Глава 5. Архитектура компьютеров x86 семейства Intel

Архитектура x86 – это победа маркетинга над здравым смыслом.

... нет процессора, кроме x86 и фон Нейман – пророк его.

http://lurkmore.to

5.1. Понятие семейства ЭВМ

…вычислительная машина – новый могущественный инструмент. Однако немногие имеют представление об источнике этого могущества.

Джозефф Вейценбаум

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*,[1] 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).
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.[2] В этой книге для упрощения изложения будут рассматриваться, в основном 32-битные, модели персональных компьютеров x86 семейства ЭВМ компании Intel.

---

[1] "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).

[2] 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 семейст-

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. 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.

В таблице 5.1 представлены некоторые модели семейства фирмы Intel.

Таблица 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	0.060-0.233	3.1	32	64	4 Гб	0.8-0.35
P6 (Pentium Pro)	1995-97	0.150-0.200	5.5	32	64	64 Гб	0.6-0.35
Pentium II	1997	0.233-0.450	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+

во 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.

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

Любая программа стремится занять всю доступную память.

14-й закон программирования

Рассматриваемый компьютер имеет архитектуру с адресуемыми регистрами, поэтому адресуемая память состоит из основной и регистровой памяти. Основная *адресуемая* память имеет объём 2^{32} ячеек длиной по 8 бит, это 4 гигабайта (Гб),[1] при этом каждая команда или данное располагаются в одной или нескольких последовательных (с возрастающими адресами) ячейках этой памяти. Устройство регистровой памяти будет рассмотрена немного позже.

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*, представленную в формализованном виде и пригодную для хранения, передачи и автоматической обработки. Далее рассматриваются большинство форматов данных, для которых в языке 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**

+ 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 вещественные типы `Single`, `Double` и `Extended`. Стоит отметить следующий важный факт. Заметим, что целые числа в различных ЭВМ по чисто историческим причинам иногда имеют разное внутреннее представление. В то же время на момент *массового* выпуска ЭВМ новых поколений, для работы с вещественными числами, уже существовал международный стандарт (ANSI/IEEE 754-1985),[2] на внутреннее представление этих чисел и операции над ними (Standard for Binary Floating-Point Arithmetic). Почти все современные машины придерживаются этого стандарта.

Большинство операций над вещественными числами выполняются 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)`.[3] Для работы с вещественными числами предназначены и *векторные* 256-разрядные регистры `YMM0-YMM15`, причём младшие 128-разряд-

---

[1] Физической памяти на компьютере может быть и больше, например, 8 или 16 Гб, т.е. длина адреса при обращении к физической памяти будет не 32, а 36 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 независимых программ, занимая всю физическую память.

[2] Стандарт разработан под руководством американской ассоциации Института инженеров по электротехнике и электронике IEEE (Institute of Electrical and Electronics Engineers). Любопытно, что этот стандарт разработал один человек, профессор математики Вильям Каган (William Kahan), который работал в Калифорнийском университете в Беркли. Отметим, что в 2008 году вышел обновлённый стандарт IEEE754-2008, в него, в частности, добавлены сверхдлинные 16-байтные вещественные числа `real16`.

[3] Вообще говоря, на этих регистрах можно выполнять и операции с целыми числами, так как происходит автоматическое преобразование целых чисел в вещественные при чтении на эти регистры, 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.

ные части этих регистров имеют имена XMM0-XMM15 и могут использоваться самостоятельно. Каждый такой регистр 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, восемь чисел типа Longint, 16 чисел типа integer или 32 числа типа Shortint). Операции (сложение, вычитание, умножения и т.д.) выполняются сразу над всеми элементами вектора параллельно. Кроме того, в новых процессорах фирмы Intel появились уже и 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-XMM7). Скоро, вероятно, появятся и регистры длиной в несколько килобит 🙂.

- **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 – работать и с векторами целых чисел.

Как видим, в этой архитектуре есть многообразие форматов целых чисел, что, как уже говорилось, позволяет писать более компактные программы.

- **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 137000 символов.

Заметим, что как таковой символьный тип данных (в смысле языка Паскаль) в машине 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, если такую команду, однако, поставить в цикл специального вида, то получается удобное средство для работы с такими массивами.

- **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 в 9 главе.

- **Двоично-десятичные целые числа**

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 настоящее время 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) могут обрабатываться и на 80-битных вещественных регистрах. Этот формат данных в этой книге рассматриваться не будет. Отметим, что в 64-битном режиме двоично-десятичные числа уже не поддерживаются, так как появились двоичные 64-битные числа.

- **Вектора вещественных и целых чисел**

Для работы с целыми и вещественными числами предназначены и *векторные* 256-разрядные регистры YMM0-YMM15, причём младшие 128-разрядные части этих регистров имеют имена XMM0-XMM15 и могут использоваться самостоятельно. Каждый такой регистр 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, восемь чисел типа Longint, 16 чисел типа integer или 32 однобайтных числа типа Shortint). Операции (сложение, вычитание, умножения и т.д.) выполняются сразу над всеми элементами вектора параллельно. Кроме того, в новых процессорах фирмы Intel появились уже и 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-XMM7). Скоро, вероятно, появятся и регистры длиной в несколько килобит 🙂.

