMIN 则转
        cmp ecx,[NMAX]
        jg Done                             ;如果 n > NMAX 则转
        test ecx,0fh
        jnz Done                            ;如果 n%16!=0 就转
        shr ecx,4                           ;16 字节块的个数

; 初始化
        mov eax,[ebp+8]                     ;指向数组 'x'
        pxor mm4,mm4
        pxor mm5,mm5                        ;mm5:mm4 - 组合和(4 双字)
        pxor mm7,mm7                        ;mm7 - 用于扩展为组合 0

; 加载下一个 16 字节块
@@:     movq mm0,[eax]
        movq mm1,[eax+8]                        ;mm1:mm0 =16 字节块

; 把数组值从字节扩展到字，然后对字求和
        movq mm2,mm0
        movq mm3,mm1
        punpcklbw mm0,mm7                   ;mm0 = 4 字
        punpcklbw mm1,mm7                   ;mm1 = 4 字
        punpckhbw mm2,mm7                   ;mm2 = 4 字
        punpckhbw mm3,mm7                   ;mm3 = 4 字
        paddw mm0,mm2
        paddw mm1,mm3
        paddw mm0,mm1                       ;mm0 = 组合和(4 字)

; 把组合和扩展为双字，然后更新 mm5:mm4 中的双字和
        movq mm1,mm0
        punpckldq mm0,mm7                   ;mm0 = 组合和(2 双字)
        punpckhwd mm1,mm7                   ;mm1 = 组合和(2 双字)
```

```
        paddd mm4,mm0
        paddd mm5,mm1                   ;mm5:mm4 = 组合和(4 双字)

        add eax,16                      ;eax = 下一个 16 字节块
        dec ecx
        jnz @B                          ;如果没完成则循环

; 计算最终的 sum_x
        paddd mm5,mm4                   ;mm5 = 组合和(2 双字)
        pshufw mm6,mm5,00001110b        ;mm6[31:0] = mm5[63:32]
        paddd mm6,mm5                   ;mm6[31:0] = 最终的 sum_x
        movd eax,mm6                    ;eax = sum_x
        emms                            ;清空 mmx 状态

; 计算均值
        mov dword ptr [ebp-8],eax       ;把 sum_x 保存为 64 位
        mov dword ptr [ebp-4],0
        fild qword ptr [ebp-8]          ;加载 sum_x
        fild dword ptr [ebp+12]         ;加载 n
        fdivp                           ;mean = sum_x / n

        mov edx,[ebp+20]
        fstp real8 ptr [edx]            ;保存均值
        mov edx,[ebp+16]
        mov [edx],eax                   ;保存 sum_x
        mov eax,1                       ;设置返回码

Done:   mov esp,ebp
        pop ebp
        ret
MmxCalcMean  endp
        end
```

MmxCalcMean 的总体结构与 MmxCalcMinMax 很类似。在 MmxCalcMean.cpp(清单 6-14)的起始部分是一个名为 MmxCalcMeanCpp 的函数，它的作用是计算 8 位无符号整数数组的均值。这个数组的最大大小是有限制的，以防止对数组元素求和时发生计算溢出。函数 MmxCalcMean 创建一个测试数组，调用 MmxCalcMeanCpp 和 MmxCalcMean_并显示结果。示例程序 MmxCalcMean 还包含一个性能测试函数，名为 MmxCalcMeanTimed(没有显示源代码)，它用来测量 MmxCalcMeanCpp 和 MmxCalcMean_的执行时间。

与 C++ 版本类似，使用 x86 汇编语言编写的 MmxCalcMean_函数(清单 6-15)首先检查数组的大小。然后这个函数建立一个处理循环，对数组的元素求和。在计算数组的和时，处理循环执行两次容量升级，以防止计算溢出。图 6-5 演示了这个过程。首先，我们使用 punpcklbw 和 punpckhbw 指令(注意源操作数 MM7 包含的是全 0)把 16 个数组元素从无符号字节升级为无符号字。然后使用一系列 paddw 指令把这些值累加起来。在执行完最后一个 paddw 指令后，寄存器 MM0 包含四个中间结果。然后我们把和值升级为双字，并累加为组合双字，放在寄存器对 MM5:MM4 中。

在完成累加循环之后，我们使用两条 paddd 指令和一条 pshufw 指令来计算最终的和值 sum_x，然后把这个值复制到寄存器 EAX。在计算最终的均值之前，程序执行了一条 emms 指令，以便让 x87 FPU 返回到正常工作状态。而后把 sum_x 的值以 64 位带符号整数的形式保存到栈上的局部内存(回忆 x87 FPU 不支持无符号整数操作数，也不支持与 x86 通用寄存器之间的数据传递)。最后，使用 x87 FPU 计算最终的均值。



注：mm7 = 0x0000000000000000

图 6-5 处理循环数组元素扩容

输出 6-5 显示了运行示例程序 MmxCalcMean 的结果。表 6-2 中列出了计算这个数组均值的算法的 C++ 版本和汇编语言版本的性能测试结果。

177

输出 6-5 示例程序 MmxCalcMean

```
Results for MmxCalcMean()
rc1: 1 sum_x1: 1069226624 mean_x1:   127.461746
rc2: 1 sum_x2: 1069226624 mean_x2:   127.461746

Results for MmxCalcMeanTimed()
sum1: 1069226624  mean1:   127.461746
sum2: 1069226624  mean2:   127.461746

Benchmark times saved to file __MmxCalcMeanTimed.csv
```

表 6-2 MmxCalcMean 函数的平均执行时间（毫秒）

CPU	MmxCalcMeanCpp (C++)	MmxCalcMean_ (x86-32 MMX)
Intel Core i7-4770	1750	843
Intel Core i7-4600U	2074	995
Intel Core i3-2310M	4184	1704

6.3 总结

本章先介绍了如何使用 MMX 指令集做一些基本的 SIMD 算术运算和移位操作。然后，我们分析了很多个示例程序，展示了使用 MMX 技术实现有实际意义的整数数组处理算法的性能优势。这些示例程序也阐释了一些 MMX 关键指令的使用方法，包括 pshufw、punpcklbw、punpckhbw、punpcklwd 和 punpckhwd。

为了充分发挥 SIMD 架构的优势，很多时候软件开发者必须"忘记"以前建立的编码习惯、技术和结构。要设计和实现出高效的 SIMD 算法常常需要把以前的编程思路做个急转弯。在学习使用 SIMD 架构时，非常有经验的程序员很可能也要把编程思想做些改变。在接下来的四章中，我们将继续探索 x86 的 SIMD 计算平台——x86 SSE。 178

流式 SIMD 扩展

在第 5 章和第 6 章中，我们介绍了 MMX 技术，这是 x86 平台的第一个 SIMD 增强。在本章中，我们将探索 MMX 技术的后续技术。x86 的流式 SIMD 扩展（x86-SSE）是对用以提升 x86 平台的 SIMD 计算能力的一系列架构化增强技术的总称。x86-SSE 针对组合浮点（packed floating-point）数据类型增加了新的寄存器和指令，并且对 MMX 技术的整数 SIMD 处理能力做了扩展。

本章先对 x86-SSE 做一个概要的概览，包括它的不同版本和能力。接下来对执行环境进行详细介绍，重点是 x86-SSE 的寄存器、支持的数据类型以及控制状态机制。然后将介绍 x86-SSE 的一些基本处理技术，目的是帮助大家理解这个平台的计算能力。最后一节将对 x86-SSE 指令集做总体介绍。

本章的阐述和讨论集中在 x86-32 执行环境下的 x86-SSE。如果你对 x86-64 环境下的 x86-SSE 编程感兴趣，那么你需要先充分理解本章的基础知识，然后再阅读第 19 章的内容。

7.1 x86-SSE 概览

流式 SIMD 扩展的最初版本称为 SSE，是随奔腾 III 处理器引入的。SSE 增加了用于对组合单精度浮点数执行 SIMD 运算的寄存器和指令。SSE 中也包含对标量单精度浮点数进行算术运算的新指令。奔腾 IV 处理器引入了 SSE 的升级版本，称为 SSE2，把 SSE 的单精度浮点能力扩展为双精度浮点运算（包括组合和标量）。SSE2 还包含了更多的用于组合整数的 SIMD 处理资源。在这一章的后面，我们将详细介绍 SSE 和 SSE2 支持的组合和标量数据类型。

在 SSE2 之后，还有很多对 x86-SSE 的增强。SSE3 和 SSSE3（Supplemental SSE3）包含了一系列新的指令，用以对组合浮点数和组合整数操作数分别执行 SIMD 水平（或相邻元素）算术计算。这些扩展还包含了很多数据传输指令，有的是为了提高性能，有的是为了增加编程的灵活度。SSE4.1 包含了用来做高级 SIMD 运算的新指令，包括点积和数据混合

（data blending）。它还增加了新的数据插入、数据提取和浮点舍入指令。SSE4.2 是对 x86-SSE 的最后一次扩展，它为 x86 平台添加了 SIMD 文本串处理能力。表 7-1 归纳了 x86-SSE 的演进过程，其中使用了 SPFP 和 DPFP 缩略语，分别用来代表单精度浮点（single-precision floating-point）和双精度浮点（double-precision floating-point）。

表 7-1 x86-SSE 的演进过程

版本	数据类型	关键功能和增强
SSE	组合 SPFP 标量 SPFP	使用组合 SPFP 的 SIMD 计算 使用 SPFP 的标量计算 新增缓存控制指令 内存对齐指令

（续）

版本	数据类型	关键功能和增强
SSE2	组合 SPFP、DPFP 标量 SPFP、DPFP 组合整数	使用组合 SPFP 和 DPFP 的 SIMD 计算 使用 SPFP 和 DPFP 的标量计算 使用组合字节的 SIMD 处理 更多的高速缓存控制指令
SSE3	与 SSE2 相同	使用组合 SPFP 和 DPFP 操作数水平加法和减法 增加的数据传输指令
SSSE3	与 SSE2 相同	使用组合整数操作数水平加法和减法 使用组合整数高增强 SIMD 处理指令
SSE4.1	与 SSE2 相同	SPFP 和 DPFP 点积 SPFP 和 DPFP 混合指令 XMM 寄存器插入和提取指令 组合整数混合指令
SSE4.2	与 SSE2 相同 组合文本字符串	组合文本字符串指令 CRC 加速指令

本章的后续内容和第 8 ~ 11 章将详细探讨表 7-1 中所列的关键功能和增强。提醒一下，本书使用 x86-SSE 这个术语来指代普遍适用于 x86 平台各种版本的流式 SIMD 扩展的通用能力，当讨论针对某个特定 SIMD 增强的内容时，我们会使用表 7-1 中的简称。

7.2 x86-SSE 执行环境

本节将介绍 x86-SSE 的执行环境。我们会从寄存器谈起，然后讨论 x86-SSE 支持的组合和标量数据类型。另外还将介绍 x86-SSE 的控制 - 状态寄存器，这个寄存器的作用是配置 x86-SSE 的处理选项和检测错误的条件。我们假定你在阅读这一节的内容之前已经对第 5 章所描述的 MMX 技术有了基本的理解。 [180]

7.2.1 x86-SSE 寄存器组

x86-SSE 向 x86 平台新增了八个 128 位宽的寄存器，如图 7-1 所示。这些寄存器被命名为 XMM0 ~ XMM7，可以用来对标量和组合单精度浮点数做计算。在支持 SSE2 的处理器中，这些 XMM 寄存器支持对标量和组合双精度浮点数做运算。SSE2 还支持使用 XMM 寄存器对组合整数做各种 SIMD 运算。

与 MMX 寄存器不同，XMM 寄存器并不是 x87 FPU 的别名。这意味着程序在 x86-SSE 和 x87-FPU 指令间切换时不需要保存和恢复任何状态信息。XMM 寄存器是可以直接寻址的，不是使用基于栈的架构。可以使用第 1 章所描述的任何寻址模式在 XMM 寄存器和内存之间传输数据。不可以使用 XMM 寄存器来寻址内存中的操作数。

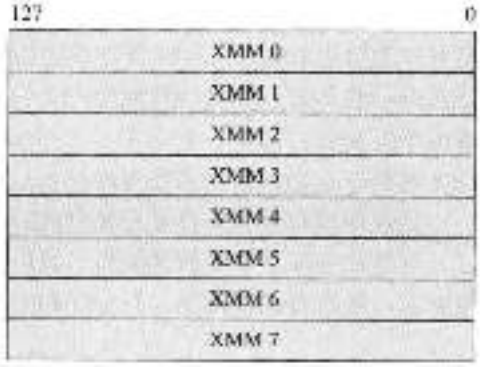


图 7-1 x86-32 模式下的 x86-SSE 寄存器组

7.2.2 x86-SSE 数据类型

x86-SSE 支持很多种组合和标量数据类型，如表 7-2 所示。一个 128 位宽的 XMM 寄存器或者内存单元可以容纳 4 个单精度浮点数或者两个双精度浮点数。当处理组合整数时，一个 128 位宽的操作数能够容纳 16 个字节、8 个字、4 个双字或者两个四字（quadword）。也可以使用 XMM 寄存器的低位双字和四字对单一的标量单精度和双精度浮点数做计算。图
[181] 7-2 列出了 x86-SSE 支持的数据类型。

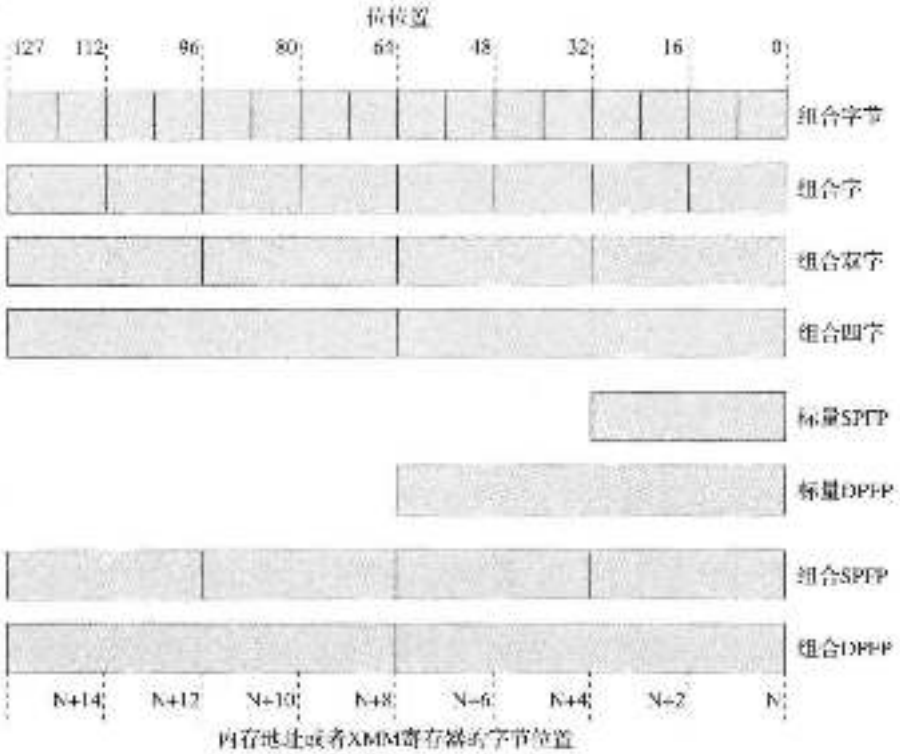


图 7-2 x86-SSE 支持的数据类型

当引用内存中的操作数时，几乎所有使用 128 位宽操作数的 x86-SSE 指令都要求操作数要正确对齐。这意味着一个组合整数或者组合浮点数在内存中的地址必须被 16 整除。这个规则只对很少的一部分 x86-SSE 移动指令有例外，这些指令是特意设计的，可以在没有正确对齐的组合数据和 XMM 寄存器之间传输数据。如果普通的 x86-SSE 指令试图访问没有对齐的内存操作数，那么处理器会产生一个异常。值得特别提醒的是，这样的内存对齐要求既适用于汇编语言函数，也适用于 C++ 函数。在第 9 章和第 10 章，我们会介绍几种在 Visual C++ 函数中指定数据对齐规则的技术。

内存中没有对齐的标量单精度和双精度浮点数可以复制到 XMM 寄存器，反之亦然。不过，与其他 x86 多字节数据类似，我们强烈推荐把标量浮点数做正确对齐，以避免可能的性
[182] 能损失。最后要说明的是，与 x87-FPU 不同，x86-SSE 不支持组合 BCD 数据类型。

7.2.3 x86-SSE 的控制 – 状态寄存器

x86-SSE 执行环境配备了一个 32 位的控制 – 状态寄存器。这个寄存器名为 MXCSR，

它包含了一系列控制标志，供程序来指定浮点运算和处理异常的选项。它还包含了一组状态标志，程序可以通过检查这些状态标志检测 x86-SSE 的浮点错误情况。图 7-3 显示了 MXCSR 寄存器中的各个位的组织顺序；表 7-2 描述了每个二进制位域的作用。



图 7-3 x86-SSE MXCSR（控制和状态寄存器）

表 7-2 x86-SSE MXCSR 控制和状态寄存器的位域

位	名 称	描 述
IE	无效操作标志	x86-SSE 浮点无效操作错误标志
DE	非规格化（denormal）标志	x86-SSE 浮点非规格化错误标志
ZE	除零标志	x86-SSE 浮点除零错误标志
OE	溢出标志	x86-SSE 浮点溢出错误标志
UE	下溢标志	x86-SSE 浮点下溢错误标志
PE	精度标志	x86-SSE 浮点精度错误标志
DAZ	非规格化为 0	当设置为 1 时，计算前把非规格化操作数强制转化为 0
IM	无效操作掩码	x86-SSE 浮点无效操作错误异常掩码
DM	非规格化掩码	x86-SSE 浮点非规格化错误异常掩码
ZM	除零掩码	x86-SSE 浮点除零错误异常掩码
OM	溢出掩码	x86-SSE 浮点溢出错误异常掩码
UM	下溢掩码	x86-SSE 浮点下溢错误异常掩码
PM	精度掩码	x86-SSE 浮点精度错误异常掩码
RC	舍入控制	指定 x86-SSE 浮点计算结果的舍入方法。有效的选项包括舍入到最近（00b），向下舍入到 $-\infty$（01b），向上舍入到 $+\infty$（10b）和向 0 舍入或者截断（11b）
FZ	冲刷为 0	当设置为 1 时，如果发生了下溢错误而且屏蔽了下溢异常，那么就把结果强制冲刷（flush）为 0

183

应用程序可以修改 MXCSR 寄存器的任意控制标志或者状态位，以满足特定 SIMD 浮点处理的需要。如果试图向保留位写入非零值，那么处理器会产生异常。错误情况发生后，处理器不会自动清除 MXCSR 的状态标志；应用程序必须手动复位。可以使用 ldmxcsr（加载 MXCSR 寄存器）或者 fxrstor（恢复 x87 FPU、MMX、XMM 和 MXCSR 状态）指令来修改 MXCSR 寄存器的控制标志和状态位。

当错误情况发生时，处理器会把 MXCSR 的错误标志设置为 1。把 MXCSR.DAZ 控制标志设置为 1 有助于提高程序的性能，前提是把非规格化数值舍入为 0 是可以接受的。类似的，当浮点下溢错误较多时，也可以使用 MXCSR.FZ 控制标志来提高计算速度。使用这些控制标志选项的不利之处是这样做与 IEEE-754 浮点标准不兼容。

7.3 x86-SSE 处理技术

x86 处理器使用多种不同的技术来处理 x86-SSE 数据类型。大多数针对组合数数操作数

的 x86-SSE SIMD 操作的执行过程与 MMX 技术是相同的，只是 SSE 使用 128 位宽的操作数而不是 64 位。举例来说，图 7-4 描述了针对无符号整数组合字节、字和双字的 x86-SSE SIMD 加法过程。x86-SSE 也支持针对组合单精度和双精度浮点数据类型的 SIMD 算术计算。图 7-5 和图 7-6 分别显示了针对组合单精度和双精度浮点数的一些常用 x86-SSE SIMD 计算过程。

[184]

x86-SSE组合无符号字节加法

	90	10	220	170	60	120	15	105	25	130	50	75	95	200	225	85	src
+	80	240	15	20	75	105	50	80	60	15	60	70	125	30	10	95	des
	170	250	235	190	135	225	65	185	85	145	110	145	220	230	235	180	des

x86-SSE组合无符号字加法

	1700	2300	6000	500	22000	15500	12000	18000	src
+	9000	80	300	900	8500	16750	32700	25000	des
	10700	2380	6300	1400	30500	32250	44700	43000	des

x86-SSE组合无符号双字加法

	120000	275000	65000	420500	src
+	800000	1800	300750	70500	des
	920000	276800	365750	491000	des

图 7-4 针对组合无符号整数的 x86-SSE 加法

x86-SSE组合单精度浮点加法

	12.0	37.25	100.875	0.125	src
+	88.0	98.5	−50.625	−0.375	des
	100.0	135.75	50.25	−0.25	des

x86-SSE组合单精度浮点乘法

	1.5	100.25	1000.0	−50.125	src
×	−200.25	3.625	250.875	−40.75	des
	−300.375	363.40625	250875.0	2042.59375	des

[185]

图 7-5 针对组合单精度浮点数的 x86-SSE 算术运算



图 7-6 针对组合双精度浮点数的 x86-SSE 算术运算

也可以用 x86-SSE 来做标量浮点算术计算，既可针对单精度浮点数，也可针对双精度浮点数。图 7-7 演示了几个例子。值得注意的是，当进行 x86-SSE 标量浮点计算时，处理器只会修改目标操作数的低位元素，高位元素不受影响。x86-SSE 的标量浮点计算能力经常被作为替代 x87-FPU 的一种现代方法。改用新技术的好处是不仅获得了更好的性能，还有可以直接寻址的寄存器。可以直接寻址寄存器让高级语言的编译器产生更高效的机器码，这也大大降低了编写做标量浮点运算的汇编语言函数的难度。 186



图 7-7 针对标量浮点数的 x86-SSE 算术运算

x86-SSE 还支持水平算术计算。水平算术计算是对组合数据类型的相邻元素进行计算。图 7-8 演示了针对单精度浮点操作数的水平加法和双精度浮点操作数的水平减法。x86-SSE 还支持针对组合字和组合双字的整数做水平加法和减法。水平运算经常被用来把组合数据操作数中包含的中间结果化简为一个结果。 187

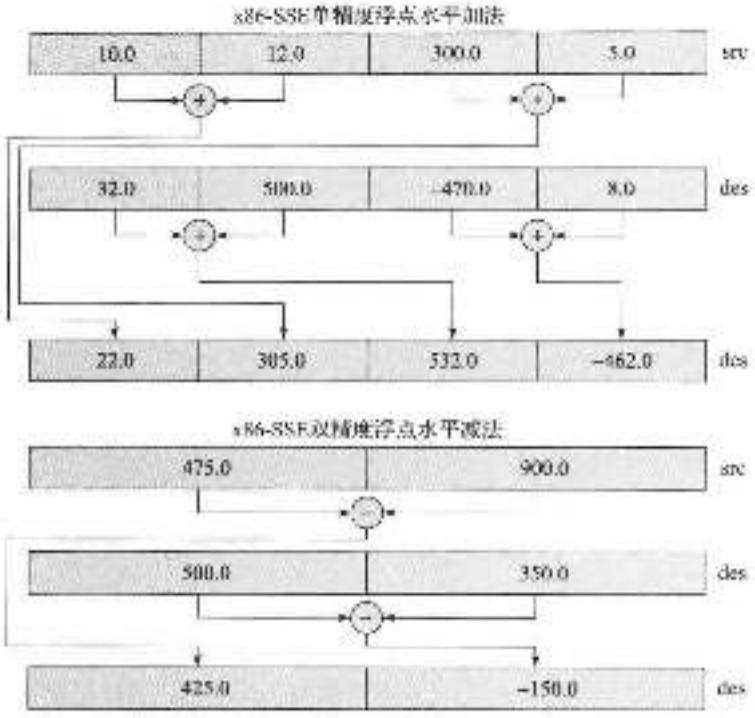


图 7-8 针对单精度和双精度浮点数的 x86-SSE 水平加法和减法

7.4 x86-SSE 指令集概览

本节将对 x86-SSE 指令集做一个简单浏览，采用的格式与本书中对其他指令集的描述是相同的。我们把 x86-SSE 指令集划分为如下这些组：

- 标量浮点数据传输
- 标量浮点算术运算
- 标量浮点比较
- 标量浮点转换
- 组合浮点数据传输
- 组合浮点算术运算
- 组合浮点比较
- 组合浮点转换
- 组合浮点重排（shuffle）和解组（unpack）
- 组合浮点插入和提取
- 组合浮点混合
- 组合浮点逻辑
- 组合整数扩展
- 组合整数数据传输
- 组合整数算术运算
- 组合整数比较
- 组合整数转换

188

- 组合整数重排（shuffle）和解组（unpack）
- 组合整数插入和提取
- 组合整数混合
- 组合整数移位
- 文本字符串处理
- 非临时数据传输和缓存控制
- 其他

除非特别说明，x86-SSE 指令的源操作数既可以是 XMM 寄存器，也可以是内存位置。位于内存中的组合 128 位宽源操作数必须按 16 字节边界对齐，但那些特别声明支持非对齐数据传输的指令除外。除了数据传输指令外，x86-SSE 指令的目标操作数必须是 XMM 寄存器。

应用程序可以交替使用 x86-SSE 指令和 MMX 指令，这对于代码维护和项目移植来说是有利的。使用 MMX 指令集的基于 x86-SSE 的程序在使用 x87 FPU 指令前需要执行 emms 指令。对于新的开发项目来说不鼓励把 x86-SSE 指令和 MMX 指令交替使用。

在下面的指令描述中，我们使用 SPFP 和 DPFP 来分别指代单精度浮点和双精度浮点。大多数 x86-SSE 浮点指令助记符用两个字母来表示操作数类型，包括 ps（组合 SPFP）、pd（组合 DPSP）、ss（标量 SPFP）和 sd（标量 DPFP）。AMD 和 Intel（英特尔）公司的参考手册里可以找到关于 x86-SSE 指令用法的更多信息，包括有效操作数和可能的异常等。第 8 章到第 11 章的示例程序也包含了某些 x86-SSE 指令的解释。 189

7.4.1 标量浮点数据传输

标量浮点数据传输组的指令用来在 XMM 寄存器和内存单元之间传输标量 SPFP 和 DPFP 值，如表 7-3 所示。

表 7-3 x86-SSE 标量浮点数据传输指令

助记符	描 述	版 本
movss movsd	在两个 XMM 寄存器之间或者内存位置和 XMM 寄存器之间复制标量浮点数	SSE/SSE2

7.4.2 标量浮点算术运算

标量浮点算术运算组的指令对标量操作数进行基本的算术运算。可以使用这些指令来替代 x87 FPU 指令或者一起使用。对于这组的所有指令，源操作数可以是内存位置或者 XMM 寄存器，而目标操作数必须是 XMM 寄存器。表 7-4 归纳了标量浮点算术运算指令。

表 7-4 x86-SSE 标量浮点算术运算指令

助记符	描 述	版 本
addss addsd	对指定操作数做标量加法	SSE/SSE2
subss subsd	对指定操作数做标量减法，源操作数指定减数，目标操作数指定被减数	SSE/SSE2
mulss mulsd	对指定操作数做标量乘法	SSE/SSE2

（续）

助记符	描　述	版　本
divss divsd	对指定操作数做标量除法，源操作数指定除数，目标操作数指定被除数	SSE/SSE2
sqrtss sqrtsd	计算指定操作数的平方根	SSE/SSE2
maxss maxsd	比较源操作数和目标操作数，并把较大的值保存到目标操作数	SSE/SSE2
minss minsd	比较源操作数和目标操作数，并把较小的值保存到目标操作数	SSE/SSE2
roundss roundsd	使用立即操作数指定的舍入方法对标量浮点数做舍入	SSE4.1
rcpss	计算指定操作数的近似倒数	SSE
rsqrtss	计算指定操作数的近似倒数平方根	SSE

[190]

7.4.3 标量浮点比较

标量浮点比较组的指令用于比较两个标量浮点数，表 7-5 归纳了这组指令的用法。

表 7-5　x86-SSE 标量浮点比较指令

助记符	描　述	版　本
cmpss cmpsd	比较两个标量浮点值，使用立即操作数指定比较操作符。比较的结果被保存到目标操作数（所有 1 表示真，所有 0 表示假）	SSE/SSE2
comiss comisd	对两个标量浮点数做有序比较，使用 EFLAGS.ZF、EFLAGS.PF 和 EFLAGS.CF 报告结果	SSE/SSE2
ucomiss ucomisd	对两个标量浮点数做无序比较，使用 EFLAGS.ZF、EFLAGS.PF 和 EFLAGS.CF 报告结果	SSE/SSE2

7.4.4 标量浮点转换

标量浮点转换组的指令用来把标量 SPFP 值转换为 DPFP 值，反之亦然。也支持转换为双字整数或者从双字整数转换。表 7-6 归纳了这组指令的用法。

[191]

表 7-6　x86-SSE 标量浮点转换指令

助记符	描　述	版　本
cvtsi2ss cvtsi2sd	把带符号双字整数转换为浮点数，源操作数可以是内存位置或者通用寄存器，目标操作数必须是 XMM 寄存器	SSE/SSE2
cvtss2si cvtsd2si	把浮点数转换为双字整数，源操作数可以是内存位置或者 XMM 寄存器，目标操作数必须是通用寄存器	SSE/SSE2
cvttss2si cvttsd2si	使用截断方法把浮点数转换为双字整数，源操作数可以是内存位置或者 XMM 寄存器，目标操作数必须是通用寄存器	SSE/SSE2
cvtss2sd	把 SPFP 值转换为 DPFP 值，源操作数可以是内存位置或者 XMM 寄存器，目标操作数必须是 XMM 寄存器	SSE2
cvtsd2ss	把 DPFP 值转换为 SPFP 值，源操作数可以是内存位置或者 XMM 寄存器，目标操作数必须是 XMM 寄存器	SSE2

7.4.5 组合浮点数据传输

组合浮点数据传输组的指令用来在 XMM 寄存器和内存单元之间或者两个 XMM 寄存器之间复制组合浮点数据，表 7-7 归纳了这一组指令的用法。

表 7-7 x86-SSE 组合浮点数据传输指令

助记符	描 述	版 本
movaps movapd	在两个 XMM 寄存器之间或者内存单元和 XMM 寄存器之间复制组合 SPFP/DPFP 值	SSE/SSE2
movups movupd	在两个 XMM 寄存器之间或者非对齐内存单元和 XMM 寄存器之间复制组合 SPFP/DPFP 值	SSE/SSE2
movlps movlpd	把一个组合 SPFP/DPFP 值的低位四字从内存复制到 XMM 寄存器，反之亦可。如果目标是 XMM 寄存器，那么高位四字不受影响	SSE/SSE2
movhps movhpd	把一个组合 SPFP/DPFP 值的高位四字从内存复制到 XMM 寄存器，反之亦可。如果目标是 XMM 寄存器，那么低位的四字不受影响	SSE/SSE2
movlhps	把组合 SPFP 源操作数的低位四字复制到目标操作数的高位四字，源和目标操作数必须都是 XMM 寄存器	SSE
movhlps	把组合 SPFP 源操作数的高位四字复制到目标操作数的低位四字，源和目标操作数必须都是 XMM 寄存器	SSE
movsldup	把源操作数中每个四字的低位 SPFP 值复制到目标操作数的相同位置，再把目标操作数中每个四字的低位 SPFP 值复制到高 32 位	SSE3
movshdup	把源操作数的每个四字的高位 SPFP 值复制到目标操作数的相同位置。每个目标操作数四字的高位 SPFP 值再被复制到低 32 位	SSE3
movddup	把源操作数的低位 DPFP 值复制到目标操作数的低位四字和高位四字	SSE3
movmskps movmskpd	提取源操作数中每个组合 SPFP/DPFP 数据元素的符号位，并把其存到通用寄存器的低 4 位中，高位被置为 0	SSE/SSE2

7.4.6 组合浮点算术运算

组合浮点算术运算组的指令可以对组合单精度浮点操作数或者双精度浮点操作数做基本的算术运算。表 7-8 列出了这些指令。

表 7-8 x86-SSE 组合浮点算术运算指令

助记符	描 述	版 本
addps addpd	对源和目标操作数中对应数据元素实施组合浮点加法	SSE/SSE2
subps subpd	对源和目标操作数中的数据元素实施组合浮点减法，源操作数包含减数，目标操作数包含被减数	SSE/SSE2
mulps mulpd	对源和目标操作数中的数据元素实施组合浮点乘法	SSE/SSE2
divps divpd	对源和目标操作数中的数据元素实施组合浮点除法，源操作数包含除数，目标操作数包含被除数	SSE/SSE2

（续）

助记符	描　述	版　本
sqrtps sqrtpd	计算源操作数中压缩浮点数据元素的平方根	SSE/SSE2
maxps maxpd	比较源和目标操作数中的浮点数据元素，把较大的值保存到目标操作数	SSE/SSE2
minps minpd	比较源和目标操作数中的浮点数据元素，把较小的值保存到目标操作数	SSE/SSE2
rcpps	计算每个浮点元素的近似倒数	SSE/SSE2
rsqrtps	计算每个浮点元素的近似倒数平方根	SSE
addsubps addsubpd	对高数字号的浮点元素做加法，对低数字号的浮点元素做减法	SSE3
dpps dppd	对组合浮点元素做条件乘法，再做加法，这条指令用来计算点积	SSE4.1
roundps roundpd	使用立即数指定的舍入模式对组合浮点数据元素做舍入操作	SSE4.1
haddps haddpd	对源和目标操作数中包含的相邻浮点数据元素做加法	SSE3
hsubps hsubpd	对源和目标操作数中包含的相邻浮点数据元素做减法	SSE3

7.4.7 组合浮点比较

组合浮点比较组的指令用于对组合浮点数中的数据元素做比较操作，表7-9归纳了这组指令的用法。

表7-9 x86-SSE组合浮点比较指令

助记符	描　述	版　本
cmpps cmppd	使用指定的立即数对源操作数和目标操作数中的浮点数据元素，比较的结果被保存到目标操作数（全为1表示真，全为0表示假）	SSE/SSE2

7.4.8 组合浮点转换

组合浮点转换组的指令可以把组合浮点操作数的数据元素从一种数据类型转换为另一种数据类型，表7-10归纳了这一组指令的用法。

表7-10 x86-SSE组合浮点转换指令

助记符	描　述	版　本
cvtpi2ps cvtpi2pd	把两个组合带符号双字整数转换为两个组合浮点数，源操作数可以是内存位置或者MMX寄存器，目标操作数必须是XMM寄存器	SSE/SSE2
cvtps2pi cvtpd2pi	把两个组合浮点数转换为两个组合带符号双字整数，源操作数可以是内存位置或者XMM寄存器，目标操作数必须是MMX寄存器	SSE/SSE2
cvttps2pi cvttpd2pi	使用截断方法把两个组合浮点数转换为两个带符号双字整数，源操作数可以是内存位置或者XMM寄存器，目标操作数必须是MMX寄存器	SSE/SSE2

（续）

助记符	描　述	版　本
cvtdq2ps	把四个组合带符号双字整数转换为四个组合单精度浮点数	SSE2
cvtdq2pd	把两个组合带符号双字整数转换为两个组合双精度浮点数	SSE2
cvtps2dq	把四个组合单精度浮点数转换为四个带符号双字整数	SSE2
cvttps2dq	使用截断作为舍入模式把四个组合单精度浮点数转换为四个组合带符号双字整数	SSE2
cvtpd2dq	把两个组合双精度浮点数转换为两个组合带符号双字整数	SSE2
cvttpd2pq	使用截断作为舍入模式把两个组合双精度浮点数转换为两个带符号双字整数	SSE2
cvtps2pd	把两个组合单精度浮点数转换为两个组合双精度浮点数	SSE2
cvtpd2ps	把两个组合双精度浮点数转换为两个组合单精度浮点数	SSE2

7.4.9　组合浮点重排和解组

组合浮点重排和解组组的指令用来对组合浮点操作数中的数据元素重新排序。表 7-11 列出了这一组的指令。 196

表 7-11　x86-SSE 组合浮点重排和解组指令

助记符	描　述	版　本
shufps shufpd	把源/目标操作数中的指定元素移动到目标操作数，使用 8 位的立即数操作数来指定要移动的元素	SSE/SSE2
unpcklps unpcklpd	解组（unpack）并交错源和目标操作数的低位元素，结果放到目标操作数	SSE/SSE2
unpckhps unpckhpd	解组（unpack）并交错源和目标操作数的高位元素，结果放到目标操作数	SSE/SSE2

7.4.10　组合浮点插入和提取

组合浮点插入和提取组的指令可以向组合单精度浮点操作数插入元素或者从中提取元素。表 7-12 归纳了这组的指令。

表 7-12　x86-SSE 组合浮点插入和提取指令

助记符	描　述	版　本
insertps	从源操作数复制出一个 SPFP 值，插入到目标操作数。源操作数可以是内存位置或者 XMM 寄存器，目标操作数必须是 XMM 寄存器。目标操作数元素是通过立即数操作数指定的	SSE4.1
extractps	从源操作数中提取出一个 SPFP 元素，并把它复制到目标操作数，源操作数必须是 XMM 寄存器，目标操作数可以是内存位置或者通用寄存器。要提取元素的位置是通过立即数操作数指定的	SSE4.1

7.4.11　组合浮点混合

组合浮点混合组的指令用来对组合浮点数据做条件复制或者融合（merge），表 7-13 列出了这一组的指令。 197

表 7-13 x86-SSE 组合浮点混合指令

助记符	描　述	版　本
blendps blendpd	从源和目标操作数中按条件复制浮点元素到目标操作数，使用立即数操作数来指定要复制的特定元素	SSE4.1
blendvps blendvpd	从源和目标操作数中按条件复制浮点元素到目标操作数，使用XMM0中的掩码值指定要复制的特定元素	SSE4.1

7.4.12 组合浮点逻辑

组合浮点逻辑组的指令用来对组合浮点操作数做按位逻辑运算，表 7-14 列出了这一组的指令。

表 7-14 x86-SSE 组合浮点逻辑指令

助记符	描　述	版　本
andps andpd	对指定组合浮点操作数中的数据元素做按位逻辑与（AND）	SSE/SSE2
andnps andnpd	对目标操作数做按位逻辑取反（NOT），然后结果和目标操作数做逻辑与（AND）	SSE/SSE2
orps orpd	对指定组合浮点操作数中的数据元素做按位逻辑或（OR）	SSE/SSE2
xorps xorpd	对指定组合浮点操作数中的数据元素做按位逻辑异或	SSE/SSE2

7.4.13 组合整数扩展

SSE2 从两个方面对 x86 平台的组合整数处理能力做了扩展。首先，所有 MMX 技术中定义的组合整数指令（pshufw 除外）都可以使用 XMM 寄存器和 128 位宽的内存单元作操作数。其次，SSE2 和后续的 x86 SIMD 扩展包含了很多新的组合整数指令，它们要求至少一个操作数是 XMM 寄存器或者 128 位宽的内存单元。下面几节将概述这些指令。

198

7.4.14 组合整数数据传输

组合整数数据传输组的指令用来在 XMM 寄存器和内存单元或者两个 XMM 寄存器之间移动组合整数。这一组也包含了用以在 XMM 寄存器和 MMX 寄存器之间执行数据移动的指令。表 7-15 列出了这一组的指令。

表 7-15 x86-SSE 组合整数数据传输指令

助记符	描　述	版　本
movdqa	把符合内存对齐规则的两个四字从内存复制到 XMM 寄存器，反之亦可。这条指令也可以执行 XMM 寄存器之间的传输	SSE2
movdqu	把不符合内存对齐规则的两个四字从内存复制到 XMM 寄存器，反之亦可	SSE2
movq2dq	把 MMX 寄存器的内容复制到 XMM 寄存器的低位四字。这条指令会导致从 x87 FPU 到 MMX 模式的切换	SSE2
movdq2q	把 XMM 寄存器的低位四字复制到 MMX 寄存器。这条指令会导致从 x87 FPU 到 MMX 模式的切换	SSE2

7.4.15 组合整数算术运算

组合整数算术运算组的指令用来对组合整数操作数做算术运算，表 7-16 描述这一组的指令。

表 7-16 x86-SSE 组合整数算术运算指令

助记符	描 述	版 本
pmulld	在源和目标操作数之间做组合带符号乘法。每个乘积的低位双字被保存到目标操作数	SSE4.1
pmuldq	把源和目标操作数的第一个和第三个带符号双字做乘法，四字乘积被保存到目标操作数	SSE4.1
pminub pminuw pminud	比较两个组合无符号整数，并把较小的数据元素保存到目标操作数。源操作数可以是内存位置或者寄存器，目标操作数必须是寄存器	SSE2（pminub）SSE4.1
pminsb pminsw pminsd	比较两个组合带符号整数，并把较小的数据元素保存到目标操作数。源操作数可以是内存位置或者寄存器，目标操作数必须是寄存器	SSE2（pminsw）SSE4.1
pmaxub pmaxuw pmaxud	比较两个组合无符号整数，并把较大的数据元素保存到目标操作数。源操作数可以是内存位置或者寄存器，目标操作数必须是寄存器	SSE2（pmaxub）SSE4.1
pmaxsb pmaxsw pmaxsd	比较两个组合带符号整数，并把较大的数据元素保存到目标操作数。源操作数可以是内存位置或者寄存器，目标操作数必须是寄存器	SSE2（pmaxsw）SSE4.1

199

7.4.16 组合整数比较

组合整数比较组的指令用来对两个组合整数做比较，表 7-17 归纳了这组指令的用法。

表 7-17 x86-SSE 组合整数比较指令

助记符	描 述	版 本
pcmpeqb pcmpeqw pcmpeqd pcmpeqq	一个元素一个元素地比较两个组合整数是否相等。如果源和目标数据元素相等，那么就把目标操作数的对应元素设置为全 1，否者设置为全 0	SSE2 SSE4.1（pcmpeqq）
pcmpgtb pcmpgtw pcmpgtd pcmpgtq	一个元素一个元素地比较两个组合带符号整数，寻找较大者。如果目标元素较大，那么就把目标操作数的对应元素设置为全 1，否者设置为全 0	SSE2 SSE4.2（pcmpgtq）

200

7.4.17 组合整数转换

组合整数数据转换组的指令可以把组合整数从一种类型转换到另一种类型。这组指令包含多种变体，既有符号扩展，又有 0 扩展。表 7-18 列出了这一组的指令。

表 7-18 x86-SSE 组合整数转换指令

助记符	描 述	版 本
packusswb packusdw	使用无符号饱和算法把源操作数和目标操作数中的 n 个组合无符号整数转换为 2 * n 个组合无符号整数	SSE2（packuswb） SSE4.1(packusdw)
pmovsxbw pmovsxbd pmovsxbq	对源操作数的低位带符号字节（signed-byte）整数做符号扩展，并把这些值复制到目标操作数	SSE4.1
pmovsxwd pmovsxwq	对源操作数的低位带符号字（signed-word）整数做符号扩展，并把这些值复制到目标操作数	SSE4.1
pmovsxdq	对源操作数的低位带符号双字（signed doubleword）整数做符号扩展，并把得到的四字值复制到目标操作数	SSE4.1
pmovzxbw pmovzxbd pmovzxbq	对源操作数的低位无符号字节（unsigned-byte）整数做 0 扩展，并把这些值复制到目标操作数	SSE4.1
pmovzxwd pmovzxwq	对源操作数的低位无符号字（unsigned-word）整数做 0 扩展，并把这些值复制到目标操作数	SSE4.1
pmovzxdq	对源操作数的两个低位无符号双字（unsigned doubleword）整数做 0 扩展，并把得到的四字值复制到目标操作数	SSE4.1

201

7.4.18 组合整数重排和解组

组合整数重排和解组的指令对组合整数数据做重新排序和解组操作。对于这一组的所有指令，源操作数可以是 XMM 寄存器或者内存位置，目标操作数必须是 XMM 寄存器。表 7-19 列出了这一组的指令。

表 7-19 x86-SSE 组合整数重排和解组指令

助记符	描 述	版 本
pshufd	使用立即数操作数指定的排序模式把源操作数的双字复制到目标操作数	SSE2
pshuflw	使用立即数操作数指定的排序模式把源操作数的低位字复制到目标操作数的低位字	SSE2
pshufhw	使用立即数操作数指定的排序模式把源操作数的高位字复制到目标操作数的高位字	SSE2
punpcklqdq	把源操作数的低位四字复制到目标操作数的高位四字，目标操作数的低位四字保持不变	SSE2
punpckhqdq	把源操作数的高位四字复制到目标操作数的高位四字，它还把目标操作数的高位四字复制到目标操作数的低位四字	SSE2

7.4.19 组合整数插入和提取

组合整数插入和提取组包括字节、字、双字和四字插入和提取指令。可以用这些指令把通用寄存器中的低位值复制到 XMM 寄存器，反之亦可。表 7-20 列出了这一组的指令。

202

表 7-20 x86-SSE 组合整数插入和提取指令

助记符	描 述	版 本
pinsrb pinsrw pinsrd	把一个整数从源操作数复制到目标操作数，这个整数在目标操作数中的位置是 1 立即数操作数指定的。源操作数可以是内存位置或者通用寄存器。目标操作数必须是 XMM 寄存器	SSE2（pinsrw） SSE4.1

（续）

助记符	描　述	版　本
pextrb pextrw pextrd	把一个整数从源操作数复制到目标操作数。这个整数在源操作数中的位置是由立即数操作数指定的。源操作数必须是 XMM 寄存器，目标操作数可以是内存位置或者通用寄存器	SSE2（pintrw） SSE4.1

7.4.20　组合整数混合

组合整数混合组的指令用来对组合整数进行条件复制或者融合，表 7-21 列出了这一组的指令。

表 7-21　x86-SSE 组合整数混合指令

助记符	描　述	版　本
pblendw	从源和目标操作数中按条件复制字（word）值到目标操作数，使用立即数掩码值来指定要复制的特定字值	SSE4.1
pblendvb	从源和目标操作数中按条件复制字节值到目标操作数，使用寄存器 XMM0 中的掩码值指定要复制的特定字节值	SSE4.1

7.4.21　组合整数移位

组合整数移位组的指令用来对 XMM 寄存器中的数值进行面向字节的逻辑移位。表 7-22 描述了这一组的指令。

表 7-22　x86-SSE 组合整数移位指令

助记符	描　述	版　本
pslldq	对 XMM 寄存器中的组合整数值数向字节的左移，用 0 填充低字字节。移动的位数是由立即数操作数来指定的	SSE2
psrldq	对 XMM 寄存器中的组合整数值数面向字节的右移，用 0 填充高位字节。移动的位数是由立即数操作数来指定的	SSE2

203

7.4.22　文本字符串处理

文本字符串处理组的指令用来对显式长度或者隐式长度的字符串做串操作。所谓显式长度文本字符串，就是预先知道其长度的文本字符串。而隐式长度文本字符串的长度必须通过搜索结束字符来决定。这一组的指令可以做 SIMD 文本字符串比较或者长度计算，也可以使用它们来优化文本字符串搜索和替换算法。表 7-23 归纳了文本字符串处理指令。

表 7-23　x86-SSE 文本字符串处理指令

助记符	描　述	版　本
pcmpestri	对两个显式长度的文本字符串做组合比较，在 ECX 中返回索引结果	SSE4.2
pcmpestrm	对两个显式长度的文本字符串做组合比较，在 XMM0 中返回掩码结果	SSE4.2
pcmpistri	对两个隐式长度的文本字符串做组合比较，在 ECX 中返回索引结果	SSE4.2
pcmpistrm	对两个隐式长度的文本字符串做组合比较，在 XMM0 中返回掩码结果	SSE4.2

7.4.23 非临时数据传输和缓存控制

非临时数据传输和缓存控制组的指令可以执行非临时内存存储、高速缓存预取（prefetching）及冲洗以及内存加载和存储设卡（fencing）。非临时内存存储通知处理器可以把要写的数据直接写到内存中，而不要存储到处理器的高速缓存。对于某些应用（比如语音和视频编码），这么做可以提高高速缓存的效率，因为这样消除了高速缓存杂波。高速缓存预取指令通知处理器加载一条缓存数据线到指定级别的缓存供将来使用。内存设卡（fence）指令把所有悬而未决的内存加载和存储类作序列化，这可以提高针对多处理器设计的“生产者－消费者”算法的性能。

值得说明的是，非临时内存存储、高速缓存预取和内存设卡都是给处理器的暗示（hint）；处理器可以选择利用这些暗示，也可以选择忽略这些暗示。另外值得注意的是，使用这些暗示并不影响处理器的程序执行状态，包括寄存器和状态标志。没有必要也不可能编写额外代码来断定处理器是否接受了某个暗示。表 7-24 列出了非临时数据传输和缓存控制
204 指令。

表 7-24 x86-SSE 非临时数据传输和缓存控制指令

助记符	描 述	版 本
movnti	使用非临时暗示把通用寄存器的内容复制到内存	SSE2
movntdq	使用非临时暗示把 XMM 寄存器的内容复制到内存	SSE2
maskmovdqu	使用非临时暗示把 XMM 寄存器的字节有条件地复制到内存。包含在第二个 XMM 寄存器中的掩码值指定要复制的字节。EDI 寄存器指向目标内存的位置	SSE2
movntdqa	使用非临时暗示把某个内存的双四字加载到 XMM 寄存器	SSE4.1
sfence	序列化以前发出的所有内存存储操作	SSE
lfence	序列化以前发出的所有内存加载操作	SSE2
mfence	序列化以前发出的所有内存加载和内存存储操作	SSE2
prefetchH	给处理器发生暗示，可以从主内存加载数据到高速缓存。该操作数指定主内存的位置。其中与 H 相占位符，用来指定高速缓存暗示的类型，有效的选项有 T0（暂时数据，所有级别的高速缓存）、T1（暂时数据，L1 高速缓存）、T2（暂时数据，L2 高速缓存）或者 NTA（对齐的非临时数据，所有级别的高速缓存）	SSE
clflush	冲洗一个高速缓存线，使其无效。该操作数指定高速缓存线的内存位置	SSE2

7.4.24 其他

其他组包含的指令用来操纵 x86-SSE 的控制－状态寄存器 MXCSR，还有几个针对特定
205 算法的加速指令。表 7-25 列出了这些指令。

表 7-25 x86-SSE 其他指令

助记符	描 述	版 本
ldmxcsr	从内存加载 x86-SSE 控制－状态寄存器 MXCSR	SSE
stmxcsr	把 x86-SSE 控制－状态寄存器 MXCSR 保存到内存	SSE
fxsave	把当前的 x87 FPU、MMX 技术、XMM 和 MXCSR 的状态保存到内存。设计这条指令的目的是支持操作系统的任务实换，但应用程序也可以使用它	SSE
fxrstor	从内存加载 x87 FPU、MMX 技术、XMM 和 MXCSR 的状态。设计这条指令的目的是支持操作系统的任务实换，但应用程序也可以使用它	SSE

（续）

助记符	描　述	版　本
crc32	主要计算 32 位循环冗余检查（CRC）。源操作数可以是内存位置或者通用寄存器。目标操作数必须是 32 位通用寄存器，并包含前一个中间结果。用于计算 CRC 的多项式（polynomial）是 0x11EDC6F41	SSE4.2
popcnt	计数（count）设操作数中设置为 1 的二进制位数。源操作数可以是内存位置或者通用寄存器，目标操作数必须是通用寄存器	SSE4.2

7.5 总结

本章介绍了 x886-SSE 的基础知识，包括寄存器集、支持的组合和标量数据类型以及基本的处理技术。我们还浏览了 x86-SSE 的指令集，目的是对 x86-SSE 的计算能力获得比较全面的理解。x86-SSE 执行环境的广泛数据类型和指令集使它成为一种非常强大的编程工具，广泛适用于很多类算法问题。在接下来的四章中，你将把本章学到的 x86-SSE 知识投入应用，去分析各类用释本章材料的示例程序。 206

x86-SSE 编程——标量浮点

在前面的章节里我们探索了 x86-SSE 的计算资源，包括数据类型和指令集。在本章，我们将学习如何使用 x86-SSE 指令集对标量浮点进行算术运算。本章内容分为两节：8.1 节演示基本的 x86-SSE 标量浮点运算，包括简单的算术计算、比较和类型转换；8.2 节通过一组示例程序来演示高级的 x86-SSE 标量浮点技术。

本章所有的示例程序都需要在支持 SSE2 的处理器上运行（包括几乎所有 2003 年以后面世的 AMD 和 Intel 处理器）。在本章及后续的 x86-SSE 相关的章节里，在每个汇编语言源代码的头部都有说明运行该程序所需 SSE 的最低版本。附录 C 列出了一些免费工具，这些工具可以帮助你检测你的 PC 机中处理器和操作系统所能支持的 x86-SSE 版本。

8.1 标量浮点运算基础

x86-SSE 的标量浮点处理能力给程序员提供了替代 x87 FPU 的一种更好选择。x86-SSE 直接访问 XMM 寄存器的能力意味着可以更加容易地进行基本的标量浮点运算。浮点型的加减乘除运算变得跟使用通用寄存器进行整数算术运算十分相似。每条指令需要一个源和目标操作数，运算的方式都是 des = des ★ src（其中★代表一种具体的运算）。由于我们不用担心寄存器栈的状态，所以程序的可读性得到了明显的改进，本章的示例程序展示了这一点。

8.1.1 标量浮点算术运算

大家看到的第一个 x86-SSE 标量浮点示例程序是 SseScalarFloatingPointArithmetic。这个程序展示了如何使用 x86-SSE 标量浮点指令集进行基本的算术运算，包括单精度和双精度数值的运算。清单 8-1 和清单 8-2 分别给出了相应的 C++ 和 x86-32 汇编语言源代码。

清单 8-1 SseScalarFloatingPointArithmetic.cpp

```
#include "stdafx.h"
#define _USE_MATH_DEFINES
#include <math.h>

extern "C" void SseSfpArithmeticFloat_(float a, float b, float c[8]);
extern "C" void SseSfpArithmeticDouble_(double a, double b, double c[8]);

void SseSpfArithmeticFloat(void)
{
    float a = 2.5f;
    float b = -7.625f;
    float c[8];

    SseSfpArithmeticFloat_(a, b, c);
    printf("\nResults for SseSfpArithmeticFloat_()\n");
    printf("  a:              %.6f\n", a);
    printf("  b:              %.6f\n", b);
    printf("  add:            %.6f\n", c[0]);
    printf("  sub:            %.6f\n", c[1]);
```

```
    printf("  mul:           %.6f\n", c[2]);
    printf("  div:           %.6f\n", c[3]);
    printf("  min:           %.6f\n", c[4]);
    printf("  max:           %.6f\n", c[5]);
    printf("  fabs(b):       %.6f\n", c[6]);
    printf("  sqrt(fabs(b)): %.6f\n", c[7]);
}

void SseSpfArithmeticDouble(void)
{
    double a = M_PI;
    double b = M_E;
    double c[8];

    SseSfpArithmeticDouble_(a, b, c);
    printf("\nResults for SseSfpArithmeticDouble_()\n");
    printf("  a:             %.14f\n", a);
    printf("  b:             %.14f\n", b);
    printf("  add:           %.14f\n", c[0]);
    printf("  sub:           %.14f\n", c[1]);
    printf("  mul:           %.14f\n", c[2]);
    printf("  div:           %.14f\n", c[3]);
    printf("  min:           %.14f\n", c[4]);
    printf("  max:           %.14f\n", c[5]);
    printf("  fabs(b):       %.14f\n", c[6]);
    printf("  sqrt(fabs(b)): %.14f\n", c[7]);
}

int _tmain(int argc, _TCHAR* argv[])
{
    SseSpfArithmeticFloat();
    SseSpfArithmeticDouble();
}
```

208

清单 8-2 SseScalarFloatingPointArithmetic_.asm

```
        .model flat,c
        .const

; 浮点绝对值的掩码值，16 位对齐
AbsMaskFloat    dword 7fffffffh,0ffffffffh,0ffffffffh,0ffffffffh
AbsMaskDouble   qword 7fffffffffffffffh,0ffffffffffffffffh
        .code

; extern "C" void SseSfpArithmeticFloat_(float a, float b, float c[8])
;
; 函数说明：本函数展示了使用标量单精度浮点值的基本的算术运算
;
; SSE 版本：SSE

SseSfpArithmeticFloat_ proc
        push ebp
        mov ebp,esp

; 载入参数值
        movss xmm0,real4 ptr [ebp+8]        ;xmm0 = a
        movss xmm1,real4 ptr [ebp+12]       ;xmm1 = b
        mov eax,[ebp+16]                    ;eax = c

; 进行单精度算术运算
        movss xmm2,xmm0
```

```
        addss xmm2,xmm1                 ;xmm2 = a + b
        movss real4 ptr [eax],xmm2

        movss xmm3,xmm0
        subss xmm3,xmm1                 ;xmm3 = a - b
        movss real4 ptr [eax+4],xmm3

        movss xmm4,xmm0
        mulss xmm4,xmm1                 ;xmm4 = a * b
        movss real4 ptr [eax+8],xmm4

        movss xmm5,xmm0
        divss xmm5,xmm1                 ;xmm5 = a / b
        movss real4 ptr [eax+12],xmm5

        movss xmm6,xmm0
        minss xmm6,xmm1                 ;xmm6 = min(a, b)
        movss real4 ptr [eax+16],xmm6

        movss xmm7,xmm0
        maxss xmm7,xmm1                 ;xmm7 = max(a, b)
        movss real4 ptr [eax+20],xmm7

        andps xmm1,[AbsMaskFloat]       ;xmm1 = fabs(b)
        movss real4 ptr [eax+24],xmm1

        sqrtss xmm0,xmm1                ;xmm0 = sqrt(fabs(b))
        movss real4 ptr [eax+28],xmm0

        pop ebp
        ret
SseSfpArithmeticFloat_ endp

; extern "C" void SseSfpArithmeticDouble_(double a, double b, double c[8])
;
; 函数说明：下面的函数演示如何对标量双精度浮点数进行基本的算术运算
;
; SSE 版本：SSE2

SseSfpArithmeticDouble_ proc
        push ebp
        mov ebp,esp

; 载入参数值
        movsd xmm0,real8 ptr [ebp+8]    ;xmm0 = a
        movsd xmm1,real8 ptr [ebp+16]   ;xmm1 = b
        mov eax,[ebp+24]                ;eax = c

; 进行双精度算术运算
        movsd xmm2,xmm0
        addsd xmm2,xmm1                 ;xmm2 = a + b
        movsd real8 ptr [eax],xmm2

        movsd xmm3,xmm0
        subsd xmm3,xmm1                 ;xmm3 = a - b
        movsd real8 ptr [eax+8],xmm3

        movsd xmm4,xmm0
        mulsd xmm4,xmm1                 ;xmm4 = a * b
        movsd real8 ptr [eax+16],xmm4

        movsd xmm5,xmm0
```

```
        divsd xmm5,xmm1                 ;xmm5 = a / b
        movsd real8 ptr [eax+24],xmm5

        movsd xmm6,xmm0
        minsd xmm6,xmm1                 ;xmm6 = min(a, b)
        movsd real8 ptr [eax+32],xmm6

        movsd xmm7,xmm0
        maxsd xmm7,xmm1                 ;xmm7 = max(a, b)
        movsd real8 ptr [eax+40],xmm7

        andpd xmm1,[AbsMaskDouble]      ;xmm1 = fabs(b)
        movsd real8 ptr [eax+48],xmm1

        sqrtsd xmm0,xmm1                ;xmm0 = sqrt(fabs(b))
        movsd real8 ptr [eax+56],xmm0

        pop ebp
        ret
SseSfpArithmeticDouble_ endp
        end
```

示例程序 SseScalarFloatingPointArithmetic 的 C++ 源代码（见清单 8-1）包含一个 SseSfpArithmeticFloat 函数。这个函数初始化一组单精度浮点变量，然后调用一个 x86 汇编语言函数进行一系列基本的浮点算术运算。这个汇编语言函数把各个算术运算的结果保存在调用者提供的数组里，然后把这些结果打印出来。与之类似，SseSfpArithmeticDouble 函数进行同样的操作，不同的是它操作的是双精度数。

汇编语言文件 SseScalarFloatingPointArithmetic .asm（见清单 8-2）包含一个 SseSfpArithmeticFloat_ 函数。这个函数演示了 x86-SSE 标量单精度浮点指令的使用方法。在函数序言之后，指令 movss xmm0, real4 ptr [ebp+8]（Move Scalar Single-Precision Floating-Point Value，移动标量单精度浮点值）把参数值 a 拷贝到寄存器 XMM0 的低 32 位，而 XMM0 的高 32 位并不会被该指令修改。这个函数使用另一个 movss 指令把参数 b 拷贝到 XMM1。然后把结果数组的指针载入到 EAX 寄存器。

寄存器初始化之后的代码块展示了标量单精度浮点算术运算指令的使用方法。这部分代码相当简单明了，不多做解释。需要注意的是，与 x87-FPU 不同，x86-SSE 没有标量浮点绝对值（fabs）指令。标量单精度浮点绝对值可以很容易地通过 andps（组合单精度浮点值按位逻辑与）指令计算出来（使用 andps 清除掉符号位即可）。此外，所有的 x86-SSE 标量单精度浮点算术运算指令只修改目标 XMM 寄存器的低 32 位，而高 32 位不会被改变。注意 andps 指令是一个组合指令，该指令不仅修改低 32 位，同时也会修改目标操作数的高 32 位。通过 movss 指令，每一算术运算指令的结果被保存在调用者提供的数组里。建议大家在调用 movss 时将操作数在内存中正确对齐。虽然操作数对齐与否不影响运算的结果，但是不好的数据对齐会影响实际的运算性能。

SseSfpArithmeticDouble_ 函数展示了 x86-SSE 标量双精度浮点运算指令的使用方法。该函数的逻辑组织与 SseSfpArithmeticFloat 完全一致。在使用 XMM 寄存器中的标量双精度浮点数时，只有寄存器的低 64 位会用到，而高 64 在运算中保持不变。与单精度浮点运算一样，正确的数据对齐虽然不是必需的，但仍然强烈建议大家将标量双精度浮点操作数在内存中正确对齐。SseScalarFloatingPointArithmetic 的运算结果如输出 8-1 所示。

211

输出 8-1 示例程序 SseScalarFloatingPointArithmetic

```
Results for SseSfpArithmeticFloat_()
  a:              2.500000
  b:              -7.625000
  add:            -5.125000
  sub:            10.125000
  mul:            -19.062500
  div:            -0.327869
  min:            -7.625000
  max:            2.500000
  fabs(a):        7.625000
  sqrt(fabs(b)): 2.761340

Results for SseSfpArithmeticDouble_()
  a:              3.14159265358979
  b:              2.71828182845905
  add:            5.85987448204884
  sub:            0.42331082513075
  mul:            8.53973422267357
  div:            1.15572734979092
  min:            2.71828182845905
  max:            3.14159265358979
  fabs(b):        2.71828182845905
  sqrt(fabs(b)): 1.64872127070013
```

8.1.2 标量浮点数的比较

第二个 x86-SSE 标量浮点型示例程序是 SseScalarFloatingPointCompare。这个程序展示了如何使用 x86-SSE 标量浮点比较指令 comiss 和 comisd。清单 8-3 和清单 8-4 分别展示了
212 相应的 C++ 和 x86-32 汇编语言源代码。

清单 8-3 SseScalarFloatingPointCompare.cpp

```
#include "stdafx.h"
#include <limits>

using namespace std;

extern "C" void SseSfpCompareFloat_(float a, float b, bool* results);
extern "C" void SseSfpCompareDouble_(double a, double b, bool* results);

const int n = 7;
const char* OpStrings[n] = {"LO", "LT", "LE", "EQ", "NE", "GT", "GE"};

void SseSfpCompareFloat()
{
    const int n = 4;
    float a[n] = {120.0, 250.0, 300.0, 42.0};
    float b[n] = {130.0, 240.0, 300.0, 0.0};

    // 设定 NAN 测试值
    b[n - 1] = numeric_limits<float>::quiet_NaN();

    printf("Results for SseSfpCompareFloat()\n");
    for (int i = 0; i < n; i++)
    {
        bool results[n];

        SseSfpCompareFloat (a[i], b[i], results);
```

```
        printf("a: %8f b: %8f\n", a[i], b[i]);

        for (int j = 0; j < n; j++)
            printf("  %s=%d", OpStrings[j], results[j]);
        printf("\n");
    }
}

void SseSfpCompareDouble(void)
{
    const int n = 4;
    double a[n] = {120.0, 250.0, 300.0, 42.0};
    double b[n] = {130.0, 240.0, 300.0, 0.0};

    // 设定NAN测试值
    b[n - 1] = numeric_limits<double>::quiet_NaN();                    [213]
    printf("\nResults for SseSfpCompareDouble()\n");
    for (int i = 0; i < n; i++)
    {
        bool results[n];

        SseSfpCompareDouble_(a[i], b[i], results);
        printf("a: %8lf b: %8lf\n", a[i], b[i]);

        for (int j = 0; j < n; j++)
            printf("  %s=%d", OpStrings[j], results[j]);
        printf("\n");
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    SseSfpCompareFloat();
    SseSfpCompareDouble();
    return 0;
}
```

清单 8-4　SseScalarFloatingPointCompare_.asm

```
        .model flat,c
        .code

; extern "C" void SseSfpCompareFloat_(float a, float b, bool* results);
;
; 函数说明：本函数演示 comiss 指令的用法
;
; SSE 版本：SSE

SseSfpCompareFloat_ proc
        push ebp
        mov ebp,esp

; 载入参数值
        movss xmm0,real4 ptr [ebp+8]        ;xmm0 = a
        movss xmm1,real4 ptr [ebp+12]       ;xmm1 = b
        mov edx,[ebp+16]                    ;edx = 结果数组

; 基于比较结果设置结果标志
        comiss xmm0,xmm1
        setp byte ptr [edx]                 ;如果无序 EFLAGS.PF = 1
        jnp @F
        xor al,al                                                      [214]
```

```
        mov byte ptr [edx+1],al          ;置于缺省的结果数值
        mov byte ptr [edx+2],al
        mov byte ptr [edx+3],al
        mov byte ptr [edx+4],al
        mov byte ptr [edx+5],al
        mov byte ptr [edx+6],al
        jmp Done

@@:     setb byte ptr [edx+1]            ;如果 a < b 则设字节
        setbe byte ptr [edx+2]           ;如果 a <= b 则设字节
        sete byte ptr [edx+3]            ;如果 a == b 则设字节
        setne byte ptr [edx+4]           ;如果 a != b 则设字节
        seta byte ptr [edx+5]            ;如果 a > b 则设字节
        setae byte ptr [edx+6]           ;如果 a >= b 则设字节

Done:   pop ebp
        ret
SseSfpCompareFloat_ endp

; extern "C" void SseSfpCompareDouble_(double a, double b, bool* results);
;
; 函数说明：本函数演示 comisd 指令的用法
;
; SSE 版本：SSE2

SseSfpCompareDouble_ proc
        push ebp
        mov ebp,esp

; 载入参数数值
        movsd xmm0,real8 ptr [ebp+8]     ;xmm0 = a
        movsd xmm1,real8 ptr [ebp+16]    ;xmm1 = b
        mov edx,[ebp+24]                 ;edx = results array

; 基于比较结果设置对应标志
        comisd xmm0,xmm1
        setp byte ptr [edx]              ;如果无序 EFLAGS.PF = 1
        jnp @F
        xor al,al
        mov byte ptr [edx+1],al          ;使用缺省的结果数值
        mov byte ptr [edx+2],al
        mov byte ptr [edx+3],al
        mov byte ptr [edx+4],al
        mov byte ptr [edx+5],al
        mov byte ptr [edx+6],al
215     jmp Done

@@:     setb byte ptr [edx+1]            ;如果 a < b 则设字节
        setbe byte ptr [edx+2]           ;如果 a <= b 则设字节
        sete byte ptr [edx+3]            ;如果 a == b 则设字节
        setne byte ptr [edx+4]           ;如果 a != b 则设字节
        seta byte ptr [edx+5]            ;如果 a > b 则设字节
        setae byte ptr [edx+6]           ;如果 a >= b 则设字节

Done:   pop ebp
        ret
SseSfpCompareDouble_ endp
        end
```

SseScalarFloatingPointCompare.cpp（清单 8-3）的逻辑结构与前一个示例程序类似。它同样包含两个函数，分别初始化单精度和双精度浮点数的测试用例。请注意，为了验证相关

指令是否可以正确处理无序比较（unordered compare），每个测试用到的数组对都包含一个 NaN（Not a Number）值。C++ 代码同样包含显示每个测试用例结果的代码段。

在 SseScalarFloatingPointCompare_asm 文件里，汇编代码包括单精度和双精度两个函数。在函数序言之后，函数 SseSfpCompareFloat_ 使用 movss 指令把参数 a 和 b 分别载入到 XMM0 和 XMM1 寄存器，然后把指向结果数组的指针载入 EDX 寄存器。Comiss xmm0, xmm1 指令对寄存器 XMM0 和 XMM1 中的标量单精度浮点数进行比较，并根据比较结果设置 EFLAG 寄存器的状态位（参见表 8-1）。需要注意的是，表 8-1 里的条件码与之前在第 4 章讨论的 x87 FPU f(u)comi(p) 指令用到的条件码是一样的。

表 8-1 comiss 和 comisd 指令的结果（EFLAG）

关系操作符	条件码	EFLAGS 测试
XMM0 < XMM1	b	CF == 1
XMM0 <= XMM1	be	CF == 1 || ZF == 1
XMM0 == XMM1	z	ZF == 1
XMM0 != XMM1	nz	ZF == 0
XMM0 > XMM1	a	CF == 0 && ZF == 0
XMM0 >= XMM1	ae	CF == 0

216

SseSfpCompareFloat_ 函数使用了一系列 setcc 指令来解码和保存比较运算的结果。请注意，如果 XMM0 或 XXM1 寄存器有一个无序的浮点数，将会导致 EFLAGS.PF 状态位被置 1，并且结果数组的每个 Boolean（布尔）标志都被设为 false，同时在 comiss 或 comisd 指令后有可能会产生条件跳转。函数 SseSfpCompareDouble 与 SseSfpCompareFloat 几乎完全一样，只是使用 movsd 替换了 movss，用 comisd 替换了 comiss，堆栈的偏移量也是标量双精度浮点值。输出 8-2 显示了示例程序 SseScalarFloatingPointCompare 的输出结果。

输出 8-2 示例程序 SseScalarFloatingPointCompare

```
Results for SseSfpCompareFloat()
a: 120.000000 b: 130.000000
  UO=0  LT=1  LE=1  EQ=0  NE=1  GT=0  GE=0
a: 250.000000 b: 240.000000
  UO=0  LT=0  LE=0  EQ=0  NE=1  GT=1  GE=1
a: 300.000000 b: 300.000000
  UO=0  LT=0  LE=1  EQ=1  NE=0  GT=0  GE=1
a: 42.000000 b: 1.#QNAN0
  UO=1  LT=0  LE=0  EQ=0  NE=0  GT=0  GE=0

Results for SseSfpCompareDouble()
a: 120.000000 b: 130.000000
  UO=0  LT=1  LE=1  EQ=0  NE=1  GT=0  GE=0
a: 250.000000 b: 240.000000
  UO=0  LT=0  LE=0  EQ=0  NE=1  GT=1  GE=1
a: 300.000000 b: 300.000000
  UO=0  LT=0  LE=1  EQ=1  NE=0  GT=0  GE=1
a: 42.000000 b: 1.#QNAN0
  UO=1  LT=0  LE=0  EQ=0  NE=0  GT=0  GE=0
```

8.1.3 标量浮点数的类型转换

x86-SSE 包含多个对不同数据类型进行转换的指令，比如在 C—— 程序里最常见的浮点

数与整型数间的相互转换操作。示例程序 SseScalarFloatingPointConversions 演示了如何使用 x86-SSE 转换指令进行数据类型转换。这个示例程序同时演示了如何通过修改 MXCSR 寄存器的浮点舍入控制位改变 x86-SSE 的浮点舍入模式。清单 8-5 和清单 8-6 分别包含了
[217] SseScalarFloatingPointConversions 示例程序的 C++ 和汇编语言的源代码。

清单 8-5 SseScalarFloatingPointConversions.cpp

```
#include "stdafx.h"
#define _USE_MATH_DEFINES
#include <math.h>
#include "MiscDefs.h"

// 简单的联合 (union), 用于数据交换
union XmmScalar
{
    float r32;
    double r64;
    Uint32 i32;
    Uint64 i64;
};

// 数据转换序列枚举与 SseScalarFloatingPointConversions_.asm 中定义的转换一致
enum CvtOp : unsigned int
{
    Cvtsi2ss,       // Int32 转成 float
    Cvtss2si,       // float 转成 Int32
    Cvtsi2sd,       // Int32 转成 double
    Cvtsd2si,       // double 转成 Int32
    Cvtss2sd,       // float 转成 double
    Cvtsd2ss,       // double 转成 float
};

// x86-SSE 舍入模式的枚举类型
enum SseRm : unsigned int
{
    Nearest, Down, Up, Truncate
};

extern "C" Uint32 SseGetMxcsr_(void);
extern "C" Uint32 SseSetMxcsr_(Uint32 mxcsr);

extern "C" SseRm SseGetMxcsrRoundingMode_(void);
extern "C" void SseSetMxcsrRoundingMode_(SseRm rm);
extern "C" bool SseSfpConversion_(XmmScalar* a, XmmScalar* b, CvtOp cvt_op);

const SseRm SseRmVals[] = {SseRm::Nearest, SseRm::Down, SseRm::Up,↵
SseRm::Truncate};
[218] const char* SseRmStrings[] = {"Nearest", "Down", "Up", "Truncate"};

void SseSfpConversions(void)
{
    XmmScalar src1, src2;
    XmmScalar des1, des2;
    const int num_rm = sizeof(SseRmVals) / sizeof (SseRm);
    Uint32 mxcsr_save = SseGetMxcsr_();

    src1.r32 = (float)M_PI;
    src2.r64 = -M_E;

    for (int i = 0; i < num_rm; i++)
```

```
    {
        SseRn rm1 = SseRnVals[i];
        SseRn rm2;

        SseSetMxcsrRoundingMode_(rm1);
        rm2 = SseGetMxcsrRoundingMode_();

        if (rm2 != rm1)
        {
            printf("  SSE rounding mode change failed)\n");
            printf("  rm1: %d  rm2: %d\n", rm1, rm2);
        }
        else
        {
            printf("X86-SSE rounding mode = %s\n", SseRnStrings[rm2]);

            SseSfpConversion_(&des1, &src1, CvtOp::Cvtss2si);
            printf("  cvtss2si: %12lf --> %6d\n", src1.r32, des1.i32);

            SseSfpConversion_(&des2, &src2, CvtOp::Cvtsd2si);
            printf("  cvtsd2si: %12lf --> %6d\n", src2.r64, des2.i32);
        }
    }

    SseSetMxcsr_(mxcsr_save);
}
int _tmain(int argc, _TCHAR* argv[])
{
    SseSfpConversions();
    return 0;
}
```

219

清单 8-6 SseScalarFloatingPointConversions_.asm

```
        .model flat,c
        .code

; extern "C" bool SseSfpConversion_(XmmScalar* des, const XmmScalar* src,
CvtOp cvt_op)
;
; 函数说明：本函数演示 x86 SSE 标量浮点转换指令的使用方法
;
; SSE 版本：SSE2

SseSfpConversion_ proc
        push ebp
        mov ebp,esp

; 载入参数值，并确保 cvt_op 是有效的
        mov eax,[ebp+16]                ;cvt_op
        mov ecx,[ebp+12]                ;ptr to src
        mov edx,[ebp+8]                 ;ptr to des
        cmp eax,CvtOpTableCount
        jae BadCvtOp                    ;如果 cvt_op 无效，跳转到 BadCvtOp
        jmp [CvtOpTable+eax*4]          ;跳转到三种标量类型转换例程

SseCvtsi2ss:
        mov eax,[ecx]                   ;载入参数
        cvtsi2ss xmm0,eax               ;转换为单精度浮点数
        movss real4 ptr [edx],xmm0      ;保存结果
        mov eax,1
```

```
        pop ebp
        ret

SseCvtss2si:
        movss xmm0,real4 ptr [ecx]          ;载入浮点数
        cvtss2si eax,xmm0                   ;转换成整数
        mov [edx],eax                       ;保存结果
        mov eax,1
        pop ebp
        ret

SseCvtsi2sd:
        mov eax,[ecx]                       ;载入整数
        cvtsi2sd xmm0,eax                   ;转换成双精度浮点数
        movsd real8 ptr [edx],xmm0          ;保存结果
        mov eax,1
        pop ebp
        ret

SseCvtsd2si:
        movsd xmm0,real8 ptr [ecx]          ;载入双精度浮点数
        cvtsd2si eax,xmm0                   ;转换成整数
        mov [edx],eax                       ;保存结果
        mov eax,1
        pop ebp
        ret

SseCvtss2sd:
        movss xmm0,real4 ptr [ecx]          ;载入单精度浮点数
        cvtss2sd xmm1,xmm0                  ;转换成双精度浮点数
        movsd real8 ptr [edx],xmm1          ;保存结果
        mov eax,1
        pop ebp
        ret

SseCvtsd2ss:
        movsd xmm0,real8 ptr [ecx]          ;载入双精度浮点数
        cvtsd2ss xmm1,xmm0                  ;转换成单精度浮点数
        movss real4 ptr [edx],xmm1          ;保存结果
        mov eax,1
        pop ebp
        ret

BadCvtOp:
        xor eax,eax                         ;设置返回错误码
        pop ebp
        ret

; 下表中这些数据的顺序必须与 SseScalarFloatingPointConversions.cpp 中定义的 enum CvtOp 一致
            align 4
CvtOpTable  dword SseCvtsi2ss, SseCvtss2si
            dword SseCvtsi2sd, SseCvtsd2si
            dword SseCvtss2sd, SseCvtsd2ss
CvtOpTableCount equ ($ - CvtOpTable) / size dword
SseSfpConversion_ endp

; extern "C" UINT32 SseGetMxcsr_(void);
;
; 函数说明：本函数用于获得 MXCSR 寄存器的当前内容
;
; 返回值：MXCSR 的内容
```

```
SseGetMxcsr_ proc
        push ebp
        mov ebp,esp
        sub esp,4

        stmxcsr [ebp-4]                 ;保存 MXCSR 寄存器
        mov eax,[ebp-4]                 ;把 MXCSR 寄存器值载入到 EAX

        mov esp,ebp
        pop ebp
        ret
SseGetMxcsr_ endp

; extern "C" Jint32 SseSetMxcsr_(Jint32 mxcsr);
;
; 函数说明：本函数把一个新数值载入 MXCSR 寄存器

SseSetMxcsr_ proc
        push ebp
        mov ebp,esp
        sub esp,4

        mov eax,[ebp+8]                 ;eax = 新 MXCSR 值
        and eax,0ffffh                  ;mxcsr[31:16] 必须为 0
        mov [ebp-4],eax
        ldmxcsr [ebp-4]                 ;加载 MXCSR 寄存器

        mov esp,ebp
        pop ebp
        ret
SseSetMxcsr_ endp

; extern "C" SseRn SseGetMxcsrRoundingMode_(void);
;
; 函数说明：本函数从 MXCSR.RC 获得当前的 x86-SSE 的浮点舍入模式
;
; 返回值：当前的 x86-SSE 舍入模式

MxcsrRcMask equ 9fffh                   ;MXCSR.RC 位模式
MxcsrRcShift equ 13                     ;MXCSR.RC 移位数

SseGetMxcsrRoundingMode_ proc
        push ebp
        mov ebp,esp
        sub esp,4                                                    222

        stmxcsr [ebp-4]                 ;保存 MXCSR
        mov eax,[ebp-4]
        shr eax,MxcsrRcShift            ;eax[1:0] = MXCSR.RC 相应位
        and eax,3                       ;屏蔽掉不关的位

        mov esp,ebp
        pop ebp
        ret
SseGetMxcsrRoundingMode_ endp

;extern "C" void SseSetMxcsrRoundingMode (SseRn rn);
;
; 函数说明：下面的函数改变 MXCSR.RC 中的舍入模式

SseSetMxcsrRoundingMode_ proc
```

```
        push ebp
        mov ebp,esp
        sub esp,4

        mov ecx,[ebp+8]                 ;ecx = rm
        and ecx,3                       ;屏蔽掉不用的位
        shl ecx,MxcsrRcShift            ;ecx[14:13] = rm

        stmxcsr [ebp-4]                 ;保存当前的MXCSR
        mov eax,[ebp-4]
        and eax,MxcsrRcMask             ;屏蔽掉旧的MXCSR.RC位
        or eax,ecx                      ;插入新的MXCSR.RC位
        mov [ebp-4],eax
        ldmxcsr [ebp-4]                 ;载入更新后的MXCSR

        mov esp,ebp
        pop ebp
        ret
SseSetMxcsrRoundingMode endp
        end
```

SseScalarFloatingPointConversions.cpp（清单8-5）的开始部分声明了一个C++联合体XmmScalar，用于交换数据。接下来定义了两个枚举类型：CvtOp和SseRm。CvtOp用于选择浮点转换类型，而SseRm用于定义x86-SSE浮点舍入模式。C++函数SseSfpConversions先初始化一组XmmScalar实例作为测试数据，并且调用一个汇编语言函数进行不同舍入模式下的浮点数到整数的数据类型转换。各个转换操作的结果最后被显示出来以便于验证和比较。

223

x86-SSE浮点舍入模式由MXCSR寄存器的浮点控制位域（第13和14位）决定。对于Visual C++，缺省的浮点舍入模式是就近舍入。依照Visual C++的调用约定，MXCSR[15:6]（MXCSR寄存器的第15到第6位）在大多数函数边界必须被保护起来。SseSfpConversions在改变x86-SSE舍入模式前保存MXCSR的内容，并在退出前还原MXCSR寄存器的内容，满足了上述要求。

汇编语言源文件SseScalarFloatingPointConversions_asm（清单8-6）包含数据类型转换和MXCSR寄存器控制函数。SseSfpConversion_函数使用特定的数据和转换操作符实现浮点型数据转换。与我们在之前章节的示例程序中已经见到的一样，这个函数使用了函数转表。汇编语言文件SseScalarFloatingPointConversions_asm同样包含多个管理MXCSR寄存器的函数。函数SseGetMxcsr_和SseSetMxcsr_分别用于读取和写入MXCSR寄存器。这两个函数分别使用了stmxcsr（Store MXCSR Register State，存储MXCSR寄存器状态）和ldmxcsr（Load MXCSR Register，载入MXCSR寄存器）指令。stmxcsr和ldmxcsr都要求它们独占的操作数为双字数据保存在内存中。函数SseGetMxcsrRoundingMode_和SseSetMxcsrRoundingMode_可以用来修改当前的x86-SSE浮点舍入模式。这两个函数使用枚举SseRm来保存或者选取x86-SSE浮点舍入模式。

其实并不是任意两种数据类型都可以转换，比如cvtss2si指令就不能把一个大的浮点数转换成带符号双字整型数。如果一个特定的转换无法进行且无效操作异常（MXCSR.IM）被屏蔽（Visual C++缺省情况），处理器会设置MXCSR的IE位（Invalid Operation Error Flag，无效操作错误标志），并把0x80000000复制到目标操作数。输出8-3显示例程序SseScalarFloatingPointConversions的执行结果。

输出 8-3 示例程序 SseScalarFloatingPointConversions

```
X86-SSE rounding mode = Nearest
  cvtss2si:     3.141593 -->      3
  cvtsd2si:    -2.718282 -->     -3
X86-SSE rounding mode = Down
  cvtss2si:     3.141593 -->      3
  cvtsd2si:    -2.718282 -->     -3
X86-SSE rounding mode = Up
  cvtss2si:     3.141593 -->      4
  cvtsd2si:    -2.718282 -->     -2
X86-SSE rounding mode = Truncate
  cvtss2si:     3.141593 -->      3
  cvtsd2si:    -2.718282 -->     -2
```

224

8.2 高级标量浮点编程

本节的示例程序将演示如何使用 x86-SSE 标量浮点指令集进行高级计算。第一个示例程序是之前一个示例程序的变形，重点在于突出演示 x87 FPU 和 x86-SSE 之间的一些关键的不同点。在第二个示例程序里，我们将学习如何在包含 x86-SSE 指令的汇编语言函数里使用数据结构和标准 C++ 库函数。

8.2.1 用标量浮点指令计算球体表面积和体积

现在我们已经对 x86-SSE 的标量浮点计算能力有所了解，是时候利用之前所学到的东西进行一些实际的运算了。在本节我们将学习 SseScalarFloatingPointSpheres 这个示例程序。这个程序包含一个汇编语言函数，使用 x86-SSE 的标量浮点指令计算球体的表面积和体积。在第 4 章有一个示例程序，使用 x87 FPU 指令集计算球体的表面积和体积。现在用 x86-SSE 替换 x87 FPU 来重构一下那个程序，我们将体验到使用 x86-SSE 来实现同样的任务是多么轻松。清单 8-7 和清单 8-8 分别列出了 C++ 源文件 SseScalarFloatingPointSpheres.cpp 和汇编源文件 SseScalarFloatingPointSpheres_.asm 的内容。

清单 8-7 SseScalarFloatingPointSpheres.cpp

```
#include "stdafx.h"

extern "C" bool SseSfpCalcSphereAreaVolume_(double r, double* sa, double* v);

int _tmain(int argc, _TCHAR* argv[])
{
    const double r[] = {-1.0, 0.0, 1.0, 2.0, 3.0, 5.0, 10.0, 20.0};
    int num_r = sizeof(r) / sizeof(double);

    for (int i = 0; i < num_r; i++)
    {
        double sa, v;
        bool rc = SseSfpCalcSphereAreaVolume_(r[i], &sa, &v);

        printf("rc: %d  r: %8.2lf  sa: %10.4lf  vol: %10.4lf\n", rc, r[i],=
sa, v);
    }

    return 0;
}
```

225

清单 8-8 SseScalarFloatingPointSpheres_.asm

```
            .model flat,c

; 计算球体表面积和体积所用到的常量
            .const
r8_pi       real8 3.14159265358979323846
r8_four     real8 4.0
r8_three    real8 3.0
r8_neg_one  real8 -1.0
            .code

; extern "C" bool SseSfpCalcSphereAreaVolume_(double r, double* sa, double* v);
;
; 函数描述：本函数计算一个球体的表面积和体积
;
; 返回：0 = 球体半径无效
;       1 = 计算成功
;
; SSE版本：SSE2

SseSfpCalcSphereAreaVolume_ proc
        push ebp
        mov ebp,esp

; 载入参数并确定球体半径值是有效的
        movsd xmm0,real8 ptr [ebp+8]        ;xmm0 = r
        mov ecx,[ebp+16]                    ;ecx = sa
        mov edx,[ebp+20]                    ;edx = v
        xorpd xmm7,xmm7                     ;xmm7 = 0.0
        comisd xmm0,xmm7                    ;比较r是不是0.0
        jp BadRadius                        ;如果r为NAN，跳转
        jb BadRadius                        ;如果r < 0.0，跳转

; 计算球体表面积
        movsd xmm1,xmm0                     ;xmm1 = r
        mulsd xmm1,xmm1                     ;xmm1 = r * r
        mulsd xmm1,[r8_four]                ;xmm1 = 4 * r * r
        mulsd xmm1,[r8_pi]                  ;xmm1 = 4 * pi r * r
226     movsd real8 ptr [ecx],xmm1          ;保存表面积

; 计算球体体积
        mulsd xmm1,xmm0                 ;xmm1 = 4 * pi * r * r * r
        divsd xmm1,[r8_three]           ;xmm1 = 4 * pi * r * r * r / 3
        movsd real8 ptr [edx],xmm1      ;保存体积
        mov eax,1                       ;设置返回值为1
        pop ebp
        ret

; 球体半径非法情况下的处理：表面积和体积都设为-1.0
BadRadius:
        movsd xmm0,[r8_neg_one]         ;xmm0 = -1.0
        movsd real8 ptr [ecx],xmm0      ;*sa = -1.0
        movsd real8 ptr [edx],xmm0      ;*v = -1.0;
        xor eax,eax                     ;设定返回值为0
        pop ebp
        ret
SseSfpCalcSphereAreaVolume_ endp
        end
```

SseScalarFloatingPointSpheres 的 C++ 部分（清单 8-7）很简单，就是使用不同的测试值（球

的半径）调用汇编语言函数 SseSfpCalcSphereAreaVolume_。在 SseSfpCalcSphere-AreaVolume_ 函数（清单 8-8）序言之后，首先把参数 r、sa 和 v 分别载入 XMM0、ECX 和 EDX。然后调用 comisd xmm0, xmm7 指令比较 r 和 0.0，并用 EFLAGS 的状态位标识比较结果。不同于 x87 FPU，x86-SSE 指令集不支持 fldz 和 fldpi 之类的载入常量数据的指令。所有浮点常量都需要从内存载入或者通过 x86-SSE 指令计算而来，这也是在调用 comisd 之前先调用 xorpd（Bitwise Logical XOR for Double-Precision Floating-Point Values，双精度浮点数的按位逻辑异或）指令的原因。之后两个条件跳转指令 jp 和 jb 用来防止函数使用非法的参数（半径值）。

接下来的代码通过调用 mulsd 指令计算球体表面积，然后调用 mulsd 和 divsd 计算球体体积。函数 SseSfpCalcSphereAreaVolume_ 很好地说明了相对于 x87 FPU 指令集，用 x86-SSE 指令集来进行标量浮点算术运算多么简单而轻松。输出 8-4 显示了示例程序 SseScalarFloatingPointSpheres 的执行结果。

输出 8-4　示例程序 SseScalarFloatingPointSpheres

```
rc: 0  r:    -1.00  sa:    -1.0000  vol:    -1.0000
rc: 1  r:     0.00  sa:     0.0000  vol:     0.0000
rc: 1  r:     1.00  sa:    12.5664  vol:     4.1888
rc: 1  r:     2.00  sa:    50.2655  vol:    33.5103
rc: 1  r:     3.00  sa:   113.0973  vol:   113.0973
rc: 1  r:     5.00  sa:   314.1593  vol:   523.5988
rc: 1  r:    10.00  sa:  1256.6371  vol:  4188.7902
rc: 1  r:    20.00  sa:  5026.5482  vol: 33510.3216
```

227

8.2.2　用标量浮点指令计算平行四边形面积和对角线长度

本章的最后一个标量浮点示例程序 SseScalarFloatingPointParallelograms 将演示如何使用 x86-SSE 标量浮点指令来根据边长和一个夹角计算平行四边形的面积和对角线长度，同时这个示例将演示如何在汇编语言函数内调用 C— 库函数。清单 8-9 和清单 8-10 分别显示了示例程序 SseScalarFloatingPointParallelograms 的 C++ 和汇编语言的源代码。

清单 8-9　SseScalarFloatingPointParallelograms.cpp

```
#include "stdafx.h"
#define _USE_MATH_DEFINES
#include <math.h>
#include <stddef.h>

// 如果要显示 PDATA 信息，请去掉下面这行注释符号
//#define DISPLAY_PDATA_INFO

// 下面这个结构体必须与 SseScalarFloatingPointParallelograms_.asm 文件里定义的结构体一致
typedef struct
{
    double A;               // 左右两条边的长度
    double B;               // 上下两条边的长度
    double Alpha;           // alpha 夹角的角度
    double Beta;            // beta 夹角的角度
    double H;               // 平行四边形的高度
    double Area;            // 平行四边形面积
    double P;               // 对角线 P 的长度
    double Q;               // 对角线 Q 的长度
    bool BadValue;          // 如果 A、B 或者 Alpha 非法，则设为 true
    char Pad[7];            // 保留给将来使用
```

```
} PDATA;

extern "C" bool SseSfpParallelograms_(PDATA* pdata, int n);
extern "C" double DegToRad = M_PI / 180.0;
extern "C" int SizeofPdataX86_;
const bool PrintPdataInfo = true;

void SetPdata(PDATA* pdata, double a, double b, double alpha)
{
    pdata->A = a;
    pdata->B = b;
    pdata->Alpha = alpha;
}

int _tmain(int argc, _TCHAR* argv[])
{
#ifdef DISPLAY_PDATA_INFO
    size_t spd1 = sizeof(PDATA);
    size_t spd2 =  SizeofPdataX86_;

    if (spd1 != spd2)
        printf("PDATA size discrepancy [%d, %d]", spd1, spd2);
    else
    {
        printf("sizeof(PDATA):      %d\n", spd1);
        printf("Offset of A:        %d\n", offsetof(PDATA, A));
        printf("Offset of B:        %d\n", offsetof(PDATA, B));
        printf("Offset of Alpha:    %d\n", offsetof(PDATA, Alpha));
        printf("Offset of Beta:     %d\n", offsetof(PDATA, Beta));
        printf("Offset of H         %d\n", offsetof(PDATA, H));
        printf("Offset of Area:     %d\n", offsetof(PDATA, Area));
        printf("Offset of P:        %d\n", offsetof(PDATA, P));
        printf("Offset of Q:        %d\n", offsetof(PDATA, Q));
        printf("Offset of BadValue  %d\n", offsetof(PDATA, BadValue));
        printf("Offset of Pad       %d\n", offsetof(PDATA, Pad));
    }
#endif

    const int n = 10;
    PDATA pdata[n];

    // 创建一些测试用的平行四边形
    SetPdata(&pdata[0], -1.0, 1.0, 60.0);
    SetPdata(&pdata[1], 1.0, -1.0, 60.0);
    SetPdata(&pdata[2], 1.0, 1.0, 181.0);
    SetPdata(&pdata[3], 1.0, 1.0, 90.0);
    SetPdata(&pdata[4], 3.0, 4.0, 90.0);
    SetPdata(&pdata[5], 2.0, 3.0, 30.0);
    SetPdata(&pdata[6], 3.0, 2.0, 60.0);
    SetPdata(&pdata[7], 4.0, 2.5, 120.0);
    SetPdata(&pdata[8], 5.0, 7.125, 135.0);
    SetPdata(&pdata[9], 8.0, 8.0, 165.0);

    SseSfpParallelograms_(pdata, n);

    for (int i = 0; i < n; i++)
    {
        PDATA* p = &pdata[i];
        printf("\npdata[%d] - BadValue = %d\n", i, p->BadValue);
        printf("A: %12.6lf B: %12.6lf\n", p->A, p->B);
        printf("Alpha: %12.6lf Beta: %12.6lf\n", p->Alpha, p->Beta);
        printf("H: %12.6lf Area: %12.6lf\n", p->H, p->Area);
```

228

229

```
        printf("P: %12.6lf Q: %12.6lf\n", p->P, p->Q);
    }

    return 0;
}
```

清单 8-10 SseScalarFloatingPointParallelograms_.asm

```
        .model flat,c

; 下面这个结构体必须与 SseScalarFloatingPointParallelograms.cpp 文件里定义的结构体一致
PDATA   struct
A       real8 ?
B       real8 ?
Alpha   real8 ?
Beta    real8 ?
H       real8 ?
Area    real8 ?
P       real8 ?
Q       real8 ?
BadVal  byte ?
Pad     byte 7 dup(?)
PDATA   ends

; 常量定义
            .const
            public SizeofPdataX86_
r8_2p0      real8 2.0
r8_180p0    real8 180.0
r8_MinusOne real8 -1.0
SizeofPdataX86_ dword size PDATA

            .code
            extern sin:proc, cos:proc
            extern DegToRad:real8

; extern "C" bool Sse5fpParallelograms_(PDATA* pdata, int n);
;
; 函数说明：本函数计算平行四边形的面积和对角线长度
;
; 返回值：0  n <= 0
;        1  n > 0
;
; 局部栈：[ebp-8] x87 FPU 控制位置
;         [ebp-16] Alpha 夹角弧度
;
; SSE 版本：SSE2

Sse5fpParallelograms_ proc
        push ebp
        mov ebp,esp
        sub esp,16                          ;为局部变量分配空间
        push ebx

; 载入并检测 n
        xor eax,eax                         ;设置错误码
        mov ebx,[ebp+8]                     ;ebx = pdata
        mov ecx,[ebp+12]                    ;ecx = n
        test ecx,ecx
        jle Done                            ;如果 n <= 0 则跳转到 Done

; 初始化常量
```

```
Loop1:  movsd xmm6,real8 ptr [r8_180p0]     ;xmm6 = 180.0
        xorpd xmm7,xmm7                     ;xmm7 = 0.0
        sub esp,8                           ;为 sin/cos arg 预留空间

; 载入并验证 A 和 B
        movsd xmm0,real8 ptr [ebx+PDATA.A]  ;xmm0 = A
        movsd xmm1,real8 ptr [ebx+PDATA.B]  ;xmm0 = B
        comisd xmm0,xmm7
        jp InvalidValue
        jbe InvalidValue                    ;如果 A <= 0.0 则跳转到 InvalidValue
        comisd xmm1,xmm7
        jp InvalidValue
        jbe InvalidValue                    ;如果 B <= 0.0 则跳转到 InvalidValue

; 载入并验证 Alpha
        movsd xmm2,real8 ptr [ebx+PDATA.Alpha]
        comisd xmm2,xmm7
        jp InvalidValue
        jbe InvalidValue                    ;如果 Alpha <= 0.0, 则跳转
        comisd xmm2,xmm6
        jae InvalidValue                    ;如果 Alpha >= 180.0, 则跳转

; 计算 Beta
        subsd xmm6,xmm2                     ;Beta = 180.0 - Alpha
231     movsd real8 ptr [ebx+PDATA.Beta],xmm6  ;保存 Beta

; 计算 sin(Alpha)
        mulsd xmm2,real8 ptr [DegToRad]     ;把 Alpha 角度值转换成弧度值
        movsd real8 ptr [ebp-16],xmm2
        movsd real8 ptr [esp],xmm2          ;把 Alpha 拷贝到堆栈
        call sin
        fstp real8 ptr [ebp-8]              ;保存 sin(Alpha)

; 计算平行四边形高度和面积
        movsd xmm0,real8 ptr [ebx+PDATA.A]  ;A
        mulsd xmm0,real8 ptr [ebp-8]        ;A * sin(Alpha)
        movsd real8 ptr [ebx+PDATA.H],xmm0  ;保存高度
        mulsd xmm0,real8 ptr [ebx+PDATA.B]  ;A * sin(Alpha) * B
        movsd real8 ptr [ebx+PDATA.AREA],xmm0  ;保存面积

; 计算 cos(Alpha)
        movsd xmm0,real8 ptr [ebp-16]       ;xmm0 = Alpha 角度的弧度值
        movsd real8 ptr [esp],xmm0          ;把 Alpha 拷贝到堆栈
        call cos
        fstp real8 ptr [ebp-8]              ;保存 cos(Alpha)

; 计算 2.0 * A * B * cos(Alpha)
        movsd xmm0,real8 ptr [r8_2p0]
        movsd xmm1,real8 ptr [ebx+PDATA.A]
        movsd xmm2,real8 ptr [ebx+PDATA.B]
        mulsd xmm0,xmm1                     ;2 * A
        mulsd xmm0,xmm2                     ;2 * A * B
        mulsd xmm0,real8 ptr [ebp-8]        ;2 * A * B * cos(Alpha)

; 计算 A * A + B * B
        movsd xmm3,xmm1
        movsd xmm4,xmm2
        mulsd xmm3,xmm3                     ;A * A
        mulsd xmm4,xmm4                     ;B * B
        addsd xmm3,xmm4                     ;A * A + B * B
        movsd xmm4,xmm5                     ;A * A + B * B
```

```
; 计算 P 和 Q
        subsd xmm3,xmm0
        sqrtsd xmm3,xmm3                    ;xmm3 = P
        movsd real8 ptr [ebx+PDATA.P],xmm3
        addsd xmm4,xmm0
        sqrtsd xmm4,xmm4                    ;xmm4 = Q
        movsd real8 ptr [ebx+PDATA.Q],xmm4
        mov dword ptr [ebx+PDATA.BadVal],0  ;把 BadVal 设为 false

NextItem:
        add ebx,size PDATA                  ;ebx = 数组中的下一个元素
        dec ecx
        jnz Loop1                           ;重复循环直到结束

        add esp,8                           ;恢复 ESP
Done:   pop ebx
        mov esp,ebp
        pop ebp
        ret

; 把结构体成员设为已知以便显示
InvalidValue:
        movsd xmm0,real8 ptr [r8_MinusOne]
        movsd real8 ptr [ebx+PDATA.Beta],xmm0
        movsd real8 ptr [ebx+PDATA.H],xmm0
        movsd real8 ptr [ebx+PDATA.Area],xmm0
        movsd real8 ptr [ebx+PDATA.P],xmm0
        movsd real8 ptr [ebx+PDATA.Q],xmm0
        mov dword ptr [ebx+PDATA.BadVal],1
        jmp NextItem

SseSfpParallelograms_ endp
        end
```

在分析 SseScalarFloatingPointParallelograms 的源代码之前，我们先来复习一下平行四边形的基本几何性质。图 8-1 展示了一个标准的平行四边形。这张图（包括源代码）用 A 表示左右两条边的长度，用 B 表示上下两条边的长度，用 H 表示高度，用 α（Alpha）和 β（Beta）表示左右两个夹角，用 P 和 Q 表示两条对角线的长度。

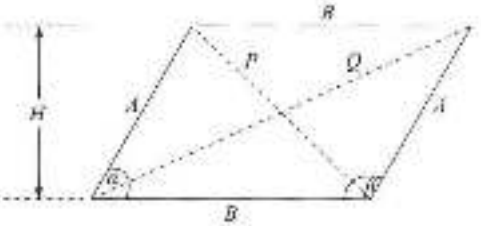


图 8-1 一个标准的平行四边形

H、β、P 和 Q 可以从 A、B 和 α 计算出来，计算公式如下：

$$\beta = 180 - \alpha \qquad H = A\sin(\alpha) \qquad Area = AB\sin(\alpha)$$

$$P = \sqrt{A^2 + B^2 - 2AB\cos(A)} \qquad Q = \sqrt{A^2 + B^2 + 2AB\cos(A)}$$

我们现在回过头来分析 SseScalarFloatingPointParallelograms 这个示例程序。SseScalarFloatingPointParallelograms 定义了一个结构体来帮助管理平行四边形的各个参数。在 C++ 源代码里，这个结构体是 PDATA，定义在源文件 SseScalarFloatingPointParallelograms.cpp 的头部。当我们在 C++ 里声明一个包含多个元素的结构体的时候，需要知道每个元素在结构体内的偏移，这样才能保证在汇编语言源代码里定义同样的数据结构的时候保持一致性。可通过定义 DISPLAY_PDATA_INFO 预处理器来启用 _tmain 函数开头的一块代码。因为汇编语言函数 SseSfpParallelograms_ 要处理包含 PDATA 项的数组，所以保证 PDATA 这个结构体在 C++ 和汇编语言两个版本里的大小一致很重要。

结构体 PDATA 的汇编语言版本定义在 SseScalarFloatingPointParallelograms_asm(清单 8-10)文件的头部。特别要注意的是，与 C++ 编译器不同，汇编编译器不会自动地把结构体的成员对齐到它们的自然边界（比如自动 4 字节对齐或 8 字节对齐）。在汇编语言里，定义结构体时经常需要加入一些额外的补齐（padding）字节来保证结构体与 C++ 的结构体一致。汇编语言函数 SseSfpParallelograms_ 使用一个循环来计算每个平行四边形的未知数值。在这个循环的开始处（接近 Loop1 的地方），sub esp, 8 语句在堆栈上创建了一个 8 字节的空间，用于存储一个将来会用到的双精度浮点数函数参数。这个循环每次执行时，都先通过一系列 comisd 指令验证 A、B 和 α，并配合一系列条件跳转指令来控制运行逻辑。当所有这三个参数都通过验证后，就会计算并保存 β 角的值。

下一步将计算 sin(α)。我们先把 α 从角度值转换成弧度值，然后用 movsd real8 ptr [ebp-16], xmm2 指令把转换后的弧度值保存在局部堆栈上以备将来使用。α 的角度值同时被保存在堆栈上（通过 movsd real8 ptr [esp], xmm2 指令）。接下来就是调用 C++ 库函数的 call sin 指令计算 sin(α) 的值。需要注意的是，与 x87 FPU 不同，x86-SSE 没有提供类似于 fsin 和 fcos 这样的高级指令。依照 Visual C++ 的调用约定，sin 函数的返回值被保存在 x87 FPU 寄存器栈上，于是我们通过 fstp real8 ptr[ebp-8] 指令把 sin(α) 的结果拷贝到局部堆栈变量，并把它从 x87 FPU 的寄存器栈上移除掉。需要特别注意的是，Visual C++ 的 32 位程序的调用约定把所有 XMM 寄存器都认为是易变的，这意味着在执行完 sin 或其他函数之后，XMM0-XMM7 寄存器的值都是不确定的。

在计算完 sin(α) 之后，平行四边形的高度和面积便可以轻松地计算出来，结果被保存在 PDATA 结构里（EBX 指向的数据位置）。再之后我们调用 C++ 库函数 cos 计算 cos(α)，最后，P 和 Q 的长度也就可以计算出来了。注意，求 P 和 Q 都需要的公共子表达式只要计算一次。所有循环变量在 NextItem 后的代码块中进行更新。在所有循环处理结束后和函数结束之前，ESP 寄存器需要被还原到适当的值（通过 add esp, 8 指令）。输出 8-5 显示了示例
234 程序 SseScalarFloatingPointParallelograms 的执行结果。

输出 8-5 示例程序 SseScalarFloatingPointParallelograms

```
pdata[0] - BadValue = 1
  A:          -1.000000  B:           1.000000
  Alpha:      60.000000  Beta:       -1.000000
  H:          -1.000000  Area:       -1.000000
  P:          -1.000000  Q:          -1.000000

pdata[1] - BadValue = 1
  A:           1.000000  B:          -1.000000
  Alpha:      60.000000  Beta:       -1.000000
  H:          -1.000000  Area:       -1.000000
  P:          -1.000000  Q:          -1.000000

pdata[2] - BadValue = 1
  A:           1.000000  B:           1.000000
  Alpha:     181.000000  Beta:       -1.000000
  H:          -1.000000  Area:       -1.000000
  P:          -1.000000  Q:          -1.000000

pdata[3] - BadValue = 0
  A:           1.000000  B:           1.000000
  Alpha:      90.000000  Beta:       90.000000
  H:           1.000000  Area:        1.000000
```

```
  P:          1.414214  Q:          1.414214

pdata[4] - BadValue = 0
  A:          3.000000  B:          4.000000
  Alpha:     90.000000  Beta:      90.000000
  H:          3.000000  Area:      12.000000
  P:          5.000000  Q:          5.000000

pdata[5] - BadValue = 0
  A:          2.000000  B:          3.000000
  Alpha:     30.000000  Beta:     150.000000
  H:          1.000000  Area:      3.000000
  P:          1.614836  Q:          4.836559

pdata[6] - BadValue = 0
  A:          3.000000  B:          2.000000
  Alpha:     60.000000  Beta:     120.000000
  H:          2.598076  Area:      5.196152
  P:          2.645751  Q:          4.358899

pdata[7] - BadValue = 0
  A:          4.000000  B:          2.500000
  Alpha:    120.000000  Beta:      60.000000
  H:          3.464102  Area:      8.660254
  P:          5.678908  Q:          3.500000

pdata[8] - BadValue = 0
  A:          5.000000  B:          7.125000
  Alpha:    135.000000  Beta:      45.000000
  H:          5.535534  Area:      25.190679
  P:         11.231517  Q:          5.038280

pdata[9] - BadValue = 0
  A:          8.000000  B:          8.000000
  Alpha:    165.000000  Beta:      15.000000
  H:          2.070552  Area:      16.564419
  P:         15.863118  Q:          2.088419
```

235

8.3 总结

在本章，我们学习了如何使用 x86-SSE 指令集进行基本的标量浮点算术运算，同时通过几个示例程序学习了一些高级的 x86-SSE 标量浮点计算技术。在下一章，我们将继续学习 x86-SSE 汇编语言编程技术，重点将集中在 x86-SSE 的紧合浮点运算能力。

236

第 9 章

Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX

x86-SSE 编程——组合浮点

在本章，我们将学习如何在汇编语言函数中操纵 x86-SSE 的组合浮点资源。我们先通过几个示例程序学习基本的 x86-SSE 组合浮点运算，包括基本的算术运算，比较以及数据类型转换。在其后一节我们将通过示例来学习使用 x86-SSE 指令对更复杂的组合单精度和组合双精度数进行计算。

本章的示例代码演示了 x86-SSE 的不同版本。每个汇编语言函数的头部列出了所需的版本号。附录 C 列出了一些免费工具，这些工具可以帮助你检测你的 PC 中处理器和操作系统所能支持的 x86-SSE 版本。

9.1 组合浮点运算基础

本节的示例代码将演示如何进行基本的组合浮点运算，包括基本的算术运算、比较和类型转换。本章的几个示例都用到了 XmmVal 这个 C++ 联合体来帮助在 C++ 和汇编语言函数之间交换数据（见清单 9-1）。该联合体内的各成员对应于 x86-SSE 所支持的各种组合数据类型。联合体 XmmVal 还包含一些文本字符串格式化函数的声明。XmmVal.cpp 包含了 ToString_ 格式化函数的定义，该文件清单未在本书中列出，读者可以在代码包的 CommonFiles 子目录下找到它。

清单 9-1 XmmVal.h

```
#pragma once

#include "MiscDefs.h"

union XmmVal
{
    Int8 i8[16];
237 Int16 i16[8];
    Int32 i32[4];
    Int64 i64[2];
    Uint8 u8[16];
    Uint16 u16[8];
    Uint32 u32[4];
    Uint64 u64[2];
    float r32[4];
    double r64[2];

    char* ToString_i8(char* s, size_t len);
    char* ToString_i16(char* s, size_t len);
    char* ToString_i32(char* s, size_t len);
    char* ToString_i64(char* s, size_t len);

    char* ToString_u8(char* s, size_t len);
    char* ToString_u16(char* s, size_t len);
    char* ToString_u32(char* s, size_t len);
    char* ToString_u64(char* s, size_t len);
```

```
    char* ToString_x8(char* s, size_t len);
    char* ToString_x16(char* s, size_t len);
    char* ToString_x32(char* s, size_t len);
    char* ToString_x64(char* s, size_t len);

    char* ToString_r32(char* s, size_t len);
    char* ToString_r64(char* s, size_t len);
};
```

9.1.1 组合浮点算术运算

第一个 x86-SSE 组合浮点示例程序是 SsePackedFloatingPointArithmetic。这个程序展示了如何对 x86-SSE 组合浮点类作数进行基本的算术运算。清单 9-2 和清单 9-3 分别列出了相应的 C++ 和 x86-32 汇编语言源代码。

清单 9-2 SsePackedFloatingPointArithmetic.cpp

```
#include "stdafx.h"
#include "XmmVal.h"
#define _USE_MATH_DEFINES
#include <math.h>

extern "C" void SsePackedFpMath32 (const XmmVal* a, const XmmVal* b, XmmVal c[8]);
extern "C" void SsePackedFpMath64 (const XmmVal* a, const XmmVal* b, XmmVal c[8]);

void SsePackedFpMath32(void)
{
    __declspec(align(16)) XmmVal a;
    __declspec(align(16)) XmmVal b;
    __declspec(align(16)) XmmVal c[8];
    char buff[256];

    a.r32[0] = 36.0f;
    a.r32[1] = (float)(1.0 / 32.0);
    a.r32[2] = 2.0f;
    a.r32[3] = 42.0f;

    b.r32[0] = -(float)(1.0 / 9.0);
    b.r32[1] = 64.0f;
    b.r32[2] = -0.0625f;
    b.r32[3] = 8.666667f;

    SsePackedFpMath32_(&a, &b, c);
    printf("\nResults for SsePackedFpMath32_\n");
    printf("a:          %s\n", a.ToString_r32(buff, sizeof(buff)));
    printf("b:          %s\n", b.ToString_r32(buff, sizeof(buff)));
    printf("\n");
    printf("addps:      %s\n", c[0].ToString_r32(buff, sizeof(buff)));
    printf("subps:      %s\n", c[1].ToString_r32(buff, sizeof(buff)));
    printf("mulps:      %s\n", c[2].ToString_r32(buff, sizeof(buff)));
    printf("divps:      %s\n", c[3].ToString_r32(buff, sizeof(buff)));
    printf("sbops a:    %s\n", c[4].ToString_r32(buff, sizeof(buff)));
    printf("sqrtps a:   %s\n", c[5].ToString_r32(buff, sizeof(buff)));
    printf("minps:      %s\n", c[6].ToString_r32(buff, sizeof(buff)));
    printf("maxps:      %s\n", c[7].ToString_r32(buff, sizeof(buff)));
}

void SsePackedFpMath64(void)
{
```

```
    _declspec(align(16)) XmmVal a;
    _declspec(align(16)) XmmVal b;
    _declspec(align(16)) XmmVal c[8];
    char buff[256];

    a.r64[0] = 2.0;
    a.r64[1] = M_PI;
    b.r64[0] = M_E;
    b.r64[1] = -M_1_PI;

    SsePackedFpMath64_(&a, &b, c);
    printf("\nResults for SsePackedFpMath64 \n");
    printf("a: %s\n", a.ToString_r64(buff, sizeof(buff)));
239 printf("b: %s\n", b.ToString_r64(buff, sizeof(buff)));
    printf("\n");
    printf("addpd:    %s\n", c[0].ToString_r64(buff, sizeof(buff)));
    printf("subpd:    %s\n", c[1].ToString_r64(buff, sizeof(buff)));
    printf("mulpd:    %s\n", c[2].ToString_r64(buff, sizeof(buff)));
    printf("divpd:    %s\n", c[3].ToString_r64(buff, sizeof(buff)));
    printf("abspd a:  %s\n", c[4].ToString_r64(buff, sizeof(buff)));
    printf("sqrtpd a: %s\n", c[5].ToString_r64(buff, sizeof(buff)));
    printf("minpd:    %s\n", c[6].ToString_r64(buff, sizeof(buff)));
    printf("maxpd:    %s\n", c[7].ToString_r64(buff, sizeof(buff)));
}

int _tmain(int argc, _TCHAR* argv[])
{
    SsePackedFpMath32();
    SsePackedFpMath64();
}
```

清单 9-3 SsePackedFloatingPointArithmetic_.asm

```
        .model flat,c
        .const

; 用于计算浮点绝对值的掩码值
        align 16
Pfp32Abs    dword 7fffffffh,7fffffffh,7fffffffh,7fffffffh
Pfp64Abs    qword 7fffffffffffffffh,7fffffffffffffffh
        .code

; extern "C" void SsePackedFpMath32_(const XmmVal* a, const XmmVal* b,↩
XmmVal c[8]);
;
; 函数说明：演示基本的组合单精度浮点数学计算
;
; SSE 版本：SSE

SsePackedFpMath32_ proc
        push ebp
        mov ebp,esp

; 加载组合单精度浮点型数据
        mov eax,[ebp+8]                 ;eax = 'a'
        mov ecx,[ebp+12]                ;ecx = 'b'
        mov edx,[ebp+16]                ;edx = 'c'
        movaps xmm0,[eax]               ;xmm0 = *a
240     movaps xmm1,[ecx]               ;xmm1 = *b

; 组合单精度浮点型加法
```

```
        movaps xmm2,xmm0
        addps xmm2,xmm1
        movaps [edx+0],xmm2

; 组合单精度浮点型减法
        movaps xmm2,xmm0
        subps xmm2,xmm1
        movaps [edx+16],xmm2

; 组合单精度浮点型乘法
        movaps xmm2,xmm0
        mulps xmm2,xmm1
        movaps [edx+32],xmm2

; 组合单精度浮点型除法
        movaps xmm2,xmm0
        divps xmm2,xmm1
        movaps [edx+48],xmm2

; 求组合单精度浮点型数值绝对值
        movaps xmm2,xmm0
        andps xmm2,xmmword ptr [Pfp32Abs]
        movaps [edx+64],xmm2

; 计算组合单精度浮点型数值平方根
        sqrtps xmm2,xmm0
        movaps [edx+80],xmm2

; 取组合单精度浮点数的最小值
        movaps xmm2,xmm0
        minps xmm2,xmm1
        movaps [edx+96],xmm2

; 求组合单精度浮点数的最大值
        maxps xmm0,xmm1
        movaps [edx+112],xmm0

        pop ebp
        ret
SsePackedFpMath32_ endp

; extern "C" void SsePackedFpMath64(const XmmVal* a, const XmmVal* b,↩
XmmVal c[8]);
;
; 函数说明：演示基本的组合双精度浮点数学计算
;
; SSE 版本：SSE2                                                    241

SsePackedFpMath64_ proc
        push ebp
        mov ebp,esp

; 加载组合双精度浮点型数据
        mov eax,[ebp+8]                 ;eax = 'a'
        mov ecx,[ebp+12]                ;ecx = 'b'
        mov edx,[ebp+16]                ;edx = 'c'
        movapd xmm0,[eax]               ;xmm0 = *a
        movapd xmm1,[ecx]               ;xmm1 = *b

; 组合双精度浮点型加法
        movapd xmm2,xmm0
        addpd xmm2,xmm1
        movapd [edx+0],xmm2
```

```
; 组合双精度浮点型减法
        movapd xmm2,xmm0
        subpd xmm2,xmm1
        movapd [edx+16],xmm2

; 组合双精度浮点型乘法
        movapd xmm2,xmm0
        mulpd xmm2,xmm1
        movapd [edx+32],xmm2

; 组合双精度浮点型除法
        movapd xmm2,xmm0
        divpd xmm2,xmm1
        movapd [edx+48],xmm2

; 求组合双精度浮点数的绝对值
        movapd xmm2,xmm0
        andpd xmm0,xmmword ptr [Pfp64Abs]
        movapd [edx+64],xmm2

; 求组合双精度浮点数的平方根
        sqrtpd xmm2,xmm0
        movapd [edx+80],xmm2

; 求两个双精度浮点数的最小值
        movapd xmm2,xmm0
        minpd xmm2,xmm1
        movapd [edx+96],xmm2

; 求组合双精度浮点数的最大值
        maxpd xmm0,xmm1
        movapd [edx+112],xmm0

        pop ebp
        ret
SsePackedFpMath64_ endp
        end
```

C++ 源文件 SsePackedFloatingPointArithmetic.cpp（见清单 9-2）包含一个名为 SsePackedFpMath32 的函数，这个函数定义了一组 XmmVal 实例，并用组合单精度浮点数初始化这些实例。请注意，这些 XmmVal 变量是通过使用 Visual C++ 的扩展属性 declspec(align(16)) 定义的，这使它们都是按 16 字节对齐的。汇编语言函数 SsePackedFpMath32_ 执行组合算术计算并把结果返回到指定的数组，最后把结果显示出来。SsePackedFpMath64 和 SsePackedFpMath64 执行一系列类似的操作，但针对的是组合双精度浮点数。

SsePackedFpMath32_ 和 SsePackedFpMath64_ 函数定义在 SsePackedFloatingPointArithmetic_asm 中。这两个函数演示了通用 x86-SSE 组合单精度浮点和组合双精度浮点指令的用法。请注意 movaps（Move Aligned Packed Single-Precision）和 movapd（Double-Precision Floating-Point Value）这两个指令要求源操作数和目标操作数在内存中要 16 字节对齐。MXCSR.RC 的舍入模式同样适用于组合浮点算术运算。输出 9-1 显示了执行 SsePackedFloatingPointArithmetic 的结果。

输出 9-1 示例程序 SsePackedFloatingPointArithmetic

```
Results for SsePackedFpMath32_
a:          86.000000     0.043250 |     2.000000    42.000000
```

