Глава 5. Архитектура компьютеров x86 фирмы Intel

Архитектура x86 – это победа маркетинга над здравым смыслом.
Старожил Кремниевой долины

Нет процессора, кроме x86-x64 и фон Нейман – пророк его.

Компьютеры фирмы Intel x86-x64 являются одним из старейших, они до сих пор непрерывно развиваются и совершенствуются.

5.1. Понятие семейства ЭВМ

«Группа ЭВМ, представляющих параметрический ряд, имеющих единую архитектуру и, в большинстве случаев, одинаковую конструктивно-технологическую базу и характеризующихся полной или ограниченной некоторыми условиями программной совместимостью».

ГОСТ 15971–90

Как известно, компьютеры могут применяться в самых различных сферах человеческой деятельности (как говорится, в различных *предметных областях*). В качестве примеров можно привести область научно-технических расчётов (там много операций с вещественными числами, и многомерными массивами), область экономических расчётов (там, в основном, выполняются операции над целыми числами, и производится обработка символьной информации), мультимедийная область (обработка звука, изображения и т.д.), область управления сложными устройствами (ракетами, доменными печами и др.).

Как уже упоминалось, компьютеры, архитектура которых в основном ориентирована на какую-то одну предметную область, называются *специализированными*, в отличие от *универсальных* ЭВМ, которые более или менее успешно можно использовать в большинстве предметных областях. В этой книге изучается архитектура только универсальных ЭВМ.

Говорят, что компьютеры образуют *семейство* (Computer Family, Computer Series) если выполняются следующие требования.

1. Одновременно выпускаются и используются несколько *моделей* семейства с различными производительностью и ценой (моделями называются компьютеры-члены семейства). Таким образом, пользователь может выбирать между дешёвыми моделями с относительно небольшими аппаратными возможностями, и более дорогими моделями с бóльшей производительностью.
2. Все модели семейства обладают *программной совместимостью снизу-вверх* – старшие модели поддерживают все команды младших (любая программа, написанная для младшей модели, безошибочно выполняется и на старшей модели). Это свойство называется ещё *обратной совместимостью*.
3. Присутствует *унификация* внешних устройств (периферии), то есть их аппаратная совместимость между моделями (например, печатающее устройство должно работать на всех выпускаемых в настоящее время моделях семейства). Значит, все внешние устройства должны сохранять «умение» работать в старых, примитивных режимах.

Не надо понимать свойства 2 и 3 слишком буквально. При выполнении старых программ ЭВМ новых моделей семейства либо с самого начала работают в этом старом режиме (и со старыми драйверами устройств!), либо умеют временно переключаться в специальные режимы работы, в которых *аппаратно эмулируют* работу компьютеров старых моделей. Например, для выполнения самых первых 16-битных программ современные процессоры Intel переключаются в режим *виртуальном машины* MS-DOS (V86 Mode), включают на новых внешних устройствах старые режимы и протоколы работы, и заставляют новые драйверы «вспомнить», как работать с такими устройствами. Аналогично, хотя в 64-битном режиме в ЭВМ уже нет некоторых машинных команд 32-битного режима, но

они умеют переключаться в так называемый режим совместимости (compatibility mode), эмулирующий работу 32-битных машин.

Отсюда следует, что и все новые устройства (процессоры, принтеры, мониторы и т.д.) тоже должны уметь «вспоминать детство» и работать в старых режимах. Например, все процессоры Intel при включении начинают работать в режиме первых 16-битных ЭВМ, а все мониторы при включении начинают сначала работать в текстовом режиме 25 строк по 80 символов в строке (так называемом VGA-режиме). Как видим, это весьма сложно, однако внешне всё выглядит так, что новый компьютер выполняет старые программы.

Программная совместимость дорого обходится современным процессорам, усложняя их архитектуру. В последнее время появилась тенденция убрать из новых процессоров аппаратную совместимость со старыми моделями. Например, фирма Intel в 2023 году выпустила процессор X86S, который работает только в 64-разрядном режиме, выполняя старые программы только в режиме программной эмуляции.

Что касается внешних устройств, то сейчас фирмы-производители с самого начала проектируют их так, чтобы они могла (с соответствующими драйверами) работать с компьютерами, выпускающимися всеми производителями, а не только с ЭВМ конкретного семейства.

4. Модели семейства организованы по принципу *модульности*, что позволяет в определённых пределах расширять возможности ЭВМ, увеличивая, например, объём памяти, качество обработки графических данных или повышая производительность путём замены центрального процессора на более быстродействующий.
5. Стандартизировано программное обеспечение (например, текстовый редактор и интернет-браузер должны работать на всех выпускаемых моделях семейства).

Большинство выпускаемых в наше время ЭВМ содержатся в каких-либо семействах. Вообще говоря, можно было бы ещё потребовать, чтобы все модели семейства выпускались одной фирмой. В то же время часто случается, что некоторая другая фирма начинает выпускать своё семейство ЭВМ, *программно совместимое* с уже выпускающимся семейством. В качестве примера можно привести семейство ЭВМ фирмы Intel и семейство ЭВМ фирмы AMD. Важно понять, что такие ЭВМ, выпускаемые разными фирмами, одинаковы на *внутреннем* уровне ви́дения архитектуры (например, при программировании на языках низкого уровня), но их архитектура различна на *инженерном* уровне. В этой книге для упрощения изложения будут рассматриваться, в основном 32-битные, модели персональных компьютеров x86 семейства ЭВМ компании Intel. В таблице 5.1 представлены некоторые модели семейства фирмы Intel.

Таблица 5.1. Семейство процессоров фирмы Intel

Название	Год Выпуска	Макс. тактовая частота, Ггц	Транзисторов ЦП, млн.	Размер регистров, бит	Ширина шины данных, бит	Адресное пространство	Проектная ширина, мкм
i8086	1978	0.010	0.029	16	16	1 Мб	3
i80286	1982	0.016	0.134	16	16	16 Мб	1.5
i80386	1985-92	0.016-0.033	0.275	32	32	4 Гб	1.5-1.0
i80486	1989-94	0.025-0.100	1.2	32	32	4 Гб	1.0-0.6
P5 (Pentium)	1993-96	0.06-0.233	3.1	32	64	4 Гб	0.8-0.35
P6 (Pentium Pro)	1995-97	0.150-0.20	5.5	32	64	64 Гб	0.6-0.35
Pentium II	1997	0.233-0.45	7.5	32	64	64 Гб	0.25-0.18
Celeron	1998-02	0.266-2.2	18.9	32	64	64 Гб	0.25-0.13
Pentium III	1999-02	0.45-1.2	28	32	64	64 Гб	0.18-0.13
Pentium 4	2000-02	1.4-3.0	42	32	2x64	64 Гб	0.18-0.13
Pentium D	2005	2.8-3.4	230	64	64	64 Гб	0.09-0.065
Intel Core	2006	1.5-2.33	250	64	64	64 Гб	0.065
Xeon	2006-07	1.6-3.0	220-300	64	128	64 Гб	0.065-0.045
Intel Core i3	2006-07	1.5-2.33	410-820	64	64	64 Гб	0.065-0.045
Intel Core i5	2009-13	2.66-2.8	731-995	64	64	64 Гб	0.045-0.032
Intel Core i7	2010-17	2.66-3.2	1300	64	64	64 Гб	0.032-0.014
Intel Core i9	2017-20	3.3-4.5	1900	64	64	128 Гб	0.014+

Одной из главных особенностей семейства ЭВМ следует считать программную совместимость, которая позволяет гарантировать, что все разработанные ранее программы будут правильно и без переделок выполняться и на всех последующих моделях ЭВМ этого семейства. Это требование должно соблюдаться по чисто экономическим соображениям, так как стоимость уже разработанного *программного* обеспечения в настоящее время сопоставима со стоимостью всего *аппаратного* обеспечения, а часто и превосходит его. В то же время требования учитывать в новых моделях семейства все те устаревшие архитектурные решения, которые были приняты ранее, становится для разработчиков всё более тягостной и трудноразрешимой задачей.

Ясно, что такое положение вещей не сможет долго продолжаться, и рано или поздно от принципа программной совместимости на внутреннем уровне придётся отказаться. Новые модели необходимо строить по самым современным архитектурным схемам, в частности, учитывающим глубокий параллелизм в обработке данных. В то же время, нельзя и потерять возможность выполнять старые программы для предшествующих моделей семейства.

Очевидно, эту проблему можно попытаться решить следующим способом. Новые процессоры будут иметь совершенно другую архитектуру и, следовательно, другую систему команд, однако предусмотрена их работа в двух режимах. В основном режиме процессор может выполнять команды только своего нового машинного языка, однако во вспомогательном режиме он имеет возможность аппаратно *интерпретировать* (полностью имитировать выполнение) программ на языке машины предыдущих моделей семейства. Разумеется, интерпретация значительно (в несколько раз) снижает скорость выполнения старых программ. Основную надежду здесь возлагают на то, что старые программы, написанные на языках *высокого уровня*, могут быть достаточно легко исправлены так, чтобы быть заново откомпилированы уже на новый машинный язык. Кроме того, возможность значительно ускорить выполнение своих программ, перейдя на новую архитектуру, должна быть хорошим стимулом для программистов. Ну, а всем остальным «не передовым» пользователям можно гарантировать, что все их старые программы на новых моделях будут считаться не медленнее, чем на старых, даже в режиме интерпретации (за счёт повышения вычислительной мощности новых процессоров). Этот процесс перехода на принципиально новую архитектуру, однако, идёт крайне медленно, так старая архитектура продолжает оставаться эффективной из-за непрерывного прогресса в аппаратуре ЭВМ. Так что пока новая архитектура не получила широкого распространения.

Сейчас пора перейти к изучению архитектуры наиболее распространённой в настоящее время 32-битной модели семейства Intel. По ходу изложения материала будут приводиться архитектурные отличия от следующих 64-битных моделей. В этой главе будут последовательно рассмотрены устройство памяти, форматы обрабатываемых данных и работа процессора этой ЭВМ.

5.2. Память

640 Kb памяти должно быть достаточно для любого [компьютера].

Билл Гейтс, 1981 г.

Рассматриваемый компьютер имеет архитектуру с адресуемыми регистрами, поэтому адресуемая память состоит из основной и регистровой. Основная *адресуемая* память имеет объём 2^{32} ячеек длиной по 8 бит, это 4 гигабайта (Гб), при этом каждая команда или данное располагаются в одной или нескольких последовательных (с возрастающими адресами) ячейках этой памяти.

Физической памяти на компьютере может быть и больше, например, 128 Гб, т.е. длина адреса при обращении к физической памяти будет не 32, а 37 бит. В то же время каждая программа, работая в так называемом плоском режиме памяти, может использовать не более 4 Гб. На компьютере, однако, могут одновременно выполняться несколько программ, занимая всю физическую память.

В 2003 году появились первые 64-разрядные процессоры, которые могли адресовать память до 2^{64} байт. Это очень много, физически ни один компьютер не может иметь такую память. Фактически современные ЭВМ имеют в шинах не более 48-ми адресных линий и могут иметь объём памяти до 2^{48} = 256 Тб. При этом современные операционные системы обычно отдают каждой программе пользователя 8 Тб, ещё 8 Тб берёт себе сама операционная система, хотя Windows и предусматривает расширение памяти до 2^{53} байт.

Адресуемая регистровая память образует несколько независимых адресных пространств. Во-первых, это 8 (в 64-битных моделях 16) целочисленных регистров, на них могут хранится и обраба-

тываться целые числа и адреса. Устройство этой памяти будет подробно рассмотрено немного позже. Во-вторых, это уже упомянутые ранее 8 регистров для работы с вещественными числами, они образуют специфическую структуру – кольцевой стек, подробно работа с вещественными числами нами изучаться не будет, но примеры использования приведены в разделе 8.9.1. И, наконец, это пространство *векторных* регистров, они появились на наших ЭВМ в 2011 году, с ними работают *трёхадресные* и *четырёхадресные* команды. Эти регистры универсальны, они могут работать с векторами как целых, так и вещественных чисел. Соответствующие типы данных называются упакованными целыми (packed integer) и упакованными вещественными (packed floating), желающие могут ознакомиться с ними в главе 17. Отдельно стоит линейное пространство портов ввода/вывода, их мы будем рассматривать в главе 14. Как видим, всё весьма сложно 🙂.

5.3. Форматы данных

Информация – это обозначение содержания, черпаемого нами из внешнего мира в процессе нашего приспособления к нему и приведения в соответствие с ним нашего мышления.

Норберт Винер

Данные принято определять как *информацию*,[1] представленную в формализованном виде и пригодную для хранения, передачи и обработки человеком или автоматическими системами обработки [ГОСТ 34.321-96, ISO/IEC 10032:1995]. Далее рассматриваются большинство форматов данных, для которых в языке нашей машины предусмотрены обрабатывающие их *команды*. Все остальные форматы (типы) данных, такие, как, например, записи и множества языка Паскаль, динамические структуры данных (очереди, списки, деревья и т.д.) придётся моделировать (отображать их на машинные форматы).

- **Вещественные числа**

Первыми ЭВМ, которые могли выполнять операции над вещественными числами (floating-point numbers), была машина Стрела, выпущенная в СССР в 1953 году (она работала с 43-разрядными числами) и выпущенная в 1954 году машина IBM-701, работавшая с 36-разрядными числами. В персональных ЭВМ операции с вещественными числами были реализованы только в 1980 году в микросхеме-сопроцессоре Intel 8087. На современных ЭВМ чаще всего используются следующие форматы вещественных чисел: короткие (длиной 4 байта), длинные (8 байт), расширенные (10 байт) вещественные числа. В языке Free Pascal им соответствуют стандартные вещественные типы `Single`, `Double` и `Extended`. Заметим, что для первых ЭВМ вещественные числа имели разное внутреннее представление (и длину). В то же время на момент *массового* выпуска ЭВМ новых поколений, для работы с вещественными числами, уже существовал международный стандарт (ANSI/IEEE 754-1985), на внутреннее представление этих чисел и операции над ними (Standard for Binary Floating-Point Arithmetic). Почти все современные машины придерживаются именно этого стандарта.

Операции над вещественными числами в старых процессорах Intel выполнялись только на восьми специальных 80-битных регистрах, которые образуют специфическую структуру данных – кольцевой стек, регистры в этом стеке обозначаются в Ассемблере как `st(0)-st(7)`.

Физически эти регистры обозначаются `R0-R7`,[2] а `st(0)` обозначает *вершину стека* на кольце из регистров `R0-R7`. Команд, непосредственно работающих с регистрами `R0-R7`, в языке машине нет.[3] Вообще говоря, в каком-то смысле на этих регистрах можно выполнять и операции с целыми числами, так как происходит автоматическое преобразование целых чисел в вещественные при чте-

---

[1] Информация (лат. informatio – разъяснение, осведомление, изложение) относится к основополагающим сущностям нашего мира, таким, как, например, пространство, время и материя. Фактически их невозможно определить через другие, более простые сущности, потому что этих более простых сущностей нет.

[2] В 64-битных процессорах уже 16 таких регистров.

[3] К младшим 64 битам этих регистров (там располагается *мантисса* числа) можно обратиться, используя имена векторных регистров `MM0-MM7`.

нии на эти регистры из папяти, и обратное преобразование при записи с этих регистров в память. Мы не будем этим пользоваться, немного о такой работе будет сказано в концевой сноске к этой главе.

Современные модели нашего компьютера могут также работать с *векторами* вещественных чисел форматов `Single` и `Double` (как и с *векторами* целых чисел длиной от 1 до 8 байт) на специальных *векторных* регистрах `XMM`, `YMM` и `ZMM`. Об этом будет рассказано в 17 Главе этой книги.

Стандарт ANSI/IEEE 754-1985 на представление вещественных чисел разработан под руководством американской ассоциации Института инженеров по электротехнике и электронике IEEE (Institute of Electrical and Electronics Engineers). Любопытно, что этот стандарт в основном разработал один человек, канадский профессор математики Уильям Каган [Вильям Кэхэн] (William Morton Kahan), лауреат премии Тьюринга, который работал в Калифорнийском университете в Беркли. Отметим, что к этому времени в математическом сопроцессоре Intel 8087 была уже реализована похожая система работы с вещественными числами, а, начиная с процессора 80387 обеспечена полная совместимость со стандартом.

В стандарте IEEE-754 были ещё предусмотрены вещественные числа половинной точности (Half precision) длиной 16 бит, в них всего 3 значащие десятичные цифры. Большинство языков высокого уровня их не реализуют, однако современные процессоры, снабжённые набором команд AVX2, могут работать с такими числами на векторных регистрах (в основном, осуществляя преобразование между числами одинарной (`Single`) и половинной (Half precision) точности.

Отметим, что в 2008 году вышел обновлённый стандарт IEEE754-2008, в него, в частности, добавлены сверхдлинные 16-байтные вещественные числа `real16` и так называемые десятичные (decimal) числа с плавающей запятой длиной 64 и 128 бит. В 2019 году вышел ещё немного обновлённый стандарт IEEE754-2019. Любопытно, что в этих новых стандартах есть и вещественные числа с десятичным показателем степени (так называемый формат Radix).

- **Целые числа**

Целые числа могут занимать в памяти 8 бит (короткое целое, байт), 16 бит (длинное целое, слово), 32 бита (сверхдлинное целое, двойное слово) и 64 бита (расширенное целое, четверное слово). В новых моделях процессоров можно также работать с целыми числами длиной 128 бит (восьмерное слово). Не следует путать термин «слово» в архитектуре Intel с «машинным словом» в машине фон Неймана, там это содержимое *одной* ячейки памяти. Кроме того, как уже упоминалось на векторных регистрах `YMM` и `ZMM` можно работать и с *векторами* целых чисел.

Как видим, в этой архитектуре есть многообразие форматов целых чисел, что, как уже говорилось, позволяет описывать данные более компактно.

- **Символьные данные**

В качестве представления символов используются короткие целые числа, которые трактуются как неотрицательные (беззнаковые) числа, задающие номер символа в некотором алфавите. Кроме того, в настоящее время существуют алфавиты, содержащие большое количество символов, для их представления, естественно, используется бóльшее и часто переменное количество байт. Например, алфавит версии Unicode 12.0 насчитывает около 137.000 символов (максимально можно закодировать 1.112.064 символов, так что место ещё есть 🙂).

Заметим, что как таковой символьный тип данных (в смысле языка Паскаль) в языке машины и Ассемблере отсутствует. И пусть Вас не будет вводить в заблуждение запись `'A'` в языке Ассемблера, который Вы вскоре станете изучать, эта запись обозначает не константу символьного типа данных, а является *целочисленной* константой и эквивалентна выражению `ord('A')` языка Паскаль.

