МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УПИВЕРСИТЕТ
ГРАЖДАНСКОЙ АВИАЦИИ

А.В. Столяров

АРХИТЕКТУРА ЭВМ И СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

«ЯЗЫК АССЕМБЛЕРА В ОС UNIX»

Часть I

Москва - 2010

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ГРАЖДАНСКОЙ АВИАЦИИ»

Кафедра прикладной математики

А.В. Столяров

АРХИТЕКТУРА ЭВМ И СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

«ЯЗЫК АССЕМБЛЕРА В ОС UNIX»

Тексты лекций

Часть I

Москва-2010

УДК [004.438Ассемблер:004.451.9UNIX](076.6)
ББК 32.973.26-018.1Ассемблер.я73-2+32.973.26-018.2я73-2
С 81

Печатается по решению редакционно-издательского совета
Московского государственного технического университета ГА

Рецензенты: д-р техн. наук, проф. В.Л. Кузнецов;
канд. физ-мат. наук, доц. И.Г. Головин

Столяров А.В.

С 81 Архитектура ЭВМ и системное программное обеспечение. Язык ассемблера в ОС Unix. Часть I. Тексты лекций. – М.: МГТУ ГА, 2010.- с.88, рис.5.

ISBN 978-5-86311-769-0

В пособии представлено содержание первой части лекций по программированию на языке ассемблера NASM, читаемых в рамках курса «Архитектура ЭВМ и системное программное обеспечение».

Целью лекций является формирование понимания устройства вычислительных машин и получения опыта работы на уровне машинных команд.

Тексты лекций соответствуют рабочей программе учебной дисциплины «Архитектура ЭВМ и системное программное обеспечение» по Учебному плану специальности 230401 для студентов II курса дневного обучения.

Рассмотрено и одобрено на заседаниях кафедры 20.04.2010 г. и методического совета 20.04.2010 г.

Издается в авторской редакции.

ББК 32.973.26-018.1Ассемблер.я73-2+32.973.26-018.2я73-2
Доп. св. план 2010 г.
поз.62

СТОЛЯРОВ Андрей Викторович
АРХИТЕКТУРА ЭВМ И СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ.
ЯЗЫК АССЕМБЛЕРА В ОС UNIX
Часть I
Тексты лекций

	Подписано в печать 05.07.10 г.	
Печать офсетная	Формат 60x84/16	4.92 уч.-изд. л.
5.11 усл.печ.л.	Заказ № 1124/ 229	Тираж 150 экз.

Московский государственный технический университет ГА
125993 Москва, Кронштадтский бульвар, д. 20
Редакционно-издательский отдел
125493 Москва, ул. Пулковская, д.6а

ISBN 978-5-86311-769-0

ПУБЛИЧНАЯ ЛИЦЕНЗИЯ

Настоящее учебное пособие Андрея Викторовича Столярова, называемое далее ≪Произведением≫, защищено действующим авторско-правовым законодательством. Все права на Произведение, предусмотренные действующим законодательством, как имущественные, так и неимущественные, принадлежат его автору. Согласно лицензионному договору между автором и МГТУГА от 24 мая 2010 г., МГТУГА предоставлена простая (неисключительная) лицензия на распространение данного файла в его оригинальной форме (без внесения каких-либо изменений) исключительно в целях обеспечения учебного процесса; договор не предоставляет МГТУГА права на коммерческое использование Произведения.

В дополнение к этому, автор и правообладатель предоставляет неограниченному кругу лиц право использования электронной версии Произведения в порядке и на условиях, изложенных ниже, при условии безоговорочного принятия этими лицами всех условий настоящей Лицензии. Любое использование Произведения, не соответствующее условиям данной Лицензии, а равно и использование Произведения лицами, не согласными с условиями Лицензии, возможно только при наличии письменного разрешения автора и правообладателя, а при отсутствии такого разрешения является нарушением действующего законодательства и преследуется в рамках гражданского, административного и уголовного права.

Автор и правообладатель настоящим **разрешает** следующие виды использования данного файла, являющегося электронным представлением Произведения, без уведомления правообладателя и без выплаты авторского вознаграждения:

1. Воспроизведение Произведения (полностью или частично) на бумаге путём распечатки с помощью принтера в одном экземпляре для удовлетворения личных бытовых или учебных потребностей, без права передачи воспроизведённого экземпляра другим лицам;

2. Копирование и распространение данного файла в электронном виде, в том числе путём записи на физические носители и путём передачи по компьютерным сетям, с соблюдением следующих условий: (1) **все воспроизведённые и передаваемые любым лицам экземпляры файла являются точными копиями исходного файла** в формате PDF, при копировании не производится никаких изъятий, сокращений, дополнений, искажений и любых других изменений, включая и изменение формата представления файла; (2) **распространение и передача копий другим лицам производится исключительно *бесплатно*, то есть при передаче не взимается никакое вознаграждение ни в какой форме**, в том числе в форме просмотра рекламы, в форме платы за носитель или за сам акт копирования и передачи, даже если такая плата оказывается значительно меньше фактической стоимости или себестоимости носителя, акта копирования и т. п.

Любые другие способы распространения данного файла при отсутствии письменного разрешения автора запрещены. В частности, **запрещается**: внесение каких-либо изменений в данный файл, создание и распространение искаженных экземпляров, в том числе экземпляров, содержащих какую-либо часть произведения; распространение данного файла в Сети Интернет через вебсайты, оказывающие платные услуги, через сайты коммерческих компаний, а также через сайты, содержащие рекламу любого рода; продажа и обмен физических носителей, содержащих данный файл, даже если вознаграждение значительно меньше себестоимости носителя; включение данного файла в состав каких-либо информационных и иных продуктов; распространение данного файла в составе какой-либо платной услуги или в дополнение к такой услуге. С другой стороны, **разрешается** дарение (бесплатная передача) носителей, содержащих данный файл, запись данного файла на носители, принадлежащие другим лицам, распространение данного файла через бесплатные файлообменные сети и т.п. Ссылки на экземпляр файла, расположенный на официальном сайте автора, разрешены без ограничений.

А. В. Столяров *запрещает* Российскому авторскому обществу и любым другим организациям производить любого рода лицензирование любых его произведений и осуществлять в интересах автора какую бы то ни было иную связанную с авторскими правами деятельность без его письменного разрешения.

Предисловие для преподавателей

В современной практике индустриального программирования языки ассемблера применяются сравнительно редко; для разработки низкоуровневых программ практически в любых ситуациях подходит язык Си, позволяющий достигать тех же целей многократно меньшими затратами труда, притом с аналогичной, а во многих случаях и более высокой эффективностью получаемого исполняемого кода (последнее достигается за счёт применения оптимизаторов). С помощью языков ассемблера сейчас реализуются разве что весьма специфические фрагменты ядер операционных систем и системных библиотек.

Тем не менее, изучение программирования на языке ассемблера является обязательным для студентов всех специальностей, связанных с программированием. Это легко объяснить: программист, не имеющий опыта работы на уровне команд процессора, попросту не ведает, что *на самом деле* творит. Вставляя в программу на языке высокого уровня те или иные операции, такой программист часто не догадывается, сколь сложную и ресурсоёмкую задачу он ставит перед процессором. На выходе мы имеем огромные программы, поражающие своей низкой эффективностью — например, приложения для автоматизации офисного документооборота, которым оказывается «тесно» в четырёх гигабайтах оперативной памяти и для которых оказывается «слишком медленным» процессор, на много порядков превосходящий по быстродействию суперкомпьютеры восьмидесятых годов.

Кроме того, иногда ссылки на реализацию на уровне машинных команд помогают объяснить студентам средства языков высокого уровня и библиотек. Так, приведя в качестве примера соответствующую ассемблерную вставку, можно наглядно показать различие между системным вызовом и его обёрткой в виде библиотечной функции. Хорошо помогают низкоуровневые иллюстрации также и для объяснения ситуаций состязания при работе с разделяемыми переменными, ими можно воспользоваться для рассказа о функциях `setjmp` и `longjmp` и для описания абстракций более высокого уровня, таких как виртуальные функции в объектно-ориентированном программировании или обработка исключений в языках, подобных Си++ или Аде. Студенту, имеющему представление о механизме стековых фреймов, оказывается гораздо проще объяснить совсем уж, казалось бы, далёкую от фоннеймановского вычислителя материю — оптимизацию остаточной рекурсии в Лиспе и других функциональных языках.

Таким образом, обучение программированию на языке ассемблера

имеет своей целью не создание технических навыков собственно разработки с использованием ассемблеров, а скорее выработку понимания того, что же на самом деле представляет собой компьютер и как с его помощью следует решать задачи.

С этой точки зрения не играет существенной роли выбор конкретной платформы, среды и собственно ассемблера: общие принципы работы центрального процессора различаются мало. С другой стороны, понятно, что использовать следует настоящий компьютер, а не эмулятор, даже если это эмулятор реально существующего компьютера, поскольку программирование эмулятора оставляет у студентов ощущение «игрушечности» используемой среды, во многих случаях превращающееся в уверенность, что «с настоящим компьютером ничего бы не вышло».

Большинство существующих учебных пособий по программированию на языке ассемблера ориентировано на ранние процессоры серии 8086, так называемый «реальный» 16-битный режим работы, операционную среду MS DOS и один из хорошо известных с тех времён ассемблеров `tasm` или `masm`. Причины такого выбора хорошо понятны. С одной стороны, с появлением компьютеров линии IBM PC составителям и преподавателям соответствующих дисциплин в ВУЗах волей-неволей пришлось перейти именно на эту платформу, поскольку другие оказались недоступны; компьютеры, основанные на архитектуре 80x86, до сих пор остаются наиболее доступными для использования в компьютерных классах, практически исключая другие аппаратные платформы из рассмотрения. С другой стороны, с развитием линейки 80x86 возможность запуска программ в режиме эмуляции DOS сохранялась, что позволило сэкономить силы, не изучая архитектуру более новых процессоров линейки и не адаптируя под них существующие учебные курсы.

Между тем, в современных реалиях такой выбор платформы для изучения уже невозможно считать удачным. В самом деле, MS DOS как среда выполнения программ безнадёжно устарела ещё к середине 1990х годов; сам «реальный режим» на современных процессорах поддерживается на уровне микрокода, то есть фактически в режиме программной эмуляции, пусть и внутри процессора. Кроме того, с переходом к 32-битным процессорам (т. е. начиная с процессора 80386) система команд стала существенно более логичной, что подчёркивает бессмысленность траты учебного времени на объяснение странностей архитектуры «реального режима» — странностей, которые заведомо никогда больше не появятся ни в одном процессоре.

Если говорить об использовании 32-битной системы команд (т. н.

платформы i386), то выбор операционной среды оказывается сравнительно невелик, хотя и более разнообразен, нежели во времена MS DOS: это либо операционные системы линии MS Windows, либо представители семейства Unix. И здесь необходимо отметить, что при обучении основам программирования (причём это относится не только к программированию на языке ассемблера) крайне желательно наличие культуры консольных приложений. Начинать обучение программированию с рисования окошек категорически неприемлемо, написание же консольных программ для операционной среды, в которой соответствующая культура отсутствует, создаёт уже знакомое ощущение «игрушечности» происходящего и, к сожалению, существенно расхолаживает студентов.

Кроме того, простой и прозрачный набор системных вызовов ОС Unix, логичные правила взаимодействия операционной системы с пользовательским процессом, использование в процессах «плоской» (flat) модели адресации памяти делают именно операционные системы семейства Unix, в особенности свободно распространяемые примеры таковых (такие, как GNU/Linux и FreeBSD) заведомо более подходящими для ознакомления студентов со спецификой программирования на языке ассемблера.

Отдельно необходимо пояснить выбор конкретного ассемблера. Как известно, для работы с процессорами семейства x86 используются два основных подхода к синтаксису языка ассемблера — это синтаксис AT&T и синтаксис Intel. Одна и та же команда процессора представляется в этих синтаксических системах совершенно по-разному: например, команда, в синтаксисе Intel выглядящая как

```
mov eax, [a+edx]
```

в синтаксисе AT&T будет записываться следующим образом:

```
movl a(%edx), %eax
```

В среде ОС Unix традиционно более популярен именно синтаксис AT&T, но в применении к поставленной учебной задаче это создаёт некоторые проблемы. Учебные пособия, ориентированные на программирование на языке ассемблера в синтаксисе Intel, всё-таки существуют, тогда как синтаксис AT&T описывается исключительно в специальной (справочной) технической литературе, не имеющей целью обучение. Кроме того, необходимо учитывать и многолетнее господство среды MS DOS в качестве платформы для аналогичных учебных курсов; всё это позволяет назвать синтаксис Intel существенно более привычным

для преподавателей (да и для некотрых студентов, как ни странно, тоже) и лучше поддерживаемым.

В среде ОС Unix доступно два основных ассемблера, поддерживающих синтаксис Intel: это NASM (≪Netwide Assembler≫), разработанный Саймоном Тетхемом и Джулианом Холлом, и FASM (≪Flat Assembler≫), созданный Томашем Гриштаром.

Сделать однозначный выбор между этими двумя ассемблерами оказывается достаточно сложно. В настоящем пособии рассматривается язык ассемблера NASM, в том числе и специфические для него макросредства; такой выбор не обусловлен никакими серьёзными причинами и попросту случаен.

Предисловие для студентов

Прежде чем приступать к изучению очередной дисциплины, желательно понять, зачем (с какой целью) эта дисциплина вообще изучается. В особенности это касается технических предметов, к которым, безусловно, относится и курс ≪Архитектура ЭВМ≫, в рамках которого обычно изучается программирование на языке ассемблера.

Учебное пособие, которое вы держите в руках, ориентировано на программирование на языке ассемблера NASM в среде ОС Unix. Между тем, подавляющее большинство профессиональных программистов, услышав о таком, лишь усмехнётся и задаст риторический вопрос: ≪да кто же пишет под Unix на ассемблере? На дворе ведь XXI век!≫

Самое интересное, что при этом они будут совершенно правы. Особенно очевидной становится их правота, если вспомнить, что именно ОС Unix — первая в мире операционная система, которая была написана на языке программирования высокого уровня, специально для этого придуманном (на языке Си). До появления ОС Unix считалось, что операционные системы можно писать только на языке ассемблера.

Более того, в современном мире программирование на языке ассемблера оказалось вытеснено даже из такой традиционно ≪ассемблерной≫ области, как программирование микроконтроллеров — маленьких однокристалльных ЭВМ, предназначенных для встраивания во всевозможную технику, от стиральных машин и сотовых телефонов до самолётов и турбин на электростанциях. В большинстве случаев прошивки микроконтроллеров сейчас пишут тоже на Си, и лишь небольшие вставки выполняют на языке ассемблера.

Конечно, совсем обойтись без фрагментов на языке ассемблера пока не получается. Отдельные ассемблерные модули, а равно и ассемблерные вставки в текст на других языках присутствуют и в ядрах операционных систем, и в системных библиотеках того же языка Си (и других языков высокого уровня); в особых случаях программисты микроконтроллеров тоже вынуждены отказываться от Си и писать «на ассемблере», чтобы, например, сэкономить дефицитную память[1]. Однако такие случаи редки, и мало кому из вас, изучающих ныне программирование на языке ассемблера, придётся хотя бы один раз за всю жизнь прибегнуть к языку ассемблера на практике.

Так зачем же тратить время на изучение ассемблера? Ведь всё равно это никогда не пригодится? Так это выглядит лишь на первый взгляд; при более внимательном рассмотрении вопроса умение мыслить в терминах машинных команд не просто «пригодится», оно оказывается жизненно необходимым любому профессиональному программисту, даже если этот программист никогда не пишет на языке ассемблера. На каком бы языке вы ни писали свои программы, *необходимо* хотя бы примерно представлять, что конкретно будет делать процессор, чтобы исполнить вашу высочайшую волю. Если такого представления нет, программист начинает бездумно применять все доступные операции, не ведая, что *на самом деле* творит. Между тем, одно присваивание, записанное, скажем, на языке Си++, может выполниться в одну машинную команду, а может повлечь *миллионы* таких команд[2]. Два таких присваивания записываются в программе совершенно одинаково (знаком равенства), но этот факт никак нам не поможет.

Вообще, профессиональный пользователь компьютеров, будь то программист или системный администратор, может себе позволить что-то *не знать*, но ни в коем случае не может позволить себе *не понимать*, как устроена вычислительная система на всех её уровнях, от электронных логических схем до громоздких прикладных программ. Не понимая чего-то, мы оставляем в своём тылу место для «ощущения магии»: на каком-то глубоком, почти подсознательном уровне нам продолжает казаться, что что-то там не чисто и без парочки чародеев с волшебными палочками не обошлось. Такое ощущение для профессионала недопустимо категорически: напротив, профессионал обязан быть уверен, вплоть до глубоких слоёв подсознания, что то устройство, с которым он

[1] Действительно, некоторые микроконтроллеры имеют всего 256 байт оперативной памяти и 8 Кб псевдопостоянной памяти для хранения кода программы.

[2] Для знающих Си++ поясним: что будет, если применить операцию присваивания к объекту типа `list<string>`, содержащему пару тысяч элементов?

имеет дело, создано такими же людьми, как и он сам, и ничего «волшебного» или «непознаваемого» собой не представляет.

В этом плане совершенно не важно, какую конкретную архитектуру и язык какого конкретного ассемблера изучать. Зная один язык ассемблера, вы сможете начать писать на любом другом, потратив два-три часа (а то и меньше) на изучение справочной информации; но главное тут в том, что, умея мыслить в терминах машинных команд, вы всегда будете знать, что делаете, и всегда сможете понять, что происходит.

В заключение скажем пару слов о причинах выбора конкретной платформы. Машины на основе процессоров семейства i386 мы избрали исключительно из-за их широкого распространения. Что касается среды ОС Unix, то среди всех возможных операционных сред, имеющихся на платформе i386, именно программирование в ОС Unix оказывается *самым простым*, ну а лишние сложности нам ни к чему.

Итак, теперь вы знаете, что ответить скептикам по поводу программирования на языке ассемблера под ОС Unix. Правильным ответом будет фраза «**нам нужно было попрактиковаться в ассемблерном программировании *под какую-нибудь существующую систему*, всё равно какую, а ОС Unix мы выбрали, потому что под ней это делать проще всего**». Отметим, что эта фраза будет нам полезна, даже если ни одного скептически настроенного профессионального программиста мы не встретим: действительно, ведь здесь одной фразой выражена и наша цель, и принципы, по которым мы выбирали средства.

Благодарности и посвящение

Автор считает своим приятным долгом выразить признательность своему коллеге по кафедре Павлу Сутырину, который прочитал один из ранних вариантов рукописи и сделал ряд ценных замечаний.

Излагаемый в пособии учебный материал был впервые апробирован автором в ходе чтения лекционного курса «Архитектура ЭВМ и язык ассемблера» первокурсникам Ташкентского филиала МГУ им. М. В. Ломоносова весной 2007 года. Самым талантливым и ярким из них — Линаре Адыловой, Максиму Болонкину, Юле Бутковой, Алисе Киреевой и другим — автор с величайшим удовольствием посвящает это пособие.

Глава 1. Введение

§ 1.1. Машинный код и ассемблер

Практически все современные цифровые вычислительные машины работают по одному и тому же принципу.

Вычислительное устройство (собственно сам компьютер) состоит из ***центрального процессора***, ***оперативной памяти*** и ***периферийных устройств***. В большинстве случаев все эти компоненты подключаются к ***общей шине*** — устройству из множества параллельных проводов (дорожек на печатной плате), позволяющему компонентам компьютера обмениваться информацией между собой.

Оперативная память состоит из одинаковых ***ячеек памяти***, каждая из которых имеет свой уникальный номер, называемый ***адресом***. Ячейка памяти может хранить (≪помнить≫) целое число из некоторого диапазона (чаще всего — от 0 до 255, что требует восьми двоичных разрядов, но бывают и другие варианты). Если требуется хранить число из бóльшего диапазона, используют несколько идущих подряд ячеек памяти: так, если одна ячейка содержит восемь двоичных разрядов, то четыре ячейки, рассматриваемые как единая область памяти, могут ≪помнить≫ число от -2^{31} до $2^{31}-1$, то есть от -2147483648 до 2147483647. Отметим, что при рассмотрении нескольких соседних ячеек как представления одного целого числа на разных машинах используют два разных подхода к порядку следования байтов. Один подход, называемый *little-endian*[1], предполагает, что первым идёт самый младший

[1] ≪Термины≫ big-endians и little-endians введены Джонатаном Свифтом в книге ≪Путешествия Гулливера≫ для обозначения непримиримых сторонников разбивания яиц соответственно с тупого конца и с острого. На русский язык эти названия обычно переводились как *тупоконечники и остроконечники*. Аргументы в пользу той или иной архитектуры действительно часто напоминают священную войну остроконечников с тупоконечниками.

байт, далее в порядке возрастания, и самый старший байт идёт последним. Второй подход, который называют *big-endian*, прямо противоположен: сначала идёт старший байт, а младший располагается в памяти последним. Процессоры, которые мы будем рассматривать, относятся к категории «little-endian», то есть хранят младший байт первым.

При необходимости содержимое ячейки памяти можно рассматривать и как строчку из отдельных двоичных разрядов (битовую строку), и другими способами (например, достаточно сложный способ интерпретации значений двоичных разрядов используется для хранения дробных чисел, так называемых ***чисел с плавающей точкой***).

Кроме того, содержимое ячейки памяти (или нескольких ячеек, идущих подряд) может быть истолковано как ***машинная инструкция*** — кодовое число, идентифицирующее одну из множества операций, которые может выполнять центральный процессор.

Важно понимать, что сама по себе ячейка памяти «не знает», как именно следует интерпретировать хранящуюся в ней информацию. Рассмотрим это на простейшем примере. Пусть у нас есть четыре идущие подряд ячейки памяти, содержимое которых соответствует шестнадцатеричным числам `41`, `4E`, `4E` и `41` (соответствующие десятичные числа — 65, 79, 79, 65). Информацию, содержащуюся в такой области памяти, можно с совершенно одинаковым успехом истолковать:

- как целое число 1095650881;
- как дробное число (т. н. число с плавающей точкой) 12.894105;
- как текстовую строку, содержащую имя `'ANNA'`;
- и, наконец, как последовательность машинных команд; в частности, на процессорах платформы i386 это будут команды, условно обозначаемые `inc ecx`, `dec esi`, `dec esi`, `inc ecx`. Что делают эти команды, мы узнаем позже.

В процессоре имеется некоторое количество ***регистров*** — схем, напоминающих ячейки памяти. В регистрах хранится информация, непосредственно находящаяся в работе. Процессор обладает способностью копировать данные из оперативной памяти в регистры и обратно, производить над содержимым регистров арифметические и другие операции; в некоторых случаях операции можно производить и непосредственно с содержимым ячеек памяти, не копируя их содержимое в регистры[2].

