Даниэль Куссвюрм

Профессиональное программирование на ассемблере

x64

С расширениями AVX, AVX2 и AVX-512

ДМК ПРЕСС
издательство

Modern X86 Assembly Language Programming

Covers x86 64-bit, AVX, AVX2, and AVX-512

Second Edition

Daniel Kusswurm

Профессиональное программирование на ассемблере x64

с расширениями AVX, AVX2 и AVX-512

Даниэль Куссвюрм

ДМК Пресс
издательство
Москва, 2021

УДК 004.4
ББК 32.972
К94

Даниэль Куссвюрм
К94 Профессиональное программирование на ассемблере x64 с расширениями AVX, AVX2 и AVX-512 / пер. с анг. В. С. Яценкова. – М.: ДМК Пресс, 2021. – 628 с.: ил.

ISBN 978-5-97060-928-6

В книге рассматривается программирование для 64-разрядной архитектуры x86 и использование расширенного набора векторных команд (AVX). Изучив этот материал, вы сможете кодировать быстродействующие функции и алгоритмы с использованием 64-разрядного языка ассемблера x86 и расширений набора команд AVX, AVX2 и AVX-512.

Примеры исходного кода разработаны с использованием Visual Studio C++ и MASM; для их запуска рекомендуется ПК на базе x86 с 64-разрядной ОС Windows 10 и процессором, поддерживающим AVX.

Предполагается, что читатели имеют опыт программирования на языках высокого уровня и базовые знания C++.

Книга предназначена разработчикам, которые хотят научиться писать код с использованием языка ассемблера x64.

УДК 004.4
ББК 32.972

Modern X86 Assembly Language Programming; Covers x86 64-bit, AVX, AVX2, and AVX-512 by Daniel Kusswurm, edition: 2

This edition has been translated and published under licence from APress Media, LLC, part of Springer Nature.

APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the accuracy of the translation.

ISBN (анг.) 978-1-4842-4062-5
ISBN (рус.) 978-5-97060-928-6

Оглавление

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте **www.dmkpress.com**, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

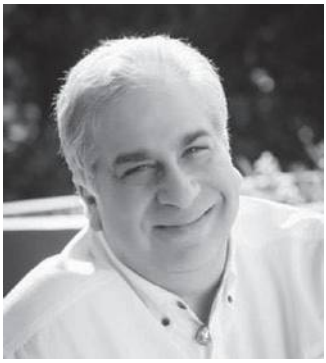
Ссылку на подозрительные материалы можно прислать по адресу **dmkpress@gmail.com**.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Даниэль Куссвюрм более 30 лет занимается разработкой программного обеспечения и информационными технологиями. За свою карьеру он разработал инновационное программное обеспечение для различного медицинского и научного оборудования и приложения для обработки изображений. Во многих из этих проектов он успешно использовал язык ассемблера x86 для значительного повышения быстродействия ресурсоемких алгоритмов и решения уникальных задач программирования. Он получил степень бакалавра электроники и электротехники в Университете Северного Иллинойса, а также степень магистра и доктора информатики в Университете ДеПола.

О техническом редакторе



Пол Коэн пришел в корпорацию Intel на самых первых этапах развития архитектуры x86, начиная с 8086, и ушел из Intel после 26 лет работы в сфере продаж, маркетинга и управления. В настоящее время он является партнером Douglas Technology Group и специализируется на написании технической литературы по заказу Intel и других корпораций. Пол также совместно с Академией молодых предпринимателей (YEA) ведет учебный курс, который превращает учеников средних и старших классов в настоящих, уверенных в себе предпринимателей. Он также является комиссаром по вопросам дорожного движения города Бивертон, штат Орегон, и входит в совет директоров нескольких некоммерческих организаций.

Благодарности

Производство фильма и издание книги в чем-то похожи. Трейлеры к фильму чествуют актеров, играющих главных героев. На обложке книги звучат имена авторов. Актеры и авторы в конечном итоге получают признание публики за свои усилия. Однако невозможно создать фильм или опубликовать книгу без преданности делу, опыта и творческого подхода профессиональной закадровой команды. Эта книга не исключение.

Я хочу поблагодарить за прекрасную работу талантливый редакционный коллектив Apress, особенно Стива Энглина, Марка Пауэрса и Мэтью Муди. Пол Коэн заслуживает отдельной похвалы за его тщательный технический обзор и практические предложения. Корректор Эд Куссвюрм заслуживает аплодисментов и признательности за его упорный труд и конструктивные отзывы. Я беру на себя полную ответственность за любые оставшиеся недостатки.

Я также хочу поблагодарить Нирмала Селвараджа, Дулси Нирмала, Кезию Эндсли, Дханеша Кумара и весь редакционный персонал Apress за их вклад, а моих профессиональных коллег за их одобрение и поддержку. Наконец, я хотел бы поблагодарить родителей Армина (вечная ему память) и Мэри, а также двоюродную сестру Мэри и двоюродных братьев Тома, Эда и Джона, которые вдохновляли меня на написание этой книги.

Вступление

С момента изобретения персонального компьютера (ПК) разработчики программного обеспечения использовали язык ассемблера x86 для создания инновационных решений применительно к широкому спектру алгоритмических задач. В первые дни эры ПК было обычной практикой писать большие фрагменты программного кода или полные приложения с использованием языка ассемблера x86. Учитывая преобладание в XXI веке языков высокого уровня, таких как C++, C#, Java и Python, может быть удивительно узнать, что многие разработчики программного обеспечения все еще используют язык ассемблера для кодирования критически важных в плане быстродействия участков своих программ. И хотя с годами компиляторы заметно улучшились с точки зрения генерации машинного кода, который стал эффективнее по объему и производительности, все еще остаются ситуации, когда разработчику программного обеспечения имеет смысл использовать преимущества программирования на языке ассемблера.

Архитектура современных процессоров x86 *с одним потоком команд и множеством потоков данных* (SIMD, single instruction stream – multiple data stream) является еще одним объяснением постоянного интереса к программированию на языке ассемблера. Процессор с поддержкой SIMD обладает вычислительными ресурсами, которые упрощают параллельные вычисления с использованием нескольких значений данных, что может значительно повысить быстродействие приложений, обеспечивающих высокую скорость отклика в реальном времени. Архитектуры SIMD также хорошо подходят для ресурсоемких проблемных областей, таких как обработка изображений, кодирование аудио и видео, автоматизированное проектирование, компьютерная графика и интеллектуальный анализ данных. К сожалению, многие языки высокого уровня и инструменты разработки по-прежнему не могут полностью или хотя бы частично использовать возможности SIMD современного процессора x86. С другой стороны, язык ассемблера открывает разработчику программного обеспечения полный доступ к ресурсам SIMD процессора.

О чем эта книга

Эта книга рассказывает о программировании на 64-битном (x86-64) языке ассемблера x86. Содержание и структура книги призваны помочь вам быстро освоить программирование на языке ассемблера x86-64 и вычислительные ресурсы векторных расширений набора команд Advanced Vector Extensions (AVX)[1]. Она также содержит множество примеров исходного кода, которые способствуют ускоренному изучению и пониманию основных конструкций языка ассемблера x86-64 и концепций программирования SIMD. После прочтения этой книги вы сможете кодировать быстродействующие функции и алгоритмы с помощью языка ассемблера x86-64 и наборов инструкций AVX, AVX2 и AVX-512.

Прежде чем продолжить, я должен обратить ваше внимание, что *в этой книге не рассматривается программирование на языке ассемблера x86-32*. В нем также не обсуждаются устаревшие технологии x86, такие как модуль с плавающей запятой x87, MMX и Streaming SIMD Extensions. Если вы заинтересованы в изучении этих тем, прочтите первое издание книги (в переводе не издавалось – *прим. перев.*). Эта книга не объясняет архитектурные особенности x86 или привилегированные команды процессора, которые используются в операционных системах. Однако вам все равно придется досконально изучить материал, представленный в данной книге, чтобы разрабатывать код на языке ассемблера x86 для использования в операционной системе.

Хотя теоретически возможно написать целую прикладную программу, используя только язык ассемблера, высокие требования к разработке современного программного обеспечения делают такой подход непрактичным и нецелесообразным. Вместо этого в данной книге основное внимание уделяется кодированию функций языка ассемблера x86-64, вызываемых из C++. Каждый пример исходного кода был создан с использованием Microsoft Visual Studio C++ и Microsoft Macro Assembler (MASM).

Целевая аудитория

Целевая аудитория этой книги – разработчики программного обеспечения, в том числе:

- разработчики, которые создают прикладные программы для платформ на базе Windows и хотят научиться писать алгоритмы и функции высокого быстродействия с использованием языка ассемблера x86-64;
- разработчики, которые создают прикладные программы для сред, отличных от Windows, и хотят изучить программирование на языке ассемблера x86-64;

[1] Расширение системы команд x86 для микропроцессоров Intel и AMD, предложенное Intel в марте 2008. – *Прим. перев.*

- разработчики, которые хотят научиться создавать вычислительные функции SIMD с использованием наборов команд AVX, AVX2 и AVX-512;
- разработчики и студенты, изучающие информатику, которые хотят добиться глубокого понимания платформы x86-64 и ее архитектуры SIMD.

Основная аудитория данной книги – это разработчики программного обеспечения на базе Windows, поскольку примеры исходного кода были разработаны с использованием Visual Studio C++ и MASM. Разработчики, ориентированные на платформы, отличные от Windows, также могут извлечь пользу из этой книги, поскольку большая часть информативного контента излагается независимо от какой-либо конкретной операционной системы. Предполагается, что читатели данной книги уже имеют опыт программирования на языках высокого уровня и базовые знания C++. Знакомство с программированием в Visual Studio или Windows PowerShell не требуется.

Обзор содержания

Основная цель этой книги – помочь вам изучить программирование на 64-битном языке ассемблера x86, а также набор команд AVX, AVX2 и AVX-512. Главы и содержание книги построены таким образом, чтобы достичь этой цели. Ознакомьтесь с кратким обзором того, что вы найдете в этой книге.

В главе 1 рассматривается основная архитектура платформы x86-64. Она включает в себя описание основных типов данных платформы, внутренней архитектуры, наборов регистров, операндов команд и режимов адресации памяти. В этой главе также описывается основной набор команд x86-64. В главах 2 и 3 объясняются основы программирования на языке ассемблера x86-64 с использованием базового набора команд и общих программных конструкций, включая массивы и структуры. Примеры исходного кода, представленные в этих (и последующих) главах, оформлены как рабочие программы, которые в процессе обучения вы можете запускать, изменять или иным образом экспериментировать с кодом.

Глава 4 посвящена архитектурным ресурсам AVX, включая наборы регистров, типы данных и набор команд. В главе 5 объясняется, как использовать набор команд AVX для выполнения скалярных арифметических операций с плавающей запятой, со значениями как с одинарной, так и с двойной точностью. В главах 6 и 7 рассказано про программирование AVX SIMD с использованием упакованных операндов, как целочисленных, так и с плавающей запятой.

Глава 8 знакомит с AVX2 и исследует его расширенные возможности, включая широковещательную передачу, сбор и перестановку данных. Здесь также объясняются операции *слитного умножения-сложения* (FMA, fused multiply-add). Главы 9 и 10 содержат примеры исходного кода, которые иллюстрируют различные вычислительные алгоритмы, использующие AVX2 с упакованными операндами. Глава 11 включает примеры исходного кода, демонстрирующие программирование FMA. В этой главе также рассмотрены примеры, иллюстрирующие недавние расширения платформы x86 с использованием регистров общего назначения.

В главе 12 детально рассмотрен набор команд AVX-512. В этой главе описаны наборы регистров и типы данных AVX-512. В ней также рассмотрены основные усовершенствования AVX-512, включая условное выполнение и слияние, встроенные широковещательные операции и округление на уровне команд.

В главах 13 и 14 содержится множество примеров исходного кода, демонстрирующих, как использовать эти расширенные функции.

В главе 15 представлен обзор современного многоядерного процессора x86 и лежащей в его основе микроархитектуры. В этой главе также описаны конкретные стратегии и методы программирования, которые можно использовать для повышения быстродействия кода на языке ассемблера x86. В главе 16 приведен обзор нескольких примеров исходного кода, иллюстрирующего передовые методы программирования на языке ассемблера x86, включая обнаружение функций процессора, ускоренный доступ к памяти и многопоточные вычисления.

В приложении A рассказано, как выполнить примеры исходного кода с помощью Visual Studio и MASM. Оно также содержит список ссылок и ресурсов, к которым вы можете обратиться для получения дополнительной информации о программировании на языке ассемблера x86.

Исходный код примеров

Примеры исходного кода этой книги доступны на веб-сайте Apress по адресу https://www.apress.com/us/book/9781484240625. Для каждой главы есть ZIP-архив, содержащий файлы исходного кода C++ и ассемблера, а также файлы проекта Visual Studio. Процедура установки не требуется. Вы можете просто извлечь содержимое ZIP-архива главы в папку по вашему выбору.

Внимание! Единственное назначение исходного кода – показать примеры программирования, которые напрямую связаны с темами, обсуждаемыми в этой книге. В этих примерах уделяется минимальное внимание важным проблемам разработки программного обеспечения, таким как надежная обработка ошибок, риски безопасности, вычислительная стабильность, ошибки округления или плохо согласованные функции. Вы принимаете на себя всю ответственность за решение этих проблем, если решите использовать исходный код каких-либо примеров в своих собственных программах.

Примеры исходного кода были созданы с помощью Visual Studio Professional 2017 (версия 15.7.1) на ПК с 64-разрядной Windows 10 Pro. На веб-сайте Visual Studio (https://visualstudio.microsoft.com) содержится дополнительная информация об этом и других выпусках Visual Studio. Технические сведения об установке, настройке и разработке приложений Visual Studio доступны по адресу https://docs.microsoft.com/en-us/visualstudio/?view=vs-2017.

Рекомендуемой аппаратной платформой для запуска примеров исходного кода является ПК на базе x86 с 64-разрядной ОС Windows 10 и процессором, поддерживающим AVX. Для запуска примеров исходного кода, в которых используются эти наборы команд, требуется процессор, совместимый с AVX2 или AVX-512. Вы можете использовать одну из свободно доступных утилит, перечисленных в приложении, чтобы определить, какие расширения набора команд x86-AVX поддерживает процессор вашего компьютера.

Глава 1

Архитектура ядра x86-64

В этой главе архитектура ядра x86-64 рассматривается с точки зрения прикладной программы. Она открывается кратким историческим обзором платформы x86, формирующим основу для понимания последующего материала. Затем следует обзор основных типов данных – числовых и SIMD. Далее рассмотрена архитектура ядра x86-64, включая описания наборов регистров процессора, флагов состояния, операндов команд и режимов адресации памяти. Глава завершается обзором набора команд ядра x86-64.

В отличие от языков высокого уровня, таких как C и C++, программирование на языке ассемблера требует от разработчика глубокого понимания конкретных архитектурных особенностей целевого процессора. Темы, обсуждаемые в данной главе, соответствуют этому требованию и заложат основу для понимания примеров кода, которые вы встретите далее в данной книге. В этой главе также представлен базовый материал, необходимый для понимания расширенных возможностей SIMD x86-64.

1.1. Исторический обзор

Прежде чем исследовать технические детали архитектуры ядра x86-64, полезно понять, как эта архитектура развивалась за минувшие годы. Следующий краткий обзор посвящен заслуживающим внимания усовершенствованиям процессоров и наборов команд, которые повлияли на то, как разработчики программного обеспечения используют язык ассемблера x86. Читатели, которым интересна более полная история происхождения x86, должны обратиться к ресурсам, перечисленным в приложении.

Платформа процессора x86-64 является расширением исходной платформы x86-32. Первым воплощением платформы x86-32 в кремнии был микропроцессор Intel 80386, представленный в 1985 году. Модель 80386 расширила архитектуру 16-битной 80286 за счет добавления 32-битных регистров и типов данных, режимов плоской модели памяти, 4 ГБ логического адресного пространства и страничной организации памяти. Процессор 80486 превзошел производительность 80386 за счет реализации кешей памяти на кристалле и оптимизированных команд. В отличие от процессора 80386, который нуждался во внешнем сопроцессоре 80387 (FPU, floating-point unit) для вычислений с плавающей запятой, большинство версий процессора 80486 также включали интегрированный модуль x87 FPU.

Расширение платформы x86-32 продолжилось выпуском в 1993 году первого процессора марки Pentium, представителя микроархитектуры P5. Улучшения производительности включали конвейер выполнения сдвоенных команд, 64-битную внешнюю шину данных и отдельные кеши памяти на кристалле как для кода, так и для данных. Более поздние версии (1997 г.) микроархитектуры P5 получили новый вычислительный ресурс, так называемую технологию MMX, которая поддерживает операции SIMD над упакованными целыми числами с использованием регистров разрядностью 64 бита. *Упакованное целое число* (packed integer) – это совокупность нескольких целочисленных значений, которые обрабатываются одновременно.

Микроархитектура P6, впервые использованная в Pentium Pro (1995 г.), а затем в Pentium II (1997 г.), расширила платформу x86-32 за счет *трехстороннего суперскалярного конвейера*. Это означает, что процессор может (в среднем) декодировать, отправлять и выполнять три отдельные команды в течение каждого тактового цикла. Другие усовершенствования P6 включали выполнение команд вне очереди, улучшенные алгоритмы предсказания ветвлений и упреждающего исполнения команд. Pentium III, также основанный на микроархитектуре P6, был выпущен в 1999 году и получил поддержку новой технологии SIMD под названием Streaming SIMD extension (SSE). Эта технология добавляет к платформе x86-32 восемь 128-битных регистров и команды, которые реализуют упакованную арифметику с плавающей запятой одинарной точности.

В 2000 году Intel представила новую микроархитектуру Netburst с технологией SSE2, которая расширила возможности SSE с плавающей запятой за счет обработки упакованных значений двойной точности. SSE2 также включала дополнительные команды, позволяющие использовать 128-битные регистры SSE для упакованных целочисленных вычислений и скалярных операций с плавающей запятой. Линейка процессоров, основанных на архитектуре Netburst, включала несколько вариантов Pentium 4. В 2004 году микроархитектура Netburst была модернизирована и теперь включает в себя SSE3 и технологию Hyper-Threading. SSE3 содержит новые команды операций с упакованными целыми числами и числами с плавающей запятой для платформы x86, в то время как технология Hyper-Threading распараллеливает конвейеры команд внешнего интерфейса процессора для повышения производительности. Процессоры с поддержкой SSE3 входят в 90-нм (и менее) версии линейки продуктов Pentium 4 и Xeon.

В 2006 году Intel запустила в массовое производство новую микроархитектуру под названием Core. Микроархитектура Core стала следствием глубокой модернизации многих интерфейсных конвейеров и исполнительных модулей Netburst с целью повышения производительности и снижения энергопотребления. Она также содержит ряд улучшений SIMD, включая SSSE3 и SSE4.1. Эти расширения добавили на платформу новые команды для работы с упакованными целыми числами и числами с плавающей запятой, но не добавили новых регистров или типов данных. К процессорам на основе микроархитектуры Core относятся процессоры серий Core 2 Duo, Core 2 Quad и Xeon 3000/5000.

Микроархитектура под названием Nehalem последовала за Core в конце 2008 года. Эта микроархитектура вернула платформе x86 гиперпоточность,

которая была исключена из микроархитектуры Core. Микроархитектура Nehalem также поддерживает SSE4.2. Это последнее усовершенствование x86-SSE добавило в набор команд x86-SSE несколько команд ускорителя для определенных приложений. SSE4.2 также содержит новые команды, упрощающие обработку текстовых строк с использованием 128-битных регистров x86-SSE. К процессорам на основе микроархитектуры Nehalem относятся процессоры Core i3, i5 и i7 первого поколения. Она также включает процессоры серий Xeon 3000, 5000 и 7000.

В 2011 году Intel запустила новую микроархитектуру под названием Sandy Bridge. В микроархитектуре Sandy Bridge появилась новая технология SIMD x86 под названием Advanced Vector Extensions (AVX). AVX поддерживает операции над упакованными числами с плавающей запятой (как одинарной, так и двойной точности) с использованием регистров шириной 256 бит. AVX также поддерживает новый синтаксис команд с тремя операндами, повышающий эффективность кода за счет уменьшения количества передач данных из регистра в регистр, которые должна выполнять функция программы. Процессоры на базе микроархитектуры Sandy Bridge включают в себя процессоры Core i3, i5 и i7 второго и третьего поколений, а также процессоры серии Xeon V2.

В 2013 году Intel представила свою микроархитектуру Haswell. Haswell поддерживает набор команд AVX2, который расширяет набор AVX командами поддержки операций с упакованными целыми числами с использованием регистров шириной 256 бит. AVX2 также поддерживает расширенные возможности передачи данных с помощью команд широковещательной передачи, сбора и перестановки. (Команды широковещательной передачи реплицируют одно значение в несколько мест; команды сбора данных загружают несколько элементов из несмежных ячеек памяти; команды перестановки переупорядочивают элементы упакованного операнда.) Другой особенностью микроархитектуры Haswell является включение в нее операции *слитного умножения-сложения* (FMA, fused multiply-add). FMA позволяет программным алгоритмам выполнять вычисления умножения-сложения (или скалярного произведения) с использованием одной операции округления с плавающей запятой, что помогает улучшить как быстродействие, так и точность. Микроархитектура Haswell также включает в себя несколько новых команд для работы с регистрами общего назначения. К процессорам на основе микроархитектуры Haswell относятся процессоры Core i3, i5 и i7 четвертого поколения. AVX2 поддерживают и более поздние поколения процессоров семейства Core, а также процессоры серий Xeon V3, V4 и V5.

Доработки платформы x86 не ограничивались усовершенствованиями SIMD. В 2003 году AMD представила свой процессор Opteron, который расширил исполняющее ядро x86 с 32 до 64 бит. Intel последовала примеру AMD в 2004 году, добавив практически такие же 64-разрядные расширения к своим процессорам, начиная с определенных версий Pentium 4. Все процессоры Intel на базе микроархитектур Core, Nehalem, Sandy Bridge, Haswell и Skylake поддерживают исполняющее ядро x86-64.

Процессоры AMD также претерпели изменения за прошедшие годы. В 2003 году AMD представила серию процессоров на базе своей микроархитектуры

K8. Оригинальные версии K8 включали поддержку MMX, SSE и SSE2, в то время как в более поздних версиях появилась поддержка SSE3. В 2007 году была запущена микроархитектура K10, которая включала расширение SIMD под названием SSE4a. Расширение SSE4a содержит несколько команд по сдвигу маски и хранению потоковых данных, которые не поддерживают процессоры Intel. Вслед за K10 в 2011 году AMD представила новую микроархитектуру Bulldozer. Микроархитектура Bulldozer включает поддержку SSSE3, SSE4.1, SSE4.2, SSE4a и AVX. Она также поддерживает FMA4 – версию слитного умножения-сложения с четырьмя операндами. Процессоры Intel не поддерживают команды FMA4. Обновление 2012 года для микроархитектуры Bulldozer под названием Piledriver включает поддержку как FMA4, так и трехоперандной версии FMA, которую некоторые утилиты для обнаружения функций ЦП и сторонние документы именуют FMA3. Самая последняя микроархитектура AMD, представленная в 2017 году, называется Zen (в 2018 году была представлена микроархитектура Zen 2 – *прим. перев.*). Эта микроархитектура включает усовершенствования набора команд AVX2 и используется в процессорах серии Ryzen.

Высокопроизводительные процессоры для настольных и серверных систем на основе микроархитектуры Intel Skylake-X, также впервые поступившие на рынок в 2017 году, включают новое расширение SIMD под названием AVX-512. Эта усовершенствованная архитектура поддерживает сжатые целые числа и операции с плавающей запятой с использованием регистров шириной 512 бит. AVX-512 также поддерживает дополнения архитектуры, которые упрощают условное слияние данных на уровне команд, управление округлением с плавающей запятой и широковещательные операции. Ожидается, что в ближайшие несколько лет и AMD, и Intel включат AVX-512 в свои основные процессоры для настольных ПК и ноутбуков.

1.2. Типы данных

Программы, написанные на ассемблере x86, могут использовать самые разные типы данных. Большинство типов данных программ происходят из небольшого набора основных типов данных, которые присущи платформе x86. Эти основные типы данных позволяют процессору выполнять числовые и логические операции с использованием целых чисел со знаком и без знака, значений с плавающей запятой одинарной (32-битные) и двойной (64-битные) точности, текстовых строк и значений SIMD. В этом разделе вы узнаете об основных типах данных, а также о нескольких различных типах данных, поддерживаемых x86.

1.2.1. Основные типы данных

Основной тип данных – это элементарная единица данных, которая обрабатывается процессором во время выполнения программы. Платформа x86 поддерживает основные типы данных с разрядностью от 8 бит (1 байт) до 128 бит (16 байт). В табл. 1.1 перечислены эти типы и обычные способы их использования.

Таблица 1.1. Основные типы данных

Тип данных	Длина (биты)	Типичное применение
Байт (byte)	8	Символы, небольшие целые числа
Слово (word)	16	Символы, целые числа
Двойное слово (doubleword)	32	Числа – целые и с плавающей запятой (одинарной точности)
Четверное слово (quadword)	64	Числа – целые и с плавающей запятой (двойной точности)
Двойное четверное слово (double quadword)	128	Упакованные целые числа, упакованные числа с плавающей запятой

Неудивительно, что размер основных типов данных зависит от целых степеней двойки. Разряды основного типа данных нумеруются справа налево, начиная с нуля и до значения `размер-1`, обозначающих младший и старший разряды числа соответственно. Основные типы данных размером более одного байта хранятся в последовательных ячейках памяти, начиная с младшего байта по наименьшему адресу памяти. Этот тип расположения байтов в памяти называется *прямым порядком байтов* (little endian). На рис. 1.1 показаны схемы нумерации битов и порядок байтов, которые применяются с основными типами данных.

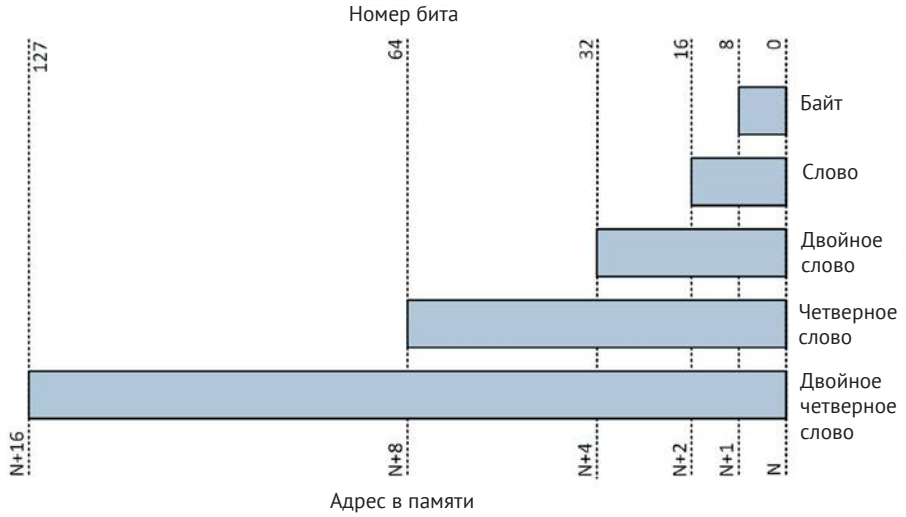


Рис. 1.1. Нумерация битов и порядок байтов для основных типов данных

Правильно выровненный основной тип данных – это тот, адрес которого без остатка делится на его размер в байтах. Например, двойное слово правильно выровнено, когда оно хранится в ячейке памяти с адресом, который делится на четыре без остатка. Точно так же четверные слова правильно выровнены по адресам, делящимся на восемь. Если это специально не оговорено требованиями

операционной системы, процессор x86 не требует правильного выравнивания многобайтовых типов данных в памяти. Однако стандартной практикой является правильное выравнивание всех многобайтовых значений, когда это возможно, чтобы избежать потенциальных потерь быстродействия, которые могут возникнуть, если процессору требуется доступ к смещенным в памяти данным.

1.2.2. Числовые типы данных

Числовой тип данных – это элементарное скалярное значение, такое как целое число или число с плавающей запятой. Все числовые типы данных, распознаваемые ЦП, представлены с использованием одного из основных типов данных, рассмотренных в предыдущем разделе. Таблица 1.2 содержит список числовых типов данных x86 вместе с соответствующими типами C/C++. Эта таблица также включает типы фиксированного размера, которые определены в заголовочном файле C++ `<cstdint>` (см. http://www.cplusplus.com/reference/cstdint/ для получения дополнительной информации об этом заголовочном файле). Набор команд x86-64 поддерживает арифметические и логические операции с использованием 8-, 16-, 32- и 64-битных целых чисел, как со знаком, так и без знака. Он также поддерживает арифметические вычисления и операции манипулирования данными с использованием значений с плавающей запятой одинарной и двойной точности.

Таблица 1.2. Числовые типы данных x86

Тип	Длина (биты)	Тип C/C++	`<cstdint>`
Целое со знаком	8	`char`	`int8_t`
	16	`short`	`int16_t`
	32	`int, long`	`int32_t`
	64	`long long`	`int64_t`
Беззнаковое целое	8	`unsigned char`	`uint8_t`
	16	`unsigned short`	`uint16_t`
	32	`unsigned int, unsigned long`	`uint32_t`
	64	`unsigned long long`	`uint64_t`
С плавающей запятой	32	`float`	Не применимо
	64	`double`	Не применимо

1.2.3. Типы данных SIMD

Тип данных SIMD – это последовательный набор байтов, используемый процессором для выполнения операций или вычислений, в которых участвуют несколько значений. Тип данных SIMD можно рассматривать как *объект-контейнер*, который содержит несколько экземпляров одного и того же основного типа данных (например, байты, слова, двойные слова или четверные слова). Как и в случае основных типов данных, разряды данных SIMD пронумерованы

справа налево от нуля и до значения размер-1, обозначающих младший и старший биты соответственно. При хранении данных SIMD в памяти тоже используется прямой порядок байтов, как показано на рис. 1.2.

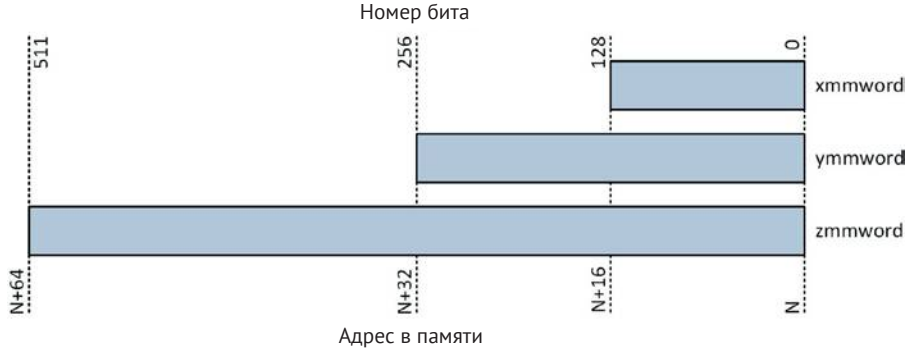


Рис. 1.2. Типы данных SIMD

Программисты могут использовать данные SIMD (или *упакованные данные*) для выполнения одновременных вычислений с использованием целых чисел или значений с плавающей запятой. Например, упакованные данные с разрядностью 128 бит можно использовать для хранения шестнадцати 8-битных целых чисел, восьми 16-битных целых чисел, четырех 32-битных целых чисел или двух 64-битных целых чисел. Упакованные данные шириной 256 бит могут содержать различные элементы данных, включая восемь значений с плавающей запятой одинарной точности или четыре значения с плавающей запятой двойной точности. Таблица 1.3 содержит полный список типов данных SIMD и максимальное количество элементов для различных числовых типов данных.

Таблица 1.3. Типы данных SIMD и максимальное количество элементов данных

Числовой тип	xmmword	ymmword	zmmword
8-битное целое число	16	32	64
16-битное целое число	8	16	32
32-битное целое число	4	8	16
64-битное целое число	2	4	8
Одинарной точности с плавающей запятой	4	8	16
Двойной точности с плавающей запятой	2	4	8

Как было сказано ранее в этой главе, усовершенствования SIMD регулярно добавлялись в платформу x86, начиная с технологии MMX в 1997 году, и вплоть до относительно недавнего обновления AVX-512. Это создает некоторые проблемы для разработчика, который хочет использовать эти технологии, поскольку типы упакованных данных, описанные в табл. 1.3, и связан-

ные с ними наборы команд не всегда поддерживаются всеми процессорами. К счастью, существуют методы, позволяющие во время выполнения программы определить наличие поддерживаемых функций SIMD и наборов команд конкретного процессора. Вы узнаете, как использовать некоторые из этих методов, в главе 16.

1.2.4. Прочие типы данных

Платформа x86 также поддерживает ряд прочих типов данных, включая строки, битовые поля и битовые строки. *Строка* x86 представляет собой непрерывный блок байтов, слов, двойных или четверных слов. Строки x86 используются для поддержки текстовых типов данных и операций обработки строк. Например, типы данных C/C++ `char` и `wchar_t` обычно реализуются с использованием байта или слова x86 соответственно. Строки x86 также можно использовать для выполнения операций обработки массивов, растровых изображений и аналогичных структур данных из непрерывных блоков. Набор команд x86 включает команды, позволяющие выполнять операции сравнения, загрузки, перемещения, сканирования и сохранения строковых данных.

Остальные типы данных представляют из себя битовые поля и битовые строки. *Битовое поле* – это непрерывная последовательность битов, которая используется некоторыми командами в качестве значения маски. Битовое поле может начинаться с любой битовой позиции в байте и содержать до 32 бит. *Битовая строка* – это непрерывная последовательность битов, содержащая до $2^{32}-1$ бит. Набор команд x86 содержит команды, которые могут сбрасывать, устанавливать, сканировать и тестировать отдельные биты в битовой строке.

1.3. Внутренняя архитектура

С точки зрения исполняемой программы внутренняя архитектура процессора x86-64 может быть логически разделена на несколько отдельных блоков. К ним относятся регистры общего назначения, флаги состояния и управления (регистр RFLAGS), указатель команд (регистр RIP), регистры XMM, а также управления и состояния операций с плавающей запятой (MXCSR). По умолчанию исполняемая программа использует только регистры общего назначения, регистр RFLAGS и регистр RIP. Программа не обязательно обращается к регистрам XMM, YMM, ZMM или MXCSR. На рис. 1.3 показана внутренняя архитектура процессора x86-64.

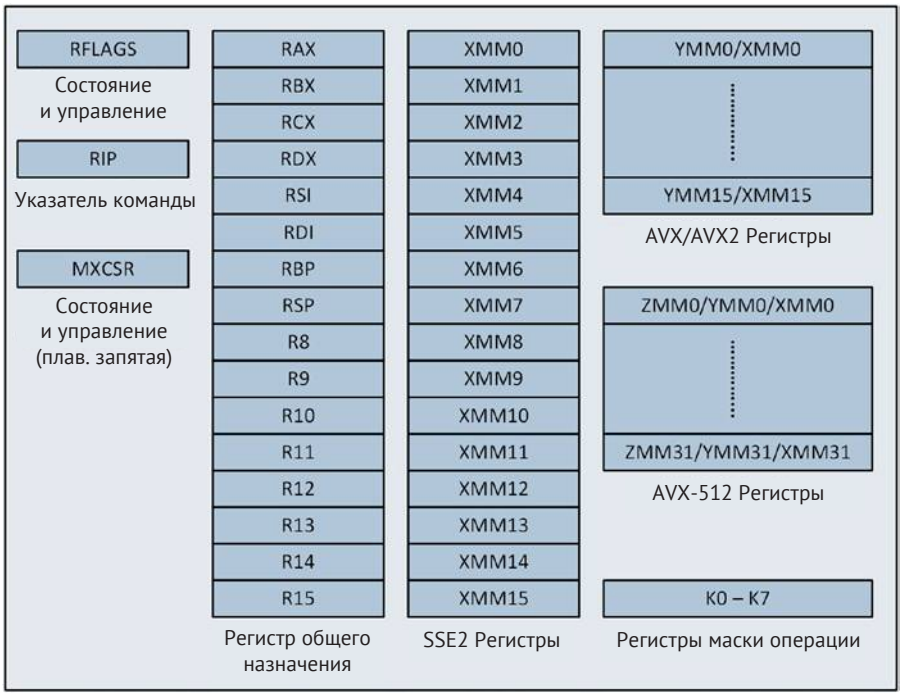


Рис. 1.3. Внутренняя архитектура процессора X86-64

Все процессоры, совместимые с x86-64, поддерживают SSE2 и имеют 16 128-битных регистров XMM, которые программисты могут использовать для выполнения скалярных вычислений с плавающей запятой. Эти регистры также могут использоваться для выполнения операций SIMD с использованием упакованных целых чисел или упакованных значений с плавающей запятой (как одинарной, так и двойной точности). Вы узнаете, как использовать регистры XMM, регистр MXCSR и набор команд AVX для выполнения вычислений с плавающей запятой, в главах 4 и 5. В этой главе также более подробно рассматривается набор регистров YMM и другие архитектурные концепции AVX. Вы узнаете о AVX2 и AVX-512 в главах 8 и 12 соответственно.

1.3.1. Регистры общего назначения

Блок исполнения x86-64 содержит 16 64-битных *регистров общего назначения*, которые используются для выполнения арифметических и логических операций, операций сравнения, передачи данных и вычисления адресов. Их также можно использовать в качестве временных хранилищ для констант, промежуточных результатов и указателей на значения данных, хранящиеся в памяти. На рис. 1.4 показан полный набор регистров общего назначения x86-64 вместе с именами их командных операндов.

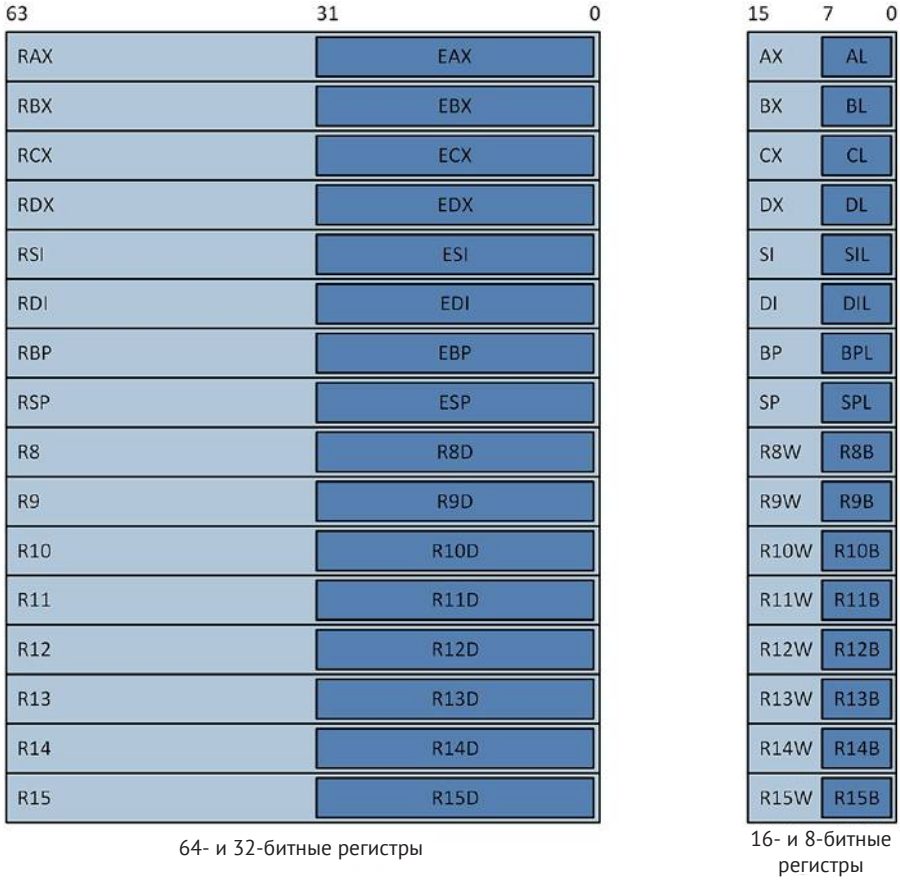


Рис. 1.4. Регистры общего назначения x86-64

Двойное слово младшего разряда, слово и байт каждого 64-битного регистра доступны независимо и могут использоваться для работы с 32-битными, 16-битными и 8-битными операндами. Например, функция может использовать регистры EAX, EBX, ECX и EDX для выполнения 32-битных вычислений в двойных словах младшего разряда регистров RAX, RBX, RCX и RDX соответственно. Точно так же регистры AL, BL, CL и DL могут использоваться для выполнения 8-битных вычислений в младших байтах. Следует отметить, что существует несоответствие в названиях некоторых байтовых регистров. 64-битный ассемблер Microsoft использует имена, показанные на рис. 1.4, а в документации Intel применяются имена R8L–R15L. В этой книге используются имена регистров Microsoft, чтобы поддерживать согласованность между текстом и примерами кода. На рис. 1.4 не показаны устаревшие байтовые регистры AH, BH, CH и DH. Эти регистры связаны со старшими байтами регистров AX, BX, CX и DX соответственно. Устаревшие байтовые регистры могут использоваться в программах x86-64, хотя и с некоторыми ограничениями, как описано далее в этой главе.

Несмотря на название регистров общего назначения, набор команд x86-64 налагает некоторые заметные ограничения на то, как их можно использовать. Некоторые команды явно требуют или неявно используют определенные регистры в качестве операндов. Это устаревший шаблон проектирования, восходящий к 8086, предположительно для повышения плотности кода. Например, некоторые варианты команды `imul` (целочисленное умножение со знаком) сохраняют вычисленное целочисленное произведение в RDX:RAX, EDX:EAX, DX:AX или AX (двоеточие означает, что конечный результат содержится в двух регистрах, где первый регистр содержит старшие биты). Команда `idiv` (целочисленное деление со знаком) требует, чтобы целочисленное делимое было загружено в RDX:RAX, EDX:EAX, DX:AX или AX.

Строковые команды x86 требуют, чтобы адреса операндов источника и приемника были помещены в регистры RSI и RDI соответственно. Строковые команды, содержащие префикс повтора, должны использовать RCX в качестве регистра счетчика, в то время как команды битового и циклического сдвига должны загружать значение счетчика в регистр CL.

Процессор использует регистр RSP для поддержки операций, связанных со стеком, таких как вызовы и возврат функций. Сам *стек* – это просто непрерывный блок памяти, который операционная система назначает процессу или потоку. Прикладные программы также могут использовать стек для передачи аргументов функции и хранения временных данных. Регистр RSP всегда указывает на самый верхний элемент стека. Операции загрузки в стек и выгрузки из стека выполняются с использованием 64-битных операндов. Это означает, что положение стека в памяти обычно выравнивается по 8-байтовой границе. Некоторые среды выполнения (например, 64-битные программы Visual C++, работающие в Windows) выравнивают стековую память и RSP по 16-байтовой границе, чтобы избежать неправильного переноса содержимого памяти между регистрами XMM и 128-битными операндами, хранящимися в стеке.

Хотя технически возможно использовать регистр RSP в качестве регистра общего назначения, такое использование непрактично и настоятельно не рекомендуется. Регистр RBP обычно используется как начальный указатель для доступа к элементам данных, которые хранятся в стеке. RSP также можно использовать в качестве начального указателя для доступа к элементам данных в стеке. Когда он не используется в качестве указателя, программы могут использовать RBP как регистр общего назначения.

1.3.2. Регистр RFLAGS

Регистр RFLAGS содержит ряд *битов состояния* (или *флагов*), которые процессор использует для обозначения результатов арифметической или логической операции, или операции сравнения. Он также содержит ряд управляющих битов, которые в основном используются операционными системами. В табл. 1.4 показана организация битов в регистре RFLAGS.

Таблица 1.4. Регистр RFLAGS

Номер бита	Название	Обозначение	Применение
0	Флаг переноса (Carry Flag)	CF	Состояние
1	Зарезервирован		1
2	Флаг четности (Parity Flag)	PF	Состояние
3	Зарезервирован		0
4	Вспомогательный флаг переноса (Auxiliary Carry Flag)	AF	Состояние
5	Зарезервирован		0
6	Флаг нулевого результата (Zero Flag)	ZF	Состояние
7	Флаг знака (Sign Flag)	SF	Состояние
8	Флаг трассировки (Trap Flag)	TF	Системный
9	Флаг разрешения прерываний (Interrupt Enable Flag)	IF	Системный
10	Флаг направления (Direction Flag)	DF	Управление
11	Флаг переполнения (Overflow Flag)	OF	Состояние
12	Бит 0 уровня привилегии ввода-вывода (I/O Privilege Level Bit 0)	IOPL	Системный
13	Бит 1 уровня привилегии ввода-вывода (I/O Privilege Level Bit 1)	IOPL	Системный
14	Вложенная задача (Nested Task)	NT	Системный
15	Зарезервирован		0
16	Флаг итога (Resume Flag)	RF	Системный
17	Виртуальный режим 8086 (Virtual 8086 Mode)	VM	Системный
18	Проверка выравнивания (Alignment Check)	AC	Системный
19	Флаг виртуального прерывания (Virtual Interrupt Flag)	VIF	Системный
20	Запрос виртуального прерывания (Virtual Interrupt Pending)	VIP	Системный
21	Флаг ID (ID Flag)	ID	Системный
22–63	Зарезервированы		0

Для прикладных программ наиболее важными битами в регистре RFLAGS являются следующие флаги состояния: флаг переноса (CF), флаг переполнения (OF), флаг четности (PF), знак знака (SF) и флаг нулевого результата (ZF). Флаг переноса устанавливается процессором для обозначения состояния переполнения при выполнении беззнаковой целочисленной арифметики. Он также используется некоторыми командами циклического и битового сдвига регистров. Флаг переполнения сигнализирует, что результат целочисленной операции со знаком слишком мал или слишком велик. Процессор устанавливает флаг четности, чтобы указать, содержит ли младший значимый байт арифметической, логической или операции сравнения четное число битов 1 (биты четности используются некоторыми протоколами связи для обнаружения ошибок передачи). Флаги знака и нулевого результата устанавливаются арифметическими и логическими командами для обозначения отрицательного, положительного или нулевого результата.

Регистр RFLAGS содержит управляющий бит, называемый флагом направления (DF). Прикладная программа может устанавливать или сбрасывать флаг направления, который определяет направление автоматического инкремента (0 = адреса от младшего к старшему, 1 = от старшего к младшему) регистров RDI и RSI во время выполнения строковых команд. Остальные биты в регистре RFLAGS используются исключительно операционной системой для управления прерываниями, ограничения операций ввода-вывода, поддержки отладки программ и обработки виртуальных операций. Они никогда не должны изменяться прикладной программой. Зарезервированные биты также никогда не должны изменяться, и не следует делать никаких предположений относительно состояния какого-либо зарезервированного бита.

1.3.3. Указатель команд

Регистр *указателя команд* (RIP) содержит логический адрес следующей команды, которая должна быть выполнена. Значение в регистре RIP обновляется автоматически во время выполнения каждой команды. Он также неявно изменяется во время выполнения команд, связанных с передачей управления. Например, команда `call` (вызов процедуры) помещает содержимое регистра RIP в стек и передает управление программой по адресу, представленному в указанном операнде. Команда `ret` (возврат из процедуры) передает управление программой, извлекая восемь самых верхних байтов из стека и загружая их в регистр RIP.

Команды `jmp` (переход) и `jcc` (переход, если условие выполнено) также передают управление программой, изменяя содержимое регистра RIP. В отличие от команд `call` и `ret`, все команды перехода x86-64 выполняются независимо от стека. Регистр RIP также используется для относительной адресации памяти операндов, как описано в следующем разделе. Выполняемая задача не может напрямую обращаться к содержимому регистра RIP.

1.3.4. Операнды команд

Все команды x86-64 используют операнды, обозначающие конкретные значения, с которыми будет работать команда. Почти все команды требуют наличия одного или нескольких операндов-*источников* вместе с одним операндом-*приемником*. Большинство команд также требуют, чтобы программист явно указал операнды источника и приемника. Однако существует ряд команд, в которых регистровые операнды либо указаны неявно, либо зависят от команды, как обсуждалось в предыдущем разделе.

Есть три основных типа операндов: непосредственные, регистровые и хранимые (запоминаемые). *Непосредственный операнд* – это постоянное значение, которое кодируется как часть команды. Обычно они используются для указания постоянных значений. Непосредственное значение можно присвоить только операнду источника. *Регистровые операнды* содержатся в регистре общего назначения или регистре SIMD. *Хранимый операнд* определяет место в памяти, где может храниться любой из типов данных, описанных ранее в этой главе. Хранимым операндом команды может быть либо исходный, либо целевой операнд, но не оба вместе. Таблица 1.5 содержит несколько примеров команд, которые используют различные типы операндов.

Таблица 1.5. Примеры основных типов операндов

Тип	Пример	Аналогичный оператор C/C++
Непосредственный	`mov rax, 42`	`rax = 42`
	`imul r12,-47`	`r12 *= -47`
	`shl r15,8`	`r15 <<= 8`
	`xor ecx,80000000h`	`ecx ^= 0x80000000`
	`sub r9b,14`	`r9b -= 14`
Регистровый	`mov rax,rbx`	`rax = rbx`
	`add rbx,r10`	`rbx += r10`
	`mul rbx`	`rdx:rax = rax * rbx`
	`and r8w,0ff00h`	`r8w &= oxffoo`
Хранимый	`mov rax, [r13]`	`rax = *r13`
	`or rcx,[rbx+rsi*8]`	`rcx |= *(rbx+rsi*8)`
	`sub qword ptr [r8],17`	`*(long long*)r8 -= 17`
	`shl word ptr [r12],2`	`*(short*)r12 <<= 2`

Команда `mul rbx` (умножение без знака), показанная в табл. 1.5, является примером неявного использования операнда. В этом примере неявный регистр RAX и явный регистр RBX используются в качестве исходных операндов, а неявная пара регистров RDX:RAX является операндом назначения. Старшие и младшие четверные слова произведения хранятся в RDX и RAX соответственно.

В предпоследнем примере табл. 1.5 `qword ptr` является оператором ассемблера, который действует как оператор приведения в C/C++. В этом случае значение 17 вычитается из 64-битного значения, расположение которого в памяти определяется содержимым регистра R8. Без оператора `qword ptr` выражение на языке ассемблера неоднозначно, поскольку ассемблер не может определить размер операнда, на который указывает R8. В этом примере приемником также может быть операнд размером 8, 16 или 32 бит. В последнем примере в табл. 1.5 оператор `word ptr` работает аналогичным образом. Вы узнаете больше об операторах и директивах ассемблера в следующих главах этой книги, посвященных программированию.

1.3.5. Адресация памяти

Для команды x86-64 требуется до четырех отдельных компонентов, чтобы указать расположение операнда в памяти. Четыре компонента включают в себя постоянное значение смещения, базовый регистр, индексный регистр и масштабный коэффициент. Используя эти компоненты, процессор вычисляет эффективный адрес для операнда памяти следующим образом:

```
EffectiveAddress = BaseReg + IndexReg * ScaleFactor + Disp
```

Базовый регистр (`BaseReg`) может быть любым регистром общего назначения. Индексный регистр (`IndexReg`) может быть любым регистром общего назначения, кроме RSP. Допустимые значения коэффициента масштабирования (`ScaleFactor`) включают 2, 4 и 8. Наконец, смещение (`Disp`) представляет собой постоянное 8-битное, 16-битное или 32-битное смещение со знаком, закодированное в команде. Таблица 1.6 иллюстрирует адресацию памяти x86-64 с использованием различных форм команды `mov`. В этих примерах регистр RAX (операнд-приемник) загружается со значением четверного слова, в соответствии со значением операнда-источника.

Обратите внимание, что в команде не обязательно явно указывать все компоненты, необходимые для вычисления действующего адреса. Например, для смещения используется значение по умолчанию, равное нулю, если не указано явное значение. Окончательный размер вычисленного адреса всегда составляет 64 бита.

Таблица 1.6. Адресация памяти операндов

Способ адресации	Пример
`RIP + Disp mov rax,[Val]`	`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]`

Способы адресации памяти, показанные в табл. 1.6, используются для прямого обращения к программным переменным и структурам данных. Например, простое смещение часто используется для доступа к простой глобальной или статической переменной. Использование базового регистра аналогично использованию указателя C/C++ и применяется для косвенной ссылки на одно значение. Доступ к отдельным полям в структуре данных можно получить с помощью базового регистра и смещения. Использование индексного регистра удобно для доступа к отдельным элементам в массиве. Коэффициенты масштабирования способствуют уменьшению объема кода, необходимого для доступа к элементам массива, содержащего целые числа или значения с плавающей запятой. На элементы в более сложных структурах данных можно ссылаться, используя базовый регистр вместе с индексным регистром, коэффициентом масштабирования и смещением.

Команда `mov rax,[Val]`, показанная в первой строке табл. 1.6, является примером адресации относительно RIP (или относительно указателя команд). При адресации относительно RIP процессор вычисляет эффективный адрес, используя содержимое регистра RIP и 32-битное значение смещения со знаком, которое закодировано в команды. Рисунок 1.5 более подробно иллюстрирует это вычисление. Обратите внимание на порядок следования байтов смещения, встроенного в команду `mov rax,[Val]`. Адресация относительно RIP позволяет процессору ссылаться на глобальные или статические операнды, используя 32-битное смещение вместо 64-битного смещения, что уменьшает пространство кода. Это также упрощает позиционно-независимый код.

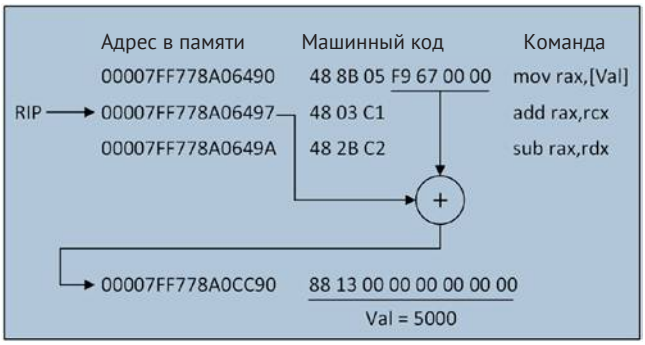


Рис. 1.5. Вычисление эффективного адреса относительно RIP

Одно незначительное ограничение относительной адресации RIP состоит в том, что операнд-приемник должен находиться в адресном окне ±2 ГБ относительно значения в регистре RIP. Для большинства программ это условие редко имеет значение. Вычисление смещения относительно RIP автоматически выполняется ассемблером во время генерации кода. Это означает, что вы можете использовать `mov rax,[Val]` или аналогичные команды, не беспокоясь о деталях вычисления смещения.

1.4. Различия между программированием x86-64 и x86-32

Между программированием на ассемблере x86-64 и x86-32 есть некоторые важные различия. Если вы впервые изучаете программирование на ассемблере x86, вы можете либо бегло просмотреть, либо пропустить этот раздел, поскольку в нем упоминаются концепции, которые не будут полностью рассмотрены в этой книге.

Большинство команд x86-32 имеют эквивалентную команду x86-64, которая позволяет использовать 64-разрядные адреса и операнды. Функции x86-64 также могут выполнять вычисления с использованием команд, управляющих 8-битными, 16-битными или 32-битными регистрами и операндами. За исключением команды `mov`, максимальный размер непосредственного значения в режиме x86-64 составляет 32 бита. Если команда манипулирует 64-битным регистром или операндом памяти, любое указанное 32-битное непосредственное значение перед использованием расширяется до 64 бит со знаком.

В табл. 1.7 представлены несколько примеров команд x86-64 с использованием операндов разного размера. Обратите внимание, что для обращения к операндам памяти в этих примерах команд используются 64-битные регистры, которые необходимы для доступа ко всему 64-битному линейному адресному пространству. Хотя в режиме x86-64 можно ссылаться на операнд памяти с помощью 32-разрядного регистра (например, `mov r10,[eax]`), операнд должен находиться в нижней 4-гигабайтной части 64-разрядного адресного пространства. Использование 32-битных регистров для доступа к операндам памяти в режиме x86-64 не рекомендуется, поскольку это вводит ненужные и потенциально опасные обфускации кода. Это также усложняет тестирование и отладку программного обеспечения.

Таблица 1.7. Примеры команд x86-64 с использованием операндов разного размера

8-Bit	16-Bit	32-Bit	64-Bit
`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],al`	`or [r8+rdi],ax`	`or [r8+rdi],eax`	`or [r8+rdi],rax`
`shl r9b,cl`	`shl r9w,cl`	`shl r9d,cl`	`shl r9,cl`

Вышеупомянутое ограничение размера непосредственного значения требует некоторого дополнительного обсуждения, поскольку иногда оно влияет на последовательности команд, которые программа должна использовать для выполнения определенных операций. На рис. 1.6 приведено несколько примеров команд, использующих 64-битный регистр с непосредственным операндом. В первом примере команда `mov rax,100` загружает непосредственное значение в регистр RAX. Обратите внимание, что машинный код использует только 32 бита для кодирования непосредственного значения `100` (подчеркнуто на рисунке). Это значение расширяется до 64 бит со знаком и сохраняется

в RAX. Следующая команда `add rax,200` также расширяет со знаком свое непосредственное значение, перед тем как произойдет сложение. Следующий пример начинается с команды `mov rcx,-2000`, которая загружает отрицательное непосредственное значение в RCX. Машинный код для этой команды также использует 32 бита для кодирования непосредственного значения `-2000`, которое расширяется до 64 разрядов со знаком и сохраняется в RCX. Последующая команда `add rcx,1000` дает 64-битный результат `-1000`.

Машинный код	Команда	DesOp Результат
48 C7 C0 64 00 00 00	mov rax,100	0000000000000064h
48 05 C0 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

Рис. 1.6. Использование 64-битных регистров с непосредственными операндами

В третьем примере для инициализации регистра RDX используется команда `mov rdx,0ffh`. Далее следует команда `or rdx,80000000h`, которая расширяет со знаком непосредственное значение `0x80000000` до `0xFFFFFFFF80000000`, а затем выполняет побитовую операцию включающего ИЛИ. Значение, оказавшееся в RDX, скорее всего, не является результатом, которого вы ожидали. Последний пример показывает, как выполнить операцию, которая требует указать непосредственное 64-битное значение. Команда `mov r8,80000000h` загружает 64-битное значение `0x0000000080000000` в R8. Как упоминалось ранее в этом разделе, команда `mov` – единственная команда, которая поддерживает 64-битные непосредственные операнды. Выполнение следующей команды `or rdx,r8` дает ожидаемое значение.

Ограничение непосредственных значений 32-битным размером также применяется к командам `jmp` и `call`, которые определяют цели относительного смещения. В этих случаях цель (или местоположение) команды `jmp` или `call` должна находиться в адресном окне ± 2 ГБ относительно текущего значения регистра RIP. К приемникам, относительные смещения которых превышают это окно, можно получить доступ только с помощью команд перехода `jmp` или `call`, которые используют косвенный операнд (например, `jmp qword ptr[FuncPtr]` или `call rax`). Как и относительная адресация RIP, ограничения размера, описанные в этом параграфе, вряд ли станут серьезными препятствиями для большинства функций языка ассемблера.

Еще одно различие между программированием на ассемблере x86-32 и x86-64 заключается в том, что некоторые команды влияют на верхние 32 бита 64-битного

регистра общего назначения. При использовании команд, оперирующих 32-битными регистрами и операндами, старшие 32 бита соответствующего 64-битного регистра общего назначения обнуляются во время выполнения. Например, предположим, что регистр RAX содержит значение 0x8000000000000000. Выполнение команды `add eax,10` генерирует результат `0x000000000000000A` в RAX. Однако при работе с 8-битными или 16-битными регистрами и операндами старшие 56 или 48 бит соответствующего 64-битного регистра общего назначения не изменяются. Снова предположим, RAX содержит значение `0x8000000000000000`, тогда выполнение команд `add al,20` или `add ax,40` даст значения RAX `0x8000000000000014` или `0x8000000000000028` соответственно.

Платформа x86-64 накладывает некоторые ограничения на использование устаревших регистров AH, BH, CH и DH. Эти регистры нельзя использовать с командами, которые одновременно ссылаются на один из новых 8-битных регистров (например, SIL, DIL, BPL, SPL и R8B – 15B). Существующие команды x86-32, такие как `mov ah,bl` и `add dh,bl`, по-прежнему разрешены в программах на языке ассемблера x86-64. Однако команды `mov ah,r8b` и `add dh,r8b` устарели.

1.4.1. Недопустимые команды

Некоторые редко используемые команды x86-32 нельзя использовать в программах x86-64. В табл. 1.8 перечислены эти команды. Несколько удивительно, что процессоры x86-64 раннего поколения не поддерживали команды `lahf` и `sahf` в режиме x86-64 (они все еще работали в режиме x86-32). К счастью, эти команды были восстановлены в правах и должны быть доступны в большинстве процессоров AMD и Intel, продаваемых с 2006 года. Программа может проверить поддержку процессором команд `lahf` и `sahf` в режиме x86-64, протестировав флаг `LAHF-SAHF`.

Таблица 1.8. Недопустимые команды в режиме X86-64

Команда	Название
`aaa`	ASCII Adjust After Addition (ASCII коррекция после сложения)
`aad`	ASCII Adjust After Division (ASCII коррекция после деления)
`aam`	ASCII Adjust After Multiplication (ASCII коррекция после умножения)
`aas`	ASCII Adjust After Subtraction (ASCII коррекция после вычитания)
`bound`	Check Array Index Against Bounds (проверка значения индекса массива на допустимость)
`daa`	Decimal Adjust After Addition (десятичная коррекция после сложения)
`das`	Decimal Adjust After Subtraction (десятичная коррекция после вычитания)
`into`	Generate interrupt if RFLAGS.OF Equals 1 (генерация прерывания, если RFLAGS.OF=1)
`pop[a|ad]`	Pop all General-Purpose Registers (выгрузка из стека всех регистров общего пользования)
`push[a|ad]`	Push all General-Purpose Registers (загрузка в стек всех регистров общего пользования)

1.4.2. Устаревшие команды

Процессоры, поддерживающие набор команд x86-64, также поддерживают вычислительные ресурсы SSE2. Это означает, что программы x86-64 могут безопасно использовать для работы с упакованными целыми числами команды SSE2 вместо MMX. Это также означает, что программы x86-64 могут использовать скалярные команды с плавающей запятой SSE2 (или AVX, если они доступны) вместо команд FPU x87. Программы X86-64 могут по-прежнему использовать наборы команд MMX и x87 FPU, и такое использование может быть оправдано при миграции устаревшего кода x86-32 на платформу x64-64. Однако для разработки нового программного обеспечения x86-64 использование наборов команд MMX и x87 FPU не рекомендуется.

1.5. Обзор набора команд

В табл. 1.9 перечислены в алфавитном порядке основные команды x86-64, которые часто используются в функциях на языке ассемблера. Для каждой мнемоники команды приведено намеренно краткое описание, поскольку детальные описания каждой команды, включая особенности выполнения, допустимые операнды, затронутые флаги и исключения, можно без труда найти в справочных руководствах, опубликованных AMD и Intel. Приложение содержит список этих руководств. Примеры программирования в главах 2 и 3 также содержат дополнительную информацию о правильном использовании этих команд.

Обратите внимание, что в табл. 1.9 в столбце мнемоники используются квадратные скобки для обозначения различных вариантов общей команды. Например, `bs[f|r]` обозначает отдельные команды `bsf` (битовое сканирование вперед) и `bsr` (битовое сканирование в обратном направлении).

Таблица 1.9. Обзор набора команд x86-64

Мнемоническая запись	Назначение команды
`adc`	Сложение с переносом
`add`	Сложение
`and`	Побитовое логическое И
`bs[f|r]`	Проверка бита в прямом|обратном направлении
`b[t|tr|ts]`	Проверка | проверка и сброс | проверка и установка бита
`call`	Вызов подпрограммы
`cld`	Сброс флага направления (RFLAGS.DF)
`cmovcc`	Условное присваивание
`cmp`	Сравнение операндов
`cmpsfb|w|d|q]`	Сравнение строковых операндов
`cpuid`	Запрос идентификатора CPU и его функций
`c[wd|dq|do]`	Конвертация операнда

Продолжение табл. 1.9

Мнемоническая запись	Назначение команды
`dec`	Декремент операнда на 1
`div`	Беззнаковое целочисленное деление
`idiv`	Знаковое целочисленное деление
`imul`	Знаковое целочисленное умножение
`inc`	Инкремент операнда на 1
`jcc`	Условный переход
`jmp`	Безусловный переход
`lahf`	Загрузка флагов состояния в регистр AH
`lea`	Загрузка эффективного адреса
`lodsfb|w|d|q]`	Загрузка строкового операнда
`mov`	Присваивание
`mov[sx|sxd]`	Присваивание и расширение с учетом знака
`movzx`	Присваивание и расширение нулевым значением
`mul`	Беззнаковое целочисленное умножение
`neg`	Смена знака
`not`	Побитовое логическое НЕ
`or`	Побитовое логическое ИЛИ
`pop`	Выгрузка значения из стека в операнд
`popfq`	Выгрузка из стека в RFLAGS
`push`	Загрузка операнда в стек
`pushfq`	Загрузка RFLAGS в стек
`rc[l|r]`	Вращение влево|вправо через флаг переноса RFLAGS.CF
`ret`	Возврат из подпрограммы
`re[p|pe|pz|pne|pnz]`	Префикс повторения строковых операций
`ro[l|r]`	Вращение влево|вправо
`sahf`	Запись AH в флаги
`sar`	Арифметический сдвиг вправо
`setcc`	Установить байт по условию
`sh[l|r]`	Логический сдвиг влево|вправо
`sbb`	Вычитание с заемом
`std`	Установка флага направления RFLAGS.DF

Окончание табл. 1.9

Мнемоническая запись	Назначение команды
`stos[b|w|d|q]`	Запись строки
`test`	Логическое сравнение (устанавливает флаги состояния)
`xchg`	Обмен значениями между операндами
`xor`	Побитовое логическое исключающее ИЛИ

Большинство арифметических и логических команд обновляют один или несколько флагов состояния в регистре RFLAGS. Как было сказано ранее в этой главе, флаги состояния предоставляют дополнительную информацию о результатах операции. Команды `jcc`, `cmovcc` и `setcc` используют так называемые коды условий для проверки флагов состояния по отдельности или в виде комбинации нескольких флагов. В табл. 1.10 перечислены коды условий, мнемонические суффиксы и соответствующие флаги RFLAGS, проверяемые с помощью этих команд.

Таблица 1.10. Коды условий, мнемонические суффиксы и проверяемые условия

Условный код	Мнемоника	Флаги RFLAGS
Выше Не ниже и не равно	A NBE	`CF == 0 && ZF == 0`
Выше или равно Не ниже	AE NB	`CF == 0`
Ниже Не выше и не равно	B NAE	`CF == 1`
Ниже или равно Не выше	BE NA	`CF == 1 || ZF == 1`
Равно Ноль	E Z	`ZF == 1`
Не равно Не ноль	NE NZ	`ZF == 0`
Больше Не меньше и не равно	G NLE	`ZF == 0 && SF == OF`
Меньше Не больше и не равно	GE NL	`SF == OF`
Меньше Не больше и не равно	L NGE	`SF != OF`
Меньше или равно Не больше	LE NG	`ZF == 1 || SF != OF`
Знаковое	S	`SF == 1`
Беззнаковое	NS	`SF == 0`
Перенос	C	`CF == 1`
Нет переноса	NC	`CF == 0`

Окончание табл. 1.10

Условный код	Мнемоника	Флаги RFLAGS
Переполнение	O	`OF == 1`
Нет переполнения	NO	`OF == 0`
Четность Четный паритет	P PE	`PF == 1`
Нет признака четности Нечетный паритет	NP PO	`PF == 0`

Альтернативные формы многих мнемонических суффиксов в табл. 1.10 приведены для обеспечения алгоритмической гибкости или улучшения читаемости программы. Коды условий, содержащие в описании слова «выше» (above) и «ниже» (below), используются для целочисленных операндов без знака, тогда как содержащие в описании слова «больше» (greater) и «меньше» (less) используются для целочисленных операндов со знаком. Если содержимое табл. 1.10 кажется немного запутанным или абстрактным, не волнуйтесь. Вы увидите множество примеров условного кода в следующих главах этой книги.

1.6. Заключение

В главе 1 рассмотрены следующие ключевые моменты.

- Основные типы данных платформы x86-64 включают байты, слова, двойные слова, четверные слова и двойные четверные слова. Типы данных внутреннего языка программирования, такие как символы, текстовые строки, целые числа и значения с плавающей запятой, являются производными от основных типов данных.
- Исполнительный блок x86-64 включает 16 64-битных регистров общего назначения, которые используются для выполнения арифметических, логических операций и операций передачи данных с использованием 8-битных, 16-битных, 32-битных и 64-битных операндов.
- Исполнительный блок x86-64 включает 16 128-битных регистров XMM, которые можно использовать для выполнения скалярных арифметических операций с плавающей запятой с использованием значений с одинарной или двойной точностью. Эти регистры также могут использоваться для выполнения операций SIMD с использованием упакованных целых чисел или упакованных значений с плавающей запятой.
- Большинство команд на языке ассемблера x86-64 можно использовать со следующими явными типами операндов: немедленный, регистр и память. Некоторые команды используют в качестве операндов неявные регистры.
- На операнд в памяти можно ссылаться, используя различные режимы адресации, которые включают один или несколько из следующих компонентов: фиксированное смещение, базовый регистр, индексный регистр и/или масштабный коэффициент.
- Большинство арифметических и логических команд обновляют один или несколько флагов состояния в регистре RFLAGS. Эти флаги можно протестировать, чтобы изменить ход выполнения программы или условно присвоить значения переменным.

Глава 2

Программирование ядра x86-64. Часть 1

В предыдущей главе вы познакомились с основными характеристиками платформы x86-64, включая ее типы данных, наборы регистров, режимы адресации памяти и основной набор команд. В этой главе вы узнаете, как кодировать базовые функции на языке ассемблера x86-64, которые можно вызывать из C++. Вы также узнаете о семантике и синтаксисе файла исходного кода ассемблера x86-64. Образец исходного кода и сопроводительные примечания к этой главе предназначены для дополнения ознакомительного материала, представленного в главе 1.

Содержание главы 2 организовано следующим образом. В первом разделе показано, как программировать функции, выполняющие простые целочисленные арифметические операции, такие как сложение и вычитание. Вы также узнаете основы передачи аргументов и возвращаемых значений между функциями, написанными на C++ и языке ассемблера x86-64. В следующем разделе рассматриваются дополнительные арифметические команды, включая целочисленное умножение и деление. В последнем разделе вы узнаете, как ссылаться на операнды в памяти и использовать условные переходы и условные присвоения.

Следует отметить, что основная цель примеров кода, представленного в этой главе, – продемонстрировать правильное использование набора команд x86-64 и основных методов программирования на языке ассемблера. Весь код на языке ассемблера прост, но не обязательно оптимален, поскольку понимание оптимизированного кода на языке ассемблера может быть сложной задачей, особенно для начинающих. В примерах кода, которые обсуждаются в последующих главах, эффективным методам кодирования уделяется больше внимания. В главе 15 также рассматриваются методы, которые можно использовать для повышения эффективности ассемблерного кода.

2.1. Простая целочисленная арифметика

В этом разделе вы изучите основы программирования на языке ассемблера x86-64. Он начинается с простой программы, демонстрирующей, как выполнять сложение и вычитание целых чисел. Далее следует пример программы, иллюстрирующий использование логических команд `and`, `or` и `xor`. Последняя программа демонстрирует выполнение операции сдвига. Все три программы иллюстрируют передачу аргументов и возвращаемых значений между функцией C++ и модулем на языке ассемблера. Они также демонстрируют использование наиболее востребованных директив ассемблера.

Как упоминалось во введении, весь пример кода, обсуждаемый в этой книге, был создан с использованием Microsoft Visual C++ и Macro Assembler (MASM), которые входят в состав пакета Visual Studio. Перед тем как взглянуть на первый пример кода, стоит узнать несколько полезных фактов об этих инструментах разработки. Чтобы упростить разработку приложений, Visual Studio использует сущности, называемые *решениями* (solution) и *проектами* (project). Решение – это набор из одного или нескольких проектов, которые используются для создания приложения. Проекты – это объекты-контейнеры, которые помогают организовать файлы приложения, включая (но не ограничиваясь) исходный код, ресурсы, иконки, растровые изображения, HTML и XML. Проект Visual Studio обычно создается для каждого собираемого компонента (например, исполняемого файла, динамически подключаемой библиотеки, статической библиотеки и т. д.) в составе приложения. Вы можете открыть и загрузить примеры кода главы в среду разработки Visual Studio, дважды щелкнув на файле решения (.sln). Приложение содержит дополнительную информацию об использовании Visual C++ и MASM.

Примечание. Все примеры исходного кода в этой книге содержат одну или несколько функций, написанных на языке ассемблера x86-64, а также дополнительный код на языке C++, демонстрирующий, как вызвать функции на языке ассемблера. Код C++ также содержит вспомогательные функции, которые выполняют необходимые инициализации и отображают результаты. Для каждого примера исходного кода используется один листинг, который включает исходный код как на C++, так и на языке ассемблера, чтобы минимизировать количество ссылок на листинги в основном тексте. Фактический исходный код использует *отдельные* файлы для кода C++ (.cpp) и на языке ассемблера (.asm).

2.1.1. Сложение и вычитание

Первый пример исходного кода в этой главе называется `Ch02_01`. В данном примере показано, как использовать команды на языке ассемблера x86-64 `add` (сложение целых чисел) и `sub` (вычитание целых чисел). Он также иллюстрирует некоторые базовые концепции программирования на языке ассемблера, включая передачу аргументов, возвращение значений и использование нескольких директив ассемблера MASM. В листинге 2.1 показан исходный код примера `Ch02_01`.

Листинг 2.1. Пример Ch02_01

```
//-----------------------------------------------
//                Ch02_01.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>

using namespace std;

extern "C" int IntegerAddSub_(int a, int b, int c, int d);

static void PrintResult(const char* msg, int a, int b, int c, int d, int result)
{
    const char nl = '\n';

    cout << msg << nl;
    cout << "a = " << a << nl;
    cout << "b = " << b << nl;
    cout << "c = " << c << nl;
    cout << "d = " << d << nl;
    cout << "результат = " << result << nl;
    cout << nl;
}

int main()
{
    int a, b, c, d, result;

    a = 10; b = 20; c = 30; d = 18;
    result = IntegerAddSub_(a, b, c, d);
    PrintResult("Пример 1", a, b, c, d, result);

    a = 101; b = 34; c = -190; d = 25;
    result = IntegerAddSub_(a, b, c, d);
    PrintResult("Пример 2", a, b, c, d, result);
    return 0;
}

;-----------------------------------------------
;               Ch02_01.asm
;-----------------------------------------------

; extern "C" int IntegerAddSub_(int a, int b, int c, int d);
        .code
IntegerAddSub_ proc

; Вычисление a + b + c - d
        mov eax,ecx         ;eax = a
        add eax,edx         ;eax = a + b
        add eax,r8d         ;eax = a + b + c
        sub eax,r9d         ;eax = a + b + c - d

        ret                 ;отправка результата вызывающей функции
IntegerAddSub_ endp
        end
```

Код C++ в листинге 2.1 в основном прост, но содержит несколько строк, требующих пояснений. Оператор `#include "stdafx.h"` определяет файл заголовка для конкретного проекта, который содержит ссылки на часто используемые системные элементы. Visual Studio автоматически создает этот файл при создании нового проекта C++. Строка `extern "C" int IntegerAddSub_(int a, int b, int c, int d)` представляет собой оператор объявления, который определяет параметры и возвращаемое значение для функции на языке ассемблера x86-64 `IntegerAddSub_` (все имена функций на языке ассемблера и общедоступные переменные, использованные в этой книге, содержат подчеркивание в конце для лучшей узнаваемости). Модификатор `"C"` в этой строке объявления указывает компилятору C++ использовать именование в стиле C для функции `IntegerAddSub_` вместо расширенного имени C++ (расширенное имя C++ включает дополнительные символы, которые помогают поддерживать перегрузку функции). Он также уведомляет компилятор об использовании связывания в стиле C для указанной функции.

Функция `main` в файле C++ содержит код, вызывающий функцию на языке ассемблера `IntegerAddSub_`. Эта функция ожидает четыре аргумента типа `int` и возвращает одно значение типа `int`. Как и многие языки программирования, Visual C++ использует комбинацию регистров процессора и стека для передачи значений аргументов функции. В данном примере компилятор C++ генерирует код, который загружает значения `a`, `b`, `c` и `d` в регистры ECX, EDX, R8D и R9D соответственно до вызова функции `IntegerAddSub_`.

В листинге 2.1 код на языке ассемблера x86-64 следует сразу после функции C++ `main`. В первую очередь обратите внимание на строки, начинающиеся с точки с запятой. Это строки комментариев. MASM рассматривает любой текст, следующий за точкой с запятой, как текст комментария. Оператор `.code` – это директива MASM, которая определяет начало раздела кода на языке ассемблера. Директива MASM – это команда, которая указывает ассемблеру, как выполнять определенные действия. В этой книге вы узнаете, как использовать дополнительные директивы.

Оператор `IntegerAddSub_ proc` определяет начало функции на языке ассемблера. В конце листинга 2.1 оператор `IntegerAddSub_ endp` определяет конец функции. Как и строка `.code`, операторы `proc` и `endp` являются не исполняемыми командами, а директивами ассемблера, которые обозначают начало и конец функции на языке ассемблера. Завершающий оператор `end` – это обязательная директива ассемблера, указывающая на завершение операторов для файла на языке ассемблера. Ассемблер игнорирует любой текст, который присутствует после директивы `end`.

Функция на языке ассемблера `IntegerAddSub_` вычисляет значение выражения `a+b+c-d` и возвращает это значение в вызывающую функцию C++. Она начинается с команды `mov eax, ecx (Move)`, которая копирует значение `a` из ECX в EAX. Обратите внимание, что команда `mov` не изменяет содержимое регистра ECX. После выполнения этой команды `mov` регистры EAX и ECX содержат значение `a`. Команда `add eax,edx` складывает значения в регистрах EAX и EDX. Затем она сохраняет сумму (т. е. `a+b`) в регистре EAX. Как и в случае предыдущей команды `mov`, содержимое регистра EDX не меняется после выполнения команды сложения. Следующая команда `add eax,r8d` вычисляет значение `a+b+c`. Затем идет команда `sub eax,r9d`, которая вычисляет окончательное значение `a+b+c-d`.

Функция на языке ассемблера x86-64 должна использовать регистр EAX для возвращения одного 32-битного целочисленного (или int в C++) значения вызывающей функции. В текущем примере для выполнения этого условия не требуется никаких дополнительных команд, поскольку EAX уже содержит правильное возвращаемое значение. Последняя команда ret (возврат из процедуры) передает управление обратно вызывающей функции main, которая отображает результат. Ниже показан результат выполнения кода примера Ch02_01:

```
Пример 1
a = 10
b = 20
c = 30
d = 18
результат = 42

Пример 2
a = 101
b = 34
c = -190
d = 25
результат = -80
```

2.1.2. Логические операции

Следующий пример исходного кода называется Ch02_02. Этот пример иллюстрирует использование команд x86-64 and (логическое И), or (логическое включающее ИЛИ) и xor (логическое исключающее ИЛИ). Он также показывает, как получить доступ к глобальной переменной C++ из функции на языке ассемблера. В листинге 2.2 показан исходный код примера Ch02_02.

Листинг 2.2. Пример Ch02_02

```
//-----------------------------------------------
//               Ch02_02.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" unsigned int IntegerLogical_(unsigned int a, unsigned int b, unsigned int c,
unsigned int d);

extern "C" unsigned int g_Val1 = 0;

unsigned int IntegerLogicalCpp(unsigned int a, unsigned int b, unsigned int c, unsigned int
d)
{
    // Вычисление (((a & b) | c ) ^ d) + g_Val1
    unsigned int t1 = a & b;
    unsigned int t2 = t1 | c;
```

```
    unsigned int t3 = t2 ^ d;
    unsigned int result = t3 + g_Val1;
    return result;
}

void PrintResult(const char* s, unsigned int a, unsigned int b, unsigned int c, unsigned
int
d, unsigned val1, unsigned int r1, unsigned int r2)
{
    const int w = 8;
    const char nl = '\n';

    cout << s << nl;
    cout << setfill('0');
    cout << "a = 0x" << hex << setw(w) << a << " (" << dec << a << ")" << nl;
    cout << "b = 0x" << hex << setw(w) << b << " (" << dec << b << ")" << nl;
    cout << "c = 0x" << hex << setw(w) << c << " (" << dec << c << ")" << nl;
    cout << "d = 0x" << hex << setw(w) << d << " (" << dec << d << ")" << nl;
    cout << "val1 = 0x" << hex << setw(w) << val1 << " (" << dec << val1<< ")" << nl;
    cout << "r1 = 0x" << hex << setw(w) << r1 << " (" << dec << r1 << ")" << nl;
    cout << "r2 = 0x" << hex << setw(w) << r2 << " (" << dec << r2 << ")" << nl;
    cout << nl;

    if (r1 != r2)
        cout << "Ошибка сравнения" << nl;
}

int main()
{
    unsigned int a, b, c, d, r1, r2 = 0;
    a = 0x00223344;
    b = 0x00775544;
    c = 0x00555555;
    d = 0x00998877;
    g_Val1 = 7;
    r1 = IntegerLogicalCpp(a, b, c, d);
    r2 = IntegerLogical_(a, b, c, d);
    PrintResult("Пример 1", a, b, c, d, g_Val1, r1, r2);

    a = 0x70987655;
    b = 0x55555555;
    c = 0xAAAAAAAA;
    d = 0x12345678;
    g_Val1 = 23;
    r1 = IntegerLogicalCpp(a, b, c, d);

    r2 = IntegerLogical_(a, b, c, d);
    PrintResult("Пример 2", a, b, c, d, g_Val1, r1, r2);
    return 0;
}

;-----------------------------------------------
; Ch02_02.asm
;-----------------------------------------------
; extern "C" unsigned int IntegerLogical_(unsigned int a, unsigned int b, unsigned int c,
unsigned int d);
```

```
        extern g_Val1:dword ;внешнее значение doubleword (32-бит)

        .code
IntegerLogical_ proc

; Вычисление (((a & b) | c ) ^ d) + g_Val1
        and ecx,edx       ;ecx = a & b
        or ecx,r8d        ;ecx = (a & b) | c
        xor ecx,r9d       ;ecx = ((a & b) | c) ^ d
        add ecx,[g_Val1]  ;ecx = (((a & b) | c) ^ d) + g_Val1

        mov eax,ecx       ;eax = окончательный результат
        ret               ;возвращает результат вызывающей функции
IntegerLogical_ endp
        end
```

Здесь, как и в первом примере, объявление функции на языке ассемблера `IntegerLogical_` использует модификатор `"C"`, чтобы указать компилятору C++ не генерировать расширенное имя для этой функции. Отсутствие данного модификатора приведет к ошибке линкера во время сборки программы. (Если модификатор `"C"` в текущем примере убрать, Visual C++ 2017 сгенерирует расширенное имя функции `?IntegerLogical  _@@YAIIIII@Z` вместо `IntegerLogical_`. Расширенные имена выводятся с использованием типов аргументов функции, и эти имена зависят от компилятора.) Функция `IntegerLogical_` ожидает четыре аргумента `unsigned int` и возвращает один результат `unsigned int`. Сразу после объявления функции `IntegerLogical_` следует определение глобальной переменной типа `unsigned int` с именем `g_Val1`. Эта переменная определена для демонстрации доступа к глобальному значению из функции на языке ассемблера. Как и объявления функций, использование модификатора `"C"` для `g_Val1` указывает компилятору использовать именование в стиле C вместо расширенного имени C++.

Далее в исходном коде C++ следует определение функции `IntegerLogicalCpp`. Назначение этой функции заключается в проверке того, вычисляет ли соответствующая функция на языке ассемблера x86-64 `IntegerLogical_` правильный результат. Хотя в этом конкретном примере такая проверка на самом деле не нужна, параллельное кодирование сложных функций с использованием как C++, так и ассемблера часто бывает полезным для тестирования и отладки программного обеспечения. Функция `main` в листинге 2.2 содержит код, который вызывает как `IntegerLogicalCpp`, так и `IntegerLogical_`. Он также вызывает функцию `PrintResult` для отображения результатов.

В листинге 2.2 код на языке ассемблера x86-64 следует сразу после функции C++ `main`. Первый оператор исходного кода на языке ассемблера, `extern g_Val1:dword`, является эквивалентом MASM соответствующего объявления для `g_Val1`, которое используется в коде C++. В этом случае директива `extern` уведомляет ассемблер о том, что место для хранения переменной `g_Val1` определено в другом модуле, а директива `dword` указывает, что `g_Val1` является двойным (или 32-битным) словом без знака.

Как и в примере в предыдущем разделе, аргументы `a`, `b`, `c` и `d` передаются функции `IntegerLogical_` через регистры ECX, EDX, R8D и R9D. Команда `and ecx,edx` выполняет побитовую операцию И над значениями в регистрах ECX

и EDX и сохраняет результат в регистре ECX. Команды `or ecx,r8d` и `xor ecx,r9d` выполняют побитовые операции включающего ИЛИ и исключающего ИЛИ соответственно. Команда `add ecx,[g_Val1]` складывает содержимое регистра ECX и значение глобальной переменной `g_Val1` и сохраняет полученную сумму в регистре ECX. Команда `mov eax,ecx` копирует окончательный результат в регистр EAX, чтобы его можно было передать обратно вызывающей функции. Вот результат выполнения примера `Ch02_02`:

```
Пример 1
a = 0x00223344 (2241348)
b = 0x00775544 (7820612)
c = 0x00555555 (5592405)
d = 0x00998877 (10061943)
val1 = 0x00000007 (7)
r1 = 0x00eedd29 (15654185)
r2 = 0x00eedd29 (15654185)

Пример 2
a = 0x70987655 (1889039957)
b = 0x55555555 (1431655765)
c = 0xaaaaaaaa (2863311530)
d = 0x12345678 (305419896)
val1 = 0x00000017 (23)
r1 = 0xe88ea89e (3901663390)
r2 = 0xe88ea89e (3901663390)
```

2.1.3. Операции сдвига

Последний пример исходного кода этого раздела, который по форме похож на предыдущие два примера, демонстрирует использование команд `shl` (логический сдвиг влево) и `shr` (логический сдвиг вправо). Он также иллюстрирует применение нескольких наиболее часто используемых команд, включая `cmp` (сравнение), `ja` (перейти, если выше) и `xchg` (обмен значениями). В листинге 2.3 показан исходный код C++ и на языке ассемблера для примера `Ch02_03`.

Листинг 2.3. Пример Ch02_03

```
//-----------------------------------------------
//               Ch02_03.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <bitset>

using namespace std;

extern "C" int IntegerShift_(unsigned int a, unsigned int count, unsigned int* a_shl,
unsigned int* a_shr);

static void PrintResult(const char* s, int rc, unsigned int a, unsigned int count, unsigned
int a_shl, unsigned int a_shr)
```

```
{
    bitset<32> a_bs(a);
    bitset<32> a_shl_bs(a_shl);
    bitset<32> a_shr_bs(a_shr);
    const int w = 10;
    const char nl = '\n';

    cout << s << '\n';
    cout << "count =" << setw(w) << count << nl;
    cout << "a = " << setw(w) << a << " (0b" << a_bs << ")" << nl;

    if (rc == 0)
        cout << "Invalid shift count" << nl;
    else
    {
        cout << "shl = " << setw(w) << a_shl << " (0b" << a_shl_bs << ")" << nl;
        cout << "shr = " << setw(w) << a_shr << " (0b" << a_shr_bs << ")" << nl;
    }

    cout << nl;
}

int main()
{
    int rc;
    unsigned int a, count, a_shl, a_shr;

    a = 3119;
    count = 6;
    rc = IntegerShift_(a, count, &a_shl, &a_shr);
    PrintResult("Пример 1", rc, a, count, a_shl, a_shr);

    a = 0x00800080;
    count = 4;
    rc = IntegerShift_(a, count, &a_shl, &a_shr);
    PrintResult("Пример 2", rc, a, count, a_shl, a_shr);

    a = 0x80000001;
    count = 31;
    rc = IntegerShift_(a, count, &a_shl, &a_shr);
    PrintResult("Пример 3", rc, a, count, a_shl, a_shr);

    a = 0x55555555;
    count = 32;
    rc = IntegerShift_(a, count, &a_shl, &a_shr);
    PrintResult("Пример 4", rc, a, count, a_shl, a_shr);

    return 0;
}

;-----------------------------------------------
;               Ch02_03.asm
;-----------------------------------------------

;
; extern "C" int IntegerShift_(unsigned int a, unsigned int count, unsigned int* a_shl,
  unsigned int* a_shr);
```

```
;
; Возвращает: 0 = ошибка (count >= 32), 1 = успешное выполнение
;

        .code
IntegerShift_ proc
        xor eax,eax             ; код возврата в случае ошибки
        cmp edx,31              ; сравнение count и 31
        ja InvalidCount         ; переход, если count > 31

        xchg ecx,edx            ; обмен содержимого между ecx и edx
        mov eax,edx             ;eax = a
        shl eax,cl              ;eax = a << count;
        mov [r8],eax            ; сохранение результата

        shr edx,cl              ;edx = a >> count
        mov [r9],edx            ; сохранение результата

        mov eax,1               ;возвращаемый код успешного выполнения

InvalidCount:
        ret                     ; возврат в вызывающую функцию

IntegerShift_ endp
        end
```

В верхней части кода C++ объявление функции на языке ассемблера x86 `IntegerShift_` несколько отличается от предыдущих примеров тем, что определяет два аргумента-указателя. Указатели нужны этой функции, поскольку ей необходимо возвращать более одного результата своей вызывающей функции. Другое незначительное отличие состоит в том, что возвращаемое из `IntegerShift_` значение `int` используется, чтобы указать, является ли допустимым значение `count`. Оставшийся код C++ в листинге 2.3 выполняет функцию на языке ассемблера `IntegerShift_` с использованием нескольких тестовых примеров и отображает результаты.

Код на языке ассемблера функции `IntegerShift_` начинается с команды `xor eax,eax`, которая устанавливает регистр EAX в ноль. Это сделано для того, чтобы регистр EAX содержал правильный код возврата, если будет обнаружено недопустимое значение аргумента `count`. Следующая команда, `cmp edx,31`, сравнивает содержимое регистра EDX, который содержит счетчик, с постоянным значением `31`. Когда процессор выполняет операцию сравнения, он вычитает второй операнд из первого операнда, устанавливает флаги состояния на основе результата этой операции и отбрасывает результат. Если значение `count` превышает `31`, команда `ja InvalidCount` выполняет переход к месту в программе, указанному операндом-адресатом. Если вы посмотрите на несколько строк вперед, вы заметите оператор с текстом `InvalidCount:`. Этот текст называется *меткой*. Если условие `count > 31` истинно, команда `ja InvalidCount` передает управление программой первой команде на языке ассемблера сразу после метки `InvalidCount`. Обратите внимание, что эта команда может находиться в той же или в другой строке, как показано в листинге 2.3.

Команда `xchg ecx,edx` меняет местами значения в регистрах ECX и EDX. Причина в том, что команды `shl` и `shr` должны использовать регистр CL для счетчика сдвигов. Команда `mov eax,edx` копирует значение `a` в регистр EAX, а последующая команда `shl eax,cl` сдвигает это значение влево на количество битов, указанное в регистре CL. Значение 64-битного указателя `a_shr` передается в функцию `IntegerShift_` с использованием регистра R8 (в 64-битном программировании все указатели 64-битные). Команда `mov [r8],eax` сохраняет результат операции сдвига в ячейку памяти, указанную содержимым регистра R8.

Последующая команда `shr edx,cl` сдвигает значение в регистре EDX (который содержит значение аргумента `a`) вправо на количество битов, указанное в регистре CL. Затем этот результат сохраняется в ячейку памяти, на которую указывает регистр R9, в свою очередь содержащий указатель на ячейку памяти, заданную `a_shr`. Команда `shr` используется в функции `IntegerShift_`, поскольку аргумент `a` объявлен как `unsigned int`. Если `a` объявить как `int`, команду `sar` (арифметический сдвиг вправо) можно использовать для сохранения знакового бита исходного операнда. Команда `mov eax,1` загружает в EAX константу, чтобы указать, что значение аргумента `count` было допустимым. Следует отметить, что проверка `count` на превышение значения `31` была реализована, чтобы проиллюстрировать проверку аргументов в функции на языке ассемблера. Для команд сдвига, которые используют прямой или переменный счетчик битового сдвига, процессор выполняет операцию маскирования, которая ограничивает отсчет сдвига значением от `0` до `31`, когда целевой операнд 32-битный (для 64-битовых операндов используются граничные значения `0` и `63`). Вот результат выполнения кода примера `Ch02_03`:

```
Пример 1
count = 6
a = 3119 (0b00000000000000000000110000101111)
shl = 199616 (0b00000000000000011000010111000000)
shr = 48 (0b00000000000000000000000000110000)

Пример 2
count = 4
a = 8388736 (0b00000000100000000000000010000000)
shl = 134219776 (0b00001000000000000000100000000000)
shr = 524296 (0b00000000000010000000000000001000)

Пример 3
count = 31
a = 2147483649 (0b10000000000000000000000000000001)
shl = 2147483648 (0b10000000000000000000000000000000)
shr = 1 (0b00000000000000000000000000000001)

Пример 4
count = 32
a = 1431655765 (0b01010101010101010101010101010101)
Invalid shift count
```

2.2. Расширенная целочисленная арифметика

В этом разделе вы узнаете, как выполнять целочисленное умножение и деление. Вы также узнаете, как использовать набор команд на языке ассемблера x86-64 для выполнения целочисленных арифметических вычислений с использованием операндов разного размера. В дополнение к этим темам в этом разделе представлены важные концепции программирования и некоторые особенности соглашения о вызовах Visual C++.

Примечание. Требования к соглашению о вызовах Visual C++, описанные в этом разделе и в последующих главах, могут отличаться от таковых для других языков программирования высокого уровня и операционных систем. Если вы читаете эту книгу, чтобы изучить язык ассемблера x86-64, и планируете использовать его с другим языком программирования высокого уровня или операционной системой, вам следует обратиться к соответствующей документации за дополнительными сведениями относительно целевой платформы.

2.2.1. Умножение и деление

Листинг 2.4 содержит исходный код примера `Ch02_04`. В этом примере функция `IntegerMulDiv_` вычисляет произведение, частное и остаток двух целых чисел, используя команды `imul` (целочисленное умножение) и `idiv` (целочисленное деление). Обратите внимание, что объявление функции `IntegerMulDiv_` на языке C++ включает пять параметров. До этого момента вы видели только объявления функций с максимум четырьмя параметрами, а значения аргументов для этих параметров передавались с использованием регистров RCX, RDX, R8 и R9 или младшей части этих регистров. Причина использования этих регистров заключается в том, что они заявлены в соглашении о вызовах Visual C++.

Листинг 2.4. Пример Ch02_04

```
//-----------------------------------------------
//                Ch02_04.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>

using namespace std;

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

void PrintResult(const char* s, int rc, int a, int b, int p, int q, int r)
{
    const char nl = '\n';

    cout << s << nl;
    cout << "a = " << a << ", b = " << b << ", rc = " << rc << nl;
```

```
    if (rc != 0)
        cout << "prod = " << p << ", quo = " << q << ", rem = " << r << nl;
    else
        cout << "prod = " << p << ", quo = undefined" << ", rem = undefined" << nl;

    cout << nl;
}

int main()
{
    int rc;
    int a, b;
    int prod, quo, rem;

    a = 47;
    b = 13;
    prod = quo = rem = 0;
    rc = IntegerMulDiv_(a, b, &prod, &quo, &rem);
    PrintResult("Пример 1", rc, a, b, prod, quo, rem);

    a = -291;
    b = 7;
    prod = quo = rem = 0;
    rc = IntegerMulDiv_(a, b, &prod, &quo, &rem);
    PrintResult("Пример 2", rc, a, b, prod, quo, rem);

    a = 19;
    b = 0;
    prod = quo = rem = 0;
    rc = IntegerMulDiv_(a, b, &prod, &quo, &rem);
    PrintResult("Пример 3", rc, a, b, prod, quo, rem);

    a = 247;
    b = 85;
    prod = quo = rem = 0;
    rc = IntegerMulDiv_(a, b, &prod, &quo, &rem);
    PrintResult("Пример 4", rc, a, b, prod, quo, rem);

    return 0;
}

;-----------------------------------------------
;                Ch02_04.asm
;-----------------------------------------------
;
; extern "C" int IntegerMulDiv_(int a, int b, int* prod, int* quo, int* rem);
;
; Возвращает: 0 = ошибка (делитель равен нулю), 1 = успех
;

        .code
IntegerMulDiv_ proc

; Нужно убедиться, что делитель не равен нулю
        mov eax,edx                 ;eax = b
        or eax,eax                  ;логическое ИЛИ устанавливает флаги состояния
```

```
        jz InvalidDivisor           ;переход, если b равно нулю

; Вычисление произведения и сохранение результата
        imul eax,ecx                ;eax = a * b
        mov [r8],eax                ;сохранение результата умножения

; Вычисление частного и остатка, сохранение результата
        mov r10d,edx                ;r10d = b
        mov eax,ecx                 ;eax = a
        cdq                         ;edx:eax содержит 64-битный делитель
        idiv r10d                   ;eax = частное, edx = остаток

        mov [r9],eax                ;сохранение частного
        mov rax,[rsp+40]            ;rax = 'rem'
        mov [rax],edx               ;сохранение остатка
        mov eax,1                   ;сохранение кода успешного выполнения

InvalidDivisor:
        ret                         ;возврат к вызывающей функции
        IntegerMulDiv_ endp
        end
```

Соглашение о вызовах – это двоичный протокол, описывающий, как между двумя функциями происходит обмен аргументами и возвращаемыми значениями. Как вы уже видели, соглашение о вызовах Visual C++ для программ x86-64 в Windows требует, чтобы вызывающая функция передавала первые четыре целочисленных аргумента (или указателя) с использованием регистров RCX, RDX, R8 и R9. Младшие части этих регистров используются для значений аргументов меньше 64 бит (например, ECX, CX или CL для 32-, 16- или 8-битных целых чисел). Любые дополнительные аргументы передаются с использованием стека. Соглашение о вызовах также определяет дополнительные требования, включая правила для передачи значений с плавающей запятой, использования регистров общего назначения и XMM, а также стековых фреймов. Вы узнаете об этих дополнительных требованиях в главе 5.

Код C++ в листинге 2.4 похож на другие примеры, которые вы уже видели. Он просто выполняет несколько тестовых вычислений и отображает результаты. При входе в функцию `IntegerMulDiv_` регистры ECX, EDX, R8 и R9 содержат значения аргументов `a`, `b`, `prod` и `quo` соответственно. Пятый аргумент `rem` передается в стек, как показано на рис. 2.1. Обратите внимание: поскольку `prod`, `quo` и `rem` являются указателями, они передаются в `IntegerMulDiv_` как 64-битные значения.

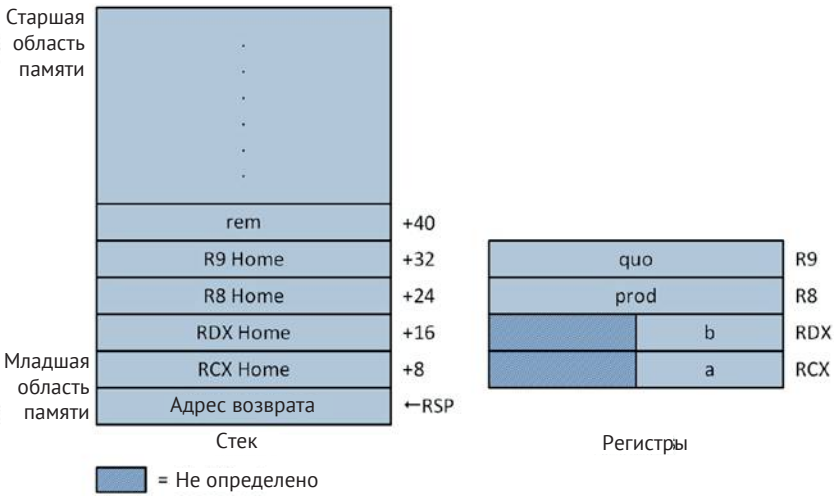


Рис. 2.1. Состояние стека и регистров аргументов при входе в функцию `IntegerMulDiv_`

На рис. 2.1 показано состояние стека и регистров аргументов при входе в `IntegerMulDiv_`, но до выполнения первой команды. Обратите внимание, что значение пятого аргумента `rem` находится по адресу памяти RSP + 40. Так же можно в случае необходимости использовать простую команду `mov` для загрузки `rem`, который является указателем, в регистр общего назначения. Обратите внимание на рис. 2.1, что регистр RSP указывает на обратный адрес вызывающей функции в стеке. Во время выполнения команды `ret` процессор копирует это значение из стека и в конечном итоге сохраняет его в регистре RIP. Команда `ret` также удаляет адрес возврата вызывающей функции из стека, добавляя 8 к значению в RSP. Ячейки стека, помеченные на рисунке как RCX Home, RDX Home, R8 Home и R9 Home, представляют собой области хранения, которые можно использовать для временного сохранения соответствующих регистров аргументов. Эти области также можно использовать для хранения других промежуточных данных. Вы узнаете больше о *домашнем пространстве* (home area) в главе 5.

Функция IntegerMulDiv_ вычисляет и сохраняет произведение `a*b`. Она также вычисляет и сохраняет частное и остаток от `a/b`. Поскольку `IntegerMulDiv_` выполняет деление с использованием свободной переменной `b`, имеет смысл проверить значение `b`, чтобы убедиться, что оно не равно нулю. В листинге 2.4 команда `mov eax,edx` копирует `b` в регистр EAX. Следующая команда `eax,eax` выполняет побитовую операцию ИЛИ для установки флагов состояния. Если значение `b` равно нулю, команда `jz InvalidDivisor` (переход, если ноль) пропускает код, выполняющий деление. Как и в предыдущем примере, функция `IntegerMulDiv_` использует нулевое возвращаемое значение для обозначения состояния ошибки. Поскольку EAX уже содержит ноль, никаких дополнительных команд не требуется.

Следующая команда `imul eax,ecx` вычисляет `a*b` и сохраняет произведение в ячейку памяти, указанную регистром R8, который содержит указатель `prod`. Набор команд x86-64 поддерживает несколько различных форм команды `imul`. Используемая здесь форма с двумя операндами фактически вычисляет

64-битный результат (напомним, что произведение двух 32-битных целых чисел всегда является 64-битным результатом), но сохраняет только младшие 32 бита в операнде назначения. Форма `imul` с одним операндом может использоваться, когда требуется неусеченный результат.

Далее происходит целочисленное деление. Команды `mov r10d,rdx` и `mov eax,ecx` загружают в регистры R10D и EAX значения аргументов `b` и `a` соответственно. Перед выполнением операции деления 32-битное делимое в EAX должно быть расширено со знаком до 64 битов, и это выполняется с помощью команды `cdq` (преобразовать двойное слово в четверное слово). После выполнения `cdq` регистровая пара EDX:EAX содержит 64-битное делимое, а регистр R10D содержит 32-битный делитель. Команда `idiv r10d` делит содержимое пары регистров EDX:EAX на значение в R10D. После выполнения команды `idiv` 32-битное частное и 32-битный остаток находятся в регистрах EAX и EDX соответственно. Последующая команда `mov [r9],eax` сохраняет частное в ячейку памяти, указанную с помощью `quo`. Чтобы сохранить остаток, необходимо получить из стека указатель `rem`, и это достигается с помощью команды `mov rax,[rsp+40]`. Команда `mov [rax],edx` сохраняет остаток в ячейку памяти, на которую указывает `rem`. Результат выполнения кода примера `Ch02_04` выглядит следующим образом:

```
Пример 1
a = 47, b = 13, rc = 1
prod = 611, quo = 3, rem = 8

Пример 2
a = -291, b = 7, rc = 1
prod = -2037, quo = -41, rem = -4

Пример 3
a = 19, b = 0, rc = 0
prod = 0, quo = undefined, rem = undefined

Пример 4
a = 247, b = 85, rc = 1
prod = 20995, quo = 2, rem = 77
```

2.2.2. Вычисления с использованием смешанных типов

Во многих программах часто необходимо выполнять арифметические вычисления с использованием нескольких целочисленных типов. Рассмотрим выражение C++ `a = b*c*d*e`, где `a`, `b`, `c`, `d` и `e` объявлены как `long long`, `long long`, `int`, `short` и `char`. Для вычисления правильного результата требуется корректное преобразование целых чисел меньшего размера в большие. В следующем примере вы познакомитесь с несколькими методами, которые можно использовать для *целочисленного расширения типов* в функции на языке ассемблера. Вы также узнаете, как получить доступ к целочисленным значениям аргументов различного размера, которые хранятся в стеке. Листинг 2.5 содержит исходный код примера `Ch02_05`.

Листинг 2.5. Пример Ch02_05

```
//-----------------------------------------------
//                Ch02_05.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <cstdint>

using namespace std;

extern "C" int64_t IntegerMul_(int8_t a, int16_t b, int32_t c, int64_t d, int8_t e, int16_t
f, int32_t g, int64_t h);

extern "C" int UnsignedIntegerDiv_(uint8_t a, uint16_t b, uint32_t c, uint64_t d, uint8_t
e,
uint16_t f, uint32_t g, uint64_t h, uint64_t* quo, uint64_t* rem);

void IntegerMul(void)
{
    int8_t a = 2;
    int16_t b = -3;
    int32_t c = 8;
    int64_t d = 4;
    int8_t e = 3;
    int16_t f = -7;
    int32_t g = -5;
    int64_t h = 10;

    // Вычисление a * b * c * d * e * f * g * h
    int64_t prod1 = a * b * c * d * e * f * g * h;
    int64_t prod2 = IntegerMul_(a, b, c, d, e, f, g, h);

    cout << "\nРезультаты IntegerMul\n";
    cout << "a = " << (int)a << ", b = " << b << ", c = " << c << ' ';
    cout << "d = " << d << ", e = " << (int)e << ", f = " << f << ' ';
    cout << "g = " << g << ", h = " << h << '\n';
    cout << "prod1 = " << prod1 << '\n';
    cout << "prod2 = " << prod2 << '\n';
}

void UnsignedIntegerDiv(void)
{
    uint8_t a = 12;
    uint16_t b = 17;
    uint32_t c = 71000000;
    uint64_t d = 90000000000;
    uint8_t e = 101;
    uint16_t f = 37;
    uint32_t g = 25;
    uint64_t h = 5;
    uint64_t quo1, rem1;
    uint64_t quo2, rem2;

    quo1 = (a + b + c + d) / (e + f + g + h);
    rem1 = (a + b + c + d) % (e + f + g + h);
```

```
    UnsignedIntegerDiv_(a, b, c, d, e, f, g, h, &quo2, &rem2);
    cout << "\nРезультаты UnsignedIntegerDiv\n";
    cout << "a = " << (unsigned)a << ", b = " << b << ", c = " << c << ' ';
    cout << "d = " << d << ", e = " << (unsigned)e << ", f = " << f << ' ';
    cout << "g = " << g << ", h = " << h << '\n';
    cout << "quo1 = " << quo1 << ", rem1 = " << rem1 << '\n';
    cout << "quo2 = " << quo2 << ", rem2 = " << rem2 << '\n';
}

int main()
{
    IntegerMul();
    UnsignedIntegerDiv();
    return 0;
}
;-----------------------------------------------
;                 Ch02_05.asm
;-----------------------------------------------
; extern "C" int64_t IntegerMul_(int8_t a, int16_t b, int32_t c, int64_t d, int8_t e,
int16_t f, int32_t g, int64_t h);

        .code
IntegerMul_ proc

; Вычисление a * b * c * d
        movsx rax,cl                    ;rax = sign_extend(a)
        movsx rdx,dx                    ;rdx = sign_extend(b)
        imul rax,rdx                    ;rax = a * b
        movsxd rcx,r8d                  ;rcx = sign_extend(c)
        imul rcx,r9                     ;rcx = c * d
        imul rax,rcx                    ;rax = a * b * c * d

; Вычисление e * f * g * h
        movsx rcx,byte ptr [rsp+40]     ;rcx = sign_extend(e)
        movsx rdx,word ptr [rsp+48]     ;rdx = sign_extend(f)
        imul rcx,rdx                    ;rcx = e * f
        movsxd rdx,dword ptr [rsp+56]   ;rdx = sign_extend(g)
        imul rdx,qword ptr [rsp+64]     ;rdx = g * h
        imul rcx,rdx                    ;rcx = e * f * g * h

; Вычисление окончательного произведения
        imul rax,rcx                    ;rax = окончательное произведение
        ret
        IntegerMul_ endp

; extern "C" int UnsignedIntegerDiv_(uint8_t a, uint16_t b, uint32_t c, uint64_t d, uint8_t
e,
uint16_t f, uint32_t g, uint64_t h, uint64_t* quo, uint64_t* rem);

UnsignedIntegerDiv_ proc

; Вычисление a + b + c + d
        movzx rax,cl                    ;rax = zero_extend(a)
        movzx rdx,dx                    ;rdx = zero_extend(b)
        add rax,rdx                     ;rax = a + b
        mov r8d,r8d                     ;r8 = zero_extend(c)
        add r8,r9                       ;r8 = c + d
```

```
        add rax,r8                      ;rax = a + b + c + d
        xor rdx,rdx                     ;rdx:rax = a + b + c + d

; Вычисление e + f + g + h
        movzx r8,byte ptr [rsp+40]      ;r8 = zero_extend(e)
        movzx r9,word ptr [rsp+48]      ;r9 = zero_extend(f)
        add r8,r9                       ;r8 = e + f
        mov r10d,[rsp+56]               ;r10 = zero_extend(g)
        add r10,[rsp+64]                ;r10 = g + h;
        add r8,r10                      ;r8 = e + f + g + h
        jnz DivOK                       ;переход, если делитель не равен нулю
        xor eax,eax                     ;установка кода ошибки
        jmp done

; Вычисление (a + b + c + d) / (e + f + g + h)
        DivOK: div r8                   ;беззнаковое деление rdx:rax / r8
        mov rcx,[rsp+72]
        mov [rcx],rax                   ;сохранение частного
        mov rcx,[rsp+80]
        mov [rcx],rdx                   ;сохранение остатка
        mov eax,1                       ;установка кода успешной процедуры

Done: ret
UnsignedIntegerDiv_ endp
        end
```

Функция на языке ассемблера `IntegerMul_` вычисляет произведение восьми целых чисел со знаком размером от 8 до 64 бит. Объявление C++ для этой функции использует целочисленные типы фиксированного размера, которые объявлены в заголовочном файле `<cstdint>` вместо обычных `long long`, `int`, `short` и `char`. Некоторые программисты на языке ассемблера (включая меня) предпочитают использовать для объявлений ассемблерных функций целочисленные типы фиксированного размера, поскольку это подчеркивает точный размер аргумента. Объявление функции `UnsignedIntegerDiv_`, которая демонстрирует выполнение беззнакового целочисленного деления, также использует целочисленные типы фиксированного размера. На рис. 2.2 показано содержимое стека при входе в `IntegerMul_`.

Первая команда ассемблерной функции, `movsx rax,cl` (присвоение с расширением знака), расширяет разрядом знака копию 8-битового целочисленного значения `a`, которое находится в регистре CL, до 64 бит и сохраняет это значение в регистре RAX. Обратите внимание, что эта операция не изменяет исходное значение в регистре CL. Далее следует другая команда `movsx`, которая сохраняет 64-битную копию 16-битного значения `d` с расширением разрядом знака в RDX. Как и в случае предыдущей команды `movsx`, исходный операнд этой операцией не изменяется. Команда `imul rax,rdx` вычисляет произведение `a` и `b`. Используемая здесь двухоперандная форма команды `imul` сохраняет только младшие 64 бита 128-битного произведения в операнде назначения RAX. Следующая команда `movsxd rcx,r8d` расширяет разрядом знака 32-битный операнд `c` до 64 бит. Обратите внимание, что при расширении разрядом знака 32-разрядного целого до 64-разрядного числа требуется другая мнемоника. Следующие две команды `imul` вычисляют промежуточное произведение `a*b*c*d`.

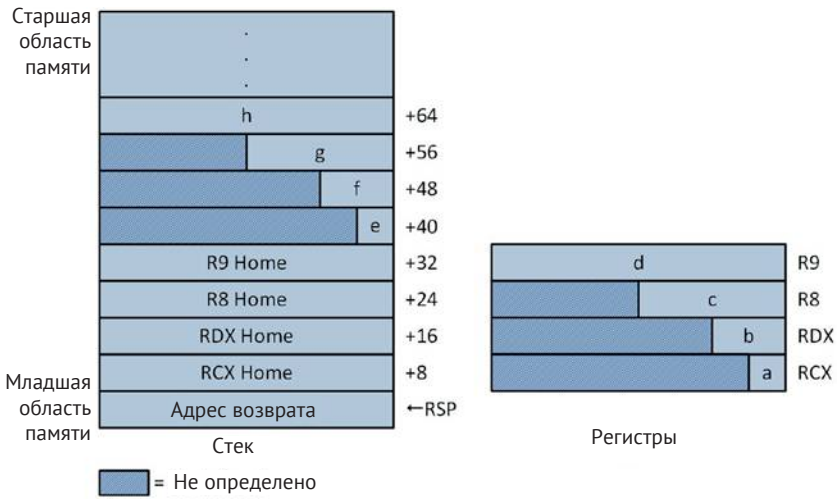


Рис. 2.2. Состояние стека и регистров аргументов при входе в функцию `IntegerMul`

Далее выполняется вычисление второго промежуточного произведения `e*f*g*h`. Все эти значения аргументов были переданы с использованием стека, как показано на рис. 2.2. Команда `movsx rcx,byte ptr [rsp + 40]` расширяет знаковым разрядом копию 8-битного значения аргумента `e`, которое находится в стеке, и сохраняет результат в регистре RCX. Текст `byte ptr` – это директива MASM, которая действует как оператор приведения в C++ и передает ассемблеру размер исходного операнда. Без директивы `byte ptr` команда `movsx` неоднозначна, поскольку исходный операнд может иметь разные размеры. Затем загружается значение аргумента `f` с помощью команды `movsx rdx,word ptr [rsp + 48]`. После вычисления промежуточного произведения `e*f` с использованием команды `imul` команда `movsxd rdx,dword ptr [rsp + 56]` загружает расширенную разрядом знака копию `g` в RDX. После этого следует команда `imul rdx,qword ptr [rsp + 64]`, которая вычисляет промежуточное произведение `g*h`. Применение директивы `qword ptr` здесь необязательно; директивы размера часто используются просто для улучшения читаемости программы. Последние две команды `imul` рассчитывают конечное произведение.

На рис. 2.3 показано содержимое стека при входе в функцию `UnsignedIntegerDiv_`. Эта функция вычисляет частное и остаток от выражения `(a + b + c + d) / (e + f + g + g)`. Как следует из названия, функция `UnsignedIntegerDiv_` использует беззнаковые целочисленные аргументы разных размеров и выполняет беззнаковое целочисленное деление. Чтобы вычислить правильные результаты, перед любыми арифметическими операциями аргументы меньшего размера должны быть расширены нулями. Команды `movzx rax,cl` и `movzx rdx,dx` загружают расширенные нулями копии значений аргументов `a` и `b` в соответствующие регистры назначения. Далее команда `add rax,rdx` вычисляет промежуточную сумму `a+b`. На первый взгляд следующая команда `mov r8d,r8d` кажется излишней, но на самом деле она выполняет необходимую операцию. Когда процессор x86 работает в 64-битном режиме, команды, использующие 32-битные операнды, дают 32-битные результаты. Если операнд-адресат является 32-битным

регистром, старшие 32 бита (то есть биты 63–32) соответствующего 64-битного регистра устанавливаются в ноль. Команда `mov r8d,r8d` используется здесь для расширения нулями 32-битного значения `c`, которое уже загружено в регистр R8D, до 64-битного значения в R8. Следующие две команды сложения вычисляют промежуточную сумму `a + b + c + d` и сохраняют результат в RAX. Последующая команда `xor rdx,rdx` дает расширенное нулями 128-битное значение делимого, которое хранится в паре регистров RDX:RAX.

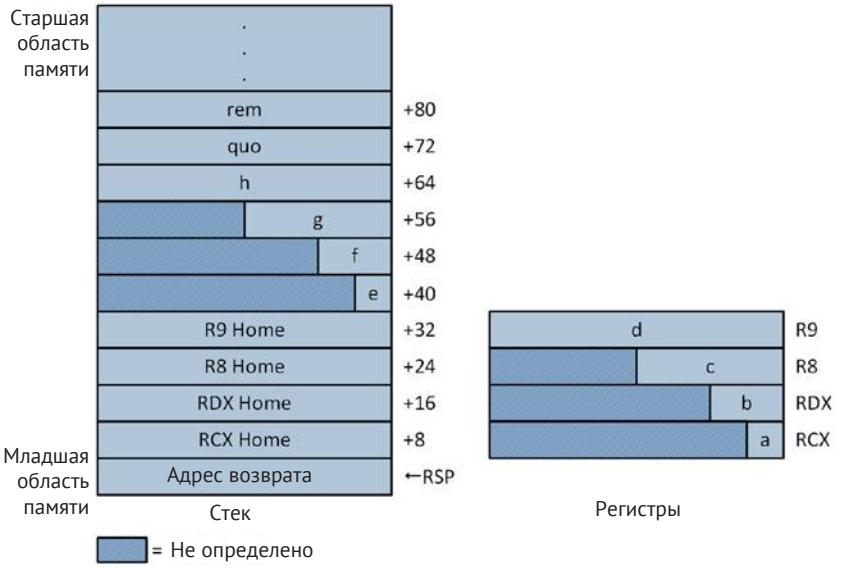


Рис. 2.3. Состояние стека и регистров аргументов при входе в функцию `UnsignedIntegerDiv_`

Аналогичная последовательность команд используется для вычисления промежуточной суммы `e + f + g + h`, с основным отличием в том, что эти аргументы загружаются из стека. Затем это значение проверяется на равенство нулю, поскольку оно будет использоваться в качестве делителя. Если делитель не равен нулю, команда `div r8` выполняет целочисленное деление без знака, используя регистровую пару RDX:RAX в качестве делимого и регистр R8 в качестве делителя. Полученное частное (RAX) и остаток (RDX) затем сохраняются в ячейки памяти, отмеченные указателями `quo` и `rem`, которые были переданы в стек. Вот результат выполнения кода из примера `Ch02_05`:

```
Результаты IntegerMul
a = 2, b = -3, c = 8 d = 4, e = 3, f = -7 g = -5, h = 10
prod1 = -201600
prod2 = -201600

Результаты UnsignedIntegerDiv
a = 12, b = 17, c = 71000000 d = 90000000000, e = 101, f = 37 g = 25, h = 5
quo1 = 536136904, rem1 = 157
quo2 = 536136904, rem2 = 157
```

2.3. Команды адресации памяти и состояния

До сих пор примеры исходного кода этой главы в основном иллюстрировали, как использовать базовые арифметические и логические команды. В этом разделе вы узнаете больше о режимах адресации памяти x86. Вы также изучите образец кода, демонстрирующий использование некоторых команд x86, основанных на условном коде.

2.3.1. Режимы адресации памяти

В главе 1 вы узнали, что набор команд x86-64 поддерживает различные режимы адресации, которые можно использовать для ссылки на операнд в памяти. В этом разделе вы изучите функцию на языке ассемблера, которая демонстрирует использование некоторых режимов. Вы также узнаете, как инициализировать таблицу перекодировки на языке ассемблера и использовать глобальные переменные ассемблера в функции C++. В листинге 2.6 показан исходный код примера `Ch02_06`.

Листинг 2.6. Пример Ch02_06

```
//-----------------------------------------------
//                Ch02_06.cpp
//-----------------------------------------------
#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

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

int main()
{
    const int w = 5;
    const char nl = '\n';
    const char* delim = ", ";

    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);
        cout << "i = " << setw(w - 1) << i << delim;
        cout << "rc = " << setw(w - 1) << rc << delim;
        cout << "v1 = " << setw(w) << v1 << delim;
        cout << "v2 = " << setw(w) << v2 << delim;
        cout << "v3 = " << setw(w) << v3 << delim;
        cout << "v4 = " << setw(w) << v4 << delim;
        cout << nl;
    }

    cout << "FibValsSum_ = " << FibValsSum_ << nl;
    return 0;
}
```

```
;-----------------------------------------------
;                Ch02_06.asm
;-----------------------------------------------
; Простая таблица перекодировки (секция данных .const только для чтения)
        .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_

; Секция данных (чтение и запись)

        .data
FibValsSum_ dword ? ;значение для демонстрации адресации относительно RIP
        public FibValsSum_

;
; extern "C" int MemoryAddressing_(int i, int* v1, int* v2, int* v3, int* v4);
;
; Возвращает: 0 = ошибка (недопустимый индекс таблицы), 1 = успех
;
        .code
MemoryAddressing_ proc

; Нужно убедиться, что значение i корректно
        cmp ecx,0
        jl InvalidIndex                     ;jump if i < 0
        cmp ecx,[NumFibVals_]
        jge InvalidIndex                    ;переход, если i >= NumFibVals_

; Расширение i знаковым разрядом для использования в вычислении адреса
        movsxd rcx,ecx                      ;расширение i
        mov [rsp+8],rcx                     ;сохранение копии i

; Пример #1 – базовый регистр
        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                       ; сохранение в v1

; Пример #2 – базовый регистр + индексный регистр
        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                        ;сохранение в v2

; Пример #3 - базовый регистр + индексный регистр * масштабный коэффициент
        mov r11,offset FibVals              ;r11 = FibVals
        mov rcx,[rsp+8]                     ;rcx = i
        mov eax,[r11+rcx*4]                 ;eax = FibVals[i]
        mov [r9],eax                        ;сохранение в v3
```

```
; Пример #4 - базовый регистр + индексный регистр * масштабный коэффициент + disp
        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 = ptr to v4
        mov [r10],eax                       ;сохранение в v4

; Пример #5 - относительно RIP
        add [FibValsSum_],eax               ;обновление суммы
        mov eax,1                           ;установка кода успешного выполнения
        ret

InvalidIndex:
        xor eax,eax                         ;установка кода ошибки
        ret

MemoryAddressing_ endp
        end
```

В верхней части кода C++ находятся необходимые операторы объявления для этого примера. Ранее в этой главе вы узнали, как ссылаться на глобальную переменную C++ в функции на языке ассемблера. В этом примере показано обратное. Пространство для хранения переменных `NumFibVals_` и `FibValsSum_` определено в коде на языке ассемблера, и на эти переменные ссылаются в функции `main` кода C++.

В функции на языке ассемблера `MemoryOperands_` аргумент `i` используется как индекс в массиве (или таблице перекодировки) постоянных целых чисел, в то время как четыре аргумента указателя используются для сохранения значений, загруженных из таблицы поиска с использованием различных режимов адресации. В верхней части листинга 2.6 находится директива `.const`, которая определяет блок памяти, содержащий данные только для чтения. Сразу после директивы `.const` определяется таблица перекодировки с именем `FibVals`. Эта таблица содержит 16 целочисленных значений двойных слов. Нотация `dword` – это директива ассемблера, которая используется для выделения места для хранения и, при необходимости, инициализации значения двойного слова (как синоним для `dword` можно использовать нотацию `dd`).

Строка `NumFibVals_ dword ($-FibVals) / sizeof dword` выделяет место для хранения одного значения двойного слова и инициализирует его количеством элементов двойного слова в таблице поиска `FibVals`. Символ `$` – это символ ассемблера, который равен текущему значению счетчика местоположений (или смещению от начала текущего блока памяти). Вычитание смещения `FibVals` из `$` дает размер таблицы в байтах. Разделив этот результат на размер значения двойного слова в байтах, вы получите правильное количество элементов. Эти операторы имитируют широко используемый метод в C++ для определения и инициализации переменной с количеством элементов в массиве:

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

Последняя строка раздела `.const` объявляет `NumFibVals_` как общедоступное символическое имя, чтобы разрешить его использование в `main`. Директива `.data`

обозначает начало блока памяти, который содержит изменяемые данные. Оператор `FibValsSum_ dword ?` определяет неинициализированное значение двойного слова, а последующий оператор `public` делает его глобально доступным.

Теперь посмотрим на код на языке ассемблера для `MemoryAddressing_`. При входе в функцию аргумент `i` проверяется на достоверность, поскольку он будет использоваться в качестве индекса в таблице поиска `FibVals`. Команда `cmp ecx,0` сравнивает содержимое ECX, которое содержит `i`, с непосредственным значением `0`. Как обсуждалось ранее в этой главе, процессор выполняет это сравнение, вычитая исходный операнд из целевого операнда. Затем он устанавливает флаги состояния на основе результата вычитания (результат не сохраняется в операнде-адресате). Если условие `ecx < 0` истинно, управление программой будет передано в место, указанное командой `jl` (перейти, если меньше). Аналогичная последовательность команд используется для определения того, не слишком ли велико значение `i`. Команда `cmp ecx,[NumFibVals_]` сравнивает ECX с количеством элементов в таблице поиска. Если `ecx > = [NumFibVals]` истинно, выполняется переход к целевому местоположению, указанному командой `jge` (перейти, если больше или равно).

Сразу после проверки `i` команда `movsxd rcx,ecx` расширяет знаковым битом значение индекса таблицы до 64 бит. Как вы скоро увидите, расширение знаком или нулем 32-битного целого числа до 64-битного целого числа часто необходимо при использовании режима адресации, в котором применяется индексный регистр. Затем команда `mov [rsp + 8],rcx` сохраняет копию расширенного знаковым битом значения индекса таблицы в домашнее пространство RCX в стеке. Эта команда в данном примере в первую очередь предназначена для иллюстрации использования домашнего пространства стека.

Остальные команды функции `MemoryAddressing_` иллюстрируют доступ к элементам в таблице поиска с использованием различных режимов адресации памяти. В первом примере для чтения элемента из таблицы используется единственный базовый регистр. Чтобы использовать один базовый регистр, функция должна явно вычислить адрес *i*-го элемента таблицы, что достигается сложением смещения (или начального адреса) `FibVals` и значения `i*4`. Команда `mov r11,offset FibVals` загружает в R11 правильное значение смещения таблицы. За ней следует команда `shl rcx,2`, которая определяет смещение *i*-го элемента относительно начала таблицы поиска. Команда `add r11,rcx` вычисляет окончательный адрес. Затем указанное табличное значение считывается с помощью команды `mov eax,[r11]` и сохраняется в ячейку памяти, указанную аргументом `v1`.

Во втором примере табличное значение считывается с использованием адресации памяти `BaseReg + IndexReg`. Этот пример похож на первый, за исключением того, что процессор вычисляет окончательный эффективный адрес во время выполнения команды `mov eax,[r11 + rcx]`. Обратите внимание, что пересчет смещения элемента таблицы с использованием команд `mov rcx,[rsp + 8]` и `shl rcx,2` здесь не требуется, но включен в листинг для иллюстрации использования домашнего пространства стека.

Третий пример демонстрирует использование адресации памяти `BaseReg + IndexReg * ScaleFactor`. В этом примере смещение `FibVals` и значение `i` загружаются в регистры R11 и RCX соответственно. Искомое табличное значение загружается в EAX с помощью команды `mov eax,[r11 + rcx * 4]`. В четвертом (и несколько

надуманном) примере демонстрируется адресация памяти `BaseReg + IndexReg * ScaleFactor + Disp`. Пятый и последний пример режима адресации памяти использует команду `add [FibValsSum_],eax` как пример адресации относительно RIP. Эта команда, которая использует ячейку памяти в качестве операнда назначения, обновляет текущую сумму, в конечном итоге отображаемую кодом C++.

Функция `main`, показанная в листинге 2.6, содержит простой цикл, который вызывает функцию `MemoryOperands_`, включая тестовые примеры с недопустимым индексом. Обратите внимание, что цикл `for` использует переменную `NumFibVals_`, которая была определена как общедоступный символ в файле на языке ассемблера. Ниже показаны выходные данные примера `Ch02_06`.

```
 i =    -1, rc =    0,  vl =       -1,  V2 =       -1,  v3 =       -1,  v4 =       -1,
 i =     0, rc =    1,  vl =        0,  V2 =        0,  v3 =        0,  v4 =        0,
 i =     1, rc =    1,  vl =        1,  V2 =        1,  v3 =        1,  v4 =        1,
 i =     2, rc =    1,  vl =        1,  V2 =        1,  v3 =        1,  v4 =        1,
 i =     3, rc =    1,  vl =        2,  V2 =        2,  v3 =        2,  v4 =        2,
 i =     4, rc =    1,  vl =        3,  V2 =        3,  v3 =        3,  v4 =        3,
 i =     5, rc =    1,  vl =        5,  V2 =        5,  v3 =        5,  v4 =        5,
 i =     6, rc =    1,  vl =        8,  V2 =        8,  v3 =        8,  v4 =        8,
 i =     7, rc =    1,  vl =       13,  V2 =       13,  v3 =       13,  v4 =       13,
 i =     8, rc =    1,  vl =       21,  V2 =       21,  v3 =       21,  v4 =       21,
 i =     9, rc =    1,  vl =       34,  V2 =       34,  v3 =       34,  v4 =       34,
 i =    10, rc =    1,  vl =       55,  V2 =       55,  v3 =       55,  v4 =       55,
 i =    11, rc =    1,  vl =       89,  V2 =       89,  v3 =       89,  v4 =       89,
 i =    12, rc =    1,  vl =      144,  V2 =      144,  v3 =      144,  v4 =      144,
 i =    13, rc =    1,  vl =      233,  V2 =      233,  v3 =     |233,  v4 =      233,
 i =    14, rc =    1,  vl =      377,  V2 =      377,  v3 =      377,  v4 =      377,
 i =    15, rc =    1,  vl =      610,  V2 =      610,  v3 =      610,  v4 =      610,
 i =    16, rc =    1,  vl =      987,  V2 =      987,  v3 =      987,  v4 =      987,
 i =    17, rc =    1,  vl =     1597,  V2 =     1597,  v3 =     1597,  v4 =     1597,
 i =    18, rc =    0,  vl =       -1,  V2 =       -1,  v3 =       -1,  v4 =       -1,
FibValsSum_ = 4180
```

Поскольку на процессоре x86 доступны несколько режимов адресации, вы можете задаться вопросом, какой режим следует использовать. Ответ на этот вопрос зависит от ряда факторов, включая доступность регистров, количество ожидаемых выполнений команды (или последовательности команд), порядок команд и компромисс между объемом памяти и временем выполнения. Также необходимо учитывать такие аппаратные особенности, как базовая микроархитектура процессора и размеры кеш-памяти.

При кодировании функции на языке ассемблера x86 рекомендуется отдавать предпочтение простой (один базовый регистр или смещение), а не сложной (несколько регистров) адресации памяти. Недостатком этого подхода является то, что более простые формы обычно требуют от программиста кодирования более длинных последовательностей команд и могут занимать больше места для кода. Использование простой формы также может быть неоптимальным, если требуются дополнительные команды для сохранения энергонезависимых регистров в стеке (энергонезависимые регистры рассмотрены в главе 3). В главе 15 более подробно рассматриваются некоторые проблемы и компромиссы, которые могут повлиять на эффективность кода на языке ассемблера.

2.3.2. Условные команды

Последний пример программы этой главы объясняет, как использовать условные команды x86 `jcc` (условный переход) и `cmovcc` (условное присвоение). Как вы уже видели в нескольких примерах исходного кода этой главы, выполнение условной команды зависит от указанного в ней кода условия и состояния одного или нескольких флагов состояния. Пример исходного кода `Ch02_07` в листинге 2.7 демонстрирует еще несколько вариантов использования ранее упомянутых команд.

Листинг 2.7. Пример Ch02_07

```
//-----------------------------------------------
//               Ch02_07.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

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);

void PrintResult(const char* s1, int a, int b, int c, int result)
{
    const int w = 4;

    cout << s1 << "(";
    cout << setw(w) << a << ", ";
    cout << setw(w) << b << ", ";
    cout << setw(w) << c << ") = ";
    cout << setw(w) << result << '\n';
}

int main()
{
    int a, b, c;
    int smin_a, smax_a, smin_b, smax_b;
```

```
    // Примеры SignedMin
    a = 2; b = 15; c = 8;
    smin_a = SignedMinA_(a, b, c);
    smin_b = SignedMinB_(a, b, c);
    PrintResult("SignedMinA", a, b, c, smin_a);
    PrintResult("SignedMinB", a, b, c, smin_b);
    cout << '\n';

    a = -3; b = -22; c = 28;
    smin_a = SignedMinA_(a, b, c);
    smin_b = SignedMinB_(a, b, c);
    PrintResult("SignedMinA", a, b, c, smin_a);
    PrintResult("SignedMinB", a, b, c, smin_b);
    cout << '\n';

    a = 17; b = 37; c = -11;
    smin_a = SignedMinA_(a, b, c);
    smin_b = SignedMinB_(a, b, c);
    PrintResult("SignedMinA", a, b, c, smin_a);
    PrintResult("SignedMinB", a, b, c, smin_b);
    cout << '\n';

    // Примеры SignedMax
    a = 10; b = 5; c = 3;
    smax_a = SignedMaxA_(a, b, c);
    smax_b = SignedMaxB_(a, b, c);
    PrintResult("SignedMaxA", a, b, c, smax_a);
    PrintResult("SignedMaxB", a, b, c, smax_b);
    cout << '\n';

    a = -3; b = 28; c = 15;
    smax_a = SignedMaxA_(a, b, c);
    smax_b = SignedMaxB_(a, b, c);
    PrintResult("SignedMaxA", a, b, c, smax_a);
    PrintResult("SignedMaxB", a, b, c, smax_b);
    cout << '\n';

    a = -25; b = -37; c = -17;
    smax_a = SignedMaxA_(a, b, c);
    smax_b = SignedMaxB_(a, b, c);
    PrintResult("SignedMaxA", a, b, c, smax_a);
    PrintResult("SignedMaxB", a, b, c, smax_b);
    cout << '\n';
}
;-----------------------------------------------
;                Ch02_07.asm
;-----------------------------------------------
; extern "C" int SignedMinA_(int a, int b, int c);
;
; Returns: min(a, b, c)
        .code
SignedMinA_ proc
        mov eax,ecx
        cmp eax,edx       ;сравнение a и b
        jle @F             ;переход, если a <= b
        mov eax,edx        ;eax = b
```

```
@@:     cmp eax,r8d        ;сравнение min(a, b) и c
        jle @F
        mov eax,r8d        ;eax = min(a, b, c)

@@:     ret
SignedMinA_ endp

; extern "C" int SignedMaxA_(int a, int b, int c);
;
; Возвращает: max(a, b, c)

SignedMaxA_ proc
        mov eax,ecx
        cmp eax,edx        ;сравнение a и b
        jge @F             ;переход, если a >= b
        mov eax,edx        ;eax = b

@@: cmp eax,r8d            ;сравнение max(a, b) и c
        jge @F
        mov eax,r8d        ;eax = max(a, b, c)

@@: ret
SignedMaxA_ endp

; extern "C" int SignedMinB_(int a, int b, int c);
;
; Возвращает: min(a, b, c)

SignedMinB_ proc
        cmp ecx,edx
        cmovg ecx,edx    ;ecx = min(a, b)
        cmp ecx,r8d
        cmovg ecx,r8d    ;ecx = min(a, b, c)
        mov eax,ecx
        ret
SignedMinB_ endp

; extern "C" int SignedMaxB_(int a, int b, int c);
;
; Возвращает: max(a, b, c)

SignedMaxB_ proc
        cmp ecx,edx
        cmovl ecx,edx   ;ecx = max(a, b)
        cmp ecx,r8d
        cmovl ecx,r8d   ;ecx = max(a, b, c)
        mov eax,ecx
        ret
SignedMaxB_ endp
        end
```

При разработке кода для реализации определенного алгоритма часто бывает необходимо определить минимальное или максимальное значение двух чисел. Стандартная библиотека C++ для выполнения этих операций определяет две шаблонные функции с именами `std::min()` и `std::max()`. Код на языке ассемблера, показанный в листинге 2.7, содержит несколько версий с тремя аргументами для функций минимума и максимума целого числа со знаком. Эти функции предназначены для демонстрации правильного использования команд `jcc` и `cmovcc`. Первая функция, называемая `SignedMinA_`, находит минимальное значение из трех целых чисел со знаком. Первый блок кода определяет `min(a, b)` с помощью двух команд: `cmp eax,ecx` и `jle @F`. Команда `cmp`, которую вы видели ранее в этой главе, вычитает исходный операнд из целевого операнда и устанавливает флаги состояния на основе результата (результат не сохраняется). Операнд `@F` команды `jle` (переход, если меньше или равно) является нотацией ассемблера, которая обозначает ближайшую прямую метку `@@` как цель условного перехода (для обратных переходов может использоваться символ `@B`). После вычисления `min(a,b)` следующий блок кода определяет `min(min(a,b),c)` с использованием той же техники. Если результат уже присутствует в регистре EAX, `SignedMinA_` может вернуться к вызывающей функции.

Функция `SignedMaxA_` использует тот же подход, чтобы найти максимум среди трех целых чисел со знаком. Единственное различие между `SignedMaxA_` и `SignedMinA_` – это использование `jge` (переход, если больше или равно) вместо команды `jle`. Версии `SignedMinA_` и `SignedMaxA_`, которые работают с целыми числами без знака, можно легко создать, изменив команды `jle` и `jge` на `jbe` (перейти, если ниже или равно) и `jae` (перейти, если выше или равно) соответственно. Напомню, в главе 1 я говорил о том, что коды условий, в мнемонике которых используются слова «**g**reater» (больше) и «**l**ess» (меньше), предназначены для целочисленных операндов со знаком, а коды со словами «**a**bove» (выше) и «**b**elow» (ниже) предназначены для целочисленных операндов без знака[2].

Код на языке ассемблера также содержит функции `SignedMinB_` и `SignedMaxB_`. Эти функции определяют минимум и максимум из трех целых чисел со знаком, используя команды условного присвоения вместо условных переходов. Команда `cmovcc` проверяет указанное условие, и, если оно истинно, исходный операнд копируется в целевой операнд. Если указанное условие ложно, операнд назначения не изменяется.

Если вы изучите функцию `SignedMinB_`, то заметите, что после команды `cmp ecx,edx` идет команда `cmovg ecx,edx`. Команда `cmovg` (присвоить, если больше) копирует содержимое EDX в ECX, если ECX больше EDX. В этом примере регистры ECX и EDX содержат значения аргументов `a` и `b`. После выполнения команды `cmovg` регистр ECX содержит `min(a,b)`. Следующая последовательность команд `cmp` и `cmovg` дает `min(a,b,c)`. Тот же метод используется в `SignedMaxB_`, который применяет `cmovl` вместо `cmovg` для сохранения наибольшего целого числа со знаком. Беззнаковые версии этих функций можно легко создать, используя `cmova` и `cmovb` вместо `cmovg` и `cmovl` соответственно. Ниже показан результат работы кода из примера `Ch02_07`.

---

[2] Жирным шрифтом выделены буквы, которые непосредственно входят в мнемонику команды. – *Прим. пер.*

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

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

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

SignedMaxA(  10,   5,   3) =  10
SignedMaxB(  10,   5,   3) =  10

SignedMaxA(  -3,  28,  15) =  28
SignedMaxB(  -3,  28,  15) =  28

SignedMaxA( -25, -37, -17) = -17
SignedMaxB( -25, -37, -17) = -17
```

Использование команды условного перехода для исключения одного или нескольких операторов условного перехода часто приводит к более быстрому выполнению кода, особенно в ситуациях, когда процессор не может точно предсказать, будет ли выполнен переход. Вы узнаете больше о некоторых проблемах, связанных с оптимальным использованием команд условного перехода и условного перемещения, в главе 15.

2.4. Заключение

Вот ключевые моменты главы 2:

- команды `add` и `sub` выполняют сложение и вычитание целых чисел (со знаком и без знака);
- команды `imul` и `idiv` выполняют целочисленное умножение и деление со знаком. Соответствующие команды для целых чисел без знака – `mul` и `div`. Команды `idiv` и `div` обычно требуют, чтобы делимое перед использованием было расширено битом знака или нулем;
- команды `and`, `or`, и `xor` используются для выполнения побитового И, включающего ИЛИ и операции исключающее ИЛИ. Команды `shl` и `shr` выполняют логический сдвиг влево и вправо; `sar` используется для арифметического сдвига вправо;
- почти все арифметические, логические команды и команды сдвига устанавливают флаги состояния, чтобы указать свойства результата операции. Команда `cmp` также устанавливает флаги состояния. Команды `jcc` и `cmovcc` можно использовать для ветвления потока программы или выполнения условного перемещения данных в зависимости от состояния одного или нескольких флагов состояния;
- набор команд x86-64 поддерживает множество различных режимов адресации для доступа к операндам, хранящимся в памяти;

- MASM использует директивы `.code`, `.data` и `.const` для обозначения разделов кода, данных и констант. Директивы `proc` и `endp` обозначают начало и конец функции на языке ассемблера;
- соглашение о вызовах Visual C++ требует, чтобы вызывающая функция использовала регистры RCX, RDX, R8 и R9 (или младшие части этих регистров для значений меньше 64 бит) для первых четырех целочисленных аргументов или аргументов указателя. Дополнительные аргументы передаются в стек;
- чтобы отключить создание расширенных имен компилятором C++, функции на языке ассемблера должны быть объявлены с использованием модификатора `extern "C"`. Глобальные переменные, общие для C++ и кода на языке ассемблера, также должны использовать модификатор `extern "C"`.

Глава 3

Программирование ядра x86-64. Часть 2

В предыдущей главе были представлены основы программирования на языке ассемблера x86-64. Вы узнали, как использовать набор команд x86-64 для выполнения сложения, вычитания, умножения и деления целых чисел. Вы также изучили исходный код, иллюстрирующий использование логических команд, операций сдвига, режимов адресации памяти и условных переходов и присваиваний. Помимо изучения часто используемых команд, первые примеры кода на языке ассемблера x86-64 также содержали важные практические детали, включая директивы ассемблера и требования соглашения о вызовах.

В этой главе вы продолжите изучение основ программирования на языке ассемблера x86-64. Вы узнаете, как использовать дополнительные команды x86-64 и директивы ассемблера. Вы также изучите исходный код, демонстрирующий работу с общими конструкциями программирования, включая массивы и структуры данных. Эта глава завершается несколькими примерами, демонстрирующими использование строковых команд x86.

3.1. Массивы

Массивы являются незаменимой конструкцией данных практически во всех языках программирования. В C++ существует внутренняя связь между массивами и указателями, поскольку имя массива по сути является указателем на его первый элемент. Более того, всякий раз, когда массив используется в качестве параметра функции C++, вместо дублирования массива в стеке передается указатель. Указатели также используются для массивов, которые динамически выделяются во время выполнения. В этом разделе исследуется код языка ассемблера x86-64, обрабатывающий массивы. Первые два примера показывают, как выполнять простые операции с одномерными массивами. Следующие два примера иллюстрируют использование методов, необходимых для доступа к элементам двумерного массива.

3.1.1. Одномерные массивы

В C++ одномерные массивы хранятся в непрерывном блоке памяти, который может быть статически выделен во время компиляции или динамически во

время выполнения программы. Доступ к элементам массива C++ осуществляется с помощью индексации с отсчетом от нуля, то есть допустимые индексы для массива размера N находятся в диапазоне от 0 до N–1. Листинг исходного кода в этом разделе включает в себя примеры выполнения базовых операций с одномерными массивами с использованием набора команд x86-64.

Доступ к элементам массива

В листинге 3.1 показан исходный код примера `Ch03_01`. В этом примере функция `CalcArraySum_` суммирует элементы целочисленного массива. В верхней части кода C++ находится уже знакомое объявление функции на языке ассемблера `CalcArraySum_`. Вычисление суммы, выполняемое этой функцией, дублируется в функции C++ `CalcArraySumCpp` для проверки правильности работы ассемблерной функции.

Листинг 3.1. Пример Ch03_01

```
//-----------------------------------------------
//               Ch03_01.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

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 main()
{
    int x[] {3, 17, -13, 25, -2, 9, -6, 12, 88, -19};
    int n = sizeof(x) / sizeof(int);

    cout << "Элементы массива x" << '\n';

    for (int i = 0; i < n; i++)
        cout << "x[" << i << "] = " << x[i] << '\n';
    cout << '\n';

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

    cout << "sum1 = " << sum1 << '\n';
    cout << "sum2 = " << sum2 << '\n';
    return 0;
}
```

```
;-----------------------------------------------
;                Ch03_01.asm
;-----------------------------------------------
; extern "C" int CalcArraySum_(const int* x, int n)
;
; Возвращает: сумма элементов массива x

        .code
CalcArraySum_ proc

; Начальное значение суммы равно нулю
        xor eax,eax         ;sum = 0

; Проверка, действительно ли n больше нуля
        cmp edx,0
        jle InvalidCount    ;переход, если n <= 0

; Сумма элементов массива
@@:     add eax,[rcx]       ;прибавление элемента к сумме (sum += *x)
        add rcx,4           ;установка указателя на следующий элемент (x++)
        dec edx             ;обновление счетчика (n -= 1)
        jnz @B              ;повторить, если не закончили

InvalidCount:
        ret
CalcArraySum_ endp
        end
```

Функция `CalcArraySum_` начинается с команды `xor eax,eax`, которая инициализирует текущую сумму нулевым значением. Команды `cmp edx,0` и `jle InvalidCount` предотвращают выполнение цикла суммирования, если условие `n <= 0` истинно. Чтобы просуммировать элементы в массиве, требуется всего четыре команды. Команда `add eax,[rcx]` добавляет текущий элемент массива к сумме в регистре EAX. Затем к регистру RCX прибавляется `4`, после чего он указывает на следующий элемент в массиве. Здесь используется константа `4`, поскольку размер каждого целого числа в массиве `x` составляет четыре байта. Команда `dec edx` (уменьшение на 1) вычитает 1 из счетчика и обновляет состояние RFLAGS.ZF. Это позволяет команде `jnz` завершить цикл после суммирования всех `n` элементов. Последовательность команд, используемая здесь для вычисления суммы элементов массива, является ассемблерным эквивалентом цикла `for`, который используется в функции `CalcArraySumCpp`. Вот результат выполнения кода примера `Ch03_01`:

```
Элементы массива x
x[0] = 3
x[1] = 17
x[2] = -13
x[3] = 25
x[4] = -2
x[5] = 9
x[6] = -6
x[7] = 12
```

```
x[8] = 88
x[9] = -19

sum1 = 114
sum2 = 114
```

Использование элементов массива в расчетах

При работе с массивами часто необходимо определять функции, выполняющие поэлементные преобразования. Следующий пример исходного кода иллюстрирует операцию преобразования массива с использованием отдельных массивов источника и приемника. В нем также представлены *прологи* и *эпилоги* функций, а также несколько новых команд. В листинге 3.2 показан исходный код примера `Ch03_02`.

Листинг 3.2. Пример Ch03_02

```
//-----------------------------------------------
//                Ch03_02.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cassert>

using namespace std;

extern "C" long long CalcArrayValues_(long long* y, const int* x, int a, short b, int n);

long long CalcArrayValuesCpp(long long* y, const int* x, int a, short b, int n)
{
    long long sum = 0;

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

    return sum;
}

int main()
{
    const int a = -6;
    const short b = -13;
    const int x[] {26, 12, -53, 19, 14, 21, 31, -4, 12, -9, 41, 7};
    const int n = sizeof(x) / sizeof(int);

    long long y1[n];
    long long y2[n];

    long long sum_y1 = CalcArrayValuesCpp(y1, x, a, b, n);
    long long sum_y2 = CalcArrayValues_(y2, x, a, b, n);
```

```
    cout << "a = " << a << '\n';
    cout << "b = " << b << '\n';
    cout << "n = " << n << "\n\n";

    for (int i = 0; i < n; i++)
    {
        cout << "i: " << setw(2) << i << " ";
        cout << "x: " << setw(6) << x[i] << " ";
        cout << "y1: " << setw(6) << y1[i] << " ";
        cout << "y2: " << setw(6) << y2[i] << '\n';
    }

    cout << '\n';
    cout << "sum_y1 = " << sum_y1 << '\n';
    cout << "sum_y2 = " << sum_y2 << '\n';

    return 0;
}

;-----------------------------------------------
;                  Ch03_02.asm
;-----------------------------------------------
; extern "C" long long CalcArrayValues_(long long* y, const int* x, int a, short b, int n);
;
; Вычисляет: y[i] = x[i] * a + b
;
; Возвращает: сумму элементов в массиве y.

        .code
CalcArrayValues_ proc frame

; Пролог функции
        push rsi        ;сохранение регистра rsi
        .pushreg rsi
        push rdi        ;сохранение регистра rdi
        .pushreg rdi
        .endprolog

; Инициализация суммы нулем и проверка валидности значения n
        xor rax,rax ;sum = 0
        mov r11d,[rsp+56] ;r11d = n
        cmp r11d,0
        jle InvalidCount ;jump if n <= 0

; Инициализация указателей на источник и приемник
        mov rsi,rdx                     ;rsi = ptr на массив x
        mov rdi,rcx                     ;rdi = ptr на массив y

; Присвоение констант и индексов массивов
        movsxd r8,r8d                   ;r8 = a (sign extended)
        movsx r9,r9w                    ;r9 = b (sign extended)
        xor edx,edx                     ;edx = индекс массива i

; Повторение до завершения
@@:     movsxd rcx,dword ptr [rsi+rdx*4] ;rcx = x[i] (sign extended)
        imul rcx,r8                     ;rcx = x[i] * a
```

```
        add rcx,r9                      ;rcx = x[i] * a + b
        mov qword ptr [rdi+rdx*8],rcx   ;y[i] = rcx
        add rax,rcx                     ;обновление значения суммы
        inc edx                         ;edx = i + i
        cmp edx,r11d                    ;проверка, i >= n?
        jl @B                           ;переход, если i < n

InvalidCount:

; Эпилог функции
        pop rdi             ;восстановление rdi вызывающей функции
        pop rsi             ;восстановление rsi вызывающей функции
        ret
CalcArrayValues_ endp
        end
```

Функция на языке ассемблера x86-64 `CalcArrayValues_` вычисляет выражение `y[i] = x[i]*a+b`. Если вы изучите объявление этой функции в коде C++, то заметите, что исходный массив `x` объявлен как `int`, а целевой массив `y` объявлен как `long`. Остальные аргументы функции `a`, `b` и `n` объявлены как `int`, `short` и `int` соответственно. Остальная часть кода C++ содержит функцию `CalcArrayValuesCpp`, которая также вычисляет указанное преобразование массива для сравнения с ассемблерной функцией. Далее следует код для отображения результатов.

Вы могли заметить, что во всех представленных до сих пор примерах исходного кода использовалась только часть регистров общего назначения. Причина этого в том, что соглашение о вызовах Visual C++ определяет каждый регистр общего назначения как энергозависимый или энергонезависимый. Функциям разрешено использовать и изменять содержимое любого энергозависимого регистра, но нельзя использовать энергонезависимый регистр, если он не сохраняет исходное значение вызывающей функции. Соглашение о вызовах Visual C++ обозначает регистры RAX, RCX, RDX, R8, R9, R10 и R11 как энергозависимые, а остальные регистры общего назначения как энергонезависимые.

Функция `CalcArrayValues_` использует энергонезависимые регистры RSI и RDI, что означает необходимость сохранения их значений. Функция обычно сохраняет значения всех энергонезависимых регистров, которые она использует в стеке, в разделе кода, называемом *прологом*. В свою очередь, *эпилог* функции содержит код, восстанавливающий значения всех сохраненных энергонезависимых регистров. Прологи и эпилоги функций также нужны для выполнения других задач инициализации в соответствии с соглашением о вызовах, о которых вы узнаете в главе 5.

В ассемблерной части примера `Ch03_02` оператор `CalcArrayValues_ proc frame` обозначает начало функции `CalcArrayValues_`. Обратите внимание на атрибут `frame` в директиве `proc`. Этот атрибут указывает, что `CalcArrayValues_` использует формальный пролог функции. Он также включает дополнительные директивы, которые должны использоваться всякий раз, когда регистр общего назначения сохраняется в стеке, или всякий раз, когда функция использует указатель фрейма стека. В главе 5 более подробно обсуждаются атрибут `frame` и указатели фрейма стека.

Первая команда языка ассемблера x86-64 `CalcArrayValues_` – это `push rsi` (поместить значение в стек), которая сохраняет текущее значение в регистре RSI в стеке. Сразу после этого следует директива `.pushreg rsi`. Эта директива указы-

вает ассемблеру сохранять информацию о команде `push rsi` в поддерживаемой ассемблером таблице, которая используется для развертывания стека во время обработки исключения. Использование исключений с кодом на языке ассемблера в этой книге не обсуждается, но все же необходимо соблюдать требования соглашения о вызовах для сохранения регистров в стеке. Затем с помощью команды `push rdi` сохраняется в стеке регистр RDI. За ней следует необходимая директива `.pushreg rdi`, а последующая директива `.endprolog` означает конец пролога для `CalcArrayValues_`.

На рис. 3.1 показано содержимое стека после выполнения команд `push rsi` и `push rdi`. Следом за прологом функции идет проверка аргумента `n` на допустимость. А команда `mov r11d,[rsp + 56]` загружает значение `n` в регистр R11D. Важно отметить, что смещение, используемое в этой команде для загрузки `n` из стека, отличается от такового в предыдущих примерах из-за команд `push`, которые применялись в прологе. Если значение `n` является допустимым, регистры RSI и RDI инициализируются как указатели на массивы `x` и `y`. Команды `movsxd r8,r8d` и `movsx r9,r9w` загружают значения аргументов `a` и `b` в регистры R8 и R9, а команда `xor edx,edx` инициализирует индекс массива `i` нулевым значением.

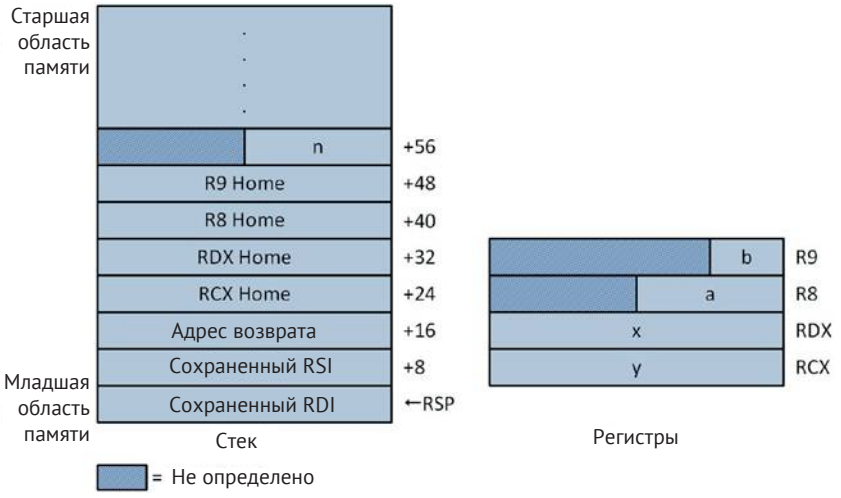


Рис. 3.1. Содержимое стека и регистров после пролога в `CalcArrayValues`

Цикл обработки `CalcArrayValues_` использует команду `movsxd rcx,dword ptr [rsi+rdx*4]` для загрузки расширенной знаковым битом копии `x[i]` в регистр RCX. Следующие команды `imul rcx,r8` и `add rcx,r9` вычисляют `x[i]*a+b`, а команда `mov qword ptr [rdi+rdx*8]` сохраняет окончательный результат в `y[i]`. Обратите внимание, что в цикле обработки две команды перемещения используют разные коэффициенты масштабирования. Это потому, что массив `x` и массив `y` объявлены как `int` и `long long`. Команда `add rax,rcx` обновляет текущую сумму, которая будет использоваться в качестве возвращаемого значения. Команда `inc edx` (увеличение на 1) добавляет 1 к значению в регистре EDX. Она также обнуляет биты 63:32 регистра RDX. Причина использования команды `inc edx` вместо команды `inc rdx` заключается в том, что первая на машинном языке требует меньше места для кода. Что еще более важно, здесь можно использо-

вать команду `inc edx`, поскольку максимальное количество обрабатываемых элементов определяется 32-битным целым числом со знаком `n`, которое заведомо больше нуля (уже проверено раньше). Следующая команда `cmp edx,r11d` сравнивает содержимое EDX (т. е. `i`) с `n`, и цикл обработки повторяется до тех пор, пока `i` не станет равным `n`.

После основного цикла обработки следует эпилог для функции `CalcArrayValues_`. Напомним, что в прологе регистры RSI и RDI вызывающего были сохранены в стеке с помощью двух команд `push`. В эпилоге команды `pop rdi` и `pop rsi` (выгрузить значения из стека) используются для восстановления регистров RDI и RSI вызывающего абонента. Порядок, в котором энергонезависимый регистр вызывающего абонента извлекается из стека в эпилоге, должен быть обратным тому, в каком они были сохранены в прологе. После восстановления энергонезависимого регистра следует команда `ret`, которая передает управление программой обратно вызывающей функции. Учитывая операции стека, которые происходят в прологе и эпилоге функции, должно быть очевидно, что неудачное сохранение или восстановление энергонезависимого регистра может вызвать сбой программы (если адрес возврата неверен) или небольшую программную ошибку, наличие которой трудно обнаружить. Вот результаты выполнения кода примера `Ch03_02`.

```
a = -6
b = -13
n = 12
  i: 0   x:     26    yl:    -169    y2:   -169
  i: 1   x:     12    yl:     -85    y2:    -85
  i: 2   x:    -53    yl:     305    y2:    305
  i: 3   x:     19    yl:    -127    y2:   -127
  i: 4   x:     14    yl:     -97    y2:    -97
  i: 5   x:     21    yl:    -139    y2:   -139
  i: 6   x:     31    yl:    -199    y2:   -199
  i: 7   x:     -4    yl:      11    y2:     11
  i: 8   x:     12    yl:     -85    y2:    -85
 i: 9   x:     -9    yl:      41    y2:     41
 i: 10  x:     41    yl:    -259    y2:   -259
 i: 11  x:      7    yl:     -55    y2:    -55

sum_y1 = -858
sum_y2 = -858
```

3.1.2. Двумерные массивы

C++ также использует непрерывный блок памяти для реализации *двумерного массива*, или *матрицы*. Элементы матрицы C++ в памяти упорядочены *по строкам*. При таком упорядочивании элементы матрицы располагаются сначала по строкам, а затем по столбцам. Например, элементы матрицы `int x[3][2]`

хранятся в памяти следующим образом: x[0][0], x[0][1], x[1][0], x[1][1], x[2][0] и x[2][1]. Чтобы получить доступ к конкретному элементу в матрице, функция (или компилятор) должна знать начальный адрес матрицы (то есть адрес ее первого элемента), индексы строки и столбца, общее количество столбцов и размер каждого элемента в байтах. Используя эту информацию, функция может использовать простую арифметику для доступа к определенному элементу в матрице, как показано в примере кода в этом разделе.

Доступ к элементам матрицы

В листинге 3.3 показан исходный код примера Ch03_03, который демонстрирует, как использовать язык ассемблера x86-64 для доступа к элементам матрицы. В этом примере функции CalcMatrixSquaresCpp и CalcMatrixSquares_ выполняют следующее вычисление матрицы: y[i][j] = x[j][i]*x[j][i]. Обратите внимание, что в данном выражении индексы i и j для матрицы x намеренно перевернуты, чтобы сделать код этого примера немного более интересным.

Листинг 3.3. Пример Ch03_03

```
//-----------------------------------------------
//               Ch03_03.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" void CalcMatrixSquares_(int* y, const int* x, int nrows, int ncols);

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

int main()
{
    const int nrows = 6;
    const int ncols = 3;
    int y2[nrows][ncols];
    int y1[nrows][ncols];
    int x[nrows][ncols] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 },
                          { 10, 11, 12 }, {13, 14, 15}, {16, 17, 18} };

    CalcMatrixSquaresCpp(&y1[0][0], &x[0][0], nrows, ncols);
    CalcMatrixSquares_(&y2[0][0], &x[0][0], nrows, ncols);
```

```
    for (int i = 0; i < nrows; i++)
    {
        for (int j = 0; j < ncols; j++)
        {
            cout << "y1[" << setw(2) << i << "][" << setw(2) << j << "] = ";
            cout << setw(6) << y1[i][j] << ' ' ;
            cout << "y2[" << setw(2) << i << "][" << setw(2) << j << "] = ";
            cout << setw(6) << y2[i][j] << ' ';
            cout << "x[" << setw(2) << j << "][" << setw(2) << i << "] = ";
            cout << setw(6) << x[j][i] << '\n';
            if (y1[i][j] != y2[i][j])
                cout << "Обнаружено несовпадение\n";
        }
    }

    return 0;
}

;-----------------------------------------------
;                Ch03_03.asm
;-----------------------------------------------
; void CalcMatrixSquares_(int* y, const int* x, int nrows, int ncols);
;
; Вычисляет: y[i][j] = x[j][i] * x[j][i]

        .code
CalcMatrixSquares_ proc frame

; Пролог функции
        push rsi          ;сохранение rsi вызывающей функции
        .pushreg rsi
        push rdi          ;сохранение rdi вызывающей функции
        .pushreg rdi
        .endprolog

; Проверка nrows и ncols на допустимость значений
        cmp r8d,0
        jle InvalidCount          ;переход, если nrows <= 0
        cmp r9d,0
        jle InvalidCount        ;переход, если ncols <= 0

; Инициализация указателей на массивы источника и приемника
        mov rsi,rdx             ;rsi = x
        mov rdi,rcx             ;rdi = y
        xor rcx,rcx             ;rcx = i
        movsxd r8,r8d           ;r8 = расширенный знаком nrows
        movsxd r9,r9d           ;r9 = расширенный знаком ncols

; Выполнение нужных вычислений
Loop1:
        xor rdx,rdx                     ;rdx = j
Loop2:
        mov rax,rdx                     ;rax = j
        imul rax,r9                     ;rax = j * ncols
        add rax,rcx                     ;rax = j * ncols + i
        mov r10d,dword ptr [rsi+rax*4] ;r10d = x[j][i]
```

```
        imul r10d,r10d ;r10d = x[j][i] * x[j][i]

        mov rax,rcx                    ;rax = i
        imul rax,r9                    ;rax = i * ncols
        add rax,rdx                    ;rax = i * ncols + j;
        mov dword ptr [rdi+rax*4],r10d ;y[i][j] = r10d

        inc rdx                        ;j += 1
        cmp rdx,r9
        jl Loop2                       ;переход, если j < ncols

        inc rcx ;i += 1
        cmp rcx,r8
        jl Loop1                       ;переход, если i < nrows
        InvalidCount:

; Эпилог функции
        pop rdi                        ;восстановление rdi
        pop rsi                        ;восстановление rsi
        ret

CalcMatrixSquares_ endp
        end
```

Функция C++ `CalcMatrixSquaresCpp` показывает, как получить доступ к элементам матрицы. Прежде всего следует отметить, что аргументы `x` и `y` указывают на блоки памяти, которые содержат соответствующие им матрицы. Внутри второго цикла `for` выражение `kx = j*ncols+i` вычисляет смещение, необходимое для доступа к элементу `x[j][i]`. Точно так же выражение `ky = i*ncols+j` вычисляет смещение для элемента `y[i][j]`.

Функция языка ассемблера `CalcMatrixSquares_` выполняет те же вычисления, что и код C++, для доступа к элементам в матрицах `x` и `y`. Эта функция начинается с пролога, который сохраняет энергонезависимые регистры RSI и RDI с использованием тех же команд и директив, что и в предыдущем примере исходного кода. Затем следует проверка значений аргументов `nrows` и `ncols`, чтобы убедиться, что они больше нуля. Перед началом вложенных циклов обработки регистры RSI и RDI инициализируются как указатели на `x` и `y`. Регистры RCX и RDX также используются как переменные индекса цикла и выполняют те же функции, что и переменные `i` и `j` в коде C++. За этим следуют две команды `movsxd`, которые загружают расширенные по знаку копии `nrows` и `ncols` в регистры R8 и R9.

Раздел кода, который обращается к элементу `x[j][i]`, начинается с команды `mov rax,rdx`, которая копирует `j` в регистр RAX. Далее следуют команды `imul rax,r9` и `add rax,rcx`, которые вычисляют значение `j*ncols+i`. Следующая команда `mov r10d,dword ptr [rsi+rax*4]` загружает регистр R10D с `x[j][i]`, а команда `imul r10d, r10d` возводит это значение в квадрат. Аналогичная последовательность команд используется для вычисления смещения `i*ncols+j`, необходимого для `y[i][j]`. Команда `mov dword ptr [rdi+rax*4],r10d` завершает выполнение выражения `y[i][j] = x[j][i]*x[j][i]`. Как и соответствующий код C++, вложенные циклы обработки в `CalcMatixSquares_` продолжают выполняться до тех пор, пока счет-

чики индексов j и i (регистры RDX и RCX) не достигнут своих соответствующих значений завершения. Последние две команды pop восстанавливают регистры RDI и RSI из стека до выполнения команды ret. Так выглядит результат выполнения кода примера Ch03_03:

```
y1[ 0][ 0] =          1 y2[ 0][ 0] =          1 x[ 0][ 0] =      1
y1[ 0][ 1] =         16 y2[ 0][ 1] =         16 x[ 1][ 0] =      4
y1[ 0][ 2] =         49 y2[ 0][ 2] =         49 x[ 2][ 0] =      7
y1[ 1][ 0] =          4 y2[ 1][ 0] =          4 x[ 0][ 1] =      2
y1[ 1][ 1] =         25 y2[ 1][ 1] =         25 x[ 1][ 1] =      5
y1[ 1][ 2] =         64 y2[ 1][ 2] =         64 x[ 2][ 1] =      8
y1[ 2][ 0] =          9 y2[ 2][ 0] =          9 x[ 0][ 2] =      3
y1[ 2][ 1] =         36 y2[ 2][ 1] =         36 x[ 1][ 2] =      6
y1[ 2][ 2] =         81 y2[ 2][ 2] =         81 x[ 2][ 2] =      9
y1[ 3][ 0] =         16 y2[ 3][ 0] =         16 x[ 0][ 3] =      4
y1[ 3][ 1] =         49 y2[ 3][ 1] =         49 x[ 1][ 3] =      7
y1[ 3][ 2] =        100 y2[ 3][ 2] =        100 x[ 2][ 3] =     10
y1[ 4][ 0] =         25 y2[ 4][ 0] =         25 x[ 0][ 4] =      5
y1[ 4][ 1] =         64 y2[ 4][ 1] =         64 x[ 1][ 4] =      8
y1[ 4][ 2] =        121 y2[ 4][ 2] =        121 x[ 2][ 4] =     11
y1[ 5][ 0] =         36 y2[ 5][ 0] =         36 x[ 0][ 5] =      6
y1[ 5][ 1] =         81 y2[ 5][ 1] =         81 x[ 1][ 5] =      9
y1[ 5][ 2] =        144 y2[ 5][ 2] =        144 x[ 2][ 5] =     12
```

Расчеты по строкам и столбцам

В листинге 3.4 показан исходный код примера Ch03_04, который показывает, как суммировать строки и столбцы матрицы. Код C++ в листинге 3.4 включает пару вспомогательных функций с именами Init и PrintResult, выполняющих инициализацию матрицы и отображающих результаты. Функция CalcMatrixRowColSumsCpp иллюстрирует алгоритм суммирования. Эта функция просматривает матрицу x, используя набор вложенных циклов for. Во время каждой итерации она прибавляет элемент матрицы x[i][j] к соответствующим записям в массивах row_sums и col_sums. Функция CalcMatrixRowColSumsCpp также использует ту же арифметику, которую вы видели в предыдущем примере, для определения смещения каждого элемента матрицы.

Листинг 3.4. Пример Ch03_04

```
//-----------------------------------------------
//               Ch03_04.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>

using namespace std;

extern "C" int CalcMatrixRowColSums_(int* row_sums, int* col_sums, const int* x, int nrows,
int ncols);

void Init(int* x, int nrows, int ncols)
{
    unsigned int seed = 13;
    uniform_int_distribution<> d {1, 200};
    default_random_engine rng {seed};

    or (int i = 0; i < nrows * ncols; i++)
        x[i] = d(rng);
}

void PrintResult(const char* msg, const int* row_sums, const int* col_sums, const int* x,
int nrows, int ncols)
{
    const int w = 6;
    const char nl = '\n';

    cout << msg;
    cout << "----------------------------------------\n";

    for (int i = 0; i < nrows; i++)
    {
        for (int j = 0; j < ncols; j++)
                cout << setw(w) << x[i* ncols + j];
            cout << " " << setw(w) << row_sums[i] << nl;
    }

    cout << nl;

    for (int i = 0; i < ncols; i++)
        cout << setw(w) << col_sums[i];
    cout << nl;
}

int CalcMatrixRowColSumsCpp(int* row_sums, int* col_sums, const int* x, int nrows, int
ncols)
{
    int rc = 0;

    if (nrows > 0 && ncols > 0)
    {
        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;
            }
        }

        rc = 1;
    }

    return rc;
}

int main()
{
    const int nrows = 7;
    const int ncols = 5;
    int x[nrows][ncols];

    Init((int*)x, nrows, ncols);

    int row_sums1[nrows], col_sums1[ncols];
    int row_sums2[nrows], col_sums2[ncols];

    const char* msg1 = "\nРезультаты CalcMatrixRowColSumsCpp\n";
    const char* msg2 = "\nРезультаты CalcMatrixRowColSums_\n";

    int rc1 = CalcMatrixRowColSumsCpp(row_sums1, col_sums1, (int*)x, nrows, ncols);
    int rc2 = CalcMatrixRowColSums_(row_sums2, col_sums2, (int*)x, nrows, ncols);

    if (rc1 == 0)
        cout << "Ошибка CalcMatrixRowSumsCpp\n";
    else
        PrintResult(msg1, row_sums1, col_sums1, (int*)x, nrows, ncols);

    if (rc2 == 0)
        cout << "Ошибка CalcMatrixRowSums_\n";
    else
        PrintResult(msg2, row_sums2, col_sums2, (int*)x, nrows, ncols);

    return 0;
}

;-----------------------------------------------
;               Ch03_04.asm
;-----------------------------------------------

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

```
; Возвращает: 0 = nrows <= 0 или ncols <= 0, 1 = успех

        .code
CalcMatrixRowColSums_ proc frame

; Пролог функции
        push rbx                        ;сохранение rbx
        .pushreg rbx
        push rsi                        ;сохранение rsi
        .pushreg rsi
        push rdi                        ;сохранение rdi
        .pushreg rdi
        .endprolog

; Проверка значений nrows и ncols на допустимость
        xor eax,eax                     ;присвоение кода ошибки (ноль)
        cmp r9d,0
        jle InvalidArg                  ;переход, если nrows <= 0
        mov r10d,[rsp+64]               ;r10d = ncols
        cmp r10d,0
        jle InvalidArg                  ;переход, если ncols <= 0

; Инициализация элементов массива col_sums нулями
        mov rbx,rcx                     ;временное сохранение row_sums
        mov rdi,rdx                     ;rdi = col_sums
        mov ecx,r10d                    ;rcx = ncols
        xor eax,eax                     ;eax = обнуление
        rep stosd                       ;заполнение массива нулями

; Дальнейший код использует следующие регистры:
; rcx = row_sums rdx = col_sums
; r9d = nrows r10d = ncols
; eax = i ebx = j
; edi = i * ncols esi = i * ncols + j
; r8 = x r11d = x[i][j]

; Инициализация переменных внешнего цикла
        mov rcx,rbx                     ;rcx = row_sums
        xor eax,eax                     ;i = 0

Lp1: mov dword ptr [rcx+rax*4],0        ;row_sums[i] = 0
        xor ebx,ebx                     ;j = 0
        mov edi,eax                     ;edi = i
        imul edi,r10d                   ;edi = i * ncols

; Внутренний цикл
Lp2:    mov esi,edi                     ;esi = i * ncols
        add esi,ebx                     ;esi = i * ncols + j
        mov r11d,[r8+rsi*4]             ;r11d = x[i * ncols + j]
        add [rcx+rax*4],r11d            ;row_sums[i] += x[i * ncols + j]
        add [rdx+rbx*4],r11d            ;col_sums[j] += x[i * ncols + j]

; Завершен ли внутренний цикл?
        inc ebx                         ;j += 1
        cmp ebx,r10d
        jl Lp2                          ;переход, если j < ncols
```

```
; Завершен ли внешний цикл?
        inc eax                         ;i += 1
        mp eax,r9d
        jl Lp1                          ;переход, если i < nrows
        mov eax,1                       ;установка кода успешной операции

; Function epilog
InvalidArg:
        pop rdi                         ;восстановление регистров и возврат
        pop rsi
        pop rbx
        ret

CalcMatrixRowColSums_ endp
        end
```

Функция языка ассемблера `CalcMatrixRowColSums_` реализует тот же алгоритм, что и код C++. Сразу после пролога функции значения аргументов `nrows` и `ncols` проверяются на допустимость. Обратите внимание, что аргумент `ncols` был передан в стек, как показано на рис. 3.2. Затем элементы `col_sums` инициализируются нулем с помощью команды `rep stosd` (запись строки двойных слов). Эта команда сохраняет содержимое EAX, которое было инициализировано нулем, в ячейку памяти, указанную RDI; затем она прибавляет 4 к RDI, после чего он указывает на следующий элемент массива. Мнемоника `rep` – это префикс команды, который сообщает процессору, что нужно повторить выполнение команды `stosd`. В частности, этот префикс инструктирует ЦП уменьшать RCX на 1 после каждого действия сохранения и повторять выполнение команды `stosd` до тех пор, пока RCX не станет равным нулю. Далее в этой главе вы более подробно ознакомитесь с командами по обработке строк x86-64.

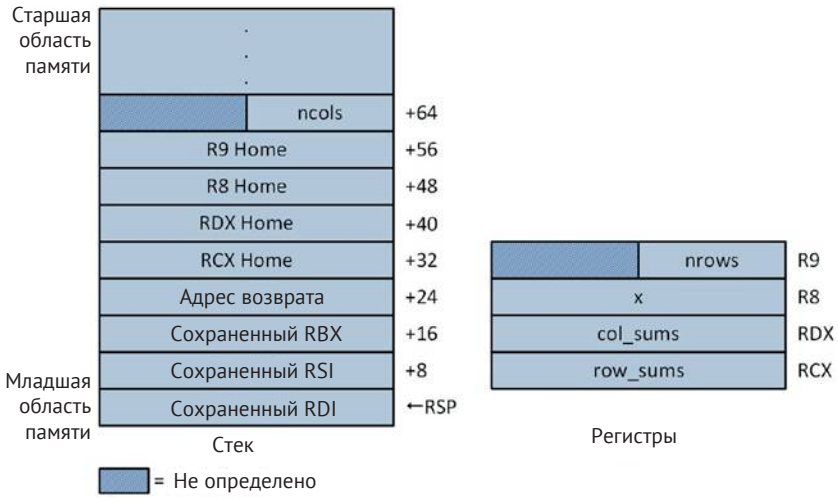


Рис. 3.2. Содержимое стека и регистров после пролога в `CalcMatrixRowColSums`

В функции `CalcMatrixRowColSums_` R8 содержит базовый адрес матрицы `x`. Регистры EAX и EBX содержат индексы строки и столбца `i` и `j` соответственно.

Каждый внешний цикл начинается с инициализации `row_sums[i]` (RCX указывает на `row_sums`) нулевым значением и вычисления промежуточного значения `i*ncols` (R10D содержит `ncols`). Во внутреннем цикле вычисляется окончательное смещение элемента матрицы `x[i][j]`. Команда `mov r11d,[r8+rsi*4]` загружает `x[i][j]` в R11D. Команды `add [rcx+rax*4],r11d` и `add [rdx+rbx*4],r11d` обновляют суммы для `row_sums[i]` и `col_sums[j]`. Обратите внимание, что эти две команды вместо регистров используют операнды-приемники, расположенные в памяти. Рисунок 3.3 иллюстрирует адресацию памяти, которая используется для ссылки на элементы в `x`, `row_sums` и `col_sums`.

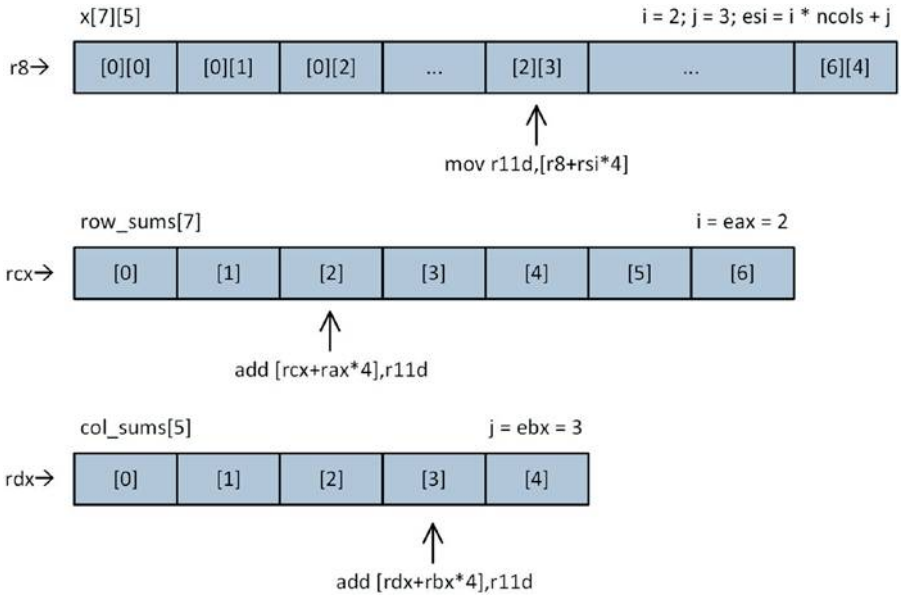


Рис. 3.3. Адресация памяти, используемая в функции `CalcMatrixRowColSums_`

Вложенные циклы обработки в `CalcMatrixRowColSums_` повторяются до тех пор, пока все элементы в матрице `x` не будут прибавлены к нужным элементам в `row_sums` и `col_sums`. Обратите внимание, что эта функция использует 32-битные регистры для своих счетчиков и индексов. Как было сказано ранее в этой главе, использование 32-битных регистров часто требует меньше места для кода, чем 64-битных регистров. Код в `CalcMatrixRowColSums_` также использует адресацию памяти типа `BaseReg+IndexReg*ScaleFactor`, которая упрощает загрузку элементов из матрицы `x` и обновление элементов как в `row_sums`, так и `col_sums`. Вот результаты выполнения примера `Ch03_04`:

```
Результаты CalcMatrixRowColSumsCpp
----------------------------------------
    19   153  155  177  119  623
    27    37  130  165   99  458
    68    27   61    7  195  358
    127  143  110   86   43  509
    114   84  109  179   17  503
```

```
    140  126   28   52   55  401
    126  100  186  115  145  672

    621  670  779  781  673

Результаты CalcMatrixRowColSums_
----------------------------------------
    19   153  155  177  119  623
    27    37  130  165   99  458
    68    27   61    7  195  358
    127  143  110   86   43  509
    114   84  109  179   17  503
    140  126   28   52   55  401
    126  100  186  115  145  672

    621  670  779  781  673
```

Структуры

Структура – это конструкция языка программирования, которая упрощает определение новых типов данных с использованием одного или нескольких существующих типов данных. В этом разделе вы узнаете, как определять и использовать общую структуру в функциях C++ и на языке ассемблера x86-64. Вы также узнаете, как справляться с потенциальными семантическими проблемами, которые могут возникнуть при работе с общей структурой, которую используют программные функции, написанные на разных языках программирования.

В C++ структура эквивалентна классу. Когда тип данных определяется с использованием ключевого слова `struct` вместо `class`, все члены по умолчанию являются общедоступными. Структура C++, объявленная без каких-либо функций-членов или операторов, эквивалентна структуре в стиле C, такой как `typedef struct {...} MyStruct;`. Объявления структуры C++ обычно помещаются в файл заголовка с расширением `.h`, поэтому на них можно легко ссылаться в нескольких файлах C++. Тот же метод можно использовать для объявления структур, используемых в коде на языке ассемблера, и ссылки на них. К сожалению, невозможно объявить единую структуру в файле заголовка и включить этот файл в исходный код как на C++, так и на языке ассемблера. Если вы хотите использовать «одинаковую» структуру как в C++, так и в коде на языке ассемблера, ее необходимо объявить дважды, и оба объявления должны быть семантически эквивалентными.