- **Массивы (строки)**

Допускаются только одномерные массивы, которые могут состоять из коротких, длинных или сверхдлинных целых чисел, а для 64-битных моделей и из чисел длиной 64 бит. Массив коротких целых чисел может рассматриваться программистом как *символьная строка*, отсюда и второе название этой структуры данных. В машинном языке присутствуют только команды для обработки *элементов* таких массивов, но, обработав текущий элемент массива, эта команда сама настраивается на обработку следующего или предыдущего элемента. Если такую команду поставить в цикл, то получается удобное средство для работы с такими массивами. Этот тип данных будут изучаться нами в разделе 8.1.

- **Логические (битовые) вектора**

В языке машины представлены команды для обработки логических векторов длиной 8, 16 и 32 бита, а в 64-битных моделях и 64 бит. Элементы таких векторов трактуются как логические переменные. Эти команды будут изучаться нами в 9 главе.

- **Двоично-десятичные целые числа**

Этот формат данных является устаревшим, в 64-битных машинах он уже отсутствует. Это целые числа в двоично-десятичной записи, имеющие размер до 16 байт. Для *неупакованных* двоично-десятичных чисел в каждом байте хранится одна десятичная цифра, а для *упакованных* – две цифры, по одной в каждом полубайте (nibble). В 32-битном режиме этот формат используются достаточно редко, в основном, когда надо обрабатывать большие целые числа (длиной до 31 десятичной цифры). Отметим, что эти числа (правда, длиной только до 18 десятичных цифр) могли обрабатываться и на 80-битных вещественных регистрах. Этот формат данных в этой книге рассматриваться не будет (как уже говорилось, в 64-битном режиме этого формате нет).

- **Данные на векторных регистрах**

Для работы с целыми и вещественными числами предназначены и *векторные* 256-разрядные регистры `YMM0-YMM15`, причём учтите, что младшие 128-разрядные части этих регистров имеют имена `XMM0-XMM15` и могут использоваться самостоятельно. Каждый такой регистр `YMM` может хранить *вектор* из 8-ми 32-разрядных вещественных чисел типа `Single` или 4-х 64-разрядных чисел типа `Double`. Кроме того, на каждом `YMM` регистре можно хранить и вектора целых чисел (4 числа типа `int64/qword`, восемь чисел типа `Longint/Longword`, 16 чисел типа `smallint/word` или 32 числа типа `Shortint/byte`). Операции (сложение, вычитание, умножения и т.д.) выполняются сразу над всеми элементами вектора параллельно. Кроме того, в новых процессорах фирмы Intel появились векторные 512-разрядные регистры `ZMM0-ZMM31`, причём, естественно, «старые» регистры `YMM0-YMM15` являются младшими частями регистров `ZMM0-ZMM15`. К сожалению, в 32-битном режиме доступно только первые 8 из этих регистров (`XMM0-XMM7`, `YMM0-YMM7`, `ZMM0-ZMM7`). Скоро, вероятно, появятся и регистры длиной в несколько килобит 🙂.

Вещественные числа, обрабатываемые на векторных регистрах, немного «неполноценные» и менее точные по сравнению со стандартными вещественными числами, отрабатываемыми на регистрах `st(0)-st(7)` (см. следующий раздел 5.4).

5.4. Вещественные числа

Всё, что познаётся, имеет число, ибо невозможно ни понять ничего, ни познать без него.
Пифагор Самосский, VI век до н.э.

В качестве примера рассмотрим представление короткого вещественного числа (Single Precision) в стандарте ANSI/IEEE 754-1985. Такое число имеет длину 32 бита и содержит три поля:

±	E	M
1 бит	8 бит	23 бита

Первое поле из одного бита определяет знак числа, знак «плюс» кодируется нулём, «минус» – единицей. Остальные биты, отведённые под хранение вещественного числа, разбиваются на два поля: *машинный (смещённый) порядок* `E` (Biased Exponent) и *мантиссу* `M` (Fraction). Мантисса задаёт двоичное число, значение которого по модулю считается меньше единицы, другими словами, это число можно записать как $(0.M)_2$. И теперь каждое представимое в этом формате вещественное число `A` (кроме вещественного нуля `0.0`) может быть представлено в виде произведения $A=\pm 1.M*2^{E-127}$. Таким образом, машинный порядок `E` является уменьшенным на 127 двоичным порядком числа (поэтому машинный порядок и называется ещё *смещённым* порядком). Такое представление вещественного числа называется *нормализованным*: его первый сомножитель удовлетворяет неравенству:

$$1.0 \le 1.M < 2.0$$

Нормализация необходима для однозначного представления ненулевого вещественного числа в виде двух сомножителей. Нулевое же число представляется по особому, нулями во всех позициях, за исключением, быть может, первой позиции знака числа. Сам процессор при вычислении всегда полу-

чает `+0.0`, при этом, к счастью, числа `-0.0` и `+0.0` при сравнении процессором считаются *равными*.[1]

Мантисса вместе с единичным битом перед точкой имеет в английском языке специальное название significant (значащая часть числа). По-видимому, впервые такой формат вещественного числа с неявно заданной (опущенной) первой единицей (implied bit) в целой части описал немецкий инженер-конструктор ЭВМ К. Цузе в своём алгоритмическом языке Планкалкэль (Plankalkül – Исчисление планов). В стандарте ANSI/IEEE единичный бит перед точкой хранится в явном виде (в левом бите мантиссы) только в так называемом расширенном (extended) представлении вещественного числа длиной 80 бит.

В качестве примера переведём десятичное число `-13.25` во внутреннее машинное представление. Для этого сначала переведём его в двоичную систему счисления:

$13.25_{10} = -1101.01_2$ Затем нормализуем это двоичное число:

$1101.01_2 = -1.10101_2 * 2^3$

Следовательно, мантисса нашего числа будет иметь вид $10101000000000000000000_2$, и осталось вычислить машинный порядок `E: 3 = E-127`; $E = 130 = 128+2 = 10000010_2$. Теперь, учитывая знак, получаем вид внутреннего машинного представления числа -13.25_{10} (запишем его в виде двоичного и, как это часто делается для компактной записи, шестнадцатеричного значения):

$1100\ 0001\ 0101\ 0100\ 0000\ 0000\ 0000\ 0000_2 = C1540000_{16}$

Шестнадцатеричные числа в Ассемблере принято записывать с буквой `h` на конце, при этом, если такое число начинается с букв `A-F`, то впереди записывается незначащий ноль, чтобы отличить запись такого числа от *имени*:

$C1500000_{16} = 0C1500000h$

Таков формат короткого вещественного числа. Исходя из этого формата, машинный порядок `E` изменяется от 0 до 255, однако, как будет показано далее, значения машинного порядка 0 и 255 зарезервированы для специальных целей, поэтому представимый диапазон порядков коротких вещественных чисел равен $2^{-126}..2^{127} \approx 10^{-38}..10^{38}$.[2]

Как и для целых чисел, машинное представление которых будет рассмотрено чуть позже, число представимых вещественных чисел *конечно*. Действительно, легко понять, что для рассмотренного выше формата таких чисел не больше, чем 2^{32}, а на самом деле, как станет вскоре ясно, даже несколько меньше. Следует также заметить, что, в отличие от целых чисел, в представлении вещественных чисел используется *симметричная* числовая ось, то есть для любого представимого положительного числа найдётся соответствующее ему отрицательное число (и наоборот). Таким образом, в отличие от целых чисел, у каждого вещественного числа есть абсолютная велчина.

Из-за конечной длины представления вещественных чисел действия с ними чаще всего выдают приближённый результат. Одним из следствий приближенного характера вычислений с вещественными числами является нарушение ассоциативного и дистрибутивного законов арифметики. Другими словами, в общем случае $(a+b)+c \neq a+(b+c)$, $(a*b)*c \neq a*(b*c)$ и $(a+b)*c \neq a*c+b*c$. Например, $(1+2^{30})-2^{30}=0.0$, но $1+(2^{30}-2^{30})=1.0$.

Чтобы показать, насколько привычная для нас арифметика отличается от арифметики машинной (дискретной), рассмотрим решение простейшего уравнения `X+A=A`. Естественно, что в обычной математика у такого уравнения для любого значения `A` существует только один корень `X=0`, однако при решении этой задачи на компьютере можно получить и не нулевые корни такого уравнения! И не следует думать, что такие «неправильные» корни будут какими-нибудь очень маленькими числами. Например, для $A=10^{19}$ это будет корень `X=0.21`, для $A=10^{21}$ – корень `X=17.0`, а для $A=10^{24}$ – уже корень `X=12000.0`.

---

[1] Сама фирма Intel рекомендует использовать `+0.0` для «настоящих» нулей, а `-0.0` для нулей, которые в процессе вычислений округлились к нулю.

[2] Отметим, что для 80-битных чисел фомата `extended` это уже диапазон $10^{-4932}..10^{4932}$.

Как уже упоминалось выше, некоторые комбинации нулей и единиц в памяти, отведённой под хранение вещественного числа, используются для служебных целей. В частности, значение машинного порядка `E=255` при мантиссе `M ≠ 0` обозначает специальное значение «*не число*» (`NaN` – Not a Number). При попытке производить арифметические операции над такими «числами» в арифметико-логическом устройстве может возникать аварийная ситуация. Например, значение «не число» может быть присвоено программистом вещественной переменной после её порождения, если эта переменная не имеет «настоящего» начального значения (как говорят, *не инициализирована*). Такой приём позволяет избежать тяжёлых семантических ошибок, возникающих при работе с неинициализированными переменными, которые при порождении могут иметь случайные значения.

В зависимости от первого (слева) бита мантиссы различают два вида таких «не чисел»: тихое (quiet) `QNaN` и громкое (signaling) `SNaN`, аварийная ситуация (исключение) возникает только при использовании в АЛУ громкого `NaN`. Это исключение программист может заблокировать, тогда процессор «по тихому» преобразует `SNaN` в `QNaN`. Сам процессор вырабатывает только `QNaN`, `SNaN` присваивается переменной самим программистом (обычно при инициализации таких переменных) для возбуждения исключения, если такая переменная не получит «нормального» значения до её использования.

Отметим ещё одну специальную комбинацию нулей и единиц в представлении вещественных чисел. Машинный порядок `E=255` при мантиссе `M = 0` задаёт, в зависимости от знака числа, специальные значения `±∞`. Эти значения выдаются в качестве результата арифметических операций с вещественными числами, если этот результат такой большой по абсолютной величине, что не представим среди множества машинных вещественных чисел.

В ранних процессорах фирмы Intel программист, установив специальный бит `IC` в управляющем регистре арифметического сопроцессора `CWR` (Control Word Register, см. далее), мог выбрать между так называемой *проективной* арифметикой, когда `-∞=+∞` (т.е. числовая ось замкнута в кольцо) и *аффинной* арифметикой, когда `-∞<+∞`. В современных процессорах по стандарту IEEE 754-1985 осталась только аффинная арифметика.

Итак, вот эти особые вещественные числа (для чисел `NaN` приведены по одному из множества таких чисел):

```
+∞   7FF00000h        -∞   FFF00000h
+0.0 00000000h        -0.0 80000000h
QNan 7FF40000h        SNan 7FF80000h
```

Ниже показана числовая ось 32-битных вещественных чисел одинарной точности (этот красивый рисунок взят с сайта `http://www.softelectro.ru/ieee754.html`):

Отметим, что операции над вещественными числами, дающими результат `±∞`, очень похожи на операции с целыми числами, выполняющимися в так называемом режиме с насыщением (with satura-

tion). Эти операции выполняются только на специальных векторных регистрах (см. Главу 17). При выходе значения такой операции за верхнюю или нижнюю допустимые границы своего типа, в качестве результата берётся соответственно эта верхняя или нижняя граница. При этом, естественно, есть, скажем, две команды сложения с насыщением (одна для знаковых и одна для беззнаковых чисел). Например, пусть для `X db 250` выполняется *беззнаковое* сложение с насыщением `X:=X+10`, тогда получится ответ `X=255`. Никакие флаги при этом не устанавливаются. Такая «хитрая» целочисленная арифметика широко применяются при обработке мультимедийных данных (изображения и звука). Действительно, например, при сложении и вычитании «двух звуков» или «двух яркостей» их значения не могут выйти за некоторый максимальный и минимальный пределы. Аналогично для вещественных чисел при выходе результатов операций за допустимый диапазон получаются значения `±∞`

Процессор достаточно «разумно» (по крайней мере, с точки зрения математика) производит арифметические операции над такими «числами». Например, пусть A любое представимое вещественное число, не равное нулю, тогда

$\pm A/0 = \pm\infty;\ \pm A/\pm\infty = \pm 0;\ A*\pm\infty = \pm\infty;\ \infty+\infty = \infty;\ (-1.0)*(-\infty) = +\infty;$
$0*(\pm\infty) = -\infty+\infty = \pm\infty/\pm\infty = \pm 0/\pm 0 = \mathrm{sqrt}(-1.0) = \mathrm{NaN};$
$A \pm \mathrm{NaN} = \mathrm{NaN} \pm \mathrm{NaN} = \mathrm{NaN} \pm \infty = \pm 0.0^{\pm 0.0} = \mathrm{NaN}$ и т.д.;
$1.0^{\mathrm{NaN}} = \mathrm{NaN}^{0.0} = 1.0$ ⚠ (хотя таких *машинных* операций и нет)

Во многих языках существуют стандартные или библиотечные функции вычисления минимума и максимума. Когда хотя бы один из аргументов у этих функций `NaN`, они ведут себя по разному. Например, в большинстве языков функции `min` и `max` возвращают в качестве ответа второй аргумент: `min(1.0,NaN)=NaN`, но `min(NaN,1.0)=1.0`. Это легко понять, если вспомнить правило выработки минимального значения:

```
min(x,y): if x<y then min:=x else min:=y
```

Действительно, `1.0<NaN=false` и `NaN<1.0=false`.

В других языках функции `min` и `max` при таком вызове с параметром `NaN` возбуждают исключение, например, для языка Free Pascal это исключение `217` (Unhandled exception occurred. Произошло неизвестное исключение). При этом остальные широко известные функции (`sin`, `cos` и т.д.) ведут себя «хорошо», например, `sin(NaN)=NaN`..

Интересно, что теперь нельзя обнулить переменную, просто вычтя её саму из себя, т.е. `X:=X-X`, т.к. `-∞-(+∞)=NaN≠0.0`.и `NaN-NaN=NaN≠0.0` Необходимо также учитывать, что существуют два нуля ±0, поэтому

$+\infty/(+0)+\infty = +\infty;\ +\infty/(-0)+\infty = \mathrm{NaN}$

Для любознательных читателей заметим, что существует и так называемое *не традиционное* построение математического анализа. В таком анализе, как и в нашей ЭВМ, бесконечно большие величины ±∞ (а также бесконечно малые величины `±ε`) определяются не в виде *пределов*, как в привычном нам математическом анализе, а существуют в виде «настоящих» вещественных чисел. Такие числа называются *гипервещественными*. Вообще говоря, существуют несколько нетрадиционных анализов. С изложением одного из них можно, например, ознакомиться по книгам [11,12]. Любопытно, что это верно и для других наук, например, существуют три построения квантовой механики: матричная, волновая и статистическая.

Теперь надо разобраться, что происходит, если после выполнения операции над вещественными числами получился такой результат, который, хоть и не равен нулю, но не может быть представлен в виде *нормализованного* числа. Другими словами, для случая рассмотренного нами формата представления вещественных чисел длиной в четыре байта, этот результат по модулю меньше $1.0*2^{-126}$, т.е. должен иметь нулевой машинный порядок. В этом случае процессор, следуя стандарту ANSI/IEEE 754-1985, пытается представить этот результат уже как *денормализованное* (ненормализованное) число (denormalized number), т.е. в виде $0.M*2^{-127}$ (здесь целая часть равна не единице, а нулю). И только в том случае, если результат и для такого представления слишком мал по модулю, выдаёт в качестве ответа “настоящие” нули `±0.0`.

Денормализованные числа имеют машинный порядок `E=0` при мантиссе `M ≠ 0`. Использование денормализованного представления позволяет расширить диапазон машинных чисел. Можно показать, что если использовать нулевой машинный порядок для представления нормализованных чисел, то самое маленькое число будет $1.0*2^{-127}$. В то же время, если зарезервировать нулевой машинный порядок для представления денормализованных чисел, то наименьшее машинное вещественное число будет равно

$$0.M_{min}*2^{-127}=0.00000000000000000000001*2^{-127}=2^{-150}$$

Такое использование денормализованных вещественных чисел позволяет более «мягко» приближаться к нулю при проведении вычислений с вещественными числами, что часто уменьшает погрешность таких вычислений. Следует, однако, отметить, что использование денормализованных чисел имеет и существенный недостаток, оно влечёт за собой уменьшение *точности* представления таких чисел, т.е. в этих числах меньшее число *значащих* цифр.

Работа с денормализованными числами на современных процессорах фирмы Intel происходит много медленнее, чем с нормализованными. Обычно такие числа обрабатываются на машинном конвейере микроподпрограммой, что занимает до 100 тактов процессора (см. разд. 14.2.1.). А вот, например, графические процессоры фирмы NVIDIA обычно работают с денормализованными числами лишь немного медленнее, чем с обычными.

Стоит также отметить, что сейчас набирает популярность новый формат представления вещественных чисел под названием posit. Этот формат должен обеспечивать бóльшую точность и более эффективную работу на аппаратном уровне, хотя пока и не реализован ни в одном компьютере.

В процессоре Intel при работе с вещественными числами на регистрах `st(0)-st(7)` наступление особых ситуаций фиксируется в специальном регистре `SWR` (Status Word Register), это аналог регистра флагов `EFLAGS` при работе с целыми числами. Режимы работы вещественных регистров задаются в управляющем 16-битном регистре `CWR` (Control Word Register). В `CWR` есть биты, разрешающие (бит=0) или блокирующие (бит=1) [1] возникновение исключений для всех специальных случаев, например (в квадратных скобках номер бита в регистре):

`IM[0]` – недействительное значением (`NaN`, `0/0` и т.д.);
`DM[1]` – денормализованный операнд (denormalised);
`ZM[2]` – деление на ноль; (divide by zero, ±∞, `NaN`);
`OM[3]` – переполнение, (overflow, ±∞);
`UM[4]` – антипереполнение (underflow, денормализованный результат);
`PM[5]` – потеря точности (precision, `1.0/3.0`);

Биты `CWR[9..8]` задают точность вычисления (число бит в мантиссе вещественных чисел):

`00` – 24 бита, тип `single`;
`01` – зарезервировано;
`10` – 53 бита, тип `double`;
`11` – 63 бита, тип `extended` (режим по умолчанию).

Биты `CWR[11..9]` задают способ округления:

`00` – к ближайшему чётному (banker's rounding – банковское округление), это режим по умолчанию;
`01` – в меньшую сторону (к −∞, floor);
`10` – в бóльшую сторону (к +∞, ceiling);
`11` – к нулю (truncate – отбрасывание).