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 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 вот теперь каждое представимое в этом формате вещественное число A (кроме вещественного нуля 0.0) может быть записано в виде произведения $A=\pm 1.M*2^{E-127}$. Таким образом, машинный порядок E является смещённым на 127 двоичным порядком числа (поэтому машинный порядок называется еще *смещённым* порядком). Такое представление вещественного числа называется *нормализованным*: его первый сомножитель удовлетворяет неравенству:

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

Нормализация необходима для однозначного представления ненулевого вещественного числа в виде двух сомножителей. Нулевое же число представляется нулями во всех позициях, за исключением, быть может, первой позиции знака числа. Сам процессор при вычислении всегда получает +0.0, при этом, к сожалению, числа −0.0 и +0.0 при сравнении процессором считаются *не равными* 🙁).

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 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):

$$1100\ 0001\ 0101\ 0100\ 0000\ 0000\ 0000\ 0000_2 = C1500000_{16}$$

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

$$C1500000_{16} = 0C1500000h$$

---

[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 в целой части описал немецкий инженер-конструктор ЭВМ К. Цузе. В стандарте ANSI/IEEE единичный бит перед точкой хранится в явном виде только в так называемом расширенном (extended) представлении вещественного числа длиной 80 бит.

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 $2^{-126}..2^{127} \approx 10^{-38}..10^{38}$.

Как и для целых чисел, машинное представление которых будет рассмотрено чуть позже, число представимых вещественных чисел *конечно*. Действительно, легко понять, что 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 положительного числа найдётся соответствующее ему отрицательное число (и наоборот).

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, часто $(a+b)+c \neq a+(b+c)$ и $(a+b)*c \neq a*c+b*c$.[1] 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 ненулевые 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.

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 $M \neq 0$ 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.[2]

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

$\pm A/0 = \pm\infty$; $A \pm \infty = \pm\infty$; $A * \pm\infty = \pm\infty$; $A/\pm\infty = +0$; $\infty + \infty = \infty$;

$0*(\pm\infty) = -\infty+\infty = \pm\infty/\pm\infty = 0/0 = NaN$;

$A \pm NaN = NaN$; $NaN \pm NaN = NaN$; $NaN \pm \infty = NaN$ и т.д.

$1.0^{NaN} = 1.0$; $NaN^{0.0} = 1.0$ (хотя таких *машинных* операций и нет)

Отметим, что при этом правильно учитывается знак числа, например, $(-8.0)*(-\infty) = +\infty$. Операции сравнения не работают, если хотя бы один из операндов NaN, говорят, что такие операнды *не*

---

[1] Для математиков: множество *машинных* вещественных чисел не является ни полем, ни даже кольцом! Действительно, нет "настоящего" нуля (т.к. $0.0*NaN \neq 0.0$), для NaN нет обратного элемента, не верны законы ассоциативности по сложению и дистрибутивности по умножению. Выполняются только коммутативные законы по сложению и умножению (само умножение определяется через сложение). Отметим, однако, что *целые* машинные числа образуют кольцо вычетов по модулю 2^N, где N – разрядность числа.

[2] В зависимости от первого (знакового) бита различают два вида таких "не чисел": тихое (quiet) NaN и громкое (signaling) NaN, аварийная ситуация (исключение) возникает только при использовании в АЛУ громкого NaN. Сам процессор формирует только тихое NaN, громкое NaN присваивается переменной самим программистом (обычно при инициализации таких переменных), для возбуждения исключения, если такая переменная не получит "нормального" значения до её использования.

сравнимы между собой (например, NaN ≠ NaN), за фиксацию этого события отвечает особый фраг в регистре состояния сопроцессора.

Для любознательных читателей заметим, что существует и так называемое *нетрадиционное* 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 обычном, привычном нам математическом анализе, а существуют в виде "настоящих" вещественных чисел.[1]

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, следуя стандарту ANSI/IEEE 754-1985, процессор пытается представить этот результат уже как *ненормализованное* число, т.е. в виде $0.M*2^{-127}$, и только в том случае, если результат и для такого представления слишком мал по модулю, выдаёт в качестве ответа машинный ноль.[2] Использование для очень маленьких по модулю чисел ненормализованного 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 ненормализованных чисел, то наименьшее машинное вещественное число будет равно

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

Такое использование ненормализованных 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 ненормализованных чисел имеет и существенный недостаток, оно влечёт за собой уменьшение *точности* представления таких чисел, т.е. в этих числах меньшее число *значащих* цифр.[3]

Из-за наличия таких "экзотических" вещественных величин, как NaN, ±∞ и ненормализованных 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) ненормализованное число меньше любого (положительного) нормализованного числа. В арифметическом процессоре фирмы Intel при сравнении вещественных чисел формируется двухбитовый признак результата (аналог регистра ω учебной ЭВМ). Этот признак занимает биты C3 и C0 в регистре состояния сопроцессора и принимает значения: 00 – результат > 0, 01 – результат < 0, 10 – результат = 0 и 11 – числа не сравнимы. В том случае, если числа не сравнимы, то дополнительно C2:=1, иначе C2:=0. По смыслу C3 аналог флага ZF, а C1 – аналог флага CF (для *беззнаковых* целых чисел), а вот C2 соответствует флагу PF (об этом флаге будет рассказано в главе, посвящённой командам переходов).

Отметим одно интересное свойство приведённого представления вещественных чисел. Пусть X – вещественное число длиной 32 бита. Обозначим через $X_{доп}$ целое двоичное число, записанное этими

---

[1] Такие числа называются *гипервещественными*. С изложением нетрадиционного математического анализа можно, например, ознакомиться по книгам [11,12].