В листинге 3.5 показан исходный код примера `Ch03_05` на C++ и языке ассемблера x86. В коде C++ объявлена простая структура с именем `TestStruct`. Эта структура использует целочисленные типы размера вместо более распространенных типов C++, чтобы выделить точный размер каждого члена. Другая примечательная деталь относительно `TestStruct` – это включение члена структуры `Pad8`. Хотя это и не требуется явно, присутствие данного члена помогает документировать тот факт, что компилятор C++ по умолчанию выравнивает элементы структуры по их естественным границам. Версия `TestStruct` на ассемблере похожа на свой аналог C++. Самая большая разница между ними заключается в том, что ассемблер *не выравнивает* автоматически элементы структуры по их естественным границам. Здесь требуется определение `Pad8`;

без члена Pad8 версии C++ и ассемблера были бы семантически разными. Символ ?, который включен в каждое объявление элемента данных, уведомляет ассемблер о необходимости ограничиться выделением памяти и обычно используется, чтобы напомнить программисту, что элементы структуры всегда не инициализированы.

Листинг 3.5. Пример Ch03_05

```
//-----------------------------------------------
//                Ch03_05.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cstdint>

using namespace std;

struct TestStruct
{
    int8_t Val8;
    int8_t Pad8;
    int16_t Val16;
    int32_t Val32;
    int64_t Val64;
};

extern "C" int64_t CalcTestStructSum_(const TestStruct* ts);

int64_t CalcTestStructSumCpp(const TestStruct* ts)
{
    return ts->Val8 + ts->Val16 + ts->Val32 + ts->Val64;
}

int main()
{
    TestStruct ts;

    ts.Val8 = -100;
    ts.Val16 = 2000;
    ts.Val32 = -300000;
    ts.Val64 = 40000000000;
    int64_t sum1 = CalcTestStructSumCpp(&ts);
    int64_t sum2 = CalcTestStructSum_(&ts);
    cout << "ts1.Val8 = " << (int)ts.Val8 << '\n';
    cout << "ts1.Val16 = " << ts.Val16 << '\n';
    cout << "ts1.Val32 = " << ts.Val32 << '\n';
    cout << "ts1.Val16 = " << ts.Val64 << '\n';
    cout << '\n';
    cout << "sum1 = " << sum1 << '\n';
    cout << "sum2 = " << sum2 << '\n';

return 0;
}
```

```
;-----------------------------------------------
;               Ch03_05.asm
;-----------------------------------------------

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

; extern "C" int64_t CalcTestStructSum_(const TestStruct* ts);
;
; Возвращает сумму переменных структуры как 64-битное целое число

        .code
CalcTestStructSum_ proc

; Вычисление ts->Val8 + ts->Val16 с расширением знаковым разрядом до 32-битного
        movsx eax,byte ptr [rcx+TestStruct.Val8]
        movsx edx,word ptr [rcx+TestStruct.Val16]
        add eax,edx

; Расширение предыдущего значения знаковым разрядом до 64-битного
        movsxd rax,eax

; Прибавление ts->Val32 к сумме
        movsxd rdx,[rcx+TestStruct.Val32]
        add rax,rdx

; Прибавление ts->Val64 к сумме
        add rax,[rcx+TestStruct.Val64]
        ret

CalcTestStructSum_ endp
        end
```

Функция C++ `CalcTestStructSumCpp` суммирует элементы переданного ей экземпляра `TestStruct`. Функция языка ассемблера x86 `CalcTestStructSum_` выполняет ту же операцию. Команды `movsx eax,byte ptr [rcx+TestStruct.Val8]` и `movsx edx,word ptr [rcx+TestStruct.Val16]` загружают расширенные знаковым разрядом копии членов структуры `TestStruct.Val8` и `TestStruct.Val16` в регистры EAX и EDX соответственно. Эти команды также иллюстрируют синтаксис, который требуется для ссылки на член структуры в команде на языке ассемблера. С точки зрения ассемблера, команды `movsx` являются экземплярами адресации памяти `BaseReg+Disp`, поскольку ассемблер в конечном итоге преобразует элементы структуры `TestStruct.Val8` и `TestStruct.Val16` в постоянные значения смещения.

Затем функция `CalcTestStructSum_` использует команду `add eax,edx` для суммирования элементов структуры `TestStruct.Val8` и `TestStruct.Val16`. Далее она расширяет эту сумму до 64 бит с помощью команды `movsxd rax,eax`. Следующая команда, `movsxd rdx,[rcx+TestStruct.Val32]`, загружает расширенную знаковым битом копию `TestStruct.Val32` в RDX и прибавляет это значение к промежу-

точной сумме в RAX. Команда `add rax, [rcx+TestStruct.Val64]` прибавляет член структуры значений `TestStruct.Val64` к текущей сумме в RAX, которая образует окончательный результат. Соглашение о вызовах Visual C++ требует, чтобы 64-разрядные возвращаемые значения помещались в регистр RAX. Поскольку конечный результат уже находится в требуемом регистре, никаких дополнительных команд `mov` не требуется. Ниже показан результат выполнения кода примера `Ch03_05`:

```
ts1.Val8 = -100
ts1.Val16 = 2000
ts1.Val32 = -300000
ts1.Val16 = 40000000000

sum1 = 39999701900
sum2 = 39999701900
```

3.3. Строки

Набор команд x86-64 включает несколько полезных команд, которые обрабатывают строки и манипулируют ими. На языке x86 строка – это непрерывная последовательность байтов, слов, двойных или четверных слов. Программы могут использовать строковые команды x86 для обработки обычных текстовых строк, таких как «Hello, World». Их также можно использовать для выполнения операций с использованием элементов массива или аналогично упорядоченных данных в памяти. В этом разделе вы изучите примеры кода, демонстрирующие, как использовать строковые команды x86-64 при работе с текстовыми строками и целочисленными массивами.

3.3.1. Подсчет символов

В листинге 3.6 показан код примера `Ch03_06` на C++ и на языке ассемблера. В этом примере объясняется, как использовать команду `lodsb` (загрузить строку байтов) для подсчета количества вхождений символов в текстовой строке.

Листинг 3.6. Пример Ch03_06

```
//-----------------------------------------------
// Ch03_06.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>

using namespace std;

extern "C" unsigned long long CountChars_(const char* s, char c);

int main()
{
    const char nl = '\n';
    const char* s0 = "Пример строки: ";
```

```
    const char* s1 = " SearchChar: ";
    const char* s2 = " Count: ";

    char c;
    const char* s;

    s = "Four score and seven seconds ago , ...";
    cout << nl << s0 << s << nl;

    c = 's';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = 'o';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = 'z';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = 'F';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = '.';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;

    s = "Red Green Blue Cyan Magenta Yellow ";
    cout << nl << s0 << s << nl;

    c = 'e';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = 'w';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = 'l';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = 'Q';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;
    c = 'n';
    cout << s1 << c << s2 << CountChars_(s, c) << nl;

    return 0;
}

;-----------------------------------------------
;                  Ch03_06.asm
;-----------------------------------------------

; extern "C" unsigned long long CountChars_(const char* s, char c);
;
; Описание: эта функция подсчитывает количество вхождений
; символа в строку
;
; Возвращает: количество найденных вхождений

        .code
CountChars_ proc frame

; Сохранение энергонезависимых регистров
        push rsi                        ;сохранение rsi
        .pushreg rsi
        .endprolog
```

```
; Загрузка параметров и инициализация регистров счетчика
        mov rsi,rcx                     ;rsi = s
        mov cl,dl                       ;cl = c
        xor edx,edx                     ;rdx = количество вхождений
        xor r8d,r8d                     ;r8 = 0 (нужно для сложения ниже)

; Повтор цикла, пока не будет просканирована вся строка
@@:     lodsb                           ;загрузка следующего символа в регистр al
        or al,al                        ;проверка на конец строки
        jz @F                           ;переход, если достигнут конец строки
        cmp al,cl                       ;проверка, текущего символа
        sete r8b                        ;r8b = 1 если совпадает, 0 в ином случае
        add rdx,r8                      ;обновление счетчика вхождений
        jmp @B

@@: mov rax,rdx                         ;rax = количество вхождений

; Восстановление энергонезависимых регистров и возврат
        pop rsi
        ret

CountChars_ endp
        end
```

Функция языка ассемблера `CountChars_` принимает два аргумента: указатель текстовой строки `s` и искомый символ `c`. Оба аргумента имеют тип `char`, то есть для каждого символа текстовой строки и искомого символа требуется один байт памяти. Функция `CountChars_` начинается с пролога, который сохраняет RSI вызывающего абонента в стеке. Затем он загружает указатель текстовой строки `s` в RSI и искомый символ `c` в регистр CL. Команда `xor edx,edx` обнуляет регистр RDX перед использованием в качестве счетчика появления символов. Цикл обработки использует команду `lodsb` для чтения каждого символа текстовой строки. Эта команда загружает регистр AL содержимым памяти, на которую указывает RSI; потом он увеличивает RSI на единицу, чтобы указать на следующий символ.

Затем функция `CountChars_` использует команду `or al,al` для проверки символа конца строки (`'\0'`). Эта команда устанавливает нулевой флаг (RFLAGS.ZF), если регистр AL равен нулю. Если символ конца строки не найден, команда `cmp al,cl` сравнивает текущий символ текстовой строки с искомым символом. Последующие команды `sete r8b` (установить байт, если равно) загружают в регистр R8B единицу, если найдено совпадение символов; в противном случае R8B сбрасывается в ноль. Здесь следует отметить один важный момент: команда `sete` не изменяет старшие 56 бит регистра R8. Если операндом-приемником команды является 8-битный или 16-битный регистр, то указанная операция не влияет на верхние 56 или 48 бит соответствующего 64-битного регистра. После команды `sete` следует команда `add rdx,r8`, которая обновляет счетчик вхождений. Этот процесс повторяется до тех пор, пока не будет найден символ конца строки. После завершения сканирования текстовой строки окончательный счетчик вхождений перемещается в регистр RAX и возвращается вызывающей функции. Вывод примера `Ch03_06` выглядит следующим образом:

```
Пример строки: Four score and seven seconds ago, ...
  SearchChar: s Count: 4
  SearchChar: o Count: 4
  SearchChar: z Count: 0
  SearchChar: F Count: 1
  SearchChar: . Count: 3

Пример строки: Red Green Blue Cyan Magenta Yellow
  SearchChar: e Count: 6
  SearchChar: w Count: 1
  SearchChar: l Count: 3
  SearchChar: Q Count: 0
  SearchChar: n Count: 3
```

Версия `CountChars_`, которая обрабатывает строки типа `wchar_t` вместо `char`, может быть легко создана путем изменения команды `lodsb` на команду `lodsw` (загрузить строку слов). Для команд сопоставления символов также необходимо использовать 16-битные регистры вместо 8-битных. Последний символ мнемоники строковой команды x86 указывает размер обрабатываемого операнда.

3.3.2. Конкатенация строк

Конкатенация, или объединение, двух текстовых строк – обычная операция, выполняемая многими программами. Программы на C++ могут использовать для объединения двух строк библиотечные функции `strcat`, `strcat_s`, `wcscat` и `wcscat_s`. Одним из ограничений этих функций является то, что они могут обрабатывать только одну исходную строку. Для объединения нескольких строк необходимо несколько вызовов. В следующем примере под названием `Ch03_07` показано, как использовать команды `scas` (сканировать строку) и `movs` (присвоить строковое значение) для объединения нескольких строк. В листинге 3.7 показан исходный код C++ и ассемблера x86.

Листинг 3.7. Пример Ch03_07

```
//-----------------------------------------------
//               Ch03_07.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <string>

using namespace std;

extern "C" size_t ConcatStrings_(char* des, size_t des_size, const char* const* src, size_t
src_n);

void PrintResult(const char* msg, const char* des, size_t des_len, const char* const* src,
size_t src_n)
{
    string s_test;
    const char nl = '\n';
```

```
    cout << nl << "Пример: " << msg << nl;
    cout << " Исходные строки" << nl;

    for (size_t i = 0; i < src_n; i++)
    {
        const char* s1 = (strlen(src[i]) == 0) ? "<empty string>" : src[i];
        cout << " i:" << i << " " << s1 << nl;

        s_test += src[i];
    }

    const char* s2 = (strlen(des) == 0) ? "<empty string>" : des;

    cout << " Результат слияния" << nl;
    cout << " " << s2 << nl;

    if (s_test != des)
        cout << " Ошибка - проверка на совпадение не пройдена" << nl;
}

int main()
{
    // Размер буфера приемника подходящий
    const char* src1[] = { "One ", "Two ", "Three ", "Four" };
    size_t src1_n = sizeof(src1) / sizeof(char*);
    const size_t des1_size = 64;
    char des1[des1_size];

    size_t des1_len = ConcatStrings_(des1, des1_size, src1, src1_n);
    PrintResult("размер буфера приемника подходящий", des1, des1_len, src1, src1_n);

    // Размер буфера приемника недостаточен
    const char* src2[] = { "Red ", "Green ", "Blue ", "Yellow " };
    size_t src2_n = sizeof(src2) / sizeof(char*);
    const size_t des2_size = 16;
    char des2[des2_size];

    size_t des2_len = ConcatStrings_(des2, des2_size, src2, src2_n);
    PrintResult("размер буфера приемника недостаточен", des2, des2_len, src2, src2_n);

    // Пустая исходная строка
    const char* src3[] = { "Plane ", "Car ", "", "Truck ", "Boat ", "Train ", "Bicycle " };
    size_t src3_n = sizeof(src3) / sizeof(char*);
    const size_t des3_size = 128;
    char des3[des3_size];

    size_t des3_len = ConcatStrings_(des3, des3_size, src3, src3_n);
    PrintResult("пустая исходная строка", des3, des3_len, src3, src3_n);

    // Все строки пустые
    const char* src4[] = { "", "", "", "" };
    size_t src4_n = sizeof(src4) / sizeof(char*);
    const size_t des4_size = 42;
    char des4[des4_size];

    size_t des4_len = ConcatStrings_(des4, des4_size, src4, src4_n);
```

```
    PrintResult("все строки пустые", des4, des4_len, src4, src4_n);

    // Минимальный des_size
    const char* src5[] = { "1", "22", "333", "4444" };
    size_t src5_n = sizeof(src5) / sizeof(char*);
    const size_t des5_size = 11;
    char des5[des5_size];
    size_t des5_len = ConcatStrings_(des5, des5_size, src5, src5_n);
    PrintResult("минимальный des_size", des5, des5_len, src5, src5_n);
    return 0;
}

;-----------------------------------------------
;                Ch03_07.asm
;-----------------------------------------------

; extern "C" size_t ConcatStrings_(char* des, size_t des_size, const char* const* src,
size_t src_n);
;
; Возвращает: -1 - неправильный 'des_size'
; или n >= 0 - длина строки после слияния

        .code
ConcatStrings_ proc frame

; Сохранение энергонезависимых регистров
        push rbx
        .pushreg rbx
        push rsi
        .pushreg rsi
        push rdi
        .pushreg rdi
        .endprolog

; Проверка правильности des_size и src_n
        mov rax,-1                      ;задать код ошибки

        test rdx,rdx                    ;проверка des_size
        jz InvalidArg                   ;переход, если des_size равен 0

        test r9,r9                      ;проверка src_n
        jz InvalidArg                   ;переход, если src_n равен 0

; Регистры, задействованные в цикле ниже
; rbx = des rdx = des_size
; r8 = src r9 = src_n
; r10 = des_index r11 = i
; rcx = длина строки
; rsi, rdi = указатели для команд scasb и movsb

; Необходимая инициализация
        xor r10,r10                     ;des_index = 0
        xor r11,r11                     ;i = 0
        mov rbx,rcx                     ;rbx = des
        mov byte ptr [rbx],0            ;*des = '\0'
```

```
; Повтор цикла, пока инициализация не завершена
        Loop1: mov rax,r8               ;rax = 'src'
        mov rdi,[rax+r11*8]             ;rdi = src[i]
        mov rsi,rdi                     ;rsi = src[i]

; Вычисление длины s[i]
        xor eax,eax
        mov rcx,-1
        repne scasb                     ;поиск '\0'
        not rcx
        dec rcx                         ;rcx = len(src[i])

; Вычисление des_index + src_len
        mov rax,r10                     ;rax = des_index
        add rax,rcx                     ;des_index + len(src[i])
        cmp rax,rdx                     ;проверка des_index + src_len >= des_size?
        jge Done                        ;переход, если des слишком мал

; Обновление des_index
        mov rax,r10                     ;des_index_old = des_index
        add r10,rcx                     ;des_index += len(src[i])

; Копирование src[i] в &des[des_index] (rsi уже хранит src[i])
        inc rcx                         ;rcx = len(src[i]) + 1
        lea rdi,[rbx+rax]               ;rdi = &des[des_index_old]
        rep movsb                       ;присвоение строки

; Обновление i и повтор, если цикл не завершен
        inc r11                         ;i += 1
        cmp r11,r9
        jl Loop1                        ;переход, если i < src_n

; Передача длины получившейся строки

Done: mov rax,r10                       ;rax = des_index (окончательная длина)

; Восстановление энергонезависимых регистров и возврат

InvalidArg:
        pop rdi
        pop rsi
        pop rbx
        ret

ConcatStrings_ endp
        end
```

Начнем с изучения кода C++ в листинге 3.7. Он начинается с оператора объявления функции `ConcatStrings_` на языке ассемблера, который включает четыре параметра: `des` – буфер приемника для последней строки; `des_size` – размер `des` в символах; а параметр `src` указывает на массив, содержащий указатели на текстовые строки `src_n`. В 64-битных программах Visual C++ тип `size_t` эквивалентен 64-битному целому числу без знака. Функция `ConcatStrings_` возвращает длину `des` в случае успеха или `-1`, если предоставленное значение `des_size` меньше или равно нулю.

Тестовые примеры, представленные в main, иллюстрируют использование ConcatStrings_. Если, например, src указывает на массив текстовых строк, состоящий из «Red», «Green», «Blue», последней строкой в des будет «RedGreenBlue», при условии что des достаточно велик, чтобы вместить результат. Если значение des_size недостаточно велико, ConcatStrings_ создает частично объединенную строку. Например, значение des_size, равное 10, приведет к формированию «RedGreen» в качестве финальной строки.

Следуя своему прологу, функция ConcatStrings_ проверяет значение аргумента des_size на правильность с помощью команды test rdx,rdx. Эта команда выполняет поразрядное И для двух своих операндов и устанавливает флаги четности (RFLAGS.PF), знака (RFLAGS.SF) и нуля (RFLAGS.ZF) на основе результата (флаги переноса (RFLAGS.CF) и переполнения (RFLAGS.OF) установлены в ноль). Результат побитовой операции И не сохраняется. Команда тестирования часто используется как альтернатива команде cmp, особенно когда функции необходимо выяснить, является ли значение меньшим, равным или большим нуля. Использование тестовой команды также может быть более эффективным с точки зрения экономии объема кода. В этом случае тестовая команда rdx,rdx требует меньше байтов кода операции, чем команда cmp rdx,0. Инициализация регистров выполняется перед началом цикла обработки конкатенации.

Последующий блок команд образует цикл конкатенации, который начинается с загрузки в регистры RSI и RDI указателя на строку src[i]. Длина src[i] определяется с помощью команды repne scasb в сочетании с несколькими вспомогательными командами. repne (повторять строковую операцию, пока не равно) – это префикс команды, который повторяет выполнение строковой команды, пока выполняется условие RCX ! = 0 && RFLAGS.ZF == 0. Детальное описание комбинации repne scasb (сканирование байта строки) выглядит следующим образом: если RCX не равен нулю, команда scasb сравнивает строковый символ, на который указывает RDI, с содержимым регистра AL и устанавливает флаги состояния в соответствии с результатами. Регистр RDI затем автоматически увеличивается на единицу, так что он указывает на следующий символ, и значение счетчика вычитается из RCX. Эта операция обработки строки повторяется, пока остаются верными вышеупомянутые условия тестирования; в противном случае повторение строковой операции прекращается.

Перед использованием команды repne scasb в регистр RCX было загружено значение –1. По завершении repne scasb регистр RCX содержит значение –(L + 2), где L обозначает фактическую длину строки src[i]. Значение L вычисляется с помощью команды not rcx (побитовое логическое НЕ), за которой следует команда dec rcx (уменьшение на 1), которая равна вычитанию 2 из побитового НЕ дополнения до двух –(L + 2). Следует отметить, что последовательность команд, используемая здесь для вычисления длины текстовой строки, является хорошо известной техникой, которая восходит к процессору 8086.

После вычисления len(src[i]) выполняется проверка, чтобы убедиться, что строка src[i] помещается в целевой буфер. Если сумма des_index+len(src[i]) больше или равна des_size, работа функции завершается. В противном случае к des_index прибавляется len(src[i]), а строка src[i] копируется в правильную позицию в des с помощью команды rep movsb (повторить присвоение байта строки).

Команда `rep movsb` копирует строку, на которую указывает RSI, в то место памяти, на которое указывает RDI, используя длину, указанную в RCX. Команда `inc rcx` выполняется перед копированием строки, чтобы гарантировать, что признак конца строки `'\ 0'` также передается в `des`. Регистр RDI инициализируется правильным смещением в `des` с помощью команды `lea rdi,[rbx+rax]` (загрузить эффективный адрес), которая вычисляет адрес указанного исходного операнда (т. е. `lea` вычисляет RDI = RBX + RAX). Цикл конкатенации может использовать команду `lea`, поскольку регистр RBX указывает на начало `des`, а RAX содержит значение `des_index` до его добавления с помощью `len(src[i])`. После операции копирования строки значение `i` обновляется, и если оно меньше `src_n`, цикл конкатенации повторяется. После завершения операции конкатенации в регистр RAX загружается значение `des_index`, которое представляет собой длину последней строки в `des`. Ниже показан результат работы кода из примера `Ch03_07`:

```
Пример: размер буфера приемника подходящий
  Исходные строки
    i:0 One
    i:1 Two
    i:2 Three
    i:3 Four
  Результат слияния
    One Two Three Four

Пример: размер буфера приемника недостаточен
  Исходные строки
    i:0 Red
    i:1 Green
    i:2 Blue
    i:3 Yellow
  Результат слияния
    Red Green Blue
  Ошибка - проверка на совпадение не пройдена

Пример: пустая исходная строка
  Исходные строки
    i:0 Plane
    i:1 Car
    i:2 <empty string>
    i:3 Truck
    i:4 Boat
    i:5 Train
    i:6 Bicycle
  Результат слияния
    Plane Car Truck Boat Train Bicycle

Пример: все строки пустые
  Исходные строки
    i:0 <empty string>
    i:1 <empty string>
    i:2 <empty string>
    i:3 <empty string>
  Результат слияния
    <empty string>
```

```
Пример: минимальный des_size
  Исходные строки
    i:0 1
    i:1 22
    i:2 333
    i:3 4444
  Результат слияния
    1223334444
```

3.3.3. Сравнение массивов

Помимо текстовых строк, строковые команды x86 также могут использоваться для выполнения операций над другими последовательно упорядоченными элементами данных. В следующем примере исходного кода показано, как использовать команду `cmps` (сравнить строковые операнды) для сравнения элементов двух массивов. Листинг 3.8 содержит исходный код примера `Ch03_08` на языке C++ и ассемблера x86-64.

Листинг 3.8. Пример Ch03_08

```
//-----------------------------------------------
//               Ch03_08.cpp
//-----------------------------------------------
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <memory>

using namespace std;

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

void Init(int* x, int* y, long long n, unsigned int seed)
{
    uniform_int_distribution<> d {1, 10000};
    default_random_engine rng {seed};

    for (long long i = 0; i < n; i++)
        x[i] = y[i] = d(rng);
}

void PrintResult(const char* msg, long long result1, long long result2)
{
    cout << msg << '\n';
    cout << " ожидалось = " << result1;
    cout << " фактически = " << result2 << "\n\n";
}

int main()
{
    // Размещение и инициализация тестовых массивов
    const long long n = 10000;
    unique_ptr<int[]> x_array {new int[n]};
    unique_ptr<int[]> y_array {new int[n]};
```

```
    int* x = x_array.get();
    int* y = y_array.get();

    Init(x, y, n, 11);

    cout << "Результаты CompareArrays_ - array_size = " << n << "\n\n";

    long long result;

    // Вариант с некорректным размером массива
    result = CompareArrays_(x, y, -n);
    PrintResult("Вариант с некорректным размером массива ", -1, result);

    // Вариант с несовпадением первого элемента
    x[0] += 1;
    result = CompareArrays_(x, y, n);
    x[0] -= 1;
    PrintResult("Вариант с несовпадением первого элемента ", 0, result);

    // Вариант с несовпадением элемента в середине
    y[n / 2] -= 2;
    result = CompareArrays_(x, y, n);
    y[n / 2] += 2;
    PrintResult("Вариант с несовпадением элемента в середине ", n / 2, result);

    // Вариант с несовпадением последнего элемента
    x[n - 1] *= 3;
    result = CompareArrays_(x, y, n);
    x[n - 1] /= 3;
    PrintResult("Вариант с несовпадением последнего элемента ", n - 1, result);

    // Вариант с полным совпадением массивов
    result = CompareArrays_(x, y, n);
    PrintResult("Вариант с полным совпадением массивов ", n, result);
    return 0;
}

;-------------------------------------------------
;                Ch03_08.asm
;-------------------------------------------------
; extern "C" long long CompareArrays_(const int* x, const int* y, long long n)
;
; Возвращает    -1          если значение 'n' некорректно
;               0 <= i < n  индекс первого несовпадающего элемента
;               n           все элементы совпадают

        .code
CompareArrays_ proc frame

; Сохранение энергонезависимых регистров
        push rsi
        .pushreg rsi
        push rdi
        .pushreg rdi
        .endprolog
```

```
; Загрузка аргументов и проверка 'n'
        mov rax,-1                      ;rax = код некорректного значения n
        test r8,r8
        jle @F                          ;переход, если n <= 0

; Сравнение массивов на совпадение
        mov rsi,rcx                     ;rsi = x
        mov rdi,rdx                     ;rdi = y
        mov rcx,r8                      ;rcx = n
        mov rax,r8                      ;rax = n
        repe cmpsd
        je @F                           ;массивы равны

; Вычисление индекса первого несовпадения
        sub rax,rcx                     ;rax = индекс несовпадения + 1
        dec rax                         ;rax = индекс несовпадения

; Восстановление энергонезависимых регистров и возврат
@@:     pop rdi
        pop rsi
        ret

CompareArrays_ endp
        end
```

Функция на языке ассемблера `CompareArrays_` сравнивает элементы двух целочисленных массивов и возвращает индекс первого несовпадающего элемента. Если массивы идентичны, возвращается количество элементов. В противном случае возвращается –1, чтобы указать на несовпадение. После пролога функции команда `test r8,r8` проверяет значение аргумента `n`, чтобы убедиться, что оно меньше или равно нулю. Как вы узнали из предыдущего раздела, эта команда выполняет побитовое И для двух операндов и устанавливает флаги состояния RFLAGS.PF, RFLAGS.SF и RFLAGS.ZF на основе результата (RFLAGS.CF и RFLAGS.OF очищаются). Результат операции И отбрасывается. Если значение аргумента `n` недопустимо, команда `jle @F` пропускает код сравнения.

Фактический код сравнения начинается с загрузки в регистр RSI указателя на `x` и в регистр RDI указателя на `y`. Затем количество элементов загружается в регистр RCX. Массивы сравниваются с помощью команды `repe cmpsd` (сравнение строки двойных слов). Эта команда сравнивает два двойных слова, на которые указывают RSI и RDI, и устанавливает флаги состояния в соответствии с результатами. Регистры RSI и RDI увеличиваются на четыре после каждой операции сравнения (используется значение 4, поскольку это размер двойного слова в байтах). Префикс `repe` (повторять, пока равно) указывает процессору повторять команду `cmpsd`, пока выполняется условие `RCX! = 0 && RFLAGS.ZF == 1`. По завершении команды `cmpsd` выполняется условный переход, если массивы равны (RAX уже содержит правильное возвращаемое значение) или вычисляется индекс первых несовпадающих элементов. Ниже показан результат выполнения кода примера `Ch03_08`:

```
Результаты для CompareArrays_ - array_size = 10000

Вариант с некорректным размером массива
  ожидалось = -1 фактически = -1

Вариант с несовпадением первого элемента
  ожидалось = 0 фактически = 0

Вариант с несовпадением элемента в середине
  ожидалось = 5000 фактически = 5000

Вариант с несовпадением последнего элемента
  ожидалось = 9999 фактически = 9999

Вариант с полным совпадением массивов
  ожидалось = 10000 фактически = 10000
```

3.3.4. Обращение массива

Последний пример исходного кода в этом разделе демонстрирует использование команды `lods` (загрузка строки) для *обращения* массива. В отличие от примеров исходного кода в предыдущем разделе, цикл обработки примера `Ch03_09` проходит по исходному массиву, начиная с последнего элемента и заканчивая первым элементом. Выполнение обратного обхода массива требует, чтобы флаг направления (RFLAGS.DF) был изменен способом, совместимым со средой выполнения Visual C++, который показан в этом примере. В листинге 3.9 показан исходный код примера `Ch03_09` на C++ и на языке ассемблера.

Листинг 3.9. Пример Ch03_09

```
//-----------------------------------------------
//              Ch03_09.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>

using namespace std;

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

void Init(int* x, int n)
{
    unsigned int seed = 17;
    uniform_int_distribution<> d {1, 1000};
    default_random_engine rng {seed};
    for (int i = 0; i < n; i++)
    x[i] = d(rng);
}

int main()
```

```
{
    const int n = 25;
    int x[n], y[n];

    Init(x, n);
    int rc = ReverseArray_(y, x, n);

    if (rc != 0)
    {
        cout << "\nРезультат ReverseArray\n";

        const int w = 5;
        bool compare_error = false;

        for (int i = 0; i < n && !compare_error; i++)
        {
            cout << " i: " << setw(w) << i;
            cout << " y: " << setw(w) << y[i];
            cout << " x: " << setw(w) << x[i] << '\n';

            if (x[i] != y[n - 1 - i])
            compare_error = true;
        }

        if (compare_error)
            cout << "Ошибка сравнения ReverseArray\n";
        else
            cout << "Сравнение ReverseArray прошло успешно\n";
    }
    else
        cout << "Ошибка выполнения ReverseArray_()\n";

    return 0;
}

;-----------------------------------------------
;                Ch03_09.asm
;-----------------------------------------------
; extern "C" int ReverseArray_(int* y, const int* x, int n);
;
; Возвращает 0 = некорректное значение n, 1 = успех

        .code
ReverseArray_ proc frame

; Сохранение энергонезависимых регистров
        push rsi
        .pushreg rsi
        push rdi
        .pushreg rdi
        .endprolog

; Проверка, корректно ли значение n
        xor eax,eax                     ;возвращаемый код ошибки
        test r8d,r8d                    ;n <= 0?
        jle InvalidArg                  ;переход, если n <= 0
```

```
; Инициализация регистров для операции обращения
        mov rsi,rdx                     ;rsi = x
        mov rdi,rcx                     ;rdi = y
        mov ecx,r8d                     ;rcx = n
        lea rsi,[rsi+rcx*4-4]           ;rsi = &x[n - 1]

; Сохранение прежнего значения RFLAGS.DF, установка RFLAGS.DF в 1
        pushfq                          ;сохранение прежнего флага RFLAGS.DF
        std                             ;RFLAGS.DF = 1

; Продолжение цикла до завершения операции
        @@: lodsd                       ;eax = *x--
        mov [rdi],eax                   ;*y = eax
        add rdi,4                       ;y++
        dec rcx                         ;n--
        jnz @B

; Восстановление флага RFLAGS.DF и передача возвращаемого значения
        popfq                           ;восстановление флага RFLAGS.DF
        mov eax,1                       ;возвращаемый код успешного выполнения

; Восстановление энергонезависимых регистров и возврат
        InvalidArg:
        pop rdi
        pop rsi
        ret

ReverseArray_ endp
        end
```

Функция `ReverseArray_` копирует элементы исходного массива в целевой массив в обратном порядке. Для этой функции требуются три параметра: указатель на массив-приемник с именем `y`, указатель на массив-источник с именем `x` и количество элементов `n`. После проверки `n` в регистры RSI и RDI загружаются указатели на массивы `x` и `y`. Команда `mov ecx,r8d` загружает количество элементов в регистр RCX. Чтобы перевернуть элементы исходного массива, необходимо вычислить адрес последнего элемента массива `x[n-1]`. Это достигается с помощью команды `lea rsi,[rsi+rcx*4-4]`, которая вычисляет эффективный адрес операнда-источника в памяти (т. е. выполняет арифметическую операцию, указанную в скобках, и сохраняет результат в регистре RSI).

Среда выполнения Visual C++ предполагает, что флаг направления (RFLAGS.DF) всегда сброшен. Если функция языка ассемблера с помощью строковой команды устанавливает RFLAGS.DF на выполнение автоматического уменьшения, флаг должен быть сброшен перед возвратом к вызывающей функции или использованием каких-либо библиотечных функций. Функция `ReverseArray_` частично выполняет это требование, сохраняя текущее состояние RFLAGS.DF в стеке с помощью команды `pushfq` (поместить регистр RFLAGS в стек). Затем она использует команду `std` (установить флаг направления), чтобы установить RFLAGS.DF в 1. Дублирование элементов массива от `x` до `y` не составляет труда. Команда `lodsd` (чтение строки двойных слов) загру-

жает элемент из x в EAX и вычитает 4 из регистра RSI. Следующая команда mov[rdi],eax сохраняет это значение в элементе y, на который указывает RDI. Команда add rdi,4 указывает EDI на следующий элемент в y. Затем происходит декремент регистра RCX, и цикл повторяется до тех пор, пока не завершится обращение массива.

После завершения цикла обращения массива команда popfq (выгрузить стек в регистр RFLAGS) восстанавливает исходное состояние RFLAGS.DF. На этом этапе может возникнуть один вопрос: если среда выполнения Visual C++ предполагает, что RFLAGS.DF всегда сброшен, почему функция ReverseArray_ не использует команду cld (сбросить флаг направления) для восстановления RFLAGS.DF вместо операций выгрузки/загрузки pushfq и popfq? Да, среда выполнения Visual C++ предполагает, что RFLAGS.DF всегда сброшен, но не может реализовать эту политику во время выполнения программы. Если функция ReverseArray_ будет включена в DLL, есть вероятность, что она будет вызвана функцией, написанной на языке, который использует другое состояние по умолчанию для флага направления. Использование pushfq и popfq гарантирует, что состояние направления вызывающей функции всегда правильно восстанавливается. Вот пример вывода на печать в результате выполнения кода примера Ch03_09:

```
Результат ReverseArray
  i:  0 y: 583 x: 560
  i:  1 y: 904 x: 586
  i:  2 y: 924 x: 752
  i:  3 y: 635 x: 743
  i:  4 y: 347 x: 511
  i:  5 y: 313 x: 370
  i:  6 y: 738 x: 809
  i:  7 y: 810 x: 214
  i:  8 y: 935 x: 823
  i:  9 y: 354 x: 456
  i: 10 y: 592 x:  13
  i: 11 y: 613 x: 240
  i: 12 y: 413 x: 413
  i: 13 y: 240 x: 613
  i: 14 y:  13 x: 592
  i: 15 y: 456 x: 354
  i: 16 y: 823 x: 935
  i: 17 y: 214 x: 810
  i: 18 y: 809 x: 738
  i: 19 y: 370 x: 313
  i: 20 y: 511 x: 347
  i: 21 y: 743 x: 635
  i: 22 y: 752 x: 924
  i: 23 y: 586 x: 904
  i: 24 y: 560 x: 583
Сравнение ReverseArray прошло успешно
```

3.4. Заключение

В главе 3 рассмотрены следующие ключевые моменты:

- адрес элемента в одномерном массиве может быть вычислен с использованием базового адреса (т. е. адреса первого элемента) массива, индекса элемента и размера каждого элемента в байтах. Адрес элемента в двумерном массиве можно вычислить, используя базовый адрес массива, индексы строки и столбца, количество столбцов и размер каждого элемента в байтах;
- соглашение о вызовах Visual C++ обозначает каждый регистр общего назначения как энергозависимый или энергонезависимый. Ассемблерная функция должна сохранять содержимое любого используемого ею энергонезависимого регистра общего назначения. Функция должна использовать команду `push` в своем прологе для сохранения содержимого энергонезависимого регистра в стеке. Функция должна использовать команду `pop` в своем эпилоге для восстановления содержимого любого ранее сохраненного энергонезависимого регистра;
- код на языке ассемблера X86-64 может определять и использовать структуры аналогично тому, как это делается в C++. Структура языка ассемблера может потребовать наличия дополнительных заполняющих элементов, чтобы гарантировать, что она семантически эквивалентна структуре C++;
- старшие 32 бита 64-битного регистра общего назначения устанавливаются в ноль в командах, которые определяют соответствующий 32-битный регистр как операнд назначения. Старшие 56 или 48 бит 64-битного регистра общего назначения не затрагиваются, если операндом назначения команды является 8-битный или 16-битный регистр;
- строковые команды x86 `cmps`, `lods`, `movs`, `scas` и `stos` могут использоваться для сравнения, загрузки, копирования, сканирования или инициализации текстовых строк. Их также можно использовать для выполнения операций с массивами и другими подобными упорядоченными структурами данных;
- префиксы `rep`, `repeat`, `repz`, `repne` и `repnz` могут использоваться со строковой командой для многократного повторения строковой операции (RCX содержит значение счетчика) или до тех пор, пока не будет выполнено указанное условие нулевого флага (RFLAGS.ZF);
- состояние флага направления (RFLAGS.DF) должно сохраняться за пределами функций;
- команда тестирования часто используется как альтернатива команде `cmp`, особенно при проверке значения, чтобы убедиться, что оно меньше, равно или больше нуля;
- команду `lea` можно использовать для упрощения эффективных вычислений адресов.

Глава 4

Векторное расширение набора команд AVX

В первых трех главах этой книги вы узнали о ядре платформы x86-64, включая ее типы данных, регистры общего назначения и режимы адресации памяти. Вы также изучили множество примеров кода, которые иллюстрируют основы программирования на языке ассемблера x86-64, включая базовые операнды, целочисленную арифметику, операции сравнения, условные переходы и манипулирование общими структурами данных.

В этой главе представлено векторное расширение набора команд AVX. Глава начинается с краткого обзора технологий AVX и концепций обработки SIMD (Single Instruction Multiple Data). Далее представлено описание среды выполнения AVX, которое охватывает наборы регистров, типы данных и синтаксис команд. В главе также обсуждаются скалярные вычисления AVX с плавающей запятой и вычислительные ресурсы SIMD. Материал, представленный в этой главе, относится не только к AVX, но также предоставляет необходимую справочную информацию для понимания AVX2 и AVX-512, которые объясняются в следующих главах.

В этой и последующих главах термин x86-AVX используется для описания общих характеристик и вычислительных ресурсов векторного расширения набора команд. Аббревиатуры AVX, AXV2 и AVX-512 используются как характеристика атрибутов или команд, связанных с конкретным расширением набора функций x86.

4.1. Обзор AVX

AMD и Intel впервые включили AVX в свои процессоры начиная с 2011 года. AVX расширяет возможности операций с упакованными числами с плавающей запятой одинарной и двойной точности x86-SSE со 128 бит до 256 бит. В отличие от команд для работы с регистрами общего назначения, команды AVX используют синтаксис с тремя операндами и *неразрушающие* (non-destructive) исходные операнды, что значительно упрощает программирование на языке ассемблера. Программисты могут использовать этот новый синтаксис команд с упакованными 128-битными целочисленными операндами, упакованными 128-битными операндами с плавающей запятой и упакованными 256-битными операндами с плавающей запятой. Синтаксис команды с тремя операнда-

ми также можно использовать для выполнения скалярных арифметических операций с плавающей запятой одинарной и двойной точности.

В 2013 году Intel выпустила процессоры с поддержкой AVX2. Это усовершенствование архитектуры расширяет возможности работы AVX с упакованными целыми числами со 128 до 256 бит. AVX2 добавляет на платформу x86 новые команды широковещательной передачи, смешивания и перестановки данных. Этот набор команд также представляет новый режим адресации векторных индексов, который облегчает загрузку (или сбор) в память элементов данных из несмежных местоположений. Самое последнее расширение x86-AVX под названием AVX-512 расширяет возможности SIMD для AVX и AVX2 с 256 бит до 512 бит. AVX-512 также вносит на платформу x86 восемь новых регистров маски K0–K7. Эти регистры облегчают выполнение условных команд и операции слияния данных с использованием поэлементной детализации. В табл. 4.1 приведена сводная характеристика текущих технологий x86-AVX. В этой таблице (и последующих таблицах) аббревиатуры SPFP и DPFP используются для обозначения чисел с плавающей запятой одинарной и двойной точности соответственно.

Таблица 4.1. Краткое описание технологий x86-AVX

Функция	AVX	AVX2	AVX-512
Синтаксис с тремя операндами; неразрушающие исходные операнды	Да	Да	Да
Операции SIMD с использованием 128-битных упакованных целых чисел	Да	Да	Да
Операции SIMD с использованием 256-битных упакованных целых чисел	Нет	Да	Да
Операции SIMD с использованием 512-битных упакованных целых чисел	Нет	Нет	Да
Операции SIMD с использованием 128-битного SPFP, DPFP	Да	Да	Да
Операции SIMD с использованием 256-битного SPFP, DPFP	Да	Да	Да
Операции SIMD с использованием 512-битного SPFP, DPFP	Нет	Нет	Да
Скалярная арифметика SPFP, DPFP	Да	Да	Да
Расширенные операции сравнения SPFP и DPFP	Да	Да	Да
Базовая широковещательная передача и перестановка SPFP, DPFP	Да	Да	Да
Расширенная широковещательная передача и перестановка SPFP, DPFP	Нет	Да	Да
Широковещание упакованных целых чисел	Нет	Да	Да
Расширенное широковещание упакованных целых чисел, сравнение, пермутация, преобразование	Нет	Нет	Да
Управление широковещанием и округлением на уровне команд	Нет	Нет	Да
Слияние–умножение–сложение	Нет	Да	Да
Сбор данных	Нет	Да	Да

Окончание табл. 4.1

Функция	AVX	AVX2	AVX-512
Распределение данных	Нет	Нет	Да
Условное выполнение и объединение данных с использованием регистров opmask	Нет	Нет	Да

Следует отметить, что слитное умножение-сложение – это отдельное расширение функций платформы x86, которое было введено в тандеме с AVX2. Программа должна подтвердить наличие этого расширения функций путем тестирования флага CPUID FMA перед использованием любой из соответствующих команд. Вы узнаете, как это сделать, в главе 16. Остальная часть главы посвящена в основном AVX. В главах 8 и 12 более подробно обсуждаются особенности AVX2 и AVX-512.

4.2. Концепции программирования SIMD

Как подразумевают слова, из которых составлена аббревиатура SIMD (single instruction, multiple data – одиночная команда, множественные данные), вычислительный элемент SIMD выполняет одну и ту же операцию над несколькими элементами данных одновременно. Универсальные операции SIMD включают в себя базовую арифметику, такую как сложение, вычитание, умножение и деление. Методы обработки SIMD также могут применяться к множеству других вычислительных задач, включая сравнение данных, преобразования, логические вычисления, перестановки и битовые сдвиги. Процессоры выполняют операции SIMD за счет своеобразной трактовки битов операнда в регистре или ячейке памяти. Например, 128-битный операнд может содержать два независимых 64-битных целых числа. Он также может содержать четыре 32-битных целых числа, восемь 16-битных целых чисел или шестнадцать 8-битных целых чисел, как показано на рис. 4.1.

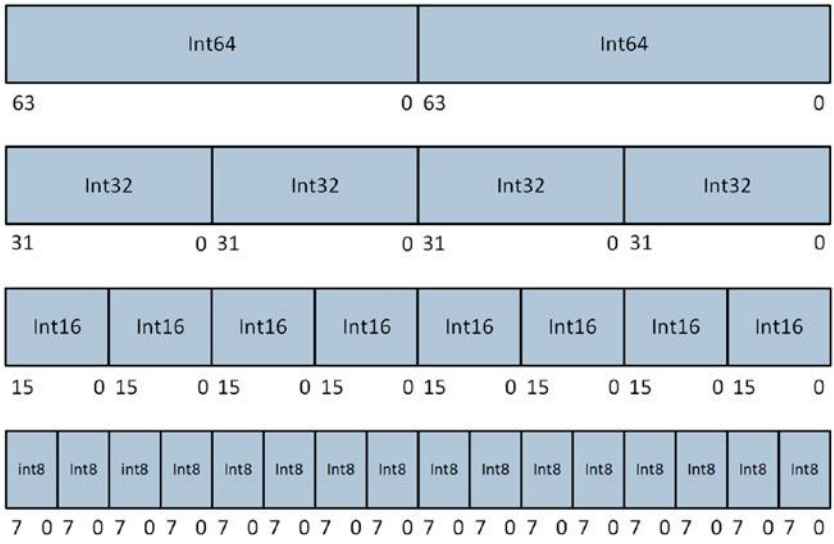


Рис. 4.1. 128-битный операнд с разделением на различные целые числа

Рисунок 4.2 более подробно иллюстрирует несколько арифметических операций SIMD. На этом рисунке сложение целых чисел проиллюстрировано с использованием двух 64-битных целых чисел, четырех 32-битных целых чисел или восьми 16-битных целых чисел. При использовании нескольких элементов данных ускоряется выполнение алгоритма, поскольку ЦП может выполнять необходимые операции параллельно. Например, когда в команде указаны 16-битные целочисленные операнды, ЦП выполняет все восемь сложений 16-битных целых чисел одновременно.

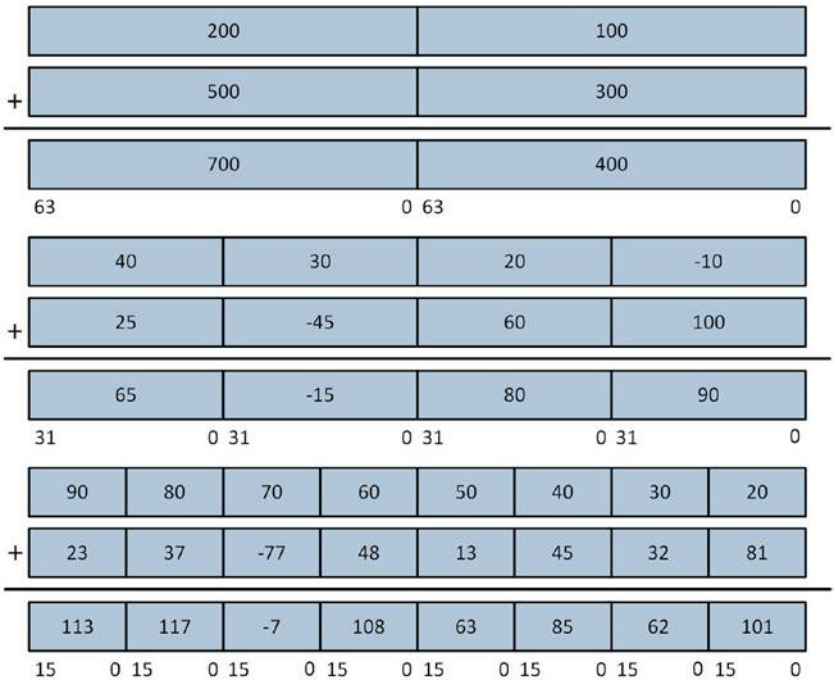


Рис. 4.2. Целочисленное сложение SIMD

4.3. Арифметика с переносом или арифметика с насыщением?

Одна чрезвычайно полезная особенность технологии x86-AVX – это поддержка целочисленной *арифметики с насыщением* (saturated arithmetic). В целочисленной арифметике с насыщением результаты вычислений автоматически обрезаются процессором, чтобы предотвратить переполнение и потерю знака. Это отличается от обычной целочисленной *арифметики с переносом*, когда сохраняется результат переполнения или потери значащих разрядов (как вы скоро увидите). Арифметика с насыщением удобна при работе со значениями пикселей, поскольку она автоматически ограничивает значения и устраняет необходимость явно проверять результат вычисления каждого пикселя на предмет переполнения или потери значащих разрядов. x86-AVX содержит команды арифметики с насыщением с использованием 8-битных и 16-битных целых чисел, как со знаком, так и без знака.

Давайте подробнее рассмотрим несколько примеров арифметики как с переносом, так и с насыщением. На рис. 4.3 показан пример сложения 16-битовых целых чисел со знаком с использованием той и другой арифметики. При сложении двух 16-разрядных чисел со знаком по правилам обычной арифметики с переносом возникает переполнение. Однако в случае арифметики с насыщением результат обрезается до максимально возможного 16-битового целого числа со знаком. На рис. 4.4 показан аналогичный пример с использованием 8-разрядных целых чисел без знака. Помимо сложения, x86-AVX также поддерживает насыщенное целочисленное вычитание, как показано на рис. 4.5. В табл. 4.2 представлены пределы диапазонов для арифметики с насыщением для всех поддерживаемых целочисленных типов.

Рис. 4.3. Сложение 16-битных целых чисел со знаком с использованием двух видов арифметики

Рис. 4-4. Сложение 8-битных беззнаковых целых чисел с использованием двух видов арифметики

Рис. 4.5. Вычитание 16-битных целых чисел со знаком с использованием двух видов арифметики

Таблица 4-2. Пределы диапазонов для арифметики с насыщением

Целое число	Нижний предел	Верхний предел
8-битовое со знаком	–128 (0x80)	+127 (0x7f)
8-битовое без знака	0	+255 (Oxff)
16-битовое со знаком	–32768 (0x8000)	+32767 (0x7fff)
16-битовое без знака	0	+65535 (Oxffff)

4.4. Среда выполнения AVX

В этом разделе вы узнаете о среде выполнения x86-AVX. Здесь приведено описание набора регистров AVX, его типов данных и синтаксиса команд. Как было сказано ранее, x86-AVX – это усовершенствование архитектуры, расширяющее технологию x86-SSE для поддержки операций SIMD с использованием 256-битных или 128-битных операндов. Материал, представленный в этом разделе, не предполагает никаких предварительных знаний или опыта работы с x86-SSE.

4.4.1. Набор регистров

Процессоры x86-64 с поддержкой AVX имеют 16 256-разрядных регистров с именами YMM0 – YMM15. Младшие 128 бит каждого регистра YMM привязаны к соответствующему регистру XMM, как показано на рис. 4.6. Большинство команд AVX могут использовать любой из регистров XMM или YMM в качестве операндов SIMD. Регистры XMM также можно задействовать для выполнения скалярных вычислений с плавающей запятой с использованием значений одинарной или двойной точности, как в x86-SSE. Программистам, имеющим опыт использования x86-SSE на языке ассемблера, необходимо знать о некоторых незначительных различиях в исполнении между этим более ранним расширением набора команд и x86-AVX. Эти различия объясняются далее в данной главе.

Среда выполнения x86-AVX также содержит *регистр управления и состояний* MXCSR. Этот регистр содержит флаги состояния, облегчающие обнаружение ошибок, вызванных арифметическими операциями с плавающей запятой. Он также включает в себя управляющие биты, которые программы могут использовать для включения или отключения исключений с плавающей запятой и определения параметров округления. Вы узнаете больше о регистре MXCSR позже в этой главе.

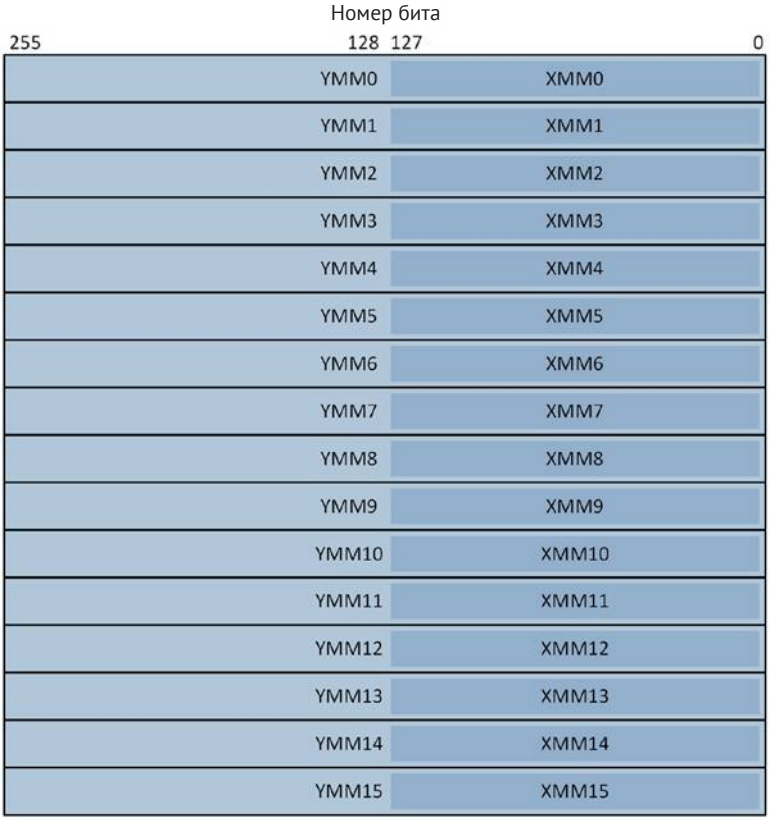


Рис. 4.6. Набор регистров AVX

4.4.2. Типы данных

Как упоминалось ранее, AVX поддерживает операции SIMD с использованием 256-битных и 128-битных упакованных операндов с плавающей запятой одинарной точности или упакованных операндов двойной точности. Регистр YMM разрядностью 256 бит или ячейка памяти может содержать восемь значений с одинарной точностью или четыре значения с двойной точностью, как показано на рис. 4.7. При использовании 128-битного регистра XMM или ячейки памяти команда AVX может обрабатывать четыре значения с одинарной точностью или два значения с двойной точностью. Подобно SSE и SSE2, команды AVX используют двойное или четверное слово младшего разряда регистра XMM для выполнения скалярных арифметических операций с плавающей запятой одинарной или двойной точности.

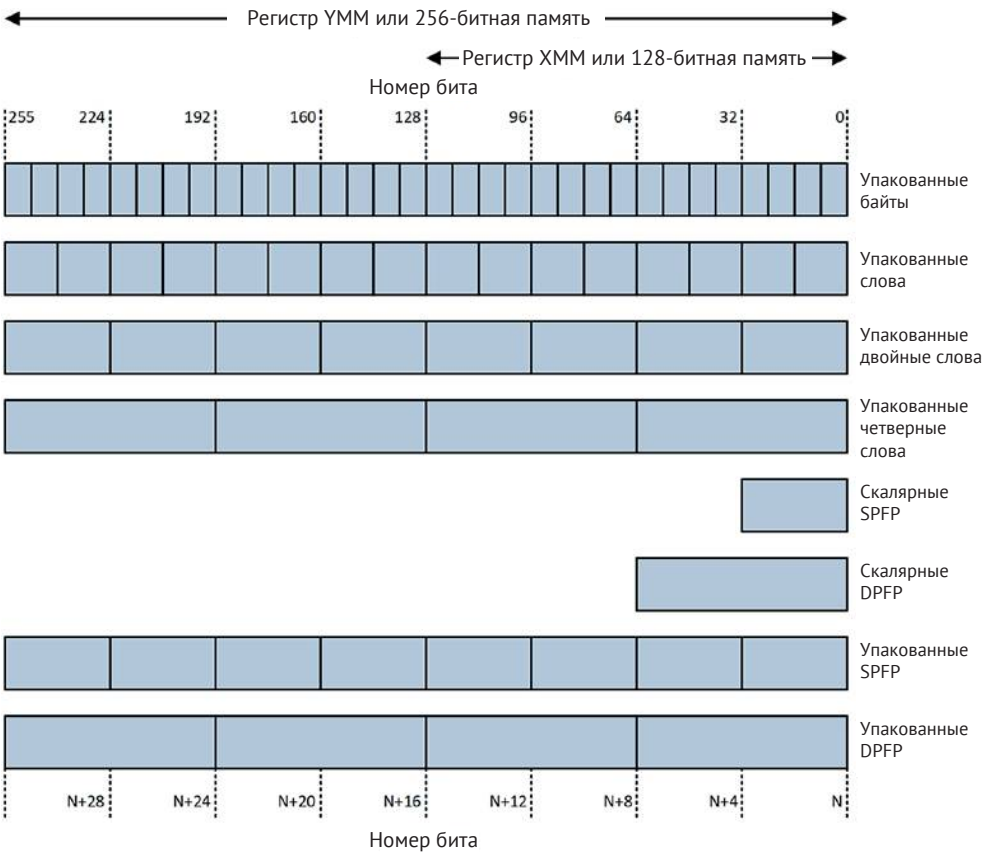


Рис. 4.7. Типы данных AVX и AVX2

AVX также включает в себя команды, которые используют регистры XMM для выполнения операций SIMD с использованием различных упакованных целочисленных операндов, включая байты, слова, двойные слова и четверные слова. AVX2 расширяет возможности обработки упакованных целых чисел AVX до регистров YMM и 256-битных операндов в памяти. На рис. 4.7 также показаны эти типы данных.

4.4.3. Синтаксис команд

Возможно, наиболее примечательным аспектом программирования x86-AVX является использование синтаксиса команд на современном языке ассемблера. Большинство команд x86-AVX используют формат с тремя операндами, который состоит из двух операндов-источников и одного операнда-приемника. Общий синтаксис, используемый для команд x86-AVX, имеет следующий вид: `InstrMnemonic DesOp, SrcOp1, SrcOp2`. Здесь `InstrMnemonic` обозначает мнемонику команды, `DesOp` представляет операнд-приемник, а `SrcOp1` и `SrcOp2` обозначают операнды-источники. Небольшое подмножество команд x86-AVX использует один или три исходных операнда вместе с операндом-приемником. Почти

все операнды источника команд x86-AVX неразрушающие. Это означает, что исходные операнды не изменяются во время выполнения команды, за исключением случаев, когда регистр операнда-приемника совпадает с одним из регистров исходного операнда. Использование неразрушающих исходных операндов часто позволяет написать более простой и быстрый код, поскольку количество передач данных из регистра в регистр, которые должна выполнять функция, уменьшается.

Способность x86-AVX поддерживать синтаксис команд с тремя операндами обусловлена новым префиксом кодирования команд. Префикс векторного расширения (VEX) позволяет кодировать команды x86-AVX с использованием более эффективного формата, чем префиксы, используемые для команд x86-SSE. Префикс VEX также использовался для добавления новых команд регистров общего назначения на платформу x86. Вы узнаете об этих командах в главе 8.

4.5. Скалярные вычисления AVX с плавающей запятой

В этом разделе рассмотрены скалярные возможности AVX с плавающей запятой. Мы начнем с краткого объяснения некоторых важных понятий с плавающей запятой, включая типы данных, битовые кодировки и специальные значения. Разработчики программного обеспечения, хорошо владеющие этими концепциями, часто могут улучшить производительность алгоритмов, которые интенсивно используют арифметику с плавающей запятой, и минимизировать возможные ошибки вычислений. В этом разделе также рассмотрены скалярные регистры с плавающей запятой AVX, включая описания регистров XMM и регистра управления и состояний MXCSR. Раздел завершается обзором набора скалярных команд AVX с плавающей запятой.

4.5.1. Концепция программирования с плавающей запятой

В математике система вещественных чисел изображает бесконечный континуум всех возможных положительных и отрицательных чисел, включая целые, рациональные и иррациональные числа. Учитывая ограниченные ресурсы, для аппроксимации системы с вещественными числами современные вычислительные архитектуры обычно используют систему с плавающей запятой. Как и многие другие вычислительные платформы, система с плавающей запятой x86 основана на стандарте IEEE 754 для двоичной арифметики с плавающей запятой. Этот стандарт включает спецификации, которые определяют битовые кодировки, пределы диапазона и точность для скалярных значений с плавающей запятой. Стандарт IEEE 754 также определяет важные детали, относящиеся к арифметическим операциям с плавающей запятой, правилам округления и числовым исключениям.

Набор команд AVX поддерживает обычные операции с плавающей запятой, используя значения одинарной точности (32 бита) и двойной точности (64 бита). Многие компиляторы C++, включая Visual C++, используют присущие x86 типы с одинарной и двойной точностью для реализации типов C++ `float` и `double`. Рисунок 4.8 иллюстрирует организацию памяти для значений с плавающей запятой как одинарной, так и двойной точности. На рисунке также представлены для сравнения целочисленные типы.

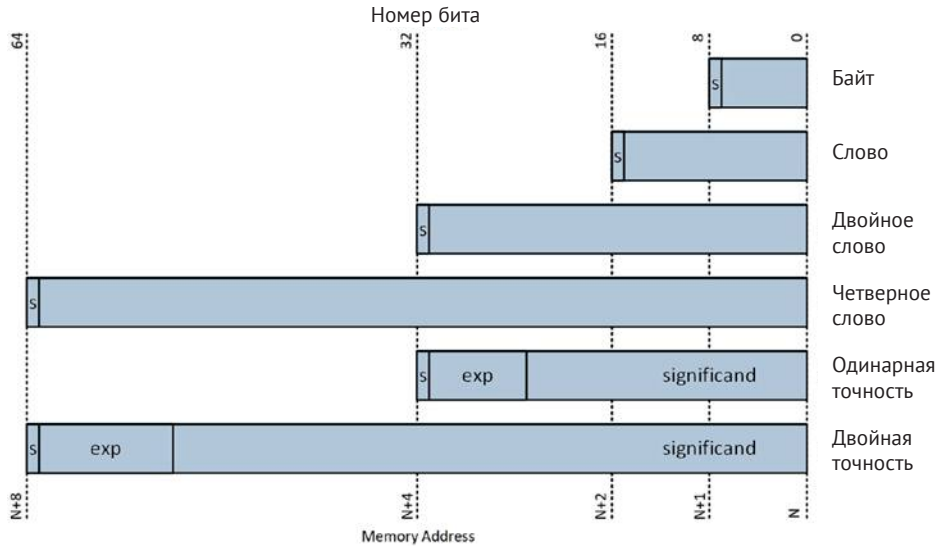


Рис. 4.8. Организация памяти для значений с плавающей запятой

Для двоичного кодирования значения с плавающей запятой требуется три различных поля: значащая часть (мантисса), показатель степени (экспонента) и бит знака. Поле мантиссы представляет собой значащие цифры числа (или дробную часть). Показатель степени определяет положение двоичной «десятичной» запятой в мантиссе, которая определяет величину. Бит знака указывает, является число положительным (s = 0) или отрицательным (s = 1). В табл. 4.3 перечислены значения разрядности, которые используются для кодирования значений с плавающей запятой одинарной и двойной точности.

Таблица 4.3. Параметры разрядности чисел с плавающей запятой

Параметр	Одинарная точность	Двойная точность
Всего разрядов	32	64
Разрядность мантиссы	23	52
Разрядность экспоненты	8	11
Разрядность знака	1	1
Смещение экспоненты	+127	+1023

На рис. 4.9 показано, как преобразовать десятичное число в кодированное значение с плавающей запятой, совместимое с x86. В этом примере число 237,8125 преобразуется из десятичного числа в его представление с плавающей запятой одинарной точности. Процесс начинается с перевода числа из основания 10 в основание 2. Затем значение по основанию 2 преобразуется в *двоичную экспоненциальную запись* (scientific notation, так называемая научная нотация). Значение справа от символа E_2 – это двоичная экспонента. Правильно закодированное значение с плавающей запятой использует *сме-*

щенную экспоненту вместо истинной экспоненты, поскольку это ускоряет операции сравнения с плавающей запятой. Для числа с плавающей запятой одинарной точности значение смещения составляет +127. Добавление значения смещения экспоненты к истинной экспоненте дает нам двоичную экспоненциальную запись со смещенным значением экспоненты. В примере, показанном на рис. 4.9, добавление 111b (+7) к 1111111b (+127) дает двоичную экспоненциальную запись со смещенным значением показателя степени 10000110b (+134).

Рис. 4.9. Процесс кодирования числа с плавающей запятой одинарной точности

При кодировании значения с плавающей запятой одинарной или двойной точности первая значащая цифра 1 мантиссы подразумевается и не включается в окончательное двоичное представление. Отбрасывание первой цифры 1 образует нормализованное значение. Теперь доступны три поля, необходимых для кодирования в соответствии с IEEE 754, как показано в табл. 4.4. Чтение битовых полей в этой таблице слева направо дает 32-битное значение 0x436DD000, которое является окончательным результатом кодирования с плавающей запятой одинарной точности числа 237,8125.

Таблица 4.4. Битовые поля IEEE 754-совместимого кодирования числа 237,8125

Знак	Смещенная экспонента	Мантисса
1	10000110	11011011101000000000000

Схема кодирования с плавающей запятой IEEE 754 резервирует небольшой набор битовых шаблонов для специальных значений, которые используются в некоторых ситуациях. Первая группа специальных значений включает *денормализованные числа*. Как показано в предыдущем примере кодирования, стандартное кодирование числа с плавающей запятой предполагает, что ведущая цифра мантиссы всегда равна 1. Одним из ограничений схемы кодирования с плавающей запятой IEEE 754 является ее неспособность точно представлять числа, очень близкие к нулю. В этих случаях значения коди-

руются с использованием денормализованного формата, который позволяет кодировать очень маленькие числа, близкие к нулю (как положительные, так и отрицательные), но с меньшей точностью. Денормализованные значения встречаются редко, но даже если это случается, ЦП может их обрабатывать. В алгоритмах, где использование денормализованного значения проблематично, функция может проверять значение с плавающей запятой, чтобы обнаружить денормализованное число, или процессор может быть сконфигурирован для генерации исключения по *ошибке потери значимости* (underflow) или денормализованному числу.

Другое применение специальных значений связано с кодировками, которые используются для представления нуля с плавающей запятой. Стандарт IEEE 754 поддерживает два разных представления нуля с плавающей запятой: *положительный ноль* (+0,0) и *отрицательный ноль* (–0,0). Отрицательный ноль может генерироваться либо алгоритмически, либо как побочный эффект режима округления с плавающей запятой. В вычислительном отношении процессор одинаково обрабатывает положительный и отрицательный нули, и программисту обычно не о чем беспокоиться.

Схема кодирования IEEE 754 также поддерживает положительное и отрицательное представления *бесконечности*. Бесконечности создаются с помощью определенных числовых алгоритмов, ситуаций переполнения или деления на ноль. Как обсуждается далее в этой главе, процессор можно настроить так, чтобы он генерировал исключение всякий раз, когда происходит переполнение с плавающей запятой или если программа пытается разделить число на ноль.

Последний специальный тип значений называется *Not a Number* (NaN). NaN – это кодировки с плавающей запятой, которые представляют недопустимые числа. Стандарт IEEE 754 определяет два типа NaN: *сигнальный* NaN (signaling NaN, SNaN) и *тихий* NaN (quiet NaN, QNaN). SNaN создаются только с помощью программного обеспечения; процессор x86-64 не создает SNaN во время каких-либо арифметических операций. Любая попытка команды использовать SNaN вызовет исключение недопустимой операции, если исключение не замаскировано. SNaN полезны для тестирования обработчиков исключений. Их также можно использовать в прикладной программе для частных целей числовой обработки. ЦП x86 использует QNaN в качестве ответа по умолчанию на определенные недопустимые арифметические операции, исключения которых замаскированы. Например, один из уникальных кодов QNaN, называемый неопределенным, заменяет результатом всякий раз, когда функция использует одну из команд скалярного вычисления квадратного корня из отрицательного числа. Программы также могут использовать QNaN для обозначения ошибок конкретного алгоритма или других необычных числовых состояний. Когда QNaN используются в качестве операндов, они позволяют продолжить обработку без генерации исключения.

При разработке программного обеспечения, которое выполняет вычисления с плавающей запятой, важно помнить, что используемая схема кодирования представляет собой просто *приближение* системы действительных чисел. Никакая система кодирования с плавающей запятой не может представить бесконечное количество значений с использованием конечного числа битов.

Это приводит к ошибкам округления с плавающей запятой, которые могут повлиять на точность расчета. Кроме того, некоторые математические свойства, которые верны для целых и действительных чисел, не обязательно верны для чисел с плавающей запятой. Например, умножение с плавающей запятой не обязательно ассоциативно; `(a*b)*c` может не равняться `a*(b*c)` для некоторых значений `a`, `b` и `c`. Разработчики алгоритмов, требующих высоких уровней точности вычислений с плавающей запятой, должны знать об этих проблемах. Приложение содержит список ссылок, которые более подробно объясняют эту и другие потенциальные ловушки арифметики с плавающей запятой. Глава 9 также содержит пример исходного кода, который иллюстрирует неассоциативность операций с плавающей запятой.

4.5.2. Набор скалярных регистров с плавающей запятой

Как ранее было показано на рис. 4.6, все процессоры, совместимые с x86-64, имеют 16 128-битных регистров с именами XMM0–XMM15. Программа может использовать любой из регистров XMM для выполнения скалярных операций с плавающей запятой, включая общие арифметические вычисления, передачу данных, сравнения и преобразования типов. ЦП использует младшие 32 бита регистра XMM для выполнения вычислений с плавающей запятой одинарной точности. В операциях с плавающей запятой двойной точности используются младшие 64 бита. На рис. 4.10 более подробно показано расположение этих регистров. Программы не могут использовать старшие биты регистра XMM для выполнения скалярных вычислений с плавающей запятой. Однако при использовании в качестве операнда назначения значения этих битов могут быть изменены во время выполнения команды скалярного вычисления с плавающей запятой AVX, как объяснено позже в этом разделе.

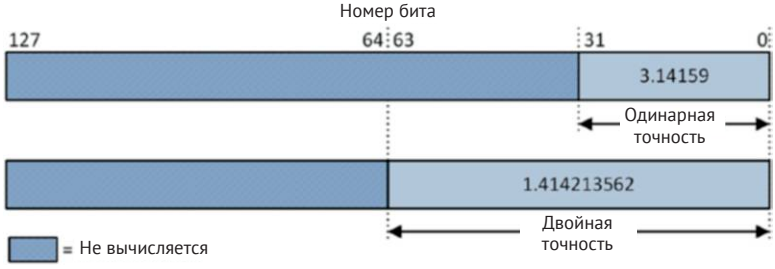


Рис. 4.10. Скалярные значения с плавающей запятой при загрузке в регистр XMM

4.5.3. Регистр управления и состояния

Помимо регистров XMM, процессоры x86-64 имеют 32-разрядный регистр управления и состояния MXCSR. Этот регистр содержит серию управляющих флагов, которые позволяют программе устанавливать опции для вычислений с плавающей запятой и исключений. Он также хранит набор флагов состояния, которые можно протестировать для обнаружения ошибок вычислений с плавающей запятой. На рис. 4.11 показана организация битов в MXCSR; табл. 4.5 описывает назначение каждого битового поля.

Рис. 4.11. Регистр управления и состояния MXCSR

Таблица 4.5. Описание битовых полей регистра MXCSR

Бит	Имя поля	Описание
IE	Флаг недопустимой операции	Флаг ошибки недопустимой операции с плавающей запятой
DE	Флаг денормализации	Флаг ошибки денормализации с плавающей запятой
ZE	Флаг деления на ноль	Флаг ошибки деления на ноль с плавающей запятой
OE	Флаг переполнения	Флаг ошибки переполнения с плавающей запятой
UE	Флаг потери значимости	Флаг потери значимости с плавающей запятой
PE	Флаг точности	Флаг ошибки точности с плавающей запятой
DAZ	Обнуление денормализованного операнда	Когда установлено значение 1, принудительно преобразует денормализованный исходный операнд в ноль перед его использованием в вычислениях
IM	Маска недопустимой операции	Маска исключения ошибки операции с плавающей запятой
DM	Маска денормализации	Маска исключения ошибки денормализации с плавающей точкой
ZM	Маска деления на ноль	Маска исключения ошибки деления на ноль
OM	Маска переполнения	Маска исключения ошибки переполнения
UM	Маска потери значимости	Маска исключения ошибки потери значимости
PM	Маска точности	Маска исключения ошибки точности
RC	Управление округлением	Задает метод округления результатов с плавающей запятой. Допустимые варианты включают округление до ближайшего (00b), округление с понижением до +∞ (01b), округление с повышением до +∞ (10b), округление до нуля или усечение (11b)
FTZ	Обнуление	Если установлено значение 1, принудительно обнуляется результат, если исключение потери значимости замаскировано и возникает ошибка потери значимости с плавающей запятой

Прикладная программа может модифицировать любой из флагов управления или битов состояния MXCSR, чтобы приспособиться к своим специфическим требованиям обработки операций с плавающей запятой SIMD. Любая попытка записать ненулевое значение в зарезервированную битовую позицию заставит процессор сгенерировать исключение. Процессор устанавливает

флаг ошибки MXCSR в 1 после возникновения состояния ошибки. Флаги ошибок MXCSR не сбрасываются процессором автоматически после обнаружения ошибки; они должны быть сброшены вручную. Флаги управления и биты состояния регистра MXCSR могут быть изменены с помощью команды `vldmxcsr` (загрузить регистр MXCSR). Установка бита маски в 1 отключает соответствующее исключение. Команда `vstmxcsr` (сохранить регистр MXCSR) может использоваться для сохранения текущего состояния MXCSR. Прикладная программа не может напрямую обращаться к внутренним таблицам процессора, в которых указаны обработчики исключений с плавающей запятой. Однако большинство компиляторов C++ предоставляют библиотечную функцию, которая позволяет прикладной программе назначать функцию обратного вызова, вызывающуюся всякий раз, когда возникает исключение с плавающей запятой.

MXCSR содержит два управляющих флага, которые можно использовать для ускорения некоторых вычислений с плавающей запятой. Установка флага управления MXCSR.DAZ в 1 может улучшить быстродействие алгоритмов, в которых допустимо округление денормализованного значения до нуля. Точно так же флаг управления MXCSR.FTZ можно использовать для ускорения вычислений, где часто встречается потеря значимости с плавающей запятой. Обратной стороной включения любой из этих опций является несоответствие стандарту IEEE 754 для операций с плавающей запятой.

4.5.4. Обзор набора команд

В табл. 4.6 перечислены в алфавитном порядке часто используемые скалярные команды AVX с плавающей запятой. В этой таблице мнемоническая нотация `[d|s]` означает, что команда может использоваться как с операндами с плавающей запятой двойной точности, так и с операндами с плавающей запятой одинарной точности. Вы узнаете, как использовать большинство из этих команд, в главе 5.

Таблица 4.6. Обзор часто используемых скалярных команд AVX с плавающей запятой

Мнемоника	Описание
`vadds[d|s]`	Скалярное сложение с плавающей запятой
`vbroadcasts[d|s]`	Рассылка скалярного значения с плавающей запятой
`vcmps[d|s]`	Скалярное сравнение с плавающей запятой
`vcomis[d|s]`	Упорядоченное скалярное сравнение с плавающей запятой и установка RFLAGS
`vcvts[d|s]2si`	Преобразование скалярного числа с плавающей запятой в двусловное целое число со знаком
`vcvtsd2ss`	Преобразовать скалярный DPFP в скалярный SPFP
`vcvtsi2s[d|s]`	Преобразование двусловного целого числа со знаком в скалярное с плавающей запятой
`vcvtss2sd`	Преобразование скалярного SPFP в DPFP
`vcvtts[d|s]2si`	Преобразование с усечением скалярных чисел с плавающей запятой в целое число со знаком

Окончание табл. 4.6

Мнемоника	Описание
vdivs[d|s]	Скалярное деление с плавающей запятой
vmaxs[d|s]	Скалярный максимум с плавающей запятой
vmins[d|s]	Скалярный минимум с плавающей запятой
vmovs[d|s]	Присвоение скалярного значения с плавающей запятой
vmuls[d|s]	Скалярное умножение с плавающей запятой
vrounds[d|s]	Округление скалярного значения с плавающей запятой
vsqrts[d|s]	Скалярный квадратный корень с плавающей запятой
vsubsfd|s]	Скалярное вычитание с плавающей запятой
vucomis[d|s]	Неупорядоченное скалярное сравнение с плавающей запятой и установка RFLAGS

Таблица 4.7 иллюстрирует работу скалярных команд AVX с плавающей запятой vadds[d|s] и vsqrts[d|s]. В этих примерах двоеточие обозначает диапазоны позиций битов в регистре (например, 31:0 обозначает позиции битов с 31 по 0 включительно). Обратите внимание, что выполнение скалярной команды с плавающей запятой AVX также копирует неиспользуемые биты первого исходного операнда в операнд-адресат. Также обратите внимание, что старшие 128 бит соответствующего регистра YMM установлены в ноль.

Таблица 4.7. Примеры скалярных команд AVX с плавающей запятой

Команда	Операция
vaddss xmm0,xmm1,xmm2	xmm0[31:0] = xmm1[31:0] + xmm2[31:0] xmm0[127:32] = xmm1[127:32] ymm0[255:128] = 0
vaddsd xmm0,xmm1,xmm2	xmm0[63:0] = xmm1[63:0] + xmm2[63:0] xmm0[127:64] = xmm1[127:64] ymm0[255:128] = 0
vsqrtss xmm0,xmm1,xmm2	xmm0[31:0] = sqrt(xmm2[31:0]) xmm0[127:32] = xmm1[127:32] ymm0[255:128] = 0
vsqrtsd xmm0,xmm1,xmm2	xmm0[63:0] = sqrt(xmm2[63:0]) xmm0[127:64] = xmm1[127:64] ymm0[255:128] = 0

4.6. Операции с упакованными числами с плавающей запятой в AVX

AVX поддерживает операции с упакованными числами с плавающей запятой с использованием операндов шириной 128 или 256 бит. На рис. 4.12 и 4.13 показаны стандартные арифметические операции с плавающей запятой, использующие 256-битные операнды с элементами одинарной и двойной точности. Подобно скалярным числам с плавающей запятой, округление для арифмети-

ческих операций AVX с упакованными числами с плавающей запятой определяется флагом управления округлением MXCSR, как показано в табл. 4.5. Процессор также использует флаги состояния MXCSR, чтобы сигнализировать о возникновении состояния ошибки обработки упакованного числа с плавающей запятой.

Рис. 4.12. Сложение средствами AVX упакованных чисел с плавающей запятой одинарной точности

Рис. 4.13. Умножение средствами AVX чисел с плавающей запятой двойной точности

Большинство арифметических команд AVX выполняют свои операции, используя соответствующие позиции элементов двух исходных операндов (*вертикальные* операции). AVX также поддерживает *горизонтальные* арифметические операции с использованием упакованных операндов с плавающей запятой или упакованных целых чисел. *Горизонтальная* арифмети-

ческая операция выполняет свои вычисления с использованием смежных элементов данных упакованного типа. На рис. 4.14 показано горизонтальное сложение с использованием операндов одинарной точности с плавающей запятой и горизонтальное вычитание с использованием операндов с плавающей запятой двойной точности. Набор команд AVX также поддерживает целочисленное горизонтальное сложение и вычитание с использованием упакованных слов и двойных слов. Горизонтальные операции обычно используются для уменьшения операнда упакованных данных, который содержит несколько промежуточных значений, до единственного конечного результата.

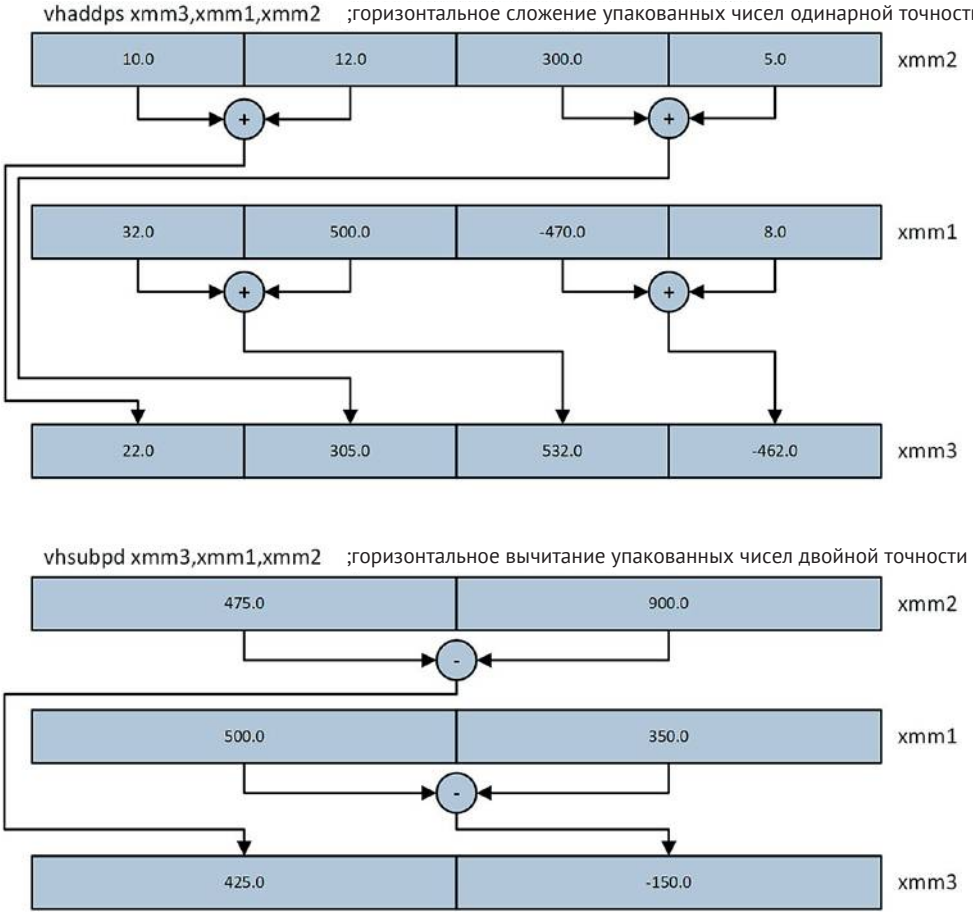


Рис. 4.14. Горизонтальное сложение и вычитание средствами AVX с использованием элементов одинарной и двойной точности

4.6.1. Обзор набора команд

В табл. 4.8 перечислены в алфавитном порядке часто используемые команды AVX с плавающей запятой. Подобно таблице скалярных операций с плавающей запятой, которую вы видели в предыдущем разделе, мнемоника `[d|s]` означает, что команда может использоваться либо с упакованными операндами с плавающей запятой двойной точности, либо с упакованными операндами с плавающей запятой одинарной точности. Вы узнаете, как использовать многие из этих команд, в главе 6.

Таблица 4.8. Обзор часто используемых команд AVX для работы с упакованными операндами с плавающей запятой

Команда	Описание
`vaddp[d|s]`	Сложение упакованных операндов с плавающей запятой
`vaddsubp[d|s]`	Сложение-вычитание упакованных операндов с плавающей запятой
`vandp[d|s]`	Побитовое И упакованных операндов с плавающей запятой
`vandnp[d|s]`	Побитовое И-НЕ упакованных операндов с плавающей запятой
`vblendp[d|s]`	Смешивание упакованных операндов с плавающей запятой
`vblendvp[d|s]`	Переменное смешивание упакованных операндов с плавающей запятой
`vcmpp[d|s]`	Сравнение упакованных операндов с плавающей запятой
`vcvtdq2p[d|s]`	Преобразование упакованных целых двухсловных чисел со знаком в числа с плавающей запятой
`vcvtp[d|s]2dq`	Преобразование упакованных чисел с плавающей запятой в двойные слова со знаком
`vcvtpd2ps`	Преобразование упакованных DPFP в упакованные SPFP
`vcvtps2pd`	Преобразование упакованных SPFP в упакованные DPFP
`vdivp[d|s]`	Деление упакованных операндов с плавающей запятой
`vdpp[d|s]`	Скалярное произведение упакованных операндов
`vhaddp[d|s]`	Горизонтальное сложение упакованных операндов с плавающей запятой
`vhsubp[d|s]`	Горизонтальное вычитание упакованных операндов с плавающей запятой
`vmaskmovp[d|s]`	Условное присвоение и сохранение упакованных операндов с плавающей запятой
`vmaxp[d|s]`	Максимум из упакованных операндов с плавающей запятой
`vminp[d|s]`	Минимум из упакованных операндов с плавающей запятой
`vmovap[d|s]`	Присвоение выровненных упакованных значений с плавающей запятой
`vmovmskp[d|s]`	Извлечение битовой маски знака упакованных операндов с плавающей запятой
`vmovup[d|s]`	Присвоение невыровненных упакованных значений с плавающей запятой
`vmulp[d|s]`	Перемножение упакованных операндов с плавающей запятой

Окончание табл. 4.8

Команда	Описание
`vorp[d|s]`	Побитовое ИЛИ упакованных операндов с плавающей запятой
`vpermilp[d|s]`	Перестановка элементов в пределах упакованных операндов с плавающей запятой
`vroundp[d|s]`	Округление упакованных значений с плавающей запятой
`vshufpfd|s]`	Перемешивание упакованных значений с плавающей запятой
`vsqrtp[d|s]`	Квадратный корень из упакованных значений с плавающей запятой
`vsubp[d|s]`	Вычитание упакованных операндов с плавающей запятой
`vunpckhp[d|s]`	Распаковка и чередование верхних упакованных значений с плавающей запятой
`vunpcklp[d|s]`	Распаковка и чередование нижних упакованных значений с плавающей запятой
`vxorp[d|s]`	Побитовое исключающее ИЛИ упакованных операндов с плавающей запятой

4.7. Операции с упакованными целыми числами в AVX

AVX поддерживает целочисленные операции с использованием упакованных операндов шириной 128 бит. Операнд шириной 128 бит упрощает операции с упакованными целыми числами с использованием двух четверных слов, четырех двойных слов, восьми слов или шестнадцати байтов, как показано на рис. 4.15. На этом рисунке команда `vpaddb` (сложение упакованных целых чисел) иллюстрирует сложение упакованных 8-битных целых чисел. Команда `vpmaxsw` сохраняет максимальное значение слова со знаком каждой пары элементов в указанном операнде назначения. Команда `vpmulld` выполняет умножение упакованных двойных слов со знаком и сохраняет младшие 32 бита каждого результата. Наконец, команда `vpsllq` выполняет логический сдвиг влево каждого элемента четверного слова, используя количество битов, указанное непосредственным операндом. Обратите внимание, что эта команда поддерживает использование непосредственного операнда для указания количества битов.

Большинство упакованных целочисленных команд AVX не обновляют флаги состояния в регистре RFLAGS. Это означает, что отсутствует оповещение о таких ошибках, как арифметическое переполнение и потеря значимости. Это также означает, что результаты операции с упакованным целым числом не влияют напрямую на выполнение условных команд `cmovcc`, `jcc` и `setb`. Однако программы могут использовать специфические для SIMD методы для принятия логических решений на основе результата операции с упакованным целым числом. Вы увидите примеры этих методов в главе 7.

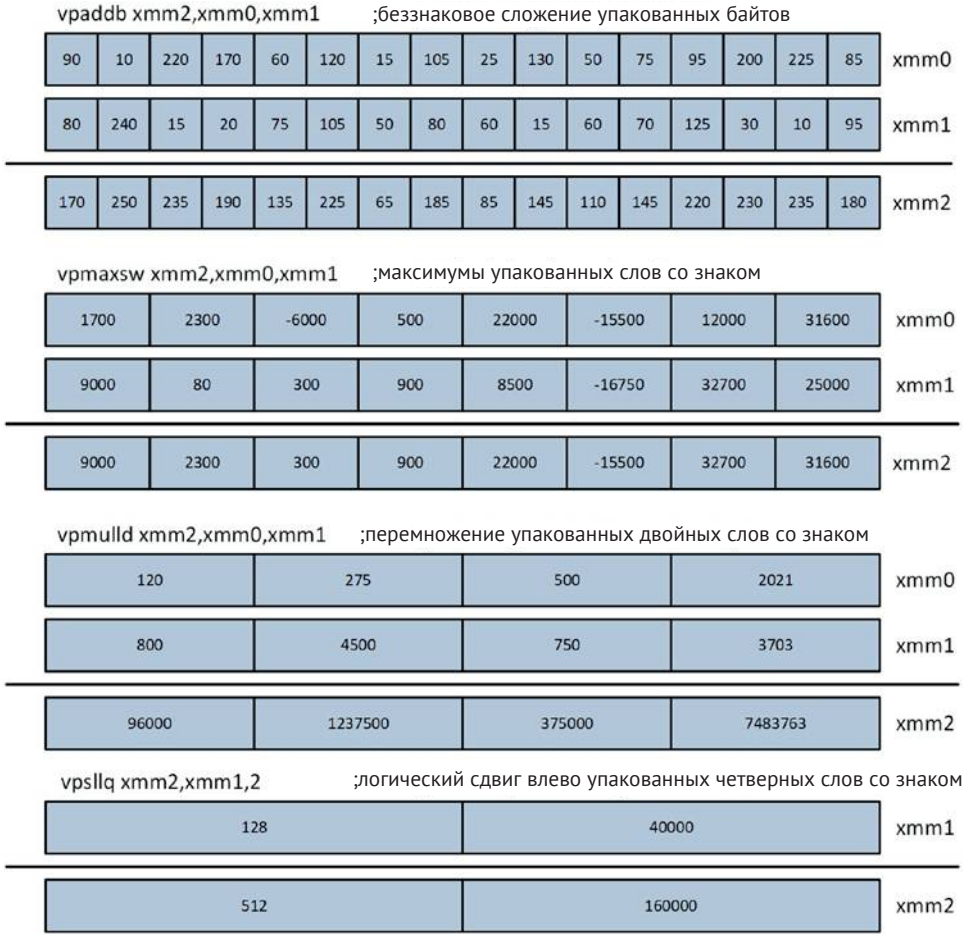


Рис. 4.15. Пример упакованных целочисленных операций AVX

4.7.1. Обзор набора команд

В табл. 4.9 перечислены в алфавитном порядке наиболее часто используемые команды AVX для работы с упакованными целыми числами. В этой таблице мнемонический текст `[b|w|d|q]` обозначает размер (байт, слово, двойное слово или четверное слово) обрабатываемых элементов. Вы узнаете, как использовать многие из этих команд, в главе 7.

Таблица 4.9. Обзор часто используемых упакованных целочисленных команд AVX

Команда	Описание
vmov[d|q]	Запись/чтение регистра XMM
vmovdqa	Присвоение выровненных упакованных целочисленных значений
vmovdqu	Присвоение невыровненных упакованных целочисленных значений
vpabs[b|w|d]	Абсолютное значение упакованного целого числа
vpackss[dw|wb]	Упаковка со знаковым насыщением
vpackus[dw|wb]	Упаковка с беззнаковым насыщением
vpadd[b|w|d|q]	Сложение упакованных целых чисел
vpadds[b|w]	Сложение упакованных целых чисел со знаковым насыщением
vpaddus[b|w]	Сложение упакованных целых чисел с беззнаковым насыщением
vpand	Побитовое И упакованных целых чисел
vpandn	Побитовое И-НЕ упакованных целых чисел
vpcmpeq[b|w|d|q]	Сравнение упакованных целых чисел на совпадение
vpcmpgt[b|w|d|q]	Сравнение упакованных целых чисел «больше, чем…»
vpextr[b|w|d|q]	Извлечение целого числа из регистра XMM
vphadd[w|d]	Горизонтальное сложение упакованных целых чисел
vphsub[w|d]	Горизонтальное вычитание упакованных целых чисел
vpinsr[b|w|d|q]	Размещение упакованного целого числа в регистре XMM
vpmaxs[b|w|d]	Максимум из упакованных целых чисел со знаком
vpmaxu[b|w|d]	Максимум из упакованных целых чисел без знака
vpmins[b|w|d]	Минимум из упакованных целых чисел со знаком
vpminu[b|w|d]	Минимум из упакованных целых чисел без знака
vpmovsx	Присвоение упакованного целого числа, расширенного знаком
vpmovzx	Присвоение упакованного целого числа, расширенного нулем
vpmuldq	Перемножение двухсловных упакованных целых чисел со знаком
vpmulhuw	Перемножение упакованных слов без знака, сохранение верхнего результата
vpmul[h|l]w	Перемножение упакованных слов со знаком, сохранение верхнего|нижнего результата
vpmull[d|w]	Перемножение упакованных слов без знака, сохранение нижнего результата
vpmuludq	Перемножение двухсловных упакованных чисел без знака
vpshuf[b|d]	Перемешивание упакованных целых чисел
vpshuf[h|l]w	Перемешивание верхних|нижних упакованных слов

Окончание табл. 4.9

Команда	Описание
vpslldq	Логический сдвиг влево двойных четверных слов
vpsll[w|d|q]	Логический сдвиг упакованных целых чисел влево
vpsra[w|d]	Арифметический сдвиг упакованных целых чисел вправо
vpsrldq	Логический сдвиг вправо упакованных целых чисел
vpsrl[w|d|q]	Логический сдвиг упакованных целых чисел вправо
vpsub[b|w|d|q]	Вычитание упакованных целых чисел
vpsubs[b|w]	Вычитание упакованных целых чисел со знаковым насыщением
vpsubus[b|w]	Вычитание упакованных целых чисел с беззнаковым насыщением
vpunpckh[bw|wd|dq]	Распаковка верхней дорожки данных
vpunpckl[bw|wd|dq]	Распаковка нижней дорожки данных

4.8. Различия между x86-AVX и x86-SSE

Если у вас есть опыт программирования на языке ассемблера x86-SSE, вы, несомненно, заметили, что существует высокая схожесть между этой средой выполнения и x86-AVX. Большинство команд x86-SSE имеют эквивалент x86-AVX, который может использовать операнды шириной 256 или 128 бит. Однако между средами выполнения x86-SSE и x86-AVX есть несколько важных различий. Остальная часть этого раздела объясняет эти различия. Даже если у вас нет опыта работы с x86-SSE, я все же рекомендую прочитать этот раздел, поскольку он разъясняет важные детали, о которых вам необходимо знать при написании кода, использующего набор команд x86-AVX.

В процессоре x86-64, который поддерживает x86-AVX, каждый 256-битный регистр YMM разделен на *верхнюю* и *нижнюю* 128-битные *дорожки* (lane). Многие команды x86-AVX выполняют свои операции с использованием элементов операндов источника и назначения в верхних и нижних дорожках по отдельности. Это независимое выполнение по дорожкам обычно незаметно при использовании команд x86-AVX, которые выполняют арифметические вычисления. Однако при использовании команд, переупорядочивающих элементы упакованных данных, эффект отдельных дорожек выполнения более очевиден. Например, команда `vshufps` (чередование упакованных значений с одинарной точностью) переупорядочивает элементы своих исходных операндов в соответствии с маской управления, указанной как непосредственный операнд. Команда `vpunpcklwd` (распаковать нижние данные) чередует младшие элементы в своих двух исходных операндах. На рис. 4.16 более подробно показано выполнение этих команд. Обратите внимание, что операции перемешивания и распаковки с плавающей запятой выполняются независимо как в верхнем (биты 255:128), так и в нижнем (биты 127:0) двойных четверных словах. Вы узнаете больше о командах `vshufps` и `vpunpcklwd` в главах 6 и 7.

Рис. 4.16. Примеры выполнения команд x86-AVX с использованием независимых дорожек

Использование псевдонимов наборов регистров XMM и YMM приводит к некоторым программным проблемам, о которых следует помнить разработчикам программного обеспечения. Первая проблема связана с обработкой процессором старших 128 бит регистра YMM, когда соответствующий регистр XMM используется в качестве операнда-приемника. При выполнении на процессоре, поддерживающем технологию x86-AVX, команда x86-SSE, которая использует регистр XMM в качестве операнда назначения, никогда не будет изменять старшие 128 бит соответствующего регистра YMM. Однако эквивалентная команда x86-AVX обнулит старшие 128 бит соответствующего регистра YMM. Рассмотрим, например, следующие экземпляры команды `(v)cvtps2pd` (преобразование упакованных чисел одинарной точности в упакованные числа двойной точности):

```
cvtps2pd xmm0, xmm1
vcvtps2pd xmm0, xmm1
vcvtps2pd ymm0, ymm1
```

Команда x86-SSE `cvtps2pd` преобразует два упакованных значения с плавающей запятой одинарной точности в младшем четверном слове XMM1 в числа с плавающей запятой двойной точности и сохраняет результат в регистре XMM0. Эта команда не изменяет старшие 128 бит регистра YMM0. Первая команда `vcvtps2pd` выполняет ту же операцию преобразования упакованного числа одинарной точности в упакованное число двойной точности; она также обнуляет старшие 128 бит YMM0. Вторая команда `vcvtps2pd` преобразует четыре упакованных значения с плавающей запятой одинарной точности в младших

128 битах YMM1 в упакованные значения с плавающей запятой двойной точности и сохраняет результат в YMM0.

x86-AVX ослабляет требования x86-SSE к выравниванию упакованных операндов в памяти. За исключением команд, которые явно указывают выровненный операнд (например, `vmovaps`, `vmovdqa` и т. д.), правильное выравнивание 128-битного или 256-битного операнда в памяти не является обязательным. Однако 128-битные и 256-битные операнды всегда должны быть должным образом выровнены, когда это возможно, чтобы предотвратить задержки обработки, которые могут возникнуть, когда процессор обращается к невыровненным операндам в памяти.

Последняя проблема, о которой должны знать программисты, связана со смешиванием кода x86-AVX и x86-SSE. Программам разрешено смешивать команды x86-AVX и x86-SSE, но любое смешивание должно быть сведено к минимуму, чтобы избежать штрафов за переход внутреннего состояния процессора, которые могут повлиять на производительность. Эти штрафы могут возникать, если процессору требуется сохранять старшие 128 бит каждого регистра YMM во время перехода от выполнения x86-AVX к выполнению команд x86-SSE. Штрафов за переход между состояниями можно полностью избежать, используя команду `vzeroupper`, которая обнуляет верхние 128 бит всех регистров YMM. Эту команду следует использовать перед любым переходом от 256-битного кода x86-AVX (т. е. любого кода x86-AVX, который использует регистр YMM) к коду x86-SSE.

Одним из распространенных способов использования команды `vzeroupper` является публичная функция, которая использует 256-битные команды x86-AVX. Функции этого типа должны содержать команду `vzeroupper` перед выполнением любой команды `ret`, поскольку это предотвращает возникновение штрафов за переход состояния процессора в любом коде языка высокого уровня, который использует команды x86-SSE. Инструкцию `vzeroupper` также следует использовать перед вызовом любых библиотечных функций, которые могут содержать код x86-SSE. Позже в этой книге вы увидите несколько примеров исходного кода, демонстрирующих правильное использование команды `vzeroupper`. Функции также могут использовать команду `vzeroall` (обнуление всех регистров YMM) вместо `vzeroupper`, чтобы избежать потенциальных штрафов за переход состояния x86-AVX/x86-SSE.

4.9. Заключение

Основные моменты главы 4 заключаются в следующем:

- технология AVX – это архитектурное усовершенствование платформы x86, упрощающее операции SIMD с использованием 128-битных и 256-битных упакованных операндов с плавающей запятой, как одинарной, так и двойной точности;
- AVX также поддерживает операции SIMD с использованием 128-битных упакованных целых чисел и скалярных операндов с плавающей запятой. AVX2 расширяет набор команд AVX для поддержки операций SIMD с использованием 256-битных упакованных целочисленных операндов;

- AVX добавляет к платформе x86-64 еще 16 регистров YMM (256-бит) и XMM (128-бит). Каждому регистру XMM сопоставляются младшие 128 бит соответствующего регистра YMM;
- большинство команд AVX используют синтаксис с тремя операндами, который включает два неразрушающих исходных операнда;
- операции с плавающей запятой AVX соответствуют стандарту IEEE 754 для операций арифметики с плавающей запятой;
- программы могут использовать флаги управления и состояния в регистре MXCSR для включения/отключения исключений, возникающих при выполнении операций с плавающей запятой, обнаружения ошибок с плавающей запятой и настройки округления с плавающей запятой;
- за исключением команд, которых явно требуют выровненные операнды, 128-битные и 256-битные операнды в памяти не требуют выравнивания. Однако операнды SIMD в памяти всегда должны быть по возможности правильно выровнены, чтобы избежать задержек, которые могут возникнуть, когда процессор обращается к невыровненному операнду в памяти;
- инструкцию `vzeroupper` или `vzeroall` следует использовать в любой функции, которая применяет регистр YMM в качестве операнда, чтобы избежать потенциальных потерь производительности при переходе между состояниями x86-AVX и x86-SSE.

Глава 5

Программирование AVX – скалярные вычисления с плавающей запятой

В предыдущей главе вы узнали об архитектуре и вычислительных возможностях AVX. В этой главе вы узнаете, как использовать набор команд AVX для выполнения скалярных вычислений с плавающей запятой. Первый раздел включает несколько примеров программ, которые иллюстрируют базовую скалярную арифметику с плавающей запятой – сложение, вычитание, умножение и деление. В следующем разделе содержится код, объясняющий использование скалярных команд сравнения и преобразования с плавающей запятой. Далее следуют два примера, демонстрирующих скалярные операции с плавающей запятой с использованием массивов и матриц. В последнем разделе этой главы формально описывается соглашение о вызовах Visual C++.

Для всего примера кода в этой главе требуется процессор и операционная система, поддерживающие AVX. Вы можете использовать один из свободно доступных инструментов, перечисленных в приложении, чтобы определить, соответствует ли ваш компьютер этому требованию. В главе 16 вы узнаете, как программно определять наличие AVX и других расширений функций процессора x86.

Примечание. Разработка программного обеспечения, использующего арифметику с плавающей запятой, всегда таит в себе подводные камни. Цель примеров кода, представленных в этой и последующих главах, – проиллюстрировать использование различных команд x86 с плавающей запятой. В примерах кода не рассматриваются важные проблемы с плавающей запятой, такие как ошибки округления, числовая стабильность или плохо обусловленные функции. Разработчики программного обеспечения всегда должны помнить об этих проблемах во время разработки и реализации любого алгоритма, использующего арифметику с плавающей запятой. Если вы хотите узнать больше о потенциальных ловушках арифметики с плавающей запятой, вам следует обратиться к ссылкам, перечисленным в приложении.

5.1. Скалярная арифметика с плавающей запятой

Скалярные возможности AVX для операций с плавающей запятой предоставляют программистам современную альтернативу ресурсам с плавающей запятой SSE2 и устаревшему модулю операций с плавающей запятой x87. Возможность использовать адресные регистры означает, что выполнение элементарных скалярных операций с плавающей запятой, таких как сложение, вычитание, умножение и деление, аналогична выполнению целочисленной арифметики с использованием регистров общего назначения. В этом разделе вы узнаете, как кодировать функции, которые выполняют базовые арифметические вычисления с плавающей запятой, используя набор команд AVX. Примеры исходного кода демонстрируют, как выполнять базовые операции, используя значения как с одинарной, так и с двойной точностью. Вы также узнаете о передаче аргументов с плавающей запятой, возвращаемых значениях и директивах MASM.

5.1.1. Арифметика с плавающей запятой одинарной точности

В листинге 5.1 (пример `Ch05_01`) показан исходный код на C++ и на языке ассемблера для простой программы, выполняющей преобразование температуры по Фаренгейту в градусы Цельсия с использованием арифметики с плавающей запятой одинарной точности. Код C++ начинается с объявления функции языка ассемблера `ConvertFtoC_`. Обратите внимание, что эта функция ожидает один аргумент типа `float` и возвращает значение типа `float`. Аналогичное объявление также используется для функции на языке ассемблера `ConvertCtoF_`. Оставшийся код C++ выполняет две функции преобразования температуры с использованием нескольких тестовых значений и отображает результаты.

Листинг 5.1. Пример Ch05_01

```
//-----------------------------------------------
//               Ch05_01.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" float ConvertFtoC_(float deg_f);
extern "C" float ConvertCtoF_(float deg_c);

int main()
{
    const int w = 10;
    float deg_fvals[] = {-459.67f, -40.0f, 0.0f, 32.0f, 72.0f, 98.6f, 212.0f};
    size_t nf = sizeof(deg_fvals) / sizeof(float);
```

```
    cout << setprecision(6);

    cout << "\n-------- Результаты ConvertFtoC --------\n";

    for (size_t i = 0; i < nf; i++)
    {
        float deg_c = ConvertFtoC_(deg_fvals[i]);

        cout << " i: " << i << " ";
        cout << "f: " << setw(w) << deg_fvals[i] << " ";
        cout << "c: " << setw(w) << deg_c << '\n';
    }

    cout << "\n-------- Результаты ConvertCtoF --------\n";

    float deg_cvals[] = {-273.15f, -40.0f, -17.777778f, 0.0f, 25.0f, 37.0f, 100.0f};
    size_t nc = sizeof(deg_cvals) / sizeof(float);

    for (size_t i = 0; i < nc; i++)
    {
        float deg_f = ConvertCtoF_(deg_cvals[i]);

        cout << " i: " << i << " ";
        cout << "c: " << setw(w) << deg_cvals[i] << " ";
        cout << "f: " << setw(w) << deg_f << '\n';
    }

    return 0;
}

;-----------------------------------------------
;                 Ch05_01.asm
;-----------------------------------------------

        .const
r4_ScaleFtoC real4 0.55555556 ; 5 / 9
r4_ScaleCtoF real4 1.8 ; 9 / 5
r4_32p0 real4 32.0

; extern "C" float ConvertFtoC_(float deg_f)
;
; Возвращает: xmm0[31:0] = температура в градусах Цельсия.

        .code
ConvertFtoC_ proc
        vmovss xmm1,[r4_32p0]           ;xmm1 = 32
        vsubss xmm2,xmm0,xmm1           ;xmm2 = f - 32

        vmovss xmm1,[r4_ScaleFtoC]      ;xmm1 = 5 / 9
        vmulss xmm0,xmm2,xmm1           ;xmm0 = (f - 32) * 5 / 9
        ret
ConvertFtoC_ endp

; extern "C" float CtoF_(float deg_c)
;
; Возвращает: xmm0[31:0] = температура в градусах Фаренгейта.
```

```
ConvertCtoF_ proc
        vmulss xmm0,xmm0,[r4_ScaleCtoF] ;xmm0 = c * 9 / 5
        vaddss xmm0,xmm0,[r4_32p0]      ;xmm0 = c * 9 / 5 + 32
        ret
ConvertCtoF_ endp

end
```

Код на языке ассемблера начинается с раздела `.const`, в котором определены константы, необходимые для преобразования значения температуры из градусов Фаренгейта в градусы Цельсия и наоборот. Мнемоника `real4` – это директива MASM, которая выделяет место для хранения значения с плавающей запятой одинарной точности. После раздела `.const` следует код функции `ConvertFtoC_`. Первая команда этой функции, `vmovss xmm1,[r4_32p0]`, загружает значение 32,0 с плавающей запятой одинарной точности из памяти в регистр XMM1 (или, точнее, в XMM1[31:0]). Здесь используется операнд в памяти, поскольку, в отличие от регистров общего назначения, значения с плавающей запятой не могут использоваться как непосредственные операнды.

В соответствии с соглашением о вызовах Visual C++ первые четыре значения аргумента с плавающей запятой передаются функции с использованием регистров XMM0, XMM1, XMM2 и XMM3. Это означает, что при входе в функцию `ConvertFtoC_` регистр XMM0 содержит значение аргумента `deg_f`. После выполнения команды `vmovss` команда `vsubss xmm2,xmm0,xmm1` вычисляет `deg_f-32.0` и сохраняет результат в XMM2. Выполнение команды `vsubss` не изменяет содержимое исходных операндов XMM0 и XMM1. Эта команда также копирует биты XMM0 [127:32] в XMM2 [127:32]. Следующая команда `vmovss xmm1,[r4_ScaleFtoC]` загружает постоянное значение 0,55555556 (или 5/9) в регистр XMM1. За ней следует команда `vmulss xmm0,xmm2,xmm1`, которая вычисляет `(deg_f-32.0)*0.55555556` и сохраняет результат умножения (то есть температуру, преобразованную в градусы Цельсия) в XMM0. Соглашение о вызовах Visual C++ определяет регистр XMM0 для возвращаемых значений с плавающей запятой. Поскольку возвращаемое значение уже находится в XMM0, никаких дополнительных команд `vmovss` не требуется.

Далее следует функция на языке ассемблера `ConvertCtoF_`. Код этой функции немного отличается от `ConvertFtoC_` тем, что арифметические команды с плавающей запятой используют операнды памяти для ссылки на требуемые константы преобразования. При входе в `ConvertCtoF_` регистр XMM0 содержит значение аргумента `deg_c`. Инструкция `vmulss xmm0,xmm0,[r4_ScaleCtoF]` вычисляет `deg_c*1.8`. Затем следует команда `vaddss xmm0,xmm0,[r4_32p0]`, которая вычисляет `deg_c*1.8+32.0`. На этом этапе с научной точки зрения было бы упущением не упомянуть, что ни `ConvertFtoC_`, ни `ConvertCtoF_` не выполняют никаких проверок корректности значений аргументов, которые физически невозможны (например, –1000 градусов по Фаренгейту). Для таких проверок требуются команды сравнения с плавающей запятой, и вы узнаете, как использовать эти команды, позже в этой главе. Ниже приведены результаты выполнения примера `Ch05_01`:

```
-------- Результаты ConvertFtoC --------
i: 0  f:  -459.67  c: -273.15
i: 1  f:      -40  c: -40
i: 2  f:        0  c: -17.7778
i: 3  f:       32  c: 0
i: 4  f:       72  c: 22.2222
i: 5  f:     98.6  c: 37
i: 6  f:      212  c: 100

-------- Результаты ConvertCtoF --------
i: 0  c:  -273.15  f: -459.67
i: 1  c:      -40  f: -40
i: 2  c: -17.7778  f: 0
i: 3  c:        0  f: 32
i: 4  c:       25  f: 77
i: 5  c:       37  f: 98.6
i: 6  c:      100  f: 212
```

5.1.1. Арифметика с плавающей запятой двойной точности

Примеры исходного кода, представленные в этом разделе, иллюстрируют простую арифметику с плавающей запятой с использованием значений двойной точности. В листинге 5.2 показан исходный код примера `Ch05_02`. В этом примере функция на языке ассемблера `CalcSphereAreaVolume_` вычисляет площадь поверхности и объем сферы, используя предоставленное значение радиуса.

Листинг 5.2. Пример Ch05_02

```
//-----------------------------------------------
//                Ch05_02.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" void CalcSphereAreaVolume_(double r, double* sa, double* vol);

int _tmain(int argc, _TCHAR* argv[])
{
    double r[] = { 0.0, 1.0, 2.0, 3.0, 5.0, 10.0, 20.0, 32.0 };
    size_t num_r = sizeof(r) / sizeof(double);
    cout << setprecision(8);
    cout << "\n--------- Результаты CalcSphereAreaVol -----------\n";
    for (size_t i = 0; i < num_r; i++)
    {
        double sa = -1, vol = -1;
        CalcSphereAreaVolume_(r[i], &sa, &vol);
        cout << "i: " << i << " ";
        cout << "r: " << setw(6) << r[i] << " ";
        cout << "sa: " << setw(11) << sa << " ";
        cout << "vol: " << setw(11) << vol << '\n';
    }
```

```
    return 0;
}

;-----------------------------------------------
;                Ch05_02.asm
;-----------------------------------------------

        .const
r8_PI real8 3.14159265358979323846
r8_4p0 real8 4.0
r8_3p0 real8 3.0

; extern "C" void CalcSphereAreaVolume_(double r, double* sa, double* vol);

        .code
CalcSphereAreaVolume_ proc

; Площадь поверхности = 4 * PI * r * r
        vmulsd xmm1,xmm0,xmm0           ;xmm1 = r * r
        vmulsd xmm2,xmm1,[r8_PI]        ;xmm2 = r * r * PI
        vmulsd xmm3,xmm2,[r8_4p0]       ;xmm3 = r * r * PI * 4

; Объем = sa * r / 3
        vmulsd xmm4,xmm3,xmm0           ;xmm4 = r * r * r * PI * 4
        vdivsd xmm5,xmm4,[r8_3p0]       ;xmm5 = r * r * r * PI * 4 / 3

; Сохранение результатов
        vmovsd real8 ptr [rdx],xmm3     ;сохранение площади поверхности
        vmovsd real8 ptr [r8],xmm5      ;сохранение объема
        ret
CalcSphereAreaVolume_ endp
        end
```

Объявление функции `CalcSphereAreaVolume_` включает значение аргумента типа `double` для радиуса и два указателя `double*` для возврата вычисленной площади поверхности и объема. Площадь поверхности и объем шара можно рассчитать по следующим формулам:

$$sa = 4\pi^2;$$

$$v = \frac{4\pi r^3}{3} = (sa)r/3.$$

Как и в предыдущем примере, код на языке ассемблера начинается с раздела `.const`, который определяет несколько констант. Запись `real8` – это директива MASM, определяющая пространство для хранения значений с плавающей запятой двойной точности. При входе в `CalcSphereAreaVolume_` XMM0 содержит радиус сферы. Команда `vmulsd xmm1,xmm0,xmm0` возводит радиус в квадрат и сохраняет результат в XMM1. Выполнение этой команды также копирует старшие 64 бита XMM0 в те же позиции в XMM1 (т. е. XMM0 [127:64] копируется в XMM1 [127:64]). Следующие команды `vmulsd xmm2,xmm1,[r8_PI]` и `vmulsd xmm3,xmm2,[r8_4p0]` вычисляют `r*r*PI*4`, что дает площадь поверхности сферы.

Следующие две команды, `vmulsd xmm4,xmm3,xmm0` и `vdivsd xmm5,xmm4,[r8_3p0]`, вычисляют объем сферы. Команды `vmovsd real8 ptr[rdx],xmm3` и `vmovsd real8 ptr[r8],xmm5` сохраняют вычисленные значения площади поверхности и объема в указанные буферы. Обратите внимание, что аргументы указателя `sa` и `vol` были переданы в `CalcSphereAreaVolume_` в регистрах RDX и R8. Когда функция использует комбинацию целочисленных аргументов (или указателей) и аргументов с плавающей запятой, позиция оных в объявлении функции определяет, какие регистры общего назначения или регистры XMM будут использоваться. Вы узнаете больше об этом аспекте соглашения о вызовах Visual C++ позже в этой главе. Вот результат выполнения кода примера `Ch05_02`.

```
--------- Результаты CalcSphereAreaVol -----------
i: 0  r:   0   sa:          0   vol:         0
i: 1  r:   1   sa:  12.566371   vol: 4.1887902
i: 2  r:   2   sa:  50.265482   vol: 33.510322
i: 3  r:   3   sa:  113.09734   vol: 113.09734
i: 4  r:   5   sa:  314.15927   vol: 523.59878
i: 5  r:  10   sa:  1256.6371   vol: 4188.7902
i: 6  r:  20   sa:  5026.5482   vol: 33510.322
i: 7  r:  32   sa:  12867.964   vol: 137258.28
```

Листинг 5.3 (пример `Ch05_03`) содержит код следующего примера, который также показывает, как выполнять вычисления с использованием арифметики с плавающей запятой двойной точности. В этом примере функция языка ассемблера `CalcDistance_` вычисляет евклидово расстояние между двумя точками в трехмерном пространстве, используя следующее уравнение:

$$dist = \sqrt{\left(x_2 - x_1\right)^2 + \left(y_2 - y_1\right)^2 + \left(z_2 - z_1\right)^2}.$$

Листинг 5.3. Пример Ch05_03

```
//-----------------------------------------------
//                Ch05_03.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <cmath>

using namespace std;

extern "C" double CalcDistance_(double x1, double y1, double z1, double x2, double y2,
double z2);

void Init(double* x, double* y, double* z, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 100};
    default_random_engine rng {seed};
```

```
    for (size_t i = 0; i < n; i++)
    {
        x[i] = ui_dist(rng);
        y[i] = ui_dist(rng);
        z[i] = ui_dist(rng);
    }
}

double CalcDistanceCpp(double x1, double y1, double z1, double x2, double y2, double z2)
{
    double tx = (x2 - x1) * (x2 - x1);
    double ty = (y2 - y1) * (y2 - y1);
    double tz = (z2 - z1) * (z2 - z1);
    double dist = sqrt(tx + ty + tz);

    return dist;
}

int main()
{
    const size_t n = 20;
    double x1[n], y1[n], z1[n];
    double x2[n], y2[n], z2[n];
    double dist1[n];
    double dist2[n];

    Init(x1, y1, z1, n, 29);
    Init(x2, y2, z2, n, 37);

    for (size_t i = 0; i < n; i++)
    {
        dist1[i] = CalcDistanceCpp(x1[i], y1[i], z1[i], x2[i], y2[i], z2[i]);
        dist2[i] = CalcDistance_(x1[i], y1[i], z1[i], x2[i], y2[i], z2[i]);
    }

    cout << fixed;

    for (size_t i = 0; i < n; i++)
    {
        cout << "i: " << setw(2) << i << " ";
        cout << setprecision(0);
        cout << "p1(";
        cout << setw(3) << x1[i] << ",";
        cout << setw(3) << y1[i] << ",";
        cout << setw(3) << z1[i] << ") | ";

        cout << "p2(";
        cout << setw(3) << x2[i] << ",";
        cout << setw(3) << y2[i] << ",";
        cout << setw(3) << z2[i] << ") | ";

        cout << setprecision(4);
        cout << "dist1: " << setw(8) << dist1[i] << " | ";
        cout << "dist2: " << setw(8) << dist2[i] << '\n';
    }

    return 0;
}
```

```
;-----------------------------------------------
;               Ch05_03.asm
;-----------------------------------------------

; extern "C" double CalcDistance_(double x1, double y1, double z1, double x2, double y2,
double z2)

        .code
CalcDistance_ proc
; Чтение аргументов из стека
        vmovsd xmm4,real8 ptr [rsp+40]  ;xmm4 = y2
        vmovsd xmm5,real8 ptr [rsp+48]  ;xmm5 = z2

; Вычисление квадратов расстояний между координатами
        vsubsd xmm0,xmm3,xmm0           ;xmm0 = x2 - x1
        vmulsd xmm0,xmm0,xmm0           ;xmm0 = (x2 - x1) * (x2 - x1)

        vsubsd xmm1,xmm4,xmm1           ;xmm1 = y2 - y1
        vmulsd xmm1,xmm1,xmm1           ;xmm1 = (y2 - y1) * (y2 - y1)

        vsubsd xmm2,xmm5,xmm2           ;xmm2 = z2 - z1
        vmulsd xmm2,xmm2,xmm2           ;xmm2 = (z2 - z1) * (z2 - z1)

; Вычисление окончательного расстояния
        vaddsd xmm3,xmm0,xmm1
        vaddsd xmm4,xmm2,xmm3           ;xmm4 = сумма квадратов
        vsqrtsd xmm0,xmm0,xmm4          ;xmm0 = значение расстояния
        ret
CalcDistance_ endp
        end
```

Если вы изучите объявление функции `CalcDistance_`, то заметите, что она определяет шесть значений аргумента двойной точности. Значения аргументов `x1`, `y1`, `z1` и `x2` передаются в регистры XMM0, XMM1, XMM2 и XMM3 соответственно. Последние два значения аргумента, `y2` и `z2`, передаются в стек, как показано на рис. 5.1. Обратите внимание, что на этом рисунке показано только младшее четверное слово каждого регистра XMM; старшие четверные слова не используются для передачи значений аргументов и не определены.

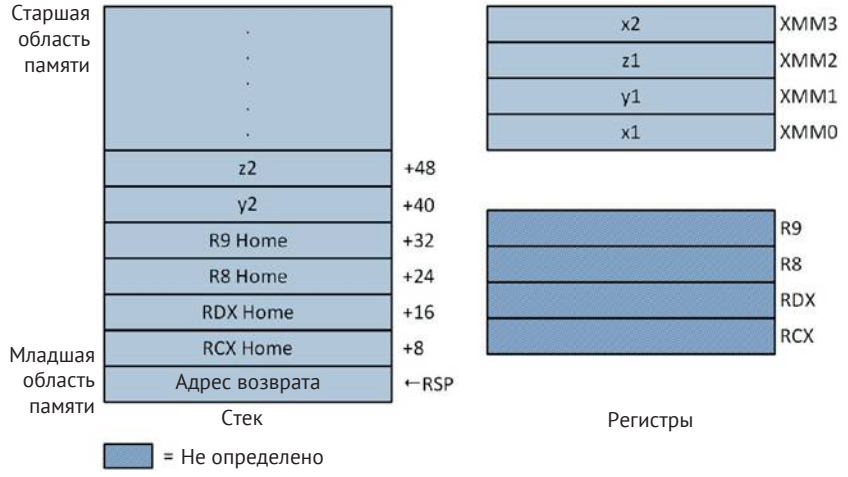


Рис. 5.1. Структура стека и регистры аргументов при входе в `CalcDistance_`

Функция `CalcDistance_` начинается с команды `vmovsd xmm4,real8 ptr [rsp+40]`, которая загружает значение аргумента `y2` из стека в регистр XMM4. За ней следует команда `vmovsd xmm5,real8 ptr [rsp+48]`, которая загружает значение аргумента `z2` в регистр XMM5. Следующие две команды, `vsubsd xmm0,xmm3,xmm0` и `vmulsd xmm0,xmm0,xmm0`, вычисляют `(x2-x1)*(x2-x1)`. Затем аналогичная последовательность команд используется для вычисления `(y2-y1)*(y2-y1)` и `(z2-z1)*(z2-z1)`. За ними следуют две команды `vaddsd`, которые суммируют три квадрата координат. Инструкция `vsqrtsd xmm0,xmm0,xmm4` вычисляет окончательное расстояние. Обратите внимание, что команда `vsqrtsd` вычисляет квадратный корень из второго операнда-источника. Подобно другим скалярным арифметическим командам с плавающей запятой двойной точности, `vsqrtsd` также копирует биты 127:64 первого исходного операнда в те же битовые позиции операнда-приемника. Вот результат выполнения примера `Ch05_03`:

```
i:  0  p1( 86, 84,  5) | p2( 32,  8, 77) | dist1: 117.7964 | dist2: 117.7964
i:  1  p1( 38, 63, 77) | p2( 28, 49, 86) | dist1:  19.4165 | dist2:  19.4165
i:  2  p1( 17, 18, 54) | p2( 79, 51, 80) | dist1:  74.8933 | dist2:  74.8933
i:  3  p1( 85, 50, 28) | p2( 40, 87, 90) | dist1:  85.0764 | dist2:  85.0764
i:  4  p1( 98, 47, 79) | p2( 28, 85, 38) | dist1:  89.5824 | dist2:  89.5824
i:  5  p1( 21, 78, 36) | p2( 92, 12, 47) | dist1:  97.5602 | dist2:  97.5602
i:  6  p1( 16, 50, 97) | p2( 61, 13, 40) | dist1:  81.5046 | dist2:  81.5046
i:  7  p1( 31, 96, 49) | p2( 31, 37, 45) | dist1:  59.1354 | dist2:  59.1354
i:  8  p1( 13, 87, 40) | p2( 95, 41, 87) | dist1: 105.1142 | dist2: 105.1142
i:  9  p1( 35, 48,  4) | p2( 26, 13, 43) | dist1:  53.1695 | dist2:  53.1695
i: 10  p1( 43, 56, 85) | p2( 88, 17, 45) | dist1:  71.7356 | dist2:  71.7356
i: 11  p1( 59, 88, 77) | p2( 26, 11, 72) | dist1:  83.9226 | dist2:  83.9226
i: 12  p1( 56, 48, 71) | p2(  3, 56, 81) | dist1:  54.5252 | dist2:  54.5252
i: 13  p1( 97, 19, 11) | p2( 36, 35, 58) | dist1:  78.6511 | dist2:  78.6511
i: 14  p1( 50, 79, 74) | p2( 60,  7, 32) | dist1:  83.9524 | dist2:  83.9524
i: 15  p1( 84, 16, 29) | p2( 91,  4, 91) | dist1:  63.5374 | dist2:  63.5374
i: 16  p1( 67, 77, 65) | p2( 86, 47, 59) | dist1:  36.0139 | dist2:  36.0139
i: 17  p1( 67,  1,  3) | p2( 34, 19, 64) | dist1:  71.6519 | dist2:  71.6519
i: 18  p1( 41, 79, 73) | p2( 17,  2, 68) | dist1:  80.8084 | dist2:  80.8084
i: 19  p1( 86, 40, 66) | p2( 76, 12, 61) | dist1:  30.1496 | dist2:  30.1496
```

5.2. Скалярные сравнения и преобразования с плавающей запятой

Любая функция, которая выполняет операции базовой арифметики с плавающей запятой, также может выполнять операции сравнения с плавающей запятой и преобразования между целочисленными значениями и значениями с плавающей запятой. Примеры исходного кода в этом разделе иллюстрируют, как выполнять скалярное сравнение с плавающей запятой и преобразование данных. Раздел начинается с пары примеров, демонстрирующих методы сравнения двух значений с плавающей запятой и принятия логического решения на основе результата. Далее следует пример, демонстрирующий операции преобразования с плавающей запятой с использованием значений разных типов.

5.2.1. Сравнение с плавающей запятой

В листинге 5.4 показан исходный код примера `Ch05_04`, который демонстрирует использование команд сравнения с плавающей запятой `vcomis[d|s]`. Подобно скалярным арифметическим командам AVX с плавающей запятой, последняя буква этих мнемоник обозначает тип операнда (`d` = двойная точность, `s` = одинарная точность). Команды `vcomis[d|s]` сравнивают два операнда с плавающей запятой и устанавливают флаги состояния в RFLAGS для обозначения одного из следующих результатов: меньше, равно, больше или не определено. Неопределенное состояние сравнения значений с плавающей запятой возникает, когда один или оба операнда команды имеют значение NaN или ошибочно закодированы. Функции на языке ассемблера `CompareVCOMISD_` и `CompareVCOMISS_` иллюстрируют использование команд `vcomisd` и `vcomiss` соответственно. Далее я опишу работу функции `CompareVCOMISS_`; ее описание также относится к `CompareVCOMISD_`.

Листинг 5.4. Пример Ch05_04

```
//-----------------------------------------------
//               Ch05_04.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <string>
#include <iostream>
#include <iomanip>
#include <limits>

using namespace std;

extern "C" void CompareVCOMISS_(float a, float b, bool* results);
extern "C" void CompareVCOMISD_(double a, double b, bool* results);

const char* c_OpStrings[] = {"UO", "LT", "LE", "EQ", "NE", "GT", "GE"};
const size_t c_NumOpStrings = sizeof(c_OpStrings) / sizeof(char*);

const string g_Dashes(72, '-');

template <typename T> void PrintResults(T a, T b, const bool* cmp_results)
{
    cout << "a = " << a << ", ";
    cout << "b = " << b << '\n';

    for (size_t i = 0; i < c_NumOpStrings; i++)
    {
        cout << c_OpStrings[i] << '=';
        cout << boolalpha << left << setw(6) << cmp_results[i] << ' ';
    }

    cout << "\n\n";
}
```

```
void CompareVCOMISS()
{
    const size_t n = 6;
    float a[n] {120.0, 250.0, 300.0, -18.0, -81.0, 42.0};
    float b[n] {130.0, 240.0, 300.0, 32.0, -100.0, 0.0};

    // Тестовое значение NaN
    b[n - 1] = numeric_limits<float>::quiet_NaN();

    cout << "\nРезультаты CompareVCOMISS\n";
    cout << g_Dashes << '\n';

    for (size_t i = 0; i < n; i++)
    {
        bool cmp_results[c_NumOpStrings];

        CompareVCOMISS_(a[i], b[i], cmp_results);
        PrintResults(a[i], b[i], cmp_results);
    }
}

void CompareVCOMISD(void)
{
    const size_t n = 6;
    double a[n] {120.0, 250.0, 300.0, -18.0, -81.0, 42.0};
    double b[n] {130.0, 240.0, 300.0, 32.0, -100.0, 0.0};

    // Тестовое значение NaN
    b[n - 1] = numeric_limits<double>::quiet_NaN();

    cout << "\nРезультаты CompareVCOMISD\n";
    cout << g_Dashes << '\n';

    for (size_t i = 0; i < n; i++)
    {
        bool cmp_results[c_NumOpStrings];

        CompareVCOMISD_(a[i], b[i], cmp_results);
        PrintResults(a[i], b[i], cmp_results);
    }
}

int main()
{
    CompareVCOMISS();
    CompareVCOMISD();
    return 0;
}

;-----------------------------------------------
;               Ch05_04.asm
;-----------------------------------------------

; extern "C" void CompareVCOMISS_(float a, float b, bool* results);

        .code
```

```
CompareVCOMISS_ proc

; Set result flags based on compare status
        vcomiss xmm0,xmm1
        setp byte ptr [r8]              ;RFLAGS.PF = 1, если не определено
        jnp @F
        xor al,al
        mov byte ptr [r8+1],al          ;значения результата по умолчанию
        mov byte ptr [r8+2],al
        mov byte ptr [r8+3],al
        mov byte ptr [r8+4],al
        mov byte ptr [r8+5],al
        mov byte ptr [r8+6],al
        jmp Done

@@:     setb byte ptr [r8+1]            ;установить байт, если a < b
        setbe byte ptr [r8+2]           ;установить байт, если a <= b
        sete byte ptr [r8+3]            ;установить байт, если a == b
        setne byte ptr [r8+4]           ;установить байт, если a != b
        seta byte ptr [r8+5]            ;установить байт, если a > b
        setae byte ptr [r8+6]           ;установить байт, если a >= b

Done: ret
CompareVCOMISS_ endp

; extern "C" void CompareVCOMISD_(double a, double b, bool* results);

CompareVCOMISD_ proc

; Установка флагов результата, исходя из результата сравнения
        vcomisd xmm0,xmm1
        setp byte ptr [r8]              ;RFLAGS.PF = 1, если не определено
        jnp @F
        xor al,al
        mov byte ptr [r8+1],al          ;значения результата по умолчанию
        mov byte ptr [r8+2],al
        mov byte ptr [r8+3],al
        mov byte ptr [r8+4],al
        mov byte ptr [r8+5],al
        mov byte ptr [r8+6],al
        jmp Done

@@:     setb byte ptr [r8+1]            ;установить байт, если a < b
        setbe byte ptr [r8+2]           ;установить байт, если a <= b
        sete byte ptr [r8+3]            ;установить байт, если a == b
        setne byte ptr [r8+4]           ;установить байт, если a != b
        seta byte ptr [r8+5]            ;установить байт, если a > b
        setae byte ptr [r8+6]           ;установить байт, если a >= b

Done:   ret
CompareVCOMISD_ endp
        end
```

Функция `CompareVCOMISS_` принимает два значения аргумента типа `float` и указатель на массив `bools` для результатов сравнения. Первая команда `CompareVCOMISS_`, `vcomiss xmm0,xmm1`, выполняет сравнение значений аргументов `a` и

b с плавающей запятой одинарной точности. Обратите внимание, что эти значения были переданы в CompareVCOMISS_ в регистрах XMM0 и XMM1. Выполнение vcomiss устанавливает RFLAGS.ZF, RFLAGS.PF и RFLAGS.ZF, как показано в табл. 5.1. Установка этих флагов состояния облегчает использование условных команд cmovcc, jcc и setcc, как показано в табл. 5.2.

Таблица 5.1. Флаги состояния, устанавливаемые командами vcomis[d|s]

Условие	RFLAGS.ZF	RFLAGS.PF	RFLAGS.CF
XMM0 > XMM1	0	0	0
XMM0 == XMM1	1	0	0
XMM0 < XMM1	0	0	1
Не определено	1	1	1

Таблица 5.2. Условные коды после выполнения команды vcomis[d|s]

Оператор отношения	Условный код	Состояние RFLAGS
XMM0 < XMM1	Ниже (b)	CF == 1
XMM0 <= XMM1	Ниже или равно (be)	CF == 1 || ZF == 1
XMM0 == XMM1	Равно (e или z)	ZF == 1
XMM0 1 = XMM1	Не равно (ne или nz)	ZF == 0
XMM0 > XMM1	Выше (a)	CF == 0 && ZF == 0
XMM0 >= XMM1	Выше или равно (ae)	CF == 0
Не определено	Четность (p)	PF == 1

Следует отметить, что флаги состояния, показанные в табл. 5.1, устанавливаются только в том случае, если исключения с плавающей запятой замаскированы (состояние по умолчанию для Visual C++) и ни один из операндов команды vcomis[d|s] не является QNaN, SNaN или денормализованным. Если недопустимая операция с плавающей запятой или исключения денормализованности не маскируются (MXCSR.IM = 0 или MXCSR.DM = 0) и одним из операндов сравнения является QNaN, SNaN или денормализованным, процессор сгенерирует исключение без обновления флагов состояния в RFLAGS. Глава 4 содержит дополнительную информацию об использовании регистра MXCSR, QNaN, SNaN и денормализованных значений.

После выполнения команды vcomiss xmm0, xmm1 следует серия команд setcc (установить байт по условию), чтобы выделить операторы отношения, показанные в табл. 5.2. Команда setp byte ptr [r8] устанавливает байт операнда-адресата в 1, если установлен RFLAGS.PF (т. е. один из операндов – QNaN или SNaN); в противном случае байт операнда назначения устанавливается в 0. Если сравнение определено, оставшиеся команды setcc в CompareVCOMISS_ сохраняют все возможные результаты сравнения, устанавливая для каждой записи в результатах массива значение 0 или 1. Как упоминалось ранее, функции также могут использовать jcc и команды cmovcc после выполнения команды vcomis[d|s] для выполнения программных переходов или условных перемеще-

ний данных на основе результата сравнения с плавающей запятой. Вот результат выполнения примера кода Ch05_04:

```
Результаты CompareVCOMISS
-------------------------------------------------------------------------
a = 120, b = 130
UO=false LT=true LE=true EQ=false NE=true GT=false GE=false

a = 250, b = 240
UO=false LT=false LE=false EQ=false NE=true GT=true GE=true

a = 300, b = 300
UO=false LT=false LE=true EQ=true NE=false GT=false GE=true

a = -18, b = 32
UO=false LT=true LE=true EQ=false NE=true GT=false GE=false

a = -81, b = -100
UO=false LT=false LE=false EQ=false NE=true GT=true GE=true

a = 42, b = nan
UO=true LT=false LE=false EQ=false NE=false GT=false GE=false

Результаты CompareVCOMISD
-------------------------------------------------------------------------
a = 120, b = 130
UO=false LT=true LE=true EQ=false NE=true GT=false GE=false

a = 250, b = 240
UO=false LT=false LE=false EQ=false NE=true GT=true GE=true

a = 300, b = 300
UO=false LT=false LE=true EQ=true NE=false GT=false GE=true

a = -18, b = 32
UO=false LT=true LE=true EQ=false NE=true GT=false GE=false

a = -81, b = -100
UO=false LT=false LE=false EQ=false NE=true GT=true GE=true

a = 42, b = nan
UO=true LT=false LE=false EQ=false NE=false GT=false GE=false
```

Листинг 5.5 содержит исходный код примера Ch05_05. В этом примере показано использование команды vcmpsd, которая сравнивает два значения с плавающей запятой двойной точности с использованием предиката сравнения, указанного как непосредственный операнд. Команда vcmpsd не использует какие-либо биты состояния в RFLAGS для индикации результатов сравнения. Вместо этого она возвращает маску четверного слова из всех единиц или всех нулей, чтобы указать истинный или ложный результат. Набор команд AVX также включает vcmpss, которую можно использовать для выполнения сравнений с плавающей запятой одинарной точности. Она эквивалентна команде vcmpsd, за исключением того, что возвращает маску двойного слова.

Листинг 5.5. Пример Ch05_05

```
//-----------------------------------------------
//               Ch05_05.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <limits>
#include <string>

using namespace std;

extern "C" void CompareVCMPSD_(double a, double b, bool* results);

const string g_Dashes(40, '-');

int main()
{
    const char* cmp_names[] =
    {
        "cmp_eq", "cmp_neq", "cmp_lt", "cmp_le",
        "cmp_gt", "cmp_ge", "cmp_ord", "cmp_unord"
    };

    const size_t num_cmp_names = sizeof(cmp_names) / sizeof(char*);

    const size_t n = 6;
    double a[n] = {120.0, 250.0, 300.0, -18.0, -81.0, 42.0};
    double b[n] = {130.0, 240.0, 300.0, 32.0, -100.0, 0.0};

    b[n - 1] = numeric_limits<double>::quiet_NaN();

    cout << "Результаты CompareVCMPSD\n";
    cout << g_Dashes << '\n';

    for (size_t i = 0; i < n; i++)
    {
        bool cmp_results[num_cmp_names];

        CompareVCMPSD_(a[i], b[i], cmp_results);

        cout << "a = " << a[i] << " ";
        cout << "b = " << b[i] << '\n';

        for (size_t j = 0; j < num_cmp_names; j++)
        {
            string s1 = cmp_names[j] + string(":");
            string s2 = ((j & 1) != 0) ? "\n" : " ";

            cout << left << setw(12) << s1;
            cout << boolalpha << setw(6) << cmp_results[j] << s2;
        }

        cout << "\n";
    }

return 0;
}
```

```
;-----------------------------------------------
;                 cmpequ.asmh
;-----------------------------------------------

; Базовые предикативы сравнения
CMP_EQ          equ 00h
CMP_LT          equ 01h
CMP_LE          equ 02h
CMP_UNORD       equ 03h
CMP_NEQ         equ 04h
CMP_NLT         equ 05h
CMP_NLE         equ 06h
CMP_ORD         equ 07h

; Расширенные предикативы сравнения для AVX
CMP_EQU_UQ      equ 08h
CMP_NGE         equ 09h
CMP_NGT         equ 0Ah
CMP_FALSE       equ 0Bh
CMP_NEQ_OQ      equ 0Ch
CMP_GE          equ 0Dh
CMP_GT          equ 0Eh
CMP_TRUE        equ 0Fh
CMP_EQ_OS       equ 10h
CMP_LT_OQ       equ 11h
CMP_LE_OQ       equ 12h
CMP_UNORD_S     equ 13h
CMP_NEQ_US      equ 14h
CMP_NLT_UQ      equ 15h
CMP_NLE_UQ      equ 16h
CMP_ORD_S       equ 17h
CMP_EQ_US       equ 18h
CMP_NGE_UQ      equ 19h
CMP_NGT_UQ      equ 1Ah
CMP_FALSE_OS    equ 1Bh
CMP_NEQ_OS      equ 1Ch
CMP_GE_OQ       equ 1Dh
CMP_GT_OQ       equ 1Eh
CMP_TRUE_US     equ 1Fh

;-----------------------------------------------
;                Ch05_05.asm
;-----------------------------------------------

        include <cmpequ.asmh>

; extern "C" void CompareVCMPSD_(double a, double b, bool* results)

        .code
CompareVCMPSD_ proc

; Проверка на равенство
        vcmpsd xmm2,xmm0,xmm1,CMP_EQ    ;операция сравнения
        vmovq rax,xmm2                  ;rax = результат сравнения (все 1 или 0)
        and al,1                        ;маскировка ненужных разрядов
        mov byte ptr [r8],al            ;сохранение результата как C++ bool
```

```
; Проверка на неравенство
        vcmpsd xmm2,xmm0,xmm1,CMP_NEQ
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+1],al

; Проверка на условие "меньше, чем"
        vcmpsd xmm2,xmm0,xmm1,CMP_LT
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+2],al

; Проверка на условие "меньше или равно, чем"
        vcmpsd xmm2,xmm0,xmm1,CMP_LE
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+3],al

; Проверка на условие "больше, чем"
        vcmpsd xmm2,xmm0,xmm1,CMP_GT
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+4],al

; Проверка на условие "больше или равно, чем"
        vcmpsd xmm2,xmm0,xmm1,CMP_GE
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+5],al

; Проверка на определенность
        vcmpsd xmm2,xmm0,xmm1,CMP_ORD
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+6],al

; Проверка на неопределенность
        vcmpsd xmm2,xmm0,xmm1,CMP_UNORD
        vmovq rax,xmm2
        and al,1
        mov byte ptr [r8+7],al

        ret
CompareVCMPSD_ endp
        end
```

Как и в предыдущем примере, код C++ примера `Ch05_05` содержит несколько тестовых операций, которые проверяют функцию на языке ассемблера `CompareVCMPSD_`. За кодом C++ в листинге 5.5 следует заголовочный файл языка ассемблера `cmpequ.asmh`. Этот файл содержит набор директив `equate`, которые используются для присвоения символьных имен числовым значениям. Директивы `equate` в `cmpequ.asmh` определяют символические имена для предикатов сравнения, которые используются рядом скалярных и упакованных команд сравнения x86-AVX, включая `vcmpsd`. Вы скоро увидите, как это работает.

Стандартного расширения файла для заголовочного файла ассемблера x86 не существует; я использую `.asmh`, но также часто применяется `.inc`.

Использование файла заголовка на языке ассемблера аналогично использованию файла заголовка C++. В текущем примере команда `include <cmpequ.asmh>` включает содержимое `cmpequ.asmh` в файл `Ch05_05_.asm` во время сборки. Угловые скобки, окружающие имя файла, можно опустить, если имя файла не содержит обратной косой черты или специальных символов MASM, но обычно проще и надежнее использовать их всегда. Помимо операторов равенства, файлы заголовков на ассемблере часто используются для определений макросов. Позже в этой главе вы узнаете о макросах.

Первая команда функции `CompareVCMPSD_`, `vcmpsd xmm2,xmm0,xmm1,CMP_EQ` проверяет содержимое регистров XMM0 и XMM1 на равенство. Эти регистры содержат значения аргументов `a` и `b`. Если `a` и `b` равны, четверное слово младшего разряда XMM2 заполняется единицами; в противном случае оно заполняется нулями. Обратите внимание, что для команды `vcmpsd` требуется четыре операнда: непосредственный операнд, определяющий предикат сравнения, два исходных операнда (первый исходный операнд должен быть регистром XMM, а второй исходный операнд может быть регистром XMM или значением в памяти) и операнд-приемник, который должен быть регистром XMM. Следующая команда `vmovq rax,xmm2` копирует младшее четверное слово XMM2 (которое содержит все нули или все единицы) для регистрации RAX. За ней следует команда `and al,1`, которая устанавливает регистр AL в 1, если предикат сравнения истинен; в противном случае AL устанавливается в 0. Последняя команда последовательности, `mov byte ptr [r8],al`, сохраняет результат сравнения в массив `results`. Затем функция `CompareVCMPSD_` использует аналогичные последовательности команд для демонстрации других часто используемых предикатов сравнения. Вот результаты выполнения примера `Ch05_05`:

```
Результаты CompareVCMPSD
----------------------------------------
a = 120   b = 130
cmp_eq:     false   cmp_neq:    true
cmp_lt:     true    cmp_le:     true
cmp_gt:     false   cmp_ge:     false
cmp_ord:    true    cmp_unord:  false

a = 250   b = 240
cmp_eq:     false   cmp_neq:    true
cmp_lt:     false   cmp_le:     false
cmp_gt:     true    cmp_ge:     true
cmp_ord:    true    cmp_unord:  false

a = 300   b = 300
cmp_eq:     true    cmp_neq:    false
cmp_lt:     false   cmp_le:     true
cmp_gt:     false   cmp_ge:     true
cmp_ord:    true    cmp_unord:  false

a = -18   b = 32
cmp_eq:     false   cmp_neq:    true
```

```
cmp_lt:     true    cmp_le:     true
cmp_gt:     false   cmp_ge:     false
cmp_ord:    true    cmp_unord:  false

a = -81   b = -100
cmp_eq:     false   cmp_neq:    true
cmp_lt:     false   cmp_le:     false
cmp_gt:     true    cmp_ge:     true
cmp_ord:    true    cmp_unord:  false

a = 42    b = nan
cmp_eq:     false   cmp_neq:    true
cmp_lt:     false   cmp_le:     false
cmp_gt:     false   cmp_ge:     false
cmp_ord:    false   cmp_unord:  true
```

Многие ассемблеры x86, включая MASM, поддерживают псевдооперационные формы команды `vcmpsd` и ее аналога с одинарной точностью `vcmpss`. Псевдооперации – это смоделированные мнемонические операции с предикатом сравнения, встроенным в мнемонический текст. В функции `CompareVCMPSD_`, например, вместо команды `vcmpsd xmm2,xmm0,xmm1,CMP_EQ` могла быть использована псевдооперация `vcmpeqsd xmm2,xmm0,xmm1`. Лично я считаю, что стандартную мнемонику справочного руководства легче читать, поскольку предикат сравнения явно указан как операнд, а не скрывается внутри псевдооперации, особенно при использовании одного из более эзотерических предикатов сравнения.

В этом разделе вы узнали, как выполнять операции сравнения с помощью команд `vcomis[d|s]` и `vcmps[d|s]`. На этом этапе вам может быть интересно, какие команды сравнения следует использовать. Для базовых скалярных операций сравнения с плавающей запятой (например, равно, не равно, меньше, меньше или равно, больше, больше или равно) команды `vcomis[d|s]` немного проще в использовании, поскольку они напрямую устанавливают флаги состояния в RFLAGS. Чтобы воспользоваться преимуществами расширенных предикатов сравнения, поддерживаемых AVX, необходимо использовать команды `vcmps[d|s]`. Другой причиной использования команд `vcmps[d|s]` является сходство между этими командами и соответствующими командами `vcmpp[d|s]` для упакованных операндов с плавающей запятой. Вы узнаете, как использовать команды сравнения упакованных значений с плавающей запятой, в главе 6.

5.2.2. Преобразования чисел с плавающей запятой

Распространенной операцией во многих программах на C++ является приведение значения с плавающей запятой одинарной или двойной точности к целому числу или наоборот. К другим частым операциям относятся расширение значения с плавающей запятой от одинарной точности к двойной точности и сужение значения двойной точности до одинарной точности. AVX включает ряд команд, которые выполняют эти типы преобразований. В листинге 5.6 показан код примера, демонстрирующего применение некоторых команд преобразования AVX. Он также показывает, как установить флаг управления округлением регистра MXCSR, чтобы изменить режим округления с плавающей запятой AVX.

Листинг 5.6. Пример Ch05_06

```
//-----------------------------------------------
//               Ch05_06.cpp
//-----------------------------------------------
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cstdint>
#include <string>
#define _USE_MATH_DEFINES
#include <math.h>

using namespace std;

// Простое объединение для обмена данными
union Uval
{
    int32_t m_I32;
    int32_t m_I64;
    float m_F32;
    double m_F64;
};

// Порядок значений ниже должен совпадать с таблицей переходов,
// которая объявлена в файле .asm

enum CvtOp : unsigned int
{
    I32_F32,    // int32_t в float
    F32_I32,    // float в int32_t
    I32_F64,    // int32_t в double
    F64_I32,    // double в int32_t
    I64_F32,    // int64_t в float
    F32_I64,    // float в int64_t
    I64_F64,    // int64_t в double
    F64_I64,    // double в int64_t
    F32_F64,    // float в double
    F64_F32,    // double в float
};

// Перечисление для режима округления
enum RoundingMode : unsigned int
{
    Nearest, Down, Up, Truncate
};

const string c_RoundingModeStrings[] = {"ближайший", "вниз", "вверх", "усечение"};
const RoundingMode c_RoundingModeVals[] = {RoundingMode::Nearest, RoundingMode::Down,
RoundingMode::Up, RoundingMode::Truncate};
const size_t c_NumRoundingModes = sizeof(c_RoundingModeVals) / sizeof (RoundingMode);

extern "C" RoundingMode GetMxcsrRoundingMode_(void);
extern "C" void SetMxcsrRoundingMode_(RoundingMode rm);
extern "C" bool ConvertScalar_(Uval* a, Uval* b, CvtOp cvt_op);

int main()
```

```
{
    Uval src1, src2, src3, src4, src5;
    src1.m_F32 = (float)M_PI;
    src2.m_F32 = (float)-M_E;
    src3.m_F64 = M_SQRT2;
    src4.m_F64 = M_SQRT1_2;
    src5.m_F64 = 1.0 + DBL_EPSILON;
    for (size_t i = 0; i < c_NumRoundingModes; i++)
    {
        Uval des1, des2, des3, des4, des5;
        RoundingMode rm_save = GetMxcsrRoundingMode_();
        RoundingMode rm_test = c_RoundingModeVals[i];

        SetMxcsrRoundingMode_(rm_test);

        ConvertScalar_(&des1, &src1, CvtOp::F32_I32);
        ConvertScalar_(&des2, &src2, CvtOp::F32_I64);
        ConvertScalar_(&des3, &src3, CvtOp::F64_I32);
        ConvertScalar_(&des4, &src4, CvtOp::F64_I64);
        ConvertScalar_(&des5, &src5, CvtOp::F64_F32);

        SetMxcsrRoundingMode_(rm_save);

        cout << fixed;
        cout << "\nРежим округления = " << c_RoundingModeStrings[rm_test] << '\n';

        cout << " F32_I32: " << setprecision(8);
        cout << src1.m_F32 << " --> " << des1.m_I32 << '\n';

        cout << " F32_I64: " << setprecision(8);
        cout << src2.m_F32 << " --> " << des2.m_I64 << '\n'
        ;
        cout << " F64_I32: " << setprecision(8);
        cout << src3.m_F64 << " --> " << des3.m_I32 << '\n';

        cout << " F64_I64: " << setprecision(8);
        cout << src4.m_F64 << " --> " << des4.m_I64 << '\n';

        cout << " F64_F32: ";
        cout << setprecision(16) << src5.m_F64 << " --> ";
        cout << setprecision(8) << des5.m_F32 << '\n';
    }

    return 0;

;-----------------------------------------------
;                Ch05_06.asm
;-----------------------------------------------

MxcsrRcMask equ 9fffh                   ;битовый шаблон для MXCSR.RC
MxcsrRcShift equ 13                     ;счетчик сдвига для MXCSR.RC

; extern "C" RoundingMode GetMxcsrRoundingMode_(void);
;
; Описание: эта функция извлекает текущий режим округления
;           с плавающей запятой из регистра MXCSR.RC
;
```

```
; Возвращает: текущий режим округления, указанный в MXCSR.RC

        .code
GetMxcsrRoundingMode_ proc
        vstmxcsr dword ptr [rsp+8]      ;сохранение регистра mxcsr
        mov eax,[rsp+8]
        shr eax,MxcsrRcShift            ;eax[1:0] = MXCSR.RC
        and eax,3                       ;маскировка ненужных битов
        ret
GetMxcsrRoundingMode_ endp

;extern "C" void SetMxcsrRoundingMode_(RoundingMode rm);
;
; Описание:  данная функция обновляет значение режима округления
;            в регистре MXCSR.RC

SetMxcsrRoundingMode_ proc
        and ecx,3                       ;маскировка ненужных битов
        shl ecx,MxcsrRcShift            ;ecx[14:13] = rm
        vstmxcsr dword ptr [rsp+8]      ;сохранение текущего MXCSR
        mov eax,[rsp+8]
        and eax,MxcsrRcMask             ;маскировка старых битов MXCSR.RC
        or eax,ecx                      ;вставка новых битов MXCSR.RC
        mov [rsp+8],eax
        vldmxcsr dword ptr [rsp+8]      ;загрузка обновленного MXCSR
        ret
SetMxcsrRoundingMode_ endp

; extern "C" bool ConvertScalar_(Uval* des, const Uval* src, CvtOp cvt_op)
;
; Примечание: для этой функции опция компоновщика /LARGEADDRESSAWARE:NO
;             должна быть установлена в явном виде

ConvertScalar_ proc

; Проверка cvt_op на правильность и переход к нужному блоку кода
        mov eax,r8d                     ;eax = CvtOp
        cmp eax,CvtOpTableCount
        jae BadCvtOp                    ;переход, если cvt_op неправильный
        jmp [CvtOpTable+rax*8]          ;переход к нужному блоку преобразования

; Преобразование между типами int32_t и float/double

I32_F32:
        mov eax,[rdx]                   ;загрузка целочисленного значения
        vcvtsi2ss xmm0,xmm0,eax         ;преобразование в число с плавающей запятой
        vmovss real4 ptr [rcx],xmm0     ;сохранение результата
        mov eax,1
        ret

F32_I32:
        vmovss xmm0,real4 ptr [rdx]     ;загрузка числа с плавающей запятой
        vcvtss2si eax,xmm0              ;преобразование в целое число
        mov [rcx],eax                   ;сохранение результата
        mov eax,1
        ret
```

```
I32_F64:
        mov eax,[rdx]                   ;загрузка целого числа
        vcvtsi2sd xmm0,xmm0,eax         ;преобразование в двойное слово
        vmovsd real8 ptr [rcx],xmm0     ;сохранение результата

        mov eax,1
ret

F64_I32:
        vmovsd xmm0,real8 ptr [rdx]     ;загрузка двойного слова
        vcvtsd2si eax,xmm0              ;преобразование в целое
        mov [rcx],eax                   ;сохранение результата

        mov eax,1
        ret

; Преобразование между int64_t и float/double

I64_F32:
        mov rax,[rdx]                   ;загрузка целого числа
        vcvtsi2ss xmm0,xmm0,rax         ;преобразование в float
        vmovss real4 ptr [rcx],xmm0     ;сохранение результата

        mov eax,1
        ret

F32_I64:
        vmovss xmm0,real4 ptr [rdx]     ;загрузка значения типа float
        vcvtss2si rax,xmm0              ;преобразование в целое число
        mov [rcx],rax                   ;сохранение результата
        mov eax,1
        ret

I64_F64:
        mov rax,[rdx]                   ;загрузка целого числа
        vcvtsi2sd xmm0,xmm0,rax         ;преобразование в двойное слово
        vmovsd real8 ptr [rcx],xmm0     ;сохранение результата

        mov eax,1
        ret

F64_I64:
        vmovsd xmm0,real8 ptr [rdx]     ;загрузка двойного слова
        vcvtsd2si rax,xmm0              ;преобразование в целое
        mov [rcx],rax                   ;сохранение результата

        mov eax,1
        ret

; Преобразование между float и double

F32_F64:
        vmovss xmm0,real4 ptr [rdx]     ;загрузка значения типа float
        vcvtss2sd xmm1,xmm1,xmm0        ;преобразование в двойное слово
        vmovsd real8 ptr [rcx],xmm1     ;сохранение результата
```

```
        mov eax,1
        ret

F64_F32:
vmovsd xmm0,real8 ptr [rdx]             ;загрузка двойного слова
vcvtsd2ss xmm1,xmm1,xmm0                ;преобразование в float
vmovss real4 ptr [rcx],xmm1             ;сохранение результата

mov eax,1
ret

BadCvtOp:
        xor eax,eax                     ;формирование кода ошибки
        ret

; Порядок значений в таблице ниже должен совпадать с перечислением CvtOp,
; определенном в файле .cpp

        align 8
CvtOpTable equ $
        qword I32_F32, F32_I32
        qword I32_F64, F64_I32
        qword I64_F32, F32_I64
        qword I64_F64, F64_I64
        qword F32_F64, F64_F32
CvtOpTableCount equ ($ - CvtOpTable) / size qword

ConvertScalar_ endp
        end
```

В верхней части кода C++ находится объявление `union Uval`, которое используется для обмена данными. За ним следуют два перечисления: одно для выбора типа преобразования с плавающей запятой (`CvtOp`) и другое для указания режима округления с плавающей запятой (`RoundingMode`). Функция `main` в C++ инициализирует пару экземпляров `Uval` в качестве тестовых примеров и вызывает функцию языка ассемблера `ConvertScalar_` для выполнения преобразований с использованием различных режимов округления. Затем результат каждой операции преобразования выводится на печать для проверки и сравнения.

Режим округления с плавающей запятой AVX определяется полем управления округлением (биты 14 и 13) регистра MXCSR, как описано в главе 4. Режим округления по умолчанию для программ Visual C++ – округление до ближайшего. Согласно соглашению о вызовах Visual C ++, значения в MXCSR [15:6] (т. е. биты регистров MXCSR с 15 по 6) должны сохраняться при переходе через границы большинства функций. Код в `main` выполняет это требование, вызывая функцию `GetMxcsrRoundingMode_`, чтобы сохранить текущий режим округления перед выполнением любых операций преобразования с использованием `ConvertScalar_`. Исходный режим округления в конечном итоге восстанавливается с помощью функции `SetMxcsrRoundingMode_`. Обратите внимание, что исходный режим округления восстанавливается до операторов `cout` в `main`. Также обратите внимание, что я несколько упростил код сохранения и восстановления в режиме округления, не сохраняя режим округления перед каждым использованием `ConvertScalar_` и восстанавливая его сразу после этого.

В листинге 5.6 также представлены функции управления режимом округления. Функция GetMxcsrRoundingMode_ использует команду vstmxcsr dword ptr [rsp+8] (сохранение состояния MXCSR) для сохранения содержимого MXCSR в домашнюю область RCX в стеке. Напомним, что функция может использовать свою домашнюю область в стеке для любых целей временного хранения. Единственный операнд команды vstmxcsr должен быть двойным словом в памяти; это не может быть регистр общего назначения. Следующая команда mov eax,[rsp+8] копирует текущее значение MXCSR в регистр EAX. За этим следует сдвиг и побитовая операция И, извлекающая биты управления округлением. Соответствующая функция SetMxcsrRoundingMode_ использует команду vldmxcsr (загрузить регистр MXCSR) для установки режима округления. Инструкция vldmxcsr тоже требует, чтобы ее единственный операнд был двойным словом в памяти. Обратите внимание, что функция SetMxcsrRoundingMode_ также использует команду vstmxcsr и операции маскирования, чтобы гарантировать, что при установке нового режима округления изменяются только биты управления округлением MXCSR.

Функция ConvertScalar_ выполняет преобразования с плавающей запятой, используя указанные числовые аргументы и оператор преобразования. После проверки аргумента cvt_op команда jmp [CvtOpTable+rax*8] передает управление соответствующему разделу кода, который выполняет фактическое преобразование. Обратите внимание, что эта команда использует таблицу переходов. Здесь регистр RAX (который содержит cvt_op) указывает индекс в таблице CvtOpTable. Таблица CvtOpTable определяется сразу после команды ret и содержит смещения для различных блоков кода преобразования. Вы узнаете больше о таблицах переходов в главе 6.

Также важно отметить, что одна и та же мнемоника команд иногда используется при преобразовании целого числа в число с плавающей запятой и наоборот. Например, команда vcvtsi2ss xmm0,xmm0,eax (расположенная рядом с меткой I32_F32) преобразует 32-битное целое число со знаком в число с плавающей запятой одинарной точности, а команда vcvtsi2ss xmm0, xmm0,rax (расположенная рядом с меткой I64_F32) преобразует 64-битное целое число со знаком для преобразования с плавающей запятой одинарной точности.

Преобразования между двумя разными числовыми типами данных не всегда возможны. Например, команда vcvtss2si не может преобразовать большие значения с плавающей запятой в 32-разрядные целые числа со знаком. Если конкретное преобразование невозможно и исключения недопустимой операции (MXCSR.IM) замаскированы (по умолчанию для Visual C++), процессор устанавливает MXCSR.IE (флаг ошибки недопустимой операции), и значение 0x80000000 копируется в операнд назначения. Например, вывод примера Ch05_06 будет следующим:

```
Режим округления = ближайшее
    F32_I32: 3.14159274 --> 3
    F32_I64: -2.71828175 --> -3
    F64_I32: 1.41421356 --> 1
    F64_I64: 0.70710678 --> 1
    F64_F32: 1.0000000000000002 --> 1.00000000
```

```
Режим округления = вниз
    F32_I32: 3.14159274 --> 3
    F32_I64: -2.71828175 --> -3
    F64_I32: 1.41421356 --> 1
    F64_I64: 0.70710678 --> 0
    F64_F32: 1.0000000000000002 --> 1.00000000

Режим округления = вверх
    F32_I32: 3.14159274 --> 4
    F32_I64: -2.71828175 --> -2
    F64_I32: 1.41421356 --> 2
    F64_I64: 0.70710678 --> 1
    F64_F32: 1.0000000000000002 --> 1.00000012

Режим округления = усечение
    F32_I32: 3.14159274 --> 3
    F32_I64: -2.71828175 --> -2
    F64_I32: 1.41421356 --> 1
    F64_I64: 0.70710678 --> 0
    F64_F32: 1.0000000000000002 --> 1.00000000
```

5.3. Скалярные массивы и матрицы с плавающей запятой

В главе 3 вы узнали, как получить доступ к отдельным элементам и выполнять вычисления с использованием целочисленных массивов и матриц. В этом разделе вы узнаете, как выполнять аналогичные операции, используя массивы и матрицы значений с плавающей запятой. Как скоро вы увидите, одни и те же методы кодирования на языке ассемблера часто используются как для целочисленных массивов, так и для массивов и матриц с плавающей запятой.

5.3.1. Массивы значений с плавающей запятой

В листинге 5.7 приведен код примера `Ch05_07`. В этом примере показано, как вычислить среднее значение и среднеквадратическое отклонение выборки для массива значений с плавающей запятой двойной точности.

Листинг 5.7. Пример Ch05_07

```
//-----------------------------------------------
//                Ch05_07.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

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

bool CalcMeanStdevCpp(double* mean, double* stdev, const double* x, int n)
{
```

```
    if (n < 2)
        return false;

    double sum = 0.0;
    for (int i = 0; i < n; i++)
        sum += x[i];

    *mean = sum / n;

    double sum2 = 0.0;
    for (int i = 0; i < n; i++)
    {
        double temp = x[i] - *mean;
        sum2 += temp * temp;
    }

    *stdev = sqrt(sum2 / (n - 1));
    return true;
}

int main()
{
    double x[] = { 10, 2, 33, 19, 41, 24, 75, 37, 18, 97, 14, 71, 88, 92, 7};
    const int n = sizeof(x) / sizeof(double);

    double mean1 = 0.0, stdev1 = 0.0;
    double mean2 = 0.0, stdev2 = 0.0;

    bool rc1 = CalcMeanStdevCpp(&mean1, &stdev1, x, n);
    bool rc2 = CalcMeanStdev_(&mean2, &stdev2, x, n);

    cout << fixed << setprecision(2);

    for (int i = 0; i < n; i++)
    {
        cout << "x[" << setw(2) << i << "] = ";
        cout << setw(6) << x[i] << '\n';
    }

    cout << setprecision(6);

    cout << '\n';
    cout << "rc1 = " << boolalpha << rc1;
    cout << " mean1 = " << mean1 << " stdev1 = " << stdev1 << '\n';
    cout << "rc2 = " << boolalpha << rc2;
    cout << " mean2 = " << mean2 << " stdev2 = " << stdev2 << '\n';
}

;-----------------------------------------------
;               Ch05_07.asm
;-----------------------------------------------

; extern "C" bool CalcMeanStdev(double* mean, double* stdev, const double* a, int n);
;
; Возвращает: 0 = недопустимое n, 1 = корректное n
```

```
        .code
CalcMeanStdev_ proc

; Проверка допустимости значения n
        xor eax,eax                     ;генерация кода ошибки (также i = 0)
        cmp r9d,2
        jl InvalidArg                   ;переход, если n < 2

; Compute sample mean
        vxorpd xmm0,xmm0,xmm0           ;sum = 0.0

@@:     vaddsd xmm0,xmm0,real8 ptr [r8+rax*8] ;sum += x[i]
        inc eax                         ;i += 1
        cmp eax,r9d
        jl @B                           ;переход, если i < n

        vcvtsi2sd xmm1,xmm1,r9d         ;преобразование n в DPFP
        vdivsd xmm3,xmm0,xmm1           ;xmm3 = mean (sum / n)
        vmovsd real8 ptr [rcx],xmm3     ;сохранение среднего значения

; Вычисление среднеквадратического отклонения (СКО)
        xor eax,eax                     ;i = 0
        vxorpd xmm0,xmm0,xmm0           ;sum2 = 0.0

@@:     vmovsd xmm1,real8 ptr [r8+rax*8] ;xmm1 = x[i]
        vsubsd xmm2,xmm1,xmm3           ;xmm2 = x[i] - mean
        vmulsd xmm2,xmm2,xmm2           ;xmm2 = (x[i] - mean) ** 2
        vaddsd xmm0,xmm0,xmm2           ;sum2 += (x[i] - mean) ** 2
        inc eax                         ;i += 1
        cmp eax,r9d
        jl @B                           ;переход, если i < n

        dec r9d ;r9d = n - 1
        vcvtsi2sd xmm1,xmm1,r9d         ;преобразование n - 1 в DPFP
        vdivsd xmm0,xmm0,xmm1           ;xmm0 = sum2 / (n - 1)
        vsqrtsd xmm0,xmm0,xmm0          ;xmm0 = СКО
        vmovsd real8 ptr [rdx],xmm0     ;сохранение СКО

        mov eax,1                       ;код успешного завершения

InvalidArg:
        ret
CalcMeanStdev_ endp
        end
```

Вот формулы, которые в примере Ch05_07 используются для вычисления среднего значения и стандартного отклонения выборки:

$$\overline{x} = \frac{1}{n}\sum_i x_i;$$

$$s = \sqrt{\frac{1}{n-1}\sum_i \left(x_i - \overline{x}\right)^2}.$$

Код C++ в примере Ch05_07 весьма прост. Он содержит функцию CalcMeanStdevCpp, которая вычисляет среднее значение и среднеквадратическое отклонение выборки для массива значений с плавающей запятой двойной точности. Обратите внимание, что эта функция и ее эквивалент на языке ассемблера возвращают вычисленное среднее значение и стандартное отклонение с помощью указателей. Оставшийся код C++ инициализирует тестовый массив и выполняет обе вычислительные функции.

При входе в функцию языка ассемблера CalcMeanStdev_ количество элементов массива n проверяется на допустимость. Обратите внимание, что для расчета среднеквадратического отклонения количество элементов массива должно быть больше одного. После проверки n команда vxorpd,xmm0,xmm0,xmm0 (побитовое исключающее ИЛИ упакованных значений с плавающей запятой двойной точности) инициализирует сумму значением 0,0. Эта команда выполняет побитовую операцию XOR, используя все 128 бит обоих исходных операндов. Здесь используется команда vxorpd для инициализации суммы нулевым значением, поскольку AVX не содержит явную команду XOR для скалярных операндов с плавающей запятой.

Блок кода, который вычисляет среднее значение, состоит всего из семи команд. Первая команда цикла суммирования, vaddsd xmm0,xmm0 real8 ptr [r8+rax*8], добавляет x[i] к сумме. Следующая команда inc eax обновляет i, и цикл суммирования повторяется, пока i не достигнет n. После цикла суммирования команда vcvtsi2sd xmm1,xmm1,r9d переводит копию n в число с плавающей запятой двойной точности, а последующая команда vdivsd xmm3,xmm0,xmm1 вычисляет окончательное среднее значение выборки. Затем среднее значение сохраняется в ячейку памяти, на которую указывает RCX.

Вычисление среднего отклонения выборки начинается с двух команд, xor eax,eax и vxorpd xmm0,xmm0,xmm0, которые инициализируют i значением 0 и sum2 значением 0,0. Следующие команды vsubsd, vmulsd и vaddsd вычисляют sum2 + = (x[i]-mean)**2, и цикл суммирования повторяется до тех пор, пока все элементы массива не будут обработаны. Выполнение команды dec r9d дает значение n-1. Затем это значение переводится в число с плавающей запятой двойной точности с помощью команды vcvtsi2sd xmm1,xmm1,r9d. Последние две арифметические команды, vdivsd xmm0,xmm0,xmm1 и vsqrtsd xmm0,xmm0,xmm0, вычисляют стандартное отклонение выборки, и это значение сохраняется в ячейке памяти, на которую указывает RDX. Вот результат выполнения примера Ch05_07:

```
x[ 0] = 10.00
x[ 1] = 2.00
x[ 2] = 33.00
x[ 3] = 19.00
x[ 4] = 41.00
x[ 5] = 24.00
x[ 6] = 75.00
x[ 7] = 37.00
x[ 8] = 18.00
x[ 9] = 97.00
x[10] = 14.00
x[11] = 71.00
```

```
x[12] = 88.00
x[13] = 92.00
x[14] = 7.00

rc1 = true mean1 = 41.866667 stdev1 = 33.530086
rc2 = true mean2 = 41.866667 stdev2 = 33.530086
```

5.3.2. Матрицы значений с плавающей запятой

В главе 3 представлен пример программы (см. листинг Ch03_03), которая выполняет вычисления с использованием элементов целочисленной матрицы. В этом разделе вы узнаете, как выполнять аналогичные вычисления, используя элементы матрицы с плавающей запятой одинарной точности. В листинге 5.8 показан исходный код примера Ch05_08.

Листинг 5.8. Пример Ch05_08

```
//-----------------------------------------------
// Ch05_08.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" void CalcMatrixSquaresF32_(float* y, const float* x, float offset, int nrows,
int
ncols);

void CalcMatrixSquaresF32Cpp(float* y, const float* x, float offset, int nrows, int ncols)
{
    for (int i = 0; i < nrows; i++)
    {
        for (int j = 0; j < ncols; j++)
        {
            int kx = j * ncols + i;
            int ky = i * ncols + j;
            y[ky] = x[kx] * x[kx] + offset;
        }
    }
}

int main()
{
    const int nrows = 6;
    const int ncols = 3;
    const float offset = 0.5;
    float y2[nrows][ncols];
    float y1[nrows][ncols];
    float x[nrows][ncols] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 },
                            { 10, 11, 12 }, {13, 14, 15}, {16, 17, 18} };

    CalcMatrixSquaresF32Cpp(&y1[0][0], &x[0][0], offset, nrows, ncols);
```

```
    CalcMatrixSquaresF32_(&y2[0][0], &x[0][0], offset, nrows, ncols);

    cout << fixed << setprecision(2);

    cout << "offset = " << setw(2) << offset << '\n';

    for (int i = 0; i < nrows; i++)
    {
        for (int j = 0; j < ncols; j++)
        {
            cout << "y1[" << setw(2) << i << "][" << setw(2) << j << "] = ";
            cout << setw(6) << y1[i][j] << " " ;
            cout << "y2[" << setw(2) << i << "][" << setw(2) << j << "] = ";
            cout << setw(6) << y2[i][j] << " ";
            cout << "x[" << setw(2) << j << "][" << setw(2) << i << "] = ";
            cout << setw(6) << x[j][i] << '\n';
            if (y1[i][j] != y2[i][j])
            cout << "Результаты не совпадают\n";
        }
    }

return 0;
}

;-----------------------------------------------
; Ch05_08.asm
;-----------------------------------------------

; void CalcMatrixSquaresF32_(float* y, const float* x, float offset, int nrows, int ncols);
;
; Вычисляет: y[i][j] = x[j][i] * x[j][i] + offset

        .code
CalcMatrixSquaresF32_ proc frame

;   Пролог функции
        push rsi                        ;сохранение значения rsi
        .pushreg rsi
        push rdi                        ;сохранение значения rdi
        .pushreg rdi
        .endprolog

; Проверка значений nrows и ncols
        movsxd r9,r9d                   ;r9 = nrows
        test r9,r9
        jle InvalidCount                ;переход, если nrows <= 0

        movsxd r10,dword ptr [rsp+56]   ;r10 = ncols
        test r10,r10
        jle InvalidCount                ;переход, если ncols <= 0

; Инициализация указателей на массивы источника и приемника
        mov rsi,rdx                     ;rsi = x
        mov rdi,rcx                     ;rdi = y
        xor rcx,rcx                     ;rcx = i
```

```
; Необходимые вычисления
Loop1:  xor rdx,rdx                     ;rdx = j

Loop2:  mov rax,rdx                     ;rax = j
        imul rax,r10                    ;rax = j * ncols
        add rax,rcx                         ;rax = j * ncols + i
        vmovss xmm0,real4 ptr [rsi+rax*4]   ;xmm0 = x[j][i]
        vmulss xmm1,xmm0,xmm0               ;xmm1 = x[j][i] * x[j][i]
        vaddss xmm3,xmm1,xmm2               ;xmm2 = x[j][i] * x[j][i] + смещение

        mov rax,rcx                         ;rax = i
        imul rax,r10                        ;rax = i * ncols
        add rax,rdx                         ;rax = i * ncols + j;
        vmovss real4 ptr [rdi+rax*4],xmm3   ;y[i][j] = x[j][i] * x[j][i] + смещение

        inc rdx                         ;j += 1
        cmp rdx,r10
        jl Loop2                        ;переход, если j < ncols

        inc rcx                         ;i += 1
        cmp rcx,r9
        jl Loop1                        ;переход, если i < nrows

InvalidCount:

; Эпилог функции
        pop rdi                         ;восстановление значения rdi
        pop rsi                         ;восстановление значения rsi
        ret
CalcMatrixSquaresF32_ endp
        end
```

Исходный код C++, показанный в листинге 5.8, аналогичен тому, что вы видели в главе 3. Методы, используемые для вычисления смещений элементов матрицы, идентичны. Самым большим изменением, внесенным в код C++, было изменение соответствующих объявлений типа матрицы с `int` на `float`. Еще одно различие между этим примером и тем, что вы видели в главе 3, – это добавление смещения аргумента к объявлениям `CalcMatrixSquaresF32Cpp` и `CalcMatrixSquaresF32_`. Обе эти функции теперь вычисляют `y[i][j] = x[j][i]*x[j][i]+смещение`.

На рис. 5.2 показаны структура стека и регистры аргументов сразу после выполнения команды `push rdi` в функции `CalcMatrixSquaresF32_`. На этом рисунке показана передача аргументов функции, которая использует сочетание целочисленных аргументов (или указателей) и аргументов с плавающей запятой. В соответствии с соглашением о вызовах Visual C++ первые четыре аргумента передаются с использованием регистра общего назначения или XMM в зависимости от типа и положения аргумента. В частности, значение первого аргумента передается с использованием регистра RCX или XMM0. Второй, третий и четвертый аргументы передаются с использованием RDX/XMM1, R8/XMM2 или R9/XMM3. Все оставшиеся аргументы передаются в стек.

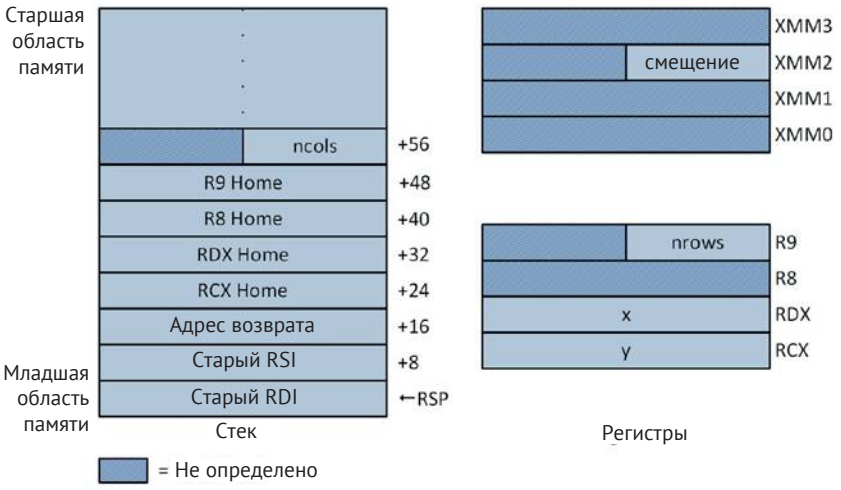


Рис. 5.2. Структура стека и регистры аргументов после выполнения команды `push rdi` в функции `CalcMatrixSquaresF32_`

Код на языке ассемблера для функции `CalcMatrixSquaresF32_` аналогичен тому, что вы изучали в главе 3. Как и код C++, методы, используемые для вычисления смещения элементов матрицы, такие же. В исходном коде вычисления элементов матрицы использовалась целочисленная арифметика, и эти команды были заменены аналогичными скалярными командами AVX с плавающей запятой одиночной точности. После вычисления правильного смещения матричного элемента команда `vmovss xmm0,real4 ptr [rsi+rax*4]` загружает регистр XMM0 с матричным элементом `x[j][i]`. Следующие команды `vmulss xmm1,xmm0,xmm0` и `vaddss xmm3,xmm1,xmm2` вычисляют требуемый результат, а команда `vmovss real4 ptr [rdi+rax*4],xmm3` сохраняет результат в `y[i][j]`. Вот результат выполнения примера `Ch05_08`:

```
offset = 0.50
y1[ 0][ 0] =   1.50   y2[ 0][ 0] =   1.50 x[ 0][ 0] =  1.00
y1[ 0][ 1] =  16.50   y2[ 0][ 1] =  16.50 x[ 1][ 0] =  4.00
y1[ 0][ 2] =  49.50   y2[ 0][ 2] =  49.50 x[ 2][ 0] =  7.00
y1[ 1][ 0] =   4.50   y2[ 1][ 0] =   4.50 x[ 0][ 1] =  2.00
y1[ 1][ 1] =  25.50   y2[ 1][ 1] =  25.50 x[ 1][ 1] =  5.00
y1[ 1][ 2] =  64.50   y2[ 1][ 2] =  64.50 x[ 2][ 1] =  8.00
y1[ 2][ 0] =   9.50   y2[ 2][ 0] =   9.50 x[ 0][ 2] =  3.00
y1[ 2][ 1] =  36.50   y2[ 2][ 1] =  36.50 x[ 1][ 2] =  6.00
y1[ 2][ 2] =  81.50   y2[ 2][ 2] =  81.50 x[ 2][ 2] =  9.00
y1[ 3][ 0] =  16.50   y2[ 3][ 0] =  16.50 x[ 0][ 3] =  4.00
y1[ 3][ 1] =  49.50   y2[ 3][ 1] =  49.50 x[ 1][ 3] =  7.00
y1[ 3][ 2] = 100.50   y2[ 3][ 2] = 100.50 x[ 2][ 3] = 10.00
y1[ 4][ 0] =  25.50   y2[ 4][ 0] =  25.50 x[ 0][ 4] =  5.00
y1[ 4][ 1] =  64.50   y2[ 4][ 1] =  64.50 x[ 1][ 4] =  8.00
y1[ 4][ 2] = 121.50   y2[ 4][ 2] = 121.50 x[ 2][ 4] = 11.00
y1[ 5][ 0] =  36.50   y2[ 5][ 0] =  36.50 x[ 0][ 5] =  6.00
y1[ 5][ 1] =  81.50   y2[ 5][ 1] =  81.50 x[ 1][ 5] =  9.00
y1[ 5][ 2] = 144.50   y2[ 5][ 2] = 144.50 x[ 2][ 5] = 12.00
```

Исходя из примеров исходного кода в этом разделе, должно быть очевидно, что при работе с массивами или матрицами для ссылки на конкретные элементы могут использоваться методы, независимые от фактического типа данных. Конструкции цикла также можно кодировать с использованием методов, которые отделены от фактического типа данных.

5.4. Соглашение о вызовах

В примерах исходного кода, представленных ранее в этой книге, неформально обсуждались различные аспекты соглашения о вызовах Visual C++. В этом разделе приведено формальное изложение соглашения о вызовах. Оно повторяет некоторые предыдущие разъяснения, а также вводит новые требования и параметры, которые еще не обсуждались. Базовое понимание соглашения о вызовах необходимо, поскольку оно широко используется в примерах кода в последующих главах. Напоминаю, что если вы читаете эту книгу, чтобы изучить программирование на языке ассемблера x86-64, и планируете использовать его с другой операционной системой или языком высокого уровня, вам следует обратиться к соответствующей документации для получения информации о деталях этого соглашения о вызовах.

Соглашение о вызовах Visual C++ определяет каждый регистр общего назначения процессора x86-64 как *энергозависимый* или *энергонезависимый*. Соглашение также применяет аналогичную классификацию к каждому регистру XMM. Функция на языке ассемблера x86-64 может изменять содержимое любого энергозависимого регистра, но *должна* сохранять содержимое любого энергонезависимого регистра, который она использует. В табл. 5.3 перечислены энергозависимые и энергонезависимые регистры общего назначения и регистры XMM.

В системах, поддерживающих AVX или AVX2, старшие 128 бит каждого регистра YMM классифицируются как энергозависимые. Точно так же старшие 384 бита регистров ZMM0-ZMM15 классифицируются как энергозависимые в системах, поддерживающих AVX-512. Регистры ZMM16-ZMM31 и соответствующие регистры YMM и XMM тоже обозначены как энергозависимые и не нуждаются в сохранении. 64-битные программы Visual C++ обычно не используют FPU x87. Функции языка ассемблера, использующие этот ресурс, не требуются для сохранения содержимого стека регистров FPU x87, что означает, что весь стек регистров классифицируется как энергозависимый.