[2]Наличие или отсутствие такой возможности зависит от конкретного процессора; так, процессоры Pentium могут, минуя регистры, прибавить заданное число к содер-

Программа, предназначенная к выполнению, записывается в оперативную память в виде последовательности машинных инструкций (команд), то есть цифровых кодов, обозначающих те или иные операции. Один из регистров процессора, так называемый ***счётчик команд***, содержит адрес той ячейки памяти, в которой располагается следующая инструкция, предназначенная к выполнению.

Процессор работает, раз за разом выполняя ***цикл обработки команды***. В начале этого цикла из ячеек памяти, на которые указывает[3] счётчик команд, считывается код очередной команды. Сразу после этого счётчик команд меняет своё значение так, чтобы указывать на следующую команду в памяти; например, если только что прочитанная команда занимала три ячейки памяти, то счётчик команд увеличивается на три. Схемы процессора дешифруют код и выполняют действия, предписанные этим кодом: например, это может быть предписание «скопировать число из одного регистра в другой» или «взять содержимое регистра A, прибавить к нему содержимое регистра B, а результат поместить обратно в регистр A», и т. п. Когда действия, предписанные командой, будут исполнены, процессор вновь возвращается к началу цикла обработки команд, так что следующий проход этого цикла выполняет уже следующую команду, и так далее до бесконечности (точнее, пока процессор не выключат).

Некоторые машинные команды могут изменить последовательность выполнения команд, предписав процессору перейти в другое место программы (то есть, попросту говоря, в явном виде изменить текущее значение счётчика команд). Такие команды называются ***командами перехода***. Различают ***условные*** и ***безусловные*** переходы; команда условного перехода сначала проверяет истинность некоторого условия и производит переход, только если условие выполнено, тогда как команда безусловного перехода просто заставляет процессор продолжить выполнение команд с заданного адреса безо всяких проверок. Процессоры обычно поддерживают и переходы с запоминанием точки возврата, которые используются для вызова подпрограмм.

Ясно, что программа, которую выполняет компьютер, должна быть представлена в виде, понятном центральному процессору. Машинные

жимому заданной ячейки памяти и произвести некоторые другие операции, тогда как процессоры SPARC, применяемые в компьютерах фирмы Sun Microsystems, могут только копировать содержимое ячейки памяти в регистр или, наоборот, содержимое регистра в ячейку памяти, но никаких других операций над ячейками памяти выполнять не могут.

[3]Выражение вида «нечто указывает на ячейку памяти» является синонимом выражения «нечто содержит адрес ячейки памяти»

команды обозначаются числами (кодами), которые легко может дешифровать процессор, но которые очень трудно запомнить человеку, тем более что во многих случаях нужное число приходится *вычислять*, подставляя в определённые места кодовые цепочки двоичных битов. Вот, например, два байта, записываемые в шестнадцатеричной системе как `01 D8` (соответствующие десятичные значения — 1, 216) обозначают на процессорах Pentium команду «взять число из регистра `EAX`, прибавить к нему число из регистра `EBX`, результат сложения поместить обратно в регистр `EAX`». Запомнить два числа `01 D8` несложно, но ведь разных команд на процессоре Pentium — несколько сотен, да к тому же здесь сама команда — только первый байт (`01`), а второй (`D8`) нам придётся вычислить в уме, вспомнив (или узнав из справочника), что младшие три бита в этом байте обозначают первый регистр (первое слагаемое, а также и место, куда следует записать результат), следущие три бита обозначают второй регистр, а самые старшие два бита здесь должны быть равны единицам, что означает, что оба операнда являются регистрами. Зная (или, опять же, подсмотрев в справочнике), что номер регистра `EAX` — 0, а номер регистра `EBX` — 3, мы теперь можем записать двоичное представление нашего байта: `11 011 000` (пробелы вставлены для наглядности), что и даёт в десятичной записи 216, а в шестнадцатеричной — искомое `D8`.

Интересно, что если нам потребуется освежить в памяти кусочек нашей программы, написанный два дня назад, то чтобы его прочитать, нам придётся вручную раскладывать байты на составляющие их биты и, сверяясь со справочником, вспоминать, что же какая команда делает. Очевидно, что, если программиста заставить составлять программы вот таким вот способом, ничего полезного он не напишет за всю свою жизнь, тем более что в любой, даже самой небольшой, но практически применимой программе таких команд будет несколько тысяч, ну а самые большие программы состоят из *сотен миллионов* машинных команд.

При работе с языками программирования высокого уровня, такими как Паскаль, Си, Лисп и др., программисту предоставляется возможность написать программу в виде, понятном и удобном для человека, а не для центрального процессора. В этом случае приходится применять ***компилятор*** — программу, принимающую на вход текст программы

на языке программирования высокого уровня и выдающую эквивалентный машинный код[4].

Программирование на языках высокого уровня удобно, но, к сожалению, не всегда применимо. Причины этого могут быть самые разные. Например, язык высокого уровня может не учитывать некоторые особенности конкретного процессора, либо программиста может не устраивать тот конкретный способ, которым компилятор реализует те или иные конструкции исходного языка с помощью машинных кодов. В этих случаях приходится отказаться от языка высокого уровня и составить программу в виде *конкретной последовательности машинных команд*. Однако, как мы уже видели, составлять программу непосредственно в машинных кодах очень и очень сложно. И здесь на помощь приходит программа, называемая ***ассемблером***.

Ассемблер — это программа, принимающая на вход текст, содержащий условные обозначения машинных команд, удобные для человека, и переводящий эти обозначения в последовательность соответствующих кодов машинных команд, понятных процессору. В отличие от самих машинных команд, их условные обозначения, называемые также ***мнемониками***, запомнить сравнительно легко. Так, команда из вышеприведённого примера, код которой, как мы с некоторым трудом выяснили, равен `01 D8`, в условных обозначениях[5] выглядит так:

```
add  eax, ebx
```

Здесь нам уже не надо заучивать числовой код команды и вычислять в уме обозначения операндов, достаточно запомнить, что словом `add` обозначается сложение, причём в таких случаях всегда первым после обозначения команды стоит первое слагаемое (не обязательно регистр, это может быть и ячейка памяти), вторым — второе слагаемое (это может быть и регистр, и ячейка памяти, и просто число), а результат всегда заносится на место первого слагаемого. Язык таких условных обозначений (мнемоник) называется ***языком ассемблера***.

Программирование на языке ассемблера коренным образом отличается от программирования на языках высокого уровня. На языке высокого уровня (на том же Паскале) мы задаём лишь общие указания, а компилятор волен сам выбирать, каким именно способом их испол-

[4] Вообще говоря, компилятор — это программа, переводящая программы с одного языка на другой; перевод на язык машинного кода — это лишь частный случай, хотя и очень важный.

[5] Здесь и далее используются условные обозначения, соответствующие ассемблеру NASM, если не сказано иное.

нить — например, какими регистрами и ячейками памяти воспользоваться для хранения промежуточных результатов, какой применить алгоритм для выполнения какой-нибудь нетривиальной инструкции, и т. д. С целью оптимизации быстродействия компилятор может переставить инструкции местами, заменить одни на другие — лишь бы результат остался неизменным. В отличие от этого, **на языке ассмеблера мы совершенно однозначно и недвусмысленно указываем, из каких машинных команд будет состоять наша программа, и никакой свободы ассемблер (в отличие от компилятора языка высокого уровня) не имеет**.

В отличие от машинных кодов, мнемоники доступны для человека, то есть программист может работать с мнемониками без особого труда, но это не означает, что программировать на языке ассемблера просто. Действие, на описание которого мы бы потратили один оператор языка высокого уровня, может потребовать десятка, если не сотни строк на языке ассемблера, а в некоторых случаях и больше. Дело тут в том, что компилятор языка высокого уровня содержит большой набор готовых «рецептов», как решать часто возникающие небольшие задачи, и предоставляет все эти «рецепты» программисту в виде удобных высокоуровневых конструкций; ассемблер же никаких таких рецептов не содержит, так что в нашем распоряжении оказываются только возможности процессора.

Интересно, что для одного и того же процессора может существовать несколько разных ассемблеров. На первый взгляд это кажется странным, ведь не может же один и тот же процессор работать с разными системами машинных кодов (так называемыми ***системами команд***). В действительности ничего странного здесь нет, достаточно вспомнить, что же такое на самом деле ассемблер. Система команд процессора, разумеется, не может измениться (если только не взять другой процессор). Однако для одних и тех же команд можно придумать разные обозначения; так, уже знакомая команда `add eax,ebx` в обозначениях, принятых в компании AT&T, будет выглядеть как `addl %ebx,%eax` — и мнемоника другая, и регистры не так обозначены, и операнды не в том порядке, хотя получаемый машинный код, разумеется, строго тот же самый — `01 D8`. Кроме того, при программировании на языке ассемблера мы обычно пишем не только мнемоники машинных команд, но и ***директивы***, представляющие собой прямые приказы ассемблеру. Следуя таким указаниям, ассемблер может зарезервировать память, объявить ту или иную метку видимой из других модулей программы, перейти к генерации другой секции программы, вычислить (прямо во время ассем-

блирования) какое-нибудь выражение и даже сам (следуя, разумеется, нашим указаниям) «написать» фрагмент программы на языке ассемблера, который сам же потом и обработает. Набор таких вот директив, поддерживаемых ассемблером, тоже может быть разным, как по возможностям, так и по синтаксису.

Поскольку ассемблер — это не более чем программа, написанная вполне обыкновенными программистами, никто не мешает другим программистам написать свою программу-ассемблер, что часто и происходит. Ассемблер NASM, упоминаемый в названии данного пособия — это один из ассемблеров, существующих для процессоров семейства x86. Существуют и другие ассемблеры; возможно даже, что какой-нибудь из них покажется вам более удобным. На самом деле, не так уж важно, язык какого конкретного ассемблера изучать. Важно понять общий принцип работы на уровне команд процессора, и после этого вы сможете без труда освоить не только другой ассемблер, но и любой другой процессор с совсем другими командами.

§ 1.2. Особенности программирования под управлением мультизадачных операционных систем

Поскольку мы собираемся запускать написанные нами программы под управлением ОС Unix, нелишним будет заранее описать некоторые особенности таких систем с точки зрения выполняемых программ; эти особенности распространяются на все программы, выполняющиеся как под операционными системами семейства Unix, так и под многими другими системами, и никак не зависят от используемого языка программирования, но при работе на языке ассемблера становятся особенно заметны.

Практически все современные операционные системы позволяют запускать программы на одновременное исполнение. Такой режим работы вычислительной системы, называемый ***мультизадачным***, порождает некоторые проблемы, требующие решения со стороны аппаратуры (прежде всего — центрального процессора).

Прежде всего, необходимо защитить выполняемые программы друг от друга и саму операционную систему от пользовательских программ. Если (пусть даже не по злому умыслу, а по ошибке) одна из выполняемых задач изменит что-то в памяти, принадлежащей другой задаче,

скорее всего это приведёт к аварии этой второй задачи, причём найти причину такой аварии окажется принципиально невозможно. Если пользовательская задача (опять-таки по ошибке) внесёт изменения в память операционной системы, это приведёт уже к аварии всей системы, причём, опять-таки, без малейшей возможности разобраться в причинах таковой. Поэтому центральный процессор должен поддерживать механизм ***защиты памяти***: каждой выполняющейся задаче выделяется определённая область памяти, и к ячейкам за пределами такой области задача обращаться не может.

Кроме того, в мультизадачном режиме работы пользовательские задачи, как правило, не допускаются к прямой работе с внешними устройствами[6]. Если бы это правило не выполнялось, задачи постоянно начинали бы конфликтовать за доступ к устройствам, и такие конфликты, разумеется, приводили бы к авариям.

Чтобы ограничить возможности пользовательской задачи, создатели центрального процессора объявили часть имеющихся машинных инструкций ***привилегированными***.

Процессор может работать либо в ***привилегированном режиме***, также называемом ***режимом суперпользователя***, либо в ***ограниченном режиме***, который также называют ***режимом задачи*** или ***пользовательским режимом***[7]. В ограниченном режиме привилегированные команды недоступны. В привилегированном режиме процессор может выполнять все имеющиеся инструкции, как обычные, так и привилегированные. Операционная система выполняется, естественно, в привилегированном режиме, а при передаче управления пользовательской задаче переключает режим в ограниченный. Вернуться в привилегированный режим процессор может только при условии одновременной передачи управления назад операционной системе, так что код пользовательской программы выполняться в привилегированном режиме не может.

К категории привилегированных относятся инструкции, осуществляющие взаимодействие с внешними устройствами; также в эту категорию попадают инструкции, используемые для настройки механизмов защиты памяти и некоторые другие команды, влияющие на работу всей

[6] Из этого правила есть исключения, связанные, например, с отображением графической информации на дисплее, но в этом случае устройство должно быть закреплено за одной пользовательской задачей и строго недоступно для других задач

[7] На самом деле процессор i386 и его потомки имеют не два, а четыре режима, называемые также *кольцами защиты*, но реально операционные системы используют только нулевое кольцо (высший возможный уровень привилегий) и третье кольцо (низший уровень привилегий).

системы в целом. Все такие «глобальные» действия являются прерогативой операционной системы.

Итак, **при работе под управлением мультизадачной операционной системы пользовательская задача может лишь преобразовывать информацию в отведённой ей области оперативной памяти. Всё взаимодействие с внешним миром задача производит через обращения к операционной системе.** Даже просто вывести на экран строку задача самостоятельно не может, ей необходимо попросить об этом операционную систему. Такое обращение пользовательской задачи к операционной системе за теми или иными услугами называется ***системным вызовом***.

Интересно, что *завершение задачи* тоже относится к категории действий, которые может выполнить только операционная система. Таким образом, корректная пользовательская задача обойтись без системных вызовов не может никак, ведь даже просто завершиться она может только с помощью соответствующего системного вызова.

Ещё один важный момент, который необходимо упомянуть перед началом изучения конкретного процессора — это наличие в нашей операционной среде механизма ***виртуальной памяти***. Попробуем понять, что это такое.

Как уже говорилось, оперативная память делится на одинаковые по своей ёмкости *ячейки* (в нашем случае каждая ячейка содержит 1 байт данных), и каждая такая ячейка имеет свой порядковый номер. Именно этот номер использует центральный процессор для работы с ячейками памяти через общую шину, чтобы отличать их одну от другой. Назовём этот номер ***физическим адресом*** ячейки памяти.

Изначально никаких других адресов, кроме физических, у ячеек памяти не было. В машинном коде программ использовались именно физические адреса, которые называли просто «адресами», без уточняющего слова «физический». Однако с развитием мультизадачного режима работы вычислительных систем оказалось, что в силу целого ряда причин использование физических адресов *неудобно*. Например, программа в машинном коде, в которой используются физические адреса ячеек памяти, не сможет работать в другой области памяти — а ведь в мультизадачной ситуации может оказаться, что нужная нам область уже занята другой задачей. Есть и другие причины, которые обычно подробно рассматриваются в учебных курсах, посвященных операционным системам.

В современных процессорах используется два вида адресов. Первый — это уже знакомые нам физические адреса. Сам процессор работает с памятью, используя именно их. А вот в программах, которые на процессоре выполняются, используются совсем другие адреса — *виртуальные*.

Виртуальный адрес — это число из некоторого абстрактного ***виртуального адресного пространства***. На тех процессорах, с которыми мы будем работать, виртуальное адресное пространство состоит из 32-битных целых чисел, то есть в качестве адресов используются числа от 0 до $2^{32} - 1$; адреса обычно записываются в шестнадцатеричной системе, так что адрес может предствлять собой число от `00000000` до `ffffffff`.

Важно понимать, что виртуальный адрес совершенно не обязан соответствовать какой-либо ячейке памяти. Точнее говоря, *некоторые* виртуальные адреса соответствуют физическим ячейкам памяти, другие — не соответствуют, а некоторые адреса и вовсе могут то соответствовать физической памяти, то не соответствовать. Такие соответствия задаются путём соответствующей настройки центрального процессора; за эту настройку отвечает операционная система. Будучи соответствующим образом настроенным, центральный процессор, получив из очередной машинной инструкции виртуальный адрес, *преобразует* его в адрес физический, и тогда уже обращается к оперативной памяти.

Таким образом, мы в программах используем в качестве адресов не физические номера ячеек памяти, а некие абстрактные адреса, которые потом уже сам процессор преобразует в настоящие номера ячеек. Это позволяет, например, каждой программе иметь своё собственное адресное пространство: действительно, никто не мешает операционной системе настроить преобразования адресов так, чтобы один и тот же виртуальный адрес в одной пользовательской задаче отображался на одну физическую ячейку, а в другой задаче — на совсем другую.

Вопросы, связанные с созданием новых операционных систем, мы в нашем пособии рассматривать не будем. Вместо этого мы ограничимся рассмотрением возможностей процессора i386, доступных пользовательской задаче, работающей в ограниченном режиме. Более того, даже эти возможности мы рассмотрим не все; дело в том, что операционные системы семейства Unix используют так называемую ***плоскую модель*** адресации памяти, в которой не используется часть регистров и некоторые виды машинных команд. На изучение этих регистров и команд

мы не будем тратить время, поскольку всё равно не сможем ими воспользоваться.

Позже в нашем курсе мы подробно рассмотрим механизмы взаимодействия с операционной системой, включая и способы организации системного вызова для систем GNU/Linux и FreeBSD; однако пока нам эти механизмы не известны, мы будем пользоваться для осуществления ввода/вывода и для завершения программы готовыми *макросами* — специальными идентификаторами, которые наш ассемблер развернёт в целые последовательности машинных команд и уже в таком виде оттранслирует.

Отметим, что к концу нашего курса мы сами научимся при необходимости создавать такие макросы.

§ 1.3. История платформы i386

В 1971 году корпорация Intel выпустила в свет семейство микросхем, получившее название MCS-4. Одна из этих микросхем, Intel 4004, представляла собой первый в мире законченный центральный процессор на одном кристалле, т. е., иначе говоря, первый в истории ***микропроцессор***. Машинное слово[8] этого процессора составляло четыре бита.

Год спустя Intel выпустила восьмибитный процессор Intel 8008, а в 1974 году — более совершенный Intel 8080. Интересно, что 8080 использовал иные коды операций, но при этом программы, написанные на языке ассемблера для 8008, могли быть без изменений оттранслированы и для 8080. Аналогичную «совместимость по исходному коду» конструкторы Intel поддержали и для появившегося в 1978 году 16-битного процессора Intel 8086. Выпущенный годом позже процессор Intel 8088 представлял собой практически такое же устройство, отличающееся только разрядностью внешней шины (для 8088 она составляла 8 бит, для 8086 — 16 бит). Именно процессор 8088 был использован в компьютере IBM PC, давшем начало многочисленному и невероятно популярному[9] семейству машин, до сих пор называемых ***IBM PC-совместимыми*** или просто ***IBM-совместимыми***.

[8] Напомним, что машинным словом называется количество битов, обрабатываемое процессором в один приём.

[9] Популярность IBM-совместимых машин представляет собой явление весьма неоднозначное; многие другие архитектурные решения, имевшие существенно лучший дизайн, не смогли выжить на рынке, затопленном IBM-совместимыми компьютерами, более дешевыми за счёт их массовости. Так или иначе, в настоящее время ситуация именно такова и никаких тенденций к её изменению не предвидится.

Процессоры 8086 и 8088 не поддерживали защиты памяти и не имели разделения команд на обычные и привилегированные. Таким образом, запустить мультизадачную операционную систему на компьютерах с этими процессорами было невозможно.

То же самое можно было сказать и относительно процессора 80186, выпущенного в 1982 году. В сравнении со своими предшественниками этот процессор работал гораздо быстрее за счёт аппаратной реализации некоторых операций, выполнявшихся в предыдущих процессорах путём исполнения микрокода, и за счёт повышения тактовой частоты. Процессор включал в себя некоторые подсистемы, которые ранее требовалось поддерживать с помощью дополнительных микросхем — такие как контроллер прерываний и контроллер прямого доступа к памяти. Кроме того, система команд процессора была расширена введением дополнительных команд; так, стало возможным с помощью одной команды занести в стек все регистры общего назначения. Адресная шина процессоров 8086, 8088 и 80186 была 20-разрядной, что позволяло адресовать не более 1 Mb оперативной памяти.

В том же 1982 году увидел свет и процессор 80286, ставший последним 16-битным процессором в рассматриваемом ряду. Этот процессор поддерживал так называемый защищённый режим работы (protected mode), в котором реализовывалась сегментная модель виртуальной памяти, подразумевающая, в том числе, и защиту памяти; четыре ***кольца защиты*** позволили запретить пользовательским задачам выполнение действий, влияющих на систему в целом, что необходимо при работе мультизадачной операционной системы. Адресная шина получила четыре дополнительных разряда, увеличив, таким образом, максимальное количество непосредственно доступной памяти до 16 Mb.

Однако по-настоящему мультизадачные операционные системы были реализованы лишь на следующем процессоре в ряду, 32-разрядном Intel 80386, для краткости обозначаемом просто «i386». Этот процессор, массовый выпуск которого начался в 1986 году, резко отличался от своих предшественников, прежде всего, увеличением регистров до 32 бит, существенным расширением системы команд, увеличением адресной шины до 32 разрядов, что позволяло непосредственно адресовать до 4 Gb физической памяти. Добавление поддержки ***страничной организации виртуальной памяти***, наилучшим образом пригодной для реализации мультизадачного режима работы, завершило картину. Именно с появлением i386 так называемые IBM-совместимые компьютеры, наконец, стали полноценными вычислительными системами.

Вместе с тем, i386 полностью сохранил совместимость с предшествующими процессорами своей серии, чем обусловлена достаточно странная на первый взгляд система регистров. Например, универсальные регистры процессоров 8086–80286 назывались `AX`, `BX`, `CX` и `DX` и содержали 16 бит данных каждый; в процессоре i386 и более поздних процессорах линейки имеются регистры, содержащие по 32 бита и называющиеся `EAX`, `EBX`, `ECX` и `EDX` (буква `E` означает слово «extended», т. е. «расширенный»), причём младшие 16 бит каждого из этих регистров сохраняют старые названия (соответственно, `AX`, `BX`, `CX` и `DX`). Большинство инструкций работает по-разному для операндов длиной 8 бит, 16 бит и 32 бита, и т. п.

Интересно, что дальнейшее развитие семейства процессоров x86 вплоть до 2003 года было чисто количественным: увеличивалась скорость, добавлялись новые команды, но принципиальных изменений архитектуры не происходило. В 2003 году компания AMD представила новый процессор, имеющий 64-битные регистры, и к настоящему времени многие операционные системы способны выполняться на таких процессорах, однако наиболее популярной остаётся до сих пор именно 32-битная платформа, родоначальником которой стал процессор i386.