[2] Во многих ЭВМ существует возможность задать такие режимы работы процессора с вещественными числами, при которых выработка в качестве результата операции таких особых значений, как ненормализованное число или ±∞ будет вызывать аварийную ситуацию (сигнал прерывания). Эти аварийные ситуации носят названия "потеря значимости" (underflow) и "вещественное переполнение" (flowing overflow) соответственно. Можно управлять и другими особыми случаями в операциях с вещественными числами: "денормализация" (denormalised) и "потеря точности" (precision).

[3] Для продвинутых читателей. Работа с ненормализованными 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. А вот, например, графические процессора NVIDIA обычно работают с ненормализованными числами лишь немного медленнее, чем с обычными.

Стоит также отметить, что сейчас набирает популярность новый формат представления вещественных чисел под названием posit. Этот формат обеспечивает бóльшую точность и более эффективную аппаратную реализацию.

же 32 битами и трактуемое как число в дополнительном коде (что это такое изучается в следующем разделе). Тогда для любых вещественных X и Y, таких, что X<Y, будет $X_{доп}<Y_{доп}$. Это, например, позволяет сортировать массив вещественных чисел, сравнивая их как целые, что легче для компьютера.

Как уже упоминалось, с вещественными числами могут работать и векторные регистры, небольшой пример использования векторных регистров приведён в сноске к 14 главе книги. В заключение рассмотрения машинного представления вещественных чисел отметим, что при изучении архитектуры ЭВМ, в основном, из-за недостатка времени операции над вещественными числами изучаться не будут.

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 трактоваться самим программистом 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`.

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 числа `00001101` равен `11110010`.

Для представления *отрицательных* знаковых чисел используется так называемый *дополнительный* (two's complement) код, 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 помещающаяся 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 |-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 трак-

товке 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 чисел.

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 привычной для нас десятичной системе счисления.

- **Пример 1.**

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

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*. Каждый флаг занимает один бит в специальном 32-битном *регистре флагов* с именем EFLAGS. 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*). Флаг CF называется *флагом переноса*, а OF – *флагом переполнения*.[1]

- **Пример 2.**

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

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	383	+127

В данном случае результат будет ошибочен как при беззнаковой, так и при знаковой трактовке складываемых чисел, поэтому формируется содержимое флагов: CF=OF=1. Можно придумать при-

---

[1] При сложении двоичных чисел "в столбик" возможен перенос "1" в следующий разряд из предыдущего. Флаг знакового переполнения OF формируется процессором по следующему 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.

мер, когда результат сложения будет правильный как для знаковых, так и для беззнаковых чисел (сделайте это самостоятельно), после такого сложения оба флага будут опущены.[1]

Кроме формирования флагов CF и OF команда сложения целых чисел меняет и значения некоторых других флагов в регистре флагов EFLAGS. При программировании важен флаг 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 и флаг 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.

Представление отрицательных чисел в дополнительном коде часто неудобно для программистов, однако, позволяет существенно упростить арифметико-логическое устройство. Другими словами, удобство программирования было принесено в жертву простоте реализации процессора.

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. 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.

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

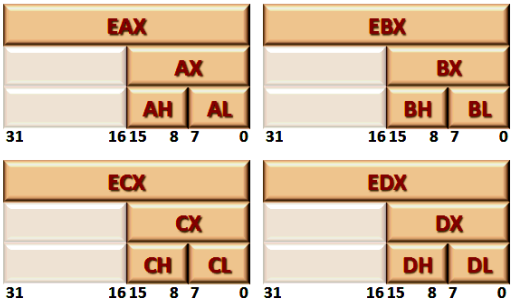
В силу того, что в ЭВМ все регистры имеют безликие двоичные (и часто весьма "хитро" закодированные) номера, программисты используют в программах на Ассемблере мнемонические названия регистров.[2] Восемь регистров общего назначения (Basic Program Registers) показаны ниже, каждый из них может участвовать в операциях сложения и вычитания или просто хранить данные, а некоторые – ещё использоваться в операциях умножения и деления. Длинные 32-битные регистры 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,

---

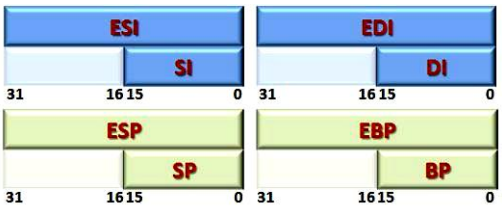
[1] Для продвинутых читателей. Наш процессор для знаковых и беззнаковых целых чисел, кроме обычных операций по модулю (циклических), умеет выполнять и арифметические операции с так называемым *насыщением* (with saturation). Эти операции выполняются на специальных векторных регистрах. При выходе значения такой операции за верхнюю или нижнюю допустимые границы своего типа, в качестве результата берётся 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. Такая "хитрая" арифметика широко применяются при обработке мультимедийных данных (изображения и звука).

[2] 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.

BH, BL, CH, CL, DH, DL.[1] 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 регистра общего назначения с именами ESI, EDI, ESP и EBP, они 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. 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. [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 RA). Этот регистр при выполнении текущей команды содержит адрес *следующей* исполняемой команды.[2]

И, наконец, как уже упоминалось, архитектурой изучаемой ЭВМ предусмотрен регистр флагов с именем EFLAGS, он содержит 32 одноразрядных флага.[3]

+ Конечно, в процессоре есть и *управляющие* 32-битные регистры CR0-CR4 (Control Registers), *системные* регистры TR, GDTR, LDTR, IDTR, *отладочные* регистры DR0-DR7 и другие, для работы с ними предусмотрены специальные команды.