Таблица 5.3. 64-разрядные энергозависимые и энергонезависимые регистры Visual C++

Группа регистров	Энергозависимые	Энергонезависимые
Универсальные	RAX, RCX, RDX, R8, R9, R10, R11	RBX, RSI, RDI, RBP, RSP, R12, R13, R14, R15
XMM	XMM0 – XMM5	XMM6 – XMM15

Требования к программированию, налагаемые на функцию языка ассемблера x86-64 соглашением о вызовах Visual C++, различаются в зависимости от того, является функция листовой или нет. *Листовые функции* (leaf function) – это функции, которые:

- не вызывают никаких других функций (конечный лист на дереве алгоритма);
- не изменяют содержимое регистра RSP;
- не выделяют пространство локального стека;
- не изменяют энергонезависимые регистры общего назначения или регистры XMM;
- не используют обработку исключений.

Листовые функции на 64-битном языке ассемблера проще кодировать, но они подходят только для относительно простых вычислительных задач. Нелистовая функция, в свою очередь, может использовать весь набор регистров x86-64, создавать фрейм стека, выделять локальное пространство стека или вызывать другие функции, при условии что она соответствует точным требованиям соглашения о вызовах для прологов и эпилогов. Примеры кода в этом разделе иллюстрируют перечисленные требования.

В оставшейся части этого раздела вы изучите четыре примера исходного кода. Первые три примера показывают, как программировать нелистовые функции с помощью явных команд и директив ассемблера. Эти программы также передают важную программную информацию, касающуюся организации фрейма стека нелистовой функции. Четвертый пример демонстрирует, как использовать несколько макросов пролога и эпилога. Эти макросы помогают автоматизировать большую часть работы по программированию, связанной с нелистовыми функциями.

5.4.1. Базовые фреймы стека

В листинге 5.9 показан исходный код примера `Ch05_09`. Эта программа демонстрирует, как инициализировать указатель фрейма стека в функции языка ассемблера. Указатели фреймов стека используются для ссылки на значения аргументов и локальные переменные в стеке. Пример `Ch05_09` также иллюстрирует некоторые протоколы программирования, которые должны соблюдаться в прологе и эпилоге функций на языке ассемблера.

Листинг 5.9. Пример Ch05_09

```
//-----------------------------------------------
//              Ch05_09.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <cstdint>

using namespace std;

extern "C" int64_t Cc1_(int8_t a, int16_t b, int32_t c, int64_t d, int8_t e, int16_t f,
int32_t g, int64_t h);

int main()
{
    int8_t a = 10, e = -20;
    int16_t b = -200, f = 400;
```

```
    int32_t c = 300, g = -600;
    int64_t d = 4000, h = -8000;

    int64_t sum = Cc1_(a, b, c, d, e, f, g, h);

    const char nl = '\n';

    cout << "Результаты Cc1\n\n";
    cout << "a = " << (int)a << nl;
    cout << "b = " << b << nl;
    cout << "c = " << c << nl;
    cout << "d = " << d << nl;
    cout << "e = " << (int)e << nl;
    cout << "f = " << f << nl;
    cout << "g = " << g << nl;
    cout << "h = " << h << nl;
    cout << "sum = " << sum << nl;

    return 0;
}

;-----------------------------------------------
;                Ch05_09.asm
;-----------------------------------------------

; extern "C" Int64 Cc1_(int8_t a, int16_t b, int32_t c, int64_t d, int8_t e, int16_t f,
int32_t g, int64_t h);

        .code
Cc1_ proc frame

; Пролог функции -
        push rbp                        ;сохранение регистра rbp
        .pushreg rbp

        sub rsp,16                      ;выделение локального пространства стека
        .allocstack 16

        mov rbp,rsp                     ;настройка указателя на фрейм
        .setframe rbp,0

RBP_RA = 24                             ;смещение rbp к адресу возврата
        .endprolog                      ;обозначение конца пролога

; Сохранение регистров аргументов в домашнем пространстве (опционально)
        mov [rbp+RBP_RA+8],rcx
        mov [rbp+RBP_RA+16],rdx
        mov [rbp+RBP_RA+24],r8
        mov [rbp+RBP_RA+32],r9

; Сумма значений аргументов a, b, c, и d
        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                    ;сохранение a + b + c + d

; Сумма значений аргументов e, f, g, и h
        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
        add r8,rcx                          ;r8 = e + f + g + h

; Финальная сумма
        mov rax,[rbp]                       ;rax = a + b + c + d
        add rax,r8                          ;rax = финальная сумма

; Эпилог функции
        add rsp,16                  ;освобождение локального пространства стека
        pop rbp                     ;восстановление регистра rbp
        ret

Cc1_ endp
        end
```

Цель кода C++ в листинге 5.9 – инициализировать тестовый пример для функции языка ассемблера `Cc1_`. Эта функция вычисляет и возвращает сумму восьми значений аргумента целого числа со знаком. Затем результаты отображаются с использованием последовательного потока, записываемого в `cout`.

В коде на языке ассемблера оператор `Cc1_ proc frame` отмечает начало функции `Cc1_`. Атрибут `frame` уведомляет ассемблер о том, что функция `Cc1_` использует указатель фрейма стека. Он также указывает ассемблеру сгенерировать данные статической таблицы, которые среда выполнения Visual C++ использует для обработки исключений. Последующая команда `push rbp` сохраняет регистр RBP вызывающей функции в стеке, поскольку функция `Cc1_` использует этот регистр в качестве указателя фрейма стека. Следующий оператор `.pushreg rbp` является директивой ассемблера, сохраняющей информацию о смещении команды `push rbp` в таблицах обработки исключений. Имейте в виду, что директивы ассемблера не являются исполняемыми командами процессора; это указания ассемблеру о том, как выполнять определенные действия во время сборки исходного кода.

Команда `sub rsp,16` выделяет 16 байт пространства стека для локальных переменных. Функция `Cc1_` использует только 8 байт этого пространства, но соглашение о вызовах Visual C++ требует, чтобы нелистовые функции поддерживали 16-байтовое выравнивание указателя стека вне пролога. Подробнее о требованиях к выравниванию указателя стека вы узнаете позже в этом разделе. Следующий оператор, `.allocstack 16`, является директивой ассемблера, которая сохраняет информацию о распределении размера локального стека в таблицах обработки исключений.

Инструкция `mov rbp,rsp` инициализирует регистр RBP как указатель кадра стека, а директива `.setframe rbp,0` уведомляет ассемблер об этом действии. Значение смещения 0, включенное в директиву `.setframe`, представляет собой разницу в байтах между RSP и RBP. В функции `Cc1_` регистры RSP и RBP одинако-

вы, поэтому значение смещения равно нулю. Позже в этом разделе вы узнаете больше о директиве `.setframe`. Следует отметить, что функции языка ассемблера могут использовать в качестве указателя кадра стека любой энергонезависимый регистр. Использование RBP обеспечивает согласованность между x86-64 и устаревшим кодом ассемблера x86. Последняя директива ассемблера, `.endprolog`, означает конец пролога для функции `Cc1_`. На рис. 5.3 показана структура стека и регистры аргументов после завершения пролога.

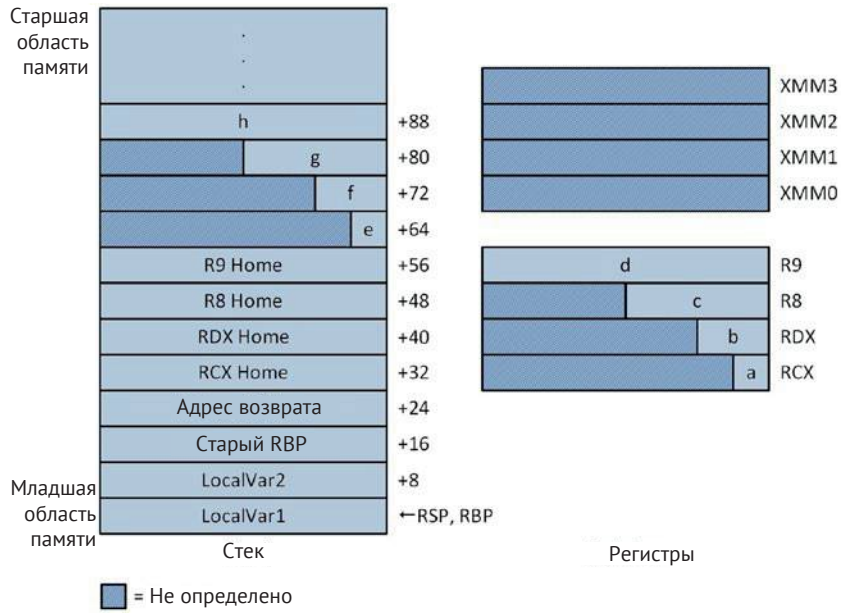


Рис. 5.3. Структура стека и регистры аргументов функции `Cc1_` после завершения пролога

Оператор `RBP_RA = 24` является директивой, аналогичной `equate`, которая присваивает значение 24 символу с именем `RBP_RA`. Это значение представляет собой дополнительные байты смещения (по сравнению со стандартной листовой функцией), необходимые для правильной ссылки на домашнюю область `Cc1_`, как показано на рис. 5.3. Следующий блок команд сохраняет регистры RCX, RDX, R8 и R9 в их соответствующих домашних областях в этом стеке. Этот шаг не является обязательным и включен в `Cc1_` в иллюстративных целях. Обратите внимание, что смещение каждой команды `mov` включает символическую константу `RBP_RA`. Другой вариант, разрешенный соглашением о вызовах Visual C++, – сохранить регистр аргументов в его соответствующей домашней области до команды `push rbp` с использованием RSP в качестве базового регистра (например, `mov [rsp+8],rcx`, `mov [rsp+16],rdx` и т. д.). Также имейте в виду, что функция может использовать свою домашнюю область для хранения других временных значений. При использовании в качестве альтернативного хранилища домашней области на нее не следует ссылаться в командах на языке ассемблера до тех пор, пока не будет указана директива `.endprolog`.

После операции сохранения домашней области функция `Cc1_` суммирует значения аргументов `a`, `b`, `c` и `d`. Затем она сохраняет эту промежуточную сум-

му в LocalVar1 в стеке с помощью команды mov [rbp],r8. Обратите внимание, что операция вычисления суммы расширяет знаковым битом значения аргументов a, b и c с помощью команды movsx или movsxd. Аналогичная последовательность команд используется для суммирования значений аргументов e, f, g и h, которые расположены в стеке и на которые ссылаются с помощью указателя кадра стека RBP и постоянного смещения. Символьная константа RBP_RA также используется здесь для учета дополнительного пространства стека, необходимого для ссылки на значения аргументов в стеке. Затем две промежуточные суммы складываются для получения окончательного результата в регистре RAX.

Эпилог функции должен освободить любое пространство хранения локального стека, которое было выделено в прологе, восстановить все энергонезависимые регистры, которые были сохранены в стеке, и выполнить возврат функции. Команда add rsp,16 освобождает 16 байт стека, выделенного Cc1_ в своем прологе. За этим следует команда pop rbp, которая восстанавливает регистр RBP вызывающей функции. Далее следует обязательная команда ret. Вот результат выполнения кода примера Ch05_09:

```
Результаты Cc1

a = 10
b = -200
c = 300
d = 4000
e = -20
f = 400
g = -600
h = -8000
sum = -4110
```

5.4.2. Использование энергонезависимых регистров общего назначения

Следующий пример программы называется Ch05_10 и демонстрирует, как использовать энергонезависимые регистры общего назначения в функции на языке 64-битного ассемблера. Он также демонстрирует дополнительные приемы программирования, касающиеся фреймов стека и использования локальных переменных. В листинге 5.10 показан исходный код C++ и ассемблера для примера программы Ch05_10.

Листинг 5.10. Пример Ch05_10

```
//-----------------------------------------------
//                Ch05_10.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cstdint>
```

```
using namespace std;

extern "C" bool Cc2_(const int64_t* a, const int64_t* b, int32_t n, int64_t * sum_a,
int64_t* sum_b, int64_t* prod_a, int64_t* prod_b);

int main()
{
    const int n = 6;
    int64_t a[n] = { 2, -2, -6, 7, 12, 5 };
    int64_t b[n] = { 3, 5, -7, 8, 4, 9 };
    int64_t sum_a, sum_b;
    int64_t prod_a, prod_b;

    bool rc = Cc2_(a, b, n, &sum_a, &sum_b, &prod_a, &prod_b);

    cout << "Результаты Cc2\n\n";

    if (rc)
    {
        const int w = 6;
        const char nl = '\n';
        const char* ws = " ";

        for (int i = 0; i < n; i++)
        {
            cout << "i: " << setw(w) << i << ws;
            cout << "a: " << setw(w) << a[i] << ws;
            cout << "b: " << setw(w) << b[i] << nl;
        }

        cout << nl;
        cout << "sum_a = " << setw(w) << sum_a << ws;
        cout << "sum_b = " << setw(w) << sum_b << nl;
        cout << "prod_a = " << setw(w) << prod_a << ws;
        cout << "prod_b = " << setw(w) << prod_b << nl;
    }
    else
        cout << "Недопустимый возвращаемый код\n";

    return 0;
}

;-----------------------------------------------
;                Ch05_10.asm
;-----------------------------------------------

; extern "C" void Cc2_(const int64_t* a, const int64_t* b, int32_t n, int64_t* sum_a,
int64_t* sum_b, int64_t* prod_a, int64_t* prod_b)

; Наименования констант:
;
; NUM_PUSHREG   = количество энергонезависимых регистров в прологе
; STK_LOCAL1    = размер в байтах пространства STK_LOCAL1 (см. число в книге)
; STK_LOCAL2    = размер в байтах пространства STK_LOCAL2 (см. число в книге)
; STK_PAD       = дополнительные байты (0 или 8) для 16-байтового выравнивания RSP
; STK_TOTAL     = полный размер локального стека в байтах
```

```
; RBP_RA        = количество байтов между RBP и адресом возврата в стеке

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

        .const
TestVal db 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

        .code
Cc2_ proc frame

; Сохранение энергонезависимых регистров в стеке
        push rbp
        .pushreg rbp
        push rbx
        .pushreg rbx
        push r12
        .pushreg r12
        push r13
        .pushreg r13

; Выделение пространства под локальный стек и настройка указателя фрейма
        sub rsp,STK_TOTAL               ;выделение пространства под стек
        .allocstack STK_TOTAL

        lea rbp,[rsp+STK_LOCAL2]        ;настройка указателя фрейма
        .setframe rbp,STK_LOCAL2

        .endprolog                      ;конец пролога

; Инициализация локальных переменных в стеке (демонстрационная)
        vmovdqu xmm5, xmmword ptr [TestVal]
        vmovdqa xmmword ptr [rbp-16],xmm5   ;сохранение xmm5 в LocalVar2A/2B
        mov qword ptr [rbp],0aah            ;сохранение 0xaa в LocalVar1A
        mov qword ptr [rbp+8],0bbh          ;сохранение 0xbb в LocalVar1B
        mov qword ptr [rbp+16],0cch         ;сохранение 0xcc в LocalVar1C
        mov qword ptr [rbp+24],0ddh         ;сохранение 0xdd в LocalVar1D

; Сохранение значений аргументов в домашнее пространство (опционально)
        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

; Необходимые инициализации для рабочего цикла
        test r8d,r8d                    ;n <= 0?
        jle Error                       ;переход, если n <= 0

        xor rbx,rbx                     ;rbx = смещение текущего элемента
        xor r10,r10                     ;r10 = sum_a
        xor r11,r11                     ;r11 = sum_b
        mov r12,1                       ;r12 = prod_a
        mov r13,1                       ;r13 = prod_b
```

```
; Вычисление суммы и произведения элементов массива
@@:     mov rax,[rcx+rbx]               ;rax = a[i]
        add r10,rax                     ;обновление sum_a
        imul r12,rax                    ;обновление prod_a
        mov rax,[rdx+rbx]               ;rax = b[i]
        add r11,rax                     ;обновление sum_b
        imul r13,rax                    ;обновление prod_b

        add rbx,8                       ;установка ebx на следующий элемент
        dec r8d                         ;выравнивание
        jnz @B                          ;повтор до завершения

; Save the final results
        mov [r9],r10                    ;сохранение sum_a
        mov rax,[rbp+RBP_RA+40]         ;rax = ptr в sum_b
        mov [rax],r11                   ;save sum_b
        mov rax,[rbp+RBP_RA+48]         ;rax = ptr в prod_a
        mov [rax],r12                   ;сохранение prod_a
        mov rax,[rbp+RBP_RA+56]         ;rax = ptr в prod_b
        mov [rax],r13                   ;сохранение prod_b
        mov eax,1                       ;код успешного выполнения

; Function epilog
Done:   lea rsp,[rbp+STK_LOCAL1+STK_PAD] ;восстановление rsp
        pop r13                          ;восстановление энергонезависимых регистров
        pop r12
        pop rbx
        pop rbp
        ret

Error: xor eax,eax                      ;код ошибки выполнения
        jmp Done
Cc2_    endp
        end
```

Как и в предыдущем примере этого раздела, цель кода C++ в листинге 5.10 состоит в том, чтобы подготовить простой тестовый пример для проверки функции языка ассемблера `Cc2_`. В этом примере функция `Cc2_` вычисляет суммы и произведения двух массивов 64-битных целых чисел со знаком. Затем результаты передаются в `cout`.

В верхней части кода на языке ассемблера находится ряд именованных констант, которые определяют, сколько места в стеке выделяется в прологе функции `Cc2_`. Как и в предыдущем примере, функция `Cc2_` включает атрибут кадра как часть своего оператора `proc`, чтобы указать, что он использует указатель кадра стека. Последовательность команд `push` сохраняет в стеке энергонезависимые регистры RBP, RBX, R12 и R13. Обратите внимание, что после каждой команды `push` используется директива `.pushreg`, которая инструктирует ассемблер добавлять информацию о каждой команде `push` в таблицы обработки исключений среды выполнения Visual C++.

Инструкция `sub rsp,STK_TOTAL` выделяет место в стеке для локальных переменных, а потом следует требуемая директива `.allocstack STK_TOTAL`. Затем регистр RBP инициализируется как указатель фрейма стека функции с по-

мощью команды `lea rbp,[rsp+STK_LOCAL2]`, которая устанавливает RBP равным `rsp+STK_LOCAL2`. Рисунок 5.4 иллюстрирует структуру стека после выполнения команды `lea`. Размещение RBP таким образом, чтобы он «разбивал» область локального стека на две части, позволяет ассемблеру генерировать машинный код, который немного более эффективен, поскольку на большую часть области локального стека можно ссылаться, используя 8-битное смещение со знаком вместо 32-битного смещения со знаком. Это также упрощает сохранение и восстановление энергонезависимых регистров XMM, о чем мы поговорим далее в этом разделе. После команды `lea` следует директива `.setframe rbp,STK_ LOCAL2`, которая позволяет ассемблеру правильно настраивать таблицы обработки исключений во время выполнения. Обратите внимание, что параметр размера этой директивы должен быть кратным 16 и меньше или равным 240. Директива `.endprolog` означает конец пролога функции `Cc2_`.

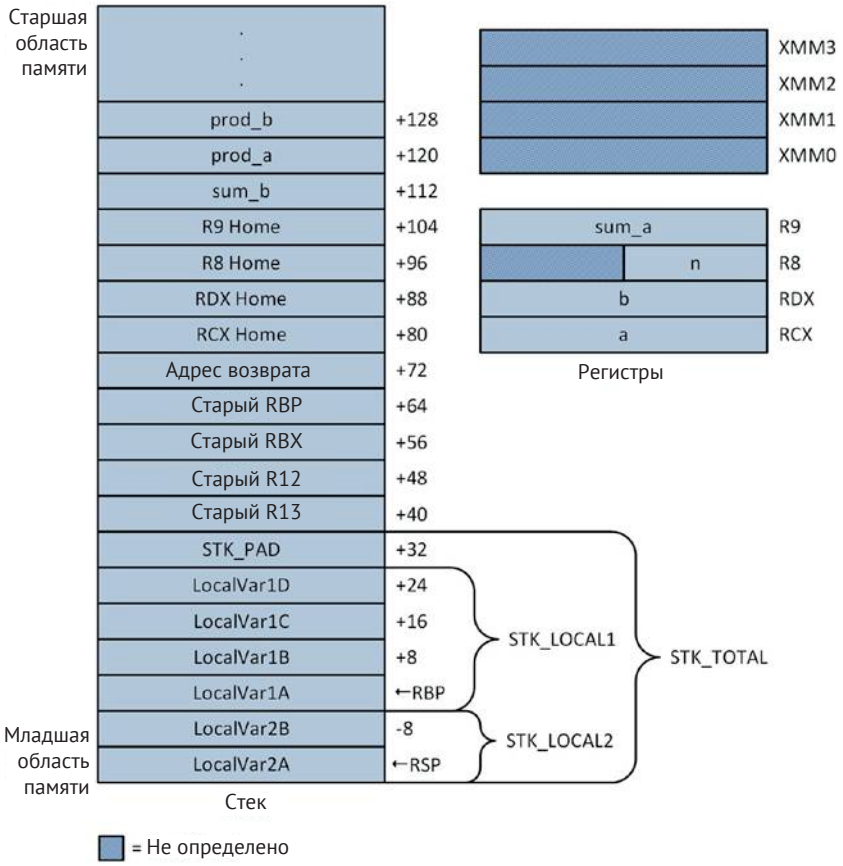


Рис. 5.4. Структура стека и регистры аргументов после выполнения команды `lea rbp,[rsp+STK_LOCAL2]` в функции `Cc2_`

Следующий блок кода содержит команды, которые инициализируют локальные переменные в стеке. Эти команды предназначены только для демонстрационных целей. Обратите внимание, что в этом блоке используется команда

vmovdqa [rbp-16],xmm5 (присвоение выровненных упакованных целочисленных значений), которая требует, чтобы операнд-приемник был выровнен по 16-байтовой границе. Эта команда воплощает в себе обязательное выравнивание регистра RSP по 16-байтовой границе согласно соглашению о вызовах. После инициализации локальных переменных регистры аргументов сохраняются в их исходных местоположениях, также просто для демонстрационных целей.

Логика основного цикла обработки проста. После проверки значения аргумента n функция Cc2_ инициализирует промежуточные значения sum_a(R10) и sum_b(R11) равными 0, а prod_a(R12) и prod_b(R13) равными 1. Затем она вычисляет сумму и произведение входных массивов a и b. Окончательные результаты сохраняются в ячейки памяти, указанные вызывающей функцией. Обратите внимание, что указатели на sum_b, prod_a и prod_b были переданы в Cc2_ с использованием стека.

Эпилог функции Cc2_ начинается с команды lea rsp,[rbp+STK_LOCAL1+STK_PAD], которая восстанавливает регистр RSP до значения, которое он имел сразу после команды push r13 в прологе. При восстановлении RSP в эпилоге соглашение о вызовах Visual C++ указывает, что необходимо использовать команду lea rsp,[RFP+X] или add rsp,X, где RFP обозначает регистр указателя фрейма, а X – постоянное значение. Это ограничивает количество шаблонов команд, которые должен идентифицировать обработчик исключений. Последующие команды pop восстанавливают прежние значения энергонезависимых регистров общего назначения до выполнения команды ret. Согласно соглашению о вызовах Visual C++, эпилоги функций не должны содержать никакой логики обработки, включая установку возвращаемого значения, поскольку это упрощает объем обработки, необходимой в обработчике исключений среды выполнения Visual C++. Подробнее о требованиях к эпилогам функций вы узнаете позже в этой главе. В результате выполнения кода примера Ch05_10 получается следующий вывод:

```
Результаты Cc2

i:  0   a:  2   b:  3
i:  1   a: -2   b:  5
i:  2   a: -6   b: -7
i:  3   a:  7   b:  8
i:  4   a: 12   b:  4
i:  5   a:  5   b:  9

sum_a =     18 sum_b =      22
prod_a = 10080 prod_b = -30240
```

5.4.3. Использование энергонезависимых регистров XMM

Ранее в этой главе вы узнали, как использовать энергозависимые регистры XMM для выполнения скалярной арифметики с плавающей запятой. Следующий пример исходного кода, Ch05_11, иллюстрирует соглашения о прологе и эпилоге, которые необходимо соблюдать, чтобы использовать энергонезависимые регистры XMM. В листинге 5.11 показан исходный код примера Ch05_11 на C++ и языке ассемблера.

Листинг 5.11. Пример Ch05_11

```
//-----------------------------------------------
//                Ch05_11.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#define _USE_MATH_DEFINES
#include <math.h>

using namespace std;

extern "C" bool Cc3_(const double* r, const double* h, int n, double* sa_cone, double* vol_
cone);

int main()
{
    const int n = 7;
    double r[n] = { 1, 1, 2, 2, 3, 3, 4.25 };
    double h[n] = { 1, 2, 3, 4, 5, 10, 12.5 };

    double sa_cone1[n], sa_cone2[n];
    double vol_cone1[n], vol_cone2[n];

    // Вычисляет площадь поверхности и объем прямого кругового конуса
    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);

    cout << fixed;
    cout << "Результаты Cc3\n\n";

    const int w = 14;
    const char nl = '\n';
    const char sp = ' ';

    for (int i = 0; i < n; i++)
    {
        cout << setprecision(2);
        cout << "r/h: " << setw(w) << r[i] << sp;
        cout << setw(w) << h[i] << nl;

        cout << setprecision(6);
        cout << "sa: " << setw(w) << sa_cone1[i] << sp;
        cout << setw(w) << sa_cone2[i] << nl;
        cout << "vol: " << setw(w) << vol_cone1[i] << sp;
        cout << setw(w) << vol_cone2[i] << nl;
        cout << nl;
    }

    return 0;
}
```

```
;-----------------------------------------------
;                Ch05_11.asm
;-----------------------------------------------

; extern "C" bool Cc3_(const double* r, const double* h, int n, double* sa_cone, double*
vol_cone)
; Наименования констант:
;
; NUM_PUSHREG = количество энергонезависимых регистров в прологе
; STK_LOCAL1 = размер в байтах пространства STK_LOCAL1 (см. число в книге)
; STK_LOCAL2 = размер в байтах пространства STK_LOCAL2 (см. число в книге)
; STK_PAD = дополнительные байты (0 или 8) для 16-байтового выравнивания RSP
; STK_TOTAL = полный размер локального стека в байтах
; RBP_RA = количество байтов между RBP и адресом возврата в стеке

NUM_PUSHREG = 7
STK_LOCAL1 = 16
STK_LOCAL2 = 64
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

        .const
r8_3p0 real8 3.0
r8_pi real8 3.14159265358979323846

        .code
Cc3_ proc frame

; Сохранение энергонезависимых регистров в стеке
        push rbp
        .pushreg rbp
        push rbx
        .pushreg rbx
        push rsi
        .pushreg rsi
        push r12
        .pushreg r12
        push r13
        .pushreg r13
        push r14
        .pushreg r14
        push r15
        .pushreg r15

; Выделение пространства под локальный стек и настройка указателя фрейма
        sub rsp,STK_TOTAL               ;выделение пространства под стек
        .allocstack STK_TOTAL

        lea rbp,[rsp+STK_LOCAL2]        ;настройка указателя фрейма
        .setframe rbp,STK_LOCAL2

; Сохранение энергонезависимых регистров XMM12-XMM15. Учтите, что STK_LOCAL2
; должен быть больше или равен количеству сохраненных регистров XMM, умноженному на 16
        vmovdqa xmmword ptr [rbp-STK_LOCAL2+48],xmm12
        .savexmm128 xmm12,48
```

```
        vmovdqa xmmword ptr [rbp-STK_LOCAL2+32],xmm13
        .savexmm128 xmm13,32
        vmovdqa xmmword ptr [rbp-STK_LOCAL2+16],xmm14
        .savexmm128 xmm14,16
        vmovdqa xmmword ptr [rbp-STK_LOCAL2],xmm15
        .savexmm128 xmm15,0
        .endprolog

; Доступ к локальным переменным в стеке (в демонстрационных целях)
        mov qword ptr [rbp],-1          ;LocalVar1A = -1
        mov qword ptr [rbp+8],-2        ;LocalVar1B = -2

; Инициализация переменных основного цикла. Учтите, что многие инициализации
; регистров приведены ниже для иллюстрации использования
; энергонезависимых регистров.
        mov esi,r8d                     ;esi = n
        test esi,esi                    ;n > 0?
        jg @F                           ;переход, если n > 0

        xor eax,eax                     ;назначение кода ошибки
        jmp done
@@:     xor rbx,rbx                     ;rbx = смещение элемента массива
        mov r12,rcx                     ;r12 = ptr в r
        mov r13,rdx                     ;r13 = ptr в h
        mov r14,r9                      ;r14 = ptr в sa_cone
        mov r15,[rbp+RBP_RA+40]         ;r15 = ptr в vol_cone
        vmovsd xmm14,real8 ptr [r8_pi]  ;xmm14 = pi
        vmovsd xmm15,real8 ptr [r8_3p0] ;xmm15 = 3.0

; Вычисление площади поверхности и объема конуса
; sa = pi * r * (r + sqrt(r * r + h * h))
; vol = pi * r * r * h / 3
@@:     vmovsd xmm0,real8 ptr [r12+rbx] ;xmm0 = r
        vmovsd xmm1,real8 ptr [r13+rbx] ;xmm1 = h
        vmovsd xmm12,xmm12,xmm0         ;xmm12 = r
        movsd xmm13,xmm13,xmm1          ;xmm13 = h

        vmulsd xmm0,xmm0,xmm0           ;xmm0 = r * r
        vmulsd xmm1,xmm1,xmm1           ;xmm1 = h * h
        vaddsd xmm0,xmm0,xmm1           ;xmm0 = r * r + h * h

        vsqrtsd xmm0,xmm0,xmm0          ;xmm0 = sqrt(r * r + h * h)
        vaddsd xmm0,xmm0,xmm12          ;xmm0 = r + sqrt(r * r + h * h)
        vmulsd xmm0,xmm0,xmm12          ;xmm0 = r * (r + sqrt(r * r + h * h))
        vmulsd xmm0,xmm0,xmm14          ;xmm0 = pi * r * (r + sqrt(r * r + h * h))

        vmulsd xmm12,xmm12,xmm12        ;xmm12 = r * r
        vmulsd xmm13,xmm13,xmm14        ;xmm13 = h * pi
        vmulsd xmm13,xmm13,xmm12        ;xmm13 = pi * r * r * h
        vdivsd xmm13,xmm13,xmm15        ;xmm13 = pi * r * r * h / 3

        vmovsd real8 ptr [r14+rbx],xmm0  ;сохранение площади поверхности
        vmovsd real8 ptr [r15+rbx],xmm13 ;сохранение объема

        add rbx,8                       ;установка указателя rbx на следующий элемент
        dec esi                         ;обновление счетчика
```

```
        jnz @B                          ;повторение до завершения

        mov eax,1                       ;назначение кода успешного выполнения

; Восстановление энергонезависимых регистров XMM
Done:   vmovdqa xmm12,xmmword ptr [rbp-STK_LOCAL2+48]
        vmovdqa xmm13,xmmword ptr [rbp-STK_LOCAL2+32]
        vmovdqa xmm14,xmmword ptr [rbp-STK_LOCAL2+16]
        vmovdqa xmm15,xmmword ptr [rbp-STK_LOCAL2]

; Эпилог функции
        lea rsp,[rbp+STK_LOCAL1+STK_PAD] ;восстановление rsp
        pop r15                          ;восстановление энергонезависимых регистров
        pop r14
        pop r13
        pop r12
        pop rsi
        pop rbx
        pop rbp
        ret

Cc3_    endp
        end
```

В примере `Ch05_11` код C++ вычисляет площадь поверхности и объем прямых круговых конусов. Он также вызывает функцию на языке ассемблера под названием `Cc3_`, которая выполняет те же вычисления площади поверхности и объема. Для расчета площади поверхности и объема конуса используются следующие формулы:

$$sa = \pi r\left(r + \sqrt{r^2 + h^2}\right);$$

$$vol = \pi r^2 h / 3.$$

Функция `Cc3_` начинается с сохранения энергонезависимых регистров общего назначения, которые она использует в стеке. Затем она выделяет указанный объем локального пространства стека и инициализирует RBP как указатель фрейма стека. Следующий блок кода сохраняет энергонезависимые регистры XMM12–XMM15 в стеке с использованием серии команд `vmovdqa`. После каждой команды `vmovdqa` должна следовать директива `.savexmm128`. Как и другие директивы пролога, директива `.savexmm128` указывает ассемблеру хранить информацию, касающуюся операции сохранения регистра XMM, в своих таблицах обработки исключений. Аргумент смещения директивы `.savexmm128` представляет смещение сохраненного регистра XMM в стеке относительно регистра RSP. Обратите внимание, что размер `STK_LOCAL2` должен быть больше или равен количеству сохраненных регистров XMM, умноженному на 16. Рисунок 5.5 иллюстрирует структуру стека после выполнения команды `vmovdqa xmmword ptr [rbp-STK_LOCAL2], xmm15`.

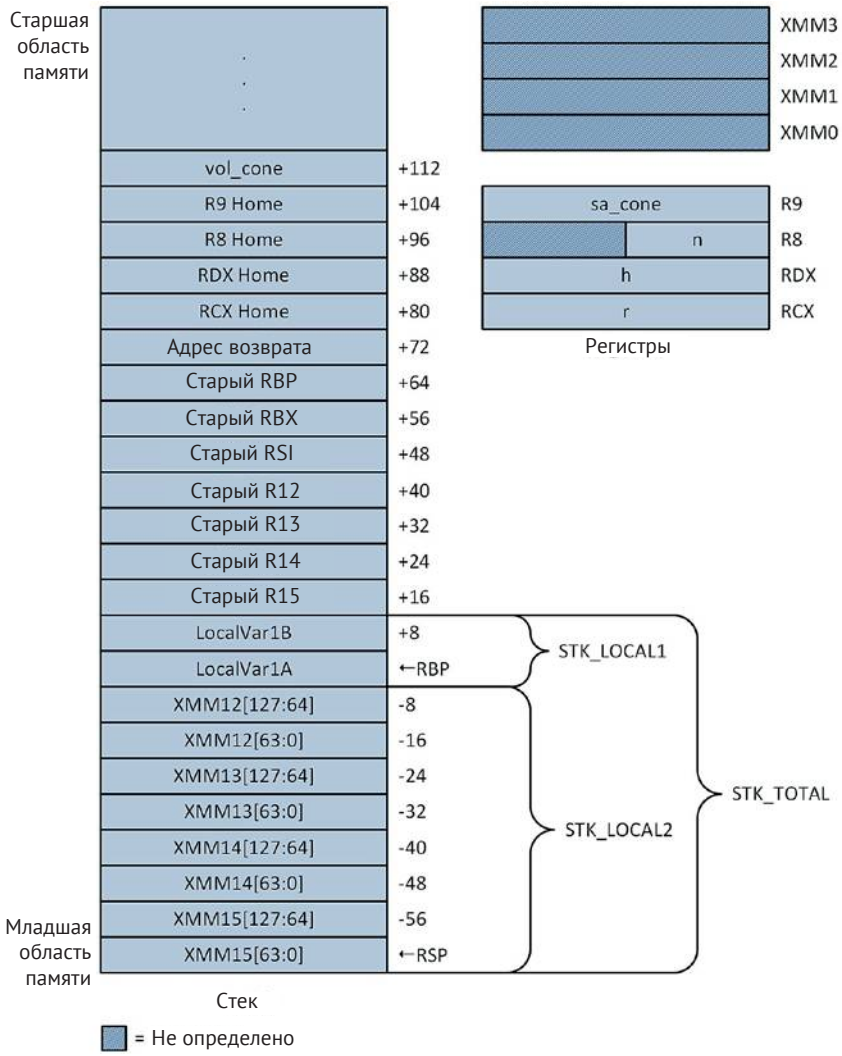


Рис. 5.5. Структура стека и регистры аргументов после выполнения команды `vmovdqa xmmword ptr [rbp-STK_LOCAL2],xmm15` в функции `Cc3_`

Следом за прологом осуществляется доступ к локальным переменным `LocalVar1A` и `LocalVar1B` (только в демонстрационных целях). Затем происходит инициализация регистров, используемых в основном цикле обработки. Обратите внимание, что многие из этих инициализаций либо неоптимальны, либо излишни; они выполняются только для пояснения использования энергонезависимых регистров, регистров общего назначения и XMM. Затем выполняется вычисление площади и объема поверхности конуса с использованием арифметики AVX с плавающей запятой двойной точности.

После завершения цикла обработки энергонезависимые регистры XMM восстанавливаются с помощью серии команд `vmovdqa`. Затем функция `Cc3_` освобождает пространство своего локального стека и восстанавливает ранее со-

храненные энергонезависимые регистры общего назначения, которые она использовала. Вот результат выполнения примера Ch05_11.

```
Результаты Cc3

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:        3.00       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

r/h:        4.25      12.50
sa:   233.025028 233.025028
vol:  236.437572 236.437572
```

5.4.4. Макросы для прологов и эпилогов

Целью предыдущих трех примеров исходного кода было разъяснение использования соглашения о вызовах Visual C++ для 64-битных нелистовых функций. Жесткие требования соглашения о вызовах к прологам и эпилогам функций довольно длинны и могут стать источником ошибок программирования. Важно понимать, что структура стека нелистовой функции в первую очередь определяется количеством энергонезависимых регистров (как общего назначения, так и XMM), которые нужно сохранить, и объемом памяти требуемого локального стека. Необходим метод для автоматизации большей части утомительной работы по кодированию, связанной с соглашением о вызовах.

В листинге 5.12 показан исходный код примера Ch05_12 на C++ и языке ассемблера. Этот пример исходного кода показывает, как использовать несколько макросов, которые я написал, чтобы упростить кодирование пролога и эпилога в функции, не являющейся листовой. Он также показывает, как вызвать библиотечную функцию C++.

Листинг 5.12. Пример Ch05_12

```
//-----------------------------------------------
//               Ch05_12.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

extern "C" bool Cc4_(const double* ht, const double* wt, int n, double* bsa1, double* bsa2,
double* bsa3);

int main()
{
    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] / 3600.0);
    }

    Cc4_(ht, wt, n, bsa1_b, bsa2_b, bsa3_b);

    cout << "Результаты Cc4_\n\n";
    cout << fixed;

    const char sp = ' ';

    for (int i = 0; i < n; i++)
    {
        cout << setprecision(1);
        cout << "высота: " << setw(6) << ht[i] << " см\n";
        cout << "вес: " << setw(6) << wt[i] << " кг\n";

        cout << setprecision(6);

        cout << "BSA (C++): ";
        cout << setw(10) << bsa1_a[i] << sp;
        cout << setw(10) << bsa2_a[i] << sp;
        cout << setw(10) << bsa3_a[i] << " (кв. м)\n";
        cout << "BSA (X86-64): ";
        cout << setw(10) << bsa1_b[i] << sp;
        cout << setw(10) << bsa2_b[i] << sp;
        cout << setw(10) << bsa3_b[i] << " (кв. м)\n\n";
    }
    return 0;
}
```

```
;-----------------------------------------------
;                Ch05_12.asm
;-----------------------------------------------

; extern "C" bool Cc4_(const double* ht, const double* wt, int n, double* bsa1, double*
bsa2, double* bsa3);

    include <MacrosX86-64-AVX.asmh>

            .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_3600p0   real8 3600.0

        .code
        extern pow:proc

Cc4_ proc frame
        _CreateFrame Cc4_,16,64,rbx,rsi,r12,r13,r14,r15
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9
        _EndProlog

; Сохранение регистров аргументов в домашнем пространстве (опционально)
; Имейте в виду, что домашнее пространство можно использовать для хранения других значений
        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

; Инициализация главного цикла. Обратите внимание, что указатели
; хранятся в энергонезависимых регистрах, что исключает
; перезагрузку после вызовов pow().
        test r8d,r8d                    ;n > 0?
        jg @F                           ;переход, если n > 0

        xor eax,eax                     ;генерация кода ошибки выполнения
        jmp Done

@@:     mov [rbp],r8d                       ;сохранение n в локальную переменную
        mov r12,rcx                         ;r12 = ptr в ht
        mov r13,rdx                         ;r13 = ptr в wt
        mov r14,r9                          ;r14 = ptr в bsa1
        mov r15,[rbp+Cc4_OffsetStackArgs]   ;r15 = ptr в bsa2
        mov rbx,[rbp+Cc4_OffsetStackArgs+8] ;rbx = ptr в bsa3
        xor rsi,rsi                         ;смещение элемента массива

; Выделение пространства в стеке для использования pow()
        sub rsp,32

; вычисление bsa1 = 0.007184 * pow(ht, 0.725) * pow(wt, 0.425);
@@:     vmovsd xmm0,real8 ptr [r12+rsi]     ;xmm0 = высота
        vmovsd xmm8,xmm8,xmm0
```

```
        vmovsd xmm1,real8 ptr [r8_0p725]
        call pow                        ;xmm0 = pow(ht, 0.725)
        vmovsd xmm6,xmm6,xmm0

        vmovsd xmm0,real8 ptr [r13+rsi]     ;xmm0 = вес
        vmovsd xmm9,xmm9,xmm0
        vmovsd xmm1,real8 ptr [r8_0p425]
        call pow                        ;xmm0 = pow(wt, 0.425)
        vmulsd xmm6,xmm6,real8 ptr [r8_0p007184]
        vmulsd xmm6,xmm6,xmm0           ;xmm6 = bsa1

; Вычисление bsa2 = 0.0235 * pow(ht, 0.42246) * pow(wt, 0.51456);
        vmovsd xmm0,xmm0,xmm8           ;xmm0 = высота
        vmovsd xmm1,real8 ptr [r8_0p42246]
        call pow                        ;xmm0 = pow(ht, 0.42246)
        vmovsd xmm7,xmm7,xmm0

        vmovsd xmm0,xmm0,xmm9           ;xmm0 = вес
        vmovsd xmm1,real8 ptr [r8_0p51456]
        call pow                        ;xmm0 = pow(wt, 0.51456)
        vmulsd xmm7,xmm7,real8 ptr [r8_0p0235]
        vmulsd xmm7,xmm7,xmm0           ;xmm7 = bsa2

; Вычисление bsa3 = sqrt(ht * wt / 60.0);
        vmulsd xmm8,xmm8,xmm9
        vdivsd xmm8,xmm8,real8 ptr [r8_3600p0]
        vsqrtsd xmm8,xmm8,xmm8          ;xmm8 = bsa3

; Сохранение результатов BSA
        vmovsd real8 ptr [r14+rsi],xmm6     ;сохранение результата bsa1
        vmovsd real8 ptr [r15+rsi],xmm7     ;сохранение результата bsa2
        vmovsd real8 ptr [rbx+rsi],xmm8     ;сохранение результата bsa3

        add rsi,8                       ;обновление смещения массива
        dec dword ptr [rbp]             ;n = n - 1
        jnz @B
        mov eax,1                       ;код успешного выполнения

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9
        _DeleteFrame rbx,rsi,r12,r13,r14,r15
        ret

Cc4_ endp
        end
```

Целью кода в `main` является инициализация нескольких тестовых примеров и вызов функции на языке ассемблера `Cc4_`. Эта функция вычисляет оценки площади поверхности человеческого тела (body surface area, BSA) с использованием нескольких хорошо известных уравнений. Эти уравнения представлены в табл. 5.4. В этой таблице каждое уравнение использует символ *H* для обозначения роста в сантиметрах, *W* для веса в килограммах и *BSA* для площади поверхности тела в квадратных метрах.

Таблица 5.4. Уравнения площади поверхности человеческого тела

Формула	Уравнение
Дюбуа и Дюбуа	$BSA = 0{,}007184 \times H^{0{,}725} \times W^{0{,}425}$
Гехана и Джорджа	$BSA = 0{,}0235 \times H^{0{,}42246} \times W^{0{,}51456}$
Мостеллера	$BSA = \sqrt{H \times W / 3600}$

Код примера `Ch05_12` на языке ассемблера начинается с оператора `include`, который подключает содержимое файла `MacrosX86-64-AVX.asmh`. Этот файл (исходный код не показан, но включен в загружаемый пакет кода главы 5) содержит ряд макросов, которые помогают автоматизировать большую часть рутинной работы по кодированию, связанной с соглашением о вызовах Visual C++. *Макрос* – это механизм замены текста ассемблера, который позволяет программисту представлять последовательность команд языка ассемблера, определений данных или других операторов с помощью одной текстовой строки. Макросы языка ассемблера обычно используются для генерации последовательностей команд, которые будут применяться более одного раза. Макросы также часто используются, чтобы избежать потери производительности при вызове функции. Пример исходного кода `Ch05_12` демонстрирует использование макросов соглашения о вызовах. Позже в этой книге вы узнаете, как определять свои собственные макросы.

На рис. 5.6 показана общая структура стека для нелистовой функции. Обратите внимание на сходство между этим рисунком и более подробными схемами стека на рис. 5.4 и 5.5. Макросы, определенные в `MacrosX86-64-AVX.asmh`, предполагают, что базовая структура стека функции будет соответствовать тому, что показано на рис. 5.6. Они позволяют функции настроить свой собственный подробный стековый фрейм, указывая необходимый объем локального стекового пространства и какие энергонезависимые регистры необходимо сохранить. Макросы также выполняют большую часть необходимых вычислений смещения стека, что снижает риск ошибки программирования в прологе или эпилоге.

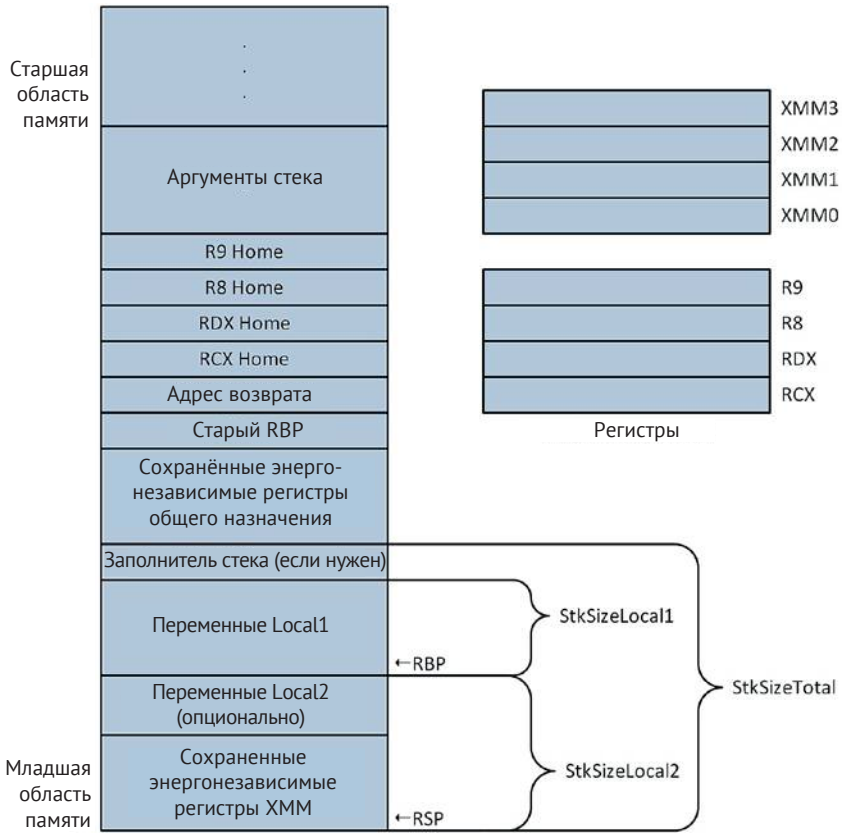


Рис. 5.6. Общая структура стека для нелистовой функции

Возвращаясь к ассемблерному коду, сразу после оператора `include` находится раздел `.const`, содержащий определения для различных значений констант с плавающей запятой, используемых в уравнениях BSA. Строка `extern pow:proc` позволяет использовать внешнюю библиотечную функцию `pow`. После оператора `Cc4_ proc frame` используется макрос `_CreateFrame` для генерации кода, который инициализирует фрейм стека функции. Он также сохраняет в стеке указанные энергонезависимые регистры общего назначения. Макрос требует нескольких дополнительных параметров, включая строку префикса и размер `StkSizeLocal1` и `StkSizeLocal2` в байтах (см. рис. 5.6). Макрос `_CreateFrame` использует указанную строку префикса для создания символических имен, которые можно использовать для ссылки на элементы в стеке. Довольно удобно использовать сокращенную версию имени функции в качестве строки префикса, но можно использовать любую уникальную текстовую строку. И `StkSizeLocal1`, и `StkSizeLocal2` должны без остатка делиться на 16. Значение `StkSizeLocal2` также должно быть меньше или равно 240 и больше или равно количеству сохраненных регистров XMM, умноженному на 16.

Следующий оператор использует макрос `_SaveXmmRegs` для сохранения указанных энергонезависимых регистров XMM в области сохранения XMM в сте-

ке. За ним следует макрос _EndProlog, означающий конец пролога функции. После завершения пролога регистр RBP настраивается как указатель фрейма стека функции. Также безопасно использовать любой из сохраненных энергонезависимых регистров общего назначения или регистров XMM после макроса _EndProlog.

Блок команд, следующий за _EndProlog, сохраняет регистры аргументов в их исходные местоположения в стеке. Обратите внимание, что каждая команда mov включает в себя символическое имя, которое соответствует смещению домашней области регистра в стеке относительно регистра RBP. Символические имена и соответствующие значения смещения были автоматически сгенерированы макросом _CreateFrame. Домашняя область также может использоваться для хранения временных данных вместо регистров аргументов, как упоминалось ранее в этой главе.

Затем происходит инициализация переменных цикла обработки. Значение n в регистре R8D проверяется на допустимость и сохраняется в стеке как локальная переменная. Затем несколько энергонезависимых регистров инициализируются как регистры-указатели. Энергонезависимые регистры используются, чтобы избежать перезагрузки регистров после каждого вызова функции pow библиотеки C++. Обратите внимание, что указатель на массив bsa2 загружается из стека с помощью команды mov r15,[rbp+Cc4_OffsetStackArgs]. Символьная константа Cc4_OffsetStackArgs также была автоматически сгенерирована макросом _CreateFrame и равна смещению первого аргумента стека относительно регистра RBP. Инструкция mov rbx,[rbp+Cc4_OffsetStackArgs+8] загружает аргумент bsa3 в регистр RBX; константа 8 включена как часть смещения исходного операнда, поскольку bsa3 – второй аргумент, передаваемый через стек.

Соглашение о вызовах Visual C++ требует, чтобы вызывающая функция выделяла домашнюю область этой функции в стеке. Эту операцию выполняет команда sub rsp,32. Следующий блок кода вычисляет значения BSA с использованием уравнений, показанных в табл. 5.4. Обратите внимание, что регистры XMM0 и XMM1 загружаются необходимыми значениями аргументов перед каждым вызовом pow. Также обратите внимание, что некоторые из возвращаемых значений pow сохраняются в энергонезависимых регистрах XMM до их фактического использования.

После завершения цикла обработки BSA идет эпилог для Cc4_. Перед выполнением команды ret функция должна восстановить все энергонезависимые регистры XMM и регистры общего назначения, которые она сохранила в прологе. Фрейм стека также должен быть правильно удален. Макрос _RestoreXmmRegs восстанавливает энергонезависимые регистры XMM. Обратите внимание, что этот макрос требует, чтобы порядок регистров в его списке аргументов соответствовал списку регистров, который использовался с макросом _SaveXmmRegs. Очистка фрейма стека и восстановление регистров общего назначения выполняются макросом _DeleteFrame. Порядок регистров, указанных в списке аргументов этого макроса, должен быть идентичен макросу _CreateFrame пролога. Макрос _DeleteFrame также восстанавливает регистр RSP из RBP – это означает, что нет необходимости включать явную команду add rsp,32 для освобождения домашней области, выделенной в стеке для pow. Вот результат выполнения кода примера Ch05_12.

```
Результаты Cc4

рост:  150.0 см
вес:    50.0 кг
BSA (C++):    1.432500 1.460836 1.443376 (кв. м)
BSA (X86-64): 1.432500 1.460836 1.443376 (кв. м)

рост:  160.0 см
вес:    60.0 кг
BSA (C++):    1.622063 1.648868 1.632993 (кв. м)
BSA (X86-64): 1.622063 1.648868 1.632993 (кв. м)

рост:  170.0 см
вес:    70.0 кг
BSA (C++):    1.809708 1.831289 1.818119 (кв. м)
BSA (X86-64): 1.809708 1.831289 1.818119 (кв. м)

рост:  180.0 см
вес:    80.0 кг
BSA (C++):    1.996421 2.009483 2.000000 (кв. м)
BSA (X86-64): 1.996421 2.009483 2.000000 (кв. м)

рост:  190.0 см
вес:    90.0 кг
BSA (C++):    2.182809 2.184365 2.179449 (кв. м)
BSA (X86-64): 2.182809 2.184365 2.179449 (кв. м)

рост:  200.0 см
вес:   100.0 кг
BSA (C++):    2.369262 2.356574 2.357023 (кв. м)
BSA (X86-64): 2.369262 2.356574 2.357023 (кв. м)
```

5.5. Заключение

В главе 5 вы изучили следующие ключевые моменты:

- команды `vadds[d|s]`, `vsubs[d|s]`, `vmuls[d|s]`, `vdivs[d|s]` и `vsqrts[d|s]` выполняют базовые операции арифметики с плавающей запятой двойной и одинарной точности;
- команды `vmovs[d|s]` копируют скалярное значение с плавающей запятой из одного регистра XMM в другой; они также используются для загрузки/хранения скалярных значений с плавающей запятой из/в память;
- команды `vcoms[d|s]` сравнивают два скалярных значения с плавающей запятой и устанавливают флаги состояния в RFLAGS для обозначения результата;
- команды `vcmps[d|s]` сравнивают два скалярных значения с плавающей запятой, используя предикат сравнения. Если предикат сравнения истинен, операнд назначения устанавливается на все единицы; в противном случае – все нули;
- команды `vcvts[d|s]2si` преобразуют скалярное значение с плавающей запятой в целое число со знаком; команды `vcvtsi2s[d|s]` выполняют противоположное преобразование;

- команда `vcvtsd2ss` преобразует скалярное значение с плавающей запятой двойной точности в одинарную точность; команда `vcvtss2sd` выполняет обратное преобразование;
- команда `vldmxcsr` загружает значение в регистр MXCSR; команда `vstmxcsr` сохраняет текущее содержимое регистра MXCSR;
- листовые функции могут использоваться для простых задач и не требуют пролога или эпилога. Нелистовая функция должна использовать пролог и эпилог для сохранения и восстановления энергонезависимых регистров, инициализации указателя фрейма стека, выделения пространства локальной памяти в стеке или вызова других функций.

Глава 6

Программирование AVX – упакованные числа с плавающей запятой

Примеры исходного кода в предыдущей главе поясняют основы программирования AVX с использованием скалярной арифметики с плавающей запятой. В этой главе вы узнаете, как использовать набор команд AVX для выполнения операций с использованием упакованных операндов с плавающей запятой. Глава начинается с трех примеров исходного кода, которые демонстрируют стандартные операции с плавающей запятой, включая базовые арифметические действия, сравнение данных и преобразование данных. Следующий набор примеров исходного кода иллюстрирует, как выполнять вычисления SIMD с использованием массивов чисел с плавающей запятой. Последние два примера исходного кода поясняют, как использовать набор команд AVX для ускорения транспонирования и умножения матриц.

В главе 4 вы узнали, что команды AVX выполняют операции с упакованными числами с плавающей запятой, используя 128-битные или 256-битные операнды. Во всех примерах исходного кода в этой главе используются 128-битные упакованные операнды с плавающей запятой, как одинарной, так и двойной точности, а также набор регистров XMM. В главе 9 вы узнаете, как использовать 256-битные упакованные операнды с плавающей запятой и набор регистров YMM.

6.1. Упакованная арифметика с плавающей запятой

В листинге 6.1 показан исходный код примера `Ch06_01`, который демонстрирует выполнение общих арифметических операций с использованием упакованных операндов с плавающей запятой одинарной и двойной точности. Обратите внимание на правильные методы выравнивания упакованных операндов с плавающей запятой в памяти.

Листинг 6.1. Пример Ch06_01

```
//-----------------------------------------------
//               XmmVal.h
//-----------------------------------------------

#pragma once
#include <string>
#include <cstdint>
#include <sstream>
#include <iomanip>

struct XmmVal
{
public:
    union
    {
    int8_t m_I8[16];
    int16_t m_I16[8];
    int32_t m_I32[4];
    int64_t m_I64[2];
    uint8_t m_U8[16];
    uint16_t m_U16[8];
    uint32_t m_U32[4];
    uint64_t m_U64[2];
    float m_F32[4];
    double m_F64[2];
    };

//-----------------------------------------------
//               Ch06_01.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#define _USE_MATH_DEFINES
#include <math.h>
#include "XmmVal.h"

using namespace std;

extern "C" void AvxPackedMathF32_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);
extern "C" void AvxPackedMathF64_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

void AvxPackedMathF32(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[8];

    a.m_F32[0] = 36.0f; b.m_F32[0] = -(float)(1.0 / 9.0);
    a.m_F32[1] = (float)(1.0 / 32.0); b.m_F32[1] = 64.0f;
    a.m_F32[2] = 2.0f; b.m_F32[2] = -0.0625f;
    a.m_F32[3] = 42.0f; b.m_F32[3] = 8.666667f;

    AvxPackedMathF32_(a, b, c);
```

```
    cout << ("\nРезультаты AvxPackedMathF32\n");
    cout << "a: " << a.ToStringF32() << '\n';
    cout << "b: " << b.ToStringF32() << '\n';
    cout << '\n';
    cout << "addps: " << c[0].ToStringF32() << '\n';
    cout << "subps: " << c[1].ToStringF32() << '\n';
    cout << "mulps: " << c[2].ToStringF32() << '\n';
    cout << "divps: " << c[3].ToStringF32() << '\n';
    cout << "absps b: " << c[4].ToStringF32() << '\n';
    cout << "sqrtps a:" << c[5].ToStringF32() << '\n';
    cout << "minps: " << c[6].ToStringF32() << '\n';
    cout << "maxps: " << c[7].ToStringF32() << '\n';
}

void AvxPackedMathF64(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[8];

    a.m_F64[0] = 2.0; b.m_F64[0] = M_E;
    a.m_F64[1] = M_PI; b.m_F64[1] = -M_1_PI;

    AvxPackedMathF64_(a, b, c);

    cout << ("\nРезультаты AvxPackedMathF64\n");
    cout << "a: " << a.ToStringF64() << '\n';
    cout << "b: " << b.ToStringF64() << '\n';
    cout << '\n';
    cout << "addpd: " << c[0].ToStringF64() << '\n';
    cout << "subpd: " << c[1].ToStringF64() << '\n';
    cout << "mulpd: " << c[2].ToStringF64() << '\n';
    cout << "divpd: " << c[3].ToStringF64() << '\n';
    cout << "abspd b: " << c[4].ToStringF64() << '\n';
    cout << "sqrtpd a:" << c[5].ToStringF64() << '\n';
    cout << "minpd: " << c[6].ToStringF64() << '\n';
    cout << "maxpd: " << c[7].ToStringF64() << '\n';
}

int main()
{
    AvxPackedMathF32();
    AvxPackedMathF64();
    return 0;
}

;-----------------------------------------------
;                Ch06_01.asm
;-----------------------------------------------

        .const
        align 16
AbsMaskF32 dword 7fffffffh, 7fffffffh, 7fffffffh, 7fffffffh ;Абсолютное значение маски для
SPFP
AbsMaskF64 qword 7fffffffffffffffh, 7fffffffffffffffh       ;Абсолютное значение маски для
DPFP
```

```
; extern "C" void AvxPackedMathF32_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

        .code
AvxPackedMathF32_ proc
; Загрузка упакованных чисел SPFP
        vmovaps xmm0,xmmword ptr [rcx]  ;xmm0 = a
        vmovaps xmm1,xmmword ptr [rdx]  ;xmm1 = b

; Сложение упакованных чисел SPFP
        vaddps xmm2,xmm0,xmm1
        vmovaps [r8+0],xmm2

; Вычитание упакованных чисел SPFP
        vsubps xmm2,xmm0,xmm1
        vmovaps [r8+16],xmm2

; Перемножение упакованных чисел SPFP
        vmulps xmm2,xmm0,xmm1
        vmovaps [r8+32],xmm2

; Деление упакованных чисел SPFP
        vdivps xmm2,xmm0,xmm1
        vmovaps [r8+48],xmm2

; Абсолютное значение числа SPFP (b)
        vandps xmm2,xmm1,xmmword ptr [AbsMaskF32]
        vmovaps [r8+64],xmm2

; Квадратный корень из упакованного числа SPFP(a)
        vsqrtps xmm2,xmm0
        vmovaps [r8+80],xmm2

; Минимальное упакованное число SPFP
        vminps xmm2,xmm0,xmm1
        vmovaps [r8+96],xmm2

; Максимальное упакованное число SPFP
        vmaxps xmm2,xmm0,xmm1
        vmovaps [r8+112],xmm2
        ret
AvxPackedMathF32_ endp

; extern "C" void AvxPackedMathF64_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

AvxPackedMathF64_ proc
; Загрузка упакованных чисел DPFP
        vmovapd xmm0,xmmword ptr [rcx]  ;xmm0 = a
        vmovapd xmm1,xmmword ptr [rdx]  ;xmm1 = b

; Сложение упакованных чисел DPFP
        vaddpd xmm2,xmm0,xmm1
        vmovapd [r8+0],xmm2

; Вычитание упакованных чисел DPFP
        vsubpd xmm2,xmm0,xmm1
        vmovapd [r8+16],xmm2
```

```
; Перемножение упакованных чисел DPFP
        vmulpd xmm2,xmm0,xmm1
        vmovapd [r8+32],xmm2

; Деление упакованных чисел DPFP
        vdivpd xmm2,xmm0,xmm1
        vmovapd [r8+48],xmm2

; Абсолютное значение упакованного числа DPFP (b)
        vandpd xmm2,xmm1,xmmword ptr [AbsMaskF64]
        vmovapd [r8+64],xmm2

; Квадратный корень из упакованного числа DPFP (a)
        vsqrtpd xmm2,xmm0
        vmovapd [r8+80],xmm2

; Минимальное упакованное число DPFP
        vminpd xmm2,xmm0,xmm1
        vmovapd [r8+96],xmm2

; Максимальное упакованное число DPFP
        vmaxpd xmm2,xmm0,xmm1
        vmovapd [r8+112],xmm2
        ret
AvxPackedMathF64_ endp
        end
```

Листинг 6.1 начинается с объявления структуры C++ с именем `XmmVal`, объявленной в заголовочном файле `XmmVal.h`. Эта структура содержит публичное анонимное объединение, которое облегчает обмен упакованными данными операндов между функциями, написанными на C++ и на языке ассемблера x86. Члены этого объединения соответствуют типам упакованных данных, которые могут использоваться с регистром XMM. Структура `XmmVal` также включает несколько функций-членов, которые форматируют и отображают содержимое переменной `XmmVal` (исходный код для этих функций-членов не показан, но включен в пакет для скачивания примеров главы).

В верхней части кода C++ находятся объявления функций на языке ассемблера x86-64 `AvxPackedMath32_` и `AvxPackedMath64_`. Эти функции выполняют обычные упакованные арифметические операции с использованием предоставленных значений аргумента `XmmVal`. Обратите внимание, что и для `AvxPackedMath32_`, и для `AvxPackedMath64_` аргументы `a` и `b` передаются по ссылке, а не прямым значением, чтобы избежать накладных расходов на операцию копирования `XmmVal`. В этом примере для передачи `a` и `b` также можно использовать указатели, поскольку они не отличаются от ссылок с точки зрения функций языка ассемблера x86-64.

Сразу после объявлений функций на языке ассемблера следует определение функции `AvxPackedMathF32`. Эта функция содержит код, демонстрирующий упакованную арифметику одинарной точности с плавающей запятой. Обратите внимание, что все переменные `XmmVal` – `a`, `b` и `c` – определены с помощью спецификатора `alignas(16)`, который инструктирует компилятор C++ выровнять каждую переменную по 16-байтовой границе. Следующий набор операторов инициализирует массивы `a.m_F32` и `b.m_F32` с тестовыми значениями. Затем код C++ вызывает функцию языка ассемблера `AvxPackedMathF32_` для выполнения раз-

личных арифметических операций с использованием упакованных операндов с плавающей запятой одинарной точности. Далее результаты отображаются с помощью вывода в cout. Код C++ также содержит функцию AvxPackedMath64, которая иллюстрирует выполнение арифметических операций с использованием упакованных операндов с плавающей запятой двойной точности. Организация этой функции аналогична AvxPackedMath32.

Код на языке ассемблера x86-64 в примере Ch06_01 начинается с раздела .const, который определяет значения упакованной маски для вычисления абсолютных значений с плавающей запятой. Оператор align 16 является директивой MASM, которая инструктирует ассемблер выровнять следующую переменную (или команду) по 16-байтовой границе. Использование этого оператора гарантирует, что маска AbsMaskF32 будет правильно выровнена. Обратите внимание, что, в отличие от x86-SSE, операнды команд x86-AVX в памяти не обязательно должны быть правильно выровнены, за исключением команд, которые явно указывают выровненные операнды (например, vmovaps). Однако по возможности настоятельно рекомендуется выполнять правильное выравнивание упакованных операндов в памяти, чтобы избежать потерь быстродействия, которые могут возникнуть, когда процессор обращается к невыровненному операнду. Для обеспечения выравнивания AbsMaskF64 не требуется вторая директива align 16, поскольку размер AbsMaskF32 составляет 16 байт, но лишней эта директива тоже не будет.

Первая команда AvxPackedMathF32_,vmovaps xmm0,xmmword ptr [rcx] загружает аргумент a (т. е. четыре значения с плавающей запятой, хранящихся в XmmVal a) в регистр XMM0. Как упоминалось в предыдущем абзаце, команда vmovaps (присвоение выровненных упакованных значений с плавающей запятой одинарной точности) требует, чтобы исходные операнды были правильно выровнены в памяти. Вот почему в коде C++ использовались спецификаторы alignas(16). Оператор xmmword ptr указывает ассемблеру обрабатывать указатель ячейки памяти RCX как 128-битный операнд. В данном случае использование оператора xmmword ptr не обязательно и используется для улучшения читаемости кода. Следующая команда vmovaps xmm1,xmmword ptr [rdx] загружает b в регистр XMM1. Команда vaddps xmm2,xmm0,xmm1 выполняет сложение упакованных операндов с плавающей запятой одинарной точности, используя содержимое регистров XMM0 и XMM1. Затем она сохраняет вычисленную сумму в регистре XMM2, как показано на рис. 6.1. Обратите внимание, что команда vaddps не изменяет содержимое двух своих исходных операндов. Следующая команда vmovaps xmmword ptr [r8],xmm2 сохраняет результат арифметического сложения упакованных чисел в c[0].

Рис. 6.1. Выполнение команды vaddps

Последующие команды vsubps, vmulps и vdivps выполняют вычитание упакованных чисел с плавающей запятой одинарной точности, умножение и деление. За ними следует команда vandps xmm2,xmm1,xmmword ptr [AbsMaskF32], которая вычисляет упакованные абсолютные значения с использованием аргумента b. Команда vandps (побитовое И упакованных значений с плавающей запятой одинарной точности) выполняет побитовое И, используя два своих исходных операнда. Обратите внимание, что все биты в каждом двойном слове AbsMaskF32 установлены в единицу, кроме самого старшего бита, который соответствует биту знака числа с плавающей запятой одинарной точности. Значение знакового бита, равное нулю, соответствует положительному числу с плавающей запятой, как сказано в главе 4. Выполнение побитового И с использованием этой 128-битной маски и упакованного операнда с плавающей запятой одинарной точности b обнуляет знаковый бит каждого элемента и генерирует упакованные абсолютные значения.

Остальные команды в AvxPackedMathF32_ вычисляют квадратный корень (vsqrtps), минимум (vminps) и максимум (vmaxps). Структура функции AvxPackedMathF64_ аналогична AvxPackedMathF32_. Функция AvxPackedMathF64_ выполняет свои вычисления, используя версии с плавающей запятой двойной точности тех же команд, которые применяются в AvxPackedMathF32_. Вот результат выполнения примера исходного кода Ch06_01:

```
Результат AvxPackedMathF32
a:          36.000000   0.031250 |     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 b:     0.111111  64.000000 |     0.062500   8.666667
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

Результат AvxPackedMathF64
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 b:          2.718281828459 |          0.318309886184
sqrtpd a:         1.414213562373 |          1.772453850906
minpd:            2.000000000000 |         -0.318309886184
maxpd:            2.718281828459 |          3.141592653590
```

6.2. Сравнение упакованных чисел с плавающей запятой

В главе 5 вы узнали, как сравнивать скалярные значения с плавающей запятой одинарной и двойной точности с помощью команд vcmps[d|s]. В этом разделе вы узнаете, как сравнивать упакованные значения с плавающей запятой одинарной и двойной точности с помощью команд vcmpp[d|s]. Подобно своим скалярным аналогам, команды сравнения упакованных значений требуют четырех операндов: операнда-приемника, двух операндов-источников и предиката немедленного сравнения. Команды сравнения упакованных значений обозначают свои результаты с использованием масок четверных (vcmppd) или двойных слов (vcmpps), состоящих из всех нулей (ложный результат сравнения) или всех единиц (истинный результат сравнения). В листинге 6.2 показан исходный код примера Ch06_02.

Листинг 6.2. Пример Ch06_02

```
//-----------------------------------------------
//              Ch06_02.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#define _USE_MATH_DEFINES
#include <math.h>
#include <limits>
#include "XmmVal.h"

using namespace std;

extern "C" void AvxPackedCompareF32_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);
extern "C" void AvxPackedCompareF64_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

const char* c_CmpStr[8] =
{
    "EQ", "NE", "LT", "LE", "GT", "GE", "ORDERED", "UNORDERED"
};

void AvxPackedCompareF32(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[8];

    a.m_F32[0] = 2.0; b.m_F32[0] = 1.0;
    a.m_F32[1] = 7.0; b.m_F32[1] = 12.0;
    a.m_F32[2] = -6.0; b.m_F32[2] = -6.0;
    a.m_F32[3] = 3.0; b.m_F32[3] = 8.0;

    for (int i = 0; i < 2; i++)
    {
        if (i == 1)
            a.m_F32[0] = numeric_limits<float>::quiet_NaN();
```

```
            AvxPackedCompareF32_(a, b, c);

            cout << "\nРезультаты AvxPackedCompareF32 (iteration = " << i << ")\n";
            cout << setw(11) << 'a' << ':' << a.ToStringF32() << '\n';
            cout << setw(11) << 'b' << ':' << b.ToStringF32() << '\n';
            cout << '\n';

            for (int j = 0; j < 8; j++)
                cout << setw(11) << c_CmpStr[j] << ':' << c[j].ToStringX32() << '\n';
    }
}

void AvxPackedCompareF64(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[8];

    a.m_F64[0] = 2.0; b.m_F64[0] = M_E;
    a.m_F64[1] = M_PI; b.m_F64[1] = -M_1_PI;

    for (int i = 0; i < 2; i++)
    {
        if (i == 1)
        {
            a.m_F64[0] = numeric_limits<double>::quiet_NaN();
            b.m_F64[1] = a.m_F64[1];
        }

        AvxPackedCompareF64_(a, b, c);

        cout << "\nРезультаты AvxPackedCompareF64 (iteration = " << i << ")\n";
        cout << setw(11) << 'a' << ':' << a.ToStringF64() << '\n';
        cout << setw(11) << 'b' << ':' << b.ToStringF64() << '\n';
        cout << '\n';

        for (int j = 0; j < 8; j++)
            cout << setw(11) << c_CmpStr[j] << ':' << c[j].ToStringX64() << '\n';
    }
}

int main()
{
    AvxPackedCompareF32();
    AvxPackedCompareF64();
    return 0;
}

;-----------------------------------------------
;               Ch06_02.asm
;-----------------------------------------------

        include <cmpequ.asmh>

; extern "C" void AvxPackedCompareF32_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

        .code
```

```
AvxPackedCompareF32_ proc
        vmovaps xmm0,[rcx]              ;xmm0 = a
        vmovaps xmm1,[rdx]              ;xmm1 = b

; Сравнение по условию EQUAL (равно)
        vcmpps xmm2,xmm0,xmm1,CMP_EQ
        vmovdqa xmmword ptr [r8],xmm2

; Сравнение по условию NOT EQUAL (не равно)
        vcmpps xmm2,xmm0,xmm1,CMP_NEQ
        vmovdqa xmmword ptr [r8+16],xmm2

; Сравнение по условию LESS THAN (меньше, чем)
        vcmpps xmm2,xmm0,xmm1,CMP_LT
        vmovdqa xmmword ptr [r8+32],xmm2

; Сравнение по условию LESS THAN OR EQUAL (меньше или равно)
        vcmpps xmm2,xmm0,xmm1,CMP_LE
        vmovdqa xmmword ptr [r8+48],xmm2

; Сравнение по условию GREATER THAN (больше, чем)
        vcmpps xmm2,xmm0,xmm1,CMP_GT
        vmovdqa xmmword ptr [r8+64],xmm2

; Сравнение по условию GREATER THAN OR EQUAL (больше или равно)
        vcmpps xmm2,xmm0,xmm1,CMP_GE
        vmovdqa xmmword ptr [r8+80],xmm2

; Сравнение по условию ORDERED (определенное)
        vcmpps xmm2,xmm0,xmm1,CMP_ORD
        vmovdqa xmmword ptr [r8+96],xmm2

; Сравнение по условию UNORDERED (неопределенное)
        vcmpps xmm2,xmm0,xmm1,CMP_UNORD
        vmovdqa xmmword ptr [r8+112],xmm2
        ret
AvxPackedCompareF32_ endp

; extern "C" void AvxPackedCompareF64_(const XmmVal& a, const XmmVal& b, XmmVal c[8]);

AvxPackedCompareF64_ proc
        vmovapd xmm0,[rcx]              ;xmm0 = a
        vmovapd xmm1,[rdx]              ;xmm1 = b

; Сравнение по условию EQUAL (равно)
        vcmppd xmm2,xmm0,xmm1,CMP_EQ
        vmovdqa xmmword ptr [r8],xmm2

; Сравнение по условию NOT EQUAL (не равно)
        vcmppd xmm2,xmm0,xmm1,CMP_NEQ
        vmovdqa xmmword ptr [r8+16],xmm2

; Сравнение по условию LESS THAN (меньше, чем)
        vcmppd xmm2,xmm0,xmm1,CMP_LT
        vmovdqa xmmword ptr [r8+32],xmm2
```

```
; Сравнение по условию LESS THAN OR EQUAL (меньше или равно)
        vcmppd xmm2,xmm0,xmm1,CMP_LE
        vmovdqa xmmword ptr [r8+48],xmm2

; Сравнение по условию GREATER THAN (больше, чем)
        vcmppd xmm2,xmm0,xmm1,CMP_GT
        vmovdqa xmmword ptr [r8+64],xmm2

; Сравнение по условию GREATER THAN OR EQUAL (больше или равно)
        vcmppd xmm2,xmm0,xmm1,CMP_GE
        vmovdqa xmmword ptr [r8+80],xmm2

; Сравнение по условию ORDERED (определенное)
        vcmppd xmm2,xmm0,xmm1,CMP_ORD
        vmovdqa xmmword ptr [r8+96],xmm2

; Сравнение по условию UNORDERED (неопределенное)
        vcmppd xmm2,xmm0,xmm1,CMP_UNORD
        vmovdqa xmmword ptr [r8+112],xmm2
        ret
AvxPackedCompareF64_ endp
        end
```

На рис. 6.2 показано выполнение команд `vcmpps xmm2,xmm0,xmm1,0` и `vcmppd xmm2,xmm0,xmm1,1`. В этих примерах операнды предиката сравнения 0 и 1 обозначают проверку условий «равно» и «меньше» соответственно.

Рис. 6.2. Выполнение команд `vcmpps` и `vcmppd`

Функция C++ `AvxPackedCompareF32` начинается с инициализации пары тестовых переменных `XmmVal`. Подобно примеру, который вы видели в предыдущем разделе, спецификатор `alignas(16)` используется с каждой переменной `XmmVal` для принудительного выравнивания по 16-байтовой границе. Оставшийся код вызывает ассемблерную функцию `AvxPackedCompareF32_` и отображает ре-

зультаты. Обратите внимание, что на второй итерации цикла for константа numeric_limits<float>::quiet_NaN() заменяет одно из значений в XmmVal a, чтобы проиллюстрировать работу предикатов определенного и неопределенного сравнений. *Определенное сравнение* истинно, если оба операнда являются допустимыми значениями. *Неопределенное сравнение* истинно, когда один или оба операнда имеют значение NaN или ошибочное представление. Подстановка numeric_limits<float>::quiet_NaN() вместо одного из значений в XmmVal a генерирует истинный результат для неопределенного сравнения. Код C++ также включает функцию AvxPackedCompareF64, которая является аналогом с двойной точностью функции AvxPackedCompareF32.

Код языка ассемблера x86-64 начинается с команды include <cmpequ.asmh>. Этот файл, который уже встречался вам в примере Ch05_05, содержит предикаты сравнения, которые будут использоваться в этом примере исходного кода с командами vcmpp[d|s]. Функция языка ассемблера AvxPackeCompareF32_ начинается с двух команд vmovaps, которые загружают аргументы a и b в регистры XMM0 и XMM1 соответственно. Следующая команда vcmpps xmm2,xmm0,xmm1,CMP_EQ сравнивает упакованные операнды a и b на равенство и сохраняет упакованный результат (четыре значения маски двойного слова) в регистр XMM2. Затем содержимое регистра XMM2 сохраняется в массив результатов с помощью команды vmovdqa xmmword ptr [r8],xmm2. Оставшийся код в AvxPackedCompareF32_ выполняет дополнительные операции сравнения с использованием распознаваемых предикатов сравнения. Функция языка ассемблера AvxPackedCompareF64_ демонстрирует, как использовать команду vcmppd для выполнения сравнений упакованных чисел с плавающей запятой двойной точности. Вот результат выполнения кода примера Ch06_02:

```
Результат AvxPackedCompareF32 (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
         NE:        FFFFFFFF     FFFFFFFF |      00000000   FFFFFFFF
         LT:        00000000     FFFFFFFF |      00000000   FFFFFFFF
         LE:        00000000     FFFFFFFF |      FFFFFFFF   FFFFFFFF
         GT:        FFFFFFFF     00000000 |      00000000   00000000
         GE:        FFFFFFFF     00000000 |      FFFFFFFF   00000000
    ORDERED:        FFFFFFFF     FFFFFFFF |      FFFFFFFF   FFFFFFFF
  UNORDERED:        00000000     00000000 |      00000000   00000000

Результат AvxPackedCompareF32 (iteration = 1)
          a:             nan     7.000000 |     -6.000000   3.000000
          b:        1.000000    12.000000 |     -6.000000   8.000000

         EQ:        00000000     00000000 |      FFFFFFFF   00000000
         NE:        FFFFFFFF     FFFFFFFF |      00000000   FFFFFFFF
         LT:        00000000     FFFFFFFF |      00000000   FFFFFFFF
         LE:        00000000     FFFFFFFF |      FFFFFFFF   FFFFFFFF
         GT:        00000000     00000000 |      00000000   00000000
         GE:        00000000     00000000 |      FFFFFFFF   00000000
    ORDERED:        00000000     FFFFFFFF |      FFFFFFFF   FFFFFFFF
```

```
  UNORDERED:        FFFFFFFF     00000000 |      00000000   00000000

Результат AvxPackedCompareF64 (iteration = 0)
          a:               2.000000000000 |           3.141592653590
          b:               2.718281828459 |          -0.318309886184

         EQ:           0000000000000000 |         0000000000000000
         NE:           FFFFFFFFFFFFFFFF |         FFFFFFFFFFFFFFFF
         LT:           FFFFFFFFFFFFFFFF |         0000000000000000
         LE:           FFFFFFFFFFFFFFFF |         0000000000000000
         GT:           0000000000000000 |         FFFFFFFFFFFFFFFF
         GE:           0000000000000000 |         FFFFFFFFFFFFFFFF
    ORDERED:           FFFFFFFFFFFFFFFF |         FFFFFFFFFFFFFFFF
  UNORDERED:           0000000000000000 |         0000000000000000

Результат AvxPackedCompareF64 (iteration = 1)
          a:                        nan |           3.141592653590
          b:               2.718281828459 |           3.141592653590
         EQ:           0000000000000000 |         FFFFFFFFFFFFFFFF
         NE:           FFFFFFFFFFFFFFFF |         0000000000000000
         LT:           0000000000000000 |         0000000000000000
         LE:           0000000000000000 |         FFFFFFFFFFFFFFFF
         GT:           0000000000000000 |         0000000000000000
         GE:           0000000000000000 |         FFFFFFFFFFFFFFFF
    ORDERED:           0000000000000000 |         FFFFFFFFFFFFFFFF
  UNORDERED:           FFFFFFFFFFFFFFFF |         0000000000000000
```

6.3. Преобразования упакованных чисел с плавающей запятой

Следующий пример исходного кода называется `Ch06_03`. В этом примере показаны преобразования упакованных целых чисел двойной точности со знаком в представление с плавающей запятой и наоборот. Он также иллюстрирует преобразование между упакованными значениями с плавающей запятой одинарной точности и упакованными значениями двойной точности. В листинге 6.3 показан исходный код примера `Ch06_03`.

Листинг 6.3. Пример Ch06_03

```
//-----------------------------------------------
//               Ch06_03.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#define _USE_MATH_DEFINES
#include <math.h>
#include "XmmVal.h"

using namespace std;

// Порядок значений в данном перечислении должен совпадать с таблицей переходов,
```

```
// которая объявлена в файле Ch06_03_.asm.
enum CvtOp : unsigned int
{
    I32_F32, F32_I32, I32_F64, F64_I32, F32_F64, F64_F32,
};

extern "C" bool AvxPackedConvertFP_(const XmmVal& a, XmmVal& b, CvtOp cvt_op);

void AvxPackedConvertF32(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;

    a.m_I32[0] = 10;
    a.m_I32[1] = -500;
    a.m_I32[2] = 600;
    a.m_I32[3] = -1024;
    AvxPackedConvertFP_(a, b, CvtOp::I32_F32);
    cout << "\nРезультат CvtOp::I32_F32\n";
    cout << "a: " << a.ToStringI32() << '\n';
    cout << "b: " << b.ToStringF32() << '\n';
    a.m_F32[0] = 1.0f / 3.0f;
    a.m_F32[1] = 2.0f / 3.0f;
    a.m_F32[2] = -a.m_F32[0] * 2.0f;
    a.m_F32[3] = -a.m_F32[1] * 2.0f;
    AvxPackedConvertFP_(a, b, CvtOp::F32_I32);
    cout << "\nРезультат CvtOp::F32_I32\n";
    cout << "a: " << a.ToStringF32() << '\n';
    cout << "b: " << b.ToStringI32() << '\n';

    // F32_F64 конвертирует два младших значения 'a'
    a.m_F32[0] = 1.0f / 7.0f;
    a.m_F32[1] = 2.0f / 9.0f;
    a.m_F32[2] = 0;
    a.m_F32[3] = 0;
    AvxPackedConvertFP_(a, b, CvtOp::F32_F64);
    cout << "\Результаты CvtOp::F32_F64\n";
    cout << "a: " << a.ToStringF32() << '\n';
    cout << "b: " << b.ToStringF64() << '\n';
}

void AvxPackedConvertF64(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;

    // I32_F64 два младших двойных слова целого числа 'a'
    a.m_I32[0] = 10;
    a.m_I32[1] = -20;
    a.m_I32[2] = 0;
    a.m_I32[3] = 0;
    AvxPackedConvertFP_(a, b, CvtOp::I32_F64);
    cout << "\nРезультат CvtOp::I32_F64\n";
    cout << "a: " << a.ToStringI32() << '\n';
    cout << "b: " << b.ToStringF64() << '\n';
```

```
    // F64_I32 обнуляет два старших двойных слова 'b'
    a.m_F64[0] = M_PI;
    a.m_F64[1] = M_E;
    AvxPackedConvertFP_(a, b, CvtOp::F64_I32);
    out << "\nРезультат CvtOp::F64_I32\n";
    cout << "a: " << a.ToStringF64() << '\n';
    cout << "b: " << b.ToStringI32() << '\n';

    // F64_F32 обнуляет два старших значения SPFP 'b'
    a.m_F64[0] = M_SQRT2;
    a.m_F64[1] = M_SQRT1_2;
    AvxPackedConvertFP_(a, b, CvtOp::F64_F32);
    cout << "\nResults for CvtOp::F64_F32\n";
    cout << "a: " << a.ToStringF64() << '\n';
    cout << "b: " << b.ToStringF32() << '\n';
}

int main()
{
    AvxPackedConvertF32();
    AvxPackedConvertF64();
    return 0;
}

;-----------------------------------------------
;                  Ch06_03.asm
;-----------------------------------------------

; extern "C" bool AvxPackedConvertFP_(const XmmVal& a, XmmVal& b, CvtOp cvt_op);
;
; Примечание: для этой функции опция компоновщика /LARGEADDRESSAWARE:NO
; должна быть указана в явном виде

        .code
AvxPackedConvertFP_ proc

; Make sure cvt_op is valid
        mov r9d,r8d                     ;r9 = cvt_op (расширен нулями)
        cmp r9,CvtOpTableCount          ;cvt_op имеет допустимое значение?
        jae InvalidCvtOp                ;переход, если cvt_op недопустимо

        mov eax,1                       ;установка возвращаемого кода при допустимом cvt_op
        jmp [CvtOpTable+r9*8]           ;переход к определенному преобразованию

; Преобразование упакованного целого двойного назначения со знаком в упакованное SPFP
        I32_F32:
        vmovdqa xmm0,xmmword ptr [rcx]
        vcvtdq2ps xmm1,xmm0
        vmovaps xmmword ptr [rdx],xmm1
        ret

; Преобразование упакованного значения SPFP в упакованное целое двойное слово со знаком
        F32_I32:
        vmovaps xmm0,xmmword ptr [rcx]
        vcvtps2dq xmm1,xmm0
        vmovdqa xmmword ptr [rdx],xmm1
        ret
```

```
; Преобразование упакованного целого двойного слова со знаком в упакованное DPFP
        I32_F64:
        vmovdqa xmm0,xmmword ptr [rcx]
        vcvtdq2pd xmm1,xmm0
        vmovapd xmmword ptr [rdx],xmm1
        ret

; Преобразование упакованного DPFP в упакованное двусловное целое со знаком
        F64_I32:
        vmovapd xmm0,xmmword ptr [rcx]
        vcvtpd2dq xmm1,xmm0
        vmovdqa xmmword ptr [rdx],xmm1
        ret

; Преобразование упакованного SPFP в упакованное DPFP
        F32_F64:
        vmovaps xmm0,xmmword ptr [rcx]
        vcvtps2pd xmm1,xmm0
        vmovapd xmmword ptr [rdx],xmm1
        ret

; Преобразование упакованного DPFP в упакованное SPFP
        F64_F32:
        vmovapd xmm0,xmmword ptr [rcx]
        vcvtpd2ps xmm1,xmm0
        movaps xmmword ptr [rdx],xmm1
        ret

InvalidCvtOp:
        xor eax,eax                     ;возвращаемый код ошибки при недопустимом cvt_op
        ret

; Порядок значений в этой таблице должен совпадать с порядком значений в перечислении
CvtOp,
; которое объявлено в файле Ch06_03.cpp.

            align 8
CvtOpTable  qword I32_F32, F32_I32
            qword I32_F64, F64_I32
            qword F32_F64, F64_F32
CvtOpTableCount equ ($ - CvtOpTable) / size qword

AvxPackedConvertFP_ endp
        End
```

Код C++ начинается с перечисления `CvtOP`, которое определяет операции преобразования, поддерживаемые функцией языка ассемблера `AvxPackedConvertFP_`. Фактические значения перечислителя в `CvtOp` имеют решающее значение, поскольку код языка ассемблера использует их как индексы в таблице переходов. Функция `AvxPackedConvertF32`, следующая за `CvtOp`, проверяет некоторые тестовые примеры, используя упакованные операнды с плавающей запятой одинарной точности. Точно так же функция `AvxPackedConvertF64` содержит тестовые примеры для упакованных операндов с плавающей запятой двойной точности. Как и в предыдущих примерах этой главы, все объявления переменных

XmmVal в этих функциях используют спецификатор alignas(16) для обеспечения правильного выравнивания.

В конце кода языка ассемблера в листинге 6.3 находится ранее упомянутая таблица переходов. CvtOpTable содержит список меток, определенных в функции AvxPackedConvertFP_. Целью каждой метки является блок короткого кода, который выполняет определенное преобразование. CvtOpTableCount определяет количество элементов в таблице переходов и используется для проверки значения аргумента cvt_op. Директива align 8 указывает ассемблеру выровнять CvtOpTable по границе четверного слова, чтобы избежать невыровненных значений в памяти при обращении к элементам в таблице. Обратите внимание, что CvtOpTable определяется внутри функции языка ассемблера AvxPackedConvertFP_ (т. е. между директивами proc и endp), что означает выделение хранилища для таблицы в разделе .code. Очевидно, что таблица переходов не содержит никаких непосредственно исполняемых команд, и поэтому таблица располагается после команды ret. Это также означает, что таблица переходов доступна только для чтения; процессор будет генерировать исключение при любой попытке записи в таблицу.

Функция на языке ассемблера AvxPackedConvertFP_ начинает свое выполнение с проверки значения аргумента cvt_op. Следующая команда jmp [CvtOpTable+r9*8] передает управление блоку кода, который выполняет фактическое преобразование упакованных данных. Во время выполнения этой команды процессор загружает в регистр RIP содержимое памяти, указанное в [CvxOpTable+r9*8]. В текущем примере регистр R9 содержит cvt_op, и это значение используется как индекс в CvtOpTable.

Блоки кода, выполняющего преобразования в AvxPackedConvertFP_, используют выровненные команды перемещения vmovaps, vmovapd и vmovdqa для передачи упакованных операндов в память и извлечения обратно. Конкретные команды преобразования AVX выполняют запрошенные операции. Например, команды vcvtps2dq и vcvtdq2ps выполняют преобразование между упакованными целочисленными значениями одинарной точности с плавающей запятой и двойными словами со знаком, и наоборот. При использовании с операндами шириной 128 бит эти команды преобразуют четыре значения одновременно. Соответствующие команды двойной точности, vcvtpd2dq и vcvtdq2pd, немного отличаются тем, что преобразуются только два значения из-за разницы в размерах элементов (32 и 64 бита). Команды vcvtps2pd и vcvtpd2ps выполняют свои преобразования аналогичным образом. Обратите внимание, что команды vcvtpd2dq и vcvtpd2ps устанавливают старшие 64 бита операнда назначения в ноль. Все упакованные команды преобразования AVX используют режим округления, указанный в поле управления округлением MXCSR.RC, как описано в главе 4. Режим округления по умолчанию для Visual C++ – округление до ближайшего. Ниже приведен результат выполнения примера Ch06_03:

```
Результат CvtOp::I32_F32
a:              10            -500 |            600           -1024
b:       10.000000     -500.000000 |     600.000000    -1024.000000

Результат CvtOp::F32_I32
```

```
a:         0.333333        0.666667 |      -0.666667       -1.333333
b:                0               1 |            -1               -1

Результат CvtOp::F32_F64
a:         0.142857        0.222222 |       0.000000        0.000000
b:                   0.142857149243 |                 0.222222223878

Результат CvtOp::I32_F64
a:               10             -20 |              0               0
b:                  10.000000000000 |               -20.000000000000

Результат CvtOp::F64_I32
a:                   3.141592653590 |                 2.718281828459
b:                3               3 |              0               0

Результат CvtOp::F64_F32
a:                   1.414213562373 |                 0.707106781187
b:         1.414214        0.707107 |       0.000000        0.000000
```

6.4. Массивы упакованных чисел с плавающей запятой

Вычислительные ресурсы AVX часто используются для ускорения вычислений с использованием массивов значений с плавающей запятой одинарной или двойной точности. В этом разделе вы узнаете, как использовать упакованную арифметику для одновременной обработки нескольких элементов массива с плавающей запятой. Вы также увидите примеры дополнительных команд AVX и узнаете, как выполнять проверки выравнивания операндов в памяти во время выполнения.

6.4.1. Квадратные корни из упакованных чисел с плавающей запятой

В листинге 6.4 приведен код примера `Ch06_04`, который показывает, как выполнить простое упакованное арифметическое вычисление с использованием массива с плавающей запятой одинарной точности. Также объясняется, как выполнить проверку адреса массива во время выполнения, чтобы убедиться, что он правильно выровнен в памяти.

Листинг 6.4. Пример Ch06_04

```
//-----------------------------------------------
//               Ch06_04.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>

using namespace std;

extern "C" bool AvxCalcSqrts_(float* y, const float* x, size_t n);
```

```
void Init(float* x, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 2000};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (float)ui_dist(rng);
}

bool AvxCalcSqrtsCpp(float* y, const float* x, size_t n)
{
    const size_t alignment = 16;

    if (n == 0)
        return false;

    if (((uintptr_t)x % alignment) != 0)
        return false;

    if (((uintptr_t)y % alignment) != 0)
        return false;

    for (size_t i = 0; i < n; i++)
        y[i] = sqrt(x[i]);

    return true;
}

int main()
{
    const size_t n = 19;
    alignas(16) float x[n];
    alignas(16) float y1[n];
    alignas(16) float y2[n];

    Init(x, n, 53);

    bool rc1 = AvxCalcSqrtsCpp(y1, x, n);
    bool rc2 = AvxCalcSqrts_(y2, x, n);

    cout << fixed << setprecision(4);
    cout << "\nРезультаты AvxCalcSqrts\n";

    if (!rc1 || !rc2)
        cout << "Ошибочный возвращаемый код\n";
    else
    {
        const char* sp = " ";

        for (size_t i = 0; i < n; i++)
        {
            cout << "i: " << setw(2) << i << sp;
            cout << "x: " << setw(9) << x[i] << sp;
            cout << "y1: " << setw(9) << y1[i] << sp;
            cout << "y2: " << setw(9) << y2[i] << '\n';
        }
    }
}
```

```
;-----------------------------------------------
;                Ch06_04.asm
;-----------------------------------------------

; extern "C" bool AvxCalcSqrts_(float* y, const float* x, size_t n);

        .code
AvxCalcSqrts_ proc
        xor eax,eax                     ;возвращаемый код ошибки (также смещение массива)

        test r8,r8
        jz Done                         ;перейти, если n равно нулю

        test rcx,0fh
        jnz Done                        ;перейти, если 'y' не выровнен

; Вычисление квадратного корня из упакованного числа
        cmp r8,4
        jb FinalVals ;jump if n < 4
@@:     vsqrtps xmm0,xmmword ptr [rdx+rax]  ;вычисление 4 корней x[i+3:i]
        vmovaps xmmword ptr [rcx+rax],xmm0  ;сохранение результатов y[i+3:i]

        add rax,16                          ;установка смещения на следующий набор
        sub r8,4
        cmp r8,4                            ;осталось ли 4 или больше элементов?
        jae @B                              ;переход, если да

; Вычисление кваратных корней из оставшихся 1-3 значений
FinalVals:
        test r8,r8                          ;остались еще элементы?
        jz SetRC                            ;переход, если кончились

        vsqrtss xmm0,xmm0,real4 ptr [rdx+rax]   ;вычисление sqrt(x[i])
        vmovss real4 ptr [rcx+rax],xmm0         ;сохранение в y[i]
        add rax,4
        dec r8
        jz SetRC

        vsqrtss xmm0,xmm0,real4 ptr [rdx+rax]
        vmovss real4 ptr [rcx+rax],xmm0
        add rax,4
        dec r8
        jz SetRC

        vsqrtss xmm0,xmm0,real4 ptr [rdx+rax]
        vmovss real4 ptr [rcx+rax],xmm0

SetRC:  mov eax,1                       ;возвращаемый код успешного выполнения

Done:   ret
AvxCalcSqrts_ endp
        end
```

Код C++ в листинге 6.4 содержит функцию `AvxCalcSqrtsCpp`, которая вычисляет `y[i] = sqrt(x[i])`. Перед выполнением любых требуемых вычислений происходит проверка аргумента размера массива `n`, чтобы убедиться, что он не равен нулю.

Также выполняется проверка указателей y и x, чтобы убедиться, что соответствующие массивы правильно выровнены по 16-байтовой границе. Массив выравнивается по 16-байтовой границе, если его адрес делится на 16. Функция возвращает код ошибки, если какая-либо из этих проверок завершилась неудачно.

Функция языка ассемблера AvxCalcSqrts_ имитирует функциональность своего аналога C++. Команды test r8,r8 и jz Done гарантируют, что количество элементов массива n больше нуля. Следующая команда test rcx,0fh проверяет массив y на соответствие 16-байтовой границе. Напомним, что тестовая команда выполняет поразрядное И для двух своих операндов и устанавливает флаги состояния в RFLAGS в соответствии с результатом (фактический результат побитового И отбрасывается). Если команда test rcx,0fh дает ненулевое значение, массив y не выровнен по 16-байтовой границе, и функция завершается без выполнения каких-либо вычислений. Аналогичный тест используется для проверки правильности выравнивания массива x.

Цикл обработки использует команду vsqrtps для вычисления требуемых квадратных корней. При применении с операндами шириной 128 бит эта команда одновременно вычисляет четыре квадратных корня с плавающей запятой одинарной точности. Использование операндов шириной 128 бит означает, что цикл обработки не может выполнить команду vsqrtps, если для обработки осталось менее четырех значений элементов. Перед выполнением любых вычислений с использованием vsqrtps значение в регистре R8 проверяется, чтобы убедиться, что оно больше или равно четырем. Если R8 меньше четырех, цикл обработки пропускается. Цикл обработки использует команду vsqrtps xmm0,xmmword ptr [rdx+rax] для вычисления квадратных корней из четырех значений с плавающей запятой одинарной точности, расположенных по адресу памяти, указанному исходным операндом. Затем он сохраняет вычисленные квадратные корни в регистре XMM0. Команда vmovaps xmmword ptr [rcx+rax],xmm0 сохраняет четыре вычисленных квадратных корня в y. Выполнение команд vsqrtps и vmovaps продолжается до тех пор, пока количество элементов, остающихся для обработки, не станет меньше четырех.

После выполнения цикла обработки блок кода, начинающийся с метки FinalVals, вычисляет квадратные корни для нескольких оставшихся значений массива x. Обратите внимание, что скалярные команды AVX vsqrtss и vmovss выполняют эти окончательные (одно, два или три) вычисления. Ниже показан результат выполнения кода примера Ch06_04:

```
Результаты AvxCalcSqrts
i:   0   x:  1354.0000   y1:  36.7967   y2:  36.7967
i:   1   x:   494.0000   y1:  22.2261   y2:  22.2261
i:   2   x:  1638.0000   y1:  40.4722   y2:  40.4722
i:   3   x:   278.0000   y1:  16.6733   y2:  16.6733
i:   4   x:  1004.0000   y1:  31.6860   y2:  31.6860
i:   5   x:   318.0000   y1:  17.8326   y2:  17.8326
i:   6   x:  1735.0000   y1:  41.6533   y2:  41.6533
i:   7   x:  1221.0000   y1:  34.9428   y2:  34.9428
i:   8   x:   544.0000   y1:  23.3238   y2:  23.3238
i:   9   x:  1568.0000   y1:  39.5980   y2:  39.5980
i:  10   x:  1633.0000   y1:  40.4104   y2:  40.4104
i:  11   x:  1577.0000   y1:  39.7115   y2:  39.7115
```

```
i:  12   x:  1659.0000   y1:  40.7308   y2:  40.7308
i:  13   x:  1565.0000   y1:  39.5601   y2:  39.5601
i:  14   x:    74.0000   y1:   8.6023   y2:   8.6023
i:  15   x:  1195.0000   y1:  34.5688   y2:  34.5688
i:  16   x:   406.0000   y1:  20.1494   y2:  20.1494
i:  17   x:   483.0000   y1:  21.9773   y2:  21.9773
i:  18   x:  1307.0000   y1:  36.1525   y2:  36.1525
```

Исходный код в листинге 6.4 можно легко модифицировать для обработки значений с двойной точностью вместо значений с плавающей запятой одинарной точности. В коде C++ достаточно заменить все переменные типа `float` на переменные типа `double`. В коде на языке ассемблера вместо `vsqrtps` и `vmovaps` необходимо использовать команды `vsqrtpd` и `vmovapd`. Переменные счетчика в `AvxCalcSqrts_` также должны быть изменены для обработки двух значений с плавающей запятой с двойной точностью вместо четырех значений с плавающей запятой с одинарной точностью на итерацию.

6.4.2. Поиск минимального и максимального значений массива упакованных значений с плавающей запятой

В листинге 6.5 показан исходный код примера `Ch06_05`. В этом примере показано, как вычислить минимальное и максимальное значения массива упакованных чисел с плавающей запятой одинарной точности с использованием команд AVX.

Листинг 6.5. Пример Ch06_05

```
//-----------------------------------------------
//               Ch06_05.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <limits>
#include "AlignedMem.h"

using namespace std;

extern "C" float g_MinValInit = numeric_limits<float>::max();
extern "C" float g_MaxValInit = -numeric_limits<float>::max();

extern "C" bool CalcArrayMinMaxF32_(float* min_val, float* max_val, const float* x, size_t
n);

void Init(float* x, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 10000};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (float)ui_dist(rng);
}
```

```
bool CalcArrayMinMaxF32Cpp(float* min_val, float* max_val, const float* x, size_t n)
{
    // Проверка правильности выравнивания x
    if (!AlignedMem::IsAligned(x, 16))
        return false;

    // Нахождение минимального и максимального значений
    float min_val_temp = g_MinValInit;
    float max_val_temp = g_MaxValInit;

    if (n > 0)
    {
        for (size_t i = 0; i < n; i++)
        {
            if (x[i] < min_val_temp)
                min_val_temp = x[i];
                if (x[i] > max_val_temp)
                    max_val_temp = x[i];
        }
    }

    *min_val = min_val_temp;
    *max_val = max_val_temp;
    return true;
}

int main()
{
    const size_t n = 31;
    alignas(16) float x[n];

    Init(x, n, 73);

    float min_val1, max_val1;
    float min_val2, max_val2;

    CalcArrayMinMaxF32Cpp(&min_val1, &max_val1, x, n);
    CalcArrayMinMaxF32_(&min_val2, &max_val2, x, n);

    cout << fixed << setprecision(1);
    cout << "-------------- Массив x --------------\n";

    for (size_t i = 0; i < n; i++)
    {
        cout << "x[" << setw(2) << i << "]: " << setw(9) << x[i];

        if (i & 1)
            cout << '\n';
        else
            cout << " ";
    }

    cout << '\n';

    cout << "\nРезультаты CalcArrayMinMaxF32Cpp\n";
    cout << " min_val = " << setw(9) << min_val1 << ", ";
    cout << " max_val = " << setw(9) << max_val1 << '\n';
```

```
    cout << "\nРезультаты CalcArrayMinMaxF32_\n";
    cout << " min_val = " << setw(9) << min_val2 << ", ";
    cout << " max_val = " << setw(9) << max_val2 << '\n';

    return 0;
}

;-----------------------------------------------
;                Ch06_05.asm
;-----------------------------------------------

        extern g_MinValInit:real4
        extern g_MaxValInit:real4

; extern "C" bool CalcArrayMinMaxF32_(float* min_val, float* max_val, const float* x,
size_t n)

        .code
CalcArrayMinMaxF32_ proc
; Проверка аргументов
        xor eax,eax                     ;возвращаемый код ошибки

        test r8,0fh                     ;выровнен ли x по 16 байтам?
        jnz Done                        ;переход, если нет

        vbroadcastss xmm4,real4 ptr [g_MinValInit] ;xmm4 = min
        vbroadcastss xmm5,real4 ptr [g_MaxValInit] ;xmm5 = max

        cmp r9,4
        jb FinalVals                    ;переход, если n < 4

; Главный цикл
@@:     vmovaps xmm0,xmmword ptr [r8]   ;загрузка следующего набора элементов
        vminps xmm4,xmm4,xmm0           ;обновление упакованного минимального значения
        vmaxps xmm5,xmm5,xmm0           ;обновление упакованного максимального значения

        add r8,16
        sub r9,4
        cmp r9,4
        jae @B

; Обработка оставшихся 1-3 значений входного массива
FinalVals:
        test r9,r9
        jz SaveResults

        vminss xmm4,xmm4,real4 ptr [r8] ;обновление упакованного минимального значения
        vmaxss xmm5,xmm5,real4 ptr [r8] ;обновление упакованного максимального значения
        dec r9
        jz SaveResults

        vminss xmm4,xmm4,real4 ptr [r8+4]
        vmaxss xmm5,xmm5,real4 ptr [r8+4]
        dec r9
        jz SaveResults

        vminss xmm4,xmm4,real4 ptr [r8+8]
        vmaxss xmm5,xmm5,real4 ptr [r8+8]

; Вычисление и сохранение оставшихся значений min и max
SaveResults:
```

```
        vshufps xmm0,xmm4,xmm4,00001110b    ;xmm0[63:0] = xmm4[128:64]
        vminps xmm1,xmm0,xmm4               ;xmm1[63:0] оставшиеся 2 значения
        vshufps xmm2,xmm1,xmm1,00000001b    ;xmm2[31:0] = xmm1[63:32]
        vminps xmm3,xmm2,xmm1               ;xmm3[31:0] последнее значение
        vmovss real4 ptr [rcx],xmm3         ;сохранение минимального значения

        vshufps xmm0,xmm5,xmm5,00001110b
        vmaxps xmm1,xmm0,xmm5
        vshufps xmm2,xmm1,xmm1,00000001b
        vmaxps xmm3,xmm2,xmm1
        vmovss real4 ptr [rdx],xmm3     ;сохранение максимального значения

        mov eax,1                       ;возвращаемый код успешного выполнения
Done:   ret
CalcArrayMinMaxF32_ endp
        end
```

Структура исходного кода C++, показанная в листинге 6.5, аналогична предыдущему примеру с массивом. Функция `CalcArrayMinMaxF32Cpp` использует простой цикл `for` для определения минимального и максимального значений массива. Перед циклом `for` шаблонная функция `AlignedMem::IsAligned` проверяет правильность выравнивания исходного массива `x`. Вы узнаете больше о классе `AlignedMem` в главе 7. Начальные минимальное и максимальное значения получаются из глобальных переменных `g_MinValInit` и `g_MaxValInit`, которые были инициализированы с помощью константы шаблона C++ `numeric_limits<float>::max()`. Здесь используются глобальные переменные, чтобы гарантировать, что функции `CalcArrayMinMaxF32Cpp` и `CalcArrayMinMaxF32_` применяют одни и те же начальные значения.

При входе в функцию на языке ассемблера `CalcArrayMinMaxF32_` массив `x` проверяется на правильное выравнивание. Если массив `x` правильно выровнен, команда `vbroadcastss xmm4,real4 ptr [g_MinValInit]` инициализирует все четыре элемента с плавающей запятой одинарной точности в регистре XMM4 значением `g_MinValInit`. Последующая команда `vbroadcastss xmm5,real4 ptr [g_MaxValInit]` транслирует `g_MaxValInit` на все четыре позиции элементов в регистре XMM5.

Как и в предыдущем примере, цикл обработки в `CalcArrayMinMaxF32_` проверяет четыре элемента массива на каждой итерации. Команды `vminps xmm4,xmm4,xmm0` и `vmaxps xmm5,xmm5,xmm0` хранят промежуточные упакованные минимальное и максимальное значения в регистрах XMM4 и XMM5 соответственно. Цикл обработки продолжается до тех пор, пока не останется менее четырех элементов. Последние элементы в массиве проверяются с помощью скалярных команд `vminss` и `vmaxss`.

После выполнения команды `vmaxss`, которая находится непосредственно над меткой `SaveResults`, регистр XMM4 содержит четыре значения с плавающей запятой одинарной точности, и одно из этих значений является минимальным для массива `x`. Затем для определения окончательного минимального значения используется серия команд `vshufps` (упакованные значения с плавающей запятой одинарной точности с чередованием в случайном порядке) и `vminps`. Команда `vshufps xmm0,xmm4,xmm4,00001110b` копирует два старших элемента с плавающей запятой в регистре XMM4 в позиции младших элементов в XMM0 (т. е.

XMM0[63:0] = XMM4[127:64]). Эта команда использует битовые значения своего непосредственного операнда в качестве индексов для выбора элементов, подлежащих копированию.

Непосредственный операнд, используемый инструкцией `vshufps`, нуждается в отдельном пояснении. В текущем примере биты 1:0 (10b) непосредственного операнда инструктируют процессор копировать элемент с плавающей запятой одиночной точности #2 (XMM4[95:64]) из первого исходного операнда в позицию элемента #0 (XMM0[31:0]) операнда-адресата. Биты 3:2 (11b) непосредственного операнда также инструктируют процессор скопировать элемент #3 (XMM4[127:64]) первого исходного операнда в позицию элемента #1 (XMM0[63:32]) целевого операнда. Биты 7:6 и 5:4 непосредственного операнда могут использоваться для копирования элементов из второго исходного операнда в позиции элементов # 2 (XMM0 [95:64]) и # 3 (XMM0 [127:96]) места назначения, но в данном примере они не нужны. За командой `vshufps` следует команда `vminps xmm1,xmm0,xmm4`, которая дает два последних минимальных значения в XMM1 [63:32] и XMM1 [31:0]. Затем используется другая последовательность команд `vshufps` и `vminps` для извлечения окончательного минимального значения.

Рисунок 6.3 более подробно иллюстрирует этот процесс сокращения.

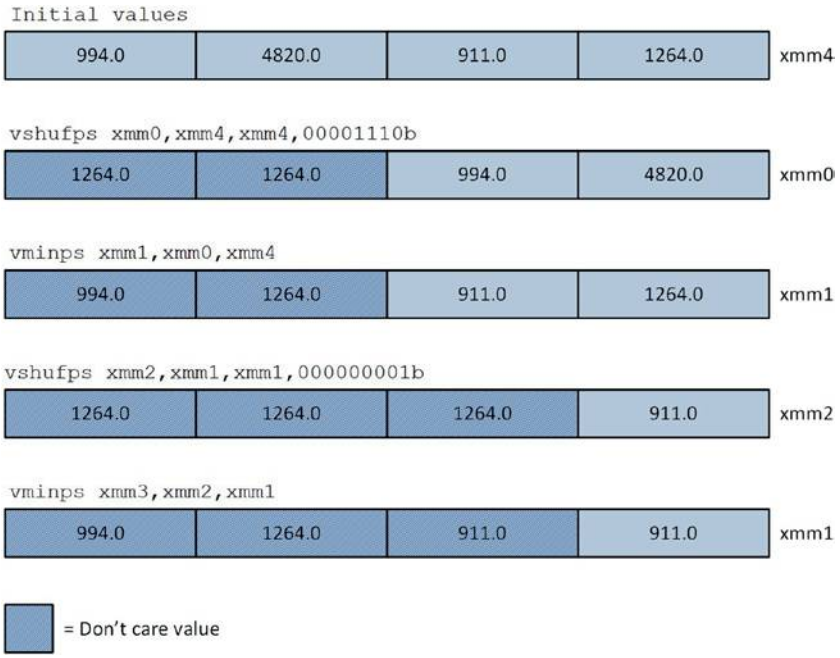


Рис. 6.3. Сокращение упакованных значений с использованием команд `vshufps` и `vminps`

После вычисления минимального значения массива аналогичная серия команд `vshufps` и `vmaxps` определяет максимальное значение, используя ту же технику сокращения. Далее приведены результаты выполнения кода примера `Ch06_05`:

```
-------------- Массив x --------------
x[ 0]:    2183.0    x[ 1]:     4547.0
x[ 2]:    9279.0    x[ 3]:     7291.0
x[ 4]:    5105.0    x[ 5]:     6505.0
x[ 6]:    4820.0    x[ 7]:      994.0
x[ 8]:    1559.0    x[ 9]:     3867.0
x[10]:    7272.0    x[11]:     9698.0
x[12]:    6181.0    x[13]:     4742.0
x[14]:    7279.0    x[15]:     1224.0
x[16]:    4840.0    x[17]:     8453.0
x[18]:    6876.0    x[19]:     1786.0
x[20]:    4022.0    x[21]:      911.0
x[22]:    6676.0    x[23]:     2979.0
x[24]:    4431.0    x[25]:     6133.0
x[26]:    7093.0    x[27]:     9892.0
x[28]:    9622.0    x[29]:     5058.0
x[30]:    1264.0

Результаты CalcArrayMinMaxF32Cpp
min_val =       911.0,    max_val =     9892.0

Результаты CalcArrayMinMaxF32_
min_val =       911.0,    max_val =     9892.0
```

6.4.3. Наименьшие квадраты упакованных чисел с плавающей запятой

Пример исходного кода Ch06_06 подробно иллюстрирует вычисление линейной регрессии методом наименьших квадратов с использованием арифметики для упакованных чисел с плавающей запятой двойной точности. В листинге 6.6 показан исходный код примера Ch06_06 на C++ и на языке асссемблера x86.

Листинг 6.6. Пример Ch06_06

```
//-----------------------------------------------
//               Ch06_06.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cstddef>
#include "AlignedMem.h"

using namespace std;

extern "C" double LsEpsilon = 1.0e-12;
extern "C" bool AvxCalcLeastSquares_(const double* x, const double* y, int n, double* m,
double* b);

bool AvxCalcLeastSquaresCpp(const double* x, const double* y, int n, double* m, double* b)
{
    if (n < 2)
        return false;
```

```
    if (!AlignedMem::IsAligned(x, 16) || !AlignedMem::IsAligned(y, 16))
        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 main()
{
    const int n = 11;
    alignas(16) double x[n] = {10, 13, 17, 19, 23, 7, 35, 51, 89, 92, 99};
    alignas(16) double y[n] = {1.2, 1.1, 1.8, 2.2, 1.9, 0.5, 3.1, 5.5, 8.4, 9.7, 10.4};
    double m1 = 0, m2 = 0;
    double b1 = 0, b2 = 0;

    bool rc1 = AvxCalcLeastSquaresCpp(x, y, n, &m1, &b1);
    bool rc2 = AvxCalcLeastSquares_(x, y, n, &m2, &b2);

    cout << fixed << setprecision(8);

    cout << "\nРезультаты AvxCalcLeastSquaresCpp\n";
    cout << " результат: " << setw(12) << boolalpha << rc1 << '\n';
    cout << " наклон: " << setw(12) << m1 << '\n';
    cout << " отсечение: " << setw(12) << b1 << '\n';

    cout << "\nРезультаты from AvxCalcLeastSquares_\n";
    cout << " результат: " << setw(12) << boolalpha << rc2 << '\n';
    cout << " наклон: " << setw(12) << m2 << '\n';
    cout << " отсечение: " << setw(12) << b2 << '\n';

    return 0;
}

;-----------------------------------------------
;                Ch06_06.asm
;-----------------------------------------------
```

```
        include <MacrosX86-64-AVX.asmh>

        extern LsEpsilon:real8          ;global value defined in C++ file

; extern "C" bool AvxCalcLeastSquares_(const double* x, const double* y, int n, double* m,
double* b);
;
; Возвращает 0 = ошибка (недопустимое n или выравнивание массива), 1 = успех

        .const
        align 16
AbsMaskF64 qword 7fffffffffffffffh, 7fffffffffffffffh ;маска абс. значения типа DPFP

        .code
AvxCalcLeastSquares_ proc frame
        _CreateFrame LS_,0,48,rbx
        _SaveXmmRegs xmm6,xmm7,xmm8
        _EndProlog

; Проверка аргументов
        xor eax,eax                     ;возвращаемый код ошибки
        cmp r8d,2
        jl Done                         ;переход, если n < 2
        test rcx,0fh
        jnz Done                        ;переход, если x не выровнен по 16 байтам
        test rdx,0fh
        jnz Done                        ;переход, если y не выровнен по 16 байтам

; Инициализация
        vcvtsi2sd xmm3,xmm3,r8d         ;xmm3 = n
        mov eax,r8d
        and r8d,0fffffffeh              ;rd8 = n / 2 * 2
        and eax,1                       ;eax = n % 2
        vxorpd xmm4,xmm4,xmm4           ;sum_x (оба слова)
        vxorpd xmm5,xmm5,xmm5           ;sum_y (оба слова)
        vxorpd xmm6,xmm6,xmm6           ;sum_xx (оба слова)
        vxorpd xmm7,xmm7,xmm7           ;sum_xy (оба слова)
        xor ebx,ebx                     ;rbx = смещение массива
        mov r10,[rbp+LS_OffsetStackArgs] ;r10 = b

; Вычисление суммы переменных. За итерацию обрабатывается два значения.
@@:     vmovapd xmm0,xmmword ptr [rcx+rbx] ;следующие два значения x
        vmovapd xmm1,xmmword ptr [rdx+rbx] ;следующие два значения y

        vaddpd xmm4,xmm4,xmm0           ;обновление sum_x
        vaddpd xmm5,xmm5,xmm1           ;обновление sum_y

        vmulpd xmm2,xmm0,xmm0           ;вычисление x * x
        vaddpd xmm6,xmm6,xmm2           ;обновление sum_xx

        vmulpd xmm2,xmm0,xmm1           ;вычисление x * y
        vaddpd xmm7,xmm7,xmm2           ;обновление sum_xy

        add rbx,16                      ;rbx = следующее смещение
        sub r8d,2                       ;выравнивание счетчика
        jnz @B                          ;повтор до завершения
```

```
; Обновление суммы финальными значениями x, y, если 'n' нечетное
        or eax,eax
        jz CalcFinalSums                ;переход, если n четное
        vmovsd xmm0,real8 ptr [rcx+rbx] ;присвоение окончательного x
        vmovsd xmm1,real8 ptr [rdx+rbx] ;присвоение окончательного y

        vaddsd xmm4,xmm4,xmm0           ;обновление sum_x
        vaddsd xmm5,xmm5,xmm1           ;обновление sum_y

        vmulsd xmm2,xmm0,xmm0           ;вычисление x * x
        vaddsd xmm6,xmm6,xmm2           ;обновление sum_xx

        vmulsd xmm2,xmm0,xmm1           ;вычисление x * y
        vaddsd xmm7,xmm7,xmm2           ;обновление sum_xy

; Вычисление окончательных sum_x, sum_y, sum_xx, sum_xy
CalcFinalSums:
        vhaddpd xmm4,xmm4,xmm4          ;xmm4[63:0] = оконч. sum_x
        vhaddpd xmm5,xmm5,xmm5          ;xmm5[63:0] = оконч. sum_y
        vhaddpd xmm6,xmm6,xmm6          ;xmm6[63:0] = оконч. sum_xx
        vhaddpd xmm7,xmm7,xmm7          ;xmm7[63:0] = оконч. sum_xy

; Вычисление знаменателя и проверка его значения на допустимость
; denom = n * sum_xx - sum_x * sum_x
        vmulsd xmm0,xmm3,xmm6           ;n * sum_xx
        vmulsd xmm1,xmm4,xmm4           ;sum_x * sum_x
        vsubsd xmm2,xmm0,xmm1           ;denom
        vandpd xmm8,xmm2,xmmword ptr [AbsMaskF64] ;fabs(denom)
        vcomisd xmm8,real8 ptr [LsEpsilon]
        jb BadDen                       ;переход, если denom < fabs(denom)

; Вычисление и сохранение наклона
; slope = (n * sum_xy - sum_x * sum_y) / denom
        vmulsd xmm0,xmm3,xmm7           ;n * sum_xy
        vmulsd xmm1,xmm4,xmm5           ;sum_x * sum_y
        vsubsd xmm2,xmm0,xmm1           ;числитель наклона
        vdivsd xmm3,xmm2,xmm8           ;итоговый наклон
        vmovsd real8 ptr [r9],xmm3      ;сохранение наклона

; Вычисление и сохранение пересечения
; intercept = (sum_xx * sum_y - sum_x * sum_xy) / denom
        vmulsd xmm0,xmm6,xmm5           ;sum_xx * sum_y
        vmulsd xmm1,xmm4,xmm7           ;sum_x * sum_xy
        vsubsd xmm2,xmm0,xmm1           ;числитель пересечения
        vdivsd xmm3,xmm2,xmm8           ;итоговое пересечение
        vmovsd real8 ptr [r10],xmm3     ;сохранение пересечения
        mov eax,1                       ;код успешного выполнения
        jmp Done

; Обнаружен некорректный знаменатель, установить m и b в 0.0
BadDen: vxorpd xmm0,xmm0,xmm0
        vmovsd real8 ptr [r9],xmm0      ;*m = 0.0
        vmovsd real8 ptr [r10],xmm0     ;*b = 0.0
        xor eax,eax                     ;возвращаемый код ошибки

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame rbx
        ret
AvxCalcLeastSquares_ endp
        end
```

Простая линейная регрессия – это статистический метод, моделирующий линейную зависимость между двумя переменными. Один из популярных методов простой линейной регрессии называется *подгонкой методом наименьших квадратов* (least squares fitting), который использует набор выборочных точек данных для определения наилучшего соответствия или оптимальной кривой между двумя переменными. В случае простой модели линейной регрессии кривая представляет собой прямую линию, уравнение которой имеет вид $y = mx + b$. В этом уравнении x обозначает независимую переменную, y представляет зависимую (или измеряемую) переменную, m – наклон линии, а b – точку пересечения оси y. Наклон и точка пересечения линии наименьших квадратов определяются с помощью серии вычислений, которые минимизируют сумму квадратов отклонений между линией и точками данных выборки. После вычисления наклона и точки пересечения линия наименьших квадратов часто используется для прогнозирования неизвестного значения y с использованием известного значения x. Если вы хотите узнать больше о теории простой линейной регрессии и аппроксимации методом наименьших квадратов, обратитесь к ссылкам, перечисленным в приложении.

В программе-примере `Ch06_06` для вычисления наклона и точки пересечения по методу наименьших квадратов используются следующие уравнения:

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

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

На первый взгляд уравнения наклона и пересечения могут показаться немного сложными. Однако при ближайшем рассмотрении становятся очевидными несколько упрощений. Во-первых, знаменатели наклона и точки пересечения совпадают, следовательно, это значение необходимо вычислить только один раз. Во-вторых, необходимо вычислить только четыре простых суммируемых количества (или суммирующие переменные), как показано в следующих уравнениях:

$$sum\_x = \sum_i x_i;$$

$$sum\_y = \sum_i y_i;$$

$$sum\_xy = \sum_i x_i y_i;$$

$$sum\_xx = \sum_i x_i^2.$$

После вычисления сумм переменных можно легко получить наклон по методу наименьших квадратов и точку пересечения с использованием прямого умножения, вычитания и деления.

Исходный код C++ в листинге 6.6 содержит функцию AvxCalcLeastSquaresCpp, которая вычисляет наклон и точку пересечения по методу наименьших квадратов для целей сравнения с ассемблерной функцией. AvxCalcLeastSquaresCpp использует AlignedMem::IsAligned() для проверки правильности выравнивания двух массивов данных. Класс C++ AlignedMem (исходный код не показан, но включен в пакет загрузки) содержит несколько простых функций-членов, которые выполняют управление выровненной памятью и проверку. Эти функции были включены в класс C++ для облегчения повторного использования кода в данном примере и последующих главах. Функция main в C++ определяет пару тестовых массивов с именами x и y, используя спецификатор C++ alignas(16), который инструктирует компилятор выровнять каждый из этих массивов по 16-байтовой границе. Остальная часть main содержит код, который выполняет реализации алгоритма наименьших квадратов как на языке ассемблера C++, так и на x86, и передает результаты в cout.

Код языка ассемблера x86-64 для функции AvxCalcLeastSquares_ начинается с сохранения энергонезависимых регистров RBX, XMM6, XMM7 и XMM8 с использованием макросов _CreateFrame и _SaveXmmRegs. Затем значение аргумента n проверяется на размер, а указатели массива x и y проверяются на правильное выравнивание. После проверки аргументов функции выполняется серия инициализаций. Команда vcvtsi2sd xmm3,xmm3,r8d преобразует значение n в число с плавающей запятой двойной точности для дальнейшего использования. Затем значение n в R8D округляется до ближайшего четного числа с помощью команды and r8d, 0fffffffeh, а EAX устанавливается в ноль или единицу в зависимости от того, является ли исходное значение n четным или нечетным. Эти настройки выполняются для обеспечения правильной обработки массивов x и y с использованием упакованной арифметики.

Напомню из сказанного ранее в этом разделе, что для вычисления наклона и точки пересечения линии регрессии методом наименьших квадратов необходимо вычислить четыре промежуточных значения суммы: sum_x, sum_y, sum_xx и sum_xy. Цикл суммирования, который вычисляет эти значения в AvxCalcLeastSquares_, использует арифметику упакованных чисел с плавающей запятой двойной точности. Это означает, что AvxCalcLeastSquares_ может обрабатывать два элемента из массивов x и y во время каждой итерации цикла, что вдвое уменьшает количество требуемых итераций. Значения суммы для элементов массива с четными индексами вычисляются с использованием четверных слов младшего разряда XMM4–XMM7, в то время как четверные слова старшего разряда используются для вычисления значений суммы элементов массива с нечетными индексами.

Перед входом в цикл суммирования каждый регистр значения суммы инициализируется нулем с помощью команды vxorpd. В верхней части цикла суммирования команда vmovapd xmm0,xmmword ptr [rcx+rbx] копирует x[i] и x[i+1] в младшие и старшие четверные слова XMM0 соответственно. Следующая команда, vmovapd xmm1,xmmword ptr [rdx+rbx], загружает y[i] и y[i+1] в младшие и старшие четверные слова XMM1. Серия команд vaddpd и vmulpd обновляет значения упакованной суммы, которые поддерживаются в XMM4–XMM7. Затем регистр смещения массива RBX увеличивается на 16 (или на размер двух значений с плавающей запятой двойной точности), а значение счетчика в R8D корректируется перед следующей итерацией цикла суммирования. После за-

вершения цикла суммирования выполняется проверка, чтобы определить, было ли исходное значение n нечетным. Если условие выполняется, последний элемент массива x и массива y должен быть добавлен к упакованным значениям суммы. Эту операцию выполняют скалярные команды AVX vaddsd и vmulsd.

После вычисления значений упакованной суммы серия команд vhaddpd (горизонтальное сложение упакованных чисел двойной точности с плавающей запятой) вычисляет окончательные значения sum_x, sum_y, sum_xx и sum_xy. Каждая команда vhaddpd DesOp,SrcOp1,SrcOp2 вычисляет DesOp[63:0] = SrcOp1[127:64]+SrcOp1[63:0] и DesOp[127:64] = SrcOp2[127:64]+SrcOp2[63:0] (рис. 4.14). После выполнения команд vhaddpd младшие четверные слова регистров XMM4–XMM7 содержат значения окончательной суммы. Четверные слова старших разрядов этих регистров также содержат значения окончательной суммы, но это является следствием использования одного и того же регистра для обоих исходных операндов. Затем вычисляется значение denom и выполняется проверка, чтобы убедиться, что его абсолютное значение больше или равно LsEpsilon; абсолютное значение меньше LsEpsilon считается слишком близким к нулю, чтобы считаться действительным. Обратите внимание, что для вычисления fabs(denom) используется команда vandpd. После проверки номинала значения наклона и точки пересечения вычисляются с использованием простой скалярной арифметики. Вот результат выполнения примера исходного кода Ch06_06:

```
Результат AvxCalcLeastSquaresCpp
  результат:          true
  наклон:       0.10324631
  пересечение: -0.10700632

Результат AvxCalcLeastSquares_
  результат:          true
  наклон:       0.10324631
  пересечение: -0.10700632
```

6.5. Упакованные матрицы значений с плавающей запятой

Прикладные программы, например для обработки компьютерной графики и автоматизированного проектирования, часто широко используют матрицы. Например, программы трехмерной (3D) компьютерной графики обычно применяют матрицы для выполнения общих преобразований, таких как перенос, масштабирование и поворот. При использовании однородных координат каждая из этих операций может быть эффективно представлена с помощью одной матрицы 4×4. Множественные преобразования можно реализовать объединением серии отдельных преобразований матриц в единую матрицу преобразования с использованием умножения матриц. Эта комбинированная матрица обычно применяется к массиву вершин объекта, который определяет 3D-модель. Для программного обеспечения трехмерной компьютерной графики критически важно как можно быстрее выполнять такие операции, как умножение «матрица–матрица» и «матрица–вектор», поскольку трехмерная модель может содержать тысячи или даже миллионы вершин объекта.

В этом разделе вы узнаете, как выполнять транспонирование и умножение матриц с использованием матриц 4×4 и набора команд AVX. Вы также узнаете больше о макросах на языке ассемблера, о том, как писать код макросов, и о некоторых простых методах тестирования быстродействия алгоритмов.

6.5.1. Транспонирование матрицы

Транспонирование матрицы выполняется путем перестановки ее строк и столбцов. Более формально, если **A** является матрицей размера $m \times n$, транспонированная матрица **A** (обозначенная здесь как **B**) является матрицей $n \times m$, где $b(i, j) = a(j, i)$. На рис. 6.4 показано транспонирование матрицы 4×4.

Матрица A

$$A = \begin{bmatrix} 2 & 7 & 8 & 3 \\ 11 & 14 & 16 & 10 \\ 24 & 21 & 27 & 29 \\ 31 & 34 & 38 & 33 \end{bmatrix}$$

Транспозиция матрицы A

$$B = \begin{bmatrix} 2 & 11 & 24 & 31 \\ 7 & 14 & 21 & 34 \\ 8 & 16 & 27 & 38 \\ 3 & 10 & 29 & 33 \end{bmatrix}$$

Рис. 6.4. Транспонирование матрицы 4×4

В листинге 6.7 показан исходный код примера `Ch06_07`, который демонстрирует, как транспонировать матрицу значений с плавающей запятой одинарной точности с размерностью 4×4.

Листинг 6.7. Пример Ch06_07

```
//-----------------------------------------------
//                 Ch06_07.h
//-----------------------------------------------

#pragma once

// Ch06_07_.asm
extern "C" void AvxMat4x4TransposeF32_(float* m_des, const float* m_src);

// Ch06_07_BM.cpp
extern void AvxMat4x4TransposeF32_BM(void);

//-----------------------------------------------
//               Ch06_07.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include "Ch06_07.h"
#include "Matrix.h"

using namespace std;

void AvxMat4x4TransposeF32(Matrix<float>& m_src)
{
    const size_t nr = 4;
    const size_t nc = 4;
    Matrix<float> m_des1(nr ,nc);
    Matrix<float> m_des2(nr ,nc);
```

```
    Matrix<float>::Transpose(m_des1, m_src);
    AvxMat4x4TransposeF32_(m_des2.Data(), m_src.Data());

    cout << fixed << setprecision(1);
    m_src.SetOstream(12, " ");
    m_des1.SetOstream(12, " ");
    m_des2.SetOstream(12, " ");

    cout << "Результаты AvxMat4x4TransposeF32\n";
    cout << "Матрица m_src \n" << m_src << '\n';
    cout << "Матрица m_des1\n" << m_des1 << '\n';
    cout << "Матрица m_des2\n" << m_des2 << '\n';

    if (m_des1 != m_des2)
        cout << "\nОшибка сравнения матриц - AvxMat4x4TransposeF32\n";
}

int main()
{
    const size_t nr = 4;
    const size_t nc = 4;
    Matrix<float> m_src(nr ,nc);

    const float src_row0[] = { 2, 7, 8, 3 };
    const float src_row1[] = { 11, 14, 16, 10 };
    const float src_row2[] = { 24, 21, 27, 29 };
    const float src_row3[] = { 31, 34, 38, 33 };

    m_src.SetRow(0, src_row0);
    m_src.SetRow(1, src_row1);
    m_src.SetRow(2, src_row2);
    m_src.SetRow(3, src_row3);

    AvxMat4x4TransposeF32(m_src);
    AvxMat4x4TransposeF32_BM();
    return 0;
}

;-----------------------------------------------
;               Ch06_07.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; _Mat4x4TransposeF32 macro
;
; Описание: этот макрос транспонирует матрицу 4×4 чисел одинарной
; точности с плавающей запятой
;
; Исходная матрица      Конечная матрица
; ------------------------------------
; 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
```

```
_Mat4x4TransposeF32 macro
        vunpcklps xmm6,xmm0,xmm1        ;xmm6 = b1 a1 b0 a0
        vunpckhps xmm0,xmm0,xmm1        ;xmm0 = b3 a3 b2 a2
        vunpcklps xmm7,xmm2,xmm3        ;xmm7 = d1 c1 d0 c0
        vunpckhps xmm1,xmm2,xmm3        ;xmm1 = d3 c3 d2 c2

        vmovlhps xmm4,xmm6,xmm7         ;xmm4 = d0 c0 b0 a0
        vmovhlps xmm5,xmm7,xmm6         ;xmm5 = d1 c1 b1 a1
        vmovlhps xmm6,xmm0,xmm1         ;xmm6 = d2 c2 b2 a2
        vmovhlps xmm7,xmm1,xmm0         ;xmm7 = d3 c3 b2 a3
        endm

; extern "C" void AvxMat4x4TransposeF32_(float* m_des, const float* m_src)

        .code
AvxMat4x4TransposeF32_ proc frame
        _CreateFrame MT_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Транспонирование матрицы m_src1
        vmovaps xmm0,[rdx]              ;xmm0 = m_src.row_0
        vmovaps xmm1,[rdx+16]           ;xmm1 = m_src.row_1
        vmovaps xmm2,[rdx+32]           ;xmm2 = m_src.row_2
        vmovaps xmm3,[rdx+48]           ;xmm3 = m_src.row_3

_Mat4x4TransposeF32

        vmovaps [rcx],xmm4              ;сохранение m_des.row_0
        vmovaps [rcx+16],xmm5           ;сохранение m_des.row_1
        vmovaps [rcx+32],xmm6           ;сохранение m_des.row_2
        vmovaps [rcx+48],xmm7           ;сохранение m_des.row_3
Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
AvxMat4x4TransposeF32_ endp
        end
```

```
//-----------------------------------------------
//              Ch06_07_BM.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <string>
#include "Ch06_07.h"
#include "Matrix.h"
#include "BmThreadTimer.h"
#include "OS.h"

using namespace std;

extern void AvxMat4x4TransposeF32_BM(void)
{
    OS::SetThreadAffinityMask();
    cout << "\nИзмерение быстродействия функции AvxMat4x4TransposeF32_BM - подождите\n";
```

```
    const size_t num_rows = 4;
    const size_t num_cols = 4;
    Matrix<float> m_src(num_rows, num_cols);
    Matrix<float> m_des1(num_rows, num_cols);
    Matrix<float> m_des2(num_rows, num_cols);
    const float m_src_r0[] = { 10, 11, 12, 13 };
    const float m_src_r1[] = { 14, 15, 16, 17 };
    const float m_src_r2[] = { 18, 19, 20, 21 };
    const float m_src_r3[] = { 22, 23, 24, 25 };

    m_src.SetRow(0, m_src_r0);
    m_src.SetRow(1, m_src_r1);
    m_src.SetRow(2, m_src_r2);
    m_src.SetRow(3, m_src_r3);

    const size_t num_it = 500;
    const size_t num_alg = 2;
    const size_t num_ops = 1000000;

    BmThreadTimer bmtt(num_it, num_alg);

    for (size_t i = 0; i < num_it; i++)
    {
        bmtt.Start(i, 0);
        for (size_t j = 0; j < num_ops; j++)
            Matrix<float>::Transpose(m_des1, m_src);
        bmtt.Stop(i, 0);

        bmtt.Start(i, 1);
        for (size_t j = 0; j < num_ops; j++)
            AvxMat4x4TransposeF32_(m_des2.Data(), m_src.Data());
        bmtt.Stop(i, 1);
    }

    string fn = bmtt.BuildCsvFilenameString("Ch06_07_AvxMat4x4TransposeF32_BM");
    bmtt.SaveElapsedTimes(fn, BmThreadTimer::EtUnit::MicroSec, 2);
    cout << "Результат сохранен в файл " << fn << '\n';
}
```

Функция `main` начинается с создания экземпляра тестовой матрицы чисел с плавающей запятой одинарной точности 4×4 с именем `m_src` с использованием шаблона `Matrix` C++. Этот шаблон, который определен в заголовочном файле `Matrix.h` (исходный код не показан), содержит код C++, реализующий простой матричный класс для целей тестирования правильности работы и быстродействия ассемблерной части. Внутренний буфер, выделенный `Matrix`, выровнен по 64-байтовой границе, то есть объекты типа `Matrix` правильно выровнены для использования с командами AVX, AVX2 и AVX-512. Функция `main` вызывает `AvxMat4x4TransposeF32`, которая выполняет функции транспонирования матриц, написанные на C++ и языке ассемблера. Результаты этих операций затем передаются в `cout`. Функция `main` также вызывает функцию тестирования с именем `AvxMat4x4TransposeF32_BM`, которая измеряет быстродействие каждой функции транспонирования, как описано далее в этом разделе.

В верхней части кода на языке ассемблера находится макрос с именем `_Mat4x4TransposeF32`. В главе 5 вы узнали, что макрос – это механизм замены текста на ассемблере, который позволяет одной текстовой строкой представлять последовательность команд на языке ассемблера, определений данных или других

операторов. Во время сборки файла исходного кода на языке ассемблера x86 ассемблер заменяет любое вхождение имени макроса операторами, объявленными между директивами `macro` и `endm`. Макросы языка ассемблера обычно используются для генерации последовательностей команд, которые будут использоваться более одного раза. Макросы также часто используются, чтобы избежать накладных расходов производительности при вызове функции.

Макрос `_Mat4x4TransposeF32` содержит команды AVX, которые транспонируют матрицу 4×4 значений с плавающей запятой одинарной точности. Этот макрос требует, чтобы перед его использованием строки исходной матрицы были загружены в регистры XMM0–XMM3. Затем он использует серию команд `vunpcklps`, `vunpckhps`, `vmovlhps` и `vmovhlps` для транспонирования исходной матрицы, как показано на рис. 6.5. После выполнения этих команд транспонированная матрица сохраняется в регистрах XMM4–XMM7.

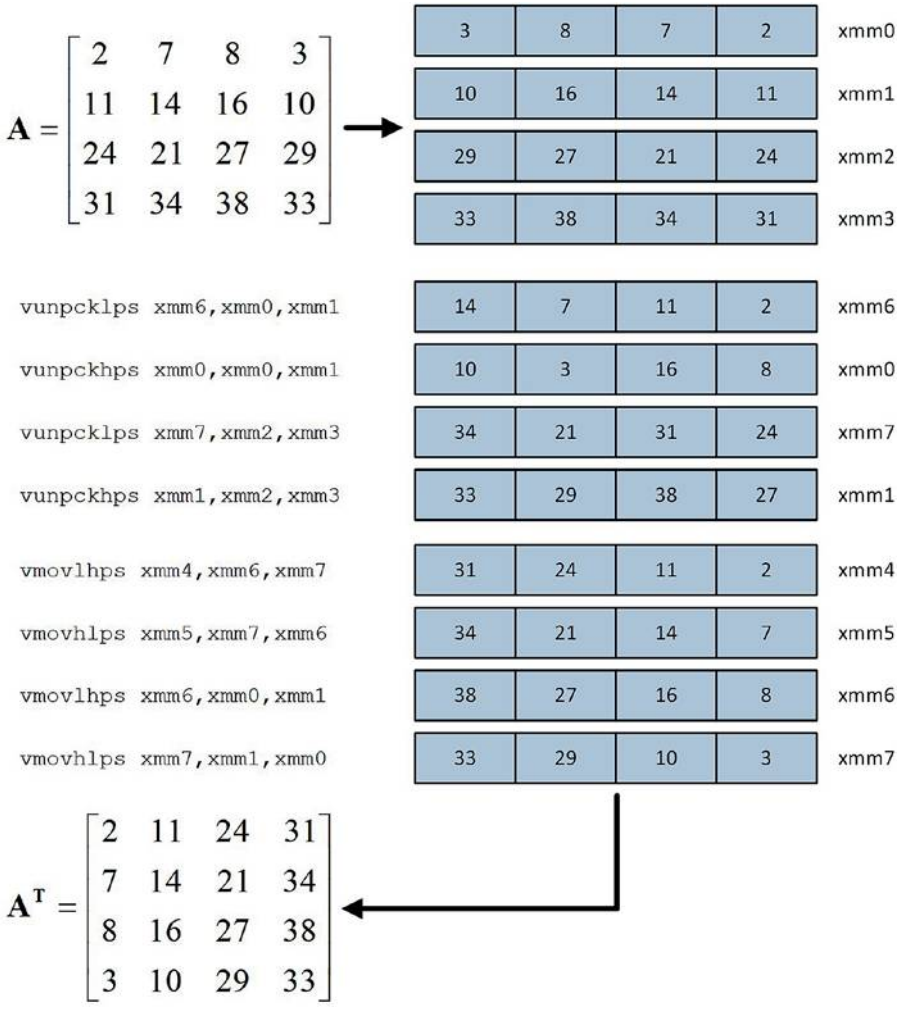


Рис. 6.5. Последовательность команд, используемая макросом `_Mat4x4TransposeF32` для транспонирования матрицы 4×4 значений с плавающей запятой одинарной точности

Макрос `_Mat4x4TransposeF32` используется функцией языка ассемблера `AvxMat4x4Transpose4x4_`. Сразу после пролога функция `AvxMat4x4Transpose4x4_` выполняет серию команд `vmovaps` для загрузки исходной матрицы в регистры XMM0–XMM3. Каждый регистр XMM содержит одну строку исходной матрицы. Затем применяется макрос `_Mat4x4TransposeF32` для транспонирования матрицы. Рисунок 6.6 содержит отрывок из файла листинга MASM, который показывает развернутый макрос `_Mat4x4TransposeF32`. На этом рисунке также показаны развернутые макросы пролога и эпилога. В файле листинга команды, подставленные из макроса, помечены символом 1 в столбце, расположенном слева от мнемоники. После вычисления транспонирования результирующая матрица сохраняется в целевой буфер с помощью другой серии команд `vmovaps`.

```
00000000                    AvxMat4x4TransposeF32_ proc frame
                                    _CreateFrame MT_,0,32
00000000  55                    1   push rbp
00000001  48/ 83 EC 20          1   sub rsp,StackSizeTotal
00000005  48/ 8D 6C 24          1   lea rbp,[rsp+32]
            20
                                    _SaveXmmRegs xmm6,xmm7
0000000A  C5 F9/ 7F 75          1   vmovdqa xmmword ptr [rbp-ValNameOffsetSaveXmmRegs],xmm6
            E0
0000000F  C5 F9/ 7F 7D          1   vmovdqa xmmword ptr [rbp-ValNameOffsetSaveXmmRegs+16],xmm7
            F0
                                    _EndProlog

                                ; Transpose matrix m_src1
00000014  C5 F8/ 28 02              vmovaps xmm0,[rdx]              ;xmm0 = m_src.row_0
00000018  C5 F8/ 28 4A              vmovaps xmm1,[rdx+16]           ;xmm1 = m_src.row_1
            10
0000001D  C5 F8/ 28 52              vmovaps xmm2,[rdx+32]           ;xmm2 = m_src.row_2
            20
00000022  C5 F8/ 28 5A              vmovaps xmm3,[rdx+48]           ;xmm3 = m_src.row_3
            30

                                    _Mat4x4TransposeF32
00000027  C5 F8/ 14 F1          1   vunpcklps xmm6,xmm0,xmm1        ;xmm6 = b1 a1 b0 a0
0000002B  C5 F8/ 15 C1          1   vunpckhps xmm0,xmm0,xmm1        ;xmm0 = b3 a3 b2 a2
0000002F  C5 E8/ 14 FB          1   vunpcklps xmm7,xmm2,xmm3        ;xmm7 = d1 c1 d0 c0
00000033  C5 E8/ 15 CB          1   vunpckhps xmm1,xmm2,xmm3        ;xmm1 = d3 c3 d2 c2
00000037  C5 C8/ 16 E7          1   vmovlhps xmm4,xmm6,xmm7         ;xmm4 = d0 c0 b0 a0
0000003B  C5 C0/ 12 EE          1   vmovhlps xmm5,xmm7,xmm6         ;xmm5 = d1 c1 b1 a1
0000003F  C5 F8/ 16 F1          1   vmovlhps xmm6,xmm0,xmm1         ;xmm6 = d2 c2 b2 a2
00000043  C5 F0/ 12 F8          1   vmovhlps xmm7,xmm1,xmm0         ;xmm7 = d3 c3 b2 a3

00000047  C5 F8/ 29 21              vmovaps [rcx],xmm4              ;save m_des.row_0
0000004B  C5 F8/ 29 69              vmovaps [rcx+16],xmm5           ;save m_des.row_1
            10
00000050  C5 F8/ 29 71              vmovaps [rcx+32],xmm6           ;save m_des.row_2
            20
00000055  C5 F8/ 29 79              vmovaps [rcx+48],xmm7           ;save m_des.row_3
            30

0000005A                    Done:   _RestoreXmmRegs xmm6,xmm7
0000005A  C5 F9/ 6F 75          1   vmovdqa xmm6,xmmword ptr [rbp-ValNameOffsetSaveXmmRegs]
            E0
0000005F  C5 F9/ 6F 7D          1   vmovdqa xmm7,xmmword ptr [rbp-ValNameOffsetSaveXmmRegs+16]
            F0
                                    _DeleteFrame
00000064  48/ 8B E5             1   mov rsp,rbp
00000067  5D                    1   pop rbp
00000068  C3                        ret
00000069                    AvxMat4x4TransposeF32_ endp
                                    end
```

Рис. 6.6. Расширение макроса `_Mat4x4TransposeF32`

Ниже показан результат выполнения исходного кода `Ch06_07`:

```
Результаты AvxMat4x4TransposeF32

Матрица m_src
      2.0   7.0   8.0   3.0
     11.0  14.0  16.0  10.0
     24.0  21.0  27.0  29.0
     31.0  34.0  38.0  33.0

Матрица m_des1
      2.0  11.0  24.0  31.0
      7.0  14.0  21.0  34.0
      8.0  16.0  27.0  38.0
      3.0  10.0  29.0  33.0

Матрица m_des2
      2.0  11.0  24.0  31.0
      7.0  14.0  21.0  34.0
      8.0  16.0  27.0  38.0
      3.0  10.0  29.0  33.0

Измерение быстродействия функции AvxMat4x4TransposeF32_BM - подождите
Результат сохранен в файл Ch06_07_AvxMat4x4TransposeF32_BM_CHROMIUM.csv
```

Пример исходного кода `Ch06_07` включает функцию `AvxMat4x4TransposeF32_BM`, которая содержит код для измерения времени выполнения функций транспонирования матриц C++ и языка ассемблера. Большая часть кода измерения времени инкапсулирована в класс C++ с именем `BmThreadTimer`. Этот класс включает две функции-члена, `BmThreadTimer::Start` и `BmThreadTimer::Stop`, которые реализуют простой программный секундомер. Класс `BmThreadTimer` также включает функцию-член с именем `BmThreadTimer::SaveElapsedTimes`, которая сохраняет измерения времени в текстовый файл с разделителями-запятыми. `AvxMat4x4Transpose_BM` тоже использует класс C++ с именем `OS`. Этот класс включает функции-члены, которые управляют соответствием процессов и потоков. В текущем примере `OS::SetThreadAffinityMask` выбирает конкретный процессор для выполнения эталонного потока. Это повышает точность измерений времени. Исходный код для классов `BmThreadTimer` и `OS` не показан в листинге 6.7, но включен как часть пакета загрузки материалов главы.

Таблица 6.1 содержит результаты измерения времени транспонирования матрицы с использованием нескольких различных процессоров Intel. Измерения проводились с применением EXE-файла, который был создан с конфигурацией Visual C++ Release и настройками по умолчанию для оптимизации кода, за исключением следующих параметров: для упрощения сравнения кода ассемблера C++ и x86-64 была выбрана генерация кода AVX `/arch:AVX` (вариант генерации кода по умолчанию для 64-битного Visual C++ – SSE2); вся оптимизация программы была отключена. Все измерения времени проводились с использованием обычных настольных ПК под управлением Windows 10. Не было предпринято никаких попыток учесть какие-либо различия в оборудовании, программном обеспечении, операционной системе или конфигурации между ПК до запуска исполняемого файла эталонного теста. Условия испытаний, описанные в этом разделе, также используются в последующих главах.

Таблица 6.1. Среднее время выполнения транспонирования матрицы (микросекунды), 1 000 000 транспонирований

CPU	C++	Ассемблер
Intel Core i7-4790S	15885	2575
Intel Core i9-7900X	13381	2203
Intel Core i7-8700K	12216	1825

Значения, показанные в табл. 6.1, были вычислены с использованием времени выполнения файла CSV и функции электронной таблицы Excel `TRIMMEAN(array,0.10)`. Реализация алгоритма транспонирования матриц на языке ассемблера явно превосходит версию C++ с большим отрывом. Нередко достигается значительное повышение скорости с использованием языка ассемблера x86, особенно с помощью алгоритмов, которые могут использовать параллелизм SIMD процессора x86. В оставшейся части книги вы увидите дополнительные примеры ускорения алгоритмической производительности.

Приведенные в этой книге измерения быстродействия дают разумное приближение времени выполнения функций. Как и оценка экономии топлива в автомобиле и времени работы аккумулятора, тестирование производительности программного обеспечения не является точной наукой и содержит множество допущений. Также важно помнить, что эта книга представляет собой вводный курс по программированию на языке ассемблера x86-64, а не по тестированию производительности. Примеры исходного кода структурированы так, чтобы облегчить изучение нового языка программирования, а не для максимальной производительности. Кроме того, параметры Visual C++, описанные ранее, были выбраны в основном из практических соображений и могут не обеспечивать оптимальную производительность во всех случаях. Как и многие высокоуровневые компиляторы, Visual C++ предлагает множество вариантов генерации кода, которые могут повлиять на быстродействие. Контрольные измерения времени выполнения всегда следует рассматривать в контексте, который коррелирует с назначением программного обеспечения. Методы, описанные в этом разделе, в целом полезны, но ваши результаты могут отличаться.

6.5.2. Умножение матриц

Произведение двух матриц определяется следующим образом. Пусть **A** будет матрицей размера $m \times n$, где m и n обозначают количество строк и столбцов соответственно. Пусть **B** – матрица размера $n \times p$. Пусть **C** будет произведением **A** и **B**, которое является матрицей размера $m \times p$. Значение каждого элемента $c(i, j)$ в **C** можно вычислить с помощью следующего уравнения:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} \quad i = 0, \ \ldots, \ m-1; \ j = 0, \ \ldots \ , \ p-1.$$

Прежде чем перейти к примеру кода, сделаю несколько комментариев. Согласно определению умножения матриц, количество столбцов в **A** должно равняться количеству строк в *B*. Например, если **A** – матрица 3×4, а **B** – матрица 4×2, произведение **AB** (матрица 3×2) можно вычислить, но произведение

BA не определено. Обратите внимание, что значение каждого элемента $c(i, j)$ в **C** является просто скалярным произведением строки i в матрице **A** и столбца j в матрице **B**. Код на языке ассемблера будет использовать этот факт для выполнения матричного умножения с использованием команд AVX для работы с упакованными числами. Также обратите внимание, что, в отличие от большинства математических нотаций, в уравнении умножения матриц используют индексацию с отсчетом от нуля. Это упрощает перевод уравнения в код на языке C++ и ассемблере.

В листинге 6.8 показан исходный код примера `Ch06_08`. В этом примере показано, как выполнить умножение матриц с использованием двух матриц значений с плавающей запятой одинарной точности с размерностью 4×4. Как и в предыдущем примере, `main` вызывает функцию `AvxMat4x4MulF32`, реализующую тестовый пример умножения матриц с использованием функций, написанных на C++ и языке ассемблера. Функция-член шаблона `Matrix<float>::Mul` (исходный код не показан) выполняет умножение матриц C++ с использованием ранее описанного уравнения. Как вы скоро увидите, функция языка ассемблера `AvxMat4x4MulF32_` использует арифметику SIMD для умножения матриц.

Листинг 6.8. Пример Ch06_08

```
//-----------------------------------------------
//                 Ch06_08.h
//-----------------------------------------------

#pragma once

// Ch06_08_.asm
extern "C" void AvxMat4x4MulF32_(float* m_des, const float* m_src1, const float* m_src2);

// Ch06_08_BM.cpp
extern void AvxMat4x4MulF32_BM(void);

//-----------------------------------------------
//                Ch06_08.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include "Ch06_08.h"
#include "Matrix.h"

using namespace std;

void AvxMat4x4MulF32(Matrix<float>& m_src1, Matrix<float>& m_src2)
{
    const size_t nr = 4;
    const size_t nc = 4;
    Matrix<float> m_des1(nr ,nc);
    Matrix<float> m_des2(nr ,nc);

    Matrix<float>::Mul(m_des1, m_src1, m_src2);
    AvxMat4x4MulF32_(m_des2.Data(), m_src1.Data(), m_src2.Data());
```

```
    cout << fixed << setprecision(1);

    m_src1.SetOstream(12, " ");
    m_src2.SetOstream(12, " ");
    m_des1.SetOstream(12, " ");
    m_des2.SetOstream(12, " ");

    cout << "\nРезультаты AvxMat4x4MulF32\n";
    cout << "Матрица m_src1\n" << m_src1 << '\n';
    cout << "Матрица m_src2\n" << m_src2 << '\n';
    cout << "Матрица m_des1\n" << m_des1 << '\n';
    cout << "Матрица m_des2\n" << m_des2 << '\n';

    if (m_des1 != m_des2)
        cout << "\nОшибка сравнения матриц - AvxMat4x4MulF32\n";
}

int main()
{
    const size_t nr = 4;
    const size_t nc = 4;
    Matrix<float> m_src1(nr ,nc);
    Matrix<float> m_src2(nr ,nc);

    const float src1_row0[] = { 10, 11, 12, 13 };
    const float src1_row1[] = { 20, 21, 22, 23 };
    const float src1_row2[] = { 30, 31, 32, 33 };
    const float src1_row3[] = { 40, 41, 42, 43 };

    const float src2_row0[] = { 100, 101, 102, 103 };
    const float src2_row1[] = { 200, 201, 202, 203 };
    const float src2_row2[] = { 300, 301, 302, 303 };
    const float src2_row3[] = { 400, 401, 402, 403 };

    m_src1.SetRow(0, src1_row0);
    m_src1.SetRow(1, src1_row1);
    m_src1.SetRow(2, src1_row2);
    m_src1.SetRow(3, src1_row3);

    m_src2.SetRow(0, src2_row0);
    m_src2.SetRow(1, src2_row1);
    m_src2.SetRow(2, src2_row2);
    m_src2.SetRow(3, src2_row3);

    AvxMat4x4MulF32(m_src1, m_src2);
    AvxMat4x4MulF32_BM();
    return 0;
}

;-----------------------------------------------
;                Ch06_08.asm
;-----------------------------------------------

    include <MacrosX86-64-AVX.asmh>

; _Mat4x4MulCalcRowF32 macro
;
```

```
; Описание: этот макрос применяется для вычисления одной строки
; в произведении матриц 4×4
;
; Регистры:     xmm0 = m_src2.row0
;               xmm1 = m_src2.row1
;               xmm2 = m_src2.row2
;               xmm3 = m_src2.row3
;               rcx = m_des ptr
;               rdx = m_src1 ptr
;               xmm4 - xmm7 = исходные регистры

_Mat4x4MulCalcRowF32 macro disp
        vbroadcastss xmm4,real4 ptr [rdx+disp]      ;передача m_src1[i][0]
        vbroadcastss xmm5,real4 ptr [rdx+disp+4]    ;передача m_src1[i][1]
        vbroadcastss xmm6,real4 ptr [rdx+disp+8]    ;передача m_src1[i][2]
        vbroadcastss xmm7,real4 ptr [rdx+disp+12]   ;передача m_src1[i][3]

        vmulps xmm4,xmm4,xmm0                       ;m_src1[i][0] * m_src2.row_0
        vmulps xmm5,xmm5,xmm1                       ;m_src1[i][1] * m_src2.row_1
        vmulps xmm6,xmm6,xmm2                       ;m_src1[i][2] * m_src2.row_2
        vmulps xmm7,xmm7,xmm3                       ;m_src1[i][3] * m_src2.row_3

        vaddps xmm4,xmm4,xmm5                       ;вычисление m_des.row_i
        vaddps xmm6,xmm6,xmm7
        vaddps xmm4,xmm4,xmm6

        vmovaps[rcx+disp],xmm4                      ;сохранение m_des.row_i
        endm

; extern "C" void AvxMat4x4MulF32_(float* m_des, const float* m_src1, const float* m_src2)
;
; Описание: эта функция вычисляет произведение двух матриц 4×4
; чисел одинарной точности с плавающей запятой

        .code
AvxMat4x4MulF32_ proc frame
        _CreateFrame MM_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Вычисление произведения матриц m_des = m_src1 * m_src2
        vmovaps xmm0,[r8]               ;xmm0 = m_src2.row_0
        vmovaps xmm1,[r8+16]            ;xmm1 = m_src2.row_1
        vmovaps xmm2,[r8+32]            ;xmm2 = m_src2.row_2
        vmovaps xmm3,[r8+48]            ;xmm3 = m_src2.row_3

        _Mat4x4MulCalcRowF32 0          ;вычисление m_des.row_0
        _Mat4x4MulCalcRowF32 16         ;вычисление m_des.row_1
        _Mat4x4MulCalcRowF32 32         ;вычисление m_des.row_2
        _Mat4x4MulCalcRowF32 48         ;вычисление m_des.row_3

Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
AvxMat4x4MulF32_ endp
        end
```

```
//-----------------------------------------------
//              Ch06_08_BM.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include "Ch06_08.h"
#include "Matrix.h"
#include "BmThreadTimer.h"
#include "OS.h"

using namespace std;

void AvxMat4x4MulF32_BM(void)
{
    OS::SetThreadAffinityMask();
    cout << "\nИзмерение быстродействия функции AvxMat4x4MulF32_BM - подождите\n";

    const size_t num_rows = 4;
    const size_t num_cols = 4;
    Matrix<float> m_src1(num_rows, num_cols);
    Matrix<float> m_src2(num_rows, num_cols);
    Matrix<float> m_des1(num_rows, num_cols);
    Matrix<float> m_des2(num_rows, num_cols);

    const float m_src1_r0[] = { 10, 11, 12, 13 };
    const float m_src1_r1[] = { 14, 15, 16, 17 };
    const float m_src1_r2[] = { 18, 19, 20, 21 };
    const float m_src1_r3[] = { 22, 23, 24, 25 };
    const float m_src2_r0[] = { 0, 1, 2, 3 };
    const float m_src2_r1[] = { 4, 5, 6, 7 };
    const float m_src2_r2[] = { 8, 9, 10, 11 };
    const float m_src2_r3[] = { 12, 13, 14, 15 };

    m_src1.SetRow(0, m_src1_r0);
    m_src1.SetRow(1, m_src1_r1);
    m_src1.SetRow(2, m_src1_r2);
    m_src1.SetRow(3, m_src1_r3);
    m_src2.SetRow(0, m_src2_r0);
    m_src2.SetRow(1, m_src2_r1);
    m_src2.SetRow(2, m_src2_r2);
    m_src2.SetRow(3, m_src2_r3);

    const size_t num_it = 500;
    const size_t num_alg = 2;
    const size_t num_ops = 1000000;

    BmThreadTimer bmtt(num_it, num_alg);

    for (size_t i = 0; i < num_it; i++)
    {
        bmtt.Start(i, 0);
        for (size_t j = 0; j < num_ops; j++)
            Matrix<float>::Mul(m_des1, m_src1, m_src2);
        bmtt.Stop(i, 0);
        bmtt.Start(i, 1);
```

```
        for (size_t j = 0; j < num_ops; j++)
            AvxMat4x4MulF32_(m_des2.Data(), m_src1.Data(), m_src2.Data());
        bmtt.Stop(i, 1);
    }

    string fn = bmtt.BuildCsvFilenameString("Ch06_08_AvxMat4x4MulF32_BM");
    bmtt.SaveElapsedTimes(fn, BmThreadTimer::EtUnit::MicroSec, 2);
    cout << "Результат сохранен в файл " << fn << '\n';
}
```

Стандартный метод выполнения матричного умножения требует трех вложенных циклов `for`, которые используют скалярное умножение и сложение с плавающей запятой (см. код `Matrix<T>::Mul` в заголовочном файле `Matrix.h`). На рис. 6.7 показаны уравнения, которые можно использовать для вычисления элементов строки 0 для произведения матриц **C** = **AB**. Обратите внимание, что каждая строка матрицы **B** умножается на один и тот же элемент из матрицы **A**. Аналогичные наборы уравнений применяются для вычисления строк 1, 2 и 3 матрицы **C**. Код на языке ассемблера в функции `AvxMatMul4x4F32_` использует эти уравнения для перемножения матриц с использованием арифметики SIMD.

$$\mathbf{C} = \mathbf{AB}$$

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix}$$

$$\begin{aligned} c_{00} &= a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} + a_{03}b_{30} \\ c_{01} &= a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21} + a_{03}b_{31} \\ c_{02} &= a_{00}b_{02} + a_{01}b_{12} + a_{02}b_{22} + a_{03}b_{32} \\ c_{03} &= a_{00}b_{03} + a_{01}b_{13} + a_{02}b_{23} + a_{03}b_{33} \end{aligned}$$

row0 row1 row2 row3

Рис. 6.7. Уравнения для вычисления первой строки матрицы **C** = **AB**

Вслед за прологом `AvxMatMul4x4F32_` загружает матрицу `m_src2` (или **B**) в регистры XMM0-XMM3. Следующие четыре строки используют макрос `_Mat4x4MulCalcRowF32` для вычисления произведений для строк 0...3 матрицы `m_des` (или **C**). Этот макрос реализует четыре уравнения, показанных на рис. 6.7. Параметр макроса `disp` указывает, какую строку использовать. Макрос `_Mat4x4MulCalcRowF32` использует четыре команды `vbroadcastss` для загрузки необходимых элементов из матрицы `m_src1` (или **A**) в регистры XMM4–XMM7. Затем он использует четыре команды `vmulps` для умножения этих значений на всю строку матрицы `m_src2`. Серия команд `vaddps` вычисляет окончательные значения элементов для строки. Команда `vmovaps [rcx+disp],xmm4` сохраняет всю строку в указанный буфер приемника. Ниже показан результат выполнения примера `Ch06_08`:

```
Результаты AvxMat4x4MulF32

Матрица m_src1
       10.0      11.0      12.0      13.0
       20.0      21.0      22.0      23.0
       30.0      31.0      32.0      33.0
       40.0      41.0      42.0      43.0

Матрица m_src2
      100.0     101.0     102.0     103.0
      200.0     201.0     202.0     203.0
      300.0     301.0     302.0     303.0
      400.0     401.0     402.0     403.0

Матрица m_des1
    12000.0   12046.0   12092.0   12138.0
    22000.0   22086.0   22172.0   22258.0
    32000.0   32126.0   32252.0   32378.0
    42000.0   42166.0   42332.0   42498.0

Матрица m_des2
    12000.0   12046.0   12092.0   12138.0
    22000.0   22086.0   22172.0   22258.0
    32000.0   32126.0   32252.0   32378.0
    42000.0   42166.0   42332.0   42498.0

Измерение быстродействия AvxMat4x4MulF32_BM - подождите
Результаты сохранены в файл Ch06_08_AvxMat4x4MulF32_BM_CHROMIUM.csv
```

Пример исходного кода `Ch06_08` также содержит функцию `AvxMat4x4MulF32_BM`, которая измеряет быстродействие функций умножения матриц. В табл. 6.2 показаны результаты измерений для нескольких различных процессоров Intel. Эти измерения были выполнены с использованием процедуры, описанной в предыдущем разделе.

Таблица 6.2. Среднее время выполнения умножения матриц (микросекунды), 1 000 000 умножений

Процессор	C++	Ассемблер
Intel Core i7-4790S	55195	5333
Intel Core i9-7900X	46008	4897
Intel Core i7-8700K	42260	4493

6.6. Заключение

Вот ключевые моменты, которые мы рассмотрели в главе 6:

- команды `vaddp[d|s]`, `vsubp[d|s]`, `vmulp[d|s]`, `vdivp[d|s]` и `vsqrtp[d|s]` выполняют обычные арифметические операции, используя упакованные операнды с плавающей запятой двойной точности и упакованные операнды одинарной точности;

- команды vcvtp[d|s]2dq и vcvtdq2p[d|s] выполняют преобразования между упакованными операндами с плавающей запятой и упакованными операндами со знаком двойного слова. Vcvtps2pd и vcvtpd2ps выполняют преобразования между упакованными операндами одинарной и двойной точности;
- команды vminp[d|s] и vmaxp[d|s] выполняют сжатые вычисления минимального и максимального значений с использованием операндов с плавающей запятой двойной и одинарной точности;
- команды vbroadcasts[d|s] транслируют (или копируют) одно скалярное значение двойной или одинарной точности во все позиции элементов регистра SIMD x86;
- функции языка ассемблера, использующие команды vmovap[d|s] и vmovdqa, могут работать только с правильно выровненными операндами в памяти. Директива MASM align 16 выравнивает элементы данных в разделе .const или .data по 16-байтовой границе. Функции C++ могут использовать спецификатор alignas, чтобы гарантировать правильное выравнивание;
- функции языка ассемблера могут использовать команды vunpck[h|l]p[d|s] для ускорения общих матричных операций, особенно матриц 4×4;
- функции языка ассемблера могут использовать команды vhaddp[d|s] и vshufp[d|s] для сокращения данных промежуточных упакованных значений;
- многие алгоритмы могут значительно повысить производительность за счет использования методов программирования SIMD и набора команд x86-AVX.

Глава 7

Программирование AVX – упакованные целые числа

В предыдущей главе вы узнали, как использовать набор команд AVX для выполнения вычислений с использованием упакованных операндов с плавающей запятой. В этой главе вы узнаете, как выполнять вычисления с использованием упакованных целочисленных операндов. Как и в предыдущей главе, первые несколько примеров исходного кода демонстрируют основные арифметические операции с применением упакованных целых чисел. Остальные примеры исходного кода иллюстрируют использование вычислительных ресурсов AVX для выполнения распространенных операций обработки изображений, включая создание гистограммы и определение пороговых значений.

Набор команд AVX поддерживает операции с упакованными целыми числами с применением операндов шириной 128 бит, и примеры исходного кода в данной главе придерживаются этого ограничения. Для выполнения аналогичных операций с использованием 256-битных операндов требуется процессор, поддерживающий AVX2. Вы узнаете о программировании AVX2 для работы с упакованными целыми числами в главе 10.

7.1. Сложение и вычитание упакованных целых чисел

В листинге 7.1 приведен исходный код примера `Ch07_01` на C++ и на языке ассемблера. В этом примере показано, как выполнять сложение и вычитание упакованных целых чисел с использованием 16-разрядных целых чисел со знаком и без знака. Он также иллюстрирует как обычную, так и насыщенную арифметику.

Листинг 7.1. Пример Ch07_01

```
//-----------------------------------------------
//               Ch07_01.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <string>
#include "XmmVal.h"
```

```
using namespace std;

extern "C" void AvxPackedAddI16_(const XmmVal& a, const XmmVal& b, XmmVal c[2]);
extern "C" void AvxPackedSubI16_(const XmmVal& a, const XmmVal& b, XmmVal c[2]);
extern "C" void AvxPackedAddU16_(const XmmVal& a, const XmmVal& b, XmmVal c[2]);
extern "C" void AvxPackedSubU16_(const XmmVal& a, const XmmVal& b, XmmVal c[2]);

//
// Сложение и вычитание упакованных чисел со знаком
//

void AvxPackedAddI16(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[2];

    a.m_I16[0] = 10;        b.m_I16[0] = 100;
    a.m_I16[1] = 200;       b.m_I16[1] = -200;
    a.m_I16[2] = 30;        b.m_I16[2] = 32760;
    a.m_I16[3] = -32766;    b.m_I16[3] = -400;
    a.m_I16[4] = 50;        b.m_I16[4] = 500;
    a.m_I16[5] = 60;        b.m_I16[5] = -600;
    a.m_I16[6] = 32000;     b.m_I16[6] = 1200;
    a.m_I16[7] = -32000;    b.m_I16[7] = -950;

    AvxPackedAddI16_(a, b, c);

    cout << "\nРезультаты AxvPackedAddI16 - сложение с переносом\n\n";
    cout << "a: " << a.ToStringI16() << '\n';
    cout << "b: " << b.ToStringI16() << '\n';
    cout << "c[0]: " << c[0].ToStringI16() << '\n';
    cout << "\nРезультаты AxvPackedAddI16 - сложение с насыщением\n";
    cout << "a: " << a.ToStringI16() << '\n';
    cout << "b: " << b.ToStringI16() << '\n';
    cout << "c[1]: " << c[1].ToStringI16() << '\n';
}

void AvxPackedSubI16(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[2];

    a.m_I16[0] = 10;        b.m_I16[0] = 100;
    a.m_I16[1] = 200;       b.m_I16[1] = -200;
    a.m_I16[2] = -30;       b.m_I16[2] = 32760;
    a.m_I16[3] = -32766;    b.m_I16[3] = 400;
    a.m_I16[4] = 50;        b.m_I16[4] = 500;
    a.m_I16[5] = 60;        b.m_I16[5] = -600;
    a.m_I16[6] = 32000;     b.m_I16[6] = 1200;
    a.m_I16[7] = -32000;    b.m_I16[7] = 950;

    AvxPackedSubI16_(a, b, c);

    cout << "\nРезультаты AxvPackedSubI16 - Вычитание с переносом\n";
```

```
    cout << "a: " << a.ToStringI16() << '\n';
    cout << "b: " << b.ToStringI16() << '\n';
    cout << "c[0]: " << c[0].ToStringI16() << '\n';
    cout << "\nРезультаты AxvPackedSubI16 - Вычитание с насыщением\n";
    cout << "a: " << a.ToStringI16() << '\n';
    cout << "b: " << b.ToStringI16() << '\n';
    cout << "c[1]: " << c[1].ToStringI16() << '\n';
}

//
// Сложение и вычитание упакованных чисел без знака
//

void AvxPackedAddU16(void)
{
    XmmVal a;
    XmmVal b;
    XmmVal c[2];

    a.m_U16[0] = 10;        b.m_U16[0] = 100;
    a.m_U16[1] = 200;       b.m_U16[1] = 200;
    a.m_U16[2] = 300;       b.m_U16[2] = 65530;
    a.m_U16[3] = 32766;     b.m_U16[3] = 40000;
    a.m_U16[4] = 50;        b.m_U16[4] = 500;
    a.m_U16[5] = 20000;     b.m_U16[5] = 25000;
    a.m_U16[6] = 32000;     b.m_U16[6] = 1200;
    a.m_U16[7] = 32000;     b.m_U16[7] = 50000;

    AvxPackedAddU16_(a, b, c);

    cout << "\nРезультаты AxvPackedAddU16 - сложение с переносом\n";
    cout << "a: " << a.ToStringU16() << '\n';
    cout << "b: " << b.ToStringU16() << '\n';
    cout << "c[0]: " << c[0].ToStringU16() << '\n';
    cout << "\nРезультаты AxvPackedAddU16 - сложение с насыщением\n";
    cout << "a: " << a.ToStringU16() << '\n';
    cout << "b: " << b.ToStringU16() << '\n';
    cout << "c[1]: " << c[1].ToStringU16() << '\n';
}

void AvxPackedSubU16(void)
{
    XmmVal a;
    XmmVal b;
    XmmVal c[2];

    a.m_U16[0] = 10;        b.m_U16[0] = 100;
    a.m_U16[1] = 200;       b.m_U16[1] = 200;
    a.m_U16[2] = 30;        b.m_U16[2] = 7;
    a.m_U16[3] = 65000;     b.m_U16[3] = 5000;
    a.m_U16[4] = 60;        b.m_U16[4] = 500;
    a.m_U16[5] = 25000;     b.m_U16[5] = 28000;
    a.m_U16[6] = 32000;     b.m_U16[6] = 1200;
    a.m_U16[7] = 1200;      b.m_U16[7] = 950;

    AvxPackedSubU16_(a, b, c);
```

```
    cout << "\nРезультаты AxvPackedSubU16 - вычитание с переносом\n";
    cout << "a: " << a.ToStringU16() << '\n';
    cout << "b: " << b.ToStringU16() << '\n';
    cout << "c[0]: " << c[0].ToStringU16() << '\n';
    cout << "\nРезультаты AxvPackedSubI16 - вычитание с насыщением\n";
    cout << "a: " << a.ToStringU16() << '\n';
    cout << "b: " << b.ToStringU16() << '\n';
    cout << "c[1]: " << c[1].ToStringU16() << '\n';
}

int main()
{
    string sep(75, '-');

    AvxPackedAddI16();
    AvxPackedSubI16();
    cout << '\n' << sep << '\n';
    AvxPackedAddU16();
    AvxPackedSubU16();
    return 0;
}

;-----------------------------------------------
;                 Ch07_01.asm
;-----------------------------------------------

; extern "C" void AvxPackedAddI16_(const XmmVal& a, const XmmVal& b, XmmVal c[2])

        .code
AvxPackedAddI16_ proc

; Сложение упакованных слов со знаком
        vmovdqa xmm0,xmmword ptr [rcx]      ;xmm0 = a
        vmovdqa xmm1,xmmword ptr [rdx]      ;xmm1 = b

        vpaddw xmm2,xmm0,xmm1               ;упакованное сложение - перенос
        vpaddsw xmm3,xmm0,xmm1              ;упакованное сложение - насыщение

        vmovdqa xmmword ptr [r8],xmm2       ;сохранение c[0]
        vmovdqa xmmword ptr [r8+16],xmm3    ;сохранение c[1]
        ret
AvxPackedAddI16_ endp

; extern "C" void AvxPackedSubI16_(const XmmVal& a, const XmmVal& b, XmmVal c[2])

AvxPackedSubI16_ proc

; Вычитание упакованных слов со знаком
        vmovdqa xmm0,xmmword ptr [rcx]      ;xmm0 = a
        vmovdqa xmm1,xmmword ptr [rdx]      ;xmm1 = b

        vpsubw xmm2,xmm0,xmm1               ;упакованное вычитание - перенос
        vpsubsw xmm3,xmm0,xmm1              ;упакованное вычитание - насыщение

        vmovdqa xmmword ptr [r8],xmm2       ;сохранение c[0]
        vmovdqa xmmword ptr [r8+16],xmm3    ;сохранение c[1]
```

```
ret
AvxPackedSubI16_ endp

; extern "C" void AvxPackedAddU16_(const XmmVal& a, const XmmVal& b, XmmVal c[2])

AvxPackedAddU16_ proc

; Сложение упакованных слов без знака
        vmovdqu xmm0,xmmword ptr [rcx]      ;xmm0 = a
        vmovdqu xmm1,xmmword ptr [rdx]      ;xmm1 = b

        vpaddw xmm2,xmm0,xmm1               ;упакованное сложение - перенос
        vpaddusw xmm3,xmm0,xmm1             ;упакованное сложение - насыщение

        vmovdqu xmmword ptr [r8],xmm2       ;сохранение c[0]
        vmovdqu xmmword ptr [r8+16],xmm3    ;сохранение c[1]
        ret
AvxPackedAddU16_ endp

; extern "C" void AvxPackedSubU16_(const XmmVal& a, const XmmVal& b, XmmVal c[2])

AvxPackedSubU16_ proc

; Вычитание упакованных слов без знака
        vmovdqu xmm0,xmmword ptr [rcx]      ;xmm0 = a
        vmovdqu xmm1,xmmword ptr [rdx]      ;xmm1 = b

        vpsubw xmm2,xmm0,xmm1               ;упакованное вычитание - перенос
        vpsubusw xmm3,xmm0,xmm1             ;упакованное вычитание - насыщение

        vmovdqu xmmword ptr [r8],xmm2       ;сохранение c[0]
        vmovdqu xmmword ptr [r8+16],xmm3    ;сохранение c[1]
        ret
AvxPackedSubU16_ endp
        end
```

В начале кода C++ расположены объявления функций на языке ассемблера, которые выполняют сложение и вычитание упакованных целых чисел. Каждая функция принимает два аргумента `XmmVal` и сохраняет свои результаты в массив `XmmVal`. Структура `XmmVal`, о которой вы узнали в главе 6 (см. листинг 6.1), содержит публичное анонимное объединение с членами, соответствующими типам упакованных данных, которые могут использоваться с регистром XMM. Структура `XmmVal` также определяет несколько функций-членов, которые форматируют содержимое `XmmVal` для отображения.

Функция C++ `AvxPackedAddI16` содержит тестовый код, который вызывает функцию на языке ассемблера `AvxPackedAddI16_`. Эта функция выполняет сложение упакованных 16-битных целых чисел (слов) со знаком, используя как обычную, так и насыщенную арифметику. Обратите внимание, что все переменные `XmmVal` `a`, `b` и `c` определены с помощью спецификатора C++ `alignas (16)`, выравнивающего каждый `XmmVal` по 16-байтовой границе. После выполнения функции `AvxPackedaddI16_` результаты отображаются с помощью нескольких потоковых записей в `cout`. Функция C++ `AvxPackedSubI16`, аналогичная `AvxPackedAddI16`, использует функцию на языке ассемблера `AvxPackedSubI16_`.

Параллельный набор функций C++, AvxPackedAddU16 и AvxPackedSubU16, содержит код, вызывающий функции на языке ассемблера AvxPackedAddU16_ и AvxPackedSubU16_. Эти функции выполняют сложение и вычитание упакованных 16-битных целых чисел без знака соответственно. Обратите внимание, что переменные XmmVal в AvxPackedAddU16 и AvxPackedSubU16 не используют спецификатор alignas(16), то есть эти значения не будут гарантированно выровнены по 16-байтовой границе. Причина этого заключается в желании продемонстрировать использование команды AVX vmovdqu (перемещение невыровненных упакованных целочисленных значений), описание применения которой вы скоро увидите.

Функция языка ассемблера AvxPackedAddI6_ начинается с команды vmovdqa xmm0,xmmword ptr [rcx], которая загружает значение аргумента a в регистр XMM0. Следующая команда vmovdqa xmm1,xmmword ptr [rdx] копирует b в регистр XMM1. Следующие две команды, vpaddw xmm2,xmm0,xmm1 и vpaddsw xmm3,xmm0,xmm1, выполняют сложение упакованных 16-битных целых чисел со знаком с использованием арифметики с переносом и насыщением соответственно. Последние две команды vmovdqa сохраняют полученные результаты в массив XmmVal c. Функция языка ассемблера AvxPackedSubI16_ похожа на AvxPackedAddI16_ и использует команды vpsubw и vpsubsw для выполнения вычитания 16-битных упакованных целых чисел со знаком.

Функция языка ассемблера AvxPackedAddU16_ начинается с команды vmovdqu xmm0,xmmword ptr [rcx], которая загружает в регистр XMM0. Здесь используется команда vmovdqu, поскольку XmmVal a был определен в коде C++ без спецификатора alignas (16). Обратите внимание, что функция AvxPackedAddU16_ использует vmovdqu только в демонстрационных целях; вместо этого следовало использовать правильно выровненный XmmVal и команду vmovdqa. Этот момент уже неоднократно упоминался в книге, но заслуживает повторения из-за его важности: операнды SIMD в памяти должны быть правильно выровнены всегда, когда это возможно, чтобы избежать потенциальных потерь производительности, которые могут возникнуть всякий раз, когда процессор обращается к невыровненному операнду в памяти.