```
b:            -0.111111    64.000000 |    -0.062500     8.666667

addps:        35.888889    64.031250 |     1.937500    50.666668
subps:        36.111111   -63.968750 |     2.062500    33.333332
mulps:        -4.000000     2.000000 |    -0.125000   364.000000
divps:      -324.000000     0.000488 |    32.000000     4.846154
absps a:      36.000000     0.031250 |     2.000000    42.000000
sqrtps a:      6.000000     0.176777 |     1.414214     6.480741
minps:        -0.111111     0.031250 |    -0.062500     8.666667
maxps:        36.000000    64.000000 |     2.000000    42.000000

Results for SsePackedDpMath64_
a:                    2.000000000000 |          3.141592653590
b:                    2.718281828459 |         -0.318309886184

addpd:                4.718281828459 |          2.823282767406
subpd:               -0.718281828459 |          3.459902539774
mulpd:                5.436563656918 |         -1.000000000000
divpd:                0.735758882343 |         -9.869604401089
abspd a:              2.000000000000 |          3.141592653590
sqrtpd a:             1.414213562373 |          1.772453850906
minpd:                2.000000000000 |         -0.318309886184
maxpd:                2.718281828459 |          3.141592653590
```

243

9.1.2 组合浮点数的比较

在第 8 章我们学习了通过 comiss 和 comisd 指令分别对标量单精度浮点数和标量双精度浮点数进行比较的方法，在本章我们将学习通过 cmpps 和 cmppd（组合单精度和双精度浮点值比较）这两条指令对两个组合浮点数进行 SIMD 比较。同样，它们通过载入一个双字的掩码值到 XMM 寄存器报告比较结果，而不是通过设置 EFLAGS 寄存器的状态位来报告。

与标量浮点数比较不同的是，组合浮点数比较指令（cmpps 和 cmppd）还需要第三个操作数来表示比较类型。这两个指令的语法如下：

cmppX CmpOp1, CmpOp2, PredOp

其中 cmppX 指 cmpps 或 cmppd；CmpOp1 是第一个源操作数且必须是一个 XMM 寄存器；CmpOp2 是第二个源操作数，可以是一个 XMM 寄存器或者是在内存中的一个 128 位的组合操作数；PredOp 是一个立即数，用于指定比较运算的类型。表 9-1 列出了 x86 SSE 支持的所有 8 种类型。组合比较的结果以一个双字掩码保存在 CmpOp1 里，所有为 1 的位表示比较结果为 true，为 0 的位表示比较结果为 false。考虑到使用立即数表示比较类型不便记忆，多数汇编编译器（包括 MASM）通过一系列伪指令来提高代码的可读性。表 9-1 列出了各比较类型码对应的伪指令。

表 9-1 cmpps 和 cmppd 指令的比较类型信息

类型码	类 型	说 明	伪指令
0	EQ	CmpOp1 == CmpOp2	cmpeqp(s|d)
1	LT	CmpOp1 < CmpOp2	cmpltp(s|d)
2	LE	CmpOp1 <= CmpOp2	cmplep(s|d)
3	UNORD	CmpOp1 && CmpOp2 are unordered	cmpunordp(s|d)
4	NEQ	!(CmpOp1 == CmpOp2)	cmpneqp(s|d)
5	NLT	!(CmpOp1 < CmpOp2)	cmpnltp(s|d)
6	NLE	!(CmpOp1 <= CmpOp2)	cmpnlep(s|d)
7	ORD	CmpOp1 && CmpOp2 are ordered	cmpordp(s|d)

244

请注意，表9-1里的NLT（不小于）谓词与GE（大于或等于）相同，NLE（不小于或等于）与GT（大于）相同。

图9-1描述了指令cmpps xmm0, xmm1, 0（或cmpeqps xmm0, xmm1）的执行。在这个例子中，保存在XMM0里的单精度浮点数与保存在XMM1里的单精度浮点数进行比较，判断它们是否相等。如果相等，则0xFFFFFFFF会被写入XMM0的相应位置，否则0x00000000会被写入XMM0的相应位置。

cmpps xmm0,xmm1,0（或cmpeqps xmm0, xmm1）

4.125	2.375	−72.5	44.125	xmm1
8.625	2.375	−72.5	15.875	xmm0
0x00000000	0xFFFFFFFF	0xFFFFFFFF	0x00000000	xmm0

图9-1 cmpps指令的执行

示例程序SsePackedFloatingPointCompare演示了使用cmpps和cmppd指令进行组合浮点数比较的方法。清单9-4和清单9-5分别列出了相应的C++和x86-32汇编语言源代码。

清单9-4 SsePackedFloatingPointCompare.cpp

```
#include "stdafx.h"
#include "XmmVal.h"
#include <limits>
using namespace std;

extern "C" void SsePfpCompareFloat_(const XmmVal* a, const XmmVal* b, XmmVal c[8]);

const char* CmpStr[8] =
{
    "EQ", "LT", "LE", "UNORDERED", "NE", "NLT", "NLE", "ORDERED"
};

void SsePfpCompareFloat(void)
{
    __declspec(align(16)) XmmVal a;
    __declspec(align(16)) XmmVal b;
    __declspec(align(16)) XmmVal c[8];
    char buff[256];

    a.r32[0] = 2.0;         b.r32[0] = 1.0;
    a.r32[1] = 7.0;         b.r32[1] = 22.0;
    a.r32[2] = -6.0;        b.r32[2] = -6.0;
245 a.r32[3] = 4.0;         b.r32[3] = 8.0;

    for (int i = 0; i < 2; i++)
    {
        if (i == 1)
            a.r32[0] = numeric_limits<float>::quiet_NaN();

        SsePfpCompareFloat_(&a, &b, c);

        printf("\nResults for SsePfpCompareFloat_ (Iteration %d)\n", i);
        printf("a: %s\n", a.ToString_r32(buff, sizeof(buff)));
        printf("b: %s\n", b.ToString_r32(buff, sizeof(buff)));
        printf("\n");

        for (int j = 0; j < 8; j++)
```

```
        {
            char* s = c[j].ToString_x32(buff, sizeof(buff));
            printf("%10s: %s\n", CmpStr[j], s);
        }
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    SsePfpCompareFloat();
    return 0;
}
```

清单 9-5 SsePackedFloatingPointCompare_.asm

```
        .model flat,c
        .code

; extern "C" void SsePfpCompareFloat_(const XmmVal* a, const XmmVal* b,↩
XmmVal c[8]);
;
; 函数说明：下面的代码将演示 cmpps 的使用方法
;
; SSE 版本：SSE2

SsePfpCompareFloat_ proc
        push ebp
        mov ebp,esp

        mov eax,[ebp+8]                 ;eax = 'a' 的位置
        mov ecx,[ebp+12]                ;ecx = 'b' 的位置
        mov edx,[ebp+16]                ;edx = 'c' 的位置
        movaps xmm0,[eax]               ;把 'a' 载入到 xmm0
        movaps xmm1,[ecx]               ;把 'b' 载入到 xmm1

; 等于比较
        movaps xmm2,xmm0
        cmpeqps xmm2,xmm1
        movdqa [edx],xmm2

; 小于比较
        movaps xmm2,xmm0
        cmpltps xmm2,xmm1
        movdqa [edx+16],xmm2

; 小于或等于比较
        movaps xmm2,xmm0
        cmpleps xmm2,xmm1
        movdqa [edx+32],xmm2

; 无序（UNORDERD）比较
        movaps xmm2,xmm0
        cmpunordps xmm2,xmm1
        movdqa [edx+48],xmm2

; 不等于比较
        movaps xmm2,xmm0
        cmpneqps xmm2,xmm1
        movdqa [edx+64],xmm2

; 不小于比较
```

246

```
        movaps xmm2,xmm0
        cmpnltps xmm2,xmm1
        movdqa [ecx+80],xmm2

; 不小于或等于比较
        movaps xmm2,xmm0
        cmpnleps xmm2,xmm1
        movdqa [ecx+96],xmm2

; 有序(ORDERED)比较
        movaps xmm2,xmm0
        cmpordps xmm2,xmm1
        movdqa [ecx+112],xmm2

        pop ebp
        ret
SsePfpCompareFloat_ endp
        end
```

[247]

C++源代码（清单9-4）包含一个简单的函数，用于初始化一组组合浮点变量，调用汇编语言函数进行比较，然后打印结果。需要特别注意的是，在第二次for循环时，为了验证有序和无序比较操作是否正确，NaN值被拷贝到XmmVal变量a。汇编语言函数SsePfpCompareFloat_（清单9-5）先是把组合浮点变量a和b分别载入寄存器XMM0和XMM1，然后依次执行八种类型的比较并把比较结果保存到相应的数组里。汇编语言函数使用了比较为指令以便提高代码的可读性（强烈建议读者使用伪指令，要知道，x86-AVX版本的cmpps支持32种比较类型，而不是像x86-SSE只支持8种，要记住所有32种类型实在是很有挑战的）。输出9-2展示了示例程序SsePackedFloatingPointCompare的执行结果。

清单9-2 SsePackedFloatingPointCompare

```
Results for SsePfpCompareFloat_ (Iteration 0)
a:     2.000000     7.000000 |    -6.000000     3.000000
b:     1.000000    12.000000 |    -6.000000     8.000000

        EQ: 00000000 00000000 | FFFFFFFF 00000000
        LT: 00000000 FFFFFFFF | 00000000 FFFFFFFF
        LE: 00000000 FFFFFFFF | FFFFFFFF FFFFFFFF
 UNORDERED: 00000000 00000000 | 00000000 00000000
        NE: FFFFFFFF FFFFFFFF | 00000000 FFFFFFFF
       NLT: FFFFFFFF 00000000 | FFFFFFFF 00000000
       NLE: FFFFFFFF 00000000 | 00000000 00000000
   ORDERED: FFFFFFFF FFFFFFFF | FFFFFFFF FFFFFFFF

Results for SsePfpCompareFloat_ (Iteration 1)
a:     1.#QNAN0     7.000000 |    -6.000000     3.000000
b:     1.000000    12.000000 |    -6.000000     8.000000

        EQ: 00000000 00000000 | FFFFFFFF 00000000
        LT: 00000000 FFFFFFFF | 00000000 FFFFFFFF
        LE: 00000000 FFFFFFFF | FFFFFFFF FFFFFFFF
 UNORDERED: FFFFFFFF 00000000 | 00000000 00000000
        NE: FFFFFFFF FFFFFFFF | 00000000 FFFFFFFF
       NLT: FFFFFFFF 00000000 | FFFFFFFF 00000000
       NLE: FFFFFFFF 00000000 | 00000000 00000000
   ORDERED: 00000000 FFFFFFFF | FFFFFFFF FFFFFFFF
```

9.1.3 组合浮点数的类型转换

下一个 x86-SSE 组合浮点示例程序是 SsePackedFloatingPointConversions。这个程序演示了如何把一个组合双字有符号整数转换成一个组合单精度浮点数或组合双精度浮点数以及反向转换。它同时演示了组合单精度浮点数和组合双精度浮点数之间的相互转换。清单 9-6 和清单 9-7 分别显示了示例程序 SsePackedFloatingPointConversions 的 C++ 和汇编语言的源代码。 248

清单 9-6 SsePackedFloatingPointConversions.cpp

```
#include "stdafx.h"
#include "XmmVal.h"
#define _USE_MATH_DEFINES
#include <math.h>

// CvtOp 成员的顺序必须与 SsePackedFloatingPointConversions_.asm 中定义的相一致
enum CvtOp : unsigned int
{
    Cvtdq2ps,           // 组合有符号双字 --> 组合单精度浮点数
    Cvtdq2pd,           // 组合有符号双字 --> 组合双精度浮点数
    Cvtps2dq,           // 组合单精度浮点数 --> 组合有符号双字
    Cvtpd2dq,           // 组合双精度浮点数 --> 组合有符号双字
    Cvtps2pd,           // 组合单精度浮点数 --> 组合双精度浮点数
    Cvtpd2ps            // 组合双精度浮点数 --> 组合单精度浮点数
};

extern "C" void SsePfpConvert (const XmmVal* a, XmmVal* b, CvtOp cvt_op);

void SsePfpConversions32(void)
{
    _declspec(align(16)) XmmVal a;
    _declspec(align(16)) XmmVal b;
    char buff[256];

    a.i32[0] = 10;
    a.i32[1] = -500;
    a.i32[2] = 600;
    a.i32[3] = -1024;
    SsePfpConvert (&a, &b, CvtOp::Cvtdq2ps);
    printf("\nResults for CvtOp::Cvtdq2ps\n");
    printf("  a: %s\n", a.ToString_i32(buff, sizeof(buff)));
    printf("  b: %s\n", b.ToString_r32(buff, sizeof(buff)));

    a.r32[0] = 1.0f / 3.0f;
    a.r32[1] = 2.0f / 3.0f;
    a.r32[2] = -a.r32[0] * 2.0f;
    a.r32[3] = -a.r32[1] * 2.0f;
    SsePfpConvert (&a, &b, CvtOp::Cvtps2dq);
    printf("\nResults for CvtOp::Cvtps2dq\n");
    printf("  a: %s\n", a.ToString_r32(buff, sizeof(buff)));
    printf("  b: %s\n", b.ToString_i32(buff, sizeof(buff)));

    // cvtps2pd 转换 a 的两个低阶双精度浮点值
    a.r32[0] = 1.0f / 7.0f;
    a.r32[1] = 2.0f / 9.0f;
    a.r32[2] = 0;
    a.r32[3] = 0;
    SsePfpConvert (&a, &b, CvtOp::Cvtps2pd);
    printf("\nResults for CvtOp::Cvtps2pd\n");
    printf("  a: %s\n", a.ToString_r32(buff, sizeof(buff)));
    printf("  b: %s\n", b.ToString_r64(buff, sizeof(buff)));
```

249

```
}

void SsePfpConversions64(void)
{
    _declspec(align(16)) XmmVal a;
    _declspec(align(16)) XmmVal b;
    char buff[256];

    // cvtdq2pd 转换 a 的两个低位双字整数
    a.i32[0] = 10;
    a.i32[1] = -20;
    a.i32[2] = 0;
    a.i32[3] = 0;
    SsePfpConvert_(&a, &b, CvtOp::Cvtdq2pd);
    printf("\nResults for CvtOp::Cvtdq2pd\n");
    printf("  a: %s\n", a.ToString_i32(buff, sizeof(buff)));
    printf("  b: %s\n", b.ToString_r64(buff, sizeof(buff)));

    // cvtpd2dq 把 b 的两个低位双字设成 0
    a.r64[0] = M_PI;
    a.r64[1] = M_E;
    SsePfpConvert_(&a, &b, CvtOp::Cvtpd2dq);
    printf("\nResults for CvtOp::Cvtpd2dq\n");
    printf("  a: %s\n", a.ToString_r64(buff, sizeof(buff)));
    printf("  b: %s\n", b.ToString_i32(buff, sizeof(buff)));

    // cvtpd2ps 把 b 的两个单位单精度浮点数值设成 0
    a.r64[0] = M_SQRT2;
    a.r64[1] = M_SQRT1_2;
    SsePfpConvert_(&a, &b, CvtOp::Cvtpd2ps);
    printf("\nResults for CvtOp::Cvtpd2ps\n");
    printf("  a: %s\n", a.ToString_r64(buff, sizeof(buff)));
    printf("  b: %s\n", b.ToString_r32(buff, sizeof(buff)));
250 }

int _tmain(int argc, _TCHAR* argv[])
{
    SsePfpConversions32();
    SsePfpConversions64();
    return 0;
}
```

清单 9-7 SsePackedFloatingPointConversions_.asm

```
        .model flat,c
        .code

; extern "C" void SsePfpConvert_(const XmmVal* a, XmmVal* b, CvtOp cvt_op);
;
; 函数说明：演示组合浮点转换指令的使用方法
;
; SSE 版本：SSE2

SsePfpConvert_ proc
        push ebp
        mov ebp,esp

; 加载参数并确保 cvt_op 是有效的
        mov eax,[ebp+8]                 ;eax = 'a'
        mov ecx,[ebp+12]                ;ecx = 'b'
        mov edx,[ebp+16]                ;edx =cvt_op
        cmp edx,CvtOpTableCount
```

```
        jae BadCvtOp
        jmp [CvtOpTable+edx*4]              ;跳转到相应的转换处理代码

; 组合有符号双字整数 --> 组合单精度浮点数
SseCvtdq2ps:
        movdqa xmm0,[eax]
        cvtdq2ps xmm1,xmm0
        movaps [ecx],xmm1
        pop ebp
        ret

; 组合有符号双字整数 --> 组合双精度浮点数
SseCvtdq2pd:
        movdqa xmm0,[eax]
        cvtdq2pd xmm1,xmm0
        movapd [ecx],xmm1
        pop ebp
        ret                                                    251

; 组合单精度浮点数 --> 组合有符号双字整数
SseCvtps2dq:
        movaps xmm0,[eax]
        cvtps2dq xmm1,xmm0
        movdqa [ecx],xmm1
        pop ebp
        ret

; 组合双精度浮点数 --> 组合有符号双字整数
SseCvtpd2dq:
        movapd xmm0,[eax]
        cvtpd2dq xmm1,xmm0
        movdqa [ecx],xmm1
        pop ebp
        ret

; 组合单精度浮点数 --> 组合双精度浮点数
SseCvtps2pd:
        movaps xmm0,[eax]
        cvtps2pd xmm1,xmm0
        movapd [ecx],xmm1
        pop ebp
        ret

; 组合双精度浮点数 --> 组合单精度浮点数
SseCvtpd2ps:
        movapd xmm0,[eax]
        cvtpd2ps xmm1,xmm0
        movaps [ecx],xmm1
        pop ebp
        ret

BadCvtOp:
        pop ebp
        ret

; 下面的函数序列与 SsePackedFloatingPointConversions.cpp 里定义的 CvtOp 一致
        align 4
CvtOpTable  dword SseCvtdq2ps, SseCvtdq2pd, SseCvtps2dq
            dword SseCvtpd2dq, SseCvtps2pd, SseCvtpd2ps
CvtOpTableCount equ ($ - CvtOpTable) / size dword
SsePfpConvert_ endp
        end
```

252

C++源文件 SsePackedFloatingPointConversions.cpp（见清单9-6）定义了一个名为CvtOp的枚举类型，CvtOp列举了所有的6种合法的转换操作符。CvtOp里的几个枚举值的名字（以及相应的汇编语言助记符）看起来有些让人困惑，要注意其中的dq代表有符号双字整数（doubleword signed integer），而不是double quadword。此外，还要注意：实际的成员转换依赖于数据的类型，比如CvtOp::Cvtps2pd（或cvtps2pd指令）把两个低位的单精度浮点数转换成两个双精度浮点数。在从组合双精度浮点数向单精度浮点数或双字有符号整数转换时，目标操作数的高位被设成0。

汇编语言文件 SsePackedFloatingPointConversions_asm（见清单9-7）使用了跳转表来选择相应的转换指令。

在每个执行类型转换的代码块内部，指令movaps、movapd或者movdqa（Move Aligned Double Quadword，对齐双四字移动）被用来把组合数据从内存中载入，或者把组合数据复制到内存。这些指令都需要内存操作数在内存中正确地对齐。组合转换指令使用MXCSR指定的舍入模式（VisualC++采用的缺省的舍入模式是就近舍入）。如果非法操作异常位（MXCSR.IM）被屏蔽且指定的转换无法执行，某些组合转换指令会设置非法操作标志位（MXCSR.IE）。输出9-3显示了示例程序SsePackedFloatingPointConversions的执行结果。

输出9-3 示例程序SsePackedFloatingPointConversions

```
Results for CvtOp::Cvtdq2ps
  a:          10         -500 |         600        -1024
  b:    10.000000  -500.000000 |  600.000000 -1024.000000

Results for CvtOp::Cvtps2dq
  a:    0.333333    0.666667 |   -0.666667   -1.333333
  b:           0           1 |          -1          -1

Results for CvtOp::Cvtps2pd
  a:    0.142857    0.222222 |    0.000000    0.000000
  b:    0.142857142943       |    0.222222223878

Results for CvtOp::Cvtdq2pd
  a:          10         -20 |           0           0
  b:   10.000000000000       |  -20.000000000000

Results for CvtOp::Cvtpd2dq
  a:   3.141592653590         |   2.718281828459
  b:           3           3 |           0           0

Results for CvtOp::Cvtpd2ps
  a:   1.414213562373         |   0.707106781187
  b:    1.414214    0.707107 |    0.000000    0.000000
```

253

9.2 高级组合浮点编程

本节的示例程序将演示使用x86-SSE组合浮点指令集对组合单精度和组合双精度浮点数进行高级数学计算。第一个示例程序将演示对双精度浮点数组进行SIMD计算，而在第二个示例程序里，我们将学习如何使用x86-SSE的计算资源提高4×4矩阵操作算法的性能。

9.2.1 组合浮点数最小二乘法

在第4章，我们通过一个示例程序学习了通过x87 FPU计算最小二乘法拟合直线的

斜率和的截距的方法，而本节的示例程序 SsePackedFloatingPointLeastSquares 将演示使用 x86-SSE 的组合双精度浮点算术指令进行同样的计算。清单 9-8 和清单 9-9 分别列出了相应的 C++ 和 x86-32 汇编语言源代码。

清单 9-8 SsePackedFloatingPointLeastSquares.cpp

```
#include "stdafx.h"
#include <stddef.h>
#include <math.h>

extern "C" double lsEpsilon = 1.0e-12;
extern "C" bool SsePfpLeastSquares_(const double* x, const double* y,↩
int n, double* m, double* b);

bool SsePfpLeastSquaresCpp(const double* x, const double* y, int n,↩
double* m, double* b)
{
    if (n < 2)
        return false;

    // 确保 x 和 y 被正确对齐
    if ((((uintptr_t)x & 0xf) != 0) || (((uintptr_t)y & 0xf) != 0))
        return false;

    double sum_x = 0, sum_y = 0.0, sum_xx = 0, sum_xy = 0.0;

    for (int i = 0; i < n; i++)
    {
        sum_x += x[i];
        sum_xx += x[i] * x[i];
        sum_xy += x[i] * y[i];
        sum_y += y[i];
    }

    double denom = n * sum_xx - sum_x * sum_x;

    if (fabs(denom) >= lsEpsilon)
    {
        *m = (n * sum_xy - sum_x * sum_y) / denom;
        *b = (sum_xx * sum_y - sum_x * sum_xy) / denom;
        return true;
    }
    else
    {
        *m = *b = 0.0;
        return false;
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    const int n = 11;
    __declspec(align(16)) double x[n] = {10, 13, 17, 19, 23, 7, 35, 51,↩
89, 92, 99};
    __declspec(align(16)) double y[n] = {1.2, 1.1, 1.8, 2.3, 1.9, 0.5,↩
3.1, 5.5, 8.4, 9.7, 10.4};

    double m1, m2, b1, b2;
    bool rc1 = SsePfpLeastSquaresCpp(x, y, n, &m1, &b1);
    bool rc2 = SsePfpLeastSquares_(x, y, n, &m2, &b2);
```

234

```
    printf("\nResults from SsePackedFloatingPointLeastSquaresCpp\n");
    printf("  rc:        %12d\n", rc1);
    printf("  slope:     %12.8lf\n", m1);
    printf("  intercept: %12.8lf\n", b1);
    printf("\nResults from SsePackedFloatingPointLeastSquares_\n");
    printf("  rc:        %12d\n", rc2);
    printf("  slope:     %12.8lf\n", m2);
    printf("  intercept: %12.8lf\n", b2);
    return 0;
}
```

清单 9-9 SsePackedFloatingPointLeastSquares_.asm

```
        .model flat,c
        extern LsEpsilon:real8
        .const
PackedFp64Abs qword 7fffffffffffffffh,7fffffffffffffffh
        .code

; extern "C" bool SsePfpLeastSquares_(const double* x, const double* y,
int n, double* m, double* b);
;
; 函数说明：计算使用最小二乘法拟合直线的斜率和截距
;
; 返回值：0 - 错误（n 非法或者数组未正确对齐）
;         1 - 正确
;
; SSE 版本：SSE3

SsePfpLeastSquares_ proc
        push ebp
        mov ebp,esp
        push ebx

; 载入并验证参数
        xor eax,eax                     ;设置错误返回码
        mov ebx,[ebp+8]                 ;ebx = 'x'
        test ebx,0fh
        jnz Done                        ;如果 x 未对齐就转
        mov edx,[ebp+12]                ;edx ='y'
        test edx,0fh
        jnz Done                        ;如果 y 未对齐就转
        mov ecx,[ebp+16]                ;ecx = n
        cmp ecx,2
        jl Done                         ;就转，如果 n < 2

; 初始化求和寄存器
        cvtsi2sd xmm3,ecx               ;xmm3 = 双精度浮点数 n
        mov eax,ecx
        and ecx,0fffffffeh              ;ecx = n / 2 * 2
        and eax,1                       ;eax = n % 2

        xorpd xmm4,xmm4                 ;sum_x（四字）
        xorpd xmm5,xmm5                 ;sum_y（四字）
        xorpd xmm6,xmm6                 ;sum_xx（四字）
        xorpd xmm7,xmm7                 ;sum_xy（四字）

; 计算和变量，每次处理两个值
@@:     movapd xmm0,xmmword ptr [ebx]   ;载入下两个 x 值
        movapd xmm1,xmmword ptr [edx]   ;载入下两个 y 值
        movapd xmm2,xmm0                ;拷贝 x
```

255

```
        addpd xmm4,xmm0                 ;更新 sum_x
        addpd xmm5,xmm1                 ;更新 sum_y
        mulpd xmm0,xmm0                 ;计算 x * x                    256
        addpd xmm6,xmm0                 ;更新 sum_xx
        mulpd xmm2,xmm1                 ;计算 x * y
        addpd xmm7,xmm2                 ;更新 sum_xy

        add ebx,16                      ;ebx = 下一个 x 数组值
        add edx,16                      ;edx = 下一个 y 数组值
        sub ecx,2                       ;调整统计数
        jnz @@                          ;重复，直到全部完成

; 用最终的 x 和 y 值更新和变量（当 n 是奇数时）
        or eax,eax
        jz CalcFinalSums                ;如果 n 是偶数则转
        movsd xmm0,real8 ptr [ebx]      ;载入最终的 x
        movsd xmm1,real8 ptr [edx]      ;载入最终的 y
        movsd xmm2,xmm0

        addsd xmm4,xmm0                 ;更新 sum_x
        addsd xmm5,xmm1                 ;更新 sum_y
        mulsd xmm0,xmm0                 ;计算 x * x
        addsd xmm6,xmm0                 ;更新 sum_xx
        mulsd xmm2,xmm1                 ;计算 x * y
        addsd xmm7,xmm2                 ;更新 sum_xy

; 计算最终的和
CalcFinalSums:
        haddpd xmm4,xmm4                ;xmm4[63:0] = 最终的 sum_x
        haddpd xmm5,xmm5                ;xmm5[63:0] = 最终的 sum_y
        haddpd xmm6,xmm6                ;xmm6[63:0] = 最终的 sum_xx
        haddpd xmm7,xmm7                ;xmm7[63:0] = 最终的 sum_xy

; 计算分母并确保其合法
; 分母 = n * sum_xx - sum_x * sum_x
        movsd xmm0,xmm3                 ;n
        movsd xmm1,xmm4                 ;sum_x
        mulsd xmm0,xmm6                 ;n * sum_xx
        mulsd xmm1,xmm1                 ;sum_x * sum_x
        subsd xmm0,xmm1                 ;xmm0 = denom
        movsd xmm2,xmm0
        andpd xmm2,xmmword ptr [PackedFp64Abs] ;xmm2 = fabs(分母)
        comisd xmm2,real8 ptr [LsEpsilon]
        jb BadDenom                     ;如果分母 < fabs(分母)则转

; 计算并保存斜率
; 斜率 = (n * sum_xy - sum_x * sum_y) / 分母
        movsd xmm1,xmm4                 ;sum_x
        mulsd xmm3,xmm7                 ;n * sum_xy
        mulsd xmm4,xmm5                 ;sum_x * sum_y                 257
        subsd xmm3,xmm1                 ;slope_numerator
        divsd xmm3,xmm0                 ;xmm3 = 最终的斜率
        mov edx,[ebp+20]                ;edx = 'n'
        movsd real8 ptr [edx],xmm3      ;保存斜率

; 计算并保存截距
; 截距 = (sum_xx * sum_y - sum_x * sum_xy) / 分母
        mulsd xmm6,xmm5                 ;sum_xx * sum_y
        mulsd xmm4,xmm7                 ;sum_x * sum_xy
        subsd xmm6,xmm4                 ;intercept_numerator
        divsd xmm6,xmm0                 ;xmm6 = 最终的截距
```

```
        mov edx,[ebp-24]                    ;edx = 'a'
        movsd real8 ptr [edx],xmm6          ;保存截距
        mov eax,1                           ;正确返回码

Done:   pop ebx
        pop ebp
        ret

; 把 n 和 b 设成 0
BadDemon:
        xor eax,eax                         ;设置错误返回码
        mov edx,[ebp+20]                    ;eax = 'n'
        mov [edx],eax
        mov [edx+4],eax                     ;*n = 0.0
        mov edx,[ebp+24]                    ;edx = 'b'
        mov [edx],eax
        mov [edx+4],eax                     ;*b = 0.0
        jmp Done

SsePfpLeastSquares_ endp
        end
```

C++ 源程序 SsePackedFloatingPointLeastSquares.cpp（见清单 9-8）包含的 C++ 函数 SsePfpLeastSquaresCpp 用来计算供比较的斜率和截距。函数 _tmain 定义了 x 和 y 两个测试数组，这两个数组使用了扩展属性 __declspec(align(16))，告诉编译器对数组进行 16 字节对齐。_tmain 的其他部分调用最小二乘法算法的两种实现并把结果显示出来。

汇编语言函数 SsePfpLeastSquares_（见清单 9-9）首先验证数组长度 n 和数组 x、y 是否正确对齐。指令 test ebx, 0fh 对数组 x 的地址与 0x0f 执行按位与操作，如果结果非 0，则说明数组没有正确对齐。然后对数组 y 进行同样的验证。参数验证之后是一系列用于初始
258 化的代码。cvtsi2sd xmm3, ecx 指令用于把 n 转换成双精度浮点数以便稍后使用。and ecx, 0fffffffeh 指令把 ECX 的值舍入到小于它的离它最近的偶数，且 EAX 被置成 0 或者 1（依赖于最初的 n 是偶数还是奇数）。这些调整是为了保证对数组 x 和 y 进行正确的组合计算。

回忆一下第 4 章，为了计算最小二乘法拟合直线的斜率和截距，我们需要计算四个中间的和：sum_x、sum_y、sum_xx 和 sum_xy。在本章的示例程序中，这几个值是使用组合双精度浮点计算出来的。这意味着在每次循环中，我们可以一次处理 x 和 y 数组中的两组数值，这样循环次数将减少一半。数组中下标为偶数的成员的和用 XMM4 ~ XMM7 的低位四字计算，而下标为奇数的成员的和用 XMM4 ~ XMM7 的高位四字计算。

在进入计算总和的循环之前，每个保存总和的寄存器被初始化为 0（使用 xorpd 指令）。在循环的开始，movapd xmm0,xmmword ptr[ebx] 指令分别把 x[i] 和 x[i+1] 复制到 XMM0 的低位四字和高位四字。接下来 movapd xmm1,xmmword ptr[edx] 指令分别把 y[i] 和 y[i+1] 复制到 XMM1 的低位四字和高位四字。之后通过一系列的 addpd 和 mulpd 指令来计算组合和值并将结果更新到 XMM4 ~ XMM7。然后数组指针寄存器 EBX 和 EDX 各加 16（两个双精度浮点数的长度），同时 ECX 保存的计数也被调整，以便进行下一个求和循环。计算总和的循环完成之后，需要判断原始输入的 n 是奇数还是偶数。如果是奇数，x 和 y 数组的最后一个数也要进行组合和的计算（注意，我们在这里使用标量计算指令 addsd 和 mulsd 来完成这个操作）。

在组合求和完成之后，我们用一系列 haddpd（Packed Double-FP Horizontal Add，组

合双精度浮点水平加法）指令来完成对最终的和的计算：sum_x、sum_y、sum_xx 和 sum_xy。每个 haddpd DesOp, SrcOp 指令计算方法如下：DesOp[63:0] =DesOp[127:64] + DesOp[63:0]，DesOp[127:64] = SrcOp[127:64] + SrcOp[63:0]。这些 haddpd 指令完成后，寄存器 XMM4 ~ XMM7 的低位四字保存了最终的总和（这些寄存器的高位四字同样也保存了这些总和，因为 haddpd 使用了同样的源操作数和目标操作数）。然后要计算出分母的值，并对其进行验证以确保它大于等于 LsEpsilon（小于 LsEpsilon 的值被认为太接近 0，所以为非法的值）。分母被验证通过之后，就可以使用简单的 x86-SSE 标量计算指令计算出斜率和截距了。输出 9-4 显示了示例程序 SsePackedFloatingPointLeastSquares 的执行结果。

输出 9-4 示例程序 SsePackedFloatingPointLeastSquares

```
Results from SsePackedFloatingPointLeastSquaresCpp
  xc:              1
  slope:       0.10424641
  intercept:  -0.10700642

Results from SsePackedFloatingPointLeastSquares_
  xc:              1
  slope:       0.10424641
  intercept:  -0.10700642
```

259

9.2.2 用组合浮点数进行 4×4 矩阵的计算

计算机图形学或计算机辅助设计程序之类的应用程序常常要进行大量的矩阵计算。比如，3D 图形软件经常使用矩阵进行各种变换，比如转化、缩放或旋转。在使用齐次坐标时，这些操作可以用一个 4×4 矩阵高效地表示。多次转换同样可以通过矩阵乘法把一系列独立的变换矩阵合并成一个变换矩阵。这种合并后的矩阵常用于表示用来定义 3D 模型的对象顶点的数组。对 3D 计算机图形软件而言，矩阵乘法和矩阵向量乘法的计算性能至关重要，因为一个 3D 模型通常包含成千上万个对象顶点。

两个矩阵的乘积定义如下：令矩阵 A 是一个 $m \times n$ 矩阵（m 和 n 分别代表行数和列数），矩阵 B 是一个 $n \times p$ 矩阵，C 是 A 和 B 的乘积，是一个 $m \times p$ 矩阵。C 的每个成员 $c(i, j)$ 可以通过下面这个公式计算：

$$c_{ij} = \sum_{r=0}^{n-1} a_{ir} b_{rj} \quad i=0, \cdots, m-1; j=0, \cdots, p-1$$

在进入示例代码之前，还需要补充说明一下：根据矩阵乘法的定义，矩阵 A 的列数必须等于矩阵 B 的行数。举例来说，如果 A 是一个 3×4 的矩阵，B 是一个 4×2 的矩阵，那么乘积 AB（3×2 矩阵）是可以计算的，而 BA 是非法的，无法计算。同样需要注意的是，C 的成员 $c(i, j)$ 是矩阵 A 的 i 行与矩阵 B 的 j 列的简单的点乘。下面的示例程序会利用这一点使用 SIMD 算术指令进行矩阵与矩阵和矩阵与向量的乘法计算。最后，与大多数数学教材不同，上面的矩阵乘法方程式的下标标识是从 0 开始的，这样翻译成 C++ 和汇编语言代码就变得简单了。

SsePackedFloatingPointMatrix4×4 示例程序演示了如何使用 x86-SSE 指令集进行 4×4 矩阵和 4×1 向量的矩阵 - 矩阵和矩阵 - 向量乘法的方法。清单 9-10 和清单 9-11 分别包含了 SsePackedFloatingPointMatrix4×4 的 C++ 和汇编语言源代码。

清单 9-10 SsePackedFloatingPointMatrix4x4.cpp

```
#include "stdafx.h"
#include "SsePackedFloatingPointMatrix4x4.h"

// Mat4x4Mul 和 Mat4x4MulVec 两个函数定义在 CommonFiles\Mat4x4.cpp 文件中

void SsePfpMatrix4x4MultiplyCpp(Mat4x4 m_des, Mat4x4 m_src1, Mat4x4 m_src2)
{
    Mat4x4Mul(m_des, m_src1, m_src2);
260 }

void SsePfpMatrix4x4TransformVectorsCpp(Vec4x1* v_des, Mat4x4 m_src,↩
Vec4x1* v_src, int num_vec)
{
    for (int i= 0; i < num_vec; i++)
        Mat4x4MulVec(v_des[i], m_src, v_src[i]);
}

void SsePfpMatrix4x4Multiply(void)
{
    __declspec(align(16)) Mat4x4 m_src1;
    __declspec(align(16)) Mat4x4 m_src2;
    __declspec(align(16)) Mat4x4 m_des1;
    __declspec(align(16)) Mat4x4 m_des2;

    Mat4x4SetRow(m_src1, 0, 10.5, 11, 12, -13.625);
    Mat4x4SetRow(m_src1, 1, 14, 15, 16, 17.375);
    Mat4x4SetRow(m_src1, 2, 18.25, 19, 20.125, 21);
    Mat4x4SetRow(m_src1, 3, 22, 23.875, 24, 25);

    Mat4x4SetRow(m_src2, 0, 7, 1, 4, 8);
    Mat4x4SetRow(m_src2, 1, 14, -5, 2, 9);
    Mat4x4SetRow(m_src2, 2, 10, 9, 3, 6);
    Mat4x4SetRow(m_src2, 3, 2, 11, -14, 13);

    SsePfpMatrix4x4MultiplyCpp(m_des1, m_src1, m_src2);
    SsePfpMatrix4x4Multiply (m_des2, m_src1, m_src2);

    printf("\nResults for SsePfpMatrix4x4Multiply()\n");
    Mat4x4Printf(m_src1, "\nMatrix m_src1\n");
    Mat4x4Printf(m_src2, "\nMatrix m_src2\n");
    Mat4x4Printf(m_des1, "\nMatrix m_des1\n");
    Mat4x4Printf(m_des2, "\nMatrix m_des2\n");
}

void SsePfpMatrix4x4TransformVectors(void)
{
    const int n = 8;
    __declspec(align(16)) Mat4x4 m_src;
    __declspec(align(16)) Vec4x1 v_src[n];
    __declspec(align(16)) Vec4x1 v_des1[n];
    __declspec(align(16)) Vec4x1 v_des2[n];

    Vec4x1Set(v_src[0], 10, 10, 10, 1);
    Vec4x1Set(v_src[1], 13, 11, 10, 1);
    Vec4x1Set(v_src[2], 11, 10, 10, 1);
    Vec4x1Set(v_src[3], 11, 11, 10, 1);
    Vec4x1Set(v_src[4], 10, 10, 12, 1);
261 Vec4x1Set(v_src[5], 10, 11, 12, 1);
    Vec4x1Set(v_src[6], 11, 10, 12, 1);
    Vec4x1Set(v_src[7], 11, 11, 12, 1);
```

```
    // m_src = scale(2, 3, 4)
    Mat4x4SetRow(m_src, 0, 2, 0, 0, 0);
    Mat4x4SetRow(m_src, 1, 0, 3, 0, 0);
    Mat4x4SetRow(m_src, 2, 0, 0, 7, 0);
    Mat4x4SetRow(m_src, 3, 0, 0, 0, 1);

    SsePfpMatrix4x4TransformVectorsCpp(v_des1, m_src, v_src, n);
    SsePfpMatrix4x4TransformVectors_(v_des2, m_src, v_src, n);

    printf("\nResults for SsePfpMatrix4x4TransformVectors()\n");
    Mat4x4Printf(m_src, "Matrix m_src\n");
    printf("\n");

    for (int i = 0; i < n; i++)
    {
        const char* fmt = "%4s %4d: %12.6f %12.6f %12.6f %12.6f\n";
        printf(fmt, "v_src  ", i, v_src[i][0], v_src[i][1], v_src[i][2],↵
v_src[i][3]);
        printf(fmt, "v_des1 ", i, v_des1[i][0], v_des1[i][1], v_des1[i][2],↵
v_des1[i][3]);
        printf(fmt, "v_des2 ", i, v_des2[i][0], v_des2[i][1], v_des2[i][2],↵
v_des2[i][3]);
        printf("\n");
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    SsePfpMatrix4x4Multiply();
    SsePfpMatrix4x4TransformVectors();

    SsePfpMatrix4x4MultiplyTimed();
    SsePfpMatrix4x4TransformVectorsTimed();
    return 0;
}
```

清单 9-11 SsePackedFloatingPointMatrix4x4_.asm

```
        .model flat,c
        .code

; _Mat4x4Transpose 宏
;
; 函数说明：这个宏计算 4 x 4 单精度浮点矩阵的转置
;

;   输入矩阵                        输出矩阵
;   xmm0    a3 a2 a1 a0             xmm4    d0 c0 b0 a0
;   xmm1    b3 b2 b1 b0             xmm5    d1 c1 b1 a1
;   xmm2    c3 c2 c1 c0             xmm6    d2 c2 b2 a2
;   xmm3    d3 d2 d1 d0             xmm7    d3 c3 b3 a3
;
; 注意：4x4 矩阵在被载入到 XMM 寄存器时，由于 x86 低字节序原因，每行会被反转
;
; SSE 版本：SSE

_Mat4x4Transpose macro
        movaps xmm4,xmm0
        unpcklps xmm4,xmm1                  ;xmm4 = b1 a1 b0 a0
        unpckhps xmm0,xmm1                  ;xmm0 = b3 a3 b2 a2
        movaps xmm5,xmm2
        unpcklps xmm5,xmm3                  ;xmm5 = d1 c1 d0 c0
        unpckhps xmm2,xmm3                  ;xmm2 = d3 c3 d2 c2
```

262

```
        movaps xmm1,xmm4
        movlhps xmm4,xmm5                   ;xmm4 = d0 c0 b0 a0
        movhlps xmm5,xmm1                   ;xmm5 = d1 c1 b1 a1
        movaps xmm6,xmm0
        movlhps xmm6,xmm2                   ;xmm6 = d2 c2 b2 a2
        movaps xmm7,xmm2
        movhlps xmm7,xmm0                   ;xmm7 = d3 c3 b2 a3
        ercn

; extern "C" void SsePfpMatrix4x4Multiply_(Mat4x4 m_des, Mat4x4 m_src1,
Mat4x4 m_src2);
;
; 函数说明：下面这个函数用于计算两个 4 x 4 单精度浮点矩阵的乘积
;
; SSE 版本：SSE4.1

SsePfpMatrix4x4Multiply_ proc
        push ebp
        mov ebp,esp
        push ebx

; 计算 m_src2 (m_src2_T) 的转置
        mov ebx,[ebp+16]                    ;ebx = m_src2
        movaps xmm0,[ebx]
        movaps xmm1,[ebx+16]
        movaps xmm2,[ebx+32]
        movaps xmm3,[ebx+48]                ;xmm3:xmm0 = m_src2
263     _Mat4x4Transpose                    ;xmm7:xmm4 = m_src2_T

; 矩阵乘积的初始化
        mov edx,[ebp+8]                     ;edx = m_des
        mov ebx,[ebp+12]                    ;ebx = m_src1
        mov ecx,4                           ;ecx = 行数
        xor eax,eax                         ;eax = 数组偏移

; 执行循环直到矩阵乘积被计算出来
        align 16
@@:     movaps xmm0,[ebx+eax]               ;xmm0 = m_src1 的第 i 行

; 计算 m_src1 第 i 行与 m_src2_T 第 0 行的点积
        movaps xmm1,xmm0
        dpps xmm1,xmm4,11110001b            ;xmm1[31:0] = 点积
        insertps xmm3,xmm1,00000000b        ;xmm3[31:0] = xmm1[31:0]

; 计算 m_src1 第 i 行与 m_src2_T 第 1 行的点积
        movaps xmm2,xmm0
        dpps xmm2,xmm5,11110001b            ;xmm2[31:0] = 点积
        insertps xmm3,xmm2,00010000b        ;xmm3[63:32] = xmm2[31:0]

; 计算 m_src1 第 i 行与 m_src2_T 第 2 行的点积
        movaps xmm1,xmm0
        dpps xmm1,xmm6,11110001b            ;xmm1[31:0] = 点积
        insertps xmm3,xmm1,00100000b        ;xmm3[95:64] = xmm1[31:0]

; 计算 m_src1 第 i 行与 m_src2_T 第 3 行的点积
        movaps xmm2,xmm0
        dpps xmm2,xmm7,11110001b            ;xmm2[31:0] = 点积
        insertps xmm3,xmm2,00110000b        ;xmm3[127:96] = xmm2[31:0]

; 保存 m_des.row i 并更新循环变量
        movaps [edx+eax],xmm3               ;保存当前行的结果
        add eax,16                          ;调整数组偏移，使之指向下一行
```

```
        dec ecx
        jnz @B

        pop ebx
        pop ebp
        ret
SsePfpMatrix4x4Multiply_ endp

; extern void SsePfpMatrix4x4TransformVectors_(Vec4x1* v_des, Mat4x4 m_src,*
Vec4x1* v_src, int num_vec);
;
; 函数说明：下面这个函数把一个变换矩阵应用到一个 4 x 1 单精度浮点向量数组
;
; SSE 版本：SSE4.1                                                    [264]

SsePfpMatrix4x4TransformVectors_ proc
        push ebp
        mov ebp,esp
        push esi
        push edi

; 确保 num_vec 是有效的
        mov ecx,[ebp+20]                ;ecx = num_vec
        test ecx,ecx
        jle Done                        ;若 num_vec <= 0, 则跳转

; 把 m_src 载入到 xmm3:xmm0
        mov eax,[ebp+12]                ;eax = 指向 m_src 的指针
        movaps xmm0,[eax]               ;xmm0 = row 0
        movaps xmm1,[eax+16]            ;xmm1 = row 1
        movaps xmm2,[eax+32]            ;xmm2 = row 2
        movaps xmm3,[eax+48]            ;xmm3 = row 3

; 初始化指向 v_src 和 v_des 的指针
        mov esi,[ebp+16]                ;esi = 指向 v_src 的指针
        mov edi,[ebp+8]                 ;edi = 指向 v_des 的指针
        xor eax,eax                     ;eax = 数组偏移

; 计算：v_des[i] = m_src * v_src[i]
        align 16
@@:     movaps xmm4,[esi+eax]           ;xmm4 = v_src[i] 向量

; 计算 m_src 第 0 行和 v_src[i] 的点积
        movaps xmm5,xmm4
        dpps xmm5,xmm0,11110001b        ;xmm5[31:0] = dot product
        insertps xmm7,xmm5,00000000b    ;xmm7[31:0] = xmm5[31:0]

; 计算 m_src 第 1 行和 v_src[i] 的点积
        movaps xmm6,xmm4
        dpps xmm6,xmm1,11110001b        ;xmm6[31:0] = dot product
        insertps xmm7,xmm6,00010000b    ;xmm7[63:32] = xmm6[31:0]

; 计算 m_src 第 2 行和 v_src[i] 的点积
        movaps xmm5,xmm4
        dpps xmm5,xmm2,11110001b        ;xmm5[31:0] = dot product
        insertps xmm7,xmm5,00100000b    ;xmm7[95:64] = xmm5[31:0]

; 计算 m_src 第 3 行和 v_src[i] 的点积
        movaps xmm6,xmm4
        dpps xmm6,xmm3,11110001b        ;xmm6[31:0] = dot product
        insertps xmm7,xmm6,00110000b    ;xmm7[127:96] = xmm6[31:0]      [265]
```

```
; 保存 v_des[i]，更新循环变量
        movaps [edi+eax],xmm7                ;保存变换后的向量
        add eax,16
        dec ecx
        jnz @B

Done:   pop edi
        pop esi
        pop ebp
        ret
SsePfpMatrix4x4TransformVectors_ endp
        end
```

C++源文件 SsePackedFloatingPointMatrix4×4.cpp（见清单 9-10）包含一组演示矩阵－矩阵乘法和矩阵－向量转换的测试函数。函数 SsePfpMatrix4×4Multiply 初始化一组测试用的矩阵，然后调用 C++和汇编版本的矩阵乘法函数，最后输出这些函数的结果以便于对比。请注意，其中一些 C++矩阵函数定义在 Mat4×4.cpp 文件中。这个文件没有在这里列出来，读者可以在网站上下载。与之类似，函数 SsePfpMatrix4×4TransformVectors 演示了矩阵－向量的转换。

示例程序 SsePackedFloatingPointMatrix4×4 最重要的部分包含在汇编源文件 SsePackedFloatingPointMatrix4×4_asm（见清单 9-11）里。汇编语言程序的开始部分是一个宏 _Mat4×4Transpose。宏是汇编器的一种文本替代机制，允许程序员把一段具有独立功能的汇编语言指令、数据定义或其他语句表示成一个字符串。在以后代码里，程序员可以通过使用宏指令名多次引用宏，而不再需要每次都重新定义。汇编源程序被编译时，汇编编译器将对每个宏调用进行宏展开，即用宏定义的宏体去代替宏指令名，并且用实际参数——取代形式参数。汇编语言的宏通常用于产生被多次使用的指令序列。与函数调用不同，宏在编译期间展开，没有函数调用产生的性能问题。在本书的余下章节，读者将学习到更多关于 MASM 宏的用法。

宏 _Mat4×4Transpose 定义了一系列汇编语言指令来计算 4×4 单精度浮点矩阵的转置。矩阵转置的定义如下：如果 A 是一个 $m \times n$ 的矩阵，则 A 的转置（标记为 B）是一个 $n \times m$ 的矩阵，其中 $b(i, j) = a(j, i)$。宏 _Mat4×4Transpose 要求源矩阵被载入到寄存器 XMM0 ~ XMM3，并把转置矩阵保存在寄存器 XMM4 ~ XMM7。实际的转置过程是由
266 movaps、unpcklps、unpckhps、movlhps 和 movhlps 等指令组合起来完成的，如图 9-2 所示。

函数 SsePfpMatrix4x4Multiply_ 调用宏 _Mat4x4Transpose 来计算两个 4×4 矩阵的乘积。在本节的开头，我们知道了矩阵乘积 $C(= AB)$ 的每个成员 $c(i, j)$ 只是矩阵 A 的第 i 行与矩阵 B 的第 j 列的点积。这意味着可以使用 x86-SSE 的 dpps 指令（Dot Product of Packed Single-Precision Floating-Point Value，组合单精度浮点数的点积）来加速两个 4×4 矩阵的乘法。C++在内存中使用行主序来组织二维矩阵，于是 4×4 单精度浮点数矩阵的一行可以被通过 movaps 指令载入到一个 XMM 寄存器。然而，没有一个 x86-SSE 指令能把矩阵的一列载入到 XMM 寄存器里。对此问题，一个可行的解决方案是把其中一个矩阵进行转置，这样就可以利用 dpps 指令来加速矩阵乘法。

接下来我们仔细解析一下 SsePfpMatrix×4Multiply 函数。在函数序言之后，通过一系列 movaps 指令将矩阵 m_src2 载入寄存器 XMM0 ~ XMM3，紧接着，通过调用宏 _Mat4×4Transpose 来计算矩阵 m_src2 的转置，得到 m_src2_T（图 9-3 包括对宏 _Mat4×4-Transpose 展开后的部分汇编语言源代码）。接下来，寄存器 EBX 和 EDX 分别被初始化为

指 m_src1 和 m_des 的指针，ECX 保存循环计数，EAX 保存 m_src1 和 m_des 的行偏移。在主循环的开头，指令 movaps xmm0,[ebx+eax] 把 m_src1 的第 0 行载入到 XMM0，之后通过指令 movaps xmm1,xmm0 把第 0 行复制到 XMM1，然后指令 dpps xmm1, xmm4,11110001b 计算 m_src1 第 0 行与 m_src2_T 第 0 行的点积。dpps 的立即数操作数的高 4 位是一个条件掩码，指定 XMM1 和 XMM4 的哪些部分进行相乘。在本示例中，所有部分都需要进行乘法计算（比如 xmm1[31:0] * xmm4[31:0], xmm1[63:32] *xmm4[63:32], 以此类推）。如果某个条件掩码位为 0，乘积结果会被设置为 0。dpps 的立即数操作数的低四位是一个广播掩码，用来指定是把点积结果（如果该位置为 1）还是 0.0 复制到目标操作数的相应元素。

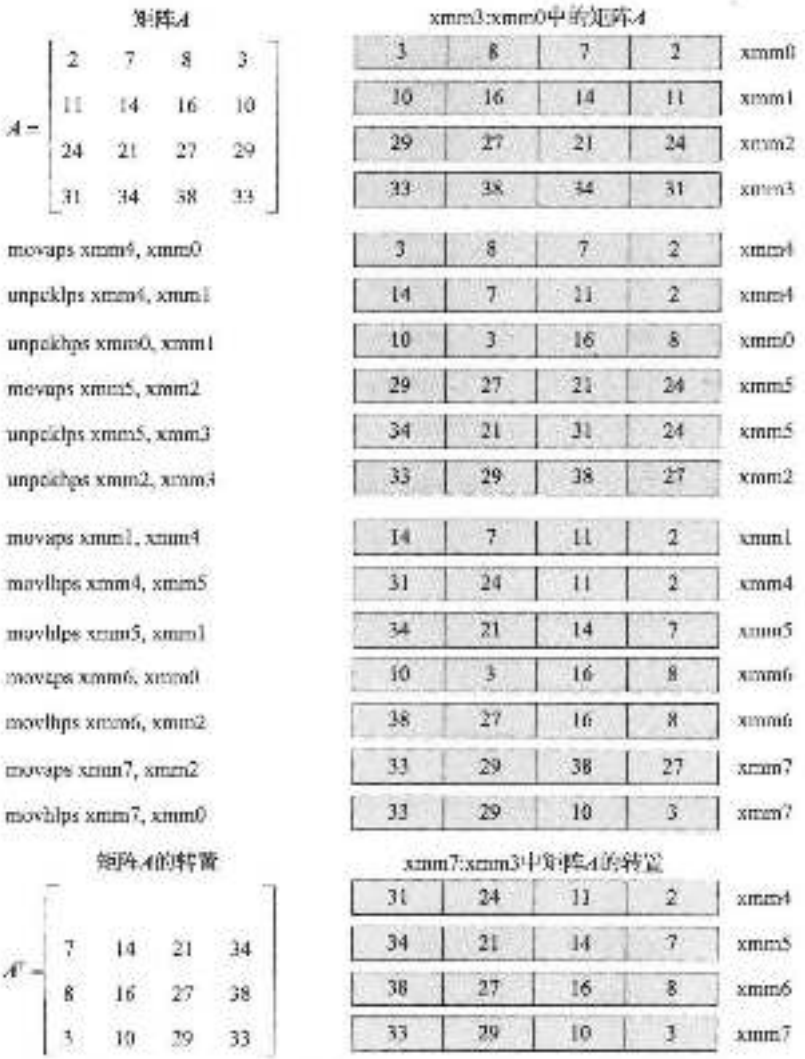


图 9-2 宏 _Mat4×4Transpose 的指令序列，用于转置一个 4×4 单精度浮点数矩阵

接下来，指令 insertps xmm3,xmm1,00000000b 把计算所得的点积复制到 m_des（XMM3 中）的对应元素位置。insertps 的立即操作数的第 7:6 位指定要复制的源操作数的元素；第 5:4 位指定目标操作数的元素；位 3:0 为零掩码，用于在某些条件下把某个目标操作数设为

0。再之后的 movaps，dpps 和 interpts 指令序列用于计算并保存 m_src1 的第 0 行与 m_src2_T 的第 1 行的点积，接下去再计算其他 m_src1 与 m_src2_T 的点积，直到所有 16 个点积都被计算并保存下来。

```
00000000                SsePfpMatrix4x4Multiply_ proc
00000000  55                    push ebp
00000001  8B EC                 mov ebp,esp
00000003  53                    push ebx

                              ;计算m_src2的转置（m_src2_T）
00000004  8B 5D 10              mov ebx,[ebp+16]        ;ebx = m_src2
00000007  0F 28 03              movaps xmm0,[ebx]
0000000A  0F 28 4B 10           movaps xmm1,[ebx+16]
0000000E  0F 28 53 20           movaps xmm2,[ebx+32]
00000012  0F 28 5B 30           movaps xmm3,[ebx+48]    ;xmm3:xmm0 = m_src2
                                _Mat4x4Transpose        ;xmm7:xmm4 = m_src2_T
00000016  0F 28 E0   1             movaps xmm4,xmm0
00000019  0F 14 E1   1             unpcklps xmm4,xmm1   ;xmm4 = b1 a1 b0 a0
0000001C  0F 15 C1   1             unpckhps xmm0,xmm1   ;xmm0 = b3 a3 b2 a2
0000001F  0F 28 EA   1             movaps xmm5,xmm2
00000022  0F 14 EB   1             unpcklps xmm5,xmm3   ;xmm5 = d1 c1 d0 c0
00000025  0F 15 D3   1             unpckhps xmm2,xmm3   ;xmm2 = d3 c3 d2 c2
00000028  0F 28 CC   1             movaps xmm1,xmm4
0000002B  0F 16 E5   1             movlhps xmm4,xmm5    ;xmm4 = d0 c0 b0 a0
0000002E  0F 12 E9   1             movhlps xmm5,xmm1    ;xmm5 = d1 c1 b1 a1
00000031  0F 28 F0   1             movaps xmm6,xmm0
00000034  0F 16 F2   1             movlhps xmm6,xmm2    ;xmm6 = d2 c2 b2 a2
00000037  0F 28 FA   1             movaps xmm7,xmm2
0000003A  0F 12 F8   1             movhlps xmm7,xmm0    ;xmm7 = d3 c3 b2 a3

                              ;初始化
0000003D  8B 55 08              mov edx,[ebp+8]         ;edx = m_des
00000040  8B 5D 0C              mov ebx,[ebp+12]        ;ebx = m_src1
```

图 9-3 宏 _Mat4x4Transpose 的展开

汇编语言文件 SsePackedFloatingPointMatrix4x4_.asm 还包含一个名为 SsePfpMatrix4x4TransformVectors 的函数，此函数把一个 4×4 变换矩阵应用到一个 4×1 向量数组的每个向量成员。它同样使用 dpps 指令对一个 4×4 矩阵与 4×1 向量的乘法进行加速。

输出 9-5 显示了 SsePackedFloatingPointMatrix4x4 示例程序的执行结果。

输出 9-5 示例程序 SsePackedFloatingPointMatrix4x4

```
Results for SsePfpMatrix4x4Multiply()

Matrix m_src1
     10.500000      11.000000      12.000000     -13.625000
     14.000000      15.000000      16.000000      17.375000
     18.250000      19.000000      20.125000      21.000000
     22.000000      24.875000      24.000000      25.000000

Matrix m_src2
      7.000000       1.000000       4.000000       8.000000
     14.000000      -5.000000       2.000000       9.000000
     10.000000       9.000000       3.000000       6.000000
      2.000000      11.000000     -14.000000      13.000000

Matrix m_des1
    320.250000     -86.375000     290.750000      77.875000
```

```
    502.750000    274.125000    -109.250000    568.875000
    637.000000    335.375000    -122.625000    710.750000
    778.250000    394.625000    -142.250000    859.875000

Matrix m_des2
    320.250000    -86.375000    290.750000    77.875000
    502.750000    274.125000    -109.250000    568.875000
    637.000000    335.375000    -122.625000    710.750000
    778.250000    394.625000    -142.250000    859.875000

Results for SsePfpMatrix4x4TransformVectors()
Matrix m_src
      2.000000      0.000000      0.000000      0.000000
      0.000000      3.000000      0.000000      0.000000
      0.000000      0.000000      7.000000      0.000000
      0.000000      0.000000      0.000000      1.000000

v_src     0:    10.000000    10.000000    10.000000    1.000000
v_des1    0:    20.000000    30.000000    70.000000    1.000000
v_des2    0:    20.000000    30.000000    70.000000    1.000000

v_src     1:    10.000000    11.000000    10.000000    1.000000
v_des1    1:    20.000000    33.000000    70.000000    1.000000
v_des2    1:    20.000000    33.000000    70.000000    1.000000

v_src     2:    11.000000    10.000000    10.000000    1.000000
v_des1    2:    22.000000    30.000000    70.000000    1.000000
v_des2    2:    22.000000    30.000000    70.000000    1.000000

v_src     3:    11.000000    11.000000    10.000000    1.000000
v_des1    3:    22.000000    33.000000    70.000000    1.000000
v_des2    3:    22.000000    33.000000    70.000000    1.000000

v_src     4:    10.000000    10.000000    12.000000    1.000000
v_des1    4:    20.000000    30.000000    84.000000    1.000000
v_des2    4:    20.000000    30.000000    84.000000    1.000000

v_src     5:    10.000000    11.000000    12.000000    1.000000
v_des1    5:    20.000000    33.000000    84.000000    1.000000
v_des2    5:    20.000000    33.000000    84.000000    1.000000

v_src     6:    11.000000    10.000000    12.000000    1.000000
v_des1    6:    22.000000    30.000000    84.000000    1.000000
v_des2    6:    22.000000    30.000000    84.000000    1.000000

v_src     7:    11.000000    11.000000    12.000000    1.000000
v_des1    7:    22.000000    33.000000    84.000000    1.000000
v_des2    7:    22.000000    33.000000    84.000000    1.000000

Results for SsePfpMatrix4x4MultiplyTimed()
Benchmark times saved to __SsePfpMatrix4x4MultiplyTimed.csv

Results for SsePfpMatrix4x4TransformVectorsTimed()
Benchmark times saved to __SsePfpMatrix4x4TransformVectorsTimed.csv
```

注意 如果在 Visual Studio IDE 里运行此示例程序，程序产生的结果文件都被保存在 Chapter##\<ProgramName>\<ProgramName> 目录下，其中 ## 代表章节号，<ProgramName> 代表示例程序的名字。

为了进行性能对比，表 9-2 和表 9-3 给出了测量执行时间的结果。测量执行时间的代码

没有在本书中列出，读者可以从网站上下载。

表 9-2 SsePfpMatrix4x4Multiply 函数的平均执行时间（单位：毫秒，2000 次矩阵乘法）

CPU	C++	X86-SSE
Intel Core i7-4770	50	31
Intel Core i7-4600U	64	41
Intel Core i3-2310M	97	68

表 9-3 SsePfpMatrix4x4TransformVectors 函数的平均执行时间（单位：毫秒，10 000 次向量变换）

CPU	C++	X86-SSE
Intel Core i7-4770	43	26
Intel Core i7-4600U	55	31
Intel Core i3-2310M	91	63

使用 x86-SSE 指令集进行矩阵乘法计算会得到 30% – 38% 的性能提升（目标处理器不同，测试结果会有差异）。对于矩阵－向量间的乘法计算，x86-SSE 指令集可以带来显著的性能提升，一般为 31% – 40%。

9.3 总结

本章集中介绍了 x86-SSE 的组合浮点计算能力。我们学习了使用 128 位组合浮点操作数进行基本算术运算的方法，还通过几个示例程序演示了对浮点数组和 4×4 矩阵进行 SIMD 处理的技巧。下一章，我们将学习创建汇编语言函数来利用 x86-SSE 的组合寄数资源。

269
-
272

x86-SSE 编程——组合整数

在第 6 章，我们学习了如何使用 MMX 的计算资源来操作组合整型数。这一章，我们将学习如何利用 x86-SSE 的计算资源来处理组合整型数。第一个示例程序侧重于使用 x86-SSE 指令集和 XMM 寄存器进行基本的组合整数操作。大家可能很快就能体会到，使用 128 位 XMM 寄存器进行组合整数操作与使用 64 位 MMX 寄存器并没有太大的差别。后面的两个示例程序演示了如何使用 x86-SSE 指令实现常见的图像处理任务，包括直方图构建和灰度图的阈值分割（thresholding）。

和前面两章一样，本章的示例代码将使用不同版本的 x86-SSE。每一个汇编语言函数清单的开头会列出所需的 x86-SSE 扩展版本。使用本书附录 C 中所列的软件工具可以检测读者的 PC 机对 x86-SSE 的支持程度。

10.1 组合整数基础

本章我们要学习的第一个 x86-SSE 组合整数示例程序名叫 SsePackedIntegerFundamentals。这个示例程序的目的是为了演示如何使用 XMM 寄存器进行通用的组合整数操作。示例程序 SsePackedIntegerFundamentals 的 C++ 和汇编语言源代码分别列在清单 10-1 和清单 10-2 中。

清单 10-1 SsePackedIntegerFundamentals.cpp

```
#include "stdafx.h"
#include "XmmVal.h"

extern "C" void SsePiAddI16_(const XmmVal* a, const XmmVal* b, XmmVal c[2]);
extern "C" void SsePiSubI32_(const XmmVal* a, const XmmVal* b, XmmVal* c);
extern "C" void SsePiMulI32_(const XmmVal* a, const XmmVal* b, XmmVal c[2]);

void SsePiAddI16(void)
{
    _declspec(align(16)) XmmVal a;
    _declspec(align(16)) XmmVal b;
    _declspec(align(16)) XmmVal c[2];
    char buff[256];

    a.i16[0] = 10;          b.i16[0] = 100;
    a.i16[1] = 200;         b.i16[1] = -200;
    a.i16[2] = 30;          b.i16[2] = 32760;
    a.i16[3] = -32766;      b.i16[3] = -400;
    a.i16[4] = 50;          b.i16[4] = 500;
    a.i16[5] = 60;          b.i16[5] = -600;
    a.i16[6] = 32000;       b.i16[6] = 1200;
    a.i16[7] = -32000;      b.i16[7] = -950;

    SsePiAddI16_(&a, &b, c);

    printf("\nResults for SsePiAddI16_\n");
```

```
    printf("a:    %s\r", a.ToString_i16(buff, sizeof(buff)));
    printf("b:    %s\r", b.ToString_i16(buff, sizeof(buff)));
    printf("c[0]: %s\r", c[0].ToString_i16(buff, sizeof(buff)));
    printf("\n");
    printf("a:    %s\n", a.ToString_i16(buff, sizeof(buff)));
    printf("b:    %s\n", b.ToString_i16(buff, sizeof(buff)));
    printf("c[1]: %s\n", c[1].ToString_i16(buff, sizeof(buff)));
}

void SsePiSubI32(void)
{
    __declspec(align(16)) XmmVal a;
    __declspec(align(16)) XmmVal b;
    __declspec(align(8)) XmmVal c;        // Misaligned XmmVal
    char buff[256];

    a.i32[0] = 800;          b.i32[0] = 200;
    a.i32[1] = 600;          b.i32[1] = -2000;
    a.i32[2] = 2000;         b.i32[2] = -40;
    a.i32[3] = 900;          b.i32[3] = 1200;

    SsePiSubI32 (&a, &b, &c);

    printf("\nResults for SsePiSubI32 \n");
    printf("a: %s\n", a.ToString_i32(buff, sizeof(buff)));
    printf("b: %s\n", b.ToString_i32(buff, sizeof(buff)));
    printf("c: %s\n", c.ToString_i32(buff, sizeof(buff)));
}

void SsePiMulI32(void)
{
    __declspec(align(16)) XmmVal a;
    __declspec(align(16)) XmmVal b;
    __declspec(align(16)) XmmVal c[2];
    char buff[256];

    a.i32[0] = 10;           b.i32[0] = 100;
    a.i32[1] = 20;           b.i32[1] = -200;
    a.i32[2] = -30;          b.i32[2] = 300;
    a.i32[3] = -40;          b.i32[3] = -400;

    SsePiMulI32_(&a, &b, c);

    printf("\nResults for SsePiMulI32 \n");
    printf("a:    %s\n", a.ToString_i32(buff, sizeof(buff)));
    printf("b:    %s\n", b.ToString_i32(buff, sizeof(buff)));
    printf("c[0]: %s\n", c[0].ToString_i32(buff, sizeof(buff)));
    printf("\n");
    printf("a:    %s\n", a.ToString_i32(buff, sizeof(buff)));
    printf("b:    %s\n", b.ToString_i32(buff, sizeof(buff)));
    printf("c[1]: %s\n", c[1].ToString_i64(buff, sizeof(buff)));
}

int _tmain(int argc, _TCHAR* argv[])
{
    SsePiAddI16();
    SsePiSubI32();
    SsePiMulI32();
    return 0;
}
```

274

清单 10-2　SsePackedIntegerFundamentals .asm

```
        .model flat,c
        .code

; extern "C" void SsePiAddI16_(const XmmVal* a, const XmmVal* b, XmmVal c[2]);
;
; 描述：下面的函数演示了使用回绕和饱和模式进行有符号双字节组合整数加法运算
;
; 需要 SSE2 支持

SsePiAddI16_ proc
        push ebp
        mov ebp,esp                                                    275

; 初始化
        mov eax,[ebp+8]                 ;eax = 指向 a
        mov ecx,[ebp+12]                ;ecx = 指向 b
        mov edx,[ebp+16]                ;edx = 指向 c

; 将 a 和 b 的值加载到 XmmVals 中
        movdqa xmm0,[eax]               ;xmm0 = a
        movdqa xmm1,xmm0
        movdqa xmm2,[ecx]               ;xmm2 = b

; 执行双字节组合整数加法
        paddw xmm0,xmm2                 ;组合加法，回绕模式
        paddsw xmm1,xmm2                ;组合加法，饱和模式

; 保存结果
        movdqa [edx],xmm0               ;保存 c[0]
        movdqa [edx+16],xmm1            ;保存 c[1]

        pop ebp
        ret
SsePiAddI16_ endp

; extern "C" void SsePiSubI32_(const XmmVal* a, const XmmVal* b, XmmVal* c);
;
; 描述：下面的函数演示了有符号双字组合整数减法
;
; 需要 SSE2 支持

SsePiSubI32_ proc
        push ebp
        mov ebp,esp

; 初始化
        mov eax,[ebp+8]                 ;eax = 指向 a
        mov ecx,[ebp+12]                ;ecx = 指向 b
        mov edx,[ebp+16]                ;edx = 指向 c

; 执行双字组合整数减法
        movdqa xmm0,[eax]               ;xmm0 = a
        psubd xmm0,[ecx]                ;xmm0 = a - b
        movdqu [edx],xmm0               ;将结果保存到未对齐的内存中

        pop ebp
        ret
SsePiSubI32_ endp                                                      276

; extern "C" void SsePiMul32_(const XmmVal* a, const XmmVal* b, XmmVal c[2]);
;
; 描述：下面的函数演示了双字组合整数乘法
```

```
;
; 需要 SSE4.1 支持

SsePiMul32_ proc
        push ebp
        mov ebp,esp

; 初始化
        mov eax,[ebp+8]                 ;eax = 指向 a
        mov ecx,[ebp+12]                ;ecx = 指向 b
        mov edx,[ebp+16]                ;edx = 指向 c

; 加载相应的值并执行算法
        movdqa xmm0,[eax]               ;xmm0 = a
        movdqa xmm1,[ecx]               ;xmm1 = b

        movdqa xmm2,xmm0
        pmulld xmm0,xmm1                ;有符号双字算法，低 32 位结果
        pmuldq xmm2,xmm1                ;有符号双字算法，8 字节结果

        movdqa [edx],xmm0               ;c[0] = pmulld 的运算结果
        movdqa [edx+16],xmm2            ;c[1] = pmuldq 的运算结果

        pop ebp
        ret
SsePiMul32_ endp
        end
```

C++ 文件 SsePackedIntegerFundamentals.cpp（程序清单 10-1）包含了 3 个测试函数，它们分别演示了基于 x86-SSE 指令集的组合整型加法、减法和乘法运算。第一个函数是 SsePiAdd16，它使用 16 位有符号整型数初始化若干个 XmmVal 实例，然后调用一个汇编语言函数进行组合整数加法，加法的运算分别使用了回绕和饱和算术运算模式。第二个测试函数是 SsePiSub32，它会初始化一些 XmmVal 变量来演示如何对 32 位有符号整数进行组合整数减法运算。注意在这个函数中，XmmVal 变量 c 被故意设成未对齐的形式，以便演示未对齐数据传输指令的用法。最后一个函数是 SsePiMul32，它初始化若干个 XmmVal 变量来对 32 位组合整数进行乘法运算。

汇编语言文件 SsePackedIntegerFundamentals_asm（见清单 10-2）包含了相应的汇编语言函数。在序言之后，函数 SsePiAdd16_ 会将参数 a、b 和 c 的值分别加载到寄存器 EAX、ECX 和
277 EDX 中。指令 movdqa xmm0,[eax]（移动对齐的双四字（16 字节））将 a 的值加载到 XMM0 中。正如其名，movdqa 指令要求其访问的所有内存操作数都是以 16 字节边界对齐的。指令 movdqa xmm1,xmm0 用来将 XMM0 的内容复制到 XMM1 中。接下来指令 movdqa xmm2,[ecx] 将 b 加载到 XMM2 中。指令 paddw 和 paddsw 分别使用回绕和饱和算术运算方法执行 16 位有符号组合整数的加法。然后函数使用一组 movdqa 指令将运算结果保存到函数调用者指定的数组中。

函数 SsePiSub32_ 使用 psubd 指令和有符号双字整数来实现组合减法。注意该函数使用了一个 movdqu 指令（复制未对齐的 16 字节整数）将运算结果保存到内存中。这条指令的目标操作数与前面的 XmmVal 相对应。XmmVal 在 C++ 函数 SsePiSub32 中被故意设成未对齐形式，对它执行 movdqa 指令会导致处理器产生一个异常。最后一个汇编语言函数 SsePiMul32_ 演示了有符号双字组合整数的乘法运算。值得一提的是 x86-SSE 指令集支持两种不同形式的有符号双字组合整数乘法。其中 pmulld（有符号双字组合整数相乘并保存结果的低 32 位）指令对 32 位数执行有符号整数乘法，然后保存每个乘积的低 32 位数值。而 pmuldq（有符号双字组合整数相乘）指令计算有符号整数乘法 des[63:0] = des[31:0] * src[31:0] 和 des[127:64] = des[95:64] * src[95:64]，然后保存整个 64 位（四字）乘积。输出

10.1 列出了示例程序 SsePackedIntegerFundamentals 的运算结果。

输出 10-1 示例程序 SsePackedIntegerFundamentals

```
Results for SsePiAddI16_
a:     10     200      30   -32766 |      50      60   32000   -32000
b:    100    -200   32760     -400 |     500    -600    1200     -950
c[0]: 110       0  -32746   32370 |     550    -540  -32336   32586

a:     10     200      30   -32766 |      50      60   32000   -32000
b:    100    -200   32760     -400 |     500    -600    1200      950
c[1]: 110       0   32767   -32768 |     550    -540   32767   -32768

Results for SsePiSubI32
a:          800          500 |         1000          900
b:          250         2000 |          -40         1200
c:          550         2500 |         1040         -300

Results for SsePiMul32_
a:           10           20 |           30           40
b:          100         -200 |          300         -400
c[0]:      1000        -4000 |        -9000        16000

a:           10           20 |           30           40
b:          100         -200 |          300         -400
c[1]:                   1000 |                     -9000
```

278

10.2 高级组合整数编程

x86-SSE 的组合整数计算能力经常被用于提升图像处理和计算机图形学算法的性能。本节我们将学习如何使用 x86-SSE 指令集构建一个图像的直方图。同时还会分析一个使用 SIMD 处理技术进行图像阈值分割的示例程序。

10.2.1 组合整数直方图

下面我们将学到的示例程序名叫 SsePackedIntegerHistogram，它会为一个包含 8 位灰度像素的图像建立一个饱和度直方图。图 10-1 展示了一个灰度图像的例子及其直方图。另外本节的示例程序还将演示如何动态分配内存缓冲区并为 x86-SSE 指令集进行必要的对齐操作。清单 10-3 和清单 10-4 分别给出了示例程序 SsePackedIntegerHistogram 的 C++ 和汇编语言源代码。

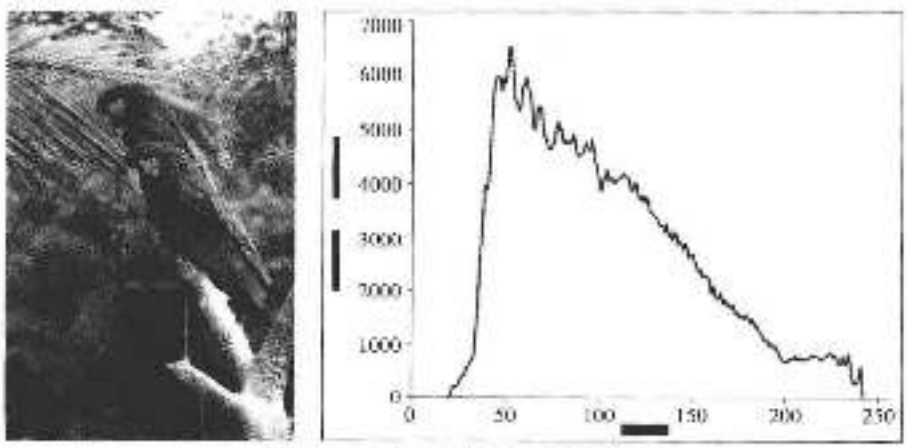


图 10-1 灰度图像示例及其直方图

清单 10-3 SsePackedIntegerHistogram.cpp

```
#include "stdafx.h"
#include "SsePackedIntegerHistogram.h"
#include <string.h>
#include <malloc.h>

279  extern "C" Uint32 NUM_PIXELS_MAX = 16777216;

bool SsePiHistogramCpp(Uint32* histo, const Uint8* pixel_buff, Uint32 num_pixels)
{
    // 确保 num_pixels 的值是合法的
    if ((num_pixels > NUM_PIXELS_MAX) || (num_pixels % 32 != 0))
        return false;

    // 确保 histo 按 16 字节边界对齐
    if (((uintptr_t)histo & 0xf) != 0)
        return false;

    // 确保 pixel_buff 按 16 字节边界对齐
    if (((uintptr_t)pixel_buff & 0xf) != 0)
        return false;

    // 创建直方图
    memset(histo, 0, 256 * sizeof(Uint32));

    for (Uint32 i = 0; i < num_pixels; i++)
        histo[pixel_buff[i]]++;

    return true;
}

void SsePiHistogram(void)
{
    const wchar_t* image_fn = L"..\\..\\..\\DataFiles\\TestImage1.bmp";
    const char* csv_fn = "..TestImage1_Histograms.csv";

    ImageBuffer ib(image_fn);
    Uint32 num_pixels = ib.GetNumPixels();
    Uint8* pixel_buff = (Uint8*)ib.GetPixelBuffer();
    Uint32* histo1 = (Uint32*) _aligned_malloc(256 * sizeof(Uint32), 16);
    Uint32* histo2 = (Uint32*) _aligned_malloc(256 * sizeof(Uint32), 16);
    bool rc1, rc2;

    rc1 = SsePiHistogramCpp(histo1, pixel_buff, num_pixels);
    rc2 = SsePiHistogram (histo2, pixel_buff, num_pixels);

    printf("Results for SsePiHistogram()\n");

    if (!rc1 || !rc2)
    {
        printf("  Bad return code: rc1=%d, rc2=%d\n", rc1, rc2);
        return;
280  }

    FILE* fp;
    bool compare_error = false;

    if (fopen_s(&fp, csv_fn, "wt") != 0)
        printf("  File open error: %s\n", csv_fn);
    else
    {
```

```
        for (Uint32 i = 0; i < 256; i++)
        {
            fprintf(fp, "%u, %u, %u\n", i, histo1[i], histo2[i]);

            if (histo1[i] != histo2[i])
            {
                printf("  Histogram compare error at index %u\n", i);
                printf("    counts: [%u, %u]\n", histo1[i], histo2[i]);
                compare_error = true;
            }
        }

        if (!compare_error)
            printf("  Histograms are identical\n");

        fclose(fp);
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    try
    {
        SsePiHistogram();
        SsePiHistogramTimed();
    }

    catch (...)
    {
        printf("Unexpected exception has occurred!\n");
        printf("File: %s (_tmain)\n", __FILE__);
    }

    return 0;
}
```

281

清单 10-4 SsePackedIntegerHistogram_.asm

```
        .model flat,c
        .code
        extern NUM_PIXELS_MAX:dword

; extern bool SsePiHistogram_(Uint32* histo, const Uint8* pixel_buff, ↵
Uint32 num_pixels);
;
; 描述：下面的函数创建一幅图像的直方图
;
; 返回值：0 表示参数值非法
;         1 表示成功
;
; 需要 SSE4.1 支持

SsePiHistogram_ proc
        push ebp
        mov ebp,esp
        and esp,0FFFFFFF0H                  ;将 ESP 对齐到 16 字节边界
        sub esp,1024                        ;分配 histo2 的空间
        mov edx,esp                         ;edx = histo2
        push ebx
        push esi
        push edi
```

```
; 确保num_pixels的值是合法的
        xor eax,eax                     ;设置返回代码为错误
        mov ecx,[ebp+16]                ;ecx = num_pixels
        cmp ecx,[MAX_PIXELS_MAX]
        ja Done                         ;如果num_pixels太大就跳转
        test ecx,1fh
        jnz Done                        ;如果num_pixels % 32 != 0就跳转

; 确保histo和pixel_buff正确对齐
        mov ebx,[ebp+8]                 ;ebx = histo
        test ebx,0fh
        jnz Done                        ;若未对齐就跳转
        mov esi,[ebp+12]                ;esi = pixel_buff
        test esi,0fh
        jnz Done                        ;若未对齐就跳转

; 初始化直方图缓冲区(将所有元素设为0)
        mov edi,ebx                     ;edi = histo
        mov ecx,256
        rep stosd                       ;初始化histo
        mov edi,edx                     ;edi = histo2
        mov ecx,256
282     rep stosd                       ;初始化histo2

; 为循环处理进行初始化操作
        mov edi,edx                     ;edi = histo2
        mov ecx,[ebp+16]                ;ecx = 像素个数
        shr ecx,5                       ;ecx = 像素块个数

; 创建直方图
; 寄存器使用情况: ebx = histo, edi = histo2, esi = pixel_buff
        align 16                        ;跳转目标对齐
@@:     movdqa xmm0,[esi]               ;加载像素块
        movdqa xmm2,[esi+16]            ;加载像素块
        movdqa xmm1,xmm0
        movdqa xmm3,xmm2

; 处理像素0 - 3
        pextrb eax,xmm0,0               ;提取像素0并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm1,1               ;提取像素1并对其计数
        add dword ptr [edi+edx*4],1
        pextrb eax,xmm0,2               ;提取像素2并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm1,3               ;提取像素3并对其计数
        add dword ptr [edi+edx*4],1

; 处理像素4 - 7
        pextrb eax,xmm0,4               ;提取像素4并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm1,5               ;提取像素5并对其计数
        add dword ptr [edi+edx*4],1
        pextrb eax,xmm0,6               ;提取像素6并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm1,7               ;提取像素7并对其计数
        add dword ptr [edi+edx*4],1

; 处理像素8 - 11
        pextrb eax,xmm2,8               ;提取像素8并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm3,9               ;提取像素9并对其计数
        add dword ptr [edi+edx*4],1
```

```
        pextrb eax,xmm0,10                  ;提取像素 10 并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm1,11                  ;提取像素 11 并对其计数
        add dword ptr [edi+edx*4],1

; 处理像素 12 - 15
        pextrb eax,xmm0,12                  ;提取像素 12 并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm1,13                  ;提取像素 13 并对其计数
        add dword ptr [edi+edx*4],1                                          283
        pextrb eax,xmm0,14                  ;提取像素 14 并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm1,15                  ;提取像素 15 并对其计数
        add dword ptr [edi+edx*4],1

; 处理像素 16 - 19
        pextrb eax,xmm2,0                   ;提取像素 16 并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm3,1                   ;提取像素 17 并对其计数
        add dword ptr [edi+edx*4],1
        pextrb eax,xmm2,2                   ;提取像素 18 并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm3,3                   ;提取像素 19 并对其计数
        add dword ptr [edi+edx*4],1

; 处理像素 20 - 23
        pextrb eax,xmm2,4                   ;提取像素 20 并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm3,5                   ;提取像素 21 并对其计数
        add dword ptr [edi+edx*4],1
        pextrb eax,xmm2,6                   ;提取像素 22 并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm3,7                   ;提取像素 23 并对其计数
        add dword ptr [edi+edx*4],1

; 处理像素 24 - 27
        pextrb eax,xmm2,8                   ;提取像素 24 并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm3,9                   ;提取像素 25 并对其计数
        add dword ptr [edi+edx*4],1
        pextrb eax,xmm2,10                  ;提取像素 26 并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm3,11                  ;提取像素 27 并对其计数
        add dword ptr [edi+edx*4],1

; 处理像素 28 - 31
        pextrb eax,xmm2,12                  ;提取像素 28 并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm3,13                  ;提取像素 29 并对其计数
        add dword ptr [edi+edx*4],1
        pextrb eax,xmm2,14                  ;提取像素 30 并对其计数
        add dword ptr [ebx+eax*4],1
        pextrb edx,xmm3,15                  ;提取像素 31 并对其计数
        add dword ptr [edi+edx*4],1

        add esi,32                          ;下一个像素块的指针
        sub ecx,1                           ;更新计数器
        jnz @B                              ;若未完成继续循环                 284

; 将 histo2 累加到 histo 中作为最后的直方图。注意，每次循环迭代都会累加 8 个直方图元素的值
        mov ecx,32                          ;迭代次数
        xor eax,eax                         ;histo 数组的偏移量
```

```
Sse:    movdqa xmm0,xmmword ptr [ebx+eax]   ;加载 histo 的计数
        movdqa xmm1,xmmword ptr [ebx+eax+16]

        paddd xmm0,xmmword ptr [edi+eax]    ;加载 histo2 中的计数
        paddd xmm1,xmmword ptr [edi+eax+16]

        movdqa xmmword ptr [ebx+eax],xmm0   ;保存最终的 histo 计数
        movdqa xmmword ptr [ebx+eax+16],xmm1

        add eax,32                          ;更新数组偏移量
        sub ecx,1                           ;更新计数器
        jnz @B                              ;若未完成继续循环
        mov eax,1                           ;设置成功返回码

Done:   pop edi
        pop esi
        pop ebx
        mov esp,ebp
        pop ebp
        ret
SsePiHistogram_ endp
        end
```

源文件 SsePackedIntegerHistogram.cpp（清单 10-3）的开始是一个名叫 SsePiHistogramCpp 的 C++ 函数，它构建了一个图像的直方图。函数首先检查 num_pixels 以确保它的值不大于 NUM_PIXELS_MAX 且能被 32 整除（整除检查是为了与对应的汇编语言直方图函数的逻辑相匹配）。接下来函数验证 histo 和 pixel_buff 的地址是否进行了适当的对齐，调用 memset 将直方图缓冲区中所有的像素计数器都清零，然后在一个简单的 for 循环中完成直方图的构建。

函数 SsePiHistogram 使用一个名叫 ImageBuffer 的 C++ 类来将图像文件中的像素加载到内存中（ImageBuffer 类的源代码在这里没有列出，但作为示例代码文件的一部分可供下载）。变量 num_pixels 和 pixel_buff 就是使用类 ImageBuffer 的成员函数初始化的。接下来，程序使用 Visual C++ 运行时函数 _aligned_malloc 动态分配两个直方图缓冲区。这个运行时函数有一个额外的参数可以让调用者指定所分配内存的对齐边界。后面的两个语句调用了 C++ 和汇编语言直方图函数。SsePiHistogram 中的其他代码会比较两个直方图是否相等并将结果写到一个 CSV 文件中。

285

x86-SSE 汇编语言版本的直方图函数名叫 SsePiHistogram_，位于文件 SsePackedIntegerHistogram_asm 中（程序清单 10-4）。SsePiHistogram_ 与相应的 C++ 函数不同，它会并行构建两个局部直方图，然后将二者合并生成最终的图像直方图。为了实现这个算法，SsePiHistogram_ 函数必须多分配一个直方图缓冲区。分配操作在函数序言中执行。在标准的 push ebp 和 mov ebp,esp 指令之后，有一条 and esp,0FFFFFFF0H 指令，会将 ESP 对齐到 16 字节边界。然后是 sub esp,1024 指令，在栈上为第二个直方图缓冲区分配存储空间。函数序言的剩余部分将非易变寄存器 EBX、ESI 和 EDI 的值保存在栈上。图 10-2 展示了 push edi 指令之后的栈结构。



图 10-2 SsePiHistogram_ 函数序言执行后的栈结构

在函数序言执行过后，与C++版本的程序一样，SsePiHistogram_会进行一些错误检查，包括验证num_pixels的值，确保histo和pixel_buff是边界对齐的。然后函数使用rep stosd指令将两个直方图缓冲区中的计数清零。在进入主循环之前，寄存器EBX指向histo，EDI指向histo2，ESI指向pixel_buff，寄存器ECX包含了图像中的32字节像素块的个数。

在主循环一开始，两个movdqa指令将下一批32个像素加载到寄存器XMM0和XMM2中。组合像素值还会被复制到XMM1和XMM3中以提升性能。pextrb eax, xmm0, 0（提取字节）指令从XMM0中提取第0个字节（或像素）然后将其复制到寄存器EAX的低位字节中（EAX的高位被清零）。add dword ptr [ebx+eax*4], 1指令将histo中对应的直方图元素加1。接下来pextrb edx, xmm0, 1指令从XMM0中提取第1个字节，add dword ptr [edi+edx*4],1指令更新对应的histo2中的直方图元素。这一系列的pextrb和add指令被重复使用，直到当前像素块中的所有32个字节都被处理过。在主循环中使用两个中间直方图，这种做法虽然看起来会导致性能降低，但实际上却比使用一个直方图缓冲区要快。原因是单个直方图缓冲区形成了内存方面的瓶颈，因为每次只有一个元素被更新。而双直方图的方法尽管还是从寄存器XMM0 ~ XMM3中提取单个像素值，但它却可以利用处理器乱序指令执行机制和内存高速缓冲机制来提升性能。在第21章，我们将学到更多关于处理器乱序指令执行和内存高速缓冲的知识。

在主循环结束后，由一系列的movdqa和paddd指令来计算中间直方图中的像素计数之和并创建最终的直方图。注意直方图求和的循环在每次迭代中都会累加8个无符号双字元素，这也能提升计算性能。输出10-2给出了示例程序SsePackedIntegerHistogram的运行结果。表10-1包含了一些计时测量数据。

输出 10-2 示例程序 SsePackedIntegerHistogram

```
Results for SsePiHistogram()
  Histograms are identical

Benchmark times saved to file _SsePackedIntegerHistogramTimed.csv
```

表 10-1 示例程序 SsePackedIntegerHistogram 中的直方图函数处理 TestImage1.bmp 的平均执行时间（单位：毫秒）

CPU	SsePiHistogramCpp (C++)	SsePiHistogram_ (x86-SSE)
Intel Core i7-4770	296	235
Intel Core i7-4600U	351	277
Intel Core i3-2310M	668	485

10.2.2 组合整数阈值分割

我们要学习的最后一个x86-SSE组合整数示例程序名叫SsePackedIntegerThreshold。这个示例程序将展示如何利用x86-SSE指令集来执行一种称为阈值分割的常见图像处理技术。该程序展示了如何在一个灰度图像中计算选定像素的平均饱和度。清单10-5、清单10-6和清单10-7给出了示例程序SsePackedIntegerThreshold的C++和汇编语言源代码。

清单 10-5 SsePackedIntegerThreshold.h

```
#pragma once
#include "ImageBuffer.h"

// 图像阈值分割数据结构。该结构必须与 SsePackedIntegerThreshold_.asm 中所定义的结构保持一致
typedef struct
{
    Uint8* PbSrc;               // 源图像像素缓冲区
    Uint8* PbMask;              // 掩码 mask 的像素缓冲区
    Uint32 NumPixels;           // 源图像的像素个数
    Uint8 Threshold;            // 图像阈值分割的数值
    Uint8 Pad[3];               // 占满备用
    Uint32 NumMaskedPixels;     // 被掩码的像素数量
    Uint32 SumMaskedPixels;     // 被掩码的像素和
    double MeanMaskedPixels;    // 被掩码像素的平均值
} ITD;

// 定义在 SsePackedIntegerThreshold.cpp 中的函数
extern bool SsePiThresholdCpp(ITD* itd);
extern bool SsePiCalcMeanCpp(ITD* itd);

// 定义在 SsePackedIntegerThreshold_.asm 中的函数
extern "C" bool SsePiThreshold_(ITD* itd);
extern "C" bool SsePiCalcMean_(ITD* itd);

// 定义在 SsePackedIntegerThresholdTimed.cpp 中的函数
extern void SsePiThresholdTimed(void);

// 其他常量
const Uint8 TEST_THRESHOLD = 96;
```

288

清单 10-6 SsePackedIntegerThreshold.cpp

```
#include "stdafx.h"
#include "SsePackedIntegerThreshold.h"
#include <stddef.h>

extern "C" Uint32 NUM_PIXELS_MAX = 16777216;

bool SsePiThresholdCpp(ITD* itd)
{
    Uint8* pb_src = itd->PbSrc;
    Uint8* pb_mask = itd->PbMask;
    Uint8 threshold = itd->Threshold;
    Uint32 num_pixels = itd->NumPixels;

    // 确保 num_pixels 值是合法的
    if ((num_pixels == 0) || (num_pixels > NUM_PIXELS_MAX))
        return false;
    if ((num_pixels & 0x1f) != 0)
        return false;

    // 确保图像缓冲区是正确对齐的
    if (((uintptr_t)pb_src & 0xf) != 0)
        return false;
    if (((uintptr_t)pb_mask & 0xf) != 0)
        return false;

    // 对图像进行阈值分割
    for (Uint32 i = 0; i < num_pixels; i++)
        *pb_mask++ = (*pb_src++ > threshold) ? 0xff : 0x00;
```

```
    return true;
}

bool SsePiCalcMeanCpp(ITD* itd)
{
    Uint8* pb_src = itd->PbSrc;
    Uint8* pb_mask = itd->PbMask;
    Uint32 num_pixels = itd->NumPixels;

    // 确保 num_pixels 仍是合法的
    if ((num_pixels == 0) || (num_pixels > NUM_PIXELS_MAX))
        return false;
    if ((num_pixels & 0x1f) != 0)
        return false;                                          289

    // 确保两个缓冲区是适当对齐的
    if (((uintptr_t)pb_src & 0xf) != 0)
        return false;
    if (((uintptr_t)pb_mask & 0xf) != 0)
        return false;

    // 计算被掩码像素的平均值
    Uint32 sum_masked_pixels = 0;
    Uint32 num_masked_pixels = 0;

    for (Uint32 i = 0; i < num_pixels; i++)
    {
        Uint8 mask_val = *pb_mask++;
        num_masked_pixels += mask_val & 1;
        sum_masked_pixels += (*pb_src++ & mask_val);
    }

    itd->NumMaskedPixels = num_masked_pixels;
    itd->SumMaskedPixels = sum_masked_pixels;

    if (num_masked_pixels > 0)
        itd->MeanMaskedPixels = (double)sum_masked_pixels /←
num_masked_pixels;
    else
        itd->MeanMaskedPixels = -1.0;

    return true;
}

void SsePiThreshold()
{
    wchar_t* fn_src = L"..\\..\\..\\DataFiles\\TestImage2.bmp";
    wchar_t* fn_mask1 = L"_TestImage2_Mask1.bmp";
    wchar_t* fn_mask2 = L"_TestImage2_Mask2.bmp";
    ImageBuffer* im_src = new ImageBuffer(fn_src);
    ImageBuffer* im_mask1 = new ImageBuffer(*im_src, false);
    ImageBuffer* im_mask2 = new ImageBuffer(*im_src, false);
    ITD itd1, itd2;

    itd1.PbSrc = (Uint8*)im_src->GetPixelBuffer();
    itd1.PbMask = (Uint8*)im_mask1->GetPixelBuffer();
    itd1.NumPixels = im_src->GetNumPixels();
    itd1.Threshold = TEST_THRESHOLD;

    itd2.PbSrc = (Uint8*)im_src->GetPixelBuffer();
    itd2.PbMask = (Uint8*)im_mask2->GetPixelBuffer();
    itd2.NumPixels = im_src->GetNumPixels();
    itd2.Threshold = TEST_THRESHOLD;                           290
```

```
    bool rc1 = SsePiThresholdCpp(&itd1);
    bool rc2 = SsePiThreshold_(&itd2);

    if (!rc1 || !rc2)
    {
        printf("Bad Threshold return code: rc1=%d, rc2=%d\n", rc1, rc2);
        return;
    }

    in_mask1->SaveToBitmapFile(fn_mask1);
    in_mask2->SaveToBitmapFile(fn_mask2);

    // 计算被掩码像素的平均值
    rc1 = SsePiCalcMeanCpp(&itd1);
    rc2 = SsePiCalcMean_(&itd2);

    if (!rc1 || !rc2)
    {
        printf("Bad CalcMean return code: rc1=%d, rc2=%d\n", rc1, rc2);
        return;
    }

    printf("Results for SsePackedIntegerThreshold\n\n");
    printf("                              C++       X86-SSE\n");
    printf("-----------------------------------------\n");
    printf("SumPixelsMasked:  ");
    printf("%12u  %12u\n", itd1.SumMaskedPixels, itd2.SumMaskedPixels);
    printf("NumPixelsMasked:  ");
    printf("%12u  %12u\n", itd1.NumMaskedPixels, itd2.NumMaskedPixels);
    printf("MeanPixelsMasked: ");
    printf("%12.6lf  %12.6lf\n", itd1.MeanMaskedPixels, itd2.MeanMaskedPixels);

    delete in_src;
    delete in_mask1;
    delete in_mask2;
}

int _tmain(int argc, _TCHAR* argv[])
{
    try
    {
        SsePiThreshold();
        SsePiThresholdTimed();
    }
    catch (...)
    {
        printf("Unexpected exception has occurred!\n");
        printf("File: %s (_tmain)\n", __FILE__);
    }
    return 0;
}
```

291

清单 10-7 SsePackedIntegerThreshold_.asm

```
        .model flat,c
        extern NUM_PIXELS_MAX:dword

; 图像阈值分割处理结构(从 SsePackedIntegerThreshold.h)
ITD             struct
PbSrc           dword ?
PbMask          dword ?
```

```
NumPixels           dword ?
Threshold           byte ?
Pad                 byte 3 dup(?)
NumMaskedPixels     dword ?
SumMaskedPixels     dword ?
MeanMaskedPixels    real8 ?
ITD                 ends

                .const
                align 16
PixelScale      byte 16 dup(80h)            ;uint8 到 int8 的缩放值
CountPixelsMask byte 16 dup(01h)            ;用于像素计数的掩码
R8_MinusOne     real8 -1.0                  ;非法的平均值
                .code

; extern "C" bool SsePiThreshold_(ITD* itd);
;
; 描述：下面的函数对一个灰度图（每像素 8 比特）进行图像阈值分割操作
;
; 返回值：0 表示非法的大小或未对齐的图像缓冲区
;         1 表示成功
;
; SSE 版本：SSSE3

SsePiThreshold_ proc
        push ebp
        mov ebp,esp
        push esi
        push edi                                                    292

; 加载并验证 ITD 结构中的参数值
        mov edx,[ebp+8]                 ;edx = 'itd'
        xor eax,eax                     ;设置错误返回码
        mov ecx,[edx+ITD.NumPixels]     ;ecx = NumPixels
        test ecx,ecx
        jz Done                         ;若 num_pixels == 0 则跳转
        cmp ecx,[NUM_PIXELS_MAX]
        ja Done                         ;若 num_pixels 太大则跳转
        test ecx,0fh
        jnz Done                        ;若 num_pixels % 16 != 0 则跳转
        shr ecx,4                       ;组合像素的个数

        mov esi,[edx+ITD.PbSrc]         ;esi = PbSrc
        test esi,0fh
        jnz Done                        ;若未对齐则跳转
        mov edi,[edx+ITD.PbMask]        ;edi = PbMask
        test edi,0fh
        jnz Done                        ;若未对齐则跳转

; 初始化组合阈值
        movzx eax,byte ptr [edx+ITD.Threshold]  ;eax = 单个阈值
        movd xmm1,eax                           ;xmm1[7:0] = 单个阈值
        pxor xmm0,xmm0                          ;pshufb 的掩码
        pshufb xmm1,xmm0                        ;组合阈值
        movdqa xmm2,xmmword ptr [PixelScale]
        psubb xmm1,xmm2                         ;组合阈值

; 创建掩码图像
@@:     movdqa xmm0,[esi]               ;加载下一个组合像素
        psubb xmm0,xmm2                 ;偏移过的图像像素
        pcmpgtb xmm0,xmm1               ;与阈值对比
        movdqa [edi],xmm0               ;保存组合阈值掩码
```

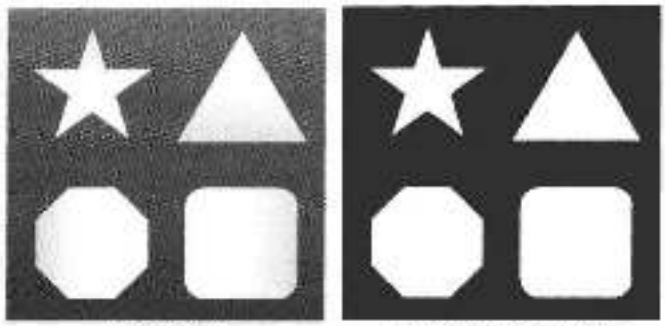
```
        add esi,16
        add edi,16
        dec ecx
        jnz @B                          ;循环直至结束
        mov eax,1                       ;设置返回码

Done:   pop edi
        pop esi
        pop ebp
        ret
293     SsePiThreshold_ endp

; extern "C" bool SsePiCalcMean_(ITD* itd);
;
; 描述：下面的函数使用函数 ssePiThreshold 创建的掩码计算所有高于阈值的图像像素的平均值
;
; 返回值：0 表示非法的图像大小或未对齐的图像缓冲区
;         1 表示成功
;
; 需要 SSSE3 支持

SsePiCalcMean_  proc
        push ebp
        mov ebp,esp
        push ebx
        push esi
        push edi

; 加载并验证 ITD 结构中的参数值
        mov eax,[ebp+8]                 ;eax = 'itd'
        mov ecx,[eax+ITD.NumPixels]     ;ecx = NumPixels
        test ecx,ecx
        jz Error                        ;若 num_pixels == 0 则跳转
        cmp ecx,[NUM_PIXELS_MAX]
        ja Error                        ;若 num_pixels 太大则跳转
        test ecx,0fh
        jnz Error                       ;若 num_pixels % 16 != 0 则跳转
        shr ecx,4                       ;ecx = 组合像素对个数

        mov edi,[eax+ITD.PbMask]        ;edi = PbMask
        test edi,0fh
        jnz Error                       ;若 PbMask 未对齐则跳转
        mov esi,[eax+ITD.PbSrc]         ;esi = PbSrc
        test esi,0fh
        jnz Error                       ;若 PbSrc 未对齐则跳转

; 为平均值计算初始化数值
        xor edx,edx             ;更新次数
        pxor xmm7,xmm7          ;组合 0 值

        pxor xmm2,xmm2          ;xmm2 = sum_masked_pixels (8 字)
        pxor xmm3,xmm3          ;xmm3 = sum_masked_pixels (8 字)
        pxor xmm4,xmm4          ;xmm4 = sum_masked_pixels (4 双字)

        pxor xmm6,xmm6          ;xmm6 = num_masked_pixels (8 字节)
294     xor ebx,ebx             ;ebx = num_masked_pixels (1 双字)

; 循环处理的寄存器使用
; esi = PbSrc, edi = PbMask, eax = itd
; ebx = num_pixels_masked, ecx = NumPixels / 16, edx = 更新次数
;
; xmm0 - 组合像素, xmm1 - 组合掩码
```

```
; xmm3:xmm2 - sum_masked_pixels (16字)
; xmm4 - sum_masked_pixels (4双字)
; xmm5 - 单像素掩码
; xmm6 - 组合比较的 num_masked_pixels
; xmm7 - 组合 0 值

@@:     movdqa xmm0,xmmword ptr [esi]       ;加载下一个组合像素
        movdqa xmm1,xmmword ptr [edi]       ;加载下一个组合比较码

; 更新 sum_masked_pixels (双字节值)
        movdqa xmm5,xmmword ptr [CountPixelsMask]
        pand xmm5,xmm1
        paddb xmm6,xmm5                 ;更新 num_masked_pixels
        pand xmm0,xmm1                  ;将未被掩码的像素设为 0
        movdqa xmm1,xmm0
        punpcklbw xmm0,xmm7
        punpckhbw xmm1,xmm7             ;被掩码的像素 (双字节值)
        paddw xmm2,xmm0
        paddw xmm3,xmm1                 ;xmm3:xmm2 - sum_masked_pixels

; 检查是否需要更新 xmm4 中的双字 sum_masked_pixels 值和 ebx 中的 num_masked_pixels 值
        inc edx
        cmp edx,255
        jb NoUpdate
        call SsePiCalcMeanUpdateSums
NoUpdate:
        add esi,16
        add edi,16
        dec ecx
        jnz @@                          ;循环直至结束

; 主循环结束。若有必要，对 xmm4 中的 sum_masked_pixels 值和 ebx 中的 num_masked_pixels 值做最后
  一次更新
        test edx,edx
        jz @F
        call SsePiCalcMeanUpdateSums

; 计算并保存最终的 sum_masked_pixels 和 num_masked_pixels 的值
@@:     phaddd xmm4,xmm7
        phaddd xmm4,xmm7
        movd edx,xmm4                   ;最终的 sum_mask_pixels 值
        mov [eax+ITD.SumMaskedPixels],edx   ;保存最终的 sum_masked_pixels
        mov [eax+ITD.NumMaskedPixels],ebx   ;保存最终的 num_masked_pixels

; 计算被掩码像素的平均值
        test ebx,ebx                    ;num_mask_pixels 是否为 0?
        jz NoMean                       ;若为 0，则过平均值的计算
        cvtsi2sd xmm0,edx               ;xmm0 - sum_masked_pixels
        cvtsi2sd xmm1,ebx               ;xmm1 - num_masked_pixels
        divsd xmm0,xmm1                 ;xmm0 - mean_masked_pixels
        jmp @F
NoMean: movsd xmm0,[R8_MinusOne]            ;使用 -1.0 表示无平均值
@@:     movsd [eax+ITD.MeanMaskedPixels],xmm0   ;保存平均值
        mov eax,1                       ;设置返回码

Done:   pop edi
        pop esi
        pop ebx
        pop ebp
        ret

Error:  xor eax,eax                     ;设置错误返回码
        jmp Done
```

```
SsePiCalcMean_ endp

; void SsePiCalcMeanUpdateSums
;
; 描述：下面的例程更新 xmm4 中的 sum_masked_pixels 和 ebx 中的 num_masked_pixels，为了防止溢出的
;       发生，它同时还会对累加的中间值
;
; 寄存器内容：
;   xmm3:xmm2 = 组合过的双字节 sum_masked_pixels 值
;   xmm4 = 组合过的四字节 sum_masked_pixels 值
;   xmm6 = 组合 num_masked_pixels
;   xmm7 = 组合 0 值
;   ebx = num_masked_pixels
;
; 临时寄存器：
;   xmm0, xmm1, xmm5, edx

SsePiCalcMeanUpdateSums proc private

; 将组合字（双字节）sum_masked_pixels 扩展到双字（四字节）
        movdqa xmm0,xmm2
        movdqa xmm1,xmm5
        punpcklwd xmm0,xmm7
        punpcklwd xmm1,xmm7
        punpckhwd xmm2,xmm7
296     punpckhwd xmm3,xmm7

; 更新 sum_masked_pixels 已组合双字和
        paddd xmm0,xmm1
        paddd xmm2,xmm3
        paddd xmm4,xmm0
        paddd xmm4,xmm2             ; xmm4 = 组合过的 sum_masked_pixels

; 计算 xmm6 中 num_masked_pixel 已组合（单字节）的和，并将结果累加到 ebx 中
        movdqa xmm5,xmm6
        punpcklbw xmm5,xmm7
        punpckhbw xmm6,xmm7         ; xmm5 = 组合过的 num_masked_pixels
        paddw xmm6,xmm5             ; xmm6 = 组合过的 num_masked_pixels
        phaddw xmm6,xmm7
        phaddw xmm6,xmm7
        phaddw xmm6,xmm7            ; xmm6[15:0] = 最终的和（双字节）
        movd edx,xmm6
        add ebx,edx                 ;ebx = num_masked_pixels

; 重置中间值
        xor edx,edx
        pxor xmm2,xmm2
        pxor xmm3,xmm3
        pxor xmm6,xmm6
        ret
SsePiCalcMeanUpdateSums endp
        end
```

图像阈值分割是一种常用的图像处理技术，它可以创建一个灰度图的二值化图像。这个二值化图像（或称掩码图像）表征了原始图像中的那些像素的饱和度大于预设的（或算法推演的）饱和度阈值。图 10-3 展示了一个阈值分割的实例。掩码图像经常被用来对原始灰度图进行各种附加计算。例如，图 10-3 演示了掩码图像的一种典型应用，在原始灰度图中计算饱和度高于某阈值的像素的平均饱和度。掩码图像的应用简化了平均饱和度的计算，因为它利用简单的布尔表达式将不需要参与计算的像素排除在外。本节的示例程序就展示

了这种方法。 297



原始灰度图　　阈值分割后的掩码图像

图 10-3　灰度图实例及其掩码图像

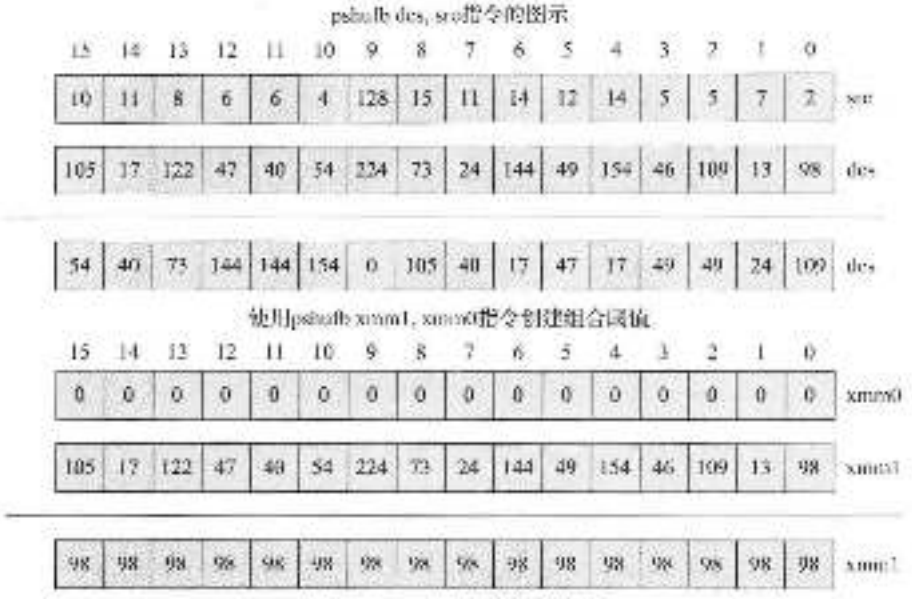
SsePackedIntegerThreshold 示例程序使用的算法包含两个阶段。第一阶段构建了图 10-3 所示的掩码图像。第二阶段计算了掩码图像中所有白色像素的平均饱和度。文件 SsePackedIntegerThreshold.h（见清单 10-5）中定义了一个名为 ITD 的结构体，用来记录算法所需的数据。注意，这个结构体包含了两个像素计数相关的变量：NumPixels 和 NumMaskedPixels。前一个变量表示总的像素数量，后一个变量用于记录灰度图像的大于 Threshold（也是结构体变量）的像素个数。

C++ 源文件 SsePackedIntegerThreshold.cpp（见清单 10-6）包含了独立的阈值分割和平均值计算函数。函数 SsePiThresholdCpp 将灰度图的每个像素值与 itd->Threshold 指定的阈值相比较，从而构建灰度图的掩码图像。如果一个灰度图的像素值大于相应阈值，其对应的掩码图像像素就被设成 0xff；否则，其掩码图像像素被设成 0x00。函数 SsePiCalcMeanCpp 使用前面得出的掩码图像来计算灰度图中所有像素值大于某阈值的像素的平均饱和度。注意函数中的 for 循环使用简单的布尔表达式而不是逻辑比较语句来计算 num_mask_pixels 和 sum_mask_pixels 的值。后一种方法通常会更快，且容易使用 SIMD 算术来实现。这一点我们很快就会看到。

清单 10-7 是阈值分割和平均饱和度计算的汇编版本。在序言之后，函数 SsePiThreshold_对 ITD.NumPixels、ITD.PbSrc 和 ITD.PbMask 进行合法性检查。movzx eax,byte ptr[edx+ITD.Threshold] 指令将给定的阈值复制到寄存器 EAX 中。接下来在 movd xmm1,eax 指令之后，函数使用 pshufb（Packed Shuffle Bytes，组合移动字节）指令来创建一个组合阈值。pshufb 指令使用源操作数中的每个字节的低位四比特，把它们作为索引值来置换目标操作数中的字节（如果高位在源操作数中被置位，则默认为 0）。该过程如图 10-4 所示。在当前程序中，指令 pshufb xmm1,xmm0 将 XMM1[7:0] 中的值复制到 XMM1 中的每个字节元素中。因为 XMM0 中包含的全部为 0。接下来组合阈值会被缩放以便在主循环中使用。 298

函数 SsePiThreshold 中的主循环使用 pcmpgtb（Compare Packed Signed Integers for Greater Than，比较组合有符号整数看是否大于）指令来创建掩码图像。这个指令对目标操作数和源操作数中的字节进行两两对比，如果目标操作数中的字节大于与之对应的源操作数的字节，则目标操作数中的字节会被设为 0xff；否则将其设为 0x00，如图 10-5 所示。这里需要注意，pcmpgtb 指令进行的是有符号整数的算术比较。而灰度图中的像素值是无符号字节值，这意味着像素值必须被转换以保持和 pcmpgtb 指令的兼容。psubb xmm0,xmm2 指令将存放于 XMM0 中的灰度图像素值从 [0, 255] 重新映射到 [-128, 127]。之后，movdqa 指令

将结果保存到掩码图像缓冲区中。



299 图 10-4 psbufb 指令的图示

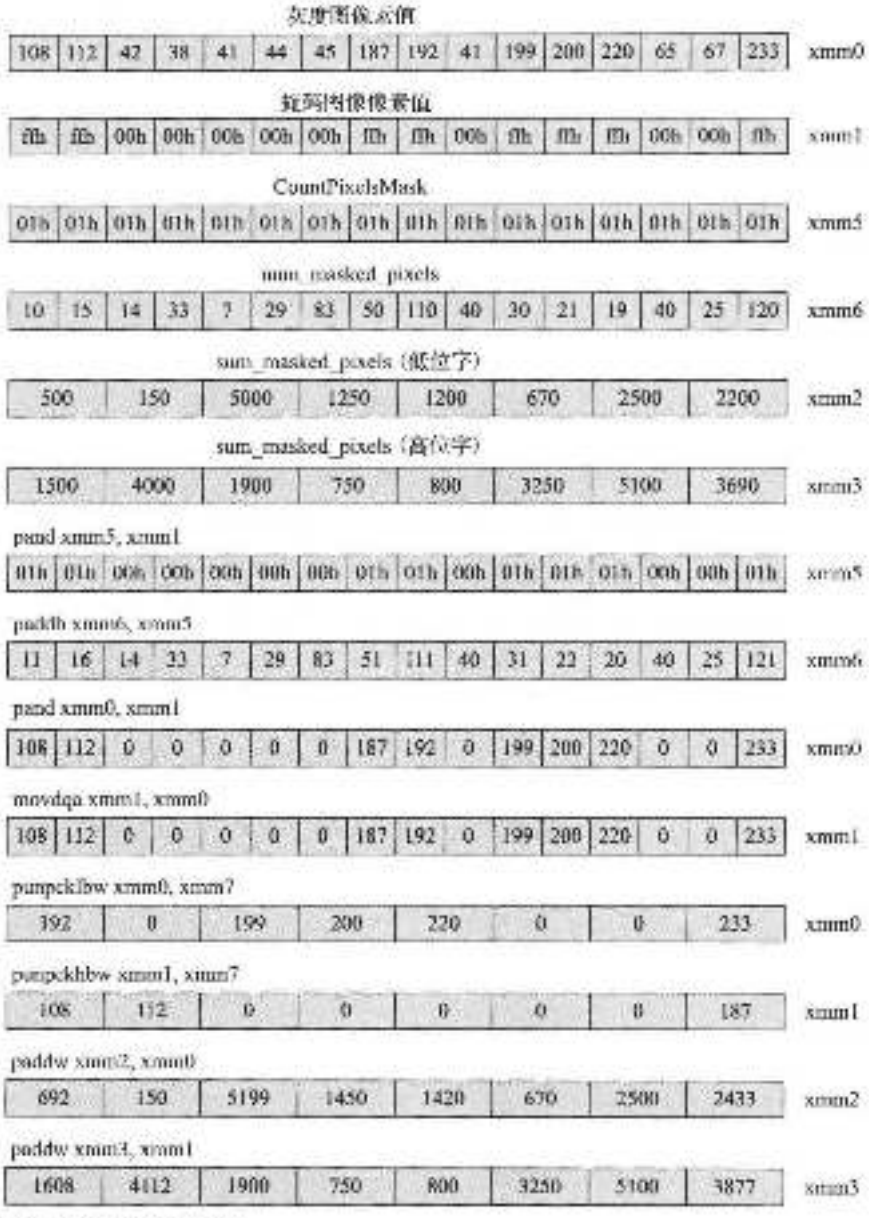


图 10-5 pcmpgtb 指令的图示

与 C++ 程序中对应的函数类似，汇编语言函数 SsePiCalcMean_ 用来计算灰度图中所有高于某阈值的像素的平均饱和度。为了计算所需的像素总和，函数主循环使用无符号双字节和无符号四字节值来操作两个中间组合像素值的和。这样就将必须进行的组合单字节到双字节和双字节到四字节值的转化次数降到最低。在每次迭代中，XMM0 中的高于阈值的灰度像素值会被转换成双字节值，并被累加到 XMM3:XMM2 中的组合双字节中。主循环还会更新保存在 XMM6 中的高于阈值的像素的个数。图 10-6 显示了进行以上运算的代码段的执行过程。注意，图 10-6 中采用的方法本质上是对 C++ 函数 SsePiCalcMeanCpp 中的 for 循环中
300 使用的技术的一种基于 SIMD 的实现。

每经过 255 次迭代，XMM3:XMM2 中的组合字（双字节）像素和就会被转换为组合双字类型，并累加到 XMM4 的组合双字像素和中。XMM6 中的组合单字节像素计数也会被汇总并累加到 EBX 中，EBX 中包含了变量 num_masked_pixels（255 次迭代的限制可以防止 XMM6 中的组合单字节像素计数的算术运算溢出）。这些计算都在一个名为 SsePiCalcMeanUpdateSums 的私有函数中进行。循环一直会执行，直到图像中所有的像素都被处理过。在循环结束以后，程序使用 phaddd 指令将 XMM4 中的四个四字节像素饱和度值简化到最终的变量 sum_masked_pixels 中。随后，函数使用两个 cvtsi2sd 指令将 EDX（sum_

masked_pixels）和 EBX（num_mask_pixels）转换为双精度浮点数；然后利用 divsd 指令计算 ITD.MeanMaskedPixels。输出 10-3 显示了示例程序 SsePackedIntegerThreshold 的执行结果。表 10-2 中显示了其执行时间的测量值。



灰度图像亮值

108	112	42	38	41	44	45	187	192	41	199	200	220	65	67	233	xmm0

短码图像像素值

ffh	ffh	00h	00h	00h	00h	00h	ffh	ffh	00h	ffh	ffh	ffh	00h	00h	ffh	xmm1

CountPixelsMask

01h	01h	01h	01h	01h	01h	01h	01h	01h	01h	01h	01h	01h	01h	01h	01h	xmm5

num_masked_pixels

10	15	14	33	7	29	83	50	110	40	30	21	19	40	25	120	xmm6

sum_masked_pixels（低位字）

500	150	5000	1250	1200	670	2500	2200	xmm2

sum_masked_pixels（高位字）

1500	4000	1900	750	800	3250	5100	3690	xmm3

pand xmm5, xmm1

01h	01h	00h	00h	00h	00h	00h	01h	01h	00h	01h	01h	01h	00h	00h	01h	xmm5

paddb xmm6, xmm5

11	16	14	33	7	29	83	51	111	40	31	22	20	40	25	121	xmm6

pand xmm0, xmm1

108	112	0	0	0	0	0	187	192	0	199	200	220	0	0	233	xmm0

movdqa xmm1, xmm0

108	112	0	0	0	0	0	187	192	0	199	200	220	0	0	233	xmm1

punpcklbw xmm0, xmm7

192	0	199	200	220	0	0	233	xmm0

punpckhbw xmm1, xmm7

108	112	0	0	0	0	0	187	xmm1

paddw xmm2, xmm0

692	150	5199	1450	1420	670	2500	2433	xmm2

paddw xmm3, xmm1

1608	4112	1900	750	800	3250	5100	3877	xmm3

注：XMM7 中保存的数值为全 0。

图 10-6 像素和以及像素计数的计算过程

输出 10-3 示例程序 SsePackedIntegerThreshold