---

[1] 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 несколько 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,'*'
```

[2] При выполнении команд переходов они записываются в этот регистр новый адрес, который чаще всего не совпадает с адресом следующей по порядку команды. Команды перехода будут описаны в следующей главе.

[3] Все 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 практически всех задач на языке Ассемблера.

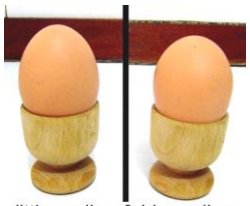
В старших моделях рассматриваемого семейства максимальная длина регистров увеличилась. Так, в 64-разрядных машинах появился 64-битный регистр RAX, младшие 32 бита которого являются регистром EAX. Такие же изменения произошли и с остальными регистрами 32-битной модели.

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

AH		AL	
1	2	3	4

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. В современной архитектуре за одно обращение из памяти получают 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 программистов.


little endian & big endian

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 операнде (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).[1]

5.7. Сегментация памяти

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

Память нашей ЭВМ имеет уже знакомую Вам сегментную организацию. Сначала необходимо заметить, что изучаемая ЭВМ может работать в нескольких принципиально различающихся друг от друга *режимах*. Далее будет рассмотрен только основной, так называемый *защищённый* режим работы (Protected Mode).

В любой момент времени в памяти 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

---

[1] Термины little endian (острый конец куриного яйца) и big endian (тупой конец) впервые появились в компьютерной литературе у автора первых сетевых протоколов Дэнни Коэна, который взял их из романа Джонатана Свифта «Путешествие Гулливера». В романе между жителями двух государств (Лиллипутией и Блефуску) шла яростная дискуссия, с какого конца (острого или тупого) нужно разбивать вареное яйцо за завтраком. В программировании эти термины являются намёком на дискуссию, следует ли представлять числа в памяти ЭВМ в перевёрнутом виде, или нет. Заметим, что у самих лилипутов дело дошло до многолетней ожесточённой войны, и "остроконечники" выиграли.

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.

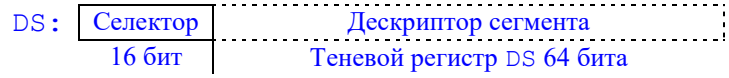
называемый *селектор*[1], чаще всего это селектор *дескриптора* (описателя) сегмента (Segment Descriptor). Дескрипторы сегментов хранятся в специальных таблицах, каждый такой дескриптор описывает конкретный сегмент и содержит:

- 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 *индикатора 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:



Выполняющейся программе предоставляется адресное пространство для размещения команд и чисел. Каждая команда, которая производит обращение к памяти по чтению или записи, процессор вычисляет адрес своего операнда (редко двух операндов) команды. Этот адрес называется *исполнительным* (executable – т.е. вычисляемым) адресом $A_{исп}$, правила такого 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

---

[1] 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.).

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 размер физической памяти сейчас ограничен длиной 2^{36} байт.

Мнемонические обозначения сегментных регистров имеют следующий смысл: кодовый сегментный регистр (`CS`), сегментный регистр данных (`DS`), сегментный регистр стека (`SS`) и дополнительный сегментный регистр (`ES`).[1]

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

Теперь рассмотрим структуру машинных команд самых распространённых форматов регистр-регистр (RR) и регистр-память (RX).

- **Формат регистр-регистр.**

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

Команды этого формата занимают в памяти 2 байта. Первое поле команды – код операции – занимает 6 первых бит, что позволяет задавать до 64 различных операций. Далее следуют однобитные поля с именами `d` и `w`, где `d` – так называемый бит *направления*, а `w` – бит *размера аргумента*, 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.

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

---

[1] 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. Эта тема относится к курсу по операционным системам.

<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.[1]

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

Выбор длины регистра для w=1 зависит от режима работы процессора, который задаётся в бите D (Default Size) дескриптора текущего кодового сегмента. 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 позволяет временно (на одну следующую за ним команду) сменить режим работы процессора (в нашем примере с 32-разрядного на 16-разрядный). Понятно, что в 32-разрядном режиме надо избегать использования 16-разрядных регистров.

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 появляется необходимость *преобразования типов* из короткого целого в длинное, и из длинного в сверхдлинное (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 зна-

---

[1] 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) вместо бита d (direction) задаётся бит размера непосредственного операнда s (size). Так как здесь операнды менять местами нельзя (формата IR нет), то 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 регистр 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

[2] Исключение составляют команды **movsx** и **movzx** для знакового и беззнакового расширения чисел, они будут рассмотрены далее при изучении команды **mov**.

чений. 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)

и

cwd (Convert Word to Double),

которые производят *знаковое* расширение соответственно значения регистра AL до AX и AX до значения пары регистров <DX:AX> (так называемой *регистровой пары*), которые в этом случае рассматриваются как один длинный 32-битный регистр. В старших моделях добавлена команда

cwde (Convert Word to Double in EAX),

для знакового расширения регистра AX до регистра EAX, и команда

cdq (Convert Double to Quad word),

для аналогичного расширения регистра EAX до пары регистров <EDX:EAX>. Кроме того, в 64-битном режиме есть команда

cqo (Convert Quad word to Octa word),

для знакового расширения регистра RAX до пары регистров <RDX:RAX> и команда

cdqe (Convert Double to Quad word in RAX),

для знакового расширения регистра EAX до регистра RAX.

Все эти команды преобразования короткого знакового числа в более длинное не меняют флагов.

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.

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

КОП	r1	A2

Второй 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 (т.е. нулевое смещение не задаётся и не занимает место в команде), B1 и I2 – так называемые *регистры-модификаторы*, причем B1 называется *базовым*, I2 – *индексным* регистрами, а Scale является числовым 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* (executable) адресом.

Рассмотрим подробнее каждый их трёх возможных видов второго операнда. При A2 = A исполнительный адрес $A_{исп} = A$, а логический адрес операнда вычисляется процессором по формуле:

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

где SEG обозначает начало в памяти соответствующего сегмента. Как уже говорилось, при использовании плоской модели памяти SEG=0, поэтому всюду далее логический адрес совпадает с исполнительным:

$$A_{лог} \equiv A_{исп}$$

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 вычисляется так:

$$A_{исп} := (A + B1) \bmod 2^{32},$$

где вместо B1 подставляется содержимое одного из указанных выше регистров-модификаторов.

Запись A2 = A[B1][Scale*I2] обозначает использование в команде *двух* регистров-модификаторов, один из которых называется базовым (B1), а второй – индексным (I2).[1] В качестве индексного регистра можно задать любой 32-битный регистр общего назначения, кроме ESP. Вычисление исполнительного адреса при этом производится по формулам:

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

или

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

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, 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	a8 или a32

где mod – двухбитовое поле, называемой полем модификатора (длины смещения), mem – трёхбитовое поле способа адресации памяти. Будем обозначать через a8 и a32 адреса (m8 и m32) или непосредственные операнды (i8 и i32) длиной в 1 или 4 байта, заданные в дополнительных байтах команды.[2] 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 поля mem=100, обозначенное как SIB (Scale Index Base), 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 вычисление исполнительного адреса второго операнда производится по формуле

---

[1] В Ассемблере MASM допускается эквивалентная запись выражения A[B1][I2] в виде A[I2][B1], A[B1+I2], A[I2+B1] и даже, к сожалению, в виде [A+B1+I2] или [B1+I2+A]. В то же время, похожая на принятую в Паскале запись A[B1,I2] запрещена, вероятно потому, что имеет в математике другой смысл (обращение к элементу *матрицы*).

[2] Вместо 4-байтных полей i32 и m32 можно использовать двухбайтные поля i16 и m16, если задать 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 🙁.

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

При этом, как уже говорилось, I2 может быть любым из восьми 32-битных регистров общего назначения, кроме ESP, а B1 – любым 32-битным регистром общего назначения. Заметьте, что, так как I2 не может быть ESP=100, то этот случай трактуется как *отсутствие* индексного регистра. Одна из "хитрых" комбинаций полей (mod=00 и mem=SIB, а в SIB B1=EBP) трактуется как *отсутствие* в команде базового регистра, и принудительно mod:=10, 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]
mov eax,A[edi+4*edi]; будет 5*edi
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