После загрузки значений аргумента a и b в регистры XMM0 и XMM1 функция AvxPackedAddU16_ выполняет сложение упакованных беззнаковых 16-битных целых чисел, используя команды vpaddw xmm2,xmm0,xmm1 (арифметика с переносом) и vpaddusw xmm3, xmm0,xmm1 (арифметика с насыщением). Две команды vmovdqu сохраняют результаты в массив c. Функция AvxPackedSubU16_ реализует упакованное вычитание 16-битных целых чисел без знака с использованием команд vpsubw и vpsubusw. Эта функция тоже использует команду vmovdqu для загрузки значений аргументов и сохранения результатов. Вот результаты выполнения примера исходного кода Ch07_01:

```
Результаты AxvPackedAddI16 - сложение с переносом
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
```

```
Результаты AxvPackedAddI16 - сложение с насыщением
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

Результаты AxvPackedSubI16 - вычитание с переносом
a:           10      200     -30  -32766   |      50       60    32000  -32000
b:          100     -200   32760     400   |     500     -600     1200     950
c[0]:       -90      400   32746   32370   |    -450      660    30800   32586

Результаты AxvPackedSubI16 - вычитание с насыщением
a:           10      200     -30  -32766   |      50       60    32000  -32000
b:          100     -200   32760     400   |     500     -600     1200     950
c[1]:       -90      400  -32768  -32768   |    -450      660    30800  -32768

--------------------------------------------------------------------------

Результаты AxvPackedAddU16 - сложение с переносом
a:           10      200     300   32766   |      50    20000    32000   32000
b:          100      200   65530   40000   |     500    25000     1200   50000
c[0]:       110      400     294    7230   |     550    45000    33200   16464

Результаты AxvPackedAddU16 - сложение с насыщением
a:           10      200     300   32766   |      50    20000    32000   32000
b:          100      200   65530   40000   |     500    25000     1200   50000
c[1]:       110      400   65535   65535   |     550    45000    33200   65535

Результаты AxvPackedSubU16 - вычитание с переносом
a:           10      200      30   65000   |      60    25000    32000    1200
b:          100      200       7    5000   |     500    28000     1200     950
c[0]:     65446        0      23   60000   |   65096    62536    30800     250

Результаты for AxvPackedSubI16 - вычитание с насыщением
a:           10      200      30   65000   |      60    25000    32000    1200
b:          100      200       7    5000   |     500    28000     1200     950
c[1]:         0        0      23   60000   |       0        0    30800     250
```

AVX также поддерживает сложение и вычитание упакованных целых чисел с использованием 8-, 32- и 64-разрядных целочисленных операндов. Команды `vpaddb`, `vpaddsb`, `vpaddusb`, `vpsubb`, `vpsubsb` и `vpsubusb` – это 8-битные (байтовые) версии упакованных 16-битных команд, которые были продемонстрированы в этом примере. Команды `vpadd[d|q]` и `vpsub[d|q]` могут использоваться для выполнения упакованного 32-битного (двойное слово) или 64-битного (четверное слово) сложения и вычитания с использованием *обычной* арифметики с переносом. AVX не поддерживает сложение и вычитание с насыщением для упакованных двойных или четверных целых чисел.

7.2. Сдвиг упакованных целых чисел

Следующий пример исходного кода называется `Ch07_02`. В этом примере показано, как выполнять операции логического и арифметического сдвига с использованием упакованных целочисленных операндов. В листинге 7.2 показан исходный код примера `Ch07_02` на C++ и языке ассемблера.

Листинг 7.2. Пример Ch07_02

```
//-----------------------------------------------
//               Ch07_02.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include "XmmVal.h"

using namespace std;

// Порядок имен констант в перечислении ниже должен совпадать
// с порядком табличных значений в файле .asm

enum ShiftOp : unsigned int
{
    U16_LL,     // логический сдвиг влево - слово
    U16_RL,     // логический сдвиг вправо - слово
    U16_RA,     // арифметический сдвиг вправо - слово
    U32_LL,     // логический сдвиг влево - двойное слово
    U32_RL,     // логический сдвиг вправо - двойное слово
    U32_RA,     // арифметический сдвиг вправо - двойное слово
};

extern "C" bool AvxPackedIntegerShift_(XmmVal& b, const XmmVal& a, ShiftOp shift_op,
unsigned int count);

void AvxPackedIntegerShiftU16(void)
{
    unsigned int count = 2;
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;

    a.m_U16[0] = 0x1234;
    a.m_U16[1] = 0xFF00;
    a.m_U16[2] = 0x00CC;
    a.m_U16[3] = 0x8080;
    a.m_U16[4] = 0x00FF;
    a.m_U16[5] = 0xAAAA;
    a.m_U16[6] = 0x0F0F;
    a.m_U16[7] = 0x0101;

    AvxPackedIntegerShift_(b, a, U16_LL, count);
    cout << "\nРезультаты ShiftOp::U16_LL (count = " << count << ")\n";
    cout << "a: " << a.ToStringX16() << '\n';
    cout << "b: " << b.ToStringX16() << '\n';

    AvxPackedIntegerShift_(b, a, U16_RL, count);
    cout << "\nРезультаты ShiftOp::U16_RL (count = " << count << ")\n";
    cout << "a: " << a.ToStringX16() << '\n';
    cout << "b: " << b.ToStringX16() << '\n';

    AvxPackedIntegerShift_(b, a, U16_RA, count);
    cout << "\nРезультаты ShiftOp::U16_RA (count = " << count << ")\n";
    cout << "a: " << a.ToStringX16() << '\n';
    cout << "b: " << b.ToStringX16() << '\n';
}
```

```
void AvxPackedIntegerShiftU32(void)
{
    unsigned int count = 4;
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;

    a.m_U32[0] = 0x12345678;
    a.m_U32[1] = 0xFF00FF00;
    a.m_U32[2] = 0x03030303;
    a.m_U32[3] = 0x80800F0F;

    AvxPackedIntegerShift_(b, a, U32_LL, count);
    cout << "\nРезультаты ShiftOp::U32_LL (count = " << count << ")\n";
    cout << "a: " << a.ToStringX32() << '\n';
    cout << "b: " << b.ToStringX32() << '\n';

    AvxPackedIntegerShift_(b, a, U32_RL, count);
    cout << "\nРезультаты ShiftOp::U32_RL (count = " << count << ")\n";
    cout << "a: " << a.ToStringX32() << '\n';
    cout << "b: " << b.ToStringX32() << '\n';

    AvxPackedIntegerShift_(b, a, U32_RA, count);
    cout << "\nРезультаты ShiftOp::U32_RA (count = " << count << ")\n";
    cout << "a: " << a.ToStringX32() << '\n';
    cout << "b: " << b.ToStringX32() << '\n';
}

int main(void)
{
    string sep(75, '-');

    AvxPackedIntegerShiftU16();
    cout << '\n' << sep << '\n';
    AvxPackedIntegerShiftU32();
    return 0;
}

;-----------------------------------------------
;                 Ch07_02.asm
;-----------------------------------------------

; extern "C" bool AvxPackedIntegerShift_(XmmVal& b, const XmmVal& a, ShiftOp shift_op,
unsigned int count)
;
; Возвращает: 0 = недопустимый аргумент shift_op, 1 = успех
;
; Примечание: для этого модуля опция компоновщика /LARGEADDRESSAWARE:NO
; должна быть указана в явном виде

        .code
AvxPackedIntegerShift_ proc
; Проверка 'shift_op' на допустимость
        mov r8d,r8d                     ;дополнение shift_op нулями
        cmp r8,ShiftOpTableCount        ;сравнение со счетчиком таблицы
        jae Error                       ;переход, если shift_op недопустим
```

```
; Переход к операции, определенной в shift_op
        vmovdqa xmm0,xmmword ptr [rdx]  ;xmm0 = a
        vmovd xmm1,r9d                  ;xmm1[31:0] = счетчик сдвига
        mov eax,1                       ;возвращаемый код успешного выполнения
        jmp [ShiftOpTable+r8*8]

; Логический сдвиг влево упакованного слова
U16_LL: vpsllw xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Логический сдвиг вправо упакованного слова
U16_RL: vpsrlw xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Арифметический сдвиг вправо упакованного слова
U16_RA: vpsraw xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Логический сдвиг влево упакованного двойного слова
U32_LL: vpslld xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Логический сдвиг вправо упакованного двойного слова
U32_RL: vpsrld xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

; Арифметический сдвиг вправо упакованного двойного слова
U32_RA: vpsrad xmm2,xmm0,xmm1
        vmovdqa xmmword ptr [rcx],xmm2
        ret

Error:  xor eax,eax                     ;возвращаемый код ошибки
        vpxor xmm0,xmm0,xmm0
        vmovdqa xmmword ptr [rcx],xmm0  ;установка результата в ноль
        ret

; Порядок меток в следующей таблице должен совпадать с
; порядком элементов перечисления в файле .cpp

                align 8
ShiftOpTable    qword U16_LL, U16_RL, U16_RA
                qword U32_LL, U32_RL, U32_RA
ShiftOpTableCount equ ($ - ShiftOpTable) / size qword

AvxPackedIntegerShift_ endp
        end
```

Код C++, показанный в листинге 7.2, начинается с определения перечисления с именем `ShiftOp`, которое используется для выбора операции сдвига. Поддерживаемые операции сдвига включают логический сдвиг влево, логический

сдвиг вправо и арифметический сдвиг вправо с использованием упакованных значений шириной в одно и два слова. Следом за перечислением идет объявление функции AvxPackedIntegerShift_. Эта функция выполняет запрошенную операцию сдвига, используя переданный аргумент XmmVal и указанное значение count. Функции C++ AvxPackedIntegerShiftU16 и AvxPackedIntegerShiftU32 инициализируют тестовые примеры для выполнения различных операций сдвига с использованием упакованных одинарных и двойных слов соответственно.

Функция на языке ассемблера AvxPackedIntegerShift_ использует таблицу переходов для выполнения указанной операции сдвига. Этот прием похож на тот, что вы видели в примерах исходного кода Ch05_06 (глава 5) и Ch06_03 (глава 6). При входе в AvxPackedIntegerShift_ значение аргумента shift_op проверяется на допустимость. После успешной проверки shift_op команда vmovdqa xmm0,xmmword ptr [rdx] загружает a в регистр XMM0. Последующая команда vmovd xmm1,r9d копирует значение аргумента count в младшее двойное слово регистра XMM1. За этим следует команда jmp [ShiftOpTable+r8*8], которая передает управление программой соответствующему блоку кода.

Каждый отдельный блок кода в AvxPackedIntegerShift_ выполняет определенную операцию сдвига. Например, кодовый блок рядом с меткой U16_LL использует команду AVX vpsllw xmm2,xmm0,xmm1 для выполнения логического сдвига влево с применением упакованных слов. Важно отметить, что каждый элемент слова в XMM0 независимо сдвигается влево на количество битов, указанное в XMM1 [31:0]. Блоки кода, смежные с метками U16_RL и U16_RA, выполняют логический и арифметический сдвиги вправо упакованных слов с использованием команд vpsrlw и vpsraw соответственно. Функция AvxPackedIntegerShift_ использует аналогичную структуру для выполнения операций сдвига упакованных двойных слов с использованием команд vpslld, vpslrd и vpsrad. Все блоки кода в AvxPackedIntegerShift_ завершаются командой vmovdqa xmmword ptr [rcx],xmm2, которая сохраняет вычисленный результат. Вот результат выполнения примера Ch07_02:

```
Результаты ShiftOp::U16_LL (count = 2)
a:      1234    FF00    00CC    8080   |   00FF    AAAA    0F0F    0101
b:      48D0    FC00    0330    0200   |   03FC    AAA8    3C3C    0404

Результаты ShiftOp::U16_RL (count = 2)
a:      1234    FF00    00CC    8080   |   00FF    AAAA    0F0F    0101
b:      048D    3FC0    0033    2020   |   003F    2AAA    03C3    0040

Результаты ShiftOp::U16_RA (count = 2)
a:      1234    FF00    00CC    8080   |   00FF    AAAA    0F0F    0101
b:      048D    FFC0    0033    E020   |   003F    EAAA    03C3    0040

--------------------------------------------------------------------------

Результаты ShiftOp::U32_LL (count = 4)
a:          12345678        FF00FF00   |       03030303        80800F0F
b:          23456780        F00FF000   |       30303030        0800F0F0

Результаты ShiftOp::U32_RL (count = 4)
```

```
a:          12345678        FF00FF00    |       03030303        80800F0F
b:          01234567        0FF00FF0    |       00303030        080800F0

Результаты ShiftOp::U32_RA (count = 4)
a:          12345678        FF00FF00    |       03030303        80800F0F
b:          01234567        FFF00FF0    |       00303030        F80800F0
```

Команды AVX `vpsllq`, `vpslrq` и `vpsraq` могут применяться для выполнения операций сдвига с использованием упакованных четверных слов. Несколько удивительно, что AVX не поддерживает операции сдвига с использованием упакованных байтовых операндов. В составе AVX также имеются команды сдвига `vps[l|r]dq`, которые выполняют логический левый или логический сдвиг правого байта 128-битного операнда в регистре XMM. В следующем разделе вы увидите, как работают эти команды.

7.3. Умножение упакованных целых чисел

Помимо сложения и вычитания упакованных целых чисел, AVX также содержит команды, выполняющие умножение упакованных целых чисел. Эти команды немного отличаются от соответствующих команд сложения и вычитания упакованных целых чисел. Частично это различие связано с тем, что для вычисления необрезанного произведения при целочисленном умножении необходимо, чтобы операнд-приемник был вдвое больше, чем исходный операнд-источник. Например, неусеченное произведение двух 16-разрядных целых чисел со знаком является 32-разрядным целым числом со знаком. В листинге 7.3 приведен исходный код примера `Ch07_03`. В этом примере показано, как выполнить умножение упакованных целых чисел с использованием 16-разрядных и 32-разрядных целых чисел со знаком.

Листинг 7.3. Пример Ch07_03

```
//-----------------------------------------------
//              Ch07_03.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include "XmmVal.h"

using namespace std;

extern "C" void AvxPackedMulI16_(XmmVal c[2], const XmmVal& a, const XmmVal& b);
extern "C" void AvxPackedMulI32A_(XmmVal c[2], const XmmVal& a, const XmmVal& b);
extern "C" void AvxPackedMulI32B_(XmmVal* c, const XmmVal& a, const XmmVal& b);

void AvxPackedMulI16(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[2];
```

```
    a.m_I16[0] = 10;        b.m_I16[0] = -5;
    a.m_I16[1] = 3000;      b.m_I16[1] = 100;
    a.m_I16[2] = -2000;     b.m_I16[2] = -9000;
    a.m_I16[3] = 42;        b.m_I16[3] = 1000;
    a.m_I16[4] = -5000;     b.m_I16[4] = 25000;
    a.m_I16[5] = 8;         b.m_I16[5] = 16384;
    a.m_I16[6] = 10000;     b.m_I16[6] = 3500;
    a.m_I16[7] = -60;       b.m_I16[7] = 6000;

    AvxPackedMulI16_(c, a, b);

    cout << "\nРезультаты AvxPackedMulI16\n";

    for (size_t i = 0; i < 8; i++)
    {
        cout << "a[" << i << "]: " << setw(8) << a.m_I16[i] << " ";
        cout << "b[" << i << "]: " << setw(8) << b.m_I16[i] << " ";

        if (i < 4)
        {
        cout << "c[0][" << i << "]: ";
        cout << setw(12) << c[0].m_I32[i] << '\n';
        }
        else
        {
        cout << "c[1][" << i - 4 << "]: ";
        cout << setw(12) << c[1].m_I32[i - 4] << '\n';
        }
    }
}

void AvxPackedMulI32A(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c[2];

    a.m_I32[0] = 10;        b.m_I32[0] = -500;
    a.m_I32[1] = 3000;      b.m_I32[1] = 100;
    a.m_I32[2] = -40000;    b.m_I32[2] = -120000;
    a.m_I32[3] = 4200;      b.m_I32[3] = 1000;

    AvxPackedMulI32A_(c, a, b);

    cout << "\nРезультаты AvxPackedMulI32A\n";

    for (size_t i = 0; i < 4; i++)
    {
        cout << "a[" << i << "]: " << setw(10) << a.m_I32[i] << " ";
        cout << "b[" << i << "]: " << setw(10) << b.m_I32[i] << " ";

        if (i < 2)
        {
            cout << "c[0][" << i << "]: ";
            cout << setw(14) << c[0].m_I64[i] << '\n';
        }
```

```
        else
        {
            cout << "c[1][" << i - 2 << "]: ";
            cout << setw(14) << c[1].m_I64[i - 2] << '\n';
        }
    }
}

void AvxPackedMulI32B(void)
{
    alignas(16) XmmVal a;
    alignas(16) XmmVal b;
    alignas(16) XmmVal c;

    a.m_I32[0] = 10;    b.m_I32[0] = -500;
    a.m_I32[1] = 3000;  b.m_I32[1] = 100;
    a.m_I32[2] = -2000; b.m_I32[2] = -12000;
    a.m_I32[3] = 4200;  b.m_I32[3] = 1000;

    AvxPackedMulI32B_(&c, a, b);

    cout << "\nРезультаты AvxPackedMulI32B\n";

    for (size_t i = 0; i < 4; i++)
    {
        cout << "a[" << i << "]: " << setw(10) << a.m_I32[i] << " ";
        cout << "b[" << i << "]: " << setw(10) << b.m_I32[i] << " ";
        cout << "c[" << i << "]: " << setw(10) << c.m_I32[i] << '\n';
    }
}

int main()
{
    string sep(75, '-');

    AvxPackedMulI16();
    cout << '\n' << sep << '\n';
    AvxPackedMulI32A();
    cout << '\n' << sep << '\n';
    AvxPackedMulI32B();
    return 0;
}

;-----------------------------------------------
;                Ch07_03.asm
;-----------------------------------------------

; extern "C" void AvxPackedMulI16_(XmmVal c[2], const XmmVal* a, const XmmVal* b)

        .code
AvxPackedMulI16_ proc
        vmovdqa xmm0,xmmword ptr [rdx]  ;xmm0 = a
        vmovdqa xmm1,xmmword ptr [r8]   ;xmm1 = b

        vpmullw xmm2,xmm0,xmm1          ;xmm2 = младшие разряды упакованного a * b
        vpmulhw xmm3,xmm0,xmm1          ;xmm3 = старшие разряды упакованного a * b
```

```
        vpunpcklwd xmm4,xmm2,xmm3       ;слияние младших и старших разрядов
        vpunpckhwd xmm5,xmm2,xmm3       ;в финальное четверное слово со знаком

        vmovdqa xmmword ptr [rcx],xmm4  ;сохранение окончательного результата
        vmovdqa xmmword ptr [rcx+16],xmm5
        ret
AvxPackedMulI16_ endp

; extern "C" void AvxPackedMulI32A_(XmmVal c[2], const XmmVal* a, const XmmVal* b)

AvxPackedMulI32A_ proc

; Умножение упакованных двойных слов со знаком. Учтите, что vpmuldq
; выполняет следующие операции:
;
; xmm2[63:0] = xmm0[31:0] * xmm1[31:0]
; xmm2[127:64] = xmm0[95:64] * xmm1[95:64]

        vmovdqa xmm0,xmmword ptr [rdx] ;xmm0 = a
        vmovdqa xmm1,xmmword ptr [r8] ;xmm1 = b
        vpmuldq xmm2,xmm0,xmm1

; Сдвиг операндов-источников вправо на 4 байта и повтор vpmuldq
        vpsrldq xmm0,xmm0,4
        vpsrldq xmm1,xmm1,4
        vpmuldq xmm3,xmm0,xmm1

; Сохранение результатов
        vpextrq qword ptr [rcx],xmm2,0      ;сохранение xmm2[63:0]
        vpextrq qword ptr [rcx+8],xmm3,0    ;сохранение xmm3[63:0]
        vpextrq qword ptr [rcx+16],xmm2,1   ;сохранение xmm2[127:63]
        vpextrq qword ptr [rcx+24],xmm3,1   ;сохранение xmm3[127:63]
        ret
AvxPackedMulI32A_ endp

; extern "C" void AvxPackedMulI32B_(XmmVal*, const XmmVal* a, const XmmVal* b)

AvxPackedMulI32B_ proc

; Умножение упакованных целых чисел со знаком и сохранение младших разрядов результата
        vmovdqa xmm0,xmmword ptr [rdx]      ;xmm0 = a
        vpmulld xmm1,xmm0,xmmword ptr [r8]  ;xmm1 = упакованное a * b
        vmovdqa xmmword ptr [rcx],xmm1      ;сохранение упакованного результата
        ret
AvxPackedMulI32B_ endp
        end
```

Функция C++ `AvxPackedMulI16` содержит код, который инициализирует переменные `XmmVal a` и `b` 16-разрядными целочисленными значениями со знаком. Затем эта функция вызывает функцию на языке ассемблера `AxvPackedMulI16_`, которая выполняет умножение упакованных 16-разрядных целых чисел со знаком. Далее результаты передаются в `cout`. Обратите внимание, что результаты, отображаемые функцией `AvxPackedMulI16`, представляют собой 32-разрядные целочисленные продукты со знаком. Две другие функции C++ в листинге 7.3, `AvxPackedMulI32A` и `AvxPackedMul32B`, инициализируют тестовые примеры

для выполнения умножения 32-разрядных целых чисел со знаком. Первая из этих функций вычисляет 64-битное упакованное целочисленное произведение со знаком, а вторая – 32-битное упакованное целочисленное произведение со знаком.

Функция на языке ассемблера `AvxPackedMulI16_` начинается с двух команд `vmovdqa`, которые загружают значения аргументов `a` и `b` в регистры XMM0 и XMM1 соответственно. Следующая команда `vpmullw xmm2,xmm0,xmm1` перемножает упакованные 16-битные целые числа со знаком в XMM0 и XMM1 и сохраняет младшие 16 бит каждого 32-битного произведения в XMM2. Затем следует команда `vpmulhw xmm3,xmm0,xmm1`, которая вычисляет и сохраняет старшие 16 бит каждого 32-битного произведения. Следующие две команды, `vpunpcklwd xmm4,xmm2,xmm3` и `vpunpckhwd xmm5,xmm2,xmm3`, формируют окончательные упакованные 32-битные целые числа со знаком, чередуя слова младших (`vpunpcklwd`) или старших (`vpunpckhwd`) разрядов исходных операндов. На рис. 7.1 схематически показана последовательность команд функции `AvxPackedMulI16_`.

Рис. 7.1. Последовательность команд функции `AvxPackedMulI16_` для перемножения упакованных 16-разрядных целых чисел со знаком

Следующая функция на языке ассемблера в листинге 7.3, `AvxPackedMulI32A_`, выполняет перемножение упакованных 32-битных целых чисел со знаком. Эта функция начинается с двух команд `vmovdqa`, которые загружают значения аргумента `XmmVal a` и `b` в регистры XMM0 и XMM1 соответственно. Далее команда `vpmuldq xmm2,xmm0,xmm1` выполняет перемножение упакованных 32-битных целых чисел со знаком с использованием четных элементов двух исходных операндов. Затем она сохраняет 64-битные произведения со знаком в XMM2. Далее две команды `vpsrldq` сдвигают вправо на четыре байта содержимое ре-

гистров XMM0 и XMM1. Затем вновь следует команда `vpmuldq`, которая вычисляет оставшиеся 64-битные произведения. На рис. 7.2 схематически показаны детали выполнения этой последовательности команд.

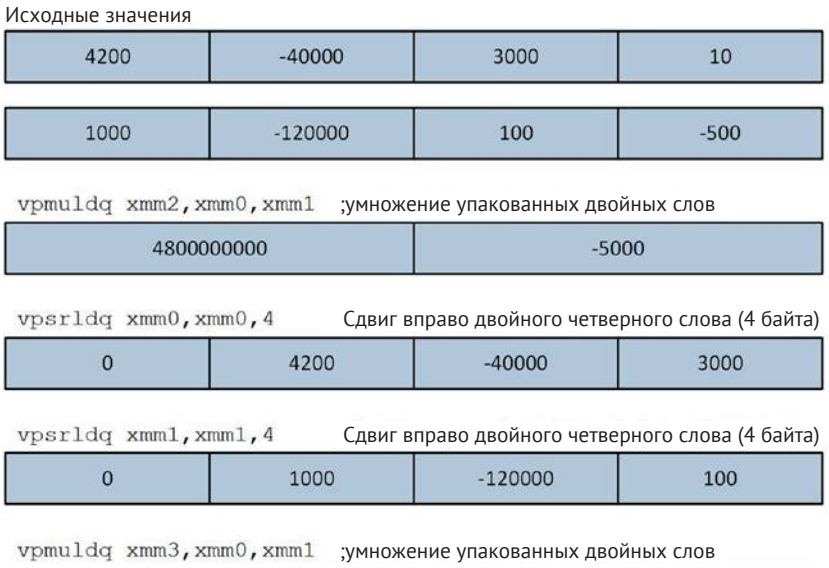


Рис. 7.2. Выполнение команд `vpmuldq` и `vpsrldq`

После выполнения второй команды `vpmuldq` регистры XMM2 и XMM3 содержат четыре 64-битных произведения со знаком. Эти значения затем сохраняются в указанном целевом буфере с помощью серии команд `vpextrq` (извлечь четверное слово). Данная команда копирует элемент четверного слова, указанный непосредственным (или вторым исходным) операндом, из первого исходного операнда и сохраняет его в операнде-приемнике. Например, команда `vpextrq qword ptr [rcx],xmm2,0` сохраняет четверное слово младшего разряда XMM2 в ячейку памяти, указанную RCX. Первым исходным операндом команды `vpextrq` должен быть регистр XMM; операнд-приемник может быть регистром общего назначения или ячейкой памяти. AVX также содержит команды, которые можно использовать для извлечения элементов в формате байта (`vpextrb`), слова (`vpextrw`) или двойного слова (`vpextrd`).

Последняя функция языка ассемблера в этом примере исходного кода называется `AvxPackedMulI32B_`. Эта функция также выполняет умножение упакованных 32-битных целых чисел со знаком, но сохраняет усеченные 32-битные произведения. Функция `AvxPackedMulI32B_` использует команду `vpmulld`, которая выполняет поэлементное перемножение двойных слов подобно сложению или вычитанию упакованных чисел. Затем младшие 32 бита каждого произведения сохраняются в операнде-приемнике. Результаты выполнения примера `Ch07_03` выглядят следующим образом:

```
Результаты AvxPackedMulI16
a[0]:       10  b[0]:       -5  c[0][0]:        -50
a[1]:     3000  b[1]:      100  c[0][1]:     300000
a[2]:    -2000  b[2]:    -9000  c[0][2]:   18000000
a[3]:       42  b[3]:     1000  c[0][3]:      42000
a[4]:    -5000  b[4]:    25000  c[1][0]: -125000000
a[5]:        8  b[5]:    16384  c[1][1]:     131072
a[6]:    10000  b[6]:     3500  c[1][2]:   35000000
a[7]:      -60  b[7]:     6000  c[1][3]:    -360000

--------------------------------------------------------------------------------

Результаты AvxPackedMulI32A
a[0]:       10  b[0]:     -500  c[0][0]:      -5000
a[1]:     3000  b[1]:      100  c[0][1]:     300000
a[2]:   -40000  b[2]:  -120000  c[1][0]: 4800000000
a[3]:     4200  b[3]:     1000  c[1][1]:    4200000

--------------------------------------------------------------------------------

Результаты AvxPackedMulI32B
a[0]:       10  b[0]:     -500     c[0]:      -5000
a[1]:     3000  b[1]:      100     c[1]:     300000
a[2]:    -2000  b[2]:   -12000     c[2]:   24000000
a[3]:     4200  b[3]:     1000     c[3]:    4200000
```

7.4. Обработка изображений с применением упакованных целых чисел

Примеры исходного кода, представленные до сих пор, были предназначены для ознакомления читателя с операциями над упакованными целыми числами при помощи команд набора AVX. Каждый пример включал простую функцию на языке ассемблера, демонстрирующую работу нескольких команд AVX с использованием экземпляров структуры `XmmVal`. Для некоторых реальных прикладных программ действительно может пригодиться небольшой набор функций, подобных тем, которые вы видели до сих пор. Однако, чтобы в полной мере использовать преимущества AVX, вам необходимо научиться кодировать функции, реализующие полные алгоритмы с использованием общих структур данных.

Примеры исходного кода в этом разделе представляют алгоритмы, которые обрабатывают массивы беззнаковых 8-битных целых чисел с использованием набора команд AVX. В первом примере вы узнаете, как определить минимальное и максимальное значения массива. Этот пример программы имеет определенную практичность, поскольку цифровые изображения часто используют массивы 8-битных целых чисел без знака для представления изображений в памяти, а многие алгоритмы обработки изображений (например, повышение контрастности) часто требуют определения минимального (самого темного) и максимального (самого светлого) пикселей изображения. Во втором примере программы показано, как вычислить среднее значение массива 8-разрядных целых чисел без знака. Это еще один пример реалистичного алгоритма, име-

ющего прямое отношение к области обработки изображений. Последние три примера исходного кода реализуют универсальные алгоритмы обработки изображений, включая преобразование пикселей, создание гистограммы и определение пороговых значений.

7.4.1. Минимальные и максимальные значения пикселей

Пример исходного кода Ch07_04, показанный в листинге 7.4, демонстрирует, как найти минимальное и максимальное значения в массиве 8-разрядных целых чисел без знака. В этом примере также объясняется, как динамически выделять выровненное пространство хранения для массива.

Листинг 7.4. Пример Ch07_04

```
//-----------------------------------------------
//               AlignedMem.h
//-----------------------------------------------

#pragma once
#include <cstdint>
#include <malloc.h>
#include <stdexcept>

class AlignedMem
{
public:
    static void* Allocate(size_t mem_size, size_t mem_alignment)
    {
        void* p = _aligned_malloc(mem_size, mem_alignment);

        if (p == NULL)
            throw std::runtime_error("Ошибка выделения памяти: AllocateAlignedMem()");

        return p;
    }

    static void Release(void* p)
    {
        _aligned_free(p);
    }

    template <typename T> static bool IsAligned(const T* p, size_t alignment)
    {
        if (p == nullptr)
            return false;

        if (((uintptr_t)p % alignment) != 0)
            return false;

        return true;
    }
};

template <class T> class AlignedArray
{
    T* m_Data;
    size_t m_Size;
```

```
public:

    AlignedArray(void) = delete;
    AlignedArray(const AlignedArray& aa) = delete;
    AlignedArray(AlignedArray&& aa) = delete;
    AlignedArray& operator = (const AlignedArray& aa) = delete;
    AlignedArray& operator = (AlignedArray&& aa) = delete;

    AlignedArray(size_t size, size_t alignment)
    {
        m_Size = size;
        m_Data = (T*)AlignedMem::Allocate(size * sizeof(T), alignment);
    }

    ~AlignedArray()
    {
        AlignedMem::Release(m_Data);
    }

    T* Data(void)           { return m_Data; }
    size_t Size(void)       { return m_Size; }

    void Fill(T val)
    {
        for (size_t i = 0; i < m_Size; i++)
            m_Data[i] = val;
    }
};

//-----------------------------------------------
//                Ch07_04.h
//-----------------------------------------------

#pragma once
#include <cstdint>

// Ch07_04.cpp
extern void Init(uint8_t* x, size_t n, unsigned int seed);
extern bool AvxCalcMinMaxU8Cpp(const uint8_t* x, size_t n, uint8_t* x_min, uint8_t* x_max);

// Ch07_04_BM.cpp
extern void AvxCalcMinMaxU8_BM(void);

// Ch07_04_.asm
extern "C" bool AvxCalcMinMaxU8_(const uint8_t* x, size_t n, uint8_t* x_min, uint8_t* x_
max);

// c_NumElements должен быть > 0 кратным 64
const size_t c_NumElements = 16 * 1024 * 1024;
const unsigned int c_RngSeedVal = 23;

//-----------------------------------------------
//                Ch07_04.cpp
//-----------------------------------------------

#include "stdafx.h"
```

```
#include <iostream>
#include <cstdint>
#include <random>
#include "Ch07_04.h"
#include "AlignedMem.h"

using namespace std;

void Init(uint8_t* x, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {5, 250};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (uint8_t)ui_dist(rng);

    // Используем известные значения min & max (для тестирования)
    x[(n / 4) * 3 + 1] = 2;
    x[n / 4 + 11] = 3;
    x[n / 2] = 252;
    x[n / 2 + 13] = 253;
    x[n / 8 + 5] = 4;
    x[n / 8 + 7] = 254;
}

bool AvxCalcMinMaxU8Cpp(const uint8_t* x, size_t n, uint8_t* x_min, uint8_t* x_max)
{
    if (n == 0 || (n & 0x3f) != 0)
        return false;

    if (!AlignedMem::IsAligned(x, 16))
        return false;

    uint8_t x_min_temp = 0xff;
    uint8_t x_max_temp = 0;

    for (size_t i = 0; i < n; i++)
    {
        uint8_t 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 AvxCalcMinMaxU8()
{
    size_t n = c_NumElements;
    AlignedArray<uint8_t> x_aa(n, 16);
    uint8_t* x = x_aa.Data();
```

```
    Init(x, n, c_RngSeedVal);
    uint8_t x_min1 = 0, x_max1 = 0;
    uint8_t x_min2 = 0, x_max2 = 0;
    bool rc1 = AvxCalcMinMaxU8Cpp(x, n, &x_min1, &x_max1);
    bool rc2 = AvxCalcMinMaxU8_(x, n, &x_min2, &x_max2);

    cout << "\nРезультаты AvxCalcMinMaxU8\n";
    cout << "rc1: " << rc1 << " x_min1: " << (int)x_min1;
    cout << " x_max1: " << (int)x_max1 << '\n';
    cout << "rc2: " << rc1 << " x_min2: " << (int)x_min2;
    cout << " x_max2: " << (int)x_max2 << '\n';
}

int main()
{
    AvxCalcMinMaxU8();
    AvxCalcMinMaxU8_BM();
    return 0;
}

;-----------------------------------------------
;                 Ch07_04_.asm
;-----------------------------------------------

; extern "C" bool AvxCalcMinMaxU8_(uint8_t* x, size_t n, uint8_t* x_min, uint8_t* x_max)
;
; Возвращает: 0 = недопустимое n или не выровнен массив, 1 = успех

        .const
        align 16
StartMinVal qword 0ffffffffffffffffh    ;начальное упакованное значение min
            qword 0ffffffffffffffffh

StartMaxVal qword 0000000000000000h     ;начальное упакованное значение max
            qword 0000000000000000h

        .code
AvxCalcMinMaxU8_ proc

; Проверка значения 'n' на допустимость
        xor eax,eax                     ;возвращаемый код ошибки
        or rdx,rdx                      ;n == 0?
        jz Done                         ;переход, если да

        test rdx,3fh                    ;кратно ли n 64?
        jnz Done                        ;переход, если нет

        test rcx,0fh                    ;выровнен ли x правильно?
        jnz Done                        ;переход, если нет

; Инициализация упакованных значений min и max
        vmovdqa xmm2,xmmword ptr [StartMinVal]
        vmovdqa xmm3,xmm2               ;xmm3:xmm2 = упакованное значение min

        vmovdqa xmm4,xmmword ptr [StartMaxVal]
        movdqa xmm5,xmm4                ;xmm5:xmm4 = упакованное значение max
```

```
; Поиск минимального и максимального значений массива
@@:     vmovdqa xmm0,xmmword ptr [rcx]      ;xmm0 = x[i + 15] : x[i]
        vmovdqa xmm1,xmmword ptr [rcx+16]   ;xmm1 = x[i + 31] : x[i + 16]
        vpminub xmm2,xmm2,xmm0
        vpminub xmm3,xmm3,xmm1              ;xmm3:xmm2 = обновленное значение min
        vpmaxub xmm4,xmm4,xmm0
        vpmaxub xmm5,xmm5,xmm1              ;xmm5:xmm4 = обновленное значение max

        vmovdqa xmm0,xmmword ptr [rcx+32]   ;xmm0 = x[i + 47] : x[i + 32]
        vmovdqa xmm1,xmmword ptr [rcx+48]   ;xmm1 = x[i + 63] : x[i + 48]
        vpminub xmm2,xmm2,xmm0
        vpminub xmm3,xmm3,xmm1              ;xmm3:xmm2 = обновленное значение min
        vpmaxub xmm4,xmm4,xmm0
        vpmaxub xmm5,xmm5,xmm1              ;xmm5:xmm4 = обновленное значение max

        add rcx,64
        sub rdx,64
        jnz @B

; Определение окончательного минимального значения
        vpminub xmm0,xmm2,xmm3              ;xmm0[127:0] = окончательные 16 значений min
        vpsrldq xmm1,xmm0,8                 ;xmm1[63:0] = xmm0[127:64]
        vpminub xmm2,xmm1,xmm0              ;xmm2[63:0] = окончательные 8 значений min
        vpsrldq xmm3,xmm2,4                 ;xmm3[31:0] = xmm2[63:32]
        vpminub xmm0,xmm3,xmm2              ;xmm0[31:0] = окончательные 4 значения min
        vpsrldq xmm1,xmm0,2                 ;xmm1[15:0] = xmm0[31:16]
        vpminub xmm2,xmm1,xmm0              ;xmm2[15:0] = окончательные 2 значения min
        vpextrw eax,xmm2,0                  ;ax = окончательные 2 значения min
        cmp al,ah
        jbe @F                              ;переход, если al <= ah
        mov al,ah                           ;al = окончательное значение min
@@:     mov [r8],al                         ;сохранение окончательного значения min

; Определение окончательного максимального значения
        vpmaxub xmm0,xmm4,xmm5              ;xmm0[127:0] = окончательные 16 значений max
        vpsrldq xmm1,xmm0,8                 ;xmm1[63:0] = xmm0[127:64]
        vpmaxub xmm2,xmm1,xmm0              ;xmm2[63:0] = окончательные 8 значений max
        vpsrldq xmm3,xmm2,4                 ;xmm3[31:0] = xmm2[63:32]
        vpmaxub xmm0,xmm3,xmm2              ;xmm0[31:0] = окончательные 4 значения max
        vpsrldq xmm1,xmm0,2                 ;xmm1[15:0] = xmm0[31:16]
        vpmaxub xmm2,xmm1,xmm0              ;xmm2[15:0] = окончательные 2 значения max
        vpextrw eax,xmm2,0                  ;ax = окончательные 2 значения max
        cmp al,ah
        jae @F                              ;переход, если al >= ah
        mov al,ah                           ;al = окончательное значение max
@@:     mov [r9],al                         ;сохранение окончательного значения max

        mov eax,1                           ;возвращаемый код успешного выполнения
Done:   ret
AvxCalcMinMaxU8_ endp
        end
```

Листинг 7.4 начинается с исходного кода заголовочного файла `AlignedMem.h`. Этот файл определяет пару простых классов C++, которые реализуют динамически выделяемые выровненные массивы. Класс `AlignedMem` – это базовый класс-оболочка для функций среды выполнения Visual C++ `_aligned_malloc`

и _aligned_free. Этот класс также включает шаблон функции-члена с именем AlignedMem::IsAligned, которая проверяет выравнивание массива в памяти. Заголовочный файл AlignedMem.h тоже определяет шаблонный класс с именем AlignedArray. Класс AlignedArray, который используется в этом и последующих примерах, содержит код, реализующий динамически выделяемые выровненные массивы и управляющий ими. Обратите внимание, что в книге этот класс содержит только минимальную функциональность для поддержки примеров исходного кода, поэтому многие стандартные конструкторы и операторы присваивания отключены.

Основной код C++ в примере Ch07_04 начинается с определения функции Init. Эта функция инициализирует массив 8-битных целых чисел без знака со случайными значениями для имитации значений пикселей изображения. Функция Init использует классы стандартной библиотеки шаблонов C++ (STL) uniform_int_distribution и default_random_engine для генерации случайных значений элементов массива. Приложение содержит список ссылок, с которыми вы можете ознакомиться, если хотите узнать больше об этих классах. Обратите внимание, что функция Init устанавливает некоторые значения пикселей в целевом массиве, чтобы иметь известные значения для целей тестирования.

Функция AvxCalcMinMaxU8Cpp реализует версию алгоритма поиска минимального и максимального значений пикселя на языке C++. Параметры этой функции включают указатель на массив, количество элементов массива и указатели на минимальное и максимальное значения. Сам алгоритм состоит из простого цикла for, который просматривает массив для нахождения минимального и максимального значений пикселей. Обратите внимание, что функция AvxCalcMinMaxU8Cpp (и ее аналог – функция на языке ассемблера AvxCalcMinMaxU8_) требует, чтобы размер массива был кратен 64. Причина этого в том, что функция языка ассемблера AvxCalcMinMaxU8_ обрабатывает по 64 пикселя во время каждой итерации цикла, как вы скоро увидите. Также обратите внимание, что исходный массив пикселей должен быть выровнен по 16-байтовой границе. Эту проверку выполняет шаблонная функция C++ AlignedMem::IsAligned.

Функция C++ AvxCalcMinMaxU8 содержит код, который инициализирует тестовый массив и вызывает две функции поиска минимального и максимального пикселей. Эта функция использует вышеупомянутый шаблонный класс AlignedArray для динамического выделения массива беззнаковых 8-битных целых чисел, выровненных по 16-байтовой границе. Аргументы конструктора для данного класса включают количество элементов массива и границу выравнивания. После команды AlignedArray<uint8_t> x_aa(n,16) функция AvxCalcMinMaxU8 получает необработанный указатель C++ на буфер массива с помощью функции-члена AlignedArray::Data(). Этот указатель передается в качестве аргумента двум функциям поиска минимума и максимума.

Функция на языке ассемблера AvxCalcMinMaxU8_ реализует тот же алгоритм, что и ее аналог на языке C++, с одним существенным отличием. Она обрабатывает элементы массива с использованием 16-байтовых пакетов, что является максимальным количеством 8-разрядных целых чисел без знака, которые могут храниться в регистре XMM. Функция AvxCalcMinMaxU8_ начинает с проверки размера аргумента n. Затем она проверяет массив x на предмет

правильного выравнивания. После проверки аргумента AvxCalcMinMaxU8_ загружает пары регистров XMM3:XMM2 и XMM5:XMM4 начальными упакованными минимальным и максимальным значениями соответственно. Это позволяет циклу обработки одновременно отслеживать 32 минимальных и максимальных значения.

Во время каждой итерации цикла обработки функция AvxCalcMinMaxU8_ загружает 32 пиксельных значения в регистровую пару XMM1:XMM0, используя команды `vmovdqa xmm0,xmmword ptr [rcx]` и `vmovdqa xmm1,xmmword ptr [rcx+16]`. Следующие две команды, `vpminub xmm2,xmm2,xmm0` и `vpminub xmm3,xmm3,xmm1`, обновляют текущие минимумы пикселей в паре регистров XMM3:XMM2. Следом за ними команды `vpmaxub` обновляют текущие максимумы пикселей в паре регистров XMM5:XMM4. Другая последовательность команд `vmovdqa`, `vpminub` и `vpmaxub` обрабатывает следующую группу из 32 пикселей. Обработка нескольких элементов данных во время каждой итерации цикла уменьшает количество выполняемых команд перехода и часто приводит к более быстрому выполнению кода. Этот метод оптимизации обычно называется *развертыванием* (или *расшиванием*) *цикла*. Вы узнаете больше о методах развертывания цикла и оптимизации команд перехода в главе 15.

После выполнения цикла обработки количество значений в парах регистров XMM3: XMM2 и XMM5: XMM4 должно быть уменьшено, чтобы получить окончательные минимальное и максимальное значения. Команда `vpminub xmm0,xmm2,xmm3` уменьшает количество минимальных значений пикселей с 32 до 16. Следующая команда, `vpsrldq xmm1,xmm0,8`, сдвигает вправо содержимое XMM0 на 8 байт и сохраняет результат в регистре XMM1 (т. е. XMM1[63:0] = XMM0[127:64]). Это облегчает использование последующей команды `vpminub xmm2,xmm1,xmm0`, которая уменьшает количество минимальных значений с 16 до 8. Затем используются еще две последовательности команд `vpsrldq` и `vpminub`, чтобы уменьшить количество минимумов пикселей до двух, как показано на рис. 7.3. Команда `vpextrw eax,xmm2,0` извлекает младшее слово (XMM2 [15:0]) из регистра XMM2 и сохраняет его в младшее слово регистра EAX (или регистра AX). Команды `cmp al,ah, jbe` и `mov al,ah` определяют окончательное минимальное значение пикселя. AvxCalcMinMaxU8_ использует аналогичный метод уменьшения количества значений для определения максимального значения пикселя.

Вот результат выполнения примера `Ch07_04`:

```
Результаты AvxCalcMinMaxU8
rc1: 1 x_min1: 2 x_max1: 254
rc2: 1 x_min2: 2 x_max2: 254

Запуск функции тестирования AvxCalcMinMaxU8_BM - подождите
Результаты тестирования сохранены в файл Ch07_04_AvxCalcMinMaxU8_BM_CHROMIUM.csv
```

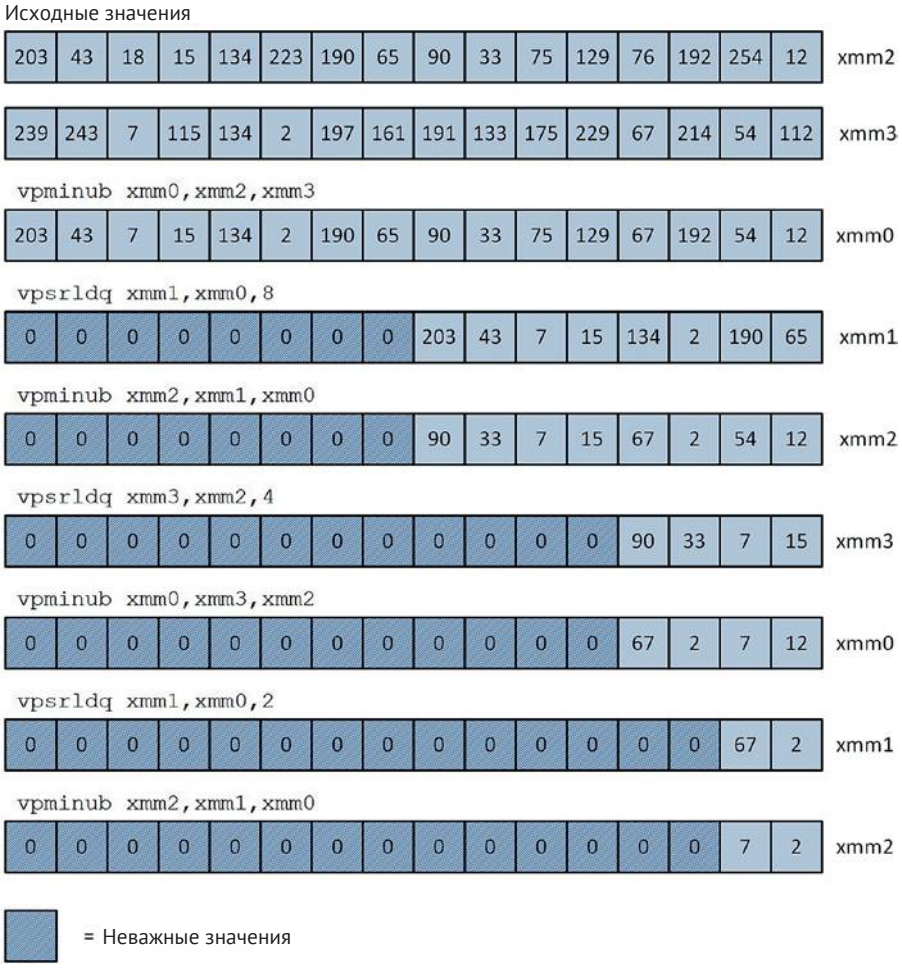


Рис. 7.3. Уменьшение количества минимальных значений пикселей с помощью команд `vpminub` и `vpsrldq`

В табл. 7.1 показаны некоторые результаты измерения быстродействия функций `AvxCalcMinMaxU8` и `AvxCalcMinMaxU8` с использованием нескольких различных процессоров Intel. Эти измерения были выполнены с использованием процедуры, описанной в главе 6. Исходный код для измерения быстродействия для этого и последующих примеров не показан, но включен в пакеты загрузки главы.

Таблица 7.1. Среднее время выполнения (мс) функций поиска минимального и максимального значений, размер массива = 16 МБ

Процессор	AvxCalcMinMaxU8Cpp	AvxCalcMinMaxU8_
i7-4790S	17642	1007
i9-7900X	13638	874
i7-8700K	12622	721

7.4.2. Средняя интенсивность пикселей

Следующий пример, Ch07_05, содержит код, вычисляющий среднее арифметическое массива 8-битовых целых чисел без знака. В этом примере также показано, как *распаковывают* (расширяют, увеличивают разрядность) упакованных целых чисел без знака. Исходный код примера показан в листинге 7.5.

Листинг 7.5. Пример Ch07_05

```
//-----------------------------------------------
//                 Ch07_05.h
//-----------------------------------------------

#pragma once
#include <cstdint>

// Ch07_05.cpp
extern void Init(uint8_t* x, size_t n, unsigned int seed);
extern bool AvxCalcMeanU8Cpp(const uint8_t* x, size_t n, int64_t* sum_x, double* mean);

// Ch07_05_BM.cpp
extern void AvxCalcMeanU8_BM(void);

// Ch07_05_.asm
extern "C" bool AvxCalcMeanU8_(const uint8_t* x, size_t n, int64_t* sum_x, double* mean);

// Общие константы
const size_t c_NumElements = 16 * 1024 * 1024;      // Должно быть кратно 64
const size_t c_NumElementsMax = 64 * 1024 * 1024;   // Для предотвращения переполнения
const unsigned int c_RngSeedVal = 29;

//-----------------------------------------------
//                Ch07_05.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include "Ch07_05.h"
#include "AlignedMem.h"

using namespace std;

extern "C" size_t g_NumElementsMax = c_NumElementsMax; // Используется в части .asm кода

void Init(uint8_t* x, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {0, 255};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (uint8_t)ui_dist(rng);
}

bool AvxCalcMeanU8Cpp(const uint8_t* x, size_t n, int64_t* sum_x, double* mean_x)
```

```
{
    if (n == 0 || n > c_NumElementsMax)
        return false;

    if ((n % 64) != 0)
        return false;

    if (!AlignedMem::IsAligned(x, 16))
    return false;

    int64_t 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 AvxCalcMeanU8()
{
    const size_t n = c_NumElements;
    AlignedArray<uint8_t> x_aa(n, 16);
    uint8_t* x = x_aa.Data();

    Init(x, n, c_RngSeedVal);

    bool rc1, rc2;
    int64_t sum_x1 = -1, sum_x2 = -1;
    double mean_x1 = -1, mean_x2 = -1;

    rc1 = AvxCalcMeanU8Cpp(x, n, &sum_x1, &mean_x1);
    rc2 = AvxCalcMeanU8_(x, n, &sum_x2, &mean_x2);

    cout << "\nРезультаты MmxCalcMeanU8\n";
    cout << fixed << setprecision(6);
    cout << "rc1: " << rc1 << " ";
    cout << "sum_x1: " << sum_x1 << " ";
    cout << "mean_x1: " << mean_x1 << '\n';
    cout << "rc2: " << rc2 << " ";
    cout << "sum_x2: " << sum_x2 << " ";
    cout << "mean_x2: " << mean_x2 << '\n';
}

int main()
{
    AvxCalcMeanU8();
    AvxCalcMeanU8_BM();
    return 0;
}

;-----------------------------------------------
;                Ch07_05.asm
;-----------------------------------------------
```

```
        include <MacrosX86-64-AVX.asmh>
        extern g_NumElementsMax:qword

; extern "C" bool AvxCalcMeanU8_(const Uint8* x, size_t n, int64_t* sum_x, double* mean);
;
; Возвращает: 0 = недопустимый n или невыровненный массив, 1 = успех

        .code
AvxCalcMeanU8_ proc frame
        _CreateFrame CM_,0,64
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9
        _EndProlog

; Проверка аргументов функции
        xor eax,eax                     ;возвращаемый код ошибки
        or rdx,rdx
        jz Done                         ;переход, если n == 0

        cmp rdx,[g_NumElementsMax]
        jae Done                        ;переход, если n > NumElementsMax

        test rdx,3fh
        jnz Done                        ;переход, если (n % 64) != 0

        test rcx,0fh
        jnz Done                        ;переход, если x не выровнен правильно

; Инициализация
        mov r10,rdx             ;сохранение n для дальнейшего использования
        add rdx,rcx             ;rdx = конец массива
        vpxor xmm8,xmm8,xmm8    ;xmm8 = упакованные промежуточные суммы (4 двойных слова)
        vpxor xmm9,xmm9,xmm9    ;xmm9 = packed zero for promotions

; Расширение 32 значений пикселей из байтов в слова, затем суммирование слов
@@:     vmovdqa xmm0,xmmword ptr [rcx]
        vmovdqa xmm1,xmmword ptr [rcx+16]   ;xmm1:xmm0 = 32 пикселя
        vpunpcklbw xmm2,xmm0,xmm9           ;xmm2 = 8 слов
        vpunpckhbw xmm3,xmm0,xmm9           ;xmm3 = 8 слов
        vpunpcklbw xmm4,xmm1,xmm9           ;xmm4 = 8 слов
        vpunpckhbw xmm5,xmm1,xmm9           ;xmm5 = 8 слов
        vpaddw xmm0,xmm2,xmm3
        vpaddw xmm1,xmm4,xmm5
        vpaddw xmm6,xmm0,xmm1               ;xmm6 = упакованные суммы (8 слов)

; Расширение других 32 значений пикселей из байтов в слова, затем суммирование слов
        vmovdqa xmm0,xmmword ptr [rcx+32]
        vmovdqa xmm1,xmmword ptr [rcx+48]   ;xmm1:xmm0 = 32 пикселя
        vpunpcklbw xmm2,xmm0,xmm9           ;xmm2 = 8 слов
        vpunpckhbw xmm3,xmm0,xmm9           ;xmm3 = 8 слов
        vpunpcklbw xmm4,xmm1,xmm9           ;xmm4 = 8 слов
        vpunpckhbw xmm5,xmm1,xmm9           ;xmm5 = 8 слов
        vpaddw xmm0,xmm2,xmm3
        vpaddw xmm1,xmm4,xmm5
        vpaddw xmm7,xmm0,xmm1               ;xmm7 = упакованные суммы (8 слов)
```

```
; Расширение упакованных сумм до двойных слов, затем обновление сумм двойных слов
        vpaddw xmm0,xmm6,xmm7               ;xmm0 = упакованные суммы (8 слов)
        vpunpcklwd xmm1,xmm0,xmm9           ;xmm1 = упакованные суммы (4 двойных слова)
        vpunpckhwd xmm2,xmm0,xmm9           ;xmm2 = упакованные суммы (4 двойных слова)
        vpaddd xmm8,xmm8,xmm1
        vpaddd xmm8,xmm8,xmm2
        add rcx,64                          ;rcx = следующий блок 64 байта
        cmp rcx,rdx
        jne @B                              ;повторение цикла до завершения

; Вычисление окончательного sum_x (vpextrd дополняет нулями извлеченное двойное слово до
64 битов)
        vpextrd eax,xmm8,0                  ;rax = частичная сумма 0
        vpextrd edx,xmm8,1                  ;rdx = частичная сумма 1
        add rax,rdx
        vpextrd ecx,xmm8,2                  ;rcx = частичная сумма 2
        vpextrd edx,xmm8,3                  ;rdx = частичная сумма 3
        add rax,rcx
        add rax,rdx
        mov [r8],rax                        ;сохранение sum_x

; Compute mean value
        vcvtsi2sd xmm0,xmm0,rax             ;xmm0 = sum_x (DPFP)
        vcvtsi2sd xmm1,xmm1,r10             ;xmm1 = n (DPFP)
        vdivsd xmm2,xmm0,xmm1               ;вычисление mean = sum_x / n
        vmovsd real8 ptr [r9],xmm2          ;сохранение среднего
        mov eax,1                           ;возвращаемый код успешного выполнения

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9
        _DeleteFrame
        ret
AvxCalcMeanU8_ endp
        end
```

Организация кода C++ в примере `Ch07_05` отчасти похожа на предыдущий пример. Функция C++ `AvxCalcMeanU8Cpp` использует простой цикл суммирования и скалярную арифметику для вычисления среднего значения массива 8-разрядных целых чисел без знака. Как и в предыдущем примере, количество элементов массива должно быть целым числом, кратным 64, а исходный массив должен быть выровнен по 16-байтовой границе. Обратите внимание, что функция `AvxCalcMeanU8Cpp` также следит, чтобы количество элементов массива не превышало `c_NumElementsMax`. Это ограничение размера позволяет функции языка ассемблера `AvcCalcMeanU8_` выполнять свои вычисления с использованием упакованных двойных слов без опасений, что возникнет арифметическое переполнение. Оставшийся код C++, показанный в листинге 7.5, выполняет инициализацию тестового массива и передает результаты в `cout`.

Функция на языке ассемблера `AvxCalcMeanU8_` начинает с выполнения тех же проверок размера массива, что и ее аналог C++. Адрес массива также проверяется на правильность выравнивания. После проверки аргумента `AvxCalcMeanU8_` выполняет необходимые инициализации. Команда `add rdx,rcx` вычисляет адрес первого байта после конца массива. Функция `AvxCalcMeanU8_` использует этот адрес вместо счетчика для завершения цикла обработки. Затем регистр

XMM8 инициализируется нулями. Цикл обработки использует этот регистр для хранения промежуточных упакованных сумм двойных слов.

Каждая итерация цикла обработки начинается с двух команд `vmovdqa`, загружающих 32 байтовых значения без знака в регистры XMM1:XMM0. Затем значения пикселей распаковываются в слова с помощью `vpunpcklbw` (распаковать младшие разряды) и `vpunpckhbw` (распаковать старшие разряды). Эти команды чередуют байтовые значения, содержащиеся в двух исходных операндах, чтобы сформировать значения слов, как показано на рис. 7.4. Обратите внимание, что регистр XMM9 заполнен нулями, то есть байтовые значения без знака расширяются нулем во время распаковки в слова. Затем серия команд `vpaddw` суммирует значения упакованных беззнаковых слов. Функция `AvxCalcMeanU8_` обрабатывает другой блок из 32 пикселей, используя ту же последовательность команд. Суммы беззнаковых слов в регистрах XMM6 и XMM7 затем суммируются с помощью команды `vpaddw`, увеличиваются по размеру до двойных слов с помощью `vpunpcklwd` и `vpunpckhwd` и прибавляются к промежуточным упакованным суммам двойных слов в регистре XMM8. Эта последовательность команд схематически показана на рис. 7.4.

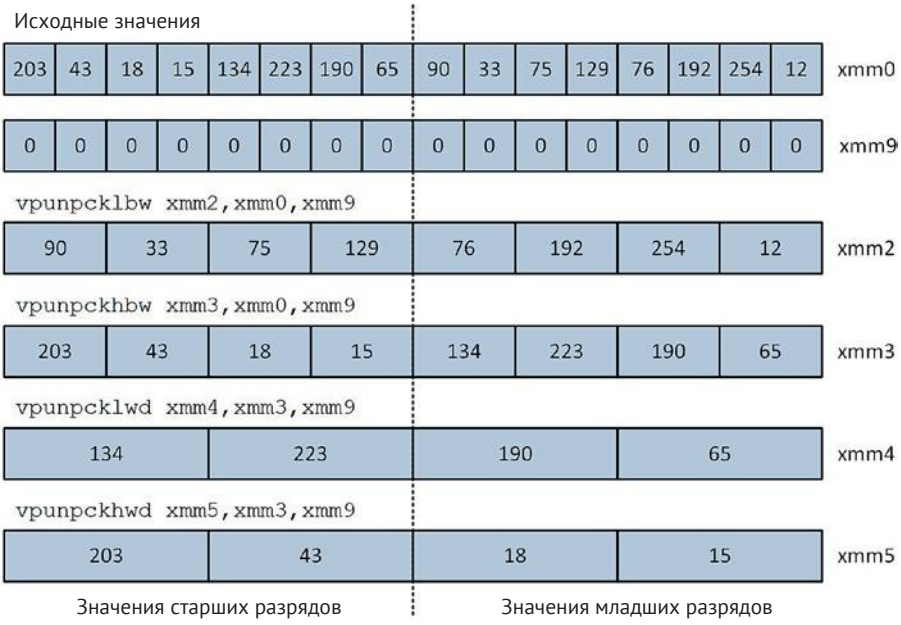


Рис. 7.4. Выполнение команд `vpunpck[h|l]bw` и `vpunpck[h|l]wd`

После завершения рабочего цикла промежуточные суммы двойных слов в регистре XMM8 суммируются для получения окончательной суммы пикселей. Функция использует несколько команд `vpextrd` для копирования каждого значения двойного слова из XMM8 в регистр общего назначения. Обратите внимание, что эта команда использует непосредственный операнд, чтобы указать, какое значение элемента копировать. После вычисления суммы пикселей `AvxCalcMeanU8_` вычисляет окончательное среднее значение пикселей, используя простую скалярную арифметику. Ниже показаны результаты выполнения примера `Ch07_05`:

```
Результаты AvxCalcMeanU8
rc1: 1 sum_x1: 2139023922 mean_x1: 127.495761
rc2: 1 sum_x2: 2139023922 mean_x2: 127.495761
Запуск функции тестирования AvxCalcMeanU8_BM - подождите
Результаты тестирования сохранены в файл Ch07_05_AvxCalcMeanU8_BM_CHROMIUM.csv
```

В табл. 7.2 показаны некоторые результаты измерения быстродействия кода из примера Ch07_05.

Таблица 7.2. Среднее время выполнения (мс) примера исходного кода Ch07_05, размер массива = 16 МБ

Процессор	AvxCalcMeanU8Cpp	AvxCalcMeanU8_
i7-4790S	7103	1063
i9-7900X	6332	1048
i7-8700K	5870	861

7.4.3. Преобразования пикселей

Чтобы реализовать определенные алгоритмы обработки изображений, часто бывает необходимо преобразовать пиксели 8-битного изображения в градациях серого из целого числа без знака в значения с плавающей запятой одинарной точности и наоборот. Следующий пример, Ch07_06, показывает, как это сделать с помощью набора команд AVX. В листинге 7.6 показан исходный код примера Ch07_06.

Листинг 7.6. Пример Ch07_06

```
//-----------------------------------------------
// Ch07_06.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cstdint>
#include <random>
#include "AlignedMem.h"

using namespace std;

// Ch07_06_Misc.cpp
extern uint32_t ConvertImgVerify(const float* src1, const float* src2, uint32_t num_
pixels);
extern uint32_t ConvertImgVerify(const uint8_t* src1, const uint8_t* src2, uint32_t num_
pixels);

// Ch07_06_.asm
extern "C" bool ConvertImgU8ToF32_(float* des, const uint8_t* src, uint32_t num_pixels);
extern "C" bool ConvertImgF32ToU8_(uint8_t* des, const float* src, uint32_t num_pixels);

extern "C" uint32_t c_NumPixelsMax = 16777216;
```

```
template <typename T> void Init(T* x, size_t n, unsigned int seed, T scale)
{
    uniform_int_distribution<> ui_dist {0, 255};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
    {
        T temp = (T)ui_dist(rng);
        x[i] = (scale == 1) ? temp : temp / scale;
    }
}

bool ConvertImgU8ToF32Cpp(float* des, const uint8_t* src, uint32_t num_pixels)
{
    // Проверка значения num_pixels
    if ((num_pixels == 0) || (num_pixels > c_NumPixelsMax))
    return false;
    if ((num_pixels % 32) != 0)
    return false;

    // Проверка выравнивания источника и приемника по 16-байтовой границе
    if (!AlignedMem::IsAligned(src, 16))
    return false;
    if (!AlignedMem::IsAligned(des, 16))
    return false;

    // Конвертация изображения
    for (uint32_t i = 0; i < num_pixels; i++)
    des[i] = src[i] / 255.0f;

    return true;
}

bool ConvertImgF32ToU8Cpp(uint8_t* des, const float* src, uint32_t num_pixels)
{
    // Проверка значения num_pixels
    if ((num_pixels == 0) || (num_pixels > c_NumPixelsMax))
        return false;
    if ((num_pixels % 32) != 0)
        return false;

    // Проверка выравнивания источника и приемника по 16-байтовой границе
    if (!AlignedMem::IsAligned(src, 16))
        return false;
    if (!AlignedMem::IsAligned(des, 16))
        return false;

    for (uint32_t 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_t)(src[i] * 255.0f);
    }
    return true;
}
```

```
void ConvertImgU8ToF32(void)
{
    const uint32_t num_pixels = 1024;
    AlignedArray<uint8_t> src_aa(num_pixels, 16);
    AlignedArray<float> des1_aa(num_pixels, 16);
    AlignedArray<float> des2_aa(num_pixels, 16);
    uint8_t* src = src_aa.Data();
    float* des1 = des1_aa.Data();
    float* des2 = des2_aa.Data();

    Init(src, num_pixels, 12, (uint8_t)1);

    bool rc1 = ConvertImgU8ToF32Cpp(des1, src, num_pixels);
    bool rc2 = ConvertImgU8ToF32_(des2, src, num_pixels);

    if (!rc1 || !rc2)
    {
        cout << "Некорректное возвращаемое значение - ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        return;
    }

    uint32_t num_diff = ConvertImgVerify(des1, des2, num_pixels);
    cout << "\nРезультаты ConvertImgU8ToF32\n";
    cout << " num_pixels = " << num_pixels << '\n';
    cout << " num_diff = " << num_diff << '\n';
}

void ConvertImgF32ToU8(void)
{
    const uint32_t num_pixels = 1024;
    AlignedArray<float> src_aa(num_pixels, 16);
    AlignedArray<uint8_t> des1_aa(num_pixels, 16);
    AlignedArray<uint8_t> des2_aa(num_pixels, 16);
    float* src = src_aa.Data();
    uint8_t* des1 = des1_aa.Data();
    uint8_t* des2 = des2_aa.Data();

    // Инициализация пиксельного буфера источника. Первые несколько значений
    // заранее известны и применяются для нужд тестирования.

    Init(src, num_pixels, 20, 1.0f);

    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;

    bool rc1 = ConvertImgF32ToU8Cpp(des1, src, num_pixels);
    bool rc2 = ConvertImgF32ToU8_(des2, src, num_pixels);
```

```
    if (!rc1 || !rc2)
    {
        cout << "Некорректное возвращаемое значение - ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        return;
    }

    uint32_t num_diff = ConvertImgVerify(des1, des2, num_pixels);
    cout << "\nРезультаты ConvertImgF32ToU8\n";
    cout << " num_pixels = " << num_pixels << '\n';
    cout << " num_diff = " << num_diff << '\n';
    }

int main()
{
    ConvertImgU8ToF32();
    ConvertImgF32ToU8();
    return 0;
}

;-----------------------------------------------
;               Ch07_06.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>
        include <cmpequ.asmh>

                    .const
                    align 16
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

        extern c_NumPixelsMax:dword

; extern "C" bool ConvertImgU8ToF32_(float* des, const uint8_t* src, uint32_t num_pixels)

        .code
ConvertImgU8ToF32_ proc frame
        _CreateFrame U2F_,0,160
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

; Проверка значения num_pixels и выравнивания пиксельного буфера
        xor eax,eax                     ;возвращаемый код ошибки
        or r8d,r8d
        jz Done                         ;переход, если num_pixels равен нулю
        cmp r8d,[c_NumPixelsMax]
        ja Done                         ;переход, если num_pixels слишком велик
        test r8d,1fh
        jnz Done                        ;переход, если num_pixels % 32 != 0
        test rcx,0fh
        jnz Done                        ;переход, если приемник не выровнен
        test rdx,0fh
        jnz Done                        ;переход, если источник не выровнен
```

```
; Инициализация регистров рабочего цикла
        shr r8d,5                                ;количество пиксельных блоков
        vmovaps xmm6,xmmword ptr [Uint8ToFloat] ;xmm6 = упакованный 255.0f
        vpxor xmm7,xmm7,xmm7                     ;xmm7 = упакованный 0

; Load the next block of 32 pixels
@@:     vmovdqa xmm0,xmmword ptr [rdx]      ;xmm0 = 16 пикселей (x[i+15]:x[i])
        vmovdqa xmm1,xmmword ptr [rdx+16]   ;xmm8 = 16 пикселей (x[i+31]:x[i+16])

; Распаковка значений пикселей в xmm0 из беззнаковых байтов в беззнаковые двойные слова
        vpunpcklbw xmm2,xmm0,xmm7
        vpunpckhbw xmm3,xmm0,xmm7
        vpunpcklwd xmm8,xmm2,xmm7
        vpunpckhwd xmm9,xmm2,xmm7
        vpunpcklwd xmm10,xmm3,xmm7
        vpunpckhwd xmm11,xmm3,xmm7          ;xmm11:xmm8 = 16 двойных слов

; Распаковка значений пикселей в xmm1 из беззнаковых байтов в беззнаковые двойные слова
        vpunpcklbw xmm2,xmm1,xmm7
        vpunpckhbw xmm3,xmm1,xmm7
        vpunpcklwd xmm12,xmm2,xmm7
        vpunpckhwd xmm13,xmm2,xmm7
        vpunpcklwd xmm14,xmm3,xmm7
        vpunpckhwd xmm15,xmm3,xmm7          ;xmm15:xmm12 = 16 двойных слов

; Конвертация значений пикселей из двойных слов в SPFP (одинарная точность, плавающая
запятая)
        vcvtdq2ps xmm8,xmm8
        vcvtdq2ps xmm9,xmm9
        vcvtdq2ps xmm10,xmm10
        vcvtdq2ps xmm11,xmm11               ;xmm11:xmm8 = 16 значений пикселей SPFP
        vcvtdq2ps xmm12,xmm12
        vcvtdq2ps xmm13,xmm13
        vcvtdq2ps xmm14,xmm14
        vcvtdq2ps xmm15,xmm15               ;xmm15:xmm12 = 16 значений пикселей SPFP

; Нормализация значений пикселей к диапазону [0.0, 1.0] и сохранение результатов
        vdivps xmm0,xmm8,xmm6
        vmovaps xmmword ptr [rcx],xmm0      ;сохранение пикселей 0-3
        vdivps xmm1,xmm9,xmm6
        vmovaps xmmword ptr [rcx+16],xmm1   ;сохранение пикселей 4-7
        vdivps xmm2,xmm10,xmm6
        vmovaps xmmword ptr [rcx+32],xmm2   ;сохранение пикселей 8-11
        vdivps xmm3,xmm11,xmm6
        vmovaps xmmword ptr [rcx+48],xmm3   ;сохранение пикселей 12-15

        vdivps xmm0,xmm12,xmm6
        vmovaps xmmword ptr [rcx+64],xmm0   ;сохранение пикселей 16-19
        vdivps xmm1,xmm13,xmm6
        vmovaps xmmword ptr [rcx+80],xmm1   ;сохранение пикселей 20-23
        vdivps xmm2,xmm14,xmm6
        vmovaps xmmword ptr [rcx+96],xmm2   ;сохранение пикселей 24-27
        vdivps xmm3,xmm15,xmm6
        vmovaps xmmword ptr [rcx+112],xmm3  ;сохранение пикселей 28-31

        add rdx,32                          ;обновление указателя источника
```

```
        add rcx,128                     ;обновление указателя приемника
        sub r8d,1
        jnz @B                          ;повтор до завершения
        mov eax,1                       ;возвращаемый код успешного выполнения

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame
        ret

ConvertImgU8ToF32_ endp

; extern "C" bool ConvertImgF32ToU8_(uint8_t* des, const float* src, uint32_t num_pixels)

ConvertImgF32ToU8_ proc frame
        _CreateFrame F2U_,0,96
        _SaveXmmRegs xmm6,xmm7,xmm12,xmm13,xmm14,xmm15
        _EndProlog

; Проверка значения num_pixels и правильности выравнивания пиксельного буфера
        xor eax,eax                     ;возвращаемый код ошибки
        or r8d,r8d
        jz Done                         ;переход, если num_pixels равен нулю
        cmp r8d,[c_NumPixelsMax]
        ja Done                         ;переход, если num_pixels слишком велик
        test r8d,1fh
        jnz Done                        ;переход, если num_pixels % 32 != 0
        test rcx,0fh
        jnz Done                        ;переход, если приемник не выровнен
        test rdx,0fh
        jnz Done                        ;переход, если источник не выровнен

; Чтение необходимых упакованных констант в регистры
        vmovaps xmm13,xmmword ptr [FloatToUint8Scale]   ;xmm13 = упакованное 255.0
        vmovaps xmm14,xmmword ptr [FloatToUint8Min]     ;xmm14 = упакованное 0.0
        vmovaps xmm15,xmmword ptr [FloatToUint8Max]     ;xmm15 = упакованное 1.0

        shr r8d,4                       ;количество пиксельных блоков
LP1:    mov r9d,4                       ;количество пиксельных квартетов в блоке

; Конвертация 16 пиксельных значений в uint8_t
LP2:    vmovaps xmm0,xmmword ptr [rdx]  ;xmm0 = следующий пиксельный квартет
        vcmpps xmm1,xmm0,xmm14,CMP_LT   ;сравнение пикселей с 0.0
        vandnps xmm2,xmm1,xmm0          ;приведение пикселей < 0.0 к 0.0

        vcmpps xmm3,xmm2,xmm15,CMP_GT   ;сравнение пикселей с 1.0
        vandps xmm4,xmm3,xmm15          ;усечение пикселей > 1.0 до 1.0
        vandnps xmm5,xmm3,xmm2          ;xmm5 = пиксели <= 1.0
        vorps xmm6,xmm5,xmm4            ;xmm6 = окончательные усеченные пиксели
        vmulps xmm7,xmm6,xmm13          ;xmm7 = пиксели с плавающей запятой [0.0, 255.0]

        vcvtps2dq xmm0,xmm7             ;xmm0 = пиксельные двойные слова [0, 255]
        vpackusdw xmm1,xmm0,xmm0        ;xmm1[63:0] = пиксельные слова
        vpackuswb xmm2,xmm1,xmm1        ;xmm2[31:0] = пиксельные байты

; Сохранение текущего квартета пиксельных байтов
        vpextrd eax,xmm2,0              ;eax = новый пиксельный квартет
```

```
        vpsrldq xmm12,xmm12,4           ;выравнивание xmm12 для нового квартета
        vpinsrd xmm12,xmm12,eax,3       ;xmm12[127:96] = новый квартет

        add rdx,16                      ;обновление указателя на источник
        sub r9d,1
        jnz LP2                         ;повтор до завершения

; Save the current byte pixel block (16 pixels)
        vmovdqa xmmword ptr [rcx],xmm12 ;сохранение текущего пиксельного блока
        add rcx,16                      ;обновление указателя на приемник
        sub r8d,1
        jnz LP1                         ;повтор до завершения
        mov eax,1                       ;возвращаемый код успешного выполнения

Done:   _RestoreXmmRegs xmm6,xmm7,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame
        ret
ConvertImgF32ToU8_ endp
        end
```

Код C++ в листинге 7.6 относительно прост. Функция `ConvertImgU8ToF32Cpp` содержит код, преобразующий значения пикселей из `uint8_t [0,255]` в числа с плавающей запятой одинарной точности `[0.0,1.0]`. Эта функция содержит простой цикл `for`, который вычисляет `des[i]=src[i]/255.0`. Аналогичная функция `ConvertImgF32ToU8Cpp` выполняет обратную операцию. Обратите внимание, что эта функция обрезает любые значения пикселей больше 1,0 или меньше 0,0 перед выполнением преобразования чисел с плавающей запятой в `uint8_t`. Функции `ConvertImgU8ToF32` и `ConvertImgF32ToU8` содержат код, который инициализирует тестовые массивы и выполняет процедуры преобразования C++ и языка ассемблера. Обратите внимание, что последняя функция инициализирует первые несколько записей исходного буфера известными значениями, чтобы продемонстрировать вышеупомянутую операцию отсечения.

Рабочий цикл функции на языке ассемблера `ConvertImgU8ToF32_` во время каждой итерации преобразует 32 пиксельных значения из `uint8_t` (или байта) в числа с плавающей запятой одинарной точности. Процесс преобразования начинается с увеличения разрядности упакованных пикселей из беззнакового байта в беззнаковое целое двойное слово с использованием серии команд `vpunpck[h|l]bw` и `vpunpck[h|l]wd`. Затем полученные двойные слова преобразуются в значения с плавающей запятой одинарной точности с помощью команды `vcvtdq2ps` (преобразование упакованных целых чисел двойного слова в упакованные числа с плавающей запятой одинарной точности). Результирующие упакованные значения с плавающей запятой нормализуются до [0.0, 1.0] и сохраняются в приемном буфере.

Функция на языке ассемблера `ConvertImgF32ToU8_` выполняет преобразование упакованных чисел с плавающей запятой одинарной точности в упакованные байты без знака. Внутренний цикл (начиная с метки `LP2`) этой функции преобразования использует команды `vcmpps xmm1,xmm0,xmm14,CMP_LT`, `vcmpps xmm3,xmm2,xmm15,CMP_GT` и операции двоичной логики для обрезки любых значений пикселей меньше 0,0 или больше 1,0. Этот метод наглядно представлен на рис. 7.5. Команда `vcvtps2dq xmm0,xmm7` преобразует четыре значения с плава-

ющей запятой одинарной точности в XMM7 в целочисленные двойные слова и сохраняет результаты в регистре XMM0. Следующие две команды, `vpackusdw xmm1,xmm0,xmm0` и `vpackuswb xmm2,xmm1,xmm1`, уменьшают размер упакованных целочисленных двойных слов до упакованных байтов без знака. После выполнения команды `vpackuswb` регистр XMM2 [31:0] содержит четыре упакованных байтовых значения без знака. Этот квартет пикселей затем копируется в XMM12 [127: 96] с использованием последовательности команд `vpextrd eax,xmm2,0`, `vpsrldq xmm12,xmm12,4` и `vpinsrd xmm12,xmm12,eax,3`. Используемые здесь команды `vpinsrd` (вставить двойное слово) копируют значение двойного слова в регистре EAX в позицию элемента двойного слова 3 в регистре XMM12 (или XMM12[127: 96]).

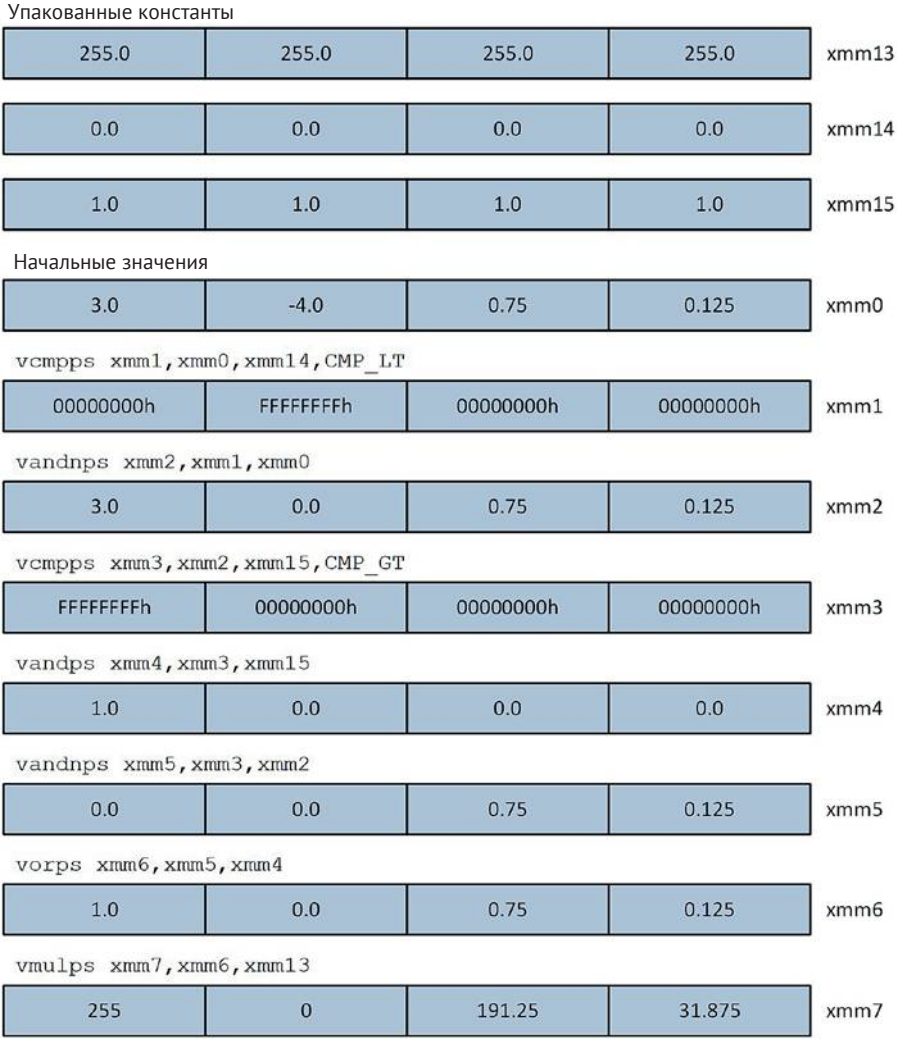


Рис. 7.5. Иллюстрация процесса усечения чисел с плавающей запятой, применяемого в функции `ConvertImgF32ToU8_`

Упомянутый выше процесс преобразования выполняется в течение четырех итераций вложенного цикла. После завершения этого цикла XMM12 содержит 16 байтовых пиксельных значений без знака. Затем этот блок пикселей сохраняется в приемный буфер с помощью команды `vmovdqa xmmword ptr [rcx],xmm12`. Внешний цикл повторяется до тех пор, пока не будут преобразованы все пиксели. Ниже приведен результат выполнения кода примера `Ch07_06`.

```
Результаты ConvertImgU8ToF32
  num_pixels = 1024
  num_diff = 0

Результаты ConvertImgF32ToU8
  num_pixels = 1024
  num_diff = 0
```

7.4.4. Гистограммы изображений

Многие алгоритмы обработки изображений нуждаются в гистограммах значений яркости пикселей изображения. На рис. 7.6 приведен пример изображения в оттенках серого и его гистограмма. В примере исходного кода `Ch07_07` показано, как построить гистограмму значений яркости пикселей для изображения, содержащего 8-битные значения пикселей в градациях серого. В этом примере также объясняется, как использовать стек в функции на языке ассемблера для хранения промежуточных результатов. В листинге 7.7 показан исходный код примера `Ch07_07`.

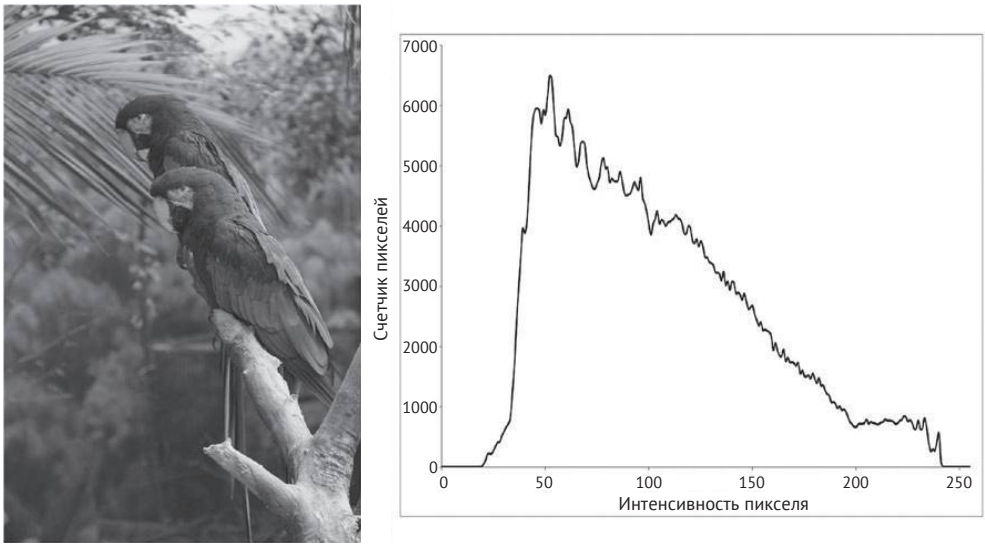


Рис. 7.6. Пример изображения в оттенках серого и его гистограмма

Листинг 7.7. Пример Ch07_07

```
//-----------------------------------------------
//              Ch07_07.h
//-----------------------------------------------

#pragma once
#include <cstdint>

// Ch07_07.cpp
extern bool AvxBuildImageHistogramCpp(uint32_t* histo, const uint8_t* pixel_buff, uint32_t
num_pixels);

// Ch07_07_.asm
// Функции, определенные в Sse64ImageHistogram_.asm
extern "C" bool AvxBuildImageHistogram_(uint32_t* histo, const uint8_t* pixel_buff,
uint32_t
num_pixels);

// Ch07_07_BM.cpp
extern void AvxBuildImageHistogram_BM(void);

//-----------------------------------------------
//              Ch07_07.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include "Ch07_07.h"
#include "AlignedMem.h"
#include "ImageMatrix.h"

using namespace std;

extern "C" uint32_t c_NumPixelsMax = 16777216;

bool AvxBuildImageHistogramCpp(uint32_t* histo, const uint8_t* pixel_buff, uint32_t num_
pixels)
{
    // Проверка num_pixels
    if ((num_pixels == 0) || (num_pixels > c_NumPixelsMax))
        return false;

    if (num_pixels % 32 != 0)
        return false;

    // Проверка выравнивания histo по 16 байтам
    if (!AlignedMem::IsAligned(histo, 16))
        return false;

    // Проверка выравнивания pixel_buff по 16 байтам
    if (!AlignedMem::IsAligned(pixel_buff, 16))
        return false;
```

```
    // Построение гистограммы
    memset(histo, 0, 256 * sizeof(uint32_t));

    for (uint32_t i = 0; i < num_pixels; i++)
        histo[pixel_buff[i]]++;

    return true;
}

void AvxBuildImageHistogram(void)
{
    const wchar_t* image_fn = L"..\\Ch07_Data\\TestImage1.bmp";
    const wchar_t* csv_fn = L"Ch07_07_AvxBuildImageHistogram_Histograms.csv";

    ImageMatrix im(image_fn);
    uint32_t num_pixels = im.GetNumPixels();
    uint8_t* pixel_buff = im.GetPixelBuffer<uint8_t>();
    AlignedArray<uint32_t> histo1_aa(256, 16);
    AlignedArray<uint32_t> histo2_aa(256, 16);

    bool rc1 = AvxBuildImageHistogramCpp(histo1_aa.Data(), pixel_buff, num_pixels);
    bool rc2 = AvxBuildImageHistogram_(histo2_aa.Data(), pixel_buff, num_pixels);

    cout << "\nРезультаты AvxBuildImageHistogram\n";

    if (!rc1 || !rc2)
    {
        cout << "Код ошибки: ";
        cout << "rc1 = " << rc1 << ", rc2 = " << rc2 << '\n';
        return;
    }

    ofstream ofs(csv_fn);

    if (ofs.bad())
        cout << "Ошибка создания файла - " << csv_fn << '\n';
    else
    {
        bool compare_error = false;
        uint32_t* histo1 = histo1_aa.Data();
        uint32_t* histo2 = histo2_aa.Data();
        const char* delim = ", ";

    for (uint32_t i = 0; i < 256; i++)
    {
        ofs << i << delim;
        ofs << histo1[i] << delim << histo2[i] << '\n';

        if (histo1[i] != histo2[i])
        {
            compare_error = true;
            cout << " Несовпадение гистограмм в индексе " << i << '\n';
            cout << " counts: " << histo1[i] << delim << histo2[i] << '\n';
        }
    }
```

```
    if (!compare_error)
        cout << " Гистограммы идентичны\n";

    ofs.close();
    }
}

int main()
{
    try
    {
        AvxBuildImageHistogram();
        AvxBuildImageHistogram_BM();
    }

    catch (...)
    {
        cout << "Произошла ошибка в работе программы\n";
        cout << "Файл = " << __FILE__ << '\n';
    }

return 0;
}

;-----------------------------------------------
;               Ch07_07.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; extern bool AvxBuildImageHistogram_(uint32_t* histo, const uint8_t* pixel_buff, uint32_t
num_pixels)
;
; Возвращает:      0 = некорректный аргумент, 1 = успех

        .code
        extern c_NumPixelsMax:dword

AvxBuildImageHistogram_ proc frame
        _CreateFrame BIH_,1024,0,rbx,rsi,rdi
        _EndProlog

; Проверка num_pixels
        xor eax,eax                         ;возвращаемый код ошибки
        test r8d,r8d
        jz Done                             ;переход, если num_pixels равен нулю
        cmp r8d,[c_NumPixelsMax]
        ja Done                             ;переход, если num_pixels слишком велик
        test r8d,1fh
        jnz Done                            ;переход, если num_pixels % 32 != 0

; Проверка правильности выравнивания histo & pixel_buff
        mov rsi,rcx                         ;rsi = указатель на histo
        test rsi,0fh
        jnz Done                            ;переход, если histo не выровнен
        mov r9,rdx
```

```
        test r9,0fh
        jnz Done                        ;переход, если pixel_buff не выровнен

; Инициализация локального буфера гистограмм (обнуление)
        xor eax,eax
        mov rdi,rsi                     ;rdi = указатель на histo
        mov rcx,128                     ;rcx = размер в четверных словах
        rep stosq                       ;обнуление histo
        mov rdi,rbp                     ;rdi = указатель на histo2
        mov rcx,128                     ;rcx = размер в четверных словах
        rep stosq                       ;обнуление histo2

; Инициализация рабочего цикла
        shr r8d,5                       ;количество пиксельных блоков (32 пикселя/блок)
        mov rdi,rbp                     ;указатель на histo2

; Построение гистограммы
        align 16                        ;выравнивание точки перехода
@@:     vmovdqa xmm0,xmmword ptr [r9]   ;чтение пиксельного блока
        vmovdqa xmm1,xmmword ptr [r9+16]    ;чтение пиксельного блока

; Обработка пикселей 0-3
        vpextrb rax,xmm0,0
        add dword ptr [rsi+rax*4],1     ;учет пикселя 0
        vpextrb rbx,xmm0,1
        add dword ptr [rdi+rbx*4],1     ;учет пикселя 1
        vpextrb rcx,xmm0,2
        add dword ptr [rsi+rcx*4],1     ;учет пикселя 2
        vpextrb rdx,xmm0,3
        add dword ptr [rdi+rdx*4],1     ;учет пикселя 3

; Обработка пикселей 4-7
        vpextrb rax,xmm0,4
        add dword ptr [rsi+rax*4],1     ;учет пикселя 4
        vpextrb rbx,xmm0,5
        add dword ptr [rdi+rbx*4],1     ;учет пикселя 5
        vpextrb rcx,xmm0,6
        add dword ptr [rsi+rcx*4],1     ;учет пикселя 6
        vpextrb rdx,xmm0,7
        add dword ptr [rdi+rdx*4],1     ;учет пикселя 7

; Обработка пикселей 8-11
        vpextrb rax,xmm0,8
        add dword ptr [rsi+rax*4],1     ;учет пикселя 8
        vpextrb rbx,xmm0,9
        add dword ptr [rdi+rbx*4],1     ;учет пикселя 9
        vpextrb rcx,xmm0,10
        add dword ptr [rsi+rcx*4],1     ;учет пикселя 10
        vpextrb rdx,xmm0,11
        add dword ptr [rdi+rdx*4],1     ;учет пикселя 11

; Обработка пикселей 12-15
        vpextrb rax,xmm0,12
        add dword ptr [rsi+rax*4],1     ;учет пикселя 12
        vpextrb rbx,xmm0,13
        add dword ptr [rdi+rbx*4],1     ;учет пикселя 13
```

```
        vpextrb rcx,xmm0,14
        add dword ptr [rsi+rcx*4],1         ;учет пикселя 14
        vpextrb rdx,xmm0,15
        add dword ptr [rdi+rdx*4],1         ;учет пикселя 15

; Обработка пикселей 16-19
        vpextrb rax,xmm1,0
        add dword ptr [rsi+rax*4],1         ;учет пикселя 16
        vpextrb rbx,xmm1,1
        add dword ptr [rdi+rbx*4],1         ;учет пикселя 17
        vpextrb rcx,xmm1,2
        add dword ptr [rsi+rcx*4],1         ;учет пикселя 18
        vpextrb rdx,xmm1,3
        add dword ptr [rdi+rdx*4],1         ;учет пикселя 19

; Обработка пикселей 20-23
        vpextrb rax,xmm1,4
        add dword ptr [rsi+rax*4],1         ;учет пикселя 20
        vpextrb rbx,xmm1,5
        add dword ptr [rdi+rbx*4],1         ;учет пикселя 21
        vpextrb rcx,xmm1,6
        add dword ptr [rsi+rcx*4],1         ;учет пикселя 22
        vpextrb rdx,xmm1,7
        add dword ptr [rdi+rdx*4],1         ;учет пикселя 23

; Обработка пикселей 24-27
        vpextrb rax,xmm1,8
        add dword ptr [rsi+rax*4],1         ;учет пикселя 24
        vpextrb rbx,xmm1,9
        add dword ptr [rdi+rbx*4],1         ;учет пикселя 25
        vpextrb rcx,xmm1,10
        add dword ptr [rsi+rcx*4],1         ;учет пикселя 26
        vpextrb rdx,xmm1,11
        add dword ptr [rdi+rdx*4],1         ;учет пикселя 27

; Обработка пикселей 28-31
        vpextrb rax,xmm1,12
        add dword ptr [rsi+rax*4],1         ;учет пикселя 28
        vpextrb rbx,xmm1,13
        add dword ptr [rdi+rbx*4],1         ;учет пикселя 29
        vpextrb rcx,xmm1,14
        add dword ptr [rsi+rcx*4],1         ;учет пикселя 30
        vpextrb rdx,xmm1,15
        add dword ptr [rdi+rdx*4],1         ;учет пикселя 31

        add r9,32                           ;r9  = следующий пиксельный блок
        sub r8d,1
        jnz @B                              ;повторение цикла до завершения

; Сборка финальной гистограммы из промежуточных гистограмм
        mov ecx,32                          ;ecx = количество итераций
        xor eax,eax                         ;rax = смещение

@@:     vmovdqa xmm0,xmmword ptr [rsi+rax]          ;чтение счетчика histo
        vmovdqa xmm1,xmmword ptr [rsi+rax+16]
        vpaddd xmm0,xmm0,xmmword ptr [rdi+rax]      ;добавление отсчетов из histo2
```

```
        vpaddd xmm1,xmm1,xmmword ptr [rdi+rax+16]
        vmovdqa xmmword ptr [rsi+rax],xmm0          ;сохранение результата
        vmovdqa xmmword ptr [rsi+rax+16],xmm1

        add rax,32
        sub ecx,1
        jnz @B
        mov eax,1                               ;возвращаемый код успешного завершения

Done:  _DeleteFrame rbx,rsi,rdi
        ret
AvxBuildImageHistogram_ endp
        end
```

В верхней части кода C++ расположена функция `AvxBuildImageHistogramCpp`. Эта функция строит гистограмму изображения, используя элементарную технику. Перед фактическим построением гистограммы количество пикселей изображения проверяется на предмет размера (больше 0 и не больше `c_NumPixelMax`) и делимости на 32. Тест делимости необходим для обеспечения совместимости с функцией `AvxBuildImageHistogram_`. Затем адреса `histo` и `pixel_buff` проверяются на правильное выравнивание. Вызов `memset` инициализирует каждую ячейку счетчика пикселей гистограммы нулевым значением. Затем для построения гистограммы используется простой цикл `for`.

Функция `AvxBuildImageHistogram` использует класс C++ с именем `ImageMatrix` для загрузки пикселей изображения в память. (Исходный код для `ImageMatrix` здесь не показан, но включен в скачиваемый пакет файлов главы.) Затем переменные `num_pixels` и `pixel_buff` инициализируются с помощью функций-членов `ImageMatrix::GetNumPixels` и `ImageMatrix::GetPixelBuffer`. Потом выделяются два буфера гистограммы с использованием класса шаблона C++ `AlignedArray <uint32_t>`. После построения гистограмм с использованием функций `AvxBuildImageHistogramCpp` и `AvxBuildImageHistogram_` количество пикселей в двух буферах гистограммы сравнивается на эквивалентность и записывается в текстовый файл со значениями, разделенными запятыми.

Функция на языке ассемблера `AvxBuildImageHistogram_` создает гистограмму изображения, используя набор команд AVX. Для повышения быстродействия эта функция строит две промежуточные гистограммы и объединяет их в окончательную гистограмму. `AvxBuildImageHistogram_` начинается с создания фрейма стека с помощью макроса `_CreateFrame`. Обратите внимание, что данный фрейм стека занимает 1024 байта (256 двойных слов, по одному для каждого уровня яркости оттенков серого) локального пространства хранения, которое используется для одного из промежуточных буферов гистограммы. После выполнения кода, сгенерированного макросом `_CreateFrame`, регистр RBP указывает на промежуточную гистограмму в стеке (рис. 5.6). Буфер `histo`, предоставленный вызывающей функцией, используется в качестве второго промежуточного буфера гистограммы. После макроса `_EndProlog` функция `AvxBuildImageHistogram_` проверяет размер `num_pixels` и его делимость на 32; а также проверяет адреса `histo` и `pixel_buff` на правильное выравнивание. Затем значения счетчика в обеих промежуточных гистограммах инициализируются нулевым значением с помощью команды `stosq`.

Основной рабочий цикл начинается с двух команд vmovdqa, которые загружают 32 пикселя изображения в регистры XMM1:XMM0. Обратите внимание, что до первой команды vmovdqa применяется директива MASM align 16 для выравнивания этой команды по 16-байтовой границе. Выравнивание конечной точки команды перехода по 16-байтовой границе – это метод оптимизации, который часто улучшает быстродействие. Этот и другие методы оптимизации более подробно рассмотрены в главе 15. Затем команда vpextrb rax,xmm0,0 извлекает элемент пикселя 0 (т. е. XMM0 [7: 0]) из регистра XMM0 и копирует его в младшие биты регистра RAX; старшие биты RAX устанавливаются в ноль. Последующая команда add dword ptr [rsi+rax*4],1 обновляет соответствующий интервал подсчета пикселей в первой промежуточной гистограмме. Следующие две команды, vpextrb rbx,xmm0,1 и add dword ptr [rdi+rbx*4],1, обрабатывают пиксельный элемент 1 таким же образом, используя вторую промежуточную гистограмму. Затем этот метод обработки пикселей повторяется для оставшихся пикселей в текущем блоке.

После рабочего выполнения цикла полученные значения подсчета пикселей в двух промежуточных гистограммах суммируются с использованием арифметики упакованных целых чисел для создания окончательной гистограммы. Затем используется макрос _DeleteFrame для освобождения фрейма локального стека и восстановления ранее сохраненных энергонезависимых регистров общего назначения. Результат выполнения кода примера Ch07_07 выглядит следующим образом:

```
Результаты AvxBuildImageHistogram
Гистограммы совпадают

Измерение быстродействия AvxBuildImageHistogram_BM - подождите
Результаты сохранены в файл Ch07_07_AvxBuildImageHistogram_BM_CHROMIUM.csv
```

В табл. 7.3 показаны результаты измерения быстродействия функций построения гистограммы.

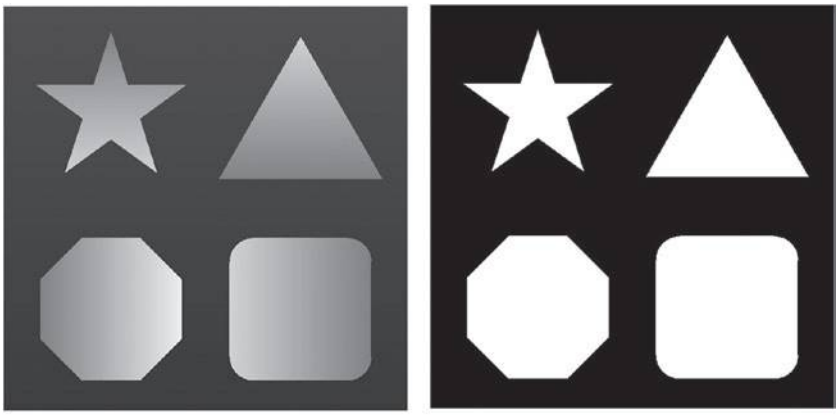
Таблица 7.3. Среднее время построения гистограммы (мс) для изображения TestImage1.bmp

Процессор	AvxBuildImageHistogramCpp	AvxBuildImageHistogram_
i7-4790S	277	230
i9-7900X	255	199
i7-8700K	241	191

7.4.5. Пороговая обработка изображений

Пороговая обработка изображений – это метод обработки изображений, который создает двоичное изображение (т. е. изображение только с двумя цветами) из изображения в градациях серого. Это двоичное изображение (или маска) показывает, какие пиксели в исходном изображении больше, чем предварительно определенное или полученное алгоритмом пороговое значение интенсивности. Рисунок 7.7 иллюстрирует операцию пороговой обработки. Изображения маски часто используются для выполнения дополнительных вы-

числений с использованием значений пикселей исходного изображения в градациях серого. Например, одним из типичных способов использования маски изображения, показанной на рис. 7.7, является вычисление среднего значения интенсивности всех пикселей выше порога в исходном изображении. Применение изображения маски упрощает вычисление среднего значения, поскольку облегчает использование простых логических выражений для исключения нежелательных пикселей из вычислений.



Исходное изображение в тонах серого Маска после пороговой обработки

Рис. 7.7. Примеры изображения в оттенках серого и его маски после пороговой обработки

Пример исходного кода `Ch07_08` демонстрирует, как вычислить среднюю интенсивность пикселей изображения выше указанного порога. Он также показывает, как вызвать функцию C++ из функции на языке ассемблера. В листинге 7.8 показан исходный код примера `Ch07_08`.

Листинг 7.8. Пример Ch07_08

```
//-----------------------------------------------
//              Ch07_08.h
//-----------------------------------------------

#pragma once
#include <cstdint>

// Структура данных для пороговой обработки изображений. Эта структура должна
// соответствовать структуре, определенной в Ch07_08_.asm
struct ITD
{
    uint8_t* m_PbSrc;                // Пиксельный буфер источника
    uint8_t* m_PbMask;               // Пиксельный буфер маски
    uint32_t m_NumPixels;            // Количество пикселей источника
    uint32_t m_NumMaskedPixels;      // Количество маскированных пикселей
    uint32_t m_SumMaskedPixels;      // Сумма маскированных пикселей
    uint8_t m_Threshold;             // Значение порога для изображения
    uint8_t m_Pad[3];                // Для последующего использования
    double m_MeanMaskedPixels;       // Среднее значение маскированных пикселей
};
```

```
// Функции, определенные в Ch07_08.cpp
extern bool AvxThresholdImageCpp(ITD* itd);
extern bool AvxCalcImageMeanCpp(ITD* itd);
extern "C" bool IsValid(uint32_t num_pixels, const uint8_t* pb_src, const uint8_t* pb_
mask);

// Функции, определенные в Ch07_08_.asm
extern "C" bool AvxThresholdImage_(ITD* itd);
extern "C" bool AvxCalcImageMean_(ITD* itd);

// Функции, определенные в Ch07_08_BM.cpp
extern void AvxThreshold_BM(void);

// Различные константы
const uint8_t c_TestThreshold = 96;

//-----------------------------------------------
//               Ch07_08.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#include "Ch07_08.h"
#include "AlignedMem.h"
#include "ImageMatrix.h"

using namespace std;

extern "C" uint32_t c_NumPixelsMax = 16777216;

bool IsValid(uint32_t num_pixels, const uint8_t* pb_src, const uint8_t* pb_mask)
{
    const size_t alignment = 16;

    // Проверка значения num_pixels
    if ((num_pixels == 0) || (num_pixels > c_NumPixelsMax))
        return false;
    if ((num_pixels % 64) != 0)
        return false;

    // Проверка выравнивания буферов изображения
    if (!AlignedMem::IsAligned(pb_src, alignment))
        return false;
    if (!AlignedMem::IsAligned(pb_mask, alignment))
        return false;

    return true;
}

bool AvxThresholdImageCpp(ITD* itd)
{
    uint8_t* pb_src = itd->m_PbSrc;
    uint8_t* pb_mask = itd->m_PbMask;
    uint8_t threshold = itd->m_Threshold;
    uint32_t num_pixels = itd->m_NumPixels;
```

```
    // Проверка счетчика пикселей и выравнивания буфера
    if (!IsValid(num_pixels, pb_src, pb_mask))
        return false;

    // Пороговая обработка
    for (uint32_t i = 0; i < num_pixels; i++)
        *pb_mask++ = (*pb_src++ > threshold) ? 0xff : 0x00;

    return true;
}

bool AvxCalcImageMeanCpp(ITD* itd)
{
    uint8_t* pb_src = itd->m_PbSrc;
    uint8_t* pb_mask = itd->m_PbMask;
    uint32_t num_pixels = itd->m_NumPixels;

    // Проверка счетчика пикселей и выравнивания буфера
    if (!IsValid(num_pixels, pb_src, pb_mask))
        return false;

    // Вычисление среднего значения маскированных пикселей
    uint32_t sum_masked_pixels = 0;
    uint32_t num_masked_pixels = 0;

    for (uint32_t i = 0; i < num_pixels; i++)
    {
        uint8_t mask_val = *pb_mask++;
        num_masked_pixels += mask_val & 1;
        sum_masked_pixels += (*pb_src++ & mask_val);
    }

    itd->m_NumMaskedPixels = num_masked_pixels;
    itd->m_SumMaskedPixels = sum_masked_pixels;

    if (num_masked_pixels > 0)
        itd->m_MeanMaskedPixels = (double)sum_masked_pixels / num_masked_pixels;
    else
        itd->m_MeanMaskedPixels = -1.0;

    return true;
}

void AvxThreshold(void)
{
    const wchar_t* fn_src = L"..\\Ch07_Data\\TestImage2.bmp";
    const wchar_t* fn_mask1 = L"Ch07_08_AvxThreshold_TestImage2_Mask1.bmp";
    const wchar_t* fn_mask2 = L"Ch07_08_AvxThreshold_TestImage2_Mask2.bmp";

    ImageMatrix im_src(fn_src);
    int im_h = im_src.GetHeight();
    int im_w = im_src.GetWidth();
    ImageMatrix im_mask1(im_h, im_w, PixelType::Gray8);
    ImageMatrix im_mask2(im_h, im_w, PixelType::Gray8);
    ITD itd1, itd2;
```

```
    itd1.m_PbSrc = im_src.GetPixelBuffer<uint8_t>();
    itd1.m_PbMask = im_mask1.GetPixelBuffer<uint8_t>();
    itd1.m_NumPixels = im_src.GetNumPixels();
    itd1.m_Threshold = c_TestThreshold;

    itd2.m_PbSrc = im_src.GetPixelBuffer<uint8_t>();
    itd2.m_PbMask = im_mask2.GetPixelBuffer<uint8_t>();
    itd2.m_NumPixels = im_src.GetNumPixels();
    itd2.m_Threshold = c_TestThreshold;

    // Пороговое изображение
    bool rc1 = AvxThresholdImageCpp(&itd1);
    bool rc2 = AvxThresholdImage_(&itd2);

    if (!rc1 || !rc2)
    {
        cout << "\nВозвращенный код ошибки: ";
        cout << "rc1 = " << rc1 << ", rc2 = " << rc2 << '\n';
        return;
    }

    im_mask1.SaveToBitmapFile(fn_mask1);
    im_mask2.SaveToBitmapFile(fn_mask2);

    // Вычисление среднего значения маскированных пикселей
    rc1 = AvxCalcImageMeanCpp(&itd1);
    rc2 = AvxCalcImageMean_(&itd2);

    if (!rc1 || !rc2)
    {
        cout << "\nВозвращенный код ошибки: ";
        cout << "rc1 = " << rc1 << ", rc2 = " << rc2 << '\n';
        return;
    }

    // Вывод результатов на печать
    const int w = 12;
    cout << fixed << setprecision(4);

    cout << "\nРезультаты AvxThreshold\n\n";
    cout << "                              C++       X86-AVX\n";
    cout << "-----------------------------------------\n";
    cout << "SumPixelsMasked:   ";
    cout << setw(w) << itd1.m_SumMaskedPixels << "  ";
    cout << setw(w) << itd2.m_SumMaskedPixels << '\n';
    cout << "NumPixelsMasked:   ";
    cout << setw(w) << itd1.m_NumMaskedPixels << "  ";
    cout << setw(w) << itd2.m_NumMaskedPixels << '\n';
    cout << "MeanMaskedPixels:  ";
    cout << setw(w) << itd1.m_MeanMaskedPixels << "  ";
    cout << setw(w) << itd2.m_MeanMaskedPixels << '\n';
}

int main()
{
    try
```

```
    {
        AvxThreshold();
        AvxThreshold_BM();
    }

    catch (...)
    {
        cout << "Аварийное прерывание программы";
    }

    return 0;
}

;-----------------------------------------------
;               Ch07_08.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; Структура данных для пороговой обработки изображения (см. Ch07_08.h)
ITD                 struct
PbSrc               qword ?
PbMask              qword ?
NumPixels           dword ?
NumMaskedPixels     dword ?
SumMaskedPixels     dword ?
Threshold           byte ?
Pad                 byte 3 dup(?)
MeanMaskedPixels    real8 ?
ITD                 ends

                .const
                align 16
PixelScale      byte 16 dup(80h)            ;uint8 в int8
CountPixelsMask byte 16 dup(01h)            ;маска подсчета пикселей
R8_MinusOne     real8 -1.0                  ;недействительное среднее значение

                .code
                extern IsValid:proc

; extern "C" bool AvxThresholdImage_(ITD* itd);
;
; Возвращает: 0 = недопустимый размер или невыровненный буфер, 1 = успех

AvxThresholdImage_ proc frame
        _CreateFrame TI_,0,0,rbx
        _EndProlog

; Проверка аргументов в структуре ITD
        mov rbx,rcx                         ;копирование указателя itd
        mov ecx,[rbx+ITD.NumPixels]         ;ecx = num_pixels
        mov rdx,[rbx+ITD.PbSrc]             ;rdx = pb_src
        mov r8,[rbx+ITD.PbMask]             ;r8 = pb_mask
        sub rsp,32                          ;выделение домашней области для IsValid
        call IsValid                        ;проверка аргументов
        or al,al
        jz Done                             ;переход, если ошибка аргументов
```

```
; Инициализация регистров для главного цикла
        mov ecx,[rbx+ITD.NumPixels]             ;ecx = num_pixels
        shr ecx,6                               ;ecx = количество 64-битовых блоков
        mov rdx,[rbx+ITD.PbSrc]                 ;rdx = pb_src
        mov r8,[rbx+ITD.PbMask]                 ;r8 = pb_mask

        movzx r9d,byte ptr [rbx+ITD.Threshold]  ;r9d = порог
        vmovd xmm1,r9d                          ;xmm1[7:0] = порог
        vpxor xmm0,xmm0,xmm0                    ;маска для vpshufb
        vpshufb xmm1,xmm1,xmm0                  ;xmm1 = упакованный порог

        vmovdqa xmm4,xmmword ptr [PixelScale]   ;упакованный коэффициент масштабирования
        vpsubb xmm5,xmm1,xmm4                   ;масштабированный порог

; Создание маски изображения
@@:     vmovdqa xmm0,xmmword ptr [rdx]      ;оригинальные пиксели изображения
        vpsubb xmm1,xmm0,xmm4               ;масштабированные пискели изображения
        vpcmpgtb xmm2,xmm1,xmm5             ;маскирование пикселей
        vmovdqa xmmword ptr [r8],xmm2       ;сохранение результата маскирования

        vmovdqa xmm0,xmmword ptr [rdx+16]
        vpsubb xmm1,xmm0,xmm4
        vpcmpgtb xmm2,xmm1,xmm5
        vmovdqa xmmword ptr [r8+16],xmm2

        vmovdqa xmm0,xmmword ptr [rdx+32]
        vpsubb xmm1,xmm0,xmm4
        vpcmpgtb xmm2,xmm1,xmm5
        vmovdqa xmmword ptr [r8+32],xmm2

        vmovdqa xmm0,xmmword ptr [rdx+48]
        vpsubb xmm1,xmm0,xmm4
        vpcmpgtb xmm2,xmm1,xmm5
        vmovdqa xmmword ptr [r8+48],xmm2

        add rdx,64
        add r8,64                           ;обновление указателей
        sub ecx,1                           ;обновление счетчика
        jnz @B                              ;повторять до завершения

        mov eax,1                           ;возвращаемый код успешного выполнения

Done:   _DeleteFrame rbx
        ret
AvxThresholdImage_ endp

;
; Макрос _UpdateBlockSums
;

_UpdateBlockSums macro disp
        vmovdqa xmm0,xmmword ptr [rdx+disp] ;xmm0 = 16 пикселей изображения
        vmovdqa xmm1,xmmword ptr [r8+disp]  ;xmm1 = 16 пикселей маски
        vpand xmm2,xmm1,xmm8                ;xmm2 = 16 пикселей маски (0x00 или 0x01)
        vpaddb xmm6,xmm6,xmm2               ;обновление блока num_masked_pixels
        vpand xmm2,xmm0,xmm1
```

```
        vpunpcklbw xmm3,xmm2,xmm9           ;распаковка значений пикселей из байта в слово
        vpunpckhbw xmm4,xmm2,xmm9
        vpaddw xmm4,xmm4,xmm3
        vpaddw xmm7,xmm7,xmm4               ;обновление sum_mask_pixels
        endm

; extern "C" bool AvxCalcImageMean_(ITD* itd);
;
; Возвращает:  0 = недопустимый размер или невыровненный буфер, 1 = успех

AvxCalcImageMean_ proc frame
        _CreateFrame CIM_,0,64,rbx
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9
        _EndProlog

; Проверка аргументов в структуре ITD
        mov rbx,rcx                         ;rbx = itd ptr
        mov ecx,[rbx+ITD.NumPixels]         ;ecx = num_pixels
        mov rdx,[rbx+ITD.PbSrc]             ;rdx = pb_src
        mov r8,[rbx+ITD.PbMask]             ;r8 = pb_mask
        sub rsp,32                          ;выделение домашней области для IsValid
        call IsValid                        ;проверка аргументов
        or al,al
        jz Done                             ; переход, если ошибка аргументов

; Инициализация регистров для рабочего цикла
        mov ecx,[rbx+ITD.NumPixels]         ;ecx = num_pixels
        shr ecx,6                           ;ecx = количество 64-битовых пиксельных блоков
        mov rdx,[rbx+ITD.PbSrc]             ;rdx = pb_src
        mov r8,[rbx+ITD.PbMask]             ;r8 = pb_mask

        vmovdqa xmm8,xmmword ptr [CountPixelsMask]  ;маска учитываемых пикселей
        vpxor xmm9,xmm9,xmm9                        ;xmm9 = упакованный ноль

        xor r10d,r10d                       ;r10d = num_masked_pixels (1 двойное слово)
        vpxor xmm5,xmm5,xmm5                ;sum_masked_pixels (4 двойных слова)

; Вычисление num_mask_pixels и sum_mask_pixels
LP1:    vpxor xmm6,xmm6,xmm6                ;num_masked_pixels_tmp (16 байтовых значений)
        vpxor xmm7,xmm7,xmm7                ;sum_masked_pixels_tmp (8 однословных значений)

        _UpdateBlockSums 0
        _UpdateBlockSums 16
        _UpdateBlockSums 32
        _UpdateBlockSums 48

; Обновление num_masked_pixels
        vpsrldq xmm0,xmm6,8
        vpaddb xmm6,xmm6,xmm0               ;num_mask_pixels_tmp (8 байтовых значений)
        vpsrldq xmm0,xmm6,4
        vpaddb xmm6,xmm6,xmm0               ;num_mask_pixels_tmp (4 байтовых значения)
        vpsrldq xmm0,xmm6,2
        vpaddb xmm6,xmm6,xmm0               ;num_mask_pixels_tmp (2 байтовых значения)
        vpsrldq xmm0,xmm6,1
        vpaddb xmm6,xmm6,xmm0               ;num_mask_pixels_tmp (1 байтовое значение)
        vpextrb eax,xmm6,0
        add r10d,eax                        ;num_mask_pixels += num_mask_pixels_tmp
```

```
; Обновление sum_masked_pixels
        vpunpcklwd xmm0,xmm7,xmm9           ;распаковка sum_mask_pixels_tmp в dword
        vpunpckhwd xmm1,xmm7,xmm9
        vpaddd xmm5,xmm5,xmm0
        vpaddd xmm5,xmm5,xmm1               ;sum_mask_pixels += sum_masked_pixels_tmp

        add rdx,64                          ;обновление указателя pb_src
        add r8,64                           ;обновление указателя pb_mask

        sub rcx,1                           ;обновление счетчика цикла
        jnz LP1                             ;повтор до завершения

; Вычисление среднего значения маскированных пикселей
        vphaddd xmm0,xmm5,xmm5
        vphaddd xmm1,xmm0,xmm0
        vmovd eax,xmm1                      ;eax = финальное значение sum_mask_pixels

        test r10d,r10d                      ;равно ли num_mask_pixels нулю?
        jz NoMean                           ;если да, то пропустить вычисление среднего
        vcvtsi2sd xmm0,xmm0,eax             ;xmm0 = sum_masked_pixels
        vcvtsi2sd xmm1,xmm1,r10d            ;xmm1 = num_masked_pixels
        vdivsd xmm2,xmm0,xmm1               ;xmm2 = mean_masked_pixels
        jmp @F

NoMean: vmovsd xmm2,[R8_MinusOne]               ; -1.0 при отсутствии среднего значения

@@:     mov [rbx+ITD.SumMaskedPixels],eax       ;сохранение sum_masked_pixels
        mov [rbx+ITD.NumMaskedPixels],r10d      ;сохранение num_masked_pixels
        vmovsd [rbx+ITD.MeanMaskedPixels],xmm2  ;сохранение среднего значения
        mov eax,1                               ;возвращаемый код успешного выполнения

Done:   _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9
        _DeleteFrame rbx
        ret
AvxCalcImageMean_  endp
        end
```

Алгоритм, используемый в примере `Ch07_08`, состоит из двух этапов. На первом этапе создается изображение маски, показанное на рис. 7.7. На втором этапе вычисляется средняя интенсивность всех пикселей в изображении в градациях серого, соответствующие пиксели изображения маски которых являются белыми (то есть выше указанного порога). Файл `Ch07_08.h`, показанный в листинге 7.8, определяет структуру с именем `ITD`, которая хранит данные, необходимые алгоритму. Обратите внимание, что эта структура содержит два значения счетчика: `m_NumPixels` и `m_NumMaskedPixels`. Первое значение – это общее количество пикселей изображения, а второе значение представляет количество пикселей изображения, превышающих `m_Threshold`.

Код C++ в листинге 7.8 содержит отдельные функции определения порога и вычисления среднего. Функция `AvxThresholdImageCpp` создает изображение маски, сравнивая каждый пиксель в изображении в градациях серого с пороговым значением, заданным параметром `itd->m_Threshold`. Если пиксель изображения в градациях серого больше этого значения, соответствующему пикселю в изображении маски присваивается значение `0xff`; в противном случае этому

пикселю присваивается значение 0x00. Функция AvxCalcImageMeanCpp использует это изображение маски для вычисления среднего значения интенсивности всех пикселей изображения в градациях серого, превышающих пороговое значение. Обратите внимание, что цикл for в этой функции вычисляет num_mask_pixels и sum_mask_pixels с использованием простых выражений двоичной логики вместо операций логического сравнения. Первый метод часто быстрее и проще реализовать с помощью арифметики SIMD.

В листинге 7.8 также показаны реализации на языке ассемблера функций определения порога и вычисления среднего значения. Следом за прологом функция AvxThresholdImage_ проверяет аргументы в предоставленной структуре ITD, вызывая функцию C++ IsValid. Перед командой вызова AvxThresholdImage_ загружает необходимые значения аргументов для IsValid в соответствующие регистры и выделяет домашнюю область с помощью команды sub rsp,32. После проверки аргумента команда movzx r9d,byte ptr [rbx+ITD.Threshold] загружает пороговое значение в регистр R9D. Последующая команда vpshufb xmm1,xmm1,xmm0 «транслирует» пороговое значение на все позиции байтов в регистре XMM1. Команда vpshufb использует четыре младших бита каждого байта во втором исходном операнде в качестве индекса для перестановки байтов в целевом операнде (если во втором исходном операнде старший бит установлен, копируется ноль). Этот процесс наглядно представлен на рис. 7.8. Затем упакованное пороговое значение масштабируется с помощью команды vpsubb. Причина масштабирования объясняется ниже.

Рис. 7.8. Примеры выполнения команды vpshufb

Главный цикл в функции AvxThreshholdImage_ использует команду vpcmpgtb (сравнить упакованные целые числа со знаком по условию «больше, чем») для создания изображения маски. Эта команда выполняет попарное сравнение байтовых элементов в двух исходных операндах. Если байт в первом исходном операнде больше, чем соответствующий байт во втором операнде, байт операнда-приемника устанавливается в 0xff; в противном случае байт операнда-приемника устанавливается в 0x00. На рис. 7.9 наглядно показано выполнение команды vpcmpgtb. Важно отметить, что команда vpcmpgtb выполняет свои срав-

нения, используя целочисленную арифметику со знаком. Это означает, что значения пикселей в изображении в градациях серого, которые представляют собой байтовые значения без знака, должны быть повторно масштабированы для совместимости с командой `vpcmpgtb`. Команда `vpsubb` переносит значения пикселей изображения в оттенках серого из диапазона [0, 255] в диапазон [–128, 127]. Это также причина того, что команда `vpsubb` применялась к упакованному пороговому значению до начала цикла. После каждой операции сравнения команда `vmovdqa` сохраняет пиксели маски в указанный буфер. Подобно примеру `Ch07_04`, функция `AvxThresholdImage_` использует частично развернутый цикл обработки для обслуживания 64 пикселей на итерацию.

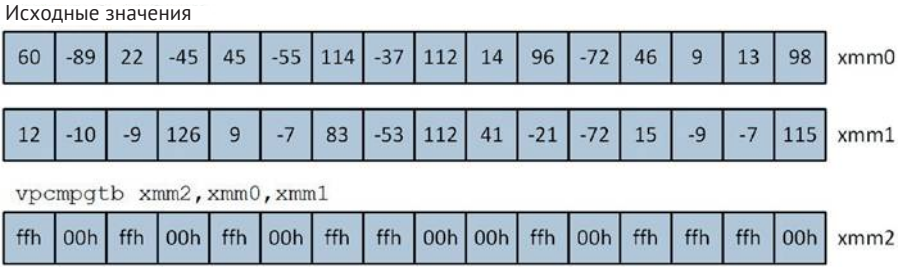


Рис. 7.9. Выполнение команды `vpcmpgtb`

Функция на языке ассемблера `AvxCalcImageMean_` также начинается с проверки аргументов с помощью функции `IsValid` C++. После проверки аргументов команды `xor r10d,r10d` и `vpxor xmm5,xmm5,xmm5` инициализируют `num_masked_pixels` и `sum_masked_pixels` (четыре двойных слова) нулями. Цикл обработки в функции `AvxCalcImageMean_` использует макрос с именем `_UpdateBlockSums` для вычисления промежуточных значений `num_masked_pixels_tmp` и `sum_masked_pixels_tmp` для блока из 64 пикселей. Этот макрос выполняет свои вычисления с использованием упакованной арифметики байтов и слов, которая сокращает количество байтов до увеличения размера двойного слова, которое необходимо выполнить. На рис. 7.10 показаны арифметические и логические операции, выполняемые макросом `_UpdateBlockSums`. Затем обновляются значения `num_masked_pixels` (R10D) и `sum_masked_pixels` (XMM5), и цикл обработки повторяется до тех пор, пока не будут обработаны все пиксели.

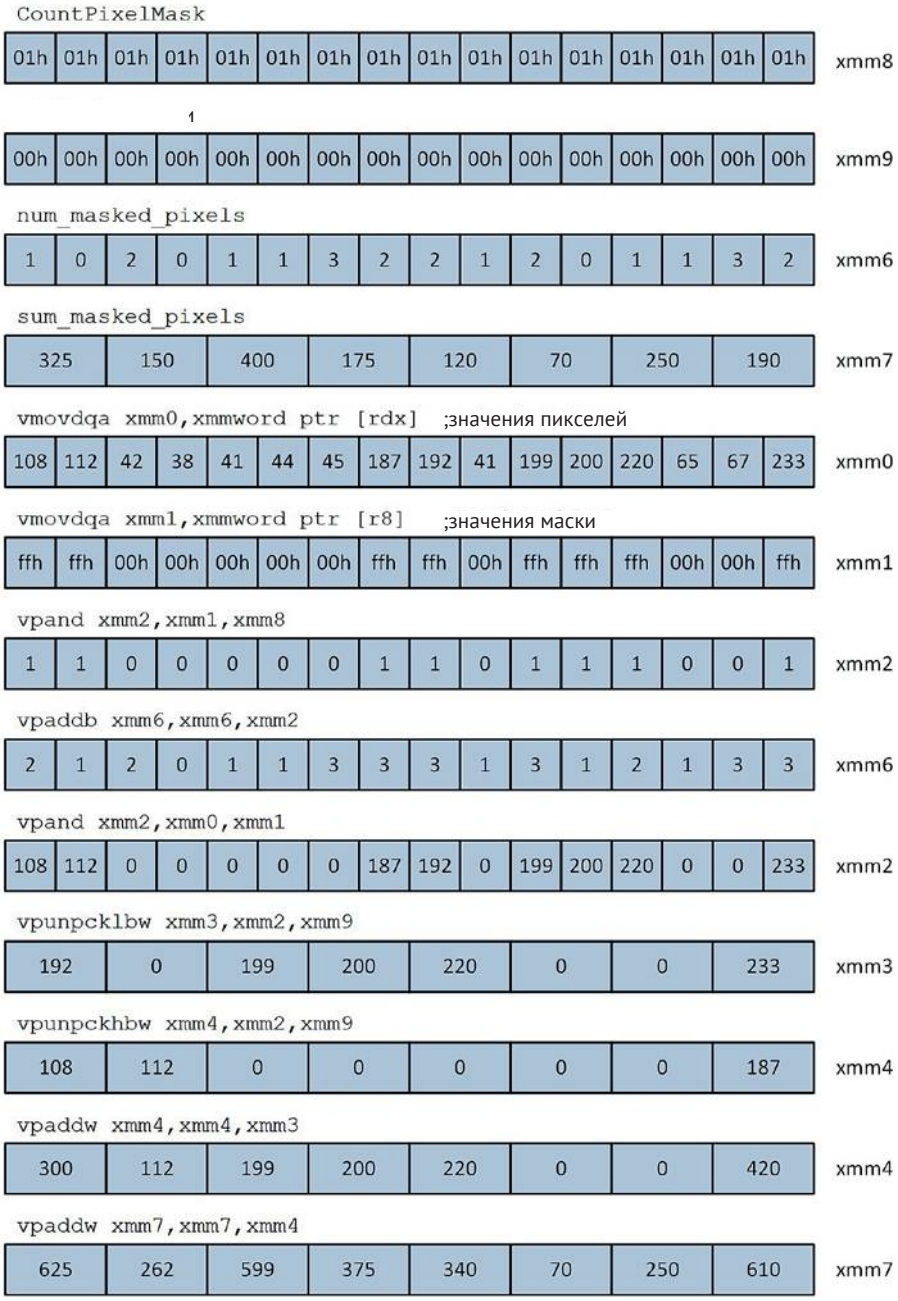


Рис. 7.10. Вычисления суммы маскированных пикселей и количества пикселей, выполняемые макросом `_UpdateBlockSums`

После завершения рабочего цикла функция AvxCalcImageMean_ вычисляет окончательное среднее значение интенсивности, используя скалярную арифметику с плавающей запятой двойной точности. Обратите внимание, что num_mask_pixels проверяется перед вычислением среднего значения, чтобы избежать ошибки деления на ноль. Вот результат выполнения примера исходного кода Ch07_08:

```
Результаты AvxThreshold

                            C++       X86-AVX
-------------------------------------------
SumPixelsMasked:       23813043      23813043
NumPixelsMasked:         138220        138220
MeanMaskedPixels:      172.2836      172.2836

Измерение быстродействия AvxThreshold_BM - подождите
Результаты сохранены в файл Ch07_08_AvxThreshold_BM_CHROMIUM.csv
```

В табл. 7.4 показаны результаты измерения быстродействия примера исходного кода Ch07_08. Обратите внимание, что измерения в этой таблице охватывают этапы пороговой обработки всего изображения и вычисления среднего значения.

Таблица 7.4. Среднее время выполнения (мс) для определения порога изображения и вычисления среднего значения с использованием TestImage2.bmp

Процессор	C++	Ассемблер
i7-4790S	289	50
i9-7900X	250	40
i7-8700K	242	39

7.5. Заключение

В этой главе были рассмотрены следующие ключевые моменты:

- команды vpadd[b|w|d|q] выполняют сложение упакованных чисел. Команды vpadds[b|w] и vpaddus[b|w] выполняют сложение упакованных знаковых и беззнаковых операндов с насыщением;
- команды vpsub[b|w|d|q] выполняют вычитание упакованных чисел. Команды vpsubs[b|w] и vpsubus[b|w] выполняют вычитание упакованных знаковых и беззнаковых операндов с насыщением;
- команды vpmul[h|l]w выполняют умножение с использованием упакованных слов;
- команды vpmuldq и vpmulld выполняют умножение с использованием упакованных операндов двойных слов;

- команды `vpsll[w|d|q]` и `vpsrl[w|d|q]` выполняют логические сдвиги влево и вправо с использованием упакованных операндов. Команды `vpsra[w|d|q]` выполняют арифметические сдвиги вправо с использованием упакованных операндов. Команды `vps[l|r]dq` выполняют логические сдвиги влево и вправо с использованием операндов шириной 128 бит;
- функции на языке ассемблера могут использовать команды `vpand`, `vpor` и `vpxor` для выполнения операций побитового И, включающего ИЛИ и исключающего ИЛИ с использованием упакованных целочисленных операндов;
- команды `vpextr[b|w|d|q]` извлекают значение элемента из упакованного операнда. Команды `vpinsr[b|w|d|q]` вставляют значение элемента в упакованный операнд;
- команды `vpunpckl[bw|dw|dq]` и `vpunpckh[bw|dw|dq]` распаковывают и чередуют содержимое своих двух исходных операндов. Эти команды часто применяются для увеличения размера упакованных целочисленных операндов. Команды `vpackus[bw|dw]` уменьшают размер упакованных целочисленных операндов с помощью беззнаковой арифметики с насыщением;
- команды `vpminu[b|w|d]` и `vpmaxu[b|w|d]` выполняют сравнение упакованных беззнаковых целых чисел, возвращая минимум и максимум соответственно;
- команда `vpshufb` переупорядочивает байты упакованного операнда в соответствии с маской;
- команды `vpcmpgt[b|w|d|q]` выполняют сравнение «больше, чем» упакованных целых чисел со знаком;
- выравнивание точки назначения команды перехода по 16-байтовой границе часто приводит к более быстрому выполнению циклов `for`.

Глава 8

Подробнее про AVX2

В предыдущих четырех главах вы узнали об архитектуре и функциональных возможностях AVX. В этих главах подробно описаны наборы регистров, типы данных и команды AVX. Они также содержат многочисленные примеры исходного кода, иллюстрирующие скалярную арифметику с плавающей запятой, операции над упакованными операндами с плавающей запятой и упакованными целочисленными операндами. Многие из этих примеров иллюстрируют важные стратегии и приемы программирования SIMD, использование которых часто приводит к более быстрому выполнению кода.

В данной главе описываются архитектура и вычислительные ресурсы набора Advanced Vector Extensions 2 (AVX2). Вы узнаете о расширенных возможностях AVX2 для обработки упакованных операндов с плавающей запятой и упакованных целочисленных операндов. Вы также ознакомитесь с важными деталями, касающимися последних расширений набора команд платформы x86, включая преобразования с плавающей запятой половинной точности, операции слитного умножения-сложения (FMA) и новые команды для работы с регистрами общего назначения.

Материал, представленный в этой главе, предполагает, что вы хорошо разбираетесь в AVX. Если вы чувствуете, что вашего понимания набора команд AVX, типов данных или возможностей обработки SIMD в какой-то области недостаточно, вы можете перечитать соответствующие разделы в предыдущих главах, прежде чем двигаться дальше.

8.1. Среда выполнения AVX2

AVX2 использует те же наборы регистров YMM и XMM, что и AVX (рис. 4.6). AVX2 также применяет регистр состояния и управления MXCSR для сигнализации об арифметических ошибках вычислений с плавающей запятой, настройки параметров округления и управления генерацией исключений при операциях с плавающей запятой (рис. 4.11). Как и AVX, AVX2 поддерживает операции SIMD с плавающей запятой, используя 128-битные или 256-битные операнды, содержащие значения с одинарной или двойной точностью. AVX2 расширяет возможности AVX по обработке упакованных целых чисел, чтобы задействовать как 128-битные, так и 256-битные операнды (AVX поддерживает только 128-битные целочисленные операнды). При использовании с 256-битным упакованным целочисленным операндом команда AVX2 может одновременно

обрабатывать 32 байта, 16 слов, 8 двойных слов или 4 четверных слова. AVX2 также содержит ряд полезных команд, которые управляют упакованными операндами с плавающей запятой и упакованными целыми числами. Вы узнаете больше об этих инструкциях позже в этой главе.

Инструкции AVX2 используют тот же синтаксис инструкций, что и AVX. Большинство инструкций AVX2 используют формат с тремя операндами, который состоит из двух операндов-источников и одного операнда-приемника. Почти все операнды-источники команд AVX2 неразрушающие. Это означает, что исходные операнды не изменяются во время выполнения инструкции, за исключением случаев, когда регистр операнда-приемника совпадает с одним из регистров операнда-источника. Несколько команд AVX2 используют третий непосредственный операнд-источник, который обычно используется как маска управления.

Требования к выравниванию операндов AVX2 в памяти такие же, как и для AVX. За исключением команд передачи данных, которые явно ссылаются на выровненный операнд в памяти (например, `vmovdqa,vmovap[d|s]` и т. д.), правильное выравнивание операнда AVX2 в памяти не является обязательным. Однако 128-битные операнды в памяти всегда должны быть выровнены по 16-байтовой границе, когда это возможно, чтобы максимизировать быстродействие. Точно так же 256-битные операнды должны быть выровнены по 32-байтовой границе.

8.2. Команды AVX2 для упакованных чисел с плавающей запятой

AVX2 расширяет возможности обработки упакованных операндов с плавающей запятой в AVX за счет добавления операций извлечения данных. Инструкции `vgatherdp[d|s]` и `vgatherqp[d|s]` загружают несколько элементов из несмежных ячеек памяти (обычно массива) в регистры XMM или YMM. Эти инструкции используют специальный режим адресации памяти, называемый векторным доступом по масштабированной индексной базе (vector scale-index-base, VSIB). При адресации памяти VSIB используются следующие компоненты для определения местоположения элементов в памяти:

- *масштаб*: масштабный коэффициент размера элемента (1, 2, 4 или 8);
- *индекс*: регистр векторных индексов (XMM или YMM), который содержит индексы двойных слов со знаком или индексов четверных слов со знаком;
- *база*: регистр общего назначения, указывающий на начало массива в памяти;
- *смещение*: необязательное фиксированное смещение от начала массива.

Перед выполнением инструкции `vgatherdp[d|s]` или `vgatherqp[d|s]` в регистр операнда индекса должны быть загружены правильные индексы. Процессор использует эти индексы для выбора элементов из массива. На рис. 8.1 показано выполнение команды `vgatherdps xmm0,[rax+xmm1*4],xmm2`. В этом примере регистр RAX указывает на начало массива, содержащего значения с плавающей запятой одинарной точности; регистр XMM1 содержит четыре индекса

массива двойных слов со знаком; а регистр XMM2 содержит маску управления копированием. Маска управления копированием определяет, копирует ли инструкция `vgatherdps` конкретный элемент массива в операнд-приемник. Если установлен самый старший бит элемента маски управления, соответствующий элемент массива, указанный в индексном регистре, копируется в операнд-приемник; в противном случае элемент операнда назначения не изменяется. После успешного выполнения инструкции `vgatherdp[d|s]` или `vgatherqp[d|s]` регистр маски управления копированием (который является операндом-источником) содержит все нули.

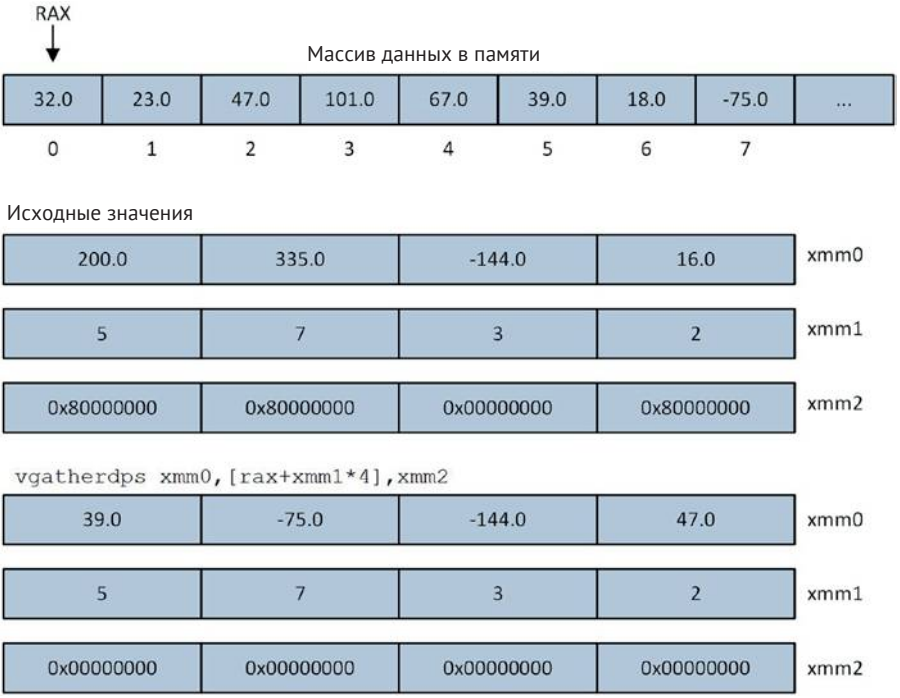


Рис. 8.1. Иллюстрация выполнения инструкции `vgatherdps`

Операнд-приемник и второй операнд-источник (маска управления копированием) команды `vgatherdp[d|s]` или `vgatherqp[d|s]` должны быть регистром XMM или YMM. Первый операнд-источник определяет компоненты VSIB (т. е. базовый регистр, регистр векторных индексов, масштабный коэффициент и необязательное смещение). Команды `vgatherdp[d|s]` или `vgatherqp[d|s]` не выполняют никаких проверок на недопустимые индексы. Недопустимый индекс – это любое значение регистра индекса, указывающее команде извлечения данных загрузить элемент из области памяти, которая выходит за пределы массива. Использование недопустимого индекса приведет к неверному результату и, возможно, заставит процессор сгенерировать исключение.

Другое примечательное дополнение AVX2 для операций с плавающей запятой включает команды `vbroadcasts[d|s]`. Для процессоров, поддерживающих AVX2, операндом-источником этих команд может быть регистр XMM (AVX

поддерживает только операнды-источники vbroadcasts[d|s], размещенные в памяти). При таком использовании команды vbroadcasts[d|s] копируют младшие разряды элемента с плавающей запятой одинарной или двойной точности из регистра XMM в каждую позицию элемента в операнде-приемнике.

8.3. Команды AVX2 для упакованных целых чисел

Как упоминалось ранее в этой главе, AVX2 расширяет функциональные возможности AVX по работе с упакованными целыми числами для поддержки как 128-битных, так и 256-битных операндов. В системах, поддерживающих AVX2, большинство команд, работающих с упакованными целыми числами, могут использовать в качестве операндов регистры XMM или YMM. Наиболее заметным исключением из этого правила являются команды vpextr[b|w|d|q] (извлечь целочисленное значение) и vpinsr[b|w|d|q] (вставить целочисленное значение), которые нельзя использовать с операндом в регистре YMM. AVX2 также добавляет ряд новых команд для работы с упакованными целыми числами, у которых нет соответствующего аналога AVX (или x86-SSE). В табл. 8.1 эти команды перечислены в алфавитном порядке.

Таблица 8.1. Перечень новых команд AVX2 для работы с упакованными целыми числами

Мнемоника	Описание
vbroadcasti128	Передача 128 бит целочисленных данных
vextracti128	Извлечение 128 бит целочисленных данных
vinserti128	Вставка 128 бит целочисленных данных
vpblendd	Перемешивание упакованных двойных слов
vpbroadcast[b|w|d|q]	Передача целочисленного значения
vperm2i128	Перестановка 128-битовых целочисленных данных
vperm[d|q]	Перестановка упакованных целочисленных данных
vpgatherd[d|q]	Извлечение упакованных целочисленных данных с использованием двойных слов со знаком в качестве индексов
vpgatherq[d|q]	Извлечение упакованных целочисленных данных с использованием четверных слов со знаком в качестве индексов
vpmaskmov[d|q]	Условное присвоение упакованных целочисленных данных
vpsllv[d|q]	Логический сдвиг влево с использованием подсчета битов отдельных элементов
vpsravd	Арифметический сдвиг вправо с использованием подсчета битов отдельных элементов
vpsrlv[d|q]	Логический сдвиг вправо с использованием подсчета битов отдельных элементов

Команды vpgatherd[d|q] и vpgatherq[d|q], показанные в табл. 8.1, используют ту же схему адресации памяти VSIB, что и их аналоги с плавающей запятой.

8.4. Расширения набора команд x86

В последние годы в платформу x86 был добавлен ряд расширений набора команд, помимо AVX и AVX2. Многие из этих расширений включают команды, которые выполняют специализированные операции или ускоряют выполнение определенных алгоритмов. В табл. 8.2 перечислены расширения набора команд x86, использование которых обсуждается и иллюстрируется в следующих главах. Важно помнить, что все расширения, показанные в этой таблице, представляют собой отдельные наборы команд процессора. С точки зрения программирования это означает, что вы не должны предполагать, что конкретный набор команд или конкретная команда доступны в зависимости от того, поддерживает исполняющий процессор AVX или AVX2. Доступность определенного расширения набора команд, включая AVX и AVX2, всегда следует явно проверять при помощи команды `cupid`. Это особенно важно для программной совместимости с будущими процессорами AMD и Intel. Вы узнаете, как это сделать, в главе 16.

Таблица 8.2. Последние расширения набора команд x86

Расширение набора команд	Флаг CPUID
Расширенное сложение беззнаковых целых чисел	ADX
Расширенные операции с битами (группа 1)	BMI1
Расширенные операции с битами (группа 2)	BMI2
Преобразование в число с плавающей запятой половинной точности	F16C
Слитное умножение-сложение	FMA
Подсчет ведущих нулевых битов	LZCNT
Подсчет единичных битов	POPCNT

В оставшейся части этого раздела кратко описаны расширения набора команд, показанные в табл. 8.2. Главы 10 и 11 содержат примеры исходного кода, иллюстрирующие применение некоторых инструкций, включенных в эти расширения. Информацию о расширениях набора команд, не показанных в табл. 8.2, можно найти в справочных руководствах по программированию, опубликованных AMD и Intel. Приложение содержит список этих руководств.

8.4.1. Числа с плавающей запятой половинной точности

Последние процессоры от AMD и Intel поддерживают команды, выполняющие преобразования чисел с плавающей запятой *половинной точности*. По сравнению со стандартным значением с плавающей запятой одинарной точности, значение с плавающей запятой половинной точности – это число, которое содержит три поля: показатель степени (5 бит), мантиссу (11 бит) и бит знака. Каждое число с плавающей запятой половинной точности имеет ширину 16 бит; подразумевается ведущая цифра мантиссы. Совместимые процессоры поддерживают команды, которые могут преобразовывать упакованные значения с плавающей запятой половинной точности в упакованные значения с пла-

вающей запятой одинарной точности, и наоборот. Эти команды представлены в табл. 8.3. Значения с плавающей запятой половинной точности в первую очередь предназначены для уменьшения требований к пространству для хранения данных, будь то в памяти или на физическом устройстве. К недостаткам использования таких значений относятся пониженная точность и ограниченный диапазон. Процессоры, поддерживающие команды преобразования, показанные в табл. 8.3, *не поддерживают* команды для выполнения общих арифметических операций, таких как сложение, вычитание, умножение и деление, с использованием значений с плавающей запятой половинной точности.

Таблица 8.3. Команды преобразования чисел с плавающей запятой половинной точности

Мнемоника	Описание
`vcvtph2ps`	Преобразование числа половинной точности с плавающей запятой в число одинарной точности с плавающей запятой
`vcvtps2ph`	Преобразование числа одинарной точности с плавающей запятой в число половинной точности с плавающей запятой

8.4.2. Слитное умножение-сложение (FMA)

Современные процессоры AMD и Intel также поддерживают команды, выполняющие операции *слитного умножения-сложения* (fused-multiply-add, FMA). Команда FMA объединяет умножение и сложение (или вычитание) в одну операцию. Если точнее, вычисление слитного умножения-сложения (или слитного умножения-вычитания) означает умножение с плавающей запятой с последующим сложением (или вычитанием) с плавающей запятой с использованием одной операции округления. Например, рассмотрим выражение `d=b*c+a`. Используя стандартную арифметику с плавающей запятой, процессор сначала вычисляет произведение `b*c`, которое включает операцию округления. За этим следует вычисление результата сложения с плавающей запятой, которое также включает операцию округления. Если выражение вычисляется с использованием арифметики FMA, процессор не округляет промежуточное произведение `b*c`. Округление выполняется только один раз с использованием рассчитанного значения выражения `b*c+a`. Инструкции FMA часто используются для повышения производительности и точности вычислений умножения с накоплением, таких как скалярные произведения и умножение матрицы на вектор. Многие алгоритмы обработки сигналов также широко используют операции FMA.

Мнемоника команд FMA использует трехзначную схему упорядочивания операндов, которая указывает, какие операнды использовать для умножения и сложения (или вычитания). В этой схеме все три операнда команд используются как исходные операнды. Первая мнемоническая позиция указывает исходный операнд для использования в качестве множимого; вторая позиция указывает исходный операнд для использования в качестве множителя; а на третьей позиции располагается исходный операнд, который прибавляется к произведению (или вычитается из него). Например, рассмотрим команду `vfmadd132sd xmm4,xmm5,xmm6` (слитное умножение-сложение скалярных значений с плавающей запятой двойной точности). В этом примере регистры XMM4, XMM5 и XMM6 являются исходными операндами на позициях 1, 2 и 3 соответ-

ственно. Команда `vfmadd132sd` вычисляет `xmm4[63:0]*xmm6[63:0]+xmm5[63:0]`, округляет значение выражения в соответствии с режимом округления, заданным MXCSR.RC, и сохраняет окончательный результат в `xmm4[63:0]`.

Расширение FMA набора команд x86 поддерживает операции с использованием скалярных или упакованных значений с плавающей запятой, как с одинарной, так и с двойной точностью. Операции FMA с упакованными операндами могут выполняться с использованием регистров XMM или YMM. Регистры XMM (YMM) поддерживают упакованные вычисления FMA с использованием двух (четырех) значений двойной точности или четырех (восьми) значений с плавающей запятой одинарной точности. Скалярные вычисления FMA выполняются с использованием набора регистров XMM. Для всех инструкций FMA первый и второй исходные операнды должны быть регистрами. Третий исходный операнд может быть регистром или ячейкой памяти. Если инструкция FMA использует регистр XMM в качестве операнда-приемника, старшие 128 бит соответствующего регистра YMM устанавливаются в ноль. Инструкции FMA выполняют свою единственную операцию округления с использованием режима, указанного MXCSR. RC, как сказано в предыдущем абзаце.

В табл. 8.4 представлен набор команд FMA. В мнемонике команд в этой таблице используются следующие двухбуквенные суффиксы: `pd` (упакованные с плавающей запятой двойной точности), `ps` (упакованные с плавающей запятой одинарной точности), `sd` (скалярные с плавающей запятой двойной точности) и `ss` (скалярные с плавающей запятой одинарной точности). Символы `src1`, `src2` и `src3` обозначают три исходных операнда; операндом-приемником всегда является `src1`.

Таблица 8.4. Обзор команд FMA

Подгруппа	Мнемоника	Операция
VFMADD	`vfmaddl32[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`
VFMSUB	`vfmsubl32[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`
VFMADDSUB	`vfmaddsubl32[pd|ps]`	`des = src1 * src3 + src2 (нечетные)`
		`des = src1 * src3 - src2 (четные)`
	`vfmaddsub213[pd|ps]`	`des = src2 * src1 + src3 (нечетные)`
		`des = src2 * src1 - src3 (четные)`
	`vfmaddsub231[pd|ps]`	`des = src2 * src3 + src1 (нечетные)`
		`des = src2 * src3 - src1 (четные)`

Окончание табл. 8.4

Подгруппа	Мнемоника	Операция
VFMSUBADD	vfmsubaddl32[pd|ps]	des = srcl * src3 - src2 (нечетные)
		des = srcl * src3 + src2 (четные)
	vfmsubadd213[pd|ps]	des = src2 * srcl - src3 (нечетные)
		des = src2 * srcl + src3 (четные)
	vfmsubadd231[pd|ps]	des = src2 * src3 - srcl (нечетные)
		des = src2 * src3 + srcl (четные)
VFNMADD	vfnmaddl32[pd|ps|sd|ss]	des = -(srcl * src3) i + src2
	vfnmadd213[pd|ps|sd|ss]	des = -(src2 * srcl) i + src3
	vfnmadd231[pd|ps|sd|ss]	des = -(src2 * src3) i + srcl
VFNMSUB	vfnmsubl32[pd|ps|sd|ss]	des = -(srcl * src3) i - src2
	vfnmsub213[pd|ps|sd|ss]	des = -(src2 * srcl) i - src3
	vfnmsub231[pd|ps|sd|ss]	des = -(src2 * src3) i - srcl

Команды FMA, показанные в табл. 8.4, многие утилиты обнаружения функций ЦП и онлайн-документация называют командами FMA3. Некоторые процессоры AMD также поддерживают дополнительные команды FMA4, которые выполняют свои операции FMA с использованием трех операндов-источников и одного операнда-приемника (трехместная схема упорядочения операндов не используется). Эти инструкции не показаны в табл. 8.4.

8.4.3. Расширения набора команд для регистров общего назначения

Недавние усовершенствования платформы x86 также включают ряд расширений набора команд для регистров общего назначения. Расширения набора команд ADX, BMI1, BMI2, LZCNT и POPCNT поддерживают улучшенную целочисленную арифметику без знака, расширенные манипуляции с битами и операции поворота и сдвига регистров без флагов (операция без флага не обновляет ни один из флагов состояния в RFLAGS). Многие из этих инструкций предназначены для ускорения работы определенных алгоритмов, таких как арифметические действия с большими целыми числами и шифрование данных. Некоторые из этих команд используют синтаксис языка ассемблера с тремя операндами, аналогичный AVX. В табл. 8.5 перечислены инструкции, входящие в состав расширений ADX, BMI1, BMI2, LZCNT и POPCNT.

Таблица 8.5. Обзор инструкций ADX, BMI1, BMI2, LZCNT и POPCNT

Мнемоника	Флаг CPUID	Описание
`adcx`	ADX	Целочисленное беззнаковое сложение с флагом переноса
`adox`	ADX	Целочисленное беззнаковое сложение с флагом переполнения
`andn`	BMI1	Побитовое И инвертированного операнда 1 с операндом 2
`bextr`	BMI1	Извлечение битового поля
`blsi`	BMI1	Извлечение самого младшего единичного бита
`blsmsk`	BMI1	Применить маску до самого младшего установленного бита
`blsr`	BMI1	Сбросить самый младший единичный бит
`bzhi`	BMI2	Обнулить старшие биты, начиная с указанной позиции
`Izcnt`	LZCNT	Подсчет ведущих нулевых битов
`mulx`	BMI2	Целочисленное беззнаковое умножение без флагов
`pdep`	BMI2	Параллельная запись битов
`pext`	BMI2	Параллельное извлечение битов
`popent`	POPCNT	Подсчет количества единичных битов
`rorx`	BMI2	Вращение вправо без флага
`sarx`	BMI2	Арифметический сдвиг вправо без флага
`shlx`	BMI2	Логический сдвиг влево без флага
`shrx`	BMI2	Логический сдвиг вправо без флага
`tzent`	BMI1	Посчет количества завершающих нулевых битов

8.5. Заключение

Ниже перечислены ключевые моменты главы 8:

- AVX2 использует те же наборы регистров, типы данных и синтаксис команд, что и AVX;
- AVX2 расширяет возможности обработки упакованных целых чисел AVX для поддержки операций с использованием операндов шириной 256 бит;
- AVX2 содержит новые инструкции обработки упакованных целых чисел, которые выполняют операции широковещательной передачи, перестановки и различного битового сдвига;
- команды `vgather[d|q]p[d|s]` и `vpgather[d|q][d|q]` загружают значения с плавающей запятой или целые числа в регистр XMM или YMM из несмежных мест в памяти. Эти команды используют режим адресации VSIB для выполнения своих операций;
- команды `vcvtph2ps` и `vcvtps2ph` выполняют преобразование упакованных значений половинной точности в значения с плавающей запятой одинарной точности;

- все команды FMA выполняют умножение с плавающей запятой с последующим сложением (или вычитанием) с плавающей запятой с использованием одной операции округления. Расширение набора инструкций x86 FMA поддерживает множество операций FMA с использованием как скалярных, так и упакованных значений с плавающей запятой одинарной или двойной точности;
- расширения набора команд ADX, BMI1, BMI2, LZCNT и POPCNT содержат команды, которые поддерживают улучшенное сложение целых чисел без знака, расширенное управление битами и операции сдвига и поворота без флагов.

Глава 9

Программирование AVX2 – упакованные числа с плавающей запятой

В главе 6 вы узнали, как использовать набор команд AVX для выполнения операций над упакованными числами с плавающей запятой с использованием набора регистров XMM и 128-битных операндов. В этой главе вы узнаете, как выполнять аналогичные операции, используя набор регистров YMM и 256-битные операнды. Глава начинается с простого примера, демонстрирующего основы арифметики упакованных чисел с плавающей запятой и использование регистра YMM. Далее следуют три примера исходного кода, которые иллюстрируют, как выполнять вычислительные операции над массивами упакованных чисел с плавающей запятой.

В главе 6 также представлены примеры исходного кода, в которых используется набор команд AVX для ускорения транспонирования и умножения матриц с применением значений с плавающей запятой одинарной точности. В этой главе вы узнаете, как выполнять те же вычисления, используя значения с плавающей запятой двойной точности. Вы также изучите пример исходного кода, который вычисляет обратную матрицу. Последние два примера исходного кода в этой главе объясняют, как выполнять смешивание, перестановку и извлечение данных с использованием упакованных операндов с плавающей запятой.

Вы можете вспомнить, что в примерах исходного кода в главе 6 с командами AVX использовались только операнды, размещенные в регистрах XMM. Это было сделано, чтобы избежать перегрузки читателя информацией и сохранить разумную длину главы. Почти все команды AVX с плавающей запятой в качестве операндов могут использовать регистры XMM или YMM. Многие примеры исходного кода в этой главе будут выполняться на процессоре, поддерживающем AVX. Имена функций в этих примерах используют префикс `Avx`. Точно так же в примерах исходного кода, которые требуют процессора, совместимого с AVX2, используется префикс имени функции `Avx2`. Вы можете использовать один из свободно доступных инструментов, перечисленных в приложении, чтобы определить, поддерживает ли ваш компьютер только AVX или AVX и AVX2.

9.1. Арифметика упакованных чисел с плавающей запятой

В листинге 9.1 приведен исходный код примера Ch09_01. В этом примере показано, как выполнять общие арифметические операции с использованием 256-битных операндов с плавающей запятой одинарной и двойной точности. Он также показывает, как использовать команду vzeroupper и несколько директив MASM для 256-битных операндов.

Листинг 9.1. Пример Ch09_01

```
//-----------------------------------------------
//              YmmVal.h
//-----------------------------------------------

#pragma once
#include <string>
#include <cstdint>
#include <sstream>
#include <iomanip>

struct YmmVal
{
public:
    union
    {
        int8_t m_I8[32];
        int16_t m_I16[16];
        int32_t m_I32[8];
        int64_t m_I64[4];
        uint8_t m_U8[32];
        uint16_t m_U16[16];
        uint32_t m_U32[8];
        uint64_t m_U64[4];
        float m_F32[8];
        double m_F64[4];
    };
//-----------------------------------------------
//               Ch09_01.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#define _USE_MATH_DEFINES
#include <math.h>
#include "YmmVal.h"

using namespace std;

extern "C" void AvxPackedMathF32_(const YmmVal& a, const YmmVal& b, YmmVal c[8]);
extern "C" void AvxPackedMathF64_(const YmmVal& a, const YmmVal& b, YmmVal c[8]);

void AvxPackedMathF32(void)
{
```

```
    alignas(32) YmmVal a;
    alignas(32) YmmVal b;
    alignas(32) YmmVal c[8];

    a.m_F32[0] = 36.0f;                b.m_F32[0] = -0.1111111f;
    a.m_F32[1] = 0.03125f;             b.m_F32[1] = 64.0f;
    a.m_F32[2] = 2.0f;                 b.m_F32[2] = -0.0625f;
    a.m_F32[3] = 42.0f;                b.m_F32[3] = 8.666667f;
    a.m_F32[4] = 7.0f;                 b.m_F32[4] = -18.125f;
    a.m_F32[5] = 20.5f;                b.m_F32[5] = 56.0f;
    a.m_F32[6] = 36.125f;              b.m_F32[6] = 24.0f;
    a.m_F32[7] = 0.5f;                 b.m_F32[7] = -98.6f;

    AvxPackedMathF32_(a, b, c);

    cout << ("\nРезультаты AvxPackedMathF32\n");

    cout << "a[0]:       " << a.ToStringF32(0) << '\n';
    cout << "b[0]:       " << b.ToStringF32(0) << '\n';
    cout << "addps[0]:   " << c[0].ToStringF32(0) << '\n';
    cout << "subps[0]:   " << c[1].ToStringF32(0) << '\n';
    cout << "mulps[0]:   " << c[2].ToStringF32(0) << '\n';
    cout << "divps[0]:   " << c[3].ToStringF32(0) << '\n';
    cout << "absps b[0]: " << c[4].ToStringF32(0) << '\n';
    cout << "sqrtps a[0]:" << c[5].ToStringF32(0) << '\n';
    cout << "minps[0]:   " << c[6].ToStringF32(0) << '\n';
    cout << "maxps[0]:   " << c[7].ToStringF32(0) << '\n';

    cout << '\n';

    cout << "a[1]:       " << a.ToStringF32(1) << '\n';
    cout << "b[1]:       " << b.ToStringF32(1) << '\n';
    cout << "addps[1]:   " << c[0].ToStringF32(1) << '\n';
    cout << "subps[1]:   " << c[1].ToStringF32(1) << '\n';
    cout << "mulps[1]:   " << c[2].ToStringF32(1) << '\n';
    cout << "divps[1]:   " << c[3].ToStringF32(1) << '\n';
    cout << "absps b[1]: " << c[4].ToStringF32(1) << '\n';
    cout << "sqrtps a[1]:" << c[5].ToStringF32(1) << '\n';
    cout << "minps[1]:   " << c[6].ToStringF32(1) << '\n';
    cout << "maxps[1]:   " << c[7].ToStringF32(1) << '\n';
}

void AvxPackedMathF64(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b;
    alignas(32) YmmVal c[8];

    a.m_F64[0] = 2.0;           b.m_F64[0] = M_PI;
    a.m_F64[1] = 4.0 ;          b.m_F64[1] = M_E;
    a.m_F64[2] = 7.5;           b.m_F64[2] = -9.125;
    a.m_F64[3] = 3.0;           b.m_F64[3] = -M_PI;

    AvxPackedMathF64_(a, b, c);
    cout << ("\nРезультаты AvxPackedMathF64\n");
```

```
    cout << "a[0]:       " << a.ToStringF64(0) << '\n';
    cout << "b[0]:       " << b.ToStringF64(0) << '\n';
    cout << "addpd[0]:   " << c[0].ToStringF64(0) << '\n';
    cout << "subpd[0]:   " << c[1].ToStringF64(0) << '\n';
    cout << "mulpd[0]:   " << c[2].ToStringF64(0) << '\n';
    cout << "divpd[0]:   " << c[3].ToStringF64(0) << '\n';
    cout << "abspd b[0]: " << c[4].ToStringF64(0) << '\n';
    cout << "sqrtpd a[0]:" << c[5].ToStringF64(0) << '\n';
    cout << "minpd[0]:   " << c[6].ToStringF64(0) << '\n';
    cout << "maxpd[0]:   " << c[7].ToStringF64(0) << '\n';

    cout << '\n';

    cout << "a[1]:       " << a.ToStringF64(1) << '\n';
    cout << "b[1]:       " << b.ToStringF64(1) << '\n';
    cout << "addpd[1]:   " << c[0].ToStringF64(1) << '\n';
    cout << "subpd[1]:   " << c[1].ToStringF64(1) << '\n';
    cout << "mulpd[1]:   " << c[2].ToStringF64(1) << '\n';
    cout << "divpd[1]:   " << c[3].ToStringF64(1) << '\n';
    cout << "abspd b[1]: " << c[4].ToStringF64(1) << '\n';
    cout << "sqrtpd a[1]:" << c[5].ToStringF64(1) << '\n';
    cout << "minpd[1]:   " << c[6].ToStringF64(1) << '\n';
    cout << "maxpd[1]:   " << c[7].ToStringF64(1) << '\n';
}

int main()
{
    AvxPackedMathF32();
    AvxPackedMathF64();
    return 0;
}
;-----------------------------------------------
;               Ch09_01.asm
;-----------------------------------------------

; Значения маски, используемые для вычисления абсолютных значений с плавающей запятой
            .const
AbsMaskF32  dword 8 dup(7fffffffh)
AbsMaskF64  qword 4 dup(7fffffffffffffffh)

; extern "C" void AvxPackedMathF32_(const YmmVal& a, const YmmVal& b, YmmVal c[8]);

            .code
AvxPackedMathF32_ proc

; Загрузка упакованных значений одинарной точности с плавающей запятой
        vmovaps ymm0,ymmword ptr [rcx]      ;ymm0 = *a
        vmovaps ymm1,ymmword ptr [rdx]      ;ymm1 = *b

; Сложение упакованных значений одинарной точности с плавающей запятой
        vaddps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8],ymm2

; Вычитание упакованных значений одинарной точности с плавающей запятой
        vsubps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+32],ymm2
```

```
; Умножение упакованных значений одинарной точности с плавающей запятой
        vmulps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+64],ymm2

; Деление упакованных значений одинарной точности с плавающей запятой
        vdivps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+96],ymm2

; Абсолютное значение упакованных значений одинарной точности с плавающей запятой (b)
        vandps ymm2,ymm1,ymmword ptr [AbsMaskF32]
        vmovaps ymmword ptr [r8+128],ymm2

; Квадратный корень из упакованных значений одинарной точности с плавающей запятой (a)
        vsqrtps ymm2,ymm0
        vmovaps ymmword ptr [r8+160],ymm2

; Минимум упакованных значений одинарной точности с плавающей запятой
        vminps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+192],ymm2

; Максимум упакованных значений одинарной точности с плавающей запятой
        vmaxps ymm2,ymm0,ymm1
        vmovaps ymmword ptr [r8+224],ymm2

        vzeroupper
        ret
AvxPackedMathF32_ endp

; extern "C" void AvxPackedMathF64_(const YmmVal& a, const YmmVal& b, YmmVal c[8]);

AvxPackedMathF64_ proc

; Загрузка упакованных значений двойной точности с плавающей запятой
        vmovapd ymm0,ymmword ptr [rcx]       ;ymm0 = *a
        vmovapd ymm1,ymmword ptr [rdx]       ;ymm1 = *b

; Сложение упакованных значений двойной точности с плавающей запятой
        vaddpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8],ymm2

; Вычитание упакованных значений двойной точности с плавающей запятой
        vsubpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+32],ymm2

; Умножение упакованных значений двойной точности с плавающей запятой
        vmulpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+64],ymm2

; Деление упакованных значений двойной точности с плавающей запятой
        vdivpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+96],ymm2

; Абсолютное значение упакованных значений двойной точности с плавающей запятой (b)
        vandpd ymm2,ymm1,ymmword ptr [AbsMaskF64]
        vmovapd ymmword ptr [r8+128],ymm2
```

```
; Квадратный корень из упакованных значений двойной точности с плавающей запятой (a)
        vsqrtpd ymm2,ymm0
        vmovapd ymmword ptr [r8+160],ymm2

; Минимум упакованных значений двойной точности с плавающей запятой
        vminpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+192],ymm2

; Максимум упакованных значений двойной точности с плавающей запятой
        vmaxpd ymm2,ymm0,ymm1
        vmovapd ymmword ptr [r8+224],ymm2

        vzeroupper
        ret
AvxPackedMathF64_ endp
        end
```

Листинг 9.1 начинается с объявления структуры C++ с именем `YmmVal`, которая объявлена в заголовочном файле `YmmVal.h`. Эта структура похожа на структуру `XmmVal`, которую вы видели в главе 6. `YmmVal` содержит публичное анонимное объединение, которое облегчает обмен упакованными данными операндов между функциями, написанными на C++ и языке ассемблера x86. Члены этого объединения соответствуют типам упакованных данных, которые могут использоваться с регистром YMM. Структура `YmmVal` также включает несколько функций форматирования и отображения (исходный код для этих функций-членов в листинге не показан).

Код C++ в примере `Ch09_01` начинается с объявлений функций языка ассемблера `AvxPackedMathF32_` и `AvxPackedMathF64_`. Эти функции выполняют различные арифметические операции над упакованными числами с плавающей запятой одинарной и двойной точности с использованием предоставленных аргументов `YmmVal`. За объявлениями функций на языке ассемблера следует функция `AvxPackedMathF32`. Эта функция начинается с инициализации переменных `YmmVal` `a` и `b`. Обратите внимание, что спецификатор C++ `alignas(32)` используется с каждым объявлением `YmmVal`. Этот спецификатор инструктирует компилятор C++ выровнять каждую переменную `YmmVal` по 32-байтовой границе. После инициализации переменной `YmmVal` функция `AvxPackedMathF32` вызывает функцию языка ассемблера `AvxPackedMathF32_` для выполнения арифметических действий. Затем результаты передаются в `cout`. Функция `AvxPackedMathF64` является аналогом `AvxPackedMathF32` для чисел с плавающей запятой двойной точности.

В верхней части ассемблерного кода в листинге 9.1 находится раздел `.const`, который определяет упакованные значения констант для вычисления абсолютных значений с плавающей запятой. Мнемоника `dup` – это оператор MASM, который выделяет и при необходимости инициализирует несколько значений данных. В текущем примере команда `AbsMaskF32 dword 8 dup(7fffffffh)` выделяет пространство для хранения восьми значений двойного слова, и каждое значение инициализируется значением `0x7fffffff`. Следующий оператор `AbsMaskF64 qword 4 dup(7fffffffffffffffh)` выделяет четыре четверных слова `0x7fffffffffffffff`. Обратите внимание, что ни одному из этих 256-битных операндов не предшествует оператор `align`, то есть они могут быть неправильно

выровнены в памяти. Причина этого в том, что директива MASM `align` не поддерживает 32-байтовое выравнивание в разделах `.const`, `.data` или `.code`. Позже в этой главе вы узнаете, как определить настраиваемый сегмент постоянных значений, поддерживающий 32-байтовое выравнивание.

Следом за разделом `.const` первая команда функции `AvxPackedMathF32_`, `vmovaps ymm0,ymmword ptr [rcx]`, загружает аргумент `a` (т. е. восемь значений `YmmVal a` с плавающей запятой) в регистр YMM0. Здесь можно использовать `vmovaps`, поскольку `YmmVal a` был определен с использованием спецификатора `alignas(32)` в коде C++. Оператор `ymmword ptr` указывает ассемблеру обрабатывать ячейку памяти, на которую указывает RCX, как 256-битный операнд. Использование оператора `ymmword ptr` в этом случае необязательно и используется для улучшения читаемости кода. Следующая команда `vmovaps ymm1,ymmword ptr [rdx]` загружает `b` в регистр YMM1. Далее команда `vaddps ymm2,ymm0,ymm1` суммирует упакованные значения с плавающей запятой одинарной точности в YMM0 и YMM1; затем она сохраняет результат в YMM2. Команда `vmovaps ymmword ptr [r8],ymm2` сохраняет упакованные суммы в `c[0]`.

Последующие команды `vsubps`, `vmulps` и `vdivps` выполняют упакованное вычитание с плавающей запятой одинарной точности, умножение и деление. За этим следует команда `vandps ymm2,ymm1,ymmword ptr [AbsMaskF32]`, которая вычисляет упакованные абсолютные значения с использованием аргумента `b`. Остальные команды в `AvxPackedMathF32_` вычисляют упакованные квадратные корни с плавающей запятой одинарной точности, минимумы и максимумы.

Перед своей командой `ret` функция `AvxPackedMath32_` использует команду `vzeroupper`, которая обнуляет старшие 128 бит каждого регистра YMM. Как объяснялось в главе 4, команда `vzeroupper` здесь необходима, чтобы избежать потенциальных задержек производительности, которые могут возникнуть всякий раз, когда процессор переходит от выполнения команд x86-AVX, использующих операнды шириной 256 бит, к выполнению команд x86-SSE. Любая функция языка ассемблера, которая использует один или несколько регистров YMM и вызывается из кода, потенциально использующего команды x86-SSE, всегда должна обеспечивать выполнение команды `vzeroupper` до того, как управление программой будет передано обратно вызывающей функции. В этой и последующих главах вы увидите дополнительные примеры использования команд `vzeroupper`.

Структура функции `AvxPackedMathF64_` аналогична `AvxPackedMathF32_`. `AvxPackedMathF64_` выполняет свои вычисления, используя версии с двойной точностью тех же команд, которые применяются в `AvxPackedMathF32_`. Вот результат выполнения примера исходного кода `Ch09_01`:

```
Результаты AvxPackedMathF32
a[0]:              36.000000         0.031250  |   2.000000      42.000000
b[0]:              -0.111111        64.000000  |  -0.062500       8.666667
addps[0]:          35.888889        64.031250  |   1.937500      50.666668
subps[0]:          36.111111       -63.968750  |   2.062500      33.333332
mulps[0]:          -4.000000         2.000000  |  -0.125000     364.000000
divps[0]:        -324.000031         0.000488  | -32.000000       4.846154
absps b[0]:         0.111111        64.000000  |   0.062500       8.666667
sqrtps a[0]:        6.000000         0.176777  |   1.414214       6.480741
```

```
minps[0]:          -0.111111         0.031250  |  -0.062500       8.666667
maxps[0]:          36.000000        64.000000  |   2.000000      42.000000

a[1]:               7.000000        20.500000  |  36.125000       0.500000
b[1]:             -18.125000        56.000000  |  24.000000     -98.599998
addps[1]:         -11.125000        76.500000  |  60.125000     -98.099998
subps[1]:          25.125000       -35.500000  |  12.125000      99.099998
mulps[1]:        -126.875000      1148.000000  | 867.000000     -49.299999
divps[1]:          -0.386207         0.366071  |   1.505208      -0.005071
absps b[1]:        18.125000        56.000000  |  24.000000      98.599998
sqrtps a[1]:        2.645751         4.527693  |   6.010407       0.707107
minps[1]:         -18.125000        20.500000  |  24.000000     -98.599998
maxps[1]:           7.000000        56.000000  |  36.125000       0.500000

Результаты AvxPackedMathF64
a[0]:                          2.000000000000  |            4.000000000000
b[0]:                          3.141592653590  |            2.718281828459
addpd[0]:                      5.141592653590  |            6.718281828459
subpd[0]:                     -1.141592653590  |            1.281718171541
mulpd[0]:                      6.283185307180  |           10.873127313836
divpd[0]:                      0.636619772368  |            1.471517764686
abspd b[0]:                    3.141592653590  |            2.718281828459
sqrtpd a[0]:                   1.414213562373  |            2.000000000000
minpd[0]:                      2.000000000000  |            2.718281828459
maxpd[0]:                      3.141592653590  |            4.000000000000

a[1]:                          7.500000000000  |            3.000000000000
b[1]:                         -9.125000000000  |           -3.141592653590
addpd[1]:                     -1.625000000000  |           -0.141592653590
subpd[1]:                     16.625000000000  |            6.141592653590
mulpd[1]:                    -68.437500000000  |           -9.424777960769
divpd[1]:                     -0.821917808219  |           -0.954929658551
abspd b[1]:                    9.125000000000  |            3.141592653590
sqrtpd a[1]:                   2.738612787526  |            1.732050807569
minpd[1]:                     -9.125000000000  |           -3.141592653590
maxpd[1]:                      7.500000000000  |            3.000000000000
```

9.2. Массивы упакованных чисел с плавающей запятой

В предыдущих главах вы узнали, как выполнять вычисления, в которых задействованы элементы массивов – как целочисленные, так и с плавающей запятой, – используя наборы регистров общего назначения и XMM. В этом разделе вы узнаете, как выполнять операции с участием элементов массивов с плавающей запятой, используя набор регистров YMM.

9.2.1. Простые вычисления

В листинге 9.2 показан исходный код примера `Ch09_02`. В этом примере показано, как выполнять простые вычисления элементов массива с использованием 256-битных упакованных операндов с плавающей запятой. Он также демонстрирует, как обнаруживать и исключать недопустимые элементы массива из вычислений. Пример исходного кода `Ch09_02` – это альтернативная реализация примера `Ch05_02` из главы 5, в котором вычислялись площадь поверхно-

сти и объем сферы. В этом примере функция на языке ассемблера `CalcSphereAreaVolume_` вычисляет площадь поверхности и объем нескольких сфер. Радиусы сфер передаются через массив вычисляющим функциям, написанным с использованием C++ и языка ассемблера. Чтобы сделать пример немного более интересным, вычисляющие функции C++ и ассемблера проверяют радиусы на значение меньше нуля. Если обнаружен недопустимый радиус, вычислительные функции устанавливают для соответствующих элементов в массивах площади поверхности и объема значение `QNaN`.

Листинг 9.2. Пример Ch09_02

```
//-----------------------------------------------
//              Ch09_02.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <limits>
#define _USE_MATH_DEFINES
#include <math.h>

using namespace std;

extern "C" void AvxCalcSphereAreaVolume_(float* sa, float* vol, const float* r, size_t n);

extern "C" float c_PI_F32 = (float)M_PI;
extern "C" float c_QNaN_F32 = numeric_limits<float>::quiet_NaN();

void Init(float* r, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 100};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        r[i] = (float)ui_dist(rng) / 10.0f;

    // Некорректное значение радиуса для целей тестирования
    if (n > 2)
    {
        r[2] = -r[2];
        r[n / 4] = -r[n / 4];
        r[n / 2] = -r[n / 2];
        r[n / 4 * 3] = -r[n / 4 * 3];
        r[n - 2] = -r[n - 2];
    }
}

void AvxCalcSphereAreaVolumeCpp(float* sa, float* vol, const float* r, size_t n)
{
    for (size_t i = 0; i < n; i++)
    {
        if (r[i] < 0.0f)
            sa[i] = vol[i] = c_QNaN_F32;
```

```
        else
        {
            sa[i] = r[i] * r[i] * 4.0f * c_PI_F32;
            vol[i] = sa[i] * r[i] / 3.0f;
        }
    }
}

void AvxCalcSphereAreaVolume(void)
{
    const size_t n = 21;
    alignas(32) float r[n];
    alignas(32) float sa1[n];
    alignas(32) float vol1[n];
    alignas(32) float sa2[n];
    alignas(32) float vol2[n];

    Init(r, n, 93);

    AvxCalcSphereAreaVolumeCpp(sa1, vol1, r, n);
    AvxCalcSphereAreaVolume_(sa2, vol2, r, n);

    cout << "\nРезультаты AvxCalcSphereAreaVolume\n";
    cout << fixed;

    const float eps = 1.0e-6f;

    for (size_t i = 0; i < n; i++)
    {
        cout << setw(2) << i << ":  ";
        cout << setprecision(2);
        cout << setw(5) << r[i] << " | ";
        cout << setprecision(6);
        cout << setw(12) << sa1[i] << "   ";
        cout << setw(12) << sa2[i] << " | ";
        cout << setw(12) << vol1[i] << "   ";
        cout << setw(12) << vol2[i];

        bool b0 = (fabs(sa1[i] - sa2[i]) > eps);
        bool b1 = (fabs(vol1[i] - vol2[i]) > eps);

        if (b0 || b1)
            cout << " Расхождение значений";
        cout << '\n';
    }
}

int main()
{
    AvxCalcSphereAreaVolume();
    return 0;
}
;-----------------------------------------------
;              Ch09_02.asm
;-----------------------------------------------
```

```
        include <cmpequ.asmh>
        include <MacrosX86-64-AVX.asmh>

        .const
r4_3p0  real4 3.0
r4_4p0  real4 4.0

        extern c_PI_F32:real4
        extern c_QNaN_F32:real4

; extern "C" void AvxCalcSphereAreaVolume_(float* sa, float* vol, const float* r, size_t
n);

        .code
AvxCalcSphereAreaVolume_ proc frame
        _CreateFrame CC_,0,64
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9
        _EndProlog

; Инициализация
        vbroadcastss ymm0,real4 ptr [r4_4p0]        ;упакованное 4.0
        vbroadcastss ymm1,real4 ptr [c_PI_F32]      ;упакованное PI
        vmulps ymm6,ymm0,ymm1                       ;упакованное 4.0 * PI
        vbroadcastss ymm7,real4 ptr [r4_3p0]        ;упакованное 3.0
        vbroadcastss ymm8,real4 ptr [c_QNaN_F32]    ;упакованное QNaN
        vxorps ymm9,ymm9,ymm9                       ;упакованное 0.0

        xor eax,eax                             ;общее смещение для массивов

        cmp r9,8
        jb FinalR                               ;пропустить главный цикл, если n < 8

; Вычисление значений площади поверхности и объема с использованием упакованных операндов
@@:     vmovdqa ymm0,ymmword ptr [r8+rax]       ;загрузить следующие 8 радиусов
        vmulps ymm2,ymm6,ymm0                   ;4.0 * PI * r
        vmulps ymm3,ymm2,ymm0                   ;4.0 * PI * r * r

        vcmpps ymm1,ymm0,ymm9,CMP_LT            ;ymm1 = маска радиуса < 0.0

        vandps ymm4,ymm1,ymm8                   ;площадь равна QNaN, если радиус < 0.0
        vandnps ymm5,ymm1,ymm3                  ;сохранить площадь, если радиус >= 0.0
        vorps ymm5,ymm4,ymm5                    ;окончательное упакованное значение площади
        vmovaps ymmword ptr[rcx+rax],ymm5       ;сохранение упакованного значения площади

        vmulps ymm2,ymm3,ymm0                   ;4.0 * PI * r * r * r
        vdivps ymm3,ymm2,ymm7                   ;4.0 * PI * r * r * r / 3.0
        vandps ymm4,ymm1,ymm8                   ;объем равен QNaN, если радиус < 0.0
        vandnps ymm5,ymm1,ymm3                  ;сохранить объем, если радиус >= 0.0
        vorps ymm5,ymm4,ymm5                    ;окончательное упакованное значение объема
        vmovaps ymmword ptr[rdx+rax],ymm5       ;сохранение упакованного значения объема

        add rax,32                              ;rax = смещение для нового набора радиусов
        sub r9,8
        cmp r9,8
        jae @B                                  ;повторять, пока не будет n < 8
```

```
; Финальные вычисления с применением скалярной арифметики
FinalR: test r9,r9
        jz Done                                 ;пропустить цикл, если больше нет элементов

@@:     vmovss xmm0,real4 ptr [r8+rax]
        vmulss xmm2,xmm6,xmm0                   ;4.0 * PI * r
        vmulss xmm3,xmm2,xmm0                   ;4.0 * PI * r * r

        vcmpss xmm1,xmm0,xmm9,CMP_LT

        vandps xmm4,xmm1,xmm8
        vandnps xmm5,xmm1,xmm3
        vorps xmm5,xmm4,xmm5
        vmovss real4 ptr[rcx+rax],xmm5          ;сохранение площади

        vmulss xmm2,xmm3,xmm0                   ;4.0 * PI * r * r * r
        vdivss xmm3,xmm2,xmm7                   ;4.0 * PI * r * r * r / 3.0
        vandps xmm4,xmm1,xmm8
        vandnps xmm5,xmm1,xmm3
        vorps xmm5,xmm4,xmm5
        vmovss real4 ptr[rdx+rax],xmm5          ;сохранение объема

        add rax,4
        dec r9
        jnz @B                                  ;повторять до завершения

Done:   vzeroupper

        _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9
        _DeleteFrame
        ret
AvxCalcSphereAreaVolume_ endp
        end
```

Код C++ в листинге 9.2 содержит функцию `AvxCalcSphereAreaVolumeCpp`. Эта функция вычисляет площадь поверхности и объем сферы. Радиусы сферы передаются в `AvxCalcSphereAreaVolumeCpp` через массив. Перед вычислением площади поверхности или объема радиус сферы (`r[i]`) проходит проверку, чтобы убедиться, что он не отрицательный. Если радиус отрицательный, соответствующие элементы в массивах площади поверхности и объема (`sa[i]` и `vol[i]`) устанавливаются в `c_QNaN_F32`. Оставшийся код C++ выполняет необходимые инициализации, запускает вычислительные функции C++ и на языке ассемблера и отображает результаты. Обратите внимание, что функция `AvxCalcSphereAreaVolume` использует спецификатор `alignas(32)` с каждым объявлением массива.