```
Results for SsePackedIntegerThreshold

                          C++        X86-SSE
-------------------------------------------
SumPixelsMasked:      23813043      23813043
```

```
NumPixelsMasked:        138220        138220
MeanPixelsMasked:    172.283628    172.283628

Benchmark times saved to file __SsePackedImageThresholdTimed.csv
```

表 10-2 示例程序 SsePackedImageThreshold 中使用的算法针对 TestImage2.bmp 的平均执行时间（单位：微秒）

CPU	C++	x86-SSE
Intel Core i7-4770	515	49
Intel Core i7-4600U	608	60
Intel Core i3-2310M	1199	108

10.3 总结

本章我们学习了如何使用 x86-SSE 中的组合整数功能进行基本的算术运算。我们还分析了一些示例程序，它们演示了通用图像处理算法的加速实现。下一章我们将继续学习 x86-SSE 编程，重点学习处理文本串的 x86-SSE 指令的用法。

x86-SSE 编程——字符串

前面二章的示例代码集中展示了如何利用 x86-SSE 指令集实现数值算法。在这一章里，我们将学习 x86-SSE 的字符串处理指令。x86-SSE 的字符串处理指令与 x86-SSE 的其他指令有所不同，因为它们的执行依赖于一个立即数控制字节。11.1 节讲述控制字节的不同选项及操作范例，同时还将讨论在使用 SIMD 技术处理字符串时需要注意的一些事项。11.2 节中，我们将学习若干个示例程序，这些程序展示了 x86-SSE 字符串处理指令的基本用法。

支持 SSE4.2 的处理器都具备 x86-SSE 字符串处理指令，如 AMD FX 和 Intel 酷睿处理器家族。读者可以使用附录 C 中提及的工具来确认你的 PC 处理器是否支持 SSE4.2。在第 16 章，我们将学习如何使用 cpuid 指令在运行时检测处理器所支持的功能和扩展（如 SSE4.2）。

11.1 字符串基础知识

编程语言通常使用两种方式管理和处理文本字符串，即显式长度（explicit-length）和隐式长度（implicit-length）。显式长度字符串由一系列连续排列的字符组成，它们的长度预先已经算好并与字符串本身一同管理。比如 PASCAL 就使用这种管理方法。隐式长度字符串使用一个字符串终结符（EOS，通常为 0）来表示字符串结束，同时该终结符还可用于辅助一些字符串处理函数，如字符串长度计算和字符串连接等。这种管理方法被用在 C++ 中，第 2 章中曾讨论过（示例程序 CountChars 和 ConcatStrings）。

字符串处理容易导致较高的 CPU 占用率，可能比我们想象的还要高。主要的原因是很多字符串处理函数需要逐个字符或以若干个字符组合起来对字符串进行处理。另外，字符串处理函数还广泛使用循环结构，这导致处理器的前端指令管道的使用不够优化。本节将要介绍的 x86-SSE 字符串处理指令可以用于加速很多通用的字符串基础运算，包括长度计算，比较操作和子串查找等。这些指令还可以显著提高字符串搜索和语法解析的效率。

303

x86-SSE 包含四个 SIMD 字符串指令，它们都能够处理最长为 128 位的字符串片段。这些指令（表 11-1）既支持显式长度又支持隐式长度的字符串。表中有两种输出格式选项——index 和 mask，这些选项的含义后面会讲到。处理器使用 EFLAGS 寄存器中的状态位来通报字符串指令的附加结果。每一个 x86-SSE 字符串指令都需要一个 8 位的立即数控制字节，这个值可以让程序员来选择指令选项，包括字符的大小（8 位或 16 位）、字符比较和聚合方法，以及输出格式。由于 C++ 使用隐式长度字符串，接下来的说明主要集中在 pcmpistri 和 pcmpistrm 两个指令上。显式长度指令 pcmpestri 和 pcmpestrm 本质上是一样的，只是字符串片段的长度必须用寄存器 EAX 和 EDX 来指定。

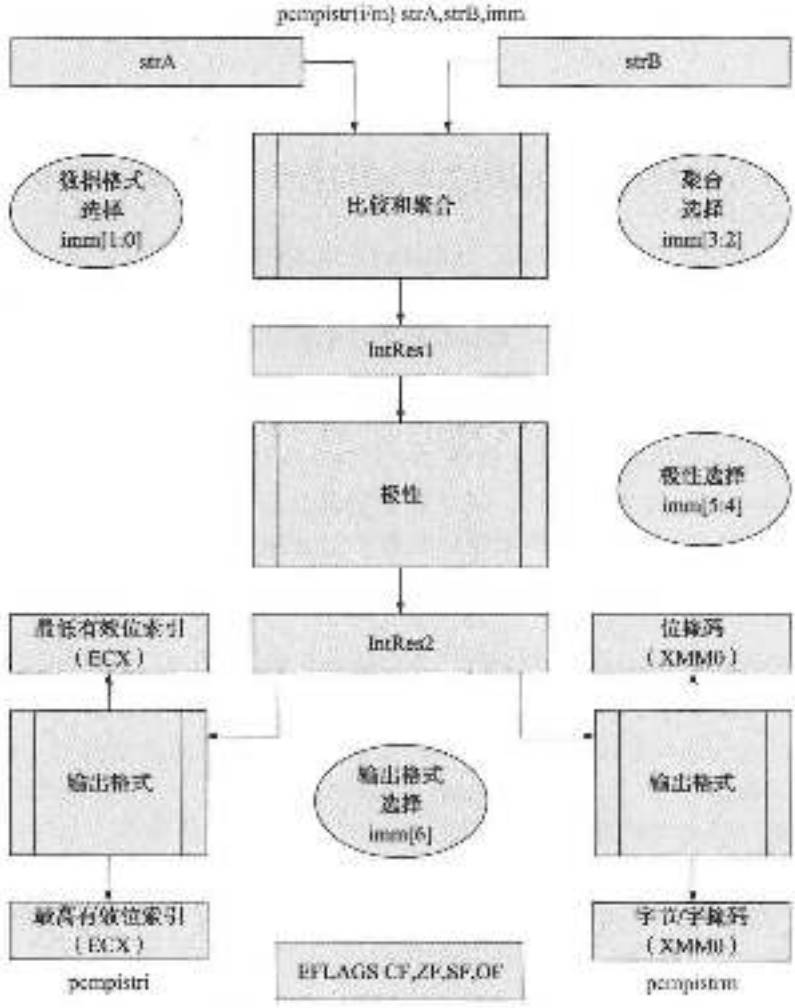
表 11-1 x86-SSE 字符串指令总结

助记符	字符串类型	输出格式
pcmpestri	显式长度	Index (ECX)
pcmpestrm	显式长度	Mask (XMM0)
pcmpistri	隐式长度	Index (ECX)
pcmpistrm	隐式长度	Mask (XMM0)

x86-SSE 的字符串指令非常强大和灵活，但缺点是指令复杂，使一些程序员在开始接触时很困惑。为了尽量减少这种困惑，我们会集中讲解少数几个常用的字符串处理操作。这样可以为更高级的字符串指令的使用打下坚实的基础。如果读者想进一步学习高级 x86-SSE 字符串指令，请阅读 Intel 和 AMD 的参考手册，详细信息已经在附录 C 中给出。

图 11-1 显示了指令 pcmpistri 和 pcmpistrm 的执行流程图。操作数 strA 和 strB 都是源操作数，能够存放字符串片段或单独的字符值。操作数 strA 必须是一个 XMM 寄存器，操作数 strB 可以是一个 XMM 寄存器或存放于内存中的 128 位数值。操作数 imm 用于指定指令的控制选项，如流程图中的椭圆框所示。表 11-2 描述了每一个 imm 控制选项的含义。

304



305

图 11-1 pcmpistrX 指令的流程图

表 11-2 pcmpistrX 指令的控制选项含义

控制选项	值	描述
Data Format[1:0]	00	组合无符号单字节
	01	组合无符号双字节
	10	组合有符号单字节
	11	组合有符号双字节
Aggregation[3:2]	00	任意等价（匹配字符）
	01	区间等价（匹配某范围内的字符）
	10	逐个等价（字符串比较）
	11	有序等价（子串查找）
Polarity[5:4]	00	取正（IntRes2[i] = IntRes1[i]）
	01	取负（IntRes2[i] = -IntRes1[i]）
	10	掩码取正（IntRes2[i] = IntRes1[i]）
	11	掩码取负（若 str2[i] 非法，IntRes2[i] = IntRes1[i]；否则 IntRes2[i] = -IntRes1[i]）
Output Format[6]	0	pcmpistri-ECX：IntRes2 中被置位为最低位的索引值
	1	pcmpistri-ECX：IntRes2 中被置位的最高位的索引值
	0	pcmpistrm-IntRes2 被保存为 XMM0 中的位掩码（高位被清零）
	1	pcmpistrm-IntRes2 被保存为 XMM0 中的单字节或双字节掩码

为了更好地理解 pcmpistrX 指令的流程图以及每个控制选项的含义，我们看几个简单的例子。假设有一个字符串片段，我们需要创建一个掩码来指定字符串中大写字符的位置。例如，掩码 1000110000010010b 中的每个 1 都表示字符串“Ab1cDE23f4gHi5J6”中对应位置上的字符是大写字符。我们可以创建一个 C++ 函数来生成这样的掩码。这样的函数需要一个循环来扫描和检测字符串中的每个字符，看它的值是否位于 A 到 Z 的区间内（含 A 和 Z）。另外一个（或许是更好的）途径是使用 pcmpistrm 指令来生成这个掩码，如图 11-2 所示。在这个例子里，所需的字符区间和字符串片段被分别加载到寄存器 XMM1 和 XMM2 中。立即数控制值 00000100b 指定 pcmpistrm 指令使用表 11-3 中描述的选项执行。 306

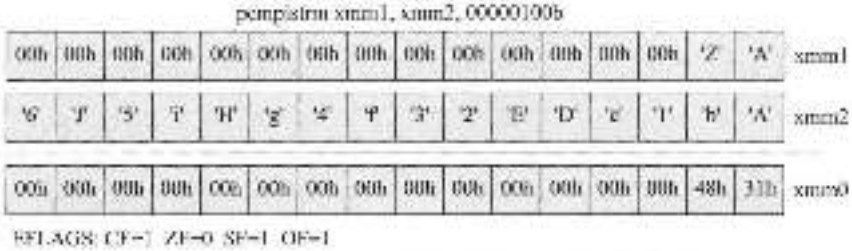


图 11-2 使用 pcmpistrm 指令创建大写字符的位掩码

表 11-3 pcmpistrm xmm1,xmm2,00000100b 的控制功能

位域	值	控制功能
7	0	保留，必须为 0
6	0	IntRes2 被作为字节掩码返回
5	0	未使用，因为 IntRes1 不需取反
4	0	不对 IntRes1 取反
3:2	01	使用区间等价比较与聚合
1:0	00	源数据格式为组合无符号字节

图 11-2 中，XMM1 低位的两个字节包含了所需字符区间的上限和下限，XMM2 包含了字符串片段，注意 XMM2 中的字符串片段使用小端方式存放，这种存放方式是在用 movdqa 或 movdqu 指令加载字符串片段时自动做的。pcmpistrm 指令进行的大部分运算都发生在图 11-1 中被标为 Compare and Aggregation（比较和聚合）的方框中。处理器在这里执行的具体操作视所选的比较和聚合方法的不同而有所区别，这在后续的示例程序中会详细讲述。比较和聚合操作的输出被保存到 IntRes1，其中包含 XMM2 中的大写字符的位掩码。控制值指定 IntRes1 中的中间结果不需要反转，这意味着 IntRes2 中的值与 IntRes1 中相同。同时控制值还指定 16 位的掩码不需要扩展为字节值。最终的掩码值被保存在 XMM0 中最低位的两个字节中，且排列顺序与 XMM2 中的字符串片段的小端排列顺序一致。

图 11-3 显示了 pcmpistrm 指令执行的更多实例。第一个实例与图 11-2 中的实例类似，唯一的差异是输出格式的第 6 位被置为 1，这意味着掩码值被展开为字节值，这对使用简单布尔表达式对字符串片段进行进一步处理非常有用。第二个实例展示了 pcmpistrm 指令用于多个字符区间。在这个例子中，XMM1 包含两个区间对：一个针对大写字符，一个针对小写字符。图 11-3 中的最后一个例子展示了 pcmpistrm 指令用于包含内嵌的 EOS 字符
[307] 的字符串片段。注意最终的掩码值不包含 EOS 字符后面的位于字符区间中的字符。另外，EFLAGS.ZF 被置 1 表示字符串片段中含有 EOS 字符。最后一个 pcmpistrm 实例如图 11-4 所示，它展示了字符串片段中的单个字符匹配，实现方式是在“比较和聚合”控制选项中选择“任意等价”。

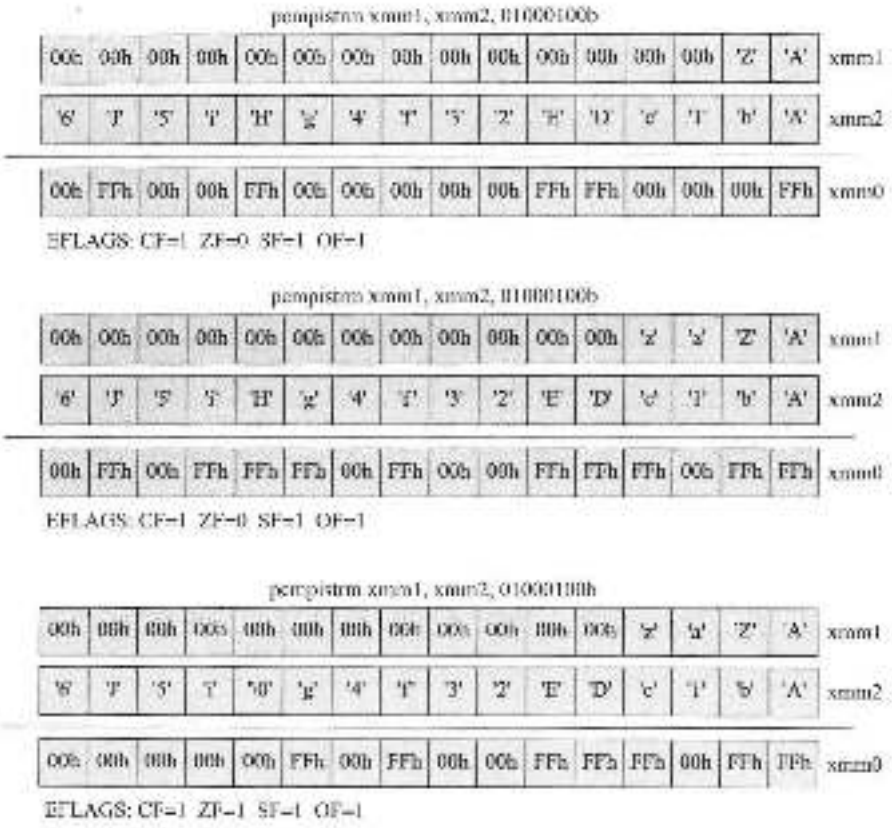


图 11-3 pcmpistrm 指令的执行实例



图 11-4 使用 pcmpistrm 指令来匹配字符

308

可以用 pcmistri 指令来确定一个字符串片段中某个或某几个字符的索引。该指令的一个用途就是在某字符串片段中寻找 EOS 字符的索引，如图 11-5 所示。在第一个实例中，字符串片段不包含 EOS 字符，在这种情况下 pcmpistri 指令的执行会将 EFLAGS.ZF 清零，表示字符串片段不包含 EOS 字符。同时该指令还会在 ECX 中保存已经过检测的字符数量，这个值恰好是一个非法的索引值。在第二个实例里，EFLAGS.ZF 被置位以表示 EOS 字符存在，且寄存器 ECX 包含了与其对应的索引值。在前后两个实例中，控制选项指示的比较和聚合方法为“任意等价”，且 IntRes1 保存的中间结果被倒置。

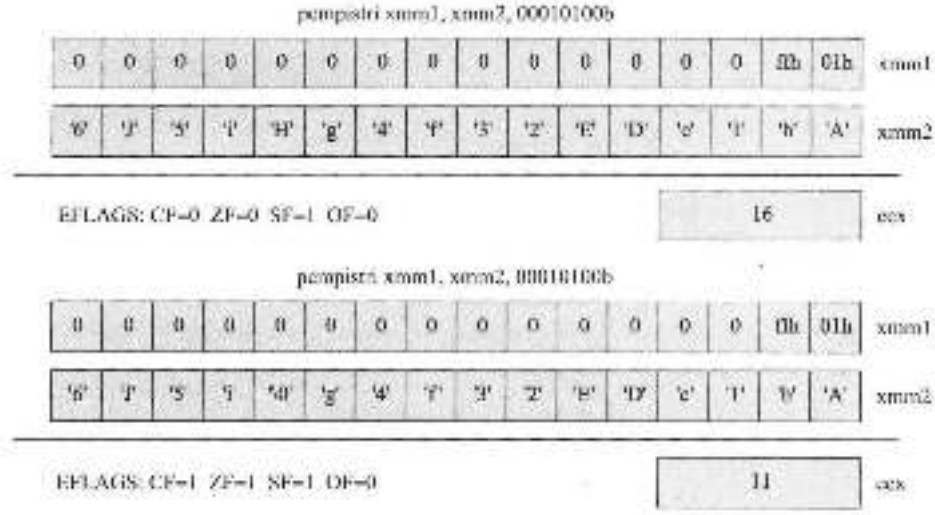


图 11-5 使用 pcmpistri 寻找某个字符串片段中的 EOS 字符

表 11-6 包含了对 pcmpistri 指令执行过程更加详细的解释。在处理器内部，处理器使用比较与聚合类型来选择一个比较与聚合的方法，并使用该方法来比较指定操作数中的所有字符对。在当前的实例中，所选的聚合类型是“区间等价”，表 11-4 给出了具体的比较操作。逻辑上来说，处理器是在构建一个 8×8 的矩阵，该矩阵中存放了字符对的比较结果，处理器使用这个矩阵来计算 IntRes1（对单字节字符生成的是一个 16×16 的矩阵）。EOS 终结符及其后面的字符对比结果被强制设为 0。在构建了比较矩阵之后，处理器使用清单 11-1 中的算法来计算 IntRes1。这个算法随所选的比较与聚合方法的不同而有所变化。IntRes2 的值是通过倒置 IntRes1 中的值得出的，正如控制值中的 Polarity（极性）字段所指定的。EOS 字符

309 的索引值与 IntRes2 中第一个被置为 1 的比特位相对应。

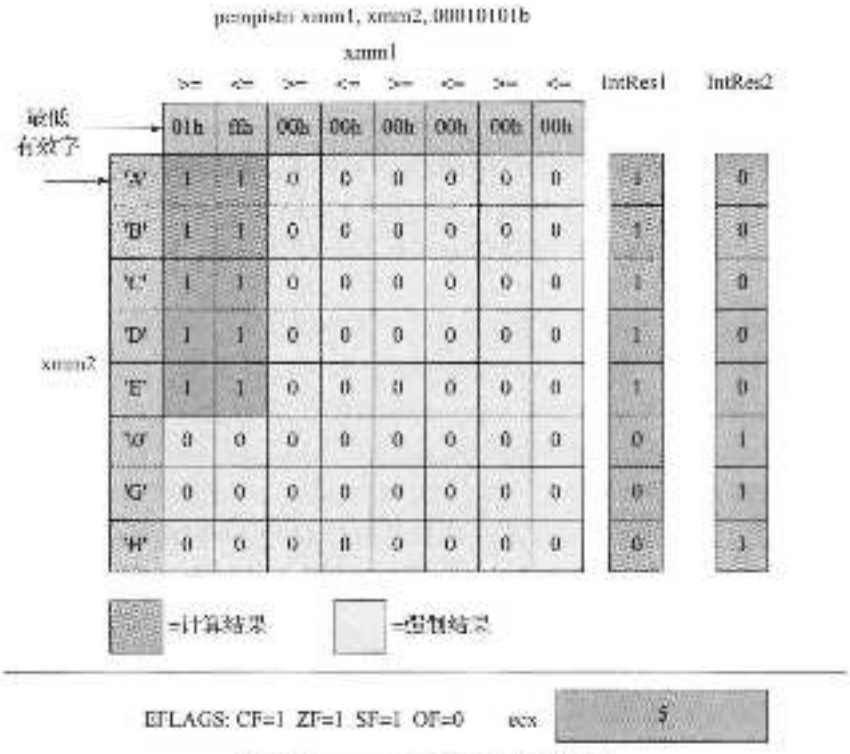


图 11-6 pcmpistri 指令的详细图示

表 11-4 pcmpistri 指令的比较操作真值表

XMM1 索引	比较结果
j 为偶数	CmpRes[i, j] = (xmm2[i] >= xmm1[j]) ? 1 : 0;
j 为奇数	CmpRes[i, j] = (xmm2[i] <= xmm1[j]) ? 1 : 0;

清单 11-1 IntRes1 的计算

```
for (i = 0; i < 8; i++)
{
    for (j = 0; j < 8; j += 2)
        IntRes[i] |= CmpRes[i, j] & CmpRes[i, j+1];
}
```

310

为了在汇编语言函数中使用 x86-SSE 字符串指令，我们必须学习若干个编程中的注意事项。与多字节整数和浮点数不一样，字符串在内存中没有天然的对齐边界。这意味着字符串片段通常是使用 movdqu 指令来加载到 XMM 寄存器中的，movdqa 指令只有在字符串片段进行过适当的对齐操作以后才可以使用。使用 SIMD 技术处理字符串需要程序员确保 EOS 字符之后的任何字符都不会被无意更改。同时还需要注意，在内存中读写接近于某个页面末尾的字符串时很可能需要处理器来访问下一个页面，如图 11-7 所示。在这种情况下，如果接下来的页面不属于当前进程，处理器异常就会被触发。下一节中的代码实例展示了若干种技术用来防止这个问题。



图 11-7 位于内存页面边界处的字符串加载

11.2 字符串编程

上节中我们集中讨论了 x86-SSE 字符串指令的关键内容。在这一节中，我们将学习如何使用 pcmpistri 和 pcmpistrm 指令对字符串做常规操作。我们还将学到一些使用 SIMD 技术处理字符串必须采用的编程策略。

11.2.1 计算字符串长度

我们将要学习的第一个 x86-SSE 字符串示例程序名叫 SseTextStringCalcLength。这个程序展示了如何使用 pcmpistri 指令来计算一个以 null 结尾的字符串的长度。同时该程序还将展示如何应付 SIMD 文本处理中的一些常见问题，这些问题我们已经在本节早些时候讨论过。清单 11-2 和清单 11-3 中分别给出了这个程序的 C++ 和汇编语言源代码。 [311]

清单 11-2 SseTextStringCalcLength.cpp

```
#include "stdafx.h"
#include <malloc.h>
#include <string.h>

extern "C" int SseTextStringCalcLength_(const char* s);

const char * TestStrings[] =
{
        "0123456",                                  // length = 7
        "0123456789abcde",                          // length = 15
        "0123456789abcdef",                         // length = 16
        "0123456789abcdefg",                        // length = 17
        "0123456789abcdefghijklmnopqrstu",          // length = 31
        "0123456789abcdefghijklmnopqrstuv",         // length = 32
        "0123456789abcdefghijklmnopqrstuvw",        // length = 33
        "0123456789abcdefghijklmnopqrstuvwxyz",     // length = 36
        "",                                         // length = 0
};

const int OffsetMin = 4096 - 40;
const int OffsetMax = 4096 + 40;
const int NumTestStrings = sizeof(TestStrings) / sizeof(char*);

void SseTextStringCalcLength(void)
{
    const int buff_size = 8192;
    const int page_size = 4096;
    char* buff = (char*)_aligned_malloc(buff_size, page_size);
```

```
    printf("\nResults for SseTextStringCalcLength()\n");

    for (int i = 0; i < NumTestStrings; i++)
    {
        bool error = false;
        const char* ts = TestStrings[i];

        printf("Test string: \"%s\"\n", ts);

        for (int offset = OffsetMin; offset <= OffsetMax; offset++)
        {
            char* s2 = buff + offset;

            memset(buff, 0x55, buff_size);
[312]       strcpy_s(s2, buff_size - offset, ts);

            int len1 = strlen(s2);
            int len2 = SseTextStringCalcLength_(s2);

            if ((len1 != len2) && !error)
            {
                error = true;
                printf(" String length compare failed!\n");
                printf(" buff: 0x%p  offset: %3d  s2: 0x%p", buff, offset, s2);
                printf(" len1: %3d  len2: %3d\n",len1, len2);
            }
        }

        if (!error)
            printf("No errors detected\n");
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    SseTextStringCalcLength();
    return 0;
}
```

清单 11-3 SseTextStringCalcLength_.asm

```
        .model flat,c
        .code

; extern "C" int SseTextStringCalcLength_(const char* s);
;
; 描述：下面的函数使用 x86-SSE 指令 pcmpistri 计算一个字符串的长度
;
; 返回值：字符串长度
;
; 需要 SSE4.2 支持

SseTextStringCalcLength_ proc
        push ebp
        mov ebp,esp

; 为字符串长度计算初始化寄存器
        mov eax,[ebp+8]                 ;eax = 's'
        sub eax,16                      ; 为循环中的使用调整 eax
        mov edx,0ffffh
[313]   movd xmm1,edx                   ;xmm1[15:0] = 字符区间
```

```
; 计算下一个地址并检测是否已接近于页面尾部
@@:     add eax,16              ;eax = 下一个文本块
        mov edx,eax
        and edx,0fffh           ;edx = 地址的低 12 位
        cmp edx,0ff0h
        ja NearEndOfPage        ;若距离页面边界已小于 16 字节则跳转

; 检测当前文本块中的 '\0' 字节
        pcmpistri xmm1,[eax],14h        ;字符区间与文本相比较
        jnz @B                          ;如果未找到 '\0' 字节则跳转

; 在当前文本块中找到 '\0' 字节（索引位于 ECX 中）
; 计算并返回字符串长度
        add eax,ecx                     ;eax = '\0' 字节的指针
        sub eax,[ebp+8]                 ;eax = 最终字符串长度
        pop ebp
        ret

; 通过检查每个字符来寻找 '\0' 终结符
NearEndOfPage:
        mov ecx,4096                    ;ecx = 页面的大小，单位为字节
        sub ecx,edx                     ;ecx = 要检查的字节数

@@:     mov dl,[eax]                    ;dl = 下一个字符串中的字符
        or dl,dl
        jz FoundNull                    ;若找到 '\0' 则跳转
        inc eax                         ;eax = 下一个字符的指针
        dec ecx
        jnz @B                          ;若还有字符需要检查则跳转

; 剩下的字符串可以使用 16 字节块来查找
; EAX 现已对齐于 16 字节边界
        sub eax,16                      ;为循环中的使用调整 eax
@@:     add eax,16                      ;eax = 下一个文本块的指针
        pcmpistri xmm1,[eax],14h        ;字符区间与文本相比较
        jnz @B                          ;如果未找到 '\0' 字节则跳转

; 在当前文本块中找到 '\0' 字节（索引位于 ECX 中）
        add eax,ecx                     ;eax = '\0' 字节的指针

; 计算并返回字符串长度
FoundNull:
        sub eax,[ebp+8]                 ;eax = 最终字符串长度
        pop ebp
        ret
SseTextStringCalcLength_ endp
        end
```

314

SseTextStringCalcLength 程序的 C++ 部分（清单 11-2）一开始就分配了一块页面边界对齐的内存。这个内存块用来初始化不同的测试场景，以便验证 x86-32 汇编语言函数 SseTextStringCalcLength_ 可以正确处理位于内存页面尾部的字符串。这些场景中包括 EOS 终结符位于一个内存页面的最后一个字节的情形。不可否认，程序中的测试代码有些暴力，但它可以验证完全位于某一页面内或者跨页面的字符串都能被正确处理。SseTextStringCalcLength_ 函数返回的每个字符串长度都会与 C++ 运行时函数 strlen 的计算结果相比较。如果长度不一致，测试代码就会显示错误信息。

汇编语言函数 SseTextStringCalcLength_（见清单 11-3）首先将字符串指针 s 加载到寄存器 EAX 中。sub eax, 16 这条指令将 EAX 中的指针值做了调整以使在循环处理中能够

使用。pcmpistri 指令所需的区间值被加载到寄存器 XMM2 中。在第一个循环处理的最开始，add eax, 16 指令将 EAX 寄存器的值更新，使其指向内存中下一个 16 字节字符块。在循环开始的地方使用常数来更新 EAX 的值可以省去跳转指令，同时也可以防止循环执行依赖的情形出现。这种情况会极大地影响性能。（在一个循环中重复执行某一条指令，且该指令的执行依赖于上一次迭代的结果时，循环执行依赖的情形就会发生。要了解更多关于循环执行依赖的信息，请参考 Intel 64 和 IA-32 架构优化参考手册。）接下来，程序检测 EAX 中的地址以确认下一个字符串片段是否位于内存页面的尾部。由于当前我们尚不知道一个页面是否属于当前进程，程序会执行一个条件跳转指令以避免访问到跨页面的字符串片段。

如果一个字符串片段没有位于内存页面的尾部，程序就使用 pcmpistri 指令来确定 EOS 字符是否位于字符串片段内。如果 EOS 字符被找到（EFLAGS.ZF 被置 1），最终的字符串长度会被算出并返回给调用者。否则，循环会继续。pcmpistri 指令的控制选项被设为无符号单字节、“区间等价”以及 IntRes1 反向。

标识符 NearEndOfPage 后面的代码对每个靠近内存页面尾部的字符单独进行测试，以确认其中是否有 EOS 字符。如果 EOS 字符被找到，最终的字符串长度会被算出来并返回给调用者。如果 EOS 字符在当前页面中没有找到，那么该字符串就是跨页面的，且使用 pcmpistri 指令继续寻找 EOS 字符就是安全的。注意这里检查是否位于页面边界已无必要，因为 EAX 中的指针值已经是 16 字节边界对齐的了。还要注意的是，这时可以使用 movdqa 指令了，因为字符串指针已经做了适当的对齐。输出 11-1 给出了示例程序
315 SseTextStringCalcLength 的执行结果。

输出 11-1 示例程序 SseTextStringCalcLength

```
Results for SseTextStringCalcLength()
Test string: "0123456"
  No errors detected
Test string: "01234567 89abcde"
  No errors detected
Test string: "01234567 89abcdef"
  No errors detected
Test string: "01234567 89abcdefg"
  No errors detected
Test string: "01234567 89abcdefghijklmnopqrstu"
  No errors detected
Test string: "01234567 89abcdefghijklmnopqrstuv"
  No errors detected
Test string: "01234567 89abcdefghijklmnopqrstuvw"
  No errors detected
Test string: "01234567 89abcdefghijklmnopqrstuvwxyz"
  No errors detected
Test string: ""
  No errors detected
```

11.2.2 字符替换

我们将要学习的第二个 x86-SSE 字符串示例程序名为 SseTextStringReplaceChar。这个程序对字符串进行扫描并将指定的字符替换掉。清单 11-4 和清单 11-5 分别给出了 SseTextStringReplaceChar.cpp 和 SseTextStringReplaceChar_.asm 的源代码。

清单 11-4 SseTextStringReplaceChar.cpp

```
#include "stdafx.h"
#include <string.h>
#include <malloc.h>

extern "C" int SseTextStringReplaceChar_(char* s, char old_char, char new_char);

const char* TestStrings[] =
{
    "*Red*Green*Blue*",
    "Cyan*Magenta Yellow*Black Tan",
    "White*Pink Brown Purple*Gray Orange*",
    "Beige Silver Indigo Fuchsia Maroon",
    "****************",
    "*****,*****,*****,*****,*****",
    ""
};

const char OldChar = '*';
const char NewChar = '@';
const int OffsetMin = 4096 - 40;
const int OffsetMax = 4096 + 40;
const int NumTestStrings = sizeof(TestStrings) / sizeof (char*);
const unsigned int CheckNum = 0x03345678;

int SseTextStringReplaceCharCpp(char* s, char old_char, char new_char)
{
    char c;
    int r = 0;

    while ((c = *s) != '\0')
    {
        if (c == OldChar)
        {
            *s = NewChar;
            r++;
        }

        s++;

    }

    return r;
}

void SseTextStringReplaceChar(void)
{
    const int buff_size = 8192;
    const int page_size = 4096;
    char* buff1 = (char*)_aligned_malloc(buff_size, page_size);
    char* buff2 = (char*)_aligned_malloc(buff_size, page_size);

    printf("\nResults for SseTextStringReplaceChars()\n");
    printf("OldChar = '%c' NewChar = '%c'\n", OldChar, NewChar);

    for (int i = 0; i < NumTestStrings; i++)
    {
        const char* s = TestStrings[i];
        int s_len = strlen(s);

        for (int offset = OffsetMin; offset <= OffsetMax; offset++)
        {
            bool print = (offset == OffsetMin) ? true : false;
```

316

```
            char* s1 = buff1 + offset;
            char* s2 = buff2 + offset;
            int size = buff_size - offset;
            int n1 = -1, n2 = -1;

            strcpy_s(s1, size, s);
            *(s1 + s_len + 1) = OldChar;
            *((unsigned int*)(s1 + s_len + 2)) = CheckNum;

            strcpy_s(s2, size, s);
            *(s2 + s_len + 1) = OldChar;
            *((unsigned int*)(s2 + s_len + 2)) = CheckNum;

            if (print)
                printf("\ns1 before replace: \"%s\"\n", s1);
            n1 = SseTextStringReplaceCharCpp(s1, OldChar, NewChar);
            if (print)
                printf("s1 after replace: \"%s\"\n", s1);

            if (print)
                printf("\ns2 before replace: \"%s\"\n", s2);
            n2 = SseTextStringReplaceChar (s2, OldChar, NewChar);
            if (print)
                printf("s2 after replace: \"%s\"\n", s2);

            if (strcmp(s1, s1) != 0)
                printf("Error - string compare failed\n");
            if (n1 != n2)
                printf("Error - character count compare failed\n");

            if (*(s1 + s_len + 1) != OldChar)
                printf("Error - buff1 OldChar overwrite\n");
            if (*(s2 + s_len + 1) != OldChar)
                printf("Error - buff2 OldChar overwrite\n");

            if (*((unsigned int*)(s1 + s_len + 2)) != CheckNum)
                printf("Error - buff1 CheckNum overwrite\n");
            if (*((unsigned int*)(s2 + s_len + 2)) != CheckNum)
                printf("Error - buff2 CheckNum overwrite\n");
        }
    }

    _aligned_free(buff1);
    _aligned_free(buff2);
}

int _tmain(int argc, _TCHAR* argv[])
{
    SseTextStringReplaceChar();
    return 0;
}
```

清单 11-5 SseTextStringReplaceChar_asm

```
        .model flat,c
        .const
        align 16
PxorNotMask db 16 dup(0ffh)                    ;pxor 求反掩码
        .code

; extern "C" int SseTextStringReplaceChar (char* s, char old_char, char new_char);
;
```

```
; 描述：下面的函数将给定的字符串中的所有 old_char 替换成 new_char
;
; 需要 SSE4.2 和 POPCNT 指令支持

SseTextStringReplaceChar_ proc
        push ebp
        mov ebp,esp
        push ebx
        push esi
        push edi

; Initialize
        mov eax,[ebp+8]                 ;eax = 's'
        sub eax,16                      ;为后面的循环调整 EAX 的值
        xor edi,edi                     ;edi = 被替换的字符个数

; 建立组合的 old_char 和 new_char
        movzx ecx,byte ptr [ebp+12]
        movd xmm1,ecx                   ;xmm1[7:0] = old_char
        movzx ecx,byte ptr [ebp+16]     ;ecx = new_char
        movd xmm6,ecx
        pxor xmm5,xmm5
        pshufb xmm6,xmm5                ;xmm6 = 组合的 new_char
        movdqa xmm7,xmmword ptr [PxorNotMask]   ;xmm7 = pxor 逻辑非掩码

; 计算下一个字符串地址并检查是否已经接近页面尾部
Loop1:  add eax,16              ;eax = 下一个文本块
        mov edx,eax
        and edx,0fffh           ;edx = 地址的低 12 位
        cmp edx,0ff0h
        ja NearEndOfPage        ;若距离页面尾部小于 16 字节则跳转                    [319]

; 通过比较当前文本块来查找字符
        movdqu xmm2,[eax]               ;函数下一个文本块
        pcmpistrm xmm1,xmm2,40h         ;检测是否与 old_char 匹配
        setz cl                         ;若找到 '\0' 则置位
        jc FoundMatch1                  ;若发现匹配则跳转
        jz Done                         ;若找到 '\0' 则跳转
        jmp Loop1                       ;若未发现匹配则跳转

; 发现匹配字符 [xmm0 = 匹配掩码]
; 更新位于 edi 中的字符匹配计数
FoundMatch1:
        pmovmskb edx,xmm0               ;edx = 匹配掩码
        popcnt edx,edx                  ;计算匹配的次数
        add edi,edx                     ;edi = 总匹配数

; 将所有 old_char 替换成 new_char
        movdqa xmm4,xmm0                ;xmm0 = 匹配掩码
        pxor xmm0,xmm7
        pand xmm0,xmm2                  ;移除 old_char
        pand xmm3,xmm6
        por xmm0,xmm5                   ;插入 new_char
        movdqu [eax],xmm0               ;保存更新过的字符串
        or cl,cl                        ;当前块中是否包含 '\0'?
        jnz Done                        ;若包含则跳转
        jmp Loop1                       ;继续处理字符串

; 在页面末端将 old_char 替换为 new_char
NearEndOfPage:
        mov ecx,4096                    ;页面大小，单位为字节
        sub ecx,edx                     ;ecx = 要检查的字节数
```

```
        mov dl,[ebp+12]                 ;dl = old_char
        mov dh,[ebp+16]                 ;dh = new_char

Loop2:  mov bl,[eax]                    ;加载下一个输入的字符
        or bl,bl
        jz Done                         ;若发现'\0'则跳转
        cmp dl,bl
        jne @F                          ;若不匹配则跳转
        mov [eax],dh                    ;将old_char替换成new_char
        inc edi                         ;更新替换过的字符数量
@@:     inc eax                         ;eax = 下一个字符的地址
        dec ecx
        jnz Loop2                       ;循环直到页面尾部
        sub eax,16                      ;调整eax的值以避免跳转

; 处理剩下的字符串。注意：此时可以使用movdqa
Loop3:  add eax,16                      ;eax = 下一个文本块
        movdqa xmm2,[eax]               ;加载下一个文本块
        pcmpistrm xmm1,xmm2,20h         ;检测是否与old_char匹配
        setz cl                         ;若找到'\0'则置位
        jc FoundMatch3                  ;若发现匹配则跳转
        jz Done                         ;若发现'\0'则跳转
        jmp Loop3                       ;若未发现匹配则跳转

FoundMatch3:
        pmovmskb edx,xmm0               ;edx = 匹配掩码
        popcnt edx,edx                  ;计算匹配的次数
        add edi,edx                     ;edi = 总匹配次数

; 将所有old_char替换为new_char
        movdqa xmm4,xmm0                ;xmm3 = 匹配掩码
        pxor xmm0,xmm7
        pand xmm0,xmm2                  ;清除所有old_char数据
        pand xmm3,xmm6
        por xmm0,xmm3                   ;插入new_char
        movdqa [eax],xmm0               ;保存更新过的字符串
        or cl,cl                        ;当前块中是否包含'\0'?
        jnz Done                        ;若包含则跳转
        jmp Loop3                       ;继续处理下一个字符

Done:   mov eax,edi                     ;eax = 被替换字符个数
        pop edi
        pop esi
        pop ebx
        pop ebp
        ret
SseTextStringReplaceChar_ endp
        end
```

源文件SseTextStringReplaceChar.cpp（见清单11-4）的开始部分是一个名为SseText-StringReplaceCharCpp的函数，该函数实现了C++版本的字符替换算法。为了验证两个字符替换函数的正确性，函数SseTextStringReplaceChar初始化了若干个测试用例。由于汇编版本字符替换算法会使用SIMD技术来更新字符串，用来替换的字符和一个检验数会被写入EOS字符后面的内存缓冲区中。这个签名可用来检查EOS字符后面的字节是否被错误地改写。与上一个示例程序一样，函数SseTextStringReplaceChar也会将各个测试字符串复制到内存缓冲区中多个不同的位置，以便验证位于内存页面末尾和跨页面的字符串都能得到正确的处理。

在汇编语言函数 SseTextStringReplaceChar（见清单 11-5）的开始部分，它将 EAX 初始化，指向字符串。寄存器 EDI 被设为 0，并用来记录被替换掉的字符的个数。参数 old_char 和 new_char 被分别加载到寄存器 XMM1 和 XMM6 中。注意 old_char 只占用了 XMM1 的最低位字节（XMM1 的其余字节为 0），而 new_char 则驻留于 XMM6 的每个字节当中。pshufb xmm6,xmm5 指令（XMM5 包含全零）创建了一个 new_char 的组合版本（也就是说，XMM5 的每个字节中的值都等于 new_char），这个值会被用于替换操作之中。XMM7 中加载了一个字符求反的掩码值。

在 Loop1 一开始，程序就检测 EAX 中的字符串指针以确定一个字符串片段是否跨越了页面边界。如果当前字符串片段不位于一个页面的尾部，就使用 movdqu 指令将其加载到 XMM2 中。pcmpistrm xmm1,xmm2,40h 指令用于检测当前的字符串片段中是否含有 old_char 中的字符。该指令的控制值被指定为无符号组合字节、“任意等价”以及字节掩码。如果发现了匹配的字符，那么 EFLAGS.CF 就会被置位。如果当前字符串片段中发现了 EOS 字符，那么 EFLAGS.ZF 就会被置位。值得注意的是，这两种情形并不是互斥的，这也是为什么 setz cl 指令会被用来保存 EFLAGS.ZF 的状态以备后续使用。在 pcmpistrm 指令执行过后，XMM0 中包含了匹配字符的掩码（0x00 表示不匹配，0xff 表示匹配）。

标识符 FoundMatch1 后面的代码片段会更新匹配字符的计数并进行组合字符替换。指令 pmovmskb edx,xmm0（移动字节掩码）使用 XMM0 中的每一个字节的最高位创建一个掩码，然后将这个掩码保存到 EDX 中的最低位双字节中（高位双字节以 0 填充）。指令 popcntedx,edx（返回被置为 1 的位的个数）对 EDX 中被置位的位进行计数，这个计数就等于匹配到的字符的个数。这个计数后来被累加到 EDI 寄存器中，EDI 寄存器保存了匹配字符的总个数。图 11-8 展示了将 old_char 中的字符替换为 new_char 中的字符的技术。在组合字符替换操作过后，程序的控制权被转移到 Loop1 的开始处，若当前字符串中包含 EOS 终结符，则控制权被转移到函数结语处。

322



图 11-8　组合字符替换技术的图示

标识符 NearEndOfPage 标识了一个代码段的开始，该段代码对位于页面尾部的字符进行替换。每个字符会被单独检测以确认其是否与 old_char 匹配，若匹配则被替换为 new_char。这段代码还会检测每个字符是否为 EOS 终结符。如果找到 EOS 终结符，循环替换的操作就会结束。

在处理完了位于页面尾部的字符以后，函数就可以继续使用 pcmpistrm 指令来进行 SIMD 字符匹配了。由于此时寄存器 EAX 已经位于 16 字节边界，程序可以使用 movdqa 指令来将剩下的字符串片段加载到 XMM2 中。标识符 Loop3 后面的循环处理采用了相同的办法来替换匹配的字符，如图 11-8 所示。输出 11-2 给出了示例程序 SseTextStringReplaceChar
323 的执行结果。

输出 11-2 示例程序 SseTextStringReplaceChar

```
Results for SseTextStringReplaceChars()
OldChar = '*'  NewChar = '#'

s1 before replace: "*Red*Green*Blue*"
s1 after replace:  "#Red#Green#Blue#"

s2 before replace: "*Red*Green*Blue*"
s2 after replace:  "#Red#Green#Blue#"

s1 before replace: "Cyan*Magenta Yellow*Black Tan"
s1 after replace:  "Cyan#Magenta Yellow#Black Tan"

s2 before replace: "Cyan*Magenta Yellow*Black Tan"
s2 after replace:  "Cyan#Magenta Yellow#Black Tan"

s1 before replace: "White*Pink Brown Purple*Gray Orange*"
s1 after replace:  "White#Pink Brown Purple#Gray Orange#"

s2 before replace: "White*Pink Brown Purple*Gray Orange*"
s2 after replace:  "White#Pink Brown Purple#Gray Orange#"

s1 before replace: "Beige Silver Indigo Fuchsia Maroon"
s1 after replace:  "Beige Silver Indigo Fuchsia Maroon"

s2 before replace: "Beige Silver Indigo Fuchsia Maroon"
s2 after replace:  "Beige Silver Indigo Fuchsia Maroon"

s1 before replace: "****************"
s1 after replace:  "################"

s2 before replace: "****************"
s2 after replace:  "################"

s1 before replace: "*****,*****,*****,*****,*****"
s1 after replace:  "#####,#####,#####,#####,#####"

s2 before replace: "*****,*****,*****,*****,*****"
s2 after replace:  "#####,#####,#####,#####,#####"

s1 before replace: ""
s1 after replace:  ""

s2 before replace: ""
s2 after replace:  ""
```

324

11.3 总结

本章介绍了如何使用 x86-SSE 的字符串处理指令来进行基本的操作，同时还介绍了使用 SIMD 技术处理字符串时需要注意的事项。本章开始我们就提到了 x86-SSE 字符串指令是非常强大和灵活的，但使用起来会感觉有一些迷惑。希望这些迷惑已经消除了，至少是在某种程度上减少了。

此前的四章我们考察了大量的 x86-SSE 示例代码，这么多的示例程序（也许太多了）以及大量的讲解展现了 x86-SSE 的重要性和优势。本书的标题中包含了“现代”一词，之所以选这个词就是为了鼓励大家在任何可能的情况下都使用当前处理器的扩展技术，诸如 x86-SSE，而不是拘泥于老的指令和架构。在接下来的章节中，我们将通过学习最新的 x86 平台 SIMD 扩展技术来使大家的现代汇编语言编程知识面得到进一步的拓宽，这种新的 SIMD 扩展叫作高级向量扩展技术。

325
~
326

第 12 章
Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX

AVX——高级向量扩展

通过前面七章，我们学习了 MMX 和 x86-SSE 的 SIMD（单指令多数据流）处理能力。MMX 技术为 x86 平台引入了初等的整数 SIMD 算术以及相关运算的能力。x86-SSE 技术则增强了这些能力，包括增大操作数宽度、增加额外的寄存器以及对标量和组合操作数做高级浮点算术运算等。本章我们将介绍 x86 平台上最近的增强 SIMD——高级向量扩展（x86-AVX）。

与先前的扩展类似，x86-AVX 增加了新的寄存器、新的数据类型和新的指令。与此同时，x86-AVX 还引入了现代三目运算符汇编语言指令以简化汇编语言编程，并提高性能。在介绍 x86-AVX 过程中，我们也将介绍 x86-AVX 的几个独特扩展功能，包括单精度浮点数变换、Fused-Multiply-Add（FMA，融合乘加）运算以及新的通用寄存器指令。

学习本章的内容需要读者能基本理解 x86-SSE 及其指令集。与第 7 章中介绍 x86-SSE 类似，本章主要讨论如何在 x86-32 执行环境中使用 x86-AVX 指令。在后面的第 19 章和第 20 章中，我们将学习如何在 x86-64 平台上使用 x86-AVX 的计算资源。

12.1 x86-AVX 概述

第一代的 x86-AVX 扩展是在 2011 年由 Sandy Bridge 微架构引入的。AVX 将 x86-SSE 的组合单单精度浮点和双精度浮点能力从 128 位扩展到 256 位。同时它利用无损源操作数（non-destructive source operand）支持了新的三目运算符指令语法，大大地简化了汇编语言编程。编程人员可以利用这一新的指令语法来操作 128 位组合整数、128 位组合浮点数和 256 位组合浮点数。此外这一新的指令语法也可以用来做标量单精度或双精度浮点算术运算。在 2012 年，Intel 发布了 Sandy Bridge 微架构的更新版——Ivy Bridge，引入了新的指令来变换
327 半精度浮点数。关于半精度浮点数，我们将在本章的后面介绍。

2013 年，Intel 又发布了新的微架构——Haswell，并在 Haswell 系列处理器中引入了 AVX2。AVX2 将组合整型数宽度从 128 位扩展到了 256 位。同时它还包含了增强的数据广播、混合和排列指令，引入了新的向量索引寻址模式，加速了从不连续地址空间载入（或者收集）数据的能力。所有 Haswell 系列处理器都包含了几种 AVX2 相关的技术，包括 FMA、增强的位操作以及不带标志位的循环和位移指令。

2013 年 7 月，Intel 宣布了 AVX-512 会在将来的处理器中把 AVX 和 AVX2 的 SIMD 能力从 256 位扩展到 512 位。表 12-1 列出了当前已经支持的和计划支持的 x86-AVX 技术。表 12-1 用 SPFP（Singe-Precision Floating-Point，单精度浮点）和 DPFP（Double-Precision Floating-Point，双精度浮点）表示单精度浮点数和双精度浮点数。

表 12-1 x86-AVX 技术一览

版本	支持的数据类型	关键特性和增强
AVX	组合 128 位整数	新支持的数据类型做 SIMD 运算和三目运算符指令语法
	组合 128 位 SPFP	有条件地加载和存储组合整数
	组合 128 位 DPFP	广播和排列组合数据

（续）

版本	支持的数据类型	关键特性和增强
AVX	组合 256 位 SPFP 组合 256 位 DPFP 标量 SPFP、DPFP	新的浮点运算指令 半精度浮点数支持
AVX2	组合 256 位整数	对组合 256 位整数做 SIMD 运算 数据收集指令 增强的广播和置换指令 FMA 指令 增强的位操作指令 Atom 上的循环和位移指令
AVX-512	组合 512 位整数 组合 512 位 SPFP 组合 512 位 DPFP	对组合 512 位操作数做 SIMD 运算 根据掩码对组合整数元素做运算 指令级舍入覆盖（instruction-level rounding override） 数据散列指令（data scatter instruction）

Sandy Bridge 微架构被用在第二代 Intel Core 处理器中（i3、i5 和 i7 系列），第三代和第四代处理器则分别采用了 Ivy Bridge 和 Haswell 微架构。面向工作站和服务器的 Xeon E3、E3 v2 和 E3 v3 处理器家族分别采用了 Sandy Bridge、Ivy Bridge 和 Haswell 微架构。更多关于 Intel 处理器家族和它们所对应的微架构的信息可参阅附录 C 中列出的 Intel 产品信息网站。 328

12.2 x86-AVX 执行环境

在下面的几节中我们将阐述 x86-AVX 的执行环境，包括它的寄存器组和它所支持的数据类型。我们也将阐述 x86-AVX 中新引入的三目运算符汇编语言指令语法。理解本章内容需要读者熟悉第 7 章到第 11 章的 x86-SSE 内容。

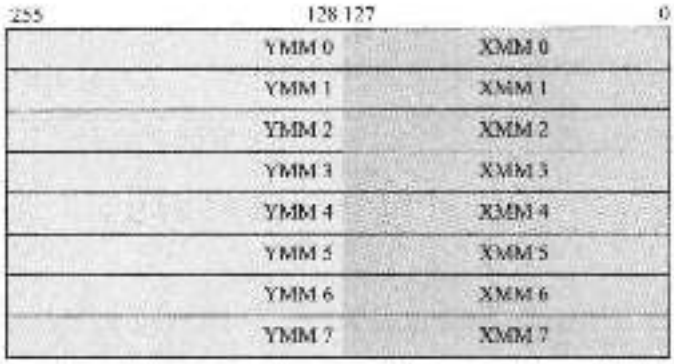
12.2.1 x86-AVX 寄存器组

x86-AVX 为 x86 平台增加了 8 个新的 256 位寄存器——YMM0 ~ YMM7。这些可用来直接寻址的寄存器能够用来操作不同的数据类型，包括组合整数、组合浮点数以及标量浮点数。每个 YMM 寄存器的低 128 位有对应的 XMM 寄存器别名，如图 12-1 所示。x86-AVX 指令既可以用 XMM 寄存器作为操作数也可以用 YMM 寄存器作为操作数别名。如果在执行过程中 x86-AVX 指令以 XMM 寄存器作为目标操作数，处理器会在执行过程中将相应的 YMM 寄存器的高 128 位设置为 0。在那些支持 x86-AVX 指令的处理器上执行 x86-SSE 指令时，YMM 寄存器的高 128 位是永远不会被改动的。关于在执行过程中如何默认地处理 YMM 寄存器的高 128 位，我们将在后续的章节中讨论。

12.2.2 x86-AVX 数据类型

AVX 支持在 SIMD 操作中使用 256 位和 128 位的组合型单精度或双精度浮点操作数。如图 12-2 所示，一个 256 位的 YMM 寄存器或者内存单元可以存储 8 个单精度浮点数或者 4 个双精度浮点数。当使用 128 位宽的 XMM 寄存器或者内存单元时，AVX 指令可以同时处理 4 个单精度浮点数或者两个双精度浮点数。与 SSE 和 SSE2 类似，AVX 用 XMM 寄存器的低位双字（译者注：1 字 =2 字节，1 字节即 1byte）来进行标量单精度浮点数的算术运算，

用低位的四字来进行标量双精度浮点数的算术运算。



329 图 12-1 x86-32 模式下的 x86-AVX 寄存器组

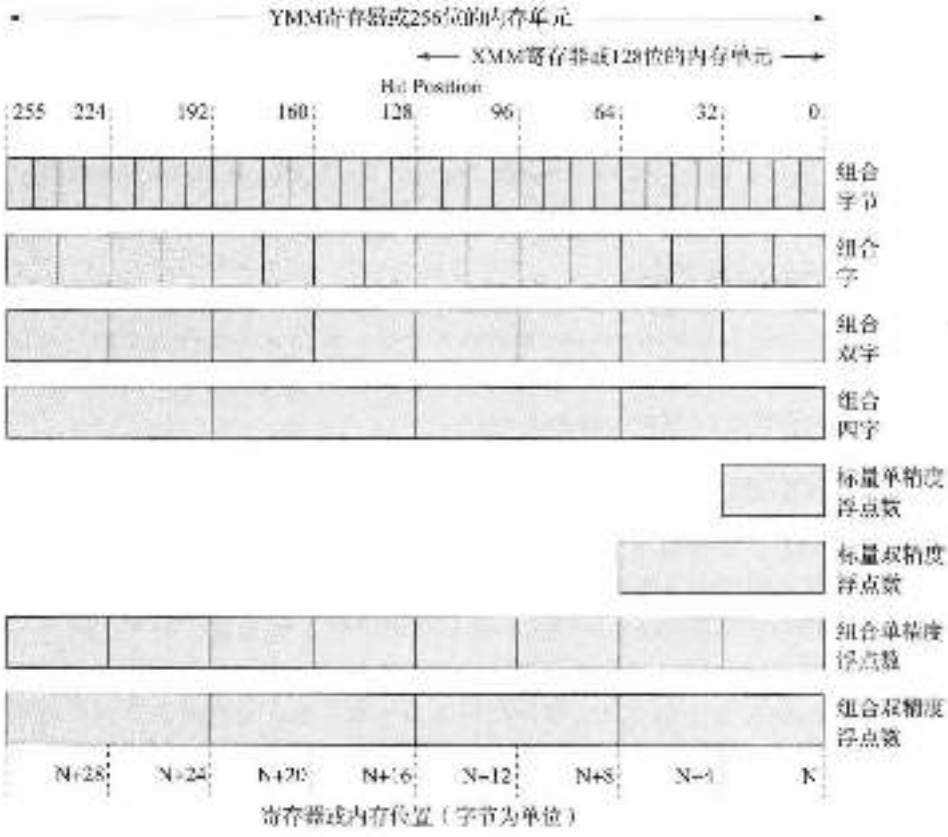


图 12-2 x86-AVX 数据类型

AVX 在执行 SIMD 操作时，也可用 XMM 寄存器来操作不同的组合整型操作数，包括字节（byte）、字（word）、双字（doubleword）和四字（quadword）。AVX2 则通过使用 YMM 寄存器和 256 位宽度的内存单元扩展了处理这些组合型操作数的能力。图 12-2 展示了这些数据类型。

12.2.3 x86-AVX 指令语法

也许在 x86-AVX 中最值得大书特书的是其与时俱进的汇编语言指令的语法。大多数

x86-AVX 指令是三目运算符格式，由两个源操作数和一个目标操作数组成。通用的表达格式为：InstrMnemonic DesOp, SreOp1, SreOp2，其中 InstrMnemonic 表示 x86-AVX 指令助记符，DesOp 表示目标操作数，SreOp1 和 SreOp2 表示源操作数。剩下的那些 x86-AVX 指令要么是只有一个源操作数，要么是有三个源操作数。几乎所有 x86-AVX 指令的源操作数都是无损的（non-destructive）（在指令执行过程中，这些操作数都不会被改动），不过目标操作数和源操作数使用同一寄存器时除外。 330

表 12-2 通过一些例子演示了 x86-AVX 指令的一般语法格式。需要说明的是，本章中所有以字母 v 开头的指令助记符大都是对应的 x86-SSE 指令的扩展。如果读者想查看到底是对哪个 SSE 指令的扩展，只需要将 v 去掉即可。

表 12-2 x86-AVX 指令语法示例

指　令	操　作
vaddpd ymm0,ymm1,ymm2（组合双精度浮点数加法）	ymm0[63:0] = ymm1[63:0] + ymm2[63:0] ymm0[127:64] = ymm1[127:64] + ymm2[127:64] ymm0[191:128] = ymm1[191:128] + ymm2[191:128] ymm0[255:192] = ymm1[255:192] + ymm2[255:192]
vmulps xmm0,xmm1,xmm2（组合单精度浮点数乘法）	xmm0[31:0] = xmm1[31:0] * xmm2[31:0] xmm0[63:31] = xmm1[63:31] * xmm2[63:31] xmm0[95:64] = xmm1[95:64] * xmm2[95:64] xmm0[127:96] = xmm1[127:96] * xmm2[127:96] ymm0[255:128] = 0
vunpcklps xmm0,xmm1,xmm2（以低字（low-order）方式解组单精度浮点数）	xmm0[31:0] = xmm1[31:0] xmm0[63:31] = xmm2[31:0] xmm0[95:64] = xmm1[63:32] xmm0[127:96] = xmm2[63:32] ymm0[255:128] = 0
vpxor ymm0,ymm1,ymm2（逻辑异或）	ymm0[255:0] = ymm1[255:0] ^ ymm2[255:0]
vmovdqa ymm0,ymm1（移动对齐的双四字）	ymm0[255:0] = ymm1[255:0]
vblendpd ymm0,ymm1,ymm2,05h（混合组合双精度浮点数）	ymm0[63:0] = ymm1[63:0] ymm0[127:64] = ymm2[127:64] ymm0[191:128] = ymm2[191:128] ymm0[255:192] = ymm1[255:192]

x86-AVX 之所以能支持三目运算符指令语法，是因为使用了新的指令编码前缀。向量扩展（Vector EXtension-VEX）前缀使得 x86-AVX 指令能够编码成比 x86-SSE 指令更高效的格式，同时它也为将来对 x86-AVX 指令增强奠定了基础。大部分新的通用寄存器指令也使用 VEX 作为前缀。 331

12.3 x86-AVX 功能扩展

在引入 x86-AVX 和 AVX2 的同时，也借机会对 x86 平台做了一些扩展，包括半精度浮点数变换、FMA 计算以及增强的通用寄存器指令。本节主要描述这些扩展功能以及开发者在使用这些扩展功能时的注意事项。

基于 Ivy Bridge 和 Haswell 微架构的处理器包含了可以处理半精度浮点数变换的指令。与标准的单精度相比，半精度浮点数的精度变低了，它包含三个部分：指数部分（5 位）、有效数据部分（10 位）以及符号位。每个半精度浮点数都是 16 位的，其中第一位是符号位。

那些向后兼容的处理器可以将半精度浮点数转换为单精度浮点数，反之亦然。然而，我们无法对半精度浮点数进行通用的算术运算（譬如加、减、乘、除）。半精度浮点数主要是用来降低存储空间的，既包括内存空间也包括数据存储设备的存储空间。使用半精度浮点数所带来的主要缺点是降低了数据精度以及数据的表示范围。

基于Haswell的处理器还包含了可以执行FMA运算的指令。FMA指令将乘法指令和加法/减法指令合并到一条指令中。特别地，一个融合乘加（或者融合乘减）运算在一次操作中完成乘法运算和加法（或减法）运算，只做一次舍入操作。举个例子，假如我们需要计算a = (b * c) − d这个表达式的值。如果利用标准的浮点数算术模式，处理器会先做乘法运算和舍入操作，然后再做加法运算和另一次舍入操作；但是如果采用FMA来计算这个表达式的值，处理器不会对(b * c)的值做舍入，只有当处理器得到最终的表达式(b * c) + d结果时，才会做唯一的一次舍入操作。可以用FMA指令来提高乘法累加运算（譬如点积运算和矩阵乘法运算）的性能和精度。许多信号处理算法也可以采用FMA运算。FMA指令集同时支持标量型和组合型单精度浮点数和双精度浮点数。

x86-AVX的最后一个扩展功能是增加了新的通用寄存器指令。基于Haswell的处理器使用了这些新增加的指令。这些指令增强了位操作，并且支持无标记的寄存器循环和位移操作以及无标记的无符号整数乘法运算。其中无标记的循环和位移操作以及乘法运算不会改变EFLAGS寄存器中的任何标记位的状态。这样可以提高很多整数运算的效率和算法的性能。大多数新的通用寄存器指令采用新型的三目运算符汇编语言语法。

前面所讨论的扩展功能可以认为是因处理器而异的。从编程的角度看，这意味着开发者不能假设处理器支持AVX/AVX2指令抑或不支持。比如说，将来用在手持设备上的处理器可能支持AVX2但却不支持FMA以满足特殊的设计需求。因此开发者每次都应该显式地通过CPUID指令来检测当前处理器是否支持某些特定的扩展功能。在第16章中，我们将通过
332 示例代码来演示如何检测。

12.4 x86-AVX指令集概述

可以把x86-AVX指令集大体分为三类。第一类是三目运算符的x86-SSE指令升级版本。第二类是那些AVX/AVX2新引入的指令。最后一类则是那些功能扩展指令，包括半精度浮点数变换、FMA指令和新的通用寄存器指令。

在开始介绍之前，需要说明一下x86-AVX指令集的一些语法和执行指令时的一些共同事项。正如本章前面所提到的那样，所有x86-AVX指令的汇编语法都包含一个指令助记符、一个目标操作数以及最多三个源操作数。如果一个指令执行数据传输操作，那么目标操作数需指定内存地址，目标操作数只能为XMM或者YMM寄存器。此外，源操作数中只能有一个可以指定为内存地址，其他的只能是XMM寄存器、YMM寄存器或者立即数。

x86-AVX放宽了指令操作数内存对齐的要求。除了数据传输指令需要显式地要求16字节或者32字节对齐外，其他的x86-AVX指令的操作数不需要严格对齐。尽管如此，我们仍然强烈建议指令操作数以16字节或32字节对齐以达到最好的性能。此外，在支持x86-AVX指令的处理器上运行x86-SSE指令时仍然需要内存对齐。

12.4.1 升级版的x86-SSE指令

大部分操作128位操作数（包括组合单精度浮点数、组合双精度浮点数和组合整数）的

x86-SSE 指令都有与之对应的 x86-AVX 指令。例如，如果我们需要计算 XMM0 和 XMM1 的乘积，并将结果保存到 XMM0，其中 XMM0 和 XMM1 为组合型单精度浮点数，则可以利用 x86-SSE 指令“mulps xmm0,xmm1”来实现。如果用与之相对应的 x86-AVX 指令来实现，则是“vmulps xmm0,xmm0,xmm1”。另外一个例子是 x86-SSE 中组合型整数的加法指令“paddb xmm0,xmm1”，与之相对应的 x86-AVX 指令为“vpaddb xmm0,xmm0,xmm1”。需要指出的是，上面两个例子都是有损操作，因为它们改变了 XMM0 的值。而 vmulps xmm0,xmm1,xmm2 和 vpaddb xmm0,xmm1,xmm2 则是无损的，因为它们不会改变 XMM1 和 XMM2 的值。

几乎所有 128 位的 x86-SSE 指令都有相应的 x86-AVX 指令，用来操作 256 位的操作数。譬如 vsubpd ymm7,ymm0,ymm1 指令在进行组合型浮点数加法运算时，用了四对双精度的值；vdivps ymm7,ymm0,ymm1 在进行组合单精度浮点数除法运算时用了八对值；而 vpsubb ymm7,ymm0,ymm1 则用了 32 对整型字节的值。

在处理器内，每个 256 位的 x86-AVX 寄存器都可以分为高 128 位和低 128 位两部分。大多数 x86-AVX 指令在执行时，目标操作数和源操作数都使用相同的部分分别做运算，其中目标操作数的低 128 位与源操作数的低 128 位做运算，高 128 位则与源操作数的高 128 位做运算。不过在利用 x86-AVX 指令做算术运算时，这种运算方式并不明显。 333

然而当利用 x86-AVX 指令重新排列组合型数据时（例如 vshufps 和 vpunpcklwd），这种分部运算的方式就变得非常明显，如图 12-3 所示。例子中，将会对操作数的高位双四字（128 – 255 位）以及低位双四字（0 – 128 位）分别独立地执行重排（shuffle）操作或者解组合操作。

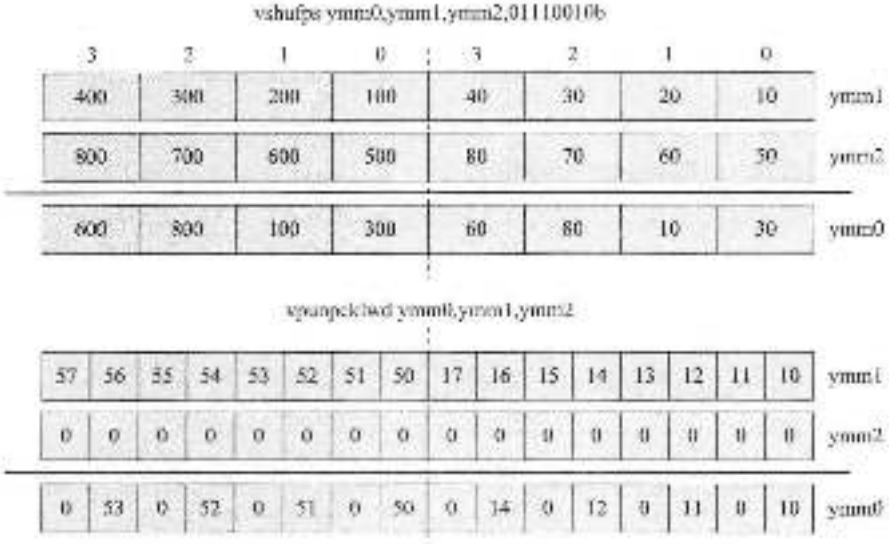


图 12-3 x86-AVX 指令独立执行示例

x86-AVX 指令集也以三目运算等的形式支持 x86-SSE 的标量浮点指令。例如 vaddss xmm0,xmm1,xmm2 指令将存储在 xmm1 和 xmm2 中的标量单精度浮点数相加，并将结果保存在 xmm0 中；vmulsd xmm0,xmm1,xmm2 指令将存储在 xmm1 和 xmm2 中的标量双精度浮点数相乘，并将结果保存在 xmm0 中。

在 Intel 和 AMD 的用户手册中，罗列了所有 x86-AVX 中升级版的 x86-SSE 指令，读者

可以从附录 C 提供的网站链接中下载。

x86-SSE 和 x86-AVX 指令之间的高度对称性，以及 XMM 和 YMM 寄存器间的别名关系，导致了开发者需要牢记一些注意事项。首先是以 XMM 寄存器作为目标操作数时，处理器对相应的 YMM 寄存器高 128 位的处理。当 x86-SSE 指令以 XMM 寄存器作为目标操作
334 数时，永远不应访问相应的 YMM 寄存器高 128 位。但是与该 SSE 指令相对应的 x86-AVX 指令则会将 YMM 寄存器的高 128 位置为 0。例如，请思考下面 (v)cvtps2pd（将组合型单精度浮点数转换为组合型双精度浮点数）的实例：

```
cvtps2pd xmm0,xmm1
vcvtps2pd xmm0,xmm1
vcvtps2pd ymm0,ymm1
```

cvtps2pd 指令把存储在 XMM1 低位（low-order）四字的两个组合型单精度浮点数转换为双精度浮点数，并将结果保存在 XMM0 中，而 YMM0 的高 128 位则没有任何改变。第一个 vcvtps2pd 实例同样也是将组合单精度浮点数转换为双精度浮点数，但它同时将 YMM0 的高 128 位设置为 0。第二个 vcvtps2pd 实例将存储在 YMM1 低 128 位中的四个组合型单精度浮点数转换为双精度浮点数，并将结果保存在 YMM0 中。

所有 x86-AVX 标量浮点指令都将 YMM 寄存器的高 128 位设置为 0。这些指令同时也将第一个源操作数的那些不用的位拷贝到目标操作数中，如表 12-3 中的 vaddss 和 vaddsd（标量单精度 / 双精度浮点数相加）指令所示。表 12-3 同时展示了 vsqrtss 和 vsqrtsd（计算单精度 / 双精度浮点数的标量平方根）指令的执行方式。需要说明的是这些指令都需要两个源操作数，即使是只用了第二个源操作数的一元操作。

表 12-3 x86-AVX 标量浮点指令示例

指　令	操　作
vaddss xmm0,xmm1,xmm2（标量单精度浮点数加法）	xmm0[31:0] = xmm1[31:0] + xmm2[31:0] xmm0[127:32] = xmm1[127:32] ymm0[255:128] = 0
vaddsd xmm0,xmm1,xmm2（标量双精度浮点数加法）	xmm0[63:0] = xmm1[63:0] + xmm2[63:0] xmm0[127:64] = xmm1[127:64] ymm0[255:128] = 0
vsqrtss xmm0,xmm1,xmm2（单精度浮点数平方根计算）	xmm0[31:0] = sqrt(xmm2[31:0]) xmm0[127:32] = xmm1[127:32] ymm0[255:128] = 0
vsqrtsd xmm0,xmm1,xmm2（双精度浮点数平方根计算）	xmm0[63:0] = sqrt(xmm2[63:0]) xmm0[127:64] = xmm1[127:64] ymm0[255:128] = 0

335

最后一个注意事项是 x86-AVX 和 x86-SSE 的混合编程。可以使用 x86-AVX 与 x86-SSE 进行混合编程，但是应尽量避免处理器内部状态频繁切换而影响性能。在 x86-AVX 切换到 x86-SSE 的过程中，如果要求保留每个 YMM 寄存器的高 128 位，就可能影响性能。不过我们可以使用 vzeroupper（YMM 寄存器高位置 0）指令将所有 YMM 寄存器的高 128 位置 0，从而完全避免这个问题。当需要从 256 位的 x86-AVX 代码（例如，任意的 x86-AVX 指令使用了 YMM 寄存器）切换到 x86-SSE 代码时，应该首先使用 vzeroupper 指令。

vzeroupper 指令常用在使用了 256 位 x86-AVX 指令的公共函数中。这些类型的函数应该在 ret 指令之前调用 vzeroupper 指令，从而降低 x86-SSE 代码调用此类函数时处理器进行状态切换所带来的性能损失。在调用那些有可能包含 x86-SSE 代码的库函数之前，也应该使用 vzeroupper 指令。在第 14 章到第 16 章中包含了合理使用 vzeroupper 指令的实例。此外函数也可以使用 vzeroall 指令将所有的 YMM 寄存器初始化为 0，以降低 x86-AVX /x86-SSE 间状态切换所带来的不利影响。

12.4.2 新指令

本小节我们简要地浏览器的 x86-AVX 指令。这些指令被分成了以下几类：

- 广播指令（Broadcast）
- 混合指令（Blend）
- 排列指令（Permute）
- 提取和插入指令（Extract and Insert）
- 掩码移动指令（Masked Move）
- 变长移位指令（Variable Bit Shift）
- 数据收集指令（Gather）

下面的指令集列表概要地罗列了各类指令。需要说明的是，如果表中的版本列把 AVX 和 AVX2 都列出来，则表示该指令在 AVX2 中新加了其他形式。

1. 广播指令

那些可以将单一数据拷贝（或广播）至组合数据的多个子元素的指令，被归类为广播指令。广播指令对所有的组合数据均适用，包括单精度浮点数、双精度浮点数和整数。表 12-4 概述了广播指令。 336

表 12-4 x86-AVX 广播指令

助记符	描 述	版本
vbroadcastss	将单精度浮点数拷贝至目标操作数的所有子元素中	AVX AVX2
vbroadcastsd	将双精度浮点数拷贝至目标操作数的所有子元素中	AVX AVX2
vbroadcastf128	从内存中拷贝组合 128 位浮点数至目标操作数的低位双四字和高位双四字中	AVX
vbroadcasti128	从内存中拷贝组合 128 位整数至目标操作数的低位双四字和高位双四字中	AVX2
vpbroadcastb vpbroadcastw vpbroadcastd vpbroadcastq	拷贝一个 8 位/16 位/32 位/64 位整数至目标操作数的所有子元素中	AVX2

2. 混合指令

那些可以将两种组合数据有条件地合并在一起的指令，被归类为混合指令。这些指令如表 12-5 所示。

表 12-5 x86-AVX 混合指令

助记符	描 述	版本
vpblendd	根据立即数所指定的控制标记，有条件地将前两个源操作数的双字拷贝至目标操作数中	AVX2

3. 排列指令

那些可以重排或复制组合型数据子元素的指令，被归类为排列指令。排列指令支持多种
337 组合型数据，包括双字、单精度浮点数和双精度浮点数。表 12-6 概述了这些指令。

表 12-6 x86-AVX 排列指令

助记符	描 述	版本
vpermd	根据第一个源操作数所指定的索引，以双字为单位来排列第二个源操作数。此指令可以对第二个源操作数中的双字进行重排或者复制	AVX2
vpermpd	根据立即数所指定的索引，以双精度浮点数为单位来排列第一个源操作数。此指令可以对源操作数中的双精度浮点数进行重排或者复制	AVX2
vpermps	根据第一个源操作数所指定的索引，以单精度浮点数为单位来排列第二个源操作数。此指令可以对第二个源操作数中的单精度浮点数进行重排或者复制	AVX2
vpermq	根据立即数所指定的索引，以四字为单位来排列单一个源操作数。此指令可以对源操作数中的四字进行重排或者复制	AVX2
vperm2i128	根据立即数所指定的索引，以组合型 128 位整数为单位来排列前两个源操作数。此指令可以对前两个源操作数中的组合型 128 位整数进行重排、复制或者交叉（interleave）	AVX2
vpermilpd	根据第二个源操作数所指定的控制值，排列第一个源操作数中的双精度浮点数。每 128 位独立调换	AVX
vpermilps	根据第二个源操作数所指定的控制值，排列第一个源操作数中的单精度浮点数。每 128 位独立调换	AVX
vperm2f128	根据立即数掩码所指定的索引，排列前两个源操作数中的组合型 128 位浮点数。此指令可以对前两个源操作数中的组合 128 位浮点数进行重排、复制或者交叉（interleave）	AVX

4. 提取和插入指令

在 YMM 寄存器和 XMM 寄存器或者内存之间，拷贝组合型 128 位整数的指令，归类为
338 提取和插入指令。表 12-7 概要地说明了这些指令。

表 12-7 x86-AVX 提取和插入指令

助记符	描 述	版本
vextracti128	根据立即数指定的值，提取源操作数的高位或者低位组合 128 位整数，并将其拷贝至目标操作数中	AVX2
vinsert128	将第二个源操作数中的组合型 128 位整数插入目标操作数中。插入的位置（低 128 位或高 128 位）由一个立即数指定。目标操作数中保留的部分（高 128 位或低 128 位）用第一个操作数与之对应的部分（高 128 位或低 128 位）填充	AVX2

5. 掩码移动指令

掩码移动指令是指有条件地移动组合数据的指令。控制掩码决定了是否将一个指定的元素从源操作数拷贝至目标操作数中。如果源操作数中的元素没有被拷贝，那么目标操作数中所对应的元素被设置为 0。表 12-8 列出了掩码移动指令。

表 12-8 x86-AVX 掩码移动指令

助记符	描 述	版本
vmaskmovps	根据第一个源操作数所指定的控制掩码，有条件地将第二个源操作数中的单精度浮点数元素拷贝到目标操作数对应的位置	AVX
vmaskmovpd	根据第一个源操作数所指定的控制掩码，有条件地将第二个源操作数中的双精度浮点数元素拷贝到目标操作数对应的位置	AVX

（续）

助记符	描　述	版本
vpcmpeqd vpcmpeqq	根据第一个源操作数所指定的比较条件，与条件比较第二个源操作数中的双字元素将复制到目标操作数对应的位置	AVX2
vpcmpgtd vpcmpgtq	根据第一个源操作数所指定的比较条件，与条件比较第二个源操作数中的四字元素将复制到目标操作数对应的位置	AVX2

339

6. 变长移位指令

对组合双字或组合四字数据中的元素进行不同位数的算术移位或逻辑移位的指令，归类为变长移位指令。表 12-9 概述了这些指令。

表 12-9　x86-AVX 变长移位指令

助记符	描　述	版本
vpsllvd vpsllvq	根据第二个源操作数所指定的对应移位计数，对第一个源操作数中的每个双字/四字元素进行左移并以 0 填充	AVX2
vpsravd	根据第二个源操作数所指定的对应移位计数，对第一个源操作数中的每个双字元素进行右移并以元素的符号位填充	AVX2
vpsrlvd vpsrlvq	根据第二个源操作数所指定的对应移位计数，对第一个源操作数中的每个双字/四字元素进行右移并以 0 填充	AVX2

7. 数据收集指令

数据收集指令包含那些将数据元素从内存拷贝至 XMM 寄存器或者 YMM 寄存器的指令。这些指令采用特殊的内存寻址方式，叫作 VSIB（vector scale-index-base）寻址。VSIB 内存寻址方式包含下面几个元素：

- *Base*——一个通用寄存器，保存数组在内存中的首地址。
- *Scale*——数组元素大小的比例因子（1、2、4 或 8）。
- *Index*——一个向量寄存器（XMM 或 YMM），包含有符号双字或者四字数组的下标。
- *Displacement*——可选元素，用来指定距离数据起始位置的偏移。

根据指令需要，向量寄存器必须包含 2、4 或者 8 个有符号整数下标。这些下标用来从数组中选取元素。图 12-4 展示了指令 vgatherdps xmm0,[esi+xmm1*4],xmm2 的执行过程。在这个例子中，寄存器 ESI 指向了单精度浮点数数组的起始位置。XMM1 中保存了四个有符号双字的数组下标，XMM2 中保存了条件掩码控制掩码。

340

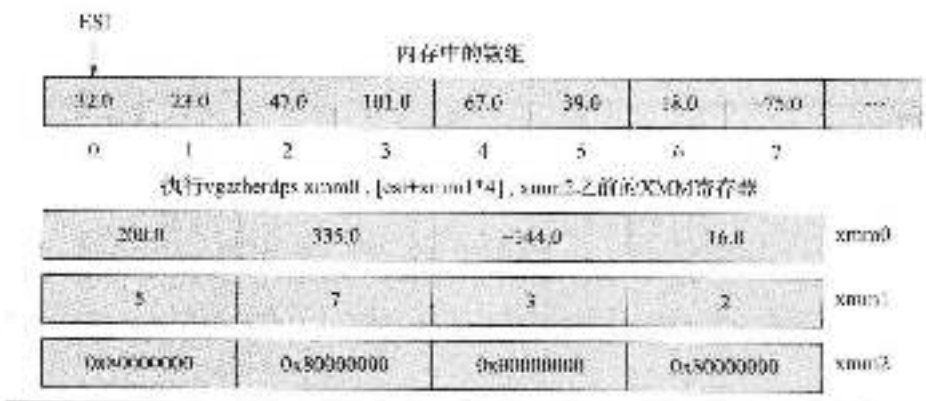


图 12-4　vgatherps 指令图示

执行vgatherdps xmm0, [esi+xmm1*4], xmm2之后的XMM寄存器

39.0	-75.0	-144.0	47.0	xmm0
5	7	3	2	xmm1
0x00000000	0x00000000	0x00000000	0x00000000	xmm2

图 12-4 （续）

数据收集指令的目标操作数和第二个源操作数（拷贝控制掩码）必须是 XMM 或者 YMM 寄存器。第一个源操作数指定了 VSIB 的元素（数组的基址寄存器、放大因子、数组下标和可选的偏移）。需要注意的是，数据收集指令不会检查数组的下标是否有效，引用无效的数据下标将会导致错误的结果。表 12-10 概述了数据收集指令。表中，每个数据收集指令助记符都用 vgatherd 或 vgatherq 作为前缀来分别指定数据的下标是双字还是四字。

341

表 12-10 x86-AVX 数据收集指令

助记符	描述	版本
vgatherdpd vgatherqpd	利用 VSIB 寻址，有条件地拷贝内存数组中的两个或者四个双精度浮点数	AVX2
vgatherdps vgatherqps	利用 VSIB 寻址，有条件地拷贝内存数组中的四个或者八个单精度浮点数	AVX2
vgatherdd vgatherqd	利用 VSIB 寻址，有条件地拷贝内存数组中的四个或者八个双字	AVX2
vgatherdq vgatherqq	利用 VSIB 寻址，有条件地拷贝内存数组中的两个或者四个四字	AVX2

12.4.3 功能扩展指令

本节描述 x86-AVX 的功能扩展指令，包括半精度浮点数指令、FMA 指令以及增强的通用寄存器指令。处理器通过 cpuid 指令来表示是否支持这些指令。半精度浮点数指令和 FMA 指令需要处理器支持 AVX 或 AVX2，并且要求操作系统在进行线程或者进程上下文切换的时候保存 YMM 寄存器状态。

1. 半精度浮点数指令

半精度浮点数指令包括那些将组合半精度浮点数转换为单精度浮点数（反之亦然）的指令。

342

处理器通过 cpuid F16C 功能标识来表示是否支持这些指令。表 12-11 概述了半精度浮点数指令。

表 12-11 x86-AVX 半精度浮点数指令

助记符	描述	版本
vcvtph2ps	将源操作数中的四个或八个半精度浮点数转换为单精度浮点数，并将结果保存到目标操作数中。具体转换的数目由目标操作数的大小决定，此操作数必须为 XMM 或者 YMM 寄存器	AVX
vcvtps2ph	将源操作数中的四个或八个单精度浮点数转换为半精度浮点数，并将结果保存到目标操作数中。具体转换的数目依赖于第一个源操作数的大小，此操作数必须为 XMM 或者 YMM 寄存器。此指令同时需要一个立即数来指定舍入模式	AVX

2. FMA 指令

FMA 指令（融合乘加指令）包括那些对组合浮点数或标量浮点数进行融合乘加 / 融合乘

减运算的指令。FMA 指令的运算表达式如下所示：

```
a = (b * c) + d
a = (b * c) - d
a = -(b * c) + d
a = -(b * c) - d
```

对于上面的每个表达式，处理器仅会对最终运行结果进行一次舍入处理，这样能提高运算的速度和精度。

所有 FMA 指令助记符都包含三个采顺序的操作符标识（如 vfmadd132sd），用来指定执行乘加（减）运算的源操作数。第一个数字（1）指定了作为被乘数的源操作数。第二个数字（3）指定了作为乘数的源操作数，第三个数字（2）指定了加（减）到乘法结果的源操作数。譬如 vfmadd132sd xmm0,xmm1,xmm2 指令（变址型双精度浮点数的融合乘加运算），寄存器 XMM0、XMM1 和 XMM2 分别为源操作数 1、2 和 3。vfmadd132sd 指令计算 (xmm0[63:0] * xmm2[63:0]) + xmm1[63:0]，按 MXCSR.RC 指定的舍入模式对最终结果进行舍入处理，并将舍入处理后的最终结果保存到 xmm0[63:0] 中。

FMA 指令集支持组合型和标量型的单精度浮点数和双精度浮点数。组合型 FMA 指令可以使用 XMM 寄存器或 YMM 寄存器。XMM 寄存器支持使用两个（四个）双精度浮点数或者四个（八个）单精度浮点数进行运算的组合型 FMA 指令。标量型浮点指令则仅支持 [343]
XMM 寄存器。对于所有的 FMA 指令，第一个和第二个源操作数必须是寄存器，第三个源操作数则既可以是寄存器也可以是内存地址。如果一个 FMA 指令使用 XMM 寄存器作为目标操作数，那么与之对应的 YMM 寄存器的高 128 位被设置为 0。如前面所表述的那样，FMA 根据 MXCSR.RC 指定的舍入模式仅执行一次舍入操作。

下面介绍 FMA 指令集。这些指令集被分为 6 个小类以便理解。在描述这些小类的指令列表中，助记符的后两个字符分别表示不同的数据类型，pd 表示组合双精度浮点数；ps 表示组合单精度浮点数；sd 表示标量双精度浮点数；ss 表示标量单精度浮点数。src1、src2 和 src3 分别代表三个源操作数，des 则表示目标操作数，与 src1 相同。

（1）VFMADD 小类

VFMADD 小类包含那些对组合浮点数或标量浮点数进行融合乘加运算的指令。表 12-12 概述了这类指令。

表 12-12 FMA VFMADD 小类指令

助记符	描述
vfmadd132(pd|ps|sd|ss)	des = src1 * src3 + src2
vfmadd213(pd|ps|sd|ss)	des = src2 * src1 + src3
vfmadd231(pd|ps|sd|ss)	des = src2 * src3 + src1

（2）VFMSUB 小类

VFMSUB 小类包含那些对组合型浮点数或标量型浮点数进行融合乘减运算的指令。表 12-13 概述了这类指令。

表 12-13 FMA VFMSUB 小类指令

助记符	描述
vfmsub132(pd|ps|sd|ss)	des = src1 * src3 - src2
vfmsub213(pd|ps|sd|ss)	des = src2 * src1 - src3
vfmsub231(pd|ps|sd|ss)	des = src2 * src3 - src1

[344]

（3）VFMADDSUB 小类

VFMADDSUB 小类包含那些对组合型数据执行乘法运算，并对源操作数的奇元素执行加法运算、偶元素执行减法运算的指令。表 12-14 罗列了这些指令。

表 12-14 FMA VFMADDSUB 小类指令

助记符	描述
vfmaddsub132(pd|ps)	des = src1 * src3 + src2（src2 的奇元素）des = src1 * src3 – src2（src2 的偶元素）
vfmaddsub213(pd|ps)	des = src2 * src1 + src3（src3 的奇元素）des = src2 * src1 – src3（src3 的偶元素）
vfmaddsub231(pd|ps)	des = src2 * src3 + src1（src1 的奇元素）des = src2 * src3 – src1（src1 的偶元素）

（4）VFMSUBADD 小类

VFMSUBADD 小类包含那些对组合型数据执行乘法运算，并对源操作数的奇元素执行减法运算、偶元素执行加法运算的指令。表 12-15 罗列了这些指令。

表 12-15 FMA VFMSUBADD 小类指令

助记符	描述
vfmsubadd132(pd|ps)	des = src1 * src3 + src2（src2 的偶元素）des = src1 * src3 – src2（src2 的奇元素）
vfmsubadd213(pd|ps)	des = src2 * src1 + src3（src3 的偶元素）des = src2 * src1 – src3（src3 的奇元素）
345 Vfmsubadd231(pd|ps)	des = src2 * src3 + src1（src1 的偶元素）des = src2 * src3 – src1（src1 的奇元素）

（5）VFNMADD 小类

VFMNADD 小类包含那些负的融合乘加运算指令。表 12-16 罗列了这些指令。

表 12-16 FMA VFNMADD 小类指令

助记符	描述
vfnmadd132(pd|ps|sd|ss)	des = –(src1 * src3) + src2
vfnmadd213(pd|ps|sd|ss)	des = –(src2 * src1) + src3
vfnmadd231(pd|ps|sd|ss)	des = –(src2 * src3) + src1

（6）VFNMSUB 小类

VFMNSUB 小类包含那些负的融合乘减运算指令。表 12-17 罗列了这些指令。

表 12-17 FMA VFNMSUB 小类指令

助记符	描述
vfnmsub132(pd|ps|sd|ss)	des = –(src1 * src3) – src2
vfnmsub213(pd|ps|sd|ss)	des = –(src2 * src1) – src3
vfnmsub231(pd|ps|sd|ss)	des = –(src2 * src3) – src1

3. 通用寄存器指令

这一组包含了一些新指令，有支持增强位操作的指令，无标记旋转和位移操作指令，以及无标记无符号整数乘法指令。表 12-18 概述了这些指令。可以通过 CPUID 的功能标志位
346 检查当前处理器是否支持这些指令。

表 12-18 通用寄存器指令

助记符	描述	功能标记
andn	将第二个源操作数和反转后的第一个源操作数按位进行逻辑与操作，并将结果保存到目标操作数中。第一个源操作数和目标操作数必须是通用寄存器，第二个源操作数可以为内存地址或者通用寄存器	BMI1

（续）

助记符	描述	功能标记
bextr	根据第二个源操作数所指定的下标和长度，提取第一个源操作数的二进制位，并将其写入目标操作数中。第二个源操作数和目标操作数必须为通用寄存器。第一个源操作数可以为内存地址或通用寄存器	BMI1
blsi	获取源操作数的最低一位，并将设置到目标操作数相应位置。目标操作数其他的所有二进制位则被设置为 0。目标操作数必须为通用寄存器。源操作数可以为内存地址或通用寄存器	BMI1
blsmsk	决定源操作数是否设置值为全真。将此位以及所在之此位低为位均设置为 1。目标操作数中无论两标志的位，均设置为 0。源操作数可以是内存地址或通用寄存器。目标操作数必须为通用寄存器	BMI1
blsr	将源操作数按以三目标操作数，找到源操作数二进制位为 1 的最低位，将目标操作数中对应此最低位的位置置为 0。源操作数可以是为内地址或通用寄存器。目标操作数必须为通用寄存器	BMI1
bzhi	将从第一个源操作数至目标操作数中，根据第二个源操作数所指定的下标，将目标操作数中高于此下标的那为给清 0。目标操作数和第二个源操作数必须为通用寄存器。第一个源操作数可以为内存地址或通用寄存器	BMI2
lzcnt	统计源操作数中前导零的个数，并将结果值保存至目标操作数。如果源操作数为零则为 0，则目标操作数设置为操作数大小。此指令可以作为 bsr 指令的另一个选项：bsr 指令中，如果源操作数的值为 0，则目标操作数为未定义状态	LZCNT
mulx	将寄存器 EDX 与源操作数做无符号乘法运算。运算结果的高位和低位分别保存至第一个和第二个目标操作数中。EFLAGS 的状态位不会更新。源操作数可以为内存地址或通用寄存器。第一个和第二个目标操作数必须为通用寄存器	BMI2
pdep	根据第二个源操作数所指定为掩码，将第一个源操作数低位分散转移到目标操作数中，目标操作数不在掩码范围内的位被置 0。目标操作数和第一个源操作数必须为通用寄存器，第二个源操作数可以为内存地址或通用寄存器	BMI2
pext	按从低到高的顺序，根据第二个源操作数所指定的掩码，将第一个源操作数对应的二进制位依次传输至目标操作数中。目标操作数和第一个源操作数必须为通用寄存器。第二个源操作数可以为内存地址或通用寄存器	BMI2
rdrand	将通过硬件产生的随机数加载到目标操作数中。目标操作数必须为通用寄存器	RDRAND
rorx	根据立即数指定的反转次数，将源操作数进行反转操作。此指令不改变 EFLAGS 的状态位。源操作数可以为内存地址或通用寄存器。目标操作数必须为通用寄存器	BMI2
sarx shlx shrx	根据第二个源操作数所指定的移位位数，对第一个源操作数进行移位操作（算术右移、逻辑左移、逻辑右移），并将结果保存至目标操作数中。不改变 EFLAGS 的状态位。第一个源操作数可以为内存地址或通用寄存器。第二个源操作数和目标操作数必须为通用寄存器	BMI2
tzcnt	统计源操作数中尾部零的个数。如果源操作数的值为 0，则目标操作数设置为源操作数的大小。此指令可以作为 bsf 指令的另一个选项：bsf 指令中，如果源操作数为 0，则目标操作数为未定义状态	BMI1

12.5 总结

在本章中，我们学习了 x86-AVX 的关键内容，包括它的执行环境，支持的数据类型和汇编语言指令语法，同时我们也探索了几种 x86-AVX 相关的扩展功能，包括半精度浮点数转换、FMA 指令和增强的通用寄存器指令。不过这些知识只是 x86-AVX 学习征途的开始，我们将继续在第 13 章到第 16 章中通过一系列示例代码来阐释本章的内容。

x86-AVX 标量浮点编程

前一章中，我们学习了 x86-AVX 的各类计算资源，包括它的执行环境、支持的数据类型以及指令集。本章介绍 x86-AVX 编程，着重介绍它的标量浮点能力，用若干示例程序描述如何执行基本的标量浮点运算。同时，本章也准备了另外几个使用 x86-AVX 的程序，演示更高级的标量浮点编程方法。所有示例程序必须在支持 AVX 的处理器上运行。附录 C 中列出了一些工具，可以用来确定你的 PC 处理器所支持的 x86-AVX 版本以及安装的操作系统。

13.1 编程基础

在本节中，将学习如何使用 x86-AVX 指令集进行基本的标量浮点运算。第一个示例程序演示了基本的标量浮点算术运算，包括加、减、乘、除各类运算。第二个示例程序用来说明标量浮点的比较操作。两个示例程序中使用的大部分 x86-AVX 指令，都较于其相应的 x86-SSE，有简化编程的辅助效果，后面会详细说明。

13.1.1 标量浮点运算

第一个程序名为 AvxScalarFloatingPointArithmetic，演示了使用 x86-AVX 指令集执行基本的标量双精度浮点运算。清单 13-1 和清单 13-2 分别列出了相应的 C++ 和汇编语言源代码。

351

清单 13-1 AvxScalarFloatingPointArithmetic.cpp

```
#include "stdafx.h"

extern "C" void AvxSfpArithmetic_(double a, double b, double results[8]);

int _tmain(int argc, _TCHAR* argv[])
{
    const int n = 8;
    const char* lnames[n] =
    {
        "vaddsd", "vsubsd", "vmulsd", "vdivsd",
        "vminsd", "vmaxsd", "vsqrtsd a", "fabs b"
    };

    double a = 17.75;
    double b = -39.1875;
    double c[n];

    AvxSfpArithmetic_(a, b, c);

    printf("\nResults for AvxScalarFloatingPointArithmetic\n");
    printf("a:              %.6lf\n", a);
    printf("b:              %.6lf\n", b);
    for (int i = 0; i < n; i++)
        printf("%-14s  %-12.6lf\n", lnames[i], c[i]);
```

```
    return 0;
}
```

清单 13-2 AvxScalarFloatingPointArithmetic_.asm

```
        .model flat,c
        .const
AbsMask qword 7fffffffffffffffh, 7fffffffffffffffh
        .code

; extern "C" void AvxSfpArithmetic_(double a, double b, double results[8]);
;
; 描述：本函数演示如何使用基本的标量 DPFP 运算指令
;
; 需要：AVX

AvxSfpArithmetic_ proc
        push ebp
        mov ebp,esp                                                    352

; 加载参数值
        mov eax,[ebp+24]                ;eax = 指向 results 数组
        vmovsd xmm0,real8 ptr [ebp+8]   ;xmm0 = a
        vmovsd xmm1,real8 ptr [ebp+16]  ;xmm1 = b

; 使用 AVX 标量 DPFP 指令执行基本运算
        vaddsd xmm2,xmm0,xmm1           ;xmm2 = a + b
        vsubsd xmm3,xmm0,xmm1           ;xmm3 = a - b
        vmulsd xmm4,xmm0,xmm1           ;xmm4 = a * b
        vdivsd xmm5,xmm0,xmm1           ;xmm5 = a / b
        vmovsd real8 ptr [eax+0],xmm2   ;保存 a + b
        vmovsd real8 ptr [eax+8],xmm3   ;保存 a - b
        vmovsd real8 ptr [eax+16],xmm4  ;保存 a * b
        vmovsd real8 ptr [eax+24],xmm5  ;保存 a / b

; 计算 min(a, b), max(a, b), sqrt(a) 和 fabs(b)
        vminsd xmm2,xmm0,xmm1           ;xmm2 = min(a, b)
        vmaxsd xmm3,xmm0,xmm1           ;xmm3 = max(a, b)
        vsqrtsd xmm4,xmm0,xmm0          ;xmm4 = sqrt(a)
        vandpd xmm5,xmm1,xmmword ptr [AbsMask]  ;xmm5 = fabs(b)
        vmovsd real8 ptr [eax+32],xmm2  ;保存 min(a, b)
        vmovsd real8 ptr [eax+40],xmm3  ;保存 max(a, b)
        vmovsd real8 ptr [eax+48],xmm4  ;保存 sqrt(a)
        vmovsd real8 ptr [eax+56],xmm5  ;保存 trunc(sqrt(a))

        pop ebp
        ret
AvxSfpArithmetic_ endp
        end
```

AvxScalarFloatingPointArithmetic.cpp（见清单 13-1）中的 C++ 代码很清晰，顶部声明了汇编语言函数 AvxSpfArithmetic_，它有三个参数：两个双精度浮点参数和指向 results 数组的指针。函数 _tmain 中的代码执行基本的程序任务，包括初始化测试变量 a 和 b，然后调用函数 AvxSpfArithmetic ，显示执行结果。

用汇编语言编写的函数 AvxSpfArithmetic_ 使用 x86-AVX 指令集来执行几种常见的双精度浮点运算。在函数序言之后，指令 vmovsd xmm0,real8 ptr [ebp+8] 把参数值 a 加载到 XMM0 的低四字部分。x86-AVX 指令 vmovsd 几乎与 x86-SSE 中相应的指令 movsd 等同，

除了一点：它会将 YMM0 寄存器的高 192 位（ymm0[255:64]）清零。下一条指令 vmovsd xmm1,real8 ptr [ebp+16] 执行类似的操作，把参数值 b 加载到寄存器 XMM1。

在参数值 a 和 b 分别加载到寄存器 XMM0 和 XMM1 之后，指令 vaddsd xmm2,
353 xmm0,xmm1 把 XMM0 和 XMM1 中的标量双精度浮点值相加，并把结果存入 XMM2 的低四字部分。与对应的 x86-SSE 指令不同，vaddsd 会执行两个辅助操作：将 XMM0 的高四字拷贝到 XMM2 的高四字（xmm2[127:64] = xmm0[127:64]），同时把 YMM2 的高 128 位（ymm2[255:128]）清零。

接下来的三条指令 vsubsd、vmulsd 和 vdivsd 分别执行标量双精度浮点减法、乘法和除法，这些指令也会对相应的目标操作数执行前面描述的从属操作。执行完四个基本运算指令后，使用一系列的 vmovsd 指令将计算结果保存到 results 数组中。注意，当使用 vmovsd 指令时的目标操作数是内存地址时，只有源操作数的低四字部分会被拷贝到内存，不会发生额外的四字数据传输或清零操作。

下一个代码块描述了指令 vminsd、vmaxsd 和 vsqrtsd 的用法，同时演示了使用 vandpd 指令计算标量双精度浮点数的绝对值。指令 vminsd xmm2,xmm0,xmm1 和 vmaxsd xmm3,xmm0,xmm1 分别计算 min(a,b) 和 max(a,b)。注意 x86-AVX vsqrtsd 指令需要两个源操作数，但是只计算第二个源操作数的平方根。这三条指令会将第一个源操作数的高四字部分拷贝到目标操作数的高四字部分，同时将目标操作数对应的 YMM 寄存器的高 128 位清零。

指令 vandpd xmm5,xmm1,xmmword ptr [AbsMask] 对两个组合双精度浮点数执行按位与运算，并把结果保存到目标操作数中（没有 vandsd 这样的指令）。这条指令的第二个源操作数在 .const 段中定义，它包含了一个用来清除两个 128 位宽组合双精度浮点数符号位的位组合。这里使用的 vandpd 指令也会把 ymm5[255:128] 清零。

读者可能已经注意到函数 AvxSpfArithmetic_ 中没有任何寄存器到寄存器的数据传输指令，这是因为 x86-AVX 的三操作数语法有意如此设计，以简化编程。类似功能采用 x86-SSE 实现，会需要若干额外的寄存器到寄存器或内存到寄存器的 movsd 指令。输出 13-1 显示了 AvxScalarFloatingPointArithmetic 的运行结果。

输出 13-1 示例程序 AvxScalarFloatingPointArithmetic

```
Results for AvxScalarFloatingPointArithmetic
a:              17.750000
b:              -39.187500
vaddsd          -21.437500
vsubsd          56.937500
vmulsd          -695.578125
vdivsd          -0.452951
vminsd          -39.187500
vmaxsd          17.750000
vsqrtsd a       4.213075
fabs b          39.187500
```
354

13.1.2 标量浮点比较

x86-AVX 指令集包含几种不同的指令来执行标量浮点比较。vcomisd 和 vcomiss 指令通过设置 EFLAGS 寄存器的状态位来表示比较结果。回顾第 8 章的示例程序，用了等效的 x86-SSE 指令

comisd 和 comiss 用来进行标量浮点比较，本节的示例程序名叫 AvxScalarFloatingPointCompare，它描述了如何使用 x86-AVX 的 vcmpsd 指令（比较标量双精度浮点数）来比较两个标量双精度浮点数。为示例程序 AvxScalarFloatingPointCompare 所写的 C++ 和汇编语言的源代码分别见清单 13-3 和清单 13-4。

清单 13-3 AvxScalarFloatingPointCompare.cpp

```
#include "stdafx.h"
#include <limits>

using namespace std;

extern "C" void AvxSfpCompare_(double a, double b, bool results[8]);

int _tmain(int argc, _TCHAR* argv[])
{
    const int n = 4;
    const int m = 8;

    const char* inames[8] =
    {
        "vcmpeqsd", "vcmpneqsd", "vcmpltsd", "vcmplesd",
        "vcmpgtsd", "vcmpgesd", "vcmpordsd", "vcmpunordsd"
    };

    double a[n] = { 20.0, 50.0, 75.0, 42.0 };
    double b[n] = { 30.0, 40.0, 75.0, 0.0 };
    bool results[n][m];

    b[3] = numeric_limits<double>::quiet_NaN();

    printf("Results for AvxScalarFloatingPointCompare\n");

    for (int i = 0; i < n; i++)
    {
        AvxSfpCompare_(a[i], b[i], results[i]);

        printf("\na: %8lf b: %8lf\n", a[i], b[i]);
        for (int j = 0; j < m; j++)
            printf("%12s = %d\n", inames[j], results[i][j]);
    }

    return 0;
}
```

355

清单 13-4 AvxScalarFloatingPointCompare_.asm

```
        .model flat,c
        .code

; extern "C" void AvxSfpCompare_(double a, double b, bool results[8]);
;
; 描述：本程序演示如何使用 x86-AVX 比较指令 vcmpsd
;
; 需要：AVX

AvxSfpCompare_ proc
        push ebp
        mov ebp,esp
```

```
; 加载参数值
        vmovsd xmm0,real8 ptr [ebp+8]       ;xmm0 = a
        vmovsd xmm1,real8 ptr [ebp+16]      ;xmm1 = b;
        mov eax,[ebp+24]                    ;eax = 指向 results 数组的指针

; 执行等于比较
        vcmpeqsd xmm2,xmm0,xmm1             ;执行比较操作
        vmovmskpd ecx,xmm2                  ;把比较结果移至 ecx 的 bit 0
        test ecx,1                          ;测试结果位
        setnz byte ptr [eax+0]              ;以 C++ bool 类型保存结果

; 执行不等于比较。注意如果使用了 QNaN 或 SNaN 操作数，vcmpneqsd 返回 true
        vcmpneqsd xmm2,xmm0,xmm1
        vmovmskpd ecx,xmm2
        test ecx,1
        setnz byte ptr [eax+1]

; 执行小于比较
        vcmpltsd xmm2,xmm0,xmm1
        vmovmskpd ecx,xmm2
        test ecx,1
356     setnz byte ptr [eax+2]

; 执行小于等于比较
        vcmplesd xmm2,xmm0,xmm1
        vmovmskpd ecx,xmm2
        test ecx,1
        setnz byte ptr [eax+3]

; 执行大于比较
        vcmpgtsd xmm2,xmm0,xmm1
        vmovmskpd ecx,xmm2
        test ecx,1
        setnz byte ptr [eax+4]

; 执行大于等于比较
        vcmpgesd xmm2,xmm0,xmm1
        vmovmskpd ecx,xmm2
        test ecx,1
        setnz byte ptr [eax+5]

; 执行有序比较
        vcmpordsd xmm2,xmm0,xmm1
        vmovmskpd ecx,xmm2
        test ecx,1
        setnz byte ptr [eax+6]

; 执行无序比较
        vcmpunordsd xmm2,xmm0,xmm1
        vmovmskpd ecx,xmm2
        test ecx,1
        setnz byte ptr [eax+7]

        pop ebp
        ret
AvxSfpCompare_ endp
        end
```

AvxScalarFloatingPointCompare.cpp（见清单13-3）中的函数 _tmain 包含一些基本的C++语句，使用了不同的测试值来调用汇编语言函数 AvxSfpCompare_。需要注意的是，双

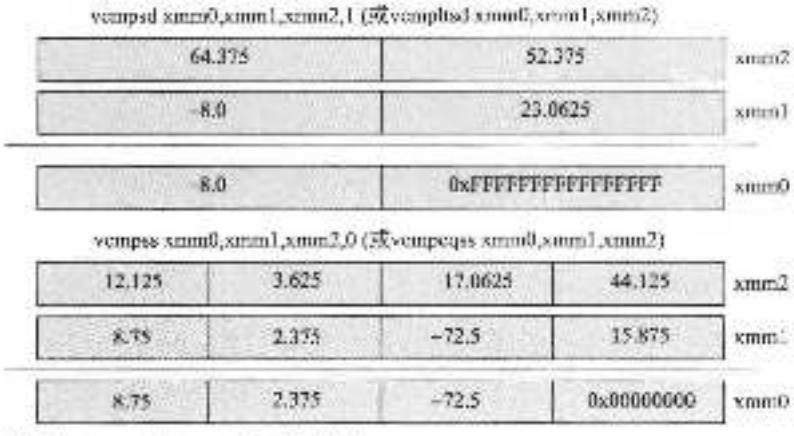
精度浮点数组 b 的最后一个元素被赋值为 QNaN（QNaN 值在第 3 章中介绍过），这样做是为了演示无序的浮点比较操作。对于每对测试值，函数 _tmain 显示了八种不同双精度浮点比较的结果。

在阅读汇编语言源代码前，还需要了解一些关于 vcmpsd 指令的细节。接下来的解释也适用于 vcmpss（比较标量单精度浮点数）指令。这些指令支持两种不同的汇编语言语法格式。第一种格式需要四个操作数：一个目标操作数、两个源操作数和一个指定比较谓词的立即数。包括 MASM 在内的大部分 x86 汇编器，也支持集成了比较谓词的三操作数伪指令格式。 357

vcmpsd 和 vcmpss 指令支持 32 种不同的比较谓词。表 13-1 列出了大多数常用的比较谓词和它们对应的操作。更多关于比较谓词的内容可以参考附录 C 列出的 Intel 和 AMD 指令参考手册。vcmpsd（vcmpss）指令的执行会产生一个四字（双字）掩码结果，保存到目标操作数。掩码可能的值只有两种：全 1（比较结果为真）和全 0（比较结果为假）。图 13-1 描述了 vcmpsd 和 vcmpss 指令的执行情况。第 14 章会介绍与 vcmpsd 和 vcmpss 指令对应的组合版本。

表 13-1 vcmpsd 和 vcmpss 常用的比较谓词

谓词操作值	谓词	描 述	伪指令
0	EQ	Src1 == Src2	vcmpeqs(s|d)
1	LT	Src1 < Src2	vcmplts(s|d)
2	LE	Src1 <= Src2	vcmples(s|d)
3	UNORD	Src1 && Src2 是无序的	vcmpunords(s|d)
4	NEQ	Src1 != Src2	vcmpneqs(s|d)
13	GE	Src1>= Src2	vcmpges(s|d)
14	GT	Src1 > Src2	vcmpgts(s|d)
7	ORD	Src1 && Src2 是有序的	vcmpords(s|d)



注：上述的例子中，ymm0[255:128] 都会被清零

图 13-1 vcmpsd 和 vcmpss 指令的执行示意图 358

在函数 AvxSfpCompare（见清单 13-4）的序言之后，使用两条 vmovsd 指令把参数 a 和 b 分别加载到寄存器 XMM0 和 XMM1。然后初始化指针，指向 results 数组。指令 vcmpeqsd xmm2,xmm0,xmm1 对 a 和 b 进行等于比较，并把产生的组合掩码值存到 XMM2 的低四字部分。这条指令也会执行在上一节中我们了解到的辅助操作。更具体来说，所有

vcmpsd（vcmpss）指令会把源操作数高 64（96）位的值拷贝到目标操作数相应的位置，同时将目标操作数对应的 YMM 寄存器的高 128 位清零。

下一条指令 vmovmskpd ecx,xmm2 将源操作数中每个双精度浮点数的符号位拷贝到目标操作数的低位（目标操作数中没有使用的高位会被清零）。当前这个示例程序中，XMM2 的低四字部分包含 vcmpeqsd 比较操作的结果，全为 0 或者全为 1。XMM2 的高四字部分的值不用在意。根据比较结果，使用 vmovmskpd 指令对寄存器 ECX 清零。后续的指令 test ecx,1 和 setnz byte ptr [eax+0] 把比较结果保存到 results 数组。

除了指定的比较指令，剩余的比较操作都使用同样的指令序列。需要注意的是，不同于 x86-SSE cmpsd 和 cmpss 指令，vcmpsd 和 vcmpss 指令明确支持比较谓词 GT 和 GE。另外需要注意的是，如果使用了 QNaN 或 SNaN 操作数，vcmpneqsd 指令会返回真值（true）。你可能会问如何在 vcmpsd/vcmpss 和 vcomisd/vcomiss 指令间选择。后两条指令用起来比较简单，但是只支持比较少的比较操作。前一组指令除了支持更广泛的比较谓词外，在需要位掩码来执行规定的布尔操作时，也很有用处。输出 13-2 显示了示例程序 AvxScalarFloatingPointCompare 的执行结果。

输出 13-2 示例程序 AvxScalarFloatingPointCompare

```
Results for AvxScalarFloatingPointCompare

a: 20.000000 b: 30.000000
    vcmpeqsd = 0
    vcmpneqsd = 1
    vcmpltsd = 1
    vcmplesd = 1
    vcmpgtsd = 0
    vcmpgesd = 0
    vcmpordsd = 1
    vcmpunordsd = 0

a: 50.000000 b: 40.000000
    vcmpeqsd = 0
    vcmpneqsd = 1
    vcmpltsd = 0
    vcmplesd = 0
    vcmpgtsd = 1
    vcmpgesd = 1
    vcmpordsd = 1
    vcmpunordsd = 0

a: 75.000000 b: 75.000000
    vcmpeqsd = 1
    vcmpneqsd = 0
    vcmpltsd = 0
    vcmplesd = 1
    vcmpgtsd = 0
    vcmpgesd = 1
    vcmpordsd = 1
    vcmpunordsd = 0

a: 42.000000 b: 1.#QNAN0
    vcmpeqsd = 0
    vcmpneqsd = 1
    vcmpltsd = 0
    vcmplesd = 0
    vcmpgtsd = 0
```

```
vcmppsd = 0
vcmpeqsd = 0
vcmpunordsd = 1
```

13.2 高级编程

本节中，将学习如何使用x86-AVX指令集执行高级标量浮点运算。第一个示例程序演示如何计算一元二次方程的根。第二个示例程序利用x86-AVX标量浮点计算能力实现球坐标系的转换。此例中还演示了在使用x86-AVX指令集的汇编语言函数中如何使用标准C一库函数。相对于x86-SSE，x86-AVX具有较好的计算优势，同时程序上的编写也更简单。两个示例程序都反映了这一优势。

13.2.1 一元二次方程的根

接下来的示例程序AvxScalarFloatingPointQuadEqu将演示使用x86-AVX指令集计算一元二次方程的根，同时进一步演示如何使用x86-AVX标量浮点运算指令。清单13.5和清单13.6中分别包含了AvxScalarFloatingPointQuadEqu的C++和x86-AVX汇编语言代码。 360

清单 13-5 AvxScalarFloatingPointQuadEqu.cpp

```
#include "stdafx.h"
#include <math.h>

extern "C" void AvxSfpQuadEqu_(const double coef[3], double roots[2],↩
double epsilon, int* dis);

void AvxSfpQuadEquCpp(const double coef[3], double roots[2], double↩
epsilon, int* dis)
{
    double a = coef[0];
    double b = coef[1];
    double c = coef[2];
    double delta = b * b - 4.0 * a * c;
    double temp = 2.0 * a;

    if (fabs(a) < epsilon)
    {
        *dis = 9999;
        return;
    }

    if (fabs(delta) < epsilon)
    {
        roots[0] = -b / temp;
        roots[1] = -b / temp;
        *dis = 0;
    }
    else if (delta > 0)
    {
        roots[0] = (-b + sqrt(delta)) / temp;
        roots[1] = (-b - sqrt(delta)) / temp;
        *dis = 1;
    }
    else
```

```
    {
        // roots[0] 包含实数部分，roots[1] 包含虚数部分
        // 完整的结果为 (r0, +r1), (r0, -r1)
        roots[0] = -b / temp;
        roots[1] = sqrt(-delta) / temp;
        *dis = -1;
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    const int n = 4;
    const double coef[n * 3] =
    {
        2.0, 8.0, -15.0,        // 不同实数根 (b * b > 4 * a * c)
        1.0, 6.0, 9.0,          // 相同实数根 (b * b = 4 * a * c)
        3.0, 2.0, 4.0,          // 复数根 (b * b < 4 * a * c)
        1.0e-13, 7.0, -5.0,     // 无效的 a 值
    };

    const double epsilon = 1.0e-12;

    printf("Results for AvxScalarFloatingPointQuadEqu\n");

    for (int i = 0; i < n * 3; i += 3)
    {
        double roots1[2], roots2[2];
        const double* coef2 = &coef[i];
        int dis1, dis2;

        AvxSfpQuadEquCpp(coef2, roots1, epsilon, &dis1);
        AvxSfpQuadEqu_(coef2, roots2, epsilon, &dis2);

        printf("\na: %lf, b: %lf c: %lf\n", coef2[0], coef2[1], coef2[2]);

        if (dis1 != dis2)
        {
            printf("Discriminant compare error\n");
            printf("dis1/dis2: %d/%d\n", dis1, dis2);
        }

        switch (dis1)
        {
            case 1:
                printf("Distinct real roots\n");
                printf("C++ roots: %lf %lf\n", roots1[0], roots1[1]);
                printf("AVX roots: %lf %lf\n", roots2[0], roots2[1]);
                break;

            case 0:
                printf("Identical roots\n");
                printf("C++ root: %lf\n", roots1[0]);
                printf("AVX root: %lf\n", roots2[0]);
                break;

            case -1:
                printf("Complex roots\n");
                printf("C++ roots: (%lf %lf) ", roots1[0], roots1[1]);
                printf("(%lf %lf)\n", roots1[0], -roots1[1]);
                printf("AVX roots: (%lf %lf) ", roots2[0], roots2[1]);
                printf("(%lf %lf)\n", roots2[0], -roots2[1]);
```

[361] [362]

```
            break;

        case 9999:
            printf("Coefficient 'a' is invalid\n");
            break;

        default:
            printf("Invalid discriminant value: %d\n", dis1);
            return 1;
        }
    }

    return 0;
}
```

清单 13-6 AvxScalarFloatingPointQuadEqu_.asm

```
        .model flat,c
        .const
FpNegateMask    qword 8000000000000000h,0   ;双精度浮点负值的掩码
FpAbsMask       qword 7FFFFFFFFFFFFFFFh,-1  ;计算 fabs() 的掩码
r8_0p0          real8 0.0
r8_2p0          real8 2.0
r8_4p0          real8 4.0
        .code

; extern "C" void AvxSfpQuadEqu_(const double coef[3], double roots[2],↩
double epsilon, int* dis);
;
; 描述：本例使用公式从求解一元二次方程的根
;
; 需要：AVX

AvxSfpQuadEqu  proc
        push ebp
        mov ebp,esp                                                    363

; 加载参数值
        mov eax,[ebp+8]                 ;eax = 指向 coef 数组
        mov ecx,[ebp+12]                ;ecx = 指向 roots 数组
        mov edx,[ebp+24]                ;edx = 指向 dis
        vmovsd xmm0,real8 ptr [eax]     ;xmm0 = a
        vmovsd xmm1,real8 ptr [eax+8]   ;xmm1 = b
        vmovsd xmm2,real8 ptr [eax+16]  ;xmm2 = c
        vmovsd xmm7,real8 ptr [ebp+16]  ;xmm7 = epsilon

; 确保系数 a 是有效的
        vandpd xmm6,xmm0,[FpAbsMask]    ;xmm2 = fabs(a)
        vcomisd xmm6,xmm7
        jb Error                        ;如果 fabs(a) < epsilon, 跳转

; 计算中间值
        vmulsd xmm3,xmm1,xmm1           ;xmm3 = b * b
        vmulsd xmm4,xmm0,[r8_4p0]       ;xmm4 = 4 * a
        vmulsd xmm4,xmm4,xmm2           ;xmm4 = 4 * a * c
        vsubsd xmm3,xmm4,xmm4           ;xmm3 = b * b - 4 * a * c
        vmulsd xmm0,xmm0,[r8_2p0]       ;xmm0 = 2 * a
        vxorpd xmm1,xmm1,[FpNegateMask] ;xmm1 = -b

; 测试 delta 值，确定根的类型
        vandpd xmm2,xmm3,[FpAbsMask]    ;xmm2 = fabs(delta)
        vcomisd xmm2,xmm7
```

```
        jb IdenticalRoots                   ;如果 fabs(delta) < epsilon, 跳转
        vcomisd xmm3,[r8_0p0]
        jb ComplexRoots                     ;如果 delta < 0.0, 跳转