```
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.[iv] [см. сноску в конце главы]

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

Многие вещи нам непонятны не потому, что наши понятия слабы; но потому, что сии вещи не входят в круг наших понятий.

Козьма Прутков

Далее будет изучен синтаксис машинных команд, записанных так, как это принято в языке Ассемблер MASM 6.14 (можно считать, что Вы уже начали изучать этот язык низкого уровня), и их семантика (способ выполнения команд процессором).

5.9.1. Команды пересылки

Фактически, ассемблерная программа наполовину состоит из команд пересылки данных.

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 режи-

ме 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

Общий вид команды пересылки в нашей двухадресной ЭВМ такой (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`.[1] Надо отметить, что запрещена команда пересылки вида **`mov`** `CS,op2`, так как по своей сути это будет уже команда *передачи управления*, а это совсем другой класс команд нашего компьютера.

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`. Существует также команда пересылки формата `SS` (память-память), но она имеет другое мнемоническое обозначение, является безадресной, и будет изучаться в главе, посвящённой так называемым строковым (или цепочечным) командам.

Иногда при задании второго операнда возникает неоднозначность в его формате, 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 указание на сегментный регистр данных `ds:`,[2] например

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

Отметим также полезную команду *обмена* содержимым двух операндов

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

Таблица допустимых операндов для этой команды:

op1	**op2**

---

[1] Любопытно, что в 64-битном режиме нет команды **`mov`** `m64,i64` (её длина превышает максимальную длину команды в 15 байт) и приходится использовать две команды **`mov`** `r64,i64` и **`mov`** `m64,r64`. Что-то неладно в доме Intel... ☹.

[2] 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. В командах, записывающих данные в память, например **`mov`** `ds:[4],eax`, префикс `cs:` использовать нельзя, т.к. кодовый сегмент закрыт на запись.

r8	r8, m8
r16	r16, m16
r32	r32, m32
m8	r8
m16	r16
m32	r32

В старших моделях появились операнды форматов r64 и m64. 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 вычисленному 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,[ecx+2*ebx+7]
```

Вместо умножения

