Оглавление

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.

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.

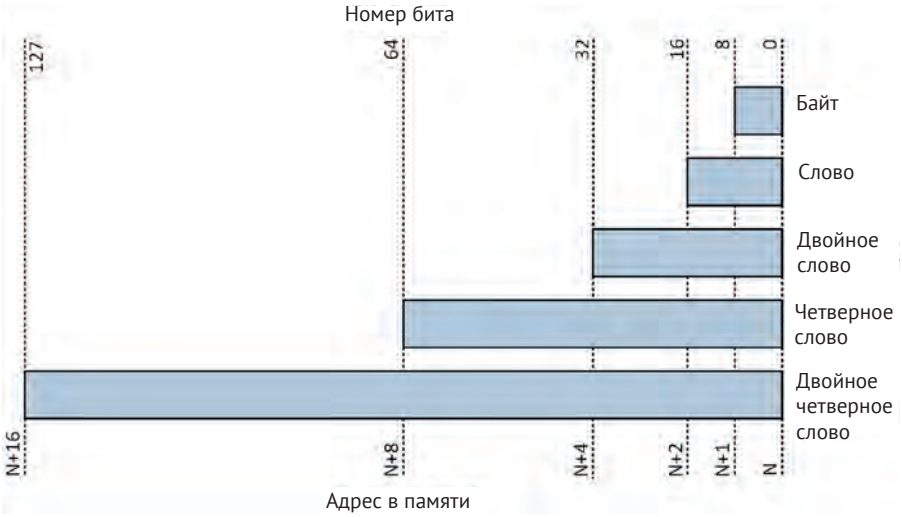

Рис. 1.1. Нумерация битов и порядок байтов для основных типов данных

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.

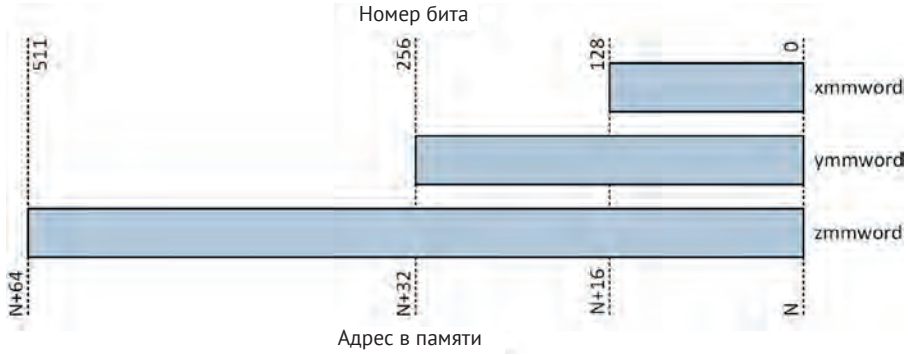

Рис. 1.2. Типы данных SIMD

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.