; 相异实数根
; r1 = (-b + sqrt(delta)) / 2 * a, r2 = (-b - sqrt(delta)) / 2 * a
        vsqrtsd xmm3,xmm3,xmm3              ;xmm3 = sqrt(delta)
        vaddsd xmm4,xmm1,xmm3               ;xmm4 = -b + sqrt(delta)
        vsubsd xmm5,xmm1,xmm3               ;xmm5 = -b - sqrt(delta)
        vdivsd xmm4,xmm4,xmm0               ;xmm4 = 实数根 r1
        vdivsd xmm5,xmm5,xmm0               ;xmm5 = 实数根 r2
        vmovsd real8 ptr [ecx],xmm4         ;保存 r1
        vmovsd real8 ptr [ecx+8],xmm5       ;保存 r2
        mov dword ptr [edx],1               ;*dis = 1
        jmp done

; 相同实数根
; r1 = r2 = -b / 2 * a
IdenticalRoots:
        vdivsd xmm4,xmm1,xmm0               ;xmm4 = -b / 2 * a
        vmovsd real8 ptr [ecx],xmm4         ;保存 r1
        vmovsd real8 ptr [ecx+8],xmm4       ;保存 r2
        mov dword ptr [edx],0               ;*dis = 0
364     jmp done

; 复数根
; 实部 real = -b / 2 * a, 虚部 imag= sqrt(-delta) / 2 * a
; 方程根: r1 = (real, imag), r2 = (real, -imag)
ComplexRoots:
        vdivsd xmm4,xmm1,xmm0               ;xmm4 = -b / 2 * a
        vxorpd xmm3,xmm3,[FpNegateMask]     ;xmm3 = -delta
        vsqrtsd xmm3,xmm3,xmm3              ;xmm3 = sqrt(-delta)
        vdivsd xmm5,xmm3,xmm0               ;xmm5 = sqrt(-delta) / 2 * a
        vmovsd real8 ptr [ecx],xmm4         ;保存实部
        vmovsd real8 ptr [ecx+8],xmm5       ;保存虚部
        mov dword ptr [edx],-1              ;*dis = -1

Done:   pop ebp
        ret

Error:  mov dword ptr [edx],9999            ;*dis = 9999 (错误码)
        pop ebp
        ret
AvxSfpQuadEqu_ endp
        end
```

一元二次方程是形如 $ax^2+bx+c=0$ 的多项式等式。其中，x 是未知数，a、b 和 c 是常量系数，并且系数 a 必须不为 0。对于 x，一元二次方程总是有两个解，称为根。一元二次方程的根可以使用如下公式计算：

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

根据公式，根的情况有三种，这取决于 a、b、c 的值。表 13-2 描述了这三种情况。示例程序 AvxScalarFloatingPointQuadEqu 使用此公式和表 13-2 所示的判别式来计算一元二次方程的根。

注意 本节中所使用的算法旨在说明如何使用 x86-AVX 指令集计算一元二次方程的根。如想了解更多关于使用二次公式解一元二次方程的信息，请参考附录 C 中所列的文档。

表 13-2 一元二次方程解的三种形式

判别式	根类型	报 述
$b^2-4ac>0$	相异实数根	root1 和 root2 不等
$b^2-4ac=0$	相同实数根	root1 和 root2 相等
$b^2-4ac<0$	复数根	root1 和 root2 不等

365

用 C++ 编写的示例程序 AvxScalarFloatingPointQuadEqu（见清单 13-5）中包含函数 AvxSfpQuadEquCpp，它用来计算二次方程的根。此函数模仿汇编语言函数 AvxSfpQuadEqu_ 编写，其目的是验证计算结果。在 _tmain 函数中包含了一些语句来执行测试用例初始化、结果比较和输出显示。

在 x86-AVX 汇编语言函数 AvxSfpQuadEqu_（见清单 13-6）的开始，使用三条 vmovsd 指令把系数 a、b 和 c 分别加载到寄存器 XMM0、XMM1 和 XMM2，接着的 vmovsd 指令把参数 epsilon 加载到寄存器 XMM7。计算根之前，函数 AvxSfpQuadEqu 必须验证系数 a 是不是不等于 0。需要注意的是，对于浮点数，不推荐直接使用常数比较，浮点运算的特性可能会导致这样的比较突然失效。函数中采用了一种替代的方法，测试系数 a 是否接近 0。由两条指令 vandpd xmm6,xmm0,[FpAbsMask] 和 vcomisd xmm6,xmm7 来完成这项工作，它们判断关系表达式 fabs(a)<epsilon 是否为真。如果表达式为真，则系数 a 被认为是无效的，函数会中止计算。

验证完系数 a 的有效性后，函数 AvxSfpQuadEqu_ 接着计算几个中间值，包括判别式 delta=b*b-4*a*x。如果关系表达式 fabs(delta)<epsilon 为真，delta 被认为等于 0，则跳转到处理相同实数根的部分；如果 delta<0 为真，跳转到处理复数根的部分；如果 delta>0 为真，则跳转到不同实数根的部分进行处理。

表 13-3 总结了函数 AvxSfpQuadEqu_ 用来计算根所用的方程式。计算时使用了前一个中间值和 x86-AVX 标量双精度浮点运算指令 vsqrtsd、vaddsd、vsubsd 和 vdivsd。函数也设置 dis 为如下值：-1（相异实数根）、0（相同实数根）、-1（复数根）、9999（系数 a 无效）。这用来通知调用者所解决的类型，同时便于对结果的进一步处理。

366

表 13-3 根计算公式

条 件	根计算公式
相异实数根	r1 = (-b + sqrt(delta)) / 2 * a r2 = (-b - sqrt(delta)) / 2 * a
相同实数根	r1 = -b / 2 * a r2 = -b / 2 * a
复数根	real = -b / 2 * a; imag = sqrt(-delta) / 2 * a r1 = (real, imag) r2 = (real, -imag)

读者可能已经注意到了，函数 AvxSfpQuadEqu_ 没有使用任何 vmovsd 指令执行寄存器到寄存器的数据传输。实际上这是因为 x86 AVX 的三操作数语法指令实现了同样的直接效果。输出 13-3 中列出了示例程序 AvxScalarFloatingPointQuadEqu 的执行结果。

输出 13-3 示例程序 AvxScalarFloatingQuadEqu

```
Results for AvxScalarFloatingPointQuadEqu
```

```
a: 2.000000, b: 8.000000 c: -15.000000
Distinct real roots
  C++ roots: 1.391165 -5.391165
  AVX roots: 1.391165 -5.391165

a: 1.000000, b: 6.000000 c: 9.000000
Identical roots
  C++ root: -3.000000
  AVX root: -3.000000

a: 3.000000, b: 2.000000 c: 4.000000
Complex roots
  C++ roots: (-0.333333 1.105542) (-0.333333 -1.105542)
  AVX roots: (-0.333333 1.105542) (-0.333333 -1.105542)

a: 0.000000, b: 7.000000 c: -5.000000
Coefficient 'a' is invalid
```

367

13.2.2 球坐标系

本节最后的 x86-AVX 标量浮点示例程序名叫 AvxScalarFloatingPointSpherical，它包含两个汇编语言函数，能够把三维坐标在直角坐标系和球坐标系之间进行转换。示例程序中还演示了如何在使用 x86-AVX 指令的汇编语言函数中调用标准库函数。清单 13-7 和清单 13-8 分别列出了示例程序的 C++ 和汇编语言源代码。

清单 13-7 AvxScalarFloatingPointSpherical.cpp

```
#include "stdafx.h"
#include <float.h>
#define _USE_MATH_DEFINES
#include <math.h>

extern "C" bool RectToSpherical_(const double r_coord[3], double↩
s_coord[3]);
extern "C" bool SphericalToRect_(const double s_coord[3], double↩
r_coord[3]);

extern "C" double DegToRad = M_PI / 180.0;
extern "C" double RadToDeg = 180.0 / M_PI;

void PrintCoord(const char* s, const double c[3])
{
    printf("%s %14.8lf %14.8lf %14.8lf\n", s, c[0], c[1], c[2]);
}

int _tmain(int argc, _TCHAR* argv[])
{
    const int n = 7;

    const double r_coords[n * 3] =
    {
//      x coord     y coord         z coord
        2.0,        3.0,            5.0,
        -2.0,       -2.0,           2.0 * M_SQRT2,
        0.0,        M_SQRT2 / 2.0,  -M_SQRT2 / 2.0,
        M_SQRT2,    1.0,            -1.0,
        0.0,        0.0,            M_SQRT2,
        -1.0,       0.0,            0.0,
```

```
        0.0,        0.0,            0.0,
    };                                                              368
    printf("Results for AvxScalarFloatingPointSpherical\n\n");

    for (int i = 0; i < n; i++)
    {
        double r_coord1[3], s_coord1[3], r_coord2[3];

        r_coord1[0] = r_coords[i * 3];
        r_coord1[1] = r_coords[i * 3 + 1];
        r_coord1[2] = r_coords[i * 3 + 2];

        RectToSpherical (r_coord1, s_coord1);
        SphericalToRect (s_coord1, r_coord2);

        PrintCoord("r_coord1 (x,y,z): ", r_coord1);
        PrintCoord("s_coord1 (r,t,p): ", s_coord1);
        PrintCoord("r_coord2 (x,y,z): ", r_coord2);
        printf("\n");
    }

    return 0;
}
```

清单 13-8 AvxScalarFloatingPointSpherical_.asm

```
                .model flat,c
                .const
Epsilon         real8 1.0e-15
r8_000          real8 0.0
r8_9000         real8 90.0

                .code
                extern DegToRad:real8, RadToDeg:real8
                extern sin:proc, cos:proc, acos:proc, atan2:proc

; extern "C" bool RectToSpherical (const double r_coord[3], double↩
s_coord[3]);
;
; 描述：本函数把直角坐标系转换为球坐标系
;
; 需要：AVX

RectToSpherical_ proc
        push ebp
        mov ebp,esp
        push esi
        push edi
        sub esp,16                          ;为 acos 和 atan2 的参数预留空间          369

; 加载参数值
        mov esi,[ebp+8]                     ;esi = 指向 r_coord
        mov edi,[ebp+12]                    ;edi = 指向 s_coord
        vmovsd xmm0,real8 ptr [esi]         ;xmm0 = x 坐标
        vmovsd xmm1,real8 ptr [esi+8]       ;xmm1 = y 坐标
        vmovsd xmm2,real8 ptr [esi+16]      ;xmm2 = z 坐标

; 计算 r = sqrt(x * x + y * y + z * z)
        vmulsd xmm3,xmm0,xmm0               ;xmm3 = x * x
        vmulsd xmm4,xmm1,xmm1               ;xmm4 = y * y
        vmulsd xmm5,xmm2,xmm2               ;xmm5 = z * z
```

```
        vaddsd xmm6,xmm3,xmm4
        vaddsd xmm6,xmm6,xmm5
        vsqrtsd xmm7,xmm7,xmm6              ;xmm7 = r

; 计算 phi = acos(z / r)
        vcomisd xmm7,real8 ptr [Epsilon]
        jae LB1                             ;如果 r >= epsilon 则跳转
        vmovsd xmm4,real8 ptr [r8_0p0]      ;设置 r 为 0.0
        vmovsd real8 ptr [edi],xmm4         ;保存 r
        vmovsd xmm4,real8 ptr [r8_90p0]     ;phi = 90.0 度
        vmovsd real8 ptr [edi+16],xmm4      ;保存 phi
        jmp LB2

LB1:    vmovsd real8 ptr [edi],xmm7         ;保存 r
        vdivsd xmm4,xmm2,xmm7               ;xmm4 = z / r
        vmovsd real8 ptr [esp],xmm4         ;保存在栈上
        call acos
        fmul real8 ptr [RadToDeg]           ;转换 phi 为角度单位
        fstp real8 ptr [edi+16]             ;保存 phi

; 计算 theta = atan2(y, x)
LB2:    vmovsd xmm0,real8 ptr [esi]         ;xmm0 = x
        vmovsd xmm1,real8 ptr [esi+8]       ;xmm1 = y
        vmovsd real8 ptr [esp+8],xmm0
        vmovsd real8 ptr [esp],xmm1
        call atan2
        fmul real8 ptr [RadToDeg]           ;转换 theta 为角度单位
        fstp real8 ptr [edi+8]              ;保存 theta

        add esp,16
        pop edi
        pop esi
        pop ebp
        ret
370]    RectToSpherical_ endp

; extern "C" bool SphericalToRect(const double s_coord[3], double*
r_coord[3]);
;
;描述：本函数把球坐标系转换为直角坐标系
;
;需要：AVX
;
;局部变量
;   ebp-8     sin(theta)
;   ebp-16    cos(theta)
;   ebp-24    sin(phi)
;   ebp-32    cos(phi)

SphericalToRect_ proc
        push ebp
        mov ebp,esp
        sub esp,32                          ;预留局部变量所用空间
        push esi
        push edi
        sub esp,8                           ;预留 sin 和 cos 参数所需空间

; 函数参数值
        mov esi,[ebp+8]                     ;esi = 指向 s_coord
        mov edi,[ebp-12]                    ;edi = 指向 r_coord

; 计算 sin(theta) 和 cos(theta)
```

```
        vmovsd xmm0,real8 ptr [esi+8]          ;xmm0 = theta
        vmulsd xmm1,xmm0,real8 ptr [DegToRad]  ;xmm1 = theta (弧度单位)
        vmovsd real8 ptr [ebp-16],xmm1         ;保存最后使用的 theta
        vmovsd real8 ptr [esp],xmm1
        call sin
        fstp real8 ptr [ebp-8]             ;保存 sin(theta)
        vmovsd xmm1,real8 ptr [ebp-16]     ;xmm1 =theta (弧度单位)
        vmovsd real8 ptr [esp],xmm1
        call cos
        fstp real8 ptr [ebp-16]            ;保存 cos(theta)

; 计算 sin(phi) 和 cos(phi)
        vmovsd xmm0,real8 ptr [esi+16]         ;xmm0 = phi
        vmulsd xmm1,xmm0,real8 ptr [DegToRad]  ;xmm1 = phi (弧度单位)
        vmovsd real8 ptr [ebp-32],xmm1         ;保存最后使用的 phi
        vmovsd real8 ptr [esp],xmm1
        call sin                                                    371
        fstp real8 ptr [ebp-24]            ;保存 sin(phi)
        vmovsd xmm1,real8 ptr [ebp-32]     ;xmm1 = phi (弧度单位)
        vmovsd real8 ptr [esp],xmm1
        call cos
        fstp real8 ptr [ebp-32]            ;保存 cos(phi)

; 计算 x = r * sin(phi) * cos(theta)
        vmovsd xmm0,real8 ptr [esi]        ;xmm0 = r
        vmulsd xmm1,xmm0,real8 ptr [ebp-24] ;xmm1 = r * sin(phi)
        vmulsd xmm2,xmm1,real8 ptr [ebp-16] ;xmm2 = r*sin(phi)*cos(theta)
        vmovsd real8 ptr [edi],xmm2        ;保存 x

; 计算 y = r * sin(phi) * sin(theta)
        vmulsd xmm2,xmm1,real8 ptr [ebp-8]  ;xmm2 = r*sin(phi)*sin(theta)
        vmovsd real8 ptr [edi+8],xmm2       ;计算 y

; 计算 z = r * cos(phi)
        vmulsd xmm1,xmm0,real8 ptr [ebp-32] ;xmm1 = r * cos(phi)
        vmovsd real8 ptr [edi+16],xmm1      ;计算 z

        add esp,8
        pop edi
        pop esi
        mov esp,ebp
        pop ebp
        ret
SphericalToRect endp
        end
```

在阅读源代码前，我们快速回顾一下三维坐标系的基础知识。三维空间中的点能够被有序元组 (x, y, z) 唯一确定，x、y 和 z 分别表示从原点到两个相互垂直平面的有符号距离。有序元组 (x, y, z) 通常称为直角坐标或者笛卡儿坐标。三维空间中的点也可以由向量 r、角 θ 和角 φ 唯一确定，如图 13-2 所示，有序元组 (r, θ, φ) 被称为球坐标。

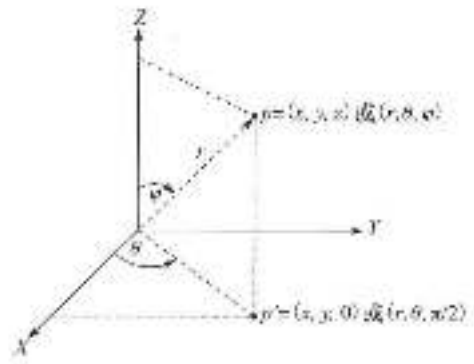


图 13-2 用直角坐标系和球坐标系表示三维空间中的点 372

三维空间中的点可以用直角坐标或者球坐标表示，转换公式如下：

$$r = \sqrt{x^2 + y^2 + z^2}, \quad r \geq 0$$

$$\theta = \arctan\left(\frac{y}{x}\right), \quad -\pi \leq \theta \leq \pi$$

$$\varphi = \cos^{-1}\left(\frac{z}{r}\right), \quad 0 \leq \varphi \leq \pi$$

$$x = r\sin\varphi\cos\theta, \quad y = r\sin\varphi\sin\theta, \quad z = r\cos\varphi$$

计算 θ 值的反正切函数对应于 C++ 标准函数 atan2，它使用有符号的 x 和 y 来确定正确的象限。

在 AvxScalarFloatingPointSpherical.cpp 文件（见清单 13-7）的顶部声明了坐标转换函数。两个函数都使用包含三个双精度浮点元素的数组来表示直角坐标和球坐标。数组 r_coord 中的元素 0、1 和 2 分别对应直角坐标的 x、y 和 z 值，同样，数组 s_coord 中的三个元素分别对应球坐标的 r、θ 和 φ。tmain 函数中准备了一些实例，使用一个简单的循环来
373 测试汇编语言编写的坐标系转换函数，并打印出测试结果。

源文件 AvxScalarFloatingPointSpherical .asm（见清单 13-8）包含两个汇编语言函数：RectToSpherical_ 和 SphericalToRect_。在函数 RectToSpherical_ 的序言之后，指令 sub esp,16 为两个双精度浮点数分配了栈空间，用来存储调用 C++ 标准函数 acos 和 atan2 时的参数。接着把直角坐标的 x、y 和 z 值分别加载到寄存器 XMM0、XMM1 和 XMM2，使用的是一系列 vmovsd 指令。然后，根据之前定义的公式计算 r 值，并将其保存在 s_coord 数组的适当位置。

计算 phi（也就是 φ）之前，函数 RectToSpherical 必须确定 r 是否小于 Epsilon。如果小于，r 会被截断为 0 同时 phi 设置为 90 度；否则，函数 RectToSpherical_ 计算 z/r 并把商存储到堆栈上。然后调用 C++ 标准库函数 acos 计算 z/r 的反余弦值。注意，根据 32 位程序的 Visual C++ 调用约定，调用函数没有必要保存 XMM 寄存器的内容，这意味着在执行完 acos 之后，XMM 寄存器的内容是不确定的。

函数 acos 把返回值保存在 x87 FPU 寄存器栈上，该值从弧度单位转换为角度单位，并存储到数组 s_coord 中。该着计算 theta 值（即 0 值）。C++ 库函数 atan2 所需的参数 x 和 y 在调用 atan2 前复制到了堆栈中。执行完 atan2 之后，x87 FPU 栈中包含以弧度单位表示的 theta。此值也被转换为角度单位，并存储到数组 s_coord 中。

函数 SphericalToRect_ 相比于 RectToSpherical_ 来说复杂一些。在球坐标系转换到直角坐标系时，需要计算若干中间值。SphericalToRect_ 在程序开始时为中间值分配了 32 字节的栈空间，包括 sin(θ)、cos(θ)、sin(π) 和 cos(π)。指令 sub esp,8 为双精度浮点参数值分配了栈空间，用来为库函数 sin 和 cos 传递参数值。

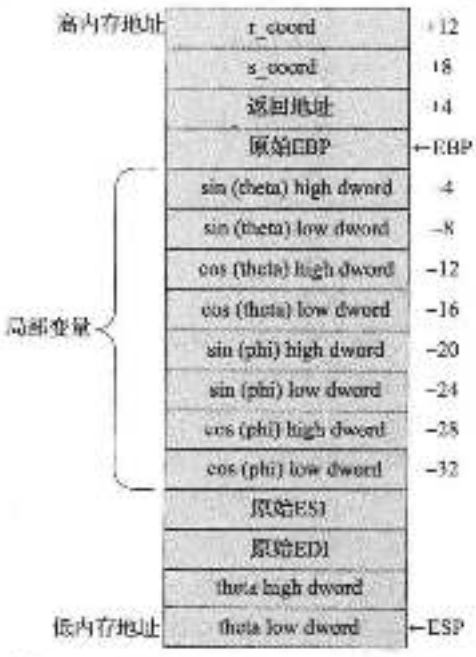


图 13-3 调用 sin 和 cos 指令前栈的内容

在库首和寄存器初始化之后，函数 SphericalToRect 开始计算 sin(θ) 和 cos(θ)。图 13-3 显示了调用 sin 和 cos 前栈的内容。注意在调用 cos 指令前 θ 必须拷贝一份存储在栈上，因为库函数 sin 可能会改变其栈上的原始值。然后用同样的方式计算 sin(π) 和 cos(π)。所需要的正弦值和余弦值得到后，计算相应的直角坐标系分量 x、y 和 z，并保存到数组 r_coord 中。输出 13-4 显示了示例程序 AvxScalarFloatingPointSpherical 的输出结果。

输出 13-4 示例程序 AvxScalarFloatingPointSpherical

```
Results for AvxScalarFloatingPointSpherical

r_coord1 (x,y,z):     2.00000000     3.00000000     6.00000000
s_coord1 (r,t,p):     7.00000000    56.30993247    31.00271913
r_coord2 (x,y,z):     2.00000000     3.00000000     6.00000000

r_coord1 (x,y,z):    -2.00000000    -2.00000000     2.82842712
s_coord1 (r,t,p):     4.00000000  -135.00000000    45.00000000
r_coord2 (x,y,z):    -2.00000000    -2.00000000     2.82842712

r_coord1 (x,y,z):     0.00000000     0.70710678    -0.70710678
s_coord1 (r,t,p):     1.00000000    90.00000000   135.00000000
r_coord2 (x,y,z):     0.00000000     0.70710678    -0.70710678

r_coord1 (x,y,z):     1.41421356     1.00000000     1.00000000
s_coord1 (r,t,p):     2.00000000    35.26438968   120.00000000
r_coord2 (x,y,z):     1.41421356     1.00000000     1.00000000

r_coord1 (x,y,z):     0.00000000     0.00000000     1.41421356
s_coord1 (r,t,p):     1.41421356     0.00000000     0.00000000
r_coord2 (x,y,z):     0.00000000     0.00000000     1.41421356

r_coord1 (x,y,z):    -1.00000000     0.00000000     0.00000000
s_coord1 (r,t,p):     1.00000000   180.00000000    90.00000000
r_coord2 (x,y,z):    -1.00000000     0.00000000     0.00000000

r_coord1 (x,y,z):     0.00000000     0.00000000     0.00000000
s_coord1 (r,t,p):     0.00000000     0.00000000    90.00000000
r_coord2 (x,y,z):     0.00000000     0.00000000     0.00000000
```

374

375

13.3 总结

在本章中，我们学习了如何使用 x86-AVX 指令集执行标量浮点运算。你应该渐渐了解了一些 x86-SSE 和 x86_AVX 的差异。正如本章的示例程序所展示的，x86_AVX 为编程人员提供了许多有益特征，包括简化汇编编程以及减少寄存器到寄存器的数据传输。下一章中，我们将继续学习 x86-AVX 指令集的组合浮点运算功能。

376

第 14 章

Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX

x86-AVX 组合浮点编程

第 9 章中，我们已经熟悉了 x86-SSE 的组合浮点相关知识，本章将探索 x86-AVX 的组合浮点运算能力。本章的示例程序将演示如何对 256 位宽的操作数做基本的组合浮点运算，也将展示如何使用 x86-AVX 指令来对浮点数组和矩阵做运算。本章中的例子只能在支持 AVX 的处理器和操作系统上运行。另外，附录 C 中列出了一些可自由使用的工具，可以用它们来确定你的 PC 是否支持 AVX。

14.1 编程基础

本节中有两个示例程序，用来演示使用 x86-AVX 指令集对组合浮点数做基本操作。第一个示例程序对 256 位宽组合操作数执行单精度和双精度浮点运算，第二个示例程序演示组合浮点数的比较操作。这些示例程序还演示了 x86-AVX 编程的几个注意事项，包括操作数对齐和合理使用 vzeroupper 指令。

本章和随后章节的几个例子都会使用一个名为 YmmVal 的 C++ 联合结构，如清单 14-1 所示，它用来在 C++ 和汇编语言函数间传递数据。在此联合体中声明的各个项与 256 位宽操作数的组合数据类型相对应。联合体 YmmVal 还包含若干文本字符串格式化函数的声明。在子文件夹 CommonFiles 中包含了文件 YmmVal.cpp（源代码没有列出），其中有 ToString_ 格式化函数的定义。

377

清单 14-1 YmmVal.h

```
#pragma once
#include "MiscDefs.h"

union YmmVal
{
    Int8 i8[32];
    Int16 i16[16];
    Int32 i32[8];
    Int64 i64[4];
    Uint8 u8[32];
    Uint16 u16[16];
    Uint32 u32[8];
    Uint64 u64[4];
    float r32[8];
    double r64[4];

    char* ToString_i8(char* s, size_t len, bool upper_half);
    char* ToString_i16(char* s, size_t len, bool upper_half);
    char* ToString_i32(char* s, size_t len, bool upper_half);
    char* ToString_i64(char* s, size_t len, bool upper_half);

    char* ToString_u8(char* s, size_t len, bool upper_half);
    char* ToString_u16(char* s, size_t len, bool upper_half);
    char* ToString_u32(char* s, size_t len, bool upper_half);
    char* ToString_u64(char* s, size_t len, bool upper_half);
```

```
    char* ToString_x8(char* s, size_t len, bool upper_half);
    char* ToString_x16(char* s, size_t len, bool upper_half);
    char* ToString_x32(char* s, size_t len, bool upper_half);
    char* ToString_x64(char* s, size_t len, bool upper_half);

    char* ToString_r32(char* s, size_t len, bool upper_half);
    char* ToString_r64(char* s, size_t len, bool upper_half);
};
```

14.1.1 组合浮点运算

第一个示例程序名为 AvxPackedFloatingPointArithmetic，演示了如何对 256 位宽的组合浮点操作数执行常见的算术运算。同时也演示了如何正确地使用 vzeroupper 指令，以防止函数使用 YMM 寄存器时产生可能的性能损失。清单 14-2 和清单 14-3 列出了 AvxPackedFloatingPointArithmetic 程序的 C++ 和汇编语言源代码。 378

清单 14-2 AvxPackedFloatingPointArithmetic.cpp

```
#include "stdafx.h"
#include "YmmVal.h"

extern "C" void AvxPfpArithmeticFloat_(const YmmVal* a, const YmmVal* b, ↩
YmmVal c[6]);
extern "C" void AvxPfpArithmeticDouble_(const YmmVal* a, const YmmVal* b, ↩
YmmVal c[5]);

void AvxPfpArithmeticFloat(void)
{
    __declspec(align(32)) YmmVal a;
    __declspec(align(32)) YmmVal b;
    __declspec(align(32)) YmmVal c[6];

    a.r32[0] = 3.0f;      b.r32[0] = 12.5f;
    a.r32[1] = 1.5f;      b.r32[1] = 32.125f;
    a.r32[2] = -10.75f;   b.r32[2] = 17.5f;
    a.r32[3] = 15.0f;     b.r32[3] = 14.982f;
    a.r32[4] = -32.125f;  b.r32[4] = -4.75f;
    a.r32[5] = 3.875f;    b.r32[5] = 3.0625f;
    a.r32[6] = 2.0f;      b.r32[6] = 7.875f;
    a.r32[7] = -6.25f;    b.r32[7] = -48.1875f;

    AvxPfpArithmeticFloat_(&a, &b, c);

    printf("Results for AvxPfpArithmeticFloat()\n\n");

    printf(" i       a        b      Add      Sub      Mul      Div ↩
      Abs      Neg\n");
    printf("-----------------------------------------------------------\n");

    for (int i = 0; i < 8; i++)
    {
        const char* fs = "%8.3f ";

        printf("%2d ", i);
        printf(fs, a.r32[i]);
        printf(fs, b.r32[i]);
        printf(fs, c[0].r32[i]);
        printf(fs, c[1].r32[i]);
        printf(fs, c[2].r32[i]);
        printf(fs, c[3].r32[i]);
```

```
            printf(fs, c[4].r32[1]);
            printf(fs, c[5].r32[1]);
            printf("\n");
        }
[379]   }

        void AvxPfpArithmeticDouble(void)
        {
            __declspec(align(32)) YmmVal a;
            __declspec(align(32)) YmmVal b;
            __declspec(align(32)) YmmVal c[5];

            a.r64[0] = 12.0;      b.r64[0] = 0.875;
            a.r64[1] = 13.5;      b.r64[1] = -125.75;
            a.r64[2] = 18.75;     b.r64[2] = 72.5;
            a.r64[3] = 5.0;       b.r64[3] = -98.375;

            AvxPfpArithmeticDouble_(&a, &b, c);

            printf("\n\nResults for AvxPfpArithmeticDouble()\r\n");

            printf(" i     a       b      Min     Max    Sqrt a    HorAdd HorSub\n");
            printf("-------------------------------------------------------------\n");

            for (int i = 0; i < 4; i++)
            {
                const char* fs = "%9.3lf ";

                printf("%2d ", i);
                printf(fs, a.r64[i]);
                printf(fs, b.r64[i]);
                printf(fs, c[0].r64[i]);
                printf(fs, c[1].r64[i]);
                printf(fs, c[2].r64[i]);
                printf(fs, c[3].r64[i]);
                printf(fs, c[4].r64[i]);
                printf("\n");
            }
        }

        int _tmain(int argc, _TCHAR* argv[])
        {
            AvxPfpArithmeticFloat();
            AvxPfpArithmeticDouble();
            return 0;
        }
[380]
```

清单 14-3 AvxPackedFloatingPointArithmetic_.asm

```
        .model flat,c
        .const
        align 16

; 组合单精度浮点数绝对值的掩码
AbsMask dword 7fffffffh,7fffffffh,7fffffffh,7fffffffh
        dword 7fffffffh,7fffffffh,7fffffffh,7fffffffh

; 组合单精度浮点负数的掩码
NegMask dword 80000000h,80000000h,80000000h,80000000h
        dword 80000000h,80000000h,80000000h,80000000h
        .code
```

```
; extern "C" void AvxPfpArithmeticFloat_(const YmmVal* a, const YmmVal* b, ←
YmmVal c[6]);
;
; 描述：本例程演示如何使用常用的组合单精度浮点运算指令（使用 YMM 寄存器）
;
; 需要：AVX

AvxPfpArithmeticFloat_ proc
        push ebp
        mov ebp,esp

; 加载参数值，注意 vmovaps 指令对内存中的操作数有相应的对齐要求
        mov eax,[ebp+8]                 ;eax = 指向 a
        mov ecx,[ebp+12]                ;ecx = 指向 b
        mov edx,[ebp+16]                ;edx = 指向 c
        vmovaps ymm0,ymmword ptr [eax]  ;ymm0 = a
        vmovaps ymm1,ymmword ptr [ecx]  ;ymm1 = b

; 执行组合单精度浮点数的加、减、乘、除运算
        vaddps ymm2,ymm0,ymm1           ;a + b
        vmovaps ymmword ptr [edx],ymm2

        vsubps ymm3,ymm0,ymm1           ;a - b
        vmovaps ymmword ptr [edx+32],ymm3

        vmulps ymm4,ymm0,ymm1           ;a * b
        vmovaps ymmword ptr [edx+64],ymm4                                    381

        vdivps ymm5,ymm0,ymm1           ;a / b
        vmovaps ymmword ptr [edx+96],ymm5

; 计算组合单精度浮点数的绝对值
        vmovups ymm6,ymmword ptr [AbsMask]      ;ymm6 = AbsMask
        vandps ymm7,ymm0,ymm6                   ;ymm7 = 组合绝对值 (packed fabs)
        vmovaps ymmword ptr [edx+128],ymm7

; 计算组合单精度浮点数的负值
        vxorps ymm7,ymm0,ymmword ptr [NegMask]  ;ymm7 = 组合负值 (packed neg)
        vmovaps ymmword ptr [edx+160],ymm7

; 对 YMM 寄存器的高 128 位清零，以避免 x86-AVX 到 x86-SSE 潜在的转换问题
        vzeroupper

        pop ebp
        ret
AvxPfpArithmeticFloat_ endp

; extern "C" void AvxPfpArithmeticDouble_(const YmmVal* a, const YmmVal* b, ←
YmmVal c[5]);
;
; 描述：本函数演示如何使用常用的组合双精度浮点运算指令（使用 YMM 寄存器）
;
; 需要：AVX


AvxPfpArithmeticDouble_ proc
        push ebp
        mov ebp,esp

; 加载参数值，注意 vmovaps 指令对内存中的操作数有相应的对齐要求
        mov eax,[ebp+8]                 ;eax = 指向 a
```

```
        mov ecx,[ebp+12]                    ;ecx = 指向 b
        mov edx,[ebp+16]                    ;edx = 指向 c
        vmovapd ymm0,ymmword ptr [eax]      ;ymm0 = a
        vmovapd ymm1,ymmword ptr [ecx]      ;ymm1 = b

; 计算最小值、最大值和平方根
        vminpd ymm2,ymm0,ymm1
        vmaxpd ymm3,ymm0,ymm1
        vsqrtpd ymm4,ymm0

; 进行水平加减运算
        vhaddpd ymm5,ymm0,ymm1
        vhsubpd ymm6,ymm0,ymm1

; 保存结果
        vmovapd ymmword ptr [edx],ymm2
        vmovapd ymmword ptr [edx+32],ymm3
        vmovapd ymmword ptr [edx+64],ymm4
        vmovapd ymmword ptr [edx+96],ymm5
        vmovapd ymmword ptr [edx+128],ymm6

; 对 YMM 寄存器的高 128 位清零，以避免 x86-AVX 到 x86-SSE 潜在的转换问题
        vzeroupper

        pop ebp
        ret
AvxPfpArithmeticDouble_ endp
        end
```

AvxPackedFloatingPointArithmetic 的 C++ 代码（见清单 14-2）中包含一个函数 AvxPfpArithmeticFloat，其开始部分使用单精度浮点测试值初始化了若干 YmmVal 变量。注意，每个 YmmVal 实例都使用了 Visual C++ 扩展属性 __declspec(align(32)) 进行 32 字节对齐。函数 AvxPfpArithmeticFloat 调用了 x86-AVX 汇编语言函数 AvxPfpArithmeticFloat_，后者用来演示常用的组合单精度浮点运算。使用一系列的 printf 语句，以阵列的方式打印出计算的结果。在 C++ 代码中还有一个 AvxPfpArithmeticDouble 函数，用来演示组合双精度浮点数运算，其逻辑组织类似于组合单精度浮点运算。

清单 14-3 列出了示例程序 AvxPackedFloatingPointArithmetic 的 x86-AVX 汇编代码。在其顶部的 .const 区定义了 32 字节宽的组合掩码 AbsMask，用来计算组合单精度浮点数的绝对值。第二个组合掩码是 NegMask，用来计算单精度浮点数的负值。需要注意的是，在 .const 区，align 指示的参数值不能超过 16。这意味着 AbsMask 和 NegMask 可能不能正确对齐。由于 x86-AVX 宽泛的对齐要求（见第 12 章），大多数 x86-AVX 指令引用这些值时不会触发处理器异常。另一种选择是使用 MASM 段指令来定义一个常量区域，允许数据 32 字节对齐。这种方法将在第 15 章介绍。

383 继序言和参数寄存器初始化之后，指令 vmovaps ymm0, ymmword ptr [eax] 加载组合值 a 到寄存器 YMM0。vmovaps 指令要求基于内存的源操作数正确对齐。然后，指令 vmovaps ymm1,ymmword ptr [ecx] 把组合值 b 加载到寄存器 YMM1。接下来的指令 vaddps ymm2,ymm0,ymm1 将 YMM0 和 YMM1 中的组合单精度数进行相加，并将结果存入 YMM2。函数接着分别使用指令 vsubps、vmulps 和 vdivps 执行组合单精度浮点数的减法、乘法和除法。

随后的两条指令 vmovups ymm6,ymmword ptr [AbsMask] 和 vandps ymm7,ymm0,ymm6 计算 a 的组合绝对值。注意不能用 vmovaps 指令来加载 AbsMask 到 YMM 寄存器，因为此掩码值没有正确对齐。指令 vxorps ymm7,ymm0,ymmword ptr [NegMask] 计算元素 a 的负值，这也体现了 x86-AVX 宽泛的内存对齐要求，因为 NegMask 并没有明确对齐到 32 字节边界。函数结语前的最后指令是 vzeroupper，这是为了预防 x86 处理器从执行 x86-AVX 指令过渡到执行 x86-SSE 指令时触发潜在的性能损失。第 12 章中讨论了这个问题的详细细节。

函数 AvxPfpArithmeticDouble（见清单 14-3）演示了若干其他组合浮点运算操作，使用的是双精度浮点数，而不是单精度浮点数。指令 vminpd、vmaxpd 和 vsqrtpd 分别计算组合双精度最小值、最大值和平方根。水平（相邻元素）加减运算使用了指令 vhaddpd 和 vhsubpd（见图 7-8）。这些操作的结果被调用者使用一系列 vmovapd 指令保存到指定的数组中。AvxPfpArithmeticDouble 函数的结语前使用的最后一条指令是 vzeroupper。需要重申的是，当函数使用 YMM 寄存器时，为了防止处理器状态转换延迟，都应该使用这条指令（在任何 ret 指令之前）。第 12 章中包含了更详细的关于正确使用 vzeroupper 指令的信息。输出 14-1 显示了示例程序 AvxPackedFloatingPointArithmetic 的运行结果。

输出 14-1 示例程序 AvxPackedFloatingPointArithmetic

```
Results for AvxPfpArithmeticFloat()

i       a       b      Add      Sub      Mul      Div      Abs      Neg
-------------------------------------------------------------------------
0   2.000   12.500   14.500  -10.500   25.000    0.160    2.000   -2.000
1   3.500   52.125   55.625  -48.625  182.438    0.067    3.500   -3.500
2 -10.750   17.500    6.750  -28.250 -188.125   -0.614   10.750   10.750
3  15.000   13.982   28.982    1.018  209.730    1.073   15.000  -15.000
4 -12.125   -4.750  -16.875   -7.375   57.594    2.553   12.125   12.125
5   3.875    3.066    6.938    0.814   11.867    1.265    3.875   -3.875
6   2.000    7.875    9.875   -5.875   15.750    0.254    2.000   -2.000
7  -6.350  -48.188  -54.537   41.838  305.991    0.132    6.350    6.350

Results for AvxPfpArithmeticDouble()

i       a         b       Min       Max    Sqrt a    HorAdd    HorSub
-------------------------------------------------------------------------
0  12.000     0.875     0.875    12.000     3.464    25.500    -1.500
1  13.500  -125.250  -125.250    13.500     3.674  -124.375   126.125
2  18.750    72.500    18.750    72.500     4.330    23.750    13.750
3   5.000   -98.375   -98.375     5.000     2.236   -25.875   170.875
```

384

14.1.2 组合浮点比较

第 13 章中，我们学习了如何使用 vcmpsd 指令对两个标量双精度浮点数进行比较。本节中，我们将学习使用 vcmppd 指令对两个组合双精度浮点数进行比较。该指令对成对的两个源操作数进行比较，然后设置相应的目标操作数来显示比较结果（全 1 为真，全 0 为假）。示例程序 AvxPackedFloatingPointCompare 的 C++ 和汇编语言源代码分别见清单 14-4 和清单 14-5。

清单 14-4 AvxPackedFloatingPointCompare.cpp

```
#include "stdafx.h"
#include "YmmVal.h"
```

```
#include <limits>
using namespace std;

extern "C" void AvxPfpCompare (const YmmVal* a, const YmmVal* b, YmmVal c[8]);

int _tmain(int argc, _TCHAR* argv[])
{
    char buff[256];
    __declspec(align(32)) YmmVal a;
    __declspec(align(32)) YmmVal b;
    __declspec(align(32)) YmmVal c[8];

    const char* instr_names[8] =
    {
        "vcmpeqpd", "vcmpneqpd", "vcmpltpd", "vcmplepd",
        "vcmpgtpd", "vcmpgepd", "vcmpordpd", "vcmpunordpd"
    };

    a.r64[0] = 42.125;
    a.r64[1] = -36.875;
    a.r64[2] = 22.95;
    a.r64[3] = 3.75;
    b.r64[0] = -0.0625;
    b.r64[1] = -67.375;
    b.r64[2] = 22.95;
    b.r64[3] = numeric_limits<double>::quiet_NaN();

    AvxPfpCompare_(&a, &b, c);

    printf("Results for AvxPackedFloatingPointCompare\n");
    printf("a: %s\n", a.ToString_r64(buff, sizeof(buff), false));
    printf("a: %s\n", a.ToString_r64(buff, sizeof(buff), true));
    printf("\n");
    printf("b: %s\n", b.ToString_r64(buff, sizeof(buff), false));
    printf("b: %s\n", b.ToString_r64(buff, sizeof(buff), true));

    for (int i = 0; i < 8; i++)
    {
        printf("\n%s results\n", instr_names[i]);
        printf("  %s\n", c[i].ToString_x64(buff, sizeof(buff), false));
        printf("  %s\n", c[i].ToString_x64(buff, sizeof(buff), true));
    }

    return 0;
}
```

385

清单 14-5 AvxPackedFloatingPointCompare_.asm

```
        .model flat,c
        .code

; extern "C" void AvxPfpCompare_(const YmmVal* a, const YmmVal* b, YmmVal c[8]);
;
; 描述：本函数演示 x86-AVX 比较指令 vcmppd 的使用方法
;
; 需要：AVX

AvxPfpCompare_ proc
        push ebp
        mov ebp,esp

; 加载参数值
```

```
        mov eax,[ebp+8]                 ;eax = 指向 a
        mov ecx,[ebp+12]                ;ecx = 指向 b
        mov edx,[ebp+16]                ;edx = 指向 c
        vmovapd ymm0,ymmword ptr [eax]  ;ymm0 = a
        vmovapd ymm1,ymmword ptr [ecx]  ;ymm1 = b

; 相等比较
        vcmpeqpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [edx],ymm2

; 不等比较
        vcmpneqpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [edx+32],ymm2

; 小于比较
        vcmpltpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [edx+64],ymm2

; 小于等于比较
        vcmplepd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [edx+96],ymm2

; 大于比较
        vcmpgtpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [edx+128],ymm2

; 大于等于比较
        vcmpgepd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [edx+160],ymm2

; 有序比较
        vcmpordpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [edx+192],ymm2

; 无序比较
        vcmpunordpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [edx+224],ymm2

; 对 YMM 寄存器的高 128 位清零，以避免 x86-AVX 到 x86-SSE 潜在的转换问题
        vzeroupper
        pop ebp
        ret
AvxPfpCompare_ endp
        end
```

AvxPackedFloatingPointCompare.cpp（见清单 14-4）的 _tmain 函数使用 C++ 联合结构 YmmVal 初始化两个组合双精度浮点数供测试使用。注意，最后一个元素 b 被设置为 QNaN，这是为了进行无序浮点比较。_tmain 中剩余的代码调用了汇编语言函数 _AvxPfpCompare，并且把执行结果打印出来。其中的 printf 语句把每个组合双精度比较的结果以掩码形式显示在屏幕上。

清单 14-5 包含了汇编语言函数 AvxPfpCompare_，该函数使用 vmovapd 指令把参数 a 和 b 分别加载到寄存器 YMM0 和 YMM1，然后使用指令 vcmppd 的常用形式执行比较，并把每个比较结果的掩码存储到指定数组中。注意，在函数结语前仍旧使用了 vzeroupper 指令，以避免 x86-AVX 到 x86-SSE 状态转换中的性能损失。

与其标量版本类似，指令 vcmppd 支持两种格式：四操作数格式使用一个立即数来指定比较谓词；三操作数形式的比较谓词在其助记符的字符串中表现，比如 vcmpeqpd 表示

进行相等比较。和 vcmpsd 指令一样，vcmppd 也支持同样的 32 种比较谓词。在标量中操作单精度浮点量的时候，指令 vcmpps 也可以用来执行比较操作。输出 14-2 显示了示例程序 AvxPackedFloatingPointCompare 的运行结果。

输出 14-2 示例程序 AvxPackedFloatingPointCompare

```
Results for AvxPackedFloatingPointCompare
a:          42.125000000000 |          -36.875000000000
a:          22.950000000000 |            3.750000000000

b:          -0.062500000000 |          -67.375000000000
b:          22.950000000000 |            1.NQNAN0000000

vcmpeqpd results
  0000000000000000 | 0000000000000000
  FFFFFFFFFFFFFFFF | 0000000000000000

vcmpneqpd results
  FFFFFFFFFFFFFFFF | FFFFFFFFFFFFFFFF
  0000000000000000 | FFFFFFFFFFFFFFFF

vcmpltpd results
  0000000000000000 | 0000000000000000
  0000000000000000 | 0000000000000000

vcmplepd results
  0000000000000000 | 0000000000000000
  FFFFFFFFFFFFFFFF | 0000000000000000

vcmpgtpd results
  FFFFFFFFFFFFFFFF | FFFFFFFFFFFFFFFF
  0000000000000000 | 0000000000000000

vcmpgepd results
  FFFFFFFFFFFFFFFF | FFFFFFFFFFFFFFFF
388 FFFFFFFFFFFFFFFF | 0000000000000000

vcmpordpd results
  FFFFFFFFFFFFFFFF | FFFFFFFFFFFFFFFF
  FFFFFFFFFFFFFFFF | 0000000000000000

vcmpunordpd results
  0000000000000000 | 0000000000000000
  0000000000000000 | FFFFFFFFFFFFFFFF
```

14.2 高级编程

本节的几个示例程序用来演示如何对组合双精度浮点操作数进行 x86-AVX 高级编程。第一个示例程序演示如何使用 x86-AVX 指令集计算相关系数（correlation coefficient）。第二个示例程序计算双精度浮点数矩阵的列均值。这些例子使用的方法可以应用到处理单精度或双精度浮点数组和矩阵的类似程序中。

14.2.1 相关系数

接下来的示例程序名为 AvxPackedFloatingPointCorrCoef，演示了使用 x86-AVX 的组合浮点功能来计算统计学中的相关系数。同时演示了对组合浮点数做各种常见操作，包括提

取和组合水平加法。清单 14-6 和清单 14-7 分别列出了 AvxPackedFloatingPointCorrCoef 的 C++ 和汇编语言源代码。

清单 14-6 AvxPackedFloatingPointCorrCoef.cpp

```
#include "stdafx.h"
#include <math.h>
#include <stdio.h>

extern "C" __declspec(align(32)) double CcEpsilon = 1.0e-12;
extern "C" bool AvxPfpCorrCoef (const double* x, const double* y, int n,
double sums[5], double* rho);

bool AvxPfpCorrCoefCpp(const double* x, const double* y, int n, double
sums[5], double* rho)
{
    double sum_x = 0, sum_y = 0;
    double sum_xx = 0, sum_yy = 0, sum_xy = 0;

    // 确保 x 和 y 正确地对齐到 32 字节边界
    if (((uintptr_t)x & 0x1f) != 0)
        return false;
    if (((uintptr_t)y & 0x1f) != 0)
        return false;

    // 确保 n 是有效的
    if ((n < 4) || ((n & 3) != 0))
        return false;

    // 计算、保存和变量
    for (int i = 0; i < n; i++)
    {
        sum_x += x[i];
        sum_y += y[i];
        sum_xx += x[i] * x[i];
        sum_yy += y[i] * y[i];
        sum_xy += x[i] * y[i];
    }

    sums[0] = sum_x;
    sums[1] = sum_y;
    sums[2] = sum_xx;
    sums[3] = sum_yy;
    sums[4] = sum_xy;

    // 计算 rho
    double rho_num = n * sum_xy - sum_x * sum_y;
    double rho_den = sqrt(n * sum_xx - sum_x * sum_x) * sqrt(n * sum_yy -
sum_y * sum_y);

    if (rho_den >= CcEpsilon)
    {
        *rho = rho_num / rho_den;
        return true;
    }
    else
    {
        *rho = 0;
        return false;
    }
}
```

389

```
int _tmain(int argc, _TCHAR* argv[])
{
    const int n = 100;
    __declspec(align(32)) double x[n];
390 __declspec(align(32)) double y[n];
    double sums1[5], sums2[5];
    double rho1, rho2;

    srand(17);
    for (int i = 0; i < n; i++)
    {
        x[i] = rand();
        y[i] = x[i] + ((rand() % 6000) - 3000);
    }

    bool rc1 = AvxPfpCorrCoefCpp(x, y, n, sums1, &rho1);
    bool rc2 = AvxPfpCorrCoef_(x, y, n, sums2, &rho2);

    printf("Results for AvxPackedFloatingPointCorrCoef\n\n");

    if (!rc1 || !rc2)
    {
        printf("Invalid return code (rc1: %d, rc2: %d)\n", rc1, rc2);
        return 1;
    }

    printf("rho1: %.8lf  rho2: %.8lf\n", rho1, rho2);
    printf("\n");
    printf("sum_x:  %12.0lf %12.0lf\n", sums1[0], sums2[0]);
    printf("sum_y:  %12.0lf %12.0lf\n", sums1[1], sums2[1]);
    printf("sum_xx: %12.0lf %12.0lf\n", sums1[2], sums2[2]);
    printf("sum_yy: %12.0lf %12.0lf\n", sums1[3], sums2[3]);
    printf("sum_xy: %12.0lf %12.0lf\n", sums1[4], sums2[4]);
    return 0;
}
```

清单 14-7 AvxPackedFloatingPointCorrCoef_.asm

```
        .model flat,c
        .code
        extern CrEpsilon:real8

; extern "C" bool AvxPfpCorrCoef_(const double* x, const double* y, int n, ↩
double sums[5], double* rho);
;
; 描述：本函数计算给定 x 和 y 数组的相关系数
;
; 需要：AVX

AvxPfpCorrCoef_ proc
        push ebp
391     mov ebp,esp

; 加载并验证参数值
        mov eax,[ebp+8]                 ;eax = 指向 x
        test eax,1fh
        jnz BadArg                      ;如果 x 没有对齐则跳转
        mov edx,[ebp+12]                ;edx = 指向 y
        test edx,1fh
        jnz BadArg                      ;如果 y 没有对齐则跳转
```

```
        mov ecx,[ebp+16]                ;ecx = n
        cmp ecx,4
        jl BadArg                       ;如果 n < 4 则跳转
        test ecx,3
        jnz BadArg                      ;不是 4 整数倍则跳转
        shr ecx,2                       ;ecx = 循环数

; 初始化和变量为 0
        vxorpd ymm3,ymm3,ymm3           ;ymm3 = 组合 sum_x
        vmovapd ymm4,ymm3               ;ymm4 = 组合 sum_y
        vmovapd ymm5,ymm3               ;ymm5 = 组合 sum_xx
        vmovapd ymm6,ymm3               ;ymm6 = 组合 sum_yy
        vmovapd ymm7,ymm3               ;ymm7 = 组合 sum_xy

; 计算中间组合和变量
@@:     vmovapd ymm0,ymmword ptr [eax]  ;ymm0 = 组合 x 值
        vmovapd ymm1,ymmword ptr [edx]  ;ymm1 = 组合 y 值

        vaddpd ymm3,ymm3,ymm0           ;更新组合 sum_x
        vaddpd ymm4,ymm4,ymm1           ;更新组合 sum_y

        vmulpd ymm2,ymm0,ymm1           ;ymm2 = 组合 xy 值
        vaddpd ymm7,ymm7,ymm2           ;更新组合 sum_xy

        vmulpd ymm0,ymm0,ymm0           ;ymm0 = 组合 xx 值
        vmulpd ymm1,ymm1,ymm1           ;ymm1 = 组合 yy 值
        vaddpd ymm5,ymm5,ymm0           ;更新组合 sum_xx
        vaddpd ymm6,ymm6,ymm1           ;更新组合 sum_yy

        add eax,32                      ;更新 x 指针
        add edx,32                      ;更新 y 指针
        dec ecx                         ;更新循环计数
        jnz @@                          ;如果没有完成，重复上述过程

; 计算最后的和变量
        vextractf128 xmm0,ymm3,1
        vaddpd xmm3,xmm0,xmm3
        vhaddpd xmm3,xmm3,xmm3          ;xmm3[63:0] = sum_x

        vextractf128 xmm0,ymm4,1
        vaddpd xmm4,xmm0,xmm4
        vhaddpd xmm4,xmm4,xmm4          ;xmm4[63:0] = sum_y

        vextractf128 xmm0,ymm5,1
        vaddpd xmm5,xmm0,xmm5
        vhaddpd xmm5,xmm5,xmm5          ;xmm5[63:0] = sum_xx

        vextractf128 xmm0,ymm6,1
        vaddpd xmm6,xmm0,xmm6
        vhaddpd xmm6,xmm6,xmm6          ;xmm6[63:0] = sum_yy

        vextractf128 xmm0,ymm7,1
        vaddpd xmm7,xmm0,xmm7
        vhaddpd xmm7,xmm7,xmm7          ;xmm7[63:0] = sum_xy

; 保存最后的和变量
        mov eax,[ebp+20]                ;eax = 指向 sums 数组
        vmovsd real8 ptr [eax],xmm3     ;保存 sum_x
        vmovsd real8 ptr [eax+8],xmm4   ;保存 sum_y
        vmovsd real8 ptr [eax+16],xmm5  ;保存 sum_xx
        vmovsd real8 ptr [eax+24],xmm6  ;保存 sum_yy
        vmovsd real8 ptr [eax+32],xmm7  ;保存 sum_xy
```

[392]

```
; 计算 rho 分子
; rho_num = n * sum_xy - sum_x * sum_y;
        vcvtsi2sd xmm2,xmm2,dword ptr [ebp+16]  ;xmm2 = n
        vmulsd xmm0,xmm2,xmm7                   ;xmm0= n * sum_xy
        vmulsd xmm1,xmm3,xmm4                   ;xmm1 = sum_x * sum_y
        vsubsd xmm7,xmm0,xmm1                   ;xmm7 = rho_num

; 计算 rho 分母
; t1 = sqrt(n * sum_xx - sum_x * sum_x)
; t2 = sqrt(n * sum_yy - sum_y * sum_y)
; rho_den = t1 * t2
        vmulsd xmm0,xmm2,xmm5       ;xmm0 = n * sum_xx
        vmulsd xmm3,xmm3,xmm3       ;xmm3 = sum_x * sum_x
        vsubsd xmm3,xmm0,xmm3       ;xmm3 = n * sum_xx - sum_x * sum_x
        vsqrtsd xmm3,xmm3,xmm3      ;xmm3 = t1

        vmulsd xmm0,xmm2,xmm6       ;xmm0 = n * sum_yy
        vmulsd xmm4,xmm4,xmm4       ;xmm4 = sum_y * sum_y
        vsubsd xmm4,xmm0,xmm4       ;xmm4 = n * sum_yy - sum_y * sum_y
        vsqrtsd xmm4,xmm4,xmm4      ;xmm4 = t2

393     vmulsd xmm0,xmm3,xmm4       ;xmm0 = rho_den

; 计算最后的 rho 值
        xor eax,eax                 ;eax 的各位清零
        vcomisd xmm0,[CcEpsilon]    ;is rho_den < CcEpsilon?
        setae al                    ;设置返回码
        jb BadRho                   ;如果 rho_den < CcEpsilon 则失败

        vdivsd xmm1,xmm7,xmm0       ;xmm1 = rho
SvRho:  mov edx,[ebp+24]            ;eax = 指向 rho
        vmovsd real8 ptr [edx],xmm1 ;保存 rho

        vzeroupper
Done:   pop ebp
        ret

; 出错处理
BadRho: vxorpd xmm1,xmm1,xmm1       ;rho = 0
        jmp SvRho
BadArg: xor eax,eax                 ;eax = 无效参数返回码
        jmp Done

AvxPtpCorrCoef_ endp
        end
```

相关系数反映两组数据集合或者变量之间线性相关的程度。相关系数的取值范围在 -1 到 +1 之间，+1 和 -1 分别表示变量间完美负线性相关和完美正线性相关，现实世界中的相关系数不大可能等于这两个值。相关系数为 0 表示数据集不是线性相关的。示例程序 AvxPackedFloatingPointCorrCoef 使用下面的公式计算相关系数：

$$\rho = \frac{n\sum_i x_i y_i - \sum_i x_i \sum_i y_i}{\sqrt{\left[n\sum_i x_i^2 - \left(\sum_i x_i\right)^2\right]}\sqrt{\left[n\sum_i y_i^2 - \left(\sum_i y_i\right)^2\right]}}$$

从公式中可以看出，程序在计算相关系数的过程中必须计算如下五个和变量：

$$\text{sum\_x} = \sum_i x_i \quad \text{sum\_y} = \sum_i y_i$$

$$sum\_xx = \sum x_i^2 \quad sum\_yy = \sum y_i^2 \quad sum\_xy = \sum x_i y_i$$

清单 14-6 展示了一种相关系数的 C++ 实现算法，函数 AvxPfpCorrCoefCpp 计算数据数组 x 和 y 之间的相关系数。注意这些数组必须按 32 字节边界对齐，另外要注意的是，数组元素总数 n 必须可以整除 4。这些限制是为了便于 x86-AVX 汇编语言函数进行 256 位宽的 394
组合算术运算。AvxPfpCorrCoefCpp 循环扫过数据数组，计算所需的和变量。把这些值保存到 sums 数组，同时计算中间值 rho_num 和 rho_den。在计算 rho 的最终值之前，函数会验证 rho_den 是否大于等于 CcEpsilon。

紧跟其序言之后，汇编语言函数 AvxPfpCorrCoef_（见清单 14-7）进行必要的数组对齐及大小验证。然后分别使用寄存器 YMM3 到 YMM7 初始化组合版本的 sum_x、sum_y、sum_xx、sum_yy 和 sum_xy 为 0。在函数主循环的每次迭代中，从数组 x 和 y 中取出相应的值使用双精度浮点运算更新上述四个和变量。

完成主循环后，函数必须减少组合和变量的宽度，得到一个最终结果。指令 vextractf128（提取组合浮点值）把每个 256 位宽组合和变量的高 128 位拷贝到 XMM 寄存器。接着函数使用 vaddpd 和 vhaddpd 指令计算每个和变量的最终值。图 14-1 描述了对变量 sum_x 采用这种处理方法的过程。最终的和变量存储到 sums 数组中，为 C++ 算法实现中的比较做好准备。

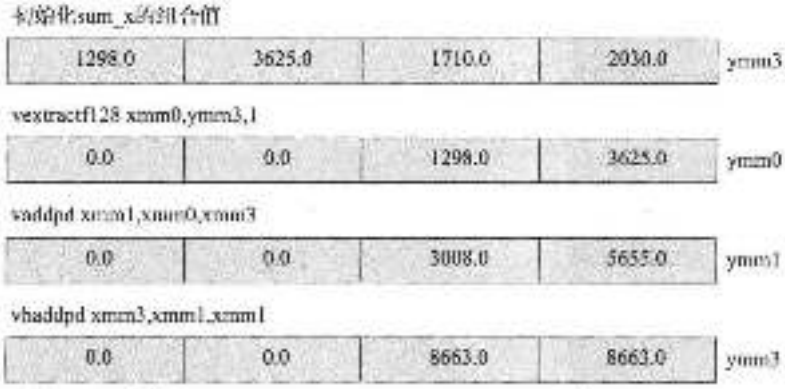


图 14-1 使用 vextractf128、vaddpd 和 vhaddpd 计算最终的 sum_x

而后使用普通的 x86-AVX 标量双精度浮点运算计算出 rho 的值。注意指令 vcvtsi2sd xmm2,xmm2,dword ptr [ebp+16] 需要两个源操作数，第二个源操作数指定的有符号双字整型数被转换为双精度浮点数。该指令也使得 des[127:64] = src1[127:64]（即目标操作数 [127:64]= 第一个源操作数 [127:64]）。另外，也要注意到在计算 rho 的最终值前，函数 AvxPfpCorrCoef_ 使用了指令 vcomisd 来确定 rho_den 大于等于 CcEpsilon。函数结语前还调用了 vzeroupper 指令。示例程序 AvxPackedFloatingPointCorrCoef 的执行结果见输出 14-3。 395

输出 14-3 示例程序 AvxPackedFloatingPointCorrCoef

```
Results for AvxPackedFloatingPointCorrCoef

rho1: 0.98083554  rho2: 0.98083554

sum_x:      1549166      1549166
sum_y:      1537789      1537789
```

```
sum_xx:  32934744842  32934744842
sum_yy:  32471286601  32471286601
sum_xy:  32532024390  32532024390
```

14.2.2 矩阵列均值

本章的最后一个示例程序名叫 AvxPackedFloatingPointColMeans，使用 x86-AVX 指令集对给定的双精度浮点数矩阵计算每列的算术平均值。程序的 C++ 和汇编语言源代码分别列在清单 14-8 和清单 14-9 中。

清单 14-8 AvxPackedFloatingPointColMeans.cpp

```
#include "stdafx.h"
#include <memory.h>
#include <stdlib.h>

extern "C" bool AvxPfpColMeans_(const double* x, int nrows, int ncols,↵
double* col_means);

bool AvxPfpColMeansCpp(const double* x, int nrows, int ncols, double*↵
col_means)
{
    // 确保 nrows 和 ncols 有效
    if ((nrows <= 0) || (ncols <= 0))
        return false;

    // 确保 col_means 正确对齐
    if (((uintptr_t)col_means & 0x1f) != 0)
        return false;

    // 计算列均值
    memset(col_means, 0, ncols * sizeof(double));

    for (int i = 0; i < nrows; i++)
    {
        for (int j = 0; j < ncols; j++)
            col_means[j] += x[i * ncols + j];
    }

    for (int j = 0; j < ncols; j++)
        col_means[j] /= nrows;

    return true;
}

int _tmain(int argc, _TCHAR* argv[])
{
    const int nrows = 13;
    const int ncols = 11;
    double* x = (double*)malloc(nrows * ncols * sizeof(double));
    double* col_means1 = (double*)_aligned_malloc(ncols * sizeof(double), 32);
    double* col_means2 = (double*)_aligned_malloc(ncols * sizeof(double), 32);

    srand(47);
    rand();

    for (int i = 0; i < nrows; i++)
    {
        for (int j = 0; j < ncols; j++)
```

```
            x[i * ncols + j] = rand() % 511;
    }

    bool rc1 = AvxPfpColMeansCpp(x, nrows, ncols, col_means1);
    bool rc2 = AvxPfpColMeans_(x, nrows, ncols, col_means2);

    printf("Results for sample program AvxPackedFloatingPointColMeans\n");

    if (rc1 != rc2)
    {
        printf("Bad return code (rc1 = %d, rc2 = %d)\n", rc1, rc2);
        return 1;
    }

    printf("\nTest Matrix\n");
    for (int i = 0; i < nrows; i++)
    {
        printf("row %2d: ", i);
        for (int j = 0; j < ncols; j++)
            printf("%5.0lf ", x[i * ncols + j]);
        printf("\n");
    }
    printf("\n");                                                    397
    for (int j = 0; j < ncols; j++)
    {
        printf("col_means1[%2d]: %12.4lf  ", j, col_means1[j]);
        printf("col_means2[%2d]: %12.4lf  ", j, col_means2[j]);
        printf("\n");
    }

    free(x);
    _aligned_free(col_means1);
    _aligned_free(col_means2);
    return 0;
}
```

清单 14-9 AvxPackedFloatingPointColMeans_.asm

```
        .model flat,c
        .code

; extern "C" bool AvxPfpColMeans_(const double* x, int nrows, int ncols,↩
double* col_means)
;
; 描述：本函数计算双精度浮点矩阵的每列数值的均值
;
; 需要：AVX

AvxPfpColMeans_ proc
        push ebp
        mov ebp,esp
        push ebx
        push esi
        push edi

; 加载并验证参数
        mov esi,[ebp+8]                 ;esi = 指向 x

        xor eax,eax
        mov edx,[ebp+12]                ;edx = nrows
        test edx,edx
```

```
        jle BadArg                      ;如果 nrows <= 0, 跳转

        mov ecx,[ebp+16]                ;ecx = ncols
        test ecx,ecx
        jle BadArg                      ;如果 ncols <= 0, 跳转

        mov edi,[ebp+20]                ;edi = 指向 col_means
        test edi,1fh
398     jnz BadArg                      ;如果 col_means 没有对齐, 则转

; 将 col_means 清零
        mov ebx,ecx                     ;ebx = ncols
        shl ecx,1                       ;ecx = col_means 中双字个数
        rep stosd                       ;将 col_means 清零

; 计算 x 中每列的和
LP1:    mov edi,[ebp+20]                ;edi = 指向 col_means
        xor ecx,ecx                     ;ecx = col_index

LP2:    mov eax,ecx                     ;eax = col_index
        add eax,4
        cmp eax,ebx                     ;还有 4 列或者更多列?
        jg  8F                          ;如果 col_index + 4 > ncols, 跳转

; 使用接下来的 4 列数据更新 col_means
        vmovupd ymm0,ymmword ptr [esi]  ;加载当前行中的下 4 列
        vaddpd ymm1,ymm0,ymmword ptr [edi] ;加至 col_means
        vmovapd ymmword ptr [edi],ymm1  ;保存更新后的 col_means
        add ecx,4                       ;col_index += 4
        add esi,32                      ;更新 x 指针
        add edi,32                      ;更新 col_means 指针
        jmp NextColSet

8:      sub eax,2
        cmp eax,ebx                     ;还有 2 列或者更多列?
        jg  8F                          ;如果 col_index + 2 > ncols, 则转

; 使用接下来的 2 列数据更新 col_means
        vmovupd xmm0,xmmword ptr [esi]  ;加载当前行中的下 2 列
        vaddpd xmm1,xmm0,xmmword ptr [edi] ;加至 col_means
        vmovapd xmmword ptr [edi],xmm1  ;保存更新后的 col_means
        add ecx,2                       ;col_index += 2
        add esi,16                      ;更新 x 指针
        add edi,16                      ;更新 col_means 指针
        jmp NextColSet

; 使用接下来的列(或当前行中的最后一列)数据更新 col_means
8:      vmovsd xmm0,real8 ptr [esi]     ;从最后一列中加载 x
        vaddsd xmm1,xmm0,real8 ptr [edi] ;加至 col_means
        vmovsd real8 ptr [edi],xmm1     ;保存更新后的 col_means
        add ecx,1                       ;col_index += 1
        add esi,8                       ;更新 x 指针

NextColSet:
        cmp ecx,ebx                     ;当前行中还有列数据?
        jl LP2                          ;如果是, 则转
        dec edx                         ;nrows -= 1
399     jnz LP1                         ;还有行数据没有处理, 跳转

; 计算最后的 col_means
        mov eax,[ebp+12]                ;eax = nrows
```

```
        vcvtsi2sd xmm2,xmm2,eax          ;xmm2 = DPFP nrows
        mov edx,[ebp+16]                 ;edx = ncols
        mov edi,[ebp+20]                 ;edi = ptr to col_means

@@:     vmovsd xmm0,real8 ptr [edi]      ;edi = 指向 col_means
        vdivsd xmm1,xmm0,xmm2            ;计算最后均值
        vmovsd real8 ptr [edi],xmm1      ;保存 col_mean[i]
        add edi,8                        ;更新 col_means 指针
        dec edx                          ;ncols -= 1
        jnz @@                           ;所有列处理完
        mov eax,1                        ;设置成功的返回码
        vzeroupper

BadArg: pop edi
        pop esi
        pop ebx
        pop ebp
        ret

AvxPfpColMeans_ endp
        end
```

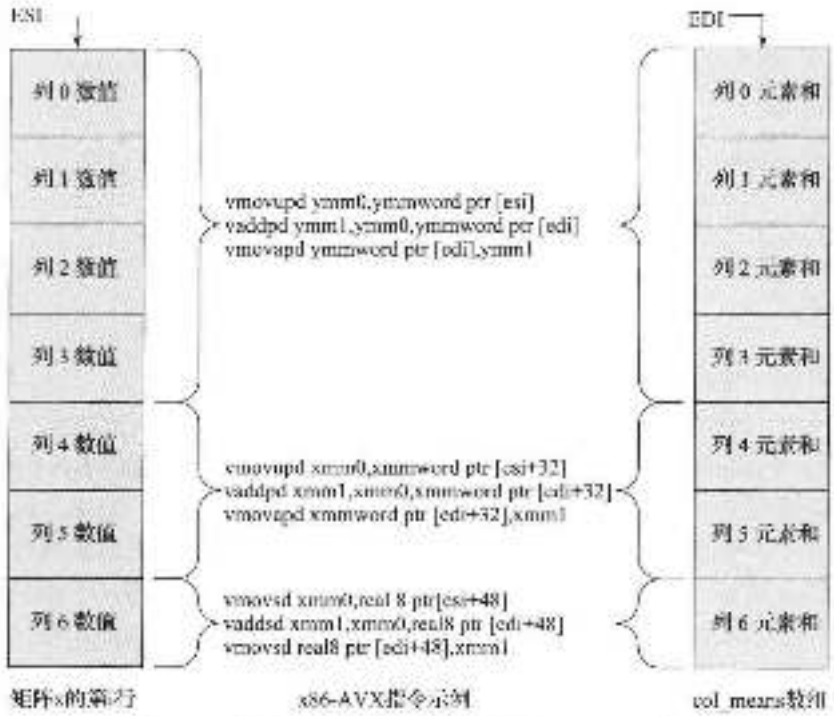
源程序 AvxPackedFloatingPointColMeans.cpp（见清单 14-8）的顶部声明了函数 AvxPfpColMeans，它使用 C++ 语言编写的简单算法来计算矩阵每列的均值。注意，此函数对输入的数组 x 不验证是否正确对齐。这样做的原因是，此算法必须可以处理任何行列的标准双精度浮点的 C++ 矩阵。回顾前面，C++ 矩阵中的元素以行主（row-major）排序的方式存储在一个连续的内存块上（见第 2 章），这意味着无法对指定的行、列或元素进行对齐。代码中对 col_means 数组进行是否正确对齐的检测，是为了让 x86-AVX 汇编语言函数可以使用 vmovapd 指令访问一维数组中的每个元素，而不用担心多行的问题。

函数 _tmain 分配和初始化含有测试值的矩阵。注意，malloc 和 _aligned_malloc 函数分别用来为矩阵 x 和数组 col_means 动态分配存储空间，相应的内存释放函数是在 _tmain 的末尾调用的。_tmain 中剩余的语句调用 C++ 和汇编语言列均值计算函数并把结果打印出来。

如同其相应的 C++ 函数一样，x86-AVX 汇编语言函数 AvxPfpColMeans_（见清单 14-9）检测参数的有效性。然后把 col_means 数组的每个元素都初始化为 0，它们将被用来计算矩阵列的和。清零动作由指令 rep stosd 来实现。为了提高工作效率，根据矩阵中的当前列索引和列中的元素数目，计算列和的循环采用了不同的 x86-AVX 数据转移和累加指令。比如，假设矩阵 x 有 7 列，x 中的前 4 列可以使用 256 位宽组合加法累加到 col_means 中，接下来 [400]
的两列使用 128 位宽组合加法累加到 col_means 中，最后一列元素则使用标量加法累加到 col_means 中。图 14-2 描述了这种方法的更多细节。

在列求和循环的顶端，标号 LP1 之后，是函数开始对矩阵 x 中每一行进行处理的起始点。进入第一个求和循环迭代前，寄存器 EDX 中包含 nrows，ESI 中包含指向 x 的指针。每个求和循环都以 mov edi,[ebp+20] 指令开始，这条指令把指向 col_means 的指针加载到 EDI 中。xor ecx,ecx 指令初始化 col_index 为 0。在标号 LP2 之后，函数使用一系列指令来确定在当前行中还有多少列数据没有被处理。如果剩余 4 列或者更多列，函数会使用操作 256 位宽组合操作数的指令将接下来的四个元素累加到 col_means 数组。指令 vmovupd ymm0,ymmword ptr [esi] 将四个元素从矩阵 x 加载到 YMM0（回顾一下，矩阵 x 的元素没有对齐到 16 或 32 字节边界）。接着，指令 vaddpd ymm1,ymm0,ymmword ptr [edi] 对在 YMM0

中的当前矩阵元素和相应的 col_means 元素求和。同时将更新后的和存储到 col_means 中，col_means 已经使用 vmovapd ymmword ptr [edi],ymm1 指令进行了正确对齐。而后更新寄存器 ECX、ESI 和 EDI，为计算下一个矩阵列元素集做好准备。



401 图 14-2 使用不同位宽的操作数更新 col_means 数组

上面描述的求和循环被反复执行，直到当前行中没有处理的列元素少于 4 为止。当这个条件满足之后，函数必须使用处理 128 位宽组合操作数或 64 位宽标量操作数的指令，对剩下的列（如果有的话）元素进行处理。如果还有 3 列元素需要处理，则上述两种指令都将被使用。函数针对上述场景，准备了相应的代码块进行处理。计算完列和之后，用 col_means 中的每个元素去除以 nrows，得到最后的列元素均值。输出 14-4 显示了示例程序 AvxPackedFloatingPointColMeans 的运行结果。

输出 14-4 示例程序 AvxPackedFloatingPointColMeans

```
Results for sample program AvxPackedFloatingPointColMeans

Test Matrix
row  0:   423   199   393    76   320    72   225    63   220   499    22
row  1:   311   277   174   359   189   380   509    95   449   210   324
row  2:   318   347   439   267   450   302   183   154   246   239   150
row  3:   360   508   466   274   402   240   327   442   365   291   353
row  4:   452   432   389   386   155   438   471    93   313   148   430
row  5:    76   341   341   329   388   315   336    36    75   328   224
row  6:   333   277   250   504    80   481    20   109   445   407   252
row  7:   202   131     6   338    49    41   144   428     3   240   145
row  8:   239   336   419   223   336   484   433   296   208   459   407
row  9:   398   501   208    24   475    75    30   246   462   436    36
row 10:   508   161   291   503   386   352   492   226   291   258   276
row 11:    53   499   132   339    26   346   422   359   292   411    62
```

```
row 12:    7   230   301    16   160    71   109   479   166   417    85

col_means1[ 0]:     252.3077  col_means2[ 0]:     252.3077
col_means1[ 1]:     323.0000  col_means2[ 1]:     323.0000
col_means1[ 2]:     293.0000  col_means2[ 2]:     293.0000
col_means1[ 3]:     280.6154  col_means2[ 3]:     280.6154
col_means1[ 4]:     262.7692  col_means2[ 4]:     262.7692
col_means1[ 5]:     268.7692  col_means2[ 5]:     268.7692
col_means1[ 6]:     284.6154  col_means2[ 6]:     284.6154
col_means1[ 7]:     216.6154  col_means2[ 7]:     216.6154
col_means1[ 8]:     271.8462  col_means2[ 8]:     271.8462
col_means1[ 9]:     334.0769  col_means2[ 9]:     334.0769
col_means1[10]:     212.7692  col_means2[10]:     212.7692
```

402

14.3 总结

本章集中介绍了 x86-AVX 的组合浮点功能。在这一章里，我们学习了使用 256 位宽组合浮点操作数进行基本的运算，还通过一系列示例代码学习了浮点数组和矩阵的 SIMD 处理技术。从这些示例程序中，我们又一次感受到了 x86-AVX 指令集的优势，它减少了大部分寄存器到寄存器的数据传输操作，同时使得我们可以更简单地进行汇编语言编程。在下一章学习使用 x86-AVX 的组合整型资源的时候，仍然可以看到这个优势。

403
404

x86-AVX 组合整型编程

本章将演示如何使用 x86-AVX 指令集对 256 位组合整型操作数进行各种运算。我们将通过若干示例程序演示如何执行基本的组合整型运算和解组操作，还准备了几个示例程序针对 8 位无符号整型数执行通用的图像处理算法。本章中所有的示例程序必须在支持 AVX2 的处理器和操作系统上才能运行。

15.1 组合整型基础

本节将演示如何使用 x86-AVX 指令集执行组合整型操作。第一个示例程序演示 256 位宽操作数的基本组合整型运算，第二个示例程序演示使用 YMM 寄存器执行解组和组合操作，特别演示了对分开的两个 128 位位区执行操作。

15.1.1 组合整型运算

第一个示例程序 AvxPackedIntegerArithmetic 演示了如何对 256 位宽的组合整型操作数执行基本运算，同时展示了 x86-AVX 和 x86-SSE 在执行组合整型运算时的差异。此程序的
405 C++ 和 x86-AVX II 编语言源代码分别列在清单 15-1 和清单 15-2 中。

清单 15-1 AvxPackedIntegerArithmetic.cpp

```
#include "stdafx.h"
#include "YmmVal.h"

extern "C" void AvxPiI16_(YmmVal* a, YmmVal* b, YmmVal c[6]);
extern "C" void AvxPiI32_(YmmVal* a, YmmVal* b, YmmVal c[5]);

void AvxPiI16(void)
{
    __declspec(align(32))  YmmVal a;
    __declspec(align(32))  YmmVal b;
    __declspec(align(32))  YmmVal c[6];

    a.i16[0] = 10;       b.i16[0] = 1000;
    a.i16[1] = 20;       b.i16[1] = 2000;
    a.i16[2] = 3000;     b.i16[2] = 30;
    a.i16[3] = 4000;     b.i16[3] = 40;

    a.i16[4] = 30000;    b.i16[4] = 3000;      // 加法溢出
    a.i16[5] = 6000;     b.i16[5] = 32000;     // 加法溢出
    a.i16[6] = 2000;     b.i16[6] = -31000;    // 减法溢出
    a.i16[7] = 4000;     b.i16[7] = -30000;    // 减法溢出

    a.i16[8]  = 4000;    b.i16[8]  = -2500;
    a.i16[9]  = 3600;    b.i16[9]  = -1200;
    a.i16[10] = 6000;    b.i16[10] = 9000;
    a.i16[11] = -20000;  b.i16[11] = -20000;

    a.i16[12] = -25000;  b.i16[12] = -27000;   // 加法溢出
    a.i16[13] = 8000;    b.i16[13] = 28700;    // 加法溢出
    a.i16[14] = 5;       b.i16[14] = -32766;   // 减法溢出
```

```
    a.i16[15] = -15000;  b.i16[15] = 24000;     // 减法溢出

    AvxPiI16_(&a, &b, c);

    printf("\nResults for AvxPiI16()\n\n");
    printf("i       a       b   vpaddw vpaddsw vpsubw vpsubsw vpminsw
vpmaxsw\n");
    printf("-------------------------------------------------------------\n");

    for (int i = 0; i < 16; i++)
    {
        const char* fs = "%7d ";

        printf("%2d ", i);
        printf(fs, a.i16[i]);
        printf(fs, b.i16[i]);
        printf(fs, c[0].i16[i]);
        printf(fs, c[1].i16[i]);
        printf(fs, c[2].i16[i]);
        printf(fs, c[3].i16[i]);
        printf(fs, c[4].i16[i]);
        printf(fs, c[5].i16[i]);
        printf("\n");
    }
}

void AvxPiI32(void)
{
    __declspec(align(32))  YmmVal a;
    __declspec(align(32))  YmmVal b;
    __declspec(align(32))  YmmVal c[5];

    a.i32[0] = 64;        b.i32[0] = 4;
    a.i32[1] = 1024;      b.i32[1] = 5;
    a.i32[2] = -2048;     b.i32[2] = 2;
    a.i32[3] = 8192;      b.i32[3] = 5;
    a.i32[4] = -256;      b.i32[4] = 8;
    a.i32[5] = 4096;      b.i32[5] = 7;
    a.i32[6] = 16;        b.i32[6] = 3;
    a.i32[7] = 512;       b.i32[7] = 6;

    AvxPiI32_(&a, &b, c);

    printf("\nResults for AvxPiI32()\n\n");
    printf("i       a       b    vphaddd vphsubd vpmulld vpsllvd
vpsravd\n");
    printf("-------------------------------------------------------------\n");

    for (int i = 0; i < 8; i++)
    {
        const char* fs = "%8d ";

        printf("%2d ", i);
        printf(fs, a.i32[i]);
        printf(fs, b.i32[i]);
        printf(fs, c[0].i32[i]);
        printf(fs, c[1].i32[i]);
        printf(fs, c[2].i32[i]);
        printf(fs, c[3].i32[i]);
        printf(fs, c[4].i32[i]);
        printf("\n");
    }
}
```

406

407

```
int _tmain(int argc, _TCHAR* argv[])
{
    AvxPiI16();
    AvxPiI32();
    return 0;
}
```

清单 15-2 AvxPackedIntegerArithmetic_asm

```
        .model flat,c
        .code

; extern "C" void AvxPiI16_(YmmVal* a, YmmVal* b, YmmVal c[6]);
;
; 描述：本函数演示多个组合 16 位整型运算指令，使用的是 256 位寄存器操作数
;
; 需要：AVX2

AvxPiI16_ proc
        push ebp
        mov ebp,esp

; 加载参数值
        mov eax,[ebp+8]                 ;eax = ptr to a
        mov ecx,[ebp+12]                ;ecx = ptr to b
        mov edx,[ebp+16]                ;edx = ptr to c

; 加载 a 和 b，必须正确对齐
        vmovdqa ymm0,ymmword ptr [eax]  ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [ecx]  ;ymm1 = b

; 执行组合运算
        vpaddw ymm2,ymm0,ymm1           ;加
        vpaddsw ymm3,ymm0,ymm1          ;与有符号饱和数相加（即与上限或下限相加）
        vpsubw ymm4,ymm0,ymm1           ;减
        vpsubsw ymm5,ymm0,ymm1          ;与有符号饱和数相减
        vpminsw ymm6,ymm0,ymm1          ;有符号最小
        vpmaxsw ymm7,ymm0,ymm1          ;有符号最大

; 保存结果
        vmovdqa ymmword ptr [edx],ymm2      ;保存 vpaddw 结果
        vmovdqa ymmword ptr [edx+32],ymm3   ;保存 vpaddsw 结果
        vmovdqa ymmword ptr [edx+64],ymm4   ;保存 vpsubw 结果
        vmovdqa ymmword ptr [edx+96],ymm5   ;保存 vpsubsw 结果
        vmovdqa ymmword ptr [edx+128],ymm6  ;保存 vpminsw 结果
        vmovdqa ymmword ptr [edx+160],ymm7  ;保存 vpmaxsw 结果

        vzeroupper
        pop ebp
        ret
AvxPiI16_ endp

; extern "C" void AvxPiI32_(YmmVal* a, YmmVal* b, YmmVal c[5]);
; 描述：本函数演示多个组合 32 位整型运算指令，使用的是 256 位寄存器操作数
;
; 需要：AVX2
; Requires:     AVX2

AvxPiI32_ proc
        push ebp
        mov ebp,esp
```

```
; 加载参数值
        mov eax,[ebp+8]                 ;eax = ptr to a
        mov ecx,[ebp+12]                ;ecx = ptr to b
        mov edx,[ebp+16]                ;edx = ptr to c

; 加载 a 和 b，必须正确对齐
        vmovdqa ymm0,ymmword ptr [eax]  ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [ecx]  ;ymm1 = b

; 执行组合算术运算
        vphaddd ymm2,ymm0,ymm1          ;水平加法
        vphsubd ymm3,ymm0,ymm1          ;水平减法
        vpmulld ymm4,ymm0,ymm1          ;有符号乘法（低 32 位）
        vpsllvd ymm5,ymm0,ymm1          ;逻辑左移
        vpsravd ymm6,ymm0,ymm1          ;算术右移

; 保存结果
        vmovdqa ymmword ptr [edx],ymm2      ;保存 vphaddd 结果
        vmovdqa ymmword ptr [edx+32],ymm3   ;保存 vphsubd 结果
        vmovdqa ymmword ptr [edx+64],ymm4   ;保存 vpmulld 结果
        vmovdqa ymmword ptr [edx+96],ymm5   ;保存 vpsllvd 结果
        vmovdqa ymmword ptr [edx+128],ymm6  ;保存 vpsravd 结果

        vzeroupper
        pop ebp
        ret
AvxPiI32_ endp
        end
```

409

在 C++ 源文件 AvxPackedIntegerArithmetic.cpp（见清单 15-1）的函数 AvxPiI16 中，首先使用 16 位有符号整型数初始化若干 YmmVal 变量，接着调用汇编语言函数 AvxPiI16_。AvxPiI16_ 执行了一系列的通用组合运算，包含加、减、有符号最小值和有符号最大值，这些操作的执行结果用几个 printf 语句打印了出来。在 AvxPackedIntegerArithmetic.cpp 的另外一个函数 AvxPiI32 中，先准备了若干 32 位有符号整型数初始化 YmmVal 实例，然后传给汇编语言函数 AvcPiI32_ 执行水平加法、减法、乘法和变量移位操作。

在函数序言之后，AvxPiI16_ 把参数 a、b、c 的指针分别加载到寄存器 EAX、ECX 和 EDX。指令 vmovdqa ymm0,ymmword ptr [eax] 把变量 a 加载到寄存器 YMM0。注意 vmovdqa 指令要求任一 256 位宽的内存操作数都是按 32 字节边界对齐的（vmovdqu 指令可以在未对齐的操作数下使用）。另一条 vmovdqa 指令把 b 加载到 YMM1。接下来是一系列的运算指令，包括组合有符号整型加法（vpaddw 和 vpaddsw）、减法（vpsubw 和 vpsubsw）、有符号最小值（vpminsw）以及有符号最大值（vpmaxsw）。计算的结果存储到数组 c，它也必须被正确对齐。

函数 AvxPiI32_ 使用和 AvxPiI16_ 相同的指令序列将 a 和 b 加载到寄存器 YMM0 和 YMM1。然后执行几个通用的运算操作，包括水平加法（vphaddd）、水平减法（vphsubd）和有符号 32 位乘法（vpmulld）。AvxPiI32_ 还演示了位移指令的用法，包括 vpsllvd（变量位逻辑左移）和 vpsravd（变量位算术右移）。这些指令在 AVX2 中首次出现，如图 15-1 所描述，对第一个源操作数的双字元素进行移位，移动的位数在相应的第二个源操作数中给定。注意，AvxPiI16_ 和 AvxPiI32_ 在函数结语前都调用了 vzeroupper 指令。输出 15-1 显示了示例程序 AvxPackedIntegerArithmetic 的执行结果。

410

vpsllvd ymm2,ymm0,ymm1

512	16	4096	-256	8192	-2048	1024	64	ymm0
6	3	7	8	5	2	5	4	ymm1
32768	128	524288	-65536	262144	-8192	32768	1024	ymm2

vpsravd ymm2,ymm0,ymm1

512	16	4096	-256	8192	-2048	1024	64	ymm0
6	3	7	8	5	2	5	4	ymm1
8	2	32	-1	256	-512	32	4	ymm2

图 15-1 指令 vpsllvd 和 vpsravd 的执行

输出 15-1 示例程序 AvxPackedIntegerArithmetic

```
Results for AvxPiI16()

i       a       b   vpaddw vpaddsw  vpsubw vpsubsw  vpminsw vpmaxsw
-----------------------------------------------------------------
0      10    1000     1010    1010    -990    -990       10    1000
1      20    2000     2020    2020   -1980   -1980       20    2000
2    3000      30     3030    3030    2970    2970       30    3000
3    4000      40     4040    4040    3960    3960       40    4000
4   30000    3000   -32536   32767   27000   27000     3000   30000
5    6000   32000   -27536   32767  -26000  -26000     6000   32000
6    2000  -31000   -29000  -29000  -32536   32767  -31000    2000
7    4000  -30000   -26000  -26000  -31536   32767  -30000    4000
8    4000   -2500     1500    1500    6500    6500   -2500    4000
9    3600   -1200     2400    2400    4800    4800   -1200    3600
10   6000    9000    15000   15000   -3000   -3000    6000    9000
11 -20000  -20000    25536  -32768       0       0  -20000  -20000
12 -25000  -27000    13536  -32768    2000    2000  -27000  -25000
13   8000   28700   -28836   32767  -20700  -20700    8000   28700
14      3  -32766   -32763  -32763  -32767   32767  -32766       3
15 -15000   24000     9000    9000   26536  -32768  -15000   24000

Results for AvxPiI32()

i       a       b   vphaddd  vphsubd   vpmulld  vpsllvd  vpsravd
-----------------------------------------------------------------
0      64       4     1088     -960       256     1024        4
1    1024       5     6144   -10240      5120    32768       32
2   -2048       2        9       -1     -4096    -8192     -512
3    8192       5        7       -3     40960   262144      256
4    -256       8     3840    -4352     -2048   -65536       -1
5    4096       7      528     -496     28672   524288       32
6      16       3       13        1        48      128        2
7     512       6        9       -3      3072    32768        8
```

15.1.2 组合整数解组操作

类似于 MMX 和 x86-SSE，x86-AVX 指令集也支持对各种字长元素的数据进行解组操作。

在下一个示例程序 AvxPackedIntegerUnpack 中，我们将学习如何对 256 位宽的操作数进行双字到四字的解组操作，以及如何使用有符号饱和数算法将 256 位宽的双字操作数组合成字。清单 15-3 和清单 15-4 分别列出了示例程序 AvxPackedIntegerUnpack 的 C++ 和汇编语言源代码。

清单 15-3 AvxPackedIntegerUnpack.cpp

```
#include "stdafx.h"
#include "YmmVal.h"

extern "C" void AvxPiUnpackDQ (YmmVal* a, YmmVal* b, YmmVal c[2]);
extern "C" void AvxPiPackDW (YmmVal* a, YmmVal* b, YmmVal* c);

void AvxPiUnpackDQ(void)
{
    __declspec(align(32))  YmmVal a;
    __declspec(align(32))  YmmVal b;
    __declspec(align(32))  YmmVal c[2];

    a.i32[0] = 0x00000000;  b.i32[0] = 0x88888888;
    a.i32[1] = 0x11111111;  b.i32[1] = 0x99999999;
    a.i32[2] = 0x22222222;  b.i32[2] = 0xaaaaaaaa;
    a.i32[3] = 0x33333333;  b.i32[3] = 0xbbbbbbbb;

    a.i32[4] = 0x44444444;  b.i32[4] = 0xcccccccc;
    a.i32[5] = 0x55555555;  b.i32[5] = 0xdddddddd;
    a.i32[6] = 0x66666666;  b.i32[6] = 0xeeeeeeee;
    a.i32[7] = 0x77777777;  b.i32[7] = 0xffffffff;

    AvxPiUnpackDQ_(&a, &b, c);                                    412
    printf("\nResults for AvxPiUnpackDQ()\n\n");
    printf("i   a           b           vpunpckldq  vpunpckhdq\n");
    printf("-------------------------------------------------\n");

    for (int i = 0; i < 8; i++)
    {
        const char* fs = "0x%08X  ";

        printf("%-2d  ", i);
        printf(fs, a.u32[i]);
        printf(fs, b.u32[i]);
        printf(fs, c[0].u32[i]);
        printf(fs, c[1].u32[i]);
        printf("\n");
    }
}

void AvxPiPackDW(void)
{
    char buff[256];
    __declspec(align(32))  YmmVal a;
    __declspec(align(32))  YmmVal b;
    __declspec(align(32))  YmmVal c;

    a.i32[0] = 10;          b.i32[0] = 32768;
    a.i32[1] = -200000;     b.i32[1] = 5500;
    a.i32[2] = 300000;      b.i32[2] = 42000;
    a.i32[3] = -4000;       b.i32[3] = -68000;

    a.i32[4] = 9000;        b.i32[4] = 25000;
    a.i32[5] = 80000;       b.i32[5] = 500000;
    a.i32[6] = 200;         b.i32[6] = -7000;
```

```
    a.i32[7] = -32769;      b.i32[7] = 12500;

    AvxPiPackDW (&a, &b, &c);
    printf("\nResults for AvxPiPackDW()\n\n");

    printf("a lo %s\n", a.ToString_i32(buff, sizeof(buff), false));
    printf("a hi %s\n", a.ToString_i32(buff, sizeof(buff), true));
    printf("\n");

    printf("b lo %s\n", b.ToString_i32(buff, sizeof(buff), false));
    printf("b hi %s\n", b.ToString_i32(buff, sizeof(buff), true));
    printf("\n");

    printf("c lo %s\n", c.ToString_i16(buff, sizeof(buff), false));
    printf("c hi %s\n", c.ToString_i16(buff, sizeof(buff), true));
413 }

int _tmain(int argc, _TCHAR* argv[])
{
    AvxPiUnpackDQ();
    AvxPiPackDW();
    return 0;
}
```

清单 15-4 AvxPackedIntegerUnpack_.asm

```
        .model flat,c
        .code

; extern "C" void AvxPiUnpackDQ_(YmmVal* a, YmmVal* b, YmmVal c[2]);
;
; 描述：本函数使用 256 位宽操作数演示 vpunpckldq 和 vpunpckhdq 指令的用法
;
; 需要：AVX2

AvxPiUnpackDQ_ proc
        push ebp
        mov ebp,esp

; 加载参数值
        mov eax,[ebp+8]                 ;指向 a 的指针
        mov ecx,[ebp+12]                ;指向 b 的指针
        mov edx,[ebp+16]                ;指向 c 的指针
        vmovdqa ymm0,ymmword ptr [eax]  ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [ecx]  ;ymm1 = b

; 执行双字到四字的解组操作
        vpunpckldq ymm2,ymm0,ymm1       ;解组低双字部分
        vpunpckhdq ymm3,ymm0,ymm1       ;解组高双字部分
        vmovdqa ymmword ptr [edx],ymm2  ;保存低位部分结果
        vmovdqa ymmword ptr [edx+32],ymm3 ;保存高位部分结果

        vzeroupper
        pop ebp
        ret
AvxPiUnpackDQ_ endp

; extern "C" void AviPiPackDW_(YmmVal* a, YmmVal* b, YmmVal* c);
;
; 描述：本函数使用 256 位宽操作数演示 vpackssdw 指令的用法
;
414 ; 需要：AVX2
```

```
AvxPiPackDW_ proc
        push ebp
        mov ebp,esp

; 加载参数值
        mov eax,[ebp+8]                 ;指向 a 的指针
        mov ecx,[ebp+12]                ;指向 b 的指针
        mov edx,[ebp+16]                ;指向 c 的指针
        vmovdqa ymm0,ymmword ptr [eax]  ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [ecx]  ;ymm1 = b

; 使用有符号饱和算术双字组合为字
        vpackssdw ymm2,ymm0,ymm1        ;ymm2 = 组合字
        vmovdqa ymmword ptr [edx],ymm2  ;保存结果

        vzeroupper
        pop ebp
        ret
AvxPiPackDW_ endp
        end
```

在清单 15-3 的开始部分，函数 AvxPiUnpackDQ 使用双字测试值初始化 YmmVal 实例 a 和 b，然后调用汇编语言函数 AvxPiUnpackDQ_，执行 x86-AVX 解压指令 vpunpckldq 和 vpunpckhdq。执行的结果使用了一个简单的循环和几个 printf 语句打印出来。在 C++ 代码中还包含函数 AvxPiPackDW，它初始化两个 YmmVal 变量，同时调用 AvxPiPackDW_，主要用来演示 vpackssdw（使用有符号饱和数进行双字到字的组合）指令的用法。

汇编语言文件 AvxPackedIntegerUnpack .asm（见清单 15-4）包含函数 AvxPiUnpackDQ_ 和 AvxPiPackDW_ 的定义。在函数 AvxPiUnpackDQ_ 中，首先将参数值 a 和 b 分别加载到寄存器 YMM0 和 YMM1，然后执行 vpunpckldq 和 vpunpckhdq 指令，并将结果保存到数组 c 中。回顾第 12 章学习的内容，许多 256 位宽 x86-AVX 指令执行操作的时候，使用两个相对独立的 128 位的位区。图 15-2 中显示了在这一原则下 vpunpckldq 和 vpunpckhdq 指令执行的详细细节，这些指令执行两个独立的解组操作：一个使用寄存器位 255:128（高位区），

415

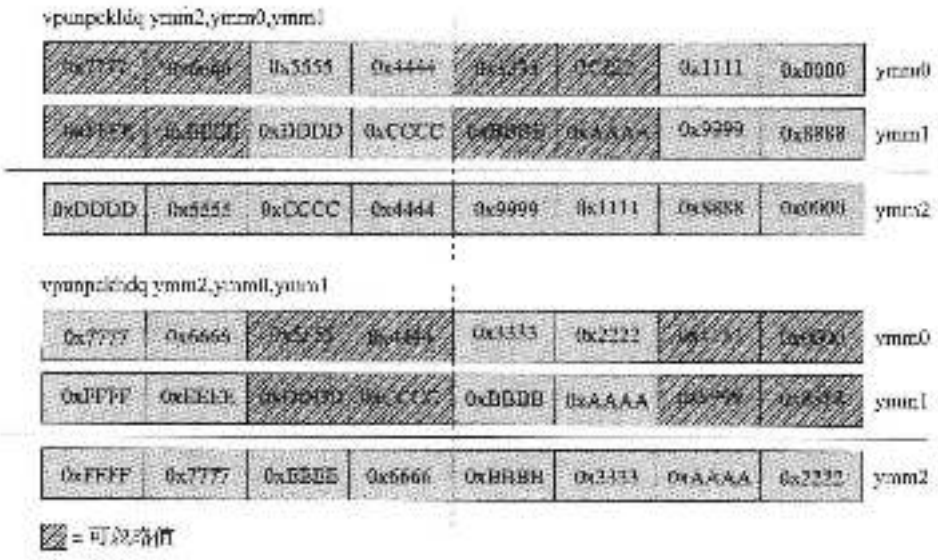


图 15-2 vpunpckldq 和 vpunpckhdq 指令的执行演示

另一个使用寄存器位 127:0（低位区）。最后，在函数 AvxPiUnpackDQ_ 中使用了 YMM 寄存器，为了预防潜在的性能问题，在函数结束前必须调用 vzeroupper 指令。

汇编语言函数 AvxPiPackDW_ 将参数值 a 和 b 分别加载到寄存器 YMM0 和 YMM1。指令 vpackssdw ymm2,ymm0,ymm1 使用有符号饱和数将 YMM0 和 YMM1 中的组合有符号双字整数转换成组合有符号字整数，并将结果保存到 YMM2。图 15-3 显示了 vpackssdw 指令执行的详细细节。输出 15-2 给出了示例程序 AvxPackedIntegerUnpack 的运行结果。

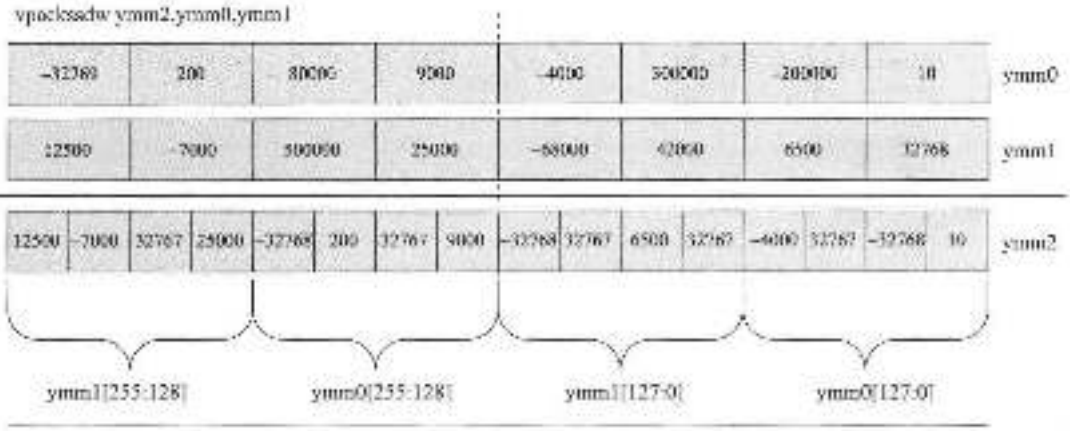


图 15-3 vpackssdw 指令的执行演示

416

输出 15-2 示例程序 AvxPackedIntegerUnpack

```
Results for AvxPiUnpackDQ()

i   a           b           vpunpckldq  vpunpckhdq
-------------------------------------------------
0   0x0000      0x8888      0x0000      0x2222
1   0x1111      0x9999      0x8888      0xAAAA
2   0x2222      0xAAAA      0x1111      0x3333
3   0x3333      0xBBBB      0x9999      0xBBBB
4   0x4444      0xCCCC      0x4444      0x6666
5   0x5555      0xDDDD      0xCCCC      0xEEEE
6   0x6666      0xEEEE      0x5555      0x7777
7   0x7777      0xFFFF      0xDDDD      0xFFFF

Results for AvxPiPackDW()

a lo          10      -200000 |       300000       -4000
a hi        9000       80000 |          200      -32769

b lo       32768        6500 |        42000      -68000
b hi       25000      500000 |        -7000       12500

c lo   10 -32768   32767   -4000 |   32767    6500   32767  -32768
c hi 9000  32767     200  -32768 |   25000   32767   -7000   12500
```

15.2 高级编程

本节的示例程序着重介绍使用 x86 AVX 组合器型资源的高级编程技巧。第一个示例程

序使用 x86-AVX 指令集实现像素裁剪算法。第二个示例程序是第 10 章中图像阈值算法的 x86-AVX 实现。在两个例子中，除了描述 x86-SSE 和 x86-AVX 的一些差异外，还演示了如何使用 AVX2 中一些新的组合整型指令。

15.2.1 图像像素裁剪

像素裁剪是一种图像处理技术，对处于两个阈值间的每个像素，限定其亮度值。这种技术常用来消除图像中极亮和极暗的像素，以减少其动态范围。本小节中的示例程序 AvxPackedIntegerPixelClip 演示如何使用 x86-AVX 指令集裁剪 8 位灰度图像。其相关的源代码见清单 15-5、清单 15-6 和清单 15-7。

417

清单 15-5 AvxPackedIntegerPixelClip.h

```
#pragma once

#include "VscDefs.h"

// 此结构必须与 AvxPackedIntegerPixelClip_.asm 中声明的结构一致
typedef struct
{
    Uint8* Src;                 // 源缓冲区
    Uint8* Des;                 // 目标缓冲区
    Uint32 NumPixels;           // 像素个数
    Uint32 NumClippedPixels;    // 裁剪的像素个数
    Uint8 ThreshLo;             // 低阈值
    Uint8 ThreshHi;             // 高阈值
} PcData;

// 函数定义在 AvxPackedIntegerPixelClip.cpp 中
bool AvxPiPixelClipCpp(PcData* pc_data);

// 函数定义在 AvxPackedIntegerPixelClip_.asm 中
extern "C" bool AvxPiPixelClip_(PcData* pc_data);

// 函数定义在 AvxPackedIntegerPixelClipTimed.cpp 中
void AvxPackedIntegerPixelClipTimed(void);
```

清单 15-6 AvxPackedIntegerPixelClip.cpp

```
#include "stdafx.h"
#include "AvxPackedIntegerPixelClip.h"
#include <malloc.h>
#include <memory.h>
#include <stdlib.h>

bool AvxPiPixelClipCpp(PcData* pc_data)
{
    Uint32 num_pixels = pc_data->NumPixels;
    Uint8* src = pc_data->Src;
    Uint8* des = pc_data->Des;

    if ((num_pixels < 32) || ((num_pixels & 0x1f) != 0))
        return false;

    if (((uintptr_t)src & 0x1f) != 0)
        return false;
    if (((uintptr_t)des & 0x1f) != 0)
        return false;
```

418

```
    Uint8 thresh_lo = pc_data->ThreshLo;
    Uint8 thresh_hi = pc_data->ThreshHi;
    Uint32 num_clipped_pixels = 0;

    for (Uint32 i = 0; i < num_pixels; i++)
    {
        Uint8 pixel = src[i];

        if (pixel < thresh_lo)
        {
            des[i] = thresh_lo;
            num_clipped_pixels++;
        }
        else if (pixel > thresh_hi)
        {
            des[i] = thresh_hi;
            num_clipped_pixels++;
        }
        else
            des[i] = src[i];
    }

    pc_data->NumClippedPixels = num_clipped_pixels;
    return true;
}

void AvxPackedIntegerPixelClip(void)
{
    const Uint8 thresh_lo = 10;
    const Uint8 thresh_hi = 245;
    const Uint32 num_pixels = 4 * 1024 * 1024;
    Uint8* src = (Uint8*)_aligned_malloc(num_pixels, 32);
    Uint8* des1 = (Uint8*)_aligned_malloc(num_pixels, 32);
    Uint8* des2 = (Uint8*)_aligned_malloc(num_pixels, 32);

    srand(157);
    for (int i = 0; i < num_pixels; i++)
    src[i] = (Uint8)(rand() % 256);

    PcData pc_data1;
    PcData pc_data2;

    pc_data1.Src = pc_data2.Src = src;
    pc_data1.Des = des1;
    pc_data2.Des = des2;
    pc_data1.NumPixels = pc_data2.NumPixels = num_pixels;
    pc_data1.ThreshLo = pc_data2.ThreshLo = thresh_lo;
419 pc_data1.ThreshHi = pc_data2.ThreshHi = thresh_hi;
    AvxPiPixelClipCpp(&pc_data1);
    AvxPiPixelClip_(&pc_data2);

    printf("Results for AvxPackedIntegerPixelClip\n");

    if (pc_data1.NumClippedPixels != pc_data2.NumClippedPixels)
        printf("  NumClippedPixels compare error!\n");

    printf("  NumClippedPixels1: %u\n", pc_data1.NumClippedPixels);
    printf("  NumClippedPixels2: %u\n", pc_data2.NumClippedPixels);

    if (memcmp(des1, des2, num_pixels) == 0)
        printf("  Destination buffer memory compare passed\n");
    else
```

```
        printf("  Destination buffer memory compare failed!\n");

    _aligned_free(src);
    _aligned_free(des1);
    _aligned_free(des2);
}

int _tmain(int argc, _TCHAR* argv[])
{
    AvxPackedIntegerPixelClip();
    AvxPackedIntegerPixelClipTimed();
    return 0;
}
```

清单 15-7 AvxPackedIntegerPixelClip_asm

```
        .model flat,c

; 此结构必须与 AvxPackedIntegerPixelClip.h 中声明的结构一致

PcData              struct
Src                 dword ?             ;源缓冲区
Des                 dword ?             ;目标缓冲区
NumPixels           dword ?             ;像素个数
NumClippedPixels    dword ?             ;裁剪的像素个数
ThreshLo            byte ?              ;低阈值
ThreshHi            byte ?              ;高阈值
PcData              ends

; 常量区域(自定义的段)
PcConstVals segment readonly align(32) public

PixelScale  byte 32 dup(80h)            ;像素从 Uint8 转换为 Int8 的调节值

; 下面定义的值主要是为了演示
; 并查对齐指令 align 32 在 .const 段中无法使用
Test1       dword 10
Test2       qword -20
            align 32
Test3       byte 32 dup(7fh)
PcConstVals ends
            .code

; extern "C" bool AvxPiPixelClip_(PcData* pc_data);
;
; 描述：本函数裁剪图像缓冲区中值在 ThreshLo 和 ThreshHi 间的像素
;
; 需要：AVX2, POPCNT

AvxPiPixelClip_ proc
        push ebp
        mov ebp,esp
        push ebx
        push esi
        push edi

; 加载并验证参数
        xor eax,eax
        mov ebx,[ebp+8]                 ;ebx = pc_data
        mov ecx,[ebx+PcData.NumPixels]  ;ecx = num_pixels
        cmp ecx,32
        jl BadArg                       ;如果 num_pixels < 32 则跳转
```

420

```
        test ecx,1fh
        jnz BadArg                      ;如果 num_pixels % 32 != 0 则跳转

        mov esi,[ebx+PcData.Src]        ;esi = Src
        test esi,1fh
        jnz BadArg                      ;如果 Src 没有对齐则跳转

        mov edi,[ebx+PcData.Des]        ;edi = Des
        test edi,1fh
        jnz BadArg                      ;如果 Des 没有对齐则跳转

; 创建包含 thresh_lo 和 thresh_hi 数据值
        vmovdqa ymm5,ymmword ptr [Pixel5cale]

        vpbroadcastb ymm0,[ebx+PcData.ThreshLo]     ;ymm0 = thresh_lo
        vpbroadcastb ymm1,[ebx+PcData.ThreshHi]     ;ymm1 = thresh_hi

        vpsubb ymm6,ymm0,ymm5           ;ymm6 = 设置 thresh_lo
421     vpsubb ymm7,ymm1,ymm5           ;ymm7 = 设置 thresh_hi

        xor edx,edx                     ;edx = num_clipped_pixels
        shr ecx,5                       ;ecx = 32字节块的数目

; 扫描图像缓冲区，并钳位裁剪像素
@@:     vmovdqa ymm0,ymmword ptr [esi]  ;ymm0 = 未调节像素
        vpsubb ymm0,ymm0,ymm5           ;ymm0 = 已调节像素

        vpcmpgtb ymm1,ymm0,ymm7         ;像素值大于 thresh_hi 的掩码
        vpand ymm2,ymm1,ymm7            ;大于阈值的像素的新值

        vpcmpgtb ymm3,ymm6,ymm0         ;像素值小于 thresh_lo 的掩码 thresh_lo
        vpand ymm4,ymm3,ymm6            ;小于阈值的像素的新值

        vpor ymm1,ymm1,ymm3             ;所有裁剪像素的掩码

        vpor ymm2,ymm2,ymm4             ;裁剪像素
        vpandn ymm3,ymm1,ymm0           ;未裁剪像素

        vpor ymm4,ymm3,ymm2             ;最终调节的裁剪像素
        vpaddb ymm4,ymm4,ymm5           ;最终未调节的裁剪像素

        vmovdqa ymmword ptr [edi],ymm4  ;保存裁剪像素

; 更新 num_clipped_pixels
        vpmovmskb eax,ymm1              ;eax = 裁剪像素掩码
        popcnt eax,eax                  ;计算裁剪像素数目
        add edx,eax                     ;更新 num_clipped_pixels
        add esi,32
        add edi,32
        dec ecx
        jnz @@

; 保存 num_clipped_pixels
        mov eax,1                       ;设置成功返回码
        mov [ebx+PcData.NumClippedPixels],edx
        vzeroupper

BadArg: pop edi
        pop esi
        pop ebx
        pop ebp
        ret
```

```
AvxPiPixelClip_ endp
        end
```

C++ 头文件 AvxPackedIntegerPixelClip.h（见清单 15-5）声明了一个名为 PcData 的结构体，这个结构体与其汇编语言的等价定义共用来维护裁剪算法用的各个数据项。代码文件 AvxPackedInteger-PixelClip.cpp（见清单 15-6）的开头定义了函数 AvxPiPixelClipCpp，其功 422
能是用指定的阈值对源图像缓冲区进行像素裁剪。函数开始时对 num_pixels 进行验证，包括大小验证以及是否能被 32 整除。图像的像素数能整除 32 这个要求对算法的约束并不像看上去那么奇怪。实际上，大多数的数字图像都使用 64 像素的整数倍来储存，这是 JPEG 压缩技术的处理要求。函数接下来对源缓冲区和目标缓冲区是否数据对齐进行验证。

主循环使用的算法比较简单，对从源图像缓冲区输入的像素进行判断，看是不是在 thresh_lo 和 thresh_hi 区间内。如果不是则以相应的阈值替代并存入目标图像缓冲区内，如果在区间内，则不做改变存入。处理循环中还计算了裁剪像素的数目，用来和该算法的汇编语言版本做对比。

清单 15-6 还包含了 C++ 函数 AvxPackedIntegerPixelClip。在函数开始处为模拟的源图像缓冲区和目标图像缓冲区动态分配了存储空间。接着使用 0 到 255 间的随机数初始化源图像缓冲区中的像素。初始化 PcData 型变量 pc_data1 和 pc_data2 之后，函数调用 C++ 和汇编版本的像素裁剪算法，并对两个算法执行的结果进行差异比较。

清单 15-7 显示了像素裁剪算法的汇编实现版本。函数开头定义了结构体 PcData 的汇编版本。接着定义了一个独立的内存段 PcConstVals，用来存放算法所需的常量。在这里定义独立的内存段，而不是使用 .const 段，是因为 .const 段无法进行 32 字节边界对齐。PcConstVals 段使用 readonly align(32) public 语句定义了一个允许 32 字节数据对齐的只读内存段。注意在此内存段中的第一个数据值 PixelScale，是自动对齐到 32 字节边界的。对 256 位宽组合值进行适当对齐后就可以使用 vmovdqa 指令了。PcConstVals 中包含的其他数据值只是为了演示使用。

在函数序言之后，与 C++ 版本类似，AvxPiPixelClip_ 进行像素大小和缓冲区对齐的验证。指令 vmovdqa ymm5,ymmword ptr[PixelScale] 把像素映射值加载到寄存器 YMM5，这个参数值用来把像素值从 [0, 255] 重新调节到 [-128, 127]。下一条指令 vpbroadcastb ymm0,[ebx+PcData.ThreshLo]（广播整型数据）把源操作数中的字节 ThreshLo 拷贝到 YMM0 中的所有 32 字节元素中。另一条 vpbroadcastb 指令执行同样的操作，不过操作数换为寄存器 YMM1 和 ThreshHi。接下来，所有这些值均被 vpsubb 指令使用 PixelScale 进行重新调节。

图 15-4 显示了执行像素裁剪过程的指令序列。在开始时，主处理循环的每个迭代使用指令 vmovdqa 从源图像缓冲区加载 32 像素块到寄存器 YMM0。接着函数使用指令 vpsubb ymm0,ymm0,ymm5 重新调节 YMM0 中的像素（回顾一下，YMM5 中已经包含了 PixelScale）。上述过程完成后，执行 vpcmpgtb 指令，该指令对所有大于 ThreshHi 的像素计算掩码。注意，vpcmpgtb 指令执行字节比较的时候使用的是有符号整型运算，这就是之前进行像素重新调节的原因。第二条 vpcmpgtb 指令计算所有小于 ThreshLo 的像素的掩码。特别要注意的是，这里的源操作数被反转了（即第一个操作数包含阈值，第二个操作数包含像素值），目的是计算小于 ThreshLo 的像素掩码。函数使用这些阈值掩码和一些组合布尔代 423
数，对处于阈值之外（大于 ThreshHi，小于 ThreshLo）的像素值进行替换。更新后的像素块保存到目标缓冲区中。



注：上述的指令序列只展示了每个 YMM 寄存器中低八字节内容

424 图 15-4 用于处理像素裁剪的 x86 AVX 指令序列

指令 vpmovmskb eax,ymm1 创建了已裁剪像素的掩码，并将其存入寄存器 EAX（此操作由 vpmovmskb 执行，相当于 eax[i] = ymm1[i*8+7]，其中 i = 0,1,2,…,31）。接着执行指令 popcnt eax,eax，计算当前像素块中已裁剪像素数目，存入 EAX 中。下一指令 add edx,eax 更新保存在 EDX 中的裁剪像素数目。执行完上述主处理循环，寄存器 EDX 中的值转存入 PcData 结构成员 NumClippedPixels 中。

示例程序 AvxPackedIntegerPixelClip 的执行结果如输出 15-3 所示。表 15-1 列出了在使用 8MB 图像缓冲×的情况下 C— 和汇编版本裁剪算法的执行时间。和本书之前所采用的时间测量不同，表 15-1 中没有包含 Intel Core i3-2310M 的基准时间，因为该款处理器不支持 AVX2 指令集。

输出 15-3 示例程序 AvxPackedIntegerPixelClip

```
Results for AvxPackedIntegerPixelClip
  NumClippedPixels1: 327228
  NumClippedPixels2: 327228
  Destination buffer memory compare passed

Benchmark times saved to file __AvxPackedIntegerPixelClipTimed.csv
```

表 15-1 AvxPiPixelClip 函数的平均执行时间（单位：微秒）

CPU	AvxPiPixelClipCpp (C++)	AvxPiPixelClip_ (x86-AVX)
Intel Core i7-4770	8866	1075
Intel Core i7-4600U	10 235	1021

15.2.2 图像阈值二分法

第 10 章中，我们学习了示例程序 SsePackedIntegerThreshold，它使用 x86-SSE 指令集执行灰度图像的阈值计算。它也计算了超过阈值的灰度像素的平均强度值。本节将给出实现类似功能的图像阈值计算程序的 x86-AVX 版本。新的示例程序名叫 AvxPackedIntegerThreshold，它再次显示了 x86-AVX 相比于 x86-SSE 的性能优势。示例程序的源代码分别参见清单 15-8、清单 15-9 和清单 15-10。 425

清单 15-8 AvxPackedIntegerThreshold.h

```
#pragma once
#include "ImageBuffer.h"

// 图像阈值数据结构。此结构与 AvxPackedIntegerThreshold_.asm 中定义的结构一致
typedef struct
{
    Uint8* PbSrc;               // 源图像像素缓冲区
    Uint8* PbMask;              // 掩码像素缓冲区
    Uint32 NumPixels;           // 源图像像素的数目
    Uint8 Threshold;            // 图像阈值
    Uint8 Pad[3];               // 供将来使用
    Uint32 NumMaskedPixels;     // 掩码像素数目
    Uint32 SumMaskedPixels;     // 掩码像素和
    double MeanMaskedPixels;    // 掩码像素均值
} ITD;

// 函数定义在 AvxPackedIntegerThreshold.cpp 中
extern bool AvxPiThresholdCpp(ITD* itd);
extern bool AvxPiCalcMeanCpp(ITD* itd);

// 函数定义在 AvxPackedIntegerThreshold_.asm 中
extern "C" bool AvxPiThreshold_(ITD* itd);
extern "C" bool AvxPiCalcMean_(ITD* itd);

// 函数定义在 AvxPackedIntegerThresholdTimed.cpp 中
extern void AvxPiThresholdTimed(void);

// 其他常数
const Uint8 TEST_THRESHOLD = 96;
```

清单 15-9 AvxPackedIntegerThreshold.cpp

```
#include "stdafx.h"
#include "AvxPackedIntegerThreshold.h"
#include <stddef.h>

extern "C" Uint32 NUM_PIXELS_MAX = 16777216;

bool AvxPiThresholdCpp(ITD* itd)
{
    Uint8* pb_src    = itd->PbSrc;
    Uint8* pb_mask   = itd->PbMask;
    Uint8 threshold  = itd->Threshold;
426 Uint32 num_pixels = itd->NumPixels;

    // 确保 num_pixels 有效
    if ((num_pixels == 0) || (num_pixels > NUM_PIXELS_MAX))
        return false;
    if ((num_pixels & 0x1f) != 0)
        return false;

    // 确保图像缓冲区正确对齐了
    if (((uintptr_t)pb_src & 0x1f) != 0)
        return false;
    if (((uintptr_t)pb_mask & 0x1f) != 0)
        return false;

    // 开始算处理图像
    for (Uint32 i = 0; i < num_pixels; i++)
        *pb_mask++ = (*pb_src++ > threshold) ? 0xff : 0x00;

    return true;
}

bool AvxPiCalcMeanCpp(ITD* itd)
{
    Uint8* pb_src = itd->PbSrc;
    Uint8* pb_mask = itd->PbMask;
    Uint32 num_pixels = itd->NumPixels;

    // 确保 num_pixels 有效
    if ((num_pixels == 0) || (num_pixels > NUM_PIXELS_MAX))
        return false;
    if ((num_pixels & 0x1f) != 0)
        return false;