```
     mov ebx,20
     mul ebx;   eax:=20*eax
```

можно использовать команды

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

Последние две команды как говорят "спариваются" и выполняются процессором параллельно на двух исполнительных устройствах конвейера всего за один такт! [vi] [см. сноску в конце главы] Учтите, однако, что проконтролировать *правильность* полученного командой **lea** результата невозможно, так как никакие флаги не устанавливаются. В то же время вместо команды с простым вторым операндом

lea eax,X; формат r32,m32

рекомендуется использовать команду

mov eax,**offset** X; формат r32,i32

которая на один байт короче.

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 регистр AX, EAX и EDX, как для команд **cbw**, **cwd** и **cdq**). Кроме того, можно расширять как знаковые, так и беззнаковые числа. Команда

movzx op1,op2; op1:=op2

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

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

Команда пересылает второй операнд в младшую часть первого, обнуляя старшую часть первого операнда. Аналогичная команда с такими же допустимыми типами операндов **movsx** (MOV Signed eXtention) производит *знаковое* расширение второго операнда при пересылке на место первого операнда.[1]

---

[1] Как описывалось ранее, для команды **movsx** eax,ax есть и короткая безадресная команды **cwde**.

5.9.2. Арифметические команды

Математика – царица наук, арифметика – царица математики.

Карл Фридрих Гаусс

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). 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 схеме:[1]

`op1 := op1 ± op2`

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**. Подробнее об этом следует прочитать в учебнике по Ассемблеру (например, в [5-7]). Таблица допустимых операндов для этих команд:

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

В старших моделях появились также операнды форматов `r64`, `m64` и `i64`. 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 это 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`, 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. Эти команды работают только с 32-битными операндами, их 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).

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

neg `op1;` взятие обратной величины знакового числа, `op1 := 0-op1`

---

[1] 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;  т.к. это  add Y,-1
```

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

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 и m32. 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. Последнее обстоятельсто препятствует эффективному выполнению команд **inc** и **dec** на конвейере современных процессоров, поэтому вместо них рекомендуется использовать команды **add** и **sub**, хотя они и длиннее на один байт.

+ Кроме того, при переходе в 64-битный режим команды **inc** и **dec** с операндом-регистром перестают быть однобайтными (их коды операций 40h-4Fh отданы под команды-префиксы 64-битных регистров).

Начиная с процессора Intel 486 появилась "хитрая" модификация команды сложения
xadd op1,op2; eXchange and ADD – обмен, затем сложение

При выполнении команды сначала производится операция обмена значениями аргументов **xchg**(op1,op2), а лишь затем собственно сложение. Для этой команды у второго операнда, естественно, недопустимы *непосредственные* операнды (i8, i16, i32 и i64). Заметьте, что эта команда 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 примерно на 1000 тактов. Так же ведут себя описанные позже команды **cmpxchg** и **cmpxchg8b**.

Далее рассмотрим команды умножения и деления целых чисел. Формат этих команд накладывает сильные ограничения на месторасположение их операндов, это *одноадресные* команды, очень похожи на команды учебной одноадресной ЭВМ. Первый операнд всех команд этого класса явно в команде не указывается и находится в фиксированном регистре, заданном *по умолчанию*. Есть следующие команды умножения и деления, в них, как и в уже знакомой Вам учебной одноадресной ЭВМ, явно задаётся только второй операнд (т.е. второй сомножитель или делитель):

mul op2; беззнаковое целочисленное умножение (MULtiply),
imul op2; знаковое целочисленное умножение (sIgn MULtiply),
div op2; беззнаковое целочисленное деление (DIVision),
idiv op2; знаковое целочисленное деление (sIgn DIVision).

Как уже было сказано, в самой команде явно задаётся только второй операнд op2, он может быть форматов r8 и m8 (соответственно, тогда говорят о коротком умножении или делении), r16 и m16 (это длинное умножение и деление) или форматов r32 и m32 (сверхдлинное умножение и 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 ☺). 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 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 производится по формуле:

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*).

---

[1] В старших моделях этого семейства ЭВМ у второго операнда добавляются форматы r64, m64 (для 64-битного режима). Первый сомножитель располагается при этом в регистре RAX, а произведение будет в регистровой паре <RDX:RAX>.

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

<EDX:EAX> := EAX * op2

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 *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").

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

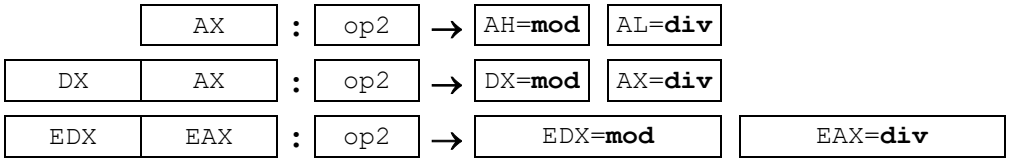
При делении на длинный операнд формата r16 и m16 вычисление производится по формулам:

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

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 <DX:AX> обозначает 32-разрядное целое число, расположенное сразу в двух регистрах DX и AX, а op2, как уже говорилось, может иметь формат r16 или m16. Заметьте, что все команды деления *одновременно* получают два результата (целое частное и остаток).[1] Именно поэтому под результат этой операции отводится в *два раза больше места*, чем под делитель. Ниже показана схема выполнения короткого и длинного деления.



Как видно, команды умножения *всегда* дают точный результат, так как под хранение произведения выделяется в два раза больше места, чем под каждый из сомножителей. В то же время команды деления могут вызывать аварийную ситуацию, 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, приводит к аварийному прекращению выполнения программы. Разумеется, похожую аварийную ситуацию вызывает и деление на ноль. 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 флаги). Можно сказать, что деление "портит" определённые флаги (в частности, портятся "полезные" флаги CF, OF, ZF и SF).

---

[1] В старших моделях у делителя добавляются форматы r64, m64 (для 64-битного режима). 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 ответом является кортеж (вектор) из двух целых чисел.