§ 1.4. Знакомимся с инструментом

Прежде чем написать первую самостоятельную программу на языке ассемблера, нам необходимо изучить процессор, с которым мы будем работать (пусть даже не все его возможности, но хотя бы некоторую существенную их часть), а также синтаксис языка ассемблера. К сожалению, здесь возникает определённая проблема: изучать эти две вещи одновременно не получается, но, в то же время, изучать систему команд процессора, не имея никакого представления о синтаксисе языка ассемблера, а равно и изучать синтаксис, не имея представления о системе команд — задача неблагодарная, так что, с чего бы мы ни начали, результат получится несколько странный.

Мы попробуем пойти иным путём. Некоторое представление о системе команд у нас уже есть, пусть даже оно весьма и весьма слабое; попробуем получить аналогичное представление и о синтаксисе языка ассемблера, а затем уже приступим к систематическому изучению того и другого.

Сейчас мы напишем работающую программу на языке ассемблера, оттранслируем её и запустим. Поначалу в тексте программы будет да-

леко не всё понятно; что-то мы объясним прямо сейчас, что-то оставим до более подходящего момента.

Задачу мы для себя выберем очень простую: напечатать[10] пять раз слово «Hello». Как мы уже говорили на стр. 19, для вывода строки на экран, а также для корректного завершения программы нам потребуется обращаться к операционной системе, но мы пока воспользуемся для этого уже готовыми *макросами*, которые описаны в отдельном файле. Ассемблер, сверяясь с этим файлом и с нашими указаниями, преобразует каждое использование такого макроса во фрагмент кода на языке ассемблера, и сам же эти фрагменты затем оттранслирует. Поэтому в нашей программе будет очень мало мнемоник, обозначающих собственно машинные команды; в основном текст программы будет состоять из *директив*.

Итак, пишем текст программы:

```
%include "stud_io.inc"

global  _start

section .text
_start: mov     eax, 0
again:  PRINT   "Hello"
        PUTCHAR 10
        inc     eax
        cmp     eax, 5
        jl      again
        FINISH
```

Попробуем теперь кое-что объяснить. Первая строчка программы содержит директиву `%include`; эта директива предписывает ассемблеру вставить на место самой директивы всё содержимое некоторого файла, в данном случае — файла `stud_io.inc`. Этот файл также написан на языке ассемблера и содержит описания макросов `PRINT`, `PUTCHAR` и `FINISH`, которые мы будем использовать для печати строки, для перехода на следующую строку на экране, а также для завершения программы. Таким образом, увидев директиву `%include`, ассемблер прочитает

[10] Т.е. вывести на экран, или, если говорить строго, *вывести в поток стандартного вывода*; отметим, что процессор сам по себе ничего не знает о выводе на экран, все операции ввода-вывода требуют работы с внешними устройствами и организуются операционной системой, она же предоставляет нашей задаче абстрактные «стандартные потоки ввода-вывода».

файл с описаниями макросов, в результате чего мы сможем их использовать.

Важно отметить, что директива `%include` обязательно должна стоять в тексте программы *раньше*, чем там встретятся имена макросов. Ассемблер просматривает наш текст сверху вниз. Изначально он ничего не знает о макросах и не сможет их обработать, если ему о них не сообщить. Просмотрев файл, содержащий описания макросов, ассемблер запоминает эти описания и продолжает их помнить до окончания трансляции, так что мы можем их использовать в программе — но не раньше, чем о них узнает ассемблер. Именно поэтому мы поставили директиву `%include` в самое начало программы: теперь макросы можно использовать во всём её тексте.

После директивы `%include` мы видим строку со словом `global`; это тоже директива, но к ней мы вернёмся чуть подзнее.

Следующая строка программы содержит директиву `section`. Исполняемый файл в ОС Unix устроен так, что в нём машинные команды хранятся в одном месте, а инициализированные (т. е. такие, которым прямо в программе задаётся начальное значение) данные — в другом, и, наконец, в третьем месте содержится информация о том, сколько программе потребуется памяти под неинициализированные данные. В связи с этим мы должны наш исполняемый код поместить в одну «секцию», описания областей памяти с заданным начальным значением — в другую «секцию», описания областей памяти без задания начальных значений — в третью «секцию». Соответствующие секции называются `.text`, `.data` и `.bss`. В нашей простой программе мы обходимся только секцией `.text`, и рассматриваемая директива как раз и приказывает ассемблеру приступить к формированию этой секции. В будущем при рассмотрении более сложных программ нам придётся встретиться со всеми тремя секциями.

Далее в программе мы видим строку

```
_start: mov     eax, 0
```

Как мы уже знаем, словом `mov` обозначается команда, заставляющая процессор переслать некоторые данные из одного места в другое; для команды `mov` мы всегда должны указывать два *операнда*, причём первый из них будет задавать то место, *куда* следует скопировать данные, а второй операнд указывает, *какие* данные следует туда скопировать. В данном конкретном случае команда требует занести число `0` (ноль)

в регистр `EAX`[11]. Значение, хранимое в регистре `EAX`, мы будем использовать в качестве счётчика цикла, то есть оно будет означать, сколько раз мы уже напечатали слово «Hello»; ясно, что в начале этот счётчик должен быть равен нулю, поскольку мы пока не напечатали ничего.

Итак, рассматриваемая строка означает приказ процессору занести ноль в `EAX`; но что за загадочное «`_start:`» в начале строки?

Слово `_start` (знак подчёркивания в данном случае является частью слова) представляет собой пример так называемых ***меток***. Попробуем сначала объяснить, что такое собой представляют эти метки «вообще», а потом расскажем, зачем нужна метка в данном конкретном случае.

Команду `mov eax,0` ассемблер преобразует в некий машинный код[12], который во время выполнения программы будет находиться в какой-то области оперативной памяти (в данном случае — в пяти ячейках, идущих подряд). В некоторых случаях нам нужно знать, какой адрес будет иметь та или иная область памяти; если говорить о командах, то знать адрес нам может потребоваться, например, чтобы в какой-то момент заставить процессор произвести в это место программы условный или безусловный переход (про переходы мы уже говорили, см. стр. 11).

Конечно, мы можем использовать оперативную память и для хранения данных, а не только команд. Области памяти, предназначенные для данных, мы обычно называем ***переменными***, и даём им имена почти так же, как и в привычных нам языках программирования высокого уровня. Естественно, нам требуется знать, какой адрес имеет начало области памяти, отведённой под переменную.

Адрес, как мы уже говорили, задаётся[13] числом из восьми шестнадцатеричных цифр, например, `18b4a0f0`. Запоминать такие числа нам неудобно, к тому же на момент написания программы мы ещё не знаем, в каком именно месте памяти в итоге окажется размещена та или иная команда или переменная. И здесь нам на помощь как раз и приходят

[11] Читатель, уже имеющий опыт программирования на языке ассемблера, может заметить, что «правильнее» это сделать совсем другой командой: `xor eax, eax`, поскольку это позволяет достичь того же эффекта быстрее и с меньшими затратами памяти; однако для простейшего учебного примера такой трюк слишком сложен и требует неоправданно длинных пояснений. Впрочем, позже мы к этому вопросу вернёмся и обязательно рассмотрим этот и другие подобные трюки.

[12] Отметим для наглядности, что машинный код этой команды состоит из пяти байтов: `b8 00 00 00 00`, первый из которых задаёт собственно действие «поместить заданное число в регистр», а также и номер регистра `EAX`. Остальные четыре байта задают (все вместе) то число, которое должно быть помещено в регистр; в данном случае это число 0.

[13] Во всяком случае, для того процессора и той системы, которые мы рассматриваем

метки. **Метка — это вводимое программистом слово (идентификатор), с которым ассемблер ассоциирует некоторое число, чаще всего — адрес в памяти**. В данном случае `_start` как раз и есть такая метка. Если ассемблер видит метку *перед* командой (или, как мы увидим позже, директивой, выделяющей память под переменную), он воспринимает это как указание завести в своих внутренних таблицах новую метку и связать с ней соответствующий адрес, если же метка встречается в параметрах команды, то ассемблер «вспоминает», какой именно адрес (или просто число) связано с данной меткой и подставляет этот адрес (число) вместо метки в команду. Таким образом, с меткой `_start` в нашей программе будет связано число, представляющее собой адрес ячейки, начиная с которой в оперативной памяти будет размещён машинный код, соответствующий команде `mov eax,0` (код `b8 00 00 00 00`).

Важно понимать, что метки существуют только в памяти самого ассемблера и только во время трансляции программы. Готовая к исполнению программа на машинном коде не будет содержать никаких меток, а только подставленные вместо них адреса.

После метки мы поставили символ двоеточия. Интересно, что мы могли бы его и не ставить. Некоторые ассемблеры отличают метки, снабженные двоеточиями, от меток без двоеточий; но наш NASM к таким не относится. Иначе говоря, мы сами решаем, ставить двоеточие после метки или нет. Обычно программисты ставят двоеточия после меток, которыми помечены машинные команды (то есть после таких меток, куда можно передать управление), но не ставят двоеточия после меток, помечающих данные в памяти (переменные). Поскольку метка `_start` как раз и помечает команду, после неё мы двоеточие решили поставить.

Однако внимательный читатель может обратить внимание, что никаких переходов на метку `_start` в нашей программе не делается. Зачем же она тогда нужна? Дело в том, что слово «`_start`» — это специальная метка, которой помечается ***точка входа в программу***, то есть то место в программе, куда операционная система должна передать управление после загрузки программы в оперативную память; иначе говоря, метка `_start` обозначает то место, с которого начнётся выполнение программы.

Вернёмся к тексту программы и рассмотрим следующую строчку. Она имеет вид

```
again:  PRINT   "Hello"
```

Как несложно догадаться, слово `again` в начале строки — это ещё одна метка. Слово «again» по-английски означает «снова». Дело в том, что сюда нам придётся вернуться ещё четыре раза, чтобы в итоге слово `Hello` оказалось напечатано пять раз; отсюда и название метки. Стоящее далее в строке слово `PRINT` является ***именем макроса***, а строка `"Hello"` — ***параметром*** этого макроса. Сам макрос описан, как уже говорилось, в файле `stud_io.inc`. «Увидев» имя макроса и параметр, наш ассемблер подставит вместо него целый ряд команд и директив, исполнение которых приведёт в конечном итоге к выдаче на экран строки «Hello».

Очень важно понимать, что `PRINT` не имеет никакого отношения к возможностям центрального процессора. Мы уже несколько раз упоминали этот факт, но тем не менее повторим ещё раз: `PRINT` — это не имя какой-либо команды процессора, процессор как таковой не умеет ничего печатать. Рассматриваемая нами строчка программы представляет собой не команду, а директиву, также называемую ***макровызовом***. Повинуясь этой директиве, ассемблер сформирует фрагмент текста на языке ассемблера (отметим для наглядности, что в данном случае этот фрагмент будет состоять из 23 строк в случае применения ОС GNU/Linux и из 15 строчек — для ОС FreeBSD) и сам же оттранслирует этот фрагмент, получив последовательность машинных инструкций. Эти инструкции будут содержать, в числе прочего, и обращение к операционной системе за услугой вывода данных (системный вызов `write`).

Набор макросов, включающий в себя и макрос `PRINT`, введён для удобства работы на первых порах, пока мы ещё не знаем, как обращаться к операционной системе. Позже мы узнаем это, и тогда макросы, описанные в файле `stud_io.inc`, станут нам не нужны; более того, мы и сами научимся создавать такие макросы.

Вернёмся к тексту нашего примера. Следующая строчка имеет вид

```
PUTCHAR 10
```

Это тоже вызов макроса, называемого `PUTCHAR` и предназначенного для вывода на печать одного символа. В данном случае мы используем его для вывода символа с кодом `10`; это специальный символ, обозначающий ***перевод строки***, то есть при выводе этого символа на печать курсор перейдёт на следующую строку. Обратите внимание, что в этой и последующих строках присутствуют только команды и макровызовы, а меток нет. Они нам не нужны, поскольку ни на одну из последующих команд мы не собираемся делать переходы, и, значит, нам не нужна информация об адресах в памяти, где будут располагаться эти команды.

Следующая строка в программе такая:

```
inc     eax
```

Здесь мы видим машинную команду `inc`, означающую приказ увеличить заданный регистр на 1. В данном случае увеличивается регистр `EAX`. Напомним, что в регистре `EAX` мы условились хранить информацию о том, сколько раз уже напечатано слово «Hello». Поскольку выполнение двух предыдущих строчек программы, содержащих вызовы макросов `PRINT` и `PUTCHAR`, привело в конечном счёте как раз к печати слова «Hello», следует отразить этот факт в регистре, что мы и делаем. Отметим, что машинный код этой команды оказывается очень коротким — всего один байт (шестнадцатеричное `40`, десятичное 64).

Далее в нашей программе идёт команда сравнения:

```
cmp     eax, 5
```

Машинная команда сравнения двух целых чисел обозначается мнемоникой `cmp` от английского «to compare» — сравнивать. В данном случае сравниваются содержимое регистра `EAX` и число `5`. Результаты сравнения записываются в специальный регистр процессора, называемый ***регистром флагов***. Это позволяет, например, произвести *условный переход* в зависимости от результатов предшествующего сравнения, что мы в следующей строчке программы и делаем:

```
jl      again
```

Здесь `jl` (от слов «Jump if Lower») — это мнемоника для машинной команды условного перехода, который выполняется в случае, если предшествующее сравнение дало результат «первый операнд меньше второго», то есть, в нашем случае, если число в регистре `EAX` оказалось меньше, чем `5`. В терминах нашей задачи это означает, что слово «Hello» было напечатано меньше пяти раз и, стало быть, необходимо продолжать его печатать, что и делается переходом (***передачей управления***) на команду, помеченную меткой `again`.

Если результат сравнения был любым другим, кроме «меньше», команда `jl` не произведёт никаких действий, и процессор, таким образом, перейдёт к выполнению следующей по порядку команды. Это произойдёт в случае, если слово «Hello» уже было напечатано 5 раз, так что цикл пора заканчивать. После окончания цикла наша исходная задача оказывается решена, и, стало быть, программу тоже пора завершать. Для этого и предназначена следующая строка программы:

```
        FINISH
```

Слово `FINISH` тоже обозначает макрос; этот макрос разворачивается в последовательность команд, осуществляющих обращение к операционной системе с просьбой завершить выполнение нашей программы.

Нам осталось вернуться к началу программы и рассмотреть строку, имеющую вид

```
    global  _start
```

Слово `global` — это директива, которая требует от ассемблера считать некоторую метку «глобальной», то есть как бы видимой извне (если говорить строго, видимой извне объектного модуля; эти понятия мы будем рассматривать позднее). В данном случае «глобальной» объявляется метка `_start`. Как мы уже знаем, это специальная метка, которой помечается ***точка входа в программу***, то есть то место в программе, куда операционная система должна передать управление после загрузки программы в оперативную память. Ясно, что эта метка должна быть видна извне, то есть ассемблер должен оставить информацию об этой метке видимой для системного компоновщика. Это и достигается директивой `global`.

Итак, наша программа состоит из трёх частей: подготовки, *цикла*, начало которого отмечено меткой `again`, и завершающей части, состоящей из одной строчки `FINISH`. Перед началом цикла мы заносим в регистр `EAX` число `0`, затем на каждой итерации цикла печатаем слово «Hello», делаем перевод строки, увеличиваем на единицу содержимое регистра `EAX`, сравниваем его с числом `5`; если в регистре `EAX` всё ещё содержится число, меньшее пяти, переходим снова к началу цикла (то есть на метку `again`), в противном случае выходим из цикла и завершаем выполнение программы.

Чтобы попробовать приведённую программу, как говорится, в деле, необходимо войти в систему Unix, вооружиться каким-нибудь редактором текстов, набрать вышеприведённую программу и сохранить её в файле с именем, заканчивающимся[14] на `.asm` — именно так обычно называют файлы, содержащие исходный текст на языке ассемблера.

Итак, допустим, мы сохранили текст программы в файле `hello5.asm`. Для получения исполняемого файла нам необходимо выполнить два действия. Первое — это вызов ассемблера NASM, который,

[14] Работая в системе семейства Windows, мы, возможно, сказали бы, что `.asm` — это «расширение» файла. В ОС Unix понятие «расширения» обычно не используется, вместо него мы говорим, что имя *заканчивается на* `.asm` или что имя *имеет суффикс* `.asm`.

используя заданный нами исходный текст, построит *объектный модуль*. Объектный модуль — это ещё не исполняемый файл; дело в том, что большие программы обычно состоят из целого набора исходных файлов, называемых ***модулями***, плюс к тому мы можем захотеть воспользоваться чьими-то сторонними подпрограммами, объединёнными в ***библиотеки***. Таким образом, нам нужно будет соединить несколько модулей воедино и подключить к ним библиотеки; этим занимается ***системный компоновщик***, также называемый иногда ***редактором связей*** или ***линкером***.

Наша примерная программа состоит всего из одного модуля и не нуждается ни в каких библиотеках, но стадии сборки (компоновки) это не исключает. Это и есть второе действие, необходимое для построения исполняемого файла: небходимо вызвать компоновщик, чтобы он нам из объектного файла построил файл исполняемый.

Итак, для начала вызываем ассемблер NASM:

```
nasm -f elf hello5.asm
```

Флажок «`-f elf`» указывает ассемблеру, что на выходе мы ожидаем объектный файл в формате ELF — именно этот формат используется в нашей системе для исполняемых файлов[15]. Результатом запуска ассемблера станет файл `hello5.o`, содержащий объектный модуль. Теперь мы можем запустить компоновщик, который называется `ld`:

```
ld hello5.o -o hello5
```

Флажком `-o` мы задали *имя исполняемого файла* (`hello5`, на сей раз безо всяких суффиксов). Запустим его на исполнение:

```
./hello5
```

Если мы нигде не ошиблись, мы увидим пять строчек `Hello`.

Макросы `PRINT`, `PUTCHAR` и `FINISH` нам неоднократно потребуются в дальнейшем, поэтому, чтобы не возвращаться к ним, ещё раз приведём описание их возможностей. Отметим, что кроме этих трёх макросов наш файл `stud_io.inc` поддерживает ещё один макрос, называемый `GETCHAR`. В нашем примере он не понадобился, но для полноты картины опишем и его. Итак, всего в файле `stud_io.inc` описаны четыре макроса:

[15] Это верно по крайней мере для современных версий операционных систем GNU/Linux и FreeBSD. В других системах вам может потребоваться другой формат объектных и исполняемых файлов; сведения об этом обычно есть в технической документации.

- `PRINT` предназначен для печати строки; его аргументом должна быть строка в апострофах или двойных кавычках, ничего другого он печатать не умеет.

- `PUTCHAR` предназначен для вывода на печать одного символа. В качестве аргумента он принимает код символа, записанный в виде числа или в виде самого символа, взятого в кавычки или апострофы; также можно в качестве аргумента этого макроса использовать однобайтовый регистр — `AL`, `AH`, `BL`, `BH`, `CL`, `CH`, `DL` или `DH`. **Использовать другие регистры в качестве аргумента** `PUTCHAR` **нельзя!** Наконец, аргументом этого макроса может выступать исполнительный адрес, заключённый в квадратные скобки — в этом случае код символа будет взят из ячейки памяти по этому адресу.

- `GETCHAR` считывает символ из потока стандартного ввода (с клавиатуры). После считывания код символа записывается в регистр `EAX`; поскольку код символа всегда умещается в один байт, его можно извлечь из регистра `AL`, остальные разряды `EAX` будут равны нулю. Если символов больше нет (достигнута так называемая ***ситуация конца файла***, которая в ОС Unix обычно имитируется нажатием Ctrl-D), в `EAX` будет занесено значение `-1` (шестнадцатеричное `FFFFFFFF`, то есть все 32 разряда регистра равны единицам). Никаких параметров этот макрос не принимает.

- `FINISH` завершает выполнение программы. Этот макрос можно вызвать без параметров, а можно вызвать с одним числовым параметром, задающим так называемый ***код завершения процесса***; обычно используют код `0`, если наша программа отработала успешно, и код `1`, если в процессе работы возникли ошибки.

Глава 2. Процессор i386

§ 2.1. Система регистров i386

Регистром называют электронное устройство в составе центрального процессора, способное содержать в себе определённое количество данных в виде двоичных разрядов. В большинстве случаев (но не всегда) содержимое регистра трактуется как целое число, записанное в двоичной системе счисления.

Регистры процессора i386 можно условно разделить на ***регистры общего назначения***, ***сегментные регистры*** и ***специальные регистры***. Каждый регистр имеет своё название[1], состоящее из двух-трёх латинских букв.

Сегментные регистры (`CS`, `DS`, `SS`, `ES`, `GS` и `FS`) в «плоской» модели памяти не используются. Точнее говоря, перед передачей управления пользовательской задаче операционная система заносит в эти регистры некоторые значения, которые задача теоретически может изменить, но ничего хорошего из этого всё равно не выйдет — скорее всего, произойдёт аварийное завершение. Таким образом, мы принимаем во внимание существование этих регистров, но более к ним возвращаться не будем.

Регстры общего назначения процессора i386 — это 32-битные регистры `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP` и `ESP`. Как уже отмечалось на стр. 21, буква `E` в названии этих регистров означает слово «extended», подчёркивая тот факт, что в их современном виде эти регистры появились только в процессоре i386. Для совместимости с предыдущими процессорами семейства x86 каждый 32-битный регистр имеет обособленную младшую половину (младшие 16 бит), имеющую отдельное название, получаемое отбрасыванием буквы `E`, то есть, иначе говоря, мы

[1] Этм процессоры семейства x86 отличаются от многих других процессоров, в которых регистры имеют номера

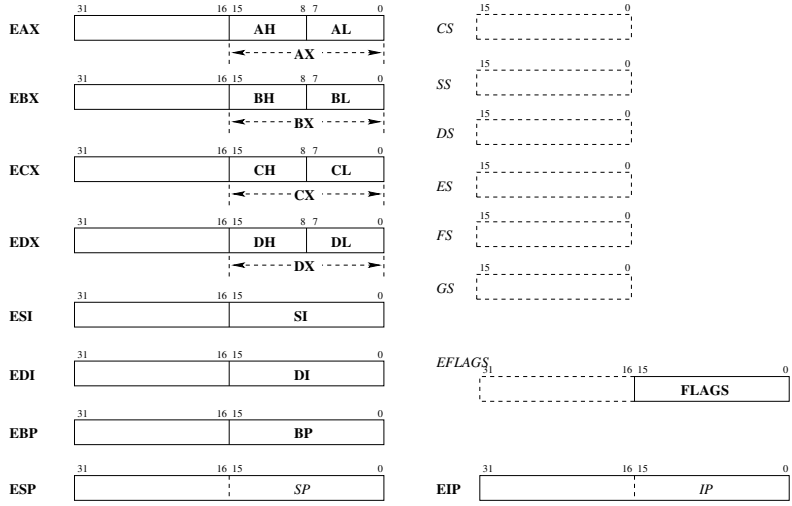

Рис. 2.1. Система регистров i386

можем работать также с 16-битными регистрами `AX`, `BX`, `CX`, `DX`, `SI`, `DI`, `BP` и `SP`, которые представляют собой младшие половины соответствующих 32-битных регистров.