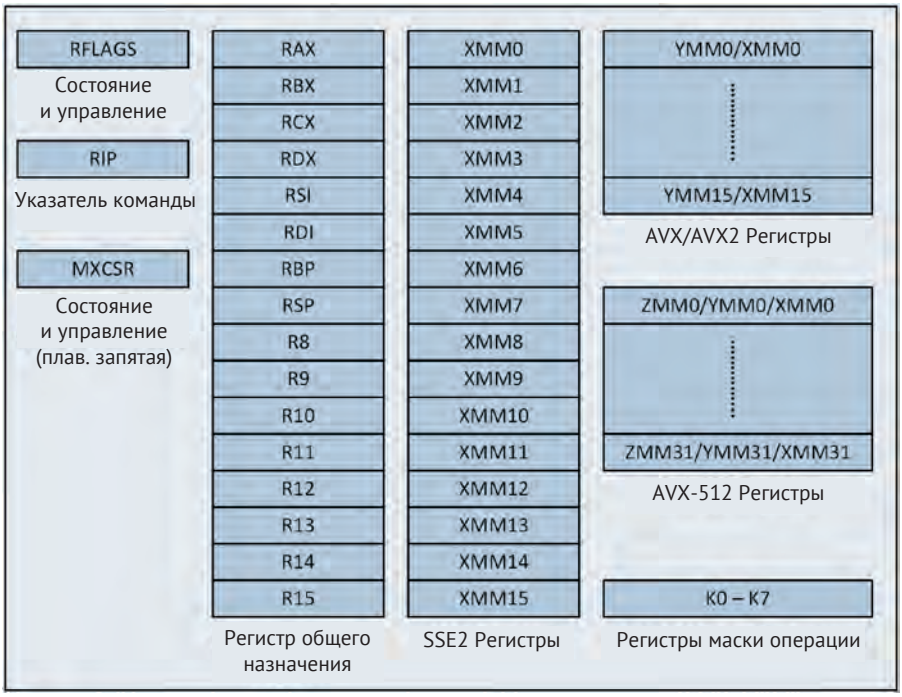

Рис. 1.3. Внутренняя архитектура процессора X86-64

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.

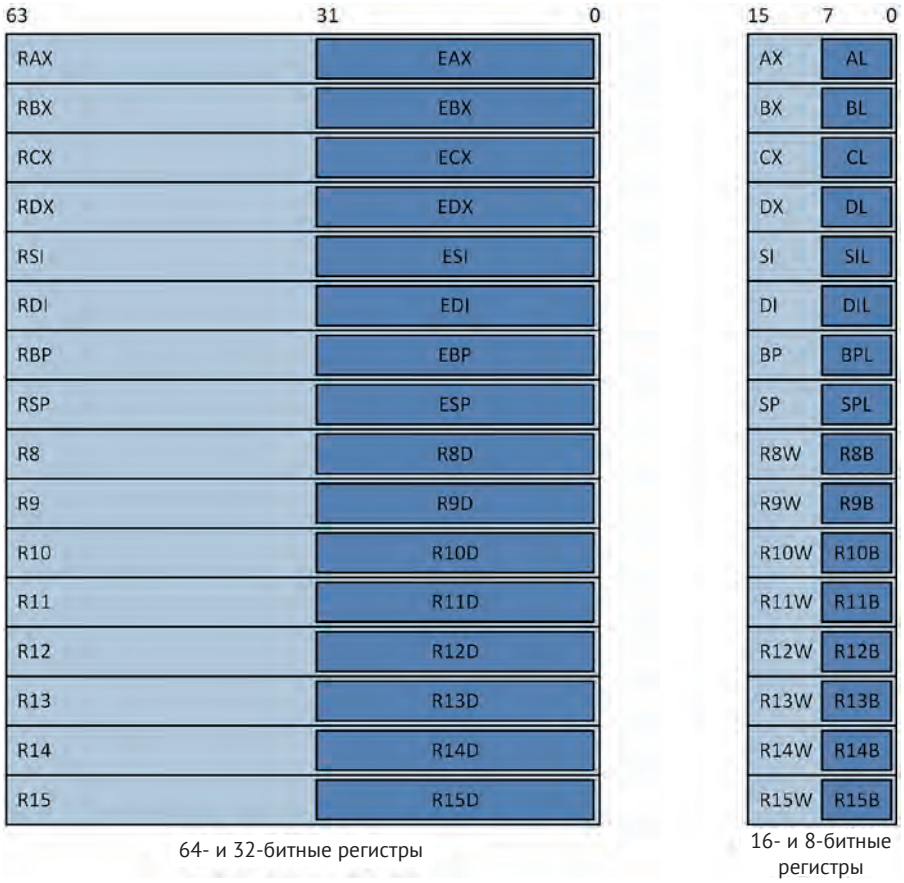

Рис. 1.4. Регистры общего назначения x86-64

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.

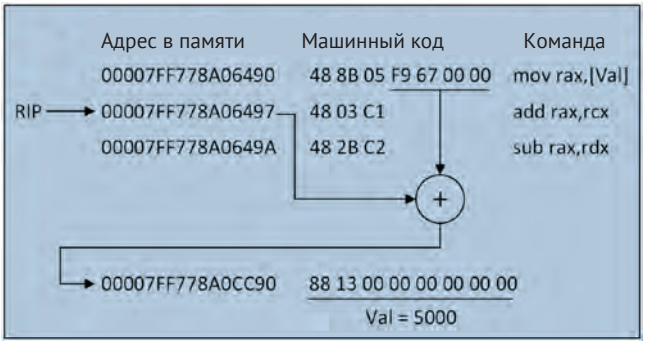

Рис. 1.5. Вычисление эффективного адреса относительно RIP

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.