Стоит отметить, что команды целочисленного деления даже на современных процессорах выполняются примерно в 6 раз медленнее, чем команды умножения (знаковое деление примерно вдвое медленнее, чем беззнаковое). Исходя из этого оптимизирующие компиляторы заменяют команду деления *на константу* "хитрыми" командами умножения и сдвига.[vii] [см. сноску в конце главы]

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, i16
r32	r32, m32, i32

Команда даёт неправильный (усечённый) результат, если произведение не помещается в первый операнд, 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

Когда произведение помещается в первый операнд, то 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, поэтому беззнаковые команды умножения этих форматов не нужны.

Следует также учесть, что языки высокого уровня при реализации операции умножения получают результат, по размеру не превышающий величины сомножителей. Например:

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

Оператор Z:=X*Y примерно так компилируется в 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. Чтобы получать всегда правильный результат операции умножения на языках высокого уровня, программистам надо использовать явное преобразование типов Z:=int64(X)*Y, тогда 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 r32₁,r32₂,op2; <r32₁:r32₂>:=eax*op2
```

она снимает ограничение на размещение произведения только в регистровой паре <EDX:EAX>, размещая результат в двух любых 32-битных регистрах. Эта команда, в отличие от **mul**, не меняет флаги, а второй операнд op2 только форматов r32 или m32. Такую команду можно использовать для более эффективной реализации алгоритма умножения длинных чисел.

Существуют и достаточно специфичные команды для работы с целыми числами. Например, команда **rdrand** (ReaDRANDom):

rdrand op1=r32

Когда команда устанавливает CF=1, то в операнде возвращается "настоящее" беззнаковое 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.

Как уже упоминалось, для работы со (знаковыми) целыми числами можно использовать и вещественные регистры.[viii] [см. сноску в конце главы]

Вопросы и упражнения

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 нашей архитектуре 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 "*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. Что такое бит размера операнда w в машинной команде ?
14. Чем адрес байта памяти в команде отличается от его логического адреса ?
15. Что такое регистр-модификатор ?
16. Что такое задание операндов команды по умолчанию ? Какие операнды задаются по умолчанию в командах целочисленного умножения и деления ?
17. Почему, в отличие от команд сложения и вычитания, необходимы различные команды для умножения и деления знаковых и беззнаковых целых чисел ?
18. Какой будет результат выполнения команды **div** edx ⚠ ?
19. 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] 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 $X_{доп} = X$ **mod** 2^N, где операция **mod** определена как дающая неотрицательный результат. Так принято в математике и описано в стандарте Паскаля, но практически все ЭВМ отступают здесь от стандарта, давая знаковый остаток. Например, Вас в школе 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 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] 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 регистра, например, регистра EAX (но не AX или AL !) в RAX обнуляет старшую половину этого регистра. Ещё более запутанными являются правила использования младших частей векторных регистров ZMM → YMM → XMM, здесь эти правила рассматриваться не будут.

Кроме того, в 64-битной архитектуре доступ к старшим 8-битным регистрам AH, BH, CH и DH уже ограничен. Их нельзя использовать в одной команде вместе с младшими 8-битными регистрами AL, BL, CL и DL, 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). 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] 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 выхода адреса за границы сегмента, а также 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 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 размеров 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 чтение и исполнение команд, но закрыты на запись.

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

[iv] Как уже упоминалась, такая сложная адресация приводит к тому, что по сути одна и та же команда имеет разные битовые представления, например, команду Ассемблера **mov** eax,[esi] можно закодировать такими 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.

---

[v] 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 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` всего за 1 такт. 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 префикс **lock**. 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).

[vi] 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, что, конечно, преувеличение.

[vii] Рассмотрим, например, *беззнаковое* деление числа X формате **dd** на число 10 (`X:=X div 10`). Классический вариант выглядит так:

```
mov eax,X
xor edx,edx
mov ebx,10
div ebx
mov X,eax
```

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

```
mov eax,66666667h; = 1717986919₁₀ Что за число !?
mul X;             <edx:eax>:=X*66666667h
shr edx,2
; edx=X*(66666667h div 2^(32+2)) =
;     X*(0.0001 1001 1001 1001 1001 1001 1001 1001 1100b) =
;     X*(0.19999999Ch) ≈
;     X*0.1000000000349₁₀ ≈ X div 10 !
mov X,edx; X:=X div 10
```

Первый вариант выполняется примерно за 30-40 тактов процессора, а второй – всего за 10 (!). Как можно заметить для деления на 10 нужная константа равна 2^{34} **div** 10. 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 `shr edx,2`) располагается как раз между этими регистрами. Аналогичный подход можно применять и для случая, когда требуется выполнять много делений на переменную X, значение которой вводится (тогда надо один раз вычислить величину 2^{34} **div** X, но есть трудности с обеспечением точности).

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

```
mov  eax,66666667h
imul X
shr  edx,2
shl  X,1;   знаковый бит
```

```
    adc  edx,0; корректировка
    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).

[viii] Здесь будет немного рассказано об использовании вещественных регистров для работы с целыми числами. Это использование основано на способности процессора автоматически преобразовывать знаковые целые числа в вещественные числа так называемого расширенного (Extended) 80-битного формата. В расширенном формате под мантиссу вещественного числа отводится 64 бита, что позволяет преобразовывать целые числа длиной до 64 бит в вещественные без потери точности. Обратное преобразование из вещественного числа в целое, разумеется, производится уже не всегда без потери точности. Преобразование из целого числа в вещественное производится при загрузки из памяти целых чисел форматов `m16`, `m32` и `m64` на вещественные регистры, и, наоборот, есть команды, которые число с вещественного регистра преобразуется в целый формат перед его записью в память. Таким образом, в сопроцессоре производится обработка *только* вещественных чисел.