Кроме того, регистры `AX`, `BX`, `CX` и `DX` также делятся на младшие и старшие части, теперь уже восьмибитные. Так, для регистра `AX` его младший байт имеет также название `AL`, а старший байт — `AH` (от слов «low» и «high»). Аналогично мы можем работать с регистрами `BL`, `BH`, `CL`, `CH`, `DL` и `DH`, которые представляют собой младшие и старшие байты регистров `BX`, `CX` и `DX`. Остальные регистры общего назначения таких обособленных однобайтовых подрегистров не имеют.

Каждый из регистров общего назначения, несмотря на такое название, в некоторых случаях играет специфическую, только ему присущую роль, частично закодированную в имени регистра. Так, в имени регистра `AX` буква `A` обозначает слово «accumulator»; на многих архитектурах, включая знаменитый IAS Джона фон Неймана, ***аккумулятором*** называли регистр, участвующий (по определению) во всех арифметических операциях, во-первых, в качестве одного из операндов, и, во-вторых, в качестве места, куда следует поместить результат. Связанная с этим особая роль регистров `AX` и `EAX` проявляется в командах целочисленного умножения и деления (см. §2.3.4).

В имени регистра `BX` буква `B` обозначает слово «base», но никакой особой роли в 32-битных процессорах этому регистру не отведено (хотя в 16-битных процессорах такая роль существовала).

В имени CX буква C обозначает слово ≪counter≫ (счётчик). Регистры ECX, CX, а в некоторых случаях даже CL используются во многих машинных командах, предполагающих (в том или ином смысле) определённое количество итераций.

Имя регистра DX символизирует слово ≪data≫ (данные). В особой роли регистр EDX (или DX, если выполняется шестнадцатиразрядная операция) выступает при выполнении операций целочисленного умножения (для хранения части результата, не поместившейся в аккумулятор) и целочисленного деления (для хранения старшей части делимого).

Имена регистров SI и DI означают, соответственно, ≪source index≫ и ≪destination index≫ (индекс источника и индекс назначения). Регистры ESI и EDI используются в командах, работающих с массивами данных, причём ESI хранит адрес текущей позиции в массиве-источнике (например, в области памяти, которую нужно куда-то скопировать), а EDI хранит адрес текущей позиции в массиве-цели (в области памяти, *куда* производится копирование).

Имя регистра BP обозначает ≪base pointer≫ (базовый указатель). Как правило, регистр EBP используется для хранения базового адреса стекового фрейма при вызове подпрограмм, имеющих параметры и локальные переменные.

Наконец, имя регистра SP обозначает ≪stack pointer≫ (указатель стека). Несмотря на принадлежность регистра ESP к группе регистров общего назначения, в реальности он всегда используется именно в качестве указателя стека, то есть хранит адрес текущей позиции вершины аппаратного стека. Поскольку обойтись без стека тяжело, а другие регистры для этой цели не подходят, можно считать, что ESP никогда не выступает ни в какой иной роли.

К регистрам специального назначения мы отнесём ***регистр счётчика команд*** EIP и ***регистр флагов*** FLAGS.

Регистр EIP, имя которого образовано от слов ≪extended instruction pointer≫, хранит в себе *адрес в оперативной памяти, по которому процессору следует извлечь следующую машинную инструкцию, предназначенную к выполнению.* После извлечения инструкции из памяти значение в регистре EIP автоматически увеличивается на длину прочитанной инструкции (отметим, что инструкция может занимать в памяти от одной до одиннадцати идущих подряд ячеек), так что регистр снова содержит адрес команды, которую нужно выполнить следующей.

Как и для регистров общего назначения, младшая половина регистра EIP имеет имя IP, однако использовать его, работая под управлением 32-битной операционной системы, мы всё равно никак не сможем.

Регистр флагов FLAGS — единственный из рассматриваемых нами регистров, который очень редко используется как единое целое, и вовсе никогда не рассматривается как число. Вместо этого каждый двоичный разряд (бит) этого регистра представляет собой ***флаг***, имеющий собственное имя. Некоторые из этих флагов процессор сам устанавливает в ноль или единицу в зависимости от результата очередной выпол-

ненной команды; другие флаги устанавливаются в явном виде соответствующими инструкциями и в дальнейшем влияют на ход выполнения некоторых команд. Перечислим некоторые флаги:

- `ZF` — флаг нулевого результата (zero flag). Этот флаг устанавливается в ходе выполнения арифметических операций и операций сравнения: если в результате операции получился ноль, `ZF` устанавливается в единицу.

- `CF` — флаг переноса (carry flag). После выполнения арифметической операции над беззнаковыми числами этот флаг выставляется в единицу, если потребовался перенос из старшего разряда (то есть результат не поместился в регистр), либо потребовался заём из несуществующего разряда при вычитании (то есть вычитаемое оказалось больше, чем уменьшаемое). В противном случае флаг выставляется в ноль.

- `SF` — флаг знака (sign flag). Устанавливается равным старшему биту результата; напомним, что при работе со знаковыми целыми числами старший бит равен нулю для положительных чисел и единице для отрицательных.

- `OF` — флаг переполнения (overflow flag). Выставляется в единицу, если при сложении двух положительных целых чисел результат (из-за потери старших разрядов) получился отрицательным, или наоборот; речь идёт, естественно, о знаковых числах.

- `DF` — флаг направления (direction flag). Этот флаг можно установить командой `STD` и обнулить командой `CLD`; в зависимости от его значения строковые операции, которые мы будем рассматривать несколько позже, выполняются в прямом или в обратном направлении.

- `PF` — флаг чётности (parity flag) соответствует чётности количества единиц в младшем байте результата последней команды.

- `AF` — флаг полупереноса (auxiliary carry flag). Нам этот флаг не потребуется.

- `IF` — флаг разрешения прерываний (interrupt flag). Нам этот флаг *недоступен*, его можно изменять только в привилегированном режиме работы.

- `TF` — флаг ловушки (trap flag). Нам этот флаг, опять таки, недоступен.

На самом деле такой набор флагов существовал до процессора i386; при переходе к процессору i386 регистр флагов, как и все остальные регистры, увеличился в размерах и превратился в регистр `EFLAGS`, но все новые флаги нам в ограниченном режиме недоступны, так что рассматривать их мы не будем.

§ 2.2. Память, регистры и команда `mov`

Программа может хранить информацию в регистрах центрального процессора и в оперативной памяти. В нашем случае каждая ячейка памяти, способная хранить ровно один байт, имеет свой уникальный *адрес* — число из 32 бит (речь идёт, естественно, о виртуальных адресах, которые мы обсуждали на стр. 18).

§ 2.2.1. Директивы для отведения памяти

Содержимое этого параграфа не имеет прямого отношения к архитектуре процессора i386; здесь мы рассмотрим директивы, являющиеся особенностью конкретного ассемблера. Дело, однако, в том, что нам очень сложно будет обойтись без них при изучении дальнейшего материала.

Ясно, что регистров центрального процессора заведомо не хватит для хранения всей информации, нужной в любой более-менее сложной программе. Поэтому регистры используются лишь для краткосрочного хранения промежуточных результатов, которые вот-вот понадобятся снова, а основную часть информации программа хранит в оперативной памяти.

Один из основополагающих принципов, определяющих архитектуру фон Неймана, состоит в ***однородности памяти***: и сама программа (то есть составляющие её машинные команды), и все данные, с которыми она работает, располагаются в ячейках памяти, одинаковых по своему устройству и имеющих адреса из единого адресного пространства.

Написанные нами условные обозначения машинных команд ассемблер транслирует в некий *образ области памяти* — массив чисел (данных), которые нужно будет записать в смежные ячейки оперативной памяти. Затем при запуске программы в эту область памяти будет передано управление (то есть, попросту говоря, адрес какой-то из этих ячеек будет записан в регистр `EIP`) и центральный процессор начнёт

выполнение нашей программы, используя числа из созданного ассемблером образа в качестве кодов команд.

Аналогично можно использовать ассемблер и для создания образа области памяти, содержащей данные, а не команды. Для этого нужно сообщить ассемблеру, сколько памяти нам необходимо под те или иные нужды, и при этом, возможно, задать те значения, которые в эту память будут помещены перед стартом программы.

Пользуясь нашими указаниями, ассемблер соответствующим образом сформирует отдельно образ памяти, содержащий команды, и отдельно образ памяти, содержащий наши данные, а кроме того, сосчитает, сколько нам нужно такой памяти, о начальном значении которой мы не беспокоимся и для которой, соответственно, не нужно формировать образ, а нужно лишь указать её количество.

Всё это ассемблер запишет в файл с объектным кодом, а системный компоновщик из таких файлов (возможно, нескольких) сформирует исполняемый файл, содержащий (кроме собственно машинного кода), во-первых, те данные, которые нужно записать в память перед стартом программы, и, во-вторых, указания на то, сколько программе понадобится ещё памяти, кроме той, что нужна под размещение машинного кода и исходных данных.

Уже во время выполнения программа может затребовать у операционной системы некоторое количество дополнительной памяти для размещения данных, относительно которых во время создания программы не был известен размер, либо вообще не было точно известно, понадобится их размещать или нет. Такую память, получаемую во время работы программы, обычно называют ***динамической памятью***. В нашем курсе мы не будем рассматривать работу с динамической памятью, но для любознательных читателей сообщим, что в ОС Linux выделение дополнительной памяти производится системным вызовом `brk`, о котором можно узнать из технической документации по ядру. Выделение дополнительной памяти в ОС FreeBSD производится средствами системного вызова `mmap`, который, к сожалению, гораздо сложнее, особенно для использования в программах на языке ассемблера.

Сообщить ассемблеру о наших потребностях в оперативной памяти можно с помощью ***директив резервирования памяти***, которые делятся на два вида: директивы резервирования неинициализированной памяти и директивы задания исходных данных. Обычно перед директивами обоих видов ставится метка, чтобы можно было ссылаться с её помощью на адрес в памяти, где ассемблер отвёл для нас требуемые ячейки.

Директивы резервирования неинициализированной памяти сообщают ассемблеру, что необходимо зарезервировать заданное

количество ячеек памяти, причём ничего, кроме количества, не уточняется. Мы не требуем от ассемблера заполнять отведённую память какими-либо конкретными значениями, нам достаточно, чтобы эта память вообще была в наличии. **Неинициализированная память не занимает места в исполняемом файле**, исполняемый файл лишь содержит указание на то, *сколько* такой памяти нужно. Занимать место (уже в оперативной памяти) неинициализированные области начинают лишь после запуска программы.

Для резервирования заданного количества однобайтовых ячеек используется директива `resb`, для резервирования памяти под определённое количество «слов», то есть *двухбайтовых* значений (например, коротких целых чисел) — директива `resw`, для «двойных слов» (то есть четырёхбайтных значений) используется `resd`; после директивы указывается (в качестве параметра) число, обозначающее количество значений, под которое мы резервируем память. Как уже говорилось, обычно перед директивой резервирования памяти ставится метка. Таким образом, если мы напишем, например, следующие строки:

```
string  resb 20
count   resw 256
x       resd 1
```

то по адресу, связанному с меткой `string`, будет расположен массив из `20` однобайтовых ячеек (такой массив можно, например, использовать для хранения строки символов); по адресу `count` ассемблер отведёт массив из `256` двубайтных «слов», которые можно использовать, например, для каких-нибудь счётчиков; наконец, по адресу `x` будет располагаться одно «двойное слово», то есть четыре байта памяти, которые можно использовать для хранения достаточно большого целого числа.

Директивы второго типа, называемые ***директивами задания исходных данных***, не просто резервируют память, а указывают, какие значения в этой памяти должны находиться к моменту запуска программы. Соответствующие значения указываются после директивы через запятую; памяти отводится столько, сколько указано значений. Для задания однобайтовых значений используется директива `db`, для задания «слов» — директива `dw` и для задания «двойных слов» — директива `dd`. Например, строка

```
fibon   dw 1, 1, 2, 3, 5, 8, 13, 21
```

зарезервирует память под восемь двубайтных «слов» (то есть всего 16 байт), причём в первые два «слова» будет занесено число 1, в третье

слово — число два, в четвёртое — число 5 и т. д. С адресом первого байта отведённой и заполненной таким образом памяти будет ассоциирована метка `fibon`.

Числа можно задавать не только в десятичном виде, но и в шестнадцатеричном, восьмеричном и двоичном. Шестнадцатеричное число в ассемблере NASM можно задать тремя способами: прибавив в конце числа букву `h` (например, `2af3h`), либо написав перед числом символ `$` (`$2af3`), либо поставив перед числом символы `0x`, как в языке Си (`0x2af3`). При использовании символа `$` необходимо следить, чтобы сразу после `$` стояла цифра, а не буква, так что если число начинается с буквы, необходимо добавить `0` (например, `$0f9` вместо просто `$f9`). Аналогично нужно следить за первым символом и при использовании буквы `h`: например, `a21h` ассемблер воспримет как идентификатор, а не как число. Чтобы избежать проблемы, следует написать `0a21h`. С другой стороны, с числом `2fah` такой проблемы изначально не возникает, поскольку первый символ в его записи является арабской цифрой. Восьмеричное число обозначается добавлением после числа буквы `o` или `q` (например, `634o`, `754q`). Наконец, двоичное число обозначается буквой `b` (`10011011b`).

Отдельного упоминания заслуживают коды символов и *текстовые строки*. Для хранения строк символов обычно используются массивы однобайтовых ячеек. Чтобы программисту не нужно было запоминать коды, соответствующие печатным символам (буквам, цифрам и т. п.), вместо кода можно написать сам символ, взяв его в апострофы или двойные кавычки. Например, строка

```
fig7    db '7'
```

разместит в памяти байт, содержащий число 55 — код символа «семёрки», а адрес этой ячейки свяжет с меткой `fig7`. Мы можем написать и сразу целую строку, например, вот так:

```
welmsg  db 'Welcome to Cyberspace!'
```

В этом случае по адресу `welmsg` будет располагаться строка из 16 символов (то есть массив однобайтовых ячеек, содержащих коды соответствующих символов). Как уже было сказано, кавычки можно использовать как одинарные, так и двойные, так что следующая строка полностью аналогична предыдущей:

```
welmsg  db "Welcome to Cyberspace!"
```

Внутри двойных кавычек одинарные рассматриваются как обычный символ; то же самое можно сказать и о символе двойных кавычек внутри одинарных. Таким образом, например, фразу ≪So I say: "Don't panic!"≫ можно задать следующим образом:

```
panic   db 'So I say: "Don', "'", 't panic"'
```

Здесь мы сначала открыли одинарные кавычки, чтобы символ двойных кавычек, обозначающий прямую речь, вошел в нашу строку. Затем, когда нам в строке потребовался апостроф, мы закрыли одинарные кавычки и воспользовались двойными, чтобы набрать символ апострофа. Наконец, мы снова воспользовались одинарными кавычками, чтобы задать остаток нашей фразы, включая и заканчивающий прямую речь символ двойных кавычек.

Отметим, что строками в одинарных и двойных кавычках можно пользоваться не только с директивой `db`, но и с директивами `dw` и `dd`, однако при этом необходимо учитывать некоторые тонкости, которые мы рассматривать не будем.

При написании программ обычно директивы задания исходных данных располагают в секции `.data` (то есть перед описанием данных ставят директиву `section .data`), а директивы резервирования памяти выделяют в секцию `.bss`. Это обусловлено уже упоминавшимся различием в их природе: инициализированные данные нужно хранить в исполняемом файле, тогда как для неинициализированных достаточно указать их общее количество. Секция `.bss` как раз и отличается от `.data` тем, что в исполняемом файле от неё хранится только указание размера; иначе говоря, размер исполняемого файла не зависит от размера секции `.bss`. Так, если мы добавим в секцию `.data` директиву

```
db "This is a string"
```

то размер исполняемого файла увеличится на 16 байт (надо же где-то хранить строку `"This is a string"`), тогда как если мы добавим в секцию `.bss` директиву

```
resd 16
```

размер исполняемого файла вообще никак не изменится, несмотря на то, что памяти выделяется ровно столько же.

§ 2.2.2. Команда `mov`

Одна из самых часто встречающихся в программах на языке ассемблера команд — это команда пересылки данных из одного места в

другое. Она называется `mov` (от слова «move»). Для нас эта команда интересна ещё и тем, что на её примере можно обсудить целый ряд очень важных вопросов, таких как виды операндов, понятие длины операнда, прямую и косвенную адресацию, общий вид исполнительного адреса, научиться работать с метками и т. д.

Итак, команда `mov` имеет два *операнда*, т. е. два параметра, записываемых после команды и задающих объекты, над которыми команда будет работать. Первый операнд задаёт то место, *куда* будут помещены данные, а второй операнд — то, *откуда* данные будут взяты. Так, например, уже знакомая нам по вводным примерам инструкция

```
mov eax, ebx
```

копирует данные из регистра `EBX` в регистр `EAX`.

Важно отметить, что **команда `mov` только копирует данные, не выполняя никаких преобразований**. Для любых преобразований следует воспользоваться другими командами, имеющими соответствующее предназначение.

§ 2.2.3. Виды операндов

В примерах, рассматривавшихся выше, мы встречали по меньшей мере два варианта использования команды `mov`:

```
mov eax, ebx
mov ecx, 5
```

Первый вариант копирует содержимое одного регистра в другой регистр, тогда как второй вариант *заносит в регистр некоторое число, заданное непосредственно в самой команде* (в данном случае число `5`).

На этом примере наглядно видно, что *операнды бывают разных видов*. Если в роли операнда выступает название регистра, то говорят о ***регистровом операнде***; если же значение указано прямо в самой команде, такой операнд называется ***непосредственным операндом***.

На самом деле, в рассматриваемом случае следует говорить не просто о разных типах операндов, а о *двух разных командах*, которые просто обозначаются одинаковой мнемоникой. Две команды `mov` из нашего примера переводятся в совершенно разные машинные коды, причём первая из них занимает в памяти два байта, а вторая — пять (четыре из них тратятся на размещение непосредственного операнда).

Кроме непосредственных и регистровых операндов, существует ещё и третий вид операнда — ***адресный операнд***, называемый также операндом типа «память». В этом случае операнд задаёт (тем или иным

способом) *адрес ячейки или области памяти, с которой надлежит произвести заданное командой действие.*

Необходимо помнить, что **в языке ассемблера NASM операнд типа «память» абсолютно всегда обозначается квадратными скобками**, в которых и пишется собственно адрес.

В простейшем случае адрес задаётся *в явном виде*, то есть в форме числа; обычно при программировании на языке ассемблера вместо чисел мы, как уже говорилось, используем метки. Например, мы можем написать:

```
section .data
; ...
count   dd 0
```

(символ «;» задаёт а языке ассемблера комментарий), описав *область памяти размером в 4 байта, с адресом которой связана метка* `count`, *и в которой исходно хранится число 0*. Если теперь написать

```
section .text
; ...
        mov [count], eax
```

эта команда `mov` будет обозначать копирование данных из регистра `EAX` в область памяти, помеченную меткой `count`, а, например, команда

```
        mov edx, [count]
```

будет, наоборот, обозначать копирование из памяти по адресу `count` в регистр `EDX`.

Чтобы понять, зачем нужны квадратные скобки, рассмотрим команду

```
        mov edx, count
```

Вспомним, что метку (в данном случае `count`), как мы уже говорили на стр. 25, ассемблер просто заменяет на некоторое *число*, в данном случае — адрес области памяти. Таким образом, если, например, область память `count` расположена в ячейках, адреса которых начинаются с `40f2a008`, то вышеприведённая команда — это абсолютно то же самое, как если бы мы написали

```
        mov edx, 40f2a008h
```

Теперь очевидно, что это просто уже знакомая нам форма команды `mov` с непосредственным операндом, т. е. эта команда *заносит в регистр `EDX` число `40f2a008`*, не вникая в то, является ли это число адресом какой-либо ячейки памяти или нет. Если же мы добавим квадратные скобки, речь пойдёт уже об *обращении к памяти* по заданному адресу.

§ 2.2.4. Прямая и косвенная адресация

Задать адрес области памяти в виде числа или метки возможно не всегда. Во многих случаях нам приходится тем или иным способом *вычислять* адрес, и уже затем обращаться к области памяти по такому вычисленному адресу. Например, именно так будут обстоять дела, если нам потребуется заполнить все элементы какого-нибудь массива заданными значениями: адрес начала массива нам наверняка известен, но нужно будет организовать цикл (по элементам массива) и на каждом шаге цикла выполнять копирование заданного значения в *очередной* (каждый раз другой) элемент массива. Самый простой способ исполнить это — перед входом в цикл задать некий адрес равным адресу начала массива и на каждой итерации увеличивать его.

Важное отличие от простейшего случая, рассмотренного в предыдущем параграфе, состоит в том, что *адрес, используемый для доступа к памяти, будет вычисляться во время исполнения программы*, а не задаваться при её написании. Таким образом, вместо указания процессору «обратись к области памяти по такому-то адресу» нам нужно потребовать действия более сложного: «возьми там-то (например, в регистре) значение, используй это значение в качестве адреса и по этому адресу обратись к памяти».

Такой способ обращения к памяти называют ***косвенной адресацией*** (в отличие от ***прямой адресации***, при которой адрес задаётся явно).

Процессор i386 позволяет для косвенной адресации использовать только значения, хранимые в регистрах процессора. Простейший вид косвенной адресации — это обращение к памяти по адресу, хранящемуся в одном из регистров общего назначения. Например, команда

```
mov ebx, [eax]
```

означает «возьми значение в регистре `EAX`, используй это значение в качестве адреса, по этому адресу обратись к памяти, возьми оттуда 4 байта и занеси эти 4 байта в регистр `EBX`», тогда как команда

```
mov ebx, eax
```

означала, как мы уже видели, просто «скопируй содержимое регистра EAX в регистр EBX».