Прочитать значение `CWR` в память можно командой **`fstcw`** `m16`, а записать в регистр из памяти командой **`fldcw`** `m16`. Отметим, что язык Free Pascal считает стандартным режим, в котором все управляющие биты установлены (равны "1"), т.е. «всё разрешено».

В регистре состояния `SWR` фиксируется наступление определённых условий (бит=1), например:

---

[1] Любопытно, что трактовка этих битов другая, чем у флага разрешения прерываний `IF` в регистре `EFLAGS` (см. главу 7), где наоборот, бит=0 запрещает, а бит=1 разрешает наступление прерывания.

IE[0] – недействительная операция;
DE[1] – денормализованный операнд (denormalised);
ZE[2] – деление на ноль; (divide by zero, ±∞, NaN);
OE[3] – переполнение, (overflow, ±∞);
UE[4] – антипереполнение (underflow, денормализованный результат);
PE[5] – потеря точности (precision, 1.0/3.0)
SF[6] – ошибка стека (исчерпание – C1=0, переполнение – C1=1);
ES[7] – была какая-нибудь ошибка;

Биты SWR[13..11] задают номер вершины стека st(0) в кольцевом стеке регистров R0-R7. Флаг C1[9] фиксирует ошибку стека (см. бит SF[6]). В этом же регистре устанавливаются флаги C0[8], C2[10] и C3[14], фиксирующие результат сравнения вещественных чисел (см. разд. 8.9). Отметим, что язык Free Pascal считает стандартным режим, в котором все управляющие биты установлены (равны единицам).

Из-за наличия таких «экзотических» вещественных величин, как NaN, ±∞ и денормализованных чисел, усложняются операции сравнения. Две вещественные величины могут быть *сравнимы* между собой (в обычном смысле), *несравнимы* (например, NaN и 1.0, NaN и +∞ или даже NaN и NaN), а также *условно сравнимы*. Например, можно условиться, что любое (положительное) денормализованное число меньше любого (положительного) нормализованного числа.

Операции сравнения вещественных чисел работают по-особому, если хотя бы один из операндов NaN, говорят, что такие операнды *не сравнимы* между собой (например, NaN ≠ NaN), за фиксацию этого события отвечает флаг IM в упомянутом выше регистре SWR состояния процессора. Как следствие, для сравнения числа X на равенство NaN надо использовать не X=NaN, а X<>X (будет **true** при X=NaN 🙄). Это *единственный* способ определить NaN с помощью машинной команды сравнения.

Свойство NaN<>NaN=**true** может вызвать недоумение (почему так сделано?) и требует обоснования. Отметим, что ещё а античности среди философов ведутся дискуссии о “вещах, которые не равны самим себе”, но там всё туманно и запутано, нам надо что-нибудь конкретное.

Заметим, что, вообще говоря, NaN есть неопределённое значение. Вспомните, как мы говорили, что при порождении (например, по **var** x,y:real;) переменные x и y получают *неопределённые* значения, однако было бы неверно утверждать, что x=y ! Далее, например, sqrt(-3)=NaN и sqrt(-5)=NaN, но нельзя утверждать, что sqrt(-3)=sqrt(-5). Отсюда понятно, почему NaN<>NaN=**true**. С другой стороны, иногда хотелось бы, чтобы SNaN<>QNaN…

Для нас, как математиков 🙂. Множество *машинных* вещественных чисел не является ни полем, ни кольцом, ни даже группой! Действительно, нет «настоящего» нуля (т.к. 0.0*NaN ≠ 0.0, +0.0*∞ ≠ −0.0*∞ и т.д.), для NaN нет обратного элемента, не верны законы ассоциативности по сложению и дистрибутивности по умножению. Выполняются только коммутативные законы по сложению и умножению. Забегая вперёд, отметим, что *целые* машинные числа образуют кольцо вычетов по модулю 2^N, где N – разрядность числа.

Отметим одно интересное свойство приведённого представления вещественных чисел. Пусть X – неотрицательное вещественное число длиной 32 бита. Обозначим через X_{int} целое двоичное число, записанное этими же 32 битами и трактуемое как число в прямом коде (что это такое изучается в следующем разделе). Тогда для любых вещественных X и Y, таких, что X<Y, будет $X_{int}<Y_{int}$. Это, например, позволяет сортировать массив неотрицательных вещественных чисел, сравнивая их как целые, что легче для компьютера. С отрицательными числами все сложнее, так как мантиссы записаны в прямом коде, а целые числа будут считаться записанные в дополнительном коде, но можно ухитриться быстро инвертировать младшие 31 бит вещественного числа. Здесь, однако, надо быть бдительным, т.к. вещественные числа могут быть и *несравнимы* между собой, а целые числа всегда сравнимы. Например, $(\pm\infty)_{int}<(NaN)_{int}$ (что неверно для вещественных чисел). Такие алгоритмы часто используют библиотечные процедуры сортировки вещественных чисел.

Отметим важный факт. С вещественными числами форматов `Single` и `Double` «напрямую» могут работать только особые *векторные* регистры нашей ЭВМ (см. главу 17). На «старых» вещественных регистрах `st(0)-st(7)` могут обрабатываться только числа формата `Extended`. Вещественные (и целые!) числа остальных форматов при загрузки на эти регистры автоматически преобразуются в формат `Extended`, а при записи с этих регистров в память производится обратное преобразование. Это позволяет все промежуточные вычисления вести с повышенной точностью (80 бит).[1] На векторных регистрах числа обрабатываются быстрее (хотя и с меньшей точностью).

Отметим, что сейчас бóльшая часть работы с вещественными числами производится на векторных регистрах, работа с ними описана в дополнительной главе 17. Кроме того, небольшой пример использования векторных регистров показан в сноске к 14 главе. В заключение рассмотрения машинного представления вещественных чисел отметим, что при изучении архитектуры ЭВМ операции над вещественными числами, в основном из-за недостатка времени, в обязательную программу нашего курса не входят (см. разд. 8.9).

5.5. Целые числа

Бог создал целые числа, всё остальное – дело рук человека.

Леопольд Кронекер

Как уже говорилось, хранимые в памяти машинные слова (наборы битов) могут трактоваться поразному. При вызове в устройство управления этот набор битов трактуется как команда, а при вызове в арифметико-логическое устройство – как число. В дополнении к этому в рассматриваемой нами архитектуре каждое хранимое целое число может уже самим программистом трактоваться как *знаковое* или *беззнаковое* (неотрицательное). По внешнему виду *невозможно* определить, какое целое число храниться в определённом месте памяти, только сам программист может знать, как он *рассматривает* это число (вспомните соответствующий принцип фон Неймана). Таким образом, определены две различные машинные *системы счисления* для представления знаковых и беззнаковых целых чисел соответственно.

Беззнаковые (unsigned) числа представляются в уже известной Вам двоичной системе счисления, такое представление называется прямым кодом (signed magnitude representation) неотрицательного числа. Например, десятичное число 13, представленное в формате одного байта, будет записано как прямой код 00001101_2.

Если *инвертировать* прямой код (т.е. заменить все "1" на "0", а все "0" на "1"), то получим так называемый обратный код (ones' complement) целого числа. Например, обратный код двоичного числа 00001101_2 равен 11110010_2.

Для представления *отрицательных* знаковых чисел используется так называемый дополнительный код (two's complement), который можно получить из обратного кода модуля исходного числа прибавлением единицы. Например, получим дополнительный код числа −13:

```
Прямой код модуля       00001101
Обратный код            11110010
                    +          1
Дополнительный код      11110011
```

Существует и другой способ получения дополнительного кода отрицательного числа `X`. Для этого необходимо записать в прямом коде значение $2^N-|X|$, где значение `N` равно максимальному числу бит в представлении целого числа (в предыдущем примере целое число имеет длину один байт и $N=8$). Таким образом, дополнительный код числа −13 можно вычислить и так:

$$2^8-13 = 256-13 = 243 = 11110011$$

Отметим очевидное свойство дополнительного кода: если сложить дополнительный код отрицательного числа с прямым кодом модуля этого числа, то получится ноль и «лишняя» единица, не по-

---

[1] Правда, можно установить режим работы, что 64-битная мантасса будет округляться так же, как в формате `Single` (24 бита) или в формате `Double` (53 бита), но обычно так не делается.

мещающаяся в отводимое число разрядов (именно поэтому этот код и называется *дополнительным*, т.е. он «дополняет» прямой код до нуля). Описанный выше алгоритм преобразования из прямого кода в дополнительный и обратно для двоичных чисел очень прост, он часть машинной команды **`neg X`**, которая выполняется как `X:=0-X` (эта команда ещё устанавливает некоторые флаги, о которых говорится далее). Возвращаясь к представлению числа −13, имеем:

Дополнительный код `-13`		`11110011`
	+	
Прямой код `abs(-13)`		`00001101`
		`100000000`

Итак, в знаковой системе счисления отрицательные числа для нашего компьютера представляются в дополнительном коде, а неотрицательные – в прямом коде. Заметим, что при знаковой трактовке целых чисел крайний левый бит определяет знак числа («1» для отрицательных чисел). Этот бит так и называется *знаковым* битом целого числа. Для знаковых целых чисел числовая ось несимметрична: количество отрицательных чисел на единицу больше, чем количество положительных.

Очень важно понять, что все арифметические операции над знаковыми и беззнаковыми целыми числами производятся по абсолютно одинаковым алгоритмам, что и естественно, потому что процессор «не знает», какие это числа на самом деле.[i] [см. сноску в конце главы]

В то же время, с точки зрения программиста, результаты таких операций могут быть разными для знаковых и беззнаковых чисел. Рассмотрим примеры сложения в нашей ЭВМ двух чисел длиной в один байт. В первом столбике будет записано внутреннее двоичное представление чисел, а во втором и третьем – беззнаковое и знаковое значения этих же чисел в *десятичной* системе счисления.

- **Пример 1.**

	Б/з.	Знак.
`11111100`	`252`	`-4`
`00000101`	`5`	`5`
`100000001`	`1`	`1`

Митрофан (вычисляя, шепчет). Нуль да нуль – нуль. Один да один... (Задумался).
Денис Фонвизин. «Недоросль»

Из этого примера видно, что для знаковой трактовки чисел операция сложения выполнена правильно, а при рассмотрении чисел как беззнаковые, результат будет неправильным (1 вместо правильной суммы 257). Это произошло потому, что при сложении получается девятизначное двоичное число, «не умещающееся» в один байт, поэтому левый бит пришлось отбросить. Так как процессор «не знает», как программист будет трактовать складываемые числа, то он «на всякий случай» будет сигнализировать о том, что при сложении беззнаковых чисел произошла ошибка.

Для обозначения таких (и некоторых других) ситуаций в архитектуре компьютера введено понятие *флагов* (status flags). Каждый флаг занимает один бит в специальном *регистре флагов* с именем `FLAGS`.[1] Для рассмотренного выше примера флаг `CF` (Carry Flag) после сложения примет значение, равное единице (иногда говорят, что флаг *поднят* или *установлен*), сигнализируя программисту о том, что при беззнаковом сложении произошла ошибка. Рассматривая результат нашего примера в знаковых числах, получен *правильный* ответ, поэтому соответствующий флаг результата знакового сложения `OF` (Overflow Flag) будет равным нулю (или, как говорят, *опущен* или *сброшен*). Флаг `CF` называется флагом переноса, а `OF` – флагом переполнения.

При сложении двоичных чисел «в столбик» для каждой пары бит возможен перенос "1" в следующий разряд из предыдущего. Флаг знакового переполнения `OF` формируется процессором по следующему «хитрому» правилу: перенос в `CF` *не совпадает* с переносом в `SF` (это самый левый бит суммы). Аналогично при вычитании может производиться заём "1" из старшего разряда, и тогда `OF:=1`, если заём из `CF` *не совпадает* с заёмом из `SF`.

- **Пример 2.**

	Б/з.	Знак.
`01111001`	`121`	`121`
`00001011`	`11`	`11`
`10000100`	`132`	`-124`

Разве есть в жизни что-либо более прекрасное, чем поиск ответов на вопросы?
Айзек Азимов.
«Прелюдия к Академии»

[1] В 32-битном режиме этот регистр имеет длину 32 бита и имя `EFLAGS`, а в 64-битном длину 64 бита и имся `RFLAGS`.

В данном примере ошибка будет, наоборот, в случае со знаковой трактовкой складываемых чисел, поэтому флаги принимают после сложения соответственно значения `CF=0` (флаг опущен, ошибки нет) и `OF=1` (флаг поднят, была ошибка). Заметьте, что изменить значение `CF` можно и напрямую с помощью машинных команд: **`stc`** (SeT Carry, `CF:=1`), **`clc`** (CLear Carry, `CF:=0`) и **`cmc`** (CoMplement Carry, `CF:=not CF`). Эти команды не имеют явных операндов и остальные флаги не меняют.

- **Пример 3.**

	Б/з.	Знак.
11110110	246	−10
10001001	137	−119
101111111	127	+127

Признак просвещенного ума – способность обдумывать мысль, не соглашаясь с ней.
Аристотель, IV век до н.э.
«Никомахова этика»

В данном случае результат будет ошибочен как при беззнаковой, так и при знаковой трактовке складываемых чисел, поэтому формируется содержимое флагов: `CF=OF=1`. Сами придумайте пример, когда результат сложения будет правильный как для знаковых, так и для беззнаковых чисел, после такого сложения оба флага будут опущены.

Наш процессор для знаковых и беззнаковых целых чисел, кроме обычных операций по модулю (циклических), умеет выполнять и операции с так называемым *насыщением* (with saturation, saturated arithmetic). Эти операции выполняются только на специальных векторных регистрах (см. Главу 17). При выходе значения такой операции за верхнюю или нижнюю допустимые границы своего типа, в качестве результата берётся соответственно эта верхняя или нижняя граница. При этом, естественно, есть, скажем, две команды сложения с насыщением (одна для знаковых и одна для беззнаковых чисел). Например, пусть для `X db 250` выполняется беззнаковое сложение с насыщением `X:=X+10`, тогда получится ответ `X=255`. Никакие флаги при этом не устанавливаются. Такая «хитрая» арифметика широко применяются при обработке мультимедийных данных (изображения и звука). Действительно, например, при сложении и вычитании «двух звуков» или «двух яркостей» их значения не могут выйти за некоторый максимальный и минимальный пределы.

Вместе с формированием флагов `CF` и `OF` команда сложения целых чисел меняет и значения некоторых других флагов в регистре флагов. При программировании важен флаг знака `SF` (Sign Flag), в который всегда копируется знаковый (крайний левый) бит результата, таким образом, при знаковой трактовке чисел этот флаг сигнализирует, что результат получился отрицательным. Важно отметить, что анализировать флаг знака числа `SF` имеет смысл только тогда, когда флаг переполнения `OF` опущен (нулевой), иначе это бесполезно, так как правильный результат не получен и говорить, что этот результат отрицательный, не имеет смысла. Таким образом, признаком отрицательного результата будет истинность логического выражения `(OF=0) and (SF=1)`.

Кроме того, при программировании часто представляет интерес флаг нуля `ZF` (Zero Flag), который устанавливается в 1, если результат тождественно равен нулю, в противном случае этот флаг устанавливается в 0. Заметим, что флаги в этой архитектуре выполняют ту же роль, что и регистр признака результата ω в изученной ранее учебной ЭВМ УМ-3.

Основная причина использования двух систем счисления для представления целых чисел заключается в том, что при *одновременном* использовании в программе *обеих* систем счисления диапазон представимых целых чисел увеличивается в полтора раза (обязательно поймите, почему это так!). Это было весьма существенно для первых ЭВМ с их весьма небольшим объёмом памяти. Сейчас это уже не имеет такого большого значения при программировании, однако, нельзя просто отказаться от этих двух систем счисления для представления целых чисел из-за принципа программной совместимости старших моделей семейства ЭВМ с младшими, несмотря на то, что эти младшие модели уже давно не выпускаются (см. разд. 5.1).

На первых ЭВМ существовала и «естественная» целочисленная система счисления. Как мы и привыкли в школе, первый бит обозначал *знак* числа, а в остальных битах записывался *модуль* этого числа. Например, однобайтное число `-13` записывалось как `10001101`. У такой системы счисления, однако, есть два недостатка. Во-первых появляются два целых нуля `±0` (интересный вопрос, они равны или нет?). Во-вторых, сложение и вычитание усложняются, достаточно вспомнить, что сложение с отрицательным числом в школе нас учили выполнять как вычитание! Именно поэтому

принятая у нас система счисления с дополнительным кодом так нравится инженерам-конструкторам ЭВМ.

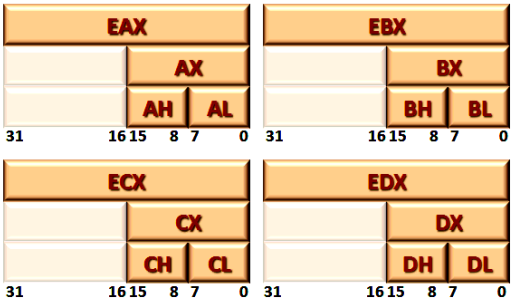
Итак, отметим важный факт. Операции с целыми числами при выходе ответа за разрядную сетку «со спокойной совестью» дают неправильный ответ и продолжают счёт программы. Отсюда важна необходимость проверки установленных флагов, если этого не делать, результат может оказаться печальным.

5.6. Мнемонические обозначения регистров

Преимущество плохой памяти состоит в том, что одними и теми же хорошими вещами можно несколько раз наслаждаться впервые.

Фридрих Ницше

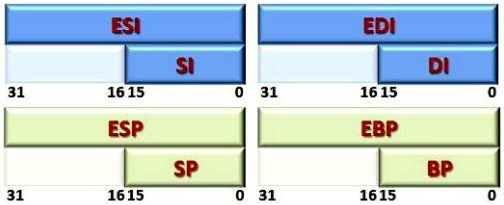
В силу того, что в ЭВМ все регистры имеют безликие двоичные (и часто весьма «хитро» закодированные) номера, программисты на Ассемблере используют мнемонические названия регистров. Восемь регистров общего назначения (GPR – Basic Program Registers) показаны ниже, каждый из них может участвовать в операциях сложения и вычитания или просто хранить данные, а некоторые – ещё использоваться в операциях умножения и деления.



В младших моделях компьютеров фирмы Intel регистры делились по функциональному назначению, буква в названии регистра обозначала: `A` – сумматор (Accumulator), `B` – базовый (Base), `I` – индексный (Index), `C` – счётчик цикла (cycle Counter), `D` – данные (Data) и т.д. В современных моделях бóльшая часть этой мнемоники не имеет смысла, например, практически любой регистр можно использовать и в качестве индексного, и в качестве базового, так что эта тема не обсуждается.