Вещественные регистры в Ассемблере MASM обозначаются `st(0)`, `st(1)`,…,`st(7)`, они образуют кольцевой стек, причём регистр `st(0)`, является вершиной стека, под ним расположен регистр `st(1)`, и т.д., а за регистром `st(7)`, опять следует `st(0)`. С каждым регистром связан двухбитный *тэг*, все такие тэги собраны в 16-битный регистр `TWR` (Tags Word Register). Каждый тэг определяет содержимое своего регистра: тэг=0 – допустимое ненулевое число, тэг=1 – ноль (как `+0.0`, так и `-0.0`), тэг=2 – недопустимое число, не число (`NaN`), тэг=3 – "пустой" (без данных) регистр стека. Будем использовать мнемонику `st(i)` для обозначение любого из этих регистров.

Как Вы уже догадываетесь, позволяется наряду с обычными, использовать и стековые (безадресные) операции над вещественными числами в этих регистрах. Здесь нужно отметить, что этот стек "кольцевой" только для смещения вершины стека `st(0)`, а для операции записи и чтения это "обычный" стек, переполнение и исчерпание которого приводит к ошибке. Таким образом, при записи в стек тэг регистра `st(7)` должен равняться 3 (этот регистр пустой), а при чтении из стека тэг регистра `st(0)` должен отличаться от 3 (регистр не пустой и стек не пуст). При ошибке работы со стеком результат не определён и устанавливается в единицу бит `SF` (Stack Fault) 16-разрядного регистра состояний сопроцессора `SWR` (Status Word Register), при этом бит `C1` этого же регистра будет равен 1 при переполнении и равен 0 при исчерпании стека.

Команды записи в стек и чтения из стека реализованы в двух модификациях: адресном и стековом, коды операций стековых команд оканчиваются на букву **p**, например, команда записи из стека в целую переменную формата `m32`:

```
    fist  m32;<m32>:=Longint(st(0))
    fistp m32; из стека (m32), т.е.
;<m32>:=Longint(st(0));st(0):=st(1);st(1):=st(2);...;st(6):=st(7)
;тэг(st(7)):=3 (регистр теперь пуст)
    fstp  st(0); st(0):=st(0); ИЗСТЕКА (т.е. просто ИЗСТЕКА)
    fxch  st(1); xchg(st(0),st(1))
```

Отметим, что существуют и команды, записывающие в стек часто используемые вещественные константы, а также некоторые тригонометрические операции, например:

```
    fld2e ;Встек(log2(e))
    fldpi ;Встек(π)
    fsin  ;st(0):=sin(st(0))
```

Размер каждого регистра 80 бит и все операции сопроцессор проводит именно над числами в этом формате. Однако при обмене с памятью обычно берутся не более 64 бит, остальные биты используются для промежуточных вычислений с целью снизить ошибки округления.

Далее приведены некоторые команды для (знаковых) целых чисел. Команда записи целого числа в стек

```
    fild op1;     ВСТЕК(Extended(op1))
```

Допустимые значения `op1=m16, m32, m64`. Команда чтения целого числа из стека

```
    fist[p] op1; op1:=st(0); [ИЗСТЕКА]
```

Допустимые значения `op1=m16, m32` и `m64` (только для **fistp**). Вещественное число из `st(0)` преобразуется в знаковое целое значение нужной точности и записывается в `op1`.Арифметические команды

```
    fiadd op1; st(0):=st(0)+Extended(op1)
    fisub op1; st(0):=st(0)-Extended(op1)
    fimul op1; st(0):=st(0)*Extended(op1)
    fidiv op1; st(0):=st(0)/Extended(op1)
```

Допустимые значения `op1=m16, m32`. В качестве примера рассмотрим вычисление `Q:=-9*Z*(X+Y)`:

```
    .data
      X  dq 1; X: int64;
      Y  dd 2; Y: longint;
```

```
    Z  dw 3; Z: integer;
    Q  dd ?; Q: longint;
    T  dw -9;T: integer=-9;
  .code
    fild  X; ВСТЕК(Extended(X))
    fiadd Y; st(0):=st(0)+Extended(Y)=X+Y
    fimul Z; st(0):=st(0)*Extended(Z)=Z*(X+Y)
    fimul T; st(0):=st(0)*Extended(-9)=-9*Z*(X+Y)
    fistp Q; Q:=Longint(st(0)); ИЗСТЕКА
```

Необходимо отметить, что, кроме команд записи и чтения из стека целых чисел, остальные целочисленные операции с памятью избыточны, их можно (и нужно) заменять на стековые операции с вещественными числами, например, перепишем последний пример:

```
  .code
    fild  X; ВСТЕК(Extended(X))
    fild  Y; ВСТЕК(Extended(Y))
    fadd   ; st(1):=st(1)+st(0)=X+Y; ИЗСТЕКА
    fild  Z; ВСТЕК(Extended(Z))
    fmul   ; st(1):=st(1)*st(0)=Z*(X+Y); ИЗСТЕКА
    fild  T; ВСТЕК(Extended(-2))
    fmul  ; st(0):=st(1)*st(0)=-2*Z*(X+Y); ИЗСТЕКА
    fistp Q; Q:=Longint(st(0)); ИЗСТЕКА
```

Здесь хорошо вспомнить безадресную учебную ЭВМ УМ-0.

В заключении стоит сказать, что сейчас почти все вычисления с вещественными числами проводятся не на вещественных, а на векторных регистрах.