Рассмотрим небольшой пример. Пусть у нас есть массив из однобайтовых элементов, предназначенный для хранения строки символов, и нам необходимо в каждый элемент этого массива занести код символа '@'. Посмотрим, с помощью какого фрагмента кода мы можем это сделать (воспользуемся командами, уже знакомыми нам из примера на стр. 22)[2]

```
section .bss
array   resb 256        ; массив размером 256 байт

section .text
; ...
        mov ecx, 256    ; кол-во элементов -> в счётчик (ECX)
        mov edi, array  ; адрес массива -> в EDI
        mov al, '@'     ; нужный код -> в однобайтовый AL
again:  mov [edi], al   ; заносим код в очередной элемент
        inc edi         ; увеличиваем адрес
        dec ecx         ; уменьшаем счётчик
        jnz again       ; если там не ноль, повторяем цикл
```

Здесь мы использовали регистр ECX для хранения числа итераций цикла, которые ещё осталось выполнить (изначально 256, на каждой итерации уменьшаем на единицу, а достигнув нуля — заканчиваем цикл), а для хранения адреса мы воспользовались регистром EDI, в который перед входом в цикл занесли адрес начала массива array, а на каждой итерации увеличивали его на единицу, переходя, таким образом, к следующей ячейке.

Внимательный читатель может заметить, что фрагмент кода написан не совсем рационально. Во-первых, можно было бы использовать лишь один изменяемый регистр, либо сравнивая его не с нулём, а с числом 256, либо просматривая массив с конца. Во-вторых, не совсем понятно, зачем для хранения кода символа использовался регистр AL, ведь можно было использовать непосредственный операнд прямо в команде, заносящей значение в очередной элемент массива.

Всё это действительно так, но для этого нам пришлось бы воспользоваться, во-первых, явным указанием размера операнда, а это мы ещё не обсуждали;

[2]Здесь и далее комментарии к текстам примеров приводятся на русском языке. Это допустимо в учебном пособии, исходя из соображений наглядности. Следует, однако, учитывать, что в практическом программировании наличие кириллических символов в тексте программы представляет собой пример крайне плохого стиля. Комментарии в программах следует писать по-английски, что позволит любому программисту в мире прочитать текст вашей программы.

и, во вторых, пришлось бы использовать команду cmp, либо усложнить команду присваивания начального значения адреса. Таким образом, причина применения нами такого нерационального кода здесь — желание ограничиться наименьшим количеством пояснений, отвлекающих внимание от основной задачи.

§ 2.2.5. Общий вид исполнительного адреса

Как видно из предыдущего параграфа, адрес для обращения к памяти не всегда задан заранее; мы можем вычислить адрес уже во время выполнения программы, занести результат вычислений в регистр процессора и воспользоваться косвенной адресацией.

Адрес, по которому очередная машинная команда произведёт обращение к памяти (неважно, задан ли этот адрес явно или вычислен) называется ***исполнительным адресом***.

В предыдущем параграфе мы рассматривали ситуации, когда адрес вычислен, результат вычислений занесён в регистр и именно значение, хранящееся в регистре, используется в качестве исполнительного адреса. Для удобства программирования процессор i386 позволяет и более сложные способы задания исполнительного адреса, при которых *исполнительный адрес вычисляется уже в ходе выполнения команды.*

Если быть точным, мы можем потребовать от процессора взять некоторое заранее заданное значение (возможно, равное нулю, а возможно, и не нулевое), прибавить к нему значение, хранящееся в одном из регистров, а затем взять значение, хранящееся в ещё одном из регистров, умножить на 1, 2, 4 или 8 и прибавить результат к уже имеющемуся адресу. Например, мы можем написать

```
mov eax, [array+ebx+2*edi]
```

В результате такой команды процессор сложит число (заданное меткой `array`) с содержимым регистра `EBX` и удвоенным содержимым регистра `EDI`, результат такого сложения использует в качестве исполнительного адреса, извлечёт из области памяти по этому адресу 4 байта и скопирует их в регистр `EAX`.

Каждое из трёх слагаемых, используемых в исполнительном адресе, является необязательным, то есть мы можем использовать только два слагаемых или всего одно (как, собственно, мы и поступали в предыдущих параграфах).

Важно понимать, что выражение в квадратных скобках никоим образом не может быть произвольным. Например, мы не можем взять три регистра, не можем умножить один регистр на 2, а другой на 4, не можем умножать на иные числа, кроме 1, 2, 4 и 8, не можем, например,

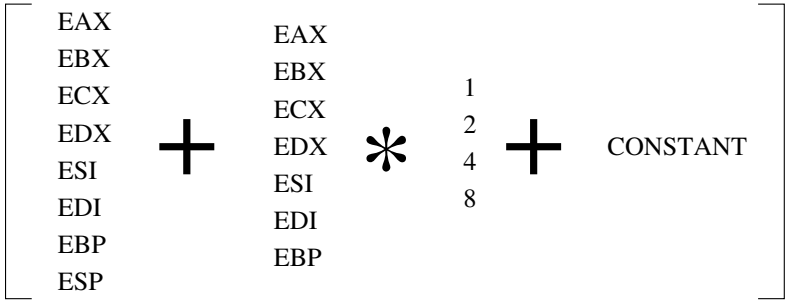

Рис. 2.2. Общий вид исполнительного адреса

перемножить два регистра между собой или вычесть значение регистра, вместо того чтобы прибавлять его. Общий вид исполнительного адреса показан на рис. 2.2; как можно заметить, в качестве регистра, подлежащего домножению на коэффициент, мы не можем использовать `ESP`, в качестве же регистра, значение которого просто добавляется к заданному адресу, можно использовать любой из восьми регистров общего назначения.

Чтобы понять, зачем может понадобиться такой сложный вид исполнительного адреса, достаточно представить себе *двумерный* массив, состоящий, например, из 10 строк, каждая из которых содержит 15 четырёхбайтных целых чисел. Назовём этот массив `matrix`, поставив перед его описанием соответствующую метку:

```
matrix  dd 10*15
```

Для доступа к элементам N-й строки такого массива мы можем вычислить смещение от начала массива до начала этой N-й строки (для этого нужно умножить N на длину строки, составляющую $15 * 4 = 60$ байт), занести результат вычислений, скажем, в `EAX`, затем в другой регистр (например, в `EBX`) занести номер нужного элемента в строке — и исполнительный адрес вида `[matrix+eax+4*ebx]` в точности задаст нам место в памяти, где расположен нужный элемент.

§ 2.2.6. Размеры операндов и их допустимые комбинации

Итак, мы ввели три типа операндов:

1. непосредственные операнды, задающее значение прямо в команде;

2. регистровые операнды, предписывающие взять значение из заданного регистра и/или поместить результат выполнения команды в этот регистр

3. операнды типа «память», задающие адрес, по которому в памяти находится нужное значение и/или по которому в память нужно записать результат работы команды.

Ясно, что не в любой ситуации нам подойдёт любой тип операнда. Например, очевидно, что непосредственный операнд нельзя использовать в качестве первого аргумента команды `mov`, ведь этот аргумент должен задавать то место, *куда* производится копирование данных; мы можем копировать данные в регистр или в область оперативной памяти, однако непосредственные операнды ни того, ни другого не задают.

Имеются и другие ограничения, налагаемые, как правило, устройством самого процессора как электронной схемы. Так, например, ни в команде `mov`, ни в других командах нельзя использовать сразу два операнда типа «память». Если необходимо, скажем, скопировать значение из области памяти `x` в область памяти `y`, необходимо делать это через регистр:

```
mov eax, [x]
mov [y], eax
```

Команда `mov [y],[x]` будет отвергнута ассемблером как ошибочная, поскольку ей не соответствует никакой машинный код: процессор попросту *не умеет* выполнять такое копирование за одну инструкцию.

Все остальные комбинации типов операндов для команды `mov` являются допустимыми, то есть за одну команду `mov` мы можем:

1. скопировать значение из регистра в регистр

2. скопировать значение из регистра в память

3. скопировать значение из памяти в регистр

4. задать (непосредственным операндом) значение регистра

5. задать (непосредственным операндом) значение ячейки или области памяти.

Последний вариант заслуживает особого рассмотрения. До сих пор во всех командах, которые мы использовали в примерах, хотя бы один из операндов был регистровым; это позволяло не думать о *размере* операндов, то есть о том, являются ли наши операнды отдельными байтами, двухбайтовыми «словами» или четырёхбайтовыми «двойными словами». Отметим, что команда `mov` не может пересылать данные между операндами разного размера (например, между однобайтовым регистром `AL` и двухбайтовым регистром `CX`); поэтому всегда, если хотя бы один из операндов является регистровым, можно однозначно сказать, какого размера порция данных подлежит обработке (в данном случае простому копированию).

Однако же в варианте, когда первый операнд команды `mov` задаёт адрес в памяти, куда нужно записать значение, а второй является непосредственным (то есть записываемое значение задано прямо в команде), ассемблер не знает и не имеет оснований предполагать, какого конкретно размера нужно переслать порцию данных, или, иначе говоря, сколько байт памяти, начиная с заданного адреса, должно быть записано. Поэтому, например, команда

```
mov [x], 25     ; ОШИБКА!!!
```

будет отвергнута как ошибочная: непонятно, имеется ли в виду *байт* со значением 25, «*слово*» со значением 25 или «*двойное слово*» со значением 25.

Тем не менее, команда, подобная вышеприведённой, вполне может понадобиться, и процессор умеет такую команду выполнять. Чтобы воспользоваться такой командой, нам нужно просто указать ассемблеру, что конкретно мы имеем в виду. Это делается указанием ***спецификатора размера*** перед любым из операндов; в качестве такого спецификатора может выступать слово `byte`, `word` или `dword`, обозначающие, соответственно, байт, слово или двойное слово (т. е. размер 1, 2 или 4 байта). Так, например, если мы хотели записать число `25` в четырёхбайтную область памяти, находящуюся по адресу `x`, мы можем написать

```
mov [x], dword 25
```

или

```
mov dword [x], 25
```

Сделаем одно важное замечание. Несмотря на использование одной и той же мнемоники, командам языка ассемблера могут соответствовать совершенно разные машинные команды. Так,

```
mov eax, 2
mov eax, [x]
mov [x], eax
mov [x], al
```

представляют собой четыре совершенно разные машинные команды. Вместе с тем, команды

```
mov eax, 2
mov eax, x
```

используют один и тот же машинный код операции и различаются только значением второго операнда, который в обоих случаях непосредственный (действительно, ведь метка `x` будет заменена на адрес, то есть просто число).

§ 2.2.7. Команда `lea`

Возможности процессора по вычислению исполнительного адреса можно задействовать и отдельно от обращения к памяти. Для этого предусмотрена команда `lea` (название образовано от слов «load effective address»).

Команда имеет два операнда, причём первый из них обязан быть регистровым (размером 2 или 4 байта), а второй — операндом типа «память». При этом никакого обращения к памяти команда не делает; вместо этого в регистр, указанный первым операндом, заносится *адрес*, вычисленный обычным способом для второго операнда. Если первый операнд — двухбайтный регистр, то в него будут записаны младшие 16 бит вычисленного адреса.

Например, команда

```
lea eax, [1000+ebx+8*ecx]
```

возьмёт значение регистра `ECX`, умножит его на 8, прибавит к этому значение регистра `EBX` и число 1000, а полученный результат занесёт в регистр `EAX`. Разумеется, вместо числа можно использовать и метку. Ограничения на выражение в скобках точно такие же, как и в других случаях использования операнда типа «память» (см. рис. 2.2 на стр. 45).

Подчеркнём ещё раз, что **команда `lea` только вычисляет адрес, не обращаясь к памяти**, несмотря на использование операнда типа «память».

§ 2.3. Целочисленная арифметика

§ 2.3.1. Простые команды сложения и вычитания

Операции сложения и вычитания над целыми числами производятся, соответственно, командами `add` и `sub`. Обе команды имеют по два операнда, причём первый из них задаёт и одно из чисел, и место, куда следует записать результат; второй операнд задаёт второе число для операции (второе слагаемое, либо вычитаемое). Ясно, что первый операнд обязан быть регистровым либо типа «память»; второй операнд у обеих команд может быть любого типа. Как и для команды `mov`, для команд `add` и `sub` нельзя использовать два операнда типа «память» одновременно.

Например, команда

```
add eax, ebx
```

означает «взять значение из регистра `EAX`, прибавить к нему значение из регистра `EBX`, а результат записать обратно в регистр `EAX`». Команда

```
sub [x], ecx
```

означает «взять четырёхбайтное число из памяти по адресу `x`, вычесть из него значение из регистра `ECX`, результат записать обратно в память по тому же адресу». Команда

```
add edx, 12
```

увеличит на `12` содержимое регистра `EDX`, а команда

```
add dword [x], 12
```

сделает то же самое с четырёхбайтной областью памяти по адресу `x`; обратите внимание, что нам пришлось явно указать размер операнда (см. § 2.2.6, стр. 47).

Интересно, что команды `add` и `sub` работают правильно вне зависимости от того, считаем ли мы их операнды числами знаковыми или беззнаковыми.

В зависимости от полученного результата команды `add` и `sub` выставляют значения *флагов* `OF`, `CF`, `ZF` и `SF` (см. стр. 34), однако не всегда эти флаги имеет смысл рассматривать.

Флаг `ZF` устанавливается в единицу, если в результате последней операции получился ноль, в противном случае флаг сбрасывается; ясно, что значение этого флага осмысленно как для знаковых, так и для беззнаковых чисел.

Флаг SF устанавливается в единицу, если получено отрицательное число, иначе он сбрасывается в ноль. Процессор производит установку этого флага, попросту копируя в него старший бит результата; для знаковых чисел этот бит действительно означает знак числа, но для беззнаковых значение флага SF не имеет никакого смысла.

Флаг OF устанавливается, если произошло *переполнение*, что означает, что в результате сложения двух положительных получилось отрицательное, либо, наоборот, в результате сложения двух отрицательных получилось положительное, и т. д. Ясно, что этот флаг, как и предыдущий, не имеет никакого смысла для беззнаковых чисел.

Наконец, флаг CF устанавливается, если (для беззнаковых чисел) произошел перенос из старшего разряда, либо произошел заём из несуществующего разряда. По смыслу этот флаг является аналогом OF в применении к беззнаковым числам (результат не поместился в размер операнда, либо получился отрицательным). Для знаковых чисел этот флаг смысла не имеет.

Подчеркнём, что **при сложении и вычитании процессор не знает, работает ли он со знаковыми или с беззнаковыми числами**. Схематически сложение и вычитание производится абсолютно одинаково вне зависимости от «знаковости» операндов; флаги процессор выставляет все, т. е. и те, что имеют смысл только для знаковых, и те, что имеют смысл только для беззнаковых. Помнить о том, какие числа имеются в виду — это обязанность программиста; именно программист должен использовать набор флагов, соответствующий знаковости обрабатываемых чисел.

§ 2.3.2. Сложение и вычитание с переносом

Наличие флага переноса позволяет организовать сложение и вычитание чисел, не помещающихся в регистры, способом, напоминающим школьное сложение и вычитание «в столбик». Для этого в процессоре i386 предусмотрены команды adc и sbb. По своей работе и свойствам они полностью аналогичны командам add и sub, но отличаются от них тем, что учитывают значение флага переноса (CF) на момент начала выполнения операции. Команда adc *добавляет* к своему итоговому результату значение флага переноса, команда sbb, напротив, *вычитает* значение флага переноса из своего результата. После того как результат сформирован, обе команды заново выставляют все флаги, включая и CF, уже в соответствии с новым результатом.

Приведём пример. Пусть у нас есть два 64-битных целых числа, причём первое записано в регистры EDX (старшие 32 бита) и EAX (младшие 32 бита), а второе точно так же записано в регистры EBX и ECX. Тогда сложить эти два числа можно командами

```
add eax, ecx   ; складываем младшие части
adc edx, ebx   ; теперь старшие, с учётом переноса
```

если же нам понадобится произвести вычитание, то это делается командами

```
sub eax, ecx   ; вычитаем младшие части
sbb edx, ebx   ; теперь старшие, с учётом заёма
```

§ 2.3.3. Команды inc, dec, neg и cmp

Чтобы завершить рассмотрение простейших арифметических операций, рассмотрим ещё четыре команды.

Команды inc и dec, с которыми мы уже сталкивались в ранее приведённых примерах имеют всего один операнд (регистровый или типа «память») и производят, соответственно, увеличение и уменьшение на единицу.

Обе команды устанавливают флаги ZF, OF и SF, но не затрагивают флаг CF.

Отметим, что при использовании этих команд с операндом типа «память» указание размера операнда оказывается *обязательным*: действительно, для ассемблера нет другого способа понять, какого размера область памяти имеется в виду.

Команда neg, также имеющая один операнд, обозначает *смену знака*, то есть операцию «унарный минус». Ясно, что применение её имеет смысл только к знаковым числам.

Наконец, команда cmp (от слова «compare» — «сравнить») производит точно такое же вычитание, как и команда sub, за исключением того, что результат никуда не записывается. Команда вызывается ради установки флагов, обычно сразу после неё следует команда условного перехода.

§ 2.3.4. Целочисленное умножение и деление

В отличие от сложения и вычитания, умножение и деление схематически реализуется сравнительно сложно[3], так что команды умножения и деления могут показаться организованными очень неудобно для программиста. Причина этого, по-видимому, в том, что создатели процессора i386 и его предшественников действовали здесь прежде всего из соображений удобства реализации самого процессора.

Надо сказать, что умножение и деление доставляет некоторые сложности не только разработчикам процессоров, но и программистам, и отнюдь не только в силу неудобности соответствующих команд, но и по самой своей природе. Во-первых, в отличие от сложения и вычитания, умножение и деление для знаковых и беззнаковых чисел производится совершенно по-разному, так что небходимы и различные команды.

Во-вторых, интересные вещи происходят с размерами операндов. При умножении размер (количество значащих битов) результата может быть *вдвое* больше, чем исходных операндов, так что, если мы не хотим потерять информацию, то одним флажком, как при сложении и вычитании, мы тут не обойдёмся: нужен дополнительный регистр для хранения старших битов результата. С делением ситуация ещё интереснее: размер результата всегда меньше размера делимого (как раз на количество значащих разрядов делителя), так что желательно иметь возможность задавать делимое более длинное, чем делитель и результат. Кроме того, целочисленное деление даёт в качестве результата не одно, а два числа: частное и остаток. Разделять между собой операции нахождения частного и остатка нежелательно, поскольку может привести к двухкратному выполнению (на уровне электронных схем) одних и тех же действий.

Все команды целочисленного умножения и деления имеют только один операнд[4], задающий второй множитель в командах умножения и делитель в командах деления, причём этот операнд может быть регистровым или типа «память», но не непосредственным. Что касается первого множителя и делимого, то для их задания, а также в качестве цели для записи результата используются ***неявный операнд***, в качестве которого в данном случае выступают регистры `AL`, `AX`, `EAX`,

[3] На некоторых процессорах класса микроконтроллеров этих операций вообще нет, и причина этого — исключительно сложность их реализации

[4] На самом деле, из этого правила есть исключение: команда целочисленного умножения знаковых чисел `imul` имеет двухместную и даже трёхместную формы, но рассматривать эти формы мы не будем: пользоваться ими ещё сложнее, чем обычной одноместной формой

разрядн. (бит)	умножение		деление		
	неявный множитель	результат умножения	делимое	частное	остаток
8	AL	AX	AX	AL	AH
16	AX	DX:AX	DX:AX	AX	DX
32	EAX	EDX:EAX	EDX:EAX	EAX	EDX

Таблица 2.1. Расположение неявного операнда и результатов для операций целочисленного деления и умножения в зависимости от разрядности явного операнда

а при необходимости — и регистровые пары DX:AX и EDX:EAX (напомним, что буква A означает слово «аккумулятор»; это и есть особая роль регистра EAX, о которой говорилось на стр. 32).

Для умножения беззнаковых чисел применяют команду mul, для умножения знаковых — команду imul. В обоих случаях, в зависимости от разрядности операнда (второго множителя) первый множитель берётся из регистра AL (для однобайтной операции), либо AX (для двухбайтной операции), либо EAX (для четырёхбайтной), а результат помещается в регистр AX (если операнды были однобайтными), либо в регистровую пару DX:AX (для двухбайтной операции), либо в регистровую пару EDX:EAX (для четырёхбайтной операции). Это можно более наглядно представить в виде таблицы (см. табл. 2.1).

Команды mul и imul устанавливают флаги CF и OF в ноль, если старшая половина результата равна нулю (то есть все значащие биты результата уместились в младшей половине), в противном случае CF и OF устанавливаются в единицу. Значения остальных флагов после выполнения mul и imul *не определены* (то есть ничего осмысленного сказать об их значениях нельзя, причём разные процессоры могут устанавливать их по-разному и даже в результате выполнения одной и той же команды на одном и том же процессоре флаги могут получить разные значения).

Для деления (и нахождения остатка от деления) целых чисел применяют команду div (для беззнаковых) и idiv (для знаковых). Единственный операнд команды, как уже говорилось выше, задаёт *делитель*. В зависимости от разрядности этого делителя (1, 2 или 4 байта) делимое берётся из регистра AX, регистровой пары DX:AX или регистровой пары EDX:EAX, частное помещается в регистр AL, AX или EAX, а остаток от деления — в регистры AH, DX или EDX, соответственно (см. табл. 2.1).

Значения флагов после выполнения целочисленного деления не определены.

Отметим, что частное всегда округляется в сторону нуля (для беззнаковых и положительных — в меньшую, для отрицательных — в большую сторону). Знак остатка, вычисляемого командой `imul`, всегда совпадает со знаком делимого, а абсолютная величина (модуль) остатка всегда строго меньше модуля делителя.

§ 2.4. Условные и безусловные переходы

Как уже отмечалось, в обычное последовательное выполнение команд можно вмешаться, выполнив ***передачу управления***, называемую также ***переходом***. Различают команды ***безусловных переходов***, выполняющие немедленную передачу управления в другое место программы без всяких проверок, и команды ***условных переходов***, которые могут, в зависимости от результата проверки некоторого условия, либо выполнить переход в заданную точку, либо не выполнять его — в этом случае выполнение программы, как обычно, продолжится со следующей команды.

§ 2.4.1. Безусловный переход и виды переходов

В системе команд процессора i386 все команды передачи управления подразделяются, в зависимости от «дальности» такой передачи, на *три типа*.

1. ***Дальние*** (`far`) переходы подразумевают передачу управления во фрагмент программы, расположенный *в другом сегменте*. Поскольку под управлением ОС Unix мы используем «плоскую» модель памяти, в которой разделение на сегменты отсутствует (точнее, имеет место лишь один сегмент, «накрывающий» всё наше виртуальное адресное пространство), такие переходы нам понадобиться не могут: у нас попросту нет других сегментов.

2. ***Близкие*** (`near`) переходы — это передача управления в произвольное место внутри одного сегмента; фактически такие переходы представляют собой явное изменение значения `EIP`. В «плоской» модели памяти это именно тот вид переходов, с помощью которого мы можем «прыгнуть» в произвольное место в нашем адресном пространстве.