Первые четыре длинных 32-битных регистра обозначаются служебными именами: `EAX`, `EBX`, `ECX`, `EDX`. Для обеспечения возможности хранить и обрабатывать двухбайтные данные, в каждом из них выделена младшая часть длиной по 16 бит с именами `AX`, `BX`, `CX`, `DX`. Эти 16-битные регистры в свою очередь разбиты на два регистра по 8 бит с именами `AH`, `AL`, `BH`, `BL`, `CH`, `CL`, `DH`, `DL`. Биты в регистрах нумеруются справа налево, начиная с нуля, такая нумерация естественна для записи целых чисел: последняя двоичная цифра задаёт *младший* разряд этого числа (с показателем основания в *нулевой* степени).

Есть ещё четыре 32-битных регистра общего назначения с именами `ESI`, `EDI`, `ESP` и `EBP`, также имеют младшие части с именами `SI`, `DI`, `SP` и `BP`, которые, однако, уже не делятся на половинки по 8 бит.



Каждый из этих восьми регистров может быть использован в машинных командах как самостоятельный регистр. В основном эти регистры используются как *индексные*, т.е. на них обычно храниться положение конкретного элемента в некотором массиве, или как *базовые* в режиме относительной адресации. Заметьте, что у нас 32-битных, 16-битных и 8-битных регистров по 8 штук, так как под номер каждого из них в команде отводится по три бита. В дальнейшем условное обозначение

`r8` будет использоваться для обозначения любого короткого (8-разрядного) адресуемого регистра, `r16` для любого 16-разрядного и `r32` для любого 32-разрядного из этих регистров.

Имена регистров являются в Ассемблере служебными словами и не могут использоваться ни в каком другом смысле. Большие и малые буквы при этом не различаются, так что имена `RAX`, `rax`, `Rax` и т.д. задают один и тот же регистр. [ii] [см. сноску в конце главы]

Кроме перечисленных выше регистров программист имеет дело с регистром `EIP` (Extended Instruction Pointer), длиной 32 бита, который в русскоязычной литературе называется счётчиком адреса (в учебной машине он обозначался как регистр счётчик адреса `RA`). Этот регистр при выполнении текущей команды содержит адрес *следующей* исполняемой команды, здесь надо, однако, учесть, что при выполнении команд переходов в этот регистр может быть записан новый адрес, который чаще всего не совпадает с адресом следующей по порядку команды. Команды перехода будут описаны в следующей главе.

И, наконец, как уже упоминалось, архитектурой изучаемой ЭВМ предусмотрен регистр флагов с именем `EFLAGS`, он содержит 32 одноразрядных флага. Все часто используемые флаги (`CF`, `OF`, `ZF`, `SF` и некоторые другие, которые мы будем изучать позже, находятся в 16 младших битах этого регистра. Например, `CF` хранится в бите 0, `OF` в бите 11, `ZF` в бите 6 и `SF` в бите 7. Конкретные номера битов, содержащих тот или иной флаг, для понимания архитектуры несущественны, эти номера не надо будет знать и при программировании задач на языке Ассемблера.

Конечно, в процессоре есть и служебные регистры: *управляющие* 32-битные регистры `CR0-CR4` (Control Registers, многие из них пока не используются), *системные* регистры `TR`, `GDTR`, `LDTR`, `IDTR`, *отладочные* регистры `DR0-DR7` и другие, для работы с ними предусмотрены специальные команды.

Рассмотрим теперь особенности хранения чисел в регистровой и основной памяти ЭВМ. Поместим, например, шестнадцатеричное число `1234h` в 16-разрядный регистр `AX` (каждая шестнадцатеричная цифра занимает по 4 бита):

AH	AL	⇒	Ячейка 100	Ячейка 101
12h	34h		34h	12h

Теперь запишем содержимое этого регистра в память в ячейки с адресами, например, 100 и 101. Так вот: в ячейку с адресом 100 при такой пересылке запишется число из младшего байта регистра `34h`, а в ячейку со вторым адресом 101 запишется число из первого (старшего) байта регистра `12h`. Говорят, что целое число представлено в основной памяти (в отличие от регистров) в *перевёрнутом* виде. Это связано с тем, что в самых младших процессорах фирмы Intel при каждом обращении к памяти в процессор читался всего один байт. Таким образом, для того, чтобы считать двухбайтное целое число, было необходимо дважды обратиться к памяти, поэтому было удобно (например, для проведения операции сложения «в столбик») получать из памяти сначала младшие цифры числа, а затем – старшие.

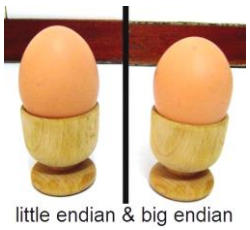
Если подумать, то записывать и десятичные числа «задом наперёд» для выполнения арифметических операций удобно не только для компьютера, но и для человека. Действительно, в большинстве стран люди пишут текст слева направо, поэтому и операции над числами удобнее производить, когда первой записывается *младшая* десятичная цифра. Теперь должно быть понятным, почему наши десятичные цифры называются *арабскими*: в арабских странах текст пишется и читается не слева направо, а справа налево, таким образом, там и числа записываются от младших разрядов к старшим!

Сам Алан Тьюринг на своих лекциях повергал присутствующих в шок, складывая и умножая на доске десятичные числа, записанные «задом наперёд», потому, что так ему было удобнее писать. А на претензии людей, что из-за этого они плохо понимают лекцию, Тьюринг искренне удивлялся, как такая мелочь может помешать усвоению материала 🙂.

Сейчас за одно обращение из памяти процессор получает сразу 8, 16 и бóльшее число байт, но из-за совместимости моделей семейства пришлось оставить *перевёрнутое*, от младших разрядов к старшим (little endian), представление чисел, что, конечно неудобно для людей, поэтому компилятор Ассемблера в листинге программы сам переворачивает эти числа, чтобы не «пугать» программиста.

Вероятно, это так «достало» программистов, что, начиная с процессора Intel 486, в языке машины появилась новая команда **`bswap`** `r32` (Byte Swap), которая переставляет все 4 байта в своём опе-

ранде (32-битном регистре) в обратном порядке. Заметим, что в отличие от чисел, в *командах* в перевёрнутом виде хранятся только *операнды* команды (адреса и непосредственные значения), а сама команда начинается с кода операции и в этом смысле хранится в не перевернутом виде (big endian).


little endian & big endian

Термины little endian (острый конец куриного яйца) и big endian (тупой конец) сначала появились в компьютерной литературе у автора первых сетевых протоколов Дэнни Коэна (Danny Cohen). Он взял их из романа Джонатана Свифта «Путешествие Гулливера», изданного в 1726 году. В этом романе между жителями двух соседних государств (Лиллипутией и Блефуску) шла яростная дискуссия, с какого конца (острого или тупого) нужно разбивать варёное яйцо за завтраком. Сам Свифт таким образом высмеивал религиозные споры между католиками и протестантами. В программировании эти термины являются намёком на дискуссию, следует ли представлять числа в памяти ЭВМ в перевёрнутом виде, или нет. Следует сказать, что у самих лилипутов дело дошло до ожесточённой трёхлетней войны, и «остроконечники» (с помощью Гулливера) победили 🙂.

Заметим, что некоторые процессоры имели в управляющем регистре специальный бит, который определяет, в каком виде (прямом или перевёрнутом) числа хранятся в памяти. Вообще говоря, при работе с длинными *вещественными* числами иногда (например, в процессорах ARM, часто используемых в планшетах и смартфонах) реализовано и смешанное (middle endian) представление числа длиной 4 (`Single`) и 8 (`Double`) байт (2 или 4 слова). В этом представлении двухбайтные *слова* в числе идут в обратном порядке, а *байты* в каждом слове – в прямом 🙁. Представление целых чисел в формате middle endian было и в древней ЭВМ PDP-11. При порядке little endian процессору удобно работать с целыми числами, а при big endian – со строками символов, поэтому big endian является стандартным для многих сетевых протоколов, например TCP/IP. В нашей ЭВМ данные в порты ввода/вывода (см. разд. 14.3.1) тоже направляются в прямом порядке, например, команды

```
mov ax,1234h
out 100,ax
```

пошлют байт `12h` в порт с адресом `100`, а байт `34h` в порт с адресом `101`.

5.7. Сегментация памяти

Знание некоторых принципов легко возмещает незнание некоторых фактов.
Клод Адриан Гельвеций

Круглое невежество – не самое большое зло: накопление плохо усвоенных знаний ещё хуже.
Платон, V век до н.э.

Материал этого раздела трудный, при первом чтении выделенный синим текст можно пропустить.

Древние 16-битные компьютеры нашего семейства имели так называемую сегментную организацию оперативной памяти.[1] Мы уже знакомы с понятием сегмента (см. разд. 4.7) – сплошного участка памяти, на начало которого установлен специальный сегментный регистр, а вся адресация производится относительно начала этого сегмента. Такая организация памяти была тогда ма, так как максимальная длина сегмента была много меньше общего объёма оперативной памяти. Вот и приходилось «перетаскивать» сегменты по памяти во время работы программы.

При переходе к 32-битным ЭВМ острая необходимость в активном использовании сегментов отпала, так как теперь максимальная длина сегмента (2^{32} байт) стала равна всей адресуемой памяти. В программировании возобладала так называемая «плоская» модель памяти, при этом все сегменты устанавливались на самое начало оперативной памяти (на ноль) и имели максимальную длину на всю адресуемую память, т.е. все сегменты полностью накладывались друг на друга. Таким образом, программисты могли вообще не использовать сегментные регистры при написании программ. Кроме то-

[1] Это такая «фигура речи», на самом деле память всех наших ЭВМ обычная линейная и ни о какой сегментной организации они даже не подозревают, она читает и пишет данные только по «настоящим», физическим адресам. Сегментный доступ в память производила аппаратура центрального процессора.

го, 16-битные сегментные регистры (они назывались CS, DS, ES, SS, FS и GS) не могли хранить 32-битный адрес начала сегменты, и в них содержался только *индекс* (порядковый номер) 64-битного *дескритора* (описателя) сегмента в специальной таблице дескрипторов.

Мнемонические обозначения сегментных регистров имеют следующий смысл: кодовый сегментный регистр (CS), сегментный регистр данных (DS), сегментный регистр стека (SS) и дополнительный сегментный регистр (ES). Остальные два сегментных регистра используются для служебных целей. Например, регистр FS содержит селектор сегмента, содержащего список информационных блоков текущего выполняемого потока команд `TIB` (Thread Information Block). Такой блок, в частности, используется для обработки так называемых структурных исключений SEH (Structured Exception Handling) в выполняемом программном потоке (немного об этом см. разд 6.13). Эта тема относится к курсу по операционным системам.

На сегменты в плоской модели по-прежнему возлагалась задача хранения привилегий и прав доступа к памяти сегментов. Так, кодовый сегмент CS обычно имел в своём дескрипторе признак, что он закрыт на запись, но открыт на выполнение команд, сегмент данных DS открыт на чтение и запись, но закрыт на выполнение из него команд и т.д. Необходимо понять, однако, что эта защита была чисто фиктивной. Действительно, какой смысл закрывать сегмент команд на запись, если на него полностью «наложен» сегмент данных, используя который, можно спокойно писать по этим же самым адресам.

«Настоящая» защита и задание прав доступа в 32-битной архитектуре возлагаются на совершенно другой механизм, на так называемую страничную организация памяти. Вся память при этом разбивается на страницы (обыно длиной по 4096 байт) и уже с каждой страницей связываются механизмы защиты. Полностью механизм страничной организации оперативной памяти Вы будете изучать в курсе по операционным системам.

Итак, в 32-битном режиме основным является так называемая плоская модель памяти [iii] [см. сноску в конце главы]. При этом в каждый момент времени определены шесть сегментов. Это означает, что есть шесть *сегментных* регистров с именами `CS`, `DS`, `ES`, `SS`, `FS` и `GS`. В каждом таком регистре хранится так называемый *селектор*, чаще всего это селектор *дескриптора* (описателя) сегмента (Segment Descriptor). Селектором называется 16-битовое значение, первые 13 бит из них являются *индексом* дескриптора в одной из двух таблиц дескрипторов (таблица – это *массив* дескрипторов). Ещё один бит-индикатор указывает, в какой таблице дескрипторов (глобальной или локальной) находится дескриптор. Последние два бита задают уровень привилегий данного селектора (об этом будет рассказано далее). Дескриптор описывает какой-либо важный объект системы (сегмент, шлюз прерывания, задачу (процесс), локальную таблицу дескрипторов и т.д.).

Дескрипторы сегментов хранятся в специальных таблицах, каждый такой дескриптор описывает конкретный сегмент и содержит:

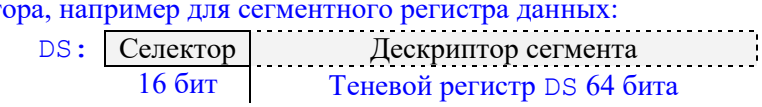
- адрес начала этого сегмента в оперативной памяти,
- максимальную длину (предел) сегмента,
- права доступа (разрешено ли чтение, запись и выполнение команд),
- уровень необходимых привилегий для работы с сегментом (об этом позже),
- и некоторые другие атрибуты.

Производить обмен с памятью можно только относительно одного из этих сегментов, т.е. каждая команда машины, обращающаяся в память, однозначно знает свой сегментный регистр (редко два регистра), к которые эта команда должна использовать по умолчанию. При необходимости, поставив впереди особую однобайтную команду-префикс, можно сменить сегмент по умолчанию, но только при обращении в память за *данными*, в то время как *команды* всегда выбираются устройством управления из сегмента, на который указывает кодовый сегментный регистр `CS`. Кодовый сегмент должен допускать выполнение команд.

Все дескрипторы «личных» сегментов, с которыми работает конкретная задача (процесс), собраны в её так называемой локальной дескрипторной таблице `LDT` (Local Descriptor Table). Эта таблица сама хранится в служебном *сегменте*, дескриптор которого, в свою очередь, хранится в глобальной дескрипторной таблице `GDT` (Global Descriptor Table). Глобальная таблица, в частности, хранит дескрипторы сегментов, «общих» для всех программ, на начало этой таблицы указывает системный регистр `GDTR`. Сам селектор `LDT` хранится в своём системном регистре `LDTR`. В каждом селекторе сегмента (в частности, в тех, которые находятся в сегментных регистрах `CS`, `DS` и т.д.) есть бит *ин-*

дикатора таблицы TI (Table Indicator). Этот бит показывает, где хранится сам дескриптор сегмента, для TI=0 дескриптор сегмента находится в GDT, иначе в LDT. (Ну, как всё запутано 🙂).

У каждого сегментного регистра есть его так называемый *теневой* (shadow) регистр длиной 8 байт, который хранит сам дескриптор этого сегмента. Это сделано для ускорения работы ЭВМ, чтобы часто не обращаться к дескрипторам сегментов, расположенным в оперативной памяти. Теневые регистры невидимы из программы, для работы с ними нет отдельных команд. Дескриптор сегмента загружается в теневой регистр автоматически при каждой записи в сегментный регистр значения селектора, например для сегментного регистра данных:



При программировании на Ассемблере в этом курсе сегментные регистры в явном виде использоваться не будут.

Выполняющейся программе предоставляется адресное пространство для размещения команд и чисел. Для каждой команды, которая производит обращение к памяти по чтению или записи, процессор вычисляет адрес её операнда (редко двух операндов). Этот адрес называется исполнительным (executable – т.е. вычисляемым) адресом $A_{исп}$, правила его вычисления зависят от формата конкретной команды и будут подробно изучены далее.

Каждый исполнительный адрес по существу является *смещением* от начала определенного сегмента. Адрес числа или команды в адресном пространстве программы называется логическим (иногда линейным) адресом (logical (flat) address), он вычисляется процессором по формуле

$$A_{лог} := (SEG + A_{исп}) \bmod 2^{32},$$

где SEG – адрес начала нужного сегмента в памяти. Адрес берётся по модулю 2^{32}, чтобы он не вышел за допустимые границы адресного пространства оперативной памяти.

Теперь необходимо сказать, что дальнейшая работа с логическим адресом определяется режимом работы процессора. В режиме *реальной* адресации логический адрес считается *физическим* адресом (physical address) оперативной памяти, именно по этому адресу производится обращение по чтению или записи.

В режиме *виртуальной* памяти логический адрес считается *виртуальным* адресом (virtual address), его преобразование в физический адрес производится по достаточно сложным правилам, эта тема изучается в курсе по операционным системам. Сейчас стоит только сказать, что преобразование виртуального адреса в физический является полностью прозрачным (невидимым) для программиста, как на языках высокого уровня, так и на Ассемблере. Таким образом, программист работает *только* с логическими адресами, так что в дальнейшем под адресом понимается именно логический адрес. При работе в 32-битном режиме логическое адресное пространство имеет 2^{32} байт, а, например, размер физической памяти для процессора Intel Core i9 ограничен длиной 2^{37} байт = 128 Гб.

Сразу надо сказать, что сама оперативная память «не знает» ни о каких сегментах, она читает и пишет только по «настоящим», физическим адресам. Логические и виртуальные адреса – это «кухня» самого процессора (так ему удобнее выполнять команды).

Каждый из сегментов может иметь длину до 2^{32} байт. Так как логический адрес в приведённой выше формуле берётся по модулю 2^{32}, то, очевидно, что память, как и в нашей учебной ЭВМ, как бы замкнута в кольцо.

Стоит отметить, что сегментные регистры являются специализированными, предназначенными только для хранения селекторов, поэтому арифметические операции (сложение, вычитание и др.) над их содержимым в языке машины не предусмотрены.

В настоящее время для программ в большинстве случаев используется так называемая *плоская* модель памяти (flat memory model). В этой модели предполагается, что все четыре сегмента CS, DS, ES и SS начинаются с нулевого адреса и имеют максимально возможную длину 2^{32} байт. Таким образом, эти сегменты в памяти полностью перекрываются (совпадают).[iv] [см. сноску в конце главы].

Такая установка сегментов задаётся служебной программой (загрузчиком) перед началом счёта и в дальнейшем самим программистом не меняется. Загрузка новых значений в сегментные регистры (CS, DS и т.д.) на языке Ассемблера в принципе возможна, однако надо чётко понимать, что при этом

происходит, иначе чаще всего программа просто аварийно завершается. В этой книге команды, *непосредственно* читающие и записывающие значения в сегментные регистры, не изучается.

5.8. Структура команд

Низкоуровневое программирование – это разговор с компьютером на естественном для него языке, радость общения с «голым» железом, высший пилотаж полёта свободной мысли и безграничное пространство для самовыражения.