    // 确保图像缓冲区正确对齐了
    if (((uintptr_t)pb_src & 0x1f) != 0)
        return false;
    if (((uintptr_t)pb_mask & 0x1f) != 0)
        return false;

    // 计算掩码像素均值
    Uint32 sum_masked_pixels = 0;
    Uint32 num_masked_pixels = 0;

    for (Uint32 i = 0; i < num_pixels; i++)
    {
        Uint8 mask_val = *pb_mask++;
        num_masked_pixels += mask_val & 1;
        sum_masked_pixels += (*pb_src++ & mask_val);
    }
427
```

```
    itd->NumMaskedPixels = num_masked_pixels;
    itd->SumMaskedPixels = sum_masked_pixels;

    if (num_masked_pixels > 0)
        itd->MeanMaskedPixels = (double)sum_masked_pixels /=
num_masked_pixels;
    else
        itd->MeanMaskedPixels = -1.0;

    return true;
}

void AvxPiThreshold()
{
    wchar_t* fn_src = L"..\\..\\..\\DataFiles\\TestImage2.bmp";
    wchar_t* fn_mask1 = L"__TestImage2_Mask1.bmp";
    wchar_t* fn_mask2 = L"__TestImage2_Mask2.bmp";
    ImageBuffer* in_src = new ImageBuffer(fn_src);
    ImageBuffer* in_mask1 = new ImageBuffer(*in_src, false);
    ImageBuffer* in_mask2 = new ImageBuffer(*in_src, false);
    ITD itd1, itd2;

    itd1.PbSrc     = (UInt8*)in_src->GetPixelBuffer();
    itd1.PbMask    = (UInt8*)in_mask1->GetPixelBuffer();
    itd1.NumPixels = in_src->GetNumPixels();
    itd1.Threshold = TEST_THRESHOLD;

    itd2.PbSrc = (UInt8*)in_src->GetPixelBuffer();
    itd2.PbMask = (UInt8*)in_mask2->GetPixelBuffer();
    itd2.NumPixels = in_src->GetNumPixels();
    itd2.Threshold = TEST_THRESHOLD;

    bool rc1 = AvxPiThresholdCpp(&itd1);
    bool rc2 = AvxPiThreshold_(&itd2);

    if (!rc1 || !rc2)
    {
        printf("Bad Threshold return code: rc1=%d, rc2=%d\n", rc1, rc2);
        return;
    }

    in_mask1->SaveToBitmapFile(fn_mask1);
    in_mask2->SaveToBitmapFile(fn_mask2);

    // 计算掩码像素均值
    rc1 = AvxPiCalcMeanCpp(&itd1);
    rc2 = AvxPiCalcMean_(&itd2);

    if (!rc1 || !rc2)
    {
        printf("Bad CalcMean return code: rc1=%d, rc2=%d\n", rc1, rc2);
        return;
    }

    printf("Results for AvxPackedIntegerThreshold\n\n");
    printf("                              C++       X86-AVX\n");
    printf("-------------------------------------------\n");
    printf("SumPixelsMasked:  ");
    printf("%12u  %12u\n", itd1.SumMaskedPixels, itd2.SumMaskedPixels);
    printf("NumPixelsMasked:  ");
    printf("%12u  %12u\n", itd1.NumMaskedPixels, itd2.NumMaskedPixels);
    printf("MeanPixelsMasked: ");
```

428

```
    printf("%12.6lf  %12.6lf\n", itd1.MeanMaskedPixels,↩
 itd2.MeanMaskedPixels);

    delete im_src;
    delete im_mask1;
    delete im_mask2;
}

int _tmain(int argc, _TCHAR* argv[])
{
    try
    {
        AvxPiThreshold();
        AvxPiThresholdTimed();
    }

    catch (...)
    {
        printf("Unexpected exception has occurred!\n");
        printf("File: %s (_tmain)\n", __FILE__);
    }
    return 0;
}
```

清单 15-10 AvxPackedIntegerThreshold_.asm

```
        .model flat,c
         extern NUM_PIXELS_MAX:dword

; 图像阈值数据结构 ( 见 AvxPackedIntegerThreshold.h)
ITD                 struct
PbSrc               dword ?
PbMask              dword ?
NumPixels           dword ?
Threshold           byte ?
Pad                 byte 3 dup(?)
NumMaskedPixels     dword ?
SumMaskedPixels     dword ?
MeanMaskedPixels    real8 ?
ITD                 ends

; 为常量准备的用户自定义段
ItConstVals segment readonly align(32) public
PixelScale      byte 32 dup(80h)            ;从 uint8 转换为 Int8 的调节值
R8_MinusOne     real8 -1.0                  ;无效的均值
ItConstVals     ends

                .code

; extern "C" bool AvxPiThreshold_(ITD* itd);
;
; 描述：本函数对 8 位灰度图像进行图像阈值计算
;
; 返回：0 = 无效的大小或者图像缓冲区没有对齐
;       1 = 成功
;
; 需要：AVX2

AvxPiThreshold_ proc
        push ebp
        mov ebp,esp
```

```
        push esi
        push edi

; 加载并验证 ITD 结构中的参数值
        mov edx,[ebp+8]                 ;edx = 'itd'
        xor eax,eax                     ;设置错误返回码
        mov ecx,[edx+ITD.NumPixels]     ;ecx = NumPixels
        test ecx,ecx
        jz Done                         ;如果 num_pixels == 0 则跳转
        cmp ecx,[NUM_PIXELS_MAX]
        ja Done                         ;如果 num_pixels 太大则跳转
        test ecx,1fh
        jnz Done                        ;如果 num_pixels % 32 != 0 则跳转
        shr ecx,5                       ;ecx = 组合像素数目

        mov esi,[edx+ITD.PbSrc]         ;esi = PbSrc
        test esi,1fh
        jnz Done                        ;未对齐则跳转
        mov edi,[edx+ITD.PbMask]        ;edi = PbMask
        test edi,1fh
        jnz Done                        ;未对齐则跳转

; 初始化组合阈值
        vpbroadcastb ymm0,[edx+ITD.Threshold]   ;ymm0 = 组合阈值
        vmovdqa ymm7,ymmword ptr [PixelScale]   ;ymm7 = uint8 转换为 int8
        vpsubb ymm2,ymm0,ymm7                   ;ymm1 = 已调节阈值

; 创建掩码图像
@@:     vmovdqa ymm0,ymmword ptr [esi]  ;加载下一个组合像素
        vpsubb ymm1,ymm0,ymm7           ;ymm1 = 已调节图像像素
        vpcmpgtb ymm3,ymm1,ymm2         ;与阈值比较
        vmovdqa ymmword ptr [edi],ymm3  ;保存组合阈值掩码

        add esi,32
        add edi,32
        dec ecx
        jnz @B                          ;重复直到完成
        mov eax,1                       ;设置返回码

Done:   pop edi
        pop esi
        pop ebp
        ret
AvxPiThreshold_ endp

; Macro AvxPiCalcMeanUpdateSums
;
; 描述：下面定义的宏，其功能是更新 ymm4 中的 sum_masked_pixels。为了
;       防止溢出，它会对中间值进行必要的复位
;
; 寄存器内容：
;   ymm3:ymm2 = 组合字 sum_masked_pixels
;   ymm4 = 组合双字 sum_masked_pixels
;   ymm7 = 组合 0
;
; 临时寄存器：
;   ymm0, ymm1, ymm5, ymm6

AvxPiCalcMeanUpdateSums macro

; 转换组合字 sum_masked_pixels 为双字
        vpunpcklwd ymm0,ymm2,ymm7
```

430

```
        vpunpcklwd ymm2,ymm3,ymm7
        vpunpckhwd ymm5,ymm2,ymm7
431     vpunpckhwd ymm6,ymm3,ymm7

; 更新 sum_masked_pixels 中的组合双字和
        vpaddd ymm0,ymm0,ymm1
        vpaddd ymm5,ymm5,ymm6
        vpaddd ymm4,ymm4,ymm0
        vpaddd ymm4,ymm4,ymm5

; 重置中间值
        xor edx,edx                     ;重置更新计数器
        vpxor ymm2,ymm2,ymm2            ;重置 sum_masked_pixels 低价部分
        vpxor ymm3,ymm3,ymm3            ;重置 sum_masked_pixels 高价部分
        endm

; extern "C" bool AvxPiCalcMean_(ITD* itd);
;
; 描述：本函数计算所有高于阈值的图像像素的均值，使用的掩码由函数
;       AvxPiThreshold_ 创建
;
; 返回：0 = 无效的图像大小或图像缓冲区未对齐
;       1 = 成功
;
; 需要：AVX2, POPCNT

AvxPiCalcMean_  proc
        push ebp
        mov ebp,esp
        push ebx
        push esi
        push edi

; 加载并验证 ITD 结构中的参数值
        mov eax,[ebp+8]                 ;eax = 'itd'
        mov ecx,[eax+ITD.NumPixels]     ;ecx = NumPixels
        test ecx,ecx
        jz Error                        ;如果 num_pixels == 0 则跳转
        cmp ecx,[NUM_PIXELS_MAX]
        ja Error                        ;如果 num_pixels 太大则跳转
        test ecx,1fh
        jnz Error                       ;如果 num_pixels % 32 != 0 则跳转
        shr ecx,5                       ;ecx = 组合像素数目

        mov edi,[eax+ITD.PbMask]        ;edi = PbMask
        test edi,1fh
        jnz Error                       ;如果 PbMask 没有对齐则跳转
        mov esi,[eax+ITD.PbSrc]         ;esi = PbSrc
        test esi,1fh
432     jnz Error                       ;如果 PbSrc 没有对齐则跳转

; 初始化均值计算的参数值
        xor edx,edx             ;edx = 更新计数器
        vpxor ymm7,ymm7,ymm7    ;ymm7 = 组合 0
        vmovdqa ymm2,ymm7       ;ymm2 = sum_masked_pixels (16 字)
        vmovdqa ymm3,ymm7       ;ymm3 = sum_masked_pixels (16 字)
        vmovdqa ymm4,ymm7       ;ymm4 = sum_masked_pixels (8 双字)
        xor ebx,ebx             ;ebx = num_masked_pixels (1 双字)

; 处理循环中的寄存器使用
; esi = PbSrc, edi = PbMask, eax = 机动使用的寄存器
; ebx = num_pixels_masked, ecx = NumPixels / 32, edx = 更新计数器
```

```
;
; ymm0 = 组合像素，ymm1 = 组合掩码
; ymm3:ymm2 = sum_masked_pixels (32 字)
; ymm4 = sum_masked_pixels (8 双字)
; ymm5 = 机动使用的寄存器
; ymm6 = 机动使用的寄存器
; ymm7 = 组合 0

@@:     vmovdqa ymm0,ymmword ptr [esi]      ;加载下一个组合像素
        vmovdqa ymm1,ymmword ptr [edi]      ;加载下一个组合掩码

; 更新 num_masked_pixels
        vpmovmskb eax,ymm1
        popcnt eax,eax
        add ebx,eax

; 更新 sum_masked_pixels (字值)
        vpand ymm6,ymm0,ymm1                ;设置非掩码像素为 0
        vpunpcklbw ymm0,ymm6,ymm7
        vpunpckhbw ymm1,ymm6,ymm7           ;ymm1:ymm0 = 掩码像素 (字)
        vpaddw ymm2,ymm2,ymm0
        vpaddw ymm3,ymm3,ymm1               ;ymm3:ymm2 = sum_masked_pixels

; 检查 xmm4 中的双字 sum_masked_pixels 和 ebx 中的 num_masked_pixels
; 是否有必要更新
        inc edx
        cmp edx,255
        jb NoUpdate
        AvxPiCalcMeanUpdateSums
NoUpdate:
        add esi,32
        add edi,32
        dec ecx
        jnz @@                              ;重复循环直到完成                    433

; 主处理循环已经完成。如果有必要，最后更新 xmm4 中的 sum_masked_pixels 和
; ebx 中的 num_masked_pixels
        test edx,edx
        jz @F
        AvxPiCalcMeanUpdateSums

; 计算并保存最后的 sum_masked_pixels 和 num_masked_pixels
@@:     vextracti128 xmm0,ymm4,1
        vpaddd xmm1,xmm0,xmm4
        vphaddd xmm2,xmm1,xmm7
        vphaddd xmm3,xmm2,xmm7
        vmovd edx,xmm3                      ;edx = 最后的 sum_mask_pixels

        mov eax,[ebp+8]                     ;eax = 'itd'
        mov [eax+ITD.SumMaskedPixels],edx   ;保存最后为 sum_masked_pixels
        mov [eax+ITD.NumMaskedPixels],ebx   ;保存最后为 num_masked_pixels

; 计算掩码像素均值
        test ebx,ebx                        ;num_mask_pixels 是否为 0?
        jz NoMean                           ;如果是，就过计算均值
        vcvtsi2sd xmm0,xmm0,edx             ;xmm0 = sum_masked_pixels
        vcvtsi2sd xmm1,xmm1,ebx             ;xmm1 = num_masked_pixels
        vdivsd xmm0,xmm0,xmm1               ;xmm0 = mean_masked_pixels
        jmp @F
NoMean: vmovsd xmm0,[R8 MinusOne]           ;没有均值时，使用 -1.0
@@:     vmovsd [eax+ITD.MeanMaskedPixels],xmm0  ;保存均值
        mov eax,1                           ;设置返回码
```

```
        vzeroupper

Done:   pop edi
        pop esi
        pop ebx
        pop ebp
        ret

Error:  xor eax,eax                    ;设置错误返回码
        jmp Done
AvxPiCalcMean_  endp
        end
```

示例程序 AvxPackedIntegerThreshold（见清单 15-8 和清单 15-9）和 SsePackedIntegerThreshold 的 C++ 代码差不多相同。其中最显著的区别是 num_pixels 值的检测部分，不同于 AvxPiThresholdCpp 和 AvxPiCalcMeanCpp 中的 16 倍的整数倍检测，现在是进行 32 倍的整数倍检测。这些函数也对源图像缓冲区和目的图像缓冲区进行了 32 字节边界对齐的检查。

434

汇编语言文件 AvxPackedIntegerThreshold_.asm（见清单 15-10）中包含了一些值得详细讨论的修改。最明显的区别是，x86-SSE 执行时使用的是 XMM 寄存器，而 x86-AVX 版本用 YMM 寄存器进行了替代。这意味着新的算法可以处理 32 个像素的块，而不是 16 个像素的块。x86-AVX 版本使用的指令也不被 SSSE3 支持（SSSE3 是一种 x86-SSE 级指令集，常用在编码原始图像阈值和平均值计算算法中）。函数定义了一个定制的内存段 ItConstVals，与上节中程序类似，是为了便于对 256 位宽的常量值 PixelScale 进行正确的边界对齐。

相比于 x86-SSE 的版本，AvxPiThreshold_ 函数有几个小的改动。其一为结构成员 NumPixels 的值，从整除 16 检测变为了整除 32 检测；其二为对源图像缓冲区 PbSrc 和目标图像缓冲区 PbDes 的边界对齐检测，变成了 32 字节对齐检测；其三为替换了指令 vpshufb，使用 vpbroadcastb ymm0, [edx+ITD.Threshold] 创建组合阈值。

在原来的 x86-SSE 版本中，包含一个私有函数 SsePiCalcMeanUpdateSums，其作用为定期更新算法中的双字中间值像素和以及像素计数。x86-AVX 版本的实现算法中，相应的 AvxPiCalcMeanUpdateSums 使用宏的方式执行，这样做所需的处理器时间更少。减少处理器需求的主要原因是统计掩码像素的方法更高效。

函数 AvxPiCalcMean_ 也包括上述的 NumPixels 和图像缓冲区的大小和对齐检查，主处理循环中使用 vpmovmskb 和 popcnt 指令计算掩码像素的数目。这些变化消除了一对数据传输指令，简化了 AvxPiCalcMeanUpdateSums 宏的编程。如图 15-5 所示，在主处理循环之后，使用了指令 vextracti128 计算最终的 SumMaskedPixels 值。最后一个改变是调用 vcvtsi2sd 指令，此指令需要两个源操作数。与所有其他 x86-AVX 标量双精度浮点指令一样，第一个源操作数的高四字部分会被复制到目标操作数的高四字部分。第二个源操作数包含的有符号整型数会被转换为双精度浮点数，结果保存到目标操作数的低四字部分。输出 15-4 显示了示例程序 AvxPackedIntegerThreshold 的执行结果。

435

输出 15-4 示例程序 AvxPackedIntegerThreshold

```
Results for AvxPackedIntegerThreshold

                          C++        X86-AVX
-------------------------------------------
SumPixelsMasked:      23813043      23813043
```

```
NumPixelsMasked:        138220        138220
MeanPixelsMasked:    172.283628    172.283628

Benchmark times saved to file __AvxPackedImageThresholdTimed.csv
```

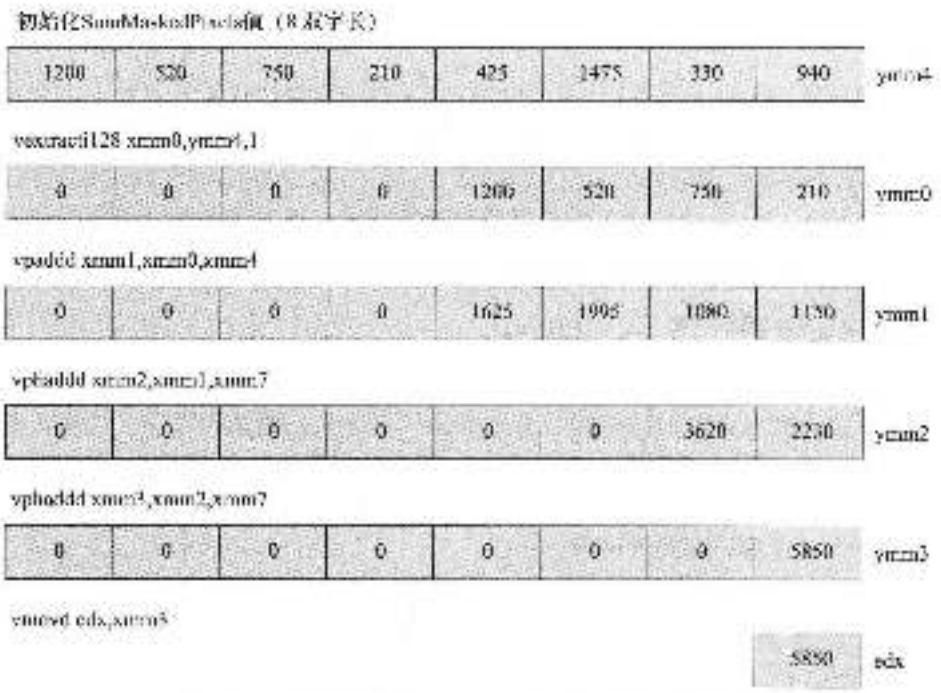


图 15-5 计算最终 SumMaskedPixels 值的指令序列

注意：寄存器 XMM7 总会全 0 出现

表 15-2 给出了阈值算法的 C++ 和 x86-AVX 汇编语言版本的运行时间测量数据。x86-SSE 实现算法的时间测量数据见表 10-2。 436

表 15-2 使用 TestImage2.bmp 图像阈值算法的平均执行时间（单位：微秒）

CPU	C++	x86-AVX	x86-SSE
Intel Core i7-4770	518	39	49
Intel Core i7-4600U	627	50	60

15.3 总结

本章演示了如何使用 x86-AVX 的组合整型能力。我们学习了如何对 256 位宽组合整型操作数执行各种基本运算，并通过两个示例程序演示了如何使用 x86-AVX 指令集实现通用的图像处理算法。在本章以及前两章的例子中，我们集中讨论了处理组合整型、组合浮点和标量浮点操作数时 x86-SSE 和 x86-AVX 的差异。在下一章中，我们将学习 AVX2 引入的一些新指令以及随之引入的扩展特性。 437 - 438

x86-AVX 编程——新指令

通过前面三章，我们学习了如何使用 x86-AVX 指令集操作标量浮点数、紧合浮点数和紧合整数。本章将讲解 x86-AVX 的一些新编程特性。我们将从一个示例程序开始，说明如何使用 cpuid 指令确定处理器是否支持 x86-SSE、x86-AVX 或者特定的子功能扩展。接着通过一系列示例程序，演示 x86-AVX 的高级数据操作指令。本章的最后一节将描述 x86 的一些新的通用寄存器指令。

16.1 检测处理器特性（CPUID）

在编写软件时，当使用 x86 处理器功能扩展如 x86-SSE，x86-AVX 或某个增强的指令组时，永远不要通过处理器的微架构、型号或品牌名称来判断是否可用，而应该总是使用 cpuid（CPU Identification）指令来测试期望的功能扩展是否可用。本节的示例程序 AvxCpuid 演示了如何使用这个指令来检测特定处理器的扩展功能。清单 16-1 和清单 16-2 分别列出了示例程序的 C++ 和 ASM 编语言的源代码。

清单 16-1 AvxCpuid.cpp

```
#include "stdafx.h"
#include "MiscDefs.h"
#include <memory.h>

439 // 该结构用来存储 cpuid 指令的结果。它必须和 AvxCpuid_.asm 中定义的结构一致
typedef struct
{
    Uint32 EAX;
    Uint32 EBX;
    Uint32 ECX;
    Uint32 EDX;
} CpuidRegs;

// 该结构包含了本书用到的 cpuid 支持的特性标志
typedef struct
{
    // 基本信息
    Uint32 MaxEAX;      // cpuid 支持的最大 EAX 值
    char VendorId[13];  // 处理器供应商 ID 字符串

    // 处理器特性标志。如果特性扩展或者指令可用，则设置为真
    bool SSE;
    bool SSE2;
    bool SSE3;
    bool SSSE3;
    bool SSE4_1;
    bool SSE4_2;
    bool AVX;
    bool AVX2;
    bool F16C;
    bool FMA;
```

```
    bool POPCNT;
    bool BMI1;
    bool BMI2;
    bool LZCNT;
    bool VOWBE;

    // OS 是否启用某些特性的信息
    bool OSXSAVE;       // 如果 XSAVE 被 OS 激活则为真
    bool SSE_STATE;     // 如果 XMM 状态被 OS 激活则为真
    bool AVX_STATE;     // 如果 YMM 状态被 OS 激活则为真
} CpuidFeatures;

extern "C" Uint32 Cpuid (Uint32 r_eax, Uint32 r_ecx, CpuidRegs* out);
extern "C" void Xgetbv (Uint32 r_ecx, Uint32* r_eax, Uint32* r_edx);

// 该函数不能在较旧型号的 CPU 上工作，尤其是那些 2006 年之前生产的。该函数已在
// Windows 7 (SP1) 和 Windows 8.1 测试过                                    440
void GetCpuidFeatures(CpuidFeatures* cf)
{
    CpuidRegs r_out;

    memset(cf, 0, sizeof(CpuidFeatures));

    // 获得 MaxEAX 和 VendorID
    Cpuid (0, 0, &r_out);
    cf->MaxEAX = r_out.EAX;
    *(Uint32 *)(cf->VendorId + 0) = r_out.EBX;
    *(Uint32 *)(cf->VendorId + 4) = r_out.EDX;
    *(Uint32 *)(cf->VendorId + 8) = r_out.ECX;
    cf->VendorId[12] = '\0';

    // 处理器太老则退出
    if (cf->MaxEAX < 1)
        return;

    // 获取 CPUID.01H 特性标志
    Cpuid (1, 0, &r_out);
    Uint32 cpuid01_ecx = r_out.ECX;
    Uint32 cpuid01_edx = r_out.EDX;

    // 获取 CPUID [EAX=07H,ECX=00H] 特性标志
    Cpuid (7, 0, &r_out);
    Uint32 cpuid07_ebx = r_out.EBX;

    // CPUID.01H:EDX.SSE[bit 25]
    cf->SSE = (cpuid01_edx & (0x1 << 25)) ? true : false;

    // CPUID.01H:EDX.SSE2[bit 26]
    if (cf->SSE)
        cf->SSE2 = (cpuid01_edx & (0x1 << 26)) ? true : false;

    // CPUID.01H:ECX.SSE3[bit 0]
    if (cf->SSE2)
        cf->SSE3 = (cpuid01_ecx & (0x1 << 0)) ? true : false;

    // CPUID.01H:ECX.SSSE3[bit 9]
    if (cf->SSE3)
        cf->SSSE3 = (cpuid01_ecx & (0x1 << 9)) ? true : false;

    // CPUID.01H:ECX.SSE4.1[bit 19]
    if (cf->SSSE3)
        cf->SSE4_1 = (cpuid01_ecx & (0x1 << 19)) ? true : false;          441
```

```
    // CPUID.01H:ECX.SSE4.2[bit 20]
    if (cf->SSE4_1)
        cf->SSE4_2 = (cpuid01_ecx & (0x1 << 20)) ? true : false;

    // CPUID.01H:ECX.POPCNT[bit 23]
    if (cf->SSE4_2)
        cf->POPCNT = (cpuid01_ecx & (0x1 << 23)) ? true : false;

    // CPUID.01H:ECX.OSXSAVE[bit 27]
    cf->OSXSAVE = (cpuid01_ecx & (0x1 << 27)) ? true : false;

    // 测试 OSXSAVE 状态来确定 XGETBV 是否启用
    if (cf->OSXSAVE)
    {
        // 使用 XGETBV 获取下列信息：
        // 如果 (XCR0[1]==1), XSAVE 用 SSE 状态
        // 如果 (XCR0[2]==1), XSAVE 用 AVX 状态

        Uint32 xgetbv_eax, xgetbv_edx;

        Xgetbv_(0, &xgetbv_eax, &xgetbv_edx);
        cf->SSE_STATE = (xgetbv_eax & (0x1 << 1)) ? true : false;
        cf->AVX_STATE = (xgetbv_eax & (0x1 << 2)) ? true : false;

        // OS 支持 SSE 和 AVX 状态信息吗?
        if (cf->SSE_STATE && cf->AVX_STATE)
        {
            // CPUID.01H:ECX.AVX[bit 28] = 1
            cf->AVX = (cpuid01_ecx & (0x1 << 28)) ? true : false;

            if (cf->AVX)
            {
                // CPUID.01H:ECX.F16C[bit 29]
                cf->F16C = (cpuid01_ecx & (0x1 << 29)) ? true : false;

                // CPUID.01H:ECX.FMA[bit 12]
                cf->FMA = (cpuid01_ecx & (0x1 << 12)) ? true : false;

                // CPUID.(EAX = 07H, ECX = 00H):EBX.AVX2[bit 5]
                cf->AVX2 = (cpuid07_ebx & (0x1 << 5)) ? true : false;
            }
        }
442 }

    // CPUID.(EAX = 07H, ECX = 00H):EBX.BMI1[bit 3]
    cf->BMI1 = (cpuid07_ebx & (0x1 << 3)) ? true : false;

    // CPUID.(EAX = 07H, ECX = 00H):EBX.BMI2[bit 8]
    cf->BMI2 = (cpuid07_ebx & (0x1 << 8)) ? true : false;

    // CPUID.80000001H:ECX.LZCNT[bit 5]
    Cpuid_(0x80000001, 0, &r_out);
    cf->LZCNT = (r_out.ECX & (0x1 << 5)) ? true : false;

    // 获取 MOVBE
    // CPUID.01H:ECX.MOVBE[bit 22]
    cf->MOVBE = cpuid01_ecx & (0x1 << 22) ? true : false;
}

int _tmain(int argc, _TCHAR* argv[])
{
    CpuidFeatures cf;
```

```
    GetCpuidFeatures(&cf);
    printf("Results for AvxCpuid\n");
    printf("MaxEAX:    %d\n", cf.MaxEAX);
    printf("VendorId:  %s\n", cf.VendorId);
    printf("SSE:       %d\n", cf.SSE);
    printf("SSE2:      %d\n", cf.SSE2);
    printf("SSE3:      %d\n", cf.SSE3);
    printf("SSSE3:     %d\n", cf.SSSE3);
    printf("SSE4_1:    %d\n", cf.SSE4_1);
    printf("SSE4_2:    %d\n", cf.SSE4_2);
    printf("POPCNT:    %d\n", cf.POPCNT);
    printf("AVX:       %d\n", cf.AVX);
    printf("F16C:      %d\n", cf.F16C);
    printf("FMA:       %d\n", cf.FMA);
    printf("AVX2:      %d\n", cf.AVX2);
    printf("BMI1       %d\n", cf.BMI1);
    printf("BMI2       %d\n", cf.BMI2);
    printf("LZCNT      %d\n", cf.LZCNT);
    printf("MOVBE      %d\n", cf.MOVBE);
    printf("\n");
    printf("OSXSAVE    %d\n", cf.OSXSAVE);
    printf("SSE_STATE  %d\n", cf.SSE_STATE);
    printf("AVX_STATE  %d\n", cf.AVX_STATE);

    return 0;
}
```

443

清单 16-2 AvxCpuid_asm

```
        .model flat,c

; 该结构必须和 AvxCpuid.cpp 中定义的结构一致

CpuidRegs   struct
RegEAX      dword ?
RegEBX      dword ?
RegECX      dword ?
RegEDX      dword ?
CpuidRegs   ends
            .code

; extern "C" Uint32 Cpuid_(Uint32 r_eax, Uint32 r_ecx, CpuidRegs* r_out);
;
; 描述：下面的函数使用 CPUID 指令来查询处理器标记和特性信息
;
; 返回值：eax==0  不支持该 CPUID 叶子
;         eax!=0  支持该 CPUID 叶子
;
;             仅在 r_eax<=MaxEAX 时，返回值才有意义

Cpuid_  proc
        push ebp
        mov ebp,esp
        push ebx
        push esi

; 载入 eax 和 ecx 值，然后使用 cpuid
        mov eax,[ebp+8]
        mov ecx,[ebp+12]
        cpuid
```

```
    ;保存结果
            mov esi,[ebp+16]
            mov [esi+CpuidRegs.RegEAX],eax
            mov [esi+CpuidRegs.RegEBX],ebx
            mov [esi+CpuidRegs.RegECX],ecx
            mov [esi+CpuidRegs.RegEDX],edx

    ;测试CPUID叶子是否支持该功能
            or eax,ebx
            or ecx,edx
444         or eax,ecx                          ;eax-返回值
            pop esi
            pop ebx
            pop ebp
            ret
    Cpuid_  endp

    ; extern "C" void Xgetbv (UInt32 r_ecx, UInt32* r_eax, UInt32* r_edx);
    ;
    ;描述：下面的函数使用XGETBV指令来获取r_ecx指定的扩展控制寄存器的内容
    ;
    ;注意：如果r_ecx不合法或者XSAVE没有被启用，则会发生一个处理器异常

    Xgetbv_  proc
            push ebp
            mov ebp,esp

            mov ecx,[ebp+8]                 ;ecx = 扩展控制寄存器
            xgetbv

            mov ecx,[ebp+12]
            mov [ecx],eax                   ;保存结果（低双字值）
            mov ecx,[ebp+16]
            mov [ecx],edx                   ;保存结果（高双字值）

            pop ebp
            ret
    Xgetbv_ endp
            end
```

阅读源代码之前，你需要了解 cpuid 指令是如何工作的。使用这个指令之前，必须向寄存器 EAX 装载特定叶子（leaf）值，指定 cpuid 指令应该返回什么类型的信息。根据 EAX 不同的功能值，可能会需要装载次级叶子或子叶子（sub-leaf）值到 ECX 寄存器内。cpuid 指令则通过寄存器 EAX、EBX、ECX 和 EDX 返回结果。本节的示例程序重点关注通过 cpuid 指令检测书中涉及的体系结构特性和指令组。如果你有兴趣学习如何使用 cpuid 指令识别其他的处理器特性和硬件功能，可以参考附录 C 中列出的 Intel 或 AMD 参考手册和操作说明书。

AvxCpuid.cpp 文件顶部声明了两个 C++ 结构（参见清单 16-1）。第一个结构名为 CpuidRegs，用于保存 cpuid 指令返回的结果。第二个名为 CpuidFeatures 的结构包含各种标
445 志，表明是否支持某个特定的处理器特性。

结构声明后面是两个汇编语言函数 Cpuid_ 和 Xgetbv 的声明，分别用于执行 cpuid 和 xgetbv 指令（取得扩展控制寄存器的值）。

函数 GetCpuidFeatures 中，语句 Cpuid (0,0,&r_out) 获取 cpuid 指令支持的最大 EAX 值。Cpuid_ 的前两个参数用来在执行 cpuid 指令前初始化寄存器 EAX 和 ECX；第三个参

数指定一个 CpuidRegs 结构，以保存 cpuid 返回的结果。请注意，在函数 GetCpuidFeatures 中，除非 EAX 等于 7，否则 cpuid 指令将忽略 ECX 的值。在 Cpuid_ 返回前，r_out.EAX 保存了 cpuid 指令所支持的最大 EAX 值，而 r_out.EBX、r_out.ECX 和 r_out.EDX 则组成了厂家的标识字符串。所有这些值都保存到了指定的 CpuidFeatures 结构。

为了保证剩下的逻辑合理，在检测到运行于一个老处理器时，GetCpuidFeatures 函数将终止。接着，函数 Cpuid_ 被调用两次，以获得必要的 cpuid 功能状态标志。与 x86-SSE 有关的 cpuid 状态标志被解码并保存。

应用程序只有在处理器和它的主机操作系统都支持的情况下才能使用 x86-AVX 指令集的计算资源。处理器的 OSXSAVE 标志指示操作系统在任务切换时是否保存 x86-AVX 的状态信息。OSXSAVE 标志为真也表明其可以放心地使用 xgetbv 指令判断操作系统是否支持 XMM 和 YMM 寄存器。一旦这些信息确定，GetCpuidFeatures 函数继续解码 cpuid，获得 x86-AVX 及其相关功能扩展的标志状态。它还确定几个需要操作系统支持的 x86-AVX 指令组的可用性。

AvxCpuid_asm 文件的顶部是汇编语言版本的 CpuidRegs 结构（见清单 16-2）。函数 Cpuid_ 的代码很直观：加载 EAX 寄存器和 ECX 寄存器提供的参数值，执行 cpuid 指令，并将结果保存在内存中指定的位置。注意，如果 EAX 提供的叶子值是无效的，并且小于或等于处理器支持的最大叶子值，cpuid 指令将返回 0 值到 EAX、EBX、ECX 和 EDX。

函数 Xgetbv_ 的代码也很简单。然而，需要注意的是，如果 r_ecx 指定的扩展控制寄存器无效，或者处理器的 OSXSAVE 状态标志设置为 false，处理器将产生一个异常。这就解释了为什么在 C++ 函数 GetCpuidFeatures 中，调用 Xgetbv_ 前要测试 OSXSAVE 标志。表 16-1 总结了在几种不同的处理器上运行 AvxCpuid 示例程序得到的结果。 446

表 16-1 示例程序 AvxCpuid 的运行结果汇总

特性	E6700	Q9650	i3-2310M	i7-4600U	i7-4770
最大 EAX 值	10	13	13	13	13
厂商标识	GenuineIntel	GenuineIntel	GenuineIntel	GenuineIntel	GenuineIntel
SSE	1	1	1	1	1
SSE2	1	1	1	1	1
SSE3	1	1	1	1	1
SSSE3	1	1	1	1	1
SSE4_1	0	1	1	1	1
SSE4_2	0	0	1	1	1
POPCNT	0	0	1	1	1
AVX	0	0	1	1	1
F16C	0	0	0	1	1
FMA	0	0	0	1	1
AVX2	0	0	0	1	1
BMI1	0	0	0	1	1
BMI2	0	0	0	1	1
LZCNT	0	0	0	1	1
MOVBE	0	0	0	1	1

16.2 数据操作指令

x86-AVX 包含各种增强的数据操作指令，既可用于组合浮点数，也可用于组合整型操作数。这些指令中，许多是替代现有某个 x86-SSE 指令或一段指令序列。增强的数据处理包括广播、混合、排列和收集操作。本节将通过示例代码演示每组典型指令的用法。

16.2.1 数据广播

第一个示例程序 AvxBroadcast 将演示如何使用 x86-AVX 的整数和浮点数广播指令。广播指令将单个数据值复制到目标操作数的每个元素。这些指令通常用于创建组合常量值。清
[447] 单 16-3 和清单 16-4 分别列出了示例程序 AvxBroadcast 的 C++ 和汇编语言源代码。

清单 16-3 AvxBroadcast.cpp

```
#include "stdafx.h"
#include "XmmVal.h"
#include "YmmVal.h"
#include <memory.h>
#define _USE_MATH_DEFINES
#include <math.h>

// 下面这个 enum 值的顺序，必须和 AvxBroadcast_.asm 中定义的一致
enum Bcop : unsigned int
{
    Byte, Word, Dword, Qword
};

extern "C" void AvxBroadcastIntegerYmm_(YmmVal* des, const XmmVal* src, Bcop op);
extern "C" void AvxBroadcastFloat_(YmmVal* des, float val);
extern "C" void AvxBroadcastDouble_(YmmVal* des, double val);

void AvxBroadcastInteger(void)
{
    char buff[512];
    __declspec(align(16)) XmmVal src;
    __declspec(align(32)) YmmVal des;

    memset(&src, 0, sizeof(XmmVal));

    src.i16[0] = 42;
    AvxBroadcastIntegerYmm_(&des, &src, Bcop::Word);

    printf("\nResults for AvxBroadcastInteger() - Bcop::Word\n");
    printf("src    %s\n", src.ToString_i16(buff, sizeof(buff)));
    printf("des lo %s\n", des.ToString_i16(buff, sizeof(buff), false));
    printf("des hi %s\n", des.ToString_i16(buff, sizeof(buff), true));

    src.i64[0] = -80;
    AvxBroadcastIntegerYmm_(&des, &src, Bcop::Qword);

    printf("\nResults for AvxBroadcastInteger() - Bcop::Qword\n");
    printf("src: %s\n", src.ToString_i64(buff, sizeof(buff)));
    printf("des lo: %s\n", des.ToString_i64(buff, sizeof(buff), false));
    printf("des hi: %s\n", des.ToString_i64(buff, sizeof(buff), true));
}
[448]

void AvxBroadcastFloatingPoint(void)
{
    char buff[512];
```

```
    __declspec(align(32)) YmmVal des;

    AvxBroadcastFloat_(&des, (float)M_SQRT2);
    printf("\nResults for AvxBroadcastFloatingPoint() - float\n");
    printf("des lo: %s\n", des.ToString_r32(buff, sizeof(buff), false));
    printf("des hi: %s\n", des.ToString_r32(buff, sizeof(buff), true));

    AvxBroadcastDouble_(&des, M_PI);
    printf("\nResults for AvxBroadcastFloatingPoint() - double\n");
    printf("des lo: %s\n", des.ToString_r64(buff, sizeof(buff), false));
    printf("des hi: %s\n", des.ToString_r64(buff, sizeof(buff), true));
}

int _tmain(int argc, _TCHAR* argv[])
{
    AvxBroadcastInteger();
    AvxBroadcastFloatingPoint();
    return 0;
}
```

清单 16-4 AvxBroadcast_.asm

```
        .model flat,c
        .code

; extern "C" void AvxBroadcastIntegerYmm_(YmmVal* des, const XmmVal* src,←
Brop op);
;
; 描述：下面的函数展示了 vpbroadcastX 指令的用法
;
; 需要：AVX2

AvxBroadcastIntegerYmm_ proc
        push ebp
        mov ebp,esp

; 确保 op 是合法的
        mov eax,[ebp+16]                ;eax = op
        cmp eax,BropTableCount
        jae BadOp                       ;如果 op 不合法则跳转                    449

; 载入参数并跳转到指定的指令
        mov ecx,[ebp+8]                 ;ecx = des
        mov edx,[ebp+12]                ;ecx = src
        vmovdqa xmm0,xmmword ptr [edx]  ;xmm0-广播值 [低项]
        mov edx,[BropTable+eax*4]
        jmp edx

; 执行字节广播
BropByte:
        vpbroadcastb ymm1,xmm0
        vmovdqa ymmword ptr [ecx],ymm1
        vzeroupper
        pop ebp
        ret

; 执行字广播
BropWord:
        vpbroadcastw ymm1,xmm0
        vmovdqa ymmword ptr [ecx],ymm1
        vzeroupper
```

```
        pop ebp
        ret

;执行双字广播
BropDword:
        vpbroadcastd ymm1,xmm0
        vmovdqa ymmword ptr [ecx],ymm1
        vzeroupper
        pop ebp
        ret

;执行四字广播
BropQword:
        vpbroadcastq ymm1,xmm0
        vmovdqa ymmword ptr [ecx],ymm1
        vzeroupper
        pop ebp
        ret

BadOp:  pop ebp
        ret

; 下表定义的数值，必须和 AvxBroadcast.cpp 中定义的 enum Brop 一致

        align 4
BropTable dword BropByte, BropWord, BropDword, BropQword
BropTableCount equ ($ - BropTable) / size dword
AvxBroadcastIntegerYmm_ endp

; extern "C" void AvxBroadcastFloat_(YmmVal* des, float val);
;
;描述：下面的函数展示了 vbroadcastss 指令的用法
;
;需要：AVX

AvxBroadcastFloat_ proc
        push ebp
        mov ebp,esp

;广播 val 到 des 的所有元素
        mov eax,[ebp+8]
        vbroadcastss ymm0,real4 ptr [ebp+12]
        vmovaps ymmword ptr [eax],ymm0

        vzeroupper
        pop ebp
        ret
AvxBroadcastFloat_ endp

; extern "C" void AvxBroadcastDouble_(YmmVal* des, double val);
;
;描述：下面的函数展示了 vbroadcastsd 指令的用法
;
;需要：AVX

AvxBroadcastDouble_ proc
        push ebp
        mov ebp,esp

;广播 val 到 des 的所有元素
        mov eax,[ebp+8]
```

```
        vbroadcastsd ymm0,real8 ptr [ebp+12]
        vmovapd ymmword ptr [eax],ymm0

        vzeroupper
        pop ebp
        ret
AvxBroadcastDouble_ endp
        end
```

451

AvxBroadcast.cpp 的 C++ 文件包括两个函数，分别用来初始化整型和浮点型广播操作的测试用例（参见清单 16-3）。第一个函数 AvxBroadcastInteger 调用汇编语言函数 AvxBroadcastInteger_ 来演示针对字和四字的整数广播。第二个函数 AvxBroadcastFloat 使用两个汇编语言函数展示针对单精度和双精度浮点值的广播操作。

汇编语言文件 AvxBroadcast_asm 包含的函数执行实际的广播操作（参见清单 16-4）。名为 AvxBroadcastInteger_ 的函数演示了 vpbroadcast (b| w| d| q)（字节、字、双字和四字）指令的使用。这些指令的源操作数必须是 XMM 寄存器（低位的元素）或一个内存地址。目标操作数可以是一个 XMM 或 YMM 寄存器。除了 vpbroadcast (b| w| d| q) 指令，x86-AVX 指令集还包括一个 vbroadcasti128 指令，用于从一个内存地址广播 128 位整型数到 YMM 寄存器的低 128 位和高 128 位。

函数 AvxBroadcastFloat_ 和 AvxBroadcastDouble_ 演示了如何使用 vbroadcastss 和 vbroadcastsd（广播浮点数据）指令。这些指令的源操作数必须是一个内存地址或一个 XMM 寄存器。注意，AVX2 需要和 XMM 源操作数使用这些指令。vbroadcastss 指令的目标操作数可以是 XMM 或 YMM 寄存器；vbroadcastsd 指令的目标操作数必须是一个 YMM 寄存器。x86-AVX 指令集还支持一个 128 位浮点数广播指令，名为 vbroadcastf128。输出 16-1 显示示例程序 AvxBroadcast 的运行结果。

输出 16-1　示例程序 AvxBroadcast

```
Results for AvxBroadcastInteger() - Brop::Word
src     42      0       0       0 |       0       0       0       0
des lo  42      42      42      42 |      42      42      42      42
des hi  42      42      42      42 |      42      42      42      42

Results for AvxBroadcastInteger() - Brop::Qword
src:                -80 |                0
des lo:             -80 |              -80
des hi:             -80 |              -80

Results for AvxBroadcastFloatingPoint() - float
des lo:     1.414214    1.414214 |     1.414214    1.414214
des hi:     1.414214    1.414214 |     1.414214    1.414214

Results for AvxBroadcastFloatingPoint()   double
des lo:           3.141592653590 |           3.141592653590
des hi:           3.141592653590 |           3.141592653590
```

452

16.2.2　数据混合

数据混合操作有条件地复制两个组合源操作数到一个目标组合操作数，并使用控制值指定哪个元素被复制。下一个示例程序 AvxBlend 演示了如何使用两条 x86-AVX 混合指令操作

组合浮点数和组合整数。清单16-5和清单16-6分别列出了示例程序AvxBlend的C++和汇编语言的源代码。

清单16-5 AvxBlend.cpp

```
#include "stdafx.h"
#include "YmmVal.h"

extern "C" void AvxBlendFloat_(YmmVal* des, YmmVal* src1, YmmVal* src2,=
YmmVal* src3);
extern "C" void AvxBlendByte_(YmmVal* des, YmmVal* src1, YmmVal* src2,=
YmmVal* src3);

void AvxBlendFloat(void)
{
    char buff[256];
    const Uint32 select1 = 0x00000000;
    const Uint32 select2 = 0x80000000;
    __declspec(align(32)) YmmVal des, src1, src2, src3;

    src1.r32[0] = 100.0f;       src2.r32[0] = -1000.0f;
    src1.r32[1] = 200.0f;       src2.r32[1] = -2000.0f;
    src1.r32[2] = 300.0f;       src2.r32[2] = -3000.0f;
    src1.r32[3] = 400.0f;       src2.r32[3] = -4000.0f;
    src1.r32[4] = 500.0f;       src2.r32[4] = -5000.0f;
    src1.r32[5] = 600.0f;       src2.r32[5] = -6000.0f;
    src1.r32[6] = 700.0f;       src2.r32[6] = -7000.0f;
    src1.r32[7] = 800.0f;       src2.r32[7] = -8000.0f;

    src3.u32[0] = select2;
    src3.u32[1] = select2;
    src3.u32[2] = select1;
    src3.u32[3] = select2;
    src3.u32[4] = select1;
    src3.u32[5] = select1;
    src3.u32[6] = select2;
    src3.u32[7] = select1;

    AvxBlendFloat_(&des, &src1, &src2, &src3);

    printf("\nResults for AvxBlendFloat()\n");
    printf("src1 lo: %s\n", src1.ToString_r32(buff, sizeof(buff), false));
453 printf("src1 hi: %s\n", src1.ToString_r32(buff, sizeof(buff), true));
    printf("src2 lo: %s\n", src2.ToString_r32(buff, sizeof(buff), false));
    printf("src2 hi: %s\n", src2.ToString_r32(buff, sizeof(buff), true));
    printf("\n");
    printf("src3 lo: %s\n", src3.ToString_x32(buff, sizeof(buff), false));
    printf("src3 hi: %s\n", src3.ToString_x32(buff, sizeof(buff), true));
    printf("\n");
    printf("des lo:  %s\n", des.ToString_r32(buff, sizeof(buff), false));
    printf("des hi:  %s\n", des.ToString_r32(buff, sizeof(buff), true));
}

void AvxBlendByte(void)
{
    char buff[256];
    __declspec(align(32)) YmmVal des, src1, src2, src3;

    // 使用 vpblendvb 指令执行双字混合时，需要用到控制值
    const Uint32 select1 = 0x00000000;      // select src1
    const Uint32 select2 = 0x80808080;      // select src2
```

```
    src1.i32[0] = 100;          src2.i32[0] = -1000;
    src1.i32[1] = 200;          src2.i32[1] = -2000;
    src1.i32[2] = 300;          src2.i32[2] = -3000;
    src1.i32[3] = 400;          src2.i32[3] = -4000;
    src1.i32[4] = 500;          src2.i32[4] = -5000;
    src1.i32[5] = 600;          src2.i32[5] = -6000;
    src1.i32[6] = 700;          src2.i32[6] = -7000;
    src1.i32[7] = 800;          src2.i32[7] = -8000;

    src3.u32[0] = select1;
    src3.u32[1] = select1;
    src3.u32[2] = select2;
    src3.u32[3] = select1;
    src3.u32[4] = select2;
    src3.u32[5] = select2;
    src3.u32[6] = select1;
    src3.u32[7] = select2;

    AvxBlendByte_(&des, &src1, &src2, &src3);

    printf("\nResults for AvxBlendByte() - doublewords\n");
    printf("src1 lo: %s\n", src1.ToString_i32(buff, sizeof(buff), false));
    printf("src1 hi: %s\n", src1.ToString_i32(buff, sizeof(buff), true));
    printf("src2 lo: %s\n", src2.ToString_i32(buff, sizeof(buff), false));
    printf("src2 hi: %s\n", src2.ToString_i32(buff, sizeof(buff), true));
    printf("\n");
    printf("src3 lo: %s\n", src3.ToString_x32(buff, sizeof(buff), false));
    printf("src3 hi: %s\n", src3.ToString_x32(buff, sizeof(buff), true));
    printf("\n");
    printf("des lo:  %s\n", des.ToString_i32(buff, sizeof(buff), false));
    printf("des hi:  %s\n", des.ToString_i32(buff, sizeof(buff), true));
}

int _tmain(int argc, _TCHAR* argv[])
{
    AvxBlendFloat();
    AvxBlendByte();
    return 0;
}
```

454

清单 16-6 AvxBlend_.asm

```
        .model flat,c
        .code

; extern "C" void AvxBlendFloat_(YmmVal* des, YmmVal* src1, YmmVal* src2,=
YmmVal* src3);
;
; 描述：该函数演示使用 YMM 寄存器的 vblendvps 指令的用法
;
; 需要：AVX

AvxBlendFloat_ proc
        push ebp
        mov ebp,esp

; 载入参数
        mov eax,[ebp+12]                ;eax = ptr to src1
        mov ecx,[ebp+16]                ;ecx = ptr to src2
        mov edx,[ebp+20]                ;edx = ptr to src3

        vmovaps ymm1,ymmword ptr [eax]  ;ymm1 = src1
```

```
        vmovaps ymm2,ymmword ptr [ecx]      ;ymm2 = src2
        vmovdqa ymm3,ymmword ptr [edx]      ;ymm3 = src3

; 执行可变单精度浮点数混合
        vblendvps ymm0,ymm1,ymm2,ymm3       ;ymm0 = 混合结果
        mov eax,[ebp+8]
        vmovaps ymmword ptr [eax],ymm0      ;保存混合结果

        vzeroupper
        pop ebp
        ret
AvxBlendFloat_ endp

; extern "C" void AvxBlendByte (YmmVal* des, YmmVal* src1, YmmVal* src2,←
YmmVal* src3);
;
; 描述：该函数演示了 vpblendvb 指令的用法
;
; 需要：AVX2

AvxBlendByte_ proc
        push ebp
        mov ebp,esp

; 载入参数
        mov eax,[ebp+12]                    ;eax = ptr to src1
        mov ecx,[ebp+16]                    ;ecx = ptr to src2
        mov edx,[ebp+20]                    ;edx = ptr to src3

        vmovdqa ymm1,ymmword ptr [eax]      ;ymm1 = src1
        vmovdqa ymm2,ymmword ptr [ecx]      ;ymm2 = src2
        vmovdqa ymm3,ymmword ptr [edx]      ;ymm3 = src3

; 执行可变字节混合
        vpblendvb ymm0,ymm1,ymm2,ymm3       ;ymm0 = 混合结果
        mov eax,[ebp+8]
        vmovdqa ymmword ptr [eax],ymm0      ;保存混合结果
        vzeroupper
        pop ebp
        ret
AvxBlendByte_ endp
        end
```

AvxBlend.cpp 的 C++ 文件包含一个函数 AvxBlendFloat，它用单精度浮点值初始化 YmmVal 变量 src1 和 src2（参见清单 16-5）。它还初始化了名为 src3 的第三个 YmmVal 实例，用作混合控制值。src3 中高位双字元素的每一位，指定了相应的元素是从 src1（高位为 0）还是 src2（高位为 1）复制到目标操作数。vblendvps（Variable Blend Packed Single-Precision Floating-Point Values，可变混合组合型单精度浮点数）指令使用这三个源操作数，在汇编语言函数 AvxBlendFloat_ 中被执行。执行这个函数后，有一系列的 printf 语句来显示结果。

AvxBlend.cpp 文件还包含一个函数 AvxBlendInt32，它初始化 YmmVal 变量 src1 和 src2，展示了组合双字整数的混合操作。该函数也使用了第三个源操作数来指定哪个源操作数的元素被复制到目标操作数。变量 src1、src2 和 src3 最终被 vpblendvb（Variable Blend Packed Byte，可变混合组合型字节）指令使用，该指令在汇编语言函数 AvxBlendInt32_ 中。

AvxBlendFloat_ 函数（参见清单 16-6）首先将参数 src1、src2 和 src3 分别加载到寄存器 YMM1、YMM2 和 YMM3。然后执行 vblendvps ymm0，ymm1，ymm2，ymm3 指令被

浮点数的混合操作，结果如图 16-1 所示。vblendvps 指令及其对应的双精度 vblendvpd 指令，是要求三个源操作数的 x86-AVX 指令的例子。vblendps 和 vblendpd 指令可以使用一个立即控制数进行浮点数混合操作。



注：YMM3 寄存器的每个双字都是十六进制数值。

图 16-1 vblendvps 指令的执行

x86-AVX 包括几个可以用来进行整数混合操作的指令。vpblendw（Blend Packed Word，混合组合型字）和 vpblendd（Blend Packed Dword，混合组合型双字）指令分别对字和双字执行组合整型数混合操作。这两个指令都需要一个 8 位立即操作数来指定混合操作的控制值。

x86-AVX 指令集还包括 vpblendvb 指令，它使用一个可变控制值来混合字节。这个指令使用第三个源操作数每个字节的高位，从前两个源操作数中选择一个字节。如果使用一个合适的控制值，vpblendvb 指令也可以用来混合字、双字和四字。例如，为了混合双字，AvxBlendInt32 函数（参见清单 16-6）使用控制值 0x00000000 或 0x80808080，分别从第一个或第二个源操作数选择一个双字元素。输出 16-2 包含示例程序 AvxBlend 的结果。

输出 16-2 示例程序 AvxBlend

```
Results for AvxBlendFloat()
src1 lo:   100.000000   200.000000 |   300.000000   400.000000
src1 hi:   500.000000   600.000000 |   700.000000   800.000000
src2 lo: -1000.000000 -2000.000000 | -3000.000000 -4000.000000
src2 hi: -5000.000000 -6000.000000 | -7000.000000 -8000.000000

src3 lo: 80000000 80000000 | 00000000 80000000
src3 hi: 00000000 00000000 | 80000000 00000000
des lo:  -1000.000000 -2000.000000 |   300.000000 -4000.000000
des hi:    500.000000   600.000000 | -7000.000000   800.000000

Results for AvxBlendByte() - doubleWords
src1 lo:          100          200 |          300          400
src1 hi:          500          600 |          700          800
src2 lo:        -1000        -2000 |        -3000        -4000
src2 hi:        -5000        -6000 |        -7000        -8000

src3 lo: 00000000 00000000 | 80808080 00000000
src3 hi: 80808080 80808080 | 00000000 80808080

des lo:          100          200 |        -3000          400
des hi:        -5000        -6000 |          700        -8000
```

16.2.3 数据排列

x86-AVX 指令集包括几条数据排列指令，用来根据一个控制值重新排列组合型数据的元素。本节的示例程序 AvxPermute 解释了如何使用其中的几个指令。清单 16-7 和清单 16-8 展现了示例程序 AvxPermute 的源代码。

清单 16-7 AvxPermute.cpp

```
#include "stdafx.h"
#include "YmmVal.h"
#include <math.h>

extern "C" void AvxPermuteInt32_(YmmVal* des, YmmVal* src, YmmVal* ind);
extern "C" void AvxPermuteFloat_(YmmVal* des, YmmVal* src, YmmVal* ind);
extern "C" void AvxPermuteFloatII_(YmmVal* des, YmmVal* src, YmmVal* ind);

void AvxPermuteInt32(void)
{
    __declspec(align(32)) YmmVal des, src, ind;

    src.i32[0] = 10;        ind.i32[0] = 3;
    src.i32[1] = 20;        ind.i32[1] = 7;
    src.i32[2] = 30;        ind.i32[2] = 0;
    src.i32[3] = 40;        ind.i32[3] = 4;
    src.i32[4] = 50;        ind.i32[4] = 6;
    src.i32[5] = 60;        ind.i32[5] = 5;
    src.i32[6] = 70;        ind.i32[6] = 1;
    src.i32[7] = 80;        ind.i32[7] = 2;

458 AvxPermuteInt32_(&des, &src, &ind);

    printf("\nResults for AvxPermuteInt32()\n");
    for (int i = 0; i < 8; i++)
    {
        printf("des[%d]: %5d  ", i, des.i32[i]);
        printf("ind[%d]: %5d  ", i, ind.i32[i]);
        printf("src[%d]: %5d  ", i, src.i32[i]);
        printf("\n");
    }
}

void AvxPermuteFloat(void)
{
    __declspec(align(32)) YmmVal des, src, ind;

    // src1 indices must be between 0 and 7.
    src.r32[0] = 800.0f;        ind.i32[0] = 3;
    src.r32[1] = 700.0f;        ind.i32[1] = 7;
    src.r32[2] = 600.0f;        ind.i32[2] = 0;
    src.r32[3] = 500.0f;        ind.i32[3] = 4;
    src.r32[4] = 400.0f;        ind.i32[4] = 6;
    src.r32[5] = 300.0f;        ind.i32[5] = 5;
    src.r32[6] = 200.0f;        ind.i32[6] = 1;
    src.r32[7] = 100.0f;        ind.i32[7] = 2;

    AvxPermuteFloat_(&des, &src, &ind);

    printf("\nResults for AvxPermuteFloat()\n");
    for (int i = 0; i < 8; i++)
    {
```

```
        printf("des[%d]: %8.1f  ", i, des.r32[i]);
        printf("ind[%d]: %8d  ", i, ind.i32[i]);
        printf("src[%d]: %8.1f  ", i, src.r32[i]);
        printf("\n");
    }
}

void AvxPermuteFloatII(void)
{
    __declspec(align(32)) YmmVal des, src, ind;

    // 低行
    src.r32[0] = sqrt(10.0f);       ind.i32[0] = 3;
    src.r32[1] = sqrt(20.0f);       ind.i32[1] = 2;
    src.r32[2] = sqrt(30.0f);       ind.i32[2] = 2;
    src.r32[3] = sqrt(40.0f);       ind.i32[3] = 0;

    // 高行
    src.r32[4] = sqrt(50.0f);       ind.i32[4] = 1;
    src.r32[5] = sqrt(60.0f);       ind.i32[5] = 3;
    src.r32[6] = sqrt(70.0f);       ind.i32[6] = 3;
    src.r32[7] = sqrt(80.0f);       ind.i32[7] = 2;

    AvxPermuteFloatII_(&des, &src, &ind);

    printf("\nResults for AvxPermuteFloatII()\n");
    for (int i = 0; i < 8; i++)
    {
        if (i == 0)
            printf("Lower lane\n");
        else if (i == 4)
            printf("Upper lane\n");

        printf("des[%d]: %8.4f  ", i, des.r32[i]);
        printf("ind[%d]: %8d  ", i, ind.i32[i]);
        printf("src[%d]: %8.4f  ", i, src.r32[i]);
        printf("\n");
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    AvxPermuteInt32();
    AvxPermuteFloat();
    AvxPermuteFloatII();
    return 0;
}
```

清单 16-8 AvxPermute_asm

```
        .model flat,c
        .code

; extern "C" void AvxPermuteInt32_(YmmVal* des, YmmVal* src, YmmVal* ind);
;
; 描述：该函数演示了如何使用 vpermd 指令
;
; 需要：AVX2

AvxPermuteInt32_ proc
        push ebp
        mov ebp,esp
```

459

460

```
; 载入参数值
        mov eax,[ebp+8]                 ;eax = ptr to des
        mov ecx,[ebp+12]                ;ecx = ptr to src
        mov edx,[ebp+16]                ;edx = ptr to ind

; 进行双字排列
        vmovdqa ymm1,ymmword ptr [edx]  ;ymm1 = ind
        vpermd ymm0,ymm1,ymmword ptr [ecx]
        vmovdqa ymmword ptr [eax],ymm0  ;保存结果

        vzeroupper
        pop ebp
        ret
AvxPermuteInt32_ endp

; extern "C" void AvxPermuteFloat_(YmmVal* des, YmmVal* src, YmmVal* ind);
;
; 描述：该函数演示了如何使用 vpermps 指令
;
; 需要：AVX2

AvxPermuteFloat_ proc
        push ebp
        mov ebp,esp

; 载入参数值
        mov eax,[ebp+8]                 ;eax = ptr to des
        mov ecx,[ebp+12]                ;ecx = ptr to src
        mov edx,[ebp+16]                ;edx = ptr to ind

; 进行单精度浮点数排列
        vmovdqa ymm1,ymmword ptr [edx]  ;ymm1 = ind
        vpermps ymm0,ymm1,ymmword ptr [ecx]
        vmovaps ymmword ptr [eax],ymm0  ;保存结果

        vzeroupper
        pop ebp
        ret
AvxPermuteFloat_ endp

; extern "C" void AvxPermuteFloatII_(YmmVal* des, YmmVal* src, YmmVal* ind);
;
; 描述：演示了如何使用 vpermilps 指令
;
461 ; 需要：AVX2

AvxPermuteFloatII_ proc
        push ebp
        mov ebp,esp

;载入参数值
        mov eax,[ebp+8]                 ;eax = ptr to des
        mov ecx,[ebp+12]                ;ecx = ptr to src
        mov edx,[ebp+16]                ;edx = ptr to ind

;执行行从单精度浮点数排列，注意 vpermilps 的第二个源操作数指定了索引
        vmovdqa ymm1,ymmword ptr [ecx]  ;ymm1 = src
        vpermilps ymm0,ymm1,ymmword ptr [edx]
        vmovaps ymmword ptr [eax],ymm0  ;save result

        vzeroupper
        pop ebp
```

```
        ret
AvxPermuteFloatIl_ endp
        end
```

源文件 AvxPermute.cpp（参见清单 16-7）包括一个名为 AvxPermuteInt32 的函数，它初始化测试用例来演示组合双字整数的排列方法。变量 ind 包含了一组索引，指定 src 的哪个元素复制到 des。例如，声明 ind.i32[0]= 3 意味着排列操作应该执行 des.i32[0]= src.i32[3]。每个 ind 索引值必须介于 0 和 7 之间。ind 的每个索引可以使用多次，以将 src 的一个元素拷贝到 des 的多个位置。函数 AvxPermuteFloat 类似于 AvxPermuteInt32，不过它包含的是一个处理组合单精度浮点数排列的测试用例。

源文件 AvxPermute.cpp 还包含一个名为 AvxPermuteFloatIl 的函数，它初始化 YmmVal 变量 src 和 ind，目的是演示行内（in-lane）排列组合单精度浮点数。行内排列使用两个独立的 128 位行（lane）来完成操作。行内排列的控制索引必须在 0 和 3 之间，并且每行都需要有自己的一组索引。

汇编语言文件 AvxPermute_asm（参见清单 16-8）含 AvxPermuteInt32_ 和 AvxPermuteFloat_ 函数。这些函数使用 x86-AVX 指令 vpermd 和 vpermps 对组合双字整数和单精度浮点数进行排列。这些指令都要求第一个源操作数包含控制索引，第二个源操作数包含要排列的组合数据值。AvxPermute_asm 文件还包含 AvxPermuteFloatIl_ 函数，它演示了行内排列指令 vpermilps 的使用。注意，对于这个指令，第一个源操作数包含了要排列的组合型数据值，第二个源操作数包含控制索引。输出 16-3 展示了示例程序 AvxPermute 的运行结果。 462

输出 16-3　示例程序 AvxPermuteResults for AvxPermuteInt32()

```
Results for AvxPermuteInt32()
des[0]:    40  ind[0]:     3  src[0]:    10
des[1]:    80  ind[1]:     7  src[1]:    20
des[2]:    10  ind[2]:     0  src[2]:    30
des[3]:    50  ind[3]:     4  src[3]:    40
des[4]:    70  ind[4]:     6  src[4]:    50
des[5]:    70  ind[5]:     6  src[5]:    60
des[6]:    20  ind[6]:     1  src[6]:    70
des[7]:    30  ind[7]:     2  src[7]:    80

Results for AvxPermuteFloat()
des[0]:   500.0  ind[0]:     3  src[0]:   800.0
des[1]:   100.0  ind[1]:     7  src[1]:   700.0
des[2]:   800.0  ind[2]:     0  src[2]:   600.0
des[3]:   400.0  ind[3]:     4  src[3]:   500.0
des[4]:   200.0  ind[4]:     6  src[4]:   400.0
des[5]:   200.0  ind[5]:     6  src[5]:   300.0
des[6]:   700.0  ind[6]:     1  src[6]:   200.0
des[7]:   600.0  ind[7]:     2  src[7]:   100.0

Results for AvxPermuteFloatIl()
Lower lane
des[0]:   6.3246  ind[0]:     3  src[0]:   3.1623
des[1]:   5.4772  ind[1]:     2  src[1]:   4.4721
des[2]:   5.4772  ind[2]:     2  src[2]:   5.4772
des[3]:   3.1623  ind[3]:     0  src[3]:   6.3246
Upper lane
des[4]:   7.7460  ind[4]:     1  src[4]:   7.0711
des[5]:   8.9443  ind[5]:     3  src[5]:   7.7460
des[6]:   8.9443  ind[6]:     3  src[6]:   8.3666
des[7]:   8.3666  ind[7]:     2  src[7]:   8.9443
```

16.2.4 数据收集

本节的最后一个示例程序 AvxGather 将演示 x86-AVX 数据收集指令的使用方法。收集指令有条件地从一个内存数组拷贝元素到 XMM 或 YMM 寄存器。收集指令需要一组索引和一个控制掩码，指定拷贝数组中的哪些元素。清单 16-9 和清单 16-10 给出了示例程序
463 AvxGather 的 C++ 和汇编语言源代码。

清单 16-9 AvxGather.cpp

```
#include "stdafx.h"
#include "XmmVal.h"
#include "YmmVal.h"
#include <stdlib.h>

extern "C" void AvxGatherFloat_(YmmVal* des, YmmVal* indices, YmmVal* mask,↵
const float* x);
extern "C" void AvxGatherI64_(YmmVal* des, XmmVal* indices, YmmVal* mask,↵
const Int64* x);

void AvxGatherFloatPrint(const char* msg, YmmVal& des, YmmVal& indices,↵
YmmVal& mask)
{
    printf("\n%s\n", msg);

    for (int i = 0; i < 8; i++)
    {
        printf("ElementID: %d  ", i);
        printf("des: %8.1f  ", des.r32[i]);
        printf("indices: %4d  ", indices.i32[i]);
        printf("mask: 0x%08X\n", mask.i32[i]);
    }
}

void AvxGatherI64Print(const char* msg, YmmVal& des, XmmVal& indices,↵
YmmVal& mask)
{
    printf("\n%s\n", msg);

    for (int i = 0; i < 4; i++)
    {
        printf("ElementID: %d  ", i);
        printf("des: %8lld  ", des.i64[i]);
        printf("indices: %4d  ", indices.i32[i]);
        printf("mask: 0x%016llX\n", mask.i64[i]);
    }
}

void AvxGatherFloat(void)
{
    const int merge_no = 0;
    const int merge_yes = 0x80000000;
    const int n = 15;
    float x[n];
    __declspec(align(32)) YmmVal des;
    __declspec(align(32)) YmmVal indices;
464 __declspec(align(32)) YmmVal mask;

    // 初始化测试数组
    srand(22);
    for (int i = 0; i < n; i++)
        x[i] = (float)(rand() % 1000);
```

```
    // 初始化 des
    for (int i = 0; i < 8; i++)
        des.r32[i] = -1.0f;

    // 初始化索引
    indices.i32[0] = 2;
    indices.i32[1] = 1;
    indices.i32[2] = 6;
    indices.i32[3] = 5;
    indices.i32[4] = 4;
    indices.i32[5] = 13;
    indices.i32[6] = 11;
    indices.i32[7] = 9;

    // 初始化掩码值
    mask.i32[0] = merge_yes;
    mask.i32[1] = merge_yes;
    mask.i32[2] = merge_no;
    mask.i32[3] = merge_yes;
    mask.i32[4] = merge_yes;
    mask.i32[5] = merge_no;
    mask.i32[6] = merge_yes;
    mask.i32[7] = merge_yes;

    printf("\nResults for AvxGatherFloat()\n");
    printf("Test array\n");
    for (int i = 0; i < n; i++)
        printf("x[%02d]: %6.1f\n", i, x[i]);
    printf("\n");

    const char* s1 = "Values BEFORE call to AvxGatherFloat_()";
    const char* s2 = "Values AFTER call to AvxGatherFloat_()";

    AvxGatherFloatPrint(s1, des, indices, mask);
    AvxGatherFloat_(&des, &indices, &mask, x);
    AvxGatherFloatPrint(s2, des, indices, mask);
}

void AvxGather164(void)
{
    const Int64 merge_no = 0;
    const Int64 merge_yes = 0x8000000000000000LL;
    const int n = 15;                                                    465
    Int64 x[n];
    __declspec(align(32)) YmmVal des;
    __declspec(align(16)) XmmVal indices;
    __declspec(align(32)) YmmVal mask;

    // 初始化测试数组
    srand(36);
    for (int i = 0; i < n; i++)
        x[i] = (Int64)(rand() % 1000);

    // 初始化 des
    for (int i = 0; i < 4; i++)
        des.i64[i] = -1;

    // 初始化索引和掩码
    indices.i32[0] = 2;
    indices.i32[1] = 7;
    indices.i32[2] = 9;
    indices.i32[3] = 12;
```

```
    mask.i64[0] = merge_yes;
    mask.i64[1] = merge_yes;
    mask.i64[2] = merge_no;
    mask.i64[3] = merge_yes;

    printf("\nResults for AvxGatherI64()\n");
    printf("Test array\n");
    for (int i = 0; i < n; i++)
        printf("x[%02d]: %8lld\n", i, x[i]);
    printf("\n");

    const char* s1 = "Values BEFORE call to AvxGatherI64 ()";
    const char* s2 = "Values AFTER call to AvxGatherI64 ()";

    AvxGatherI64Print(s1, des, indices, mask);
    AvxGatherI64_(&des, &indices, &mask, x);
    AvxGatherI64Print(s2, des, indices, mask);
}

int _tmain(int argc, _TCHAR* argv[])
{
    AvxGatherFloat();
    AvxGatherI64();
    return 0;
}
```

466

清单 16-10 AvxGather_.asm

```
        .model flat,c
        .code

; extern "C" void AvxGatherFloat_(YmmVal* des, YmmVal* indices, YmmVal*
mask, const float* x);
;
; 描述：本函数演示了如何使用 vgatherdps 指令
;
; 需要：AVX2

AvxGatherFloat_ proc
        push ebp
        mov ebp,esp
        push ebx

; 载入参数值，执行指令 vgatherdps 前，des 的内容被装载到 ymm0，用来演示控制掩码的条件作用
        mov eax,[ebp+8]                 ;eax = 指向 des
        mov ebx,[ebp+12]                ;ebx = 指向索引
        mov ecx,[ebp+16]                ;ecx = 指向掩码
        mov edx,[ebp+20]                ;edx = 指向 x
        vmovaps ymm0,ymmword ptr [eax]  ;ymm0 = des (初始值)
        vmovdqa ymm1,ymmword ptr [ebx]  ;ymm1 = 索引
        vmovdqa ymm2,ymmword ptr [ecx]  ;ymm2 = 掩码

; 进行收集操作并保存结果
        vgatherdps ymm0,[edx+ymm1*4],ymm2  ;ymm0 = 收集的元素
        vmovaps ymmword ptr [eax],ymm0     ;保存 des
        vmovdqa ymmword ptr [ebx],ymm1     ;保存索引（未变）
        vmovdqa ymmword ptr [ecx],ymm2     ;保存掩码（全部为 0）

        vzeroupper
        pop ebx
        pop ebp
```

```
        ret
AvxGatherFloat_ endp

; extern "C" void AvxGatherI64_(YmmVal* des, XmmVal* indices, YmmVal* mask,↩
const int64* x);
;
; 描述：本函数演示了如何使用 vpgatherdq 指令
;
; 需要：AVX2                                                          467

AvxGatherI64_ proc
        push ebp
        mov ebp,esp
        push ebx

; 载入参数值。注意索引被载入 XMM1 寄存器
        mov eax,[ebp+8]                 ;eax = 指向 des
        mov ebx,[ebp+12]                ;ebx = 指向索引
        mov ecx,[ebp+16]                ;ecx = 指向掩码
        mov edx,[ebp+20]                ;edx = 指向 x
        vmovdqa ymm0,ymmword ptr [eax]  ;ymm0 = des (初始值)
        vmovdqa xmm1,xmmword ptr [ebx]  ;xmm1= 索引
        vmovdqa ymm2,ymmword ptr [ecx]  ;ymm2= 掩码

; 进行收集并保存结果。注意比例因子和收集的元素数目是一致的
        vpgatherdq ymm0,[edx+xmm1*8],ymm2  ;ymm0 = 收集的元素
        vmovdqa ymmword ptr [eax],ymm0     ;保存 des
        vmovdqa xmmword ptr [ebx],xmm1     ;保存索引 (未变)
        vmovdqa ymmword ptr [ecx],ymm2     ;保存掩码 (全部为 0)

        vzeroupper
        pop ebx
        pop ebp
        ret
AvxGatherI64_ endp
        end
```

第 12 章概要介绍了 x86-AVX 的数据收集指令，其中图 12-4 展示了收集指令的执行过程。在阅读本节的示例代码和备注之前，阅读前面的介绍会有所帮助。

C++ 文件 AvxGather.cpp（参见清单 16-9）包含一个名为 AvxGatherFloat 的函数，它初始化一个测试用例来演示 vgatherdps（Gather Packed SPFP Values Using Signed Dword Index，用有符号双字索引收集组合型单精度浮点数）指令的使用方法。这个函数首先初始化一个单精度浮点数组，用作 vgatherdps 指令的源数组。然后加载必要的数组索引到 YmmVal，作为实例索引。注意，索引中的值被视为有符号整数。接下来，YmmVal 变量被初始化为掩码，以选择那些值从源数组复制到目标数组。文件 AvxGather.cpp 还包括一个名为 AvxGatherI64 的函数。该函数准备了一个测试数组、一组索引以及一个控制掩码，以演示 vpgatherdq（Gather Packed Qword Values Using Signed Dword Index，用有符号双字索引收集组合型四字）指令的用法。

清单 16-10 是示例程序 AvxGather 的汇编语言源代码。AvxGatherFloat_ 函数分别加载 des、索引和掩码到 YMM0、YMM1 和 YMM2。它也装入一个指向源数组的变量 x 到寄存器 EDX。 468

指令 vgatherdps ymm0, [EDX+ ymm1*4], ymm2 执行实际的收集操作。注意，VSIB 操作数使用的比例因子为 4，表示源数组中的每个元素大小为 4 字节。比例因子也定义

了控制掩码中元素的大小。在收集指令中使用不正确的比例因子会产生无效的返回结果。vgatherdps 指令执行后，寄存器 YMM0、YMM1、YMM2 分别保存 des、索引和掩码。汇编语言函数 AvxGatherI64_ 的功能类似于 AvxGatherFloat_ 函数。这两个函数中，特别要注意的区别是，前者使用 XMM1 寄存器保存索引。指令 vpgatherdq ymm0,[edx + xmm1 * 8],ymm2 使用的比例因子为 8，因为该指令从源数组中采集四字值。

输出 16-4 显示了示例程序 AvxGather 的运行结果。x86-AVX 收集指令在一些特殊情况下会修改包含控制掩码的源操作数寄存器。在输出中对此进行了说明。如果收集指令执行完成没有导致处理器异常，那么其控制掩码寄存器将被设置为全零。

输出 16-4 示例程序 AvxGather

```
Results for AvxGatherFloat()
Test array
x[00]:  110.0
x[01]:  808.0
x[02]:   34.0
x[03]:  542.0
x[04]:  399.0
x[05]:  649.0
x[06]:  303.0
x[07]:  653.0
x[08]:  257.0
x[09]:  427.0
x[10]:  599.0
x[11]:   70.0
x[12]:  446.0
x[13]:  852.0
x[14]:  245.0

Values BEFORE call to AvxGatherFloat_()
ElementID: 0  des:     -1.0  indices:    2  mask: 0x80000000
ElementID: 1  des:     -1.0  indices:    1  mask: 0x80000000
ElementID: 2  des:     -1.0  indices:    6  mask: 0x00000000
ElementID: 3  des:     -1.0  indices:    5  mask: 0x80000000
ElementID: 4  des:     -1.0  indices:    4  mask: 0x80000000
ElementID: 5  des:     -1.0  indices:   13  mask: 0x00000000
ElementID: 6  des:     -1.0  indices:   11  mask: 0x80000000
ElementID: 7  des:     -1.0  indices:    9  mask: 0x80000000

Values AFTER call to AvxGatherFloat_()
ElementID: 0  des:     34.0  indices:    2  mask: 0x00000000
ElementID: 1  des:    808.0  indices:    1  mask: 0x00000000
ElementID: 2  des:     -1.0  indices:    6  mask: 0x00000000
ElementID: 3  des:    649.0  indices:    5  mask: 0x00000000
ElementID: 4  des:    399.0  indices:    4  mask: 0x00000000
ElementID: 5  des:     -1.0  indices:   13  mask: 0x00000000
ElementID: 6  des:     70.0  indices:   11  mask: 0x00000000
ElementID: 7  des:    427.0  indices:    9  mask: 0x00000000

Results for AvxGatherI64()
Test array
x[00]:      156
x[01]:      446
x[02]:      988
x[03]:      748
x[04]:      731
x[05]:       87
x[06]:      109
```

```
x[07]:      207
x[08]:       43
x[09]:      890
x[10]:      528
x[11]:      686
x[12]:      710
x[13]:      125
x[14]:      255

Values BEFORE call to AvxGather164_()
ElementID: 0  des:       -1  indices:    2  mask: 0x8000000000000000
ElementID: 1  des:       -1  indices:    7  mask: 0x8000000000000000
ElementID: 2  des:       -1  indices:    9  mask: 0x8000000000000000
ElementID: 3  des:       -1  indices:   12  mask: 0x8000000000000000

Values AFTER call to AvxGather164_()
ElementID: 0  des:      988  indices:    2  mask: 0x0000000000000000
ElementID: 1  des:      207  indices:    7  mask: 0x0000000000000000
ElementID: 2  des:       -1  indices:    9  mask: 0x0000000000000000
ElementID: 3  des:      710  indices:   12  mask: 0x0000000000000000
```

16.3 融合乘加编程

融合乘加（Fused-Multiply-Add）运算在各种算法领域都有广泛应用，比如计算机图形学和信号处理。本节的示例程序 AvxFma 演示了如何使用 FMA 指令实现常见的信号处理算法。清单 16-11 和清单 16-12 分别给出示例程序 AvxFma 的 C++ 和汇编语言源代码。 470

清单 16-11 AvxFma.cpp

```
#include "stdafx.h"
#include "Avx-ma.h"
#include <stdio.h>
#include <malloc.h>
#define  USE_MATH_DEFINES
#include <math.h>
#include <stdlib.h>

bool AvxFmaInitX(float* x, UInt32 n)
{
    const float degtorad = (float)(M_PI / 180.0);
    const float t_start = 0;
    const float t_step = 0.002f;
    const UInt32 r = 4;
    const float amp[n] = {1.0f, 0.80f, 1.20f};
    const float freq[n] = {5.0f, 10.0f, 15.0f};
    const float phase[n] = {0.0f, 45.0f, 90.0f};
    float t = t_start;

    srand(97);

    for (UInt32 i = 0; i < r; i++, t += t_step)
    {
        float x_total = 0;

        for (UInt32 j = 0; j < n; j++)
        {
            float omega = 2.0f * (float)M_PI * freq[j];
            float x_temp1 = amp[j] * sin(omega * t + phase[j] * degtorad);
            float noise = (float)((rand() % 300) - 150) / 10.0f;
```

```
            float x_temp2 = x_temp1 + x_temp1 * noise / 100.0f;

            x_total += x_temp2;
        }

        x[i] = x_total;
    }

    return true;
}

void AvxFmaSmooth5Cpp(float* y, const float*x, Uint32 n, const float*
sm5_mask)
{
    for (Uint32 i = 2; i < n - 2; i++)
    {
        float sum = 0;

        sum += x[i - 2] * sm5_mask[0];
        sum += x[i - 1] * sm5_mask[1];
        sum += x[i + 0] * sm5_mask[2];
        sum += x[i + 1] * sm5_mask[3];
        sum += x[i + 2] * sm5_mask[4];
        y[i] = sum;
    }
}

void AvxFma(void)
{
    const Uint32 n = 512;
    float* x = (float*)_aligned_malloc(n * sizeof(float), 32);
    float* y_cpp = (float*)_aligned_malloc(n * sizeof(float), 32);
    float* y_a = (float*)_aligned_malloc(n * sizeof(float), 32);
    float* y_b = (float*)_aligned_malloc(n * sizeof(float), 32);
    float* y_c = (float*)_aligned_malloc(n * sizeof(float), 32);
    const float sm5_mask[] = { 0.0625f, 0.25f, 0.375f, 0.25f, 0.0625f };

    printf("Results for AvxFma\n");

    if (!AvxFmaInitX(x, n))
    {
        printf("Data initialization failed\n");
        return;
    }

    AvxFmaSmooth5Cpp(y_cpp, x, n, sm5_mask);
    AvxFmaSmooth5a_(y_a, x, n, sm5_mask);
    AvxFmaSmooth5b_(y_b, x, n, sm5_mask);
    AvxFmaSmooth5b_(y_c, x, n, sm5_mask);

    FILE* fp;
    const char* fn = "__AvxFmaRawData.csv";

    if (fopen_s(&fp, fn, "wt") != 0)
    {
        printf("File open failed (%s)\n", fn);
        return;
    }

fprintf(fp, "i, x, y_cpp, y_a, y_b, y_c, ");
fprintf(fp, "diff_ab, diff_ac, diff_bc\n");
```

```
    UInt32 num_diff_ab = 0, num_diff_ac = 0, num_diff_bc = 0;

    for (UInt32 i = 2; i < n - 2; i++)
    {
        bool diff_ab = false, diff_ac = false, diff_bc = false;

        if (y_a[i] != y_b[i])
        {
            diff_ab = true;
            num_diff_ab++;
        }

        if (y_a[i] != y_c[i])
        {
            diff_ac = true;
            num_diff_ac++;
        }

        if (y_b[i] != y_c[i])
        {
            diff_bc = true;
            num_diff_bc++;
        }

        const char* fs1 = "%15.8f, ";
        fprintf(fp, "%4u, ", i);
        fprintf(fp, fs1, x[i]);
        fprintf(fp, fs1, y_cpp[i]);
        fprintf(fp, fs1, y_a[i]);
        fprintf(fp, fs1, y_b[i]);
        fprintf(fp, fs1, y_c[i]);
        fprintf(fp, "%d, %d, %d, ", diff_ab, diff_ac, diff_bc);
        fprintf(fp, "\n");
    }

    fclose(fp);
    printf("\nRaw data saved to file %s\n", fn);
    printf("\nNumber of data point differences between\n");
    printf("  y_a and y_b: %u\n", num_diff_ab);
    printf("  y_a and y_c: %u\n", num_diff_ac);
    printf("  y_b and y_c: %u\n", num_diff_bc);
    _aligned_free(x);
    _aligned_free(y_cpp);
    _aligned_free(y_a);
    _aligned_free(y_b);
    _aligned_free(y_c);
}

int _tmain(int argc, _TCHAR* argv[])
{
    try
    {
        AvxFma();
        AvxFmaTimed();
    }

    catch (...)
    {
        printf("Unexpected exception has occurred!\n");
        printf("File: %s (_tmain)\n", __FILE__);
    }
```

473

```
    return 0;
}
```

清单 16-12 AvxFma_asm

```
        .model flat,c
        .code

; void AvxFmaSmooth5a_(float* y, const float*x, Uintsz n, const float*__
sm5_mask);
;
; 描述：本函数使用标量 SPFP 乘法和加法，对输入的数组 x 进行加权平滑变换
;
; 需要：AVX

AvxFmaSmooth5a_ proc
        push ebp
        mov ebp,esp
        push esi
        push edi

; 汇数参数值
        mov edi,[ebp+8]                 ;edi = 指向 y
        mov esi,[ebp+12]                ;esi = 指向 x
        mov ecx,[ebp+16]                ;ecx = n
474     mov eax,[ebp+20]                ;eax = 指向 sm5_mask
        add esi,8                       ;调整指针，略过开始两个元素和最后两个元素的计算
        add edi,8
        sub ecx,4
        align 16

; 对 x 中的每个元素进行平滑操作
next:   vxorps xmm6,xmm6,xmm6           ;x_total=0

; 计算 x_total+= x[i-2]*sm5_mask[0]
        vmovss xmm0,real4 ptr [esi-8]
        vmulss xmm1,xmm0,real4 ptr [eax]
        vaddss xmm6,xmm6,xmm1

; 计算 x_total+= x[i-1]*sm5_mask[1]
        vmovss xmm2,real4 ptr [esi-4]
        vmulss xmm3,xmm2,real4 ptr [eax+4]
        vaddss xmm6,xmm6,xmm3

; 计算 x_total+=x[i]*sm5_mask[2]
        vmovss xmm0,real4 ptr [esi]
        vmulss xmm1,xmm0,real4 ptr [eax+8]
        vaddss xmm6,xmm6,xmm1

; 计算 x_total+=x[i+1]*sm5_mask[3]
        vmovss xmm2,real4 ptr [esi+4]
        vmulss xmm3,xmm2,real4 ptr [eax+12]
        vaddss xmm6,xmm6,xmm3

; 计算 x_total+=x[i+2]*sm5_mask[4]
        vmovss xmm0,real4 ptr [esi+8]
        vmulss xmm1,xmm0,real4 ptr [eax+16]
        vaddss xmm6,xmm6,xmm1

; 保存 x_total
        vmovss real4 ptr [edi],xmm6
```

```
        add esi,4
        add edi,4
        sub ecx,1
        jnz @@

        pop edi
        pop esi
        pop ebp
        ret
AvxFmaSmooth5a_ endp                                                    475

; void AvxFmaSmooth5b_(float* y, const float*x, Uint32 n, const float*w
sm5_mask);
;
; 描述：本函数使用标量 SPFP 融合乘加运算，对输入的数组 x 进行加权平滑变换
;
; 需要：AVX2, FMA

AvxFmaSmooth5b_ proc
        push ebp
        mov ebp,esp
        push esi
        push edi

; 加载参数值
        mov edi,[ebp+8]                 ;edi = 指向 y
        mov esi,[ebp+12]                ;esi = 指向 x
        mov ecx,[ebp+16]                ;ecx = n
        mov eax,[ebp+20]                ;eax = 指向 sm5_mask

        add esi,8                       ;调整指针，略过对最开始两个元素和最后两个元素的计算
        add edi,8
        sub ecx,4
        align 16

; 对 x 中的每个元素进行平滑操作
@@:     vxorps xmm6,xmm6,xmm6           ;设置 x_total1 = 0
        vxorps xmm7,xmm7,xmm7           ;设置 x_total2 = 0

; 计算 x_total1=x[i-2]*sm5_mask[0]+x_total1
        vmovss xmm0,real4 ptr [esi-8]
        vfmadd231ss xmm6,xmm0,real4 ptr [eax]

; 计算 x_total2=x[i-1]*sm5_mask[1]+x_total2
        vmovss xmm2,real4 ptr [esi-4]
        vfmadd231ss xmm7,xmm2,real4 ptr [eax+4]

; 计算 x_total1=x[i]*sm5_mask[2]+x_total1
        vmovss xmm0,real4 ptr [esi]
        vfmadd231ss xmm6,xmm0,real4 ptr [eax+8]

; 计算 x_total2=x[i+1]*sm5_mask[3]+x_total2
        vmovss xmm2,real4 ptr [esi+4]
        vfmadd231ss xmm7,xmm2,real4 ptr [eax+12]                        476

; 计算 x_total1=x[i+2]*sm5_mask[4]+x_total1
        vmovss xmm0,real4 ptr [esi+8]
        vfmadd231ss xmm6,xmm0,real4 ptr [eax+16]

; 计算 final x_total 并保存结果
        vaddss xmm5,xmm6,xmm7
        vmovss real4 ptr [edi],xmm5
```

```
        add esi,4
        add edi,4
        sub ecx,1
        jnz @B

        pop edi
        pop esi
        pop ebp
        ret
AvxFmaSmooth5b_ endp

; void AvxFmaSmooth5c_(float* y, const float*x, Uint32 n, const float*↩
sm5_mask);
; 描述：本函数使用标量 SPFP 融合乘加运算，对输入的数组 x 进行加权平滑处理
;
; 需要：AVX2, FMA

AvxFmaSmooth5c_ proc
        push ebp
        mov ebp,esp
        push esi
        push edi

; 加载参数值
        mov edi,[ebp+8]                 ;edi= 指向 y
        mov esi,[ebp+12]                ;esi= 指向 x
        mov ecx,[ebp+16]                ;ecx = n
        mov eax,[ebp+20]                ;eax= 指向 sm5_mask

        add esi,8                       ;调整指针，跳过对最开始两个元素和最后两个元素的计算
        add edi,8
477     sub ecx,4
        align 16

; 对 x 中的每个元素进行平滑操作，结果保存在 y 中
@@:     vxorps xmm6,xmm6,xmm6           ;设置 x_total = 0

; 计算 x_total=x[i-2]*sm5_mask[0]+x_total
        vmovss xmm0,real4 ptr [esi-8]
        vfmadd231ss xmm6,xmm0,real4 ptr [eax]

; 计算 x_total=x[i-1]*sm5_mask[1]+x_total
        vmovss xmm0,real4 ptr [esi-4]
        vfmadd231ss xmm6,xmm0,real4 ptr [eax+4]

; 计算 x_total=x[i]*sm5_mask[2]+x_total
        vmovss xmm0,real4 ptr [esi]
        vfmadd231ss xmm6,xmm0,real4 ptr [eax+8]

; 计算 x_total=x[i+1]*sm5_mask[3]+x_total
        vmovss xmm0,real4 ptr [esi+4]
        vfmadd231ss xmm6,xmm0,real4 ptr [eax+12]

; 计算 x_total=x[i+2]*sm5_mask[4]+x_total
        vmovss xmm0,real4 ptr [esi+8]
        vfmadd231ss xmm6,xmm0,real4 ptr [eax+16]

; 保存结果
        vmovss real4 ptr [edi],xmm6

        add esi,4
        add edi,4
```

```
        sub ecx,1
        jnz @@

        pop edi
        pop esi
        pop ebp
        ret
AvxFmaSmooth5c_ endp
        end
```

平滑变换常用来降低信号中存在的噪声，如图 16-2 所示。上图所示的原始信号包含大量的噪声，下图是应用平滑变换操作后的信号数据。平滑操作使用一组常量权值，对原始信号数据点及其相邻的数据点进行加权操作。图 16-3 更详细地说明了这种技术。要对每个原始数据点进行平滑操作，通常涉及连续的乘法和加法运算。这意味着数据平滑算法非常适合使用 FMA 指令来实现。

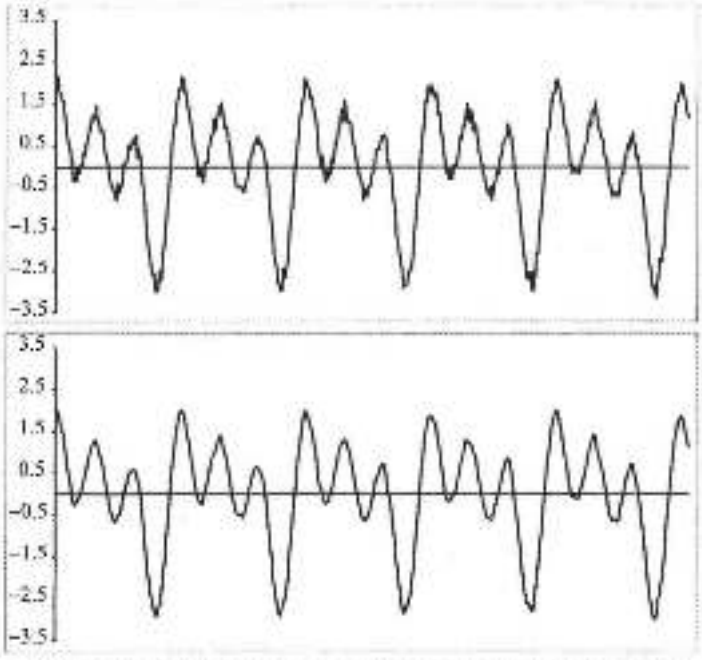


图 16-2 原始数据信号（上图）及其平滑的对应信号（下图）

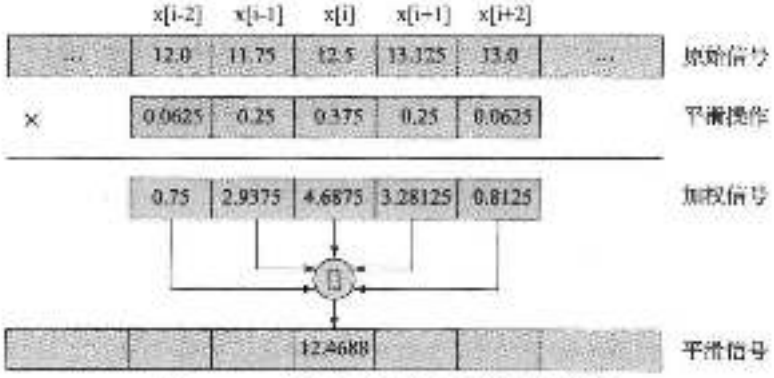


图 16-3 对原始信号数据点执行平滑操作

C++文件 AvxFma.cpp（参见清单 16-11）包含一个名为 AvxFmaInitX 的函数。这个函数构造了一个合成的原始数据信号，并将相应的数据保存在数组中，用来测试示例函数。原始数据信号包括三个正弦波形以及一些随机噪声，它对应图 16-2 上面部分的图形。AvxFmaInitX 之后是函数 AvxFmaSmooth5Cpp，它用于对原始数据信号 x 做平滑操作。注意：平滑操作没有施加到 x 中的最前两个和最后两个数据点，以使简化 C++ 语言和汇编语言的对应算法。

值得注意的另一个函数是 AvxFma.cpp 中的 AvxFma。这个函数分配信号数组所需的空间，调用各种平滑函数，并保存结果。除了函数 AvxFmaSmooth5Cpp，AvxFma 还调用平滑处理函数 AvxFmaSmooth5a_、AvxFmaSmooth5b_ 和 AvxFmaSmooth5c_。这些函数使用不同的汇编语言指令序列实现平滑算法。

清单 16-12 显示了用于上述平滑函数的汇编语言源代码。AvxFmaSmooth5a_ 函数通过使用一系列 vmulss 和 vaddss 指令实现平滑技术。需要注意的是，执行乘法时，AvxFmaSmooth5a_ 函数通过交替使用 XMM 寄存器对提高了性能。AvxFmaSmooth5b_ 函数利用 vfmadd231ss（标量单精度浮点数的融合乘加运算）指令来执行数据平滑处理。这个函数使用多个 XMM 寄存器来计算两个中间 FMA 结果，这些中间值最后通过使用 vaddss 指令求和生成最终结果。

有效利用 FMA 指令通常需要一些编程技巧，以避免无意间造成性能延迟。例如，AvxFmaSmooth5c_ 函数也调用 vfmadd231ss 指令，但每条指令使用 XMM6 作为目的寄存器，这就造成了对不良数据的依赖，妨碍了处理器充分利用其所有 FMA 执行单元（处理器执行单元将在第 21 章中讨论）。

对示例程序 AvxFma.h 中的各类平滑处理函数的时间测量，包含在表 16-2 中。需要注意的是 Smooth5b_ 函数，它使用的 FMA 指令的平均执行时间比非 FMA 函数 Smooth5a_ 快约 5%。换句话说，执行 5 个 FMA 操作明显比 5 个独立的乘法和加法快。另外请注意，因为故意向 Smooth5c_ 中添加了数据依赖，导致其执行速度比 Smooth5b_ 慢了大概 7%，虽然这两个函数都使用了 vfmadd231ss 指令。同时，Smooth5c_ 函数也比非 FMA 函数 Smooth5a_ 慢。示例程序 AvxFma 给我们的教训是，通过把独立的乘法和加法简单替换为等效 FMA 指令并
480 不能保证提高性能，而且还可能因为代码中包含对重要数据的依赖而变得更慢。

表 16-2 平滑处理函数（50 000 个数据点）的平均执行时间（单位：微秒）

CPU	C++	Smooth5a_ vmulss,vaddss	Smooth5b_ vfmadd231ss	Smooth5c_vfmadd231ss（一个 XMM 寄存器）
Intel Core i7-4770	76.4	75.7	71.2	75.1
Intel Core i7-4600U	116.7	109.1	104.5	112.4

并不奇怪的是，采用 FMA 操作进行计算的函数，与使用独立的乘法和加法运算所得到的结果通常略有不同。这可以通过示例程序 AvxFma 的运行结果得到证实（见输出 16-5），其中显示的是各种结果数据之间数据点的差值。输出文件 __AvxFmaRawData.csv 包含原始数据点与由各个算法的实现计算出的平滑值。表 16-3 包含输出文件中的若干数据点例子，显示了这种差异。对于这个示例程序，这些差异的大小无关紧要，因为平滑算法对每个数据点只执行了 5 次乘法和加法操作。但是在需要大量乘法和加法运算的场合，或者程序需要将舍入误差控制在最小的情况下，FMA 计算的优势非常明显，它将大大提高计算的精度。

输出 16-5 示例程序 AvxFmaResultsforAvxFma

```
Results for AvxFma

Raw data saved to file __AvxFmaRawData.csv

Number of data point differences between
  y_a and y_b: 178
  y_a and y_c: 178
  y_b and y_c: 0

Benchmark times saved to file __AvxFmaTimed.csv
```

表 16-3 对数据点使用各种平滑算法的例子

索引	x	C++	Smooth5a_	Smooth5b_	Smooth5c_
4	1.89155102	1.90318263	1.90318263	1.90318251	1.90318251
17	−0.323192	−0.28281462	−0.28281462	−0.28281465	−0.28281465
120	0.37265474	0.19113629	0.19113629	−0.19113627	−0.19113627
135	1.30851579	1.29426527	1.29426527	1.29426503	1.29426503
482	−2.9402566	−2.96991372	−2.96991372	−2.96991324	−2.96991324

[481]

16.4 通用寄存器指令

基于 Haswell 这样最新微架构的处理器，包含了一些新的通用寄存器指令。这些指令可以用来进行不影响标志位的乘法和移位操作。这比它们影响标志位的版本速度更快。它们也支持增强的位处理操作。本节中的示例代码演示如何执行不影响标志位的乘法和移位操作。另外还演示了如何使用几个增强的位操作指令。

16.4.1 不影响标志位的乘法和移位操作

在本小节中，我们将研究一个名为 AvxGprMulxShiftx 的示例程序，说明如何使用不影响标志位的无符号整型数乘法和移位指令。清单 16-13 和清单 16-14 分别显示了示例程序 AvxGprMulxShiftx 的 C++ 和汇编语言源代码。

清单 16-13 AvxGprMulxShiftx.cpp

```
#include "stdafx.h"
#include "MiscDefs.h"

extern "C" Uint64 AvxGprMulx_(Uint32 a, Uint32 b, Uint8 flags[2]);
extern "C" void AvxGprShiftx_(Int32 x, Uint32 count, Int32 results[3]);

void AvxGprMulx(void)
{
    const int n = 3;
    Uint32 a[n] = {64, 3200, 100000000};
    Uint32 b[n] = {1001, 12, 250000000};

    printf("Results for AvxGprMulx()\n");
    for (int i = 0; i < n; i++)
    {
        Uint8 flags[2];
        Uint64 c = AvxGprMulx_(a[i], b[i], flags);

        printf("Test case %d\n", i);
        printf("  a: %u  b: %u  c: %llu\n", a[i], b[i], c);
```

```
        printf("  status flags before mulx: 0x%02X\n", flags[0]);
        printf("  status flags after mulx:  0x%02X\n", flags[1]);
    }
}

void AvxGprShiftx(void)
{
    const int n = 4;
    Int32 x[n] = { 0x00000008, 0x80000080, 0x00000040, 0xfffffc10 };
    Uint32 count[n] = { 2, 5, 3, 4 };
    printf("\nResults for AvxGprShiftx()\n");
    for (int i = 0; i < n; i++)
    {
        Int32 results[3];

        AvxGprShiftx_(x[i], count[i], results);
        printf("Test case %d\n", i);
        printf("  x:    0x%08X (%11d) count: %u\n", x[i], x[i], count[i]);
        printf("  sarx: 0x%08X (%11d)\n", results[0], results[0]);
        printf("  shlx: 0x%08X (%11d)\n", results[1], results[1]);
        printf("  shrx: 0x%08X (%11d)\n", results[2], results[2]);
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    AvxGprMulx();
    AvxGprShiftx();
    return 0;
}
```

清单 16-14 AvxGprMulxShiftx_.asm

```
        .model flat,c
        .code

; extern "C" Uint64 AvxGprMulx_(Uint32 a, Uint32 b, Uint8 flags[2]);
;
; 描述：本函数演示指令 mulx 的用法，它是一个不影响标志位的无符号修整乘法指令
;
; 需要：BMI2

AvxGprMulx_ proc
        push ebp
        mov ebp,esp

; 执行 mulx 前，保存一份标志寄存器的状态备份
        mov ecx,[ebp+16]
        lahf
        mov byte ptr [ecx],ah

; 执行不影响标志位的无符号乘法运算，mulx 指令包含两个源操作数，即显式源操作数
; [ebp+8] 和隐式源操作数 edx，乘积为 64 位值，存放在寄存器对 edx:eax 中
        mov edx,[ebp+12]                ;edx = b
        mulx edx,eax,[ebp+8]            ;edx:eax = [ebp+8] * edx

; 执行完 mulx 指令，保存一份标志寄存器的状态备份
        push eax
        lahf
        mov byte ptr [ecx+1],ah
        pop eax
```

```
        pop ebp
        ret
AvxGprMulx  endp

; extern "C" void AvxGprShiftx_(Int32 x, UInt32 count, Int32 results[3]);
;
; 描述：本函数演示不影响标志位的移位指令 sarx、shlx 和 shrx 的用法
;
; 需要：BMI2

AvxGprShiftx_ proc
        push ebp
        mov ebp,esp

; 加载参数并执行移位操作。注意每条移位指令都需要三个操作数：DesOp、
; SrcOp 和 CountOp
        mov ecx,[ebp+12]            ;ecx = 移位计数
        mov edx,[ebp+16]            ;edx = 指向结果

        sarx eax,[ebp+8],ecx        ;算术右移
        mov [edx],eax
        shlx eax,[ebp+8],ecx        ;逻辑左移
        mov [edx+4],eax
        shrx eax,[ebp+8],ecx        ;逻辑右移
        mov [edx+8],eax

        pop ebp
        ret
AvxGprShiftx_ endp
        end
```

AvxGprMulxShiftx.cpp 文件（见清单 16-13）包含两个函数，分别为 AvxGprMulx 和 AvxGprShiftx。这两个函数分别准备了测试用例，用来演示不影响标志位的乘法和移位操作。函数 AvxGprMulx 中，在每次不影响标志位乘法操作之前和之后，flags 数组都保存了 EFLAGS 中的状态位。

清单 16-14 列出了示例程序 AvxGprMulxShiftx 的汇编代码。该函数使用 mulx edx, eax, [ebp+8]（不影响标志位的无符号乘法）指令来执行不影响标志位的乘法。该指令将内存地址 [ebp+8] 中的数值与隐式操作数 EDX 相乘，所得 64 位无符号乘积存入寄存器对 EDX:EAX 中。请注意，lahf（把状态标志存入 AH 寄存器）指令用于验证在调用 mulx 前后 EFLAGS 状态位是否被修改。 484

AvxGprShift_ 函数包含 sarx、shlx 和 shrx（不影响标志位的移位操作）指令的示例，这些指令对第一个源操作数进行移位操作，移位的数目由第二个源操作数指定。移位操作的结果保存到目标操作数中。输出 16-6 给出了示例程序 AvxGprMulxShiftx 的运行结果。

输出 16-6　示例程序 AvxGprMulxShiftX

```
Results for AvxGprMulx()
Test case 0
  a: 64  b: 1001  c: 64064
  status flags before mulx: 0x46
  status flags after mulx:  0x46
Test case 1
  a: 3200  b: 12  c: 38400
  status flags before mulx: 0x93
  status flags after mulx:  0x93
```

```
Test case 2
  a: 100000000  b: 250000000  c: 2500000000000000
  status flags before mulx: 0x97
  status flags after mulx:  0x97

Results for AvxGprShiftLx()
Test case 0
  x:    0x00000008 (          8) count: 2
  sarx: 0x00000002 (          2)
  shlx: 0x00000020 (         32)
  shrx: 0x00000002 (          2)
Test case 1
  x:    0x80000080 (-2147483520) count: 5
  sarx: 0xFC000004 (  -67108860)
  shlx: 0x00001000 (       4096)
  shrx: 0x04000004 (   67108868)
Test case 2
  x:    0x00000040 (         64) count: 3
  sarx: 0x00000008 (          8)
  shlx: 0x00000200 (        512)
  shrx: 0x00000008 (          8)
Test case 3
  x:    0xFFFFFC10 (      -1008) count: 4
  sarx: 0xFFFFFFC1 (        -63)
  shlx: 0xFFFFC100 (     -16128)
  shrx: 0x0FFFFFC1 (  268435393)
```

485

16.4.2 增强的位操作

本章最后的示例程序 AvxGprBitManip 演示如何使用一些新的位操作指令，同时演示了另一种访问栈上参数的方法。清单 16-15 和清单 16-16 为示例程序 AvxGprBitManip 的源代码。

清单 16-15 AvxGprBitManip.cpp

```
#include "stdafx.h"
#include "MiscDefs.h"

extern "C" void AvxGprCountZeroBits_(Uint32 x, Uint32* lzcnt, Uint32* tzcnt);
extern "C" Uint32 AvxGprBextr_(Uint32 x, Uint8 start, Uint8 length);
extern "C" Uint32 AvxGprAndNot_(Uint32 x, Uint32 y);

void AvxGprCountZeroBits(void)
{
    const int n = 5;
    Uint32 x[n] = { 0x001000008, 0x00008000, 0x80000000, 0x00000001, 0 };

    printf("\nResults for AvxGprCountZeroBits()\n");
    for (int i = 0; i < n; i++)
    {
        Uint32 lzcnt, tzcnt;

        AvxGprCountZeroBits_(x[i], &lzcnt, &tzcnt);
        printf("x: 0x%08X  ", x[i]);
        printf("lzcnt: %2u  ", lzcnt);
        printf("tzcnt: %2u\n", tzcnt);
    }
}
```

```
void AvxGprExtractBitField(void)
{
    const int n = 3;
    Uint32 x[n] = { 0x12345678, 0x80808080, 0xfedcba98 };
    Uint8 start[n] = { 4, 7, 24 };
    Uint8 len[n] = { 16, 9, 8 };

    printf("\nResults for AvxGprExtractBitField()\n");
    for (int i = 0; i < n; i++)
    {
        Uint32 bextr = AvxGprBextr_(x[i], start[i], len[i]);

        printf("x: 0x%08X  ", x[i]);
        printf("start: %2u  ", start[i]);
        printf("len:  %2u  ", len[i]);
        printf("bextr: 0x%08X\n", bextr);
    }
}

void AvxGprAndNot(void)
{
    const int n = 3;
    Uint32 x[n] = { 0xf000000f, 0xff00ff00, 0xaaaaaaaa };
    Uint32 y[n] = { 0x12345678, 0x12345678, 0xffaa5500 };

    printf("\nResults for AvxGprAndNot()\n");
    for (int i = 0; i < n; i++)
    {
        Uint32 andn = AvxGprAndNot_(x[i], y[i]);
        printf("x: 0x%08X  y: 0x%08X  z: 0x%08X\n", x[i], y[i], andn);
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    AvxGprCountZeroBits();
    AvxGprExtractBitField();
    AvxGprAndNot();
    return 0;
}
```

486

清单 16-16 AvxGrpBitManip_.asm

```
        .model flat,c
        .code

; extern "C" void AvxGprCountZeroBits (Uint32 x, Uint32* lzcnt, Uint32* tzcnt);
;
; 描述：本函数演示了指令 lzcnt 和 tzcnt 的用法
;
; 需要：BMI1,LZCNT

AvxGprCountZeroBits_ proc
        mov eax,[esp+4]                 ;eax = x

        lzcnt ecx,eax                   ;计算前导 0 的个数
        mov edx,[esp+8]
        mov [edx],ecx                   ;保存结果
        tzcnt ecx,eax                   ;计算后缀 0 的个数
        mov edx,[esp+12]
        mov [edx],ecx                   ;保存结果
```

487

```
        ret
AvxGprCountZeroBits_ endp

; extern "C" UInt32 AvxGprBextr_(UInt32 x, UInt8 start, UInt8 length);
;
; 描述：本函数演示 bextr 指令的用法
;
; 需要：BMI1

AvxGprBextr_ proc
        mov cl,[esp+8]                  ;cl= 起始位置
        mov ch,[esp+12]                 ;ch= 需提取的位域长度
        bextr eax,[esp+4],ecx           ;eax = 提取的位域
        ret
AvxGprBextr_ endp

; extern "C" UInt32 AvxGprAndNot_(UInt32 x, UInt32 y);
;
; 描述：本函数演示 andn 指令的用法
;
; 需要：BMI1

AvxGprAndNot_ proc
        mov ecx,[esp+4]
        andn eax,ecx,[esp+8]            ;eax = ~ecx & [esp+8]
        ret
AvxGprAndNot_ endp
        end
```

大多数新的位操作指令是为了提高某些专有算法的性能而引入的，比如数据加密和解密。当然，它们还可以用来在更平常的算法中简化位操作。示例程序 AvxGprBitManip 演示了如何使用那些具有广泛用途的位操作指令。

C++ 源文件 AvxGprBitManip.cpp（参见清单 16-15）包含三个简短的函数，用以测试对应的汇编语言函数。第一个函数 AvxGprCountZeroBits 初始化用于 lzcnt（计算前导 0 个数）和 tzcnt（计算后缀 0 个数）的测试数组。第二个函数为 AvxGprExtractBitField，演示了 bextr（提取位域）指令的用法。在示例程序 AvxGprBitManip 中，最后被调用的函数是

AvxGprAndNot，它准备了几个测试值来检测 andn（逻辑与或）指令。

使用上述位操作指令的函数代码在 AvxGprBitManip_asm 文件中（参见清单 16-16）。AvxGprCountZeroBits_ 函数演示了 lzcnt 和 tzcnt 指令的用法，分别计算在源操作数中前导 0 和后缀 0 的个数。同时演示了另外一种访问栈上参数的方法——使用 ESP 寄存器，而不是 EBP 寄存器。图 16-4 显示了执行指令 mov eax,[esp+4] 之前栈上的内容，它使用 ESP 寄存器作为基本指针，访问堆栈上的参数 x。这种方法适用于短叶子函数（即不调用其他函数的函数）。请注意寄存器 EBP 仍然被认为是易变寄存器，其内容必须在调用前保存。使用 ESP 代替 EBP 的缺点是参数值的偏移量是不固定的；改变 ESP 的值将导致堆栈上的任何参数值（和局部变量）的偏移量发生改变。

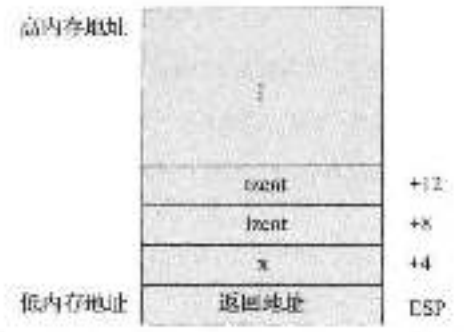


图 16-4 函数 AvxGprCountZeroBits_ 入口处的堆栈内容

函数 AvxGprBextr_ 执行了 bextr 指令，该指令从

第一个源操作数中提取一段连续的位域，起始位置和提取的长度由第二个源操作数指定。需要注意的是，AvxGprBextr 函数使用指令 mov cl,[esp+8] 和 mov ch,[esp+12] 来传递起始位置和长度，组合成 ecx，成为第二个源操作数。最后一个汇编语言函数为 AvxGprAndNot，此函数演示了如何使用 andn 指令，该指令等价于计算式 DesOp = ~SrcOp1 & SrcOp2。andn 指令经常用于简化布尔掩码运算。输出 16-7 给出了示例程序 AvxGprBitManip 的运行结果。

输出 16-7 示例程序 AvxGprBitManip

```
Results for AvxGprCountZeroBits()
x: 0x01000000  lzcnt:  7  tzcnt:  3
x: 0x00008000  lzcnt: 16  tzcnt: 15
x: 0x08000000  lzcnt:  4  tzcnt: 27
x: 0x00000001  lzcnt: 31  tzcnt:  0
x: 0x00000000  lzcnt: 32  tzcnt: 32

Results for AvxGprExtractBitField()
x: 0x12345678  start:  4  len:  16  bextr: 0x00004567
x: 0x80808080  start:  7  len:   9  bextr: 0x00000101
x: 0xFEDCBA98  start: 24  len:   8  bextr: 0x000000FE