3. ***Короткие*** (`short`) переходы используются для оптимизации в случае, если точка, куда надлежит «прыгнуть», отстоит от текущей команды не более чем на 127 байт вперёд или 128 байт назад. В машинном коде такой команды смещение задаётся всего одним байтом, отсюда соответствующее ограничение.

При написании команды перехода мы можем явно указать вид нужного нам перехода, поставив после команды слово `short` или `near` (ассемблер понимает, разумеется, и слово `far`, но нам это не нужно). Если этого не сделать, ассемблер выбирает тип перехода *по умолчанию*, причём для безусловных переходов это `near`, что нас обычно устраивает, а вот для условных переходов по умолчанию используется `short`. Вытекающие из этого сложности и способы их преодоления мы обсудим в следующем параграфе, который посвящён условным переходам, а пока вернёмся к переходам безусловным.

Команда безусловного перехода называется `jmp` (от слова «jump», которое буквально переводится как «прыжок»). У команды предусмотрен один операнд, определяющий собственно адрес, куда следует передать управление. Чаще всего используется форма команды `jmp` с непосредственным операндом, то есть адресом, указанным прямо в команде. Естественно, указываем мы не числовой адрес (которого обычно не знаем), а метку. Возможно, однако, использовать и регистровый операнд (в этом случае переход производится по адресу, взятому из регистра), и операнд типа «память» (адрес читается из двойного слова, расположенного в заданной позиции в памяти); такие переходы называют ***косвенными***, в отличие от ***прямых***, для которых адрес задаётся явно. Приведём несколько примеров:

```
jmp cycle   ; переход на метку cycle
jmp eax     ; переход по адресу из регистра EAX
jmp [addr]  ; переход по адресу, содержащемуся
            ; в памяти, которая помечена меткой addr
jmp [eax]   ; переход по адресу, прочитанному из
            ; памяти, расположенной по адресу,
            ; взятому из регистра EAX
```

Здесь первая команда задаёт прямой переход, а остальные — косвенный.

Если метка, на которую нужно перейти, находится достаточно близко к текущей позиции, можно попытаться соптимизировать машинный код, применив слово `short`:

```
mylabel:
```

```
; ...
; небольшое количество команд
; ...
jmp short mylabel
```

На глаз обычно тяжело определить, действительно ли метка находится достаточно близко, тем более что макросы (например, GETCHAR) могут сгенерировать целый ряд команд, иногда слабо предсказуемый по длине. Но на этот счёт можно не беспокоиться: если расстояние до метки окажется больше допустимого, ассемблер выдаст ошибку примерно такого вида:

```
file.asm:35: error: short jump is out of range
```

и останется только найти строку с указанным номером (в данном случае 35) и убрать ≪несработавшее≫ слово short.

§ 2.4.2. Простые условные переходы

В противоположность командам безусловного перехода, команды условного перехода ассемблер по умолчанию считает ≪короткими≫, если не указать тип перехода явно.

Такой, на первый взгляд, странный подход к командам переходов обусловлен историческими причинами: на ранних процессорах линейки x86 условные переходы былы только короткими, других команд просто не было. Процессор i386 и все более поздние процессоры, конечно же, поддерживают и близкие условные переходы; дальние условные переходы до сих пор не поддерживаются, но нам они всё равно не нужны.

Простейшие команды условного перехода производят переход по указанному адресу в случае, если один из флагов равен нулю (сброшен) или единице (установлен). Имена этих команд образуются из буквы J (от слова ≪jump≫, первой буквы названия флага (например, Z для флага ZF) и, возможно, вставленной между ними буквы N (от слова ≪not≫), если переход нужно произвести при условии равенства флага нулю. Все эти команды приведены в табл. 2.2. Напомним, что смысл каждого из флагов мы рассмотрели на стр. 34.

Такие команды условного перехода обычно ставят непосредственно после арифметической операции (например, сразу после команды cmp, см. стр. 51). Например, две команды

```
cmp eax, ebx
jz are_equal
```

можно прочитать как приказ ≪сравнить значения в регистрах EAX и EBX и если они равны, перейти на метку are_equal≫.

команда	условие перехода	команда	условие перехода
jz	ZF=1	jnz	ZF=0
js	SF=1	jns	SF=0
jc	CF=1	jnc	CF=0
jo	OF=1	jno	OF=0
jp	PF=1	jnp	PF=0

Таблица 2.2. Простейшие команды условного перехода

§ 2.4.3. Переходы по результатам сравнений

Если нам нужно сравнить два числа на *равенство*, всё довольно просто: достаточно, как в предыдущем примере, воспользоваться флагом ZF. Но что делать, если нас интересует, например, условие $a < b$? Сначала мы, естественно, применим команду

cmp a, b

(в качестве a и b могут быть любые операнды, нужно только помнить, что они не могут быть оба одновременно операндами типа «память»). Команда выполнит сравнение своих операндов — точнее говоря, вычтет из a значение b и соответствующим образом выставит значения флагов. Но вот дальнейшее, как мы сейчас увидим, оказывается несколько сложнее.

Если числа a и b — знаковые, то на первый взгляд всё просто: вычитание $a - b$ при условии $a < b$ даёт число строго отрицательное, так что флаг знака (SF, sign flag) должен быть установлен, и мы можем воспользоваться командой js или jns. Но ведь результат мог и не поместиться в длину операнда (например, в 32 бита, если мы сравниваем 32-разрядные числа), то есть могло возникнуть переполнение! В этом случае значение флага SF окажется прямо противоположным ожидавшемуся, зато будет взведён флаг OF (overflow flag). Таким образом, условие $a < b$ выполняется в двух случаях: если SF=1, но OF=0 (то есть переполнения не было, число получилось отрицательное), либо если SF=0, но OF=1 (число получилось положительное, но это результат переполнения, а на самом деле результат отрицательный). Иначе говоря, нас интересует, чтобы флаги SF и OF *не были равны друг другу*: SF≠OF. Для такого случая в процессоре i386 предусмотрена команда jl (от слов «**j**ump if **l**ess than»), обозначаемая также мнемоникой jnge («jump if not greater or equal»).

Рассмотрим теперь ситуацию, если числа a и b — беззнаковые. Как

имя ком.	jump if...	выр. $a \vee b$	условие перехода	синоним
равенство				
`je`	equal	$a = b$	`ZF`$= 1$	`jz`
`jne`	not equal	$a \neq b$	`ZF`$= 0$	`jnz`
неравенства для знаковых чисел				
`jl`	less	$a < b$	`SF`$\neq$`OF`	
`jnge`	not greater or equal			
`jle`	less or equal	$a \leq b$	`SF`$\neq$`OF` или `ZF`$= 1$	
`jng`	not greater			
`jg`	greater	$a > b$	`SF`$=$`OF` и `ZF`$= 0$	
`jnle`	not less or equal			
`jge`	greater or equal	$a \geq b$	`SF`$=$`OF`	
`jnl`	not less			
неравенства для беззнаковых чисел				
`jb`	below	$a < b$	`CF`$= 1$	`jc`
`jnae`	not above or equal			
`jbe`	below or equal	$a \leq b$	`CF`$= 1$ или `ZF`$= 1$	
`jna`	not above			
`ja`	above	$a > b$	`CF`$= 0$ и `ZF`$= 0$	
`jnbe`	not below or equal			
`jae`	above or equal	$a \geq b$	`CF`$= 0$	`jnc`
`jnb`	not below			

Таблица 2.3. Команды условного перехода по результатам арифметического сравнения (`cmp` a, b)

мы уже обсуждали в § 2.3.1 (см. стр. 49), по итогам арифметических операций над беззнаковыми числами флаги `OF` и `SF` рассматривать не имеет смысла, но зато осмысленным становится рассмотрение флага `CF` (carry flag), который выставляется в единицу, если по итогам арифметической операции произошел перенос из старшего разряда (при сложении), либо заём из несуществующего разряда (для вычитания). Именно это нам здесь и нужно: если a и b рассматриваются как беззнаковые и $a < b$, то при вычитании $a - b$ как раз и произойдёт такой заём. Таким образом, нам достаточно воспользоваться значением флага `CF`, то есть выполнить команду `jc`, которая специально для данной ситуации имеет синонимы `jb` (≪jump if below≫) и `jnae` (≪jump if not above or equal≫).

Когда нас интересуют соотношения «больше» и «меньше либо равно», необходимо включить в рассмотрение и флаг ZF, который (как для знаковых, так и для беззнаковых чисел) обозначает равенство аргументов предшествующей команды cmp.

Все команды условных переходов по результату арифметического сравнения приведены в табл. 2.3.

§ 2.4.4. Условные переходы и регистр ECX; циклы

Как уже говорилось, некоторые регистры общего назначения в некоторых случаях имеют особую роль; в частности, регистр ECX лучше других приспособлен к роли *счётчика цикла*. Выражается это в том, что в системе команд процессора i386 имеются специальные команды, учитывающие значение ECX, а для других регистров таких команд нет.

Одна из таких команд называется loop и предназначена для организации циклов с заранее известным количеством итераций. В качестве счётчика цикла она использует регистр ECX, в который перед началом цикла необходимо занести нужное число итераций. Сама команда loop выполняет два действия: уменьшает на единицу значение в регистре ECX и, если в результате значение не стало равным нулю, производит переход на заданную метку.

Отметим, что команда loop имеет одно важное ограничение: она выполняет только «короткие» переходы, то есть с её помощью невозможно осуществить переход на метку, отстоящую от самой команды более чем на 128 байт.

Пусть, например, у нас есть массив из 1000 двойных слов, заданный с помощью директивы

```
array   resd 1000
```

и мы хотим посчитать сумму его элементов. Это можно сделать с помощью следующего фрагмента кода:

```
        mov ecx, 1000   ; кол-во итераций
        mov esi, array  ; адрес первого элемента
        mov eax, 0      ; начальное значение суммы
lp:     add eax, [esi]  ; прибавляем число к сумме
        add esi, 4      ; адрес следующего элемента
        loop lp         ; уменьшаем счётчик;
                        ; если нужно - продолжаем
```

Здесь мы использовали фактически две переменные цикла — регистр ECX в качестве счётчика и регистр ESI для хранения адреса текущего элемента массива.

Конечно, можно произвести аналогичное действие и для любого другого регистра общего назначения, воспользовавшись двумя командами. Например, мы можем уменьшить на единицу регистр EAX и осуществить переход на метку lp при условии, что полученный в EAX результат не равен нулю; это будет выглядеть так:

```
dec eax
jnz lp
```

Точно так же можно записать две команды и для регистра ECX:

```
dec ecx
jnz lp
```

Однако команда

```
loop lp
```

делая те же действия, работает гораздо быстрее и занимает меньше памяти.

В примере с массивом можно обойтись и без ESI, одним только счётчиком:

```
        mov ecx, 1000
        mov eax, 0
lp:     add eax, [array+4*ecx-4]
        loop lp
```

Здесь есть два интересных момента. Во-первых, массив мы вынуждены проходить с конца в начало. Во-вторых, исполнительный адрес в команде add имеет несколько странный вид. Действительно, регистр ECX пробегает значения от 1000 до 1 (для нулевого значения цикл уже не выполняется), тогда как адреса элементов массива пробегают значения от array+4*999 до array+4*0, так что умножать на 4 следовало бы не ECX, а (ecx-1). Однако этого мы сделать не можем и просто вычитаем 4. На первый взгляд это противоречит сказанному в §2.2.5 относительно общего вида исполнительного адреса (слагаемое в виде константы должно быть одно, либо ни одного), однако на самом деле ассемблер NASM прямо во время трансляции вычтет значение 4 из значения array и уже в таком виде оттранслирует, так что в итоговом машинном коде константное слагаемое как раз и будет одно.

Рассмотрим теперь две дополнительные команды условного перехода. Команда jcxz (jump if CX iz zero) производит условный переход, *если в регистре CX содержится ноль*. Флаги при этом не учитываются.

Аналогичным образом команда `jecxz` производит переход, если ноль содержится в регистре `ECX`. Как и для команды `loop`, этот переход всегда короткий. Чтобы понять, зачем введены эти команды, представьте себе, что на момент входа в цикл в регистре `ECX` *уже* содержится ноль. Тогда сначала выполнится тело цикла, а потом команда `loop` уменьшит счётчик на единицу, в результате чего счётчик окажется равен максимально возможному целому беззнаковому числу (двоичная запись этого числа состоит из всех единиц), так что тело цикла будет выполнено 2^{32} раз, тогда как по смыслу его, скорее всего, не следовало выполнять вообще. Чтобы избежать таких неприятностей, перед циклом можно поставить команду `jecxz`:

```
        ; заполняем ecx
        jecxz lpq
lp:     ; тело цикла
        ; ...
        loop lp
lpq:
```

В заключение рассмотрим две модификации команды `loop`. Команда `loope`, называемая также `loopz`, производит переход, если в регистре `ECX` — не ноль и при этом флаг `ZF` установлен, тогда как команда `loopne` (или, что то же самое, `loopnz`) — если в регистре `ECX` не ноль и флаг `ZF` сброшен.

§ 2.5. Побитовые операции

§ 2.5.1. Логические операции

Информацию, записанную в регистры и память в виде байтов, слов и двойных слов можно рассматривать не только как представление целых чисел, но и как строки, состоящие из отдельных и (в общем случае) никак не связанных между собой битов.

Для работы с такими битовыми строками используются специальные команды ***побитовых операций***. Простейшими из них являются двухместные команды `and`, `or` и `xor`, выполняющие соответствующую логическую операцию (≪и≫, ≪или≫, ≪исключающее или≫) отдельно над первыми битами обоих операндов, отдельно над вторыми битами и т. д.; результат, представляющий собой битовую строку той же длины, что и операнды, заносится, как обычно для арифметических команд, в регистр или область памяти, определяемую первым операндом. Ограни-

чения на используемые операнды у этих команд такие же, как и у двухместных арифметических команд: первый операнд должен быть либо регистровым, либо типа «память», второй операнд может быть любого типа; нельзя использовать операнд типа «память» одновременно для первого и второго операнда; если ни один из операндов не является регистровым, необходимо указать разрядность операции с помощью одного из слов `byte`, `word` и `dword`.

Интересно, что в программах очень часто встречается команда `xor`, оба операнда которой представляют собой один и тот же регистр, например,

```
xor eax, eax
```

Это означает *обнуление* указанного регистра, т. е. то же самое, что и

```
mov eax, 0
```

Команду `xor` для этого используют, потому что она занимает меньше места (2 байта против 5 для команды `mov`) и работает на несколько тактов быстрее.

Осуществить побитовое отрицание (инверсию) можно с помощью команды `not`, имеющей один операнд. Операнд может быть регистровый или типа «память»; в последнем случае, естественно, необходимо задать длину операнда словом `byte`, `word` или `dword`.

Все эти команды устанавливают флаги `ZF`, `SF` и `PF` в соответствии с результатом; обычно используется только флаг `ZF`.

В случае, если необходимо просто проверить наличие в числе одного из заданных битов, может оказаться удобной команда `test`, которая работает так же, как и команда `and` (то есть выполняет побитовое «и» над своими операндами), но результат никуда не записывает, а только выставляет флаги.

В частности, команду вида

```
test eax, eax
```

часто используют вместо

```
cmp eax, 0
```

для проверки на равенство нулю: она занимает меньше памяти и работает быстрее.

§ 2.5.2. Операции сдвига

Очень часто приходится применять ***операции побитового сдвига***. Простейшие из них — команды ***простого побитового сдвига***

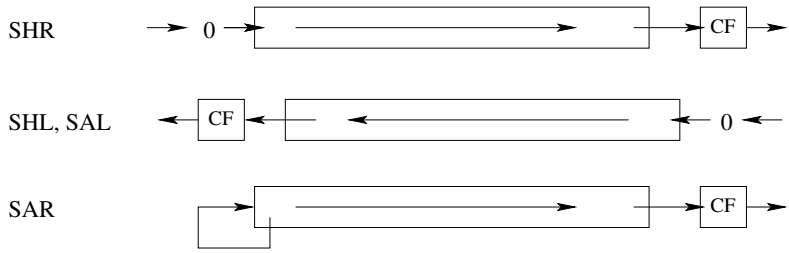

Рис. 2.3. Схема работы команд побитового сдвига

`shr` (**sh**ift **r**ight) и `shl` (**sh**ift **l**eft). Команды имеют два операнда, первый из который указывает, *что* сдвигать, а второй — на сколько битов производить сдвиг. Первый операнд может быть регистровым или типа «память» (во втором случае обязательно указание разрядности). Второй операнд может быть либо непосредственным, то есть числом от 1 до 31 (на самом деле, можно указать любое число, но от него будут использоваться только младшие пять разрядов), либо *регистром* `CL`; никакие другие регистры использовать нельзя. При выполнении этих команд с регистром `CL` в качестве второго операнда процессор игнорирует все разряды `CL`, кроме пяти младших.

Схема сдвига на 1 бит следующая. При сдвиге влево старший бит сдвигаемого числа переносится во флаг `CF`, остальные биты сдвигаются влево (то есть бит № n получает значение, которое до операции имел бит № $n-1$), в младший бит записывается ноль. При сдвиге вправо, наоборот, во флаг `CF` заносится младший бит, все биты сдвигаются вправо (то есть бит № n получает значение, которое до операции имел бит № $n+1$), в старший бит записывается ноль.

Отметим, что для **беззнаковых** чисел сдвиг на n бит влево эквивалентен умножению на 2^n, а сдвиг вправо — целочисленному делению на 2^n с отбрасыванием остатка.

Интересно, что для **знаковых** чисел ситуация со сдвигом влево абсолютно аналогична, а вот сдвиг вправо для любого отрицательного числа даст положительное, ведь в знаковый бит будет записан ноль. Поэтому наряду с командами простого сдвига вводятся также и команды ***арифметического побитового сдвига*** `sal` (shift **a**rithmetic **l**eft) и `sar` (shift **a**rithmetic **r**ight). Команда `sal` делает то же самое, что и команда `shl` (на самом деле, это одна и та же машинная команда). Что касается команды `sar`, то она работает аналогично команде `shr`, за исключением того, что в старшем бите значение сохраняется таким же,

каким оно было до операции; таким образом, если рассматривать сдвигаемую битовую строку как запись знакового целого числа, то операция `sar` не изменит знак числа (положительное останется положительным, отрицательное — отрицательным). Иначе говоря, операция арифметического сдвига вправо эквивалентна делению на 2^n с отбрасыванием остатка *для знаковых целых чисел*.

Операции простых и арифметических сдвигов схематически показаны на рис. 2.3.

Кроме рассмотренных, процессор i386 поддерживает также команды «сложных» побитовых сдвигов `shrd` и `shld`, работающих через два регистра; команды ***циклического побитового сдвига*** `ror` и `rol`; команды циклического сдвига через флаг `CF` — `rcr` и `rcl`. Все эти команды мы рассматривать не будем; при желании читатель может освоить их самостоятельно.

§ 2.6. Стек, подпрограммы, рекурсия

§ 2.6.1. Понятие стека и его предназначение

Как известно, под *стеком* в программировании подразумевают структуру данных, построенную по принципу «первый вошел — последний вышел», т. е. такой объект, над которым определены операции «добавить элемент» и «извлечь элемент», причём элементы, которые были добавлены, извлекаются в обратном порядке.

В применении к низкоуровневому программированию понятие стека существенно у́же: здесь под стеком понимается непрерывная область памяти, для которой хранится ***адрес вершины стека***, причём память в рассматриваемой области ниже вершины (т. е. с адресами, меньшими адреса вершины) считается *свободной*, а память от вершины до конца области (до старших адресов), включая и саму вершину, считается *занятой*; операция добавления в стек некоторого значения уменьшает адрес вершины, сдвигая тем самым вершину вниз и в новую вершину записывает добавляемое значение; операция извлечения считывает значение с вершины стека и сдвигает вершину вверх, увеличивая её адрес.

Стек используется, во-первых, для временного хранения значений регистров; если некоторый регистр хранит важное для нас значение, а нам при этом нужно временно задействовать этот регистр для хранения другого значения, то самый простой способ выйти из положения — это сохранить значение регистра в стеке, затем использовать регистр под

другие нужды, и, наконец, восстановить исходное значение регистра путём извлечения этого значения из стека обратно в регистр.

Во-вторых (и это гораздо более важно) стек используется при вызовах подпрограмм для хранения адресов возврата, для передачи фактических параметров в подпрограммы и для хранения локальных переменных. Именно использование стека позволяет реализовать механизм рекурсии, при котором подпрограмма может прямо или косвенно вызвать сама себя.

§ 2.6.2. Организация стека в процессоре i386

Большинство существующих процессоров поддерживают работу со стеком на уровне машинных команд, и i386 в этом плане не исключение. Команды работы со стеком позволяют заносить в стек и извлекать из него двухбайтные «слова» и четырёхбайтные «двойные слова»; отдельные байты записывать в стек нельзя, так что адрес вершины стека всегда остаётся чётным.

Как уже говорилось (см. стр. 33), регистр `ESP`, формально относящийся к группе регистров общего назначения, тем не менее практически никогда не используется ни в какой иной роли, кроме роли ***указателя стека***; название этого регистра как раз и означает «stack pointer».

Считается, что адрес, содержащийся в `ESP`, указывает на вершину стека, то есть на ту область памяти, где хранится последнее занесённое в стек значение. Стек «растёт вниз», то есть при занесении в стек нового значения `ESP` уменьшается, при извлечении значения — увеличивается.

Занесение значения в стек производится командой `push`, имеющей один операнд. Этот операнд может быть непосредственным, регистровым или типа «память» и иметь размер `word` или `dword` (если операнд не регистровый, то размер необходимо указать явно).

Для извлечения значения из стека используется команда `pop`, операнд которой может быть регистровым или типа «память»; естественно, операнд должен иметь размер «слово» или «двойное слово».

Команды `push` и `pop` совмещают копирование данных (на вершину стека или с неё) со сдвигом самой вершины, то есть изменением значения регистра `ESP`. Понятно, что можно, вообще говоря, обратиться к значению на вершине стека, не извлекая его из стека — применив (в любой команде, допускающей операнд типа «память») операнд `[esp]`. Например, команда

```
mov eax, [esp]
```

скопирует четырёхбайтное значение с вершины стека в регистр EAX.

Как говорилось выше, стек очень удобно использовать для временного хранения значений из регистров:

```
push eax    ; запоминаем eax
;
;     используем eax под посторонние нужды
;
pop eax     ; восстанавливаем eax
```

Рассмотрим более сложный пример. Пусть регистр ESI содержит адрес некоторой строки символов в памяти, причём известно, что строка заканчивается байтом со значением 0 (но неизвестно, какова длина строки) и нам необходимо «обратить» эту строку, то есть записать составляющие её символы в обратном порядке в том же месте памяти[5]. Один из способов сделать это — последовательно записать коды символов в стек, а затем снова пройти строку с начала в конец, извлекая из стека символы и записывая их в ячейки, составляющие строку.