Крис Касперски ака мыщъх

Сейчас мы рассмотрим структуру машинных команд 32-битного режима. Большинство команд будут изучаться на языке Ассемблера, но два самых распространённых формата команд регистр-регистр (RR) и регистр-память (RX) мы рассмотрим на внутреннем уровне (битового представления).

5.8.1. Формат регистр-регистр [1]

Недостаточно овладеть премудростью.
Нужно так же уметь пользоваться ею.

Марк Туллий Цицерон

Команды этого формата занимают в памяти 2 байта, перед которыми могут стоять один или два однобайтных команд-префиксов (о них будем говорить далее):

6 бит	1 бит	1 бит	2 бита	3 бита	3 бита
КОП	d	w	11	r1	r2

Первые 6 бит команды занимает код операции (будем обозначать его как ⊗), что позволяет задавать 64 различные операции. Далее следуют однобитные поля с именами d – так называемый бит направления и w – бит размера *операнда*. Последующие два бита для этого формата всегда равны 11, а два последних поля (по 3 бита каждое) задают номера (от 0 до 7) регистров-операндов команды (там всего 8 регистров каждого размера).

Стоит подробнее рассмотреть назначение битов d и w. Бит d задаёт *направление* выполнения операции, код которой обозначен как ⊗, а именно:

```
<r1> := <r1> ⊗ <r2>      при d = 1
<r2> := <r2> ⊗ <r1>      при d = 0.
```

Для формата регистр-регистр этот бит не имеет большого значения, так как программист всегда может поменять в команде местами регистры первого и второго операнда, однако для формата регистр-память этот бит очень важен, так как может превращать формат регистр-память (RX) в формат память-регистр (XR). Именно поэтому в форматах команд регистр-память указывается только один вид RX. [v] [см. сноску в конце главы]

Бит w задаёт размер регистров-операндов, соответствие двоичных номеров регистров и их имён можно определить по приведённой ниже таблице.

$r_{1,2}$	w=0	w=1
000	AL	AX,EAX
001	CL	CX,ECX
010	DL	DX,EDX
011	BL	BX,EBX
100	AH	SP,ESP
101	CH	BP,EBP
110	DH	SI,ESI
111	BH	DI,EDI

[1] Здесь только регистры общего назначения, команды с векторными регистрами рассмотрены в главе 17, а работа с вещественными регистрами st(0)-st(7) и xmm0-xmm7 рассматривается в разд. 8.9.

Выбор длины регистра для `w=1` зависит от режима работы процессора, который задаётся в бите `D` (Default Size) дескриптора текущего кодового сегмента (см. разд. 5.7). Различают 16 и 32-разрядные режимы работы (`D=0` и `D=1`), в каждом из них используются регистры-операнды соответствующей длины. Трудность возникает, если, в 32-разрядном режиме надо, например, выполнить команду **`mov`** `bx, ax` с 16-разрядными регистрами. В этом случае компилятор с Ассемблера автоматически ставит перед такой командой специальную однобайтную команду-префикс смены размера операнда с кодом операции `66h`. Такой префикс заставляет процессор временно (на одну следующую за ним команду) сменить свой режим работы (в нашем примере с 32-разрядного на 16-разрядный). Т.е. команда с префиксом `66h` **`mov`** `ebx,eax` будет выполняться как команда **`mov`** `bx,ax`. Понятно, что в 32-разрядном режиме надо избегать использования 16-разрядных регистров.

Забавно, но, так как префикс `66h` является хоть и вспомогательной, но всё же командой, которая к тому же не «спаривается» с командой **`mov`** на конвейере (т.е. не выполняется с ней параллельно на разных обрабатывающий устройствах конвейера), то по времени часто оказывается выгоднее вместо одной команды **`mov`** `bx,ax` использовать две команды **`mov`** `bl,al` и **`mov`** `bh,ah`. Но самая быстрая всё же команда **`mov`** `ebx,eax`.

Далее мы познакомимся и с другими префексами, например, префикс `LOCK` `F0h` задаёт блокировку шины обмена с памятью, префиксы `REPNE/REPNZ` `F2h` и `REP/REPE/REPZ` `F3h` задают циклы обработки так называемых строковых команд (см. разд. 8.1), префикс `67h` задаёт смену длины адреса, префиксы VEX, VEX2 и VEX3 (Vector EXtension, длиной 1, 2 и 3 байта) кодирует векторные команды и т.д.

Отметим, что префиксы `66h`, `67h`, REX, VEX, VEX2 и VEX3 в Ассемблере безымянные, они автоматически вставляются в программу компилятором Ассемблера, программист об этом не заботиться. Сам же программист при необходимости может вставить эти префиксы в свою программу только в виде констант.

Префиксы можно ставить и перед командами, на которые они не влияют, например, REX перед 32-битной командой, `66h` перед командой, где нет регистров, используются байтовые или 64-битные регистры, `REP` не перед строковой командой и т.д., в этом случае они не влияют на выполнение программы, но процессор всё равно выполняет их как пустые команды, т.е. тратит на это время. Когда перед командой стоят несколько одинаковых префиксов, либо префиксы идут не в том порядке (например, REX перед `66h`), то в документации Intel сказано, что действие процессора не определено, но обычно тоже «всё будет разумно» 🙂.

При необходимости задать команду в 64-битном режиме (там уже 16 регистров каждого размера), перед командой добавляется однобайтная команда-префикс под названием REX,[1] которая имеет показанную ниже структуру:

	7..4 бит	3-й бит	2-й бит	1-й бит	0-й бит
REX:	0100	W	R	X	B

Бит REX[`W`] определяет 64-битный режим работы, теперь номера регистров в командах рассматриваются как `r64`, т.е. `EAX` становится `RAX`. Бит REX[`R`] задаёт в номере регистра дополнительный старший бит, так что их становится по 16 штук каждого размера, и теперь, например, доступны регистры `R8b-R15b`, `R8d-R15d` и `R8-R15`. При необходимости задать в этом режиме работу с 16-битными регистрами `R8w-R15w`, как и раньше, перед REX-префисом `4Ch` надо поставить ещё и префикс смены длины операнда `66h` (префиксы множатся, как грибы после дождя 🙂).

Как видно из приведённой выше таблицы, архитектурой этого компьютера не предусмотрены (т.е. запрещены) команды над регистрами разной длины. Таким образом, команды типа **`add`** `al,bx` являются *неправильными*.[2] Исходя из этого, для проведения операций над числами разной длины появля-

---

[1] REX (Register EXtension) – эта команда-префикс в основном предназначена для расширения адресного пространства регистров (с 8 до 16).

[2] К немногочисленным исключениям можно отнести команды **`movsx`** и **`movzx`** для знакового и беззнакового расширения чисел, они будут рассмотрены далее при изучении команды пересылки **`mov`**.

ется необходимость *преобразования типов* из короткого целого в более длинное, и наоборот. Такое преобразование, как можно (и нужно!) понять, зависит от знаковой или беззнаковой трактовки числа.

Беззнаковое число всегда расширяется из короткого формата в более длинный приписыванием слева двоичных нулей, а для знакового числа слева приписывается (как часто говорят, *размножается*) его знаковый (крайний слева) бит. Вам необходимо понять, что для *знаковых* чисел незначащими левыми двоичными цифрами будут "0" для неотрицательных и "1" для отрицательных значений. Для преобразования *знаковых* целых чисел из короткого формата в более *длинный* в языке машины предусмотрены безадресные команды, имеющие в Ассемблере такую мнемонику:

cbw (Convert Byte to Word)	AL → AX
cwd (Convert Word to Double)	AX → <DX:AX>
cdq (Convert Double to Quad)	EAX → <EDX:EAX>
cqo (Convert Quad to Octa)	RAX → <RDX:RAX>

Эти команды производят *знаковое* расширение короткого числа в более длинное, которое чаще всего, расположено в так называемой *регистровой паре* <DX:AX>, <EDX:EAX> и <RDX:RAX> (для 64-битного режима). Регистровая пара рассматривается процессором, как содержащая одно более длинное целое число.

С регистровыми парами (кроме команд деления) работать не всегда удобно, поэтому добавлены команды:

cwde (Convert Word to Double)	AX → EAX
cdqe (Convert Double to Quad)	EAX → RAX

Все эти команды преобразования короткого знакового числа в более длинное не меняют флагов. Рассмотренные команды не всегда удобны, так как работают только с фиксированными регистрами, как уже упоминалось, есть и более удобные команды **movsx** и **movzx**, они будут рассмотрены далее при изучении команды пересылки **mov**.

Преобразование целого значения из длинного формата в более короткий (усечение) производится путём отбрасывания соответствующего числа *левых* битов целого числа. Усечённое число будет иметь то же значение, что и исходное число, если слева будут отброшены только *незначащие* биты. Для беззнаковых чисел незначащими всегда будут нулевые биты, а для знаковых – биты, совпадающие со знаковым битом *усечённого* числа. Вам необходимо уметь объяснять, почему наш процессор в принципе не может выполнять арифметические операции (сложения, вычитания и т.д.) с операндами разной длины.

Преобразование чисел из короткого формата в более длинный и наоборот часто используется в программировании. Например, пусть на языке Free Pascal описаны переменные:

```
var x: shortint; y: longint;
```

Присваивание y:=x производится с неявным преобразованием типа как y:=longint(x), забегая немного вперёд (см. разд. 5.9.1) на языке Ассемблера в первых моделях нашего семейства это можно было реализовать командами:

```
x  dw ?
y  dd ?
   mov al,x
   cbw      ; ax:=smallint(x)
   cwd      ; <dx:ax>:=longint(x)
   mov word ptr y,ax
   mov word ptr y+2,dx; y:=longint(x)
```

Сейчас это можно сделать более коротко (см. далее команды **movsx** и **movzx**).

```
   mov   ax,x
   movsx y,ax; y:=longint(x)
```

Оператор **ptr**. В этих примерах мы впервые использовали важный и часто используемый в Ассемблере оператор **ptr**, он производит «принудительное» приведение типов операндов команды (type coercion, type casting). Команда **mov** ax,**word ptr** y «приказывает» считать, что имя y из последнего примера ссылается на переменную в памяти длиной не 4 байта (**dd**), как описано, а толь-

ко 2 байта (**dw**). Аналогично команда `mov eax,dword ptr x` будет читать на регистр EAX 4 последовательных байта, начиная с начала переменной x, туда войдут 2 байта из x и первые 2 байта из y. Здесь не надо путать, оператор языка Free Pascal `x:=shortint(y)` производит преобразование типа (type conversion), он преобразует (возможно, с ошибкой) значение из 4-х байт переменной y в двухбайтное число и записывает это число в переменную x, а оператор `mov word ptr y,ax` просто записывает два байта из регистра AX (без всякого преобразования!) в два первых байта переменной y (оставляя вторые 2 байта этой переменной без изменения). Операция **ptr** является аналогом *явного* преобразования типа в языках высокого уровня, но работает принципиально по другому. Рассмотрим пример:

```
var x: byte; y longword;

   y:=x; {y:=longword x;}
{ это 1 байт x преобразуется в
 4 байта и записывается в y.
 На Ассемблере это будет  }
   movzx eax,x [1]
   mov   y,eax
```

```
.data
x db ?
y dd ?
  mov y,dword ptr x
; это y:=4 байта из памяти,
; начиная с  x, т.е. это
; это y:=x,x+1,x+2,x+3
; т.е. y:=x,y,y+1,y+2
```

Для переменных, описанных на языке Free Pascal как

```
var x: shortint; y: longint; { x db, y dd }
```

преобразование из длинного формата в короткий при присваивании `x:=y` производится с неявным преобразованием типов как `x:=shortint(y)`, при этом должен учитываться режим работы исполнителя. В режиме {$R-} без контроля выхода значения за допустимый диапазон всё просто:

```
mov eax,y; eax:=y
mov x,al;  x:=shortint(y)
```

Сложнее обстоит дело в режиме с контролем {$R+}, приходится это делать примерно так:

```
  mov   eax,y;  eax:=y
  mov   x,al;   x:=shortint(y)
  movsx eax,al; eax:=longint(al)
; if longint(shortint(y))<>y then goto Error
  cmp   eax,y
  jne   Error; Range Checking Error
```

5.8.2. Формат регистр-память (и память-регистр)

Если вы думаете, что на что-то способны, вы правы; если думаете, что у вас ничего не получится – вы тоже правы.

Генри Форд, автомобильный магнат

Рассмотрим теперь битовое представление команд регистр-память (RX) и память-регистр (XR) для 32-битного режима работы ЭВМ:

КОП	r1	A2

Первый операнд r1 задаётся номером регистра, а торой операнд A2 может в этом формате иметь один из приведённых ниже трёх видов:

1. A2 = A
2. A2 = A[B1]
3. A2 = A[B1][Scale*I2] или A2 = A[Scale*I2]

Здесь A – задаваемый в команде адрес (смещение – displacement) длиной 0, 1 или 4 байта (причём нулевое смещение не задаётся и не занимает место в команде), B1 и I2 – так называемые регистры-модификаторы, B1 называется базовым, I2 – индексным регистрами, а Scale является числовым

[1] Как уже говорилось, команды команды **movsx** и **movzx** для знакового и беззнакового расширения чисел, они будут рассмотрены далее при изучении команды пересылки **mov**.

множителем, который может принимать значения 1, 2, 4 или 8. Как сейчас будет показано, значение адреса второго операнда A2 *вычисляется* по определённым правилам, поэтому, как уже упоминалось, этот адрес часто называют исполнительным (вычислимым – executable) адресом.

Рассмотрим подробнее каждый их трёх возможных видов второго операнда. При `A2 = A` исполнительный адрес операнда вычисляется процессором совсем просто:

$A_{исп} \equiv A$

Запись `A2 = A[B1]` означает использование в команде базового *регистра-модификатора,* которым при работе в 32-битном режиме может быть любым из 8 регистров общего назначения. Исполнительный адрес операнда при это вычисляется так:

$A_{исп} := (A + B1) \bmod 2^{32}$,

где вместо B1 подставляется содержимое одного из указанных выше регистров-модификаторов. На рис. 5.1 показана схема доступа к памяти с обычным A и базированным `A[B1]` адресом.

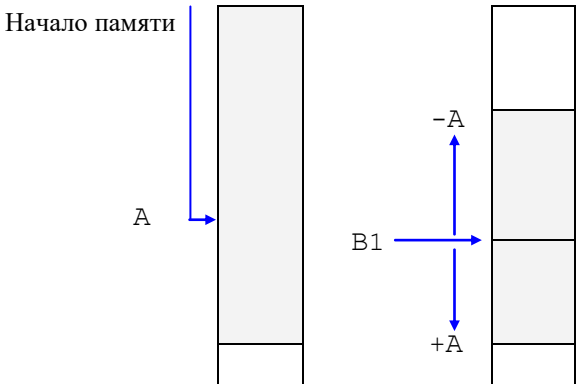


Рис. 5.1. Доступ к памяти с обычным и базированным адресом A.

Запись `A2 = A[B1][Scale*I2]` обозначает использование в команде *двух* регистров-модификаторов, как уже говорилось, один из которых называется базовым (B1), а второй – индексным (I2). В качестве индексного регистра можно задать любой 32-битный регистр общего назначения, кроме `ESP=100`. Вычисление исполнительного адреса при этом производится по формулам:

$A_{исп} := (A + B1 + Scale*I2) \bmod 2^{32}$

или

$A_{исп} := (A + Scale*I2) \bmod 2^{32}$

В нашем Ассемблере MASM допускается запись выражения `A[B1][Scale*I2]` в эквивалентном виде `±A[Scale*I2][B1]`, `±A[B1+Scale*I2]`, `±A[Scale*I2+B1]`, `[±A+B1+Scale*I2]`, `[B1+Scale*I2±A]` и даже, к сожалению, в виде `±A+[B1+Scale*I2]` или `[B1+Scale*I2]±A`. В то же время, похожая на принятую в Паскале запись `A[B1,Scale*I2]` запрещена, нак как по смыслу является обращением к элементу матрицы, вычисление адреса которого производится по другой формуле:

A[i,j]=<Начало массива A>+<длина строки>*i+<длина элемента>*j

Возвращаясь к способу вычисления исполнительного и логического адресов можно заметить, что вся оперативная память, как бы замкнута в кольцо. Другими словами, при последовательном увеличении исполнительного адреса с последнего байта памяти попадём в начало памяти (на её нулевой байт). Видно, что базовые и индексные регистры практически взаимозаменяемы, они призваны обеспечить удобный способ доступа к элементам одномерных и двумерных массивов (матриц). Заметим также, что теперь отпадает необходимость делать самомодифицирующиеся программы для обработки массивов (вспомним учебную ЭВМ УМ-3), т.к. изменяя значение регистра, можно получить доступ к нужным элементам массивов без изменения внешнего вида самой команды.

Рассмотрим теперь внутреннее машинное представление формата команды регистр-память. Длина этой команды от 2 до 7 байт:

8 бит			2 бита	3 бита	3 бита	От 0 до 5 дополнительных байт	
КОП	d	w	mod	r1	mem	SIB	❶ a8 или a32

где mod – двухбитовое поле, называемой полем модификатора (длины смещения), mem – трёхбитовое поле способа адресации памяти. Будем обозначать через a8 и a32 ❶ адреса в памяти переменных (m8 и m32) или непосредственные операнды (i8 и i32) длиной в 1 или 4 байта, заданные в дополнительных байтах команды.[1] Биты d и w уже описаны в предыдущем формате регистр-регистр и имеют тот же смысл. Все возможные комбинации значения полей mod и mem приведены в таблице 5.2.

Таблица 5.2. Значения полей mod и mem в 32-битном режиме.

mem\mod	00	01	10	11
	0 доп. байт	1 доп. байт	4 доп. байт	Это уже формат RR
000	[EAX]	[EAX]+a8	[EAX]+a32	
001	[ECX]	[ECX]+a8	[ECX]+a32	
010	[EDX]	[EDX]+a8	[EDX]+a32	
011	[EBX]	[EBX]+a8	[EBX]+a32	
100	SIB	SIB+a8	SIB+a32	
101	a32	[EBP]+a8	[EBP]+a32	
110	[ESI]	[ESI]+a8	[ESI]+a32	
111	[EDI]	[EDI]+a8	[EDI]+a32	

Как видно, в 32-битном режиме работы поле адреса в команде может иметь длину 0, 1 (a8) или 4 (a32) байта. Особое значение $mem=100_2$, соответствующее запрещённому регистру ESP, обозначено как поле SIB (Scale Index Base). Это поле показывает, что в команду непосредственно перед полем a8 или a32 добавлен *дополнительный* байт SIB, который снимает практически все ограничения на выбор двух регистров-модификаторов. Значение битов из этого байта:

2 бита	3 бита	3 бита
Scale	Индекс I2	База B1

Значения поля Scale (0..3) соответствует множителям (1, 2, 4 и 8) соответственно. С помощью байта SIB задаётся самый сложный способ вычисления вычисление исполнительного адреса второго операнда по формуле

$$A_{исп} := (A + B1 + Scale*I2) \bmod 2^{32}$$

При этом, как уже говорилось, I2 может быть любым из восьми 32-битных регистров общего назначения, кроме ESP, а B1 – вообще любым 32-битным регистром общего назначения. Заметьте, что, так как индексный регистр I2 не может быть ESP=100, то этот случай трактуется как *отсутствие* индексного регистра. Одна из «хитрых» комбинаций полей ($mod=00_2$ и mem=SIB, а в SIB B1=EBP) трактуется как *отсутствие* в команде базового регистра, и принудительно $mod:=10_2$, т.е. используется обязательное смещение a32. Таким образом операндов [EBP+Scale*I2] с нулевым смещение A не бывает, они трактуются как [Scale*I2] со смещением a32.