Функция на языке ассемблера `AvxCalcSphereAreaVolume_` выполняет те же вычисления, что и ее аналог на языке C++. Следом за прологом `AvxCalcSphereAreaVolume_` использует серию команд `vbroadcastss` для инициализации упакованных версий необходимых констант. Перед началом рабочего цикла команда `cmp r9,8` проверяет значение `n`. Необходимость этой проверки заключается в том, что цикл обработки одновременно выполняет восемь вычислений площади поверхности и объема с использованием операндов шириной 256 бит. Команда условного перехода `jb FinalR` пропускает рабочий цикл, если нужно обработать менее восьми радиусов.

Каждая итерация рабочего цикла начинается с команды `vmovdqa ymm0,ymmword ptr [r8+rax]`, которая загружает восемь значений радиусов с плавающей запятой одинарной точности в регистр YMM0. Далее команды `vmulps` вычисляют площадь поверхности сферы. Следующая команда, `vcmpps ymm1,ymm0,ymm9,CMP_LT`, проверяет радиус каждой сферы на значение меньше 0,0 (регистр YMM9 содержит упакованное значение 0,0). Напомним, что команда `vcmpps` обозначает результат сравнения, устанавливая для элементов в операнде назначения значение `0x00000000` (ложный предикат сравнения) или `0xffffffff` (истинный предикат сравнения). Затем команды `vandps`, `vandnps` и `vorps` присваивают площади поверхности каждой сферы с радиусом меньше 0,0 значение `c_QNaN_F32`. Рисунок 9.1 наглядно представляет эту операцию. Команда `vmovaps ymmword ptr [rcx+rax],ymm5` сохраняет восемь значений площади поверхности сферы в массив `sa`.

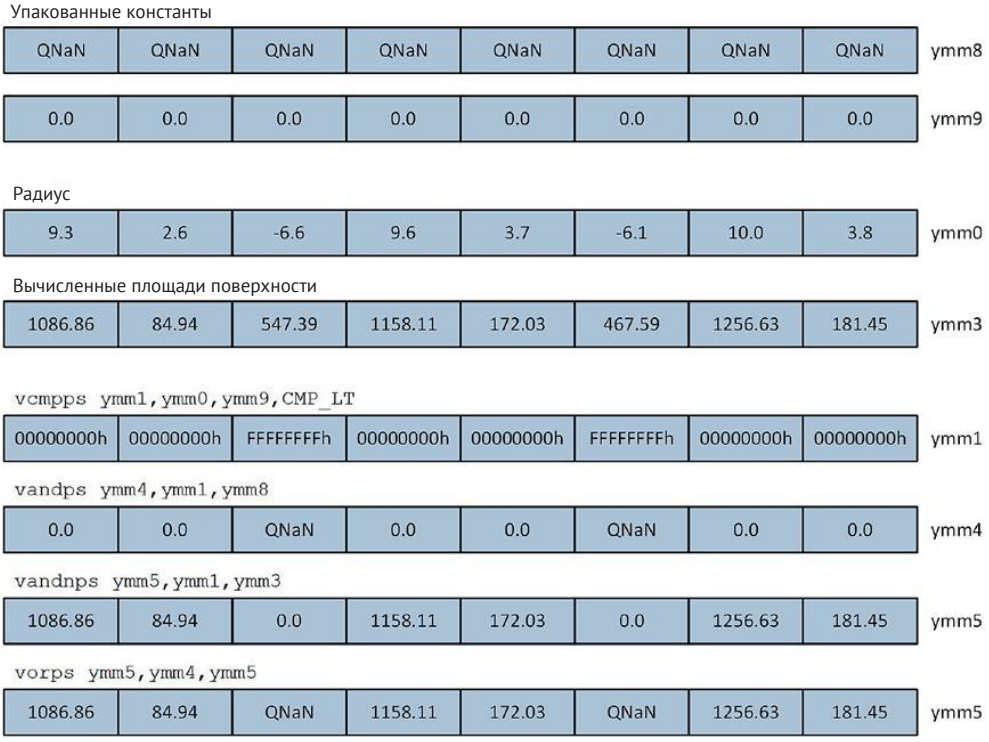


Рис. 9.1. Присвоение значения `QNaN` площади поверхности для сфер радиусом менее 0,0

После вычисления площадей команды `vmulps ymm2,ymm3,ymm0` и `vdivps ymm3,ymm2,ymm7` вычисляют объемы сфер. Здесь рабочий цикл использует еще одну последовательность команд `vandps`, `vandnps` и `vorps` для присвоения объему сферы отрицательного радиуса значения `c_QNaN_F32`. Эти значения затем сохраняются в массиве `vol`. Итерации рабочего цикла повторяются до тех пор, пока не останется менее восьми радиусов.

В следующем блоке кода вычисляются площади поверхности и объемы сфер для оставшихся (1…7) радиусов. Обратите внимание, что `AvxCalcSphereAreaVolume_`

выполняет эти вычисления, используя скалярную арифметику с плавающей запятой одинарной точности. Рабочий цикл скалярной арифметики выполняет те же арифметические и логические операции, что и цикл упакованной арифметики. Как и в предыдущем примере, `AvxCalcSphereAreaVolume_` использует команду `vzeroupper` сразу после цикла скалярной обработки. Эта команда необходима, поскольку `AvxCalcSphereAreaVolume_` выполнила свои вычисления с использованием набора регистров YMM. Когда требуется команда `vzeroupper`, ее всегда следует размещать перед любыми макросами эпилога функции (например, `_RestoreXmmRegs` и `_DeleteFrame`) и командой `ret`. Вот результаты выполнения примера исходного кода `Ch09_02`:

```
Результаты AvxCalcSphereAreaVolume
0:    3.80 |   181.458389     181.458389 |   229.847290     229.847290
1:   10.00 |  1256.637085    1256.637085 |  4188.790527    4188.790527
2:   -6.10 |          nan            nan |          nan            nan
3:    3.70 |   172.033630     172.033630 |   212.174805     212.174805
4:    9.60 |  1158.116821    1158.116821 |  3705.973877    3705.973877
5:   -6.60 |          nan            nan |          nan            nan
6:    2.60 |    84.948662      84.948654 |    73.622169      73.622162 Расхождение значений
7:    9.30 |  1086.865479    1086.865479 |  3369.283203    3369.283203
8:    9.00 |  1017.876038    1017.876038 |  3053.628174    3053.628174
9:    5.80 |   422.732758     422.732758 |   817.283386     817.283386
10:  -2.90 |          nan            nan |          nan            nan
11:   8.10 |   824.479675     824.479675 |  2226.095215    2226.095215
12:   3.00 |   113.097336     113.097336 |   113.097328     113.097328
13:   8.00 |   804.247742     804.247742 |  2144.660645    2144.660645
14:   1.40 |    24.630087      24.630085 |    11.494040      11.494039 Расхождение значений
15:  -1.80 |          nan            nan |          nan            nan
16:   4.30 |   232.352219     232.352219 |   333.038177     333.038177
17:   6.60 |   547.391113     547.391113 |  1204.260376    1204.260376
18:   4.50 |   254.469009     254.469009 |   381.703522     381.703522
19:  -1.20 |          nan            nan |          nan            nan
20:   4.50 |   254.469009     254.469009 |   381.703522     381.703522
```

Выходные данные для примера исходного кода `Ch09_02` содержат пару строк с текстом «Расхождение значений». Этот текст был сгенерирован кодом сравнения в `AvxCalcSphereAreaVolume`, чтобы продемонстрировать неассоциативность арифметики с плавающей запятой. В этом примере функции `AvxCalcSphereAreaVolumeCpp` и `AvxCalcSphereAreaVolume_` выполнили свои соответствующие вычисления с плавающей запятой, используя различный порядок операндов. Для каждой площади поверхности сферы код C++ вычисляет `sa[i]=r[i]*r[i]*4.0*c_PI_F32`, а код языка ассемблера вычисляет `sa[i]=4.0*c_PI_F32*r[i]*r[я]`. Подобные крошечные расхождения чисел не являются чем-то необычным при сравнении значений с плавающей запятой, которые вычисляются с использованием разного порядка операндов независимо от языка программирования. Вы должны иметь это в виду, если разрабатываете производственный код, который включает несколько версий одной и той же вычислительной функции (например, одну, написанную с использованием C++, и ускоренную версию AVX/AVX2, реализованную с использованием языка ассемблера x86).

Наконец, вы могли заметить, что функция AvxCalcSphereAreaVolume_ обрабатывает недопустимые радиусы без каких-либо команд условного перехода x86. Сведение к минимуму количества команд условного перехода в функциях, особенно зависимых от данных, часто приводит к более быстрому выполнению кода. Вы узнаете больше о методах оптимизации команд перехода в главе 15.

9.2.2. Среднее арифметическое значение столбца

В листинге 9.3 приведен исходный код примера Ch09_03. В этом примере показано, как вычислить среднее арифметическое для каждого столбца в двумерном массиве значений с плавающей запятой двойной точности.

Листинг 9.3. Пример Ch09_03

```
//-----------------------------------------------
//               Ch09_03.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <memory>

using namespace std;

extern "C" size_t c_NumRowsMax = 1024 * 1024;
extern "C" size_t c_NumColsMax = 1024 * 1024;

extern "C" bool AvxCalcColumnMeans_(const double* x, size_t nrows, size_t ncols, double*
col_means);

void Init(double* x, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 2000};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (double)ui_dist(rng) / 10.0;
}

bool AvxCalcColumnMeansCpp(const double* x, size_t nrows, size_t ncols, double* col_means)
{
    // Проверка значений nrows и ncols
    if (nrows == 0 || nrows > c_NumRowsMax)
        return false;
    if (ncols == 0 || ncols > c_NumColsMax)
        return false;

    // Инициализация среднего значения (ноль)
    for (size_t i = 0; i < ncols; i++)
        col_means[i] = 0.0;

    // Вычисление среднего значения столбца
    for (size_t i = 0; i < nrows; i++)
```

```
    {
        for (size_t j = 0; j < ncols; j++)
            col_means[j] += x[i * ncols + j];
    }

    for (size_t j = 0; j < ncols; j++)
        col_means[j] /= nrows;

    return true;
}

void AvxCalcColumnMeans(void)
{
    const size_t nrows = 20;
    const size_t ncols = 11;
    unique_ptr<double[]> x {new double[nrows * ncols]};
    unique_ptr<double[]> col_means1 {new double[ncols]};
    unique_ptr<double[]> col_means2 {new double[ncols]};

    Init(x.get(), nrows * ncols, 47);

    bool rc1 = AvxCalcColumnMeansCpp(x.get(), nrows, ncols, col_means1.get());
    bool rc2 = AvxCalcColumnMeans_(x.get(), nrows, ncols, col_means2.get());

    cout << "Результаты AvxCalcColumnMeans\n";

    if (!rc1 || !rc2)
    {
        cout << "Возвращен код ошибки: ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        return;
    }

    cout << "\nТестовая матрица\n";
    cout << fixed << setprecision(1);

    for (size_t i = 0; i < nrows; i++)
    {
        cout << "row " << setw(2) << i;

        for (size_t j = 0; j < ncols; j++)
            cout << setw(7) << x[i * ncols + j];
        cout << '\n';
    }

    cout << "\nСреднее значение столбца\n";
    cout << setprecision(2);

    for (size_t j = 0; j < ncols; j++)
    {
        cout << "col_means1[" << setw(2) << j << "] =";
        cout << setw(10) << col_means1[j] << "   ";
        cout << "col_means2[" << setw(2) << j << "] =";
        cout << setw(10) << col_means2[j] << '\n';
    }
}
```

```
int main()
{
    AvxCalcColumnMeans();
    return 0;
}
;-----------------------------------------------
;              Ch09_03.asm
;-----------------------------------------------

; extern "C" bool AvxCalcColMeans_(const double* x, size_t nrows, size_t ncols, double*
col_means)

        extern c_NumRowsMax:qword
        extern c_NumColsMax:qword

        .code
AvxCalcColumnMeans_ proc

; Проверка nrows и ncols
        xor eax,eax                     ;возвращаемый код ошибки (и индекс col_mean)
        test rdx,rdx
        jz Done                         ;переход, если nrows равен нулю
        cmp rdx,[c_NumRowsMax]
        ja Done                         ;переход, если nrows слишком велик
        test r8,r8
        jz Done                         ;переход, если ncols равен нулю
        cmp r8,[c_NumColsMax]
        ja Done                         ;переход, если ncols слишком велик

; Инициализация col_means нулевым значением
        vxorpd xmm0,xmm0,xmm0           ;xmm0[63:0] = 0.0
@@:     vmovsd real8 ptr[r9+rax*8],xmm0 ;col_means[i] = 0.0
        inc rax
        cmp rax,r8
        jb @B                           ;повторять до завершения

        vcvtsi2sd xmm2,xmm2,rdx       ;преобразование nrows для дальнейшего использования

; Вычисление суммы каждого столбца в x
LP1:    mov r11,r9                      ;r11 = указатель на col_means
        xor r10,r10                     ;r10 = col_index

LP2:    mov rax,r10                     ;rax = col_index
        add rax,4
        cmp rax,r8                      ;осталось ли 4 или более столбцов?
        ja @F                           ;переход, если не (col_index + 4 > ncols)

; Обновление col_means для следующих четырех столбцов
        vmovupd ymm0,ymmword ptr [rcx]      ;загрузить 4 столбца текущей строки
        vaddpd ymm1,ymm0,ymmword ptr [r11]  ;прибавить к col_means
        vmovupd ymmword ptr [r11],ymm1      ;сохранить обновленный col_means
        add r10,4                           ;col_index += 4
        add rcx,32                          ;обновить указатель x
        add r11,32                          ;обновить указатель col_means
        jmp NextColSet
```

```
@@:     sub rax,2
        cmp rax,r8                      ;осталось ли 2 или более строк?
        ja @F                           ;перейти, если не (col_index + 2 > ncols)

; Обновить col_means, используя следующие два столбца
        vmovupd xmm0,xmmword ptr [rcx]      ;загрузить 2 столбца текущей строки
        vaddpd xmm1,xmm0,xmmword ptr [r11]  ;прибавить к col_means
        vmovupd xmmword ptr [r11],xmm1      ;сохранить обновленный col_means
        add r10,2                       ;col_index += 2
        add rcx,16                      ;обновить указатель x
        add r11,16                      ;обновить указатель col_means
        jmp NextColSet

; Обновить col_means значением из следующего столбца (или последнего в текущей строке)
@@:     vmovsd xmm0,real8 ptr [rcx]         ;загрузить x из последнего столбца
        vaddsd xmm1,xmm0,real8 ptr [r11]    ;прибавить к col_means
        vmovsd real8 ptr [r11],xmm1         ;сохранить обновленный col_means
        inc r10                         ;col_index += 1
        add rcx,8                       ;обновить указатель x

NextColSet:
        cmp r10,r8                      ;остались столбцы в текущей строке?
        jb LP2                          ;переход, если да
        dec rdx                         ;nrows -= 1
        jnz LP1                         ;переход, если остались строки

; Вычисление окончательного col_means
@@:     vmovsd xmm0,real8 ptr [r9]          ;xmm0 = col_means[i]
        vdivsd xmm1,xmm0,xmm2               ;вычисление финального среднего значения
        vmovsd real8 ptr [r9],xmm1          ;сохранить col_mean[i]
        add r9,8                        ;обновить col_means
        dec r8                          ;ncols -= 1
        jnz @B                          ;повторять до завершения

        mov eax,1                       ;возвращаемый код успешного выполнения

Done:   vzeroupper
        ret

AvxCalcColumnMeans_ endp
        end
```

В верхней части кода C++ находится функция `AvxCalcColumnMeansCpp`. Эта функция вычисляет средние значения по столбцам двумерного массива, используя простой набор вложенных циклов `for` и базовую арифметику. Функция `AvxCalcColumnMeans` содержит код, который применяет класс интеллектуального указателя C++ `unique_ptr<>` для управления своими динамически выделяемыми массивами. Обратите внимание, что пространство для хранения для тестового массива `x` выделяется с помощью оператора `new` C++, что означает, что массив не может быть выровнен по 16- или 32-байтовой границе. В этом конкретном примере выравнивание начала массива `x` по определенной границе принесет мало пользы, поскольку невозможно выровнять отдельные строки или столбцы стандартного двумерного массива C++ (напомним, что элементы двумерного массива C++ хранятся в непрерывном блоке памяти с использованием строкового порядка, как описано в главе 2).

Функция AvxCalcColumnMeans также использует класс unique_ptr<> и оператор new для одномерных массивов col_means1 и col_means2. Использование unique_ptr<> в этом примере несколько упрощает код C++, поскольку его деструктор автоматически вызывает оператор delete[], чтобы освободить пространство памяти, выделенное оператором new. Если вам интересно узнать больше о классе интеллектуального указателя unique_ptr<>, в приложении содержится список ссылок на C++, с которыми вы можете ознакомиться. Оставшийся код в AvxCalcColumnMeans вызывает функции вычисления средних значений столбцов на языке C++ и ассемблере и передает результаты в cout.

После проверки аргумента ассемблерная функция AvxCalcColMeans_ инициализирует каждый элемент в col_means значением 0,0. Эти элементы будут поддерживать суммы промежуточных столбцов. Чтобы максимизировать быстродействие, код суммирования столбцов использует несколько разные последовательности команд в зависимости от текущего столбца и общего количества столбцов в массиве. Например, предположим, что массив x содержит семь столбцов. Для каждой строки элементы первых четырех столбцов в x могут быть добавлены в col_means с использованием 256-битного упакованного сложения; элементы следующих двух столбцов могут быть добавлены в col_means с использованием 128-битного упакованного сложения; и последний столбец должен быть прибавлен к col_means при помощи скалярного сложения. Этот метод наглядно представлен на рис. 9.2.

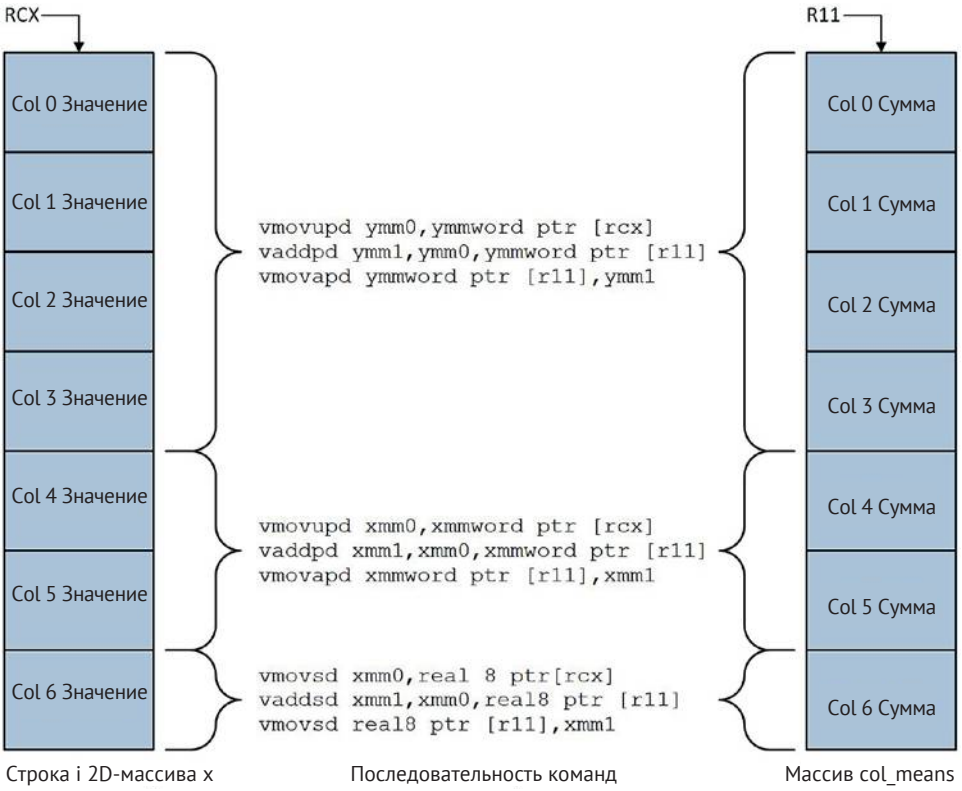


Рис. 9.2. Обновление массива col_means с использованием разных размеров операндов

Команда `mov r11,r9` рядом с меткой `LP1` является отправной точкой для добавления элементов в текущей строке `x` в `col_means`. Эта команда инициализирует R11 первой записью в `col_means`. Счетчик `col_index` в регистре R10 затем устанавливается в ноль. Группа команд рядом с меткой `LP2` определяет количество столбцов, оставшихся для обработки в текущей строке. Если остается четыре или более столбца, следующие четыре элемента из текущей строки добавляются к суммам столбцов в `col_means`. Инструкция `vmovupd ymm0,ymmword ptr [rcx]` загружает четыре значения с плавающей запятой двойной точности из `x` в YMM0 (команда `vmovapd` здесь не используется, поскольку выравнивание элементов неизвестно). Следующая команда `vaddpd ymm1, ymm0,ymmword ptr [r11]` суммирует текущие элементы массива с соответствующими элементами в `col_means`, а команда `vmovupd ymmword ptr [r11],ymm1` сохраняет обновленные результаты обратно в `col_means`. Затем обновляются различные указатели и счетчики функции для подготовки следующего набора элементов из текущей строки `x`.

Код суммирования повторяет шаги, описанные в предыдущем абзаце, до тех пор, пока количество элементов массива, которые остаются в текущей строке, не станет меньше четырех. Как только это условие выполнено, элементы в оставшихся столбцах (если есть) должны быть обработаны с использованием 128-битных или 64-битных операндов. Это причина того, что отдельные блоки кода в `AvxCalcColumnMeans_` обрабатывают четыре элемента, два элемента или один элемент в строке. После вычисления сумм столбцов каждый элемент `col_means` делится на `n`, что дает окончательное среднее значение столбца. Вот результаты выполнения примера исходного кода `Ch09_03`:

```
Результаты AvxCalcColumnMeans
row 0   125.6   59.9    100.0   170.5   140.1   197.2   73.7    15.2    92.4    155.3
159.2
row 1   77.6    105.4   45.0    176.8   65.9    12.3    189.1   102.0   56.2    112.8
17.2
row 2   198.9   199.3   74.6    137.9   65.0    125.0   19.8    32.1    58.6    94.1
123.5
row 3   1.7     29.1    99.1    200.0   109.0   123.7   130.0   125.3   146.2   90.6
52.2
row 4   8.7     88.7    84.8    174.6   164.4   106.2   114.0   151.8   130.8   101.9
116.2
row 5   42.7    130.5   180.4   199.4   196.6   99.7    163.6   34.2    5.5     146.1
108.5
row 6   120.0   159.5   26.0    83.4    58.7    10.1    170.1   20.5    10.8    48.3
121.9
row 7   148.9   148.4   142.0   106.6   198.4   60.3    72.1    137.8   74.5    75.7
44.8
row 8   25.7    192.0   12.1    23.4    98.7    145.3   196.8   43.9    143.1   25.1
122.6
row 9   5.4     134.7   165.1   61.8    46.7    183.3   173.7   146.9   76.5    186.2
24.9
row 10  174.5   158.9   127.8   58.9    42.9    182.9   7.8     50.3    68.0    62.0
66.1
row 11  47.3    166.2   8.2     71.2    98.5    12.4    179.0   100.2   29.7    167.4
155.2
row 12  23.9    196.6   148.7   7.1     128.2   128.8   66.3    153.7   60.7    115.4
71.6
```

```
row 13  103.4   184.3   161.5   57.9    199.2   79.3    28.1    73.1    12.5    71.3
100.4
row 14  130.3   154.2   127.5   29.7    198.2   170.3   121.9   80.4    159.8   70.0
82.6
row 15  26.7    45.6    67.7    109.7   5.1     96.2    188.7   100.7   48.3    164.2
75.4
row 16  115.4   25.5    58.8    148.5   80.7    149.1   156.7   153.8   42.0    103.7   4.2
row 17  67.9    161.5   16.9    102.1   77.3    3.9     104.7   97.2    181.8   182.0
155.1
row 18  169.5   122.4   102.2   5.5     14.5    105.1   181.5   83.3    117.6   52.1
111.2
row 19  47.1    146.9   21.0    8.6     130.3   24.7    95.7    6.7     159.9   38.8
82.6

Column Means
col_meansl[ 0] =    83.06   col_means2[ o] =    83.06
col_meansl[ 1] =    130.48  col_means2[ 1] =    130.48
col_meansl[ 2] =    88.47   col_means2[ 2] =    88.47
col_meansl[ 3] =    96.68   col_means2[ 3] =    96.68
col_meansl[ 4] =    105.92  col_means2[ 4] =    105.92
col_meansl[ 5] =    100.79  col_means2[ 5] =    100.79
col_meansl[ 6] =    121.66  col_means2[ 6] =    121.66
col_meansl[ 7] =    85.46   col_means2[ 7] =    85.46
col_meansl[ 8] =    83.75   col_means2[ 8] =    83.75
col_meansl[ 9] =    103.15  col_means2[ 9] =    103.15
col_meansl[10] =    89.77   col_means2[10] =    89.77
```

9.2.3. Коэффициент корреляции

В следующем примере исходного кода показано, как вычислить коэффициент корреляции с использованием арифметических действий над упакованными операндами с плавающей запятой двойной точности. Этот пример также демонстрирует, как выполнить несколько общих вспомогательных операций с упакованными операндами с плавающей запятой, включая 128-битное извлечение и горизонтальное сложение. В листинге 9.4 показан исходный код примера `Ch09_04`.

Листинг 9.4. Пример Ch09_04

```
//-----------------------------------------------
//               Ch09_04.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <string>
#include <random>
#include "AlignedMem.h"

using namespace std;

extern "C" bool AvxCalcCorrCoef_(const double* x, const double* y, size_t n, double
sums[5], double epsilon, double* rho);
```

```
void Init(double* x, double* y, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 999};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
    {
        x[i] = (double)ui_dist(rng);
        y[i] = x[i] + (ui_dist(rng) % 6000) - 3000;
    }
}

bool AvxCalcCorrCoefCpp(const double* x, const double* y, size_t n, double sums[5], double
epsilon, double* rho)
{
    const size_t alignment = 32;

    // Проверка n на допустимое значение
    if (n == 0)
        return false;

    // Проверка x и y на выравнивание
    if (!AlignedMem::IsAligned(x, alignment))
        return false;
    if (!AlignedMem::IsAligned(y, alignment))
        return false;

    // Вычисление и сохранение переменных sum
    double sum_x = 0, sum_y = 0, sum_xx = 0, sum_yy = 0, sum_xy = 0;

    for (size_t 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;

    // Вычисление 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 >= epsilon)
    {
        *rho = rho_num / rho_den;
        return true;
    }
    else
    {
```

```
        *rho = 0;
        return false;
    }
}

int main()
{
    const size_t n = 103;
    const size_t alignment = 32;
    AlignedArray<double> x_aa(n, alignment);
    AlignedArray<double> y_aa(n, alignment);
    double sums1[5], sums2[5];
    double rho1, rho2;
    double epsilon = 1.0e-12;
    double* x = x_aa.Data();
    double* y = y_aa.Data();

    Init(x, y, n, 71);

    bool rc1 = AvxCalcCorrCoefCpp(x, y, n, sums1, epsilon, &rho1);
    bool rc2 = AvxCalcCorrCoef_(x, y, n, sums2, epsilon, &rho2);

    cout << "Результаты AvxCalcCorrCoef\n\n";

    if (!rc1 || !rc2)
    {
        cout << "Возвращен код ошибки ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        return 1;
    }

    int w = 14;
    string sep(w * 3, '-');

    cout << fixed << setprecision(8);
    cout << "Value    " << setw(w) << "C++" << " " << setw(w) << "x86-AVX" << '\n';
    cout << sep << '\n';
    cout << "rho:     " << setw(w) << rho1 << " " << setw(w) << rho2 << "\n\n";

    cout << setprecision(1);
    cout << "sum_x:   " << setw(w) << sums1[0] << " " << setw(w) << sums2[0] << '\n';
    cout << "sum_y:   " << setw(w) << sums1[1] << " " << setw(w) << sums2[1] << '\n';
    cout << "sum_xx:  " << setw(w) << sums1[2] << " " << setw(w) << sums2[2] << '\n';
    cout << "sum_yy:  " << setw(w) << sums1[3] << " " << setw(w) << sums2[3] << '\n';
    cout << "sum_xy:  " << setw(w) << sums1[4] << " " << setw(w) << sums2[4] << '\n';
    return 0;
}

;-----------------------------------------------
;              Ch09_04.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; extern "C" bool AvxCalcCorrCoef_(const double* x, const double* y, size_t n, double
sums[5], double epsilon, double* rho)
```

```
;
; Возвращает       0 = ошибка, 1 = успех

        .code
AvxCalcCorrCoef_ proc frame
        _CreateFrame CC_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Проверка аргументов
        or r8,r8
        jz BadArg                       ;переход, если n == 0
        test rcx,1fh
        jnz BadArg                      ;переход, если x не выровнен
        test rdx,1fh
        jnz BadArg                      ;переход, если y не выровнен

; Инициализация переменных нулями
        vxorpd ymm3,ymm3,ymm3           ;ymm3 = упакованный sum_x
        vxorpd ymm4,ymm4,ymm4           ;ymm4 = упакованный sum_y
        vxorpd ymm5,ymm5,ymm5           ;ymm5 = упакованный sum_xx
        vxorpd ymm6,ymm6,ymm6           ;ymm6 = упакованный sum_yy
        vxorpd ymm7,ymm7,ymm7           ;ymm7 = упакованный sum_xy
        mov r10,r8                      ;r10 = n

        cmp r8,4
        jb LP2                          ;переход, если n >= 1 && n <= 3

; Вычисление промежуточных упакованных переменных суммы
LP1:    vmovapd ymm0,ymmword ptr [rcx]  ;ymm0 = упакованные значения x
        vmovapd ymm1,ymmword ptr [rdx]  ;ymm1 = упакованные значения y

        vaddpd ymm3,ymm3,ymm0           ;обновление упакованного sum_x
        vaddpd ymm4,ymm4,ymm1           ;обновление упакованного sum_y

        vmulpd ymm2,ymm0,ymm1           ;ymm2 = упакованные значения xy
        vaddpd ymm7,ymm7,ymm2           ;обновление упакованного sum_xy

        vmulpd ymm0,ymm0,ymm0           ;ymm0 = упакованные значения xx
        vmulpd ymm1,ymm1,ymm1           ;ymm1 = упакованные значения yy
        vaddpd ymm5,ymm5,ymm0           ;обновление упакованного sum_xx
        vaddpd ymm6,ymm6,ymm1           ;обновление упакованного sum_yy

        add rcx,32                      ;обновление указателя x
        add rdx,32                      ;обновление указателя y
        sub r8,4                        ;n -= 4
        cmp r8,4                        ;проверка n >= 4?
        jae LP1                         ;переход, если да

        or r8,r8                        ;проверка n == 0?
        jz FSV                          ;переход, если да

; Обновление переменных суммы финальными значениями x и y
LP2:    vmovsd xmm0,real8 ptr [rcx]     ;xmm0[63:0] = x[i], ymm0[255:64] = 0
        vmovsd xmm1,real8 ptr [rdx]     ;xmm1[63:0] = y[i], ymm1[255:64] = 0
```

```
        vaddpd ymm3,ymm3,ymm0               ;обновление упакованного sum_x
        vaddpd ymm4,ymm4,ymm1               ;обновление упакованного sum_y

        vmulpd ymm2,ymm0,ymm1               ;ymm2 = упакованные значения xy
        vaddpd ymm7,ymm7,ymm2               ;обновление упакованного sum_xy

        vmulpd ymm0,ymm0,ymm0               ;ymm0 = упакованные значения xx
        vmulpd ymm1,ymm1,ymm1               ;ymm1 = упакованные значения yy
        vaddpd ymm5,ymm5,ymm0               ;обновление упакованного sum_xx
        vaddpd ymm6,ymm6,ymm1               ;обновление упакованного sum_yy

        add rcx,8                           ;обновление указателя x
        add rdx,8                           ;обновление указателя y
        sub r8,1                            ;n -= 1
        jnz LP2                             ;повтор до завершения

; Вычисление финальных значений сумм
FSV:    vextractf128 xmm0,ymm3,1
        vaddpd xmm1,xmm0,xmm3
        vhaddpd xmm3,xmm1,xmm1              ;xmm3[63:0] = sum_x

        vextractf128 xmm0,ymm4,1
        vaddpd xmm1,xmm0,xmm4
        vhaddpd xmm4,xmm1,xmm1              ;xmm4[63:0] = sum_y

        vextractf128 xmm0,ymm5,1
        vaddpd xmm1,xmm0,xmm5
        vhaddpd xmm5,xmm1,xmm1              ;xmm5[63:0] = sum_xx

        vextractf128 xmm0,ymm6,1
        vaddpd xmm1,xmm0,xmm6
        vhaddpd xmm6,xmm1,xmm1              ;xmm6[63:0] = sum_yy

        vextractf128 xmm0,ymm7,1
        vaddpd xmm1,xmm0,xmm7
        vhaddpd xmm7,xmm1,xmm1              ;xmm7[63:0] = sum_xy

; Сохранение финальных значений сумм
        vmovsd real8 ptr [r9],xmm3          ;сохранение sum_x
        vmovsd real8 ptr [r9+8],xmm4        ;сохранение sum_y
        vmovsd real8 ptr [r9+16],xmm5       ;сохранение sum_xx
        vmovsd real8 ptr [r9+24],xmm6       ;сохранение sum_yy
        vmovsd real8 ptr [r9+32],xmm7       ;сохранение sum_xy

; Вычисление числителя rho
; rho_num = n * sum_xy - sum_x * sum_y;
        vcvtsi2sd xmm2,xmm2,r10             ;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

; Вычисление знаменателя rho
; 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

        vmulsd xmm0,xmm3,xmm4       ;xmm0 = rho_den

; Вычисление и сохранение финального значения rho
        xor eax,eax
        vcomisd xmm0,real8 ptr [rbp+CC_OffsetStackArgs] ;rho_den < epsilon?
        setae al                                ;возвращаемый код
        jb BadRho                               ;переход, если rho_den < epsilon
        vdivsd xmm1,xmm7,xmm0                   ;xmm1 = rho

SavRho: mov rdx,[rbp+CC_OffsetStackArgs+8]      ;rdx = указатель на rho
        vmovsd real8 ptr [rdx],xmm1             ;сохранение rho

Done:   vzeroupper
        _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret

; Обработка кода ошибки
BadRho: vxorpd xmm1,xmm1,xmm1               ;rho = 0
        jmp SavRho

BadArg: xor eax,eax                         ;eax = код ошибки некорректного аргумента
        jmp Done

AvxCalcCorrCoef_ endp
        end
```

Коэффициент корреляции измеряет силу связи между двумя переменными. Коэффициенты корреляции могут иметь значение от –1,0 до +1,0, что означает идеальную отрицательную или положительную связь между двумя переменными. Коэффициенты корреляции в реальном мире редко бывают равными этим теоретическим пределам. Коэффициент корреляции 0,0 указывает, что переменные не связаны. Код C++ и ассемблера в этом примере вычисляет широко известный *коэффициент корреляции Пирсона*, используя следующее уравнение:

$$\rho = \frac{n\sum_i x_i y_i - \sum_i x_i \sum_i y_i}{\sqrt{n\sum_i x_i^2 - \left(\sum_i x_i\right)^2}\sqrt{n\sum_i y_i^2 - \left(\sum_i y_i\right)^2}}.$$

Чтобы вычислить коэффициент корреляции по этой формуле, функция должна вычислить значения следующих пяти переменных суммы:

$$sum\_x = \sum_i x_i;$$
$$sum\_y = \sum_i y_i;$$
$$sum\_xx = \sum_i x_i^2;$$
$$sum\_yy = \sum_i y_i^2;$$
$$sum\_xy = \sum_i x_i y_i.$$

Функция C++ `AvxCalcCorrCoefCpp` демонстрирует расчет коэффициента корреляции. Эта функция начинается с проверки значения `n`, чтобы убедиться, что оно больше нуля. Она также проверяет правильность выравнивания двух массивов данных `x` и `y`. Вышеупомянутые переменные суммы затем вычисляются с использованием простого цикла `for`. После завершения цикла `for` функция `AvxCalcCorrCoefCpp` сохраняет переменные сумм в массиве для последующего сравнения и отображения. Затем она вычисляет промежуточные значения `rho_num` и `rho_den`. Перед вычислением окончательного значения коэффициента корреляции `rho` выполняется проверка, что `rho_den` действительно больше или равно `epsilon`.

Следом за прологом функция на языке ассемблера `AvxCalcCorrCoef_` выполняет те же проверки размера и выравнивания, что и ее аналог в C++. Затем она инициализирует упакованные версии `sum_x`, `sum_y`, `sum_xx`, `sum_yy` и `sum_xy` до нуля в регистрах YMM3–YMM7. Во время каждой итерации цикл `LP1` обрабатывает четыре элемента из массивов `x` и `y`, используя арифметику упакованных чисел с плавающей запятой двойной точности. Это означает, что регистры YMM3–YMM7 хранят четыре различных промежуточных значения для каждой переменной суммы. Выполнение цикла `LP1` продолжается до тех пор, пока для обработки не останется менее четырех элементов.

После завершения цикла `LP1` цикл `LP2` обрабатывает оставшиеся (1–3) записи в массивах `x` и `y`. Инструкции `vmovsd xmm0,real8 ptr [rcx]` и `vmovsd xmm1,real8 ptr [rdx]` загружают `x[i]` и `y[i]` в регистры XMM0 и XMM1 соответственно. Обратите внимание, что эти команды `vmovsd` также обнуляют биты YMM0 [255:64] и YMM1 [255:64], а это означает, что та же цепочка команд `vaddpd` и `vmulpd`, которая испольльзовалась в цикле `LP1` для обновления переменных промежуточной суммы, может использоваться и в цикле `LP2` (скалярные команды `vaddsd` и `vmulsd` не могут использоваться здесь для обновления переменных суммы без дополнительного кода, поскольку эти команды устанавливают биты 255:128 своего регистра операнда назначения в ноль). После завершения цикла `LP2` каждая переменная упакованной суммы сокращается до одного значения с помощью команд `vextractf128`, `vaddpd` и `vhaddpd`, как показано на рис. 9.3. Окончательные значения суммы затем сохраняются в массив сумм.

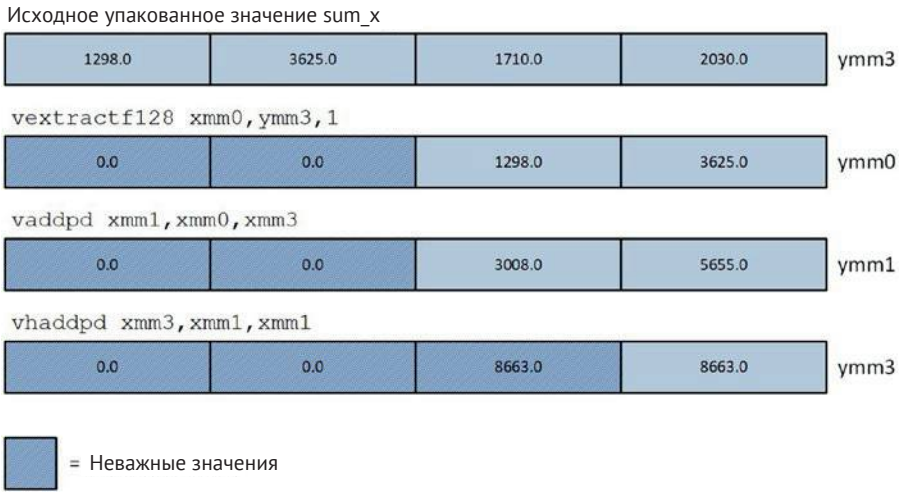


Рис. 9.3. Расчет `sum_x` с использованием команд `vextractf128`, `vaddpd` и `vhaddpd`

Функция `AvxCalcCorrCoef_` использует простую скалярную арифметику для вычисления промежуточных значений `rho_num` и `rho_den`. Как и соответствующая функция C++, `AvxCalcCorrCoef_` проверяет `rho_den`, чтобы узнать, меньше ли это значение, чем `epsilon` (значение ниже `epsilon`, вероятно, является ошибкой округления и считается слишком близким к нулю, чтобы считаться действительным). Если `rho_den` действителен и достоин внимания, вычисляется и сохраняется коэффициент корреляции `rho`. Вот результаты выполнения примера исходного кода `Ch09_04`:

```
Результаты AvxCalcCorrCoef
Value               C++         x86-AVX
-----------------------------------------
rho:         0.70128193     0.70128193
sum_x:          53081.0        53081.0
sum_y:        -199158.0      -199158.0
sum_xx:      35732585.0     35732585.0
sum_yy:     401708868.0    401708868.0
sum_xy:     -94360528.0    -94360528.0
```

9.3. Умножение и транспонирование матриц

В главе 6 вы узнали, как выполнять транспонирование и умножение матрицы 4×4 с использованием значений с плавающей запятой одинарной точности (см. примеры исходного кода `Ch06_07` и `Ch06_08`). Пример исходного кода в этом разделе показывает, как выполнять те же операции с матрицами, используя значения с плавающей запятой двойной точности. В листинге 9.5 показан исходный код примера `Ch09_05`. Основы транспонирования и умножения матриц объясняются в главе 6. Если вам не хватает понимания этих математических операций, вы можете просмотреть соответствующие разделы в главе 6, прежде чем продолжить.

Листинг 9.5. Пример Ch09_05

```
//-----------------------------------------------
//               Ch09_05.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include "Ch09_05.h"
#include "Matrix.h"

using namespace std;

void AvxMat4x4TransposeF64(Matrix<double>& m_src1)
{
    const size_t nr = 4;
    const size_t nc = 4;
    Matrix<double> m_des1(nr ,nc);
    Matrix<double> m_des2(nr ,nc);

    Matrix<double>::Transpose(m_des1, m_src1);
    AvxMat4x4TransposeF64_(m_des2.Data(), m_src1.Data());

    cout << fixed << setprecision(1);
    m_src1.SetOstream(12, "  ");
    m_des1.SetOstream(12, "  ");
    m_des2.SetOstream(12, "  ");

    cout << "Результаты AvxMat4x4TransposeF64\n";
    cout << "Матрица m_src1\n" << m_src1 << '\n';
    cout << "Матрица m_des1\n" << m_des1 << '\n';
    cout << "Матрица m_des2\n" << m_des2 << '\n';

    if (m_des1 != m_des2)
        cout << "\nНесовпадение матриц - AvxMat4x4TransposeF64\n";
}

void AvxMat4x4MulF64(Matrix<double>& m_src1, Matrix<double>& m_src2)
{
    const size_t nr = 4;
    const size_t nc = 4;
    Matrix<double> m_des1(nr ,nc);
    Matrix<double> m_des2(nr ,nc);

    Matrix<double>::Mul(m_des1, m_src1, m_src2);
    AvxMat4x4MulF64_(m_des2.Data(), m_src1.Data(), m_src2.Data());

    cout << fixed << setprecision(1);

    m_src1.SetOstream(12, "  ");
    m_src2.SetOstream(12, "  ");
    m_des1.SetOstream(12, "  ");
    m_des2.SetOstream(12, "  ");

    cout << "\nРезультаты AvxMat4x4MulF64\n";
    cout << "Матрица m_src1\n" << m_src1 << '\n';
```

```
    cout << "Матрица m_src2\n" << m_src2 << '\n';
    cout << "Матрица m_des1\n" << m_des1 << '\n';
    cout << "Матрица m_des2\n" << m_des2 << '\n';

    if (m_des1 != m_des2)
        cout << "\nНесовпадение матриц - AvxMat4x4MulF64\n";
}

int main()
{
    const size_t nr = 4;
    const size_t nc = 4;
    Matrix<double> m_src1(nr ,nc);
    Matrix<double> m_src2(nr ,nc);

    const double src1_row0[] = { 10, 11, 12, 13 };
    const double src1_row1[] = { 20, 21, 22, 23 };
    const double src1_row2[] = { 30, 31, 32, 33 };
    const double src1_row3[] = { 40, 41, 42, 43 };

    const double src2_row0[] = { 100, 101, 102, 103 };
    const double src2_row1[] = { 200, 201, 202, 203 };
    const double src2_row2[] = { 300, 301, 302, 303 };
    const double src2_row3[] = { 400, 401, 402, 403 };

    m_src1.SetRow(0, src1_row0);
    m_src1.SetRow(1, src1_row1);
    m_src1.SetRow(2, src1_row2);
    m_src1.SetRow(3, src1_row3);

    m_src2.SetRow(0, src2_row0);
    m_src2.SetRow(1, src2_row1);
    m_src2.SetRow(2, src2_row2);
    m_src2.SetRow(3, src2_row3);

    // Функции тестирования
    AvxMat4x4TransposeF64(m_src1);
    AvxMat4x4MulF64(m_src1, m_src2);

    // Функции измерения быстродействия
    AvxMat4x4TransposeF64_BM();
    AvxMat4x4MulF64_BM();
    return 0;
}

;-----------------------------------------------
;              Ch09_05.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; _Mat4x4TransposeF64 macro
;
; Описание:  этот макрос траспонирует матрицу 4×4
;            значений двойной точности с плавающей запятой
;
```

```
;  Исходная матрица                Результат
;  ------------------------------------------------
;  ymm0    a3 a2 a1 a0             ymm0    d0 c0 b0 a0
;  ymm1    b3 b2 b1 b0             ymm1    d1 c1 b1 a1
;  ymm2    c3 c2 c1 c0             ymm2    d2 c2 b2 a2
;  ymm3    d3 d2 d1 d0             ymm3    d3 c3 b3 a3
;

_Mat4x4TransposeF64 macro
        vunpcklpd ymm4,ymm0,ymm1            ;ymm4 = b2 a2 b0 a0
        vunpckhpd ymm5,ymm0,ymm1            ;ymm5 = b3 a3 b1 a1
        vunpcklpd ymm6,ymm2,ymm3            ;ymm6 = d2 c2 d0 c0
        vunpckhpd ymm7,ymm2,ymm3            ;ymm7 = d3 c3 d1 c1

        vperm2f128 ymm0,ymm4,ymm6,20h       ;ymm0 = d0 c0 b0 a0
        vperm2f128 ymm1,ymm5,ymm7,20h       ;ymm1 = d1 c1 b1 a1
        vperm2f128 ymm2,ymm4,ymm6,31h       ;ymm2 = d2 c2 b2 a2
        vperm2f128 ymm3,ymm5,ymm7,31h       ;ymm3 = d3 c3 b3 a3
        endm

; extern "C" void AvxMat4x4TransposeF64_(double* m_des, const double* m_src1)

        .code
AvxMat4x4TransposeF64_ proc frame
        _CreateFrame MT_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Транспонирование матрицы m_src1
        vmovaps ymm0,[rdx]                  ;ymm0 = m_src1.row_0
        vmovaps ymm1,[rdx+32]               ;ymm1 = m_src2.row_1
        vmovaps ymm2,[rdx+64]               ;ymm2 = m_src3.row_2
        vmovaps ymm3,[rdx+96]               ;ymm3 = m_src4.row_3

        _Mat4x4TransposeF64

        vmovaps [rcx],ymm0                  ;сохранение m_des.row_0
        vmovaps [rcx+32],ymm1               ;сохранение m_des.row_1
        vmovaps [rcx+64],ymm2               ;сохранение m_des.row_2
        vmovaps [rcx+96],ymm3               ;сохранение m_des.row_3

        vzeroupper
Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
AvxMat4x4TransposeF64_ endp

; _Mat4x4MulCalcRowF64 macro
;
; Описание:  этот макрос вычисляет одну строку произведения матриц 4×4
;
; Регистры:    ymm0 = m_src2.row0
;               ymm1 = m_src2.row1
;               ymm2 = m_src2.row2
;               ymm3 = m_src2.row3
;               rcx = m_des ptr
```

```
;               rdx = m_src1 ptr
;               ymm4 - ymm4 = начальные регистры

_Mat4x4MulCalcRowF64 macro disp
        vbroadcastsd ymm4,real8 ptr [rdx+disp]      ;передача m_src1[i][0]
        vbroadcastsd ymm5,real8 ptr [rdx+disp+8]    ;передача m_src1[i][1]
        vbroadcastsd ymm6,real8 ptr [rdx+disp+16]   ;передача m_src1[i][2]
        vbroadcastsd ymm7,real8 ptr [rdx+disp+24]   ;передача m_src1[i][3]

        vmulpd ymm4,ymm4,ymm0                       ;m_src1[i][0] * m_src2.row_0
        vmulpd ymm5,ymm5,ymm1                       ;m_src1[i][1] * m_src2.row_1
        vmulpd ymm6,ymm6,ymm2                       ;m_src1[i][2] * m_src2.row_2
        vmulpd ymm7,ymm7,ymm3                       ;m_src1[i][3] * m_src2.row_3

        vaddpd ymm4,ymm4,ymm5                       ;вычисление m_des.row_i
        vaddpd ymm6,ymm6,ymm7
        vaddpd ymm4,ymm4,ymm6

        vmovapd [rcx+disp],ymm4                     ;сохранение m_des.row_i
        endm

; extern "C" void AvxMat4x4MulF64_(double* m_des, const double* m_src1, const double* m_
src2)

AvxMat4x4MulF64_ proc frame
        _CreateFrame MM_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

; Загрузка m_src2 в YMM3:YMM0
        vmovapd ymm0,[r8]                   ;ymm0 = m_src2.row_0
        vmovapd ymm1,[r8+32]                ;ymm1 = m_src2.row_1
        vmovapd ymm2,[r8+64]                ;ymm2 = m_src2.row_2
        vmovapd ymm3,[r8+96]                ;ymm3 = m_src2.row_3

; Вычисление произведения матриц
        _Mat4x4MulCalcRowF64 0              ;calculate m_des.row_0
        _Mat4x4MulCalcRowF64 32             ;calculate m_des.row_1
        _Mat4x4MulCalcRowF64 64             ;calculate m_des.row_2
        _Mat4x4MulCalcRowF64 96             ;calculate m_des.row_3

        vzeroupper
Done:   _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
AvxMat4x4MulF64_ endp
        end
```

Исходный код C++, показанный в листинге 9.5, очень похож на тот, что вы видели в главе 6. Он начинается с функции `AvxMat4x4TransposeF64`, которая выполняет процедуры транспонирования матриц как на C++, так и на языке ассемблера и отображает результаты. Следующая функция `AvxMat4x4MulF64` реализует те же задачи для умножения матриц. Подобно примерам исходного кода в главе 6, версии C++ транспонирования и умножения матриц реализуются шаблонными функциями.

В верхней части кода языка ассемблера находится макрос с именем _Mat4x4TransposeF64. Этот макрос содержит команды, которые транспонируют матрицу 4×4, составленную из значений двойной точности с плавающей запятой. Четыре строки исходной матрицы с плавающей запятой двойной точности должны быть загружены в регистры YMM0–YMM3 перед ее использованием. Макрос _Mat4x4TransposeF64 использует команду vperm2f128 для перестановки 128-битных полей с плавающей запятой двух своих исходных операндов. Эта команда применяет непосредственную 8-битную маску управления, чтобы выбрать, какие поля копируются из операндов источника в операнд назначения, как показано в табл. 9.1. На рис. 9.4 более подробно показана процедура транспонирования матрицы 4×4. Функция на языке ассемблера AvxMat4x4TransposeF64_ использует макрос _Mat4x4TransposeF64 для транспонирования матрицы 4×4, состоящей из значений двойной точности с плавающей запятой.

Таблица 9.1. Выбор поля для команды `vperm2f128 ymm0,ymm1,ymm2,imm8`

Приемник	Источник	imm8[1:0]	imm8[4:3]
ymm0[127:0]	ymm1[127:0]	0	
	ymm1[255:128]	1	
	ymm2[127:0]	2	
	ymm2[255:128]	3	
ymm0[255:128]	ymm1[127:0]		0
	ymm1[255:128]		1
	ymm2[127:0]		2
	ymm2[255:128]		3

В листинге 9.5 за функцией AvxMat4x4TransposeF64_ следует определение макроса _Mat4x4MulCalcRowF64. Этот макрос содержит команды, которые вычисляют одну строку произведения матриц 4×4. Используемая здесь техника перемножения строк идентична той, которая применялась в примере исходного кода Ch06_08 в главе 6 (рис. 6.7). Функция AvxMat4x4MulF64_ использует макрос _Mat4x4MulCalcRowF64 для перемножения двух матриц значений двойной точности с плавающей запятой 4×4. Вот результаты выполнения примера исходного кода Ch09_05:

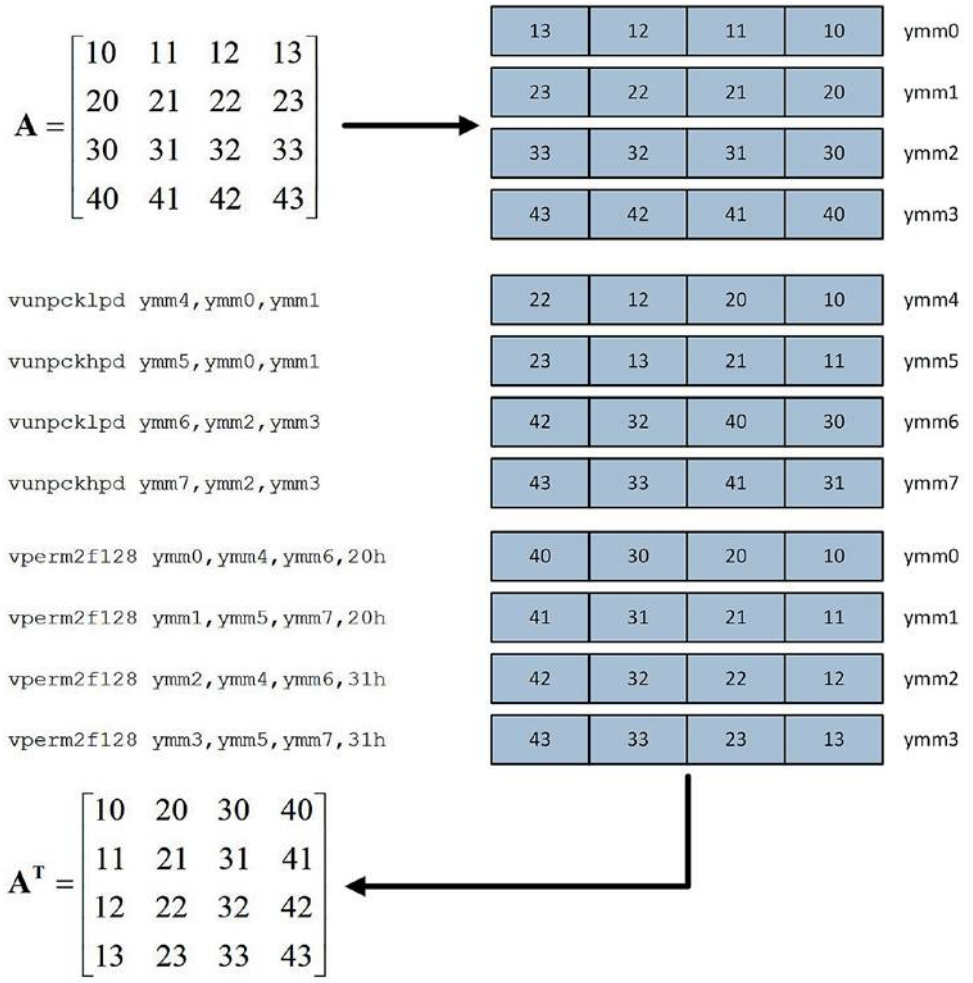


Рис. 9.4. Последовательность команд функции _Max4x4TransposeF64 для транспонирования матрицы 4×4, состоящей из значений двойной точности с плавающей запятой

```
Результаты AvxMat4x4TransposeF64

Матрица m_src1
        10.0        11.0        12.0        13.0
        20.0        21.0        22.0        23.0
        30.0        31.0        32.0        33.0
        40.0        41.0        42.0        43.0

Матрица m_des1
        10.0        20.0        30.0        40.0
        11.0        21.0        31.0        41.0
        12.0        22.0        32.0        42.0
        13.0        23.0        33.0        43.0
```

```
Матрица m_des2
        10.0        20.0        30.0        40.0
        11.0        21.0        31.0        41.0
        12.0        22.0        32.0        42.0
        13.0        23.0        33.0        43.0

Результаты AvxMat4x4MulF64

Матрица m_src1
        10.0        11.0        12.0        13.0
        20.0        21.0        22.0        23.0
        30.0        31.0        32.0        33.0
        40.0        41.0        42.0        43.0

Матрица m_src2
       100.0       101.0       102.0       103.0
       200.0       201.0       202.0       203.0
       300.0       301.0       302.0       303.0
       400.0       401.0       402.0       403.0

Матрица m_des1
     12000.0     12046.0     12092.0     12138.0
     22000.0     22086.0     22172.0     22258.0
     32000.0     32126.0     32252.0     32378.0
     42000.0     42166.0     42332.0     42498.0

Матрица m_des2
     12000.0     12046.0     12092.0     12138.0
     22000.0     22086.0     22172.0     22258.0
     32000.0     32126.0     32252.0     32378.0
     42000.0     42166.0     42332.0     42498.0

Выполняется измерение быстродействия AvxMat4x4TransposeF64_BM - подождите
Результаты сохранены в файл Ch09_05_AvxMat4x4TransposeF64_BM_CHROMIUM.csv

Выполняется измерение быстродействия AvxMat4x4MulF64_BM - подождите
Результаты сохранены в файл Ch09_05_AvxMat4x4MulF64_BM_CHROMIUM.csv
```

Таблицы 9.2 и 9.3 содержат результаты измерения быстродействия функций транспонирования и умножения матриц, представленных в этом разделе. Эти измерения были выполнены с использованием процедуры, описанной в главе 6.

Таблица 9.2. Среднее время выполнения транспонирования матрицы (мс), 1 000 000 транспонирований

Процессор	C++	Ассемблер
i7-4790S	15562	2670
i9-7900X	13167	2112
i7-8700K	12194	1963

Таблица 9.3. Среднее время выполнения перемножения матриц (мс), 1 000 000 перемножений

Процессор	C++	Ассемблер
i7-4790S	55652	5874
i9-7900X	46910	5286
i7-8700K	43118	4505

9.4. Обращение матриц

Помимо транспонирования и умножения, еще одна распространенная операция, которая часто применяется к матрицам 4×4, – это *обращение* (инверсия) матриц. В этом разделе вы исследуете программу, которая вычисляет обратную матрицу 4×4 значений двойной точности с плавающей запятой. В листинге 9.6 показан исходный код примера `Ch09_06`.

Листинг 9.6. Пример Ch09_06

```
//-----------------------------------------------
//              Ch09_06.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <cmath>
#include "Ch09_06.h"
#include "Matrix.h"

using namespace std;

bool Avx2Mat4x4InvF64Cpp(Matrix<double>& m_inv, const Matrix<double>& m, double epsilon,
bool* is_singular)
{
    // Промежуточные матрицы, представленные ниже, объявлены статическими
    // для нужд измерения быстродействия
    static const size_t nrows = 4;
    static const size_t ncols = 4;
    static Matrix<double> m2(nrows, ncols);
    static Matrix<double> m3(nrows, ncols);
    static Matrix<double> m4(nrows, ncols);
    static Matrix<double> I(nrows, ncols, true);
    static Matrix<double> tempA(nrows, ncols);
    static Matrix<double> tempB(nrows, ncols);
    static Matrix<double> tempC(nrows, ncols);
    static Matrix<double> tempD(nrows, ncols);

    Matrix<double>::Mul(m2, m, m);
    Matrix<double>::Mul(m3, m2, m);
    Matrix<double>::Mul(m4, m3, m);

    double t1 = m.Trace();
    double t2 = m2.Trace();
    double t3 = m3.Trace();
    double t4 = m4.Trace();
```

```
    double c1 = -t1;
    double c2 = -1.0 / 2.0 * (c1 * t1 + t2);
    double c3 = -1.0 / 3.0 * (c2 * t1 + c1 * t2 + t3);
    double c4 = -1.0 / 4.0 * (c3 * t1 + c2 * t2 + c1 * t3 + t4);

    // Проверка, что матрица несингулярная
    *is_singular = (fabs(c4) < epsilon);

    if (*is_singular)
        return false;

    // Вычисление = -1.0 / c4 * (m3 + c1 * m2 + c2 * m + c3 * I)
    Matrix<double>::MulScalar(tempA, I, c3);
    Matrix<double>::MulScalar(tempB, m, c2);
    Matrix<double>::MulScalar(tempC, m2, c1);
    Matrix<double>::Add(tempD, tempA, tempB);
    Matrix<double>::Add(tempD, tempD, tempC);
    Matrix<double>::Add(tempD, tempD, m3);
    Matrix<double>::MulScalar(m_inv, tempD, -1.0 / c4);

    return true;
}

void Avx2Mat4x4InvF64(const Matrix<double>& m, const char* msg)
{
    cout << '\n' << msg << " - Test Matrix\n";
    cout << m << '\n';

    const double epsilon = 1.0e-9;
    const size_t nrows = m.GetNumRows();
    const size_t ncols = m.GetNumCols();
    Matrix<double> m_inv_a(nrows, ncols);
    Matrix<double> m_ver_a(nrows, ncols);
    Matrix<double> m_inv_b(nrows, ncols);
    Matrix<double> m_ver_b(nrows, ncols);

    for (int i = 0; i <= 1; i++)
    {
        string fn;
        const size_t nrows = m.GetNumRows();
        const size_t ncols = m.GetNumCols();
        Matrix<double> m_inv(nrows, ncols);
        Matrix<double> m_ver(nrows, ncols);
        bool rc, is_singular;

        if (i == 0)
        {
            fn = "Avx2Mat4x4InvF64Cpp";
            rc = Avx2Mat4x4InvF64Cpp(m_inv, m, epsilon, &is_singular);

            if (rc)
                Matrix<double>::Mul(m_ver, m_inv, m);
        }
        else
        {
            fn = "Avx2Mat4x4InvF64_";
```

```
            rc = Avx2Mat4x4InvF64_(m_inv.Data(), m.Data(), epsilon, &is_singular);

            if (rc)
                Avx2Mat4x4MulF64_(m_ver.Data(), m_inv.Data(), m.Data());
        }

        if (rc)
        {
            cout << msg << " - " << fn << " - Обращенная матрица\n";
            cout << m_inv << '\n';

            // Округление до нуля в целях отображения, эту процедуру можно удалить.
            cout << msg << " - " << fn << " - проверка матрицы\n";
            m_ver.RoundToZero(epsilon);
            cout << m_ver << '\n';
        }
        else
        {
            if (is_singular)
                cout << msg << " - " << fn << " - сингулярная матрица\n";
            else
                cout << msg << " - " << fn << " - обнаружена ошибка\n";
        }
    }
}

int main()
{
    cout << "\nРезультаты Avx2Mat4x4InvF64\n";

    // Тестовая матрица #1 - несингулярная
    Matrix<double> m1(4, 4);
    const double m1_row0[] = { 2, 7, 3, 4 };
    const double m1_row1[] = { 5, 9, 6, 4.75 };
    const double m1_row2[] = { 6.5, 3, 4, 10 };
    const double m1_row3[] = { 7, 5.25, 8.125, 6 };
    m1.SetRow(0, m1_row0);
    m1.SetRow(1, m1_row1);
    m1.SetRow(2, m1_row2);
    m1.SetRow(3, m1_row3);

    // Тестовая матрица #2 - несингулярная
    Matrix<double> m2(4, 4);
    const double m2_row0[] = { 0.5, 12, 17.25, 4 };
    const double m2_row1[] = { 5, 2, 6.75, 8 };
    const double m2_row2[] = { 13.125, 1, 3, 9.75 };
    const double m2_row3[] = { 16, 1.625, 7, 0.25 };
    m2.SetRow(0, m2_row0);
    m2.SetRow(1, m2_row1);
    m2.SetRow(2, m2_row2);
    m2.SetRow(3, m2_row3);

    // Тестовая матрица #3 - сингулярная
    Matrix<double> m3(4, 4);
    const double m3_row0[] = { 2, 0, 0, 1 };
    const double m3_row1[] = { 0, 4, 5, 0 };
```

```
    const double m3_row2[] = { 0, 0, 0, 7 };
    const double m3_row3[] = { 0, 0, 0, 6 };
    m3.SetRow(0, m3_row0);
    m3.SetRow(1, m3_row1);
    m3.SetRow(2, m3_row2);
    m3.SetRow(3, m3_row3);

    Avx2Mat4x4InvF64(m1, "Тест #1");
    Avx2Mat4x4InvF64(m2, "Тест #2");
    Avx2Mat4x4InvF64(m3, "Тест #3");

    Avx2Mat4x4InvF64_BM(m1);
    return 0;
}

;-----------------------------------------------
;              Ch09_06.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; Настраиваемый сегмент для констант
ConstVals segment readonly align(32) 'const'
Mat4x4I real8 1.0, 0.0, 0.0, 0.0
        real8 0.0, 1.0, 0.0, 0.0
        real8 0.0, 0.0, 1.0, 0.0
        real8 0.0, 0.0, 0.0, 1.0

r8_SignBitMask  qword 4 dup (8000000000000000h)
r8_AbsMask      qword 4 dup (7fffffffffffffffh)

r8_1p0          real8 1.0
r8_N1p0         real8 -1.0
r8_N0p5         real8 -0.5
r8_N0p3333      real8 -0.33333333333333
r8_N0p25        real8 -0.25
ConstVals ends
        .code

; _Mat4x4TraceF64 macro
;
; Описание:  этот макрос содержит команды, вычисляющие след матрицы 4×4
; значений двойной точности с плавающей запятой в ymm3:ymm0

_Max4x4TraceF64 macro
        vblendpd ymm0,ymm0,ymm1,00000010b       ;ymm0[127:0] = диаг.знач строки 1,0
        vblendpd ymm1,ymm2,ymm3,00001000b       ;ymm1[255:128] = диаг.знач строки 3,2
        vperm2f128 ymm2,ymm1,ymm1,00000001b     ;ymm2[127:0] = диаг.знач строки 3,2
        vaddpd ymm3,ymm0,ymm2
        vhaddpd ymm0,ymm3,ymm3                  ;xmm0[63:0] = след
        endm

; extern "C" double Avx2Mat4x4TraceF64_(const double* m_src1)
;
; Описание:  следующая функция вычисляет след матрицы 4×4
;            значений двойной точности с плавающей запятой
```

```
Avx2Mat4x4TraceF64_ proc
            vmovapd ymm0,[rcx]              ;ymm0 = m_src1.row_0
            vmovapd ymm1,[rcx+32]           ;ymm1 = m_src1.row_1
            vmovapd ymm2,[rcx+64]           ;ymm2 = m_src1.row_2
            vmovapd ymm3,[rcx+96]           ;ymm3 = m_src1.row_3

            _Max4x4TraceF64                 ;xmm0[63:0] = m_src1.trace()
            vzeroupper
            ret
Avx2Mat4x4TraceF64_ endp

; _Mat4x4MulCalcRowF64 macro
;
; Описание:  этот макрос применяется для вычисления одной строки
;            произведения матриц 4×4
;
; Регистры:    ymm0 = m_src2.row0
;               ymm1 = m_src2.row1
;               ymm2 = m_src2.row2
;               ymm3 = m_src2.row3
;               ymm4 - ymm7 = рабочие регистры

_Mat4x4MulCalcRowF64 macro dreg,sreg,disp
        vbroadcastsd ymm4,real8 ptr [sreg+disp]     ;передача m_src1[i][0]
        vbroadcastsd ymm5,real8 ptr [sreg+disp+8]   ;передача m_src1[i][1]
        vbroadcastsd ymm6,real8 ptr [sreg+disp+16]  ;передача m_src1[i][2]
        vbroadcastsd ymm7,real8 ptr [sreg+disp+24]  ;передача m_src1[i][3]

        vmulpd ymm4,ymm4,ymm0                       ;m_src1[i][0] * m_src2.row_0
        vmulpd ymm5,ymm5,ymm1                       ;m_src1[i][1] * m_src2.row_1
        vmulpd ymm6,ymm6,ymm2                       ;m_src1[i][2] * m_src2.row_2
        vmulpd ymm7,ymm7,ymm3                       ;m_src1[i][3] * m_src2.row_3

        vaddpd ymm4,ymm4,ymm5                       ;вычисление m_des.row_i
        vaddpd ymm6,ymm6,ymm7
        vaddpd ymm4,ymm4,ymm6
        vmovapd[dreg+disp],ymm4                     ;сохранение m_des.row_i
        endm

; extern "C" void Avx2Mat4x4MulF64_(double* m_des, const double* m_src1, const double*
m_src2)

Avx2Mat4x4MulF64_ proc frame
        _CreateFrame MM_,0,32
        _SaveXmmRegs xmm6,xmm7
        _EndProlog

        vmovapd ymm0,[r8]                   ;ymm0 = m_src2.row_0
        vmovapd ymm1,[r8+32]                ;ymm1 = m_src2.row_1
        vmovapd ymm2,[r8+64]                ;ymm2 = m_src2.row_2
        vmovapd ymm3,[r8+96]                ;ymm3 = m_src2.row_3

        _Mat4x4MulCalcRowF64 rcx,rdx,0      ;вычисление m_des.row_0
        _Mat4x4MulCalcRowF64 rcx,rdx,32     ;вычисление m_des.row_1
        _Mat4x4MulCalcRowF64 rcx,rdx,64     ;вычисление m_des.row_2
        _Mat4x4MulCalcRowF64 rcx,rdx,96     ;вычисление m_des.row_3
```

```
        vzeroupper
        _RestoreXmmRegs xmm6,xmm7
        _DeleteFrame
        ret
Avx2Mat4x4MulF64_ endp

; extern "C" bool Avx2Mat4x4InvF64_(double* m_inv, const double* m, double epsilon, bool*
is_singular);

; Смещения непосредственных матриц в стеке относительно rsp
OffsetM2 equ 32
OffsetM3 equ 160
OffsetM4 equ 288

Avx2Mat4x4InvF64_ proc frame
        _CreateFrame MI_,0,160
        _SaveXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

; Сохранение аргументов в домашнее пространство для последующего использования
        mov qword ptr [rbp+MI_OffsetHomeRCX],rcx        ;сохранение указателя m_inv
        mov qword ptr [rbp+MI_OffsetHomeRDX],rdx        ;сохранение указателя m ptr
        vmovsd real8 ptr [rbp+MI_OffsetHomeR8],xmm2     ;сохранение epsilon
        mov qword ptr [rbp+MI_OffsetHomeR9],r9          ;сохр. указ. is_singular

; Выделение 384 стекового пространства для временных матриц + 32 байта
; для вызовов функции
        and rsp,0fffffffe0h                 ;выравнивание rsp по 32 байтам
        sub rsp,416                         ;выделение стекового пространства

; Вычисление m2
        lea rcx,[rsp+OffsetM2]              ;rcx = m2 ptr
        mov r8,rdx                          ;rdx, r8 = m ptr
        call Avx2Mat4x4MulF64_              ;вычисление и сохранение m2

; Вычисление m3
        lea rcx,[rsp+OffsetM3]              ;rcx = m3 ptr
        lea rdx,[rsp+OffsetM2]              ;rdx = m2 ptr
        mov r8,[rbp+MI_OffsetHomeRDX]       ;r8 = m
        call Avx2Mat4x4MulF64_              ;вычисление и сохранение m3

; Вычисление m4
        lea rcx,[rsp+OffsetM4]              ;rcx = m4 ptr
        lea rdx,[rsp+OffsetM3]              ;rdx = m3 ptr
        mov r8,[rbp+MI_OffsetHomeRDX]       ;r8 = m
        call Avx2Mat4x4MulF64_              ;вычисление и сохранение m4

; Вычисление следа m, m2, m3, и m4
        mov rcx,[rbp+MI_OffsetHomeRDX]
        call Avx2Mat4x4TraceF64_
        vmovsd xmm8,xmm8,xmm0               ;xmm8 = t1

        lea rcx,[rsp+OffsetM2]
        call Avx2Mat4x4TraceF64_
        vmovsd xmm9,xmm9,xmm0               ;xmm9 = t2
```

```
        lea rcx,[rsp+OffsetM3]
        call Avx2Mat4x4TraceF64_
        vmovsd xmm10,xmm10,xmm0             ;xmm10 = t3

        lea rcx,[rsp+OffsetM4]
        call Avx2Mat4x4TraceF64_
        vmovsd xmm11,xmm11,xmm0             ;xmm10 = t4

; Вычисление необходимых коэффициентов
; 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);
;
; Используемые регистры:
;   t1-t4 = xmm8-xmm11
;   c1-c4 = xmm12-xmm15

        vxorpd xmm12,xmm8,real8 ptr [r8_SignBitMask]    ;xmm12 = c1

        vmulsd xmm13,xmm12,xmm8         ;c1 * t1
        vaddsd xmm13,xmm13,xmm9         ;c1 * t1 + t2
        vmulsd xmm13,xmm13,[r8_N0p5]    ;c2

        vmulsd xmm14,xmm13,xmm8         ;c2 * t1
        vmulsd xmm0,xmm12,xmm9          ;c1 * t2
        vaddsd xmm14,xmm14,xmm0         ;c2 * t1 + c1 * t2
        vaddsd xmm14,xmm14,xmm10        ;c2 * t1 + c1 * t2 + t3
        vmulsd xmm14,xmm14,[r8_N0p3333] ;c3

        vmulsd xmm15,xmm14,xmm8         ;c3 * t1
        vmulsd xmm0,xmm13,xmm9          ;c2 * t2
        vmulsd xmm1,xmm12,xmm10         ;c1 * t3
        vaddsd xmm2,xmm0,xmm1           ;c2 * t2 + c1 * t3
        vaddsd xmm15,xmm15,xmm2         ;c3 * t1 + c2 * t2 + c1 * t3
        vaddsd xmm15,xmm15,xmm11        ;c3 * t1 + c2 * t2 + c1 * t3 + t4
        vmulsd xmm15,xmm15,[r8_N0p25]   ;c4

; Проверка, что матрица несингулярная
        vandpd xmm0,xmm15,[r8_AbsMask]                  ;вычисление fabs(c4)
        vmovsd xmm1,real8 ptr [rbp+MI_OffsetHomeR8]
        vcomisd xmm0,real8 ptr [rbp+MI_OffsetHomeR8]    ;сравнение с epsilon
        setp al                                         ;set al = если не упорядочено
        setb ah                                         ;уст. ah = если fabs(c4) < epsilon
        or al,ah                                        ;al = is_singular
        mov rcx,[rbp+MI_OffsetHomeR9]                   ;rax = is_singular указатель
        mov [rcx],al                                    ;сохр. состояние is_singular
        jnz Error                                       ;переход, если сингулярная

; Calculate m_inv = -1.0 / c4 * (m3 + c1 * m2 + c2 * m1 + c3 * I)
        vbroadcastsd ymm14,xmm14                        ;ymm14 = упакованное c3
        lea rcx,[Mat4x4I]                               ;rcx = указатель I
        vmulpd ymm0,ymm14,ymmword ptr [rcx]
        vmulpd ymm1,ymm14,ymmword ptr [rcx+32]
        vmulpd ymm2,ymm14,ymmword ptr [rcx+64]
        vmulpd ymm3,ymm14,ymmword ptr [rcx+96]          ;c3 * I
```

```
        vbroadcastsd ymm13,xmm13                    ;ymm13 = упакованное c2
        mov rcx,[rbp+MI_OffsetHomeRDX]              ;rcx = указатель m
        vmulpd ymm4,ymm13,ymmword ptr [rcx]
        vmulpd ymm5,ymm13,ymmword ptr [rcx+32]
        vmulpd ymm6,ymm13,ymmword ptr [rcx+64]
        vmulpd ymm7,ymm13,ymmword ptr [rcx+96]      ;c2 * m1
        vaddpd ymm0,ymm0,ymm4
        vaddpd ymm1,ymm1,ymm5
        vaddpd ymm2,ymm2,ymm6
        vaddpd ymm3,ymm3,ymm7                       ;c2 * m1 + c3 * I

        vbroadcastsd ymm12,xmm12                    ;ymm12 = упакованное c1
        lea rcx,[rsp+OffsetM2]                      ;rcx = указатель m2
        vmulpd ymm4,ymm12,ymmword ptr [rcx]
        vmulpd ymm5,ymm12,ymmword ptr [rcx+32]
        vmulpd ymm6,ymm12,ymmword ptr [rcx+64]
        vmulpd ymm7,ymm12,ymmword ptr [rcx+96]      ;c1 * m2
        vaddpd ymm0,ymm0,ymm4
        vaddpd ymm1,ymm1,ymm5
        vaddpd ymm2,ymm2,ymm6
        vaddpd ymm3,ymm3,ymm7                       ;c1 * m2 + c2 * m1 + c3 * I

        lea rcx,[rsp+OffsetM3]                      ;rcx = m3 ptr
        vaddpd ymm0,ymm0,ymmword ptr [rcx]
        vaddpd ymm1,ymm1,ymmword ptr [rcx+32]
        vaddpd ymm2,ymm2,ymmword ptr [rcx+64]
        vaddpd ymm3,ymm3,ymmword ptr [rcx+96]       ;m3 + c1 * m2 + c2 * m1 + c3 * I

        vmovsd xmm4,[r8_N1p0]
        vdivsd xmm4,xmm4,xmm15              ;xmm4 = -1.0 / c4
        vbroadcastsd ymm4,xmm4
        vmulpd ymm0,ymm0,ymm4
        vmulpd ymm1,ymm1,ymm4
        vmulpd ymm2,ymm2,ymm4
        vmulpd ymm3,ymm3,ymm4               ;ymm3:ymm0 = m_inv

; Save m_inv
        mov rcx,[rbp+MI_OffsetHomeRCX]
        vmovapd ymmword ptr [rcx],ymm0
        vmovapd ymmword ptr [rcx+32],ymm1
        vmovapd ymmword ptr [rcx+64],ymm2
        vmovapd ymmword ptr [rcx+96],ymm3
        mov eax,1                           ;возвращаемый код успешного выполнения

Done:    vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm8,xmm9,xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame
        ret

Error:  xor eax,eax
        jmp Done

Avx2Mat4x4InvF64_ endp
        end
```

Мультипликативная обратная матрица определяется следующим образом: пусть **A** и **X** представляют матрицы размером *n*×*n*. Матрица **X** является обратной к **A**, если **AX** = **XA** = **I**, где **I** обозначает *единичную матрицу* размера *n*×*n* (то есть матрицу, состоящую из нулей, кроме диагональных элементов, которые равны единице). На рис. 9.5 показан пример обратной матрицы. Важно отметить, что обратные матрицы существуют не для всех матриц размера *n*×*n*. Матрица, для которой нет обратной матрицы, называется *сингулярной*.

$$A=\begin{bmatrix}6 & 2 & 2\\ 2 & -2 & 2\\ 0 & 4 & 2\end{bmatrix}\quad X=\begin{bmatrix}0.1875 & -0.0625 & -0.125\\ 0.0625 & -0.1875 & 0.125\\ -0.125 & 0.375 & 0.25\end{bmatrix}\quad \text{AX=XA=I=}\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

Рис. 9.5. Матрица A и ее мультипликативная обратная матрица X

Матрица, обратная матрице 4×4, может быть вычислена с использованием различных математических методов. В примере исходного кода `Ch09_06` применен вычислительный метод, основанный на *теореме Кэли–Гамильтона*, в котором используются стандартные матричные операции, относительно легко выполнимые с использованием арифметики SIMD. Ниже приведены необходимые уравнения:

$$\mathbf{A}^1=\mathbf{A};\mathbf{A}^2=\mathbf{AA};\mathbf{A}^3=\mathbf{AAA};\mathbf{A}^4=\mathbf{AAAA};$$

$$trace(\mathbf{A})=\sum_i a_{ii};$$

$$t_n=trace(\mathbf{A}^n);$$

$$c_1=-t_1;$$

$$c_2=-\frac{1}{2}(c_1t_1+t_2);$$

$$c_3=-\frac{1}{3}(c_2t_1+c_1t_2+t_3);$$

$$c_4=-\frac{1}{4}(c_3t_1+c_2t_2+c_1t_3+t_4);$$

$$\mathbf{A}^{-1}=-\frac{1}{c_4}(\mathbf{A}^3+c_1\mathbf{A}^2+c_2\mathbf{A}^3+c_3\mathbf{I}).$$

В начале кода C++ расположена функция `Avx2Mat4x4InvF64Cpp`. Эта функция вычисляет обратную матрицу 4×4 значений двойной точности с плавающей запятой с использованием вышеупомянутых уравнений. Функция `Avx2Mat4x4InvF64Cpp` использует класс C++ `Matrix<>` для выполнения многих требуемых промежуточных вычислений, включая сложение матриц, умножение и нахождение следа. Исходный код для класса `Matrix<>` не показан, но включен в набор файлов для скачивания этой главы. Обратите внимание, что промежуточные матрицы объявляются с использованием квалификатора `static`, чтобы избежать накладных расходов конструктора при выполнении измере-

ний быстродействия. Недостаток использования здесь квалификатора static заключается в том, что функция не является поточно-ориентированной (поточно-ориентированная функция может одновременно использоваться несколькими потоками). После вычисления значений *следа матрицы* t1–t4 функция Avx2Mat4x4InvF64Cpp вычисляет c1–c4, используя простую скалярную арифметику. Затем она проверяет, не является ли исходная матрица m сингулярной, сравнивая c4 с epsilon. Если матрица m не является сингулярной, вычисляется последняя обратная матрица. Оставшийся код C++ выполняет инициализацию тестового примера и вызывает функции обращения матриц как на C++, так и на языке ассемблера.

Код на языке ассемблера в листинге 9.6 начинается с настраиваемого сегмента, который содержит определения постоянных значений, необходимых для функций обращения матрицы на языке ассемблера. Оператор ConstVals segment readonly align (32) 'const' отмечает начало сегмента, который начинается на 32-байтовой границе и содержит данные только для чтения. Причина использования здесь настраиваемого сегмента заключается в том, что директива MASM align не поддерживает выравнивание элементов данных по 32-байтовой границе. В этом примере правильное выравнивание упакованных констант важно для максимизации производительности. Обратите внимание, что скалярные константы двойной точности с плавающей запятой определяются после 256-битных упакованных констант и выравниваются по 8-байтовой границе. Оператор MASM ConstVals ends завершает настраиваемый сегмент.

За настраиваемым сегментом констант следует макрос _Max4x4TraceF64. Этот макрос содержит команды, вычисляющие след матрицы 4×4 значений двойной точности с плавающей запятой. Макрос _Max4x4TraceF64 требует, чтобы четыре строки исходной матрицы были загружены в регистры YMM0–YMM3, и использует команды vblendpd, vperm2f128 и vhaddpd для вычисления следа матрицы, как показано на рис. 9.6. Команда vblendpd (смешивание упакованных значений двойной точности с плавающей запятой) объединяет значения из двух исходных операндов в соответствии с непосредственной управляющей маской. Если бит 0 управляющей маски равен 0, элемент 0 (то есть биты 63:0) из первого операнда-источника копируется в соответствующую позицию элемента в операнде-приемнике; в противном случае элемент 0 из второго операнда-источника копируется в операнд-источник. Биты 1–3 управляющей маски аналогичным образом используются для трех других элементов. После выполнения команды vhaddpd регистр XMM0 [63:0] содержит значение следа матрицы.

$$\mathbf{M} = \begin{bmatrix} 7 & 2 & 19 & 3 \\ 8 & 6 & 5 & 10 \\ 22 & 3 & 1 & 12 \\ 13 & 25 & 9 & 4 \end{bmatrix}$$

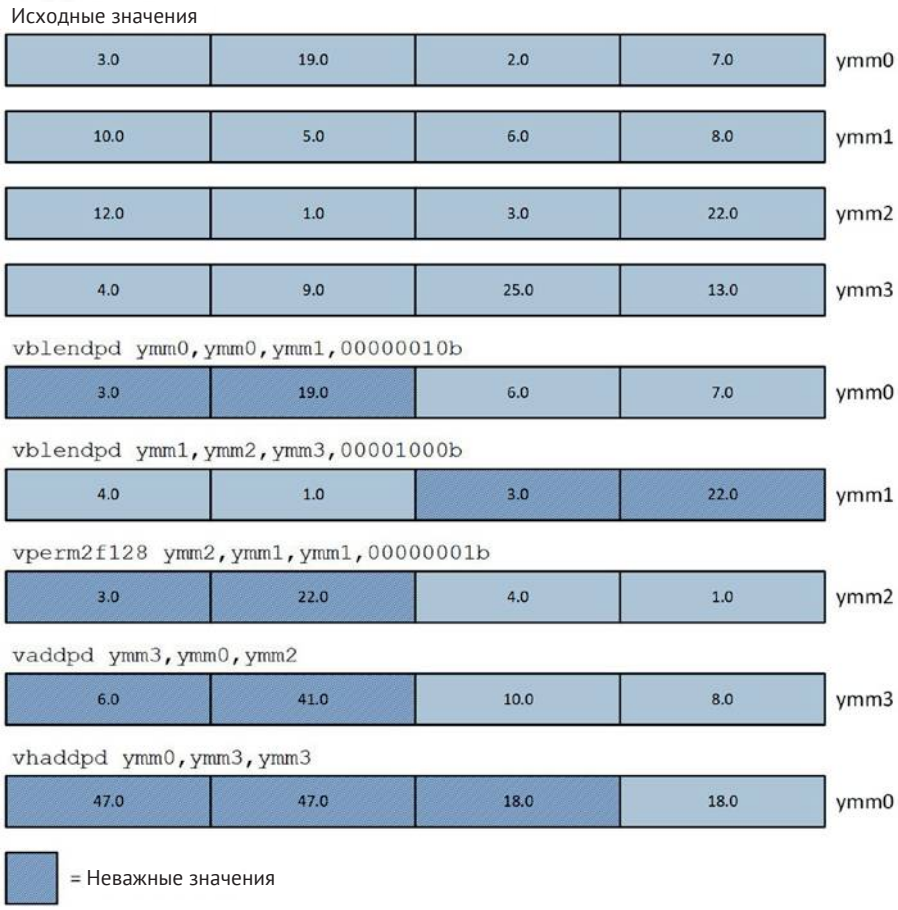


Рис. 9.6. Вычисление следа матрицы 4×4

Функция на языке ассемблера `Avx2Mat4x4InvF64_` вычисляет обратную матрицу, используя тот же метод, что и соответствующая функция C++. После пролога функция `Avx2Mat4x4InvF64_` сохраняет значения своих аргументов в домашнюю область для дальнейшего использования. Затем она выделяет место в стеке для хранения промежуточных результатов. В частности, команда `and rsp,0ffffffe0h` выравнивает RSP по 32-байтовой границе, а команда `sub rsp,416` выделяет пространство локального стека, необходимое для промежуточных матриц `m2`, `m3` и `m4`, плюс 32 байта для вызовов функций. Затем выполняется серия вызовов

функций Avx2Mat4x4MulF64_ и Avx2Mat4x4TraceF64_ для вычисления значений следа t1–t4. Код перемножения матриц, который используется в этом примере, по сути эквивалентен коду, который вы видели в примере Ch09_05. Затем вычисляются коэффициенты алгоритма c1–c4 с использованием простой скалярной арифметики с плавающей запятой. Потом проверяется коэффициент c4, чтобы убедиться, что исходная матрица не является сингулярной. Если исходная матрица действительно не является сингулярной, функция вычисляет обратную матрицу m_inv. Обратите внимание, что вся арифметика, необходимая для вычисления m_inv, выполняется с использованием простого умножения и сложения упакованных чисел двойной точности с плавающей запятой. Вот результат выполнения примера исходного кода Ch09_06:

```
Результаты Avx2Mat4x4InvF64

Тест #1 - тестовая матрица
         2           7           3           4
         5           9           6        4.75
       6.5           3           4          10
         7        5.25       8.125           6

Тест #1 - Avx2Mat4x4InvF64Cpp - обратная матрица
 -0.943926     0.91657    0.197547   -0.425579
-0.0568818    0.251148  0.00302831   -0.165952
  0.545399   -0.647656   -0.213597    0.505123
  0.412456   -0.412053   0.0561248    0.124363

Тест #1 - Avx2Mat4x4InvF64Cpp - проверочная матрица
         1           0           0           0
         0           1           0           0
         0           0           1           0
         0           0           0           1

Тест #1 - Avx2Mat4x4InvF64_ - обратная матрица
 -0.943926     0.91657    0.197547   -0.425579
-0.0568818    0.251148  0.00302831   -0.165952
  0.545399   -0.647656   -0.213597    0.505123
  0.412456   -0.412053   0.0561248    0.124363

Тест #1 - Avx2Mat4x4InvF64_ - проверочная матрица
         1           0           0           0
         0           1           0           0
         0           0           1           0
         0           0           0           1

Тест #2 - тестовая матрица
       0.5          12       17.25           4
         5           2        6.75           8
    13.125           1           3        9.75
        16       1.625           7        0.25

Тест #2 - Avx2Mat4x4InvF64Cpp - обратная матрица
0.00165165  -0.0690239   0.0549591   0.0389347
  0.135369   -0.359846    0.242038  -0.0903252
```

```
-0.0350097    0.239298   -0.183964   0.0772214
-0.0053352    0.056194   0.0603606  -0.0669085

Тест #2 - Avx2Mat4x4InvF64Cpp - проверочная матрица
         1           0           0           0
         0           1           0           0
         0           0           1           0
         0           0           0           1

Тест #2 - Avx2Mat4x4InvF64_ - обратная матрица
0.00165165  -0.0690239   0.0549591   0.0389347
  0.135369   -0.359846    0.242038  -0.0903252
-0.0350097    0.239298   -0.183964   0.0772214
-0.0053352    0.056194   0.0603606  -0.0669085

Тест #2 - Avx2Mat4x4InvF64_ - проверочная матрица
         1           0           0           0
         0           1           0           0
         0           0           1           0
         0           0           0           1

Тест #3 - тестовая матрица
         2           0           0           1
         0           4           5           0
         0           0           0           7
         0           0           0           6

Test #3 - Avx2Mat4x4InvF64Cpp - сингулярная матрица
Test #3 - Avx2Mat4x4InvF64_ - сингулярная матрица

Выполняется проверка быстродействия  Avx2Mat4x4InvF64_BM - подождите
Результаты сохранены в файл Ch09_06_Avx2Mat4x4InvF64_BM_CHROMIUM.csv
```

Таблица 9.4 содержит результаты измерения быстродействия функций обращения матрицы.

Таблица 9.4. Среднее время выполнения обращения матрицы (мс), 100 000 обращений

Процессор	C++	Ассемблер
i7-4790S	30417	4168
i9-7900X	26646	3773
i7-8700K	24485	2941

9.5. Команды смешивания и перестановки

Операция *смешивания* данных копирует по условию элементы из двух упакованных исходных операндов в упакованный целевой операнд, используя управляющую маску, которая указывает, какие элементы копировать. Операция *перестановки* данных переупорядочивает элементы упакованного исходного операнда в соответствии с управляющей маской. В этой главе вы уже видели несколько примеров исходного кода, в которых использовались опера-

ции смешивания и перестановки данных. Пример Ch09_07 демонстрирует применение дополнительных команд смешивания и перестановки. В листинге 9.7 показан исходный код этого примера.

Листинг 9.7. Пример Ch09_07

```
//-----------------------------------------------
//              Ch09_07.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include "YmmVal.h"

using namespace std;

extern "C" void AvxBlendF32_(YmmVal* des1, YmmVal* src1, YmmVal* src2, YmmVal* idx1);
extern "C" void Avx2PermuteF32_(YmmVal* des1, YmmVal* src1, YmmVal* idx1, YmmVal* des2,
YmmVal* src2, YmmVal* idx2);

void AvxBlendF32(void)
{
    const uint32_t sel0 = 0x00000000;
    const uint32_t sel1 = 0x80000000;
    alignas(32) YmmVal des1, src1, src2, idx1;

    src1.m_F32[0] = 10.0f;  src2.m_F32[0] = 100.0f;  idx1.m_I32[0] = sel1;
    src1.m_F32[1] = 20.0f;  src2.m_F32[1] = 200.0f;  idx1.m_I32[1] = sel0;
    src1.m_F32[2] = 30.0f;  src2.m_F32[2] = 300.0f;  idx1.m_I32[2] = sel0;
    src1.m_F32[3] = 40.0f;  src2.m_F32[3] = 400.0f;  idx1.m_I32[3] = sel1;
    src1.m_F32[4] = 50.0f;  src2.m_F32[4] = 500.0f;  idx1.m_I32[4] = sel1;
    src1.m_F32[5] = 60.0f;  src2.m_F32[5] = 600.0f;  idx1.m_I32[5] = sel0;
    src1.m_F32[6] = 70.0f;  src2.m_F32[6] = 700.0f;  idx1.m_I32[6] = sel1;
    src1.m_F32[7] = 80.0f;  src2.m_F32[7] = 800.0f;  idx1.m_I32[7] = sel0;

    AvxBlendF32_(&des1, &src1, &src2, &idx1);

    cout << "\nРезультаты AvxBlendF32 (vblendvps)\n";
    cout << fixed << setprecision(1);

    for (size_t i = 0; i < 8; i++)
    {
        cout << "i: " << setw(2) << i << "  ";
        cout << "src1: " << setw(8) << src1.m_F32[i] << "  ";
        cout << "src2: " << setw(8) << src2.m_F32[i] << "  ";
        cout << setfill('0');
        cout << "idx1: 0x" << setw(8) << hex << idx1.m_U32[i] << "  ";
        cout << setfill(' ');
        cout << "des1: " << setw(8) << des1.m_F32[i] << '\n';
    }
}

void Avx2PermuteF32(void)
{
    alignas(32) YmmVal des1, src1, idx1;
```

```
    alignas(32) YmmVal des2, src2, idx2;

    // значение idx1 должно быть между 0 и 7.
    src1.m_F32[0] = 100.0f;       idx1.m_I32[0] = 3;
    src1.m_F32[1] = 200.0f;       idx1.m_I32[1] = 7;
    src1.m_F32[2] = 300.0f;       idx1.m_I32[2] = 0;
    src1.m_F32[3] = 400.0f;       idx1.m_I32[3] = 4;
    src1.m_F32[4] = 500.0f;       idx1.m_I32[4] = 6;
    src1.m_F32[5] = 600.0f;       idx1.m_I32[5] = 6;
    src1.m_F32[6] = 700.0f;       idx1.m_I32[6] = 1;
    src1.m_F32[7] = 800.0f;       idx1.m_I32[7] = 2;

    // значение idx2 должно быть между 0 и 3.
    src2.m_F32[0] = 100.0f;       idx2.m_I32[0] = 3;
    src2.m_F32[1] = 200.0f;       idx2.m_I32[1] = 1;
    src2.m_F32[2] = 300.0f;       idx2.m_I32[2] = 1;
    src2.m_F32[3] = 400.0f;       idx2.m_I32[3] = 2;
    src2.m_F32[4] = 500.0f;       idx2.m_I32[4] = 3;
    src2.m_F32[5] = 600.0f;       idx2.m_I32[5] = 2;
    src2.m_F32[6] = 700.0f;       idx2.m_I32[6] = 0;
    src2.m_F32[7] = 800.0f;       idx2.m_I32[7] = 0;

    Avx2PermuteF32_(&des1, &src1, &idx1, &des2, &src2, &idx2);

    cout << "\nРезультаты Avx2PermuteF32 (vpermps)\n";
    cout << fixed << setprecision(1);

    for (size_t i = 0; i < 8; i++)
    {
        cout << "i: " << setw(2) << i << "  ";
        cout << "src1: " << setw(8) << src1.m_F32[i] << "  ";
        cout << "idx1: " << setw(8) << idx1.m_I32[i] << "  ";
        cout << "des1: " << setw(8) << des1.m_F32[i] << '\n';
    }

    cout << "\nРезультаты Avx2PermuteF32 (vpermilps)\n";

    for (size_t i = 0; i < 8; i++)
    {
        cout << "i: " << setw(2) << i << "  ";
        cout << "src2: " << setw(8) << src2.m_F32[i] << "  ";
        cout << "idx2: " << setw(8) << idx2.m_I32[i] << "  ";
        cout << "des2: " << setw(8) << des2.m_F32[i] << '\n';
    }
}

int main()
{
    AvxBlendF32();
    Avx2PermuteF32();
    return 0;
}

;-----------------------------------------------
;               Ch09_07.asm
;-----------------------------------------------
```

```
; extern "C" void AvxBlendF32_(YmmVal* des1, YmmVal* src1, YmmVal* src2, YmmVal* idx1)

        .code
AvxBlendF32_ proc
        vmovaps ymm0,ymmword ptr [rdx]  ;ymm0 = src1
        vmovaps ymm1,ymmword ptr [r8]   ;ymm1 = src2
        vmovdqa ymm2,ymmword ptr [r9]   ;ymm2 = idx1
        vblendvps ymm3,ymm0,ymm1,ymm2   ;смешивание ymm0 & ymm1, ymm2 "индексы"
        vmovaps ymmword ptr [rcx],ymm3  ;сохранение результата в des1

        vzeroupper
        ret
AvxBlendF32_ endp

; extern "C" void Avx2PermuteF32_(YmmVal* des1, YmmVal* src1, YmmVal* idx1, YmmVal* des2,
YmmVal* src2, YmmVal* idx2)

Avx2PermuteF32_ proc

; Перестановка vpermps
        vmovaps ymm0,ymmword ptr [rdx]      ;ymm0 = src1
        vmovdqa ymm1,ymmword ptr [r8]       ;ymm1 = idx1
        vpermps ymm2,ymm1,ymm0              ;перестановка ymm0, индексы в ymm1
        vmovaps ymmword ptr [rcx],ymm2      ;сохранение результата в des1

; Перестановка vpermilps
        mov rdx,[rsp+40]                    ;rdx = src2 ptr
        mov r8,[rsp+48]                     ;r8 = idx2 ptr
        vmovaps ymm3,ymmword ptr [rdx]      ;ymm3 = src2
        vmovdqa ymm4,ymmword ptr [r8]       ;ymm4 = idx1
        vpermilps ymm5,ymm3,ymm4            ;перестановка ymm3, индексы в ymm4
        vmovaps ymmword ptr [r9],ymm5       ;сохранение результата в des2

        vzeroupper
        ret
Avx2PermuteF32_ endp
        end
```

Код C++ в листинге 9.7 начинается с функции `AvxBlendF32`, которая инициализирует переменные `src1` и `src2` `YmmVal`, используя значения с плавающей запятой одинарной точности. Она также инициализирует третью переменную `YmmVal` с именем `src3` для использования в качестве маски управления смешиванием. Старший бит каждого элемента двойного слова в `src3` указывает, копируется ли соответствующий элемент из `src1` (старший бит = 0) или `src2` (старший бит = 1) в операнд-приемник. Эти три исходных операнда используются командой `vblendvps` (условное смешивание упакованных значений одинарной точности с плавающей запятой), которая находится в ассемблерной функции `AvxBlendF32_`. После выполнения этой функции результаты передаются в `cout`.

Код C++ в листинге 9.7 также содержит функцию `Avx2PermuteF32`. Эта функция инициализирует несколько переменных `YmmVal`, демонстрирующих использование команд `vpermps` и `vpermips`. Обе эти команды требуют набора индексов,

которые определяют, какие элементы исходного операнда копируются в целевой операнд. Например, оператор `idx1.m_I32[0]=3` указывает команде `vpermps` в `Avx2PermuteF32_` выполнить операцию `des1.m_F32[0]=src1.m_F32[3]`. Команда `vpermps` требует, чтобы каждый индекс в `idx1` был от нуля до семи. Индекс в `idx1` можно использовать более одного раза, чтобы скопировать элемент из `src1` в несколько мест в `des1`. Команда `vpermilps` требует, чтобы ее индексы были от нуля до трех.

Функция на языке ассемблера `AvxBlendF32_` начинается с загрузки операндов исходных данных в регистры YMM0 и YMM1 с помощью двух команд `vmovaps`. Затем команда `vmovdqa` загружает маску управления смешением в регистр YMM2. Следующая команда `vblendvps ymm3,ymm0,ymm1,ymm2` смешивает элементы из регистров YMM0 и YMM1 в YMM3 в соответствии с управляющими значениями в YMM2. Старший бит каждого элемента двойного слова в YMM2 указывает, копируется ли соответствующий элемент из YMM0 (старший бит = 0) или YMM1 (старший бит = 1) в YMM3. Рисунок 9.7 более подробно иллюстрирует выполнение этой команды. Инструкция `vblendvps` и ее аналог с двойной точностью `vblendvpd` являются примерами команд AVX, для которых требуется три исходных операнда. Операции смешивания с плавающей запятой с использованием непосредственной маски управления реализуются с помощью команд `vblendp[d|s]`.

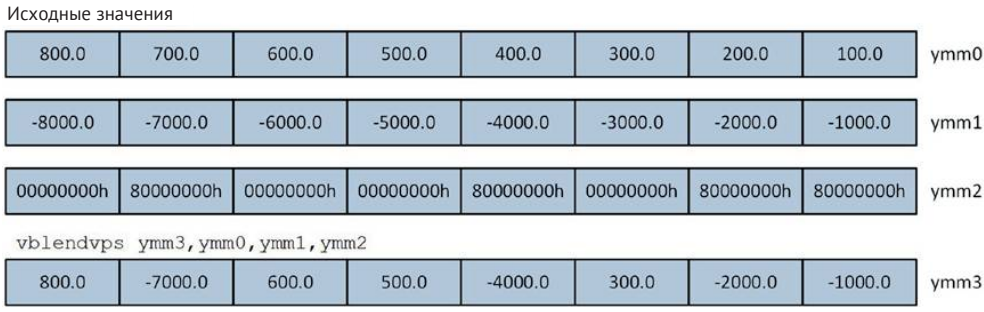


Рис. 9.7. Выполнение команды `vblendvps`

Следом за `AvxBlendF32_` в листинге 9.7 идет функция `Avx2PermuteF32_`, которая демонстрирует использование команд `vpermps` и `vpermilps`. Инструкция `vpermps` переставляет (или переупорядочивает) элементы своего первого исходного операнда (который имеет ширину 256 бит и содержит восемь значений с плавающей запятой одинарной точности) в соответствии с индексами во втором исходном операнде. Команда `vpermilps` (перестановка в полосе значений с плавающей запятой одинарной точности) выполняет свои перестановки, используя две независимые полосы шириной 128 бит (т. е. биты [255:128] и [127:0]). Управляющие индексы для внутренней перестановки должны находиться в диапазоне от нуля до трех, и каждая полоса использует свой собственный отдельный набор индексов. На рис. 9.8 более подробно показано выполнение этих команд. AVX и AVX2 также содержат команды перестановки с плавающей запятой двойной точности `vpermilpd` и `vpermpd`.

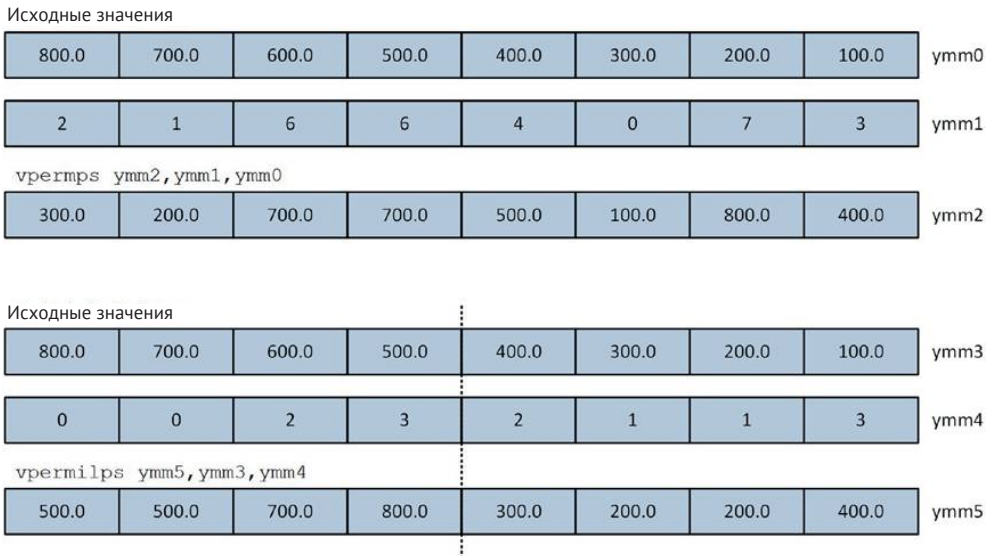


Рис. 9.8. Выполнение команд `vpermps` и `vpermilps`

Ниже показан результат выполнения примера исходного кода `Ch09_07`:

```
Результаты AvxBlendF32 (vblendvps)
i:  0  src1:     10.0  src2:    100.0  idx1: 0x80000000  des1:  100.0
i:  1  src1:     20.0  src2:    200.0  idx1: 0x00000000  des1:   20.0
i:  2  src1:     30.0  src2:    300.0  idx1: 0x00000000  des1:   30.0
i:  3  src1:     40.0  src2:    400.0  idx1: 0x80000000  des1:  400.0
i:  4  src1:     50.0  src2:    500.0  idx1: 0x80000000  des1:  500.0
i:  5  src1:     60.0  src2:    600.0  idx1: 0x00000000  des1:   60.0
i:  6  src1:     70.0  src2:    700.0  idx1: 0x80000000  des1:  700.0
i:  7  src1:     80.0  src2:    800.0  idx1: 0x00000000  des1:   80.0

Результаты Avx2PermuteF32 (vpermps)
i:  0  src1:    100.0  idx1:        3  des1:    400.0
i:  1  src1:    200.0  idx1:        7  des1:    800.0
i:  2  src1:    300.0  idx1:        0  des1:    100.0
i:  3  src1:    400.0  idx1:        4  des1:    500.0
i:  4  src1:    500.0  idx1:        6  des1:    700.0
i:  5  src1:    600.0  idx1:        6  des1:    700.0
i:  6  src1:    700.0  idx1:        1  des1:    200.0
i:  7  src1:    800.0  idx1:        2  des1:    300.0

Результаты Avx2PermuteF32 (vpermilps)
i:  0  src2:    100.0  idx2:        3  des2:    400.0
i:  1  src2:    200.0  idx2:        1  des2:    200.0
i:  2  src2:    300.0  idx2:        1  des2:    200.0
i:  3  src2:    400.0  idx2:        2  des2:    300.0
i:  4  src2:    500.0  idx2:        3  des2:    800.0
i:  5  src2:    600.0  idx2:        2  des2:    700.0
i:  6  src2:    700.0  idx2:        0  des2:    500.0
i:  7  src2:    800.0  idx2:        0  des2:    500.0
```

9.6. Команды извлечения данных

Последний пример исходного кода этой главы, Ch09_08, объясняет, как использовать команды извлечения данных AVX2. Команда извлечения данных по условию загружает элементы из несмежных ячеек памяти (обычно массива) в регистры XMM или YMM. Для команды извлечения данных требуется набор индексов и маска управления слиянием, которая указывает, какие элементы копировать. В листинге 9.8 показан исходный код примера Ch09_08. В главе 8 представлен обзор команд извлечения данных AVX2, включая рис. 8.1, поясняющий выполнение команды vgatherdps. Возможно, вам будет полезно вернуться к этому материалу перед просмотром исходного кода и его обсуждением в этом разделе.

Листинг 9.8. Пример Ch09_08

```
//-----------------------------------------------
//               Ch09_08.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <string>
#include <cstdint>
#include <iostream>
#include <iomanip>
#include <array>
#include <stdexcept>

using namespace std;

extern "C" void Avx2Gather8xF32_I32_(float* y, const float* x,
    const int32_t* indices, const int32_t* masks);
extern "C" void Avx2Gather8xF32_I64_(float* y, const float* x,
    const int64_t* indices, const int32_t* masks);
extern "C" void Avx2Gather8xF64_I32_(double* y, const double* x,
    const int32_t* indices, const int64_t* masks);
extern "C" void Avx2Gather8xF64_I64_(double* y, const double* x,
    const int64_t* indices, const int64_t* masks);

template <typename T, typename I, typename M, size_t N>
    void Print(const string& msg, const array<T, N>& y, const array<I, N>& indices,
    const array<M, N>& merge)
{
    if (y.size() != indices.size() || y.size() != merge.size())
        throw runtime_error("Non-conforming arrays - Print");

    cout << '\n' << msg << '\n';

    for (size_t i = 0; i < y.size(); i++)
    {
        string merge_s = (merge[i] == 1) ? "Yes" : "No";

        cout << "i: " << setw(2) << i << "   ";
        cout << "y: " << setw(10) << y[i] << "   ";
        cout << "index: " << setw(4) << indices[i] << "   ";
        cout << "merge: " << setw(4) << merge_s << '\n';
    }
}
```

```
void Avx2Gather8xF32_I32()
{
    array<float, 20> x;

    for (size_t i = 0; i < x.size(); i++)
        x[i] = (float)(i * 10);

    array<float, 8> y { -1, -1, -1, -1, -1, -1, -1, -1 };
    array<int32_t, 8> indices { 2, 1, 6, 5, 4, 13, 11, 9 };
    array<int32_t, 8> merge { 1, 1, 0, 1, 1, 0, 1, 1 };

    cout << fixed << setprecision(1);
    cout << "\nРезультаты Avx2Gather8xF32_I32\n";

    Print("Значения до", y, indices, merge);
    Avx2Gather8xF32_I32_(y.data(), x.data(), indices.data(), merge.data());
    Print("Значения после", y, indices, merge);
}

void Avx2Gather8xF32_I64()
{
    array<float, 20> x;

    for (size_t i = 0; i < x.size(); i++)
        x[i] = (float)(i * 10);

    array<float, 8> y { -1, -1, -1, -1, -1, -1, -1, -1 };
    array<int64_t, 8> indices { 19, 1, 0, 5, 4, 3, 11, 11 };
    array<int32_t, 8> merge { 1, 1, 1, 1, 0, 0, 1, 1 };

    cout << fixed << setprecision(1);
    cout << "\nРезультаты Avx2Gather8xF32_I64\n";

    Print("Values before", y, indices, merge);
    Avx2Gather8xF32_I64_(y.data(), x.data(), indices.data(), merge.data());
    Print("Значения до", y, indices, merge);
}

void Avx2Gather8xF64_I32()
{
    array<double, 20> x;

    for (size_t i = 0; i < x.size(); i++)
        x[i] = (double)(i * 10);

    array<double, 8> y { -1, -1, -1, -1, -1, -1, -1, -1 };
    array<int32_t, 8> indices { 12, 11, 6, 15, 4, 13, 18, 3 };
    array<int64_t, 8> merge { 1, 1, 0, 1, 1, 0, 1, 0 };

    cout << fixed << setprecision(1);
    cout << "\nРезультаты Avx2Gather8xF64_I32\n";

    Print("Значения до", y, indices, merge);
    Avx2Gather8xF64_I32_(y.data(), x.data(), indices.data(), merge.data());
    Print("Значения после", y, indices, merge);
}
```

```
void Avx2Gather8xF64_I64()
{
    array<double, 20> x;

    for (size_t i = 0; i < x.size(); i++)
        x[i] = (double)(i * 10);

    array<double, 8> y { -1, -1, -1, -1, -1, -1, -1, -1 };
    array<int64_t, 8> indices { 11, 17, 1, 6, 14, 13, 8, 8 };
    array<int64_t, 8> merge { 1, 0, 1, 1, 1, 0, 1, 1 };

    cout << fixed << setprecision(1);
    cout << "\nРезультаты Avx2Gather8xF64_I64\n";

    Print("Значения до", y, indices, merge);
    Avx2Gather8xF64_I64_(y.data(), x.data(), indices.data(), merge.data());
    Print("Значения после", y, indices, merge);
}

int main()
{
    Avx2Gather8xF32_I32();
    Avx2Gather8xF32_I64();
    Avx2Gather8xF64_I32();
    Avx2Gather8xF64_I64();
    return 0;
}

;-----------------------------------------------
;               Ch09_08.asm
;-----------------------------------------------

; Для каждой из следующих команд содержимое y присваивается ymm0
; до выполнения команды vgatherXXX с целью демонстрации эффекта
; последовательного объединения.

        .code
; extern "C" void Avx2Gather8xF32_I32_(float* y, const float* x, const int32_t* indices,
const int32_t* merge)

Avx2Gather8xF32_I32_ proc
        vmovups ymm0,ymmword ptr [rcx]      ;ymm0 = y[7]:y[0]
        vmovdqu ymm1,ymmword ptr [r8]       ;ymm1 = indices[7]:indices[0]
        vmovdqu ymm2,ymmword ptr [r9]       ;ymm2 = merge[7]:merge[0]
        vpslld ymm2,ymm2,31                 ;сдвиг значений в старшие разряды
        vgatherdps ymm0,[rdx+ymm1*4],ymm2   ;ymm0 = извлеченные элементы
        vmovups ymmword ptr [rcx],ymm0      ;сохранение извлеченных элементов

        vzeroupper
        ret
Avx2Gather8xF32_I32_ endp

; extern "C" void Avx2Gather8xF32_I64_(float* y, const float* x, const int64_t* indices,
const int32_t* merge)

Avx2Gather8xF32_I64_ proc
```

```
        vmovups xmm0,xmmword ptr [rcx]      ;xmm0 = y[3]:y[0]
        vmovdqu ymm1,ymmword ptr [r8]       ;ymm1 = indices[3]:indices[0]
        vmovdqu xmm2,xmmword ptr [r9]       ;xmm2 = merge[3]:merge[0]
        vpslld xmm2,xmm2,31                 ;сдвиг значений в старшие разряды
        vgatherqps xmm0,[rdx+ymm1*4],xmm2   ;xmm0 = извлеченные элементы
        vmovups xmmword ptr [rcx],xmm0      ;сохранение извлеченных элементов

        vmovups xmm3,xmmword ptr [rcx+16]   ;xmm0 = des[7]:des[4]
        vmovdqu ymm1,ymmword ptr [r8+32]    ;ymm1 = indices[7]:indices[4]
        vmovdqu xmm2,xmmword ptr [r9+16]    ;xmm2 = merge[7]:merge[4]
        vpslld xmm2,xmm2,31                 ;сдвиг значений в старшие разряды
        vgatherqps xmm3,[rdx+ymm1*4],xmm2   ;xmm0 = извлеченные элементы
        vmovups xmmword ptr [rcx+16],xmm3   ;сохранение извлеченных элементов

        vzeroupper
        ret
Avx2Gather8xF32_I64_ endp

; extern "C" void Avx2Gather8xF64_I32_(double* y, const double* x, const int32_t* indices,
const int64_t* merge)

Avx2Gather8xF64_I32_ proc
        vmovupd ymm0,ymmword ptr [rcx]      ;ymm0 = y[3]:y[0]
        vmovdqu xmm1,xmmword ptr [r8]       ;xmm1 = indices[3]:indices[0]
        vmovdqu ymm2,ymmword ptr [r9]       ;ymm2 = merge[3]:merge[0]
        vpsllq ymm2,ymm2,63                 ;сдвиг значений в старшие разряды
        vgatherdpd ymm0,[rdx+xmm1*8],ymm2   ;ymm0 = извлеченные элементы
        vmovupd ymmword ptr [rcx],ymm0      ;сохранение извлеченных элементов

        vmovupd ymm0,ymmword ptr [rcx+32]   ;ymm0 = y[7]:y[4]
        vmovdqu xmm1,xmmword ptr [r8+16]    ;xmm1 = indices[7]:indices[4]
        vmovdqu ymm2,ymmword ptr [r9+32]    ;ymm2 = merge[7]:merge[4]
        vpsllq ymm2,ymm2,63                 ;shift merge vals to high-order bits
        vgatherdpd ymm0,[rdx+xmm1*8],ymm2   ;ymm0 = извлеченные элементы
        vmovupd ymmword ptr [rcx+32],ymm0   ;сохранение извлеченных элементов

        vzeroupper
        ret
Avx2Gather8xF64_I32_ endp

; extern "C" void Avx2Gather8xF64_I64_(double* y, const double* x, const int64_t* indices,
const int64_t* merge)

Avx2Gather8xF64_I64_ proc
        vmovupd ymm0,ymmword ptr [rcx]      ;ymm0 = y[3]:y[0]
        vmovdqu ymm1,ymmword ptr [r8]       ;ymm1 = indices[3]:indices[0]
        vmovdqu ymm2,ymmword ptr [r9]       ;ymm2 = merge[3]:merge[0]
        vpsllq ymm2,ymm2,63                 ;сдвиг значений в старшие разряды
        vgatherqpd ymm0,[rdx+ymm1*8],ymm2   ;ymm0 = извлеченные элементы
        vmovupd ymmword ptr [rcx],ymm0      ;сохранение извлеченных элементов

        vmovupd ymm0,ymmword ptr [rcx+32]   ;ymm0 = y[7]:y[4]
        vmovdqu ymm1,ymmword ptr [r8+32]    ;ymm1 = indices[7]:indices[4]
        vmovdqu ymm2,ymmword ptr [r9+32]    ;ymm2 = merge[7]:merge[4]
        vpsllq ymm2,ymm2,63                 ;сдвиг значений в старшие разряды
        vgatherqpd ymm0,[rdx+ymm1*8],ymm2   ;ymm0 = извлеченные элементы
```

```
        vmovupd ymmword ptr [rcx+32],ymm0   ;сохранение извлеченных элементов

        vzeroupper
        ret
Avx2Gather8xF64_I64_ endp
        end
```

Исходный код C++ в примере `Ch09_08` включает четыре функции, которые инициализируют тестовые примеры для выполнения операций извлечения данных с плавающей запятой одинарной и двойной точности с использованием индексов в виде двойного слова или четверного слова со знаком. Функция `Avx2Gather8xF32_I32` начинается с инициализации элементов исходного массива `x` тестовыми значениями. Обратите внимание, что эта функция вместо необработанного массива C++ использует класс STL `array<>`, чтобы продемонстрировать его применение с функцией на языке ассемблера. Приложение содержит список ссылок на документацию C++, с которой вы можете ознакомиться, если хотите узнать больше об этом классе. Затем каждый элемент в целевом массиве `y` устанавливается в `-1.0`, чтобы проиллюстрировать эффекты слияния по условию. Массивы `indices` и `merges` также заполняются необходимыми индексами команд сбора и значениями маски управления слиянием соответственно. Потом вызывается ассемблерная функция `Avx2Gather8xF32_I32_` для выполнения операции извлечения данных. Обратите внимание, что необработанные указатели для различных массивов STL получаются с помощью шаблонной функции `array<>.data`. Другие функции C++ в этом исходном примере – `Avx2Gather8xF32_I64`, `Avx2Gather8xF64_I32` и `Avx2Gather8xF64_I64` – имеют аналогичную структуру.

Функция языка ассемблера `Avx2Gather8xF32_I32_` начинается с загрузки в регистры YMM0, YMM1 и YMM2 тестовых массивов `y`, `indices` и `merges` соответственно. Регистр RDX содержит указатель на исходный массив `x`. Команда `vpslld ymm2,ymm2,31` сдвигает значения маски управления слиянием (каждое значение в этой маске равно нулю или единице) в старший бит каждого элемента двойного слова. Следующая команда `vgatherdps ymm0,[rdx+ymm1*4],ymm2` загружает восемь значений с плавающей запятой одинарной точности из массива `x` в регистр YMM0. Маска управления слиянием в YMM2 определяет, какие элементы массива фактически копируются в целевой операнд YMM0. Если старший бит двойного слова элемента маски установлен в 1, соответствующий элемент в YMM0 обновляется; в противном случае он не изменяется. После успешной загрузки элемента массива команда `vgatherdps` обнуляет соответствующий элемент двойного слова в маске управления слиянием. Затем команда `vmovups ymmword ptr [rcx],ymm0` сохраняет результат извлечения в `y`.

Функции на языке ассемблера `Avx2Gather8xF32_I64_`, `Avx2Gather8xF64_I32_` и `Avx2Gather8xF64_I64_` аналогичны `Avx2Gather8xF32_I32_`. Обратите внимание, что команды извлечения, используемые в этих функциях, – `vgatherqps`, `vgatherdpd` и `vgatherqpd` – извлекают только четыре элемента, что объясняет их двукратное использование. Вот результаты выполнения примера исходного кода `Ch09_08`:

```
Результаты Avx2Gather8xF32_I32

Значения до
i:  0   y:      -1.0 index:     2  merge:  Yes
i:  1   y:      -1.0 index:     1  merge:  Yes
i:  2   y:      -1.0 index:     6  merge:   No
i:  3   y:      -1.0 index:     5  merge:  Yes
i:  4   y:      -1.0 index:     4  merge:  Yes
i:  5   y:      -1.0 index:    13  merge:   No
i:  6   y:      -1.0 index:    11  merge:  Yes
i:  7   y:      -1.0 index:     9  merge:  Yes

Значения после
i:  0   y:      20.0 index:     2  merge:  Yes
i:  1   y:      10.0 index:     1  merge:  Yes
i:  2   y:      -1.0 index:     6  merge:   No
i:  3   y:      50.0 index:     5  merge:  Yes
i:  4   y:      40.0 index:     4  merge:  Yes
i:  5   y:      -1.0 index:    13  merge:   No
i:  6   y:     110.0 index:    11  merge:  Yes
i:  7   y:      90.0 index:     9  merge:  Yes

Результаты Avx2Gather8xF32_I64

Значения до
i:  0   y:      -1.0 index:    19  merge:  Yes
i:  1   y:      -1.0 index:     1  merge:  Yes
i:  2   y:      -1.0 index:     0  merge:  Yes
i:  3   y:      -1.0 index:     5  merge:  Yes
i:  4   y:      -1.0 index:     4  merge:   No
i:  5   y:      -1.0 index:     3  merge:   No
i:  6   y:      -1.0 index:    11  merge:  Yes
i:  7   y:      -1.0 index:    11  merge:  Yes

Значения после
i:  0   y:     190.0 index:    19  merge:  Yes
i:  1   y:      10.0 index:     1  merge:  Yes
i:  2   y:       0.0 index:     0  merge:  Yes
i:  3   y:      50.0 index:     5  merge:  Yes
i:  4   y:      -1.0 index:     4  merge:   No
i:  5   y:      -1.0 index:     3  merge:   No
i:  6   y:     110.0 index:    11  merge:  Yes
i:  7   y:     110.0 index:    11  merge:  Yes

Результаты Avx2Gather8xF64_I32

Значения до
i:  0   y:      -1.0 index:    12  merge:  Yes
i:  1   y:      -1.0 index:    11  merge:  Yes
i:  2   y:      -1.0 index:     6  merge:   No
i:  3   y:      -1.0 index:    15  merge:  Yes
i:  4   y:      -1.0 index:     4  merge:  Yes
i:  5   y:      -1.0 index:    13  merge:   No
i:  6   y:      -1.0 index:    18  merge:  Yes
i:  7   y:      -1.0 index:     3  merge:   No
```

```
Значения после
i:  0   y:     120.0 index:    12  merge:  Yes
i:  1   y:     110.0 index:    11  merge:  Yes
i:  2   y:      -1.0 index:     6  merge:   No
i:  3   y:     150.0 index:    15  merge:  Yes
i:  4   y:      40.0 index:     4  merge:  Yes
i:  5   y:      -1.0 index:    13  merge:   No
i:  6   y:     180.0 index:    18  merge:  Yes
i:  7   y:      -1.0 index:     3  merge:   No

Результаты Avx2Gather8xF64_I64

Значения до
i:  0   y:      -1.0 index:    11  merge:  Yes
i:  1   y:      -1.0 index:    17  merge:   No
i:  2   y:      -1.0 index:     1  merge:  Yes
i:  3   y:      -1.0 index:     6  merge:  Yes
i:  4   y:      -1.0 index:    14  merge:  Yes
i:  5   y:      -1.0 index:    13  merge:   No
i:  6   y:      -1.0 index:     8  merge:  Yes
i:  7   y:      -1.0 index:     8  merge:  Yes

Значения после
i:  0   y:     110.0 index:    11  merge:  Yes
i:  1   y:      -1.0 index:    17  merge:   No
i:  2   y:      10.0 index:     1  merge:  Yes
i:  3   y:      60.0 index:     6  merge:  Yes
i:  4   y:     140.0 index:    14  merge:  Yes
i:  5   y:      -1.0 index:    13  merge:   No
i:  6   y:      80.0 index:     8  merge:  Yes
i:  7   y:      80.0 index:     8  merge:  Yes
```

9.7. Заключение

В главе 9 мы рассмотрели следующие ключевые моменты:

- почти все команды AVX для операций с упакованными числами с плавающей запятой одинарной и двойной точности могут использоваться как с 128-битными, так и с 256-битными операндами. Упакованные операнды с плавающей запятой всегда должны быть правильно выровнены, когда это возможно, как описано в данной главе;
- директива MASM `align` не может применяться для выравнивания 256-битных операндов по 32-байтовой границе. Код на языке ассемблера может выравнивать 256-битные константы или изменяемые операнды по 32-байтовой границе с помощью директивы MASM `segment`;
- при выполнении упакованных арифметических операций команды `vcmpp[d|s]` могут использоваться с командами `vandp[d|s]`, `vandnp[d|s]` и `vorp[d|s]` для принятия логических решений без какой-либо команды условного перехода;
- неассоциативность арифметики с плавающей запятой означает, что при сравнении значений, рассчитанных с использованием функций на языке C++ и ассемблера, могут возникнуть расхождения;

- функции на языке ассемблера могут использовать команды `vperm2f128`, `vpermp[d|s]` и `vpermilp[d|s]` для перестановки элементов упакованного операнда с плавающей запятой;
- функции на языке ассемблера могут использовать команды `vblendp[d|s]` и `vblendvp[d|s]` для чередования элементов двух упакованных операндов с плавающей запятой;
- функции на языке ассемблера могут использовать команды `vgatherdp[d|s]` и `vgatherqp[d|s]` для условной загрузки значений с плавающей запятой из несмежных участков памяти в регистры XMM или YMM;
- функции на языке ассемблера, выполняющие вычисления с использованием регистра YMM, также должны использовать команду `vzeroupper` перед любым кодом эпилога или командой `ret`, чтобы избежать потенциальных задержек при переходе от состояния x86-AVX к x86-SSE.

Глава 10

Программирование AVX2 – упакованные целые числа

В главе 7 вы узнали, как использовать набор команд AVX для выполнения упакованных целочисленных операций с использованием 128-битных операндов и набора регистров XMM. В этой главе вы узнаете, как выполнять аналогичные операции, используя команды AVX2 с операндами шириной 256 бит и набором регистров YMM. Примеры исходного кода главы 10 разделены на два основных раздела. Первый раздел содержит элементарные примеры, которые иллюстрируют основные операции с использованием команд AVX2 и 256-битных упакованных целочисленных операндов. Второй раздел включает примеры, которые являются продолжением методов обработки изображений, впервые представленных в главе 7.

Для всех примеров исходного кода в этой главе требуется процессор и операционная система, поддерживающие AVX2. Вы можете использовать одну из бесплатных утилит, перечисленных в приложении, чтобы проверить возможности вашей системы.

10.1. Основные операции над упакованными целыми числами

В этом разделе вы узнаете, как выполнять операции над упакованными целыми числами с помощью команд AVX2. В первом примере исходного кода излагаются основы арифметики с использованием операндов шириной 256 бит и набора регистров YMM. Второй пример исходного кода демонстрирует команды AVX2, которые выполняют операции упаковки и распаковки целых чисел. В этом примере также объясняется, как вернуть структуру по значению из функции на языке ассемблера. Последний пример исходного кода освещает команды AVX2, которые выполняют увеличение размера упакованного целого числа с использованием дополнения знаковым битом или нулем.

10.1.1. Основные арифметические операции

В листинге 10.1 показан исходный код примера Ch10_01. В этом примере показано, как выполнять основные арифметические операции с использованием упакованных операндов в формате слова и двойного слова.

Листинг 10.1. Пример Ch10_01

```
//-----------------------------------------------
//              Ch10_01.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include "Ymmval.h"

using namespace std;

extern "C" void Avx2PackedMathI16_(const YmmVal& a, const YmmVal& b, YmmVal c[6]);
extern "C" void Avx2PackedMathI32_(const YmmVal& a, const YmmVal& b, YmmVal c[5]);

void Avx2PackedMathI16(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b;
    alignas(32) YmmVal c[6];

    a.m_I16[0] = 10;       b.m_I16[0] = 1000;
    a.m_I16[1] = 20;       b.m_I16[1] = 2000;
    a.m_I16[2] = 3000;     b.m_I16[2] = 30;
    a.m_I16[3] = 4000;     b.m_I16[3] = 40;

    a.m_I16[4] = 30000;    b.m_I16[4] = 3000;       // переполнение при сложении
    a.m_I16[5] = 6000;     b.m_I16[5] = 32000;      // переполнение при сложении
    a.m_I16[6] = 2000;     b.m_I16[6] = -31000;     // переполнение при вычитании
    a.m_I16[7] = 4000;     b.m_I16[7] = -30000;     // переполнение при вычитании

    a.m_I16[8] = 4000;     b.m_I16[8] = -2500;
    a.m_I16[9] = 3600;     b.m_I16[9] = -1200;
    a.m_I16[10] = 6000;    b.m_I16[10] = 9000;
    a.m_I16[11] = -20000;  b.m_I16[11] = -20000;

    a.m_I16[12] = -25000;  b.m_I16[12] = -27000;    // переполнение при сложении
    a.m_I16[13] = 8000;    b.m_I16[13] = 28700;     // переполнение при сложении
    a.m_I16[14] = 3;       b.m_I16[14] = -32766;    // переполнение при вычитании
    a.m_I16[15] = -15000;  b.m_I16[15] = 24000;     // переполнение при вычитании

    Avx2PackedMathI16_(a, b, c);

    cout <<"\nРезультаты Avx2PackedMathI16_\n\n";
    cout << " i        a        b   vpaddw  vpaddsw   vpsubw  vpsubsw  vpminsw  vpmaxsw\n";
    cout << "-------------------------------------------------------------------------\n";

    for (int i = 0; i < 16; i++)
    {
```

```
        cout << setw(2)  << i << ' ';
        cout << setw(8) << a.m_I16[i] << ' ';
        cout << setw(8) << b.m_I16[i] << ' ';
        cout << setw(8) << c[0].m_I16[i] << ' ';
        cout << setw(8) << c[1].m_I16[i] << ' ';
        cout << setw(8) << c[2].m_I16[i] << ' ';
        cout << setw(8) << c[3].m_I16[i] << ' ';
        cout << setw(8) << c[4].m_I16[i] << ' ';
        cout << setw(8) << c[5].m_I16[i] << '\n';
    }
}

void Avx2PackedMathI32(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b;
    alignas(32) YmmVal c[6];

    a.m_I32[0] = 64;        b.m_I32[0] = 4;
    a.m_I32[1] = 1024;      b.m_I32[1] = 5;
    a.m_I32[2] = -2048;     b.m_I32[2] = 2;
    a.m_I32[3] = 8192;      b.m_I32[3] = 5;
    a.m_I32[4] = -256;      b.m_I32[4] = 8;
    a.m_I32[5] = 4096;      b.m_I32[5] = 7;
    a.m_I32[6] = 16;        b.m_I32[6] = 3;
    a.m_I32[7] = 512;       b.m_I32[7] = 6;

    Avx2PackedMathI32_(a, b, c);

    cout << "\nРезультаты Avx2PackedMathI32\n\n";
    cout << " i      a      b   vpaddd   vpsubd  vpmulld  vpsllvd  vpsravd   vpabsd\n";
    cout << "-------------------------------------------------------------------------\n";

    for (int i = 0; i < 8; i++)
    {
        cout << setw(2) << i << ' ';
        cout << setw(6) << a.m_I32[i] << ' ';
        cout << setw(6) << b.m_I32[i] << ' ';
        cout << setw(8) << c[0].m_I32[i] << ' ';
        cout << setw(8) << c[1].m_I32[i] << ' ';
        cout << setw(8) << c[2].m_I32[i] << ' ';
        cout << setw(8) << c[3].m_I32[i] << ' ';
        cout << setw(8) << c[4].m_I32[i] << ' ';
        cout << setw(8) << c[5].m_I32[i] << '\n';
    }
}

int main()
{
    Avx2PackedMathI16();
    Avx2PackedMathI32();
    return 0;
}
```

```
;-----------------------------------------------
;              Ch10_01.asm
;-----------------------------------------------

; extern "C" void Avx2PackedMathI16_(const YmmVal& a, const YmmVal& b, YmmVal c[6])

        .code
Avx2PackedMathI16_ proc
; Присвоение значений a и b, которые должны быть правильно выровнены
        vmovdqa ymm0,ymmword ptr [rcx]      ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [rdx]      ;ymm1 = b

; Выполнение пакетных арифметических операций
        vpaddw ymm2,ymm0,ymm1               ;сложение
        vmovdqa ymmword ptr [r8],ymm2       ;сохранение результата

        vpaddsw ymm2,ymm0,ymm1              ;сложение со знаковым насыщением
        vmovdqa ymmword ptr [r8+32],ymm2    ;сохранение результата

        vpsubw ymm2,ymm0,ymm1               ;sub
        vmovdqa ymmword ptr [r8+64],ymm2    ;сохранение результата

        vpsubsw ymm2,ymm0,ymm1              ;sub with signed saturation
        vmovdqa ymmword ptr [r8+96],ymm2    ;сохранение результата

        vpminsw ymm2,ymm0,ymm1              ;signed minimums
        vmovdqa ymmword ptr [r8+128],ymm2   ;сохранение результата

        vpmaxsw ymm2,ymm0,ymm1              ;signed maximums
        vmovdqa ymmword ptr [r8+160],ymm2   ;сохранение результата

        vzeroupper
        ret
Avx2PackedMathI16_ endp

; extern "C" void Avx2PackedMathI32_(const YmmVal& a, const YmmVal& b, YmmVal c[6])

Avx2PackedMathI32_ proc
; Load values a and b, which must be properly aligned
        vmovdqa ymm0,ymmword ptr [rcx]      ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [rdx]      ;ymm1 = b

; Perform packed arithmetic operations
        vpaddd ymm2,ymm0,ymm1               ;сложение
        vmovdqa ymmword ptr [r8],ymm2       ;сохранение результата

        vpsubd ymm2,ymm0,ymm1               ;вычитание
        vmovdqa ymmword ptr [r8+32],ymm2    ;сохранение результата

        vpmulld ymm2,ymm0,ymm1              ;знаковое умножение (мл. 32 бита)
        vmovdqa ymmword ptr [r8+64],ymm2    ;сохранение результата

        vpsllvd ymm2,ymm0,ymm1              ;логический сдвиг влево
        vmovdqa ymmword ptr [r8+96],ymm2    ;сохранение результата

        vpsravd ymm2,ymm0,ymm1              ;арифметический сдвиг вправо
```

```
        vmovdqa ymmword ptr [r8+128],ymm2   ;сохранение результата

        vpabsd ymm2,ymm0                    ;абсолютное значение
        vmovdqa ymmword ptr [r8+160],ymm2   ;сохранение результата

        vzeroupper
        ret
Avx2PackedMathI32_ endp
        end
```

Функция C++ `Avx2PackedMathI16` содержит код, демонстрирующий арифметические действия над упакованными словами со знаком. Эта функция начинается с определения переменных `YmmVal` `a`, `b` и `c`. Обратите внимание, что для обеспечения выравнивания по 32-байтовой границе спецификатор C++ `alignas(32)` используется с каждым определением `YmmVal`. Затем элементы слова со знаком как `a`, так и `b` инициализируются тестовыми значениями. После инициализации переменной `Avx2PackedMathI16` вызывает функцию на языке ассемблера `Avx2PackedMathI16_`, которая выполняет несколько упакованных арифметических операций. Затем результаты передаются в `cout`. Далее следует функция C++ `Avx2PackedMathI32`. Структура этой функции аналогична `Avx2PackedMathI16`, с основным отличием в том, что она обрабатывает упакованные операнды в формате двойного слова.

Функция на языке ассемблера `Avx2PackedMathI16_` начинается с команды `vmovdqa ymm0,ymmword ptr [rcx]`, которая загружает `YmmVal a` в регистр YMM0. Затем команда `vmovdqa ymm1,ymmword ptr [rdx]` загружает `YmmVal b` в регистр YMM1. За ней следует `vpaddw ymm2,ymm0,ymm1`, которая выполняет сложение упакованных слов `a` и `b`. Потом команда `vmovdqa ymmword ptr [r8],ymm2` сохраняет упакованные суммы слов в `c[0]`. Оставшийся код на языке ассемблера в `Avx2PackedMathI16_` выполняет команды `vpaddsw,vpsubw,vpsubsw,vpminsw` и `vpmaxsw` для выполнения дополнительных арифметических операций. Подобно примерам исходного кода, которые вы видели в главе 9, функция `Avx2PackedMathI16_` использует команду `vzeroupper` перед своей командой `ret`. Это позволяет избежать потенциального снижения быстродействия, которое может возникнуть, когда процессор переходит от выполнения команд x86-AVX к командам x86-SSE, как описано в главе 8. Функция на языке ассемблера `Avx2PackedMathI32_` использует аналогичную структуру для выполнения часто используемых команд для работы с упакованными двойными словами, включая `vpaddd`, `vpsubd`, `vpmulld`, `vpsllvd`, `vpsravd` и `vpabsd`. Вот результаты для примера исходного кода `Ch10_01`:

```
Результаты Avx2PackedMathI16_

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

Результаты Avx2PackedMathI32

i       a      b    vpaddd   vpsubd   vpmulld  vpsllvd  vpsravd  vpabsd
-------------------------------------------------------------------------
0      64      4        68       60       256     1024        4      64
1    1024      5      1029     1019      5120    32768       32    1024
2   -2048      2     -2046    -2050     -4096    -8192     -512    2048
3    8192      5      8197     8187     40960   262144      256    8192
4    -256      8      -248     -264     -2048   -65536       -1     256
5    4096      7      4103     4089     28672   524288       32    4096
6      16      3        19       13        48      128        2      16
7     512      6       518      506      3072    32768        8     512
```

В системах, поддерживающих AVX2, большинство команд, применяемых в этом примере, можно использовать с различными упакованными целочисленными операндами шириной 256 бит. Например, команды `vpadd[b|q]` и `vpsub[b|q]` выполняют сложение и вычитание, используя 256-битные упакованные байтовые или четырехсловные операнды. Команды `vpaddsb` и `vpsubsb` выполняют сложение и вычитание с насыщением со знаком, используя упакованные байтовые операнды. Команды `vpmins[b|d]` и `vpmaxs[b|d]` вычисляют минимумы и максимумы упакованных чисел со знаком соответственно. Команды переменного битового сдвига `vpsllv[d|q]`, `vpsravd` и `vpsrlv[d|q]` – это новые команды AVX2. Эти команды недоступны в системах, поддерживающих только AVX.

10.1.2. Упаковка и распаковка

Следующий пример исходного кода показывает, как выполнять целочисленные операции упаковки и распаковки. Эти операции часто используются для уменьшения или увеличения размера упакованных целочисленных операндов. В данном примере также объясняется, как вернуть структуру по значению из функции на языке ассемблера. В листинге 10.2 показан исходный код примера `Ch10_02`.

Листинг 10.2. Пример Ch10_02

```
//-----------------------------------------------
//              Ch10_02.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
```

```
#include "YmmVal.h"

using namespace std;

struct alignas(32) YmmVal2
{
    YmmVal m_YmmVal0;
    YmmVal m_YmmVal1;
};

extern "C" YmmVal2 Avx2UnpackU32_U64_(const YmmVal& a, const YmmVal& b);
extern "C" void Avx2PackI32_I16_(const YmmVal& a, const YmmVal& b, YmmVal* c);

void Avx2UnpackU32_U64(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b;

    a.m_U32[0] = 0x00000000;  b.m_U32[0] = 0x88888888;
    a.m_U32[1] = 0x11111111;  b.m_U32[1] = 0x99999999;
    a.m_U32[2] = 0x22222222;  b.m_U32[2] = 0xaaaaaaaa;
    a.m_U32[3] = 0x33333333;  b.m_U32[3] = 0xbbbbbbbb;

    a.m_U32[4] = 0x44444444;  b.m_U32[4] = 0xcccccccc;
    a.m_U32[5] = 0x55555555;  b.m_U32[5] = 0xdddddddd;
    a.m_U32[6] = 0x66666666;  b.m_U32[6] = 0xeeeeeeee;
    a.m_U32[7] = 0x77777777;  b.m_U32[7] = 0xffffffff;

    YmmVal2 c = Avx2UnpackU32_U64_(a, b);

    cout << "\nРезультаты Avx2UnpackU32_U64\n\n";

    cout << "a lo           " << a.ToStringX32(0) << '\n';
    cout << "b lo           " << b.ToStringX32(0) << '\n';
    cout << '\n';

    cout << "a hi           " << a.ToStringX32(1) << '\n';
    cout << "b hi           " << b.ToStringX32(1) << '\n';

    cout << "\результат nvpunpckldq\n";
    cout << "c.m_YmmVal0 lo " << c.m_YmmVal0.ToStringX64(0) << '\n';
    cout << "c.m_YmmVal0 hi " << c.m_YmmVal0.ToStringX64(1) << '\n';

    cout << "\результат nvpunpckhdq\n";
    cout << "c.m_YmmVal1 lo " << c.m_YmmVal1.ToStringX64(0) << '\n';
    cout << "c.m_YmmVal1 hi " << c.m_YmmVal1.ToStringX64(1) << '\n';
}

void Avx2PackI32_I16(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b;
    alignas(32) YmmVal c;

    a.m_I32[0] = 10;          b.m_I32[0] = 32768;
    a.m_I32[1] = -200000;     b.m_I32[1] = 6500;
```

```
    a.m_I32[2] = 300000;      b.m_I32[2] = 42000;
    a.m_I32[3] = -4000;       b.m_I32[3] = -68000;

    a.m_I32[4] = 9000;        b.m_I32[4] = 25000;
    a.m_I32[5] = 80000;       b.m_I32[5] = 500000;
    a.m_I32[6] = 200;         b.m_I32[6] = -7000;
    a.m_I32[7] = -32769;      b.m_I32[7] = 12500;

    Avx2PackI32_I16_(a, b, &c);

    cout << "\nРезультаты Avx2PackI32_I16\n\n";

    cout << "a lo " << a.ToStringI32(0) << '\n';
    cout << "a hi " << a.ToStringI32(1) << '\n';
    cout << '\n';

    cout << "b lo " << b.ToStringI32(0) << '\n';
    cout << "b hi " << b.ToStringI32(1) << '\n';
    cout << '\n';

    cout << "c lo " << c.ToStringI16(0) << '\n';
    cout << "c hi " << c.ToStringI16(1) << '\n';
    cout << '\n';
}

int main()
{
    Avx2UnpackU32_U64();
    Avx2PackI32_I16();
    return 0;
}

;-----------------------------------------------
;               Ch10_02.asm
;-----------------------------------------------

; extern "C" YmmVal2 Avx2UnpackU32_U64_(const YmmVal& a, const YmmVal& b);

        .code
Avx2UnpackU32_U64_ proc

; Загрузка значений аргументов
        vmovdqa ymm0,ymmword ptr [rdx]      ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [r8]       ;ymm1 = b

; Распаковка двойного слова в квадрослово
        vpunpckldq ymm2,ymm0,ymm1           ;распаковка младшего двойного слова
        vpunpckhdq ymm3,ymm0,ymm1           ;распаковка старшего двойного слова

; Сохранение результата в буфер YmmVal2
        vmovdqa ymmword ptr [rcx],ymm2      ;сохранение младшего результата
        vmovdqa ymmword ptr [rcx+32],ymm3   ;сохранение старшего результата

        mov rax,rcx                         ;rax = указатель на YmmVal2

        vzeroupper
        ret
```

```
Avx2UnpackU32_U64_ endp

; extern "C" void Avx2PackI32_I16_(const YmmVal& a, const YmmVal& b, YmmVal* c);

Avx2PackI32_I16_ proc

; Загрузка значений аргументов
        vmovdqa ymm0,ymmword ptr [rcx]      ;ymm0 = a
        vmovdqa ymm1,ymmword ptr [rdx]      ;ymm1 = b

; Упаковка двойного слова в слово с насыщением со знаком
        vpackssdw ymm2,ymm0,ymm1            ;ymm2 = упакованные слова
        vmovdqa ymmword ptr [r8],ymm2       ;сохранение результата

        vzeroupper
        ret
Avx2PackI32_I16_ endp

Foo1_ proc
        ret
Foo1_ endp
        end
```

Код C++ в листинге 10.2 начинается с объявления структуры с именем `YmmVal2`. Эта структура содержит два члена `YmmVal`: `m_YmmVal0` и `m_YmmVal1`. Обратите внимание, что спецификатор `alignas(32)` используется сразу после ключевого слова `struct`. Применение этого спецификатора гарантирует, что все экземпляры `YmmVal2` выровнены по 32-байтовой границе, включая временные экземпляры, созданные компилятором. Подробнее об этом будет сказано чуть позже. Функция на языке ассемблера `Avx2UnpackU32_U64_`, объявление которой следует ниже, возвращает экземпляр `YmmVal2` по значению.

Функция C++ `AvxUnpackU32_U64` начинается с инициализации элементов `YmmVal` – переменных `a` и `b` в формате двойных слов без знака. Следующую инициализацию переменной выполняет оператор `YmmVal2 c=Avx2UnpackU32_U64_(a,b)`, который вызывает функцию на языке ассемблера `Avx2UnpackU32_U64_` для распаковки элементов `a` и `b` из двойных слов в четверные слова. В отличие от предыдущих примеров, `Avx2UnpackU32_U64_` возвращает результат `YmmVal2` по значению. Прежде чем продолжить, важно отметить, что в большинстве случаев возврат определенной пользователем структуры, такой как `YmmVal2`, по значению менее эффективен, чем передача аргумента указателя переменной типа `YmmVal2`. Функция `Avx2UnpackU32_U64_` использует возврат по значению в основном в демонстрационных целях и для разъяснения протоколов соглашения о вызовах Visual C++, которые функция на языке ассемблера должна соблюдать, когда требуется возврат структуры по значению. Остальные операторы в `AvxUnpackU32_U64` передают результаты из `Avx2UnpackU32_U64_` в `cout`.

Следом за `AvxUnpackU32_U64` идет функция C++ `Avx2PackI32_I16`. Эта функция инициализирует элементы `YmmVal` – переменные `a` и `b` в формате двойного слова со знаком. Эти значения будут уменьшены до упакованных слов. После инициализации переменной `YmmVal Avx2PackI32_I16` вызывает функцию на языке ассемблера `Avx2PackI32_I16_` для выполнения вышеупомянутого уменьшения размера. Затем результаты передаются в `cout`.

Соглашение о вызовах, которое Visual C++ использует для функций, возвращающих структуру по значению, несколько отличается от обычного соглашения о вызовах. После входа в функцию на языке ассемблера `Avx2UnpackU32_U64_` регистр RCX указывает на временный буфер, где `Avx2UnpackU32_U64_` должен хранить свой возвращаемый результат `YmmVal2`. Важно отметить, что этот буфер не обязательно совпадает с местом в памяти, которое занимает целевая переменная `YmmVal2` в операторе C++, вызвавшем `Avx2UnpackU32_U64_`. Чтобы реализовать вычисление выражений и перегрузку операторов, компилятор C++ часто генерирует код, который выделяет временные переменные (или *rvalue*) для хранения промежуточных результатов. Значение *rvalue*, которое необходимо сохранить, в конечном итоге копируется в именованную переменную (или *lvalue*) с использованием оператора присваивания по умолчанию или перегруженного оператора присваивания. Эта операция копирования является причиной того, почему возврат структуры по значению обычно происходит медленнее, чем передача аргумента-указателя. Спецификатор `alignas(32)`, который используется в объявлении структуры `YmmVal2`, предписывает компилятору Visual C++ выровнять все переменные типа `YmmVal2`, *включая rvalue*, по 32-байтовой границе.

Если содержание предыдущего абзаца кажется вам непонятным, не волнуйтесь. Выделение временного пространства для хранения структур, возвращаемых по значению, автоматически обрабатывается компилятором C++. Более важно понимать следующие требования соглашения о вызовах Visual C++, которые должны соблюдаться любой функцией, возвращающей большую структуру (любую структуру, размер которой превышает 8 байт) по значению:

- функция, которая вызывает другую функцию, возвращающую большую структуру по значению, должна выделить место для хранения возвращаемой структуры. Указатель на это пространство памяти должен быть передан вызываемой функции в регистре RCX;
- обычные регистры аргументов соглашения о вызовах «сдвинуты вправо» на единицу. Это означает, что первые три аргумента передаются с использованием регистров RDX/XMM1, R8/XMM2 и R9/XMM3. Все оставшиеся аргументы передаются в стек;
- перед возвратом вызываемая функция должна загрузить в регистр RAX указатель на возвращаемую структуру.

Если размер структуры, возвращаемой по значению, меньше или равен 8 байтам, он должен быть возвращен в регистре RAX. В этих ситуациях используются обычные регистры аргументов соглашения о вызовах.

Возвращаясь к коду, первая команда функции `Avx2UnpackU32_U64_` использует команду `vmovdqa ymm0,ymmword ptr [rdx]` для загрузки `YmmVal a` (первого аргумента функции) в регистр YMM0. Следующая команда `vmovdqa ymm1,ymmword ptr [r8]` загружает `YmmVal b` (второй аргумент функции) в регистр YMM1. Следующие две команды, `vpunpckldq ymm2,ymm0,ymm1` и `vpunpckhdq ymm3,ymm0,ymm1`, распаковывают двойные слова в четверные слова, как показано на рис. 10.1. Затем результаты сохраняются в буфер `YmmVal2`, на который указывает RCX, с использованием двух команд `vmovdqa`. Обратите внимание, что здесь потребуются две команды `vmovdqu`, если структура `YmmVal2` была объявлена без спецификатора `alignas(32)`. Как упоминалось ранее, соглашение о вызовах Visual C++ требует,

чтобы любая функция, возвращающая структуру по значению, загружала копию указателя на буфер структуры в регистр RAX перед возвратом. Команда `mov rax,rcx` выполняет это требование (напомним, что RCX содержит указатель на буфер структуры).

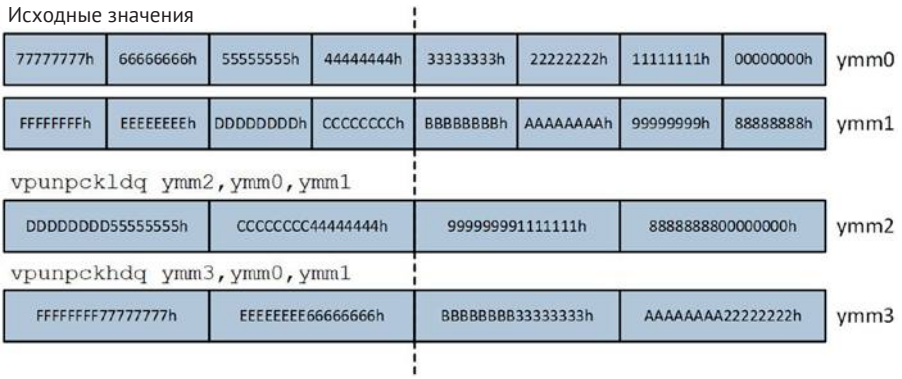


Рис. 10.1. Выполнение команд `vpunpckldq` и `vpunpckhdq`

Функция на языке ассемблера `Avx2PackI32_I16_` демонстрирует использование команды `vpackssdw` (упаковка со знаковым насыщением). В этой функции команда `vpackssdw ymm2,ymm0,ymm1` преобразует 16 целочисленных двойных слов в регистрах YMM0 и YMM1 в однословные целые числа, используя насыщение со знаком. Затем она сохраняет 16 однословных целых чисел в регистре YMM2. Рисунок 10.2 иллюстрирует выполнение этой команды. x86-AVX также содержит команду `vpacksswb`, которая выполняет сокращение слова со знаком до байтового размера. Команды `vpackus[dw|wb]` могут использоваться для сжатия упакованных целых чисел без знака.

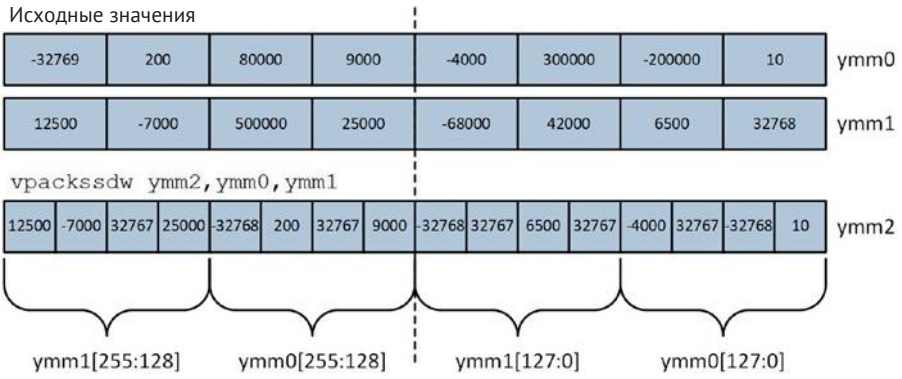


Рис. 10.2. Выполнение команды `vpackssdw`

Обратите внимание, что на рис. 10.1 и 10.2 команды `vpunpckldq`, `vpunpckhdq` и `vpackssdw` выполняют свои операции, используя две 128-битные независимые дорожки, как описано в главе 4. Вот результаты выполнения примера исходного кода `Ch10_02`:

```
Results for Avx2UnpackU32_U64

a lo                  00000000       11111111 |      22222222       33333333
b lo                  88888888       99999999 |      AAAAAAAA       BBBBBBBB
a hi                  44444444       55555555 |      66666666       77777777
b hi                  CCCCCCCC       DDDDDDDD |      EEEEEEEE       FFFFFFFF

vpunpckldq result
c.m_YmmVal0 lo               8888888800000000 |             9999999911111111
c.m_YmmVal0 hi               CCCCCCCC44444444 |             DDDDDDDD55555555

vpunpckhdq result
c.m_YmmVal1 lo               AAAAAAAA22222222 |             BBBBBBBB33333333
c.m_YmmVal1 hi               EEEEEEEE66666666 |             FFFFFFFF77777777

Results for Avx2PackI32_I16

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
```

10.1.3. Увеличение размера

В главе 7 вы узнали, как использовать команды `vpunpckl[bw|dw]` и `vpunpckh[bw|wd]` для увеличения размера упакованных целых чисел (см. примеры исходного кода `Ch07_05`, `Ch07_06` и `Ch07_08`). Следующий пример исходного кода, `Ch10_03`, демонстрирует применение команд `vpmovzx[bw|bd]` и `vpmovsx[wd|wq]` для увеличения размера упакованных целых чисел с помощью расширения знаковым битом или нулем. В листинге 10.3 показан исходный код примера `Ch10_03`.

Листинг 10.3. Пример Ch10_03

```
//-----------------------------------------------
//              Ch10_03.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <string>
#include "YmmVal.h"

using namespace std;

extern "C" void Avx2ZeroExtU8_U16_(YmmVal*a, YmmVal b[2]);
extern "C" void Avx2ZeroExtU8_U32_(YmmVal*a, YmmVal b[4]);
extern "C" void Avx2SignExtI16_I32_(YmmVal*a, YmmVal b[2]);
extern "C" void Avx2SignExtI16_I64_(YmmVal*a, YmmVal b[4]);

const string c_Line(80, '-');
```

```
void Avx2ZeroExtU8_U16(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b[2];

    for (int i = 0; i < 32; i++)
        a.m_U8[i] = (uint8_t)(i * 8);

    Avx2ZeroExtU8_U16_(&a, b);

    cout << "\nРезультаты Avx2ZeroExtU8_U16_\n";
    cout << c_Line << '\n';

    cout << "a (0:15):   " << a.ToStringU8(0) << '\n';
    cout << "a (16:31):  " << a.ToStringU8(1) << '\n';
    cout << '\n';
    cout << "b (0:7):    " << b[0].ToStringU16(0) << '\n';
    cout << "b (8:15):   " << b[0].ToStringU16(1) << '\n';
    cout << "b (16:23):  " << b[1].ToStringU16(0) << '\n';
    cout << "b (24:31):  " << b[1].ToStringU16(1) << '\n';
}

void Avx2ZeroExtU8_U32(void)
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b[4];

    for (int i = 0; i < 32; i++)
        a.m_U8[i] = (uint8_t)(255 - i * 8);

    Avx2ZeroExtU8_U32_(&a, b);

    cout << "\nРезультаты Avx2ZeroExtU8_U32_\n";
    cout << c_Line << '\n';

    cout << "a (0:15):   " << a.ToStringU8(0) << '\n';
    cout << "a (16:31):  " << a.ToStringU8(1) << '\n';
    cout << '\n';
    cout << "b (0:3):    " << b[0].ToStringU32(0) << '\n';
    cout << "b (4:7):    " << b[0].ToStringU32(1) << '\n';
    cout << "b (8:11):   " << b[1].ToStringU32(0) << '\n';
    cout << "b (12:15):  " << b[1].ToStringU32(1) << '\n';
    cout << "b (16:19):  " << b[2].ToStringU32(0) << '\n';
    cout << "b (20:23):  " << b[2].ToStringU32(1) << '\n';
    cout << "b (24:27):  " << b[3].ToStringU32(0) << '\n';
    cout << "b (28:31):  " << b[3].ToStringU32(1) << '\n';
}

void Avx2SignExtI16_I32()
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b[2];

    for (int i = 0; i < 16; i++)
        a.m_I16[i] = (int16_t)(-32768 + i * 4000);

    Avx2SignExtI16_I32_(&a, b);

    cout << "\nРезультаты Avx2SignExtI16_I32_\n";
```

```
    cout << c_Line << '\n';

    cout << "a (0:7):    " << a.ToStringI16(0) << '\n';
    cout << "a (8:15):   " << a.ToStringI16(1) << '\n';
    cout << '\n';
    cout << "b (0:3):    " << b[0].ToStringI32(0) << '\n';
    cout << "b (4:7):    " << b[0].ToStringI32(1) << '\n';
    cout << "b (8:11):   " << b[1].ToStringI32(0) << '\n';
    cout << "b (12:15):  " << b[1].ToStringI32(1) << '\n';
}

void Avx2SignExtI16_I64()
{
    alignas(32) YmmVal a;
    alignas(32) YmmVal b[4];

    for (int i = 0; i < 16; i++)
        a.m_I16[i] = (int16_t)(32767 - i * 4000);

    Avx2SignExtI16_I64_(&a, b);

    cout << "\nРезультаты Avx2SignExtI16_I64_\n";
    cout << c_Line << '\n';

    cout << "a (0:7):    " << a.ToStringI16(0) << '\n';
    cout << "a (8:15):   " << a.ToStringI16(1) << '\n';
    cout << '\n';
    cout << "b (0:1):    " << b[0].ToStringI64(0) << '\n';
    cout << "b (2:3):    " << b[0].ToStringI64(1) << '\n';
    cout << "b (4:5):    " << b[1].ToStringI64(0) << '\n';
    cout << "b (6:7):    " << b[1].ToStringI64(1) << '\n';
    cout << "b (8:9):    " << b[2].ToStringI64(0) << '\n';
    cout << "b (10:11):  " << b[2].ToStringI64(1) << '\n';
    cout << "b (12:13):  " << b[3].ToStringI64(0) << '\n';
    cout << "b (14:15):  " << b[3].ToStringI64(1) << '\n';
}

int main()
{
    Avx2ZeroExtU8_U16();
    Avx2ZeroExtU8_U32();
    Avx2SignExtI16_I32();
    Avx2SignExtI16_I64();
    return 0;
}

;-----------------------------------------------
;               Ch10_03.asm
;-----------------------------------------------

; extern "C" void Avx2ZeroExtU8_U16_(YmmVal*a, YmmVal b[2]);

        .code
Avx2ZeroExtU8_U16_ proc
        vpmovzxbw ymm0,xmmword ptr [rcx]        ;расш. нулем a[0]-a[15]
        vpmovzxbw ymm1,xmmword ptr [rcx+16]     ;расш. нулем a[16]-a[31]

        vmovdqa ymmword ptr [rdx],ymm0          ;сохранение результата
        vmovdqa ymmword ptr [rdx+32],ymm1

        vzeroupper
        ret
```

```
Avx2ZeroExtU8_U16_ endp

; extern "C" void Avx2ZeroExtU8_U32_(YmmVal*a, YmmVal b[4]);

Avx2ZeroExtU8_U32_ proc
        vpmovzxbd ymm0,qword ptr [rcx]          ;расш. нулем a[0]-a[7]
        vpmovzxbd ymm1,qword ptr [rcx+8]        ;расш. нулем a[8]-a[15]
        vpmovzxbd ymm2,qword ptr [rcx+16]       ;расш. нулем a[16]-a[23]
        vpmovzxbd ymm3,qword ptr [rcx+24]       ;расш. нулем a[24]-a[31]

        vmovdqa ymmword ptr [rdx],ymm0          ;сохранение результата
        vmovdqa ymmword ptr [rdx+32],ymm1
        vmovdqa ymmword ptr [rdx+64],ymm2
        vmovdqa ymmword ptr [rdx+96],ymm3

        vzeroupper
        ret
Avx2ZeroExtU8_U32_ endp

; extern "C" void Avx2SignExtI16_I32_(YmmVal*a, YmmVal b[2])

Avx2SignExtI16_I32_ proc
        vpmovsxwd ymm0,xmmword ptr [rcx]        ;расш. знаком a[0]-a[7]
        vpmovsxwd ymm1,xmmword ptr [rcx+16]     ;расш. знаком a[8]-a[15]

        vmovdqa ymmword ptr [rdx],ymm0          ;сохранение результата
        vmovdqa ymmword ptr [rdx+32],ymm1

        vzeroupper
        ret
Avx2SignExtI16_I32_ endp

; extern "C" void Avx2SignExtI16_I64_(YmmVal*a, YmmVal b[4])

Avx2SignExtI16_I64_ proc
        vpmovsxwq ymm0,qword ptr [rcx]          ;расш. знаком a[0]-a[3]
        vpmovsxwq ymm1,qword ptr [rcx+8]        ;расш. знаком a[4]-a[7]
        vpmovsxwq ymm2,qword ptr [rcx+16]       ;расш. знаком a[8]-a[11]
        vpmovsxwq ymm3,qword ptr [rcx+24]       ;расш. знаком a[12]-a[15]

        vmovdqa ymmword ptr [rdx],ymm0          ;сохранение результата
        vmovdqa ymmword ptr [rdx+32],ymm1
        vmovdqa ymmword ptr [rdx+64],ymm2
        vmovdqa ymmword ptr [rdx+96],ymm3

        vzeroupper
        ret
Avx2SignExtI16_I64_ endp
        end
```

Код C++ в листинге 10.3 содержит четыре функции, которые инициализируют тестовые примеры для различных операций по увеличению размера упакованных значений. Первая функция, `Avx2ZeroExtU8_U16`, начинается с инициализации беззнаковых байтовых элементов `YmmVal a`. Затем она вызывает функцию на языке ассемблера `Avx2ZeroExtU8_U16_` для расширения упакованных байтов без знака в упакованные слова без знака. Функция `Avx2ZeroExtU8_U32` выполняет аналогичный набор инициализаций, чтобы продемонстрировать расширение упакованного беззнакового байта в упакованное беззнаковое двойное слово. Функции `Avx2SignExtI16_I32` и `Avx2SignExtI16_I64` инициализируют тестовые при-

меры для расширения упакованного слова со знаком до упакованного двойного слова со знаком и упакованного квадрослова со знаком.

Первая команда в функции на языке ассемблера `Avx2ZeroExtU8_U16_`, `vpmovzxbw ymm0,xmmword ptr [rcx]` загружает и расширяет до нуля 16 младших байт `YmmVal a` (на которые указывает регистр RCX) и сохраняет эти значения в регистре YMM0. Следующая команда `vpmovzxbw ymm1,xmmword ptr [rcx+16]` выполняет ту же операцию, используя 16 старших байт `YmmVal a`. Затем функция `Avx2ZeroExtU8_U16_` использует две команды `vmovdqa` для сохранения результатов увеличенного размера.

Функция на языке ассемблера `Avx2ZeroExtU8_U32_` выполняет преобразование упакованного байта в двойное слово. Первая команда, `vpmovzxbd ymm0,qword ptr [rcx]`, загружает и расширяет нулями 8 младших байт `YmmVal a` до двойных слов и сохраняет эти значения в регистре YMM0. Три последующие команды `vpmovzxbd` расширяют оставшиеся байтовые значения в `YmmVal a`. Затем результаты сохраняются с помощью серии команд `vmovdqa`. При работе с беззнаковыми 8-битными значениями иногда (в зависимости от алгоритма) более целесообразно использовать команду `vpmovzxbd` для преобразования упакованного байта в упакованное двойное слово вместо семантически эквивалентной последовательности команд `vpunpckl[bw|dw]` и `vpunpckh[bw|dw]`. Вы увидите пример этого в главе 14.

Функции на языке ассемблера `Avx2SignExtI16_I32_` и `Avx2SignExtI16_I64_` демонстрируют, как использовать команды `vpmovsxwd` и `vpmovsxwq` соответственно. Эти команды расширяют знаковым битом упакованные целочисленные слова до двойных и четверных слов. x86-AVX также включает команды присвоения упакованных чисел с расширением знаковым битом `vpmovsx[bw|bd|bq]` и `vpmovsxdq`. Вот результат выполнения примера исходного кода `Ch10_03`:

```
Results for Avx2ZeroExtU8_U16_
-------------------------------------------------------------------------------
a (0:15):      0   8  16  24  32  40  48  56   |  64  72  80  88  96 104 112 120
a (16:31):   128 136 144 152 160 168 176 184   | 192 200 208 216 224 232 240 248
b (0:7):           0       8      16      24   |      32      40      48      56
b (8:15):         64      72      80      88   |      96     104     112     120
b (16:23):       128     136     144     152   |     160     168     176     184
b (24:31):       192     200     208     216   |     224     232     240     248

Results for Avx2ZeroExtU8_U32_
-------------------------------------------------------------------------------
a (0:15):    255 247 239 231 223 215 207 199   | 191 183 175 167 159 151 143 135
a (16:31):   127 119 111 103  95  87  79  71   |  63  55  47  39  31  23  15   7
b (0:3):                 255             247   |             239             231
b (4:7):                 223             215   |             207             199
b (8:11):                191             183   |             175             167
b (12:15):               159             151   |             143             135
b (16:19):               127             119   |             111             103
b (20:23):                95              87   |              79              71
b (24:27):                63              55   |              47              39
b (28:31):                31              23   |              15               7

Results for Avx2SignExtI16_I32_
-------------------------------------------------------------------------------
a (0:7):      -32768  -28768  -24768 - 20768   |  -16768  -12768   -8768   -4768
```

```
a (8:15):       -768    3232    7232    1232   |   15232   19232   23232   27232
b (0:3):              -32768          -28768   |           -24768           -20768
b (4:7):              -16768          -12768   |            -8768            -4768
b (8:11):               -768            3232   |             7232            11232
b (12:15):             15232           19232   |            23232            27232

Results for Avx2SignExtI16_I64_
--------------------------------------------------------------------------------
a (0:7):       32767   28767   24767   20767   |   16767   12767    8767    4767
a (8:15):        767   -3233   -7233  -11233   |  -15233  -19233  -23233  -27233
b (0:1):                               32767   |                           28767
b (2:3):                               24767   |                           20767
b (4:5):                               16767   |                           12767
b (6:7):                                8767   |                            4767
b (8:9):                                 767   |                           -3233
b (10:11):                              -7233   |                          -11233
b (12:13):                             -15233   |                          -19233
b (14:15):                             -23233   |                          -27233
```

10.2. Обработка изображений с упакованными целочисленными пикселями

В главе 7 вы узнали, как использовать набор команд AVX для выполнения некоторых распространенных операций обработки изображений с использованием 128-битных упакованных целочисленных операндов без знака. Примеры исходного кода в этом разделе демонстрируют дополнительные методы обработки изображений с использованием команд AXV2 с 256-битными упакованными целочисленными операндами без знака. Первый пример исходного кода демонстрирует ограничение значений интенсивности пикселей изображения в градациях серого. Далее следует пример, в котором показано определение минимального и максимального значений интенсивности пикселей изображения формата RGB. В последнем примере исходного кода используется набор команд AVX2 для преобразования изображения из формата RGB в оттенки серого.

10.2.1. Усечение пикселей

Усечение пикселей – это метод обработки изображения, который ограничивает значения интенсивности каждого пикселя изображения двумя пороговыми пределами. Этот метод часто используется для уменьшения динамического диапазона изображения за счет устранения очень темных и светлых пикселей. Пример исходного кода `Ch10_04` показывает, как использовать набор команд AVX2 для усечения значений пикселей 8-битного изображения в градациях серого. В листинге 10.4 показан исходный код примера `Ch10_04` на языках C++ и ассемблера.

Листинг 10.4. Пример Ch10_04

```
//-----------------------------------------------
//              Ch10_04.cpp
//-----------------------------------------------

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

```
#include <random>
#include <memory.h>
#include <limits>
#include "Ch10_04.h"
#include "AlignedMem.h"

using namespace std;

void Init(uint8_t* x, uint64_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {0, 255};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (uint8_t)ui_dist(rng);
}

bool Avx2ClipPixelsCpp(ClipData* cd)
{
    uint8_t* src = cd->m_Src;
    uint8_t* des = cd->m_Des;
    uint64_t num_pixels = cd->m_NumPixels;

    if (num_pixels == 0 || (num_pixels % 32) != 0)
        return false;

    if (!AlignedMem::IsAligned(src, 32) || !AlignedMem::IsAligned(des, 32))
        return false;

    uint64_t num_clipped_pixels = 0;
    uint8_t thresh_lo = cd->m_ThreshLo;
    uint8_t thresh_hi = cd->m_ThreshHi;

    for (uint64_t i = 0; i < num_pixels; i++)
    {
        uint8_t 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];
    }

    cd->m_NumClippedPixels = num_clipped_pixels;
    return true;
}

void Avx2ClipPixels(void)
```

```
{
    const uint8_t thresh_lo = 10;
    const uint8_t thresh_hi = 245;
    const uint64_t num_pixels = 4 * 1024 * 1024;

    AlignedArray<uint8_t> src(num_pixels, 32);
    AlignedArray<uint8_t> des1(num_pixels, 32);
    AlignedArray<uint8_t> des2(num_pixels, 32);

    Init(src.Data(), num_pixels, 157);

    ClipData cd1;
    ClipData cd2;

    cd1.m_Src = src.Data();
    cd1.m_Des = des1.Data();
    cd1.m_NumPixels = num_pixels;
    cd1.m_NumClippedPixels = numeric_limits<uint64_t>::max();
    cd1.m_ThreshLo = thresh_lo;
    cd1.m_ThreshHi = thresh_hi;

    cd2.m_Src = src.Data();
    cd2.m_Des = des2.Data();
    cd2.m_NumPixels = num_pixels;
    cd2.m_NumClippedPixels = numeric_limits<uint64_t>::max();
    cd2.m_ThreshLo = thresh_lo;
    cd2.m_ThreshHi = thresh_hi;

    Avx2ClipPixelsCpp(&cd1);
    Avx2ClipPixels_(&cd2);

    cout << "\nРезультаты Avx2ClipPixels\n";
    cout << "  cd1.m_NumClippedPixels1: " << cd1.m_NumClippedPixels << '\n';
    cout << "  cd2.m_NumClippedPixels2: " << cd2.m_NumClippedPixels << '\n';

    if (cd1.m_NumClippedPixels != cd2.m_NumClippedPixels)
        cout << "  Ошибка сравнения NumClippedPixels\n";

    if (memcmp(des1.Data(), des2.Data(), num_pixels) == 0)
        cout << "  Сравнение буферной памяти пикселей пройдено\n";
    else
        cout << "  Сравнение буферной памяти пикселей не пройдено\n";
}

int main(void)
{
    Avx2ClipPixels();
    Avx2ClipPixels_BM();
    return 0;
}

;-----------------------------------------------
;              Ch10_04.asm
;-----------------------------------------------

; Следующая структура должна совпадать со структурой, объявленной в файле .h
ClipData            struct
```

```
Src                 qword ?             ;указатель буфера источника
Des                 qword ?             ;указатель буфера приемника
NumPixels           qword ?             ;количество пикселей
NumClippedPixels    qword ?             ;количество усеченных пикселей
ThreshLo            byte ?              ;нижний порог
ThreshHi            byte ?              ;верхний порог
ClipData            ends

; extern "C" bool Avx2ClipPixels_(ClipData* cd)

            .code
Avx2ClipPixels_ proc

; Загрузка и проверка аргументов
        xor eax,eax                         ;возвращаемый код ошибки
        xor r8d,r8d                         ;r8 = количество усеченных пикселей

        mov rdx,[rcx+ClipData.NumPixels]    ;rdx = num_pixels
        or rdx,rdx
        jz Done                             ;переход, если num_pixels равен нулю
        test rdx,1fh
        jnz Done                            ;переход, если num_pixels % 32 != 0

        mov r10,[rcx+ClipData.Src]          ;r10 = Src
        test r10,1fh
        jnz Done                            ;переход, если Src не выровнен

        mov r11,[rcx+ClipData.Des]          ;r11 = Des
        test r11,1fh
        jnz Done                            ;переход, если Des не выровнен

; Создание упакованных значений thresh_lo и thresh_hi
        vpbroadcastb ymm4,[rcx+ClipData.ThreshLo]   ;ymm4 = упакованных thresh_lo
        vpbroadcastb ymm5,[rcx+ClipData.ThreshHi]   ;ymm5 = упакованных thresh_hi

; Усечение пикселей до пороговых значений
@@:     vmovdqa ymm0,ymmword ptr [r10]      ;ymm0 = 32 пикселя
        vpmaxub ymm1,ymm0,ymm4              ;усечение до thresh_lo
        vpminub ymm2,ymm1,ymm5              ;усечение до thresh_hi
        vmovdqa ymmword ptr [r11],ymm2      ;сохранение усеченного значения

; Подсчет количества усеченных пикселей
        vpcmpeqb ymm3,ymm2,ymm0             ;сравнение усеченных пикселей с оригиналом
        vpmovmskb eax,ymm3                  ;eax = маска неусеченных пикселей
        not eax                             ;eax = маска усеченных пикселей
        popcnt eax,eax                      ;eax = количество усеченных пикселей
        add r8,rax                          ;обновление счетчика усеченных пикселей

; Обновление указателей и счетчика цикла
        add r10,32                          ;обновление указателя источника
        add r11,32                          ;обновление указателя приемника
        sub rdx,32                          ;обновление счетчика цикла
        jnz @B                              ;повторение до завершения

        mov eax,1                           ;возвращаемый код успешного выполнения
```

```
; Сохранение num_clipped_pixels

Done:   mov [rcx+ClipData.NumClippedPixels],r8  ;сохранение num_clipped_pixels
        vzeroupper
        ret

Avx2ClipPixels_ endp
        end
```

Код C++ начинается с объявления структуры с именем `ClipData`. Эта структура и ее эквивалент на языке ассемблера используются для хранения данных алгоритма усечения пикселей. После объявлений функций в заголовочном файле `Ch10_04.h` следует определение функции C++ с именем `Init`. Эта функция инициализирует элементы массива `uint8_t`, используя случайные значения, которые имитируют значения пикселей изображения в градациях серого. Функция `Avx2ClipPixelCpp` – это реализация алгоритма усечения пикселей на C++. Эта функция начинается с проверки правильности размера `num_pixels` и делимости на 32. Ограничение алгоритма изображениями, которые содержат четное число, кратное 32 пикселям, не так негибко, как может показаться. Большинство изображений с цифровых камер имеют размер, кратный 64 пикселям, из-за требований к обработке алгоритмов сжатия JPEG. После проверки `num_pixels` исходный и целевой пиксельные буферы проверяются на правильное выравнивание.

Процедура, используемая в `Avx2ClipPixelCpp` для выполнения усечения пикселей, проста. Простой цикл `for` проверяет каждый пиксельный элемент в буфере исходного изображения. Если значение интенсивности пикселя в буфере исходного изображения оказывается ниже `thresh_lo` или выше `thresh_hi`, значение соответствующего порогового предела сохраняется в целевом буфере. Пиксели исходного изображения, значения интенсивности которых лежат между двумя пороговыми пределами, копируются в буфер пикселей назначения без изменений. Цикл обработки в `Avx2ClipPixelCpp` также подсчитывает количество усеченных пикселей для сравнения с версией алгоритма на языке ассемблера.

Функция `Avx2ClipPixels` использует класс шаблона C++ `AlignedArray` для выделения требуемых буферов под пиксели изображения и управления ими (описание этого класса см. в главе 7). После инициализации пиксельного буфера исходного изображения `Avx2ClipPixels` заполняет два экземпляра `ClipData` (`cd1` и `cd2`) для использования функциями усечения пикселей `Avx2ClipPixelsCpp` и `Avx2ClipPixels_`. Затем она вызывает эти функции и сравнивает результаты на предмет любых расхождений.

В верхней части кода на языке ассемблера находится объявление структуры данных `ClipPixel`, которая семантически эквивалентна своему аналогу в C++. Функция `Avx2ClipPixels_` начинает свое выполнение с проверки `num_pixels` на предмет размера и делимости на 32. Затем она проверяет исходный и целевой буферы пикселей на предмет правильного выравнивания. После проверки аргумента `Avx2ClipPixels_` использует две команды `vpbroadcastb` для создания упакованных версий пороговых предельных значений `thresh_lo` и `thresh_hi` в регистрах YMM4 и YMM5 соответственно. Во время каждой итерации цикла обработки команда `vmovdqa ymm0,ymmword ptr [r10]` загружает 32 пиксельных значения из буфера исходного изображения в регистр YMM0. Следующая ко-

манда `vpmaxub ymm1,ymm0,ymm4` обрезает значения пикселей в YMM0 до `thresh_lo`. За ней следует команда `vpminub ymm2,ymm1,ymm5`, которая обрезает значения пикселей до `thresh_hi`. Затем команда `vmovdqa ymmword ptr [r11], ymm2` сохраняет усеченные значения интенсивности пикселей в приемный буфер пикселей изображения.

Функция `Avx2ClipPixels_` подсчитывает количество усеченных пикселей, используя простую последовательность команд. Команда `vpcmpeqb ymm3,ymm2,ymm0` сравнивает исходные значения пикселей в YMM0 с усеченными значениями пикселей в YMM2 на предмет совпадения. Каждому байтовому элементу в YMM3 присваивается значение `0xff`, если значения интенсивности исходного и обрезанного пикселей равны; в противном случае байтовый элемент YMM3 устанавливается в `0x00`. Затем команда `vpmovmskb eax,ymm3` создает маску старшего разряда каждого байтового элемента в YMM3 и сохраняет эту маску в регистре EAX. Более конкретно, эта команда вычисляет `eax[i] = ymm3[i*8+7]` для i = 0, 1, 2, … 31, то есть каждый единичный бит в регистре EAX означает неусеченный пиксель. Последующая команда `not eax` преобразует битовый шаблон в EAX в маску усеченных пикселей, а команда `popcnt eax,eax` подсчитывает количество единичных битов в EAX. Это значение счетчика, которое соответствует количеству усеченных пикселей в YMM2, затем добавляется к общему количеству усеченных пикселей в регистре R8. Цикл обработки повторяется до тех пор, пока не будут обработаны все пиксели. Вот результаты выполнения примера исходного кода `Ch10_04`:

```
Результаты Avx2ClipPixels
  cd1.m_NumClippedPixels1: 328090
  cd2.m_NumClippedPixels2: 328090
  Сравнение буферной памяти пикселей пройдено

Выполняется проверка быстродействия Avx2ClipPixels_BM - подождите
Результаты сохранены в файл Ch10_04_Avx2ClipPixels_BM_CHROMIUM.csv
```

В табл. 10.1 показаны результаты измерения быстродействия для функций усечения пикселей `Avx2ClipPixelsCpp` и `Avx2ClipPixels_`.

Таблица 10.1. Среднее время выполнения (мс) для функций усечения пикселей (размер буфера изображения = 8 МБ)

Процессор	Avx2ClipPixelsCpp	Avx2ClipPixels_
i7-4790S	13005	1078
i9-7900X	11617	719
i7-8700K	11252	644

10.2.2. Поиск минимального и максимального значений RGB

В листинге 10.5 показан исходный код примера `Ch10_05` на C++ и на языке ассемблера, который иллюстрирует поиск минимального и максимального значений интенсивности пикселей в изображении RGB. В этом примере также

объясняется, как использовать некоторые расширенные возможности обработки макросов MASM.

Листинг 10.5. Пример Ch10_05

```
//-----------------------------------------------
//               Ch10_05.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#include <random>
#include "AlignedMem.h"

using namespace std;

extern "C" bool Avx2CalcRgbMinMax_(uint8_t* rgb[3], size_t num_pixels, uint8_t min_vals[3],
uint8_t max_vals[3]);

void Init(uint8_t* rgb[3], size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {5, 250};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
    {
        rgb[0][i] = (uint8_t)ui_dist(rng);
        rgb[1][i] = (uint8_t)ui_dist(rng);
        rgb[2][i] = (uint8_t)ui_dist(rng);
    }

    // Известные значения минимума и максимума для нужд тестирования
    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;
}

bool Avx2CalcRgbMinMaxCpp(uint8_t* rgb[3], size_t num_pixels, uint8_t min_vals[3], uint8_t
max_vals[3])
{
    // Проверка корректности num_pixels
    if ((num_pixels == 0) || (num_pixels % 32 != 0))
        return false;

    if (!AlignedMem::IsAligned(rgb[0], 32))
        return false;
    if (!AlignedMem::IsAligned(rgb[1], 32))
        return false;
    if (!AlignedMem::IsAligned(rgb[2], 32))
        return false;

    // Нахождение минимума и максимума для каждого цвета
    min_vals[0] = min_vals[1] = min_vals[2] = 255;
    max_vals[0] = max_vals[1] = max_vals[2] = 0;
```

```
   for (size_t i = 0; i < 3; i++)
   {
        for (size_t 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 main(void)
{
    const size_t n = 1024;
    uint8_t* rgb[3];
    uint8_t min_vals1[3], max_vals1[3];
    uint8_t min_vals2[3], max_vals2[3];

    AlignedArray<uint8_t> r(n, 32);
    AlignedArray<uint8_t> g(n, 32);
    AlignedArray<uint8_t> b(n, 32);

    rgb[0] = r.Data();
    rgb[1] = g.Data();
    rgb[2] = b.Data();

    Init(rgb, n, 219);
    Avx2CalcRgbMinMaxCpp(rgb, n, min_vals1, max_vals1);
    Avx2CalcRgbMinMax_(rgb, n, min_vals2, max_vals2);

    cout << "Результаты Avx2CalcRgbMinMax\n\n";
    cout << "                R   G   B\n";
    cout << "------------------------\n";

    cout << "min_vals1: ";
    cout << setw(4) << (int)min_vals1[0]  << ' ';
    cout << setw(4) << (int)min_vals1[1]  << ' ';
    cout << setw(4) << (int)min_vals1[2]  << '\n';
    cout << "min_vals2: ";
    cout << setw(4) << (int)min_vals2[0]  << ' ';
    cout << setw(4) << (int)min_vals2[1]  << ' ';
    cout << setw(4) << (int)min_vals2[2]  << "\n\n";

    cout << "max_vals1: ";
    cout << setw(4) << (int)max_vals1[0]  << ' ';
    cout << setw(4) << (int)max_vals1[1]  << ' ';
    cout << setw(4) << (int)max_vals1[2]  << '\n';
    cout << "max_vals2: ";
    cout << setw(4) << (int)max_vals2[0]  << ' ';
    cout << setw(4) << (int)max_vals2[1]  << ' ';
    cout << setw(4) << (int)max_vals2[2]  << "\n\n";

    return 0;
}
```

```
;-----------------------------------------------
;              Ch10_05.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; 256-битные константы
ConstVals       segment readonly align(32) 'const'
InitialPminVal  db 32 dup(0ffh)
InitialPmaxVal  db 32 dup(00h)
ConstVals       ends

; Macro _YmmVpextrMinub
;
; Этот макрос генерирует код, который извлекает младший беззнаковый байт из регистра YmmSrc

_YmmVpextrMinub macro GprDes,YmmSrc,YmmTmp

; Проверка, что YmmSrc и YmmTmp различаются
.erridni <YmmSrc>, <YmmTmp>, <Invalid registers>

; Составление текстовых строк для соответствующих регистров XMM
        YmmSrcSuffix SUBSTR <YmmSrc>,2
        XmmSrc CATSTR <X>,YmmSrcSuffix

        YmmTmpSuffix SUBSTR <YmmTmp>,2
        XmmTmp CATSTR <X>,YmmTmpSuffix

; Уменьшение числа минимальных значений в YmmSrc до одного наименьшего
        vextracti128 XmmTmp,YmmSrc,1
        vpminub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = финальные 16 мин. значений

        vpsrldq XmmTmp,XmmSrc,8
        vpminub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = финальные 8 мин. значений

        vpsrldq XmmTmp,XmmSrc,4
        vpminub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = финальные 4 мин. значения

        vpsrldq XmmTmp,XmmSrc,2
        vpminub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = финальные 2 мин. значения

        vpsrldq XmmTmp,XmmSrc,1
        vpminub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = финальное 1 мин. значение

        vpextrb GprDes,XmmSrc,0             ;размещение финального минимума в Gpr
        endm

; Macro _YmmVpextrMaxub
;
; Этот макрос генерирует код, который извлекает старший беззнаковый байт из регистра
YmmSrc.