Поскольку записывать в стек однобайтовые значения нельзя, мы будем записывать значения двухбайтовые, причём старший байт просто не будем использовать. Конечно, можно сделать всё более рационально, но нам в данном случае важнее наглядность нашей иллюстрации.

Для промежуточного хранения будем использовать регистр BX, причём только его младший байт (BL) будет содержать полезную информацию, но записывать в стек и извлекать из стека мы будем весь BX целиком. Задача будет решена в два цикла. Перед первым циклом мы занесём ноль в регистр ECX, потом на каждом шаге будем извлекать байт по адресу [esi+ecx] и помещать этот байт (в составе слова) в стек, а ECX увеличивать на единицу, и так до тех пор, пока очередной извлечённый байт не окажется нулевым, что по условиям задачи означает конец строки. В итоге все ненулевые элементы строки окажутся в стеке, а в регистре ECX будет длина строки.

Поскольку для второго цикла заранее известно количество его итераций (длина строки) и оно уже содержится в ECX, мы организуем этот цикл с помощью команды loop. Перед входом в цикл мы проверим, не пуста ли строка (то есть не равен ли ECX нулю), и если строка была пуста, сразу же перейдём в конец нашего фрагмента. Поскольку значение в ECX будет уменьшаться, а строку нам нужно пройти в прямом

[5] Нулевой байт, играющий роль ограничителя, остаётся на месте и никуда не копируется.

направлении — наряду с ECX мы воспользуемся регистром EDI, который в начале установим равным ESI (то есть указывающим на начало строки), а на каждой итерации будем его сдвигать.

Итак, пишем:

```
        xor ebx, ebx        ; обнуляем ebx
        xor ecx, ecx        ; обнуляем ecx
lp:     mov bl, [esi+ecx]   ; очередной байт из строки
        cmp bl, 0           ; конец строки?
        je lpquit           ; если да - конец цикла
        push bx             ; bl нельзя, приходится bx
        inc ecx             ; следующий индекс
        jmp lp              ; повторить цикл
lpquit: jecxz done          ; если строка пустая,
                            ;   больше ничего не делать
        mov edi, esi        ; опять сначала
lp2:    pop bx              ; извлекаем
        mov [edi], bl       ; записываем
        inc edi             ; следующий адрес
        loop lp2            ; повторять ecx раз
done:
```

§ 2.6.3. Дополнительные команды работы со стеком

При необходимости можно занести в стек значение всех регистров общего назначения одной командой; эта команда называется pushad (**push a**ll **d**oublewords). Уточним, что эта команда заносит в стек содержимое регистров EAX, ECX, EDX, EBX, ESP, EBP, ESI и EDI (в указанном порядке), причём значение ESP заносится в том виде, в котором оно было *до* выполнения команды.

Соответствующая команда извлечения из стека называется popad (**pop a**ll **d**oublewords). Она извлекает из стека восемь четырёхбайтных значений и заносит эти значения в регистры в порядке, обратном приведённому для команды pushad, при этом значение, соответствующее регистру ESP, игнорируется (то есть из стека извлекается, но в регистр не заносится).

Регистр флагов (EFLAGS) может быть записан в стек командой pushfd и считан обратно командой popfd, однако при этом, если мы работаем в ограниченном режиме, только некоторые флаги (а именно — флаги, доступные к изменению в ограниченном режиме) могут быть изменены, на остальные команда popfd никак не повлияет.

Существуют аналогичные команды для 16-битных регистров, поддерживаемые для совместимости со старыми процессорами; они называются `pushaw`, `popaw`, `pushfw` и `popfw`, и работают полностью аналогично, но вместо 32-битных регистров используют соответствующие 16-битные. Команды `pushaw` и `popaw` практически никогда не используются, что касается команд `pushfw` и `popfw`, то их использование, в принципе, имеет смысл, если учесть, что в ≪расширенной≫ части регистра `EFLAGS` нет ни одного флага, значение которого мы могли бы поменять в ограниченном режиме работы.

§ 2.6.4. Подпрограммы: общие принципы

Подпрограммой называется некоторая обособленная часть программного кода, которая может быть *вызвана* из главной программы (или из другой подпрограммы); под ***вызовом*** в данном случае понимается временная передача управления подпрограмме с тем, чтобы, когда подпрограмма сделает свою работу, она вернула управление в точку, откуда её вызвали.

Читатель, вне всякого сомнения, уже встречался с подпрограммами. Это, например, процедуры и функции языка Паскаль, функции в языке Си и т. п.

При вызове подпрограммы необходимо запомнить ***адрес возврата***, то есть адрес начала машинной команды, следующей за командой вызова подпрограммы, причём сделать это так, чтобы сама вызываемая подпрограмма могла, когда закончит свою работу, воспользоваться этим сохранённым адресом для возврата управления.

Кроме того, подпрограммы часто получают ***параметры***, влияющие на их работу, и используют в работе ***локальные переменные***. Подо всё это необходимо предусмотреть выделение оперативной памяти (или регистров).

Самый простой способ решения этого вопроса — выделить каждой подпрограмме свою собственную область памяти под хранение всей локальной информации, включая и адрес возврата, и параметры, и локальные переменные. Тогда вызов подпрограммы потребует прежде всего записать в принадлежащую подпрограмме область памяти (в заранее оговорённые места) значения параметров и адрес возврата, а затем передать управление в начало подпрограммы.

Интересно, что когда-то давно именно так с подпрограммами и поступали. Однако с развитием методов и приёмов программирования возникла потребность в ***рекурсии*** — таком построении программы, при котором некоторые подпрограммы могут прямо или косвенно вызывать

сами себя, притом потенциально неограниченное[6] число раз. Ясно, что при каждом рекурсивном вызове требуется новый экземпляр области памяти для хранения адреса возврата, параметров и локальных переменных, причём чем позже такой экземпляр будет создан, тем раньше соответствующий вызов закончит работу, то есть рекурсивные вызовы подпрограмм в определённом смысле подчиняются правилу ≪последний пришел — первый ушел≫. Совершенно логично из этого вытекает идея использования при вызовах подпрограмм уже знакомого нам стека.

В современных вычислительных системах перед вызовом подпрограммы в стек помещаются значения параметров вызова, затем производится собственно вызов, то есть передача управления, которая совмещена с сохранением в том же стеке адреса возврата. Наконец, когда подпрограмма получает управление, она резервирует в стеке определённое количество памяти для хранения локальных переменных, обычно просто сдвигая адрес вершины вниз на соответствующее количество ячеек. Область стековой памяти, содержащую связанные с одним вызовом значения параметров, адрес возврата и локальные переменные, называют ***стековым фреймом***.

§ 2.6.5. Вызов подпрограмм и возврат из них

Вызов подпрограммы, как уже стало ясно из вышесказанного,— это передача управления по адресу начала подпрограммы с одновременным запоминанием в стеке адреса возврата (то есть адреса машинной команды, непосредственно следующей за командой вызова). Процессор i386 предусматривает для этой цели команду `call`; аналогично команде `jmp`, аргумент команды `call` может быть непосредственным (адрес перехода задан непосредственно в команде, например, меткой), регистровым (адрес передачи управления находится в регистре) и типа ≪память≫ (переход нужно осуществить по адресу, прочитанному из заданного места памяти). Команда `call` не имеет ≪короткой≫ формы; поскольку ≪дальняя≫ форма нам, как обычно, не требуется в силу отсутствия сегментов, остаётся только одна форма — близкая (`near`), которую мы всегда и используем.

Возврат из подпрограммы производится командой `ret` (от слова return). В своей простейшей форме эта команда не имеет аргументов. Выполняя эту команду, процессор извлекает 4 байта с вершины стека и записывает их в регистр `EIP`, в результате чего управление передаётся по адресу, который находился в памяти на вершине стека.

[6] Точнее говоря, ограниченное только объемом памяти

Рассмотрим простой пример. Допустим, в нашей программе часто приходится заполнять каким-то однобайтовым значением области памяти разной длины. Такое действие вполне можно оформить в виде подпрограммы. Для простоты картины примем соглашение, что адрес нужной области памяти передаётся через регистр `EDI`, количество однобайтовых ячеек, которые нужно заполнить — через регистр `ECX`, ну а само значение, которое надо записать во все эти ячейки — через регистр `AL`. Код соответствующей подпрограммы может выглядеть, например, так:

```
; fill memory (edi=address, ecx=length, al=value)
fill_memory:
        jecxz   fm_q
fm_lp:  mov     [edi], al
        inc edi
        loop fm_lp
fm_q:   ret
```

Обратиться к такой подпрограмме можно, например, так:

```
        mov edi, my_array
        mov ecx, 256
        mov al, '@'
        call fill_memory
```

В результате такого вызова 256 байт памяти, начиная с адреса, заданного меткой `my_array`, окажутся заполнены кодом символа `'@'` (число 64).

§ 2.6.6. Организация стековых фреймов

Подпрограмма, приведённая в качестве примера в предыдущем параграфе, фактически не использовала механизм стековых фреймов, сохраняя в стеке только адрес возврата. Этого оказалось достаточно, поскольку подпрограмме не требовались локальные переменные, а параметры мы передали через регистры.

Как показывает практика, подпрограммы редко бывают такими простыми. В более сложных случаях нам, безусловно, потребуются локальные переменные, поскольку регистров на всё не хватит. Кроме того, передача параметров через регистры тоже может оказаться неудобна: во-первых, регистров может и не хватить, а во-вторых, подпрограмме могут быть долго нужны значения, переданные через регистры, и

это фактически лишит её возможности использовать под свои внутренние нужды те из регистров, которые были задействованы при передаче параметров. Наконец, передача параметров через регистры (а равно и через какую-либо фиксированную область памяти) лишает нас возможности использовать рекурсию, что тоже, разумеется, плохо.

Поэтому обычно (в особенности при трансляции программы с какого-либо языка высокого уровня, с того же Паскаля или Си) параметры в функции передаются через стек, и на стеке же размещаются локальные переменные. Как было сказано выше, параметры в стеке размещает вызывающая программа, затем в при вызове подпрограммы в стек заносится адрес возврата, и, наконец, уже сама вызванная подпрограмма резервирует место в стеке под локальные переменные. Всё это вместе и образует стековый фрейм.

К содержимому стекового фрейма можно обращаться, используя адреса, «привязанные» к адресу, по которому содержится адрес возврата, то есть, иначе говоря, ту ячейку памяти, начиная с которой в стек был занесён адрес возврата, используют в качестве своего рода реперной точки. Так, если в стек занести три четырёхбайтных параметра, а потом вызвать процедуру, то адрес возврата будет лежать в памяти по адресу `[esp]`, ну а параметры, очевидно, окажутся доступны по адресам `[esp+4]`, `[esp+8]` и `[esp+12]`. Если же разместить в стеке локальные четырёхбайтные переменные, то они окажутся доступны по адресам `[esp-4]`, `[esp-8]` и т. д.

Заметим, что использовать для доступа к параметрам регистр `ESP` оказывается тоже не слишком удобно, ведь в самой процедуре нам тоже может потребоваться стек (как для временного хранения данных, так и для вызова других подпрограмм). Поэтому первым же своим действием подпрограмма обычно сохраняет значение регистра `ESP` в каком-то другом регистре (чаще всего `EBP`) и именно его использует для доступа к параметрам и локальным переменным, ну а регистр `ESP` продолжает играть свою роль указателя стека, изменяясь по мере необходимости; перед возвратом из подпрограммы его обычно восстанавливают в исходном значении (попросту пересылая в него значение из `EBP`), чтобы он снова указывал на адрес возврата.

Наконец, возникает ещё один вопрос: а что если другие подпрограммы тоже используют регистр `EBP` для тех же целей? Ведь в этом случае первый же вызов другой подпрограммы испортит нам всю работу. Можно, конечно, сохранять `EBP` в стеке перед вызовом каждой подпрограммы, но поскольку в программе обычно гораздо больше *вызовов подпрограмм*, чем собственно самих подпрограмм, экономнее оказыва-

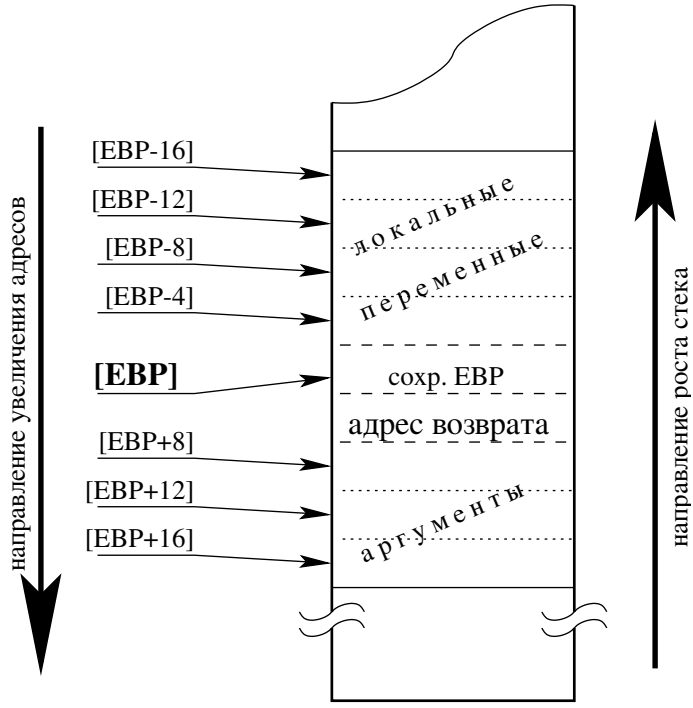

Рис. 2.4. Структура стекового фрейма

ется следовать простому правилу: *каждая подпрограмма должна сама сохранить старое значение EBP и восстановить его перед возвратом управления.*

Для сохранения значения `EBP` используется, разумеется, тоже стек, причём сохранение выполняется простой командой `push ebp` сразу после получения управления. Таким образом, старое значение `EBP` ложится в стек непосредственно после адреса возврата из подпрограммы, и в качестве «точки привязки» используется в дальнейшем именно этот адрес вершины стека. Для этого следующей командой выполняется `mov ebp, esp`. В результате регистр `EBP` указывает на то место в стеке, где находится его же, `EBP`, сохранённое значение; если теперь обратиться к памяти по адресу `[ebp+4]`, мы обнаружим там адрес возврата из подпрограммы, ну а параметры, занесённые в стек перед вызовом подпрограммы, оказываются доступны по адресам `[esp+8]`, `[esp+12]`,

[esp+16] и т. д. Память под локальные переменные выделяется путём простого вычитания нужной длины из текущего значения ESP; так, если под локальные переменные нам нужно 16 байт, то сразу после сохранения EBP и копирования в него содержимого ESP нужно выполнить команду sub esp, 16; если (для простоты картины) все наши локальные переменные тоже занимают по 4 байта, они окажутся доступны по адресам [ebp-4], [ebp-8] и т. д.

Структура стекового фрейма с тремя четырёхбайтными параметрами и четырьмя четырёхбайтными локальными переменными показана на рис. 2.4.

Повторим, что в начале своей работы, согласно нашим договорённостям, каждая подпрограмма должна выполнить

```
push ebp
mov ebp, esp
sub esp, 16   ; вместо 16 подставьте объем
              ; памяти под локальные переменные
```

а завершение подпрограммы теперь должно выглядеть так:

```
mov esp, ebp
pop ebp
ret
```

Интересно, что процессор i386 поддерживает даже специальные команды для обслуживания стековых фреймов. Так, в начале подпрограммы вместо трёх команд, приведённых выше, можно было бы дать одну команду

```
enter 16, 0
```

а вместо двух команд перед ret можно было бы написать

```
leave
```

Проблема, как ни странно, в том, что команды enter и leave работают *медленнее*, чем соответствующий набор простых команд, так что их практически никогда не используют; если дизассемблировать машинный код, сгенерированный компилятором языка Си или Паскаль, мы, скорее всего, обнаружим в начале любой процедуры или функции именно такие команды, как показано выше, и ничего похожего на enter. Единственным оправданием существования команд enter и leave может служить их короткая запись (например, машинная команда leave занимает в памяти всего 1 байт), но в наше время об экономии памяти на машинном коде обычно никто не задумывается; быстродействие практически всегда оказывается важнее.

Сделаем ещё одно важное замечание. При работе под управлением ОС Unix мы можем не беспокоиться ни о наличии стека, ни о задании его размера. Операционная система создаёт стек автоматически при запуске любой задачи и, более того, уже во время её исполнения при необходимости увеличивает размер доступной для стека памяти: по мере того как вершина стека продвигается вниз по виртуальному адресному пространству, операционная система ставит в соответствие виртуальным адресам всё новые и новые страницы физической памяти. Именно поэтому на рис. 2.4 мы изобразили верхний край стека как нечто нечёткое.

§ 2.6.7. Основные конвенции вызовов подпрограмм

Несмотря на подробное описание механизма стековых фреймов, данное в предыдущем параграфе, в некоторых вопросах остаётся возможность для манёвра. Так, например, в каком порядке следует заносить в стек значения фактических параметров? Если мы пишем программу на языке ассемблера, этот вопрос, собственно говоря, не встаёт; однако он оказывается неожиданно принципиальным при создании компиляторов языков программирования высокого уровня.

Создатели компиляторов языка Паскаль обычно идут «очевидным» путём: вызов процедуры или функции транслируется с Паскаля в виде серии команд занесения в стек значений, причём значения заносятся в естественном (для человека) порядке — слева направо; затем в код вставляется команда `call`. Таким образом, когда процедура получает управление, значения фактических параметров располагаются в стеке сверху вниз, то есть *последний* параметр оказывается размещён ближе других к реперной точке фрейма (доступен по адресу `[ebp+8]`) Из этого, в свою очередь, следует, что **для доступа к первому (а равно и к любому другому) фактическому параметру процедуре или функции языка Паскаль необходимо знать общее количество этих параметров**, поскольку расположение n-го параметра в стековом фрейме получается зависящим от общего количества. Так, если у процедуры три четырёхбайтных параметра, то первый из них окажется в стеке по адресу `[ebp+16]`, если же их пять, то первый придётся искать по адресу `[ebp+24]`.

Таким образом, язык Паскаль не допускает создание процедур или функций с переменным числом аргументов, так называемых ***вариадических*** подпрограмм (что вполне нормально для учебного языка, но не совсем приемлемо для языка профессионального). Входящие в язык

Паскаль псевдопроцедуры с переменным числом аргументов, такие как `WriteLn`, на самом деле являются частью самого языка Паскаль; компилятор трансформирует их вызовы в нечто весьма далёкое от вызова подпрограммы на уровне машинного кода. Так или иначе, программист не может на Паскале описать свою процедуру подобного рода.

Создатели языка Си пошли иным путём. При трансляции вызова функции языка Си параметры помещаются в стек *в обратном порядке* и оказываются размещёнными во фрейме в порядке снизу вверх. Таким образом, первый параметр всегда оказывается доступен по адресу `[ebp+8]`, второй — по адресу `[ebp+12]` и т. д., вне всякой зависимости от общего количества параметров (конечно, параметры по крайней мере должны присутствовать, то есть если функция, например, вызвана вообще без параметров, никакого первого параметра в стеке не будет).

Это, с одной стороны, позволяет создание вариадических функций; в частности, в сам по себе язык Си не входит вообще ни одной функции, что же касается таких функций, как `printf`, `scanf` и др., то они реализуются в *библиотеке*, а не в самом языке, и, более того, сами эти функции тоже написаны на Си (как сказано выше, в Паскале так сделать не получается).

С другой стороны, отсутствие в Паскале вариадических подпрограмм позволяет возложить заботы об очистке стека на *вызываемого*. Действительно, подпрограмма языка Паскаль всегда знает, сколько места занимают фактические параметры в её стековом фрейме (поскольку для каждой подпрограммы это количество задано раз и навсегда и не может измениться) и, соответственно, может принять на себя заботу об очистке стека. Как уже говорилось, вызовов подпрограмм в любой программе больше, чам самих подпрограмм, так что за счёт переклады-вания заботы об очистке стека с вызывающего на вызываемого достигается определённая экономия памяти (количества машинных команд).

При использовании соглашений языка Си такая экономия невозможна, поскольку подпрограмма не знает и не может знать (в общем случае[7]), сколько параметров ей передали, так что забота об очистке стека от параметров остаётся на вызывающем; обычно это делается простым увеличением значения `ESP` на число, равное совокупной длине фактических параметров. Например, если подпрограмма `proc1` принимает на

[7] В разных ситуациях используются различные способы фиксации количества параметров; так, функция `printf` узнаёт, сколько параметров нужно извлечь из стека, путём анализа форматной строки, а функция `execlp` извлекает аргументы, пока не наткнётся на нулевой указатель, но и то и другое — лишь частные случаи.

вход три четырёхбайтных параметра (назовём их a1, a2 и a3), её вызов будет выглядеть примерно так:

```
push dword a3  ; заносим в стек параметры
push dword a2
push dword a1
call proc1     ; вызываем подпрограмму
add esp, 12    ; убираем параметры из стека
```

В случае же использования соглашений языка Паскаль последняя команда (add) оказывается не нужна, обо всём позаботится вызываемый. Процессор i386 даже имеет для этого специальную форму команды ret с одним операндом (выше в примерах мы использовали ret без операндов). Этот операнд, который может быть только непосредственным и всегда имеет длину два байта («слово»), задаёт количество памяти (в байтах), занятой параметрами функции. Например, процедуру, принимающую три четырёхбайтных параметра, комиплятор Паскаля закончит командой

```
ret 12
```

Эта команда, как и обычная команда ret, извлечёт из стека адрес возврата и передаст по нему управление, но кроме этого (одновременно с этим) увеличит значение ESP на заданное число (в данном случае 12), избавляя, таким образом, вызвавшего от обязанности по очистке стека.

§ 2.6.8. Локальные метки

Прежде чем мы приведём пример подпрограммы, выполняющей рекурсивный вызов, необходимо рассмотреть ещё одно важное средство, предоставляемое ассемблером NASM — ***локальные метки***.

Суть и основное достоинство подпрограмм состоит в их *обособленности*. Иначе говоря, в процессе написания одной подпрограммы мы обычно не помним, как устроены внутри себя другие подпрограммы и воспринимаем все подпрограммы, кроме одной (той, что пишется прямо сейчас) в виде этакой одной большой команды. Это позволяет не держать в голове лишних деталей и сосредоточиться на реализации конкретного фрагмента программы, а по окончании такой реализации выкинуть её детали из головы и перейти к другому фрагменту.

Проблема состоит в том, что в теле любой сколь бы то ни было сложной подпрограммы нам обязательно понадобятся метки, и нужно

сделать так, чтобы при выборе имён для таких меток нам не нужно было вспоминать, есть ли уже где-нибудь (в другой подпрограмме) метка с таким же именем.