Results for AvxGprAndNot()
x: 0xF000000F  y: 0x12345678  z: 0x02345670
x: 0xFF00FF00  y: 0x12345678  z: 0x00340078
x: 0xAAAAAAAA  y: 0xFFAA5500  z: 0x55005500
```

489

16.5 总结

在本章中，我们先学习了如何使用 cpuid 指令检测处理器是否支持 x86-SSE、x86-AVX 和其他 x86 扩展指令集的方法，还学习了一些 x86-AVX 的数据操作指令。最后，我们通过一些示例程序重点学习了新的通用寄存器指令的用法，包括不影响标志位的操作指令和增强的位操作指令。

到目前为止，本书的讨论和示例代码只关注 x86-32 的汇编语言编程。从下一章开始，我们将重点学习 x86-64 的计算资源和编程环境。

490

x86-64 核心架构

本章将探索 x86-64 核心架构的基础内容。我们将从浏览内部架构开始，包括执行单元、通用寄存器、指令操作数以及寻址方式。接下来将讨论使用汇编语言编程时必须注意的 x86-64 和 x86-32 执行环境差异。最后，将扼要地介绍 x86-64 指令集。本章的所有内容，都假设读者对前面介绍的 x86-32 核心架构和指令集已经有了基本了解。

17.1 内部架构

从应用程序的角度来看，可以从逻辑上把 x86-64 处理器分为几个不同的执行单元，如图 17-1 所示。其中，核心执行单元包含了通用寄存器、RFLAGS 寄存器以及 RIP（即指令指针）寄存器。其他的执行单元包括 x87 FPU 处理器，以及执行 SIMD 指令的计算部件。在处理器上执行的每个任务，都一定会用到核心执行单元的资源，可能会用到 x87 FPU 和 SIMD
491 部件。换句话说，前者是必要的，后者是可选的。



图 17-1 x86-64 内部架构

本节剩下的内容将更详细地介绍 x86-64 执行单元。首先集中介绍应用程序使用的执行

元素，包括通用寄存器、RFLAGS 寄存器和指针寄存器。接下来，将探讨 x86-64 的操作数和内存寻址模式。本章后面会提及如何在汇编语言函数中使用 x87 FPU 和 MMX 执行单元。第 19 章将介绍 SIMD 执行单元。

17.1.1 通用寄存器

x86-64 核心执行单元包含 16 个 64 位通用寄存器，这些通用寄存器可以用来进行算术运算、逻辑运算、以及地址计算，也可用作引用内存数据的地址指针。这些 64 位通用寄存器的低位双字、字或字节都可以被独立访问，也可以用作 32 位、16 位和 8 位的操作数。图 17-2 列出了全部的 16 个通用寄存器。 492

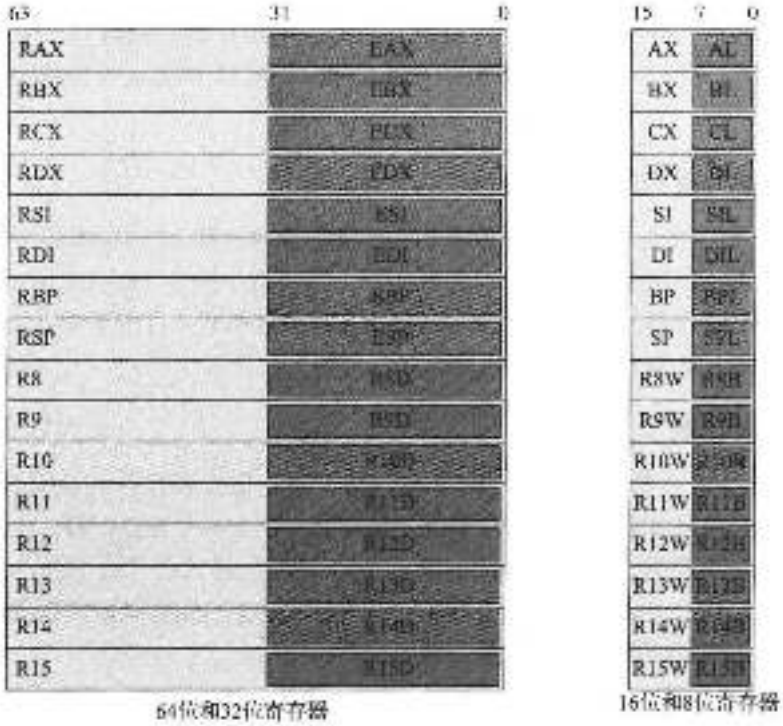


图 17-2 x86-64 通用寄存器

需要注意的是，对于 x86-64 中新引入的 8 个字节寄存器，存在一个已知的命名不一致的问题。Intel 文档把它们称为 R8L-R15L。但微软的 64 位汇编编译器则要求按照图 17-2 的命名方式进行命名。所以本书将使用 R8B-R15B 的命名方式，以保证示例代码和文字叙述一致。尽管存在一些限制，但 x86-64 平台仍然支持老的 AH、BH、CH 和 DH 寄存器的使用，我们将在本章的后面讨论这些限制。

与 x86-32 指令集类似，x86-64 指令集也对寄存器的使用做了一些规定或约束。比如，单操作数版本的 imul 指令，总是将得到的 128 位运算结果通过 RDX:RAX 寄存器对返回。而 idiv 指令，则总是从 RDX:RAX 寄存器对中加载 128 位的被除数，运算得到的 64 位的商 493
和余数被分别保存在 RAX 和 RDX 寄存器中。再如，所有的字符串指令都从 RSI 和 RDI 这两个寄存器中读取源字符串和目的字符串的地址，所有使用循环前缀的字符串指令都必须使用 RCX 寄存器作为计数器。最后，所有的 64 位旋转和移位指令，都只能使用寄存器 CL 进

行可变长的位操作。

处理器使用栈指针寄存器RSP来实现函数的调用和返回。call和ret指令在读写栈的时候，使用的是64位操作数。push和pop指令同样使用64位操作数。这就意味着栈在内存中的起始地址总应该是8字节对齐的。一些运行时环境会把栈和RSP对齐到16字节边界，以便在使用XMM寄存器进行数据传输时能自动对齐。寄存器RBP可以被用作栈帧指针，在栈帧指针被编译器优化的情况下，RBP可作为一个通用寄存器被使用。

17.1.2 RFLAGS寄存器

RFLAGS是一个64位寄存器，保存了处理器的各种状态标志和控制位。它的低32位对应于x86-32的EFLAGS寄存器，包括辅助进位标志（AF）、进位标志（CF）、溢出标志（OF）、奇偶校验标志（PF）、符号标志（SF）和零标志（ZF），这些都保持不变。RFLAGS寄存器的高32位目前保留不用。可使用指令pushfq和popfq将RFLAGS的值分别进行压栈或出栈。

17.1.3 指令指针寄存器

64位的RIP寄存器包含接下来将执行指令的地址偏移。和32位版本的寄存器（EIP）一样，RIP的值是由控制传输指令自动维护和控制的，这些指令包括call、ret、jmp和jcc。另外，处理器也利用RIP寄存器来实现一种新的内存操作数寻址模式（将在本章后面讲述），但是，在程序中直接访问RIP寄存器是被禁止且无法实现的。

17.1.4 指令操作数

大多数x86-64指令都带有一个或多个操作数，这些值将在指令的执行过程中被用到。几乎所有的指令都需要指定一个目标操作数，并需要一个或多个源操作数。大多数指令都需要显式地指定这些操作数，但也有一些x86-64指令是使用隐式操作数的。x86-64模式同样支持三种操作数类型，这一点和x86-32模式并无区别，这三种操作数类型是：立即数、寄存器和内存。表17-1列出了使用不同类型操作数的例子。

494

表17-1 基本操作数类型的例子

类型	例子	等价的C/C++语句
立即数	mov rax,42	rax = 42
	imul r12, 47	r12 *= 47
	shl r15,8	r15 <<= 8
	xor ecx,80000000h	ecx ^= 0x80000000
寄存器	sub r9b,14	r9b -= 14
	mov rax,rbx	rax = rbx
	add rbx,r10	rbx += r10
	mul rbx	rdx:rax = rax * rbx
	and r8w,0ff00h	r8w &= 0xff00
内存	mov rax,[r13]	rax = *r13
	or rcx,[rbx+rsi*8]	rcx |= *(rbx+rsi*8)
	mov qword ptr [r8],17	*(long long*)r8 = 17
	shl word ptr [r12],2	*(short*)r12 <<= 2

17.1.5 内存寻址模式

要确定一个操作数在内存中的位置，x86-64 指令集最多需要用到 4 个部分。这 4 个部分是：一个常量偏移值，一个基址寄存器，一个索引寄存器，一个放大因子。处理器利用这 4 个值来为操作数计算有效的内存地址：

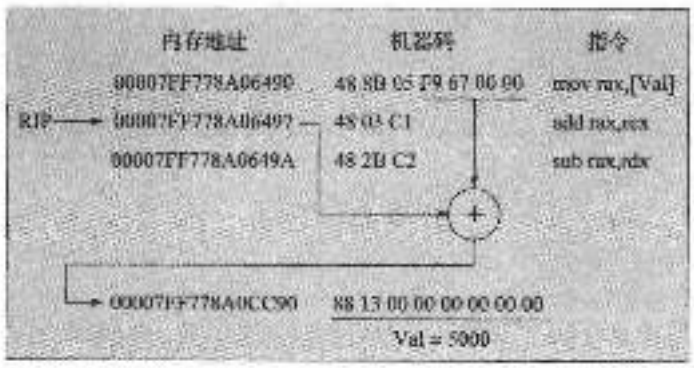
有效地址（EffectiveAddress）= 基址寄存器（BaseReg）+ 索引寄存器（IndexReg）* 放大因子（ScaleFactor）+ 偏移（Disp）

可见 x86-64 计算有效地址的方法和 x86-32 是相似的。其中，基址寄存器（BaseReg）可以是任何一个通用寄存器，索引寄存器（IndexReg）可以使用除 RSP 外的任意通用寄存器。放大因子（ScaleFactor）的取值只能是 1、2、4 和 8 之一。最后，偏移（Disp）是一个 8 位、16 位或 32 位的有符号常量值，这个常量值被编码在指令中。表 17-2 演示了不同形式的 mov 指令使用各种不同内存寻址模式的方法。请注意，在编程时，4 个值不是全都需要显式指定的。另外，最终计算得到的地址总是 64 位长度的。 495

表 17-2 x86-64 内存操作数寻址方式

寻址方式	例子
RIP + Disp	mov rax,[Val]
BaseReg	mov rax,[rbx]
BaseReg + Disp	mov rax,[rbx-16]
IndexReg * SF + Disp	mov rax,[r15*8+48]
BaseReg + IndexReg	mov rax,[rbx+r15]
BaseReg + IndexReg + Disp	mov rax,[rbx+r15-32]
BaseReg + IndexReg * SF	mov rax,[rbx+r15*8]
BaseReg + IndexReg * SF + Disp	mov rax,[rbx+r15*8+64]

x86-64 指令还使用一种新的寻址模式来计算内存中的静态操作数的有效地址。表 17-2 第一行中的指令 mov rax, [Val] 就是 RIP 相对寻址的例子。在使用 RIP 相对寻址的过程中，处理器仅需处理 RIP 寄存器的值以及包含在指令中的 32 位有符号偏移值。图 17-3 演示了此计算过程的细节。这种寻址方式，允许处理器在引用一个静态操作数时仅使用一个 32 位偏移值，而不必用 64 位值，从而节约了代码空间。同时，这也使编写位置无关代码成为可能。（图 17-3 中提到的小端字节序，其概念已在第 1 章图 1-1 中讨论过。）



注：对于 Val 的偏移值，机器码使用小端字节序。

图 17-3 使用 RIP 相对寻址的过程中有效地址计算示意 496

RIP 相对寻址的唯一局限在于，操作数所在的位置相对于 RIP 寄存器值的偏移必须在 ±2GB 以内。对绝大多数程序而言，这个局限不会产生真正的问题。32 位偏移值由汇编器在生成最终代码时自动计算出来，这样你在写汇编代码的时候，就可以直接写类似于 mov cax, [MyVal] 的指令，而完全用不着考虑最终在机器指令中出现的偏移值应该是多少。

17.2 x86-64 和 x86-32 的区别

大多数的 x86-32 指令都有一个对应的 x86-64 指令，使用的是 64 位宽的地址和操作数。x86-64 指令仍然可以只使用 8 位、16 位或 32 位宽的地址和操作数，x86-64 函数可以使用这些指令。除了 mov 指令外，其他 x86-64 模式指令中使用的立即数的最大宽度是 32 位。如果一个指令同时使用了 64 位宽的寄存器或内存操作数，以及一个 32 位的立即数，则处理器在计算前自动将 32 位立即数扩展为 64 位有符号数。

关于立即数的宽度限制，还需要一些额外的讨论，因为有些操作的正确实现需要使用特殊的指令序列。图 17-4 包含一些这样的例子，指令中同时使用了 64 位寄存器以及一个立即数。第一个例子中，mov rax, 100 指令把一个立即数加载到 RAX 寄存器中。注意，机器码只使用了 32 位数来编码立即数 100。但这个值被自动扩展为 64 位有符号整数，并保存到 RAX 中。接下来的指令 add rax, 200 也类似地先自动将 200 扩展为 64 位有符号整数，再和 RAX 进行加法运算。接下来的例子中指令 mov rcx, -2000 把一个负立即数加载到 RCX 寄存器中。此例中负数 -2000 的机器码也是 32 位的，在执行时也同样先被扩展为带符号的 64 位整数，然后保存在 RCX 中。接下来的例子中，指令 add rcx, -1000 也会自动扩展出一个 64 位的 -1000。

[497]

机器码	指令	目标操作数结果
48 C7 C0 64 00 00 00	mov rax,100	0000000000000064h
48 05 C8 00 00 00	add rax,200	000000000000012Ch
48 C7 C1 30 F8 FF FF	mov rcx,-2000	FFFFFFFFFFFFF830h
48 81 C1 E8 03 00 00	add rcx,-1000	FFFFFFFFFFFFFC18h
48 C7 C2 FF 00 00 00	mov rdx,0ffh	00000000000000FFh
48 81 CA 00 00 00 80	or rdx,80000000h	FFFFFFFF800000FFh
48 C7 C2 FF 00 00 00	mov rdx,0ffh	00000000000000FFh
49 B8 00 00 00 80 00 00 00 00	mov r8,80000000h	0000000080000000h
49 0B D0	or rdx,r8	00000000800000FFh

注：机器码中加下划线的是立即数。

图 17-4 带立即数的 64 位寄存器示例

第三个例子中，先是用指令 mov rdx, 0ffh 初始化寄存器 RDX，然后执行 or rdx, 80000000h。此时，立即数 0x80000000 将被扩展为 0xFFFFFFFF80000000，然后执行逻辑或运算。RDX 中得到最终结果，显然不是我们期望的。最后一个例子则演示了如何正确地使用 64 位立即数。前面已经提到过，只有 mov 指令能支持 64 位立即数。所以，mov r8, 80000000h 指令能正确地把 64 位数 0x0000000080000000 加载到 R8 中。这个例子最后执行

的 or rdx, r8 指令将能得到正确的结果。

立即数的 32 位宽限制同样适用于 jmp 和 call 指令，二者需指定目标地址的相对偏移值。在此情况下，jmp 或 call 指令的目标地址偏移值，必须在 RIP 值的 ±2GB 以内。程序跳转的时候，如果要超出此范围，只能使用间接操作数（比如，jmp qword ptr[FuncPtr] 或 call rax）。和 RIP 相对寻址类似，我们在此讨论的立即数宽度限制，对于大多数的程序不会产生实质性影响。

x86-32 和 x86-64 之间的另一个区别涉及 64 位寄存器的高 32 位。当有些指令使用 32 位寄存器和操作数的时候，64 位通用寄存器的高 32 位将在执行过程中清零。比如，假设寄存器 RAX 包含值 0x8000000000000000。执行指令 add eax, 10 后，将导致 RAX 的 [498]
值变为 0x000000000000000A。但是，当执行 8 位或 16 位寄存器和操作数运算时，其对应的 64 位通用寄存器的高 56 位或高 48 位却并不会被清零或修改。再假设 RAX 的当前值为 0x8000000000000000，执行指令 add al, 20 或 add ax, 40 后，RAX 的值将分别变成 0x8000000000000014 或 0x8000000000000028。

x86-64 和 x86-32 之间最后值得一提的区别与 8 位寄存器 AH、BH、CH 和 DH 有关。这几个 8 位寄存器不可以用在使用了新的 8 位寄存器（即 SIL、DIL、BPL、SPL 和 R8B-15B）的指令中。在 x86-64 程序中还可以使用原来的 8 位寄存器，如 mov ah, bl 和 add dh, bl。不过 mov ah, r8b 和 add dh, r8b 这样的指令是非法的。

17.3 指令集概览

本节将浏览 x86-64 指令集，先对基本的指令使用进行综述，然后针对 x86-64 模式下不再使用的指令进行总结。接下来再快速检视一下 x86-64 模式下全新或变化的指令。本节的最后，会讨论在 x86-64 模式下不再使用的若干计算资源。

17.3.1 基本指令使用

实际上，x86-64 指令集是对 x86-32 的逻辑扩展，即对大多数 x86-32 指令进行了升级，以支持 64 位操作数。指令集是向下兼容的，用 64 位汇编语言编写函数时，仍可以使用 8 位、16 位或 32 位的操作数。表 17-3 描述了此种情况。请注意，表中的例子涉及的指令在使用内存操作数时，都是用 64 位寄存器引用其内存地址的，因为这样就能访问到全部的 64 位有效地址空间。但你也可以在 x86-64 模式下，用 32 位寄存器引用内存操作数的内存地址（比如指令 mov r10, [eax]），其局限性就是被引用的操作数仅能位于 64 位有效地址空间的低 4GB 部分。所以，我们不推荐在 x86-64 模式下使用 32 位寄存器访问内存操作数，以免在编码时引入不必要的混乱，以及在软件测试和调试时增加不必要的复杂性。

表 17-3 使用不同长度操作数的 x86-64 指令示例

8 位	16 位	32 位	64 位
add al,bl	add ax,bx	add eax,ebx	add rax,rbx
cmp dl,[r15]	cmp dx,[r15]	cmp edx,[r15]	cmp rdx,[r15]
mul r10b	mul r10w	mul r10d	mul r10
or [r8+rdi],cl	or [r8+rdi*2],ax	or [r8+rdi*4],eax	or [r8+rdi*8],rax
shl r9b,cl	shl r9w,cl	shl r9d,cl	shl r9,cl

[499]

我们在第1章中对大部分x86-32指令的描述，同样适用于其对应的x86-64指令。Intel和AMD公司公布的指令参考手册中有关于这些x86-64指令的更多介绍，用户可以从其网站下载参考手册。

17.3.2 无效指令

一些x86-32下很少用到的指令，在x86-64模式下被放弃了。表17-4列出了这些指令。让人惊讶的是，早期的x86-64处理器在x86-64模式下竟然是不支持lahf（把标志寄存器的值加载到AH寄存器）和sahf（把AH的值保存到标志寄存器）指令的，但在x86-32模式下却支持。幸运的是，大概2006年以后，这些指令陆续在大多数的AMD和Intel处理器中恢复使用了。程序可通过测试cpuid特性标志位LAHF/SAHF来判断处理器在x86-64模式下是否支持lahf和sahf这两条指令。

表17-4 x86-64模式下的无效指令

助记符	描述
aaa	加法后ASCII调整
aad	除法前ASCII调整
aam	乘法后ASCII调整
aas	减法后ASCII调整
bound	针对边界检测验证数组索引
das	加法后十进制调整
daa	减法后十进制调整
into	若EFLAGS.OF为1中断
popa/popad	弹出所有通用寄存器
pusha/pushad	压入所有通用寄存器

17.3.3 新指令

x86-64指令集包含一些可对64位宽操作数进行运算的新指令，同时，对既存的部分指令的
500 行为进行了修改。表17-5总结了这些指令。缩写GPR代表通用寄存器（General-Purpose Register）。

表17-5 x86-64新指令

助记符	描　述
cdqe	对EAX中的双字值进行符号扩展并将结果存到RAX
cmpsb cmpsw cmpsd cmpsq	比较寄存器RSI和RDI指向的内存地址中的值，设置状态标志来指示结果
cmpxchg16b	用16字内存操作数比较RDX:RAX，根据结果进行交换
cqo	将RAX的内容符号扩展到RDX:RAX
jrcxz	如果条件RCX—0为真就转到指定的内存地址
lodsb lodsw lodsd lodsq	将寄存器RSI指向的内存地址中的值加载到AL、AX、EAX或RAX寄存器
movsb movsw movsd movsq	将寄存器RSI指定的内存地址中的值拷贝到寄存器RDI指定的内存地址
movxsd	将又并符号扩展源操作数中的双字值，并将其保存到目标操作数
pop	弹出栈顶元素。该指令将复RSP指向的内存地址的内容到指定的GPR或内存地址，RSP自动调整以反映弹出操作。该指令不能使用32位宽的操作数
popfq	弹出栈顶双字，并将低位双字保存到RFLAGS低位32位。RFLAGS的高32位被置为0。该指令不能修改RFLAGS中的保留位和特定的控制位

（续）

助记符	描　述
push	将 GPR、内存地址或立即数压入栈。RSP 自动调整以反映压入操作。该指令不能使用 32 位宽的操作数
pushfq	将 RFLAGS 压入栈
rep repe/repz repne/repnz	RCX 非 0 并指定的比较条件为真时，重复指定的字符串指令
scash scasw scasd scasq	将寄存器 RDI 指定的内存地址值与寄存器 AL、AX、EAX 或 RAX 中的值进行比较，基于比较结果设置状态标志
stosb stosw stosd stosq	将寄存器 AL、AX、EAX 或 RAX 中的内容保存到寄存器 RDI 指定的内存地址

17.3.4 不鼓励使用的资源

支持 x86-64 指令集的处理器也包括 SSE2 计算资源。这表示 x86-64 程序可以安全地使用 SSE2 的组合整型数的能力，以代替 MMX。这也意味着，x86-64 程序可以使用 SSE2 的标量型浮点数资源来替代 x86 FPU。在 x86-64 执行环境下，程序仍可以使用 MMX 和 x87 FPU 的指令集。在对遗留代码进行迁徙的过程中，这会很方便。但对于新开发的软件，我们不再推荐使用 MMX 和 x87 FPU。

17.4 总结

这一章中，我们学习了 x86-64 平台的核心架构，包括它的执行单元、通用寄存器、指令操作数和内存寻址模式。我们也学习了 x86-64 和 x86-32 这两个执行环境之间的区别，以及它们的指令集之间的关联。在第 18 章，我们将通过示例代码学习 x86-64 汇编语言编程的基础部分。

x86-64 核心编程

通过前一章，我们已经学习了 x86-64 平台的核心架构，包括执行单元、通用寄存器、指令操作数和内存寻址模式。对 x86-32 和 x86-64 的执行环境以及各自指令集之间的差别，也有了较深刻的理解。在这一章中，我们将专注于 x86-64 汇编语言编程的基础知识。

本章的内容安排如下：

- 18.1 节介绍 x86-64 汇编语言编程的基础知识，包括如何进行整型数算术运算，如何使用各种内存寻址模式，以及如何执行标量浮点数算术运算。
- 18.2 节阐述编写 x86-64 汇编语言的函数时所需注意的调用约定，以便它能被高级语言（如 C++）正确地调用。
- 18.3 节展示在 x86-64 中使用数组和文本串的一些编程技巧。

本章所有的示例代码都需运行在 x86-64 兼容的处理器和操作系统上。

18.1 x86-64 编程基础

本节将介绍 x86-64 汇编语言编程的若干要点。首先我们简要介绍从 C++ 程序调用汇编语言函数所需遵循的调用约定。然后以一个示例程序来演示如何利用 x86-64 指令集实现基本的整数算术运算。第二个示例程序对各种常用的内存寻址模式分别进行说明。最后的两个示例程序演示如何在 x86-64 函数中使用整型操作数，以及进行标量浮点数算术运算。

和 32 位编程类似，Visual C++ 64 位运行时环境为 x86-64 汇编语言函数定义了一个必须遵守的调用约定。调用约定指定处理器的每一个通用寄存器是易变的还是非易变的。它也给各个 XMM 寄存器进行了易变或非易变的分类。x86-64 汇编语言函数可以修改任一易变型寄存器的内容，但必须确保非易变寄存器的内容不被修改。表 18-1 列出了 64 位易变和非易变寄存器。Visual C++ 调用约定的其他方面将在本章中陆续地讲解，附录 B 包含了一份关于调用约定的完整总结。

[503]

表 18-1 Visual C++ 64 位易变和非易变寄存器

寄存器类型	易变寄存器	非易变寄存器
General-purpose	RAX, RCX, RDX, R8, R9, R10, R11	RBX, RSI, RDI, RBP, RSP, R12, R13, R14, R15
X86-SSE XMM	XMM0-XMM5	XMM6-XMM15

在支持 x86-AVX 的系统中，每个 YMM 寄存器的高 128 位被划分为易变的。Visual C++ 64 位程序一般很少使用 x87 FPU。如果使用的话，x86-64 汇编语言函数不需要维持 x87 FPU 寄存器栈的内容不变，也就是说，整个寄存器栈都是易变的。

和 32 位调用约定对比，Visual C++ 64 位调用约定对汇编语言函数所规定的编程要求更为严格。根据函数是叶函数还是非叶函数，编程要求也有所变化。叶函数被定义为：

- 不调用任何其他函数。
- 不修改 RSP 寄存器的值。
- 不申请任何栈空间。
- 不修改任何非易变型通用寄存器或 XMM 寄存器。
- 不使用异常处理。

x86-64 汇编语言叶函数更易于编写，但仅适合完成相对简单的计算任务。非叶函数能使用所有的 x86-64 寄存器，创建栈帧，从栈上申请空间，或调用别的函数——只要它能完全遵从调用约定所定义的函数序言和结语。本节中的示例代码包含了若干叶函数，以阐明 x86-64 汇编语言编程的基础。我们将在本章的后面学习如何创建非叶函数。

18.1.1 整数算术运算

我们将要考察的第一个示例程序叫作 IntegerArithmetic，演示如何使用 x86-64 指令集来实现基本的整型数算术运算。这个示例程序也演示了如何通过调用约定来指定由 C++ 函数 [504] 向汇编语言函数传递参数值。清单 18-1 和清单 18-2 分别列出了示例程序 Integer-Arithmetic 的 C++ 和汇编语言源代码。

清单 18-1 IntegerArithmetic.cpp

```
#include "stdafx.h"
#include "MiscDefs.h"

extern "C" Int64 IntegerAdd_(Int64 a, Int64 b, Int64 c, Int64 d, Int64 e,↩
Int64 f);
extern "C" Int64 IntegerMul_(Int8 a, Int16 b, Int32 c, Int64 d, Int8 e,↩
Int16 f, Int32 g, Int64 h);
extern "C" void IntegerDiv_(Int64 a, Int64 b, Int64 quo_rem_ab[2], Int64 c,↩
Int64 d, Int64 quo_rem_cd[2]);

void IntegerAdd(void)
{
    Int64 a = 200;
    Int64 b = 200;
    Int64 c = -300;
    Int64 d = 400;
    Int64 e = -500;
    Int64 f = 600;

    // 计算 a + b + c + d + e + f
    Int64 sum = IntegerAdd_(a, b, c, d, e, f);

    printf("\nResults for IntegerAdd\n");
    printf("a: %5lld b: %5lld c: %5lld\n", a, b, c);
    printf("d: %5lld e: %5lld f: %5lld\n", d, e, f);
    printf("sum: %lld\n", sum);
}

void IntegerMul(void)
{
    Int8 a = 2;
    Int16 b = -3;
    Int32 c = 8;
    Int64 d = 4;
    Int8 e = 3;
    Int16 f = -7;
```

```
    Int32 g = -5;
    Int64 h = 10;

    // 计算 a * b * c * d * e * f * g * h
    Int64 result = IntegerMul (a, b, c, d, e, f, g, h);

    printf("\nResults for IntegerMul\n");
    printf("a: %5d b: %5d c: %5d d: %5lld\n", a, b, c, d);
    printf("e: %5d f: %5d g: %5d h: %5lld\n", e, f, g, h);
    printf("result: %5lld\n", result);
}

void IntegerDiv(void)
{
    Int64 a = 102;
    Int64 b = 7;
    Int64 quo_rem_ab[2];
    Int64 c = 61;
    Int64 d = 9;
    Int64 quo_rem_cd[2];

    // 计算 a / b 和 c / d
    IntegerDiv (a, b, quo_rem_ab, c, d, quo_rem_cd);

    printf("\nResults for IntegerDiv\n");
    printf("a:   %5lld b:   %5lld ", a, b);
    printf("quo: %5lld rem: %5lld\n", quo_rem_ab[0], quo_rem_ab[1]);
    printf("c:   %5lld d:   %5lld ", c, d);
    printf("quo: %5lld rem: %5lld\n", quo_rem_cd[0], quo_rem_cd[1]);
}

int _tmain(int argc, _TCHAR* argv[])
{
    IntegerAdd();
    IntegerMul();
    IntegerDiv();
    return 0;
}
```

清单 18-2 IntegerArithmetic_.asm

```
        .code

; extern "C" Int64 IntegerAdd_(Int64 a, Int64 b, Int64 c, Int64 d, Int64 e,↩
Int64 f)
;
; 描述：下面的函数演示了 64 位计算机加法

IntegerAdd_ proc

; 计算参数之和
        add rcx,rdx                     ;rcx = a + b
        add r8,r9                       ;r8 = c + d
        mov rax,[rsp+40]                ;rax = e
        add rax,[rsp+48]                ;rax = e + f

        add rcx,r8                      ;rcx = a + b + c + d
        add rax,rcx                     ;rax = a + b + c + d + e + f

        ret
IntegerAdd_ endp
```

```
; extern "C" Int64 IntegerMul_(Int8 a, Int16 b, Int32 c, Int64 d, Int8 e,↵
Int16 f, Int32 g, Int64 h);
;
; 描述：下面的函数演示了 64 位带符号整数乘法运算

IntegerMul_ proc

; 计算 a * b
        movsx r10,cl                    ;r10 = sign_extend(a)
        movsx r11,dx                    ;r11 = sign_extend(b)
        imul r10,r11                    ;r10 = a * b

;计算 c * d
        movsxd rcx,r8d                  ;rcx = sign_extend(c)
        imul rcx,r9                     ;rcx = c * d

; 计算 e * f
        movsx r8,byte ptr [rsp+40]      ;r8 = sign_extend(e)
        movsx r9,word ptr [rsp+48]      ;r9 = sign_extend(f)
        imul r8,r9                      ;r8 = e * f

; 计算 g * h
        movsxd rax,dword ptr [rsp+56]   ;rax = sign_extend (g)
        imul rax,[rsp+64]               ;rax = g * h

; 计算得到最终结果
        imul r10,rcx                    ;r10 = a * b * c * d
        imul rax,r8                     ;rax = e * f * g * h
        imul rax,r10                    ;rax = final product

        ret
IntegerMul_ endp

; extern "C" void IntegerDiv_(Int64 a, Int64 b, Int64 quo_ren_ab[2],↵
Int64 c, Int64 d, Int64 quo_ren_cd[2]);
;
; 描述：下面的函数演示了 64 位带符号整数除法运算                                    507
IntegerDiv_ proc

; 计算 a/b，保存商和余数
        mov [rsp+16],rdx                ;保存 b
        mov rax,rcx                     ;rax = a
        cqo                             ;rdx:rax = sign_extend(a)
        idiv qword ptr [rsp+16]         ;rax = quo a/b, rdx = rem a/b
        mov [r8],rax                    ;保存商
        mov [r8+8],rdx                  ;保存余数

; 计算 c/d，保存商和余数
        mov rax,r9                      ;rax = c
        cqo                             ;rdx:rax = sign_extend(c)
        idiv qword ptr [rsp+40]         ;rax = quo c/d, rdx = rem c/d
        mov r10,[rsp+48]                ;r10 = 指向 quo_ren_cd
        mov [r10],rax                   ;保存商
        mov [r10+8],rdx                 ;保存余数

        ret
IntegerDiv_ endp
        end
```

C++ 源文件 IntegerArithmetic.cpp（见清单 18-1）包含的头文件 MiscDefs.h 中以 typedef

声明的形式定义了一系列不同长度的整数类型。这个头文件与 32 位 C++ 示例代码使用的是同一个。IntegerArithmetic.cpp 剩余部分包含三个简单的函数，分别用于测试变量初始化、执行函数，以及呈现结果。这几个函数的主要目的是为了演示 64 位参数传递、栈的布局以及 64 位整数算术运算。

清单 18-2 是示例程序 IntegerArithmetic 的汇编语言源代码。IntegerArithmetic_asm 源文件的开头是一个 .code 指示符，用来定义代码段的起始。x86-32 汇编源码中用到的 .model 指示符，在 64 位版本的 MASM 中已不需要了，也不再被支持。声明 IntegerAdd_ proc 定义了汇编语言函数 IntegerAdd_ 的入口点。与之对应地，在此函数的结尾处由 IntegerAdd_ endp 声明。

64 位的 Visual C++ 函数的前 4 个参数通过寄存器 RCX、RDX、R8 和 R9 传递，如有更多参数，则通过栈来传递。Visual C++ 的 64 位调用约定要求函数调用者在栈上分配 32 字节空间，给被调函数使用。这段未经初始化的栈空间称作备份空间（home area），其主要作用是给寄存器参数作暂存空间。但被调函数也可将此备份空间用于存储其他变量。需注意的是，不管传递多少个参数，调用者都必须申请 32 字节的备份空间。图 18-1 展示了在进入函数 IntegerAdd_ 时栈的布局和参数寄存器的内容。

508

函数 IntegerAdd_ 将 6 个 64 位带符号的整型参数相加作为结果返回。它的前两条指令 add rcx, rdx 和 add r8, r9 分别对应于 a+b 和 c+d 的加法运算。指令 mov rax, [rsp+40] 将参数 e 加载到寄存器 RAX，后面的指令 add rax, [rsp+48] 用来计算 e+f。接下来的两条指令 add rcx, r8 和 add rax, rcx 最终计算出 6 个参数的和。64 位汇编语言函数通过寄存器 RAX 返回一个 64 位整型数给调用者。此时，因为 RAX 已经包含了计算结果，就无须再使用 mov 指令了。IntegerAdd_ 的最后一条指令是函数返回指令 ret。

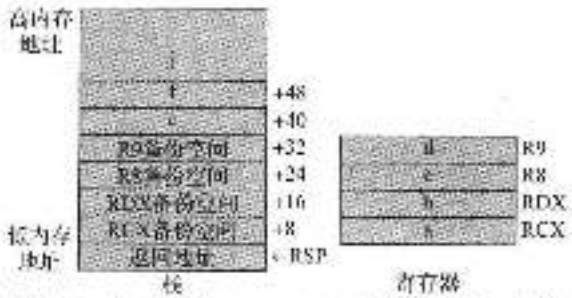


图 18-1 进入函数 IntegerAdd_ 时的栈布局和寄存器值

接下来的汇编语言函数 IntegerMul_ 演示了如何进行整数的乘法运算。图 18-2 展示了在进入函数 IntegerMul_ 时栈的布局和参数寄存器的值。请注意，不管是通过寄存器还是栈传递的参数，当长度小于 64 位时，需进行右对齐，高位空出的部分是未定义的。

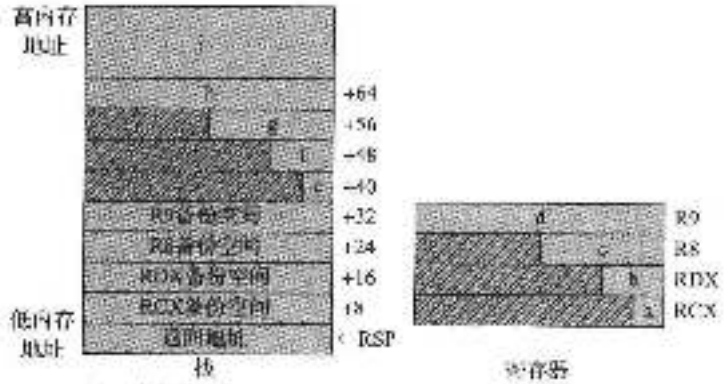


图 – 未定义

509

图 18-2 进入 IntegerMul_ 函数时的栈布局和寄存器值

函数 IntegerMul 用来计算 8 个带符号整型参数的乘积。首先是指令 movsx r10, cl 把参数 a 进行符号扩展并加载到寄存器 R10。后面的指令 movsx r11, dx 对参数 b 执行类似操作。接下来的指令 imul r10, r11 是计算 a * b。再后面的两条指令 movsxd rcx, r8d 和 imul rcx, r9 用来计算 c*d。请注意，x86-64 指令集特地定义了一个助记符（movsxd），用来实现 32 位值到 64 位寄存器的符号扩展移动。接下来用一系列的 movsx、movsxd 和 imul 指令实现了 e * f 和 g * h，作为中间结果保存。参数变量 e、f、g 和 h 都位于栈上，通过 RSP 寄存器加一个常量偏移来引用它们。最后面的三个 imul 指令，计算出最终的 64 位结果。

最后一个汇编语言函数是 IntegerDiv_，演示了如何进行 64 位带符号的整数除法运算。图 18-3 展示了在进入 IntegerDiv_ 函数时的栈布局。函数的第一条指令 mov [rsp+16], rdx 把寄存器 RDX 的值（即变量 b）保存到栈的备份空间中。后面一条指令 mov rax, rcx 把被除数 a 拷贝到寄存器 RAX。指令 cqo（Convert Quandword to Double Quadword，将四字转换为双四字）把寄存器 RAX 中的值符号扩展到寄存器对 RDX:RAX 中。接下来的指令 idiv qword ptr[rsp+16]，是用 qword ptr[rsp+16] 的值除以寄存器对 RDX:RAX 的值（即 a/b）。idiv 指令执行之后，寄存器 RAX 和 RDX 将分别保存除法运算的商和余数。这两个值保存到一个由寄存器 R8 指定的返回值数组中，R8 所对应的参数变量是 quo_rem_ab。再后面的一段指令采用类似的方式计算 c/d。输出 18-1 列出了示例程序 IntegerArithmetic 的运行结果。

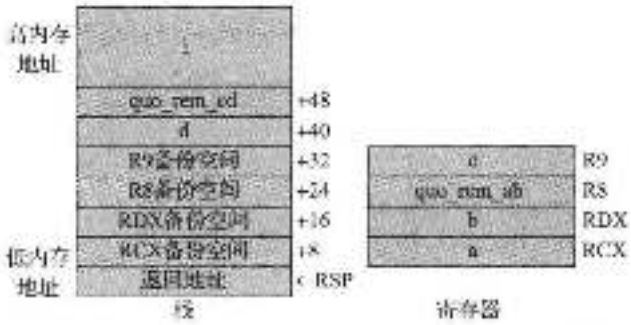


图 18-3 进入 IntegerDiv_ 函数时的栈布局和寄存器值 510

输出 18-1 示例程序 IntegerArithmetic

```
Results for IntegerAdd
a:   100 b:   200 c:  -300
d:   400 e:  -500 f:   600
sum: 500

Results for IntegerMul
a:     2 b:    -3 c:     8 d:     4
e:     3 f:    -7 g:    -5 h:    10
result: -201600

Results for IntegerDiv
a:   102 b:     7 quo:    14 rem:     4
c:    61 d:     9 quo:     6 rem:     7
```

18.1.2 内存寻址

下面的示例程序 MemoryAddressing 将对各种常用的 x86-64 内存寻址模式进行演示。

这个示例是第2章中曾介绍过的同名程序的64位版本。清单18-3和清单18-4是64位版本MemoryAddressing的实现源码。

清单18-3 MemoryAddressing.cpp

```
#include "stdafx.h"

extern "C" int NumFibVals_, FibValsSum_;
extern "C" int MemoryAddressing_(int i, int* v1, int* v2, int* v3, int* v4);

int _tmain(int argc, _TCHAR* argv[])
{
    FibValsSum_ = 0;

    for (int i = -1; i < NumFibVals_ + 1; i++)
    {
        int v1 = -1, v2 = -1, v3 = -1, v4 = -1;
        int rc = MemoryAddressing_(i, &v1, &v2, &v3, &v4);

        printf("i: %2d  rc: %2d - ", i, rc);
        printf("v1: %5d v2: %5d v3: %5d v4: %5d\n", v1, v2, v3, v4);
    }

    printf("FibValsSum_: %d\n", FibValsSum_);
    return 0;
}
```

511

清单18-4 MemoryAddressing_.asm

```
; 简单的查找表(.const数据段只读)

            .const
FibVals     dword 0, 1, 1, 2, 3, 5, 8, 13
            dword 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597
NumFibVals  dword ($ - FibVals) / sizeof dword
            public NumFibVals

            .data
FibValsSum_ dword ?             ;演示RIP-relative寻址的值
            public FibValsSum_

            .code

; extern "C" int MemoryAddressing_(int i, int* v1, int* v2, int* v3,↩
int* v4);
;
; 描述：此函数演示了用于访问内存操作数的各种寻址模式
;
; 返回：0 - 错误(无效索引)
;       1 - 成功

MemoryAddressing_ proc

; 确保变量i有效
        cmp ecx,0
        jl InvalidIndex                 ;如果i < 0则跳转
        cmp ecx,NumFibVals_
        jge InvalidIndex                ;如果i >= NumFibVals_则跳转

; 对变量i进行符号扩展，以进行地址计算
```

```
        movsxd rcx,ecx                  ;对 i 进行符号扩展
        mov [rsp+8],rcx                 ;保存 i

; 示例 1 — 基址寄存器
        mov r11,offset FibVals          ;r11 = FibVals
        shl rcx,2                       ;rcx = i * 4
        add r11,rcx                     ;r11 = FibVals + i * 4
        mov eax,[r11]                   ;eax = FibVals[i]
        mov [rdx],eax                   ;保存到 v1

; 示例 2 — 基址寄存器 + 索引寄存器
        mov r11,offset FibVals          ;r11 = FibVals
        mov rcx,[rsp+8]                 ;rcx = i
        shl rcx,2                       ;rcx = i * 4
        mov eax,[r11+rcx]               ;eax = FibVals[i]
        mov [r8],eax                    ;保存到 v2

; 示例 3 — 基址寄存器 + 索引寄存器 * 放大因子
        mov r11,offset FibVals          ;r11 = FibVals
        mov rcx,[rsp+8]                 ;rcx = i
        mov eax,[r11+rcx*4]             ;eax = FibVals[i]
        mov [r9],eax                    ;保存到 v3

; 示例 4 — 基址寄存器 + 索引寄存器 * 放大因子 + 偏移
        mov r11,offset FibVals-42       ;r11 = FibVals - 42
        mov rcx,[rsp+8]                 ;rcx = i
        mov eax,[r11+rcx*4+42]          ;eax = FibVals[i]
        mov r10,[rsp+40]                ;r10 = 指向 v4
        mov [r10],eax                   ;保存到 v4

; 示例 5 — RIP 相对寻址
        add [FibValsSum ],eax           ;更新和

        mov eax,1                       ;设置成功返回码
        ret

InvalidIndex:
        xor eax,eax                     ;设置错误返回码
        ret

MemoryAddressing_ endp
        end
```

在 C++ 源代码文件 MemoryAddressing.cpp（见清单 18-3）中，函数 _tmain 有一个简单的循环逻辑，执行汇编语言函数 MemoryAddressing_。它以变量 i 为索引从一个 32 位整型数组中取值。在 MemoryAddressing_ 被调用的过程中，整型变量 v1、v2、v3 和 v4 以各种不同的寻址模式访问数组。这些值再通过 printf 显示出来，以供对比。

汇编源文件 MemoryAddressing_.asm（见清单 18-4）在一开始定义了一个数组 FibVals，该数组包含了一组 32 位整数常量，可通过不同的内存寻址模式进行访问。文件接下来定义了一个 .data 数据段，里面定义了一个 32 位整型数 FibValsSum_，程序将会借助它来实现 RIP 相对寻址。函数 MemoryAddressing_ 先对参数 i（通过 ECX 寄存器传递）进行有效性判断。指令 movsxd rcx, ecx 将 32 位的变量 i 符号扩展到 64 位，扩展是有必要的，因为 i 将被用于计算 FibVals 成员的 64 位地址。接着，RCX 的值被保存在栈的备份空间中，以供后用。

MemoryAddressing_ 函数剩下部分的指令就是以各种不同的寻址模式来访问指定的 FibVals 成员变量。这里用到的各种寻址模式，本质上和 32 位版本的 MemoryAddressing_ 函

数都是一样的，只是使用的是 64 位的地址寄存器。请注意，在指令 mov r11, offset FibVals
[513] 中使用的是 64 位立即数，除此之外，x86-64 所有的其他指令都只能使用 32 位立即数，这一点我们在第 17 章中已经讨论过了。指令 cmp ecx, [NumFibvals_] 和 add [FibValsSum_], eax 使用的是 RIP 相对寻址模式。输出 18-2 列出了示例程序 MemoryAddressing 的运行结果。

输出 18-2 示例程序 MemoryAddressing

```
i: -1  rc:  0 - v1:    1 v2:    -1 v3:    -1 v4:    -1
i:  0  rc:  1 - v1:    0 v2:     0 v3:     0 v4:     0
i:  1  rc:  1 - v1:    1 v2:     1 v3:     1 v4:     1
i:  2  rc:  1 - v1:    1 v2:     1 v3:     1 v4:     1
i:  3  rc:  1 - v1:    2 v2:     2 v3:     2 v4:     2
i:  4  rc:  1 - v1:    3 v2:     3 v3:     3 v4:     3
i:  5  rc:  1 - v1:    5 v2:     5 v3:     5 v4:     5
i:  6  rc:  1 - v1:    8 v2:     8 v3:     8 v4:     8
i:  7  rc:  1 - v1:   13 v2:    13 v3:    13 v4:    13
i:  8  rc:  1 - v1:   21 v2:    21 v3:    21 v4:    21
i:  9  rc:  1 - v1:   34 v2:    34 v3:    34 v4:    34
i: 10  rc:  1 - v1:   55 v2:    55 v3:    55 v4:    55
i: 11  rc:  1 - v1:   89 v2:    89 v3:    89 v4:    89
i: 12  rc:  1 - v1:  144 v2:   144 v3:   144 v4:   144
i: 13  rc:  1 - v1:  233 v2:   233 v3:   233 v4:   233
i: 14  rc:  1 - v1:  377 v2:   377 v3:   377 v4:   377
i: 15  rc:  1 - v1:  610 v2:   610 v3:   610 v4:   610
i: 16  rc:  1 - v1:  987 v2:   987 v3:   987 v4:   987
i: 17  rc:  1 - v1: 1597 v2:  1597 v3:  1597 v4:  1597
i: 18  rc:  0 - v1:   -1 v2:    -1 v3:    -1 v4:    -1
FibValsSum_: 4180
```

18.1.3 整型操作数

大多数的 x86-64 指令可使用变长的操作数，长度范围在 8 位到 64 位之间。示例程序 IntegerOperands 演示了如何对不同长度的整型数进行位运算。清单 18-5 和清单 18-6 分别包含了本示例程序的 C++ 和汇编语言源代码。

清单 18-5 IntegerOperands.cpp

```
#include "stdafx.h"
#include "MiscDefs.h"

// 下面的结构体定义必须和 IntegerOperands_.asm 中声明的一致
typedef struct
{
    Uint8 a8;
    Uint16 a16;
    Uint32 a32;
    Uint64 a64;
    Uint8 b8;
    Uint16 b16;
    Uint32 b32;
    Uint64 b64;
} ClVal;

extern "C" void CalcLogical_(ClVal* cl_val, Uint8 c8[3], Uint16 c16[3],
Uint32 c32[3], Uint64 c64[3]);

int _tmain(int argc, _TCHAR* argv[])
```

[514]

```
{
    ClVal x;
    Uint8 c8[3];
    Uint16 c16[3];
    Uint32 c32[3];
    Uint64 c64[3];

    x.a8 = 0x81;                    x.b8 = 0x88;
    x.a16 = 0xF0F0;                 x.b16 = 0x0FF0;
    x.a32 = 0x87654321;             x.b32 = 0xF000F000;
    x.a64 = 0x0000FFFF00000000;     x.b64 = 0x0000FFFF00008888;

    CalcLogical_(&x, c8, c16, c32, c64);

    printf("\nResults for CalcLogical()\n");

    printf("\n8-bit operations\n");
    printf("0x%02X & 0x%02X = 0x%02X\n", x.a8, x.b8, c8[0]);
    printf("0x%02X | 0x%02X = 0x%02X\n", x.a8, x.b8, c8[1]);
    printf("0x%02X ^ 0x%02X = 0x%02X\n", x.a8, x.b8, c8[2]);

    printf("\n16-bit operations\n");
    printf("0x%04X & 0x%04X = 0x%04X\n", x.a16, x.b16, c16[0]);
    printf("0x%04X | 0x%04X = 0x%04X\n", x.a16, x.b16, c16[1]);
    printf("0x%04X ^ 0x%04X = 0x%04X\n", x.a16, x.b16, c16[2]);

    printf("\n32-bit operations\n");
    printf("0x%08X & 0x%08X = 0x%08X\n", x.a32, x.b32, c32[0]);
    printf("0x%08X | 0x%08X = 0x%08X\n", x.a32, x.b32, c32[1]);
    printf("0x%08X ^ 0x%08X = 0x%08X\n", x.a32, x.b32, c32[2]);

    printf("\n64-bit operations\n");
    printf("0x%016llX & 0x%016llX = 0x%016llX\n", x.a64, x.b64, c64[0]);
    printf("0x%016llX | 0x%016llX = 0x%016llX\n", x.a64, x.b64, c64[1]);
    printf("0x%016llX ^ 0x%016llX = 0x%016llX\n", x.a64, x.b64, c64[2]);

    return 0;
}
```

515

清单 18-6 IntegerOperands_.asm

```
; 下面声明的结构体定义必须和 IntegerOperands.cpp 中声明的一致
; 请注意，为了符合 C++ 编译器关于结构体成员变量的对齐要求，下面定义中含有一些填充字节

ClVal   struct
a8      byte ?
pad1    byte ?
a16     word ?
a32     dword ?
a64     qword ?
b8      byte ?
pad2    byte ?
b16     word ?
b32     dword ?
b64     qword ?
ClVal   ends

        .code

; extern "C" void CalcLogical_(ClVal* cl_val, Uint8 c8[3], Uint16 c16[3],↵
Uint32 c32[3], Uint64 c64[3]);
;
```

```
; 描述：此函数演示了对八位长度的整数进行各种逻辑操作
CalcLogical_ proc

; 8位逻辑操作
        mov r10b,[rcx+ClVal.a8]          ;r10b = a8
        mov r11b,[rcx+ClVal.b8]          ;r11b = b8
        mov al,r10b
        and al,r11b                      ;计算 a8 & b8
        mov [rdx],al
        mov al,r10b
        or al,r11b                       ;计算 a8 | b8
        mov [rdx+1],al
        mov al,r10b
        xor al,r11b                      ;计算 a8 ^ b8
        mov [rdx+2],al

; 16位逻辑操作
        mov rdx,r8                       ;rdx = 指向 c16
        mov r10w,[rcx+ClVal.a16]         ;r10w = a16
        mov r11w,[rcx+ClVal.b16]         ;r11w = b16
        mov ax,r10w
        and ax,r11w                      ;计算 a16 & b16
        mov [rdx],ax
        mov ax,r10w
        or ax,r11w                       ;计算 a16 | b16
        mov [rdx+2],ax
        mov ax,r10w
        xor ax,r11w                      ;计算 a16 ^ b16
        mov [rdx+4],ax

; 32位逻辑操作
        mov rdx,r9                       ;rdx = 指向 c32
        mov r10d,[rcx+ClVal.a32]         ;r10d = a32
        mov r11d,[rcx+ClVal.b32]         ;r11d = b32
        mov eax,r10d
        and eax,r11d                     ;计算 a32 & b32
        mov [rcx],eax
        mov eax,r10d
        or eax,r11d                      ;计算 a32 | b32
        mov [rdx+4],eax
        mov eax,r10d
        xor eax,r11d                     ;计算 a32 ^ b32
        mov [rdx+8],eax

; 64位逻辑操作
        mov rdx,[rsp+40]                 ;rdx = 指向 c64
        mov r10,[rcx+ClVal.a64]          ;r10 = a64
        mov r11,[rcx+ClVal.b64]          ;r11 = b64
        mov rax,r10
        and rax,r11                      ;计算 a64 & b64
        mov [rdx],rax
        mov rax,r10
        or rax,r11                       ;计算 a64 | b64
        mov [rdx+8],rax
        mov rax,r10
        xor rax,r11                      ;计算 a64 ^ b64
        mov [rdx+16],rax

        ret
CalcLogical_ endp
        end
```

在 C++ 源文件 IntegerOperands.cpp（见清单 18-5）的开头定义了一个结构体 CIVal，它包含了各种标准长度的整型成员变量。我们将通过这个结构体将源操作数传递给汇编语言函数 CalcLogical_。在 _tmain 函数中创建一个 CIVal 结构体的实例，将它作为参数调用 CalcLogical_，并显示运算结果。

在汇编源文件 IntegerOperands .asm（见清单 18-6）中也声明了一个汇编版本的 CIVal 结构体定义。在声明结构体时，要注意的一点是，大部分 C++ 编译器默认都会对成员变量进行多字节对齐，并因此会进行字节填充。在本示例程序中，Visual C++ 编译器会自动地对成员变量 a8 和 b8 进行字节填充以实现对齐[⊖]。但汇编编译器是不会自动对齐的，所以必须手动添加填充字节。这是为什么在 CIVal 的汇编实现中会出现 pad1 和 pad2。

CalcLogical_ 函数有四段代码块，使用不同长度的整型操作数进行位逻辑操作。每段代码的运算结果被保存到对应的结果数组中。该函数也演示了如何通过添加适当的后缀来引用 R8 ~ R15 这些 64 位寄存器的低 8 位、低 16 位以及低 32 位。输出 18-3 列出了示例程序 IntegerOperands 的执行结果。

输出 18-3　示例程序 IntegerOperands

```
Results for CalcLogical()

8-bit operations
0x81 & 0x88 = 0x80
0x81 | 0x88 = 0x89
0x81 ^ 0x88 = 0x09

16-bit operations
0xF0F0 & 0x0FF0 = 0x00F0
0xF0F0 | 0x0FF0 = 0xFFF0
0xF0F0 ^ 0x0FF0 = 0xFF00

32-bit operations
0x87654321 & 0xF000F000 = 0x80004000
0x87654321 | 0xF000F000 = 0xF765F321
0x87654321 ^ 0xF000F000 = 0x77651321

64-bit operations
0x0000FFFF00000000 & 0x0000FFFF00008888 = 0x0000FFFF00000000
0x0000FFFF00000000 | 0x0000FFFF00008888 = 0x0000FFFF00008888
0x0000FFFF00000000 ^ 0x0000FFFF00008888 = 0x0000000000008888
```

518

18.1.4　浮点数运算

在第 17 章，我们已知道所有 x86-64 兼容的处理器都支持 SSE2，并因此能使用它的标量浮点数资源。也就是说，我们可以用 XMM 寄存器代替 x87 FPU 进行浮点运算。更进一步，由于 SSE2 的存在，我们可以用 XMM 寄存器传递浮点参数，或从函数返回浮点数。本节的示例程序 FloatingPointArithmetic 演示了如何在 x86-64 汇编语言函数中实现标量浮点数运算。清单 18-7 和清单 18-8 分别包含了本示例程序的 C++ 和汇编语言源代码。

⊖ 在这里，变量 a16 和 b16 被定义为双字节整数，所以其应该对齐到 2 字节边界的。但 a8 和 b8 定义为单字节整数，需要增加一个字节的填充才能实现 2 字节对齐。否则，位于它们后面的变量 a16 和 b16 就不对齐了。——译者注

清单 18-7 FloatingPointArithmetic.cpp

```
#include "stdafx.h"

extern "C" double CalcSum_(float a, double b, float c, double d, float e,↩
double f);

extern "C" double CalcDist_(int x1, double x2, long long y1, double y2,↩
float z1, short z2);

void CalcSum(void)
{
    float a = 10.0f;
    double b = 20.0;
    float c = 0.5f;
    double d = 0.0625;
    float e = 15.0f;
    double f = 0.125;

    double sum = CalcSum_(a, b, c, d, e, f);

    printf("\nResults for CalcSum()\n");
    printf("a: %10.4f  b: %10.4lf c: %10.4f\n", a, b, c);
    printf("d: %10.4lf  e: %10.4f f: %10.4lf\n", d, e, f);
    printf("\nsum: %10.4lf\n", sum);
}

void CalcDist(void)
{
    int x1 = 5;
    double x2 = 12.875;
    long long y1 = 17;
    double y2 = 23.1875;
    float z1 = -2.0625;
    short z2 = -6;

519     double dist = CalcDist_(x1, x2, y1, y2, z1, z2);

    printf("\nResults for CalcDist()\n");
    printf("x1: %10d  x2: %10.4lf\n", x1, x2);
    printf("y1: %10lld  y2: %10.4lf\n", y1, y2);
    printf("z1: %10.4f  z2: %10d\n", z1, z2);
    printf("\ndist: %12.6lf\n", dist);
}

int _tmain(int argc, _TCHAR* argv[])
{
    CalcSum();
    CalcDist();
    return 0;
}
```

清单 18-8 FloatingPointArithmetic_.asm

```
        .code

; extern "C" double CalcSum_(float a, double b, float c, double d, float e,↩
double f);
;
; 描述：下面的函数演示了如何在 x86-64 汇编中传递浮点参数
;
```

```
CalcSum_ proc

; Sum the argument values
        cvtss2sd xmm0,xmm0              ;将 a 提升为 DPFP
        addsd xmm0,xmm1                 ;xmm0 = a + b

        cvtss2sd xmm2,xmm2              ;将 c 提升为 DPFP
        addsd xmm0,xmm2                 ;xmm0 = a + b + c
        addsd xmm0,xmm3                 ;xmm0 = a + b + c + d

        cvtss2sd xmm4,real4 ptr [rsp+40]    ;将 e 提升为 DPFP
        addsd xmm0,xmm4                 ;xmm0 = a + b + c + d + e

        addsd xmm0,real8 ptr [rsp+48]       ;xmm0 = a + b + c + d + e + f

        ret
CalcSum_ endp

; extern "C" double CalcDist_(int x1, double x2, long long y1, double y2,
float z1, short z2);
;
; 注意：下面的函数演示了如何在 x86-64 函数中混合传递浮点参数和整型参数
;
CalcDist_ proc

; 计算 xd = (x2 - x1) * (x2 - x1)
        cvtsi2sd xmm4,ecx               ;将 x1 转换为 DPFP
        subsd xmm1,xmm4                 ;xmm1 = x2 - x1
        mulsd xmm1,xmm1                 ;xmm1 = xd

; 计算 yd = (y2 - y1) * (y2 - y1)
        cvtsi2sd xmm5,r8                ;将 y1 转换为 DPFP
        subsd xmm3,xmm5                 ;xmm3 = y2 - y1
        mulsd xmm3,xmm3                 ;xmm3 = yd

; 计算 zd = (z2 - z1) * (z2 - z1)
        movss xmm0,real4 ptr [rsp+40]       ;xmm0 = z1
        cvtss2sd xmm0,xmm0              ;将 z1 转换为 DPFP
        movsx eax,word ptr [rsp+48]     ;eax = 符号扩展 z2
        cvtsi2sd xmm4,eax               ;将 z2 转换为 DPFP
        subsd xmm4,xmm0                 ;xmm4 = z2 - z1
        mulsd xmm4,xmm4                 ;xmm4 = zd

; 计算 final distance sqrt(xd + yd + zd)
        addsd xmm1,xmm3                 ;xmm1 = xd + yd
        addsd xmm4,xmm1                 ;xmm4 = xd + yd + zd
        sqrtsd xmm0,xmm4                ;xmm0 = sqrt(xd + yd + zd)

        ret
CalcDist_ endp
        end
```

C++ 源文件 FloatingPointArithmetic.cpp（见清单 18-7）中包含的两个函数准备了若干测试项，用来测试汇编语言函数 CalcSum_ 和 CalcDist_。请注意，CalcSum_ 只有浮点参数，而 CalcDist_ 则既有浮点参数，又有整型参数。这两个测试函数的目的是向大家演示 Visual C++ 调用约定是如何处理不同类型的数字参数的。

根据 Visual C++ 调用约定，前四个浮点参数通过寄存器 XMM0 ~ XMM3 传递，其他的浮点参数通过栈传递。如果一个函数既有浮点参数又有整型参数，则前四个参数（数

数或浮点数）通过通用寄存器或 XMM 寄存器传递，其他参数通过栈传递。此调用约定将 XMM0 ~ XMM5 视为易变的，将 XMM6 ~ XMM15 视为非易变的。如果函数的返回值是浮点数，必须通过寄存器 XMM0 返回。

清单 18-8 中包含了函数 CalcSum_ 的 x86-64 汇编实现源代码，它将 6 个浮点参数进行相加，并返回总和。图 18-4 展示了函数 CalcSum_ 被调用时的栈布局和寄存器内容。寄存器
521 XMM0 ~ XMM3 分别用来传递参数 a、b、c 和 d（XMM 寄存器的 64 ~ 127 位内容是未定义的，故未在图中显示）。参数 e 和 f 通过栈传递。请注意，因为函数 CalcSum_ 没有整型参数，所以通用寄存器 RCX、RDX、R8 和 R9 的内容是未定义的。CalcSum_ 的代码逻辑非常直白，若干个 addsd 指令把参数相加。cvtss2sd 指令用来将单精度浮点数 a、c 和 e 提升为双精度浮点数。最后，XMM0 寄存器包含了 6 个参数的和，函数可直接返回。

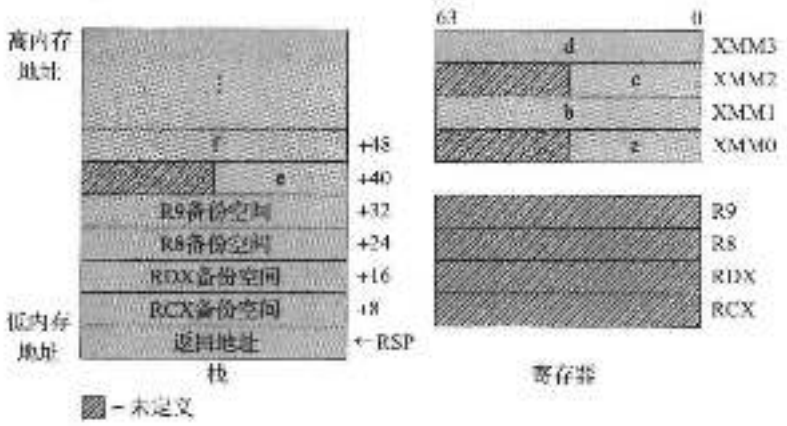


图 18-4 进入 CalcSum_ 函数时的栈布局和寄存器值

对于一个既有浮点参数又有整型参数的函数，调用者必须根据前四个参数的类型混合使用通用寄存器或 XMM 寄存器。如果被调函数的第一个参数类型是整型（或者指针类型），则它通过通用寄存器 RCX 传递。如果被调用函数的第一个参数是浮点型，则需要使用寄存器 XMM0 传递。类似地，根据参数类型（整型或浮点型），第二个参数通过寄存器 RDX 或 XMM1 传递，第三个参数通过寄存器 R8 或 XMM2 传递，第四个参数通过寄存器 R9 或 XMM3 传递。其他参数都通过栈传递。

函数 CalcDist_ 计算三维空间中两个点的距离，它的前四个参数是整型数和浮点数混合在一起的，所以在调用时，根据参数的具体类型，通过通用寄存器或者 XMM 寄存器传递。其他参数则通过栈传递，如图 18-5 所示。CalcDist_ 函数执行的运算十分简单。注意，函数用到了指令 cvtsi2sd 和 cvtss2sd，目的是把整数和单精度浮点数转换为双精度浮点数。输出 18-4 列出了示例程序
522 FloatingPointArithmetic 的执行结果。

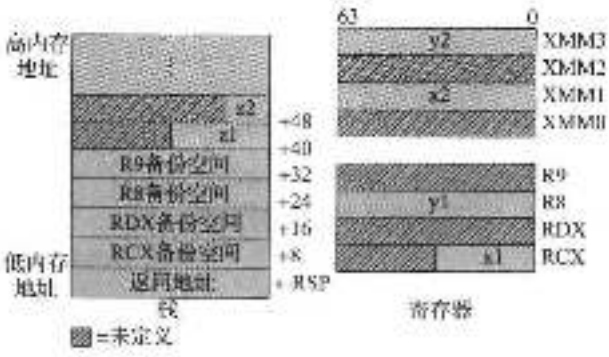


图 18-5 进入 CalcDist_ 函数时的栈布局和寄存器值

输出 18-4 示例程序 FloatingPointArithmetic

```
Results for CalcSum()
a:    10.0000  b:    20.0000 c:     0.5000
d:     0.0625  e:    15.0000 f:     0.1250

sum:    45.6875

Results for CalcDist()
x1:          5  x2:    12.8750
y1:         17  y2:    23.1875
z1:    -2.0625  z2:         -6

dist:    10.751259
```

18.2 x86-64 调用约定

本节学习 x86-64 非叶函数编程。Visual C++ 调用约定对非叶函数的序言和结语[⊖]提出了很严格的编程要求。调用约定还新增了一些指示符供汇编器生成静态数据，Visual C++ 运行时环境利用这些数据来处理异常。使用非叶函数的好处很多，包括可以使用个部通用寄存器和 XMM 寄存器，可使用栈帧指针，可通过栈创建局部变量，以及调用其他函数。 523

本节的前三个示例程序演示了如何通过显式的指令和指示符来编写 x86-64 非叶函数，同时呈现了组织非叶函数时的一些关键编程信息。在第四个例子中，对序言和结语的宏做了说明，在编写非叶函数时，这些宏能自动完成大部分编码工作。

18.2.1 基本栈帧

本节的第一个例子是 CallingConvention1，演示了如何在汇编语言函数中初始化栈帧指针，并通过它来引用栈上的参数和局部变量。同时演示了编写 x86-64 汇编语言函数序言和结语时必须注意的一些编程约定。清单 18-9 和清单 18-10 分别列出了示例程序 CallingConvention1 的 C++ 和汇编实现代码。

清单 18-9 CallingConvention1.cpp

```
#include "stdafx.h"
#include "MiscDefs.h"

extern "C" Int64 Cc1_(Int8 a, Int16 b, Int32 c, Int64 d, Int8 e, Int16 f,↩
Int32 g, Int64 h);

int _tmain(int argc, _TCHAR* argv[])
{
    Int8 a = 10, e = -20;
    Int16 b = -200, f = 400;
    Int32 c = 300, g = -600;
    Int64 d = 4000, h = -8000;

    Int64 x = Cc1_(a, b, c, d, e, f, g, h);

    printf("\nResults for CallingConvention1\n");
    printf("  a, b, c, d:  %8d %8d %8d %8lld\n", a, b, c, d);
    printf("  e, f, g, h:  %8d %8d %8d %8lld\n", e, f, g, h);
```

⊖ 在函数被调用时，需要有一段程序来保存当前的栈帧信息，以及一些寄存器，当函数调用结束后，就要恢复现场。保存现场的代码为序言，而恢复现场的代码则为结语。——译者注

```
    printf("  x:          %lld\n", x);
    return 0;
}
```

524

清单 16-10 CallingConvention1_.asm

```
        .code

; extern "C" Int64 Cc1_(Int8 a, Int16 b, Int32 c, Int64 d, Int8 e, Int16 f,↩
Int32 g, Int64 h);
;
; 描述：下面的函数演示了如何创建和使用基本的 x86-64 栈帧指针
;

Cc1_ proc frame

; 函数序言
        push rbp                        ;保存调用者的 rbp 寄存器
        .pushreg rbp

        sub rsp,16                      ;分配局部栈空间
        .allocstack 16

        mov rbp,rsp                     ;设置帧指针
        .setframe rbp,0

RBP_RA = 24                             ;从 rbp 到 ret 地址的偏移
        .endprolog                      ;序言结束标记

; 把变量寄存器中的值保存到备份空间（可选）
        mov [rbp+RBP_RA+8],rcx
        mov [rbp+RBP_RA+16],rdx
        mov [rbp+RBP_RA+24],r8
        mov [rbp+RBP_RA+32],r9

; 对变量 a、b、c 和 d 求和
        movsx rcx,cl                    ;rcx = a
        movsx rdx,dx                    ;rdx = b
        movsxd r8,r8d                   ;r8 = c;
        add rcx,rdx                     ;rcx = a + b
        add r8,r9                       ;r8 = c + d
        add r8,rcx                      ;r8 = a + b + c + d
        mov [rbp],r8                    ;save a + b + c + d

; 对变量 e、f、g 和 h 求和
        movsx rcx,byte ptr [rbp+RBP_RA+40]  ;rcx = e
        movsx rdx,word ptr [rbp+RBP_RA+48]  ;rdx = f
        movsxd r8,dword ptr [rbp+RBP_RA+56] ;r8 = g
        add rcx,rdx                     ;rcx = e + f
        add r8,qword ptr [rbp+RBP_RA+64]    ;r8 = g + h
525     add r8,rcx                      ;r8 = e + f + g + h

; 计算总和
        mov rax,[rbp]                   ;rax = a + b + c + d
        add rax,r8                      ;rax = 总和

; 函数结语
        add rsp,16                      ;释放局部栈空间
        pop rbp                         ;恢复调用者的 rbp 寄存器
        ret
Cc1_    endp
        end
```

C++ 源文件 CallingConvention1.cpp（见清单 18-9）中实现的功能主要是为了给函数 Ce1_ 提供一个测试用例，Ce1 计算 8 个整型参数的和，并将其返回。计算的结果以一系列 printf 输出到屏幕上。

函数 Ce1_ 在汇编源文件 CallingConvention1_.asm（见清单 18-10）中，.code 指示符之后即是声明 Ce1 proc frame。声明 proc 标识了函数序言的开始。而属性 frame 用来告知汇编器 Ce1 函数使用栈帧指针，并告诉汇编器产生静态表单，供 Visual C++ 运行时环境用来处理异常。push rbp 指令把调用者的 RBP 寄存器值保存到栈上，因为 Ce1_ 将用此寄存器作为自己的栈帧指针。接着是一个声明 .pushreg rbp，它是一个汇编编译器指示符，要求汇编编译器在异常处理表中保存指令 push rbp 的偏移信息。请务必注意，汇编编译器指示符并非可执行的指令，它是用来告诉汇编编译器在源码汇编过程中应该完成某个特定的动作。

指令 sub rsp, 16 在栈上为局部变量申请 16 字节的空间，Ce1 函数仅使用了其中的 8 个字节，但 x86-64 调用约定要求非叶函数的栈指针在函数序言以外，必须保持 16 字节对齐。我们在本节的后面会学习更多关于栈指针对齐的要求。下面的声明 .allocstack 16 是一个汇编编译器指示符，要求汇编编译器把局部栈的长度保存在运行时异常处理表中。

指令 mov rbp, rsp 把寄存器 RBP 初始化为栈帧指针，指示符 .setframe rbp 0 把这个操作告知汇编编译器，偏移值 0 表示寄存器 RSP 和 RBP 之间的差。函数 Ce1_ 中，寄存器 RSP 和 RBP 值相同，所以二者之差为 0。在本节的后面，我们会学习更多关于 .setframe 指示符的知识。需要注意的是，x86-64 汇编语言函数可以使用任何非易变寄存器作为栈帧指针。但使用 RBP 作为栈帧指针，能保持 x86-32 和 x86-64 函数之间的一致性。最后的汇编编译器指示符 .endprolog 指示了 Ce1_ 函数序言的结束。图 18-6 展示了序言结束时的栈布局和参数寄存器的内容。

526

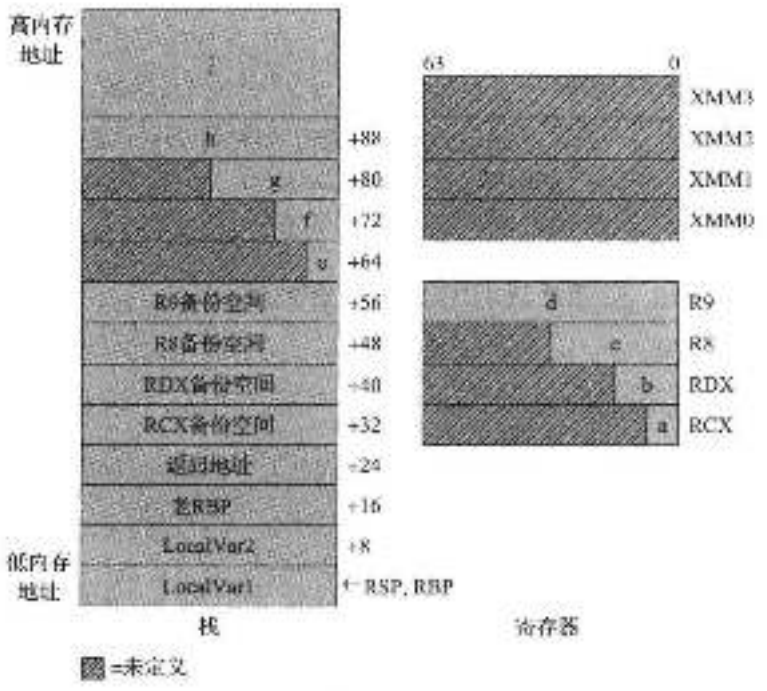


图 18-6 Ce1_ 函数序言结束时的栈布局和参数寄存器内容

后面的一段指令把寄存器 RCX、RDX、R8 和 R9 保存到栈上各自的备份空间中，此操作是可选的，Cc1_ 实现此操作的主要目的是为了给读者做演示。请注意，这段代码的 mov 指令中含有符号 RBP_RA，它的值为 24，表示为了能正确引用备份空间的地址而需要额外增加的偏移值（相比于叶函数）。Visual C++ 64 位调用约定允许的另一个可选操作是在执行 push rbp 指令前，利用 RSP 作为基址寄存器，把参数寄存器的值保存到备份空间（例如：mov [rsp+8], rcx 或 mov [rcx+16], rdx 等）。也请务必留心的是，函数可利用备份空间存储其他的临时值。当备份空间被用作此目的时，必须是指示符 .endprolog 之后的汇编代码才可以这样操作。

接下来是参数寄存器间的加法操作，函数 Cc1_ 要对变量 a、b、c 和 d 做求和运算。然后把求得的总和保存到局部变量 LocalVar1 中，执行指令 mov [rbp], r8。注意，在求和过程中，函数使用了 movsx 和 movsxd 指令将参数 a、b、c 和 d 进行符号扩展。用类似的逻辑对 e、f、g 和 h 进行求和运算，只是这几个参数都是通过栈传递的，并通过栈帧寄存器 RBP 加一个常量偏移来引用。符号 RBP_RA 在此两次被用到，作为引用参数时的额外偏移值。最后，
[527] 两次求和运算得到的总和相加，得到最终结果并保存在寄存器 RAX 中。

在 x86-64 函数的结语部分，必须释放所有在序言部分申请的栈空间，从栈上恢复所有的非易变寄存器的值，并执行函数返回。指令 add rsp, 16 释放在函数序言部分申请的 16 字节的栈空间。随后的指令 pop rbp 恢复 RBP 寄存器的值。最后还必须有一个 ret 指令。输出 18-5 列出了示例程序 CallingConvention1 的执行结果。

输出 18-5 示例程序 CallingConvention1

```
Results for CallingConvention1
  a, b, c, d:        10      -200       300      4000
  e, f, g, h:       -20       400      -600     -8000
  x:              -4110
```

18.2.2 使用非易变寄存器

下一个示例程序名为 CallingConvention2，演示了如何在 x86-64 函数中使用非易变寄存器。它同时也提供了一些关于栈帧和局部变量的编程细节。示例程序 CallingConvention2 的 C++ 和汇编语言源代码实现分别包含在清单 18-11 和清单 18-12 中。

清单 18-11 CallingConvention2.cpp

```
#include "stdafx.h"
#include "MiscDefs.h"

extern "C" bool Cc2_(const Int64* a, const Int64* b, Int32 n, Int64 * sum_a,=
Int64* sum_b, Int64* prod_a, Int64* prod_b);

int _tmain(int argc, _TCHAR* argv[])
{
    const __int32 n = 6;
    Int64 a[n] = { 2, -2, -6, 7, 12, 5 };
    Int64 b[n] = { 3, 5, -7, 8, 4, 9 };
    Int64 sum_a, sum_b;
    Int64 prod_a, prod_b;

    printf("\nResults for CallingConvention2\n");
    bool rc = Cc2_(a, b, n, &sum_a, &sum_b, &prod_a, &prod_b);
```

```
    if (!rc)
        printf("Invalid return code from Cc2_()\n");
    else
    {
        for (int i = 0; i < n; i++)
            printf("%7lld %7lld\n", a[i], b[i]);
        printf("\n");
        printf("sum_a:  %7lld sum_b:  %7lld\n", sum_a, sum_b);
        printf("prod_a: %7lld prod_b: %7lld\n", prod_a, prod_b);
    }

    return 0;
}
```

528

清单 18-12 CallingConvention2_.asm

```
        .code

; extern "C" void Cc2_(const Int64* a, const Int64* b, Int32 n, Int64*↩
sum_a, Int64* sum_b, Int64* prod_a, Int64* prod_b);
;
; 描述：下面的函数演示了如何初始化并使用栈帧指针，也示范了非易变通用寄存器的用法
;
; 命名常量表达式

; NUM_PUSHREG   = 函数序言压栈的非易变寄存器的数量
; STK_LOCAL1    = STK_LOCAL1 区域的长度
; STK_LOCAL2    = STK_LOCAL2 区域的长度
; STK_PAD       = 为保持 RSP 寄存器 16 字节对齐所需额外字节（0 或 8）
; STK_TOTAL     = 局部栈的总长度
; RBP_RA        = RBP 和返回地址之间的长度

NUM_PUSHREG     = 4
STK_LOCAL1      = 32
STK_LOCAL2      = 16
STK_PAD         = ((NUM_PUSHREG AND 1) XOR 1) * 8
STK_TOTAL       = STK_LOCAL1 + STK_LOCAL2 + STK_PAD
RBP_RA          = NUM_PUSHREG * 8 + STK_LOCAL1 + STK_PAD

Cc2     proc frame

; 把非易变寄存器的值保存在栈上
        push rbp
        .pushreg rbp
        push rbx
        .pushreg rbx
        push r12
        .pushreg r12
        push r13                                                    529
        .pushreg r13

; 申请局部栈空间，并设置栈指针
        sub rsp,STK_TOTAL                   ;分配局部栈空间
        .allocstack STK_TOTAL

        lea rbp,[rsp+STK_LOCAL2]            ;设置帧指针
        .setframe rbp,STK_LOCAL2

        .endprolog                          ;序言结束标记

; 初始化栈上的局部变量（仅是为了演示）
```

```
        pxor xmm5,xmm5
        movdqa [rbp-16],xmm6            ;将 xmm6 保存到 LocalVar2A/2B
        mov qword ptr [rbp],0aah        ;将 0xaa 保存到 LocalVar1A
        mov qword ptr [rbp+8],0bbh      ;将 0xbb 保存到 LocalVar1B
        mov qword ptr [rbp+16],0cch     ;将 0xcc 保存到 LocalVar1C
        mov qword ptr [rbp+24],0ddh     ;将 0xdd 保存到 LocalVar1D

; 4 个参数值保存到备份区域(可选):
        mov qword ptr [rbp+RBP_RA+8],rcx
        mov qword ptr [rbp+RBP_RA+16],rdx
        mov qword ptr [rbp+RBP_RA+24],r8
        mov qword ptr [rbp+RBP_RA+32],r9

; 为处理循环进行必要的初始化
        test r8d,r8d                    ;n <= 0?
        jle Error                       ;若 n <= 0, 则出错

        xor rbx,rbx                     ;rbx = 当前元素偏移
        xor r10,r10                     ;r10 = sum_a
        xor r11,r11                     ;r11 = sum_b
        mov r12,1                       ;r12 = prod_a
        mov r13,1                       ;r13 = prod_b

; 计算数组的和与乘积
@@:     mov rax,[rcx+rbx]               ;rax = a[i]
        add r10,rax                     ;更新 sum_a
        imul r12,rax                    ;更新 prod_a
        mov rax,[rdx+rbx]               ;rax = b[i]
        add r11,rax                     ;更新 sum_b
        imul r13,rax                    ;更新 prod_b

        add rbx,8                       ;调整 ebx 指向下一个成员
        dec r8d                         ;递减
        jnz @@                          ;循环结束

; 保存总和
        mov [r9],r10                    ;保存 sum_a
        mov rax,[rbp+RBP_RA+40]         ;rax = 指向 sum_b
        mov [rax],r11                   ;保存 sum_b
        mov rax,[rbp+RBP_RA+48]         ;rax = 指向 prod_a
        mov [rax],r12                   ;保存 prod_a
        mov rax,[rbp+RBP_RA+56]         ;rax = 指向 prod_b
        mov [rax],r13                   ;保存 prod_b
        mov eax,1                       ;set return code to true

; 函数结语
Done:   lea rsp,[rbp+STK_LOCAL1+STK_PAD]  ;恢复 rsp
        pop r13                         ;恢复 MV 寄存器
        pop r12
        pop rbx
        pop rbp
        ret

Error:  xor eax,eax                     ;设返回值为假
        jmp Done
Cc2_    endp
        end
```

530

源文件 CallingConvention2.cpp（见清单 18-11）中 C++ 代码的主要目的是为汇编语言函数 Cc2_ 准备一些简单的测试项。在这个简单的示例程序里，Cc2_ 读取数组和操作，并产生两个带符号的 64 位模型数组。最后，把执行结果通过 printf 函数显示在屏幕上。

在汇编源文件 CallingConvention2_.asm（见清单 18-12）的开始有若干命名常量，它们用来控制 Cc2_ 应在函数序言部分中请多大栈空间。和前一个例子一样，本示例也在 proc 声明中使用了 frame 属性，以告知汇编编译器本函数将会使用栈帧指针。随后的若干个 push 指令将多个非易变寄存器（RBP、RBX、R12 和 R13）的值保存到栈上。注意到每个 push 指令后面都跟着一个 .pushreg 指示符，这是指示汇编编译器把每次 push 操作都添加到 Visual C++ 运行时环境的异常处理表中。

指令 sub rsp, STK_TOTAL 为局部变量预留栈空间，其后是一条指示符 .allocstack STK_TOTAL。接下来 RBP 被初始化为栈帧指针，通过指令 lea rbp, [rsp+STK_LOCAL2]，RBP 的值等于 rsp+STK_LOCAL2。图 18-7 演示了 RBP 初始化之后的栈布局和变量寄存器的内容。通过设置 RBP，局部栈被划分成两个部分，从而使汇编编译器能够产生更有效的机器码，这是因为局部栈可以通过 8 位偏移而非 32 位偏移来引用。这种做法也令非易变类型 XMM 寄存器的保存和恢复变得简单，本节的后面会具体讨论此方面内容。在 lea 指令的后面是一条 .setframe rbp, STK_LOCAL2 指示符指示汇编编译器正确地配置异常处理表。这里要注意一点，.setframe 指示符中包含的长度值必须是一个小于等于 240 且为 16 的奇数倍的数值。后面的指示符 .endprolog 标识了 Cc2_ 函数的序言已结束。 531

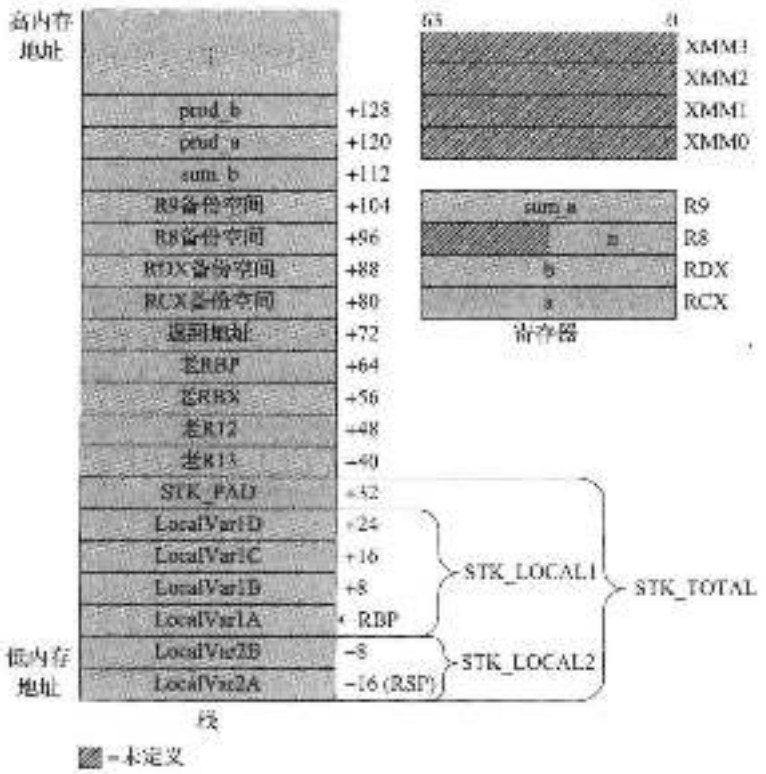


图 18-7 指令 lea rbp, [rsp+STK_LOCAL2] 执行后的栈布局和变量寄存器内容

接下来的代码块在栈上初始化局部变量，这仅是出于示例的目的。请注意，代码块中有一个指令是 movdqa [rpb-16], xmm5，它要求目标操作数的地址必须是 16 字节对齐的。这也是调用约定要求 RSP 寄存器必须符合 16 字节对齐的另一个原因。局部变量初始化之后，参数寄存器的值被保存到备份空间中。当然，这也纯粹是为了示例目的才这样做。

函数 Cc2_ 的主循环的逻辑比较简单，在验证了参数 n 的值之后，把两个中间值 sum_a（R10）和 sum_b（R11）初始化为 0，并把另两个中间值 prod_a（R12）和 prod_b（R13）初始化为 1。然后计算输入数组 a 和 b 的和，结果被保存在主调函数指定的内存中。注意，指
532 向 sum_b、prod_a 和 prod_b 的指针都在栈上。

此函数的结语部分，第一个指令是 lea rsp, [rbp+STK_LOCAL1+STK_PAD]，用来恢复 RSP 寄存器的值。Visual C++ 调用约定规定，必须通过指令 lea rsp, [rfp+X] 或 add rsp, X 来恢复 RSP 寄存器的值，其中的 rfp 代表栈帧指针，X 代表一个常量值。通过这样的限定，可大大地简化异常处理表的复杂度，只需要区分有限的几种指令模式。后面的 pop 指令用来恢复非易变类型通用寄存器的值。根据 Visual C++ 调用约定，函数结语不能含有任何处理逻辑，返回值的赋值也不能放在结语中。输出 18-6 列出了示例程序 CallingConvention2 的执行结果。

输出 18-6 示例程序 CallingConvention2

```
Results for CallingConvention2
       2       3
      -2       5
      -6      -7
       7       8
      12       4
       5       9

sum_a:       18 sum_b:       22
prod_a:   10080 prod_b:  -30240
```

18.2.3 使用非易变类型 XMM 寄存器

在本章开头，我们已经学习了如何借助 XMM 寄存器来实现标量浮点数的运算。本节中，我们将继续用示例程序 CallingConvention3 演示在使用非易变类型 XMM 寄存器时，函数序言和结语所必须注意的若干事项。清单 18-13 和清单 18-14 分别是 CallingConvention3 的 C++ 和汇编语言源代码。

清单 18-13 CallingConvention3.cpp

```
#include "stdafx.h"
#define _USE_MATH_DEFINES
#include <math.h>

extern "C" void Cc3_(const double* r, const double* h, int n, double*=
sa_cone, double* vol_cone);

int _tmain(int argc, _TCHAR* argv[])
{
    const int n = 6;
    double r[n] = { 1, 1, 2, 2, 3, 3 };
    double h[n] = { 2, 2, 3, 4, 5, 10 };
    double sa_cone1[n], sa_cone2[n];
533  double vol_cone1[n], vol_cone2[n];
    // 计算圆锥体侧面积和体积
    for (int i = 0; i < n; i++)
    {
        sa_cone1[i] = M_PI * r[i] * (r[i] + sqrt(r[i] * r[i] + h[i] * h[i]));
```

```
        vol_cone1[i] = M_PI * r[i] * r[i] * h[i] / 3.0;
    }

    Cc3 (r, h, n, sa_cone2, vol_cone2);

    printf("\nResults for CallingConvention3\n");
    for (int i = 0; i < n; i++)
    {
        printf("  r/h: %14.2lf %14.2lf\n", r[i], h[i]);
        printf("  sa:  %14.6lf %14.6lf\n", sa_cone1[i], sa_cone2[i]);
        printf("  vol: %14.6lf %14.6lf\n", vol_cone1[i], vol_cone2[i]);
        printf("\n");
    }

    return 0;
}
```

清单 18-14　CallingConvention4_.asm

```
            .const
I8_3p0      real8 3.0
I8_pi       real8 3.14159265358979323846
            .code

; extern "C" bool Cc3 (const double* r, const double* h, int n, double*↩
sa_cone, double* vol_cone);
;
; 描述：下面的函数演示了如何初始化并使用栈帧指针，同时举例说明了非易变类型通用寄存器和 XMM 寄存器的用法

; 定义常量的符号表达式
;
; NUM_PUSHREG   - 函数序言中非易变寄存器压栈个数
; STK_LOCAL1    - STK_LOCAL1 区域的字节数（参见正文插图）
; STK_LOCAL2    - STK_LOCAL2 区域的字节数（参见正文插图）
; STK_PAD       - 16 字节对齐 RSP 所需额外字节数
; STK_TOTAL     - 栈上局部变量的总字节数
; RBP_RA        - RBP 与栈上返回地址间的字节数

NUM_PUSHREG     = 7
STK_LOCAL1      = 16
STK_LOCAL2      = 64                                                    534
STK_PAD         = ((NUM_PUSHREG AND 1) XOR 1) * 8
STK_TOTAL       = STK_LOCAL1 + STK_LOCAL2 + STK_PAD
RBP_RA          = NUM_PUSHREG * 8 + STK_LOCAL1 + STK_PAD

Cc3_    proc frame

; 保存栈上的非易变寄存器
        push rbp
        .pushreg rbp
        push rbx
        .pushreg rbx
        push rsi
        .pushreg rsi
        push r12
        .pushreg r12
        push r13
        .pushreg r14
        push r14
        .pushreg r14
        push r15
```

```
        .pushreg r15

; 分配局部栈空间并初始化基础
        sub rsp,STK_TOTAL                   ;分配局部栈空间
        .allocstack STK_TOTAL
        lea rbp,[rsp+STK_LOCAL2]            ;rbp=栈帧指针
        .setframe rbp,STK_LOCAL2

; 保存非易变寄存器 XMM12 - XMM15。注意 STK_LOCAL2 必须不小于等于板保存
; XMM 寄存器个数的 16 倍
        movdqa xmmword ptr [rbp-STK_LOCAL2-48],xmm12
        .savexmm128 xmm12,48
        movdqa xmmword ptr [rbp-STK_LOCAL2-32],xmm13
        .savexmm128 xmm13,32
        movdqa xmmword ptr [rbp-STK_LOCAL2-16],xmm14
        .savexmm128 xmm14,16
        movdqa xmmword ptr [rbp-STK_LOCAL2],xmm15
        .savexmm128 xmm15,0
        .endprolog

; 访问栈上的局部变量（演示之用）
        mov qword ptr [rbp],-1              ;LocalVar1A = -1
535     mov qword ptr [rbp+8],-2            ;LocalVar1B = -2

; 初始化循环变量。注意，下面很多寄存器板初始化，仅是为了演示非易变类型
; 寄存器 GP 和 XMM 的使用
        movsxd rsi,r8d                      ;rsi = n
        test rsi,rsi                        ;n <= 0?
        jle Error                           ;jump if n <= 0

        xor rbx,rbx                         ;rbx = 数组元素偏移
        mov r12,rcx                         ;r12 = 指向 r
        mov r13,rdx                         ;r13 = 指向 h
        mov r14,r9                          ;r14 = 指向 sa_cone
        mov r15,[rbp+RBP_RA+40]             ;r15 = 指向 vol_cone
        movsd xmm14,[r8_pi]                 ;xmm14 = pi
        movsd xmm15,[r8_3p0]                ;xmm15 = 3.0

; 计算圆锥体表面积和体积
; sa = pi * r * (r + sqrt(r * r + h * h))
; vol = pi * r * r * h / 3
@@:     movsd xmm0,real8 ptr [r12+rbx]      ;xmm0 = r
        movsd xmm1,real8 ptr [r13+rbx]      ;xmm1 = h
        movsd xmm12,xmm0                    ;xmm12 = r
        movsd xmm13,xmm1                    ;xmm13 = h

        mulsd xmm0,xmm0         ;xmm0 = r * r
        mulsd xmm1,xmm1         ;xmm1 = h * h
        addsd xmm0,xmm1         ;xmm0 = r * r + h * h

        sqrtsd xmm0,xmm0        ;xmm0 = sqrt(r * r + h * h)
        addsd xmm0,xmm12        ;xmm0 = r + sqrt(r * r + h * h)
        mulsd xmm0,xmm12        ;xmm0 = r * (r + sqrt(r * r + h * h))
        mulsd xmm0,xmm14        ;xmm0 = pi * r * (r + sqrt(r * r + h * h))

        mulsd xmm12,xmm12       ;xmm12 = r * r
        mulsd xmm13,xmm14       ;xmm13 = h * pi
        mulsd xmm13,xmm12       ;xmm13 = pi * r * r * h
        divsd xmm13,xmm15       ;xmm13 = pi * r * r * h / 3

        movsd real8 ptr [r14+rbx],xmm0      ;保存面积
```

```
        movsd real8 ptr [r15+rbx],xmm13     ;保存面积

        add rbx,8                           ;设置 rbx 指向下一个元素
        dec rsi                             ;更新计数器
        jnz @@                              ;重复，直到完成
        mov eax,1                           ;设置成功返回码                    536

; 恢复非易变类型 XMM 寄存器
Done:   movdqa xmm12,xmmword ptr [rbp-STK_LOCAL2+48]
        movdqa xmm13,xmmword ptr [rbp-STK_LOCAL2+32]
        movdqa xmm14,xmmword ptr [rbp-STK_LOCAL2+16]
        movdqa xmm15,xmmword ptr [rbp-STK_LOCAL2]

; 函数结语
        lea rsp,[rbp+STK_LOCAL1+STK_PAD]    ;恢复 rsp
        pop r15                             ;恢复非易变通用寄存器
        pop r14
        pop r13
        pop r12
        pop rsi
        pop rbx
        pop rbp
        ret

Error:  xor eax,eax                         ;设置错误返回码
        jmp Done
Cc3_    endp
        end
```

tmain 函数（见清单 18-13）通过调用 x86-64 汇编语言函数 Cc3 计算圆锥体的表面积和体积。下面是计算圆锥体表面积和体积的公式：

$$sa = \pi r\left(r + \sqrt{r^2 + h^2}\right) \quad vol = \pi r^2 h/3$$

在 Cc3_ 函数（见清单 18-14）的开头，非易变类型寄存器的值被保存到栈上，然后申请了一段栈空间，并将 RBP 寄存器初始化为当前函数的栈帧指针。接着又通过若干个 movdqa 指令，把非易变类型的寄存器 XMM12 – XMM15 的值保存到栈上。每个 movdqa 指令的后面都跟着一个 .savexmm128 指示符，和序言中的其他指示符类似，这个指示符用来通知汇编编译器在异常处理表中添加相关的数据，以记录 XMM 寄存器被保存到了栈上的事实。.savexmm128 指示符中的偏移值用来指示被保存在栈中的 XMM 寄存器的值所在的栈地址相对于 RSP 寄存器的偏移。请注意，STK_LOCAL2 的值必须大于等于被保存到栈中的 XMM 寄存器的数量乘以 16，这是为了确保有足够的栈空间保存 XMM 寄存器的值。图 18-8 展示了在执行完指令 movdqa xmmword ptr [rbp-STK_LOCAL2], xmm15 后的栈布局和寄存器值。 537

函数序言之后的局部变量 LocalVar1A 和 LocalVar1B 只是用作演示目的。随后对函数主体循环部分用到的寄存器进行初始化。请注意，从实际效果看，这些初始化很多是没必要的，但我们每次都进行显式的初始化，以展示非易变类型通用寄存器和 XMM 寄存器的使用方法。接下来的代码实现了表面积和体积的计算，并使用了 SSE2 双精度浮点数算法。

主体循环部分结束后，调用了一系列的 movdqa 指令恢复 XMM 寄存器的值。函数 Cc3_ 接着释放占用的栈空间，并把先前保存的非易变类型通用寄存器的值恢复回去。输出 18-7 展示了示例程序 CallingConvention3 的执行结果。

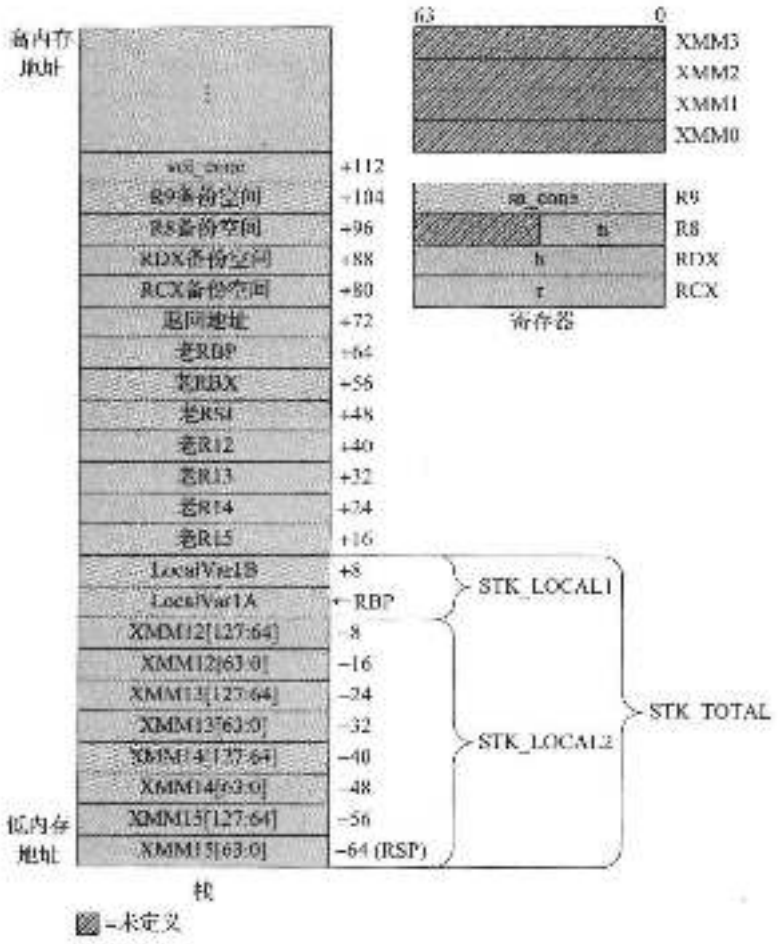


图 18-8 Cc3_ 函数在执行完指令 movdqa xmmword ptr [rbp-STK_LOCAL2], xmm15 之后的栈布局和寄存器值

输出 18-7 示例程序 CallingConvention3

```
Results for CallingConvention3
  r/h:          1.00          1.00
  sa:       7.584476      7.584476
  vol:      1.047198      1.047198

  r/h:          1.00          2.00
  sa:      10.166407     10.166407
  vol:      2.094395      2.094395

  r/h:          2.00          3.00
  sa:      35.220717     35.220717
  vol:     12.566371     12.566371

  r/h:          2.00          4.00
  sa:      40.665630     40.665630
  vol:     16.755161     16.755161

  r/h:          4.00          5.00
  sa:      83.229761     83.229761
  vol:     47.123890     47.123890
```

```
r/h:          3.00          10.00
sa:       126.671905    126.671905
vol:       94.247780     94.247780
```

18.2.4 简化序言和结语的宏

前面三个示例程序的目的是演示如何遵循 Visual C++ 调用约定编写 64 位非叶函数。此调用约定对函数序言和结语有着很严格的规定，某种程度上导致了编程的烦琐和啰嗦，也可能潜藏着难以发现的错误。64 位非叶函数的栈布局，主要取决于需要用到的非易变类型（包括通用的和 XMM 的）寄存器的数量，以及其他局部变量需要用到的空间，正确地认识这一点十分重要。我们最好是能够想到某种方法，自动完成和调用约定有关的烦琐代码。

538 | 539

本节的示例程序演示作者在编写 64 位非叶函数时使用的宏定义方法，该方法简化了栈帧的创建并为非易变寄存器保留足够空间。清单 18-5 和清单 18-6 分别列出了示例程序 CallingConvention4 的 C++ 和汇编语言源代码。

清单 18-15 CallingConvention4.cpp

```
#include "stdafx.h"
#include <math.h>

extern "C" bool Cc4_(const double* ht, const double* wt, int n, double*↩
bsa1, double* bsa2, double* bsa3);

int _tmain(int argc, _TCHAR* argv[])
{
    const int n = 6;
    const double ht[n] = { 150, 160, 170, 180, 190, 200 };
    const double wt[n] = { 50.0, 60.0, 70.0, 80.0, 90.0, 100.0 };
    double bsa1_a[n], bsa1_b[n];
    double bsa2_a[n], bsa2_b[n];
    double bsa3_a[n], bsa3_b[n];

    for (int i = 0; i < n; i++)
    {
        bsa1_a[i] = 0.007184 * pow(ht[i], 0.725) * pow(wt[i], 0.425);
        bsa2_a[i] = 0.0235 * pow(ht[i], 0.42246) * pow(wt[i], 0.51456);
        bsa3_a[i] = sqrt(ht[i] * wt[i]) / 60.0;
    }

    Cc4_(ht, wt, n, bsa1_b, bsa2_b, bsa3_b);

    printf("Results for CallingConvention4\n\n");

    for (int i = 0; i < n; i++)
    {
        printf("Height: %6.1lf cm\n", ht[i]);
        printf("Weight: %6.1lf kg\n", wt[i]);
        printf("BSA (C++): %10.6lf %10.6lf %10.6lf (sq. m)\n", bsa1_a[i],↩
        bsa2_a[i], bsa3_a[i]);
        printf("BSA (X86-64): %10.6lf %10.6lf %10.6lf (sq. m)\n", bsa1_b[i],↩
        bsa2_b[i], bsa3_b[i]);
        printf("\n");
    }
    return 0;
}
```

清单 18-16 CallingConvention4.asm

```
        include <MacrosX86-64.inc>

; 为 BSA 函数准备的浮点常量
                .const
r8_0p007184     real8 0.007184
r8_0p725        real8 0.725
r8_0p425        real8 0.425
r8_0p0235       real8 0.0235
r8_0p42246      real8 0.42246
r8_0p51456      real8 0.51456
r8_60p0         real8 60.0

        .code
        extern pow:proc

; extern "C" bool Cc4_(const double* ht, const double* wt, int n, double*
bsa1, double* bsa2, double* bsa3);
;
; 描述：下面系统数据小宏的使用：_CreateFrame、_DeleteFrame、_EndProlog、
;       _SaveXmmRegs、_RestoreXmmRegs

Cc4_    proc frame
        _CreateFrame Cc4_,16,64,rbx,rsi,r12,r13,r14,r15
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9
        _EndProlog

; 保存变量寄存器
        mov qword ptr [rbp+Cc4_OffsetHomeRCX],rcx
        mov qword ptr [rbp+Cc4_OffsetHomeRDX],rdx
        mov qword ptr [rbp+Cc4_OffsetHomeR8],r8
        mov qword ptr [rbp+Cc4_OffsetHomeR9],r9

; 初始化循环指针。注意，指针保存在非易变寄存器中，以避免调用 pow() 函数后
; 重新加载
        test r8d,r8d                            ;n <= 0?
        jle Error                               ;若 n <= 0，则跳转
        mov [rbp],r8d                           ;保存 n 到局部变量

        mov r12,rcx                             ;r12 = 指向 ht
        mov r13,rdx                             ;r13 = 指向 wt
        mov r14,r9                              ;r14 = 指向 bsa1
        mov r15,[rbp+Cc4_OffsetStackArgs]       ;r15 = 指向 bsa2
        mov rbx,[rbp+Cc4_OffsetStackArgs+8]     ;rbx = 指向 bsa3
        xor rsi,rsi                             ;数组元素偏移

; 在栈上为 pow() 函数分配空间
        sub rsp,32

; 计算 bsa1 = 0.007184 * pow(ht, 0.725) * pow(wt, 0.425);
next:   movsd xmm0,real8 ptr [r12+rsi]          ;xmm0 = 身高
        movsd xmm8,xmm0
        movsd xmm1,real8 ptr [r8_0p725]
        call pow                                ;xmm0 = pow(ht,0.725)
        movsd xmm6,xmm0

        movsd xmm0,real8 ptr [r13+rsi]          ;xmm0 = 体重
        movsd xmm9,xmm0
        movsd xmm1,real8 ptr [r8_0p425]
```

```
        call pow                                ;xmm0 = pow(wt,0.425)
        mulsd xmm6,real8 ptr [r8_0p007184]
        mulsd xmm6,xmm0                         ;xmm6 = bsa1

; 计算bsa2 = 0.0235 * pow(ht, 0.42246) * pow(wt, 0.51456);
        movsd xmm0,xmm8                         ;xmm0 = 身高
        movsd xmm1,real8 ptr [r8_0p42246]
        call pow                                ;xmm0 = pow(ht,0.42246)
        movsd xmm7,xmm0

        movsd xmm0,xmm9                         ;xmm0 = 体重
        movsd xmm1,real8 ptr [r8_0p51456]
        call pow                                ;xmm0 = pow(wt,0.51456)
        mulsd xmm7,real8 ptr [r8_0p0235]
        mulsd xmm7,xmm0                         ;xmm7 = bsa2

; 计算bsa3 = sqrt(ht * wt) / 60.0;
        mulsd xmm8,xmm9
        sqrtsd xmm8,xmm8
        divsd xmm8,real8 ptr [r8_60p0]          ;xmm8 = bsa3

; 保存BSA结果
        movsd real8 ptr [r14+rsi],xmm6          ;保存bsa1结果
        movsd real8 ptr [r15+rsi],xmm7          ;保存bsa2结果
        movsd real8 ptr [rbx+rsi],xmm8          ;保存bsa3结果

        add rsi,8                               ;更新数值偏移
        dec dword ptr [rbp]                     ;n = n - 1
        jnz @@
        mov eax,1                               ;设置成功返回码

; 恢复前面保存的寄存器。注意，_DeleteFrame 宏从 rbp 中恢复 rsp 的值，忽略着
; 不必要地通过指令 add rsp,32 来恢复 pow() 函数的栈空间                          542
Done:   _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9
        _DeleteFrame rbx,rsi,r12,r13,r14,r15
        ret

Error:  xor eax,eax                             ;设置错误返回码
        jmp Done
Cc4_    endp
        end
```

本示例程序和前一个示例程序的设计逻辑类似。tmain（见清单 18-15）的主要目的是执行 64 位汇编语言函数 Cc4_。这个函数用来估算人的体表面积（Body Surface Area，BSA），它用到了几个广泛使用的计算 BSA 的方程式，定义于表 18-2 中。表中的三个方程式都使用符号 H 表示以厘米（cm）为单位的身高，符号 W 表示以千克（kg）为单位的体重，并用 BSA 表示身体表面积，单位是平方米（m^2）。

表 18-2 体表面积（BSA）计算方程式

F	方程式
DuBois 和 DuBois	$BSA = 0.007184 \times H^{0.725} \times W^{0.425}$
Gehan 和 George	$BSA = 0.0235 \times H^{0.42246} \times W^{0.51456}$
Mosteller	$BSA = \sqrt{H \times W} / 3600$

在汇编源文件 CallingConvention4_asm（见清单 18-16）的最开始部分，是一个 include 声明，将文件 Macrosx86-64.inc（此头文件中的代码未列出）包含进来。此文件位于子目录 CommonFiles 中，包含了若干将用于程序 CallingConvention4 以及后续示例程序中的宏定义。在 include 声明的后面定义了一个 .const 段，其中定义了若干 BSA 计算方程式中将会用到的浮点数常量。

图 18-9 展示了一个通用的 64 位非叶函数的栈布局。请大家注意它和图 18-7 及图 18-8 之间的差异。后者包含了更多细节。文件 Macrosx86-64.inc 中定义的所有宏，假设所有函数的基本栈布局都符合图 18-9 的设计。但同时，这些宏允许函数自定义栈的大小，即具体要为非易变寄存器预留多少栈空间。宏在其内部实现了大多数计算逻辑，从而省去了函数自身
543 的工作量。

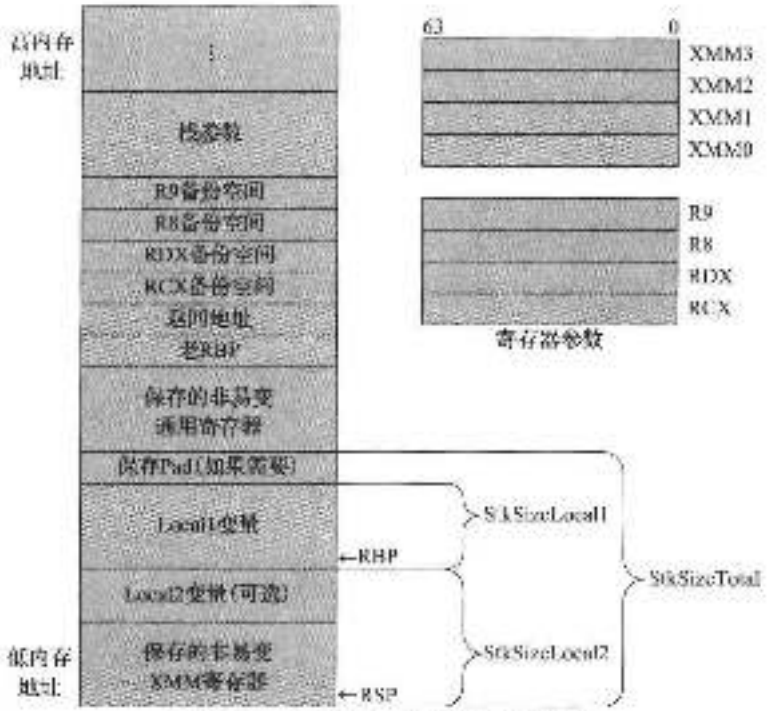


图 18-9 64 位非叶函数的通用栈布局

随后是 Cer4 proc frame 声明，接着就是宏 _CreateFrame，用以产生初始化函数栈帧的代码，同时也会把指定的非易变通用寄存器的值保存在栈上。这个宏在调用时是需要传入一些参数的，包括一个用于标识的前缀字符串以及以字节为单位的变量 StkSizeLocal1 和 StkSizeLocal2（见图 18-9）。前缀字符串用于符号名的创建，通过互相区别的符号名，可以引用栈中的变量。前缀字符串可以取任意的唯一值，一般的习惯是以函数名为其取值内容。变量 StkSizeLocal1 和 StkSizeLocal2 必须是 16 的倍数倍，同时，StkSizeLocal2 不能大于 240，但又必须大于或等于被保存的 XMM 寄存器数量的 16 倍。

接下来，宏 SaveXmmRegs 用以保存指定的非易变 XMM 寄存器的值到栈的特定区域。随后执行的是宏 _EndProlog，指示了函数序言的结束。序言结束后，寄存器 RBP 被配置为
544 本函数的栈帧指针。这时候，如果要继续在栈上保存非易变的通用寄存器或 XMM 寄存器，也是安全的。

宏 _EndProlog 之后的一大块指令，是保存变量寄存器的值到栈的特定备份空间中。大家可能已经注意到，每个 mov 指令都包含一个符号名，代表了寄存器在栈上相对于 RBP 的位置偏移。这些符号名和对应的偏移值，都是在宏 _CreateFrame 中定义好的。另外，前面章节中已经提到过，备份空间也可以用来保存临时数据。

接下来的代码初始化与循环处理相关的变量。寄存器 R8D 中保存的变量 n 用来进行合法性判断，作为局部变量保存在栈中。多个非易变寄存器被初始化为指针寄存器。非易变寄存器的使用，可避免每次在调用库函数 pow 后重新加载寄存器值。注意，指向数组 bsa2 的指针的值，是通过指令 mov r15, [rbp+Cc4_OffsetStackArgs] 从栈上获取的。符号常量 Cc4_OffsetStackArgs 也是经由宏 _CreateFrame 而自动创建的，其值等于第一个栈变量相对 RBP 的栈偏移。指令 mov rbx, [rbp+Cc4_OffsetStackArgs+8] 将变量 bsa3 的值加载到 RBX 寄存器，常量偏移 +8 的存在，是因为 bsa3 是栈上的第二个参数变量。

Visual C++ 调用约定需要主调函数为被调函数申请备份空间，指令 sub rsp 32 就是完成这个任务的。随后的指令块就是实现 BSA 计算的逻辑部分，使用的是表 18-2 中的方程式。请注意，每次调用函数 pow 之前，都会为寄存器 XMM0 和 XMM1 加载相应的值。同时，函数 pow 的返回值，通常也是先被保存在 XMM 寄存器中。

计算 BSA 的逻辑部分处理完后，就是函数结语。在 ret 指令被执行前，Cc4 函数需要先恢复所有非易变通用寄存器和 XMM 寄存器的值，同时释放栈帧。宏 _RestoreXmmRegs 用来恢复 XMM 寄存器。请注意，传递给这个宏的寄存器参数的数量和顺序，必须和函数序言部分的宏 _SaveXmmRegs 被调用时保持一致。宏 _DeleteFrame 用来恢复其他非易变的通用寄存器，并清除栈帧。同样，寄存器参数的传递顺序，也必须和宏 _CreateFrame 保持一致。注意，宏 _DeleteFrame 用 RBP 的值来恢复 RSP 寄存器的值，这样避免了通过显式地执行指令 add rsp, 32 来恢复栈。输出 18-8 显示了示例程序 CallingConvention4 的执行结果。

输出 18-8 Sample Program CallingConvention4

```
Results for CallingConvention4

height:  150.0 cm
weight:   50.0 kg
BSA (C++):       1.432500   1.460836   1.443376 (sq. m)
BSA (X86-64):    1.432500   1.460836   1.443376 (sq. m)

height:  160.0 cm
weight:   60.0 kg
BSA (C++):       1.622063   1.648868   1.632993 (sq. m)
BSA (X86-64):    1.622063   1.648868   1.632993 (sq. m)

height:  170.0 cm
weight:   70.0 kg
BSA (C++):       1.809708   1.831289   1.818119 (sq. m)
BSA (X86-64):    1.809708   1.831289   1.818119 (sq. m)

height:  180.0 cm
weight:   80.0 kg
BSA (C++):       1.996421   2.009483   2.000000 (sq. m)
BSA (X86-64):    1.996421   2.009483   2.000000 (sq. m)

height:  190.0 cm
weight:   90.0 kg
```

545

```
BSA (C++):      2.182809   2.184365   2.179449 (sq. m)
BSA (X86-64):   2.182809   2.184365   2.179449 (sq. m)

height:  200.0 cm
weight:  100.0 kg
BSA (C++):      2.369262   2.356574   2.357023 (sq. m)
BSA (X86-64):   2.369262   2.356574   2.357023 (sq. m)
```

18.3 x86-64 数组和字符串

本节的示例程序将演示如何使用 x86-64 指令集操纵常见的编程结构。第一个程序演示如何利用 64 位指针来处理二维数组中的元素。第二个程序演示若干字符串处理指令的使用。这两个程序都遵循和使用前面讲到的调用约定以及相关的宏。

18.3.1 二维数组

第 2 章中，我们已经学习了如何利用一块连续的内存和简单的指针操作，实现一个二维数组和或矩阵。本示例程序使用类似的指针操作方法，实现一个 x86-64 的矩阵乘法函数。
546 清单 18-7 和清单 18-8 分别包含了示例程序 MatrixMul 的 C++ 和汇编语言源代码。

清单 18-17 MatrixMul.cpp

```
#include "stdafx.h"
#include <stdlib.h>

extern "C" double* MatrixMul_(const double* m1, int nr1, int nc1, const↩
 double* m2, int nr2, int nc2);

void MatrixPrint(const double* m, int nr, int nc, const char* s)
{
    printf("%s\n", s);

    if (m != NULL)
    {
        for (int i = 0; i < nr; i++)
        {
            for (int j = 0; j < nc; j++)
            {
                double m_val = m[i * nc + j];
                printf("%8.1lf ", m_val);
            }
            printf("\n");
        }
    }
    else
        printf("NULL pointer\n");
}

double* MatrixMulCpp(const double* m1, int nr1, int nc1, const double* m2,↩
int nr2, int nc2)
{
    if ((nr1 < 0) || (nc1 < 0) || (nr2 < 0)  | (nc2 < 0))
        return NULL;
    if (nc1 != nr2)
        return NULL;

    double* m3 = (double*)malloc(nr1 * nc2 * sizeof(double));
```

```
    for (int i = 0; i < nr1; i++)
    {
        for (int j = 0; j < nc2; j++)
        {
            double sum = 0;                                          547
            for (int k = 0; k < nc1; k++)
            {
                double m1_val = m1[i * nc1 + k];
                double m2_val = m2[k * nc2 + j];
                sum += m1_val * m2_val;
            }
            m3[i * nc2 + j] = sum;
        }
    }

    return m4;
}

void MatrixMul1(void)
{
    const int nr1 = 3;
    const int nc1 = 2;
    const int nr2 = 2;
    const int nc2 = 3;
    double m1[nr1 * nc1] = { 6, 2, 4, 4, -5, -2 };
    double m2[nr2 * nc2] = { -2, 3, 4, -4, 6, 7 };
    double* m3_a = MatrixMulCpp(m1, nr1, nc1, m2, nr2, nc2);
    double* m3_b = MatrixMul_(m1, nr1, nc1, m2, nr2, nc2);

    printf("\nResults for MatrixMul1()\n");
    MatrixPrint(m1, nr1, nc1, "Matrix m1");
    MatrixPrint(m2, nr2, nc2, "Matrix m2");
    MatrixPrint(m3_a, nr1, nc2, "Matrix m3_a");
    MatrixPrint(m3_b, nr1, nc2, "Matrix m3_b");
    free(m3_a);
    free(m3_b);
}

void MatrixMul2(void)
{
    const int nr1 = 2;
    const int nc1 = 3;
    const int nr2 = 3;
    const int nc2 = 4;
    double m1[nr1 * nc1] = { 5, -3, 2, -2, 5, 4 };
    double m2[nr2 * nc2] = { 7, -4, 3, 3, 2, 6, -2, 5, 4, 9, 3, 5 };
    double* m3_a = MatrixMulCpp(m1, nr1, nc1, m2, nr2, nc2);
    double* m3_b = MatrixMul_(m1, nr1, nc1, m2, nr2, nc2);

    printf("\nResults for MatrixMul2()\n");
    MatrixPrint(m1, nr1, nc1, "Matrix m1");
    MatrixPrint(m2, nr2, nc2, "Matrix m2");                          548
    MatrixPrint(m3_a, nr1, nc2, "Matrix m3_a");
    MatrixPrint(m3_b, nr1, nc2, "Matrix m3_b");
    free(m3_a);
    free(m3_b);
}

int _tmain(int argc, _TCHAR* argv[])
{
    MatrixMul1();
```

```
    MatrixMul2();
    return 0;
}
```

清单 18-18 MatrixMul_.asm

```
        include <MacrosX86-64.inc>
        .code
        extern malloc:proc

; extern "C" double* MatrixMul_(const double* m1, int nr1, int nc1, const-
double* m2, int nr2, int nc2);
;
; 描述：下面的函数计算两个矩阵乘的积

MatrixMul_ proc frame
        CreateFrame MatMul_,0,0,rbx,r12,r13,r14,r15
        EndProlog

; 验证矩阵大小
        movsxd r12,edx                          ;r12 = nr1
        test r12,r12
        jle Error                               ;若 nr1 <= 0, 则跳转

        movsxd r13,r8d                          ;r13 = nc1
        test r13,r13
        jle Error                               ;若 nc1 <= 0, 则跳转

        movsxd r14,dword ptr [rbp+MatMul_OffsetStackArgs]     ;r14 = nr2
        test r14,r14
        jle Error                               ;若 nr2 <= 0, 则跳转

        movsxd r15,dword ptr [rbp+MatMul_OffsetStackArgs+8]   ;r15 = nc2
        test r15,r15
        jle Error                               ;若 nc2 <= 0, 则跳转

        cmp r13,r14
549     jne Error                               ;若 nc1 != nr2, 则跳转

; 分配空间
        mov [rbp+MatMul_OffsetHomeRCX],rcx  ;save m1
        mov [rbp+MatMul_OffsetHomeR9],r9    ;save m2
        mov rcx,r12                         ;rcx = nr1
        imul rcx,r15                        ;rcx = nr1 * nc2
        shl rcx,3                           ;rcx = nr1 * nc2 * size real8
        sub rsp,32                          ;分配空间
        call malloc
        mov rbx,rax                         ;rbx = 指向 m3

; 初始化源矩阵指针和行索引 i
        mov rcx,[rbp+MatMul_OffsetHomeRCX]  ;rcx = 指向 m1
        mov rdx,[rbp+MatMul_OffsetHomeR9]   ;rdx = 指向 m2
        xor r8,r8                           ;i = 0

; 初始化列索引变量 j
lp1:    xor r9,r9                           ;j = 0

; 初始化 sum 和索引 k
lp2:    xorpd xmm4,xmm4                     ;sum = 0;
        xor r10,r10                         ;k = 0;
```

```
;计算 sum += m1[i * nc1 + k] * m2[k * nc2 + j]
Lp3:    mov rax,r8                      ;rax = i
        imul rax,r13                    ;rax = i * nc1
        add rax,r10                     ;rax = i * nc1 + k
        movsd xmm0,real8 ptr [rcx+rax*8]    ;xmm0 = m1[i * nc1 + k]

        mov r11,r10                     ;r11 = k;
        imul r11,r15                    ;r11 = k * nc2
        add r11,r9                      ;r11 = k * nc2 + j
        movsd xmm1,real8 ptr [rdx+r11*8]    ;xmm1 = m2[k * nc2 + j]

        mulsd xmm0,xmm1         ;xmm0 = m1[i * nc1 + k] * m2[k * nc2 + j]
        addsd xmm4,xmm0         ;更新 sum

        inc r10                         ;k++
        cmp r10,r13
        jl Lp3                          ;若 k < nc1 则跳转

;保存 sum 到 m3[i * nc2 + j]
        mov rax,r8                      ;rax = i
        imul rax,r15                    ;rax = i * nc2
        add rax,r9                      ;rax = i * nc2 + j
        movsd real8 ptr [rbx+rax*8],xmm4    ;m3[i * nc2 + j] = sum

;更新循环计数并重复，直到结束
        inc r9                          ;j++
        cmp r9,r15
        jl Lp2                          ;若 j < nc2 则跳转
        inc r8                          ;i++
        cmp r8,r12
        jl Lp1                          ;若 i < nr1 则跳转

        mov rax,rbx                     ;rax = 指向 m3

Done:    DeleteFrame rbx,r12,r13,r14,r15
        ret

Error:  xor rax,rax                     ;返回 NULL
        jmp Done
MatrixMul_ endp
        end
```

下面给出矩阵乘法的定义。假设 A 是一个 m 行 n 列的矩阵，B 是 n 行 p 列的矩阵，则矩阵 $C=AB$（C 是一个 m 行 p 列的矩阵）的计算公式如下：

$$c_{ij}=\sum_{k=0}^{n-1}a_{ik}b_{kj} \qquad i=0,\cdots,m-1,\ j=0,\cdots,p-1$$

在 C++ 源文件 MatrixMul.cpp（见清单 18-17）中，包含了一个名为 MatrixMulCpp 的函数，它计算矩阵乘积。它首先对矩阵的大小进行必要的验证，然后申请一块内存，以容纳新生成的矩阵。此后根据上面得到的公式，对两个矩阵进行乘法运算。文件 MatrixMul.cpp 中剩余的代码，创建了一些矩阵乘积的测试用例，并打印结果以展示和比较。

清单 18-8 包含了函数 MatrixMul_ 的 64 位汇编语言源代码。开头部分是对函数的声明，紧接着宏 _CreateFrame 被执行，保存必要的非易变寄存器的值，并初始化栈帧指针。注意，传递给宏 _CreateFrame 的两个栈空间变量的值都是 0，这是因为函数 MatrixMul_ 并没有用到任何局部变量或 XMM 寄存器。定义矩阵大小的变量 nr1、nc1、nr2、nc2 被分别加载到寄存器 R12、R13、R14 和 R15 中。另外请注意，在作为函数参数传递时，nr1 和 nc1 通过

550

寄存器 EDX 和 R8D 传递，而 nr2 和 nc2 通过栈传递。

用于保存新生成的矩阵的内存空间，是通过调用标准库函数 malloc 来实现的。调用 malloc 前，指向源矩阵 m1 和 m2 的指针被保存在各自的栈备份空间中，后面它们将分别以寄存器 RCX 和 R9 作为调用参数传递给函数 MatrixMul_。指令 sub rsp 32 用来为 malloc 调
551 用申请必要的栈空间。

目标矩阵的内存申请好之后，源矩阵指针 m1 和 m2 的值被分别加载到寄存器 RCX 和 RDX 中。函数接着会通过一个三层循环结构计算矩阵乘积，逻辑和 C++ 代码相同。所有双精度浮点数运算，都是经由易变 XMM 寄存器来完成的。三层循环结束后，宏 DeleteFrame 被调用，以恢复非易变通用寄存器的值。输出 18-9 列出了示例程序 MatrixMul 的执行结果。

输出 18-9 示例程序 MatrixMul