_YmmVpextrMaxub macro GprDes,YmmSrc,YmmTmp

; Проверка, что YmmSrc и YmmTmp различаются
.erridni <YmmSrc>, <YmmTmp>, <Invalid registers>
```

```
; Составление текстовых строк для соответствующих регистров XMM
        YmmSrcSuffix SUBSTR <YmmSrc>,2
        XmmSrc CATSTR <X>,YmmSrcSuffix

        YmmTmpSuffix SUBSTR <YmmTmp>,2
        XmmTmp CATSTR <X>,YmmTmpSuffix

; Уменьшение числа максимальных значений в YmmSrc до одного наибольшего
        vextracti128 XmmTmp,YmmSrc,1
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = финальные 16 макс. значений

        vpsrldq XmmTmp,XmmSrc,8
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = финальные 8 макс. значений

        vpsrldq XmmTmp,XmmSrc,4
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = финальные 4 макс. значения

        vpsrldq XmmTmp,XmmSrc,2
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = финальные 2 макс. значения

        vpsrldq XmmTmp,XmmSrc,1
        vpmaxub XmmSrc,XmmSrc,XmmTmp        ;XmmSrc = финальное 1 макс. значение

        vpextrb GprDes,XmmSrc,0             ;размещение финального максимума в Gpr
        endm

; extern "C" bool Avx2CalcRgbMinMax_(uint8_t* rgb[3], size_t num_pixels, uint8_t min_
vals[3], uint8_t max_vals[3])

        .code
Avx2CalcRgbMinMax_ proc frame
        _CreateFrame CalcMinMax_,0,48,r12
        _SaveXmmRegs xmm6,xmm7,xmm8
        _EndProlog

; Проверка, что num_pixels и массивы цветов корректны
        xor eax,eax                         ;возвращаемый код ошибки

        test rdx,rdx
        jz Done                             ;переход, если num_pixels == 0
        test rdx,01fh
        jnz Done                            ;переход, если num_pixels % 32 != 0

        mov r10,[rcx]                       ;r10 = цветовая плоскость R
        test r10,1fh
        jnz Done                            ;переход, если цветовая плоскость R не
выровнена

        mov r11,[rcx+8]                     ;r11 = цветовая плоскость G
        test r11,1fh
        jnz Done                            ;переход, если цветовая плоскость G не
выровнена

        mov r12,[rcx+16]                    ;r12 = цветовая плоскость B
        test r12,1fh
        jnz Done                           ;переход, если цветовая плоскость B не выровнена
```

```
; Инициализация регистров рабочего цикла
        vmovdqa ymm3,ymmword ptr [InitialPminVal]   ;ymm3 = R минимумы
        vmovdqa ymm4,ymm3                           ;ymm4 = G минимумы
        vmovdqa ymm5,ymm3                           ;ymm5 = B минимумы

        vmovdqa ymm6,ymmword ptr [InitialPmaxVal]   ;ymm6 = R максимумы
        vmovdqa ymm7,ymm6                           ;ymm7 = G максимумы
        vmovdqa ymm8,ymm6                           ;ymm8 = B максимумы

        xor rcx,rcx                         ;rcx = общее смещение массива

; Сканирование массивов плоскостей RGB для поиска упакованных минимумов и максимумов
        align 16
@@:     vmovdqa ymm0,ymmword ptr [r10+rcx]  ;ymm0 = R pixels
        vmovdqa ymm1,ymmword ptr [r11+rcx]  ;ymm1 = G pixels
        vmovdqa ymm2,ymmword ptr [r12+rcx]  ;ymm2 = B pixels

        vpminub ymm3,ymm3,ymm0              ;обновление минимумов R
        vpminub ymm4,ymm4,ymm1              ;обновление минимумов G
        vpminub ymm5,ymm5,ymm2              ;обновление минимумов B

        vpmaxub ymm6,ymm6,ymm0              ;обновление максимумов R
        vpmaxub ymm7,ymm7,ymm1              ;обновление максимумов G
        vpmaxub ymm8,ymm8,ymm2              ;обновление максимумов B

        add rcx,32
        sub rdx,32
        jnz @B

; Вычисление окончательных минимумов RGB
        _YmmVpextrMinub rax,ymm3,ymm0
        mov byte ptr [r8],al                ;сохранение минимального R
        _YmmVpextrMinub rax,ymm4,ymm0
        mov byte ptr [r8+1],al              ;сохранение минимального G
        _YmmVpextrMinub rax,ymm5,ymm0
        mov byte ptr [r8+2],al              ;сохранение минимального B

; Вычисление окончательных максимумов RGB
        _YmmVpextrMaxub rax,ymm6,ymm1
        mov byte ptr [r9],al                ;сохранение максимального R
        _YmmVpextrMaxub rax,ymm7,ymm1
        mov byte ptr [r9+1],al              ;сохранение максимального G
        _YmmVpextrMaxub rax,ymm8,ymm1
        mov byte ptr [r9+2],al              ;сохранение максимального B

        mov eax,1                           ;возвращаемый код успешного выполнения

Done:   vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame r12
        ret
Avx2CalcRgbMinMax_ endp
        end
```

Функция Avx2CalcRgbMinMaxCpp, представленная в листинге 10.5, является реализацией на C++ алгоритма поиска минимума и максимума RGB. Эта функция использует набор вложенных циклов for для определения минимального и максимального значений интенсивности пикселей для каждой цветовой плоскости. Эти значения хранятся в массивах min_val и max_val. Функция main использует класс шаблона C++ AlignedArray для выделения трех массивов, имитирующих буферы цветовой плоскости изображения RGB. Эти буферы загружаются случайными значениями функцией Init. Обратите внимание, что функция Init присваивает известные значения нескольким элементам в каждом буфере цветовой плоскости. Эти известные значения используются для проверки правильного выполнения функций поиска минимума и максимума как на языке C++, так и ассемблера.

В начале кода на языке ассемблера находится пользовательский константный сегмент с именем ConstVals, который определяет упакованные версии минимального и максимального значений начального пикселя. Пользовательский сегмент используется здесь для обеспечения выравнивания упакованных значений шириной 256 бит по границе 32 байтов, как объяснено в главе 9. Далее следуют определения макросов _YmmVpextrMinub и _YmmVpextrMaxub. Эти макросы содержат команды, извлекающие наименьшее и наибольшее байтовые значения из регистра YMM. Внутреннее устройство этих макросов будет объяснено немного позже.

Функция Avx2CalcRgbMinMax_ использует регистры YMM3–YMM5 и YMM6–YMM8 для сохранения минимального и максимального значений RGB соответственно. Во время каждой итерации основного цикла серия команд vpminub и vpmaxub обновляет текущие минимумы и максимумы RGB. По завершении основного цикла вышеупомянутые регистры YMM содержат по 32 минимальных и максимальных значения интенсивности пикселей для каждого цветового компонента. Далее при помощи макросов _YmmVpextrMinub и _YmmVpextrMaxub происходит извлечение окончательных минимальных и максимальных значений пикселей RGB. Затем эти значения сохраняются в результирующих массивах min_val и max_val соответственно.

Определения макросов _YmmVpextrMinub и _YmmVpextrMaxub идентичны, за исключением команд vpminub и vpmaxub. В нижеследующем тексте все пояснения для _YmmVpextrMinub также относятся к _YmmVpextrMaxub. Макрос _YmmVpextrMinub нуждается в трех параметрах: целевой регистр общего назначения (GprDes), исходный регистр YMM (YmmSrc) и временный регистр YMM (YmmTmp). Обратите внимание, что параметры макроса YmmSrc и YmmTmp должны быть разными регистрами. Если они совпадают, директива .erridni (ошибка, если текстовые элементы идентичны) генерирует сообщение об ошибке во время сборки. MASM также поддерживает несколько других директив условных ошибок, помимо .erridni, и они описаны в документации Visual Studio.

Чтобы сгенерировать правильный код на языке ассемблера, макросу _YmmVpextrMinub требуется текстовая строка регистра XMM (XmmSrc), которая соответствует младшей части указанного регистра YmmSrc. Например, если YmmSrc равно YMM0, то XmmSrc должно быть равно XMM0. Для инициализации XmmSrc используются директивы MASM substr (возвращает подстроку текстового объекта) и catstr (конкатенация строк). Оператор YmmSrcSuffix SUBSTR <YmmSrc>,2

назначает значение текстовой строки для YmmSrcSuffix, где исключен начальный символ параметра макроса YmmSrc. Например, если YmmSrc равно YMM0, то YmmSrcSuffix равно MM0. Следующий оператор, XmmSrc CATSTR <X>, YmmSrcSuffix, добавляет ведущий X к значению YmmSrcSuffix и присваивает его XmmSrc. Если продолжить предыдущий пример, это означает, что XmmSrc присвоена текстовая строка XMM0. Затем директивы SUBSTR и CATSTR используются для присвоения значения текстовой строке XmmTmp.

После инициализации требуемых строк текста макроса следуют команды, извлекающие наименьшее значение байта из указанного регистра YMM. Команда vextracti128 XmmTmp,YmmSrc,1 копирует старшие 16 байт регистра YmmSrc в XmmTmp. (Команда vextracti128 также поддерживает использование непосредственного операнда 0 для копирования младших 16 байт.) Команда vpminub XmmSrc,XmmSrc,XmmTmp загружает последние 16 минимальных значений в XmmSrc. Команда vpsrldq XmmTmp,XmmSrc,8 сдвигает копию значения в XmmSrc вправо на 8 байт и сохраняет результат в XmmTmp. Это облегчает использование другой команды vpminub, которая уменьшает количество минимальных значений байтов с 16 до 8. Затем используются повторяющиеся наборы команд vpsrldq и vpminub до тех пор, пока окончательное минимальное значение не будет находиться в младшем байте XmmSrc. Команда vpextrb GprDes,XmmSrc,0 копирует окончательное минимальное значение в указанный регистр общего назначения. Вот результаты выполнения исходного кода примера Ch10_05:

```
Результаты Avx2CalcRgbMinMax

              R    G    B
-------------------------
min_vals1:    4    1    3
min_vals2:    4    1    3
max_vals1:  254  251  252
max_vals2:  254  251  252
```

10.2.3. Преобразование RGB в оттенки серого

Последний пример исходного кода этой главы, Ch10_06, объясняет, как выполнить преобразование изображения RGB в оттенки серого. Этот пример объединяет возможности работы AVX2 с упакованными целыми числами, о которых вы узнали в данной главе, с методами обработки упакованных чисел с плавающей запятой, представленными в главе 9. В листинге 10.6 показан исходный код примера Ch10_06.

Листинг 10.6. Пример Ch10_06

```
//-----------------------------------------------
//              Ch10_06.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <stdexcept>
#include "Ch10_06.h"
```

```
#include "AlignedMem.h"
#include "ImageMatrix.h"

using namespace std;

// Ограничения размера изображения
extern "C" const int c_NumPixelsMin = 32;
extern "C" const int c_NumPixelsMax = 256 * 1024 * 1024;

// Коэффициенты преобразования RGB в градации серого
const float c_Coef[4] {0.2126f, 0.7152f, 0.0722f, 0.0f};

bool CompareGsImages(const uint8_t* pb_gs1,const uint8_t* pb_gs2, int num_pixels)
{
    for (int i = 0; i < num_pixels; i++)
    {
        if (abs((int)pb_gs1[i] - (int)pb_gs2[i]) > 1)
            return false;
    }

    return true;
}

bool Avx2ConvertRgbToGsCpp(uint8_t* pb_gs, const RGB32* pb_rgb, int num_pixels, const float
coef[4])
{
    if (num_pixels < c_NumPixelsMin || num_pixels > c_NumPixelsMax)
        return false;
    if (num_pixels % 8 != 0)
        return false;

    if (!AlignedMem::IsAligned(pb_gs, 32))
        return false;
    if (!AlignedMem::IsAligned(pb_rgb, 32))
        return false;

    for (int i = 0; i < num_pixels; i++)
    {
        uint8_t r = pb_rgb[i].m_R;
        uint8_t g = pb_rgb[i].m_G;
        uint8_t b = pb_rgb[i].m_B;

        float gs_temp = r * coef[0] + g * coef[1] + b * coef[2] + 0.5f;

        if (gs_temp < 0.0f)
            gs_temp = 0.0f;
        else if (gs_temp > 255.0f)
            gs_temp = 255.0f;

        pb_gs[i] = (uint8_t)gs_temp;
    }

    return true;
}

void Avx2ConvertRgbToGs(void)
```

```
{
    const wchar_t* fn_rgb = L"..\\Ch10_Data\\TestImage3.bmp";
    const wchar_t* fn_gs1 = L"Ch10_06_Avx2ConvertRgbToGs_TestImage3_GS1.bmp";
    const wchar_t* fn_gs2 = L"Ch10_06_Avx2ConvertRgbToGs_TestImage3_GS2.bmp";

    ImageMatrix im_rgb(fn_rgb);
    int im_h = im_rgb.GetHeight();
    int im_w = im_rgb.GetWidth();
    int num_pixels = im_h * im_w;
    ImageMatrix im_gs1(im_h, im_w, PixelType::Gray8);
    ImageMatrix im_gs2(im_h, im_w, PixelType::Gray8);
    RGB32* pb_rgb = im_rgb.GetPixelBuffer<RGB32>();
    uint8_t* pb_gs1 = im_gs1.GetPixelBuffer<uint8_t>();
    uint8_t* pb_gs2 = im_gs2.GetPixelBuffer<uint8_t>();

    cout << "Результаты Avx2ConvertRgbToGs\n";
    wcout << "Конвертация изображения RGB " << fn_rgb << '\n';
    cout << "  im_h = " << im_h << " pixels\n";
    cout << "  im_w = " << im_w << " pixels\n";

    // Выполнение функций преобразования
    bool rc1 = Avx2ConvertRgbToGsCpp(pb_gs1, pb_rgb, num_pixels, c_Coef);
    bool rc2 = Avx2ConvertRgbToGs_(pb_gs2, pb_rgb, num_pixels, c_Coef);

    if (rc1 && rc2)
    {
        wcout << "Сохранение изображения в тонах серого #1 - " << fn_gs1 << '\n';
        im_gs1.SaveToBitmapFile(fn_gs1);

        wcout << " Сохранение изображения в тонах серого #2 - " << fn_gs2 << '\n';
        im_gs2.SaveToBitmapFile(fn_gs2);

        if (CompareGsImages(pb_gs1, pb_gs2, num_pixels))
            cout << "Изображения совпадают\n";
        else
            cout << "Изображения не совпадают\n";
    }
    else
        cout << "Возвращен код ошибки\n";
}

int main()
{
    try
    {
        Avx2ConvertRgbToGs();
        Avx2ConvertRgbToGs_BM();
    }

    catch (runtime_error& rte)
    {
        cout << "Ошибка при выполнении программы - " << rte.what() << '\n';
    }

    catch (...)
    {
```

```
        cout << "Обнаружено исключение\n";
    }

    return 0;
}

;-----------------------------------------------
;               Ch10_06.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>

                .const
GsMask          dword 0ffffffffh, 0, 0, 0, 0ffffffffh, 0, 0, 0
r4_0p5          real4 0.5
r4_255p0        real4 255.0

                extern c_NumPixelsMin:dword
                extern c_NumPixelsMax:dword

; extern "C" bool Avx2ConvertRgbToGs_(uint8_t* pb_gs, const RGB32* pb_rgb, int num_pixels,
const float coef[4])
;
; Примечание: указатели памяти pb_rgb расположены следующим образом:
;       R(0,0), G(0,0), B(0,0), A(0,0), R(0,1), G(0,1), B(0,1), A(0,1), ...

        .code
Avx2ConvertRgbToGs_ proc frame
        _CreateFrame RGBGS_,0,112
        _SaveXmmRegs xmm6,xmm7,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

; Проверка значений аргументов
        xor eax,eax                         ;Возвращаемый код ошибки
        cmp r8d,[c_NumPixelsMin]
        jl Done                             ;переход, если num_pixels < мин.значения
        cmp r8d,[c_NumPixelsMax]
        jg Done                             ;переход, если num_pixels > макс.значения
        test r8d,7
        jnz Done                            ;переход, если (num_pixels % 8) != 0

        test rcx,1fh
        jnz Done                            ;переход, если pb_gs не выровнен
        test rdx,1fh
        jnz Done                            ;переход, если pb_rgb не выровнен

; Выполнение инициализации
        vbroadcastss ymm11,real4 ptr [r4_255p0]     ;ymm11 = упакованное 255.0
        vbroadcastss ymm12,real4 ptr [r4_0p5]       ;ymm12 = упакованное 0.5
        vpxor ymm13,ymm13,ymm13                     ;ymm13 = упакованный ноль

        vmovups xmm0,xmmword ptr [r9]
        vperm2f128 ymm14,ymm0,ymm0,00000000b        ;ymm14 = упакованный coef

        vmovups ymm15,ymmword ptr [GsMask]          ;ymm15 = GsMask (SPFP)
```

```
; Чтение значений следующих 8 пикселей RGB32 (P0-P7)
        align 16
@@:     vmovdqa ymm0,ymmword ptr [rdx]      ;ymm0 = 8 пикселей rgb32 (P7-P0)

; Расширение цветовых компонентов RGB32 из байтов в двойные слова
        vpunpcklbw ymm1,ymm0,ymm13
        vpunpckhbw ymm2,ymm0,ymm13
        vpunpcklwd ymm3,ymm1,ymm13          ;ymm3 = P1, P0 (dword)
        vpunpckhwd ymm4,ymm1,ymm13          ;ymm4 = P3, P2 (dword)
        vpunpcklwd ymm5,ymm2,ymm13          ;ymm5 = P5, P4 (dword)
        vpunpckhwd ymm6,ymm2,ymm13          ;ymm6 = P7, P6 (dword)

; Конвертация значений цветовых компонентов в формат одинарной точности с плавающей запятой
        vcvtdq2ps ymm0,ymm3                 ;ymm0 = P1, P0 (SPFP)
        vcvtdq2ps ymm1,ymm4                 ;ymm1 = P3, P2 (SPFP)
        vcvtdq2ps ymm2,ymm5                 ;ymm2 = P5, P4 (SPFP)
        vcvtdq2ps ymm3,ymm6                 ;ymm3 = P7, P6 (SPFP)

; Умножение значений цветовых компонентов на коэффициенты конвертации
        vmulps ymm0,ymm0,ymm14
        vmulps ymm1,ymm1,ymm14
        vmulps ymm2,ymm2,ymm14
        vmulps ymm3,ymm3,ymm14

; Суммирование взвешенных значений цветовых компонентов для получения
; финального значения серого
        vhaddps ymm4,ymm0,ymm0
        vhaddps ymm4,ymm4,ymm4              ;ymm4[159:128] = P1, ymm4[31:0] = P0
        vhaddps ymm5,ymm1,ymm1
        vhaddps ymm5,ymm5,ymm5              ;ymm5[159:128] = P3, ymm4[31:0] = P2
        vhaddps ymm6,ymm2,ymm2
        vhaddps ymm6,ymm6,ymm6              ;ymm6[159:128] = P5, ymm4[31:0] = P4
        vhaddps ymm7,ymm3,ymm3
        vhaddps ymm7,ymm7,ymm7              ;ymm7[159:128] = P7, ymm4[31:0] = P6

; Слияние значений серого в одном регистре YMM
        vandps ymm4,ymm4,ymm15              ;маскирование ненужных значений
        vandps ymm5,ymm5,ymm15
        vandps ymm6,ymm6,ymm15
        vandps ymm7,ymm7,ymm15
        vpslldq ymm5,ymm5,4
        vpslldq ymm6,ymm6,8
        vpslldq ymm7,ymm7,12
        vorps ymm0,ymm4,ymm5                ;слияние значений
        vorps ymm1,ymm6,ymm7
        vorps ymm2,ymm0,ymm1                ;ymm2 = 8 значений пикселей (SPFP)

; Добавление фактора округления 0.5 и усечение до диапазона 0.0-255.0
        vaddps ymm2,ymm2,ymm12              ;добавление фактора округления 0.5f
        vminps ymm3,ymm2,ymm11              ;усечение пикселей больше 255.0
        vmaxps ymm4,ymm3,ymm13              ;усечение пикселей больше 0.0

; Конвертация значений одинарной точности с плавающей запятой в байты и сохранение
        vcvtps2dq ymm3,ymm2                 ;конвертация значений серого в двойные слова
        vpackusdw ymm4,ymm3,ymm13           ;конвертация двойных слов в одинарные
        vpackuswb ymm5,ymm4,ymm13           ;конвертация одинарных слов в байты

        vperm2i128 ymm6,ymm13,ymm5,3        ;xmm5 = GS P3:P0, xmm6 = GS P7:P4
```

```
        vmovd dword ptr [rcx],xmm5          ;сохранение P3-P0
        vmovd dword ptr [rcx+4],xmm6        ;сохранение P7-P4

        add rdx,32                          ;обновление указателя pb_rgb
        add rcx,8                           ;обновление указателя pb_gs
        sub r8d,8                           ;num_pixels -= 8
        jnz @B                              ;повторение до завершения

        mov eax,1                           ;возвращаемый код успешного выполнения

Done:   vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm11,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame
        ret
Avx2ConvertRgbToGs_ endp
        end
```

Существует множество алгоритмов преобразования изображения RGB в изображение в оттенках серого. Один из часто используемых методов вычисляет значения пикселей в градациях серого с использованием взвешенной суммы компонентов цвета RGB. В этом примере исходного кода пиксели RGB преобразуются в пиксели в градациях серого с использованием следующего уравнения:

$$GS(x,y) = R(x,y)W_r + G(x,y)W_g + B(x,y)W_b.$$

Вес (или коэффициент) каждого компонента цвета RGB представляет собой число с плавающей запятой от 0,0 до 1,0, а сумма трех компонентных коэффициентов обычно равна 1,0. Точные значения, применяемые в качестве коэффициентов цветовых компонентов, обычно основаны на опубликованных стандартах, которые учитывают множество визуальных факторов, включая свойства целевого цветового пространства, характеристики устройства отображения и воспринимаемое качество изображения. Если вы хотите узнать больше о преобразовании изображений из RGB в оттенки серого, в приложении содержатся некоторые ссылки, к которым вы можете обратиться.

Исходный код примера `Ch10_06` начинается с объявления структуры `RGB32`. Эта структура объявлена в заголовочном файле `ImageMatrix.h` и определяет схему упорядочения цветовых компонентов для каждого пикселя RGB. Функция `Avx2ConvertRgbToGsCpp` содержит реализацию алгоритма преобразования RGB в оттенки серого на языке C++. Эта функция использует обычный цикл `for`, который просматривает буфер изображения `RGB32 pb_rgb` и вычисляет значения пикселей в градациях серого с использованием вышеупомянутого уравнения преобразования. Обратите внимание, что элемент `m_A RGB32` не используется ни в одном из вычислений в этом примере. Каждое вычисленное значение пикселя в градациях серого округляется с учетом коэффициента округления и усекается до [0,0, 255,0] перед сохранением в буфере изображения в градациях серого, на который указывает `pb_gs`.

Код на языке ассемблера начинается с раздела `.const`, который определяет необходимые константы. После пролога функция `Avx2ConvertRgbToGs_` выполняет стандартные проверки размера изображения и выравнивания буфера. Затем она загружает необходимые алгоритму упакованные константы в ре-

гистры YMM11–YMM15. Обратите внимание, что регистр YMM14 содержит упакованную версию коэффициентов преобразования цвета, как показано на рис. 10.3. Цикл обработки на языке ассемблера начинается с команды `vmovdqa ymm0,ymmword ptr [rdx]`, которая загружает восемь значений пикселей RGB32 в регистр YMM0. Затем размер цветовых компонентов этих пикселей увеличивается до двойных слов с помощью серии команд `vpunpck[l|h]bw` и `vpunpck[l|h]wd`. Далее команды `vcvtdq2ps` преобразуют цветовые компоненты пикселей из двойных слов в значения с плавающей запятой одинарной точности. После выполнения четырех команд `vcvtdq2ps` каждый регистр YMM0–YMM3 содержит два пикселя RGB32, и каждый компонент цвета является значением с плавающей запятой одинарной точности. На рис. 10.3 также показаны увеличение и преобразование размера RGB32, обсуждаемые в этом абзаце.

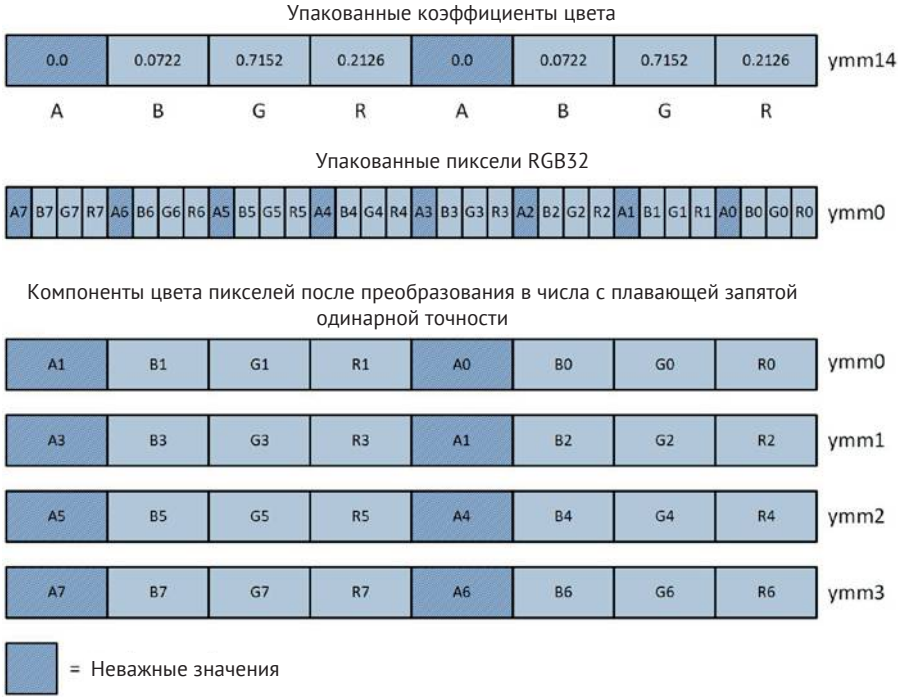


Рис. 10.3. Увеличение и преобразование размера цветовых компонентов пикселей RGB32

Четыре команды `vmulps` умножают восемь пикселей RGB32 на коэффициенты преобразования цвета. Следующие за ними команды `vhaddps` суммируют взвешенные цветовые компоненты каждого пикселя для создания требуемых значений оттенков серого. После выполнения этих команд каждый регистр YMM4–YMM7 содержит по два значения пикселей по шкале градаций серого с плавающей запятой одинарной точности, одно в позиции элемента [31:0], а другое в позиции [159:128], как показано на рис. 10.4. Затем восемь значений оттенков серого в YMM4–YMM7 объединяются в YMM2 с помощью серии команд `vandps`, `vpslldq` и `vorps`. На рис. 10.4 также показан окончательный результат объединения. Последующие команды `vaddps`, `vminps` и `vmaxps` добавляют

коэффициент округления (0,5) и усекают пиксели шкалы серого до диапазона [0,0, 255,0]. Затем эти значения преобразуются в байты без знака с помощью команд `vcvtps2dq`, `vpackusdw` и `vpackuswb`. Две команды `vmovd` сохраняют четыре байтовых значения пикселей без знака в XMM5 [31:0] и XMM6 [31:0] в буфер изображения в градациях серого.

Рис. 10.4. Значения пикселей с плавающей запятой одинарной точности в градациях серого до и после объединения

Ниже приведены результаты выполнения примера исходного кода `Ch10_06`:

```
Результаты Avx2ConvertRgbToGs

Конвертация изображения RGB ..\Ch10_Data\TestImage3.bmp
  im_h = 960 pixels
  im_w = 640 pixels
Сохранение изображения в тонах серого #1 - Ch10_06_Avx2ConvertRgbToGs_TestImage3_GS1.bmp
Сохранение изображения в тонах серого #2 - Ch10_06_Avx2ConvertRgbToGs_TestImage3_GS2.bmp
Изображения совпадают

Выполняется проверка быстродействия Avx2ConvertRgbToGs_BM - подождите
Результаты сохранены в файл Ch10_06_Avx2ConvertRgbToGs_BM_CHROMIUM.csv
```

В табл. 10.2 показаны результаты измерения быстродействия функций преобразования изображения RGB в оттенки серого `Avx2ConvertRgbToGsCpp` и `Avx2ConvertRgbToGs_`. Прирост производительности этого примера исходного кода скромен по сравнению с некоторыми другими примерами в данной книге. Причина этого в том, что компоненты цвета `RGB32` в буфере исходного изображения чередуются друг с другом, что требует использования более медленной

горизонтальной арифметики. Переупорядочивание данных RGB32 таким образом, чтобы пиксели каждого цветового компонента находились в отдельных буферах изображения, часто приводит к значительно более высокой производительности. Вы увидите пример этого подхода в главе 14.

Таблица 10.2. Среднее время выполнения (мс) преобразования изображения TestImage3.bmp из RGB в оттенки серого

Процессор	Avx2ConvertRgbToGsCpp	vx2ConvertRgbToGs_
i7-4790S	1504	843
i9-7900X	1075	593
i7-8700K	1031	565

10.3. Заключение

В главе 10 рассмотрены следующие ключевые моменты:

- AVX2 расширяет возможности AVX по работе с упакованными целыми числами. Большинство команд x86-AVX для работы с упакованными целыми числами можно использовать со 128-битными или 256-битными операндами. Эти операнды должны быть правильно выровнены всегда, когда это возможно;
- подобно x86-AVX с плавающей запятой, функции на языке ассемблера, которые выполняют вычисления упакованных целых чисел с использованием регистра YMM, должны использовать команду `vzeroupper` перед любым кодом эпилога или командой `ret`. Это позволяет избежать потенциальных задержек производительности, которые могут возникнуть при переходе процессора от выполнения команд x86-AVX к командам x86-SSE;
- для функций на языке ассемблера, возвращающих структуру по значению, действует несколько иное соглашение о вызовах Visual C++. Такая функция должна скопировать большую структуру (больше 8 байт) в буфер, на который указывает регистр RCX. Обычные регистры соглашения о вызовах также «сдвигаются вправо», о чем было сказано в этой главе;
- функции на языке ассемблера могут использовать команды `vpunpckl[bw|wd|dq]` и `vpunpckh[bw|wd|dq]` для распаковки 128-битных или 256-битных целочисленных операндов;
- функции на языке ассемблера могут использовать команды `vpackss[dw|wb]` и `vpackus[dw|wb]` для упаковки 128-битных или 256-битных целочисленных операндов с использованием насыщения со знаком или без знака;
- функции на языке ассемблера могут использовать команды `vmovzx[bw|bd|bq|wd|wq|dq]` и `vmovsx[bw|bd|bq|wd|wq|dq]` для выполнения расширения упакованных целых чисел с дополнением нулем или знаковым битом;
- MASM поддерживает директивы, которые могут выполнять элементарные операции обработки строк. Эти директивы можно использовать для создания текстовых строк мнемоник, операндов и меток макросов. MASM также поддерживает директивы условных ошибок, которые можно использовать для сигнализации об ошибках во время сборки исходного кода.

Глава 11

Программирование AVX2 – расширенные команды

В этой главе вы узнаете о практическом применении расширений набора команд, которые были представлены в главе 8. Первый раздел содержит несколько примеров исходного кода, иллюстрирующих использование команд слитного умножения-сложения в скалярной и упакованной формах. Во втором разделе рассмотрены команды для работы с регистрами общего назначения. В этот раздел включены примеры исходного кода, объясняющие умножение без флагов и битовый сдвиг. В нем также рассмотрены некоторые из улучшенных команд по манипулированию битами. В последнем разделе представлены команды, выполняющие преобразования половинной точности с плавающей запятой.

Примеры исходного кода в первых двух разделах этой главы правильно работают на большинстве процессоров AMD и Intel, поддерживающих AVX2. Пример исходного кода с плавающей запятой половинной точности работает на процессорах AMD и Intel, которые поддерживают AVX и расширение набора команд F16C. Напоминаю, что вы никогда не должны быть уверены в доступности конкретного расширения набора команд независимо от того, поддерживает процессор AVX или AVX2. Производственный код всегда должен проверять доступность определенного расширения набора команд с помощью команды `cpuid`. Вы узнаете, как это сделать, в главе 16.

11.1. Программирование операций FMA

Команды FMA выполняют умножение с плавающей запятой и последующее сложение с плавающей запятой с использованием единственной операции округления. В главе 8 были представлены операции FMA и подробно рассмотрены их особенности. В этом разделе вы узнаете, как использовать команды FMA для реализации функций *дискретной свертки*. Раздел начинается с краткого обзора математических основ свертки. Цель этого обзора – изложить теорию, достаточную для понимания примеров исходного кода. Далее следует пример исходного кода, который реализует практическую функцию

дискретной свертки с использованием скалярных команд FMA. Раздел завершается примером исходного кода, в котором для повышения производительности функции, выполняющей дискретные свертки, используются упакованные команды FMA.

11.1.1. Свертки

Свертка – это математическая операция, которая смешивает входной сигнал с сигналом отклика для создания выходного сигнала. Формально свертка входного сигнала *f* и сигнала отклика *g* определяется следующим образом:

$$h(t) = \int_{-\infty}^{\infty} f(t-\tau)\,g(\tau)\,d\tau,$$

где *h* представляет собой выходной сигнал. Для обозначения свертки сигналов (или функций) *f* и *g* обычно используется запись $f * g$.

Свертки широко используются в самых разных научных и инженерных приложениях. Многие методы обработки сигналов и изображений основаны на теории свертки. В этих областях дискретные массивы точек выборки данных представляют сигналы входа, отклика и выхода. *Дискретную свертку* можно рассчитать с помощью следующего уравнения:

$$h[i] = \sum_{k=-M}^{M} f[i-k]\,g[k],$$

где $i = 0, 1, \ldots, N-1$ и $M = \lfloor N_g/2 \rfloor$. В предыдущих уравнениях *N* обозначает количество элементов в массивах входных и выходных сигналов, а N_g символизирует размер массива сигналов отклика. Все объяснения и примеры исходного кода в этом разделе предполагают, что N_g – нечетное целое число, большее или равное трем. Если вы внимательно изучите уравнение дискретной свертки, то заметите, что каждый элемент в массиве выходных сигналов *h* можно найти при помощи относительно несложного расчета суммы произведений, который охватывает элементы массива входного сигнала *f* и массива сигнала отклика *g*. Вычисления такого типа легко реализуются с помощью команд FMA.

При цифровой обработке сигналов многие приложения используют *операторы сглаживания*, чтобы уменьшить количество шума, присутствующего в необработанном сигнале. Например, верхний график на рис. 11.1 показывает необработанный сигнал, который содержит изрядное количество шума. Нижний график на рис. 11.1 показывает тот же сигнал после применения оператора сглаживания. В этом случае оператор сглаживания свернул исходный необработанный сигнал с набором дискретных коэффициентов, которые аппроксимируют *фильтр Гаусса* (фильтр нижних частот). Эти коэффициенты соответствуют массиву сигналов отклика *g*, который включен в уравнение дискретной свертки. Массив сигналов отклика часто называют *ядром свертки*, или *маской свертки*.

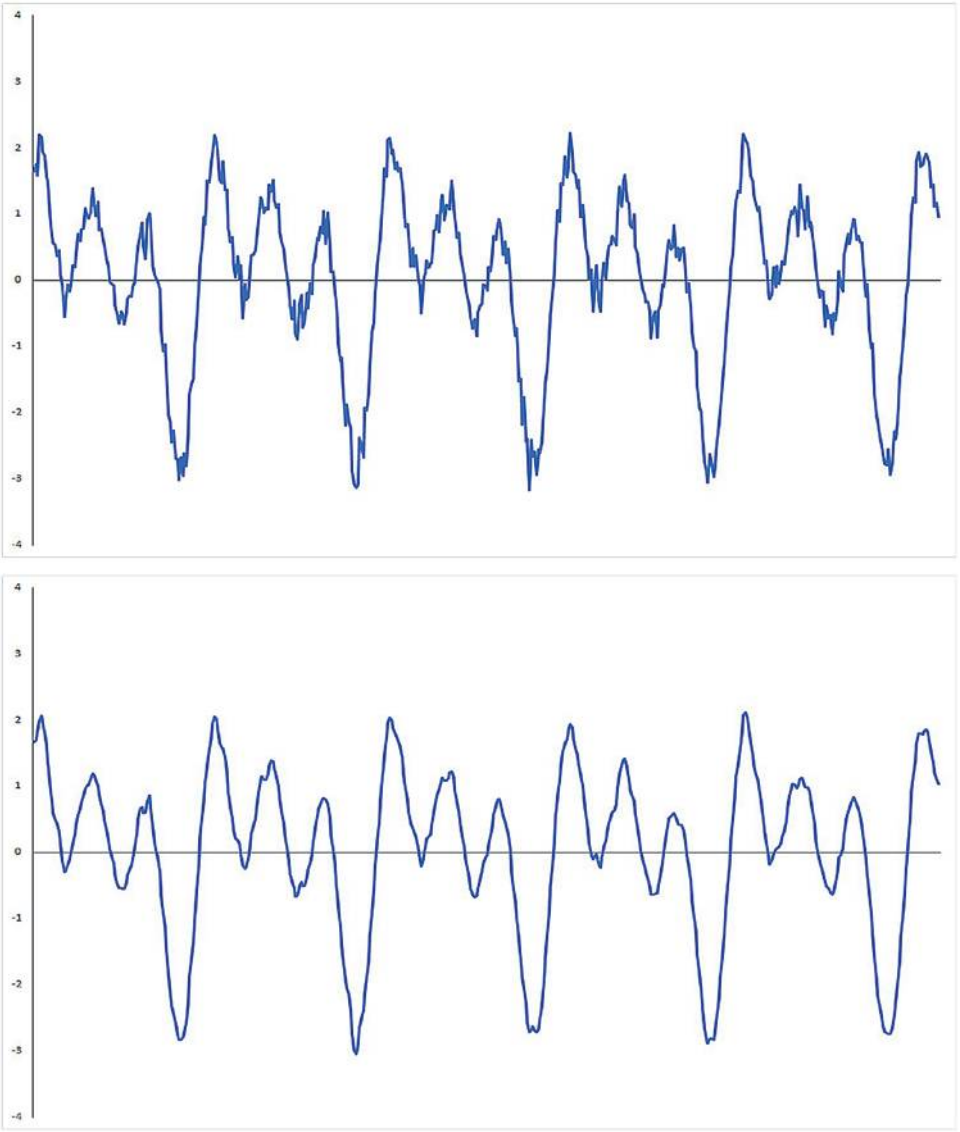


Рис. 11.1. Необработанный сигнал (вверху) и его сглаженный аналог (внизу)

Уравнение дискретной свертки может быть реализовано в исходном коде с помощью пары вложенных циклов `for`. Во время каждой итерации внешнего цикла центральная точка ядра свертки `g[0]` накладывается на текущий элемент массива входных сигналов `f[i]`. Внутренний цикл вычисляет промежуточные произведения, как показано на рис. 11.2. Эти промежуточные произведения затем суммируются и сохраняются в элементе массива выходных сигналов `h[i]`, который также показан на рис. 11.2. Примеры исходного кода FMA, представленные в этом разделе, реализуют функции свертки с использованием данного метода.

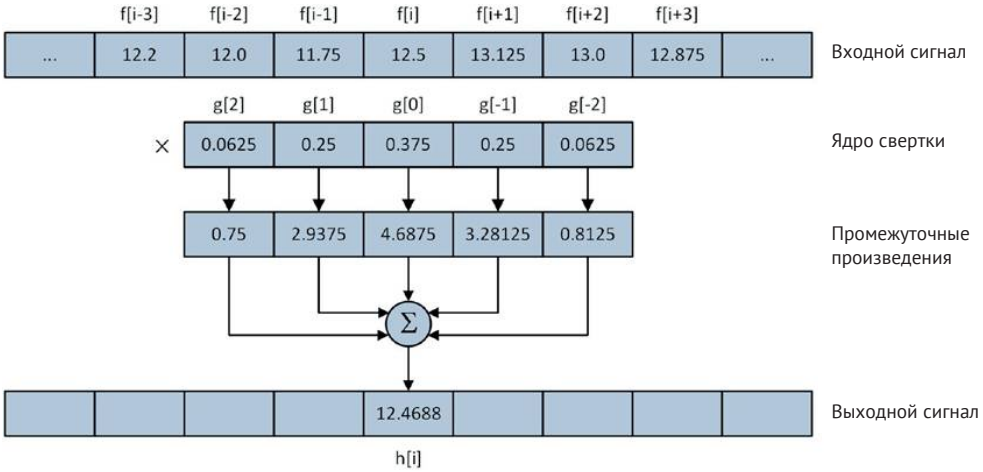


Рис. 11.2. Применение оператора сглаживания к элементу входного сигнала

В данном обзоре я постарался представить математические основы, достаточные для понимания примеров исходного кода. Существует обширная литература, в которой теорию свертки и обработки сигналов объясняют значительно более подробно. Приложение содержит список ссылок, к которым вы можете обратиться для получения дополнительной информации о теории свертки и обработки сигналов.

11.1.2. Скалярные операции FMA

Пример исходного кода `Ch11_01` объясняет, как реализовать одномерную дискретную функцию свертки с использованием скалярных команд FMA. Он также демонстрирует преимущества в производительности функций свертки, которые используют ядра свертки фиксированного размера по сравнению с ядрами свертки переменного размера. В листинге 11.1 показан исходный код примера `Ch11_01`.

Листинг 11.1. Пример Ch11_01

```
//-----------------------------------------------
//              Ch11_01.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <memory>
#include <fstream>
#include <stdexcept>
#include "Ch11_01.h"

using namespace std;

extern "C" const int c_NumPtsMin = 32;
extern "C" const int c_NumPtsMax = 16 * 1024 * 1024;
```

```
extern "C" const int c_KernelSizeMin = 3;
extern "C" const int c_KernelSizeMax = 15;
unsigned int g_RngSeedVal = 97;

void Convolve1(void)
{
    const int n1 = 512;
    const float kernel[] { 0.0625f, 0.25f, 0.375f, 0.25f, 0.0625f };
    const int ks = sizeof(kernel) / sizeof(float);
    const int ks2 = ks / 2;
    const int n2 = n1 + ks2 * 2;

    // Создание массива выборок сигнала
    unique_ptr<float[]> x1_up {new float[n1]};
    unique_ptr<float[]> x2_up {new float[n2]};
    float* x1 = x1_up.get();
    float* x2 = x2_up.get();

    CreateSignal(x1, n1, ks, g_RngSeedVal);
    PadSignal(x2, n2, x1, n1, ks2);

    // Выполнение свертки
    const int num_pts = n1;
    unique_ptr<float[]> y1_up {new float[num_pts]};
    unique_ptr<float[]> y2_up {new float[num_pts]};
    unique_ptr<float[]> y3_up {new float[num_pts]};
    unique_ptr<float[]> y4_up {new float[num_pts]};
    float* y1 = y1_up.get();
    float* y2 = y2_up.get();
    float* y3 = y3_up.get();
    float* y4 = y4_up.get();

    bool rc1 = Convolve1Cpp(y1, x2, num_pts, kernel, ks);
    bool rc2 = Convolve1_(y2, x2, num_pts, kernel, ks);
    bool rc3 = Convolve1Ks5Cpp(y3, x2, num_pts, kernel, ks);
    bool rc4 = Convolve1Ks5_(y4, x2, num_pts, kernel, ks);

    cout << "Результаты Convolve1\n";
    cout << "  rc1 = " << boolalpha << rc1 << '\n';
    cout << "  rc2 = " << boolalpha << rc2 << '\n';
    cout << "  rc3 = " << boolalpha << rc3 << '\n';
    cout << "  rc4 = " << boolalpha << rc4 << '\n';

    if (!rc1 || !rc2 || !rc3 || !rc4)
        return;

    // Сохранение данных
    const char* fn = "Ch11_01_Convolve1Results.csv";
    ofstream ofs(fn);

    if (ofs.bad())
        cout << "Ошибка создания файла - " << fn << '\n';
    else
    {
        const char* delim = ", ";
```

```
        ofs << fixed << setprecision(7);
        ofs << "i, x1, y1, y2, y3, y4\n";

        for (int i = 0; i < num_pts; i++)
        {
            ofs << setw(5) << i << delim;
            ofs << setw(10) << x1[i] << delim;
            ofs << setw(10) << y1[i] << delim;
            ofs << setw(10) << y2[i] << delim;
            ofs << setw(10) << y3[i] << delim;
            ofs << setw(10) << y4[i] << '\n';
        }

        ofs.close();
        cout << "\nРезультат свертки сохранен в файл " << fn << '\n';
    }
}

bool Convolve1Cpp(float* y, const float* x, int num_pts, const float* kernel, int kernel_
size)
{
    int ks2 = kernel_size / 2;

    if ((kernel_size & 1) == 0)
        return false;

    if (kernel_size < c_KernelSizeMin || kernel_size > c_KernelSizeMax)
        return false;

    if (num_pts < c_NumPtsMin || num_pts > c_NumPtsMax)
        return false;

    x += ks2;   // x указывает на первую выборку сигнала

    for (int i = 0; i < num_pts; i++)
    {
        float sum = 0;

        for (int k = -ks2; k <= ks2; k++)
        {
            float x_val = x[i - k];
            float kernel_val = kernel[k + ks2];

            sum += kernel_val * x_val;
        }

        y[i] = sum;
    }

    return true;
}

bool Convolve1Ks5Cpp(float* y, const float* x, int num_pts, const float* kernel, int
kernel_size)
{
    int ks2 = kernel_size / 2;
```

```
    if (kernel_size != 5)
        return false;

    if (num_pts < c_NumPtsMin || num_pts > c_NumPtsMax)
        return false;

    x += ks2;   // x указывает на первую выборку сигнала

    for (int i = 0; i < num_pts; i++)
    {
        float sum = 0;
        int j = i + ks2;

        sum += x[j] * kernel[0];
        sum += x[j - 1] * kernel[1];
        sum += x[j - 2] * kernel[2];
        sum += x[j - 3] * kernel[3];
        sum += x[j - 4] * kernel[4];

        y[i] = sum;
    }

    return true;
}

int main()
{
    int ret_val = 1;

    try
    {
        Convolve1();
        Convolve1_BM();
        ret_val = 0;
    }

    catch (runtime_error& rte)
    {
        cout << "Сбой выполнения программы\n";
        cout << rte.what() << '\n';
    }

    catch (...)
    {
        cout << "Возникло нераспознанное исключение\n";
    }

    return ret_val;
}

;-----------------------------------------------
;               Ch11_01_.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>
        extern c_NumPtsMin:dword
```

```
        extern c_NumPtsMax:dword
        extern c_KernelSizeMin:dword
        extern c_KernelSizeMax:dword

; extern "C" bool Convolve1_(float* y, const float* x, int num_pts, const float* kernel,
int kernel_size)

        .code
Convolve1_ proc frame
        _CreateFrame CV_,0,0,rbx,rsi
        _EndProlog

; Проверка значений аргументов
        xor eax,eax                         ;код ошибки

        mov r10d,dword ptr [rbp+CV_OffsetStackArgs]
        test r10d,1
        jz Done                             ;переход, если kernel_size четный
        cmp r10d,[c_KernelSizeMin]
        jl Done                             ;переход, если kernel_size слишком мал
        cmp r10d,[c_KernelSizeMax]
        jg Done                             ;переход, если kernel_size слишком велик

        cmp r8d,[c_NumPtsMin]
        jl Done                             ;переход, если num_pts слишком мал
        cmp r8d,[c_NumPtsMax]
        jg Done                             ;переход, если num_pts слишком велик

; Выполнение необходимой инициализации
        mov r8d,r8d                         ;r8 = num_pts
        shr r10d,1                          ;ks2 = ks / 2
        lea rdx,[rdx+r10*4]                 ;rdx = x + ks2 (первая выборка данных)

; Выполнение свертки
LP1:    vxorps xmm5,xmm5,xmm5               ;sum = 0.0;
        mov r11,r10
        neg r11                             ;k = -ks2

LP2:    mov rbx,rax
        sub rbx,r11                         ;rbx = i - k
        vmovss xmm0,real4 ptr [rdx+rbx*4]   ;xmm0 = x[i - k]
        mov rsi,r11
        add rsi,r10                         ;rsi = k + ks2
        vfmadd231ss xmm5,xmm0,[r9+rsi*4]    ;sum += x[i - k] * kernel[k + ks2]

        add r11,1                           ;k++
        cmp r11,r10
        jle LP2                             ;переход, если k <= ks2

        vmovss real4 ptr [rcx+rax*4],xmm5   ;y[i] = sum

        add rax,1                           ;i += 1
        cmp rax,r8
        jl LP1                              ;переход, если i < num_pts

        mov eax,1                           ;возвращаемый код успешного выполнения
```

```
Done:   vzeroupper
        _DeleteFrame rbx,rsi
        ret
Convolve1_ endp

; extern "C" bool Convolve1Ks5_(float* y, const float* x, int num_pts, const float* kernel,
int kernel_size)

Convolve1Ks5_ proc
; Проверка аргументов
        xor eax,eax                         ;код ошибки

        cmp dword ptr [rsp+40],5
        jne Done                            ;переход, если kernel_size не равен 5

        cmp r8d,[c_NumPtsMin]
        jl Done                             ;переход, если num_pts слишком мал
        cmp r8d,[c_NumPtsMax]
        jg Done                             ;переход, если num_pts слишком велик

; Выполнение необходимой инициализации
        mov r8d,r8d                         ;r8 = num_pts
        add rdx,8                           ;x += 2

; Выполнение свертки
@@:     vxorps xmm4,xmm4,xmm4                   ;инициализация суммы
        vxorps xmm5,xmm5,xmm5
        mov r11,rax
        add r11,2                               ;j = i + ks2

        vmovss xmm0,real4 ptr [rdx+r11*4]       ;xmm0 = x[j]
        vfmadd231ss xmm4,xmm0,[r9]              ;xmm4 += x[j] * kernel[0]

        vmovss xmm1,real4 ptr [rdx+r11*4-4]     ;xmm1 = x[j - 1]
        vfmadd231ss xmm5,xmm1,[r9+4]            ;xmm5 += x[j - 1] * kernel[1]

        vmovss xmm0,real4 ptr [rdx+r11*4-8]     ;xmm0 = x[j - 2]
        vfmadd231ss xmm4,xmm0,[r9+8]            ;xmm4 += x[j - 2] * kernel[2]

        vmovss xmm1,real4 ptr [rdx+r11*4-12]    ;xmm1 = x[j - 3]
        vfmadd231ss xmm5,xmm1,[r9+12]           ;xmm5 += x[j - 3] * kernel[3]

        vmovss xmm0,real4 ptr [rdx+r11*4-16]    ;xmm0 = x[j - 4]
        vfmadd231ss xmm4,xmm0,[r9+16]           ;xmm4 += x[j - 4] * kernel[4]

        vaddps xmm4,xmm4,xmm5
        vmovss real4 ptr [rcx+rax*4],xmm4       ;save y[i]

        inc rax                                 ;i += 1
        cmp rax,r8
        jl @B                                   ;переход, если i < num_pts

        mov eax,1                               ;возвращаемый код успешного выполнения

Done:   vzeroupper
        ret
Convolve1Ks5_ endp
        end
```

Код C++ в листинге 11.1 начинается с файла заголовка `Ch11_01.h`, который содержит необходимые объявления функций для этого примера. Далее следует исходный код функции `CreateSignal`. Эта функция создает синтетический входной сигнал для тестовых целей. Данный сигнал состоит из трех отдельных синусоидальных сигналов, которые суммируются. Каждая полуволна содержит небольшое количество случайного шума. Входной сигнал, генерируемый функцией `CreateSignal`, – это тот же сигнал, который показан на верхнем графике рис. 11.1.

При выполнении сверток часто необходимо дополнить массив входных сигналов заполняющими элементами, чтобы избежать недействительных обращений к памяти, когда центральная точка ядра свертки накладывается на элементы массива входных сигналов, расположенные рядом с началом и концом массива. Функция `PadSignal` создает дополненную копию массива входных сигналов `x1`, отражая краевые элементы `x1` и сохраняя эти элементы вместе с исходными элементами массива входных сигналов в `x2`. На рис. 11.3 показан пример массива входных сигналов с дополнениями, совместимого с пятиэлементным сверточным ядром. Обратите внимание, что `n2`, размер заполненного буфера, должен равняться `n1+ks2*2`, где `n1` представляет количество элементов массива входных сигналов в `x1`, а `ks2` соответствует `floor(kernel_size / 2)`.

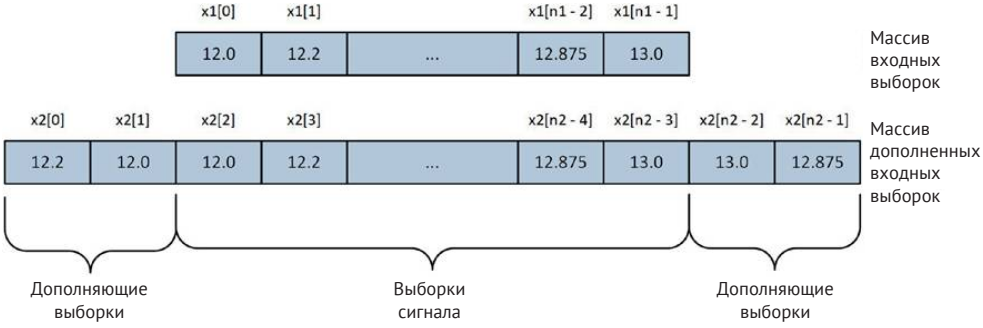


Рис. 11.3. Заполненный массив входных сигналов после выполнения `PadSignal` с использованием пятиэлементного ядра свертки

Функция на языке C++ `Convolve1` содержит код, который выполняет несколько различных реализаций алгоритма дискретной свертки. В верхней части этой функции находится массив с плавающей запятой одинарной точности с именем `kernel`, который содержит коэффициенты ядра свертки. Коэффициенты в ядре представляют собой дискретное приближение фильтра Гаусса (или фильтра нижних частот). При свертке с массивом входного сигнала эти коэффициенты уменьшают количество шума, присутствующего в сигнале, как показано на нижнем графике рис. 11.1. Затем создается дополненный массив входного сигнала `x2` с использованием ранее описанных функций `CreateSignal` и `PadSignal`. Обратите внимание, что класс шаблона C++ `unique_ptr<>` используется для управления пространством хранения как для `x2`, так и для незаполненного массива `x1`.

После генерации массива входного сигнала `x2` функция `Convolve1` выделяет пространство для хранения четырех массивов выходных сигналов. Затем вызываются функции, реализующие различные варианты алгоритмов свертки.

Первые две функции, Convolve1Cpp и Convolve1_, содержат код C++ и на языке ассемблера, который выполняет свои свертки с использованием техники вложенного цикла for, описанной ранее в этом разделе. Функции Convolve1Ks5Cpp и Convolve1Ks5_ оптимизированы для ядер свертки, содержащих пять элементов. В реальных программах обработки сигналов регулярно используются функции свертки, оптимизированные для определенных размеров ядра, поскольку они, как вы скоро увидите, зачастую значительно быстрее.

Функция Convolve1Cpp начинает свое выполнение с проверки значений аргументов kernel_size и num_pts. Следующий оператор, x += ks2, настраивает указатель массива входных сигналов так, чтобы он указывал на первый истинный элемент массива входных сигналов. Напомним, что массив x входного сигнала дополняется дополнительными значениями для обеспечения правильной обработки, когда ядро свертки накладывается на первые два и последние два элемента входного сигнала. После корректировки указателя x следует фактический код, выполняющий свертку. Вложенные циклы for реализуют уравнение дискретной свертки, описанное ранее в этом разделе. Обратите внимание, что значение индекса, используемое для ядра, смещено на ks2 для учета отрицательных индексов внутреннего цикла. Следом за Convolve1Cpp идет функция Convolve1Ks5Cpp, которая использует явные операторы C++ для вычисления суммы произведений свертки вместо цикла for.

Функции Convolve1_ и Convolve1Ks5_ являются ассемблерными аналогами функций Convolve1Cpp и Convolve1Ks5Cpp соответственно. После пролога Convolve1_ проверяет значения аргументов kernel_size и num_pts. За этим следует блок кода инициализации, который начинается с загрузки num_pts в R8. Следующая команда shr r10d,1 загружает ks2 в регистр R10D. Последняя команда блока инициализации, lea rdx, [rdx+r10*4], загружает регистр RDX с адресом первого элемента массива входных сигналов в x.

По аналогии с кодом C++, функция Convole1_ использует для выполнения свертки два вложенных цикла for. Внешний цикл, обозначенный как LP1, начинается с команды vxorps xmm5,xmm5,xmm5, которая устанавливает начальную сумму равной 0,0. Последующие команды mov r11,r10 и neg r11 устанавливают счетчик индекса внутреннего цикла k (R11) на -ks2. Метка LP2 отмечает начало внутреннего цикла. Команды mov rbx,rax и sub rbx,r11 вычисляют индекс, или i-k, следующего элемента в x. За ними следует команда vmovss xmm0,real4 ptr [rdx+rbx*4], которая загружает x[i-k] в XMM0. Затем команды mov rsi,r11 и add rsi,r10 вычисляют k+ks2. Последующая команда vfmadd231ss xmm5,xmm0,[r9+rsi*4] вычисляет sum += x[i-k]*kernel[k+ks2]. Как было сказано в главе 8, команда FMA vfmadd231ss выполняет свои операции с использованием единственной операции округления. В зависимости от алгоритма использование команды vfmadd231ss вместо эквивалентной последовательности команд vmulss и vaddss может привести к несколько более быстрому выполнению.

После выполнения команды vfmadd231ss команда add r11,1 вычисляет k++, и внутренний цикл повторяется до тех пор, пока k>ks2 не станет истинным. После завершения внутреннего цикла команда vmovss real4 ptr [rcx rax*4],xmm5 сохраняет текущий результат суммы произведений в y[i]. Команда add rax,1 обновляет счетчик индекса i, и внешний цикл LP1 повторяется до тех пор, пока не будут обработаны все выборки входного сигнала.

Функция языка ассемблера `Convolve1Ks5_` оптимизирована по размеру для ядер свертки, содержащих пять элементов. Эта функция заменяет внутренний цикл, который использовался в `Convolve1_`, пятью прямыми командами `vfmadd231ss`. Обратите внимание, что эта последовательность команд FMA использует два отдельных регистра, XMM4 и XMM5, для промежуточных сумм. Большинство процессоров Intel, поддерживающих AVX2 и FMA, могут одновременно выполнять две скалярные команды FMA, что увеличивает общую производительность алгоритма. Использование здесь единственного регистра суммы привело бы к ухудшению производительности, поскольку каждая команда `vfmadd231ss` должна была бы завершить свою операцию до того, как могла бы начаться следующая. Вы узнаете больше о зависимостях данных на уровне команд и исполняющих модулях FMA в главе 15. Вот результат выполнения примера исходного кода `Ch11-01`:

```
Htpekmnfns Convolve1

  rc1 = true
  rc2 = true
  rc3 = true
  rc4 = true

Результаты свертки сохранены в файл Ch11_01_Convolve1Results.csv
Запуск измерения быстродействия Convolve1_BM - подождите
Результаты измерения сохранены в файл Ch11_01_Convolve1_BM_CHROMIUM.csv
```

Таблица 11.1 содержит среднее время выполнения функций свертки, представленных в этом разделе. Как упоминалось ранее, оптимизированные по размеру функции свертки `Convolve1Ks5Cpp` и `Convolve1Ks5_` работают значительно быстрее, чем их независимые от размера аналоги. Обратите внимание, что производительность функции C++ `Convolve1Cpp` несколько лучше, чем ее эквивалента на языке ассемблера `Convolve1_`. Причина этого в том, что компилятор Visual C++ сгенерировал код, который частично развернул внутренний цикл и заменил его серией последовательных скалярных команд умножения и сложения с плавающей запятой одинарной точности. Я мог бы легко реализовать ту же технику оптимизации в функции `Convolve1_`, но это повысит производительность всего на несколько процентных пунктов. Для достижения максимальной производительности FMA функция свертки на языке ассемблера должна использовать упакованные команды FMA вместо скалярных. Вы увидите пример этого вычисления в следующем разделе.

Таблица 11.1. Среднее время выполнения (мс) для функций свертки с использованием пятиэлементного ядра свертки (2 000 000 значений выборок)

Процессор	Convolve1Cpp	Convolve1_	Convolve1Ks5Cpp	Convolve1Ks5_
i7-4790S	6148	6844	2926	2841
i9-7900X	5607	6072	2808	2587
i7-8700K	5149	5576	2539	2394

11.1.3. Операции FMA с упакованными операндами

Функции свертки, описанные в предыдущем разделе, можно использовать с небольшими массивами сигналов. Однако во многих реальных приложениях свертки часто выполняются с применением массивов сигналов, содержащих тысячи или миллионы точек данных. В случае необходимости работы с большими массивами основной алгоритм свертки может быть адаптирован для использования упакованных операндов FMA вместо скалярных. В листинге 11.2 приведен исходный код примера Ch11_02, который демонстрирует реализацию дискретной свертки с применением команд FMA для упакованных операндов.

Листинг 11.2. Пример Ch11_02

```
//-----------------------------------------------
//              Ch11_02.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <fstream>
#include "Ch11_02.h"
#include "AlignedMem.h"

using namespace std;

extern "C" const int c_NumPtsMin = 32;
extern "C" const int c_NumPtsMax = 16 * 1024 * 1024;
extern "C" const int c_KernelSizeMin = 3;
extern "C" const int c_KernelSizeMax = 15;
unsigned int g_RngSeedVal = 97;

void Convolve2(void)
{
    const int n1 = 512;
    const float kernel[] { 0.0625f, 0.25f, 0.375f, 0.25f, 0.0625f };
    const int ks = sizeof(kernel) / sizeof(float);
    const int ks2 = ks / 2;
    const int n2 = n1 + ks2 * 2;
    const unsigned int alignment = 32;

    // Создание массива входных выборок
    AlignedArray<float> x1_aa(n1, alignment);
    AlignedArray<float> x2_aa(n2, alignment);
    float* x1 = x1_aa.Data();
    float* x2 = x2_aa.Data();

    CreateSignal(x1, n1, ks, g_RngSeedVal);
    PadSignal(x2, n2, x1, n1, ks2);

    // Выполнение свертки
    AlignedArray<float> y5_aa(n1, alignment);
    AlignedArray<float> y6_aa(n1, alignment);
    AlignedArray<float> y7_aa(n1, alignment);
    float* y5 = y5_aa.Data();
```

```
    float* y6 = y6_aa.Data();
    float* y7 = y7_aa.Data();

    bool rc5 = Convolve2_(y5, x2, n1, kernel, ks);
    bool rc6 = Convolve2Ks5_(y6, x2, n1, kernel, ks);
    bool rc7 = Convolve2Ks5Test_(y7, x2, n1, kernel, ks);

    cout << "Результаты Convolve2\n";
    cout << "  rc5 = " << boolalpha << rc5 << '\n';
    cout << "  rc6 = " << boolalpha << rc6 << '\n';
    cout << "  rc7 = " << boolalpha << rc7 << '\n';

    if (!rc5 || !rc6 || !rc7)
        return;

    // Сохранение данных
    const char* fn = "Ch11_02_Convolve2Results.csv";
    ofstream ofs(fn);

    if (ofs.bad())
        cout << "Ошибка создания файла - " << fn << '\n';
    else
    {
        const char* delim = ", ";

        ofs << fixed << setprecision(7);
        ofs << "i, x1, y5, y6, y7\n";

        for (int i = 0; i < n1; i++)
        {
            ofs << setw(5) << i << delim;
            ofs << setw(10) << x1[i] << delim;
            ofs << setw(10) << y5[i] << delim;
            ofs << setw(10) << y6[i] << delim;
            ofs << setw(10) << y7[i];

            if (y6[i] != y7[i])
                ofs << delim << '*';

            ofs << '\n';
        }

        ofs.close();
        cout << "\nРезультаты сохранены в файл " << fn << '\n';
    }
}

int main()
{
    int ret_val = 1;

    try
    {
        Convolve2();
        Convolve2_BM();
        ret_val = 0;
    }
```

```
    catch (runtime_error& rte)
    {
        cout << "аварийное завершение программы\n";
        cout << rte.what() << '\n';
    }

    catch (...)
    {
        cout << "Возникла нераспознанная ошибка\n";
    }

    return ret_val;
}

;-----------------------------------------------
;               Ch11_02_.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>
        extern c_NumPtsMin:dword
        extern c_NumPtsMax:dword
        extern c_KernelSizeMin:dword
        extern c_KernelSizeMax:dword

; extern bool Convolve2_(float* y, const float* x, int num_pts, const float* kernel, int
kernel_size)

        .code
Convolve2_ proc frame
        _CreateFrame CV2_,0,0,rbx
        _EndProlog

; Проверка значений аргументов
        xor eax,eax                         ;возвращаемый код ошибки

        mov r10d,dword ptr [rbp+CV2_OffsetStackArgs]
        test r10d,1
        jz Done                             ;kernel_size четный
        cmp r10d,[c_KernelSizeMin]
        jl Done                             ;kernel_size слишком мал
        cmp r10d,[c_KernelSizeMax]
        jg Done                             ;kernel_size слишком велик

        cmp r8d,[c_NumPtsMin]
        jl Done                             ;num_pts слишком мал
        cmp r8d,[c_NumPtsMax]
        jg Done                             ;num_pts слишком велик
        test r8d,7
        jnz Done                            ;num_pts не кратен 8

        test rcx,1fh
        jnz Done                            ;y неправильно выровнен

; Инициализация переменных цикла свертки
        shr r10d,1                          ;r10 = kernel_size / 2 (ks2)
        lea rdx,[rdx+r10*4]                 ;rdx = x + ks2 (первая выборка данных)
        xor ebx,ebx                         ;i = 0
```

```
; Реализация свертки
LP1:    vxorps ymm0,ymm0,ymm0              ;sum = 0.0
        mov r11,r10                        ;r11 = ks2
        neg r11                            ;k = -ks2

LP2:    mov rax,rbx                            ;rax = i
        sub rax,r11                            ;rax = i - k
        vmovups ymm1,ymmword ptr [rdx+rax*4]   ;загрузка x[i - k]:x[i - k + 7]

        mov rax,r11
        add rax,r10                            ;rax = k + ks2
        vbroadcastss ymm2,real4 ptr [r9+rax*4] ;ymm2 = kernel[k + ks2]
        vfmadd231ps ymm0,ymm1,ymm2             ;ymm0 += x[i-k]:x[i-k+7] * kernel[k+ks2]

        add r11,1                              ;k += 1
        cmp r11,r10
        jle LP2                                ;повтор, пока не станет k > ks2

        vmovaps ymmword ptr [rcx+rbx*4],ymm0   ;сохранение y[i]:y[i + 7]

        add rbx,8                          ;i += 8
        cmp rbx,r8
        jl LP1                             ;повтор до завершения
        mov eax,1                          ;возвращаемый код успешного выполнения

Done:   vzeroupper
        _DeleteFrame rbx
        ret
Convolve2_ endp

; extern bool Convolve2Ks5_(float* y, const float* x, int num_pts, const float* kernel, int
kernel_size)

Convolve2Ks5_ proc frame
        _CreateFrame CKS5_,0,48
        _SaveXmmRegs xmm6,xmm7,xmm8
        _EndProlog

; Проверка значений аргументов
        xor eax,eax                        ;код ошибки

        cmp dword ptr [rbp+CKS5_OffsetStackArgs],5
        jne Done                           ;переход, если kernel_size не равен 5

        cmp r8d,[c_NumPtsMin]
        jl Done                            ;переход, если num_pts слишком мал
        cmp r8d,[c_NumPtsMax]
        jg Done                            ;переход, если num_pts слишком велик
        test r8d,7
        jnz Done                           ;num_pts не кратен 8

        test rcx,1fh
        jnz Done                           ;y неправильно выровнен

; Выполнение необходимой инициализации
        vbroadcastss ymm4,real4 ptr [r9]       ;kernel[0]
```

```
        vbroadcastss ymm5,real4 ptr [r9+4]      ;kernel[1]
        vbroadcastss ymm6,real4 ptr [r9+8]      ;kernel[2]
        vbroadcastss ymm7,real4 ptr [r9+12]     ;kernel[3]
        vbroadcastss ymm8,real4 ptr [r9+16]     ;kernel[4]
        mov r8d,r8d                             ;r8 = num_pts
        add rdx,8                               ;x += 2

; Реализация свертки
@@:     vxorps ymm2,ymm2,ymm2                   ;инициализация переменных суммы
        vxorps ymm3,ymm3,ymm3
        mov r11,rax
        add r11,2                               ;j = i + ks2

        vmovups ymm0,ymmword ptr [rdx+r11*4]    ;ymm0 = x[j]:x[j + 7]
        vfmadd231ps ymm2,ymm0,ymm4              ;ymm2 += x[j]:x[j + 7] * kernel[0]

        vmovups ymm1,ymmword ptr [rdx+r11*4-4]  ;ymm1 = x[j - 1]:x[j + 6]
        vfmadd231ps ymm3,ymm1,ymm5              ;ymm3 += x[j - 1]:x[j + 6] * kernel[1]

        vmovups ymm0,ymmword ptr [rdx+r11*4-8]  ;ymm0 = x[j - 2]:x[j + 5]
        vfmadd231ps ymm2,ymm0,ymm6              ;ymm2 += x[j - 2]:x[j + 5] * kernel[2]

        vmovups ymm1,ymmword ptr [rdx+r11*4-12] ;ymm1 = x[j - 3]:x[j + 4]
        vfmadd231ps ymm3,ymm1,ymm7              ;ymm3 += x[j - 3]:x[j + 4] * kernel[3]

        vmovups ymm0,ymmword ptr [rdx+r11*4-16] ;ymm0 = x[j - 4]:x[j + 3]
        vfmadd231ps ymm2,ymm0,ymm8              ;ymm2 += x[j - 4]:x[j + 3] * kernel[4]

        vaddps ymm0,ymm2,ymm3                   ;финальные значения
        vmovaps ymmword ptr [rcx+rax*4],ymm0    ;сохранение y[i]:y[i + 7]

        add rax,8                               ;i += 8
        cmp rax,r8
        jl @B                                   ;переход, если i < num_pts
        mov eax,1                               ;возвращаемый код успешного выполнения

Done:   vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame
        ret
Convolve2Ks5_ endp
        end
```

Функции свертки в примере исходного кода `Ch11_01` использовали массивы сигналов с плавающей запятой одинарной точности и ядра свертки. Напомню, что регистр YMM шириной 256 бит может содержать восемь значений с плавающей запятой одинарной точности, а это означает, что реализация алгоритма свертки средствами SIMD может выполнять восемь вычислений FMA одновременно. На рис. 11.4 показаны две схемы, которые иллюстрируют ядро свертки из пяти элементов вместе с произвольным сегментом массива выборок входного сигнала. Ниже этих схем приведены уравнения, которые можно использовать для свертки восьми точек входного сигнала `f[i]:f[i+7]` с помощью пятиэлементного ядра свертки. Эти уравнения представляют собой простое разложение уравнения дискретной свертки, которое обсуждалось ранее в этом разделе. Обратите

внимание, что каждый столбец набора уравнений свертки SIMD включает в себя одно значение ядра и восемь последовательных элементов из массива входных сигналов. Это означает, что функция свертки SIMD может быть легко реализована с помощью широковещательной передачи данных, присвоения упакованных чисел и команд FMA для упакованных операндов. Скоро вы это увидите.

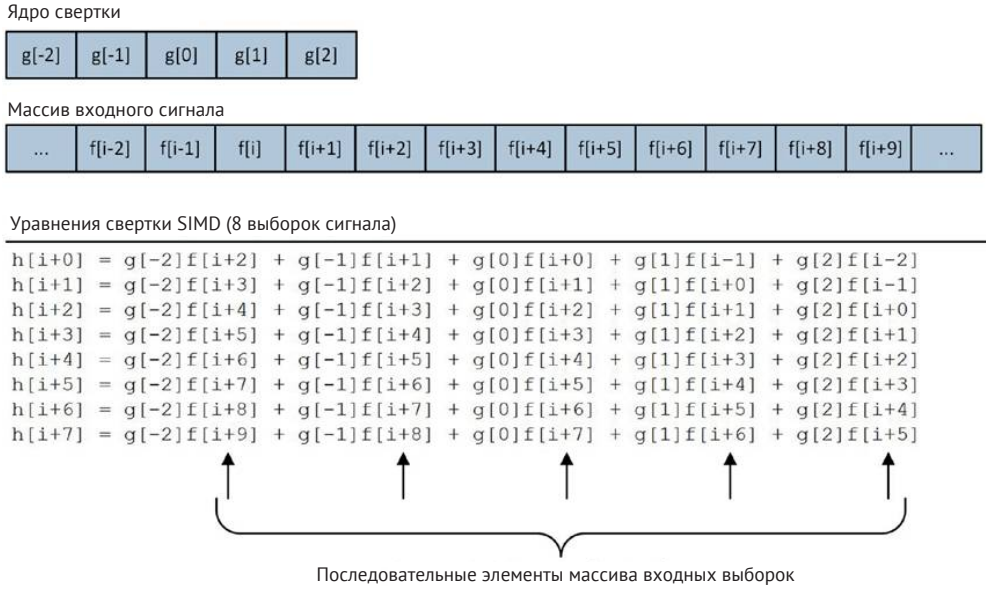


Рис. 11.4. Уравнения свертки SIMD для пятиэлементного ядра свертки

Функция на языке C++ `Convolve2` расположена в верхней части листинга 11.2. Эта функция создает и инициализирует дополненный массив входных сигналов `x2`, используя те же функции `CreateSignal` и `PadSignal`, которые применены в примере `Ch11_01`. Она также использует класс шаблона C++ `AlignedArray` для выделения памяти под массивы выходных сигналов `y5` и `y6`. В этом примере массивы выходных сигналов должны быть правильно выровнены, поскольку функции языка ассемблера используют команду `vmovaps` для сохранения вычисленных результатов. Правильное выравнивание массивов входных сигналов `x1` и `x2` является необязательным и применяется здесь для единообразия. После выделения места под массивы сигналов `Convolve2` вызывает функции на языке ассемблера `Convolve2_` и `ConvolveKs5_`; затем она сохраняет данные массива выходного сигнала в файл CSV.

Функция `Convolve2_` начинает свою работу с проверки значений аргументов `kernel_size` и `num_pts`. Она также проверяет, правильно ли выровнен массив выходного сигнала `y`. Блок кода свертки `Convolve2_` использует ту же вложенную конструкцию цикла `for`, которая применялась в примере `Ch11_01`. Каждая итерация внешнего цикла `LP1` начинается с команды `vxorps ymm0,ymm0,ymm0`, которая инициализирует нулями восемь значений суммы с плавающей запятой одинарной точности. Последующие команды `mov r11,r10` и `neg r11` инициализируют `k` значением `-ks2`. Внутренний цикл начинается с вычисления и сохранения `i-k` в регистре RAX. Последующая команда `vmovups ymm1,ymmword ptr [rdx+rax*4]` загружает элементы массива входного сигнала `x[i-k]:x[i-k+7]` в регистр YMM1.

За этим следует команда vbroadcastss ymm2,real4 ptr [r9+rax*4], которая передает ядро [k+ks2] на каждую позицию элемента с плавающей запятой одинарной точности в YMM2. Команда vfmadd231ps ymm0,ymm1,ymm2 умножает каждый элемент массива входных сигналов в YMM1 на ядро [k+ks2] и добавляет этот промежуточный результат к упакованным суммам в регистре YMM0. Внутренний цикл повторяется до тех пор, пока условие k> ks2 не станет истинным. После завершения внутреннего цикла команда vmovaps ymmword ptr [rcx+rbx*4],ymm0 сохраняет восемь результатов свертки в элементы массива выходного сигнала y[i]:y[i+7]. Затем обновляется счетчик индекса внешнего цикла i, и цикл повторяется до тех пор, пока все элементы входного сигнала не будут обработаны.

Функция языка ассемблера Convolve2Ks5_ оптимизирована для ядер с пятиэлементной сверткой. После необходимых проверок аргументов серия команд vbroadcastss загружает коэффициенты ядра kernel[0]-kernel[4] в регистры YMM4–YMM8 соответственно. Две команды vxorps, расположенные в верхней части цикла обработки, инициализируют нулями промежуточные упакованные суммы. Затем вычисляется индекс массива j=i+ks2, результат сохраняется в регистре R11. Следующая команда vmovups ymm0,ymmword ptr [rdx+r11*4] загружает элементы массива входного сигнала x[j]:x[j+7] в регистр YMM0. За ней следует команда vfmadd231ps ymm2,ymm0,ymm4, которая умножает каждый элемент массива входных сигналов в YMM0 на kernel[0]; затем она прибавляет эти значения к промежуточным упакованным суммам в YMM2. Далее Convolve2Ks5_ использует четыре дополнительных набора команд vmovups и vfmadd231ps для вычисления результатов с использованием коэффициентов kernel[1]-kernel[4]. Подобно функции Convolve1Ks5_ в примере исходного кода Ch11_01, функция Convolve2Ks5_ также использует два регистра YMM для своих промежуточных сумм FMA, что облегчает одновременное выполнение двух команд vfmadd231ps на процессорах с двумя исполнительными блоками FMA шириной 256 бит. После операций FMA команда vmovaps ymmword ptr [rcx+rax*4],ymm0 сохраняет восемь элементов массива выходного сигнала в y[i]:y[i+7].

Код примера Ch11_02 на языке ассемблера также включает функцию с именем Convolve2Ks5Test_. Эта функция заменяет все вхождения команды vfmadd231ps эквивалентной последовательностью команд vmulps и vaddps для целей тестирования и сравнения быстродействия, о чем мы вскоре поговорим. Исходный код функции Convolve2Ks5Test_ не показан в листинге 11.2, но включен в пакет файлов для скачивания. Результаты выполнения примера исходного кода Ch11_02 выглядят следующим образом:

```
Results for Convolve2

  rc5 = true
  rc6 = true
  rc7 = true

Результаты сохранены в файл Ch11_02_Convolve2Results.csv

Выполняется измерение быстродействия Convolve2_BM - подождите
Результаты измерения сохранены в файл Ch11_02_Convolve2_BM_CHROMIUM.csv
```

В табл. 11.2 показаны результаты измерения быстродействия для реализаций функций свертки средствами SIMD. Как и ожидалось, версии SIMD работают значительно быстрее, чем их аналоги без SIMD (см. табл. 11.1). Среднее время выполнения функций свертки с пятиэлементным ядром `Convolve2Ks5_` и `Convolve2Ks5Test_` практически одинаково.

Таблица 11.2. Среднее время выполнения (мс) для функций свертки SIMD с использованием пятиэлементного ядра свертки (2 000 000 выборок сигнала)

Процессор	Convolve2_	Convolve2Ks5_	Convolve2Ks5Test_
i7-4790S	1244	1067	1074
i9-7900X	956	719	709
i7-8700K	859	595	597

Нередко между функцией, использующей команды FMA, и эквивалентной функцией, которая применяет отдельные команды умножения и сложения, возникают небольшие расхождения значений. Это подтверждается сравнением результатов работы функций `Convolve2Ks5_` и `Convolve2Ks5Test_`. Таблица 11.3 показывает несколько примеров расхождений значений из выходного файла `Ch11_02_Convolve2Results.csv`. В реальном приложении эти расхождения значений, скорее всего, будут несущественными. Тем не менее потенциальные расхождения в значениях – это то, о чем вы всегда должны помнить, если разрабатываете производственный код, который включает как FMA, так и не-FMA версии одной и той же функции. Особенно это касается функций, которые выполняют множество операций FMA.

Таблица 11.3. Примеры расхождений значений с использованием последовательностей команд версий FMA и не-FMA

Индекс	x[]	Convolve2Ks5_	Convolve2Ks5Test_
33	1.3856432	1.1940877	1.1940879
108	1.3655651	1.4466031	1.4466029
180	–2.8778596	–2.7348523	–2.7348526
277	–1.7654022	–2.0587211	–2.0587208
403	2.0683382	2.0299273	2.0299270

11.2. Команды для работы с регистрами общего назначения

Как было сказано в главе 8, в последние годы в платформу x86 было добавлено несколько расширений набора команд, оперирующих регистрами общего назначения (см. табл. 8.2 и 8.5). В этом разделе вы узнаете, как использовать некоторые из этих команд. Первый пример исходного кода показывает, как использовать команды умножения и сдвига без флагов. Бесфлаговая команда выполняет свою операцию без изменения каких-либо флагов состояния

в RFLAGS, что может быть быстрее, чем эквивалентная флаговая команда, в зависимости от конкретного варианта использования. Второй пример исходного кода демонстрирует несколько расширенных команд манипулирования битами. Примеры исходного кода в этом разделе требуют наличия процессора, который поддерживает расширения набора команд BMI1, BMI2 и LZCNT.

11.2.1. Бесфлаговое умножение и сдвиги

В листинге 11.3 показан исходный код примера `Ch11_03`. В этом примере показано, как использовать команду умножения беззнаковых целых чисел без флага `mulx`. В нем также продемонстрировано использование команд сдвига без флагов `sarx`, `shlx` и `shrx`.

Листинг 11.3. Пример Ch11_03

```
//-----------------------------------------------
//              Ch11_03.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#include <sstream>

using namespace std;

#include "stdafx.h"

extern "C" uint64_t GprMulx_(uint32_t a, uint32_t b, uint64_t flags[2]);
extern "C" void GprShiftx_(uint32_t x, uint32_t count, uint32_t results[3], uint64_t
flags[4]);

string ToString(uint64_t flags)
{
    ostringstream oss;

    oss << "OF=" << ((flags & (1ULL << 11)) ? '1' : '0') << ' ';
    oss << "SF=" << ((flags & (1ULL <<  7)) ? '1' : '0') << ' ';
    oss << "ZF=" << ((flags & (1ULL <<  6)) ? '1' : '0') << ' ';
    oss << "PF=" << ((flags & (1ULL <<  2)) ? '1' : '0') << ' ';
    oss << "CF=" << ((flags & (1ULL <<  0)) ? '1' : '0') << ' ';

    return oss.str();
}

void GprMulx(void)
{
    const int n = 3;
    uint32_t a[n] = {64, 3200, 100000000};
    uint32_t b[n] = {1001, 12, 250000000};

    cout << "\nРезультат AvxGprMulx\n";

    for (int i = 0; i < n; i++)
```

```
    {
        uint64_t flags[2];
        uint64_t c = GprMulx_(a[i], b[i], flags);

        cout << "\nКонтрольный пример " << i << '\n';
        cout << "  a: " << a[i] << "  b: " << b[i] << "  c: " << c << '\n';

        cout << setfill ('0') << hex;
        cout << "  флаги состояния перед mulx: " << ToString(flags[0]) << '\n';
        cout << "  флаги состояния после mulx:  " << ToString(flags[1]) << '\n';
        cout << setfill (' ') << dec;
    }
}

void GprShiftx(void)
{
    const int n = 4;
    uint32_t x[n] = { 0x00000008, 0x80000080, 0x00000040, 0xfffffc10 };
    uint32_t count[n] = { 2, 5, 3, 4 };

    cout << "\nРезультаты AvxGprShiftx\n";

    for (int i = 0; i < n; i++)
    {
        uint32_t results[3];
        uint64_t flags[4];

        GprShiftx_(x[i], count[i], results, flags);

        cout << setfill(' ') << dec;
        cout << "\nКонтрольный пример " << i << '\n';

        cout << setfill('0') << hex << "  x:    0x" << setw(8) << x[i] << " (";
        cout << setfill(' ') << dec << x[i] << ") count: " << count[i] << '\n';

        cout << setfill('0') << hex << "  sarx: 0x" << setw(8) << results[0] << " (";
        cout << setfill(' ') << dec << results[0] << ")\n";

        cout << setfill('0') << hex << "  shlx: 0x" << setw(8) << results[1] << " (";
        cout << setfill(' ') << dec << results[1] << ")\n";

        cout << setfill('0') << hex << "  shrx: 0x" << setw(8) << results[2] << " (";
        cout << setfill(' ') << dec << results[2] << ")\n";

        cout << "  флаги состояния после сдвигов: " << ToString(flags[0]) << '\n';
        cout << "  флаги состояния после sarx:    " << ToString(flags[1]) << '\n';
        cout << "  флаги состояния после shlx:    " << ToString(flags[2]) << '\n';
        cout << "  флаги состояния после shrx:    " << ToString(flags[3]) << '\n';
    }
}

int main()
{
    GprMulx();
    GprShiftx();
    return 0;
}
```

```
;-----------------------------------------------
;              Ch11_03_.asm
;-----------------------------------------------

; extern "C" uint64_t GprMulx_(uint32_t a, uint32_t b, uint64_t flags[2]);
;
; Requires      BMI2

        .code
GprMulx_ proc

; Сохранение копии флагов состояния перед умножением
        pushfq
        pop rax
        mov qword ptr [r8],rax              ;сохранение исходных флагов

; Бесфлаговое умножение. Команда mulx вычисляет произведение
; прямого операнда-источника ecx (a) и неявного операнда-приемника edx (b).
; 64-битовый результат сохраняется в регистровую пару r11d:r10d.
        mulx r11d,r10d,ecx                  ;r11d:r10d = a * b

; Сохранение копии флагов состояния после mulx
        pushfq
        pop rax
        mov qword ptr [r8+8],rax            ;сохранение флагов состояния после mulx

; Отправка 64-битного результата в rax
        mov eax,r10d
        shl r11,32
        or rax,r11
        ret
GprMulx_ endp

; extern "C" void GprShiftx_(uint32_t x, uint32_t count, uint32_t results[3], uint64_t
flags[4])
;
; Requires      BMI2

GprShiftx_ proc

; Сохранение копии флагов состояния перед сдвигами
        pushfq
        pop rax
        mov qword ptr [r9],rax              ;сохранение исходных флагов состояния

; Загрузка аргументов и выполнение сдвигов. Каждая команда сдвига
; использует три операнда: DesOp, SrcOp, и CountOp.

        sarx eax,ecx,edx                    ;арифметический сдвиг вправо
        mov dword ptr [r8],eax
        pushfq
        pop rax
        mov qword ptr [r9+8],rax

        shlx eax,ecx,edx                    ;логический сдвиг влево
        mov dword ptr [r8+4],eax
```

```
        pushfq
        pop rax
        mov qword ptr [r9+16],rax

        shrx eax,ecx,edx                    ;логический сдвиг вправо
        mov dword ptr [r8+8],eax
        pushfq
        pop rax
        mov qword ptr [r9+24],rax

        ret
GprShiftx_ endp
        end
```

Код на языке C++ в примере `Ch11_03` содержит две функции с именами `GprMulx` и `GprShiftx`. Эти функции инициализируют тестовые примеры, демонстрирующие операции умножения и сдвига без флагов соответственно. Обратите внимание, что и `GprMulx`, и `GprShiftx` определяют массив типа `uint64_t` с именем `flags`. Этот массив используется для отображения содержимого флагов состояния в RFLAGS до и после выполнения каждой команды без флагов. Оставшийся код `GprMulx` и `GprShiftx` форматирует и отправляет поток результатов в `cout`.

Функция на языке ассемблера `GprMulx` начинает свою работу с сохранения копии RFLAGS. Далее команда `mulx r11d,r10d,ecx` выполняет 32-битное целочисленное умножение без знака с использованием неявного исходного операнда EDX (значение аргумента `b`) и явного исходного операнда ECX (значение аргумента `a`). Затем 64-битное произведение сохраняется в регистровой паре R11D:R10D. После выполнения команды `mulx` содержимое RFLAGS снова сохраняется для сравнения. Команда `mulx` также поддерживает умножение без флагов с использованием 64-битных операндов. При использовании с 64-битными операндами регистр RDX применяется как неявный операнд, а в качестве операнда-приемника должны использоваться два 64-битных регистра общего назначения.

Функция `GprShiftx_` содержит примеры использования команд `sarx`, `shlx` и `shrx` с 32-битными операндами. Эти команды используют синтаксис с тремя операндами, аналогичный командам AVX. Первый операнд-источник сдвигается на значение счетчика, указанное во втором операнде-источнике. Затем результат сохраняется в операнде-приемнике. Команды сдвига без флагов также могут работать с операндами шириной 64 бита; 8- и 16-битные операнды не поддерживаются. Ниже показан результат выполнения примера исходного кода `Ch11_03`:

```
Результаты AvxGprMulx

Контрольный пример 0
  a: 64  b: 1001  c: 64064
  флаг состояния перед mulx: OF=0 SF=0 ZF=1 PF=1 CF=0
  флаг состояния после mulx: OF=0 SF=0 ZF=1 PF=1 CF=0

Контрольный пример 1
  a: 3200 b: 12 c: 38400
```

```
  флаг состояния перед mulx: OF=0 SF=1 ZF=0 PF=0 CF=1
  флаг состояния после mulx: OF=0 SF=1 ZF=0 PF=0 CF=1

Контрольный пример 2
  a: 100000000 b: 250000000 c: 25000000000000000
  флаг состояния перед mulx: OF=0 SF=1 ZF=0 PF=1 CF=1
  флаг состояния после mulx: OF=0 SF=1 ZF=0 PF=1 CF=1

Результаты AvxGprShiftx

Контрольный пример 0
  x:    0x00000008 (8) count: 2
  sarx: 0x00000002 (2)
  shlx: 0x00000020 (32)
  shrx: 0x00000002 (2)
  флаги состояния после сдвигов: OF=0 SF=0 ZF=1 PF=1 CF=0
  флаги состояния после sarx:    OF=0 SF=0 ZF=1 PF=1 CF=0
  флаги состояния после shlx:    OF=0 SF=0 ZF=1 PF=1 CF=0
  флаги состояния после shrx:    OF=0 SF=0 ZF=1 PF=1 CF=0

Контрольный пример 1
  x:    0x80000080 (2147483776) count: 5
  sarx: 0xfc000004 (4227858436)
  shlx: 0x00001000 (4096)
  shrx: 0x04000004 (67108868)
  флаги состояния после сдвигов: OF=0 SF=1 ZF=0 PF=0 CF=1
  флаги состояния после sarx:    OF=0 SF=1 ZF=0 PF=0 CF=1
  флаги состояния после shlx:    OF=0 SF=1 ZF=0 PF=0 CF=1
  флаги состояния после shrx:    OF=0 SF=1 ZF=0 PF=0 CF=1

Контрольный пример 2
  x:    0x00000040 (64) count: 3
  sarx: 0x00000008 (8)
  shlx: 0x00000200 (512)
  shrx: 0x00000008 (8)
  флаги состояния после сдвигов: OF=0 SF=1 ZF=0 PF=0 CF=1
  флаги состояния после sarx: OF=0 SF=1 ZF=0 PF=0 CF=1
  флаги состояния после shlx: OF=0 SF=1 ZF=0 PF=0 CF=1
  флаги состояния после shrx: OF=0 SF=1 ZF=0 PF=0 CF=1

Контрольный пример 3
  x:    0xfffffc10 (4294966288) count: 4
  sarx: 0xfffffff1 (4294967233)
  shlx: 0xffffc100 (4294951168)
  shrx: 0x0ffffff1 (268435393)
  флаги состояния после сдвигов: OF=0 SF=1 ZF=0 PF=1 CF=1
  флаги состояния после sarx:    OF=0 SF=1 ZF=0 PF=1 CF=1
  флаги состояния после shlx:    OF=0 SF=1 ZF=0 PF=1 CF=1
  флаги состояния после shrx:    OF=0 SF=1 ZF=0 PF=1 CF=1
```

11.2.2. Расширенные манипуляции битами

Большинство команд, включенных в расширения набора команд BMI1 и BMI2, ориентированы на повышение производительности определенных алгоритмов, таких как шифрование и дешифрование данных. Их также можно исполь-

зовать для упрощения битовых операций в более приземленных алгоритмах. Пример исходного кода Ch11_04 содержит три простые функции на языке ассемблера, которые демонстрируют использование команд для работы с битами lzcnt, tzcnt, bextr и andn. В листинге 11.4 показан исходный код на языках C++ и ассемблера для этого примера.

Листинг 11.4. Пример Ch11_04

```
//-----------------------------------------------
//              Ch11_04.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" void GprCountZeroBits_(uint32_t x, uint32_t* lzcnt, uint32_t* tzcnt);
extern "C" uint32_t GprBextr_(uint32_t x, uint8_t start, uint8_t length);
extern "C" uint32_t GprAndNot_(uint32_t x, uint32_t y);

void GprCountZeroBits(void)
{
    const int n = 5;
    uint32_t x[n] = { 0x001000008, 0x00008000, 0x8000000, 0x00000001, 0 };

    cout << "\nРезультаты AvxGprCountZeroBits\n";

    for (int i = 0; i < n; i++)
    {
        uint32_t lzcnt, tzcnt;

        GprCountZeroBits_(x[i], &lzcnt, &tzcnt);

        cout << setfill('0') << hex;
        cout << "x: 0x" << setw(8) << x[i] << "  ";
        cout << setfill(' ') << dec;
        cout << "lzcnt: " << setw(3) << lzcnt << "  ";
        cout << "tzcnt: " << setw(3) << tzcnt << '\n';
    }
}

void GprExtractBitField(void)
{
    const int n = 3;
    uint32_t x[n] = { 0x12345678, 0x80808080, 0xfedcba98 };
    uint8_t start[n] = { 4, 7, 24 };
    uint8_t len[n] = { 16, 9, 8 };

    cout << "\nРезультаты GprExtractBitField\n";

    for (int i = 0; i < n; i++)
    {
        uint32_t bextr = GprBextr_(x[i], start[i], len[i]);
```

```
        cout << setfill('0') << hex;
        cout << "x: 0x" << setw(8) << x[i] << "  ";

        cout << setfill(' ') << dec;
        cout << "start: " << setw(3) << (uint32_t)start[i] << "  ";
        cout << "len:   " << setw(3) << (uint32_t)len[i] << "  ";
        cout << setfill('0') << hex;
        cout << "bextr: 0x" << setw(8) << bextr << '\n';
    }
}

void GprAndNot(void)
{
    const int n = 3;
    uint32_t x[n] = { 0xf000000f, 0xff00ff00, 0xaaaaaaaa };
    uint32_t y[n] = { 0x12345678, 0x12345678, 0xffaa5500 };

    cout << "\nРезультаты GprAndNot\n";

    for (int i = 0; i < n; i++)
    {
        uint32_t andn = GprAndNot_(x[i], y[i]);

        cout << setfill('0') << hex;
        cout << "x: 0x" << setw(8) << x[i] << "  ";
        cout << "y: 0x" << setw(8) << y[i] << "  ";
        cout << "andn: 0x" << setw(8) << andn << '\n';
    }
}

int main()
{
    GprCountZeroBits();
    GprExtractBitField();
    GprAndNot();
    return 0;
}

;-----------------------------------------------
;               Ch11_04_.asm
;-----------------------------------------------

; extern "C" void GprCountZeroBits_(uint32_t x, uint32_t* lzcnt, uint32_t* tzcnt);
;
; Требуется:     BMI1, LZCNT

        .code
GprCountZeroBits_ proc
        lzcnt eax,ecx                       ;подсчет ведущих нулей
        mov dword ptr [rdx],eax             ;сохранение результата

        tzcnt eax,ecx                       ;подсчет замыкающих нулей
        mov dword ptr [r8],eax              ;сохранение результата
        ret
GprCountZeroBits_ endp
```

```
; extern "C" uint32_t GprBextr_(uint32_t x, uint8_t start, uint8_t length);
;
; Требуется:     BMI1

GprBextr_ proc
        mov al,r8b
        mov ah,al                           ;ah = длина
        mov al,dl                           ;al = старт
        bextr eax,ecx,eax                   ;eax = битовое поле (из x)
        ret
GprBextr_ endp

; extern "C" uint32_t GprAndNot_(uint32_t x, uint32_t y);
;
; Требуется:     BMI1

GprAndNot_ proc
        andn eax,ecx,edx                    ;eax = ~x & y
        ret
GprAndNot_ endp
        end
```

Код C++ в листинге 11.4 содержит три короткие функции, которые настраивают контрольные примеры для функций на языке ассемблера. Первая функция, `GprCountZeroBits`, инициализирует тестовый массив, применяемый для демонстрации команд `lzcnt` (подсчет количества ведущих нулевых битов) и `tzcnt` (подсчет количества замыкающих нулевых битов). Вторая функция, `GprExtractBitField`, подготавливает тестовые данные для команды `bextr` (извлечение битового поля). Последняя функция C++ в листинге 11.4 называется `GprAndNot`. Эта функция загружает тестовые массивы с данными, которые используются для демонстрации выполнения команды `andn` (побитовое И-НЕ).

Первая функция на языке ассемблера называется `GprCountZeroBits_`. Эта функция использует команды `lzcnt` и `tzcnt` для подсчета количества начальных и конечных нулевых битов в соответствующих 32-битных исходных операндах. Затем вычисленное количество битов сохраняется в указанном операнде-приемнике. Следующая функция, `GprBextr_`, выполняет команду `bextr`. Первый исходный операнд этой команды содержит данные, из которых будет извлечено битовое поле. Биты 7:0 и 15:8 второго исходного операнда определяют начало поля, битовую позицию и длину соответственно. Наконец, функция `GprAndNot_` демонстрирует использование команды `andn`. Эта команда вычисляет `DesOp = ~ SrcOp1 & SrcOp2` и часто применяется для упрощения операций логического маскирования. Ниже показаны результаты выполнения примера исходного кода `Ch11_04`.

```
Результаты AvxGprCountZeroBits
x: 0x01000008  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
```

```
Результаты GprExtractBitField
x: 0x12345678  start:   4  len:   16  bextr: 0x00004567
x: 0x80808080  start:   7  len:    9  bextr: 0x00000101
x: 0xfedcba98  start:  24  len:    8  bextr: 0x000000fe

Результаты GprAndNot
x: 0xf000000f  y: 0x12345678  andn: 0x02345670
x: 0xff00ff00  y: 0x12345678  andn: 0x00340078
x: 0xaaaaaaaa  y: 0xffaa5500  andn: 0x55005500
```

Расширения набора команд BMI1 и BMI2 также включают другие команды по манипулированию битами, которые можно использовать для реализации определенных алгоритмов или выполнения специализированных операций. Эти команды перечислены в табл. 8.5.

11.3. Преобразования с плавающей запятой половинной точности

Последний пример исходного кода этой главы, `Ch11_05`, иллюстрирует использование команд преобразования операндов половинной точности `vcvtps2ph` и `vcvtph2ps`. В листинге 11.5 показан исходный код этого примера. Если понимание типа данных с плавающей запятой половинной точности вызывает у вас затруднения, вы можете перечитать главу 8, прежде чем изучать исходный код этого раздела и пояснения.

Листинг 11.5. Пример `Ch11_05`

```
//-----------------------------------------------
//               Ch11_05.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <string>
#include <iostream>
#include <iomanip>

using namespace std;

extern "C" void SingleToHalfPrecision_(uint16_t x_hp[8], float x_sp[8], int rc);
extern "C" void HalfToSinglePrecision_(float x_sp[8], uint16_t x_hp[8]);

int main()
{
    float x[8];

    x[0] = 4.125f;
    x[1] = 32.9f;
    x[2] = 56.3333f;
    x[3] = -68.6667f;
    x[4] = 42000.5f;
    x[5] = 75600.0f;
```

```
    x[6] = -6002.125f;
    x[7] = 170.0625f;

    uint16_t x_hp[8];
    float rn[8], rd[8], ru[8], rz[8];

    SingleToHalfPrecision_(x_hp, x, 0);
    HalfToSinglePrecision_(rn, x_hp);
    SingleToHalfPrecision_(x_hp, x, 1);
    HalfToSinglePrecision_(rd, x_hp);
    SingleToHalfPrecision_(x_hp, x, 2);
    HalfToSinglePrecision_(ru, x_hp);
    SingleToHalfPrecision_(x_hp, x, 3);
    HalfToSinglePrecision_(rz, x_hp);

    unsigned int w = 15;
    string line(76, '-');

    cout << fixed << setprecision(4);
    cout << setw(w) << "x";
    cout << setw(w) << "RoundNearest";
    cout << setw(w) << "RoundDown";
    cout << setw(w) << "RoundUp";
    cout << setw(w) << "RoundZero";
    cout << '\n' << line << '\n';

    for (int i = 0; i < 8; i++)
    {
        cout << setw(w) << x[i];
        cout << setw(w) << rn[i];
        cout << setw(w) << rd[i];
        cout << setw(w) << ru[i];
        cout << setw(w) << rz[i];
        cout << '\n';
    }

    return 0;
}

;-----------------------------------------------
;               Ch11_05_.asm
;-----------------------------------------------

; extern "C" void SingleToHalfPrecision_(uint16_t x_hp[8], float x_sp[8], int rc);

        .code
SingleToHalfPrecision_ proc

; Преобразование упакованного числа одинарной точности
; в упакованное число половинной точности
        vmovups ymm0,ymmword ptr [rdx]          ;ymm0 = 8 значений SPFP

        cmp r8d,0
        jne @F
        vcvtps2ph xmm1,ymm0,0                   ;округление до ближайшего
        jmp SaveResult
```

```
@@:     cmp r8d,1
        jne @F
        vcvtps2ph xmm1,ymm0,1                   ;округление вниз
        jmp SaveResult

@@:     cmp r8d,2
        jne @F
        vcvtps2ph xmm1,ymm0,2                   ;округление вверх
        jmp SaveResult

@@:     cmp r8d,3
        jne @F
        vcvtps2ph xmm1,ymm0,3                   ;усечение
        jmp SaveResult

@@:     vcvtps2ph xmm1,ymm0,4                   ;использование MXCSR.RC

SaveResult:
        vmovdqu xmmword ptr [rcx],xmm1          ;сохранение 8 значений HPFP
        vzeroupper
        ret

SingleToHalfPrecision_ endp

; extern "C" void HalfToSinglePrecision_(float x_sp[8], uint16_t x_hp[8]);

HalfToSinglePrecision_ proc

; Преобразование упакованного числа половинной точности в
; упакованное число одинарной точности
        vcvtph2ps ymm0,xmmword ptr [rdx]
        vmovups ymmword ptr [rcx],ymm0          ;сохранение 8 значений SPFP

        vzeroupper
        ret

HalfToSinglePrecision_ endp
        end
```

Функция C++ `main` начинается с загрузки тестовых значений с плавающей запятой одинарной точности в массив `x`. Затем она запускает функции преобразования с плавающей запятой половинной точности `SingleToHalfPrecision_` и `HalfToSinglePrecision_`. Обратите внимание, что для функции `SingleToHalfPrecision_` требуется третий аргумент, который задает режим округления, используемый при преобразовании значений с плавающей запятой одинарной точности в значения половинной точности. Также обратите внимание, что для хранения результатов с плавающей запятой половинной точности используется тип массива `uint16_t`, поскольку C++ изначально не поддерживает тип данных с плавающей запятой половинной точности.

Функция на языке ассемблера `SingleToHalfPrecision_` использует команду `vcvtps2ph` для преобразования восьми значений с плавающей запятой одинарной точности в массиве `x_sp` в значения с плавающей запятой половинной точности. Этой команде нужен явный операнд, который задает режим округления,

используемый во время преобразования типа. В табл. 11.4 показаны параметры режима округления для команды `vcvtps2ph`.

Таблица 11.4. Параметры режима округления для команды `vcvtps2ph`

Биты прямого операнда	Значение	Описание
1:0	00b	Округление до ближайшего
	01b	Округление вниз (в сторону $-\infty$)
	10b	Округление вверх (в сторону $+\infty$)
	11b	Округление в сторону нуля (усечение)
2	0	Использовать режим округления, указанный в битах 1:0
	1	Использовать режим округления, указанный в MXCSR.RC
7:3	Ignored	Не используется

Функция `HalfToSinglePrecision_` использует команду `vcvtph2ps` для преобразования восьми значений с плавающей запятой половинной точности в числа с плавающей запятой одинарной точности. Ниже показан результат выполнения исходного кода `Ch11_05`. Обратите внимание на разницу значений между различными режимами округления. Также обратите внимание, что при использовании режима округления `RoundNearest` или `RoundUp` значение `76000.0f` преобразуется в `inf` (или бесконечность), поскольку эта величина превышает максимально возможное значение с плавающей запятой половинной точности.

```
             x    RoundNearest       RoundDown       RoundUp      RoundZero
--------------------------------------------------------------------------
        4.1250          4.1250          4.1250        4.1250         4.1250
       32.9000         32.9063         32.8750       32.9063        32.8750
       56.3333         56.3438         56.3125       56.3438        56.3125
      -68.6667        -68.6875        -68.6875      -68.6250       -68.6250
    42000.5000      42016.0000      41984.0000    42016.0000     41984.0000
    75600.0000             inf      65504.0000           inf     65504.0000
    -6002.1250      -6004.0000      -6004.0000    -6000.0000     -6000.0000
      170.0625        170.0000        170.0000      170.1250       170.0000
```

11.4. Заключение

В главе 11 изложены следующие ключевые моменты:

- команды FMA часто используются для реализации численно ориентированных алгоритмов, таких как дискретные свертки, которые широко используются в самых различных областях, включая цифровую обработку сигналов и изображений;
- функции на языке ассемблера, в которых задействованы цепочки последовательных команд FMA, должны использовать несколько регистров XMM или YMM для хранения промежуточных сумм. Использование нескольких регистров помогает избежать зависимостей данных, которые

могут препятствовать одновременному выполнению процессором нескольких команд FMA;

- между функциями, реализующими один и тот же алгоритм или операцию с использованием последовательностей команд FMA и не-FMA, обычно возникают расхождения значений. Значимость этих расхождений зависит от приложения;
- функции на языке ассемблера могут использовать команды `mulx`, `sarx`, `shlx` и `shrx` для выполнения бесфлагового беззнакового целочисленного умножения и сдвигов. Эти команды могут обеспечить немного более высокую производительность, чем их флаговые аналоги в алгоритмах, выполняющих последовательные операции умножения и сдвига;
- функции на языке ассемблера могут использовать команды `lzcnt`, `tzcnt`, `bextr` и `andn` для выполнения расширенных операций манипулирования битами;
- функции на языке ассемблера могут использовать команды `vcvtps2ph` и `vcvtph2ps` для выполнения преобразований между значениями с плавающей запятой одинарной и половинной точности.

Глава 12

Система векторных команд AVX-512

В предыдущих главах вы узнали о возможностях AVX и AVX2 в отношении операций над скалярными и упакованными операндами с плавающей запятой и упакованными целочисленными операндами. В этой главе вы узнаете о системе векторных команд Advance Vector Extensions 512 (AVX-512). AVX-512, несомненно, является крупнейшим и, возможно, наиболее значимым расширением платформы x86 на сегодняшний день. Оно удваивает количество доступных регистров SIMD и увеличивает ширину каждого регистра с 256 до 512 бит. AVX-512 также расширяет синтаксис команд AVX и AVX2 для поддержки дополнительных возможностей, недоступных в более ранних расширениях, включая условное выполнение и слияние, встроенное широковещание и управление округлением на уровне команд для операций с плавающей запятой.

Содержание данной главы организовано следующим образом. В первом разделе представлен краткий обзор AVX-512, который включает информацию о различных расширениях набора команд AVX-512. Затем следует подробное описание среды выполнения AVX-512, включая ее наборы регистров, типы данных, синтаксис команд и расширенные вычислительные возможности. Глава завершается синопсисом расширений набора команд AVX-512, которые включены в недавно выпущенные процессоры для серверных платформ и рабочих станций.

12.1. Обзор AVX-512

В отличие от AVX и AVX2, AVX-512 не является отдельным расширением набора команд. Скорее, это согласованный набор взаимосвязанных расширений набора команд. Процессор x86 соответствует стандарту AVX-512, если он поддерживает расширение набора команд AVX512F (фундаментальное). Процессор, соответствующий стандарту AVX-512, может, кроме этого, поддерживать дополнительные расширения набора команд AVX-512, и они различаются в зависимости от целевого сегмента рынка (например, высокопроизводительные вычисления, сервер, настольный компьютер, мобильный телефон и т. д.). В табл. 12.1 перечислены расширения набора команд AVX-512, которые в настоящее время доступны в некоторых процессорах Intel. Эта таблица также

включает расширения набора команд AVX-512, которые Intel анонсировала для будущих процессоров. На момент написания данного текста AMD не продает процессоры, поддерживающие AVX-512.

Описания в этой главе и примеры исходного кода в главах 13 и 14 в основном сосредоточены на расширениях набора команд AVX-512, включенных в микроархитектуру Intel Skylake Server, запущенную в 2017 году. Эта микроархитектура используется в процессорах Intel Xeon Scalable (серверы), Xeon W (рабочие станции) и Core i7-7800X и i9-7900X (настольные ПК высокого класса). Процессоры на основе микроархитектуры Skylake Server поддерживают следующие расширения набора команд AVX-512: AVX512F, AVX512CD, AVX512BW, AVX512DQ и AVX512VL. Ожидается, что будущие массовые процессоры от AMD и Intel будут включать те же расширения AVX-512. В главе 16 объясняется, как использовать команду `cpuid` для проверки доступности расширений набора команд AVX-512, показанных в табл. 12.1.

Таблица 12.1. Обзор расширений набора команд AVX-512

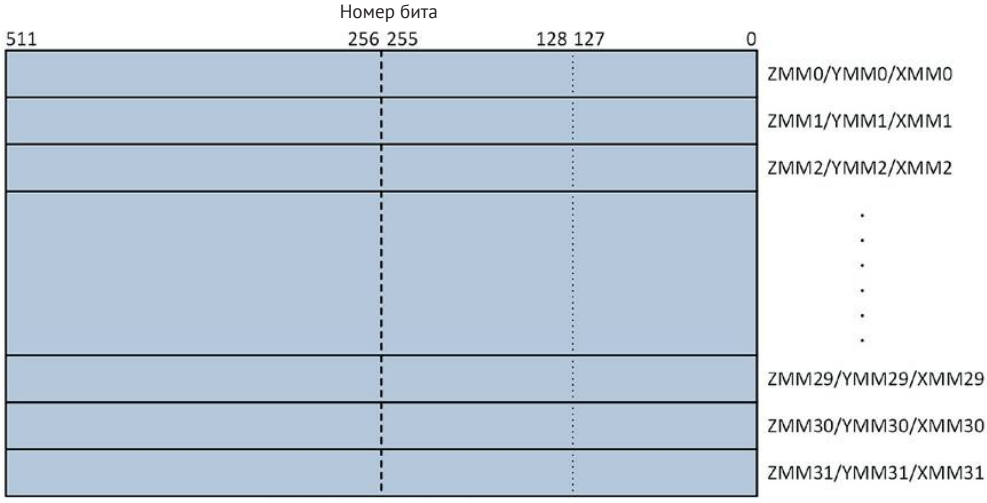
Флаг CPUID	Описание
AVX512F	Базовые команды
AVX512ER	Возведение в степень и обратная величина
AVX512PF	Команды предварительной выборки
AVX512CD	Команды обнаружения конфликта
AVX512DQ	Команды для работы с двойными и четверными словами
AVX512BW	Команды для работы с байтами и словами
AVX512VL	128- и 256-битные векторные команды
AVX512JFMA	Слитное умножение-сложение целых чисел
AVX512.VBMI	Дополнительные векторные байтовые команды
AVX512_4FMAPS	Упакованные операции FMA одинарной точности (4 итерации)
AVX512_4VNNI	Векторные нейросетевые команды (4 итерации)
AVX512_VPOPCNTDQ	Команды vpopcnt[d|q]
AVX512_VNNI	Векторные нейросетевые команды
AVX512.VBMI2	Новые векторные команды (байт, слово, двойное слово, квадрослово)
AVX512_BITALG	Команды vpopcnt[b|w] и vpshufbitqmb

12.2. Среда выполнения AVX-512

Набор AVX-512 расширяет среду исполнения платформы x86 за счет добавления новых регистров и типов данных. Он также расширяет синтаксис команд на языке ассемблера AVX и AVX2 для поддержки расширенных операций, таких как условное выполнение и слияние, встроенные широковещательные рассылки и управление округлением на уровне команд. В данном разделе эти улучшения обсуждаются более подробно.

12.2.1. Наборы регистров

На рис. 12.1 показаны наборы регистров AVX-512. AVX-512 увеличивает ширину каждого регистра AVX SIMD с 256 до 512 бит. Регистры шириной 512 бит известны как набор регистров ZMM. Процессоры с поддержкой AVX-512 имеют 32 регистра ZMM с именами ZMM0-ZMM31. Наборы регистров YMM и XMM привязаны к младшим 256 битам и 128 битам каждого регистра ZMM соответственно. Процессоры AVX-512 также имеют восемь новых регистров *маски операции* (opmask register), обозначаемых K0–K7. Эти регистры в основном используются в качестве масок предикатов для выполнения условных операций и операций слияния. Их также можно использовать в качестве операндов назначения для команд, которые генерируют результаты векторной маски. Я расскажу подробнее об этих регистрах позже в данной главе.



Набор регистров SIMD AVX-512

Набор регистров маски операции AVX-512

Рис. 12.1. Наборы регистров AVX-512

12.2.2.Типы данных

Подобно регистрам YMM и XMM, функции программы могут использовать регистры ZMM для выполнения операций SIMD с использованием упакованных операндов, – как целочисленных, так и с плавающей запятой. В табл. 12.2 показано максимальное количество элементов, которое может содержать регистр ZMM для каждого поддерживаемого типа данных. В этой таблице также для целей сравнения показано максимальное количество элементов, которое могут содержать регистры YMM и XMM.

Таблица 12.2. Максимальное количество элементов для операндов регистра AVX-512

Data Type	ZMM	YMM	XMM
Целочисленный байт	64	32	16
Целочисленное слово	32	16	8
Целочисленное двойное слово	16	8	4
Целочисленное квадрослово	8	4	2
С плавающей запятой одинарной точности	16	8	4
С плавающей запятой двойной точности	8	4	2

Требования к выравниванию для 512-битных операндов в памяти аналогичны другим операндам x86 SIMD. За исключением команд, которые явно указывают выровненный операнд (например, `vmovdqa[32|64]`, `vmovap[d|s]` и т. д.), правильное выравнивание 512-битного операнда в памяти не является обязательным. Однако операнды шириной 512 бит всегда должны быть выровнены по границе 64 байта, когда это возможно, чтобы избежать задержек обработки, которые могут возникнуть, если процессор вынужден обращаться к невыровненному операнду в памяти. Команды AVX-512, которые обращаются к 256-битным или 128-битным операндам в памяти, также должны гарантировать, что эти типы операндов правильно выровнены по их соответствующим естественным границам.

12.2.3. Синтаксис команды

AVX-512 расширяет синтаксис команд AVX и AVX2. Большинство команд AVX-512 могут использовать тот же синтаксис команд с тремя операндами, что и команды AVX и AVX2, который состоит из двух неразрушающих операндов-источников и одного операнда-приемника. Команды AVX-512 также могут использовать несколько новых необязательных операндов. Эти операнды облегчают условное выполнение и слияние, встроенные широковещательные операции и управление округлением с плавающей запятой. В следующих разделах более подробно рассматриваются операнды дополнительных команд AVX-512.

Условное исполнение и слияние

Большинство команд AVX-512 поддерживают условное выполнение и слияние. Операция условного выполнения и слияния использует биты регистра маски операции в качестве маски предиката для управления выполнением команд и обновлением операндов приемника для каждого элемента. Рисунок 12.2 представляет эту концепцию более наглядно. На этом рисунке каждый регистр ZMM0, ZMM1 и ZMM2 содержит по 16 значений с плавающей запятой одинарной точности. 16 младших бит регистра маски операции K1 составляют маску предиката. Когда регистр маски операции используется таким образом, каждый бит маски управляет тем, как вычисляется и обновляется результат соответствующей позиции элемента в операнде-адресате.

На рис. 12.2 также показан результат выполнения трех команд `vaddps` с одинаковыми начальными значениями, но с разным результатом. Первая команда, `vaddps zmm2,zmm0,zmm1`, выполняет сложение упакованных чисел с плавающей запятой одинарной точности элементов в ZMM0 и ZMM1 и сохраняет результирующие суммы в регистре ZMM2. Выполнение этой команды ничем не отличается от выполнения команды AVX `vaddps`, которая использует регистровые операнды XMM или YMM. Следующая команда, `vaddps zmm2{k1},zmm0,zmm1`, иллюстрирует, как биты регистра маски K1 используются для условного сложения и обновления операнда-приемника для каждого элемента. Если точнее, сумма элементов вычисляется и сохраняется в операнде-приемнике только в том случае, если соответствующая битовая позиция регистра маски установлена в единицу; в противном случае элемент операнда-приемника на этой позиции остается неизменным. Это называется *маскированием слиянием*. Последний вариант команды на рис. 12.2, `vaddps zmm2 {k1} {z},zmm0,zmm1`, аналогичен предыдущей команде. Дополнительный операнд `{z}` указывает процессору выполнить *маскирование нулем* вместо маскирования слияния. Нулевое маскирование устанавливает элемент операнда-приемника в ноль, если его соответствующая битовая позиция в регистре маски операции установлена в ноль; в противном случае в этот элемент сохраняется вычисленная сумма.

Здесь уместно сказать несколько слов о регистрах маски операции. Восемь регистров маски чем-то похожи на регистры общего назначения. На процессорах, поддерживающих AVX-512, каждый регистр маски имеет ширину 64 бита. Однако при использовании в качестве маски предиката во время выполнения команды используются только младшие биты. Точное количество используемых младших битов варьируется в зависимости от количества элементов вектора. На рис. 12.2 маску предиката образуют биты 0–15 регистра маски K1, поскольку команда `vaddps` использует операнды регистра ZMM, которые содержат 16 значений с плавающей запятой одинарной точности.

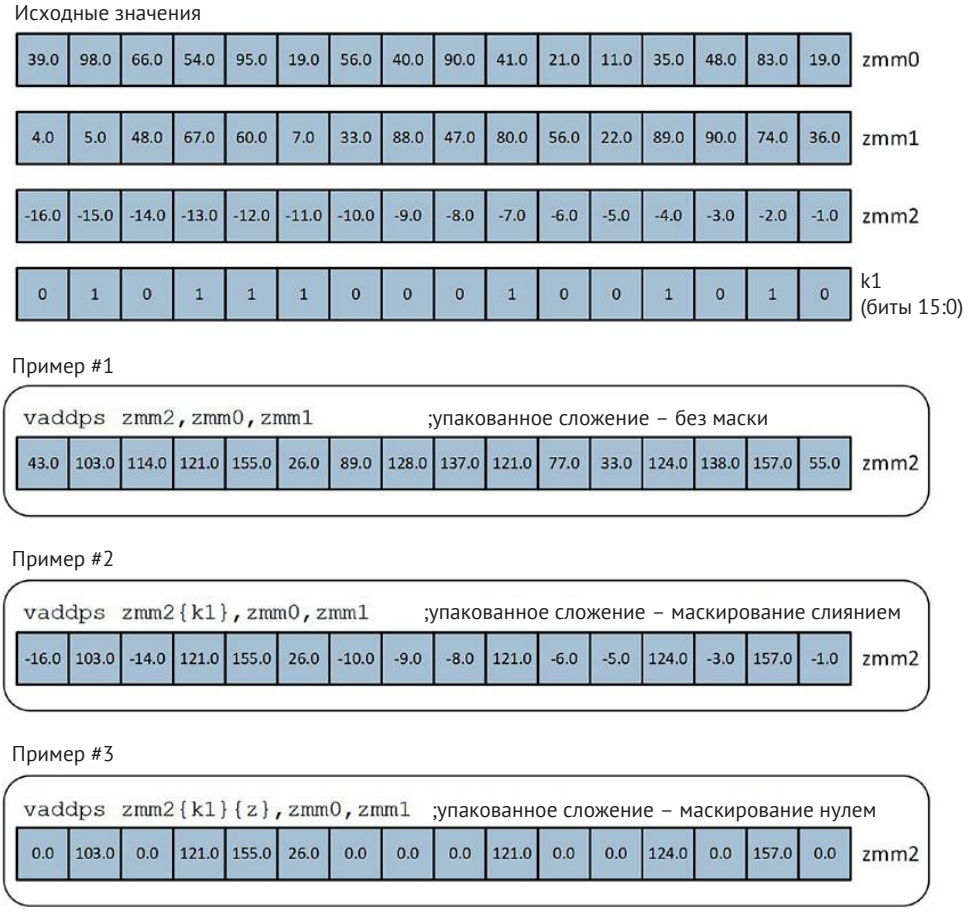


Рис. 12.2. Примеры выполнения команды vaddps без маскирования, с маскированием слияния и маскированием нулем

AVX-512 включает несколько новых команд, которые можно использовать для чтения значений и записи значений в регистр маски, а также для выполнения логических операций. Вы узнаете об этих командах позже в данной главе. Регистр маски также может использоваться в качестве операнда назначения с командами, которые генерируют результат векторной маски, например `vcmpp[d|s]` и `vpcmp[b|w|d|q]`. Примеры исходного кода в главах 13 и 14 показывают, как использовать эти команды с регистром маски. Команды AVX-512 могут использовать регистры маски K1–K7 в качестве маски предиката. Регистр маски K0 не может использоваться в качестве операнда маски предиката, но его можно использовать в любой команде, которая требует регистр маски в качестве операнда источника или приемника. Если команда AVX-512 пытается использовать K0 в качестве маски предиката, процессор использует неявный операнд из всех единиц, что запрещает все операции условного выполнения и маскирования.

Встроенное широковещание

Многие команды AVX-512 могут выполнять вычисления SIMD с использованием встроенного широковещательного операнда. *Встроенный широковещательный операнд* – это скалярное значение на основе памяти, которое *N* раз реплицируется во временное упакованное значение, где *N* представляет количество элементов вектора, на которые ссылается команда. Это временное упакованное значение затем используется в качестве операнда в вычислении SIMD.

Рисунок 12.3 содержит два примера последовательности команд, которые иллюстрируют операции широковещательной рассылки. В первом примере команда `vbroadcastss` применяется для загрузки константы с плавающей запятой одинарной точности 2.0 в каждую позицию элемента ZMM1. Следующая команда `vmulps zmm2,zmm0,zmm1` умножает каждое значение в ZMM0 на 2,0 и сохраняет результаты в ZMM2. Второй пример команды на рис. 12.3, `vmulps zmm2,zmm0,real4 bcst [rax]`, выполняет ту же операцию, используя встроенный операнд широковещательной рассылки. Запись `real4 bcst` – это директива MASM, которая указывает ассемблеру обрабатывать ячейку памяти, на которую указывает регистр RAX, как операнд встроенной широковещательной передачи.

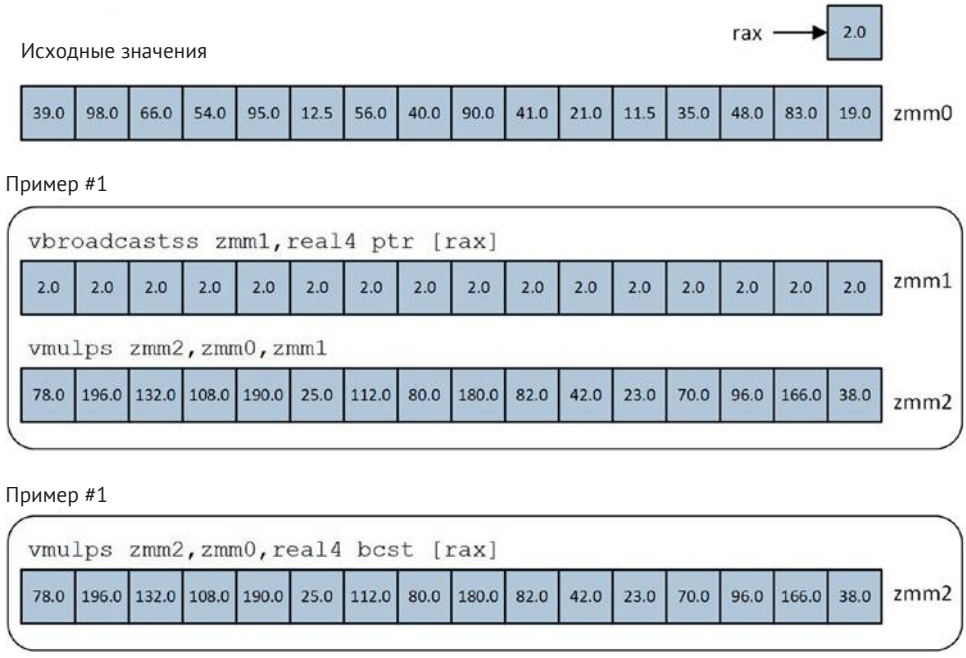


Рис. 12.3. Пакетное умножение с плавающей запятой одинарной точности с использованием команд `vbroadcastss` и `vmulps` по сравнению с командой `vmulps` со встроенным широковещательным операндом

AVX-512 поддерживает встроенные широковещательные операции с использованием элементов шириной 32 и 64 бита. Встроенное широковещание не работает с элементами шириной 8 и 16 бит.

Округление на уровне команды

Последнее усовершенствование синтаксиса команд AVX-512 включает управление округлением на уровне команд для операций с плавающей запятой. В главе 5 вы узнали, как использовать команды `vldmxcsr` и `vstmxcsr` для изменения глобального режима округления процессора в операциях с плавающей запятой (см. пример `Ch05_06`). AVX-512 позволяет некоторым командам указывать операнд режима округления с плавающей запятой, который отменяет текущий режим округления в MXCSR.RC. В табл. 12.3 показаны поддерживаемые операнды режима округления, которые также называются статическими режимами округления. Суффикс `-sae`, добавляемый к каждой строке операнда статического режима округления, является аббревиатурой для подавления всех исключений. Этот суффикс служит напоминанием о том, что исключения операций с плавающей запятой маскируются *всегда*, когда указывается операнд статического режима округления; обновления флага MXCSR при этом также отключены.

Таблица 12.3. Операнды режима статического округления на уровне команд AVX-512

Операнд режима округления	Описание
`{rn-sae}`	Округление до ближайшего
`{rd-sae}`	Округление вниз (в сторону $-\infty$)
`{ru-sae}`	Округление вверх (в сторону $+\infty$)
`{rz-sae}`	Округление к нулю (усечение)

Операнды режима статического округления могут использоваться со многими (но не всеми) командами AVX-512, которые выполняют операции с плавающей запятой, используя 512-битные упакованные операнды; 256-битные и 128-битные упакованные операнды не поддерживаются. Операнды режима статического округления также можно использовать с командами, которые выполняют скалярные операции с плавающей запятой. В обоих случаях использования все операнды команд должны быть регистрами. Например, команды `vmulps zmm2,zmm0,zmm1 {rz-sae}` и `vmulss xmm2,xmm0,xmm1 {rz-sae}` действительны, тогда как `vmulps zmm2,zmm0,zmmword ptr [rax] {rz-sae}` и `vmulss xmm2,xmm0,real4 ptr [rax] {rz-sae}` недопустимы. Некоторые команды AVX-512 с плавающей запятой не поддерживают спецификацию операнда режима статического округления, но эти команды по-прежнему могут использовать операнд `{sae}` для подавления всех исключений.

12.3. Обзор набора команд

В этом разделе представлен обзор следующих расширений набора команд AVX-512: AVX512F, AVX512CD, AVX512BW и AVX512DQ. Он также включает в себя сводку команд для работы с регистрами маски операции. Таблицы в этом разделе содержат только новые команды для AVX-512. Они не включают команды, которые представляют собой простое расширение существующей команды AVX или AVX2. Большинство команд в этих таблицах можно использовать с операндами шириной 512 бит; 256-битные и 128-битные операнды могут использоваться на процессорах, поддерживающих расширение AVX512VL.

12.3.1. AVX512F

В табл. 12.4 перечислены команды AVX512F. Как упоминалось в обзорном разделе этой главы, все процессоры, совместимые с AVX-512, должны как минимум поддерживать команды, включенные в эту таблицу.

Таблица 12.4. Обзор набора команд AVX512F

Мнемоника	Описание
`valign[d|q]`	Выровнять векторы двойного слова | квадрослова
`vblendmp[d|s]`	Смешать векторы с плавающей запятой в соответствии с маской операции
`vbroadcastf[32x4164x4]`	Передача кортежей с плавающей запятой
`vbroadcasti[32x4164x4]`	Передача целочисленных кортежей
`vcompressp[d|s]`	Сохранение разреженных упакованных значений с плавающей запятой
`vcvtp[d|s]2udq`	Преобразование упакованных чисел с плавающей запятой в упакованные целочисленные двойные слова без знака
`vcvts[d|s]2usi`	Преобразование скалярного операнда с плавающей запятой в беззнаковое целое двойное слово
`vcvttp[d|s]2udq`	Преобразование упакованных чисел с плавающей запятой в упакованные беззнаковые целочисленные двойные слова с усечением
`vcvtts[d|s]2usi`	Преобразование скалярного операнда с плавающей запятой в беззнаковое целочисленное двойное слово с усечением
`vcvtudq2p[d|s]`	Преобразование упакованных беззнаковых целочисленных двойных слов в упакованные числа с плавающей запятой
`vcvtusi2s[d|s]`	Преобразование целочисленного беззнакового двойного слова в число с плавающей запятой
`vexpandp[d|s]`	Загрузка разреженных упакованных значений с плавающей запятой
`vextractf[32x4|64x4]`	Извлечение упакованных значений с плавающей запятой
`vextracti[32x4|64x4]`	Извлечение упакованных целочисленных значений
`vfixupimmp[d|s]`	Исправить специальные упакованные значения с плавающей запятой
`vfixupimms[d|s]`	Исправить специальные скалярные значения с плавающей запятой
`vgetexpp[d|s]`	Преобразование экспонент упакованных значений с плавающей запятой
`vgetexps[d|s]`	Преобразование экспонент скалярных значений с плавающей запятой
`vgetmantpfd|s]`	Получение нормализованных мантисс из упакованных значений с плавающей запятой
`vgetmants[d|s]`	Получение нормализованных мантисс из скалярных значений с плавающей запятой
`vinsertf[32x4|64x4]`	Вставить упакованные значения с плавающей запятой
`vinserti[32x4|64x4]`	Вставить упакованные целочисленные значения

Продолжение табл. 12.4

Мнемоника	Описание
`vmovdqa[32|64]`	Присвоить (переместить) выровненные упакованные целые числа
`vmovdqu[32|64]`	Присвоить (переместить) невыровненные упакованные целые числа
`vpblendm[d|q]`	Смешивание упакованных целых чисел с учетом маски операции
`vpbroadcast[d|q]`	Трансляция целого числа из регистра общего назначения
`vpcmp[d|q]`	Сравнение упакованных целых чисел со знаком
`vpcmpu[d|q]`	Сравнение упакованных целых чисел без знака
`vpcompress[d|q]`	Хранение редко упакованных целых чисел
`vpermi2[d|q|ps|pd]`	Перестановка из двух таблиц, перезаписывающая индекс
`vpermt2[d|q|ps|pd]`	Перестановка из двух таблиц с перезаписью одной таблицы
`vpmov[db|sdb|usdb]`	Конвертация упакованных двойных слов в упакованные байты
`vpexpand[d|q]`	Загрузка разреженных упакованных целых чисел
`vpmax[s|u]q`	Вычисление максимума упакованных четверных слов
`vpmin[s|u]q`	Вычисление минимума упакованных четверных слов
`vpmov[db|sdb|usdb]`	Преобразование упакованных двойных слов в упакованные байты
`vpmov[dw|sdw|usdw]`	Преобразование упакованных двойных слов в упакованные слова
`vpmov[qb|sqb|usqb]`	Понижающее преобразование упакованных четверных слов в упакованные байты
`vpmov[qd|sqd|usqd]`	Понижающее преобразование упакованных четверных слов в упакованные двойные слова
`vpmov[qw|sqw|usqw]`	Понижающее преобразование упакованных четверных слов в упакованные слова
`vprol[d|q]`	Повернуть влево упакованные целые числа с постоянным отсчетом
`vprolv[d|q]`	Повернуть влево упакованные целые числа с переменным отсчетом
`vpror[d|q]`	Повернуть вправо упакованные целые числа с постоянным отсчетом
`vprorv[d|q]`	Повернуть вправо упакованные целые числа с переменным отсчетом
`vpscatterd[d|q]`	Разделение упакованных целых чисел с использованием индексов двойных слов
`vpscatterq[d|q]`	Разброс упакованных целых чисел с использованием индексов четверного слова
`vpsraq`	Арифметический сдвиг вправо упакованных целочисленных четверных слов с постоянным отсчетом
`vpsravq`	Арифметический сдвиг вправо упакованных целочисленных четверных слов с переменным отсчетом
`vpternlog[d|q]`	Побитовая троичная логика

Окончание табл. 12.4

Мнемоника	Описание
`vptestm[d|q]`	Упакованное целочисленное побитовое И и установка маски
`vptestnm[d|q]`	Упакованное целочисленное побитовое И-НЕ и установка маски
`vrcpl4p[d|s]`	Вычисление приблизительных обратных величин упакованных значений с плавающей запятой
`vrcpl4s[d|s]`	Вычисление приблизительных обратных величин скалярных значений с плавающей запятой
`vreducep[d|s]`	Выполнение преобразования сокращения для упакованных значений с плавающей запятой
`vreduces[d|s]`	Выполнение преобразования сокращения для скалярных значений с плавающей запятой
`vrndscalep[d|s]`	Округление упакованных значений с плавающей запятой до числа дробных битов
`vrndscales[d|s]`	Округление значений с плавающей запятой до числа дробных битов
`vrsqrtl4p[d|s]`	Вычисление приблизительных обратных величин квадратных корней упакованных операндов с плавающей запятой
`vrsqrtl4s[d|s]`	Вычисление приблизительных обратных величин квадратных корней скалярных операндов с плавающей запятой
`vscalefp[d|s]`	Масштабирование упакованных значений с плавающей запятой
`vscalefs[d|s]`	Масштабирование скалярных значений с плавающей запятой
`vscatterdp[d|s]`	Разделение упакованных значений с плавающей запятой с использованием индексов двойного слова
`vscatterqp[d|s]`	Разделение упакованных значений с плавающей запятой с использованием индексов четверного слова
`vshufff32x4[64x2]`	Перемешивание упакованных значений с плавающей запятой
`vshufif32x4[64x2]`	Перемешивание упакованных целых значений

12.3.2. AVX512CD

В табл. 12.5 перечислены команды AVX512CD. Эти команды часто применяются для обнаружения и уменьшения зависимостей данных, которые могут возникать при выполнении вычислений разреженных массивов или операций разделения. Их также можно использовать с другими командами AVX-512 для выполнения обычных вычислений.

Таблица 12.5. Обзор набора команд AVX512CD

Мнемоника	Описание
`vpbroadcastm[b2q|w2d]`	Рассылка маски в векторный регистр
`vpconflict[d|q]`	Обнаружение конфликтов в упакованных целых числах
`vplzcnt[d|q]`	Подсчет количества ведущих нулей в упакованных целых числах

12.3.3. AVX512BW

В табл. 12.6 перечислены команды AVX512BW. Эти команды выполняют свои операции, используя упакованные операнды размера байтов и слов.

Таблица 12.6. Обзор набора команд AVX512BW

Мнемоника	Описание
`vdbpsadbw`	Двойная блочная упакованная сумма абсолютных разностей с использованием беззнаковых байтов
`vmovdq[u8|ul6]`	Присвоение (перемещение) невыровненных упакованных целых чисел
`vpblendm[b|w]`	Смешивание упакованных целых чисел на основании маски
`vpbroadcast[b|w]`	Трансляция целого числа из регистра общего назначения
`vpcmp[b|w]`	Сравнение упакованных целых чисел со знаком
`vpcmpu[b|w]`	Сравнение упакованных целых чисел без знака
`vpermw`	Перестановка упакованных слов
`vpermi2w`	Перестановка целых чисел из двух таблиц с перезаписью индекса
`vpermt2w`	Перестановка целых слов из двух таблиц с перезаписью одной таблицы
`vpmov[b|w]2m`	Преобразование векторного регистра в регистр маски
`vpmovm2[b|w]`	Преобразование регистра маски в векторный регистр
`vpmovw[b|sb|usb]`	Преобразование упакованных слов в упакованные байты
`vpsllvw`	Логический сдвиг упакованного слова влево с использованием переменного количества битов
`vpsravw`	Арифметический сдвиг упакованного слова вправо с использованием переменного количества битов
`vpsrlvw`	Логический сдвиг упакованного слова вправо с использованием переменного количества битов
`vptestm[b|w]`	Побитовое И упакованных целых чисел и установка маски
`vptestnm[b|w]`	Побитовое И-НЕ упакованных целых чисел и установка маски

12.3.4. AVX512DQ

В табл. 12.7 перечислены команды AVX512DQ. Эти команды выполняют свои операции с использованием операндов в формате упакованных двойных и четверных слов. AVX512DQ также включает в себя команды, которые выполняют преобразование между упакованными четверными словами с плавающей запятой и целыми числами.

Таблица 12.7. Обзор набора команд AVX512DQ

Мнемоника	Описание
`vcvtp[d|s]2qq`	Преобразование упакованных чисел с плавающей запятой в целые числа со знаком в формате четверного слова
`vcvtp[d|s]2uqq`	Преобразование упакованных чисел с плавающей запятой в целые числа без знака
`vcvttp[d|s]2qq`	Преобразование упакованных чисел с плавающей запятой в целые числа со знаком в формате четверного слова с усечением
`vcvttp[d|s]2uqq`	Преобразование упакованных чисел с плавающей запятой в целые числа без знака с усечением
`vcvtuqq2p[d|s]`	Преобразование упакованных беззнаковых целых четверных слов в числа с плавающей запятой
`vextractf64x2`	Извлечение упакованных значений с плавающей запятой двойной точности
`vextracti64x2`	Извлечь упакованные значения в формате четверного слова
`vfpclass[pd|ps]`	Тестирование упакованного класса с плавающей запятой
`vfpclass[sd|ss]`	Тестирование скалярного класса с плавающей запятой
`v inserts 4x2`	Вставка упакованных значений с плавающей запятой двойной точности
`vinserti64x2`	Вставка упакованных значений в формате четверного слова
`vpmov[d|q]2m`	Преобразование векторного регистра в регистр маски
`vpmovm2[d|q]`	Преобразование регистра маски в векторный регистр
`vpmullq`	Умножение упакованных целых чисел в формате четверного слова и сохранение младшего результата
`vrangep[d|s]`	Расчет ограничения диапазона для упакованных чисел с плавающей запятой
`vranges[d|s]`	Расчет ограничения диапазона для скалярных чисел с плавающей запятой
`vreducep[d|s]`	Выполнение сокращения упакованных значений с плавающей запятой
`vreduces[d|s]`	Выполнение сокращения скалярных значений с плавающей запятой

12.3.5. Регистры маски операции

В табл. 12.8 перечислены команды для работы с регистрами маски операции. Версии этих команд для работы со словами требуют AVX512F, за исключением kaddw и ktestw, которым нужно AVX512DQ. Версии двойного слова и четверного слова команд регистра opmask требуют AVX512BW; для байтовых версий необходимо AVX512DQ.

Таблица 12.8. Обзор набора команд для работы с регистром маски

Mnemonic	Description
kadd[b|w|d|q]	Добавление значений маски
kand[b|w|d|q]	Побитовое И
kandn[b|w|d|q]	Побитовое И-НЕ
kmov[b|w|d|q]	Перенос значения в регистр маски / из регистра
knot[b|w|d|q]	Побитовое НЕ
kor[b|w|d|q]	Побитовое включающее ИЛИ
kortest[b|w|d|q]	Побитовое включающее ИЛИ; обновление RFLAGS.ZF и RFLAGS.CF
kshiftl[b|w|d|q]	Сдвиг влево
kshiftr[b|w|d|q]	Сдвиг вправо
ktest[b|w|d|q]	Побитовое И и И-НЕ; обновление RFLAGS.ZF и RFLAGS.CF
kunpck[bw|wd|dq]	Распаковка
kxnor[b|w|d|q]	Побитовое исключающее ИЛИ-НЕ
kxor[b|w|d|q]	Побитовое исключающее ИЛИ

12.4. Заключение

В главе 12 рассмотрены следующие ключевые моменты:

- все процессоры, совместимые с AVX-512, поддерживают расширение набора команд AVX512F. Наличие дополнительных расширений набора команд AVX-512 зависит от целевого рынка процессора;
- набор регистров AVX-512 включает 32 регистра шириной 512 бит с именами ZMM0-ZMM31. Младшие 256 и 128 бит связаны с регистрами YMM0–YMM31 и XMM0–XMM31 соответственно;
- набор регистров AVX-512 также включает 8 регистров маски операции с именами K0–K7. Регистры маски K1–K7 могут использоваться для выполнения условного выполнения на уровне команд с маскированием слияния или нулевым маскированием;
- многие команды AVX-512, требующие упакованного операнда с постоянными значениями, могут использовать встроенный широковещательный операнд вместо отдельной широковещательной команды;
- со многими командами AVX-512, которые выполняют операции с плавающей запятой, используя 512-битные упакованные или скалярные операнды регистров с плавающей запятой, может быть указан операнд режима статического округления.

Глава 13

Программирование AVX-512 – числа с плавающей запятой

В предыдущих главах вы узнали, как выполнять операции над упакованными и скалярными значениями с плавающей запятой, используя наборы команд AVX и AVX2. В этой главе вы узнаете, как выполнять подобные операции, используя набор команд AVX-512. Первая часть данной главы содержит примеры исходного кода, иллюстрирующие основные концепции программирования AVX-512 с использованием скалярных операндов с плавающей запятой. Сюда входят примеры, иллюстрирующие условное выполнение, слияние и маскирование нуля, а также округление на уровне команд. Вторая часть этой главы демонстрирует использование набора команд AVX-512 для выполнения упакованных вычислений с плавающей запятой с использованием операндов шириной 512 бит и набора регистров ZMM.

Примеры исходного кода в этой главе требуют наличия процессора и операционной системы, которые поддерживают AVX-512 и следующие расширения набора команд: AVX512F, AVX512CD, AVX512BW, AVX512DQ и AVX512VL. Как обсуждалось в главе 12, эти расширения поддерживаются процессорами, основанными на микроархитектуре Intel Skylake Server. Будущие процессоры от AMD и Intel, вероятно, также будут включать ранее упомянутые расширения набора команд. Вы можете использовать одну из свободно доступных утилит, перечисленных в приложении, чтобы определить, какие наборы команд AVX-512 поддерживает ваша система. В главе 16 вы узнаете, как использовать команду `cpuid` для обнаружения определенных расширений набора команд AVX-512 во время выполнения.

13.1. Скалярные операнды с плавающей точкой

Набор команд AVX-512 предоставляет дополнительные возможности по работе со скалярными операндами с плавающей запятой, включая маскирование слияния, маскирование нуля и управление округлением на уровне команд. Примеры исходного кода в этом разделе объясняют, как использовать эти возможности. Они также иллюстрируют некоторые незначительные отличия, о которых необходимо помнить при написании кода для работы со скалярными значениями с плавающей запятой с использованием команд AVX-512.

13.1.1. Маскирование слиянием

В листинге 13.1 показан исходный код примера Ch13_01. В этом примере описывается, как выполнить маскирование слиянием с помощью команд AVX-512 для скалярных операндов с плавающей запятой. Он также иллюстрирует использование нескольких команд для работы с регистрами маски операций.

Листинг 13.1. Пример Ch13_01

```
//-----------------------------------------------
//              Ch13_01.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <string>
#include <iostream>
#include <iomanip>
#include <limits>
#define _USE_MATH_DEFINES
#include <math.h>

using namespace std;

extern "C" double g_PI = M_PI;
extern "C" bool Avx512CalcSphereAreaVol_(double* sa, double* vol, double radius, double
error_val);

bool Avx512CalcSphereAreaVolCpp(double* sa, double* vol, double radius, double error_val)
{
    bool rc;

    if (radius < 0.0)
    {
        *sa = error_val;
        *vol = error_val;
        rc = false;
    }
    else
    {
        *sa = 4.0 * g_PI * radius * radius;
        *vol = *sa * radius / 3.0;
        rc = true;
    }

    return rc;
}

int main()
{
    const double error_val = numeric_limits<double>::quiet_NaN();
    const double radii[] = {-1.0, 0.0, 1.0, 2.0, 3.0, 4.0, -7.0, 10.0, -18.0, 20.0};
    int num_r = sizeof(radii) / sizeof(double);

    string sp {"  "};
    string sep(75, '-');
```

```
    cout << setw(10) << "radius" << sp;
    cout << setw(6) << "rc1" << sp;
    cout << setw(6) << "rc2" << sp;
    cout << setw(10) << "sa1" << sp;
    cout << setw(10) << "sa2" << sp;
    cout << setw(10) << "vol1" << sp;
    cout << setw(10) << "vol2" << '\n';
    cout << sep << '\n';

    cout << fixed << setprecision(4);

    for (int i = 0; i < num_r; i++)
    {
        double sa1, sa2;
        double vol1, vol2;
        double r = radii[i];

        bool rc1 = Avx512CalcSphereAreaVolCpp(&sa1, &vol1, r, error_val);
        bool rc2 = Avx512CalcSphereAreaVol_(&sa2, &vol2, r, error_val);

        cout << setw(10) << r << sp;
        cout << setw(6) << boolalpha << rc1 << sp;
        cout << setw(6) << boolalpha << rc2 << sp;
        cout << setw(10) << sa1 << sp;
        cout << setw(10) << sa2 << sp;
        cout << setw(10) << vol1 << sp;
        cout << setw(10) << vol2 << '\n';
    }

    return 0;
}

;-----------------------------------------------
;               Ch13_01.asm
;-----------------------------------------------

            include <cmpequ.asmh>
            .const
r8_three    real8 3.0
r8_four     real8 4.0

            extern g_PI:real8

; extern "C" bool Avx512CalcSphereAreaVol_(double* sa, double* v, double r, double error_
val);
;
; Возвращает:  false = некорректный радиус, true = корректный радиус

        .code
Avx512CalcSphereAreaVol_ proc

; Проверка, что радиус >= 0.0
        vmovsd xmm0,xmm0,xmm2               ;xmm0 = радиус
        vxorpd xmm5,xmm5,xmm5               ;xmm5 = 0.0
        vmovsd xmm16,xmm16,xmm3             ;xmm16 = error_val
        vcmpsd k1,xmm0,xmm5,CMP_GE          ;k1[0] = 1, если радиус >= 0.0
```

```
; Вычисление площади поверхности и объема с использованием маски
        vmulsd xmm1{k1},xmm0,xmm0           ;xmm1 = r * r
        vmulsd xmm2{k1},xmm1,[r8_four]      ;xmm2 = 4 * r * r
        vmulsd xmm3{k1},xmm2,[g_PI]         ;xmm3 = 4 * PI * r * r (sa)
        vmulsd xmm4{k1},xmm3,xmm0           ;xmm4 = 4 * PI * r * r * r
        vdivsd xmm5{k1},xmm4,[r8_three]     ;xmm5 = 4 * PI * r * r * r / 3 (vol)

; Присвоение переменным площади и объема error_val, если radius < 0.0 истинно
        knotw k2,k1                         ;k2[0] = 1 if radius < 0.0
        vmovsd xmm3{k2},xmm3,xmm16          ;xmm3 = error_val если radius < 0.0
        vmovsd xmm5{k2},xmm5,xmm16          ;xmm5 = error_val если radius < 0.0

; Сохранение результатов
        vmovsd real8 ptr [rcx],xmm3         ;сохранение площади поверхности
        vmovsd real8 ptr [rdx],xmm5         ;сохранение объема

        kmovw eax,k1                        ;eax = возвращаемый код результата
        ret
Avx512CalcSphereAreaVol_ endp
        end
```

Код C++ в листинге 13.1 начинается с функции `Avx512CalcSphereAreaVolCpp`. Эта функция вычисляет площадь поверхности и объем любой сферы, радиус которой больше или равен нулю. Если радиус сферы меньше нуля, `Avx512CalcSphereAreaVolCpp` устанавливает для площади поверхности и объема значение `error_val`. Оставшийся код C++ в листинге 13.1 выполняет инициализацию тестового примера, вызывает функции `Avx512CalcSphereAreaVolumeCpp` и `Avx512CalcSphereAreaVolume_` и передает результаты в `cout`.

Функция на языке ассемблера `Avx512CalcSphereAreaVol_` реализует тот же алгоритм, что и ее аналог C++. Эта функция начинается с команды `vmovsd xmm0,xmm0,xmm2`, которая копирует значение аргумента `r` в регистр XMM0. Затем она загружает в регистр XMM5 нулевое значение. Команда `vmovsd xmm16,xmm16,xmm3` копирует `error_val` в регистр XMM16. Согласно соглашению о вызовах Visual C++, новые регистры AVX-512 ZMM16–ZMM31 вместе с младшими аналогами YMM и XMM изменяемы вне границ функций. Это означает, что данные регистры могут использоваться любой функцией на языке ассемблера без сохранения их значений. Следующая команда, `vcmpsd k1,xmm0,xmm5,CMP_GE`, устанавливает бит регистра маски `K1[0]` в единицу, если радиус `r` больше или равен нулю; в противном случае этот бит устанавливается в ноль.

Первая команда блока кода вычисления площади поверхности и объема, `vmulsd xmm1 {k1},xmm0,xmm0`, вычисляет `r*r`, если бит `K1[0]` установлен в единицу (условие `r >= 0.0` истинно); затем она сохраняет вычисленное произведение в XMM1[63:0]. Если бит K1[0] установлен в ноль (условие `r < 0.0` истинно), процессор пропускает вычисление умножения с плавающей запятой двойной точности и оставляет регистр XMM1 без изменений. Следующая команда, `vmulsd xmm2 {k1},xmm1,[r8_four]`, вычисляет `4.0*r*r`, используя ту же операцию маскирования слиянием, что и предыдущая команда. Далее команды `vmulsd` и `vdivsd` завершают расчет искомой площади поверхности (XMM3) и объема (XMM5). Операции маскирования слиянием в этом кодовом блоке иллюстрируют одну из ключевых вычислительных возможностей AVX-512: процессор выполняет

арифметические вычисления с плавающей запятой двойной точности, только если бит K1[0] установлен в единицу; в противном случае вычисления не выполняются, и соответствующие регистры операндов назначения остаются неизменными.

После вычислений площади поверхности и объема `knotw k2,k1` инвертирует младшие 16 бит K1 и сохраняет этот результат в K2[15:0]. Эта команда также обнуляет биты K2[63:16]. Бит K2[0] теперь устанавливается в единицу, если `r < 0,0` истинно. Здесь используется команда `knotw`, поскольку она является частью расширения набора команд AVX512F; `knot[b|d|q]` также будет здесь работать. Следующая команда, `vmovsd xmm3 {k2},xmm3,xmm16`, устанавливает для площади поверхности значение `error_val`, если условие `r < 0,0` истинно. Затем команда `vmovsd xmm5 {k2},xmm5,xmm16` выполняет ту же операцию для нахождения объема. Последняя команда `kmovw eax,k1` загружает в EAX возвращаемый условный код успешного выполнения функции. Вот результаты выполнения примера исходного кода `Ch13_01`:

```
   radius        rc1        rc2       sa1        sa2       vol1       vol2
-------------------------------------------------------------------------
  -1.0000      false      false       nan        nan        nan        nan
   0.0000       true       true    0.0000     0.0000     0.0000     0.0000
   1.0000       true       true   12.5664    12.5664     4.1888     4.1888
   2.0000       true       true   50.2655    50.2655    33.5103    33.5103
   3.0000       true       true  113.0973   113.0973   113.0973   113.0973
   4.0000       true       true  201.0619   201.0619   268.0826   268.0826
  -7.0000      false      false       nan        nan        nan        nan
  10.0000       true       true 1256.6371  1256.6371  4188.7902  4188.7902
 -18.0000      false      false       nan        nan        nan        nan
  20.0000       true       true 5026.5482  5026.5482 33510.3216 33510.3216
```

13.1.2. Маскирование нулем

Следующий пример исходного кода называется `Ch13_02`. В этом примере показано, как использовать маскирование нулем для исключения зависимых от данных условных переходов из вычислений. В листинге 13.2 приведен исходный код данного примера.

Листинг 13.2. Пример Ch13_02

```
//-----------------------------------------------
//               Ch13_02.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <array>
#include <random>

using namespace std;

extern "C" bool Avx512CalcValues_(double* c, const double* a, const double* b, size_t n);

template<typename T> void Init(T* x, size_t n, unsigned int seed)
```

```
{
    uniform_int_distribution<> ui_dist {1, 200};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (T)(ui_dist(rng) - 25);
}

bool Avx512CalcValuesCpp(double* c, const double* a, const double* b, size_t n)
{
    if (n == 0)
        return false;

    for (size_t i = 0; i < n; i++)
    {
        double val = a[i] * b[i];
        c[i] = (val >= 0.0) ? sqrt(val) : val * val;
    }

    return true;
}

int main()
{
    const size_t n = 20;
    array<double, n> a;
    array<double, n> b;
    array<double, n> c1;
    array<double, n> c2;

    Init<double>(a.data(), n, 13);
    Init<double>(b.data(), n, 23);

    bool rc1 = Avx512CalcValuesCpp(c1.data(), a.data(), b.data(), n);
    bool rc2 = Avx512CalcValues_(c2.data(), a.data(), b.data(), n);

    if (!rc1 || !rc2)
    {
        cout << "Возвращен ошибочный код - ";
        cout << "rc1 = " << boolalpha << rc1 << "  ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
    }
    else
    {
        cout << fixed << setprecision(4);

        for (size_t i = 0; i < n; i++)
        {
            cout << "i:  " << setw(2) << i << "  ";
            cout << "a:  " << setw(9) << a[i] << "  ";
            cout << "b:  " << setw(9) << b[i] << "  ";
            cout << "c1: " << setw(13) << c1[i] << "  ";
            cout << "c2: " << setw(13) << c2[i] << "\n";
        }
    }
}
```

```
;-----------------------------------------------
;               Ch13_02.asm
;-----------------------------------------------

        include <cmpequ.asmh>

; extern "C" bool Avx512CalcValues_(double* c, const double* a, const double* b, size_t n);

        .code
Avx512CalcValues_ proc

; Проверка n и инициализация индекса массива i
        xor eax,eax                         ;возвращаемый код ошибки (также i = 0)
        test r9,r9                          ;действительно ли n == 0?
        jz Done                             ;переход, если n равен нулю

        vxorpd xmm5,xmm5,xmm5               ;xmm5 = 0.0

; Чтение следующих a[i] и b[i], вычисление val
@@:     vmovsd xmm0,real8 ptr [rdx+rax*8]   ;xmm0 = a[i];
        vmovsd xmm1,real8 ptr [r8+rax*8]    ;xmm1 = b[i];
        vmulsd xmm2,xmm0,xmm1               ;val = a[i] * b[i]

; Вычисление c[i] = (val >= 0.0) ? sqrt(val) : val * val
        vcmpsd k1,xmm2,xmm5,CMP_GE          ;k1[0] = 1 if val >= 0.0
        vsqrtsd xmm3{k1}{z},xmm3,xmm2       ;xmm3 = (val > 0.0) ? sqrt(val) : 0.0
        knotw k2,k1                         ;k2[0] = 1 if val < 0.0
        vmulsd xmm4{k2}{z},xmm2,xmm2        ;xmm4 = (val < 0.0) ? val * val : 0.0
        vorpd xmm0,xmm4,xmm3                ;xmm0 = (val >= 0.0) ? sqrt(val) : val * val
        vmovsd real8 ptr [rcx+rax*8],xmm0   ;сохранение результата в c[i]

; Обновление индекса i и повтор до завершения
        inc rax                             ;i += 1
        cmp rax,r9
        jl @B
        mov eax,1                          ;возвращаемый код успешного выполнения

Done:   ret
Avx512CalcValues_ endp
        end
```

В коде C++ функция `Avx512CalcValuesCpp` выполняет простые арифметические вычисления с использованием массивов с плавающей запятой двойной точности. Каждая итерация цикла начинается с вычисления промежуточного значения `val = a[i]*b[i]`. Следующий оператор `c[i] = (val >= 0.0) ? sqrt(val) : val * val` загружает в `c[i]` количество, которое изменяется в зависимости от значения `val`. Функция на языке ассемблера `Avx512CalcValues_` выполняет те же вычисления. Функция `main` в C++ содержит код, который инициализирует тестовые массивы, вызывает функции `Avx512CalcValuesCpp` и `Avx512CalcValues_` и отображает результаты.

Цикл обработки `Avx512CalcValues_` начинается с двух команд `vmovsd`, которые загружают `a[i]` и `b[i]` в регистры XMM0 и XMM1 соответственно. Следующая команда `vmulsd xmm2,xmm0,xmm1` вычисляет промежуточный продукт `val =`

a[i]*b[i]. После вычисления val команда vcmpsd k1,xmm2,xmm5,CMP_GE сравнивает val с 0,0 и устанавливает бит K1[0] в единицу, если val больше или равно нулю; в противном случае бит K1[0] устанавливается в ноль. Следующая команда vsqrtsd xmm3 {k1} {z},xmm3,xmm2 вычисляет квадратный корень из val, если K1[0] установлен в единицу, и сохраняет результат в XMM3. Если K1[0] равен нулю, процессор пропускает вычисление квадратного корня и устанавливает регистр XMM3 на 0,0.

Команда knotw k2,k1 устанавливает K2[0] в единицу, если val меньше 0,0. Следующая команда vmulsd xmm4 {k2} {z},xmm2,xmm2 вычисляет и сохраняет значение произведения val*val в XMM4, если бит K2[0] установлен в единицу; в противном случае XMM4 устанавливается равным 0,0. После выполнения команды vmulsd регистр XMM3 содержит sqrt(val), а XMM4 содержит 0,0, или XMM3 содержит 0,0, а XMM4 содержит значение val*val. Эти значения регистров упрощают использование команды vorpd xmm0,xmm4,xmm3 для загрузки в XMM0 окончательного значения c[i].

Как и в предыдущем примере исходного кода, функция Avx512CalcValues_ демонстрирует важные возможности AVX-512. Использование маскирования нулем и некоторой простой двоичной логики позволило функции Avx512CalcValues_ принимать логические решения без каких-либо команд условного перехода. Это примечательно, поскольку команды условного перехода, зависящие от данных, часто работают медленнее, чем прямой код. Вот результат выполнения примера исходного кода Ch13_02:

```
i:  0  a:    -6.0000  b:   67.0000  c1:   161604.0000  c2:  161604.0000
i:  1  a:   128.0000  b:   22.0000  c1:       53.0660  c2:      53.0660
i:  2  a:   130.0000  b:   -8.0000  c1:  1081600.0000  c2: 1081600.0000
i:  3  a:   152.0000  b:   73.0000  c1:      105.3376  c2:     105.3376
i:  4  a:    94.0000  b:    6.0000  c1:       23.7487  c2:      23.7487
i:  5  a:     2.0000  b:   88.0000  c1:       13.2665  c2:      13.2665
i:  6  a:    12.0000  b:  103.0000  c1:       35.1568  c2:      35.1568
i:  7  a:   105.0000  b:  117.0000  c1:      110.8377  c2:     110.8377
i:  8  a:   140.0000  b:  -20.0000  c1:  7840000.0000  c2: 7840000.0000
i:  9  a:    74.0000  b:    3.0000  c1:       14.8997  c2:      14.8997
i: 10  a:    43.0000  b:   -9.0000  c1:   149769.0000  c2:  149769.0000
i: 11  a:     2.0000  b:  122.0000  c1:       15.6205  c2:      15.6205
i: 12  a:    36.0000  b:    9.0000  c1:       18.0000  c2:      18.0000
i: 13  a:   -18.0000  b:  123.0000  c1:  4901796.0000  c2: 4901796.0000
i: 14  a:   170.0000  b:  134.0000  c1:      150.9304  c2:     150.9304
i: 15  a:   102.0000  b:    3.0000  c1:       17.4929  c2:      17.4929
i: 16  a:   118.0000  b:  -19.0000  c1:  5026564.0000  c2: 5026564.0000
i: 17  a:    85.0000  b:  148.0000  c1:      112.1606  c2:     112.1606
i: 18  a:    61.0000  b:   65.0000  c1:       62.9682  c2:      62.9682
i: 19  a:    18.0000  b:   74.0000  c1:       36.4966  c2:      36.4966
```

13.1.3. Округление на уровне команды

Последний пример исходного кода этого раздела, Ch13_03, объясняет, как использовать операнды округления на уровне команд. Он также иллюстрирует использование команд AVX-512, которые выполняют преобразование между целочисленными значениями с плавающей запятой и беззнаковыми целыми числами. В листинге 13.3 показан исходный код примера Ch13_03.

Листинг 13.3. Пример Ch13_03

```
//-----------------------------------------------
//               Ch13_03.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#define _USE_MATH_DEFINES
#include <math.h>

using namespace std;

extern "C" void Avx512CvtF32ToU32_(uint32_t val_cvt[4], float val);
extern "C" void Avx512CvtF64ToU64_(uint64_t val_cvt[4], double val);
extern "C" void Avx512CvtF64ToF32_(float val_cvt[4], double val);

void ConvertF32ToU32(void)
{
    uint32_t val_cvt[4];
    const float val[] {(float)M_PI, (float)M_SQRT2};
    const int num_vals = sizeof(val) / sizeof(float);

    cout << "\nConvertF32ToU32\n";

    for (int i = 0; i < num_vals; i++)
    {
        Avx512CvtF32ToU32_(val_cvt, val[i]);

        cout << "  Пример #" << i << " val = " << val[i] << '\n';
        cout << "    val_cvt[0] {rn-sae} = " << val_cvt[0] << '\n';
        cout << "    val_cvt[1] {rd-sae} = " << val_cvt[1] << '\n';
        cout << "    val_cvt[2] {ru-sae} = " << val_cvt[2] << '\n';
        cout << "    val_cvt[3] {rz-sae} = " << val_cvt[3] << '\n';
    }
}

void ConvertF64ToU64(void)
{
    uint64_t val_cvt[4];
    const double val[] {(float)M_PI, (float)M_SQRT2};
    const int num_vals = sizeof(val) / sizeof(double);

    cout << "\nConvertF64ToU64\n";

    for (int i = 0; i < num_vals; i++)
    {
        Avx512CvtF64ToU64_(val_cvt, val[i]);

        cout << "  Пример #" << i << " val = " << val[i] << '\n';
        cout << "    val_cvt[0] {rn-sae} = " << val_cvt[0] << '\n';
        cout << "    val_cvt[1] {rd-sae} = " << val_cvt[1] << '\n';
        cout << "    val_cvt[2] {ru-sae} = " << val_cvt[2] << '\n';
        cout << "    val_cvt[3] {rz-sae} = " << val_cvt[3] << '\n';
    }
}
```

```
void ConvertF64ToF32(void)
{
    float val_cvt[4];
    const double val[] {M_PI, -M_SQRT2};
    const int num_vals = sizeof(val) / sizeof(double);

    cout << "\nConvertF64ToF32\n";

    for (int i = 0; i < num_vals; i++)
    {
        Avx512CvtF64ToF32_(val_cvt, val[i]);

        cout << fixed << setprecision(7);

        cout << "  Пример #" << i << " val = " << val[i] << '\n';
        cout << "    val_cvt[0] {rn-sae} = " << val_cvt[0] << '\n';
        cout << "    val_cvt[1] {rd-sae} = " << val_cvt[1] << '\n';
        cout << "    val_cvt[2] {ru-sae} = " << val_cvt[2] << '\n';
        cout << "    val_cvt[3] {rz-sae} = " << val_cvt[3] << '\n';
    }
}

int main()
{
    ConvertF32ToU32();
    ConvertF64ToU64();
    ConvertF64ToF32();
    return 0;
}

;-----------------------------------------------
;               Ch13_03.asm
;-----------------------------------------------

; extern "C" void Avx512CvtF32ToU32_(uint32_t val_cvt[4], float val);

        .code
Avx512CvtF32ToU32_ proc
        vcvtss2usi eax,xmm1{rn-sae}      ;Преобразование с округлением до ближайшего
        mov dword ptr [rcx],eax

        vcvtss2usi eax,xmm1{rd-sae}      ;Преобразование с округлением вниз
        mov dword ptr [rcx+4],eax

        vcvtss2usi eax,xmm1{ru-sae}      ;Преобразование с округлением вверх
        mov dword ptr [rcx+8],eax

        vcvtss2usi eax,xmm1{rz-sae}      ;Преобразование с округлением до нуля (усечение)
        mov dword ptr [rcx+12],eax
        ret
Avx512CvtF32ToU32_ endp

; extern "C" void Avx512CvtF64ToU64_(uint64_t val_cvt[4], double val);

Avx512CvtF64ToU64_ proc
        vcvtsd2usi rax,xmm1{rn-sae}
        mov qword ptr [rcx],rax
```

```
        vcvtsd2usi rax,xmm1{rd-sae}
        mov qword ptr [rcx+8],rax

        vcvtsd2usi rax,xmm1{ru-sae}
        mov qword ptr [rcx+16],rax

        vcvtsd2usi rax,xmm1{rz-sae}
        mov qword ptr [rcx+24],rax
        ret
Avx512CvtF64ToU64_ endp

; extern "C" void Avx512CvtF64ToF32_(float val_cvt[4], double val);

Avx512CvtF64ToF32_ proc
        vcvtsd2ss xmm2,xmm2,xmm1{rn-sae}
        vmovss real4 ptr [rcx],xmm2

        vcvtsd2ss xmm2,xmm2,xmm1{rd-sae}
        vmovss real4 ptr [rcx+4],xmm2

        vcvtsd2ss xmm2,xmm2,xmm1{ru-sae}
        vmovss real4 ptr [rcx+8],xmm2

        vcvtsd2ss xmm2,xmm2,xmm1{rz-sae}
        vmovss real4 ptr [rcx+12],xmm2
        ret
Avx512CvtF64ToF32_ endp
        end
```

Код C++ в листинге 13.3 начинается с функции `ConvertF32ToU32`. Эта функция инициализирует тестовый пример и вызывает функцию на языке ассемблера `Avx512CvtF32ToU32_`, которая преобразует значение с плавающей запятой одинарной точности в целочисленное двойное слово без знака (32-битное), используя различные режимы округления. Затем результаты передаются в `cout`. Функции C++ `ConvertF64ToU64` и `ConvertF64ToF32` выполняют аналогичную инициализацию тестового примера для функций на языке ассемблера `Avx512CvtF64ToU64_` и `Avx512CvtF64ToF32_` соответственно.

Первая команда функции на языке ассемблера `Avx512CvtF32ToU32_`, `vcvtss2usi eax,xmm1 {rnsae}` преобразует скалярное значение с плавающей запятой одинарной точности в XMM1 (или `val`) в целочисленное двойное слово без знака, используя режим округления до ближайшего. Как упоминалось в главе 12, суффикс `-sae`, добавляемый к параметру встроенного режима округления, является напоминанием о том, что исключения с плавающей запятой и обновления флага MXCSR всегда отключаются, если указан операнд управления округлением на уровне команды. Следующая команда `mov dword ptr [rcx],eax` сохраняет преобразованный результат в `val_cvt[0]`. Затем `Avx512CvtF32ToU32_` использует дополнительные команды `vcvtss2usi` для выполнения той же операции преобразования с использованием режимов округления: в меньшую, большую и в сторону нуля. Организация функции `Avx512CvtF64ToU64_` аналогична `Avx512CvtF32ToU32_` и использует команду `vcvtsd2usi` для преобразования значения с плавающей запятой двойной точности в целое число без знака в формате четверного слова. Обратите внимание, что и `vcvtss2usi`, и `vcvtsd2usi` – это новые

команды AVX-512. Набор AVX-512 также включает команды `vcvtusi2s[d|s]`, которые выполняют преобразования беззнаковых целых чисел в числа с плавающей запятой. Ни AVX, ни AVX2 не содержат команды, выполняющие такие преобразования.

Последняя функция на языке ассемблера, `Avx512CvtF64ToF32_`, применяет команду `vcvtsd2ss` для преобразования значения с плавающей запятой двойной точности в значение с плавающей запятой одинарной точности. Команда `vcvtsd2ss` входит в набор команд AVX и может использоваться с операндом управления округлением на уровне команд в системах, поддерживающих AVX-512. Вот результат выполнения примера исходного кода `Ch13_03`.