Ассемблер NASM для этого предусматривает специальные *локальные* метки. Синтаксически эти метки отличаются от обычных тем, что начинаются с точки. Ассемблер локализует такие метки во фрагменте программы, ограниченном с обеих сторон обычными (нелокальными) метками. Иначе говоря, локальную метку ассемблер рассматривает не саму по себе, а как нечто подчинённое последней (ближайшей сверху) нелокальной метке. Например, в следующем фрагменте:

```
first_proc:
        ; ... ...
.cycle:
        ; ... ...
second_proc:
        ; ... ...
.cycle:
        ; ... ...
third_proc:
```

первая метка `.cycle` подчинена метке `first_proc`, а вторая — метке `second_proc`, так что между собой они не конфликтуют. Если метка `.cycle` встретится в параметрах той или иной команды между метками `first_proc` и `second_proc`, ассемблер будет знать, что имеется в виду именно первая из меток `.cycle`, если она встретится после `second_proc`, но перед `third_proc` — то задействуется вторая, тогда как появление метки `.cycle` до `first_proc` или после `third_proc` будет рассматриваться как ошибка.

На самом деле, ассемблер достигает такого эффекта за счёт ≪не очень честного≫ приёма — видя метку, имя которой начинается с точки, он просто добавляет к ней спереди имя последней встречавшейся ему метки без точки. Таким образом, в примере выше речь идёт не о двух одинаковых метках `.cycle`, а о двух *разных* метках `first_proc.cycle` и `second_proc.cycle`. Полезно помнить об этом и не применять в программе в явном виде метки, содержащие точку, несмотря на то, что ассемблер это допускает.

Таким образом, если каждую подпрограмму начинать с обычной метки, а внутри подпрограммы использовать только локальные метки, то в разных подпрограммах мы можем использовать локальные метки с одинаковыми именами, и ассемблер в них не запутается.

§ 2.6.9. Пример

Приведём пример подпрограммы, использующей рекурсию. Одна из простейших классических задач, решаемых рекурсивно — это сопоставление строки с образцом, её мы и используем в примере.

Для начала уточним задачу. Даны две строки символов, длина которых заранее неизвестна, но известно, что каждая из них ограничена нулевым байтом. Первую строку мы рассматриваем как *сопоставляемую*, вторую воспринимаем как *образец*. В образце символ `'?'` может сопоставляться с произвольным символом, символ `'*'` — с произвольной *подцепочкой символов* (возможно даже пустой), остальные символы обозначают сами себя и только сами с собой сопоставляются. Так, образцу `'abc'` соответствует только строка `'abc'`; образцу `'a?c'` соответствует любая строка из трёх символов, начинающаяся на `'a'` и заканчивающаяся на `'c'` (символ в середине может быть любым). Наконец, образцу `'a*'` соответствует любая строка, начинающаяся на `'a'`, ну а образцу `'*a*'` соответствует любая строка, содержащая букву `'a'` в любом месте. Необходимо определить, соответствует ли заданная строка заданному образцу, и вернуть результат `0`, если не соответствует, и результат `1`, если соответствует.

Алгоритм такого сопоставления, если при этом можно использовать рекурсию, окажется достаточно простым. На каждом шаге мы рассматриваем *оставшуюся часть* строки и образца; сначала эти оставшиеся части совпадают со строкой и образцом, затем, по мере продвижения алгоритма, от них отбрасываются символы, стоящие в начале, и мы предполагаем, что для уже отброшенных символов сопоставление прошло успешно.

Первое, что нужно сделать в начале каждого шага — это проверить, не кончился ли у нас образец. Если он кончился, то результат зависит от того, кончилась ли при этом и строка тоже. Если кончилась, то мы возвращаем единицу (истину), если не кончилась — возвращаем ноль (ложь); действительно, с пустым образцом можно сопоставить только пустую строку.

Если образец ещё не кончился, проверяем, не находится ли в начале него (то есть в первом символе остатка образца) символ `'*'`. Если нет, то всё просто: мы производим сопоставление первых символов строки и образца; если первый символ образца не является символом `'?'` и не равен первому символу строки, то алгоритм на этом завершается и мы возвращаем ложь, в противном случае считаем, что очередные символы образца и строки успешно сопоставлены, отбрасываем их (то есть

укорачиваем остатки обеих строк спереди) и возвращаемся к началу алгоритма.

Самое интересное происходит, если на очередном шаге первый символ образца оказался символом `'*'`. В этом случае нам нужно последовательно перебрать возможности сопоставления этой «звёздочки» с пустой подцепочкой строки, с одним символом строки, с двумя символами и т. д., пока не кончится сама строка. Делаем мы это следующим образом. Заводим целочисленную переменную `I`, которая будет у нас обозначать текущий рассматриваемый вариант. Присваиваем этой переменной ноль (начинаем рассмотрение с пустой цепочки). Теперь для каждой рассматриваемой альтернативы отбрасываем от образца один символ (звёздочку), а от строки — столько символов, какое сейчас число в переменной `I`. Полученные остатки *пытаемся сопоставить, используя для этого вызов той самой подпрограммы, которую мы сейчас пишем*, то есть производим рекурсивный вызов «самих себя». Если результат вызова — истина, то мы на этом завершаемся, тоже вернув истину. Если же результат — ложь, то мы проверяем, можно ли ещё увеличивать переменную `I` (не вылетим ли мы при этом за пределы сопоставляемой строки). Если увеличиваться уже некуда, завершаем работу, вернув ложь. В противном случае возвращаемся к началу цикла и рассматриваем следующее возможное значение `I`.

Для читателей, знакомых с языком программирования Си, отметим, что на этом языке вышеописанный алгоритм может быть реализован следующей функцией:

```
int match(const char *str, const char *pat)
{
    int i;
    for(;; str++, pat++) {
        switch(*pat) {
            case 0:
                return *str == 0;
            case '*':
                for(i=0; ; i++) {
                    if(match(str+i, pat+1)) return 1;
                    if(!str[i]) return 0;
                }
            case '?':
                if(!*str) return 0;
                break;
            default:
                if(*str != *pat) return 0;
```

```
        }
    }
}
```

К сожалению, привести аналогичный пример для языка Паскаль невозможно ввиду отсутствия в этом языке арифметики указателей.

Реализуем мы этот алгоритм в виде подпрограммы, которую назовём `match`. Подпрограмма будет предполагать, что ей передано два параметра — адрес строки (`[ebp+8]`) и адрес образца (`[ebp+12]`); сама подпрограмма будет использовать одну четырёхбайтную переменную для хранения `I`; под неё будет выделяться место в стековом фрейме и она будет, соответственно, располагаться по адресу `[ebp-4]`.

«Отбрасывание» символов из начала строк мы будем производить простым увеличением рассматриваемого адреса строки: действительно, если по адресу `string` находится строка, мы можем считать, что по адресу `string+1` находится та же строка, кроме первой буквы.

Необходимо обратить внимание, что, поскольку подпрограмма будет рекурсивно вызывать сама себя, хранить в регистрах мы почти ничего не сможем. Все данные мы будем хранить и модифицировать в памяти, а регистры будем использовать только для результатов промежуточных вычислений.

Возвращать результат наша подпрограмма будет через регистр `EAX`.

```
match:                          ; НАЧАЛО ПРОЦЕДУРЫ
        push ebp                ; организуем стековый фрейм
        mov ebp, esp
        sub esp, 4              ; выделяем 4 байта под I
.again:                         ; начало главного цикла
        mov esi, [ebp+8]        ; загружаем адреса строки
        mov edi, [ebp+12]       ;   и образца в ESI и EDI
        cmp byte [edi], 0       ; образец кончился?
        jne .not_end            ; прыгаем, если нет
        cmp byte [esi], 0       ; да; а кончилась ли строка?
        jne near .false         ; если нет - возвратим ложь
        jmp .true               ; иначе возвратим истину
.not_end:                       ; образец не кончился
        cmp byte [edi], '*'     ; не звёздочка ли в его начале?
        jne .not_star           ; если нет -- прыгаем
        mov dword [ebp-4], 0    ; да, звёздочка. Начинаем цикл
.star_loop:                     ;    с нулевого значения I
        mov esi, [ebp+8]        ; esi и edi могли быть
```

```
        mov edi, [ebp+12]    ;    испорчены, загружаем снова
        add esi, [ebp-4]     ; отбрасываем I символов строки
        inc edi              ;    и один символ образца
        push edi             ; подготовка к рекурсивному
        push esi             ;    вызову
        call match           ; вызов
        add esp, 8           ; очистка стека от параметров
        test eax, eax        ; результат -- истина? (<>0)
        jnz .true            ; если да, вернём истину
        mov esi, [ebp+8]     ; если нет, проверим,
        add esi, [ebp-4]     ;    не кончилась ли строка
        cmp byte [esi], 0    ;    (можно ли увеличивать I)
        je .false            ; если больше некуда, ложь
        inc dword [ebp-4]    ; иначе увеличиваем I
        jmp .star_loop       ;    и продолжаем цикл
.not_star:                   ; не звёздочка
        mov bl, [edi]        ; первый символ образца в BL
        cmp bl, '?'          ; это символ '?'
        je .quest            ; если да, можно не проверять
        cmp bl, [esi]        ; иначе сравниваем с символом
        jne .false           ;    строки и возвр. ложь если
.quest:                      ;    они оказались не равны
        inc dword [ebp+8]    ; теперь отбрасываем по одному
        inc dword [ebp+12]   ;    символу от образца и строки
        jmp .again           ; и повторяем большой цикл
.true:                       ; сюда мы прыгали, чтобы вернуть
        mov eax, 1           ;    истину, её и возвращаем
        jmp .quit            ;    и прыгаем в конец
.false:                      ; сюда мы прыгали, чтобы вернуть
        xor eax, eax         ;    ложь, её и возвращаем
.quit:                       ; сюда прыгали, чтобы закончить
        mov esp, ebp         ; убираем стековый фрейм
        pop ebp              ;
        ret                  ; возвращаем управление
                             ; КОНЕЦ ПРОЦЕДУРЫ
```

Если, например, ваша строка располагается в памяти, помеченной меткой `string`, а образец — в памяти, помеченной меткой `pattern`, то вызов подпрограммы `match` будет выглядеть вот так:

```
        push dword pattern
```

```
        push dword string
        call match
        add esp, 8
```

После этого результат сопоставления (0 или 1) окажется в регистре EAX.

Обратите внимание, что в начале подпрограммы при попытке перейти на метку .false мы были вынуждены явно указать, что переход является «ближним» (near). Дело в том, что метка .false оказалась чуть дальше от команды перехода, чем это допустимо для «короткого» перехода. См. обсуждение на стр. 56.

§ 2.7. Строковые операции

Для упрощения выполнения действий над массивами (непрерывными областями памяти) процессор i386 вводит несколько команд, объединяемых в категорию ***строковых операций***. Именно эти команды используют регистры ESI и EDI в их особой роли, обсуждавшейся на стр. 33.

Общая идея строковых команд состоит в том, что чтение из памяти выполняется по адресу из регистра ESI, запись в память — по адресу из регистра EDI, а затем эти регистры увеличиваются (или уменьшаются) в зависимости от команды на 1, 2 или 4. Некоторые команды производят чтение в регистр или запись в память из регистра; в этом случае используется регистр «аккумулятор» соответствующего размера, то есть регистр AL, AX или EAX. Строковые команды не имеют операндов, всегда используя одни и те же регистры.

«Направление» изменения адресов определяется флагом DF (напомним, его имя означает «direction flag», т. е. «флаг направления»). Если этот флаг сброшен, адреса увеличиваются (то есть строковая операция выполняется слева направо), если флаг установлен — адреса уменьшаются (соотв., работаем справа налево). Установить DF можно командой std (**s**e**t** **d**irection), а сбросить — командой cld (**cl**ear **d**irection).

Самые простые из строковых команд — команды stosb, stosw и stosd, которые записывают в память по адресу [edi], соответственно, байт, слово или двойное слово из регистра AL, AX или EAX, после чего увеличивают или уменьшают (в зависимости от значения DF) регистр EDI на 1, 2 или 4. Например, если у нас есть массив

```
buf     resb 1024
```

и нам нужно заполнить его нулями, мы можем применить следующий код:

```
        xor al, al    ; обнуляем al
        mov edi, buf  ; адрес начала массива
        mov ecx, 1024 ; длина массива
        cld           ; работаем в прямом направлении
lp:     stosb         ; al -> [edi], увел. edi
        loop lp
```

Эти и другие строковые команды удобно использовать с ***префиксом*** `rep`. Команда, снабженная таким префиксом, будет выполнена столько раз, какое число было в регистре `ECX` (кроме команды `stosw`: если её снабдить префиксом, то будет использоваться регистр `CX`; это обусловлено историческими причинами). С помощью префикса `rep` мы можем переписать вышеприведённый пример без использования метки:

```
        xor al, al
        mov edi, buf
        mov ecx, 1024
        cld
        rep stosb
```

Команды `lodsb`, `lodsw` и `lodsd`, наоборот, считывают байт, слово или двойное слово из памяти по адресу, находящемуся в регистре `ESI`, и помещают прочитанное в регистр `AL`, `AX` или `EAX`, после чего увеличивают или уменьшают значение регистра `ESI` на 1, 2 или 4.

Использование этих команд с префиксом `rep` обычно бессмысленно, поскольку мы не сможем между последовательными исполнениями строковой команды вставить какие-то ещё действия, обрабатывающие значение, прочитанное и помещённое в регистр. Однако использование команд серии `lods` без префикса может оказаться весьма полезным. Пусть, например, у нас есть массив четырёхбайтных чисел

```
array   resd 256
```

и нам необходимо сосчитать сумму его элементов. Это можно сделать следующим образом:

```
        xor ebx, ebx   ; зануляем сумму
        mov esi, resd
        mov ecx, 256
        cld
lp:     lodsd
        add ebx, eax
        loop lp
```

Часто оказывается удобным сочетание команд серии `lods` с соответствующими командами `stos`. Пусть, например, нам нужно увеличить на единицу все элементы того же самого массива. Это можно сделать так:

```
        mov esi, resd
        mov edi, esi
        mov ecx, 256
        cld
lp:     lodsd
        inc eax
        stosd
        loop lp
```

Если же необходимо просто скопировать данные из одной области памяти в другую, очень удобны оказываются команды `movsb`, `movsw` и `movsd`. Эти команды копируют байт, слово или двойное слово из памяти по адресу `[esi]` в память по адресу `[edi]`, после чего увеличивают (или уменьшают) сразу оба регистра `ESI` и `EDI` (соответственно, на 1, 2 или 4). Например, если у нас есть два строковых массива

```
buf1    resb 1024
buf2    resb 1024
```

и нужно скопировать содержимое одного из них в другой, можно сделать это так:

```
        mov ecx, 1024
        mov esi, buf1
        mov edi, buf2
        cld
        rep movsb
```

Благодаря возможности изменять направление работы (с помощью `DF`), мы можем производить копирование *частично перекрывающихся* областей памяти. Пусть, например, в массиве `buf1` содержится строка `"This is a string"` и нам нужно перед словом `"string"` вставить слово `"long"`. Для этого сначала нужно скопировать область памяти, начиная с адреса `[buf1+10]`, на пять байт вперёд, чтобы освободить место для слова `"long"` и пробела. Ясно, что производить такое копирование мы можем только из конца в начало, иначе часть букв будет затёрта до того, как мы их скопируем. Таким образом, если слово `"long "` (вместе с пробелом) содержится в буфере `buf2`, то вставить его во фразу, находящуюся в `buf1`, мы можем так:

```
std
mov edi, buf1+17+5
mov esi, buf1+17
mov ecx, 8
rep movsb
mov esi, buf2+4
mov ecx, 5
rep movsb
```

Кроме перечисленных, процессор i386 реализует команды `cmpsb`, `cmpsw` и `cmpsd` (**co**m**p**are **s**tring), а также `scasb`, `scasw` и `scasd` (**sca**n **s**tring).

Команды серии `scas` сравнивают аккумулятор (соответственно, `AL`, `AX` или `EAX`) с байтом, словом или двойным словом по адресу `[edi]`, устанавливая соответствующие флаги подобно команде `cmp`, и увеличивают/уменьшают `EDI`

Команды серии `cmps` сравнивают байты, слова или двойные слова, находящиеся в памяти по адресам `[esi]` и `[edi]`, устанавливают флаги и увеличивают/уменьшают оба регистра.

Кроме префикса `rep`, можно воспользоваться также префиксами `repz` и `repnz` (также называемыми `repe` и `repne`), которые, кроме уменьшения и проверки регистра `ECX` (или `CX`, если команда двухбайтная) также проверяют значение флага `ZF` и продолжают работу, только если этот флаг установлен (`repz`/`repe`) или сброшен `repnz`/`repne`). Обычно эти префиксы используют как раз в сочетании с командами серий `scas` и `cmps`.

§ 2.8. Ещё несколько интересных команд

В завершение изучения системы команд процессора i386 рассмотрим ещё несколько команд.

Команды `cbw`, `cwd`, `cwde` и `cdq` предназначены для *увеличения разрядности знакового числа*; попросту говоря, они заполняют дополнительные разряды значением знакового бита исходного числа. Все эти четыре команды не имеют операндов и всегда работают с одними и теми же регистрами. Команда `cbw` расширяет число в регистре `AL` до всего регистра `AX`, т. е. заполняет разряды регистра `AH`. Команда `cwd` расширяет число в регистре `AX` до регистровой пары `DX:AX`, то есть заполняет разряды регистра `DX`. Команда `cwde` расширяет тот же регистр `AX` до регистра `EAX`, заполняя старшие 16 разрядов этого регистра. Наконец,

команда `cdq` расширяет `EAX` до регистровой пары `EDX:EAX`, заполняя разряды регистра `EDX`. Особенно актуальными эти команды оказываются в сочетании с командой целочисленного деления (`div`, см. § 2.3.4).

Команды `movsx` (**mov**e **s**igned e**x**tension) и `movzx` (**mov**e **z**ero e**x**tension) позволяют совместить копирование с увеличением разрядности. Обе команды имеют по два операнда, причём первый операнд обязан быть регистровым, а второй может быть регистром или памятью, и в любом случае длина первого операнда должна быть вдвое больше длины второго (то есть можно копировать из байта в слово или из слова в двойное слово). Недостающие разряды команда `movzx` заполняет нулями, а команда `movsx` — значением старшего бита исходного операнда.

Наконец, рассмотрение системы команд не может считаться законченным без команды `nop`. Она выполняет очень важное действие: *не делает ничего*. Само её название образовано от слов «No OPeration».

§ 2.9. Заключительные замечания

Конечно, мы не рассмотрели и десятой доли возможностей процессора i386, если же говорить о расширениях его возможностей, появившихся в более поздних процессорах (например, `MMX`-регистры), то доля изученного нами окажется ещё скромнее. Однако *писать программы на языке ассемблера* мы теперь можем, и это позволит нам получить опыт программирования в терминах машинных команд, что, как было сказано в предисловии, является необходимым условием качественного программирования *вообще на любом языке программирования*: нельзя создавать хорошие программы, не понимая, что на самом деле происходит.

Читатели, у которых возникнет желание изучить аппаратную платформу i386 более глубоко, могут обратиться к технической документации и справочникам, которые в более чем достаточном количестве представлены в сети Интернет. Хочется, однако, заранее предупредить всех, у кого возникнет такое желание, что процессор i386 (отчасти «благодаря» тяжелому наследию 8086) имеет одну из самых хаотичных и нелогичных систем команд в мире; особенно это становится заметно, как только мы покидаем уютный мир ограниченного режима и «плоской» модели памяти, в котором нас заботливо устроила операционная система, и встречаемся лицом к лицу с программированием дескрипторов сегментов, нелепыми прыжками между кольцами защиты и прочи-

ми «прелестями» платформы, с которыми приходится бороться создателям современных операционных систем.

Так что, если вас всерьёз заинтересовало низкоуровневое программирование, мы можем посоветовать поизучать другие архитектуры, например, процессоры SPARC. Впрочем, любопытство в любом случае не порок, и если вы готовы к определённым трудностям — то найдите любой справочник по i386 и изучайте на здоровье :-)

Содержание

Предисловие для преподавателей 3
Предисловие для студентов 6
Благодарности и посвящение 8

1. Введение **9**
§ 1.1. Машинный код и ассемблер 9
§ 1.2. Особенности программирования под управлением мультизадачных операционных систем 15
§ 1.3. История платформы i386 19
§ 1.4. Знакомимся с инструментом 21

2. Процессор i386 **31**
§ 2.1. Система регистров i386 31
§ 2.2. Память, регистры и команда `mov` 35
§ 2.2.1. Директивы для отведения памяти 35
§ 2.2.2. Команда `mov` 39
§ 2.2.3. Виды операндов 40
§ 2.2.4. Прямая и косвенная адресация 42
§ 2.2.5. Общий вид исполнительного адреса 44
§ 2.2.6. Размеры операндов и их допустимые комбинации . 45
§ 2.2.7. Команда `lea` 48
§ 2.3. Целочисленная арифметика 49
§ 2.3.1. Простые команды сложения и вычитания 49
§ 2.3.2. Сложение и вычитание с переносом 50
§ 2.3.3. Команды `inc`, `dec`, `neg` и `cmp` 51
§ 2.3.4. Целочисленное умножение и деление 52
§ 2.4. Условные и безусловные переходы 54
§ 2.4.1. Безусловный переход и виды переходов 54
§ 2.4.2. Простые условные переходы 56

§ 2.4.3. Переходы по результатам сравнений 57
§ 2.4.4. Условные переходы и регистр ECX; циклы 59
§ 2.5. Побитовые операции 61
§ 2.5.1. Логические операции 61
§ 2.5.2. Операции сдвига 62
§ 2.6. Стек, подпрограммы, рекурсия 64
§ 2.6.1. Понятие стека и его предназначение 64
§ 2.6.2. Организация стека в процессоре i386 65
§ 2.6.3. Дополнительные команды работы со стеком 67
§ 2.6.4. Подпрограммы: общие принципы 68
§ 2.6.5. Вызов подпрограмм и возврат из них 69
§ 2.6.6. Организация стековых фреймов 70
§ 2.6.7. Основные конвенции вызовов подпрограмм 74
§ 2.6.8. Локальные метки 76
§ 2.6.9. Пример 78
§ 2.7. Строковые операции 82
§ 2.8. Ещё несколько интересных команд 85
§ 2.9. Заключительные замечания 86

На официальном сайте А. В. Столярова
http://www.stolyarov.info
можно найти электронные версии этой и других книг автора.

Домашняя страница этой книги, расположенная по адресу http://www.stolyarov.info/books/asm_unix, содержит дополнительные материалы, в том числе тексты программ, упоминаемых в пособии.