Примеры команд:

```
mov eax,[eax+8*edi];  смещение A=0 байт
mov eax,A[edi+4*edi]; будет 5*edi;  A длиной 1 или 4 байта
mov eax,A[8*edi];     нет базового регистра, A всегда занимает 32 бита
mov eax,[8*esi];      нет базового регистра, A=0, но занимает 32 бита 😕
mov eax,[ebp+8*edi];  принудительное поле A=0 длиной 8 бит
mov eax,[esp]
```

Обратите внимание, что, так как сумма берется по модулю 2^{32}, то можно указывать как положительные, так и отрицательные смещения, например

```
mov esi,-21; -21=2^32-21
mov eax,[edi+4*esi-8]
```

На месте первого операнда можно задавать и 16-битный регистр общего назначения, например

---

[1] Вместо 4-байтных полей i32 и m32 можно использовать двухбайтные поля i16 и m16, если задать перед командой префикс 67h, он временно (на одну команду) переключает процессор в режим двухбайтных операндов. Таким образом, для небольших операндов можно сделать команду на один байт короче, но она будет выполняться на один такт дольше, так как префикс тоже команда 😕.

```
mov ax,A[edi+4*edi]
```

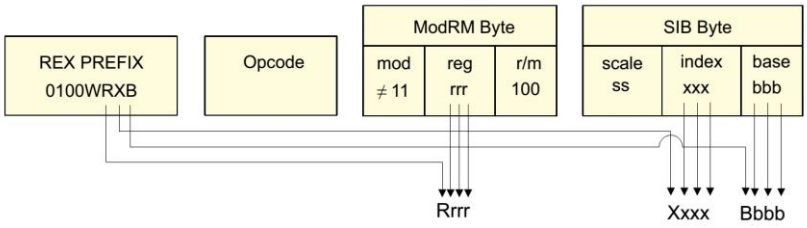
В этом случае, как уже говорилось, компилятор с Ассемблера ставит перед такой командой однобайтный префикс смены размера операнда `66h`, временно переключающий процессор для работы с 16-битными регистрами.[vi] [см. сноску в конце главы]

> Отметим общий принцип. Когда при развитии архитектуры ЭВМ в команде не хватает бит для задания нужной информации (дополнительных форматов, смены режимов работы, новых регистров и т.д.), то к команде либо добавляется служебная команда-префикс (например, `66h` или REX), либо внутрь команды вставляется служебный байт (например, байт `SIB`). В последних моделях нашей ЭВМ для графических команд можно уже использовать и три префикса.

При работе в 64-битном режиме (там уже 16 регистров каждого размера), перед командой добавляется однобайтный префикс под названием REX, который имеет показанную ниже структуру:

	7..4 бит	3 бит	2 бит	1 бит	0 бит
REX:	0100	W	R	X	B

Бит REX[W]=1 определяет 64-битный режим работы, теперь номера регистров в командах рассматриваются как `r64`, т.е. `EAX` становится `RAX`, `EBX` становится `RBX` и т.д. Бит REX[R] задаёт дополнительный старший бит в номере регистра первого операнда `r1`, так что их становится по 16 штук каждого размера, и теперь, например, доступны регистры `R8b-R15b` и `R8-R15` При необходимости задать в этом режиме 16-битный регистр первого операнда `R8w-R15w`, как и раньше, перед REX-префисом `4Ch` надо поставить ещё и префикс смены длины операнда `66h`. Биты REX[X] и REX[B] задают дополнительные биты для соответсвенно индексного `I2` и базового `B1` регистров. Как видим, всё очень запутано 😕. Ниже показана красочная схема из документации Intel, иллюстрирующая работу с битами из префикса REX:



Машинный вид остальных форматов команд рассматриваться не будет, эти команды будут изучаться только на языке Ассемблера. Напомним только, что рассматриваемые форматы команд имеют следующие мнемонические обозначения:

- **RR** –регистр – регистр;
- **RX** – регистр – память или память – регистр

(в зависимости от значения бита `d` в команде);

- **RI** – регистр – непосредственный операнд в команде;
- **SI** – память – непосредственный операнд в команде;
- **SS** – память – память, т.е. оба операнда в основной памяти.

5.9. *Команды языка машины*

Компьютеры – вещь слишком сложная, чтобы работать в принципе. Поэтому то, что они работают хоть как-то, уже чудо.

Студент DOLBY из Воронежа

Далее будет изучен синтаксис машинных команд, записанных так, как это принято в языке Ассемблера MASM (можно считать, что Вы уже начали изучать этот язык низкого уровня), и их семантика (способ выполнения команд процессором). Команды изучаются на концептуальном уровне, без рассмотрения их битового представления.

5.9.1. Команды пересылки

Фактически, ассемблерная программа наполовину состоит из команд пересылки данных.[1]

Крис Касперски ака мыщъх

Команды пересылки – одни из самых распространённых команд в языке машины. Все они пересылают значение одного, двух или четырех байт из одного места памяти в другое (в 64-битном режиме можно пересылать и по 8 байт). Для более компактного описания синтаксиса этих Ассемблерных команд вводятся следующие условные обозначения (с некоторыми из них Вы уже знакомы):

`r8` – любой из регистров `AH,AL,BH,BL,CH,CL,DH,DL`;
`r16` – любой из регистров `AX,BX,CX,DX,SI,DI,SP,BP`;
`r32` – любой из регистров `EAX,EBX,ECX,EDX,ESI,EDI,ESP,EBP`;
`m8, m16, m32, m64` – операнды в основной памяти длиной 1, 2, 4 и 8 байт;
`i8, i16, i32, i64` – непосредственные операнды в команде длиной 1, 2, 4 и 8 байт;
`SR` – один из трёх сегментных регистров `SS, DS, ES`;
`CS` – кодовый сегментный регистр.

Вот общий вид команды пересылки в нашей двухадресной ЭВМ (как уже говорилось, после точки с запятой будем записывать, как это принято в Ассемблере, *комментарий* к команде):

```
mov op1,op2; op1 := op2
```

В комментарии указано правило выполнения этой команды: копия второго операнда пересылается на место первого операнда, который, таким образом, меняется. Существуют следующие допустимые форматы первого и второго операндов команды пересылки, запишем их в виде таблицы, где во второй колонке перечислены все возможные операнды, допустимые для операнда из первой колонки:

op1	op2
r8	r8, m8, i8
r16	r16, m16, i16, SR, CS
r32	r32, m32, i32
m8	r8, i8
m16	r16, i16, SR, CS
m32	r32, i32
SR	r16, m16

В старших моделях появились операнды-регистры формата `r64` и `m64`.

Надо запомнить три основные правила задания операндов, они действуют для большинства двухадресных команд:
1) Операнды должны быть одинаковой длины.
2) Нельзя задать два операнда в основной памяти.[2]
3) Нельзя переслать значение на место *непосредственного* операнда (изменив команду).

Команды пересылок не меняют флаги в регистре `EFLAGS`. Как видим, в языке машины существует несколько десятков команд пересылок различных форматов. Из приведённой выше таблицы следует, что команды пересылок с кодом операции **`mov`** бывают форматов `RR`, `RX` (и `XR`), `RI` и `SI`.

Как ни странно, но команда пересылки, как говорится, полна по Тьюрингу, т.е. возможно, например, построить универсальную двухадресную ЭВМ (УМ-MOV 🙂), в языке которой будет *только* эта команда (правда, с «хитрыми» форматами). Например, так с помощью **`mov`** можно реализовать условный оператор

---

[1] Это преувеличение, подсчитано, что машинные программы содержат только примерно 30% команд пересылок. Для англоязычных читателей мнемоника **`mov`** сбивает с толка, так как означает «переместить», т.е. удалить с прежнего и записать на новое место. На самом деле, конечно, производится копирование данных.

[2] Исключением является команда пересылки формата `SS` (память-память), но она имеет другое мнемоническое обозначение, является безадресной, и будет изучаться в разделе 8.1, посвящённом так называемым строковым (или цепочечным) командам.

```
; if A=B then X:=0 else X:=1
  mov eax,offset A
  mov ebx,offset B
  mov [eax],0; A↑:=0
  mov [ebx],1; B↑:=1
  mov X,[eax]
```

Правда, здесь не всё так просто. В нашем алгоритме операнды A↑ и B↑ меняются (портятся), так что надо их предварительно сохранить, а потом восстановить. Что поделать, программы для простых исполнителей очень громоздкие, достаточно вспомнить машину Тьюринга. Вместо команды **mov** в этой машине с таким же успехом можно использовать и команду **xor**. В сети есть сайты, которые «компилируют» (не очень большие) программы с языка C в такую «MOV-программу», объём программы при этом, конечно, возрастает во много раз, соответственно падает скорость работы.

Иногда при задании второго операнда возникает неоднозначность его формата, например, для команды

mov eax,4; op2=i32=4 или op2=m32=<4> ?

т.е. непонятно, является ли число 4 константой i32 или адресом операнда m32, расположенного, начиная с 4-го байта памяти. По умолчанию Ассемблер считает такие операнды имеющими формат i32, а для задания формата m32 необходимо явно указать, что операнд находится в памяти, для чего используется явное указание на сегментный регистр данных ds:, например

mov eax,ds:4 или (более красиво?) **mov** eax,ds:[4] [1]

В плоской модели для *чтения* из памяти по явному адресу можно задавать любой сегментный регистр, так как все сегменты наложены друг на друга, например, **mov** eax,cs:[4], но с ds: (который подразумевается по умолчанию) команда работает на один такт быстрее. Отметим, однако, что в командах, записывающих данные в память, например **mov** ds:[4],eax, префикс cs: использовать нельзя, т.к. кодовый сегмент по умолчанию закрыт на запись.

В 64-битном режиме стало невозможно задавать операнд в виде непосредственного адреса, например, нет команды **mov** rax,ds:i64, надо использовать две команды **xor** rbx,rbx и **mov** rax,i64[rbx].

Отметим также полезную команду *обмена* содержимым двух операндов

xchg op1,op2; обмен значениями операндов:
; op1 ↔ op2.[vii] [см. сноску в конце главы]

Таблица допустимых операндов для этой команды:

op1	op2
r8	r8, m8
r16	r16,m16
r32	r32,m32
m8	r8
m16	r16
m32	r32

Как видно, здесь требуется строгое соответствие длин операндов. Эта команда также не меняет флаги. Полезной является команда загрузки исполнительного адреса в регистр

lea op1,op2; op1 := $A_{исп}$(op2)

Для этой команды существуют следующие допустимые форматы первого и второго операндов:

op1	op2
r16,r32	m8,m16,m32

При задании в качестве первого операнда r16 Ассемблер автоматически вставляет префикс смены режима 66h. Обратите внимание, что команда **lea** *никогда* не обращается в память по вычис-

---

[1] В нашей ЭВМ первые и последние 64Кб для программ пользователя закрыты на чтение и запись, так что в качестве примера вместо числа 10 надо использовать, например, число 10000h.

ленному исполнительному адресу, а просто записывает этот адрес на регистр-первый операнд. Команда **lea** не меняет флаги. Эта команда иногда позволяет эффективно запрограммировать вычисление выражений, например, вместо команд

```
; eax:=ecx+2*ebx+7
mov eax,ebx
add eax,eax; eax:=2*ebx
add eax,ecx; eax:=ecx+2*ebx
add eax,7;   eax:=ecx+2*ebx+7
```

можно использовать одну команду

```
lea eax,[7+ecx+2*ebx]; [A+B1+2*I2]
```

Вместо умножения (беззнаковых величин) можно использовать команды

```
; eax:=20*ebx
lea ebx,[4*ebx];      ebx:=4*ebx
lea eax,[ebx+4*ebx]; eax:=5*ebx=20*ebx
```

Последние две команды как говорят «спариваются» (pairing) и выполняются процессором параллельно на двух исполнительных устройствах конвейера всего за один такт! [viii] [см. сноску в конце главы]

Учтите, однако, что проконтролировать *правильность* полученного командой **lea** результата невозможно, так как никакие флаги не устанавливаются, т.е. по аналогии с языком Free Pascal её можно использовать только в режиме {$R-}. А вот в режиме {$R+} компилятор будет транслировать присваивание `eax:=ecx+2*ebx+7` (для беззнаковых величин) не в «красивую» команду

```
lea eax,[7+ecx+2*ebx]
```

а в команды

```
mov eax,ebx
add eax,eax; eax:=2*ebx
jc  Run_Time_Error
add eax,ecx; eax:=ecx+2*ebx
jc  Run_Time_Error
add eax,7;   eax:=ecx+2*ebx+7
jc  Run_Time_Error
```

При написании программы на Ассемблере, по аналогии с директивами {$R±} языка Free Pascal, лучше предусмотреть выполнения программы в обычном и отладочном режиме. Для этого можно использовать директиву условной компиляции **ifdef** имя, здесь условие истинно, если имя уже как-то описано в программе:

```
ifdef DEBUG; истинно, если DEBUG описано
mov eax,ebx
add eax,eax; eax:=2*ebx
jc  Run_Time_Error
add eax,ecx; eax:=ecx+2*ebx
jc  Run_Time_Error
add eax,7;   eax:=ecx+2*ebx+7
jc  Run_Time_Error
else
lea eax,[7+ecx+2*ebx]
endif
```

Теперь, например, задав директиву присваивания `DEBUG=1` (и, таким образом описав имя DEBUG) мы включим отладочный режим, а при отсутствии этой директивы (например, закомментировав её) будет обычный режим. Директива присваивания относится к макросредствам Ассемблера и рассматривается в Главе 11.

Начиная с процессора Intel 386, появились также команды пересылки, для операндов разной длины, с их помощью производится преобразование короткого операнда в более длинный формат с записью результата на любой регистр (а не только на регистры AX, EAX, RAX, <DX:AX> и <EDX:EAX>, как для команд **cbw**, **cwd** и **cdq**, **cwde** и **cdqe**, см. разд. 5.8.1). Для знакового расширения это команда:

movsx op1,op2; op1:=op2 (MOV Sign eXtention)

Заметим, что для команды **movsx** eax,ax есть, как уже говорилось, и короткая безадресная команда **cwde**. Для беззнакового расширения используется команда:

movzx op1,op2; op1:=op2 (MOV Zero eXtention)

Команда пересылает второй операнд в младшую часть первого, обнуляя старшую часть первого операнда. Для этих команд существуют следующие допустимые форматы первого и второго операндов:

op1	op2
r16	r8, m8
r32	r8, m8, r16, m16

Любопытно, что в 64-битном режиме команды **movzx** r64,r32/m32 нет вообще, она не нужна, так как запись в регистр r32 автоматически обнуляет старшую часть соответствующего регистра r64, т.е., например, команда **xor** eax,eax выполняется как **xor** rax,rax. Аналогично нет и команды **movsx** r64,r32/m32, вместо неё есть команда **movsxd** r64,r32/m32.

5.9.2. Арифметические команды для целых чисел

Таким образом, можно сказать, что число управляет всем миром количественного, а четыре правила арифметики можно рассматривать как полное снаряжение математика.

Джеймс Клерк Максвелл

Изучение команд для выполнения арифметических операций начнём с самых распространённых команд сложения и вычитания целых чисел (работа с вещественными числами, как уже говорилось, в этой книге изучаться не будет, пример работы с ними приведён в разд. 8.9.1).

- Команды сложения и вычитания целых чисел.

– Сложению тебя обучили? – спросила Белая Королева. – Сколько будет один плюс один плюс один плюс один плюс один плюс один плюс один плюс один?

– Я не знаю, – ответила Алиса. – Я сбилась со счёта.

– Сложения не знает, – сказала Чёрная Королева.

Льюис Кэрролл. «Алиса в Зазеркалье»

Определим вид и допустимые операнды у этих двухадресных команд:

КОП op1,op2; **КОП** = **add**, **sub**, **adc**, **sbb**

Команды с мнемоническими кодами операций (*мнемокодами*) **add** (сложение) и **sub** (вычитание) выполняются по естественной схеме:

op1 := op1 ± op2

В математике вычитание определяется как операция, обратная к сложению, поэтому действия X-1 и X+(-1) тождественны. В языке машины это не совсем верно, например, пусть X=Y=2, тогда после команд

```
sub X,1;   X=1, CF=0, OF=0
add Y,-1;  Y=1, CF=1, OF=0
```

Как видно, хотя численно результаты и одинаковы, но флаги выставлены по разному. Это легко понять, так как для, например, однобайтной переменной Y сложение **add** Y,-1 это «на самом деле» **add** Y,0FFh.

Команды с кодами операций **adc** (сложение с учётом флага переноса) и **sbb** (вычитание с учётом флага переноса) имеют *три* операнда, два из которых задаются в команде явно, а третий по умолчанию является значением флага переноса CF:

```
op1 := op1 ± op2 ± CF
```

Эти команды используются в основном для работы со сверхдлинными целыми числами, которые не могут непосредственно складываться и вычитаться командами **add** и **sub**, такие примеры приводятся во многих учебниках. Таблица допустимых операндов для команд сложения и вычитания:

op1	op2
r8	r8, m8, i8
r16	r16, m16, i8, i16
r32	r32, m32, i8, i32
m8	r8, i8
m16	r16, i16
m32	r32, i8, i32

Глядя на эту таблицу, мы замечаем, что, когда второе слагаемое константа, то «маленькие» константы имеют длину в один байт (производится знаковое расширение до нужной длины).

В результате выполнения всех этих операций всегда изменяются флаги CF, OF, ZF, SF, которые отвечают соответственно за перенос, переполнение, нулевой результат и знак результата (флагу SF всегда присваивается знаковый бит результата). Эти команды меняют и некоторые другие флаги, но здесь они рассматриваться не будет.