```
ConvertF32ToU32
  Пример #0 val = 3.14159
    val_cvt[0] {rn-sae} = 3
    val_cvt[1] {rd-sae} = 3
    val_cvt[2] {ru-sae} = 4
    val_cvt[3] {rz-sae} = 3
  Пример #1 val = 1.41421
    val_cvt[0] {rn-sae} = 1
    val_cvt[1] {rd-sae} = 1
    val_cvt[2] {ru-sae} = 2
    val_cvt[3] {rz-sae} = 1

ConvertF64ToU64
  Пример #0 val = 3.14159
    val_cvt[0] {rn-sae} = 3
    val_cvt[1] {rd-sae} = 3
    val_cvt[2] {ru-sae} = 4
    val_cvt[3] {rz-sae} = 3
  Пример #1 val = 1.41421
    val_cvt[0] {rn-sae} = 1
    val_cvt[1] {rd-sae} = 1
    val_cvt[2] {ru-sae} = 2
    val_cvt[3] {rz-sae} = 1

ConvertF64ToF32
  Пример #0 val = 3.1415927
    val_cvt[0] {rn-sae} = 3.1415927
    val_cvt[1] {rd-sae} = 3.1415925
    val_cvt[2] {ru-sae} = 3.1415927
    val_cvt[3] {rz-sae} = 3.1415925
  Пример #1 val = -1.4142136
    val_cvt[0] {rn-sae} = -1.4142135
    val_cvt[1] {rd-sae} = -1.4142137
    val_cvt[2] {ru-sae} = -1.4142135
    val_cvt[3] {rz-sae} = -1.4142135
```

13.2. Упакованные числа с плавающей запятой

Примеры исходного кода в этом разделе показывают, как использовать команды AVX-512 для выполнения вычислений с использованием упакованных операндов с плавающей запятой. Первые три примера исходного кода демонстрируют основные операции с 512-битными упакованными операнда-

ми с плавающей запятой, включая простую арифметику, операции сравнения и маскирование слиянием. Остальные примеры посвящены конкретным алгоритмам, включая вычисления векторных перекрестных произведений, умножение матрицы на вектор и свертки.

13.2.1. Арифметика упакованных чисел с плавающей запятой

В листинге 13.4 показан исходный код примера `Ch13_04`. В этом примере показано, как выполнять общие арифметические операции с использованием операндов с плавающей запятой одинарной и двойной точности шириной 512 бит. Он также подчеркивает некоторые элементы сходства между программированием AVX/AVX2 и AVX-512.

Листинг 13.4. Пример Ch13_04

```
//-----------------------------------------------
//                     ZmmVal.h
//-----------------------------------------------

#pragma once
#include <string>
#include <cstdint>
#include <sstream>
#include <iomanip>

struct ZmmVal
{
public:
    union
    {
        int8_t m_I8[64];
        int16_t m_I16[32];
        int32_t m_I32[16];
        int64_t m_I64[8];
        uint8_t m_U8[64];
        uint16_t m_U16[32];
        uint32_t m_U32[16];
        uint64_t m_U64[8];
        float m_F32[16];
        double m_F64[8];
    };

//-----------------------------------------------
//              Ch13_04.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#define _USE_MATH_DEFINES
#include <math.h>
#include "ZmmVal.h"

using namespace std;
```

```
extern "C" void Avx512PackedMathF32_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[8]);
extern "C" void Avx512PackedMathF64_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[8]);

void Avx512PackedMathF32(void)
{
    alignas(64) ZmmVal a;
    alignas(64) ZmmVal b;
    alignas(64) ZmmVal c[8];

    a.m_F32[0] = 36.0f;                 b.m_F32[0] = -0.1111111f;
    a.m_F32[1] = 0.03125f;              b.m_F32[1] = 64.0f;
    a.m_F32[2] = 2.0f;                  b.m_F32[2] = -0.0625f;
    a.m_F32[3] = 42.0f;                 b.m_F32[3] = 8.666667f;

    a.m_F32[4] = 7.0f;                  b.m_F32[4] = -18.125f;
    a.m_F32[5] = 20.5f;                 b.m_F32[5] = 56.0f;
    a.m_F32[6] = 36.125f;               b.m_F32[6] = 24.0f;
    a.m_F32[7] = 0.5f;                  b.m_F32[7] = -158.6f;

    a.m_F32[8] = 136.0f;                b.m_F32[8] = -9.1111111f;
    a.m_F32[9] = 2.03125f;              b.m_F32[9] = 864.0f;
    a.m_F32[10] = 32.0f;                b.m_F32[10] = -70.0625f;
    a.m_F32[11] = 442.0f;               b.m_F32[11] = 98.666667f;

    a.m_F32[12] = 57.0f;                b.m_F32[12] = -518.125f;
    a.m_F32[13] = 620.5f;               b.m_F32[13] = 456.0f;
    a.m_F32[14] = 736.125f;             b.m_F32[14] = 324.0f;
    a.m_F32[15] = 80.5f;                b.m_F32[15] = -298.6f;

    Avx512PackedMathF32_(&a, &b, c);

    cout << ("\nРезультаты Avx512PackedMathF32\n");

    for (int i = 0; i < 4; i++)
    {
        cout << "Группа #" << i << '\n';

        cout << "  a:       " << a.ToStringF32(i) << '\n';
        cout << "  b:       " << b.ToStringF32(i) << '\n';
        cout << "  addps:   " << c[0].ToStringF32(i) << '\n';
        cout << "  subps:   " << c[1].ToStringF32(i) << '\n';
        cout << "  mulps:   " << c[2].ToStringF32(i) << '\n';
        cout << "  divps:   " << c[3].ToStringF32(i) << '\n';
        cout << "  absps:   " << c[4].ToStringF32(i) << '\n';
        cout << "  sqrtps:  " << c[5].ToStringF32(i) << '\n';
        cout << "  minps:   " << c[6].ToStringF32(i) << '\n';
        cout << "  maxps:   " << c[7].ToStringF32(i) << '\n';

        cout << '\n';
    }
}

void Avx512PackedMathF64(void)
{
    alignas(64) ZmmVal a;
    alignas(64) ZmmVal b;
    alignas(64) ZmmVal c[8];
```

```
    a.m_F64[0] = 2.0;           b.m_F64[0] = M_PI;
    a.m_F64[1] = 4.0 ;          b.m_F64[1] = M_E;

    a.m_F64[2] = 7.5;           b.m_F64[2] = -9.125;
    a.m_F64[3] = 3.0;           b.m_F64[3] = -M_PI;

    a.m_F64[4] = 12.0;          b.m_F64[4] = M_PI / 2;
    a.m_F64[5] = 24.0;          b.m_F64[5] = M_E / 2;

    a.m_F64[6] = 37.5;          b.m_F64[6] = -9.125 / 2;
    a.m_F64[7] = 43.0;          b.m_F64[7] = -M_PI / 2;

    Avx512PackedMathF64_(&a, &b, c);
    cout << ("\nРезультаты Avx512PackedMathF64\n");

    for (int i = 0; i < 4; i++)
    {
        cout << "Группа #" << i << '\n';

        cout << "  a:       " << a.ToStringF64(i) << '\n';
        cout << "  b:       " << b.ToStringF64(i) << '\n';
        cout << "  addpd:   " << c[0].ToStringF64(i) << '\n';
        cout << "  subpd:   " << c[1].ToStringF64(i) << '\n';
        cout << "  mulpd:   " << c[2].ToStringF64(i) << '\n';
        cout << "  divpd:   " << c[3].ToStringF64(i) << '\n';
        cout << "  abspd:   " << c[4].ToStringF64(i) << '\n';
        cout << "  sqrtpd:  " << c[5].ToStringF64(i) << '\n';
        cout << "  minpd:   " << c[6].ToStringF64(i) << '\n';
        cout << "  maxpd:   " << c[7].ToStringF64(i) << '\n';

        cout << '\n';
    }
}

int main()
{
    Avx512PackedMathF32();
    Avx512PackedMathF64();
    return 0;
}

;-----------------------------------------------
;              Ch13_04.asm
;-----------------------------------------------

; Значения маски, используемые для вычисления абсолютных значений с плавающей запятой
ConstVals   segment readonly align(64) 'const'
AbsMaskF32  dword 16 dup(7fffffffh)
AbsMaskF64  qword 8 dup(7fffffffffffffffh)
ConstVals   ends

; extern "C" void Avx512PackedMathF32_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[8]);

            .code
Avx512PackedMathF32_ proc
```

```
; Загрузка упакованных значений одинарной точности с плавающей запятой
        vmovaps zmm0,zmmword ptr [rcx]      ;zmm0 = *a
        vmovaps zmm1,zmmword ptr [rdx]      ;zmm1 = *b

; Сложение упакованных значений одинарной точности с плавающей запятой
        vaddps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+0],zmm2

; Вычитание упакованных значений одинарной точности с плавающей запятой
        vsubps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+64],zmm2

; Умножение упакованных значений одинарной точности с плавающей запятой
        vmulps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+128],zmm2

; Деление упакованных значений одинарной точности с плавающей запятой
        vdivps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+192],zmm2

; Абсолютное значение упакованных значений одинарной точности с плавающей запятой (b)
        vandps zmm2,zmm1,zmmword ptr [AbsMaskF32]
        vmovaps zmmword ptr [r8+256],zmm2

; Квадратный корень из упакованных значений одинарной точности с плавающей запятой (a)
        vsqrtps zmm2,zmm0
        vmovaps zmmword ptr [r8+320],zmm2

; Минимум упакованных значений одинарной точности с плавающей запятой
        vminps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+384],zmm2

; Максимум упакованных значений одинарной точности с плавающей запятой
        vmaxps zmm2,zmm0,zmm1
        vmovaps zmmword ptr [r8+448],zmm2

        vzeroupper
        ret
Avx512PackedMathF32_ endp

; extern "C" void Avx512PackedMathF64_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[8]);

Avx512PackedMathF64_ proc

; Загрузка упакованных значений двойной точности с плавающей запятой
        vmovapd zmm0,zmmword ptr [rcx]      ;zmm0 = *a
        vmovapd zmm1,zmmword ptr [rdx]      ;zmm1 = *b

; Сложение упакованных значений двойной точности с плавающей запятой
        vaddpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+0],zmm2

; Вычитание упакованных значений двойной точности с плавающей запятой
        vsubpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+64],zmm2
```

```
; Умножение упакованных значений двойной точности с плавающей запятой
        vmulpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+128],zmm2

; Деление упакованных значений двойной точности с плавающей запятой
        vdivpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+192],zmm2

; Абсолютное значение упакованных значений двойной точности с плавающей запятой (b)
        vandpd zmm2,zmm1,zmmword ptr [AbsMaskF64]
        vmovapd zmmword ptr [r8+256],zmm2

; Квадратный корень из упакованных значений двойной точности с плавающей запятой (a)
        vsqrtpd zmm2,zmm0
        vmovapd zmmword ptr [r8+320],zmm2

; Минимум упакованных значений двойной точности с плавающей запятой
        vminpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+384],zmm2

; Максимум упакованных значений двойной точности с плавающей запятой
        vmaxpd zmm2,zmm0,zmm1
        vmovapd zmmword ptr [r8+448],zmm2

        vzeroupper
        ret
Avx512PackedMathF64_ endp
        end
```

Листинг 13.4 начинается с объявления структуры C++ `ZmmVal` в заголовочном файле `ZmmVal.h`. Эта структура аналогична структурам `XmmVal` и `YmmVal`, которые использовались в примерах исходного кода в главах 6 и 9. Структура `ZmmVal` содержит публичное анонимное объединение, которое упрощает обмен упакованными данными операндов между функциями, написанными на C++ и языке ассемблера x86. Члены этого объединения соответствуют типам упакованных данных, которые могут использоваться с регистром ZMM. Структура `ZmmVal` также включает несколько функций форматирования строк для отображения (исходный код для этих функций-членов не показан).

Оставшийся код C++ в листинге 13.4 аналогичен коду, который использовался в примере `Ch09_01`. Объявления для функций на языке ассемблера `Avx512PackedMathF32_` и `Avx512PackedMathF64_` следуют за объявлением структуры `ZmmVal`. Эти функции выполняют различные упакованные арифметические операции с плавающей запятой одинарной и двойной точности с использованием предоставленных аргументов `ZmmVal`. Функции C++ `Avx512PackedMathF32` и `Avx512PackedMathF64` выполняют инициализацию переменных `ZmmVal`, вызывают функции вычисления на языке ассемблера и отображают результаты. Обратите внимание, что спецификатор `alignas(64)` используется с каждым определением переменной `ZmmVal`.

Код на языке ассемблера в листинге 13.4 начинается с выровненного по 64 байтам пользовательского сегмента памяти с именем `ConstVals`. Этот сегмент содержит определения упакованных значений констант, которые ис-

пользуются в вычислительных функциях. Здесь используется настраиваемый сегмент, поскольку директива MASM `align` не поддерживает выравнивание элементов данных по 64-байтовой границе. Глава 9 содержит дополнительную информацию о настраиваемых сегментах памяти. Сегмент `ConstVals` содержит константы `AbsMaskF32` и `AbsMaskF64`, которые применяются для вычисления абсолютных значений для 512-битных упакованных чисел с плавающей запятой одинарной и двойной точности.

Первая команда функции `Avx512PackedMathF32_`, `vmovaps zmm0,ymmword ptr [rcx]`, загружает аргумент `a` (16 значений с плавающей запятой в `ZmmVal a`) в регистр YMM0. Здесь можно использовать `vmovaps`, поскольку `ZmmVal a` был определен с использованием спецификатора `alignas(64)`. Оператор `zmmword ptr` указывает ассемблеру обрабатывать ячейку памяти, на которую указывает RCX, как 512-битный операнд. Как и операторы `xmmword ptr` и `ymmword ptr`, оператор `zmmword ptr` часто используется для улучшения читаемости кода, даже если он не требуется явно. Следующая команда `vmovaps zmm1,zmmword ptr [rdx]` загружает `ZmmVal b` в регистр ZMM1. Затем команда `vaddps zmm2,zmm0,zmm1` суммирует упакованные значения с плавающей запятой одинарной точности в ZMM0 и ZMM1 и сохраняет результат в ZMM2. Команда `vmovaps zmmword ptr [r8],zmm2` сохраняет упакованные суммы в `c[0]`.

Далее команды `vsubps`, `vmulps` и `vdivps` выполняют вычитание, умножение и деление упакованных чисел с плавающей запятой одинарной точности. За ними следует команда `vandps zmm2,zmm1,zmmword ptr [AbsMaskF32]`, которая вычисляет упакованные абсолютные значения с использованием аргумента `b`. Остальные команды в `Avx512PackedMathF32_` вычисляют упакованные квадратные корни с плавающей запятой одинарной точности, минимумы и максимумы.

Перед своей командой `ret` функция `AvxPackedMath32_` использует команду `vzeroupper`, которая обнуляет старшие 384 бита регистров ZMM0–ZMM15. Как объяснялось в главе 4, команда `vzeroupper` используется здесь, чтобы избежать потенциальных задержек производительности, которые могут возникнуть всякий раз, когда процессор переходит от выполнения кода x86-AVX к выполнению кода x86-SSE. Любая функция на языке ассемблера, использующая один или несколько регистров YMM или ZMM и вызываемая из кода, который потенциально использует команды x86-SSE, должна гарантировать выполнение команды `vzeroupper` до того, как управление программой будет передано обратно вызывающей функции. Следует отметить, что согласно «Справочному руководству по оптимизации архитектур Intel 64 и IA-32» рекомендации по использованию `vzeroupper` применяются к функциям, которые используют команды x86-AVX с регистрами ZMM0–ZMM15 или YMM0–YMM15. Функции, которые используют только регистры ZMM16–ZMM31 или YMM16–YMM31, не нуждаются в соблюдении рекомендаций по использованию `vzeroupper`.

Структура функции `Avx512PackedMathF64_` аналогична `Avx512PackedMathF32_`. Функция `Avx512PackedMathF64_` выполняет свои вычисления с применением версий с двойной точностью тех же команд AVX-512, которые используются в `Avx512PackedMathF32_`. Результаты выполнения примера `Ch13_04` выглядят следующим образом:

```
Результаты Avx512PackedMathF32

Группа #0
  a:              36.000000       0.031250 |       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:        -324.000031       0.000488 |     -32.000000       4.846154
  absps:          0.111111      64.000000 |       0.062500       8.666667
  sqrtps:         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

Группа #1
  a:               7.000000      20.500000 |      36.125000       0.500000
  b:             -18.125000      56.000000 |      24.000000    -158.600006
  addps:         -11.125000      76.500000 |      60.125000    -158.100006
  subps:          25.125000     -35.500000 |      12.125000     159.100006
  mulps:        -126.875000    1148.000000 |     867.000000     -79.300003
  divps:          -0.386207       0.366071 |       1.505208      -0.003153
  absps:          18.125000      56.000000 |      24.000000     158.600006
  sqrtps:          2.645751       4.527693 |       6.010407       0.707107
  minps:         -18.125000      20.500000 |      24.000000    -158.600006
  maxps:           7.000000      56.000000 |      36.125000       0.500000

Группа #2
  a:             136.000000       2.031250 |      32.000000     442.000000
  b:              -9.111111     864.000000 |     -70.062500      98.666664
  addps:         126.888885     866.031250 |     -38.062500     540.666687
  subps:         145.111115    -861.968750 |     102.062500     343.333344
  mulps:       -1239.111084    1755.000000 |   -2242.000000   43610.664063
  divps:         -14.926830       0.002351 |      -0.456735       4.479730
  absps:           9.111111     864.000000 |      70.062500      98.666664
  sqrtps:         11.661903       1.425219 |       5.656854      21.023796
  minps:          -9.111111       2.031250 |     -70.062500      98.666664
  maxps:         136.000000     864.000000 |      32.000000     442.000000

Группа #3
  a:              57.000000     620.500000 |     736.125000      80.500000
  b:            -518.125000     456.000000 |     324.000000    -298.600006
  addps:        -461.125000    1076.500000 |    1060.125000    -218.100006
  subps:         575.125000     164.500000 |     412.125000     379.100006
  mulps:      -29533.125000  282948.000000 |  238504.500000  -24037.300781
  divps:          -0.110012       1.360746 |       2.271991      -0.269591
  absps:         518.125000     456.000000 |     324.000000     298.600006
  sqrtps:          7.549834      24.909838 |      27.131624       8.972179
  minps:        -518.125000     456.000000 |     324.000000    -298.600006
  maxps:          57.000000     620.500000 |     736.125000      80.500000

Результаты Avx512PackedMathF64

Группа #0
  a:                        2.000000000000 |               4.000000000000
  b:                        3.141592653590 |               2.718281828459
```

```
  addpd:                5.141592653590 |               6.718281828459
  subpd:               -1.141592653590 |               1.281718171541
  mulpd:                6.283185307180 |              10.873127313836
  divpd:                0.636619772368 |               1.471517764686
  abspd:                3.141592653590 |               2.718281828459
  sqrtpd:               1.414213562373 |               2.000000000000
  minpd:                2.000000000000 |               2.718281828459
  maxpd:                3.141592653590 |               4.000000000000

Группа #1
  a:                    7.500000000000 |               3.000000000000
  b:                   -9.125000000000 |              -3.141592653590
  addpd:               -1.625000000000 |              -0.141592653590
  subpd:               16.625000000000 |               6.141592653590
  mulpd:              -68.437500000000 |              -9.424777960769
  divpd:               -0.821917808219 |              -0.954929658551
  abspd:                9.125000000000 |               3.141592653590
  sqrtpd:               2.738612787526 |               1.732050807569
  minpd:               -9.125000000000 |              -3.141592653590
  maxpd:                7.500000000000 |               3.000000000000

Группа #2
  a:                   12.000000000000 |              24.000000000000
  b:                    1.570796326795 |               1.359140914230
  addpd:               13.570796326795 |              25.359140914230
  subpd:               10.429203673205 |              22.640859085770
  mulpd:               18.849555921539 |              32.619381941509
  divpd:                7.639437268411 |              17.658213176229
  abspd:                1.570796326795 |               1.359140914230
  sqrtpd:               3.464101615138 |               4.898979485566
  minpd:                1.570796326795 |               1.359140914230
  maxpd:               12.000000000000 |              24.000000000000

Группа #3
  a:                   37.500000000000 |              43.000000000000
  b:                   -4.562500000000 |              -1.570796326795
  addpd:               32.937500000000 |              41.429203673205
  subpd:               42.062500000000 |              44.570796326795
  mulpd:             -171.093750000000 |             -67.544242052181
  divpd:               -8.219178082192 |             -27.374650211806
  abspd:                4.562500000000 |               1.570796326795
  sqrtpd:               6.123724356958 |               6.557438524302
  minpd:               -4.562500000000 |              -1.570796326795
  maxpd:               37.500000000000 |              43.000000000000
```

13.2.2. Сравнение упакованных чисел с плавающей запятой

В главе 6 вы узнали, как использовать команды `vcmpp[s|d]` для выполнения операций сравнения упакованных чисел одинарной и двойной точности с плавающей запятой (см. пример исходного кода `Ch06_02`). Напомним, что версия этих команд для AVX устанавливает для элементов операнда SIMD все нули или все единицы, чтобы указать результат операции сравнения. В этом разделе вы узнаете, как использовать версию AVX-512 команды `vcmpps`, которая сохраняет результат сравнения в регистре маски операции. В листинге 13.5 показан код примера `Ch13_05`.

Листинг 13.5. Пример Ch13_05

```
//-----------------------------------------------
//               Ch13_05.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#include <limits>
#include "ZmmVal.h"

using namespace std;

extern "C" void Avx512PackedCompareF32_(const ZmmVal* a, const ZmmVal* b, uint16_t c[8]);

const char* c_CmpStr[8] =
{
    "EQ", "NE", "LT", "LE", "GT", "GE", "ORDERED", "UNORDERED"
};

void ToZmmVal(ZmmVal des[8], uint16_t src[8])
{
    for (size_t i = 0; i < 8; i++)
    {
        uint16_t val_src = src[i];

        for (size_t j = 0; j < 16; j++)
            des[i].m_U32[j] = val_src & (1 << j) ? 1 : 0;
    }
}

void Avx512PackedCompareF32(void)
{
    alignas(64) ZmmVal a;
    alignas(64) ZmmVal b;
    uint16_t c[8];

    a.m_F32[0] = 2.0;       b.m_F32[0] = 1.0;
    a.m_F32[1] = 7.0;       b.m_F32[1] = 12.0;
    a.m_F32[2] = -6.0;      b.m_F32[2] = -6.0;
    a.m_F32[3] = 3.0;       b.m_F32[3] = 8.0;

    a.m_F32[4] = -2.0;      b.m_F32[4] = 1.0;
    a.m_F32[5] = 17.0;      b.m_F32[5] = 17.0;
    a.m_F32[6] = 6.5;       b.m_F32[6] = -9.125;
    a.m_F32[7] = 4.875;     b.m_F32[7] = numeric_limits<float>::quiet_NaN();

    a.m_F32[8] = 2.0;       b.m_F32[8] = 101.0;
    a.m_F32[9] = 7.0;       b.m_F32[9] = -312.0;
    a.m_F32[10] = -5.0;     b.m_F32[10] = 15.0;
    a.m_F32[11] = -33.0;    b.m_F32[11] = -33.0;

    a.m_F32[12] = -12.0;    b.m_F32[12] = 198.0;
    a.m_F32[13] = 107.0;    b.m_F32[13] = 107.0;
    a.m_F32[14] = 16.125;   b.m_F32[14] = -2.75;
    a.m_F32[15] = 42.875;   b.m_F32[15] = numeric_limits<float>::quiet_NaN();
```

```
    Avx512PackedCompareF32_(&a, &b, c);

    cout << "\nРезультаты Avx512PackedCompareF32\n";

    ZmmVal c_display[8];

    ToZmmVal(c_display, c);

    for (int sel = 0; sel < 4; sel++)
    {
        cout << setw(12) << "a[" << sel << "]:" << a.ToStringF32(sel) << '\n';
        cout << setw(12) << "b[" << sel << "]:" << b.ToStringF32(sel) << '\n';
        cout << '\n';

        for (int j = 0; j < 8; j++)
            cout << setw(14) << c_CmpStr[j] << ':' << c_display[j].ToStringU32(sel)  <<
'\n';
        cout << '\n';
    }
}

int main()
{
    Avx512PackedCompareF32();
    return 0;
}

;-------------------------------------------------
;               Ch13_05.asm
;-------------------------------------------------

        include <cmpequ.asmh>

; extern "C" void Avx512PackedCompareF32_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[8]);

        .code
Avx512PackedCompareF32_ proc
        vmovaps zmm0,[rcx]                  ;zmm0 = a
        vmovaps zmm1,[rdx]                  ;zmm1 = b

; Сравнение упакованных чисел по условию EQUAL (равно)
        vcmpps k1,zmm0,zmm1,CMP_EQ
        kmovw word ptr [r8],k1

; Сравнение упакованных чисел по условию NOT EQUAL (не равно)
        vcmpps k1,zmm0,zmm1,CMP_NEQ
        kmovw word ptr [r8+2],k1

; Сравнение упакованных чисел по условию LESS THAN (меньше, чем)
        vcmpps k1,zmm0,zmm1,CMP_LT
        kmovw word ptr [r8+4],k1

; Сравнение упакованных чисел по условию LESS THAN OR EQUAL (меньше или равно)
        vcmpps k1,zmm0,zmm1,CMP_LE
        kmovw word ptr [r8+6],k1
```

```
; Сравнение упакованных чисел по условию GREATER THAN (больше, чем)
        vcmpps k1,zmm0,zmm1,CMP_GT
        kmovw word ptr [r8+8],k1

; Сравнение упакованных чисел по условию GREATER THAN OR EQUAL (больше или равно)
        vcmpps k1,zmm0,zmm1,CMP_GE
        kmovw word ptr [r8+10],k1

; Сравнение упакованных чисел по условию ORDERED (упорядочено)
        vcmpps k1,zmm0,zmm1,CMP_ORD
        kmovw word ptr [r8+12],k1

; Сравнение упакованных чисел по условию UNORDERED (не упорядочено)
        vcmpps k1,zmm0,zmm1,CMP_UNORD
        kmovw word ptr [r8+14],k1

        vzeroupper
        ret
Avx512PackedCompareF32_ endp
        end
```

Функция `Avx512PackedCompareF32` на языке C++, показанная в листинге 13.5, начинает свое выполнение с загрузки тестовых значений в элементы с плавающей запятой одинарной точности переменных `ZmmVal` `a` и `b`. Обратите внимание, что эти переменные определены с помощью спецификатора C++ `alignas(64)`. После инициализации переменной функция `Avx512PackedCompareF32` вызывает функцию на языке ассемблера `Avx512PackedCompareF32_` для выполнения сравнений упакованных чисел. Затем она передает результаты в `cout`.

Функция на языке ассемблера `Avx512PackedCompareF32_` начинает свое выполнение с двух команд `vmovaps`, которые загружают переменные `a` и `b` `ZmmVal` в регистры ZMM0 и ZMM1 соответственно. Далее команда `vcmpps k1,zmm0,zmm1,CMP_EQ` сравнивает элементы с плавающей запятой одинарной точности в регистрах ZMM0 и ZMM1 на предмет равенства. Для каждой позиции элемента эта команда устанавливает соответствующую битовую позицию в регистре маски K1 в единицу, если значения в ZMM0 и ZMM1 равны; в противном случае бит регистра маски устанавливается в ноль. Эта операция наглядно показана на рис. 13.1. Затем команда `kmovw word ptr [r8],k1` сохраняет результирующую маску в `c[0]`.

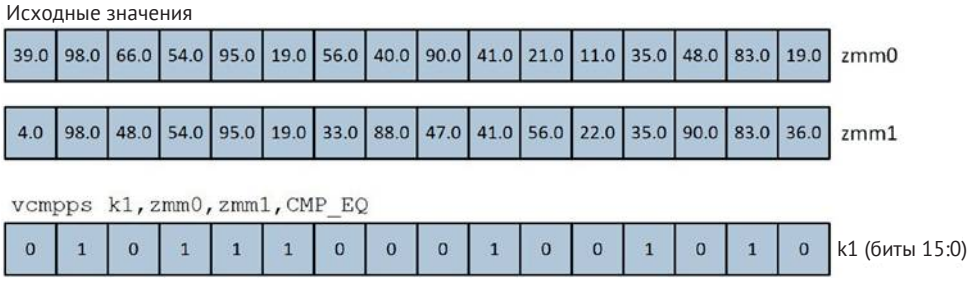


Рис. 13.1. Пример выполнения команды `vcmpps k1,zmm0,zmm1,CMP_EQ`

Оставшийся код в `Avx512PackedCompareF32_` демонстрирует дополнительные операции сравнения с использованием команды `vcmpps`, переменных `ZmmVal a` и `b` и общих предикатов сравнения. Обратите внимание, что, как и в предыдущем примере, `Avx512PackedCompareF32_` использует команду `vzeroupper` перед своей командой `ret`. Ниже показаны результаты выполнения примера `Ch13_05`.

```
Результаты Avx512PackedCompareF32

         a[0]:        2.000000       7.000000 |       -6.000000       3.000000
         b[0]:        1.000000      12.000000 |       -6.000000       8.000000

           EQ:               0              0 |              1              0
           NE:               1              1 |              0              1
           LT:               0              1 |              0              1
           LE:               0              1 |              1              1
           GT:               1              0 |              0              0
           GE:               1              0 |              1              0
      ORDERED:               1              1 |              1              1
    UNORDERED:               0              0 |              0              0

         a[1]:       -2.000000      17.000000 |        6.500000       4.875000
         b[1]:        1.000000      17.000000 |       -9.125000            nan

           EQ:               0              1 |              0              0
           NE:               1              0 |              1              1
           LT:               1              0 |              0              0
           LE:               1              1 |              0              0
           GT:               0              0 |              1              0
           GE:               0              1 |              1              0
      ORDERED:               1              1 |              1              0
    UNORDERED:               0              0 |              0              1

         a[2]:        2.000000       7.000000 |       -5.000000     -33.000000
         b[2]:      101.000000    -312.000000 |       15.000000     -33.000000

           EQ:               0              0 |              0              1
           NE:               1              1 |              1              0
           LT:               1              0 |              1              0
           LE:               1              0 |              1              1
           GT:               0              1 |              0              0
           GE:               0              1 |              0              1
      ORDERED:               1              1 |              1              1
    UNORDERED:               0              0 |              0              0

         a[3]:      -12.000000     107.000000 |       16.125000      42.875000
         b[3]:      198.000000     107.000000 |       -2.750000            nan

           EQ:               0              1 |              0              0
           NE:               1              0 |              1              1
           LT:               1              0 |              0              0
           LE:               1              1 |              0              0
           GT:               0              0 |              1              0
           GE:               0              1 |              1              0
      ORDERED:               1              1 |              1              0
    UNORDERED:               0              0 |              0              1
```

В системах, поддерживающих AVX-512, функции на языке ассемблера также могут использовать команду vcmppd с регистром маски операнда-приемника для выполнения сравнений упакованных чисел с плавающей запятой двойной точности. В этих случаях результирующая маска сохраняется в восьми младших битах регистра маски операнда-приемника.

13.2.3. Средние значения столбца упакованных чисел с плавающей запятой

В листинге 13.6 показан исходный код примера Ch13_06. В этом примере, который является версией для AVX-512 исходного кода примера Ch09_03, вычисляются средние значения столбца двумерного массива чисел с плавающей запятой двойной точности. Чтобы сделать текущий пример исходного кода немного более интересным, средние значения столбца вычисляются с использованием только тех элементов массива, которые превышают заранее определенное пороговое значение.

Листинг 13.6. Пример Ch13_06

```
//-----------------------------------------------
//              Ch13_06.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <memory>

using namespace std;

// Проверка размерности в качестве иллюстрации проверки аргументов
extern "C" size_t c_NumRowsMax = 1000000;
extern "C" size_t c_NumColsMax = 1000000;

extern "C" bool Avx512CalcColumnMeans_(const double* x, size_t nrows, size_t ncols, double*
col_means, size_t* col_counts, double x_min);

void Init(double* x, size_t n, int rng_min, int rng_max, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {rng_min, rng_max};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (double)ui_dist(rng);
}

bool Avx512CalcColumnMeansCpp(const double* x, size_t nrows, size_t ncols, double* col_
means, size_t* col_counts, double x_min)
{
    // Проверка корректности nrows и ncols
    if (nrows == 0 || nrows > c_NumRowsMax)
        return false;
    if (ncols == 0 || ncols > c_NumColsMax)
        return false;
```

```
    // Инициализация среднего значения и счетчика нулевым значением
    for (size_t i = 0; i < ncols; i++)
    {
        col_means[i] = 0.0;
        col_counts[i] = 0;
    }

    // Вычисление среднего значения столбца
    for (size_t i = 0; i < nrows; i++)
    {
        for (size_t j = 0; j < ncols; j++)
        {
            double val = x[i * ncols + j];

            if (val >= x_min)
            {
                col_means[j] += val;
                col_counts[j]++;
            }
        }
    }

    for (size_t j = 0; j < ncols; j++)
        col_means[j] /= col_counts[j];

    return true;
}

void Avx512CalcColumnMeans(void)
{
    const size_t nrows = 20000;
    const size_t ncols = 23;
    const int rng_min = 1;
    const int rng_max = 999;
    const unsigned int rng_seed = 47;
    const double x_min = 75.0;

    unique_ptr<double[]> x {new double[nrows * ncols]};
    unique_ptr<double[]> col_means1 {new double[ncols]};
    unique_ptr<double[]> col_means2 {new double[ncols]};
    unique_ptr<size_t[]> col_counts1 {new size_t[ncols]};
    unique_ptr<size_t[]> col_counts2 {new size_t[ncols]};

    Init(x.get(), nrows * ncols, rng_min, rng_max, rng_seed);

    bool rc1 = Avx512CalcColumnMeansCpp(x.get(), nrows, ncols, col_means1.get(), col_
counts1.get(), x_min);
    bool rc2 = Avx512CalcColumnMeans_(x.get(), nrows, ncols, col_means2.get(), col_counts2.
get(), x_min);

    cout << "Результаты Avx512CalcColumnMeans\n";

    if (!rc1 || !rc2)
    {
        cout << "Ошибочный возвращаемый код: ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
```

```
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        return;
    }

    cout << "Тестовая матрица (nrows = " << nrows << ", ncols = " << ncols << ")\n";
    cout << "\nСреднее значение столбца\n";
    cout << fixed << setprecision(4);

    for (size_t j = 0; j < ncols; j++)
    {
        cout << setw(4) << j << ": ";
        cout << "col_means = ";
        cout << setw(10) << col_means1[j] << ", ";
        cout << setw(10) << col_means2[j] << "    ";

        cout << "col_counts = ";
        cout << setw(6) << col_counts1[j] << ", ";
        cout << setw(6) << col_counts2[j] << '\n';

        if (col_means1[j] != col_means2[j])
            cout << "ошибка сравнения col_means\n";

        if (col_counts1[j] != col_counts2[j])
            cout << "ошибка сравнения col_counts\n";
    }
}

int main()
{
    Avx512CalcColumnMeans();
    return 0;
}

;-----------------------------------------------
;               Ch13_06.asm
;-----------------------------------------------

        include <cmpequ.asmh>
        include <MacrosX86-64-AVX.asmh>

        extern c_NumRowsMax:qword
        extern c_NumColsMax:qword

; extern "C" bool Avx512CalcColumnMeans_(const double* x, size_t nrows, size_t ncols,
double* col_means, size_t* col_counts, double x_min);

        .code
Avx512CalcColumnMeans_ proc frame
        _CreateFrame CCM_,0,0,rbx,r12,r13
        _EndProlog

; Проверка nrows и ncols
        xor eax,eax                     ;возвращаемый код ошибки
        test rdx,rdx
        jz Done                         ;переход, если nrows равен нулю
        cmp rdx,[c_NumRowsMax]
```

```
        ja Done                             ;переход, если nrows слишком велик
        test r8,r8
        jz Done                             ;переход, если ncols равен нулю
        cmp r8,[c_NumColsMax]
        ja Done                             ;переход, если ncols слишком велик

; Чтение значений аргументов col_counts и x_min
        mov ebx,1
        vpbroadcastq zmm4,rbx               ;zmm4 = 8 четверных слов значения 1
        mov rbx,[rbp+CCM_OffsetStackArgs]   ;rbx = указатель col_counts
        lea r13,[rbp+CCM_OffsetStackArgs+8] ;r13 = указатель на x_min

; Обнуление начальных значений col_means и col_counts
        xor r10,r10
        vxorpd xmm0,xmm0,xmm0
@@:     vmovsd real8 ptr[r9+rax*8],xmm0         ;col_means[i] = 0.0
        mov [rbx+rax*8],r10                     ;col_counts[i] = 0
        inc rax
        cmp rax,r8
        jne @B                                  ;повтор до завершения цикла

; Вычисление суммы каждого столбца массива x
LP1:    xor r10,r10                             ;r10 = col_index
        mov r11,r9                              ;r11 = указатель на col_means
        mov r12,rbx                             ;r12 = указатель на col_counts

LP2:    mov rax,r10                             ;rax = col_index
        add rax,8
        cmp rax,r8                              ;осталось ли 8 и более столбцов?
        ja @F                                   ;переход, если col_index + 8 > ncols

; Обновление значений col_means и col_counts с использованием следующих 8 столбцов
        vmovupd zmm0,zmmword ptr [rcx]          ;чтение 8 столбцов текущей строки
        vcmppd k1,zmm0,real8 bcst [r13],CMP_GE  ;k1 = маска значений >= x_min
        vmovupd zmm1{k1}{z},zmm0                ;значения >= x_min или 0.0
        vaddpd zmm2,zmm1,zmmword ptr [r11]      ;сложение значений с col_means
        vmovupd zmmword ptr [r11],zmm2          ;сохранение обновленного col_means

        vpmovm2q zmm0,k1                        ;преобразование маски в вектор
        vpandq zmm1,zmm0,zmm4                   ;четверные слова значений
        vpaddq zmm2,zmm1,zmmword ptr [r12]      ;обновление col_counts
        vmovdqu64 zmmword ptr [r12],zmm2        ;сохранение обновленного col_counts

        add r10,8                               ;col_index += 8
        add rcx,64                              ;x += 8
        add r11,64                              ;col_means += 8
        add r12,64                              ;col_counts += 8
        jmp NextColSet

; Обновление col_means и col_counts с использованием следующих 4 столбцов
@@:     sub rax,4
        cmp rax,r8                              ;осталось ли 4 или более столбцов?
        ja @F                                   ;переход, если col_index + 4 > ncols

        vmovupd ymm0,ymmword ptr [rcx]          ;переход, если 4 столбца текущей строки
        vcmppd k1,ymm0,real8 bcst [r13],CMP_GE  ;k1 = маска значений >= x_min
        vmovupd ymm1{k1}{z},ymm0                ;значения >= x_min или 0.0
```

```
        vaddpd ymm2,ymm1,ymmword ptr [r11]      ;сложение значений с col_means
        vmovupd ymmword ptr [r11],ymm2          ;сохранение обновленного col_means

        vpmovm2q ymm0,k1                        ;преобразование маски в вектор
        vpandq ymm1,ymm0,ymm4                   ;четверные слова значений
        vpaddq ymm2,ymm1,ymmword ptr [r12]      ;обновление col_counts
        vmovdqu64 ymmword ptr [r12],ymm2        ;сохранение обновленного col_counts

        add r10,4                               ;col_index += 4
        add rcx,32                              ;x += 4
        add r11,32                              ;col_means += 4
        add r12,32                              ;col_counts += 4
        jmp NextColSet

; Обновление col_means и col_counts с использованием следующих 2 столбцов
@@:     sub rax,2
        cmp rax,r8                              ;осталось ли 2 столбца или более?
        ja @F                                   ;переход, если col_index + 2 > ncols

        vmovupd xmm0,xmmword ptr [rcx]          ;чтение 2 столбцов текущей строки
        vcmppd k1,xmm0,real8 bcst [r13],CMP_GE  ;k1 = маска значений >= x_min
        vmovupd xmm1{k1}{z},xmm0                ;значения >= x_min или 0.0
        vaddpd xmm2,xmm1,xmmword ptr [r11]      ;сложение значений с col_means
        vmovupd xmmword ptr [r11],xmm2          ;сохранение обновленного col_means

        vpmovm2q xmm0,k1                        ;преобразование маски в вектор
        vpandq xmm1,xmm0,xmm4                   ;четверные слова значений
        vpaddq xmm2,xmm1,xmmword ptr [r12]      ;обновление col_counts
        vmovdqu64 xmmword ptr [r12],xmm2        ;сохранение обновленного col_counts

        add r10,2                               ;col_index += 2
        add rcx,16                              ;x += 2
        add r11,16                              ;col_means += 2
        add r12,16                              ;col_counts += 2
        jmp NextColSet

; Обновление col_means с использованием последнего столбца текущей строки
@@:     vmovsd xmm0,real8 ptr [rcx]             ;чтение x из последнего столбца
        vcmpsd k1,xmm0,real8 ptr [r13],CMP_GE   ;k1 = маска значений >= x_min
        vmovsd xmm1{k1}{z},xmm1,xmm0            ;значение или 0.0
        vaddsd xmm2,xmm1,real8 ptr [r11]        ;сложение с col_means
        vmovsd real8 ptr [r11],xmm2             ;сохранение обновленного col_means
        kmovb eax,k1                            ;eax = 0 or 1
        add qword ptr [r12],rax                 ;обновление col_counts

        add r10,1                               ;col_index += 1
        add rcx,8                               ;обновление указателя x

NextColSet:
        cmp r10,r8                      ;еще есть столбцы в текущей строке?
        jb LP2                          ;переход, если да
        dec rdx                         ;nrows -= 1
        jnz LP1                         ;переход, если еще есть строки

; Вычисление окончательного col_means
@@:     vmovsd xmm0,real8 ptr [r9]      ;xmm0 = col_means[i]
```

```
        vcvtsi2sd xmm1,xmm1,qword ptr [rbx] ;xmm1 = col_counts[i]
        vdivsd xmm2,xmm0,xmm1               ;вычисление финального среднего значения
        vmovsd real8 ptr [r9],xmm2          ;сохранение col_mean[i]
        add r9,8                            ;обновление указателя col_means
        add rbx,8                           ;обновление указателя col_counts
        sub r8,1                            ;ncols -= 1
        jnz @B                              ;повтор до завершения цикла

        mov eax,1                           ;возвращаемый код успешного выполнения

Done:   _DeleteFrame rbx,r12,r13
        vzeroupper
        ret

Avx512CalcColumnMeans_ endp
        end
```

Функция `Avx512CalcColumnMeansCpp` содержит реализацию на языке C++ алгоритма вычисления средних значений столбцов. Эта функция использует два вложенных цикла `for` для суммирования элементов каждого столбца в двумерном массиве. Во время каждой итерации внутреннего цикла значение элемента массива `x[i][j]` добавляется к текущей сумме столбца в `col_means[j]`, только если оно больше или равно `x_min`. Количество элементов, которые больше или равны `x_min`, в каждом столбце сохраняется в массиве `col_counts`. После циклов суммирования средние значения последнего столбца вычисляются с помощью простого цикла `for`.

Вслед за прологом функция `Avx512CalcColumnMeans_` проверяет значения аргументов `nrows` и `ncols`. Затем она выполняет необходимые инициализации. Команды `mov ebx,1` и `vpbroadcastq zmm4,rbx` загружают значение 1 в каждый элемент четверного слова ZMM4. Регистры RBX и R13 затем инициализируются как указатели на `col_counts` и `x_min` соответственно. В последней задаче инициализации используется простой цикл `for`, который присваивает каждому элементу в `col_means` и `col_counts` нулевое значение.

Подобно примеру исходного кода `Ch09_03`, внутренний цикл `for` в `Avx512CalcColumnMeans_` использует несколько разные последовательности команд для суммирования элементов столбца, которые различаются в зависимости от количества столбцов в массиве (рис. 9.2) и текущего индекса столбца. Для каждой строки элементы в первых восьми столбцах `x` могут быть добавлены к `col_means` при помощи 512-битного сложения упакованных чисел с плавающей запятой двойной точности. Значения элементов остальных столбцов прибавляются к `col_means` с использованием упакованного или скалярного сложения с плавающей запятой двойной точности шириной 512, 256 или 128 бит.

Метка внешнего цикла `LP1` является отправной точкой для добавления элементов из текущей строки `x` в `col_means`. Команда `xor r10,r10` инициализирует `col_index` нулевым значением; команда `mov r11,r9` загружает в R11 указатель на `col_means`; команда `mov r12,rbx` загружает в R12 указатель на `col_counts`. Каждая итерация внутреннего цикла `LP2` начинается с проверки, что в текущей строке доступны для обработки не менее восьми столбцов. Если доступно восемь столбцов, команда `vmovupd zmm0,zmmword ptr [rcx]` загружает следующие

восемь элементов текущей строки в регистр ZMM0. Следующая команда `vcmppd k1,zmm0,real8 bcst [r13],CMP_GE` сравнивает каждый элемент в ZMM0 с `x_min` и устанавливает соответствующую битовую позицию в регистре маски K1, чтобы указать результат. Обратите внимание, что встроенный широковещательный операнд команды `vpcmpd` используется здесь в демонстрационных целях. В этом примере исходного кода было бы более эффективно инициализировать упакованную версию `x_min` до начала циклов обработки. Следующая команда, `vmovupd zmm1 {k1} {z},zmm0`, использует нулевую маскировку для эффективного исключения значений меньше `x_min` из последующих вычислений. Следующие две команды, `vaddpd zmm2,zmm1,zmmword ptr [r11]` и `vmovupd zmmword ptr [r11],zmm2`, обновляют текущие суммы столбцов, которые хранятся в `col_means`. Эта операция наглядно показана на рис. 13.2.

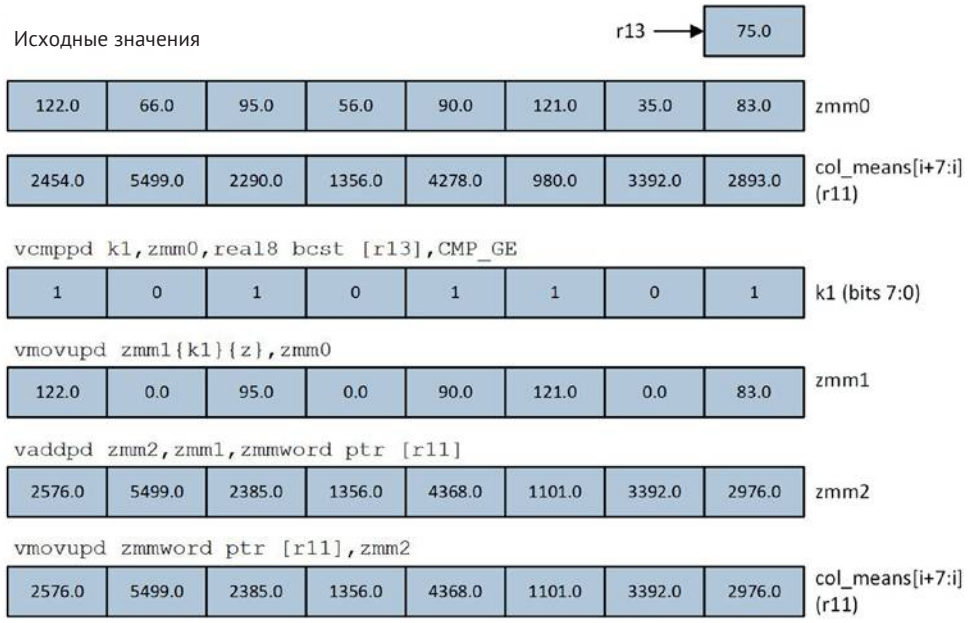


Рис. 13.2. Обновление промежуточных сумм в `col_means` с использованием нулевого слияния

Следующий блок кода обновляет количество элементов в `col_counts`. Команда `vpmovm2q zmm0,k1` (преобразование регистра маски в регистр вектора) устанавливает для каждого элемента четверного слова в ZMM0 все разряды в единицы (`0xFFFFFFFFFFFFFFF`) или нули (`0x0000000000000000`) в соответствии со значением соответствующей битовой позиции в K1. Последующая команда `vpandq zmm1,zmm0, zmm4` обнуляет старшие 63 бита каждого значения четверного слова в ZMM0 и сохраняет этот результат в ZMM1. Следующие две команды, `vpaddq zmm2,zmm1,zmmword ptr [r12]` и `vmovdqu64 zmmword ptr [r12],zmm2`, обновляют значения счетчика в `col_counts`, как показано на рис. 13.3. Команда `vmovdqu64` сохраняет 512-битное упакованное четверное слово в ZMM2 в место, указанное регистром R12. AVX512F также содержит команду `vmovdqu32` для перемещения упакованного двойного слова шириной 512 бит.

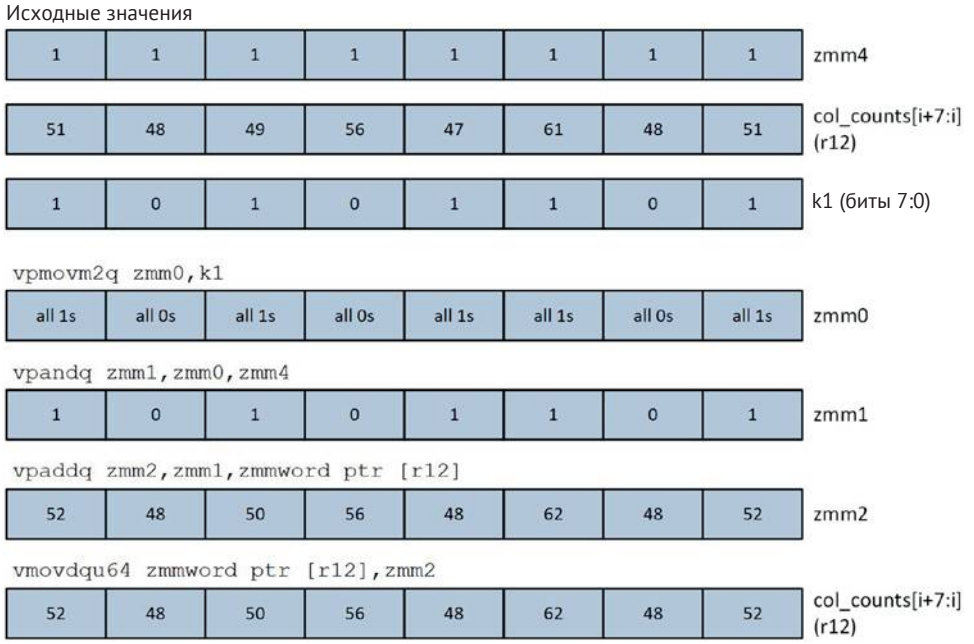


Рис. 13.3. Обновление количества промежуточных элементов в `col_counts`

После выполнения команды `vmovdqu64` различные указатели и счетчики алгоритма обновляются, чтобы отразить факт обработки восьми элементов. Код суммирования повторяется до тех пор, пока количество элементов массива, которые остаются в текущей строке, не станет меньше восьми. Как только это условие выполнено, остальные элементы столбца (если таковые имеются) обрабатываются с использованием операндов шириной 256, 128 или 64 бит, используя тот же метод, описанный в предыдущем абзаце. Обратите внимание, что в функции `Avx512CalcColumnMeans_` применяются команды AVX-512, которые используют регистры YMM или XMM со встроенной широковещательной передачей и нулевыми операндами слияния. Для этих команд требуется процессор, соответствующий стандарту AVX-512, который поддерживает расширение набора команд AVX512VL. После вычисления сумм столбцов каждый элемент в `col_means` делится на соответствующий элемент в `col_counts`, чтобы получить окончательное среднее значение столбца. Вот результаты выполнения примера исходного кода `Ch13_06`:

```
Результаты Avx512CalcColumnMeans

Тестовая матрица (nrows = 20000, ncols = 23)

Средние значения столбцов
   0: col_means =   536.6483,   536.6483   col_counts =   18548,   18548
   1: col_means =   535.8669,   535.8669   col_counts =   18538,   18538
   2: col_means =   534.7049,   534.7049   col_counts =   18457,   18457
   3: col_means =   535.8747,   535.8747   col_counts =   18544,   18544
```

```
  4: col_means =   540.7477,   540.7477   col_counts =   18501,   18501
  5: col_means =   535.9465,   535.9465   col_counts =   18493,   18493
  6: col_means =   539.0142,   539.0142   col_counts =   18528,   18528
  7: col_means =   536.6623,   536.6623   col_counts =   18496,   18496
  8: col_means =   532.1445,   532.1445   col_counts =   18486,   18486
  9: col_means =   543.4736,   543.4736   col_counts =   18479,   18479
 10: col_means =   535.2980,   535.2980   col_counts =   18552,   18552
 11: col_means =   536.4255,   536.4255   col_counts =   18486,   18486
 12: col_means =   537.6472,   537.6472   col_counts =   18473,   18473
 13: col_means =   537.9775,   537.9775   col_counts =   18511,   18511
 14: col_means =   538.4742,   538.4742   col_counts =   18514,   18514
 15: col_means =   539.2965,   539.2965   col_counts =   18497,   18497
 16: col_means =   537.9710,   537.9710   col_counts =   18454,   18454
 17: col_means =   536.7826,   536.7826   col_counts =   18566,   18566
 18: col_means =   538.3274,   538.3274   col_counts =   18452,   18452
 19: col_means =   538.2181,   538.2181   col_counts =   18491,   18491
 20: col_means =   532.6881,   532.6881   col_counts =   18514,   18514
 21: col_means =   537.0067,   537.0067   col_counts =   18554,   18554
 22: col_means =   539.0643,   539.0643   col_counts =   18548,   18548
```

13.2.4. Векторные перекрестные произведения

В примере исходного кода `Ch13_07` демонстрируются вычисления векторных перекрестных произведений с использованием массивов трехмерных векторов. Он также показывает, как выполнять операции сбора и распределения данных с использованием команд AVX-512. В листинге 13.7 показан исходный код примера `Ch13_07`.

Листинг 13.7. Пример Ch13_07

```
//-----------------------------------------------
//               Ch13_07.h
//-----------------------------------------------

#pragma once

// Простая векторная структура
typedef struct
{
    double X;        // компонент X
    double Y;        // компонент Y
    double Z;        // компонент Z
} Vector;

// Векторная структура массивов
typedef struct
{
    double* X;       // указатель на компоненты X
    double* Y;       // указатель на компоненты Y
    double* Z;       // указатель на компоненты Z
} VectorSoA;

// Ch13_07.cpp
void InitVec(Vector* a_aos, Vector* b_aos, VectorSoA& a_soa, VectorSoA& b_soa, size_t
num_vec);
```

```
bool Avx512VcpAosCpp(Vector* c, const Vector* a, const Vector* b, size_t num_vec);
bool Avx512VcpSoaCpp(VectorSoA* c, const VectorSoA* a, const VectorSoA* b, size_t num_vec);

// Ch13_07_.asm
extern "C" bool Avx512VcpAos_(Vector* c, const Vector* a, const Vector* b, size_t num_vec);
extern "C" bool Avx512VcpSoa_(VectorSoA* c, const VectorSoA* a, const VectorSoA* b, size_t
num_vec);

// Ch13_07_BM.cpp
extern void Avx512Vcp_BM(void);

//-----------------------------------------------
//               Ch13_07.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <memory>
#include "Ch13_07.h"
#include "AlignedMem.h"

using namespace std;

void InitVec(Vector* a_aos, Vector* b_aos, VectorSoA& a_soa, VectorSoA& b_soa, size_t
num_vec)
{
    uniform_int_distribution<> ui_dist {1, 100};
    default_random_engine rng {103};

    for (size_t i = 0; i < num_vec; i++)
    {
        double a_x = (double)ui_dist(rng);
        double a_y = (double)ui_dist(rng);
        double a_z = (double)ui_dist(rng);
        double b_x = (double)ui_dist(rng);
        double b_y = (double)ui_dist(rng);
        double b_z = (double)ui_dist(rng);

        a_aos[i].X = a_soa.X[i] = a_x;
        a_aos[i].Y = a_soa.Y[i] = a_y;
        a_aos[i].Z = a_soa.Z[i] = a_z;

        b_aos[i].X = b_soa.X[i] = b_x;
        b_aos[i].Y = b_soa.Y[i] = b_y;
        b_aos[i].Z = b_soa.Z[i] = b_z;
    }
}

void Avx512Vcp(void)
{
    const size_t align = 64;
    const size_t num_vec = 16;

    unique_ptr<Vector> a_aos_up {new Vector[num_vec] };
```

```
unique_ptr<Vector> b_aos_up {new Vector[num_vec] };
unique_ptr<Vector> c_aos_up {new Vector[num_vec] };
Vector* a_aos = a_aos_up.get();
Vector* b_aos = b_aos_up.get();
Vector* c_aos = c_aos_up.get();

VectorSoA a_soa, b_soa, c_soa;
AlignedArray<double> a_soa_x_aa(num_vec, align);
AlignedArray<double> a_soa_y_aa(num_vec, align);
AlignedArray<double> a_soa_z_aa(num_vec, align);
AlignedArray<double> b_soa_x_aa(num_vec, align);
AlignedArray<double> b_soa_y_aa(num_vec, align);
AlignedArray<double> b_soa_z_aa(num_vec, align);
AlignedArray<double> c_soa_x_aa(num_vec, align);
AlignedArray<double> c_soa_y_aa(num_vec, align);
AlignedArray<double> c_soa_z_aa(num_vec, align);
a_soa.X = a_soa_x_aa.Data();
a_soa.Y = a_soa_y_aa.Data();
a_soa.Z = a_soa_z_aa.Data();
b_soa.X = b_soa_x_aa.Data();
b_soa.Y = b_soa_y_aa.Data();
b_soa.Z = b_soa_z_aa.Data();
c_soa.X = c_soa_x_aa.Data();
c_soa.Y = c_soa_y_aa.Data();
c_soa.Z = c_soa_z_aa.Data();

InitVec(a_aos, b_aos, a_soa, b_soa, num_vec);

bool rc1 = Avx512VcpAos_(c_aos, a_aos, b_aos, num_vec);
bool rc2 = Avx512VcpSoa_(&c_soa, &a_soa, &b_soa, num_vec);

cout << "Результаты Avx512VectorCrossProd\n";

if (!rc1 || !rc2)
{
    cout << "Ошибочный возвращаемый код - ";
    cout << "rc1 = " << boolalpha << rc1 << ", ";
    cout << "rc2 = " << boolalpha << rc2 << ", ";
    return;
 }

cout << fixed << setprecision(1);

for (size_t i = 0; i < num_vec; i++)
{
    cout << "Векторное перекрестное произведение #" << i << '\n';

    const unsigned int w = 9;

    cout << "  a:      ";
    cout << setw(w) << a_aos[i].X << ' ';
    cout << setw(w) << a_aos[i].Y << ' ';
    cout << setw(w) << a_aos[i].Z << '\n';

    cout << "  b:      ";
    cout << setw(w) << b_aos[i].X << ' ';
```

```
        cout << setw(w) << b_aos[i].Y << ' ';
        cout << setw(w) << b_aos[i].Z << '\n';

        cout << "  c_aos:  ";
        cout << setw(w) << c_aos[i].X << ' ';
        cout << setw(w) << c_aos[i].Y << ' ';
        cout << setw(w) << c_aos[i].Z << '\n';

        cout << "  c_soa:  ";
        cout << setw(w) << c_soa.X[i] << ' ';
        cout << setw(w) << c_soa.Y[i] << ' ';
        cout << setw(w) << c_soa.Z[i] << '\n';

        bool is_valid_x = c_aos[i].X == c_soa.X[i];
        bool is_valid_y = c_aos[i].Y == c_soa.Y[i];
        bool is_valid_z = c_aos[i].Z == c_soa.Z[i];

        if (!is_valid_x || !is_valid_y || !is_valid_z)
        {
            cout << "Несовпадение элементов с индексом " << i << '\n';
            cout << "  is_valid_x = " << boolalpha << is_valid_x << '\n';
            cout << "  is_valid_y = " << boolalpha << is_valid_y << '\n';
            cout << "  is_valid_z = " << boolalpha << is_valid_z << '\n';
            return;
        }
    }
}

int main()
{
    Avx512Vcp();
    Avx512Vcp_BM();
    return 0;
}

;-----------------------------------------------
;               Ch13_07.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; Индексы для команд сбора и раздачи
ConstVals   segment readonly align(64) 'const'
GS_X        qword 0, 3, 6,  9, 12, 15, 18, 21
GS_Y        qword 1, 4, 7, 10, 13, 16, 19, 22
GS_Z        qword 2, 5, 8, 11, 14, 17, 20, 23
ConstVals   ends

; extern "C" bool Avx512VcpAos_(Vector* c, const Vector* a, const Vector* b, size_t num_
vectors);

        .code
Avx512VcpAos_ proc

; Проверка значения num_vec
        xor eax,eax                         ;возвращаемый код ошибки (also i = 0)
```

```
        test r9,r9
        jz Done                             ;переход, если num_vec равен нулю
        test r9,07h
        jnz Done                            ;переход, если num_vec % 8 != 0 истинно

; Загрузка индексов для операций сбора и раздачи
        vmovdqa64 zmm29,zmmword ptr [GS_X]  ;zmm29 = индексы компонента X
        vmovdqa64 zmm30,zmmword ptr [GS_Y]  ;zmm30 = индексы компонента Y
        vmovdqa64 zmm31,zmmword ptr [GS_Z]  ;zmm31 = индексы компонента Z

; Загрузка следующих 8 векторов
        align 16
@@:     kxnorb k1,k1,k1
        vgatherqpd zmm0{k1},[rdx+zmm29*8]       ;zmm0 = значения A.X

        kxnorb k2,k2,k2
        vgatherqpd zmm1{k2},[rdx+zmm30*8]       ;zmm1 = значения A.Y

        kxnorb k3,k3,k3
        vgatherqpd zmm2{k3},[rdx+zmm31*8]       ;zmm2 = значения A.Z

        kxnorb k4,k4,k4
        vgatherqpd zmm3{k4},[r8+zmm29*8]        ;zmm3 = значения B.X

        kxnorb k5,k5,k5
        vgatherqpd zmm4{k5},[r8+zmm30*8]        ;zmm4 = значения B.Y

        kxnorb k6,k6,k6
        vgatherqpd zmm5{k6},[r8+zmm31*8]        ;zmm5 = значения B.Z

; Вычисление 8 векторных перекрестных произведений
        vmulpd zmm16,zmm1,zmm5
        vmulpd zmm17,zmm2,zmm4
        vsubpd zmm18,zmm16,zmm17                ;c.X = a.Y * b.Z - a.Z * b.Y

        vmulpd zmm19,zmm2,zmm3
        vmulpd zmm20,zmm0,zmm5
        vsubpd zmm21,zmm19,zmm20                ;c.Y = a.Z * b.X - a.X * b.Z

        vmulpd zmm22,zmm0,zmm4
        vmulpd zmm23,zmm1,zmm3
        vsubpd zmm24,zmm22,zmm23                ;c.Z = a.X * b.Y - a.Y * b.X

; Сохранение вычисленных перекрестных произведений
        kxnorb k4,k4,k4
        vscatterqpd [rcx+zmm29*8]{k4},zmm18     ;сохранение компонентов C.X

        kxnorb k5,k5,k5
        vscatterqpd [rcx+zmm30*8]{k5},zmm21     ;сохранение компонентов C.Y

        kxnorb k6,k6,k6
        vscatterqpd [rcx+zmm31*8]{k6},zmm24     ;сохранение компонентов C.Z

; Обновление указателей и счетчиков
        add rcx,192                             ;c += 8
        add rdx,192                             ;a += 8
```

```
        add r8,192                          ;b += 8
        add rax,8                           ;i += 8
        cmp rax,r9
        jb @B

        mov eax,1                           ;возвращаемый код успешного выполнения

Done:   vzeroupper
        ret
Avx512VcpAos_ endp

; extern "C" bool Avx512VcpSoa_(VectorSoA* c, const VectorSoA* a, const VectorSoA* b,
size_t num_vectors);

Avx512VcpSoa_ proc frame
        _CreateFrame CP2_,0,0,rbx,rsi,rdi,r12,r13,r14,r15
        _EndProlog

; Проверка значения num_vec
        xor eax,eax
        test r9,r9
        jz Done                             ;переход, если num_vec равен нулю
        test r9,07h
        jnz Done                            ;переход, если num_vec % 8 != 0 истинно

; Загрузка указателей на векторные массивы и проверка правильности выравнивания
        mov r10,[rdx]                       ;r10 = a.X
        or rax,r10
        mov r11,[rdx+8]                     ;r11 = a.Y
        or rax,r11
        mov r12,[rdx+16]                    ;r12 = a.Z
        or rax,r12

        mov r13,[r8]                        ;r13 = b.X
        or rax,r13
        mov r14,[r8+8]                      ;r14 = b.Y
        or rax,r14
        mov r15,[r8+16]                     ;r15 = b.Z
        or rax,r15

        mov rbx,[rcx]                       ;rbx = c.X
        or rax,rbx
        mov rsi,[rcx+8]                     ;rsi = c.Y
        or rax,rsi
        mov rdi,[rcx+16]                    ;rdi = c.Z
        or rax,rdi

        and rax,3fh                         ;невыровненный массив компонентов?
        mov eax,0                           ;возвращаемый код ошибки (также i = 0)
        jnz Done

; Чтение следующего блока (8 векторов) из a и b
        align 16
@@:     vmovapd zmm0,zmmword ptr [r10+rax*8]    ;zmm0 = a.X
        vmovapd zmm1,zmmword ptr [r11+rax*8]    ;zmm1 = a.Y
        vmovapd zmm2,zmmword ptr [r12+rax*8]    ;zmm2 = a.Z
```

```
        vmovapd zmm3,zmmword ptr [r13+rax*8]    ;zmm3 = b.X
        vmovapd zmm4,zmmword ptr [r14+rax*8]    ;zmm4 = b.Y
        vmovapd zmm5,zmmword ptr [r15+rax*8]    ;zmm5 = b.Z

; Вычисление перекрестных произведений
        vmulpd zmm16,zmm1,zmm5
        vmulpd zmm17,zmm2,zmm4
        vsubpd zmm18,zmm16,zmm17                ;c.X = a.Y * b.Z - a.Z * b.Y

        vmulpd zmm19,zmm2,zmm3
        vmulpd zmm20,zmm0,zmm5
        vsubpd zmm21,zmm19,zmm20                ;c.Y = a.Z * b.X - a.X * b.Z

        vmulpd zmm22,zmm0,zmm4
        vmulpd zmm23,zmm1,zmm3
        vsubpd zmm24,zmm22,zmm23                ;c.Z = a.X * b.Y - a.Y * b.X

; Сохранение вычисленных перекрестных произведений
        vmovapd zmmword ptr [rbx+rax*8],zmm18   ;сохранение значений C.X
        vmovapd zmmword ptr [rsi+rax*8],zmm21   ;сохранение значений C.Y
        vmovapd zmmword ptr [rdi+rax*8],zmm24   ;сохранение значений C.Z

        add rax,8                               ;i += 8
        cmp rax,r9
        jb @B                                   ;повтор до завершения

Done:   vzeroupper
        _DeleteFrame rbx,rsi,rdi,r12,r13,r14,r15
        ret
Avx512VcpSoa_ endp
        end
```

Перекрестное произведение двух трехмерных векторов `a` и `b` является третьим вектором `c`, который перпендикулярен как `a`, так и `b`. Компоненты `x`, `y` и `z` вектора `c` можно вычислить с помощью следующих уравнений:

$$c_x = a_y b_z - a_z b_y;$$
$$c_y = a_z b_x - a_x b_z;$$
$$c_z = a_x b_y - a_y b_x.$$

Заголовочный файл C++ `Ch13_07.h`, показанный в листинге 13.7, включает определения структур `Vector` и `VectorSoA`. Структура `Vector` содержит три значения с плавающей запятой двойной точности – `X`, `Y` и `Z`, – которые представляют компоненты трехмерного вектора. Структура `VectorSoA` включает три указателя на массивы с плавающей запятой двойной точности. Каждый массив содержит значения для одного компонента вектора. В примере `Ch13_07` эти структуры используются для сравнения производительности двух различных алгоритмов вычисления векторного произведения. Первый алгоритм выполняет свои вычисления с использованием *массива структур* (array of structures, AOS), а второй алгоритм использует *структуру массивов* (structure of arrays, SOA).

Функция C++ `Avx512Vcp` начинает свое выполнение с выделения памяти для наборов векторных структур данных. Эта функция использует класс шаблона

C++ `unique_ptr<Vector>` для выделения памяти для трех AOS. Обратите внимание, что каждый объект `Vector` *не выравнивается явно* по 64-байтовой границе, так как это потребует значительного объема хранилища данных, которое никогда не используется. Каждый AOS `unique_ptr<Vector>` также демонстрирует использование этого типа конструкции данных во многих реальных программах. `Avx512Vcp` использует класс шаблона C++ `AlignedArray<double>` для выделения правильно выровненного пространства хранения для векторных SOA. После выделения структуры данных функция `InitVec` инициализирует оба набора векторов `a` и `b`, используя случайные значения. Затем она вызывает функции перекрестного произведения векторов ассемблера `Avx512VcpAos_` и `Avx512VcpSoa_`.

В верхней части файла ассемблера находится пользовательский сегмент констант с именем `ConstVals`, который содержит индексы для команд `vgatherqpd` и `vscatterqpd`, использующиеся в `Avx512VcpAos_`. Значения индекса в этом сегменте соответствуют порядку следования компонентов вектора `X`, `Y` и `Z` в массиве объектов вектора. Рисунок 13.4 более подробно иллюстрирует этот порядок. Обратите внимание, что индексы, определенные в `ConstVals`, позволяют командам `vgatherqpd` и `vscatterqpd` загружать и сохранять восемь векторных объектов.

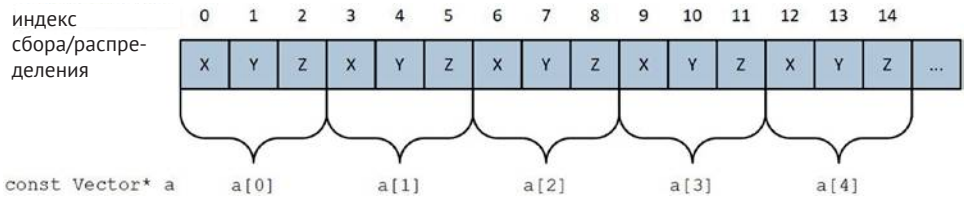


Рис. 13.4. Организация памяти компонентов `X`, `Y` и `Z` в массиве объектов `Vector`

После проверки `num_vec` три команды `vmovdqa64` (перемещение/присвоение выровненных упакованных значений в форме четверных слов) загружают индексы сбора/раздачи для компонентов `X`, `Y` и `X` вектора в регистры ZMM29, ZMM30 и ZMM31 соответственно. Цикл обработки начинается с команды `kxnorb k1,k1,k1`, которая устанавливает восемь младших битов регистра маски операции K1 в единицу. Далее команда `vgatherqpd zmm0 {k1},[rdx+zmm29*8]` загружает восемь значений компонентов `X` из набора `Vector a` в регистр ZMM0. Команда `vgatherqpd` загружает восемь значений, поскольку все младшие 8 бит регистра маски операции K1 установлены в единицу.

Еще пять наборов команд `kxnorb` и `vgatherqpd` загружают оставшиеся компоненты вектора в регистры ZMM1–ZMM5. Обратите внимание, что во время своего выполнения команда `vgatherqpd` устанавливает весь регистр маски операции в ноль, если только не возникает исключение из-за недопустимого доступа к памяти, которое может быть вызвано неправильным индексом или неправильным значением базового регистра. Это обновление регистра маски вводит потенциальную зависимость регистров, которая устраняется использованием разных регистров маски для каждой команды `vgatherqpd`. Следующий блок кода вычисляет восемь векторных перекрестных произведений, используя простую арифметику упакованных чисел с плавающей запятой двойной точности. Затем результаты перекрестного произведения сохраняются в целевом векторном массиве `c` с помощью трех команд `vscatterqpd`. Как и команда

vgatherqpd, команда vscatterqpd также устанавливает свой операнд регистра маски равным нулю, если только не возникает исключение.

Функция Avx512VcpSoa_ начинает свою работу с проверки num_vec. Затем он проверяет, что девять указателей массива компонентов вектора правильно выровнены по 64-байтовой границе. Цикл обработки в Avx512VcpSoa_ использует для вычисления векторных перекрестных произведений прямую арифметику упакованных чисел с плавающей запятой двойной точности. Обратите внимание, что Avx512VcpSoa_ использует команду vmovapd выровненного перемещения/присвоения для выполнения всех присвоений и сохранения компонентов вектора. Ниже приведен результат выполнения примера исходного кода Ch13_07:

```
Результаты Avx512VectorCrossProd

Векторное перекрестное произведение #0
  a:            96.0      30.0     52.0
  b:            64.0      62.0     79.0
  c_aos:      -854.0   -4256.0   4032.0
  c_soa:      -854.0 -  4256.0   4032.0
Векторное перекрестное произведение #1
  a:            26.0 3     3.0     66.0
  b:            89.0      36.0     20.0
  c_aos:     -1716.0    5354.0  -2001.0
  c_soa:     -1716.0    5354.0  -2001.0
Векторное перекрестное произведение #2
  a:            56.0      60.0     53.0
  b:            16.0      45.0     46.0
  c_aos:       375.0   -1728.0   1560.0
  c_soa:       375.0   -1728.0   1560.0
Векторное перекрестное произведение #3
  a:            79.0      27.0     22.0
  b:            18.0      75.0     45.0
  c_aos:      -435.0   -3159.0   5439.0
  c_soa:      -435.0   -3159.0   5439.0
Векторное перекрестное произведение #4
  a:            77.0      30.0     46.0
  b:            44.0      77.0     99.0
  c_aos:      -572.0   -5599.0   4609.0
  c_soa:      -572.0   -5599.0   4609.0
Векторное перекрестное произведение #5
  a:            30.0      21.0     26.0
  b:            43.0      61.0     47.0
  c_aos:      -599.0    -292.0    927.0
  c_soa:      -599.0    -292.0    927.0
Векторное перекрестное произведение #6
  a:            58.0      56.0     46.0
  b:            84.0      37.0     76.0
  c_aos:     2554.0    -544.0  -2558.0
  c_soa:     2554.0    -544.0  -2558.0
Векторное перекрестное произведение #7
  a:            34.0      28.0     95.0
  b:            20.0      51.0     36.0
  c_aos:     -3837.0     676.0   1174.0
  c_soa:     -3837.0     676.0   1174.0
```

```
Векторное перекрестное произведение #8
  a:           34.0      50.0     35.0
  b:           48.0       1.0     24.0
  c_aos:     1165.0     864.0  -2366.0
  c_soa:     1165.0     864.0  -2366.0
Векторное перекрестное произведение #9
  a:           28.0      12.0     46.0
  b:            6.0      53.0     77.0
  c_aos:    -1514.0   -1880.0   1412.0
  c_soa:    -1514.0   -1880.0   1412.0
Векторное перекрестное произведение #10
  a:           43.0      78.0     86.0
  b:           12.0      61.0     97.0
  c_aos:     2320.0   -3139.0   1687.0
  c_soa:     2320.0   -3139.0   1687.0
Векторное перекрестное произведение #11
  a:           53.0      78.0     85.0
  b:           78.0      34.0     65.0
  c_aos:     2180.0    3185.0  -4282.0
  c_soa:     2180.0    3185.0  -4282.0
Векторное перекрестное произведение #12
  a:            9.0      66.0      2.0
  b:           54.0      45.0     55.0
  c_aos:     3540.0    -387.0  -3159.0
  c_soa:     3540.0    -387.0  -3159.0
Векторное перекрестное произведение #13
  a:           15.0      59.0     35.0
  b:           94.0      67.0     22.0
  c_aos:    -1047.0    2960.0  -4541.0
  c_soa:    -1047.0    2960.0  -4541.0
Векторное перекрестное произведение #14
  a:           95.0      20.0     24.0
  b:           45.0      85.0     55.0
  c_aos:     -940.0   -4145.0   7175.0
  c_soa:     -940.0   -4145.0   7175.0
Векторное перекрестное произведение #15
  a:           76.0      77.0     15.0
  b:           29.0      95.0     23.0
  c_aos:      346.0   -1313.0   4987.0
  c_soa:      346.0   -1313.0   4987.0

Выполняется измерение быстродействия Avx512VectorCrossProd_BM - подождите
Результаты сохранены в файл Ch13_07_Avx512VectorCrossProd_BM_CHROMIUM.csv
```

В табл. 13.1 показаны результаты измерения быстродействия для двух функций вычисления перекрестных произведений.

В этой таблице дефисы используются для обозначения процессоров, не поддерживающих AVX-512. В примере исходного кода `Ch13_07` метод SOA несколько быстрее, чем метод AOS.

Таблица 13.1. Результаты измерения быстродействия для функций вычисления векторных перекрестных произведений (1 000 000 перекрестных произведений)

Процессор	Avx512VcpAos_	Avx512VcpSoa_
i7-4790S	-----	-----
i9-7900X	4734	4141
i7-8700K	-----	-----

13.2.5. Умножение матрицы на вектор

Многие алгоритмы компьютерной графики и обработки изображений выполняют умножение матрицы на вектор с использованием матриц 4×4 и векторов 4×1. В программном обеспечении трехмерной компьютерной графики эти типы вычислений повсеместно используются для выполнения аффинных преобразований (например, смещения, поворота и масштабирования) с использованием однородных координат. На рис. 13.5 показаны уравнения, которые можно использовать для умножения матрицы 4×4 на вектор 4×1. Обратите внимание, что компоненты вектора **b** представляют собой простое вычисление суммы произведений столбцов матрицы и отдельных компонентов вектора **a**. На рис. 13.5 также показан пример вычисления умножения матрицы на вектор с использованием действительных чисел.

$$\begin{bmatrix} b_w \\ b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} a_w \\ a_x \\ a_y \\ a_z \end{bmatrix} \qquad \begin{bmatrix} 304 \\ 564 \\ 824 \\ 1084 \end{bmatrix} = \begin{bmatrix} 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \\ 30 & 31 & 32 & 33 \\ 40 & 41 & 42 & 43 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix}$$

$$b_w = m_{00}a_w + m_{01}a_x + m_{02}a_y + m_{03}a_z \qquad 304 = 10(5) + 11(6) + 12(7) + 13(8)$$

$$b_x = m_{10}a_w + m_{11}a_x + m_{12}a_y + m_{13}a_z \qquad 564 = 20(5) + 21(6) + 22(7) + 23(8)$$

$$b_y = m_{20}a_w + m_{21}a_x + m_{22}a_y + m_{23}a_z \qquad 824 = 30(5) + 31(6) + 32(7) + 33(8)$$

$$b_z = m_{30}a_w + m_{31}a_x + m_{32}a_y + m_{33}a_z \qquad 1084 = 40(5) + 41(6) + 42(7) + 43(8)$$

col0 col1 col2 col3

Рис. 13.5. Уравнения для умножения матрицы на вектор и пример вычислений

В листинге 13.8 показан исходный код примера `Ch13_08`. В этом примере показано, как умножить одну матрицу 4×4 на набор векторов 4×1, которые хранятся в массиве.

Листинг 13.8. Пример Ch13_08

```
//-----------------------------------------------
//               Ch13_08.h
//-----------------------------------------------

#pragma once

// Простая векторная структура 4×1
struct Vec4x1_F32
{
    float W, X, Y, Z;
};

// Ch13_08.cpp
extern void InitVecArray(Vec4x1_F32* va, size_t num_vec);
extern bool Avx512MatVecMulF32Cpp(Vec4x1_F32* vec_b, float mat[4][4], Vec4x1_F32* vec_a,
size_t num_vec);

// Ch13_08_.asm
extern "C" bool Avx512MatVecMulF32_(Vec4x1_F32* vec_b, float mat[4][4], Vec4x1_F32* vec_a,
size_t num_vec);

// Ch13_08_BM.cpp
extern void Avx512MatVecMulF32_BM(void);

//-----------------------------------------------
//               Ch13_08.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <random>
#include <cmath>
#include "Ch13_08.h"
#include "AlignedMem.h"

using namespace std;

bool VecCompare(const Vec4x1_F32* v1, const Vec4x1_F32* v2)
{
    static const float eps = 1.0e-12f;

    bool b0 = (fabs(v1->W - v2->W) <= eps);
    bool b1 = (fabs(v1->X - v2->X) <= eps);
    bool b2 = (fabs(v1->Y - v2->Y) <= eps);
    bool b3 = (fabs(v1->Z - v2->Z) <= eps);

    return b0 && b1 && b2 && b3;
}

void InitVecArray(Vec4x1_F32* va, size_t num_vec)
{
    uniform_int_distribution<> ui_dist {1, 500};
    default_random_engine rng {187};
```

```
    for (size_t i = 0; i < num_vec; i++)
    {
        va[i].W = (float)ui_dist(rng);
        va[i].X = (float)ui_dist(rng);
        va[i].Y = (float)ui_dist(rng);
        va[i].Z = (float)ui_dist(rng);
    }

    if (num_vec >= 4)
    {
        // Тестовые значения
        va[0].W =  5; va[0].X =  6; va[0].Y =  7; va[0].Z =  8;
        va[1].W = 15; va[1].X = 16; va[1].Y = 17; va[1].Z = 18;
        va[2].W = 25; va[2].X = 26; va[2].Y = 27; va[2].Z = 28;
        va[3].W = 35; va[3].X = 36; va[3].Y = 37; va[3].Z = 38;
    }
}

bool Avx512MatVecMulF32Cpp(Vec4x1_F32* vec_b, float mat[4][4], Vec4x1_F32* vec_a, size_t
num_vec)
{
    if (num_vec == 0 || num_vec % 4 != 0)
        return false;

    if (!AlignedMem::IsAligned(vec_a, 64) || !AlignedMem::IsAligned(vec_b, 64))
        return false;

    if (!AlignedMem::IsAligned(mat, 64))
        return false;

    for (size_t i = 0; i < num_vec; i++)
    {
        vec_b[i].W =  mat[0][0] * vec_a[i].W + mat[0][1] * vec_a[i].X;
        vec_b[i].W += mat[0][2] * vec_a[i].Y + mat[0][3] * vec_a[i].Z;

        vec_b[i].X =  mat[1][0] * vec_a[i].W + mat[1][1] * vec_a[i].X;
        vec_b[i].X += mat[1][2] * vec_a[i].Y + mat[1][3] * vec_a[i].Z;

        vec_b[i].Y =  mat[2][0] * vec_a[i].W + mat[2][1] * vec_a[i].X;
        vec_b[i].Y += mat[2][2] * vec_a[i].Y + mat[2][3] * vec_a[i].Z;

        vec_b[i].Z =  mat[3][0] * vec_a[i].W + mat[3][1] * vec_a[i].X;
        vec_b[i].Z += mat[3][2] * vec_a[i].Y + mat[3][3] * vec_a[i].Z;
    }

    return true;
}

void Avx512MatVecMulF32(void)
{
    const size_t num_vec = 8;

    alignas(64) float mat[4][4]
    {
       10.0, 11.0, 12.0, 13.0,
       20.0, 21.0, 22.0, 23.0,
```

```
        30.0, 31.0, 32.0, 33.0,
        40.0, 41.0, 42.0, 43.0
    };

    AlignedArray<Vec4x1_F32> vec_a_aa(num_vec, 64);
    AlignedArray<Vec4x1_F32> vec_b1_aa(num_vec, 64);
    AlignedArray<Vec4x1_F32> vec_b2_aa(num_vec, 64);

    Vec4x1_F32* vec_a = vec_a_aa.Data();
    Vec4x1_F32* vec_b1 = vec_b1_aa.Data();
    Vec4x1_F32* vec_b2 = vec_b2_aa.Data();

    InitVecArray(vec_a, num_vec);

    bool rc1 = Avx512MatVecMulF32Cpp(vec_b1, mat, vec_a, num_vec);
    bool rc2 = Avx512MatVecMulF32_(vec_b2, mat, vec_a, num_vec);

    cout << "Результаты Avx512MatVecMulF32\n";

    if (!rc1 || !rc2)
    {
        cout << "Ошибочный возвращаемый код\n";
        cout << "  rc1 = " << boolalpha << rc1 << '\n';
        cout << "  rc2 = " << boolalpha << rc2 << '\n';
        return;
    }

    const unsigned int w = 8;
    cout << fixed << setprecision(1);

    for (size_t i = 0; i < num_vec; i++)
    {
        cout << "Пример #" << i << '\n';

        cout << "vec_b1: ";
        cout << "  " << setw(w) << vec_b1[i].W << ' ';
        cout << "  " << setw(w) << vec_b1[i].X << ' ';
        cout << "  " << setw(w) << vec_b1[i].Y << ' ';
        cout << "  " << setw(w) << vec_b1[i].Z << '\n';

        cout << "vec_b2: ";
        cout << "  " << setw(w) << vec_b2[i].W << ' ';
        cout << "  " << setw(w) << vec_b2[i].X << ' ';
        cout << "  " << setw(w) << vec_b2[i].Y << ' ';
        cout << "  " << setw(w) << vec_b2[i].Z << '\n';

        if (!VecCompare(&vec_b1[i], &vec_b2[i]))
        {
            cout << "Ошибка – несовпадение векторов\n";
            return;
        }
    }
}

int main()
{
```

```
    Avx512MatVecMulF32();
    Avx512MatVecMulF32_BM();
    return 0;
}

;-----------------------------------------------
;               Ch13_07.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>

; Индексы для команд сбора/раздачи
ConstVals   segment readonly align(64) 'const'
GS_X        qword 0, 3, 6,  9, 12, 15, 18, 21
GS_Y        qword 1, 4, 7, 10, 13, 16, 19, 22
GS_Z        qword 2, 5, 8, 11, 14, 17, 20, 23
ConstVals   ends

; extern "C" bool Avx512VcpAos_(Vector* c, const Vector* a, const Vector* b, size_t num_
vectors);

        .code
Avx512VcpAos_ proc

; Проверка num_vec
        xor eax,eax                         ;возвращаемый код ошибки (also i = 0)
        test r9,r9
        jz Done                             ;переход, если num_vec равен нулю
        test r9,07h
        jnz Done                            ;переход, если num_vec % 8 != 0 истинно

; Чтение индексов для операций сбора/раздачи
        vmovdqa64 zmm29,zmmword ptr [GS_X]  ;zmm29 = индексы компонента X
        vmovdqa64 zmm30,zmmword ptr [GS_Y]  ;zmm30 = индексы компонента Y
        vmovdqa64 zmm31,zmmword ptr [GS_Z]  ;zmm31 = индексы компонента Z

; Чтение следующих 8 векторов
        align 16
@@:     kxnorb k1,k1,k1
        vgatherqpd zmm0{k1},[rdx+zmm29*8]       ;zmm0 = значения A.X

        kxnorb k2,k2,k2
        vgatherqpd zmm1{k2},[rdx+zmm30*8]       ;zmm1 = значения A.Y

        kxnorb k3,k3,k3
        vgatherqpd zmm2{k3},[rdx+zmm31*8]       ;zmm2 = значения A.Z

        kxnorb k4,k4,k4
        vgatherqpd zmm3{k4},[r8+zmm29*8]        ;zmm3 = значения B.X

        kxnorb k5,k5,k5
        vgatherqpd zmm4{k5},[r8+zmm30*8]        ;zmm4 = значения B.Y

        kxnorb k6,k6,k6
        vgatherqpd zmm5{k6},[r8+zmm31*8]        ;zmm5 = значения B.Z
```

```
; Вычисление 8 векторных перекрестных произведений
        vmulpd zmm16,zmm1,zmm5
        vmulpd zmm17,zmm2,zmm4
        vsubpd zmm18,zmm16,zmm17                ;c.X = a.Y * b.Z - a.Z * b.Y

        vmulpd zmm19,zmm2,zmm3
        vmulpd zmm20,zmm0,zmm5
        vsubpd zmm21,zmm19,zmm20                ;c.Y = a.Z * b.X - a.X * b.Z

        vmulpd zmm22,zmm0,zmm4
        vmulpd zmm23,zmm1,zmm3
        vsubpd zmm24,zmm22,zmm23                ;c.Z = a.X * b.Y - a.Y * b.X

; Сохранение вычисленных перекрестных произведений
        kxnorb k4,k4,k4
        vscatterqpd [rcx+zmm29*8]{k4},zmm18     ;сохранение компонентов C.X

        kxnorb k5,k5,k5
        vscatterqpd [rcx+zmm30*8]{k5},zmm21     ;сохранение компонентов C.Y

        kxnorb k6,k6,k6
        vscatterqpd [rcx+zmm31*8]{k6},zmm24     ;сохранение компонентов C.Z

; Обновление указателей и счетчиков
        add rcx,192                             ;c += 8
        add rdx,192                             ;a += 8
        add r8,192                              ;b += 8
        add rax,8                               ;i += 8
        cmp rax,r9
        jb @B

        mov eax,1                               ;возвращаемый код успешного выполнения

Done:   vzeroupper
        ret
Avx512VcpAos_ endp

; extern "C" bool Avx512VcpSoa_(VectorSoA* c, const VectorSoA* a, const VectorSoA* b,
size_t num_vectors);

Avx512VcpSoa_ proc frame
        _CreateFrame CP2_,0,0,rbx,rsi,rdi,r12,r13,r14,r15
        _EndProlog

; Проверка num_vec
        xor eax,eax
        test r9,r9
        jz Done                                 ;переход, если num_vec равен нулю
        test r9,07h
        jnz Done                                ;переход, если num_vec % 8 != 0 истинно

; Чтение указателей векторного массива и проверка правильности выравнивания
        mov r10,[rdx]                           ;r10 = a.X
        or rax,r10
        mov r11,[rdx+8]                         ;r11 = a.Y
        or rax,r11
```

```
        mov r12,[rdx+16]                    ;r12 = a.Z
        or rax,r12

        mov r13,[r8]                        ;r13 = b.X
        or rax,r13
        mov r14,[r8+8]                      ;r14 = b.Y
        or rax,r14
        mov r15,[r8+16]                     ;r15 = b.Z
        or rax,r15

        mov rbx,[rcx]                       ;rbx = c.X
        or rax,rbx
        mov rsi,[rcx+8]                     ;rsi = c.Y
        or rax,rsi
        mov rdi,[rcx+16]                    ;rdi = c.Z
        or rax,rdi

        and rax,3fh                         ;невыровненный массив компонентов?
        mov eax,0                           ;возвращаемый код ошибки (также i = 0)
        jnz Done

; Чтение следующего блока (8 векторов) из a и b
        align 16
@@:     vmovapd zmm0,zmmword ptr [r10+rax*8]    ;zmm0 = значения a.X
        vmovapd zmm1,zmmword ptr [r11+rax*8]    ;zmm1 = значения a.Y
        vmovapd zmm2,zmmword ptr [r12+rax*8]    ;zmm2 = значения a.Z
        vmovapd zmm3,zmmword ptr [r13+rax*8]    ;zmm3 = значения b.X
        vmovapd zmm4,zmmword ptr [r14+rax*8]    ;zmm4 = значения b.Y
        vmovapd zmm5,zmmword ptr [r15+rax*8]    ;zmm5 = значения b.Z

; Вычисление перекрестных произведений
        vmulpd zmm16,zmm1,zmm5
        vmulpd zmm17,zmm2,zmm4
        vsubpd zmm18,zmm16,zmm17                ;c.X = a.Y * b.Z - a.Z * b.Y

        vmulpd zmm19,zmm2,zmm3
        vmulpd zmm20,zmm0,zmm5
        vsubpd zmm21,zmm19,zmm20                ;c.Y = a.Z * b.X - a.X * b.Z

        vmulpd zmm22,zmm0,zmm4
        vmulpd zmm23,zmm1,zmm3
        vsubpd zmm24,zmm22,zmm23                ;c.Z = a.X * b.Y - a.Y * b.X

; Сохранение вычисленных перекрестных произведений
        vmovapd zmmword ptr [rbx+rax*8],zmm18   ;сохранение значений C.X
        vmovapd zmmword ptr [rsi+rax*8],zmm21   ;сохранение значений C.Y
        vmovapd zmmword ptr [rdi+rax*8],zmm24   ;сохранение значений C.Z

        add rax,8                               ;i += 8
        cmp rax,r9
        jb @B                                   ;повторение до завершения

Done:   vzeroupper
        _DeleteFrame rbx,rsi,rdi,r12,r13,r14,r15
        ret
Avx512VcpSoa_ endp
        end
```

Код C++ в листинге 13.8 начинается с заголовочного файла `Ch13_08.h`, который содержит необходимые объявления функций. Этот файл также содержит объявление структуры `Vec4x1_F32`, включающей четыре компонента вектора-столбца 4×1. Файл исходного кода `Ch13_08.cpp` включает функцию `Avx512MatVecMulF32Cpp`. Эта функция реализует уравнения умножения матрицы на вектор, показанные на рис. 13.5. Оставшийся код C++ в листинге 13.8 выполняет инициализацию тестового примера, вызывает функции вычисления и отображает результаты.

Код на языке ассемблера в листинге 13.8 начинается с постоянного сегмента данных, который определяет серию индексов упакованных перестановок. Реализация алгоритма умножения матрицы на вектор на языке ассемблера использует эти значения для изменения порядка элементов исходной матрицы и векторов. Причина этого переупорядочения состоит в том, чтобы облегчить одновременное вычисление четырех произведений матрица–вектор. Функция `Avx512MatVecMulF32_` начинает свою работу с проверки делимости `num_vec` на четыре. Затем она проверяет указатели матричного и векторного буферов на предмет правильного выравнивания по 64-байтовой границе.

После проверки аргумента четыре команды `vmovdqa32` загружают индексы перестановки матриц в регистры ZMM16–ZMM19. За этим следует еще одна серия из четырех команд `vmovdqa32`, которые загружают индексы перестановки векторов в регистры ZMM24–ZMM27. Затем команда `vmovaps zmm0,zmmword ptr [rdx]` загружает все 16 элементов матрицы `mat` с плавающей запятой одинарной точности в ZMM0. Команда `vpermps zmm20, zmm16,zmm0` (перестановка элементов с плавающей запятой одинарной точности) переупорядочивает элементы в ZMM0 в соответствии с индексами в ZMM16. Эта команда загружает четыре копии столбца `0` из матрицы `mat` в регистр ZMM20. Затем еще три команды `vpermps` используются для выполнения той же операции с использованием столбцов 1, 2 и 3. Выполнение этих перестановок наглядно показано на рис. 13.6.

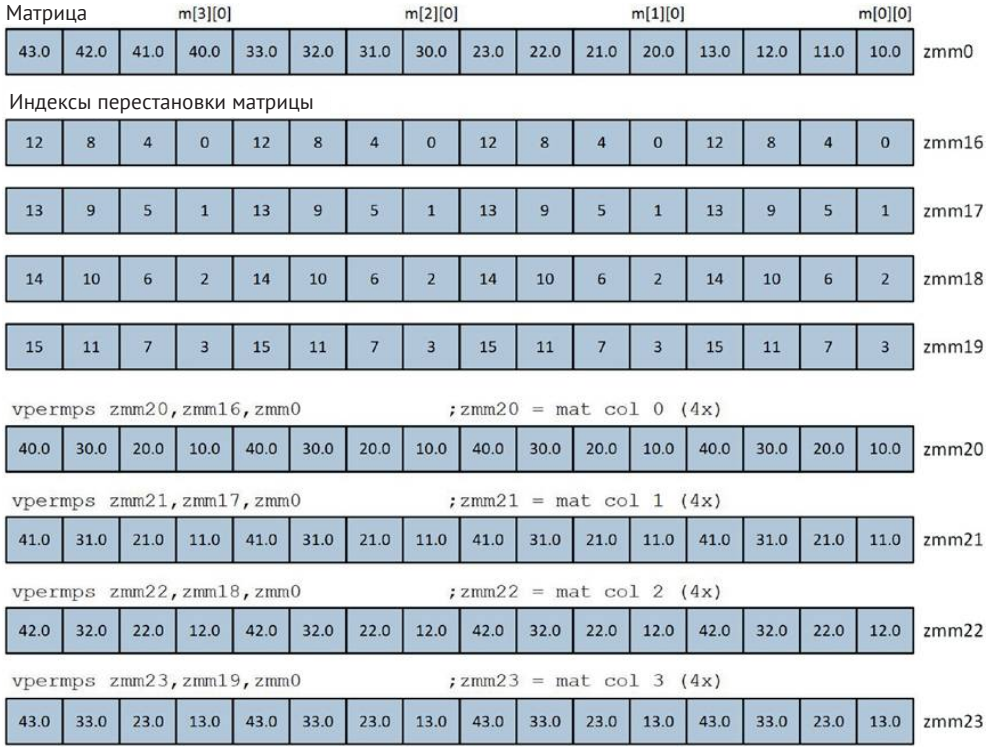


Рис. 13.6. Перестановка столбцов матрицы с помощью команд `vpermps`

Цикл обработки в `Avx512MatVecMulF32_` начинается с команды `vmovaps zmm4,zmmword ptr [r8+rax]`, которая загружает четыре вектора `Vec4x1_F32` в регистр ZMM4. Компоненты `W`, `X`, `Y` и `Z` этих векторов затем перегруппировываются с помощью другой серии команд `vpermps`. После выполнения этих команд регистры ZMM0–ZMM3 содержат повторяющиеся наборы компонентов вектора, как показано на рис. 13.7.

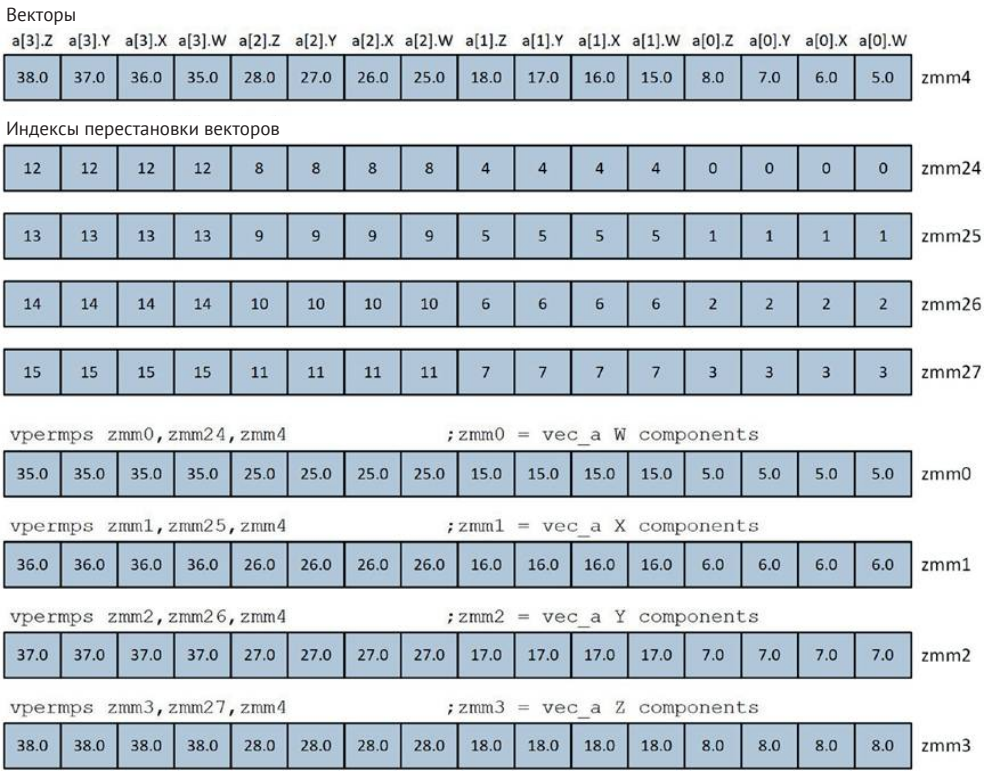


Рис. 13.7. Перестановка компонентов вектора с помощью команд `vpermps`

После перестановок компонентов вектора последовательность команд `vmulps` и `vaddps` выполняет четыре одновременных умножения матрицы на вектор. Эта процедура схематически показана на рис. 13.8. Следующая команда `vmovaps zmmword ptr [rcx+rax],zmm4` сохраняет четыре результирующих вектора 4×1 в массиве `vec_b`. Затем цикл обработки повторяется до тех пор, пока не будут обработаны все векторы в `vec_a`.

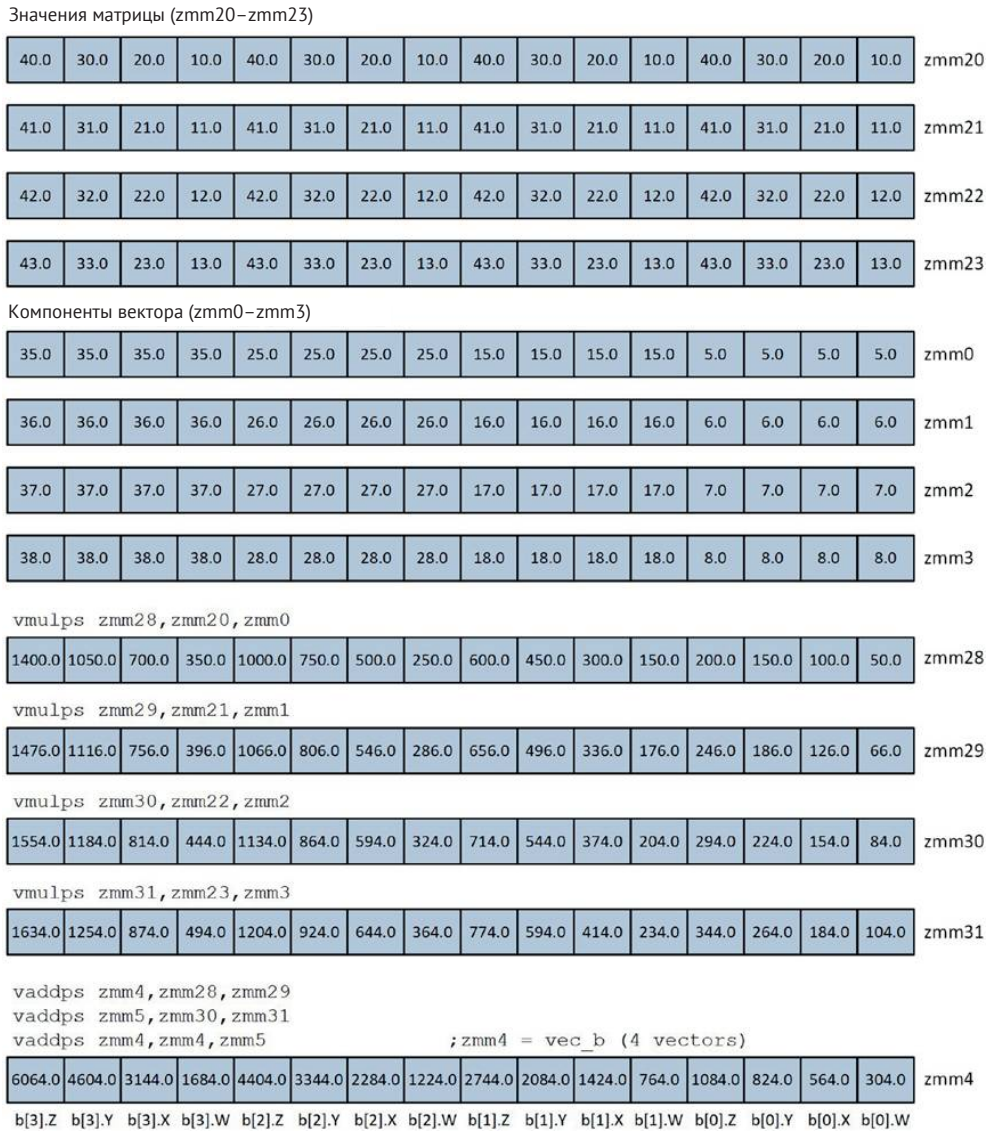


Рис. 13.8. Умножение матрицы на вектор с использованием `vmulps` и `vaddps`

Ниже показаны результаты выполнения примера исходного кода `Ch13_08`. В табл. 13.2 приведены результаты измерения времени выполнения функций умножения матрицы на вектор на языке C++ и на языке ассемблера.

```
Результаты Avx512MatVecMulF32
Пример #0
vec_b1:      304.0      564.0      824.0     1084.0
vec_b2:      304.0      564.0      824.0     1084.0
```

```
Пример #1
vec_b1:      764.0     1424.0     2084.0     2744.0
vec_b2:      764.0     1424.0     2084.0     2744.0
Пример #2
vec_b1:     1224.0     2284.0     3344.0     4404.0
vec_b2:     1224.0     2284.0     3344.0     4404.0
Пример #3
vec_b1:     1684.0     3144.0     4604.0     6064.0
vec_b2:     1684.0     3144.0     4604.0     6064.0
Пример #4
vec_b1:    11932.0    22452.0    32972.0    43492.0
vec_b2:    11932.0    22452.0    32972.0    43492.0
Пример #5
vec_b1:    17125.0    31705.0    46285.0    60865.0
vec_b2:    17125.0    31705.0    46285.0    60865.0
Пример #6
vec_b1:    12723.0    23873.0    35023.0    46173.0
vec_b2:    12723.0    23873.0    35023.0    46173.0
Пример #7
vec_b1:    15121.0    27871.0    40621.0    53371.0
vec_b2:    15121.0    27871.0    40621.0    53371.0

Выполняется измерение быстродействия Avx512MatVecMulF32_BM - подождите
Результаты сохранены в файл Ch13_08_Avx512MatVecMulF32_BM_CHROMIUM.csv
```

Таблица 13.2. Результаты измерения времени выполнения функций умножения матрицы на вектор (1 000 000 векторов)

Процессор	Avx512MatVecMulF32Cpp	Avx512MatVecMulF32_
i7-4790S	-----	-----
i9-7900X	6174	1778
i7-8700K	-----	-----

13.2.6. Свертки

В листинге 13.9 показан исходный код примера `Ch13_09`. Этот пример представляет собой реализацию программы свертки AVX-512, которая была представлена в примере исходного кода `Ch11_02`. Основная цель данного примера – подчеркнуть разницу между функциями, использующими команды AVX2, и функциями, использующими команды AVX-512. Данный пример также дает возможность сравнить время выполнения функций свертки на базе команд AVX2 и AVX-512.

Листинг 13.9. Пример Ch13_09

```
;-----------------------------------------------
;              Ch13_09_.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>
        extern c_NumPtsMin:dword
        extern c_NumPtsMax:dword
        extern c_KernelSizeMin:dword
        extern c_KernelSizeMax:dword
```

```
; extern bool Avx512Convolve2_(float* y, const float* x, int num_pts, const float* kernel,
int kernel_size)

        .code
Avx512Convolve2_ proc frame
        _CreateFrame CV2_,0,0,rbx
        _EndProlog

; Проверка аргументов
        xor eax,eax                         ;возвращаемый код ошибки

        mov r10d,dword ptr [rbp+CV2_OffsetStackArgs]
        test r10d,1
        jz Done                             ;размер ядра четный
        cmp r10d,[c_KernelSizeMin]
        jl Done                             ;размер ядра слишком мал
        cmp r10d,[c_KernelSizeMax]
        jg Done                             ;размер ядра слишком велик

        cmp r8d,[c_NumPtsMin]
        jl Done                             ;num_pts слишком мал
        cmp r8d,[c_NumPtsMax]
        jg Done                             ;num_pts слишком велик
        test r8d,15
        jnz Done                            ;num_pts не кратен 16

        test rcx,3fh
        jnz Done                            ;y не выровнен правильно

; Инициализация переменных цикла свертки
        shr r10d,1                          ;r10 = kernel_size / 2 (ks2)
        lea rdx,[rdx+r10*4]                 ;rdx = x + ks2 (первая выборка данных)
        xor ebx,ebx                         ;i = 0

; Процедура свертки
LP1:    vxorps zmm0,zmm0,zmm0               ;упакованная сумма = 0.0;
        mov r11,r10                         ;r11 = ks2
        neg r11                             ;k = -ks2

LP2:    mov rax,rbx                             ;rax = i
        sub rax,r11                             ;rax = i - k
        vmovups zmm1,zmmword ptr [rdx+rax*4]    ;чтение x[i - k]:x[i - k + 15]

        mov rax,r11
        add rax,r10                             ;rax = k + ks2
        vbroadcastss zmm2,real4 ptr [r9+rax*4]  ;zmm2 = kernel[k + ks2]
        vfmadd231ps zmm0,zmm1,zmm2              ;zmm0 += x[i-k]:x[i-k+15] * kernel[k+ks2]

        add r11,1                               ;k += 1
        cmp r11,r10
        jle LP2                                 ;повторять, пока не станет k > ks2

        vmovaps zmmword ptr [rcx+rbx*4],zmm0    ;сохранение y[i]:y[i + 15]

        add rbx,16                             ;i += 16
        cmp rbx,r8
```

```
        jl LP1                          ;повтор до завершения
        mov eax,1                       ;возвращаемый код успешного выполнения

Done:   vzeroupper
        _DeleteFrame rbx
        ret
Avx512Convolve2_ endp

; extern bool Avx512Convolve2Ks5_(float* y, const float* x, int num_pts, const float*
kernel, int kernel_size)

Avx512Convolve2Ks5_ proc frame
        _CreateFrame CKS5_,0,48
        _SaveXmmRegs xmm6,xmm7,xmm8
        _EndProlog

; Проверка аргументов
        xor eax,eax                     ;возвращаемый код ошибки

        cmp dword ptr [rbp+CKS5_OffsetStackArgs],5
        jne Done                        ;переход, если kernel_size не равен 5

        cmp r8d,[c_NumPtsMin]
        jl Done                         ;переход, если num_pts слишком мал
        cmp r8d,[c_NumPtsMax]
        jg Done                         ;переход, если num_pts слишком велик
        test r8d,15
        jnz Done                        ;num_pts не кратен 15

        test rcx,3fh
        jnz Done                        ;y не выровнен правильно

; Начальная инициализация
        vbroadcastss zmm4,real4 ptr [r9]        ;kernel[0]
        vbroadcastss zmm5,real4 ptr [r9+4]      ;kernel[1]
        vbroadcastss zmm6,real4 ptr [r9+8]      ;kernel[2]
        vbroadcastss zmm7,real4 ptr [r9+12]     ;kernel[3]
        vbroadcastss zmm8,real4 ptr [r9+16]     ;kernel[4]

        mov r8d,r8d                     ;r8 = num_pts
        add rdx,8                       ;x += 2

; Процедура свертки
@@:     vxorps zmm2,zmm2,zmm2           ;инициализация переменных суммы
        vxorps zmm3,zmm3,zmm3

        mov r11,rax
        add r11,2                       ;j = i + ks2

        vmovups zmm0,zmmword ptr [rdx+r11*4]    ;zmm0 = x[j]:x[j + 15]
        vfmadd231ps zmm2,zmm0,zmm4              ;zmm2 += x[j]:x[j + 15] * kernel[0]

        vmovups zmm1,zmmword ptr [rdx+r11*4-4]  ;zmm1 = x[j - 1]:x[j + 14]
        vfmadd231ps zmm3,zmm1,zmm5              ;zmm3 += x[j - 1]:x[j + 14] * kernel[1]

        vmovups zmm0,zmmword ptr [rdx+r11*4-8]  ;zmm0 = x[j - 2]:x[j + 13]
```

```
        vfmadd231ps zmm2,zmm0,zmm6              ;zmm2 += x[j - 2]:x[j + 13] * kernel[2]

        vmovups zmm1,zmmword ptr [rdx+r11*4-12] ;zmm1 = x[j - 3]:x[j + 12]
        vfmadd231ps zmm3,zmm1,zmm7              ;zmm3 += x[j - 3]:x[j + 12] * kernel[3]

        vmovups zmm0,zmmword ptr [rdx+r11*4-16] ;zmm0 = x[j - 4]:x[j + 11]
        vfmadd231ps zmm2,zmm0,zmm8              ;zmm2 += x[j - 4]:x[j + 11] * kernel[4]

        vaddps zmm0,zmm2,zmm3                   ;финальные значения
        vmovaps zmmword ptr [rcx+rax*4],zmm0    ;save y[i]:y[i + 15]

        add rax,16                               ;i += 16
        cmp rax,r8
        jl @B                                   ;переход, если i < num_pts
        mov eax,1                               ;возвращаемый код успешного выполнения

Done:   vzeroupper
        _RestoreXmmRegs xmm6,xmm7,xmm8
        _DeleteFrame
        ret
Avx512Convolve2Ks5_ endp
        end
```

Часть исходного кода на языке C++ примера `Ch13_09` не показана в листинге 13.9, поскольку она почти идентична коду C++ в примере `Ch11_02`. Модификации, внесенные в код C++ `Ch13_09`, включают лишь несколько изменений имен функций. Тестовые массивы также располагаются по 64-байтовой границе вместо 32-байтовой.

Функция на языке ассемблера `Avx512Convolve2_` реализует алгоритм свертки с ядром переменного размера, описанный в главе 11. Основное различие между этой функцией и ее аналогом в AVX2 `Convolve2_` (см. листинг 11.2) заключается в использовании регистров ZMM вместо регистров YMM. Код, который управляет счетчиком индекса в регистре RBX, также был изменен, чтобы отразить обработку 16 точек данных на итерацию вместо 8. Аналогичные изменения были также внесены в функцию свертки с ядром фиксированного размера `Avx512Convolve2Ks5_`.

Вывод результатов выполнения примера исходного кода `Ch13_09` не показан, поскольку он такой же, как вывод для примера исходного кода `Ch11_02`. В табл. 13.3 показаны результаты измерения времени выполнения для функций `Avx512Convolve2_` и `Avx512Convolve2Ks5_`. Эта таблица также содержит результаты измерений для функций AVX2 `Convolve2_` и `ConvolveKs2_` из табл. 11.2. Реализации AVX-512 работают быстрее, чем их аналоги AVX2, особенно в случае независимой от размера функции свертки `Avx512Convolve2_`. Конечно, было бы неосмотрительно экстраполировать какие-либо общие выводы относительно производительности AVX-512 по сравнению с AVX2, основываясь исключительно на измерениях времени, показанных в табл. 13.3. Другие примеры вы увидите в главе 14.

Таблица 13.3. Среднее время выполнения (мс) для функций свертки AVX2 и AVX-512 с использованием ядра свертки с пятью элементами (2 000 000 выборок сигнала)

Процессор	Convolve2_	Avx512Convolve2_	Convolve2Ks5_	Avx512Convolve2Ks5_
i7-4790S	1244	-----	1067	-----
i9-7900X	956	757	719	693
i7-8700K	859	-----	595	-----

13.3. Заключение

В главе 13 рассмотрены следующие ключевые моменты:

- при использовании маскирования слиянием со скалярными или упакованными операндами процессор выполняет вычисление команды только в том случае, если соответствующий бит регистра маски установлен в единицу. В противном случае вычисление не выполняется, и элемент операнда-адресата остается неизменным;
- функции на языке ассемблера AVX-512 могут использовать адрес регистра маски в качестве операнда с большинством команд, выполняющих скалярные или упакованные операции сравнения. Затем биты регистра маски могут использоваться для принятия управляемых данными логических решений без каких-либо команд условного перехода с использованием маскирования слиянием или нулем и (при необходимости) простых логических операций;
- функции на языке ассемблера AVX-512 должны использовать команды `vmovdqu[32|64]` и `vmovdqa[32|64]` для выполнения операций перемещения/присвоения с использованием 512-битных упакованных двойных слов и целочисленных операндов в форме четверного слова. Эти команды также можно использовать с операндами шириной 256 и 128 бит;
- в отличие от AVX и AVX2, набор AVX-512 включает команды, которые выполняют преобразование между операндами с плавающей запятой и целыми числами без знака;
- функции AVX-512 должны гарантировать, что упакованные операнды шириной 128, 256 и 512 бит выровнены по правильной границе всегда, когда это возможно;
- функции на языке ассемблера, использующие команды AVX-512 с регистрами ZMM0–ZMM15 или YMM0–YMM15, всегда должны применять команду `vzeroupper` до того, как управление программой будет передано обратно вызывающей функции;
- функции и алгоритмы на языке ассемблера, которые используют структуру массивов, часто работают быстрее, чем те, которые используют массив структур;
- соглашение о вызовах Visual C++ рассматривает регистры AVX-512 ZMM16–ZMM31, YMM16–YMM31 и XMM16–XMM31 как изменчивые вне границ функций. Это означает, что функция может использовать упомянутые регистры без необходимости сохранять их значения.

Глава 14

Программирование AVX-512 – упакованные целые числа

В главах 7 и 10 вы узнали, как использовать наборы команд AVX и AVX2 для выполнения операций упакованной целочисленной арифметики с использованием 128-битных и 256-битных операндов. В этой главе вы узнаете, как использовать набор команд AVX-512 для выполнения аналогичных операций с использованием операндов шириной 512 бит. Вы также узнаете, как использовать команды AVX-512 с упакованными целочисленными операндами шириной 256 и 128 бит. Первый пример исходного кода объясняет, как выполнять базовые арифметические действия с упакованными целыми числами с использованием регистров ZMM. Далее следует несколько примеров, иллюстрирующих алгоритмы и методы обработки изображений с использованием команд AVX-512. Как и в предыдущей главе, для всех примеров исходного кода в этой главе требуется процессор и операционная система, поддерживающие AVX-512 и следующие расширения набора команд: AVX512F, AVX512CD, AVX512BW, AVX512DQ и AVX512VL. Вы можете использовать одну из свободно доступных утилит, перечисленных в приложении, чтобы определить, поддерживает ли ваша система эти расширения.

14.1. Базовая арифметика

В листинге 14.1 показан исходный код примера `Ch14_01`. В этом примере показано, как выполнять базовую арифметику с упакованными целыми числами с использованием операндов шириной 512 бит и набора регистров ZMM.

Листинг 14.1. Пример Ch14_01

```
//-----------------------------------------------
//              Ch14_01.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#include "Zmmval.h"

using namespace std;
```

```
extern "C" void Avx512PackedMathI16_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[6]);
extern "C" void Avx512PackedMathI64_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[5],
uint32_t opmask);

void Avx512PackedMathI16(void)
{
    alignas(64) ZmmVal a;
    alignas(64) ZmmVal b;
    alignas(64) ZmmVal c[6];

    a.m_I16[0] = 10;       b.m_I16[0] = 100;
    a.m_I16[1] = 20;       b.m_I16[1] = 200;
    a.m_I16[2] = 30;       b.m_I16[2] = 300;
    a.m_I16[3] = 40;       b.m_I16[3] = 400;
    a.m_I16[4] = 50;       b.m_I16[4] = 500;
    a.m_I16[5] = 60;       b.m_I16[5] = 600;
    a.m_I16[6] = 70;       b.m_I16[6] = 700;
    a.m_I16[7] = 80;       b.m_I16[7] = 800;

    a.m_I16[8] = 1000;     b.m_I16[8] = -100;
    a.m_I16[9] = 2000;     b.m_I16[9] = 200;
    a.m_I16[10] = 3000;    b.m_I16[10] = -300;
    a.m_I16[11] = 4000;    b.m_I16[11] = 400;
    a.m_I16[12] = 5000;    b.m_I16[12] = -500;
    a.m_I16[13] = 6000;    b.m_I16[13] = 600;
    a.m_I16[14] = 7000;    b.m_I16[14] = -700;
    a.m_I16[15] = 8000;    b.m_I16[15] = 800;

    a.m_I16[16] = -1000;   b.m_I16[16] = 100;
    a.m_I16[17] = -2000;   b.m_I16[17] = -200;
    a.m_I16[18] = 3000;    b.m_I16[18] = 303;
    a.m_I16[19] = 4000;    b.m_I16[19] = -400;
    a.m_I16[20] = -5000;   b.m_I16[20] = 500;
    a.m_I16[21] = -6000;   b.m_I16[21] = -600;
    a.m_I16[22] = -7000;   b.m_I16[22] = 700;
    a.m_I16[23] = -8000;   b.m_I16[23] = 800;

    a.m_I16[24] = 30000;   b.m_I16[24] = 3000;      // переполнение сложения
    a.m_I16[25] = 6000;    b.m_I16[25] = 32000;     // переполнение сложения
    a.m_I16[26] = -25000;  b.m_I16[26] = -27000;    // переполнение сложения
    a.m_I16[27] = 8000;    b.m_I16[27] = 28700;     // переполнение сложения
    a.m_I16[28] = 2000;    b.m_I16[28] = -31000;    // переполнение вычитания
    a.m_I16[29] = 4000;    b.m_I16[29] = -30000;    // переполнение вычитания
    a.m_I16[30] = -3000;   b.m_I16[30] = 32000;     // переполнение вычитания
    a.m_I16[31] = -15000;  b.m_I16[31] = 24000;     // переполнение вычитания

    Avx512PackedMathI16_(&a, &b, c);

    cout <<"\nРезультаты Avx512PackedMathI16\n\n";
    cout << " i        a        b   vpaddw  vpaddsw   vpsubw  vpsubsw  vpminsw  vpmaxsw\n";
    cout << "-------------------------------------------------------------------------\n";

    for (int i = 0; i < 32; i++)
    {
        cout << setw(2)  << i << ' ';
        cout << setw(8) << a.m_I16[i] << ' ';
```

```
        cout << setw(8) << b.m_I16[i] << ' ';
        cout << setw(8) << c[0].m_I16[i] << ' ';
        cout << setw(8) << c[1].m_I16[i] << ' ';
        cout << setw(8) << c[2].m_I16[i] << ' ';
        cout << setw(8) << c[3].m_I16[i] << ' ';
        cout << setw(8) << c[4].m_I16[i] << ' ';
        cout << setw(8) << c[5].m_I16[i] << '\n';
    }
}

void Avx512PackedMathI64(void)
{
    alignas(64) ZmmVal a;
    alignas(64) ZmmVal b;
    alignas(64) ZmmVal c[6];
    uint32_t opmask = 0x7f;

    a.m_I64[0] = 64;        b.m_I64[0] = 4;
    a.m_I64[1] = 1024;      b.m_I64[1] = 5;
    a.m_I64[2] = -2048;     b.m_I64[2] = 2;
    a.m_I64[3] = 8192;      b.m_I64[3] = 5;
    a.m_I64[4] = -256;      b.m_I64[4] = 8;
    a.m_I64[5] = 4096;      b.m_I64[5] = 7;
    a.m_I64[6] = 16;        b.m_I64[6] = 3;
    a.m_I64[7] = 512;       b.m_I64[7] = 6;

    Avx512PackedMathI64_(&a, &b, c, opmask);

    cout << "\nРезультаты Avx512PackedMathI64\n\n";
    cout << "op_mask = " << hex << opmask << dec << '\n';
    cout << " i      a      b   vpaddq   vpsubq  vpmullq  vpsllvq  vpsravq   vpabsq\n";
    cout << "-------------------------------------------------------------------------\n";

    for (int i = 0; i < 8; i++)
    {
        cout << setw(2) << i << ' ';
        cout << setw(6) << a.m_I64[i] << ' ';
        cout << setw(6) << b.m_I64[i] << ' ';
        cout << setw(8) << c[0].m_I64[i] << ' ';
        cout << setw(8) << c[1].m_I64[i] << ' ';
        cout << setw(8) << c[2].m_I64[i] << ' ';
        cout << setw(8) << c[3].m_I64[i] << ' ';
        cout << setw(8) << c[4].m_I64[i] << ' ';
        cout << setw(8) << c[5].m_I64[i] << '\n';
    }
}

int main()
{
    Avx512PackedMathI16();
    Avx512PackedMathI64();
    return 0;
}
```

```
;-----------------------------------------------
;              Ch14_01.asm
;-----------------------------------------------

; extern "C" void Avx512PackedMathI16_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[6])

        .code
Avx512PackedMathI16_ proc
        vmovdqu16 zmm0,zmmword ptr [rcx]        ;zmm0 = a
        vmovdqu16 zmm1,zmmword ptr [rdx]        ;zmm1 = b

; Операции с упакованным словом
        vpaddw zmm2,zmm0,zmm1                   ;сложение
        vmovdqa64 zmmword ptr [r8],zmm2         ;сохранение результата vpaddw

        vpaddsw zmm2,zmm0,zmm1                  ;сложение с насыщением со знаком
        vmovdqa64 zmmword ptr [r8+64],zmm2      ;сохранение результата vpaddsw

        vpsubw zmm2,zmm0,zmm1                   ;вычитание
        vmovdqa64 zmmword ptr [r8+128],zmm2     ;сохранение результата vpsubw

        vpsubsw zmm2,zmm0,zmm1                  ;вычитание с насыщением со знаком
        vmovdqa64 zmmword ptr [r8+192],zmm2     ;сохранение результата vpsubsw

        vpminsw zmm2,zmm0,zmm1                  ;минимумы со знаком
        vmovdqa64 zmmword ptr [r8+256],zmm2     ;сохранение результата vpminsw

        vpmaxsw zmm2,zmm0,zmm1                  ;максимумы со знаком
        vmovdqa64 zmmword ptr [r8+320],zmm2     ;сохранение результата vpmaxsw

        vzeroupper
        ret
Avx512PackedMathI16_ endp

; extern "C" void Avx512PackedMathI64_(const ZmmVal* a, const ZmmVal* b, ZmmVal c[5],
unsigned int opmask)

Avx512PackedMathI64_ proc
        vmovdqa64 zmm0,zmmword ptr [rcx]        ;zmm0 = a
        vmovdqa64 zmm1,zmmword ptr [rdx]        ;zmm1 = b

        and r9d,0ffh                            ;r9d = значение маски
        kmovb k1,r9d                            ;k1 = маска

; Операции с упакованным четверным словом
        vpaddq zmm2{k1}{z},zmm0,zmm1            ;сложение
        vmovdqa64 zmmword ptr [r8],zmm2         ;сохранение результата vpaddq

        vpsubq zmm2{k1}{z},zmm0,zmm1            ;вычитание
        vmovdqa64 zmmword ptr [r8+64],zmm2      ;сохранение результата vpsubq

        vpmullq zmm2{k1}{z},zmm0,zmm1           ;умножение со знаком (мл. 64 бита)
        vmovdqa64 zmmword ptr [r8+128],zmm2     ;сохранение результата vpmullq

        vpsllvq zmm2{k1}{z},zmm0,zmm1           ;логический сдвиг влево
        vmovdqa64 zmmword ptr [r8+192],zmm2     ;сохранение результата vpsllvq
```

```
        vpsravq zmm2{k1}{z},zmm0,zmm1           ;арифметический сдвиг вправо
        vmovdqa64 zmmword ptr [r8+256],zmm2     ;сохранение результата vpsravq

        vpabsq zmm2{k1}{z},zmm0                 ;абсолютное значение
        vmovdqa64 zmmword ptr [r8+320],zmm2     ;сохранение результата vpabsq

        vzeroupper
        ret
Avx512PackedMathI64_ endp
        end
```

Функции на языке C++ `Avx512PackedMathI16` и `Avx512PackedMathI64` выполняют операции AVX-512 над упакованными целыми числами с использованием значений в формате слова и четверного слова. Каждая функция начинает свое выполнение с инициализации соответствующих целочисленных элементов двух переменных `ZmmVal`. Обратите внимание, что спецификатор C++ `alignas(64)` используется с каждым `ZmmVal`. После инициализации переменной каждая базовая процедура вызывает соответствующую функцию языка ассемблера: `Avx512PackedMathI16_` или `Avx512PackedMathI64_`. Затем результаты передаются в `cout`.

Функция на языке ассемблера `Avx512PackedMathI16_` начинает свое выполнение с двух команд `vmovdqa64`, которые загружают переменные `ZmmVal a` и `b` в регистры ZMM0 и ZMM1 соответственно. Несколько удивительно, что AVX512BW не содержит выровненных команд перемещения/присвоения для 512-битных упакованных байтовых и однословных операндов. Альтернативой здесь было бы использование команды `vmovdqu16`. Обратите внимание, что эта последняя команда должна использоваться в случаях, когда требуется маскирование слиянием или нулем. AVX512BW также содержит команду `vmovdqu8` для 512-битных упакованных байтовых операндов. После загрузки значения операнда `Avx512PackedMathI16_` демонстрирует команды, выполняющие операции с упакованными словами: `vpaddw`, `vpaddsw`, `vpsubw`, `vpsubsw`, `vpminsw`, `vpmaxsw`. Каждый результат в виде 512-битного упакованного слова сохраняется в массиве `c`. Обратите внимание, что `Avx512PackedMathI16_` содержит команду `vzeroupper` перед своей командой `ret`.

Функция на языке ассемблера `Avx512PackedMathI64_` иллюстрирует различные арифметические операции с использованием упакованных четверных слов шириной 512 бит. `Avx512PackedMathI64_` также использует команду `vzeroupper` перед своей командой `ret`. Ниже показаны результаты выполнения примера исходного кода `Ch14_01`.

```
Результаты Avx512PackedMathI16

      i        a        b      vpaddw     vpaddsw     vpsubw     vpsubsw     vpminsw     vpmaxsw
      0       10      100         110          no        -90         -90          10         100
      1       20      200         220         220       -180        -180          20         200
      2       30      300         330         330       -270        -270          30         300
      3       40      400         440         440       -360        -360          40         400
      4       50      500         550         550       -450        -450          50         500
```

5	60	600	660	660	-540	-540	60	600
6	70	700	770	770	-630	-630	70	700
7	80	800	880	880	-720	-720	80	800
8	1000	-100	900	900	1100	1100	-100	1000
9	2000	200	2200	2200	1800	1800	200	2000
10	3000	-300	2700	2700	3300	3300	-300	3000
11	4000	400	4400	4400	3600	3600	400	4000
12	5000	-500	4500	4500	5500	5500	-500	5000
13	6000	600	6600	6600	5400	5400	600	6000
14	7000	-700	6300	6300	7700	7700	-700	7000
15	8000	800	8800	8800	7200	7200	800	8000
16	-1000	100	-900	-900	-1100	-1100	-1000	100
17	-2000	-200	-2200	-2200	-1800	-1800	-2000	-200
18	3000	303	3303	3303	2697	2697	303	3000
19	4000	-400	3600	3600	4400	4400	-400	4000
20	-5000	500	-4500	-4500	-5500	-5500	-5000	500
21	-6000	-600	-6600	-6600	-5400	-5400	-6000	-600
22	-7000	700	-6300	-6300	-7700	-7700	-7000	700
23	-8000	800	-7200	-7200	-8800	-8800	-8000	800
24	30000	3000	-32536	32767	27000	27000	3000	30000
25	6000	32000	-27536	32767	-26000	-26000	6000	32000
26	-25000	-27000	13536	-32768	2000	2000	-27000	-25000
27	8000	28700	-28836	32767	-20700	-20700	8000	28700
28	2000	-31000	-29000	-29000	-32536	32767	-31000	2000
29	4000	-30000	-26000	-26000	-31536	32767	-30000	4000
30	-3000	32000	29000	29000	30536	-32768	-3000	32000
31	-15000	24000	9000	9000	26536	-32768	-15000	24000

Результаты Avx512PackedMathI64

op_mask = 7f

i	a	b	vpaddq	vpsubq	vpmullq	vpsllvq	vpsravq	vpabsq
0	64	4	68	60	256	1024	4	64
1	1024	5	1029	1019	5120	32768	32	1024
2	-2048	2	-2046	-2050	-4096	-8192	-512	2048
3	8192	5	8197	8187	40960	262144	256	8192
4	-256	8	-248	-264	-2048	-65536	-1	256
5	4096	7	4103	4089	28672	524288	32	4096
6	16	3	19	13	48	128	2	16
7	512	6	0	0	0	0	0	0

14.2. Обработка изображений

Примеры исходного кода в этом разделе иллюстрируют алгоритмы и методы обработки изображений с использованием команд AVX-512 для упакованных целочисленных операндов. Большинство примеров исходного кода – это обновленные версии примеров из предыдущих глав, в которых применялись команды AVX или AVX2. Помимо демонстрации использования команд AVX-512, последующие примеры исходного кода также подчеркивают альтернативные алгоритмические подходы и последовательности команд, которые часто приводят к повышению производительности.

14.2.1. Пиксельные преобразования

В главе 7 вы узнали, как использовать набор команд AVX для преобразования 8-битных пикселей без знака в пиксели с плавающей запятой одинарной точности и наоборот (см. пример `Ch07_06`). Пример исходного кода `Ch14_02` демонстрирует, как можно выполнить те же преобразования, используя команды AVX-512. В листинге 14.2 показан исходный код примера `Ch14_02`.

Листинг 14.2. Пример Ch14_02

```
//-----------------------------------------------
//               Ch14_02.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cstdint>
#include <random>
#include "AlignedMem.h"

using namespace std;

// Ch14_02_Misc.cpp
extern bool Avx512ConvertImgU8ToF32Cpp(float* des, const uint8_t* src, uint32_t num_
pixels);
extern bool Avx512ConvertImgF32ToU8Cpp(uint8_t* des, const float* src, uint32_t num_
pixels);
extern uint32_t Avx512ConvertImgVerify(const float* src1, const float* src2, uint32_t
num_pixels);
extern uint32_t Avx512ConvertImgVerify(const uint8_t* src1, const uint8_t* src2, uint32_t
num_pixels);

// Ch14_02_.asm
extern "C" bool Avx512ConvertImgU8ToF32_(float* des, const uint8_t* src, uint32_t num_
pixels);
extern "C" bool Avx512ConvertImgF32ToU8_(uint8_t* des, const float* src, uint32_t num_
pixels);

void InitU8(uint8_t* x, uint32_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {0, 255};
    default_random_engine rng {seed};
```

```
    for (uint32_t i = 0; i < n; i++)
        x[i] = ui_dist(rng);
}

void InitF32(float* x, uint32_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {0, 1000};
    default_random_engine rng {seed};

    for (uint32_t i = 0; i < n; i++)
        x[i] = (float)ui_dist(rng) / 1000.0f;
}

void Avx512ConvertImgU8ToF32(void)
{
    const size_t align = 64;
    const uint32_t num_pixels = 1024;
    AlignedArray<uint8_t> src_aa(num_pixels, align);
    AlignedArray<float> des1_aa(num_pixels, align);
    AlignedArray<float> des2_aa(num_pixels, align);
    uint8_t* src = src_aa.Data();
    float* des1 = des1_aa.Data();
    float* des2 = des2_aa.Data();

    InitU8(src, num_pixels, 12);

    bool rc1 = Avx512ConvertImgU8ToF32Cpp(des1, src, num_pixels);
    bool rc2 = Avx512ConvertImgU8ToF32_(des2, src, num_pixels);

    cout << "\nРезультаты Avx512ConvertImgU8ToF32\n";

    if (!rc1 || !rc2)
    {
        cout << "Ошибочный возвращаемый код - ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        return;
    }

    uint32_t num_diff = Avx512ConvertImgVerify(des1, des2, num_pixels);
    cout << "  Количество ошибок сравнения пикселей (num_diff) = " << num_diff << '\n';
}

void Avx512ConvertImgF32ToU8(void)
{
    const size_t align = 64;
    const uint32_t num_pixels = 1024;
    AlignedArray<float> src_aa(num_pixels, align);
    AlignedArray<uint8_t> des1_aa(num_pixels, align);
    AlignedArray<uint8_t> des2_aa(num_pixels, align);
    float* src = src_aa.Data();
    uint8_t* des1 = des1_aa.Data();
    uint8_t* des2 = des2_aa.Data();

    InitF32(src, num_pixels, 20);
```

```
    // Тестовые значения для демонстрации обрезки в функциях преобразования
    src[0] = 0.5f;          src[8]  = 3.33f;
    src[1] = -1.0f;         src[9]  = 0.67f;
    src[2] = 0.38f;         src[10] = 0.75f;
    src[3] = 0.62f;         src[11] = 0.95f;
    src[4] = 2.1f;          src[12] = -0.33f;
    src[5] = 0.25f;         src[13] = 0.8f;
    src[6] = -1.25f;        src[14] = 0.12f;
    src[7] = 0.45f;         src[15] = 4.0f;

    bool rc1 = Avx512ConvertImgF32ToU8Cpp(des1, src, num_pixels);
    bool rc2 = Avx512ConvertImgF32ToU8_(des2, src, num_pixels);

    cout << "\nРезультаты Avx512ConvertImgF32ToU8\n";

    if (!rc1 || !rc2)
    {
        cout << "Ошибочный возвращаемый код - ";
        cout << "rc1 = " << boolalpha << rc1 << ", ";
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        return;
    }

    uint32_t num_diff = Avx512ConvertImgVerify(des1, des2, num_pixels);
    cout << "  Количество ошибок сравнения пикселей (num_diff) = " << num_diff << '\n';
}

int main()
{
    Avx512ConvertImgU8ToF32();
    Avx512ConvertImgF32ToU8();
    return 0;
}

;-----------------------------------------------
;               Ch14_02.asm
;-----------------------------------------------

        include <cmpequ.asmh>
        extern c_NumPixelsMax:dword

            .const
r4_1p0      real4 1.0
r4_255p0    real4 255.0

; extern "C" bool Avx512ConvertImgU8ToF32_(float* des, const uint8_t* src, uint32_t num_
pixels)

        .code
Avx512ConvertImgU8ToF32_ proc

; Проверка правильности значения num_pixels и выравнивания пиксельных буферов
        xor eax,eax                         ;возвращаемый код ошибки
        or r8d,r8d
        jz Done                             ;переход, если num_pixels равен нулю
        cmp r8d,[c_NumPixelsMax]
```

```
        ja Done                         ;переход, если num_pixels слишком велик
        test r8d,3fh
        jnz Done                        ;переход, если num_pixels % 64 != 0
        test rcx,3fh
        jnz Done                        ;переход, если приемник не выровнен
        test rdx,3fh
        jnz Done                        ;переход, если источник не выровнен

; Начальная инициализация
        shr r8d,6                       ;количество блоков (64 пикселя/блок)
        vmovss xmm0,real4 ptr [r4_1p0]
        vdivss xmm1,xmm0,real4 ptr [r4_255p0]
        vbroadcastss zmm5,xmm1          ;упакованный коэфф. масштабирования (1.0 /
255.0)

        align 16
@@:     vpmovzxbd zmm0,xmmword ptr [rdx]
        vpmovzxbd zmm1,xmmword ptr [rdx+16]
        vpmovzxbd zmm2,xmmword ptr [rdx+32]
        vpmovzxbd zmm3,xmmword ptr [rdx+48] ;zmm3:zmm0 = 64 пикселей U32

; Преобразование пикселей из uint8_t в число с плав. запятой [0.0, 255.0]
        vcvtudq2ps zmm16,zmm0
        vcvtudq2ps zmm17,zmm1
        vcvtudq2ps zmm18,zmm2
        vcvtudq2ps zmm19,zmm3           ;zmm19:zmm16 = 64 пикселей F32

; Нормализация пикселей к диапазону [0.0, 1.0]
        vmulps zmm20,zmm16,zmm5
        vmulps zmm21,zmm17,zmm5
        vmulps zmm22,zmm18,zmm5
        vmulps zmm23,zmm19,zmm5         ;zmm23:zmm20 = 64 пикселей F32
(нормализованных)

; Сохранение пикселей F32 в приемник
        vmovaps zmmword ptr [rcx],zmm20
        vmovaps zmmword ptr [rcx+64],zmm21
        vmovaps zmmword ptr [rcx+128],zmm22
        vmovaps zmmword ptr [rcx+192],zmm23

; Обновление указателей и счетчиков
        add rdx,64
        add rcx,256
        sub r8d,1
        jnz @B

        mov eax,1                   ;возвращаемый код успешного выполнения

Done:   vzeroupper
        ret
Avx512ConvertImgU8ToF32_ endp

; extern "C" bool Avx512ConvertImgF32ToU8_(uint8_t* des, const float* src, uint32_t num_
pixels)

Avx512ConvertImgF32ToU8_ proc
```

```
; Проверка правильности значения num_pixels и выравнивания пиксельных буферов
        xor eax,eax                         ;возвращаемый код ошибки
        or r8d,r8d
        jz Done                             ;переход, если num_pixels равен нулю
        cmp r8d,[c_NumPixelsMax]
        ja Done                             ;переход, если num_pixels слишком велик
        test r8d,3fh
        jnz Done                            ;переход, если num_pixels % 64 != 0
        test rcx,3fh
        jnz Done                            ;переход, если приемник не выровнен
        test rdx,3fh
        jnz Done                            ;переход, если источник не выровнен

; Начальная инициализация
        shr r8d,4                           ;number of pixel blocks (16 pixels / block)
        vxorps zmm29,zmm29,zmm29            ;packed 0.0
        vbroadcastss zmm30,[r4_1p0]         ;packed 1.0
        vbroadcastss zmm31,[r4_255p0]       ;packed 255.0

        align 16
@@:     vmovaps zmm0,zmmword ptr [rdx]      ;zmm0 = block of 16 pixels

; Усечение пикселей текущего блока к диапазону [0,0. 1.0]
        vcmpps k1,zmm0,zmm29,CMP_GE         ;k1 = пиксельная маска >= 0.0
        vmovaps zmm1{k1}{z},zmm0            ;все пиксели >= 0.0

        vcmpps k2,zmm1,zmm30,CMP_GT         ;k2 = пиксельная маска > 1.0
        vmovaps zmm1{k2},zmm30              ;все пиксели усечены до [0.0, 1.0]

; Преобразование пикселей в uint8_t и сохранение в приемник
        vmulps zmm2,zmm1,zmm31              ;все пиксели [0.0, 255.0]
        vcvtps2udq zmm3,zmm2{ru-sae}        ;все пиксели [0, 255]
        vpmovusdb xmmword ptr [rcx],zmm3    ;сохранение пикселей как байтов без знака

; Обновление указателей и счетчиков
        add rdx,64
        add rcx,16
        sub r8d,1
        jnz @B

        mov eax,1                           ;возвращаемый код успешного выполнения

Done:   vzeroupper
        ret
Avx512ConvertImgF32ToU8_ endp
        end
```

Код C++ в листинге 14.2 начинается с объявления необходимых функций. Первый набор объявлений предназначен для функций `Avx512ConvertImgU8ToF32Cpp` и `Avx512ConvertImgU8ToF32Cpp`, которые определены в файле `Ch14_02_Misc.cpp`. Исходный код этих функций не показан, поскольку они почти идентичны функциям-аналогам AVX2, которые использовались в примере исходного кода `Ch07_06`. Были внесены два незначительных изменения: пиксельные буферы источника и назначения выровнены по 64-байтовой границе вместо 16-байто-

вой границы; количество пикселей в этих буферах должно делиться без остатка на 64 вместо 32.

Функция Avx512ConvertImgU8ToF32 инициализирует тестовые массивы для преобразования значений пикселей из uint8_t в float. Эта функция использует класс C++ AlignedArray<> для размещения этих массивов на границе 64 байта. После инициализации тестового массива Avx512ConvertImgU8ToF32 вызывает функции преобразования на языках C++ и ассемблера. Затем она вызывает Avx512ConvertImgVerify для проверки результатов. Функция Avx512ConvertImgF32ToU8 преобразует значения пикселей из числа с плавающей точкой в uint8_t. Обратите внимание, что эта функция намеренно инициализирует первые несколько значений исходного пиксельного буфера src известными значениями, чтобы убедиться, что функции преобразования правильно отсекают значения пикселей, выходящие за пределы допустимого диапазона.

Функция языка ассемблера Avx512ConvertImgU8ToF32_ начинает свое выполнение с проверки аргумента num_pixels. Затем она проверяет, что пиксельные буферы src и des правильно выровнены по 64-байтовой границе. В примере исходного кода Ch07_06 из главы 7 нормализация пикселей была выполнена путем деления каждого значения пикселя на 255,0. Функция Avx512ConvertImgU8ToF32_ выполняет нормализацию пикселей с использованием мультипликативного коэффициента масштабирования 1,0/255,0, поскольку умножение с плавающей запятой обычно выполняется быстрее, чем деление с плавающей запятой. Команда vbroadcastss zmm5,xmm1 загружает упакованную версию этого коэффициента масштабирования в регистр ZMM5.

Каждая итерация цикла обработки начинается с команды vpmovzxbd zmm0,xmmword ptr [rdx]. Эта команда копирует и расширяет нулями 16-байтовые (или uint8_t) пиксели, на которые указывает RDX, до двойных слов; затем она сохраняет эти значения в регистре ZMM0. Далее используются еще три команды vpmovzxbd для загрузки еще 48 пикселей в регистры ZMM1, ZMM2 и ZMM3. За ними следуют четыре команды vcvtudq2ps, преобразующие каждое значение пикселя в формате двойного слова без знака в регистрах ZMM0–ZMM3 в числа с плавающей запятой одинарной точности. Последующие команды vmulps умножают эти значения на масштабирующий коэффициент нормализации; затем результаты сохраняются в приемном пиксельном буфере при помощи серии команд vmovaps.

В примере исходного кода Ch07_06 все значения пикселей с плавающей запятой были обрезаны до [0.0, 1.0] перед преобразованием в значения uint8_t. Функция Avx512ConvertImgF32ToU8_ выполняет ту же операцию. После проверки правильности аргументов Avx512ConvertImgF32ToU8_ загружает регистры ZMM29, ZMM30 и ZMM31 с упакованными версиями констант с плавающей запятой одинарной точности 0.0, 1.0 и 255.0 соответственно. Рабочий цикл функции Avx512ConvertImgF32ToU8_ начинает каждую итерацию с команды vmovaps zmm0,zmmword ptr [rdx], которая загружает блок из 16 пикселей с плавающей запятой одинарной точности в регистр ZMM0. Последующая команда vcmpps k1,zmm0,zmm29,CMP_GE сравнивает каждый элемент пикселя в ZMM0 с 0,0 и сохраняет результирующую маску сравнения в регистре маски K1. Затем команда vmovaps zmm1 {k1}{z},zmm0 использует маскировку нулем для удаления всех значений пикселей меньше 0,0. Эти операции наглядно показаны на рис. 14.1.

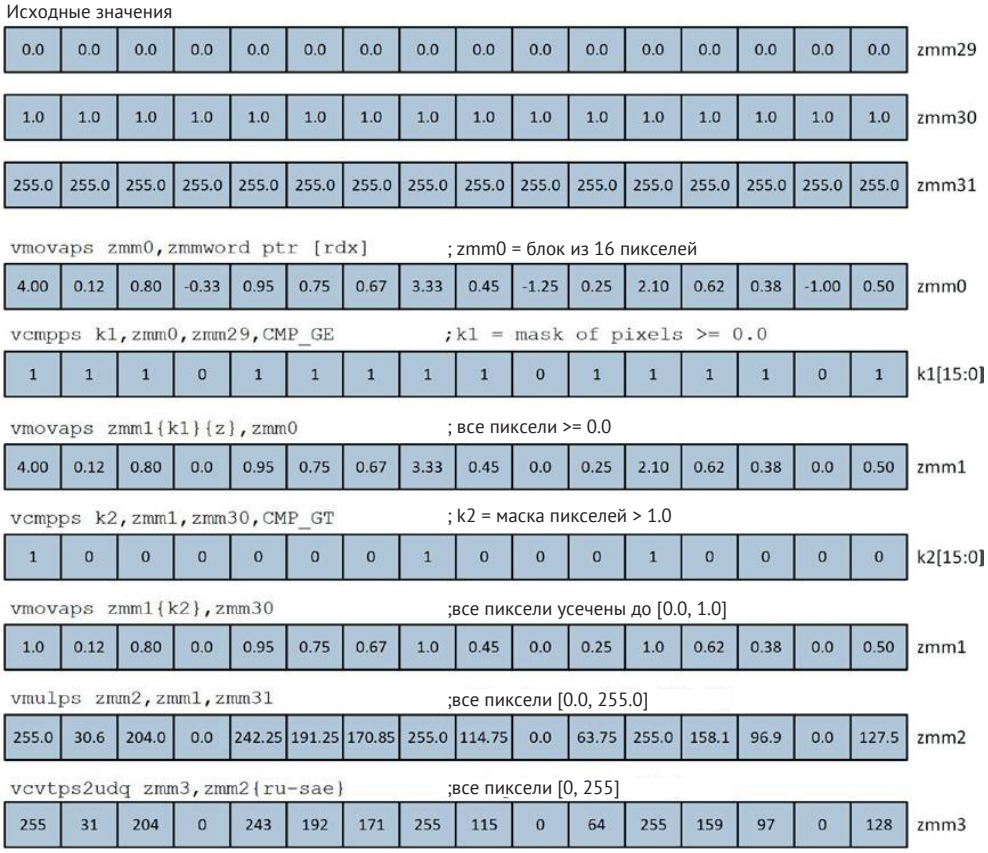


Рис. 14.1. Последовательность команд для преобразования упакованных значений пикселей из чисел с плавающей запятой в целые числа в формате двойного слова без знака

Последующая команда `vcmpps k2,zmm1,zmm30,CMP_GT` создает маску значений пикселей больше 1.0 и сохраняет эту маску в регистре маски K2. После выполнения команды `vmovaps zmm1 {k2},zmm30` все значения пикселей в регистре ZMM1 больше или равны 0,0 и меньше или равны 1,0. Следующие две команды, `vmulps zmm2,zmm1,zmm31` и `vcvtps2udq zmm3,zmm2{ru-sae}`, преобразуют нормализованные значения пикселей с плавающей запятой в целые числа в формате двойного слова без знака. Обратите внимание, что команда `vcvtps2udq` использует операнд управления округлением на уровне команды (округление в большую сторону) в первую очередь для демонстрационных целей. Затем команда `vpmovusdb xmmword ptr [rcx],zmm3` уменьшает двойные слова пиксельных значений до байтов с использованием беззнакового насыщения и сохраняет их в буфере-приемнике, на который указывает RCX. Результаты выполнения примера исходного кода Ch14_02 выглядят следующим образом:

```
Результаты Avx512ConvertImgU8ToF32
  Количество ошибок сравнения пикселей (num_diff) = 0

Результаты Avx512ConvertImgF32ToU8
  Количество ошибок сравнения пикселей (num_diff) = 0
```

14.2.2. Пороговая обработка изображений

В примере исходного кода Ch07_08 вы узнали о пороговой обработке изображений и о том, как создать двоичное (или двухцветное) изображение маски. Напомню, что *пороговая обработка* – это метод обработки изображения, который устанавливает для пикселя в изображении маски значение 0xff, если значение интенсивности соответствующего пикселя в изображении в градациях серого больше, чем предварительно определенное пороговое значение; в противном случае пиксель изображения маски устанавливается в 0x00. Пример исходного кода Ch14_03 расширяет технику определения порога изображения, которая использовалась в Ch07_08, добавляя поддержку нескольких операторов сравнения. В листинге 14.3 показан исходный код примера Ch14_03.

Листинг 14.3. Пример Ch14_03

```
//-----------------------------------------------
//               Ch14_03.h
//-----------------------------------------------

#pragma once
#include <cstdint>

// Операторы сравнения
enum CmpOp { EQ, NE, LT, LE, GT, GE };

// Ch14_03_Misc.cpp
extern void Init(uint8_t* x, size_t n, unsigned int seed);
extern void ShowResults(const uint8_t* des1, const uint8_t* des2, size_t num_pixels, CmpOp
cmp_op,
    uint8_t cmp_val, size_t test_id);

// Ch14_03_.asm
extern "C" bool Avx512ComparePixels_(uint8_t* des, const uint8_t* src, size_t num_pixels,
    CmpOp cmp_op, uint8_t cmp_val);

//-----------------------------------------------
//               Ch14_03.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <cassert>
#include "Ch14_03.h"
#include "AlignedMem.h"

using namespace std;

extern "C" const size_t c_NumPixelsMax = 16777216;
```

```
bool Avx512ComparePixelsCpp(uint8_t* des, const uint8_t* src, size_t num_pixels, CmpOp
cmp_op, uint8_t cmp_val)
{
    // Проверка допустимости num_pixels
    if ((num_pixels == 0) || (num_pixels > c_NumPixelsMax))
        return false;
    if ((num_pixels & 0x3f) != 0)
        return false;

    // Проверка выравнивания буферов src и des по 64-битной границе
    if (!AlignedMem::IsAligned(src, 64))
        return false;
    if (!AlignedMem::IsAligned(des, 64))
        return false;

    bool rc = true;
    const uint8_t cmp_false = 0x00;
    const uint8_t cmp_true = 0xff;

    switch (cmp_op)
    {
        case CmpOp::EQ:
            for (size_t i = 0; i < num_pixels; i++)
                des[i] = (src[i] == cmp_val) ? cmp_true : cmp_false;
            break;

        case CmpOp::NE:
            for (size_t i = 0; i < num_pixels; i++)
                des[i] = (src[i] != cmp_val) ? cmp_true : cmp_false;
            break;

        case CmpOp::LT:
            for (size_t i = 0; i < num_pixels; i++)
                des[i] = (src[i] < cmp_val) ? cmp_true : cmp_false;
            break;

        case CmpOp::LE:
            for (size_t i = 0; i < num_pixels; i++)
                des[i] = (src[i] <= cmp_val) ? cmp_true : cmp_false;
            break;

        case CmpOp::GT:
            for (size_t i = 0; i < num_pixels; i++)
                des[i] = (src[i] > cmp_val) ? cmp_true : cmp_false;
            break;

        case CmpOp::GE:
            for (size_t i = 0; i < num_pixels; i++)
                des[i] = (src[i] >= cmp_val) ? cmp_true : cmp_false;
            break;

        default:
            cout << "Неправильный CmpOp: " << cmp_op << '\n';
            rc = false;
    }

    return rc;
}
```

```
int main()
{
    const size_t align = 64;
    const size_t num_pixels = 4 * 1024 * 1024;
    AlignedArray<uint8_t> src_aa(num_pixels, align);
    AlignedArray<uint8_t> des1_aa(num_pixels, align);
    AlignedArray<uint8_t> des2_aa(num_pixels, align);
    uint8_t* src = src_aa.Data();
    uint8_t* des1 = des1_aa.Data();
    uint8_t* des2 = des2_aa.Data();

    const uint8_t cmp_vals[] {197, 222, 43, 43, 129, 222};
    const CmpOp cmp_ops[] {CmpOp::EQ, CmpOp::NE, CmpOp::LT, CmpOp::LE, CmpOp::GT,
CmpOp::GE};
    const size_t num_cmp_vals = sizeof(cmp_vals) / sizeof(uint8_t);
    const size_t num_cmp_ops = sizeof(cmp_ops) / sizeof(CmpOp);

    assert(num_cmp_vals == num_cmp_ops);

    Init(src, num_pixels, 511);

    cout << "Результаты выполнения Ch14_03\n";

    for (size_t i = 0; i < num_cmp_ops; i++)
    {
        Avx512ComparePixelsCpp(des1, src, num_pixels, cmp_ops[i], cmp_vals[i]);
        Avx512ComparePixels_(des2, src, num_pixels, cmp_ops[i], cmp_vals[i]);
        ShowResults(des1, des2, num_pixels, cmp_ops[i], cmp_vals[i], i + 1);
    }

    return 0;
}

;-------------------------------------------------
;               Ch14_03.asm
;-------------------------------------------------

        include <cmpequ.asmh>
        extern c_NumPixelsMax:qword

; Макрос CmpPixels

_CmpPixels macro CmpOp
        align 16
@@:     vmovdqa64 zmm0,zmmword ptr [rdx+rax]    ;чтение следующего блока 64 пикселя
        vpcmpub k1,zmm0,zmm4,CmpOp              ;операция сравнения
        vmovdqu8 zmm1{k1}{z},zmm5               ;установка пикселей маски в 0 или 255
        vmovdqa64 zmmword ptr [rcx+rax],zmm1    ;сохранение пикселей маски

        add rax,64                              ;обновление смещения
        sub r8,64
        jnz @B                                  ;повтор до завершения
        mov eax,1                               ;возвращаемый код успешного выполнения
        vzeroupper
        ret
        endm
```

```
; extern "C" bool Avx512ComparePixels_(uint8_t* des, const uint8_t* src,
;   size_t num_pixels, CmpOp cmp_op, uint8_t cmp_val);

        .code
Avx512ComparePixels_ proc

; Проверка значения num_pixels и правильности выравнивания пиксельных буферов
        xor eax,eax                         ;нулевой код ошибки (а также смещение массива)

        or r8,r8
        jz Done                             ;переход, если num_pixels равен нулю
        cmp r8,[c_NumPixelsMax]
        ja Done                             ;переход, если num_pixels слишком велик
        test r8,3fh
        jnz Done                            ;переход, если num_pixels % 64 != 0

        test rcx,3fh
        jnz Done                            ;переход, если массив-приемник не выровнен
        test rdx,3fh
        jnz Done                            ;переход, если массив-источник не выровнен

; Начальная инициализация
        vpbroadcastb zmm4,byte ptr [rsp+40] ;zmm4 = упакованные cmp_val
        mov r10d,255
        vpbroadcastb zmm5,r10d              ;zmm5 = упакованные 255

; Операции сравнения
        cmp r9d,0
        jne LB_NE
        _CmpPixels CMP_EQ                   ;CmpOp::EQ

LB_NE:  cmp r9d,1
        jne LB_LT
        _CmpPixels CMP_NEQ                  ;CmpOp::NE

LB_LT:  cmp r9d,2
        jne LB_LE
        _CmpPixels CMP_LT                   ;CmpOp::LT

LB_LE:  cmp r9d,3
        jne LB_GT
        _CmpPixels CMP_LE                   ;CmpOp::LE

LB_GT:  cmp r9d,4
        jne LB_GE
        _CmpPixels CMP_NLE                  ;CmpOp::GT

LB_GE:  cmp r9d,5
        jne Done
        _CmpPixels CMP_NLT                  ;CmpOp::GE

Done:   vzeroupper
        ret
Avx512ComparePixels_ endp
        end
```

В верхней части заголовочного файла Ch14_03.h находится перечисление с именем CmpOp, которое содержит идентификаторы для общих операций сравнения. Далее следуют объявления функций из примера. Функции C++ Init и ShowResults – это вспомогательные функции, которые выполняют инициализацию тестового массива и отображают результаты. Исходный код этих функций не показан в листинге 14.3, но включен в пакет загрузки главы. Функция Avx512ComparePixels_ реализует на языке ассемблера AVX-512 алгоритм определения порога пикселей.

Функция Avx512ComparePixelsCpp содержит реализацию на языке C++ обновленного алгоритма определения порога. Эта функция начинает свое выполнение с проверки размера num_pixels и делимости на 64. Затем она проверяет, что пиксельные буферы src и des правильно выровнены по 64-байтовой границе. Следующий код проверки аргумента – это оператор switch, который применяет селектор cmp_op для выбора операции сравнения. Каждый блок кода case оператора switch представляет собой простой цикл for, который сравнивает src[i] с cmp_val с использованием указанного оператора и устанавливает для пикселей изображения маски значение 0xff (истина) или 0x00 (ложь). Функция main содержит код, который выделяет буферы пикселей изображения, вызывает функции Avx512ComparePixelsCpp и Avx512ComparePixels_ с использованием различных операторов сравнения и отображает результаты.

Код на языке ассемблера в листинге 14.3 начинается с макроса _CmpPixels. Этот макрос генерирует код AVX-512, реализующий цикл обработки для оператора сравнения пикселей. Макрос _CmpPixels требует следующих инициализаций регистров перед его использованием: RAX = 0, RCX = буфер пикселей изображения маски, RDX = буфер пикселей изображения в градациях серого, R8 = количество пикселей, ZMM4 = пороговые значения упакованных байтов и ZMM5 = упакованные байты значений 0xff. Каждая итерация рабочего цикла _CmpPixels начинается с команды vmovdqa64 zmm0,zmmword ptr [rdx+rax], которая загружает 64 8-разрядных целых числа без знака в регистр ZMM0. Затем команда vpcmpub k1,zmm0,zmm4,CmpOp сравнивает значения интенсивности пикселей в градациях серого в ZMM0 с упакованными значениями в ZMM4 и сохраняет результирующую маску в регистре маски K1. Последующая команда vmovdqu8 zmm1{k1}{z},zmm5 устанавливает каждое значение пикселя маски в ZMM1 равным 0xff (истинное сравнение) или 0x00 (ложное сравнение) в соответствии со значением соответствующей позиции бита в K1. Затем команда vmovdqa64 zmmword ptr [rcx+rax],zmm1 сохраняет 64 пикселя маски в буфер пикселей изображения маски.

Функция Avx512ComparePixels_ использует макрос _CmpPixels для реализации того же алгоритма, что и его аналог в C++ Avx512ComparePixelsCpp. После проверки допустимости аргументов команда vpbroadcastb zmm4,byte ptr [rsp+40] передает cmp_val в широковещательном режиме каждому байтовому элементу в регистре ZMM4. Следующие две команды, mov r10d,255 и vpbroadcastb zmm5,r10d, загружают значение 0xff в каждый байтовый элемент ZMM5. Оставшийся код в Avx512ComparePixels_ использует значение аргумента cmp_val для реализации специального оператора переключения, который использует макрос _CmpPixels. Обратите внимание, что эта функция применяет сравнение CMP_NLE (не меньше или равно) или CMP_NLT (не меньше) для аргумента макроса _CmpPixels CmpOp вместо CMP_GT или CMP_GE. Причина этого в том, что команда vpcmpub в _CmpPixels не поддерживает использование сравнений CMP_GT и CMP_GE (математически эти

последние сравнения эквивалентны CMP_NLE и CMP_NLT, но им присваиваются разные значения в cmpequ.asmh). Ниже показаны результаты выполнения примера исходного кода Ch14_03:

```
Результаты Ch14_03

Пример #1
  num_pixels: 4194304
  cmp_op: EQ
  cmp_val: 197
  Пиксельные маски идентичны
  Количество ненулевых пикселей маски = 16424

Пример #2
  num_pixels: 4194304
  cmp_op: NE
  cmp_val: 222
  Пиксельные маски идентичны
  Number of non-zero mask pixels = 4177927

Пример #3
  num_pixels: 4194304
  cmp_op: LT
  cmp_val: 43
  Пиксельные маски идентичны
  Количество ненулевых пикселей маски = 703652

Пример #4
  num_pixels: 4194304
  cmp_op: LE
  cmp_val: 43
  Пиксельные маски идентичны
  Количество ненулевых пикселей маски = 719787

Пример #5
  num_pixels: 4194304
  cmp_op: GT
  cmp_val: 129
  Пиксельные маски идентичны
  Количество ненулевых пикселей маски = 2065724

Пример #6
  num_pixels: 4194304
  cmp_op: GE
  cmp_val: 222
  Пиксельные маски идентичны
  Количество ненулевых пикселей маски = 556908
```

14.2.3. Статистика изображений

В листинге 14.4 показан исходный код примера Ch14_04. В этом примере показано, как вычислить среднее значение и среднеквадратическое отклонение изображения в градациях серого с использованием значений интенсивности пикселей. Чтобы сделать пример исходного кода Ch14_04 немного более инте-

ресным, функции языка C++ и ассемблера используют только значения пикселей, которые находятся между двумя пороговыми пределами. Значения пикселей вне этих пределов исключаются из любых вычислений.

Листинг 14.4. Пример Ch14_04

```
//-----------------------------------------------
//              Ch14_04.h
//-----------------------------------------------

#pragma once
#include <cstdint>

// Эта структура должна совпадать со структурой, объявленной в Ch14_04_.asm.
struct ImageStats
{
    uint8_t* m_PixelBuffer;
    uint64_t m_NumPixels;
    uint32_t m_PixelValMin;
    uint32_t m_PixelValMax;
    uint64_t m_NumPixelsInRange;
    uint64_t m_PixelSum;
    uint64_t m_PixelSumOfSquares;
    double m_PixelMean;
    double m_PixelSd;
};

// Ch14_04.cpp
extern bool Avx512CalcImageStatsCpp(ImageStats& im_stats);

// Ch14_04_.asm
extern "C" bool Avx512CalcImageStats_(ImageStats& im_stats);

// Ch04_04_BM.cpp
extern void Avx512CalcImageStats_BM(void);

// Общие константы
const uint32_t c_PixelValMin = 40;
const uint32_t c_PixelValMax = 230;

//-----------------------------------------------
//              Ch14_04.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <cstdint>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include <stdexcept>
#include "Ch14_04.h"
#include "AlignedMem.h"
#include "ImageMatrix.h"

using namespace std;
```

```
extern "C" uint64_t c_NumPixelsMax = 256 * 1024;

bool Avx512CalcImageStatsCpp(ImageStats& im_stats)
{
    uint64_t num_pixels = im_stats.m_NumPixels;
    const uint8_t* pb = im_stats.m_PixelBuffer;

    // Проверка правильности аргументов
    if ((num_pixels == 0) || (num_pixels > c_NumPixelsMax))
        return false;
    if (!AlignedMem::IsAligned(pb, 64))
        return false;

    // Проверка промежуточных сумм
    im_stats.m_PixelSum = 0;
    im_stats.m_PixelSumOfSquares = 0;
    im_stats.m_NumPixelsInRange = 0;

    for (size_t i = 0; i < num_pixels; i++)
    {
        uint32_t pval = pb[i];

        if (pval >= im_stats.m_PixelValMin && pval <= im_stats.m_PixelValMax)
        {
            im_stats.m_PixelSum += pval;
            im_stats.m_PixelSumOfSquares += pval * pval;
            im_stats.m_NumPixelsInRange++;
        }
    }

    // Вычисление среднего значения и среднеквадратического отклонения
    double temp0 = (double)im_stats.m_NumPixelsInRange * im_stats.m_PixelSumOfSquares;
    double temp1 = (double)im_stats.m_PixelSum * im_stats.m_PixelSum;
    double var_num = temp0 - temp1;
    double var_den = (double)im_stats.m_NumPixelsInRange * (im_stats.m_NumPixelsInRange -
1);
    double var = var_num / var_den;

    im_stats.m_PixelMean = (double)im_stats.m_PixelSum / im_stats.m_NumPixelsInRange;
    im_stats.m_PixelSd = sqrt(var);

    return true;
}

void Avx512CalcImageStats()
{
    const wchar_t* image_fn = L"..\\Ch14_Data\\TestImage4.bmp";

    ImageStats is1, is2;
    ImageMatrix im(image_fn);
    uint64_t num_pixels = im.GetNumPixels();
    uint8_t* pb = im.GetPixelBuffer<uint8_t>();

    is1.m_PixelBuffer = pb;
    is1.m_NumPixels = num_pixels;
    is1.m_PixelValMin = c_PixelValMin;
```

```
    is1.m_PixelValMax = c_PixelValMax;

    is2.m_PixelBuffer = pb;
    is2.m_NumPixels = num_pixels;
    is2.m_PixelValMin = c_PixelValMin;
    is2.m_PixelValMax = c_PixelValMax;

    const char nl = '\n';
    const char* s = " | ";
    const unsigned int w1 = 22;
    const unsigned int w2 = 12;

    cout << fixed << setprecision(6) << left;
    wcout << fixed << setprecision(6) << left;

    cout << "\nРезультаты Avx512CalcImageStats\n";
    wcout << setw(w1) << "image_fn:" << setw(w2) << image_fn << nl;
    cout << setw(w1) << "num_pixels:" << setw(w2) << num_pixels << nl;
    cout << setw(w1) << "c_PixelValMin:" << setw(w2) << c_PixelValMin << nl;
    cout << setw(w1) << "c_PixelValMax:" <<  setw(w2) << c_PixelValMax << nl;

    bool rc1 = Avx512CalcImageStatsCpp(is1);
    bool rc2 = Avx512CalcImageStats_(is2);

    if (!rc1 || !rc2)
    {
        cout << "Неправильный возвращаемый код\n";
        cout << "  rc1 = " << rc1 << '\n';
        cout << "  rc2 = " << rc2 << '\n';
        return;
    }

    cout << nl;

    cout << setw(w1) << "m_NumPixelsInRange: ";
    cout << setw(w2) << is1.m_NumPixelsInRange << s;
    cout << setw(w2) << is2.m_NumPixelsInRange << nl;

    cout << setw(w1) << "m_PixelSum:";
    cout << setw(w2) << is1.m_PixelSum << s;
    cout << setw(w2) << is2.m_PixelSum << nl;

    cout << setw(w1) << "m_PixelSumOfSquares:";
    cout << setw(w2) << is1.m_PixelSumOfSquares << s;
    cout << setw(w2) << is2.m_PixelSumOfSquares << nl;

    cout << setw(w1) << "m_PixelMean:";
    cout << setw(w2) << is1.m_PixelMean << s;
    cout << setw(w2) << is2.m_PixelMean << nl;

    cout << setw(w1) << "m_PixelSd:";
    cout << setw(w2) << is1.m_PixelSd << s;
    cout << setw(w2) << is2.m_PixelSd << nl;
}

int main()
```

```
{
    try
    {
        Avx512CalcImageStats();
        Avx512CalcImageStats_BM();
    }

    catch (runtime_error& rte)
    {
        cout << "Обнаружено исключение при выполнении программы - " << rte.what() << '\n';
    }

    catch (...)
    {
        cout << "Аварийное завершение программы\n";
        cout << "Файл = " << __FILE__ << '\n';
    }

    return 0;
}

;-----------------------------------------------
;              Ch14_04.asm
;-----------------------------------------------

        include <cmpequ.asmh>
        include <MacrosX86-64-AVX.asmh>
        extern c_NumPixelsMax:qword

; Эта структура должна совпадать со структурой, объявленной в Ch14_04.h
ImageStats          struct
PixelBuffer         qword ?
NumPixels           qword ?
PixelValMin         dword ?
PixelValMax         dword ?
NumPixelsInRange    qword ?
PixelSum            qword ?
PixelSumOfSquares   qword ?
PixelMean           real8 ?
PixelSd             real8 ?
ImageStats          ends

_UpdateSums macro Disp
        vpmovzxbd zmm0,xmmword ptr [rcx+Disp]   ;zmm0 = 16 пикселей
        vpcmpud k1,zmm0,zmm31,CMP_GE            ;k1 = пиксельная маска >= pixel_val_min
        vpcmpud k2,zmm0,zmm30,CMP_LE            ;k2 = пиксельная маска <= pixel_val_max
        kandw k3,k2,k1                          ;k3 = маска пикселей внутри диапазона
        vmovdqa32 zmm1{k3}{z},zmm0              ;zmm1 = пиксели внутри диапазона
        vpaddd zmm16,zmm16,zmm1                 ;обновление упакованного значения pixel_sum
        vpmulld zmm2,zmm1,zmm1
        vpaddd zmm17,zmm17,zmm2                 ;обновление упакованного pixel_sum_of_
squares
        kmovw rax,k3
        popcnt rax,rax                          ;количество пикселей, входящих в диапазон
        add r10,rax                             ;обновление num_pixels_in_range
        endm
```

```
; extern "C" bool Avx512CalcImageStats_(ImageStats& im_stats);

        .code
Avx512CalcImageStats_ proc frame
        _CreateFrame CIS_,0,0,rsi,r12,r13
        _EndProlog

; Проверка корректности num_pixels и выравнивания pixel_buff
        xor eax,eax                         ;возвращаемый код ошибки

        mov rsi,rcx                                 ;rsi = указатель im_stats
        mov rcx,qword ptr [rsi+ImageStats.PixelBuffer] ;rcx = указатель пиксельного буфера
        mov rdx,qword ptr [rsi+ImageStats.NumPixels]   ;rdx = num_pixels

        test rdx,rdx
        jz Done                             ;переход, если num_pixels
        cmp rdx,[c_NumPixelsMax]
        ja Done                             ;переход, если num_pixels слишком велик

        test rcx,3fh
        jnz Done                            ;переход, если pixel_buff не выровнен

; Начальная инициализация
        mov r8d,dword ptr [rsi+ImageStats.PixelValMin]
        mov r9d,dword ptr [rsi+ImageStats.PixelValMax]

        vpbroadcastd zmm31,r8d              ;упакованное значение pixel_val_min
        vpbroadcastd zmm30,r9d              ;упакованное значение pixel_val_max

        vpxorq zmm29,zmm29,zmm29            ;упакованное значение pixel_sum
        vpxorq zmm28,zmm28,zmm28            ;упакованное значение pixel_sum_of_squares
        xor r10d,r10d                       ;num_pixels_in_range = 0

; Вычисление упакованной версии pixel_sum и pixel_sum_of_squares
        cmp rdx,64
        jb LB1                              ;переход, если меньше, чем 64 пикселя

        align 16
@@:     vpxord zmm16,zmm16,zmm16            ;упакованное pixel_sum = 0
        vpxord zmm17,zmm17,zmm17            ;упакованное pixel_sum_of_squares = 0

        _UpdateSums 0                       ;обработка pixel_buff[i+15]:pixel_buff[i]
        _UpdateSums 16                      ;обработка pixel_buff[i+31]:pixel_buff[i+16]
        _UpdateSums 32                      ;обработка pixel_buff[i+47]:pixel_buff[i+32]
        _UpdateSums 48                      ;обработка pixel_buff[i+63]:pixel_buff[i+48]

        vextracti32x8 ymm0,zmm16,1          ;извлечение старших 8 pixel_sum (dwords)
        vpaddd ymm1,ymm0,ymm16
        vpmovzxdq zmm2,ymm1
        vpaddq zmm29,zmm29,zmm2             ;обновление упакованного pixel_sum (qwords)

        vextracti32x8 ymm0,zmm17,1          ;извлечение старших 8 pixel_sum_of_squares
(dwords)
        vpaddd ymm1,ymm0,ymm17
        vpmovzxdq zmm2,ymm1
        vpaddq zmm28,zmm28,zmm2             ;обновление упакованного pixel_sum_of_squares
(qwords)
```

```
        add rcx,64                          ;обновление указателя pb
        sub rdx,64                          ;обновление num_pixels
        cmp rdx,64
        jae @B                              ;повтор до завершения

        align 16
LB1:    test rdx,rdx
        jz LB3                              ;переход, если больше не осталось пикселей

        xor r13,r13                         ;pixel_sum = 0
        xor r12,r12                         ;pixel_sum_of_squares = 0
        mov r11,rdx                         ;количество оставшихся пикселей

@@:     movzx rax,byte ptr [rcx]            ;чтение следующего пикселя
        cmp rax,r8
        jb LB2                              ;переход, если пиксель < pval_min
        cmp rax,r9
        ja LB2                              ;переход, если пиксель > pval_max

        add r13,rax                         ;сложение с pixel_sum
        mul rax
        add r12,rax                         ;сложение с pixel_sum_of_squares
        add r10,1                           ;обновление num_pixels_in_range

LB2:    add rcx,1
        sub r11,1
        jnz @B                              ;повтор до завершения

; Сохранение num_pixel_in_range
LB3:    mov qword ptr [rsi+ImageStats.NumPixelsInRange],r10

; Сокращение упакованного pixel_sum до одиночного четверного слова
        vextracti64x4 ymm0,zmm29,1
        vpaddq ymm1,ymm0,ymm29
        vextracti64x2 xmm2,ymm1,1
        vpaddq xmm3,xmm2,xmm1
        vpextrq rax,xmm3,0
        vpextrq r11,xmm3,1
        add rax,r11                         ;rax = сумма четверных слов в zmm29
        add r13,rax                         ;прибавление скалярного значения pixel_sum

        mov qword ptr [rsi+ImageStats.PixelSum],r13

; Сокращение упакованного pixel_sum_of_squares до одиночного четверного слова
        vextracti64x4 ymm0,zmm28,1
        vpaddq ymm1,ymm0,ymm28
        vextracti64x2 xmm2,ymm1,1
        vpaddq xmm3,xmm2,xmm1
        vpextrq rax,xmm3,0
        vpextrq r11,xmm3,1
        add rax,r11                         ;rax = сумма четверных слов в zmm28
        add r12,rax                         ; прибавление скалярного pixel_sum_of_squares

        mov qword ptr [rsi+ImageStats.PixelSumOfSquares],r12
```

```
; Вычисление финального среднего значения и среднеквадратического отклонения
        vcvtusi2sd xmm0,xmm0,r10            ;num_pixels_in_range (DPFP)
        sub r10,1
        vcvtusi2sd xmm1,xmm1,r10            ;num_pixels_in_range - 1 (DPFP)
        vcvtusi2sd xmm2,xmm2,r13            ;pixel_sum (DPFP)
        vcvtusi2sd xmm3,xmm3,r12            ;pixel_sum_of_squares (DPFP)
        vdivsd xmm4,xmm2,xmm0               ;финальное значение pixel_mean

        vmovsd real8 ptr [rsi+ImageStats.PixelMean],xmm4

        vmulsd xmm4,xmm0,xmm3               ;num_pixels_in_range * pixel_sum_of_squares
        vmulsd xmm5,xmm2,xmm2               ;pixel_sum * pixel_sum
        vsubsd xmm2,xmm4,xmm5               ;var_num
        vmulsd xmm3,xmm0,xmm1               ;var_den
        vdivsd xmm4,xmm2,xmm3               ;вычисление отклонения
        vsqrtsd xmm0,xmm0,xmm4              ;финальное значение pixel_sd

        vmovsd real8 ptr [rsi+ImageStats.PixelSd],xmm0

        mov eax,1                           ;возвращаемый код успешного выполнения

Done:   vzeroupper
        _DeleteFrame rsi,r12,r13
        ret
Avx512CalcImageStats_ endp
        end
```

Среднее значение и среднеквадратическое отклонение пикселей в изображении в градациях серого можно рассчитать с помощью следующих уравнений:

$$\overline{x} = \frac{1}{n}\sum_i x_i;$$

$$s = \sqrt{\frac{n\sum_i x_i^2 - \left(\sum_i x_i\right)^2}{n(n-1)}}.$$

В уравнениях среднего значения и среднеквадратического отклонения символ x_i представляет пиксель буфера изображения, а n обозначает количество пикселей. Если вы внимательно изучите эти уравнения, то заметите, что необходимо вычислить две промежуточные суммы: сумму всех пикселей и сумму всех значений пикселей в квадрате. Как только эти величины станут известны, можно с помощью простых арифметических действий определить среднее значение и среднеквадратичное отклонение. Приведенное здесь уравнение среднеквадратического отклонения легко вычисляется и подходит для этого примера исходного кода. Однако в других случаях это же уравнение часто бывает не применимо, особенно в случае значений с плавающей запятой. Рекомендую обратиться к справочным материалам по вычислению статистической дисперсии, которые перечислены в приложении, прежде чем использовать это уравнение в одной из ваших собственных программ.

Листинг 14.4 начинается с заголовочного файла C++ `Ch14_04.h`, который содержит объявление структуры с именем `ImageStats`. Эта структура используется

для передачи данных изображения в вычислительные функции C++ и на языке ассемблера и возврата результатов. Семантически эквивалентная структура также определена в файле кода на языке ассемблера Ch14_04_.asm. Файл Ch14_04.h включает в себя и определения констант c_PixelValMin и c_ PixelValueMax, обозначающие пределы диапазона, между которыми должно находиться значение пикселя, чтобы его можно было включить в любые статистические вычисления.

Функция Avx512CalcImageStatsCpp является основной вычислительной функцией в коде C++. Она требует указателя на структуру ImageStats в качестве единственного аргумента. После проверки аргумента Avx512CalcImageStatsCpp инициализирует нулевыми значениями промежуточные суммы ImageStats m_ PixelSum, m_PixelSumOfSquares и m_NumPixelsInRange. Далее следует простой цикл for, который вычисляет m_ PixelSum и m_PixelSumOfSquares. Во время каждой итерации цикла значения пикселей проверяются на предмет допустимости перед включением в какие-либо вычисления. После вычисления промежуточных сумм функция Avx512CalcImageStatsCpp вычисляет окончательное среднее значение и среднеквадратическое отклонение. Обратите внимание, что для вычисления этих статистических величин вместо m_NumPixels используется m_NumPixelsInRange. Оставшийся код в Ch14_04.cpp выполняет инициализацию тестового примера, вызывает функции вычисления и передает результаты в cout.

В начале файла Ch14_04_.asm находится ассемблерная версия структуры ImageStats. За ней следует макроопределение _UpdateSums, работа которого будет описана немного позже. Выполнение функции Avx512CalcImageStats_ начинается с тех же проверок достоверности аргументов, что и в аналоге на языке C++. Затем она инициализирует упакованные версии промежуточных значений PixelValMin и PixelValMax. Потом команды vpxorq инициализируют нулевыми значениями упакованные четырехсловные версии PixelSum и PixelSumOfSquares. Обратите внимание, что команды vpxor[d|q] (и другие побитовые логические значения AVX-512) могут дополнительно указывать операнд регистра маски для выполнения маскирования слиянием или нулем элементов двойного слова или четверного слова. Последняя команда инициализации, xor r10d,r10d, устанавливает значение NumPixelsInRange равным нулю.

Рабочий цикл в функции Avx512CalcImageStats_ обрабатывает 64 пикселя на каждой итерации. Перед началом итерации регистр RDX проверяется, чтобы убедиться, что осталось не менее 64 пикселей. Каждая итерация цикла начинается с двух команд vpxord, которые инициализируют нулем упакованные версии двойного слова pixel_sum и pixel_sum_of_squares. Далее следуют четыре экземпляра макроса _UpdateSum, которые в совокупности обрабатывают следующую группу из 64 пикселей. Первая команда этого макроса, vpmovzxbd zmm0,xmmword ptr [rcx+Disp], загружает 16 значений байтов без знака из исходного пиксельного буфера и сохраняет эти значения как двойные слова без знака в регистре ZMM0. Следующие команды vpcmpud k1,zmm0,zmm31,CMP_GE, vpcmpud k2,zmm0,zmm30,CMP_LE и kandw k3,k2,k1 загружают регистр маски K3 значением маски пикселей, которые больше или равны pixel_val_min и меньше или равны pixel_val_max. Затем команда vmovdqa32 zmm1{k3}{z},zmm0 использует нулевую маскировку, чтобы эффективно исключить значения пикселей, выходящие за пределы допустимого диапазона, из дальнейших вычислений. Последующие

команды `vpaddd` и `vpmulld` обновляют упакованные количества двойных слов `pixel_sum` и `pixel_sum_of_squares`. Затем обновляется общее количество пикселей в диапазоне в R10 с использованием команд `kmovw rax,k3`, `popcnt rax,rax` и `add r10,rax`. Эти операции наглядно представлены на рис. 14.2. Обратите внимание, что на этом рисунке показаны только младшие 256 бит каждого регистра ZMM и младшие 8 бит каждого регистра маски.

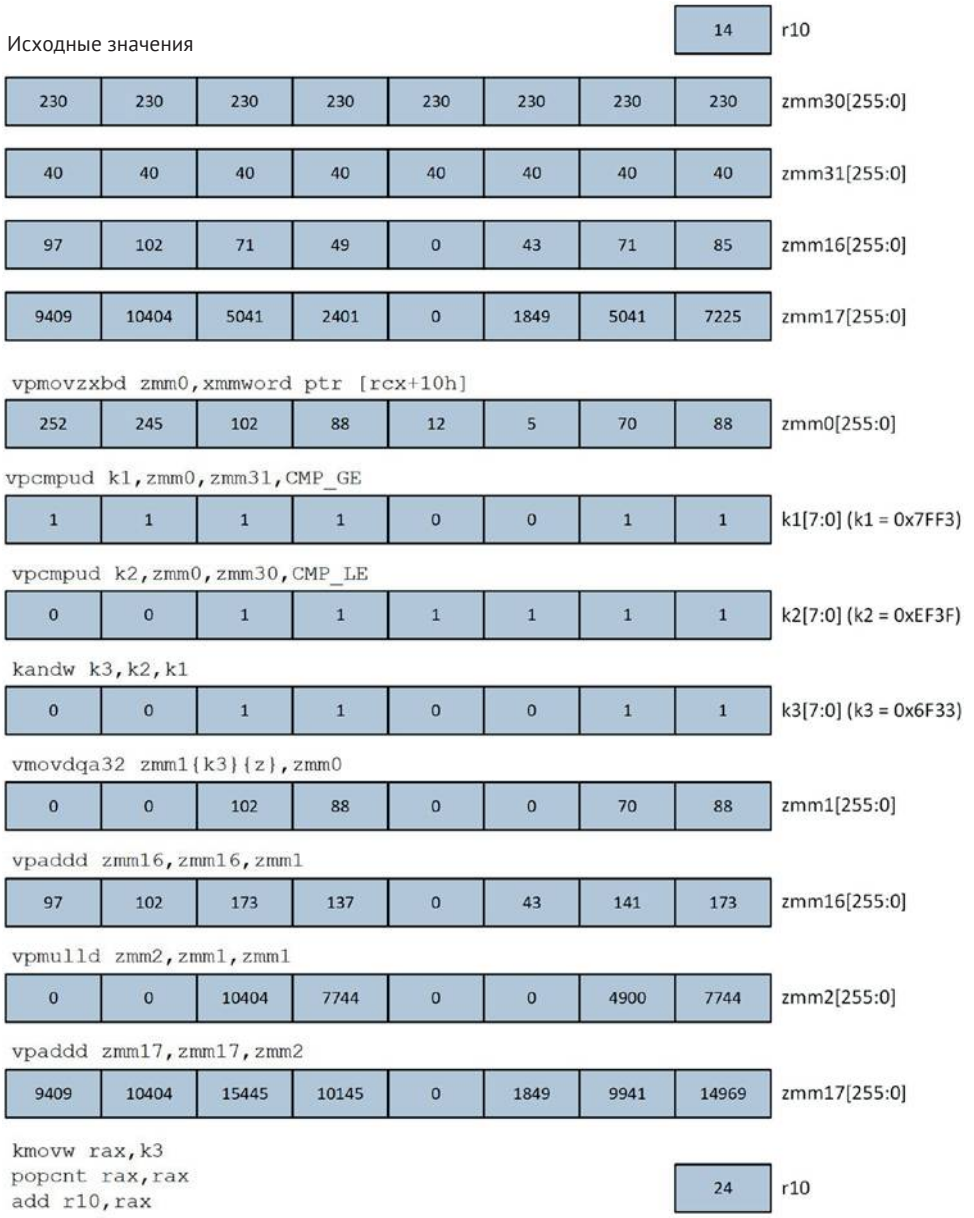


Рис. 14.2. Расчеты, выполняемые командами в макросе `_UpdateSums`

После четырех использований _UpdateSums двухсловные элементы регистров ZMM16 и ZMM17 содержат упакованные копии значений pixel_sum и pixel_sum_of_squares для текущего блока из 64 пикселей. Команды vextracti32x8 ymm0,zmm16,1 и vpaddd ymm1,ymm0,ymm16 уменьшают количество значений двойного слова в регистре ZMM16 с 16 до 8. Последующая команда vpmovzxdq zmm2,ymm1 преобразует эти двойные слова в четверные, а команда vpaddq zmm29,zmm29,zmm2 обновляет глобальные упакованные значения pixel_sum четверного слова, которые хранятся в регистре ZMM29. Затем аналогичная последовательность команд используется для обновления глобальных упакованных значений квадратов pixel_sum_of_squares в регистре ZMM28. Следом за этими командами рабочий цикл обновляет регистр указателя и счетчики; цикл повторяется до тех пор, пока количество оставшихся пикселей не станет меньше 64.

Блок кода, который начинается с метки LB1, вычисляет pixel_sum и pixel_sum_of_squares для последних нескольких пикселей (если есть), используя скалярную целочисленную арифметику и регистры общего назначения. Последовательность команд extract (vextracti64x4, vextracti64x2 и vpextrq) и vpaddq сокращает восемь упакованных значений pixel_sum в ZMM29 до одного значения в форме четверного слова. Затем аналогичная последовательность команд используется для вычисления окончательного значения pixel_sum_of_squares. Обратите внимание, что эти промежуточные результаты сохраняются в структуре ImageStats, на которую указывает регистр RCX. Затем функция Avx512CalcImageStats_ выполняет цепочку команд vcvtusi2sd для преобразования промежуточных результатов из беззнаковых целочисленных четверных слов в числа с плавающей запятой двойной точности. Окончательное среднее значение и среднеквадратическое отклонение вычисляются с использованием скалярной арифметики с плавающей запятой двойной точности. Ниже приведены результаты выполнения примера исходного кода Ch14_04. В табл. 14.1 показаны результаты измерения быстродействия для вычислительных функций Avx512CalcImageStatsCpp и Avx512CalcImageStats_ на языке C++ и на языке ассемблера соответственно.