```
Results for MatrixMul1()
Matrix m1
     6.0      2.0
     4.0      3.0
    -5.0     -2.0
Matrix m2
    -2.0      3.0      4.0
    -1.0      6.0      7.0
Matrix m3_a
   -18.0     30.0     38.0
   -17.0     30.0     37.0
    16.0    -27.0    -34.0
Matrix m3_b
   -18.0     30.0     38.0
   -17.0     30.0     37.0
    16.0    -27.0    -34.0

Results for MatrixMul2()
Matrix m1
     5.0     -3.0      2.0
    -2.0      5.0      4.0
Matrix m2
     7.0     -4.0      3.0      3.0
     2.0      6.0     -2.0      5.0
     4.0      9.0      5.0      5.0
Matrix m3_a
    37.0    -20.0     27.0     10.0
    12.0     74.0     -4.0     39.0
Matrix m3_b
    37.0    -20.0     27.0     10.0
552 12.0     74.0     -4.0     39.0
```

18.3.2 字符串

本章最后一个示例程序是 ConcatStrings，它是第 2 章的 x86-32 字符串拼接程序的 x86-64 版本。当时我们利用 scasw 和 movsw 指令把多个字符串拼接在一起。清单 18-19 和清单 18-20 分别包含了示例程序 ConcatStrings 的 C++ 和汇编语言源代码。

清单 18-19 ConcatStrings.cpp

```
#include "stdafx.h"

extern "C" int ConcatStrings_(wchar_t* des, int des_size, const wchar_t*←
```

```
const* src, int src_n);

int _tmain(int argc, _TCHAR* argv[])
{
    printf("\nResults for ConcatStrings\n");

    // 目标缓冲区足够大
    wchar_t* src1[] = { L"One ", L"Two ", L"Three ", L"Four" };
    int src1_n = sizeof(src1) / sizeof(wchar_t*);
    const int des1_size = 64;
    wchar_t des1[des1_size];

    int des1_len = ConcatStrings_(des1, des1_size, src1, src1_n);
    wchar_t* des1_temp = (*des1 != '\0') ? des1 : L"<empty>";
    wprintf(L"  des_len: %d (%d) des: %s \n", des1_len, wcslen(des1_temp),↵
    des1_temp);

    // 目标缓冲区太小
    wchar_t* src2[] = { L"Red ", L"Green ", L"Blue ", L"Yellow " };
    int src2_n = sizeof(src2) / sizeof(wchar_t*);
    const int des2_size = 16;
    wchar_t des2[des2_size];

    int des2_len = ConcatStrings_(des2, des2_size, src2, src2_n);
    wchar_t* des2_temp = (*des2 != '\0') ? des2 : L"<empty>";
    wprintf(L"  des_len: %d (%d) des: %s \n", des2_len, wcslen(des2_temp),↵
    des2_temp);

    // 空字符串测试
    wchar_t* src3[] = { L"Airplane ", L"Car ", L"", L"Truck ", L"Boat " };
    int src3_n = sizeof(src3) / sizeof(wchar_t*);
    const int des3_size = 128;
    wchar_t des3[des3_size];                                          553
    int des3_len = ConcatStrings_(des3, des3_size, src3, src3_n);
    wchar_t* des3_temp = (*des3 != '\0') ? des3 : L"<empty>";
    wprintf(L"  des_len: %d (%d) des: %s \n", des3_len, wcslen(des3_temp),↵
    des3_temp);

    return 0;
}
```

清单 18-20　ConcatStrings_.asm

```
        include <MacrosX86-64.inc>
        .code

; extern "C" int ConcatStrings_(wchar_t* des, int des_size, const wchar_t*↵
const* src, int src_n)
;
; 描述：本函数把两个字符串合并为一个目标字符串
;
; 返回：-1          无效的 des_size 或 src_n
;       n >= 0      连接后的字符串长度

ConcatStrings_ proc frame
        _CreateFrame ConcatStrings_,0,0,rbx,rsi,rdi
        _EndProlog

; 确保 des_size 和 src_n 大于 0
        movsxd rdx,edx                  ;rdx = des_size
        test rdx,rdx
```

```
        jle Error                       ;若 des_size <= 0 则跳转
        movsxd r9,r9d                   ;r9 = src_n
        test r9,r9
        jle Error                       ;若 src_n <= 0 则跳转

; 做必要初始化
        mov rbx,rcx                     ;rbx = des
        xor r10,r10                     ;des_index = 0
        xor r11,r11                     ;i = 0
        mov word ptr [rbx],r10w         ;*des = '\0';

; 重复循环，直到合并完成
lp1:    mov rdi,[r8+r11*8]              ;rdi = src[i]
        mov rsi,rdi                     ;rsi = src[i]

; 计算 s[i] 长度
        xor rax,rax
554     mov rcx,-1
        repne scasw                     ;发现 '\0'
        not rcx
        dec rcx                         ;rcx = len(src[i])

; 计算 des_index + src_len
        mov rax,r10                     ;rax= des_index
        add rax,rcx                     ;rax = des_index + len(src[i])

; des_index + src_len >= des_size?
        cmp rax,rdx
        jge Done

; 拷贝 src[i] 到 &des[des_index] (rst 总是包含 src[i])
        inc rcx                         ;rcx = len(src[i]) + 1
        lea rdi,[rbx+r10*2]             ;rdi = &des[des_index]
        rep movsw                       ;以字符串移动

; 更新 des_index
        mov r10,rax                     ;des_index += len(src[i])
                                         若 src_n <= 0 则跳转
; 更新 i 并重复循环，直到完成
        inc r11                         ;i += 1
        cmp r11,r9                      ;i >= src_n?
        jl lp1                          ;若 i < src_n 则跳转

; 返回合并后字符串的长度
Done:   mov eax,r10d                    ;eax = trunc(des_index)
        _DeleteFrame rbx,rsi,rdi
        ret

; 返回错误代码
Error:  mov eax,-1                      ;eax = 错误代码
        _DeleteFrame rbx,rsi,rdi
        ret

ConcatStrings_ endp
        end
```

细心的读者可能注意到，C++ 源文件 ConcatStrings.cpp（见清单 18-19）和第 2 章是一模一样的。主要功能是创建若干个测试用例来调用汇编语言函数 ConcatStrings 并打印其结果。清单 18-20 是函数 ConcatStrings 的 x86-64 汇编实现。因为这个函数本身没有使用任何

局部变量，所以直接使用宏 _CreateFrame 来简单保存非易变寄存器 RBX、RSI 和 RDI 的值。函数接下来验证长度变量 des_size（通过 RDX 传递）和 src_n（通过寄存器 R9 传递）。注意，在代码实现中，是先把这两个值符号扩展为 64 位然后再进行验证的。 555

指令 mov rbx, rcx 将寄存器 RBX 的值复制到变量 des 中，RCX 用来给指令 scasw 和 movsw 传递数量值。在主循环部分，先是通过指令 mov rdi, [r8+r11*8] 将 src[i] 的内容加载到寄存器 RDI。这里用到了一个值为 8 的乘数，是因为 src[i] 中所有的字符串的地址长度都是 8 字节。接下来，用 scasw 指令计算字符串的长度。函数此时会判断 des 是否有足够的空间容纳新的字符串内容。如果发现空间不足，循环就中止。

函数 ConcatStrings_ 有两个函数结语。虽然这种设计对于本函数并非必要，但有些函数却可以通过这种方式有效提高执行的效率。这种方式避免了 jmp 指令的使用。两个函数结语都使用宏 _DeleteFrame 来恢复非易变寄存器的值。输出 18-10 列出了示例程序 ConcatStrings 的执行结果。

输出 18-10 示例程序 ConcatStrings

```
Results for ConcatStrings
  des_len: 18 (18) des: One Two Three Four
  des_len: 15 (15) des: Red Green Blue
  des_len: 24 (24) des: Airplane Car Truck Boat
```

18.4 总结

本章集中介绍了使用汇编语言编写 x86-64 程序。我们学习了 x86 汇编语言编程的基础知识，包括整数运算、内存寻址和标量浮点数运算；另外还学习了关于调用约定的使用技巧和编程经验。接下来的两章将继续探索 x86-64 平台，解析 SIMD 部分。 556

x86-64 单指令多数据流架构

前两章中，我们集中介绍了 x86-64 平台的基础及其核心架构。本章我们将进一步探讨 x86-64 平台的单指令多数据流（SIMD）架构，包括 x86-SSE 和 x86-AVX 计算资源。19.1 节我们将学习 64 位 x86-SSE 的执行环境，包括其寄存器集合、所支持的数据类型以及指令集。19.2 节我们将以同样的顺序介绍 64 位 x86-AVX 的执行环境。

对这一章的学习，我们假定读者已对本书前几章中关于 x86-SSE 和 x86-AVX 的内容有了基本的了解。基于 x86-32 和 x86-64 平台上的 SIMD 架构有很高的相似度，本章将刻意从简。在接下来的讨论中，我们会在必要的时候将“-32”或“-64”加在 x86-SSE 和 x86-AVX 两个术语后面，以便区分 32 位和 64 位 SIMD 架构。

19.1 x86-SSE-64 执行环境

本节将讨论 x86-SSE-64 的执行环境，包括其寄存器集合和支持的数据类型。从应用程序的角度看，大多数 x86-SSE-64 和 x86-SSE-32 之间的差异都是微不足道的。两个环境使用相同的指令、操作数和组合数据类型。

正如我们在第 17 章中提到的，所有 x86-64 兼容的处理器都包含 SSE2 的计算资源。但不同的 x86-64 处理器对 SSE2 后续扩展（SSE3、SSSE3、SSE4.1 和 SSE4.2）的支持有所不同，主要取决于处理器所使用的微架构。应用程序应该使用 cpuid 指令来检测某个特定的 SSE2 扩展功能是否可用。

19.1.1 x86-SSE-64 寄存器组

x86-SSE-64 寄存器组包括 16 个 128 位寄存器，这些寄存器依次被命名为 XMM0 ~ XMM15，如图 19-1 所示。XMM 寄存器支持使用组合整型操作数的 SIMD 操作。同时，这些寄存器还可用来进行标量和组合浮点计算，使用单精度值和双精度值均可。

[557]

x86-64 汇编语言函数可以使用 x86-SSE 控制 - 状态寄存器 MXCSR 来选择 SIMD 的浮点配置选项。通过对 MXCSR 中的状态位的检查我们可以获取 SIMD 浮点运算的错误状态。在 x86-64 和 x86-32 的执行环境中，MXCSR 中的每个控制位和状态位的用途和操作方法都是一样的。关于这一点，在第 7 章，特别是在图 7-3 和表 7-2 中已有详细论述。第 8 章中包含了一个演示

[558]

MXCSR 寄存器的使用方法的示例程序。

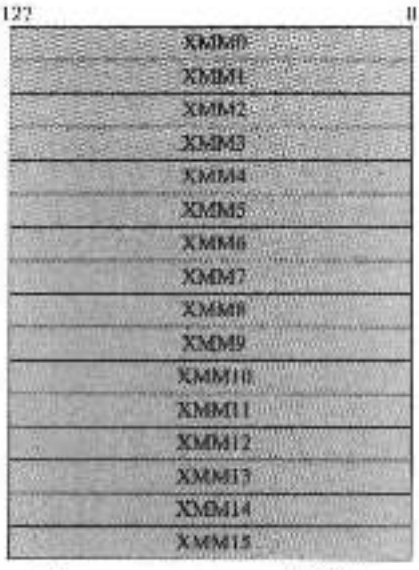


图 19-1 x86-SSE-64 寄存器组

19.1.2 x86-SSE-64 数据类型

x86-SSE-64 与 x86-SSE-32 支持相同的数据类型，包括组合整型数（8 位、16 位、32 位和

64 位）、标量浮点数（32 位单精度和 64 位双精度）和组合浮点数（32 位单精度和 64 位双精度）。图 7-2 显示了 x86-SSE 的数据类型。除了一小部分指令以外，内存中所有 128 位的组合整数和浮点操作数都必须进行 16 字节边界对齐。标量浮点操作数的对齐不是必需的，但出于性能的考虑我们仍强烈建议对齐。第 7 章中包含了关于 x86-SSE 数据类型的更多信息。

19.1.3 x86-SSE-64 指令集概述

除了一小部分指令需要通用寄存器作为操作数以外，x86-SSE-64 的指令集本质上与 x86-SSE-32 是一样的。所有使用通用寄存器操作数的 x86-SSE 指令都已被扩展成支持 64 位通用寄存器操作数。表 19-1 列出了可以使用 64 位通用寄存器操作数的 x86-SSE 指令。表中使用了 DPFP 和 SPFP 两个缩写来分别代表双精度浮点数和单精度浮点数。

表 19-1 x86-SSE 64 位通用寄存器指令

助记符	描 述
cvtsd2si	将标量 DPFP 转换成有符号整型数
cvtsi2sd	将有符号整型数转换成标量 DPFP
cvtsi2ss	将有符号整型数转换成标量 SPFP
cvtss2si	将标量 SPFP 转换成有符号整型数
cvttsd2si	将标量 DPFP 转换成有符号整型数，带数据截断
cvttss2si	将标量 SPFP 转换成有符号整型数，带数据截断
movmskpd	提取组合 DPFP 的符号位
movmskps	提取组合 SPFP 的符号位
movq	移动 64 位数据
pextrq	提取 64 位数据
pinsrq	插入 64 位数据
pmovmskb	移动 8 位掩码

559

x86-SSE 指令在 64 位和 32 位处理器模式下的执行过程都是一样的。当处于 64 位模式时，有一些 x86-SSE 组合字符串指令使用 64 位模式寄存器操作数，而不是 32 位。例如，pcmpestri 和 pcmppestrm 两个指令中需要使用的字符串片段长度就必须放入 RAX 和 RDX 中，而不是 EAX 和 EDX 中。同理，pcmpestri 和 pcmpistri 指令会将计算出来的字符索引放入 RCX 中而不是 ECX 中。

19.2 x86-AVX 执行环境

本节我们将讨论 x86-AVX-64 的执行环境，包括其寄存器集和支持的数据类型。和 x86-SSE 类似，绝大多数 x86-AVX-64 和 x86-AVX-32 执行环境之间的差异相对来讲都是很小的。但需要注意的是，并不是所有兼容 x86-64 的处理器都支持 x86-AVX。应用程序需要使用 cpuid 指令来检测其宿主处理器是否支持 AVX、AVX2 或任何 x86-AVX 衍生的扩展功能（如 FMA）。

19.2.1 x86-AVX-64 寄存器组

x86-AVX-64 寄存器组包含了 16 个 256 位寄存器，它们依次被命名为 YMM0 – YMM15。这些寄存器可以用来操作各种数据类型，包括组合整型数、组合浮点数和标量浮点数。每一个

YMM 寄存器的低 128 位都以对应的 XMM 寄存器来作为别名，如图 19-2 所示。大多数 x86-
560 AVX-64 指令都可以使用任何 XMM 或 YMM 寄存器作为操作数。

19.2.2 x86-AVX-64 数据类型

x86-AVX-64 支持的数据类型与 x86-AVX-32 相同，包括组合整型、标量浮点数和组合浮点数。第 12 章详细讨论了这些数据类型，它们也被列在了表 12-2 中。大多数 x86-AVX 指令都可以使用 XMM 或 YMM 来操作相应的 128 位和 256 位的组合操作数。我们在第 12 章中讨论过的宽松的内存对齐要求同样适用于 x86-AVX-64 的内存操作数。此处我们再强调一下，除了显式访问对齐的 128 位或 256 位内存操作数的数据传输指令以外，x86-AVX 操作数的对齐不是必需的。但我们强烈建议进行这样的内存对齐，以获得最佳
561 的性能。

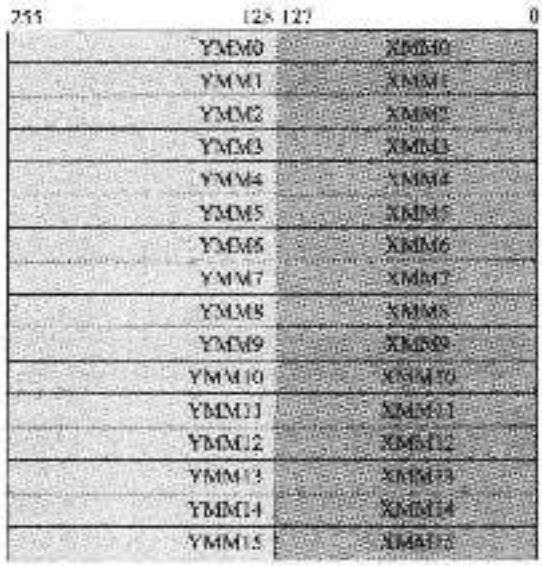


图 19-2 x86-AVX-64 寄存器组

19.2.3 x86-AVX-64 指令集概述

除了一些需要通用寄存器操作数的指令以外，x86-AVX-64 指令集与 x86-AVX-32 指令集基本上一样。表 19-1 中所列指令的 x86-AVX 形式都可以使用 64 位通用寄存器操作数。使用 VSIB 内存寻址的指令（如 vgatherdpd、vgatherdps 等）还可以将 64 位通用寄存器作为它们的基址寄存器操作数。

x86-AVX 指令的执行在 64 位和 32 位的状态下也没什么不同，包括使用 XMM 寄存器作为操作数时对 YMM 寄存器高 128 位的清零，以及会影响到 x86-AVX 标量浮点操作数中未使用的寄存器位的处理规则。x86-64 汇编语言函数应该使用 vzeroupper 或 vzeroall 指令来避免潜在的状态转换延迟。这种延迟可能会在 x86-AVX 和 x86-SSE 指令切换的时候发生。第 12 章中，我们详细讨论了上述问题，以及使用 x86-AVX 指令集编程的其他问题。

19.3 总结

在这一章里，你了解了 x86-SSE-64 和 x86-AVX-64 的架构，应该也体会到了 64 位 SIMD 架构和与其对应的 32 位架构之间的相似性。x86-SSE-64 和 x86-AVX-64 中的大量寄存器为我们提供了很多好处，包括简化汇编语言编码和算法性能的提升。下一章中，我们将
562 分析很多不同的示例程序来强化本章讲的内容。

x86-64 单指令多数据流编程

本章将介绍如何使用 x86-SSE 和 x86-AVX 的计算资源来编写 x86-64 的汇编语言函数。20.1 节用若干示例程序演示 x86-SSE 指令集的使用方法。20.2 节演示 x86-AVX 指令集的使用方法。本章示例代码中使用的 x86-SSE 和 x86-AVX 指令，大部分都已在前一章中讨论过。这样安排可以让我们的讨论更集中于算法和 64 位处理方法的层面，而不是指令执行的细枝末节上。

20.1 x86-SSE-64 编程

在第 9 章和第 10 章，我们已经学习了如何使用 x86-SSE 指令来编写处理组合浮点数和整型数的函数。在这一节中，我们将体验如何在 64 位汇编语言函数中使用 x86-SSE 资源。第一个示例程序是以 64 位方式实现的直方图绘图算法，该算法在第 10 章已经学习过。接下来的两个例子演示了如何使用 x86-SSE 指令集处理组合浮点数。本节的示例程序着重于介绍 x86-64 执行环境中新增计算资源的应用。

20.1.1 直方图绘制

在第 10 章，我们已经看到了如何利用 x86-SSE 指令集为 8 位灰度图像构建直方图。本小节我们将学习直方图绘制算法的 64 位实现。清单 20-1 和清单 20-2 分别列出了示例程序 Sse64ImageHistogram 的 C++ 和汇编代码。 [563]

清单 20-1 Sse64ImageHistogram.cpp

```
#include "stdafx.h"
#include "Sse64ImageHistogram.h"
#include <string.h>
#include <malloc.h>

extern "C" UInt32 NUM_PIXELS_MAX = 16777216;

bool Sse64ImageHistogramCpp(UInt32* histo, const UInt8* pixel_buff, UInt32 num_pixels)
{
    // 确保 num_pixels 的值是有效的
    if ((num_pixels == 0) || (num_pixels > NUM_PIXELS_MAX))
        return false;
    if (num_pixels % 32 != 0)
        return false;

    // 确保 histo 是 16 字节边界对齐的
    if (((uintptr_t)histo & 0xf) != 0)
        return false;

    // 确保 pixel_buff 是 16 字节边界对齐的
    if (((uintptr_t)pixel_buff & 0xf) != 0)
        return false;
```

```
    // 构建直方图
    memset(histo, 0, 256 * sizeof(Uint32));

    for (Uint32 i = 0; i < num_pixels; i++)
        histo[pixel_buff[i]]++;

    return true;
}

void Sse64ImageHistogram(void)
{
    const wchar_t* image_fn = L"..\\..\\..\\DataFiles\\TestImage1.bmp";
    const char* csv_fn = "__TestImage1_Histograms.csv";

    ImageBuffer ib(image_fn);
    Uint32 num_pixels = ib.GetNumPixels();
    Uint8* pixel_buff = (Uint8*)ib.GetPixelBuffer();
    Uint32* histo1 = (Uint32*)_aligned_malloc(256 * sizeof(Uint32), 16);
    Uint32* histo2 = (Uint32*)_aligned_malloc(256 * sizeof(Uint32), 16);
564 bool rc1, rc2;
    rc1 = Sse64ImageHistogramCpp(histo1, pixel_buff, num_pixels);
    rc2 = Sse64ImageHistogram_(histo2, pixel_buff, num_pixels);

    printf("Results for Sse64ImageHistogram()\n");

    if (!rc1 || !rc2)
    {
        printf("  Bad return code: rc1=%d, rc2=%d\n", rc1, rc2);
        return;
    }

    FILE* fp;
    bool compare_error = false;

    if (fopen_s(&fp, csv_fn, "wt") != 0)
        printf("  File open error: %s\n", csv_fn);
    else
    {
        for (Uint32 i = 0; i < 256; i++)
        {
            fprintf(fp, "%u, %u, %u\n", i, histo1[i], histo2[i]);

            if (histo1[i] != histo2[i])
            {
                printf("  Histogram compare error at index %u\n", i);
                printf("    counts: [%u, %u]\n", histo1[i], histo2[i]);
                compare_error = true;
            }
        }

        if (!compare_error)
            printf("  Histograms are identical\n");

        fclose(fp);
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    try
    {
```

```
        Sse64ImageHistogram();
        Sse64ImageHistogramTimed();
    }                                                                    565
    catch (...)
    {
        printf("Unexpected exception has occurred!\n");
        printf("File: %s (_train)\n", __FILE__);
    }

    return 0;
}
```

清单 20-2 Sse64ImageHistogram_.asm

```
        include <MacrosX86-64.inc>
        .code
        extern NUM_PIXELS_MAX:dword

; extern bool Sse64ImageHistogram (Uint32* histo, const UintR* pixel_buff,↩
Uint32 num_pixels);
;
; 描述：下面的函数用来构建一个直方图
;
; 返回值：0 = 非法参数值
;         1 = 成功
;
; 概要：x86-64, SSE4.1 支持

Sse64ImageHistogram_ proc frame
        _CreateFrame Sse64Ih_,1024,0,rbx,rsi,rdi
        _EndProlog

; 确保 num_pixels 是有效的
        test r8d,r8d
        jz Error                        ;如果 num_pixels 为 0 则跳转
        cmp r8d,[NUM_PIXELS_MAX]
        ja Error                        ;如果 num_pixels 太大则跳转
        test r8d,1fh
        jnz Error                       ;如果 num_pixels % 32 != 0 则跳转

; 确保 histo 和 pixel_buff 已进行合适的内存对齐
        mov rsi,rcx                     ;rsi = 指向 histo
        test rsi,0fh
        jnz Error                       ;如果 histo 未对齐则跳转
        mov r9,rdx
        test r9,0fh
        jnz Error                       ;如果 pixel_buff 未对齐则跳转                566

; 初始化本地直方图缓冲区（将所有元素清零）
        xor rax,rax
        mov rdi,rsi                     ;rdi 指向 histo
        mov rcx,128                     ;rcx 是 32 位的缓冲长度
        rep stosq                       ;histo 清零
        mov rdi,rbp                     ;rdi 指向 histo2
        mov rcx,128                     ;rcx 是 32 位的缓冲长度
        rep stosq                       ;hito2 清零

; 在处理循环之前进行的初始化
        shr r8d,5                       ;r8d 为像素块的个数
        mov rdi,rbp                     ;rdi 指向 histo2

; 构建直方图
```

```
        align 16                        ;跳转目标地址对齐
next:   movdqa xmm0,[r9]                ;加载像素块
        movdqa xmm2,[r9+16]             ;加载像素块
        movdqa xmm1,xmm0
        movdqa xmm3,xmm2

; 处理像素 0 - 3
        pextrb rax,xmm0,0
        add dword ptr [rsi+rax*4],1     ;像素 0 计数加 1
        pextrb rbx,xmm1,1
        add dword ptr [rdi+rbx*4],1     ;像素 1 计数加 1
        pextrb rcx,xmm0,2
        add dword ptr [rsi+rcx*4],1     ;像素 2 计数加 1
        pextrb rdx,xmm1,3
        add dword ptr [rdi+rdx*4],1     ;像素 3 计数加 1

; 处理像素 4 - 7
        pextrb rax,xmm0,4
        add dword ptr [rsi+rax*4],1     ;像素 4 计数加 1
        pextrb rbx,xmm1,5
        add dword ptr [rdi+rbx*4],1     ;像素 5 计数加 1
        pextrb rcx,xmm0,6
        add dword ptr [rsi+rcx*4],1     ;像素 6 计数加 1
        pextrb rdx,xmm1,7
        add dword ptr [rdi+rdx*4],1     ;像素 7 计数加 1

; 处理像素 8 - 11
        pextrb rax,xmm0,8
        add dword ptr [rsi+rax*4],1     ;像素 8 计数加 1
        pextrb rbx,xmm1,9
567     add dword ptr [rdi+rbx*4],1     ;像素 9 计数加 1
        pextrb rcx,xmm0,10
        add dword ptr [rsi+rcx*4],1     ;像素 10 计数加 1
        pextrb rdx,xmm1,11
        add dword ptr [rdi+rdx*4],1     ;像素 11 计数加 1

; 处理像素 12 - 15
        pextrb rax,xmm0,12
        add dword ptr [rsi+rax*4],1     ;像素 12 计数加 1
        pextrb rbx,xmm1,13
        add dword ptr [rdi+rbx*4],1     ;像素 13 计数加 1
        pextrb rcx,xmm0,14
        add dword ptr [rsi+rcx*4],1     ;像素 14 计数加 1
        pextrb rdx,xmm1,15
        add dword ptr [rdi+rdx*4],1     ;像素 15 计数加 1

; 处理像素 16 - 19
        pextrb rax,xmm2,0
        add dword ptr [rsi+rax*4],1     ;像素 16 计数加 1
        pextrb rbx,xmm3,1
        add dword ptr [rdi+rbx*4],1     ;像素 17 计数加 1
        pextrb rcx,xmm2,2
        add dword ptr [rsi+rcx*4],1     ;像素 18 计数加 1
        pextrb rdx,xmm3,3
        add dword ptr [rdi+rdx*4],1     ;像素 19 计数加 1

; 处理像素 20 - 23
        pextrb rax,xmm2,4
        add dword ptr [rsi+rax*4],1     ;像素 20 计数加 1
        pextrb rbx,xmm3,5
        add dword ptr [rdi+rbx*4],1     ;像素 21 计数加 1
        pextrb rcx,xmm2,6
```

```
        add dword ptr [rsi+rcx*4],1         ;像素 22 计数加 1
        pextrb rdx,xmm3,7
        add dword ptr [rdi+rdx*4],1         ;像素 23 计数加 1

; 处理像素 24 - 27
        pextrb rax,xmm2,8
        add dword ptr [rsi+rax*4],1         ;像素 24 计数加 1
        pextrb rbx,xmm3,9
        add dword ptr [rdi+rbx*4],1         ;像素 25 计数加 1
        pextrb rcx,xmm2,10
        add dword ptr [rsi+rcx*4],1         ;像素 26 计数加 1
        pextrb rdx,xmm3,11
        add dword ptr [rdi+rdx*4],1         ;像素 27 计数加 1

; 处理像素 28 - 31
        pextrb rax,xmm2,12
        add dword ptr [rsi+rax*4],1         ;像素 28 计数加 1
        pextrb rbx,xmm3,13
        add dword ptr [rdi+rbx*4],1         ;像素 29 计数加 1
        pextrb rcx,xmm2,14
        add dword ptr [rsi+rcx*4],1         ;像素 30 计数加 1
        pextrb rdx,xmm3,15
        add dword ptr [rdi+rdx*4],1         ;像素 31 计数加 1

        add r9,32                           ;r9 指向下一个像素块
        sub r8d,1
        jnz @B                              ;若没有完成则继续循环

; 将两个柱状图合并成最终的直方图
        mov ecx,32                          ;ecx 为迭代次数
        xor rax,rax                         ;rax 为两个柱状直方图存储区的公共偏移量

@@:     movdqa xmm0,xmmword ptr [rsi+rax]       ;加载 histo 的计数
        movdqa xmm1,xmmword ptr [rsi+rax+16]
        paddd xmm0,xmmword ptr [rdi+rax]        ;加上 histo2 中的计数
        paddd xmm1,xmmword ptr [rdi+rax+16]
        movdqa xmmword ptr [rsi+rax],xmm0       ;保存最终结果
        movdqa xmmword ptr [rsi+rax+16],xmm1

        add rax,32
        sub ecx,1
        jnz @@
        mov eax,1                           ;设置成功返回码

Done:   DeleteFrame rbx,rsi,rdi
        ret

Error:  xor eax,eax                         ;设置错误返回码
        jmp Done
Sse64ImageHistogram_ endp
        end
```

示例程序 Sse64ImageHistogram 的 C++ 源代码（清单 20-1）和我们在第 10 章看到的代码几乎完全一样。在程序的开头是一个名为 Sse64ImageHistogramCpp 的函数，该函数使用一个简单的 for 循环构建一幅图像的直方图。函数 Sse64ImageHistogram 中的代码首先加载一幅 8 位灰度的测试图像，然后调用两个图像的直方图处理函数，并比较它们的处理结果以发现任何不一致的地方。该函数还会将直方图的像素计数保存在一个 CSV 文件中，方便后续的处理或使用表格程序绘图。

文件 Sse64ImageHistogram_asm（清单 20-2）中包含一个名为 Sse64ImageHistogram_ 的函数，此函数使用 x86-64 指令集和 SSE4.1 来构建图像的直方图。和第 10 章中的直方图
[569] 示例程序类似，该函数会建立两个中间直方图，然后将它们合并成一个最终的直方图。函数 Sse64ImageHistogram_ 在代码开始处使用宏 _CreateFrame 创建了一个栈帧，包含一个 1024 字节的本地存储区域，该存储区域用来存放其中一个中间直方图。函数调用者提供的缓冲区 histo 用来存放第二个中间直方图和最终的直方图。在宏 _EndProlog 被调用后，对函数参数 histo 和 pixel_buff 进行检查，以确保它们经过了适当的内存对齐。对 num_pixels 也进行了检查，以确保它的大小是合适的。两个中间直方图存储区域中的像素计数器通过 stosq 指令被初始化成 0。

Sse64ImageHistogram_ 函数中的主循环和与其对应的 32 位程序有一些小的差异，主要是该函数使用了更多的通用寄存器。在主循环一开始，就使用两个 movdqa 指令往寄存器 XMM0/XMM1 和 XMM2/XMM3 中加载下一个能存放 32 个像素的区域。指令 pextrb rax, xmm0, 0 从 XMM0 中提取像素 0，然后把它复制到寄存器 RAX 中（RAX 中的高位被设成 0）。指令 add dword ptr[rsi + rax*4], 1 在第一个中间直方图中将相应的像素计数器数值更新。接下来的两条指令 pextrb rbx, xmm1,1 和 add dword ptr [rdi+rbx*4], 1 以同样的方式使用第二个中间直方图处理了像素 1。这种像素处理方法在这个循环中一直被重复使用。

在主循环结束以后，两个中间直方图中的像素计数值被加在一起以创建最终的直方图。_DeleteFrame 宏用来释放本地栈空间并恢复之前保存的非易变通用寄存器。输出 20-1 显示了示例程序 Sse64ImageHistogram 的执行结果。

输出 20-1 示例程序 Sse64ImageHistogram

```
Results for Sse64ImageHistogram()
  Histograms are identical

Benchmark times saved to file _Sse64ImageHistogramTimed.csv
```

表 20-1 包含了 Sse64ImageHistogram 程序的一些计时测量结果。该表中的测量结果和表 10-1 中的测量结果本质上是一样的，只不过表 10-1 显示的是 32 位版本直方图生成算法的测试结果。由于构建直方图的算法受限于其访问的内容，且一条 add 指令只能更新每个中间直方图中的一个像素计数器，所以这样的结果是在预料中的。

表 20-1 示例程序 Sse64ImageHistogram 中的直方图函数处理 TestImage1.bmp 的平均执行时间（单位：微秒）

CPU	C++	x86-SSE-64
Intel Core i7-4770	300	234
Intel Core i7-4600U	354	278
[570] Intel Core i3-2310M	679	485

20.1.2 图像转换

为了实现特定的图像处理算法，有时候要将一个 8 位灰度图像的像素从整型数转换成单精度浮点数，或者逆向转换。本节中的示例程序就演示了如何利用 x86-SSE 指令集来完成这

序的工作。清单 20-3 和清单 20-4 分别给出了示例程序 Sse64ImageConvert 的 C++ 和汇编语言源代码。

清单 20-3 Sse64ImageConvert.cpp

```
#include "stdafx.h"
#include "MiscDefs.h"
#include <malloc.h>
#include <stdlib.h>

extern "C" Uint32 NUM_PIXELS_MAX = 16777216;
extern "C" bool ImageUint8ToFloat_(float* des, const Uint8* src, Uint32↩
num_pixels);
extern "C" bool ImageFloatToUint8_(UInt8* des, const float* src, Uint32↩
num_pixels);

bool ImageUint8ToFloatCpp(float* des, const Uint8* src, Uint32 num_pixels)
{
    // 确保 num_pixels 的值是有效的
    if ((num_pixels == 0) || (num_pixels > NUM_PIXELS_MAX))
        return false;
    if (num_pixels % 32 != 0)
        return false;

    // 确保 src 和 des 是 16 字节边界对齐的
    if (((uintptr_t)src & 0xf) != 0)
        return false;
    if (((uintptr_t)des & 0xf) != 0)
        return false;

    // 转换图像
    for (Uint32 i = 0; i < num_pixels; i++)
        des[i] = src[i] / 255.0f;

    return true;
}

bool ImageFloatToUint8Cpp(Uint8* des, const float* src, Uint32 num_pixels)
{
    // 确保 num_pixels 的值是有效的
    if ((num_pixels == 0) || (num_pixels > NUM_PIXELS_MAX))
        return false;
    if (num_pixels % 32 != 0)
        return false;

    // 确保 src 和 des 是 16 字节边界对齐的
    if (((uintptr_t)src & 0xf) != 0)
        return false;
    if (((uintptr_t)des & 0xf) != 0)
        return false;

    for (Uint32 i = 0; i < num_pixels; i++)
    {
        if (src[i] > 1.0f)
            des[i] = 255;
        else if (src[i] < 0.0)
            des[i] = 0;
        else
            des[i] = (Uint8)(src[i] * 255.0f);
    }

    return true;
```

571

```
}

Uint32 ImageCompareFloat(const float* src1, const float* src2, Uint32↵
num_pixels)
{
    Uint32 num_diff = 0;
    for (Uint32 i = 0; i < num_pixels; i++)
    {
        if (src1[i] != src2[i])
            num_diff++;
    }
    return num_diff;
}

Uint32 ImageCompareUint8(const Uint8* src1, const Uint8* src2, Uint32↵
num_pixels)
{
    Uint32 num_diff = 0;
    for (Uint32 i = 0; i < num_pixels; i++)
    {
        // 由于浮点运算存在一些小的误差，像素值允许差 1
        if (abs((int)src1[i] - (int)src2[i]) > 1)
            num_diff++;
    }
    return num_diff;
}

572

void ImageUint8ToFloat(void)
{
    const Uint32 num_pixels = 1024;
    Uint8* src = (Uint8*)_aligned_malloc(num_pixels * sizeof(Uint8), 16);
    float* des1 = (float*)_aligned_malloc(num_pixels * sizeof(float), 16);
    float* des2 = (float*)_aligned_malloc(num_pixels * sizeof(float), 16);

    srand(12);

    for (Uint32 i = 0; i < num_pixels; i++)
        src[i] = (Uint8)(rand() % 256);

    bool rc1 = ImageUint8ToFloatLpp(des1, src, num_pixels);
    bool rc2 = ImageUint8ToFloat (des2, src, num_pixels);

    if (!rc1 || !rc2)
    {
        printf("Invalid return code - |%d, %d|\n", rc1, rc2);
        return;
    }

    Uint32 num_diff = ImageCompareFloat(des1, des2, num_pixels);
    printf("\nResults for ImageUint8ToFloat\n");
    printf("  num_diff = %u\n", num_diff);

    _aligned_free(src);
    _aligned_free(des1);
    _aligned_free(des2);
}

void ImageFloatToUint8(void)
{
    const Uint32 num_pixels = 1024;
    float* src = (float*)_aligned_malloc(num_pixels * sizeof(float), 16);
```

```
    Uint8* des1 = (Uint8*) aligned_malloc(num_pixels * sizeof(Uint8), 16);
    Uint8* des2 = (Uint8*) aligned_malloc(num_pixels * sizeof(Uint8), 16);

    // 初始化输入缓冲区 src。出于测试的目的，src 中的前几个元素被设成已知的值
    src[0] = 0.125f;        src[8] = 0.01f;
    src[1] = 0.75f;         src[9] = 0.99f;
    src[2] = -4.0f;         src[10] = 1.1f;
    src[3] = 3.0f;          src[11] = -1.1f;
    src[4] = 0.0f;          src[12] = 0.99999f;
    src[5] = 1.0f;          src[13] = 0.5f;
    src[6] = -0.01f;        src[14] = -0.0;
    src[7] = +1.01f;        src[15] = .333333f;                    573
    srand(20);
    for (Uint32 i = 16; i < num_pixels; i++)
        src[i] = (float)rand() / RAND_MAX;

    bool rc1 = ImageFloatToUint8Cpp(des1, src, num_pixels);
    bool rc2 = ImageFloatToUint8_(des2, src, num_pixels);

    if (!rc1 || !rc2)
    {
        printf("Invalid return code - [%d, %d]\n", rc1, rc2);
        return;
    }

    Uint32 num_diff = ImageCompareUint8(des1, des2, num_pixels);
    printf("\nResults for ImageFloatToUint8\n");
    printf("  num_diff = %u\n", num_diff);

    _aligned_free(src);
    _aligned_free(des1);
    _aligned_free(des2);
}

int _tmain(int argc, _TCHAR* argv[])
{
    ImageUint8ToFloat();
    ImageFloatToUint8();
    return 0;
}
```

清单 20-4 Sse64ImageConvert_.asm

```
        include <MacrosX86-64.inc>
        extern NUM_PIXELS_MAX:dword

        .const
; 后面的值都必须以 16 字节边界对齐
Uint8ToFloat        real4 255.0, 255.0, 255.0, 255.0
FloatToUint8Min     real4 0.0, 0.0, 0.0, 0.0
FloatToUint8Max     real4 1.0, 1.0, 1.0, 1.0
FloatToUint8Scale   real4 255.0, 255.0, 255.0, 255.0
        .code

; extern "C" bool ImageUint8ToFloat_(float* des, const Uint8* src, Uint32↵
num_pixels);
;
; 描述：下面的函数将 Uint8 像素缓冲区中的值转换成从一化为 [0.0, 1.0]SPFP
;
; 需要 x86-64 和 SSE2 的支持                                        574
ImageUint8ToFloat_ proc frame
```

```
        _CreateFrame U2F ,0,64
        _SaveXmmRegs xmm10,xmm11,xmm12,xmm13
        _EndProlog

; 确保 num_pixels 为值是有效的且像素缓冲区边界对齐
        test r8d,r8d
        jz Error                            ;若 num_pixels 为 0 则跳转
        cmp r8d,[NUM_PIXELS_MAX]
        ja Error                            ;若 num_pixels 太大则跳转
        test r8d,1Fh
        jnz Error                           ;若 num_pixels % 32 != 0 则跳转
        test rcx,0Fh
        jnz Error                           ;若 des 没有对齐则跳转
        test rdx,0Fh
        jnz Error                           ;若 src 没有对齐则跳转

; 初始化循环处理所用的寄存器
        shr r8d,5                           ;计算像素区域的数量
        movaps xmm4,xmmword ptr [Uint8ToFloat] ;xmm4 = 255.0f (组合值)
        pxor xmm5,xmm5                      ;xmm5 = 0 (组合值)
        align 16

; 函数下一个存放 32 个像素的区域
@@:     movdqa xmm0,xmmword ptr [rdx]           ;xmm0 指向像素区域
        movdqa xmm10,xmmword ptr [rdx+16]       ;xmm10 指向像素区域

; 将 xmm0 中的像素值由无符号 8 位扩展为无符号 32 位
        movdqa xmm2,xmm0
        punpcklbw xmm0,xmm5
        punpckhbw xmm2,xmm5                 ;xmm2 和 xmm0 都存放了 8 个 16 位像素值
        movdqa xmm1,xmm0
        movdqa xmm3,xmm2
        punpcklwd xmm0,xmm5
        punpckhwd xmm1,xmm5
        punpcklwd xmm2,xmm5
        punpckhwd xmm3,xmm5                 ;xmm3:xmm0 都存放了 16 个 32 位像素值

; 将 xmm10 中的像素值由无符号 8 位扩展为无符号 32 位
        movdqa xmm12,xmm10
        punpcklbw xmm10,xmm5
        punpckhbw xmm12,xmm5                ;xmm12 和 xmm10 都存放了 8 个 16 位像素值
        movdqa xmm11,xmm10
        movdqa xmm13,xmm12
        punpcklwd xmm10,xmm5
        punpckhwd xmm11,xmm5
        punpcklwd xmm12,xmm5
575     punpckhwd xmm13,xmm5                ;xmm13:xmm10 都存放了 16 个 32 位像素值

; 将像素值由 32 位整型转换成 SPFP
        cvtdq2ps xmm0,xmm0
        cvtdq2ps xmm1,xmm1
        cvtdq2ps xmm2,xmm2
        cvtdq2ps xmm3,xmm3                  ;xmm3:xmm0 都存放了 16 个 SPFP 像素值
        cvtdq2ps xmm10,xmm10
        cvtdq2ps xmm11,xmm11
        cvtdq2ps xmm12,xmm12
        cvtdq2ps xmm13,xmm14                ;xmm13:xmm10 都存放了 16 个 SPFP 像素值

; 将所有像素值归一化为 [0.0, 1.0] 并保存结果
        divps xmm0,xmm4
        movaps xmmword ptr [rcx],xmm0       ;保存像素 0 - 3
        divps xmm1,xmm4
```

```
        movaps xmmword ptr [rcx+16],xmm1    ;保存像素 4 - 7
        divps xmm2,xmm4
        movaps xmmword ptr [rcx+32],xmm2    ;保存像素 8 - 11
        divps xmm3,xmm4
        movaps xmmword ptr [rcx+48],xmm3    ;保存像素 12 - 15

        divps xmm10,xmm4
        movaps xmmword ptr [rcx+64],xmm10   ;保存像素 16 - 19
        divps xmm11,xmm4
        movaps xmmword ptr [rcx+80],xmm11   ;保存像素 20 - 23
        divps xmm12,xmm4
        movaps xmmword ptr [rcx+96],xmm12   ;保存像素 24 - 27
        divps xmm13,xmm4
        movaps xmmword ptr [rcx+112],xmm13  ;保存像素 28 - 31

        add rdx,32                          ;更新 src 指针
        add rcx,128                         ;更新 des 指针
        sub r8d,1
        jnz @@                              ;一直循环直到完成
        mov eax,1                           ;设置成功返回码

Done:   _RestoreXmmRegs xmm10,xmm11,xmm12,xmm13
        _DeleteFrame
        ret

Error:  xor eax,eax                         ;设置错误返回码
        jmp done
ImageUint8ToFloat_ endp

; extern "C" bool ImageFloatToUint8_(Uint8* des, const float* src, Uint32↩
num_pixels);
;                                                                          S76

; 描述：下面的函数将归一化的 [0.0, 1.0]SPFP 像素缓冲区转换成 Uint8 型
;
; 需要：   x86-64 和 SSE4.1 的支持

ImageFloatToUint8_ proc frame
        _CreateFrame F2U_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; 确保 num_pixels 的值合法且像素缓冲区是边界对齐的
        test r8d,r8d
        jz Error                            ;若 num_pixels 为 0 则跳转
        cmp r8d,[NUM_PIXELS_MAX]
        ja Error                            ;若 num_pixels 的值太大则跳转
        test r8d,1fh
        jnz Error                           ;若 num_pixels % 32 != 0 则跳转
        test rcx,0fh
        jnz Error                           ;若没有对齐则跳转
        test rdx,0fh
        jnz Error                           ;若 src 没有对齐则跳转

; 将所需的组合常量加载到寄存器中
        movaps xmm5,xmmword ptr [FloatToUint8Scale] ;xmm5 = 255.0 (组合值)
        movaps xmm6,xmmword ptr [FloatToUint8Min]   ;xmm6 = 0.0 (组合值)
        movaps xmm7,xmmword ptr [FloatToUint8Max]   ;xmm7 = 1.0 (组合值)

        shr r8d,4                           ;像素区域的个数
@@1:    mov r9d,4                           ;每个像素区域中的 4 个字节像素个数
```

```
; 将 16 个浮点像素值转换成 Uint8 型
LP2:    movaps xmm0,xmmword ptr [rdx]       ;xmm0 指向 4 位字节缓冲区
        movaps xmm1,xmm0
        cmpltps xmm1,xmm6                   ;像素值与 0.0 比较
        andnps xmm1,xmm0                    ;将值小于 0.0 的像素设为 0.0
        movaps xmm0,xmm1                    ;保存结果

        cmpnleps xmm1,xmm7                  ;像素值与 1.0 比较
        movaps xmm2,xmm1
        andps xmm1,xmm7                     ;将值大于 1.0 的像素被剪为 1.0
        andnps xmm2,xmm0                    ;xmm2 大值小于等于 1.0 的像素个数
        orps xmm2,xmm1                      ;xmm2 大量经被剪过为像素个数
        mulps xmm2,xmm5                     ;xmm2 大位于区间 [0.0, 255.0] 内的浮点像素个数

        cvtps2dq xmm1,xmm2                  ;xmm1 为位于区间 [0, 255] 内的 32 位像素个数
        packusdw xmm1,xmm1                  ;xmm1[63:0] 用于存放 16 位像素值
577     packuswb xmm1,xmm1                  ;xmm1[31:0] 用于存放 8 位像素值

; 保存当前 8 位像素值中的 4 位字节
        pextrd eax,xmm1,0                   ;eax 指向新的 4 位字节像素值
        psrldq xmm3,4                       ;针对新的 4 位字节对 xmm3 进行调整
        pinsrd xmm3,eax,3                   ;xmm3[127:96] 用于保存新的 4 位字节

        add rdx,16                          ;更新 src 指针
        sub r9d,1
        jnz LP2                             ;循环直至结束

; 保存当前的像素区域 ( 16 个像素值 )
        movdqa xmmword ptr [rcx],xmm3       ;保存当前像素区域
        add rcx,16                          ;更新 des 指针
        sub r8d,1
        jnz LP1                             ;循环直至结束
        mov eax,1                           ;设置成功返回码

Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret

Error:  mov eax,eax                         ;设置错误返回码
        jmp Done
ImageFloatToUint8_ endp
        end
```

在文件 Sse64ImageConvert.cpp（清单 20-3）的开始，有一个名叫 ImageUint8ToFloat-Cpp 的函数。该函数将存储区域 src 中的所有像素从 Uint8[0, 255] 转换成单精度浮点数 [0.0, 1.0]。这个函数中有一个简单的循环用来对 src 中的每一个像素进行计算：des[i] = src[i] / 255（i 为像素）。接下来的函数 ImageFloatToUint8Cpp 进行反向操作。值得注意的是，这个函数会将所有值大于 1.0 和小于 0.0 的浮点像素都进行裁剪。接下来的两个函数 ImageCompareFloat 和 ImageCompareUint8 用来在转换操作之后比较两个单精度浮点数或 Uint8 像素缓冲区是否相等。注意后一个函数 ImageCompareUint8 允许 Uint8 像素值出现一次误差，这种误差是由 C++ 和汇编语言像素转换函数之间进行浮点运算的微小偏差而产生的（还记得吗？在第 3 章中我们曾经解释过，浮点运算并不总是满足结合律）。

函数 ImageUint8ToFloat 建立了一个测试用例，检验 C++ 和汇编语言版本从 Uint8 到浮点数的转换函数。ImageFloatToUint8 是一个与之类似的函数，用来测试对应的浮点数到 Uint8 的转换函数。值得一提的是，为了测试上述像素裁剪的需求，该函数将 src 像素缓冲

区中的前几个值设成了已知的值。在每一个测试用例的转换操作过程中，被检测到的像素的数量都会被显示出来。

汇编语言版本的转换函数 ImageUint8ToFloat_ 列在清单 20-4 中。主处理循环的每次迭代会将 32 个像素从 Uint8 转换成单精度浮点数。这里的像素转换方法一开始就使用一系列的 x86-SSE 非组合指令（punpcklbw、punpckhbw、punpcklwd 和 punpckhwd）将无符号单字节像素值转换成双字节值，接下来双字节的值又被 cvtdq2ps 指令转换成单精度浮点数。最后得到的浮点数值被归一化到 [0.0, 1.0] 并保存在目的缓冲区中。

清单 20-4 中还包含了函数 ImageFloatToUint8_ 的汇编代码。这个转换函数的内部循环使用 cmpltps 和 cmpnleps 指令以及一些布尔逻辑来裁剪不在 [0.0, 1.0] 区间中的所有浮点像素值。图 20-1 中显示了这种方法。被裁剪过的浮点像素值最后被指令 cvtps2dq、packusdw 和 packuswb 转换成无符号的单字节数。最后使用指令 pextrd、psrldq 和 pinsrd 将得到的单字节数保存到了 XMM3[127:96] 中。这一过程会再重复三次，完成后 XMM3 将包含 16 个经过转换的像素。这一组像素将会由 movdqa 指令保存到目的缓冲区中。输出 20-2 显示了示例程序 Sse64ImageConvert 的执行结果。

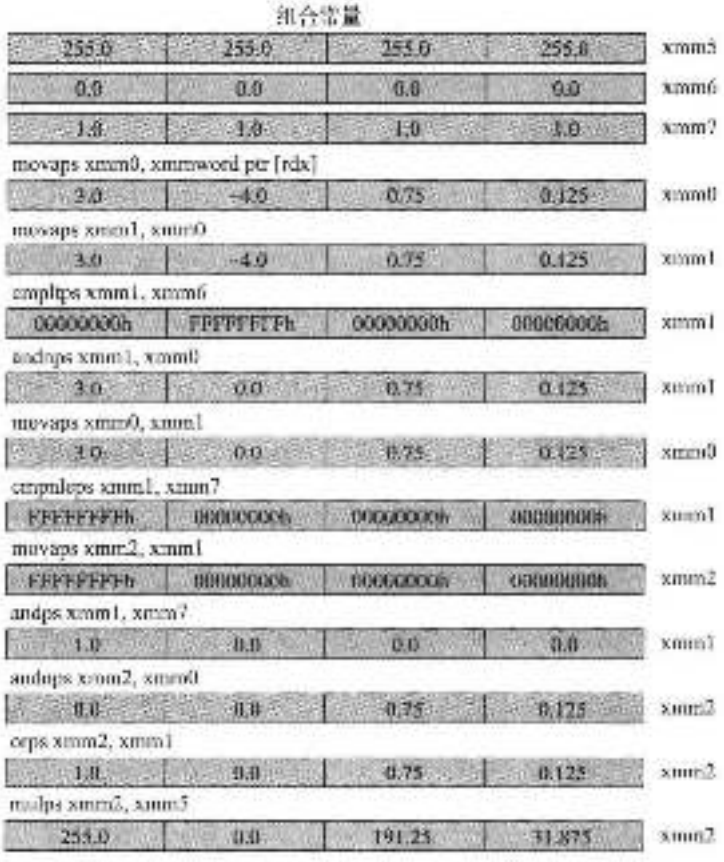


图 20-1 函数 ImageFloatToUint8_ 中使用的浮点像素裁剪方法的图示

输出 20-2 示例程序 Sse64ImageConvert

```
Results for ImageUint8ToFloat
 num_diff = 0
```

```
Results for ImageFloatToUint8
  num diff = 0
```

20.1.3 向量数组

x86-SSE 和 x86-AVX 指令集经常被用来优化某些对大向量数组执行运算的算法。本小节中的示例程序名叫 Sse64VectorArrays。它演示了如何对存放在一个数组中的向量计算向量积。同时该程序还列举了两种不同的数据存储方法以及它们在性能上的实际差异。清单 20-5、清单 20-6 和清单 20-7 中列出了示例程序 Sse64VectorArrays 的源代码。

清单 20-5 Sse64VectorArrays.h

```
// 简化的向量数据结构
typedef struct
{
    float X;        // 向量的 X 分量
    float Y;        // 向量的 Y 分量
    float Z;        // 向量的 Z 分量
    float Pad;      // 将数据结构补齐到 16 字节
} Vector;

// 优化数组的向量数据结构
typedef struct
{
    float* X;       // X 分量指针
    float* Y;       // Y 分量指针
    float* Z;       // Z 分量指针
} VectorSoA;
```

清单 20-6 Sse64VectorArrays.cpp

```
#include "stdafx.h"
#include "Sse64VectorArrays.h"
#include <stdlib.h>

void Sse64VectorCrossProd(void)
{
    const UInt32 num_vectors = 8;
    const size_t vsize1 = num_vectors * sizeof(Vector);
580 const size_t vsize2 = num_vectors * sizeof(float);
Vector* a1 = (Vector*) aligned_malloc(vsize1, 16);
Vector* b1 = (Vector*) aligned_malloc(vsize1, 16);
Vector* c1 = (Vector*) aligned_malloc(vsize1, 16);
VectorSoA a2, b2, c2;

a2.X = (float*) aligned_malloc(vsize2, 16);
a2.Y = (float*) aligned_malloc(vsize2, 16);
a2.Z = (float*) aligned_malloc(vsize2, 16);
b2.X = (float*) aligned_malloc(vsize2, 16);
b2.Y = (float*) aligned_malloc(vsize2, 16);
b2.Z = (float*) aligned_malloc(vsize2, 16);
c2.X = (float*) aligned_malloc(vsize2, 16);
c2.Y = (float*) aligned_malloc(vsize2, 16);
c2.Z = (float*) aligned_malloc(vsize2, 16);

srand(105);
for (UInt32 i = 0; i < num_vectors; i++)
```

```
{
    float a_x = (float)(rand() % 100);
    float a_y = (float)(rand() % 100);
    float a_z = (float)(rand() % 100);
    float b_x = (float)(rand() % 100);
    float b_y = (float)(rand() % 100);
    float b_z = (float)(rand() % 100);

    a1[i].X = a2.X[i] = a_x;
    a1[i].Y = a2.Y[i] = a_y;
    a1[i].Z = a2.Z[i] = a_z;
    b1[i].X = b2.X[i] = b_x;
    b1[i].Y = b2.Y[i] = b_y;
    b1[i].Z = b2.Z[i] = b_z;
    a1[i].Pad = b1[i].Pad = 0;
}

Sse64VectorCrossProd1 (c1, a1, b1, num_vectors);
Sse64VectorCrossProd2 (&c2, &a2, &b2, num_vectors);

bool error = false;
printf("Results for Sse64VectorCrossProd()\n\n");

for (Uint32 i = 0; i < num_vectors && !error; i++)
{
    const char* fs = "[%8.1f %8.1f %8.1f]\n";

    printf("Vector cross product %d\n", i);                    581
        printf("  a1/a2: ");
        printf(fs, a1[i].X, a1[i].Y, a1[i].Z);
        printf("  b1/b2: ");
        printf(fs, b1[i].X, b1[i].Y, b1[i].Z);
        printf("  c1:    ");
        printf(fs, c1[i].X, c1[i].Y, c1[i].Z);
        printf("  c2:    ");
        printf(fs, c2.X[i], c2.Y[i], c2.Z[i]);
        printf("\n");

        bool error_x = c1[i].X != c2.X[i];
        bool error_y = c1[i].Y != c2.Y[i];
        bool error_z = c1[i].Z != c2.Z[i];

        if (error_x || error_y || error_z)
        {
            printf("Compare error at index %d\n", i);
            printf("  %d, %d, %d\n", error_x, error_y, error_z);
            error = true;
        }
    }

    _aligned_free(a1);   _aligned_free(b1);   _aligned_free(c1);
    _aligned_free(a2.X); _aligned_free(a2.Y); _aligned_free(a2.Z);
    _aligned_free(b2.X); _aligned_free(b2.Y); _aligned_free(b2.Z);
    _aligned_free(c2.X); _aligned_free(c2.Y); _aligned_free(c2.Z);
}

int _tmain(int argc, _TCHAR* argv[])
{
    Sse64VectorCrossProd();
    Sse64VectorCrossProdTimed();
}
```

清单 20-7 Sse64VectorArrays_.asm

```
        include <MacrosX86-64.inc>
        .code

; 下面的数据结构定义与 Sse64VectorArray.h 中定义的 VectorSoA 结构一样
VectorSoA struct

X       qword ?     ;向量的 X 分量指针
Y       qword ?     ;向量的 Y 分量指针
Z       qword ?     ;向量的 Z 分量指针
582
VectorSoA ends
; extern "C" bool Sse64VectorCrossProd1_(Vector* c, const Vector* a, const*
Vector* b, Uint32 num_vectors);
;
; 描述：下面的函数用于计算两个 3D 向量的向量积

Sse64VectorCrossProd1_ proc frame
        _CreateFrame Vcp1_,0, 32, r12, r13
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; 进行参数检查
        test r9d,r9d
        jz Error                        ;若 num_vectors == 0 则跳转
        test r9d,3
        jnz Error                       ;若 num_vectors % 4 != 0 则跳转

        test rcx,0fh
        jnz Error                       ;若 a 未对齐则跳转
        test rdx,0fh
        jnz Error                       ;若 b 未对齐则跳转
        test r8,0fh
        jnz Error                       ;若 c 未对齐则跳转
        xor rax,rax                     ;rax 为两个数组的公共偏移量

        align 16
; 从 a 和 b 中加载下一个两个向量
@@:     movaps xmm0,[rdx+rax]           ;a[i]
        movaps xmm1,[r8+rax]            ;b[i]
        movaps xmm2,xmm0
        movaps xmm3,xmm1
        movaps xmm4,[rdx+rax+16]        ;a[i+1]
        movaps xmm5,[r8+rax+16]         ;b[i+1]
        movaps xmm6,xmm4
        movaps xmm7,xmm5

; 计算向量积并保存结果 (# 表示可以忽略)
        shufps xmm0,xmm0,11001001b      ;xmm0 = # | ax | az | ay
        shufps xmm1,xmm1,11010010b      ;xmm1 = # | by | bx | bz
        mulps xmm0,xmm1
        shufps xmm2,xmm2,11010010b      ;xmm2 = # | ay | ax | az
        shufps xmm3,xmm3,11001001b      ;xmm3 = # | bx | bz | by
        mulps xmm2,xmm3
        subps xmm0,xmm2                 ;xmm0 = # | cz | cy | cx
583     movaps [rcx+rax],xmm0           ;保存 c[i]
        shufps xmm4,xmm4,11001001b      ;xmm4 = # | ax | az | ay
        shufps xmm5,xmm5,11010010b      ;xmm5 = # | by | bx | bz
        mulps xmm4,xmm5
        shufps xmm6,xmm6,11010010b      ;xmm6 = # | ay | ax | az
        shufps xmm7,xmm7,11001001b      ;xmm7 = # | bx | bz | by
        mulps xmm6,xmm7
```

```
        subps xmm4,xmm6                 ;xmm4 = w | cz | cy | cx
        movaps [rcx+rax+16],xmm4        ;保存 c[i]

        add rax,32                      ;更新数组偏移
        sub r9d,2
        jnz @@                          ;循环直至结束
        mov eax,1                       ;设置成功返回码

Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame r12, r13
        ret

Error:  xor eax,eax                     ;设置错误返回码
        jmp Done
Sse64VectorCrossProd1_ endp

; extern "C" bool Sse64VectorCrossProd2 (VectorSoA* c, const VectorSoA* a,↩
const VectorSoA* b, Uint32 num_vectors);
;
; 描述： 下面的函数用于计算两个VectorSoA型向量的向量积
Sse64VectorCrossProd2_ proc frame
        _CreateFrame Vcp2_,0,32,rbx,rsi,rdi,r12,r13,r14,r15
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; 确保 num_vectors 的值合法
        test r9d,r9d
        jz Error                        ;若 num_vectors == 0 则跳转
        test r9d,3
        jnz Error                       ;若 num_vectors % 4 != 0 则跳转
        shr r9d,2

; 初始化向量分量数组指针
        xor rax,rax                     ;用于检测未对齐的指针

        mov rbx,[rcx+VectorSoA.X]       ;rbx 为指向 c.X 的指针
        or rax,rbx
        mov rsi,[rcx+VectorSoA.Y]       ;rsi 为指向 c.Y 的指针
        or rax,rsi
        mov rdi,[rcx+VectorSoA.Z]       ;rdi 为指向 c.Z 的指针
        or rax,rdi

        mov r10,[rdx+VectorSoA.X]       ;r10 为指向 a.X 的指针
        or rax,r10
        mov r11,[rdx+VectorSoA.Y]       ;r11 为指向 a.Y 的指针
        or rax,r11
        mov r12,[rdx+VectorSoA.Z]       ;r12 为指向 a.Z 的指针
        or rax,r12

        mov r13,[r8+VectorSoA.X]        ;r13 为指向 b.X 的指针
        or rax,r13
        mov r14,[r8+VectorSoA.Y]        ;r14 为指向 b.Y 的指针
        or rax,r14
        mov r15,[r8+VectorSoA.Z]        ;r15 为指向 b.Z 的指针
        or rax,r15

        and rax,0fh                     ;指针是否未对齐?
        jnz Error                       ;若是，则跳转

        xor rax,rax                     ;rax 为两个数组的公共偏移量
        align 16
```

584

```
; 正数接下来的四个向量
@@:     movaps xmm0,xmmword ptr [r10+rax]   ;xmm0 指向 a.X 分量
        movaps xmm1,xmmword ptr [r11+rax]   ;xmm1 指向 a.Y 分量
        movaps xmm2,xmmword ptr [r12+rax]   ;xmm2 指向 a.Z 分量
        movaps xmm6,xmm1
        movaps xmm7,xmm2
        movaps xmm3,xmmword ptr [r13+rax]   ;xmm3 指向 b.X 分量
        movaps xmm4,xmmword ptr [r14+rax]   ;xmm4 指向 b.Y 分量
        movaps xmm5,xmmword ptr [r15+rax]   ;xmm5 指向 b.Z 分量

; 计算四个向量的向量积
; c.X[i] = a.Y[i] * b.Z[i] - a.Z[i] * b.Y[i]
; c.Y[i] = a.Z[i] * b.X[i] - a.X[i] * b.Z[i]
; c.Z[i] = a.X[i] * b.Y[i] - a.Y[i] * b.X[i]
        mulps xmm6,xmm5                     ;xmm6 = a.Y * b.Z
        mulps xmm7,xmm4                     ;xmm7 = a.Z * b.Y
        subps xmm6,xmm7                     ;xmm6 指向 c.X 分量

        mulps xmm2,xmm3                     ;xmm2 = a.Z * b.X
        mulps xmm5,xmm0                     ;xmm5 = a.X * b.Z
        subps xmm2,xmm5                     ;xmm2 指向 c.Y 分量

        mulps xmm0,xmm4                     ;xmm0 = a.X * b.Y
        mulps xmm1,xmm3                     ;xmm1 = a.Y * b.X
        subps xmm0,xmm1                     ;xmm0 指向 c.Z 分量
        movaps [rdi+rax],xmm0               ;保存 c.Z
        movaps [rsi+rax],xmm2               ;保存 c.Y
        movaps [rbx+rax],xmm6               ;保存 c.X

        add rax,16                          ;更新数组偏移量
        sub r9d,1
        jnz @0                              ;循环直至结束
        mov eax,1                           ;设置成功返回码

Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame rbx,rsi,rdi,r12,r13,r14,r15
        ret

Error:  xor eax,eax                         ;设置错误返回码
        jmp Done
Sse64VectorCrossProd2_ endp
        end
```

两个三维向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的向量积是一个向量 $\boldsymbol{c}$，$\boldsymbol{c}$ 与 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 都成正交关系。向量 $\boldsymbol{c}$ 的 x、y 和 z 分量可以使用下列公式计算：

$$c_x = a_y b_z - a_z b_y \quad c_y = a_z b_x - a_x b_z \quad c_z = a_x b_y - a_y b_x$$

示例程序 Sse64VectorArrays 使用两种不同的存储方法来管理向量数组。第一种方法使用了元素类型为 Vector 的结构体数组（AOS），该结构体的声明在 C++ 头文件 Sse64VectorArrays.h（清单 20-5）中。这个结构体包含了向量分量 X、Y 和 Z 的声明。结构体 Vector 还包括一个 Pad 元素，使整个结构体的大小凑足到 16 字节。第二种数据存储方法使用了三个独立的浮点数组来维护向量分量的值。C++ 头文件 Sse64VectorArrays.h 中声明了一个 VectorSoA 结构体，该结构体实现了第二种数据存储方式，即数组结构体（SOA）。示例程序 Sse64VectorArrays 包含了进行向量积运算的汇编语言函数，为了说明 AOS 和 SOA 两种数据存储方法在指令层面和性能层面的不同，程序分别使用这两种方法进行了运算。

清单 20-6 中列出了示例程序 Sse64VectorArrays 的 C++ 源代码。函数 Sse64VectorCross-

Prod 的功能是分配和初始化所需的向量数据存储空间。同时它还调用了两个不同的汇编语言函数，分别使用 AOS(Sse64VectorCrossProd1_) 和 SOA(Sse64VectorCrossProd2_) 方式来计算向量积。

汇编源文件 Sse64VectorArrays_.asm（清单 20-7）包含了两个向量积计算函数。第一个函数名为 Sse64VectorCrossProd1_，使用元素为结构体 Vector 的数组来计算向量积。在这个函数的主循环里，向量 a[i]、b[i]、a[i+1] 和 b[i−1] 被分别加载到寄存器 XMM0 ~ XMM7 中， 586
然后使用一系列的 shufps、mulps 和 subps 指令来计算向量积 c[i] = a[i] × b[i]。图 20-2 详细显示了这种方法。同样的指令序列被用来计算向量积 c[i + 1] = a[i + 1] × b[i + 1]。注意，进行向量计算的各种指令会忽略每个 XMM 寄存器中的高位浮点部分（127:96 位），这意味着处理器的 SIMD 资源并没有被完全利用。

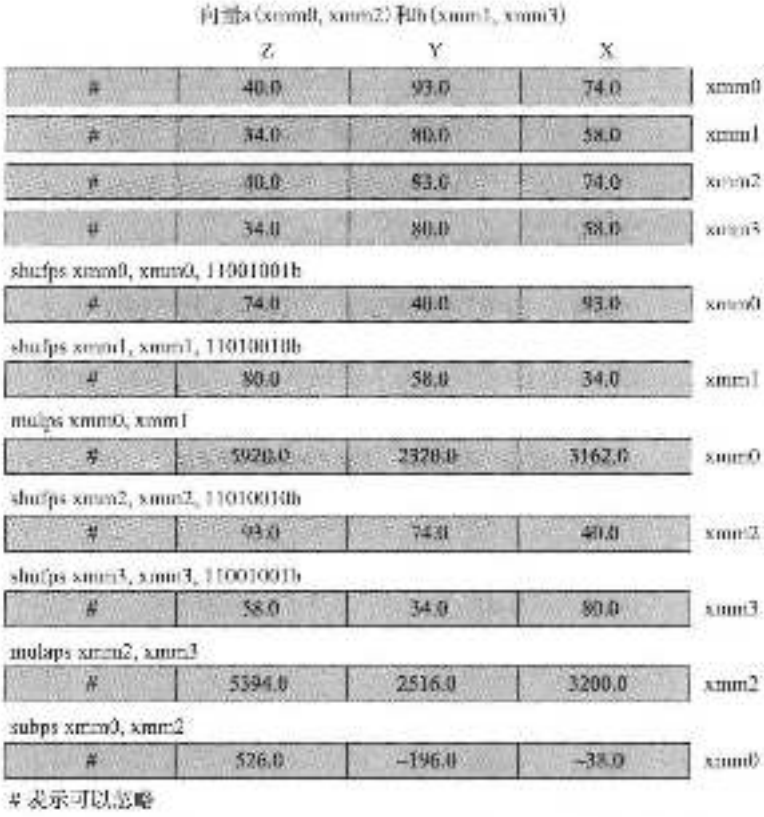


图 20-2 函数 Sse64VectorCrossProd1_ 中使用的向量积计算方法

第二个向量积函数 Sse64VectorCrossProd2_ 使用 VectorSoA 的结构体实例来计算其结果。在这个例子里，采用 SOA 来组织向量分量的数据，这种方式避免了需要进行数据重组的麻烦。此外，该函数还可通过并行计算 4 个向量积来加快计算速度。在函数 Sse64vectorCrossProd2_ 中，主循环一开始就为每一个 XMM 寄存器加载 4 个 X、Y 或 Z 分量。接下来就是使用 6 587
个 mulps 和 3 个 subps 指令来计算 4 个向量积，如图 20-3 所示。最终的结果存放在名为 c 的 VectorSoA 结构缓冲区中。

输出 20-3 显示了示例程序 Sse64VectorArrays 的执行结果。表 20-2 列出了函数 Sse64-VectorCrossProd1_ 和 Sse64VectorCrossProd2_ 计算向量积的时间。对此示例程序而言，SOA 方法比 AOS 方法要快得多。



图 20-3 函数 Sse64VectorCrossProd2_ 中使用的向量积计算方法

输出 20-3 示例程序 Sse64VectorArrays

```
Results for Sse64VectorCrossProd()

Vector cross product 0
  a1/a2: [    74.0     93.0     40.0]
  b1/b2: [    58.0     80.0     34.0]
  c1:    [   -38.0   -196.0    526.0]
  c2:    [   -38.0   -196.0    526.0]

Vector cross product 1
  a1/a2: [    68.0     88.0      8.0]
  b1/b2: [    53.0     87.0     37.0]
  c1:    [  2560.0  -2092.0   1252.0]
  c2:    [  2560.0  -2092.0   1252.0]

Vector cross product 2
  a1/a2: [    55.0     16.0     70.0]
  b1/b2: [    72.0     86.0     49.0]
  c1:    [ -5396.0   2895.0   3578.0]
  c2:    [ -5396.0   2895.0   3578.0]

Vector cross product 3
  a1/a2: [    21.0     89.0     25.0]
  b1/b2: [    95.0     86.0     14.0]
  c1:    [  -904.0   2081.0  -6649.0]
  c2:    [  -904.0   2081.0  -6649.0]

Vector cross product 4
  a1/a2: [    36.0     65.0      5.0]
  b1/b2: [    68.0     92.0     20.0]
```

```
c1:     [   840.0   -380.0  -1108.0]
c2:     [   840.0   -380.0  -1108.0]

Vector cross product 5
  a1/a2: [    31.0     86.0     15.0]
  b1/b2: [    47.0     97.0     94.0]
  c1:    [  6823.0  -2303.0  -1035.0]
  c2:    [  6823.0  -2303.0  -1035.0]

Vector cross product 6
  a1/a2: [    78.0     58.0     41.0]
  b1/b2: [    47.0     48.0     42.0]
  c1:    [   372.0  -1255.0   1018.0]
  c2:    [   372.0  -1255.0   1018.0]

Vector cross product 7
  a1/a2: [    23.0     64.0     86.0]
  b1/b2: [    10.0     42.0     71.0]
  c1:    [   932.0   -773.0    326.0]
  c2:    [   932.0   -773.0    326.0]
```

588 ~ 589

表 20-2 示例程序 Sse64VectorArrays（num_vectors = 50 000）中的向量积计算函数的平均执行时间（单位：秒）

CPU	Sse64VectorCrossProd1_(SOA)	Sse64VectorCrossProd2_(AOS)
Intel Core i7-4770	67	50
Intel Core i7-4600U	106	74
Intel Core i3-2310M	165	126

20.2 x86-AVX-64 编程

本节的示例代码将演示如何在 64 位汇编语言函数中使用 x86-AVX 计算资源，包括标量浮点、组合整型和组合浮点指令。同时还将演示 C++ 库函数和其他一些宏处理技术的运用。

20.2.1 椭圆体计算

本小节中我们将学习一个利用 x86-AVX 中的标量浮点功能来计算椭圆体体积和表面积的示例程序。同时该程序还演示了使用 x86-AVX 指令的 64 位函数如何调用 C++ 库函数。清单 20-8 和清单 20-9 分别列出了示例程序 Avx64CalcEllipsoid 的 C++ 和汇编语言源代码。

清单 20-8 Avx64CalcEllipsoid

```
#include "stdafx.h"
#define _USE_MATH_DEFINES
#include <math.h>

extern "C" bool Avx64CalcEllipsoid_(const double* a, const double* b, const double*
c, int n, double p, double* sa, double* vol);
bool Avx64CalcEllipsoidCpp(const double* a, const double* b, const double*
c, int n, double p, double* sa, double* vol)
{
    if (n <= 0)
        return false;

    for (int i = 0; i < n; i++)
    {
        double a_p = pow(a[i], p);
```

590

```
        double b_p = pow(b[i], p);
        double c_p = pow(c[i], p);

        double temp1 = (a_p * b_p + a_p * c_p + b_p * c_p) / 3;

        sa[i] = 4 * M_PI * pow(temp1, 1.0 / p);
        vol[i] = 4 * M_PI * a[i] * b[i] * c[i] / 3;
    }

    return true;
}

int _tmain(int argc, _TCHAR* argv[])
{
    const int n = 8;
    const double p = 1.6075;
    const double a[n] = { 1, 2, 6, 3, 4,  5, 5, 2};
    const double b[n] = { 1, 2, 1, 7, 2,  6, 5, 7};
    const double c[n] = { 1, 2, 7, 4, 3, 11, 5, 9};
    double sa1[n], vol1[n];
    double sa2[n], vol2[n];

    Avx64CalcEllipsoidCpp(a, b, c, n, p, sa1, vol1);
    Avx64CalcEllipsoid_(a, b, c, n, p, sa2, vol2);

    printf("Results for Avx64CalcEllipsoid\n\n");

    for (int i = 0; i < n; i++)
    {
        printf("\na, b, c: %6.2lf %6.2lf %6.2lf\n", a[i], b[i], c[i]);
        printf("  sa1, vol1: %14.8lf %14.8lf\n", sa1[i], vol1[i]);
        printf("  sa2, vol2: %14.8lf %14.8lf\n", sa2[i], vol2[i]);
    }

    return 0;
}
```

591

清单 20-9 Avx64CalcEllipsoid_.asm

```
        include <MacrosX86-64.inc>
        extern pow:proc

        .const
ymm_1p0  real8 1.0
ymm_3p0  real8 3.0
ymm_4p0  real8 4.0
ymm_pi   real8 3.14159265358979323846
        .code

; extern "C" bool Avx64CalcEllipsoid(const double* a, const double* b,↩
const double* c, int n, double p, double* sa, double* vol);
;
; 描述：一组的函数用于计算椭圆体的表面积和体积
;
; 需要：x86-64 和 AVX 支持

Avx64CalcEllipsoid_ proc frame
        _CreateFrame Lc_,0,144,rbx,rsi,rdi,r12,r13,r14,r15
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

; 进行必要的寄存器初始化。注意这个函数使用了非易变寄存器，因为它需要调用 pow() 函数
```

```
        test r9d,r9d                        ;n <= 0?
        jle Error                           ;如果则是跳转
        mov r12,rcx                         ;r12 为指向 a 的指针
        mov r13,rdx                         ;r13 为指向 b 的指针
        mov r14,r8                          ;r14 为指向 c 的指针
        mov r15d,r9d                        ;r15 = n

        vmovsd xmm12,real8 ptr [rbp+Ce_OffsetStackArgs]  ;xmm12 = p
        vmovsd xmm0,real8 ptr [r8_1o0]
        vdivsd xmm13,xmm0,xmm12             ;xmm13 = 1 / p
        vmovsd xmm1,real8 ptr [r8_4p0]
        vmulsd xmm14,xmm1,[r8_pi]           ;xmm14 = 4 * pi
        vmovsd xmm15,[r8_3p0]               ;xmm15 = 3

        mov rsi,[rbp+Ce_OffsetStackArgs+8]  ;rsi 为指向 sa 的指针
        mov rdi,[rbp+Ce_OffsetStackArgs+16] ;rdi 为指向 vol 的指针
        xor rbx,rbx                         ;rbx 为两个数组的公共像移量
        sub rsp,32                          ;为 pow() 分配内存空间

next:   vmovsd xmm6,real8 ptr [r12+rbx]     ;xmm6 = a
        vmovsd xmm7,real8 ptr [r13+rbx]     ;xmm7 = b
        vmovsd xmm8,real8 ptr [r14+rbx]     ;xmm8 = c                                    592

; 计算椭圆体的体积
        vmulsd xmm0,xmm14,xmm6              ;xmm0 = 4 * pi * a
        vmulsd xmm1,xmm7,xmm8               ;xmm1 = b * c;
        vmulsd xmm0,xmm0,xmm1               ;xmm0 = 4 * pi * a * b * c
        vdivsd xmm0,xmm0,xmm15              ;xmm0 = 4 * pi * a * b * c / 3
        vmovsd real8 ptr [rdi+rbx],xmm0     ;保存椭圆体的体积

; 计算椭圆体的表面积
        vmovsd xmm0,xmm0,xmm6               ;xmm0 = a
        vmovsd xmm1,xmm1,xmm12              ;xmm1 = p
        call pow
        vmovsd xmm9,xmm9,xmm0               ;xmm9 = pow(a,p)

        vmovsd xmm0,xmm0,xmm7               ;xmm0 = b
        vmovsd xmm1,xmm1,xmm12              ;xmm1 = p
        call pow
        vmovsd xmm10,xmm10,xmm0             ;xmm10 = pow(b,p)

        vmovsd xmm0,xmm0,xmm8               ;xmm0 = c
        vmovsd xmm1,xmm1,xmm12              ;xmm1 = p
        call pow                            ;xmm0 = pow(c,p)

        vmulsd xmm1,xmm9,xmm10              ;xmm1 = pow(a,p) * pow(b,p)
        vmulsd xmm2,xmm9,xmm0               ;xmm2 = pow(a,p) * pow(c,p)
        vmulsd xmm3,xmm10,xmm0              ;xmm3 = pow(b,p) * pow(c,p)

        vaddsd xmm0,xmm1,xmm2
        vaddsd xmm0,xmm0,xmm3
        vdivsd xmm0,xmm0,xmm15              ;xmm0 用来保存括号中子表达式的值 (subexpr)
        vmovsd xmm1,xmm1,xmm13              ;xmm1 = 1 / p
        call pow                            ;xmm0 = pow(subexpr,1/p)
        vmulsd xmm0,xmm0,xmm14              ;xmm0 用于保存最终得到的表面积
        vmovsd real8 ptr [rsi+rbx],xmm0     ;保存表面积

; 更新计数器和偏移量，若未完成则继续循环
        add rbx,8
        sub r15,1
        jnz @@
        mov eax,1                           ;设置成功返回码
```

```
Done:   _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame rbx,rsi,rdi,r12,r13,r14,r15
        ret

Error:  xor eax,eax                     ;设置错误返回码
        jmp done
Avx64CalcEllipsoid_ endp
        end
```

593

椭圆体是一个三维立体图形，其横截面是一个椭圆形。椭圆体的大小由它的三个半轴（a、b 和 c）的长度决定。使用这三个半轴的长度计算椭圆体体积是很容易的。然而，精确计算椭圆体的表面积却需要进行多个步骤的椭圆积分。幸运的是，在很多应用场合下，人们可以使用科努兹·汤姆森逼近（参见附录 C）来计算椭圆体的表面积。下面是程序 Avx64CalcEllipsoid 使用的计算椭圆体表面积的公式：

$$V = \frac{4}{3}\pi abc \qquad SA = 4\pi\left[\frac{1}{3}\left(a^p b^p + a^p c^p + b^p c^p\right)\right]^{1/p} \qquad p \approx 1.6075$$

程序 Avx64CalcEllipsoid（清单 20-8）的 C++ 源文件包含两个简单的函数。第一个函数名为 Avx64CalcEllipsoidCpp，用来计算不同椭圆体的体积和表面积，椭圆体的三个半轴的长度由数组 a、b 和 c 给出。另一个函数 _tmain 包含了为 C++ 函数 Avx64CalcEllipsoidCpp 和汇编语言函数 Avx64CalcEllipsoid 初始化测试用例的代码。

清单 20-9 列出了汇编语言函数 Avx64CalcEllipsoid_ 的源代码。在函数序言结束之后，参数 a、b、c 和 n 的值被分别复制到寄存器 R12 ~ R15 中。使用这些寄存器是因为它们的值会被用来给 C++ 库函数 pow 传递参数。接下来的指令块将所需的双精度浮点常数的值放入寄存器 XMM12 ~ XMM15 中。随后是其他一些初始化动作，包括初始化指针寄存器以指向保存计算结果的数组以及为函数 pow 分配栈空间。

程序的主循环首先将半轴的长度值 a[i]、b[i] 和 c[i] 分别加载到寄存器 XMM6 ~ XMM8 中，然后使用 vmulsd 和 vdivsd 指令来计算椭圆体的体积。接下来就是使用前面定义的逼近公式来计算椭圆体的表面积。值得注意的是，在调用 pow 函数之前，程序将所需的参数值复制到了寄存器 XMM0 和 XMM1 中。另外还要注意，在下一次调用 pow 函数之前，Avx64CalcEllispoid_ 会将 pow 函数的返回值保存在一个非易变寄存器中。输出 20-4 显示了示例程序 Avx64CalcEllipsoid 的执行结果。

输出 20-4 示例程序 Avx64CalcEllipsoid

```
Results for Avx64CalcEllipsoid

a, b, c:   1.00   1.00   1.00
  sa1, vol1:     12.56637061     4.18879020
  sa2, vol2:     12.56637061     4.18879020

a, b, c:   2.00   2.00   2.00
  sa1, vol1:     50.26548246    33.51032164
  sa2, vol2:     50.26548246    33.51032164

a, b, c:   6.00   1.00   7.00
  sa1, vol1:    282.73569300   175.92918860
  sa2, vol2:    282.73569300   175.92918860

a, b, c:   3.00   7.00   4.00
  sa1, vol1:    263.60452668   351.85837720
```

594

```
  sa2, vol2:   263.60352668   351.85837720

a, b, c:   4.00   2.00   3.00
  sa1, vol1:   111.60403108   100.53096491
  sa2, vol2:   111.60403108   100.53096491

a, b, c:   5.00   6.00  11.00
  sa1, vol1:   649.98183211  1382.30076758
  sa2, vol2:   649.98183211  1382.30076758

a, b, c:   5.00   5.00   5.00
  sa1, vol1:   314.15926536   523.59877560
  sa2, vol2:   314.15926536   523.59877560

a, b, c:   2.00   7.00   9.00
  sa1, vol1:   452.93733288   527.78756580
  sa2, vol2:   452.93733288   527.78756580
```

20.2.2 RGB 图像处理

接下来的示例程序名为 Avx64CalcRgbMinMax，演示如何计算一幅 RGB 图像的红、绿、蓝三种像素值中的最大值和最小值。同时，该程序还演示了更多的宏处理技术。Avx64CalcRgbMinMax 的 C++ 和汇编语言源代码分别如清单 20-10 和清单 20-11 所示。

清单 20-10 Avx64CalcRgbMinMax.cpp

```
#include "stdafx.h"
#include "MiscDefs.h"
#include <stdlib.h>
#include <malloc.h>

extern "C" bool Avx64CalcRgbMinMax_(UInt8* rgb[3], UInt32 num_pixels, UInt8*
min_vals[3], UInt8 max_vals[3]);

bool Avx64CalcRgbMinMaxCpp(UInt8* rgb[3], UInt32 num_pixels, UInt8*
min_vals[3], UInt8 max_vals[3])
{
    // 确保 num_pixels 的值有效
    if ((num_pixels == 0) || (num_pixels % 32 != 0))
        return false;                                                    595
    // 确保调色板存储区域边界对齐
    for (UInt32 i = 0; i < 3; i++)
    {
        if (((uintptr_t)rgb[i] & 0x1f) != 0)
            return false;
    }

    // 寻找每一个调色板的最大值和最小值
    for (UInt32 i = 0; i < 3; i++)
    {
        min_vals[i] = 255;   max_vals[i] = 0;

        for (UInt32 j = 0; j < num_pixels; j++)
        {
            if (rgb[i][j] < min_vals[i])
                min_vals[i] = rgb[i][j];
            else if (rgb[i][j] > max_vals[i])
                max_vals[i] = rgb[i][j];
        }
    }
```

```
    return true;
}

int _tmain(int argc, _TCHAR* argv[])
{
    const UInt32 n = 1024;
    UInt8* rgb[3];

    rgb[0] = (UInt8*) aligned_malloc(n * sizeof(UInt8), 32);
    rgb[1] = (UInt8*) aligned_malloc(n * sizeof(UInt8), 32);
    rgb[2] = (UInt8*) aligned_malloc(n * sizeof(UInt8), 32);

    for (UInt32 i = 0; i < n; i++)
    {
        rgb[0][i] = 5 + rand() % 245;
        rgb[1][i] = 5 + rand() % 245;
        rgb[2][i] = 5 + rand() % 245;
    }

    // 将 min 和 max 的值设为已知的值（出于测试的目的）
    rgb[0][n / 4] = 4;   rgb[1][n / 2] = 1;       rgb[2][3 * n / 4] = 3;
    rgb[0][n / 3] = 254; rgb[1][2 * n / 5] = 251; rgb[2][n - 1] = 252;

    UInt8 min_vals1[3], max_vals1[3];
    UInt8 min_vals2[3], max_vals2[3];
    Avx64CalcRgbMinMaxCpp(rgb, n, min_vals1, max_vals1);
    Avx64CalcRgbMinMax_(rgb, n, min_vals2, max_vals2);

    printf("Results for Avx64CalcRgbMinMax\n\n");
    printf("             R   G   B\n");
    printf("----------------------\n");
    printf("min_vals1: %3d %3d %3d\n", min_vals1[0], min_vals1[1],←
min_vals1[2]);
    printf("min_vals2: %3d %3d %3d\n", min_vals2[0], min_vals2[1],←
min_vals2[2]);
    printf("\n");
    printf("max_vals1: %3d %3d %3d\n", max_vals1[0], max_vals1[1],←
max_vals1[2]);
    printf("max_vals2: %3d %3d %3d\n", max_vals2[0], max_vals2[1],←
max_vals2[2]);

    _aligned_free(rgb[0]);
    _aligned_free(rgb[1]);
    _aligned_free(rgb[2]);
    return 0;
}
```

596

清单 20-11 Avx64CalcRgbMinMax_.asm

```
        include <MacrosX86-64.inc>

; 256 位常量
ConstVals segment readonly align(32)
InitialPminVal db 32 dup(0ffh)
InitialPmaxVal db 32 dup(00h)
ConstVals ends
        .code

; 宏 _YmmVpextrMinub
;
; 描述：下面的宏定义生成代码，用于从寄存器 YmmSrc 中返回无符号 8 位数值的最小值
```

```
_YmmVpextrVminub macro GprDes,YmmSrc,YmmTmp

; 确保 YmmSrc 和 YmmTmp 不是相同的寄存器
.erridni <YmmSrc>, <YmmTmp>, <Invalid registers>

; 为相应的 XMM 寄存器生成字符串
        YmmSrcSuffix SUBSTR <YmmSrc>,2
        XmmSrc CATSTR <X>,YmmSrcSuffix                                  597
        YmmTmpSuffix SUBSTR <YmmTmp>,2
        XmmTmp CATSTR <X>,YmmTmpSuffix

; 将 YmmSrc 中的 32 字节值减少到最小值
        vextracti128 XmmTmp,YmmSrc,1
        vpminub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc 用于保存最后 16 个最小值

        vpsrldq XmmTmp,XmmSrc,8
        vpminub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc 用于保存最后 8 个最小值

        vpsrldq XmmTmp,XmmSrc,4
        vpminub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc 用于保存最后 4 个最小值

        vpsrldq XmmTmp,XmmSrc,2              ;XmmSrc 用于保存最后 2 个最小值
        vpminub XmmSrc,XmmSrc,XmmTmp        ;

        vpsrldq XmmTmp,XmmSrc,1
        vpminub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc 用于保存最后 1 个最小值

        vpextrb GprDes,XmmSrc,0             ;将最终的最小值保存到 Gpr 中
        endm

; 宏 _YmmVpextrMaxub
;
; 描述：下面的宏定义生成代码，用于从寄存器 YmmSrc 中找到无符号 8 位数值的
;       最大值

_YmmVpextrMaxub macro GprDes,YmmSrc,YmmTmp

; 确保 YmmSrc 和 YmmTmp 不是相同的寄存器
.erridni <YmmSrc>, <YmmTmp>, <Invalid registers>

; 为相应的 XMM 寄存器生成字符串
        YmmSrcSuffix SUBSTR <YmmSrc>,2
        XmmSrc CATSTR <X>,YmmSrcSuffix

        YmmTmpSuffix SUBSTR <YmmTmp>,2
        XmmTmp CATSTR <X>,YmmTmpSuffix

; 将 YmmSrc 中的 32 字节值减少到最大值
        vextracti128 XmmTmp,YmmSrc,1
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc 用于保存最后 16 个最大值

        vpsrldq XmmTmp,XmmSrc,8
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc 用于保存最后 8 个最大值

        vpsrldq XmmTmp,XmmSrc,4
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc 用于保存最后 4 个最大值  598
        vpsrldq XmmTmp,XmmSrc,2
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc 用于保存最后 2 个最大值

        vpsrldq XmmTmp,XmmSrc,1
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc 用于保存最后 1 个最大值
```

```
        vpextrh GprDes,XmmSrc,0             ;将最终的最大值保存到 Gpr 中
        endm

; extern "C" bool Avx64CalcRgbMinMax_(Uint8* rgb[3], Uint32 num_pixels,
Uint8 min_vals[3], Uint8 max_vals[3]);
;
; 描述：下面的函数用于确定每个颜色数组中的最大和最小像素值
;
; 需要 x86-64 和 AVX2 支持

Avx64CalcRgbMinMax_ proc frame
        _CreateFrame CalcMinMax_,0,48,r12
        _SaveXmmRegs xmm6,xmm7,xmm8
        _EndProlog

; 确保 num_pixels 和颜色数组是有效的
        test edx,edx
        jz Error                            ;若 num_pixels == 0 则跳转
        test edx,01fh
        jnz Error                           ;若 num_pixels % 32 != 0, 则跳转

        xor rax,rax
        mov r10,[rcx]                       ;r10 指向 R
        or rax,r10
        mov r11,[rcx+8]                     ;r11 指向 G
        or rax,r11
        mov r12,[rcx+16]                    ;r12 指向 B
        or rax,r12
        test rax,1fh
        jnz Error                           ;若 R、G 或 B 未对齐则跳转

; 初始化循环用到的寄存器
        shr edx,5                           ;edx 为像素区域的个数
        xor rcx,rcx                         ;rcx 为两个数组的公共偏移量

        vmovdqa ymm3,ymmword ptr [InitialPminVal]   ;ymm3 为 R 中的最小值
        vmovdqa ymm4,ymm3                           ;ymm4 为 G 中的最小值
599     vmovdqa ymm5,ymm3                           ;ymm5 为 B 中的最小值
        vmovdqa ymm6,ymmword ptr [InitialPmaxVal]   ;ymm6 为 R 中的最大值
        vmovdqa ymm7,ymm6                           ;ymm7 为 G 中的最大值
        vmovdqa ymm8,ymm6                           ;ymm8 为 B 中的最大值

; 在 RGB 颜色数组中寻找最大值和最小值
@@:     vmovdqa ymm0,ymmword ptr [r10+rcx]  ;ymm0 为 R 中的像素个数
        vmovdqa ymm1,ymmword ptr [r11+rcx]  ;ymm1 为 G 中的像素个数
        vmovdqa ymm2,ymmword ptr [r12+rcx]  ;ymm2 为 B 中的像素个数

        vpminub ymm3,ymm3,ymm0              ;更新 R 中的最小值
        vpminub ymm4,ymm4,ymm1              ;更新 G 中的最小值
        vpminub ymm5,ymm5,ymm2              ;更新 B 中的最小值

        vpmaxub ymm6,ymm6,ymm0              ;更新 R 中的最大值
        vpmaxub ymm7,ymm7,ymm1              ;更新 G 中的最大值
        vpmaxub ymm8,ymm8,ymm2              ;更新 B 中的最大值

        add rcx,32
        sub edx,1
        jnz @8

; 计算最终的 RGB 最小值
        YmmVpextrMinub rax,ymm3,ymm0
```

```
        mov byte ptr [r8],al                ;保存R中的最小值
        _YmmVpextrMinub rax,ymm4,ymm0
        mov byte ptr [r8+1],al              ;保存G中的最小值
        _YmmVpextrMinub rax,ymm5,ymm0
        mov byte ptr [r8+2],al              ;保存B中的最小值

; 计算最终的RGB最大值
        _YmmVpextrMaxub rax,ymm6,ymm1
        mov byte ptr [r9],al                ;保存R中的最大值
        _YmmVpextrMaxub rax,ymm7,ymm1
        mov byte ptr [r9+1],al              ;保存G中的最大值
        _YmmVpextrMaxub rax,ymm8,ymm1
        mov byte ptr [r9+2],al              ;保存B中的最大值

        mov eax,1                           ;设置成功返回码
        vzeroupper

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame r12
        ret

Error:  xor eax,eax                         ;设置错误返回码
        jmp Done
Avx64CalcRgbMinMax_ endp
        end
```

600

C++ 文件 Avx64CalcRgbMinMax.cpp（清单 20-10）的开头是一个名为 Avx64CalcRgbMinMaxCpp 的函数。这个函数包含一个简单的 for 循环，它确定输入的颜色数组中的 RGB 最大值和最小值。这个 C++ 文件还包含 _tmain 函数，它用来初始化测试用的颜色数组。同时这个函数还调用了 C++ 和汇编语言版本的 RGB 最大最小值计算函数并显示输出结果。

清单 20-11 列出了汇编语言文件 Avx64CalcRgbMinMax_.asm 的源代码。代码段 ConstVals 的声明之后是两个宏定义：_YmmVpextrMinub 和 _YmmVpextrMaxub。这两个宏所生成的代码用于从一个 YMM 寄存器中提取最大和最小的无符号字节值。两个宏的源代码声明是一样的，除了对 vpminub 或 vpmaxub 指令的使用之外。下面我们来详细解释一下 _YmmVpextrMinub 这个宏定义的声明和逻辑。

宏 _YmmVpextrMinub 需要三个参数：一个通用目的寄存器（GprDes），一个 YMM 源寄存器（YmmSrc），一个 YMM 临时寄存器（YmmTmp）。注意 YmmSrc 和 YmmTmp 必须是不同的寄存器。如果它们相同，那么 .erridni 指示符（在不区分大小写的情况下，如果两个符号字面上相同，就会报错）就会在代码编译时报告一个错误。

为了能够生成正确的汇编语言代码，宏 _YmmVpextrMinub 需要一个包含 XMM 寄存器名字的字符串（XmmSrc），这个字符串与所指定的 YmmSrc 寄存器的低位相对应。举例来说，如果 YmmSrc 是“YMM0”，那么宏定义的字符串 XmmSrc 就是“XMM0”。宏指示符 substr（返回一个字符串的子串）和 catstr（连接两个字符串）用于初始化 XmmSrc。语句 YmmSrcSuffix SUBSTR <YmmSrc>, 2 将一个字符串赋给 YmmSrcSuffix，其中不包括宏参数 YmmSrc 的前导字符。下一条语句 XmmSrc CATSTR <X>, YmmSrcSuffix 会在 YmmSrcSuffix 所对应的字符串前面加一个前导的“X”并把它赋给 XmmSrc。同样的一组指示符用来把字符串赋给 XmmTmp。

在初始化所需的宏定义字符串之后，接下来的指令从指定的 YMM 寄存器中提取最小的单字节值。指令 vextracti128 XmmTmp, YmmSrc, 1 将寄存器 YmmSrc 的高 16 字节中的内容复制到 XmmTmp 中。指令 vpminub XmmSrc, XmmSrc, XmmTmp 将最终的 16 个最小值加载到

XmmSrc 中。指令 vpsrldq XmmTmp, XmmSrc, 8 为 XmmSrc 中的值建立一个副本，并把副本中的值向右移位 8 个字节，然后将结果保存到 XmmTmp 中。这样的操作为另一个 vpminub 指令的使用提供了方便，该指令将单字节最小值的数量从 16 个减少到 8 个。vpsrldq 和 vpminub 两个指令被重复使用，直到最后的最小值保存在 XmmSrc 的低位字节中。指令 vpextrb GprDes, XmmSrc, 0 将最终的最小值复制到指定的通用寄存器当中。

函数 Avx64CalcRgbMinMax 使用了寄存器 YMM3 ~ YMM5 和 YMM6 ~ YMM8 分别来维护 RGB 最小值和最大值。在主循环的每次迭代中，一系列的 vpminub 和 vpmaxub 指令被用来更新当前的 RGB 最小值和最大值。到主循环结束的时候，上面提到的 YMM 寄存器包含了最终的 32 个 RGB 的最小像素值和最大像素值。然后，_YmmVpextrMinub 和 _YmmVpextrMaxub 两个宏被用来提取最终的 RGB 最小像素值和最大像素值。这些值最终被
601 保存到指定的数组中作为运算结果。

函数 Avx64CalcRgbMinMax_ 中使用了 YMM 寄存器，这意味着在函数结语之前需要调用 vzeroupper。结语以 _RestoreXmmRegs 的宏调用语句作为开始。请注意，在结语之前使用 vzeroall 指令是不合理的，因为 64 位函数必须保存寄存器 XMM6 ~ XMM15 中的内容。不过，vzeroall 指令仍可被用于 64 位函数体内，只要非易变 XMM 寄存器的内容能够被保存。输出 20-5 显示了示例程序 Avx64CalcRgbMinMax 的执行结果。

输出 20-5 示例程序 Avx64CalcRgbMinMax

```
Results for Avx64CalcRgbMinMax

            R   G   B
-----------------------
min_vals1:  4   1   3
min_vals2:  4   1   3

max_vals1: 254 251 252
max_vals2: 254 251 252
```

20.2.3 矩阵求逆

在第 9 章，我们已经学习了如何利用 x86-SSE 指令集计算两个 4 × 4 单精度浮点数矩阵的积。本小节我们将学习有关矩阵的另外一个示例程序，该程序可以利用 64 位 x86-AVX 指令集来计算一个 4 × 4 单精度浮点数矩阵的逆矩阵。清单 20-12 和清单 20-13 分别列出了示例程序 Avx64CalcMat4 × 4Inv 的 C++ 和汇编语言源代码。

清单 20-12 Avx64CalcMat4 × 4Inv.cpp

```
#include "stdafx.h"
#include <math.h>
#include "Avx64CalcMat4x4Inv.h"

//#define MAT_INV_DEBUG     // 去掉注释可以启用额外的打印输出

bool Mat4x4InvCpp(Mat4x4 m_inv, Mat4x4 m, float epsilon, bool* is_singular)
{
    __declspec(align(32)) Mat4x4 m2;
    __declspec(align(32)) Mat4x4 m3;
    __declspec(align(32)) Mat4x4 m4;
    float t1, t2, t3, t4;
602 float c1, c2, c3, c4;
```

```
    // 确保所有矩阵都是对齐的
    if (((uintptr_t)m_inv & 0x1f) != 0)
        return false;
    if (((uintptr_t)m & 0x1f) != 0)
        return false;

    // 计算所需的矩阵迹（主对角线上所有元素之和）
    Mat4x4Mul(m2, m, m);
    Mat4x4Mul(m3, m2, m);
    Mat4x4Mul(m4, m3, m);
    t1 = Mat4x4Trace(m);
    t2 = Mat4x4Trace(m2);
    t3 = Mat4x4Trace(m3);
    t4 = Mat4x4Trace(m4);

#ifdef MAT_INV_DEBUG
    printf("t1: %16e\n", t1);
    printf("t2: %16e\n", t2);
    printf("t3: %16e\n", t3);
    printf("t4: %16e\n", t4);
#endif

    c1 = -t1;
    c2 = -1.0f / 2.0f * (c1 * t1 + t2);
    c3 = -1.0f / 3.0f * (c2 * t1 + c1 * t2 + t3);
    c4 = -1.0f / 4.0f * (c3 * t1 + c2 * t2 + c1 * t3 + t4);

#ifdef MAT_INV_DEBUG
    printf("c1: %16e\n", c1);
    printf("c2: %16e\n", c2);
    printf("c3: %16e\n", c3);
    printf("c4: %16e\n", c4);
#endif

    // 确保矩阵是非奇异的
    if ((*is_singular = (fabs(c4) < epsilon)) != false)
        return false;

    // 计算 -1.0 / c4 * (m3 + c1 * m2 + c2 * m + c3 * I)
    __declspec(align(32)) Mat4x4 I;
    __declspec(align(32)) Mat4x4 tempA, tempB, tempC, tempD;

    Mat4x4SetI(I);
    Mat4x4MulScalar(tempA, I, c3);
    Mat4x4MulScalar(tempB, m, c2);
    Mat4x4MulScalar(tempC, m2, c1);
    Mat4x4Add(tempD, tempA, tempB);
    Mat4x4Add(tempD, tempD, tempC);
    Mat4x4Add(tempD, tempD, m3);
    Mat4x4MulScalar(m_inv, tempD, -1.0f / c4);
    return true;
}

void Avx64Mat4x4Inv(Mat4x4 m, const char* s)
{
    Mat4x4Printf(m, s);

    for (int i = 0; i <= 1; i++)
    {
        const float epsilon = 1.0e-9f;
        __declspec(align(32)) Mat4x4 m_inv;
```

603

```
        __declspec(align(32)) Mat4x4 r_ver;
        bool rc, is_singular;

        if (i == 0)
        {
            printf("\nCalculating inverse matrix - Mat4x4InvCpp\n");
            rc = Mat4x4InvCpp(m_inv, m, epsilon, &is_singular);
        }
        else
        {
            printf("\nCalculating inverse matrix - Mat4x4Inv_\n");
            rc = Mat4x4Inv_(m_inv, m, epsilon, &is_singular);
        }

        if (!rc)
        {
            if (is_singular)
                printf("Matrix 'm' is singular\n");
            else
                printf("Error occurred during calculation of matrix=
                inverse\n");
        }
        else
        {
          Mat4x4Printf(m_inv, "\nInverse matrix\n");
          Mat4x4Mul(m_ver, m_inv, m);
          Mat4x4Printf(m_ver, "\nInverse matrix verification\n");
        }
    }
}

void Avx64CalcMat4x4Inv(void)
{
    __declspec(align(32)) Mat4x4 r;

    printf("Results for Avx64CalcMat4x4Inv\n");

    Mat4x4SetRow(r, 0, 2, 7, 3, 4);
    Mat4x4SetRow(r, 1, 5, 9, 6, 4.75);
    Mat4x4SetRow(r, 2, 6.5, 3, 4, 10);
    Mat4x4SetRow(r, 3, 7, 5.25, 8.125, 6);
    Avx64Mat4x4Inv(r, "\nTest Matrix #1\n");

    Mat4x4SetRow(r, 0, 0.5, 12, 17.25, 4);
    Mat4x4SetRow(r, 1, 5, 2, 6.75, 8);
    Mat4x4SetRow(r, 2, 13.125, 1, 3, 9.75);
    Mat4x4SetRow(r, 3, 16, 1.625, 7, 0.25);
    Avx64Mat4x4Inv(r, "\nTest Matrix #2\n");

    Mat4x4SetRow(r, 0, 2, 0, 0, 1);
    Mat4x4SetRow(r, 1, 0, 4, 5, 0);
    Mat4x4SetRow(r, 2, 0, 0, 0, 7);
    Mat4x4SetRow(r, 3, 0, 0, 0, 6);
    Avx64Mat4x4Inv(r, "\nTest Matrix #3\n");
}

int _tmain(int argc, _TCHAR* argv[])
{
#ifdef _DEBUG
    Avx64CalcMat4x4InvTest();
#endif
    Avx64CalcMat4x4Inv();
```

```
    Avx64CalcMat4x4InvTimed();
    return 0;
}
```

清单 20-13 Avx64CalcMat4x4Inv_.asm

```
        include <MacrosX86-64.inc>

ConstVals segment readonly align(32) 'const'
VpermpsTranspose    dword 0,4,1,5,2,6,3,7
VpermsTrace         dword 0,2,5,7,0,0,0,0

Mat4x4I         real4 1.0, 0.0, 0.0, 0.0
                real4 0.0, 1.0, 0.0, 0.0
                real4 0.0, 0.0, 1.0, 0.0
                real4 0.0, 0.0, 0.0, 1.0                                    605

r4 SignBitMask  dword 80000000h,80000000h,80000000h,80000000h
r4 AbsMask      dword 7fffffffh,7fffffffh,7fffffffh,7fffffffh

r4_1p0          real4 1.0
r4_N1p0         real4 -1.0
r4_N0p5         real4 -0.5
r4_N0p3333      real4 -0.3333333333
r4_N0p25        real4 -0.25
ConstVals ends
        .code

; 宏定义 _Mat4x4TraceYmm
;
; 描述：下面的宏定义生成代码，用于计算一个存放于 ymm1:ymm0 中的 4x4 SPFP 矩阵的迹

_Mat4x4TraceYmm macro
        vblendps ymm0,ymm0,ymm1,84h             ;复制对角线元素到 ymm0 中
        vmovdqa ymm2,ymmword ptr [VpermsTrace]
        vpermps ymm1,ymm2,ymm0                  ;ymm1[127:0] 用于对角线元素
        vhaddps ymm0,ymm1,ymm1
        vhaddps ymm0,ymm0,ymm0                  ;ymm0[31:0] 用于存放矩阵的迹
        endm

; Mat4x4Mul
;
; 描述：下面的过程用于计算两个 4x4 矩阵的积
;
; 输入：        ymm1:ymm0   m1
;               ymm3:ymm2   m2
;
; 输出：        ymm1:ymm0   m1 * m2
;
; 注意：在下面的注释中，m2T 代表矩阵 m2 的转置矩阵

Mat4x4Mul proc private

; 计算 m2 的转置矩阵
        vmovdqa ymm6,ymmword ptr [VpermpsTranspose] ;ymm6 用于存放 vperms 的索引值
        vunpcklps ymm4,ymm2,ymm3
        vunpckhps ymm5,ymm2,ymm3            ;ymm4 用于存放部分转置矩阵
        vpermps ymm2,ymm6,ymm4
        vpermps ymm3,ymm6,ymm5              ;ymm2 用于存放 m2T                  606

; 将 m2T 中的行复制到 ymm*[255:128] 和 ymm*[127:0] 中
        vperm2f128 ymm4,ymm2,ymm2,00000000b     ;ymm4 用于存放 m2T 的第 0 行元素
```

```
        vperm2f128 ymm5,ymm2,ymm2,00010001b     ;ymm5 用于存放 m2T 的第 1 行元素
        vperm2f128 ymm6,ymm3,ymm3,00000000b     ;ymm6 用于存放 m2T 的第 2 行元素
        vperm2f128 ymm7,ymm3,ymm3,00010001b     ;ymm7 用于存放 m2T 的第 3 行元素

; 执行第 0、1 行的 mat4x4 乘法运算
; 所有没有使用低 vdpps 目的寄存器上函数被设成 0
; ymm8[31:0]      - dp(m1.row0, m2T.row0)
; ymm8[159:128]   - dp(m1.row2, m2T.row0)
; ymm9[63:32]     - dp(m1.row0, m2T.row1)
; ymm9[191:160]   - dp(m1.row2, m2T.row1)
; ymm10[95:64]    - dp(m1.row0, m2T.row2)
; ymm10[223:192]  - dp(m1.row2, m2T.row2)
; ymm11[127:96]   - dp(m1.row0, m2T.row3)
; ymm11[255:224]  - dp(m1.row2, m2T.row3)
        vdpps ymm8,ymm0,ymm4,11110001b
        vdpps ymm9,ymm0,ymm5,11110010b
        vdpps ymm10,ymm0,ymm6,11110100b
        vdpps ymm11,ymm0,ymm7,11111000b
        vorps ymm8,ymm8,ymm9
        vorps ymm10,ymm10,ymm11
        vorps ymm0,ymm8,ymm10                   ;保存第 0、1 行的运算结果

; 执行第 2、3 行的 mat4x4 乘法运算
; ymm8[31:0]      - dp(m1.row2, m2T.row0)
; ymm8[159:128]   - dp(m1.row3, m2T.row0)
; ymm9[63:32]     - dp(m1.row2, m2T.row1)
; ymm9[191:160]   - dp(m1.row3, m2T.row1)
; ymm10[95:64]    - dp(m1.row2, m2T.row2)
; ymm10[223:192]  - dp(m1.row3, m2T.row2)
; ymm11[127:96]   - dp(m1.row2, m2T.row1)
; ymm11[255:224]  - dp(m1.row3, m2T.row3)
        vdpps ymm8,ymm1,ymm4,11110001b
        vdpps ymm9,ymm1,ymm5,11110010b
        vdpps ymm10,ymm1,ymm6,11110100b
        vdpps ymm11,ymm1,ymm7,11111000b
        vorps ymm8,ymm8,ymm9
        vorps ymm10,ymm10,ymm11
        vorps ymm1,ymm8,ymm10                   ;保存第 2、3 行的运算结果
        ret
Mat4x4Mul endp

; extern "C" bool Mat4x4Inv_(Mat4x4 m_frv, Mat4x4 n, float epsilon, bool*=
is_singular);
;
; 描述：下面的函数计算一个 4x4 矩阵的逆矩阵
;
; 需要：          x86_64 和 AVX2 支持
;
; 注意：由下面的注释中，m2 = n * n，m3 = n * n * n，等等

; 栈中存放的临时变量的偏移量
OffsetM2Lo equ 0                                ;m2 rows 0 and 1
OffsetM2Hi equ 32                               ;m2 rows 2 and 3
OffsetM3Lo equ 64                               ;m3 rows 0 and 1
OffsetM3Hi equ 96                               ;m3 rows 2 and 3

Mat4x4Inv  proc frame
        _CreateFrame Minv_,16,160
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

; 执行必要的初始化和验证
```

```
        test rcx,01fh
        jnz Error                       ;若 m_inv 未对齐则跳转
        test rdx,01fh
        jnz Error                       ;若 m 未对齐则跳转
        vmovaps ymm14,[rdx]
        vmovaps ymm15,[rdx+32]          ;ymm15:ymm14 = m
        vmovss real4 ptr [rbp],xmm2     ;保存 epsilon 以备后续使用

; 为临时矩阵分配 128 字节的空间，以 32 字节边界对齐
        and rsp,0ffffffe0h              ;将 rsp 对齐到 32 字节边界
        sub rsp,128                     ;为临时矩阵分配空间

; 计算 m2 的值
        vmovaps ymm0,ymm14
        vmovaps ymm1,ymm15              ;ymm1:ymm0 = m
        vmovaps ymm2,ymm14
        vmovaps ymm3,ymm15              ;ymm3:ymm2 = m
        call Mat4x4Mul                  ;ymm1:ymm0 = m2
        vmovaps [rsp+OffsetM2Lo],ymm0
        vmovaps [rsp+OffsetM2Hi],ymm1   ;保存 m2 的值

; 计算 m3 的值
        vmovaps ymm2,ymm14
        vmovaps ymm3,ymm15              ;ymm3:ymm2 = m
        call Mat4x4Mul                  ;ymm1:ymm0 = m3
        vmovaps [rsp+OffsetM3Lo],ymm0
        vmovaps [rsp+OffsetM3Hi],ymm1   ;保存 m3 的值
        vmovaps ymm12,ymm0
        vmovaps ymm13,ymm1              ;ymm13:ymm12 = m3                    608

; 计算 m4 的值
        vmovaps ymm2,ymm14
        vmovaps ymm3,ymm15              ;ymm3:ymm2 = m
        call Mat4x4Mul                  ;ymm1:ymm0 = m4

; 计算并保存矩阵的迹
        _Mat4x4TraceYmm
        vmovss xmm10,xmm0,xmm0          ;xmm10 = t4

        vmovaps ymm0,ymm12
        vmovaps ymm1,ymm13
        _Mat4x4TraceYmm
        vmovss xmm9,xmm0,xmm0           ;xmm9 = t3

        vmovaps ymm0,[rsp+OffsetM2Lo]
        vmovaps ymm1,[rsp+OffsetM2Hi]
        _Mat4x4TraceYmm
        vmovss xmm8,xmm0,xmm0           ;xmm8 = t2

        vmovaps ymm0,ymm14
        vmovaps ymm1,ymm15
        _Mat4x4TraceYmm
        vmovss xmm7,xmm0,xmm0           ;xmm7 = t1

; 计算所需的系数
; c1 = -t1;
; c2 = -1.0f / 2.0f * (c1 * t1 + t2);
; c3 = -1.0f / 3.0f * (c2 * t1 + c1 * t2 + t3);
; c4 = -1.0f / 4.0f * (c3 * t1 + c2 * t2 + c1 * t3 + t4);
;
; 使用的寄存器：t1 - t4 = xmm7 - xmm10, c1 - c4 = xmm12 - xmm15
```

```
        vxorps xmm2,xmm7,real4 ptr [r4_SignBitMask]    ;xmm2 = r1

        vmulss xmm13,xmm12,xmm7         ;c1 * t1
        vaddss xmm14,xmm13,xmm8         ;c1 * t1 + t2
        vmulss xmm13,xmm15,[r4_NOp5]    ;xmm13 = c2

        vmulss xmm14,xmm13,xmm7         ;c2 * t1
        vmulss xmm0,xmm12,xmm8          ;c1 * t2
        vaddss xmm14,xmm14,xmm0         ;c2 * t1 + c1 * t2
        vaddss xmm14,xmm14,xmm9         ;c2 * t1 + c1 * t2 - t3
        vmulss xmm14,xmm14,[r4_NOp3333] ;xmm14 = c3

        vmulss xmm15,xmm14,xmm7         ;c3 * t1
        vmulss xmm0,xmm13,xmm8          ;c2 * t2
        vmulss xmm1,xmm12,xmm9          ;c1 * t3
609     vaddss xmm2,xmm0,xmm1           ;c2 * t2 + c1 * t3
        vaddss xmm15,xmm15,xmm2         ;c3 * t1 + c2 * t2 + c1 * t3
        vaddss xmm15,xmm15,xmm10        ;c3 * t1 + c2 * t2 + c1 * t3 + t4
        vmulss xmm15,xmm15,[r4_NOp25]   ;xmm15 = c4

; 确保矩阵不是奇异的
        vandps xmm1,xmm15,[r4_AbsMask] ;计算 fabs(c4) 的值
        vcomiss xmm1,real4 ptr [rbp]   ;结果与 epsilon 进行比较
        setp al                        ;若比较结果为无序则 al 置位
        setb ah                        ;若 fabs(c4) < epsilon 则 ah 置位
        or al,ah                       ;al 代表是否为奇异矩阵
        mov [r9],al                    ;保存是否为奇异矩阵的状态
        jnz Error                      ;若是奇异矩阵则跳转

; 计算 m_inv = -1.0 / c4 * (m3 + c1 * m2 + c2 * m1 + c3 * I)
        vmovaps ymm0,[rsp+OffsetM3Lo]
        vmovaps ymm1,[rsp+OffsetM3Hi]       ;ymm1:ymm0 = m3

        vbroadcastss ymm12,xmm12
        vmulps ymm2,ymm12,[rsp+OffsetM2Lo]
        vmulps ymm3,ymm12,[rsp+OffsetM2Hi]  ;ymm3:ymm2 = c1 * m2

        vbroadcastss ymm13,xmm13
        vmulps ymm4,ymm13,[rdx]
        vmulps ymm5,ymm13,[rdx+32]          ;ymm5:ymm4 = c2 * m

        vbroadcastss ymm14,xmm14
        vmulps ymm6,ymm14,[Mat4x4I]
        vmulps ymm7,ymm14,[Mat4x4I+32]      ;ymm7:ymm6 = c3 * I

        vaddps ymm0,ymm0,ymm2
        vaddps ymm1,ymm1,ymm3               ;ymm1:ymm0 = m3 + c1*m2
        vaddps ymm4,ymm4,ymm6
        vaddps ymm9,ymm5,ymm7               ;ymm9:ymm8 = c2*m + c3*I
        vaddps ymm0,ymm0,ymm8
        vaddps ymm1,ymm1,ymm9               ;ymm1:ymm0 = matrix sum

        vmovss xmm2,[r4_NOp0]
        vdivss xmm2,xmm2,xmm15              ;xmm2 = -1.0 / c4
        vbroadcastss ymm2,xmm2
        vmulps ymm0,ymm0,ymm2
        vmulps ymm1,ymm1,ymm2               ;ymm1:ymm0 = m_inv

        vmovaps [rcx],ymm0
        vmovaps [rcx+32],ymm1               ;保存 m_inv
610     mov rax,1                           ;设置成功返回码
```

```
Done:   vzeroupper
        RestoreXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,↩
xmm14,xmm15
        DeleteFrame
        ret