В новых процессорах у команды **adc** появились две модификации: **adcx** и **adox**. Команда **adcx** работает так же, как и команда **adc**, но меняет только флаг CF (не меняя, в частности, флаг OF), а команда **adox** выполняется по правилу

```
op1 := op1 + op2 + OF
```

она меняет только флаг OF, записывая в него (а не во флаг CF) бит переноса (а не переполнения!). Таблица допустимых операндов для этих команд:

op1	op2
r32	r32, m32
r64	r64, m64

Эти команды можно использовать (чередуя их между собой) для более эффективной реализации алгоритма сложения сразу 2-х пар длинных чисел при использовании конвейера (так как у них будет меньшая зависимость по данным, см. разд. 14.2).[1]

При программировании иногда полезны также следующие *унарные* арифметические операции:

neg op1; обратная величина знакового числа, `op1 := 0-op1`,
inc op1; увеличение (инкремент) аргумента на единицу, `op1 := op1+1`,
dec op1; уменьшение (декремент) аргумента на единицу, `op1 := op1-1`.

Здесь операнд op1 может быть форматов r8, m8, r16, m16, r32, m16 (и r64 и m64 в 64-битном режиме). Применение этих команд вместо соответствующих по действию команд вычитания и сложения приводит к более компактным программам. Необходимо, однако, отметить, что команды `inc op1` и `dec op1`, в отличие от эквивалентных им более длинных команд `add op1,1` и `sub op1,1` никогда не меняют флаг CF (оставляют его прежнее значение). Последнее обстоятельство замедляет выполнение команд **inc** и **dec** на конвейере современных процессоров (см. разд. 14.2), поэтому вместо них рекомендуется всё же использовать команды **add** и **sub**, хотя они и длиннее на один байт.

К сожалению, при переходе в 64-битный режим команды **inc** и **dec** с операндом-регистром перестают быть однобайтными (коды операций этих однобайтных команд 40h-4Fh отданы под команды-префиксы 64-битных регистров).

---

[1] Эти команды входят в так называемое расширегие языка команд BMI2 (Bit Manipulation Instruction Set 2), это расширение впервые реализовано в процессоре Haswell, оно входит не во все машинные языки процессоров фирм Intel и AMD.

Начиная с процессора Intel 486 появилась «хитрая» модификация команды сложения

xadd op1,op2; **ex**change and **add** – обмен, затем сложение

Таблица допустимых операндов для этих команд:

op1	op2
r8, m8	r8
r16, m16	r16
r32, m32	r32
r64, m64	r64

При выполнении этой команды сначала производится операция обмена значениями аргументов **xchg**(op1,op2), а лишь затем собственно сложение. Для этой команды второй операнда op2, который после обмена становится первым операндом op2), может быть только регистром. Например, вот получение при помощи этой команды n-го числа Фибоначчи (если при первом чтении что-то будет непонятно, надо вернуться сюда после освоения Ассемблера):[1]

```
  xor  eax,eax; Fib0:=0
  mov  edx,1;   Fib1:=1
  mov  ecx,n
L:xadd eax,edx
  loop L
; теперь eax=n-е число Фибоначчи=Fibn
```

Заметьте, что команда **xadd** меняет *оба* свои операнда, а флаги устанавливаются по значению суммы op1+op2. С префиксом блокировки **lock xadd** r32,m32 эта команда до своего окончания блокирует доступ к памяти, что заставляет «замереть» все остальные процессорные ядра ЭВМ примерно на 1000 тактов ⚠. Так же ведут себя описанные позже команды **cmpxchg** и **cmpxchg8b**.

- Команды умножения целых чисел.

Существует множество вещей, с которыми мы свыкаемся, не понимая их.

Айзек Азимов. «Конец Вечности»

Формат этих команд накладывает сильные ограничения на месторасположение их операндов, это одноадресные команды, очень похожие на команды учебной одноадресной ЭВМ УМ-1. Первый операнд и результат всех этих команд явно не указываются и находятся в фиксированных регистрах, заданных *по умолчанию*. Есть следующие команды умножения, в них, как и в уже знакомой Вам учебной одноадресной ЭВМ УМ-1, явно задаётся только второй операнд (т.е. второй сомножитель):

mul op2; беззнаковое целое умножение (MULtiply)
imul op2; знаковое целое умножение (sIgn MULtiply)

Итак, в самой команде явно задаётся только второй операнд op2, он может быть форматов r8 и m8 (соответственно, тогда говорят о коротком умножении), r16 и m16 (это длинное умножение) или форматов r32 и m32 (сверхдлинное умножение).

Обратите особое внимание на то, что операнд op2 не может быть форматов i8, i16 и i32 (это типичная ошибка учащихся, очень уж им хочется, чтобы такая команда была 🙂)[2]. Как можно заметить, в отличие от команд сложения и вычитания, умножение и деление знаковых и беззнаковых целых чисел выполняются *разными* командами (по разным алгоритмам), беззнаковвые операции работают немного быстрее, чем знаковые. То, что эти алгоритмы различаются, можно понять, если, например, вспомнить знакомое нам ещё со школы правило умножения знаковых чисел «минус на минус даёт плюс».

В случае с коротким вторым операндом форматов r8 и m8 при умножении вычисление результата производится по формуле:

---

[1] Пример из работы анонимного автора 2014 года издания "Ассемблерные хаки из книги «xchg rax,rax»".

[2] Для любознательных. В коде операции тот бит, который отвечает в командах сложения и вычитания за задание непосредственных операндов, для команд умножения и деления потрачен на различение знаковых и беззнаковых операций.

```
AX := AL * op2
```

В случае с длинным вторым операндом форматов r16 и m16 при умножении вычисление производится по формуле:

```
<DX:AX> := AX * op2
```

Как видим, в этом случае произведение располагается сразу в двух регистрах <DX:AX> (как уже упоминалось, это называется *регистровой парой*).

В случае со сверхдлинным вторым операндом форматов r32 и m32 при умножении вычисление производится по формуле:

```
<EDX:EAX> := EAX * op2
```

В 64-битных ЭВМ у второго операнда добавляются форматы r64 и m64. Первый сомножитель располагается при этом в регистре RAX, а произведение будет в регистровой паре <RDX:RAX>.

Обратите внимание, что умножение (в отличие от сложение и вычитания) всегда даёт точный результат, так как под него отводится в два раза больше байт, чем под каждый сомножитель.

После выполнения команд умножения устанавливаются флаги переполнения и переноса (CF и OF). Как известно, для операций сложения и вычитания эти флаги сигнализируют о беззнаковом и знаковом переполнении результата. При умножении, однако, ошибок переполнения *никогда* не бывает, и эти флаги используются для другой цели. Они устанавливаются по следующему правилу: CF=OF=1, если в произведении столько значащих (двоичных) цифр, что они не помещаются в *младшей* половине произведения. На практике это означает, что при значениях флагов CF=OF=1 произведение коротких целых чисел не помещается в регистр AL и частично «переползает» в регистр AH. Аналогично произведение длинных целых чисел не помещается в регистре AX и «на самом деле» занимает оба регистра <DX:AX> и т.д. И наоборот, если CF=OF=0, то в старшей половине произведения (соответственно в регистрах AH, DX и EDX) находятся только *незначащие* двоичные цифры произведения. Напоминаем, что это двоичные нули для положительных и двоичные единицы для отрицательных произведений. Другими словами, при CF=OF=0 в качестве результата произведения можно взять только его младшую половину, что может оказаться полезным при программировании. Флаги ZF и SF после команд умножения принимают неопределённые значения («портятся»).

Рассмотрим пример. Пусть на языке Free Pascal описано

```
var x,y: word; z: longword;
begin {$R+} y:=x*y; z:=x*y
```

Этот фрагмент на Паскале можно «откомпилировать» в такой фрагмент на Ассемблере (команды переходов, например, **jc**, мы будем изучать позже):

```
x: dw ?
y: dw ?
z: dd ?
; x:=x*y; z:=x*y
   mov ax,x
   mul y; <dx:ax>:=x*y
   jc  Range_Checking_Error
   mov y,ax; x*y помещается в y
   mov word ptr z,ax
   mov word ptr z+2,dx; z:=longword(x*y)
```

Кроме того, есть ещё два формата команд умножения целых чисел, это двухадресная команда

```
imul op1,op2; op1:=op1*op2
```

Таблица допустимых операндов для этой команды:

op1	op2
r16	r16, m16, i8, i16
r32	r32, m32, i8, i32

Операнды i8, i16 и i32 рассматриваются как знаковые, причём маленькие константы занимают один байт,[1] а большие 2 или 4 байта. Когда произведение не помещается в первый операнд, то команда даёт неправильный (усечённый) результат, при этом устанавливаются флаги CF=OF=1, иначе CF=OF=0. Флаги ZF и SF после этой команды принимают неопределённые значения.

Существует также и трёхадресная команда умножения, очень похожая на команду учебной машины УМ-3.

imul op1,op2,op3; op1:=op2*op3

Таблица допустимых операндов для этой команды:

op1	op2	op3
r16	r16, m16	i16
r32	r32, m32	i32

Операнды i16 и i32 рассматриваются как знаковые. Когда произведение помещается в первый операнд, то, как обычно, устанавливаются флаги CF=OF=0, иначе произведение усекается и устанавливаются флаги CF=OF=1. Флаги ZF и SF после этой команды принимают неопределённые значения.

Двух и трёхадресные команды умножения выполняются быстрее, чем одноадресная (т.к. вычисляется только *младшая* половина произведения), и не требуют обязательного использования регистра EDX. Заметим, что для этих команд существует только знаковая форма (**imul**), так как в результате получается младшая половина «полного» произведения, а она для знаковых и беззнаковых сомножителей одинаковая (постарайтесь понять, почему это так) поэтому беззнаковые команды умножения этих форматов не нужны.

Эти «усечённые» команды умножения часто используются компиляторами с языков высокого уровня. Дело в том, что эти языки при реализации арифметических операций (умножения, сложения и т.д.) получают результат, по размеру не превышающий 32-х бит (как на 32-битных, так и на 64-битных ЭВМ). Например:

var X,Y: longint; Z: int64; . . . Z:=X*Y

Оператор Z:=X*Y примерно так компилируется на язык Ассемблер

```
mov  eax,X
imul eax,Y; OF=1 при ошибке
cdq
mov  dword ptr Z,eax
mov  dword ptr Z+4,edx
```

Т.е. при умножении X*Y получается (вообще говоря, неправильный) результат длиной 32 бита, который затем, после знакового расширения до 64 бит, присваивании переменной Z 🙁. И только тогда, когда хотя бы один операнд по размеру больше, чем 32 бита, то производится вычисления с разрядностью 64 бита. Таким образом, чтобы получать всегда правильный результат операции умножения на языках высокого уровня, программистам надо использовать явное преобразование типов Z:=int64(X)*Y, тогда, хотя и немного медленнее, будет получен код с «настоящим» (всегда правильным) произведением:

```
mov  eax,X
imul Y
mov  dword ptr Z,eax
mov  dword ptr Z+4,edx
```

В новых процессорах у команды *беззнакового* умножения **mul** op2 появилась модификация

mulx $r32_1$,$r32_2$,op2; <$r32_1$:$r32_2$>:=eax*op2

Она снимает ограничение на размещение произведения только в регистровой паре <EDX:EAX>, размещая результат в двух любых 32-битных регистрах <$r32_1$,$r32_2$>. Эта команда, в отличие от **mul**, не меняет флаги, а второй операнд op2 может быть только форматов r32 или m32. Такую ко-

---

[1] Константа i8 знаково расширяется до 2-х или 4-х байт.

манду можно использовать для более эффективной реализации алгоритма умножения длинных чисел [ix] [см. сноску в конце главы].

- Команды деления целых чисел.

Не в совокупности ищи единства, но более – в единообразии разделения 🙄.
Козьма Прутков.

Формат команд деления, как и команд умножения, накладывает сильные ограничения на месторасположение их операндов, это одноадресные команды, очень похожи на команды учебной одноадресной ЭВМ. Первый операнд и результат всех этих команд явно не указываются и находятся в фиксированных регистрах, заданных *по умолчанию*. Есть следующие команды деления, в них, как и в уже знакомой Вам учебной одноадресной ЭВМ УМ-1, явно задаётся только второй операнд (т.е. делитель):

div op2; беззнаковое целое деление (DIVision)
idiv op2; знаковое целое деление (sIgn DIVision)

Итак, в самой команде явно задаётся только второй операнд op2, он может быть форматов r8 и m8 (соответственно, тогда говорят о коротком делении), r16 и m16 (это длинное деление) или форматов r32 и m32 (сверхдлинное деление).[1] Обратите особое внимание на то, что операнд op2 не может быть форматов i8, i16 и i32 (это типичная ошибка учащихся, очень уж им хочется, чтобы такая команда была 🙂). При делении на короткий операнд форматов r8 и m8 производятся следующие действия (операции **div** и **mod** здесь понимаются в смысле языка Free Pascal):

```
AL := AX div op2
AH := AX mod op2
```

При делении на длинный операнд формата r16 и m16 вычисление производится по формулам:

```
AX := <DX:AX> div op2
DX := <DX:AX> mod op2
```

В этих командах операнд запись <DX:AX> обозначает 32-разрядное целое число, расположенное сразу в двух регистрах DX и AX (регистровой паре). При делении на сверхдлинный операнд формата r32 и m32 производятся вычисления:

```
EAX := <EDX:EAX> div op2
EDX := <EDX:EAX> mod op2
```

В этих командах операнд запись <EDX:EAX> обозначает 64-разрядное целое число, расположенное сразу в двух регистрах EDX и EAX, а op2, как уже говорилось, может иметь формат r32 или m32.[2]

Заметьте, что все команды деления *одновременно* получают два результата (целое частное и целый остаток). Именно поэтому под результат этой операции отводится в *два раза больше места*, чем под делитель. Ниже показана схема выполнения короткого, длинного и сверхдлинного деления.

	AX	: op2	→	AH=**mod**	AL=**div**	
DX	AX	: op2	→	DX=**mod**	AX=**div**	
EDX	EAX	: op2	→	EDX=**mod**		EAX=**div**

В отличие от команд умножения, которые *всегда* дают точный результат, команды деления могут вызывать аварийную ситуацию (исключение), если частное не помещается в отведённое для него место, т.е. в регистры AL, AX и EAX соответственно. Такая ситуация называется целочисленным *переполнением*, при этом происходит прерывание вычислительного процесса, что, как правило, приво-

---

[1] В 64-битном режиме есть также форматы r64 и m64 (учетверённое деление).

[2] В 64-битном режиме добавляются форматы r64, m64. Делимое при этом будет в регистровой паре <RDX:RAX>. В языках высокого уровня операции деления, дающие сразу частное и остаток, встречаются редко. Например, в языке Python: **divmod**(9,4)→(2,1); здесь, однако, ответом является кортеж (вектор) из двух целых чисел.

дит к аварийному прекращению выполнения программы. Похожую аварийную ситуацию вызывает и деление на ноль, к сожалению, у них одинаковые номера аварийных ситуаций. В то же время заметьте, что остаток от деления *всегда* помещается в отводимое для него место на регистрах `AH`, `DX` или `EDX` соответственно.

Команды деления после своего выполнения как-то устанавливают некоторые флаги, но никакой полезной информации из значения этих флагов программист извлечь не может (так как результатов два, то вообще непонятно, как с пользой установить эти флаги). Можно сказать, что деление «портит» определённые флаги (в частности, портятся «полезные» флаги `CF`, `OF`, `ZF` и `SF`).

Обратите внимание, в отличие от операций целочисленного сложения и вычитания, которые могут выполняться на *любых* регистрах общего назначения, команды умножения и деления по существу «припаяны» только к регистрам `EAX` и `EDX`. Такое решение принято «по бедности», из-за недостатка у конструкторов первых ЭВМ этой серии аппаратных ресурсов. Для команд умножения в следующем поколении наших компьютеров это ограничение частично снято для рассмотренных выше двух и трёхадресных команд умножения, но они не всегда дают точный результат или (для команды **`mulx`**) существуют только в беззнаковом варианте. В последних моделях были добавлены новые, так называемые векторные регистры (см. Главу 17), на каждом из которых уже можно производить *все* арифметические операции, причём как с целыми, так и с вещественными числами.

Стоит отметить, что команды целочисленного деления даже на современных процессорах выполняются примерно в 3-5 раз медленнее, чем команды умножения (знаковое деление медленнее, чем беззнаковое). Исходя из этого оптимизирующие компиляторы заменяют команду деления «хитрыми» командами умножения. [x] [см. сноску в конце главы]

Существуют и достаточно специфичные команды для работы с целыми числами. Например, команда **`rdrand`** (**`r`**ea**`drand`**):

```
rdrand op1; op1=r32
```

Когда команда после выполнения устанавливает `CF=1`, то в операнде-регистре возвращается «настоящее» беззнаковое целое случайное число, полученное *аппаратным* генератором случайных чисел. При `CF=0` генератор «не успевает» за процессором, новое случайное число ещё не готово, и надо просто немного подождать, потом выполнить команду снова:

```
L: rdrand EAX
   jnc    L; Результат пока не готов
```

Как уже упоминалось, для работы со (знаковыми) целыми числами можно использовать и вещественные регистры.

Вопросы и упражнения

Образование есть то, что остаётся после того, когда забывается всё, чему нас учили 🙂.

Альберт Эйнштейн

1. Что такое специализированные и универсальные ЭВМ ?
2. Чем отличаются модели семейства ЭВМ друг от друга ?
3. Что такое программная совместимость и почему она является обязательной в любом семействе ЭВМ?
4. Что такое в архитектуре процессоров Intel машинное слово ?
5. Какое представление вещественного числа называется нормализованным ?
6. Используя какой-нибудь язык программирования высокого уровня (скажем, Паскаль) получите такое значение вещественной константы `A`, чтобы для числа $X=10^4$ выполнялось машинное равенство `X+A=A`.
7. Что такое вещественное значение `NaN` «*не число*» и для чего оно нужно ?
8. Для чего может потребоваться представлять в программе целые числа одновременно в двух машинных системах счисления – знаковой и беззнаковой ?
9. Для чего необходимы сегментные регистры ?
10. Что такое перевёрнутое представление целых чисел и для чего оно может быть нужно ?

11. Почему на регистрах, в отличие от основной памяти, числа хранятся в обычном (не перевёрнутом) виде?
12. Почему целые числа хранятся в памяти в перевёрнутом виде, а команды формата RR и вещественные числа в прямом виде ?
13. Для чего нужен оператор **ptr** ?
14. Что такое бит размера операнда w в машинной команде ?
15. Чем адрес байта памяти в команде отличается от его логического адреса ?
16. Что такое регистр-модификатор ?
17. Что такое задание операндов команды по умолчанию ? Какие операнды задаются по умолчанию в командах целочисленного умножения и деления ?
18. Почему, в отличие от команд сложения и вычитания, необходимы различные команды для умножения и деления знаковых и беззнаковых целых чисел ?
19. Объясните, какой будет результат выполнения команды `div edx` ⚠ ?
20. Объясните, почему в общем случае для реализации операции `x div y` (где x и y – целочисленные операнды размером в слово) необходимо использовать команду длинного, а не короткого деления.