```
Результаты Avx512CalcImageStats

image_fn:            ..\Ch14_Data\TestImage4.bmp
num_pixels:          258130
c_PixelValMin:       40
c_PixelValMax:       230

m_NumPixelsInRange:  229897     | 229897
m_PixelSum:          32574462   | 32574462
m_PixelSumOfSquares: 5139441032 | 5139441032
m_PixelMean:         141.691549 | 141.691549
m_PixelSd:           47.738056  | 47.738056

Измерение скорости выполнения Avx512CalcImageStats_BM - подождите
Результаты сохранены в файл Ch14_04_Avx512CalcImageStats_BM_CHROMIUM.csv
```

Таблица 14.1. Результаты измерения быстродействия функций вычисления статистики изображений с использованием `TestImage4.bmp`

Процессор	Avx512CalcImageStatsCpp	vx512CalcImageStats_
i7-4790S	----	----
i9-7900X	404	29
i7-8700K	----	----

14.2.4. Преобразование формата RGB в оттенки серого

В главе 10 вы узнали, как использовать набор команд AVX2 для преобразования изображения RGB в изображение в градациях серого (см. пример `Ch10_06`). В листинге 14.5 показан исходный код примера `Ch14_05`, который иллюстрирует преобразование изображения RGB в оттенки серого с использованием набора команд AVX-512.

Листинг 14.5. Пример Ch14_05

```
//-----------------------------------------------
//              Ch14_05.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <stdexcept>
#include <iomanip>
#include "Ch14_05.h"
#include "ImageMatrix.h"
#include "AlignedMem.h"

using namespace std;

extern "C" const int c_NumPixelsMin = 64;
extern "C" const int c_NumPixelsMax = 16 * 1024 * 1024;

// Коэффициенты преобразования RBG в градации серого
const float c_Coef[3] {0.2126f, 0.7152f, 0.0722f};

bool CompareGsImages(const uint8_t* pb_gs1,const uint8_t* pb_gs2, int num_pixels)
{
    for (int i = 0; i < num_pixels; i++)
    {
        if (abs((int)pb_gs1[i] - (int)pb_gs2[i]) > 1)
            return false;
    }

    return true;
}

bool Avx512RgbToGsCpp(uint8_t* pb_gs, const uint8_t* const* pb_rgb, int num_pixels, const
float coef[3])
{
    if (num_pixels < c_NumPixelsMin || num_pixels > c_NumPixelsMax)
        return false;
```

```
    if (num_pixels % 64 != 0)
        return false;
    if (!AlignedMem::IsAligned(pb_gs, 64))
        return false;

    const size_t align = 64;
    const uint8_t* pb_r = pb_rgb[0];
    const uint8_t* pb_g = pb_rgb[1];
    const uint8_t* pb_b = pb_rgb[2];

    if (!AlignedMem::IsAligned(pb_r, align))
        return false;
    if (!AlignedMem::IsAligned(pb_g, align))
        return false;
    if (!AlignedMem::IsAligned(pb_b, align))
        return false;

    for (int i = 0; i < num_pixels; i++)
    {
        uint8_t r = pb_r[i];
        uint8_t g = pb_g[i];
        uint8_t b = pb_b[i];

        float gs_temp = r * coef[0] + g * coef[1] + b * coef[2] + 0.5f;

        if (gs_temp < 0.0f)
            gs_temp = 0.0f;
        else if (gs_temp > 255.0f)
            gs_temp = 255.0f;

        pb_gs[i] = (uint8_t)gs_temp;
    }

    return true;
}

void Avx512RgbToGs(void)
{
    const wchar_t* fn_rgb = L"..\\Ch14_Data\\TestImage3.bmp";
    const wchar_t* fn_gs1 = L"Ch14_05_Avx512RgbToGs_TestImage3_GS1.bmp";
    const wchar_t* fn_gs2 = L"Ch14_05_Avx512RgbToGs_TestImage3_GS2.bmp";
    const wchar_t* fn_gs3 = L"Ch14_05_Avx512RgbToGs_TestImage3_GS3.bmp";

    ImageMatrix im_rgb(fn_rgb);
    int im_h = im_rgb.GetHeight();
    int im_w = im_rgb.GetWidth();
    int num_pixels = im_h * im_w;
    ImageMatrix im_r(im_h, im_w, PixelType::Gray8);
    ImageMatrix im_g(im_h, im_w, PixelType::Gray8);
    ImageMatrix im_b(im_h, im_w, PixelType::Gray8);
    RGB32* pb_rgb = im_rgb.GetPixelBuffer<RGB32>();
    uint8_t* pb_r = im_r.GetPixelBuffer<uint8_t>();
    uint8_t* pb_g = im_g.GetPixelBuffer<uint8_t>();
    uint8_t* pb_b = im_b.GetPixelBuffer<uint8_t>();
    uint8_t* pb_rgb_cp[3] {pb_r, pb_g, pb_b};
```

```
    for (int i = 0; i < num_pixels; i++)
    {
        pb_rgb_cp[0][i] = pb_rgb[i].m_R;
        pb_rgb_cp[1][i] = pb_rgb[i].m_G;
        pb_rgb_cp[2][i] = pb_rgb[i].m_B;
    }

    ImageMatrix im_gs1(im_h, im_w, PixelType::Gray8);
    ImageMatrix im_gs2(im_h, im_w, PixelType::Gray8);
    ImageMatrix im_gs3(im_h, im_w, PixelType::Gray8);
    uint8_t* pb_gs1 = im_gs1.GetPixelBuffer<uint8_t>();
    uint8_t* pb_gs2 = im_gs2.GetPixelBuffer<uint8_t>();
    uint8_t* pb_gs3 = im_gs3.GetPixelBuffer<uint8_t>();

    // Функции преобразования
    bool rc1 = Avx512RgbToGsCpp(pb_gs1, pb_rgb_cp, num_pixels, c_Coef);
    bool rc2 = Avx512RgbToGs_(pb_gs2, pb_rgb_cp, num_pixels, c_Coef);
    bool rc3 = Avx2RgbToGs_(pb_gs3, pb_rgb_cp, num_pixels, c_Coef);

    if (rc1 && rc2 && rc3)
    {
        im_gs1.SaveToBitmapFile(fn_gs1);
        im_gs2.SaveToBitmapFile(fn_gs2);
        im_gs2.SaveToBitmapFile(fn_gs3);

        bool c1 = CompareGsImages(pb_gs1, pb_gs2, num_pixels);
        bool c2 = CompareGsImages(pb_gs2, pb_gs3, num_pixels);

        if (c1 && c2)
            cout << "Изображения в тонах серого совпадают\n";
        else
            cout << "Изображения в тонах серого различаются\n";
    }
    else
        cout << "Ошибочный возвращаемый код\n";
}

int main()
{
    try
    {
        Avx512RgbToGs();
        Avx512RgbToGs_BM();
    }

    catch (runtime_error& rte)
    {
        cout << "сбой выполнения программы - " << rte.what() << '\n';
    }

    catch (...)
    {
        cout << "Произошло аварийное завершение программы\n";
    }

    return 0;
}
```

```
;-----------------------------------------------
;              Ch14_05.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>
        extern c_NumPixelsMin:dword
        extern c_NumPixelsMax:dword

            .const
r4_0p5      real4 0.5
r4_255p0    real4 255.0

; extern "C" bool Avx512RgbToGs_(uint8_t* pb_gs, const uint8_t* const* pb_rgb, int num_
pixels, const float coef[3]);

        .code
Avx512RgbToGs_ proc frame
        _CreateFrame RGBGS0_,0,96,r13,r14,r15
        _SaveXmmRegs xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

        xor eax,eax              ;возвращаемый код ошибки (и смещение пиксельного буфера)
        cmp r8d,[c_NumPixelsMin]
        jl Done                          ;переход, если num_pixels < мин. значения
        cmp r8d,[c_NumPixelsMax]
        jg Done                          ;переход, если num_pixels > макс. значения
        test r8d,3fh
        jnz Done                         ;переход, если (num_pixels % 64) != 0

        test rcx,3fh
        jnz Done                         ;переход, если pb_gs не выровнен

        mov r13,[rdx]
        test r13,3fh
        jnz Done                         ;переход, если pb_r не выровнен
        mov r14,[rdx+8]
        test r14,3fh
        jnz Done                         ;переход, если pb_g не выровнен
        mov r15,[rdx+16]
        test r15,3fh
        jnz Done                         ;переход, если pb_b не выровнен

; Начальная инициализация
        vbroadcastss zmm10,real4 ptr [r9]       ;zmm10 = packed coef[0]
        vbroadcastss zmm11,real4 ptr [r9+4]     ;zmm11 = packed coef[1]
        vbroadcastss zmm12,real4 ptr [r9+8]     ;zmm12 = packed coef[2]
        vbroadcastss zmm13,real4 ptr [r4_0p5]   ;zmm13 = packed 0.5
        vbroadcastss zmm14,real4 ptr [r4_255p0] ;zmm14 = packed 255.0
        vxorps zmm15,zmm15,zmm15                ;zmm15 = packed 0.0
        mov r8d,r8d                             ;r8 = num_pixels
        mov r10,16                              ;r10 – число пикселей/итераций

; Чтение следующего блока пикселей
        align 16
@@:     vpmovzxbd zmm0,xmmword ptr [r13+rax]    ;zmm0 = 16 пикселей (компонент r)
        vpmovzxbd zmm1,xmmword ptr [r14+rax]    ;zmm1 = 16 пикселей (компонент g)
        vpmovzxbd zmm2,xmmword ptr [r15+rax]    ;zmm2 = 16 пикселей (компонент b)
```

```
; Преобразование двойных слов в SPFP и умножение на коэффициенты
        vcvtdq2ps zmm0,zmm0                 ;zmm0 = 16 пикселей SPFP (r)
        vcvtdq2ps zmm1,zmm1                 ;zmm1 = 16 пикселей SPFP (g)
        vcvtdq2ps zmm2,zmm2                 ;zmm2 = 16 пикселей SPFP (b)
        vmulps zmm0,zmm0,zmm10              ;zmm0 = r * coef[0]
        vmulps zmm1,zmm1,zmm11              ;zmm1 = g * coef[1]
        vmulps zmm2,zmm2,zmm12              ;zmm2 = b * coef[2]

; Суммирование цветовых компонентов и усечение до [0.0, 255.0]
        vaddps zmm3,zmm0,zmm1               ;r + g
        vaddps zmm4,zmm3,zmm2               ;r + g + b
        vaddps zmm5,zmm4,zmm13              ;r + g + b + 0.5
        vminps zmm0,zmm5,zmm14              ;усечение пикселей выше 255.0
        vmaxps zmm1,zmm0,zmm15              ;усечение пикселей ниже 0.0

; Преобразование значений серого из SPFP в байты, сохранение результата
        vcvtps2dq zmm2,zmm1                 ;преобразование SPFP в двойные слова

        vpmovusdb xmm3,zmm2                 ;преобразование в байты
        vmovdqa xmmword ptr [rcx+rax],xmm3  ;сохранение пикселей изображения в полутонах

        add rax,r10
        sub r8,r10
        jnz @B

        mov eax,1                           ;возвращаемый код успешного выполнения
Done:   vzeroupper
        _RestoreXmmRegs xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _DeleteFrame r13,r14,r15
        ret
Avx512RgbToGs_ endp

; extern "C" bool Avx2RgbToGs_(uint8_t* pb_gs, const uint8_t* const* pb_rgb, int num_
pixels, const float coef[3]);

        .code
Avx2RgbToGs_ proc frame
        _CreateFrame RGBGS1_,0,96,r13,r14,r15
        _SaveXmmRegs xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
        _EndProlog

        xor eax,eax                         ;возвращаемый код ошибки (и смещение pixel_
buffer)
        cmp r8d,[c_NumPixelsMin]
        jl Done                             ;переход, если num_pixels < мин. значения
        cmp r8d,[c_NumPixelsMax]
        jg Done                             ;переход, если num_pixels > макс. значения
        test r8d,3fh
        jnz Done                            ;переход, если (num_pixels % 64) != 0

        test rcx,3fh
        jnz Done                            ;переход, если pb_gs не выровнен

        mov r13,[rdx]
        test r13,3fh
        jnz Done                            ;переход, если pb_r не выровнен
```

```
        mov r14,[rdx+8]
        test r14,3fh
        jnz Done                            ;переход, если pb_g не выровнен
        mov r15,[rdx+16]
        test r15,3fh
        jnz Done                            ;переход, если pb_b не выровнен

; Начальная инициализация
        vbroadcastss ymm10,real4 ptr [r9]       ;ymm10 = упакованный coef[0]
        vbroadcastss ymm11,real4 ptr [r9+4]     ;ymm11 = упакованный coef[1]
        vbroadcastss ymm12,real4 ptr [r9+8]     ;ymm12 = упакованный coef[2]
        vbroadcastss ymm13,real4 ptr [r4_0p5]   ;ymm13 = упакованный 0.5
        vbroadcastss ymm14,real4 ptr [r4_255p0] ;ymm14 = упакованный 255.0
        vxorps ymm15,ymm15,ymm15                ;ymm15 = упакованный 0.0
        mov r8d,r8d                             ;r8 = num_pixels
        mov r10,8                               ;r10 – число пикселей/итераций

; Чтение следующего блока пикселей
        align 16
@@:     vpmovzxbd ymm0,qword ptr [r13+rax]      ;ymm0 = 8 пикселей (r)
        vpmovzxbd ymm1,qword ptr [r14+rax]      ;ymm1 = 8 пикселей (g)
        vpmovzxbd ymm2,qword ptr [r15+rax]      ;ymm2 = 8 пикселей (b)

; Преобразование двойных слов в SPFP и умножение на коэффициенты
        vcvtdq2ps ymm0,ymm0                 ;ymm0 = 8 пикселей SPFP (r)
        vcvtdq2ps ymm1,ymm1                 ;ymm1 = 8 пикселей SPFP (g)
        vcvtdq2ps ymm2,ymm2                 ;ymm2 = 8 пикселей SPFP (b)
        vmulps ymm0,ymm0,ymm10              ;ymm0 = r * coef[0]
        vmulps ymm1,ymm1,ymm11              ;ymm1 = g * coef[1]
        vmulps ymm2,ymm2,ymm12              ;ymm2 = b * coef[2]

; Суммирование цветовых компонентов и усечение до [0.0, 255.0]
        vaddps ymm3,ymm0,ymm1               ;r + g
        vaddps ymm4,ymm3,ymm2               ;r + g + b
        vaddps ymm5,ymm4,ymm13              ;r + g + b + 0.5
        vminps ymm0,ymm5,ymm14              ;clip pixels above 255.0
        vmaxps ymm1,ymm0,ymm15              ;clip pixels below 0.0

; Преобразование значений серого из SPFP в байты, сохранение результата
        vcvtps2dq ymm2,ymm1                 ;преобразование SPFP в двойные слова

        vpackusdw ymm3,ymm2,ymm2
        vextracti128 xmm4,ymm3,1
        vpackuswb xmm5,xmm3,xmm4            ;байты пикселей в xmm5[31:0] и xmm5[95:64]
        vpextrd r11d,xmm5,0                 ;r11d = 4 пикселя в полутонах серого
        mov dword ptr [rcx+rax],r11d        ;сохранение пикселей
        vpextrd r11d,xmm5,2                 ;r11d = 4 пикселя в полутонах серого
        mov dword ptr [rcx+rax+4],r11d      ; сохранение пикселей

        add rax,r10
        sub r8,r10
        jnz @B

        mov eax,1                           ;возвращаемый код успешного выполнения
Done:   vzeroupper
        _RestoreXmmRegs xmm10,xmm11,xmm12,xmm13,xmm14,xmm15
```

```
        _DeleteFrame r13,r14,r15
        ret
Avx2RgbToGs_ endp
        end
```

Алгоритм, который используется в этом примере для преобразования RGB в оттенки серого, аналогичен алгоритму, который использовался в примере `Ch10_06`. Как объяснялось в главе 10, алгоритм использует простое взвешенное среднее значение для преобразования пикселя изображения RGB в пиксель изображения в градациях серого. Функция на языке C++ `Avx512RgbToGs` начинает свою работу с загрузки файла тестового изображения. Затем она копирует пиксели RGB `im_rgb` в три отдельных буфера изображения компонентов цвета. Причина этого заключается в том, что для функций преобразования RGB в оттенки серого в этом примере требуется структура массивов (AOS) вместо массива структур (SOA), который использовался в примере исходного кода `Ch10_06`. После выделения пространства под буферы изображений в градациях серого `Avx512RgbToGs` вызывает функции преобразования на языках C++ и ассемблера. Полученные буферы изображений в градациях серого затем сравниваются на совпадение и сохраняются.

Код на языке ассемблера в листинге 14.5 включает две функции: `Avx512Rgb2Gs_` и `Avx2Rgb2Gs_`. Как следует из соответствующих префиксов имен, эти функции выполняют преобразование изображений из RGB в оттенки серого с использованием команд AVX-512 и AVX2 соответственно. Функция `Avx512Rgb2Gs_` начинается с проверки `num_pixels` на предмет размера и делимости на 64. Затем выполняется проверка пиксельных буферов источника и приемника на предмет правильного выравнивания. Далее серия команд `vbroadcastss` загружает упакованные версии коэффициентов преобразования цвета в регистры ZMM10, ZMM11 и ZMM12. За ней следует другой набор команд `vbroadcastss`, которые передают константы с плавающей запятой одинарной точности 0.5, 255.0 и 0.0 в регистры ZMM13, ZMM14 и ZMM15. Команда `mov r8d,r8d` расширяет нулями `num_pixels` в регистре R8, а команда `mov r10,16` загружает в R10 количество пикселей для обработки во время каждой итерации цикла.

Каждая итерация рабочего цикла `Avx512Rgb2Gs_` начинается с трех команд `vpmovzxbd`, которые загружают 16 значений красного, зеленого и синего пикселей в регистры ZMM0, ZMM1 и ZMM2. Последующие команды `vcvtdq2ps` преобразуют значения пикселей в формате двойного слова в числа с плавающей запятой одинарной точности. Затем значения цвета с плавающей запятой умножаются на соответствующие цветовые коэффициенты с использованием серии команд `vmulps`. Далее эти значения суммируются с применением трех команд `vaddps`. Результирующие 16 значений пикселей в градациях серого затем обрезаются до [0,0, 255,0] и преобразуются в значения в формате двойного слова. Команда `vpmovusdb xmm3,zmm2` уменьшает размер значения из двойного слова в байты, используя беззнаковое насыщение, а команда `vmovdqa xmmword ptr [rcx+rax],xmm3` сохраняет 16-байтовые значения пикселей в целевом буфере изображения в градациях серого.

Функция языка ассемблера `Avx2Rgb2Gs_` идентична своему аналогу AVX-512, за исключением двух незначительных изменений: `Avx2Rgb2Gs_` использует команды AVX2 и регистр YMM, настроенный для выполнения необходимых вычислений; она также применяет команды `vpackusdw` и `vpackuswb` в сочетании с не-

сколькими другими командами для уменьшения размера из двойного слова до байта. Причина в том, что AVX2 не поддерживает команду vpmovusdb. Ниже показан результат выполнения примера исходного кода Ch14_05:

```
Изображения в тонах серого совпадают

Выполняется измерение быстродействия Avx512RgbToGs_BM - подождите
Результаты сохранены в файл Ch14_05_Avx512RgbToGs_BM_CHROMIUM.csv
```

В табл. 14.2 показаны результаты измерения быстродействия для примера исходного кода Ch14_05.

Таблица 14.2. Среднее время выполнения (мс) в задаче преобразования изображения из RGB в оттенки серого

Процессор	Avx512RgbToGsCpp	Avx512Rgb2Gs_	Avx2Rgb2Gs_
i7-4790S	----	----	----
i9-7900X	1125	134	259
i7-8700K	----	----	----

Разница в скорости выполнения эталонных тестов между реализациями AVX-512 и AVX2 алгоритма преобразования RGB в оттенки серого согласуется с тем, что можно было ожидать. Интересно сравнить эти числа с измерениями из примера исходного кода Ch10_06 (табл. 10.2). В этом более раннем примере в качестве буфера исходного изображения использовался массив пикселей RGB32 (или AOS), а среднее время выполнения функции преобразования Avx2ConvertRgbToGs_ составляло 593 мс. В текущем примере используются отдельные буферы пикселей изображения для каждого цветового компонента (или SOA), что значительно повышает производительность.

14.3. Заключение

В главе 14 рассмотрены следующие ключевые моменты:

- функции на языке ассемблера могут использовать продвинутые версии AVX-512 большинства команд AVX и упакованные целочисленные команды AVX2 для выполнения операций с использованием операндов шириной 512, 256 и 128 бит;
- функции на языке ассемблера могут использовать команды vmovdqa[32|64] и vmovdqu[8|16|32|64] для выполнения выровненных и невыровненных перемещений/присвоений упакованных целочисленных операндов;
- функции на языке ассемблера могут использовать команды vpmovus[qd|qw|qb|dw|db|wb] для выполнения сокращений размера упакованных целочисленных значений с использованием беззнакового насыщения. AVX-512 также поддерживает аналогичный набор команд по уменьшению размера упакованных целочисленных значений с использованием знакового насыщения;

- команды `vpcmpu[b|w|d|q]` выполняют операции сравнения упакованных целых чисел без знака и сохраняют результирующую маску сравнения в регистре маски;
- команды `vpand[d|q]`, `vpandn[d|q]`, `vpor[d|q]` и `vpxor[d|q]` могут использоваться с регистром маски для выполнения маскирования слиянием или нулем с использованием элементов в формате двойного или четверного слова;
- команды `vextracti[32x4|32x8|64x2|64x4]` могут применяться для извлечения упакованных значений в формате двойного или четверного слова из упакованного целочисленного операнда;
- при выполнении вычислений SIMD с использованием упакованных целочисленных операндов или операндов с плавающей запятой конструкция на основе структуры массивов часто работает значительно быстрее, чем конструкция на основе массива структур.

Глава 15

Стратегии и методы оптимизации

В предыдущих главах вы изучили основы программирования на языке ассемблера x86-64. Вы также узнали, как использовать вычислительные ресурсы AVX для выполнения операций SIMD. Чтобы максимизировать производительность вашего кода на языке ассемблера x86, часто бывает необходимо понимать важные детали внутреннего устройства процессора x86. В этой главе вы исследуете внутренние аппаратные компоненты современного многоядерного процессора x86 и лежащую в его основе микроархитектуру. Вы также узнаете, как применять определенные стратегии и методы кодирования для повышения производительности кода на языке ассемблера x86-64.

Содержание главы 15 следует рассматривать лишь как вводное руководство по рассмотренным в ней вопросам. Для всестороннего изучения микроархитектур x86 и методов оптимизации языка ассемблера как минимум потребуется несколько длинных глав или, возможно, целая книга. Основным справочным источником для материалов этой главы является «Справочное руководство по оптимизации архитектур Intel 64 и IA-32» (*Intel 64 and IA-32 Architectures Optimization Reference Manual*). Предлагаю вам самостоятельно ознакомиться с этим важным справочным руководством для получения дополнительной информации о микроархитектурах Intel x86 и методах оптимизации языка ассемблера. «Руководство по оптимизации программного обеспечения для процессоров AMD семейства 17h» (*Software Optimization Guide for AMD Family 17h Processors*) также содержит полезные рекомендации по оптимизации для программистов на ассемблере x86. Приложение включает ссылки, которые содержат дополнительную информацию о стратегиях и методах оптимизации языка ассемблера x86.

15.1. Микроархитектура процессора

Производительность процессора x86 в основном определяется его базовой микроархитектурой. *Микроархитектура процессора* характеризуется организацией и работой следующих внутренних аппаратных компонентов: конвейеров команд, декодеров, планировщиков, модулей выполнения, шин данных и кешей. Разработчики программного обеспечения, которые понимают осно-

вы микроархитектуры процессоров, часто могут почерпнуть конструктивные идеи, которые позволяют им разрабатывать более эффективный код.

В этом разделе объясняются концепции микроархитектуры процессора на примере микроархитектуры Intel Skylake. Микроархитектура Skylake используется в последних массовых процессорах Intel, в том числе в процессорах серий Core i3, i5 и i7 шестого поколения. Процессоры Core серий седьмого (Kaby Lake) и восьмого (Coffee Lake) поколений также основаны на микроархитектуре Skylake. Структурная организация и работа более ранних микроархитектур Intel, таких как Sandy Bridge и Haswell, схожи со Skylake. Большинство концепций, представленных в этом разделе, также применимы к микроархитектурам, разработанным и используемым AMD в своих процессорах, хотя лежащие в их основе аппаратные реализации различаются. Прежде чем продолжить, следует отметить, что микроархитектура Skylake, обсуждаемая в этом разделе, похожа, но отличается от микроархитектуры Skylake Server, которая упоминалась в главах, описывающих концепции и программирование AVX-512.

15.1.1. Обзор архитектуры процессора

Архитектурные детали процессора на базе Skylake или любой другой современной микроархитектуры лучше всего изучать на примере структуры многоядерного процессора. На рис. 15.1 показана упрощенная блок-схема типичного четырехъядерного процессора на базе Skylake. Обратите внимание, что каждое *ядро* ЦП включает кеши команд и данных первого уровня (L1), которые помечены как I-Cache и D-Cache соответственно. Как следует из их названий, эти кеши памяти содержат команды и данные, к которым ядро ЦП может быстро получить доступ. Каждое ядро ЦП также включает в себя унифицированный кеш второго уровня (L2), в котором хранятся как команды, так и данные. Помимо повышения производительности, кеши L1 и L2 позволяют ядрам ЦП выполнять независимые потоки команд параллельно, не обращаясь к совместно используемому кешу L3 более высокого уровня или к основной памяти.

Если ядру ЦП требуется команда или элемент данных, которых нет в его кеше L1 или L2, его необходимо загрузить из кеша L3 или основной памяти. Кеш L3 процессора разделен на несколько «срезов». Каждый срез состоит из логического контроллера и массива данных. Логический контроллер управляет доступом к соответствующему массиву данных. Он также обрабатывает *промахи кеша* (cache miss) и запись в основную память. Промах кеша случается, когда запрошенные данные отсутствуют в кеше L3 и должны быть загружены из основной памяти (промахи кеша также происходят, когда данные недоступны в кешах L1 или L2). Каждый массив данных L3 включает в себя кеш-память, которая организована в пакеты размером 64 байта, называемые *строками кеша*. Высокоскоростная *внутренняя шина* упрощает передачу данных между ядрами ЦП, кешем L3, графическим модулем и системным агентом. *Системный агент* обрабатывает трафик данных между процессором, его внешними шинами данных и основной памятью.

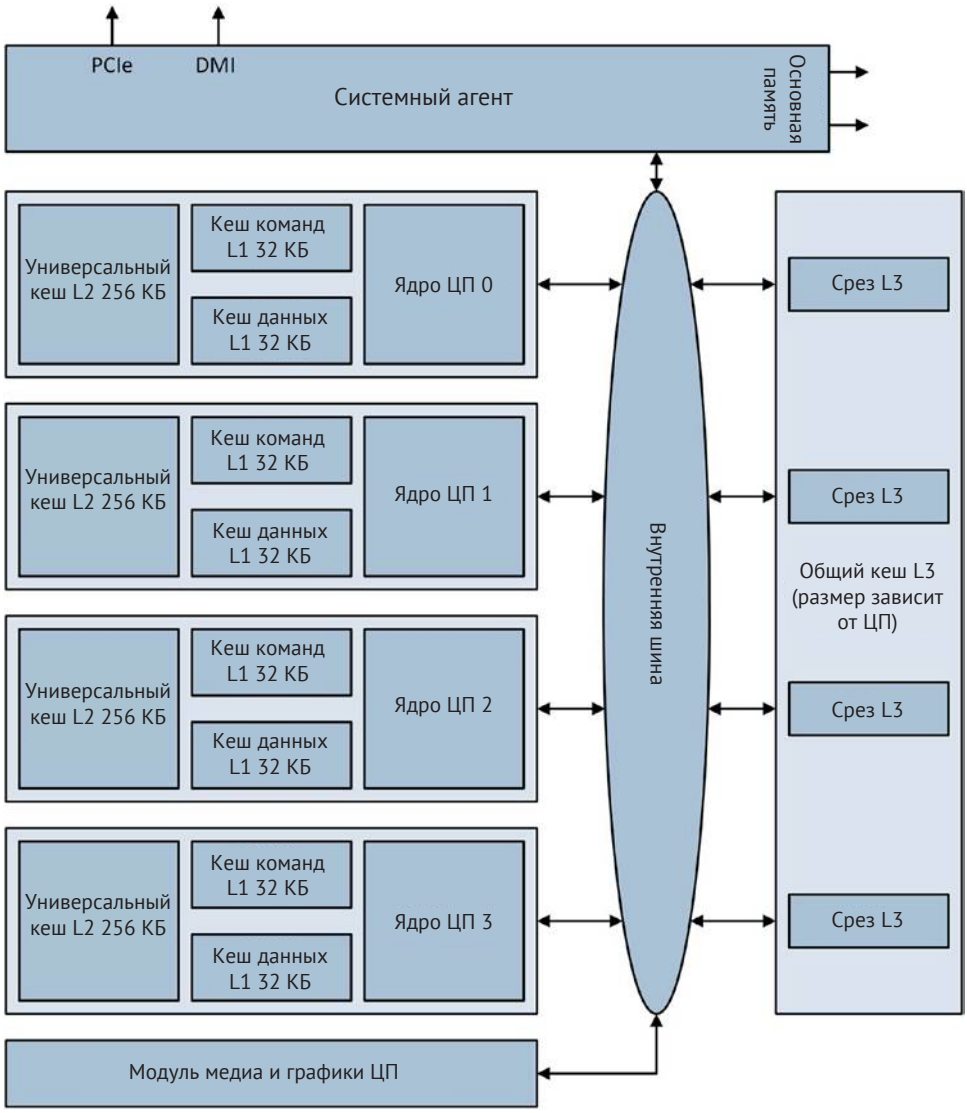


Рис. 15.1. Упрощенная блок-схема четырехъядерного процессора на базе Skylake

15.1.2. Функциональная схема конвейера микроархитектуры

Во время выполнения программы ядро ЦП выполняет пять элементарных действий: *выборку* (fetch), *декодирование* (decode), *отправку* (dispatch), *выполнение* (execute) и *завершение* (retire). Детали этих операций зависят от структуры конвейера микроархитектуры. На рис. 15.2 показана упрощенная блок-схема конвейерных функций ядра ЦП на базе Skylake. В следующих параграфах более подробно рассматриваются операции, выполняемые этими конвейерными узлами.

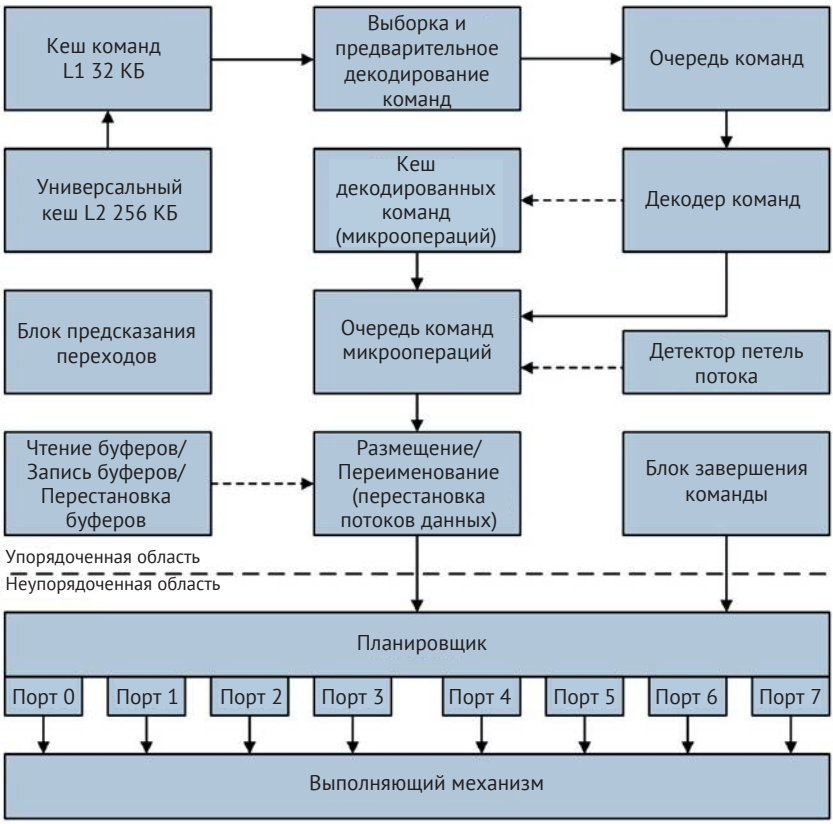


Рис. 15.2. Функциональность ядра процессора Skylake

Блок выборки и предварительного декодирования команд получает команды из I-кеша L1 и начинает процесс их подготовки к выполнению. Шаги, выполняемые на этом этапе, включают определение длины команды, декодирование префиксов x86 и маркировку свойств для помощи декодерам нисходящего потока. Блок выборки и предварительного декодирования команд также отвечает за подачу постоянного потока команд в очередь, откуда они поступают к декодерам команд.

Декодеры команд переводят команды x86 в микрооперации. *Микрооперация* – это автономная низкоуровневая команда, которая в конечном итоге выполняется одним из модулей выполнения, которые обсуждаются в следующем разделе. Количество микроопераций, генерируемых декодерами для команды x86, зависит от ее сложности. Простые команды регистр–регистр, такие как `add eax,edx` и `vpxor xmm0,xmm0,xmm0`, декодируются в одну микрооперацию. Команды, которые выполняют более сложные операции, такие как `idiv rcx` и `vdivsd ymm0,ymm1,ymm2`, требуют нескольких микроопераций. Преобразование команд x86 в микрооперации обеспечивает ряд преимуществ с точки зрения архитектуры и производительности, включая параллелизм на уровне команд и выполнение вне очереди.

Декодеры команд также выполняют две вспомогательные операции, которые улучшают использование доступной полосы пропускания конвейера. Первая из

этих операций называется *микрослиянием* (micro-fusion), которая объединяет простые микрооперации из одной и той же команды x86 в единую сложную микрооперацию. Примеры микрослияния команд включают в себя сохранение в памяти (`mov dword ptr [rbx+16],eax`) и команды вычисления, которые ссылаются на операнды в памяти (`sub r9,qword ptr [rbp+48]`). Объединенные сложные микрооперации отправляются механизмом выполнения несколько раз (каждая отправка приводит к выполнению простой микрооперации из исходной команды). Вторая вспомогательная операция, выполняемая декодерами команд, называется *макрослиянием* (macro-fusion). Макрослияние объединяет некоторые часто используемые пары команд x86 в одну микрооперацию. Примеры парного макрослияния команд включают множество (но не все) команд условного перехода, которым предшествуют команды `add`, `and`, `cmp`, `dec`, `inc`, `sub` или `test`.

Микрооперации от декодеров команд передаются в *очередь команд микроопераций* (micro-op instruction queue) для последующей отправки *планировщиком*. При необходимости они также кешируются в кеше декодированных команд. Очередь команд микроопераций также используется *детектором петель потока* (loop stream detector), который выявляет и блокирует небольшие программные циклы в очереди команд микроопераций. Это повышает производительность, поскольку небольшой цикл может выполняться многократно, не требуя дополнительных операций выборки, декодирования и чтения из кеша микроопераций.

Блок размещения/переименования (allocate/rename block) служит мостом между упорядоченными интерфейсными конвейерами и неупорядоченным планировщиком и механизмом выполнения. Он выделяет все необходимые внутренние буферы для микроопераций, а также устраняет *ложные зависимости* между микрооперациями, что облегчает выполнение вне очереди. Ложная зависимость возникает, когда двум микрооперациям требуется одновременный доступ к разным версиям одного и того же аппаратного ресурса. (В коде на языке ассемблера ложные зависимости могут возникать при использовании команд, обновляющих только младшие 8 или 16 бит 32-битного регистра.) Затем микрооперации передаются планировщику. Этот модуль ставит в очередь микрооперации до тех пор, пока не станут доступны необходимые исходные операнды. Затем он отправляет готовые к выполнению микрооперации соответствующему *исполняющему блоку* (execution unit) в *механизме выполнения* (execution engine). Модуль *сохранения и удаления* (retire unit) сохраняет результат и удаляет микрооперации, которые завершили свое выполнение с учетом исходного порядка команд программы. Он также сигнализирует о любых исключениях процессора, которые могли произойти во время выполнения микрооперации.

Наконец, *модуль прогнозирования переходов* (branch prediction unit) помогает выбрать следующий набор команд для выполнения, прогнозируя переходы, которые с наибольшей вероятностью будут выполнены, на основе последних выполненных команд. *Точка перехода* – это просто операнд-адресат команды управления переходом, такой как `jcc`, `jmp`, `call` или `ret`. Модуль прогнозирования переходов позволяет ядру процессора «обдуманно» выполнять микрооперации команды до того, как станет известен результат решения о переходе. При необходимости ядро процессора просматривает (по порядку) кеш декодированных команд, I-кеш L1, унифицированный кеш L2, кеш L3 и основную память в поисках команд для выполнения.

15.1.3. Механизм выполнения

Механизм выполнения выполняет микрооперации, переданные ему планировщиком. На рис. 15.3 показана высокоуровневая блок-схема ядра механизма выполнения на базе процессора Skylake. Прямоугольные блоки под каждым портом передачи представляют собой отдельные исполнительные блоки микроопераций. Обратите внимание, что четыре порта планировщика облегчают доступ к исполнительным модулям, которые выполняют вычислительные функции, включая целочисленные, с плавающей запятой и арифметические операции SIMD. Остальные четыре порта поддерживают операции чтения из памяти и сохранения в память.

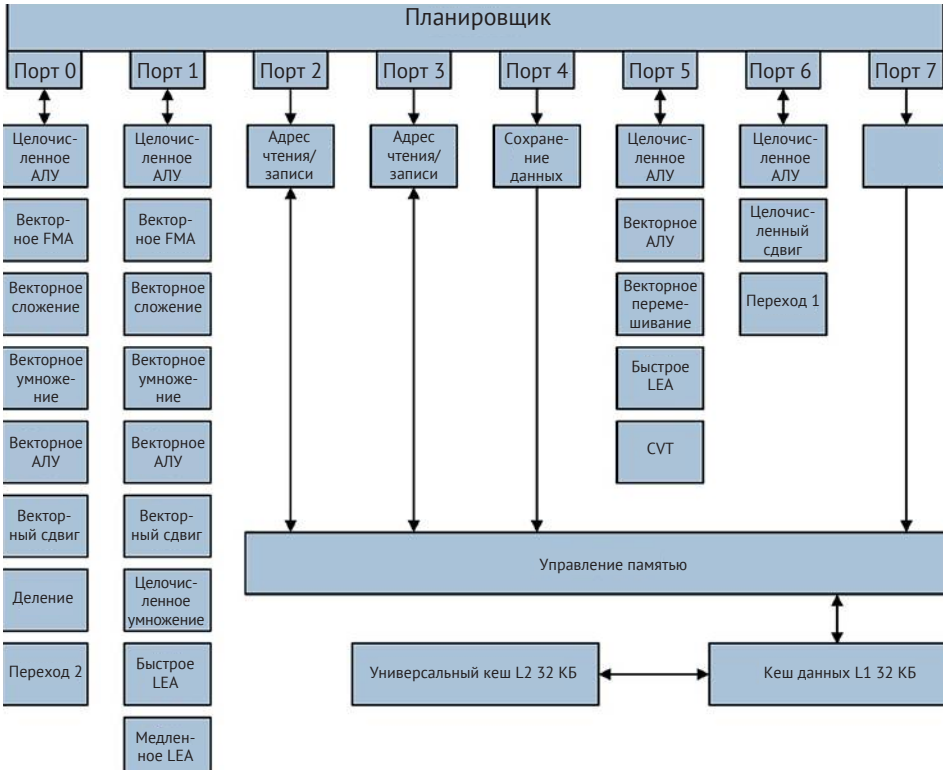


Рис. 15.3. Ядро процессора Skylake и его механизм выполнения

Каждый исполняющий блок выполняет определенное вычисление или операцию. Например, исполняющие блоки *целочисленного АЛУ* (арифметико-логическое устройство) выполняют операции сложения, вычитания и сравнения целых чисел. Исполняющие блоки *векторного АЛУ* обрабатывают целочисленные арифметические операции SIMD и побитовые логические операции. Обратите внимание, что механизм выполнения содержит несколько экземпляров исполняющих блоков. Это позволяет механизму выполнения одновременно выполнять несколько экземпляров определенных микроопераций. Например, механизм выполнения может одновременно выполнять три отдельные побитовые логические операции SIMD, используя исполняющие блоки векторного АЛУ.

Каждый планировщик ядра Skylake может отправлять в механизм выполнения до восьми микроопераций за цикл (по одной на порт). Механизм внеочередного выполнения, который включает в себя планировщик, механизм выполнения и модуль сохранения/удаления, поддерживает до 224 «действующих» (или одновременно существующих) микроопераций. В табл. 15.1 показаны размеры ключевых буферов для последних микроархитектур Intel.

Таблица 15.1. Сравнение размеров ключевых буферов для последних микроархитектур Intel

Параметр	Sandy Bridge, поколение 2	Haswell, поколение 4	Skylake, поколение 6
Порты отправки команд	6	8	8
Буфер микроопераций	168	192	244
Одновременное чтение	64	72	72
Одновременное сохранение	36	42	56
Входы планировщика	54	60	97
Целочисленные регистры	160	168	180
Регистры с плавающей запятой	144	168	168

15.2. Оптимизация кода на языке ассемблера

В этом разделе рассмотрены некоторые базовые стратегии и методы оптимизации, которые можно применять для повышения быстродействия вашего 64-разрядного кода на языке ассемблера x86. Эти методы рекомендуется использовать в коде, предназначенном для последних микроархитектур Intel, включая Skylake Server, Skylake, Haswell и Sandy Bridge. Большинство методов также подходят для использования в коде, который будет выполняться на последних процессорах AMD. Стратегии и методы оптимизации разделены на пять общих категорий:

- основные методы;
- арифметика с плавающей запятой;
- ветвление программы;
- выравнивание данных;
- методы SIMD.

Важно помнить, что методы оптимизации, упомянутые в данном разделе, следует применять осмотрительно. Например, не имеет смысла добавлять в функцию дополнительные команды `push` и `pop` (или другие) лишь для того, чтобы однократно использовать рекомендованную форму команды. Более того, ни одна из стратегий и методов оптимизации, описанных в этом разделе, не исправит неподходящий или плохо спроектированный алгоритм. Справочное руководство по оптимизации архитектур Intel 64 и IA-32 содержит дополнительную информацию о методах оптимизации, обсуждаемых в этом разделе. Приложение также содержит дополнительные ссылки, к которым вы можете обратиться для получения дополнительной информации об оптимизации кода на языке ассемблера x86.

15.2.1. Основные методы

Следующие ниже стратегии и методы кодирования часто используются для повышения производительности кода на языке ассемблера x86-64:

- по возможности используйте команду `test` вместо команды `cmp`, особенно для выполнения простого теста «меньше, равно или больше нуля»;
- избегайте использования команд `cmp` и `test` с непосредственными операндами в памяти (например, `cmp dword ptr [rbp+40],100` или `test byte ptr [r12],0fh`). Вместо этого загрузите значение из памяти в регистр и используйте команду `cmp` или `test` непосредственно с регистром (например, `mov eax,dword ptr [rbp+40]`, за которой следует `cmp eax,100`);
- сведите к минимуму использование команд, которые выполняют частичное обновление флагов состояния в RFLAGS. Например, команды `add eax,1` или `sub rax,1` могут быть быстрее, чем `inc eax` или `dec rax`, особенно в циклах, критичных к производительности (команды `inc` и `dec` не обновляют RFLAGS.CF);
- вместо команды `mov` используйте для обнуления регистра команду `xor` или `sub`. Например, используйте команду `xor eax,eax` или `sub eax,eax` вместо `mov eax,0`. Команду `mov` можно применять, когда необходимо избежать изменения флагов состояния в RFLAGS;
- избегайте использования команд, требующих префикса в размере операнда для загрузки 16-битового непосредственного значения, поскольку команды с префиксом в размере операнда требуют больше времени для декодирования. Вместо этого применяйте эквивалентное 32-битное прямое значение. Например, используйте `mov edx,42` вместо `mov dx,42`;
- по возможности применяйте 32-битные команды вместо 64-битных и регистры общего назначения. Например, если максимальное количество итераций цикла `for` не превышает пределы диапазона 32-битного целого числа, используйте для счетчика цикла 32-битный регистр общего назначения вместо 64-битного;
- используйте 32-битные команды для загрузки в 64-битные регистры положительных постоянных значений. Например, команды `mov eax,16` и `mov r8d,42` эффективно присваивают RAX значение 16 и R8 значение 42;
- используйте двух- или трехоперандную форму команды `imul` для умножения двух целых чисел со знаком, когда не требуется произведение полной ширины. Например, используйте `imul rax,rcx`, когда достаточно 64-битного усеченного произведения, вместо `imul rcx`, которая возвращает 128-битный продукт в RDX:RAX. Это правило также применимо к умножению 32-битных целых чисел со знаком;
- избегайте объявления значений данных внутри раздела кода. В ситуациях, когда это необходимо сделать (например, при определении таблицы переходов только для чтения), поместите данные после безусловной команды `jmp` или `ret`;
- в критических для быстродействия циклах минимизируйте использование команды `lea`, которая содержит три действующих компонента адреса (например, базовый регистр, индексный регистр и смещение). Эти команды могут быть отправлены в блок медленного выполнения LEA только через порт 1. Более короткие формы (один или два действующих

компонента адреса) команды lea могут быть отправлены через порты 1 или 5 одному из блоков быстрого выполнения LEA;

- загружайте в регистр любые значения памяти, необходимые для нескольких вычислений. Если значение в памяти требуется только для одного вычисления, используйте вычисляющую команду в форме регистр–память. В табл. 15.2 показано несколько примеров.

Таблица 15.2. Примеры команд для одно- и многократного использования памяти

Формат «регистр–память» (однократное использование данных)	Формат «регистр–регистр» и move (многократное использование данных)
`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`
`vmulpd xmm0,xmm2,xmmword ptr [rdx]`	`vmovapd xmm1,xmmword ptr [rdx]` `vmulpd xmm0,xmm2,xmm1`

15.2.2. Операции с плавающей запятой

Следующие стратегии и методы кодирования могут использоваться для повышения быстродействия кода ассемблера x86-64, который выполняет операции с плавающей запятой. Эти рекомендации применимы к вычислениям как со скалярными, так и с упакованными операндами с плавающей запятой:

- всегда используйте вычислительные ресурсы x86-AVX для выполнения скалярных арифметических операций с плавающей запятой. Не используйте устаревший модуль с плавающей запятой x87 для выполнения таких вычислений;
- по возможности используйте значения с плавающей запятой одинарной точности вместо значений двойной точности;
- упорядочивайте последовательности команд с плавающей запятой, чтобы минимизировать зависимости регистров. Используйте несколько регистров назначения для сохранения промежуточных результатов, затем уменьшите промежуточные результаты до одного значения (см. пример Ch11_01);
- частично (или полностью) разворачивайте циклы, содержащие вычисления с числами с плавающей запятой, особенно циклы, которые содержат последовательности операций сложения с плавающей запятой, умножения или слитного умножения-сложения;
- по возможности избегайте арифметической потери значимости и денормализованных значений во время арифметических вычислений;
- избегайте использования денормализованных констант с плавающей запятой;
- если ожидаются чрезмерные арифметические потери значимости, рассмотрите возможность включения режимов сброса до нуля (MXCSR.FTZ) и денормализованного значения нуля (MXCSR.DAZ). См. главу 4, в которой рассказано о правильном использовании этих режимов.

15.2.3. Ветвление программы

Команды ветвления программы, такие как `jmp`, `call` и `ret`, потенциально требуют много времени для выполнения, поскольку они могут повлиять на содержимое интерфейсных конвейеров и внутренних кешей. Команда условного перехода `jcc` также является источником проблем с быстродействием, учитывая частоту ее использования. Воспользуйтесь следующими методами оптимизации, чтобы минимизировать неблагоприятное влияние на производительность команд ветвления и повысить точность блока прогнозирования переходов:

- правильно организуйте код, чтобы минимизировать количество возможных команд перехода;
- частично (или полностью) разверните короткие циклы обработки, чтобы минимизировать количество выполняемых команд условного перехода. Избегайте чрезмерного развертывания цикла, поскольку это может привести к более медленному выполнению кода из-за менее эффективного использования детектора петель потока (см. рис. 15.2);
- устраните непредсказуемые ветвления, зависящие от данных, с помощью команд `setcc` или `cmovcc`;
- выровняйте целевые точки ветвлений в критических для быстродействия циклах по 16-байтовым границам;
- перенесите условный код, выполнение которого маловероятно (например, код обработки ошибок), в другой раздел программы (или `.code`) или на страницу памяти.

Блок прогнозирования переходов использует как статические, так и динамические методы для прогнозирования целевой точки команды перехода. Ошибочные предсказания ветвлений можно минимизировать, если блоки кода, содержащие команды условного перехода, расположены так, чтобы они согласовывались с алгоритмом статического предсказания блока прогнозирования переходов:

- используйте условные переходы вперед, когда вероятность выполнения кода передачи управления выше;
- используйте условные переходы в обратном направлении, когда выполнение кода передачи управления менее вероятно.

Метод с условным переходом вперед часто используется в блоках кода, которые выполняют проверку аргументов функции. Метод обратного условного перехода нередко используется в нижней части блока кода цикла после обновления счетчика или другой проверки окончания цикла. Листинг 15.1 содержит короткую функцию на языке ассемблера, которая более подробно иллюстрирует эти методы.

Листинг 15.1. Пример `Ch15_01`

```
;-----------------------------------------------
;              Ch15_01.asm
;-----------------------------------------------

        .const
r8_2p0  real8 2.0

; extern "C" int CalcResult_(double* y, const double* x, size_t n);

        .code
CalcResult_ proc
```

```
; В этом блоке кода используется условный переход вперед, поскольку
; вероятность перехода вперед выше
        test r8,r8
        jz Done                         ;переход, если n == 0

        test r8,7h
        jnz Error                       ;переход, если (n % 8) != 0
        test rcx,1fh
        jnz Error                       ;переход, если y не выровнен по границе 32 бит
        test rdx,1fh
        jnz Error                       ;переход, если x не выровнен по границе 32 бит

; Инициализация
        xor eax,eax                                 ;нулевое смещение массива
        vbroadcastsd ymm5,real8 ptr [r8_2p0]        ;упакованное число 2.0

; Простой цикл обработки массива
        align 16
@@:     vmovapd ymm0,ymmword ptr [rdx+rax]      ;чтение x[i+3]:[i]
        vdivpd ymm1,ymm0,ymm5
        vsqrtpd ymm2,ymm1
        vmovapd ymmword ptr [rcx+rax],ymm2      ;сохранение y[i+3]:y[i]

        vmovapd ymm0,ymmword ptr [rdx+rax+32]   ;чтение x[i+7]:x[i+4]
        vdivpd ymm1,ymm0,ymm5
        vsqrtpd ymm2,ymm1
        vmovapd ymmword ptr [rcx+rax+32],ymm2   ;сохранение y[i+7]:y[i+4]

; Здесь применяется условный переход назад, поскольку
; переход вниз (вперед) менее вероятен
        add rax,64
        sub r8,8
        jnz @B

Done:   xor eax,eax                         ;возвращаемый код успешного выполнения
        vzeroupper
        ret

; Код обработки ошибки, вероятность выполнения которого невелика

Error:  mov eax,1                           ;возвращаемый код ошибки
        ret
CalcResult_ endp
        end
```

15.2.4. Выравнивание данных

Я уже говорил об этом несколько раз (возможно, чересчур назойливо), но важность использования правильно выровненных данных невозможно переоценить. Программы, которые манипулируют неправильно выровненными данными, могут заставить процессор выполнять дополнительные циклы обращения к памяти и лишние микрооперации, что отрицательно влияет на общую производительность системы. Следующие методы выравнивания данных следует рассматривать как универсальные истины и всегда соблюдать их:

- выравнивайте многобайтовые целые числа и значения с плавающей запятой по их естественным границам;
- выравнивайте 128-, 256- и 512-битные упакованные целые числа и значения с плавающей запятой по их надлежащим границам;
- при необходимости дополните структуры данных заполнителями, чтобы обеспечить правильное выравнивание каждого элемента структуры;
- используйте соответствующие спецификаторы языка C++ и библиотечные функции для выравнивания элементов данных, выделенных в высокоуровневом коде. Функции Visual C++ могут использовать спецификатор `alignas(n)` или вызывать `_aligned_malloc` для правильного выравнивания элементов данных;
- отдавайте предпочтение выровненным хранилищам данных.

Также рекомендуется использовать следующие методы организации данных:

- выравнивайте и размещайте небольшие массивы и короткие текстовые строки в структуре данных, чтобы избежать разделения строк кеша. Разделение строки кеша происходит, когда байты многобайтового значения разделяются по 64-байтовой границе. Размещение небольших многобайтовых значений в одной строке кеша помогает минимизировать количество циклов чтения памяти, которые должен выполнять процессор;
- оцените влияние различных строений данных на производительность, например структуры массивов по сравнению с массивом структур.

15.2.5. Методы SIMD

Следующие методы должны применяться, когда это уместно, любой функцией, которая выполняет вычисления SIMD с использованием команд AVX, AVX2 или AVX-512:

- не разрабатывайте функции, которые смешивают команды x86-AVX и x86-SSE. Можно применять функции, которые смешивают команды AVX, AVX2 и AVX-512;
- сведите к минимуму зависимости регистров, чтобы использовать несколько исполняющих модулей в механизме выполнения;
- загружайте в регистр многоразовые операнды из памяти и упакованные константы;
- в системах, поддерживающих AVX-512, используйте дополнительные регистры SIMD, чтобы минимизировать зависимости данных и выпадение регистров. *Выпадение регистра* (register spill) происходит, когда функция вынуждена временно сохранить содержимое регистра в памяти, чтобы освободить регистр для других вычислений;
- для обнуления регистра вместо команды присвоения используйте команды `vpxor, vxorp[d|s]` и т. д. Например, `vxorps xmm0,xmm0` предпочтительнее, чем `vmovaps xmm0,xmmword ptr [XmmZero];`
- используйте маскирование x86-AVX и логические операции, чтобы минимизировать или исключить команды условного перехода, зависящие от данных;
- выполняйте загрузку и сохранение упакованных данных с помощью выровненных команд перемещения (например, `vmovdqa`, `vmovap[d|s]` и т. д.);

- обрабатывайте массивы SIMD с использованием небольших блоков данных для максимального повторного использования данных резидентного кеша;
- при необходимости используйте команду `vzeroupper`, чтобы избежать штрафов за переход состояния с x86-AVX на x86-SSE;
- по возможности используйте формат двойных слов в командах сбора и распределения вместо формата четверных слов (например, используйте `vgatherdp[d|s]` и индексы двойных слов вместо `vgatherqp[d|s]` и индексов четверных слов). Выполняйте все необходимые операции сбора задолго до того, как данные потребуются.

Следующие методы можно использовать для повышения производительности определенных алгоритмов, которые выполняют операции кодирования и декодирования SIMD:

- используйте команды невременного хранения (например, `vmovntdqa`, `vmovntp[d|s]` и т. д.), чтобы минимизировать раздувание кеша;
- используйте команды предварительной выборки данных (например, `prefetcht0`, `prefetchnta` и т. д.), чтобы уведомить процессор об ожидаемых элементах данных.

Глава 16 содержит несколько примеров исходного кода, которые иллюстрируют использование невременного хранилища и команды предварительной выборки данных.

15.3. Заключение

В главе 15 были рассмотрены некоторые ключевые моменты:

- быстродействие большинства функций на языке ассемблера можно улучшить, реализовав стратегии и методы оптимизации, описанные в этой главе;
- рекомендованные методы оптимизации должны применяться разумно. В программировании нередко встречаются ситуации, когда рекомендованная стратегия или методика – не лучший подход;
- для достижения оптимальной производительности конкретного алгоритма или функции может потребоваться разработка нескольких версий кода и сравнение результатов измерения быстродействия на одинаковых примерах;
- при разработке кода на языке ассемблера не тратьте слишком много времени на попытки добиться максимального быстродействия. Сосредоточьтесь на приросте быстродействия, которого относительно легко добиться (например, реализация алгоритма с использованием SIMD вместо скалярной арифметики);
- ни одна из стратегий и методов оптимизации, представленных в этой главе, не улучшит неподходящий или плохо спроектированный алгоритм.

Глава 16

Продвинутое программирование

В последней главе этой книги приведен обзор нескольких примеров исходного кода, демонстрирующих передовые методы программирования на языке ассемблера x86. В первом примере показано, как использовать команду `cpuid` для обнаружения определенных расширений набора команд x86. Далее следуют два примера, которые иллюстрируют, как ускорить функции обработки SIMD с помощью команд предварительной выборки данных из редко запрашиваемых данных в памяти. В заключительном примере показано использование функции вычисления на языке ассемблера в многопоточном приложении.

16.1. Команда CPUID

В этой книге несколько раз упоминалось, что прикладная программа никогда не должна предполагать, что конкретное расширение набора команд, такое как AVX, AVX2 или AVX-512, доступно, просто исходя из микроархитектуры процессора, номера модели или торговой марки. Прикладная программа всегда должна проверять наличие расширения набора команд с помощью команды `cpuid` (идентификация ЦП). Прикладные программы могут использовать эту команду, чтобы убедиться, что процессор поддерживает одно из ранее упомянутых расширений набора команд x86-AVX. Команду `cpuid` можно также использовать для получения дополнительной информации о функциях процессора, которая полезна или необходима как в прикладных программах, так и в программном обеспечении операционной системы.

В листинге 16.1 показан исходный код примера `Ch16_01`. Этот пример демонстрирует, как использовать команду `cupid` для определения поддержки процессором различных расширений набора команд. Исходный код примера `Ch16_01` фокусируется на использовании `cpuid` для обнаружения архитектурных функций и расширений набора команд, связанных с содержанием данной книги. Если вам интересно узнать, как использовать команду `cpuid` для определения других функций процессора, вам следует обратиться к справочным руководствам AMD и Intel, которые перечислены в приложении.

Листинг 16.1. Пример Ch16_01

```
//-----------------------------------------------
//              CpuidInfo.h
//-----------------------------------------------

#pragma once
#include <cstdint>
#include <vector>
#include <string>

struct CpuidRegs
{
    uint32_t EAX;
    uint32_t EBX;
    uint32_t ECX;
    uint32_t EDX;
};

class CpuidInfo
{
public:
    class CacheInfo
    {
    public:
        enum class Type
        {
            Unknown, Data, Instruction, Unified
        };

    private:
        uint32_t m_Level = 0;
        Type m_Type = Type::Unknown;
        uint32_t m_Size = 0;

    public:
        uint32_t GetLevel(void) const             { return m_Level; }
        uint32_t GetSize(void) const              { return m_Size; }
        Type GetType(void) const                  { return m_Type; }

        // Эта функция определена в CacheInfo.cpp
        CacheInfo(uint32_t level, uint32_t type, uint32_t size);
        std::string GetTypeString(void) const;
    };

private:
    uint32_t m_MaxEax;                              // Макс. EAX для базового CPUID
    uint32_t m_MaxEaxExt;                           // Макс. EAX расширенного CPUID
    uint64_t m_FeatureFlags;                        // Флаги свойств процессора
    std::vector<CpuidInfo::CacheInfo> m_CacheInfo;  // Информация кеша процессора
    char m_VendorId[13];                            // Строка идентификатора процессора
    char m_ProcessorBrand[49];                      // Строка названия производителя
    bool m_OsXsave;                                 // XSAVE включен для нужд приложения
    bool m_OsAvxState;                              // AVX включен операционной системой
    bool m_OsAvx512State;                           // AVX-512 включен операционной
системой
```

```
    void Init(void);
    void InitProcessorBrand(void);
    void LoadInfo0(void);
    void LoadInfo1(void);
    void LoadInfo2(void);
    void LoadInfo3(void);
    void LoadInfo4(void);
    void LoadInfo5(void);

public:
    enum class FF : uint64_t
    {
        FXSR               = (uint64_t)1 << 0,
        MMX                = (uint64_t)1 << 1,
        MOVBE              = (uint64_t)1 << 2,
        SSE                = (uint64_t)1 << 3,
        SSE2               = (uint64_t)1 << 4,
        SSE3               = (uint64_t)1 << 5,
        SSSE3              = (uint64_t)1 << 6,
        SSE4_1             = (uint64_t)1 << 7,
        SSE4_2             = (uint64_t)1 << 8,
        PCLMULQDQ          = (uint64_t)1 << 9,
        POPCNT             = (uint64_t)1 << 10,
        PREFETCHW          = (uint64_t)1 << 11,
        PREFETCHWT1        = (uint64_t)1 << 12,
        RDRAND             = (uint64_t)1 << 13,
        RDSEED             = (uint64_t)1 << 14,
        ERMSB              = (uint64_t)1 << 15,
        AVX                = (uint64_t)1 << 16,
        AVX2               = (uint64_t)1 << 17,
        F16C               = (uint64_t)1 << 18,
        FMA                = (uint64_t)1 << 19,
        BMI1               = (uint64_t)1 << 20,
        BMI2               = (uint64_t)1 << 21,
        LZCNT              = (uint64_t)1 << 22,
        ADX                = (uint64_t)1 << 23,
        AVX512F            = (uint64_t)1 << 24,
        AVX512ER           = (uint64_t)1 << 25,
        AVX512PF           = (uint64_t)1 << 26,
        AVX512DQ           = (uint64_t)1 << 27,
        AVX512CD           = (uint64_t)1 << 28,
        AVX512BW           = (uint64_t)1 << 29,
        AVX512VL           = (uint64_t)1 << 30,
        AVX512_IFMA        = (uint64_t)1 << 31,
        AVX512_VBMI        = (uint64_t)1 << 32,
        AVX512_4FMAPS      = (uint64_t)1 << 33,
        AVX512_4VNNIW      = (uint64_t)1 << 34,
        AVX512_VPOPCNTDQ   = (uint64_t)1 << 35,
        AVX512_VNNI        = (uint64_t)1 << 36,
        AVX512_VBMI2       = (uint64_t)1 << 37,
        AVX512_BITALG      = (uint64_t)1 << 38,
        CLWB               = (uint64_t)1 << 39,
    };

    CpuidInfo(void) { Init(); };
    ~CpuidInfo() {};
```

```
    const std::vector<CpuidInfo::CacheInfo>& GetCacheInfo(void) const
    {
        return m_CacheInfo;
    }

    bool GetFF(FF flag) const
    {
        return ((m_FeatureFlags & (uint64_t)flag) != 0) ? true : false;
    }

    std::string GetProcessorBrand(void) const   { return std::string(m_ProcessorBrand); }
    std::string GetVendorId(void) const         { return std::string(m_VendorId); }

    void LoadInfo(void);
};

// Cpuinfo_.asm
extern "C" void Xgetbv_(uint32_t r_ecx, uint32_t* r_eax, uint32_t* r_edx);
extern "C" uint32_t Cpuid_(uint32_t r_eax, uint32_t r_ecx, CpuidRegs* r_out);
```

```
;-----------------------------------------------
;               CpuidInfo_.asm
;-----------------------------------------------

; Следующая структура должна совпадать со структурой CpuidRegs,
; которая объявлена в CpuidInfo.h

CpuidRegs   struct
RegEAX      dword ?
RegEBX      dword ?
RegECX      dword ?
RegEDX      dword ?
CpuidRegs   ends

; extern "C" uint32_t Cpuid_(uint32_t r_eax, uint32_t r_ecx, CpuidRegs* r_out);
;
; Возвращает:   eax == 0     неподдерживаемая ветка CPUID
;               eax != 0     поддерживаемая ветка CPUID
;
; Примечание: возвращаемый код действителен, только если r_eax <= MaxEAX

        .code
Cpuid_  proc frame
        push rbx
        .pushreg rbx
        .endprolog

; Чтение eax и ecx
        mov eax,ecx
        mov ecx,edx

; Получение информации cpuid и сохранение результатов
        cpuid
        mov dword ptr [r8+CpuidRegs.RegEAX],eax
        mov dword ptr [r8+CpuidRegs.RegEBX],ebx
        mov dword ptr [r8+CpuidRegs.RegECX],ecx
        mov dword ptr [r8+CpuidRegs.RegEDX],edx
```

```
; Тест неподдерживаемой ветки cpuid
        or eax,ebx
        or ecx,edx
        or eax,ecx                          ;eax = возвращаемый код

        pop rbx
        ret
Cpuid_  endp

; extern "C" void Xgetbv_(uint32_t r_ecx, uint32_t* r_eax, uint32_t* r_edx);

Xgetbv_ proc
        mov r9,rdx                          ;r9 = r_eax ptr
        xgetbv

        mov dword ptr [r9],eax              ;сохранение младшего слова результата
        mov dword ptr [r8],edx              ;сохранение старшего слова результата
        ret
Xgetbv_ endp
        end
```

```
//-----------------------------------------------
//              Ch16_01.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <string>
#include "CpuidInfo.h"

using namespace std;

static void DisplayCacheInfo(const CpuidInfo& ci);
static void DisplayFeatureFlags(const CpuidInfo& ci);

int main()
{
    CpuidInfo ci;

    ci.LoadInfo();

    cout << ci.GetVendorId() << '\n';
    cout << ci.GetProcessorBrand() << '\n';

    DisplayCacheInfo(ci);
    DisplayFeatureFlags(ci);
    return 0;
}

static void DisplayCacheInfo(const CpuidInfo& ci)
{
    const vector<CpuidInfo::CacheInfo>& cache_info = ci.GetCacheInfo();

    for (const CpuidInfo::CacheInfo& x : cache_info)
    {
```

```
        uint32_t cache_size = x.GetSize();
        string cache_size_str;

        if (cache_size < 1024 * 1024)
        {
            cache_size /= 1024;
            cache_size_str = "KB";
        }
        else
        {
            cache_size /= 1024 * 1024;
            cache_size_str = "MB";
        }

        cout << "Cache L" << x.GetLevel() << ": ";
        cout << cache_size << cache_size_str << ' ';
        cout << x.GetTypeString() << '\n';
    }
}

static void DisplayFeatureFlags(const CpuidInfo& ci)
{
    const char nl = '\n';

    cout << "----- Флаги свойств CPUID -----" << nl;
    cout << "ADX:          " << ci.GetFF(CpuidInfo::FF::ADX) << nl;
    cout << "AVX:          " << ci.GetFF(CpuidInfo::FF::AVX) << nl;
    cout << "AVX2:         " << ci.GetFF(CpuidInfo::FF::AVX2) << nl;
    cout << "AVX512F:      " << ci.GetFF(CpuidInfo::FF::AVX512F) << nl;
    cout << "AVX512BW:     " << ci.GetFF(CpuidInfo::FF::AVX512BW) << nl;
    cout << "AVX512CD:     " << ci.GetFF(CpuidInfo::FF::AVX512CD) << nl;
    cout << "AVX512DQ:     " << ci.GetFF(CpuidInfo::FF::AVX512DQ) << nl;
    cout << "AVX512ER:     " << ci.GetFF(CpuidInfo::FF::AVX512ER) << nl;
    cout << "AVX512PF:     " << ci.GetFF(CpuidInfo::FF::AVX512PF) << nl;
    cout << "AVX512VL:     " << ci.GetFF(CpuidInfo::FF::AVX512VL) << nl;
    cout << "AVX512_IFMA:  " << ci.GetFF(CpuidInfo::FF::AVX512_IFMA) << nl;
    cout << "AVX512_VBMI:  " << ci.GetFF(CpuidInfo::FF::AVX512_VBMI) << nl;
    cout << "BMI1:         " << ci.GetFF(CpuidInfo::FF::BMI1) << nl;
    cout << "BMI2:         " << ci.GetFF(CpuidInfo::FF::BMI2) << nl;
    cout << "F16C:         " << ci.GetFF(CpuidInfo::FF::F16C) << nl;
    cout << "FMA:          " << ci.GetFF(CpuidInfo::FF::FMA) << nl;
    cout << "LZCNT:        " << ci.GetFF(CpuidInfo::FF::LZCNT) << nl;
    cout << "POPCNT:       " << ci.GetFF(CpuidInfo::FF::POPCNT) << nl;
}

//-----------------------------------------------
//              CpuidInfo.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <string>
#include <cstring>
#include <vector>
#include "CpuidInfo.h"

using namespace std;
```

```
void CpuidInfo::LoadInfo(void)
{
    // Примечание: LoadInfo0 следует вызывать первой
    LoadInfo0();
    LoadInfo1();
    LoadInfo2();
    LoadInfo3();
    LoadInfo4();
    LoadInfo5();
}

void CpuidInfo::LoadInfo0(void)
{
    CpuidRegs r1;

    // Начальная инициализация
    Init();

    // Получение MaxEax и VendorID
    Cpuid_(0, 0, &r1);
    m_MaxEax = r1.EAX;
    *(uint32_t *)(m_VendorId + 0) = r1.EBX;
    *(uint32_t *)(m_VendorId + 4) = r1.EDX;
    *(uint32_t *)(m_VendorId + 8) = r1.ECX;
    m_VendorId[sizeof(m_VendorId) - 1] = '\0';

    // Получение MaxEaxExt
    Cpuid_(0x80000000, 0, &r1);
    m_MaxEaxExt = r1.EAX;

    // Инициализация строки марки процессора
    InitProcessorBrand();
}

void CpuidInfo::LoadInfo1(void)
{
    CpuidRegs r;

    if (m_MaxEax < 1)
        return;

    Cpuid_(1, 0, &r);

    //
    // Декодирование флагов r.ECX
    //

    // CPUID.(EAX=01H, ECX=00H):ECX.SSE3[bit 0]
    if (r.ECX & (0x1 << 0))
        m_FeatureFlags |= (uint64_t)FF::SSE3;

    // CPUID.(EAX=01H, ECX=00H):ECX.PCLMULQDQ[bit 1]
    if (r.ECX & (0x1 << 1))
        m_FeatureFlags |= (uint64_t)FF::PCLMULQDQ;

    // CPUID.(EAX=01H, ECX=00H):ECX.SSSE3[bit 9]
```

```
    if (r.ECX & (0x1 << 9))
        m_FeatureFlags |= (uint64_t)FF::SSSE3;

    // CPUID.(EAX=01H, ECX=00H):ECX.SSE4.1[bit 19]
    if (r.ECX & (0x1 << 19))
        m_FeatureFlags |= (uint64_t)FF::SSE4_1;

    // CPUID.(EAX=01H, ECX=00H):ECX.SSE4.2[bit 20]
    if (r.ECX & (0x1 << 20))
        m_FeatureFlags |= (uint64_t)FF::SSE4_2;

    // CPUID.(EAX=01H, ECX=00H):ECX.MOVBE[bit 22]
    if (r.ECX & (0x1 << 22))
        m_FeatureFlags |= (uint64_t)FF::MOVBE;

    // CPUID.(EAX=01H, ECX=00H):ECX.POPCNT[bit 23]
    if (r.ECX & (0x1 << 23))
        m_FeatureFlags |= (uint64_t)FF::POPCNT;

    // CPUID.(EAX=01H, ECX=00H):ECX.RDRAND[bit 30]
    if (r.ECX & (0x1 << 30))
        m_FeatureFlags |= (uint64_t)FF::RDRAND;

    //
    // Декодирование флагов r.RDX
    //

    // CPUID.(EAX=01H, ECX=00H):EDX.MMX[bit 23]
    if (r.EDX & (0x1 << 23))
        m_FeatureFlags |= (uint64_t)FF::MMX;

    // CPUID.(EAX=01H, ECX=00H):EDX.FXSR[bit 24]
    if (r.EDX & (0x1 << 24))
        m_FeatureFlags |= (uint64_t)FF::FXSR;

    // CPUID.(EAX=01H, ECX=00H):EDX.SSE[bit 25]
    if (r.EDX & (0x1 << 25))
        m_FeatureFlags |= (uint64_t)FF::SSE;

    // CPUID.(EAX=01H, ECX=00H):EDX.SSE2[bit 26]
    if (r.EDX & (0x1 << 26))
        m_FeatureFlags |= (uint64_t)FF::SSE2;
}

void CpuidInfo::LoadInfo2(void)
{
    CpuidRegs   r;

    if (m_MaxEax < 7)
        return;

    Cpuid_(7, 0, &r);

    // CPUID.(EAX=07H, ECX=00H):ECX.PREFETCHWT1[bit 0]
    if (r.ECX & (0x1 << 0))
        m_FeatureFlags |= (uint64_t)FF::PREFETCHWT1;
```

```
    // CPUID.(EAX=07H, ECX=00H):EBX.BMI1[bit 3]
    if (r.EBX & (0x1 << 3))
        m_FeatureFlags |= (uint64_t)FF::BMI1;

    // CPUID.(EAX=07H, ECX=00H):EBX.BMI2[bit 8]
    if (r.EBX & (0x1 << 8))
        m_FeatureFlags |= (uint64_t)FF::BMI2;

    // CPUID.(EAX=07H, ECX=00H):EBX.ERMSB[bit 9]
    // ERMSB = Enhanced REP MOVSB/STOSB
    if (r.EBX & (0x1 << 9))
        m_FeatureFlags |= (uint64_t)FF::ERMSB;

    // CPUID.(EAX=07H, ECX=00H):EBX.RDSEED[bit 18]
    if (r.EBX & (0x1 << 18))
        m_FeatureFlags |= (uint64_t)FF::RDSEED;

    // CPUID.(EAX=07H, ECX=00H):EBX.ADX[bit 19]
    if (r.EBX & (0x1 << 19))
        m_FeatureFlags |= (uint64_t)FF::ADX;

    // CPUID.(EAX=07H, ECX=00H):EBX.CLWB[bit 24]
    if (r.EBX & (0x1 << 24))
        m_FeatureFlags |= (uint64_t)FF::CLWB;
}

void CpuidInfo::LoadInfo3(void)
{
    CpuidRegs r;

    if (m_MaxEaxExt < 0x80000001)
        return;

    Cpuid_(0x80000001, 0, &r);

    // CPUID.(EAX=80000001H, ECX=00H):ECX.LZCNT[bit 5]
    if (r.ECX & (0x1 << 5))
        m_FeatureFlags |= (uint64_t)FF::LZCNT;

    // CPUID.(EAX=80000001H, ECX=00H):ECX.PREFETCHW[bit 8]
    if (r.ECX & (0x1 << 8))
        m_FeatureFlags |= (uint64_t)FF::PREFETCHW;
}

void CpuidInfo::LoadInfo4(void)
{
    CpuidRegs r_eax01h;
    CpuidRegs r_eax07h;

    if (m_MaxEax < 7)
        return;

    Cpuid_(1, 0, &r_eax01h);
    Cpuid_(7, 0, &r_eax07h);

    // Тест CPUID.(EAX=01H, ECX=00H):ECX.OSXSAVE[bit 27]
```

```
// для проверки возможности использования XGETBV
m_OsXsave = (r_eax01h.ECX & (0x1 << 27)) ? true : false;

if (m_OsXsave)
{
    // Применение XGETBV для получения следующей информации
    // AVX включен средствами ОС, если (XCR0[2:1] == '11b') истинно
    // AVX512 включен средствами ОС, если (XCR0[7:5] == '111b') истинно

    uint32_t xgetbv_eax, xgetbv_edx;

    Xgetbv_(0, &xgetbv_eax, &xgetbv_edx);
    m_OsAvxState = (((xgetbv_eax >> 1) & 0x03) == 0x03) ? true : false;

    if (m_OsAvxState)
    {
        // CPUID.(EAX=01H, ECX=00H):ECX.AVX[bit 28]
        if (r_eax01h.ECX & (0x1 << 28))
        {
            m_FeatureFlags |= (uint64_t)FF::AVX;

            // CPUID.(EAX=01H, ECX=00H):ECX.FMA[bit 12]
            if (r_eax01h.ECX & (0x1 << 12))
                m_FeatureFlags |= (uint64_t)FF::FMA;

            // CPUID.(EAX=01H, ECX=00H):ECX.F16C[bit 29]
            if (r_eax01h.ECX & (0x1 << 29))
                m_FeatureFlags |= (uint64_t)FF::F16C;

            // CPUID.(EAX=07H, ECX=00H):EBX.AVX2[bit 5]
            if (r_eax07h.EBX & (0x1 << 5))
                m_FeatureFlags |= (uint64_t)FF::AVX2;

            m_OsAvx512State = (((xgetbv_eax >> 5) & 0x07) == 0x07) ? true : false;

            if (m_OsAvx512State)
            {
                // CPUID.(EAX=07H, ECX=00H):EBX.AVX512F[bit 16]
                if (r_eax07h.EBX & (0x1 << 16))
                {
                    m_FeatureFlags |= (uint64_t)FF::AVX512F;

                    //
                    // Декодирование флагов EBX
                    //

                    // CPUID.(EAX=07H, ECX=00H):EBX.AVX512DQ[bit 17]
                    if (r_eax07h.EBX & (0x1 << 17))
                        m_FeatureFlags |= (uint64_t)FF::AVX512DQ;

                    // CPUID.(EAX=07H, ECX=00H):EBX.AVX512_IFMA[bit 21]
                    if (r_eax07h.EBX & (0x1 << 21))
                        m_FeatureFlags |= (uint64_t)FF::AVX512_IFMA;

                    // CPUID.(EAX=07H, ECX=00H):EBX.AVX512PF[bit 26]
                    if (r_eax07h.EBX & (0x1 << 26))
                        m_FeatureFlags |= (uint64_t)FF::AVX512PF;
```

```
                        // CPUID.(EAX=07H, ECX=00H):EBX.AVX512ER[bit 27]
                        if (r_eax07h.EBX & (0x1 << 27))
                            m_FeatureFlags |= (uint64_t)FF::AVX512ER;

                        // CPUID.(EAX=07H, ECX=00H):EBX.AVX512CD[bit 28]
                        if (r_eax07h.EBX & (0x1 << 28))
                            m_FeatureFlags |= (uint64_t)FF::AVX512CD;

                        // CPUID.(EAX=07H, ECX=00H):EBX.AVX512BW[bit 30]
                        if (r_eax07h.EBX & (0x1 << 30))
                            m_FeatureFlags |= (uint64_t)FF::AVX512BW;

                        // CPUID.(EAX=07H, ECX=00H):EBX.AVX512VL[bit 31]
                        if (r_eax07h.EBX & (0x1 << 31))
                            m_FeatureFlags |= (uint64_t)FF::AVX512VL;

                        //
                        // Декодирование флагов ECX
                        //

                        // CPUID.(EAX=07H, ECX=00H):ECX.AVX512_VBMI[bit 1]
                        if (r_eax07h.ECX & (0x1 << 1))
                            m_FeatureFlags |= (uint64_t)FF::AVX512_VBMI;

                        // CPUID.(EAX=07H, ECX=00H):ECX.AVX512_VBMI2[bit 6]
                        if (r_eax07h.ECX & (0x1 << 6))
                            m_FeatureFlags |= (uint64_t)FF::AVX512_VBMI2;

                        // CPUID.(EAX=07H, ECX=00H):ECX.AVX512_VNNI[bit 11]
                        if (r_eax07h.ECX & (0x1 << 11))
                            m_FeatureFlags |= (uint64_t)FF::AVX512_VNNI;

                        // CPUID.(EAX=07H, ECX=00H):ECX.AVX512_BITALG[bit 12]
                        if (r_eax07h.ECX & (0x1 << 12))
                            m_FeatureFlags |= (uint64_t)FF::AVX512_BITALG;

                        // CPUID.(EAX=07H, ECX=00H):ECX.AVX512_VPOPCNTDQ[bit 14]
                        if (r_eax07h.ECX & (0x1 << 14))
                            m_FeatureFlags |= (uint64_t)FF::AVX512_VPOPCNTDQ;

                        //
                        // Декодирование флагов EDX
                        //

                        // CPUID.(EAX=07H, ECX=00H):EDX.AVX512_4FMAPS[bit 2]
                        if (r_eax07h.EDX & (0x1 << 2))
                            m_FeatureFlags |= (uint64_t)FF::AVX512_4FMAPS;

                        // CPUID.(EAX=07H, ECX=00H):EDX.AVX512_4VNNIW[bit 3]
                        if (r_eax07h.EDX & (0x1 << 3))
                            m_FeatureFlags |= (uint64_t)FF::AVX512_4VNNIW;
                    }
                }
            }
        }
    }
}
```

```
void CpuidInfo::LoadInfo5(void)
{
    if (m_MaxEax < 4)
        return;

    bool done = false;
    uint32_t index = 0;

    while (!done)
    {
        CpuidRegs r;

        Cpuid_(4, index, &r);

        uint32_t cache_type = r.EAX & 0x1f;
        uint32_t cache_level = ((r.EAX >> 5) & 0x3);

        if (cache_type == 0)
            done = true;
        else
        {
            uint32_t ways = ((r.EBX >> 22) & 0x3ff) + 1;
            uint32_t partitions = ((r.EBX >> 12) & 0x3ff) + 1;
            uint32_t line_size = (r.EBX & 0xfff) + 1;
            uint32_t sets = r.ECX + 1;
            uint32_t cache_size = ways * partitions * line_size * sets;

            CacheInfo ci(cache_level, cache_type, cache_size);
            m_CacheInfo.push_back(ci);
            index++;
        }
    }
}

void CpuidInfo::Init(void)
{
    m_MaxEax = 0;
    m_MaxEaxExt = 0;
    m_FeatureFlags = 0;
    m_OsXsave = false;
    m_OsAvxState = false;
    m_OsAvx512State = false;
    m_VendorId[0] = '\0';
    m_ProcessorBrand[0] = '\0';
    m_CacheInfo.clear();
}

void CpuidInfo::InitProcessorBrand(void)
{
    if (m_MaxEaxExt >= 0x80000004)
    {
        CpuidRegs r2, r3, r4;
        char* p = m_ProcessorBrand;

        Cpuid_(0x80000002, 0, &r2);
        Cpuid_(0x80000003, 0, &r3);
        Cpuid_(0x80000004, 0, &r4);
```

```
        *(uint32_t *)(p + 0) = r2.EAX;
        *(uint32_t *)(p + 4) = r2.EBX;
        *(uint32_t *)(p + 8) = r2.ECX;
        *(uint32_t *)(p + 12) = r2.EDX;
        *(uint32_t *)(p + 16) = r3.EAX;
        *(uint32_t *)(p + 20) = r3.EBX;
        *(uint32_t *)(p + 24) = r3.ECX;
        *(uint32_t *)(p + 28) = r3.EDX;
        *(uint32_t *)(p + 32) = r4.EAX;
        *(uint32_t *)(p + 36) = r4.EBX;
        *(uint32_t *)(p + 40) = r4.ECX;
        *(uint32_t *)(p + 44) = r4.EDX;

        m_ProcessorBrand[sizeof(m_ProcessorBrand) - 1] = '\0';
    }
    else
        strcpy_s(m_ProcessorBrand, "Неизвестен");
}
```

Перед изучением исходного кода будет полезно получить базовое представление о команде `cpuid` и о том, как она работает. Перед использованием `cpuid` функция должна загрузить в регистр EAX «листовое» (leaf) значение, которое указывает, какую информацию должна возвращать команда `cpuid`. В регистре ECX также может потребоваться второе, или «подлистовое» (sub-leaf), значение. Команда `cpuid` возвращает свои результаты в регистрах EAX, EBX, ECX и EDX. Затем вызывающая функция должна декодировать значения в этих регистрах, чтобы убедиться, что процессор поддерживает определенные функции. Как вы скоро увидите, программе часто бывает необходимо использовать команду `cpuid` несколько раз. Большинство прикладных программ обычно используют `cpuid` во время инициализации и сохраняют результаты для дальнейшего использования. Причина этого в том, что `cpuid` является последовательной командой, то есть она заставляет процессор завершить выполнение всех ранее выбранных команд и выполнить любые ожидающие записи в память перед выборкой следующей команды. Другими словами, выполнение команды `cpuid` занимает много времени.

Листинг 16.1 начинается с заголовочного файла `CpuidInfo.h`. В верхней части этого файла находится структура с именем `CpuidRegs`, которая применяется для сохранения результатов, возвращаемых `cpuid`. Далее следует класс C++ с именем `CpuidInfo`. Этот класс содержит код и данные, связанные с использованием команд `cpuid`. Открытая часть `CpuidInfo` включает подкласс с именем `CacheInfo`. Этот класс используется для выдачи информации о кешах памяти процессора. Класс `CpuidInfo` также включает перечисление с именем `FF`. Прикладная программа может использовать это перечисление в качестве значения аргумента с функцией-членом `CpuidInfo::GetFF`, чтобы определить, поддерживает ли хост-процессор определенный набор команд. Позже вы увидите, как это работает. В нижней части заголовочного файла `CpuidInfo.h` находятся два оператора объявления для функций на языке ассемблера `Cpuid_` и `Xgetbv_`. Эти функции выполняют команды `cpuid` и `xgetbv` (получение значения расширенного регистра управления) соответственно.

После `CpuidInfo.h` в листинге 16.1 идет файл исходного кода `CpuidInfo_.asm`. Этот файл содержит функции на языке ассемблера `Cpuid_` и `Xgetbv_`, которые являются

простыми функциями-оболочками для команд x86 cupid и xgetbv. Функция Cpuid_ начинает свое выполнение с сохранения регистра RBX в стеке. Затем она загружает значения аргументов r_eax и r_ecx в регистры EAX и ECX. Команда cpuid следует сразу за загрузкой регистров EAX и ECX. После выполнения cpuid результаты, размещенные в регистрах EAX, EBX, ECX и EDX, сохраняются в упомянутой выше структуре CpuidRegs. Функция на языке ассемблера Xgetbv_ выполняет команду xgetbv. Эта команда загружает содержимое расширенного регистра управления процессором, указанного в ECX, в регистровую пару EDX:EAX. Команда xgetbv позволяет прикладной программе определять, поддерживает ли операционная система хоста AVX, AVX2 или AVX-512, как описано далее в этом разделе.

Следующий файл в листинге 16.1 – Ch16_01.cpp. Функция main содержит код, демонстрирующий использование класса C++ CpuidInfo. Оператор ci.LoadInfo() вызывает функцию-член CpuidInfo::LoadInfo, которая выполняет несколько запусков cpuid для получения информации о процессоре. Обратите внимание, что CpuidInfo::LoadInfo вызывается только один раз. Функция DisplayCacheInfo передает информацию о кешах памяти процессора в cout. Эта функция вызывает CpuidInfo::GetCacheInfo, чтобы извлечь информацию о кеше, полученную во время выполнения CpuidInfo::LoadInfo. Функция DisplayFeatureFlags показывает информацию о некоторых расширениях набора команд, которые поддерживает процессор. Каждый оператор cout в этой функции использует CpuidInfo::GetFF с другим значением CpuidInfo::FF. Функция-член CpuidInfo::GetFF возвращает одно логическое значение, которое указывает, поддерживает ли процессор расширение набора команд, указанное в значении ее аргумента. Как и данные о кеше, информация о расширении набора команд процессора была получена и сохранена во время вызова CpuidInfo::LoadInfo. Обратите внимание, что структура CpuidInfo позволяет прикладной программе выполнять несколько вызовов CpuidInfo::GetFF без дополнительных запусков cpuid.

За файлом Ch16_01.cpp в листинге 16.1 следует исходный код для CpuidInfo.cpp, который содержит нетривиальные функции-члены для класса CpuidInfo. Функция-член CpuidInfo::LoadInfo, о которой говорилось ранее, вызывает шесть частных функций-членов, которые выполняют множество запросов cupid. Первая из этих функций, CpuidInfo::LoadInfo0, начинает свое выполнение с вызова CpuidInfo::Init для выполнения необходимых инициализаций. Затем она вызывает функцию на языке ассемблера Cpuid_ для получения максимального конечного значения cpuid, которое поддерживается процессором, и строки идентификатора поставщика процессора. Потом следует другой вызов Cpuid_, чтобы получить максимальное значение листа для расширенной информации о cpuid. Далее следует вызов CpuidInfo::InitProcessorBrand, которая использует несколько вызовов Cpuid_ для запроса и сохранения строки наименования производителя процессора. Исходный код этой функции находится в конце файла CpuidInfo.cpp.

Функции-члены CpuidInfo::LoadInfo1, CpuidInfo::LoadInfo2 и CpuidInfo::LoadInfo3 также используют Cpuid_, чтобы убедиться, что процессор поддерживает различные расширения набора команд. Код, содержащийся в этих функциях-членах, в основном представляет собой расшифровку «в лоб» различных результатов Cpuid_. Справочные руководства по программированию AMD и Intel содержат дополнительную информацию о битах флага функции cpuid, которые обозначают поддержку процессором определенного расширения набора команд. Закрытая функция-член

CpuidInfo::LoadInfo4 содержит код, проверяющий AVX, AVX2 и AVX-512. Эта функция-член требует более внимательного изучения.

Прикладная программа может использовать вычислительные ресурсы x86-AVX только в том случае, если они поддерживаются как процессором, так и операционной системой хоста. Функцию Xgetbv_ можно использовать для определения поддержки операционной системы хоста. Перед использованием Xgetbv_ необходимо протестировать флаг OSXSAVE cpuid, чтобы убедиться, что использование команды xgetbv безопасно для прикладной программы. Если OSXSAVE имеет значение true, функция CpuidInfo::LoadInfo4 вызывает функцию Xgetbv_ для получения информации о поддержке со стороны ОС информации о состоянии x86-AVX (т. е. о том, правильно ли ОС сохраняет регистры XMM, YMM и ZMM во время переключения задачи). При использовании функции Xgetbv_ процессор генерирует исключение, если номер расширенного регистра управления недействителен или если флаг OSXSAVE процессора установлен в значение false. Это объясняет, почему перед вызовом Xgetbv_ необходимо проверить программный флаг m_OsXsave. Если операционная система хоста поддерживает информацию о состоянии x86-AVX, функция CpuidInfo::LoadInfo4 продолжает декодировать флаги функций cpuid, относящиеся к AVX и AVX2. Обратите внимание, что здесь также проверяются флаги функций FMA и F16C. Остающийся код в CpuidInfo::LoadInfo4 декодирует флаги функций cupid, которые означают поддержку различных расширений набора команд AVX-512.

Последняя закрытая функция-член, вызываемая CpuidInfo::LoadInfo, называется CpuidInfo::LoadInfo5. Эта функция-член использует cpuid и класс CpuidInfo::CacheInfo для сохранения информации о типе и размере кешей памяти процессора. Вспомогательный код для класса CpuidInfo::CacheInfo не показан в листинге 16.1, но включен в пакет файлов для скачивания. Результат выполнения кода примера Ch16_01 выглядит так:

```
GenuineIntel
Intel(R) Core(TM) i9-7900X CPU @ 3.30GHz
Cache L1: 32KB Data
Cache L1: 32KB Instruction
Cache L2: 1MB Unified
Cache L3: 13MB Unified
----- Флаги свойств CPUID -----
ADX: 1
AVX: 1
AVX2: 1
AVX512F: 1
AVX512BW: 1
AVX512CD: 1
AVX512DQ: 1
AVX512ER: 0
AVX512PF: 0
AVX512VL: 1
AVX512_IFMA: 0
AVX512_VBMI: 0
BMI1: 1
BMI2: 1
F16C: 1
```

```
FMA: 1
LZCNT: 1
POPCNT: 1
```

В табл. 16.1 представлена сводная информация о `cpuid` для нескольких процессоров Intel. Прежде чем перейти к следующему примеру исходного кода, следует отметить, что при использовании команды `cpuid` для определения поддержки процессором различных расширений набора команд AVX-512 прикладной программе часто бывает необходимо протестировать несколько флагов функций. Например, прикладная программа должна удостовериться, что установлены все флаги функций AVX512F, AVX512DQ и AVX512VL, прежде чем использовать какие-либо команды AVX512DQ с операндами шириной 256 или 128 бит. Руководство разработчика программного обеспечения для архитектур Intel 64 и IA-32 (том 1) содержит дополнительную информацию об использовании команд `cpuid` и тестировании флагов функций.

Таблица 16.1. Сводка информации из команды `cpuid` для некоторых процессоров Intel

Опция CPUID	i3-2310m	i7-4790s	i9-7900x	i7-8700k
LI данные (КБ, на ядро)	32	32	32	32
LI команды (КБ, на ядро)	32	32	32	32
L2 общий (КБ, на ядро)	256	256	1024	256
L3 общий (МБ)	3	8	13	12
ADX	0	0	1	1
AVX	1	1	1	1
AVX2	0	1	1	1
AXV512F	0	0	1	0
AVX512BW	0	0	1	0
AVX512CD	0	0	1	0
AVX512DQ	0	0	1	0
AVX512ER	0	0	0	0
AVX512PF	0	0	0	0
AVX512VL	0	0	1	0
AVX512JFMA	0	0	0	0
AVX512_VBMI	0	0	0	0
BMI1	0	1	1	1
BMI2	0	1	1	1
F16C	0	1	1	1
FMA	0	1	1	1
LZCNT	0	1	1	1
POPCNT	1	1	1	1

16.2. Постоянные хранилища в памяти

С точки зрения кеша памяти данные можно классифицировать как востребованные часто (temporal) либо востребованные однократно или редко (non-temporal). *Часто востребованные данные* – это любое значение, доступ к которому осуществляется более одного раза в течение короткого периода времени. Примеры востребованных данных включают элементы массива или структуры данных, на которые ссылаются несколько раз во время выполнения программного цикла. К ним также относятся байты команд программы. *Редко востребованные данные* – это любое значение, к которому осуществляется однократный доступ и которое не используется повторно в скором времени. Принимающие массивы многих алгоритмов обработки SIMD часто содержат отложенные данные. Различие между востребованными и отложенными данными имеет значение, поскольку производительность процессора нередко ухудшается, если его кеш-память содержит чрезмерные объемы отложенных данных. Это состояние обычно называется *загрязнением кеша* (cache pollution). В идеале кеш-память процессора содержит только востребованные данные, поскольку кешировать элементы, которые используются только один раз, не имеет смысла.

В листинге 16.2 показан исходный код примера `Ch16_02`. Этот пример иллюстрирует использование команды отложенного хранения `vmovntps`. Он также сравнивает быстродействие этой команды со стандартной командой `vmovaps`.

Листинг 16.2. Пример Ch16_02

```
//-----------------------------------------------
//              Ch16_02.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <string>
#include <random>
#include "Ch16_02.h"
#include "AlignedMem.h"

using namespace std;

void Init(float* x, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 1000};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
        x[i] = (float)ui_dist(rng);
}

bool CalcResultCpp(float* c, const float* a, const float* b, size_t n)
{
    size_t align = 32;

    if ((n == 0) || ((n & 0x0f) != 0))
        return false;
```

```
    if (!AlignedMem::IsAligned(a, align))
        return false;
    if (!AlignedMem::IsAligned(b, align))
        return false;
    if (!AlignedMem::IsAligned(b, align))
        return false;

    for (size_t i = 0; i < n; i++)
        c[i] = sqrt(a[i] * a[i] + b[i] * b[i]);

    return true;
}

void CompareResults(const float* c1, const float* c2a, const float*c2b, size_t n)
{
    bool compare_ok = true;
    const float epsilon = 1.0e-9f;

    cout << fixed << setprecision(4);

    for (size_t i = 0; i < n && compare_ok; i++)
    {
        bool b1 = fabs(c1[i] - c2a[i]) > epsilon;
        bool b2 = fabs(c1[i] - c2b[i]) > epsilon;

        cout << setw(2) << i << " - ";
        cout << setw(10) << c1[i] << ' ';
        cout << setw(10) << c2a[i] << ' ';
        cout << setw(10) << c2b[i] << '\n';

        if (b1 || b2)
            compare_ok = false;
    }

    if (compare_ok)
        cout << "Массивы совпадают\n";
    else
        cout << "Массивы НЕ совпадают\n";
}

void NonTemporalStore(void)
{
    const size_t n = 16;
    const size_t align = 32;

    AlignedArray<float> a_aa(n, align);
    AlignedArray<float> b_aa(n, align);
    AlignedArray<float> c1_aa(n, align);
    AlignedArray<float> c2a_aa(n, align);
    AlignedArray<float> c2b_aa(n, align);
    float* a = a_aa.Data();
    float* b = b_aa.Data();
    float* c1 = c1_aa.Data();
    float* c2a = c2a_aa.Data();
    float* c2b = c2b_aa.Data();
```

```
    Init(a, n, 67);
    Init(b, n, 79);

    bool rc1 = CalcResultCpp(c1, a, b, n);
    bool rc2 = CalcResultA_(c2a, a, b, n);
    bool rc3 = CalcResultB_(c2b, a, b, n);

    if (!rc1 || !rc2 || !rc3)
    {
        cout << "Неправильный возвращаемый код\n";
        cout << "rc1 = " << boolalpha << rc1 << '\n';
        cout << "rc2 = " << boolalpha << rc2 << '\n';
        cout << "rc3 = " << boolalpha << rc3 << '\n';
        return;
    }

    cout << "Результаты NonTemporalStore\n";
    CompareResults(c1, c2a, c2b, n);
}

int main()
{
    NonTemporalStore();
    NonTemporalStore_BM();
    return 0;
}

;-----------------------------------------------
;               Ch16_02.asm
;-----------------------------------------------

; _CalcResult Macro
;
; Данный макрос - простой вычислительный цикл, применяемый для
; сравнения быстродействия команд vmovaps и vmovntps

_CalcResult macro MovInstr

; Чтение и проверка аргументов
        xor eax,eax                     ;возвращаемый код ошибки
        test r9,r9
        jz Done                         ;переход, если n <= 0
        test r9,0fh
        jnz Done                        ;переход, если (n % 16) != 0

        test rcx,1fh
        jnz Done                        ;переход, если c не выровнен
        test rdx,1fh
        jnz Done                        ;переход, если a не выровнен
        test r8,1fh
        jnz Done                        ;переход, если b не выровнен

; Вычисление c[i] = sqrt(a[i] * a[i] + b[i] * b[i])
        align 16
@@:     vmovaps ymm0,ymmword ptr [rdx+rax]      ;ymm0 = a[i+7]:a[i]
        vmovaps ymm1,ymmword ptr [r8+rax]       ;ymm1 = b[i+7]:b[i]
```

```
        vmulps ymm2,ymm0,ymm0                   ;ymm2 = a[i] * a[i]
        vmulps ymm3,ymm1,ymm1                   ;ymm3 = b[i] * b[i]
        vaddps ymm4,ymm2,ymm3                   ;ymm4 = sum
        vsqrtps ymm5,ymm4                       ;ymm5 = финальный результат
        MovInstr ymmword ptr [rcx+rax],ymm5     ;сохранение финального значения в c

        vmovaps ymm0,ymmword ptr [rdx+rax+32]   ;ymm0 = a[i+15]:a[i+8]
        vmovaps ymm1,ymmword ptr [r8+rax+32]    ;ymm1 = b[i+15]:b[i+8]
        vmulps ymm2,ymm0,ymm0                   ;ymm2 = a[i] * a[i]
        vmulps ymm3,ymm1,ymm1                   ;ymm3 = b[i] * b[i]
        vaddps ymm4,ymm2,ymm3                   ;ymm4 = sum
        vsqrtps ymm5,ymm4                       ;ymm5 = финальный результат
        MovInstr ymmword ptr [rcx+rax+32],ymm5  ;сохранение финального значения в c

        add rax,64                              ;обновление смещения
        sub r9,16                               ;обновление счетчика
        jnz @B

        mov eax,1                              ;возвращаемый код успешного выполнения

Done:   vzeroupper
        ret
        endm

; extern bool CalcResultA_(float* c, const float* a, const float* b, size_t n)

        .code
CalcResultA_ proc
        _CalcResult vmovaps
CalcResultA_ endp

; extern bool CalcResultB_(float* c, const float* a, const float* b, int n)

CalcResultB_ proc
        _CalcResult vmovntps
CalcResultB_ endp
        end
```

В верхней части листинга 16.2 находится функция C++ `CalcResultCpp`. Эта функция выполняет простые арифметические вычисления с использованием элементов двух исходных массивов с плавающей запятой одинарной точности. Затем она сохраняет результат в массив-приемник. Следующая функция C++ в листинге 16.2 называется `CompareResults`. Эта функция проверяет эквивалентность выходных массивов, сформированных функциями на разных языках программирования. Функция `NonTemporalStore` выделяет место под тестовые массивы и инициализирует их. Затем она вызывает вычислительные функции C++ и ассемблера. Далее выходные массивы трех вычислительных функций сравниваются на предмет каких-либо расхождений.

Код на языке ассемблера в листинге 16.2 начинается с определения макроса с именем `_CalcResult`. Этот макрос генерирует команды AVX, которые выполняют те же вычисления, что и функция C++ `CalcResultCpp`. Макрос `_CalcResult` используется в функциях на языке ассемблера `CalcResultA_` и `CalcResultB_`. Обратите внимание, что `CalcResultA_` предоставляет команду `vmovaps` для параметра

макроса MovInstr, а CalcResultB_ предоставляет vmovntaps. Это означает, что код, выполняемый функциями CalcResultA_ и CalcResultB_, идентичен, за исключением команды перемещения, которая сохраняет результаты в целевой массив. Ниже показан результат выполнения исходного кода примера Ch16_02:

```
Результаты NonTemporalStore

 0 -   240.8319   240.8319   240.8319
 1 -   747.1814   747.1814   747.1814
 2 -   285.1561   285.1561   285.1561
 3 -   862.3062   862.3062   862.3062
 4 -   604.8810   604.8810   604.8810
 5 -  1102.4504  1102.4504  1102.4504
 6 -   347.1441   347.1441   347.1441
 7 -   471.8315   471.8315   471.8315
 8 -   890.6739   890.6739   890.6739
 9 -   729.0878   729.0878   729.0878
10 -   458.3536   458.3536   458.3536
11 -   639.8031   639.8031   639.8031
12 -  1053.1063  1053.1063  1053.1063
13 -  1016.0079  1016.0079  1016.0079
14 -   610.4507   610.4507   610.4507
15 -  1161.7935  1161.7935  1161.7935
Массивы совпадают

Измерение быстродействия функций NonTemporalStore_BM - подождите
Результаты сохранены в файл Ch16_02_NonTemporalStore_BM_CHROMIUM.csv
```

В табл. 16.2 показаны результаты измерения времени выполнения функций из примера Ch16_02 с использованием нескольких различных процессоров Intel. В этом примере использование команды vmovntps вместо команды vmovaps привело к заметному повышению производительности на всех трех компьютерах. Следует отметить, что команды перемещения/присвоения редко запрашиваемых данных x86 дают процессору только подсказку относительно использования памяти. Они не во всех случаях гарантируют повышение производительности. Любой прирост производительности определяется конкретным порядком доступа к памяти и базовой микроархитектурой процессора.

Таблица 16.2. Среднее время выполнения (мс) функций CalcResultCpp, CalcResultA_ и CalcResultB_ (n = 2 000 000)

Процессор	CalcResultCpp	CalcResultA_ (vmovaps)	CalcResultB_ (vmovntps)
i7-4790s	1553	1554	1242
i9-7900x	1173	1139	934
i7-8700k	847	801	590

16.3. Предварительная выборка данных

Прикладная программа также может использовать команду `prefetch` (предварительная выборка данных в кеши) для повышения производительности определенных алгоритмов. Эта команда облегчает предварительную загрузку ожидаемых данных в иерархию кеш-памяти процессора. Есть две основные формы команды предварительной выборки. Первая форма, `prefetcht[0|1|2]`, предварительно загружает многократно используемые данные в определенный уровень кеша. Вторая форма, `prefetchnta`, предварительно загружает редко используемые данные, сводя к минимуму загрязнение кеша. Обе формы команды предварительной выборки предоставляют процессору подсказки о данных, которые программа ожидает использовать; процессор может выбрать выполнение операции предварительной выборки или игнорировать подсказку.

Команды предварительной выборки подходят для использования с различными структурами данных, включая большие массивы и связанные списки. *Связанный список* – это последовательно упорядоченный набор *узлов* (node). Каждый узел включает в себя раздел данных и один или несколько указателей (или ссылок) на соседние узлы. На рис. 16.1 представлен простой связанный список. Связанные списки полезны, поскольку их размер может увеличиваться или уменьшаться (т. е. узлы могут быть добавлены или удалены) в зависимости от требований к хранению данных. Одним из недостатков связанного списка является то, что узлы обычно не хранятся в непрерывном выделенном блоке памяти. Это приводит к увеличению времени доступа при обходе узлов связанного списка.

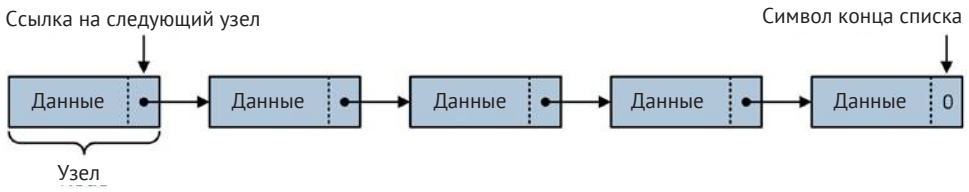


Рис. 16.1. Простой связанный список

Пример исходного кода `Ch16_03` иллюстрирует обход связанного списка как с командой `prefetchnta`, так и без нее. В листинге 16.3 показан исходный код на языках C++ и ассемблера для этого примера.

Листинг 16.3. Пример `Ch16_03`

```
//-----------------------------------------------
//              Ch16_03.h
//-----------------------------------------------

#pragma once
#include <cstdint>

// Эта структура должна совпадать с определением структуры в Ch16_03.asmh
struct LlNode
{
    double ValA[4];
    double ValB[4];
```

```
    double ValC[4];
    double ValD[4];
    uint8_t FreeSpace[376];
    LlNode* Link;
};

// Ch16_03_Misc.cpp
extern bool LlCompare(int num_nodes, LlNode* l1, LlNode* l2, LlNode* l3, int* node_fail);
extern LlNode* LlCreate(int num_nodes);
extern void LlDelete(LlNode* p);
extern bool LlPrint(LlNode* p, const char* fn, const char* msg, bool append);
extern void LlTraverse(LlNode* p);

// Ch16_03_.asm
extern "C" void LlTraverseA_(LlNode* p);
extern "C" void LlTraverseB_(LlNode* p);

// Ch16_03_BM.cpp
extern void LinkedListPrefetch_BM(void);

//-----------------------------------------------
//              Ch16_03.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <cmath>
#include <random>
#include "Ch16_03.h"
#include "AlignedMem.h"

using namespace std;

void LinkedListPrefetch(void)
{
    const int num_nodes = 8;
    LlNode* list1 = LlCreate(num_nodes);
    LlNode* list2a = LlCreate(num_nodes);
    LlNode* list2b = LlCreate(num_nodes);

    LlTraverse(list1);
    LlTraverseA_(list2a);
    LlTraverseB_(list2b);

    int node_fail;
    const char* fn = "Ch16_03_LinkedListPrefetchResults.txt";

    cout << "Результаты LinkedListPrefetch\n";

    if (LlCompare(num_nodes, list1, list2a, list2b, &node_fail))
        cout << "Связанные списки совпадают\n";
    else
        cout << "Связанные списки НЕ совпадают - node_fail = " << node_fail << '\n';

    LlPrint(list1, fn,  "----- list1 -----",  0);
    LlPrint(list2a, fn, "----- list2a -----", 1);
    LlPrint(list2b, fn, "----- list2b -----", 1);
```

```
    cout << "Результаты сохранены в файл " << fn << '\n';

    LlDelete(list1);
    LlDelete(list2a);
    LlDelete(list2b);
}

int main()
{
    LinkedListPrefetch();
    LinkedListPrefetch_BM();
    return 0;
}

//-----------------------------------------------
//               Ch16_03_Misc.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <cmath>
#include <random>
#include "Ch16_03.h"
#include "AlignedMem.h"

using namespace std;

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 == nullptr) || (l2 == nullptr) || (l3 == nullptr))
            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 != nullptr) || (l2 != nullptr) || (l3 != nullptr))
        return false;

    *node_fail = -1;
    return true;
}

LlNode* LlCreate(int num_nodes)
{
    const size_t align = 64;
    const unsigned int seed = 83;
    LlNode* first = nullptr;
    LlNode* last = nullptr;
    uniform_int_distribution<> ui_dist {1, 500};
    default_random_engine rng {seed};

    for (int i = 0; i < num_nodes; i++)
    {
        LlNode* p = (LlNode*)AlignedMem::Allocate(sizeof(LlNode), align);
        p->Link = nullptr;

        if (i == 0)
            first = last = p;
        else
        {
            last->Link = p;
            last = p;
        }

        for (int j = 0; j < 4; j++)
        {
            p->ValA[j] = (double)ui_dist(rng);
            p->ValB[j] = (double)ui_dist(rng);
            p->ValC[j] = 0;
            p->ValD[j] = 0;
        }
    }

    return first;
}

void LlDelete(LlNode* p)
{
    while (p != nullptr)
    {
        LlNode* q = p->Link;

        AlignedMem::Release(p);
        p = q;
    }
}

bool LlPrint(LlNode* p, const char* fn, const char* msg, bool append)
{
    FILE* fp;
    const char* mode = (append) ? "at" : "wt";
```

```
    if (fopen_s(&fp, fn, mode) != 0)
        return false;

    int i = 0;
    const char* fs = "%14.4lf %14.4lf %14.4lf %14.4lf\n";

    if (msg != nullptr)
        fprintf(fp, "\n%s\n", msg);

    while (p != nullptr)
    {
        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;
    }

    fclose(fp);
    return true;
}

void LlTraverse(LlNode* p)
{
    while (p != nullptr)
    {
        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;
    }
}
```

```
;-------------------------------------------------
;               Ch16_03_.asmh
;-------------------------------------------------

; Эта структура должна совпадать с определением Ch16_03.h

LlNode      struct
ValA        real8 4 dup(?)
ValB        real8 4 dup(?)
ValC        real8 4 dup(?)
ValD        real8 4 dup(?)
```

```
FreeSpace   byte 376 dup(?)
Link        qword ?
LlNode      ends

;-----------------------------------------------
;               Ch16_03_.asm
;-----------------------------------------------

        include <Ch16_03_.asmh>

; Macro _LlTraverse
;
; Данный макрос генерирует связанный список, используемый
; командой prefetchnta, если UsePrefetch равно 'Y'

_LlTraverse macro UsePrefetch
        mov rax,rcx                             ;rax = указатель на 1-й узел
        test rax,rax
        jz Done                                 ;переход, если список пуст

        align 16
@@::    mov rcx,[rax+LlNode.Link]               ;rcx = next следующий узел
        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                    ;еще остались узлы?
        cmovz rdx,rax                   ;исключить выборку по указателю null
        prefetchnta [rdx]               ;старт выборки следующего узла
ENDIF

; Вычисление 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 = суммы
        vsqrtpd ymm5,ymm4               ;ymm5 = квадратные корни

        vmovntpd ymmword ptr [rax+LlNode.ValC],ymm5 ;сохранение результата

; Вычисление 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 = sums
        vsqrtpd ymm5,ymm4               ;ymm5 = square roots

        vmovntpd ymmword ptr [rax+LlNode.ValD],ymm5 ;сохранение результата

        mov rax,rcx                     ;rax = указатель на следующий узел
        test rax,rax
        jnz @B

Done:   vzeroupper
        ret
        endm
```

```
; extern "C" void LlTraverseA_(LlNode* first);

        .code
LlTraverseA_ proc
        _LlTraverse n
LlTraverseA_ endp

; extern "C" void LlTraverseB_(LlNode* first);

LlTraverseB_ proc
        _LlTraverse y
LlTraverseB_ endp
        end
```

Листинг 16.3 начинается с заголовочного файла `Ch16_03.h`. В верхней части этого файла находится объявление структуры `LlNode`. Код C++ использует эту структуру для создания связанных списков тестовых данных. Члены структуры от `ValA` до `ValD` содержат значения данных, которыми манипулируют функции обхода связанного списка. Член `FreeSpace` добавлен для увеличения размера `LlNode` в демонстрационных целях, поскольку предварительная выборка лучше всего работает с более крупными структурами данных. Реальная реализация `LlNode` могла бы использовать это пространство для дополнительных элементов данных. Последний член `LlNode` – это указатель с именем `Link`, который указывает на следующую структуру `LlNode`. Аналог `LlNode` на языке ассемблера объявлен в файле `Ch16_03_.asmh`.

Базовая функция для примера исходного кода `Ch16_03` называется `LinkedListPrefetch` и располагается в файле исходного кода `Ch16_03.cpp`. Эта функция создает несколько тестовых связанных списков, вызывает функции обхода на языках C++ и ассемблера и сравнивает результаты. Файл исходного кода `Ch16_03_misc.cpp` содержит набор различных функций, которые выполняют основные операции обработки связанного списка. Функция `LlCompare` сравнивает узлы данных своих связанных списков, переданных ей в виде аргументов, на эквивалентность. Функции `LlCreate` и `LlDelete` выполняют выделение и удаление связанного списка. `LlPrint` выгружает содержимое данных связанного списка в файл. Наконец, `LlTraverse` просматривает связанный список и выполняет имитацию вычислений с использованием элементов данных `LlNode` `ValA`, `ValB`, `ValC` и `ValD`.

В начале файла `Ch16_03_.asm` находится макрос с именем `_LlTraverse`. Этот макрос генерирует код, выполняющий такой же обход связанного списка и моделирование вычислений, что и функция `LlTraverse` на языке C++. Макросу `_LlTraverse` требуется один параметр, который включает или отключает предварительную выборку данных. Если предварительная выборка включена, макрос генерирует короткий блочный код, включающий команду `prefetchnta [rdx]`. Эта команда предписывает процессору выполнить предварительную выборку однократных данных, на которые указывает регистр RDX. В данном примере RDX указывает на следующий `LlNode` в связанном списке. Фактическое количество байтов, извлекаемых этой командой, зависит от базовой микроархитектуры; процессоры Intel извлекают минимум 32 байта. Чрезвычайно важно отметить, что `prefetchnta [rdx]` располагается *перед* командами вычисления с плавающей запятой. Это дает процессору возможность получить данные для следующе-

го узла, одновременно выполняя арифметические вычисления для текущего узла. Также обратите внимание, что перед выполнением команды `prefetchnta [rdx]` содержимое RDX проверяется, чтобы избежать использования команды `prefetchnta` с адресом `nullptr` (или нулевым) в памяти, поскольку это снижает производительность процессора. Это важно, т. к. `nullptr` применяется в качестве значения признака конца списка. Функции на языке ассемблера `LlTraverseA_` и `LlTraverseB_` используют макрос `_LlTraverse` для выполнения обхода связанного списка как без предварительной выборки, так и с предварительной выборкой соответственно. Ниже показан результат выполнения кода примера `Ch16_03`.

```
Результаты LinkedListPrefetch
Связанные списки совпадают
Результаты сохранены в файл Ch16_03_LinkedListPrefetchResults.txt

Измерение быстродействия функции LinkedListPrefetch_BM - подождите
Результаты сохранены в файл Ch16_03_LinkedListPrefetch_BM_CHROMIUM.csv
```

В табл. 16.3 показаны результаты измерения времени выполнения функций для нескольких процессоров Intel. Важно помнить, что любые преимущества в производительности, обеспечиваемые командами предварительной выборки, во многом зависят от текущей загрузки процессора, шаблонов доступа к данным и базовой микроархитектуры. Согласно «Справочному руководству по оптимизации архитектур Intel 64 и IA-32», команды предварительной выборки данных «зависят от реализации». Это означает, что для максимальной производительности предварительной выборки алгоритм должен быть «настроен на каждую реализацию» или микроархитектуру. Рекомендуем вам обратиться к вышеупомянутому справочному руководству для получения дополнительной информации об использовании команд по предварительной выборке данных x86.

Таблица 16.3. Среднее время выполнения (мс) функций обхода связанного списка (num_nodes = 50 000)

Процессор	LlTraverse (C++)	LlTraverseA_ (без `prefetchnta`)	LlTraverseB_ (с `prefetchnta`)
i7-4790s	5685	3093	2680
i9-7900x	5885	3064	2842
i7-8700k	5031	2384	2319

16.4. Многопоточность

Все примеры исходного кода, представленные в этой книге, имеют одну общую характеристику: все они содержат однопоточный код. Поскольку вы читаете эту книгу, вам наверняка уже известно, что большинство современных приложений используют хотя бы несколько потоков, чтобы лучше применять несколько ядер современных процессоров. Например, многие высокопроизводительные вычислительные приложения часто выполняют арифметические

вычисления с использованием больших массивов данных, содержащих миллионы элементов с плавающей запятой. Одна из стратегий, которая часто используется для повышения производительности этих типов вычислений, состоит в том, чтобы распределить элементы массива по нескольким потокам и заставить каждый поток выполнять подмножество требуемых вычислений. В следующем примере исходного кода показано, как выполнять арифметические вычисления с использованием больших массивов с плавающей запятой и нескольких потоков. В листинге 16.4 приведен исходный код примера `Ch16_04`.

Внимание! При выполнении многопоточного кода, в котором широко используются команды x86-AVX, процессор может сильно нагреться. Перед запуском кода примера `Ch16_04` вы должны убедиться, что процессор вашего компьютера имеет достаточно эффективную систему охлаждения.

Листинг 16.4. Пример Ch16_04

```
//-----------------------------------------------
//               Ch16_04.h
//-----------------------------------------------

#pragma once
#include <vector>

struct CalcInfo
{
    double* m_X1;
    double* m_X2;
    double* m_Y1;
    double* m_Y2;
    double* m_Z1;
    double* m_Z2;
    double* m_Result;
    size_t m_Index0;
    size_t m_Index1;
    int m_Status;
};

struct CoutInfo
{
    bool m_ThreadMsgEnable;
    size_t m_Iteration;
    size_t m_NumElements;
    size_t m_ThreadId;
    size_t m_NumThreads;
};

// Ch16_04_Misc.cpp
extern size_t CompareResults(const double* a, const double* b, size_t n);
extern void DisplayThreadMsg(const CalcInfo* ci, const CoutInfo* cout_info, const char*
msg);
extern void Init(double* a1, double* a2, size_t n, unsigned int seed);
std::vector<size_t> GetNumElementsVec(size_t* num_elements_max);
std::vector<size_t> GetNumThreadsVec(void);
```

```
// Ch16_04_WinApi.cpp
extern bool GetAvailableMemory(size_t* mem_size);

// Ch16_04_.asm
extern "C" void CalcResult_(CalcInfo* ci);

// Различные константы
const size_t c_ElementSize = sizeof(double);

const size_t c_NumArrays = 8;   // Общее число размещенных массивов
const size_t c_Align = 32;      // Граница выравнивания (обновите Ch16_04_.asm при
изменении)
const size_t c_BlockSize = 8;   // Элементов на итерацию (обновите Ch16_04_.asm при
изменении)

//-----------------------------------------------
//               Ch16_04_Misc.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <random>
#include <memory.h>
#include <cmath>
#include <mutex>
#include <vector>
#include <algorithm>
#include "Ch16_04.h"

using namespace std;

void Init(double* a1, double* a2, size_t n, unsigned int seed)
{
    uniform_int_distribution<> ui_dist {1, 2000};
    default_random_engine rng {seed};

    for (size_t i = 0; i < n; i++)
    {
        a1[i] = (double)ui_dist(rng);
        a2[i] = (double)ui_dist(rng);
    }
}

size_t CompareResults(const double* a, const double* b, size_t n)
{
    if (memcmp(a, b, n * sizeof(double)) == 0)
        return n;

    const double epsilon = 1.0e-15;

    for (size_t i = 0; i < n; i++)
    {
        if (fabs(a[i] - b[i]) > epsilon)
            return i;
    }

    return n;
}
```

```
void DisplayThreadMsg(const CalcInfo* ci, const CoutInfo* cout_info, const char* msg)
{
    static mutex mutex_cout;
    static const char nl = '\n';

    mutex_cout.lock();
    cout << nl << msg << nl;
    cout << "  m_Iteration:   " << cout_info->m_Iteration << nl;
    cout << "  m_NumElements: " << cout_info->m_NumElements << nl;
    cout << "  m_ThreadId:    " << cout_info->m_ThreadId << nl;
    cout << "  m_NumThreads:  " << cout_info->m_NumThreads << nl;
    cout << "  m_Index0:      " << ci->m_Index0 << nl;
    cout << "  m_Index1:      " << ci->m_Index1 << nl;
    mutex_cout.unlock();
}

vector<size_t> GetNumElementsVec(size_t* num_elements_max)
{
//    vector<size_t> ne_vec {64, 192, 384, 512};      // Требует 32 ГБ и более
      vector<size_t> ne_vec {64, 128, 192, 256};      // Требует 16 ГБ и более
//    vector<size_t> ne_vec {64, 96, 128, 160};       // Требует 10 ГБ и более

    size_t mem_size_extra_gb = 2;       // Предотвращает заполнение всей доступной памяти

    size_t ne_max = *std::max_element(ne_vec.begin(), ne_vec.end());

    if ((ne_max % c_BlockSize) != 0)
        throw runtime_error("ne_max должен быть кратен c_BlockSize");

    size_t mem_size;

    if (!GetAvailableMemory(&mem_size))
        throw runtime_error ("Не удалось выполнить GetAvailableMemory");

    size_t mem_size_gb = mem_size / (1024 * 1024 * 1024);
    size_t mem_size_min = ne_max * 1024 * 1024 * c_ElementSize * c_NumArrays;
    size_t mem_size_min_gb = mem_size_min / (1024 * 1024 * 1024);

    if (mem_size_gb < mem_size_min_gb + mem_size_extra_gb)
        throw runtime_error ("Недостаточно свободной памяти");

    *num_elements_max = ne_max * 1024 * 1024;
    return ne_vec;
}

vector<size_t> GetNumThreadsVec(void)
{
    vector<size_t> num_threads_vec {1, 2, 4, 6, 8};

    return num_threads_vec;
}
```

```
//-----------------------------------------------
//              Ch16_04.cpp
//-----------------------------------------------

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <sstream>
#include <stdexcept>
#include <thread>
#include <vector>
#include "Ch16_04.h"
#include "AlignedMem.h"
#include "BmThreadTimer.h"

using namespace std;

// Контрольный флаг для передачи информации о статусе потока в cout.
const bool c_ThreadMsgEnable = false;

void CalcResultCpp(CalcInfo* ci)
{
    size_t al = c_Align;
    size_t i0 = ci->m_Index0;
    size_t i1 = ci->m_Index1;
    size_t num_elements = i1 - i0 + 1;

    ci->m_Status = 0;

    if (num_elements == 0 || (num_elements % c_BlockSize) != 0)
        return;

    for (size_t i = i0; i <= i1; i++)
    {
        double xx = ci->m_X1[i] - ci->m_X2[i];
        double yy = ci->m_Y1[i] - ci->m_Y2[i];
        double zz = ci->m_Z1[i] - ci->m_Z2[i];

        ci->m_Result[i] = sqrt(1.0 / sqrt(xx * xx + yy * yy + zz * zz));
    }

    ci->m_Status = 1;
}

static void CalcResultThread(CalcInfo* ci, CoutInfo* cout_info)
{
    if (cout_info->m_ThreadMsgEnable)
        DisplayThreadMsg(ci, cout_info, "Вход в CalcResultThread()");

    CalcResult_(ci);

    if (cout_info->m_ThreadMsgEnable)
        DisplayThreadMsg(ci, cout_info, "Выход из CalcResultThread()");
}

void RunMultipleThreads(bool thread_msg_enable)
```

```
{
    // Секция кода #1

    size_t align = c_Align;
    size_t num_elements_max;
    vector<size_t> num_elements_vec = GetNumElementsVec(&num_elements_max);
    vector<size_t> num_threads_vec = GetNumThreadsVec();

    AlignedArray<double> x1_aa(num_elements_max, align);
    AlignedArray<double> x2_aa(num_elements_max, align);
    AlignedArray<double> y1_aa(num_elements_max, align);
    AlignedArray<double> y2_aa(num_elements_max, align);
    AlignedArray<double> z1_aa(num_elements_max, align);
    AlignedArray<double> z2_aa(num_elements_max, align);
    AlignedArray<double> result1_aa(num_elements_max, align);
    AlignedArray<double> result2_aa(num_elements_max, align);

    double* x1 = x1_aa.Data();
    double* x2 = x2_aa.Data();
    double* y1 = y1_aa.Data();
    double* y2 = y2_aa.Data();
    double* z1 = z1_aa.Data();
    double* z2 = z2_aa.Data();
    double* result1 = result1_aa.Data();
    double* result2 = result2_aa.Data();

    cout << "Начало инициализации тестовых массивов\n";
    cout << "  Инициализация тестовых массивов x1, x2\n";
    Init(x1, x2, num_elements_max, 307);
    cout << "  Инициализация тестовых массивов y1, y2\n";
    Init(y1, y2, num_elements_max, 401);
    cout << "  Инициализация тестовых массивов z1, z2\n";
    Init(z1, z2, num_elements_max, 503);
    cout << "Инициализация тестовых массивов завершена\n";

    CalcInfo ci1;
    ci1.m_X1 = x1;  ci1.m_X2 = x2;
    ci1.m_Y1 = y1;  ci1.m_Y2 = y2;
    ci1.m_Z1 = z1;  ci1.m_Z2 = z2;
    ci1.m_Result = result1;
    ci1.m_Index0 = 0;
    ci1.m_Index1 = num_elements_max - 1;
    ci1.m_Status = -1;

    // CalcResultCpp применяется для целей сравнения
    cout << "Начало выполнения CalcResultCpp\n";
    CalcResultCpp(&ci1);
    cout << "Выполнение CalcResultCpp завершено\n";

    size_t iteration = 0;
    const size_t block_size = c_BlockSize;
    BmThreadTimer bmtt(num_elements_vec.size(), num_threads_vec.size());

    // Секция кода #2

    cout << "Начало выполнения вычислительных потоков\n";
```

```
for (size_t i = 0; i < num_elements_vec.size(); i++)
{
    size_t num_elements = num_elements_vec[i] * 1024 * 1024;
    size_t num_blocks = num_elements / block_size;
    size_t num_blocks_rem = num_elements % block_size;

    if (num_blocks_rem != 0)
        throw runtime_error("num_elements должен быть кратен block_size");

    for (size_t j = 0; j < num_threads_vec.size(); j++)
    {
        size_t num_threads = num_threads_vec[j];

        bmtt.Start(i, j);

        size_t num_blocks_per_thread = num_blocks / num_threads;
        size_t num_blocks_per_thread_rem = num_blocks % num_threads;

        vector<CalcInfo> ci2(num_threads);
        vector<CoutInfo> cout_info(num_threads);
        vector<thread*> threads(num_threads);

        // Код запуска потока
        for (size_t k = 0; k < num_threads; k++)
        {
            ci2[k].m_X1 = x1;   ci2[k].m_X2 = x2;
            ci2[k].m_Y1 = y1;   ci2[k].m_Y2 = y2;
            ci2[k].m_Z1 = z1;   ci2[k].m_Z2 = z2;

            ci2[k].m_Result = result2;
            ci2[k].m_Index0 = k * num_blocks_per_thread * block_size;
            ci2[k].m_Index1 = (k + 1) * num_blocks_per_thread * block_size - 1;
            ci2[k].m_Status = -1;

            if ((k + 1) == num_threads)
                ci2[k].m_Index1 += num_blocks_per_thread_rem * block_size;

            cout_info[k].m_ThreadMsgEnable = thread_msg_enable;
            cout_info[k].m_Iteration = iteration;
            cout_info[k].m_NumElements = num_elements;
            cout_info[k].m_NumThreads = num_threads;
            cout_info[k].m_ThreadId = k;

            threads[k] = new thread(CalcResultThread, &ci2[k], &cout_info[k]);
        }

        // Ожидание завершения всех потоков
        for (size_t k = 0; k < num_threads; k++)
            threads[k]->join();

        bmtt.Stop(i, j);

        size_t cmp_index = CompareResults(result1, result2, num_elements);

        if (cmp_index != num_elements)
        {
```

```
                ostringstream oss;
                oss << "  несовпадение данных с индексом " << cmp_index;
                throw runtime_error(oss.str());
            }

            for (size_t k = 0; k < num_threads; k++)
            {
                if (ci2[k].m_Status != 1)
                {
                    ostringstream oss;
                    oss << "  ошибочный код состояния " << ci2[k].m_Status;
                    throw runtime_error(oss.str());
                }

                delete threads[k];
            }
        }

        iteration++;
    }

    cout << "Выполнение вычислительных потоков завершено\n";

    string fn = bmtt.BuildCsvFilenameString("Ch16_04_MultipleThreads_BM");
    bmtt.SaveElapsedTimes(fn, BmThreadTimer::EtUnit::MilliSec, 0);
    cout << "Результаты измерения быстродействия сохранены в файл " << fn << '\n';
}

int main()
{
    try
    {
        RunMultipleThreads(c_ThreadMsgEnable);
    }

    catch (runtime_error& rte)
    {
        cout << "обнаружена ошибка выполнения программы - " << rte.what() << '\n';
    }

    catch (...)
    {
        cout << "Произошло аварийное завершение программы\n";
    }

    return 0;
}

;-----------------------------------------------
;               Ch16_04_.asm
;-----------------------------------------------

        include <MacrosX86-64-AVX.asmh>

CalcInfo struct
    X1 qword ?
```

```
    X2 qword ?
    Y1 qword ?
    Y2 qword ?
    Z1 qword ?
    Z2 qword ?
    Result qword ?
    Index0 qword ?
    Index1 qword ?
    Status dword ?
CalcInfo ends

        .const
r8_1p0  real8 1.0

; extern "C" void CalcResult_(CalcInfo* ci)

        .code
CalcResult_ proc frame
        _CreateFrame CR,0,16,r12,r13,r14,r15
        _SaveXmmRegs xmm6
        _EndProlog

        mov dword ptr [rcx+CalcInfo.Status],0

; Проверка значения num_elements
        mov rax,[rcx+CalcInfo.Index0]       ;rax = начальный индекс
        mov rdx,[rcx+CalcInfo.Index1]       ;rdx = конечный индекс
        sub rdx,rax
        add rdx,1                           ;rdx = num_elements
        test rdx,rdx
        jz Done                             ;переход, если num_elements == 0
        test rdx,7
        jnz Done                            ;переход, если num_elements % 8 != 0

; Проверка правильности выравнивания всех массивов
        mov r8d,1fh
        mov r9,[rcx+CalcInfo.Result]
        test r9,r8
        jnz Done

        mov r10,[rcx+CalcInfo.X1]
        test r10,r8
        jnz Done
        mov r11,[rcx+CalcInfo.X2]
        test r11,r8
        jnz Done

        mov r12,[rcx+CalcInfo.Y1]
        test r12,r8
        jnz Done
        mov r13,[rcx+CalcInfo.Y2]
        test r13,r8
        jnz Done

        mov r14,[rcx+CalcInfo.Z1]
        test r14,r8
```

```
        jnz Done
        mov r15,[rcx+CalcInfo.Z2]
        test r15,r8
        jnz Done

        vbroadcastsd ymm6,real8 ptr [r8_1p0]        ;ymm6 = packed 1.0 (DPFP)

; Выполнение симуляционных вычислений
        align 16
LP1:    vmovapd ymm0,ymmword ptr [r10+rax*8]
        vmovapd ymm1,ymmword ptr [r12+rax*8]
        vmovapd ymm2,ymmword ptr [r14+rax*8]
        vsubpd ymm0,ymm0,ymmword ptr [r11+rax*8]
        vsubpd ymm1,ymm1,ymmword ptr [r13+rax*8]
        vsubpd ymm2,ymm2,ymmword ptr [r15+rax*8]
        vmulpd ymm3,ymm0,ymm0
        vmulpd ymm4,ymm1,ymm1
        vmulpd ymm5,ymm2,ymm2
        vaddpd ymm0,ymm3,ymm4
        vaddpd ymm1,ymm0,ymm5
        vsqrtpd ymm2,ymm1
        vdivpd ymm3,ymm6,ymm2
        vsqrtpd ymm4,ymm3
        vmovntpd ymmword ptr [r9+rax*8],ymm4

        add rax,4

        vmovapd ymm0,ymmword ptr [r10+rax*8]
        vmovapd ymm1,ymmword ptr [r12+rax*8]
        vmovapd ymm2,ymmword ptr [r14+rax*8]
        vsubpd ymm0,ymm0,ymmword ptr [r11+rax*8]
        vsubpd ymm1,ymm1,ymmword ptr [r13+rax*8]
        vsubpd ymm2,ymm2,ymmword ptr [r15+rax*8]
        vmulpd ymm3,ymm0,ymm0
        vmulpd ymm4,ymm1,ymm1
        vmulpd ymm5,ymm2,ymm2
        vaddpd ymm0,ymm3,ymm4
        vaddpd ymm1,ymm0,ymm5
        vsqrtpd ymm2,ymm1
        vdivpd ymm3,ymm6,ymm2
        vsqrtpd ymm4,ymm3
        vmovntpd ymmword ptr [r9+rax*8],ymm4

        add rax,4
        sub rdx,8
        jnz LP1

        mov dword ptr [rcx+CalcInfo.Status],1

Done:   vzeroupper
        _RestoreXmmRegs xmm6
        _DeleteFrame r12,r13,r14,r15
        ret
CalcResult_ endp
        end
```

Пример исходного кода Ch16_04 выполняет моделирование вычислений с использованием больших массивов значений с плавающей запятой двойной точности в нескольких потоках. Он использует класс C++ STL thread для запуска нескольких экземпляров функции вычисления на языке ассемблера AVX2. Каждый поток выполняет свои вычисления, используя только часть данных массива. Управляющая программа C++ реализует алгоритм вычислений, используя различные комбинации размеров массивов и одновременно выполняя потоки. Она также выполняет измерение времени выполнения, чтобы количественно оценить преимущества производительности многопоточной технологии.

Листинг 16.4 начинается с заголовочного файла Ch16_04.h. В этом файле структура CalcInfo содержит данные, необходимые каждому потоку для выполнения своих вычислений. Члены структуры m_X1, m_X2, m_Y1, m_Y2, m_Z1 и m_Z2 указывают на исходные массивы, в то время как m_Result указывает на массив-приемник. Члены m_Index0 и m_Index1 являются индексами массива, которые определяют диапазон уникальных элементов для каждого вычислительного потока. Заголовочный файл Ch16_04.h также включает структуру с именем CoutInfo, предоставляющую информацию о состоянии, которая может отображаться во время выполнения программы.

Следующий файл в листинге 16.4 – Ch16_04_Misc.cpp. Этот файл содержит исходный код вспомогательных функций программы. Функции Init и CompareResults выполняют инициализацию и проверку массива соответственно. Функция DisplayThreadMsg отображает информацию о состоянии каждого выполняющегося потока. Обратите внимание, что операторы cout в DisplayThreadMsg синхронизируются с объектом mutex C ++ STL. Это объект синхронизации, который обеспечивает контролируемый доступ к одному ресурсу для нескольких потоков. Когда mutex_cout заблокирован, только один поток может передавать результаты своего статуса в cout. Другие выполняющиеся потоки заблокированы от потоковой передачи своих результатов в cout до тех пор, пока mutex_cout не будет разблокирован. Без этого компонента текст информации о состоянии выполняющихся потоков смешивался бы на дисплее (если вы хотите увидеть, что произойдет, попробуйте закомментировать операторы mutex_cout.lock и mutext_cout.unlock).

Функция GetNumElementsVec возвращает вектор, содержащий размеры тестовых массивов. Обратите внимание, что объем памяти, необходимый для самого большого тестового массива, должен быть меньше объема доступной памяти плюс небольшой поправочный коэффициент. Этот поправочный коэффициент не позволяет программе выделить всю доступную память. Также обратите внимание, что GetNumElementsVec выдает исключение, если доступной памяти недостаточно, поскольку выполнение программы в этом случае происходит очень медленно из-за того, что происходит подкачка страниц. Функция GetNumThreadsVec возвращает вектор количества тестовых потоков. Вы можете изменить значения в num_threads_vec, чтобы поэкспериментировать с разными значениями счетчика потоков.

Файл исходного кода Ch16_04.cpp содержит управляющие процедуры для примера исходного кода Ch16_04. Функция CalcResultCpp – это реализация моделируемого вычислительного алгоритма на C++, которая используется для проверки результатов. Следующая функция, CalcResultThread, является функцией основного потока. Эта функция вызывает вычислительную функцию на языке

ассемблера CalcResult_. Она также отображает сообщения о статусе потока, если они включены.

За CalcResultThread следует функция RunMultipleThreads. Эта функция выполняет алгоритм вычислений с использованием заданных комбинаций размеров массива и количества одновременно выполняющихся потоков. Первый раздел кода RunMultipleThreads выполняет выделение массива и инициализацию элементов. Он также вызывает функцию CalcResultCpp для вычисления значений результатов в целях проверки алгоритма. Обратите внимание, что перед вызовом этой функции экземпляр CalcInfo инициализируется данными, необходимыми для выполнения требуемых вычислений.

Второй раздел кода RunMultipleThreads, который начинается сразу после строки комментария Раздел кода #2, запускает алгоритм вычислений, распределяя элементы тестового массива по нескольким потокам. Элементы тестового массива разбиты на группы в зависимости от количества выполняемых потоков. Например, если количество элементов тестового массива равно 64 млн, запуск четырех потоков приведет к тому, что каждый поток обработает 16 млн элементов. Внутренний цикл for, следующий за строкой комментария «Код запуска потока», начинает каждую итерацию с инициализации экземпляра CalcInfo для следующего потока. Оператор threads[k] = new thread(CalcResultThread, &ci2[k], &cout_info[k]) создает новый объект потока и запускает выполнение функции потока CalcResultThread, используя значения аргументов &ci2[k] и &cout_info[k]. Этот внутренний цикл for повторяется до тех пор, пока не будет запущено необходимое количество выполняющихся потоков. Пока потоки выполняются, функция RunMultipleThreads осуществляет небольшой цикл for, который вызывает функцию thread::join. Это фактически заставляет RunMultipleThreads ждать, пока все выполняющиеся потоки не закончатся. Оставшийся код в RunMultipleThreads выполняет проверку данных и очистку объекта.

Перед рассмотрением кода на языке ассемблера имеет смысл сделать несколько дополнительных комментариев относительно кода C++ в RunMultipleThreads. Прежде всего следует отметить, что проверочный код измеряет как время, необходимое для выполнения необходимых вычислений, так и накладные расходы, связанные с управлением потоками. Если бы алгоритм, используемый RunMultipleThreads, применялся в реальном приложении, любые измерения времени выполнения были бы бессмысленными без учета этих накладных расходов. Следует также отметить, что RunMultipleThreads реализует крайне рудиментарную форму многопоточности, которая пропускает многие важные реальные операции для упрощения кода для этого примера. Если вам интересно узнать больше о классе потоков STL C++ и других классах STL, которые упрощают многопоточную обработку, настоятельно рекомендуется ознакомиться со ссылками, перечисленными в приложении.

Листинг 16.4 завершается файлом Ch16_04_.asm. В верхней части этого файла расположен аналог структуры данных CalcInfo на языке ассемблера. Дальше идет функция на языке ассемблера CalcResult_. Вслед за прологом она выполняет команды mov rax,[rcx+CalcInfo.Index0] и mov rdx,[rcx+CalcInfo.Index1] для загрузки регистров RAX и RDX с индексами первого и последнего элементов массива. Затем она вычисляет и проверяет num_elements. Далее тестовые массивы проверяются на правильность выравнивания. Цикл обработки в CalcResult_

использует арифметику AVX упакованных чисел с плавающей запятой двойной точности для выполнения того же имитационного вычисления, что и CalcResultCpp. Ниже показан результат выполнения примера кода Ch16_04. Обратите внимание, что этот вывод был создан с параметром c_ThreadMsgEnable, установленным в false. Установка данного флага в значение true предписывает функции CalcResultThread отображать сообщения о состоянии каждого выполняющегося потока. Флаг c_ThreadMsgEnable определен в верхней части файла Ch16_04.cpp.

```
Начало инициализации тестовых массивов
  Инициализация тестовых массивов x1, x2
  Инициализация тестовых массивов y1, y2
  Инициализация тестовых массивов z1, z2
Инициализация тестовых массивов завершена
Начало выполнения CalcResultCpp
Выполнение CalcResultCpp завершено
Начало выполнения вычислительных потоков
Выполнение вычислительных потоков завершено
Результаты измерения быстродействия сохранены в файл Ch16_04_MultipleThreads_BM_CHROMIUM.
csv
```

Таблицы 16.4, 16.5 и 16.6 содержат результаты измерения быстродействия примера исходного кода Ch16_04. Измерения, показанные в этих таблицах, представляют собой среднее время выполнения из 10 отдельных запусков и были выполнены с 32 ГБ SDRAM, установленными на каждом тестовом компьютере. Все три тестовых компьютера показывают значительное улучшение производительности при использовании нескольких потоков для моделирования расчетов. На тестовых компьютерах i7-4900s и i7-8700k оптимальная производительность достигается при использовании четырех потоков. Тестовый компьютер i9-7900x показывает значительный прирост производительности при применении шести или восьми потоков. Интересно сравнить измерения быстродействия для систем i9-7900x и i7-8700k. При использовании одного или двух потоков i7-8700k обгоняет i9-7900x, в то время как обратное верно, когда используются четыре или более потоков. Эти измерения имеют смысл при рассмотрении аппаратных различий между тестовыми процессорами, которые показаны в табл. 16.7. Несмотря на то что i7-8700k использует более высокие тактовые частоты, дополнительные каналы памяти i9-7900x позволяют ему лучше применять ядра процессора и быстрее выполнять необходимые вычисления.

Таблица 16.4. Результаты измерения времени выполнения (мс) для RunMultipleThreads с использованием процессора Intel i7-4790s

	Количество потоков				
Количество элементов (млн)	**1**	**2**	**4**	**6**	**8**
64	686	226	178	180	172
128	1146	491	345	355	347
192	1592	702	513	530	516
256	2054	942	679	714	688

Таблица 16.5. Результаты измерения времени выполнения (мс) для `RunMultipleThreads` с использованием процессора Intel i9-7900x

	Количество потоков				
Количество элементов (млн)	**1**	**2**	**4**	**6**	**8**
64	492	137	84	69	61
128	765	300	163	131	121
192	1110	454	233	193	178
256	1330	582	313	260	238

Таблица 16.6. Результаты измерения времени выполнения (мс) для `RunMultipleThreads` с использованием процессора Intel i7-8700k

	Количество потоков				
Количество элементов (млн)	**1**	**2**	**4**	**6**	**8**
64	332	125	120	123	123
128	522	265	240	245	246
192	839	387	363	366	369
256	919	499	478	484	492

Таблица 16.7. Сводка аппаратных характеристик тестовых процессоров, использованных в примере `Ch16_04`

Параметр	i7-4790s	i9-7900x	i7-8700k
Количество ядер	4	10	6
Количество потоков	8	20	12
Базовая частота (ГГц)	3,2	3,3	3,7
Максимальная частота (ГГц)	4,0	4,5	4,7
Тип памяти	DDR3-1600	DDR4-2666	DDR4-2666
Количество каналов памяти	2	4	2

16.5. Заключение

В главе 16 были рассмотрены следующие ключевые моменты:

- прикладная программа всегда должна использовать команду `cpuid` для проверки поддержки процессором определенных расширений набора команд. Это чрезвычайно важно для программной совместимости с будущими процессорами AMD и Intel;
- функция на языке ассемблера может использовать команды для работы с редко или однократно извлекаемыми данными `vmovntp[d|s]` вместо команд `vmovap[d|s]` для повышения производительности алгоритмов, выполняющих вычисления с использованием больших массивов однократно извлекаемых данных с плавающей запятой;

- функция на языке ассемблера может использовать команды `prefetch[0|1|2]` для предварительной загрузки многократно используемых данных в иерархию кеша процессора. Функция также может использовать команду `prefetchnta` для предварительной загрузки однократных данных и минимизации загрязнения кеша. Выигрыш производительности от применения команд предварительной выборки различается в зависимости от шаблонов доступа к данным и базовой микроархитектуры процессора;
- многопоточный алгоритм, реализованный на языке высокого уровня, таком как C++, может использовать вычислительные функции на языке ассемблера AVX, AVX2 или AVX-512 для повышения общей производительности алгоритма.

Приложение

Данное приложение включает дополнительные материалы по следующим вопросам:

- программные утилиты для процессоров x86;
- Visual Studio;
- рекомендации.

П.1. Программные утилиты для процессоров x86

Следующие утилиты можно использовать для определения того, какие расширения набора инструкций x86 поддерживаются процессором вашего компьютера:

- CPUID CPU-Z (https://www.cpuid.com);
- утилиты диагностики HWiNFO (https://www.hwinfo.com);
- Piriform SPECCY (https://www.ccleaner.com/speccy).

П.2. Visual Studio

В этом разделе вы узнаете, как использовать среду разработки Microsoft Visual Studio для запуска примеров исходного кода, описанных в основном тексте. Вы также узнаете, как создать простой проект Visual Studio C++. Прежде чем продолжить, вы можете обратиться к введению для получения дополнительной информации о Visual Studio и рекомендуемых аппаратных платформах для запуска примеров исходного кода. Введение также содержит важные сведения о загрузке архива файлов с исходным кодом для каждой главы.

Чтобы упростить разработку приложений, Visual Studio использует логические сущности, называемые решениями и проектами. *Решение* – это набор из одного или нескольких проектов, которые используются для создания приложения. *Проект* – это объект-контейнер, который организует файлы приложения. Проект Visual Studio обычно создается для каждого собираемого компонента приложения (например, исполняемого файла, динамически подключаемой библиотеки, статической библиотеки и т. д.).

Стандартный проект Visual Studio C++ включает две конфигурации решения с именами Debug и Release. Как следует из их названий, эти конфигурации поддерживают отдельные исполняемые сборки для начальной разработки и окончательного выпуска. Стандартный проект Visual Studio C++ также включает *платформы решений*. Платформы решений по умолчанию называются Win32 и x64; они содержат необходимые настройки для создания 32-битных и 64-битных исполняемых файлов соответственно. Файлы решения и проекта Visual Studio для примеров исходного кода этой книги включают только платформу x64.

П.2.1. Запуск примера исходного кода

Для запуска любого примера из этой книги:

1) в проводнике дважды щелкните файл решения Visual Studio (`.sln`) главы. Файл решения включен в ZIP-файл с исходным кодом главы;
2) в строке меню выберите **Build | Configuration Manager** (Сборка | Менеджер конфигурации). В диалоговом окне **Cofiguration Manager** установите для параметра **Active Solution Configuration** (Конфигурация активного решения) значение **Release** (Выпуск). Затем для параметра **Active Solution Platform** (Платформа активного решения) установите значение **x64**. Обратите внимание, что эти параметры могут быть уже выбраны по умолчанию;
3) при необходимости выберите пункт меню **View | Solution Explorer** (Просмотр | Обозреватель решений), чтобы открыть окно **Solution Explorer** (Обозреватель решений);
4) в окне **Solution Explorer** щелкните правой кнопкой мыши по названию проекта, который нужно запустить, и выберите опцию **Set as StartUp Project** (Установить как запускаемый проект);
5) выберите пункт меню **Select Debug | Start Without Debugging** (Отладка | Запустить без отладки), чтобы запустить программу.

Некоторые примеры исходного кода ссылаются на файлы данных в разных папках, используя фиксированные пути. Чтобы запустить соответствующие исполняемые файлы с использованием структуры папок, отличной от структуры, используемой для разработки Visual Studio, вам может потребоваться изменить строки с указанием пути в исходном коде C++.

П.2.2. Создание проекта Visual Studio C ++

В этом разделе вы узнаете, как создать простой проект Visual Studio, который включает файлы исходного кода как на языке C++, так и на языке ассемблера. Ниже приведено описание основной процедуры, которая использовалась для создания примеров исходного кода в тексте данной книги. Она включает в себя следующие этапы:

- создание проекта C ++;
- включение поддержки MASM;
- добавление файла на языке ассемблера;
- настройка свойств проекта;
- редактирование исходного кода;
- сборка и запуск проекта.

Создание проекта C++

Чтобы создать проект Visual Studio C++, выполните следующие действия.

1. Запустите Visual Studio.
2. Выберите меню **File | New Project** (Файл | Новый проект).
3. В дереве элементов управления диалогового окна **New Project** (Новый проект) выберите строку **Installed | Visual C++ | Windows Desktop** (Установленные | Visual C++ | Рабочий стол Windows).
4. Выберите **Windows Console Application** (Консольное приложение Windows) в качестве типа проекта.

5. В текстовом поле **Name** (Имя) введите `Example1`.
6. В текстовом поле **Location** (Расположение) введите имя папки, в которой будет расположен проект. Вы также можете использовать кнопку **Browse** (Обзор), чтобы выбрать папку, или оставить содержимое поля без изменений, чтобы использовать расположение по умолчанию.
7. В текстовом поле **Solution** (Решение) введите `TestSolution`.
8. Убедитесь, что настройки диалогового окна **New Project** (Новый проект) такие же, как на рис. П.1 (расположение проекта может быть другим). Нажмите книпку **OK**.
9. При необходимости выберите пункт меню **View | Solution Explorer**, чтобы открыть окно **Solution Explorer**.
10. В древовидном элементе управления обозревателя решений щелкните правой кнопкой мыши текст верхнего уровня с надписью **Solution 'Example 1' (1 Project)** и выберите пункт **Rename** (Переименовать). Измените имя решения на `TestSolution`.
11. Выберите пункт **Build | Configuration Manager**. В диалоговом окне **Configuration Manager** выберите **<Edit ...>** (<Редактировать>) в разделе **Active Solution Platforms** (рис. П.2).
12. В диалоговом окне **Edit Solution Platforms** (Редактировать платформы решения) выберите **x86** и нажмите **Remove** (Удалить) (рис. П.3). Нажмите **Close** (Закрыть), чтобы закрыть диалоговое окно **Edit Solutions Platforms**; затем еще раз нажмите **Close**, чтобы закрыть диалоговое окно **Configuration Manager**.

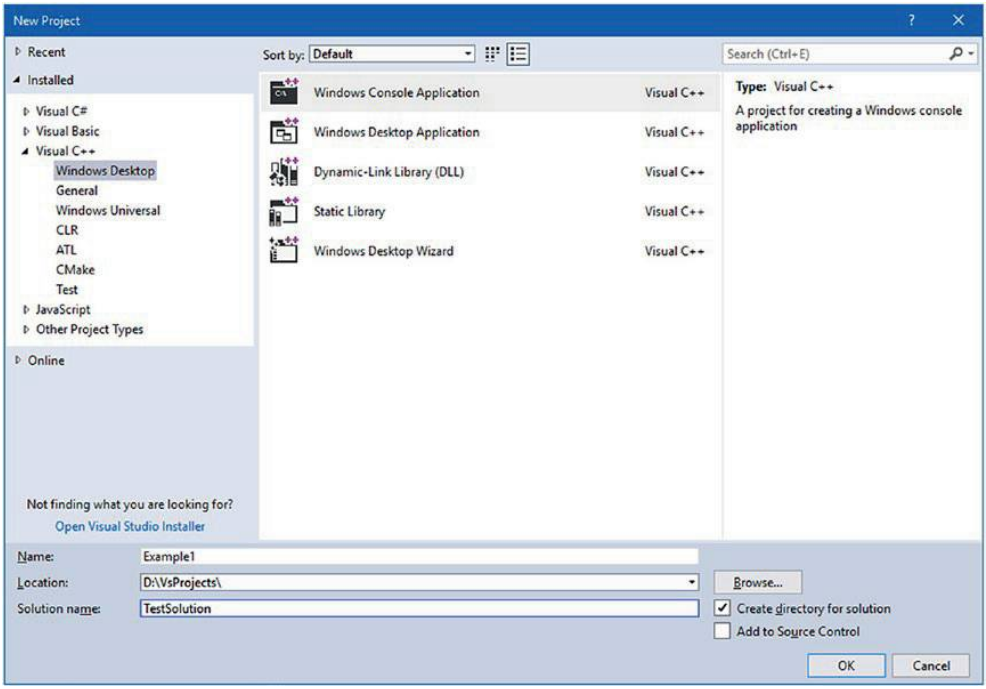


Рис. П.1. Диалоговое окно **New Project**

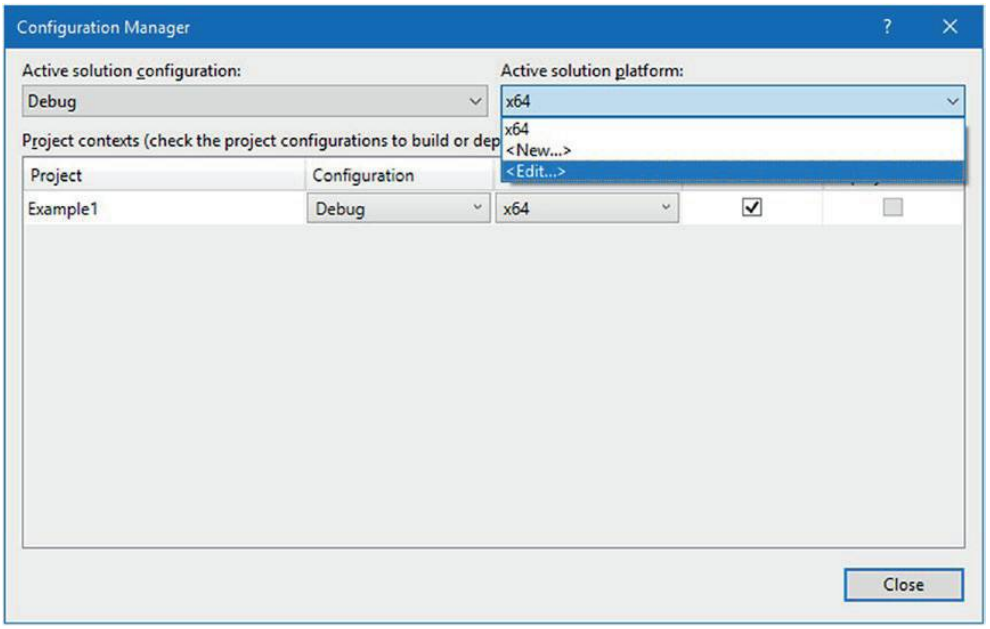


Рис. П.2. Диалоговое окно **Configuration Manager**

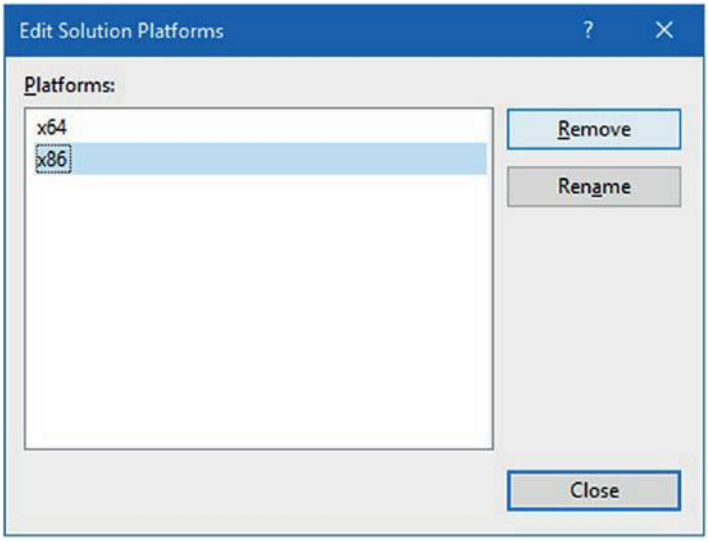


Рис. П.3. Диалоговое окно **Solution Platforms**

Включение поддержки MASM

Чтобы включить поддержку Microsoft Macro Assembler, выполните следующие действия.

1. В древовидном элементе управления обозревателя решений щелкните правой кнопкой мыши **Example1** и выберите **Build Dependencies | Build Customizations** (Зависимости сборки | Настройки сборки).

2. В диалоговом окне **Visual C++ Build Customizations** (Настройки сборки Visual C++) установите флажок **masm (.targets, .props)**.
3. Нажмите **OK**.

Добавление файла на языке ассемблера

Чтобы добавить файл исходного кода на языке ассемблера (`.asm`) в проект Visual Studio C++, выполните следующие действия.

1. В древовидном элементе управления Solution Explorer нажмите правой кнопкой мыши на имени проекта **Example1** и выберите пункт **Add | New Item** (Добавить | Новый элемент).
2. Выберите опцию **C++ File (.cpp)** в качестве типа файла.
3. В текстовом поле **Name** (Имя) измените имя на `Example1_.asm`, как показано на рис. П.4. Обратите внимание, что завершающее подчеркивание является обязательным, поскольку все файлы исходного кода C++ и языка ассемблера в проекте должны иметь уникальное базовое имя.
4. Нажмите кнопку **Add** (Добавить).

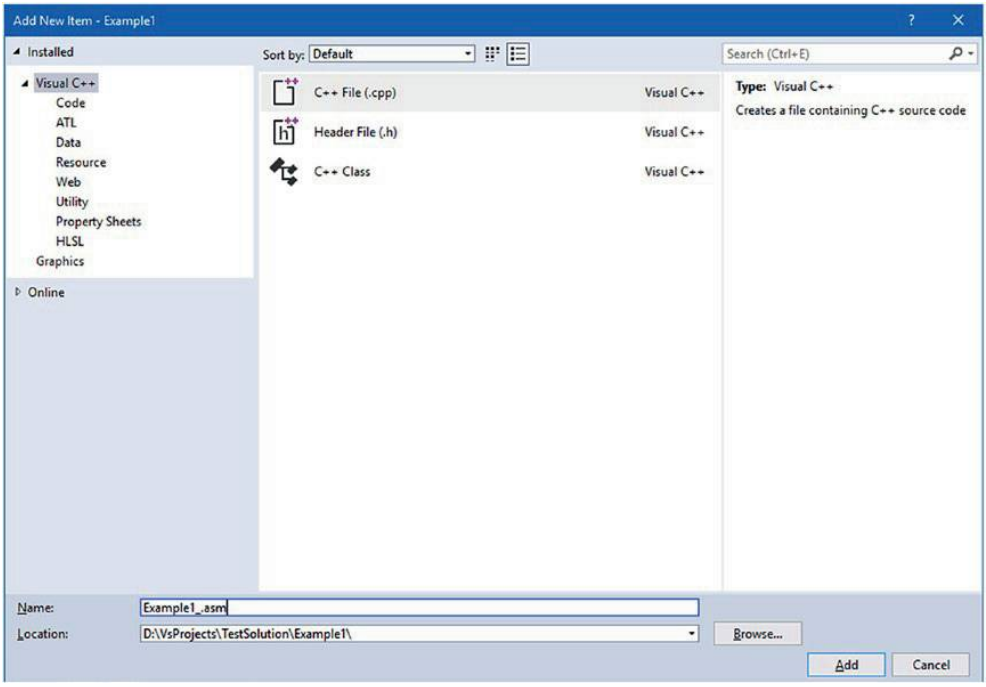


Рис. П.4. Диалоговое окно **Add New Item**

Настройка свойств проекта

Выполните следующие шаги, чтобы настроить свойства проекта. Опции, управляющие генерацией файла листинга (шаги 5–8), не являются обязательными.

1. В древовидном элементе управления окна Solution Explorer нажмите правой кнопкой мыши на **Example1** и выберите **Properties** (Характеристики).

2. В диалоговом окне **Property Pages** (Страницы свойств) измените параметр **Configuration** (Конфигурация) на **All Configurations** (Все конфигурации), а параметр **Platform** (Платформа) – на **All Platforms** (Все платформы). Обратите внимание, что один или оба параметра могут быть уже установлены.
3. В древовидном элементе управления выберите **Configuration Properties | General** (Свойства конфигурации | Общий). Измените настройку **Whole Program Optimization** (Оптимизация всей программы) на **No Whole Program Optimization** (Без оптимизации всей программы) (рис. П.5).
4. Выберите меню **Configuration Properties | C/C++ | Code Generation** (Свойства конфигурации | C/C++ | Генерация кода). Измените параметр **Enable Enhanced Instruction Set** (Включить расширенный набор команд) на **Advanced Vector Extensions (/arch:AVX)** (рис. П.6).
5. Выберите меню **Configuration Properties | C/C++ | Output Files** (Свойства конфигурации | C/C++ | Выходные файлы). Измените настройку **Assembler Output** (Вывод ассемблера) на **Assembly Machine and Source Code (/FAcs)** (рис П.7).
6. Выберите меню **Configuration Properties | Microsoft Macro Assembler | Listing File** (Свойства конфигурации | Ассемблер макросов Microsoft | Файл листинга). Измените параметр **Enable Assembly Generated Code Listing** (Включить листинг кода, созданного ассемблером) на **Yes (/Sg)** (рис. П.8).
7. Измените содержимое текстового поля **Assembled Code Listing File** (Файл листинга собранного кода) на строку `$(IntDir)\%(filename).lst` (рис. П.8). Этот шаблон строки указывает на промежуточный каталог проекта, который является подпапкой основной папки проекта.
8. Нажмите **OK**.

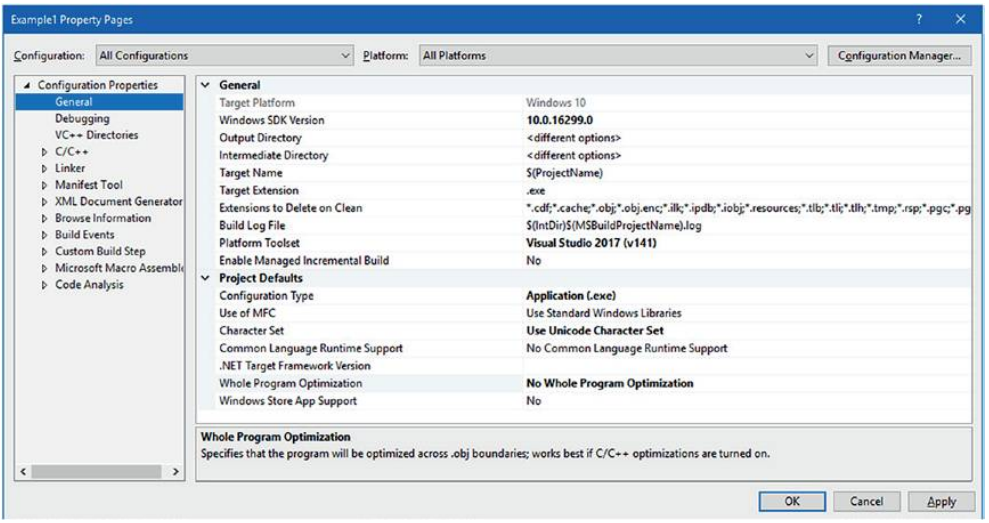


Рис. П.5. Диалоговое окно **Property Pages** (Whole Program Optimization)

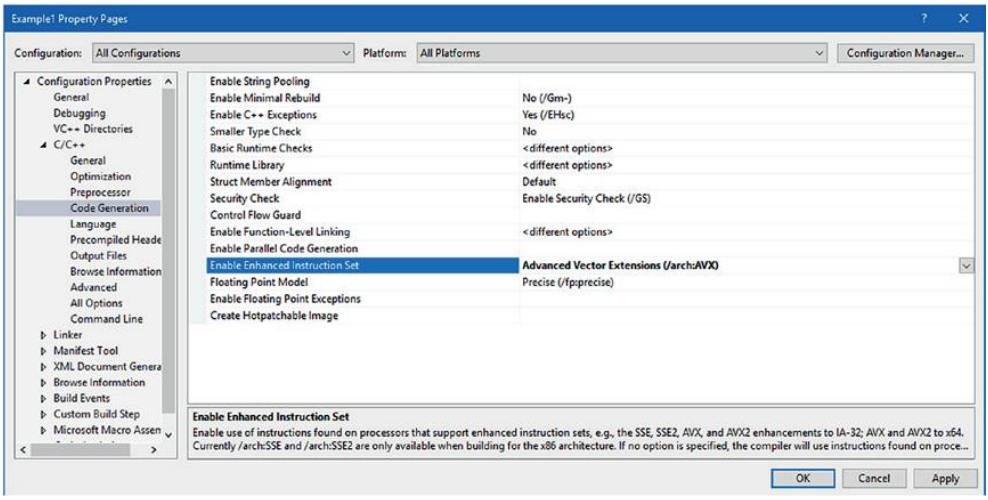


Рис. П.6. Диалоговое окно **Property Pages** (Enable Enhanced Instruction Set)

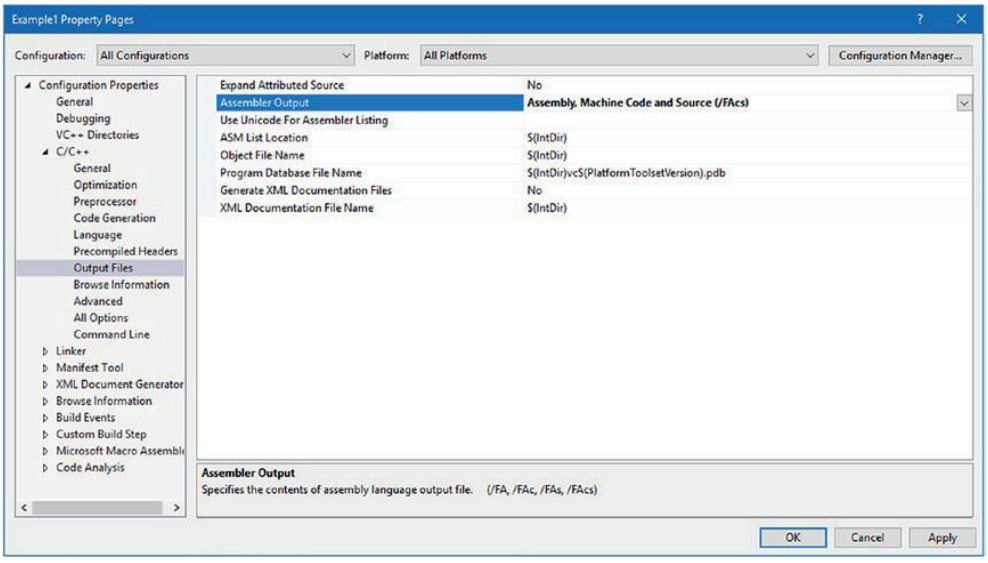


Рис. П.7. Диалоговое окно **Property Pages** (Assembler Output)

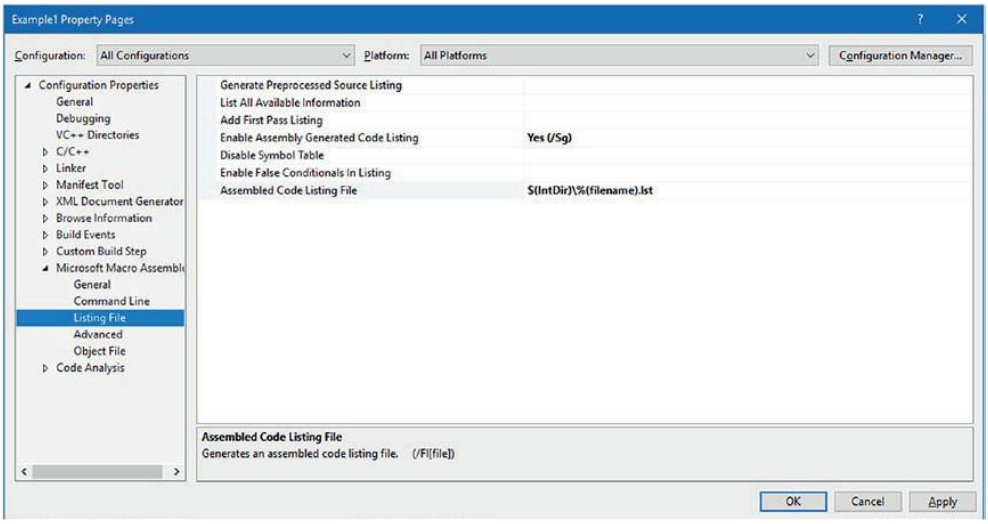


Рис. П.8. Диалоговое окно **Property Pages** (Microsoft Macro Assembler Listing File)

Редактирование исходного кода

Для редактирования исходного кода проекта выполните следующие действия.

1. В окне редактора щелкните по вкладке с именем **Example1.cpp**.
2. Отредактируйте исходный код на языке C++, чтобы он соответствовал коду, показанному в листинге П.1.
3. Щелкните по вкладке с именем **Example1_.asm**.
4. Отредактируйте исходный код на языке ассемблера, чтобы он соответствовал коду, показанному в листинге П.2.
5. Выберите пункт меню **File | Save All** (Файл | Сохранить все).

Листинг П.1. Example1.cpp

```
// Example1.cpp : Начальный код консольного приложения.
//

#include "stdafx.h"
#include <iostream>

using namespace std;

extern "C" int CalcResult1_(int val1, int val2, int* quo, int* rem);

int main()
{
    int val1 = 42;
    int val2 = 9;
    int quo;
    int rem;
    int prod = CalcResult1_(val1, val2, &quo, &rem);

    cout << "Результаты Example1\n";
    cout << "val1 = " << val1 << '\n';
```

```
    cout << "val2 = " << val2 << '\n';
    cout << "quo = " << quo << '\n';
    cout << "rem = " << rem << '\n';
    cout << "prod = " << prod << '\n';
    return 0;
}
```

Листинг П.2. Example1_.asm

```
; extern "C" int CalcResult1_(int val1, int val2, int* quo, int* rem);

        .code
CalcResult1_ proc
        mov r10d,ecx                ;r10d = val1
        mov r11d,edx                ;r11d = val2

        mov eax,ecx                 ;eax = val1
        cdq                         ;edx:eax = val1

        idiv r11d                   ;calc val1 / val2
        mov dword ptr [r8],eax      ;сохранение частного
        mov dword ptr [r9],edx      ;сохранение остатка

        imul r10d,r11d              ;r10d = val1 * val2
        mov eax,r10d                ;eax = val1 * val2
        ret
CalcResult1_ endp
        end
```

Сборка и запуск проекта

Для сборки и запуска проекта выполните следующие шаги.

1. Выберите пункт меню **Build | Build Solution** (Сборка | Сборка решения).
2. При необходимости исправьте все обнаруженные ошибки компилятора C++ или MASM и повторите шаг 1.
3. Выберите пункт **Select Debug | Start Without Debugging**.
4. Убедитесь, что вывод соответствует окну консоли, показанному на рис. П.9.
5. Нажмите **Enter**, чтобы закрыть окно консоли.

Рис. П.9. Вывод окна консоли

П.3. Основные и дополнительные материалы

В этом разделе содержится список основных источников информации, которые использовались при подготовке текста данной книги. Он также включает дополнительные ссылки и ресурсы, в которых вы можете найти полезную для себя информацию. Ссылки сгруппированы по следующим категориям:

- справочные руководства по программированию для x86;
- справочные материалы по x86 и микроархитектуре;
- вспомогательные ресурсы;
- ссылки на ресурсы по алгоритмам;
- ссылки на ресурсы по C++.

П.3.1. Справочные руководства по программированию на X86

Ниже приводится список справочных руководств по программированию x86, опубликованных AMD и Intel:

AMD64 Architecture Programmer's Manual Volume 1: Application Programming https://support.amd.com/TechDocs/24592.pdf;

AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions, https://support.amd.com/TechDocs/24594.pdf;

AMD64 Architecture Programmer's Manual Volume 4: 128-bit and 256-bit Media Instructions, https://support.amd.com/TechDocs/26568.pdf;

Software Optimization Guide for AMD Family 17h Processors, Publication Number 55723, June 2017, https://developer.amd.com/resources/developerguides-manuals;

Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4, https://www.intel.com/content/www/us/en/processors/architectures-software-developermanuals.html;

Intel 64 and IA-32 Architectures Optimization Reference Manual, https://www.intel.com/content/www/us/en/processors/architectures-softwaredeveloper-manuals.html;

Intel Architecture Instruction Set Extensions and Future Features Programming Reference, https://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

П.3.2. Справочные материалы по x86 и микроархитектуре

Следующие ресурсы содержат обширную информацию о программировании на языке ассемблера x86, процессорах и микроархитектурах:

Guy Ben-Haim, Itai Neoran, and Ishay Tubi, Practical Intel AVX Optimization on 2nd Generation Intel Core Processors, https://software.intel.com/sites/default/files/m/d/4/1/d/8/Practical_Optimization_with_AVX.pdf;

Ian Cutress, The Intel Skylake Mobile and Desktop Launch, with Architecture Analysis, September 2015, https://www.anandtech.com/show/9582/intelskylake-mobile-desktop-launch-architecture-analysis;

Ian Cutress, The Intel Skylake-X Review: Core i9-7900X, i7-7820X and i7-7800X Tested, June 2017, https://www.anandtech.com/show/11550/the-intelskylakex-review-core-i9-7900x-i7-7820x-and-i7-7800x-tested;

Anger Fog, The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers, August 2018, https://agner.org/optimize/#manuals;

Agner Fog, Optimizing subroutines in assembly language: An optimization guide for x86 platforms, April 2018, https://agner.org/optimize/#manuals;

Chris Kirkpatrick, Intel AVX State Transitions: Migrating SSE Code to AVX, https://software.intel.com/en-us/articles/intel-avx-statetransitions-migrating-sse-code-to-avx;

Patrick Konsor, Avoiding AVX-SSE Transition Penalties, https://software.intel.com/en-us/articles/avoiding-avx-sse-transition-penalties;

Patrick Konsor, Performance Benefits of Half-Precision Floats, https://software.intel.com/en-us/articles/performance-benefits-of-half-precisionfloats;

Daniel Kusswurm, Modern x86 Assembly Language Programming, Apress, ISBN 978-1-4842-0065-0, 2014;

Max Locktyukhin, How to Detect New Instruction Support in the 4th Generation Intel Core Processor Family, August 2013, https://software.intel.com/en-us/node/405250;

John Morgan, Microsoft Visual Studio 2017 Supports Intel AVX-512, https://blogs.msdn.microsoft.com/vcblog/2017/07/11/microsoft-visualstudio-2017-supports-intel-avx-512;

Erdinc Ozturk, James Guilford, Vinodh Gopal, and Wajdi Feghal, New Instructions Supporting Large Integer Arithmetic on Intel Architecture Processors, August 2012, https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-large-integer-arithmetic-paper.pdf;

James Reinders, AVX-512 May Be a Hidden Gem in Intel Xeon Scalable Processors, June 2017, https://www.hpcwire.com/2017/06/29/reinders-avx-512-mayhidden-gem-intel-xeon-scalable-processors;

Anand Lal Shimpi, Intel's Haswell Architecture Analyzed: Building a New PC and a New Intel, October 2012, http://www.anandtech.com/show/6355/intelshaswell-architecture.

П.3.3. Вспомогательные ресурсы

Следующие ресурсы содержат полезную информацию о процессорах x86 и микроархитектурах:

Processors for Desktops, AMD, https://www.amd.com/en/products/processorsdesktop;

List of AMD Accelerated Processing Unit Microprocessors, Wikipedia, https://en.wikipedia.org/wiki/List_of_AMD_Accelerated_Processing_Unit_microprocessors;

List of AMD CPU Microarchitectures, Wikipedia, https://en.wikipedia.org/wiki/List_of_AMD_CPU_microarchitectures;

List of AMD Microprocessors, Wikipedia, https://en.wikipedia.org/wiki/List_of_AMD_processors;

Product Information Website, Intel, https://ark.intel.com;

List of Intel CPU Microarchitectures, Wikipedia, https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures;

List of Intel Microprocessors, Wikipedia, https://en.wikipedia.org/wiki/Intel_processor;

List of Intel Xeon Microprocessors, Wikipedia, https://en.wikipedia.org/wiki/List_of_Intel_Xeon_microprocessors;

Register Renaming, Wikipedia, https://en.wikipedia.org/wiki/Register_renaming.

П.3.4. Ссылки на ресурсы по алгоритмам

Следующие ресурсы использовались для разработки алгоритмов, используемых в примерах исходного кода:

S. Acton, REAL Computing Made REAL – Preventing Errors in Scientific and Engineering Calculations, ISBN 978-0486442211, Dover Publications, 2005;

Tony Chan, Gene Golub, Randall LeVeque, Algorithms for Computing the Sample Variance: Analysis and Recommendations, The American Statistician, Volume 37 Number 3 (1983), p. 242–247;

James F. Epperson, An Introduction to Numerical Methods and Analysis, Second Edition, ISBN 978-1-118-36759-9, Wiley, 2013;

David Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic, ACM Computing Surveys, Volume 23 Issue 1 (March 1991), p. 5–48;

Rafael C. Gonzalez and Richard E. Woods, Digital Image Processing, Fourth Edition, ISBN 978-0-133-35672-4, 2018;

James E. Miller, David G. Moursund, Charles S. Duris, Elementary Theory & Application of Numerical Analysis, Revised Edition, ISBN 978-0486479064, Dover Publications, 2011;

Anthony Pettofrezzo, Matrices and Transformations, ISBN 0-486-63634-8, Dover Publications, 1978;

Hans Schneider and George Barker, Matrices and Linear Algebra, ISBN 0-486-66014-1, Dover Publications, 1989;

Eric W. Weisstein, Convolution, MathWorld, http://mathworld.wolfram.com/Convolution.html;

Eric W. Weisstein, Correlation Coefficient, MathWorld, http://mathworld.wolfram.com/CorrelationCoefficient.html;

Eric W. Weisstein, Cross Product, MathWorld, http://mathworld.wolfram.com/CrossProduct.html;

Eric W. Weisstein, Least Squares Fitting, MathWorld, http://mathworld.wolfram.com/LeastSquaresFitting.html;

Eric W. Weisstein, Matrix Multiplication, MathWorld, http://mathworld.wolfram.com/MatrixMultiplication.html;

David M. Young and Robert Todd Gregory, A Survey of Numerical Mathematics,Volume 1, ISBN 0-486-65691-8, Dover Publications, 1988;

Algorithms for calculating variance, Wikipedia, https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance;

Body Surface Area Calculator, http://www.globalrph.com/bsa2.htm;

Grayscale, Wikipedia, https://en.wikipedia.org/wiki/Grayscale;

Linked List, Wikipedia, https://en.wikipedia.org/wiki/Linked_list.

П.3.5. Ссылки на ресурсы по C ++

Следующие ресурсы содержат ценную информацию о программировании на C++, стандартных библиотеках шаблонов C++ и программировании на C++ с использованием нескольких потоков:

Ivor Horton, Using the C++ Standard Template Libraries, Apress, ISBN 978-1-4842-0005-6, 2015;

Nicolai M. Josuttis, The C++ Standard Library – A Tutorial and Reference, Second Edition, Addison Wesley, ISBN 978-0-321-62321-8, 2012;

Bjarne Stroustrup, The C++ Programming Language, Fourth Edition, Addison Wesley, ISBN 978-0-321-56384-2, 2013;

Anthony Williams, C++ Concurrency in Action – Practical Multithreading, ISBN 978-1-933-98877-1, Manning Publications, 2012;

cplusplus.com, http://www.cplusplus.com.

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Даниэль Куссвюрм

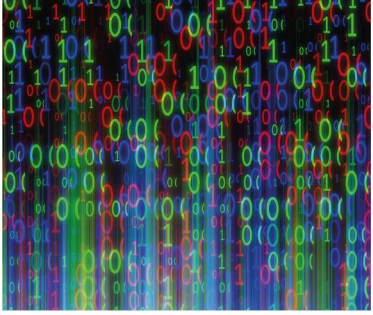
Профессиональное программирование на ассемблере x64

с расширениями AVX, AVX2 и AVX-512

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Яценков В. С.*
Корректор *Синяева Г. И.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 51,03. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com



Профессиональное программирование **на ассемблере x64**

Изучите язык ассемблера x64, сосредоточившись на обновлениях набора команд x86, наиболее актуальных для разработки прикладных программ.

Рассматриваемые темы:

- 64-разрядная платформа x86: архитектура, типы данных, регистры, режимы адресации памяти и базовый набор команд;
- набор команд x86 для создания быстродействующих функций, которые можно вызывать из языка высокого уровня (C++);
- использование языка ассемблера x64 для эффективной работы с общими типами данных и конструкциями программирования, включая целые числа, текстовые строки, массивы и структуры;
- использование набора команд AVX для выполнения скалярных арифметических операций с плавающей запятой;
- повышение быстродействия ресурсоемких алгоритмов в проблемных областях, таких как обработка изображений, компьютерная графика, математика и статистика, за счет команд AVX, AVX2 и AVX-512;
- применение различных стратегий и методов кодирования, а также наборов команд x64, AVX, AVX2 и AVX-512 для достижения максимального быстродействия.

Большая часть материала не привязана к какой-либо конкретной операционной системе. Примеры исходного кода разработаны с использованием Visual Studio C++ и MASM; для их запуска рекомендуется ПК на базе ОС Windows 10 и процессором, поддерживающим AVX.

Предполагается, что читатели имеют опыт программирования на языках высокого уровня и базовые знания C++.

Книга предназначена разработчикам, которые хотят научиться писать код с использованием языка ассемблера 64.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК "Галактика"
books@alians-kniga.ru

Apress

ДМК ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-97060-928-6