Error:  xor eax,eax
        jmp Done
Mat4x4Inv_ endp

; 下面的函数用于软件测试和调试
Mat4x4Trace_ proc
        Mat4x4TraceYmm
        ret
Mat4x4Trace_ endp
Mat4x4Mul_ proc
        call Mat4x4Mul
        ret
Mat4x4Mul_ endp
        end
```

矩阵的乘法逆矩阵定义如下：

设 A 和 X 为两个 $n \times n$ 的矩阵，若 $AX = XA = I$ 成立，则矩阵 X 是 A 的逆矩阵。其中 I 表示一个 $n \times n$ 的单位矩阵。图 20-4 给出了一个逆矩阵的例子。注意并不是所有的 $n \times n$ 矩阵都有逆矩阵，没有逆矩阵的矩阵叫作奇异矩阵。

$$A = \begin{bmatrix} 6 & 2 & 2 \\ 2 & -2 & 2 \\ 0 & 4 & 2 \end{bmatrix} \quad X = \begin{bmatrix} 0.1875 & -0.0625 & -0.125 \\ 0.0625 & 0.1875 & 0.125 \\ -0.125 & 0.375 & 0.25 \end{bmatrix} \quad AX = XA = I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

图 20-4　矩阵 A 及其乘法逆矩阵 X

矩阵的逆矩阵可以使用多种数学方法求出。示例程序 Avx64CalcMat4 x 4Inv 使用了一个基于凯莱 – 哈密顿定理的计算方法，其中使用了一些容易利用 SIMD 算术方法展开的矩阵运算。图 20-5 定义了一些计算 4 × 4 矩阵的逆矩阵所必需的公式，其中，矩阵的迹（trace）就是其主对角线元素的和。

611

$$A^1 = A,\ A^2 = AA,\ A^3 = AAA,\ A^4 = AAAA$$

$$t_n = \mathrm{trace}(A^n) \qquad \mathrm{trace}(A) = \sum_{i=1}^{n} a_{ii}$$

$$c_1 = -t_1$$

$$c_2 = -\frac{1}{2}\left(c_1 t_1 + t_2\right)$$

$$c_3 = -\frac{1}{3}\left(c_2 t_1 + c_1 t_2 + t_3\right)$$

$$c_4 = -\frac{1}{4}\left(c_3 t_1 + c_2 t_2 + c_1 t_3 + t_4\right)$$

$$A^{-1} = -\frac{1}{c_4}\left(A^3 + c_1 A^2 + c_2 A + c_3 I\right)$$

图 20-5　计算 4 × 4 矩阵的逆矩阵

清单 20-12 列出了示例程序 Avx64CalcMat4 × 4Inv 的 C++ 代码。在程序一开始有一个函数 Mat4 × 4InvCpp，它使用图 20-5 中的公式来计算一个 4 × 4 单精度浮点数矩阵的逆矩阵。在对 Mat4 × 4 型参数 m 和 m_inv 进行过对齐验证之后，函数 Mat4 × 4InvCpp 使用两个辅助函数

Mat4×4Mul 和 Max4×4Trace（这两个函数的代码没有列出，仅包含在可下载的软件包里）来计算 t1 ～ t4 的值。c1 ～ c4 的值则在后面的代码中使用简单的标量浮点算术运算得出。如果 c4 的值为 0，则矩阵 m 是奇异的，此时函数 Mat4×4InvCpp 会终止。否则，最终的逆矩阵会被算出并存放在 m_inv 中。

汇编语言文件 Avx64CalcMat4×4Inv .asm（清单 20-13）定义了一个名为 _Mat4×4TraceYmm 的宏，用来计算一个 4×4 单精度浮点数矩阵的迹。这个宏需要将它处理的源矩阵存放到寄存器 YMM0（第 0 行和第 1 行）和 YMM1（第 2 行和第 3 行）中。该宏使用 vblendps、vpermps 和
[612] vhaddps 指令来计算矩阵的迹，如图 20-6 所示。

$$M = \begin{bmatrix} 7 & 2 & 19 & 3 \\ 8 & 6 & 5 & 10 \\ 22 & 3 & 1 & 12 \\ 13 & 25 & 9 & 4 \end{bmatrix}$$

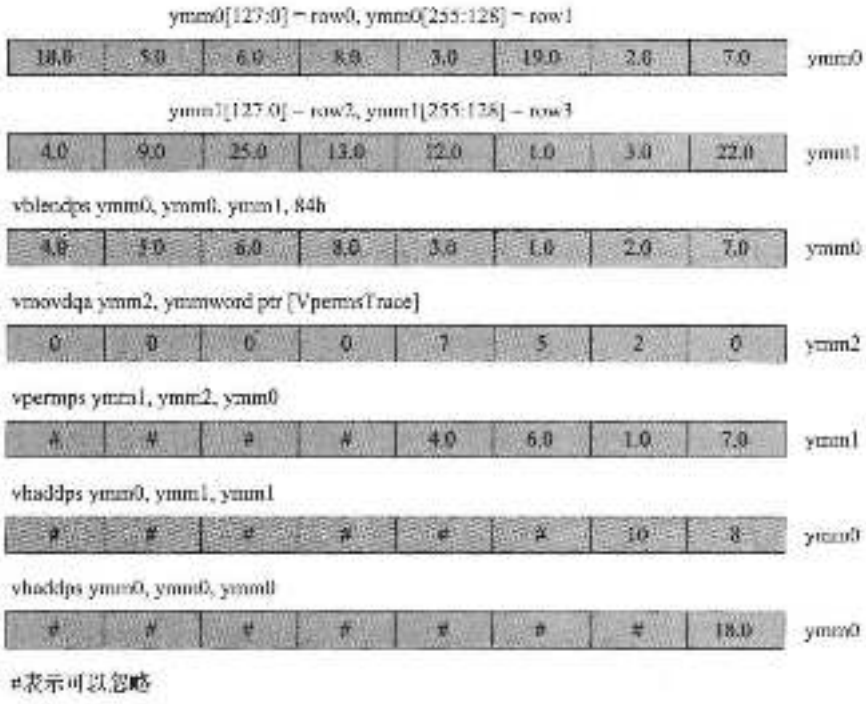


图 20-6 计算矩阵的迹

宏定义 _Max4×4TraceYmm 后是一个名为 Mat4×4Mul 的私有函数。这个函数计算两个 4×4 单精度浮点数矩阵的积。这里所使用的方法与第 10 章中使用的方法类似。首先，使用指令 vunpcklps、vunpckhps 和 vperms 计算矩阵 m2 的转置矩阵，如图 20-7 所示。然后，使用指令 vperm2f128（排列浮点值）将转置矩阵的每一行拷贝到 YMM 寄存器的低 128 位和高 128 位。每一行的复制将必须进行的点积计算数从 16 减少到 8。最后，使用一系列 vdpps 和 vorps 指令计算矩阵积。注意，每个 vdpps YMM 目标操作数未使用的元素被设置为 0.0，
[613] 以方便使用 vorps 指令计算最终的值。

函数 Max4×4Inv_ 使用与其对应的 C++ 代码同样的逻辑来计算逆矩阵。首先，矩阵的迹 t1 ～ t4 使用函数 Mat4×4Mul 和宏 _Mat4×4TraceYmm 来求出。接下来系数 c1 ～ c4 由

x86-AVX 标量浮点数算术运算求出。系数 c4 会被检测以确保矩阵不是奇异的。最后是计算所需的逆矩阵。注意，求逆矩阵所需的所有算术运算都直接使用组合乘法（vmulps）和加法（vaddps）。输出 20-6 显示了示例程序 Avx64Mat4 × 4Inv 的执行结果。表 20-3 给出了一些时间测量数据。

$$M=\begin{bmatrix}6 & 10 & 3 & 7\\ 5 & 2 & 12 & 11\\ 13 & 4 & 3 & 9\\ 8 & 5 & 1 & 2\end{bmatrix} \qquad M^{T}=\begin{bmatrix}6 & 5 & 13 & 5\\ 10 & 2 & 4 & 5\\ 3 & 12 & 3 & 1\\ 7 & 11 & 9 & 2\end{bmatrix}$$

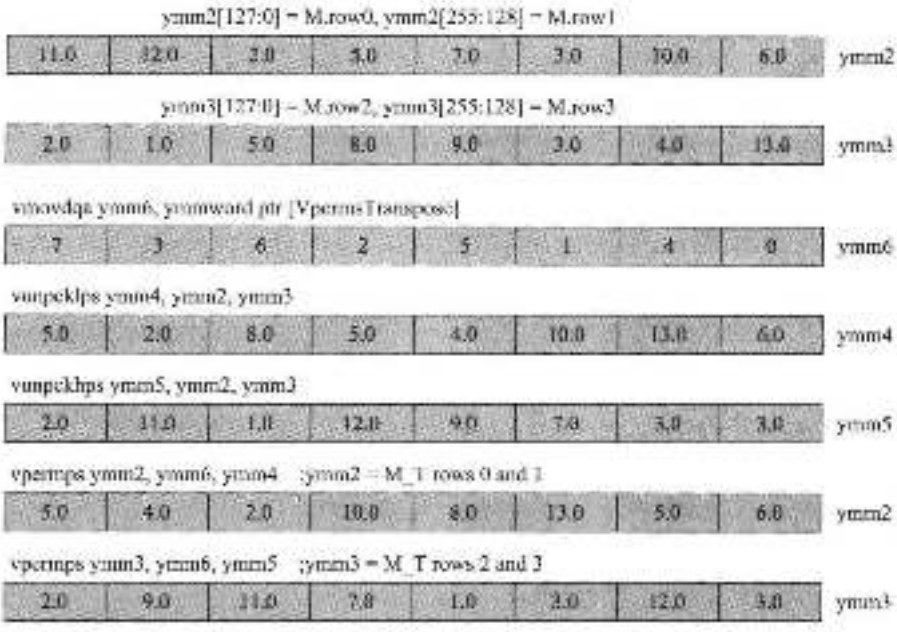


图 20-7 计算矩阵的转置矩阵 614

输出 20-6 示例程序 Avx64CalcMat4 × 4Inv

```
Results for Avx64CalcMat4x4Inv

Test Matrix #1
      2.000000      7.000000      3.000000      4.000000
      5.000000      9.000000      6.000000      4.750000
      6.500000      3.000000      4.000000     10.000000
      7.000000      5.250000      8.125000      6.000000

Calculating inverse matrix - Mat4x4InvCpp

Inverse matrix
     -0.943926      0.916570      0.197547     -0.425579
     -0.056882      0.251248      0.003028     -0.165952
      0.545399     -0.647646     -0.213597      0.505123
      0.412456     -0.412053      0.056124      0.124363

Inverse matrix verification
      1.000000     -0.000000      0.000000     -0.000000
      0.000000      1.000000      0.000000      0.000000
     -0.000000      0.000000      1.000000      0.000000
      0.000000      0.000000      0.000000      1.000000
```

```
Calculating inverse matrix - Mat4x4Inv_

Inverse matrix
     -0.943926       0.916570       0.197547      -0.425579
     -0.056882       0.251148       0.003028      -0.165957
      0.545399      -0.647656      -0.213597       0.305173
      0.412456      -0.412053       0.056125       0.124363

Inverse matrix verification
      1.000000      -0.000000       0.000000      -0.000000
      0.000000       1.000000       0.000000       0.000000
     -0.000000       0.000000       1.000000       0.000000
      0.000000       0.000000       0.000000       1.000000

Test Matrix #2
      0.500000      12.000000      17.250000       4.000000
      5.000000       2.000000       6.750000       8.000000
     13.125000       1.000000       3.000000       9.750000
     16.000000       1.625000       7.000000       0.250000

Calculating inverse matrix - Mat4x4InvCpp

Inverse matrix
      0.001652      -0.069024       0.054959       0.038935
      0.135369      -0.359846       0.242038      -0.090325
     -0.035010       0.239298      -0.183964       0.077221
     -0.005335       0.056194       0.060361      -0.066908

Inverse matrix verification
      1.000000       0.000000       0.000000       0.000000
     -0.000000       1.000000      -0.000000      -0.000000
      0.000000       0.000000       1.000001       0.000000
      0.000000       0.000000       0.000000       1.000001

Calculating inverse matrix - Mat4x4Inv

Inverse matrix
      0.001652      -0.069024       0.054959       0.038935
      0.135369      -0.359846       0.242038      -0.090325
     -0.035010       0.239298      -0.183964       0.077221
     -0.005335       0.056194       0.060361      -0.066908

Inverse matrix verification
      1.000001       0.000000       0.000000       0.000000
     -0.000000       1.000000      -0.000000      -0.000000
      0.000000       0.000000       1.000001       0.000000
      0.000000       0.000000       0.000000       1.000001

Test Matrix #3
      2.000000       0.000000       0.000000       1.000000
      0.000000       4.000000       5.000000       0.000000
      0.000000       0.000000       0.000000       7.000000
      0.000000       0.000000       0.000000       6.000000

Calculating inverse matrix - Mat4x4InvCpp
Matrix 'm' is singular

Calculating inverse matrix - Mat4x4Inv_
Matrix 'm' is singular

Benchmark times saved to file __Avx64CalcMat4x4InvTimed.csv
```

表 20-3 示例程序 Avx64CalcMat4x4Inv（10 000 次矩阵求逆操作）中的矩阵求逆函数的平均执行时间（单位：微秒）

CPU	C++	x86-AVX-64
Intel Core i7-4770	980	420
Intel Core i7-4600U	1194	491

20.2.4 其他指令

本章的最后一个示例程序名为 Avx64MiscInstructions，它演示了如何在一个 64 位汇编语言函数中使用选择收集（select gather）和半精度浮点指令。清单 20-14 和清单 20-15 分别给出了示例程序 Avx64MiscInstructions 的 C++ 和汇编语言源代码。

清单 20-14 Avx64MiscInstructions.cpp

```
#include "stdafx.h"
#include "MiscDefs.h"
#define _USE_MATH_DEFINES
#include <math.h>

extern "C" void Avx64GatherFloatIndx32_(float g[8], const float* x, Int32*
indices[8]);
extern "C" void Avx64GatherFloatIndx64_(float g[4], const float* x, Int64*
indices[4]);
extern "C" void Avx64FloatToHp_(Uint16 x_hp[8], float x1[8]);
extern "C" void Avx64HpToFloat_(float x[8], Uint16 x_hp[8]);

void Avx64GatherFloat(void)
{
    const int n = 20;
    float x1[n];

    printf("Results for Avx64GatherFloat()\n");
    printf("\nSource array\n");

    for (int i = 0; i < n; i++)
    {
        x1[i] = i * 100.0f;
        printf("x1[%02d]: %6.1f\n", i, x1[i]);
    }
    printf("\n");

    float g1_32[8], g1_64[4];
    Int32 g1_indices32[8] = {2, 3, 7, 1, 1, 12, 4, 17};
    Int64 g1_indices64[4] = {5, 0, 19, 13};

    Avx64GatherFloatIndx32_(g1_32, x1, g1_indices32);
    for (int i = 0; i < 8; i++)
        printf("g1_32[%02d] = %6.1f (gathered from x[%02d])\n", i, g1_32[i],*
g1_indices32[i]);

    printf("\n");

    Avx64GatherFloatIndx64_(g1_64, x1, g1_indices64);
    for (int i = 0; i < 4; i++)
        printf("g1_64[%02d] = %6.1f (gathered from x[%02lld])\n", i,*
g1_64[i], g1_indices64[i]);
}
```

617

```
void Avx64HalfPrecision(void)
{
    float x1[8], x2[8];
    Uint16 x_hp[8];

    x1[0] = 0.5f;           x1[1] = 1.0f / 512.0f;
    x1[2] = 1004.0625f;     x1[3] = 5003.125f;
    x1[4] = 42000.5f;       x1[5] = 75600.875f;
    x1[6] = -6002.125f;     x1[7] = (float)M_PI;

    Avx64FloatToHp_(x_hp, x1);
    Avx64HpToFloat_(x2, x_hp);

    printf("\rResults for Avx64HalfPrecision()\n");

    for (int i = 0; i < 8; i++)
        printf("%d %16.6f %16.6f\n", i, x1[i], x2[i]);
}

int _tmain(int argc, _TCHAR* argv[])
{
    Avx64GatherFloat();
    Avx64HalfPrecision();
    return 0;
}
```

清单 20-15 Avx64MiscInstructions_.asm

```
        include <MacrosX86-64.inc>

        .const
MaskVgatherdps  dword 80000000h,80000000h,80000000h,80000000h
                dword 80000000h,80000000h,80000000h,80000000h
MaskVgatherqps  dword 80000000h,80000000h,80000000h,80000000h
618]    .code
; extern "C" void Avx64GatherFloatIndx32_(float g[8], const float* x, Int32↩
indices[8]);
;
; 描述：下面的函数演示了 vgatherdps 指令的用法
;
; 需要：x86-64 和 AVX2 支持

Avx64GatherFloatIndx32_ proc
        vmovdqu ymm0,ymmword ptr [r8]
        vmovdqu ymm1,ymmword ptr [MaskVgatherdps]

        vgatherdps ymm2,[rdx+ymm0*4],ymm1   ;ymm2 用来保存收集的 SPFP 值

        vmovups ymmword ptr [rcx],ymm2      ;保存运算结果
        vzeroupper
        ret
Avx64GatherFloatIndx32_ endp

; extern "C" void Avx64GatherFloatIndx64_(float g[4], const float* x, Int64↩
indices[4]);
;
; 描述：下面的函数演示了 vgatherqps 指令的用法
;
; 需要：x86-64 和 AVX2 支持
```

```
Avx64GatherFloatIndx64_ proc
        vmovdqu ymm0,ymmword ptr [r8]
        vmovdqu xmm1,xmmword ptr [MaskVgatherqps]

        vgatherqps xmm2,[rdx+ymm0*4],xmm1   ;xmm2 用于保存收集的 SPFP 值

        vmovups xmmword ptr [rcx],xmm2      ;保存运算结果
        vzeroupper
        ret
Avx64GatherFloatIndx64_ endp

; extern "C" void Avx64FloatToHp_(Uint16 x_hp[8], float x1[8]);
;
; 描述：下面的函数将一个 8 个 SPFP 元素的数组转换成 HPFP 型的数组
;
; 需要：x86-64、AVX 和 F16C 支持

Avx64FloatToHp_ proc
        vmovups ymm0,ymmword ptr [rdx]
        vcvtps2ph xmmword ptr [rcx],ymm0,00000100b  ;使用就近舍入
        ret
Avx64FloatToHp_ endp

; extern "C" void Avx64HpToFloat_(float x[8], Uint16 x_hp[8]);
;
; 描述：下面的函数将一个 8 个 HPFP 元素的数组转换成 SPFP 型的数组
;
; 需要：x86-64、AVX 和 F16C 支持

Avx64HpToFloat_ proc
        vcvtph2ps ymm0,xmmword ptr [rdx]
        vmovups ymmword ptr [rcx],ymm0
        ret
Avx64HpToFloat_ endp
        end
```

C++ 文件 Avx64MiscInstructions.cpp（清单 20-14）包含了一个名为 Avx64GatherFloat 的函数。这个函数会初始化一些数据和索引数组，以供汇编语言函数 Avx64GatherFloatIndx32_ 和 Avx64GatherFloatIndx64_ 使用。Avx64MiscInstructions.cpp 中还包含一个名为 Avx64Half-Precision 的函数，这个函数会调用执行半精度浮点数转换的函数 Avx64FloatToHp_ 和 Avx64HpToFloat_。注意，由于 C++ 本身并不支持半精度浮点数据类型，所以，程序使用了一个 Uint16 型的数组来临时存放半精度浮点数。

清单 20-15 给出了示例程序 Avx64MiscInstructions 的汇编语言函数。函数 Avx64GatherFloatIndx32_ 和 Avx64GatherFloatIndx64 分别演示了 vgatherdps 和 vgatherqps 指令的使用。（图 12-4 显示了 vgatherdps 指令的执行。）注意第一个指令使用了 32 位索引而第二个指令使用了 64 位索引。vgatherqps 指令使用 64 位索引意味着它只能收集 4 个而不是 8 个单精度浮点数。

汇编语言文件 Avx64MiscInstructions_.asm 还包括两个用于半精度转换的函数 Avx64FloatToHp_ 和 Avx64HpToFloat_。这两个函数使用 vcvtps2ph（将单精度浮点数转换成 16 位浮点数）和 vcvtph2ps（将 16 位浮点数转换成单精度浮点数）两个转换指令来执行单精度浮点数和半精度浮点数之间的相互转换。注意 vcvtps2ph 指令包含了一个立即数，用来指定转换过程中使用的舍入方法。如表 20-4 给出了 vcvtps2ph 指令的舍入选项。

表 20-4 vcvtps2ph 指令的舍入选项

操作数位	数值	描述
1:0	00	就近舍入
	01	向下舍入
	10	向上舍入
	11	截后
2	0	使用 1:0 位进行舍入
	1	使用 MXCSR.RC 位进行舍入
7:3		未使用

输出 20-7 给出了示例程序 Avx64MiscInstructions 的执行结果。从中可以看出，在单精度浮点数向半精度浮点数转换时出现了大量的舍入。另外，数值 75600.875 被转换成了无穷大，因为它比半精度浮点数据类型的最大值还要大（函数 printf 会将无穷大显示为文本 1.#INF00）。正如我们在 12 章所讨论的，半精度浮点数转换指令主要是用于降低存储空间。

输出 20-7 示例程序 Avx64MiscInstructions

```
Results for Avx64GatherFloat()

Source array
x1[00]:    0.0
x1[01]:  100.0
x1[02]:  200.0
x1[03]:  300.0
x1[04]:  400.0
x1[05]:  500.0
x1[06]:  600.0
x1[07]:  700.0
x1[08]:  800.0
x1[09]:  900.0
x1[10]: 1000.0
x1[11]: 1100.0
x1[12]: 1200.0
x1[13]: 1300.0
x1[14]: 1400.0
x1[15]: 1500.0
x1[16]: 1600.0
x1[17]: 1700.0
x1[18]: 1800.0
x1[19]: 1900.0

g1_32[00] =  200.0 (gathered from x[02])
g1_32[01] =  300.0 (gathered from x[03])
g1_32[02] =  700.0 (gathered from x[07])
g1_32[03] =  100.0 (gathered from x[01])
g1_32[04] =  100.0 (gathered from x[01])
g1_32[05] = 1200.0 (gathered from x[12])
g1_32[06] =  400.0 (gathered from x[04])
g1_32[07] = 1700.0 (gathered from x[17])

g1_64[00] =  500.0 (gathered from x[05])
g1_64[01] =    0.0 (gathered from x[00])
g1_64[02] = 1900.0 (gathered from x[19])
g1_64[03] = 1300.0 (gathered from x[13])

Results for Avx64HalfPrecision()
```

```
0          0.500000          0.500000
1          0.001953          0.001953
2       1004.062500       1004.000000
3       5004.125000       5004.000000
4      42000.500000      42016.000000
5      75600.875000          1.#INF00
6      -6002.125000      -6004.000000
7          3.141593          3.140625
```

20.3 总结

在本章中，我们学习了如何在 x86-64 执行环境中使用 x86-SSE 和 x86-AVX 的计算资源。我们还讨论了采用不同数据结构对 SIMD 算法效率的影响。在接下来的两章中，我们将学习更多的编程技巧，这些技巧可用来优化 x86 汇编语言函数，提高它的执行效率。 622

第 21 章 |

Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX

高级主题和优化技巧

为最大化汇编代码的性能，我们需要理解 x86 处理器内部工作的几个关键细节。本章中，我们将学习现代 x86 多核处理器的内部架构及其依赖的微架构。要提升汇编语言软件性能，还需要恰当地使用一些编程策略和技巧，本章也会论述这些内容。

要详细论述 x86 微架构和汇编语言的优化技术至少需要几个章节，甚至是整整一本书。因此，本章内容只是一个入门型的指南。关于本章内容的最重要参考资料是《Intel 64 和 IA-32 架构优化参考手册》(Intel 64 and IA-32 Architectures Optimization Reference Manual)。如需获得 x86 微架构和汇编语言优化技巧的更多信息和内容，建议查阅这个重要的文献。

注意　可以从下面的 Intel 网站下载《Intel 64 和 IA-32 架构优化参考手册》和其他重要的 x86 软件开发人员手册：http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html。

21.1　处理器微架构

x86 处理器的性能主要是由它内部的微架构决定的。一个处理器微架构的特性是由其内部硬件组件的构造和运行方式决定的。这些组件包括指令流水线、解码器、调度器、执行单元、数据总线和缓存。了解处理器微架构基本概念的开发人员，通常具有建设性的洞察力，
623 能够开发出更高效的代码。

AMD 和 Intel 公司总是定期地宣传基于加强的或者新微架构的处理器。在本章后续的讨论中，我们将描述 Intel Haswell 微架构的构造。该微架构用于第四代 Core i7、i5 和 i3 系列处理器。Intel 之前的微架构和 Huswell 相似，包括 Nehalem、Sandy Bridge 和 Ivy Bridge（换言之，第一代、第二代和第三代 Core i7、i5、i3 系列处理器），但 Haswell 在性能和减少功耗上有显著的增强。

21.1.1　多核处理器概述

要探讨 Haswell 或任何现代微架构的架构细节，最好使用多核处理器框架。图 21-1 显示了一个基于 Haswell 的四核处理器的简化框图。请注意每个 CPU 核心包含一级（L1）指令缓存和数据缓存。它们被标注为 I-Cache 和 D-Cache。顾名思义，这些内存缓存包含了 CPU 核心可以快速访问的指令和数据。每个 CPU 核心还包含了二级（L2）统一缓存，用于同时保存指令和数据。L1 和 L2 缓存使得 CPU 核心在无须访问更高级的 L3 共享缓存或主内
624 存的情况下，并发地执行独立的运算。

如果 CPU 核心请求的指令或者数据不在它的 L1 或者 L2 缓存，那它就必须从 L3 缓存或主存载入。L3 缓存被分成了许多切片。每一个切片由一个逻辑控制器和一个数据阵列组成。逻辑控制器管理其相应的数据阵列的访问。它也负责处理缓存未命中（cache miss）和主存写入（当请求的数据不在缓存中并且必须从主存中载入时，缓存未命中就发生了）。数据阵列包括实际的缓存数据，被组织为 64 字节宽的数据包，称为缓存行（Cache Line）。环形互联（Ring

Interconnect）是一个高速的内部总线，用于CPU多个核心、L3缓存、图形单元和系统代理（System Agent）之间传递数据。系统代理负责处理器、外部数据总线和主存之间的数据交换。

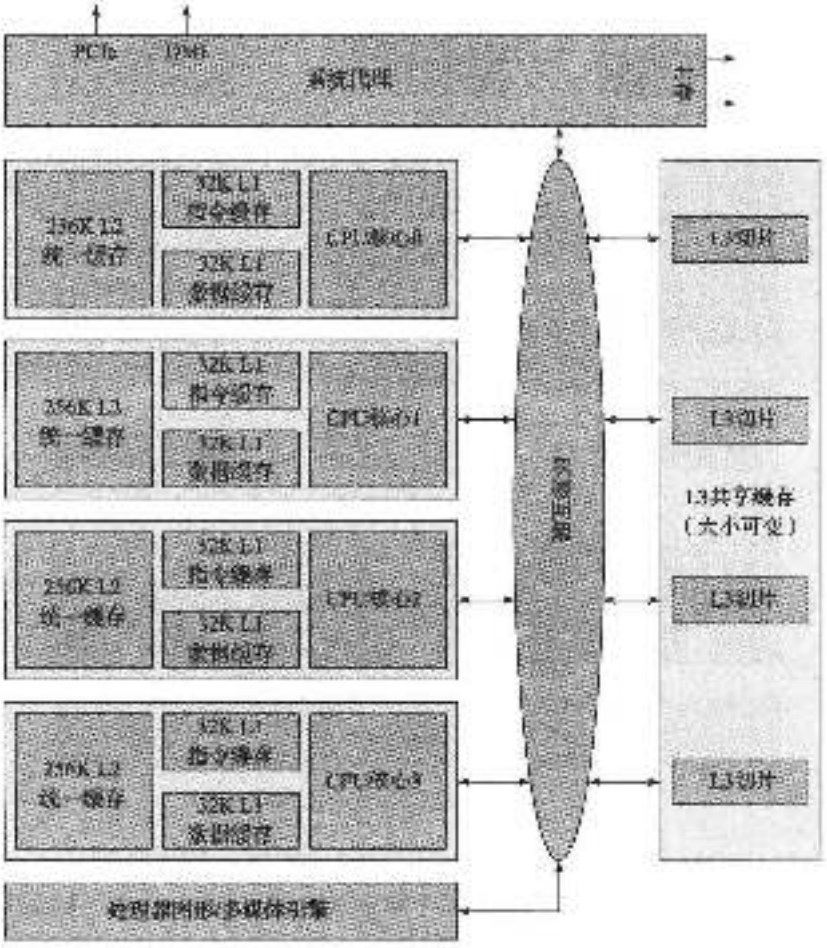


图21-1 基于Haswell四核处理器的简化框图 625

21.1.2 微架构流水线功能

程序执行过程中，一个CPU核心执行五个基本的指令操作：取指（fetch）、解码、分发、执行、老化（retire）。这些操作的细节是由CPU微架构流水线的功能决定的。图21-2显示了基于Haswell处理器的CPU流水线功能的流程框图。

取指令和预解码单元从L1指令缓存抢取指令，并且开始为执行这些指令做准备。这一阶段的步骤包括：指令长度解析，x86指令前缀解码，为协助后续的解码器做性质标记。取指令和预解码单元还负责持续地为指令队列提供指令流（指令队列排队指令，以提供给指令解码器）。

指令解码器将x86指令翻译成微操作（micro-ops）。微操作是一种独立的低级指令，最终会被执行引擎中的一个执行单元所执行（下一节讨论执行单元）。解码器为一条x86指令所产生的微操作的数目，随这条指令的复杂度而变化。简单的寄存器到寄存器指令，例如add eax, edx 和 pxor xmm0, xmm0 被解码为一个单一的微操作。完成更复杂操作的指令，例

如 idiv rcx 和 vdivpd ymm0, ymm1, ymm2，需要多条微操作。将 x86 指令翻译成微操作，带来了一些架构上和性能上的好处，包括指令级的并行和乱序执行。

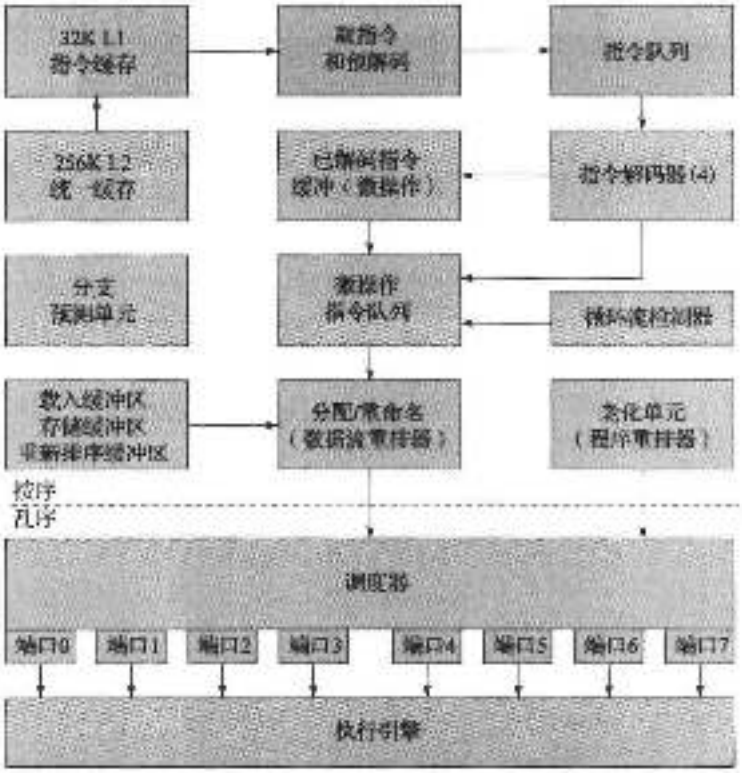


图 21-2 Haswell CPU 核心流水线功能

指令解码器还完成了两个辅助操作，能提高流水线带宽的利用率。第一个被称为微融合（micro-fusion），即将来自同一x86指令的多个简单微操作合并为一个单一的复杂微操作。微融合指令的例子包括内存存储（mov [ebx+16], eax）和使用内存操作数的计算指令（sub r9, qword ptr [rbp+48]）。对于融合后的复杂微操作，执行引擎会分发多次（每次分发执行一个来自原始指令的简单微操作）。指令解码器的第二个辅助操作被称为宏融合。宏融合将某些常用的x86指令对结合成一个单一的微操作。宏融合指令对的例子包括很多（但不是所有）紧跟 add、and、cmp、dec、inc、sub 或 test 指令之后的条件转移指令。

来自指令解码器的微操作被转移到微操作指令队列，并最终被调度器分发。必要时，它们也会被缓存到已解码指令缓存。微操作指令队列也被用于循环流检测器（识别和锁定微操作指令队列中小的程序循环）。这样可以提高性能，因为小的循环能重复执行，而无须额外的取指令、解码和微操作缓存读取。

分配和重命名块承担着桥梁角色，把按序的前端流水线和乱序的调度器及执行引擎连接起来。它给微操作按需分配内部缓冲区。它还会消除微操作之间的假依赖，以便于乱序执行。（当两个微操作需要同时访问相同硬件资源的不同版本时，就会导致一个假依赖。）而后微操作会被转移到调度器。该单元将微操作入队，直到所有需要的源操作数都可用。然后它将准备好的微操作分发到适当执行引擎的执行单元。老化单元按照程序原始指令顺序，移除已经执行完成的微操作。它还会激发在微指令执行中可能发生的处理器异常。

626 / 627

最后，分支预测单元在当前代码执行模式的基础上，预测最有可能执行的分支目标，帮助选择下一组要执行的指令。分支目标只是一个转移控制指令的目标操作数，例如 jcc、jmp、call 或者 ret。分支预测单元使得 CPU 内核在分支被决定之前，能够投机地执行一条指令的微操作。必要时，CPU 内核会已解码指令缓存、L1 指令缓存、L2 统一缓存、L3 缓存和主存的顺序来搜索要执行的指令。

21.1.3 执行引擎

执行引擎执行调度器传给它的微操作。图 21-3 显示了 Haswell CPU 内核执行引擎的概要框图。每个分发端口下面的矩形方框，表示了该类作的执行单元。请注意，调度器端口中有四个是支持计算功能的执行单元，包括整数、浮点数和 SIMD 算术运算。剩余的四个端口支持内存加载和存储操作。

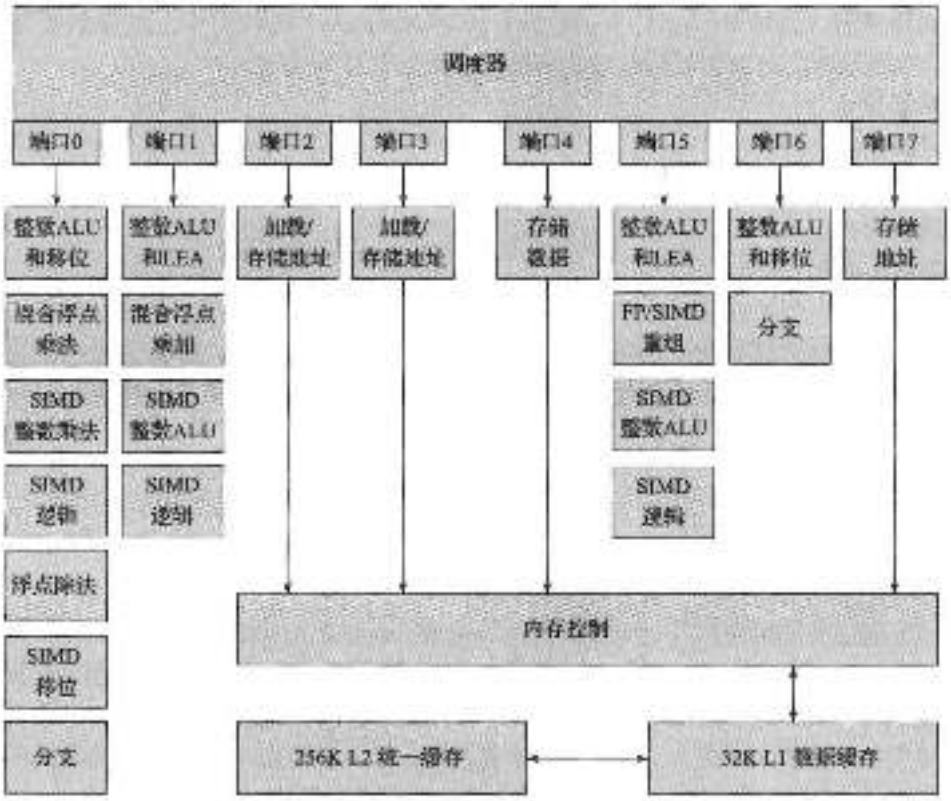


图 21-3 Hanswell CPU 内核执行引擎及其执行单元

628

每个执行单元完成一个特定的计算或者操作。例如，“整数 ALU 和移位”执行单元实现了整数算术和移位操作。“SIMD 整数 ALU”执行单元被设计为完成 SIMD 整数算术运算。请注意，执行引擎包含某些执行单元的多个实例。这使得执行引擎可以并发地执行多个特定的微操作。例如，执行引擎可以使用“SIMD 逻辑”执行单元并发地执行三个独立的 SIMD 逻辑操作。

Haswell 调度器可以在每个周期给执行引擎分发最多八个微操作（每个端口一个）。乱序

引擎，包括调度器、执行引擎和老化单元，最多可以支持192个同时运行（in-flight）的微操作。表21-1归纳了Intel最近一些年的微架构的关键缓冲区大小。

表21-1 微架构关键缓冲区大小的比较

参数	Nehalem	Sandy Bridge	Haswell
分发端口	6	6	8
同时运行的微操作	128	168	192
同时运行的加载操作	48	64	72
同时运行的存储操作	32	36	42
调度器实体（Entries）	36	54	60

21.2 优化汇编语言代码

本节讨论一些优化x86汇编语言代码的简单编程技巧。建议把这些技巧应用在运行于Intel最新微架构（包括Haswell、Sandy Bridge和Nehalem）的代码中。大多数技巧同样适用于更早的微架构。可以把优化技巧和辅助性的指导方针分为五大类：

- 基本优化
- 浮点算术
- 程序分支
- 数据对齐
- SIMD技巧

需要谨记的一个要点是，接下来的所有优化技巧都必须谨慎使用。例如，如果仅为避免使用不推荐的指令形式一次，就增加多条额外的push和pop指令，那么是没有意义的。此外，本章介绍的所有优化技巧，都无法补救一个不适当的或者设计很差的算法。《Intel 64和IA-32架构优化参考手册》包含本节讨论的优化技巧的更详细内容。

[629]

21.2.1 基本优化

下面列出了一些常用于提高x86汇编语言代码性能的基本优化技巧：

- 尽可能使用test指令，而不是cmp指令。
- 尽可能避免使用内存与立即数形式的cmp和test指令（例如，cmp dword ptr [ebp+16], 100或者test byte ptr [r12], 0fh）。最好先将内存值载入寄存器，然后使用寄存器与立即数形式的cmp和test指令（例如，mov eax, dword ptr [ebp-16]，接着cmp eax, 100）。
- 使用add或者sub指令，而不是inc或者dec指令，特别是在性能关键的循环中。后面的两个指令不会更新EFLAGS寄存器中的所有标志位，通常会慢一些。
- 使用xor、sub、pxor、xorps等指令将一个寄存器置0，而不是用数据传送指令。例如，xor eax, eax和xorps xmm0, xmm0比mov eax, 0和movaps xmm0, xmmword ptr [XmmZero]要好。
- 在有类似数宽度前缀的指令中，避免使用16位立即数，而应该使用对应的32位或者8位立即数。例如，使用mov edx, 42而不是mov dx, 42。
- 展开（或者部分展开）循环次数是常数的小循环。
- 将在计算中多次使用的内存值载入寄存器。如果一个内存值只在一次计算中用到，用寄存器到内存形式的计算指令。表21-2显示了几个例子。

表 21-2 一次使用和多次使用内存值的指令形式

寄存器到内存形式（一次使用的数据）	载入和寄存器到寄存器形式（多次使用的数据）
add edx,dword ptr [x]	mov eax,dword ptr [x]
	add edx,eax
and rax,qword ptr [rbx+16]	mov rcx,[rbx+16]
	and rax,rcx
cmp ecx,dword ptr [n]	mov eax,dword ptr [n]
	cmp ecx,eax
mulpd xmm0,xmmword ptr [rdx]	movapd xmm1,xmmword ptr [rdx]
	mulpd xmm0,xmm1

630

下列优化技巧适用于 x86-64 代码：

- 当操作数是 32 位时，使用 32 位通用寄存器和指令形式。
- 操作 32 位宽数值时，优先使用通用寄存器 EAX、EBX、ECX、EDX、ESI 和 EDI，而不是寄存器 R8D-R15D。对于后面的寄存器组，指令解码要多一个字节。
- 利用额外的通用寄存器和 SIMD 寄存器，以减少数据依赖和寄存器溢出（当程序必须暂时地存储一个寄存器的值到内存中，为其他计算腾出这个寄存器的时候，寄存器溢出就发生了）。
- 如果不需要完整的 128 位结果，用二操作数或三操作数形式的 imul 指令进行两个 64 位整数乘法。

21.2.2 浮点算术

使用汇编语言进行浮点算术运算时，应考虑下面的指导方针：

- 在新代码中使用 x86-SSE 或者 x86-AVX 而不是 x87 FPU 的标量浮点指令。
- 在算术计算中，尽可能避免算术下溢和非正规值。
- 避免使用非正规浮点常量。
- 如果预知会有多次算术下溢，考虑启用清洗到零（MXCSR.FZ）和非正规为零（MXCSR.DAZ）模式。对于如何正确地使用这些模式，第 7 章中有更多的信息。

21.2.3 程序分支

程序分支指令如 jmp、call 和 ret 在执行时是潜在的耗时操作，因为它们可能影响前端流水线和内部缓存的内容。考虑到使用的频率，条件跳转指令 jcc 也可能带来性能问题。下面的优化技巧能最小化分支指令对性能的影响，并且提高分支预测单元的准确性：

- 组织代码，尽量少使用分支指令。
- 使用 setcc 和 cmovcc 指令，以消除不可预测的数据相关的分支。 631
- 在性能关键的循环中，对齐分支目标的边界到 16 字节。
- 将不太可能执行的条件代码（例如错误处理代码）移到另外的程序段或内存页。

当预测一个分支语句的目标时，分支预测单元采用静态和动态技术。当包含条件跳转指令的代码能够组织成与分支预测单元的静态预测算法一致时，那么错误的分支预测就可以被最小化：

- 当贯穿（fall-through）代码可能被执行时，使用向前条件跳转。
- 当贯穿代码不可能被执行时，使用向后条件跳转。

向前条件跳转方法经常用在检查函数参数的代码块中。向后条件跳转技术可以用在程序循环代码块的底部（紧跟着一个计数器更新或者其他循环结束判断）。清单 21-1 包含了一小段汇编语言函数，展示了这些实践经验的细节。

清单 21-1 使用符合静态分支预测算法的条件跳转指令

```
        .model flat,c
        .code

; extern "C" bool CalcResult (double* des, const double* src, int n);

CalcResult_ proc
        push ebp
        mov ebp,esp
        push esi
        push edi

; 本段代码使用向前条件跳转，因为错误的情况更可能发生
        mov edi,[ebp+8]                 ;edi = des
        test edi,0fh
        jnz Error                       ;如果 des 没有对齐则跳转
        mov esi,[ebp+12]                ;esi =src
        test esi,0fh
        jnz Error                       ;如果 src 没有对齐则跳转
        mov ecx,[ebp+16]                ;ecx = n
        cmp ecx,2
        jl Error                        ;如果 n<2 则跳转
        test ecx,1
        jnz Error                       ;如果 n&1!=0 则跳转

; 简单的数组处理循环
        xor eax,eax
@@:     movapd xmm0,xmmword ptr [esi+eax]
        mulpd xmm0,xmm0
        movapd xmmword ptr [edi+eax],xmm0

; 本段代码使用向后条件跳转，因为错误的情况更不可能发生
        add eax,16
        sub ecx,2
        jnz @@

        mov eax,1
        pop edi
        pop esi
        pop ebp
        ret

; 错误处理代码，不太可能执行
Error:  xor eax,eax
        pop edi
        pop esi
        pop ebp
        ret
CalcResult_ endp
        end
```

21.2.4 数据对齐

本书已经多次提及正确的数据对齐的重要性，这个问题怎么强调也不过分。操作对齐错

误的数据时，可能导致处理器花费额外的内存周期和执行更多微指令，这将会给整个系统的性能带来负面影响。下面的数据对齐实践应该被认为是普遍真理并一直遵守：

- 将多字节整数和浮点数对齐到自然的边界。
- 将 64、128 和 256 位宽的组合数据对齐到它们本身的边界。
- 必要时填补数据结构，以保证正确对齐。
- 使用恰当的编译器指令和库函数，以对齐高层代码分配的数据项。例如，__declspec(align(n)) 指示符和 _aligned_malloc 函数能用来正确地对齐 Visual C++ 函数中分配的数据项。
- 更多地使用存储对齐，而不是加载对齐。 633

下面这些对齐技巧也推荐使用：

- 将小数组和短字符串对齐安置在数据结构中，以避免缓存行分割。
- 评估不同的数据布局对性能的影响，例如数组结构与结构数组。

21.2.5 SIMD 技巧

在任何函数中，用 x86-SSE 和 x86-AVX 计算资源时应该考察下面这些技巧是否适用：

- 消除寄存器依赖，以利用执行引擎的多个执行单元。
- 用组合的单精度浮点数代替双精度浮点数。
- 将多次使用的内存操作数和组合常量加载到寄存器。
- 用对齐的传送指令存储和加载组合数据，例如 movdqa、movaps、movapd 等。
- 用小的数据块处理 SIMD 数组，以最大化重用驻留缓存数据。
- 在 x86-AVX 代码中，使用数据混合而不是数据重组。
- 当需要避免 x86-AVX 到 x86-SSE 状态迁移的损失时，使用 vzeroupper 指令。
- 使用 x86-AVX vgather 指令的双字形式，而不是四字形式。在数据要用到之前就完成需要的收集操作。

下面这些实践可以用于提高特定算法的性能（进行 SIMD 编码和解码操作）：

- 使用无时态存储指令（例如 movntdqa、movntpd、movntps 等），以最小化缓存污染。
- 使用数据预取指令（例如 prefetcht0、prefetchnta 等），以通知处理器预期要使用的数据项。

第 22 章包含了使用无时态（non-temporal）存储和数据预取指令的示例代码。 634

21.3 总结

本章中，我们考察了现代 x86 处理器的内部设施，包括多核构造和内部体系架构，还学习了一些有用的技巧用于提高 x86 汇编语言代码的性能。在本书的最后一章，我们将通过一些示例代码来实践本章介绍的高级内容。 635 ↓ 636

高级主题编程

本章将通过两个示例程序来阐释 x86 汇编语言编程的一些高级技巧。第一个示例程序解释如何使用无时态（non-temporal）内存存储来加速 SIMD 处理算法。第二个示例程序介绍如何用软件数据预取加速链表遍历。两个示例程序都包含了 x86-32 和 x86-64 两种实现，以比较两个执行环境的性能。

22.1 无时态内存存储

从内存缓存的角度，数据可以分为有时态数据和无时态数据。有时态数据是在短时间内被多次访问的任意值。有时态数据的例子包括在程序循环中被多次引用的数组或者数据结构，还包括程序的代码。无时态数据是被访问一次后不会立即再次使用的数据。许多 SIMD 处理算法的目标数组常常包含了无时态数据。

如果缓存中包含过多的无时态数据，则处理器性能会下降，这通常被称为缓存污染（cache pollution）。理想状态下，一个处理器的内存缓存只包含时态数据，因为缓存只被访问一次的数据项是没有意义的。x86-SSE 指令集包括了几个无时态内存存储指令，程序可以用它们来最小化缓存污染。

本节的示例程序 NonTemporalStore 说明了无时态内存存储指令 movntps 的使用（用无时态提示存储组合单精度浮点数），还比较了该指令与标准的 movaps 指令的性能。清单 22-1、
637 清单 22-2 和清单 22-3 包含了示例程序 NonTemporalStore 的 C++ 和汇编语言代码。

清单 22-1 NonTemporalStore.cpp

```
#include "stdafx.h"
#include "NonTemporalStore.h"
#include <math.h>
#include <malloc.h>
#include <stdlib.h>
#include <stddef.h>

bool CalcResultCpp(float* c, const float* a, const float* b, int n)
{
    if ((n <= 0) || ((n & 0x3) != 0))
        return false;

    if (((uintptr_t)a & 0xf) != 0)
        return false;
    if (((uintptr_t)b & 0xf) != 0)
        return false;
    if (((uintptr_t)c & 0xf) != 0)
        return false;

    for (int i = 0; i < n; i++)
        c[i] = sqrt(a[i] * a[i] + b[i] * b[i]);

    return true;
}

bool CompareResults(const float* c1, const float* c2a, const float*c2b,↩
```

```
int n, bool pf)
{
    const float epsilon = 1.0e-9f;
    bool compare_ok = true;

    for (int i = 0; i < n; i++)
    {
        if (pf)
            printf("%2d - %10.4f %10.4f %10.4f\n", i, c1[i], c2a[i],↵
c2b[i]);

        bool b1 = fabs(c1[i] - c2a[i]) > epsilon;
        bool b2 = fabs(c1[i] - c2b[i]) > epsilon;

        if (b1 || b2)
        {
            compare_ok = false;                                          638
            if (pf)
                printf("Compare error at index %2d: %f %f %f\n", i, c1[i],↵
c2a[i], c2b[i]);
        }
    }

    return compare_ok;
}

void NonTemporalStore()
{
    const int n = 16;
    const int align = 16;
    float* a = (float*)_aligned_malloc(n * sizeof(float), align);
    float* b = (float*)_aligned_malloc(n * sizeof(float), align);
    float* c1 = (float*)_aligned_malloc(n * sizeof(float), align);
    float* c2a = (float*)_aligned_malloc(n * sizeof(float), align);
    float* c2b = (float*)_aligned_malloc(n * sizeof(float), align);

    srand(67);
    for (int i = 0; i < n; i++)
    {
        a[i] = (float)(rand() & 100);
        b[i] = (float)(rand() & 100);
    }

    CalcResultCpp(c1, a, b, n);
    CalcResultA_(c2a, a, b, n);
    CalcResultB_(c2b, a, b, n);

#ifdef _WIN64
    const char* platform = "Win64";
#else
    const char* platform = "Win32";
#endif

    printf("Results for NonTemporalStore (platform = %s)\n", platform);
    bool rc = CompareResults(r1, c2a, c2b, n, true);

    if (rc)
        printf("Array compare OK\n");
    else
        printf("Array compare FAILED\n");

    _aligned_free(a);
```

```
    _aligned_free(b);
639 _aligned_free(c1);
    _aligned_free(c2a);
    _aligned_free(c2b);
}

int _tmain(int argc, _TCHAR* argv[])
{
    NonTemporalStore();
    NonTemporalStoreTimed();
    return 0;
}
```

清单 22-2 NonTemporalStore32_.asm

```
IFDEF ASMX86_32
        .model flat,c
        .code

; _CalcResult32 Macro
;
; 下面的宏包含了一个简单的计算循环，用来比较指令 movaps 和 movntps 的性能差异

_CalcResult32 macro MovInstr
        push ebp
        mov ebp,esp
        push ebx
        push edi

; 加载并检验参数
        mov edi,[ebp+8]                 ;edi = c
        test edi,0fh
        jnz Error                       ;如果 c 没有对齐则跳转
        mov ebx,[ebp+12]                ;ebx = a
        test ebx,0fh
        jnz Error                       ;如果 a 没有对齐则跳转
        mov edx,[ebp+16]                ;edx = b
        test edx,0fh
        jnz Error                       ;如果 b 没有对齐则跳转

        mov ecx,[ebp+20]                ;ecx = n
        test ecx,ecx
        jle Error                       ;如果 n<=0 则跳转
        test ecx,3
        jnz Error                       ;如果 n%4!=0 则跳转

; 计算 c[i] = sqrt(a[i] * a[i] + b[i] * b[i])
        xor eax,eax                     ;eax=数组偏移
640     align 16

@@:     movaps xmm0,xmmword ptr [ebx+eax]   ;xmm0 = a[] 的值
        movaps xmm1,xmmword ptr [edx+eax]   ;xmm1 = b[] 的值
        mulps xmm0,xmm0                 ;xmm0 = a[i] * a[i]
        mulps xmm1,xmm1                 ;xmm1 = b[i] * b[i]
        addps xmm0,xmm1                 ;xmm0 = 和
        sqrtps xmm0,xmm0                ;xmm0 = 最终结果
        MovInstr xmmword ptr [edi+eax],xmm0 ;将最终结果保存到 c

        add eax,16                      ;更新偏移
        sub ecx,4                       ;更新计数值
        jnz @@
```

```
        mov eax,1                       ;设置成功返回码
        pop edi
        pop ebx
        pop ebp
        ret

Error:  xor eax,eax                     ;设置失败返回码
        pop ebx
        pop ebp
        ret
        endp

;extern bool CalcResultA_(float* c, const float* a, const float* b, int n)
CalcResultA_ proc
        _CalcResult32 movaps
CalcResultA_ endp

;extern bool CalcResultB_(float* c, const float* a, const float* b, int n)
CalcResultB_ proc
        _CalcResult32 movntps
CalcResultB_ endp
ENDIF
        end
```

清单 22-3 NonTemporalStore64_.asm

```
IFDEF ASMX86_64
        .code

; _CalcResult64 Macro
;
; 下面宏包含了一个简单的计算循环，用来比较指令 movaps 和 movntps 的性能差异                    [64]

_CalcResult64 macro MovInstr

; 加载并检验参数
        test rcx,0fh
        jnz Error                       ;如果 c 没有对齐则跳转
        test rdx,0fh
        jnz Error                       ;如果 a 没有对齐则跳转
        test r8,0fh
        jnz Error                       ;如果 b 没有对齐则跳转

        test r9d,r9d
        jle Error                       ;如果 n <= 0 则跳转
        test r9d,3
        jnz Error                       ;如果 n % 4 != 0 则跳转

; 计算 c[i] = sqrt(a[i] * a[i] + b[i] * b[i])
        xor eax,eax                     ;eax = 数组偏移
        align 16
@@:     movaps xmm0,xmmword ptr [rdx+rax]   ;xmm0 = a[] 的值
        movaps xmm1,xmmword ptr [r8+rax]    ;xmm1 = b[] 的值
        mulps xmm0,xmm0                 ;xmm0 = a[i] * a[i]
        mulps xmm1,xmm1                 ;xmm1 = b[i] * b[i]
        addps xmm0,xmm1                 ;xmm0 = 和
        sqrtps xmm0,xmm0                ;xmm0 = 最终结果
        MovInstr xmmword ptr [rcx+rax],xmm0 ;将最终结果保存到 c

        add rax,16                      ;更新偏移
        sub r9d,4                       ;更新计数值
        jnz @@
```

```
        mov eax,1                       ;设置成功返回码
        ret
  函数并检验参数
Error:  xor eax,eax                     ;设置失败返回码
        ret
        endm

;extern bool CalcResultA_(float* c, const float* a, const float* b, int n)
CalcResultA_ proc
        _CalcResult64 movaps
CalcResultA_ endp

;extern bool CalcResultB_(float* c, const float* a, const float* b, int n)
CalcResultB_ proc
        _CalcResult64 movntps
CalcResultB_ endp
ENDIF
642     end
```

文件 NonTemporalStore.cpp（清单 22-1）的顶部是一个名为 CalcResultCpp 的函数。这个函数对两个单精度浮点数组（a 和 b）的元素计算一个简单的算术值，然后将结果写入目标数组 c。示例中的汇编语言函数计算相同的结果。之后的函数 CompareResults 用来确认 C++ 和汇编语言输出的数组是相等的。函数 NonTemporalStore 分配和初始化测试数组，然后调用上述的 CalcResultCpp 函数。接着调用相应的汇编语言函数 CalcResultA_ 和 CalcResultB_（本节后面会描述这两个函数）。最后比较这三个计算函数输出的数组是否存在差异。

示例程序 NonTemporalStore 包含了计算函数 CalcResultA_ 和 CalcResultB_ 的 x86-32 和 x86-64 实现。清单 22-2 列出了 x86-32 版本的汇编语言源代码。在文件 NonTemporalStore32_.asm 靠近底部的位置是函数 CalcResultA_ 和 CalcResultB_ 的定义。这些函数使用了一个名为 _CalcResult32 的宏，该宏生成了计算代码。请注意每次调用 _CalcResult32 宏时，移动指令使用了不同的参数值。

文件 NonTemporalStore32_.asm 的顶部定义了宏 _CalcResult32。参数验证之后，是一段使用 x86-SSE 组合单精度浮点运算的指令，计算 c[i] = sqrt(a[i] * a[i] + b[i] * b[i])。语句 MovInstr xmmword ptr[edi+eax], xmm0 使用 movaps 或者 movntps 指令（取决于宏参数 MovInstr 的值），将最终结果保存到目标数组 c。这意味着函数 CalcResultA_ 和 CalcResultB_ 执行的代码是完全相同的，除了将结果存入目标数组的指令。

清单 22-3 列出了汇编语言文件 NonTemporalStore64_.asm 的源代码。在组织和逻辑上，CalcResultA_ 和 CalcResultB_ 的 x86-64 实现与其相对应的 x86-32 实现相似。宏 _CalcResult64 也使用了与宏 _CalcResult32 相同的 x86-SSE 运算指令。

请注意除 end 编译指令之外，文件 NonTemporalStore32_.asm 和 NonTemporalStore64_.asm 中所有的语句都在编译指令 IFDEF 之内。在 Visual C++ 每个执行平台的属性页，MASM 预处理符号 ASMx86_32 和 ASMx86_64 会被正确定义。这使得示例程序 NonTemporalStore 在 Visual C++ 中的项目可以同时支持 Win32 和 Win64。附录 A（可以从 http://www.apress.com/9781484200650 下载）包含了如何为 Visual C++ 项目配置多执行目标的更多信息。

输出 22-1 显示了 Win32 版本 NonTemporalStore 的输出。Win64 版本的输出与之完全相同，除了平台和指标文件的名字。（译者注：此处为 c[i] = sqrt(a[i] * a[i] + b[i] * b[i]) 的计算结果，所以不论是 32 位还是 64 位，不论使用哪个传送指令，结果都是相同的。）表 22-1 和表 22-2 显示了两个执行环境下的时间度量，包含了一些有趣的结果。在基于 Haswell 的 i7-4770 和 i7-4600U 处理器上，示例程序 NonTemporalStore 使用 movntps 指令时比相应的 movaps 指令快很多。在

基于 Sandy Bridge 的 i3-2310M 上，执行时间是相同的（记住，movntps 指令只是向处理器提供暗示，并不确保性能提升）。x86-32 和 x86-64 版本的函数 CalcResultCpp 在执行时间上有相当大 643
的不同，也令人好奇。这些数字背后的原因是，Visual C++ 编译器为 64 位版本生成了使用 x86-SSE SIMD 浮点运算的代码，而为 32 位版本生成了使用 x86-SSE 标量浮点运算的代码。

输出 22-1 示例程序 NonTemporalStore

```
Results for NonTemporalStore (platform - Win32)
 0 -    87.2066    87.2066    87.2066
 1 -    51.4781    51.4781    51.4781
 2 -    44.1022    44.1022    44.1022
 3 -   112.4144   112.4144   112.4144
 4 -    16.5529    16.5529    16.5529
 5 -    53.1507    53.1507    53.1507
 6 -    96.1769    96.1769    96.1769
 7 -   125.8196   125.8196   125.8196
 8 -    91.5478    91.5478    91.5478
 9 -    85.8021    85.8021    85.8021
10 -    64.6003    64.6003    64.6003
11 -    76.0066    76.0066    76.0066
12 -    67.1863    67.1863    67.1863
13 -    91.2853    91.2853    91.2853
14 -    96.3172    96.3172    96.3172
15 -    27.0185    27.0185    27.0185
Array compare OK

Benchmark times saved to file __NonTemporalStore32.csv
```

表 22-1 x86-32 函数 CalcResultCpp、CalcResultA_ 和 CalcResultB_ 在 n = 1 000 000 时平均执行时间（单位：微秒）

CPU	CalcResultCpp	CalcResultA_(movaps)	CalcResultB_(movntps)
Intel Core i7-4770	1864	572	468
Intel Core i7-4600U	2377	812	595
Intel Core i3-2310M	5145	1707	1702

644

表 22-2 x86-64 函数 CalcResultCpp、CalcResultA_ 和 CalcResultB_ 在 n = 1 000 000 时平均执行时间（单位：微秒）

CPU	CalcResultCpp	CalcResultA_(movaps)	CalcResultB_(movntps)
Intel Core i7-4770	585	572	468
Intel Core i7-4600U	776	768	583
Intel Core i3-2310M	1714	1707	1702

22.2 数据预取

应用程序也可以使用 prefetch（预取数据到缓存）指令来提高某些算法的性能。该指令将预期要使用的数据预装载到处理器的缓存层级。有两种基本的 prefetch 指令形式。第一种形式（prefetcht0）预装载时态数据到处理器缓存的每个层级。第二种形式（prefetchnta）预装载非时态数据到 L2 缓存，并且可以使缓存污染最小化。需要注意的是，这两种形式的 prefetch 指令只是暗示处理器程序预期要用的数据。处理器可以选择执行数据预取操作，也可以忽略该暗示。

预取指令适用于多种数据结构，包括大数组和链表。链表是一个按序组织的节点集合。每个节点包括数据段和一个或多个指针（或链接）指向它的相邻节点。图 22-1 列举了一个简单的链表。因

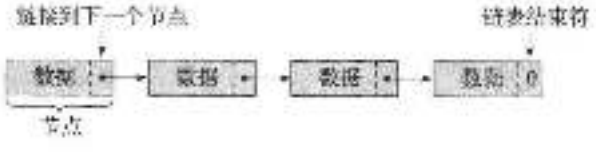


图 22-1 简单链表

为链表的大小可以按数据存储的需求增长和缩小（例如节点可以添加和删除）。链表的一个缺点是节点通常不存储在连续分配的内存块中，这导致遍历链表的时候，可能花较多访问时间。

下一个示例程序 LinkedListPrefetch 包含了遍历链表的 x86-32 和 x86-64 函数，有使用和没有使用 prefetchnta 指令两种情况。清单 22-4 和清单 22-5 显示了示例程序 Linked-
645 ListPrefetch 的 C++ 和汇编语言头文件。相应的源代码显示在清单 22-6 到清单 22-8 中。

清单 22-4 LinkedListPrefetch.h

```
#pragma once
#include "MiscDefs.h"

// 该结构必须和 LinkedListPrefetch.inc 中对应的结构定义匹配
typedef struct LlNode
{
    double ValA[4];
    double ValB[4];
    double ValC[4];
    double ValD[4];
    Uint8 FreeSpace[376];

    LlNode* Link;

#ifndef _WIN64
    Uint8 Pad[4];
#endif

} LlNode;

extern void LlTraverseCpp(LlNode* p);
extern LlNode* LlCreate(int num_nodes);
extern bool LlCompare(int num_nodes, LlNode* l1, LlNode* l2, LlNode* l3,↩
int* node_fail);

extern "C" void LlTraverseA_(LlNode* p);
extern "C" void LlTraverseB_(LlNode* p);

extern void LinkedListPrefetchTimed(void);
```

清单 22-5 LinkedListPrefetch.inc

```
; 该结构必须和 LinkedListPrefetch.h 中对应的结构定义匹配

LlNode  struct
ValA        real8 4 dup(?)
ValB        real8 4 dup(?)
ValC        real8 4 dup(?)
ValD        real8 4 dup(?)
FreeSpace   byte 376 dup(?)

IFDEF ASMX86_32
Link        dword ?
Pad         byte 4 dup(?)
646 ENDIF
IFDEF ASMX86_64
Link        qword ?
ENDIF

LlNode  ends
```

清单 22-6 LinkedListPrefetch.cpp

```
#include "stdafx.h"
#include "LinkedListPrefetch.h"
#include <stdlib.h>
#include <math.h>
#include <stddef.h>

bool LlCompare(int num_nodes, LlNode* l1, LlNode* l2, LlNode* l3, int*↩
node_fail)
{
    const double epsilon = 1.0e-9;

    for (int i = 0; i < num_nodes; i++)
    {
        *node_fail = i;

        if ((l1 == NULL) || (l2 == NULL) || (l3 == NULL))
            return false;

        for (int j = 0; j < 4; j++)
        {
            bool b12_c = fabs(l1->ValC[j] - l2->ValC[j]) > epsilon;
            bool b13_c = fabs(l1->ValC[j] - l3->ValC[j]) > epsilon;
            if (b12_c || b13_c)
                return false;

            bool b12_d = fabs(l1->ValD[j] - l2->ValD[j]) > epsilon;
            bool b13_d = fabs(l1->ValD[j] - l3->ValD[j]) > epsilon;
            if (b12_d || b13_d)
                return false;
        }

        l1 = l1->Link;
        l2 = l2->Link;
        l3 = l3->Link;
    }

    *node_fail = -2;
    if ((l1 != NULL) || (l2 != NULL) || (l3 != NULL))
        return false;                                          647
    *node_fail = -1;
    return true;
}

void LlPrint(LlNode* p, FILE* fp, const char* msg)
{
    int i = 0;
    const char* fs = "%14.6lf %14.6lf %14.6lf %14.6lf\n";

    if (msg != NULL)
        fprintf(fp, "%s\n", msg);

    while (p != NULL)
    {
        fprintf(fp, "\nLlNode %d [0x%p]\n", i, p);
        fprintf(fp, "  ValA: ");
        fprintf(fp, fs, p->ValA[0], p->ValA[1], p->ValA[2], p->ValA[3]);

        fprintf(fp, "  ValB: ");
        fprintf(fp, fs, p->ValB[0], p->ValB[1], p->ValB[2], p->ValB[3]);

        fprintf(fp, "  ValC: ");
```

```
            fprintf(fp, fs, p->ValC[0], p->ValC[1], p->ValC[2], p->ValC[3]);

            fprintf(fp, "  ValD: ");
            fprintf(fp, fs, p->ValD[0], p->ValD[1], p->ValD[2], p->ValD[3]);

            i++;
            p = p->Link;
        }
    }

    LlNode* LlCreate(int num_nodes)
    {
        LlNode* first = NULL;
        LlNode* last = NULL;

        srand(85);
        for (int i = 0; i < num_nodes; i++)
        {
            LlNode* p = (LlNode*)_aligned_malloc(sizeof(LlNode), 64);
648         p->Link = NULL;
            if (i == 0)
                first = last = p;
            else
            {
                last->Link = p;
                last = p;
            }

            for (int i = 0; i < 4; i++)
            {
                p->ValA[i] = rand() % 500 + 1;
                p->ValB[i] = rand() % 500 + 1;
                p->ValC[i] = 0;
                p->ValD[i] = 0;
            }
        }

        return first;
    }

    void LlTraverseCpp(LlNode* p)
    {
        while (p != NULL)
        {
            for (int i = 0; i < 4; i++)
            {
                p->ValC[i] = sqrt(p->ValA[i] * p->ValA[i] + p->ValB[i] *=
    p->ValB[i]);
                p->ValD[i] = sqrt(p->ValA[i] / p->ValB[i] + p->ValB[i] /=
    p->ValA[i]);
            }
            p = p->Link;
        }
    }

    void LinkedListPrefetch(void)
    {
        const int num_nodes = 8;
        LlNode* list1 = LlCreate(num_nodes);
        LlNode* list2a = LlCreate(num_nodes);
        LlNode* list2b = LlCreate(num_nodes);
```

```
#ifdef _WIN64
    const char* platform = "X86-64";
    size_t sizeof_ll_node = sizeof(LLNode);
    const char* fn = "__LinkedListPrefetchResults64.txt";

#else
    const char* platform = "X86-32";
    size_t sizeof_ll_node = sizeof(LLNode);
    const char* fn = "__LinkedListPrefetchResults32.txt";
#endif

    printf("\nResults for LinkedListPrefetch\n");
    printf("Platform target:  %s\n", platform);
    printf("sizeof(LLNode):   %d\n", sizeof_ll_node);
    printf("LLNode member offsets\n");
    printf("  Val4:           %d\n", offsetof(LLNode, Val4));
    printf("  Val8:           %d\n", offsetof(LLNode, Val8));
    printf("  ValC:           %d\n", offsetof(LLNode, ValC));
    printf("  Val0:           %d\n", offsetof(LLNode, Val0));
    printf("  FreeSpace:      %d\n", offsetof(LLNode, FreeSpace));
    printf("  Link:           %d\n", offsetof(LLNode, Link));
    printf("\n");

    LLTraverseCop(list1);
    LLTraverseA_(list2a);
    LLTraverseB_(list2b);

    int node_fail;

    if (!LLCompare(num_nodes, list1, list2a, list2b, &node_fail))
        printf("\nLinked list compare FAILED - node_fail = %d\n",↩
node_fail);
    else
        printf("\nLinked list compare OK\n");

    FILE* fp;
    if (fopen_s(&fp, fn, "wt") == 0)
    {
        LLPrint(list1, fp, "\n----- list1 -----");
        LLPrint(list2a, fp, "\n ----- list2a -----");
        LLPrint(list2b, fp, "\n ----- list2b -----");
        fclose(fp);

        printf("\nLinked list results saved to file %s\n", fn);
    }
}

int _tmain(int argc, _TCHAR* argv[])
{
    LinkedListPrefetch();
    LinkedListPrefetchUnwd();
    return 0;
}
```

清单 22-7 LinkedListPrefetch32_.asm

```
IFDEF ASMX86_32
        include <LinkedListPrefetch.inc>
        .model flat,c
        .code

; Macro _LLTraverse()
```

```
;
; 下面的求和成绩表遍历的代码，如果 UsePrefetch 等于 'Y'，则使用 prefetchnta 指令

_LITraverse32 macro UsePrefetch
        mov eax,[esp+4]                         ;eax 指向第一个节点
        test eax,eax
        jz Done                                 ;到达链表尾部则跳转

        align 16
@@:     mov ecx,[eax+LINode.Link]               ;ecx 指向下一个节点
        vmovapd ymm0,ymmword ptr [eax+LINode.ValA]  ;ymm0 = ValA
        vmovapd ymm1,ymmword ptr [eax+LINode.ValB]  ;ymm1 = ValB

IFIDNI <UsePrefetch>,<Y>
        mov edx,ecx
        test edx,edx                    ;还有其他节点吗?
        cmovz edx,eax                   ;避免预取到 NULL
        prefetchnta [edx]               ;预取下一个节点
ENDIF

; 计算 ValC[i] = sqrt(ValA[i] * ValA[i] + ValB[i] * ValB[i])
        vmulpd ymm2,ymm0,ymm0                   ;ymm2 = ValA * ValA
        vmulpd ymm3,ymm1,ymm1                   ;ymm3 = ValB * ValB
        vaddpd ymm4,ymm2,ymm3                   ;ymm4 = 和
        vsqrtpd ymm5,ymm4                       ;ymm5 = 平方根
        vmovntpd ymmword ptr [eax+LINode.ValC],ymm5 ;保存结果

; 计算 ValD[i] = sqrt(ValA[i] / ValB[i] + ValB[i] / ValA[i]);
        vdivpd ymm2,ymm0,ymm1                   ;ymm2 = ValA / ValB
        vdivpd ymm3,ymm1,ymm0                   ;ymm3 = ValB / ValA
        vaddpd ymm4,ymm2,ymm3                   ;ymm4 = 和
        vsqrtpd ymm5,ymm4                       ;ymm5 = 平方根
        vmovntpd ymmword ptr [eax+LINode.ValD],ymm5 ;保存结果

        mov eax,ecx                     ;eax 指向下一个节点
        test eax,eax
        jnz @B
631     vzeroupper

Done:   ret
        endm

; extern "C" void LITraverseA_(LINode* first);
LITraverseA_ proc
        _LITraverse32 n
LITraverseA_ endp

; extern "C" void LITraverseB_(LINode* first);
LITraverseB_ proc
        _LITraverse32 y
LITraverseB_ endp

ENDIF
        end
```

清单 22-8 LinkedListPrefetch64_.asm

```
IFDEF ASMX86_64
        include <LinkedListPrefetch.Inc>
        .code

; Macro _LITraverse64
```

```
;
; 下面的宏生成链表遍历的代码。如果 UsePrefetch 等于 'Y'，则使用 prefetchnta 指令

_LLTraverse64 macro UsePrefetch
        mov rax,rcx                                 ;rax 指向第一个节点
        test rax,rax
        jz Done                                     ;到达链表尾部则跳转

        align 16
@@:     mov rcx,[rax+LLNode.Link]                   ;rcx 指向下一个节点
        vmovapd ymm0,ymmword ptr [rax+LLNode.ValA]  ;ymm0 = ValA
        vmovapd ymm1,ymmword ptr [rax+LLNode.ValB]  ;ymm1 = ValB

IFIDNI <UsePrefetch>,<Y>
        mov rdx,rcx
        test rdx,rdx                        ;还有其他节点吗?
        cmovz rdx,rax                       ;避免线程到 NULL
        prefetchnta [rdx]                   ;指向下一个节点
ENDIF

; 计算 ValC[i] = sqrt(ValA[i] * ValA[i] + ValB[i] * ValB[i])
        vmulpd ymm2,ymm0,ymm0                       ;ymm2 = ValA * ValA
        vmulpd ymm3,ymm1,ymm1                       ;ymm3 = ValB * ValB
        vaddpd ymm4,ymm2,ymm3                       ;ymm4 = 和                    652
        vsqrtpd ymm5,ymm4                           ;ymm5 = 平方根
        vmovntpd ymmword ptr [rax+LLNode.ValC],ymm5 ;保存结果

; 计算 ValD[i] = sqrt(ValA[i] / ValB[i] + ValB[i] / ValA[i]);
        vdivpd ymm2,ymm0,ymm1                       ;ymm2 = ValA / ValB
        vdivpd ymm3,ymm1,ymm0                       ;ymm3 = ValB / ValA
        vaddpd ymm4,ymm2,ymm3                       ;ymm4 = 和
        vsqrtpd ymm5,ymm4                           ;ymm5 = 平方根
        vmovntpd ymmword ptr [rax+LLNode.ValD],ymm5 ;保存结果

        mov rax,rcx                         ;rax 指向下一个节点
        test rax,rax
        jnz @@
        vzeroupper

Done:   ret
        endm

; extern "C" void LLTraverseA_(LLNode* first);
LLTraverseA_ proc
        _LLTraverse64 n
LLTraverseA_ endp

; extern "C" void LLTraverseB_(LLNode* first);
LLTraverseB_ proc
        _LLTraverse64 y
LLTraverseB_ endp

ENDIF
        end
```

头文件 LinkedListPrefetch.h（清单 22-4）包含了 C++ 结构 LLNode 的声明。示例程序 LinkedListPrefetch 用这个结构构造链表的测试数据。结构成员 ValA 到 ValD 保存了链表遍历函数要操作的数据。移位预取最适用于大的数据结构，为验证目的，加入了成员 FreeSpace 来增加 LLNode 的大小。真实实现的 LLNode 可以将该空间用于更多的数据项。LLNode 的最后

一个成员是一个名为 Link 的指针，指向链表中的下一个 LLNode 结构。请注意 Win32 版本的 LLNode 包含了额外四个字节的名为 Pad 的成员，是为了使得 32 位和 64 位执行环境中结构的大小是相等的。（译者注：在 64 位执行环境中，指针成员 Link 是 64 位的，而在 32 位执行环境中，指针成员 Link 是 32 位的。）清单 22-5 显示了汇编语言中 LLNode 的相应声明。

文件 LinkedListPrefetch.cpp（清单 22-6）的头部是一个名为 LLCompare 的辅助函数，用
653 来比较示例程序操作的链表数据是否相等。接下来是另一个名为 LLPrint 的辅助函数，用来将链表的数据成员打印到指定的 FILE 流。函数 LLCreate 构造了一个链表，包含 num_nodes 个 LLNode 的实例。请注意每个 LLNode 被分配在 64 字节的边界，以避免数组数据被分割到不同的缓存行。函数 LLTraverse 包含了遍历指定链表的代码，并且使用链表中每个 LLNode 的数据数组执行需要的计算。最后，函数 LinkedListPrefetch 包含了构造测试链表的代码。然后该函数调用 C++ 和汇编语言的遍历函数 LLTraverseCpp、LLTraverseA_ 和 LLTraverseB_。

汇编语言文件 LinkedListPrefetch32.asm（清单 22-7）和 LinkedListPrefetch64.asm（清单 22-8）分别包含了链表遍历函数 LLTraverseA_ 和 LLTraverseB_ 的 32 位和 64 位实现。定义在相应文件的尾部，使用名为 LLTraverse32 或 LLTraverse64 的宏来生成需要的代码。这两个宏都需要一个参数来指定遍历代码是否使用 prefetchnta 指令。从逻辑和结构上来说，宏 _LLTraverse32 和 _LLTraverse64 是相同的，除了指针大小之外。后面的讨论将集中在宏 LLTraverse32 上。

在链表遍历循环的顶部，当前节点的数据数组 ValA 和 ValB 被分别加载到寄存器 YMM0 和 YMM1。指向下一个节点的指针也被加载到寄存器 ECX（并且有条件地加载到寄存器 EDX）。如果宏参数 UsePrefetch 和字符 Y 相等，prefetchnta [edx] 指令就会生效。该指令将包括数据数组 ValA 和 ValB 的下一个节点的开始字节无时态预取到 L2 级存。在执行 prefetchnta [edx] 指令之前，测试了 EDX，以避免使用指向 NULL 内存地址的指针执行预取操作，这将降低处理器的性能。另外需要注意的是，一个程序永远不要试图在其他程序的内存地址空间执行预取指令。

如果在 CPU 内核继续执行指令的同时，处理器在后台能执行请求的内存操作，预取指令工作得最好。_LLTraverse32 的计算部分进行了一些不相干的组合双精度浮点运算来模拟耗时操作。计算结果用 vmovntpd 保存到目的数组 ValC 和 ValD，因为这些数据只被引用一次。

输出 22-2 显示了示例程序 LinkedListPrefetch 的结果。表 22-3 和表 22-4 显示了 Win-32 和 Win64 版本的时间度量。在基于 Haswell 的处理器特别是 i7-4770 上，示例程序 LinkedListPrefetch 在使用 prefetchnta 指令时获得了更好的性能。需要注意的是，预取指令带来的任何性能提升，都高度依赖于数据的使用模式和基本的微架构。依照《Intel 64 和 IA-32 架构优化参考手册》（Intel 64 and IA-32 Architectures Optimization Reference Manual），数据预取指令是“实现相关的”。换言之，为了最大化预取性能，每个算法都必须“为每个实现”，
654 或为微架构单独优化。前述的参考手册包含了使用数据预取指令更多的信息。

输出 22-2 示例程序 LinkedListPrefetch

```
Results for LinkedListPrefetch
Platform target:  X86-32
sizeof(LLNode):   512
LLNode member offsets
  ValA:           0
  ValB:           32
  ValC:           64
  ValD:           96
```

```
FreeSpace:      128
Link:           504

Linked list compare OK

Linked list results saved to file __LinkedListPrefetchResults32.txt

Benchmark times saved to file __LinkedListPrefetch32.csv
```

表 22-3 x86-32 函数 LITraverseCpp、LITraverseA_ 和 LITraverseB_ (num_nodes=20 000) 的平均执行时间（单位：微秒）

CPU	LITraverseCpp	LITraverseA_	LITraverseB_(prefetchnta)
Intel Core i7-4770	1912	867	799
Intel Core i7-4600U	1911	969	955
Intel Core i3-2310M	3601	1676	1669

表 22-4 x86-64 函数 LITraverseCpp、LITraverseA_ 和 LITraverseB_ (num_nodes=20 000) 的平均执行时间（单位：微秒）

CPU	LITraverseCpp	LITraverseA_	LITraverseB_(prefetchnta)
Intel Core i7-4770	1660	843	793
Intel Core i7-4600U	1645	902	879
Intel Core i3-2310M	3391	1676	1669

655

22.3 总结

在本章中，我们学习了无时态（non-temporal）内存存储的概念，并学习了它的应用。我们还学习了 x86 数据预取指令的用法。本章的例子只是 x86 汇编语言高级编程的入门读物，鼓励读者查阅附录 C（从 http://www.apress.com/9781484200650 下载）中列出的参考文献，以获取 x86 汇编语言高级编程的更多信息。

656

索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

A

acos function（acos 函数），374
andn instruction（andn 指令），347
Arrays（数组）
  one-dimensional arrays（一维数组）
    CalcArraySquares program（CalcArraySquares 程序），58，60
    CalcArraySum program（CalcArraySum 程序），57
    source code implementation（源码实现），55
  two-dimensional arrays（二维数组）
    CalcMatrixRowColSums，62，65
    row-major ordering（行优先排序），61
AVX，327-328
AVX2，328
AVX-512，328

B

bextr instruction（bextr 指令），347
Blend instructions（Blend 指令），337
blsi instruction（blsi 指令），347
blsmsk instruction（blsmsk 指令），347
blsr instruction（blsr 指令），347
Bonnell（Bonnell 微架构），3
Branch Prediction Unit（分支预测单元）
  backward conditional jump（向后条件跳转），633
  forward conditional jump（向前条件跳转），632
Broadcast instructions（广播指令），336
Bulldozer microarchitecture（Bulldozer 微架构），3
bzhi instruction（bzhi 指令），347

C

CalcLeastSquares program（CalcLeastSquares 程序）
  code implementation（编码实现），124
  fadd instructions（fadd 指令），129
  formula（公式），128
  slope-intercept denominator（斜率和截距的公共分母），130
  stack register（栈寄存器），129
  statistical technique（统计技巧），128
CalcMeanStdev program（CalcMeanStdev 程序），112，116
  code implementation（编码实现），112
  ECX registers（ECX 寄存器），115
  EDX registers（EDX 寄存器），115
  fldz instruction（fldz 指令），115
  formula（公式），115
  FPU registers（FPU 寄存器），115
CalcMinMax program（CalcMinMax 程序）
  code implementation（编码实现），116
  stack register（栈寄存器），119
  validation check（有效性验证），118
CalcSphereAreaVolume program（CalcSphereAreaVolume 程序）
  code implementation（编码实现），108
  flag status（状态标志），110
  fmulp instruction（fmulp 指令），111
  formula（公式），109
  f(u)comi(p) instructions（f(u)comi(p) 指令），110
  stack register（栈寄存器），111
ConvertCoordinates program（ConvertCoordinates 程序）
  code implementation（编码实现），120
  formula（公式），122
  polar coordinate（极坐标），122
  PolarToRect_，123
  two-dimensional plane（二维平面），122
Core microarchitecture（核心微架构），2

Correlation coefficient（相关系数），394
Correlation coefficient program（相关系数程序）
AvxPackedFloatingPointCorrCoef .asm，391
AvxPackedFloatingPointCorrCoef.cpp，389，394
program output（程序输出），396
Cpuid instruction（Cpuid 指令）
AvxCpuid_.asm，444
AvxCpuid.cpp，439
cvtps2pd instruction（cvtps2pd 指令），335

D

Data blend（数据混合），453
AvxBlend_.asm，455
AvxBlend.cpp，453，456
program output（程序输出），457
vblendvps instruction（vblendvps 指令），457
vpblendvb instruction（vpblendvb 指令），457
vpblendw and vpblendd instructions（vpblendw 和 vpblendd 指令），457
Data broadcast（数据广播），447
AvxBroadcast .asm，449，452
AvxBroadcast.cpp，448，452
program output（程序输出），452
Data gather（数据收集），463
AvxGather_.asm，467-469
AvxGather.cpp，464，468
Data-manipulation instructions（数据操作指令），447
data blend（数据混合）
AvxBlend .asm，455
AvxBlend.cpp，453，456
program output（程序输出），457
vpblendvb instruction（vpblendvb 指令），457
vpblendw and vpblendd instructions（vpblendw 和 vpblendd 指令），457
data broadcast（数据广播），447
AvxBroadcast .asm，449，452
AvxBroadcast.cpp，452
program output（程序输出），452
data gather（数据收集），463
AvxGather_.asm，467-469
AvxGather.cpp，464，468
data permute（数据排列）
AvxPermute .asm，460，462
AvxPermute.cpp，458，462
program output（程序输出），463
Data permute（数据排列）
AvxPermute .asm，460，462
AvxPermute.cpp，458，462
program output（程序输出），463
Data prefetch（数据预取），645
Data types，x86-32（x86-32 数据类型），4
BCD values（BCD 值），8
bit fields（位域），7
bit strings（二进制位串），8
fundamental（基本），4
numerical（数值），5
packed（组合），6
strings（字符串），7

E

End of string（EOS）character（字符串终止符），303
Extract and insert instructions（提取和插入指令），338

F

Feature extension instructions（功能扩展相关指令）
fused-multiply-add group（融合乘加组），343
VFMADD subgroup（VFMADD 子组），344
VFMADDSUB subgroup（VFMADDSUB 子组），345
VFMSUBADD subgroup（VFMSUBADD 子组），345
VFMSUB subgroup（VFMSUB 子组），344
VFNMADD subgroup（VFNMADD 子组），346
VFNMSUB subgroup（VFNMSUB 子组），346
general-purpose register group（通用寄存器组），346
half-precision floating-point group（半精度浮点组），342
Floating-point shuffle and word unpack operations

（浮点数求和和子解组操作），334
Fused-multiply-add（FMA）instructions（融合乘加指令），343
VFMADD subgroup（VFMADD 子组），344
VFMADDSUB subgroup（VFMADDSUB 子组），345
VFMSUBADD subgroup（VFMSUBADD 子组），345
VFMSUB subgroup（VFMSUB 子组），344
VFNMADD subgroup（VFNMADD 子组），346
VFNMSUB subgroup（VFNMSUB 子组），346
Fused-multiply-add programming（融合乘加编程），470
AvxFma.asm，474
AvxFma.cpp，471

G

Gather instructions（收集指令），340
General-purpose register instructions（通用寄存器指令），346，482
enhanced bit manipulation（增强易位操作）
AvxGprBitManip.cpp，486，488
AvxGrpBitManip_asm，487，489
program output（程序输出），489
stack contents（栈内容），489
flagless multiplication and bit shifts（不带标志的乘法和移位操作）
AvxGprMulxShiftx_asm，483-484
AvxGprMulxShiftx.cpp，482，484
program output（程序输出），485

H

Half-precision floating-point instructions（半精度浮点指令），342
Haswell-based quad-core processor（基于 Haswell 架构的 4 核处理器），625
Execution Engine（执行引擎），628
pipeline functionality（流水线功能），626
allocate/rename block（地址分配/重命名块），627
Branch Prediction Unit（分支预测单元），627
Instruction Decoders（指令解码器），627
Instruction Fetch（取指），627
micro-fusion（微融合），627
micro-ops（微操作），627
Pre-Decode Unit（预解码单元），627
Haswell microarchitecture（Haswell 微架构），3
Haswell processors（Haswell 处理器），328

I

Image-processing（图像处理）
image-thresholding program（图像阈值化程序），425
AvxPackedIntegerThreshold .asm，429
AvxPackedIntegerThreshold.h，426
pixel clipping（像素裁剪）
AvxPackedIntegerPixelClip_asm，420
AvxPackedIntegerPixelClip.cpp，418，423
AvxPackedIntegerPixelClip.h，418，422
definition（定义），417
instructions sequence（指令序列），425
mean execution times（平均执行时间），425
program output（程序输出），425
Image-thresholding program（图像阈值化程序），425
AvxPackedIntegerThreshold .asm，429
AvxPackedIntegerThreshold.h，426
Instruction pointer register（EIP，指令指针寄存器），13
Instruction set（指令集）
MMX technology（MMX 技术）
arithmetic group（算术运算组），140
comparison group（比较组），142
conversion group（转换组），142
data transfer group（数据传输组），139
insertion and extraction group（插入和提取组），144
local and shift group（逻辑和位移组），143
state and cache control group（状态和缓存控制组），145
unpack and shuffle group（解压和重排组），144

x86-32
binary arithmetic group（二进制算术组），18
byte set and bit string instruction group（字节集和位串指令组），22
conditional codes（条件代码），16
control transfer group（控制转移组），24
data comparison group（数据比较组），20
data conversion group（数据转换组），20
data transfer group（数据传输组），18
flag manipulation group（标志操作组），23
functional categories（功能类别），17
logical group（逻辑运算组），21
miscellaneous group（其他组），25
mnemonic suffixes（助记符后缀），16
purposes of（其目的），15
rotate and shift group（旋转和移位组），21
string instruction group（字符串指令组），22
test conditions（测试条件），16
x87 floating-point unit（FPU）（x87 浮点单元），95
arithmetic group（算术运算组），96
constants group（常量组），101
control group（控制组），101
data-comparison group（数据比较组），98
data transfer group（数据传输组），95
transcendental group（超越函数组），100
IntegerMulDiv program（IntegerMulDiv 程序），32
epilog，36
InvalidDivisior instruction（InvalidDivisor 指令），35
push ebx instruction（push ebx 指令），33
signed-integer division（有符号整数除法），36
source code implementation（源代码实现），32
Intel 80386 microprocessor（Intel 80386 微处理器），1
Internal architecture，x86-32（x86-32 内部架构），8
EFLAGS register（EFLAGS 寄存器），11
EIP register（EIP 寄存器），13
general-purpose registers（通用寄存器），9
instruction operands（指令操作数），13
memory addressing modes（内存寻址模式），14
segment registers（段寄存器），9

L

lzcnt instruction（lzcnt 指令），348

M

Masked move instructions（带掩码的移动指令），339
4 × 4 Matrices（4 × 4 矩阵）
definition（定义），260
formula（公式），260
SsePackedFloatingPointMatrix4 × 4 program（SsePackedFloatingPointMatrix4 × 4 程序），260
Matrix column means（矩阵列平均值）
AvxPackedFloatingPointColMeans .asm，398
AvxPackedFloatingPointColMeans.cpp，396，400
col_means array updation(col_means 数组更新)，401
program output（程序输出），402
Microarchitectures（微架构），2
MmxAddition program（MmxAddition 程序）
AddOpTable，153
cmp instruction（cmp 指令），154
coding implementation（编码实现），149-153
emms instruction（emms 指令），155
enumerators（枚举器），153
MmxAddBytes function（MmxAddBytes 函数），153
MmxAddOp parameter（MmxAddOp 参数），153
MmxAddWords function（MmxAddWords 函数），153
MmxVal parameter（MmxVal 参数），153
movq instruction（movq 指令），154
pshufw instruction（pshufw 指令），154
test cases（测试用例），155-156
Visual C++，154
MmxCalcMean program（MmxCalcMean 程序），172，178
array element（数组元素），177
coding implementation（编码实现），173-176
MmxCalcMeanTimed，176

pshufw instruction（pshufw 指令），177
punpcklbw and punpckhbw instructions（punpcklbw 和 punpckhbw 指令），176
MmxCalcMinMax program(MmxCalcMinMax 程序)
C++ counterpart（C++ 实现版本），168
coding implementation（编码实现），164-168
MmxCalcMinMax.exe，172
MmxCalcMinMax Timed.cpp file（MmxCalcMinMaxTimed.cpp 文件），170-171
pmaxub instruction（pmaxub 指令），169
pminub instruction（pminub 指令），168
pointer（指针），168
pshufw/pminub instruction（pshufw/pminub 指令），169
restrictions（限制），168
standard library function（标准库函数），168
MmxMultiplication
coding implementation（编码实现），160-161
MmxMulSignedWord，162-163
pmullw and pmulhw instructions（pmullw 和 pmulhw 指令），162
punpckhwd and punpckhwd instructions（punpcklbw 和 punpckhbw 指令），163
MmxShift program（MmxShift 程序），160
coding implementation（编码实现），156-159
counterators（效率器），159
jmp instruction（jmp 指令），159
movq instruction（movq 指令），159
pshufw instruction（pshufw 指令），159
MMX technology（MMX 技术），2，133，147
and x87 FPU（及 x87 FPU），172
data types（数据类型），137
instruction set（指令集），138
arithmetic group（算术运算组），139
comparison group（比较组），142
conversion group（转换组），142
data transfer group（数据传输组），139
insertion and extraction group（插入和提取组），144
local and shift group（逻辑和位移组），142
state and cache control group（状态和缓存控制组），145
unpack and shuffle group（解组和重排组），143
integer array processing（整数数组处理），164
MiscDefs.h，148
MmxVal.h，148
packed integer addition（组合型整数加法），147，149，164
packed integer multiplication（组合型整数乘法），147，160
packed integer shifts（组合型整数移位），156
register set（寄存器组），137
SIMD processing（SIMD 处理），133
arithmetic operations（算术运算），134
bit patterns（位模式），134
pmaxub instruction（pmaxub 指令），135
wraparound vs. saturated arithmetic（环绕和饱和运算），135
mulps instruction（mulps 指令），333
mulx instruction（mulx 指令），348

N

Nehalem microarchitecture（Nehalem 微架构），2
Netburst（Netburst 微架构），2
New instructions（新指令）
blend group（混合组），337
broadcast group（广播组），336
extract and insert group（提取和插入组），338
gather group（数据收集组），340
masked move group（掩码移动组），339
permute group（排列组），337
variable bit shift group（变长移位组），340
Non-temporal memory stores（无时态内存存储），637

O

Optimization techniques（优化技术），623，629
basic optimizations（基本优化），630
Branch Prediction Unit（分支预测单元），631
data alignment（数据对齐），633
floating-point arithmetic（浮点算术运算），631
SMID techniques（SMID 技术），634

P

Packed floating-point arithmetic（组合型浮点算术运算）
AvxPackedFloatingPointArithmetic_.asm，381，383
AvxPackedFloatingPointArithmetic.cpp，397，383
compare instruction（比较指令），244
conversions（数据转换），249
correlation coefficient（相关系数）
AvxPackedFloatingPointCorrCoef_.asm，391
AvxPackedFloatingPointCorrCoef.cpp，389，394
program output（程序输出），396
least squares（最小平方），254
matrix column means（矩阵列均值）
AvxPackedFloatingPointColMeans_.asm，398
AvxPackedFloatingPointColMeans.cpp，396，400
col_means array updation（col_means 数组更新），401
program output（程序输出），402
operations（操作），238
program output（程序输出），384
ToString_ formatting functions（ToString_ formatting 函数），237
YmmVal.h，378
Packed floating-point compares(组合型浮点数比较)
AvxPackedFloatingPointCompare_.asm，386，388
AvxPackedFloatingPointCompare.cpp，385，387
program output（程序输出），388-389
Packed integers（组合型整数）
arithmetic operations（算术运算），405
AvxPackedIntegerArithmetic_.asm，408
AvxPackedIntegerArithmetic.cpp，406
program output（程序输出），411
vpsllvd and vpsravd instructions（vpsllvd 和 vpsravd 指令），410
fundamentals（基本原则），273
histogram（直方图），279
threshold（阈值），288
unpack operations（组合操作），412
AvxPackedIntegerUnpack_.asm，414-415
AvxPackedIntegerUnpack.cpp，412，415
program output（程序输出），417
vpackssdw instruction（vpackssdw 指令），416
vpunpckldq and vpunpckhdq instructions（vpunpckldq 和 vpunpckhdq 指令），415-416
paddb instruction（paddb 指令），333
pdep instruction（pdep 指令），348
Permute instructions（排列指令），337
pext instruction（pext 指令），348
Pixel clipping algorithm（像素裁剪算法），337
AvxPackedIntegerPixelClip_.asm，420
AvxPackedIntegerPixelClip.cpp，418，423
AvxPackedIntegerPixelClip.h，418，422
definition（定义），417
instructions sequence（指令序列），423
mean execution times（平均执行时间），425
sample program（示例程序），425

Q

Quark microarchitecture（Quark 微架构），3

R

rdrand instruction（rdrand 指令），348
Roots of quadratic equation（二次方程的根），360
AvxScalarFloatingPointQuadEqu_.asm，363
AvxScalarFloatingPointQuadEqu.cpp，361
program output（程序输出），367
root computation equations（求根方程），367
solution forms（解决方案列表），365
roxr instruction（roxr 指令），348

S

Sandy Bridge，2
sarx instruction（sarx 指令），349
Scalar floating-point arithmetic（标量浮点算术运算），351
AvxScalarFloatingPointArithmetic_.asm，352-

353
AvxScalarFloatingPointArithmetic.cpp, 352-353
compare instructions (比较指令), 212
conversions (数据转换), 217
operation (操作), 207
parallelograms (平行四边形), 228
program output (程序输出), 354
roots of quadratic equation (二次方程的根), 360
AvxScalarFloatingPointQuadEqu_.asm, 363
AvxScalarFloatingPointQuadEqu.cpp, 361
program output (程序输出), 367
root computation equations (求根方程), 367
solution forms (解决方案列表), 365
spheres (球面), 225
spherical coordinates (球面坐标)
acos function (acos (反余弦) 函数), 374
AvxScalarFloatingPointSpherical_.asm, 369, 374
AvxScalarFloatingPointSpherical.cpp, 368, 373
call sin and call cos instructions (调用正弦和余弦指令), 375
program output (程序输出), 375
rectangular coordinates (直角坐标), 373
three-dimensional coordinate system (三维坐标系), 372
Scalar floating-point compares (标量浮点数比较)
AvxScalarFloatingPointCompare_.asm, 356
AvxScalarFloatingPointCompare.cpp, 355, 357
program output (程序输出), 359-360
vcmpsd and vcmpss instructions (vcmpsd 和 vcmpss 指令), 358
shlx instruction (shlx 指令), 349
shrx instruction (shrx 指令), 349
Silvermont microarchitecture (Silvermont 微架构), 3
Silvermont System on a Chip (SoC) microarchitecture (Silvermont 片上系统微架构), 3
Single instruction multiple data (SIMD, 单指令多数据), 133, 563
AVX-64 execution environment (AVX-64 执行环境)
data types (数据类型), 561
instruction set (指令集), 562
register set (寄存器组), 560
AVX-64 programming (AVX-64 编程)
ellipsoid calculations (椭圆体计算), 590
matrix inverse (矩阵求逆), 602
miscellaneous instructions (其他指令), 617
RGB image processing (RGB 图像处理), 595
processing algorithm (处理算法)
data prefetch (数据预取), 645
non-temporal memory stores (无时态内存存储), 637
SSE-64 execution environment (SSE-64 执行环境)
data types (数据类型), 559
instruction set (指令集), 559
register set (寄存器组), 557
SSE-64 programming (SSE-64 编程)
image conversion (图像转换), 571
image histogram (图像直方图), 563
vector arrays (向量数组), 580
Spherical coordinates program (球面坐标编程)
acos function (acos (反余弦) 函数), 374
AvxScalarFloatingPointSpherical_.asm, 369, 374
AvxScalarFloatingPointSpherical.cpp, 368, 373
call sin and call cos instructions (调用正弦 (sin) 和余弦 (cos) 指令), 375
program output (程序输出), 375
rectangular coordinates (直角坐标), 373
three-dimensional coordinate system (三维坐标系), 372
SsePackedFloatingPointArithmetic program (SsePackedFloatingPointArithmetic 程序)
coding implementation (编码实现), 238
movaps and movapd (movaps 和 movapd), 243
SsePackedFpMath32_ and SsePackedFpMath64_ functions (SsePackedFpMath32_ 和 Sse-

PackedFpMath64 函数), 243
XmmVal instances (XmmVal 实例), 243
SsePackedFloatingPointCompare program (SsePackedFloatingPointCompare 程序), 248
cmpps and cmppd instructions (cmpps 和 cmppd 指令), 244
coding implementation (编码实现), 245-247
SsePackedFloatingPointConversions program (SsePackedFloatingPointConversions 程序)
coding implementation (编码实现), 249-252
cvtps2pd instruction (cvtps2pd 指令), 253
SsePackedFloatingPointLeastSquares program (SsePackedFloatingPointLeastSquares 程序)
coding implementation (编码实现), 254
tmain function (tmain 函数), 258
xorpd instruction (xorpd 指令), 259
SsePackedFloatingPointMatrix4×4 program (SsePackedFloatingPointMatrix4×4 程序), 260
SsePackedIntegerFundamentals program (SsePackedIntegerFundamentals 程序)
coding implementation (编码实现), 273
pmulld instruction (pmulld 指令), 278
SsePiSubI32 function (SsePiSubI 函数), 277
SsePackedIntegerHistogram program (SsePackedIntegerHistogram 程序)
coding implementation (编码实现), 279
grayscale image (灰度图), 279
SsePackedIntegerThreshold program (SsePackedIntegerThreshold 程序), 288
SseScalarFloatingPointArithmetic program (SseScalarFloatingPointArithmetic 程序), 212
andps instruction (andps 指令), 212
caller-supplied array (用户提供的数组), 211
coding implementation (编码实现), 208-211
movss instruction (movss 指令), 211
SseScalarFloatingPointCompare program (SseScalarFloatingPointCompare 程序)
coding implementation (编码实现), 213-215
comiss and comisd instructions (comiss 和 comisd 指令), 216
single-precision and double precision (单精度和双精度), 216
SseSfpCompareFloat function (SseSfpCompareFloat 函数), 217
SseScalarFloatingPointConversions program (SseScalarFloatingPointConversions 程序), 217-218
SseScalarFloatingPointParallelograms program (SseScalarFloatingPointParallelograms 程序)
coding implementation (编码实现), 228
cos function (cos (余弦) 函数), 234
formulas (公式), 234
illustration of (图解), 233
sin function (sin (正弦) 函数), 234
sub esp,8 statement (sub esp, 8 语句), 234
transcendental instructions (超越数指令), 234
SseScalarFloatingPointSpheres program (SseScalarFloatingPointSpheres 程序), 225
coding implementation (编码实现), 225-226
mulsd instruction (mulsd 指令), 227
xorpd instruction (xorpd 指令), 227
SseTextStringCalcLength program (SseTextStringCalcLength 程序), 311
coding implementation (编码实现), 312
pcmpistri instruction (pcmpistri 指令), 315
strlen function (strlen 函数), 315
SseTextStringReplaceChar program (SseTextStringReplaceChar 程序), 324
coding implementation (编码实现), 316
EOS terminator byte (EOS 终止符), 322
pcmpistrm xmm1,xmm2,40h instruction (pcmpistrm xmm1,xmm2,40h 指令), 322
popcnt edx,edx instruction (popcnt edx,edx 指令), 322
pshufb xmm6,xmm5 instruction (pshufb xmm6,xmm5 指令), 322
State transition penalties (状态转换的额外代价), 336
Streaming SIMD Extension (SSE, 流式 SIMD 扩展), 179
Strings (字符串)
CompareArrays

repne cmpsd，83
results for（其结果），83
source code implementation（源代码实现），80
test instruction（测试指令），83
ConcatStrings
declaration statement（声明语句），79
des_index，79
rep movsw，80
repne scasw，80
counting characters（计算字符长度）
CountChars program（CountChars 程序），76
search character（字符搜索），75
source code implementation（源代码实现），74
text string（文本字符串），75
ReverseArray
parameters（参数），85
pushfd and popfd（pushfd 和 popfd），85
results for（其结果），86
source code implementation（源代码实现），83

T

TemperatureConversions program（TemperatureConversions 程序），104，107
code implementation（编码实现），104
.const directive（.const 指示符），106
fild instruction（fild 指令），106
floating-point value（浮点值），107
formula（公式），106
memory operands（内存中的操作数），107
real8 directive（real8 指示符），106
stack register（栈寄存器），106
Text string（文本字符串），303
fundamentals（基础内容）
EOS character（EOS 终止符），303
explicit/implicit length（显式/隐式长度），304
IntRes1 algorithm（IntRes1 算法），310
pcmpestri instruction（pcmpestri 指令），304
pcmpestrm instruction（pcmpestrm 指令），304
pcmpistri instruction（pcmpistri 指令），309-310
pcmpistrm instruction（pcmpistrm 指令），307
pcmpistrX flow diagram（pcmpistrX 流程图），305
processor exception（处理器异常），311
length calculation（长度计算），311
replace characters（字符替换），316
tzcnt instruction（tzcnt 指令），349

V

vaddsd instruction（vaddsd 指令），335
vaddss instruction（vaddss 指令），334-335
Variable bit shift instructions（变长移位指令），340
vblendvps instruction（vblendvps 指令），457
vbroadcastf128 instruction（vbroadcastf128 指令），337
vbroadcasti128 instruction（vbroadcasti128 指令），337
vbroadcastsd instruction（vbroadcastsd 指令），337
vbroadcastss instruction（vbroadcastss 指令），337
vcmpsd and vcmpss instructions（vcmpsd 和 vcmpss 指令），338
vcvtps2pd instruction（vcvtps2pd 指令），335
vdivps instruction（vdivps 指令），333
vextracti128 instruction（vextracti128 指令），339
VFMADD instructions（VFMADD 指令），344
VFMADDSUB instructions（VFMADDSUB 指令），345
VFMSUBADD instructions（VFMSUBADD 指令），345
VFMSUB instructions（VFMSUB 指令），344
VFNMADD instructions（VFNMADD 指令），346
VFNMSUB instructions（VFNMSUB 指令），346
vgatherdd instruction（vgatherdd 指令），342
vgatherd instruction（vgatherd 指令），341
vgatherdpd instruction（vgatherdpd 指令），342
vgatherdps instruction（vgatherdps 指令），340，342
vgatherdq instruction（vgatherdq 指令），342
vgatherqd instruction（vgatherqd 指令），342
vgatherq instruction（vgatherq 指令），341

vgatherqpd instruction（vgatherqpd 指令），342
vgatherqps instruction（vgatherqps 指令），342
vgatherqq instruction（vgatherqq 指令），342
vinserti128 instruction（vinserti128 指令），339
vmaskmovpd instruction（vmaskmovpd 指令），339
vmaskmovps instruction（vmaskmovps 指令），339
vmulps instruction（vmulps 指令），333
vmulsd instruction（vmulsd 指令），334
vpackssdw instruction（vpackssdw 指令），416
vpaddb instruction（vpaddb 指令），333
vpblendd instruction（vpblendd 指令），337
vpblendvb instruction（vpblendvb 指令），457
vpblendw and vpblendd instructions（vpblendw 和 vpblendd 指令），457
vpbroadcastb instruction（vpbroadcastb 指令），337
vpbroadcastd instruction（vpbroadcastd 指令），337
vpbroadcastq instruction（vpbroadcastq 指令），337
vpbroadcastw instruction（vpbroadcastw 指令），337
vperm2f128 instruction（vperm2f128 指令），338
vperm2i128 instruction（vperm2i128 指令），338
vpermd instruction（vpermd 指令），338
vpermilpd instruction（vpermilpd 指令），338
vpermilps instruction（vpermilps 指令），338
vpermpd instruction（vpermpd 指令），338
vpermps instruction（vpermps 指令），338
vpermq instruction（vpermq 指令），338
vpmaskmovd instruction（vpmaskmovd 指令），339
vpmaskmovq instruction（vpmaskmovq 指令），339
vpsllvd and vpsravd instructions（vpsllvd 指令），340
vpsllvd instruction（vpsllvq 指令），340
vpsllvq instruction（vpsravd 指令），340
vpsrlvd instruction（vpsrlvd 指令），340
vpsrlvq instruction（vpsrlvq 指令），340
vpsubb instruction（vpsubb 指令），333
vpunpckldq and vpunpckhdq instructions（vpunpckldq 和 vpunpckhdq 指令），415-416
vsqrtsd instruction（vsqrtsd 指令），335
vsqrtss and vsqrtsd instructions（vsqrtss 和 vsqrtsd 指令），335
vsqrtss instruction（vsqrtss 指令），335
vsubpd instruction（vsubpd 指令），333
vzeroall instruction（vzeroall 指令），336
vzeroupper instruction（vzeroupper 指令），336

X

x86-32 assembly language programming（x86-32 汇编语言编程），27
arrays（数组），54
one-dimensional arrays（一维数组），55
two-dimensional arrays（二维数组），60
CalcSum，28
argument passing（参数传递），28
function epilogs（epilogs 函数），31
function prologs（prologs 函数），31
MASM directives（MASM 指示符），30
stack-related operations（栈相关操作），28
calling convention（调用约定）
CalculateSums_，39
C++ and assemble language listings（C++ 和汇编语言代码清单），37
LocalVars program（LocalVars 程序），37
prolog and epilog（序言和结语），41
results for（其结果），41
stack organization（栈的组织形式），40
condition codes（条件码），49
_min and _max macros（_min 和 _max 宏），53
results for（其结果），53
setcc instruction（setcc 指令），54
SignedMinA_ and SignedMaxA_（SignedMinA_ 和 SignedMaxA_），53
source code implementation（源代码实现），49
integer addition（整数加法）
extern directive（extern 指示符），48
register AX（寄存器 AX），48
results for（其结果），49
source code implementation（源代码实现），46
IntegerMulDiv，32
memory addressing modes（内存寻址模式），41
.const directive（.const 指示符），44
hardware features（硬件特性），46
MemoryAddressing_ function（Memory

Addressing_ 函数), 44
MemoryOperands_ function (MemoryOperands_ 函数), 44
results for (其结果), 45
source code implementation (源代码实现), 41
strings (字符串)
CompareArrays, 80
ConcatStrings, 76
counting characters (计算字符长度), 74
ReverseArray, 83
structures (结构体), 67
dynamic structure creation(动态结构体创建), 70
simple structures (简单结构体), 67
Visual Studio project (Visual Studio 工程), 28
x86-32 core architecture (x86-32 核心架构), 1
data types (数据类型)
BCD values (BCD 值), 8
bit fields (位域), 7
bit strings (位串), 8
fundamental (基本), 4
numerical (数值), 5
packed (组合), 6
strings (字符串), 7
history of (历史回顾), 1
instruction set (指令集)
binary arithmetic group (二进制算术组), 18
byte set and bit string instruction group (字节设置和二进制位串指令组), 22
conditional codes (条件码), 16
control transfer group (条件传输组), 24
data comparison group (数据比较组), 20
data conversion group (数据转换组), 20
data transfer group (数据传输组), 18
flag manipulation group (标志操作组), 23
functional categories (功能类别), 17
logical group (逻辑运算组), 21
miscellaneous group (其他组), 25
mnemonic suffixes (助记符后缀), 16
purposes of (其目的), 15
rotate and shift group (旋转和移位组), 21
string instruction group (串指令组), 22
test conditions (测试条件), 16
internal architecture (内部架构), 8
EFLAGS register (EFLAGS 寄存器), 11
EIP register (EIP 寄存器), 13
general-purpose registers (通用寄存器), 9
instruction operands (指令操作数), 13
memory addressing modes (内存寻址模式), 14
segment registers (段寄存器), 9
x86-64
instruction set (指令集), 499
deprecated resources (不鼓励使用的资源), 502
invalid instructions (无效指令), 500
new instructions (新指令), 500
operand sizes (操作数长度), 499
internal architecture (内部架构), 491
general-purpose registers (通用寄存器), 9
instruction operands (指令操作数), 494
memory addressing modes (内存寻址模式), 495
RFLAGS register (RFLAGS 寄存器), 494
RIP register (RIP 寄存器), 494
optimization techniques (优化技术), 629
basic optimizations (基本优化), 630
Branch Prediction Unit (分支预测单元), 631
data alignment (数据对齐), 633
floating-point arithmetic (浮点算术), 631
SMID techniques (SIMD 技术), 634
processor microarchitecture (处理器微架构), 623
Execution engine (执行引擎), 628
multi-core processor (多核处理器), 624
pipeline functionality (流水线功能), 626
vs. x86-32 (与 x86-32), 32, 497
x86-64 assembly language programming (x86-64 汇编语言编程), 503
calling convention (调用约定), 523
non-volatile registers (非易变寄存器), 528
non-volatile XMM registers (非易变 XMM 寄存器), 533

prologs and epilogs（序言和结语），539
stack frame pointer（栈帧指针），524
ConcatStrings，533
FloatingPointArithmetic，519
IntegerArithmetic，505
argument registers（参数寄存器），508-509
IntegerAdd_ function（IntegerAdd_ 函数），509
IntegerDiv_ function（IntegerDiv_ 函数），510
IntegerMul_ function（IntegerMul_ 函数），510
stack layout（栈布局），509
typedef statements（typedef 声明），508
uninitialized stack space(未经初始化的栈空间)，508
IntegerOperands，514
leaf functions（叶函数），504
MemoryAddressing，511
non-leaf function（非叶函数），504
overview（概览），503
two-dimensional array（二维数组），546
volatile and non-volatile registers（易变与非易变寄存器），504
x86-AVX data types（x86-AVX 数据类型），329
x86-AVX execution environment（x86-AVX 执行环境），329
data types（数据类型），329
instruction syntax（指令语法），330
register set（寄存器组），329
x86-AVX feature extensions（x86-AVX 功能扩展），329
x86-AVX instruction set（x86-AVX 指令集），329
cvtps2pd instruction（cvtps2pd 指令），335
feature extension instructions（功能扩展指令），342
fused-multiply-add group（融合乘加组），343
general-purpose register group（通用寄存器组），346
half-precision floating-point group（半精度浮点组），342
floating-point shuffle and word unpack operations（浮点数重排和字解组操作），334
mulps instruction（mulps 指令），333
new instructions（新指令）
blend group（混合组），337
broadcast group（广播组），336
extract and insert group（提取和插入组），338
gather group（收集组），340
masked move group（掩码移动组），339
permute group（排列组），337
variable bit shift group（变长移位组），340
paddb instruction（paddb 指令），333
state transition penalties（状态转换导致的额外代价），336
vaddsd instruction（vaddsd 指令），335
vaddss instruction（vaddss 指令），334-335
vcvtps2pd instruction（vcvtps 指令），335
vdivps instruction（vdivps 指令），333
vmulps instruction（vmulps 指令），333
vmulsd instruction（vmulsd 指令），334
vpaddb instruction（vpaddb 指令），333
vpsubb instruction（vpsubb 指令），333
vsqrtsd instruction（vsqrtsd 指令），335
vsqrtss and vsqrtsd instructions（vsqrtss 和 vsqrtsd 指令），335
vsqrtss instruction（vsqrtss 指令），335
vsubpd instruction（vsubpd 指令），333
vzeroall instruction（vzeroall 指令），336
vzeroupper instruction（vzeroupper 指令），336
x86-AVX instruction syntax(x86-AVX 指令语法)，330
x86-AVX programming（x86-AVX 编程）
cpuid instruction（cpuid 指令），439
AvxCpuid_asm，444
AvxCpuid.cpp，439
data-manipulation instructions（数据操作指令），447
data blend（数据混合），453
data broadcast（数据广播），447
data gather（数据收集），463
data permute（数据排列），458
fused-multiply-add programming（融合乘加编程），470
AvxFma_asm，474

AvxFma.cpp，471
general-purpose register instructions（通用寄存器指令），482
enhanced bit manipulation（增强型位操作），486
flagless multiplication and bit shifts（不带标志位的乘法和移位操作），482
image processing（图像处理）
image-thresholding program（图像阈值化程序），425
pixel clipping algorithm（像素裁剪算法），417
packed floating-point arithmetic（组合型浮点数算术运算）
AvxPackedFloatingPointArithmetic_.asm，381，383
AvxPackedFloatingPointArithmetic.cpp，379，383
correlation coefficient（相关系数），389，394
matrix column means（矩阵列均值），396
program output（程序输出），384
YmmVal.h，378
packed floating-point compares（组合型浮点数比较），385
AvxPackedFloatingPointCompare_.asm，386，388
AvxPackedFloatingPointCompare.cpp，379，383
program output（程序输出），384
packed integer（组合型整数），385
arithmetic operations（算术运算），405
unpack operations（解组操作），412
scalar floating-point arithmetic（标量浮点数算术运算），351
AvxScalarFloatingPointArithmetic_.asm，352-353
AvxScalarFloatingPointArithmetic.cpp，352-353
program output（程序输出），354
roots of quadratic equation（二次方程的根），360
spherical coordinates（球面坐标），368
scalar floating-point compares（标量浮点数比较），355
AvxScalarFloatingPointCompare_.asm，356
AvxScalarFloatingPointCompare.cpp，355，357
program output（程序输出），359-360
vcmpsd and vcmpss instructions（vcmpsd 和 vcmpss 指令），358
x86-AVX register set（x86-AVX 寄存器组），329
x86-AVX technologies（x86-AVX 技术），327
AVX，327-328
AVX2，328
AVX-512，328
Haswell Processors（Haswell 处理器），328
Sandy Bridge microarchitecture（Sandy Bridge 微架构），328
x86-SSE，179
evolution（演变），180
execution environment（执行环境），181
control-status register（控制－状态寄存器），183
data types（数据类型），181
register set（寄存器组），181
instruction set（指令集），188，197
cache control（缓存控制），204
miscellaneous（其他），205
non-temporal data transfer（无时态数据传输），204
packed floating-point arithmetic（组合浮点数算术运算），193
packed floating-point blend（组合浮点混合），197
packed floating-point comparison（组合浮点比较），195
packed floating-point conversion（组合浮点转换），195
packed floating-point data transfer（组合浮点数据传输），192
packed floating-point logical（组合浮点逻辑），198
packed floating-point shuffle and unpack（组

合浮点重排和解组），196
packed integer arithmetic（组合整数算术运算），199
packed integer blend（组合整数混合），203
packed integer comparison（组合整数比较），200
packed integer conversion（组合整数转换），201
packed integer data transfer（组合整数数据传输），199
packed integer extensions（组合整数扩展），198
packed integer insertion and extraction（组合整数插入和提取），202
packed integer shift（组合整数移位），203
packed integer shuffle and unpack（组合整数重排和解组），202
scalar floating-point arithmetic（标量浮点算术运算），190
scalar floating-point comparison（标量浮点比较），191
scalar floating-point conversion（标量浮点转换），191
scalar floating-point data transfer（标量浮点数据传输），190
text string processing（文本字符串处理），204
overview（概览），179
processing techniques（处理技巧），188
x86-SSE programming（x86-SSE 编程）
packed floating-point（组合浮点数）
packed integers（组合整数）
fundamentals（基本），273
histogram（直方图），279
threshold（阈值），288
scalar floating-point（标量浮点）
text string（字符串）
x86 Streaming SIMD Extensions（x86 流式 SIMD 扩展）
x87 floating-point unit（FPU）（x87 浮点单元），87，103
architecture（架构），87
control register（控制寄存器），89
data registers（数据寄存器），88
encoding process（编码过程），93
compliant fields（兼容位域），93
negative zero（负零值），94
Not a Number（NaN，非数），94
positive zero（正零值），94
exception handlers（异常处理），91
floating-point array（浮点数组），112
floating-point values（浮点数值），108
instruction set（指令集），95
arithmetic group（算术运算组），96
constants group（常量组），101
control group（控制组），101
data-comparison group（数值比较组），98
data transfer group（数据传输组），95
transcendental group（超越组），100
memory operands（内存操作数），92
parameters（参数），92
simple arithmetic（简单算术运算），104
stack register（栈寄存器），124
status register（状态寄存器），90
transcendental instructions（超越指令），120

现代x86汇编语言程序设计

Modern X86 Assembly Language Programming 32-bit, 64-bit, SSE, and AVX

本书讲述x86汇编语言编程的基础知识，重点关注与应用软件开发相关的x86指令集。作者从应用程序编程的角度来解释x86处理器的内部架构，并且提供大量示例代码，帮助读者快速理解x86汇编语言编程和x86平台的计算资源。

本书既适用于想要学习如何使用x86汇编语言编写性能增强算法和函数的软件开发者，也适用于对x86汇编语言编程有基本了解并且想学习x86的SSE和AVX指令集的软件开发者。

通过阅读本书，你将学到

- 如何使用x86-32和x86-64指令集构建可被高级语言C++调用的性能增强函数。
- 如何使用x86汇编语言操纵常见的编程结构体，包括整数、浮点值、字符串、数组和结构。
- 如何使用SSE和AVX扩展改进计算密集型算法的性能，如图像处理、计算机图形学、数学和统计学等领域。
- 如何使用不同的编码策略和技巧优化x86的微架构，以最大化性能。

作者简介



丹尼尔·卡斯沃姆（Daniel Kusswurm）在软件开发和计算机科学领域拥有超过30年的专业经验。在几十年的职业生涯中，他曾为各种医疗设备、科学仪器和图像处理应用编写了大量创新性的代码。在这些项目中，他有很多使用x86汇编语言的成功经验，有些是显著提高计算密集型算法的性能，有些是巧妙解决技术难题。丹尼尔拥有北伊利诺伊大学电子工程技术硕士学位和德保罗大学计算机科学博士学位。

Apress

编辑热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字网站：www.rmmedia.com.cn

上架书号：计算机/程序设计

ISBN 978-7-111-54278-0

定价：79.00元

[illegible]
书名吼现代嗢叡工编语言程序设计 = [illegible]
[illegible]
作者吼(美)丹尼尔咭卡斯沃姆著
页数吼[illegible]
[illegible]
[illegible]
出版日期[illegible]
出版社吼机械工业出版社