---

[i] Для продвинутых читателей. Операции сложения и вычитания целых чисел реализованы в компьютере как операции по модулю 2^N, где, как уже говорилось, значение N равно максимальному числу бит в представлении целого числа. В математике эти целые числа образуют кольцо вычетов по модулю 2^N. Заметим, что операция умножения определяется в этом кольце через операцию сложения, а вот операции деления вообще нет, и с этим надо что-то делать .

Такая арифметика обычно называется циклической (wraparound). С математической точки зрения $X_{доп} = X \bmod 2^N$, где операция **mod**, определена как дающая неотрицательный результат. Так принято в математике и описано в стандарте Паскаля, но все ЭВМ отступают здесь от стандарта, давая *знаковый* остаток. Например, в школе Вас учили, да и компьютер считает, что `-1:256=0` (частное) и `(-1)` (остаток), т.е. частное округляется в бóльшую сторону (к *положительной бесконечности*), в Паскале это операция `Trunc`. А вот в математике будет `-1:256=-1` (частное) и `255` (остаток), это округление частного в меньшую сторону (к *отрицательной бесконечности*), в Паскале это `Round`. Из такого определения дополнительного кода следует, что для получения, например, дополнительного кода суммы двух чисел достаточно сложить дополнительные коды слагаемых без анализа их знаков. К сожалению, для операций умножения и деления это уже не верно, и приходится использовать разные алгоритмы для знаковых и беззнаковых чисел.

[ii] Для продвинутых читателей. На самом деле в современных процессорах короткие регистры часто физически не являются частью более длинных, где-то «в глубине» микросхемы они реализованы по отдельности и хранятся в полном регистре. Например, после команды `mov al,1` процессор меняет регистр AX не сразу, а только после того, как он будет использован как операнд в команде, например `add ax,1` (а может регистр AX и совсем не потребуется, чего зря работать 🙂). Правда, если потом будет команда `push eax`, то придётся потратить пару микроопераций для сборки EAX из составляющих его частичных регистров. Большинство компиляторов с языков высокого уровня по возможности избегают использования частичных регистров, например:

```
mov al,[ebx]          movzx eax,byte ptr [ebx]
cmp al,'*'      →     cmp   eax,'*'
```

Далее, в 32-битной архитектуре изменение младшей части регистра (например, AX или AL для EAX) оставляет старшую часть неизменной. В 64-битной архитектуре изменение младшей половины 64-битного регистра, например, регистра EAX (но не AX или AL !) в RAX обнуляет старшую половину этого регистра. Такие же правила действуют и в отношении использования младших частей векторных регистров ZMM → YMM → XMM, старшие части этих регистров тоже обнуляются. Такое «нелогичное» использование регистров призвано не создавать так называемых ложных зависимостей по данным, что позволяет более эффективно использовать конвейер процессора.

В 64-битном режиме уже 16 регистров общего назначения, они показаны в приведённой ниже таблице:

Регистры общего назгачения x86-64

Биты 63-0	Биты 31-0	Биты 15-0	Биты 15-8	Биты 7-0
RAX	EAX	AX	AH	AL
RBX	EBX	BX	BH	BL
RCX	ECX	CX	CH	CL

RDX	EDX	DX	DH	DL
RSI	ESI	SI		SIL
RDI	EDI	DI		DIL
RBP	EBP	BP		BPL
RSP	ESP	SP		SPL
R8	R8D	R8W		R8B
R9	R9D	R9W		R9B
R10	R10D	R10W		R10B
R11	R11D	R11W		R11B
R12	R12D	R12W		R12B
R13	R13D	R13W		R13B
R14	R14D	R14W		R14B
R15	R15D	R15W		R15B

Все регистры имеют отдельную адресацию своей младшей половина (разряды 31-0), младшей четверти (разряды 15-0) и младшей «восьмушки» (разряды 7-0). Кроме того, первые четыре регистра `RAX,RBX,RCX,RDX` имеют адресуемую часть `AH,BH,CH,DH` в разрядах 15-8.

В 64-битной архитектуре доступ к старшим 8-битным регистрам `AH`, `BH`, `CH` и `DH` уже ограничен. Их нельзя использовать в одной команде вместе со всеми 64-битными регистрами и с младшими 8-битными регистрами `R8b-R15b`, так как их номера в этом случае отдаются новым 8-битным регистрам `SIL`, `DIL`, `BPL` и `SPL` (это младшие части регистров `SI`, `DI`, `BP` и `SP`, старшие части этих регистров как самостоятельные по-прежнему использовать нельзя). Например, можно `mov al,dl`, но нельзя `mov ah,r8b`, `movzx rbx,ah` и т.д.

Добавлены 8 дополнительных 64-битных регистров общего назначения `R8-R15`, у каждого из которых можно обращаться к младшим 32-х, 16-ти и 8-ми битам, например, `R9` (64 бита), `R9d` (32 бита), `R9w` (16 бит) и `R9b` (8 бит).

По умолчанию в 64-битном режиме длина адреса равна 8 байт, а вот длина данных – 4 байта (`r32` или `m32`), а для работы с регистрами `r64` используется команда-префикс REX. В этом префиксе, в частности, указываются старшие биты в номерах регистров, что позволяет увеличить количество 64-битных регистров с 8 до 16. Использование префикса, естественно, увеличивает длину и время выполнения команды.

[iii] Современные 64-битные компьютеры могут работать в двух основных режимах:

- Режим совместимости с 32-битный режимом (Compatibility Mode), здесь современные 64-битные операционные системы могут правильно выполнять программы, написанные для основного, так называемого защищённого режима (Protected Mode) предыдущих, 32-битных ЭВМ.
- Режим 64-битных компьютеров (64-Bit Mode) – основном режиме работы современных ЭВМ.

Так что 32-битные программы по-прежнему могут выполняться.

[iv] Для продвинутых читателей. Значения регистров `DS`, `ES` и `SS` полностью совпадают (содержат селектор одного и того же дескриптора сегмента). Регистр `CS` ссылается на другой дескриптор, но у него такой же адрес начала сегмента и длина. Плоская модель памяти полностью снимает с сегментов функцию контроля прав доступа (разрешено ли в сегменте чтение, запись и выполнение команд). Действительно, хотя кодовый сегмент и закрыт на запись (чтобы не было возможности "испортить" команды), но это "фиктивная" защита, так как в эту область логической памяти возможна запись, например, через сегмент данных 🙁.

В архитектуре процессоров Intel, однако, существует и второй уровень контроля привилегий и прав доступа в память, это контроль на уровне так называемых *страниц* памяти. Дело в том, что отведённая задаче логическая память делится на одинаковые страницы, обычно размером в 4096 байт. *Каждая* страница, как и сегмент, имеет аналогичные атрибуты привилегий и прав доступа. Программа пользователя делится на секции по функциональному назначению, т.е. программист описывает секцию команд, секцию данных, секцию констант и т.д. При размещении секций в памяти они занимают целое число страниц и все их страницы получают соответствующие атрибуты. Например, все страницы секции кода имеют разрешения на исполнение команд и чтение данных, но закрыты на запись.

Обычно говорится, что такая память имеет *сегментно-страничную* организацию, хотя, как уже говорилось сама физическая память об этом даже не догадывается, весь механизм контроля возлагается на процессор. Программист на Ассемблере может до начала счета установить у страниц каждой секции нужные ему атрибуты доступа (например, открыть секцию команд на запись), такой пример будет приведён в разд. 9.2. Кроме того, он может во время счета менять атрибуты каждой страницы по отдельности (разумеется, только у *своей* памяти). Полностью тема организации сегментно-страничной памяти изучается в курсе по операционным системам.

[v] Для продвинутых читателей. Наличие в команде бита направления приводит к тому, что по сути одна и та же команда может кодироваться разными способами. Например, команда **mov** ecx,eax имеет код 03C8h с битом d=1 и код 01C8h – это **mov** eax,ecx с d=0.

Для формата RI (регистр-непосредственный операнд) операнды менять местами нельзя (формата IR нет), поэтому вместо бита d (direction) задаётся бит размера непосредственного операнда s (size). Бит s=1 вместе с битом w=1 принудительно задают размер непосредственного операнда как i8 (1 байт) вместо стандартного i32 (4 байта), что позволяет сэкономить 3 байта. Любопытно также отметить, что практически для всех команд форматов RI и RR, если первым операндом является AL/EAX, существует аналог на один байт короче, где AL/EAX не указан (задан по умолчанию), что тоже позволяет сэкономить один байт. Например, команда **mov** ebx,1 имеет длину 2 байта, а команда **mov** eax,1 – только один байт. Следовательно, в командах предпочтительно использовать регистры AL/EAX. При этом, к сожалению, у многих команд на Ассемблере появляются различные представления в битовой кодировке. Например, команда **xchg** eax,eax с кодами 90h (явно задан только второй регистр, первый EAX по умолчанию) и 87h,C0h (стандартный формат RR, два байта):

87h,c0h = **xchg**=100001 d=1 w=1 RR=11 eax=000 eax=000

[vi] Для продвинутых читателей. Как уже упоминалась, такая сложная адресация у процессоров Intel приводит к тому, что по сути одна и та же команда имеет разные битовые представления. Например, команду Ассемблера **mov** eax,[esi] можно закодировать такими семью способами:

Команда	Представление в памяти
mov eax,0[esi]	8B 06 **06**=(mod=00,eax=000,mem=110=esi)
mov eax,(i8=0)[esi]	8B 46 00 **46**=(mod=01=i8,eax=000,mem=110=esi)
mov eax,(i32=0)[esi]	8B 86 00000000 **86**=(mod=10=i32,eax=000,mem=110=esi)
mov eax,0[esi+0*I2]	8B 04 26 **04**=(mod=00,eax=000,mem=100=SIB) **26**=(Scale=00=1,I2=100=SIB=>**I2=0**,B1=110=esi)
mov eax,(i8=0)[esi+0*I2]	8B 44 26 00 **44**=(mod=01,eax=000,mem=100=SIB) **26**=(Scale=00=1,I2=100=SIB=>**I2=0**,B1=110=esi)
mov eax,(i32=0)[esi+0*I2]	8B 84 26 00000000 **84**=(mod=10,eax=000,mem=100=SIB) **26**=(Scale=00=1,I2=100=SIB,B1=110=esi)
mov eax,(i32=0)[0*B1+1*esi]	8B 04 35 00000000 **04**=(mod=00,eax=000,mem=100=SIB) **35**=(Scale=00=1,I2=110=esi,B1=101=**ebp**=>**B1=0**)

Язык процессоров Intel очень сложный и запутанный. Когда многообразие комбинации значения полей mod и mem для конкретной команды не нужно, то эти поля используются (дополнительно к полю КОП) для задания других кодов операций.

Отметим, что в 64-битном режиме является способ адресации, при котором в качестве базового используется регистр-счётчик адреса RIP, который, как Вы знаете, при выполнении каждой команды указывает на начало следующей команды. Этот формат можно записать как r64,i32[rip]. Главное преимущество здесь в том, что как базовый регистр RIP не надо специально загружать, на нём всегда есть база в виде адреса следующей команды. При этом, вместо «обычного» формата r8,m64 с длинным 64-битным абсолютным адресом используется «нестандартный» формат r8,i32 с 32-битным относительным адресом i32=Δ (сэкономили 4 байта). Вообще говоря, в 64-битной архитектуре допускаются и «красивые» машинные команды

mov al,[rip] **lea** rax,[rip+rbx] и т.д.,

но 64-битный компилятор ml64 считает, что программисту их использовать бесполезно (он не знает конкретного значения RIP), поэтому «не понимает» их в тексте программы 🐷.

[vii] Для продвинутых читателей. Команда **xchg** формата регистр-регист на современных процессорах не занимает при выполнении ни одного такта работы конвейера, её выполнение сводится просто к переименованию регистров и производится на стадии декодирования (более подробно о регистровом файле конвейера см. разд. 14.2.1).

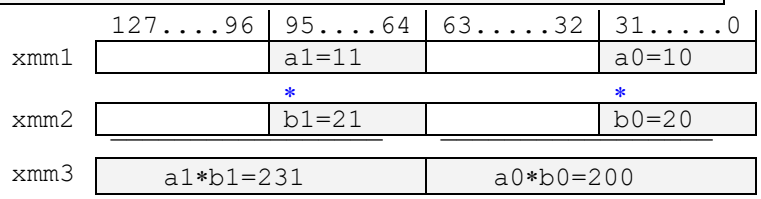
Команда **xchg** формата регистр-память читает старое значение некоторой переменной из оперативной памяти и сразу записывать в эту же переменную новое значение из регистра. Такие команды называют *атомарными* (т.е. неделимыми), они выполняются с блокировкой шины связи с оперативной памятью (или памятью типа кэш), что не позволяет другим устройствам (в частности, другим процессорным ядрам) производить в это время обмен с этой памятью. Сейчас эта команда выполняется примерно за 8 тактов процессора + 23 такта задержки (Latency), в то время как, например, команда `add r32,m32` за 2 такта, а команда `xchg r32,r32` совсем не занимает тактов конвейера. Команду **xchg** можно использовать для синхронизации параллельных процессов с помощью так называемых *семафоров*, эта тема изучается в курсе по операционным системам, в обычных программах команду **xchg** формата регистр-память следует избегать.

Остальные команды, которые обращается к памяти (кроме команды **mov**), например, `add m32,r32`, можно сделать атомарными, поставив перед ними команду-префикс **lock** с кодом операции `0F0h`. Отметим, что даже простое чтение из памяти `mov eax,X` может быть неатомарным, если переменная X не выровнена на границу 4-х байт (попадает в разные строки кэш памяти). Следует отметить, что атомарность операции в архитектуре Intel обеспечивается даже в многоядерных процессорах, когда копия переменной из памяти присутствует в кэше другого ядра (здесь работает сложный аппаратный алгоритм, обеспечивающий так называемую когерентность кэшей).

[viii] Для продвинутых читателей. Как будет ясно далее из описания схемы конвейера, практически вся работа по выполнению этой команды производится на подготовительных этапах (декодирования и вычисления адресов операндов). Так образом, собственно вычислительные устройства конвейера не используются, что позволяет выполнять команду **lea** параллельно с остальными командами (сложения, вычитания, логическими и т.д.). Иногда даже говорят, что **lea** выполняется на конвейере за ноль тактов.

[ix] Для продвинутых читателей. Точный результат произведение целых чисел может давать и на векторных регистрах. Например, вот команда умножения двух пар беззнаковых 32-разрядных целых чисел на 128-разрядных векторных регистрах XMM1 и XMM2:

```
a01  dd 10,?,11,?; a0=10,?; a1=11,?
b01  dd 20,?,21,?; b0=20,?; b1=21,?
a0b0 dq ?; var a0b0: qword;
a1b1 dq ?; var a1b1: qword;
     vmovdqa xmm1,xmmword ptr a01; xmm1:=?,a1,?,a0
     vmovdqa xmm2,xmmword ptr a01; xmm1:=?,b1,?,b0
     pmuludq xmm3,xmm1,xmm2; dd*dd → dq
; xmm3[63..0]:=xmm1[31..0]*xmm2[31..0]=a0*b0
; xmm3[127..64]:=xmm1[95..64]*xmm2[95..64]=a1*b1
     vmovdqa xmmword ptr a0b0,xmm3; dd*dd → dq
; a0b0:=xmm3[63..0]=a0*b0; a1b1:=xmm3[127..64]=a1*b1
```



Аналогичные команды есть и для 256-разрядных YMM векторных регистров (умножение 4-х пар чисел) и 512-разрядных ZMM векторных регистров (умножение 8-ми пар чисел).

---

[x] Для продвинутых читателей. Рассмотрим, например, *беззнаковое* деление числа X формата **dd** на число 10 (`X:=X div 10`). Классический вариант на Ассемблере выглядит так:

```
mov eax,X;   eax:=X
xor edx,edx; edx:=0
mov ebx,10
div ebx; eax:=X div 10
;        edx:=X mod 10
mov X,eax
```

Рассмотрим теперь замену деления на умножение оптимизирующим компилятором (например, компилятором языка Free Pascal, ну, или *хорошим* программистом на Ассемблере ☺):

```
   mov eax,4294966730; =Round(2^32/10)
   mul X; <edx:eax>:=X*Round(2^32/10)
;  edx=X*(Round(2^32/10)) div 2^32=X div 10 !
   mov X,edx; X:=X div 10
```

Первый вариант выполняется примерно за 40 тактов процессора, а второй – всего за 8 (!). Замена деления умножением базируется на работе с так называемыми вещественными числами с *фиксированной точкой*. В таких числах точка, отделяющая целую часть числа от дробной, не указывается в явном виде, а подразумевается находящейся в определённой позиции целого числа. В приведённом примере такая точка в 64-разрядном числе `<EDX:EAX>` (после команды `mul X`) располагается как раз между этими двумя регистрами. Аналогичный подход можно применять и для случая, когда требуется выполнять много делений на *переменную* X: надо один раз вычислить величину `K=Round(2^32/X)`, но здесь есть трудности с обеспечением точности.

Для знакового деления нужно дополнительно произвести корректировку результата с помощью знакового бита числа X:

```
mov  eax,4294966730
imul X
shl  X,1;     знаковый бит в CF
adc  edx,0;  корректировка EDX
mov  X,edx; X:=X idiv 10
```

Зная `M=X div K` можно вычислить и остаток от деления `X mod K=X-K*M`, где `K*M` вычисляется для небольших значений K без умножения (обычно с помощью команды **lea** и сдвигов).

Любопытно, что с помощью целочисленных операций иногда можно обрабатывать и вещественные числа. Одной из знаменитых является задача приближённого вычисления значения функции (для x≥0):

$$y = \frac{1}{\sqrt{x}}$$

На языке Free Pascal:

```
var x,y: single;
    a: longword absolute x; b: longword absolute y;
begin b:=$5F3759DF - a shr 1; {y:=1/sqrt(x)}
```

На Ассемблере:

```
.data
x real4 ?
y real4 ?
.code
  mov ebx,x
  shr ebx,1
  mov eax,5F3759DFh; магическая константа 🙂
  sub eax,ebx
  mov y,eax; y:=1/sqrt(x)
```

Точность плохая, около 0,2%, но для многих задач этого достаточно. Сам алгоритм долгое время был секретом крупных компьютерных фирм, производящих электронные игры, он использовался в для быстрой отрисовки трёхмерной графики. Алгоритм обнаружили только в начале 2000-х годов при дизассемблировании игры Quake III: Arena. Сейчас этот алгоритм потерял свою актуальность, так как начиная с 2000 года в процессорах Intel появилась специальная векторная команда RSQRTSS, которая вычисляет эту функцию с точностью 0,04%.